UNIVERSIDAD NACIONAL DE MAR DEL PLATA

Laboratorio de Procesos y Mediciones. Facultad de Ingeniería.

Reproductor de MP3

PROYECTO FINAL

Reproductor de MP3

Profesor a cargo: Raúl Rivera.

Autor: Gabriel Jacobo.
MAT: 8221.
gabriel@noticiasyprotagonistas.com

Tabla de contenido

Capítulo

Anteproyecto

En este capítulo introductorio se explicará cual es la motivación del presente trabajo, las características y ventajas del formato de compresión de audio conocido como MP3, y la estrategia de implementación que se ha utilizado.

El presente informe fue creado para detallar y explicar el funcionamiento de una placa reproductora de MP3, basada en un procesador Intel 80486 DX4 la cual fue presentada como proyecto final de la carrera de Ingeniería Electrónica. Las motivaciones para realizar tal diseño son varias, de las cuales se mencionan las más importantes:

- ✓ La tecnología del procesador 486 es compatible con la gran cantidad de computadoras compatibles con la IBM PC existentes.

- ✓ Una vez realizada la base de hardware y software necesarios para hacer correr el programa que reproduce los archivos MP3, el sistema puede ser expandido sin dificultades a una infinidad de aplicaciones, con una capacidad de procesamiento sin par en procesadores de otra tecnología.

- ✓ Acompañando el citado poder de procesamiento, esta la cuestión del costo, que es muy inferior al de otras alternativas por la amplia difusión que tienen estos procesadores en las computadoras personales. De hecho, el procesador 486 tiene ya 10 años de edad, y ha salido prácticamente del mercado comercial de computadoras, lo cual lo torna aun mas barato.

- ✓ Con pequeñas modificaciones se pueden usar herramientas estándar de PC y de Windows (como el Visual C++ de Microsoft) para desarrollar software para esta placa, haciendo innecesario aprender un lenguaje de maquina particular, o de utilizar compiladores no estándar. Esto

implica también la posibilidad de utilizar la amplia base de código abierto para arquitectura x86 existente en la actualidad.

- ✓ El formato MP3 se encuentra actualmente en el pico de su popularidad, y ofrece una compresión de audio de 1:10 en el peor de los casos, con una perdida de calidad imperceptible. La creación de un reproductor de este formato es una de las cientos de aplicaciones posibles que puede tener el hardware presentado.

Como se mencionó anteriormente, estas son algunas de las muchas razones que me impulsaron a encarar un proyecto de estas características. La mayoría de las razones no explicitadas aquí, se pueden inferir de las diferentes secciones con las que cuenta este informe. Y precisamente para dar comienzo al mismo, comenzaremos con un repaso sobre los formatos de audio.

Formatos de Audio

El formato de audio digital más simple y de uso común en el ambiente informático es el conocido como "WAV". El formato WAV no define en si un método de compresión particular, sino que se trata de una especie de "cuadro" (frame en ingles) fijo donde se almacenan varios valores necesarios (como ser la frecuencia de muestreo, la duración del audio, la cantidad de canales, el tipo de compresión, etc.) para reproducir el audio original. Junto con la mencionada información, va incluido el audio en si, que a su vez puede esta comprimido en uno de los varios formatos disponibles (dentro de los que se cuenta el MP3). De estos formatos de compresión, el más simple es el PCM (Pulse Code Modulation), cuya principal característica es no poseer compresión de información alguna.

El Wave PCM

La compresión PCM no es un sistema de compresión en si, ya que no ofrece reducción de tamaño alguna respecto de la información original. Los bytes extraídos directamente de un WAV PCM se pueden enviar a un conversor digital analógico (previa conversión del tamaño de la palabra digital y la frecuencia de muestreo, si esto fuera necesario) y escuchar el sonido sin más dificultades. Esta simplicidad de uso, se paga por el lado del tamaño: a 44,1 Khz., stereo y 16 bits de cuantificación, una hora de audio ocupa algo más de 620 MB (a un promedio de 1378 kilobits por segundo). Con la velocidad promedio alcanzable actualmente vía internet, es imposible transmitir sonido en tiempo real con la citada calidad, sin recurrir a algún método de compresión.

Métodos de compresión

Los sistemas de compresión se dividen básicamente en dos grandes grupos: los que comprimen "sin perdida" (lossless) y los que comprimen "con perdida" (lossy). El primer sistema garantiza que la información se ve inalterada luego del proceso de "compresión-descompresión". Tal sistema es necesario por ejemplo en el código ejecutable de un procesador, donde no son admisibles los cambios, por menores que estos sean ya que provocarían comportamientos impredecibles en la maquina que ejecute dicho código. Sin embargo, hay otro tipo de información, que

generalmente tiene por objetivo ser recibida por el ser humano, que es pasible de ser comprimida sin que la información original pueda ser reconstruida exactamente, y sin que esto implique mayores consecuencias (y con un poco de suerte, sin que el usuario final se percate de este hecho).

Estos algoritmos de compresión aprovechan diferentes características de los sentidos humanos, para recortar (en mayor o menor grado, de acuerdo a la cantidad de compresión necesaria) la información original, tratando a la vez que dicho recorte "pase desapercibido" por el receptor.

El Formato MP3

El formato MP3 se puede encontrar actualmente en las computadoras en dos versiones: como un archivo de extensión MP3, o como un archivo de extensión WAV. En el primer caso, decimos que es el archivo "puro", mientras que en el segundo, la información es **idéntica**, pero tiene un marco fijo propio del estándar WAV. Este marco le dice al programa encargado de la reproducción que la compresión que tiene el WAV es de tipo "ACM" (una tecnología de descompresión donde diferentes fabricantes pueden agregar sus "codecs" para tratar distintos métodos de compresión), y que dentro del ACM, debe usar el "codec" para MP3. Existen numerosos programas que pueden pasar de un formato al otro, ya que intrínsicamente son equivalentes. El citado "codec" no es otra cosa que un "codificador-decodificador", o un programa que se encarga de convertir el MP3 a PCM, y de ahí enviarlo a la tarjeta de sonido, por supuesto que realizando todo esto en tiempo real (lo va descomprimiendo mientras se oye el tema).

Lo que se conoce popularmente como MP3 es en realidad la especificación ISO-MPEG I Layer III, un estándar de compresión de audio creado conjuntamente por el Instituto Fraunhofer de Alemania y la Universidad de Erlangen. El nombre popular del sistema viene de la extensión que comúnmente se da a los archivos comprimidos con este formato. Así como existe el layer III, también están el layer I y el layer II, los cuales son versiones previas, que brindan una menor compresión si se mantiene la calidad de audio constante. A su vez, como existe el MPEG I [1]para audio, esta la especificación para video (que es la que se usa para comprimir en el formato VideoCD), y otras mas avanzadas como la MPEG II (la que se usa en los SuperVideo CDs y los DVDs), y la MPEG IV (el estándar implementado en el formato DIVX y XVID).[2].

Realizando una comparación entre los tres layers del MPEG I, podemos decir que el layer I ofrece una compresión de 1:4, el II una compresión de 1:6 y el III de 1:10 (con un MP3 de 128 kbit/s como referencia, lo cual según el estándar garantiza una compresión con perdida imperceptible de calidad). Esto reduce una hora de audio a algo más de 60 MB de información.

---

1 Sitio de referencia: www.mpeg.org y askmp3.com

2 Sitios de referencia: www.divx.com (comercial) y www.xvid.org (open source).

Las técnicas utilizadas en el formato MP3 para lograr una reducción de tamaño de uno a diez son varias. El proceso de compresión comienza por el filtrado del audio mediante un filtro híbrido que consiste en una serie de filtros polifásicos y una Transformada Discreta Coseno Modificada (MDCT). La razón por la cual se usa un híbrido es por compatibilidad con los Layers I y II. Este filtrado es a su vez controlado por un modelo perceptual del oído humano, cuya función es controlar que el ruido de compresión y cuantificación introducido en el proceso se mantenga por debajo del nivel de umbral máximo permitido para la calidad que se desea lograr. En el caso de archivos stereo, se aplica la técnica de "joint stereo" que consiste en aprovechar el hecho de que gran parte de la información de los canales derecho e izquierdo es la misma la mayor parte del tiempo, con lo cual se logra una reducción "rápida" a casi la mitad de espacio necesario.

Luego de esta reducción por filtrado controlado, sigue la etapa de cuantificación y codificación, donde generalmente se usa un sistema anidado de doble iteración. La cuantificación se realiza elevando las muestras a la potencia ¾ (0.75). De esta manera, los valores mas grandes se codifican automáticamente con menor precisión (y por lo tanto con mayor ruido de cuantificación), lo cual implica que una forma de conformado de dicho ruido (noise shaping) ya se incluye directamente en el proceso.

Es oportuno recordar que el ruido de cuantificación es aquel que se produce al reducir la precisión de las muestras digitales (por ejemplo, de 16 bits a 8 bits). Cuando se hace esto, se reduce un grupo grande de valores y se los asigna a uno solo (por ejemplo, en el paso de 16 bits a 8 bits, todos los valores que están entre 0 y 255 en 16 bits, se asignan al valor 0 en 8 bits). Este redondeo introduce un ruido que se puede percibir auditivamente, y tiene características aleatorias ya que depende de la diferencia entre las muestras originales y las redondeadas (por ejemplo, si el valor original es 0 y se lo redondea a 0, no hay ruido introducido...pero si el valor original es 255, el ruido introducido es máximo...y nuevamente, si el valor es 256, se redondea a 1, con lo cual no hay ruido agregado).

A continuación, los valores cuantificados son codificados con un código Huffman. Este código no posee pérdidas de información (lossless), por lo cual no introduce ruido adicional al proceso. En el código Huffman, se asignan códigos más cortos a las cadenas más comunes. La determinación de la asignación de los códigos puede requerir varias iteraciones del proceso citado (cuantificación-codificación). Si luego de un primer intento no se logra la cantidad de bits por segundo máxima permitida, se ajustan las ganancias del proceso de cuantificación (aumentando el paso de cuantificación, e incrementando el ruido en consecuencia), y se codifica nuevamente hasta lograr el resultado deseado.

