Branch, Call and Time Delay Loop

Hoda Roodaki
hroodaki@kntu.ac.ir

Branch, Call and Time Delay Loop

- The instructions in AVR to transfer program control to a different location.

Looping in AVR

- Repeating a sequence of instructions or an operation a certain number of times is called a *loop.*
- 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)
```

  - Too much code space would be needed to increase the number of repetitions to 50 or 100

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 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.

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
```

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, 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.

Example 3-3

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

Solution:

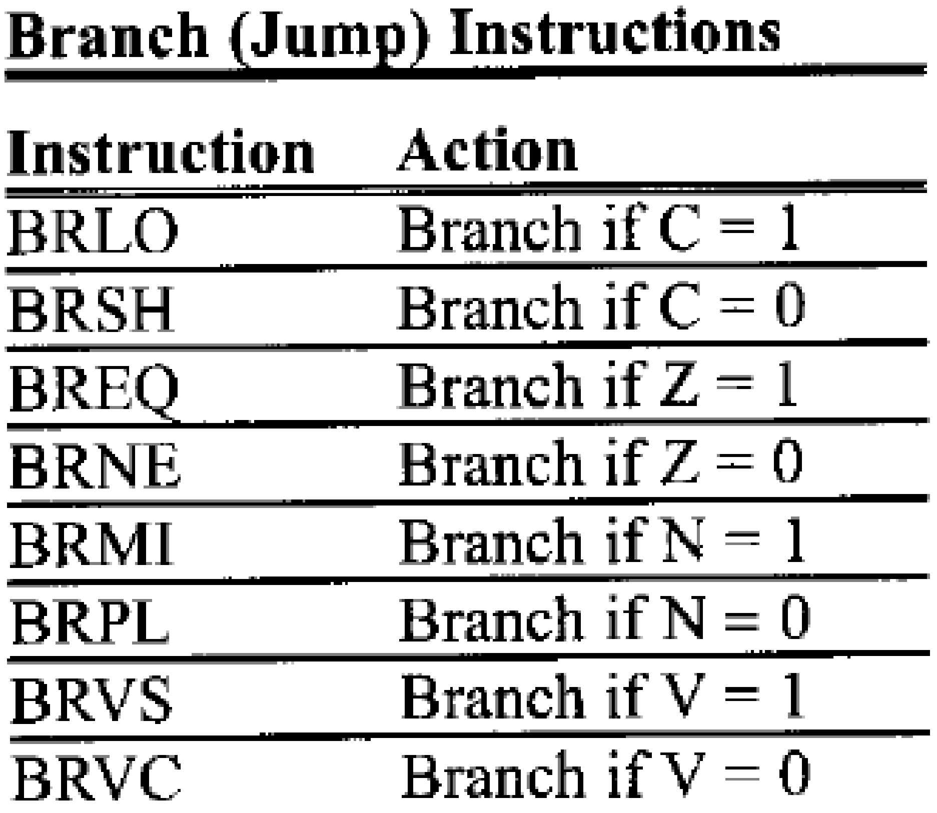
Because 700 is larger than 255 (the maximum capacity of any general purpose register), we use two registers to hold the count. The following code shows how to use R20 and R21 as a register for counters.

```
.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
```

Looping 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



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)

- The Z flag is checked.
- If it is high, the CPU jumps to the target address.

```
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

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:   ...
```

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 will execute the next instruction below BRSH.

Example 3-5

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

All conditional branches are short jumps

- 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.

