PRÁCTICAS DE OPTIMIZACIÓN DE CÓDIGO

Julio Ortega Lopera
Departamento de Arquitectura y Tecnología de Computadores

Práctica 3.

Optimización de Código

ÍNDICE

1. INTRODUCCIÓN

En estas prácticas se estudiarán distintas alternativas para optimizar código, teniendo en cuenta las características de la arquitectura y microarquitectura de computador donde se van a ejecutar. Se considerará, fundamentalmente la implementación de la arquitectura x86 ó IA32 (Intel) en los microprocesadores con microarquitecturas P6 (Pentium Pro, II, y III) ó Netburst (P4) pero también se pueden considerar otras microarquitecturas, como las de AMD.

Para realizar las prácticas se utilizará el compilador de C gcc para DOS/WINDOWS que se puede obtener en la dirección www.delorie.com/djgpp/ o la alternativa gcc de Minimalistic GNU for Windows en www.mingw.org/. En todo caso, si se quieren emplear las opciones de optimización para micros x64 (Intel o AMD) deben emplearse versiones de GCC (4.7 o superior). Otra alternativa interesante es el compilador de C++ de Intel y herramientas como VTune (se pueden conseguir versiones de evaluación en las direcciones http://software.intel.com/en-us/c-compilers y http://software.intel.com/en-us/intel-vtune-amplifier-xe).

En esta sección, se describen algunas de las alternativas para optimizar código. Se pretende poner de manifiesto la importancia que en el proceso de optimización tiene el conocimiento de la arquitectura y la microarquitectura del computador donde se ejecuta el programa. Si bien es posible escribir programas correctos para un computador a partir del modelo de programación propio del lenguaje de alto nivel que se utiliza, y sin conocer prácticamente nada de las características de la máquina, la generación de código eficaz, que saque el máximo partido de las características del computador, requiere que se tengan en cuenta detalles de la arquitectura (realizando ciertas optimizaciones a nivel ensamblador) y de la forma en que se ejecutan las instrucciones máquina (características de la microarquitectura).

El compilador permite realizar ciertas optimizaciones que se pueden utilizar sin conocer el tipo de transformación que consideran. Así, en el caso del compilador *gcc* se tienen las opciones *–O1, -O2, -O3,* y *–Os* que permiten obtener códigos con distintos niveles de optimización.

Las posibles optimizaciones se pueden clasificar como:

- Optimizaciones desde el lenguaje de alto nivel (OHLL)
- Optimizaciones desde el lenguaje ensamblador

(OASM) Además, se pueden considerar:

- Optimizaciones aplicables a cualquier procesador (OGP).
- Optimizaciones específicas para un procesador determinado (OEP).

Todos los ejemplos de optimización OEP proporcionados en este guión están basados en los procesadores Pentium Pro, II y III fundamentalmente, dada la mayor disponibilidad de información sobre su diseño. Estos ejemplos se proporcionan con un propósito didáctico y no tienen porque que suponer el entorno de optimización de los ejercicios prácticos (salvo que el alumno disponga de un microprocesador de estas familias para la realización de las

prácticas).

A continuación se indican algunas optimizaciones posibles teniendo en cuenta:

- La captación y envío a decodificación.
- La decodificación y el renombrado de registros.
- La ejecución de instrucciones y las operaciones en coma flotante
- El acceso a memoria principal y cache.
- La predicción de salto

El conocimiento de las opciones posibles para optimizar código parte necesariamente del estudio de las características de la arquitectura y microarquitectura del procesador, así como la organización de la jerarquía de memoria, considerando los niveles existentes de memoria cache, la forma de mapear la memoria principal en la memoria cache, etc. Para ilustrar estas cuestiones, aquí se hará una descripción resumida del cauce de los procesadores Pentium Pro, Pentium II, y Pentium III. A partir de dicha descripción se detallarán algunas estrategias para optimizar código. Las mismas estrategias se pueden aplicar en otras microarquitecturas (una vez conocidos los detalles necesarios de las mismas).

1.1. La microarquitectura de los Pentium Pro, Pentium II y Pentium III.

En la Figuras 1.1, 1.2, y 1.3 se muestran, respectivamente, un esquema con las etapas del cauce, las unidades de captación y decodificación, y las unidades de emisión y ejecución del Pentium II (aunque lo que se va a explicar respecto a la optimización es válido tanto para el Pentium Pro como para el Pentium III, a no ser que se indique explícitamente). La operación del Pentium II se resume en los siguientes pasos:

1. El procesador capta las instrucciones de memoria en el orden en que están en el programa.
2. Cada instrucción se traduce a una o más instrucciones de tipo RISC llamadas micro-operaciones (Intel)
3. El procesador ejecuta las micro-operaciones (fuera de orden) en la microarquitectura superescalar.
4. El procesador actualiza los resultados obtenidos al ejecutar las micro-operaciones en los registros del procesador siguiendo el orden del flujo de programa original.

Figura 1.1 Etapas del cauce del Pentium II

El número de etapas del cauce es como mínimo 11, aunque en algunos casos las micro-operaciones necesitan varios ciclos de ejecución, dando lugar a un cauce más largo incluso. Recuérdese que el Pentium tenía un cauce de sólo 5 etapas. A continuación se describen con algo más de detalle las etapas del cauce.

a) Captación (tres etapas):

- **IF1** capta instrucciones de la cache de instrucciones: Una línea (32 bytes) cada vez.
- **IF2** recibe instrucciones desde la etapa IF1, a razón de 16 bytes cada vez. Esta unidad hace dos cosas en paralelo: (1) explora los bytes para determinar los límites de las instrucciones, y (2) si alguna de las instrucciones es de salto, pasa la dirección de la instrucción al predictor de salto dinámico.
- **IF3** alinea las instrucciones recibidas de IF2 (16 bytes cada vez) para pasarlas al decodificador.

b) Decodificación (dos etapas):

- **ID1** puede manejar tres instrucciones máquina del Pentium Pro en paralelo y traduce cada instrucción a (de una a cuatro) micro-operaciones de tipo RISC de 118 bits (esta longitud es necesaria para poder codificar las instrucciones del repertorio máquina que son muy complejas). Tiene tras decodificadores D0, D1, D2. Existe un secuenciador (MIS: Microcode instruction sequencer) que genera la secuencia de micro-operaciones en el caso de instrucciones máquina complejas que necesiten 5 o más micro-operaciones (se trata, por tanto de una unidad microprogramada).
- **ID2** recibe micro-operaciones desde ID1 (a razón de hasta 6 al mismo tiempo) y las encola según el orden del programa. Si alguna micro-operación es un salto se pasa a la unidad de predicción de salto estática que la pasa a la unidad de predicción de salto dinámica.

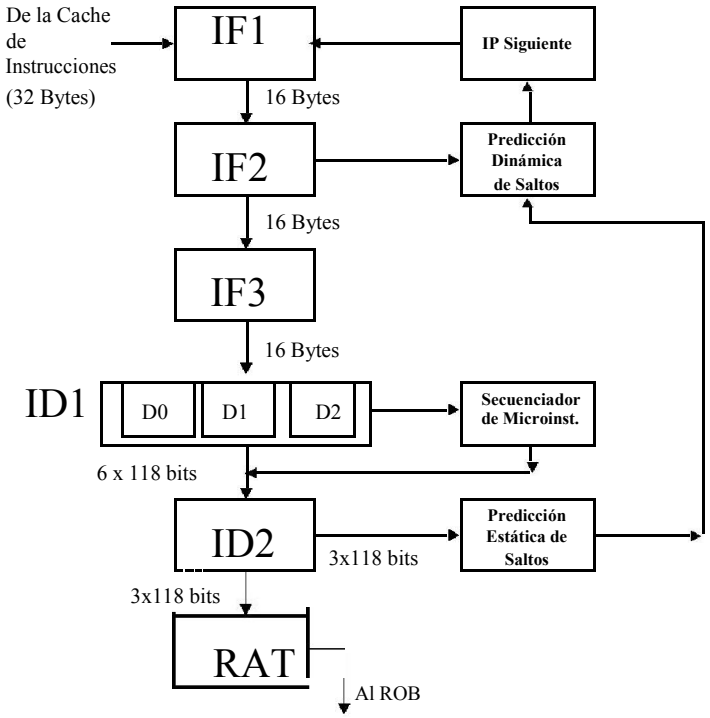

Figura 1.2. Unidades de captación y decodificación del Pentium II

c) Renombrado de Registros:

En esta etapa se realiza en el RAT ('Register Allocator') que reasigna las referencias a los 16 registros de la arquitectura (los 8 de coma flotante más EAX, EBX, ECX, EDX, ESI, EDI, EBP) a 40 registros físicos. Elimina las dependencias falsas (WAW, WAR).

El RAT no puede manejar más de tres uoperaciones por ciclo de reloj, por lo que no puede tener un rendimiento de más de tres uoperaciones por ciclo de reloj. Se pueden renombrar tres registros por ciclo de reloj (incluso se puede renombrar el mismo registro tres veces por ciclo).

