Principios de Mecatrónica – SDI-11561
Ingeniería en Mecatrónica

Hugo Rodríguez Cortés

Departamento de Ingeniería Eléctrica y Electrónica

Instituto Tecnológico Autónomo de México

Agosto 2023

Registro de estatus

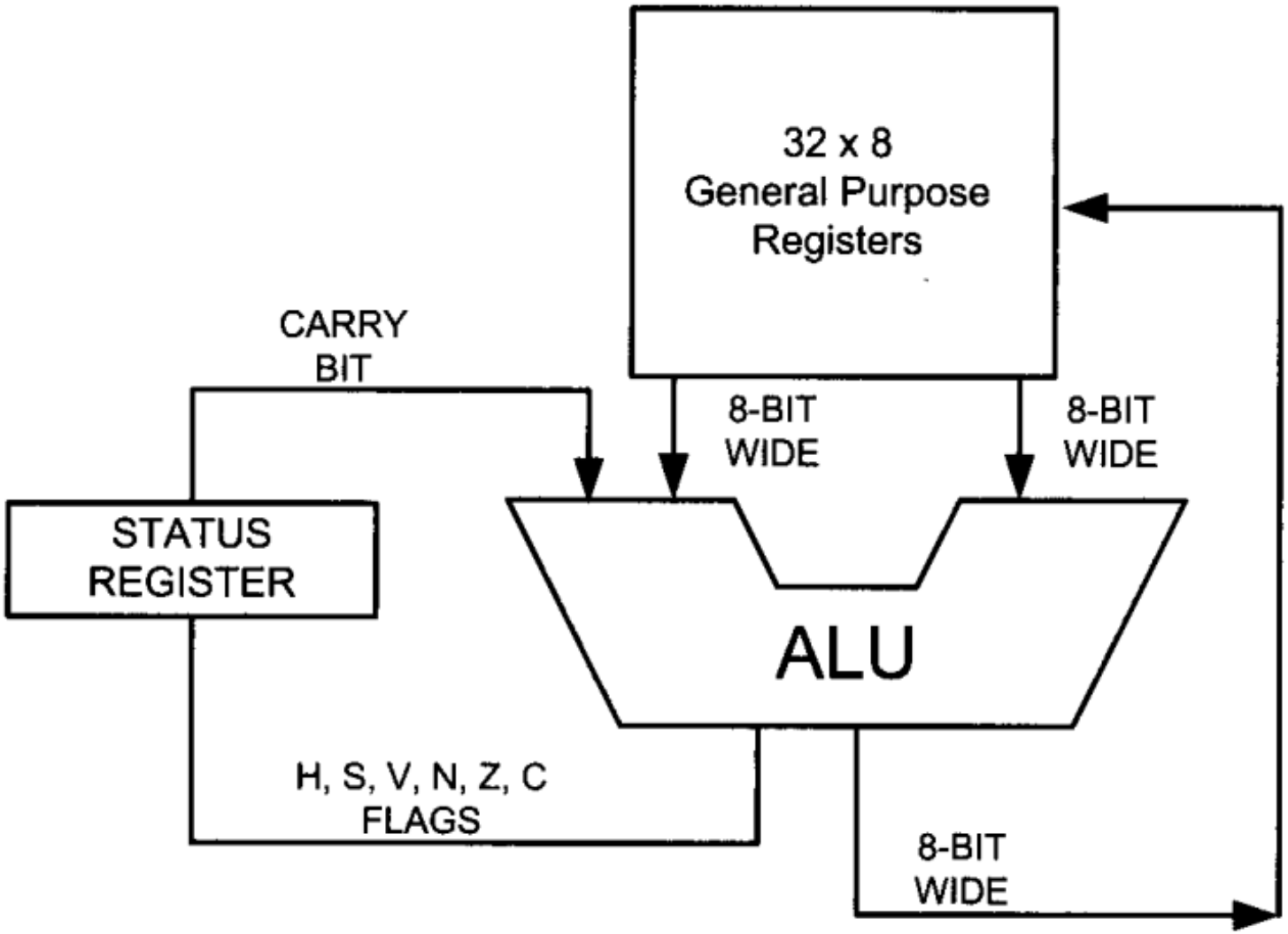
- Determine el valor de R21 después de las siguientes instrucciones

```
LDI   R21, 0 × F5   ;   –
LDI   R22, 0 × 0B   ;   –
ADD    R21, R22     ;   –
```

- La suma en binario es

$$\begin{array}{r} 11110101 \\ +\ 00001011 \\ \hline \end{array}$$

ALU


32 x 8
General Purpose
Registers
CARRY
BIT
8-BIT
WIDE
8-BIT
WIDE
STATUS
REGISTER
ALU
H, S, V, N, Z, C
FLAGS
8-BIT
WIDE

Registro de estatus

- El microcontrolador AVR tiene un registro para indicar la condición aritmética de una operación. Este registro se conoce como registro de estatus (SReg).

- El registro de estatus tiene la siguiente estructura

Bit	D7	D6	D5	D4	D3	D2	D1	D0
SREG	I	T	H	S	V	N	Z	C

  - C (carry flag). C=1, si lleva 1 en el bit D7.
  - Z (zero flag). Z=1 si el resultado de la operación es igual a cero.
  - N (negative flag). Se utiliza para hacer operaciones con signo, si D7 = 0, número positivo, N = 0, si D7 =1, N=1, número negativo.

Registro de estatus

Bit	D7	D6	D5	D4	D3	D2	D1	D0
SREG	I	T	H	S	V	N	Z	C

- V (overflow flag). V $= 1$ si el resultado de una operación de números con signo causa un desbordamiento hacia el bit que indica el signo.
- S (sign bit). Es el resultado de una OR-exclusiva (XOR) entre las banderas N y V.
- H (half carry flag) H=1 si durante ADD o SUB se lleva 1 de D3 a D4.
- Las banderas I y T se analizarán después.

Registro de estatus (ADD)

- Determine el estatus de la bandera Z en cada línea de la siguiente secuencia

```
LDI   R20, 4   ;   –
DEC    R20     ;   –
DEC    R20     ;   –
DEC    R20     ;   –
DEC    R20     ;   –
```

En operaciones sin signo, la operación ADD puede modificar las banderas C, H y Z.

- Determine el valor de las banderas C, H y Z en las siguientes operaciones

```
LDI   R16, 0 × 38   ;   –
LDI   R17, 0 × 2F   ;   –
ADD    R16, R17     ;   –
```

Registro de estatus(ADD)

- Determine el valor de las banderas C, H y Z en las siguientes operaciones

```
LDI   R20, 0×9C   ;   –
LDI   R21, 0×64   ;   –
ADD   R20, R21    ;   –
```

```
LDI   R20, 0×88   ;   –
LDI   R21, 0×93   ;   –
ADD   R20, R21    ;   –
```

Registro de Estatus y Condicionales

Se cuenta con una serie de instrucciones de que realizan un salto condicional (branch) en función del valor de algunos bits del registro de estatus.

Instrucción	Resultado
BRLO	Branch si C=1
BRSH	Branch si C=0
BREQ	Branch si Z=1
BRNE	Branch si Z=0
BRMI	Branch si N=1
BRPL	Branch si N=0
BRVS	Branch si V=1
BRVC	Branch si V=0

Formato de datos

Se tienen cuatro formas para representar los bytes de datos en el lenguaje ensamblador para AVR.

- Números hexadecimales. Se utiliza $0\times$ ó $\$$ frente al número. $0 \times 99 = \$99$.

- Números binarios. Se utiliza 0b ó 0B frente al número. $0\text{b}00100101 = 0\text{B}00100101$.

- Números decimales. Se utiliza el número decimal sin índicadores. LDI $R17, 12$.
- Caracteres ASCII. Se utiliza un apóstrofe simple. $'2' = 0\text{b}00110010 = 0 \times 32 = \$32$.

Directivas en ensamblador

Las directivas dan instrucciones al ensamblador, no al CPU. Se conocen como pseudo-instrucciones. Las directivas ayudan a desarrollar el programa y hacerlo legible. No generan código de máquina.

- .EQU Esta directiva permite fijar valores o direcciones constantes

```
.EQU COUNT = 0 × 25      ; COUNT toma el valor de 0 × 25.
  ⋮                      ⋮
LDI   R21, COUNT         ; R21 = COUNT = 0 × 25
```

- .SET Se utiliza para definir un valor constante o una dirección fija. El valor asignado por .SET es modificable.
- .ORG Se utiliza para indicar el inicio de la dirección, para código y datos.
- .INCLUDE Agrega el contenido de un archivo al programa.

