# Tema 4: 68HC11 – EVBU. Parte I

- Estructura del HC11
- Modos de direccionamiento
- Conjunto de instrucciones básico
- Operaciones matemáticas
- Saltos
- La Pila
- Subrutinas: Buffalo

**Bibliografía:**


- *"Introduction to 6811 programming" Fred G. Martin. Motorola. 1994.*
- *"M68HC11 Microcontrolers. Referente Manual". Motorola. 2002.*

# El Microcontrolador Motorola MC68HC11

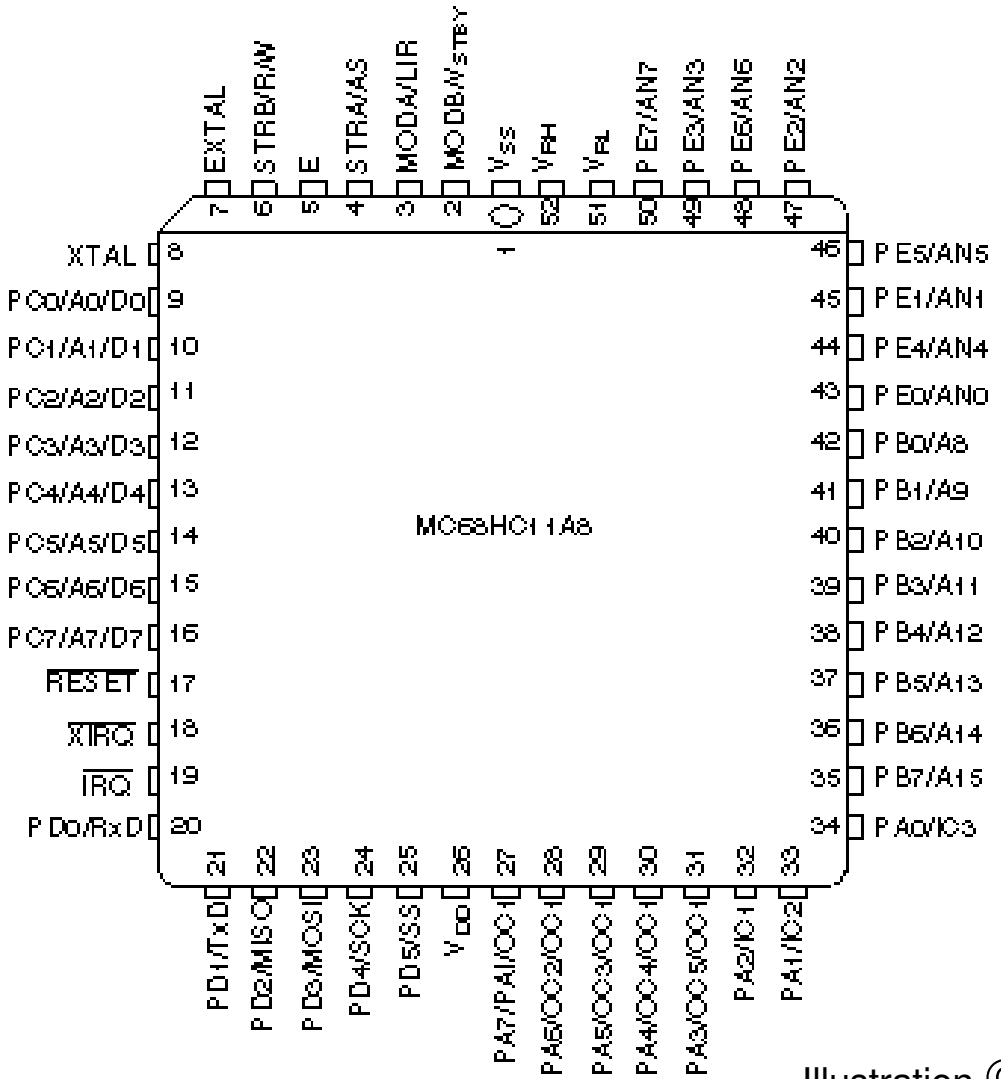
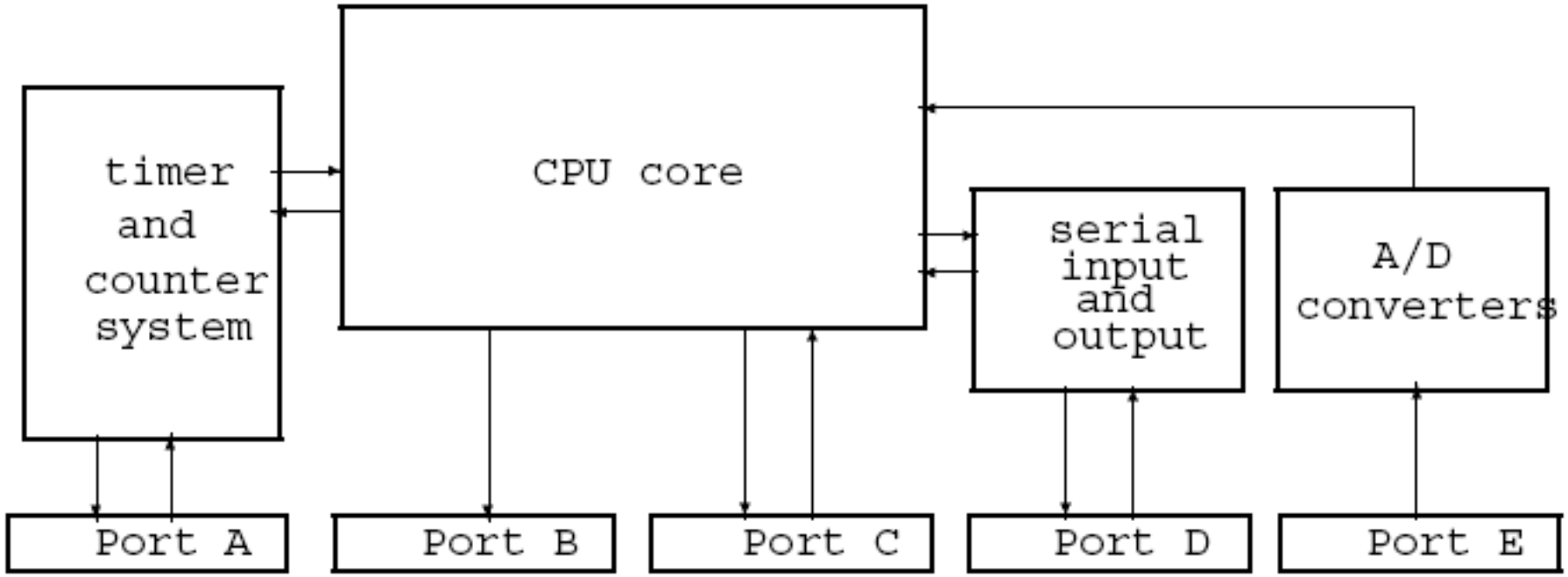



Illustration ©Motorola

- Se encuentra normalmente en un encapsulado PLCC (plastic leaded chip carrier) de 52 pines
- **Consume 15-35 mA a 5-volt (165 mW consumo máx.)**
- **Modo especial *sleep* que consume solo 250 µW**

