Intel 8086

Departamento de Arquitectura[1]

[1]Instituto de Computación
Facultad de Ingeniería
Universidad de la República

Arquitectura de Computadoras, 2016

Temas

Lenguajes

- Lenguaje de alto nivel.
- **Lenguaje ensamblador**.
- Lenguaje máquina.

Por qué estudiar lenguajes ensamblador

- Clarifica la ejecución de instrucciones.
- Muestra como los datos se almacenan en memoria.
- Muestran como interactuar con el SO, el procesador y la E/S.
- Clarifican como los programas acceden a los dispositivos externos.
- Mejora el desempeño como programador de alto nivel.

Lenguaje ensamblador

- Lenguaje de programación alejado un paso del lenguaje máquina.
- Cada instrucción ensamblador se traduce a una única instrucción máquina.
- Depende del hardware.
- El programador debe conocer la arquitectura.

Desventajas

- Tiempo de desarrollo.
- Confiabilidad y seguridad
- Debugging y verificación.
- Mantenimiento.
- Portabilidad.
- Los compiladores han mejorado mucho en los últimos años.

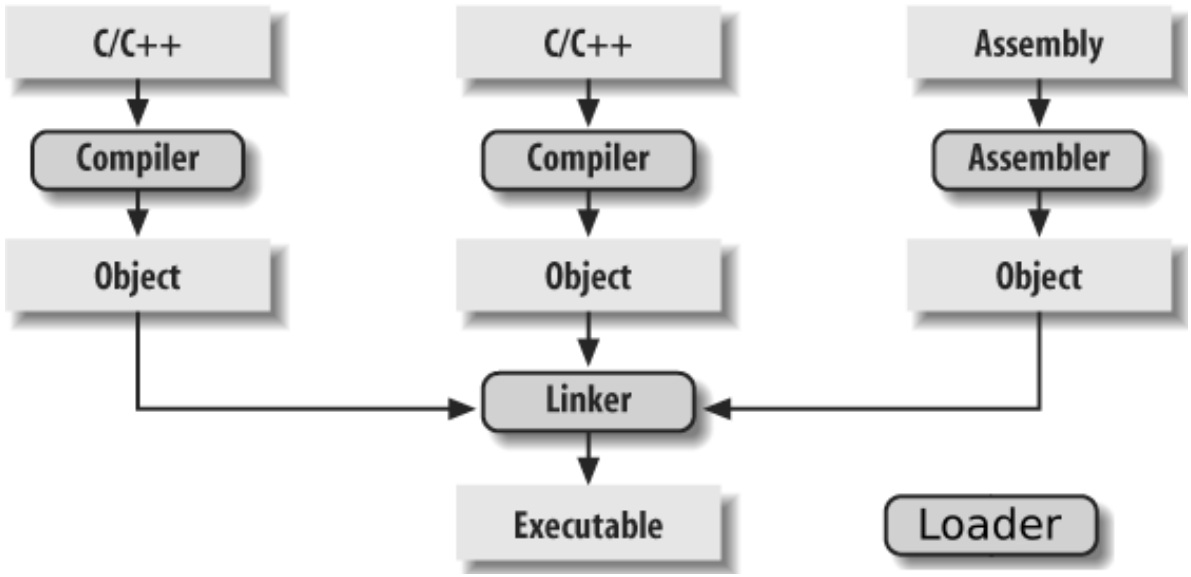
Ventajas

- Debugging y verificación.
- Desarrollo de compiladores.
- Sistemas embebidos.
- Drivers y código del sistema.
- Acceder a instrucciones no accesible desde los lenguajes de alto nivel.
- Optimizar el tamaño del código.
- Optimizar el código en tiempo de ejecución.

Assembler, Linker y Loader

- Assembler, herramienta (programa utilitario) que traduce lenguaje ensamblador a lenguaje máquina.
- Compiler, herramienta que convierte un programa de un lenguaje de programación a lenguaje máquina (código objeto).
- Linker, herramienta que combina uno o más códigos objetos en un único archivo que contiene código cargable (loadable) o código ejecutable.
- Loader, herramienta que copia un programa ejecutable en memoria para su ejecución.

