
CHAPTER 3

BRANCH, CALL, AND TIME DELAY LOOP

OBJECTIVES

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

- >> Code AVR Assembly language instructions to create loops
- >> Code AVR Assembly language conditional branch instructions
- >> Explain conditions that determine each conditional branch instruction
- >> Code JMP (long jump) instructions for unconditional jumps
- >> Calculate target addresses for conditional branch instructions
- >> Code AVR subroutines
- >> Describe the stack and its use in subroutines
- >> Discuss pipelining in the AVR
- >> Discuss crystal frequency versus instruction cycle time in the AVR
- >> Code AVR programs to generate a time delay

In the sequence of instructions to be executed, it is often necessary to transfer program control to a different location. There are many instructions in AVR to achieve this. This chapter covers the control transfer instructions available in AVR Assembly language. In Section 3.1, we discuss instructions used for looping, as well as instructions for conditional and unconditional branches (jumps). In Section 3.2, we examine the stack and the CALL instruction. In Section 3.3, instruction pipelining of the AVR is examined. Instruction timing and time delay subroutines are also discussed in Section 3.3.

SECTION 3.1: BRANCH INSTRUCTIONS AND LOOPING

In this section we first discuss how to perform a looping action in AVR and then the branch (jump) instructions, both conditional and unconditional.

Looping in AVR

Repeating a sequence of instructions or an operation a certain number of times is called a *loop*. The loop is one of most widely used programming techniques. In the AVR, there are several ways to repeat an operation many times. One way is to repeat the operation over and over until it is finished, as shown below:

```
LDI R16,0          ;R16 = 0
LDI R17,3          ;R17 = 3
ADD R16,R17        ;add value 3 to R16 (R16 = 0x03)
ADD R16,R17        ;add value 3 to R16 (R16 = 0x06)
ADD R16,R17        ;add value 3 to R16 (R16 = 0x09)
ADD R16,R17        ;add value 3 to R16 (R16 = 0x0C)
ADD R16,R17        ;add value 3 to R16 (R16 = 0x0F)
ADD R16,R17        ;add value 3 to R16 (R16 = 0x12)
```

In the above program, we add 3 to R16 six times. That makes $6 \times 3 = 18 = 0x12$. One problem with the above program is that too much code space would be needed to increase the number of repetitions to 50 or 100. A much better way is to use a loop. Next, we describe the method to do a loop in AVR.

Using BRNE instruction for looping

The BRNE (branch if not equal) instruction uses the zero flag in the status register. The BRNE instruction is used as follows:

```
BACK: .....      ;start of the loop
        .....      ;body of the loop
        .....      ;body of the loop
DEC Rn   ;decrement Rn, Z = 1 if Rn = 0
BRNE BACK ;branch to BACK if Z = 0
```

In the last two instructions, the Rn (e.g., R16 or R17) is decremented; if it is not zero, it branches (jumps) back to the target address referred to by the label. Prior to the start of the loop, the Rn is loaded with the counter value for the number of repetitions. Notice that the BRNE instruction refers to the Z flag of the status register affected by the previous instruction, DEC. This is shown in Example 3-1.

In the program in Example 3-1, register R16 is used as a counter. The counter is first set to 10. In each iteration, the DEC instruction decrements the R16 and sets the flag bits accordingly. If R16 is not zero ($Z = 0$), it jumps to the target address associated with the label "AGAIN". This looping action continues until R16 becomes zero. After R16 becomes zero ($Z = 1$), it falls through the loop and executes the instruction immediately below it, in this case "OUT PORTB, R20". See Figure 3-1.

Example 3-1

Write a program to (a) clear R20, then (b) add 3 to R20 ten times, and (c) send the sum to PORTB. Use the zero flag and BRNE.

Solution:

```
;this program adds value 3 to the R20 ten times
.INCLUDE "M32DEF.INC"
    LDI R16, 10      ;R16 = 10 (decimal) for counter
    LDI R20, 0       ;R20 = 0
    LDI R21, 3       ;R21 = 3
AGAIN:ADD R20, R21    ;add 03 to R20 (R20 = sum)
    DEC R16          ;decrement R16 (counter)
    BRNE AGAIN       ;repeat until COUNT = 0
    OUT PORTB, R20    ;send sum to PORTB
```

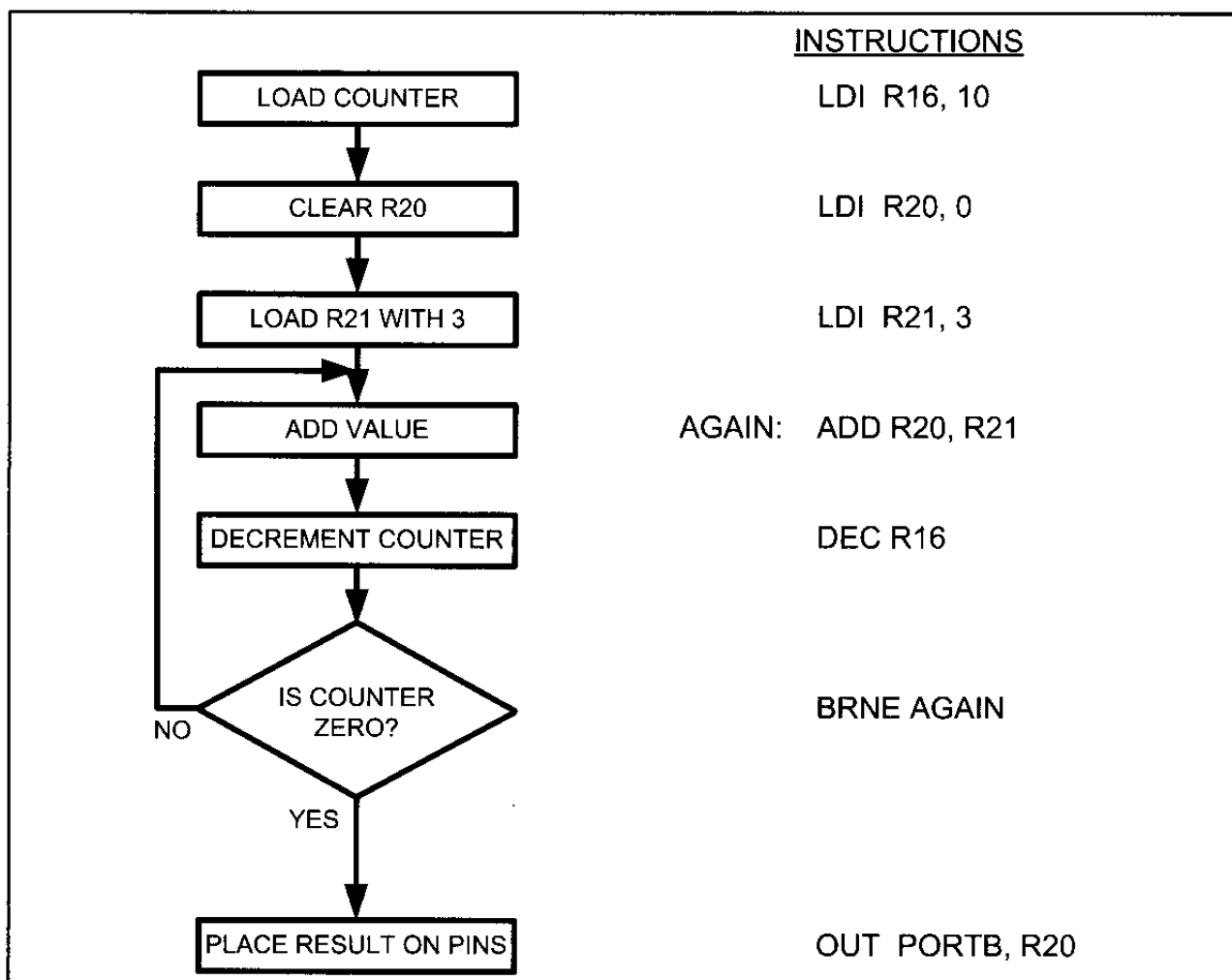


Figure 3-1. Flowchart for Example 3-1

Example 3-2

What is the maximum number of times that the loop in Example 3-1 can be repeated?

Solution:

Because location R16 is an 8-bit register, it can hold a maximum of 0xFF (255 decimal); therefore, the loop can be repeated a maximum of 255 times. Example 3-3 shows how to solve this limitation.

Loop inside a loop

As shown in Example 3-2, the maximum count is 255. What happens if we want to repeat an action more times than 255? To do that, we use a loop inside a loop, which is called a *nested loop*. In a nested loop, we use two registers to hold the count. See Example 3-3.

Example 3-3

Write a program to (a) load the PORTB register with the value 0x55, and (b) complement Port B 700 times.

Solution:

Because 700 is larger than 255 (the maximum capacity of any general purpose register), we use two registers to hold the count. The following code shows how to use R20 and R21 as a register for counters.

```
.INCLUDE "M32DEF.INC"
.ORG 0
    LDI R16, 0x55    ;R16 = 0x55
    OUT PORTB, R16   ;PORTB = 0x55
    LDI R20, 10      ;load 10 into R20 (outer loop count)
LOP_1:LDI R21, 70     ;load 70 into R21 (inner loop count)
LOP_2:COM R16         ;complement R16
    OUT PORTB, R16   ;load PORTB SFR with the complemented value
    DEC R21          ;dec R21 (inner loop)
    BRNE LOP_2       ;repeat it 70 times
    DEC R20          ;dec R20 (outer loop)
    BRNE LOP_1       ;repeat it 10 times
```

In this program, R21 is used to keep the inner loop count. In the instruction "BRNE LOP_2", whenever R21 becomes 0 it falls through and "DEC R20" is executed. The next instructions force the CPU to load the inner count with 70 if R20 is not zero, and the inner loop starts again. This process will continue until R20 becomes zero and the outer loop is finished.

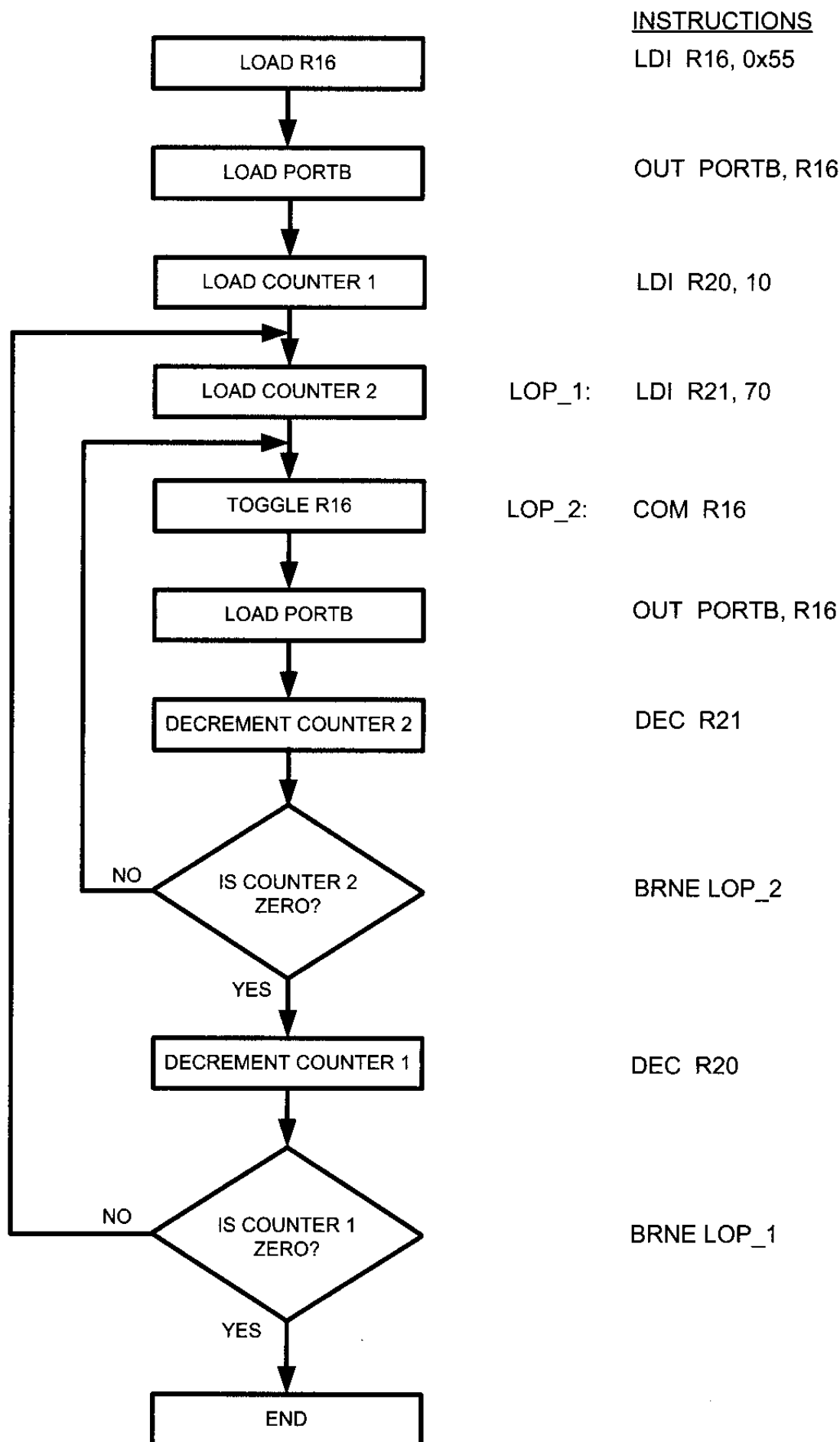


Figure 3-2. Flowchart for Example 3-3

Looping 100,000 times

Because two registers give us a maximum value of 65,025 ($255 \times 255 = 65,025$), we can use three registers to get up to more than 16 million (2^{24}) iterations. The following code repeats an action 100,000 times:

```
LDI    R16, 0x55
OUT     PORTB, R16
LDI     R23, 10
LOP_3: LDI    R22, 100
LOP_2: LDI    R21, 100
LOP_1: COM    R16
      DEC     R21
      BRNE    LOP_1
      DEC     R22
      BRNE    LOP_2
      DEC     R23
      BRNE    LOP_3
```

Other conditional jumps

In Table 3-1 you see some of the conditional branches for the AVR. More details of each instruction are provided in Appendix A. In Table 3-1 notice that the instructions, such as BREQ (Branch if Z = 1) and BRLO (Branch if C = 1), jump only if a certain condition is met. Next, we examine some conditional branch instructions with examples.

BREQ (branch if equal, branch if Z = 1)

In this instruction, the Z flag is checked. If it is high, the CPU jumps to the target address. For example, look at the following code.

```
OVER: IN     R20, PINB    ;read PINB and put it in R20
      TST    R20          ;set the flags according to R20
      BREQ   OVER         ;jump if R20 is zero
```

In this program, if PINB is zero, the CPU jumps to the label OVER. It stays in the loop until PINB has a value other than zero. Notice that the TST instruction can be used to examine a register and set the flags according to the contents of the register without performing an arithmetic instruction such as decrement.

When the TST instruction executes, if the register contains the zero value, the zero flag is set; otherwise, it is cleared. It also sets the N flag high if the D7 bit of the register is high, otherwise N = 0. See Example 3-4.

BRSR (branch if same or higher, branch if C = 0)

In this instruction, the carry flag bit in the Status register is used to make the decision whether to jump. In executing “BRSR label”, the processor looks at the carry flag to see if it is raised (C = 1). If it is not, the CPU starts to fetch and execute instructions from the address of the label. If C = 1, it will not branch but

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

Instruction	Action
BRLO	Branch if C = 1
BRSR	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

Example 3-4

Write a program to determine if RAM location 0x200 contains the value 0. If so, put 0x55 into it.

Solution:

```
.EQU MYLOC=0x200
LDS R30, MYLOC
TST R30 ;set the flag
; (Z=1 if R30 has zero value)
BRNE NEXT ;branch if R30 is not zero (Z=0)
LDI R30, 0x55 ;put 0x55 if R30 has zero value
STS MYLOC, R30 ;and store a copy to loc $200
NEXT: ...
```

will execute the next instruction below BRSH. Study Example 3-5 to see how BRSH is used to add numbers together when the sum is higher than \$FF. Note that there is also a “BRLO label” instruction. In the BRLO instruction, if C = 1 the CPU jumps to the target address. We will give more examples of these instructions in the context of some applications in Chapter 5.

The other conditional branch instructions in Table 3-1 are discussed in Chapter 5 when arithmetic operations with signed numbers are discussed.

Example 3-5

Find the sum of the values 0x79, 0xF5, and 0xE2. Put the sum into R20 (low byte) and R21 (high byte).

Solution:

```
.INCLUDE "M32DEF.INC"
.ORG 0
LDI R21, 0 ;clear high byte (R21 = 0)
LDI R20, 0 ;clear low byte (R20 = 0)
LDI R16, 0x79
ADD R20, R16 ;R20 = 0 + 0x79 = 0x79, C = 0
BRSH N_1 ;if C = 0, add next number
INC R21 ;C = 1, increment (now high byte = 0)
N_1: LDI R16, 0xF5
ADD R20, R16 ;R20 = 0x79 + 0xF5 = 0x6E and C = 1
BRSH N_2 ;branch if C = 0
INC R21 ;C = 1, increment (now high byte = 1)
N_2: LDI R16, 0xE2
ADD R20, R16 ;R20 = 0x6E + 0xE2 = 0x50 and C = 1
BRSH OVER ;branch if C = 0
INC R21 ;C = 1, increment (now high byte = 2)
OVER: ;now low byte = 0x50, and high byte = 02
```

	R21 (high byte)	R20 (low byte)
At first	\$0	\$00
Before LDI R16,0xF5	\$0	\$79
Before LDI R16,0xE2	\$1	\$6E
At the end	\$2	\$50

All conditional branches are short jumps

It must be noted that all conditional jumps are short jumps, meaning that the address of the target must be within 64 bytes of the program counter (PC). This concept is discussed next.

Example 3-6

Using the following list file, verify the jump forward address calculation.

```
LINE    ADDRESS    Machine Mnemonic Operand
3:      +00000000:  E050    LDI      R21, 0      ;clear high byte (R21 = 0)
4:      +00000001:  E040    LDI      R20, 0      ;clear low byte (R20 = 0)
5:      +00000002:  E709    LDI      R16, 0x79
6:      +00000003:  0F40    ADD      R20, R16      ;R20 = 0 + 0x79 = 0x79, C = 0
7:      +00000004:  F408    BRSH     N_1        ;if C = 0, add next number
8:      +00000005:  9543    INC      R21          ;C = 1, increment (now high byte = 0)
9:      +00000006:  EF05    N_1:LDI     R16, 0xF5
10:     +00000007:  0F40    ADD      R20, R16      ;R20 = $79 + $F5 = 6E and C = 1
11:     +00000008:  F408    BRSH     N_2        ;branch if C = 0
12:     +00000009:  9553    INC      R21          ;C = 1, increment (now high byte = 1)
@00000000A: n_2
13:     +0000000A:  EE02    N_2:LDI     R16, 0xE2
14:     +0000000B:  0F40    ADD      R20, R16      ;R20 = $6E + $E2 = $50 and C = 1
15:     +0000000C:  F408    BRSH     OVER       ;branch if C = 0
16:     +0000000D:  9553    INC      R21          ;C = 1, increment (now high byte = 2)
@00000000E: over
```

Solution:

First notice that the BRSH instruction jumps forward. The target address for a forward jump is calculated by adding the PC of the following instruction to the second byte of the branch instruction. Recall that each instruction takes 2 bytes. In line 7 the instruction “BRSH N_1” has the machine code of F408. To distinguish the operand and opcode parts, we should compare the machine code with the BRSH instruction format. In the following, you see the format of the BRSH instruction. The bits marked by k are

1111	01kk	kkkk	k000
------	------	------	------

the operand bits while the remainder are the opcode bits. In this example the machine code's equivalent in binary is 1111 0100 0000 1000. If we compare it with the BRSH format, we see that the operand is 000001 and the opcode is 111101000. The 01 is the relative address, relative to the address of the next instruction INC R21, which is 000005. By adding 000001 to 000005, the target address of the label N_1, which is 000006, is generated. Likewise for line 11, the “BRSH N_2” instruction, and line 000015, the “BRSH OVER” instruction jumps forward because the relative value is positive.

All the conditional jump instructions, whose mnemonics begin with BR (e.g., BRNE and BRIE), have the same instruction format, and the opcode changes from instruction to instruction. So, we can calculate the short branch address for any of them, as we did in this example.

Calculating the short branch address

All conditional branches such as BRSH, BREQ, and BRNE are short branches due to the fact that they are all 2-byte instructions. In these instructions the opcode is 9 bits and the relative address is 7 bits. The target address is relative to the value of the program counter. If the relative address is positive, the jump is forward. If the relative address is negative, then the jump is backwards. The relative address can be a value from -64 to $+63$. To calculate the target address, the relative address is added to the PC of the next instruction (target address = relative address + PC). See Example 3-6. We do the same thing for the backward branch, although the second byte is negative. That is, we add it to the PC value of the next instruction. See Example 3-7.

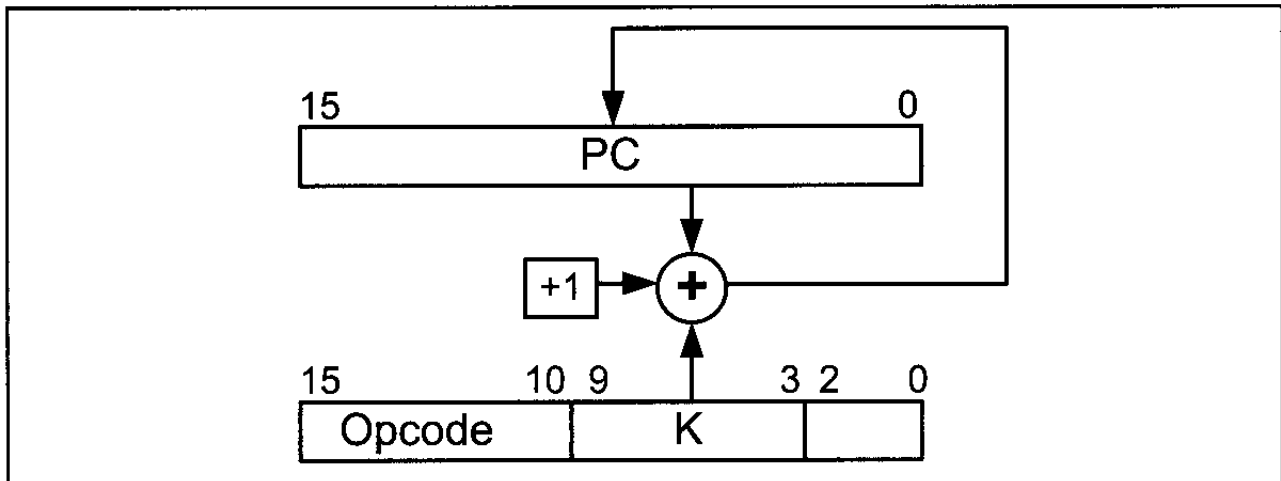


Figure 3-3. Calculating the Target Address in Conditional Branch Instructions

Example 3-7

Verify the calculation of backward jumps for the listing of Example 3-1, shown below.

Solution:

LINE	ADDRESS	Machine	Mnemonic	Operand
3:	+00000000:	E00A	LDI	R16, 10 ;R16 = 10 (decimal) for counter
4:	+00000001:	E040	LDI	R20, 0 ;R20 = 0
5:	+00000002:	E053	LDI	R21, 3 ;R21 = 3
6:	+00000003:	0F45	AGAIN:ADD	R20, R21 ;add 03 to R20 (R20 = sum)
7:	+00000004:	950A	DEC	R16 ;decrement R16 (counter)
8:	+00000005:	F7E9	BRNE	AGAIN ;repeat until COUNT = 0
9:	+00000006:	BB48	OUT	PORTB,R20 ;send sum to PORTB SFR

In the program list, “BRNE AGAIN” has machine code F7E9. To specify the operand and opcode, we compare the instruction with the branch instruction format, which you saw in the previous example. Because the binary equivalent of the instruction is 1111 0111 1110 1001, the opcode is 111101001 and the operand (relative address) is 1111101. The 1111101 gives us -3 , which means the displacement is -3 . When the relative address of -3 is added to 000006, the address of the instruction below, we have $-3 + 06 = 03$ (the carry is dropped). Notice that 000003 is the address of the label AGAIN.

1111101 is a negative number, which means it will branch backward. For further discussion of the addition of negative numbers, see Chapter 5.

You might ask why we add the relative address to the address of the next instruction. (Why don't we add the relative address to the address of the current instruction?) Before an instruction is executed, it should be fetched. So, the branch instructions are executed after they are fetched. The PC points to the instruction that should be fetched next. So, when the branch instructions are executed, the PC is pointing to the next instruction. That is why we add the relative address to the address of the next instruction. We will discuss the execution of instructions in the last section of this chapter.

Unconditional branch instruction

The unconditional branch is a jump in which control is transferred unconditionally to the target location. In the AVR there are three unconditional branches: JMP (jump), RJMP (relative jump), and IJMP (indirect jump). Deciding which one to use depends on the target address. Each instruction is explained next.

