DCC

DEPARTAMENTO DE CIENCIA
DE LA COMPUTACIÓN

IIC2343

Arquitectura de Computadores

Clase 8 - Arquitectura x86

Profesor: Germán Leandro Contreras Sagredo

Objetivos de la clase

- Conocer la arquitectura x86 a nivel general y su evolución.
- Entender el funcionamiento de una convención de llamadas en la arquitectura x86.

Hasta ahora…

- Ya conocemos más sobre la arquitectura de nuestro computador y cómo se categoriza.

- Por otra parte, sabemos distinguir sobre lo que es la microarquitectura de lo que son los *sets* de instrucciones.

En esta clase aprenderemos sobre la ISA x86.

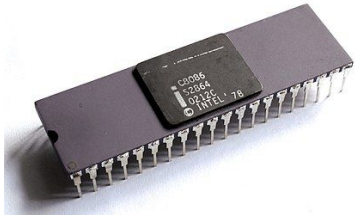
Arquitectura x86

- ISA de tipo **CISC** desarrollada inicialmente por Intel.
- De las más utilizadas. Antes, se licenciaba tanto la ISA como la microarquitectura. Hoy en día, solo se licencia la ISA y el resto de fabricantes se encargan de que su microarquitectura la soporte.
- Presenta **diferencias clave** con la arquitectura del computador básico.

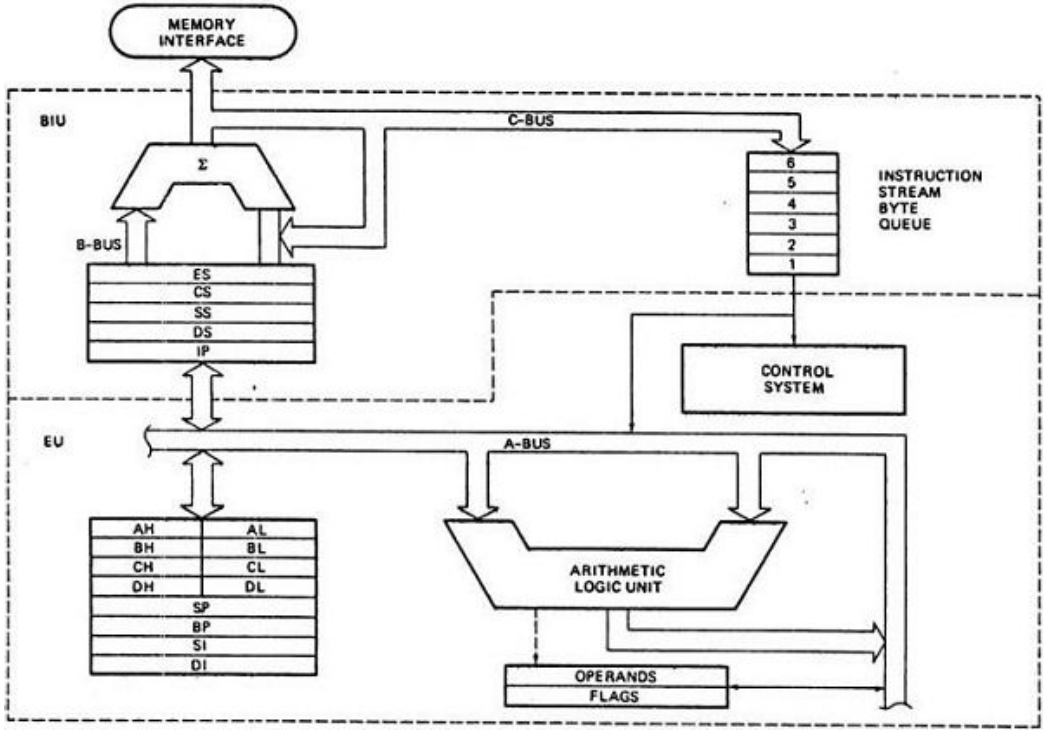
Arquitectura x86 - i8086

Primer microprocesador de Intel de 16 bits que presenta la arquitectura x86 (1978).

¿Qué diferencias podemos observar respecto al computador básico?



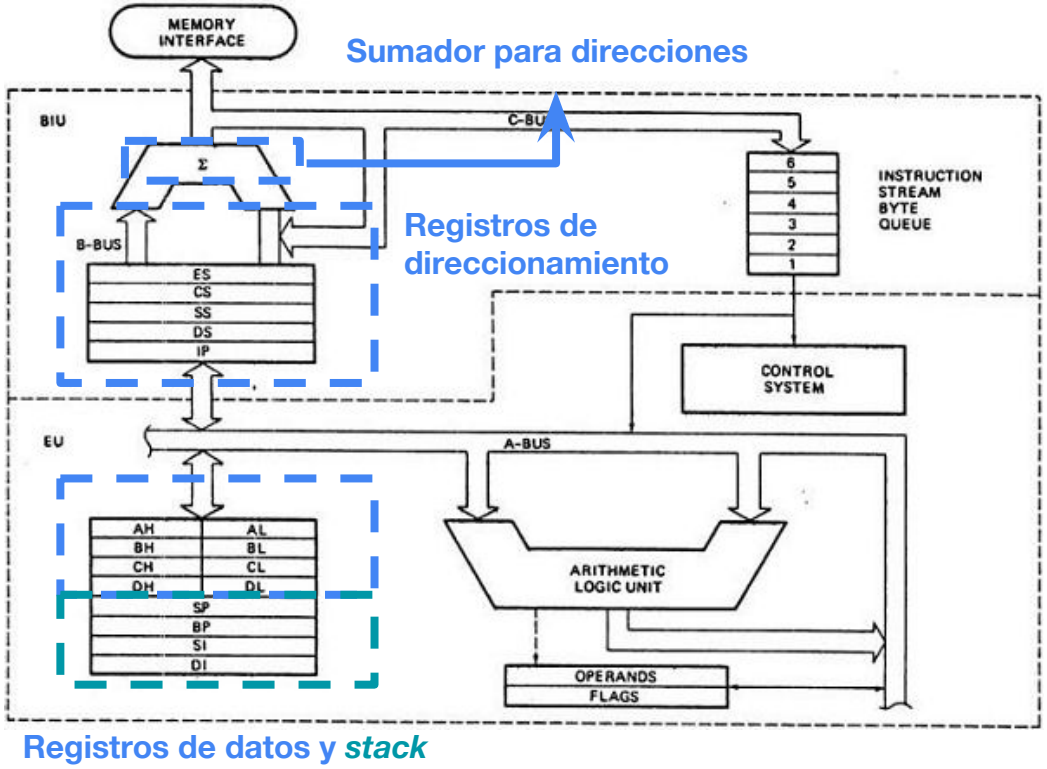
Chip i8086 y su diagrama.