Assembler, Linker y Loader



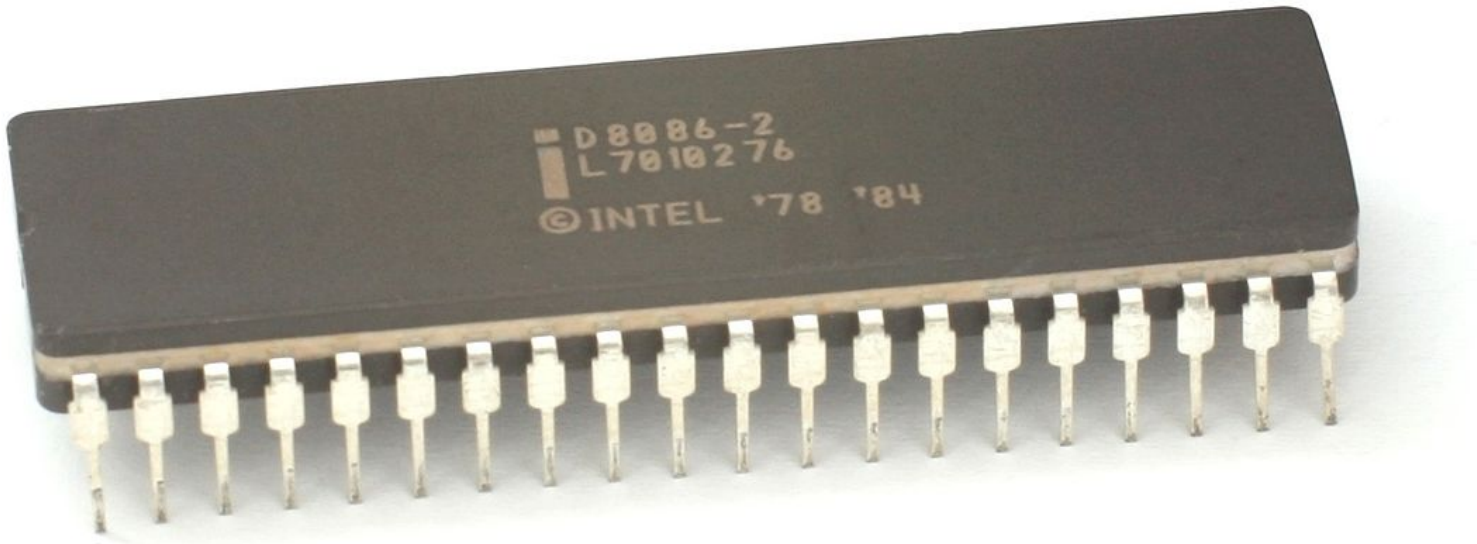
Historia

Desarrollado a fines de la década de 1970, es el pilar de la exitosisima familia x86 de Intel.
Una variante de este dispositivo, el 8088, fue el utilizado en el IBM PC.
Los procesadores modernos de Intel aún responden a las ideas básicas del 8086.

Historia

Intel se basó en la compatibilidad hacia atrás para dar soportes a *software* ya existente como estrategia comercial.
Eso impactó en el diseño de la arquitectura enormemente.

Intel 8086



Características

- Procesador de *16 bits*.
- Memoria *segmentada*.
- 12 registros visibles, con *personalidad*.
- *Little Endian*.
- Diseño CISC, incluyendo instrucciones de multiplicación y división enteras.
- Stack por Hardware.

Ancho de Palabra

Es un procesador de 16 bits, o sea, *su camino de datos* tiene un ancho de 16 bits. Esto incluye Registros, ALU y buses internos.
Diferentes implementaciones trabajan hacia el exterior con una cantidad diferente de bits (8088 trabaja con buses de 8).
Cuando digamos palabra, nos referiremos a 16 bits. La memoria en un sistema x86 se direcciona de a byte.
Nota: $2^{16} == 65536$

Intro Instrucciones

Brevemente, en Intel 8086, las instrucciones aceptan a lo sumo dos operandos.

Ejemplo

```
ADD AX, BX ; Realiza AX = AX + BX
MOV CX, AX ; Realiza CX = AX
NOT CX ; Realiza CX = !CX
```

Más adelante retomaremos el tema

Registros

12 registros visibles: CS, DS, SS, ES, AX, BX, CX, DX, SI, DI, SP y BP
Son registros visibles pero no de uso genérico.
Registros parcialmente visibles: FLAGS e IP

Descripción Registros

- CS, DS, SS, ES: Registros de segmento. Code, Data, Stack y Extra Segments.
- AX, BX, CX, DX: Registros de uso general. Acumulator, Base, Counter y Data.
- SP y BP: Registros Stack y Base Pointer. Implementación de stack.
- SI y DI: Registros Source y Destination Index.

Registros de 8 bits

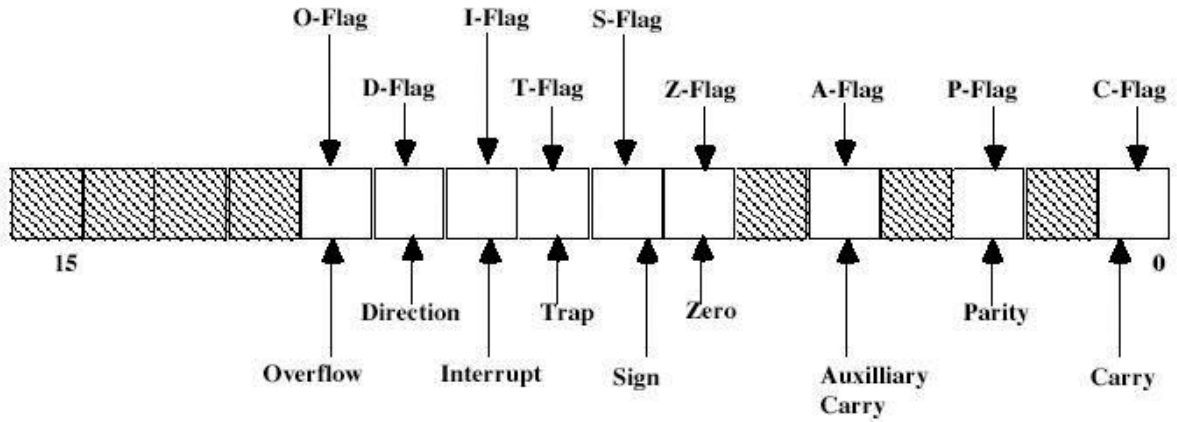
Los registros _X permiten acceder a su parte alta o baja independientemente.
Los registros High y Low. Ej: CH se corresponde con la parte alta de CX.