Una limitación importante es que sólo se pueden leer dos registros (de la arquitectura) diferentes cada ciclo (no cuentan los registros que sólo se utilizan para escritura. Por ejemplo, en la secuencia de instrucciones:

```
mov [edi+esi],eax
mov ebx,[edi+esi]
```

la primera instrucción genera dos uoperaciones, y se leen los registros *eax*, *edi*, y *esi*, y la segunda instrucción genera una uoperación que lee de *edi* y *esi*, y escribe en *ebx*. Como se leen más de dos registros diferentes, si las dos instrucciones entran a la vez en el RAT se necesitan dos ciclos. En cambio, en la secuencia de instrucciones siguiente sólo se necesita un ciclo porque se realizan lecturas de dos registros (*edi*, y *esi*):

```
mov [edi+esi],edi
mov ebx,[edi+esi]
```

No obstante, hay que tener en cuenta que si la lectura de un registro se hace desde un registro del RAT (en lugar de hacerse desde el banco de registros) porque se ha renombrado ese registros en el RAT, entonces ese registro no se cuenta para establecer la limitación de dos lecturas que se ha indicado antes. En la secuencia siguiente

```
mov eax,ebx
sub  ecx,eax
inc ebx
mov edx,[eax]
add esi,ebx
add esi,ecx
```

todas las instrucciones generan una uoperación. Si entran en el RAT las tres primeras, se pueden hacer las lecturas en un ciclo porque sólo se hacen lecturas de *ebx*, y *ecx* en el banco de registros. La lectura de *eax* se hace desde el RAT puesto que al escribirse antes en *eax*, se habrá hecho el correspondiente renombrado del mismo en el RAT. En las siguientes tres instrucciones, las correspondientes tres uoperaciones también realizarán sólo dos lecturas de registros en el banco de registros (*edx*, y *esi*), y por lo tanto sólo necesitan un ciclo para procesarse en esta etapa.

Las uoperaciones pasan desde los codificadores al RAT a través de una cola, y desde el RAT pasan al Buffer de Reorden (ROB) y a la estación de reserva.

d) Buffer de Reorden:

Este buffer circular puede almacenar hasta 40 microoperaciones. Tiene los campos típicos de Estado, Dirección de Memoria de la instrucción del Pentium, Micro-operación, Registro hardware asignado al resultado ('Alias Register').
Las micro-operaciones se introducen el el ROB en orden, pero se emiten desordenadamente a medida que los operandos y la unidad funcional necesaria están disponibles. Al final, las instrucciones se retiran del ROB en orden.

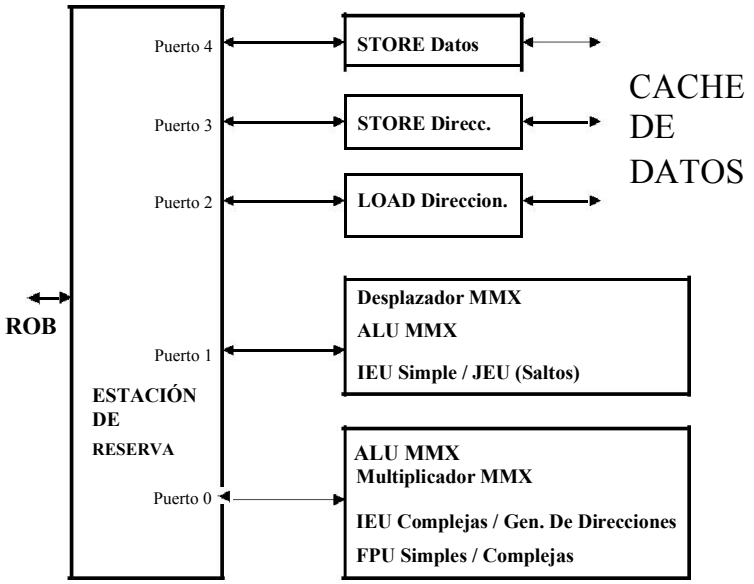

Figura 1.3. Unidades de emisión y ejecución del Pentium II

e) Unidad de Emisión/Ejecución:

Se dispone de una estación de reserva (RS) que recibe las micro-operaciones del ROB, las emite a las unidades funcionales, y devuelve los resultados obtenidos al ROB. La RS capta las micro-operaciones del ROB cuando los operandos de la misma y la unidad funcional utilizada está disponible.

La RS puede emitir hasta 5 micro-operaciones por ciclo. Si hay más micro-operaciones disponibles se emiten según el orden en el que están en el ROB. Hay cinco puertos en la RS asociados cada uno a diferentes unidades funcionales.

Si hay un salto que se ha predicho erróneamente deben eliminarse las correspondientes micro-operaciones de las etapas en las que se encuentren. Esto lo hace la JEU (Unidad de Ejecución de Saltos) que comprueba si el resultado de salto es el predicho. Si no coinciden se deshabilitan todas las microoperaciones que hay detrás del salto. Entonces el predictor de salto utiliza la dirección correcta para reiniciar el procesamiento de instrucciones.

f) Unidad de Final de Instrucción (Retire Unit):

Se actúa sobre el ROB para almacenar los resultados de la ejecución en los registros de la arquitectura. Para ello va retirando del ROB las micro-operaciones que se han ejecutado a partir de la primera (y hasta la primera que no se ha completado). También tiene en

cuenta si se ha producido un salto mal predicho y ciertas instrucciones no se deben retirar del ROB. Puesto que el ROB tiene una capacidad de 40 uoperaciones, es conveniente que las uoperaciones que ocasionan retardos elevados no estén muy próximas (todas las uoperaciones que les siguen se mantendrían en el ROB hasta que dichas uoperaciones costosas no se retiren)

1.2. Optimización de la Captación de Instrucciones.

Las instrucciones se captan en bloques alineados de 16 bytes. Estos bloques pasan a un buffer en el que hay capacidad para dos de ellos (el tamaño de una línea de cache). Desde el buffer se pasan bloques de cómo mucho 16 bytes a los decodificadores (bloques *ifecht*) que no tienen que estar alineados (su comienzo lo marca el comienzo de una instrucción).
Los bloques *ifetch* normalmente son de 16 bytes salvo que haya una instrucción de salto para la que hay predicción. Si se produce salto, pueden pasar dos ciclos hasta que se capte la instrucción siguiente (si la instrucción de destino cruza un límite de 16 bytes, y por tanto se necesitan dos ciclos para cargar dos bloques de 16 bytes antes de que se tenga un bloque *ifetch* válido). Por ello, puede ser beneficioso que en mismo bloque *ifetch* de la instrucción de salto haya varios grupos de decodificación ya que de esta manera, se realiza la decodificación de las instrucciones de esos grupos mientras se accede a los bloques necesarios (ver sección siguiente). Las reglas que se aplican en estas situaciones se muestran en la siguiente tabla.

Número de Grupos de Decodificación en el bloque *ifetch* que contiene un salto	Hay límite de 16 bytes en el bloque *ifetch*	Límite de 16 bytes en la primera instrucción después de un salto	Retardo para empezar la decodificación	Alineamiento del primer bloque ifetch después de un salto
1			0	a 16 bytes
1		X	1	a la instrucción
1	X		1	a 16 bytes
1	X	X	2	a la instrucción
2			0	a la instrucción
2		X	0	a la instrucción
2	X		0	a 16 bytes
2	X	X	1	a la instrucción
3 -			0	a la instrucción
3 -		X	0	a la instrucción
3 -	X		0	a la instrucción
3 -	X	X	0	a la instrucción

Hasta que no se termina con un bloque *ifetch* no se envía a decodificar el siguiente. Si el final de un bloque *ifetch* tenía una instrucción incompleta, el siguiente bloque *ifetch* empieza con ella. La primera instrucción siempre va a D0.
El tiempo necesario para decodificar un trozo de código puede variar bastante según donde empiece el primer bloque ifetch. Si este tiempo es crítico, para optimizarlo habría que conocer donde empieza cada bloque. Para ello:

a) Alinear el código con los límites de 16 bytes para saber donde están éstos en el código.
b) Determinar qué longitud (bytes) tiene cada instrucción y tener en cuenta que cada bloque *ifetch* tiene 16 bytes y que si en un bloque *ifetch* no contiene una instrucción completa (el límite del bloque corta una instrucción), el siguiente bloque *ifetch* empieza justamente a partir de dicha instrucción.
c) El primer bloque *ifetch* después de un jump, call, o return empieza en la primera instrucción de un bloque *ifetch* o en la más cercana a dicho límite. Si se alinea esta instrucción en los límites de 16 bytes se sabrá donde va a empezar el bloque *ifetch*.
d) El primer bloque *ifetch* después de un salto mal predicho empieza en un límite de 16 bytes.
e) Otros eventos que hacen que el siguiente bloque *ifetch* empiece en un límite de 16 bytes son las interrupciones, excepciones, IN, OUT, y CPUID.

En la Figura 2.1 se muestra un ejemplo de código correspondiente a un bucle, junto con las direcciones de memoria donde se almacenan las instrucciones, el número de bytes de cada instrucción, las microoperaciones que generan cada una de ella, y los decodificadores de la etapa ID1 donde se introducirá.

```
address       instruction                 length    uops     expected decoder
-------------------------------------------------------------------------
1000h         MOV ECX, 1000                 5         1        D0
1005h   LL:   MOV [ESI], EAX                2         2        D0
1007h         MOV [MEM], 0                 10         2        D0
1011h         LEA EBX, [EAX+200]            6         1        D1
1017h         MOV BYTE PTR [ESI], 0         3         2        D0
101Ah         BSR EDX, EAX                  3         2        D0
101Dh         MOV BYTE PTR [ESI+1],0        4         2        D0
1021h         DEC ECX                       1         1        D1
1022h         JNZ LL                        2         1        D2
```

Figura 2.1 Ejemplo de código para ilustrar los bloques *ifetch*

El primer *ifetch* va desde la dirección 1000h a la 1010h (sin incluir) y contiene a las dos primeras instrucciones completas (como se verá en la siguiente sección tarda 2 ciclos en decodificarse). El segundo *ifetch* va desde la dirección 1007h a la1017h (sin incluir) y contiene las dos instrucciones siguientes (tarda 1 ciclo en decodificarse, utilizando los decodificadores D0 y D1). El tercer *ifetch* va desde 1017h a 1022h (inclusive) incluyendo las restantes 5 instrucciones (tarda 3 ciclos en decodificarse). La primera iteración del bucle LL necesita tres *ifetch* (y tarda 5 ciclos en decodificarse, puesto que la primera instrucción del primer *ifetch* no es una instrucción del bucle).