# Esquema básico del 68HC11



# Estructura interna del Microcontrolador 68HC11

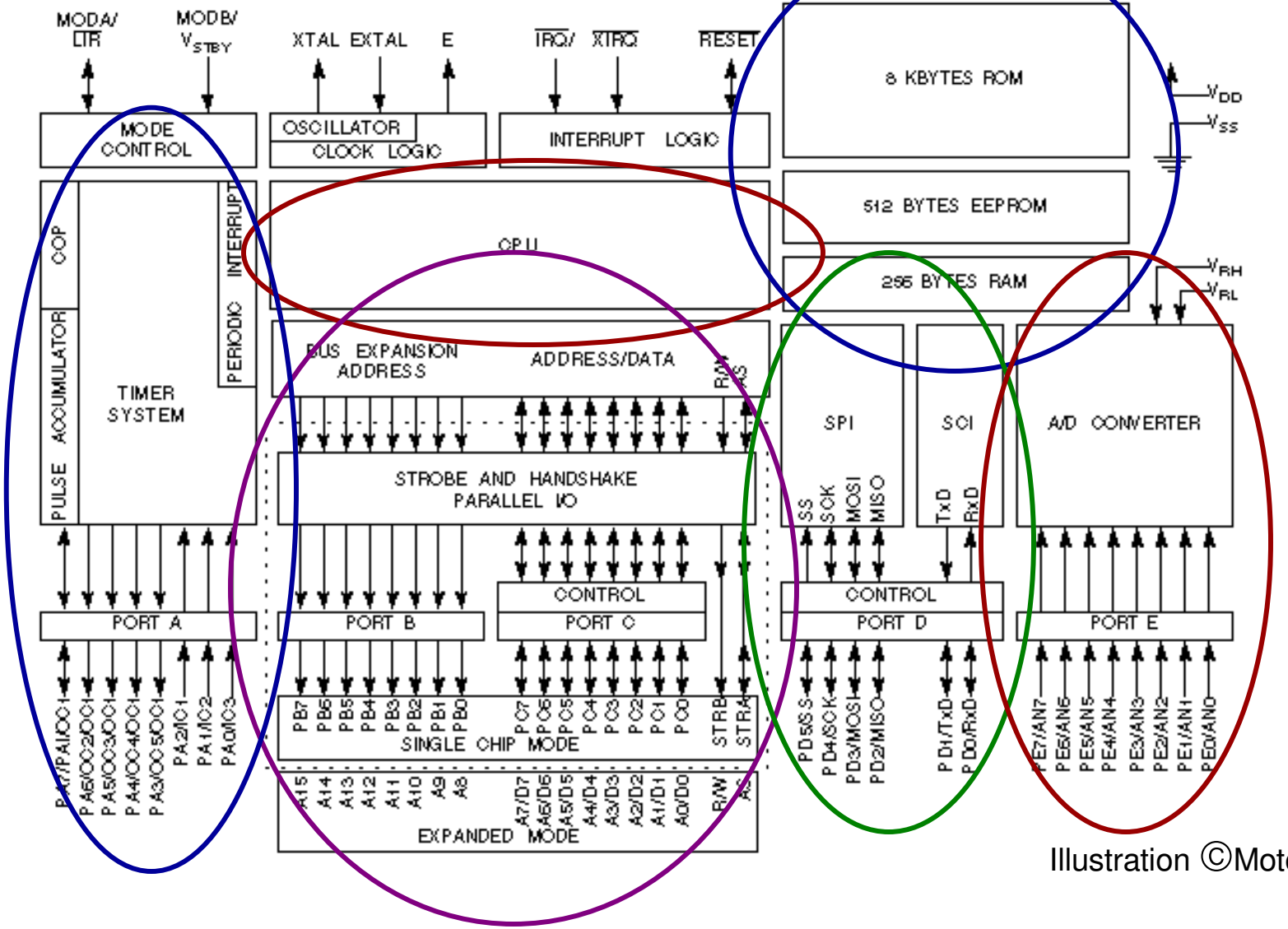
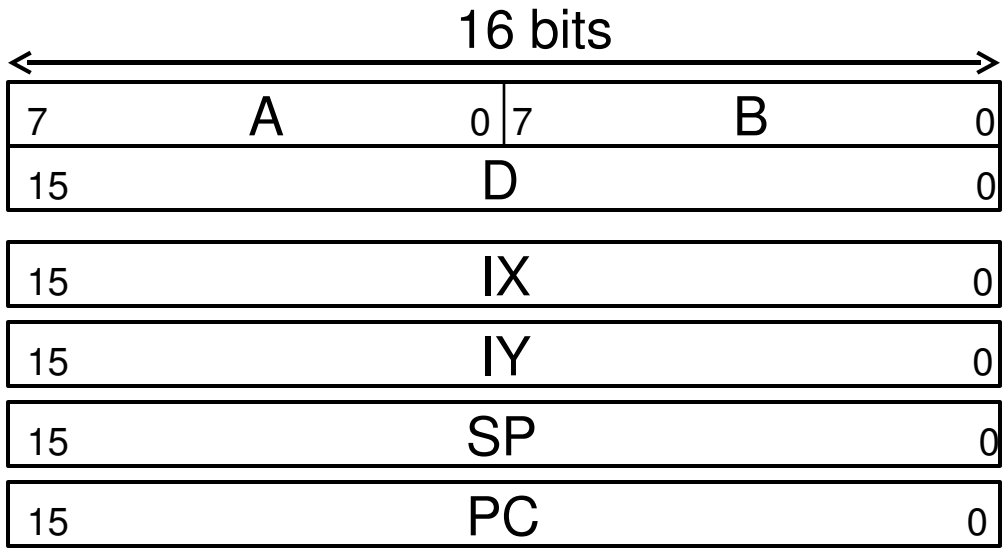



Illustration ©Motorola

# Registros de la arquitectura



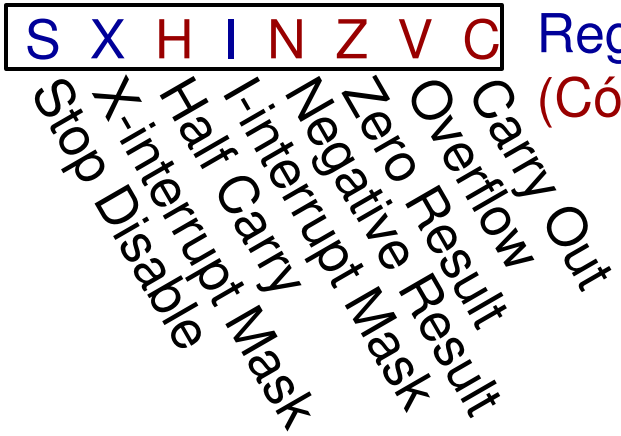

Registros Generales:
De 8-bit “A” y “B”
o uno de 16-bit “D”

Registros Index: para direccionar la memoria

Puntero Stack (pila)

Contador Programa

Los registros de propósito general (A,B,D) se llaman *accumuladores*.




Registro Estado - CC (Códigos Condición)

# !Más memoria!

- Memoria externa (o dispositivos emplazados (mapped) en memoria) pueden conectarse al HC11
  - Útil cuando se necesitan más de 512 bytes de RAM
  - Puede ser también ROM u otros dispositivos que tienen una interfaz del tipo de la memoria

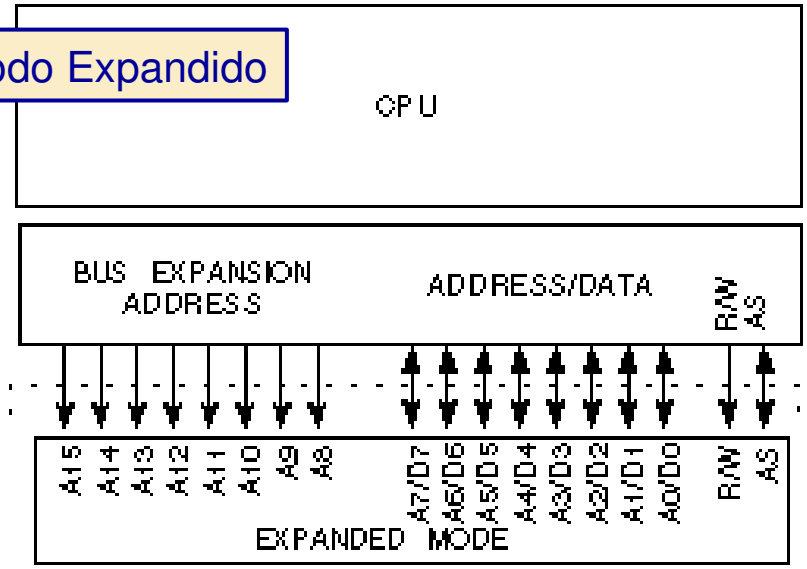
- **La memoria se conecta mediante un bus de direcciones y un bus de datos**

# Tipos de memoria

- **RAM - Random Access Memory**
  - Estática (SRAM) – Fácil de usar, rápida, cara
    - Disponible en una variante con batería
  - Dinámica (DRAM) - Requiere controlador de refresco, pero es barata (raramente usada con un microcontrolador)
- **ROM - Read Only Memory**
  - Mask-programmed – Programada en fábrica
  - PROM - Programada por el usuario con un programador
  - EPROM - PROM que puede ser borrada y reprogramada
  - EEPROM - Puede ser borrada eléctricamente sin quitarla y re-escrita

# Modo *Single Chip* vs. Expandido

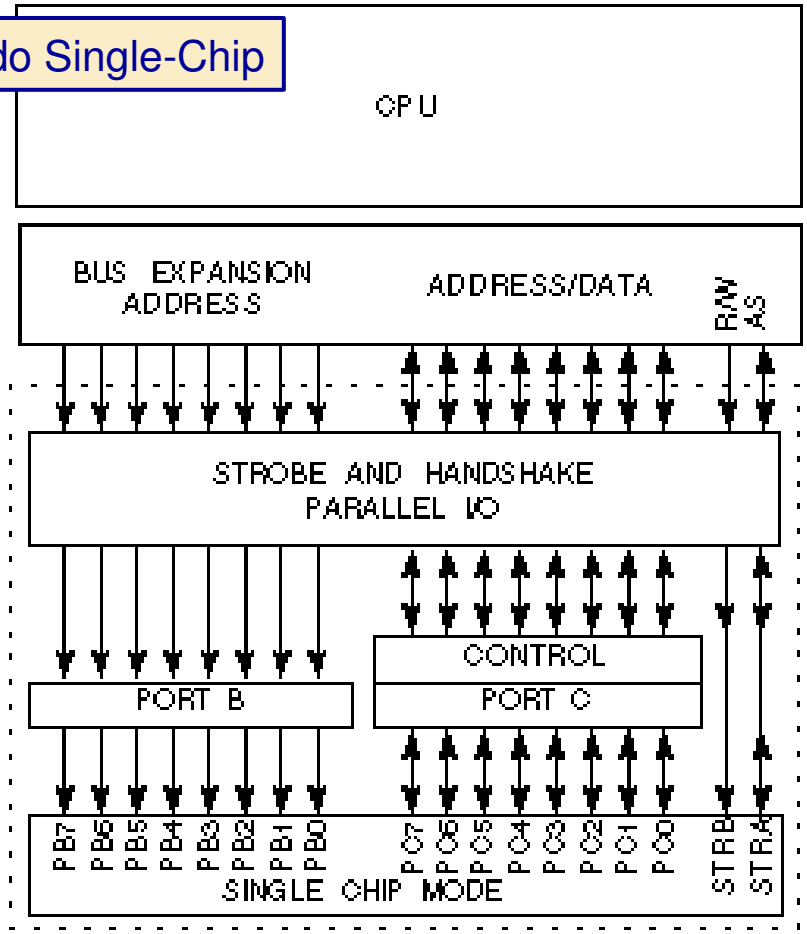
Modo Expandido




Interfaz Memoria Externa

- 16 bits de direcc. --> $2^{16}$ byte de espacio de direcc. (65536 or 64K bytes)
- 8 bits de datos (un byte)
- Nota: los buses están multiplexados