Ejemplo

```
mov CH, 1
mov CL, 0
; CX vale 0x0100
```

Flags

Existe un registro parcialmente visible en donde se almacenan bits de resultado y configuración.



La CPU lo actualiza de acuerdo al resultado de algunas instrucciones.

Flags

Esto permite conocer características de un resultado (*¿es cero?*) y del CPU (*¿están habilitadas las interrupciones?*).
Para implementar estructuras de control (if, while, for) utilizaremos saltos condicionales, en base a las flags.

Fundamento segmentación

Con direcciones de 16 bits, se pueden direccionar únicamente 64 KB.
Para sobreponerse a esto, se accede a la memoria de manera segmentada pero con direcciones de 20 bits que permiten acceder a 1MB.
No por esto deja de ser de 16 bits la arquitectura.

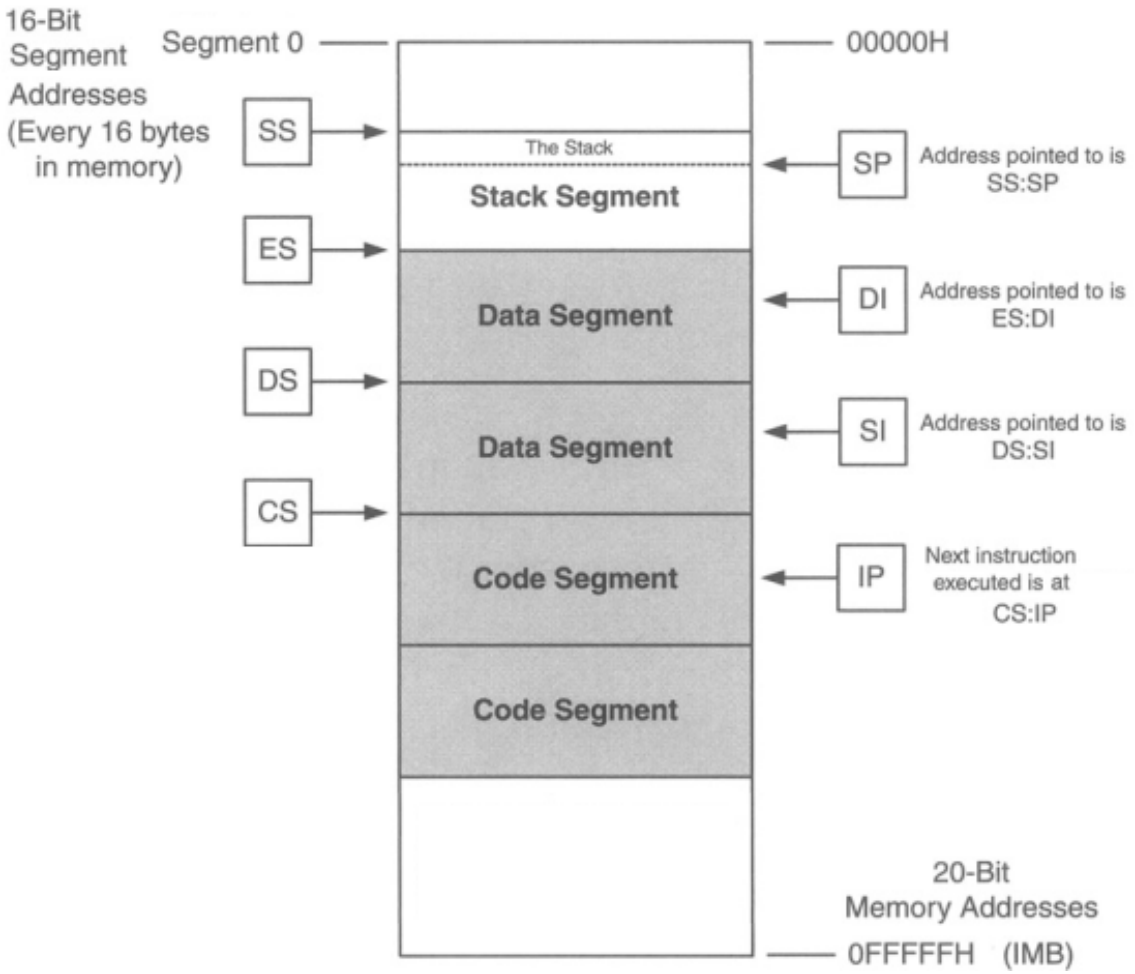
Funcionamiento

Se disponen 4 registros de segmento (CS, DS, SS y ES) que se utilizaran exclusivamente para el acceso a memoria.
Para acceder a memoria **siempre** utilizamos un registro de segmento, el cual se multiplicará por 16 y se sumará a la dirección indicada.
Dirección final [20] = Registro de segmento * 2^4 + Dirección Indicada

Segmentos

Esto de facto genera segmentos en la memoria.
Si CS = 0x3100, su segmento abarca desde 0x31000 hasta 0x40FFF.
Esto es una *construcción* y no un elemento visible en los datos de la memoria. Puede existir solapamiento, incluso completo.

