AVR Microcontroller

Microprocessor Course

Third Chapter

BRANCH, CALL, AND TIME DELAY LOOP

Aban 1393

BRANCH, CALL, AND TIME DELAY LOOP

3.1 BRANCH INSTRUCTIONS AND LOOPING

Loop is Repeating a sequence of instructions or an operation a certain number of times. 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 = OxOC)
ADD R16,R17     ;add value 3 to R16 (R16 = OxOF)
ADD R16,R17     ;add value 3 to R16 (R16 = 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.

BRANCH, CALL, AND TIME DELAY LOOP

3.1 BRANCH INSTRUCTIONS AND LOOPING

Using BRNE instruction for looping

A much better way is to use a loop. 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
```


mashhoun@iust.ac.ir Iran Univ of Science & Tech

BRANCH, CALL, AND TIME DELAY LOOP

3.1 BRANCH INSTRUCTIONS AND LOOPING

Using BRNE instruction for looping

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
```

BRANCH, CALL, AND TIME DELAY LOOP

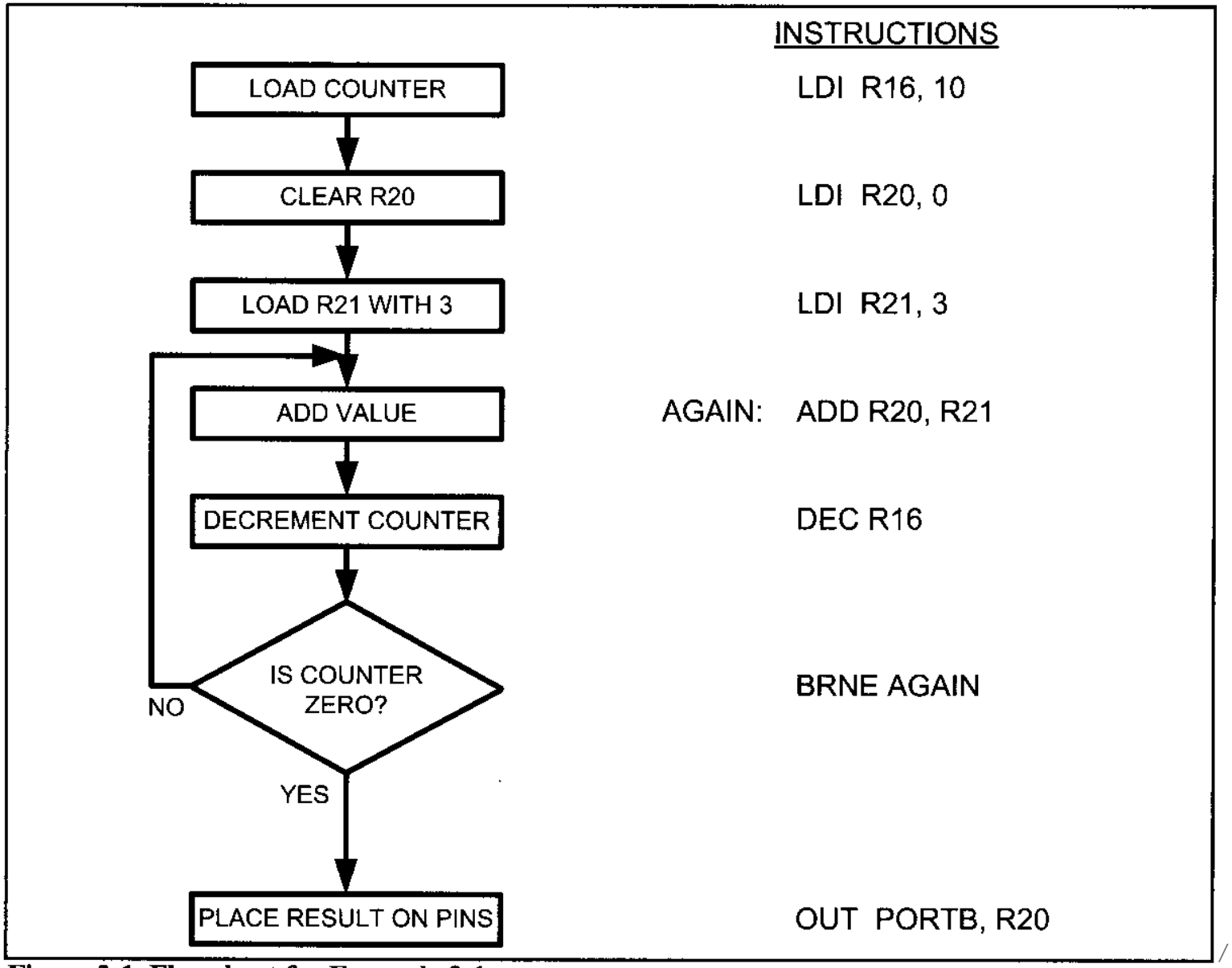


Figure 3-1. Flowchart for Example 3-1

BRANCH, CALL, AND TIME DELAY LOOP
3.1 BRANCH INSTRUCTIONS AND LOOPING

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.

BRANCH, CALL, AND TIME DELAY LOOP

3.1 BRANCH INSTRUCTIONS AND LOOPING

Loop inside a loop

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.

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.

BRANCH, CALL, AND TIME DELAY LOOP

3.1 BRANCH INSTRUCTIONS AND LOOPING

Loop inside a loop

```
.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.

BRANCH, CALL, AND TIME DELAY LOOP

3.1 BRANCH INSTRUCTIONS AND LOOPING

Looping 100,000 times

Because two registers give us a maximum value of 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
```

BRANCH, CALL, AND TIME DELAY LOOP
3.1 BRANCH INSTRUCTIONS AND LOOPING

Other conditional jumps

In Table 3-1 you see some of the conditional branches for the AVR. In Table 3-1 notice that the instructions, such as BREQ (Branch if Z = 1) and BRLO (Branch if C = I), 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

BRANCH, CALL, AND TIME DELAY LOOP

3.1 BRANCH INSTRUCTIONS AND LOOPING

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.

