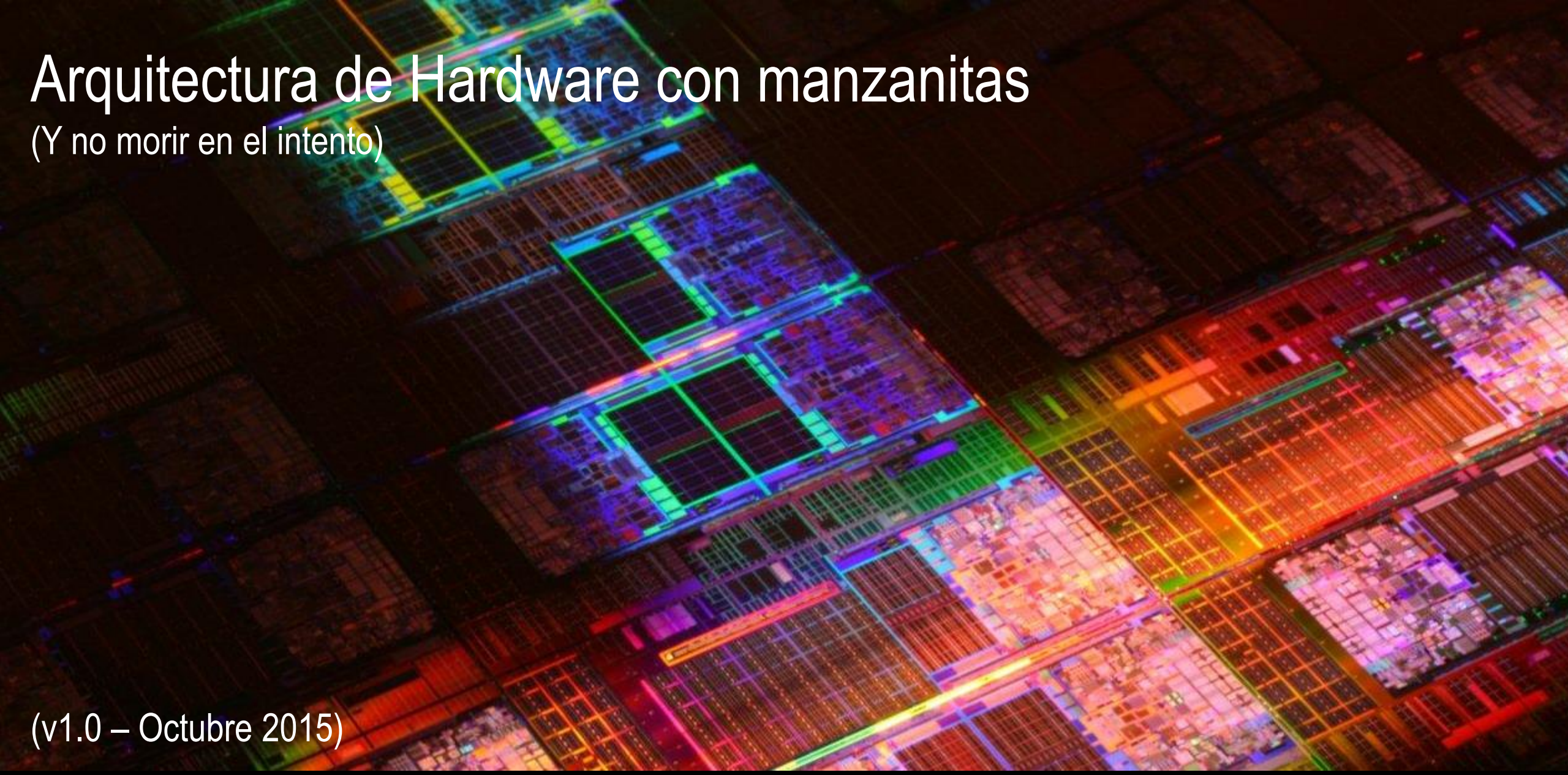
Arquitectura de Hardware con manzanitas

(Y no morir en el intento)



(v1.0 – Octubre 2015)

Enzo Barbaguelatta D.

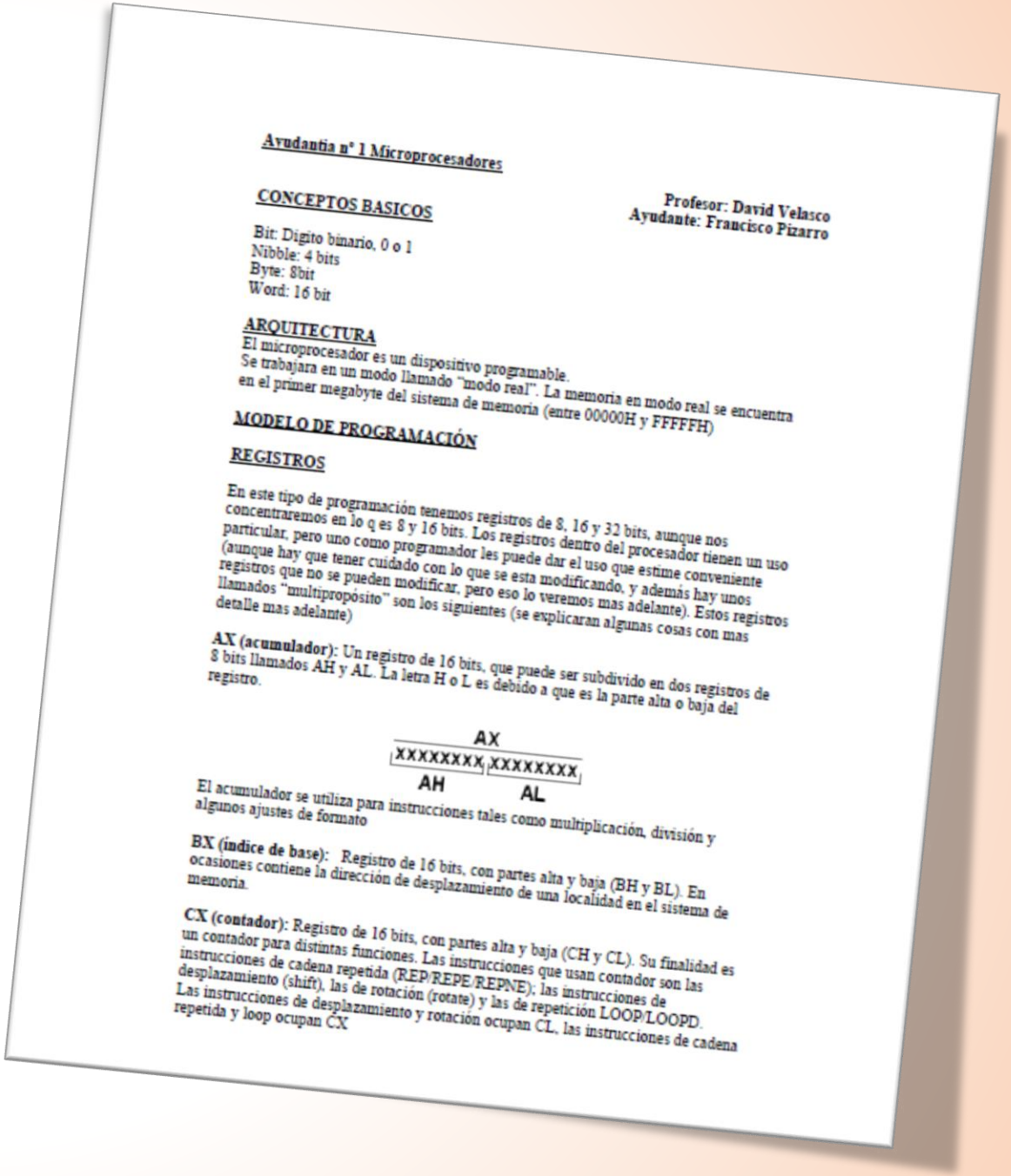
PPT de apoyo para los cursos de Arquitectura de Hardware de Ingeniería Civil Informática PUCV

Acerca de.

- ¡Hola! =)
- Esta es una PPT de apoyo a la asignatura, hecha por algún ayudante ya inexistente de esta asignatura en informática. ¿Porque? En este ramo se ve mucha terminología no obvia, y muchos se pierden (es un ramo netamente electrónico para informáticos), por lo que consideré un tanto necesario confeccionar algo como esto.
- Esta PPT no necesariamente refleja TODA la verdad de lo que se ve en el curso de arquitectura. **¡PUEDE TENER ERRORES!** (Si, soy humano, me puedo equivocar) Por ende, si encuentras algo sospechoso: ¡por supuesto: Puedes editarlo y corregir las cosas que no parezcan correctas para que los pobres de futuras generaciones no sufran… tanto. Y no te sientas ofendido si digo algo que no es verdad. Solo trato de explicar las cosas lo mejor que se puede (y es probable que 60% de las cosas que diga en la PPT sean mentira ☺).
- No esperen aquí un set de diapositivas guía para el curso. Esto es, como ya dije "*Un material de apoyo*", con bastante lenguaje coloquial (y espero me entiendas y no te sientas ofendido si te trato mal a veces). El material oficial son las PPTs del profesor, y por supuesto: ¡El ir a clases!

Material de apoyo:

- Esta PPT esta basada en varias referencias, pero principalmente del legendario **Manual de Ayudantía de Microprocesadores**.
- En informática, es mas conocido como "**La biblia de arqui**"
- Es un documento bastante completo hecho por otro ex-ayudante hace varios años atras, sobre como usar assembler de buena forma, además de tener buenos ejercicios tipo prueba ☺
- Puedes conseguirlo en la fotocopiadora del FIN/bioquimica , bajarlo del aula (si es que está disponible), o bien vía el link al final de la PPT.



Ayudantia n° 1 Microprocesadores

CONCEPTOS BASICOS

Profesor: David Velasco
Ayudante: Francisco Pizarro

Bit: Digito binario, 0 o 1
Nibble: 4 bits
Byte: 8bit
Word: 16 bit

ARQUITECTURA

El microprocesador es un dispositivo programable.
Se trabajara en un modo llamado "modo real". La memoria en modo real se encuentra en el primer megabyte del sistema de memoria (entre 00000H y FFFFFH)

MODELO DE PROGRAMACIÓN

REGISTROS

En este tipo de programación tenemos registros de 8, 16 y 32 bits, aunque nos concentraremos en lo q es 8 y 16 bits. Los registros dentro del procesador tienen un uso particular, pero uno como programador les puede dar el uso que estime conveniente (aunque hay que tener cuidado con lo que se esta modificando, y además hay unos registros que no se pueden modificar, pero eso lo veremos mas adelante). Estos registros llamados "multipropósito" son los siguientes (se explicaran algunas cosas con mas detalle mas adelante)

AX (acumulador): Un registro de 16 bits, que puede ser subdivido en dos registros de 8 bits llamados AH y AL. La letra H o L es debido a que es la parte alta o baja del registro.

AX
XXXXXXXX XXXXXXXX
AH AL

El acumulador se utiliza para instrucciones tales como multiplicación, división y algunos ajustes de formato

BX (indice de base): Registro de 16 bits, con partes alta y baja (BH y BL). En ocasiones contiene la dirección de desplazamiento de una localidad en el sistema de memoria.

CX (contador): Registro de 16 bits, con partes alta y baja (CH y CL). Su finalidad es un contador para distintas funciones. Las instrucciones que usan contador son las instrucciones de cadena repetida (REP/REPE/REPNE); las instrucciones de desplazamiento (shift), las de rotación (rotate) y las de repetición LOOP/LOOPD. Las instrucciones de desplazamiento y rotación ocupan CL, las instrucciones de cadena repetida y loop ocupan CX

¿Empecemos?

¿Arquitectura de Hardware?

- No, no es lo que piensas.

Entonces ¿que?

- Simple: Este ramo se llama así porque conoceremos como funciona el computador a muy bajo nivel.
- (Entiéndase bajo como nivel *ultra detalloso*).
- De hecho, este ramo antes se llamaba Arquitectura de Comptuadores.
- … y en electrónica se llama Microprocesadores
- … y sí, es la misma w3@.

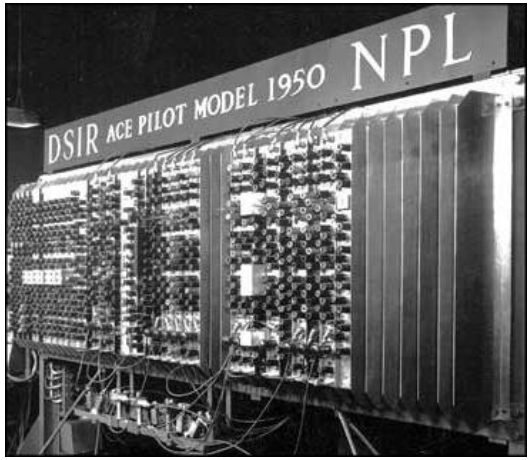
Primero… que es un computador.

- Esto es un computador

- Esto también

- Este también… y es igual de poderoso que tu compu hace 5 años atrás.

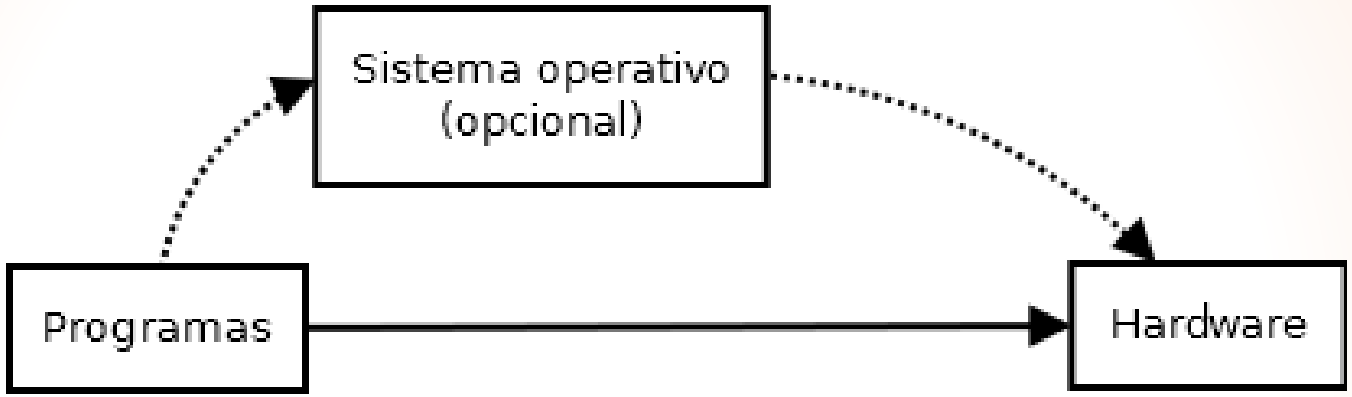
- Esto también



- Y esto

- Bueno… esto no

Como funciona esto?

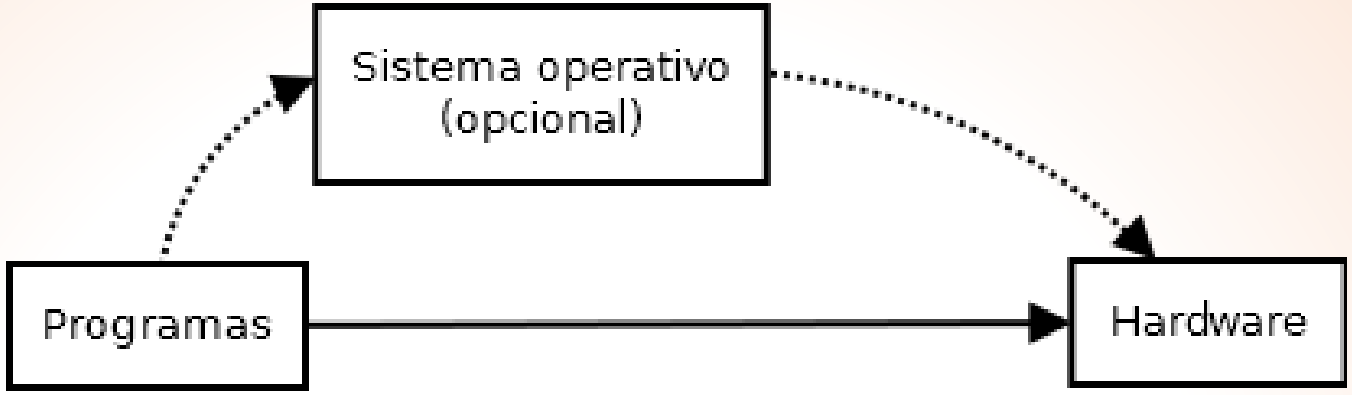


¿Porque el sistema operativo es opcional?

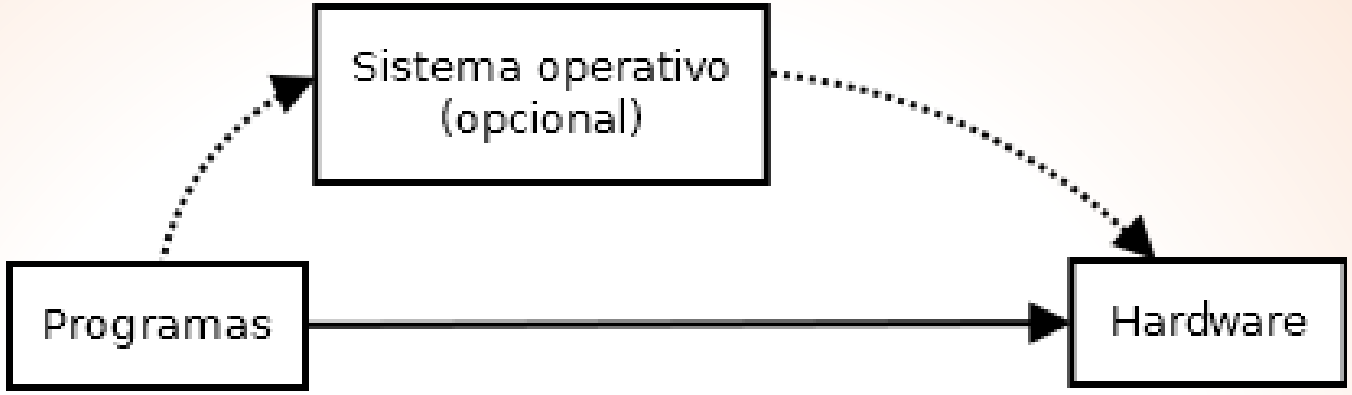
Programas simples a veces ni lo necesitan, pueden hacer todo por ellos mismos. Pero si la cosa es mas compleja, y necesitas correr muchas cosas a la vez, tener algo que maneje la RAM y todo de manera eficiente y bonita, ahí necesitaras algo que maneje eso (eso es lo que hace el Sistema Operativo).