Vista


16-Bit Segment Addresses (Every 16 bytes in memory)
Segment 0
00000H
SS
The Stack
Stack Segment
SP
Address pointed to is SS:SP
ES
Data Segment
DI
Address pointed to is ES:DI
DS
Data Segment
SI
Address pointed to is DS:SI
CS
Code Segment
IP
Next instruction executed is at CS:IP
Code Segment
20-Bit Memory Addresses
0FFFFFH (IMB)

Ejercicio Segmentos

Si quisiera guardar un dato en la dirección física 0x31002, dados los siguientes valores de registro de segmento. ¿Cuál debería ser la dirección segmentada?

Ejemplo

```
CS=0x3200
DS=0x5200
ES=0x3000
SS=0x0100
```

Direcciones segmentadas

Desplazamiento

{ Bx | Bp } [+ { Si | Di }] [+ desplazamiento] |

{ Si | Di } [+ desplazamiento] | desplazamiento

Direcciones segmentadas

Segmento

	CS	SS	DS	ES
IP	Si			
SP		si		
BP	Prefijo	por defecto	prefijo	prefijo
BX	Prefijo	prefijo	por defecto	prefijo
SI	Prefijo	prefijo	por defecto	prefijo
DI	Prefijo	prefijo	por defecto	por defecto (cadenas)

Endianess

Una palabra de memoria es un byte, una palabra del CPU son 2. ¿Cómo se dispone cuando se guarda en memoria?

- El *endianess* como se almacena en memoria una palabra de la arquitectura de tamaño mayor.
- *Little endian* indica que en la posición menor, va el byte menos significativo.
- *Big endian* indica que en la posición menor, va el byte más significativo. (Internet)

Ejercicio Segmentos

Si ejecutamos el siguiente código:

Ejemplo

```
mov AX, 0x1234
mov ES:[0x1000], AX
```

¿Qué hay en ES:[0x1001]?

Introducción

La arquitectura Intel provee como herramienta para el programador un stack implementado por hardware sobre memoria.
Este stack es un elemento central en la arquitectura, principalmente en el pasaje de parámetros y la invocación a funciones.

Funcionamiento

El stack se encuentra implementado en hardware como una pila circular en el Stack Segmenta (SS) mediante dos punteros:

- Stack Pointer, es el puntero a la última posición ocupada del Stack.
- Base Pointer, para direccionar en el stack.

Las posiciones del stack siempre ocupan dos bytes, independientemente del tamaño del dato que se inserta.

Instrucciones

Con la instrucción PUSH agregaremos datos al stack.

Ejemplo

```
PUSH AX; Inserta el contenido del registro AX
       ; en el tope del stack.
```

Instrucciones

Con la instrucción POP los retiraremos hacia otra posición. El destino puede ser un registro u otra posición de memoria.

Ejemplo

```
POP CX; Carga en CX el contenido en el tope del stack.
POP ES:[BX]; Carga en la posición de memoria ES*16 +BX
           ; el contenido del tope del stack.
```

Ejemplo

1

Instrucción: MOV CX, -1

Resultado:

	0x00000	
	...	
	0x3f002	...
	0x3f003	
	0x3f004	...
	0x3f005	
SP→	0x3f006	0xAAC3
	0x3f007	
	...	
	0xFFFFF	

CX	**0xFFFF**
DX	0x0018

2

Instrucción: PUSH DX

Resultado:

	0x00000	
	...	
	0x3f002	...
	0x3f003	
SP→	0x3f004	**0x0018**
	0x3f005	
	0x3f006	0xAAC3
	0x3f007	
	...	
	0xFFFFF	

