Teórica 3

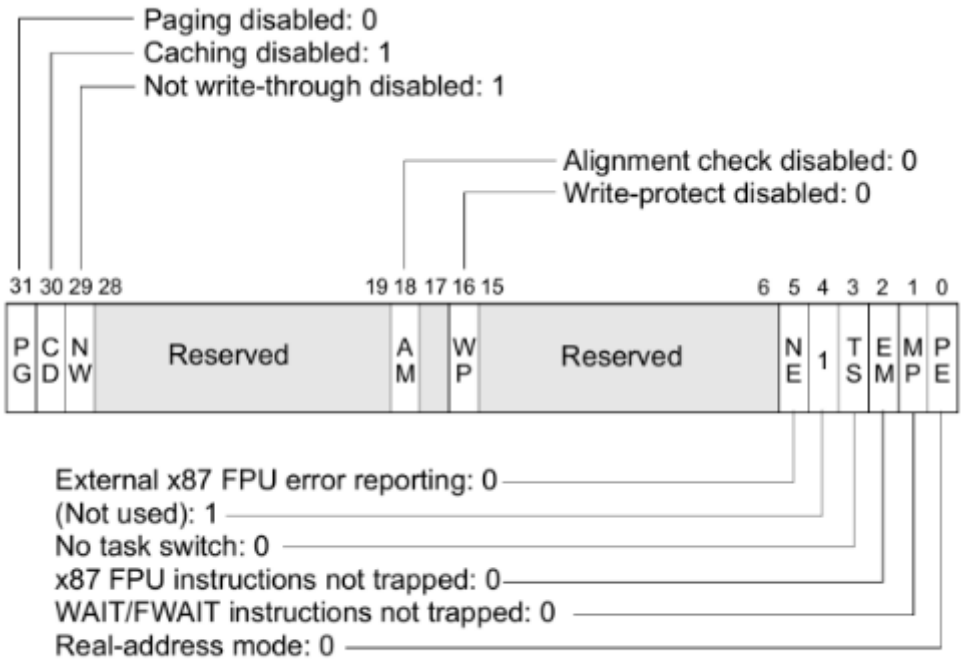
Bare Metal Programming

Built In Self Test

Test interno cuando se activa el pin **#RESET**

- Como resultado inicializa el registro *eax* en 0x0 solo si todos los tests pasaron OK. Si no está en 0 es porque hay un problema.
- El resto de los registros toman un valor determinado.
- Se invalidan las memorias cache internas.
- Se invalidan las LTB y se limpian los Branch Target Buffers.
- La diferencia con enviar una señal #INIT es que en este caso, se mantienen intactos los valores de los Registros de la FPU, los MSR's y los MTRR's, y los caches internos.
- De acuerdo a la familia de procesador implementa con variantes la inicialización de los demas procesadores presentes en caso de un sistema SMP.

Valores iniciales de interés



Registro CR0. Valor inicial

Register	Power up	Reset	INIT
EFLAGS[1]	00000002H	00000002H	00000002H
EIP	0000FFF0H	0000FFF0H	0000FFF0H
CR0	60000010H[2]	60000010H[2]	60000010H[2]
CR2, CR3, CR4	00000000H	00000000H	00000000H
CS	Selector = F000H Base = FFFF0000H Limit = FFFFH AR = Present, R/W, Accessed	Selector = F000H Base = FFFF0000H Limit = FFFFH AR = Present, R/W, Accessed	Selector = F000H Base = FFFF0000H Limit = FFFFH AR = Present, R/W, Accessed
SS, DS, ES, FS, GS	Selector = 0000H Base = 00000000H Limit = FFFFH AR = Present, R/W, Accessed	Selector = 0000H Base = 00000000H Limit = FFFFH AR = Present, R/W, Accessed	Selector = 0000H Base = 00000000H Limit = FFFFH AR = Present, R/W, Accessed
EDX	000n06xxH[3]	000n06xxH[3]	000n06xxH[3]
EAX	0[4]	0[4]	0[4]
EBX, ECX, ESI, EDI, EBP, ESP	00000000H	00000000H	00000000H

Valor iniciales de los registros corrientes[3]