Modo Single-Chip




Dos Puertos Paralelos

Illustration ©Motorola

# Modos de direccionamiento del HC11

- Inherente
  - Para operaciones registro-a-registro - `INCA, ABA, LSRD`
- **Immediato**
  - Para constantes (número que no venga de memoria o de un acumulador) – **LDAA #13, LDAB #$C2, LDY #$23f6**
- **Directo (8 bits), Extendido (16 bits)**
  - Accede a una direccción específica de memoria – **LDAA $56, LDA $123A**
- **Indexado**
  - Accede a una posición de memoria que depende de un valor calculado – **LDAA 4,Y**
- **Relativo**
  - Solo usado en los saltos – más a continuación

# Instrucciones HC11

- Las instrucciones consisten de:
  - Opcode – Indica el tipo de instrucción
  - Operandos - 0 a 3 parámetros para la instrucción
    - *0*: `ABA` - suma acc. B al acc. A
    - *1*: `LDAA $34` - carga el valor de mem[$34] en A
    - *2*: `BSET $02, #5` - pone a uno bits 0 y 2 de mem[2]
    - *3*: `BRCLR $82, #$4, 14` - Salto a 14 si bit 2 de mem[$82] es cero

# Familias de Instrucciones

- Las instrucciones se agrupan en *familias*
  - Máquina basada en acumulador significa que los registros se especifican en la misma instrucción (inherente)
    - LDAA - carga ac. A
    - LDAB - carga ac. B
    - LDD - carga *double* en ac. D (A y B juntos)
    - LDS – carga el stack pointer SP
    - LDX - carga reg. índice X
    - LDY - carga reg. índice Y
  - Se conocen colectivamente como la instrucción LOAD

# Instrucciones de suma

- La familia ADD suma el valor de un registro, inmediato o memoria a un acumulador
  - ABA - suma acum. B a acum. A
  - ABX - suma acum. B a reg. índice X
  - ABY - suma acum. B a reg. Índice Y

    - ADDA #$13 - suma el número $13 al ac. A
    - ADDB $64 - suma mem[$64] al ac. B
    - ADDD 10,Y - suma mem[10+Y],[10+Y+1] al ac. D
      - Si Y = 30 y mem[40] = $12, mem[41]=$A2, entonces $12A2 se suma a D
- **Si ADD produce un *carry-out*, el bit C se pone a 1**

# Suma con acarreo, Resta

- Las instrucciones ADC suman el operando y el bit de acarreo al acumulador
  - Permite la suma de números mayores de 16 bits
    - ADCA #72 - suma 72 + bit acarreo al ac. A
    - ADCB $0112 - suma mem[$0112] + acarreo al ac. B

- **La Resta es similar a la suma**
  - SBA, SUBA, SUBB, SUBD, SBCA, SBCB

# Escritura en Memoria

- Escribimos en memoria cuando almacenamos los datos
- **La familia de instrucciones STORE son**
  - STAA $12,Y – almac. ac. A en mem[$12+Y]
  - STAB $3412 – almac. ac. B en mem[$3412]
  - STD $102 – almac. ac. D en mem[$102],[$103]
  - STX $3FF2 – almac. X en mem[$3FF2],[$3FF3]
  - STY 18,X – almac. Y en mem[18+X],[18+X+1]
  - STS $44 – almac. SP en mem[$44],[$45]

# Formato Instrucción máquina

| Instrucción | Lenguaje máquina (hex) |
|---|---|
| `LDAA #45` | `86 2d` |
| `LDAB 15,Y` | `18 e6 0f` |
| `ABA` | `1b` |
| `STAB $4232` | `f7 42 32` |
| `SUBB 3,X` | `e0 03` |

Nota: las instrucciones tienen diferentes longitudes

Asumiendo que LDAA está en la direcc. $2000, éstas serían las posiciones que ocuparían las instrucciones en memoria:

| | |
|---|---|
| $2000 | 86 |
| $2001 | 2d |
| $2002 | 18 |
| $2003 | e6 |
| $2004 | 0f |
| $2005 | 1b |

| | |
|---|---|
| $2006 | f7 |
| $2007 | 42 |
| $2008 | 32 |
| $2009 | e0 |
| $200A | 03 |
| $200B | ?? |

# Detalles Instrucción LDA

## Modos Direccionamiento, Código máquina, y Ejecución Ciclo-a-Ciclo:

| | LDAA (IMM) | | | LDAA (DIR) | | | LDAA (EXT) | | | LDAA (IND,X) | | | LDAA (IND,Y) | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Addr | Data | R/W | Addr | Data | R/W | Addr | Data | R/W | Addr | Data | R/W | Addr | Data | R/W |
| 1 | *OP* | *86* | *1* | *OP* | *96* | *1* | *OP* | *B6* | *1* | *OP* | *A6* | *1* | *OP* | *18* | *1* |
| 2 | *OP+1* | *ii* | *1* | *OP+1* | *dd* | *1* | *OP+1* | *hh* | *1* | *OP+1* | *ff* | *1* | *OP+1* | *A6* | *1* |
| 3 | | | | *00dd* | *(00dd)* | | *1* | *OP+2* | | *ll* | *1* | *FFFF* | — | *1* | *OP+2* |
| | | | | | | | | *ff* | *1* | | | | | | |
| 4 | | | | | | | *hhll* | *(hhll)* | *1* | *X+ff* | *(X+ff)* | | *1* | *FFFF* | — |
| | | | | | | | | *1* | | | | | | | |
| 5 | | | | | | | | | | | | | *Y+ff* | *(Y+ff)* | *1* |

OP – Búsqueda operación

ff - *offset* para modo indexado

ii – datos inmediatos

dd – direc. memoria para modo directo

hhll – direc. memoria para modo extendido

FFFF --- - ciclo sin actividad de la memoria (solo piensa…)
También como:
OP+1 ---
OP+2 ---

# Detalles Instrucción LDA (cont.)

## Modos Direccionamiento, Código máquina, y Ejecución Ciclo-a-Ciclo:

| Cycle | LDAA (IMM) | | | LDAA (DIR) | | | LDAA (EXT) | | | LDAA (IND,X) | | | LDAA (IND,Y) | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **Addr** | **Data** | **R/W** | **Addr** | **Data** | **R/W** | **Addr** | **Data** | **R/W** | **Addr** | **Data** | **R/W** | **Addr** | **Data** | **R/W** |
| 1 | *OP* | *86* | *1* | *OP* | *96* | *1* | *OP* | *B6* | *1* | *OP* | *A6* | *1* | *OP* | *18* | *1* |
| 2 | *OP+1* | *ii* | *1* | *OP+1* | *dd* | *1* | *OP+1* | *hh* | *1* | *OP+1* | *ff* | *1* | *OP+1* | *A6* | *1* |
| 3 | | | | *00dd* | *(00dd)* | *1* | *OP+2* | *ll* | *1* | *FFFF* | — | *1* | *OP+2* | *ff* | *1* |
| 4 | | | | | | | *hhll* | *(hhll)* | *1* | *X+ff* | *(X+ff)* | *1* | *FFFF* | — | *1* |
| 5 | | | | | | | | | | | | | *Y+ff* | *(Y+ff)* | *1* |

86=1000 0110 96=1001 0110 B6=1011 0110 A6=1010 0110 A6=1010 0110 18=0001 1000

| Cycle | LDAB (IMM) | | | LDAB (DIR) | | | LDAB (EXT) | | | LDAB (IND,X) | | | LDAB (IND,Y) | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **Addr** | **Data** | **R/W** | **Addr** | **Data** | **R/W** | **Addr** | **Data** | **R/W** | **Addr** | **Data** | **R/W** | **Addr** | **Data** | **R/W** |
| 1 | *OP* | *C6* | *1* | *OP* | *D6* | *1* | *OP* | *F6* | *1* | *OP* | *E6* | *1* | *OP* | *18* | *1* |
| 2 | *OP+1* | *ii* | *1* | *OP+1* | *dd* | *1* | *OP+1* | *hh* | *1* | *OP+1* | *ff* | *1* | *OP+1* | *E6* | *1* |
| 3 | | | | *00dd* | *(00dd)* | *1* | *OP+2* | *ll* | *1* | *FFFF* | — | *1* | *OP+2* | | *1* |
| 4 | | | | | | | *hhll* | *(hhll)* | *1* | *X+ff* | *(X+ff)* | *1* | *FFFF* | — | *1* |
| 5 | | | | | | | | | | | | | *Y+ff* | *(Y+ff)* | *1* |

C6=1100 0110 D6=1101 0110 F6=1111 0110 E6=1110 0110 E6=1110 0110

# Lenguajes de Alto Nivel

- Lenguajes de alto nivel, C o C++, son más fáciles de programar
  - Poseen una sintaxis y construcciones más naturales
  - Tienen soporte para sistemas de control y estructuras de datos complejos

- **Los programas de alto nivel deben convertirse en lenguaje ensamblador o lenguaje máquina antes de que puedan ejecutarse en un computador**
  - Un compilador realiza este trabajo
  - Pueden haber muchos diferentes programas ensamblador que se correspondan con el mismo programa de alto nivel

# ¿Por qué programar en ensamblador?

- "Un buen programador en ensamblador puede producir un código más rápido y reducido que un compilador"
  - No es realmente cierto siempre

- **Los lenguajes de alto nivel asumen un modelo abstracto del computador**
  - No representa todos los detalles internos
  - ¿Cómo escribimos código para programas que necesitan acceder a cada detalle del computador (p.e. device drivers)?
    - !Lenguaje ensamblador!