Arquitectura x86 - i8086

Registros

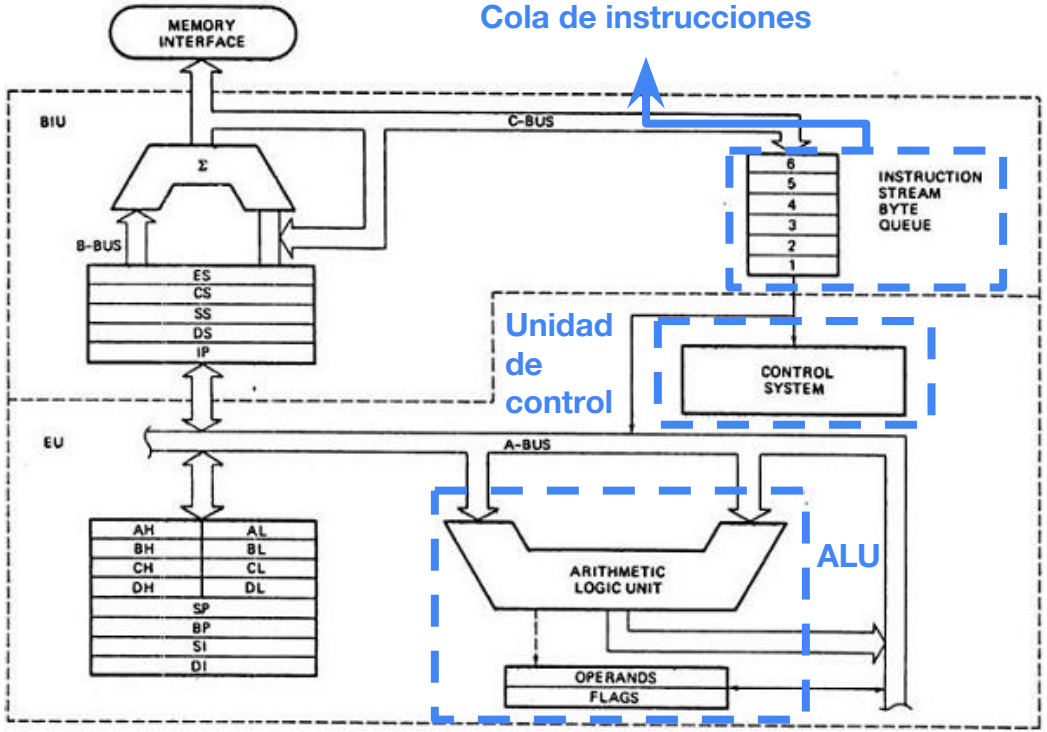
- Más registros; segmentación (`AH | AL`).
- Registros y sumador que habilitan más modos de direccionamiento.
- Más registros para el manejo de *stack*.



Arquitectura x86 - i8086

Procesamiento

- ALU con operandos y *flags*.
- Unidad de control con “cola” de instrucciones por ejecutar.