JMP (JMP is a long jump)

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

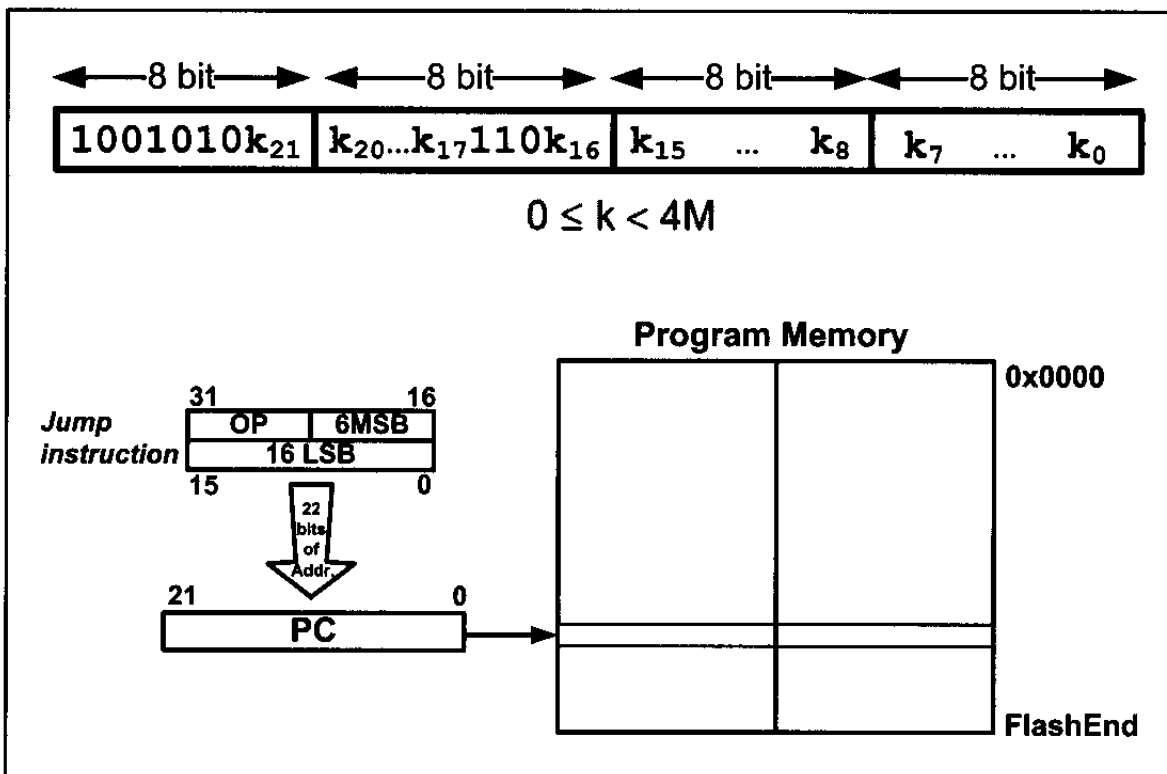


Figure 3-4. JMP Instruction

Remember that although the AVR can have ROM space of 8M bytes, not all AVR family members have that much on-chip program ROM. Some AVR family members have only 4K–32K of on-chip ROM for program space; consequently, every byte is precious. For this reason there is also an RJMP (relative jump)

instruction, which is a 2-byte instruction as opposed to the 4-byte JMP instruction. This can save some bytes of memory in many applications where ROM memory space is in short supply. RJMP is discussed next.

RJMP (relative jump)

In this 2-byte (16-bit) instruction, the first 4 bits are the opcode and the rest (lower 12 bits) is the relative address of the target location. The relative address range of 000 – \$FFF is divided into forward and backward jumps; that is, within –2048 to +2047 words of memory relative to the address of the current PC (program counter). If the jump is forward, then the relative address is positive. If the jump is backward, then the relative address is negative. In this regard, RJMP is like the conditional branch instructions except that 12 bits are used for the offset address instead of 7. This is shown in detail in Figure 3-5.

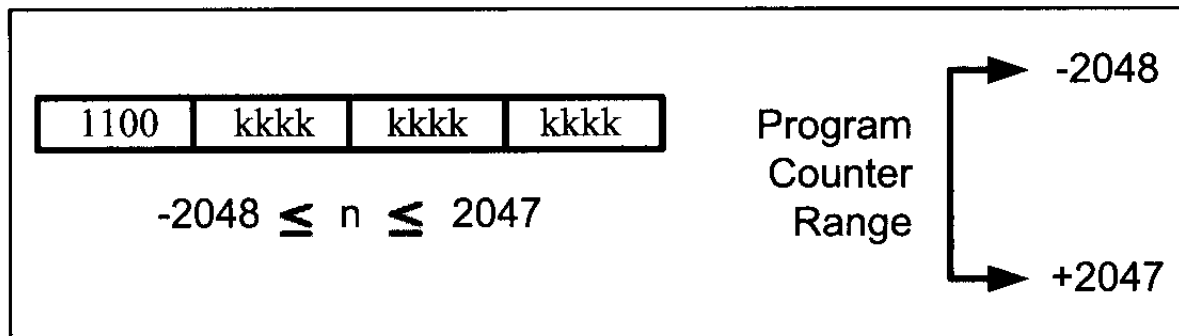


Figure 3-5. RJMP (Relative Jump) Instruction Address Range

Notice that this is a 2-byte instruction, and is preferred over the JMP because it takes less ROM space.

IJMP (indirect jump)

IJMP is a 2-byte instruction. When the instruction executes, the PC is loaded with the contents of the Z register, so it jumps to the address pointed to by the Z register. As you will see in Chapter 6, Z is a 2-byte register, so IJMP can jump within the lowest 64K words of the program memory.

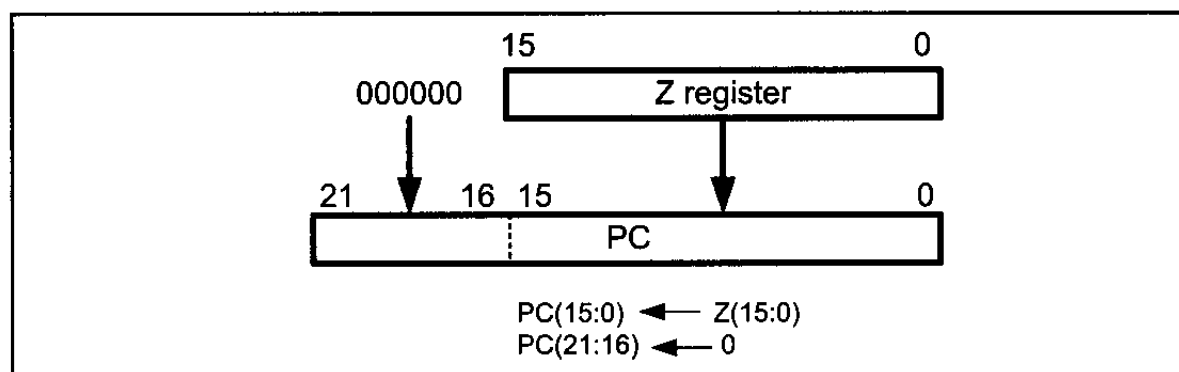


Figure 3-6. IJMP (Indirect Jump) Instruction Target Address

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

Review Questions

1. The mnemonic BRNE stands for _____.
2. True or false. "BRNE BACK" makes its decision based on the last instruction affecting the Z flag.
3. "BRNE HERE" is a ____ -byte instruction.
4. In "BREQ NEXT", which register's content is checked to see if it is zero?
5. JMP is a(n) ____ -byte instruction.

SECTION 3.2: CALL INSTRUCTIONS AND STACK

Another control transfer instruction is the CALL instruction, which is used to call a subroutine. Subroutines are often used to perform tasks that need to be performed frequently. This makes a program more structured in addition to saving memory space. In the AVR there are four instructions for the call subroutine: CALL (long call) RCALL (relative call), ICALL (indirect call to Z), and EICALL (extended indirect call to Z). The choice of which one to use depends on the target address. Each instruction is explained next.

CALL

In this 4-byte (32-bit) instruction, 10 bits are used for the opcode and the other 22 bits, k_{21} – k_0 , are used for the address of the target subroutine, just as in the JMP instruction. Therefore, CALL can be used to call subroutines located anywhere within the 4M address space of 000000–\$3FFFFFF for the AVR, as shown in Figure 3-7.

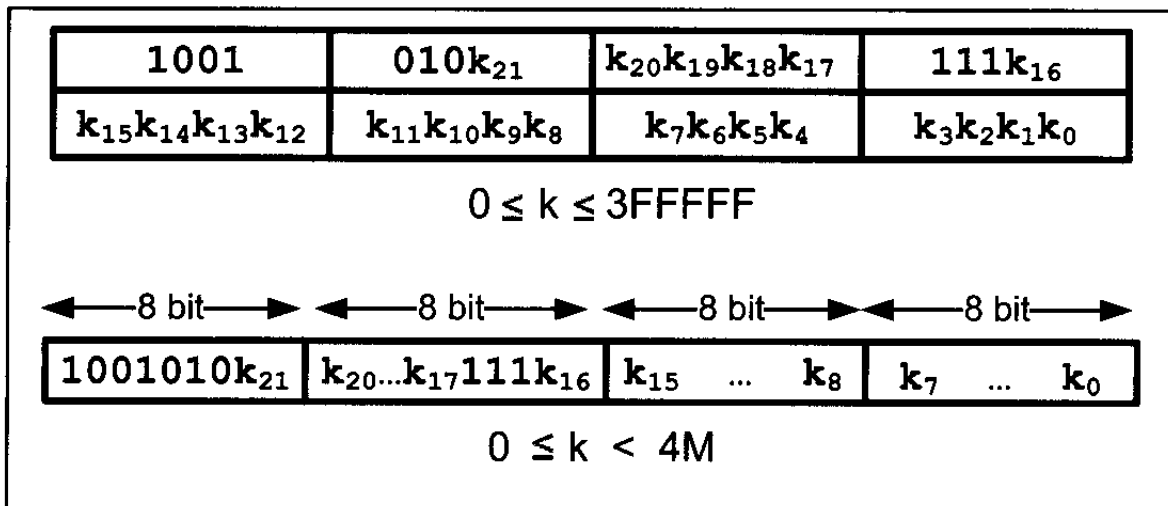


Figure 3-7. CALL Instruction Formation

To make sure that the AVR knows where to come back to after execution of the called subroutine, the microcontroller automatically saves on the stack the address of the instruction immediately below the CALL. When a subroutine is called, control is transferred to that subroutine, and the processor saves the PC (program counter) of the next instruction on the stack and begins to fetch instructions from the new location. After finishing execution of the subroutine, the RET instruction transfers control back to the caller. Every subroutine needs RET as the last instruction.

Stack and stack pointer in AVR

The stack is a section of RAM used by the CPU to store information temporarily. This information could be data or an address. The CPU needs this storage area because there are only a limited number of registers.

How stacks are accessed in the AVR

If the stack is a section of RAM, there must be a register inside the CPU to point to it. The register used to access the stack is called the SP (stack pointer) register.

In I/O memory space, there are two registers named SPL (the low byte of the SP) and SPH (the high byte of the SP). The SP is implemented as two registers. The SPH register presents the high byte of the SP while the SPL register presents the lower byte.

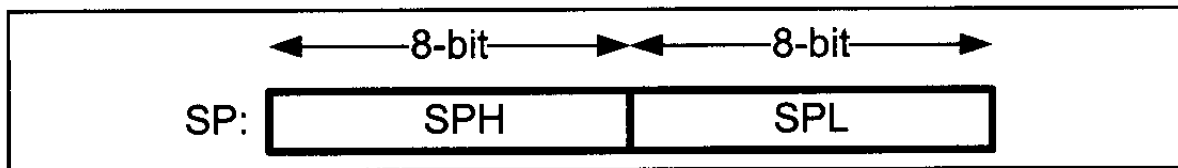


Figure 3-8. SP (Stack Pointer) in AVR

The stack pointer must be wide enough to address all the RAM. So, in the AVRs with more than 256 bytes of memory the SP is made of two 8-bit registers (SPL and SPH), while in the AVRs with less than 256 bytes the SP is made of only SPL, as an 8-bit register can address 256 bytes of memory.

The storing of CPU information such as the program counter on the stack is called a PUSH, and the loading of stack contents back into a CPU register is called a POP. In other words, a register is pushed onto the stack to save it and popped off the stack to retrieve it. The following describes each process.

Pushing onto the stack

The stack pointer (SP) points to the top of the stack (TOS). As we push data onto the stack, the data are saved where the SP points to, and the SP is decremented by one. Notice that this is the same as with many other microprocessors, notably x86 processors, in which the SP is decremented when data is pushed onto the stack.

To push a register onto stack we use the PUSH instruction.

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

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

```
PUSH R10    ;store R10 onto the stack, and decrement SP
```

Popping from the stack

Popping the contents of the stack back into a given register is the opposite process of pushing. When the POP instruction is executed, the SP is incremented and the top location of the stack is copied back to the register. That means the stack is LIFO (Last-In-First-Out) memory.

To retrieve a byte of data from stack we can use the POP instruction.

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

For example, the following instruction pops from the top of stack and copies to R10:

POP R16 ;increment SP, and then load the top of stack to R10

Example 3-8

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

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

```
.ORG 0
```

```
;initialize the SP to point to the last location of RAM (RAMEND)
```

```
LDI R16, HIGH(RAMEND) ;load SPH
```

```
OUT SPH, R16
```

```
LDI R16, LOW(RAMEND) ;load SPL
```

```
OUT SPL, R16
```

```
LDI R31, 0
```

```
LDI R20, 0x21
```

```
LDI R22, 0x66
```

```
PUSH R20
```

```
PUSH R22
```

```
LDI R20, 0
```

```
LDI R22, 0
```

```
POP R22
```

```
POP R31
```

Solution:

After the execution of	Contents of some of the registers				Stack
	R20	R22	R31	SP	
OUT SPL,R16	\$0	\$0	0	\$085F	
LDI R22, 0x66	\$21	\$66	0	\$085F	
PUSH R20	\$21	\$66	0	\$085E	
PUSH R22	\$21	\$66	0	\$085D	
LDI R22, 0	\$0	\$0	0	\$085D	
POP R22	\$0	\$66	0	\$085E	
POP R31	\$0	\$66	\$21	\$085F	

Initializing the stack pointer

When the AVR is powered up, the SP register contains the value 0, which is the address of R0. Therefore, we must initialize the SP at the beginning of the program so that it points to somewhere in the internal SRAM. In AVR, the stack grows from higher memory location to lower memory location (when we push onto the stack, the SP decrements). So, it is common to initialize the SP to the uppermost memory location.

Different AVRs have different amounts of RAM. In the AVR assembler `RAMEND` represents the address of the last RAM location. So, if we want to initialize the SP so that it points to the last memory location, we can simply load `RAMEND` into the SP. Notice that SP is made of two registers, `SPH` and `SPL`. So, we load the high byte of `RAMEND` into `SPH`, and the low byte of `RAMEND` into the `SPL`.

Example 3-8 shows how to initialize the SP and use the `PUSH` and `POP` instructions. In the example you can see how the stack changes when the `PUSH` and `POP` instructions are executed.

For more information about `RAMEND`, `LOW`, and `HIGH`, see Section 6-1.

CALL instruction and the role of the stack

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

For the AVRs whose program counter is not longer than 16 bits (e.g., ATmega128, ATmega32), the value of the program counter is broken into 2 bytes. The higher byte is pushed onto the stack first, and then the lower byte is pushed.

For the AVRs whose program counters are longer than 16 bits but shorter than 24 bits, the value of the program counter is broken up into 3 bytes. The highest byte is pushed first, then the middle byte is pushed, and finally the lowest byte is pushed. So, in both cases, the higher bytes are pushed first.

RET instruction and the role of the stack

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

To understand the role of the stack in call instruction and returning, examine the contents of the stack and stack pointer (see Examples 3-9 and Example 3-10). The following points should be noted for the program in Example 3-9:

1. Notice the `DELAY` subroutine. After the first “`CALL DELAY`” is executed, the address of the instruction right below it, “`LDI R16, 0xAA`”, is pushed onto

the stack, and the AVR starts to execute instructions at address 0x300.

2. In the DELAY subroutine, the counter R20 is set to 255 (R20 = 0xFF); therefore, the loop is repeated 255 times. When R20 becomes 0, control falls to the RET instruction, which pops the address from the top of the stack into the program counter and resumes executing the instructions after the CALL.

Example 3-9

Toggle all the bits of Port B by sending to it the values \$55 and \$AA continuously. Put a time delay between each issuing of data to Port B.

Solution:

```
.INCLUDE "M32DEF.INC"
.ORG 0
    LDI R16,HIGH(RAMEND)    ;load SPH
    OUT SPH,R16
    LDI R16,LOW(RAMEND)     ;load SPL
    OUT SPL,R16

BACK:
    LDI R16,0x55            ;load R16 with 0x55
    OUT PORTB,R16           ;send 55H to port B
    CALL DELAY              ;time delay
    LDI R16,0xAA            ;load R16 with 0xAA
    OUT PORTB,R16           ;send 0xAA to port B
    CALL DELAY              ;time delay
    RJMP BACK               ;keep doing this indefinitely
;----- this is the delay subroutine
.ORG 0x300                  ;put time delay at address 0x300
DELAY:
    LDI R20,0xFF            ;R20 = 255,the counter
AGAIN:
    NOP                     ;no operation wastes clock cycles
    NOP
    DEC R20
    BRNE AGAIN              ;repeat until R20 becomes 0
    RET                     ;return to caller
```

The amount of time delay in Example 3-9 depends on the frequency of the AVR. How to calculate the exact time will be explained in the last section of this chapter.

The upper limit of the stack

As mentioned earlier, we can define the stack anywhere in the general purpose memory. So, in the AVR the stack can be as big as its RAM. Note that we must not define the stack in the register memory, nor in the I/O memory. So, the SP must be set to point above 0x60.

In AVR, the stack is used for calls and interrupts. We must remember that upon calling a subroutine, the stack keeps track of where the CPU should return after completing the subroutine. For this reason, we must be very careful when manipulating the stack contents.

Example 3-10

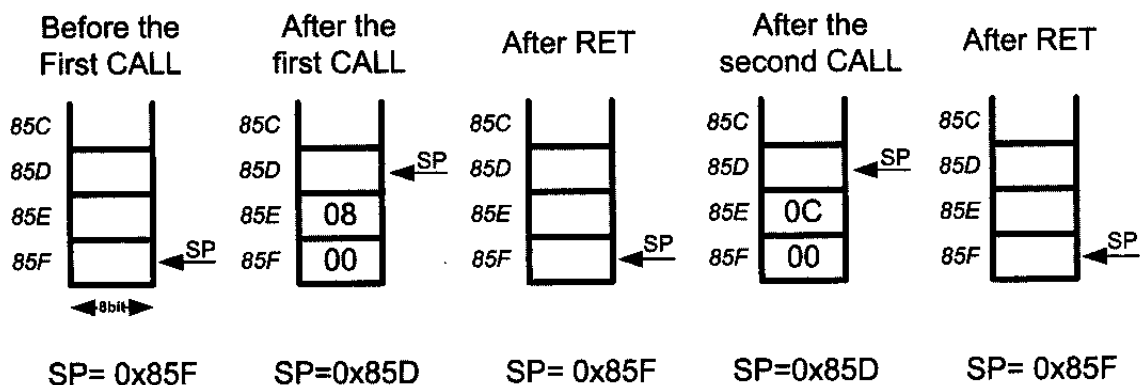
Analyze the stack for the CALL instructions in the following program.

```
.INCLUDE "M32DEF.INC"
.ORG 0
+00000000:    LDI R16,HIGH(RAMEND)    ;initialize SP
+00000001:    OUT SPH,R16
+00000002:    LDI R16,LOW(RAMEND)
+00000003:    OUT SPL,R16

BACK:
+00000004:    LDI R16,0x55           ;load R16 with 0x55
+00000005:    OUT PORTB,R16         ;send 55H to port B
+00000006:    CALL DELAY            ;time delay
+00000008:    LDI R16,0xAA           ;load R16 with 0xAA
+00000009:    OUT PORTB,R16         ;send 0xAA to port B
+0000000A:    CALL DELAY            ;time delay
+0000000C:    RJMP BACK              ;keep doing this indefinitely
;-----this is the delay subroutine
.ORG 0x300           ;put time delay at address 0x300
DELAY:
+00000300:    LDI R20,0xFF           ;R20 = 255, the counter
AGAIN:
+00000301:    NOP                    ;no operation wastes clock cycles
+00000302:    NOP
+00000303:    DEC R20
+00000304:    BRNE AGAIN             ;repeat until R20 becomes 0
+00000305:    RET                    ;return to caller
```

Solution:

When the first CALL is executed, the address of the instruction "LDI R16,0xAA" is saved (pushed) onto the stack. The last instruction of the called subroutine must be a RET instruction, which directs the CPU to pop the contents of the top location of the stack into the PC and resume executing at address 0008. The diagrams show the stack frame after the CALL and RET instructions.



Calling many subroutines from the main program

In Assembly language programming, it is common to have one main program and many subroutines that are called from the main program. See Figure 3-9. This allows you to make each subroutine into a separate module. Each module can be tested separately and then brought together with the main program. More importantly, in a large program the modules can be assigned to different programmers in order to shorten development time.

It needs to be emphasized that in using CALL, the target address of the subroutine can be anywhere within the 4M (word) memory space of the AVR. See Example 3-11. This is not the case for the RCALL instruction, which is explained next.

```
.INCLUDE "M32DEF.INC"    ;Modify for your chip

;MAIN program calling subroutines
        .ORG 0
MAIN:    CALL SUBR_1
        CALL SUBR_2
        CALL SUBR_3
        CALL SUBR_4
HERE:    RJMP HERE        ;stay here
;-----end of MAIN
;
SUBR_1:   ....
        ....
        RET
;-----end of subroutine 1
;
SUBR_2:   ....
        ....
        RET
;-----end of subroutine 2
;
SUBR_3:   ....
        ....
        RET
;-----end of subroutine 3
;
SUBR_4:   ....
        ....
        RET
;-----end of subroutine 4
```

Figure 3-9. AVR Assembly Main Program That Calls Subroutines

RCALL (relative call)

RCALL is a 2-byte instruction in contrast to CALL, which is 4 bytes. Because RCALL is a 2-byte instruction, and only 12 bits of the 2 bytes are used for the address, the target address of the subroutine must be within -2048 to +2047 words of memory relative to the address of the current PC.

Example 3-11

Write a program to count up from 00 to \$FF and send the count to Port B. Use one CALL subroutine for sending the data to Port B and another one for time delay. Put a time delay between each issuing of data to Port B.

Solution:

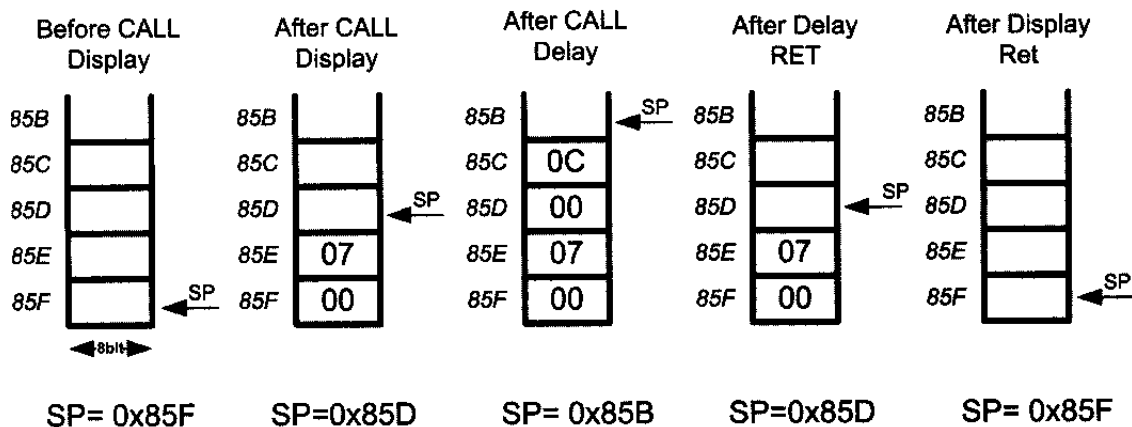
```
.INCLUDE "M32DEF.INC"

.DEF COUNT=R20
.ORG 0
+00000000: LDI R16,HIGH(RAMEND)
+00000001: OUT SPH,R16
+00000002: LDI R16,LOW(RAMEND)
+00000003: OUT SPL,R16 ;initialize stack pointer

+00000004: LDI COUNT,0 ;count = 0
BACK:
+00000005: CALL DISPLAY
+00000007: RJMP BACK

;-----Increment and put it in PORTB
DISPLAY:
+00000008: INC COUNT ;increment count
+00000009: OUT PORTB, COUNT ;send it to PORTB
+0000000A: CALL DELAY
+0000000C: RET ;return to caller

;-----This is the delay subroutine
.ORG 0x300 ;put time delay at address 0x300
DELAY:
+00000300: LDI R16,0xFF ;R16 = 255
+00000301: AGAIN:
+00000302: NOP
+00000303: NOP
+00000304: NOP
+00000305: DEC R16
+00000306: BRNE AGAIN ;repeat until R16 becomes 0
+00000307: RET ;return to caller
```



There is no difference between RCALL and CALL in terms of saving the program counter on the stack or the function of the RET instruction. The only difference is that the target address for CALL can be anywhere within the 4M address space of the AVR while the target address of RCALL must be within a 4K range.

In many variations of the AVR marketed by Atmel Corporation, on-chip ROM is as low as 4K. In such cases, the use of RCALL instead of CALL can save a number of bytes of program ROM space.

Of course, in addition to using compact instructions, we can program efficiently by having a detailed knowledge of all the instructions supported by a given microcontroller, and using them wisely. Look at Examples 3-12 and 3-13.

Example 3-12

Rewrite the main part of Example 3-9 as efficiently as you can.

Solution:

```
.INCLUDE "M32DEF.INC"
.ORG 0
    LDI R16,HIGH(RAMEND)
    OUT SPH,R16
    LDI R16,LOW(RAMEND)
    OUT SPL,R16           ;initialize stack pointer

    LDI R16,0x55          ;load R16 with 55H
BACK:
    COM R16               ;complement R16
    OUT PORTB,R16         ;load port B SFR
    RCALL DELAY           ;time delay
    RJMP BACK             ;keep doing this indefinitely

;-----this is the delay subroutine
DELAY:
    LDI R20,0xFF
AGAIN:
    NOP                  ;no operation wastes clock cycles
    NOP
    DEC R20
    BRNE AGAIN           ;repeat until R20 becomes 0
    RET
```

Example 3-13

A developer is using the AVR microcontroller chip for a product. This chip has only 4K of on-chip flash ROM. Which of the instructions, CALL or RCALL, is more useful in programming this chip?

Solution:

The RCALL instruction is more useful because it is a 2-byte instruction. It saves two bytes each time the call instruction is used. However, we must use CALL if the target address is beyond the 2K boundary.

