Resumen : Organizacion del computador 2

L^3

August 5, 2010

Abstract

Resumen del manual de intel.

Contents

1 Algunos conceptos

1.1 Arquitectura vs. Microarquitectura

Arquictura son los recursos accesibles para el programador (registros,instrucciones,estructuras de memoria,etc) Estos recursos se mantienen y evolucionaan a lo largo de diferentes modelos de procesadores de esa arquitectura.

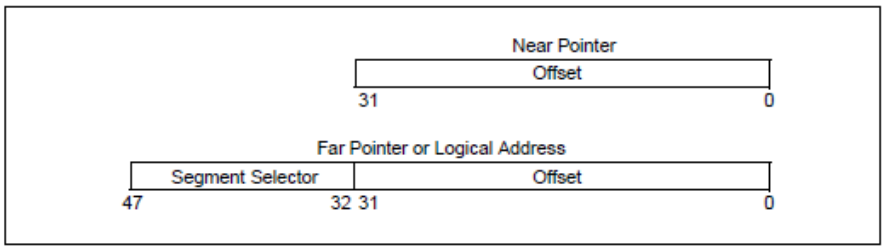
La microarequitectura es la implementacion de la arquitectura visible al programador.

- *RISC* : Reduced Instruction Set Computer, las isntrucciones realizan tareas complejas
- *CISC* : Complex Instruction Set Computer, las instruccion realizan tareas sencillas.
- *MISC* : Minimal Instruction Set Computer
- *OISC* : One Instruction Set Computer

Si hay que dar un ejemplo de *CISC* contra *RISC*, en el ultimo por lo general hay instrucciones solo para acceso a memoria (load/store), mientras que en *CISC* hay instrucciones como el mov que manejan distintos tipos de parametros (con muchos tipos de direccionamiento,etc). Tambien sino hay instrucciones como MOVS que se suele usar para strings.

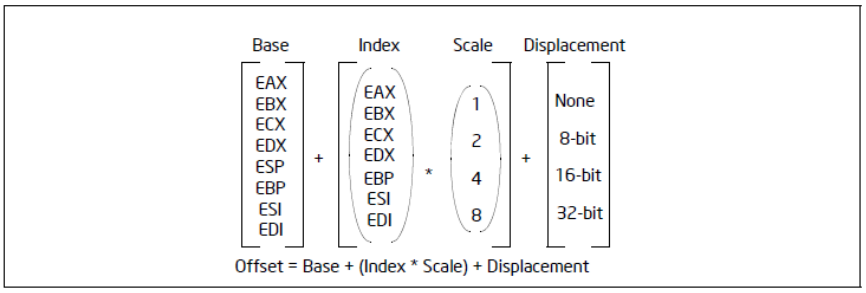
2 Arquitectura basica

2.1 Puntero Instruccion (IP/EIP)



Es un registro que contiene el offset en el actual segmento de codigo para la proxima instruccion que se va a ejecutar. El valor del offset en el registro no puede se rmodificado directamente por software, pero el valor puede modificarse implicitamente por instrucciones como CALL,JMP,RET,IRET,etc. La unica forma de leer el *EIP* es realizar un llamado y leer el top de la pila.

2.2 Modos direccionamiento



- Implicito : Es un parametro que no se pasa a la instruccion, sino que es algun valor en algun registro que siempre se usa. Por ejemplo en POP y PUSH se utiliza implicitamente el segmento de stack y segun el descriptor de segmento se incrmente/decrmenta 16 o 32 bits. Otro ejemplo (mejor) es la instruccion clc
- Inmediato : Algunas instrucciones usan un dato codificado en la propia instruccion como operando fuenta.Este tipo de operando se lo llama inmediato ya que el mismo operando es almacenado en la instruccion. Ejemplo : add eax,14h
- Registro : Son instrucciones que usan registros como operandos tanto en el origen, como en el destino(se se permite). Ejemplos inc edx, sub eax,edx
- Desplazamiento : Es cuando esta codificado en la instruccion el desplazamiento, suele utilizarse para direccionar variables estaticas (uno por lo general lo usa con labels). Ejemplos add [2c00h],ecx, dec [0x7C00]
- Base : Consiste en tener un registro que sirve para acceder a memoria, como el valor puede cambiar suele usarse para referenciar variables dinamicas y estructuras de datos. Ejemplo inc [edx], pero es base siempre y cuando no se toque el valores de edx (sino es mas bien indexado)
- Base + desplazamiento : Consiste en sumar un valor constante (no calculado) al registro base anterior. Suele utilizarse comunmente cuando hay que buscar parametros en la pila,acceso a arrays,etc para el procedimiento. Ejemplo mov eax,[EBP+8]
- Indexado : Ejemplo : inc [edx]; add edx,4 (con escala sirve para acceder a elementos de un array de elementos 2,4 u 8 bytes)
- Indexado*scala + desplazamiento : Es util para acceder a elementos de 2,4 u 8 bytes en un array. Ejemplo inc[edx*4+1] ; add edx,4
- Base + indexado : Ejemplo mov [ebx+edx],eax; inc edx
- Base + indexado + desplazamiento : Permite un manejo eficiente de array de dos dimensiones. Ejemplo mov [ebx+edx+4],eax ; add edx,4

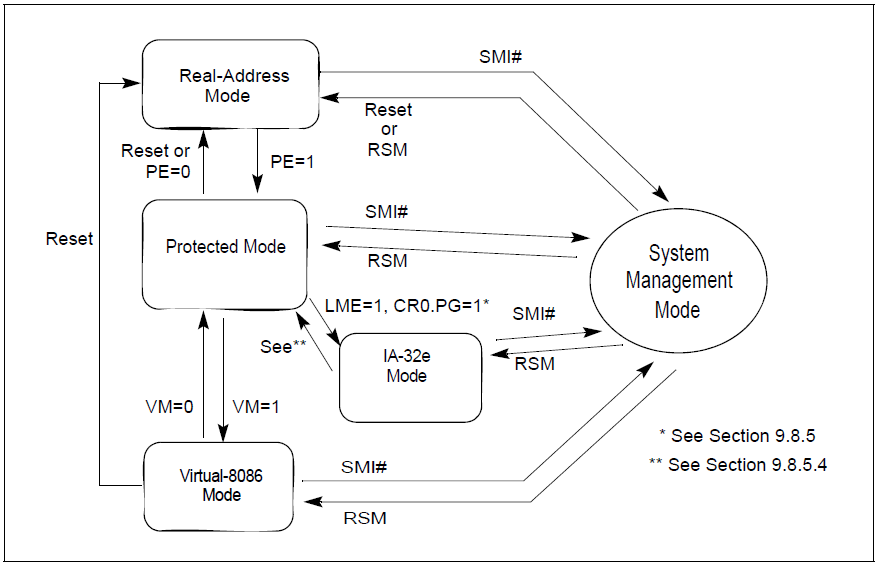
2.3 Uso del segmento selector

Un segmento selector puede ser usado explicitamente o implicitamente. Para usar implicitamente, solo hay que usar el registro adecuado (por ejemplo CS si es para codigo) cargando el valor al registro y luego utilizando operacion que hagan uso de ese registro (por ejemplo al usar PUSH y POP se hace uso del ESP).

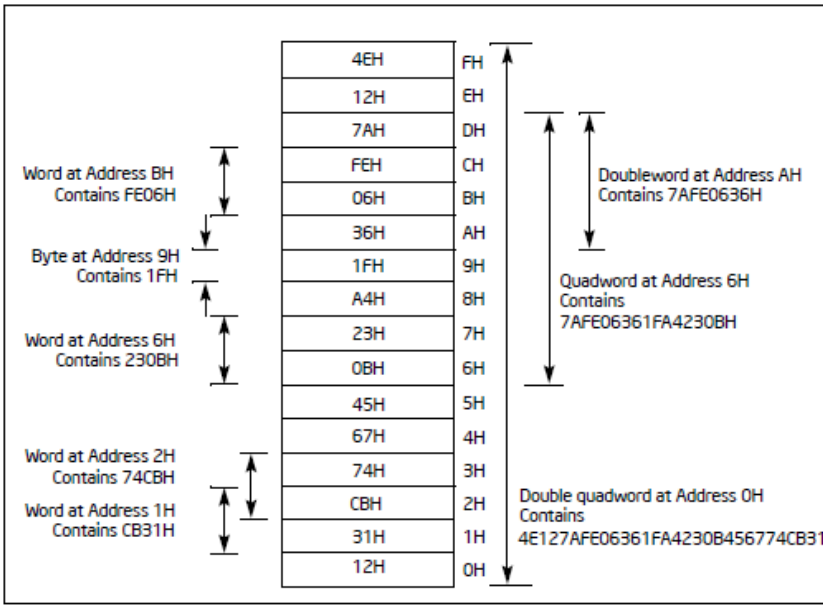
El registro se puede usar explicitamente usando : . Por ejemplo MOV ES:[EBX],EAX; utiliza ES en lugar de DS (el default para movimientos de datos.

2.4 Tipos de datos

2.4.1 Fundamentales



byte,word,double qword,quad word, double quad word.
Como la arquitectura x86 usa little-endian los bits menos significativos ocupan

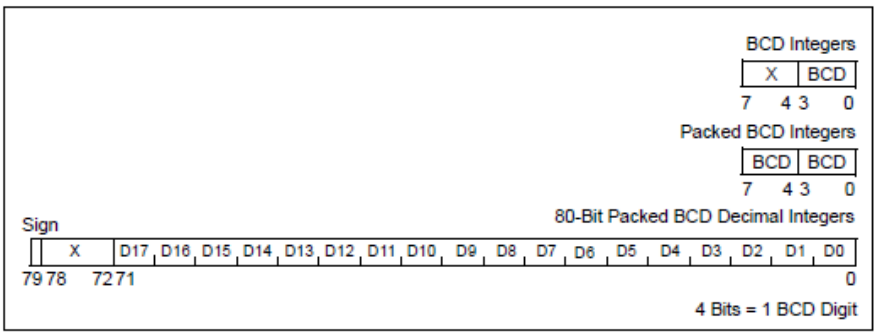


las direcciones mas chicas (o bajas).

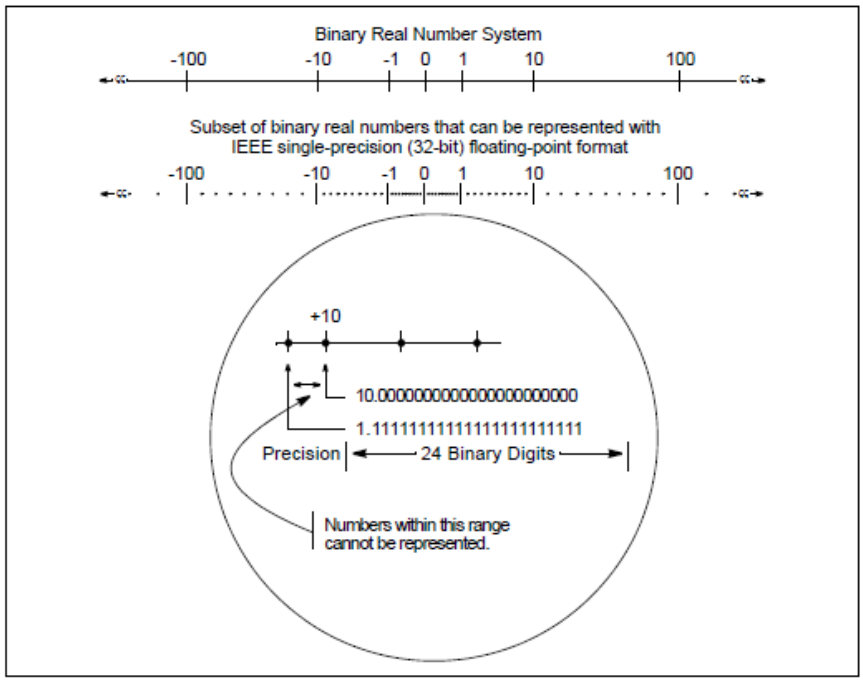
2.4.2 Enteros

Algunas instrucciones permiten interpretar enteros sin signo y enteros con signo. Esto se aplica para los tipos fundamentales, osea hay byte con signo, byte sin signo, word sin/con signo,etc. La representacion de los enteros con signos es complemento a dos.

2.4.3 Binary Coded Decimal



2.4.4 Numeros reales



3 Stack

Una pila es un array continuo de memoria. Esta contenida en un segmento y esta indentificada con el segmento selector en el registro SS. El stack suele usarse para pasaje de parametros entre funciones, almacenamiento de variables locales, almecenamiento de EIP,etc

Por cada ingreso en la pila (push) el procesador en primer lugar drecrementa el stack pointer y luego almacena el dato. Cuando se hace un pop el procesador lee el dato y luego incrementa el stack pointer.

El stack point debe estar alineado a 16bits,32bits o 64bits. Esto depende del ancho del stack segment que se use (SS,ESS,RSS).
Pushear un elemento de 16bits en un stack de 32bits puede desalinear el puntero del stack (esto es el puntero no esta alineado en doublewords), La unica excepcion es cuando se hace un push de un stack segment de 16bits, en este caso el procesador alinea a 32bits automaticamente.

Es responsabilidad del programador,programa,tarea,etc mantener el stack pointer alineado. El no alineamento del stack puede producir serios problemas de performance y hasta fallos de programas.

La funcion CALL hace uso del stack al utilizarse y guarda el EIP antes de hacer el salto (si es near) . Algo similar ocurre con RET, que recupera el valor del EIP del stack (si es near). Si el registro EBP se carga con el valor del ESP inmediatamente despues de entrar en el nuevo procedimiento entonces EBP deberia apuntar al EIP. Es responsabilidad del programador llevar el rastro del EIP en el stack, el procesador no lo mantiene. Tampoco el procesador requiere que se vuelva al mismo procesado llamador, sin embargo esto puede ser peligroso.

3.1 subrutinas

3.1.1 CALL y RET

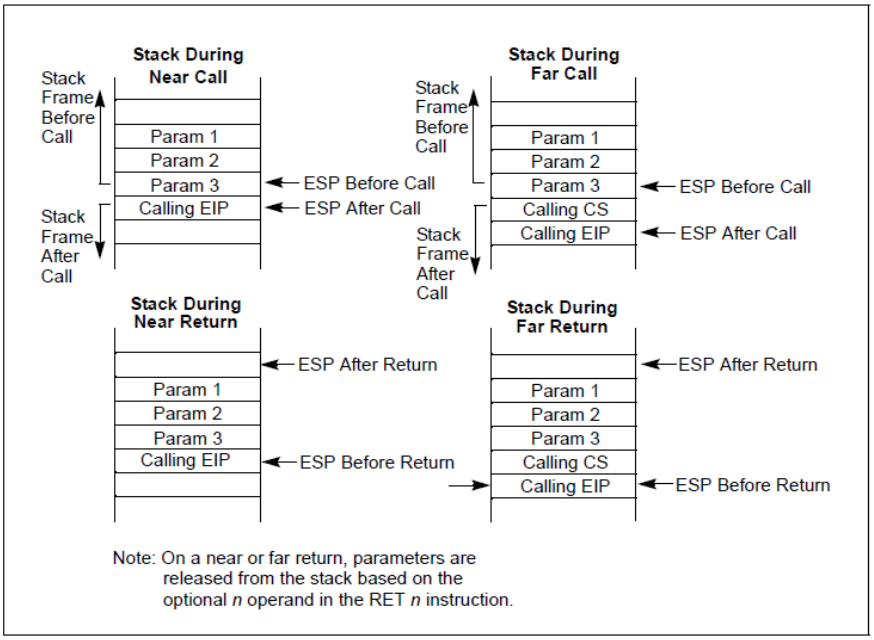
La instruccion CALL permite movimientos entre procedimientos dentro del mismo segmento de codigo (near call) y fuera del segmento de codigo (far call). Usualmente los near call, son llamadas a procedimientos locales mientras que los los far calls son llamadas del sistema operativo o de otra tarea por ejemplo.