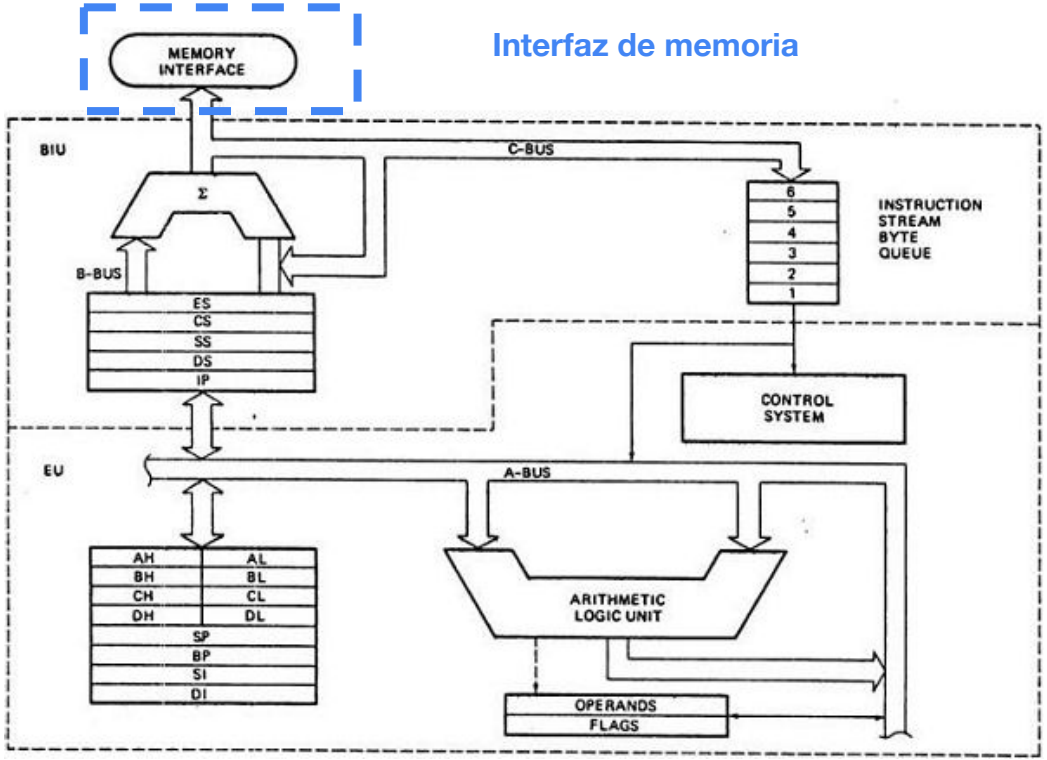


Arquitectura x86 - i8086

Memoria

- Única “interfaz de memoria”.
- Arquitectura Von Neumann.

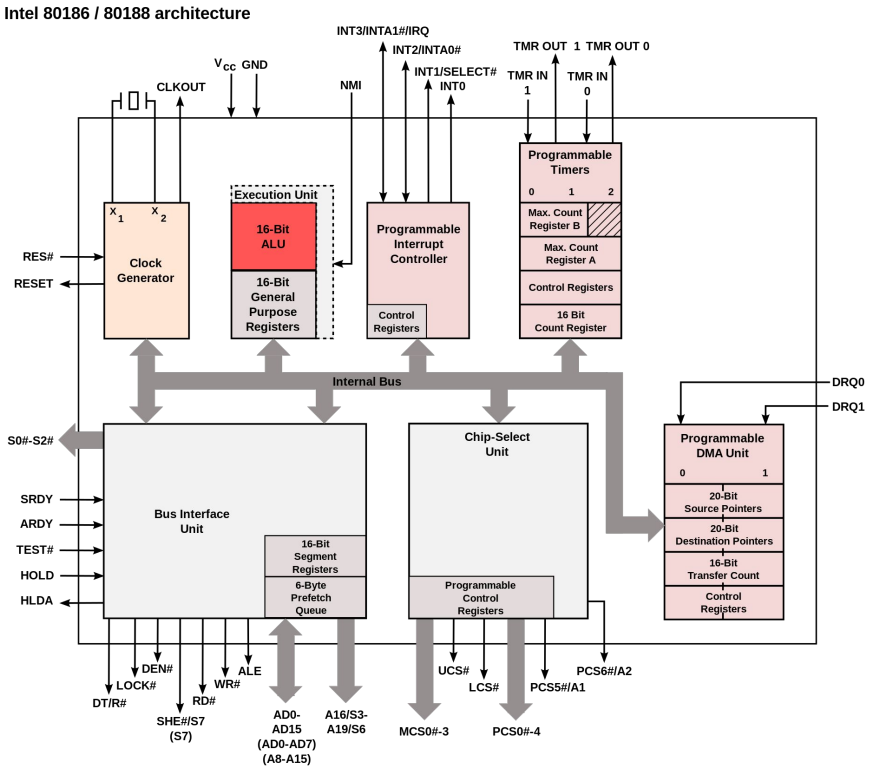
¿Cómo fue evolucionando esta arquitectura?



Arquitectura x86 - i80186

- Versión optimizada de i8086 (1982).
- Usa transistores más pequeños (chip más compacto).

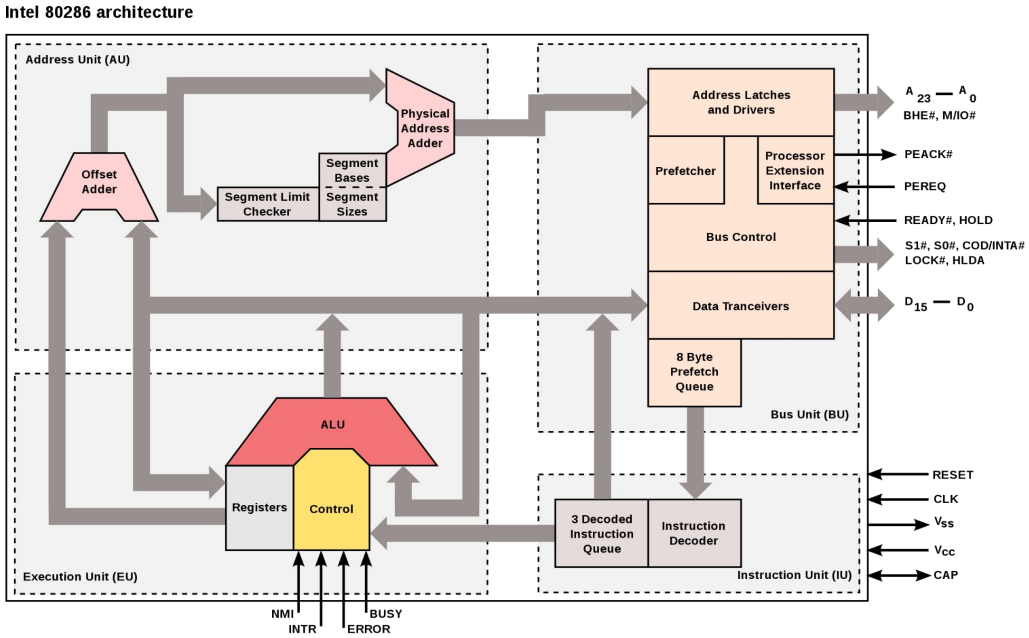
Chip i80186 y su diagrama.



