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
```

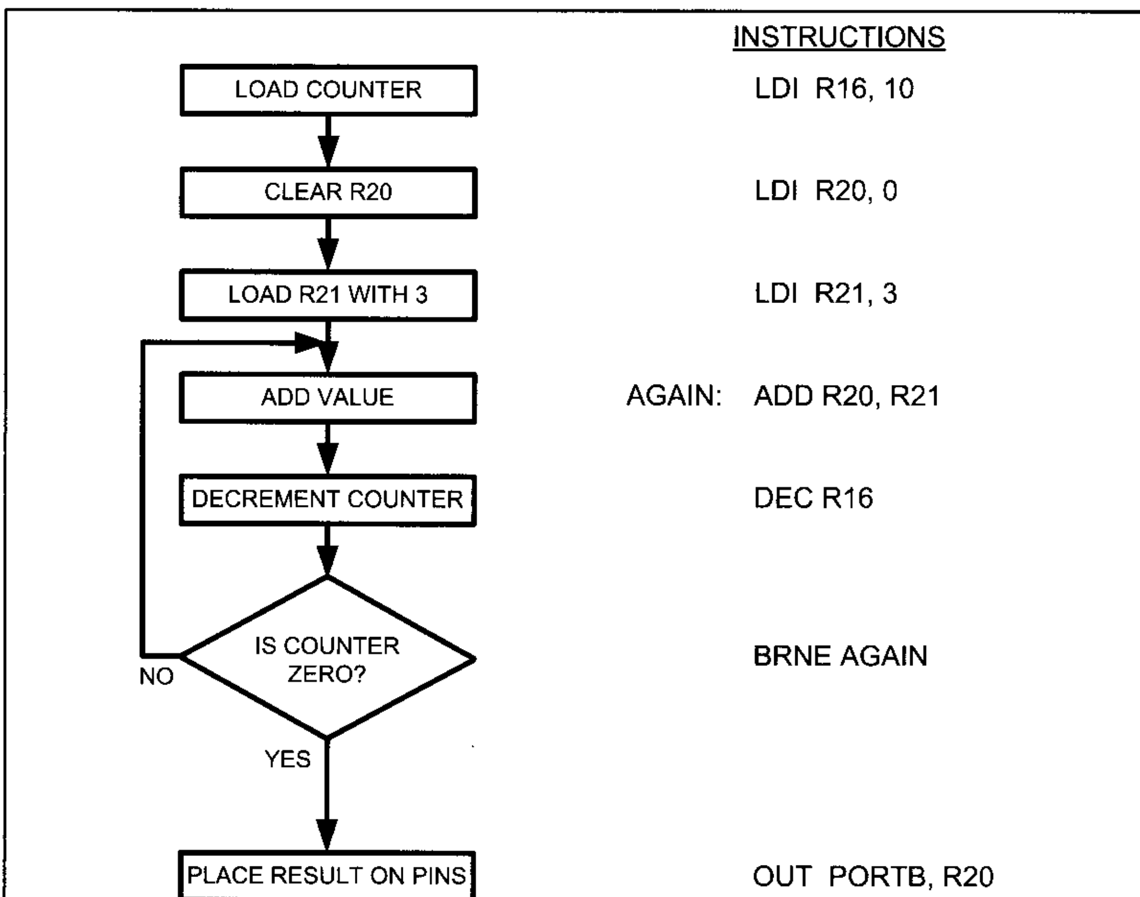


Figure 3-1. Flowchart for Example 3-1

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.

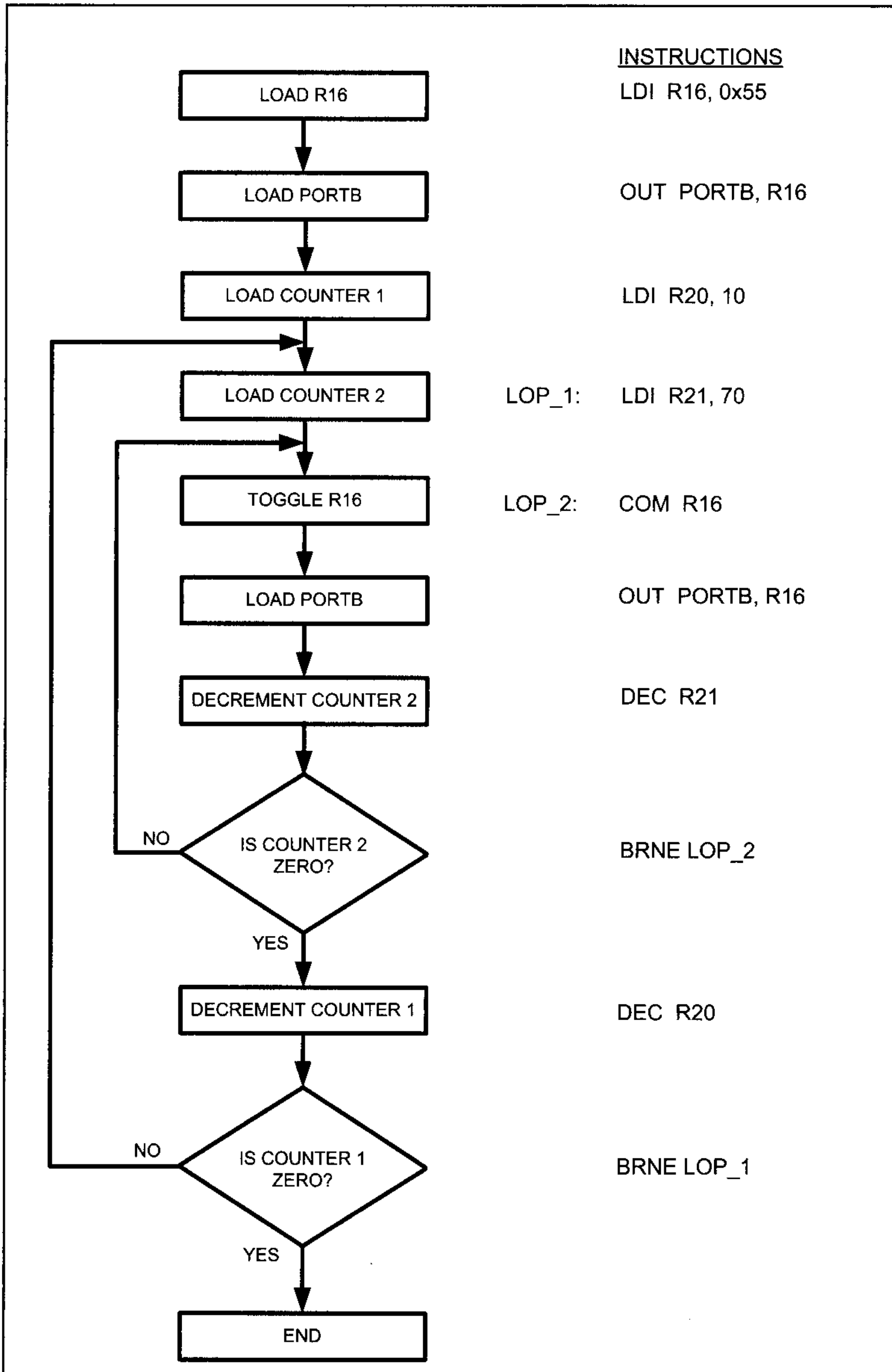


Figure 3-2. Flowchart for Example 3-3

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.

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

Instruction	Action
BRLO	Branch if C = 1
BRSH	Branch if C = 0
BREQ	Branch if Z = 1
BRNE	Branch if Z = 0
BRMI	Branch if N = 1
BRPL	Branch if N = 0
BRVS	Branch if V = 1
BRVC	Branch if V = 0

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.

BRSH (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 "`BRSH 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

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)
@0000000A: 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)
@0000000E: 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.

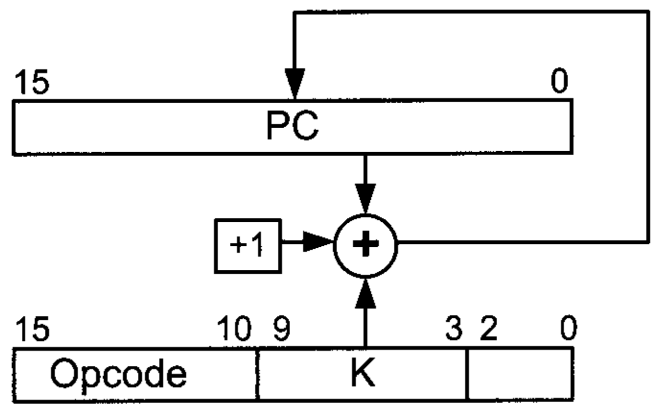


Figure 3-3. Calculating the Target Address in Conditional Branch Instructions

Example 3-7

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

Solution:

```
ADDRESS    Machine Mnemonic Operand
+00000000: E00A    LDI       R16, 10    ;R16 = 10 (decimal) for counter
+00000001: E040    LDI       R20, 0        ;R20 = 0
+00000002: E053    LDI       R21, 3        ;R21 = 3
+00000003: 0F45 AGAIN:ADD    R20, R21      ;add 03 to R20 (R20 = sum)
+00000004: 950A    DEC       R16           ;decrement R16 (counter)
+00000005: F7E9    BRNE      AGAIN         ;repeat until COUNT = 0
+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 \$3FFFFF. So, it can cover the entire address space. See Figure 3-4.

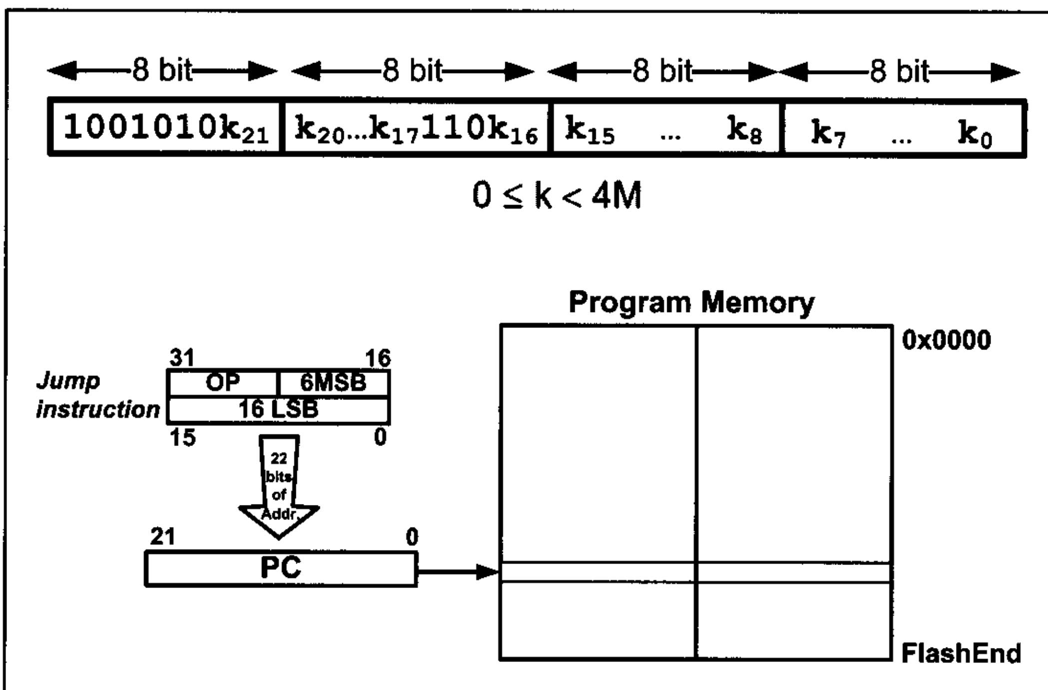


Figure 3-4. JMP Instruction

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.