La intruccion RET tambien permite hacer near call y far call y adicionalmente permite incrementar el puntero al stack para limpiar parametros que podrian existir en el stack.

Al realizar un call el procesador no garantiza que se guarde el estado de los registros o los EFLAGS. El procedimiento llamador debe guardar explicitamente los registros o los EFLAGS ya sea en el stack o en algun lugar de memoria (si es que los va a modificar). Esto solo ocurre cuando se hace un CALL, ya que si por ejemplo ocurre una interrupcion , esto no es asi.

Se provee de PUSHA y POPA que pushean y popean todos los registros (incluido el ESP, salvo para el POPA que no lo restaura). Si cualquier procedimiento llamado modifica algun registro de segmento, debe restaurarlos antes de realizar el return. Tambien de PUSHF/PUSHFD y POPD/POPDF para guardar y recuperar los EFLAGS.

Cuando se utilizan niveles de privilegio y se hace un CALL a un nivel de privilegio menor, lo que realiza el procesador es transparente excepto por la exception general-protection.



Cuando el call es near el procesdor hace :

- Guarda el actual valor de EIP en el stack
- Carga el valor del offset en el actual EIP
- Comienza la ejecucion del procedimiento llamado

Cuando el RET es near , el procesador realiza lo siguiente :

- Reemplaza el valor del EIP con el valor del tope del stack
- Si se utilizo el parametro opcional, incrementa el valor de ESP
- Comienza la ejecucion desde el EIP

Far CALL :

- Hace un push del registro CS
- Hace un push del EIP
- Carga el selector de segmento del segmento que contiene el procemiento a llamar en CS.
- Carga el offset del procedimiento llamado en el EIP
- Comienza la ejecucion del procedimiento llamado

Far RET:

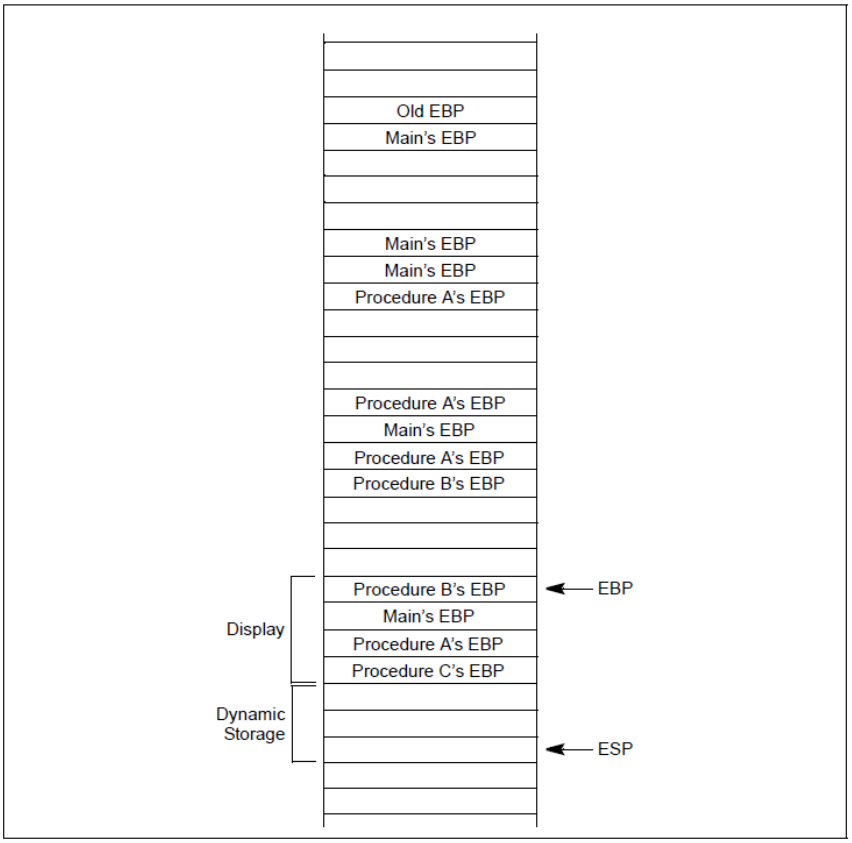
- Hace un pop y carga el valor en el EIP
- Hace un pop y carga el valor en el CS
- Si se uso el parametro opcional, se incrementa el EBP.
- Comienza la ejecucion.

Funcionamiento del CALL y RET entre niveles de privilegio:

3.1.2 ENTER y LEAVE

$ENTER$ y $LEAVE$ son instrucciones alternativas para hacer un llamado a un procedimiento, la diferencia de estas contra $CALL$ y RET es que estan pensadas para ser usadas por un lenguaje estructural como c.

$ENTER$: tiene dos parametros uno para especificar el tama;o para almacenamiento dinamico (variables locales) y otro para indicar el nivel de profundidad de anidamiento. Esta instruccion es muy util para armar el stack frame, ya que usando correctamente el nivel de anidamiento provee una forma sencilla para acceder a las variables del procedimiento llamador (se puede acceder a varios niveles). $ENTER$ hace esto almecenando en la pila los punteros EBP de los otros procedimientos



LEAVE : No tiene parametros y vuelve atras todas las acciones de la instruccion ENTER simplemente reemplando el valor de ESP con el de EBP. Se podria usar RET tambien para volver (pero hay que indicarle por parametro cuando hay que incrementar el ESP). Luego de esta instruccion es necesario realizar un *RET*.

3.2 Pasaje de parametros

Existen tres formas de pasaje de parametros, por registros, por stack y por lista de argumentos. El pasaje de parametros por registros consiste simplemente en cargar los valores de los registros (hasta 6, EBP y ESP no se pueden usar) y luego realizar un CALL. Es importante saber que el CALL no preserva o no garantiza que preserva los valores de los registros, esto es que cuando se vuelve el procedimiento que hizo el CALL los registros pueden tener cualquier valor.
El pasaje de parametros por pila consiste en poner los paramtros en la pila, para que luego el procedimiento al que se llama pueda obtenerlos desde esta. Cuando se utiliza este tipo de pasaje, utilizar EBP para obtener los parametros facilita las cosas. Para el pasaje de lista de argumento consiste simplemente en pasar un puntero a un estructura de datos que contiene todos los parametros. El pasaje del puntero puede ser por registro o por stack.

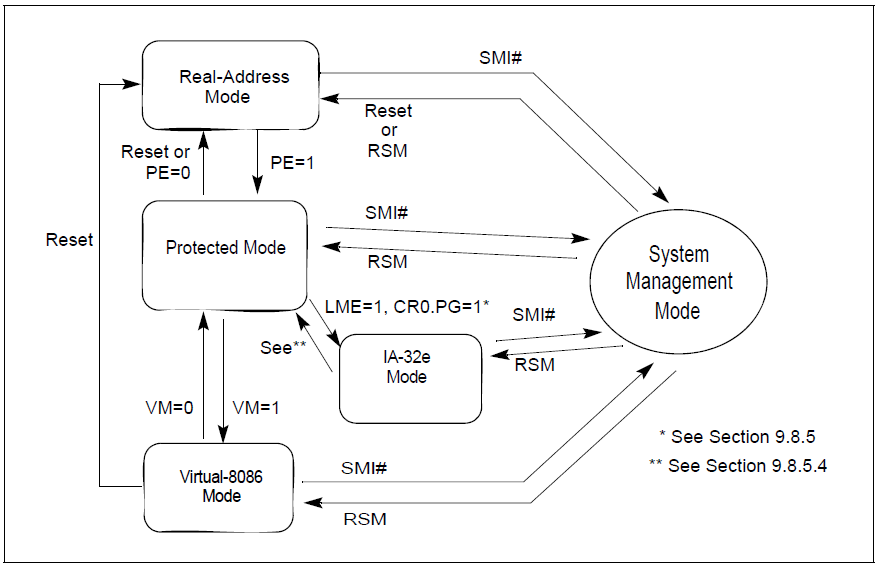
3.2.1 Convencion C

En la convencion C los parametros se pasan por stack. El compilador C pasa los parametros en el orden inverso (a lo que esta en el coodigo c). Ademas al principio de un procedimiento guarda el EBP actual en la pila y luego lo reemplaza por el vaor del ESP. Se puede decrementar el valor de ESP para permitir guardar variables locales. Los parametros se pueden acceder sumando al EBP (en general el primer parametro esta en EBP+8, ya que esta el EBP viejo y el EIP en la pila). Las variables locales se acceden restando al EBP algun valor. Luego es necesario salvar los valores de esi,edi y ebx. La restaurancion antes de llamar a RET es el proceso inverso.

3.3 Modos de operacion

EL procesador tiene los siguientes modos de operacion :

- Modo protegido
- Modo Real
- Modo mantenimiento (SMM)
- Modo virtual 8086
- IA-32e



4 Modo Real

Este es un modo que provee un entorno de programacion del 8086, con algunas extensiones (como cambiar a modo protegido,modo mantenimiento,etc). Esta caracterizado por 20 bits de direccionamiento.

La forma de calcular la direccion fisica es : $(segmento << 4) + offset$

4.1 Direccionamiento en modo Real

En el modo real no se chequea el solapamiento de los segmentoss

Este es el modelo para procesadores 8086. Utiliza un implementacion especifica de segmentacion donde el espacio de direcciones lineal se lo ve como un array de segmentos de 64kb

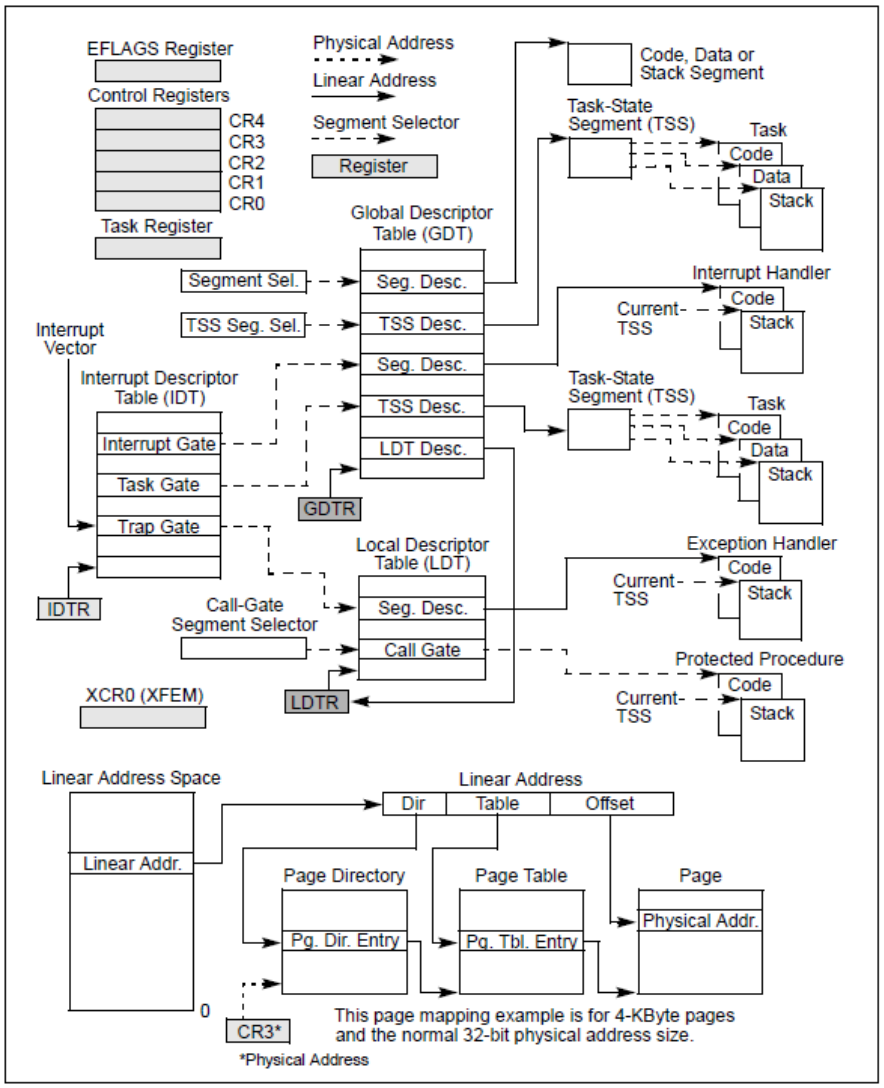
5 Modo Mantenimiento (SMM)

Es un modo de operacion que cambia el modo de direccionamiento a un espacio de direccion distinto, llamado SMRAM. El modelo usado es similar al modo real. Se utiliza para manejo de energia o seguridad.
Se accede por medio de sena; de hardware, cuando se ingresa a este modo el procesador salva el contexto de la tarea en ejecucion, ejecuta el codigo en este modo y luego vuelve a retomar la ejecucion en el punto exacto en que la abondono.

6 Modo Protegido

Es un modo de operacion de los CPU x86 compatible con la serie 80286 y posteriores. El modo protegido tiene nuevas caracteristicas dise;adas para mejorar las multitareas y la estabilidad, como proteccion de memoria, soporte de hardware para memoria virtual asi como conmutacion de tareas.

6.1 Registros de sistema y estructuras



- GDT/LDT : Son tablas que contienen en sus entradas descriptores de segmento. Estan relacionados los registros $GDTR/LDTR$ con estas tablas.
- TSS : define el estado de ejecucion de una tarea. Incluye el estado de los registros,EFLAGS,EIP,etc. Tambien incluye el selector de segmento para la LDT.
- *Gates* : Se usan para proveer acceso a procemientos del sistema o handlers que requieren un nivel mas elevado de privilegio (por ejemplo 0 que es el mas elevado). Hay varios tipos de gates : call gates,interrupt gates y trap gates.
- IDT(Interrupt Description Table) : Es una coleccion de descriptores gate (gate descriptors) que proven acceso a los handlers para la interrupcion o

excepcion. Tango como la GDT la IDT no es un segmento. La base esta guardada en el registro $IDTR$

- TR (Task Register) : Contiene el selector de segmento para la TSS de la tarea actual. (todo lo demas esta en los registrso hidden que cachean la entrada en las tablas de descriptores).
- $LDTR$ contiene el selector de segmento para la LDT.
- $GTDR$ contiene la base y limite de la GDT
- TSS (Task State Segment) : Es un segmento que se utiliza para guardar el estado de la tarea.

Estructuras de cache

- TLB(Translation Lookaside Buffer) : Es un buffer donde el procesador guarda las mas recientes entradas del directorio de paginas y tabla de paginas. Apartir del $P6$ y del Pentium se tiene un TLB para instrucciones y otor para datos.
- Para evitar acceso a la GDT y LDT cada vez que se necesita un descriptor de segmento, el procesador mantiene un cache invisible por cada registro de segmento cuyo contenido es un descriptor de segmento.

Para activar el modo protegido, lo que se tiene que hacer es activar el bit PE (bit 0 cero del registro CR0) Esto NO es todo lo que hay que hacer, solo es el principio.

7 Organizacion de la memoria (Modo Protegido)

7.1 Tablas de descriptores de segmento : LOCAL y GLOBAL (LDT y GDT)

Las tablas de descriptores de segmentos, son un array (tama;o variable) de descriptores de segmento.

Tabla Global existe una en todo el sistema, que puede ser usada por cualquier programa o tarea del sistema. Tablas locales, pueden existir muchas y usualmente cada programa puede tener una o tambien varios o todos los programas comparten una misma LDT.

La GDT no es un segmento, es una estructura de datos en el espacio de direcciones lineal. La base y el limite de la GDT se establecen en el registro GDTR y la base de la GDT deberia estar alineada a 8bytes para obtener el mejor performance.