Mientras este proceso se repite, el modelo de percepción humana actúa controlando que el ruido en cada banda en las que se divide el espectro de audio no supere el nivel critico a partir del cual el oído humano puede distinguirlo (esto es a causa de un efecto llamado enmascaramiento, que impide al oído detectar sonidos de baja intensidad si estos están presentes frente a otros de gran intensidad, algo así como lo que sucede cuando se intenta ver un objeto que esta en frente de una gran fuente de luz...el ojo humano solo distingue la luz, y nada que sea de menor brillo).

Si algún nivel se supera, el proceso se reajusta, aumentando la cuantificación en la banda necesaria, hasta que finalmente se logra la compresión del audio con las especificaciones requeridas.

A consecuencia de la complejidad y el proceso iterativo que implican la compresión de un archivo MP3, esta no se puede realizar en tiempo real, salvo con las computadoras de mayor potencia. En cambio, la descompresión si se puede realizar a una buena velocidad, e incluso un procesador de casi una década de antigüedad puede manejar la tarea sin mayores complicaciones, y con código escrito en lenguaje de alto nivel (lo cual implica una desmejora en la velocidad de decodificación), como se verá a lo largo de este informe.

Existen otras variantes dentro de la compresión MP3, como por ejemplo el cambio automático de los bits por segundo que se utilizan. Esta técnica, conocida como VBR (Variable Bit Rate), se opone a la tradicional CBR (Constant Bit Rate), y es útil por ejemplo en archivos de audio que mezclan tramos de voz humana (de bajo contenido espectral, pasible de ser codificada con menor cantidad de bits por segundo), y otros de música (que requieren mayor cantidad de bits por segundo para mantener la calidad).

Alternativas para la implementación

Reproducir un archivo MP3 es una tarea compleja, pero no imposible con el poder de procesamiento disponible en la actualidad. La alternativa mas simple, y la mas viable comercialmente, es utilizar uno de los varios circuitos integrados disponibles que aceptan en su entrada un flujo de datos MPEG-I, y a su salida entregan el audio en estéreo listo para amplificar. Esto permite armar un equipo comercial con un mínimo de componentes.

Otra alternativa es tomar una computadora, desde un 486 en adelante, cargarle el software necesario[3], y reproducir sin problemas. Una variante de esto es crear una placa ISA o PCI, con una extensión del BIOS cargada en una ROM, para alojar allí un programa reproductor de MP3 (o un sistema operativo completo como Linux para sistemas "embedded"[4], para poder manejar así discos rígidos, lectores de CD, etc.), y evitar la necesidad, falibilidad y lentitud de un disco rígido o disquetera desde la cual cargar los programas.

La tercera alternativa es tomar un procesador cualquiera, de poder de procesamiento adecuado, programar el código necesario, armar una placa con la memoria y circuitos accesorios necesarios, y luego de una larga serie de pruebas y fallas, lograr el éxito. Esta es la alternativa más compleja, y por supuesto es la que se ha elegido.

---

3 En www.winamp.com se encuentra el reproductor mas popular del momento.

4 Ver www.linuxdevices.com para encontrar diferentes distribuciones de Linux para sistemas "embedded".

Cabe aclarar que el objetivo único de este proyecto no es crear un reproductor de MP3, sino el mostrar como se puede utilizar en una aplicación práctica un procesador de la empresa Intel (el estándar a nivel mundial), disponible en gran cantidad, a bajo precio, y compatible con la mayor parte del software existente. De ahí en más, las aplicaciones que no requieran del tamaño ni la complejidad de un motherboard de PC, y que deseen aprovechar las citadas ventajas, pueden hacer un uso similar del procesador al hecho en este trabajo.

En la historia del proyecto existieron dos diseños: uno de mayor complejidad (el primero), que contenía una interfaz IDE, 2 MB de memoria ROM, y posibilidades de expansión y otro que fue el que finalmente se construyó. Con la tecnología disponible en Mar del Plata (que permite hacer circuitos impresos con caminos de un ancho mínimo de medio milímetro), una placa de tales características era muy difícil y grande de llevar a cabo, por lo cual se debió recurrir a un sistema de menor complejidad, con el objetivo de demostrar la factibilidad de la idea. Si bien el diseño de mayor complejidad no fue llevado a la práctica, el mismo es completamente funcional, y se explicará a modo informativo.

La placa construida consta de un procesador Intel i486! DX4 de 100 Mhz (frecuencia externa de 33 Mhz) con 16kb de cache interno, 128 KB de memoria estática de tipo cache (15 nanosegundos de tiempo de acceso), un circuito de interrupciones para generar la frecuencia de muestreo necesaria, un circuito para la interfaz JTAG y un circuito de salida analógica, que hace uso de un único conversor digital de 8 bits (la salida es mono).

En lo que sigue del informe se dará una explicación detallada de los componentes que conforman la placa reproductora de MP3 y su funcionamiento, asi como un recorrido paso a paso por el software que permite reproducir los sonidos comprimidos.

Capítulo

Hardware

Este capitulo dedicado al hardware del proyecto comienza con un repaso general por la placa reproductora de MP3, para luego ahondar en la funcionalidad de las diferentes secciones y su relación con el componente central, el procesador i486!

El i486! es un procesador de 32 bits, con lineas de datos y direcciones del mismo ancho. Esto le permite direccionar un total de 4 GB de información física, los cuales se amplían hasta 64 GB gracias al uso del sistema de paginación. El procesador trabaja con una velocidad máxima de bus de 33 Mhz, pero necesita al menos dos ciclos de reloj para acceder a la memoria (salvo en el modo burst, no utilizado en este diseño, donde necesita dos ciclos en la primera transferencia, y uno en las subsiguientes). Por lo tanto el acceso a memoria más rápido consume un total de 60 ns. Esto parece más que suficiente para las memorias cache utilizadas (de 15 nanosegundos de tiempo de acceso), pero como se verá mas adelante, debido a la lógica de decodificación de señales utilizada, no siempre es fácil cumplir con esta especificación. Para los casos donde no se pueda cumplir con esta velocidad (por ejemplo, en las memorias ROM con tiempo de acceso de 250 ns), el procesador permite introducir estados de espera ("wait-states") para dar tiempo al hardware externo a poder cumplir su función.

Junto con la memoria (4 integrados de 32 kb cada uno, extraídos de un motherboard para 486), se incluye un oscilador para el sistema de 33 Mhz construido con un 74F04, un oscilador de 3,57 Mhz para el generador de interrupciones, construido con un 74LS04, tres buffers 74HCT374 que funcionan como puertos de salida (uno para grabar el código de interrupción, otro para el audio digital decodificado, y otro para programar la frecuencia del generador de interrupciones), dos contadores 74C193 para dividir la señal de 3,57 Mhz y generar las interrupciones, un FF tipo "D" 74F74 de función múltiple, un 74F138 y una serie de compuertas lógicas para decodificar las señales del procesador, un DAC08 para la conversión digital-analógica, un TL082 que funciona como buffer de la señal analógica de salida, y finalmente un buffer 74HC244 para la interfaz con el puerto paralelo de la PC, que a su vez controla la interfaz JTAG del procesador.

El esquemático

Se reproduce a continuación el esquemático completo del reproductor de MP3. El mismo también puede ser consultado en la información digital que acompaña este informe, en formato para el programa Orcad 9.2