Arquitectura x86 - i80286

- Chip contemporáneo al i80186 (1982).
- Incluye soporte de multiprogramación (próximas clases).

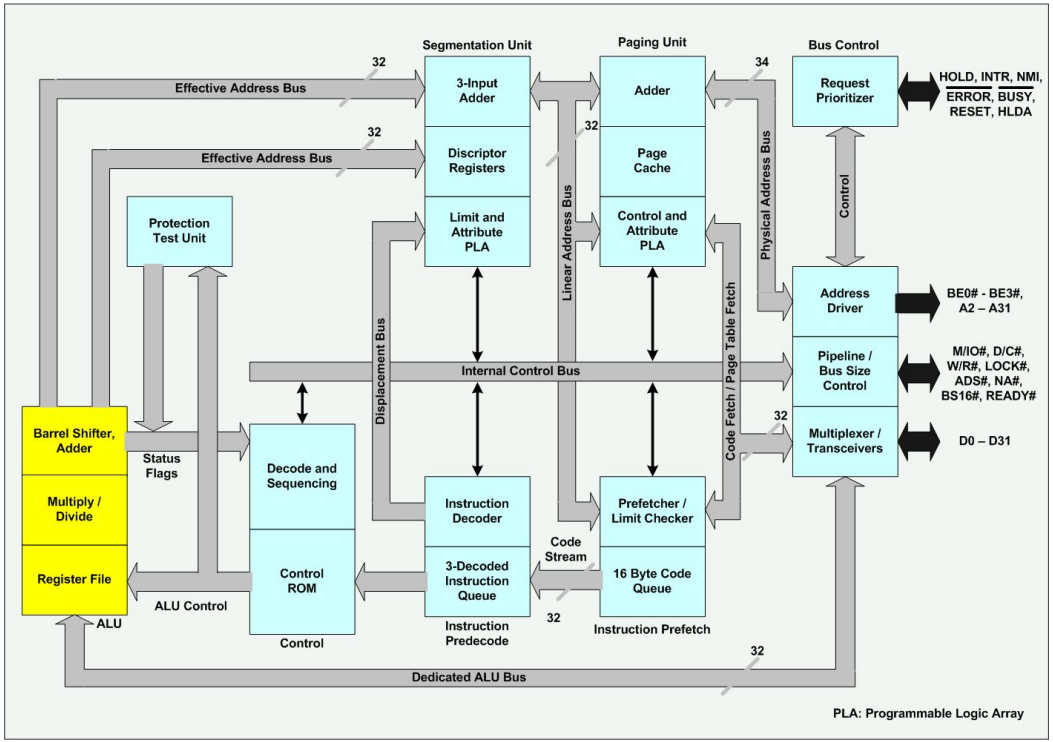
Chip i80286 y su diagrama.



Arquitectura x86 - i386 (i80386)

- Chip sucesor que funciona con 32 bits (1985).
- Implementa la ISA IA-32 (x86 de 32 bits).

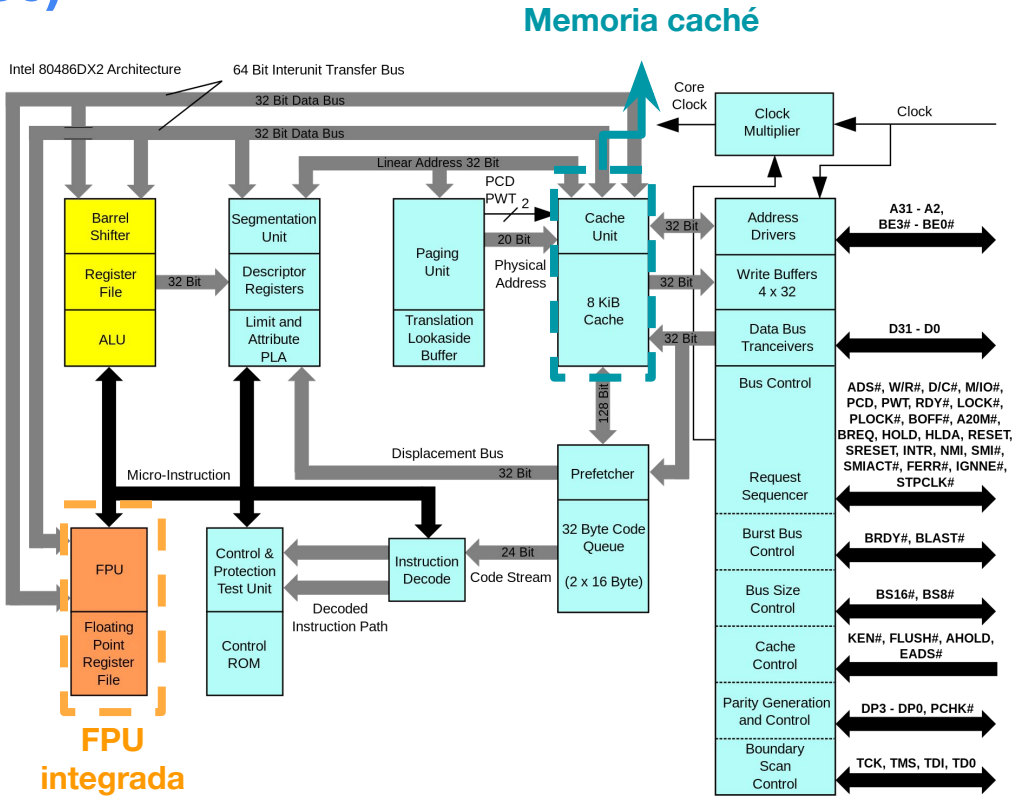
Chip i386 y su diagrama.



Arquitectura x86 - i486 (i80486)

- Chip sucesor que **posee dentro** una FPU y memoria caché (1989).
- Posee más de 1 millón de transistores.