No necesita sistema operativo

Este si (usa Linux)

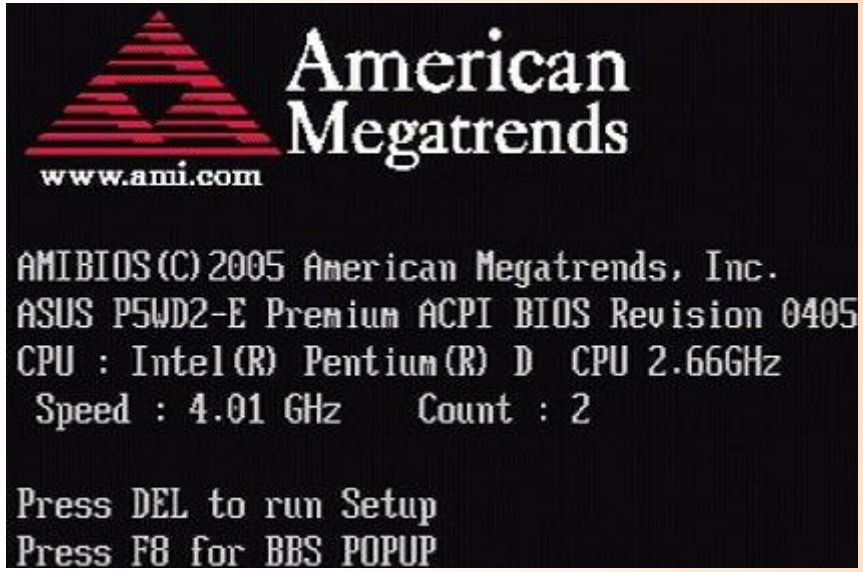


- Los programas se comunican con el Hardware mediante … alguna interfaz.
- En español chilensis: DRIVERS. *(Si, esos malditos programas que hay que buscar cuando formateamos compus).*
- Pero… Si no están instalados los drivers, ¿como cresta el compu funciona igual? *(pero mas lento/feo/sin sonido/sin internet/blablablá).*
- Windows tiene drivers genéricos por si las moscas… *¿pero fuera de Windows o en aparatos como ese lector de tarjeta BIP de la ppt anterior?*
- Ahí hay una interfaz común genérica que permite una comunicación básica entre los programas y el hardware.



- Si es un aparato especifico como el lector de BIP's o un circuito integrado… para programarlo me imagino que te pasarán la documentación para como trabajar con tal hardware.
- ¿Pero en los compus/celus/tablets? ¿Tantas marcas distintas y modelos distintos y los programas funcionan igual uno a los otros?
- ¿Recuerdan PLP/Progra Avanzada? ¿Interfaces? Si, eso que hace que todos los aparatos tengan algo en común para que en todos lados funcione igual.

Esa interfaz en los compus “tarro/notes” se llama BIOS



(Si, este fabricante de BIOS usa la triforce de logo xD)

- O mas bien “*Basic Input/Output system*”, o Sistema básico de entrada/salida.
- Cuando el compu esta fresco recién prendido (cuando ni siquiera carga Windows), la BIOS hace de puente entre los programas y tu compu… para que funcione y no tengas un pedazo de ladrillo con pantalla.
- Aunque ahora se esta cambiando por otra interfaz mas actualizada llamada UEFI (Unfied extensible firmware interface). La BIOS ha sido super útil… pero data del año 1975… Supondrías que desarrolladores de sistemas operativos les complica eso un poco, eh?

Dejemos algo claro.

Si no existiese esta interfaz, tendríamos un bonito y caro pisapapeles.

- ¿Me imagino que ya sabes que la parte mas importante en un compu es el procesador, no?
- No pienso contar los detalles aquí, ni la historia entera… vayan a clases flojos.
- ¡Los hay de muchos tipos y marcas, muchos! Seguramente al escuchar de procesadores se te venga a la mente 2 palabras: Intel y AMD (xD)
- Déjame decirte que existen muchas mas: ARM (Android/iOS), PowerPC (viejos MAC, otros compus extraños) RISC/MIPS (Muchos sistemas embebidos, Nintendo64, El computador de Richard Stallman (para forzarse a usar Linux xD)), Zilog Z80 (hartos aparatos chicos, Pac-man), Cell (PlayStation 3), EmotionEngine (PlayStation2), y un laaaaaaaaaaaaargo etc).
- Cada tipo de procesador tiene su arquitectura, y necesitas ponerle programas hechos para esa tal arquitectura. (¿Te has preguntado porque no puedes abrir juegos de Play 1 solo haciéndole dos click?)
- Obviamente un programa para cierta arquitectura no correrá en otra arquitectura. ☹
- … Bueno, puedes usar emuladores, pero eso ya es otra historia.

- Nosotros, en nuestros compus tarros/notes que usamos todos los días, usamos la arquitectura **Intel x86/AMD64.**
- En el curso veremos la arquitectura **Intel 8088.**

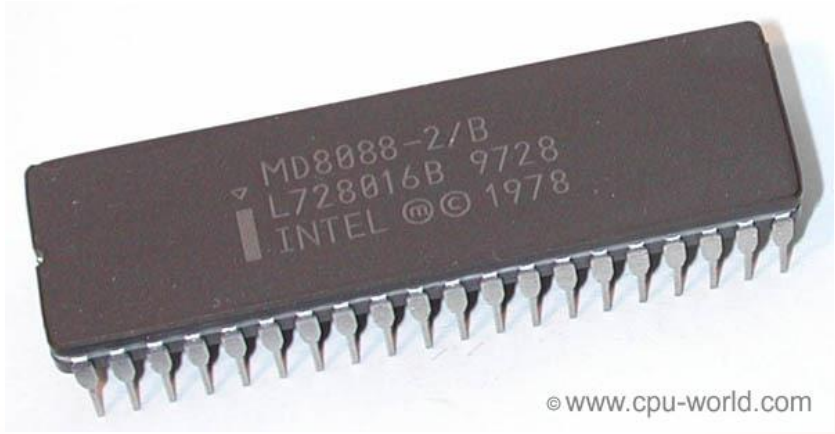
- Pero… **x86/AMD64 != 8088**

… Cagamos?

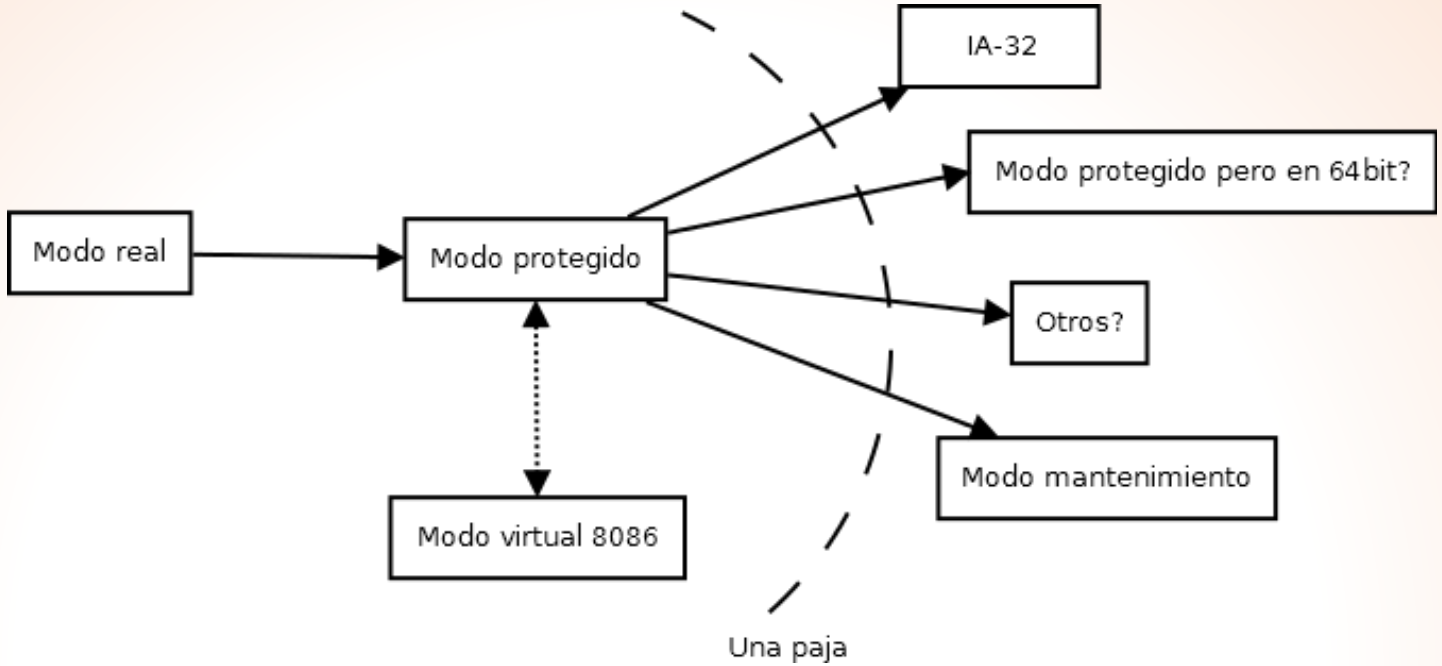
NO

- Intel x86/AMD64 es compatible con 8088… Como es eso?
- ¿Porque crees que se llama x86? Es por decir (*cualquiercosa*)86. Pero es 8088… es casi igual a su hermana cuasi gemela que es 8086, asi que nos sirve xD. (Shh, no preguntes detalles, es así porque si nomas xD)

- Para entender todo esta ensalada de términos tenemos que primero entender como rayos funcionan estos procesadores.



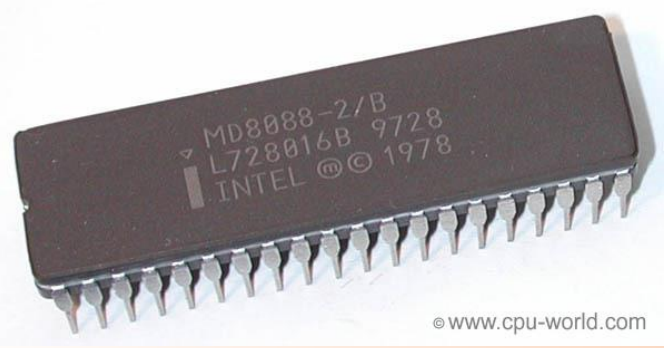
(¿Notan alguna maldita diferencia entre el 8088 y el 8086?)



- Nuestro procesador puede funcionar en hartos modos como pueden ver aquí… pero nos centraremos en 3 importantes:
- Modo real
- Modo protegido
- El resto de los modos mas allá del borde que etiquete como "*Una lata*"

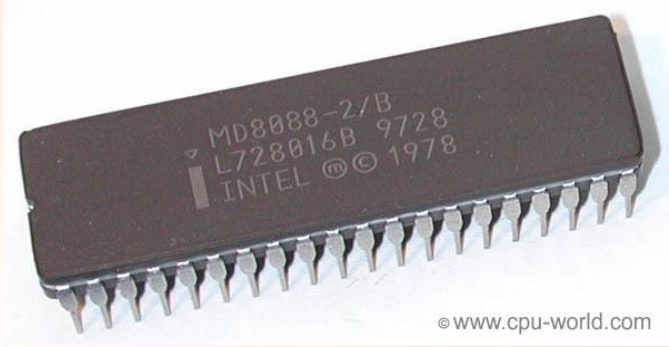
(Dice eso porque realmente trabajar con esos modos es una lata… palabras del profe… pero en términos mas bonitos… no lo intente en casa ☺)

Modo real



- O bien “Como hacer que mi Intel Core i7 de ultima generación tenga el mismo poder que un Intel 8088 del año 1978 y desaprovechar todo su potencial” ☺
- En este modo, el computador funcionará en una especie de modo de ultra-compatiblidad, para que pueda correr “teóricamente” cualquier instruccion base 8086 … Incluyendo a nuestro querido 8088 que necesitaremos ☺
- Es el modo default del procesador. Cuando prendes tu compu, el procesador estará en este modo.
- Suena bonito todo esto, pero no les he contado la mejor parte: las Desventajas.

Modo real



- Solo puedes manejar registros de 16 bit (ya veremos que es eso) y por ende, solo correr códigos para 16 bit. O sea en los días de hoy, casi ningún programa.

  (Empecemos con el hecho de que muy probablemente tu Windows sea de 64 bit).

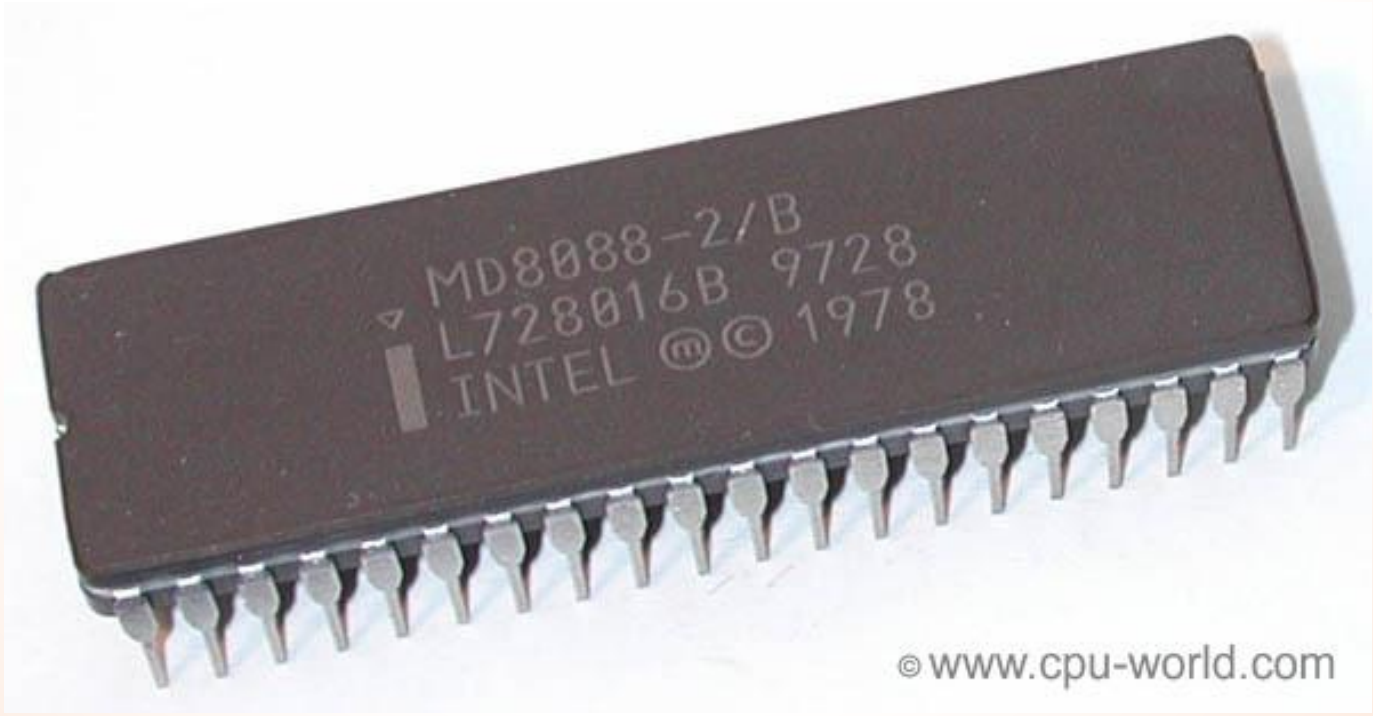
- ¿16 bit? ¡Sorpresa! Solo puedes manejar hasta 1 Mb de RAM. Si tienes 8GB de RAM, desaprovecharas 7,999 gigas :D

- ¿Que rayos puedo hacer con un mega de RAM? Con un MP3 colapsaría mi compu ☹

Dato rosa

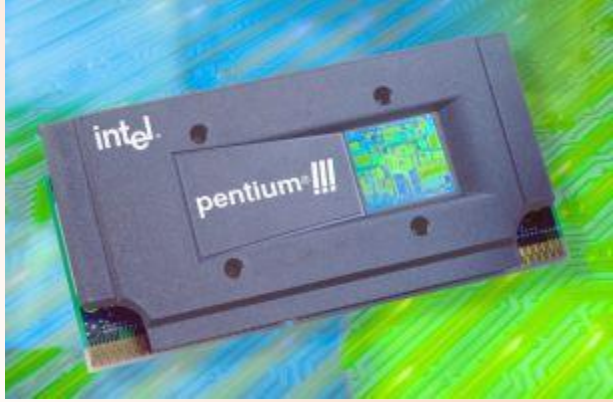
- ¡Te cuento! En 1961 en la misión Apollo 11 para llegar a la Luna, los tipos usaron un computador llamado Apollo Guidance Computer (AGC) con unos ridículos 64 kb de RAM y un procesador de 0.043 Megahertz… ¡Y lograron llegar a la Luna!
- Mientras que tu tienes un computador de 4-8 núcleos a 3 Gigahertz con 8 GB de RAM, con sistemas operativos de vanguardia y complejos sistemas de comunicación mundial para … ¿postear en Facebook y reclamar que tienes lag?
- ¡Tal computadorcillo lo programo una tal Margaret Hamilton, y incluso, programó un código extra que salvó a la tripulación de un problema bastante grande! Si, ese es el código del computador.

- ¿Te convences que con un 8088, muchísimas veces mas poderoso que el computador del Apollo 11 puedes hacer maravillas?



Modo protegido

No es la descripción mas acertada/correcta… pero para entenderlo sirve bastante.
Una mejor descripción de esto lo tendrás para la materia de la segunda parte del curso.



- Pero para usar mas RAM y tener mas características el modo Real se nos queda chico. Necesitamos un modo que maneje harta RAM… de manera incluso grosera.

- Aquí nace el Modo Protegido. Tiene este nombre porque ya que hay tanta RAM en el compu, se necesitan de mecanismos para administrarla bien entre muchos programas y mantener todo el ecosistema estable.

- Por supuesto, de esto ya no se encarga la BIOS (cuek), por lo que tenemos que tener algo que maneje un compu con modo protegido… ¡Un sistema operativo!