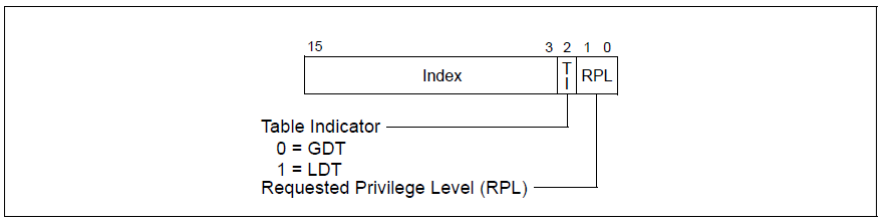
Los LDT estan ubicados en un segmento del sistema del tipo LDT. La GDT debe contener un descriptor de segmento para la LDT (y si tiene varias LDTs debe contener una por cada uno). Para eliminar las traducciones de direccion al acceder a la LDT, se utiliza un registro especial (LDTR) donde se el selector de segmento para la LDT.

En el stack descriptor :
D flag se utiliza en las instruccion de uso de pila , como PUSH y POP, para saber cuanto es el incremento o decrmeento en el stack pointer.

7.1.1 Registros hidden

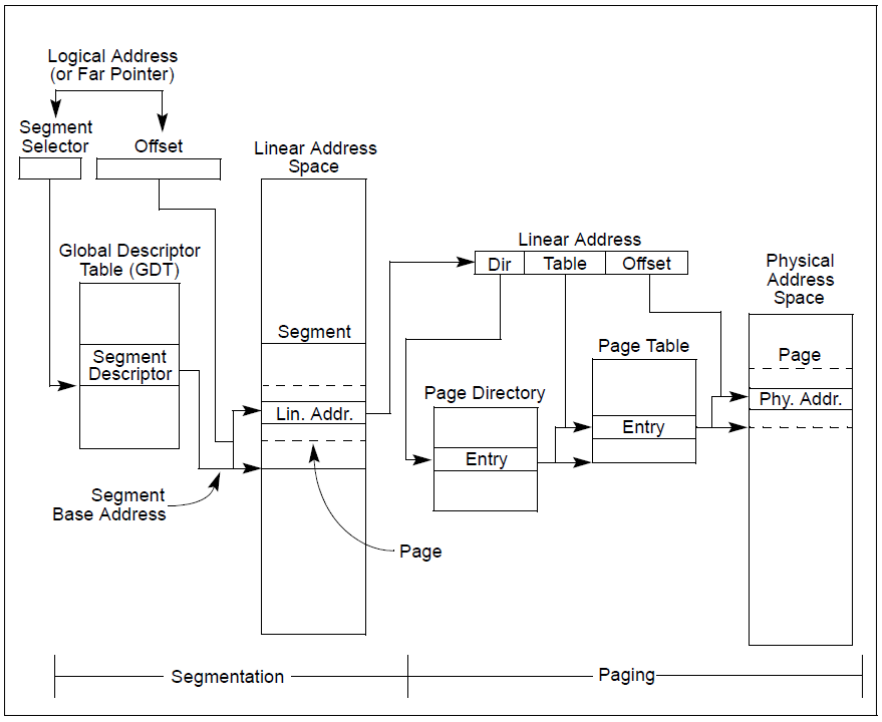
Son unos registros, totalmente transparentes e inaccesibles para el programador (ni SO) que evitan el acceso a la GDT. Estos registros hay tantos como selector de segmento existen,tambien hay uno para la LDT y TR,que evitan accesos a memoria para obtener los descriptores de segmento de estos.

7.1.2 Segmentos de sistema,descriptores de segmento

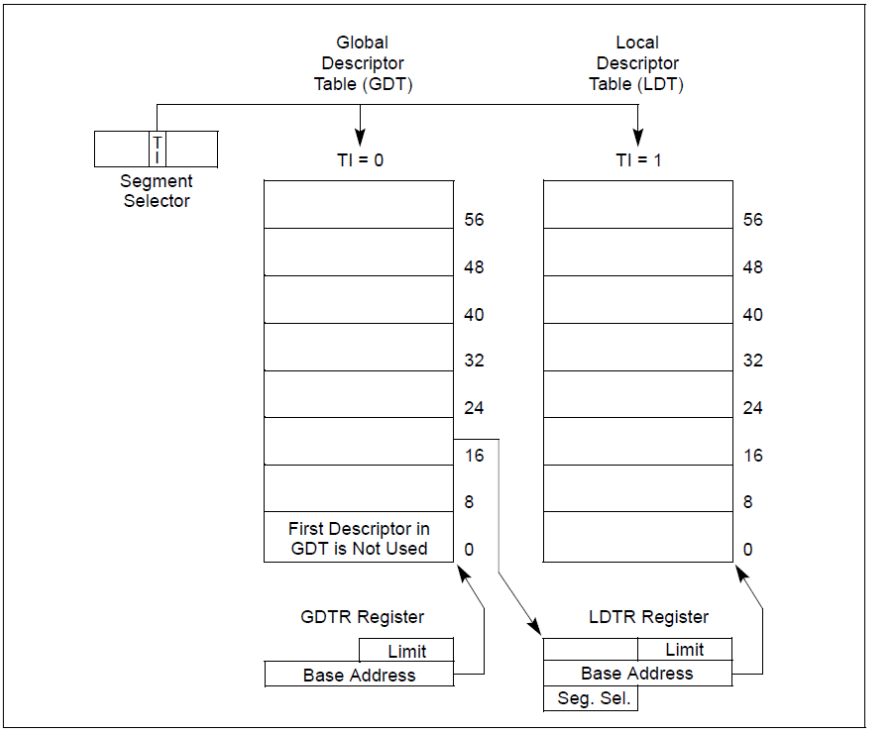


Aparte de los segmentos de codigo, datos,stack,etc exiten otros segmentos como el TSS (Task state segment) y el LDT. Es importante saber que la GDT no es un segmento ya que no tiene descritpor de segmento ni selector de segmento. Tambien existen unos descriptores especiales llamados gates (que hay del tipo call,interrupt,trap y task). Estos proveen una forma para permitir de la ejecucion de codigo mas privilegiado. Por ejemplo, cuando se realiza un CALL gate, se realiza un chequeo del CPL con el privilegio del CALL gate y con el selector del codigo destino.

7.2 Mecanismo de acceso a memoria



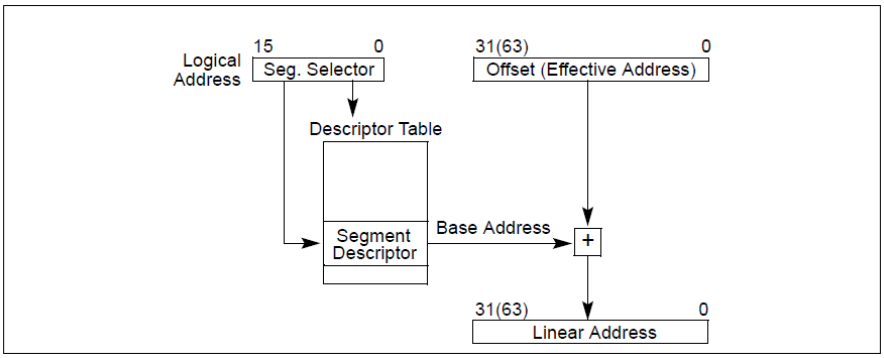
En el modo protegido todos los accesos a memoria pasan atra vez de la GDT o por la LDT (opcionalmente). Cada descriptor de segmento tiene asociado un selector de segmento. Para acceder a un byte en un segmento,se requiere un selector de segmento y un offset. Con el selector de segmento se provee acceso al descriptor de segmento, esto se realiza con el indice en el selector y se busca en la GDT (o LDT si se esta usando). Desde el descriptor de segmento, el procesador obtiene la base del segmento en el espacio lineal de direcciones. Luego el offset provee la ubicacion del byte junto con la base. Este mecanismo se utiliza para acceder cualquier tipo de datos,codigo,stack,etc siempre y cuando el nivel de privilegio se lo permita. Hasta este punto se obtiene la direccion lineal.



Ojo, cuando se utiliza la LDT (un bit en el selector lo indica) el mecanismo es el siguiente. Primero se tiene una direccion logica que consiste de un offset de 32bits y un selector de segmento (este selector tiene el bit $TI = 1$). Usan el LDTR se accede a la GDT (con el selectot que tiene el $LDTR$)para encontrar el descriptor de segmento de la LDT (tiene que existir esta entrada por requirimiento). En el descriptor de la LDT se saca la base y junto con el selector de segmento (el de la direccion logica) se busca la entrada del descriptor de segmento (del segmento donde esta el dato buscado). En el descriptor final esta la informacion (base y limite) que junto con el offset forman la direccio lineal.

Ver el mecanismo de Switch de tareas en administracion de tareas. El $LDTR$ se guarda en el TSS Ademas a esto le faltan cosas (como chequeo de privilegios,etc), pero es una aproximacion a lo que hace . Ademas aca tambien faltan cosas como paginacion, que si esta activa lo que se saca del descriptor de segmento destino se usa como direccion lineal y se lo transforma a una direccion fisica usando los directorios y tablas de paginas. Si no se usa paginacion, la direccion lineal (base del descritpor de segmento y offset de la direccion logica)

se mapea directo a memoria.



Para acceder a la memoria, los programas seguiran trabajando con segmento al igual que en el modo real. La diferencia esta en que en el modo protegido los registros de segmentos apuntan a un descriptor de segmento.

7.3 Paginacion (memoria virtual)

La paginacion es una forma de mapear el espacio lineal de direccion en peque;as partes a la memoria virtual y disco secundario. Cuando se usa paginacion el procesador divide el espacio lineal en fragmentos (4kb,2Mb o 4MB) que pueden ser mapeados a la memoria fisica o disco. El programa usa direcciones logicas, que luego el procesador traduce a direcciones fisicas. Al ir obteniendo la direccion fisica, se puede saber si la pagina o tabla esta en memoria fisica, gracias a los datos que hay en el directorio o tabla de paginas que indican si esta o no en memoria.

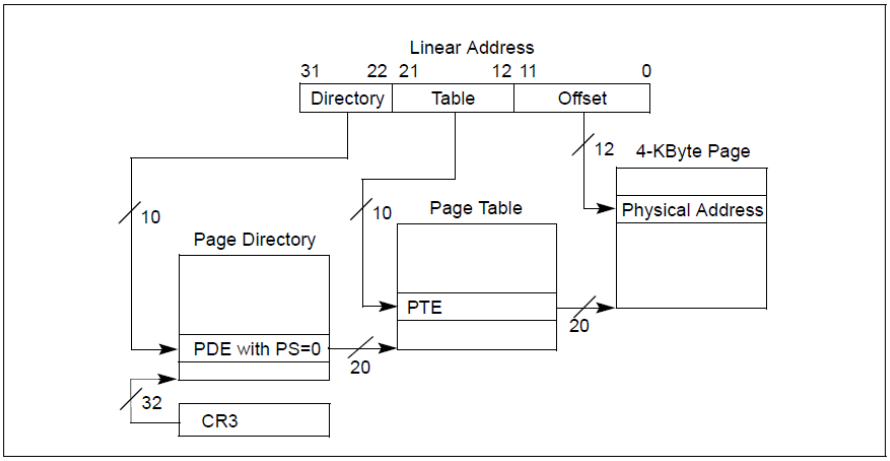
Si la pagina no esta en memoria, el procesador lanza una fallo de pagina. El handler para la execpcion de fallo de pagina, usualmente llama alguna rutina del sistema operativo para traer de disco la pagina. La informacion que utiliza el procesador para lanzar una execpcion de fallo de pagina esta contenida en los directorios de paginas y tablasd de paginas.

Las mas recientes entradas del directorio de paginas y de la tabla de paginas son cacheadas en la TLB, de esta forma se evita perder ciclos de bus. Las TLBs son inaccesiblees para programas o tareas que tengan nivel mayor a 0, solo con nivel 0 se pueden invalidar entradas de la TLB.

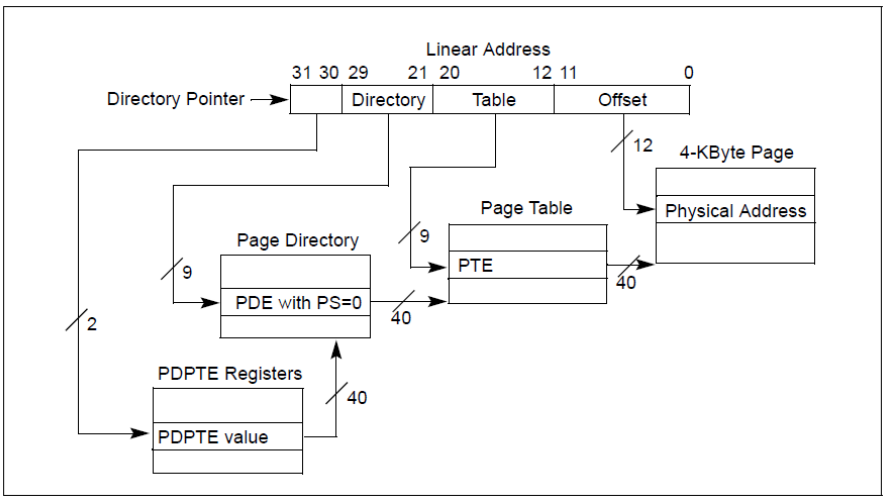
Al utilizar paginacion , se puede permitir un direccionamiento de 36bits (al activar el PAE flag). Esta extension consiste en un directorio adicional de punteros de directorios de paginas que usado con directorios de paginas y tablas de paginas permiten referenciar direcciones mayores a FFFFFFFFh. Existe otra forma adicional para extender a 36bits de direccionamiento llamada PSE-36.

El mecanismo de traduccion de paginas se ejecuta cuando el procesador ya hizo la traduccion de la direccion lineal (y de la lineal si esta la paginacion activada realiza la traduccion a la fisica con las estructuras de paginacion).

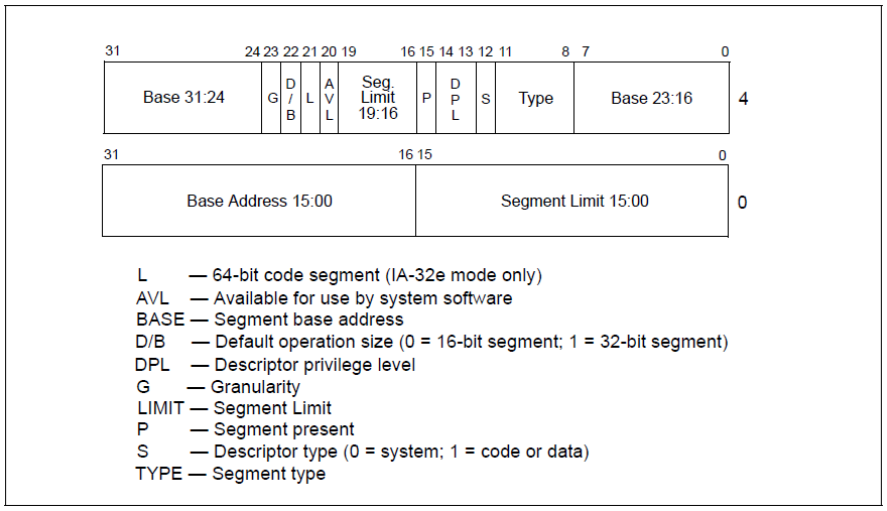
Paginacion Con 4kb:



Paginacion COn PAE activado :



7.3.1 Descriptores de segmento



Contienen :

- Direccion Base : Direccion apartir de la cual comienza el segmento
- Limite : Tama;o del segmento
- Atributos : Permitos (Read Only,Codigo/Datos) y demas caracteristicas.

El tama;o del descriptor suele ser mayor a los 64bits, se requiere 32bits para la base, 32bits para el limite y otros n bits para los atributos.

El descriptor se almacena fuera del procesador, en la memoria RAM. Los descriptores se agrupan en tablas.

Los registros de segmento, en modo protegido se utilizan para encontrar los descriptores de segmento. (en modo real contenian toda la informacion).