ICALL (indirect call)

In this 2-byte (16-bit) instruction, the Z register specifies the target address. When the instruction is executed, the address of the next instruction is pushed into the stack (like CALL and RCALL) and the program counter is loaded with the contents of the Z register. So, the Z register should contain the address of a function when the ICALL instruction is executed. Because the Z register is 16 bits

wide, the ICALL instruction can call the subroutines that are within the lowest 64K words of the program memory. (The target address calculation in ICALL is the same as for the IJMP instruction.) See Figure 3-10.

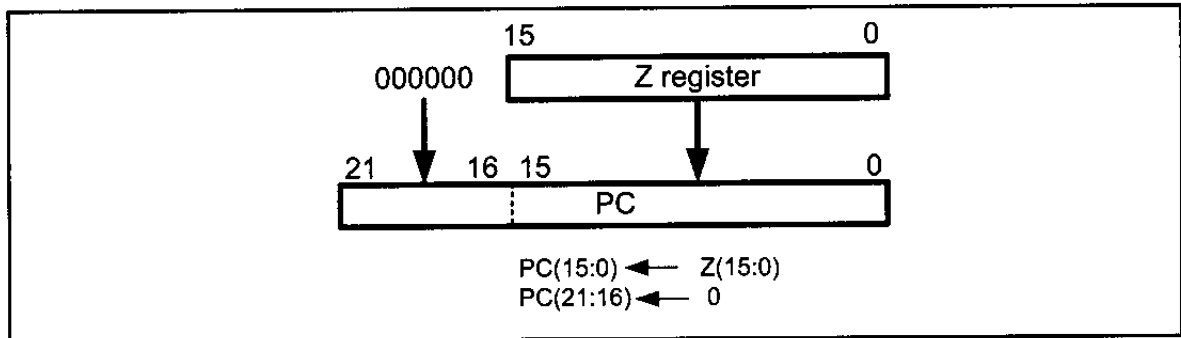


Figure 3-10. ICALL Instruction

In the AVRs with more than 64K words of program memory, the EICALL (extended indirect call) instruction is available. The EICALL loads the Z register into the lower 16 bits of the PC and the EIND register into the upper 6 bits of the PC. Notice that EIND is a part of I/O memory. See Figure 3-11.

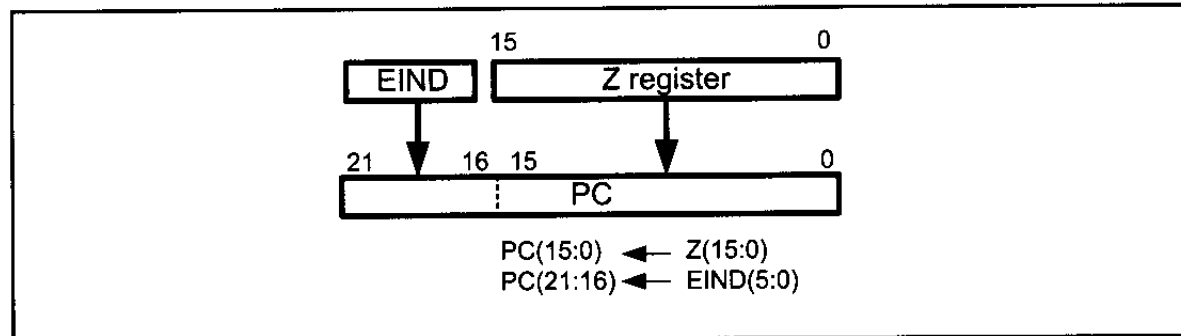


Figure 3-11. EICALL Instruction

The ICALL and EICALL instructions can be used to implement pointer to function.

Review Questions

1. True or false. In the AVR, control can be transferred anywhere within the 4M of code space by using the CALL instruction.
2. The CALL instruction is a(n) ____ -byte instruction.
3. True or false. In the AVR, control can be transferred anywhere within the 4M of code space by using the RCALL instruction.
4. With each CALL instruction, the stack pointer register, SP, is _____ (incremented, decremented).
5. With each RET instruction, the SP is _____ (incremented, decremented).
6. On power-up, the SP points to address _____.
7. How deep is the size of the stack in the AVR?
8. The RCALL instruction is a(n) ____ -byte instruction.
9. _____ (RCALL, CALL) takes more ROM space.

SECTION 3.3: AVR TIME DELAY AND INSTRUCTION PIPELINE

In the last section we used the DELAY subroutine. In this section we discuss how to generate various time delays and calculate exact delays for the AVR. We will also discuss instruction pipelining and its impact on execution time.

Delay calculation for the AVR

In creating a time delay using Assembly language instructions, one must be mindful of two factors that can affect the accuracy of the delay:

1. The crystal frequency: The frequency of the crystal oscillator connected to the XTAL1 and XTAL2 input pins is one factor in the time delay calculation. The duration of the clock period for the instruction cycle is a function of this crystal frequency.
2. The AVR design: Since the 1970s, both the field of IC technology and the architectural design of microprocessors have seen great advancements. Due to the limitations of IC technology and limited CPU design experience for many years, the instruction cycle duration was longer. Advances in both IC technology and CPU design in the 1980s and 1990s have made the single instruction cycle a common feature of many microcontrollers. Indeed, one way to increase performance without losing code compatibility with the older generation of a given family is to reduce the number of instruction cycles it takes to execute an instruction. One might wonder how microprocessors such as AVR are able to execute an instruction in one cycle. There are three ways to do that: (a) Use Harvard architecture to get the maximum amount of code and data into the CPU, (b) use RISC architecture features such as fixed-size instructions, and finally (c) use pipelining to overlap fetching and execution of instructions. We examined the Harvard and RISC architectures in Chapter 2. Next, we discuss pipelining.

Pipelining

In early microprocessors such as the 8085, the CPU could either fetch or execute at a given time. In other words, the CPU had to fetch an instruction from memory, then execute it; and then fetch the next instruction, execute it, and so on. The idea of pipelining in its simplest form is to allow the CPU to fetch and execute at the same time, as shown in Figure 3-12. (An instruction fetches while the

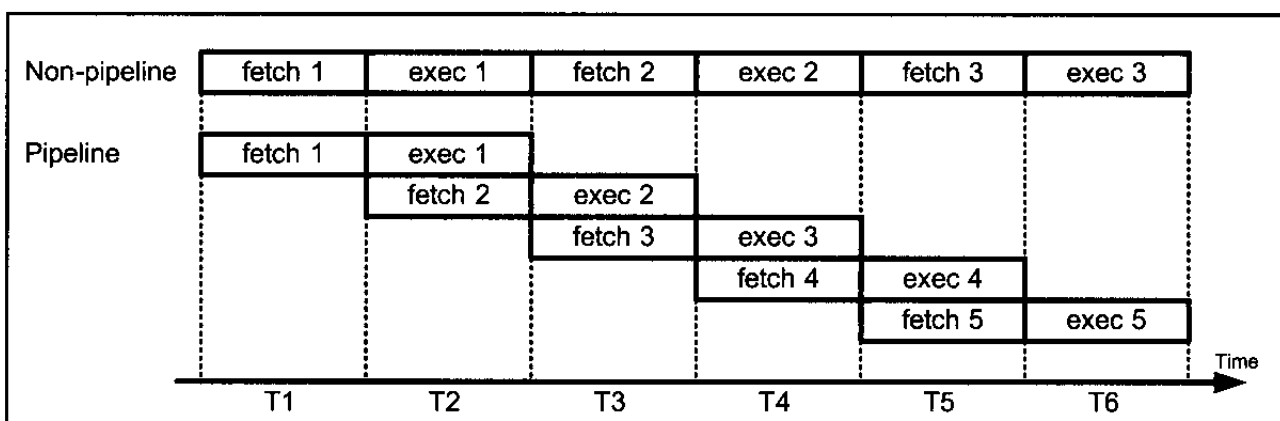


Figure 3-12. Pipeline vs. Non-pipeline

previous instruction executes.)

We can use a pipeline to speed up execution of instructions. In pipelining, the process of executing instructions is split into small steps that are all executed in parallel. In this way, the execution of many instructions is overlapped. One limitation of pipelining is that the speed of execution is limited to the slowest stage of the pipeline. Compare this to making pizza. You can split the process of making pizza into many stages, such as flattening the dough, putting on the toppings, and baking, but the process is limited to the slowest stage, baking, no matter how fast the rest of the stages are performed. What happens if we use two or three ovens for baking pizzas to speed up the process? This may work for making pizza but not for executing programs, because in the execution of instructions we must make sure that the sequence of instructions is kept intact and that there is no out-of-step execution.

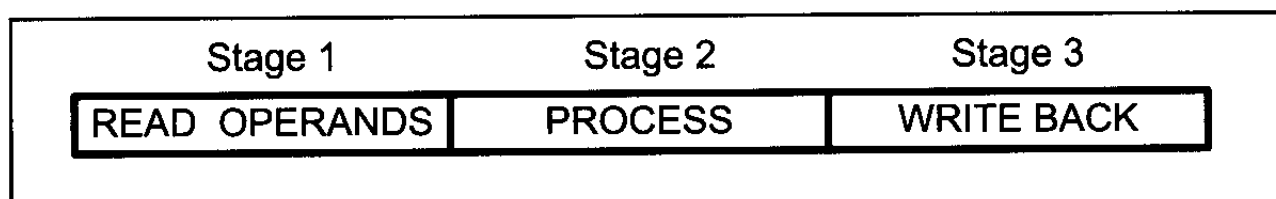


Figure 3-13. Single Cycle ALU Operation

AVR multistage execution pipeline

As shown in Figure 3-13, in the AVR, each instruction is executed in 3 stages: operand fetch, ALU operation execution, and result write back.

In step 1, the operand is fetched. In step 2, the operation is performed; for example, the adding of the two numbers is done. In step 3, the result is written into the destination register. In reality, one can construct the AVR pipeline for three instructions, as is shown in Figure 3-14.

It should be noted that in many computer architecture books, the process stage is referred to as *execute* and write back is called *write*.

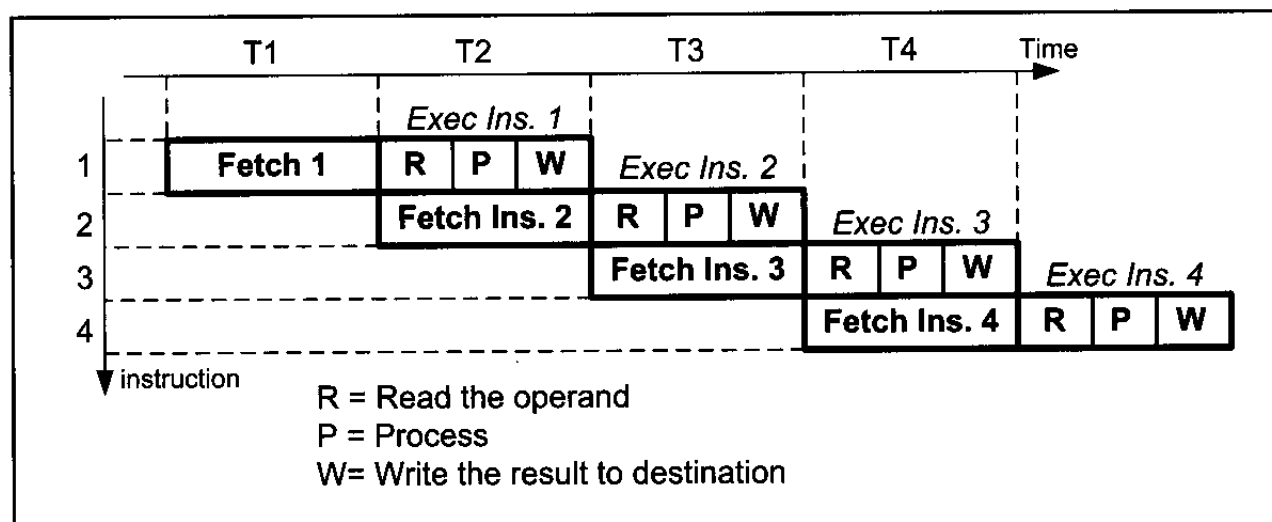


Figure 3-14. Pipeline Activity for Both Fetch and Execute

Instruction cycle time for the AVR

It takes a certain amount of time for the CPU to execute an instruction. This time is referred to as *machine cycles*. Because all the instructions in the AVR are either 1-word (2-byte) or 2-word (4-byte), most instructions take no more than one or two machine cycles to execute. (Notice, however, that some instructions such as JMP and CALL could take up to three or four machine cycles.) Appendix A provides a list of AVR instructions and their cycles. In the AVR family, the length of the machine cycle depends on the frequency of the oscillator connected to the AVR system. The crystal oscillator, along with on-chip circuitry, provide the clock source for the AVR CPU (see Chapter 8). In the AVR, one machine cycle consists of one oscillator period, which means that with each oscillator clock, one machine cycle passes. Therefore, to calculate the machine cycle for the AVR, we take the inverse of the crystal frequency, as shown in Example 3-14.

Example 3-14

The following shows the crystal frequency for four different AVR-based systems. Find the period of the instruction cycle in each case.

- (a) 8 MHz (b) 16 MHz (c) 10 MHz (d) 1 MHz

Solution:

- (a) instruction cycle is $1/8 \text{ MHz} = 0.125 \mu\text{s}$ (microsecond) = 125 ns (nanosecond)
(b) instruction cycle = $1/16 \text{ MHz} = 0.0625 \mu\text{s} = 62.5 \text{ ns}$ (nanosecond)
(c) instruction cycle = $1/10 \text{ MHz} = 0.1 \mu\text{s} = 100 \text{ ns}$
(d) instruction cycle = $1/1 \text{ MHz} = 1 \mu\text{s}$

Branch penalty

The overlapping of fetch and execution of the instruction is widely used in today's microcontrollers such as AVR. For the concept of pipelining to work, we need a buffer or queue in which an instruction is prefetched and ready to be executed. In some circumstances, the CPU must flush out the queue. For example, when a branch instruction is executed, the CPU starts to fetch codes from the new memory location, and the code in the queue that was fetched previously is discarded. In this case, the execution unit must wait until the fetch unit fetches the new instruction. This is called a *branch penalty*. The penalty is an extra instruction cycle to fetch the instruction from the target location instead of executing the instruction right below the branch. Remember that the instruction below the branch has already been fetched and is next in line to be executed when the CPU branches to a different address. This means that while the vast majority of AVR instructions take only one machine cycle, some instructions take two, three, or four machine cycles. These are JMP, CALL, RET, and all the conditional branch instructions such as BRNE, BRLO, and so on. The conditional branch instruction can take only one machine cycle if it does not jump. For example, the BRNE will jump if $Z = 0$, and that takes two machine cycles. If $Z = 1$, then it falls through and it takes only one machine cycle. See Examples 3-15 and 3-16.

Example 3-15

For an AVR system of 1 MHz, find how long it takes to execute each of the following instructions:

- | | | |
|----------|----------|----------|
| (a) LDI | (b) DEC | (c) LD |
| (d) ADD | (e) NOP | (f) JMP |
| (g) CALL | (h) BRNE | (i) .DEF |

Solution:

The machine cycle for a system of 1 MHz is 1 μ s, as shown in Example 3-14. Appendix A shows instruction cycles for each of the above instructions. Therefore, we have:

<i>Instruction</i>	<i>Instruction cycles</i>	<i>Time to execute</i>
(a) LDI	1	$1 \times 1 \mu\text{s} = 1 \mu\text{s}$
(b) DEC	1	$1 \times 1 \mu\text{s} = 1 \mu\text{s}$
(c) OUT	1	$1 \times 1 \mu\text{s} = 1 \mu\text{s}$
(d) ADD	1	$1 \times 1 \mu\text{s} = 1 \mu\text{s}$
(e) NOP	1	$1 \times 1 \mu\text{s} = 1 \mu\text{s}$
(f) JMP	3	$3 \times 1 \mu\text{s} = 2 \mu\text{s}$
(g) CALL	4	$4 \times 1 \mu\text{s} = 4 \mu\text{s}$
(h) BRNE	2/1	(2 μ s taken, 1 μ s if it falls through)
(i) .DEF	0	(directive instructions do not produce machine instructions)

Example 3-16

Find the size of the delay of the code snippet below if the crystal frequency is 10 MHz:

Solution:

From Appendix A, we have the following machine cycles for each instruction of the DELAY subroutine:

			<i>Instruction Cycles</i>
	.DEF COUNT = R20		0
DELAY:	LDI COUNT, 0xFF		1
AGAIN:	NOP		1
	NOP		1
	DEC COUNT		1
	BRNE AGAIN		2/1
	RET		4

Therefore, we have a time delay of $[1 + ((1 + 1 + 1 + 2) \times 255) + 4] \times 0.1 \mu\text{s} = 128.0 \mu\text{s}$. Notice that BRNE takes two instruction cycles if it jumps back, and takes only one when falling through the loop. That means the above number should be 127.9 μ s.

Delay calculation for AVR

As seen in the last section, a delay subroutine consists of two parts: (1) setting a counter, and (2) a loop. Most of the time delay is performed by the body of the loop, as shown in Examples 3-17 and 3-18.

Very often we calculate the time delay based on the instructions inside the loop and ignore the clock cycles associated with the instructions outside the loop.

In Example 3-16, the largest value the R20 register can take is 255. One way to increase the delay is to use NOP instructions in the loop. NOP, which stands for “no operation,” simply wastes time, but takes 2 bytes of program ROM space,

Example 3-17

Find the size of the delay in the following program if the crystal frequency is 1 MHz:

```
.INCLUDE "M32DEF.INC"
.ORG 0
    LDI R16,HIGH(RAMEND) ;initialize SP
    OUT SPH,R16
    LDI R16,LOW(RAMEND)
    OUT SPL,R16
BACK:
    LDI R16,0x55          ;load R16 with 0x55
    OUT PORTB,R16        ;send 55H to port B
    RCALL DELAY           ;time delay
    LDI R16,0xAA          ;load R16 with 0xAA
    OUT PORTB,R16        ;send 0xAA to port B
    RCALL DELAY           ;time delay
    RJMP BACK            ;keep doing this indefinitely
;-----this is the delay subroutine
    .ORG 0x300            ;put time delay at address 0x300
DELAY: LDI R20,0xFF       ;R20 = 255,the counter
AGAIN:
    NOP                  ;no operation wastes clock cycles
    NOP
    DEC R20
    BRNE AGAIN           ;repeat until R20 becomes 0
    RET                  ;return to caller
```

Solution:

From Appendix A, we have the following machine cycles for each instruction of the DELAY subroutine:

Instruction Cycles

DELAY:	LDI R20,0xFF	1
AGAIN:	NOP	1
	NOP	1
	DEC R20	1
	BRNE AGAIN	2/1
	RET	4

Therefore, we have a time delay of $[1 + (255 \times 5) - 1 + 4] \times 1 \mu\text{s} = 1279 \mu\text{s}$.

which is too heavy a price to pay for just one instruction cycle. A better way is to use a nested loop.

Loop inside a loop delay

Another way to get a large delay is to use a loop inside a loop, which is also called a *nested loop*. See Example 3-18. Compare that with Example 3-19 to see the disadvantage of using many NOPs. Also see Example 3-20.

From these discussions we conclude that the use of instructions in generating time delay is not the most reliable method. To get more accurate time delay we

Example 3-18

For an instruction cycle of $1\ \mu\text{s}$ (a) find the time delay in the following subroutine, and (b) find the amount of ROM it takes.

<i>Instruction Cycles</i>			
DELAY:	LDI	R16,200	1
AGAIN:	LDI	R17,250	1
HERE:	NOP		1
	NOP		1
	DEC	R17	1
	BRNE	HERE	2/1
	DEC	R16	1
	BRNE	AGAIN	2/1
	RET		4

Solution:

(a)

For the HERE loop, we have $[(5 \times 250) - 1] \times 1\ \mu\text{s} = 1249\ \mu\text{s}$. (We should subtract 1 for the times BRNE HERE falls through.) The AGAIN loop repeats the HERE loop 200 times; therefore, we have $200 \times 1249\ \mu\text{s} = 249,800\ \mu\text{s}$, if we do not include the overhead. However, the following instructions of the outer loop add to the delay:

AGAIN:	LDI	R17,250	1
		
	DEC	R16	1
	BRNE	AGAIN	2/1

The above instructions at the beginning and end of the AGAIN loop add $[(4 \times 200) - 1] \times 1\ \mu\text{s} = 799\ \mu\text{s}$ to the time delay. As a result we have $249,800 + 799 = 250,599\ \mu\text{s}$ for the total time delay associated with the above DELAY subroutine. Notice that this calculation is an approximation because we have ignored the "LDI R16,200" instruction and the last instruction, RET, in the subroutine.

(b)

There are 9 instructions in the above DELAY program, and all the instructions are 2-byte instructions. That means that the loop delay takes 22 bytes of ROM code space.

Example 3-19

Find the time delay for the following subroutine, assuming a crystal frequency of 1 MHz. Discuss the disadvantage of this over Example 3-18.

<i>Machine Cycles</i>			
DELAY:	LDI	R16, 200	1
AGAIN:	NOP		1
	NOP		1
	NOP		1
	NOP		1
	NOP		1
	NOP		1
	NOP		1
	NOP		1
	NOP		1
	NOP		1
	NOP		1
	NOP		1
	NOP		1
	DEC	R16	1
	BRNE	AGAIN	2
	RET		4

Solution:

The time delay inside the AGAIN loop is $[200(13 + 2)] \times 1 \mu\text{s} = 3000 \mu\text{s}$. NOP is a 2-byte instruction, even though it does not do anything except to waste cycle time. There are 16 instructions in the above DELAY program, and all the instructions are 2-byte instructions. This means the loop delay takes 32 bytes of ROM code space, and gives us only a 3000 μs delay. That is the reason we use a nested loop instead of NOP instructions to create a time delay. Chapter 9 shows how to use AVR timers to create delays much more efficiently.

use timers, as described in Chapter 9. We can use AVR Studio's simulator to verify delay time and number of cycles used. Meanwhile, to get an accurate time delay for a given AVR microcontroller, we must use an oscilloscope to measure the exact time delay.

Review Questions

1. True or false. In the AVR, the machine cycle lasts 1 clock period of the crystal frequency.
2. The minimum number of machine cycles needed to execute an AVR instruction is _____.
3. For Question 2, what is the maximum number of cycles needed, and for which instructions?
4. Find the machine cycle for a crystal frequency of 12 MHz.
5. Assuming a crystal frequency of 1 MHz, find the time delay associated with the loop section of the following DELAY subroutine:

```
DELAY:    LDI        R20, 100
HERE:     NOP
```

Example 3-20

Write a program to toggle all the bits of I/O register PORTB every 1 s. Assume that the crystal frequency is 8 MHz and the system is using an ATmega32.

Solution:

```
.INCLUDE "M32DEF.INC"
.ORG 0
    LDI    R16,HIGH(RAMEND)
    OUT    SPH,R16
    LDI    R16,LOW(RAMEND)
    OUT    SPL,R16

    LDI    R16,0x55           ;load R16 with 0x55
BACK:  COM    R16              ;complement PORTB
    OUT    PORTB,R16         ;send it to port B
    CALL   DELAY_1S          ;time delay
    RJMP   BACK              ;keep doing this indefinitely

DELAY_1S:
    LDI    R20,32
L1:    LDI    R21,200
L2:    LDI    R22,250
L3:
    NOP
    NOP
    DEC    R22
    BRNE   L3
    DEC    R21
    BRNE   L2
    DEC    R20
    BRNE   L1
    RET
```

Machine cycle = $1 / 8 \text{ MHz} = 125 \text{ ns}$

Delay = $32 \times 200 \times 250 \times 5 \times 125 \text{ ns} = 1,000,000,000 \text{ ns} = 1,000,000 \mu\text{s} = 1 \text{ s}$.

In this calculation, we have not included the overhead associated with the two outer loops. Use the AVR Studio simulator to verify the delay.

```
    NOP
    NOP
    NOP
    NOP
    DEC    R20
    BRNE   HERE
    RET
```

6. True or false. In the AVR, the machine cycle lasts 6 clock periods of the crystal frequency.
7. Find the machine cycle for an AVR if the crystal frequency is 8 MHz.
8. True or false. In the AVR, the instruction fetching and execution are done at the same time.
9. True or false. JMP and RCALL will always take 3 machine cycles.
10. True or false. The BRNE instruction will always take 2 machine cycles.

SUMMARY

The flow of a program proceeds sequentially, from instruction to instruction, unless a control transfer instruction is executed. The various types of control transfer instructions in Assembly language include conditional and unconditional branches, and call instructions.

Looping in AVR Assembly language is performed using an instruction to decrement a counter and to jump to the top of the loop if the counter is not zero. This is accomplished with the BRNE instruction. Other branch instructions jump conditionally, based on the value of the carry flag, the Z flag, or other bits of the status register. Unconditional branches can be long or short, depending on the location of the target address. Special attention must be given to the effect of CALL and RCALL instructions on the stack.

PROBLEMS

SECTION 3.1: BRANCH INSTRUCTIONS AND LOOPING

1. In the AVR, looping action with the “BRNE target” instruction is limited to ____ iterations.
2. If a conditional branch is not taken, what is the next instruction to be executed?
3. In calculating the target address for a branch, a displacement is added to the contents of register ____.
4. The mnemonic RJMP stands for _____ and it is a(n) ____-byte instruction.
5. The JMP instruction is a(n) ____-byte instruction.
6. What is the advantage of using RJMP over JMP?
7. True or false. The target of a BRNE can be anywhere in the 4M word address space.
8. True or false. All AVR branch instructions can branch to anywhere in the 4M word address space.
9. Which of the following instructions is (are) 2-byte instructions.
(a) BREQ (b) BRSH (c) JMP (d) RJMP
10. Dissect the RJMP instruction, indicating how many bits are used for the operand and the opcode, and indicate how far it can branch.
11. True or false. All conditional branches are 2-byte instructions.
12. Show code for a nested loop to perform an action 1,000 times.
13. Show code for a nested loop to perform an action 100,000 times.
14. Find the number of times the following loop is performed:

```
LDI          R20, 200
BACK: LDI    R21, 100
HERE:  DEC   R21
      BRNE  HERE
      DEC   R20
      BRNE  BACK
```
15. The target address of a BRNE is backward if the relative address of the opcode is _____ (negative, positive).