- Windows 95 fue el primer sistema que usaba ¡Modo protegido de buena forma!

Modo protegido

- Ahora tenemos registros (ya veremos eso) de 32 bits…¡y por ende mas RAM!… ¡muchísima mas RAM! Podemos usar hasta 3,algo GB.
- Seguramente conocerás este modo como "*Un sistema de 32*" xD
- Modo Protegido lo puedes ver en todos los compus que funcionan en 32 bit.
- Para entrar a modo Protegido si o si hay que pasar primero en Modo Real. Esto sucede cuando Windows empieza a cargar.
- BIOS es solo para modo real. En modo protegido no sirve. Empezamos a necesitar drivers.

Modo protegido

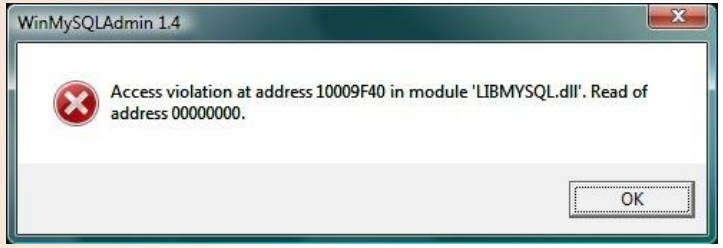
- En modo protegido puedes correr cualquier programa hecho para Intel x86 de 32 bits… en español chilensis, casi todos los programas que existen, incluido Java, tus tareas de progra, el Chrome, el Word, y lejos el mas importante: el Solitario.

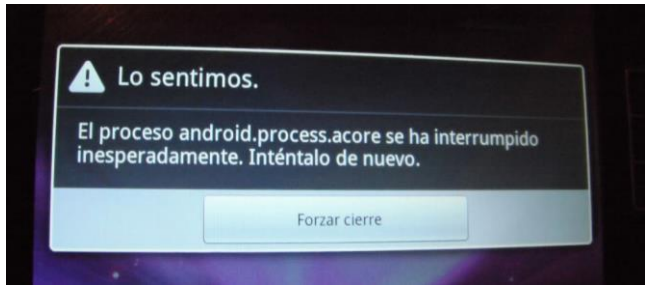
- Hay un modo de compatibilidad igual llamado Modo virtual 8086. Puedes correr programas de modo real de todas formas. (Aunque en la practica funcionan bastante mal). Este es el porque varios programas viejos de los 90 para DOS funcionan aún en Windows XP.

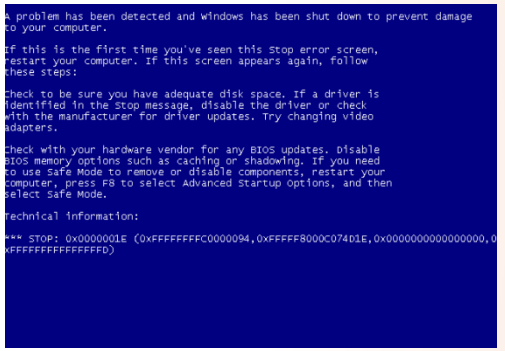
- … Y esta es la verdadera razón de porque los programas viejos no funcionan en los computadores mas modernos (teóricamente si tienes un Windows 10 de 32 bits, funcionarían juegos viejos de DOS, pero en uno de 64 no).

Modo protegido

- Ya que es el Modo Protegido, y el sistema operativo maneja todo lo que es RAM y aparatos, cuando algo funciona mal, el sistema operativo puede detectarlo, atraparlo, matar los programas problemáticos y mostrar errores bonitos.

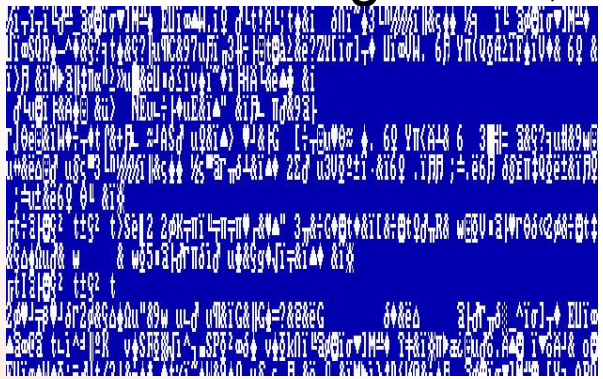






(Bueno… no tan bonitos)

- En modo real no pasa eso. Si algo falla, a tu computador le dan espasmos. (Luego veremos porque).

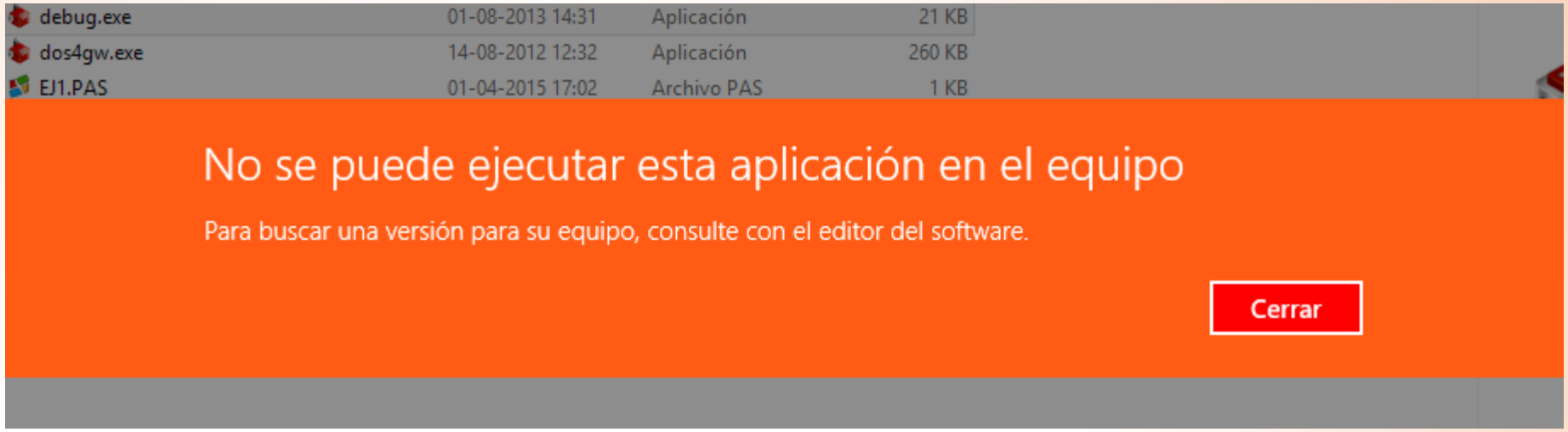


- Si quieres tener mas de 4 GB de RAM, necesitaremos registros de mas de 32 bit… ¡64 bit!

- ¿Es de suponer que ese es otro modo, no?
- Pero ahí ya no me pregunten porque no se (soy el ayudante, no el profe ☹)

- En estos modos de 64 bit ya no es posible en lo absoluto correr código 8086… ni siquiera en ese modo virtual ese. (No obstante puedes correr programas de 32 bit sin problema).

Problemas

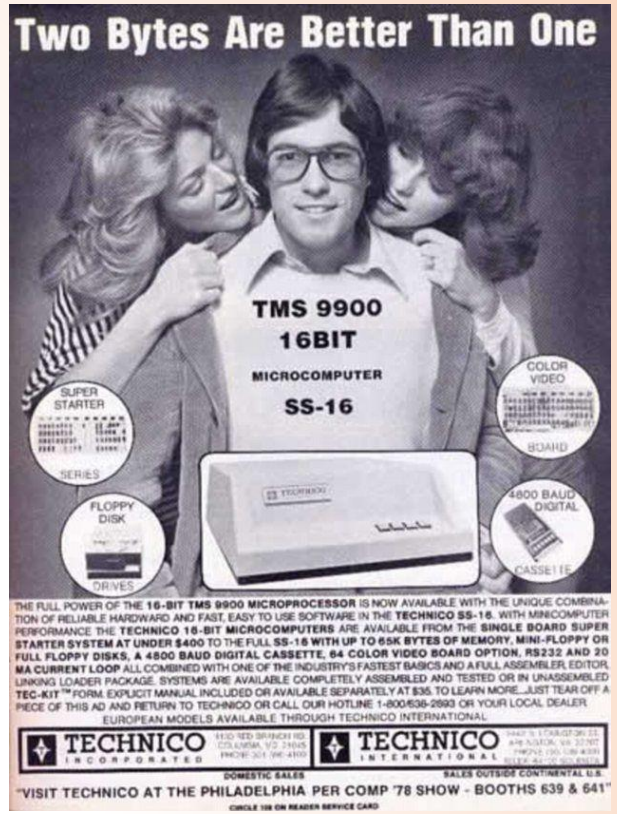


- Nosotros usaremos modo real en esta asignatura.
- Nuestros computadores la mayoría funcionan a 64 bit, con sistemas de 64. (Tienen de 32? Pueda que les funcione, pero es cosa de suerte lamentablemente).
- Por ende es imposible correr los programas de la asignatura en Windows.
- ¿Que solución tenemos entonces?

Conseguirse PC’s de los 80-90?

- ¿Conseguirse poderosos computadores de 16 bit, ultraportables (necesitaras un carrito para eso), mientras vas vestido con camisa de leñador, y lo dejas encima de la mesa mientras le dices al profesor ***“ESTOY LISTO PARA MI FUTURO LABORAL!”*** ?
- Seamos sinceros… ¿donde cresta podemos HOY conseguir un compu con características de los 80-90?
- … Mejor digamos todos juntos: NEXT!

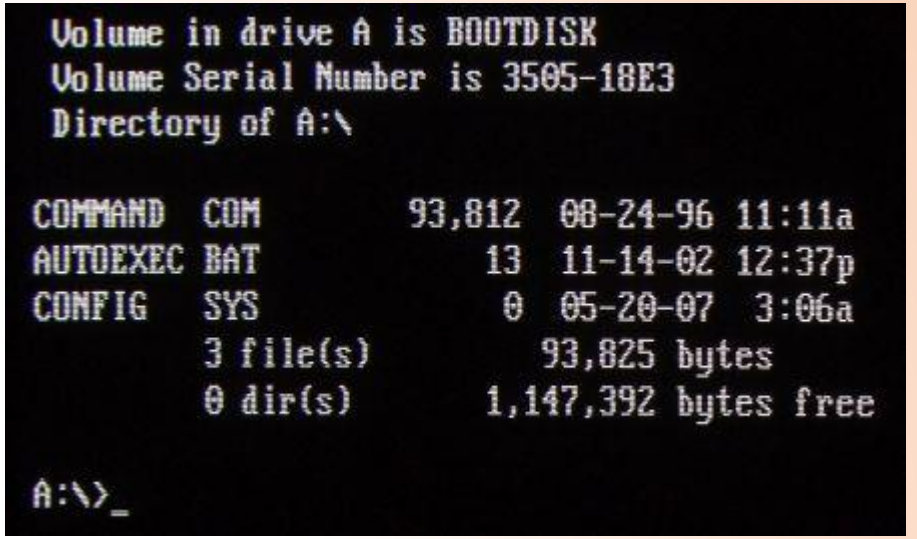
- (Hay computadores de estos en el tercer piso si lo deseas :D)

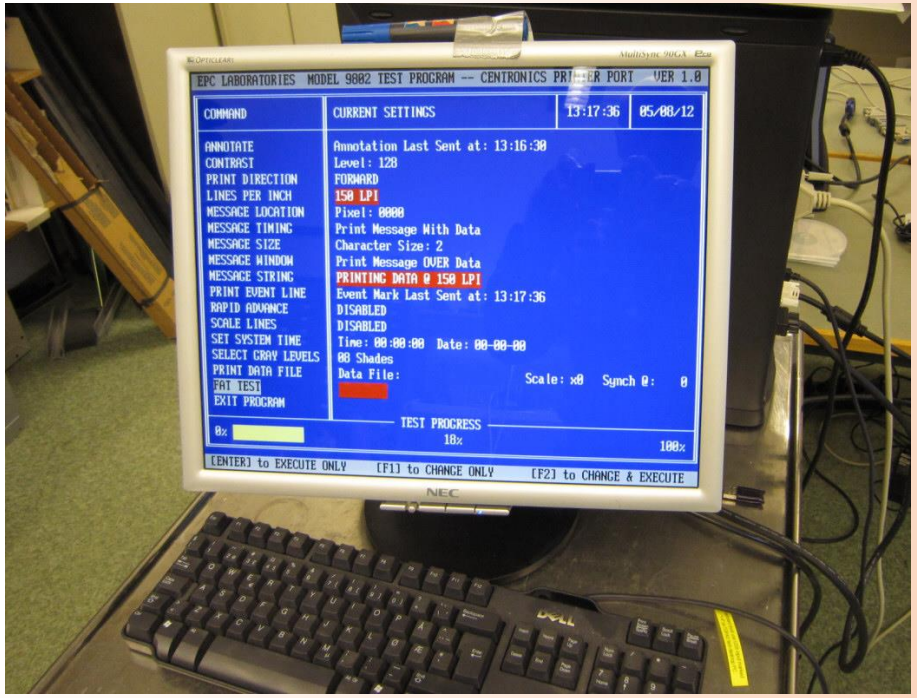


Usar nuestro compu en modo real?

- Olvidarnos de Windows y "*bootear*" nuestro compu en modo real.
- Se usaría un sistema arcaico llamado DOS, que funciona en modo real.
- El profe lo hace así en clases.

- DESVENTAJAS:
- Descontinuado. Era popular hasta los principios de los 90 pero ya no se usa.
- No tienes soporte para NTFS (tu disco duro), USB (pendrives) ni nada de eso. ¿Como guardarás archivos?
- ¡Es modo real! Y es peligroso. Si te equivocas en algo, (manejar tu disco duro por ejemplo) puedes echar a perder cosas.

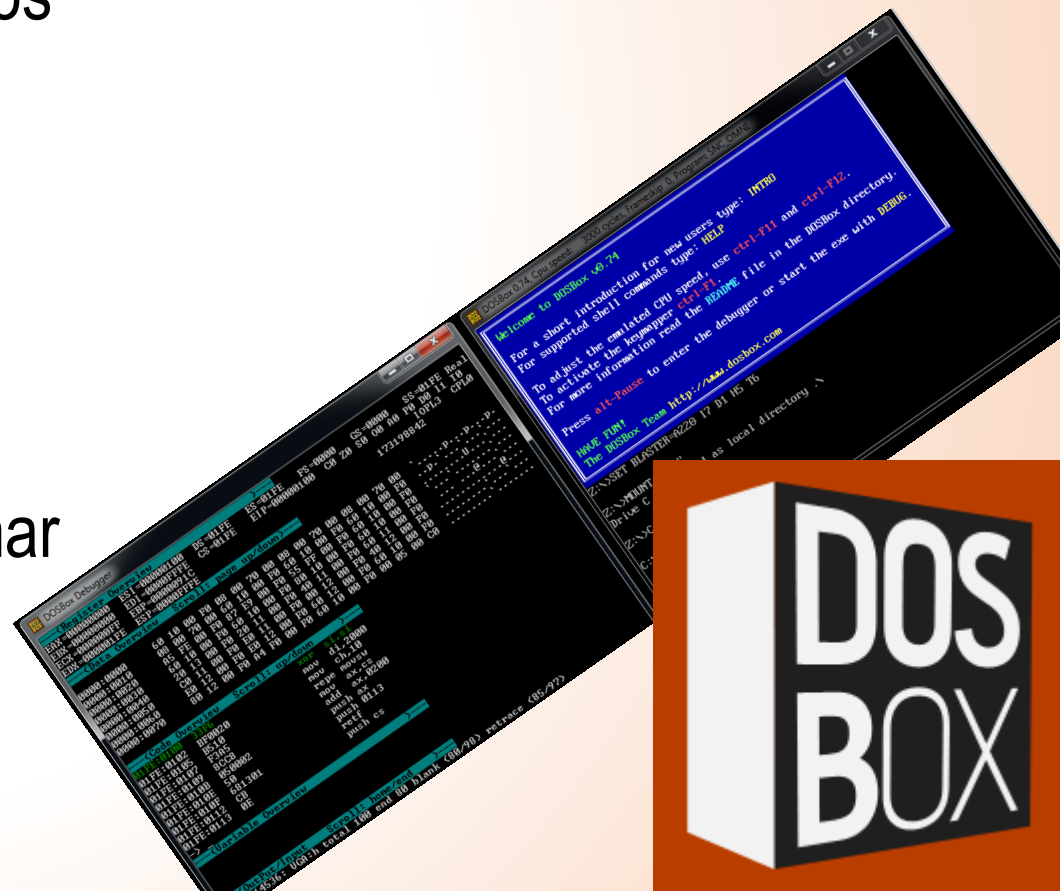




Usar emuladores?

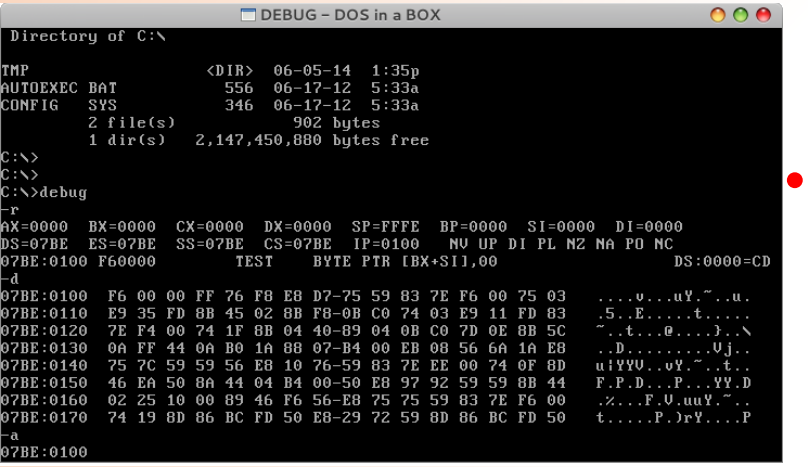
- 2 opciones.
- ¿Una maquina virtual? (VMWare, VirtualBox) puede ser útil, de hecho, lo es. Lo malo es todo el aparataje que hay que hacer para montar todos los programas que necesitaremos… y es bastante pesado. Mejor vámonos a la opción B.