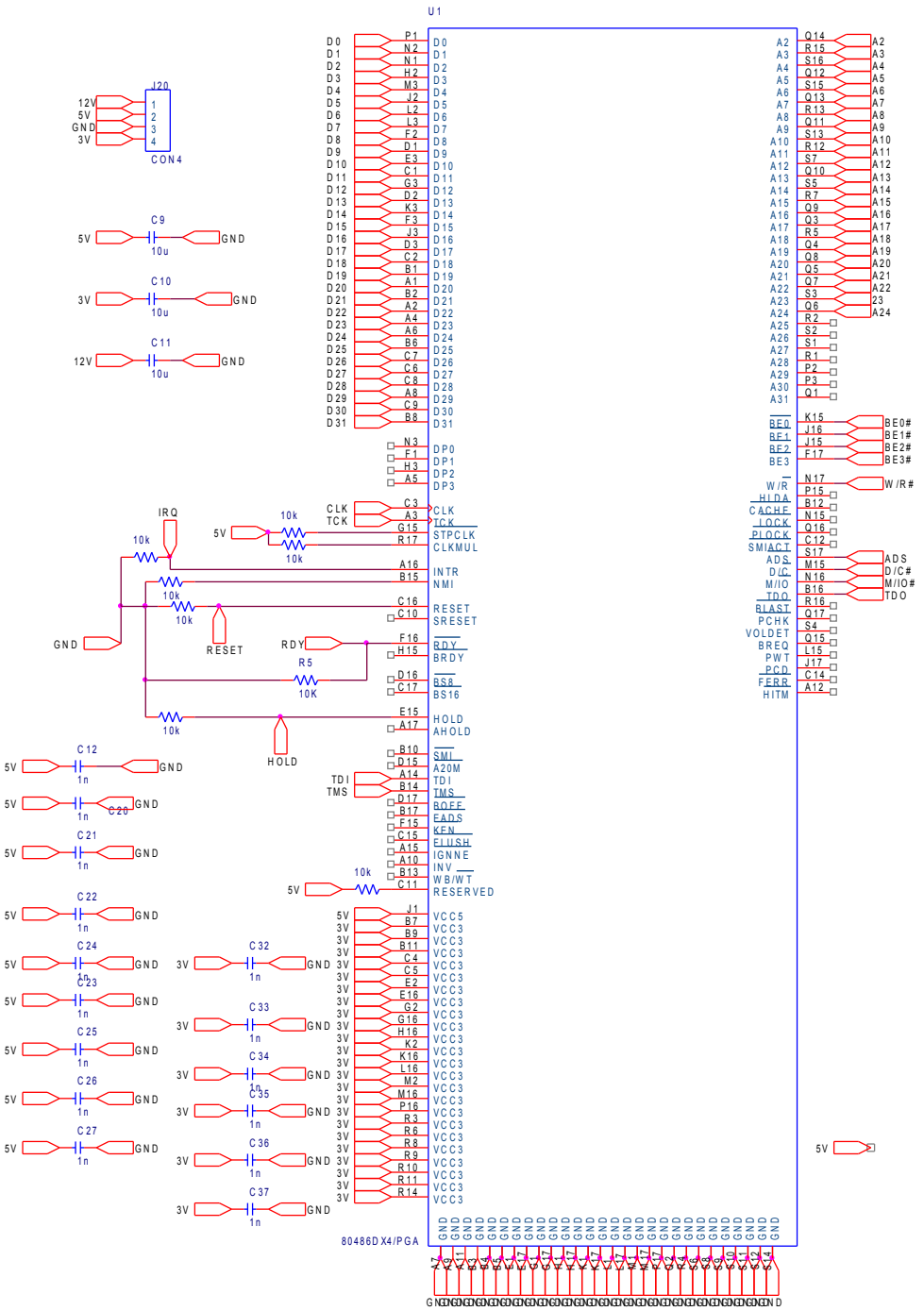

IMAGEN 1 Aquí se puede ver el procesador i486!, junto con las señales utilizadas, algunas resistencias de pull-up o pull-down según el caso, y los capacitores de desacople necesarios para toda la placa.

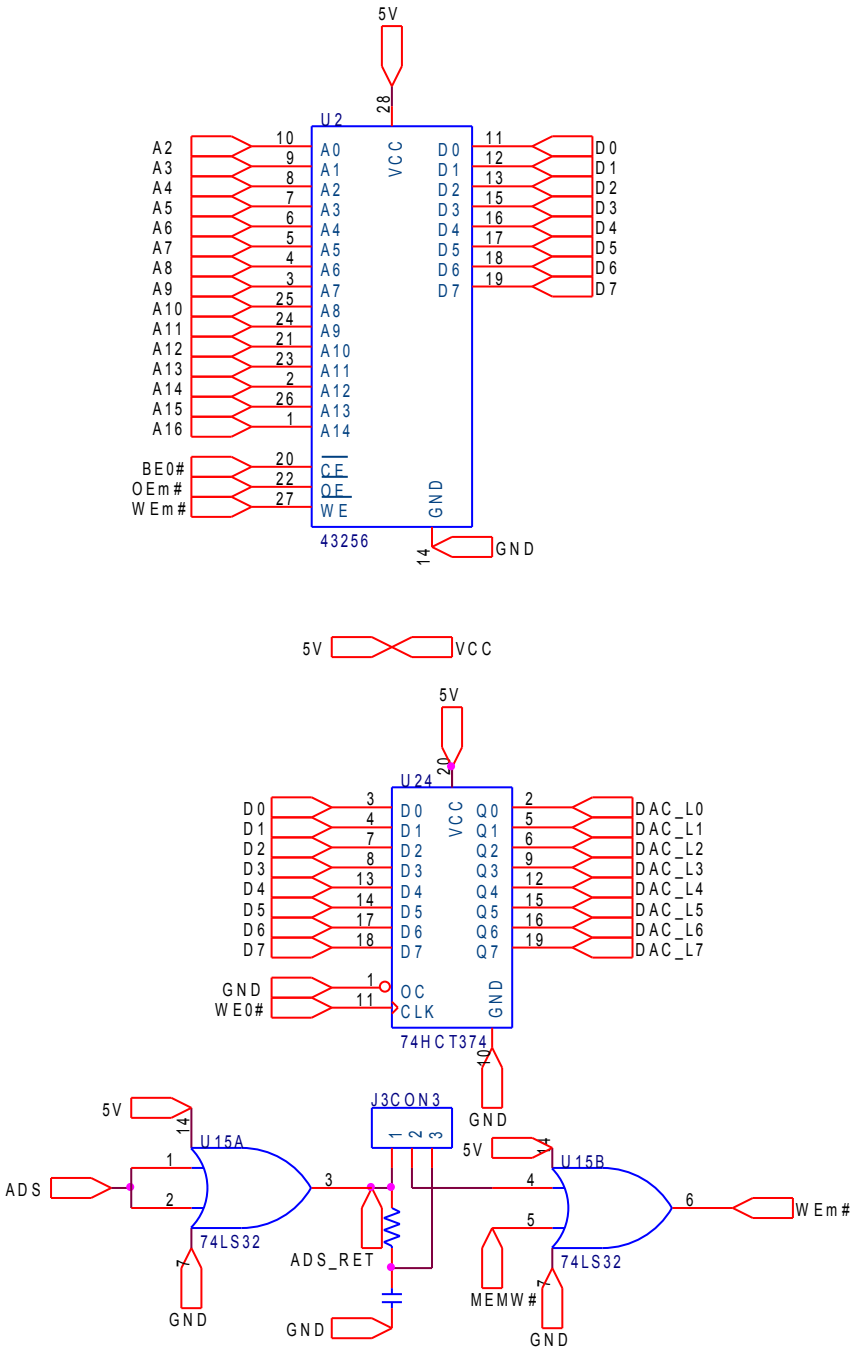

IMAGEN 2 Una de las memorias, el circuito de salida hacia el DAC, y el circuito que genera la señal de escritura de memoria.

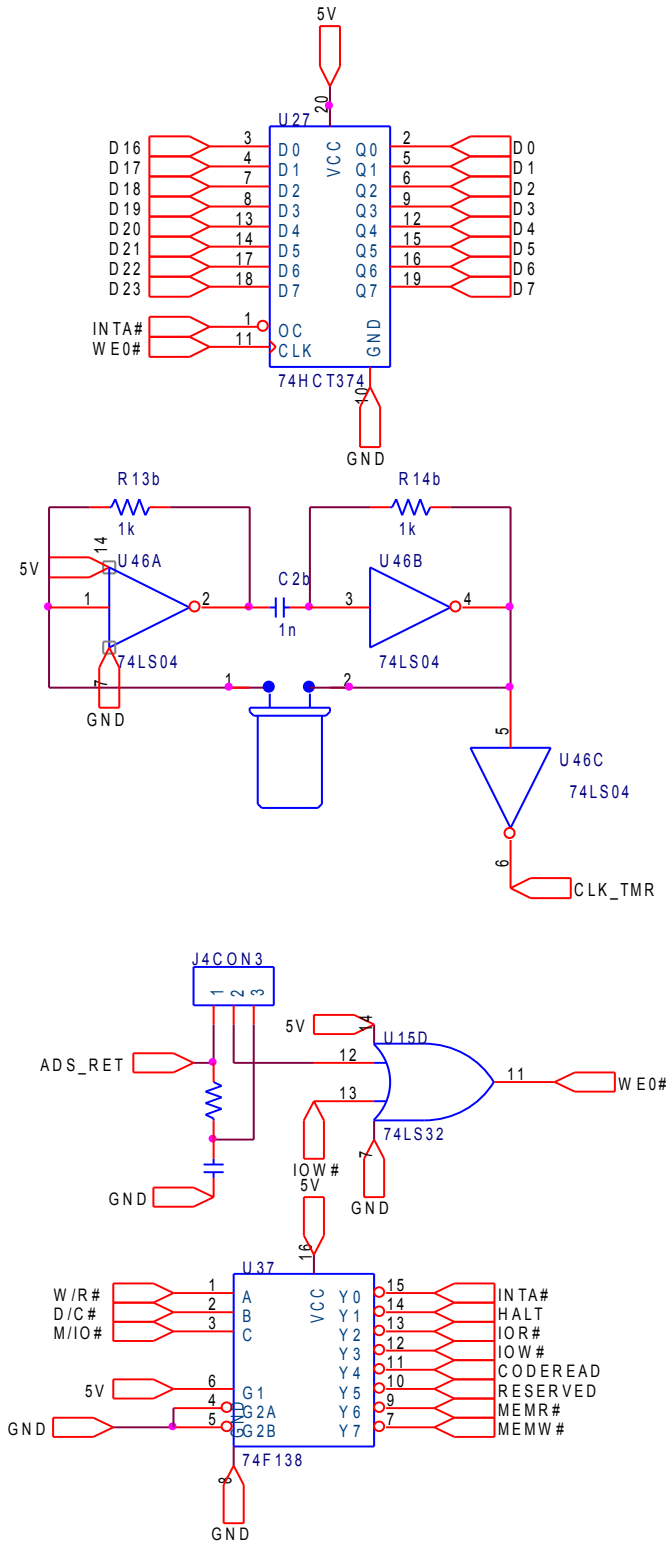

IMAGEN 3 El circuito que provee el código de interrupción, el oscilador de 3,57 Mhz, el circuito que genera la señal de escritura para los puertos, y el decodificador de tipo de ciclo de bus.