En modo protegido un registro de segmento, se lo llama selector.

El segmento selector esta formado por tres partes :

- Indice : Es un offset para la tabla de descriptores de segmento (GDT o LDE)
- TI : Flasg que indica si se usa la GDT o LDT
- RPL : Se utiliza para el chequeo de privilegios

7.3.2 Gates

7.4 Registros de administracion de memoria

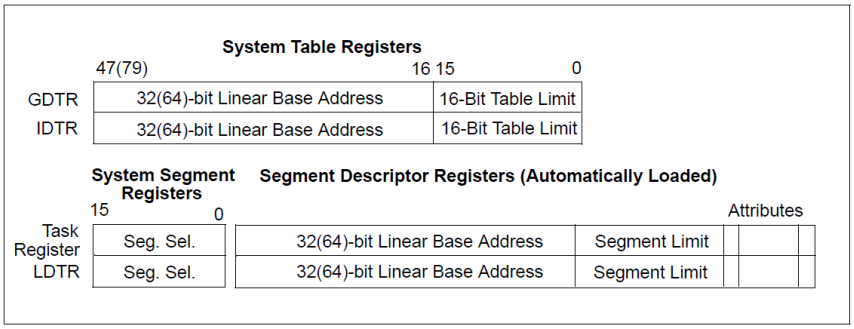


Figure 2-5. Memory Management Registers

7.4.1 GDTR

El registro consiste en una base de 32bits y un limite de 16bits. La direccion base especifica el inicio de la tabla, donde esta ubicado el byte 0 de la GDT.

7.4.2 LDTR

Consiste en un selector de segmento de 16bits. El segmento que contiene la LDT, debe tener una entrada en la GDT.

7.4.3 IDTR

El registro consiste en una base de 32bits y un limite de 16bits

7.4.4 TR : Task Register

El registro task tiene 16bits para el segmento selector del segmento. El selector referencia al descriptor de la TSS en la GDT. La base especifica la direccion lineal del byte 0 del TSS. El limite especifica el numero de bytes del TSS. Cuando ocurre un switch de tareas, el task register se carga automaticamente con el contenido del segmento selector y descriptor de la TSS.

8 Proteccion (Modo protegido)

En el modo protegido los procesadores proveen un mecanismo de proteccion por hardware para segmentacion y paginacion. Esta proteccion se basa en permitir el acceso a ciertos segmento o paginas segun niveles de privilegio. Posee los siguiente mecanismos de proteccion :

- Chequeo de limite de los segmentos
- Chequeo de tipos de los segmentos

- Chequeo de los niveles de privilegio de segmentos y paginas
- Restruccion del dominio de direcciones a las tares
- Restriccion a los puntos de entrada a los procedimientos
- Restriccion del uso de set de instrucciones

8.1 Chequeo de limites

Los chequeos que se realizan son en cuanto a la informacion que se quiere acceder de memoria este dentro de los limites del segmento. Ademas se hacen chequeos de limites en los descriptores de segmento.

8.2 Chequeo de tipos

Los descriptores de segmento contienen informacion para el chequeo en dos lugares :

- el flag S (tipo de descriptor)
- tipo del campo

El procesador utiliza esta informacion para detectar errores o usos mal intencionados de los segmentos o gates.

El flag S se utiliza para saber si el descritor es de sistema,codigo o datos. El campo de tipo son 4bits que se utilizan para definir distintos tipos de segmentos de codigo,datos y descriptores de sistema. Algunos de los chequeos de tipos que se realizan :

- Cuando se carga el selector de segmento en el registro CS se chequea que el descriptor de segmento indique que el segmento es del tipo de codigo.
- otro chequeo que sea realiza es cuando se carga el selector en el registro SS, los datos tiene que poderse escribir.

Ejemplo :

- No se permite cargar en el CS segmentos que no son de codigo
- No se permite cargar un selector de codigo cuyo descriptor no tenga permiso de lectura en ningun otro registro de segmento
- En SS solo se pueden cargar selectores que correspondan a segmentos con permiso de escritura
- $LDTR$ solo puede ser cargado con un selector de LDT (bit $S = 0$ y tipo $0010b$)
- TR solo puede ser cargado con un selector TSS

Algunas comprobaciones se hacen cuando los segmentos ya estan cargados, por ejemplo no se puede escribir un segmento de codigo, si el flag $W = 0$ no se puede escribir el segmento, no se puede leer un segmento de codigo si $R = 0$.

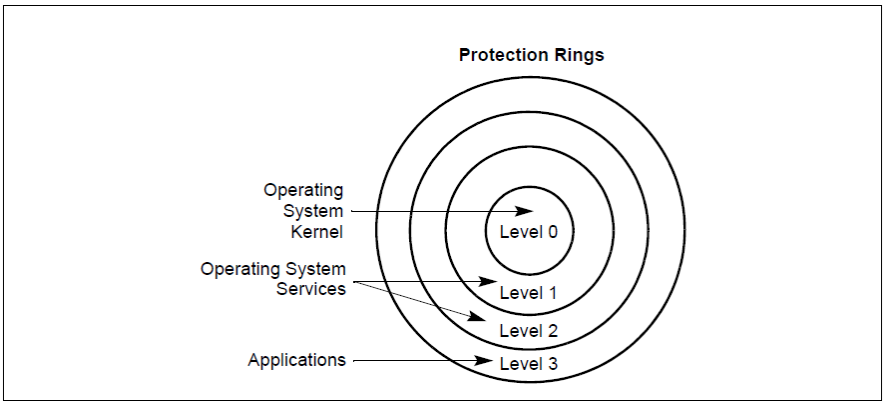
Comprobaciones durante la ejecucion de instrucciones cuyo operando es un selector de segmento :

- Las instrucciones CALL far y JMP far solo pueden acceder a descriptores de segmento de codigo (conforming o non conforming), a puertas de llamada (gates), puertas de tarea o descriptores de TSS
- $LLDT$ debe tener como operando un descriptor de LDT
- LDR debe tener como operando un descriptor TSS.
- LAR debe tener como operando fuente un descriptor de segmento de codigo,datos, TSS, LDT, puerta de tarea o puerta de llamada
- LSL debe tener como operando fuente un descriptor de segmento de codigo, datos , TSS, o LDT
- Las entradas IDT deben ser puertas de interrupcion, de excepcion o de tarea. De otro modo cualquier acceso a dicha tabla con, por ejemplo, la instruccion INT causara un excepcion de proteccion general.

8.3 Chequeo de selector null

El intento de cargar un valor nulo en los registros de segmento CS y SS generara una excepcion (general-protection). Un selector nulo puede cargarse en, DS, ES, FS, GS pero cualquier intento de acceso arrojara una general protection.

8.4 Chequeo de nivel de privilegio



El mecanismo de proteccion de segmentos reconoce hasta 4 niveles de proteccion, numerados del 0 al 3. Cuando mas grande el numero menor el privilegio. En general el nivel cero se utiliza para software critico, como el kernel de un sistema operativo. Los niveles 1,2 podria ser usados para servicios del sistema operativo Mientras que el nivel 3 se deberia utilizar para aplicaciones de usuario. Hay implementaciones de sistemas operativos que solo utilizan dos niveles, en este caso deberian usar el nivel 0 y el nivel 3.

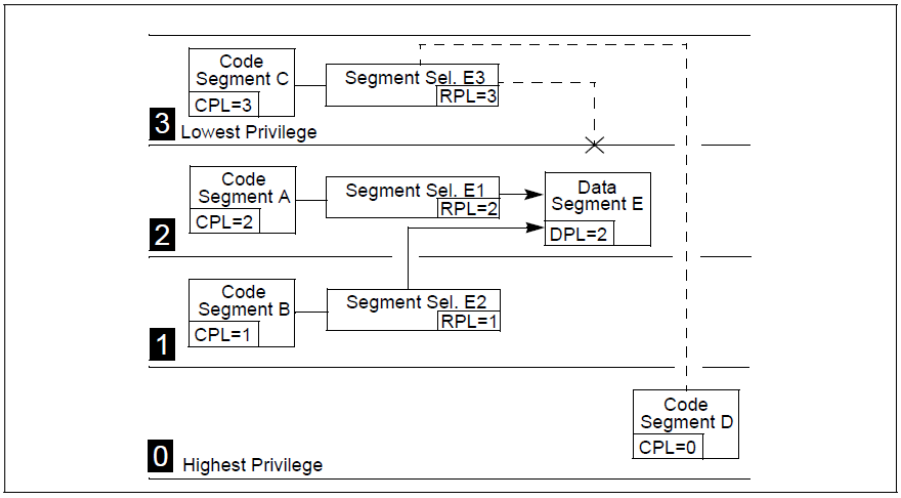
Para poder permitir el chequeo de nivel de privilegio el procesador reconoce tres tipos de niveles de privilegio.

- CPL (Current Privilege Level) : Es el actual nivel de privilegio del programa o tarea en ejecucion. Se guarda en los registros CS y SS CS y SS.
- DPL (Descritor Privilege Level) : Es el nivel de privilegio de un segemnto o un gate. El DPL se guarda en el descritor de segmento o en el descritor de gate. Cuando el codigo intenta acceder a un segmento o un gate, se chequea el DPL contra el CPL y el RPL del selector de segmento o selector de gate.
- RPL (Requested privilege Level) : El RPL es un nivel de privilegio que se asigna a los selectoresa de segmento. El procesador chequea el RPL junto con el CPL para determinar el acceso. Incluso cuando el progrma o tarea tiene el suficiente nivel de privilegio, el acceso es denegado si el RPL no tiene suficiente nivel. Este puede sobreescribirse por el programa. En general el procesador usara $EPL = MAX(CPL, RPL)$ (EPL = Effective Privilege Level).

El chequeo de nivel de privilegios se verifican cuando el selector de segmento de un descriptor de segmento se carga en un registro de segmento. El chequeo usado para el acceso a datos difiere del de tranferencia de segmentos de codigo.

8.4.1 Chequeo de nivel : acceso a datos

Para acceder a los datos en el segmento de datos, los selectores de segmento para datos debe ser cargado en alguno de los registros de segmento para datos. or en el registro de segmento stack (SS). Antes de que el procesador cargue el segmento selector en un registro de segmento, realiza un chequeo de privilegios comparando el nivel de privilegio del programa en ejecucion (CPL) contra el RPL del segmento selector, y el DPL del descriptor de segmento. El procesador carga el valor sel selector de segmento en el registro de segmento si el DPL es meyor o igual al el CPL y el RPL.



En el ejemplo :
Si un segmento de codigo con $CPL = 3$, usando un selector de segmento con $RPL = 3$ no puede acceder a un segmento de datos con $DPL = 2$.Incluso no

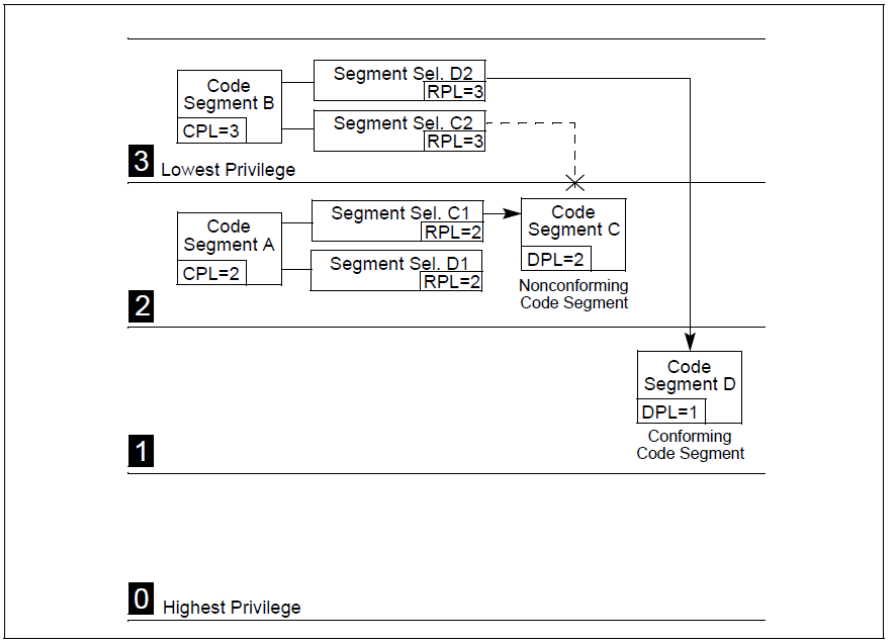
podria acceder al segmento de datos si usara el selector con $RPL = 2$ ya que su CPL se lo impediria.
Un segmento de codigo con $CPL = 2$, usando un selector de segmento con $RPL = 2$ puede acceder a un segmento de datos con $DPL = 2$.
Si un segmento de codigo con $CPL = 0$, usa un selector de segmento con $RPL = 3$ NO puede acceder a un segmento de datos con $DPL = 2$.

8.4.2 Chequeo de nivel : carga de SS

El chequeo tambien se hace al cargar un selector de segmento en el registro SS, la diferencia es que todos los niveles relacionados conceptopsel segmento de stack tienen que coincidir con el CPL, esto es que el CPL , el RPL del selecttor de segmento stack y el DPL del descriptor de segmento del stack tienen que ser iguales al CPL.

8.4.3 Chequeo de nivel : transferencia de control entre segmentos de codigo

Para transferir el control de un segmento de codigo a otro, el segmento selector del segmento de codigo destino tiene que ser cargado en CS. Mientras el procesador carga el valor, examina el descriptor de segmento del destino y verifica limites,tipos y privilegios. Si todos los chequeos son satifactorios el registro CS es cargado satisfactoriamente , el control de programa es tranferido al nuevo segmento y se comiendo la ejecucion del programa.



EXPLICAR

8.5 Descriptores Gate

Proveen acceso controlado a los segmento de codigo. Hay 4 tipos de gates :

- Call gates : Facilita una tranferencia controlada de control de programa cuando hay cambios de privilegios .Tambien suelen usarse para cambiar entre segmentos de codigo de 16 y 32 bits.
- Trap Gate : Son usados para llamar al manejador de este tipo de excepcion.
- Interrupt Gate : Son usados para llamar al manejador de la interrupcion
- Task Gate : Son usados para switcheo de tareas

A diferencia de la mayoria de los descriptores que poseen la base del segmento, este posee un selector de segmento. El DPL es el nivel de privilegio necesario para usar el descriptor.

8.5.1 Acceso por medio de call Gates

Para acceder al call gate, un puntero far es operando de un $CALL$ o JMP. Cuando el procesador tiene el selector de segmento del call gate, utiliza la GDT o LDT para acceder al descriptor.
Luego utiliza la base que consigio en el descriptor que saco de la GDT o LDT junto con el offset del descriptor gate.

Para el chequeo de privilegio se usa, el CPL , el RPL del call gate selector, el DPL del call gate descriptor y el DPL del descritor de segmento de codigo (del call gate).

Los chequeos de privilegio dependen desde que instruccion se quiere acceder a cal gate. Si es desde un $CALL$:

- $CPL \leq$ call gate DPL y $RPL \leq$ call gate DPL
- Destino es un segmento de codigo conforming $DPL \leq CPL$
- Destino es un segmento de noncodigo conforming $DPL \leq CPL$

Si es desde un JMP :

- $CPL \leq$ call gate DPL y $RPL \leq$ call gate DPL
- Destino es un segmento de codigo conforming $DPL \leq CPL$
- Destino es un segmento de noncodigo conforming $DPL = CPL$

El DPL del call-gate especifica que nivel de privilegio se requiere para acceder al call gate.
Si el chequeo de privilegio de usar el call-gate fue satisfactorio, el procesador chequea el DPL del descriptor de segmento de codigo del call-gate.