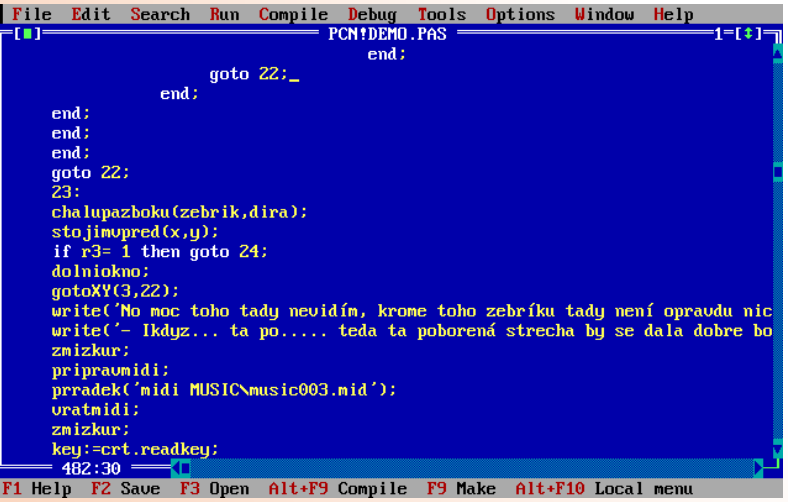
- La segunda opción es usar un emulador de DOS (recuerda, DOS es modo real) llamado **DOSBox**. Originalmente es para juegos, pero para programar en el curso basta y sobra.



- …Y funciona también en MacOSX y Linux :D

Usaremos estos programas





- **DOSBox**, obviamente, para emular el modo real.
- **MS-DOS Debug**, herramienta para ver en detalle el funcionamiento del computador.
- **TurboPascal**… No, no programaremos en Pascal, pero sirve para programar en Assembler, que es el "lenguaje" que usaremos para programar. (*Nada de TurboAssembler, ni MASM, ni FASM ni otro compilador. TurboPascal… se exigirá para la tarea*).

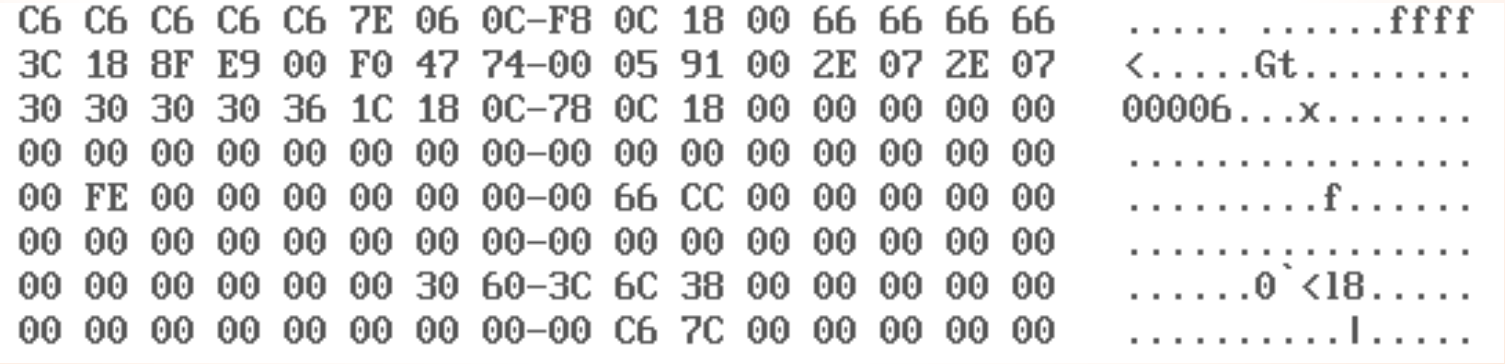
Todos estos programas están en un kit que les preparé llamado **DOSBox-arquitectura-hardware**, llegar y ejecutar.

- Descárguenlo del aula, o le preguntan al ayudante.
- También disponible para bajar en la última PPT.

¿En que programaremos?

- Vamos a trabajar a nivel procesador, por lo que trabajaremos a nivel lenguaje maquina. En otras palabras, el set de instrucciones (datos con código) en que el procesador usa para ejecutar operaciones… de procesador.

- Estas son operaciones super básicas tales como mover elementos de un lado a otro, leer en memoria, escribir en memoria, hacer sumas restas multiplicaciones, divisiones, comparar entre otros.



(Aaaagh! Que es ESTO!? Esto es chino @_@)

¿Q... QUE? Programaremos en esos números feos?

```
C6C6C6
C6C67E
06
0CF8
0C18
006666
66
66
3C18
8FE9
00F0
47
7400
059100
2E
07
2E
07
```

- Oh, por supuesto que no.
- Para empezar esos “numeritos” de la ppt anterior, las instrucciones están todas enlatadas sin ordenar. Si ponemos saltos de líneas a un pedazo de ese montón de números, veremos que realmente las instrucciones (o sea, el código) se ve así.
- Aquí cada línea se trataría de una instrucción. Lo que en un curso de programación llamarías una ”Línea de código”, solo que aquí no es código.
- Ok, les seré sincero. Esto aun no es entendible, y solo los Coreanos podrían quizá leerlos.

¿Y si traducimos esos códigos en palabritas?

```
MOV     DH,C6
MOV     DH,7E
PUSH    ES
OR      AL,F8
OR      AL,18
ADD     [BP+66],AH
SCASB
INC     BYTE PTR [BX+SI]
LOCK
INC     SI
JZ      0115
ADD     [BP+SI+BZ00],DH
PUSH    SS
CWD
ADD     [ZE07],CH
POP     ES
```

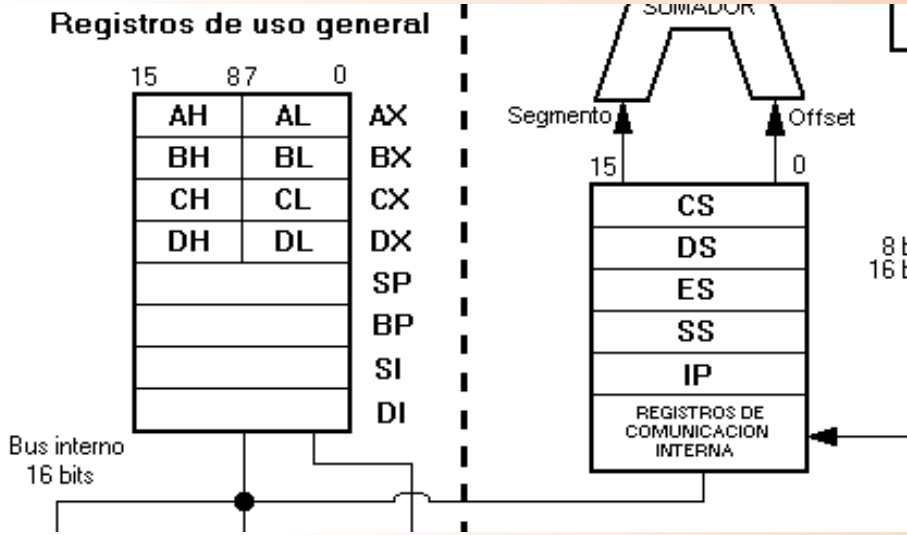
(Que bien, mucho mas entendible… oh wait)

- Buena idea.
- De hecho, eso hicieron los desarrolladores para hacer la programación mas entendible ☺
- Esto se llama “Instrucciones desensambladas”, o lenguaje ensamblador.
- … O al grano: “*Assembler*”.
- No se compila, se ensambla (solo hay que traducir a esos numeritos feos).
- Y si, este “lenguaje” usaremos en la asignatura.

¿Como declarar variables en assembler?

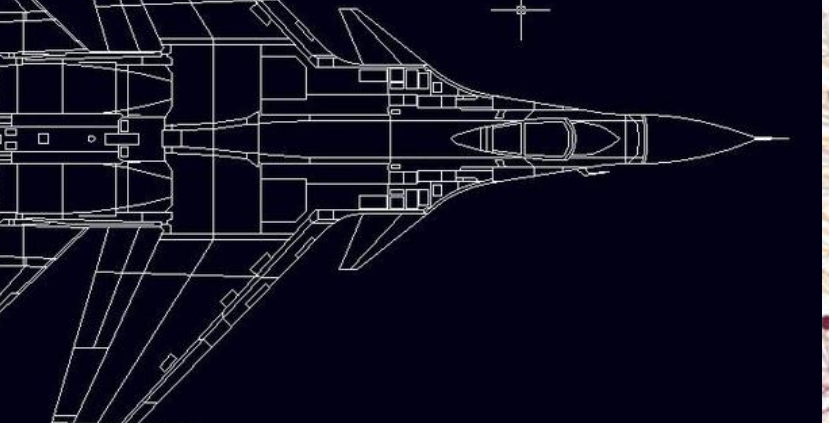
- … ¿Variables?

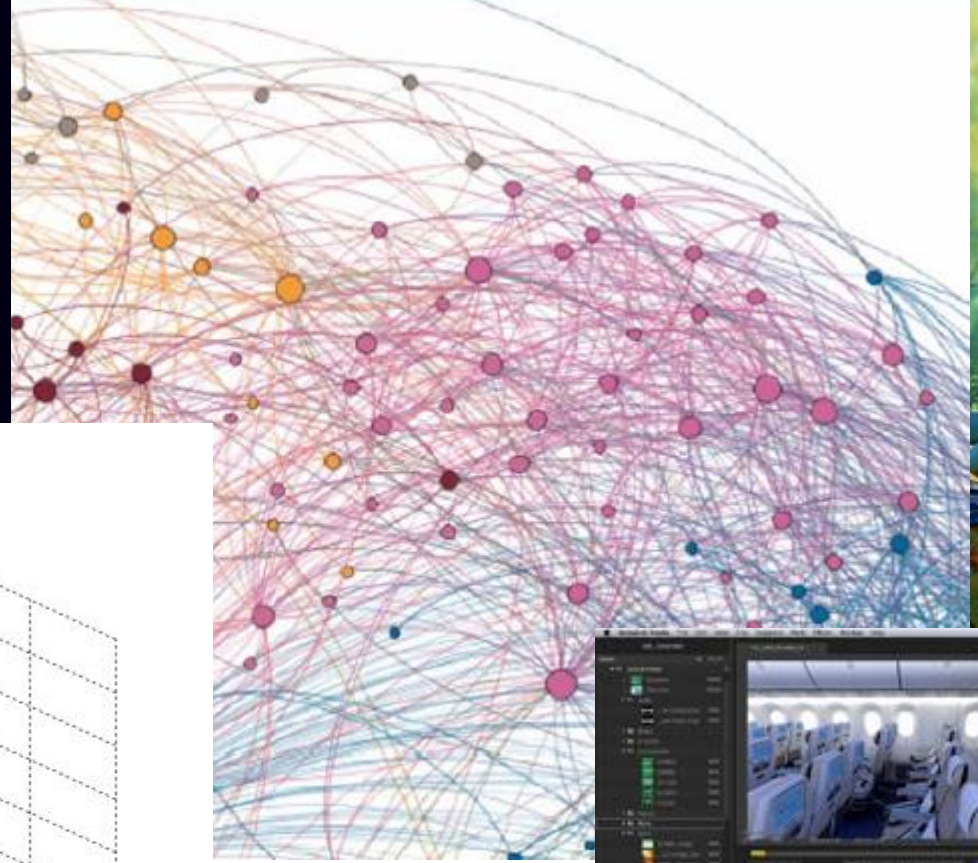
- Lo siento amigo/a. Aquí no hay variables.
- ¡Pero hay registros!
- Los registros son una especie de “variables” internas del procesador que hacen lo mismo: Guardar valores.
- ¿La diferencia? Las variables se guardan en la RAM, y puedes crear la cantidad que se te antoje. ¿Los registros? NO, pero se guardan en la CPU y son rápidas.
- Si miramos la imagen de al lado, vemos que hay 18 registros con nombres bastante feos (AX, BP, CS, etc).
- Buena parte de estos son para mantener en orden la ejecución de la CPU, y otros no se recomiendan usarlos en cualquier situación por lo que aproximadamente nos quedamos con… ¿aproximadamente 8 registros?

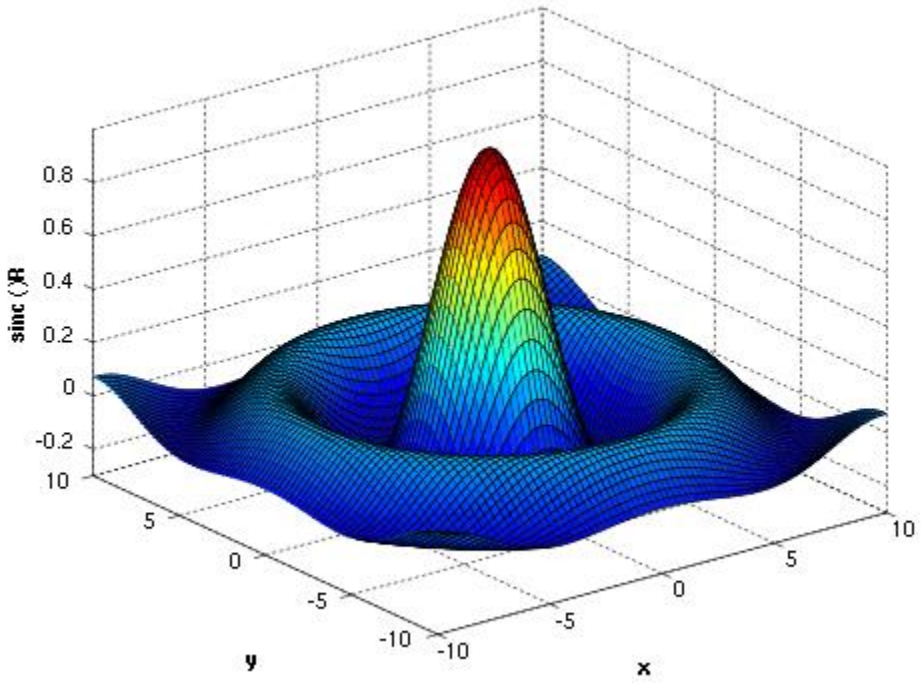


Con manzanitas:
Registros = algo así como variables ☺

¿¿¿¡Y que diantres puedo hacer con 8 malditos registros!???







Mejor dicho… ¿que no puedes hacer con 8 registros?

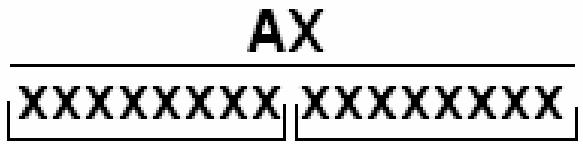
Todo tu computador con programas y juegos funcionando a la vez son gracias a estos registros...

¿Y como esto es posible?

- Imaginate un compu de 2.5 Ghz. Bastante común verlo hoy en día.
- Recuerda que 1 Hz es 1 periodo partido por unidad de tiempo.
- Además 2.5 Ghz = 2 500 000 000 Hz.
- En otras palabras en apenas 1 segundo tu procesador ha procesado el estado de tu computador con todos tus programas corriendo al menos unas 2 mil 500 millones de veces aproximadamente. (Imagínate con doble núcleo y con overclock)
- Y si lo miramos, podemos hacer lo que queramos si organizamos bien el compu, y hacer que el computador procese un programa con 500 variables… en turnos de 4 en 4, o de 8 en 8, por decir un ejemplo.
- Y esto ocurre taaaaaan rápido que ni lo notas. ☺

Registros y tamaños

- Los registros que usaremos en este curso serán de 16 bit y algunos de 8 bits. (Los hay de 32 y de 64, pero no los veremos… al menos para programar).
- Cuando la gente dice que “un compu es de 32 o 64 bits” en realidad se refiere a que este maneja registros de tamaños de 32 y 64 bits.
- Un compu de 16 bit (modo real) no puede manejar mas de 1 mega por que los registros no dan abasto para referirse a espacios de memoria mas allá. Tendrían que haber números mas grandes.
- ¿Para qué alcanza entonces en 16 bits? Nos alcanzan 2 bytes.
- Un byte: Un numero de 0x00 a 0xFF. O en decimal un número del 0 al 255.
- Entonces en un registro de 16 bits podemos manejar números de 0x0000 a 0xFFFF



Dato rosa:

Cuando en un lenguaje encuentres números con 0xalgo, ese 0x siempre se refiere a que hablamos de números hexadecimales.

Repasemos tamaños

(¡Muchas veces en los ejercicios mencionan tamaños, para que no te quedes colgado!)

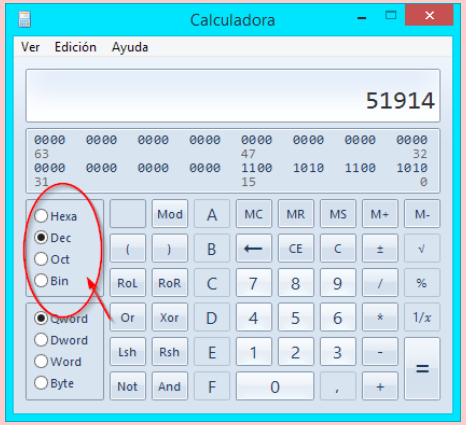
- **Bit: 1 o 0.** (10100100001 xD)
- **Nibble :4 bits.** O bien un número hexadecimal del 0x0 al 0xF (0 al 15).
- **Byte: 8 bits.** O bien un número hexadecimal del 0x00 al 0xFF (0 al 255).
- **Word (o palabra): 16 bits.** O bien un número hexadecimal del 0x0000 al 0xFFFF.

- ¿Me vas entendiendo la idea? ¿Podrás adivinar que numero hexadecimal podrás guardar en 32 o 64 bits?

Dato rosa:

Para transformar rápidamente de hexadecimal/binario a decimal y viceversa tienes esta alternativa.

Calculadora de Windows: Si vas al menú de arriba “Ver” y pones Programador, veras que al lado aparece para convertir de base rápidamente.



No te confíes mucho tampoco, en este curso es mejor pensar todo en hexadecimal porque el sistema decimal mucho no existe.