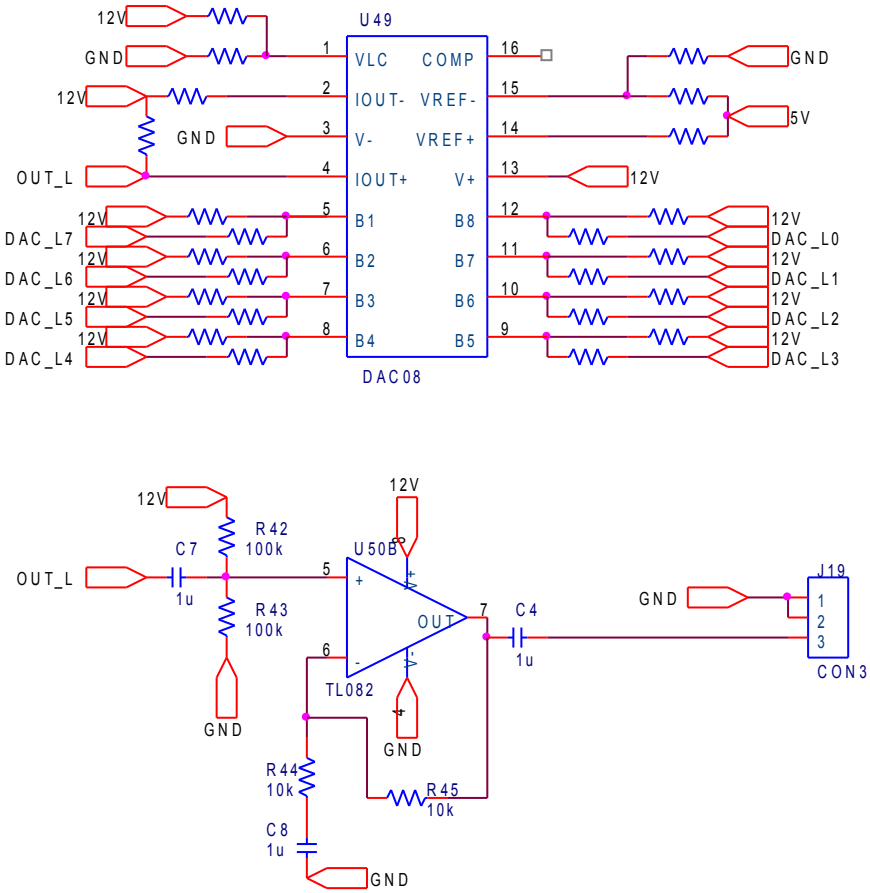

IMAGEN 4 El DAC y el TL082 que actúa como buffer de audio.

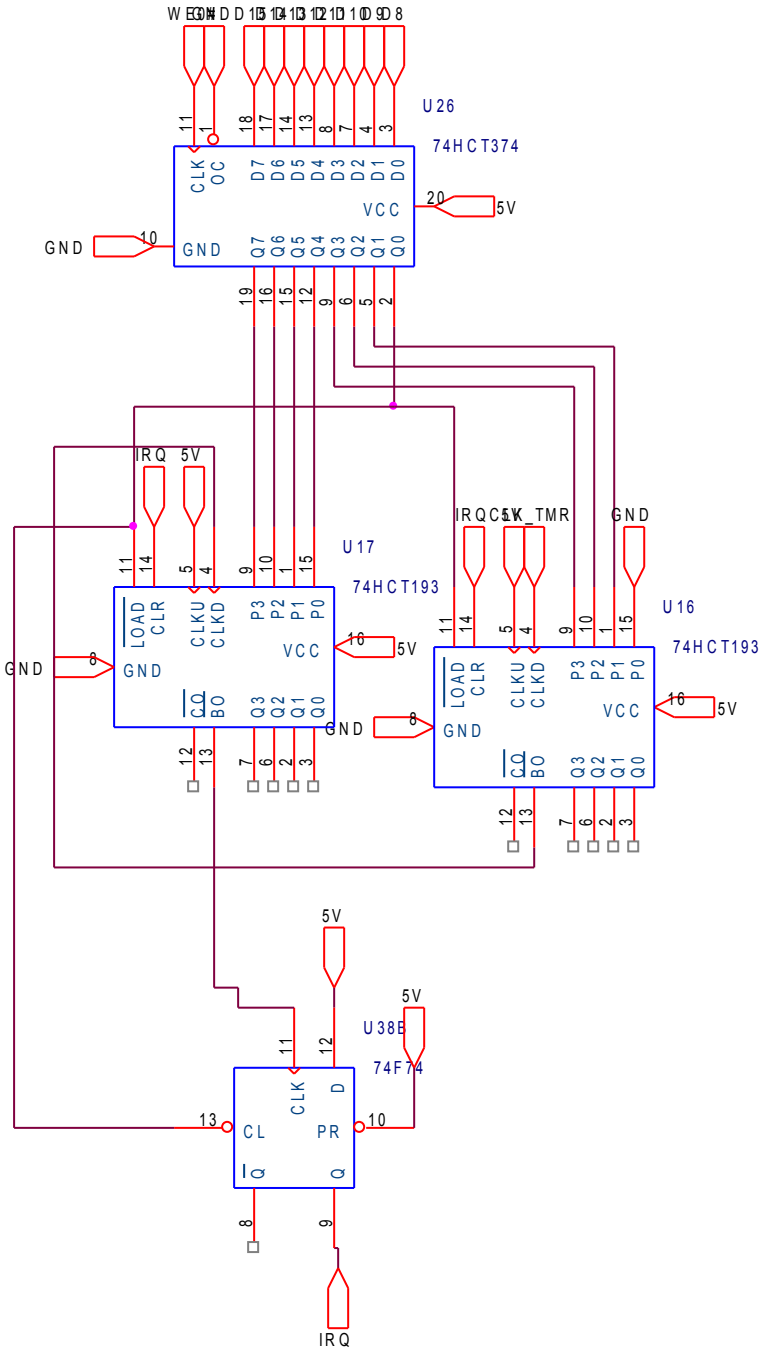

IMAGEN 5 El circuito que genera las interrupciones.

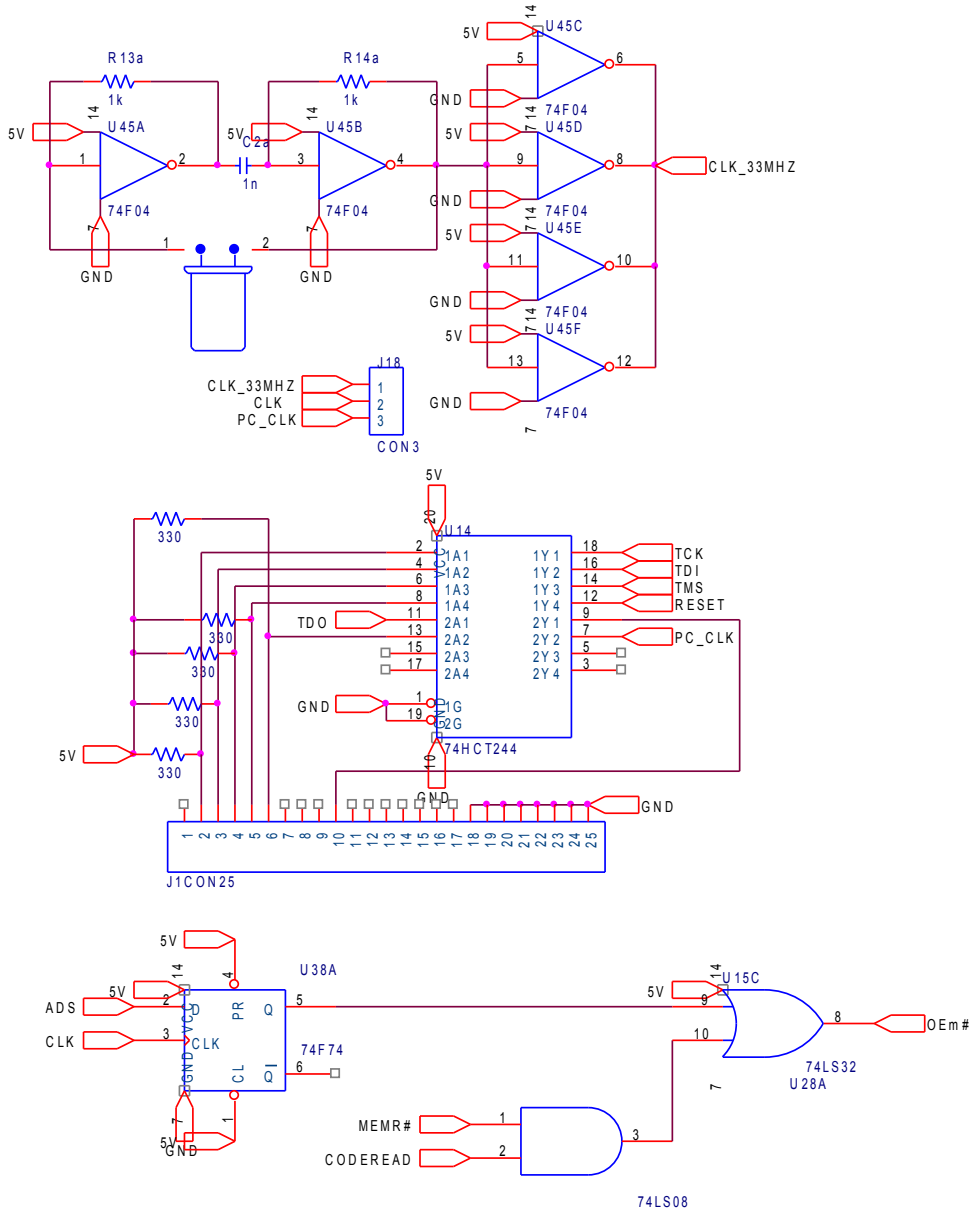

IMAGEN 6 El oscilador de 33 Mhz, la interfaz con el puerto paralelo, y el circuito generador de la señal de lectura de memoria.

Estas imágenes representan el total del circuito utilizado. En las siguientes secciones se irá detallando su funcionamiento. El primer componente a analizar es el mas complejo e importante, el procesador i486!

i486!