Example 3-6

Example 3-7

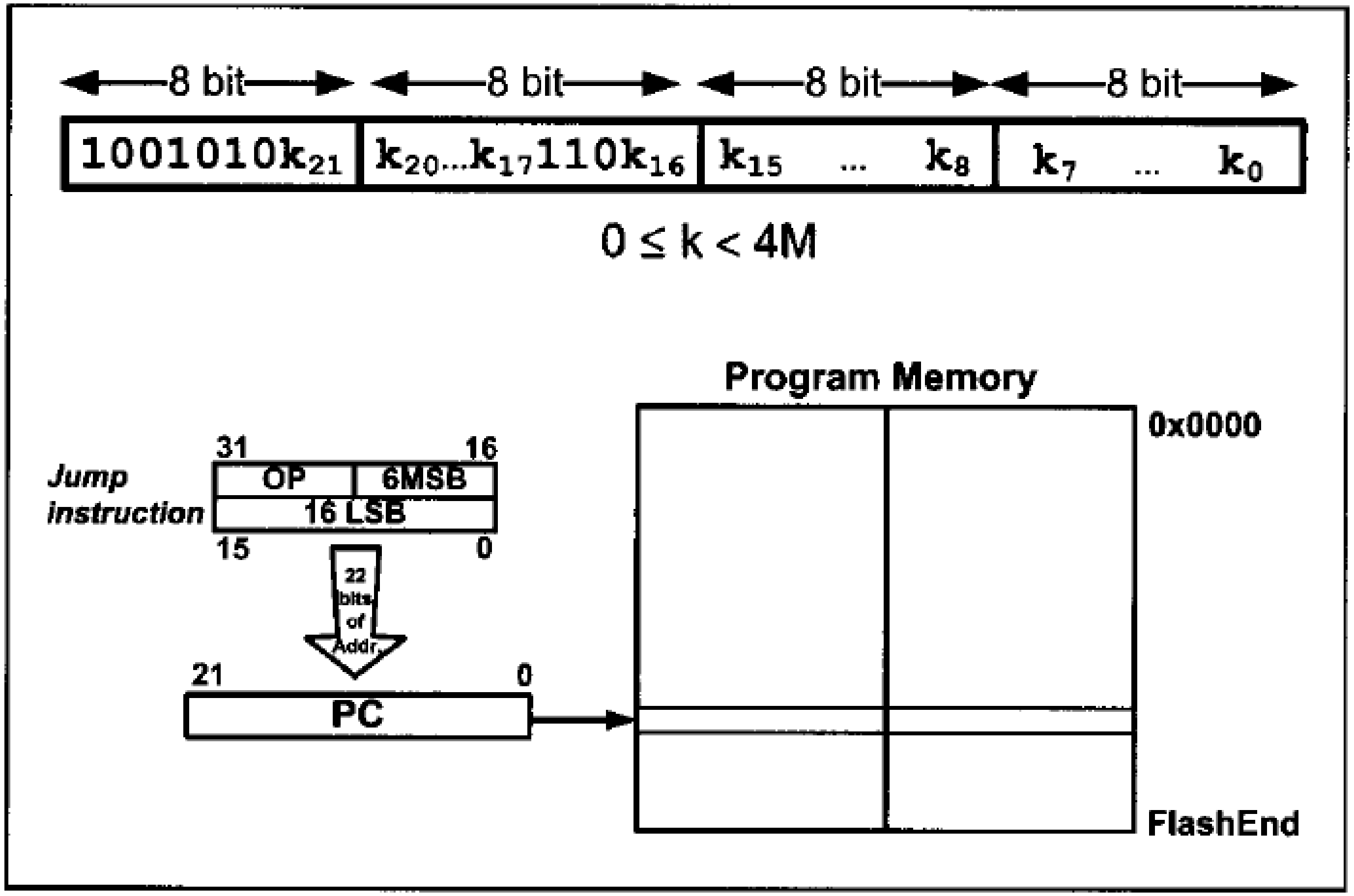
Unconditional branch instruction

- The unconditional branch is a jump in which control is transferred unconditionally to the target location.
- In the AVR there are three unconditional branch-es:
  - JMP (jump)
  - RJMP (relative jump)
  - IJMP (indirect jump)
- Deciding which one to use depends on the target address.

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.

JMP (JMP is a long jump)



RJMP (relative jump)

- 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.

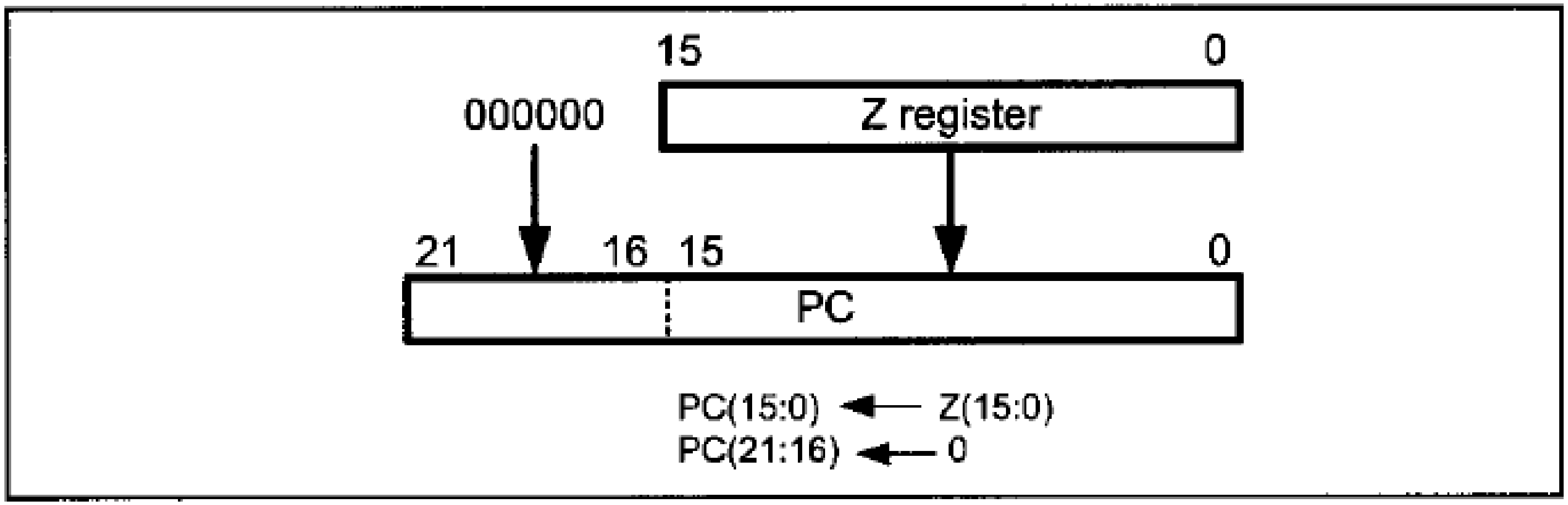
RJMP (relative jump)