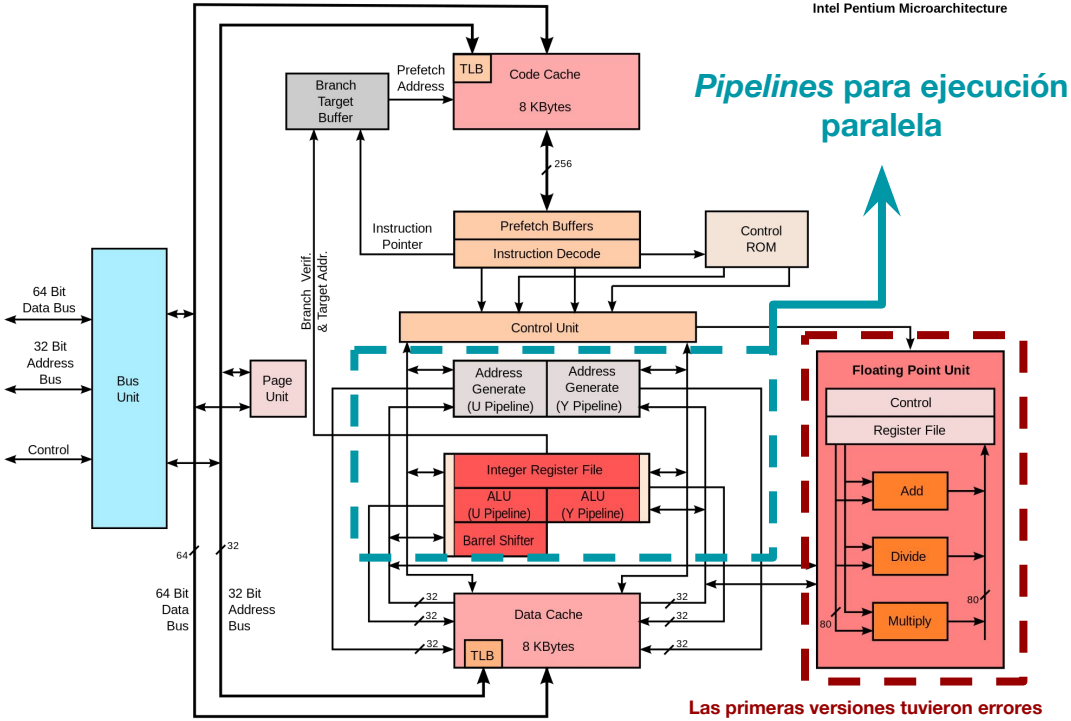
Chip i486 y su diagrama.



Arquitectura x86 - Pentium, P5, i586 (i80586)

- Chip con arquitectura superescalar, esto es, que puede ejecutar más de una instrucción en un ciclo (1993).

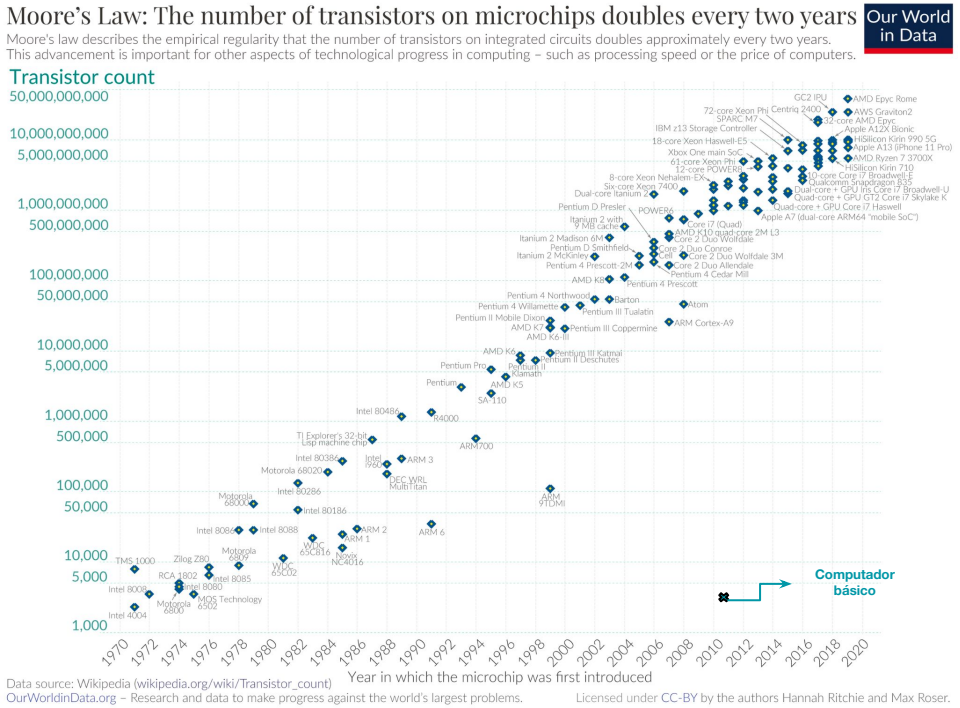
Chip Pentium y su diagrama.



Las primeras versiones tuvieron errores importantes en el cálculo de números *float*

Arquitectura x86 - Ley de Moore

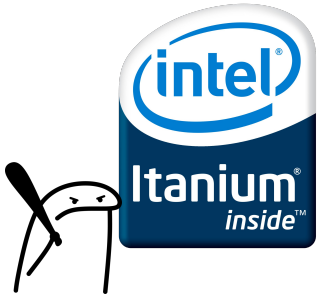
- Ley **empírica** basada en observaciones sobre la industria de Gordon Moore (cofundador de Intel).
- "El número de transistores de un circuito integrado se duplica cada dos años". Esto por la reducción del tamaño de transistores vía avances tecnológicos.