16. The target address of a BRNE is forward if the relative address of the opcode is _____ (negative, positive).

SECTION 3.2: CALL INSTRUCTIONS AND STACK

17. CALL is a(n) ____-byte instruction.
18. RCALL is a(n) ____-byte instruction.
19. True or false. The RCALL target address can be anywhere in the 4M (word) address space.
20. True or false. The CALL target address can be anywhere in the 4M address space.
21. When CALL is executed, how many locations of the stack are used?
22. When RCALL is executed, how many locations of the stack are used?
23. Upon reset, the SP points to location _____.
24. Describe the action associated with the RET instruction.
25. Give the size of the stack in AVR.
26. In AVR, which address is pushed into the stack when a call instruction is executed.

SECTION 3.3: AVR TIME DELAY AND INSTRUCTION PIPELINE

27. Find the oscillator frequency if the machine cycle = $1.25\ \mu\text{s}$.
28. Find the machine cycle if the crystal frequency is 20 MHz.
29. Find the machine cycle if the crystal frequency is 10 MHz.
30. Find the machine cycle if the crystal frequency is 16 MHz.
31. True or false. The CALL and RCALL instructions take the same amount of time to execute even though one is a 4-byte instruction and the other is a 2-byte instruction.
32. Find the time delay for the delay subroutine shown below if the system has an AVR with a frequency of 8 MHz:

```
                LDI            R16, 200
BACK:          LDI            R18, 100
HERE:          NOP
                DEC            R18
                BRNE           HERE
                DEC            R16
                BRNE           BACK
```

33. Find the time delay for the delay subroutine shown below if the system has an AVR with a frequency of 8 MHz:

```
                LDI            R20, 200
BACK:          LDI            R22, 100
HERE:          NOP
                NOP
                DEC            R22
                BRNE           HERE
                DEC            R20
                BRNE           BACK
```

34. Find the time delay for the delay subroutine shown below if the system has an AVR with a frequency of 4 MHz:

```

        LDI        R20, 200
BACK:   LDI        R21, 250
HERE:   NOP
        DEC        R21
        BRNE       HERE
        DEC        R20
        BRNE       BACK

```

35. Find the time delay for the delay subroutine shown below if the system has an AVR with a frequency of 10 MHz:

```

        LDI        R20, 200
BACK:   LDI        R25, 100
        NOP
        NOP
        NOP
HERE    DEC        R25
        BRNE       HERE
        DEC        R20
        BRNE       BACK

```

ANSWERS TO REVIEW QUESTIONS

SECTION 3.1: BRANCH INSTRUCTIONS AND LOOPING

1. Branch if not equal
2. True
3. 2
4. Z flag of SREG (status register)
5. 4

SECTION 3.2: CALL INSTRUCTIONS AND STACK

1. True
2. 4
3. False
4. Decrement
5. Increment
6. 0
7. The AVR's stack can be as big as its RAM memory.
8. 2
9. CALL

SECTION 3.3: AVR TIME DELAY AND INSTRUCTION PIPELINE

1. True
2. 1
3. 4; CALL, RET
4. $12 \text{ MHz} / 4 = 3 \text{ MHz}$, and $MC = 1/3 \text{ MHz} = 0.333 \mu\text{s}$
5. $[100 \times (1 + 1 + 1 + 1 + 1 + 1 + 2)] \times 1 \mu\text{s} = 800 \mu\text{s} = 0.8 \text{ milliseconds}$
6. False. It takes 4 clocks.
7. $\text{Machine cycle} = 1 / 8 \text{ MHz} = 0.125 \mu\text{s} = 125 \text{ ns}$
8. True
9. True
10. False. Only if it branches to the target address.

CHAPTER 4

AVR I/O PORT PROGRAMMING

OBJECTIVES

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

- >> List all the ports of the AVR
- >> Describe the dual role of AVR pins
- >> Code Assembly language to use the ports for input or output
- >> Explain the dual role of Ports A, B, C, and D
- >> Code AVR instructions for I/O handling
- >> Code I/O bit-manipulation programs for the AVR
- >> Explain the bit-addressability of AVR ports

This chapter describes I/O port programming of the AVR with many examples. In Section 4.1, we describe I/O access using byte-size data, and in Section 4.2, bit manipulation of the I/O ports is discussed in detail.

SECTION 4.1: I/O PORT PROGRAMMING IN AVR

In the AVR family, there are many ports for I/O operations, depending on which family member you choose. Examine Figure 4-1 for the ATmega32 40-pin chip. A total of 32 pins are set aside for the four ports PORTA, PORTB, PORTC, and PORTD. The rest of the pins are designated as VCC, GND, XTAL1, XTAL2, RESET, AREF, AGND, and AVCC. They are discussed in Chapter 8.

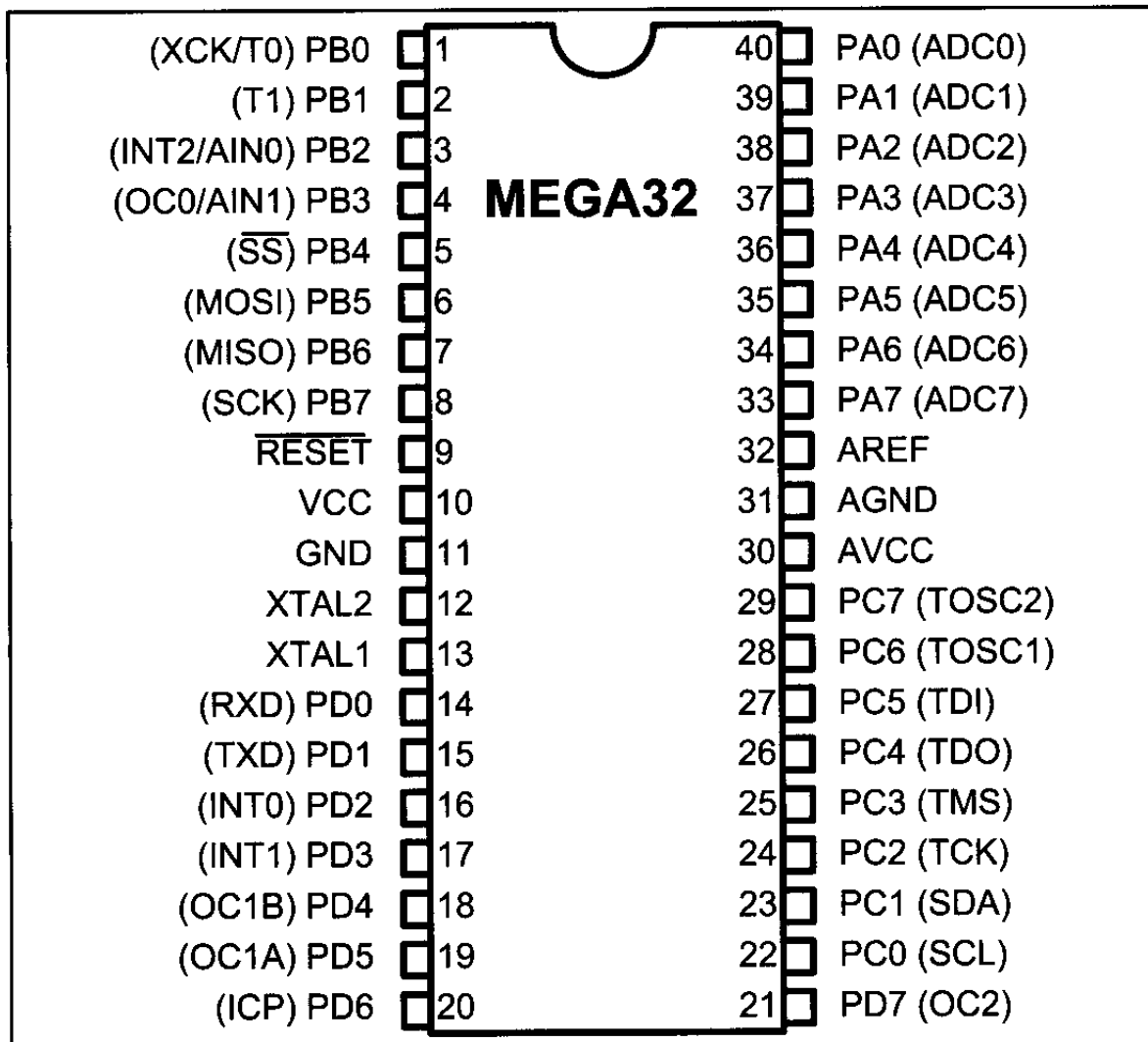


Figure 4-1. ATmega32 Pin Diagram

I/O port pins and their functions

The number of ports in the AVR family varies depending on the number of pins on the chip. The 8-pin AVR has port B only, while the 64-pin version has ports A through F, and the 100-pin AVR has ports A through L, as shown in Table 4-1. The 40-pin AVR has four ports. They are PORTA, PORTB, PORTC, and PORTD. To use any of these ports as an input or output port, it must be programmed, as we will explain throughout this section. In addition to being used for simple I/O, each

Table 4-1: Number of Ports in Some AVR Family Members

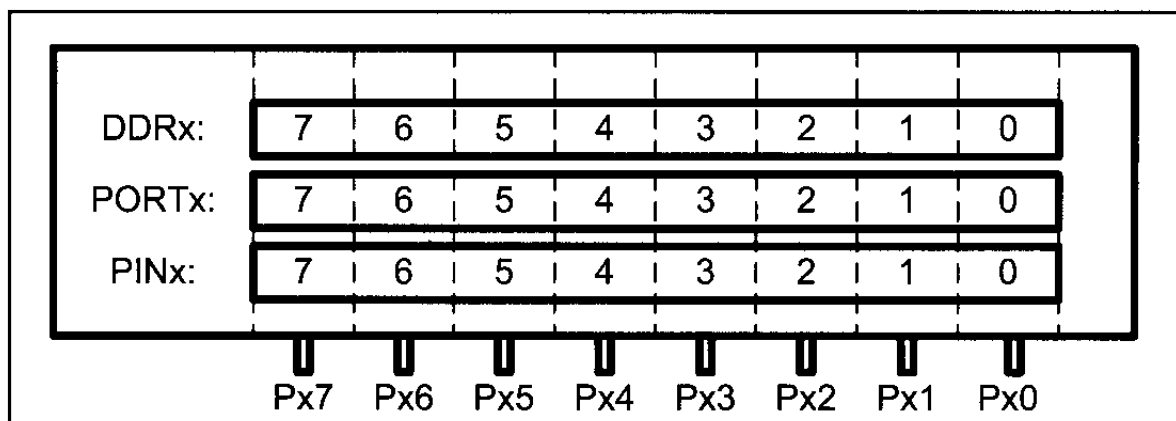
Pins	8-pin	28-pin	40-pin	64-pin	100-pin
Chip	ATtiny25/45/85	ATmega8/48/88	ATmega32/16	ATmega64/128	ATmega1280
Port A			X	X	X
Port B	6 bits	X	X	X	X
Port C		7 bits	X	X	X
Port D		X	X	X	X
Port E				X	X
Port F				X	X
Port G				5 bits	6 bits
Port H					X
Port J					X
Port K					X
Port L					X

Note: X indicates that the port is available.

port has some other functions such as ADC, timers, interrupts, and serial communication pins. Figure 4-1 shows alternate functions for the ATmega32 pins. We will study all these alternate functions in future chapters. In this chapter we focus on the simple I/O function of the AVR family. Not all ports have 8 pins. For example, in the ATmega8, Port C has 7 pins. Each port has three I/O registers associated with it, as shown in Table 4-2. They are designated as PORTx, DDRx, and PINx. For example, for Port B we have PORTB, DDRB, and PINB. Notice that DDR stands for Data Direction Register, and PIN stands for Port Input pins. Also notice that each of the I/O registers is 8 bits wide, and each port has a maximum of 8 pins; therefore each bit of the I/O registers affects one of the pins (see Figure 4-2; the content of bit 0 of DDRB represents the direction of the PB0 pin, and so on). Next, we describe how to access the I/O registers associated with the ports.

Table 4-2: Register Addresses for ATmega32 Ports

Port	Address	Usage
PORTA	\$3B	output
DDRA	\$3A	direction
PINA	\$39	input
PORTB	\$38	output
DDRB	\$37	direction
PINB	\$36	input
PORTC	\$35	output
DDRC	\$34	direction
PINC	\$33	input
PORTD	\$32	output
DDRD	\$31	direction
PIND	\$30	input

**Figure 4-2. Relations Between the Registers and the Pins of AVR**

DDRx register role in outputting data

Each of the ports A–D in the ATmega32 can be used for input or output. The DDRx I/O register is used solely for the purpose of making a given port an input or output port. For example, to make a port an output, we write 1s to the DDRx register. In other words, to output data to all of the pins of the Port B, we must first put 0b11111111 into the DDRB register to make all of the pins output.

The following code will toggle all 8 bits of Port B forever with some time delay between “on” and “off” states:

```
LDI    R16,0xFF      ;R16 = 0xFF = 0b11111111
OUT    DDRB,R16      ;make Port B an output port (1111 1111)
L1:    LDI    R16,0x55 ;R16 = 0x55 = 0b01010101
OUT    PORTB,R16     ;put 0x55 on port B pins
CALL   DELAY
LDI    R16,0xAA      ;R16 = 0xAA = 0b10101010
OUT    PORTB,R16     ;put 0xAA on port B pins
CALL   DELAY
RJMP   L1
```

It must be noted that unless we set the DDRx bits to one, the data will not go from the port register to the pins of the AVR. This means that if we remove the first two lines of the above code, the 0x55 and 0xAA values will not get to the pins. They will be sitting in the I/O register of Port B inside the CPU.

To see the role of the DDRx register in allowing the data to go from Portx to the pins, examine Figure 4-3. For more information about the internal circuitry of I/O ports, see Appendix C.

DDR register role in inputting data

To make a port an input port, we must first put 0s into the DDRx register for that port, and then bring in (read) the data present at the pins. As an aid for remembering that the port is input when the DDR bits are 0s, imagine a person who has 0 dollars. The person can only get money, not give it. Similarly, when DDR contains 0s, the port gets data.

Notice that upon reset, all ports have the value 0x00 in their DDR registers. This means that all ports are configured as input as we will see next.

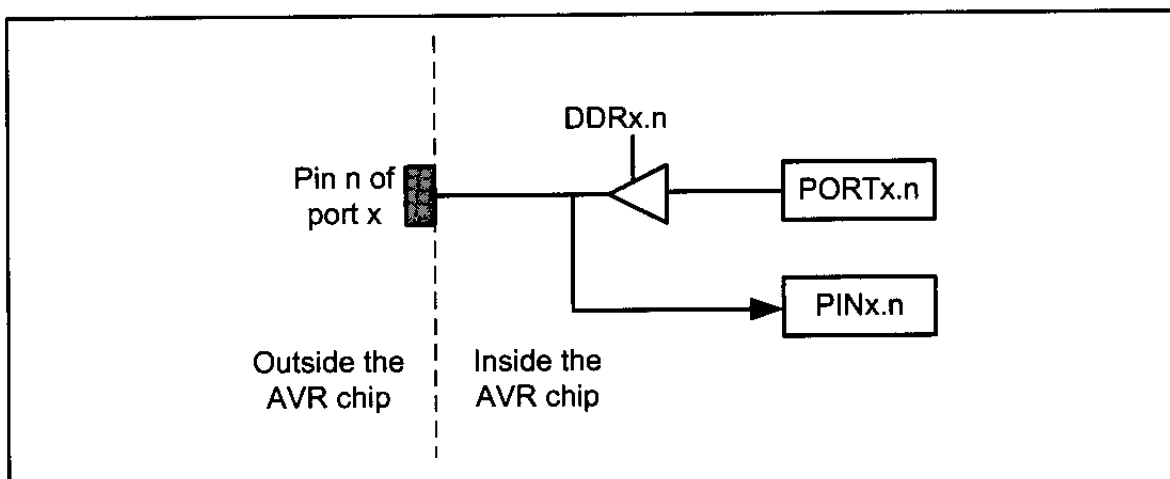


Figure 4-3. The I/O Port in AVR

PIN register role in inputting data

To read the data present at the pins, we should read the PIN register. It must be noted that to bring data into CPU from pins we read the contents of the PINx register, whereas to send data out to pins we use the PORTx register.

PORT register role in inputting data

There is a pull-up resistor for each of the AVR pins. If we put 1s into bits of the PORTx register, the pull-up resistors are activated. In cases in which nothing is connected to the pin or the connected devices have high impedance, the resistor pulls up the pin. See Figure 4-4.

If we put 0s into the bits of the PORTx register, the pull-up resistor is inactive.

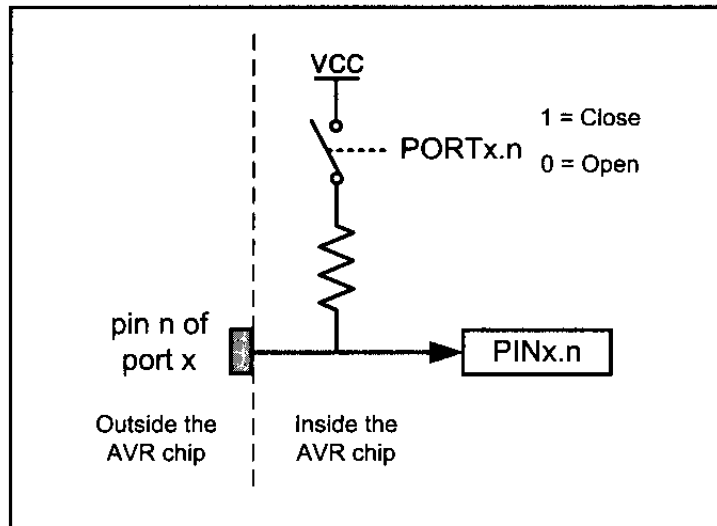


Figure 4-4. The Pull-up Resistor

The following code gets the data present at the pins of port C and sends it to port B indefinitely, after adding the value 5 to it:

```
.INCLUDE "M32DEF.INC"
    LDI    R16,0x00    ;R16 = 00000000 (binary)
    OUT    DDRC,R16    ;make Port C an input port
    LDI    R16,0xFF    ;R16 = 11111111 (binary)
    OUT    DDRB,R16    ;make Port B an output port(1 for Out)
L2:   IN     R16,PINC    ;read data from Port C and put in R16
    LDI    R17,5
    ADD    R16,R17      ;add 5 to it
    OUT    PORTB,R16    ;send it to Port B
    RJMP   L2           ;continue forever
```

If we want to make the pull-up resistors of port C active, we must put 1s into the PORTC register. The program becomes as follows:

```
.INCLUDE "M32DEF.INC"
    LDI    R16,0xFF    ;R16 = 11111111 (binary)
    OUT    DDRB,R16    ;make Port B an output port
    OUT    PORTC,R16    ;make the pull-up resistors of C active
    LDI    R16,0x00    ;R16 = 00000000 (binary)
    OUT    DDRC,R16    ;Port C an input port (0 for I)
L2:   IN     R16,PINC    ;move data from Port C to R16
    LDI    R17,5
    ADD    R16,R17      ;add some value to it
    OUT    PORTB,R16    ;send it to Port B
    RJMP   L2           ;continue forever
```

Again, it must be noted that unless we clear the DDR bits (by putting 0s there), the data will not be brought into the registers from the pins of Port C. To

see the role of the DDRx register in allowing the data to come into the CPU from the pins, examine Figure 4-3.

The pins of the AVR microcontrollers can be in four different states according to the values of PORTx and DDRx, as shown in Figure 4-5.

PORTx	DDRx	0	1
		Input & high impedance	Out 0
0			
1			

Figure 4-5. Different States of a Pin in the AVR Microcontroller

This is one of powerful features of the AVR microcontroller, since most of the other microcontrollers' pins (e.g., 8051) have fewer states.

Port A

Port A occupies a total of 8 pins (PA0–PA7). To use the pins of Port A as input or output ports, each bit of the DDRA register must be set to the proper value. For example, the following code will continuously send out to Port A the alternating values of 0x55 and 0xAA:

```
;toggle all bits of PORTA
.INCLUDE      "M32DEF.INC"
    LDI       R16,0xFF      ;R16 = 11111111 (binary)
    OUT       DDRA,R16      ;make Port A an output port
L1:   LDI       R16,0x55     ;R16 = 0x55
    OUT       PORTA,R16     ;put 0x55 on Port A pins
    CALL      DELAY
    LDI       R16,0xAA      ;R16 = 0xAA
    OUT       PORTA,R16     ;put 0xAA on Port A pins
    CALL      DELAY
    RJMP      L1
```

It must be noted that 0x55 (01010101) when complemented becomes 0xAA (10101010).

Port A as input

In order to make all the bits of Port A an input, DDRA must be cleared by writing 0 to all the bits. In the following code, Port A is configured first as an input port by writing all 0s to register DDRA, and then data is received from Port A and saved in a RAM location:

```
.INCLUDE      "M32DEF.INC"
    .EQU      MYTEMP 0x100    ;save it here
    LDI       R16,0x00      ;R16 = 00000000 (binary)
    OUT       DDRA,R16      ;make Port A an input port (0 for In)
    NOP                          ;synchronizer delay
    IN        R16,PINA       ;move from pins of Port A to R16
    STS       MYTEMP,R16     ;save it in MYTEMP
```

Synchronizer delay

The input circuit of the AVR has a delay of 1 clock cycle. In other words, the PIN register represents the data that was present at the pins one clock ago. In the above code, when the instruction “IN R16, PINA” is executed, the PINA register contains the data, which was present at the pins one clock before. That is why the NOP is put before the “IN R16, PINA” instruction. (If the NOP is omitted, the read data is the data of the pins when the port was output.)

For more information see Section C-2.

Port B

Port B occupies a total of 8 pins (PB0–PB7). To use the pins of Port B as input or output ports, each bit of the DDRB register must be set to the proper value.

For example, the following code will continuously send out the alternating values of 0x55 and 0xAA to Port B:

```
        ;toggle all bits of PORTB
.INCLUDE "M32DEF.INC"
        LDI    R16,0xFF      ;R16 = 11111111 (binary)
        OUT    DDRB,R16      ;make Port B an output port (1 for Out)
L1:      LDI    R16,0x55      ;R16 = 0x55
        OUT    PORTB,R16     ;put 0x55 on Port B pins
        CALL   DELAY
        LDI    R16,0xAA      ;R16 = 0xAA
        OUT    PORTB,R16     ;put 0xAA on Port B pins
        CALL   DELAY
        RJMP   L1
```

Port B as input

In order to make all the bits of Port B an input, DDRB must be cleared by writing 0 to all the bits. In the following code, Port B is configured first as an input port by writing all 0s to register DDRB, and then data is received from Port B and saved in some RAM location:

```
.INCLUDE "M32DEF.INC"
        .EQU    MYTEMP=0x100 ;save it here
        LDI    R16,0x00      ;R16 = 00000000 (binary)
        OUT    DDRB,R16      ;make Port B an input port (0 for In)
        NOP
        IN     R16,PINB       ;move from pins of Port B to R16
        STS    MYTEMP,R16     ;save it in MYTEMP
```

Dual role of Ports A and B

The AVR multiplexes an analog-to-digital converter through Port A to save I/O pins. The alternate functions of the pins for Port A are shown in Table 4-3. We will show how to use Port A's ADC in Chapter 13. Because many projects use an ADC, we usually do not use Port A for simple I/O functions.

The AVR multiplexes some other functions through Port B to save pins.

The alternate functions of the pins for Port B are shown in Table 4-4. We will show how to use the alternate functions of Port B in future chapters.

Table 4-3: Port A Alternate Functions

Bit	Function
PA0	ADC0
PA1	ADC1
PA2	ADC2
PA3	ADC3
PA4	ADC4
PA5	ADC5
PA6	ADC6
PA7	ADC7

Table 4-4: Port B Alternate Functions

Bit	Function
PB0	XCK/T0
PB1	T1
PB2	INT2/AIN0
PB3	OC0/AIN1
PB4	SS
PB5	MOSI
PB6	MISO
PB7	SCK

Port C

Port C occupies a total of 8 pins (PC0–PC7). To use the pins of Port C as input or output ports, each bit of the DDRC register must be set to the proper value. For example, the following code will continuously send out the alternating values of 0x55 and 0xAA to Port C:

```

;toggle all bits of PORTB
.INCLUDE "M32DEF.INC"
LDI R16,0xFF ;R16 = 11111111 (binary)
OUT DDRC,R16 ;make Port C an output port (1 for Out)
L1: LDI R16,0x55 ;R16 = 0x55
OUT PORTC,R16 ;put 0x55 on Port C pins
CALL DELAY
LDI R16,0xAA ;R16 = 0xAA
OUT PORTC,R16 ;put 0xAA on Port C pins
CALL DELAY
RJMP L1

```

Port C as input

In order to make all the bits of Port C an input, DDRC must be cleared by writing 0 to all the bits. In the following code, Port C is configured first as an input port by writing all 0s to register DDRC, and then data is received from Port C and saved in a RAM location:

```

.INCLUDE "M32DEF.INC"
.EQU MYTEMP 0x100 ;save it here
LDI R16,0x00 ;R16 = 00000000 (binary)
OUT DDRC,R16 ;make Port C an input port (0 for In)
NOP
IN R16,PINC ;move from pins of Port C to R16
STS MYTEMP,R16 ;save it in MYTEMP

```

Port D

Port D occupies a total of 8 pins (PD0–PD7). To use the pins of Port D as input or output ports, each bit of the DDRD register must be set to the proper

value. For example, the following code will continuously send out to Port D the alternating values of 0x55 and 0xAA:

```
        ;toggle all bits of PORTB
.INCLUDE "M32DEF.INC"
        LDI    R16,0xFF      ;R16 = 11111111 (binary)
        OUT    DDRD,R16      ;make Port D an output port (1 for Out)
L1:     LDI    R16,0x55      ;R16 = 0x55
        OUT    PORTD,R16     ;put 0x55 on Port D pins
        CALL   DELAY
        LDI    R16,0xAA      ;R16 = 0xAA
        OUT    PORTD,R16     ;put 0xAA on Port D pins
        CALL   DELAY
        RJMP   L1
```

Port D as input

In order to make all the bits of Port D an input, DDRD must be cleared by writing 0 to all the bits. In the following code, Port D is configured first as an input port by writing all 0s to register DDRD, and then data is received from Port D and saved in a RAM location:

```
.INCLUDE "M32DEF.INC"
.EQU MYTEMP 0x100      ;save it here

        LDI    R16,0x00      ;R16 = 00000000 (binary)
        OUT    DDRD,R16      ;make Port D an input port (0 for In)
        NOP
        IN     R16,PIND       ;move from pins of Port D to R16
        STS    MYTEMP,R16    ;save it in MYTEMP
```

Dual role of Ports C and D

The alternate functions of the pins for Port C are shown in Table 4-5. We will show how to use Port C's alternate functions in future chapters. The alternate functions of the pins for Port D are shown in Table 4-6. We will show how to use Port D's alternate functions in future chapters.