CX	0xFFFF
DX	0x0018

3

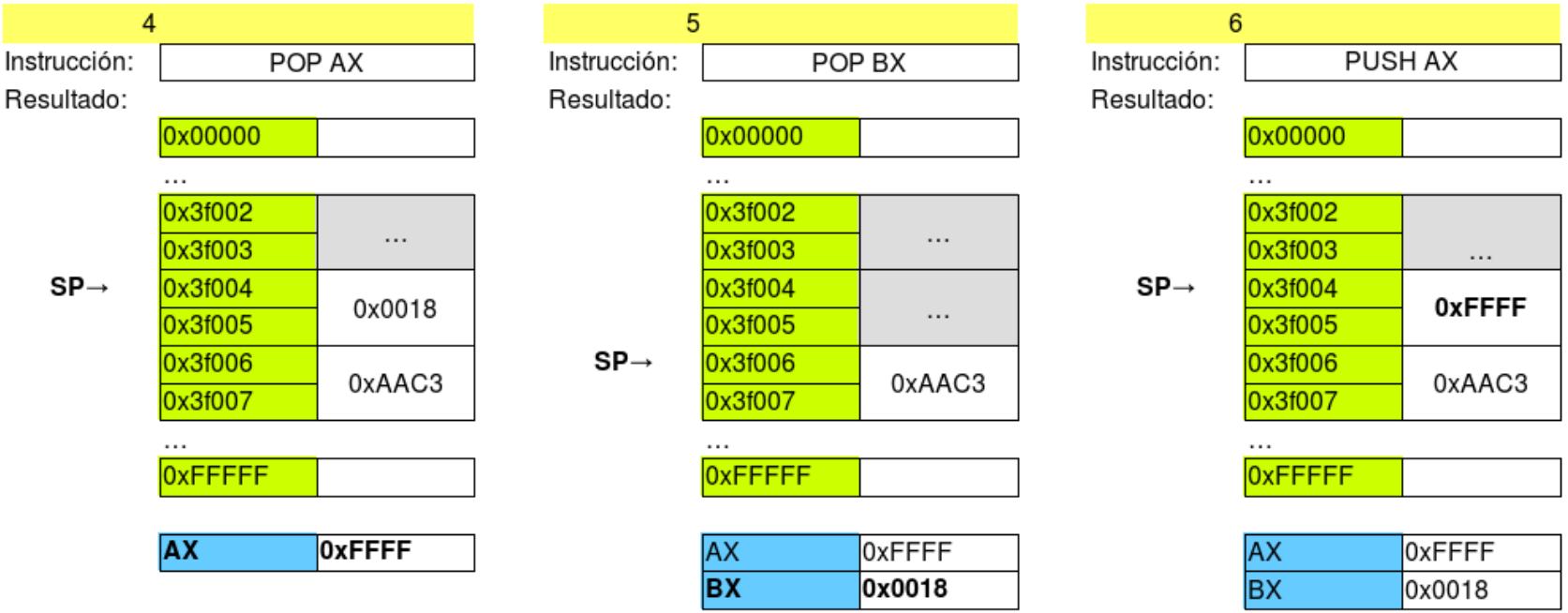
Instrucción: PUSH CX

Resultado:

	0x00000	
	...	
SP→	0x3f002	**0xFFFF**
	0x3f003	
	0x3f004	0x0018
	0x3f005	
	0x3f006	0xAAC3
	0x3f007	
	...	
	0xFFFFF	

CX	0xFFFF
DX	0x0018

Ejemplo


4
Instrucción: POP AX
Resultado:
0x00000
...
SP→
0x3f002
0x3f003
0x3f004
0x3f005
0x3f006
0x3f007
...
0x0018
0xAAC3
0xFFFFF
AX 0xFFFF
5
Instrucción: POP BX
Resultado:
0x00000
...
SP→
0x3f002
0x3f003
0x3f004
0x3f005
0x3f006
0x3f007
...
0xAAC3
0xFFFFF
AX 0xFFFF
BX 0x0018
6
Instrucción: PUSH AX
Resultado:
0x00000
...
SP→
0x3f002
0x3f003
0x3f004
0x3f005
0x3f006
0x3f007
...
0xFFFF
0xAAC3
0xFFFFF
AX 0xFFFF
BX 0x0018

Set de Instrucciones

Intel 8086 es una arquitectura tipo CISC, set de instrucciones amplio. Ej: CMPS, MUL
Instrucciones con accesos a memoria implícitos, duración variable y de largo variable.
Provee instrucciones:

- Aritméticas (ADD, SUB, INC)
- Lógicas (AND, OR, XOR)
- Control (JMP, JZ)
- Transferencia de datos (MOV, IN, OUT, PUSH, POP)
- Configuración (STI, CLI)

Características básicas

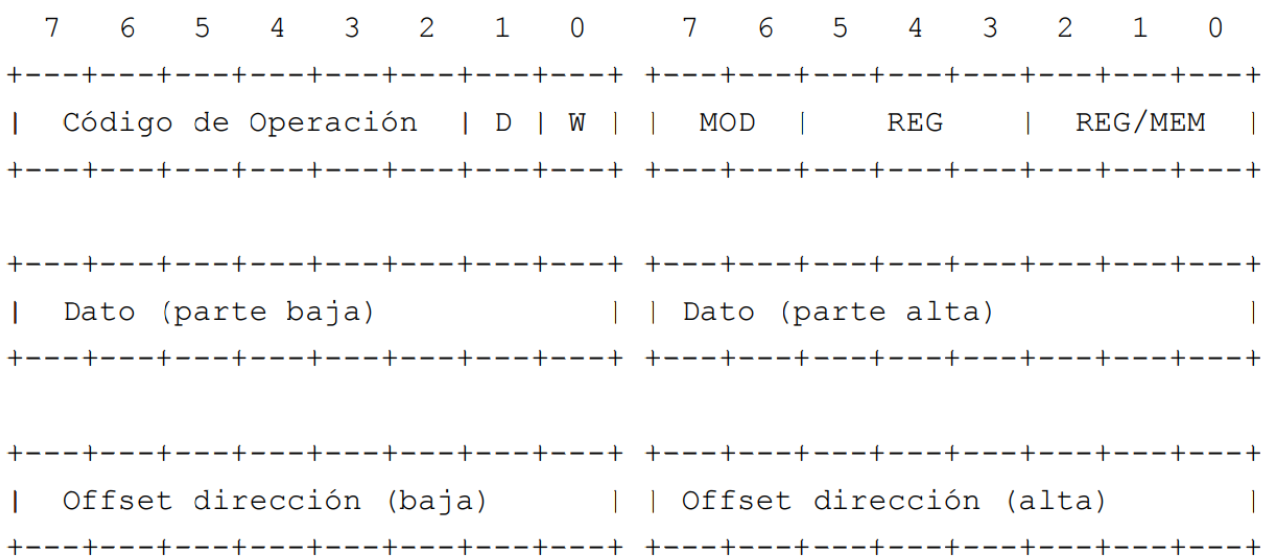
Provee instrucciones:

- Codificadas en largo variable (1 a 6 bytes).
- Aceptan (a lo sumo) 2 operandos.
- Máximo 1 operando puede ser memoria.
- Su formato nemotécnico es INSTR op1, op2

Modos de direccionamiento

- Inmediato - mov AX, **2**
- Directo a Registro - mov **AX**, 2
- Directo a Memoria - mov AX, **ES:[0x200]**
- Indirecto por registro - mov AX, **ES:[BX]**
- Indizado - mov AX, **ES:[BX+SI]**

Codificación



D - sentido operacion (0 desde reg)
W - op usa palabras (1) o bytes (0)
REG/MEM - 2do reg o regs involucrados en direccionamiento indirecto

MOD - modo de direccionamiento
REG - indica registro

Introducción

Veremos una versión básica de *assembler* Intel 8086.
Un programa escrito en *assembler* 8086 se compone de:

- Comentarios
- Constantes
- Instrucciones
- Etiquetas
- Directivas

Con esto, el *ensamblador* generará el bitcode necesario.