El primer bloque *ifetch* después del salto empezará en la instrucción LL ya que el último bloque *ifetch* tiene una alineación de 16 bytes en 1020h, y va desde 1005h a 1015h incluyendo dos instrucciones completas (la de dirección LL y la siguiente, necesitando 2 ciclos para decodificarse). El siguiente bloque *ifetch* empieza en 1011h y termina antes de la 1021h, incluyendo las cuatro instrucciones siguientes (consumiendo 4 ciclos para decodificarse). El último bloque *ifetch* de la segunda iteración va desde la dirección 1021h hasta el final, y contiene dos instrucciones, sin incluir límites de 16 bytes (necesita 1 ciclo para decodificarse). El bloque ifetch de la siguiente iteración empieza, por tanto, en un límite de 16 bytes que incluye a la dirección de salto (1000h). Por lo tanto, las iteraciones pares utilizan unos bloques ifetch distintos de las iteraciones impares (las iteraciones impares necesitan 5 ciclos para decodificarse, y las pares 7).

Cuando se alinea código (ALIGN 16, por ejemplo), el ensamblador inserta NOPs. Muchos ensambladores utilizan XCHG EBX,EBX para rellenar dos bytes, pero esto no es una buena idea puesto que esta instrucción tarda más de dos NOPs en ejecutarse. Otra posibilidad es introducir instrucciones que hagan algo útil. Esta filosofía, muy utilizada en los procesadores segmentados era menos útil en los superescalares. Sin embargo, en los Pentium se utiliza debido a la fase de adaptación CISC a RISC. Otra posibilidad para conseguir alinear código es manipular las longitudes de las instrucciones incluyendo algunos prefijos sin efecto (los procesadores los admiten mientras que la instrucción no pase de 15 bytes y no sea LOCK): DB3Eh DEC ECX (instrucción de 2 bytes).

1.3. Optimización de la Decodificación de Instrucciones y del Renombrado de Registros.

Como se ha indicado las instrucciones se captan desde la cache de instrucciones en trozos alineados de 16 bytes que se introducen en un buffer doble que puede almacenar dos trozos de código de 16 bytes. El código pasa desde el doble buffer a los decodificadores en bloques de 16 bytes de longitud (*bloques ifetch*) que no están alineados y pueden cruzar los límites de 16 bytes alineados que se captan desde la cache. En la etapa IF2 se determina donde empieza y termina cada instrucción en el *bloque ifetch*. Se pasa a la etapa IF3 donde se alinean las instrucciones recibidas de IF2 y se pasan a los tres decodificadores (D0, D1, D2) que traducen las instrucciones en micro-operaciones (*uops*) de 118 bits. Las instrucciones sencillas generan una sóla *uop* mientras que las instrucciones más complejas pueden generar varias *uops*. El decodificador D0 puede recibir cualquier instrucción, mientras que D1 y D2 sólo pueden manejar instrucciones que generan una *uop* (instrucciones de operaciones con registros y loads).

Los decodificadores pueden manejar tres instrucciones por ciclo pero sólo si se reunen una serie de condiciones que se resumen a continuación:

- La primera instrucción, decodificada en D0 no puede generar más de 4 uops en un sólo ciclo de reloj, y la segunda y tercera instrucción no deben generar más de 1 uop cada una.
- La segunda y tercera instrucción no debe tener más de 8 bytes cada una.
- Las instrucciones deben estar contenidas en el mismo bloque de ifetch de 16 bytes.

Así pues, para conseguir el máximo rendimiento (*throughput*) es aconsejable ordenar las instrucciones según el patrón de generación de uoperaciones 4-1-1. Así, en el ejemplo que se muestra a continuación, donde las instrucciones se han escrito en el formato ensamblador AT&T que genera gcc (las diferencias entre los formatos de Intel y el AT&T se resument en el Apéndice 3.1),

```
movl   _MEM1, %ebx        1 uop (D0)
incl   %ebx               1 uop (D1)
addl   _MEM2, %eax        2 uops (D0)
addl   %eax, _MEM3        4 uops (D0)
```

tarda tres ciclos en decodificarse, aunque puede reducirse a dos si se reordenan las instrucciones en dos grupos de decodificación tal y como se indica a continuación.

```
addl   %eax, _MEM3          4 uops (D0)
movl   _MEM1, %ebx          1 uop (D1)
incl   %ebx                 1 uop (D2)
addl   _MEM2, %eax          2 uops (D0)
```

Los prefijos que tienen ciertas instrucciones también pueden ocasionar pérdidas de ciclos en los decodificadores. Por ejemplo si se utiliza un *prefijo de tamaño de operando* cuando se tiene un operando de 16 bits en un entorno de 32 o viceversa (Operand-size override prefix, 66h), o un *prefijo de tamaño de dirección* (Address-size override prefix, 67h). Un prefijo produce un ciclo de penalización (si se utiliza más de un prefijo, habría una penalización de un ciclo por prefijo) si:

- El prefijo de tamaño de operando se utiliza con un operando inmediato
- El prefijo de ajuste de dirección se utiliza con una dirección que incluye un offset.

A continuación se muestran dos ejemplos de esta situación:

```
movw $0x77, _mem
```

Almacena un dato inmediato de 16 bits en una palabra de memoria. En el modo de 32 bits esto da lugar a un prefijo 66h y a un ciclo de decodificación adicional. Si se introduce el operando en un registros de 32 bits no se evita el prefijo pero se evita el ciclo extra.

```
movl $0x77,%eax
movw %ax,_mem
```

Otro ejemplo corresponde a la instrucción

```
movl %edx,_mem(%ax)
```

en la que se escribe un valor en una dirección con un offset. Se tiene un prefijo 67h y se produce un ciclo de penalización en la decodificación. Esta situación se puede evitar si se suma el offset a la dirección en una instrucción adicional

```
addw _mem,%ax
movl %edx,(%ax)
```

En cuanto a la optimización de la etapa de renombrado en el RAT, lo ideal es que, en un ciclo, no se introduzcan en el RAT grupos de uoperaciones que lean más de dos registros del banco de registros. Aunque es difícil predecir qué uoperaciones entran en el RAT en cada ciclo, se pueden seguir ciertas recomendaciones:

- Mantener las uoperaciones que leen el mismo registro lo más cerca posible para que sea más probable que entren a la vez en el RAT.
- Mantener las uoperaciones que leen de registros diferentes lo más lejos posible para que no entren a la vez en el RAT.
- Provocar renombrados de registros para evitar los ciclos perdidos en el acceso a los registros (si no se introducen muchas uoperaciones).

Desde el punto de vista de la optimización, la situación que se produce en el Pentium Pro, II, y III en las etapas de decodificación es similar a la que planteaban los procesadores segmentados, donde una reordenación del código (a mano o mediante el compilador) puede mejorar el tiempo de ejecución. Esto es debido a que la primera etapa de decodificación actúa sobre las instrucciones según el orden de programa y la eficacia de la etapa depende de dicho orden (número de uops generadas por cada instrucción y decodificadores disponibles). Esta situación surge por la necesidad de pasar el repertorio de instrucciones CISC al repertorio RISC para facilitar el diseño del núcleo superescalar del procesador.

1.4. Optimización de la Ejecución

A continuación, se indican algunas ideas para mejorar el rendimiento de los programas teniendo en cuenta las características de ejecución de las instrucciones en el procesador. En algunos casos las mejoras se pueden implementar desde el lenguaje de alto nivel, y en otros se refieren a cambios en el código en ensamblador.

a) Uso de registros parciales

Los 16 bits de la parte menos significativa de los registros de propósito general pueden utilizarse como *'registros parciales'* (EAX: AX / AH AL; EBX: BX / BH BL; ECX: CX / CH CL; EDX :DX / DH DL; ESP: SP; EBP: BP; EDI: DI; ESI: SI).

Se puede producir un 'atasco' en el cauce debido al uso de un registro parcial si se lee un registro de mayor tamaño después de una escritura en un registro parcial: incluso si las instrucciones no son dependientes, la lectura se detiene hasta que se retira la escritura. Así pues:

- Si se utilizan registros parciales, para evitar los atascos, se tendría que borrar el registro mayor con XOR o SUB antes de escribir sobre el registro parcial. Estas instrucciones tienen una implementación hardware que evita el atasco debido al uso de registros parciales. Borrar el registro mayor con MOV no evita el atasco.

  Ejemplo 4.1

```
mov eax, 0              Código original con atasco
mov ax, mem16
add  ecx, eax

xor  eax, eax           Código optimizado sin atasco
mov ax, mem16
add  ecx, eax
```

- Es conveniente sustituir siempre que se pueda una instrucción MOV que actúa sobre un registro parcial. Esto evitará atascos

  Ejemplo 4.2

```
mov ah,cl // Atasco          mov eax,ecx // Sin atascos
```

```
mov  al,dl                    shl   eax,8
mov  mem, eax                 and   edx,0xff
```

También se producen atascos de registros parciales relacionadas con los flags (Flags Stalls) cuando:

- Una escritura en alguno (no en todos) de los flags de estado del registro EFLAGS precede a una lectura tanto de los flags modificados como no modificados (no hay problema si se leen sólo los modificados o sólo los no modificados).