Eje X logarítmico, eje Y lineal → Crecimiento exponencial

Arquitectura x86

- Se considera una arquitectura **Von Neumann + CISC**.
- Revisaremos la versión de 16 bits (utilizada hasta el chip i80286).
- En la actualidad, no obstante, se utiliza la versión x86-64 (bits), **propuesta y popularizada por AMD**.



Intel trató de lanzar su propia ISA de 64 bits (IA-64) para la línea de procesadores Itanium con el objetivo de reemplazar la arquitectura x86. No obstante, presentó rendimientos muy bajos en comparación a su competencia. AMD, por su parte, propuso x64, que corresponde a IA-32 con extensiones para 64 bits que puede, además, emular programas de 16 y 32 bits en computadores de 64 bits (*compatibility mode*).

Arquitectura x86 - Microarquitectura en detalle

- 4 registros de propósito general de 16 bits (**`AX`**, **`BX`**, **`CX`**, **`DX`**).
- Registros divisibles en sectores altos y bajos (**`AX = AH | AL`**).
  Esto permite **compatibilidad con programas de 8 bits**.
- `BX` se utiliza para direccionamiento indirecto (con registro base).
- 2 registros de 16 bits **sin división** para uso general y direccionamiento indirecto (**`SI`**, **`DI`**), usados como **registro índice**.

Arquitectura x86 - Microarquitectura en detalle

- Registros adicionales: **`IP`** (*Instruction Pointer*), **`SP`** (*Stack Pointer*), **`BP`** (*Base Pointer*).
- Solo una ALU como unidad de ejecución (FPU desde i486).
- 6 *condition codes*: Z, N, O, V, **P**(*arity*), **A**(*uxiliary carry* - para BCD).
- Direcciones de memoria: 16 bits; palabras de memoria: 8 bits.
- *Stack* en memoria. Registro `SP` apunta al **último elemento ingresado al *stack***.

Arquitectura x86 - ISA en detalle

La ISA de la arquitectura x86 es más compleja que la del computador básico:

- Posee instrucciones de transferencia, aritméticas, lógicas, saltos y subrutinas.
- Acepta tipos de datos nativos de 8 y 16 bits (con y sin signo).
- Posee múltiples tipos de direccionamiento.

Arquitectura x86 - ISA en detalle

Tipos de direccionamiento en x86

- Directo (uso de literal).
- Indirecto (registro `BP`).
- Indirecto con *offset* (tamaño de dato almacenado).
- Indirecto con índice (arreglos; registros `SI` y `DI`).
- Indirecto con índice y *offset*.

Arquitectura x86 - ISA en detalle

Tipos de direccionamiento en x86 - Ejemplos

- `MOV AX,var` → `AL = Mem[var]; AH = Mem[var + 1]`
- `MOV AX,[BX]` → `AL = Mem[BX]; AH = Mem[BX + 1]`
- `MOV AL,[BX]` → `AL = Mem[BX]`
- `MOV AX,[BL]` → ¡Inválido! Direcciones deben ser de 16 bits.

¿Qué tipo de *endianness* presenta?

R: *Little endian.*

Arquitectura x86 - ISA en detalle

Definición de variables y *labels*

- Dos tipos de variable: **byte o db** (8 bits) y **word o dw** (16 bits).
- Los arreglos también pueden ser de estos tipos.
- Los datos se almacenan en orden ***little endian*** para variables dw.
- `LEA reg,var` → ***L**oad **E**ffective **A**ddress*. Almacena en el registro `reg` la dirección de la variable `var` (en vez de usar `MOV A,var` -equivalente a `MOV A,Lit`- como en el computador básico).

Arquitectura x86 - ISA en detalle

Definición de variables y *labels* - Ejemplos

- Supongamos las siguientes variables:

```
var1 db 0x0A
var2 dw 0x07D0
arr1 db 0x01,0x02,0x03
arr2 dw 0x0A0B,0x0C0D
```

- En la tabla se muestra cómo quedan almacenadas las variables y arreglos.

Variable	Dirección de memoria (16 bits)	Palabra almacenada
var1	0x0000	0x0A
var2	0x0001	0xD0
	0x0002	0x07
arr1	0x0003	0x01
	0x0004	0x02
	0x0005	0x03
arr2	0x0006	0x0B
	0x0007	0x0A
	0x0008	0x0D
	0x009	0x0C

Arquitectura x86 - ISA en detalle

Ejemplo de código - Multiplicación

```
a = 10
b = 200
res = 0
while (a > 0):
    res += b
    a -= 1
print(res)
```

Programa en pseudocódigo (Python).

```
MOV AX,0
MOV CX,0
MOV DX,0
MOV CL,a      ;CL guarda el valor de a
MOV DL,b      ;DL guarda el valor de b
start:
  CMP CL,0    ;IF a <= 0 JMP end
  JLE end
  ADD AX,DX   ;AX += b
  DEC CL      ;a -= 1
  JMP start
end:
  MOV res,AX
  RET         ;Fin del programa.
a     db 10   ;Definición variables
b     db 200
res   dw 0
```

Programa en x86.

Como las variables y las instrucciones se guardan en la misma memoria, se definen al final del código para que no se interpreten como instrucciones (deben ser antecedidas por el RET del programa).