Table 4-5: Port C Alternate Functions

Bit	Function
PC0	SCL
PC1	SDA
PC2	TCK
PC3	TMS
PC4	TDO
PC5	TDI
PC6	TOSC1
PC7	TOSC2

Table 4-6: Port D Alternate Functions

Bit	Function
PD0	PSP0/C1IN+
PD1	PSP1/C1IN-
PD2	PSP2/C2IN+
PD3	PSP3/C2IN-
PD4	PSP4/ECCP1/P1A
PD5	PSP5/P1B
PD6	PSP6/P1C
PD7	PSP7/P1D

Example 4-1

Write a test program for the AVR chip to toggle all the bits of PORTB, PORTC, and PORTD every 1/4 of a second. Assume a crystal frequency of 1 MHz.

Solution:

;tested with AVR Studio for the ATmega32 and XTAL = 1 MHz
;to select the XTAL frequency in AVR Studio, press ALT+O

```
.INCLUDE "M32DEF.INC"
    LDI    R16, HIGH(RAMEND)
    OUT    SPH, R16
    LDI    R16, LOW(RAMEND)
    OUT    SPL, R16    ;initialize stack pointer

    LDI    R16, 0xFF
    OUT    DDRB, R16    ;make Port B an output port
    OUT    DDRC, R16    ;make Port C an output port
    OUT    DDRD, R16    ;make Port D an output port

    LDI    R16, 0x55    ;R16 = 0x55
L3:   OUT    PORTB, R16    ;put 0x55 on Port B pins
    OUT    PORTC, R16    ;put 0x55 on Port C pins
    OUT    PORTD, R16    ;put 0x55 on Port D pins
    CALL   QDELAY        ;quarter of a second delay
    COM    R16            ;complement R16
    RJMP   L3

;-----1/4 SECOND DELAY
QDELAY:
    LDI    R21, 200
D1:   LDI    R22, 250
D2:   NOP
    NOP
    DEC    R22
    BRNE   D2
    DEC    R21
    BRNE   D1
    RET
```

Calculations:

$1 / 1 \text{ MHz} = 1 \mu\text{s}$

Delay = $200 \times 250 \times 5 \text{ MC} \times 1 \mu\text{s} = 250,000 \mu\text{s}$ (If we include the overhead, we will have 250,608 μs . See Example 3-18 in the previous chapter.)

Use the AVR Studio simulator to verify the delay size.

Review Questions

1. There are a total of _____ ports in the ATmega32.
2. True or false. All of the ATmega32 ports have 8 pins.
3. True or false. Upon power-up, the I/O pins are configured as output ports.
4. Code a simple program to send 0x99 to Port B and Port C.
5. To make Port B an output port, we must place _____ in register _____.
6. To make Port B an input port, we must place _____ in register _____.
7. True or false. We use a PORTx register to send data out to AVR pins.
8. True or false. We use PINx to bring data into the CPU from AVR pins.

SECTION 4.2: I/O BIT MANIPULATION PROGRAMMING

In this section we further examine the AVR I/O instructions. We pay special attention to I/O bit manipulation because it is a powerful and widely used feature of the AVR family.

I/O ports and bit-addressability

Sometimes we need to access only 1 or 2 bits of the port instead of the entire 8 bits. A powerful feature of AVR I/O ports is their capability to access individual bits of the port without altering the rest of the bits in that port. For all AVR ports, we can access either all 8 bits or any single bit without altering the rest. Table 4-7 lists the single-bit instructions for the AVR. Although the instructions in Table 4-7 can be used for any of the lower 32 I/O registers, I/O port operations use them most often. We will see the use of these instructions throughout future chapters. Table 4-8 shows the lower 32 I/O registers.

Next we describe all these instructions and examine their usage.

Table 4-7: Single-Bit (Bit-Oriented) Instructions for AVR

Instruction		Function
SBI	ioReg,bit	Set Bit in I/O register (set the bit: bit = 1)
CBI	ioReg,bit	Clear Bit in I/O register (clear the bit: bit = 0)
SBIC	ioReg,bit	Skip if Bit in I/O register Cleared (skip next instruction if bit = 0)
SBIS	ioReg,bit	Skip if Bit in I/O register Set (skip next instruction if bit = 1)

Address			Address			Address		
Mem.	I/O	Name	Mem.	I/O	Name	Mem.	I/O	Name
\$20	\$00	TWBR	\$2B	\$0B	UCSRA	\$36	\$16	PINB
\$21	\$01	TWSR	\$2C	\$0C	UDR	\$37	\$17	DDRB
\$22	\$02	TWAR	\$2D	\$0D	SPCR	\$38	\$18	PORTB
\$23	\$03	TWDR	\$2E	\$0E	SPSR	\$39	\$19	PINA
\$24	\$04	ADCL	\$2F	\$0F	SPDR	\$3A	\$1A	DDRA
\$25	\$05	ADCH	\$30	\$10	PIND	\$3B	\$1B	PORTA
\$26	\$06	ADCSRA	\$31	\$11	DDRD	\$3C	\$1C	EEDR
\$27	\$07	ADMUX	\$32	\$12	PORTD	\$3D	\$1D	EEDR
\$28	\$08	ACSR	\$33	\$13	PINC	\$3E	\$1E	EEARL
\$29	\$09	UBRR	\$34	\$14	DDRC	\$3F	\$1F	EEARH
\$2A	\$0A	UCSRB	\$35	\$15	PORTC			

Table 4-8: The Lower 32 I/O Registers

SBI (set bit in I/O register)

To set HIGH a single bit of a given I/O register, we use the following syntax:

```
SBI ioReg, bit_num
```

where `ioReg` can be the lower 32 I/O registers (addresses 0 to 31) and `bit_num` is the desired bit number from 0 to 7. In Table 4-8 you see the list of the lower 32 I/O registers. Although the bit-oriented instructions can be used for manipulation of bits D0–D7 of the lower 32 I/O registers, they are mostly used for I/O ports. For example the following instruction sets HIGH bit 5 of Port B:

```
SBI PORTB, 5
```

In Figure 4-6, you see the SBI instruction format.

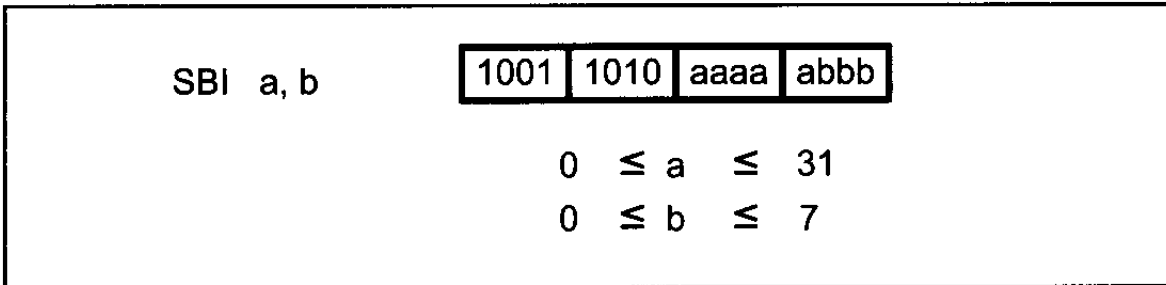


Figure 4-6. SBI (Set Bit) Instruction Format

CBI (Clear Bit in I/O register)

To clear a single bit of a given I/O register, we use the following syntax:

```
CBI ioReg, bit_number
```

For example, the following code toggles pin PB2 continuously:

```
SBI    DDRB, 2           ;bit = 1, make PB2 an output pin
AGAIN:SBI  PORTB, 2       ;bit set (PB2 = high)
CALL    DELAY
CBI    PORTB, 2           ;bit clear (PB2 = low)
CALL    DELAY
RJMP    AGAIN
```

Remember that for I/O ports, we must set the appropriate bit in the `DDRx` register if we want the pin to be output.

Notice that PB2 is the third bit of Port B (the first bit is PB0, the second bit is PB1, etc.). This is shown in Table 4-9. See Example 4-2 for an example of bit manipulation of I/O bits.

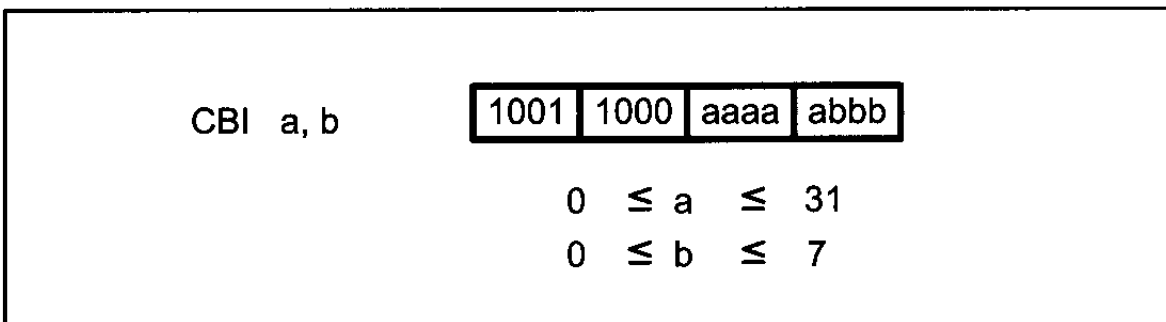


Figure 4-7. CBI (Clear Bit) Instruction Format

Table 4-9: Single-Bit Addressability of Ports for ATmega32/16

PORT	PORTB	PORTC	PORTD	Port Bit
PA0	PB0	PC0	PD0	D0
PA1	PB1	PC1	PD1	D1
PA2	PB2	PC2	PD2	D2
PA3	PB3	PC3	PD3	D3
PA4	PB4	PC4	PD4	D4
PA5	PB5	PC5	PD5	D5
PA6	PB6	PC6	PD6	D6
PA7	PB7	PC7	PD7	D7

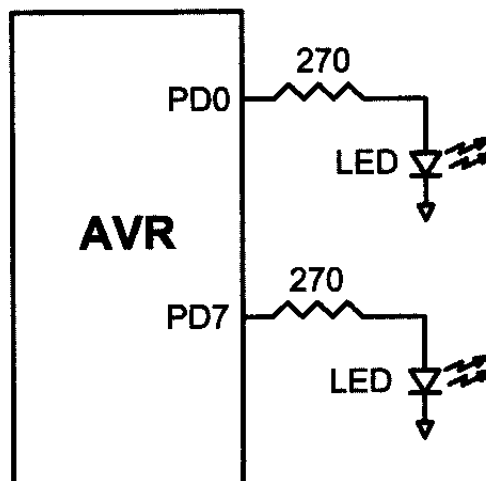
Notice in Example 4-2 that unused portions of Port C are undisturbed. This single-bit addressability of I/O ports is one of the most powerful features of the AVR microcontroller.

Example 4-2

An LED is connected to each pin of Port D. Write a program to turn on each LED from pin D0 to pin D7. Call a delay subroutine before turning on the next LED.

Solution:

```
.INCLUDE "M32DEF.INC"
LDI R20, HIGH(RAMEND)
OUT SPH, R20
LDI R20, LOW(RAMEND)
OUT SPL, R20 ;initialize stack pointer
LDI R20, 0xFF
OUT PORTD, R20 ;make PORTD an output port
SBI PORTD,0 ;set bit PD0
CALL DELAY ;delay before next one
SBI PORTD,1 ;turn on PD1
CALL DELAY ;delay before next one
SBI PORTD,2 ;turn on PD2
CALL DELAY
SBI PORTD,3
CALL DELAY
SBI PORTD,4
CALL DELAY
SBI PORTD,5
CALL DELAY
SBI PORTD,6
CALL DELAY
SBI PORTD,7
CALL DELAY
```



Example 4-3

Write the following programs:

- (a) Create a square wave of 50% duty cycle on bit 0 of Port C.
- (b) Create a square wave of 66% duty cycle on bit 3 of Port C.

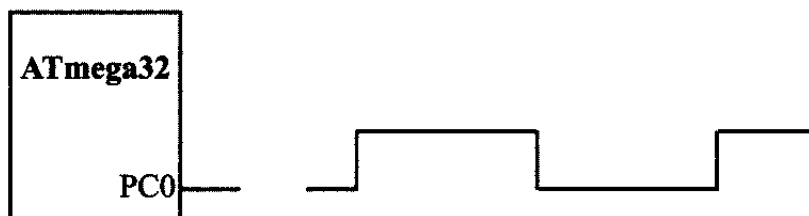
Solution:

- (a) The 50% duty cycle means that the “on” and “off” states (or the high and low portions of the pulse) have the same length. Therefore, we toggle PC0 with a time delay between each state.

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

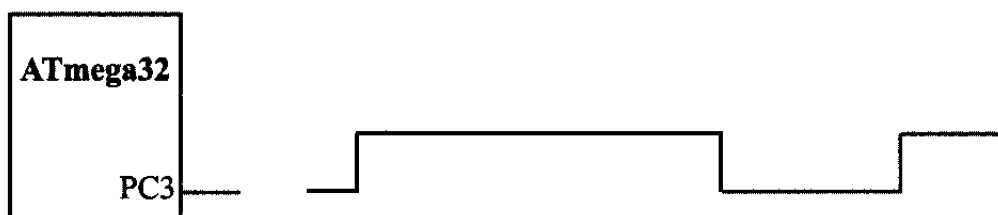
```
LDI    R20, HIGH(RAMEND)
OUT     SPH, R20
LDI     R20, LOW(RAMEND)
OUT     SPL, R20    ;initialize stack pointer

SBI     DDRC, 0      ;set bit 0 of DDRC (PC0 = out)
HERE: SBI     PORTC, 0 ;set to HIGH PC0 (PC0 = 1)
CALL    DELAY        ;call the delay subroutine
CBI     PORTC, 0      ;PC0 = 0
CALL    DELAY
RJMP    HERE         ;keep doing it
```



- (b) A 66% duty cycle means that the “on” state is twice the “off” state.

```
....
SBI     DDRC, 3      ;set bit 3 of DDRC (PC3 = out)
HERE: SBI     PORTC, 3 ;set to HIGH PC3 (PC3 = 1)
CALL    DELAY        ;call the delay subroutine
CALL    DELAY        ;call the delay subroutine
CBI     PORTC, 3      ;PC3 = 0
CALL    DELAY
RJMP    HERE         ;keep doing it
```



Checking an input pin

To make decisions based on the status of a given bit in the file register, we use the SBIC (Skip if Bit in I/O register Cleared) and SBIS (Skip if Bit in I/O register Set) instructions. These single-bit instructions are widely used for I/O operations. They allow you to monitor a single pin and make a decision depending on whether it is 0 or 1. Again it must be noted that the SBIC and SBIS instructions can be used for any bits of the lower 32 I/O registers, including the I/O ports A, B, C, D, and so on.

SBIS (Skip if Bit in I/O register Set)

To monitor the status of a single bit for HIGH, we use the SBIS instruction. This instruction tests the bit and skips the next instruction if it is HIGH. See Figure 4-8. Example 4-4 shows how it is used.

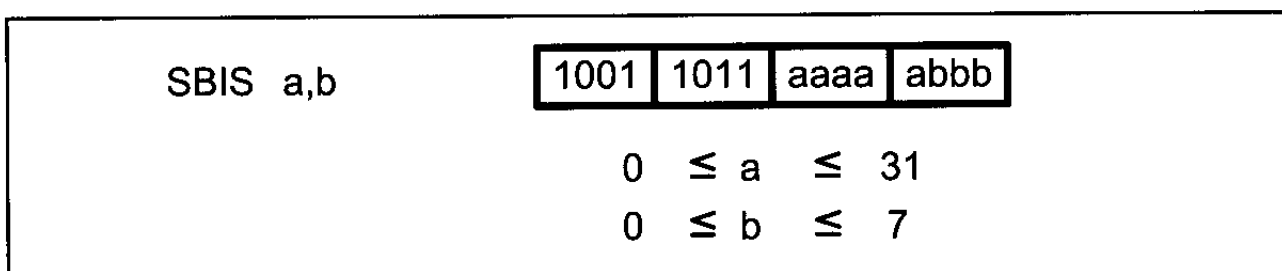


Figure 4-8. SBIS (Skip If Bit in I/O Register Set) Instruction Format

Example 4-4

Write a program to perform the following:

- (a) Keep monitoring the PB2 bit until it becomes HIGH;
- (b) When PB2 becomes HIGH, write the value \$45 to Port C, and also send a HIGH-to-LOW pulse to PD3.

Solution:

```
.INCLUDE "M32DEF.INC"

        CBI    DDRB, 2    ;make PB2 an input
        LDI    R16, 0xFF
        OUT    DDRC, R16  ;make Port C an output port
        SBI    DDRD, 3    ;make PD3 an output
AGAIN:   SBIS   PINB, 2    ;skip if Bit PB2 is HIGH
        RJMP   AGAIN      ;keep checking if LOW
        LDI    R16, 0x45
        OUT    PORTC, R16 ;write 0x45 to port C
        SBI    PORTD, 3    ;set bit PD3 (H-to-L)
        CBI    PORTD, 3    ;clear bit PD3
HERE:    RJMP   HERE
```

In this program, "SBIS PINB, 2" instruction stays in the loop as long as PB2 is LOW. When PB2 becomes HIGH, it skips the branch instruction to get out of the loop, and writes the value \$45 to Port C. It also sends a HIGH-to-LOW pulse to PD3.

SBIC (Skip if Bit in I/O register Cleared)

To monitor the status of a single bit for LOW, we use the SBIC instruction. This instruction tests the bit and skips the instruction right below it if the bit is LOW. See Figure 4-9. Example 4-5 shows how it is used.

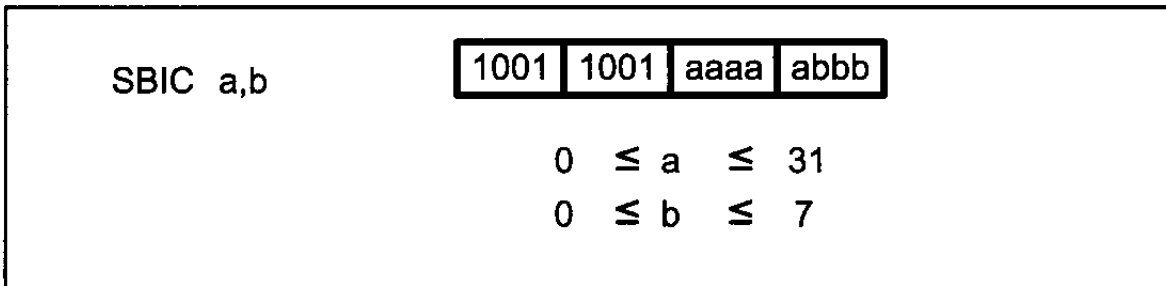


Figure 4-9. SBIC (Skip if Bit in I/O Register Cleared) Instruction Format

Monitoring a single bit

We can also use the bit test instructions to monitor the status of a single bit and make a decision to perform an action. See Examples 4-6 and 4-7.

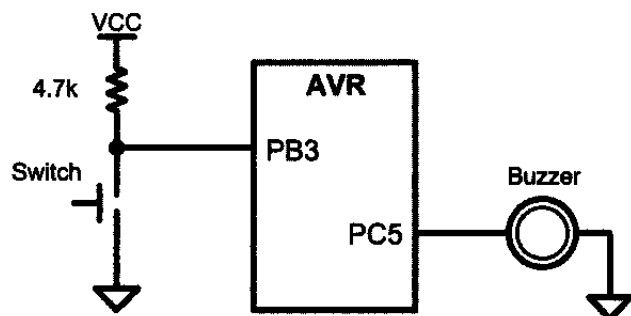
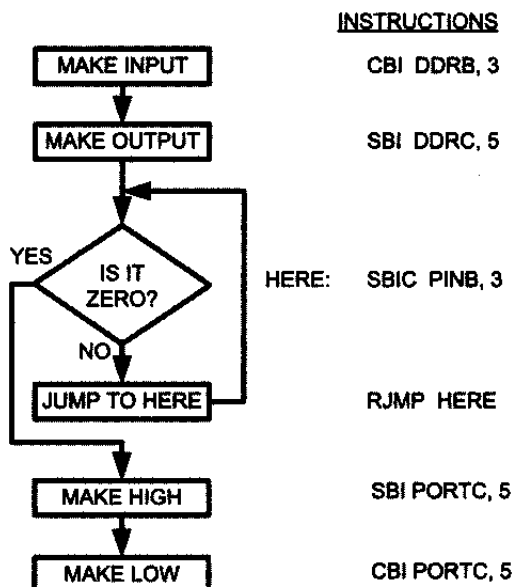
Example 4-5

Assume that bit PB3 is an input and represents the condition of a door alarm. If it goes LOW, it means that the door is open. Monitor the bit continuously. Whenever it goes LOW, send a HIGH-to-LOW pulse to port PC5 to turn on a buzzer.

Solution:

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

```
      CBI   DDRB, 3      ;make PB3 an input
      SBI   DDRC, 5      ;make PC5 an output
HERE: SBIC  PINB, 3      ;keep monitoring PB3 for HIGH
      RJMP  HERE         ;stay in the loop
      SBI   PORTC,5      ;make PC5 HIGH
      CBI   PORTC,5      ;make PC5 LOW for H-to-L
      RJMP  HERE
```



Example 4-6

A switch is connected to pin PB2. Write a program to check the status of SW and perform the following:

(a) If SW = 0, send the letter 'N' to PORTD.

(b) If SW = 1, send the letter 'Y' to PORTD.

Solution:

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

```
CBI DDRB, 2 ;make PB2 an input
```

```
LDI R16, 0xFF
```

```
OUT DDRD, R16 ;make PORTD an output port
```

```
AGAIN: SBIS PINB, 2 ;skip next if PB bit is HIGH
```

```
RJMP OVER ;SW is LOW
```

```
LDI R16, 'Y' ;R16 = 'Y' (ASCII letter Y)
```

```
OUT PORTD, R16 ;PORTD = 'Y'
```

```
RJMP AGAIN
```

```
OVER: LDI R16, 'N' ;R16 = 'N' (ASCII letter Y)
```

```
OUT PORTD, R16 ;PORTD = 'N'
```

```
RJMP AGAIN
```

INSTRUCTIONS

```
CBI DDRB, 2
```

```
LDI R16, 0xFF  
OUT DDRD, R16
```

```
AGAIN: SBIS PINB, 2
```

```
RJMP OVER
```

```
LDI R16, 'Y'
```

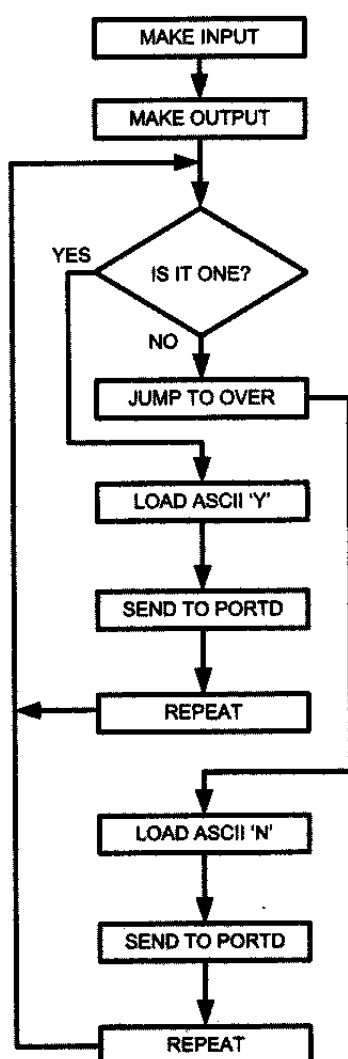
```
OUT PORTD, R16
```

```
RJMP AGAIN
```

```
OVER: LDI R16, 'N'
```

```
OUT PORTD, R16
```

```
RJMP AGAIN
```



Example 4-7

Rewrite the program of Example 4-6, using the SBIC instruction instead of SBIS.