  Ejemplo 4.3:

  ```
  sahf        // almacena el registro ah en el registro de flags excepto OF
  jg label    // jg lee el flag OF junto con los SF y ZF
  ```

  Se puede evitar el 'atasco' utilizando instrucciones que modifican todos los flags, o insertando instrucciones que no leen los flags.

  sahf // no hay atasco ***instrucciones***
  que no leen los flags jg label

- Una instrucción de rotación o desplazamiento lee el registro CL y está seguida por una instrucción que lee cualquier flag.

  Ejemplo 4.4:

  ```
  sal eax,cl          // sal lee el registro CL
  jz label            // JZ lee el flag ZF y se produce atasco
  ```

  Se puede evitar el atasco insertando una instrucción que modifique todos los *flags* después de la instrucción de desplazamiento o de rotación.

  ```
  sal eax,cl
  cmp eax,0           //cmp escribe en todos los flags de estado
  jz label            // jz puede leer el flag ZF sin producir atascos
  ```

b) Reducción de Dependencias y desenrollado de bucles

Evitar secuencias de instrucciones con dependencias para utilizar mejor los recursos de procesamiento superescalar del procesador y otros recursos como las instrucciones de los repertorios MMX o SSE.

Ejemplo 4.5:

```
float dot-product(float *a, float *b)
{
        int i; float tmp=0.0; for
        (i=0; i<ARR; i++) {
```

```
            tmp += a[i]*b[i];
        }
        ret tmp;
}
```

Puede mejorar su rendimiento mediante desenrollado del bucle y evitando dependencias.

```
float dot-product(float *a, float *b)
{
        int i;
        float tmp0=0.0, tmp1=0.0, tmp2=0.0, tmp3=0.0;
        for (i=0; i<ARR; i+=4) {
            tmp0 += a[i]*b[i]; tmp1
            += a[i+1]*b[i+1]; tmp2
            += a[i+2]*b[i+2]; tmp3
            += a[i+3]*b[i+3];
        }
        ret tmp0+tmp1+tmp2+tmp3;
}
```

El desenrollado de bucles aumenta bastante las prestaciones puesto que reduce el número de saltos, aumenta la oportunidad de encontrar instrucciones independientes y facilita la posibilidad de insertar instrucciones para ocultar las latencias. La contrapartida es que aumenta el tamaño de los códigos. Otro ejemplo de uso de desenrollado de bucles es el siguiente:

Ejemplo 4.6:

```
for (i=0;i<100;i++)
        if ((i%2) = = 0)
            a[i]=x;
        else
            a[i]=y;
```

Tras el desenrollado, el código anterior queda:

```
for (i=0;i<100;i+=2)
        { a[i]=x;
        a[i+1]=y;
}
```

c) Utilizar las nuevas instrucciones más rápidas del repertorio

Utilizar en los programas instrucciones que son más rápidas en el Pentium Pro, II, y III que en otros microprocesadores anteriores. Si el compilador no es capaz de utilizarlas, sería conveniente introducirlas *a mano* en el código en ensamblador.

Ejemplo 4.7:

Utilizar IMUL en lugar de desplazamientos y sumas (se reduce el tamaño del código)

```
mov ecx, eax          imul eax,34
shl   eax,ecx
add  eax,ecx
shl   eax,1
```

Utilizar CDQ en lugar de movimientos y desplazamientos (se reduce el tamaño del código)

```
mov  edx,eax                cdq
sar    edx,31
```

Utilizar MOVSX y MOVZX en lugar de movimientos y desplazamientos para evitar atascos por registros parciales.

```
mov ax,mem16          movsx eax,mem16
shl    eax,16
sar    eax, 16
```

La división necesita mucho más tiempo que la multiplicación y que otras operaciones. Una forma de evitar este coste, sobre todo si se deben repetir muchas multiplicaciones como en los bucles, es calcular el inverso del divisor fuera del bucle y utilizar la multiplicación.

Ejemplo 4.8:

```
for (i=0;i<100;i++)   a[i]=a[i]/y;
```

Es más eficiente utilizar

```
temp=1/y;
for (i=0;i<100;i++) a[i]=a[i]*temp;
```

d) Evitar el uso de código ambiguo

Si se utiliza código ambiguo y los compiladores no pueden resolver los punteros, tampoco pueden realizar ciertas optimizaciones, asignar variables durante la compilación. Tampoco se podrían realizar cargas de memoria mientras que un almacenamiento está en marcha, etc. Para evitar esto: *utilizar variables locales en lugar de punteros, utilizar variables globales si no se pueden utilizar las locales, y poner las instrucciones de almacenamiento después o bastante antes de las de carga de memoria*. No obstante, si no se utilizan punteros el código es más dependiente de la máquina, y a veces las ventajas de no utilizarlos no compensa.

Ejemplo 4.9:

```
int j;                          int j;
void mars (int *v) {            void mars(int v) {
      j=7.0;                          j=7.0;
      *v=15;                          v=15;
```

```
        j/=7;                                   j/=7;
        .....                                   .....
}                                       }
```

En el código no optimizado (a la izquierda), el compilador no puede asumir que *v no apunta a j. Sin embargo en el código optimizado, el compilador puede hacer directamente j=1.0.

e) Mejora del rendimiento del procesamiento en coma flotante

Hay una serie de recomendaciones que se pueden seguir para reducir el tiempo de las operaciones en coma flotante. A continuación se enumeran algunas.

- Utilizar siempre la precisión más baja posible (según las necesidades de la aplicación):
  - Precisión simple (32 bits) es más rápida que doble (64 bits) o doble-extendida (80 bits) y consume menos memoria.
  - Las instrucciones FDIV y FSQRT tienen una latencia mucho más alta en doble precisión.

- La mayoría de los lenguajes de programación redondean al valor más cercano para las operaciones en coma flotante, y truncan -'chop'- (al número menor si es positivo y al mayor si es negativo) para la conversión a entero. Esto significa que cada vez que la instrucción FISTP (convierte el valor de ST(0) a un número con signo, almacena el resultado en el registro destino y hace un pop de la pila de registros) convierte un número en coma flotante a entero, el compilador utiliza dos veces la instrucción FLDCW (carga un operando de 16 bits en la palabra de control de la unidad de coma flotante) para cambiar el modo de redondeo (trucar antes de la conversión y ajustar al más cercano después). La instrucción FLDCW tiene un costo muy elevado puesto que interrumpe el procesamiento paralelo de instrucciones para sincronizar su ejecución (antes de cambiar . Para evitar el uso de FLDCW se puede:
  - No cambiar al modo de redondeo 'chop' si no se necesita para los resultados
  - Poner fuera de los blucles la instrucción de conversión de modo de redondeo.
  - Reemplazar el algoritmo de conversión

Ejemplo 4.10:

Código Original: for (i=0; i<N; i++) (long) a[i] = (float) f[i];

Código Optimizado: for (i=0; i<N; i++) a[i] = f2i_Current RoundMode(f[i])

```
int f2i_CurrentRoundMode (double d)
_asm {      fld d
            fistp tem_place
     }
     return temp_place
```

En el código optimizado, la función f2i_CurrentRoundMode () hace la conversión sin cambiar el modo de redondeo (no genera la instrucción FLDCW) Si hay que cambiar el modo de redondeo más de una vez, y se necesita el modo 'chop' se puede sacar fuera de los bucles.

Código Original: for (i=0; i<N; i++) (long) a[i] = (float) f[i];

Código Optimizado:

```
start_chop();
for (i=0; i<N; i++) a[i] = f2i_Current
RoundMode(f[i]) end_chop();

int f2i_CurrentRoundMode (double d)
_asm {      fld d
            fistp tem_place
     }
     return temp_place
```

La función start_chop() cambia el modo de redondeo al modo truncado 'chop', después, en el bucle se utiliza f2i_CurrentRoundMode() para pasar de Coma Flotante a entero, y al final end_chop() vuelve a cambiar el modo de redondeo al más cercano.

- Se puede hacer una llamada a un procedimiento distinto del utilizado por el compilador para hacer la conversión. En ese procedimiento no se utiliza la instrucción FLDCW.

  Ejemplo 4.11:

  Código ensamblador de llamada a la función de conversión _ftol que hace la conversión de truncamiento 'chop' sin utilizar FLDCW:

```
fld d1
fadd d2
call _ftol
mov i,eax
ret
```

  Este código corresponde al código C:

```
double d1=1.3;
double d2=2.3;
int i;
.....
i=d1+d2;
.....
```

1.5. Optimización del acceso a Memoria Principal y Cache

Pentium están diseñados para código de 32 bits. El acceso a memoria segmentado para código o datos degradan las prestaciones de manera significativa por lo que es preferible un modo de memoria de tipo *flat* de 32 bits.

En el Pentium Pro, Pentium II y Pentium III, el acceso a los datos no alineados supone un costo adicional de entre 6 y 12 ciclos cuando se cruza el límite de la línea de cache. Los operandos menores de 16 bytes que no crucen una línea de cache no dan lugar a penalización aunque estén mal alineados.

Es posible controlar, desde un programa escrito en un lenguaje de alto nivel como C, la alineación de los datos que utiliza dicho programa y con ello evitar la penalización que pueda producirse por una falta de alineación.