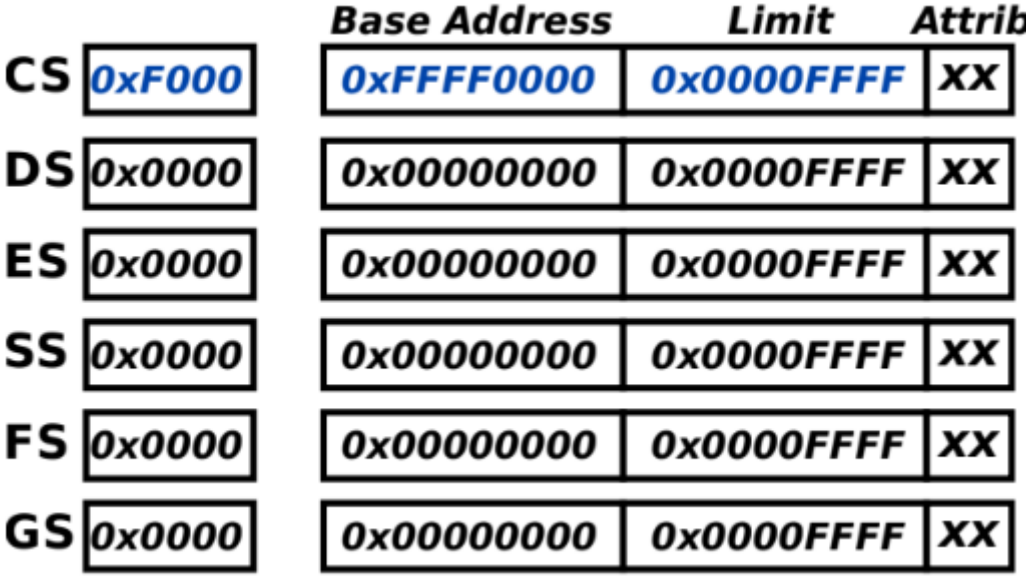
Del slide anterior nos quedamos con los registros CS:IP, cuyos valores iniciales son respectivamente 0xF000:0xFFF0.

Reset vector

- Al mismo tiempo el procesador esta en modo real de modo que su rango de direccionamiento físico es desde 0x00000 hasta 0xFFFFF, es decir 1Mbyte.
- De modo que la direccion física que generaria un procesador 8086 viene dada por la expresión: `cs << 4 + ip`
- Esto llevara al procesador a realizar el primer OPCODE FETCH en la dirección física 0xFFFF0.
- En este rango de direcciones es necesario mapear una memoria no volátil. En esta dirección de memoria tiene que haber un 'Error'.

Problemas asociados al legacy

- Uno de nuestros principales problemas consiste en que el 8086 hace tiempo que no existe y en cambio sus descendientes compatibles poseen un rango de direccionamiento muchísimo mas amplio.
- Sin embargo por compatibilidad los procesadores posteriores (aun la moderna 10ma generación Core) arrancan en Modo Real accediendo en principio al espacio de direccionamiento de 1Mbyte que referimos anteriormente. (En los procesadores multicore solo uno inicia y el resto duerme)
- Por lo tanto, necesitaríamos colocar una memoria no volátil alrededor del primer Mbyte, interrumpiendo la continuidad de los grandes bancos de DRAM que desde hace tiempo se comercializan.
- Los registros de segmento en Modo Protegido trabajan con un registro cache para guardar sus descriptores a fin de evitar acceder a las tablas de descriptores cada vez que se necesita efectuar un acceso a memoria (o sea... siempre!).
- Todo lo que hacen los procesadores posteriores al 8086 es usarlos en modo real.
- Y por lo tanto les asigna valores que resulten conveni $HOME/.config/Code/Dictionaries/entes, para diseñar el mapa de memoria de un sistema x86 de manera mas sensata.



Básicamente, Intel nos dice que a partir de la arquitectura de 32 bits, va a buscar la primer instrucción a 16 bytes al fondo del espacio de direccionamiento de 4GB. Ahí es donde hay que poner la ROM.

Mapa de memoria inicial

- El procesador inicia su operación buscando operaciones en el fondo de la memoria. Puntualmente a 16 bytes de los 4Gbytes.
- Esto coincide con el anterior comportamiento de iniciar a 16 bytes del Mbyte del 8086.
- Es decir que el procesador inicia en Modo real con su restricción de espacio de direccionamiento a 1 Mbyte de memoria, pero puede realizar opcode fetch muy por encima de esa dirección. . . raro.