BRANCH, CALL, AND TIME DELAY LOOP

3.1 BRANCH INSTRUCTIONS AND LOOPING

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

In executing "BRSH", 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 will execute the next instruction below BRSH.

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: ...
```

BRANCH, CALL, AND TIME DELAY LOOP

3.1 BRANCH INSTRUCTIONS AND LOOPING

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

BRANCH, CALL, AND TIME DELAY LOOP
3.1 BRANCH INSTRUCTIONS AND LOOPING

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

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.

BRANCH, CALL, AND TIME DELAY LOOP
3.1 BRANCH INSTRUCTIONS AND LOOPING

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.

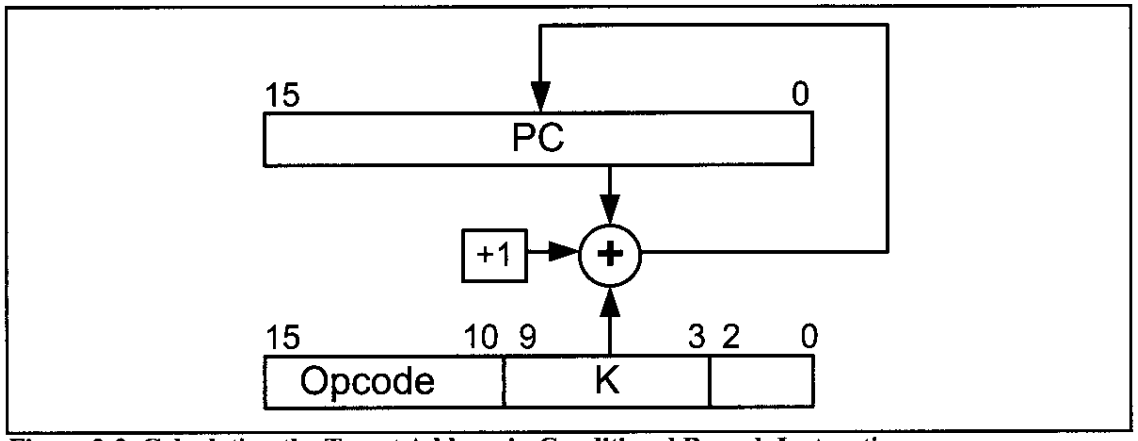


Figure 3-3. Calculating the Target Address in Conditional Branch Instructions

BRANCH, CALL, AND TIME DELAY LOOP

3.1 BRANCH INSTRUCTIONS AND LOOPING

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. 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.

BRANCH, CALL, AND TIME DELAY LOOP
3.1 BRANCH INSTRUCTIONS AND LOOPING

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).

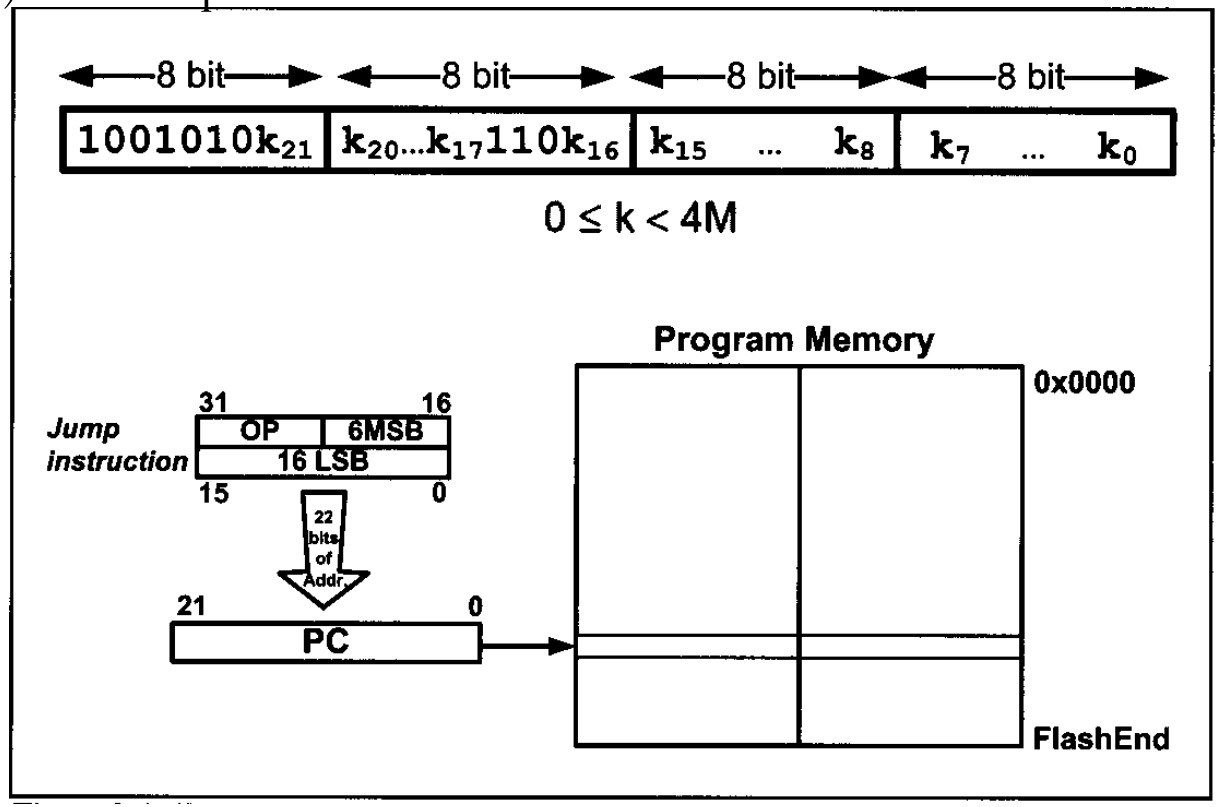
Deciding which one to use depends on the target address.

BRANCH, CALL, AND TIME DELAY LOOP

3.1 BRANCH INSTRUCTIONS AND LOOPING

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.



mashhoun@iust.ac.ir Iran Univ of Science & Tech

BRANCH, CALL, AND TIME DELAY LOOP