Características Principales

El procesador Embedded Write-Back Enhanced IntelDX4 ofrece las siguientes características:[5]

- ✓ **Núcleo de 32 bits con tecnología RISC:** El i486! tiene un set completo de funciones lógicas y aritméticas que funcionan con datos de 8, 16 y 32 bits usando una ALU de longitud completa, y ocho registros de uso general.
- ✓ **Ejecución en un solo ciclo**: Muchas instrucciones ejecutan en un solo ciclo.
- ✓ **Instruction Pipelining**: El procesador puede realizar de forma solapada las tareas de leer código, decodificarlo, ejecutarlo y traducir las direcciones virtuales a físicas.
- ✓ **Unidad de Punto Flotante interna**: El i486 incluye una FPU dentro del integrado mismo, que soporta los datos estándar de la especificación 754 del IEEE (32, 64 y 80 bits). La unidad es compatible con los coprocesadores anteriores (8087, 80287 y 80387)
- ✓ **Cache interno y control de consistencia de cache:** La unidad de cache interna de 16 kb se usa tanto para datos como para instrucciones. Se la puede configurar en los modos "write-back", "write-through" o como memoria estática de alta velocidad. Se monitorea la actividad del bus externo para realizar modificaciones en el cache interno, y un controlador externo de cache tiene la posibilidad de desagotar o invalidar el cache interno mediante una serie de señales a tal efecto.
- ✓ **Unidad de manejo de memoria interna:** La unidad de manejo y protección de memoria es central en el modo protegido de la CPU. Además permite la implementación simple de sistemas multitarea y con paginación.
- ✓ **Transferencias por ráfagas:** La transferencia por ráfagas permite que se ingresen o extraigan del procesador hacia la memoria 32 bits de

5 Extraido de la hoja de datos del procesador "Embedded Write-Back Enhanced Intel DX4" que se puede obtener gratuitamente de www.intel.com.

información por cada ciclo de reloj, lo cual resulta particularmente útil al llenar una línea de cache, que consta de 16 bytes.

- ✓ **Buffers de escritura:** El procesador tiene cuatro buffers de escritura, que pueden reordenar los datos en espera de ser escritos, para optimizar los accesos a memoria, sin detener el procesamiento interno de nuevas instrucciones.

- ✓ **Abstención de usar el bus:** Cuando otro controlador o CPU necesita acceso al bus, el procesador puede poner todas sus señales en alta impedancia, mientras monitorea las señales que circulan por el bus, para poder mantener la consistencia del cache interno con la memoria del sistema.

- ✓ **Re-ejecución de instrucciones:** Los programas pueden continuar su ejecución normal luego de que se genera una excepción producto de un acceso a una posición de memoria no disponible, lo cual permite implementar un sistema de paginación transparente al usuario final.

- ✓ **Redimensionado dinámico del bus:** Cada ciclo de bus puede ser redimensionado de forma dinámica e independiente de otros ciclos para utilizar 8, 16 o 32 bits.

- ✓ **Interfaz JTAG:** La interfaz serie JTAG permite programar individualmente todos los pines del procesador, asi como monitorearlos durante su funcionamiento normal.

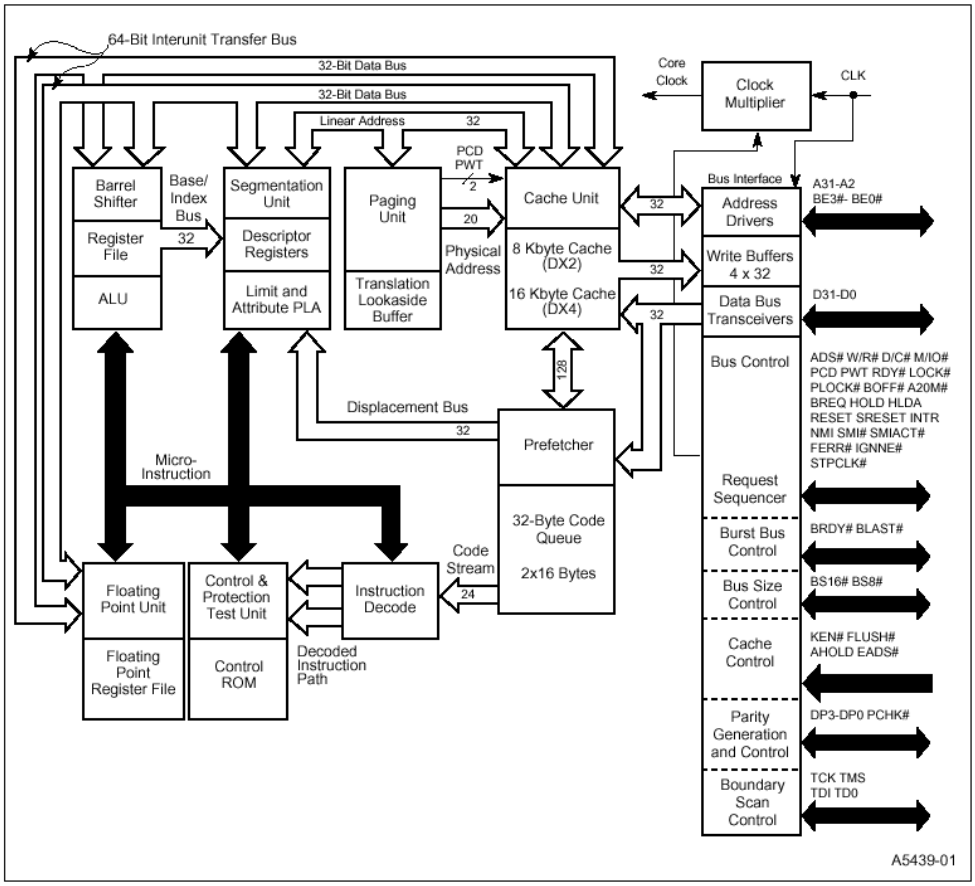

IMAGEN 7 Diagrama en bloques completo del procesador.

Resumen de las funciones de los pins

El i486! viene en un zócalo tipo PGA de 168 pines, de los cuales más de 50 pertenecen a los 3,3 V que necesita de alimentación y el camino de la masa. Se detalla en el cuadro que se encuentra a continuación las señales utilizadas en el diseño (para conocer la función del resto de las señales del procesador consultar la hoja de datos del mismo). La razón de esto es para permitir una mejor comprensión del funcionamiento del circuito, sin complicar excesivamente con señales utilizables solamente en motherboards para PC.

En la especificación que sigue de los pins del procesador el signo "#" significa "activo bajo".[6]

6 Se resumen las señales utilizadas en la placa. Las señales de control de cache, y otras no utilizadas pueden consultarse en la hoja de datos del procesador.

<table>
<tr><th>PIN</th><th>TIPO</th><th>DESCRIPCIÓN</th></tr>
<tr><td>CLK</td><td>I</td><td>La señal de clock provee la temporización fundamental y la frecuencia de operación interna. Todos los parámetros de funcionamiento externo se miden en relación a esta señal.</td></tr>
<tr><td colspan="3">BUS DE DIRECCIONES</td></tr>
<tr><td>A31-A4
A3-A2</td><td>I/O
O</td><td>Las lineas de direcciones junto con las señales de habilitación de byte (byte enable) BE3# a BE0# definen el área física de memoria o del espacio de entrada/salida que se esta accediendo. Durante el funcionamiento normal, las señales son siempre de salida. Solo cuando el procesador entra en el modo “Backoff”, se monitorean las señales A31-A4 para realizar la invalidación automática del cache interno si fuese necesario. En total el procesador puede direccionar 4 gigabytes de datos de memoria, mas 64 kilobytes de espacio de entrada/salida (lo cual se selecciona mediante la señal M/IO#).</td></tr>
<tr><td>BE3#-
BE0#</td><td>O</td><td>Como el bus del procesador es de 32 bits, se utilizan cuatro señales para activar individualmente cada uno de los bytes (BE3# activa de D31 a D24 y BE0# activa de D7 a D0).</td></tr>
<tr><td colspan="3">BUS DE DATOS</td></tr>
<tr><td>D31-D0</td><td>I/O</td><td>Las lineas de datos de entrada salida permiten la comunicación de hasta 4 bytes en forma simultánea. Si se utiliza un ancho de palabra menor, los bytes deben aparecer en las lineas correspondientes a lo indicado por BE3#-BE0#.</td></tr>
<tr><td colspan="3">DEFINICIÓN DEL TIPO DE CICLO DE BUS</td></tr>
<tr><td>M/IO#
D/C#
W/R#</td><td>O
O
O</td><td>Estas señales definen el tipo de ciclo de bus que el procesador esta ejecutando. La primera de las señales indica si se trata de un acceso al espacio de memoria o de entrada/salida. La segunda indica si se trata de datos o de código (o control) y la tercer señal indica si se trata de lectura o escritura.
Las combinaciones posibles son las siguientes:
<table>
<tr><th>M/IO#</th><th>D/C#</th><th>W/R#</th><th>Función</th></tr>
<tr><td>0</td><td>0</td><td>0</td><td>Interrupt Ack</td></tr>
<tr><td>0</td><td>0</td><td>1</td><td>HALT/Special</td></tr>
<tr><td>0</td><td>1</td><td>0</td><td>IO Read</td></tr>
<tr><td>0</td><td>1</td><td>1</td><td>IO Write</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>Code Read</td></tr>
<tr><td>1</td><td>0</td><td>1</td><td>Reserved</td></tr>
<tr><td>1</td><td>1</td><td>0</td><td>Memory Read</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>Memory Write</td></tr>
</table>
Nótese que el procesador genera un tipo de ciclo diferente cuando intenta leer un segmento de código (Code Read), que cuando intenta leer un segmento de datos (Memory Read). Esto no es un comportamiento estándar en procesadores anteriores, y tampoco se explica claramente en</td></tr>
</table>

		la hoja de datos, a pesar de que es un dato crucial para el diseño.
CONTROL DEL BUS		
ADS#	O	Address Status indica cuando hay un ciclo válido definido en los pines del procesador. ADS# se activa durante el primer flanco de reloj del ciclo, y se desactiva en el segundo y los subsiguientes (en el caso de que el ciclo dure mas de dos flancos positivos de reloj a causa de la inserción de estados de espera).
RDY#	I	Non-burst Ready se usa para indicar al procesador cuando la circuitería externa esta lista para aceptar o entregar datos. Si RDY# se mantiene desactivado, el procesador agrega "wait-states" al ciclo de bus (lo cual es útil cuando se accede a memorias o dispositivos en general mas lentos que el procesador). Existe otra señal, llamada BRDY#, que no se utiliza en este diseño y que sirve para activar el modo "burst" de transferencia de datos, que consume dos pulsos de reloj en la primer transferencia de datos, y un pulso en las subsiguientes (hasta un máximo de 16 ciclos). Si se activa la señal RDY#, la señal BRDY# es ignorada.
INTERRUPCIONES		
RESET	I	Manteniendo esta señal activa durante 15 ciclos de reloj fuerza al procesador a retornar a su estado inicial, lo cual también implica la puesta en un estado conocido de todos los registros internos del procesador.
INTR	I	La señal de interrupción enmascarable sirve para ejecutar interrupciones, las cuales en el caso del diseño presente se usan junto con un contador por software y uno por hardware para generar la frecuencia de muestreo de sonido adecuada.
TAP (PUERTO DE ACCESO PARA PRUEBAS – JTAG)		
TCK	I	Test Clock, la circuitería compatible con el estándar JTAG funciona al ritmo de esta señal.
TDI	I	Test Data Input, entrada de datos en forma serie. Se muestrea en el flanco positivo de TCK.
TDO	O	Test Data Output, salida de datos en forma serie, cambia de estado en el flanco negativo de TCK.
TMS	I	Test Mode Select, señal complementaria que permite seleccionar el estado actual de la máquina de estados finitos que controla la interfaz JTAG (permitiendo asi con una sola entrada serie ingresar a varios registros y funciones).
OTROS		
CLKMUL	I	Selección de multiplicación interna. En estado alto multiplica el reloj externo por tres, y en estado bajo por dos.
VCC5	I	Referencia de 5V para buffers TTL o CMOS de 5V.
RESERVED	I	Debe conectarse a VCC mediante resistencia de pull-up