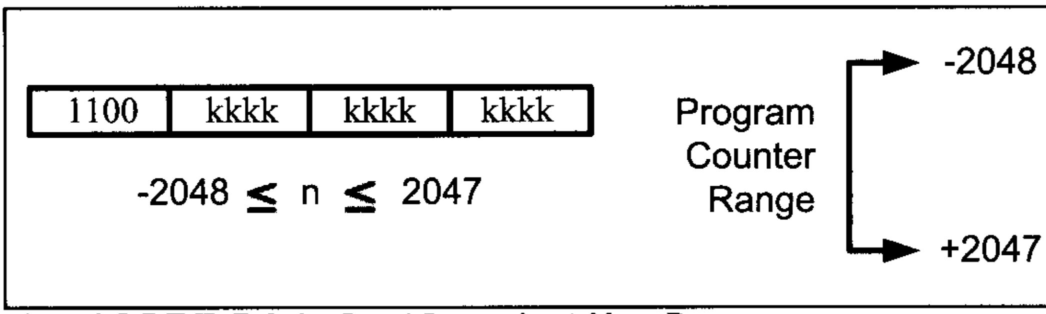


Figure 3-5. RJMP (Relative Jump) Instruction Address Range

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.

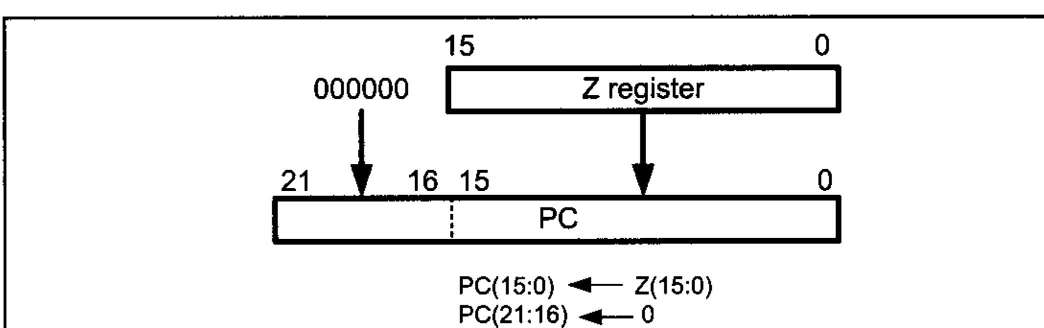


Figure 3-6. IJMP (Indirect Jump) Instruction Target Address

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, k21–k0, 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–$3FFFFF for the AVR, as shown in Figure 3-7.

1001	010k_{21}	$k_{20}k_{19}k_{18}k_{17}$	111k_{16}
$k_{15}k_{14}k_{13}k_{12}$	$k_{11}k_{10}k_9k_8$	$k_7k_6k_5k_4$	$k_3k_2k_1k_0$

$0 \leq k \leq 3FFFFF$

←8 bit→	←8 bit→	←8 bit→	←8 bit→
1001010k_{21}	k_{20}...k_{17}111k_{16}	k_{15} ... k_8	k_7 ... k_0

$0 \leq k < 4M$

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.