3.1 BRANCH INSTRUCTIONS AND LOOPING

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.

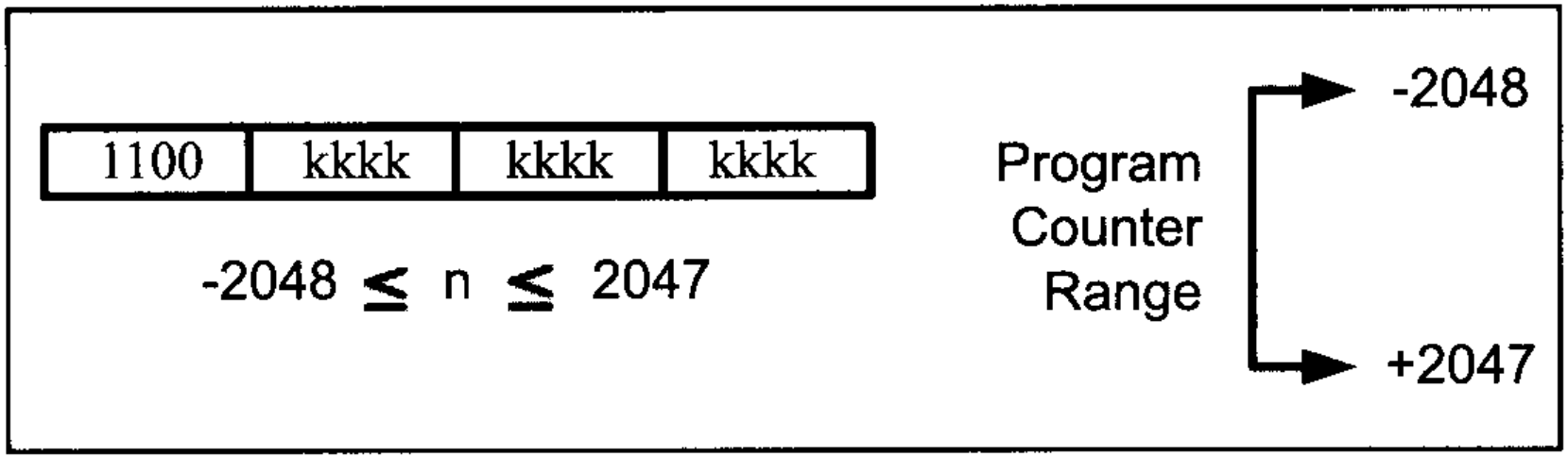


Figure 3-5. RJMP (Relative Jump) Instruction Address Range

BRANCH, CALL, AND TIME DELAY LOOP

3.1 BRANCH INSTRUCTIONS AND LOOPING

IJMP (indirect jump)

IMP 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.

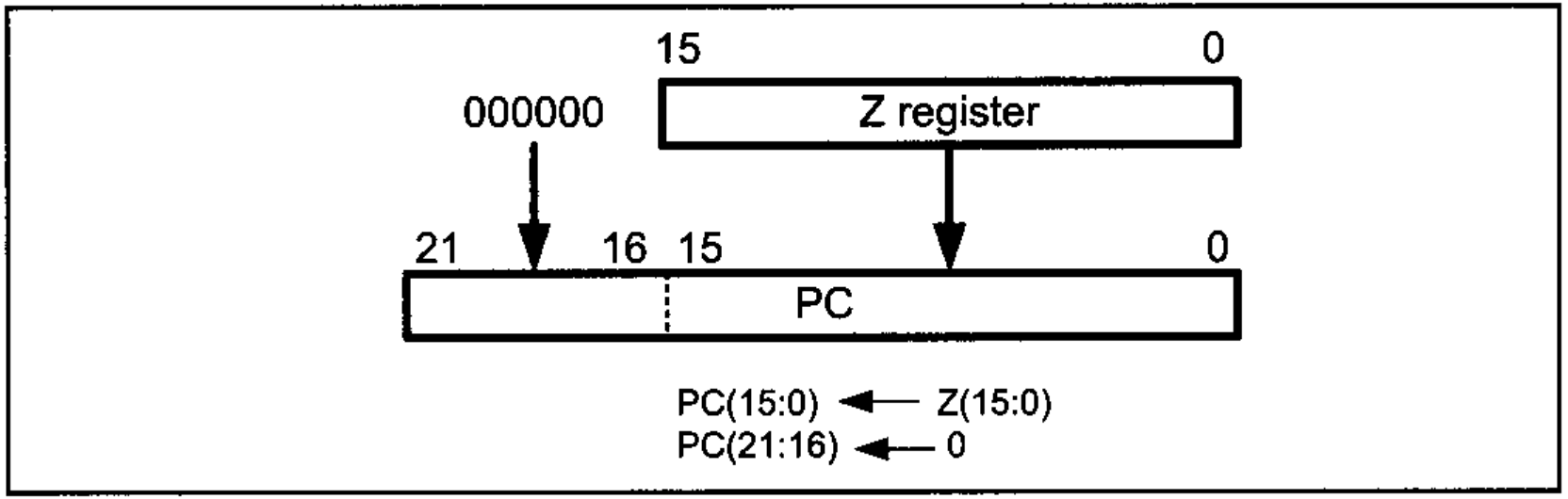


Figure 3-6. IJMP (Indirect Jump) Instruction Target Address

BRANCH, CALL, AND TIME DELAY LOOP

3.2: CALL INSTRUCTIONS AND STACK

CALL INSTRUCTIONS

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 4 to Z), and
- EICALL (extended indirect *call to Z).*

The choice of which one to use depends on the target address.

BRANCH, CALL, AND TIME DELAY LOOP

3.2: CALL INSTRUCTIONS AND STACK

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.

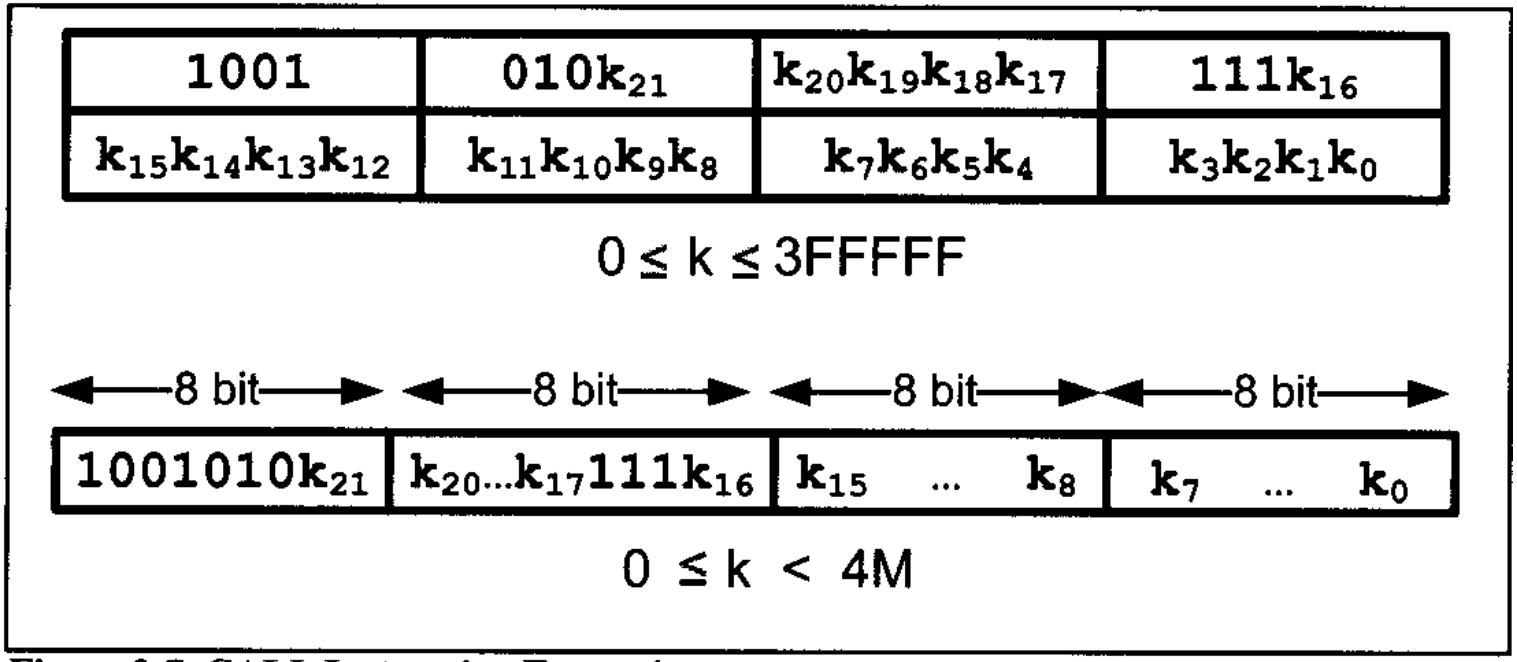


Figure 3-7. CALL Instruction Formation

BRANCH, CALL, AND TIME DELAY LOOP

3.2: CALL INSTRUCTIONS AND STACK

How stacks are accessed in the AVR

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.

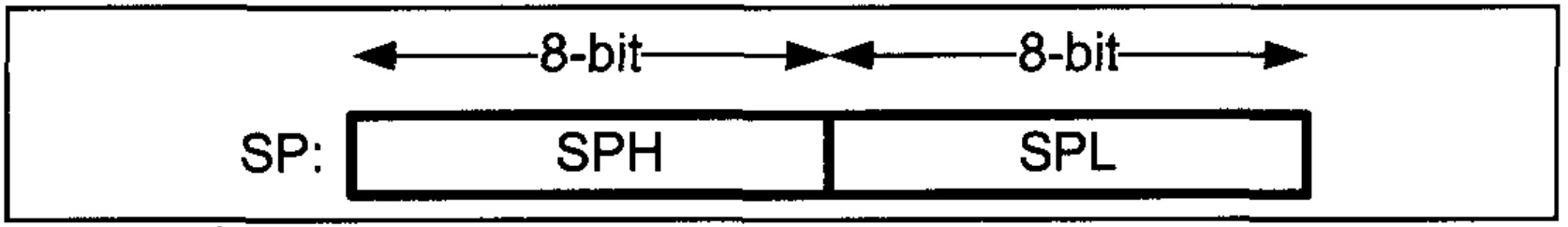


Figure 3-8. SP (Stack Pointer) in AVR

The stack pointer 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.

BRANCH, CALL, AND TIME DELAY LOOP

3.2: CALL INSTRUCTIONS AND STACK

Pushing onto the stack

As we push data onto the stack, the data are saved where the SP points to, and the SP is decremented by one.

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
```