Directivas en ensamblador

- Ejemplo

```
.INCLUDE "m256def.inc"      ; Incluye el archivo m256def.inc
.ORG   0 × 00               ; Almacenar código de máquina
                            a partir de la dirección 0 × 00
```

- .DEVICE Define el microcontrolador que va a utilizarse.

```
.DEVICE ATMega2560      ; ATMega2560 designación del chip.
```

- Ejemplo

```
.EQU SUM = 0x120      ; Define SUM = 0x120.
LDI   R20, 5          ; cargar el 5 a R20
LDI   R21, 2          ; cargar el 2 a R21
ADD    R20, R21       ; R20 = R20 + R21
STS   SUM, R20        ; almacenar R20 en la dirección 0x120
```

Lazos en el AVR

El AVR tiene instrucciones para realizar saltos condicionales y no condicionales.

- Se llama lazo a la repetición de una secuencia de instrucciones ó de una instrucción un cierto número de veces, por ejemplo

```
LDI   R16, 0      ;  –
LDI   R17, 3      ;  –
ADD   R16, R17    ;  –
ADD   R16, R17    ;  –
ADD   R16, R17    ;  –
ADD   R16, R17    ;  –
ADD   R16, R17    ;  –
ADD   R16, R17    ;  –
```

Lazos en el AVR

- BRNE (branch if not equal) utiliza la bandera Z del registro de estatus.

```
BACK:   ...          ; inicio del lazo.
        ...          ; instrucción del lazo.
        ...          ; instrucción del lazo.
        DEC   Rn     ; decrementar Rn, Z=1 si Rn = 0.
        BRNE  BACK   ; regresar a BACK si Z=0.
```

- Ejemplo. Realizar un programa para multiplicar 3 por 10 y escribir el resultado en PORT B.

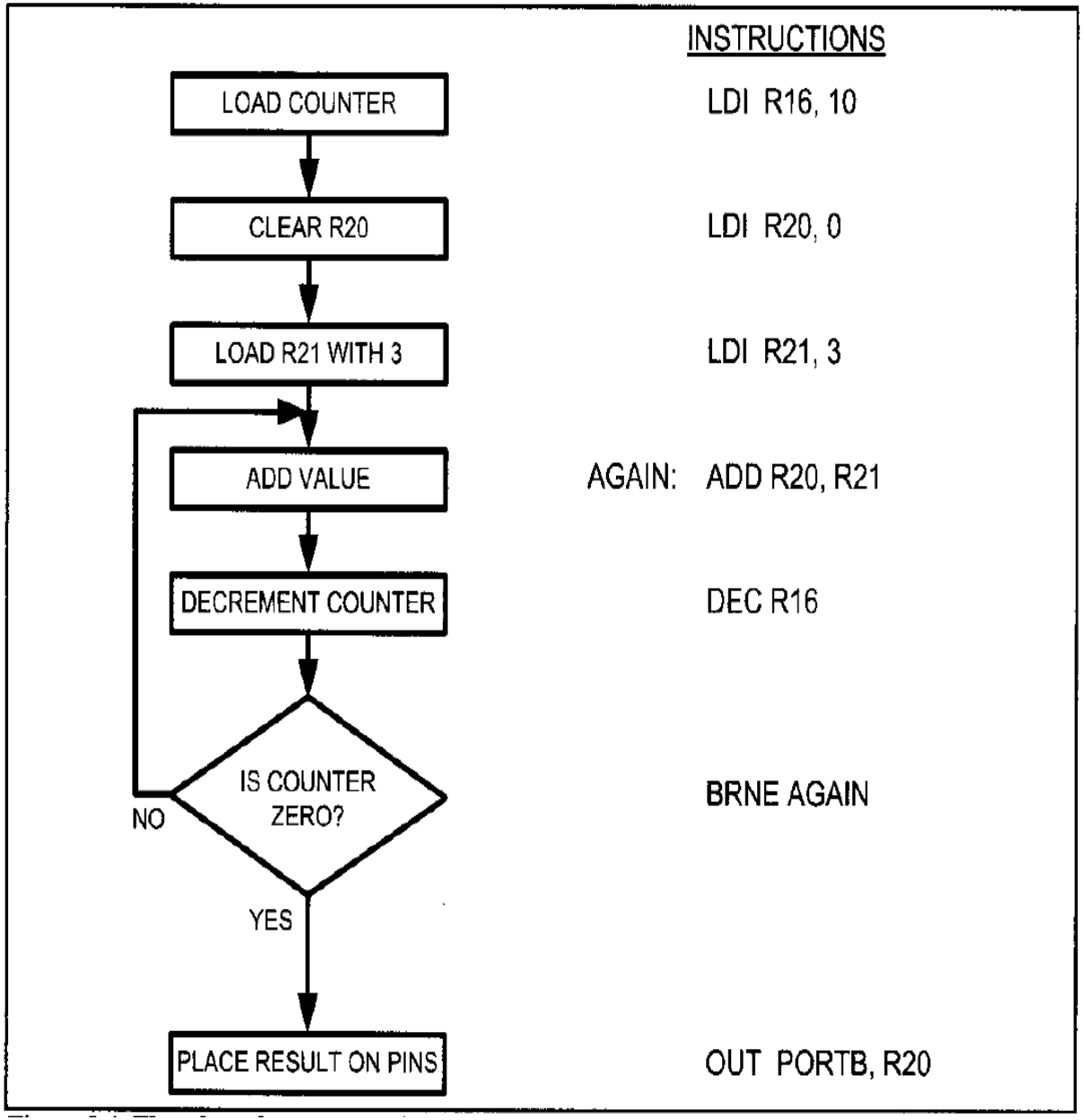
Lazos en el AVR

- Solución

```
.INCLUDE  "M32DEF.INC"      ;  –
          LDI   R16, 10     ;  –
          LDI   R20, 0      ;  –
          LDI   R21, 3      ;  –
AGAIN:    ADD   R20, R21    ;  –
          DEC   R16         ;  –
          BRNE  AGAIN       ;  –
          OUT   PORTB, R20  ;  –
```

- ¿Cuál es el máximo número de veces que puede repetirse un conjunto de instrucciones?