- 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 + 204 7 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. l
  - In this regard, RJMP is like the conditional branch instructions except that 12 bits are used for the offset address instead of 7.

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.
- Z is a 2-byte register, so IJMP can jump within the lowest 64K words of the program memory.
  - 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.

IJMP (indirect jump)



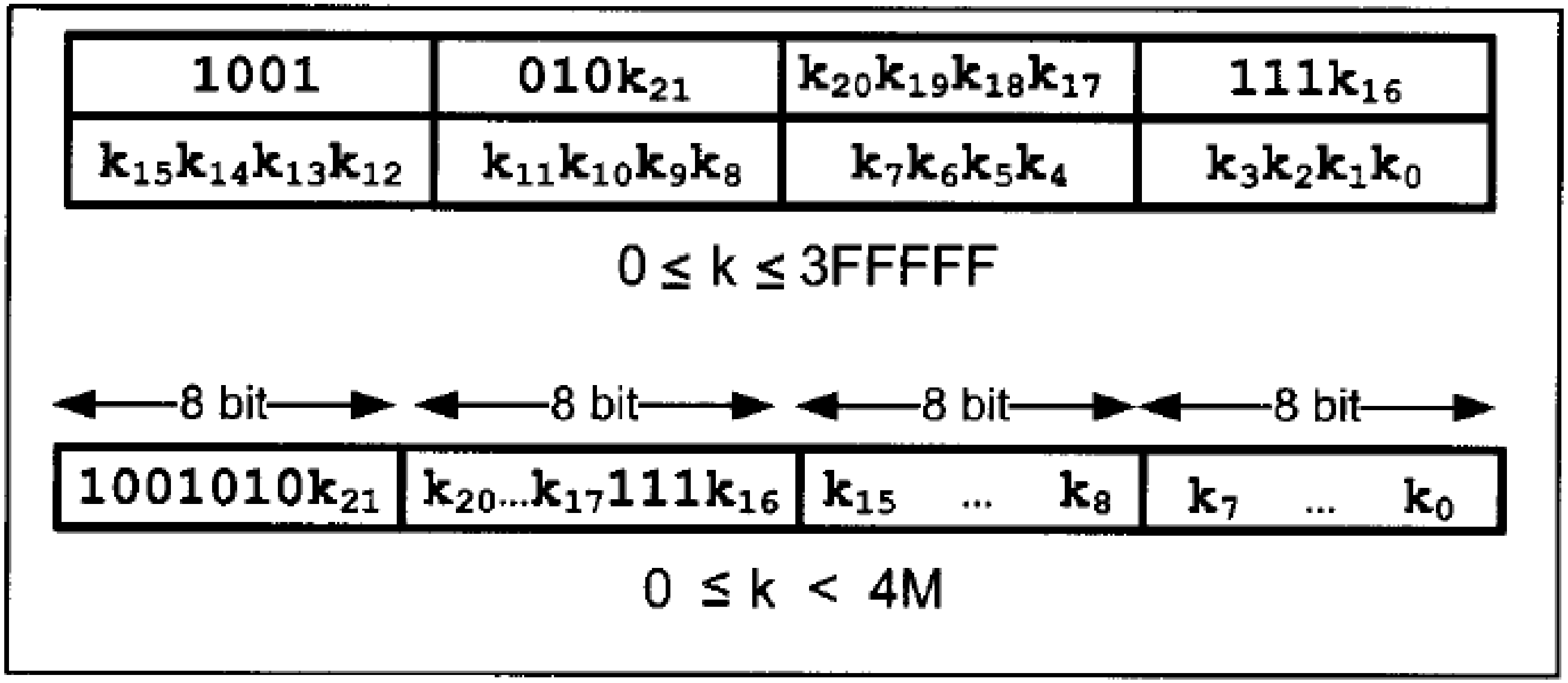
CALL INSTRUCTIONS AND STACK

- 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)
  - EICALL (extended indirect call to Z)
- The choice of which one to use depends on the target address.