Comentarios

Todo lo que se encuentre luego de un carácter ; se toma como un comentario.

Ejemplo

```
ADD AX, BX ; Esto es un comentario
; Esto también es un comentario
; MOV CX, AX esto no es una instrucción
```

Constantes

Utilizado para indicar datos constantes y datos iniciales.

- Binarios, octales y hexadecimales
  - 001101b
  - 664o
  - FE53h o 0xFE53
- Enteros
  - 23
  - -120
- Strings
  - ‘‘c’’
  - ‘‘Hola mundo!’’

Instrucciones

Las instrucciones se escribirán una a continuación de la otra, el ensamblador las traducirá a bytes y concatenará en memoria de la misma forma.
Cada instrucción tiene un largo variable, por lo que no es fácil saber cuanto ocupa un programa o una instrucción en memoria.
Lista fundamental en la cartilla del curso (disponible en exámenes y pruebas).

Instrucciones
ADD en la notas

ADD

Formato: ADD op_1, op_2

Tipo Args: (r,m) (r,m,i)

Lógica: $op_1 \leftarrow op_1 + op_2$

Descripción: Suma los dos operandos y almacena el resultado en op_1, por lo tanto ambos deben tener el mismo tipo

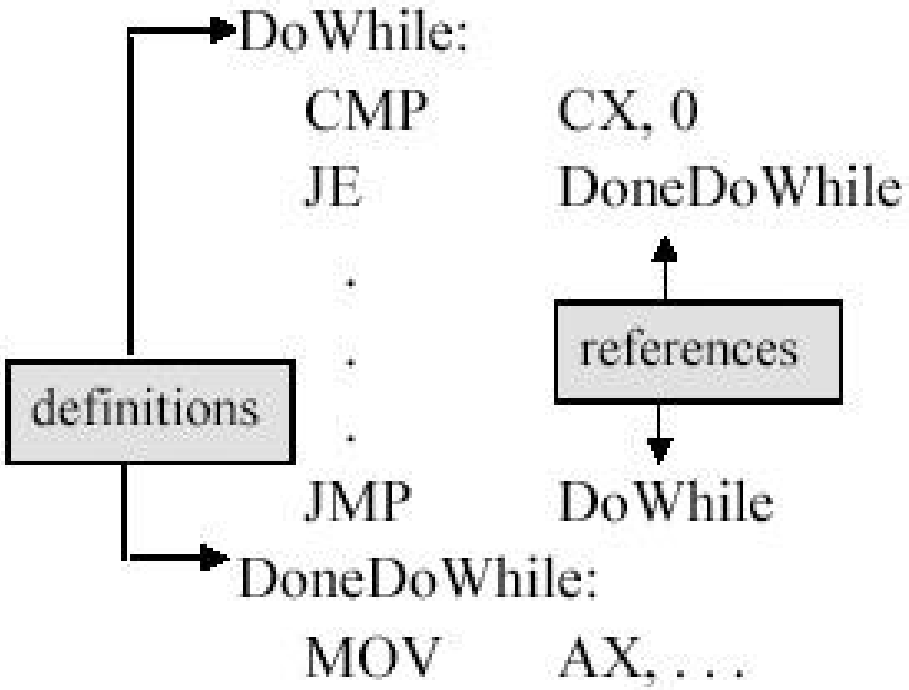
Banderas:

OF	DF	IF	TF	SF	ZF	AF	PF	CF
X	-	-	-	X	X	X	X	X

Etiquetas

Utilizado para tener referencias accesibles a diferentes posiciones de memoria de datos o código.
Referencia la primera posición del dato o instrucción a continuación.
Pueden estar formadas por letras y números.

Etiquetas


DoWhile:
CMP CX, 0
JE DoneDoWhile
.
.
.
JMP DoWhile
DoneDoWhile:
MOV AX, . . .
definitions
references

Directivas

Utilizado para indicar datos constantes y datos iniciales.

- EQU
- DB, DW, DDW, DUP
- PROC y ENDP

EQU

EQU se utiliza para definir constantes y atajos para programar.

Ejemplo

```
max_posiciones EQU 8
tam_arreglo EQU max_posiciones * 4
```

DB, DW, DDW

DB, DW y DDW se utilizan para declarar datos inicializados. *Byte*, *Word* y *Double Word* respectivamente.

Ejemplo

```
db 0x93 ; un byte con valor 0x93
db 0x94,0x95,0x96,0x97 ; cuatro bytes consecutivos
db 'q' ; un byte con el valor ascii de q
dw 0x5A0 ; 0xA0 0x05
dw 'abc' ; 4 bytes con 0x61 0x62 0x63 0x00
ddw 0x12345678 ; ocho bytes 0x78 0x56 0x34 0x12
```

DUP

DUP permite repetir declaraciones.