mashhoun@iust.ac.ir Iran Univ of Science & Tech

BRANCH, CALL, AND TIME DELAY LOOP

3.2: CALL INSTRUCTIONS AND STACK

Popping from the stack

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 (RObR31)
```

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

```
POP   R16    ;inrement SP, and then load top of Stack to R16
```

BRANCH, CALL, AND TIME DELAY LOOP

3.2: CALL INSTRUCTIONS AND STACK

Example 3-8

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

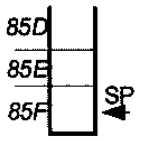
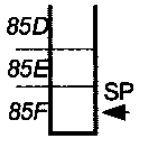
```
.INCLUDE "M32DEF.INC"
      .ORG 0
      ;initialize the SP to point to the last location of RAM (RAMEND)
      LDI   R16, HIGH(RAMEND)          ;load SPH
      OUT   SPH, R16
      LDI   R16, LOW(RAMEND)           ;load SPL
      OUT   SPL, R16

      LDI   R31, 0
      LDI   R20, 0x21
      LDI   R22, 0x66

      PUSH   R20
      PUSH   R22

      LDI    R20, 0
      LDI    R22, 0

      POP    R22
      POP    R31
```

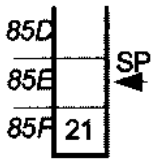
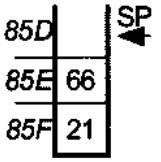
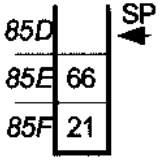
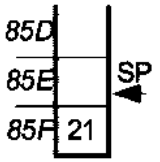
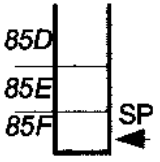
After the execution of	Contents of some of the registers				Stack
	R20	R22	R31	SP	
OUT SPL,R16	$0	$0	0	$085F	
LDI R22, 0x66	$21	$66	0	$085F	

mashhoun@iust.ac.ir Iran Univ of Science & Tech

BRANCH, CALL, AND TIME DELAY LOOP

3.2: CALL INSTRUCTIONS AND STACK

Solution:

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	

mashhoun@iust.ac.ir Iran Univ of Science & Tech

BRANCH, CALL, AND TIME DELAY LOOP

3.2: CALL INSTRUCTIONS AND STACK

Initializing the stack pointer

When the AVR is powered up, the SP register contains the value 0. Therefore, we must initialize the SP at the beginning of the program so that it points to somewhere in the internal SRAM. 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.

```
LDI R16, HIGH(RAMEND)          ;load SPH
OUT SPH, R16
LDI R16, LOW(RAMEND)           ;load SPL
OUT SPL, R16
```

BRANCH, CALL, AND TIME DELAY LOOP

3.2: CALL INSTRUCTIONS AND STACK

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.

For the AVRs whose program counter is not longer than 16 bits (e.g., ATmegal28, 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.

BRANCH, CALL, AND TIME DELAY LOOP

3.2: CALL INSTRUCTIONS AND STACK

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. 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.

BRANCH, CALL, AND TIME DELAY LOOP

3.2: CALL INSTRUCTIONS AND STACK

Example 3-9

Toggle all bits of Port B by sending to it the value 0x55 and 0xAA continuously. Put a time delay between each issuing of data to Port B.

```
1  .INCLUDE "M32DEF.INC"
2  .ORG 0
3          LDI     R16,HIGH(RAMEND)
4          OUT     SPH,R16
5          LDI     R16,LOW(RAMEND)
6          OUT     SPL,R16
7
8          LDI     R16,0xFF
9          OUT     DDRB,R16
10 BACK:
11         LDI     R16,0x55
12         OUT     PORTB,R16
13         CALL    DELAY1
14         LDI     R16,0xAA
15         OUT     PORTB,R16
16         CALL    DELAY1
17         RJMP    BACK
```

```
22 AGAIN:
23         NOP
24         NOP
25         DEC     R20
26         BRNE    AGAIN
27         RET
28
29 DELAY1:
30         LDI     R20,0xFF
31 AGAIN1: LDI     R21,0xFF
32 AGAIN2:
33         NOP
34         NOP
35         DEC     R21
36         BRNE    AGAIN2
37         DEC     R20
38         BRNE    AGAIN1
39         RET
```

BRANCH, CALL, AND TIME DELAY LOOP

3.2: CALL INSTRUCTIONS AND STACK

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.

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
```

BRANCH, CALL, AND TIME DELAY LOOP

3.2: CALL INSTRUCTIONS AND STACK

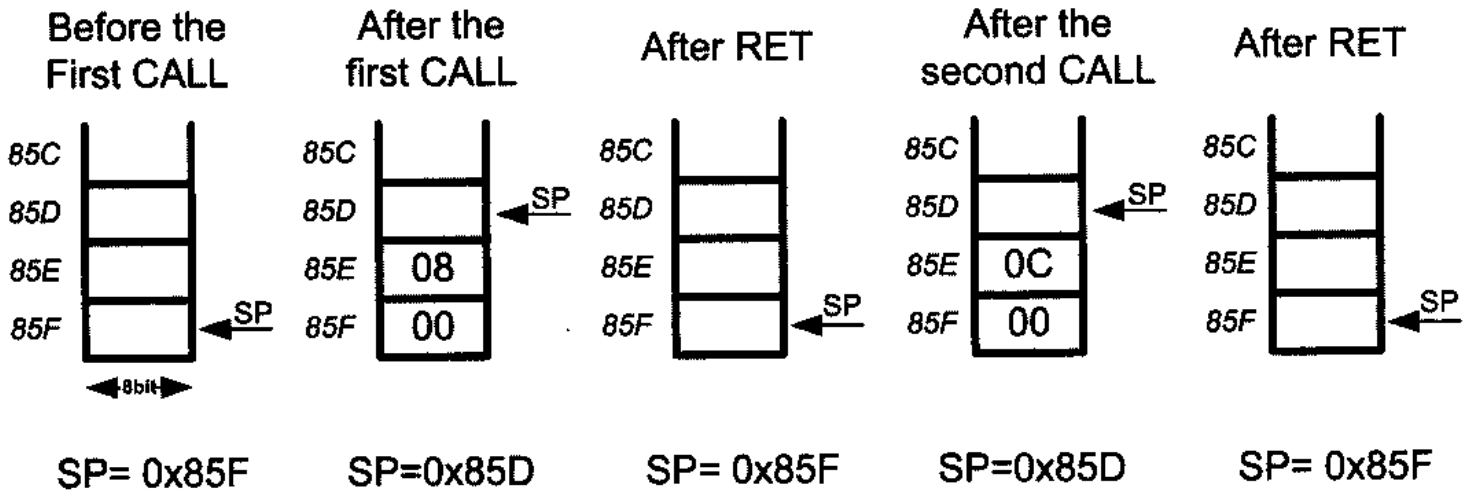
```
            BACK:
+00000004:      LDI   R16,0x55      ;load R16 with 0x55
+00000005:      OUT   PORTB,R16     ;send 55H to port B
+00000006:      CALL  DELAY         ;time delay
+00000008:      LDI   R16,0xAA      ;load R16 with 0xAA
+00000009:      OUT   PORTB,R16     ;send 0xAA to port B
+0000000A:      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
```

BRANCH, CALL, AND TIME DELAY LOOP

3.2: CALL INSTRUCTIONS AND STACK

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.