Lazos en el AVR

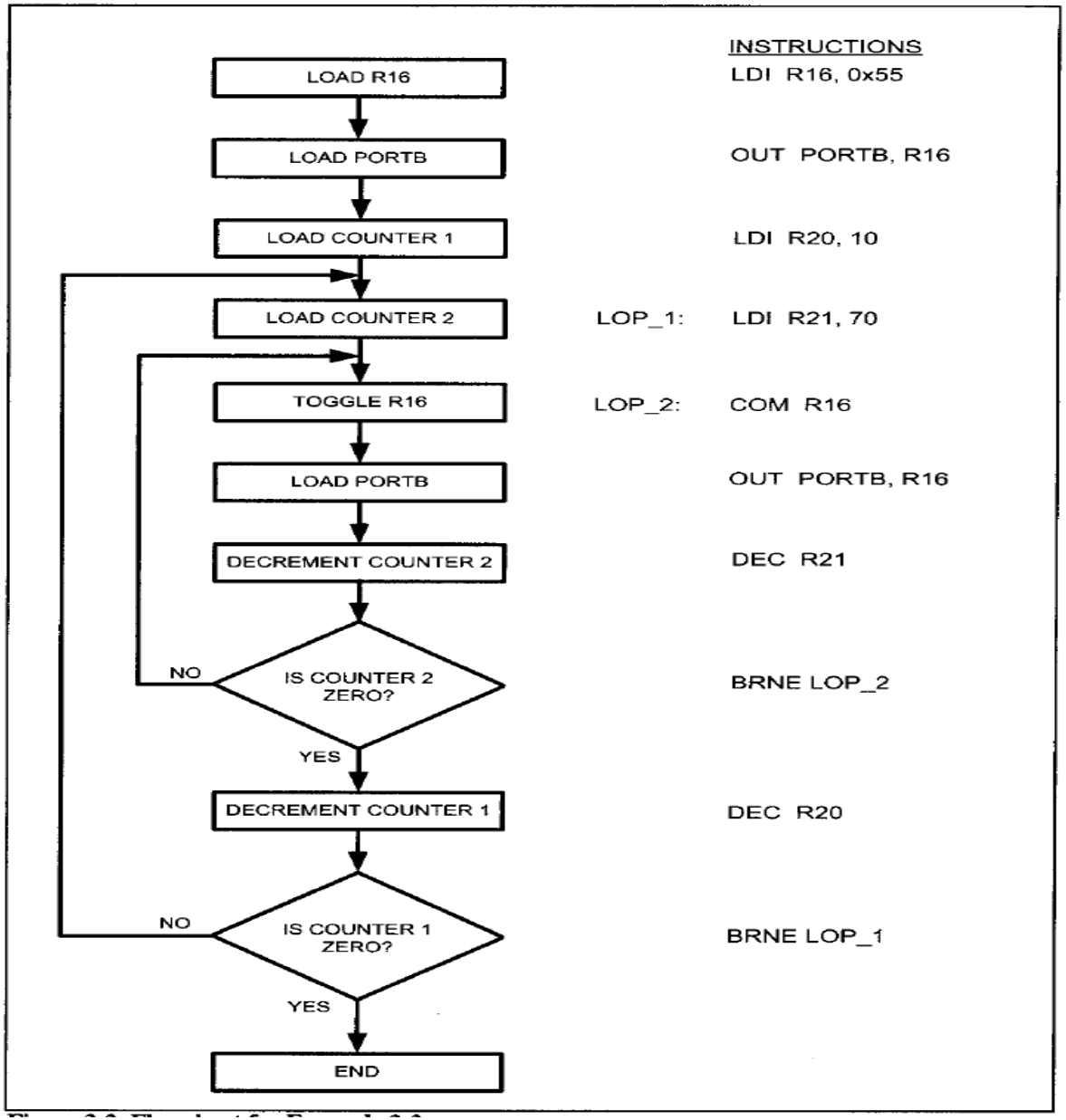


Lazo dentro de otro lazo

- Ejemplo. El siguiente programa carga al registro PORTB el valor de 0x55 y su complemento 700 veces.

```
.INCLUDE  "M32DEF.INC"   ;                              –
.ORG      0x00           ;                              –
          LDI    R16, 0 × 55   ;                               –
          OUT    PORTB, R16    ;                               –
          LDI    R20, 10       ;                               –
LOP_1:    LDI    R21, 70       ;                               –
LOP_2:    COM    R16           ;                               –
          OUT    PORTB, R16    ;                               –
          DEC    R21           ;                               –
          BRNE   LOP_2         ;                               –
          DEC    R20           ;                               –
          BRNE   LOP_1         ;                               –
```

Lazo dentro de un lazo



Otros saltos condicionales

- BREQ (brach if equal, branch if Z = 1) Ejemplo

```
OVER:  IN     R20, PINB  ;                              –
       TST    R20        ;  TST verifica si R20 = 0 ⇒ Z=1
       BREQ   OVER       ;                              –
```

El programa sale del lazo cuando PINB toma un valor diferente de cero. TST también asigna N = 1 si D7 = 1.

- BRSH (branch if same or higher, branch if C = 0) Al ejecutar "BRSH label" el CPU verifica el registro de estatus. Si C = 0 el CPU comienza a recolectar y ejecutar las instrucciones del label. Si C = 1 el programa no salta continuando con el programa.

Otros saltos condicionales

- Ejemplo. Sumar 0x79 + 0xF5 + 0xE2 colocando el resultado en dos bytes.

```
.INCLUDE   "M32DEF.INC"   ;                    –
.ORG       0x00           ;                    –
           LDI    R21, 0          ;                 –
           LDI    R20, 0          ;                 –
           LDI    R16, 0x79       ;                 –
           ADD    R20, R16        ;                 –
           BRSH   N_1             ;                 –
           INC    R21             ;                 –
N_1:       LDI    R16, 0xF5       ;                 –
           ADD    R20, R16        ;                 –
           BRSH   N_2             ;                 –
           INC    R21             ;                 –
N_2:       LDI    R16, 0xE2       ;                 –
           ADD    R20, R16        ;                 –
           BRSH   OVER            ;                 –
           INC    R21             ;                 –
OVER:
```

Saltos no condicionales

- JMP "label" Salta a cualquier dirección dentro de la memoria del programa. No está disponible en todos los dispositivos debido a que algunos tienen una capacidad de memoria limitada. Ejemplo

```
        MOV   R21, R20   ;                    –
        JMP   OVER       ;                    –
        DEC   R21        ;                    –
OVER:
        RJMP  OVER       ;                    –
```

- RJMP "label" Salto no condicional disponible en dispositivos con baja capacidad de memoria. Las direcciones relativas tienen un rango entre 0×0 y $0\times$FFF. Se prefiere sobre JMP ya que ocupa menos espacio de memoria.

Ubicación del código en ROM

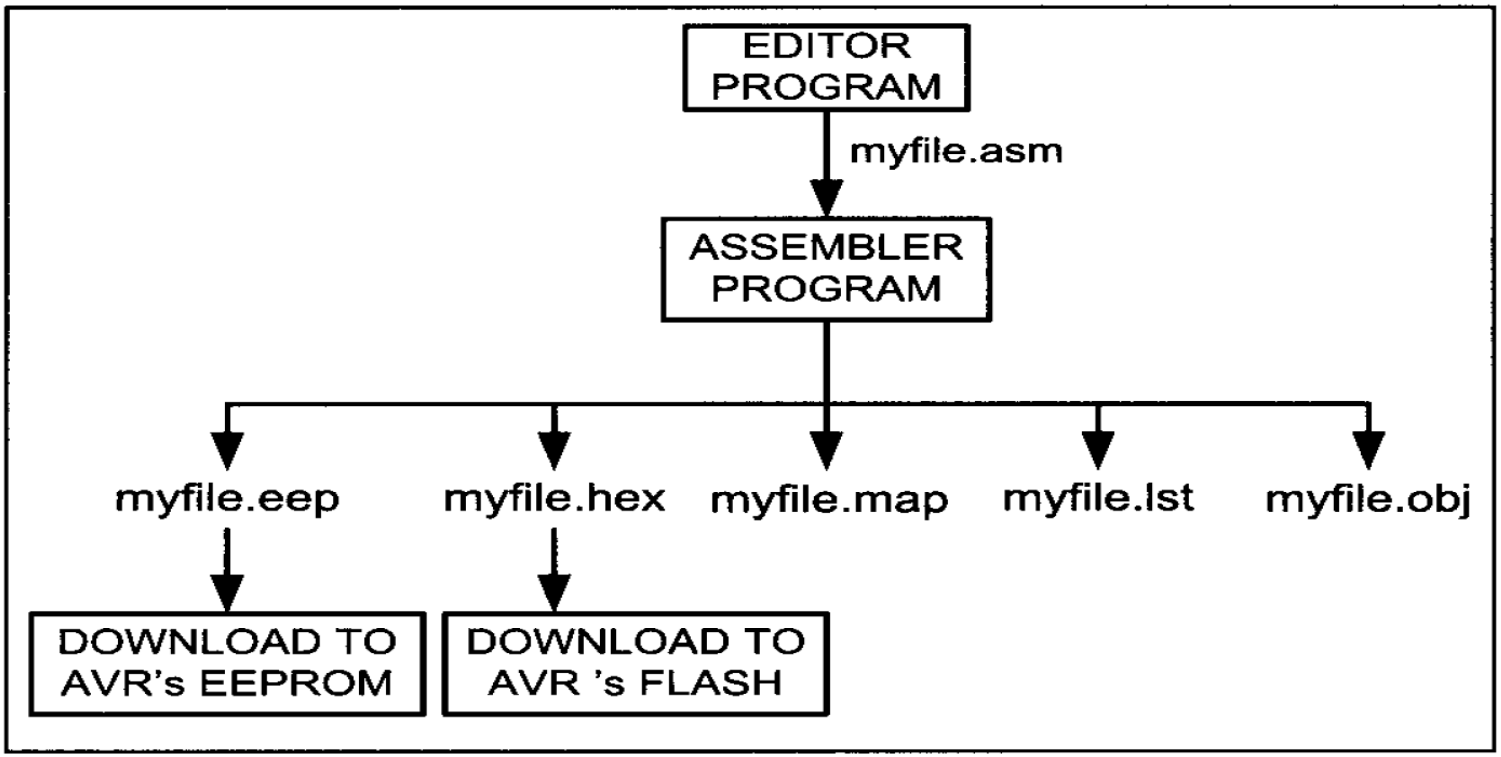
- Cuando el AVR se prende o se resetea, el CPU inicia en la localidad de memoria 0000. Por lo que el contador de programa[a] tiene el valor 0000. Esto indica que el primer código de operación esta almacenado en la dirección de memoria 0000.
- Considere el siguiente programa

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

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

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

[a] Registro que indica al CPU la dirección de la instrucción que debe ejecutarse.