Recuérdese que las señales citadas son solo algunas de las que posee el procesador. La mayoría de las señales para control de cache externo, arbitraje de bus con otros procesadores o DMA, y otras no se han mencionado porque no son necesarias para el diseño simplificado que utiliza este reproductor, aunque no por eso han sido ignoradas en el diseño. Un ejemplo de esto es la señal NMI (non maskable interrupt), que debe fijarse a un valor mediante una resistencia, para que no se ejecuten interrupciones al azar. Muchas de las señales ya poseen resistencias de pull down o pull up dentro de la pastilla del integrado, aunque en algunos pines particularmente se recomienda en la hoja de datos colocar una resistencia adicional de forma externa.

Como se observará en los diagramas de tiempo de la hoja de datos, la mayoría de las señales cambia con el flanco positivo del reloj, y tienen un retardo nominal que varia entre los 3 a 10 nanosegundos. Estas especificaciones son para un sistema de 3,3 V (el sistema implementado es un híbrido de 3,3V y 5V), y con 50 pf de carga en la señal dada. Para obtener los datos finales (tiempo de hold y de setup, entre otros) a utilizar para realizar los diagramas de tiempos, se debe usar la curva de desnormalización provista en la hoja de datos del procesador, que contempla la carga capacitiva que tiene el procesador y el hecho de que se trabaja con buffers de 5V. Si se tiene en cuenta que el ciclo de reloj a 33 Mhz es de 30 ns, una incertidumbre de más de 10 ns en el cambio de la señal puede representar una seria dificultad de diseño.

Funcionamiento del bus

Cada vez que el procesador necesita un dato de la memoria, o necesita escribir, o aceptar una interrupción genera lo que se llama un "ciclo de bus". Este ciclo de bus se define mediante las señales M/IO#, D/C# y W/R#, junto con la señal ADS# que indica cuando comienza el ciclo. A su vez el circuito externo puede intervenir en estos ciclos, finalizándolos o alargándolos, o cambiando algunas de sus características (como por ejemplo, convertir un ciclo común en uno burst o viceversa, o cambiando el tamaño de la palabra en uso a 8, 16 o 32 bits).

En el reproductor de MP3 se utiliza un solo tipo de ciclo para acceder a la memoria (tanto para leer o escribir datos, como para cargar código y para escribir en los puertos): el ciclo de lectura/escritura básico, sin burst y sin estados de espera (non-burst y non-wait state). Esto es debido a que no se implementa la opción burst por simplicidad, y los estados de espera no son necesarios ya que todas las memorias son de 15 nanosegundos de tiempo de acceso (lo cual tiene la contrapartida de no tener el mayor espacio de almacenamiento que provee una memoria DRAM o una ROM). La totalidad de los tipos de ciclos que puede ejecutar el 486 se explican en el capítulo 10 del "Embedded INTEL486 Processor Family Developer's Manual" o en el capítulo 4 del "Embedded INTEL486 Hardware Reference Manual" (ambos capítulos son idénticos).

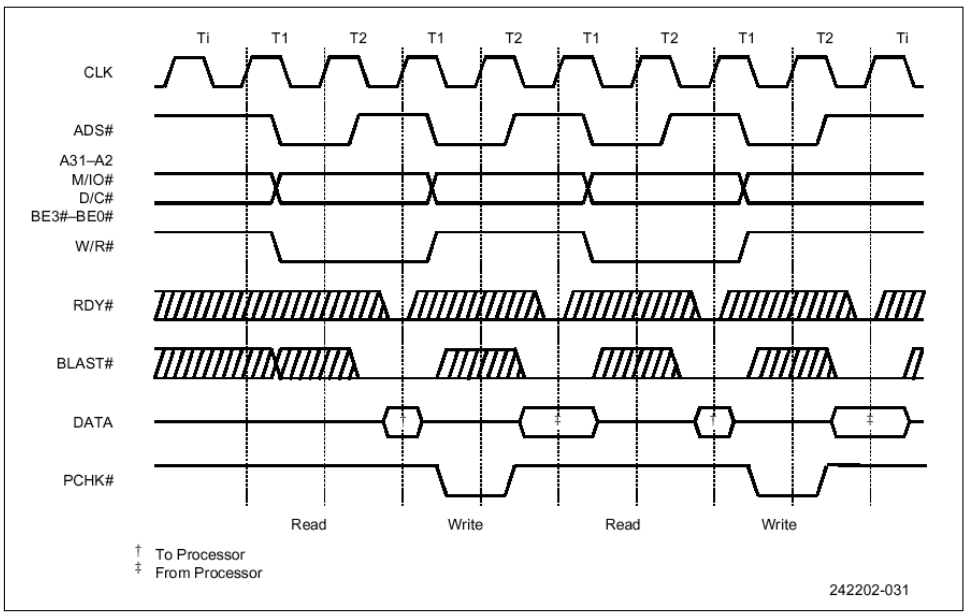

IMAGEN 8 Señales intervinientes en el ciclo de bus de lectura/escritura sin estado de espera y sin burst.

Como se ve en la Imagen 7, cada escritura o lectura (en espacio de memoria o de entrada/salida) toma dos ciclos de reloj. Luego del primer flanco positivo, la señal ADS# pasa a estado bajo (activa), indicando que las señales M/IO#, D/C#, W/R#, mas las de direcciones contienen datos válidos. En el segundo flanco de reloj, el procesador desactiva la señal ADS# y si se trata de un ciclo de escritura (W/R#=1), pone los datos en la señales D0-D31. En el tercer flanco de reloj (que estrictamente hablando es el primero del ciclo siguiente), el procesador lee los datos si se trata de un ciclo de lectura (W/R#=0), y lee la señal RDY#. Si esta se encuentra activa (nivel bajo), se procede al ciclo siguiente, sino se continua durante un pulso de reloj más (esto implica que el procesador mantiene los datos si se trata de escritura, o espera hasta el próximo flanco positivo del reloj para leer los datos del bus). En el diseño de este proyecto la señal RDY# esta fijada al valor cero, ya que tanto las memorias como los buffers que actúan como puertos de salida pueden responder en los tiempos requeridos.

Las señales PCHK# y BLAST# son para controlar la paridad de los datos y para indicar el ultimo dato de una secuencia burst respectivamente y no se utilizan en este diseño.

El ciclo recién explicado se usa tanto para acceder al espacio de memoria (4 gigabytes de datos y código), o al espacio de entrada/salida (64 kilobytes). La única diferencia entre estos dos espacios es que el procesador puede llegar a reordenar las escrituras al espacio de memoria (para optimizarlas), pero nunca reordena ni retrasa ni pone en el cache las escrituras o lecturas al espacio de entrada/salida (lo cual es absolutamente necesario en nuestro caso para poder escribir a un DAC y obtener el audio correctamente). Esta distinción no implica que no se pueda configurar un puerto de salida o entrada en una dirección de espacio de memoria. El único problema con esto es que el procesador puede llegar a poner los datos obtenidos

de allí en el cache, o reordenar las escrituras (o retrasarlas o no hacerlas nunca si esta en el modo WriteBack), y generalmente esto es perjudicial para las actividades típicas que se hacen con los puertos (como escribir a un DAC, o leer en tiempo real de un ADC).

Otro ciclo utilizado es el de INTA, que se usa para ingresar al procesador el vector de interrupción.

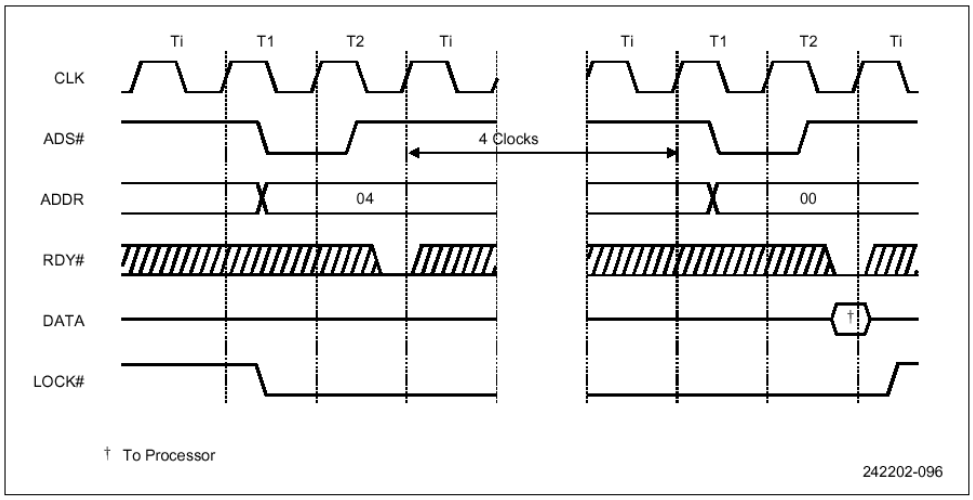

IMAGEN 9 Ciclo de Interrupt Acknowledge

Este ciclo parece más complicado, pero circuitalmente se resuelve de forma similar a una lectura de un puerto. En un circuito para un motherboard o aplicación de mayor complejidad quizás no sea posible resolver el ciclo de forma tan simple (de hecho, para tal efecto existen integrados administradores de interrupciones que hacen todo el trabajo necesario).