Ejemplo 5.1

```
(1)   struct estructura { int a; int b; } s[N];
(2)   struct estructura *p,*nuevo_p;
      .................
      .................
(3)   p= (struct estructura *) malloc(sizeof(struct estructura)*N+LIM);
(4)   nuevo_p=(struct estructura *)(((int)p+LIM-1)&~(LIM-1));
```

En el ejemplo anterior se inicializa el puntero nuevo_p que apuntará a una posición de memoria alineada con las direcciones múltiplos de LIM (en la línea (4) del ejemplo). Si se quiere que los datos estén alineados en las líneas de 32 bytes que se pasan a la cache del Pentium LIM debe hacerse LIM=32. Previamente, en la línea (3) del ejemplo se asigna una zona de memoria de un tamaño igual al que se necesita para los datos que se utilizan más una serie de posiciones adicionales iguales a LIM. El puntero p apunta a esa zona y a partir de él se obtiene el puntero nuevo_p desplazando el número de posiciones necesario para que se empiece en la dirección correspondiente adecuada.

Las caches internas del Pentium II y el Pentium III tienen 16 KBytes para datos y 16 KBytes para instrucciones. Los datos de este primer nivel de cache se pueden leer en un sólo ciclo de reloj, mientras que si se produce una falta de cache se necesitarán más ciclos para acceder al dato. El conocimiento de la forma de funcionamiento de la cache permite utilizarla de forma más eficaz. La cache de datos consta de 512 lineas de 32 bytes, y cada vez que se intenta acceder a un dato que no está en cache, el procesador leerá una línea de cache entera de la memoria principal que estará siempre alineada con las direcciones físicas divisibles por 32 (empieza en direcciones divisibles por 32). Esta circunstancia se puede aprovechar si los datos a los que se va a acceder están cerca unos de otros en bloques alineados de 32 bytes.

Puesto que la cache es asociativa por conjuntos de cuatro vias en el Pentium II y Pentium III, los bits 5 a 11 de la dirección física de memoria indican el conjunto en el que se introduce la línea correspondiente. Así, para conocer si dos líneas se almacenan en cache en el mismo conjunto, se toman dos direcciones, una de cada línea y se hacen igual a 0 los 5 bits menos significativos de ambas. Si la diferencia de las nuevas direcciones es múltiplo de 4096, las líneas a las que pertenecen esas direcciones van al mismo conjunto. Si hay varias líneas que van al mismo conjunto se pueden producir problemas de tener que asignar la misma línea en cache a dos líneas distintas de memoria principal que se

están utilizando (se deben estar utilizando más de cuatro líneas que se asignen al mismo conjunto). Para controlar el uso de zonas de memoria para los datos, de forma que utilicen conjuntos distintos se puede usar una estrategia similar a la ilustrada en el Ejemplo 5.1.

Ejemplo 5.2

```
int *tempA, *tempB;
.....................
pA= (int *) malloc (sizeof(int)*N + 31);
tempA = (int *)(((int)pA+31)&~(31));
tempB = (int *)((((int)pA+31)&~(31))+4096+32);
```

En el Ejemplo 5.2 los punteros tempA y tempB están apuntando a zonas de memoria que empiezan en posiciones que son múltiplos de 32 y que no se asignarían al mismo conjunto de cache.

1.6. Optimización de Saltos

Las instrucciones de salto pueden tener un efecto bastante pernicioso en las prestaciones de un procesador segmentado (y mucho más en un superescalar) dado que pueden romper el orden en que las instrucciones deben introducirse en el cauce.

El Pentium (Pro, PII y PIII) dispone de un BTB (*Branch Target Buffer*) con 512 líneas organizadas en 32 conjuntos de 16 elementos o vías. Cada línea se asigna utilizando los bits 4-31 de la dirección del último byte de la instrucción de salto, los bits 4-8 definen el conjunto, y el resto se introducen como una marca. El el BTB se almacenan las direcciones de las instrucciones de salto y sus bits de historia para implementar el procedimiento de predicción dinámica de salto utilizado en el Pentium. El procesamiento de una instrucción de salto se lleva a cabo como sigue:

- Cuando el procesador capta 32 bytes de cache, el procesador marca el comienzo y el final de cada instrucción en los primeros 16 bytes y comprueba en el BTB (*Branch Target Buffer*) si hay información de historia de alguna de las instrucciones:
    - Si no hay, las instrucciones pasan al decodificador (iniciándose el proceso de identificación de instrucciones en los siguientes 16 bytes). Si al decodificar las instrucciones se descubre una instrucción de salto que no se ha detectado antes (porque no estaba incluida en el BTB) se aplica un algoritmo de predicción estática (que se describe a continuación).
    - Si hay información de historia, se aplica un procedimiento de predicción dinámica (que también se describe a continuación) para establecer el nuevo valor de IP y se mandan los 16 bytes al decodificador. Los nuevos 32 bytes se captan teniendo en cuenta el nuevo valor de IP.

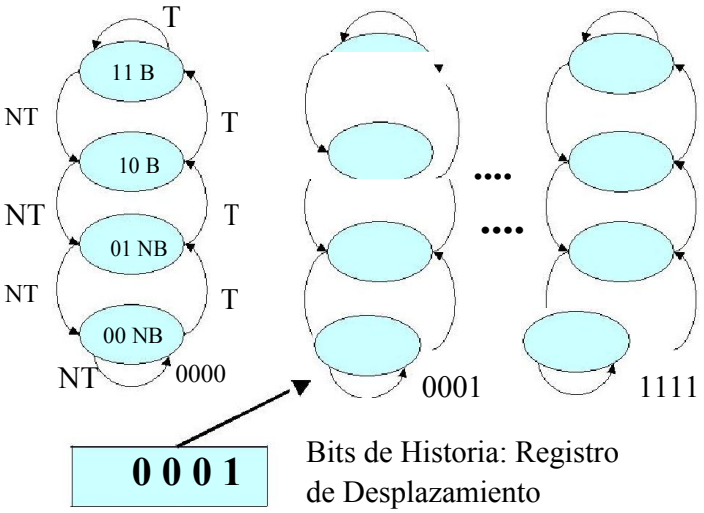

Figura 6.1. Esquema del procedimiento de predicción dinámica en el Pentium II

- El algoritmo de predicción dinámica utiliza un procedimiento basado en cuatro bits de historia (lo que ha ocurrido las cuatro últimas veces que se ha ejecutado la instrucción de salto: 0 indica que no se saltó, y 1 que sí). Esos cuatro bits designan dos bits de historia que indican la predicción que debe hacerse (como si se tuviera un esquema de predicción dinámica de dos bits). Por lo tanto, el procedimiento de predicción dinámico utiliza, en realidad, 16*2 + 4 =3. En la Figura 6.1 se muestra un esquema de este procedimiento.

- El procedimiento de predicción estático utilizado es el siguiente:
  - Si la dirección de salto no es relativa al contador de programa IP: Predice 'Saltar' si el salto es un 'return', y 'No Saltar' en caso contrario.
  - Si la dirección de salto es relativa a IP: Predice 'Saltar' si el salto es hacia atrás (situación análoga a los bucles), y 'No Saltar' si el salto es hacia delante.

A continuación se presentan algunas estrategias que pueden contribuir a evitar su incidencia, mejorando la eficacia de algunos de los recursos que el procesador implementa con el mismo fin (por ejemplo el mecanismo de predicción de saltos).

a) Mejora del rendimiento de la Predicción de Saltos

Existen transformaciones de código que pueden mejorar el rendimiento de los procedimientos de predicción de salto, evitando que se produzcan patrones de salto no predecibles. Hay que tener en cuenta, que en el Pentium (Pro, II, y II) una predicción incorrecta puede dar lugar a una penalización de hasta 20 ciclos.

El procedimiento de predicción dinámica es bastante eficaz. Por una parte, permite predecir correctamente todas las secuencias de saltos con frecuencia de 4. Además, para comprobar si un patrón de periodo n da lugar a un predicción incorrecta se escriben las n subsecuencias de 4 bits del patrón. Si son todas diferentes no habrá predicciones incorrectas después de un aprendizaje de dos periodos.

El esquema de predicción dinámica también maneja de forma eficaz una situación en la que se alterna entre dos patrones repetitivos. Por ejemplo, si se produce un patrón hasta que se ha aprendido y luego se pasa a otro patrón, se produce un mínimo de predicciones incorrectas si los dos patrones no tienen subsecuencias de 4 bits en común (no usan los mismos contadores).

Ejemplo de secuencia predecible: **1000100010001000**
Ejemplo de secuencia no predecible: **0000010000001000001**

Un ejemplo de mejora del rendimiento de la predicción dinámica es el siguiente. Si un bucle se ejecuta 20 veces no se predice correctamente la última iteración. Para evitar esto se puede utilizar dos bucles anidados de 4 y 5 iteraciones respectivamente, o desenrollar el bucle por cuatro, para que sólo haya 5 iteraciones.

Si un salto que no tiene ninguna historia almacenada en el BTB se utiliza la predicción estática, que predice los *saltos hacia delante como no tomados*. Se pueden mejorar las prestaciones del procedimiento si se sitúa el código más frecuente después del salto condicional hacia delante.

Ejemplo 6.1:

Código Ensamblador Original

```
        comp a, 5
        je L1
        Código Infrecuente
        jmp   L2
L1:
        Código Frecuente
L2:
```

Código Ensamblador Optimizado

```
        comp a, 5
        jne L1
        Código Frecuente
        jmp L2
L1:
        Código Infrecuente
L2
```

b) Reducción del número de Saltos en un programa

Se puede reducir el número de saltos de un programa reorganizando las alternativas en las sentencias *switch*, en el caso de que alguna opción se ejecute mucho más que las otras (más del 50% de las veces, por ejemplo). Ciertos compiladores que utilizan información de perfiles de ejecución del programa son capaces de realizar esta reorganización. Se recomienda utilizarla si la sentencia *switch* se implementa como una búsqueda binaria en lugar de una tabla de salto.