Ubicación del código en ROM

El editor del programa realiza la compilación y genera los archivos que deben cargarse a la memoria ROM del microcontrolador.



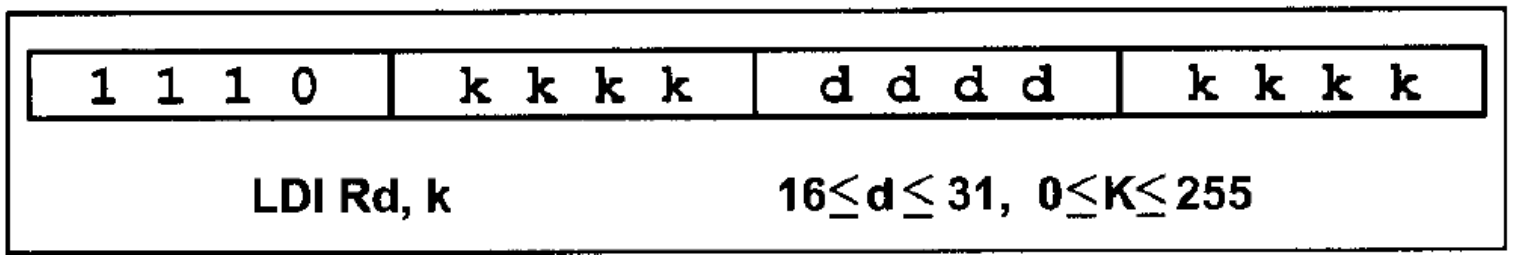
Ubicación del código en ROM

Una parte del código fuente tiene la siguiente estructura

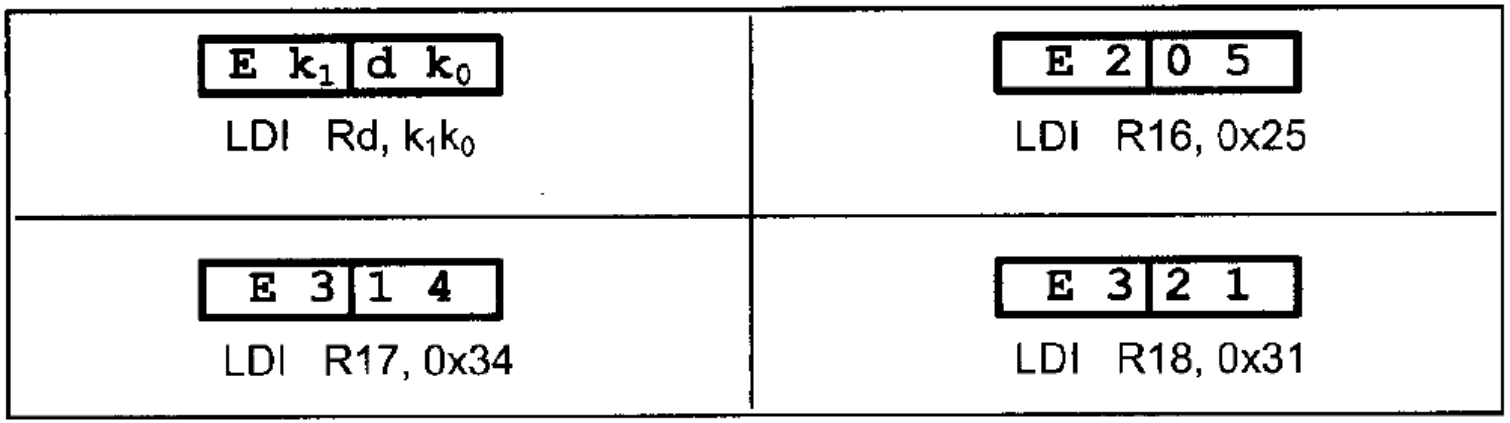
```
AVRASM ver. 2.1.2  F:\AVR\Sample\Sample.asm Tue Mar 11 11:28:34 2008

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

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

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

Assembly complete, 0 errors, 0 warnings
```

Ubicación del código en ROM

- La lista muestra que en la dirección 0000 se almacena el código operacional E205. Esto es, la instrucción LDI R16, 0x25 tiene el código operacional E205. El código operacional para LDI Rd, k tiene tiene la estructura siguiente

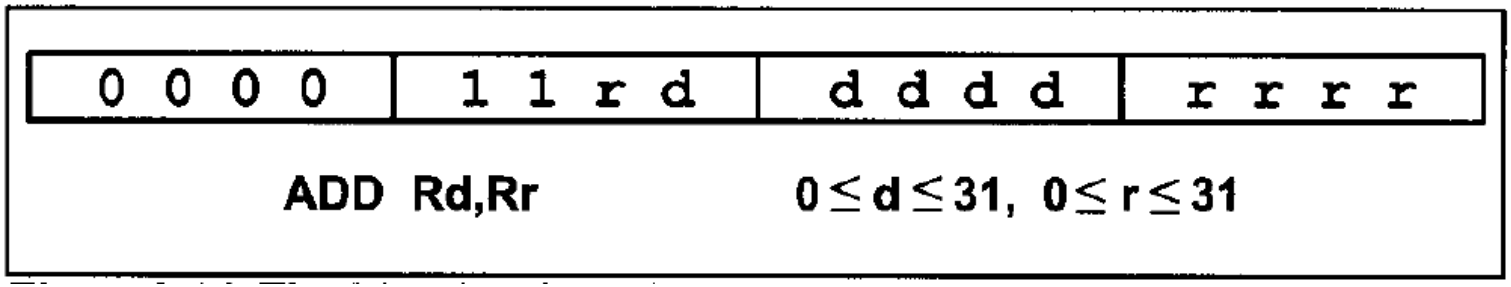


- En la lista se tienen los siguientes códigos operacionales

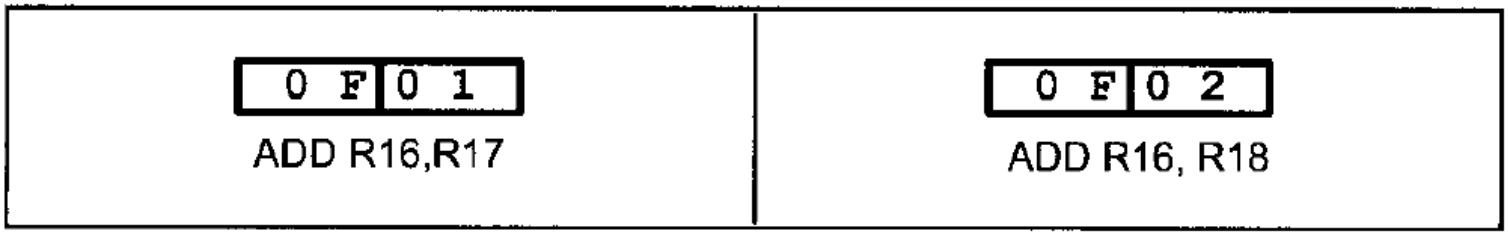


Ubicación del código en ROM

- Se puede observar que la dirección de memoria 0003 tiene el código operacional para 0F02 que corresponde a ADD R16, R17. La estructura del código operacional de ADD Rd, Rk es



- En la lista aparecen los siguientes códigos operacionales.



Ubicación del código en ROM

- La instrucción STS es de 4 bytes. Los primeros 16 bits tienen el código operacional y la dirección de la fuente, los otros 16 bits indican la dirección del destino.

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

1001	001r	rrrr	0000
kkkk	kkkk	kkkk	kkkk

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

- La instrucción LDS es de 4 bytes. Los primeros 16 bits contienen el código operacional y el registro destino. Los otros 16 bits indican la dirección de memoria dónde está la información.

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

1001	000d	dddd	0000
kkkk	kkkk	kkkk	kkkk

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

Ubicación del código en ROM

- La instrucción JMP es de 4 bytes. 10 bits son para código operacional. Los otros 22 bits indican la dirección objetivo de la instrucción.