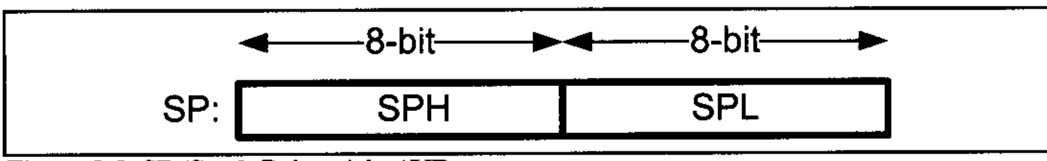


Figure 3-8. SP (Stack Pointer) in AVR

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 R20	R22	R31	SP	Stack
OUT SPL,R16	$0	$0	0	$085F	85D 85E 85F ← SP
LDI R22, 0x66	$21	$66	0	$085F	85D 85E 85F ← SP
PUSH R20	$21	$66	0	$085E	85D 85E ← SP 85F 21
PUSH R22	$21	$66	0	$085D	85D ← SP 85E 66 85F 21
LDI R22, 0	$0	$0	0	$085D	85D ← SP 85E 66 85F 21
POP R22	$0	$66	0	$085E	85D 85E ← SP 85F 21
POP R31	$0	$66	$21	$085F	85D 85E 85F ← SP

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

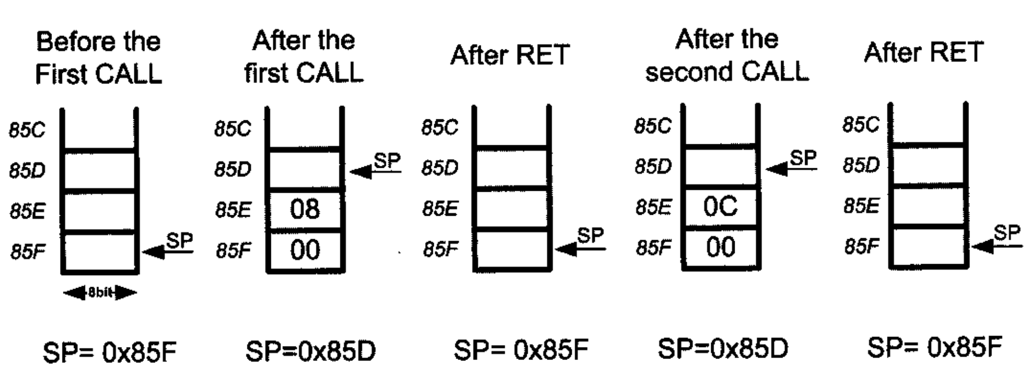
Analyze the stack for the CALL instructions in the following program.

```
            .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:

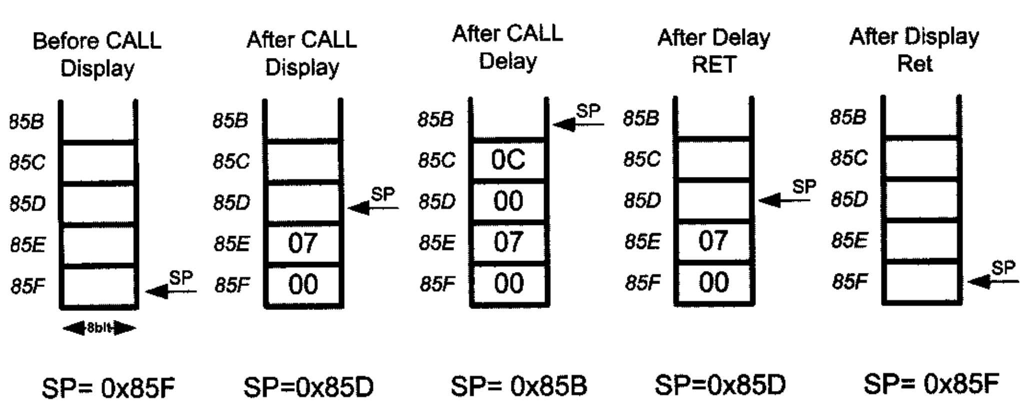
```
                  .INCLUDE "M32DEF.INC"

                  .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.