# Comparaciones

- Lenguaje de alto nivel
  - Toda los accesos a memoria se hacen a través de variables
  - Estructuras de control y datos complejas
  - No hay acceso directo a los registros de la CPU

- **Lenguaje ensamblador**
  - Acceso a los registros de la CPU
  - Accesos a memoria a través de ddireccionamiento directo o variables
  - La única estructura de control: GOTO condicional
  - No hay estructuras de datos complejas

- **Lenguaje Máquina**
  - Como el ensamblador, pero sin variables

# Programas en Lenguaje Ensamblador

- **Cada línea de un programa en ensamblador es (normalmente) una instrucción**
  - Formato :

| Etiqueta | Instrucción | Operandos | ;comentarios |
|---|---|---|---|
| ADDIT | ADDA | #10 | ; Suma 10 al contador |
| | STAA | Total | ; almacena valor total |

- **Normalmente, se usan tabuladores para separar campos, pero pueden ser espacios**
  - Si en una línea no hay una etiqueta, se pondrá un tabulador o un espacio

# Comentarios

- Los comentarios se denotan de dos maneras
  - Cualquier cosa después de un '**;**' es un comentario

```
Hello   ABA                 ;suma a total
```

  - Una línea iniciada con un '*****' en la primera columna es un comentario
    - `* Este es una línea completa de comentario!`
- **Utiliza comentarios todo el tiempo**
  - Cada línea debería comentarse!
  - Debería ser posible entender el código leyendo solo los comentarios

# Etiquetas

- Las etiquetas dan un nombre particular a una línea
  - Se usa este nombre para facilitar la programación
- **Las etiquetas se usan por dos razones principales:**
  - Para crear variable nombrando posic. de memoria
  - Para marcar una instrucción que se usa como el destino de una instrucción de salto

```
LAZO        LDAB        #99    ; principio de lazo
            BRA         LAZO
```

Las etiquetas deben comenzar en la primera columna.
Pueden terminar en ':'

# Más cosas: Directivas ensamblador

Asumimos que la variable *total* se encuentra en la posic. $2500

```
•Ese es un programa para sumar a total unos valores,
 el resultado se almacena en el ac. A

                    ORG         $2400

BEGIN       2400    LDAA        #0          ;Pone a cero total
= 2400      2402    STAA        $2500       ;Pone a cero total
            2405    ABA                     ;Suma a total
Fudge:      2406    SUBA        #1          ;Resta uno
 = 2406     2408    STAA        $2500       ;escribe total
```

Comienzo código posición $2400

Nota: las etiquestas solo comienzan en la columna 1

# Declarando constantes

Usa directiva 'EQU' para declarar constantes

```
PI    EQU    31        ;PI toma el valor de 31
      LDAA   $2500     ;carga diametro en ac. A
      LDAB   #PI       ;carga PI en ac. B
      MUL              ;multipica PI por ac. A
      STD    $2502     ;Almacena resultado en memoria
```

Nota: etiqueta

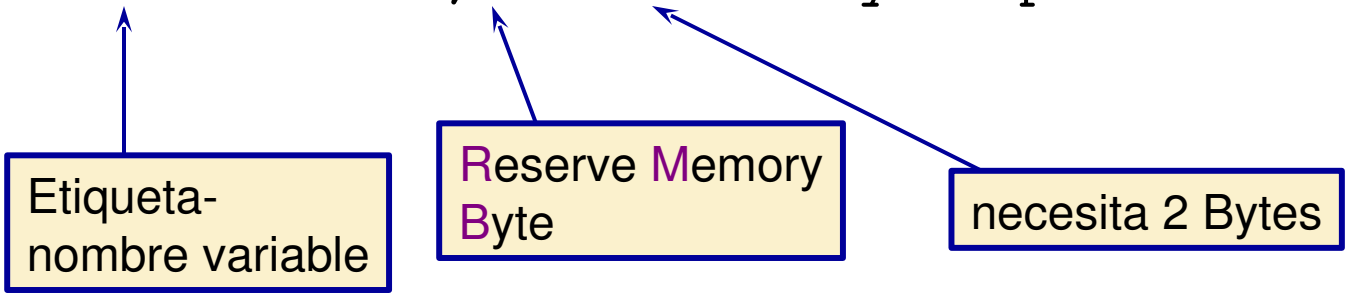
Debe usar '#'
- Carga valor 31 en Ac. B

~~LDAB PI~~
~~- Carga mem[31] en Ac. B~~

# Reserva de espacio para variables

Para declarar una variable en memoria, necesitamos reservar espacio para ella

```
        ORG    $2000
RADIO RMB    2; reserva 2 bytes para radio
```




```
STD    RADIO ; almac. ac. D en memoria
```

El ensamblador sustituye una posición de memoria de la variable 'RADIO' por $2000

# Reserva de espacio

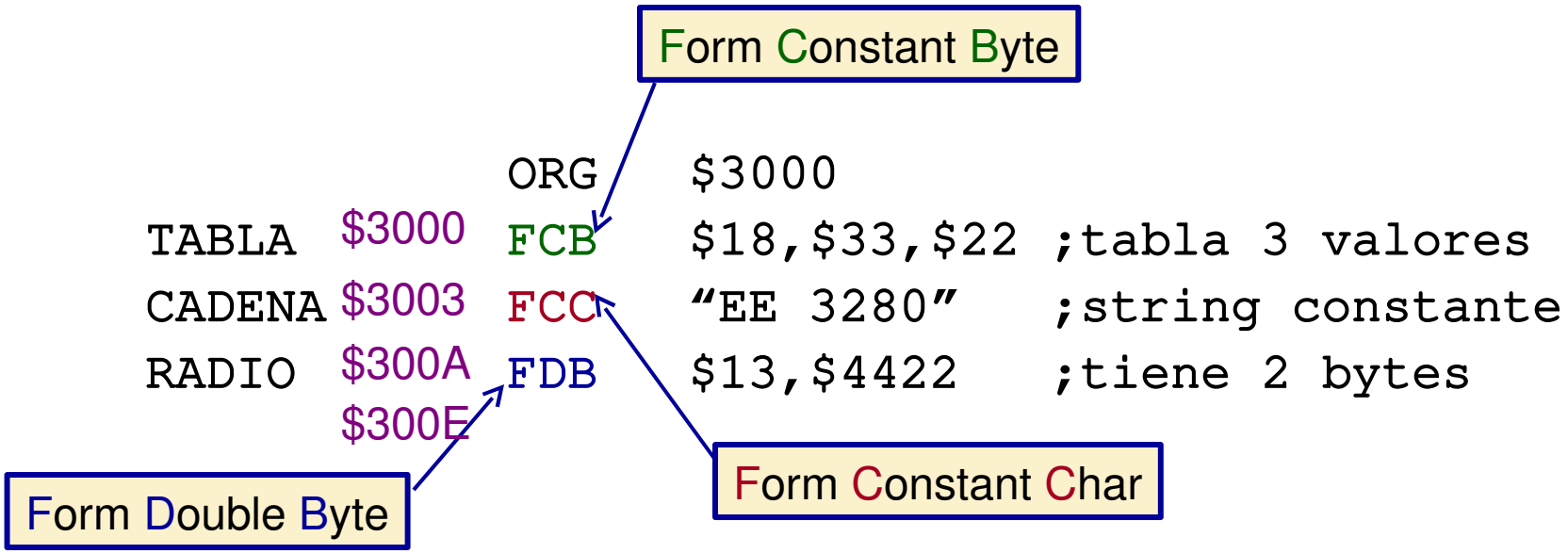
Las etiquetas nos ayudan a organizar la memoria

```
                ORG         $2600
BUFFER  $2600   RMB         40      ;reserva 40 bytes
BUFFER2 $2628   RMB         10      ;reserva 10 bytes
                ORG         $2000
BUFFER3 $2000   RMB         20      ;reserva 20 bytes
        $2014
```

- BUFFER tiene el valor $2600 (reserva espacio en las direcc. $2600 - $2627)
- BUFFER2 tiene el valor $2628 (reserva espacio en las direcc. $2628 - $2632)
- BUFFER3 tiene el valor $2000 (reserva espacio en las direcc. $2000 - $2013)

# Reserva de espacio con valores iniciales

A veces queremos variables con valores iniciales




- Variables iniciadas de este modo pueden ser cambiadas después por el programa
- Variables son iniciadas solo en la carga, no antes de cada ejecución

Nota: Cuando los datos son del tipo "immediate," no necesitan '#' en las directivas de ensamblador

# Escribir Programas

- Escribir un programa que sume un byte de una posición de memoria (valor inicial 3) a un byte en una segunda posición (valor inicial 10)
  - Primero:

```
ORG   $2500
NUM1  FCB     3
NUM2  FCB     10
```

Lee mem[NUM1]
= mem[$2500]
= 3

  - Ahora sumarlo mediante el acumulador

```
LDAA NUM1          ; posición: $2500
ADDA NUM2          ; posición: $2501
```

  - Escribe el resultado

```
STAA NUM2          ; posición: $2501
```

# Aritmética Multiprecisión

- ¿ Cómo podemos sumar números con más de 8 bits cada uno?

```
    1 1
  $22C7
+ $3159
  $5420
```

Resultado deseado

```
            ORG     $2100
RESULT      RMB     2
            LDAA    #$C7
            ADDA    #$59    ;suma byte bajo
            STAA    RESULT+1
            LDAA    #$22
ADCA        ADDA    #$31    ;suma byte alto
            STAA    RESULT
```

```
    1
  $22C7
+ $3159
  $5320
```

Resultado Real

ADCA - Add with carry to A

Nota: Chequear para asegurarse que STAA y LDAA no cambian el flag de carry !