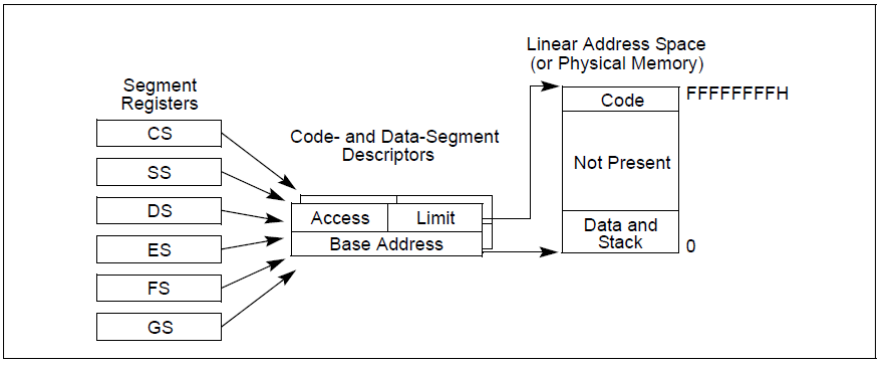
8.6 Stack Switch

Cuando se usa un call gate para tranferir el control de programa a un codigo nonconforming de nivel de privilegio menor (de 3 a 0 por ejemplo) el procesador realiza un cambio de pila. Esto se hace para prevenir que los procedimiento de mas privilegio se queden sin lugar en la pila. Tambien previene por errores o intencionales que procesos privilegiados interfieran por compartir una pila. Para cada tarea se requieren hasta 4 pilas una para cada nivel. Cuando se esta usando el nivel 3 el puntero a la pila esta en SS o ESP, y cuando se hace un cambio de tarea este se guarda. Los punteros para los niveles $0, 1, 2$ son guardados en la TSS y esto es solo lectura. El sistema operativo es responable de setear los stack al memomento de la creacion de la tarea.

9 Administracion de la memoria

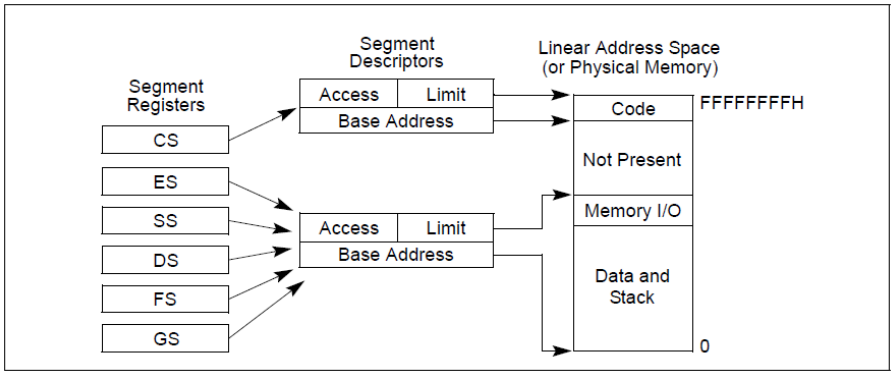
El procesador provee de dos facilidades para la administraciones de memoria : paginacion y segmentacion.
Segmentacion provee un mecanismo para separar distintas partes de un programa (stack,codigo,datos o estructuras de sistema) asi multiples programas (o tareas) puede ejecutarse en le mismo procesador sin interferencia entre estas. Paginacion provee un mecanismo para el convencional paginacion por demanda, sistema de memoria virtual donde porciones del programa se mapean a memoria fisica segun se necesita. Paginacion tambien puede usarse para separar distintas tareas. El uso de segmentacion es obligatorio, no hay forma de deshabilitarlo y el sistema operativo tiene que usar alguna forma de segmentacion. La paginacion es opcional.

9.1 Modelo Flat Basico



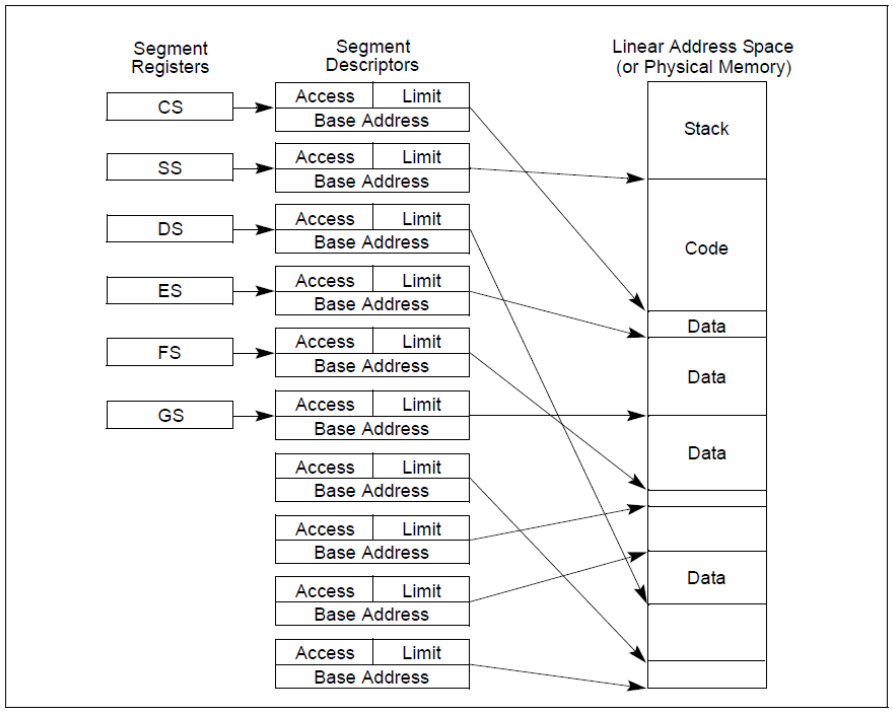
Es un modelo sencillo donde el sustema y las aplicaciones tienen acceso a un espacio de direcciones lineal sin segmentar. Para implementar este modelo es necesario dos registros de segmentos, el de codigo y datos. Estos dos registros mapean el mismo espacio de direccion y el valor de ellos es cero y tienen un limite de 4Gb. Este mecanismo nunca arroja excepciones de fuera limite, incluso cuando en la direccion no existe memoria fisica.

9.2 Modelo Flat Protegido



Es similar al modelo basico, pero los limites tienen valor igual a la cantidad de memoria fisica. De esta forma cuando se quiere acceder a direcciones que no existen en la memoria fisica se arroja una excepcion de fuera de limite. A este modelo se le puede agregar adicionales protecciones para separar codigo de usuario y de supervisor. Por ejemplo para separar codigo y datos de usuario/supervisor se requieren cuatros registros de segmento. Este tipo de administracion, junto con una estructura de paginacion para cada tarea pueden proteger aplicaciones entre ellas. Un dise;o similar se usa en sistemas operativos populares.

9.3 Modelo Multi-Segmento



En este modelo se usa la maxima capacidad del mecanismo de segmentacion para proteger codigo, estructuras de datos, programas y tareas. En este modelo

a cada programa o tarea se le asigna su propio descriptor de segmento y sus propios registros de segmentos. Los segmentos puede ser privados o puede ser compartidos entre tareas.

9.4 Traduccion de direcciones Logicas a Fisicas

9.4.1 Traduccion de direccion logica a lineal

En el modo protegido el procesador usa dos etapas para traducir un direccion logica a fisica.
La primera etapa $logical-address-translation$ y la segunda $linear-address-translation$. Incluso con el minimo uso de segmentos el procesador mapea la direccion fisica por medio de las direcciones logicas. Una direccion logica consiste en un selector de segmento de 16 bits y un offset de 32 bits.

Para traducir una direccion logica a una lineal el procesador realiza lo siguiente :

- Use el offset en el selector de segmento para ubicar el descriptor de segmento en la GDT o LDT (este paso solo se requiere cuando el procesador carga un nuevo selector de segmento en el registro de segmento)
- Examina el deescriptor de segmento para chequear privilegios de acceso y que el offset este dentor de los limites
- Del descritor de segmento usa la base para sumar con el offset (el de 32 bits) y forma la direccion lineal.

9.4.2 Traduccion de direccion lineal a fisica : Paginacion Desactivada

Si paginacion no se utiliza, el procesador mapea la direccion lineal directamente con la fisica. Si se usa , luego el procesador utiliza un segundo nivel de traduccion.

9.4.3 Traduccion de direccion lineal a fisica : Paginacion Activiada

Cuando se obtiene la direccion lineal y se utiliza paginas de 4kb la direccion se separa en tres partes

- bits 22 a 31 proveen un offset para una entrada en el directorio de paginas, que dicha entrada corresponde a la direccion fisica a una tabla de paginas.
- bits del 12 al 21 proveen un offset para una entrada en la tabla de pagina que contiene la base de la pagina.
- bits 0 al 11 es un offset que junto con la base de la pagina se suman para obtener la direccion fisica del dato.

Cuando se utiliza paginas de 4Mb, la direccion lineal se separa en dos partes. Una para el directorio y la otra es el offset. El funcionamiento es similar a pagina de 4kb, solo que tiene un nivel menos.

9.4.4 Traduccion de direccion lineal a fisica : Paginacion Activiada (PAE Activado)

Cuando el PSE-36 esta activado, paginas de 4Mb y 4kb puede ser accedidas desde el mismo directorio de paginas. En ejemplo de uso de mezclar paginas de 4mb y 4kb, es por ejemplo poner codigo del kernel en pagina de 4Mb que son muy frecuentemente usadas y dejar las paginas de 4kb para programas y tareas, esto mejora la performance ya que hay una TLB para paginas de 4Mb y 4Kb distintos.

9.5 Un ejemplo de administracion de memoria : Linux

Linux utiliza pagina de 3 niveles y segmentacion, cada pagina tiene un tama;o fijo de 4kb. El espacio de direcciones lineal esta dividido en dos partes :

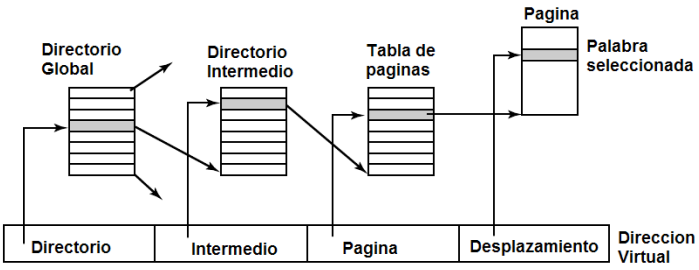
- direccion cero hasta la 0xbFFFFFFF solo puede ser direccionada cuando el proceso es de usuario o en modo kernel
- direccion cero hasta la 0xFFFFFFFF solo puede ser direccionada cuando el proceso se encuentra en modo kernel.

Los tres niveles que utiliza linux para paginacion son los siguiente :

- Directorio Global : Cada proceso debe tener solo una entrada que debe estar en memoria y ocupa una pagina
- Directorio intermedio : Puede ocupar varias paginas. Cada entrada se;ala una tabla de paginas
- Tabla de paginas : Puede ocupar varias paginas. Cada una de sus entrada hace referencia a la pagina virtual requerida.

Las direccion se partes en 4 partes (solo para 64bits), cuando el direccionamiento es de 32bits se omite el directorio intermedio.

- Directorio
- Intermedio
- Pagina
- Offset



10 Administracion Tareas (Modo Protegido)

Una tarea es una unidad de trabajo que el procesador puede despachar,ejecutar y suspender.

10.1 Estructura de una tarea

Una tarea esta compuesta de hasta dos partes : Un espacio de ejecucion (registros y todo eso) y un segmento de estado de la tarea (TSS).

10.1.1 Espacio de ejecucion

Consiste en un segmento de codigo,un segmento de pila y uno o mas segmentos de datos. Si se usa proteccion, adicionalmente en el espacio hay mecanismos para separar la pila para cada nivel de privilegio

10.1.2 Segmento de estado de la tarea (TSS)

El TSS especifica los segmentos que hacen el contexto de una tarea y provee un lugar para almacenar informacion del estado. En multitareas, el TSS provee un mecanismo para linkear tareas. Una tarea se identifica por el segmento selector del TSS. Cuando una tarea es cargada para ejecutarse por el procesador, el segmento selector es cargado en el registro TR (en el registro hidden asociado al TR se guarda el descritor de segmento)

Si se utiliza paginacion , la direccion base del directorio de paginas es guardado en el registro de control $CR3$.
Posee los valores para la pila (para usar en SS y ESP) para los niveles $0, 1, 2$, el de nivel 3 estara en $SS : ESP$.
El TSS contiene tambien la base del mapa I/O que es de 16bits y representa un offset en el segmento TSS. En el modo protegido las tareas tiene restriccion respecto a las instrucciones de entrada y salida, esto se puede cambiar

10.2 Task Switching

Existen cuatros formas de hacer un switch entre tares :

- Se ejecuta un CALL o JMP a un descritor TSS en la GDT
- Se ejecuta un CALL o JMP a un descritor de gate
- Una interrupcion o excepcion apuntar un descritor en la IDT
- Se ejecuta un $IRET$ cuando el flasg NT (Nested Task) en los $EFLAGS$ esta seteado

Los pasos que realiza el procesador son los siguientes :

- Se obtiene el selector de TSS para la nueva tarea desde los operandos del $CALL$ o JMP,task gate o desde el link previo

- Se chequeo que la tarea actual (la vieja, todavia no se hizo el sqitch) permite hacer el switch a la nueva tarea. El CPL de la actual (vieja) tarea debe ser menos o igual al RPL del selector de segmento y DPL del descritor de segmento. Interrupciones, Excepciones (exceptuando las generadas por INT n) e $IRET$ no hacen el chaqueo del DPL. Para interrupciones generadas por INT n se realiza el chequeo del DPL.
- Se chequea que el descriptor TSS esta presente y tiene un limite valido
- Se chequea que la nueva tarea esta libre (call,jmp,excepcion o interrupcion) o esta ocupada ($IRET$)
- Se chequea que el TSS de la vieja y nueva tarea,tambien de todos los descritor de segmento estan paginados.
- Si el switch se realizo de un JMP o un $IRET$ el procesador limpia el flag B en el TSS de la actual tarea (la vieja). Si el switch se realizo desde un CAL, interrupcion o excepcion el flag se deja seteado
- Si el switch de la tarea se inicio con un $IRET$ se limpia el flag NT en la imagen temporario de los $EFLAGS$. De lo contrario el flag NT se deja como estaba
- Se guarda el estado de la actual (vieja) tarea en el segmento TSS.El procesador encuentra la base del TSS en el TR (task register) y copia en la TSS : todos los registros de proposito general,selectores de segmento de los registros de segmento,la imagen temporario de los $EFLAGS$ y el puntero de instruccion (EIP)
- Si el switch fue iniciado por un $CALL$, una excepcion o una interrupcion el procesador setea el flag NT en los $EFLAGS$ cargados de la nueva tarea. Si se inicio con un $IRET$ o JMP ,el flag NT va a reflejar el estado del NT en los $EFLAGS$ cargados de la nueva tarea.
- Si la tarea se llamado desde un CALL,excepcion o una interrupcion se setea al flag B en el descritor TSS de la nueva tarea. Si fue iniciado por una instruccion $IRET$ el flasg se deja seteado.
- Carga el registro de segmento con el segmento selector y descritor del nuevo TSS.
- El estado del TSS es cargado al procesador. Esto iclute el $LDTR$, el $PDBR$ (registro de control $CR3$), $EFLAGS$, EIP, registros de proposito general, y los selectores de segmento.
- Los descriptores asociados a los selectores de segmento son cargados. Cualquier error asociado a cargar esto ocurre en el contexto de la nueva tarea
- se comienza la ejecucion de la nueva tarea.

Importante : entre los pasos 1 al 11 ,estan como existiera una transaccion. Osea que cualquier paso que falle se vuelve al punto desde donde se hizo la llamada para cambiar de tarea, como si no se hubiese ejecutado nada. En

el punto 12, al cargar los datos desde la TSS se corrompe el estado de la arquitectura, pero el error se maneja en el estado anterior al cambio de tarea (osea todavia se esta en la vieja tarea). Si ocurre algun fallo en el paso 13, el procesador cambia de tarea y genera la excepcion antes de iniciar la ejecucion de la nueva tarea.