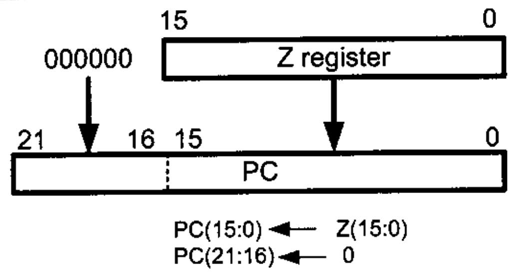


Figure 3-10. ICALL Instruction

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.

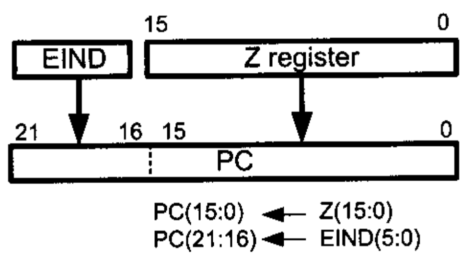


Figure 3-11. EICALL Instruction

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

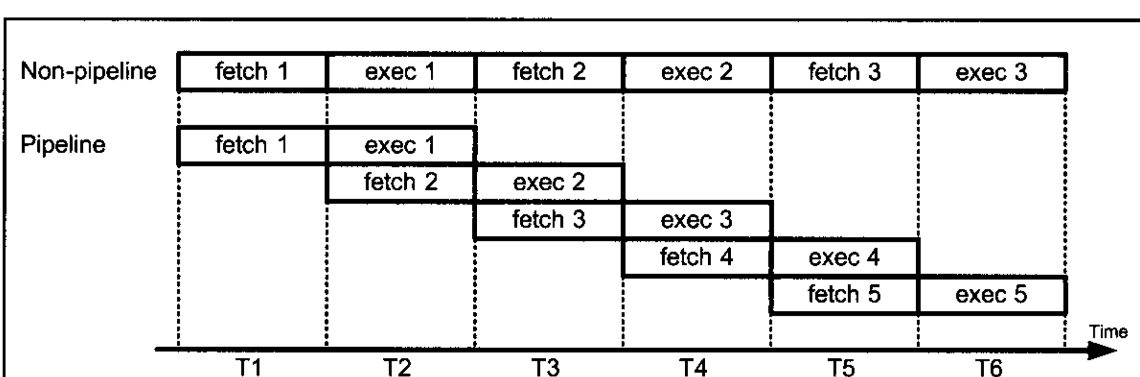


Figure 3-12. Pipeline vs. Non-pipeline

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.

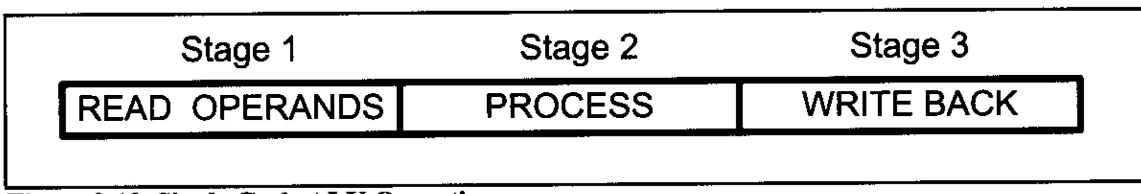


Figure 3-13. Single Cycle ALU Operation

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*.