De hecho, te eximes con un 0x32 según reglamento de escuela ;)

Tipos de registros

- Los 4 registros más básicos son el **AX, BX, CX y DX.**
- La gente sugiere ciertos usos para este tipo de registros, **pero en realidad puedes usarlos como se te de la gana.**
- El AX por ejemplo la gente lo suele usar como acumulador (ejemplo: Si tienes menos de un 39 entonces aumento la cantidad de reprobados en 1).
- El BX lo suelen usar para manejar direcciones de memoria. Casi siempre cuando leemos o escribimos de memoria, o para escribir registros mas problemáticos (luego veremos eso), la gente suele hacer operaciones con BX.
- CX lo suelen usar de contador. Si, como los famosos i,j cuando haces for en tu lenguaje de programación preferido.
- DX lo suelen usan para cuando hay que multiplicar/dividir algo y cuando quieren comunicarse con otros aparatos en el compu.

AX

BX

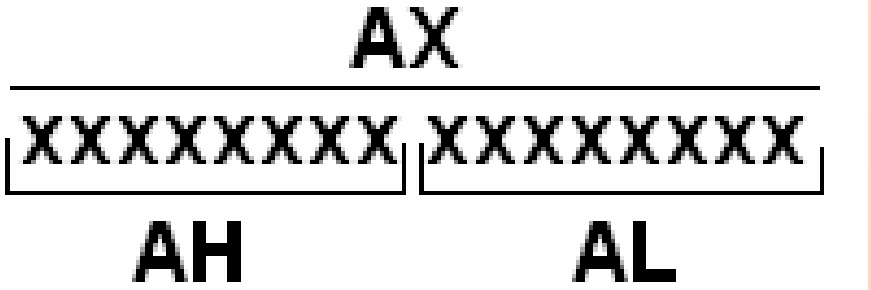
CX

DX

Un pequeño truco

- ¿Muy pocos registros? Hay una manera para ganar un puñado de registros extra a partir de AX, BX, CX, DX.
- Hemos dicho que estos eran de 16 bits, no? Pues, podemos dividir estos registos en 2 y tener 2 registros nuevos: Por ejemplo, en el caso de AX -> AH y AL.
- AH y AL se refieren a la parte alta y baja. O sea, la primera y la segunda mitad de la variable.
- En estos registros, solo puedes guardar valores de 8 bits (0x00 a 0xFF).
- Solo puedes hacer esto con AX, BX, CX, y DX.

CX -> CH - CL



Tipos de registros

- Otros registros: BP (base pointer), DI (destination index), SI (source index).
- BP lo suelen usar para apuntar a la memoria cuando trabajas con datos.
- DI y SI se usan cuando se operan con Strings u otro tipo de datos en masa. Como ya dije, tu ves si los usas para Strings.

BP

DI

SI

Registros específicos/ segmentos.

(A.K.A Mejor no los toques si no te lo piden.)

- Estos registros existen para tu deleite… Y los puedes modificar si quieres. Pero te aseguro, que si los tocas obtendrás un ataque epiléptico instantáneo en tu PC. Así que mejor no los toques.

(No obstante hay gente que lo usa ¿Usos comunes hoy en día? Hacks raros, cracks, virus, parches custom para programas, entre otros…)

- CS (Code segment) contiene la dirección de memoria donde esta el programa que tu CPU ejecuta (hey, tiene que ejecutar programa de algún lado, ¿no crees?). Se acompaña con IP (Instruction pointer).
- SS (Stack segment) contiene la dirección de memoria donde se guarda la pila. Se usa con SP (Stack pointer). Esta pila guarda cosas importantísimas como estado de programas, llamadas a "funciones", e incluso otros usos que nosotros le daremos.

CS

IP

SS

SP

Registros específicos/ segmentos.

(A.K.A Mejor no los toques si no te lo piden.)

- Resumen:

Si quieres mantener tu compu lindo bonito y estable, mejor no modifiques estas cosas y dejalas ser felices!

CS

IP

SS

SP

Otros registros específicos ya no peligrosos.

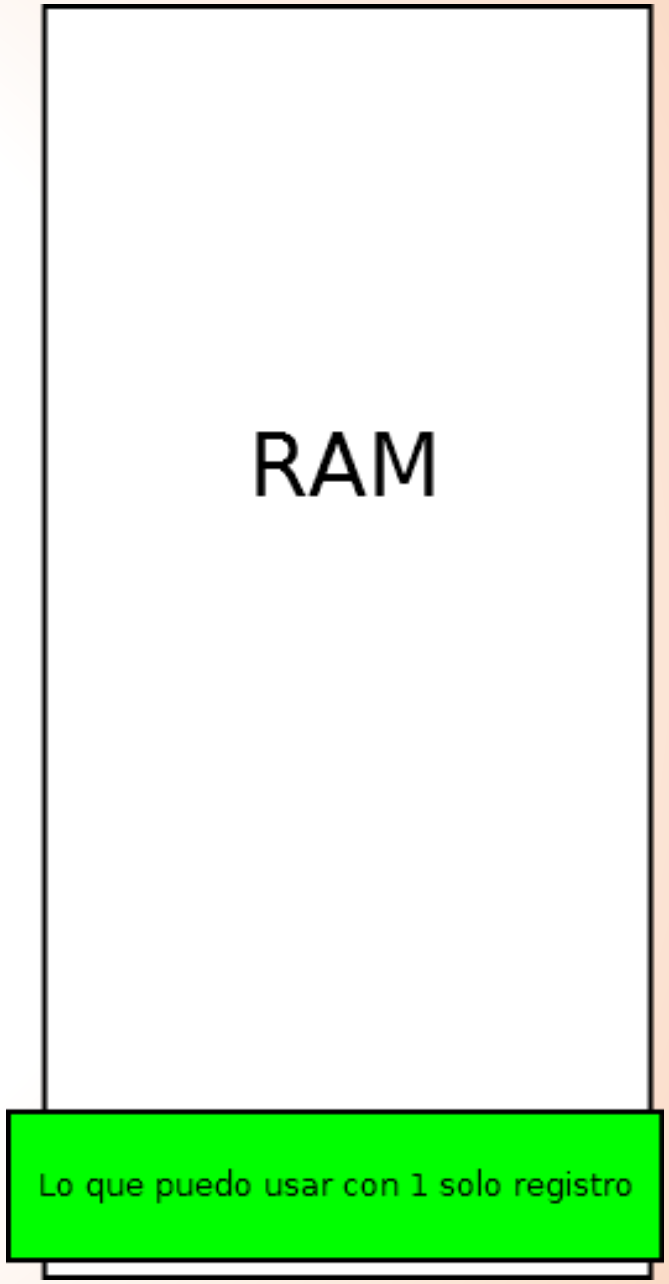
- En cambio, DS (Data segment) y ES (Extra Segment) no se usan intensivamente por el computador y puedes usarlos.
- Pero operar con ellos para un uso cualquiera puede ser bastante impráctico.
- Guardan direcciones de memoria para leer/escribir datos utilizados por tus programas.
- Estos 2 los usaremos especialmente para leer y escribir en Memoria RAM, donde éstos juegan un papel crucial. Eso lo veremos mas adelante.

DS

ES

Memoria en modo real.

- Si, ya lo dije: Solo 1 MB de RAM… pero algo no cuadra aquí.
- Podemos apuntar realmente SUPER pocos lugares en la RAM con solo un registro.
- Pero con un registro de 16 bit solo nos da abasto para ridículos 64 kb! ☹
- ¿Como leo el resto de mi escasa RAM? No me digas que tenemos aun MENOS espacio.
- … No, tranquilízate, hay forma.



¡Usando 2 registros en vez de uno!

- En el maravilloso (pero engorroso a morir) mundo del modo real, las direcciones de memoria están en un formato llamado **Segmento**:**Offset.**
- Las direcciones aquí se dicen de esta forma: **0000**:**0000**
- **La primera parte se llama segmento**. Como dice su nombre, indica en que parte de la memoria nos debemos parar en general, pero sin entrar en detalle. (Analogía: Decir **Ciudad de Viña del Mar**).
- **La segunda parte se llama offset.** Eeeeh no tiene una traducción literal pero podríamos decir que es el “corrimiento”, o la dirección mas en detalle. (Analogía: **Decir 5 Oriente**)

A000:3333 CS:IP DS:BX

Analogía: Viña del Mar:5 Oriente

¡Usando 2 registros en vez de uno!

- Ahora, para traducir a que espacio de memoria realmente estamos apuntando *(nosotros jamás en el colegio aprendimos a leer números como 3563:23421124)* hay que hacer un pequeño calculo.

Mejor lo explico con un ejemplo, transformemos A000:3333

```
A000
 3333 +
--------
A3333
```

-> A3333

(Si, todo esto solo para leer un maldito espacio de memoria.)

¡Usando 2 registros en vez de uno!

- **RECOMENDACIÓN:**
- Si vas a andar recorriendo direcciones de memoria:

- En el Segmento deja una dirección fija, ojalá de 1000.
- Y en el offset anda acumulando de a poco.
- Cuando el offset no pueda mas, súmale 1000 al segmento.
- Así estará todo en orden ☺

A000:FFFF -> AFFFF

El offset no puede mas, súmale al segmento 1000.

B000:0000 -> B0000

Como la CPU ejecuta ~~código~~ instrucciones.

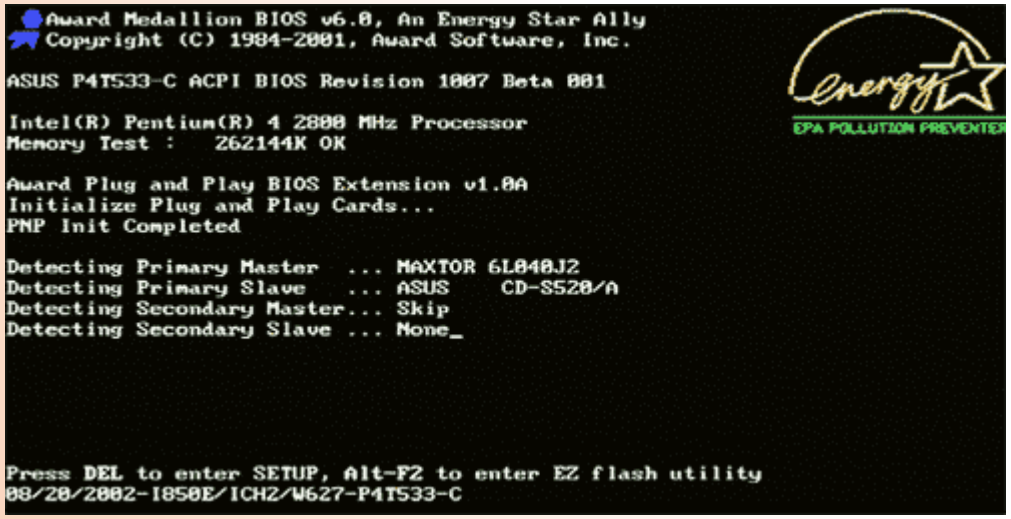
(Al profe no le gusta que le digan código… En assembler son instrucciones.)

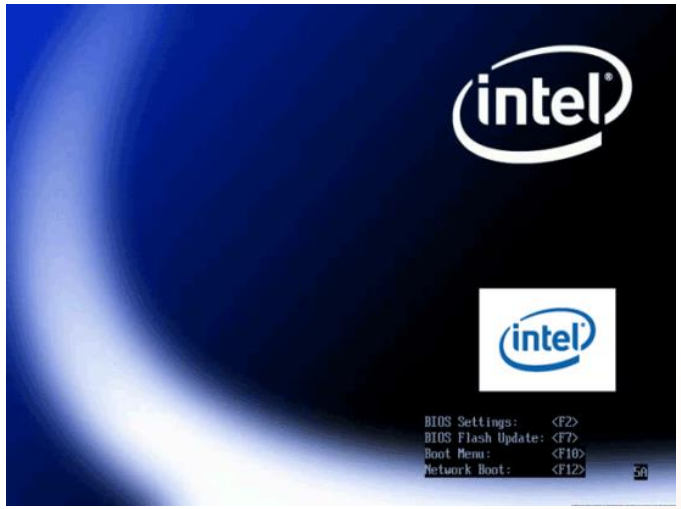
- Tu CPU fue hecho para ejecutar instrucciones.
- De hecho, su tarea mas importante es ejecutar todo lo que esté en la dirección de memoria **CS**:**IP** *(He ahí el porque no debes tocarlo).*
- A medida que va ejecutando instrucciones, IP avanza solito para leer las aun más instrucciones que siguen.
- Y así siempre… Desde que prendes hasta que apagas el PC.

- Pero… la CPU no viene con instrucciones.
- Debe haber algo en el computador entonces que provea a la CPU de instrucciones.

Como la CPU ejecuta ~~código~~ instrucciones.

- ¡Cuando tu PC se prende necesita ejecutar algo!
- Es ahí cuando la ROM BIOS toma el control del computador para hacer el proceso de encendido.
- La BIOS se coloca en la posición de memoria F000:0000 y pone al procesador a ejecutar cosas allí.
- Inicializa después la BIOS de video y incia un mini-programa llamado POST, mas conocido como "*El logo de la marca cuando prendes el compu :D* "





(Carnet al suelo con la BIOS de la estrellita! :_( )

Como la CPU ejecuta ~~código~~ instrucciones.

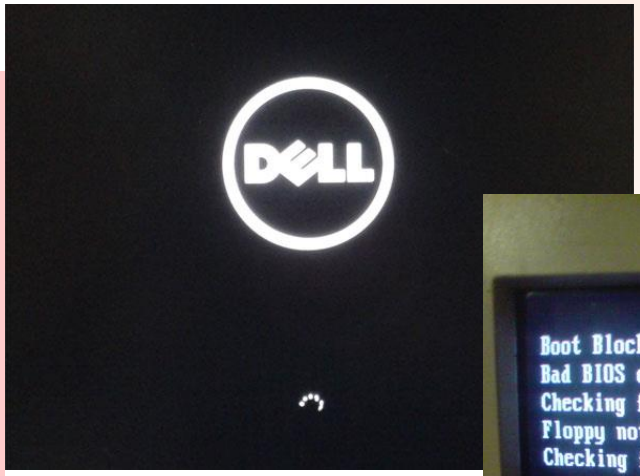
Datos rosas:

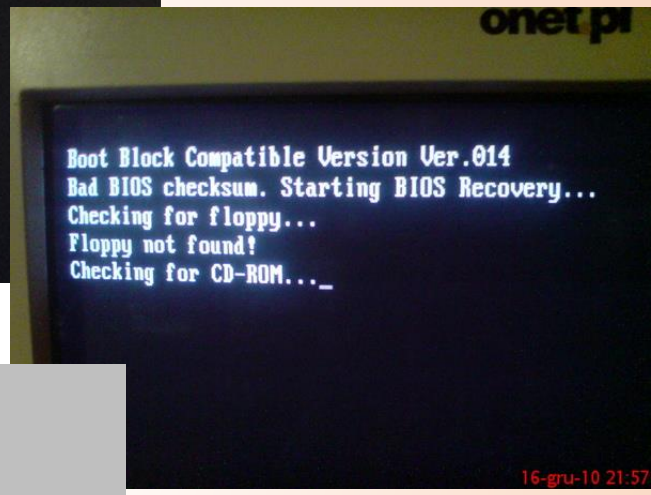
-Los computadores nuevos con UEFI ya no usan BIOS. Cuando se prenden aparece el logo de la marca e inmediatamente cargan el sistema operativo.

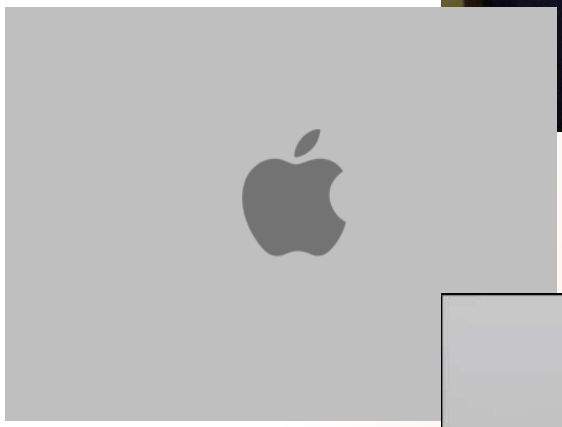
-Actualizar la BIOS (a.k.a *flashear)* es riesgoso por el simple hecho de que si falla, el comptuador al prenderse no tendrá nada cuerdo que cargar (y por ende tendrás un computador-ladrillo inservible a.k.a *brickeado*). No obstante hoy los computadores tienen mecanismos anti-fallas ante esas situaciones.

-Los Mac no usan BIOS, usan EFI, el hermano mayor del UEFI, cuya tarea es cargar el MacOSX. Todo eso ocurre antes que aparezca el logo de la manzanita.

-Incluso las consolas también tienen una especie de “BIOS” (No confundir con sistema operativo). Aunque en la mayoría son pantallas en negro que pasan desapercibidas, en otras consolas son animaciones bonitas. La mas recordada de todas es la de PlayStation 1.







Entonces tu CPU siempre ejecuta instrucciones

- Nunca para de ejecutar instrucciones. Nunca
- De hecho, el único punto donde se termina de ejecutar instrucciones es cuando el compu se apaga.
- Entonces… cuando el compu no hace nada, ¿que rayos hace?

NOP

- Este es la primera instrucción que aprenderemos.
- Si, como puedes imaginar esta instrucción hace **¡nada!**
- ¿Y para que rayos quisiera yo una instrucción que haga nada?
- Cuando el compu no carga… ¿que rayos quieres que éste ejecute? Es útil que haya algo que diga “no hagas nada”.
- El profe siempre termina sus programas con 3 o mas NOP; también puedes usarlo como separadores para dejar tus códigos bonitos, *imagina las posibilidades* (?)

¿Y asignar ~~varia…~~registros, como seria?

- Esto antes lo hacías así:

```
a = 3;
b = a;
```

- Esto era trivial en C/Java/PHP/IngresaTuLenguajePreferidoAqui.
  - ¿Y como se haría en Assembler?

MOV

- Con la instrucción de asignación/direccionamiento: **MOV**
- El nombre lo dice: ***MOV****e that value into that register.* (en realidad eso es chamullo, pero me captan la idea).
- Esto sirve para asignar un valor a un registro especifico
- Se puede usar de muchas, muchisimas formas, ya las veremos…

MOV AX,FFFF

Equivale a lo que tu conoces como `ax= 0xffff;`

Direccionamiento de registro

MOV

- En español chilensis, asignarle un registro a un registro.
- Es como si dijieras `ax = bx;`
- Recuerda, eso hace una copia, no una referencia ☺ Asi que olvídate por ahora de esa tontera de punteros y blablablá!