> El valor del registro CS no puede cambiar mientras el procesador trabaje en modo real. Ni bien lo haga, la Base Address de su registro cache se pone en 0x00000000, y

> automáticamente queda restringido a continuar buscando sus instrucciones en el primer Mega Byte de memoria.

Selección de Modo

En este punto es conveniente ocuparse de establecer el modo en que el procesador debe funcionar.
Hay tres posibilidades.

1. Dejarlo en modo Real
2. Establecer Modo Protegido Flat
3. Establecer Modo Protegido Segmentado (no recomendado para escribir firmware)
   Los modos 2 y 3 son variantes del Modo Protegido y tiene que ver con la forma en que organizaremos el modelo de segmentacion.

En la practica la ROM se mapea fısicamente en el fondo de los 4 Gbytes.

- Por ejemplo: Si usamos una EEPROM de 4 Kbytes (lo mas pequeno que podemos conseguir actualmente en el mercado), el rango de direcciones para las que se activara el la lınea Chip Select de este componente sera: 0xFFFFF000 - 0xFFFFFFFF.
- Sin embargo cuando empecemos a escribir el programa le vamos a indicar al ensamblador que trabajamos en modo real y por lo tanto en el rango de direcciones 0xFF000-0xFFFFF.
- El ensamblador solo construira un bloque de 4 Kbytes de código con una vision de un mapa de memoria de 1 Mbyte de capacidad y lo ubicara en los ultimos 4 Kbytes de ese espacio. El decodificador de memoria de hardware lo ubicara en 0xFFFFF000.

En primer lugar hay que ponerle un marco de trabajo al ensamblador. No se genera código, simplemente son directivas para el ensamblador,

```
ORG 0xFF000 ;Esto es: (1MB-4KB) -> 0x100000-0x1000=0xFF000.
USE16       ;Indica al ensamblador generar código de 16 bits
```

El único código para iniciar es un loop infinito.

```
init16:
      cli         ;Deshabilita interrupciones
      jmp init16  ;loop infinito
align 16          ;Completa hasta la siguiente dirección múltiplo
                  ;de 16 (verificar con que completa con NOP's).
```

Necesitamos que el label init16 sea mapeado por el ensamblador exactamente en la direccion del reset vector: 0xFFFF0.

> La sentencia align 16 vista solo nos asegura que el codigo generado a partir de init16, se complete con NOP’s hasta la direccion previa a la siguiente direccion alineada a 16 bits.
> Es decir, nos asegura alinear el final de los 4Kbytes.
> Necesitamos completar los primeros 4080 bytes de los 4096 totales ya que de otro

> modo init16 estara al principio de los 4K.
>
> ```
>     TIMES 4080 db 0x90   ;Generamos 4080 NOP's
> init16:
>       cli          ;Deshabilita interrupciones
>       jmp init16   ;loop infinito
> align 16           ;Completa hasta la siguiente dirección múltiplo
>                    ;de 16 (verificar con que completa con NOP's).
> ```

El codigo anterior resuelve el problema pero no es escalable.
A medida que agreguemos codigo para hacer algo mınimamente util, deberemos ajustar el parametro de TIMES para reducir de 4080 a la cantidad que haga falta de acuerdo con el tamano del codigo que incluyamos.
En realidad lo mas complejo sera llevar la cuenta del tamano del codigo.
Por lo tanto hay que buscar un metodo mas eficiente. En el siguiente codigo EQU (por Equal) le genera una constante al ensamblador cuyo valor calcula automaticamente el tamano del codigo generado y lo resta del tamano de la ROM. Problema resuelto.

```
ORG 0xFF000 ;Esto es: (1MB-4KB) -> 0x100000-0x1000=0xFF000.
USE16       ;Indica al ensamblador generar código de 16 bits

code_size EQU (end - init16) ; siempre es el tamaño del código

; Rellenamos la ROM con 0x90 (NOP)
times (4096-code_size) db 0x90

init16:
    cli         ;Deshabilita interrupciones
    jmp init16  ;loop infinito
aqui:
    hlt         ;Si por algún motivo sale del loop: HALT
    jmp aqui    ;Solo sale por reset o por interrupción
align 16        ;Completa hasta la siguiente dirección múltiplo
                ;de 16 (verificar con que completa con NOP's).
end:
```

Organización de la memoria

Dirección física

Cuando cualquier procesador necesita acceder a memoria, debera escribir en el buffer de direcciones el valor correspondiente a la direccion de memoria externa que debe acceder.