JMP k ;Jump to address k

1001	010k	kkkk	110k
kkkk	kkkk	kkkk	kkkk

$0 \leq k \leq 4M$

- RJMP label Es una instrucción de 2 bytes.

1100	kkkk	kkkk	kkkk

Instrucciones CALL y STACK

- CALL label Es una instrucción de 4 bytes, 10 bits se utilizan para el código operacional, los otros 22 bits contienen información de la dirección de la subrutina.

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

$0 \leq k \leq 3FFFFF$

- Para asegurar que el CPU sabe a que instrucción regresar después de ejecutar la subrutina label, la dirección de la instrucción abajo de CALL se guarda en el STACK.

Instrucciones CALL y STACK

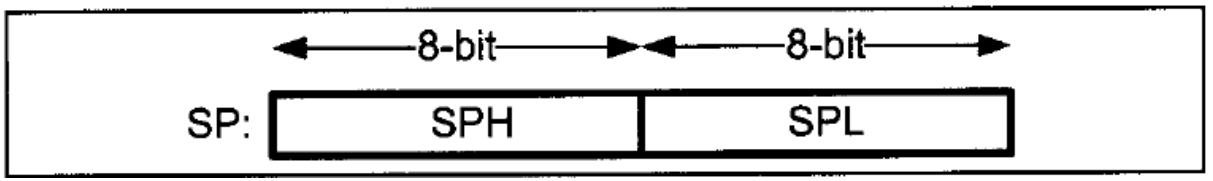
- Toda subrutina debe tener a la instrucción RET como última instrucción.

```
        MOV   R21, R20   ;                    –
        CALL   OVER      ;                    –
        STS   0xFF, R21  ;                    –
OVER:                    ;                    –
        DEC   R21        ;                    –
        RET              ;                    –
```

- Si el STACK se encuentra en la RAM debe haber un registro SP, dentro del CPU, para apuntar a el. SP se conoce como registro apuntador al STACK.

Instrucciones CALL y STACK

- En el espacio de memoria I/O el SP se implementa con dos registros. SPL (el byte bajo del SP) y SPH (el byte alto del SP)



- El SP debe tener la capacidad para apuntar a cualquier dirección de la RAM. En AVRs con más de 256 bytes de memoria se utilizan SPL y SPH, si la memoria es menor solamente se utiliza SPL.

- Para almacenar información en el STACK se utiliza la instrucción PUSH y para cargar información almacenada en el STACK se utiliza POP.

Instrucciones PUSH y POP

- Cuando se envían datos al STACK, los datos se guardan en el registro al que apunta el SP, y SP se decrementa en una unidad.

```
PUSH   Rr   ;   Rr es cualquier GPR–
```

- Para extraer información del STACK se utiliza la instrucción POP. Cuando se ejecuta POP el SP se incrementa en una unidad y la primera dirección del STACK con información se copia al registro indicado por POP. El STACK es una memoria LIFO (Last-In-First-Out).

```
POP   Rr   ;   Rr es cualquier GPR–
```

Instrucciones PUSH y POP

- Ejemplo

```
LDI     R16, HIGH(RAMEND)   ;   RAMEND dirección de la última localidad de RAM
OUT     SPH, R16            ;
LDI     R16, LOW(RAMEND)    ;
OUT     SPL, R16            ;

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

PUSH    R20                 ;
PUSH    R22                 ;

LDI     R20, 0              ;
LDI     R22, 0              ;

POP     R20                 ;
POP     R22                 ;
```

Instrucción RCALL

- RCALL label Es una instrucción de 2 bytes. Solamente 12 bits de los dos bytes se utilizan para la dirección.
- RCALL y CALL funcionan exáctamente igual, la diferencia es que la dirección objetivo para CALL puede estar en cualquier lugar de la memoria y para RCALL debe estar dentro de un rango de 4K.
- En varios modelos comercializados por AVR la ROM en el chip es de 4K por lo que se prefiere utilizar RCALL.

STACK y RET

- Cuando el AVR inicia el SP contiene el valor 0x00 que corresponde al GPR R0. El SP debe inicializarse para apuntar a alguna localidad de la SRAM.

- RAMEND es la dirección de la última localidad de memoria de la RAM. De tal forma que SPH apunta al byte alto de RAMEND, mientras que SPL apunta al byte bajo.

- Cuando aparece la instrucción CALL/RCALL el CPU guarda la dirección de memoria de la instrucción debajo de CALL/RCALL en el STACK.

- En el AVR ATMega32 el valor del contador de programa se almacena en dos bytes, el byte alto se manda primero al STACK.

STACK y RET

- Cuando se ejecuta RET, la última dirección que entró al STACK se copia al contador de programa.
- El STACK no puede apuntarse a la memoria de registro ni a la memoria I/O.
- Recordar que cuando se llama a una subrutina, el STACK sigue la pista de dónde debe regresar el CPU al completar la subrutina.
- El conocimiento detallado de las instrucciones permite utilizarlas de forma eficiente.

STACK y RET

- Ejemplo A

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

BACK:                                   ;                                       —
        LDI     R16, 0x55               ;                                       —
        OUT     PORTB, R16              ;                                       —
        CALL    DELAY                   ;                                       —
        LDI     R16, 0xAA               ;                                       —
        OUT     PORTB, R16              ;                                       —
        CALL    DELAY                   ;                                       —
        RJMP    BACK                    ;                                       —
.ORG    0x300                           ;                                       —
DELAY:                                  ;                                       —
        LDI     R20, 0xFF               ;                                       —
AGAIN:                                  ;                                       —
        NOP                             ;   sin operación, gasta ciclos de reloj—
        NOP                             ;                                       —
        DEC     R20                     ;                                       —
        BRNE    AGAIN                   ;                                       —
        RET                             ;                                       —
```

CALL

Analice el contenido del STACK en el siguiente programa

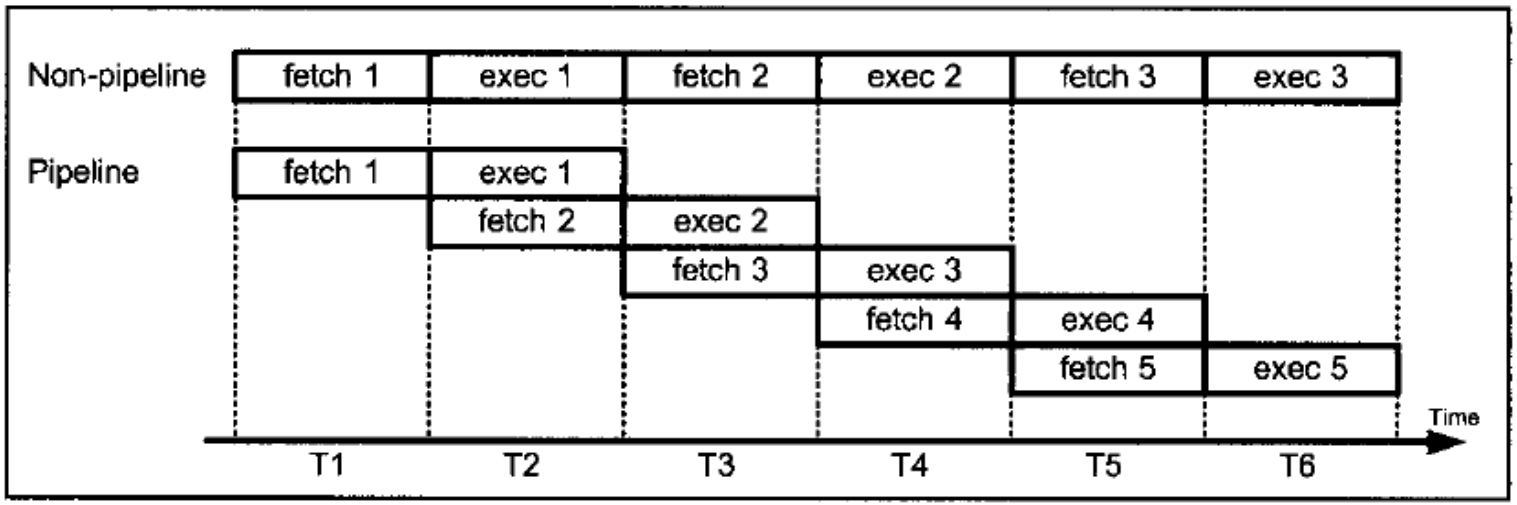
```
                .INCLUDE "M32DEF.INC"
                .ORG 0
+00000000:          LDI R16,HIGH(RAMEND)    ;initialize SP
+00000001:          OUT SPH,R16
+00000002:          LDI R16,LOW(RAMEND)
+00000003:          OUT SPL,R16