MOV AX,BX

Si BX=1234, luego de ejecutar esta instrucción verás que AX también tendrá 1234 ☺

Direccionamiento inmediato

MOV

- En español chilensis, asignarle un valor a un registro.
- Es como si digieras `ax = 4;`
- No en todos los registros puedes hacer este tipo. Para DS y ES por ejemplo, ¡no puedes! (Pero puedes asignarle a BX un valor y después decir MOV DS,BX ☺)

MOV BL,$78

Si BX=1234, luego de esta instrucción BX será 1278. ¿Porque?
Recuerda las PPT anteriores, ¡BL es la parte baja de BX!

Direccionamiento directo

MOV

MOV [1234],AX

- Este es mas latero. Aquí aprenderemos ya a escribir a la memoria.
- Primero, como puedes ver, estamos asignando lo que tenga AX a … **¿[1234]?** ¿Que rayos es eso?
- Abróchate los cinturones, **[1234]** quiere decir **DS:1234**

  (¿Qué se fumaron los creadores de assembler al suponer que ese corchete significa eso?)

MOV [1234],AX -> MOV DS:1234, AX

- En español chilensis, estamos escribiendo lo que tenga AX a la dirección de memoria **DS:1234.**

MOV [1234],AX

- ¡Problemas! Primero para poder seguir, necesito que hayas entendido las PPT anteriores porque aplicaremos ahora varias cosas de la materia.
- **PASO 1:** Antes que empieces, ¿a donde cresta queremos escribir?
- Casi siempre escribir en la **8000:0000** y la **9000:0000** son lugares relativamente seguros para escribir nuestras propias cosas (¡el resto esta todo ocupado por la BIOS, la memoria de video, y otros cachibaches!)
- ¿Escribamos entonces en la **81234**?
- **PASO 2:** Transformar **81234** al otro formato ese que aprendimos de memoria.
- **81234 -> 8000:1234.** Porque si juntamos esos 2 queda **81234**. (Si no entendiste, anda a la PPT 46-47 y vuelve a repasar eso).

MOV [1234],AX

- **PASO 3:** Preparar el direccionamiento.
- ¿Habíamos dicho que **MOV [1234],AX** significaba **MOV DS:1234, AX**, no?
- Y si queremos escribir en **8000:1234**, tenemos que hacer **DS=8000**.

¿**MOV DS, 8000**? **¡NO!**

(¿Que acabo de decir en la PPT 56? ¡Lee bien! No puedes hacer eso).

- Hay que asignar bien primero ese valor en otro registro (en este caso usaremos **BX**) y después asignarlo a **DS**.

MOV BX,8000, y luego **MOV DS,BX**. (¡Ahí si!)

- Y ahora recién con **8000** en **DS** podemos decir MOV [**1234**],**AX** para escribir lo que tenga **AX** en **8000**:**1234**.
- (Son mas pasos que la cresta, que lata ☹).

MOV [1234],AX

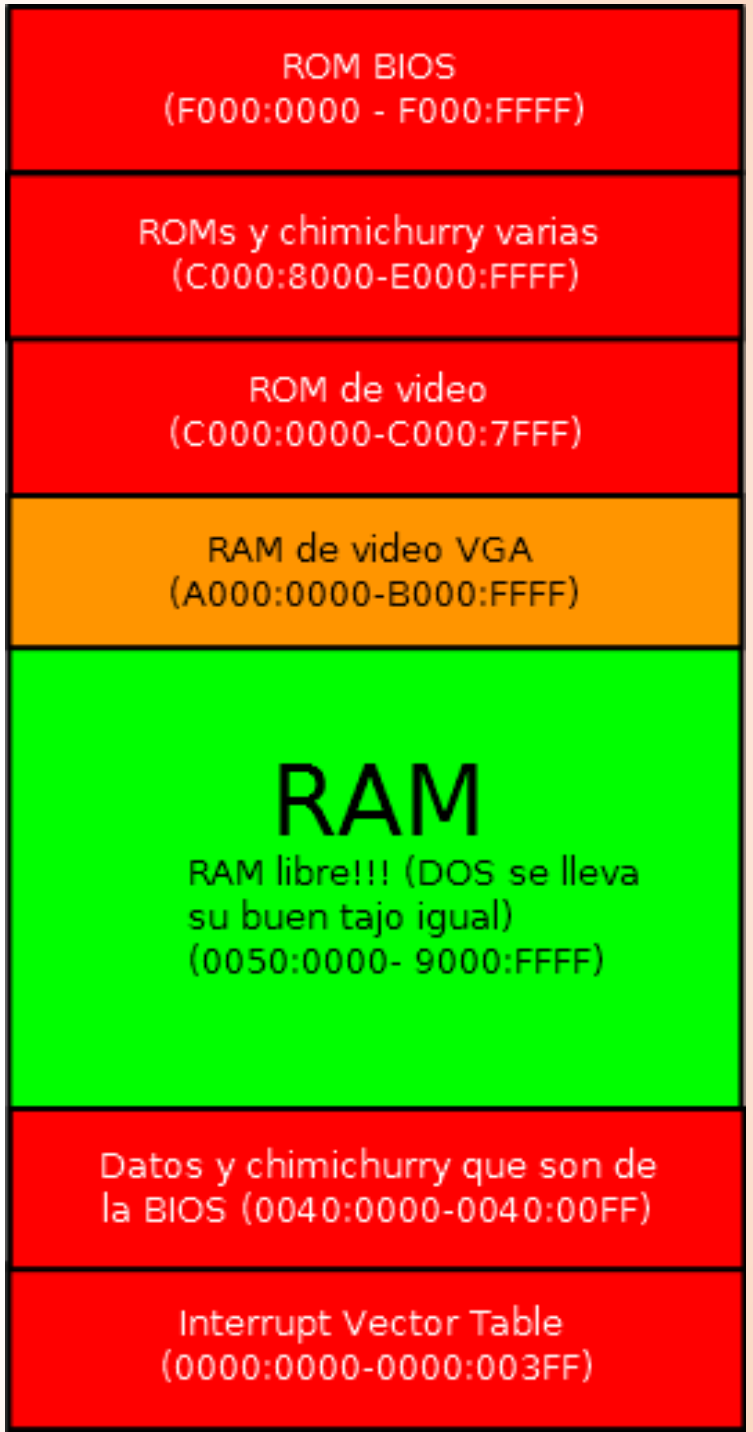
- **EN RESUMEN:** Para escribir AX en la 8000:1234

```
mov bx,8000
mov ds,bx
mov [1234],ax
```

Anexo: ¿Que espacios de mi 1 diminuto mega de memoria puedo usar?

- En modo real hemos dicho que habían 1 MB de RAM.
- Pero no podemos usar ese mega
- He aquí lo que es escribible y lo que es solo lectura.
- Igual hazme caso, la **8000:0000** y la **9000:0000** son ideales ☺

Dato rosa:
Esto lo menciona el profe, pero no aparece en las PPT. Es importante que tengas esto en mente porque puede parecer hasta en ejercicios de prueba



Direccionamiento indirecto por registro

MOV

MOV AX, [BX]

- Ahora, puedes hacer lo mismo también al revés (o sea, en vez de escribir, leer)… y con registros *Why not?*
- Si si si… aplicas las mismas confusas reglas que el anterior.

```
mov bx,1234
mov cx,8000
mov ds,cx
mov ax,[bx]
```

MOV

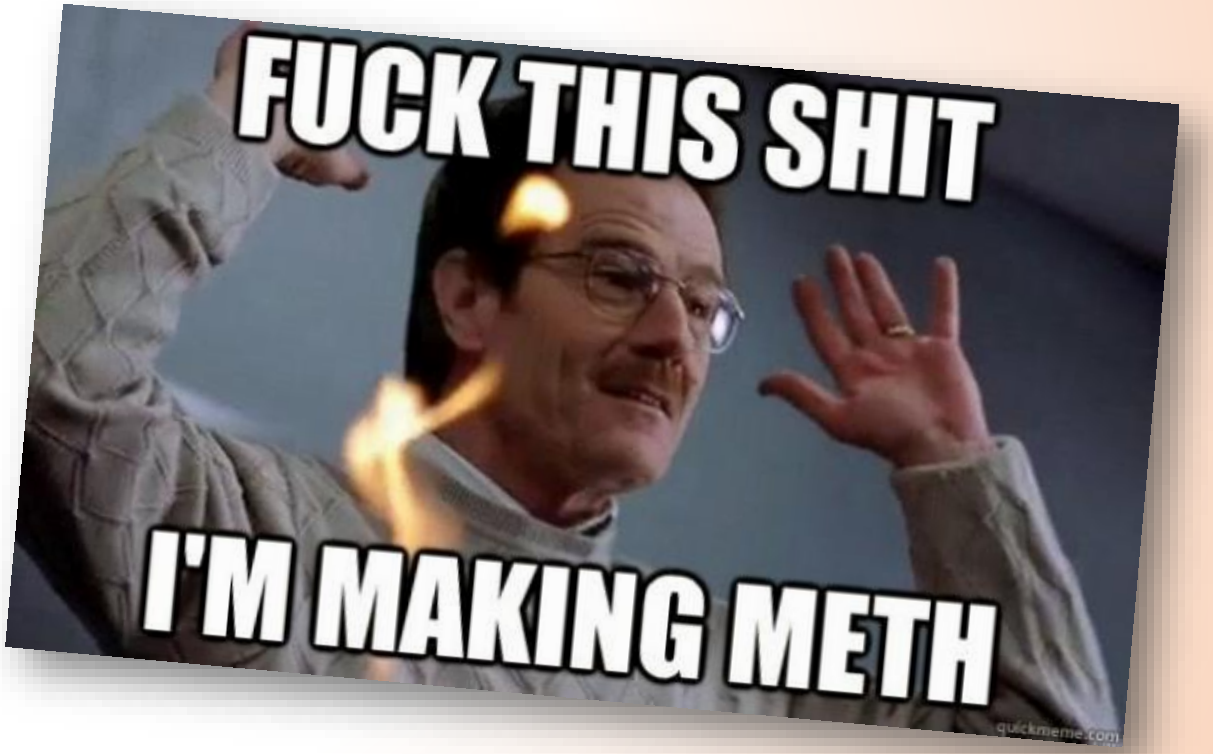
Y mas direccionamientos ultra bizarros (que quizá algún remoto día necesitarás… quizá)

- Direccionamiento base mas índice

  MOV [BX+DI],CL
- Direccionamiento relativo al registro

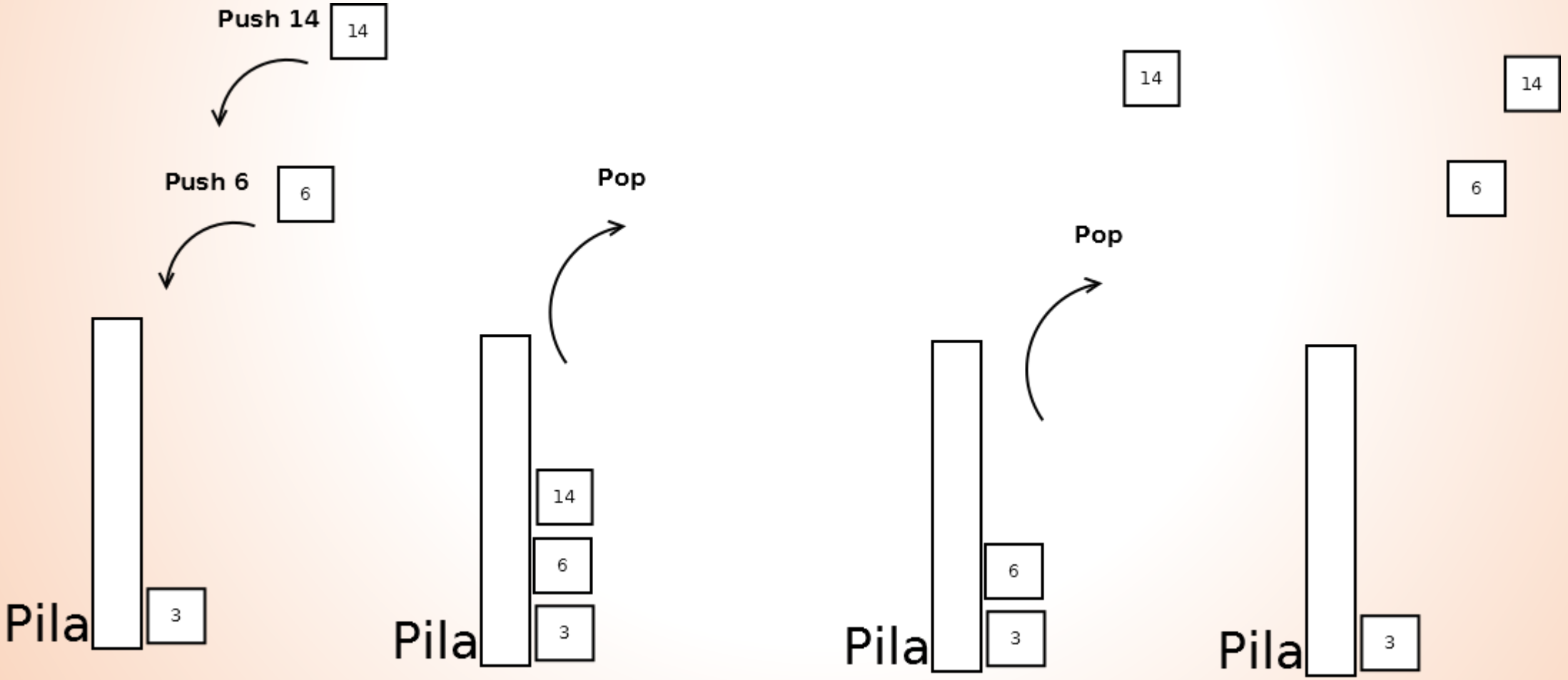
  MOV AX,[BX+4]
- Direccionamiento relativo base mas índice

  MOV AX, [BX+DI+4]



Tu PC tiene una pila.

- ¿Recuerdas estructura de datos? ¿Específicamente las pilas?
- Si, esas estructuras de datos que solo podías hacer 2 cosas: PUSH y POP.



- Bueno… tu compu tiene una pila… pero SOLO 1.
- (¿Whaaaat? ¿Y porque solo 1?)
- En esa pila no solo se guarda datos tuyos, sino también punteros de direcciones, estado de funciones, valores guardados, estados del sistema operativo, blablablá.
- Esta pila se ubica en nada mas ni nada menos que en **SS:SP** (¡es por eso que tampoco debes modificar estos registros! Si los modificas pueda que el PC siga funcionando bien… por un rato y después cuando use la pila saque solo valores erróneos).
- Esta pila es bastante delicada. No es como en C que podías comprobar si estaba vacía. Aquí no puedes. Si te "pasas de largo" leerás/escribirás datos en la RAM más allá de lo asignado a la pila … y quedará la crema, asegurado ☺
- Así también al revés: No si no controlas cuantos datos guardas allí, podrás excederte del espacio asignado a la pila y **¡KABOOOOM!** Esto es mundialmente requetearchiultraconocido como el error "**`STACK OVERFLOW`**" que seguramente te pasó mas de una vez en progra … ¿o no?

¿Y te imaginarás como usar la pila?