Arquitectura x86 - ISA en detalle

Instrucciones de multiplicación y división

- op = Registro o literal

- `MUL/DIV` = Operación sin signo
  `IMUL/IDIV` = Operación con signo

- **`MUL op`** → **`AX = AL * op`** (op 8 bits)
  `MUL op` → **`DX|AX = AX * op`** (op 16 bits)
  `DIV op` → **`AL = AX ÷ op, AH = AX % op`** (op 8 bits)
  `DIV op` → **`AX = DX|AX ÷ op, DX = DX|AX % op`** (op 16 bits)

Arquitectura x86 - ISA en detalle

Ejemplo de código - Multiplicación con `MUL`

```
MOV AX,0
MOV AL,a        ;AL = a
MUL b           ;AX = AL * b
MOV res,AX      ;res = AX
RET             ;Fin del programa.
a       db 10   ;Definición variables
b       db 200
res     dw 0
```

Programa en x86, drásticamente reducido. CISC, ¡énfasis en el *hardware* para reducir la complejidad del *software*!

Arquitectura x86 - ISA en detalle

Ejemplo de código - Promedio

```
arr = [6, 4, 2, 3, 5]
n = 5
i = 0
avg = 0
while (i < n):
    avg += arr[i]
   i += 1
avg /= n
print(avg)
```

Programa en pseudocódigo (Python).

```
MOV CL,n            ;CL guarda el valor de n
MOV SI,0            ;SI = i
MOV AX,0            ;AL = sum(arr)
MOV DX,0            ;DL = arr[i]
start:
  CMP SI,CX         ;IF i >= n JMP end
  JGE end
  LEA BX,arr        ;BX = address arr
  MOV DL,[BX+SI]    ;DL = arr[i]
  ADD AL,DL         ;AL += arr[i]
  INC SI            ;SI += 1 = i+1
  JMP start
end:
  DIV n             ;AL = AX ÷ n
  MOV avg,AL        ;avg = AL
  RET               ;Fin del programa.
n      db 5         ;Definición variables
arr    db 6,4,2,3,6
avg    db 0
```

Programa en x86.

Arquitectura x86 - Subrutinas

Las subrutinas de x86 son más complejas que en el computador básico.

- En el computador básico, el *stack* contiene la dirección de retorno.
- El computador básico **no tiene convención** para almacenamiento explícito de parámetros, valor de retorno o variables locales.
- En x86, **explicitaremos el uso de los elementos anteriores**, haciendo uso del registro `BP` para manejar estos datos.

Arquitectura x86 - Convención de llamada

Una convención de llamada define la **interfaz sobre la que trabajará el código de la subrutina**. Debe especificar lo siguiente:

- Ubicación de parámetros (*stack*, registros, ambos).
- Si se usa el *stack*, el **orden** en el que se almacenan.
- Definición de responsabilidades de restauración del *stack* entre la subrutina y el código que la llama.

Arquitectura x86 - Convención de llamada

Existen múltiples convenciones (*stdcall*, *cdecl*, *fastcall*, *safecall*, *syscall*, *thiscall*, etc.).

Estas varían, como se señaló anteriormente, entre el responsable de restaurar el *stack* y el uso de memoria o de registro para almacenar parámetros, retorno y variables locales.

Particularmente, revisaremos el uso de ***stdcall***, donde la **subrutina se encarga de limpiar el *stack***.

Arquitectura x86 - *stdcall*

Esta convención especifica lo siguiente:

- Los parámetros se almacenan en el *stack* **de derecha a izquierda**.
- El valor de retorno se almacena en el registro **`AX`**.
- La subrutina debe dejar al registro `SP` **apuntando en la misma posición que tenía antes del paso de los parámetros**.

Adicionalmente, con la ayuda del registro `BP`, podremos permitir llamadas anidadas (**recursión**) y variables locales.

Arquitectura x86 - *stdcall*

Al inicializar una subrutina, el estado del *stack* queda según lo expuesto en la figura.

Además de almacenar los parámetros y la dirección de retorno, se almacena también el valor del registro BP para poder separar las variables locales utilizadas en cada llamada.

SP → BP-4,BP-3 BP-2,BP–1	Variables locales
BP →	Base Pointer anterior a la llamada
BP+2,BP+3	Dirección de retorno
BP+4,BP+5 BP+6,BP+7	Parámetros de la subrutina

Arquitectura x86 - *stdcall*

Se deben realizar dos pasos al momento de **llamar** una subrutina:

1. Agregar los parámetros al *stack* mediante la instrucción `PUSH`. Los valores agregados **solo pueden ser de 16 bits**.

2. Llamar a la subrutina mediante la instrucción `CALL`. Al igual que con el computador básico, esto almacena la dirección de retorno y realiza el salto a la primera instrucción de la subrutina.

Arquitectura x86 - *stdcall*

Ejemplo de llamado

```
;Cálculo de potencia
MOV BX,0
MOV CX,0
MOV BL,exp
MOV CL,base
PUSH BX       ;Paso de parámetros
PUSH CX       ;de derecha a izquierda
CALL power    ;power(base, exp)
MOV pow,AL    ;Retorno viene de AX
RET
power:
...
base  db 2
exp   db 7
pow   db 0
```

Programa en x86 que ejemplifica el formato de llamado de subrutina. No nos interesa el contenido de la subrutina en sí en este caso, sino que nos importa ver el paso de parámetros, la llamada y la lectura final del valor de retorno.

Arquitectura x86 - *stdcall*

Se deben realizar seis pasos **dentro** de una subrutina:

1. Guardar el valor actual de `BP` en el *stack* y cargarle el valor de `SP`:
   ```
   PUSH BP
   MOV BP,SP
   ```

2. (Opcional) Si se usan **variables locales**, se debe reservar espacio para estas moviendo a `SP` *n* posiciones hacia arriba, siendo *n* el número de palabras de memoria que usan estas variables:
   ```
   SUB SP,n
   ```

Arquitectura x86 - *stdcall*

Se deben realizar seis pasos **dentro** de una subrutina:

3. Ejecutar la subrutina. Los parámetros se acceden mediante direccionamiento indirecto con registro `BP` y *offset*. De esta forma, el primer parámetro se ubica en la dirección `BP+4`, el segundo en `BP+6`, etc. (`BP+2` almacena la dirección de retorno y `BP` el *base pointer* anterior). Las variables locales, por su parte, se acceden con *offset* negativo: primer variable local en `BP-2`, segunda variable local en `BP-4`, etc.