BRANCH, CALL, AND TIME DELAY LOOP

3.2: CALL INSTRUCTIONS AND STACK

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. 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.

```
.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

BRANCH, CALL, AND TIME DELAY LOOP

3.2: CALL INSTRUCTIONS AND STACK

RCALL (relative call)

RCALL is a 2-byte instruction in contrast to CALL, which is 4 bytes.

1101	kkkk	kkkk	kkkk

Write a program to count up from 00 to 0xFF 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.

BRANCH, CALL, AND TIME DELAY LOOP

3.2: CALL INSTRUCTIONS AND STACK

```
1   .INCLUDE "M32DEF.INC"
2   .DEF COUNT = R20
3   .ORG 0
4           LDI     R16,HIGH(RAMEND)
5           OUT     SPH,R16
6           LDI     R16,LOW(RAMEND)
7           OUT     SPL,R16
8
9           LDI     R16,0xFF
10          OUT     DDRB,R16
11          LDI     COUNT,0
12  BACK:
13          CALL    DISPLAY
14          RJMP    BACK
```

```
16  DISPLAY:
17          INC     COUNT
18          OUT     PORTB,COUNT
19          CALL    DELAY
20          RET
21
22  ;DELAY SUBROUTINE
23  .ORG    300
24  DELAY:
25          LDI     R16,0xFF
26  AGAIN1: LDI     R17,0xFF
27  AGAIN2:
28          NOP
29          NOP
30          DEC     R17
31          BRNE    AGAIN2
32          DEC     R16
33          BRNE    AGAIN1
34          RET
```


mashhoun@iust.ac.ir Iran Univ of Science & Tech

BRANCH, CALL, AND TIME DELAY LOOP

3.2: CALL INSTRUCTIONS AND STACK

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

```
1   .INCLUDE "M32DEF.INC"
2   .ORG 0
3           LDI     R16,HIGH(RAMEND)
4           OUT     SPH,R16
5           LDI     R16,LOW(RAMEND)
6           OUT     SPL,R16
7
8           LDI     R16,0xFF
9           OUT     DDRB,R16
10          LDI     R16,0x55
11  BACK:
12          COM     R16
13          OUT     PORTB,R16
14          RCALL   DELAY
15          RJMP    BACK
```