Solution:

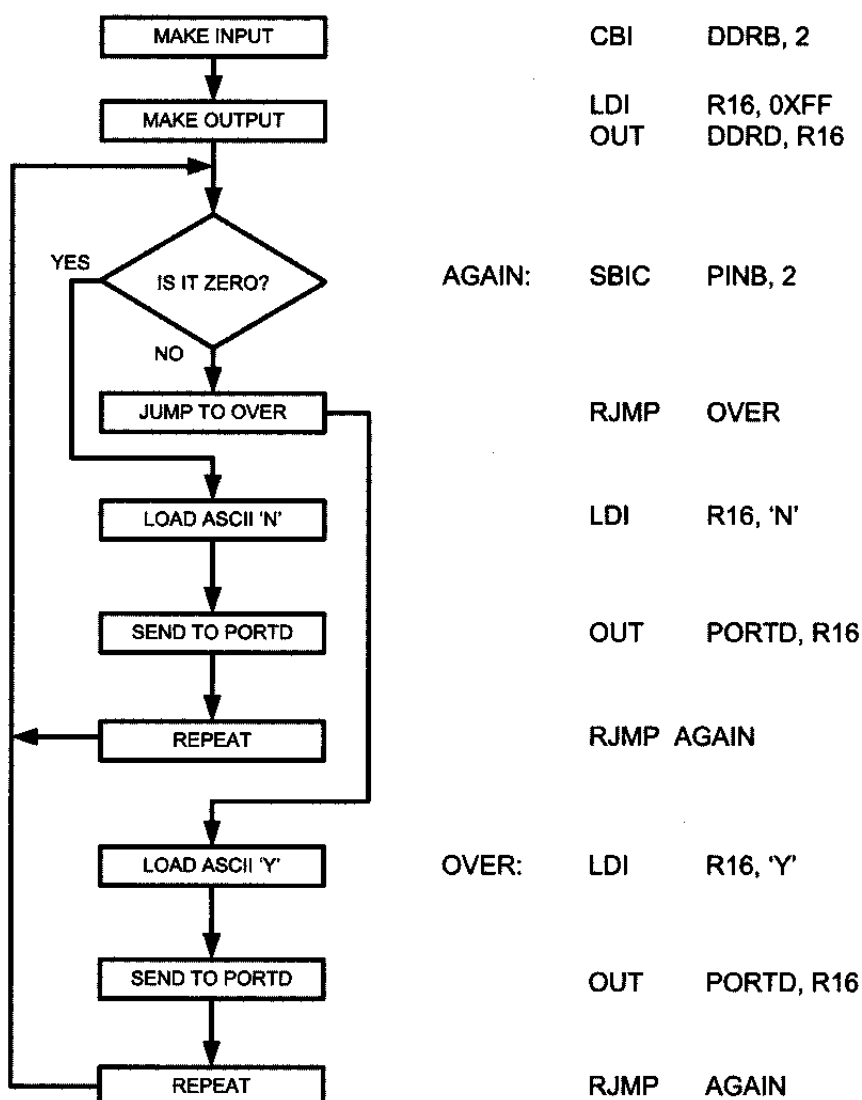
```
.INCLUDE "M32DEF.INC"

CBI    DDRB, 2      ;make PB2 an input
LDI    R16, 0xFF
OUT    DDRD, R16    ;make PORTD an output port

AGAIN: SBIC PINB, 2  ;skip next if PB bit is LOW
      RJMP OVER      ;SW is HIGH
      LDI R16, 'N'    ;R16 = 'N' (ASCII letter N)
      OUT PORTD, R16  ;PORTD = 'N'
      RJMP AGAIN

OVER:  LDI R16, 'Y'    ;R16 = 'Y' (ASCII letter Y)
      OUT PORTD, R16  ;PORTD = 'Y'
      RJMP AGAIN
```

INSTRUCTIONS



Reading a single bit

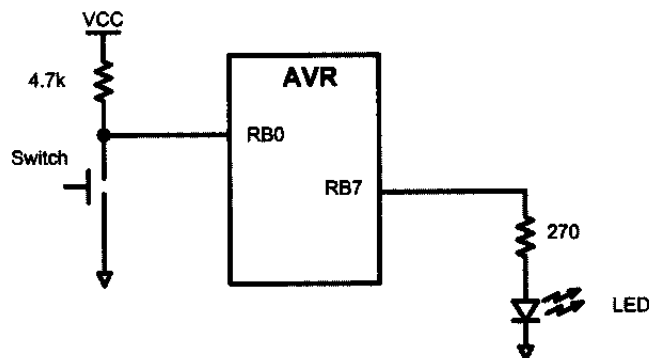
We can also use the bit test instructions to read the status of a single bit and send it to another bit or save it. This is shown in Examples 4-8 and 4-9.

Example 4-8

A switch is connected to pin PB0 and an LED to pin PB7. Write a program to get the status of SW and send it to the LED.

Solution:

```
.INCLUDE "M32DEF.INC"
    CBI    DDRB, 0      ;make PB0 an input
    SBI    DDRB, 7      ;make PB7 an output
AGAIN:SBIC PINB, 0      ;skip next if PB0 is clear
    RJMP   OVER         ;(JMP is OK too)
    CBI    PORTB, 7
    RJMP   AGAIN        ;we can use JMP too
OVER: SBI    PORTB, 7
    RJMP   AGAIN        ;we can use JMP too
```

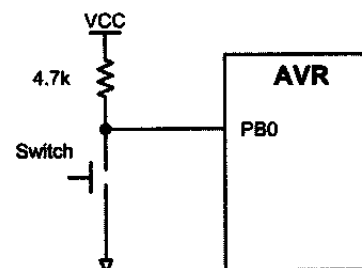


Example 4-9

A switch is connected to pin PB0. Write a program to get the status of SW and save it in location 0x200.

Solution:

```
.EQU MYTEMP = 0x200      ;set aside location 0x200
.INCLUDE "M32DEF.INC"
    CBI    DDRB, 0      ;make PB0 an input
AGAIN:SBIC PINB, 0      ;skip next if PB0 is clear
    RJMP   OVER         ;(JMP is OK too)
    LDI    R16, 0
    STS    MYTEMP,R16   ;save it in MYTEMP
    RJMP   AGAIN        ;we can use JMP too
OVER: LDI    R16,0x1     ;move 1 to R16
    STS    MYTEMP,R16   ;save it in MYTEMP
    RJMP   AGAIN        ;we can use JMP too
```



Review Questions

1. True or false. The instruction “SBI PORTB, 1” makes pin PB1 HIGH while leaving other pins of PORTB unchanged, if bit 1 of the DDR bits is configured for output.
2. Show one way to toggle the pin PB7 continuously using AVR instructions.
3. Write instructions to get the status of PB2 and put it on PB0.
4. Write instructions to toggle both bits of PD7 and PD0 continuously.
5. According to Figure 4-7, what does the machine instruction \$9819 do?

SUMMARY

This chapter focused on the I/O ports of the AVR. The four ports of the ATmega32, PORTA, PORTB, PORTC, and PORTD, were explored. These ports can be used for input or output. All the ports have alternate functions. The three registers associated with each port are PORTx, DDRx, and PINx. Their role in I/O manipulation was examined. Then, I/O instructions of the AVR were explained, and numerous examples were given. We also showed the bit-addressability of AVR ports.

CAUTION

We strongly recommend that you study Section C.2 (Appendix C) before connecting any external hardware to your AVR system. Failure to use the right instruction or the right connection to port pins can damage the ports of your AVR chip.

PROBLEMS

SECTION 4.1: I/O PORT PROGRAMMING IN AVR

1. The ATmega32 has a DIP package of ____ pins.
2. In ATmega32, how many pins are assigned to V_{CC} and GND?
3. In the ATmega32, how many pins are designated as I/O port pins?
4. How many pins are designated as PORTA in the 40-pin DIP package and what are their numbers?
5. How many pins are designated as PORTB in the 40-pin DIP package and what are their numbers?
6. How many pins are designated as PORTC in the 40-pin DIP package and what are their numbers?
7. How many pins are designated as PORTD in the 40-pin DIP package and what are their numbers?
8. Upon reset, all the bits of ports are configured as ____ (input, output).
9. Explain the role of DDRx and PORTx in I/O operations.

10. Write a program to get 8-bit data from PORTC and send it to PORTB and PORTD.
11. Write a program to get 8-bit data from PORTD and send it to PORTB and PORTC.
12. Which pins are for RxD and TxD?
13. Give data memory location assigned to DDR registers of Ports A–C for the ATmega32.
14. Write a program to toggle all the bits of PORTB and PORTC continuously
(a) using 0xAA and 0x55 (b) using the COM instruction.

SECTION 4.2: I/O BIT MANIPULATION PROGRAMMING

15. Which ports of the ATmega32 are bit-addressable?
16. What is the advantage of bit-addressability for AVR ports?
17. Is the instruction “COM PORTB” a valid instruction?
18. Write a program to toggle PB2 and PB5 continuously without disturbing the rest of the bits.
19. Write a program to toggle PD3, PD7, and PC5 continuously without disturbing the rest of the bits.
20. Write a program to monitor bit PC3. When it is HIGH, send 0x55 to PORTD.
21. Write a program to monitor the PB7 bit. When it is LOW, send \$55 and \$AA to PORTC continuously.
22. Write a program to monitor the PA0 bit. When it is HIGH, send \$99 to PORTB. If it is LOW, send \$66 to PORTB.
23. Write a program to monitor the PB5 bit. When it is HIGH, make a LOW-to-HIGH-to-LOW pulse on PB3.
24. Write a program to get the status of PC3 and put it on PC4.
25. Create a flowchart and write a program to get the statuses of PD6 and PD7 and put them on PC0 and PC7, respectively.
26. Write a program to monitor the PB5 and PB6 bits. When both of them are HIGH, send \$AA to PORTC; otherwise, send \$55 to PORTC.
27. Write a program to monitor the PB5 and PB6 bits. When either of them is HIGH, send \$AA to PORTC; otherwise, send \$55 to PORTC.
28. Referring to Figure 4-8 and Table 4-8, write the machine equivalent of “SBIS PINB,3”.
29. Referring to Figure 4-6 and Table 4-8, write the machine equivalent of the “SBI PORTA,2” instruction.

ANSWERS TO REVIEW QUESTIONS

SECTION 4.1: I/O PORT PROGRAMMING IN AVR

1. 4
2. True
3. False
4.

```
LDI R16,0xFF
OUT DDRB,R16
OUT DDRC,R16
LDI R16,0x99
OUT PORTB,R16
OUT PORTC,R16
```
5. \$FF, DDRB
6. \$00, DDRB
7. True
8. True

SECTION 4.2: I/O BIT MANIPULATION PROGRAMMING

1. True
2.

```
CBI DDRB,7
H1:   SBI PORTB,7
      CBI PORTB,7
      RJMP H1
```
3.

```
CBI   DDRB,2
SBI   DDRB,0
AGAIN: SBIS PINB,2
      RJMP OVER
SBI   PORTB,0
RJMP  AGAIN
OVER: CBI   PORTB,0
      RJMP  AGAIN
```
4.

```
SBI   DDRD,0
SBI   DDRD,7
H2:   SBI   PORTD,0
      SBI   PORTD,7
      CBI   PORTD,0
      CBI   PORTD,7
      RJMP  H2
```
5. \$9819 is 1001 1000 0001 1001 in binary; according to Figure 4-7, this is the CBI instruction, where a = 00011 = 3 and b = 001 = 1. According to Table 4-8, 3 is the I/O address of TWDR; thus, this is the “CBI TWDR,1” instruction and clears bit 1 of the TWDR register.

CHAPTER 5

ARITHMETIC, LOGIC INSTRUCTIONS, AND PROGRAMS

OBJECTIVES

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

- >> Define the range of numbers possible in AVR unsigned data
- >> Code addition and subtraction instructions for unsigned data
- >> Code AVR multiplication instructions
- >> Code AVR programs for division
- >> Code AVR Assembly language logic instructions AND, OR, and EX-OR
- >> Use AVR logic instructions for bit manipulation
- >> Use compare instructions for program control
- >> Code conditional branch instructions
- >> Code AVR rotate instructions and data serialization
- >> Contrast and compare packed and unpacked BCD data
- >> Code AVR programs for ASCII and BCD data conversion

This chapter describes AVR arithmetic and logic instructions. Program examples are given to illustrate the application of these instructions. In Section 5.1 we discuss instructions and programs related to addition, subtraction, multiplication, and division of unsigned numbers. Signed numbers are described in Section 5.2. In Section 5.3, we discuss the logic instructions AND, OR, and XOR, as well as the compare instruction. The rotate and shift instructions and data serialization are explained in Section 5.4. In Section 5.5 we introduce BCD and ASCII conversion.

SECTION 5.1: ARITHMETIC INSTRUCTIONS

Unsigned numbers are defined as data in which all the bits are used to represent data and no bits are set aside for the positive or negative sign. This means that the operand can be between 00 and FFH (0 to 255 decimal) for 8-bit data.

Addition of unsigned numbers

In the AVR, the add operation has two general purpose registers as inputs and the result will be stored in the first (left) register. One form of the ADD instruction in the AVR is:

```
ADD  Rd,Rr    ;Rd = Rd + Rr
```

The instruction adds Rr (resource) to Rd (destination) and stores the result in Rd. It could change any of the Z, C, N, V, H or S bits of the status register, depending on the operands involved. The effect of the ADD instruction on N and V is discussed in Section 5.2 because these bits are relevant mainly in signed number operations. Look at Example 5-1.

Notice that none of the AVR addition instructions support direct memory access; that is, we cannot add a memory location to another memory location or register. To add a memory location we should first load it to any of the R0–R31 registers and then use the ADD operation on it. Look at Example 5-2.

Example 5-1

Show how the flag register is affected by the following instructions.

```
LDI    R21,0xF5    ;R21 = F5H
LDI    R22,0x0B    ;R22 = 0x0BH
ADD     R21,R22     ;R21 = R21+R22 = F5+0B = 00 and C = 1
```

Solution:

F5H	1111 0101
+ 0BH	+ 0000 1011
100H	0000 0000

After the addition, register R21 contains 00 and the flags are as follows:

C = 1 because there is a carry out from D7.

Z = 1 because the result in destination register (R21) is zero.

H = 1 because there is a carry from D3 to D4.

Example 5-2

Assume that RAM location 400H has the value of 33H. Write a program to find the sum of location 400H of RAM and 55H. At the end of the program, R21 should contain the sum.

Solution:

```
LDS    R2,0x400    ;R2 = 33H (location 0x400 of RAM)
LDI    R21,0x55     ;R21 = 55
ADD    R21,R2       ;R21 = R21 + R2 = 55H + 33H = 88H, C = 0
```

ADC and addition of 16-bit numbers

When adding two 16-bit data operands, we need to be concerned with the propagation of a carry from the lower byte to the higher byte. This is called *multi-byte addition* to distinguish it from the addition of individual bytes. The instruction ADC (ADD with carry) is used on such occasions.

For example, look at the addition of 3CE7H + 3B8DH, as shown next.

$$\begin{array}{r}
 1 \\
 3C \ E7 \\
 + \quad 3B \ 8D \\
 \hline
 78 \ 74
 \end{array}$$

When the first byte is added, there is a carry ($E7 + 8D = 74$, $C = 1$). The carry is propagated to the higher byte, which results in $3C + 3B + 1 = 78$ (all in hex). Example 5-3 shows the above steps in an AVR program.

Example 5-3

Write a program to add two 16-bit numbers. The numbers are 3CE7H and 3B8DH. Assume that $R1 = 8D$, $R2 = 3B$, $R3 = E7$, and $R4 = 3C$. Place the sum in R3 and R4; R3 should have the lower byte.

Solution:

```
;R1 = 8D
;R2 = 3B
;R3 = E7
;R4 = 3C

ADD    R3,R1        ;R3 = R3 + R1 = E7 + 8D = 74 and C = 1
ADC    R4,R2        ;R4 = R4 + R2 + carry, adding the upper byte
                        ;with carry from lower byte
                        ;R4 = 3C + 3B + 1 = 78H (all in hex)
```

Notice the use of ADD for the lower byte and ADC for the higher byte.

Subtraction of unsigned numbers

In many microprocessors, there are two different instructions for subtraction: SUB and SUBB (subtract with borrow). In the AVR we have five instructions for subtraction: SUB, SBC, SUBI, SBCI, and SBIW. Figure 5-1 shows a summary of each instruction.

SUB	Rd, Rr	; Rd=Rd-Rr
SBC	Rd, Rr	; Rd=Rd-Rr-c
SUBI	Rd, K	; Rd=Rd-K
SBCI	Rd, K	; Rd=Rd-K-c
SBIW	Rd:Rd+1, K	; Rd+1:Rd=Rd+1:Rd-K

Figure 5-1.

The SBC and SBCI instructions are subtract with borrow. In the AVR, we use the C (carry) flag for the borrow and that is why they are called SBC (SuB with Carry). In this section we will examine some of these commands.

SUB Rd, Rr (Rd = Rd - Rr)

In subtraction, the AVR microcontrollers (indeed, all modern CPUs) use the 2's complement method. Although every CPU contains adder circuitry, it would be too cumbersome (and take too many transistors) to design separate subtractor circuitry. For this reason, the AVR uses adder circuitry to perform the subtraction command. Assuming that the AVR is executing a simple subtract instruction and that C = 0 prior to the execution of the instruction, one can summarize the steps of the hardware of the CPU in executing the SUB instruction for unsigned numbers as follows:

1. Take the 2's complement of the subtrahend (right-hand operand).
2. Add it to the minuend (left-hand operand).
3. Invert the carry.

Example 5-4

Show the steps involved in the following.

```
LDI    R20, 0x23      ;load 23H into R20
LDI    R21, 0x3F      ;load 3FH into R21
SUB     R21, R20       ;R21 <- R21-R20
```

Solution:

```

R21 = 3F  0011 1111      0011 1111
- R20 = 23  0010 0011    + 1101 1101 (2's complement)
    1C                                1 0001 1100
                                C = 0, D7 = N = 0 (result is positive)
```

The flags would be set as follows: N = 0, C = 0. (Notice that there is a carry but C = 0. We will discuss this more in the next section.) The programmer must look at the N (or C) flag to determine if the result is positive or negative.

These two steps are performed for every SUB instruction by the internal hardware of the CPU, regardless of the source of the operands, provided that the addressing mode is supported. It is after these two steps that the result is obtained and the flags are set. Example 5-4 illustrates the two steps.

After the execution of the SUB instruction, if $N = 0$ (or $C = 0$), the result is positive; if $N = 1$ (or $C = 1$), the result is negative and the destination has the 2's complement of the result. Normally, the result is left in 2's complement, but the NEG (negate, which is 2's complement) instruction can be used to change it. The other subtraction instructions for subtract are SUBI and SBIW, which subtract an immediate (constant) value from a register. SBIW subtracts an immediate value in the range of 0–63 from a register pair and stores the result in the register pair. Notice that only the last eight registers can be used with SBIW. See Examples 5-5 and 5-6.

Example 5-5

Write a program to subtract 18H from 29H and store the result in R21 (a) without using the SUBI instruction, and (b) using the SUBI instruction.

Solution:

(a)

```
LDI    R21,0x29    ;R21 = 29H
LDI    R22,0x18    ;R22 = 18H
SUB     R21,R22     ;R21 = R21 - R22 = 29 - 18 = 11 H
```

(b)

```
LDI    R21,0x29    ;R21 = 29H
SUBI    R21,0x18    ;R21 = R21 - 18 = 29 - 18 = 11 H
```

Example 5-6

Write a program to subtract 18H from 2917H and store the result in R25 and R24.

Solution:

```
LDI    R25,0x29    ;load the high byte (R25 = 29H)
LDI    R24,0x17    ;load the low byte (R24 = 17H)
SBIW   R25:R24,0x18 ;R25:R24 <- R25:R24 - 0x18
                        ;28FF = 2917 - 18
```

Notice that you should use SBIW $Rd+1:Rd,K$ format. If SBIW $Rd:Rd+1,K$ format is used, the assembler will assemble your code as if you had typed SBIW $Rd+1:Rd,K$. Change the third line of the code from SBIW $R25:R24,0x18$ to SBIW $R24:R25,0x18$ and examine the result.

SBC ($Rd \leftarrow Rd - Rr - C$) subtract with borrow (denoted by C)

This instruction is used for multibyte numbers and will take care of the borrow of the lower byte. If the borrow flag is set to one ($C = 1$) prior to executing the SBC instruction, this operation also subtracts 1 from the result. See Example 5-7.

Example 5-7

Write a program to subtract two 16-bit numbers: $2762H - 1296H$. Assume $R26 = (62)$ and $R27 = (27)$. Place the difference in $R26$ and $R27$; $R26$ should have the lower byte.

Solution:

```
;R26 = (62)
;R27 = (27)

LDI    R28,0x96    ;load the low byte (R28 = 96H)
LDI    R29,0x12    ;load the high byte (R29 = 12H)
SUB     R26,R28     ;R26 = R26 - R28 = 62 - 96 = CCH
                     ;C = borrow = 1, N = 1
SBC     R27,R29     ;R27 = R27 - R29 - C
                     ;R27 = 27 - 12 - 1 = 14H
```

After the SUB, $R26$ has $62H - 96H = CCH$ and the carry flag is set to 1, indicating there is a borrow (notice, $N = 1$). Because $C = 1$, when SBC is executed $R27$ has $27H - 12H - 1 = 14H$. Therefore, we have $2762H - 1296H = 14CCH$.

The C flag in subtraction for AVR

Notice that the AVR is like other CPUs such as the x86 and the 8051 when it comes to the carry flag in subtract operations. In the AVR, after subtract operations, the carry is inverted by the CPU itself and we examine the C flag to see if the result is positive or negative. This means that, after subtract operations, if $C = 1$, the result is negative, and if $C = 0$, the result is positive. If you study Example 5-4 again, you will see that there was a carry from MSB, but $C = 0$. Now you know the reason; it is because the CPU inverts the carry flag after the SUB instruction. Notice that the CPU does not invert the carry flag after the ADD instruction.

Multiplication of unsigned numbers

The AVR has several instructions dedicated to multiplication. Here we will discuss the MUL instruction. Other instructions are similar to MUL but are used for signed numbers. See Table 5-1.

MUL is a byte-by-byte multiply instruction. In byte-by-byte multiplication, operands must be in registers. After multiplication, the 16-bit unsigned product is placed in $R1$ (high byte) and $R0$ (low byte). Notice that if any of the operands is selected from $R0$ or $R1$ the result will overwrite those registers after multiplication.

Table 5-1: Multiplication Summary

Multiplication	Application	Byte1	Byte2	High byte of result	Low byte of result
MUL Rd, Rr	Unsigned numbers	Rd	Rr	R1	R0
MULS Rd, Rr	Signed numbers	Rd	Rr	R1	R0
MULSU Rd, Rr	Unsigned numbers with signed numbers	Rd	Rr	R1	R0

The following example multiplies 25H by 65H.

```

LDI    R23,0x25    ;load 25H to R23
LDI    R24,0x65    ;load 65H to R24
MUL    R23,R24     ;25H * 65H = E99 where
                    ;R1 = 0EH and R0 = 99H

```

Division of unsigned numbers

AVR has no instruction for divide operation. We can write a program to perform division by repeated subtraction. In dividing a byte by a byte, the numerator is placed in a register and the denominator is subtracted from it repeatedly. The quotient is the number of times we subtracted and the remainder is in the register upon completion. See Program 5-1.

```

.DEF    NUM = R20
.DEF    DENOMINATOR = R21
.DEF    QUOTIENT = R22

        LDI    NUM,95            ;NUM = 95
        LDI    DENOMINATOR,10    ;DENOMINATOR = 10
        CLR    QUOTIENT          ;QUOTIENT = 0

L1:      INC    QUOTIENT
        SUB    NUM, DENOMINATOR
        BRCC   L1                ;branch if C is zero

        DEC    QUOTIENT          ;once too many
        ADD    NUM, DENOMINATOR  ;add back to it

HERE:    JMP    HERE              ;stay here forever

```

Program 5-1: Divide Function

An application for division

Sometimes a sensor is connected to an ADC (analog-to-digital converter) and the ADC represents some quantity such as temperature or pressure. The 8-bit ADC provides data in hex in the range of 00–FFH. This hex data must be converted to decimal. We do that by dividing it by 10 repeatedly, saving the remainders, as shown in Examples 5-8 and 5-9.

Example 5-8

Assume that the data memory location 0x315 has value FD (hex). Write a program to convert it to decimal. Save the digits in locations 0x322, 0x323, and 0x324, where the least-significant digit is in location 0x322.

Solution:

```
.EQU HEX_NUM = 0x315

.EQU RMND_L = 0x322
.EQU RMND_M = 0x323
.EQU RMND_H = 0x324

.DEF NUM = R20
.DEF DENOMINATOR = R21
.DEF QUOTIENT = R22

        LDI    R16,0xFD                ;$FD = 253 in decimal
        STS    HEX_NUM,R16             ;store $FD in location 0x315

;=====

        LDS    NUM, HEX_NUM
        LDI    DENOMINATOR,10          ;DENOMINATOR = 10

L1:      INC    QUOTIENT                ;
        SUB    NUM, DENOMINATOR;
        BRCC   L1                      ;if C = 0 go back

        DEC    QUOTIENT                ;once too many
        ADD    NUM, DENOMINATOR        ;add back to it
        STS    RMND_L, NUM             ;store remainder as the 1st digit

        MOV    NUM, QUOTIENT
        LDI    QUOTIENT,0

L2:      INC    QUOTIENT
        SUB    NUM, DENOMINATOR
        BRCC   L2

        DEC    QUOTIENT                ;once too many
        ADD    NUM, DENOMINATOR        ;add back to it
        STS    RMND_M, NUM             ;store remainder as the 2nd digit

        STS    RMND_H, QUOTIENT        ;store quotient as the 3rd digit

HERE:    JMP    HERE                   ;stay here forever
```

To convert a single decimal digit to ASCII format, we OR it with 30H. See Section 5.5.

Example 5-9

Analyze the program in Example 5-8 for a numerator of 253.

Solution:

To convert a binary (hex) value to decimal, we divide it by 10 repeatedly until the quotient is less than 10. After each division the remainder is saved. In the case of an 8-bit binary, such as FDH, we have 253 decimal, as shown below.

	<i>Quotient</i>	<i>Remainder</i>
253/10 =	25	3 (low digit)
25/10 =	2	5 (middle digit)
		2 (high digit)

Therefore, we have FDH = 253.

Review Questions

1. In unsigned byte-by-byte multiplication, the product will be placed in register(s) _____.
2. Is "MUL R2, 0x10" a valid AVR instruction? Explain your answer.
3. In AVR, the largest two numbers that can be multiplied are _____ and _____.
4. True or false. The MUL instruction works on R0 and R1 only.
5. The instruction "ADD R20, R21" places the sum in _____.
6. Why is the following ADD instruction illegal? "ADD R1, 0x04"
7. Rewrite the instruction above in correct format.
8. The instruction "SUB R1, R2" places the result in _____.
9. Find the value of the C flags in each of the following.
 - (a) LDI R21 0x4F
LDI R22 0xB1
ADD R21, R22
 - (b) LDI R21, 0x9C
LDI R22, 0x63
ADD R21, R22
10. Show how the CPU would subtract 05H from 43H.
11. If C = 1, R1 = 95H, and R2 = 4FH prior to the execution of "SBC R1, R2", what will be the contents of R1 and C after the subtraction?