Resta: es similar, excepto que el flag C es un flag de *borrow*

# Multiplicación

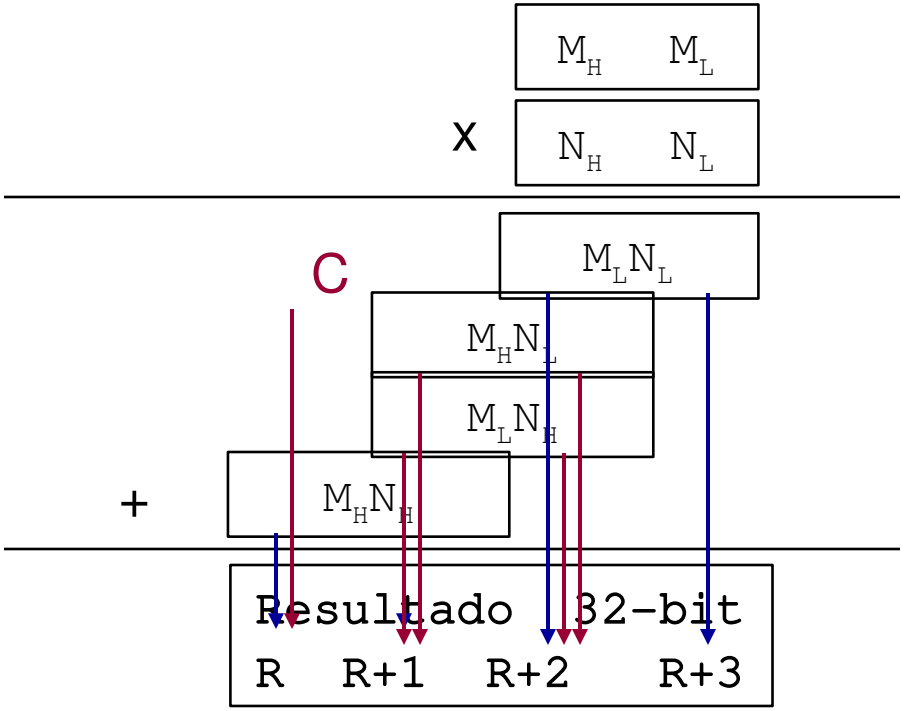
La instrucción MUL multiplica ac. A por ac. B (8 bits X 8 bits) y pune el resultado en D (16 bits)

```
        ORG     $2300
RESULT  RMB     2
        LDAA    #$15
        LDAB    #$32
        MUL
        STD     RESULT
```

Note: Ac. A y B son destruidos por la multiplicacion!

# Multiplicación Multiprecisón

Multiplica números de 16-bit, $M_H M_L$ $N_H N_L$




1. Multiplica $M_L$ x $N_L$ --> D
2. Copia D to R+2, R+3
3. Multiplica $M_H$ x $N_H$ --> D
4. Copia D to R, R+1
5. Multiplica $M_H$ x $N_L$ --> D
6. Suma D a R+1, R+2
7. Suma con carry to R
8. Multiplica $M_L$ x $N_H$ --> D
9. Suma D to R+1, R+2
10. Suma con carry to R

# División

La división entera produce un cociente y un resto

```
00 09  = 2, resto 1
-----
00 04
```

La instrucción `IDIV` divide acum. D (16 b) por el reg. índice X (16-bit)

El cociente se queda en X y el resto en D

```
        ORG     $2300
        LDD     #$0009
        LDX     #$0004
        IDIV                ; divide 9/4
        STX     $2000       ; cociente -> $2000
        STD     $2002       ; resto -> $2002
```

```
>G 2300
>MD 2000
2000  00 02 00 01 ...
```

# Instrucciones de Transferencia/Intercambio

- Como el cociente queda en X, ¿cómo podemos pasarlo al acumulador D para usarlo?
  - `XGDX` – Intercambia los contenidos X y D
  - `XGDY` – Intercambia los contenidos Y y D

- **Intercambios entre otros registros**
  - `TAB` – Transfiere (copia) A a B
  - `TBA` – B en A
  - `TSX, TSY` – Transfiere (stack pointer+1) a X (Y)
  - `TXS, TYS` – Transfiere [X (o Y)-1] al stack pointer
  - `TPA` – Transfiere Registro Status Proc. al acum. A
  - `TAP` – Transfiere acum. A al Registro Status Proc.

# Operaciones Lógicas

- `AND` – AND lógico
- `OR` – OR lógico
- `EOR` – OR exclusivo
- `COM` – complemento (bit)

Todas las operaciones lógicas son bitwise

```
      10100101
OR    00110011
      10110111
```

# Operaciones de desplazamiento

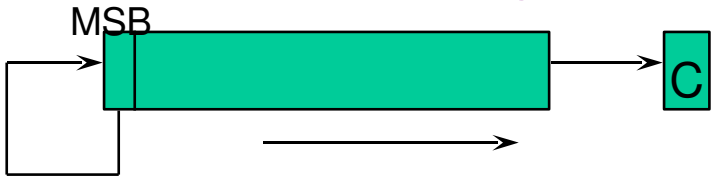
## *Logical Shift Right*


00001111 ⇨ 00000111
LSR
C = 1

## *Logical Shift Left*


00001111 ⇨ 00011110
LSL
C = 0

## *Arithmetic Shift Right*




00001111 ⇨ 00000111
ASR
C=1

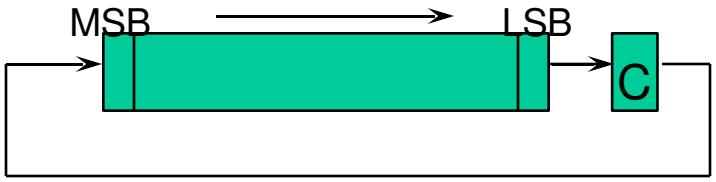
10100011 ⇨ 11010001
ASR
C=1

## *Arithmetic Shift Left*


00001111 ⇨ 00011110
ASL
C=0

# Operaciones de Rotación

*Rotate Right*




C=0

00001111 ⇨ 00000111 C=1

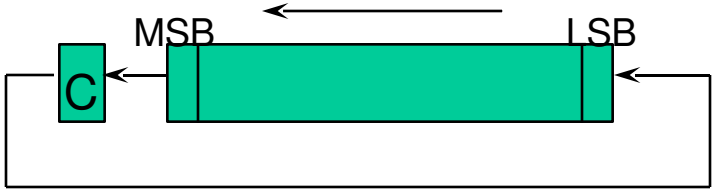
ROR

C=1

10100010 ⇨ 11010001 C=0

ROR

*Rotate Left*




C=0

10001111 ⇨ 00011110 C=1

ROL

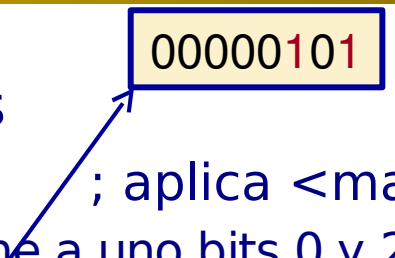
C=1

00100010 ⇨ 01000101 C=0

ROL

# Manipulación de Bits

- Trabaja con bits individuales
  - `BSET <opr>, <mask>` ; aplica <mask> bits
    - `BSET $2000,#5` ; pone a uno bits 0 y 2 de la dir. $2000




- Si M[$2000] fuera 00101001, sería ahora 00101101

- `BCLR <opr>,<mascara>` ; pone a cero <mascara> bits

- **Nota:**

  La numeración de los bits comienza en cero, no en uno!