Ejemplo

```
db 5 dup(0x93) ; 5 bytes con valor 0x93
dw 3 dup(0x5A0) ; 6 bytes 0xA0 0x05 0xA0 0x05 0xA0 0x05
```

PROC y ENDP

Marcan el principio y fin de procedimientos.
Para definir un procedimiento

Ejemplo

```
nombreProc PROC
  ADD AX, ...
  ...
nombreProc ENDP
```

Ejemplo 1

Intercambiar el contenido de AX y BX.

```
intercambiar PROC
  push AX
  mov AX, BX
  pop BX
  ret
intercambiar ENDP
```

Ejemplo 2

Calcular la suma de una posición en memoria con AX y guardar en otra.

```
; En DS
  operandoMem dw ...
  resultadoMem dw 0

; En CS
sumaMem PROC
  ; Cargo dir en BX
  mov BX, operandoMem
  push AX
  add AX, [BX]
  mov BX, resultadoMem
  mov [BX], AX
  pop AX
  ret
sumaMem ENDP
```

Introducción

Para implementar algoritmos, precisamos controlar el flujo de nuestro programa.
La *siguiente* instrucción a ejecutar por el CPU puede que no sea la que está *a continuación*.

Introducción

Para esto, contamos en assembler con las instrucciones de salto. Mediante su combinación y estructurando el código adecuadamente, lograremos compilar estructuras de control de alto nivel.

CMP

La instrucción `CMP` es útil (aunque no indispensable) para compilar estructuras de control.
Nos permite *comparar* dos operandos, sin necesidad de almacenar el resultado aunque si modificando las flags.
Realiza una resta entre los operandos y descarta el resultado.

Jumps

Las instrucciones de salto (*jumps*) son imprescindibles para compilar estructuras de control.
Típicamente:

- Si estamos frente a una bifurcación, tomaremos un salto hacia delante en el código.
- Si estamos frente al final de un bucle, tomaremos un salto hacia atrás en el código.

Jumps

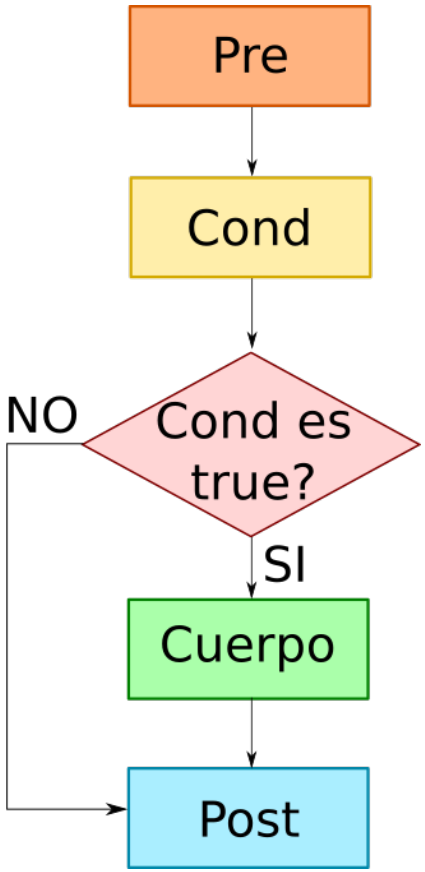
Las instrucciones de salto (*jumps*) pueden ser condicionales o incondicionales:

- El salto incondicional (`JMP`) se toma **siempre**.
- Los saltos condicionales se toman de acuerdo al valor de una condición, las cuales se calculan de acuerdo al valor de las flags.
  - `JZ` - Jump If Zero
  - `JNZ` - Jump If Not Zero
  - `JA` - Jump If Above (Sin signo)
  - `JG` - Jump If Great (Con signo)
  - `JNLE` - Jump If Not Less or Equal (Con signo)
  - ...

IF

¿Cómo es una estructura IF?

```
pre_if;
if( condicion_if ){
   cuerpo_if;
}
post_if;
```



Alto Nivel

En alto nivel, únicamente si la condición es verdadera ejecutaremos un cierto bloque de código.

```
resultadoBusqueda = false;
if( totalSuma < 0 ){
   esNegativo = true;
}
totalSuma = 0;
```

Ensamblador

En ensamblador, si la condición no es cierta, nos saltearemos un cierto bloque de código.

```
contador--;
if( contador < 0 ){
   acumulador++;
}
contador = 0;
```

```
   dec CX
   cmp CX, 0
   jnl postIf
   inc AX
postIf:
   mov CX, 0
```