Es posible también en configuraciones con más de un procesador o con controlador de DMA (es decir, con más de un bus master), que algunos ciclos sean interrumpidos a la mitad (por ejemplo entre el primer y el segundo flanco de clock). Esta situación es contemplada por el procesador, y es manejada sin mayores dificultades. En la documentación citada se explica como se administran estas situaciones. Nuevamente, en este diseño esa dificultad no se presenta al haber solamente un procesador que controla el bus el 100% del tiempo.

Finalmente, una recomendación: si bien por motivos de simplicidad no se explican aquí todas las variantes de los tipos de ciclos de bus que pueden existir, en el manual del procesador se explica una gran cantidad de ellas, y es importante tenerlas en cuenta para cualquier diseño (sobre todo si este incluye memorias de distinta velocidad, DMA, etc.).

Puerto de acceso para pruebas (TAP)

El TAP es una parte esencial del procesador. Si bien se puede realizar un diseño sin prestar atención alguna a este circuito, lo cierto es que a la hora de las pruebas prácticas brinda una capacidad de evaluación y control de errores solo comparable con analizadores lógicos de alto costo. Y en algunos casos, ni siquiera donde un analizador lógico es aplicable, el TAP puede brindar soluciones.

Básicamente se trata de una máquina de estados finitos, que posee entrada y salida serie, y tiene varios registros internos de aplicaciones múltiples. El TAP implementa la especificación Test Access Port and Boundary Scan del IEEE (conocida como interfaz JTAG, standard IEEE 1149.1). Tiene la capacidad de comunicar a un controlador externo el estado de todos los pines del procesador sin entrometerse en el funcionamiento normal del mismo, o de "desconectar" los pines del procesador, y asignarles valores de forma arbitraria (lo cual es útil para programar la memoria, como se verá mas adelante).

Con toda la importancia que tiene para un ingeniero el poseer una herramienta confiable para identificar errores en el hardware, la importancia del TAP en este diseño es aun mayor, ya que como no se cuenta con memorias ROM, se utiliza el TAP para controlar los pines del procesador, y enviando las secuencias de datos correspondientes, programar las memorias RAM estáticas con el código y datos necesarios.

La información que a continuación se detalla proviene del Apéndice B ("Testability") del Embedded INTEL486 Processor Family Developer's Manual.

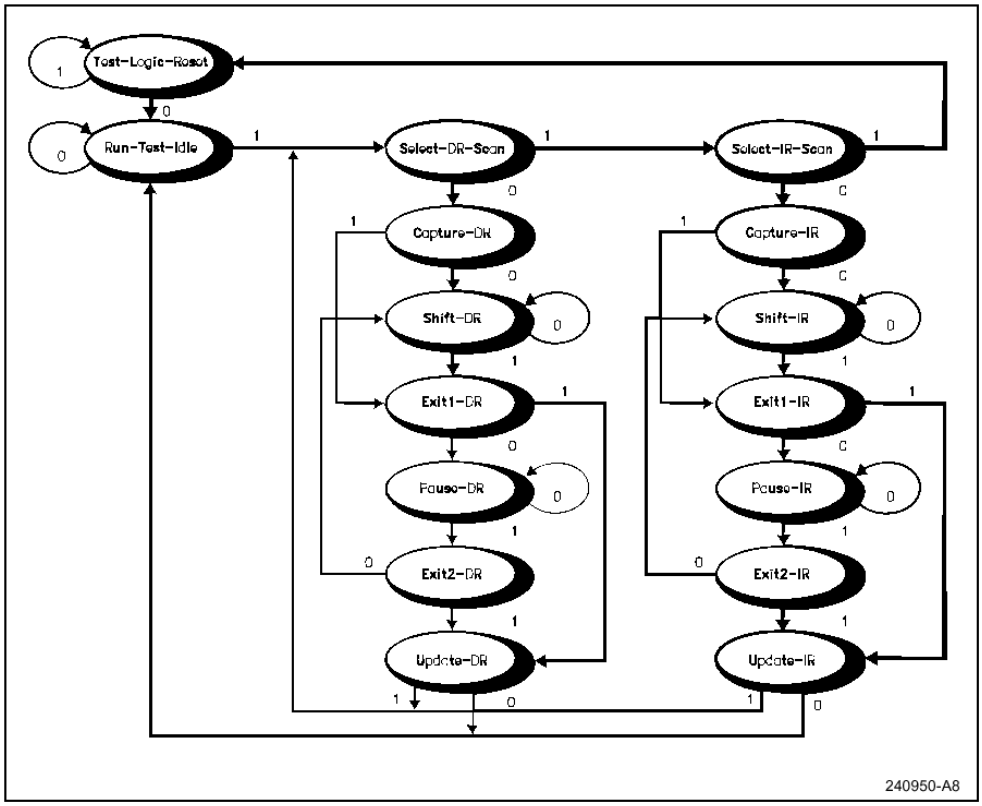

IMAGEN 10 Diagrama de estados del TAP.

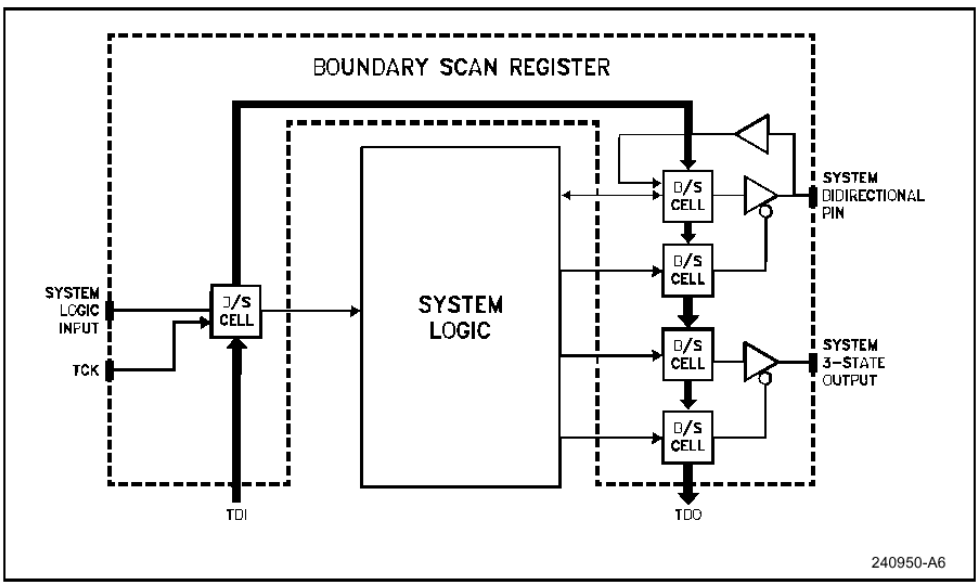

IMAGEN 11 Estructura lógica del boundary scan register.

La lógica de prueba del procesador incluye los siguientes elementos:

- ✓ Puerto de acceso para pruebas (TAP), que consiste de los pines TMS, TCK y TDI, más la salida TDO.
- ✓ Controlador TAP, que interpreta la entrada de la selección de modo de prueba (TMS) y ejecuta la operación correspondiente. Las operaciones realizadas por el TAP incluyen controlar los registros de datos e instrucciones dentro del componente.
- ✓ Registro de Instrucción (IR), el cual acepta códigos de instrucción, que se ingresan en forma serie a través de la señal TDI. Las instrucciones se usan para seleccionar una operación de prueba determinada dentro del TAP, o para acceder a un registro de datos en particular.
- ✓ Registros de datos (DR), de los cuales existen cuatro en el procesador 486. Estos son el Bypass Register (BPR), el Device Identification Register (DID), el Boundary Scan Register (BSR) y el Built In Self Test Register (BISTR).

Todos los registros de datos e instrucción están conectados entre TDI y TDO. Se accede a cada uno de ellos de acuerdo al estado del TAP, y la instrucción que esté cargada en el IR.

Registros de Datos: El procesador contiene los dos registros obligatorios según el estándar del IEEE, más un registro de identificación de dispositivo, propio de Intel y otro registro que almacena el resultado del Built In Self Test (BISTR). El registro de bypass es un registro de un solo bit, cuya función conectar TDI con TDO, y es útil cuando se esta probando la circuitería o cuando se tiene mas de un dispositivo conectado en serie a las señales del TAP. El Boundary Scan Register es el registro que se usa para programar o leer el valor de cada uno de los pines del procesador. La estructura de este registro se puede observar en la imagen 11. El orden de los bits se da mas adelante en este texto. El BISTR es un registro de un solo bit, que se coloca en uno antes de realizar una serie de pruebas internas para comprobar la integridad del procesador, y se pone en cero luego de terminadas las pruebas, si el resultado es exitoso. Finalmente el cuarto registro es el de identificación de dispositivo, que tiene un valor fijo almacenado (e inalterable por los datos que ingresen, los cuales son descartados mientras este registro este seleccionado), e identifica el procesador de acuerdo a lo siguiente:

Processor Type	V_{CC} 1=3.3V 0=5V	Intel Architecture Type	Family	Model	MFG ID Intel=009H	1st Bit	Boundary Scan ID (Hex)
Intel486‰ SX processor (3.3V)	1	000001	0100	00010	00000001001	1	x8282013H
Intel486™ SX processor (3.3V, 2X CLK)	1	000001	0100	00010	00000001001	1	x8282013H
Intel486™ SX processor (5V)	0	000001	0100	00010	00000001001	1	x0282013H
Intel486™ SX processor (5V, 2X CLK)	0	000001	0100	00010	00000001001	1	x0282013H
Intel486™ DX processor (3.3V)	1	000001	0100	00001	00000001001	1	x8281013H
Intel486™ DX processor (3.3V, 2X CLK)	1	000001	0100	00001	00000001001	1	x8281013H
Intel486™ DX processor (5V)	0	000001	0100	00001	00000001001	1	x0281013H
Intel486™ DX processor (5V, 2X CLK)	0	000001	0100	00001	00000001001	1	x0281013H
IntelDX2™ processor (3.3V)	1	000001	0100	00101	00000001001	1	x8285013H
IntelDX2™ processor (5V)	0	000001	0100	00101	00000001001	1	x0285013H
IntelDX4™ processor (3.3V)	1	000001	0100	01000	00000001001	1	x8288013H
Write-Back Enhanced IntelDX4™ processor (3.3V)	1	000001	0100	01001	00000001001	1	X8289013H

IMAGEN 12 Códigos de identificación de los procesadores de Intel.