El estado de la actual tarea se guarda siempre que se cambia a la neuva tarea satisfactoriamente. Si la tarea se habia pausado, al volver se reinicia desde el punto con el valor del EIP guardado.

Cuando ocurre un cambio de tarea el nivel de privilegio no se hereda de la tarea suspendida. La nueva tarea ejecuta con nivel de privilegio indicado en el CPL en el registro CS, que se cargo de la TSS.

10.3 Linkeo de Task

El campo link en el descritor TSS junto con el flag NT en los $EFLAGS$ sirven para voler a la ejecucion de la tarea llamadora. El flag $NT = 1$ indica que fue llamada la tarea actual por otra.

Cuando un $CALL$, interrupcion o excepcion ocurre, el procesador guarda el selector de segmento de la TSS en el campo link de su TSS Cuando se ejecuta un $IRET$ el procesador verifica el flag NT y si vale 1 cargar el selector del campo link.

10.3.1 Prevencion reentrada recursiva

U TSS permite setear el flag B (busy) para evitar que cuando la tarea fue suspendida entre otra y cambie el estado del TSS.

10.4 Espacio de direcciones

El espacio de direcciones de una tarea consiste en los segmentos que esta puede acceder. Estos segmentos incluyen los segmentos de codigo, datos,stack y segmentos de sistema referenciados en la TSS. El campo de segneto para la LDT en la TSS puede ser usar para darle a cada tarea una LDT. Recordar que cuando se hace el cambio de tarea, cuando se levantan las cosas del segmento TSS se levanta tambien el valor del $LDTR$

10.4.1 Mapeo de tareas en el espacio lineal

Existen dos formas :

- Un solo espacio lineal de direccion es compartido por todas las tareas : Sin paginacion, no existe otra forma y tiene que ser asi. Si se utiliza pagina un directorio de paginas se utiliza para todas las tareas.
- Cada tarea tiene su propio espacio lineal de direccion : Para lograr esto, gracias al TSS se guarda el $PDBR$ (que contiene el $CR3$) y permite que cada tarea tenga su propio directorio de paginas, ya que al cargar o cambiar de tarea se cambia el valor del $CR3$ (que apunta al directorio de paginas)

10.4.2 Como compartir informacion

Para compartir datos entre tareas, se deben usar las siguientes tecnicas :

- GDT : Todas las tareas deben tener acceso a los descritores de segmento de la GDT. Si algunos descriptores de segmentos de la GDT apuntan a segmento en el espacio lineal que son mapeados a la misma area fisica.
- LDT : Dos o mas tareas pueden usar la misma LDT, si el campo de la LDT en el TSS apunta a la misma LDT.Si algun descritor de segmento posee apuntan a una misma direccion fisica entonces ese segmento puede ser usado para compartir entre las tareas.
- Por descritores en distintas LDT que sus direcciones lineales mapean las misma area de direcciones fisicas. Es el metodo mas selectivo de todos.

11 Interrupciones y excepciones (Modo Protegido)

En el modo real, la IDT almacena vectores de interrupcion mientras que en el modo protegido almacena descriptores. En cualquier modo las interrupciones se identifican con un numero de un byte llmado tipo.

Cuando se genera una interrupcion o se detecta una excepcion,se pausa la ejecucion actual del programa,y se comienza la ejecucion de un manejador preparado para atender la interrupcion o excepcion . Para que el procesador sepa que hacer con cada interrupcion o excepcion posee la IDT de 256 posiciones. La IDT , al igual que la GDT y LDT posee entradas de 8 bytes que son descriptores IDT.
Los descritores de interrupcion a diferencia de los descritores de segmento poseen un selector de segmento (ademas de otras cosas).

11.1 Origen de las interrupciones

las interrupciones puede provenir de dos lugares, externas o por software. Las interrupcioens externas provinen de los pines, y en estos pines suele estar conectada el $APIC$, quien se encarga de determinar el numero de interrupcion.

11.2 Interrupciones Enmascarables

Toda interrupcion que proviene del pin $INTR$ por medio de la $APIC$ es llamada interrupcion enmascarable.

11.3 Interrupciones por software

Las interrupciones por software son aquellas que se generan por medio de la instruccion INT, y se utiliza el descriptor indicado por parametro. Este tipo de interrupcion no es enmascarable. Si se utiliza al vector NMI del procesador, este no realizara las misma operacion que si fuese una interrupcion NMI. Por ejemplo,en las interrupciones por externas en el stack el codigo de error, al realizar la int por soft este codigo de error no se pone en el stack. Luego el

handler hace un pop del EIP (en lugar del codigo de error) y se rompe todo cuand se hace el $IRET$

Los descriptores IDT pueden ser del tipo :

- Descriptor task-gate
- Descriptor interupt-gate
- Descriptor trap-gate

Los tres tipos tien el flag $S = 0$, osea son de sistema.

Los dos ultimos son similares, poseen selectores de segmento mientras que el task-gate posee un selector de segmento TSS

Al igual que con la GDT y LDT, hay un registro para la tabla IDT llamado $IDTR$ que es similar al registro GDTR (posee la base y el limite) Una diferencia de estos descriptores de interrupcion contra los descriptores de segmento es que estos en lugar de una base poseen un selector de segmento. Con este selector, buscan en la LDT o GDT la entrada al descrtior de segmento del manejador. Con el selector de segmento en el descriptor de interrupcion, se accede a la GDT o LDT para acceder al desscritor de segmento, luego este se continua como siempre usando la base del decriptor (el de la GDT o LDT), y junto con el offset del descriptor de interrupcion se accede a la rutina.

El procesador provee de dos tipos de interrupciones para detener la ejecucion de un programa :

- Interrupcion : es un evento asincronico tipicamente activado por algun dispositivo I/O
- Execpcion : es un evento sincronico que es generado por el procesador cuando se detecta una o mas condiciones. Se diferencian tres tipos : fault,abort y traps.

11.4 Excepciones

Tipos de excepciones

- Fault: Es una excepcion que en general puede corregirse y que una vez corregido el programa puede ser reiniciado sin perdida de la continuidad. Cuando un fallo ocurre, el procesador recupera el estado de antes de ejecutar la instruccion que fallo (asi cuando se recupera de la excepcion se vuelve a ejecutar esa instruccion como si nada paso, pensar en el Page fault)
- Traps: Es una excepcion que se reporta luego de ejecutar una instruccion trapping (por ejemplo : INTO). Permite seguir ejecutanto el programa sin perdida de la continuidad y la ejecucion del programa se retoma despues de la instruccion trapping. Cuando un trap se detecta mientras se ejecutant un JMP el puntero de retorno apunta al destino del JMP y no a la siguiente instruccion del JMP

- Abort: Es una excepcion que no siempre se reporta en la ubicacion de la instruccion que genero la excepcion y no permite reiniciar el programa o tarea que la genero. Se utiliza para reportar errores importantes como errores de hardware, inconsistencia o valores invalidos en tables de hardware.

Para todas las interrupciones o excepciones, salvo para el abort, se garantiza que son reportadas de tal forma que el reinicio del programa se realiza sin perdida de continuidad. Para asegurar el reinicio, el procesador guarda la informacion (registros,stack pointer,etc) necesaria para asegurar el reinicio en forma transparente. En los faults, la intruccion que realizo la excepcion se reinicia como si nunca se hubiese ejecutado. En los traps, cuando se termina de manejar la excepcion se remota la ejecucion del programa en el siguiente instruccion que genero la excepcion.

11.5 Habilitando/Deshabilitando Interrupciones

Cuando el procesador recive una interrupcion, se revisa el flag IF. Si el flag $IF = 0$ el procesador inhibe las interrupciones, caso contrario las permite. Las instrucciones para cambiar el flag son STI y CLI

12 Administracion MultiProcesador

13 APIC

14 Administracion del procesador e inicializacion

14.1 Memory Type Range Registers (MTRRS)

Permiten especificar el tipo cacheo (o no cacheo) de ciertas porciones de direcciones fisicas. Con estos registros se pueden optimizar para varios tipos de memorias como RAM,ROM, frame buffer e I/O mapeado a memoria. En general esto lo hace el BIOS y cuando se inicia todos estan en cero (que indica no cacheo).

14.2 Inicializacion para el modo real

Luego de un reset de hardware, el procesador comienzo en modo real y ejecuta el codigo en la direccion FFFFFFFF0H. EL software debe setear las estructuras basicas, como la IDT de modo real. Si el procesador continua en modo real, debera seguir cargando rutinas del S.O. Si el procesador va a cambiar a modo protegido, debera cargar las estructuras necesarias para dicho modo y luego cambiar al modo protegido.

14.3

14.4 Inicializacion para el modo protegido

Luego de un reset , el procesador esta en modo real y antes de pasar a modo protegido se necesita cargar las estructuras de datos minimas para soportar el modo protegido. Por ejemplo : $IDT, GDT, TSS, LDT(Opcional)$,si se usa pagina un directorio y una tabla de paginas,etc. Tambien la rutina debe cargar el $GDTR$,$IDTR$, registros de control del $CR1$ al $CR4$, apartir del Pentium 4 los $MTRRS$.

Con todas las estructuras de datos, modulos de codigo y registros de sistema inicializados, el procesador puede cambiar a modo protegido seteando el bit PE en el registro $CR0$.

Para procesadores Intel 64 e IA-32,hay requerimientos distintos y se seguierien algunos pasas en el manual e incluso un ejemplo (chamuyo para decir eh yo lei el libro de intel).

14.4.1 Incializacion Paginacion

Para inicializar la paginacion el bit PE del registro de control $CR0$ debe ser prendido, pero antes se deben inicializar las estructuras para soportar paginacion. Paginacion SOLO funciona en modo protegido, asi que es necesario primero estar en modo protegido

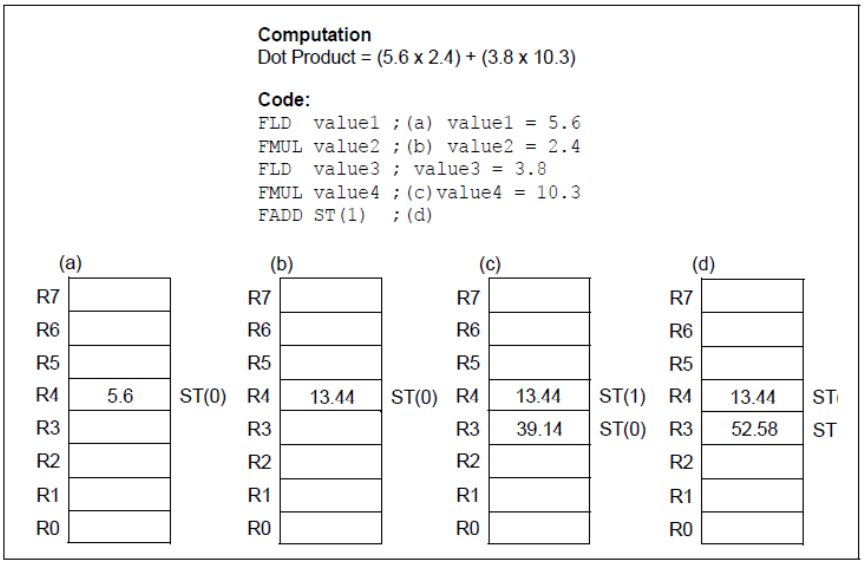
- Se debe cargar un directorio de paginas y una tabla de paginas. La tabla de paginas se puede eliminar si existe una entrada en el directorio de paginas que apunta a si mismo.
- El registro de control $CR3$ (usualmente llamado $PDBR$) se debe cargar con el puntero al directorio de paginas.
- (Opcional) El soft puede cargar en la GDT o LDT un conjunto de descriptores de segmento para un supervidor u otro conjunto de usuarios.

14.4.2 Inicializacion Multitarea

15 Control Memoria Cache

16 FPU

La forma de manejar la FPU es por medio de una pila. Esta pila consiste en 8 registros. El tope se indica por medio del registro *status*. El tope de la pila se puede referenciar por $ST(0)$ o implicitamente. La puede puede hacer tanto stack overflow como stack underflow.



16.1 Representacion

- Un bit de signo
- Una mantisa. Como por lo general los numeros estan normalizados la parte entera vale 1 y suele ser un valor implicito en la mantisa.
- Un valor para el exponente.

Una de las principales ventajas de operar en punto flotante es que se eliminan los problemas relacionados con las escalas (????) El Standard $IEEE754$ para punto flotanto es el mas ampliamente utilizado. Tiene especificaciones para 32,64 y 80bits.

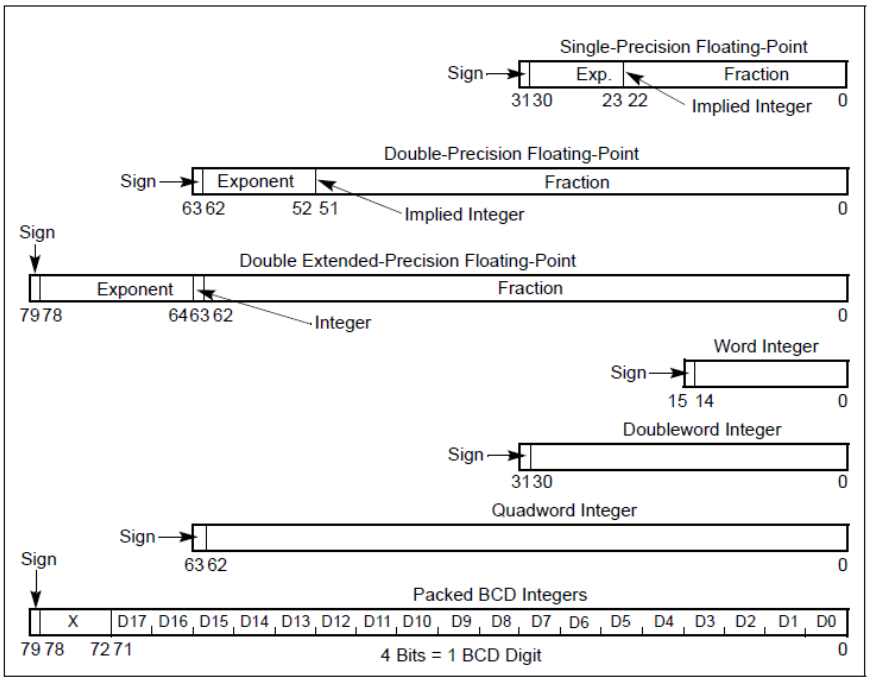
Existen ciertos valores en la codificacion que son especiales.

- Ceros Asignados
- Numeros finitos de-normalizados
- Numeros finitos normalizados
- Infinitos signados
- NaNs
- Numeros indefinidos

16.2 Tipos de datos que soporta la FPU

La FPU soporta los siguientes tipos de datos, single precision, double presicion, BCD y entero. Es importante saber que la FPU intermanete convierte todo al tipo de datos indicado en el registro de control. Por default se utiliza double presicion (64bits).

Es comun utilizar la FPU para hacer conversiones de tipos de datos, especialmente cuando se tiene el numero en presicion o doble presicion desnormalizado.



16.3 Saltos condicionales

Para poder realizar saltos condicionales con la FPU, existen dos formas (o mecanismos)

16.3.1 El viejo mecanismo

Luego de realizar la operacion con la FPU, los flags de la fpu se deben bajar a eax y subir a los $EFLAGS$ (reqiere tres instrucciones). Luego se pueden manejar los flags con instrucciones convencionales.