- Este valor se conoce como Direccion Fısica, ya que corresponde a la direccion que sera decodificada por el hardware externo para acceder a la memoria RAM o ROM segun corresponda. Es decir a la memoria fısica.
- Siempre el procesador pone la Direccion Fısica en los pines de address cuando su unidad de control habilita la salida del bus de direcciones

Dirección lógica

- Es la direccion de memoria expresada en terminos abstractos por el programador en su codigo fuente.
- Es una forma muy simple de trabajar en forma transparente a la memoria.
- En lugar de referirse al valor numerico de la direccion de hardware, reemplazamos este valor por una etiqueta que incluso resulte ilustrativa de lo que representa esa direccion, y dejamos que el toolchain resuelva su valor numerico.

Dirección virtual

Es la direccion de memoria expresada en terminos abstractos por el programador en su codigo fuente.

- Es una forma muy simple de trabajar en forma transparente a la memoria.
- En lugar de referirse al valor numerico de la direccion de hardware, reemplazamos este valor por una etiqueta que incluso resulte ilustrativa de lo que representa esa direccion, y dejamos que el toolchain resuelva su valor numerico.

MMU

La MMU es fundamental para implementar el modelo de proceso en un Sistema Operativo (como es el caso de Linux).
En este caso cada tarea tiene una o mas areas de memoria para su codigo y datos.
Cuando el scheduler retoma la ejecucion de una tarea, la MMU mapea su espacio de direccionamiento logico (que comienza en 0) en un area de memoria fisica exclusiva y diferente de la de las demas tareas y del propio Sistema Operativo. De este modo cada tarea (proceso) gana su espacio de memoria fisico exclusivo y queda protegido el del resto de las tareas (procesos).
La desventaja es el overhead que genera remapear la memoria cada vez que se conmuta de un proceso al siguiente.

La MMU asegura a cada tarea (proceso), la visibilidad de su propia area de memoria, y de las partes relevantes del sistema operativo.

La division de la memoria en bloques, se puede realizar mediante dos criterios diferentes:

- Paginacion.
- Segmentacion.
  O en algunos casos como una combinacion o superposición de ambos.

Espacio físico

Los procesadores IA-32 organizan la memoria como una secuencia de bytes, direccionables a traves de su Bus de Address.
La memoria conectada a este bus se denomina memoria física .El espacio de direcciones que pueden volcarse sobre este bus se denomina direcciones físicas.

> Los procesadores IA-32 son la continuidad del 8086. Este procesador fue el primer de 16 bits de ancho de palabra, y por ende todos sus registros internos tienen ese tamano. Su espacio de Direccionamiento es de 1 Mbyte ∴ Address Bus es de 20 líneas. Este espacio de direccionamiento se administra por segmentacion.

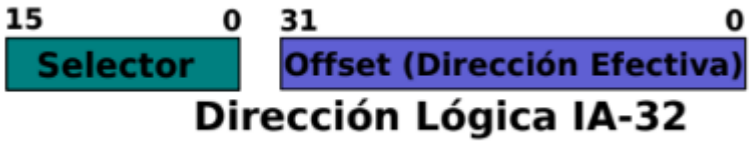
Espacio Lógico

Segmentacion

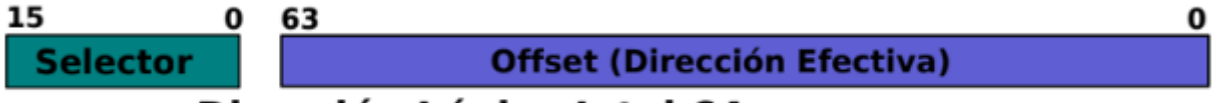
Por diversos motivos que en su momento tuvieron sentido, Intel definio organizar el espacio de direccionamiento de la Familia iAPx86 en segmentos. El compromiso de compatibilidad ato a los siguientes procesadores a mantener este esquema.