```
17  ;DELAY SUBROUTINE
18  DELAY:
19          LDI     R20,0xFF
20  AGAIN1: LDI     R21,0xFF
21  AGAIN2:
22          NOP
23          NOP
24          DEC     R21
25          BRNE    AGAIN2
26          DEC     R20
27          BRNE    AGAIN1
28          RET
```


mashhoun@iust.ac.ir Iran Univ of Science & Tech

BRANCH, CALL, AND TIME DELAY LOOP

3.2: CALL INSTRUCTIONS AND STACK

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.

BRANCH, CALL, AND TIME DELAY LOOP

3.2: CALL INSTRUCTIONS AND STACK

ICALL (indirect call)

In this 2-byte (16-bit) instruction, the Z register specifies the target address.

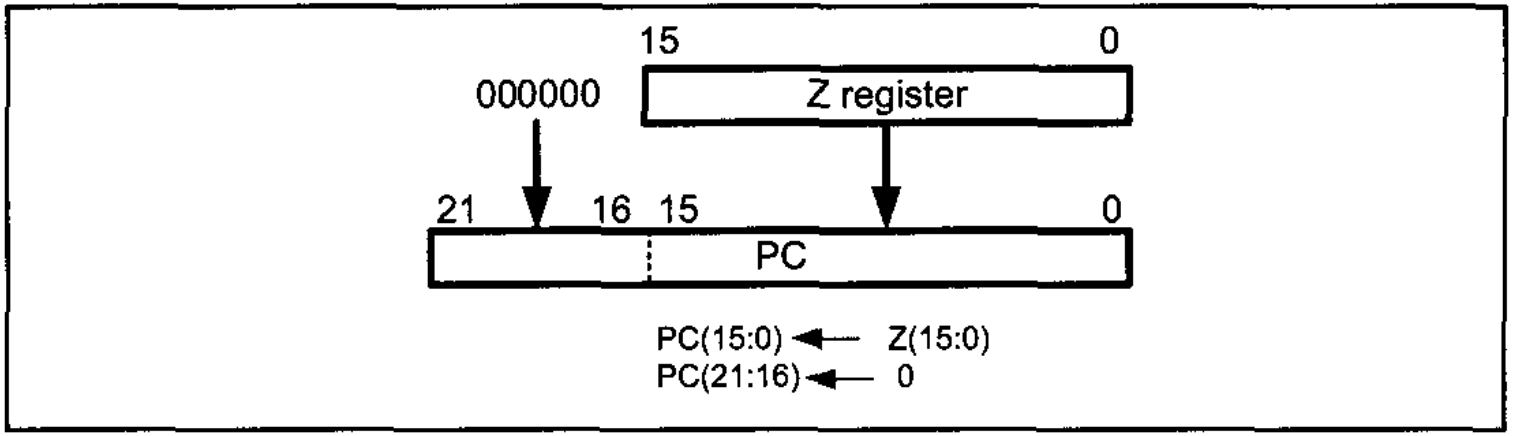


Figure 3-10. ICALL Instruction

mashhoun@iust.ac.ir Iran Univ of Science & Tech

BRANCH, CALL, AND TIME DELAY LOOP

3.2: CALL INSTRUCTIONS AND STACK

ICALL (indirect call)

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.

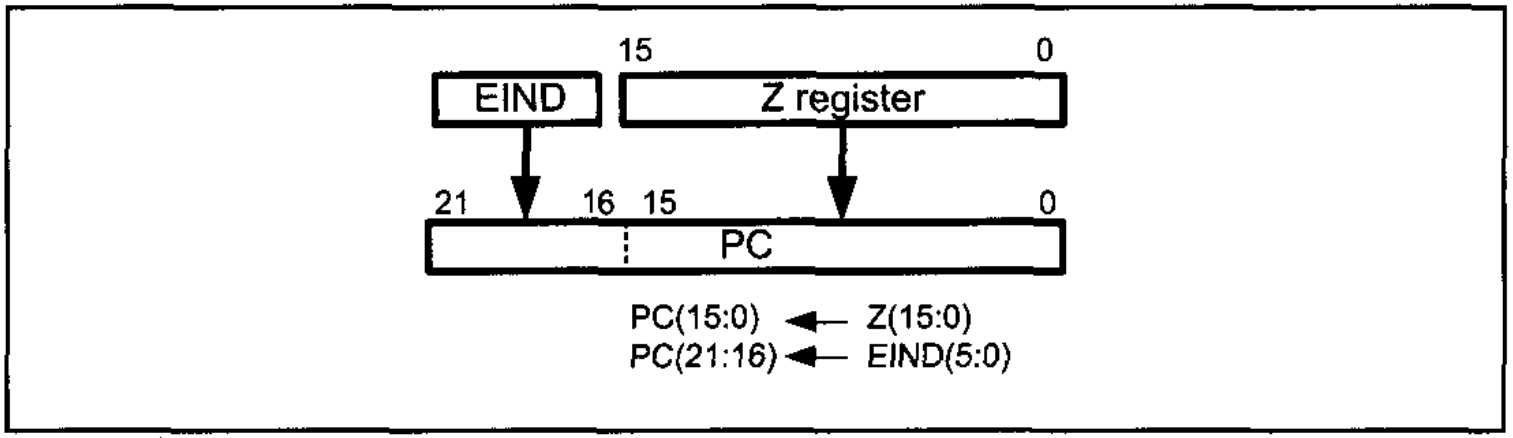


Figure 3-11. EICALL Instruction

BRANCH, CALL, AND TIME DELAY LOOP

3.3: AVR TIME DELAY AND INSTRUCTION PIPELINE

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 XTALl and XTAL2 input pins is one factor in the time delay calculation.
2. The AVR design: 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. 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.

BRANCH, CALL, AND TIME DELAY LOOP

3.3: AVR TIME DELAY AND INSTRUCTION PIPELINE

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.

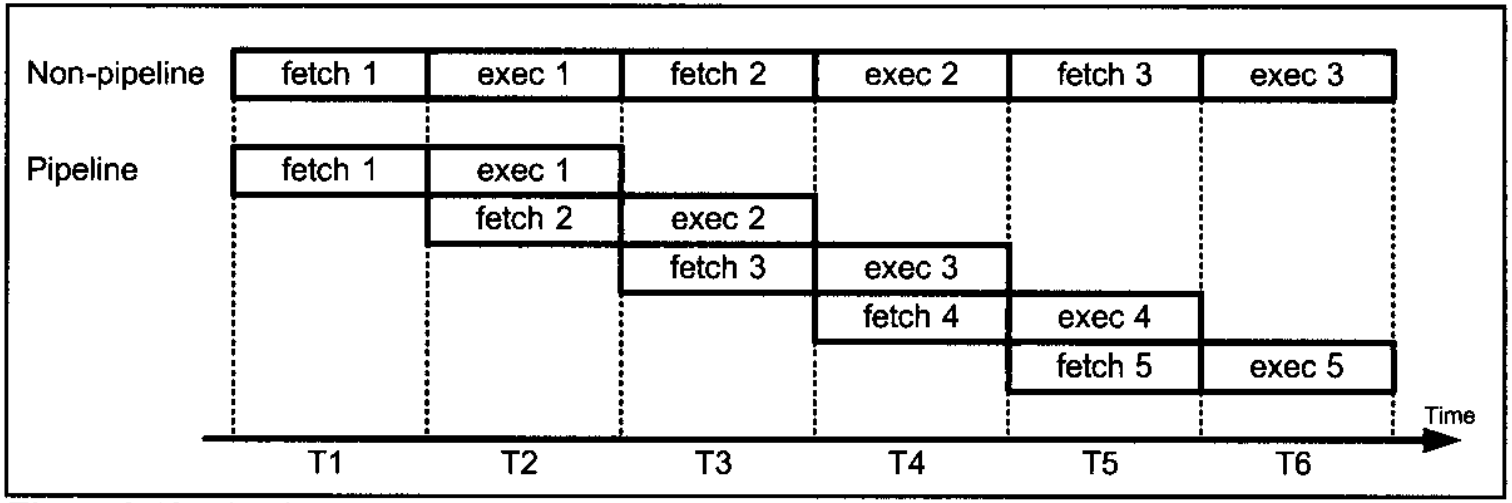


Figure 3-12. Pipeline vs. Non-pipeline

BRANCH, CALL, AND TIME DELAY LOOP

3.3: AVR TIME DELAY AND INSTRUCTION PIPELINE

Pipelining

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.