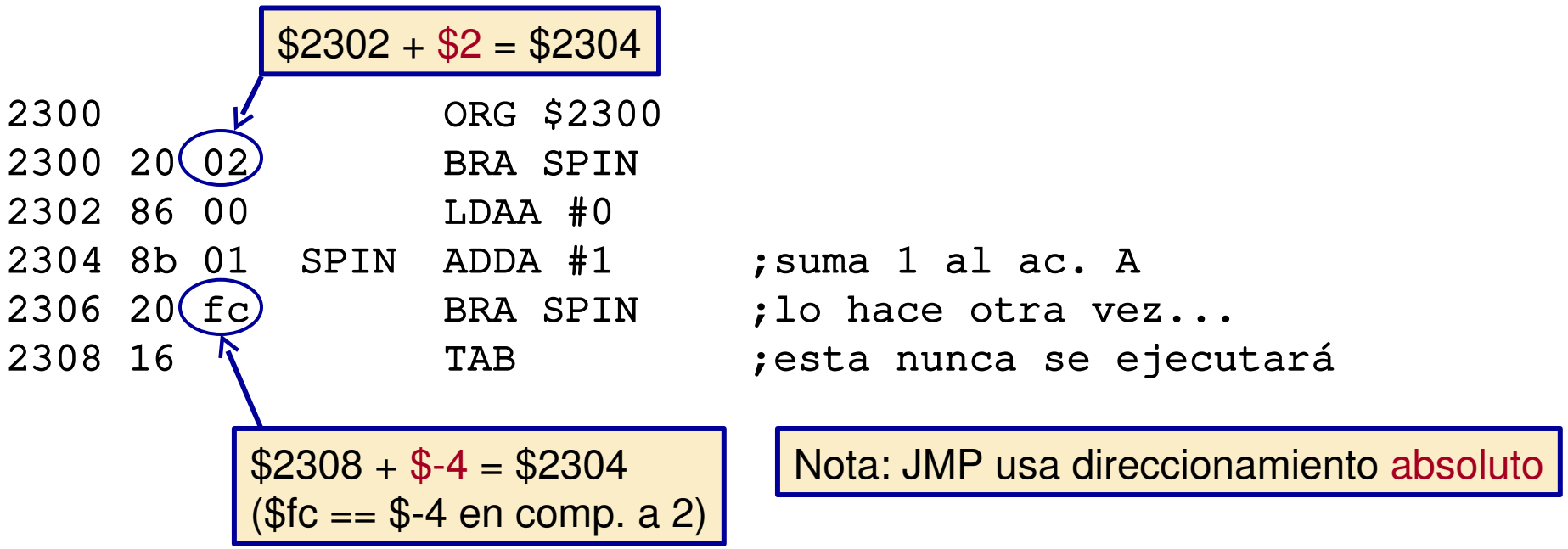
# Lazos y Saltos

- Lazos y otras estructuras de control se hacen mediante instrucciones tipo GOTO
  - Los saltos en el HC11
  - Los saltos incondicionales usan las instrucciones `BRA` o `JMP`

```
        ORG    $2300
        BRA    SPIN   ;se salta la siguiente instrucción
        LDAA   #0     ;Malo, no ejecuta esta instrucción
SPIN    ADDA   #1     ;suma uno al acumulador A
        JMP    SPIN   ;lo repite otra vez
```

# Direccionamiento en los saltos

- Los saltos usan direccionamiento relativo
  - Especificar la dirección destino para el salto
    - Expresado como un número de bytes desde el principio de la Siguiente instrucción
    - En forma de 8-bits en complemento a dos



# Códigos de Condición

- Los códigos de condición son activados por muchas operaciones en función de su resultado
  - `ADDA $2020 ; Si hay acarreo, el Carry se activa`
- Los códigos de condición son válidos para instrucción posterior a la que los ha activado
  - Permanecen válidos hasta que son cambiados por otra instrucción

```
ADDA   $2020
BCC    FOO
```

Activa códigos condición C, Z, N, V, y H

Mira el código de cond. C de la instruc. ADD

- Códigos de Condición Basicos
  - C - Carry bit (para operaciones aritméticas)
  - Z - Zero (verdadero si la operación produce cero)
  - N - Negative (verdadero si operación produce un resultado negativo)
  - V - Overflow (para operaciones aritmeticas)
  - H - Half Carry del bit 3 al 4

# Instrucciones Bxx

| | | |
|---|---|---|
| BCC | Carry Clear | $\overline{C}$ |
| BCS | Carry Set | C |
| BEQ | Equal | Z |
| BNE | Not Equal | $\overline{Z}$ |
| BMI | Minus | N |
| BPL | Plus | $\overline{N}$ |
| BVC | Overflow clear | $\overline{V}$ |
| BVS | Overflow set | V |

Las instrucciones anteriores tienen sentido si la instrucción previa a restado N1 - N2

| | | |
|---|---|---|
| BHI | Higher | $\overline{C} \cdot \overline{Z}$ |
| BHS | Higher or Same | $\overline{C}$ |
| BLO | Lower | C |
| BLS | Lower or Same | C + Z |

Para aritmética sin signo

| | | |
|---|---|---|
| BGE | Greater or Equal | $N \cdot V + \overline{N} \cdot \overline{V}$ |
| BGT | Greater | $(N \cdot V \cdot + \overline{N} \cdot \overline{V}) \cdot \overline{Z}$ |
| BLE | Less than or Equal | $Z + N \cdot \overline{V} + \overline{N} \cdot V$ |
| BLT | Less than | $N \cdot \overline{V} + \overline{N} \cdot V$ |

Para aritmética Comp. a 2

# Instrucción de Comparación

- La instrucción compara se usa para activar los códigos de condición
  - `CMPA $2030`

Activa los códigos de condición como si se hubiera realizado A - mem[$2030]

- CMP prepara los CC's para la siguiente instrucción de salto condicional

```
        LDAA    #0
LOOP    ADDA    #1
        CMPA    #32
        BNE     LOOP
```

Lazos hasta A == 32

Chequea para Z==0

Variantes: CMPA, CMPB, CBA, CPD, CPX, CPY

# Usando CMP y Bxx

Calcula A - <Z>

Executa la instrucción: **CMPA <Z>** seguida por una instrucción Bxx , significa:

| Instrucción | Salta si: |
|---|---|
| BEQ | A = Z |
| BNE | A ≠ Z |
| BLT | A < Z |
| BLE | A ≤ Z |
| BGT | A > Z |
| BGE | A ≥ Z |

Para enteros con signo

| Instrucción | Salta si: |
|---|---|
| BEQ | A = Z |
| BNE | A ≠ Z |
| BLO | A < Z |
| BLS | A ≤ Z |
| BHI | A > Z |
| BHS | A ≥ Z |

Para enteros sin signo.

# Instrucción TST

TSTA ; activa CC's como si A - 0 se hubiera ejecutado

| Instrucción | Salta si: |
|---|---|
| BEQ | A = 0 |
| BNE | A ≠ 0 |
| BLT | A < 0 |
| BLE | A ≤ 0 |
| BGT | A > 0 |
| BGE | A ≥ 0 |

TSTA es lo mismo que CMPA #0

Nota: Solo BEQ, BNE tienen sentido para enteros sin signo

Variantes: TST <opr>, TSTA, TSTB

# Otras Instrucciones

- `INC, DEC` - Incrementa o Decrementa 1
- `CLR` - limpia (pone a cero)
- `NEG` - negativo
  - `BRCLR, BRSET` – **Salta si los valores especificados en la máscara con cero o uno, respectivamente**
    - `BRSET  $20 #$81 SKIP ;salta si los bits 0 y 7`
      `$81 = 10000001 ;de Mem[$20] están a uno`
    - `BRCLR  5,X #$04 SKIP`
      - **Soporta sólo modos de direcc. directo e indexado**
      - **Algunos ensambladores pueden tener sintaxis diferente**
  - `SWI` – **Interrupción Software**
    - Causa la parada del programa y devuelve el control al monitor Buffalo

# Tiempo de ejecución de un lazo

Cada ciclo del HC11 corresponda a un ciclo del Reloj E

El reloj E se obtiene al dividir la señal de reloj externa por 4.

Típicamente, el reloj E es de 1-4 MHz. Asumamos 2 MHz en este ejemplo

```
                        ciclos
      LDX   #10000        3
LOOP  NOP                 2   }
      NOP                 2   }  ejecutado
      DEX                 3   }  10000 veces
      BNE   LOOP          3   }
```

*Y si queremos 50ms exactos?*

Tiempo= [3 + (10000*10) ciclos]*1/2MHz

= (100003 cycles) * 5E-7 s

= 0.0500015 s

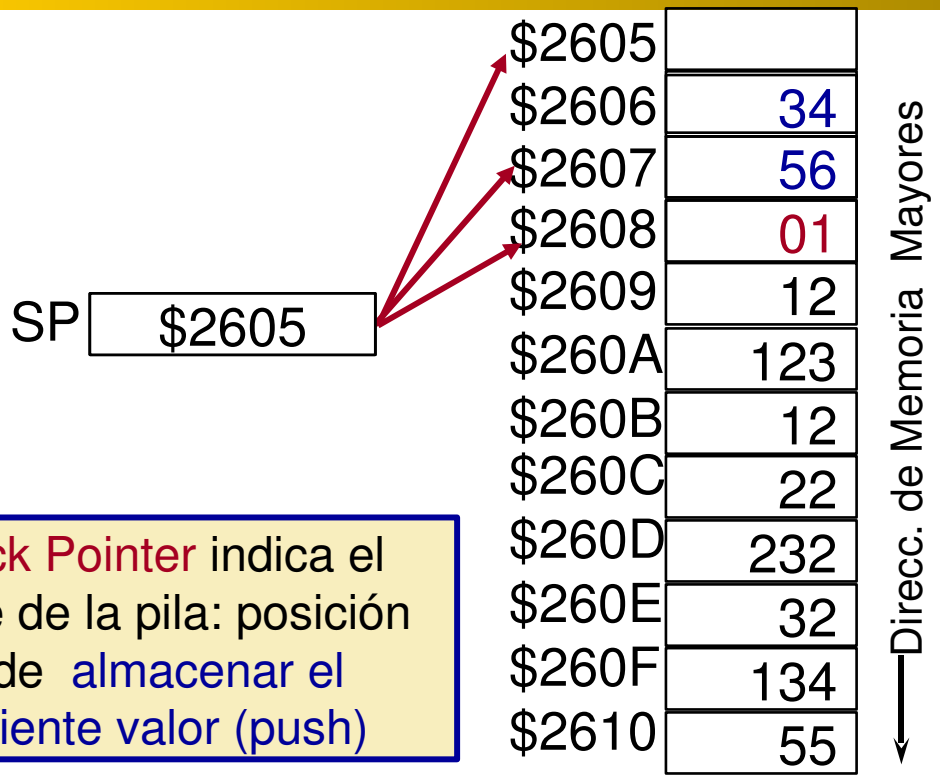
≈ 50 ms

# Estructuras de Datos

- Los lenguajes de alto nivel proporcionan útiles estructuras de datos
  - Pilas, arrays, cadenas, colas, árboles, listas