16.3.2 El nuevo mecanismo

Para evitar todas las instrucciones del mecanismo anterior se introdujeron instruccion que permiten la manipulacion de flags (en el sentido de control de programa) directamente de FPU, asi de esta forma se realiza el chaqueo de flags una sola instruccion

16.4 Redondeo

Rounding Mode	RC Field Setting	Description
Round to nearest (even)	00B	Rounded result is the closest to the infinitely precise result. If two values are equally close, the result is the even value (that is, the one with the least-significant bit of zero). Default
Round down (toward −∞)	01B	Rounded result is closest to but no greater than the infinitely precise result.
Round up (toward +∞)	10B	Rounded result is closest to but no less than the infinitely precise result.
Round toward zero (Truncate)	11B	Rounded result is closest to but no greater in absolute value than the infinitely precise result.

17 SIMD

Se utiliza en problemas donde los calculos suelen ser repetidos en una larga cantidad de datos.

17.1 MMX

Menaja datos enteros enpacados de 64 bits. Posee 8 registros de 64 bits. Maneja distintos formatos de esos registros : bytes en pacados,words y doublewords (para las 3 con signo y sin signo). Algunas cosas importante en cuanto al uso de las instrucciones MMX :

- Los registros MMX no pueden direccionar memoria
- No se puede mezclar codigo MMX con codigo FPU ya que se utilizan los mismo registros. Ademas hay que tener cuidado si se va a iniciar algun codigo MMX y lo que hay en la FPU se deberia guardar.
- Todas las instrucciones llevan un postfijo para indicar el formato que se usa en los registros. No todas las instrucciones permiten todas las combinaciones.
- El orden en que se guarda la informacion en memoria es little endian.
- la instruccion $EMMS$ se utiliza para vaciar el contenido de los registros MMX y se debe ejecutar siempre antes de finalizar la rutina.

Las instrucciones MMX fueron pensadas para necesidades como graficos, comunicaciones que suelen ser algoritmo que se ejecutan sobre grandes cantidad de datos. Por ejemplo datos de audio usualmente se representan en words de 16bits. Mientras que los videos o graficos utilizan 8bits.

17.1.1 Saturacion y Desborde

Desborde (Wraparound) : Aritmetica clasica bada en afectar un flag durenta la operacion que genera el desborde e invertir tiempo de procesamiento en el loop

para evaluar el estado de ese flag. Aritmetica con signo *saturada* : Consiste en que cuando se alcanza un oveflow se deja el maximo valor que puede representarse (depende dle formato). Lo mismo ocurre con un overflow negativo (cuando se llega el minimo numero). Osea la saturacion respeta la representacion. Aritmetica sin signo *saturada* : Es lo mismo que saturada con signo, salvo que la representacion es distinta.

En las instrucciones para indicar aritmetica con signo saturada llevan postfijo S, cuando usan aritmetica sin signo saturada llevan postfijo US.

17.1.2 Logicas

Tratan los datos como si fuesen de 64bits, esto tiene sentido para estas instrucciones (ojo no confundir con los shift). Son PAND,PANDN,POR,PXOR. No tienen encuenta ningun tipo de empaquetamiento.

17.1.3 Desplazamiento

Desplaza a derecha o izquierd la cantidad de bits que se indique se el empaquetado indicado en la instruccion. Instrucciones shift no aritmetico: $PSLLW, PSLLD, PSSQL, PSRLW, PSR$
Instrucciones shift aritmetico : $PSRAD, PSRAD$, estas conversan el signo.

17.1.4 Comparaciones

No afectan os $EFLAGS$. Lo que hacen es setear una mascara y solo existen dos instruccion PCMPEQ y PCMPGPT, que se pueden combinar con formato byte,word y double words. Cuando la condicion se cumple cada dato del empaquetado destino se pone en FF, $FFFF$ o $FFFFFFFF$. Se suelen usar para incializar registros, por ejemplo suando PCMPEQ mm0,mm0.

Ejemplo de uso de MMX con sprites de $8x8$ bits :
En el sprite cuando los valores de los pixeles son cero, no forman parte de este y deben terminar con el color del fondo. En $mm0$ se guarda el sprite, en $mm1$ es el color de fondo.

```
PCMPEQB mm2,mm0 ; mm2 ahora tiene la mascarca
PAND mm1,mm2 ; con la mascar se ponen en cero los bits donde debe ir el sprite en el fon
POR mm0,mm1 ; se junta el fondo con el sprite
```

17.1.5 Empacado y desempacado

Las instrucciones para empacar permiten especificar el tipo de datos para empacar y como la informacion disminuye en tama;o es mas sencilla de utilizar.
La forma en que funcionan es indicando con una H (parte alta del registro) y con una L (parte baja del registro). No manejan tipos de datos y el programador debe hacer que esto sea coherente. Lo que se hace es mezclar los registros se va poniendo en el registro destino una parte intercaladas de los registros

Por ejemplo para desempaquetar con signo se suele usar la mascara que da la comparacion y se estienden los datos con este misma mascara. La mascar es

el registro que se utiliza como fuente.

IMAGEN

Por ejemplo un registro podria ser todo en cero para desempacar un entero NO signado.

17.2 SSE

Las instrucciones manejan tipo de datos de puntos flotante. Se agregan 8 registros de 128 bits. En general las instrucciones se separan en las que manejan los 4 numeros de presicion simple (*packed*) o las llamadas *scalar* que solo modifica el double word menos significado del destino. A;ade soporte para cuatro numeros de punto flotante de precision simple. Menejo de aritmetica de 64 bits. A'ade soporte para cacheabilidad, prebusqueda y operaciones de ordenamiento de memoria.

Hay 4 subgrupos de instrucciones :

- Instrucciones $SIMD$ single-presicion que opera sobre los registros XMM
- Instrucciones de manejo de estado que operan sobre el registro $MXSCR$
- Instrucciones $SIMD$ que operan con enteros de 64 bits.
- Instrucciones de cacheability control,prefetch y ordenamiento de instrucciones

Tiene un registro para control y estado llamado $MXCSR$.

17.2.1 Redondeo : Flush-to-zero

Cuando el bit 15 del registro $MXCSR$ esta activado y se produce un underflow ocurre lo siguiente :
El numero se pone en cero y conserva el signo del resultado. Se activa el flag de underflow.
Si el bit no esta seteado se ignora el bit de flush-to-zero.
Si no se enmascara el bit flush-to-zero el modo no es compatible con el standard $IEEE754$. Este bit existe dado que permite una mejora de performance al estar desactiva, con el problema de perder presicion, suele ser tolerable una condicion de overflow normalmente.

17.2.2 Instrucciones : Movimiento Datos

17.2.3 Instrucciones : Comparacion

Las instrucciones de comparacion devuelven el resultado en el operando especificado o en los $EFLAGS$

17.2.4 Instrucciones : Aritmeticas

17.2.5 Instrucciones : Logicas

17.2.6 Shuffle and Unpack

Permite intercambiar el orden de los numeros en los registros. Si se utiliza el mismo registro como operando y destino permite ponerlos en cualquier ubicacion. La instruccion que permite esto es $SHUFPS$

Para desempaquetamiento : $UNPCKHPS, UNPCKLPS$ (Unpack and interleave high/low packeck single presicion)

17.2.7 Conversiones

Existen instruccion para transoformar el tipo de datos a punto flotante (asi como se hacia en la FPU) , pero las instrucciones son mas directas (la FPU no era tan obvio el uso (o abuso) para trnaformar tipos).

En general las intruccion convierten de enteros empacados a single precision empacados y viceversa. Entero a scalar, escalar a entero.

- CVTPI2PS : Convert Packed interger to Packed single precision
- CVTSI2SS : Convert Doubleword Integer to Scalar Single precision
- CVTPS2PI : Convert Packed Single Precision to packed double words
- CVTTPS2PI : Convert with truncation packed single precision to packed doublewords integer
- CVTSS2SI : Convert scalar single precision to double word

17.2.8 Cacheabilidad,prefecth y ordenamiento de memoria

El conjunto de instrucciones SSE posee instruccion para indicar control de cache, instrucciones prefecth que permiten indicar a que nivel de cache prebuscar los datos.

17.2.9 Instrucciones para $MXCSR$

las instrucciones que estan asociadas al registro de control y estado son para guardar y restaurar su contranido $LMDMXCSR$ y $STMXCSR$.

17.3 SSE2

A;ade nuevos tipos de datos, entre ellos dos doble presicion empaquetas en $128bits$. Enteros empaquetados a byte,words,double y quad en 128 bis. Y las instrucciones de soporte para los nuevos tipos de datos

17.4 SSE3

Mas instrucciones al set de $SIMD$, son unas 13 agregan por ejemplo $MONITOR$ y $MWAIT$ que aceleran la sincronizacion de threads.

17.5 Tipos de datos

18 Mezclando codigo de 16bits con codigo de 32bits

La arquitectura IA-32 tiene los siguientes mecanismo para detectar codigo de 16bits y 32bits.

- Flag D en el descritor de segmento de codigoasm
- Flag B en el descrtior de segmento de stack
- call gates,interrupt gates y trap gates de 16bits y 32bits
- Prefijos en los operandos (66h) y prefijos para las direccion (67h)
- registros de proposito general de 16bits y 32bits.

El flag *B* espeficia que registro se utiliza implicitamente (SP o ESP). El flasg *D* indica cuando incrementar o decrementar en el registro SP o ESP. Los prefijos se utilizan para cambiar el tama;o default de los operandos.

Con todo esto, dependiendo del tipo de segmento (16 o 32bits) las instruccion pueden tener distintas interpretaciones , por ejemplo un mov MEMORIA,REGISTRO. Puede mover los 32bits dle registro a la memoria, puede mover 16bits de un registro de 16bits a usando 32bits para el direccionamiento,etc.

Siempre es mejor usar 32bits, es mas performante en los procesadores modernos.

18.0.1 Compartir datos entre segmentos

Los segmentos de datos pueden ser accedidos por segmento de codigo de 16bits y 32bits. Cuando los datos que se comparten entre segmentos de codigo de 16bits y 32bits son mas grandes que 64kb, los datos que se *deben* acceder desde los segmentos de codigo de 16bits deben ser menores a 64kb (esto es porque los punteros de 16bits solo pueden apuntar a los primeros 64kb)

Un stack que su maximo tama;o alcanza los 64kb puede ser compartido por segmentos de codigo de 16bits y 32bits. En general los stacks de 32bits son mas grandes que 64kb, por lo tanto los segmentos de 16bits no pueden usarlo almenos que el codigo de segmento se modifique para que use direccionamiento de 32bits.

18.0.2 Transferencia de control

19 Microarquitectura

19.1 Pipelines

Es una tecnica muy similar a la linea de ensamblaje. Consiste en que las instrucciones se dividen en etapas, y que una vez finalizada una etapada la siguiente instruccion puede empezar a ejecutarse cuando la anterior todavia no termino toda su ejecucion completa. Esta tecnica permite ejecutar una o mas instrucciones por ciclo.

Para que el pipeline tengo su mejor desempa;o este debera estar siempre lleno, sin embargo hay situaciones en las que esto trae problemas.
Principalmente los problemas son :

- Ramificaciones en el codigo
- Dependencias de datos

Para el problema de ramificaciones del codigo se proveen varios soluciones que tratan de mejorar la situacion. Entre ella esta la prediccion de ramas, eliminacion de ramas. Para el problema de dependencia de datos, se intentan solucionar con ejecucion fuera de orden.

19.1.1 Etapas

- Busqueda Instruccion
- Decodificacion
- Busqueda de operando
- Ejecucion
- Escritura de resultado

En general se intenta que las etapas tengan la misma cantidad de ciclos.
Las microoperaciones se pueden desagregar y generar pipelines de mas etapas (super pipelines). Otro problema que puede ocurrir, es que en la etapa de busqueda de operando pueden existir conflictos (RAW, WAR, WAW).

19.1.2 Efecto de un branch en un pipeline

El problema de un branch cuando hay pipeline, es que como no se sabe a que punto de programa la rama va a tomar, el pipeline se frena hasta que se tiene la informacion de lo que se tiene que hacer. Este problema se suele tratar de solucionar con prediccion de saltos. Sin embargo si realizar prediccion de saltos esta falla, el pipeline que se lleno con el branch que se predijo que se iba a tomar, se debe descartar lo que hizo el pipeline y llenar de nuevo con el branch tomado. Pero en general la prediccion se suele basar en cosas como que la mayor parte del codigo que se ejecutan son ciclos, etc.

19.2 Arquitectura superescalar

Consiste en multiplicar el pipeline, para tener mas recursos. Trae mas problemas en cuanto a dependencias de datos.

19.3 SuperPipeline

Consiste en subdividir las etapas de las instrucciones

19.4 Prediccion de saltos

Para tratar los saltos en los pipelines se implementan unidades de prediccion de saltos, que utilizan las siguientes tecnicas:

- Asumir que nunca se salta (estatica)
- Asumir que siempre se salta (estatica)
- Predecir por OpCode
- Salta / No salta alternativamente (estatica)
- Utilizar una memoria ultrarapida para llevar un historial de saltos. (dinamica)

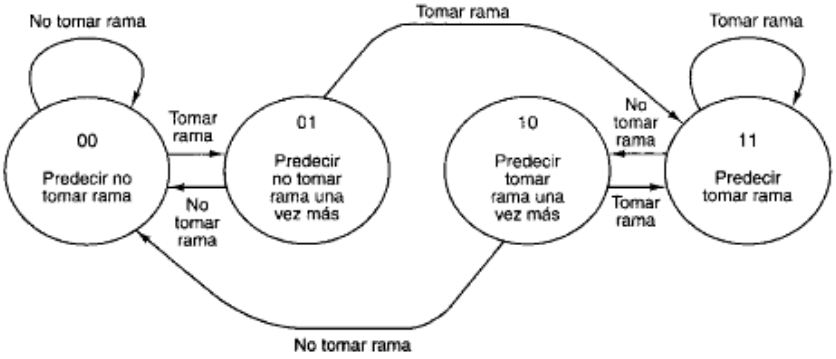
Los primeros dos son estaticos, estos no dependen la de la ejecucion del programa.

19.4.1 Prediccion siempre se salta

La prediccion consiste en que se va a realizar el salto. Es la prediccion mas simple, se basa en que la mayoria de los codigo poseen ciclos y al final de un ciclo siempre se realiza el salto. Los casos en los que falla en general son pocos, como excepciones o cuando se sale de un ciclo,etc. La prediccion nunca se salta, es similar a la de siempre se salta.

19.4.2 Prediccion dinamica

En este tipo de prediccion la CPU lleva una tabla historica en la que registra las ramificaciones condicionales conorme ocurren, para poder consultarlas cuando vuelva a ocurrir. La implementacion mas sencilla consiste en una tabla, que posee 1 bit para validez, otro bit que indica ramificacion y la direccion de la ramificacion. La prediccion consiste simplemente en que la ramificacion tomara la misma trayectoria anterior. Si esto no es asi, se cambia el valor. Un problema que tiene esta prediccion, es que nunca predice bien la ultima iteracion de un ciclo (y la proxima es no tomar la rama!) y ademas si hay un ciclo dentro de otro esto podria ocurrir mucho. Para eliminar este problema, se utiliza en la tabla dos bits para la prediccion. Entonces cuando se entra en la ultima iteracion, aunque esta falla con la prediccion la proxima vez se tomara el salto. La FSM del algoritmo es el siguiente :



Los 2 bits se pueden interpretar como, el de la izquierda como el estado de la prediccion y el de la derecha es lo que hizo la vez anterior.

19.4.3 Prediccion Estatica

En este caso el procesador en su arquitectura tiene instrucciones para predecir el proximo salto, esto junto con un compilador y un simulador se puede crear codigo estatico que se comporte de forma adecuada.

19.5 Ejecucion fuera de orden

La ejecucion fuera de orden trata de explotar las dependencias de datos para permitir ejecutar instrucciones fuera de orden. Tecnicas que se usan son renombres de registros y cambiar el orden de ejecucion de las instrucciones. Cuando la instruccion en el pipe esta en la etapa de decodificacion tiene que decidir si puede ejecutarse de inmediato o no. Para tomar esta decision la unidad necesita reconocer el estado de los registros.