Ejemplo 6.2

Código original

```
switch (i)
{
        case 16:
                Bloque16
                break;
        case 22:
                Bloque22
```

Código optimizado

```
if (i==33)
{ Bloque33 }
else
        switch (i)
        {
                case 16:
                        Bloque16
```

```
                break;                                          break;
        case 33:                                        case 22:
                Bloque33                                        Bloque22
                break;                                          break;
}                                               }
```

c) Reducción del número de instrucciones de Salto

Se pueden utilizar instrucciones con predicado que se incluyen el el repertorio del Pentium. Por ejemplo, existe un MOV condicional para enteros (CMOVcc) y para coma flotante (FCMOVcc) que permiten evitar instrucciones de bifurcación en los programas.

Ejemplo 6.3.

CMOVcc hace la transferencia de información si se cumple la condición indicada en cc

```
        test ecx,ecx                    test ecx,ecx
        jne 1h                          cmoveq eax, ebx
        mov eax,ebx
1h:
```

FCMOVcc es similar a CMOVcc pero utiliza operandos en coma flotante.

Hay que tener en cuenta, no obstante, que si los datos a los que acceden estas instrucciones no están en cache y la condición es falsa, pueden consumir tiempos mayores que si no se utilizan.

La instrucción *SETcc* es otro ejemplo de instrucción con predicado que puede permitir reducir el número de instrucciones de salto.

Ejemplo 6.4:

ebx= (A<B) ? C1 : C2; [Si (A<B) es cierto EBX se carga con C1 y si no con C2]

Código Original

```
        cmp  A,B
        jge   L30
        mov  ebx,C1
        jmp   L31
 L30:  mov    ebx,C2
 L31:
```

Código Optimizado	*Explicación:*
xor ebx,ebx	Si A>=B, *setge* hace BL=1; DEC hace EBX=0; (EBX and C1-C2)=0; EBX + C2 = C2
cmp A,B	
setge bl	Si A<B, *setge* hace BL=0; DEC hace EBX=0xFFFFFFFF; (EBX and C1-C2)= C1-C2;
dec ebx	EBX+C2=C1

and ebx, (C1-C2)
add ebx, C2

7. Optimización de Bucles

En un programa, a menudo, la mayor parte del tiempo se pasa en uno de los bucles del mismo. La forma más directa de mejorar la velocidad consiste en optimizar cuidadosamente el bucle que más tiempo consuma utilizando el lenguaje ensamblador.

```
            .file       "cambiosign.c"
gcc2_compiled.:
___gnu_compiled_c:
.text
            .p2align    2
.globl  _changesign
_changesign:
        pushl   %ebp
        movl    %esp,%ebp
        subl    $16,%esp
        pushl   %esi
        pushl   %edi
        movl    16(%ebp),%ecx
        jecxz   L2
        movl    8(%ebp),%esi
        movl    12(%ebp),%edi
        .p2align  4,,7
        cld
L1:     lodsl
        negl    %eax
        stosl
        loop    L1
L2:
        leal    -24(%ebp),%esp
        popl    %edi
        popl    %esi
        movl    %ebp,%esp
        popl    %ebp
        ret
```

Figura 7.1 Código ensamblador generado para *changesign()*

A continuación se muestran algunos ejemplos de las cuestiones a tener en cuenta a la hora de optimizar los bucles. En estos ejemplos se asume que los datos están en la cache de nivel 1. Si la velocidad está limitada por los fallos de cache habría que concentrarse primero en distribuir los datos para disminuir los fallos. Como punto de partida se tomará un procedimiento sencillo en C:

```
void changesign (int *A, int *B, int N)
        { int i;
        for (i=0; i<N; i++) B[i]= -A[i] ;}
```

Para realizar la optimización se tiene en cuenta:

- La alineación de los bloques de captación (*ifetch*) y la decodificación
- Los atascos (*stalls*) en la lectura de registros y las características de ejecución de uops
- La posibilidad de desenrollar bucles.

En la Figura 7.1 se muestra el código en ensamblador generado por el compilador para el procedimiento anterior. En este caso, se utiliza sintaxis AT&T, que es la que genera el

compilador de C de DJGPP (Apéndice 3.1). Cada iteración del bucle de la Figura 7.1 tarda en torno a 11 ciclos. No obstante, este código puede mejorarse más, si se evitan las instrucciones que generan muchas *uops*, como por ejemplo LOOP, LODS, y STOSD.

A continuación se considerarán distintas mejoras del código (en concreto de la zona representada dentro del cuadro en línea discontinua) teniendo en cuenta la microarquitectura del procesador y utilizando el desenrollado de bucles. En cuanto a la microarquitectura, se ha considerado:

- *Decodificación de instrucciones.* Las instrucciones que generan más de 1 uop van al decodificador D0. Esto debe tenerse en cuenta a la hora de reorganizar código.

- *Límites de los bloques de 16 bytes de instrucciones que se captan.* Una forma de mejorar la velocidad es conseguir que las instrucciones del bucle estén alineadas en el menor número posible de estos bloques

- *Atascos producidos por las lecturas de registros* (posibles riesgos de tipo RAW)

Análisis de las uops que van a cada puerto de ejecución. Hay que evitar las colisiones en la medida de lo posible.

- *Retirada de instrucciones del ROB.* En cada ciclo se pueden retirar como mucho 3 uops.

```
        pushl %esi
        pushl %edi
        movl 16(%ebp),%ecx
        jecxz L2
        movl 8(%ebp),%esi
        movl 12(%ebp),%edi
        .p2align 4,,7    ; alineado a 16 bytes
L1:
        movl (%esi),%eax ;  2 p2rESIwEAX
        addl $4,%esi     ;  3 p01rwESIwF
        negl %eax        ;  2 p01rwEAXwF
        movl %eax,(%edi) ;  2 p4rEAX, p3rEDI
        addl $4,%edi     ;  3 p01rwEDIwF
        decl %ecx        ;  1 p01rwECXwF
        jnz L1           ; 2 p1rF
L2:
        leal -24(%ebp),%esp
        popl %edi
        popl  %esi
```

Figura 7.2 Primera mejora del procedimiento de la Figura 7.1

En la Figura 7.2 se muestra el código correspondiente a la primer mejora aplicada al código de la Figura 7.1. En los comentarios de cada línea correspondiente a una instrucción del bucle se indica el número de bytes de la instrucción, y la uop que se genera junto con el puerto de la unidad de ejecución que se utilizará. Por ejemplo, p01rwEAXwF significa que se puede utilizar el puerto 0 o el 1 de la unidad de ejecución, y que la uop generada lee y escribe del registro EAX y escribe en el registro de estado F.
Hay tres grupos de decodificación en el bucle. Las tres primeras instrucciones genera una uop cada uno (se decodifica una en D0, otra en D1, y otra en D2), las tres instrucciones siguientes generan 2-1-1 uops (la primera se debe decodificar en D0 por ser la primera o por generar más de una uop, y las dos instrucciones restantes se pueden decodificar en

D1 y D2 porque sólo generan 1 uop cada una). Finalmente, la última instrucción de bucle se decodifica en el ciclo siguiente. Se necesitan tres ciclos para la decodificación.
Puesto que el bucle entero tiene 15 bytes, se puede introducir completamente en un bloque *ifetch* de 16 bytes si se alinea el comienzo del bucle a 16 bytes (tal y como se hace mediante la directiva .p2align).

No hay atascos de registros dado que cuando se hace una lectura de registro, la escritura en el mismo se ha hecho unos ciclos antes. En cuanto a la ejecución, para los puertos p0 o p1, hay 4 uops; para p1, 1 uop; y 1 uop para cada unos de los puertos p2, p3, y p4. Suponiendo una distribución óptima de las operaciones esto supone unos 3 ciclos.
El número de ciclos necesario para retirar las uops es igual al menor entero mayor o igual al número de uops dividido por tres. Como hay 8 uops, se necesitan tres ciclos para retirar las uops. Por tanto, cada iteración de bucle se puede terminar en tres ciclos de reloj.

En la Figura 7.3 se muestra otra mejora del código correspondiente a la Figura 7.1. En este caso, cada iteración del bucle se puede terminar en dos ciclos.

```
   pushl %esi
   pushl %edi
   movl 8(%ebp),%esi
   movl 12(%ebp),%edi
   movl 16(%ebp),%ecx
   leal (%esi,%ecx,4),%esi
   leal (%edi,%ecx,4),%ebx
   negl %ecx
   jz   L2
   .p2align 4,,7
L1:
   movl (%esi,%ecx,4),%eax ; 3 p2rESIrECXwEAX
   negl %eax               ; 2 p01rwEAXwF
   movl %eax,(%edi,%ecx,4) ; 3 p4rEAX, p3rEDIrECX
   incl %ecx               ; 1 p01rECXwF
   jnz L1                  ; 2 p1rF
L2:
   leal -24(%ebp),%esp
   popl %edi
   popl %esi
```

Figura 7.3. Mejora 2 del código de la Figura 7.1

Se han reducido a 6 las uops utilizando el mismo registro como contador e índice. Los punteros base apuntan al final de los *arrays* para que se pueda contar descendentemente.

Hay dos grupos de decodificación: las dos primeras instrucciones (1-1) y las tres últimas (2-1-1) de cada iteración. Por lo tanto, la decodificación se puede hacer en 2 ciclos.