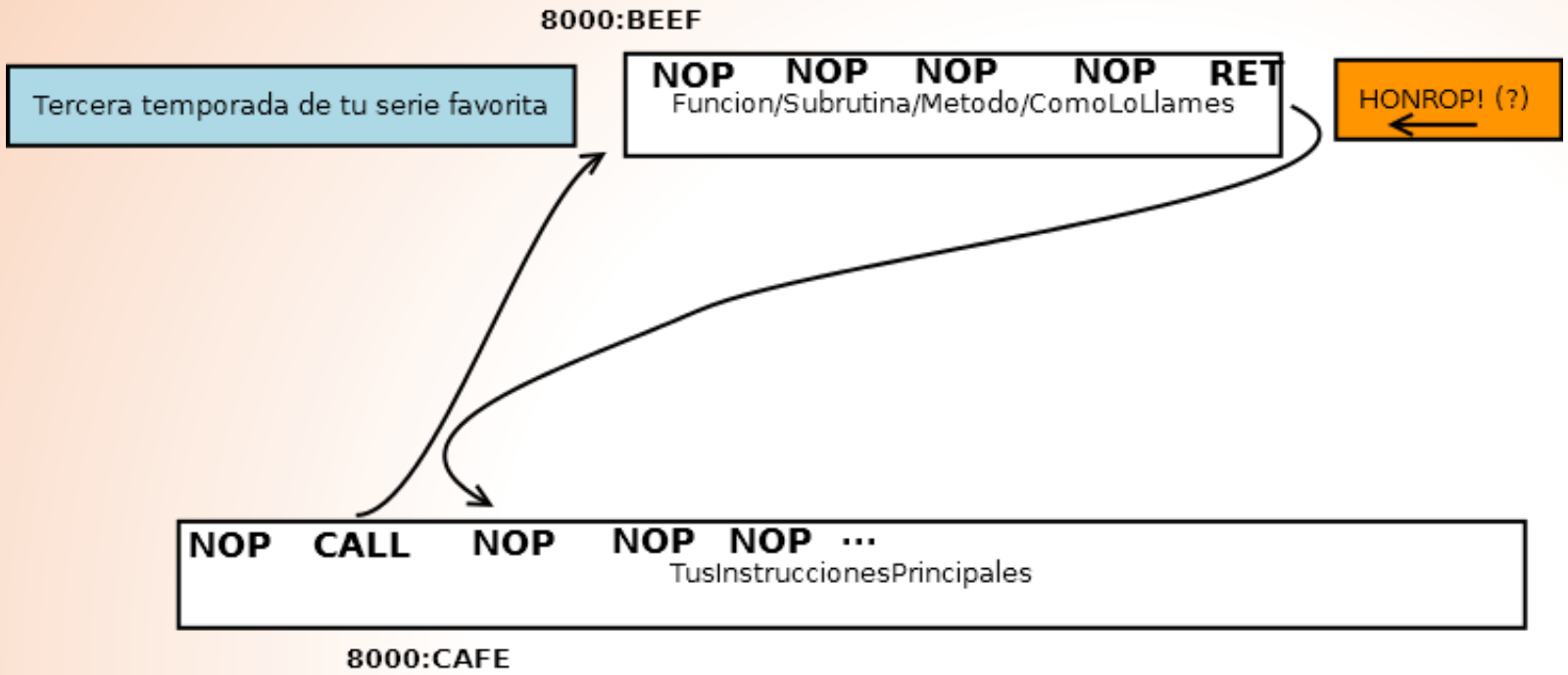
PUSH AX

POP BX

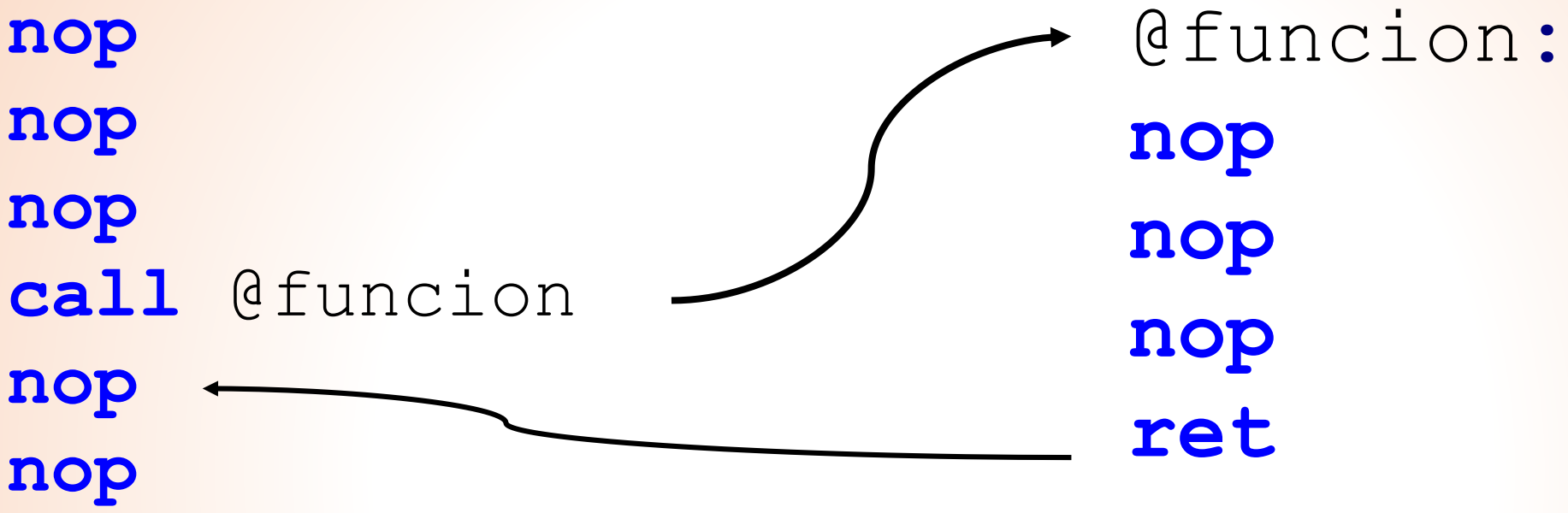
- Ninguna ciencia. Tal como lo ves.
- PUSH AX escribe lo que tiene AX en la pila… Y POP BX saca lo que haya arriba en la pila y lo escribe en BX.
- Ojo, que aquí no puedes hacer PUSH/POP con registros divididos (AH,AL)… Tienes que hacerlo con el registro entero. (AX).

¡La pila también sirve para hacer funciones/subrutinas/métodos/como lo llames!

- La gracia de las funciones, si bien te acuerdas es simplificarte el trabajo escribiendo ~~codig~~ instrucciones repetitivas.
- ¿En progra recuerdas que las funciones siempre estaban aparte del main? Aquí pasa lo mismo, las instrucciones están en otro lado que las instrucciones principales.
- Estas funciones se llaman, hacen su trabajo y después retornan hacia donde estaban, ¿no es cierto? … Pero Assembler y la CPU son un tanto “tontos”. Solo consumen y ejecutan instrucciones.
- Para implementar estas funciones/subrutinas/métodos/comolollames usaremos las instrucciones **CALL** y **RET**. (Sip, una para llamar y el otro para retornar).



- Aquí ves que tenemos unas instrucciones en la dirección **8000:CAFE** (xD) y una función en **8000:BEEF** (xDD).
- Cuando se ejecuta **CALL** la CPU se teletransporta (o sea, cambia **CS:IP**) a **8000:BEEF** (mejor que CALL lo modifique, no tu), y sigue allí.
- Cuando encuentra un **RET**, tu CPU volverá a teletransportarse a 8000:CAFÉ y seguirá como si nada.
- ¿Y como diantes sabe que tenia que volver a **8000:CAFE** y no a **8000:C02A** o cualquier otra cosa?
- La respuesta es: Porque cuando se hizo **CALL** guardo **8000:CAFÉ** en la pila, ¡waaau!



- Lo bueno es que en editores como turboPascal en vez de hacer **CALL BEEF** (u otra dirección de memoria), puedes poner etiquetas a las direcciones y así hacer todo muuuuuucho mas fácil! (Como las funciones en progra).
- Si no pones **RET** el programa funcionará igual, pero en vez de retornar hacia donde estabas (**8000**:**CAFE**), seguirá de largo a lo que esta abajo, y así perderás el control y tu CPU empezará a ejecutar instrucciones random en un rincón bizarro de por ahí (¡o sea, le dará el ataque epiléptico que explicaba hace varias PPT atrás!)

Aun así quieres usar **CALL** a la antigua?

```
call CAFE
call 8000:CAFE
```

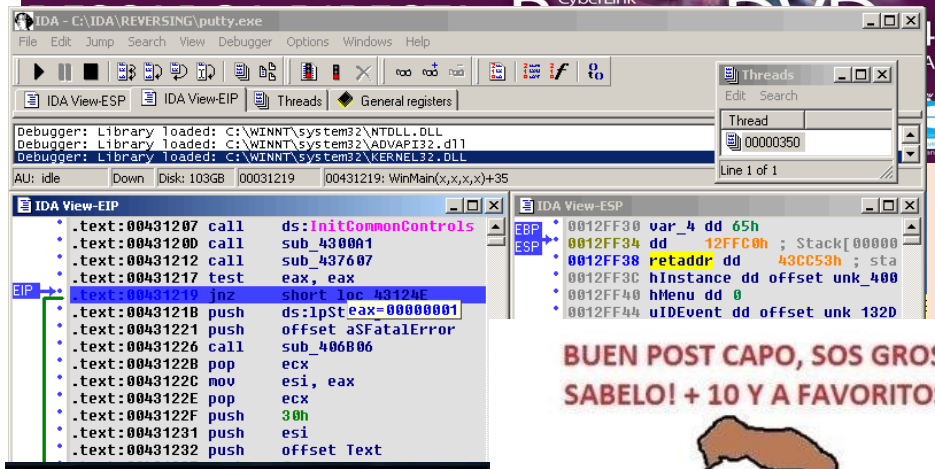
CALL
RET

- El primero irá las instrucciones que están en **CS:CAFE**
- El segundo no toma en cuenta **CS**, va a **8000:CAFE** o donde quieras. ¿En que influye esto? Guardas mas valores en la pila, solo eso.

Dato rosa

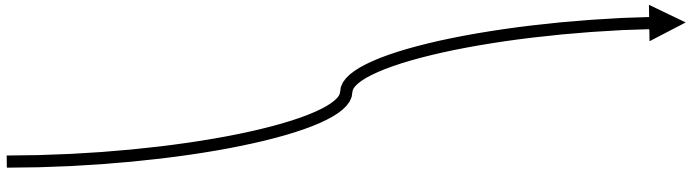
- Ya sabes que tu computador funciona a puras instrucciones en Assembler. Todos tus programas se pueden “desensamblar” para leerlos en Assembler y volverlos a ensamblar (si tocas algo, 80% que tu programa explota).
- ¡Las protecciones anti-copia de programas y juegos no son la excepción! ¡Todo lo puedes transformar en assembler!
- Ahora, ¿que pasa si en esos mecanismos anti-copia cambiamos todos los **CALL** a otros lados y las instrucciones a puros **NOP** y los volvemos a reensamblar?
- *Y es así queridos niños como se hacen los cracks!* ☺

(Y los tipos se aburren tanto que hasta hacen musiquitas y animaciones locas… si les interesa, se usan programas como IDAPro).



¿Y si no quiero retornar y solo quiero saltar a otro lado de la memoria donde leer ~~código~~ instrucciones?

```
nop
jmp @otra_parte
nop ;nunca llegare aqui
```



```
@otra_parte:
nop
nop
```

JMP

- Pues usa **JMP**
- A diferencia de **CALL**, no puedes usar **RET**, no guarda valores en la pila, ¡nada!
- De hecho, el nombre lo dice: Jump! Saltas de un lado hacia otro sin retorno!
- (Bueno, si quieres retornar haz un JMP de vuelta, corta).

¡Todo sobre FLAGS!

- ¡A todos (incluidos computadores) nos pasan cosas!
- Hay sucesos en nuestras vidas que merecen ser notificados y lanzar avisos de ello.
- Seamos simples. Estos “sucesos” pueden *suceder* o *no suceder*.

¡Tu auto se esta friendo!

¡La pagina no existe!

¡Mexicanotes exhibicionistas próximos 3 km!

¡Estos avisos en tu CPU, se llaman **FLAGS**!

C P A Z T S I D O

- Bueno… al menos veremos estos.
- Al hacer ciertas operaciones algunos de éstos pueden prenderse o apagarse dependiendo de la situación y que es lo que pasa.
- En tu procesador el día de hoy hay muchos mas, incluso algunos que ni se usan, pero eso ya es harina de otro costal.
- Veamos de que trata cada uno…

Carry

- Acarreo en español.
- Generalmente se activa cuando “te pasas de largo”… u errores en general.
- Ej: **`FFFF + 1 = 0000 (Carry!)`**

Parity

- Paridad en español.
- Simple. Se apaga cuando un numero es impar y se prende cuando hay un numero par.

AuxCarry

- Carry auxiliar
- Bit 3 y 4 del resultado de una suma y resta (o sea, bastante especifico… dudo que lo uses).

Zero

- Cero.
- Se prende cuando algo da 0. Obviamente si no es cero se apaga.

Trap

- Trampa
- Lo usan mas que nada los debugeadores.

Sign

- Signo
- Si esta activado, es negativo. De lo contrario positivo.

Interrupt

- Interrupción
- Representa el estado del terminal de Interrupciones (ya veremos eso, relajados).

Direction

- Dirección
- Representa la dirección en que se leen flujos de datos en ciertas operaciones que hace la CPU.

Overflow

- Desborde
- Se prende cuando un resultado excede la capacidad de la maquina.
- Creanme, es muchísimo mas practico usar Carry ☺

Créanme que a lo mas usaremos intensivamente estos flags solamente.

C Z

- Carry para cuando haya que comprobar si nos pasamos de FFFF a 0.
- Zero cuando … ¡cuando un resultado de 0! O para interpretar ciertos errores.

¿Y como leemos los flags?

¡Las instrucciones que veremos ahora usan flags!

ADD

- ADD o suma. Como el nombre dice, esto suma 2 valores y guarda el resultado en el primero.

ADD BX,AX

ADD BX,1234

Equivale a lo que tu conoces como `bx= bx+ax;`

NOTA: Aquí ya se usan los flags. Si el resultado es 0 (really?) Se activará Z , y si el resultado es par, se activará S

- Oh! Y que pasa si hago esto?

```
mov bx, FFFF
add bx, 2
```

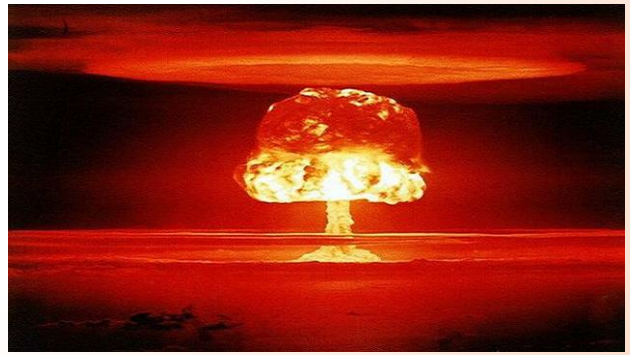
Opción A: BX será

10001

Opción B: BX será

0001

Opción C: CX s…



ئاڵاهو ئاکبار

(ALLAHU AKBAR)

- Definitivamente será la opción B. (Al menos que tu compu tenga dinamita, ahí considera la opción C).
- Y se activara el Carry!

BX será

0001

~~(JOHN CEN...~~
(CARRY!)

Un uso útil de ~~John C...~~Carry, y mas instrucciones

ADC

- Suma con Carry. Igual que ADD, solo que si Carry esta prendido suma 1 numero mas ☺

```
mov ax,00ff
add al,1
adc ah,0 ;quedará AX=0100
```

INC

- Un sumador expreso, que solo suma 1 al destino **PERO CUIDADO: No hace Carry.** Si se pasa de largo, no avisará.

```
inc ax
```

- Lo mismo sucede con la resta. SUB, y DEC son para resta, uno puede hacer Carry y el otro NO. Solo preocúpate de ver bien el asunto de los signos si vas a trabajar con ellos (*mira "la biblia" para mas info.)*

SUB DEC

```
mov ax,1234
mov bx,00ff
sub ax,bx ;queda ax=1135
```

```
dec ax
```

- Hay una variante especial de JMP (si no lo recuerdas, mira las PPT anteriores) que solo se ejecuta si la ultima instrucción dio Carry.

JC - JNC

Esto es intutivo. El primero es para cuando hay carry, y el segundo es para cuando no hay carry.

```
mov bx,0fff
add bx,1
jc @lado ;se irá a @lado porque hay carry
```

- Lo mismo existe para el flag Zero ☺

JZ -JNZ

CMP

- ¡Esta es una instrucción muy poderosa!
- **Compara** 2 registros, 2 números, 1 registro con 1número, que se yo, y dependiendo del resultado activa ciertos flags.

CMP BX,AX

CMP BX,1234

- Al igual que **JC** y **JZ**, hay muchas variantes de **JMP** que aprovechan los flags arrojados por el resultado de **CMP**.

JE	**Jump Equal ==**
JNE	**Jump Not Equal !=**
JA	**Jump Above >**
JAE	**Jump Above or Equal >=**
JNA	**Jump not Above !>**
JB	**Jump Below <**
JBE	**Jump Below or Equal <=**
JNB	**Jump not Below. !<**

Receta:
Como hacer un if en Assembler

(Comprobar si AX=33 es mayor a 0)

```
MOV AX,33
CMP AX,0
JA @SI_CUMPLE
JMP @NO_CUMPLE

@SI_CUMPLE:
NOP
JMP @FIN

@NO_CUMPLE:
NOP
JMP @FIN

@FIN:
NOP
NOP
```

Ingredientes:

- **CMP** (1x)
- **JMP** (3x)
- **MOV** (para el ejemplo)
- **JA** (o la variante que quieras)
- **NOP** (a tu gusto)

Bom pas le carry !

Multiplicar/Dividir?

- Un “caxo” =/
- Pero trataré de explicarlo con manzanitas.
- Se puede hacer de muchas formas (vea la biblia).
- Pero si estas en apuros, esta forma al menos funciona!
- Deja el multiplicando en **AX**, y el multiplicador en **BX**.
- Ahora realiza esta instrucción.

```
mov ax,2
mov bx,2
mul bx
nop
;ax=4, dx=0 porque
el numero es chico
```

```
mul bx
```

- El resultado quedara en **DXAX**. ¿Cómo así?
- Exacto. El numero que sale de esto podría ser tan grande que necesitamos 2 registros para guardarlo entero. Wau.

1234x1000=01234000

DX=0123 y AX=4000

- Ahora para dividir es al revés. Coloca el dividendo en **DXAX.**
- Ahora pon el divisor en **BX**, y lanza esta instrucción.

```
div bx
```

- El resultado quedará en **AX**, y el resto en **DX**.

0006/0002=3

AX=0003 y DX=0000

0005/0002=2

AX=0002 y DX=0001

```
mov dx,0  ; el numero es chico
mov ax,5  ;dividendo
mov bx,2  ;divisor
div bx    ;resultado en AX, DX el resto
nop
```

AND / OR / XOR / NOT

- No les recordare que hacen estas cosas: todos ya sufrimos en electrónica digital/Fundamentos de electrónica ☹
- Funcionan si de una forma bien particular. Le pasamos 2 parámetros… o sea números/registros.
- El resultado será cada bit del el primer parámetro con (operador lógico aquí) el segundo parámetro.

```
mov ax,10
mov bx,2F
and ax,bx
```

```
AX: 0000000000001010
BX: 0000000000101111   OR
----------------------------
AX: 0000000000101111
```

(**AX** tomará el valor de 2F)

Dato rosa:

Cuando el profe quiere dejar un registro en 0, lo hace de esta forma: **XOR AX,AX**
¡Hazlo de esta forma en la prueba! Al profe por alguna razón no le gusta que sus estudiantes hagan **MOV AX,0** para dejar un registro en 0 porque dice que no es eficiente… ¿Hello?

XOR AX,AX

```
AX: 0000000000001010
AX: 0000000000001010   XOR
----------------------------
AX: 0000000000000000
```

Dato rosa:

Cuando quieras manipular tu mismo un flag, puedes forzar una suma para que haga **Carry** o **Zero** ¿no? Aunque también hay formas más directas como usar **TEST** que es una comparación lógica AND pero que entrega el resultado en el flag **Z, sin modificar los registros participantes**.

TEST AX,AX

```
AX: 0000000000001010
AX: 0000000000001010   AND
----------------------------
AX: 0000000000001010 <- NO CAMBIA
```

(ZERO!)

¿Cosas bizarras con registros en binarios?

(Es poco probable que los uses en una prueba, pero uno nuuuunca sabe)

SAL /SAR

- Corre n bits a la izquierda/derecha. El bit que entra/sale lo manda al flag Carry! `sal bl,1`

SHL /SHR

- Lo mismo, pero no trabaja con signos. `shl bl,1`

ROL /ROR

- Rota n dígitos binarios a la izquierda o derecha. (Los del borde izquierdo aparecen al principio del derecho, por ejemplo.).

```
rol bl,1
```

Para hacer rotaciones usando dígitos del Carry, querrás usar **RCL, RCR**.

Buenas y malas noticias.

¡Con esto ya estas listo/a para hacer ejercicios tipo prueba!

(En la Biblia hay unos rebuenos con solución!)

Las Interrupciones

```
nop
nop
...
```

Todo iba muy bien…

…¡¡¡Cuando de repente!!!…


```
int 10
```

… y finalmente

```
nop
nop
...
```

- El nombre interrupción es muy descriptivo:

*“Algo que **interrumpe** la ejecución de alguna cosa”.*

- En el computador pueden pasar muchas cosas que requieran interrumpir tu proceso.
  - Algunas interrupciones se disparan solas, otras las invocas tú.