19.5.1 Tipos de dependencias de datos

Las dependencias de datos son las siguientes :

- RAW (Read after Write) : Si se esta escribiendo en cualquier operando, no emitir. Es cuando una instruccion quiere leer un resultado que todavia no se guardo en el registro.
- WAR (Write after Read) : Si se estas leyendo el registro del resultado , no emitir. En esta puede ocurrir que una instruccion todavia no leyo el contenido del registro y la otra quiera pisar el valor.
- WAW (Write after Write) : Si se esta escribiendo en el registro de resultado, no emitir.

Si el procesador ingresa y retira en orden las instrucciones, esta limitado en cuanto a ejecutar alguna instruccion que no tenga ninguna de las tres dependencias. Para realizar esto el procesador lleva contadores de los usos de los registros para poder chequear las dependencias. Supongamos que el procesador permite ingreso y egreso de instruccion fuera de orden, entonces podria ejecutar instrucciones que no tienen dependencias de datos y asi aprovechar mejor los ciclos. Un problema que ocurre con este es que podria existir una dependencia indirecta, para esto se agrega mas informacion, sobre instrucciones que estan paradas. En ejecucion fuera de orden, el procesador usa registros ocultos para eliminar dependencias del tipo WAR o WAW. Por ejemplo en el siguiente codigo :
$R3 = R0 * R1$
$R4 = R0 + R2$
$R5 = R0 + R1$
$R6 = R1 + R4$
$R7 = R1 * R2$
$R1 = R0 - R2$
$R3 = R3 * R1$

$R1 = R4 + R4$

En el ejemplo al usar ejecucion en orden, la instruccion 4 dependen de la escritura de $R4$ en la instruccion 2,pero esta instruccion todavia sigue en el pipeline y para cuando llega la instruccion 4 al pipeline, todavia no se tiene el valor de $R4$ y por lo tanto esta instruccion no puede ser ejecutada. Sin embargo la instruccion 5 parece no tener dependencias y podria ejecutarse (si fuese fuera de orden) pero en ejecucion con orden esta no se ejecuta.
Otro problema del codigo es en la ultima instruccion, se utiliza $R1$ para escribir pero se esta utilizando como lectura en la anterior instruccion. En estos casos el procesador agrega un registro invisible para salvar el problema (WAR),esto se conoce como renombres de registros.

19.6 Memoria Cache

19.6.1 Tecnologias

- RAM dinamica
- RAM estatica

19.7 Implementacion memoria cache

19.7.1 Principio de vecinidad

Principio de vecinidad temporal : Si un item es referenciado, la probabilidad de ser referienciado en el futuro inmediato es alta.
Principio de vecinidad espacial : Si un item es referenciado, es altamente probable que sean referenciados sus items vecinos.

19.7.2 Estructura

Linea o bloque : Elemento minimo de palabra de datos dentro del cache. Corresponde a un multiplo del tama;o de la palabra de datos de memoria. Razon : Cuando se direcciona un item en memoria generalmente se requerira de los item que lo rodean (principio de vecinidad espacial) Tag : Indica la porcion de memoria que se halla asociada la linea.

19.7.3 Mapeo Directo

En la implementacion de mapeo directo, la memoria RAM se divide en bloques de tama;o fijo llamados lineas de cache. Por lo general una linea consta de 4 a 64 bytes.
Esta implementacion es muy rapida para la busqueda, sin embargo tiene el problema de que si un programa pide el dato A y luego el dato B, y tanto A como B estan en bloques distintos en la memoria, pero caen en la misma linea de cache, y si ademas hay un programa que pide los datos en la forma A,B,A,B,... se estaria todo el tiempo accediendo a memoria (miss).

La forma de ubicar una linea en la cache es con la cuenta : (direccion del bloque) mod (Cantidad de bloques en la memoria cache) Con esta cuenta se ve

bien el problema de esta implementacion, ya que los bits bajos son los que mas cambian o se repiten mucho.

La direccion se separa en tag, linea y posicion en la linea. La cache tiene ademas un bit para invalidar la entrada y los datos que contiene la direccion. Cuando el procesador tiene la direccion calculada, utiliza los bits de linea para ubicar la linea en la cache y utiliza el tag para comparar con lo que hay en la linea de cache.

Estas caches pueden funcionar muy bien si el codigo que genera un compilador esta conciente de que la memoria cache tiene implementacion por mapeo directo.

19.7.4 Asociativo

Un cache se dice totalmente asociativo si cualquier bloque de memoria puede asociarse a cualquier bloque de la memoria cacheadas

19.7.5 Asociativo por conjuntos

Es una mezcla de mapeo directo y asociativo, consiste en que la cache tiene conjuntos. Cada conjunto se elige tal como se hacia en mapeo directo, sin embargo dentro del conjunto cualquier bloque puede ir en cualquier ubicacion En los conjuntos se tiene que buscar en todos los elementos para encontrar el elemento (se lo busca por el tag).

19.7.6 Asociativo de dos vias

Es equivalente a asociatva con conjuntos de dos lineas. Es el tipo de implementacion minima respecto a la cantidad de lineas por conjuntos (con un conjunto es mapeo directo). La direccion se separa tambien en tres partes, un tag , set y ubicacion en la linea. La memoria cache, al indicarle el conjunto debe buscar en todo el conjunto (en este caso tiene dos posibles bloques o lineas).

Los caches asociativos por conjuntos son mejores que los de mapeo directo, y son mas faciles,economicos que los totalmente asociativos. Al incrementar las vias (cantidad de lineas o bloques por conjuntos), la probabilidad de colisiones disminuye.

Un problema, que ocurre con los caches asociativos (y por conjuntos tambien) es que en los conjuntos si al traer un dato de memoria no se encontraba en la cache hay que reemplazar el contenido de la linea (en mapeo directo esto no hace falta porque no hay nada que elegir). Si hay una linea del conjunto sin usar se elige esa, el problema ocurre cuando todo el conjunto esta lleno.

Esto se aplica para cache de datos $L1$,$L2$ y $L3$. Para cache $L1$ de codifo solo shared e invalid.

19.7.7 Algoritmos de reemplazo

- LRU : Least recently used. Se corresponde con el principio de vecinidad temporal
- LFU : Last frecently used
- Random :

- FIFO :

19.7.8 Cache Miss: impacto en pipeline

Un miss en el cache detiene el pipeline produce un atascamiento en el pipeline. Una vez recuperado el operando se requiere del tiempo de acceso a memoria (en ciclos) para volver a tener el pipeline lleno.

19.7.9 Coherencia

19.7.10 Escritura en caches

El problema de la escritura en memoria cache, es que esta difiere con los contenidos en la memoria RAM. Existen dos formas para tratar el problema, una es escribir tambien en la memoria ($write - back$) o esperar a que el algoritmo de reemplazo tenga que sacar la linea de la cache. La primera opcion es la mas facil de implementar y garantiza que la memoria esta actualizada, sin embargo es costosa en cuanto al trafico que genera. La otra trae problemas de coherencia.

Existe varios formas de actualizar los datos de cache modificados hacia la memoria ram :

- Write Throught : El controlador cache escribe en la memoria cache como en la memoria RAM cuando hay modificaciones
- Write Throught Buffered : El controlador cache actualiza la memoria cache, y mientras el procesador continua ejecutando instrucciones el controlador actualiza la memoria RAM.
- Copy back : Se marcan las linea de cache como modificadas y cuando el algoritmo de reemplaza tiene que eliminar la linea en la cache se baja a memoria.

19.7.11 Coherencia en la cache : protocolo MESI

- M (Modified) : Linea presente solamente en este cache que vario respecto de su valor en memoria del sistema (dirty). Requiere write back hacia la memoria del sistema antes que otro procesador lea desde alli el dato
- E (Exclusive) : Linea presente solo en esta cache, que coincide con la copia de memoria principal (clean)
- S (Shared) : Linea del cache presenta y puede estar almacenada en los caches de otros procesadores
- I (Invalid) : Linea de cache no valida.

19.7.12 Estructura

19.8 Microarquitectura : 386

19.9 Microarquitectura : 486

Primer procesador en implementar cache multi nivel (L1 y L2)

19.10 Microarquitectura : Pentium

Tiene un arquitectura superscalar, con dos pipelines de instrucciones, uno para punto flotante y otro para enteros.

Posee un bus externo de 64bits que garantiza la lectura de dos instrucciones a la vez. En estos procesadores Intel introdujo el controlador de interrupciones APIC.

19.10.1 Prediccion de saltos

Emplea una prediccion estatica de saltos (asume que siempre salta). Brach Target Buffer (BTB): El procesador guarda en ese buffer la direccion destino de una instruccion de branch durent la etapa de decodificacion. En el caso de que la prediccion almacenada por la unidad de decodificacion del pipeline (U o V) haya sido exacta la instruccion se ejecuta sin atascos ni flushes del pipeline. Si al momento de la evaluacion de la direccion de salto esta no coincide con la prediccion almacenada en la BTB, debe buscarse el target correcto y se flushea el contenido de los pipelines.

19.10.2 Memoria Cache

Posee una memoria cache $L1$ separada en dos partes una para datos (8kb) y otra para codigo (8kb).

19.11 Microarquitectura : P6 (Pentium Pro,Pentium II/III...)

Emplea un scheduling dinamico (ejecucion fuera de orden) Esta basado en una ventana de instruccion y no en un pipeline superscalar. Las instrucciones se traducen a micro operaciones Las microoperaciones ingresan a un pool(ventana) en donde se mantienen para su ejecucion Los tres cores tiene plena visibilidad de esa ventana de ejecucion Se aplica la tecnica de ejecucion fuera de orden y ejecucion especulativa La unidad de despacho y ejecucion mantiene el modelo superscalar y lo combina con un super pipeline de 20 etapas.

EXPLICAR COMO NO SE CORTA LA EJECUCION POR TENER OUT OF ORDER

19.12 Microarquitectura : Netburst

19.12.1 Hyper Threading

Consiste en ejecutar dos flujos de programas distintos en concurrentemene compartiendo recursos de ejecucion. HT difere de multiples procesadores en cuanto a que hay un solo procesador fisica que permite la ejecucion concurrente de threads. La tecnologia consiste en dos o mas procesadores logicos, con todo lo necesario para ejecutar los dos procesos en paralelo. Los procesadores logicos consiste en el set que provee la arquitectura (registro deproposito general, registros de segmento,etc).

19.13 Microarquitectura : Core

19.13.1 Pipeline

Esta compuesto por 14 etapas. Posee entreo otras cosas tres unidades de aritmetica logixas. Cuatro decodificadores, que permiten decodificar hasta 5 instruccion por ciclo.

DIBUJITO DE LOS STAGES

19.13.2 Prediccion de saltos

La uunidad de prediccion de saltos, comienza a ejecutar el branch mucho antes que la decision de tomar el branch es realizada. Todas las ramas utilizan la unidad de prediccion (BPU).

La BPU posee las siguientes features :

-
-
-

La BPU realiza los siguientes tipos de predicciones :

- Llamados y saltos directos (CAL,JMP,etc) :
- Llamados y saltos indirectos :
- Saltos condicionales :

19.13.3 Unidad Instruction Fetch

19.14 Microarquitectura : i7

20 Optimizacion

20.1 Optimizacion de prediccion de saltos

Algunas optimizaciones que ayudan a predecir saltos :

- Mantener el codigo en paginas distintas (paginas de codigo y paginas de datos).
- Eliminar saltos siempre que sea posible
- Desarmar ciclos para hacer que la maxima cantidad de iteraciones sea menos que 16 o menos iteraciones (evitar ciclos de tama;o excesivo)

20.1.1 Eliminacion de ramas

Existen dos reglas para la eliminacion de codigo

- Intentar mantener el codigo en bloques contiguos
- Usar las instruccion CMOV y SETCC para eliminar saltos innecesarios

Ejemplo de optimizacion (de un codigo en C):

```
X = ( A < B ) ? CONST1 : CONST 2;
```

Un codigo equivalente en assembler puede hacer imposible la prediccion de saltos, como el siguiente:

```
cmp a,b
jbe L30
mov ebx const1
jml L31
L30:
  mov ebx,const2
L31:
```

El siguiente codigo, es equivalente y esta optimizado para eliminar ramificaciones

```
xor ebx,ebx
cmp A,B
setge bl ;
sub ebx,1 ; ebx - $11\dots11$ o $00\dots00$
and ebx,CONST3 ; CONST3 = CONST1 - COSNT2
add ebx.CONST2 ; ebx = CONST1 or CONST2
```

La idea es similar a la que se usa en MMX, se trata de armar una mascarca al hacer $sub\ ebx, 1$.

Otra forma de eliminacion de ramas, es utilizando la instruccion CMOV o FCMOV

Ejemplo :

```
test ecx,ecx
jne 1H
mov eax,ebx
1H:
```

Codigo optimizado :

```
test ecx,ecx
cmoveq eax,ebx
```

20.2 Loop Unrolling

Algunos beneficios :

Algunos procesadores puede predecir ramas de manera optima si la cantidad de iteracion es menor a 16.
Se pueden encontrar optimizacion que antes no se veian, por ejemplo :

```
for i in (1..100) :
  if ( n mod 2 == 0 ) a(i) = x
  else a(i) = y
```

```
for i in (1..50) :
  a(i)=x
  a(i+1)=y
```

Al desarrollar la iteracion puede esconder latencias gracias al pipeline.

21 Apendice : Compilacion en ASM y C

Primero crear los archivos objetco , ya sea de C y ASM. Para C :

```
gcc -c -o main.o main.c
```

Para ASM :

```
nasm -felf codigoasm.asm
```

Ahora hay que linkear todo (con gcc):

```
gcc -o main main.o codigoasm.o
```

22 Apendice : ASM y svgalib

23 Apendice : Medicion de performance

24 Apendice : Bochs para debug de codigo

Hay que bajar el source para activar el debugger. Configurarlo con :
./configure –enable-debugger –enable-disasm

25 Apendice : Aritmetica numeros multiplicacion

Una forma de resolver el problema es hacer lo mismo que se hace cuando se multiplica a mano. Osea agrupar, ir corriendo a medida que se avanza en las multiplicaciones de la agrupacion. Es dificil o molesto el codigo, porque cuando se hace a mano se suma todo al final mientras que en el codigo se multiplica y se suma. Hay que tener en cuenta que el resultado de la multiplicacion ocupa el doble de espacio. Ejemplo multiplicacion dos numeros de 64bits sin signo. Para resolverlo conviene separar las partes de las cuenta y despues armar en codigo en partes.

```
 ; void* producto(long long int a,long long b)
  %define BH [ebp+20]
  %define BL [ebp+16]
  %define AH [ebp+12]
  %define AL [ebp+8]
  %define RES [ebp-4]

extern malloc
global producto
producto:
  push ebp
  mov ebp,esp
  sub esp,4
  push eddi
  push esi
  push ebx

  mov ebx,16
  push ebx
  call malloc
  add esp,4
  mov RES,eax ; en RES esta el puntero al resultado

  mov eax,AL
  mov ebx,BL
  mul ebx  ; edx:eax = AL*BL

  mov ebx,RES
  mov [ebx],eax ; la primera parte del resultado ya esta lista
  mov ecx,edx ;en ecx esta lo que voy a tener que sumar a la siguiente agrupacion

  mov eax,AH ;
  mov ebx,BL ;
  mul ebx  ; edx:eax = AH*BL

  xor esi,esi
  add ecx,eax ; suma parte alta de AL*BL con parte baja de BL*AH
  adc esi,0  ; este carry se guarda para la proxima suma
```

```
mov edi,edx ; edi = parte alta de de bl*ah

mov eax, AL
mov ebx, BH
mul ebx

add ecx,eax
adc esi,0
mov ebx,RES
mov [ebx+4],ecx

add edi,esi
xor esi,esi
adc esi,0
add edi,edx
adc esi,0

mov eax,AH
mov ebx,BH
mul ebx

add esi,eax
adc edx,0
```

26 Apendice : Extension de signo en MMX