Condiciones iniciales de segmentacion

> 4 registros de segmento para almacenar hasta 4 selectores de segmento.
> Registros de 16 bits los segmentos tienen a lo sumo 64K de tamano
> Expresion de las direcciones en el modelo de programación mediante dos valores:
> 1) Identificador del segmento en el que se encuentra la variable o la instruccion que se desea direccionar,
> 2) Desplazamiento, offset, o **direccion efectiva** a partir del inicio de ese segmento en donde se encuentra efectivamente.



Dirección Lógica IA-32



Dirección Lógica Intel 64

- Estos dos valores por si solos no tienen significado físico
- Para lograr a partir de ellos identificar la dirección física de la variable o instrucción el procesador debe realizar una serie de operaciones.

> A una direccion de memoria expresada en terminos de los recursos de la arquitectura del procesador las llamaremos direccion logica.

Traduccion de direcciones

Estos procesadores en Modo Protegido generan una dirección física en el bus de address mediante un proceso de traducción en dos niveles:

1. traslacion de una dirección lógica
2. paginacion del espacio lineal

La MMU

> El procesador posee una MMU (Memory Management Unit) compuesta de dos subunidades conectadas en cascada: La Unidad de Segmentacion que se encarga de trasladar la **direccion logica** en una **direccion lineal**, y la *Unidad de Paginacion* que traduce la **direccion lineal** en una **direccion física** que enviara por el bus de address hacia la memoria externa.

Conclusion

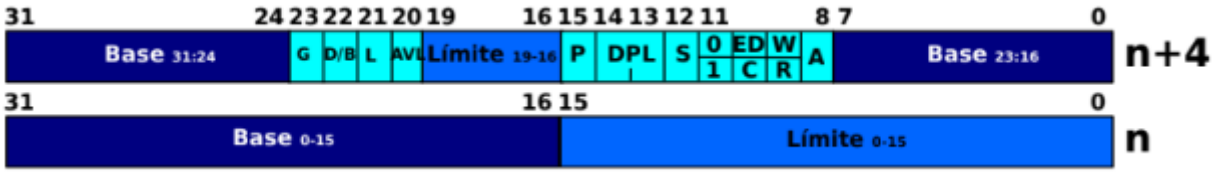
Los modestos 16 bits de un registro de segmento resultan absolutamente insuficientes para almacenar toda esta informacion. Todo lo que puede contener un registro de segmento es un valor, que como ya se ha dicho, se denomina selector de segmento, y que no es otra cosa que una referencia a una estructura de datos mas grande que contiene, la **Direccion Base**, el **Límite** , y los **Atributos** del segmento seleccionado. Como veremos esta estructura se denomina **Descriptor de Segmento**, reside en la memoria RAM del sistema, y para mejor organizacion se los agrupa en tablas, de modo de facilitar su ubicacion al procesador mediante un mecanismo predeterminado.

Selectores de segmento

- **Index**. Se utiliza como ́ındice en la tabla de descriptores. Un valor de Index = n, corresponde al n-esimo elemento de la Tabla. Al tener 13 bits indica que cada tabla puede alojar 213 (8192) descriptores.
- **TI**. Table Indicator. Selecciona en que tabla de descriptores debe buscarse el segmento seleccionado: GDT por Global Descriptor Table (si TI = 0), o LDT por Local Descriptor Table (Si TI = 1).

- **RPL**. Requested Priviledge Level. En el Cap´ıtulo Proteccion lo abordaremos en detalle. Por ahora solo interesa saber que este campo es el nivel de privilegio que declara tener el sueño del segmento, o sea el grado de autorizacion que se tiene para cada acceso: 00 es el mayor privilegio, y 11 el menor.

Descriptores de segmento en 32 bits



- Direccion Base. Es la direccion a partir de la cual se despliega en forma continua el segmento.
- Límite. El Límite de un segmento especifica el maximo offset que puede tener un byte direccionable dentro del segmento. Suele confundirse este concepto con el tamano del segmento. En realidad el Límite es el tamano del segmento menos 1, ya que el offset del primer byte del segmento es 0.

Es tanto texto que no lo quiero ni transcribir.