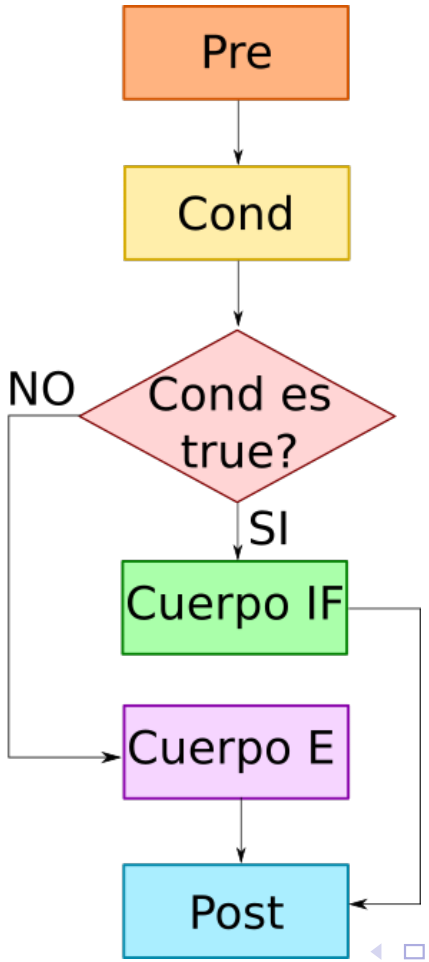
Ejemplo

Ejercicio:
Implementar una función que reciba en AX un número y devuelva en DX 1 si es múltiplo de 4 o 0 en caso contrario

IF-ELSE

¿Cómo es una estructura IF-ELSE?

```
pre_if;
if( condicion_if ){
   cuerpo_if;
} else {
   cuerpo_else;
}
post_if;
```



Alto Nivel

En alto nivel, si una condición es verdadera se ejecutará un cierto bloque de código. Si no, se ejecutará otro bloque de código.

```
resultadoBusqueda = false;
if( totalSuma < 0 ){
   esNegativo = true;
}
else {
   resultadoBusqueda = true;
}
totalSuma = 0;
```

Ensamblador

```
contador = contador-3;
if( contador % 2 ){
   acumulador++;
}
else{
   acumulador--;
}
contador = 0;
```

```
   sub CX, 3
   mov SI, CX
   and SI, 0x01
   jz else
   inc AX
   jmp postIf
else:
   dec AX
postIf:
   mov CX, 0
```

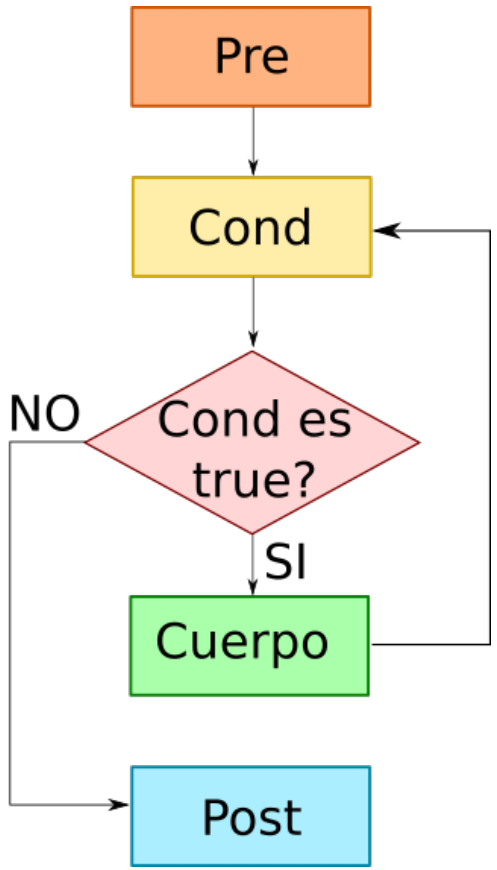
Ejemplo

Ejercicio:
Implementar una función que reciba en AX una letra y devuelva en DX 1 si es anterior a la 'Z'.

WHILE

¿Cómo es una estructura WHILE?

```
pre_while;
while( condicion_wh ){
   cuerpo_while;
}
post_while;
```



Alto Nivel

En alto nivel, si una condición es verdadera se ejecutará un cierto bloque de código. Al terminar, volverá a evaluarse la condición. Siempre que sea falsa, no se ejecutará el bloque de código y se seguirá a continuación.

```
largoLista = 0;
while( lista != null ){
   lista = lista->siguiente;
   largoLista++;
}
total += largoLista;
```

Ensamblador

```
contador = 0;
while (dato % 2 == 0){
   dato >> 1;
   contador++;
}
acumulador += contador;
```

```
   mov CX, 0
condWhile:
   mov SI, DX
   and SI, 0x01
   jnz finWhile
   shr DX, 1
   inc CX
   jmp condWhile
finWhile:
   add AX, CX
```

Anidado

Es posible anidar estructuras de control.

```
contador = 0;
while (contador < 1000){
   if (contador % 4 == 0){
      acumulador += contador;
   }
   contador++;
}
base += acumulador;
```

```
   mov CX, 0
condWhile:
   cmp CX, 1000
   jnl finWhile
   and CX, 0x04
   jnz finIf
   add AX, CX
finIf:
   inc CX
   jmp condWhile
finWhile:
   add BX, AX
```

Claridad del código

Es importante mantener el código claro mientras se realiza en papel.
Es fácil perderse y confundir saltos, olvidarse etiquetas, etc...
Más aún en un examen.

Evitar condiciones complejas

Al escribir código de alto nivel, hay que tratar de estar atento a no utilizar condiciones extremadamente complejas o de muchos términos en las estructuras de control.
Es fácil confundirse y perderse. **Más aún en un examen**.

Evitar anidaciones excesivas

Al escribir código de alto nivel, hay que tratar de estar atento a no utilizar estructuras de control anidadas en exceso.
Es fácil perderse y confundir saltos, olvidarse etiquetas, etc...
Más aún en un examen.

Preguntas