SECTION 5.2: SIGNED NUMBER CONCEPTS AND ARITHMETIC OPERATIONS

All data items used so far have been unsigned numbers, meaning that the entire 8-bit operand was used for the magnitude. Many applications require signed data. In this section the concept of signed numbers is discussed along with related instructions. If your applications do not involve signed numbers, you can bypass this section.

Concept of signed numbers in computers

In everyday life, numbers are used that could be positive or negative. For example, a temperature of 5 degrees below zero can be represented as -5 , and 20 degrees above zero as $+20$. Computers must be able to accommodate such numbers. To do that, computer scientists have devised the following arrangement for the representation of signed positive and negative numbers: The most significant bit (MSB) is set aside for the sign (+ or $-$), while the rest of the bits are used for the magnitude. The sign is represented by 0 for positive (+) numbers and 1 for negative ($-$) numbers. Signed byte representation is discussed below.

Signed 8-bit operands

In signed byte operands, D7 (MSB) is the sign, and D0 to D6 are set aside for the magnitude of the number. If $D7 = 0$, the operand is positive, and if $D7 = 1$, it is negative. The N flag in the status register is the D7 bit.

Positive numbers

The range of positive numbers that can be represented by the format shown in Figure 5-2 is 0 to $+127$. If a positive number is larger than $+127$, a 16-bit operand must be used.

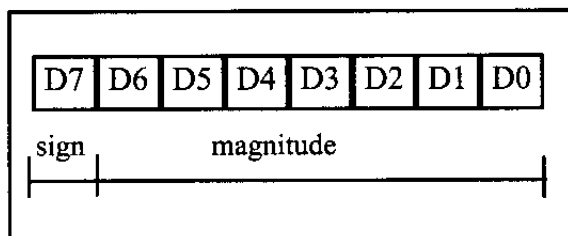


Figure 5-2. 8-Bit Signed Operand

Negative numbers

For negative numbers, D7 is 1; however, the magnitude is represented in its 2's complement. Although the assembler does the conversion, it is still important to understand how the conversion works. To convert to negative number representation (2's complement), follow these steps:

1. Write the magnitude of the number in 8-bit binary (no sign).
2. Invert each bit.
3. Add 1 to it.

Examples 5-10, 5-11, and 5-12 on the next page demonstrate these three steps.

Example 5-10

Show how the AVR would represent -5.

Solution:

Observe the following steps.

- | | | |
|----|-----------|---------------------------------|
| 1. | 0000 0101 | 5 in 8-bit binary |
| 2. | 1111 1010 | invert each bit |
| 3 | 1111 1011 | add 1 (which becomes FB in hex) |

Therefore, -5 = FBH, the signed number representation in 2's complement for -5. The D7 = N = 1 indicates that the number is negative.

Example 5-11

Show how the AVR would represent -34H.

Solution:

Observe the following steps.

- | | | |
|----|-----------|----------------------------|
| 1. | 0011 0100 | 34H given in binary |
| 2. | 1100 1011 | invert each bit |
| 3 | 1100 1100 | add 1 (which is CC in hex) |

Therefore, -34 = CCH, the signed number representation in 2's complement for 34H. The D7 = N = 1 indicates that the number is negative.

Example 5-12

Show how the AVR would represent -128.

Solution:

Observe the following steps.

- | | | |
|----|-----------|---------------------------------|
| 1. | 1000 0000 | 128 in 8-bit binary |
| 2. | 0111 1111 | invert each bit |
| 3 | 1000 0000 | add 1 (which becomes 80 in hex) |

Therefore, -128 = 80H, the signed number representation in 2's complement for -128. The D7 = N = 1 indicates that the number is negative. Notice that 128 (binary 10000000) in unsigned representation is the same as signed -128 (binary 10000000).

From the examples above, it is clear that the range of byte-sized negative numbers is -1 to -128. The following lists byte-sized signed number ranges:

Decimal	Binary	Hex
-128	1000 0000	80
-127	1000 0001	81
-126	1000 0010	82
...
-2	1111 1110	FE
-1	1111 1111	FF
0	0000 0000	00
+1	0000 0001	01
+2	0000 0010	02
..
+127	0111 1111	7F

Overflow problem in signed number operations

When using signed numbers, a serious problem sometimes arises that must be dealt with. This is the overflow problem. The AVR indicates the existence of an error by raising the V (overflow) flag, but it is up to the programmer to take care of the erroneous result. The CPU understands only 0s and 1s and ignores the human convention of positive and negative numbers. What is an overflow? If the result of an operation on signed numbers is too large for the register, an overflow has occurred and the programmer must be notified. Look at Example 5-13.

Example 5-13

Examine the following code and analyze the result, including the N and V flags.

```
LDI    R20,0x60    ;R20 = 0110 0000 (+70)
LDI    R21,0x46    ;R21 = 0100 0110 (+96)
ADD    R20,R21     ;R20 = (+96) + (+70) = 1010 0110
                        ;R20 = A6H = -90 decimal, INVALID!!
```

Solution:

```
+96    0110 0000
+ +70   0100 0110
+ 166   1010 0110  N = 1 (negative) and V = 1 Sum = -90
```

According to the CPU, the result is negative (N = 1), which is wrong. The CPU sets V = 1 to indicate the overflow error. Remember that the N flag is the D7 bit. If N = 0, the sum is positive, but if N = 1, the sum is negative.

In Example 5-13, +96 was added to +70 and the result, according to the CPU, was -90. Why? The reason is that the result was larger than what R0 could contain. Like all other 8-bit registers, R0 could only contain values less than or equal to +127. The designers of the CPU created the overflow flag specifically for the purpose of informing the programmer that the result of the signed number operation is erroneous. The N flag is D7 of the result. If N = 0, the sum is positive (+) and if N = 1, then the sum is negative.

When is the V flag set?

In 8-bit signed number operations, V is set to 1 if either of the following two conditions occurs:

1. There is a carry from D6 to D7 but no carry out of D7 ($C = 0$).
2. There is a carry from D7 out ($C = 1$) but no carry from D6 to D7.

In other words, the overflow flag is set to 1 if there is a carry from D6 to D7 or from D7 out, but not both. This means that if there is a carry both from D6 to D7 and from D7 out, $V = 0$. In Example 5-13, because there is only a carry from D6 to D7 and no carry from D7 out, $V = 1$.

In Example 5-14, because there is only a carry from D7 out and no carry from D6 to D7, $V = 1$.

Example 5-14

Examine the following code, noting the role of the V and N flags:

```
LDI    R20,0x80    ;R20 = 1000 0000 (80H = -128)
LDI    R21,0xFE    ;R21 = 1111 1110 (FEH = -2)
ADD     R20,R21     ;R20 = (-128) + (-2)
                        ;R20 = 10000000 + 11111110 = 0111 1110,
                        ;N = 0, R0 = 7EH = +126, invalid
```

Solution:

-128	1000 0000	
+ - 2	1111 1110	
- 130	0111 1110	N = 0 (positive) and V = 1

According to the CPU, the result is +126, which is wrong, and $V = 1$ indicates that. Notice that the N flag indicates the sign of the corrupted result, not the sign that the real result should have.

Further considerations on the V flag

In the ADD instruction, there are two different conditions. Either the operands have the same sign or the signs of the operands are different. When we ADD two numbers with different signs, the absolute value of the result is smaller than the operands before executing the ADD instruction. So overflow definitely cannot happen after two operands with different signs are added. Overflow is possible only when we ADD two operands with the same sign. In this case the absolute value of the result is larger than the operands before executing the ADD instruction. So it is possible that the result will be too large for the register and cause overflow. If we ADD two numbers with the same sign, the results should have the same sign too. If we add two numbers with the same sign and the result sign is different, we know that overflow has occurred. That is exactly the way that the CPU knows when to set the V flag. In the AVR the equation of the V flag is as follows:

$$V = R_{d7} \cdot R_{r7} \cdot \overline{R_7} + \overline{R_{d7}} \cdot \overline{R_{r7}} \cdot R_7$$

where R_{d7} and R_{r7} are the 7th bit of the operands and R_7 is the 7th bit of

the result. We can extend this concept to the SUB instructions ($a - b = a + (-b)$)

Study Examples 5-15 and 5-16 to understand the overflow flag in signed arithmetic.

Example 5-15

Examine the following code, noting the role of the V and N flags:

```
LDI    R20,-2           ;R20 = 1111 1110 (R20 = FEH)
LDI    R21,-5           ;R21 = 1111 1110 (R21 = FBH)
ADD     R20,R21          ;R20 = (-2) + (-5) = -7 or F9H
                        ;correct, since V = 0
```

Solution:

```
  -2      1111 1110
+ -5      1111 1011
-----
 - 7      1111 1001    and V = 0 and N = 1. Sum is negative
```

According to the CPU, the result is -7, which is correct, and the V indicates that ($V = 0$).

Example 5-16

Examine the following code, noting the role of the V and N flags:

```
LDI    R20,7            ;R20 = 0000 0111
LDI    R21,18           ;R21 = 0001 0010
ADD     R20,R21          ;R20 = (+7) + (+18)
                        ;R20 = 00000111 + 00010010 = 0001 1001
                        ;R20 = (+7) + (+18) = +25, N = 0, positive
                        ;and correct, V = 0
```

Solution:

```
  + 7 0000 0111
+ +18 0001 0010
-----
+25 0001 1001    N = 0 (positive 25) and V = 0
```

According to the CPU, this is +25, which is correct, and $V = 0$ indicates that.

From Examples 5-14 to 5-16, we conclude that in any signed number addition, V indicates whether the result is valid or not. If $V = 1$, the result is erroneous; if $V = 0$, the result is valid. We can state emphatically that in unsigned number addition, the programmer must monitor the status of C (carry flag), and in signed number addition, the V (overflow) flag must be monitored. In the AVR, instructions such as BRCS and BRCC allow the program to branch right after the addition of unsigned numbers according to the value of C flag. There are also the BRVC and the BRVS instructions for the V flag that allow us to correct the signed number error. We also have two branch instructions for the N flag (negative), BRPL and BRMI.

What is the difference between the N and S flags?

As we mentioned before, in signed numbers the N flag represents the D7 bit of the result. If the result is positive, the N flag is zero, and if the result is negative, the N flag is one, which is why it is called the Negative flag.

In operations on signed numbers, overflow is possible. Overflow corrupts the result and negates the sign bit. So if you ADD two positive numbers, in case of overflow, the N flag would be 1 showing that the result is negative! The S flag helps you to know the sign of the real result. It checks the V flag in addition to the D7 bit. If $V = 0$, it shows that overflow has not occurred and the S flag will be the same as D7 to show the sign of the result. If $V = 1$, it shows that overflow has occurred and the S flag will be opposite to the D7 to show the sign of the real (not the corrupted) result. See Example 5-17.

Example 5-17

Study Examples 5-13 through 5-16 again and state what the value of the S flag is in each of them and whether the value of the S flag is the same as that of the N flag.

Solution:

Example 5-13: Because two positive numbers are added, the sign of the real result is positive, so $S = 0$ (for positive). The value of the S flag is not the same as that of the N flag (1) because there is overflow ($V = 1$).

Example 5-14: Because two negative numbers are added, the sign of the real result is negative, so $S = 1$ (for negative). The value of the S flag is not the same as that of the N flag (0) because there is overflow ($V = 1$).

Example 5-15: Because two negative numbers are added, the sign of the real result is negative, so $S = 1$ (for negative). The value of the S flag is the same as that of the N flag (1) because there is no overflow ($V = 0$).

Example 5-16: Because two positive numbers are added, the sign of the real result is positive, so $S = 0$ (for positive). The value of the S flag is the same as that of the N flag (0) because there is overflow ($V = 0$).

Instructions to create 2's complement

The AVR has a special instruction to make the 2's complement of a number. It is called NEG (negate), which is discussed in the next section.

Review Questions

1. In an 8-bit operand, bit _____ is used for the sign bit.
2. Convert $-16H$ to its 2's complement representation.
3. The range of byte-sized signed operands is _____ to _____.
4. Show $+9$ and -9 in binary.
5. Explain the difference between a carry and an overflow.

SECTION 5.3: LOGIC AND COMPARE INSTRUCTIONS

Apart from I/O and arithmetic instructions, logic instructions are some of the most widely used instructions. In this section we cover Boolean logic instructions such as AND, OR, Exclusive-OR (XOR), and complement. We will also study the compare instruction.

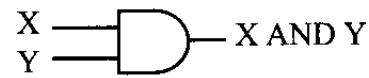
AND

AND Rd,Rr ;Rd = Rd AND Rr

This instruction will perform a logical AND on the two operands and place the result in the left-hand operand. There is also the “ANDI Rd, k” instruction in which the right-hand operand can be a constant value. The AND instruction will affect the Z, S, and N flags. N is D7 of the result, and Z = 1 if the result is zero. The AND instruction is often used to mask (set to 0) certain bits of an operand. See Example 5-18.

Logical AND Function

Inputs		Output
X	Y	X AND Y
0	0	0
0	1	0
1	0	0
1	1	1



Example 5-18

Show the results of the following.

```
LDI    R20,0x35    ;R20 = 35H
ANDI    R20,0x0F    ;R20 = R20 AND 0FH (now R20 = 05)
```

Solution:

	35H	0011 0101	
AND	0FH	0000 1111	

	05H	0000 0101	;35H AND 0FH = 05H, Z = 0, N = 0

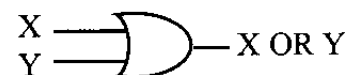
OR

OR Rd,Rr ;Rd = Rd OR Rr

This instruction will perform a logical OR on the two operands and place the result in the left-hand operand. There is also the “ORI Rd, k” instruction in which the right-hand operand can be a constant value. The OR instruction will affect the Z, S, and N flags. N is D7 of the result and Z = 1 if the result is zero. The OR instruction can be used to set certain bits of an operand to 1. See Example 5-19.

Logical OR Function

Inputs		Output
X	Y	X OR Y
0	0	0
0	1	1
1	0	1
1	1	1



Example 5-19

(a) Show the results of the following:

```
LDI    R20, 0x04           ;R20 = 04
ORI     R20, 0x30           ;now R20 = 34H
```

(b) Assume that PB2 is used to control an outdoor light, and PB5 to control a light inside a building. Show how to turn “on” the outdoor light and turn “off” the inside one.

Solution:

```
(a)      04H      0000 0100
OR       30H      0011 0000
-----
        34H      0011 0100      04 OR 30 = 34H, Z = 0 and N = 0
```

```
(b)
SBI      DDRB, 2           ;bit 2 of Port B is output
SBI      DDRB, 5           ;bit 5 of Port B is output
IN       R20, PORTB        ;move PORTB to R20. (Notice that we read
                           ;the value of PORTB instead of PINB
                           ;because we want to know the last value
                           ;of PORTB, not the value of the AVR
                           ;chip pins.)
ORI      R20, 0b00000100   ;set bit 2 of R20 to one
ANDI     R20, 0b11011111   ;clear bit 5 of R20 to zero
OUT      PORTB, R20        ;out R20 to PORTB
```

```
HERE:    JMP HERE          ;stop here
```

EX-OR

```
EOR      Rd, Rs           ;Rd = Rd XOR Rs
```

This instruction will perform a logical EX-OR on the two operands and place the result in the left-hand operand. The EX-OR instruction will affect the Z, S, and N flags. N is D7 of the result and Z = 1 if the result is zero. See Example 5-20.

Logical XOR Function

Inputs		Output
A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0



Example 5-20

Show the results of the following:

```
LDI      R20, 0x54
LDI      R21, 0x78
EOR      R20, R21
```

Solution:

```
      54H      0101 0100
XOR     78H      0111 1000
-----
      2CH      0010 1100      54H XOR 78H = 2CH, Z = 0, N = 0
```

EX-OR can also be used to see if two registers have the same value. The “EOR R0, R1” instruction will EX-OR the R0 register and R1, and put the result in R0. If both registers have the same value, 00 is placed in R0 and the Z flag is set ($Z = 1$). Then, we can use the BREQ or BRNE instruction to make a decision based on the result. See Examples 5-21 and 5-22.

Example 5-21

The EX-OR instruction can be used to test the contents of a register by EX-ORing it with a known value. In the following code, we show how EX-ORing the value 45H with itself will raise the Z flag:

```
OVER:      IN      R20,PINB
           LDI      R21,0x45
           EOR      R20,R21
           BRNE     OVER
```

Solution:

45H	01000101
<u>45H</u>	<u>01000101</u>
00	00000000

EX-ORing a number with itself sets it to zero with $Z = 1$. We can use the BREQ instruction to make the decision. EX-ORing with any other number will result in a nonzero value.

Example 5-22

Read and test PORTB to see whether it has the value 45H. If it does, send 99H to PORTC; otherwise, it is cleared.

Solution:

```
LDI      R20,0xFF      ;R20 = 0xFF
OUT      DDRC,R20      ;Port C is output
LDI      R20,0x00      ;R20 = 0
OUT      DDRB,R20      ;Port B is input
OUT      PORTC,R20     ;PORTC = 00
LDI      R21,0x45      ;R21 = 45
HERE:    IN      R20,PINB ;get a byte
         EOR      R20,R21 ;EX-OR with 0x45

         BRNE     HERE   ;branch if PORTB has value other than 45
         LDI      R20,0x99 ;R20 = 0x99
         OUT      PORTC,R20 ;PORTC = 99h
EXIT:    JMP      EXIT   ;stop here
```

Another widely used application of EX-OR is to toggle the bits of an operand. The following code demonstrates how to use EX-OR to toggle the bits of an operand.

```
LDI      R20,0xFF
EOR      R0, R20      ;EX-OR R0 with 1111 1111 will
                     ;change all the bits of R0 to
                     ;opposite
```



COM (complement)

This instruction complements the contents of a register. The complement action changes the 0s to 1s, and the 1s to 0s. This is also called *1's complement*.

```
LDI    R20, 0xAA    ; R20 = 0xAA
COM     R20           ; now R20 = 55H
```

Logical Inverter

Input	Output
X	NOT X
0	1
1	0

X  NOT X

NEG (negate)

This instruction takes the 2's complement of a register. See Example 5-23.

Example 5-23

Find the 2's complement of the value 85H. Notice that 85H is -123.

Solution:

```
LDI    R21, 0x85           ; 85H = 1000 0101
                        ; 1's = 0111 1010
                        + 1
NEG     R21                ; 2's comp 0111 1011 = 7BH
```

Compare instructions

CP Rd, Rr

The AVR has the CP instruction for the compare operation. The compare instruction is really a subtraction, except that the values of the operands do not change. There is also the "CPI Rd, k" instruction in which the right-hand operand can be a constant value.

The AVR has some conditional branch instructions that can be used after the CP instruction to make decisions based on the result of the CP instruction. In Chapter 3 we used some of them. Next, you will learn some other conditional branches.

Conditional branch instructions

As we studied in Chapter 3, conditional branches alter the flow of control if a condition is true. In the AVR there are at least two conditional jumps for each flag of the status register. Here we will describe eight of the most important conditional jumps. Others are similar but of different flags. Table 5-2 shows the conditional branch instructions that we will describe in this section.

Table 5-2: AVR Compare Instructions

BREQ	Branch if equal	Branch if Z = 1
BRNE	Branch if not equal	Branch if Z = 0
BRSH	Branch if same or higher	Branch if C = 0
BRLO	Branch if lower	Branch if C = 1
BRLT	Branch if less than (signed)	Branch if S = 1
BRGE	Branch if greater than or equal (signed)	Branch if S = 0
BRVS	Branch if Overflow flag set	Branch if V = 1
BRVC	Branch if Overflow flag clear	Branch if V = 0

BREQ and BRNE instructions

```
BREQ k      ;if (Z = 1) then branch
            ;else continue
```

The BREQ makes decisions based on the Z flag. If $Z = 1$ the BREQ instruction branches. Notice that after the CP instructions, the $Z = 1$ means that the operands were equal, and after the DEC instruction it means that the operand is now equal to zero.

The BRNE instruction, like the BREQ, makes decisions based on the Z flag, but it branches when $Z = 0$. (After the CP instructions, $Z = 0$ means that the operands were not equal, and after the DEC instruction it means that the operand is not equal to zero.) See Example 5-24.

Notice that the BREQ and BRNE instructions can be used for both signed, and unsigned numbers.

Example 5-24

Write a program to monitor PORTB continuously for the value 63H. It should stop monitoring only if $\text{PORTB} = 63\text{H}$.

Solution:

```
LDI    R20,0x00
OUT     DDRB,R20      ;PORT B is input
LDI     R21,0x63
AGAIN:
IN      R20,PINB
CP      R20,R21        ;compare with 0x63, Z = 1 if yes
BRNE    AGAIN          ;go to AGAIN if PORTB is not equal to 0x63
....
```

BRSH and BRLO instructions

```
BRSH k      ;if (C = 0) then branch
            ;else continue
```

The BRSH makes decisions based on the C flag. If $C = 0$ (which, after the CP instructions for unsigned numbers, means that the left-hand operand of the CP instruction was the same as or higher than the right-hand operand) the CPU will jump.

The BRLO instruction, like the BRSH, makes decisions based on the C flag, but it branches when $C = 1$. (After the CP instructions for unsigned numbers, $C = 1$ means that the left-hand operand of the CP instruction was lower than the right-hand operand.) See Example 5-25.

Notice that the BRSH and the BRLO instructions can be used to compare unsigned numbers. To compare signed numbers you should use the BRGE and BRLT instructions. We will discuss them in more detail next.

We can use more than one conditional branch instruction to make more complicated decisions. See Example 5-26.

Example 5-25

Write a program to find the greater of the two values 27 and 54, and place it in R20.

Solution:

```
.EQU VAL_1=27
.EQU VAL_2=54

LDI R20,VAL_1 ;R20 = VAL_1
LDI R21,VAL_2 ;R21 = VAL_2
CP R21,R20 ;compare R21 and R20
BRLO NEXT ;if R21<R20 (branch if lower) go to NEXT
LDI R20,VAL_2 ;R20 = VAL_2
NEXT:
```

Example 5-26

Assume that Port B is an input port connected to a temperature sensor. Write a program to read the temperature and test it for the value 75. According to the test results, place the temperature value into the registers indicated by the following.

```
If T = 75      then R16 = T      ; R17 = 0 ; R18 = 0
If T > 75      then R16 = 0      ; R17 = T ; R18 = 0
If T < 75      then R16 = 0      ; R17 = 0 ; R18 = T
```

Solution:

```
LDI R20,0x00 ;R20 = 0
OUT DDRB,R20 ;Port B = input

CLR R16 ;R16 = 0
CLR R17 ;R17 = 0
CLR R18 ;R18 = 0

IN R20,PINB
CPI R20,75 ;compare R20 (PORTB) and 75
BRSH SAME_HI ;executes when R20 < 75

MOV R18,R20
RJMP CNTNU

SAME_HI: ;executes when R20 >= 75

BRNE HI

MOV R16,R20 ;executes when R20 = 75
RJMP CNTNU

HI: ;executes when R20 > 75

MOV R17,R20
CNTNU: .....
```

BRGE and BRLT instructions

The BRGE makes decisions based on the S flag. If $S = 0$ (which, after the CP instruction for signed numbers, means that the left-hand operand of the CP instruction was greater than or equal to the right-hand operand) the BRGE instruction branches in a forward or backward direction relative to program counter.

The BRLT is like the BRGE, but it branches when $S = 1$. Notice that the BRGE, and the BRLT are used with signed numbers.

BRVS and BRVC instructions

As we mentioned before, the V (overflow) flag must be monitored by the programmer to detect overflow and handle the error. The BRVC and BRVS instructions let you check the value of the V flag and change the flow of the program if overflow has occurred. See Example 5-27.

Example 5-27

Write a program to add two signed numbers. The numbers are in R21 and R22. The program should store the result in R21. If the result is not correct, the program should put 0xAA on PORTA and clear R21.

Solution:

```
LDI    R21,0xFA           ;R21 = 0xFA
LDI    R22,0x05           ;R22 = 0x05
LDI    R23,0xFF           ;R23 = 0xFF
OUT    DDRA,R23           ;Port A is output
ADD    R21,R22             ;R21 = R21 + R22
BRVC   NEXT              ;if V = 0 ( no error) then go to next
LDI    R23,0xAA           ;R23 = 0xAA
OUT    PORTA,R23          ;send 0xAA to PORTA
LDI    R21,0x00           ;clear R21
NEXT:  ...
```

Review Questions

- Find the content of R20 after the following code in each case:
(a) LDI R20,0x37 (b) LDI R20,0x37 (c) LDI R20,0x37
 LDI R21,0xCA LDI R21,0xCA LDI R21,0xCA
 AND R20,R21 OR R20,21 EOR
- To mask certain bits of R20, we must AND it with _____.
- To set certain bits of R20 to 1, we must OR it with _____.
- EX-ORing an operand with itself results in _____.
- True or false. The CP instruction alters the contents of its operands.
- Find the contents of register R20 after execution of the following code:
 LDI R20,0
 LDI R21,0x99
 LDI R22,0xFF
 OR R20,R21
 EOR R20,R22

SECTION 5.4: ROTATE AND SHIFT INSTRUCTIONS AND DATA SERIALIZATION

In many applications there is a need to perform a bitwise rotation of an operand. In the AVR the rotation instructions ROL and ROR are designed specifically for that purpose. They allow a program to rotate a register right or left through the carry flag. We explore the rotate instructions next because they are widely used in many different applications.

Rotating through the carry

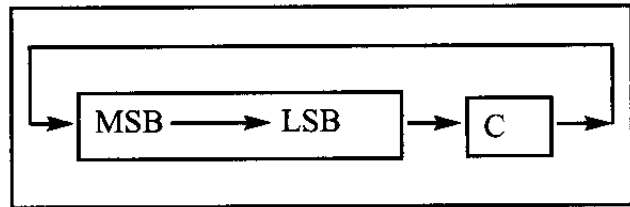
There are two rotate instructions in the AVR. They involve the carry flag. Each is shown next.

ROR instruction

ROR Rd ;rotate Rd right through carry

In the ROR, as bits are rotated from left to right, the carry flag enters the MSB, and the LSB exits to the carry flag. In other words, in ROR the C is moved to the MSB, and the LSB is moved to the C.