                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
```

Valor del retardo en el AVR

- Al crear un retardo utilizando el lenguaje ensamblador deben considerarse los siguientes factores
  - La frecuencia del cristal. EL cristal oscilador se conecta a los puertos de entrada XTAL1 y XTAL2. El periodo del reloj para un ciclo de instrucción es función de la frecuencia del cristal.
  - El diseño del AVR. Para ejecutar una instrucciónen un cyclo se utiliza. (a) Arquitectura Harvard, para alimentar la cantidad máxima de código y datos al CPU. (b) Utilizar instrucciones de tamaño fijo, característica de la arquitectura RISC. (c) canalización para superponer instrucciones de extraer y ejecutar.

Valor del retardo en el AVR

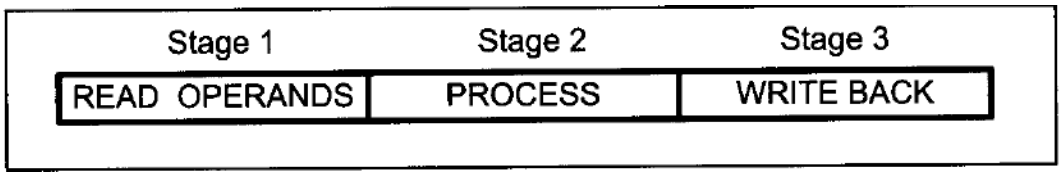
- Una idea simple de pipelining es darle la capacidad al CPU de ir a buscar una instrucción en la memoria y después ejecutarla y repetir la secuencia.



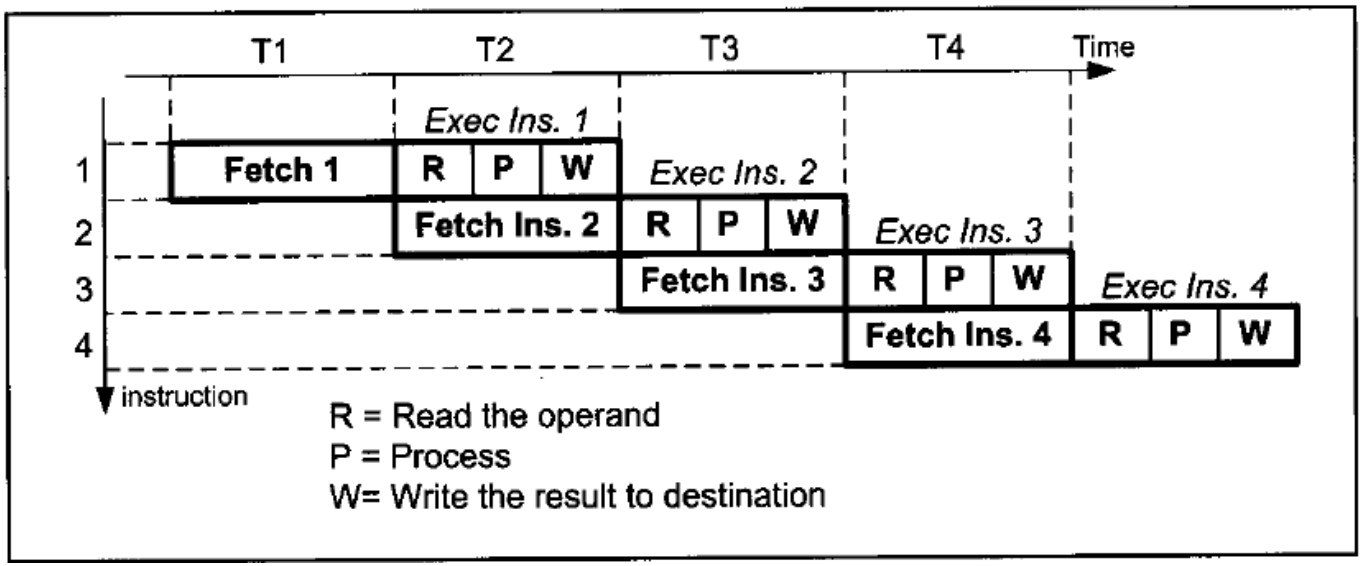
Se recolecta una instrucción mientras la instrucción previa se ejecuta.

Valor del retardo en el AVR

- En pipelining el proceso de ejecutar instrucciones de divide en tres pasos pequeños de tal forma que la ejecución de varias instrucciones puede traslaparse. Buscar operando, ejecutar operando en la ALU, entregar el resultado.



- La siguiente figura muestra la canalización para tres instrucciones.



Valor del retardo en el AVR

- El tiempo que le toma al CPU ejecutar una instrucción se conoce como ciclo de máquina.
- Como las instrucciones en el AVR son de 1 o 2 bytes, la mayoría de las instrucciones se ejecuta en uno o dos ciclos de máquina.
- JMP y CALL pueden tomar entre tres y cuatro ciclos de máquina.
- En la familia AVR el valor del ciclo de máquina depende de la frecuencia del oscilador que se conecte.
- En el AVR un ciclo de máquina es igual al periodo del oscilador.

Valor del retardo en el AVR

- Ejemplo. A un sistema basado en AVR pueden conectarse cristales con las frecuencias siguientes

$$a)8\text{MHz}, \quad b)16\text{MHz}, \quad c)10\text{MHz}, \quad d)1\text{MHz}$$

¿Cuál sería el ciclo de máquina en cada caso?

Penalización por salto

- La estructra pipelining implica que hay un buffer dónde se guarda la instrucción siguiente. ¿Qué pasa en un salto?
- En un salto, el CPU elimina la información de este buffer y recolecta la siguiente instrucción de una nueva localidad de memoria. En este caso la unidad que ejecuta el programa debe esperar a que se encuentre disponible la nueva instrucción.
- Esto significa que algunas instrucciones van a requerir dos, tres o cuatro ciclos de máquina para ejcutarse.
- Ejemplo: JMP, CALL, RET y saltos condicionales como BRNE, BRLO, etc. Los saltos condicionales toman un ciclo de máquina si no brincan.

DELAY

- Encontrar la duración del DELAY en el siguiente programa si el cristal es de 10 MHz.

```
.DEF          COUNT = R20    ;   ciclos de máquina : 0−
DELAY  LDI    COUNT, 0xFF    ;   ciclos de máquina : 1−
AGAIN  NOP                   ;   ciclos de máquina : 1−
       NOP                   ;   ciclos de máquina : 1−
       DEC    COUNT          ;   ciclos de máquina : 1−
       BRNE   AGAIN          ;   ciclos de máquina : 2/1−

       RET                   ;   ciclos de máquina : 4−
```

DELAY

- Encontrar la duración del DELAY en el siguiente programa si el cristal es de 10 MHz.

```
.DEF          COUNT = R20    ;   ciclos de máquina : 0−
DELAY  LDI    COUNT, 0xFF    ;   ciclos de máquina : 1−
AGAIN  NOP                   ;   ciclos de máquina : 1−
       NOP                   ;   ciclos de máquina : 1−
       DEC    COUNT          ;   ciclos de máquina : 1−
       BRNE   AGAIN          ;   ciclos de máquina : 2/1−

       RET                   ;   ciclos de máquina : 4−
```

Cuando BRNE salta se tiene

$$DELAY = \{1\text{LDI} + [2\text{NOP} + 1\text{DEC} + 2\text{BRNE}] \times 255 + 4\} \times 0.1\mu\text{s} = 128\mu\text{s}$$

Ajuste cuando BRNE no salta, $DELAY = 127.9\mu$s.

DELAY

- Si el ciclo de máquina es de 1μs, determinar el tamaño del DELAY y la cantidad de ROM que consume el siguiente programa

```
DELAY:  LDI   R16, 200  ;  ciclos de máquina : 1−
AGAIN:  LDI   R17, 250  ;  ciclos de máquina : 1−
HERE:   NOP             ;  ciclos de máquina : 1−
        NOP             ;  ciclos de máquina : 1−
        DEC   R17       ;  ciclos de máquina : 1−
        BRNE  HERE      ;  ciclos de máquina : 2/1−
        DEC   R16       ;  ciclos de máquina : 1−
        BRNE  AGAIN     ;  ciclos de máquina : 2/1−
        RET             ;  ciclos de máquina : 4−
```

DELAY

- Si el ciclo de máquina es de 1μs, determinar el tamaño del DELAY y la cantidad de ROM que consume el siguiente programa