Alineando el bucle en los bloques de 16, como el número de bytes es 11, sólo se necesitan 2 ciclos para captar las instrucciones. Como hay 2 uops para p0 o p1, y 1 uop para p1, p2, p3, y p4, se podrían iniciar las ejecuciones en 2 ciclos. Finalmente, como hay 6 uops, se pueden retirar en dos ciclos (tres por ciclo).

```
   .p2align 4,,7
L1:
  movl (%esi),%eax ;2  p2rESIwEAX
  negl %eax         ;2 p01rwEAXwF
  movl %eax,(%edi) ;2  p4rEAX,p3rEDI
  movl 4(%esi),%eax;3  p2rESIwEAX
  negl %eax         ;2 p01rwEAXwF
  movl %eax,4(%edi);3  p4rEAX,p3rEDI
  addl $8,%esi      ;3 p01rwESIwF
  addl $8,%edi      ;3 p01rwEDIwF
  decl %ecx         ;1 p01rwECXwF
  jnz L1            ;2 p1rF
```

Figura 7.4 Desenrollado de bucle correspondiente al código de la Figura 7.2

Otra forma de mejorar las ejecución del bucle consiste en aplicar la técnica de desenrollado, mediante la cual, en cada iteración del nuevo bucle se hacen varias del anterior. Con el desenrollado se reduce el *overhead* debido a los saltos y el número de saltos.

Por ejemplo, en el código de la Figura 7.2 el *overhead* de cada bucle es de 4 uops (actualizar punteros, contador e instrucción de salto) frente a 4 uops para hacer los cálculos en si. Si se hiciera un desenrollado como indica el código de la Figura 7.4 (dividiendo el número de iteraciones por la mitad) se pararía de un 50% de *overhead* a un 33%.

Existe un código que puede tener un mejor comportamiento que éste en cuanto a que puede reducir el número de ciclos de decodificación y el número de ciclos necesario para retirar las instrucciones del ROB. Éste se muestra en la Figura 7.5.

```
  .p2align 4,,7
L1:
  movl (%esi),%eax ;2 p2rESIwEAX
  movl 4(%esi),%ebx;3 p2rESIwEBX
  negl %eax        ;2 p01rwEAXwF
  movl %eax,(%edi) ;2 p4rEAX,p3rEDI
  addl $8,%esi     ;3 p01rwESIwF
  negl %ebx        ;2 p01rwEBXwF
  movl %ebx,4(%edi);3 p4rEBX,p3rEDI
  addl $8,%edi     ;3 p01rwEDIwF
  decl %ecx        ;1 p01rwECXwF
  jnz L1           ;2 p1rF
```

Figura 7.5 Mejora en el bucle desenrollado

Teniendo en cuenta la distribución de instrucciones, el primer grupo de 16 bytes incluiría hasta la instrucción negl %ebx, y se podrían decodificar en dos ciclos. El resto de instrucciones del bucle se podrían decodificar en otros dos ciclos. Este código se puede *decodificar en 4 ciclos en lugar de los 5* que necesitaba el anterior. Con este código también se mejora el rendimiento del ROB.

El coste de esta última modificación ha sido la necesidad de introducir un registro extra (el registro EBX).

Hasta aquí la descripción de algunas de las optimizaciones posibles a partir de las características de la arquitectura, y microarquitectura de los procesadores Pentium (Pro, II, y III). No se han considerado, no obstante, las posibles mejoras de prestaciones que surgen de utilizar los repertorios de instrucciones multimedia MMX y SSE han añadido al repertorio x86 en (algunos de) estos procesadores.

2. Ejercicios de Optimización de Código

Dado la disparidad de microarquitecturas de las que los alumnos podrán disponer para la realización de las prácticas, las mismas se centrarán en optimizaciones de código en C (OHLL) y ensamblador para cualquier procesador (OGP).

Opcionalmente, será muy valorado que el alumno aplique optimizaciones específicas para su procesador. Para ello, en el guión debe indicar la microarquitectura de la que dispone, y las referencias sobre su diseño en la que se basa para desarrollar las mejoras (en el apartado final de este guión puede encontrar algunas de estas referencias)

Ejercicio 1 - Para el programa ejemplo1.c que se muestra en la Figura 1.

A. Modifique el código C para reducir el tiempo de ejecución del mismo. Justifique los resultados obtenidos a partir de la modificación realizada.

B. Genere los programas en ensamblador para la implementación más eficiente de ejemplol.c del apartado anterior, considerando las distintas opciones de optimización del compilador (-O1, -O2,..) y explique los resultados obtenidos a partir de las características de dichos códigos.

C. Partiendo del código ensamblado del apartado anterior que justifique como más eficiente, realice modificaciones destinadas a mejorar sus prestaciones de tiempo. En las referencias encontrará recomendaciones sobre mejoras en el código ensamblador, independientes y aplicables a procesadores específicos.

Ejercicio 2 - El benchmark Linpack ha sido uno de los programas más ampliamente utilizados para evaluar las prestaciones de los computadores. De hecho, se utiliza como base en la lista de los 500 computadores más rápidos del mundo (el Top500 Report). El núcleo de este programa es una rutina denominada daxpy que multiplica un vector por una constante y los suma a otro vector:

```
for (i=1;i<=N,i++) y[i]=y[i]+a*x[i];
```

A. Siguiendo las indicaciones de diseño para la medición de prestaciones descritas en el Apéndice 3.2. Desarrolle en C un programa basado en el anterior bucle. Genere los programas en ensamblador para cada una de las opciones de optimización del compilador (-O1, -O2,..) y explique las diferencias que se observan en el código justificando las mejoras en velocidad que acarrean.

B. Para la mejor de las opciones, obtenga los tiempos de ejecución con distintos valores de N y determine para su sistema los valores de Rmax (valor máximo del número de operaciones en coma flotante por unidad de tiempo), Nmax (valor de N para el que se consigue Rmax), y N1/2 (valor de N para el que se obtiene Rmax/2). Estime el valor de la velocidad pico (Rpico) de su procesador y compárela con el valor obtenido para Rmax.

```
/*------ Ejemplo de Programa de Prueba     -------*/
#include <stdio.h>
#include <math.h>
#include <time.h>
struct { int a; int b;
      } s[500]; long ii,i;
main()
{
      clock_t start,stop;
      start= clock();      /* Inicio de medida de tiempo */
      /*       ---------- Algoritmo  a ejecutar -------------- */
      for (ii=1;ii<=M;ii++)
      {
            for (i=0;i<500;i++) s[i].a=2*s[i].a;
            for (i=0;i<500;i++) s[i].b=3*s[i].b;
      }
      /*       ----------- Final del Algoritmo --------------- */
      stop = clock();        /* Final de medida de tiempo*/
      printf("Tiempo= %f",difftime(stop,start));
      return 0;
   }
```

Figura 1. Programa ejemplo1.c (M se fija a un valor suficientemente grande para que el tiempo de ejecución del programa sea apreciable)

Apéndice 3.1. Formato AT&T del Ensamblador x86

Las principales diferencias de este formato de instrucciones en ensamblador con respecto al formato de Intel son las siguientes:

- Los operandos registros aparecen precedidos por el carácter %

  AT&T: %eax Intel:
  eax

- El orden de los operandos es el inverso ya que primero se pone el operando fuente y después el operando destino.

  AT&T: movl %eax, %ebx
  Intel: mov ebx,eax

- Los operandos inmediatos están precedidos por el carácter $

  AT&T: movl $0xfd6, %ebx movl $var, %ebx
  Intel: mov ebx, 0xfd6 mov ebx, offset var

- El tamaño del operando se especifica mediante el último carácter del código de operación (opcode): b (8 bits), w (16 bits), l (32 bits).

  AT&T: movb var,ah movw %bx,%ax
  Intel: mov ah, byte ptr var mov ax,bx

- La mayor parte de los códigos de operación son idénticos en los dos formatos excepto:

  movsSD (movsx en el formato de Intel)
  movzSD (movz) donde S y D son los prefijos de tamaño de fuente y destino cbtw (cbw)
  cwtl (cwde)
  cwtd (cwd)
  cltd (cdq)
  lcall $S,$O (call far S:O)
  ljmp $S,$O (jump far S:O)
  lret $V (ret far V)

- Los prefijos de los códigos de operación no deben escribirse en la misma línea de la instrucción sobre la que actúan.

- Las referencias a memoria también son algo diferentes. La forma usual en el formato de Intel segmento:[base+índice*escala+desplazamiento] se escribe en el formato AT&T como segmento:desplazamiento(base,índice,escala).

Intel	AT&T
[ebp+4]	4(%ebp)
[eax+eax*4]	(%eax,%eax,4)
[4*eax+array]	_array(,%eax,4)
fs:eax	%fs:(%eax)

- La forma de indicar los saltos lejanos.

AT&T:	lcall $seccion,$offset	ljump $seccion,$offset
Intel:	call far seccion:offset	jump far seccion:offset

Apéndice 3.2. Uso de DJGPP en los Ejercicios

Una forma de realizar los programas cuyas prestaciones se deben evaluar es utilizar un *programa de base* donde se realizan las entradas y salidas del programa, y se incluyen otras llamadas a funciones, como por ejemplo las necesarias para medir tiempo. Ese programa realiza llamadas a funciones que incluyen el procedimiento, algoritmo etc. cuya implementación se pretende evaluar y optimizar. Para esas funciones se pueden ensayar distintas implementaciones de alto nivel, o generar los correspondientes códigos en ensamblador, sobre los que se pueden aplicar reordenaciones de instrucciones, renombrados, y otras optimizaciones a mano (a nivel de código ensamblador).