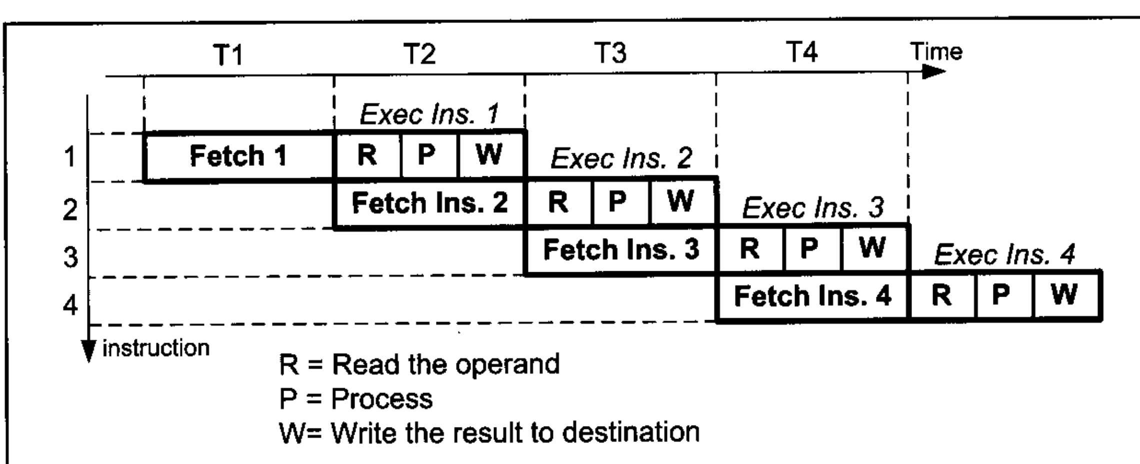


Figure 3-14. Pipeline Activity for Both Fetch and Execute

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 MHz = 0.125 μs (microsecond) = 125 ns (nanosecond)
(b) instruction cycle = 1/16 MHz = 0.0625 μs = 62.5 ns (nanosecond)
(c) instruction cycle = 1/10 MHz = 0.1 μs = 100 ns
(d) instruction cycle = 1/1 MHz = 1 μs

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 μs, as shown in Example 3-14. Appendix A shows instruction cycles for each of the above instructions. Therefore, we have:

Instruction	*Instruction cycles*	*Time to execute*
(a) LDI	1	1×1 μs = 1 μs
(b) DEC	1	1×1 μs = 1 μs
(c) OUT	1	1×1 μs = 1 μs
(d) ADD	1	1×1 μs = 1 μs
(e) NOP	1	1×1 μs = 1 μs
(f) JMP	3	3×1 μs = 2 μs
(g) CALL	4	4×1 μs = 4 μs
(h) BRNE	2/1	(2 μs taken, 1 μs 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:

```
                                     Instruction Cycles
        .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 $[1 + ((1 + 1 + 1 + 2) \times 255) + 4] \times 0.1$ μs = 128.0 μs. 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 μs.

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 $[1 + (255 \times 5) - 1 + 4] \times 1\ \mu s = 1279\ \mu s$.

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 μs (a) find the time delay in the following subroutine, and (b) find the amount of ROM it takes.

```
                                      Instruction Cycles
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 × 250) – 1] × 1 μs = 1249 μs. (We should subtract 1 for the times BRNE HERE falls through.) The AGAIN loop repeats the HERE loop 200 times; therefore, we have 200 × 1249 μs = 249,800 μs, 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 × 200) – 1] × 1 μs = 799 μs to the time delay. As a result we have 249,800 + 799 = 250,599 μs 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.

```
                                    Machine Cycles

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
         DEC     R16                1
         BRNE    AGAIN              2
         RET                        4
```

Solution:

The time delay inside the AGAIN loop is $[200(13 + 2)] \times 1\ \mu s = 3000\ \mu s$. 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 MHz = 125 ns

Delay = $32 \times 200 \times 250 \times 5 \times 125$ ns = 1,000,000,000 ns = 1,000,000 μs = 1 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 μs.
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. Decremented
5. Incremented
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 MHz / 4 = 3 MHz, and MC = 1/3 MHz = 0.333 μs
5. $[100 \times (1 + 1 + 1 + 1 + 1 + 1 + 2)] \times 1\ \mu s = 800\ \mu s = 0.8$ milliseconds
6. False. It takes 4 clocks.
7. Machine cycle = 1 / 8 MHz = 0.125 μs = 125 ns
8. True
9. True
10. False. Only if it branches to the target address.