BRANCH, CALL, AND TIME DELAY LOOP

3.3: AVR TIME DELAY AND INSTRUCTION PIPELINE

AVR multistage execution pipeline

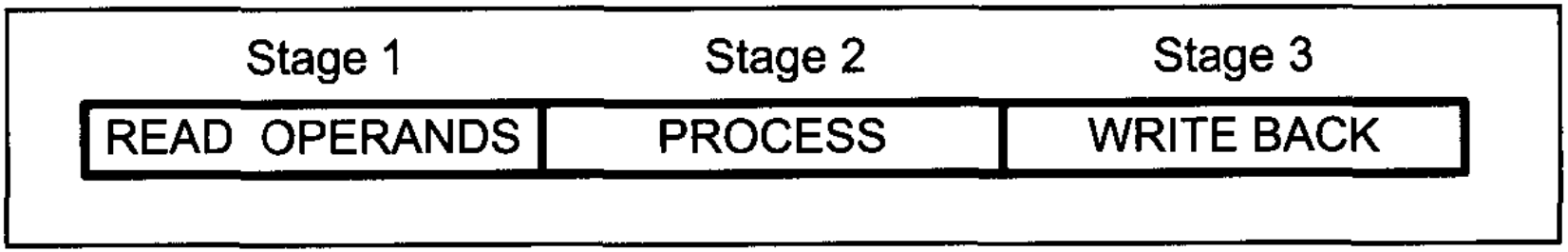


Figure 3-13. Single Cycle ALU Operation

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.

BRANCH, CALL, AND TIME DELAY LOOP

3.3: AVR TIME DELAY AND INSTRUCTION PIPELINE

AVR multistage execution pipeline

In reality, one can construct the AVR pipeline for three instructions.

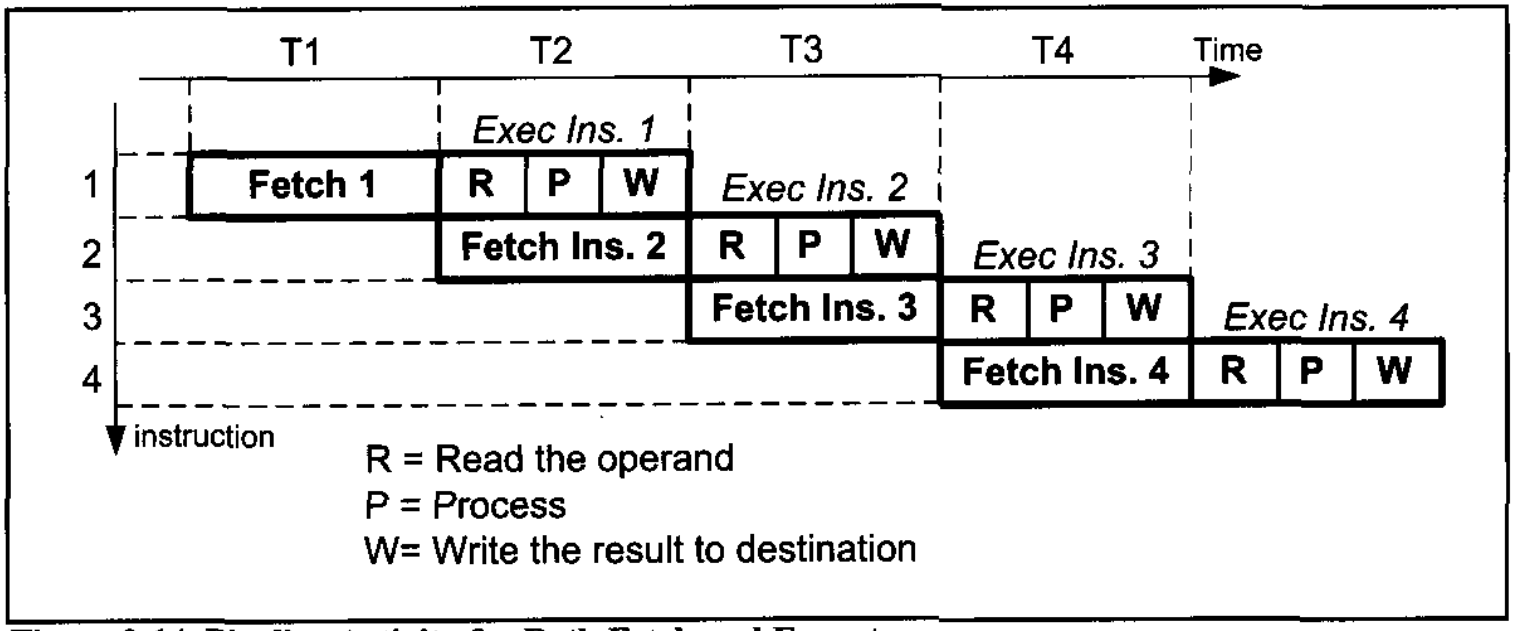


Figure 3-14. Pipeline Activity for Both Fetch and Execute

BRANCH, CALL, AND TIME DELAY LOOP

3.3: AVR TIME DELAY AND INSTRUCTION PIPELINE

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*. Most instructions take no more than one or two machine cycles to execute.

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, CALL, AND TIME DELAY LOOP

3.3: AVR TIME DELAY AND INSTRUCTION PIPELINE

Branch penalty

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.

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.

BRANCH, CALL, AND TIME DELAY LOOP

3.3: AVR TIME DELAY AND INSTRUCTION PIPELINE

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)

BRANCH, CALL, AND TIME DELAY LOOP

3.3: AVR TIME DELAY AND INSTRUCTION PIPELINE

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 $[1 + ((1 + 1 + 1 + 2) \times 255) + 4] \times 0.1\ \mu s = 128.0\ \mu s$. 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 μs.

BRANCH, CALL, AND TIME DELAY LOOP

3.3: AVR TIME DELAY AND INSTRUCTION PIPELINE

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.**

BRANCH, CALL, AND TIME DELAY LOOP

3.3: AVR TIME DELAY AND INSTRUCTION PIPELINE

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
```