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.
- Therefore, CALL can be used to call subroutines located anywhere within the 4M address space of 000000-$3FFFFF for the AVR.

CALL



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

$0 \le k \le 3FFFFF$

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

$0 \le k < 4M$

CALL

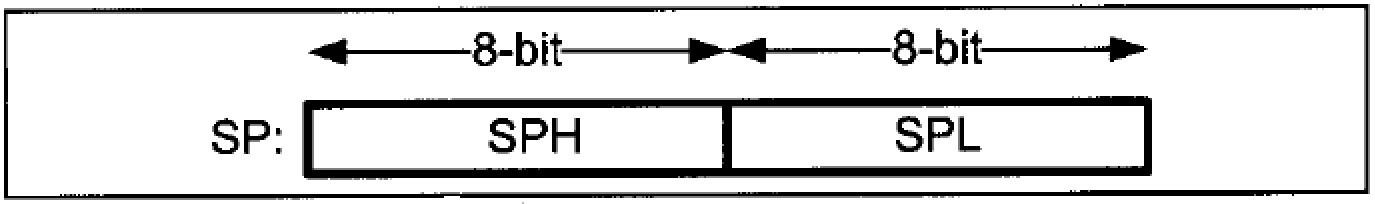
- To make sure that the AVR knows where to come back to after execution of the called subroutine, the microcontroller automatically saves on the stack the address of the instruction immediately below the CALL.
- When a subroutine is called, control is transferred to that subroutine, and the processor 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)
  - SPH (the high byte of the SP)



How stacks are accessed in the AVR

- The stack pointer must be wide enough to address all the RAM.
  - In the AVRs with more than 256 bytes of memory the SP is made of two 8-bit registers (SPL and SPH)
  - 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.
- The loading of stack contents back into a CPU register is called a POP.
  - A register is pushed onto the stack to save it and popped off the stack to retrieve it.

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.

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

```
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

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

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

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

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

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.
- 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., 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.
- 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.

The upper limit of the stack

- We can define the stack anywhere in the general purpose memory.
  - In the AVR the stack can be as big as its RAM.
  - 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.
  - 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 3-9

Example 3-10

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.
- This allows you to make each subroutine into a separate module.

Calling many subroutines from the main program

```
;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
```

Example 3-11

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.
- 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.

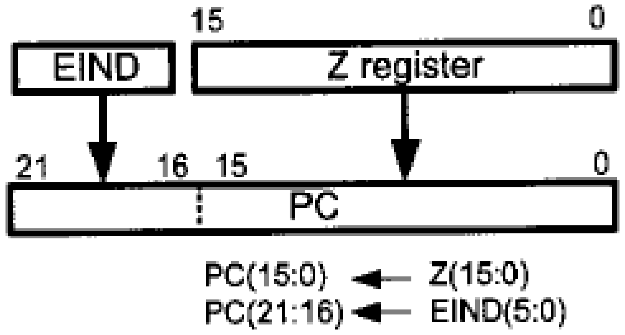
Example 3-12

ICALL (indirect call)

- 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 JMP instruction.)

EICALL

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



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:
- The crystal frequency: The frequency of the crystal oscillator.
  - The duration of the clock period for the instruction cycle is a function of this crystal frequency.
  - To calculate the clock period (machine cycle) for the AVR, we take the inverse of the crystal frequency.

Instruction cycle time for the AVR 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

Delay calculation for the AVR

- The AVR design: Due to the limitations of IC technology and limited CPU design experience for many years, the instruction cycle duration was longer.
  - One way to increase performance of a given family is to reduce the number of instruction cycles it takes to execute an instruction.
  - 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 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.

Instruction cycle time for the AVR Example 3-15

Instruction cycle time for the AVR Example 3-16

Delay calculation for AVR

- A delay subroutine consists of two parts:
  - (1) setting a counter
  - (2) a loop
  - Most of the time delay is performed by the body of the loop.

Loop inside a loop delay Example 3-18

- Another way to get a large delay is to use a loop inside a loop, which is also called a *nested loop.*

Example 3-19

Example 3-20