- **El lenguaje ensamblador también, pero el programador tiene que correr con los dettalles**
  - Un simple acceso a un array o cadena puede requerir varias instrucciones
  - Poner las funciones más usadas en librerías nos ayudará a realizar este trabajo

# Pila

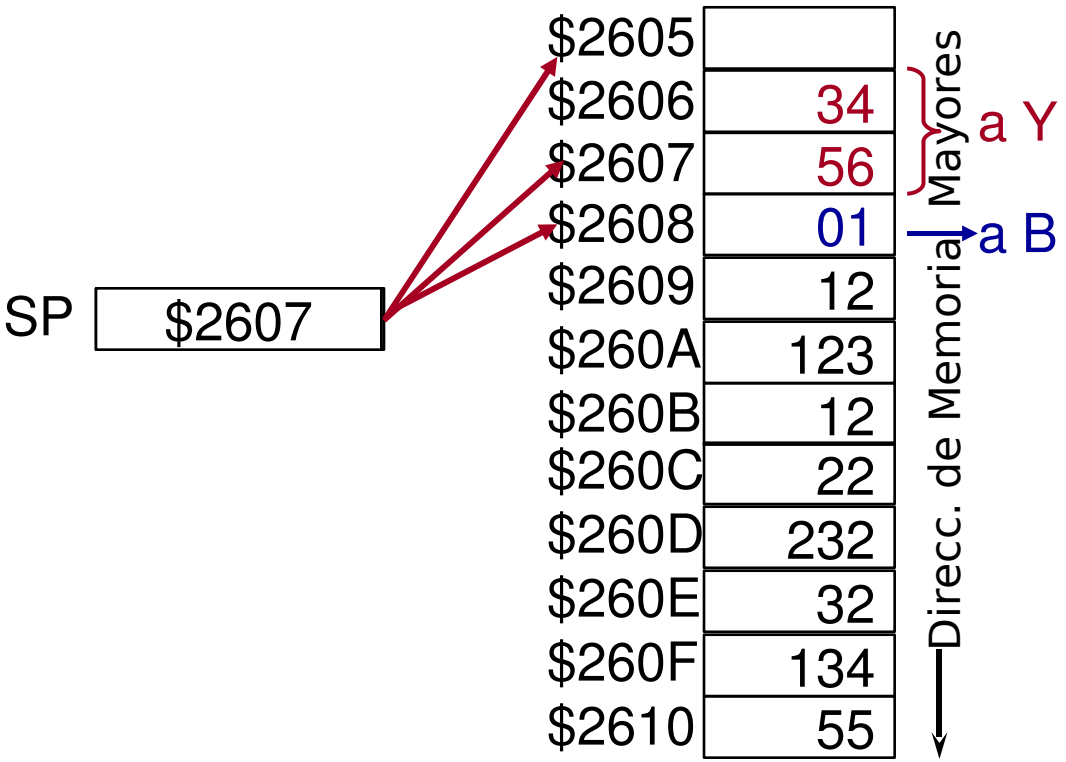



Stack Pointer indica el tope de la pila: posición donde almacenar el siguiente valor (push)

*push* un nuevo valor: Escribir en el tope de la pila y decrementarla por *el tamaño del operando*

```
   LDAA #1
   PSHA        ;Mem[SP] <- A, SP <- SP-1
*              ;SP == $2607
   LDX  #$3456
   PSHX        ;Mem[SP-1,SP] <- X, SP <- SP-2
*              ;SP == $2605
```

# Pila



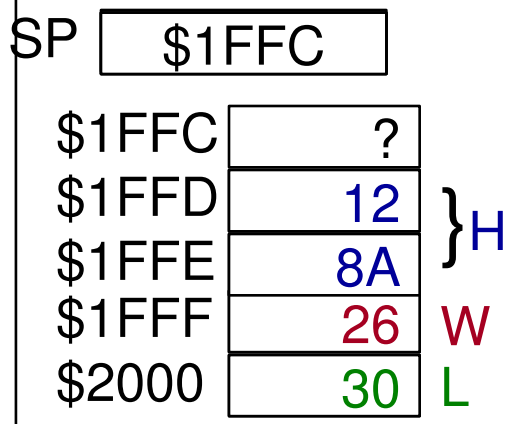

pull (o pop) un valor: Copia del tope de la pila más uno e incrementa el stack pointer por el tamaño del operando

```
   PULY        ;Y <- Mem[SP+1,SP+2], SP <- SP+2
*              ;SP == $2607, Y == $3456
   PULB        ;B <- Mem[SP+1], SP <- SP+1
*              ;SP == $2608, B == $01
```

# Ejemplo de Apilado

Tu codigo va a llamar a otra zona del codigo.
Tu código debe apilar los valores de las variables L (1 byte), W (1 byte), y H (2 bytes) en la pila en este orden.

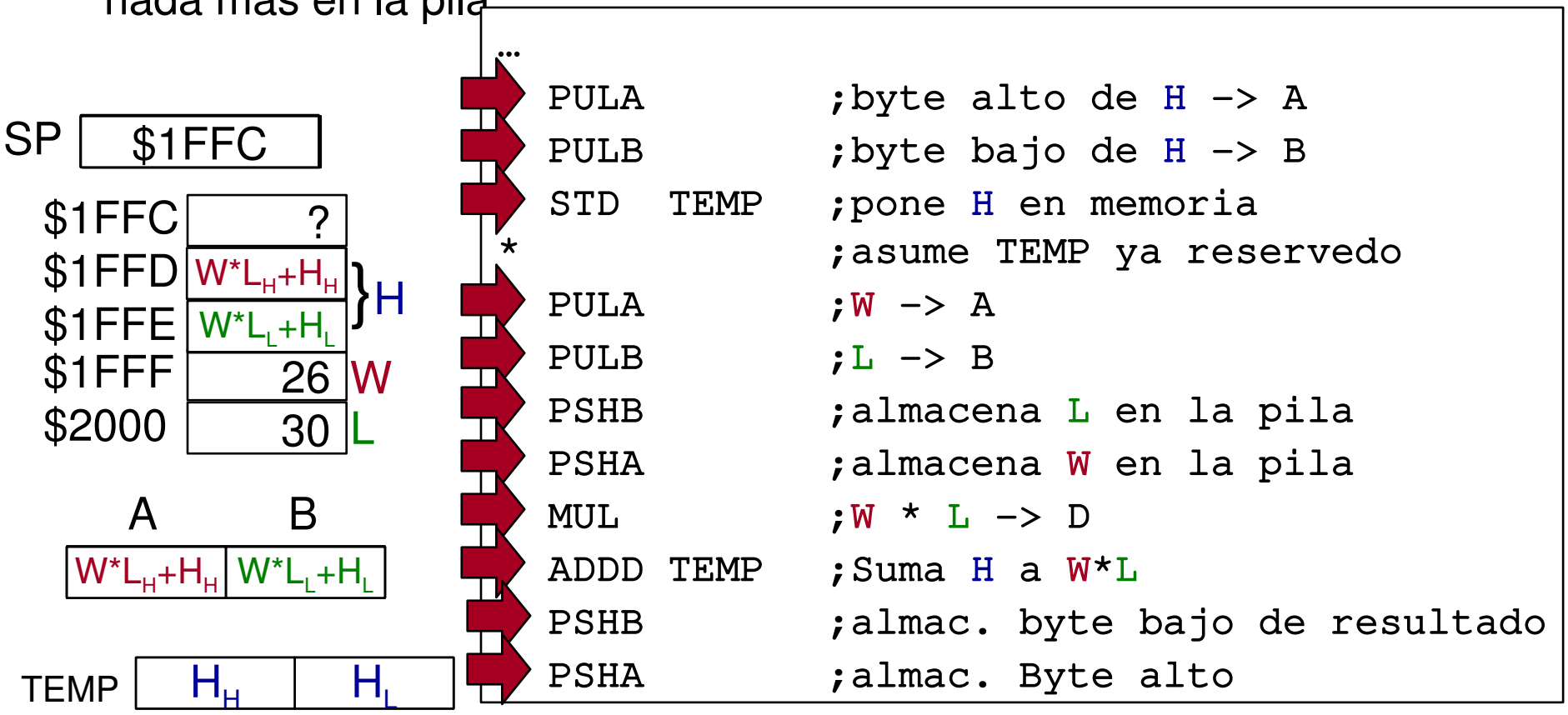
```
L   FCB   $30
W   FCB   $26
H   FDB   $128A
    LDS   #$2000 ;inicia el stack pointer
*                ;(esto es normalmente
*                ; hecho en el arranque)
    LDAA  L      ;Apila L en la pila
    PSHA         ;después de apilar,SP==$1FFF
    LDAA  W      ;Apila W en la pila
    PSHA         ;después de apilar,SP==$1FFE
    LDX   H      ;Apila H e la pila
    PSHX         ;después de apilar,SP==$1FFC
```