In reality, the carry flag acts as if it is part of the register, making it a 9-bit register. Examine the following code.

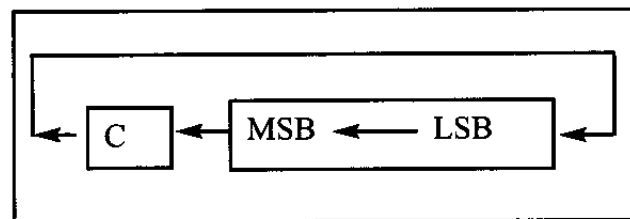


```
CLC                ;make C = 0 (carry is 0)
LDI                R20,0x26 ;R20 = 0010 0110
ROR                R20      ;R20 = 0001 0011 C = 0
ROR                R20      ;R20 = 0000 1001 C = 1
ROR                R20      ;R20 = 1000 0100 C = 1
```

ROL instruction

The other rotating instruction is ROL. In ROL, as bits are shifted from right to left, the carry flag enters the LSB, and the MSB exits to the carry flag. In other words, in ROL the C is moved to the LSB, and the MSB is moved to the C.

See the following code and diagram. Again, the carry flag acts as if it is part of the register, making it a 9-bit register. Examine the following code.



```
SEC                ;make C = 1
LDI                R20,0x15 ;R20 = 0001 0101
ROL                R20      ;R20 = 0010 1011 C = 0
ROL                R20      ;R20 = 0101 0110 C = 0
ROL                R20      ;R20 = 1010 1100 C = 0
ROL                R20      ;R20 = 0101 1000 C = 1
```

Serializing data

Serializing data is a way of sending a byte of data one bit at a time through a single pin of the microcontroller. There are two ways to transfer a byte of data serially:

1. Using the serial port. In using the serial port, programmers have very limited control over the sequence of data transfer. The details of serial port data transfer are discussed in Chapter 11.
2. The second method of serializing data is to transfer data one bit at a time and control the sequence of data and spaces between them. In many new generations of devices such as LCD, ADC, and ROM, the serial versions are becoming popular because they take up less space on a printed circuit board. Next, we discuss how to use rotate instructions in serializing data.

Serializing a byte of data

Serializing data is one of the most widely used applications of the rotate instruction. We can use the rotate instruction to transfer a byte of data serially (one bit at a time). Shift instructions can be used for the same job. After presenting rotate instructions in this section we will discuss shift instructions in more detail. Example 5-28 shows how to transfer an entire byte of data serially via any AVR pin.

Example 5-28

Write a program to transfer the value 41H serially (one bit at a time) via pin PB1. Put one high at the start and end of the data. Send the LSB first.

Solution:

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

```
SBI    DDRB, 1           ;bit 1 of Port B is output
LDI    R20,0x41          ;R20 = the value to be sent
```

```
CLC                    ;clear carry flag
LDI    R16, 8            ;R16 = 8
SBI    PORTB, 1         ;bit 1 of PORTB is 1
```

AGAIN:

```
ROR    R20              ;rotate right R20 (send LSB to C flag)
BRCS   ONE              ;if C = 1 then go to ONE
CBI    PORTB, 1         ;bit 1 of PORTB is cleared to zero
JMP    NEXT             ;go to NEXT
```

```
ONE:   SBI    PORTB, 1   ;bit 1 of PORTB is set to one
```

NEXT:

```
DEC    R16              ;decrement R16
BRNE   AGAIN           ;if R16 is not zero then go to AGAIN
SBI    PORTB, 1         ;bit 1 of PORTB is set to one
```

```
HERE:  JMP    HERE      ;RB1 = high
```

Example 5-29 also shows how to bring in a byte of data serially. We will see how to use these concepts for a serial RTC (real-time clock) chip in Chapter 16. Example 5-30 shows how to scan the bits in a byte.

Example 5-29

Write a program to bring in a byte of data serially via pin RC7 and save it in R20 register. The byte comes in with the LSB first.

Solution:

```
.INCLUDE      "M32DEF.INC"

      CBI    DDRC, 7      ;bit 7 of Port C is input
      LDI    R16, 8       ;R16 = 8
      LDI    R20, 0       ;R20 = 0
AGAIN:
      SBIC   PINC, 7      ;skip the next line if bit 7 of Port C is 0
      SEC                      ;set carry flag to one
      SBIS   PINC, 7      ;skip the next line if bit 7 of Port C is 1
      CLC                      ;clear carry flag to zero
      ROR    R20          ;rotate right R20. move C flag to MSB of R21
      DEC    R16          ;decrement R16
      BRNE   AGAIN       ;if R16 is not zero go to AGAIN

      HERE:  JMP     HERE      ;stop here
```

Example 5-30

Write a program that finds the number of 1s in a given byte.

Solution:

```
.INCLUDE      "M32DEF.INC"

      LDI    R20, 0x97
      LDI    R30, 0       ;number of 1s
      LDI    R16, 8       ;number of bits in a byte

AGAIN:
      ROR    R20          ;rotate right R20 and move LSB to C flag
      BRCC   NEXT        ;if C = 0 then go to NEXT
      INC    R30          ;increment R30
NEXT:
      DEC    R16          ;decrement R16
      BRNE   AGAIN       ;if R16 is not zero then go to AGAIN

      ROR    R20          ;one more time to leave R20 unchanged

      HERE:  JMP     HERE      ;stop here
```

Shift instructions

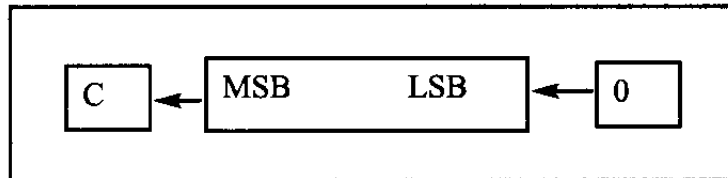
There are three shift instructions in the AVR. All of them involve the carry flag. Each is shown next.

LSL instruction

LSL Rd ;logical shift left

In LSL, as bits are shifted from right to left, 0 enters the LSB, and the MSB exits to the carry flag.

In other words, in LSL, 0 is moved to the LSB, and the MSB is moved to the C flag. Notice that this instruction multiplies the content of the register by 2 assuming that after LSL the carry flag is not set. Examine the following code.

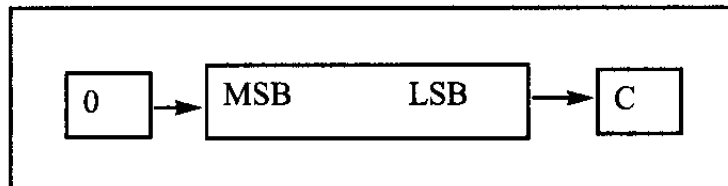


```
CLC          ;make C = 0 (carry is 0 )
LDI          R20,0x26 ;R20 = 0010 0110 (38)  c = 0
LSL          R20      ;R20 = 0100 1100 (76)  C = 0
LSL          R20      ;R20 = 1001 1000 (152) C = 0
LSL          R20      ;R20 = 0011 0000 (48)  C = 1
                ;as C = 1 and content of R20
                ;is not multiplied by 2
```

LSR instruction

The second shift instruction is LSR. In LSR, as bits are shifted from left to right, 0 enters the MSB, and the LSB exits to the carry flag. In other words, in

LSR, 0 is moved to the MSB, and the LSB is moved to the C flag. Notice that this instruction divides the content of the register by 2 and the carry flag contains the remainder of the division. Examine the following code.

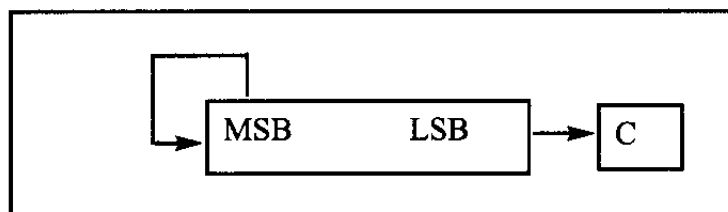


```
LDI          R20,0x26 ;R20 = 0010 0110 (38)
LSR          R20      ;R20 = 0001 0011 (19) C = 0
LSR          R20      ;R20 = 0000 1001 (9)  C = 1
LSR          R20      ;R20 = 0000 0100 (4)  C = 1
```

LSR cannot be used to divide signed numbers by 2. See Example 5-31.

ASR instruction

The third shift instruction is ASR, which means arithmetic shift right. The ASR instruction can divide signed numbers by two. In ASR, as bits are



shifted from left to right, the MSB is held constant and the LSB exits to the carry flag. In other words, MSB is not changed but is copied to D6, D6 is moved to D5, D5 is moved to D4, and so on. Examine the following code.

```
LDI      R20,0D60      ;R20 = 1101 0000 (-48)  C = 0
LSR      R20           ;R20 = 1110 1000 (-24)  C = 0
LSR      R20           ;R20 = 1111 0100 (-12)  C = 0
LSR      R20           ;R20 = 1111 1010 (-6)   C = 0
LSR      R20           ;R20 = 1111 1101 (-3)   C = 0
LSR      R20           ;R20 = 1111 1110 (-1)   C = 1
```

Example 5-32 shows how we can use ROR to divide a register by a number that is a power of 2.

Example 5-31

Assume that R20 has the number -6. Show that LSR cannot be used to divide the content of R20 by 2. Why?

Solution:

```
LDI      R20,0xFA      ;R20 = 1111 1010 (-6)
LSR      R20           ;R20 = 0111 1101 (+125)
                        ;-6 divided by 2 is not +125 and
                        ;the answer is not correct
```

Because LSR shifts the sign bit it changes the sign of the number and therefore cannot be used for signed numbers.

Example 5-32

Assume that R20 has the number 48. Show how we can use ROR to divide R20 by 8.

Solution:

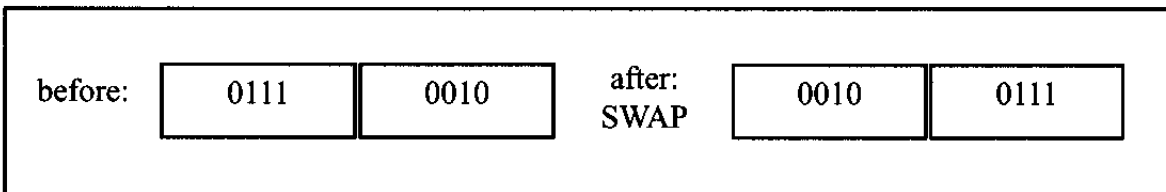
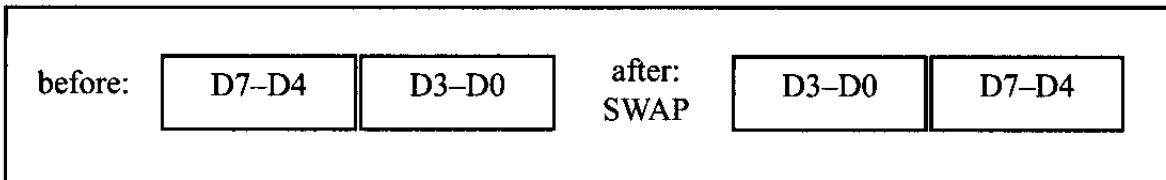
```
                        ;to divide a number by 8 we can
                        ;shift it 3 bits to the right. without
                        ;LSR we have to ROR 3 times and
                        ;clear carry flag before
                        ;each rotation

LDI      R20,0x30      ;R20 = 0011 0000 (48)
CLC      ;clear carry flag
ROR      R20           ;R20 = 0001 1000 (24)
CLC      ;clear carry flag
ROR      R20           ;R20 = 0000 1100 (12)
CLC      ;clear carry flag
ROR      R20           ;R20 = 0000 0110 (6)
                        ;48 divided by 8 is 6 and
                        ;the answer is correct
```

SWAP instruction

SWAP Rd ;swap nibbles

Another useful instruction is the SWAP instruction. It works on R0–R31. It swaps the lower nibble and the higher nibble. In other words, the lower 4 bits are put into the higher 4 bits, and the higher 4 bits are put into the lower 4 bits. See the diagrams below.



Example 5-33 shows how to exchange nibbles of a byte with and without SWAP instruction.

Example 5-33

(a) Find the contents of the R20 register in the following code .

```
LDI    R20, 0x72
SWAP   R20
```

(b) In the absence of a SWAP instruction, how would you exchange the nibbles?
Write a simple program to show the process.

Solution:

(a)

```
LDI    R20, 0x72      ;R20 = 0x72
SWAP   R20            ;R20 = 0x27
```

(b)

```
LDI    R20,0x72
LDI    R16,4
LDI    R21,0
BEGIN:
  CLC
  ROL   R20
  ROL   R21
  DEC   R16
  BRNE  BEGIN
  OR    R20, R21
  HERE: JMP  HERE
```

Review Questions

1. What is the value of R20 after the following code is executed?

```
LDI    R20,0x40
CLC
ROR     R20
ROR     R20
ROR     R20
ROR     R20
```

2. What is the value of R20 after the following code is executed?

```
LDI    R20,0x40
CLC
ROL     R20
ROL     R20
ROL     R20
ROL     R20
```

3. What is the value of R20 after execution of the following code?

```
LDI    R20,0x40
SEC
ROL     R20
SWAP    R20
```

4. What is the value of R20 after execution of the following code?

```
LDI    R20,0x00
SEC
ROL     R20
CLC
ROL     R20
SEC
ROL     R20
CLC
ROL     R20
SEC
ROL     R20
CLC
ROL     R20
SEC
ROL     R20
CLC
ROL     R20
```

5. What is the value of R20 after execution of the last code if you replace ROL with the ROR instruction?
6. How many LSR instructions are needed to divide a number by 32?

SECTION 5.5: BCD AND ASCII CONVERSION

In this section you will learn about packed and unpacked BCD numbers. We will also show how you can change packed BCD to unpacked BCD and vice versa. Then you will learn to convert ASCII codes to BCD and vice versa.

BCD (binary coded decimal) number system

BCD stands for *binary coded decimal*. BCD is needed because in everyday life we use the digits 0 to 9 for numbers, not binary or hex numbers. Binary representation of 0 to 9 is called BCD (see Figure 5-3). In computer literature, one encounters two terms for BCD numbers: (1) unpacked BCD, and (2) packed BCD. We describe each one next.

<i>Digit</i>	<i>BCD</i>
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Figure 5-3. BCD Code

Unpacked BCD

In unpacked BCD, the lower 4 bits of the number represent the BCD number, and the rest of the bits are 0. For example, “0000 1001” and “0000 0101” are unpacked BCD for 9 and 5, respectively. Unpacked BCD requires 1 byte of memory, or an 8-bit register, to contain it.

Packed BCD

In packed BCD, a single byte has two BCD numbers in it: one in the lower 4 bits, and one in the upper 4 bits. For example, “0101 1001” is packed BCD for 59H. Only 1 byte of memory is needed to store the packed BCD operands. Thus, one reason to use packed BCD is that it is twice as efficient in storing data.

ASCII numbers

On ASCII keyboards, when the key “0” is activated, “011 0000” (30H) is provided to the computer. Similarly, 31H (011 0001) is provided for key “1”, and so on, as shown in Table 5-3.

Table 5-3: ASCII and BCD Codes for Digits 0–9

Key	ASCII (hex)	Binary	BCD (unpacked)
0	30	011 0000	0000 0000
1	31	011 0001	0000 0001
2	32	011 0010	0000 0010
3	33	011 0011	0000 0011
4	34	011 0100	0000 0100
5	35	011 0101	0000 0101
6	36	011 0110	0000 0110
7	37	011 0111	0000 0111
8	38	011 1000	0000 1000
9	39	011 1001	0000 1001

It must be noted that BCD numbers are universal, although ASCII is standard in the United States (and many other countries). Because the keyboard, printers, and monitors all use ASCII, how does data get converted from ASCII to BCD, and vice versa? These are the subjects covered next.

Packed BCD to ASCII conversion

In many systems we have what is called a *real-time clock* (RTC). The RTC provides the time of day (hour, minute, second) and the date (year, month, day) continuously, regardless of whether the power is on or off (see Chapter 16). This data, however, is provided in packed BCD. For this data to be displayed on a device such as an LCD, or to be printed by the printer, it must be in ASCII format.

To convert packed BCD to ASCII, you must first convert it to unpacked BCD. Then the unpacked BCD is tagged with 011 0000 (30H). The following demonstrates converting packed BCD to ASCII. See also Example 5-34.

<i>Packed BCD</i>	<i>Unpacked BCD</i>	<i>ASCII</i>
29H	02H & 09H	32H & 39H
0010 1001	0000 0010 & 0000 1001	0011 0010 & 0011 1001

Example 5-34

Assume that R20 has packed BCD. Write a program to convert the packed BCD to two ASCII numbers and place them in R21 and R22.

Solution:

```
.INCLUDE      "M32DEF.INC"

    LDI       R20,0x29      ;the packed BCD to be converted is 29

    MOV       R21,R20       ;R21 = R20 = 29H
    ANDI      R21,0x0F       ;mask the upper nibble (R21 = 09H)
    ORI       R21,0x30       ;make it ASCII (R21 = 39H)

    MOV       R22,R20       ;R22 = R20 = 29H
    SWAP      R22            ;swap nibbles (R22 = 92H)
    ANDI      R22,0x0F       ;mask the upper nibble (R22 = 02)
    ORI       R22,0x30       ;make it ASCII (R22 = 32H)

HERE: JMP     HERE
```

ASCII to packed BCD conversion

To convert ASCII to packed BCD, you first convert it to unpacked BCD (to get rid of the 3), and then combine it to make packed BCD. For example, for 4 and 7 the keyboard gives 34 and 37, respectively. The goal is to produce 47H or "0100 0111", which is packed BCD. This process is illustrated next.

<i>Key</i>	<i>ASCII</i>	<i>Unpacked BCD</i>	<i>Packed BCD</i>
4	34	00000100	
7	37	00000111	01000111 which is 47H

```

LDI    R21, '4'      ;load character 4 to R21
LDI    R22, '7'      ;load character 7 to R22
ANDI   R21, 0x0F     ;mask upper nibble of R21
SWAP   R21           ;swap nibbles of R21
                     ;to make upper nibble of packed BCD
ANDI   R22, 0x0F     ;mask upper nibble of R22
OR     R22, R21       ;join R22 and R21 to make packed BCD
MOV    R20, R22       ;move the result to R20

```

After this conversion, the packed BCD numbers are processed and the result will be in packed BCD format.

Review Questions

- For the following decimal numbers, give the packed BCD and unpacked BCD representations.
 (a) 15 (b) 99
 (c) 25 (d) 55
- Show the binary and hex formats for “76” and its BCD version.
- Does the R20 register contain 54H after the following instruction is executed?
 LDI R20, 54
- 67H in BCD when converted to ASCII is ____ hex and ____ hex.

SUMMARY

This chapter discussed arithmetic instructions for both signed and unsigned data in the AVR. Unsigned data uses all 8 bits of the byte for data, making a range of 0 to 255 decimal. Signed data uses 7 bits for the data and 1 for the sign bit, making a range of -128 to +127 decimal.

In coding arithmetic instructions for the AVR, special attention has to be given to the possibility of a carry or overflow condition.

This chapter defined the logic instructions AND, OR, XOR, and complement. In addition, AVR Assembly language instructions for these functions were described. These functions are often used for bit manipulation purposes.

Compare and conditional branch instructions were described using different examples.

The rotate and swap instructions of the AVR are used in many applications such as serial devices. These instructions were discussed in detail.

Binary coded decimal (BCD) data represents the digits 0 through 9. Both packed and unpacked BCD formats were discussed. This chapter also described BCD and ASCII conversions.

PROBLEMS

SECTION 5.1: ARITHMETIC INSTRUCTIONS

1. Find the C, Z, and H flags for each of the following:

- | | | | | | |
|-----|-----|-----------|-----|-----|-----------|
| (a) | LDI | R20, 0x3F | (b) | LDI | R20, 0x99 |
| | LDI | R21, 0x45 | | LDI | R21, 0x58 |
| | ADD | R20, R21 | | ADD | R20, R21 |
| (c) | LDI | R20, 0xFF | (d) | LDI | R20, 0xFF |
| | CLR | R21 | | LDI | R21, 0x1 |
| | SEC | | | ADD | R20, R21 |
| | ADC | R20, R21 | | | |

2. Write a program to add 25 to the day of your birthday and save the result in R20.
3. Write a program to add the following numbers and save the result in R20.
0x25, 0x19, 0x12
4. Modify Problem 3 to add the result with 0x3D.
5. State the steps that the SUB instruction will go through for each of the following.
(a) 23H - 12H (b) 43H - 53H (c) 99 - 99
6. For Problem 5, write a program to perform each operation.
7. Write a program to add 7F9AH to BC48H and save the result in R20 (low byte) and R21 (high byte).
8. Write a program to subtract 7F9AH from BC48H and save the result in R20 (low byte) and R21 (high byte).
9. Show how to perform 77×34 in the AVR.
10. Show how to perform $64/4$ in the AVR.
11. The MUL instruction places the result in registers _____ and _____.

SECTION 5.2: SIGNED NUMBER CONCEPTS AND ARITHMETIC OPERATIONS

12. Show how the following numbers are represented by the assembler:
(a) -23 (b) +12 (c) -28
(d) +6FH (e) -128 (f) +127
13. The memory addresses in computers are _____ (signed, unsigned) numbers.
14. Write a program for each of the following and indicate the status of the V flag for each:

(a) (+15) + (-12) (b) (-123) + (-127)
(c) (+25H) + (+34H) (d) (-127) + (+127)
15. Explain the difference between the C and V flags and where each one is used.
16. When is the V flag raised? Explain.
17. Which register holds the V flag?
18. How do you detect the V flag in the AVR? How do you detect the C flag?

SECTION 5.3: LOGIC AND COMPARE INSTRUCTIONS

19. Find the contents of register R20 after each of the following instructions:

(a) LDI R20, 0x65
LDI R21, 0x76
AND R20, R21

(b) LDI R20, 0x70
LDI R21, 0x6B
OR R20, R21

(c) LDI R20, 0x95
LDI R21, 0xAA
EOR R20, R21

(d) LDI R20, 0x5D
LDI R21, 0x75
AND R20, R21

(e) LDI R20, 0x0C5
LDI R21, 0x12
OR R20, R21

(f) LDI R20, 0x6A
LDI R21, 0x6E
EOR R20, R21

(g) LDI R20, 0x37
LDI R21, 0x26
OR R20, R21

20. Explain how the BRSH instruction works.

21. Does the compare instruction affect the flag bits of the status register?

22. Assume that R20 = 85H. Indicate whether the conditional branch is executed in each of the following cases:

(a) LDI R21, 0x90
CP R20, R21
BRLO NEXT
...

(b) LDI R21, 0x70
CP R20, R21
BRSH NEXT
...

23. For Problem 22, indicate the value in R20 after execution of each program.

SECTION 5.4: ROTATE AND SHIFT INSTRUCTIONS AND DATA SERIALI-ZATION

24. Find the contents of R20 after each of the following is executed:

(a) LDI R20, 0x56
SWAP R20
CLC
ROR R20
ROR R20

(b) LDI R20, 0x39
SEC
ROL R20
ROL R0

(c) CLC
LDI R20, 0x4D
SWAP R20
ROL R20
ASR R20

(d) CP R20, R20
LDI R20, 0x7A
ROR R20

25. Show the code to replace the SWAP instruction:

(a) using the ROL instruction

(b) using the ROR instruction

26. Write a program that finds the number of zeros in an 8-bit data item.

27. Write a program that finds the position of the first high in an 8-bit data item. The data is scanned from D0 to D7. Give the result for 68H.
28. Write a program that finds the position of the first high in an 8-bit data item. The data is scanned from D7 to D0. Give the result for 68H.

SECTION 5.5: BCD AND ASCII CONVERSION

29. Write a program to convert the following packed BCD numbers to ASCII. Place the ASCII codes into R20 and R21.
 - (a) 0x76
 - (b) 0x87
30. For 3 and 2 the keyboard gives 33H and 32H, respectively. Write a program to convert these values to packed BCD and store the result in R20.

ANSWERS TO REVIEW QUESTIONS

SECTION 5.1: ARITHMETIC INSTRUCTIONS

1. R1:R0
2. No. Because immediate addressing mode is not supported.
3. 255, 255.
4. False.
5. R20.
6. We cannot use immediate addressing mode with ADD.
7. "ADI R1,0x04"
8. R1
9. (a) R21 = 00 and C = 1
(b) R21 = FF and C = 0
10.

43H 0100 0011		0100 0011
- 05H 0000 0101	2's complement	+ 1111 1011
3EH		0011 1110
11. $R1 = 95H - 4FH - 1 = 45H, C = 0$.

SECTION 5.2: SIGNED NUMBER CONCEPTS AND ARITHMETIC OPERATIONS

1. D7
2. 16H is 00010110 in binary and its 2's complement is 1110 1010 or -16H = EA in hex.
3. -128 to +127
4. +9 = 00001001 and -9 = 11110111 or F7 in hex.
5. An overflow is a carry into the sign bit (D7), but the carry is a carry out of register.

SECTION 5.3: LOGIC AND COMPARE INSTRUCTIONS

1. (a) 02 H
(b) FF H
(c) FD H
2. Zero
3. One
4. All zeros

5. False
6. 66H

SECTION 5.4: ROTATE AND SHIFT INSTRUCTIONS AND DATA SERIALIZATION

1. 04H
2. 02H
3. 18H
4. AAH
5. 55H
6. 5 LSL instructions

SECTION 5.5: BCD AND ASCII CONVERSION

1. (a) 15H = 0001 0101 packed BCD, 0000 0001,0000 0101 unpacked BCD
(b) 99H = 1001 1001 packed BCD, 0000 1001,0000 1001 unpacked BCD
(c) 25H = 0010 0101 packed BCD, 0000 0010,0000 0101 unpacked BCD
(d) 55H = 0101 0101 packed BCD, 0000 1001,0101 0101 unpacked BCD
2. 3736H = 00110111 00110110B
and in BCD we have 76H = 0111 0110B
3. No. We need to write it as 0x54 (to indicate that it is hex) or 0b01010100. The value 54 is interpreted as 36H by the assembler.
4. 36H, 37H