Arquitectura x86 - *stdcall*

Se deben realizar seis pasos **dentro** de una subrutina:

4. (Opcional) Si se usaron **variables locales**, se debe recuperar el espacio que se reservó restaurando la posición de `SP`:
   `ADD SP,n`

5. Se **rescata** el valor previo de `BP` con la instrucción `POP`:
   `POP BP`

6. Se restaura el valor de `SP` con `RET` según la cantidad de palabras de memoria utilizada por los parámetros: `RET n`

Arquitectura x86 - *stdcall*

Ejemplo completo - Cálculo de potencia

```
;Cálculo de potencia
MOV BX,0
MOV CX,0
MOV BL,exp
MOV CL,base
PUSH BX             ;Paso de parámetros
PUSH CX             ;de derecha a izquierda
CALL power          ;power(base, exp)
MOV pow,AL          ;Retorno viene de AX
RET
power:
  PUSH BP           ;Guardamos valor de BP
  MOV BP,SP         ;BP = SP
  MOV CL,[BP+4]     ;CL = base
  MOV BL,[BP+6]     ;BL = exp
  MOV AX,1          ;AX = 1 (valor de retorno)
  start:
    CMP BL,0        ;IF exp <= 0 JMP end
    JLE end
    MUL CL          ;AX = AL * CL = AL * base
    DEC BL          ;BL -= 1 === exp -= 1
    JMP start
  end:
    POP BP          ;Recuperamos BP original.
    RET 4           ;SP se desplaza 4 bytes (2 parámetros de 2 bytes cada uno).
base      db 2
exp       db 7
pow       db 0
```

Programa en x86 que calcula el valor de una base elevada a un exponente con subrutinas siguiendo la convención de llamadas stdcall.

Arquitectura x86 - *stdcall*

Ejemplo completo - Cálculo de potencia con variables locales

```
;Cálculo de potencia
MOV BX,0
MOV CX,0
MOV BL,exp
MOV CL,base
PUSH BX                 ;Paso de parámetros
PUSH CX                 ;de derecha a izquierda
CALL power              ;power(base, exp)
MOV pow,AL              ;Retorno viene de AX
RET
power:
  PUSH BP               ;Guardamos valor de BP
  MOV BP,SP             ;BP = SP
  SUB SP,2              ;Reservamos una variable local (i)
  MOV CL,[BP+4]         ;CL = base
  MOV BL,[BP+6]         ;BL = exp
  MOV AX,1              ;AX = 1 (valor de retorno)
  MOV [BP-2],0          ;i = 0
  start:
    CMP [BP-2],BL       ;IF i >= exp JMP end
    JGE end
    MUL CL              ;AX = AL * CL = AL * base
    INC [BP-2]          ;i += 1
    JMP start
  end:
    ADD SP              ;Restauramos el espacio que ocupó la variable local i.
    POP BP              ;Recuperamos BP original.
    RET 4               ;SP se desplaza 4 bytes (2 parámetros de 2 bytes cada uno).
base        db 2
exp         db 7
pow         db 0
```

Programa en x86 que calcula el valor de una base elevada a un exponente con subrutinas siguiendo la convención de llamadas stdcall y usando variables locales. Notar que en este caso no alteramos el valor del parámetro de la base, ya que la variable local *i* se encarga de actualizar su valor para manejar la iteración.

Arquitectura x86 - *stdcall*

Ejemplo completo - Uso de recursión

```
;Cálculo de factorial
MOV BX,0
MOV BL,n
PUSH BX             ;Paso de parámetros
CALL factorial      ;factorial(n)
MOV fact,AL         ;Retorno viene de AX
RET
factorial:
  PUSH BP           ;Guardamos valor de BP
  MOV BP,SP         ;BP = SP
  MOV BL,[BP+4]     ;BL = n
  CMP BL,0          ;IF n == 0 JMP basecase
  JEQ basecase
  DEC BL            ;Parámetro para llamada factorial(n-1)
  PUSH BX           ;Paso de parámetros
  CALL factorial    ;factorial(n-1)
  MOV BL,[BP+4]     ;Recuperamos el parámetro n original
  MUL BL            ;AX = AL * n = factorial(n-1)*n
  JMP end
  basecase:         ;Caso base de la recursión (0! = 1)
    MOV AX,1
  end:
    POP BP          ;Recuperamos BP original.
    RET 4           ;SP se desplaza 2 bytes (1 parámetros de 2 bytes).
n       db 5
fact    db 0
```

Programa en x86 que calcula el factorial de un parámetro *n* vía recursión. Aquí sale a la luz la importancia del registro BP: si no se utilizara, no tendríamos forma de recuperar el parámetro *n* de cada llamado recursivo.

Al igual que en cualquier otro lenguaje, no olvidar el caso base y término de la recursión.

Ejercicios

Si bien históricamente se ha evaluado la programación en x86, en las últimas versiones del curso hacemos uso de ***otra*** ISA para programar en un contexto más *real*.

Por ende, los contenidos de esta materia se evaluarán solo a nivel conceptual.

Antes de terminar

¿Dudas?

¿Consultas?

¿Inquietudes?

¿Comentarios?

DCC

DEPARTAMENTO DE CIENCIA
DE LA COMPUTACIÓN

IIC2343

Arquitectura de Computadores

Clase 8 - Arquitectura x86

Profesor: Germán Leandro Contreras Sagredo