# Ejemplo de Desapilado

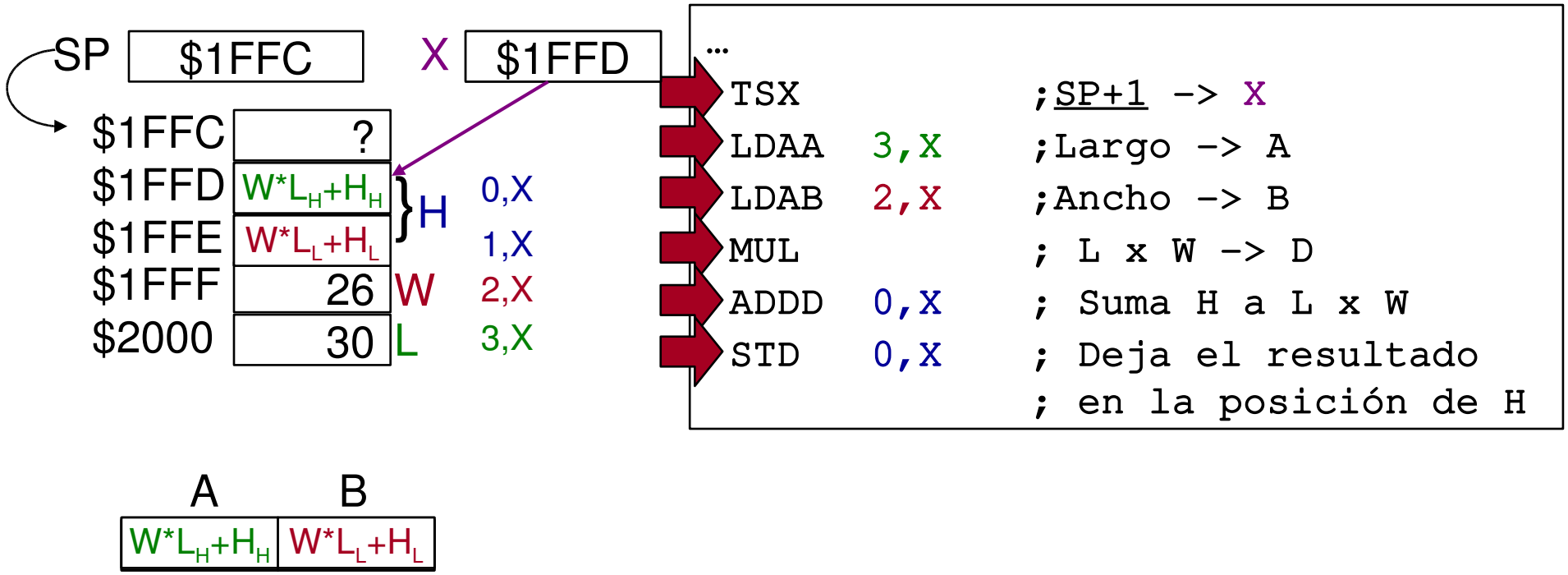
Ahora, accedes a L, W, y H desde tu código paa calcular L*W + H

Almacenas el resultado en la pila en la posición de H, pero no cambia nada más en la pila



# Uso de la pila con registros índice

Ahora, accede a L, W, y H desde tu código para calcular L*W + H
Almacena el resultado en la pila en la posición de H, pero no cambia nada más en la pila



# Subrutinas

- Las subrutinas: son bloques de código que pueden ser re-usados desde diferentes partes del programa
  - Útil para el código usado frecuentemente
  - Útil para dividir la ejecución del programa en partes fáciles de entender
- **Las librerías contienen subrutinas comunes**
  - Dejar a otro hacer la parte dura!
  - Así, solo tendremos que cortar y pegar las rutinas que queramos usar
  - En general, un *linker* inserta las rutinas de la librería en el código

# Llamada a Subrutina

- Las subrutinas son llamadas con instrucciones de tipo salto
  - `[<label>]  BSR  <rel>`
    `[<label>]  JSR  <opr>`
  - Ambas saltan al principio de la subrutina…
  - Pero, antes de hacerlo apila la *dirección de retorno en la pila*

Dir. siguiente instr.

| | |
|---|---|
| $2606 | ? |
| $2607 | 24 |
| $2608 | 2F |
| $2609 | 12 |
| $260A | 123 |
| $260B | 12 |
| $260C | 22 |

| | |
|---|---|
| SP | $2608 |
| (next) PC | $242F |

```
242d 8d 12          bsr   test
242f c6 0e          ldab  #14
                    ...
2441 86 0c  test    ldaa  #12
2443 39             rts
```

RTS – vuelve de la subrutina, desapila la dirección de retorno de la pila y regresa

# Uso de las subrutinas

- Reutilización de acumuladores/registros
  - Una subrutina puede cambiar los registros
- Paso de parámetros
  - Cómo le damos la información a la subrutina?
- Retorno de resultados
  - Cómo tomamos la información que devuelve
- Variables Locales
  - Qué pasa si la subrutina necesita espacio para ella

Clave: Todo lo puntos anteriores requieren el uso temporal de memoria

# Salvando Acumuladores/Registros

- Una subrutina necesitará usar varios acumuladores
  - No conoce cuales están siendo usados actualmente
    - Sobrescribir los registros será desastroso

- Alguien debe intervenir
  - Salvar los registros antes de usarlos
  - Reestablecerlos después de terminar la subruti

El Mejor dado que el programa principal puede no tener conocimiento del uso de la subrutina.

***Salva el que llama :***

Prog. Principal: Copia todos los registros en uso en la pila.

<call subrutina - retorno>

Prog. Principal: Restaura los registros desapilándolos de la pila.

***Salva el llamado (subrutina):***

Prog. Principal: <call subrutina>

Subrutina: Copia todos los registros que usará en la pila.

<cuerpo subrutina>

Subrutina: Restaura los registros desapilándolos de la pila.

<retorno subrutina>

# Paso de Parámetros/Valores de Retorno

- Modo simple: Usar registros
  - Poner los parámetro en registros predeterminados antes de la llamada
  - Poner los valores devueltos en registros predeterminados antes de volver

- Modo más general: Uso de la pila
  - Poner los parámetros (en orden) en la pila antes de la llamada, y quitarlos después de la vuelta
  - Poner un hueco para guardar los valores de reotrno en la pila antes de la llamada
    - La subrutina llena el hueco con los valores de retorno

# Subrutinas E/S

- El monitor BUFFALO proporciona algunos comandos
  - Entrada/salida de caracteres
  - Salida de cadenas
  - Salida numérica (hex)
  - Chequeo de espacios en blanco
  - Pasar a mayúsculas

- No proporciona
  - Entrada de cadenas
  - Entrada numérica
  - Entrada/salida decimal

# Rutinas de E/S de caracteres

`$FFB8` OUTA – Muestra el carácter ASCII del acumulador A

```
OUTA    EQU    $FFB8
        LDAA   #'H'
        JSR    OUTA
```

Muestra la letra 'H' en el terminal

`$FFCD` INCHAR – Introduce una carácter ASCII desde el terminal y lo almacena en el acumulador A

```
INCHAR  EQU    $FFCD
        JSR    INCHAR
* Caracter está ahora en ac. A
```

Introduce un solo carácter desde el terminal

Atención – las rutinas de E/S pueden cambiar los valores de sus parámetros.
Sálvalos antes de usarlas.

# Rutinas de salida de cadenas

`$FFC7` OUTSTRG – Salida cadena ASCII terminado EOT apuntado por X

```
OUTSTRG   EQU   $FFC7
          ORG   $2000
PROMPT    FCC   "Hello World."
          FCB   4

          LDX   #PROMPT
          JSR   OUTSTRG
```

terminador EOT

Saca la cadena "Hello World." al terminal. Saca un carácter CR/LF después de la cadena

`$FFCA` OUTSTRG0 – Igual que OUTSTRG, pero omitiendo CR/LF

`$FFC4` OUTCRLF – Saca un retorno de carro/línea siguiente (CR/LF) por el terminal

# Usando la E/S con Números

Los números se almacenan en la CPU en binario...

$2C = 0010 1100

Sin embargo, solo podemos mostrar o introducir valores ASCII

'A' = $41, 'a' = $61, '0' = $30, '1' = $31, '9' = $39, ...

Para sacar el valor $2C por la pantalla debemos sacar:

$32 $43 = '2C'

Por otra parte, si el usuario introduce '2C', se convierte en: $32 $43

Un byte numérico se representa mediante dos dígits y corresponde a dos caracteres ASCII (dos bytes).

# Otras subrutinas

`$FFB2` OUTLHLF – Convert the high-order half of Acc. A to an ASCII byte representing the number and output to terminal

`$FFB5` OUTRHLF – Convert the low-order half of Acc. A to an ASCII byte representing the number and output to terminal

`$FFBB` OUT1BYT – Convert the byte in memory pointed to by X to two ASCII bytes representing the number and output to terminal

`$FFBE` OUT1BSP – Convert the byte in memory pointed to by X to two ASCII bytes representing the number and output to terminal followed by a space character

`$FFC1` OUT2BSP – Convert the two consecutive bytes in memory pointed to by X to four ASCII bytes representing the number and output to terminal followed by a space character

# Otras rutinas

`$FFA0` UPCASE – Convert the ASCII character in Acc. A to uppercase

`$FFA3` WCHEK – Sets the Z bit if character in A is white space (comma, space or tab)

`$FFA6` DCHEK – Sets the Z bit if character in A is a delimiter (CR or white space)