- ***G***. **G**ranularity. Establece la unidad de medida del campo ***Límite*** . Si ***G*** = 0, el máximo offset de un byte es igual a ***Límite*** . Si ***G*** = 1, el máximo offset es igual a ***Límite*** * 0x1000 +0xFFF.
- ***D/B***. **D**efault / **B**ig. Configura el tamaño de los segmentos. Si es 0, (Default) el segmento es de 16 bits. Si es 1, (Big) es un segmento de 32 bits. Para segmentos de código, ***D/B*** = 0 implica que el tamaño de datos es de 16 bits u 8 bits, y el de direcciones de 16 bits. Si en cambio ***D/B*** = 1, el tamaño de un offset es 32 bits y el de los operandos es de 32 bits u 8 bits. En ambos casos mediante los prefijos de instrucción 66h y 67h respectivamente podemos alterar los defaults. En el caso de un segmento de datos utilizado como pila, si ***D/B*** = 0 las operaciones de la pila son de 16 bits, aunque el operando de la instrucción sea de 8 bits. Si ***D/B*** = 1, son de 32 bits independientemente del tamaño del operando. El valor tope del segmento será también consecuencia del valor de este bit.

- ***L***. El procesador solo mira este bit en el Modo IA-32e. Si en un segmento de código este bit es '1', indica que el segmento contiene código nativo de 64 bits, caso contrario, ejecutará en Modo Compatibilidad. En modo IA-32e, si ***L*** es '1', entonces ***D/B*** debe estar en '0'. Si el procesador no está en modo IA-32e, o si está en este modo pero el segmento no es de código, el bit ***L*** debe estar siempre en '0'.
- ***AVL***. **AV**ai**L**able. Este bit no es usado por el procesador para ningún propósito específico. Queda para que el programador de sistemas le asigne el uso que considere mas apropiado.
- ***P***. **P**resent. Cuando es '1' el segmento correspondiente está presente en la memoria RAM. Si es '0', el segmento está en la memoria virtual (disco). Un acceso a un segmento cuyo bit ***P*** está en '0', genera una excepción #NP (Segmento No Presente). Esto permite al kernel solucionar el problema, efectuando el "swap" entre el disco a memoria para ponerlo accesiible en RAM.

- ***A***. **A**ccedido. Se setea cada vez que se accede una dirección en el segmento. Permite al Sistema Operativo contabilizar los accesos para elaborar estadísticas de uso que permitan identificar cual es el segmento a ser desalojado llegado el momento.
- ***DPL***. **D**escriptor **P**riviledge **L**evel. Nivel de privilegio que debe tener el segmento que contiene el código que pretende acceder a éste segmento.
- ***S***. **S**ystem. Este bit, **activo bajo** permite administrar en las tablas de descriptores, dos clases bien determinadas de segmentos:
  1. Segmentos de Código o Datos
  2. Segmentos de Sistema. Tienen diferentes formatos y en general no se refieren a zonas de memoria (salvo TSS). En general se refieren a mecanismos de uso de recursos del procesador por parte del kernel (por ello reciben el nombre de descriptores de Sistema, ya que **son para uso exclusivo del Sistema Operativo**)

Tipo. Este campo de 4 bits es fuertemente dependiente del tipo (de allí el nombre, según se trate de un segmento de Código, de Datos, o de Sistema). Su detalle, se establece a continuación:

11	10	9	8	Modo 32 bits	Modo IA-32e
0	0	0	0	Reservado	8 bytes superiores en un segmento de 16 bytes
0	0	0	1	TSS de 16 bits disponible	Reservado
0	0	1	0	LDT	LDT
0	0	1	1	TSS de 16 bits Busy	Reservado
0	1	0	0	Call Gate de 16 bits	Reservado
0	1	0	1	Task Gate	Reservado
0	1	1	0	Interrupt Gate de 16 bits	Reservado
0	1	1	1	Trap Gate de 16 bits	Reservado
1	0	0	0	Reservado	Reservado
1	0	0	1	TSS de 32 bits disponible	TSS de 64 bits disponible
1	0	1	0	Reservado	Reservado
1	0	1	1	TSS de 32 bits Busy	TSS de 64 bits Busy
1	1	0	0	Call Gate de 32 bits	Call Gate de 64 bits
1	1	0	1	Reservado	Reservado
1	1	1	0	Interrupt Gate de 32 bits	Interrupt Gate de 64 bits
1	1	1	1	Trap Gate de 32 bits	Trap Gate de 64 bits

Descriptores de segmento en la GDT