BRANCH, CALL, AND TIME DELAY LOOP

3.3: AVR TIME DELAY AND INSTRUCTION PIPELINE

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

BRANCH, CALL, AND TIME DELAY LOOP

3.3: AVR TIME DELAY AND INSTRUCTION PIPELINE

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***.

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
```

(a) For the HERE loop, we have **[(5 x 250) - 11 x 1 μs = 1249 μs.** The AGAIN loop repeats the HERE loop 200 times; therefore, we have 200 **x 1249 ps = 249,800 ps, if we do not include the overhead.**

BRANCH, CALL, AND TIME DELAY LOOP

3.3: AVR TIME DELAY AND INSTRUCTION PIPELINE

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

$$[(4 \times 200) - 11 \textbf{ x 1 µs = 799 µs}$$

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 R16,020" 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.

BRANCH, CALL, AND TIME DELAY LOOP

3.3: AVR TIME DELAY AND INSTRUCTION PIPELINE

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
```


mashhoun@iust.ac.ir Iran Univ of Science & Tech

BRANCH, CALL, AND TIME DELAY LOOP

3.3: AVR TIME DELAY AND INSTRUCTION PIPELINE

Solution:

The time delay inside the AGAIN loop is [200(13 + 2)] × 1 μs = 3000 μs. 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.