```
DELAY:  LDI    R16, 200  ;  ciclos de máquina : 1−
AGAIN:  LDI    R17, 250  ;  ciclos de máquina : 1−
HERE:   NOP              ;  ciclos de máquina : 1−
        NOP              ;  ciclos de máquina : 1−
        DEC    R17       ;  ciclos de máquina : 1−
        BRNE   HERE      ;  ciclos de máquina : 2/1−
        DEC    R16       ;  ciclos de máquina : 1−
        BRNE   AGAIN     ;  ciclos de máquina : 2/1−
        RET              ;  ciclos de máquina : 4−
```

Lazo $HERE = \{[2\text{NOP} + 1\text{DEC} + 2\text{BRNE}] \times 250 - 1\} \times 0.1\mu\text{s} = 1249\mu\text{s}$
Lazo $AGAIN = \{[1\text{LDI} + 1\text{DEC} + 2\text{BRNE}] \times 200 - 1\} \times 0.1\mu\text{s} + 200 \times HERE = 799\mu\text{s} + 249800\mu\text{s} = 250599\mu\text{s}$
$DELAY = AGAIN + [1LDI + 4RET] \times 0.1\mu\text{s} = 250604\mu\text{s}.$

DELAY

- Se tienen nueve instrucciones de 2 bytes por lo que el programa consume 18 bytes de memoria.
- Considere

```
DELAY:  LDI   R16, 200  ;  ciclos de máquina : 1−
AGAIN:  NOP             ;  ciclos de máquina : 1−
        NOP             ;  ciclos de máquina : 1−
        NOP             ;  ciclos de máquina : 1−
        NOP             ;  ciclos de máquina : 1−
        NOP             ;  ciclos de máquina : 1−
        NOP             ;  ciclos de máquina : 1−
        NOP             ;  ciclos de máquina : 1−
        NOP             ;  ciclos de máquina : 1−
        NOP             ;  ciclos de máquina : 1−
        NOP             ;  ciclos de máquina : 1−
        NOP             ;  ciclos de máquina : 1−
        NOP             ;  ciclos de máquina : 1−
        NOP             ;  ciclos de máquina : 1−
        NOP             ;  ciclos de máquina : 1−
        DEC   R16       ;  ciclos de máquina : 1−
        BRNE  AGAIN     ;  ciclos de máquina : 2/1−
        RET             ;  ciclos de máquina : 4−
```

DELAY

- Se tiene

$$\begin{aligned} DELAY &= \{1LDI + 200\,[12NOP + 1DEC + 2BRNE] - 1 \\ &\quad +4RET\} \times 1\mu\text{s} \\ &= 3004\mu\text{s} \end{aligned}$$

  Son 16 instrucciones de 2 bytes, por lo tanto el tamaño en memoria es de 32 bytes.

- Para obtener un DELAY más preciso se utilizan los timers del AVR.

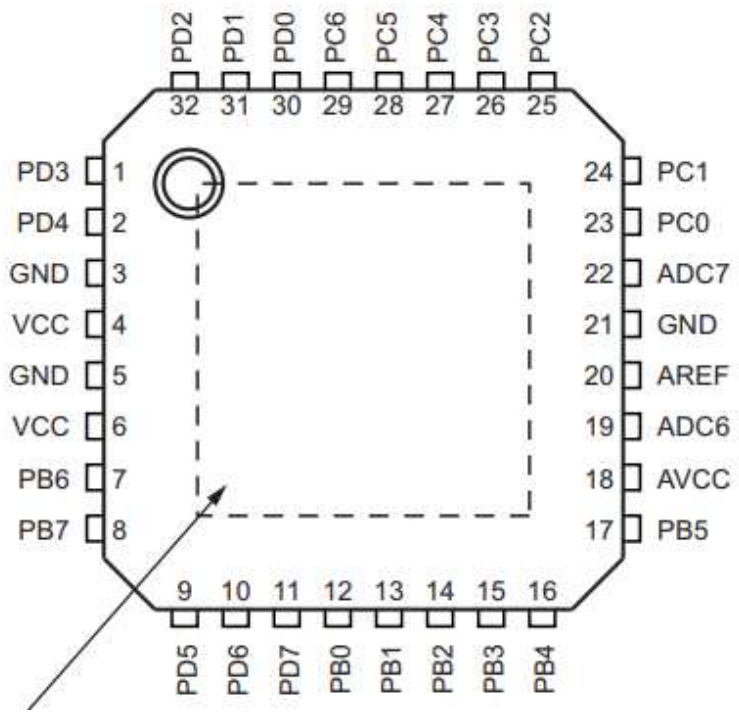
DELAY

- Considere que el ciclo de máquina es de nμs, calcule el DELAY generado por el siguiente código. Verifique el efecto del DELAY en un código para Arduino UNO.

```
DELAY:
        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                 ;   –
```

Programación de puertos I/O

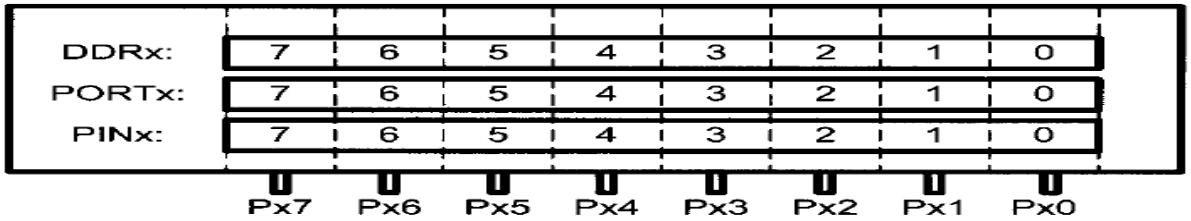
- Dependiendo del chip que se seleccione se tienen diferentes puertos I/O.



- El ATMega328P tiene 32 pines, de los cuales 9 están designados como 3GND, 2VCC, ADC7, AREF, ADC6, AVCC, los restantes se asignan a los puertos B (8 pines), C (7 pines) y D (8 pines).

Programación de puertos I/O

- Cada puerto tiene asignados tres registros PORTx, DDRx y PINx. DDR Data Direction Register, PIN PortINput Pins



- Además de ser puertos de entrada cada pin tiene una función alterna como ADC, timer, interrupciones, comunicación serial, etc. Puerto C

Port Pin	Alternate Function
PC6	$\overline{\text{RESET}}$ (reset pin) PCINT14 (pin change interrupt 14)
PC5	ADC5 (ADC input channel 5) SCL (2-wire serial bus clock line) PCINT13 (pin change interrupt 13)
PC4	ADC4 (ADC input channel 4) SDA (2-wire serial bus data input/output line) PCINT12 (pin change interrupt 12)
PC3	ADC3 (ADC input channel 3) PCINT11 (pin change interrupt 11)
PC2	ADC2 (ADC input channel 2) PCINT10 (pin change interrupt 10)
PC1	ADC1 (ADC input channel 1) PCINT9 (pin change interrupt 9)
PC0	ADC0 (ADC input channel 0) PCINT8 (pin change interrupt 8)

Programación de puertos I/O

- Puertos B y D

Port Pin	Alternate Functions
PB7	XTAL2 (chip clock oscillator pin 2) TOSC2 (timer oscillator pin 2) PCINT7 (pin change interrupt 7)
PB6	XTAL1 (chip clock oscillator pin 1 or external clock input) TOSC1 (timer oscillator pin 1) PCINT6 (pin change interrupt 6)
PB5	SCK (SPI bus master clock input) PCINT5 (pin change interrupt 5)
PB4	MISO (SPI bus master input/slave output) PCINT4 (pin change interrupt 4)
PB3	MOSI (SPI bus master output/slave input) OC2A (Timer/Counter2 output compare match A output) PCINT3 (pin change interrupt 3)
PB2	$\overline{SS}$ (SPI bus master slave select) OC1B (Timer/Counter1 output compare match B output) PCINT2 (pin change interrupt 2)
PB1	OC1A (Timer/Counter1 output compare match A output) PCINT1 (pin change interrupt 1)
PB0	ICP1 (Timer/Counter1 input capture input) CLKO (divided system clock output) PCINT0 (pin change interrupt 0)

Port Pin	Alternate Function
PD7	AIN1 (analog comparator negative input) PCINT23 (Pin Change Interrupt 23)
PD6	AIN0 (analog comparator positive input) OC0A (Timer/Counter0 output compare match A output) PCINT22 (pin change interrupt 22)
PD5	T1 (Timer/Counter 1 external counter input) OC0B (Timer/Counter0 output compare match B output) PCINT21 (pin change interrupt 21)
PD4	XCK (USART external clock input/output) T0 (Timer/Counter 0 external counter input) PCINT20 (pin change interrupt 20)
PD3	INT1 (external interrupt 1 input) OC2B (Timer/Counter2 output compare match B output) PCINT19 (pin change interrupt 19)
PD2	INT0 (external interrupt 0 input) PCINT18 (pin change interrupt 18)
PD1	TXD (USART output pin) PCINT17 (pin change interrupt 17)
PD0	RXD (USART input pin) PCINT16 (pin change interrupt 16)

Programación de puertos I/O

- Registro DDRx se utiliza para asignar el uso de un puerto como entrada o salida. Para que un puerto sea de salida se escriben 1's en este registro.