Como ejemplo, en la Figura A2.1 se muestra el programa de base que hemos llamado *test_bench.c*

```
/* Ejemplo de Programa de Prueba */

#include   <stdio.h>
#include   <math.h>
#include <time.h>

int suma_prod(int a, int b, int n);

main()
{
        /* ------------- */
        int i,j,a,b,n,c;
        /* ------------- */

        clock_t start,stop;

        start= clock();

        /* ---------------------------------------- */

        n=6000;a=1;b=2;
        for (j=1;j<=10000;j++)
        {
                printf("a=%d b=%d n=%d\n",a,b,n);
                c=suma_prod(a,b,n);
                printf("resultado= %d\n",c);
        }
        /* ---------------------------------------- */

        stop = clock();

        printf("Tiempo= %f",difftime(stop,start));

        return 0;

}
```

Figura A2.1 Código C correspondiente a un programa de base (test_bench.c)

Como se puede ver, en la Figura A2.1 se hace una llamada a una función (la función *suma_prod()*), que es para la que, en principio, interesa analizar distintas

implementaciones y estudiar posibles optimizaciones. La llamada a la función está dentro de un bucle con una serie de iteraciones para aumentar el tiempo total que consume el programa y se puedan observar intervalos de tiempo suficientemente grandes si el cálculo que hace la función no es excesivamente complicado, y/o el computador es muy rápido. También se han añadido las llamadas a las funciones (se han utilizado unas de las posibles) necesarias para medir el tiempo total del programa (que se muestra al final).

Las optimizaciones se realizarán, bien aplicando transformaciones sobre la descripción de la función a la que se llama (en este caso suma_prod()), bien a nivel de lenguaje c, bien a nivel ensamblador.

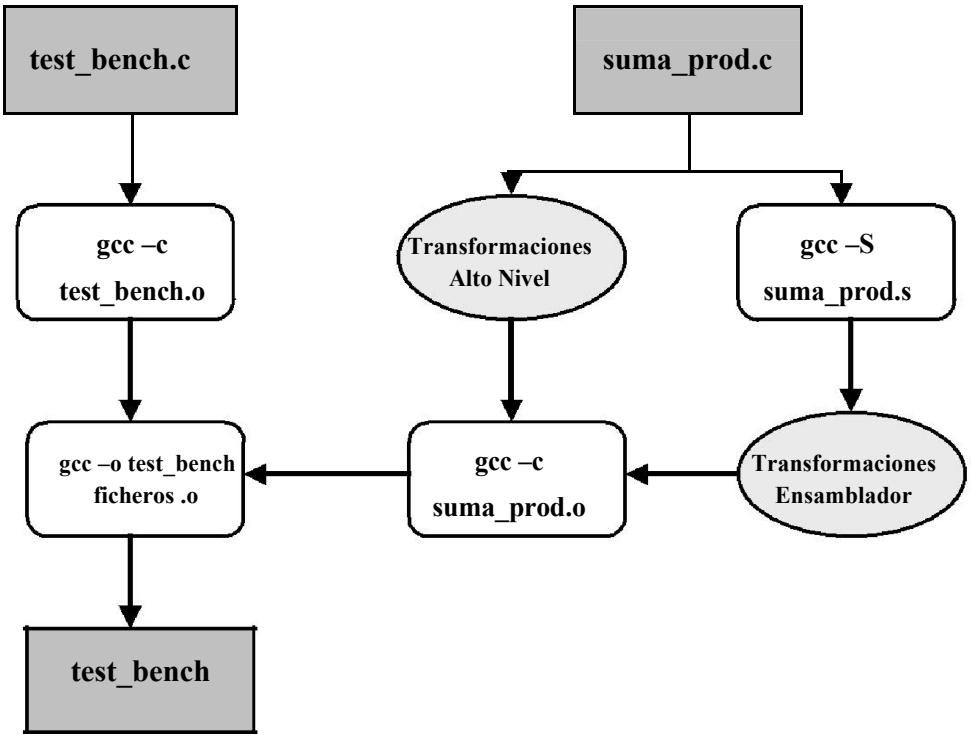

Figura A2.2 Esquema de trabajo en los ejercicios

En la Figura A2.2 se muestra el esquema de trabajo a seguir, utilizando las distintas opciones del compilador *gcc* para ir generando los distintos ficheros. Como es sabido, el proceso de compilación tiene cuatro etapas: preprocesamiento, compilación propiamente dicha, ensamblado, y enlazado (*link*). A continuación se explica el significado de las distintas opciones:

-c Se compila o se ensamblan los ficheros fuente utilizados pero no se enlazan. Es decir, se generan los ficheros objeto .o.

-S El proceso de compilación para después de la etapa de compilación propiamente dicha. Por lo tanto, se obtiene el código en ensamblador .s

Opciones de Optimización del Compilador de C (djgpp)

El compilador tiene una serie de opciones de optimización que se pueden utilizar al generar el código en ensamblador (gcc –S), permitiendo analizar el tipo de cambios que

realizan en el mismo. Las opciones se denominan: -O1, -O2, -O3, -Os. Se explican con algo más de detalle a continuación.

-O1: Con esta opción, el compilador modifica las siguientes alternativas de compilación

-fthread-jumps (comprueba si hay un salto a otra posición donde se encuentra otro salto y si se conoce la condición de salto se redirecciona el primero convenientemente).

-fdefer-pop (evita hacer pop de los argumentos de llamada a una función cuando se retorna de la llamada a la función)

-fdelayed-branch (en máquinas segmentadas con *'delays slots'*, intenta reordenar las instrucciones para eliminar ciclos desperdiciados).

-fomit-frame-pointer (en máquinas que permiten la depuración sin utilizar punteros de pila, indica que no se almacene el puntero de pila en un registro en el caso de funciones que no lo necesiten. Con esto se ahorran las correspondientes instrucciones de almacenamiento y recuperación del puntero).

-O2: Con esta opción se activan todas las alternativas de optimización menos el desenrollado y la opción –finline-functions (integra las funciones sencillas en donde están sus llamadas.

-O3: Se activan todas las alternativas de optimización.

-Os: Se activan todas las alternativas de –O2 que no suelen incrementar el tamaño del código

Es posible encontrar una descripción de las distintas alternativas de optimizaciónv en: http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

Algunos ejemplos (además de los indicados anteriormente):

-fforce-mem: fuerza a los operandos en memoria a copiarse en los registros antes de hacer operaciones aritméticas con ellos. Esta alternativa se activa con –O2.

-fforce-addr: fuerza que las direcciones de memoria constantes se copien en registros antes de hacer operaciones aritméticas sobre ellas. Se activa con –O2.

-ffast-math: se permite al gcc que viole algunas reglas y/o especificacioes ANSI o IEEE para conseguir más celocidad. Por ejemplo se asume que los argumentos de la función sqrt son no-negativos y que ningún valor en comaflotantre es NaN.

En la Figura A2.3 se muestra un ejemplo de una función sencilla para la que, en la Figura A2.4 se muestran los códigos en ensamblador para sin optimización, y con la optimización –O1.

```
/* Ejemplo de Funcion */

int suma_prod(int a, int b, int n)
{
        return a*b+n;
}
```

Figura A2.3 Ejemplo de código en C para una función sencilla

```
.file "suma_prod.c"
gcc2_compiled.:
___gnu_compiled_c:
.text
        .p2align 2
.globl _suma_prod
_suma_prod:
        pushl %ebp
        movl esp,%ebp
        movl 8(%ebp),%edx
        imull 12(%ebp),%edx
        addl 16(%ebp),%edx
        movl %edx,%eax
        jmp L2
        .p2align 4,,7
L2:
        movl %ebp,%esp
        popl %ebp
        ret
```

```
.file "suma_prod.c"
gcc2_compiled.:
___gnu_compiled_c:
.text
        .p2align 2
.globl _suma_prod
_suma_prod:
        pushl %ebp
        movl esp,%ebp
        movl 8(%ebp),%eax
        imull 12(%ebp),%eax
        addl 16(%ebp),%eax
        movl %ebp,%esp
        popl %ebp
        ret
```

Figura A2.4 Códigos en ensamblador para suma_prod() sin optimizar y con optimización –O1

Referencias

- Recomendaciones generales para la optimización de código ensablador X86
  http://mark.masmcode.com/

- Opciones de optimización para x86-64 en GCC
  http://gcc.gnu.org/onlinedocs/gcc-4.7.2/gcc/i386-and-x86_002d64-Options.html
  http://www.phoronix.com/scan.php?page=news_item&px=MTA3NjE

- Guía de optimizaciones para las arquitecturas 64 e IA-32 de Intel
  http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html

- Optimizaciones para procesadores AMD
  http://developer.amd.com/resources/documentation-articles/developer-guides-manuals/

- Manual de GCC
  http://gcc.gnu.org/onlinedocs/gcc/

- Acceso al compilador de C++ de Intel y a VTune
  http://software.intel.com/en-us/c-compilers
  http://software.intel.com/en-us/intel-vtune-amplifier-xe

- Ensamblador (formato AT&T), directivas, arquitectura, convenciones de llamada:
  http://www.delorie.com/djgpp/doc/ug/asm/about-386.html
  http://www.delorie.com/djgpp/doc/ug/asm/calling.html

- Repertorio de Instrucciones. Listado de instrucciones y número de microoperaciones. Información precisa sobre la microarquitectura:
  http://www.intel.com/design/PentiumIII/manuals/

- Compilador de C (gcc para DOS/WINDOWS)

  http://www.delorie.com/djgpp/

- Página de Compilación para Pentium

  http://www.iti.cs.tu-bs.de/soft/www.goof.com/pcg/

- Optimización de código para el procesador Athlon de AMD.

  http://www.amd.com/products/cpg/athlon/techdocs/pdf/22007.pdf