- Las interrupciones pueden hacer muchas cosas. Todo depende de que interrupción lanzas.
- Cada interrupción puede hacer muchas cosas bien distintas una de las otras. Es como una “biblioteca de funcionalidades”.
- Para lanzar una interrupción especifica, usa la instrucción indicada con el numero de la interrupción, indicando antes también que procedimiento quieres ejecutar en tal.

```
mov ah,10
int 10
```

Ejemplo, para lanzar una interrupción cuya función es escribir un carácter en pantalla.

Esa interrupción es la **10/AH= 0A.**

(Si pones AH = 00, 48, FF o que se yo, harás cualquier otra cosa que haga la interrupción 10, menos escribir).

¡Hay 256 Interrupciones!

(¡Cada una con funcionalidades distintas dentro! Unas maravillosas, otras… que nadie conoce…)

00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
20	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D	2E	2F
30	31	32	33	34	35	36	37	38	39	3A	3B	3C	3D	3E	3F
40	41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F
50	51	52	53	54	55	56	57	58	59	5A	5B	5C	5D	5E	5F
60	61	62	63	64	65	66	67	68	69	6A	6B	6C	6D	6E	6F
70	71	72	73	74	75	76	77	78	79	7A	7B	7C	7D	7E	7F
80	81	82	83	84	85	86	87	88	89	8A	8B	8C	8D	8E	8F
90	91	92	93	94	95	96	97	98	99	9A	9B	9C	9D	9E	9F
A0	A1	A2	A3	A4	A5	A6	A7	A8	A9	AA	AB	AC	AD	AE	AF
B0	B1	B2	B3	B4	B5	B6	B7	B8	B9	BA	BB	BC	BD	BE	BF
C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	CA	CB	CC	CD	CE	CF
D0	D1	D2	D3	D4	D5	D6	D7	D8	D9	DA	DB	DC	DD	DE	DF
E0	E1	E2	E3	E4	E5	E6	E7	E8	E9	EA	EB	EC	ED	EE	EF
F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	FA	FB	FC	FD	FE	FF

Mira la lista completa en http://www.ctyme.com/intr/int.htm

Dato rosa:
No hagas scroll para abajo en la pagina si tus papás/señora católica/profesor están cerca. Allí está la “interrupción” feliz N° 257 ;)

- La BIOS carga un set predeterminado de interrupciones para que puedas hacer cosas mas interesante en el PC (mas interesante que sumar y multiplicar, claro). Lo malo es:
  - **1. Solo funciona en modo real. En modo protegido, estas interrupciones no funcionan… (¿Alguien dijo que necesitamos drivers? ☺)**
  - **2. ¡Hay demasiado por donde elegir! por ejemplo…**

Int 16/AH=00h – KEYBOARD – GET KEYSTROKE

- Esta interrupción lee el teclado.

Int 19 – SYSTEM – BOOTSTRAP LOADER

- Esta interrupción hace que tu PC se resetee.

Int C7 – APL*PLUS/PC – ???

- Esta interrupción… eeeeehhhhhhh… *¡Psst! ¿Alguien sabe para que sirve? No suena para nada bonito* ☹

(Seguramente el único en el planeta que sepa que es lo que hace esta interrupción esté en un asilo de ancianos con Alzheimer. ☹)

- No obstante, en la asignatura usaremos solo algunas interrupciones.
- Estas las encontrarás en la **Lista de Interrupciones comúnmente usadas**.
- Son muchas, no las veremos en esta PPT por razones prácticas.
- Para que te hagas una idea, las interrupciones no las necesitarás mucho en la prueba, pero si para la tarea!
- Está en el aula, o pregúntale al ayudante! ☺

Lista de interrupciones comúnmente usadas en procesadores Intel 8086
(Compatible también con cualquier IBM PC, x86 y AMD).

INT 10h / AH = 0 - set video mode.

input:
AL = desired video mode.

these video modes are supported:

00h - text mode. 40x25. 16 colors. 8 pages.

03h - text mode. 80x25. 16 colors. 8 pages.

13h - graphical mode. 40x25. 256 colors. 320x200 pixels. 1 page.

example:

```
mov al, 13h
mov ah, 0
int 10h
```

INT 10h / AH = 01h - set text-mode cursor shape.

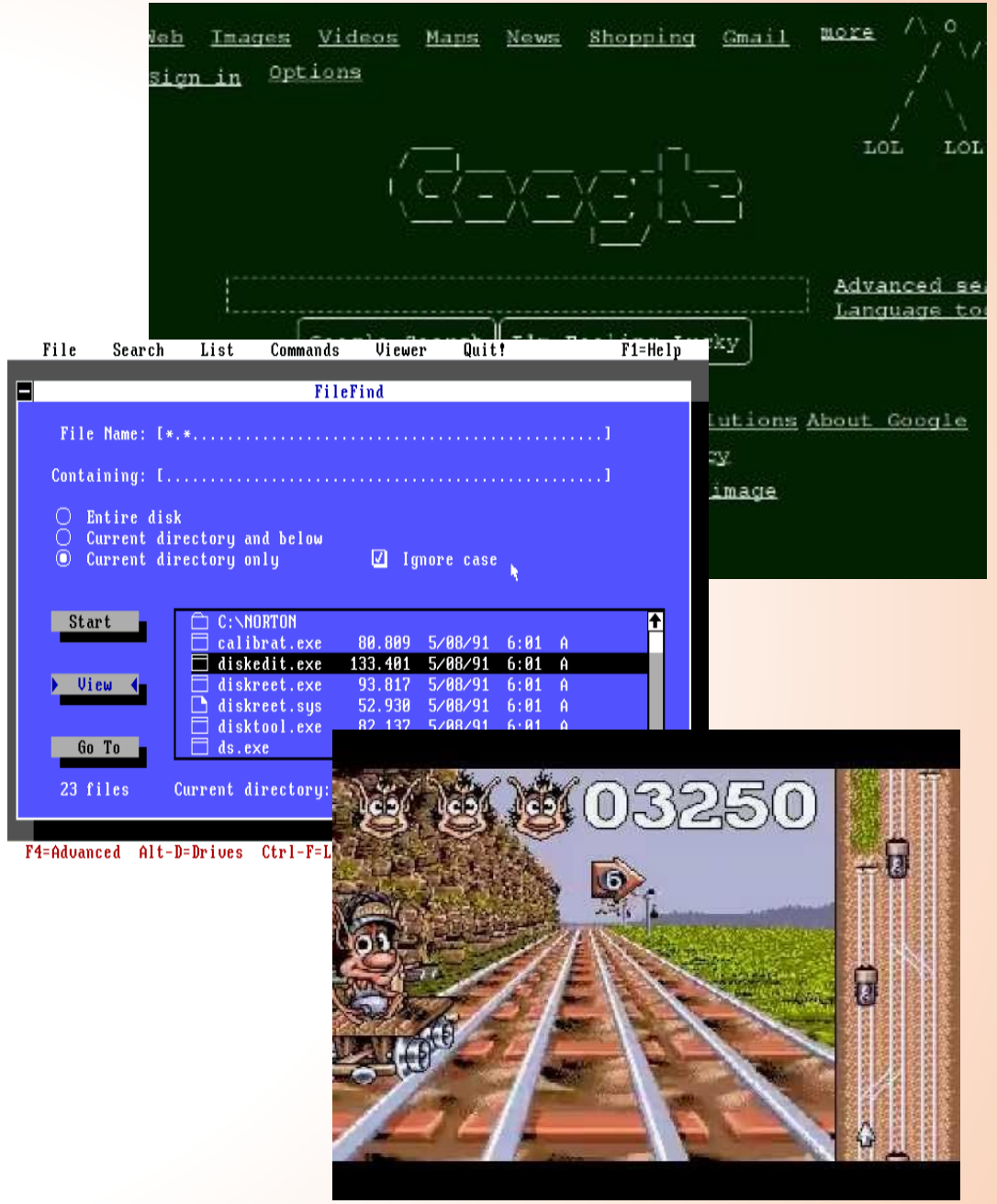
input:
CH = cursor start line (bits 0-4) and options (bits 5-7).
CL = bottom cursor line (bits 0-4).

when bit 5 of CH is set to **0**, the cursor is visible. when bit 5 is **1**, the cursor is not visible.

```
; hide blinking text cursor:
      mov ch, 32
      mov ah, 1
      int 10h

; show standard blinking text cursor:
      mov ch, 6
      mov cl, 7
      mov ah, 1
      int 10h

; show box-shaped blinking text cursor:
      mov ch, 0
      mov cl, 7
      mov ah, 1
      int 10h

;         note: some bioses required CL to be >=7,
;         otherwise wrong cursor shapes are displayed.
```

Algo sobre la pantalla

- El computador tiene muchas formas de mostrar las cosas en pantalla.
- Pero en el entorno que estamos ahora, estamos limitados a unos cuantos “modos”.
- El predeterminado es el **Modo Texto número 3** (80x25 bloques de caracteres, 16 colores, 8 páginas).
- Hay otros modos de texto que permiten ver el contenido mas grande, mas chico, mas feo.
- O inclusive modos gráficos, donde ya no hay texto, sino pixeles para dibujar en pantalla.
- Estos modos los cambias con interrupciones. (**INT 10h / AH = 0**)
- No complicaremos la existencia. Usaremos solo modo texto numero 3 para los ejercicios y la tarea.



(Hu hu hu hu Hugo! ….)

Algo sobre la pantalla

- Las paginas son “pantallas” extra que tiene tu computador.
- Lo que ves en tu pantalla está en una pagina, mientras que las demás se quedan en backstage.
- Por default, vemos en pantalla lo que está en la página 0.
- ¿Para que sirve? Imagínate que cargas algo pesado como una imagen en pantalla, y no quieres “*mostrar la hilacha*”. Entonces haces la carga “*en backstage*” mientras muestras otra cosa en pantalla.
- Por interrupciones (**INT 10h / AH = 05h**) podemos elegir que pagina ver cada vez.

Algo sobre el teclado.

- El teclado funciona con un buffer que guarda las teclas que presionas, para que así después el programa que tengas pueda consultar que rayos has presionado mientras el computador ha estado ocupado en negocios importantes.

- Pero lo que pasa en realidad el 99.9% de los casos que lo que presionas en el teclado, el compu lo procesa tan rápidamente que jamás notarás que pasó por el buffer. (Excepto cuando el compu tiene ultra-lag… ¿te ha pasado que se congela y cuando responde escribe todo a la vez?)

- Con la interrupción **INT 16h / AH = 00h** El compu sacará lo que tenga acumulado en el buffer. Y si no hay nada en el buffer pues parará el programa hasta que presiones algo xD.

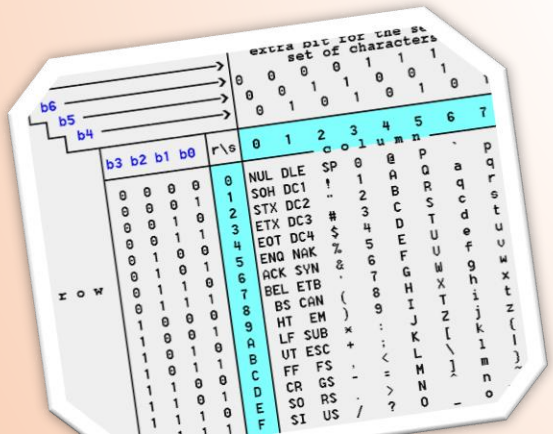
- Y como chequear si tengo algo en el buffer **(en español chilensis: si presiono una tecla sin parar el programa)?** pues usa: **INT 16h / AH = 01h** La interrupción te dirá si presionaste una tecla o no.

Algo sobre el teclado.

- Cuando lees que valor has leído del teclado te entregara 2 formatos:

Un carácter en ASCII

- Te sonará de ramos como programación.
- Obtendrás en un registro el carácter leído en código ASCII.



Un carácter en ScanCode

- No todo puedes expresarlo en ASCII. (La tecla F8, Shift, Pantallazo, Flecha izquierda, ENTER, blabla).
- Es otro set de códigos que indican que teclas has presionado.
- Es mas especifico.

Para la prueba puedes sobrevivir teniendo en cuenta estas interrupciones:

- Leer en teclado: (La tabla ASCII y de ScanCodes está al final de la PPT!)

INT 16h / AH = 00h

Esperará que presiones una tecla y retornará la tecla presionada en AH en formato ScanCode, y en AL en formato ASCII

- Mover el “cursor” para imprimir en el lugar correcto.

INT 10h / AH = 2

Pon en DH la fila, DL la columna y BH la pagina (si no sabes que es eso, pon 0).

- Imprimir un carácter por pantalla: (La tabla ASCII esta al final, el cursor avanzará solo).

INT 10h / AH = 0Eh

Pon en AL= Carácter a imprimir en ASCII

(Todas estas están en el documento de las interrupciones comúnmente usadas si quieres ver mas detalle de éstas).

Llegó la hora de decir adiós.

- Esta PPT se está terminando lamentablemente ☹
- Pero el curso no termina aquí… de hecho, ¿seguramente tendrás la prueba 1 encima, no?
- **¡Estudia!** En la biblia, en clases y en ayudantía ve ejercicios tipo prueba
- ¡Haz ejercicios! En la biblia hay varios ejercicios tipo por si te interesa.
- Para la segunda parte del curso, si bien ya no se trata de programar, sino de materia pura, ¡enfócate en la tarea!

Más material para seguir estudiando.

- La biblia de arquitectura (set completo):
http://elsemieni.net/arch/biblia_de_arquitectura.rar

- Kit DOSBox + Debug + TurboPascal:
http://elsemieni.net/arch/biblia_de_arquitectura.rar

- Set instrucciones básicas x86:
http://elsemieni.net/arch/SetDeInstruccionesBasicasX86.pdf

- Tabla ASCII (General)
http://elsemieni.net/arch/Tabla%20ASCII.pdf

- Tabla ASCII (IBM character set - DOS)
http://elsemieni.net/arch/vga_norm.gif

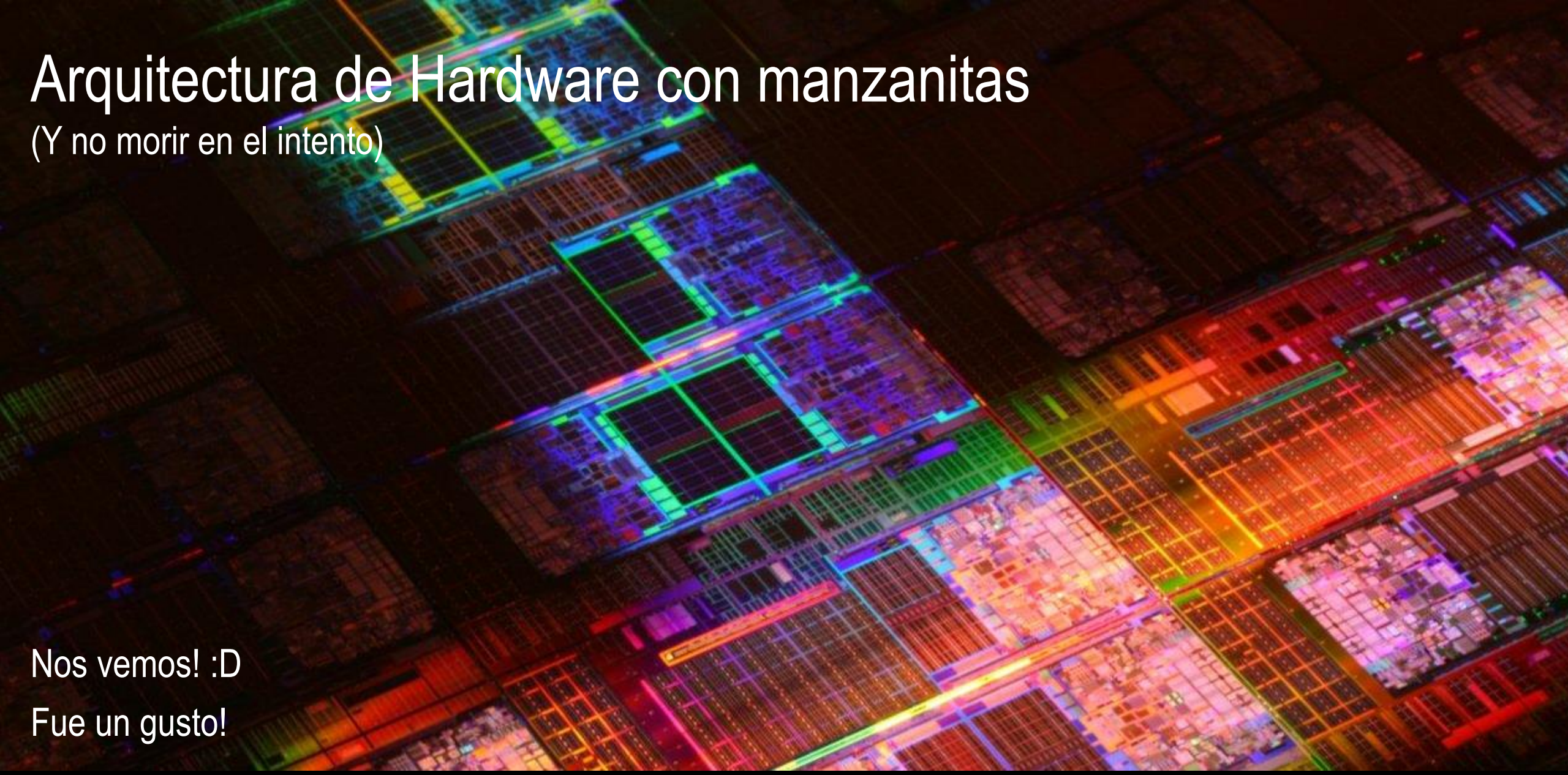
- Lista interrupciones comúnmente usadas:
http://elsemieni.net/arch/Lista%20interrupciones%20comunmente%20usadas.pdf

- BIOS Scancodes

http://elsemieni.net/arch/bios_scancodes.pdf

Arquitectura de Hardware con manzanitas

(Y no morir en el intento)



Nos vemos! :D

Fue un gusto!

Enzo Barbaguelatta D.

PPT de apoyo para los cursos de Arquitectura de Hardware de Ingeniería Civil Informática PUCV