```
      LDI   R16, 0xFF     ;  –
      OUT   DDRB, R16     ;  –
L1:   LDI   R16, 0x55     ;  –
      OUT   PORTB, R16    ;  –
      CALL  DELAY         ;  –
      LDI   R16, 0xAA     ;  –
      OUT   PORTB, R16    ;  –
      CALL  DELAY         ;  –
      RJMP  L1            ;  –
```

- Notar que sin las dos instrucciones iniciales los valores asignados a PORTB no se envían a los pines de PORTB.

Programación de puertos I/O

- Para convertir a PORTB en puerto de entrada el registro DDRB se llena ahora con ceros y ahora se pueden leer los datos presentes en el puerto.

- Cada PIN tiene un elevador de voltaje que se activa al escribir 1s en PORTx. De esta forma cuando nada esta conectado se evita la indeterminación. Analice el siguiente código

```
      LDI   R16, 0xFF    ;  -
      OUT   DDRB, R16    ;  -
      OUT   DDRD, R16    ;  -
      LDI   R16, 0x00    ;  -
      OUT   DDRD, R16    ;  -
L2:   IN    R16, PIND    ;  -
      LDI   R17, 5       ;  -
      ADD   R16, R17     ;  -
      OUT   PORTB, R16   ;  -
      OUT   PORTB, R16   ;  -
      RJMP  L2           ;  -
```

Programación de puertos I/O

- El circuito de entrada del AVR tiene un retardo de un ciclo de máquina que es necesario compensar.

```
LDI   R16, 0x00   ;   –
OUT   DDRB, R16   ;   –
NOP               ;   –
IN    R16, PINB   ;   –
STS   0x100, R16  ;   –
```

Si se omite NOP, el primer valor que se escriba en PINB será el último leído en PORTB.

- En algunas aplicaciones es necesario modificar solo uno o dos bits del puerto. En el AVR es posible modificar individualmente cada uno de los bits del puerto.

Programación de puertos I/O

- **SBI ioReg, bit_num** Pone un uno en el bit **bit_num** del registro **I/O ioReg**.

```
        SBI   PORTB, 5   ;                                    –
```

- **CBI ioReg, bit_num** Pone un cero en el bit **bit_num** del registro **I/O ioReg**.

```
        SBI     DDRB, 2    ;                                  –
AGAIN:  SBI     PORTB, 2   ;                                  –
        CALL    DELAY      ;                                  –
        CBI     PORTB, 2   ;                                  –
        CALL    DELAY      ;                                  –
        RJMP    AGAIN      ;                                  –
```

Esta capacidad de accesar a cada bit de forma independiente es una de las mejores características de los AVR.

Programación de puertos I/O

- Bosqueje la salida de los bits 0 y 3 del puerto B. Considere que la rutina DELAY tiene una duración de 10ms.

```
        SBI     DDRB, 0     ;  –
HERE:   SBI     PORTB, 0    ;  –
        CALL    DELAY       ;  –
        CBI     PORTB, 0    ;  –
        CALL    DELAY       ;  –
        RJMP    HERE        ;  –

        SBI     DDRB, 3     ;  –
HERE:   SBI     PORTB, 3    ;  –
        CALL    DELAY       ;  –
        CALL    DELAY       ;  –
        CBI     PORTB, 3    ;  –
        CALL    DELAY       ;  –
        RJMP    HERE        ;  –
```

Programación de puertos I/O

Es posible tomar decisiones de acuerdo al valor que tiene el bit de un registro I/O.

- **SBIS** ioReg, bit (Skip if Bit in I/O register Set) La instrucción verifica el estado del bit en el registro I/O y salta la instrucción siguiente si el bit tiene el valor ALTO.
- **SBIC** (Skip if Bit in I/O register Cleared) La instrucción verifica del estado el bit en el registro I/O y salta la instrucción siguiente si el bit tiene el valor BAJO.
- Ejemplo

```
        CBI     DDRB, 3     ;                   –
        SBI     DDRC, 5     ;                   –
HERE:   SBIC    PINB, 3     ;                   –
        RJMP    HERE        ;                   –
        SBI     PORTC, 5    ;                   –
        CBI     PORTC, 5    ;                   –
        RJMP    HERE        ;                   –
```

Operaciones Aritméticas

- Cuando se realiza la adición de número de 16 bits se debe tener en cuenta la propagación del bit que se lleva del byte bajo al byte alto. Esta operación se llama suma de multiple byte.

- ADC (ADD with carry). Ejemplo, 3CE7 + 3B8D

```
LDI   R16, 8D   ;   –
LDI   R17, 3B   ;   –
LDI   R18, E7   ;   –
LDI   R19, 3C   ;   –
ADD   R18, R16  ;   –
ADC   R19, R17  ;   –
```

Operaciones Aritméticas

- Cuando se realiza la adición de número de 16 bits se debe tener en cuenta la propagación del bit que se lleva del byte bajo al byte alto. Esta operación se llama suma de multiple byte.

- ADC (ADD with carry). Ejemplo, 3CE7 + 3B8D

```
LDI   R16, 8D   ;   –
LDI   R17, 3B   ;   –
LDI   R18, E7   ;   –
LDI   R19, 3C   ;   –
ADD   R18, R16  ;   –
ADC   R19, R17  ;   –
```

- En AVR se tienen cinco instrucciones para resta. SUB, SBC, SUBI, SBCI, SBIW. SBC y SBCI son sumas tomando en cuenta el prestamo que se encuentra en la bandera C del registro de estatus.

Operaciones Aritméticas

- En todos los CPUs modernos la resta se realiza utilizando el método del complemento a 2. La resta se realiza como sigue
  - Asuma que C=0, se calcula el segundo complemento del sustraendo,
  - se realiza la suma de minuendo + sustraendo,
  - se invierte la bandera C.
- Ejemplo

3F	0	0	1	1	1	1	1	1			0	0	1	1	1	1	1	1
23	0	0	1	0	0	0	1	1			1	1	0	1	1	1	0	1
1C										1	0	0	0	1	1	1	0	0

Operaciones Aritméticas

- SUBI y SBIW se utilizan para restar constantes, SBIW permite utilizar numeros de 16 bits como minuendo, el sustraendo debe estar entre 0x00 y 0x63.
- Ejemplo

```
LDI   R21, 0x29   ;                    –
LDI   R22, 0x18   ;                    –
SUB   R21, R22    ;                    –
SUBI  R21, 0x18   ;                    –

LDI   R25, 0x29     ;                  –
LDI   R24, 0x17     ;                  –
SBIW  R25:R24, 0x18 ;                  –
```

Operaciones Aritméticas

- Ejemplo. Realizar la siguiente operación 0x2762 - 0x 1296.

```
LDI   R26, 0x62   ;                    –
LDI   R27, 0x27   ;                    –
LDI   R28, 0x96   ;                    –
LDI   R29, 0x12   ;                    –
SUB   R26, R28    ;                    –
SBC   R27, R29    ;                    –
```

- Notar que 0x62 - 0x96 = 0xCC y la bandera C = 1. Cuando SBC se ejecuta 0x27 - 0x12 -0x1 = 0x14. El resultado es 0x2762 - 0x1296 = 0x14CC.
- Después de la inversión de la bandera C por el CPU, debe verificarse el valor final de la bandera para detectar números positivos C = 0 y números negativos C = 1.