En el caso del reproductor de MP3, se usa un procesador Intel 486DX4 WriteBack Enhanced. Sin embargo, ya que la pata WB/WT queda configurada como WT (Write Through) por defecto, y esto no se cambió externamente con una resistencia porque no afecta el diseño ya que el cache interno se usa como memoria estática y no como cache en si, el procesador responde y configura sus pines como si no fuera WriteBack Enhanced. Esto no es un dato menor, ya que cuando se ingresan los datos en forma serie al procesador vía el TAP, la configuración de los pines es distinta para WT que para WB, y no tener en cuenta esto puede provocar errores (y de hecho provocó errores durante la etapa de diseño del reproductor). En síntesis, el procesador usado retorna el código 08288013h (el bit mas significativo aparece primero, y luego los subsiguientes hasta el menos significativo).

Registro de Instrucciones: El registro de instrucciones permite que se ingresen en forma serie los códigos de instrucciones de 4 bits. El bit mas significativo del registro está conectado a TDI y el menos significativo a TDO (lo cual implica que los códigos de instrucciones deben ser ingresados comenzando por el menos significativo). Cuando el TAP ingresa al estado CAPTURE-IR el registro se carga

con la instrucción por defecto "0001" (Simple/Preload). Los bits ingresan desde TDI en el flanco positivo de TCK.

El conjunto de instrucciones posibles se resume en el siguiente cuadro:

CÓDIGO	INSTRUCCIÓN	CÓDIGO	INSTRUCCIÓN
0000	EXTEST	1000	PRIVATE
0001	SAMPLE	1001	PRIVATE
0010	IDCODE	1010	PRIVATE
0011	PRIVATE	1011	PRIVATE
0100	PRIVATE	1100	PRIVATE
0101	PRIVATE	1101	PRIVATE
0110	PRIVATE	1110	PRIVATE
0111	PRIVATE	1111	BYPASS

Las instrucciones privadas se reservan para uso futuro.

La instrucción EXTEST (0000) permite la prueba del circuito externo al procesador, y en el caso del reproductor de MP3, se usa para programar la memoria estática. Esto se hace colocando los valores almacenados en el Boundary Scan Register en los pines de salida del procesador, y capturando los valores presentes en los pines de entrada para trasladarlos a los bits correspondientes del BSR. Los valores que se ingresan en forma serial en los bits consignados como entrada del BSR son ignorados (su valor es reemplazado al ejecutarse la instrucción EXTEST por el valor presente en los pines del procesador). Los pines que pueden actuar tanto como entrada o salida se configuran en alguna de las dos funciones según los bits de control para tal efecto.

La instrucción SAMPLE/PRELOAD (0001) es similar a la EXTEST, pero no interfiere con el funcionamiento normal del procesador (en el caso de EXTEST, el circuito interno del procesador queda "desconectado" de sus pines). Cuando el TAP está en el estado CAPTURE-DR, y la instrucción SAMPLE/PRELOAD está cargada en el registro de instrucción, se toma una "foto" del estado de los pines del procesador (tanto de entrada como de salida) y esta información se guarda en el BSR. Posteriormente esta información puede ser leída en forma serie a través de TDO. Cuando el TAP está en el estado UPDATE-DR, el procesador efectúa una "pre-carga" de la información del BSR hacia el circuito que maneja los pines de salida, sin embargo los pines no cambian de valor hasta que se ejecute la instrucción EXTEST. Esto se hace para permitir el cambio "instantáneo" de todos

los pines, cosa que seria imposible de hacer debido a la característica serial de la entrada de datos.

La instrucción IDCODE (0010) conecta el registro de identificación de dispositivo entre TDI y TDO para poder identificar el procesador.

La instrucción BYPASS (1111) conecta el registro de bypass entre TDI y TDO. Esta instrucción no tiene otro efecto que trasladar sin cambios la información desde TDI hacia TDO, y se puede utilizar para realizar pruebas del circuito externo que tiene interfaz con el TAP, o para programar mas de un dispositivo JTAG colocado en cadena.

Finalmente, la instrucción RUNBIST (1000) ejecuta una serie de pruebas que evalúan la funcionalidad de todas las secciones del procesador. En caso de resultar exitosas todos los exámenes, coloca un cero en el registro BISTR, el cual puede ser leído desde TDO.

Máquina de estados finitos: En la imagen 10 puede observarse el total de los estados posibles en los que puede encontrarse el TAP. El cambio de estado del TAP se realiza mediante la señal TMS, la cual es leída en cada flanco positivo de TCK. El TAP no cambia nunca de estado si no es con un flanco positivo de TCK, o con el encendido del procesador. La señal de RESET, o cualquier otra fuera de las mencionadas, no lo afectan.

Los estados mas importantes son los siguientes:

ESTADO	DESCRIPCIÓN
Test Logic Reset	En este estado la lógica del TAP esta deshabilitada de forma tal que el funcionamiento del procesador se realiza normalmente. Al ingresar a este estado, el registro de instrucción se carga automáticamente con el valor 0010, correspondiente a IDCODE. No importa cual sea el estado en que se encuentre el controlador, se puede forzar al TAP al estado Test Logic Reset manteniendo TMS en 1, y generando cinco flancos positivos de TCK.
Run Test/ Idle State	Estado de espera. Si la instrucción es RUNBIST, el test se ejecuta cuando el TAP ingresa a este estado. Con cualquier otra instrucción la lógica de prueba no realiza ninguna actividad.
Capture DR	Si la instrucción es EXTEST o SAMPLE/PRELOAD, el Boundary Scan Register carga datos desde los pines del procesador en forma paralela.
Shift DR	En este estado el registro de datos seleccionado se conecta entre TDI y TDO, para efectuar su carga en forma serie, o para la lectura de los datos almacenados, o para ambas operaciones a la vez.

Update DR	Si la instrucción es EXTEST o SAMPLE/PRELOAD, el contenido del Boundary Scan Register es colocado en la salida paralela de dicho registro. Además, si la instrucción es EXTEST, esto implica también que esta información aparece en los pines del procesador.
Capture IR	En este estado el registro de instrucciones se carga con el valor 0001.
Shift IR	Similar a Shift DR, pero para el registro de instrucciones.
Update IR	En este estado es cuando la instrucción cargada durante Shift IR comienza a ejecutarse.

Bits del Boundary Scan Register: Como se mencionó anteriormente, los bits del registro citado cambian según si el procesador esta configurado con su cache como WriteBack o WriteThrough. A continuación se detallan la variante utilizada en el diseño (WriteThrough).

WriteThrough i486!: TDO <- A2, A3, A4, A5, RESERVED#, A6, A7, A8, A9, A10, A11, A12, A13, A14, A15, A16, A17, A18, A19, A20, A21, A22, A23, A24, A25, A26, A27, A28, A29, A30, A31, DP0, D0, D1, D2, D3, D4, D5, D6, D7, DP1, D8, D9, D10, D11, D12, D13, D14, D15, DP2, D16, D17, D18, D19, D20, D21, D22, D23, DP3, D24, D25, D26, D27, D28, D29, D30, D31, STPCLK#, IGNNE#, FERR#, SMI#, SMIACT#, SRESET, NMI, INTR, FLUSH#, RESET, A20M#, EADS#, PCD, PWT, D/C#, M/IO#, BE3#, BE2#, BE1#, BE0#, BREQ, W/R#, HLDA, CLK, AHOLD, HOLD, KEN#, RDY#, CLKMUL, BS8#, BS16#, BOFF#, BRDY#, PCHK#, LOCK#, PLOCK#, BLAST#, ADS#, MISCCTL, BUSCTL, ABUSCTL, WRCTL <- TDI.

En lo anterior se muestra el ordenamiento de los pines, más su conexión a la entrada y salida serie (TDO y TDI).

Los bits denominados como "*CTL", controlan el estado de los diferentes grupos de pines que son de entrada/salida. Si estos bits están en cero, el grupo de señales asociadas se configura como salida, de lo contrario se configuran en alta impedancia o como entrada. El bit WRCTL controla los bits de datos (D0:D31) y los de paridad (DP0:DP3)., mientras que el bit ABUSCTL controla las señales A2:A31. El bit MISCCTL controla las señales PCHK#, HLDA y BREQ y el bit BUSCTL maneja ADS#, BLAST#, PLOCK#, LOCK, W/R#, M/IO#, D/C#, BE0#, BE1#, BE2#, BE3#, PWT y PCD.

Descripción del circuito

Detalladas las partes mas importantes del procesador 486, continuamos por la descripción pormenorizada del circuito que conforma el reproductor de MP3. Se detallan todos los componentes de la placa, a excepción de algunas resistencias de

pull-up o pull-down que se conectan al procesador de acuerdo a lo recomendado en la hoja de datos del mismo.

Memoria

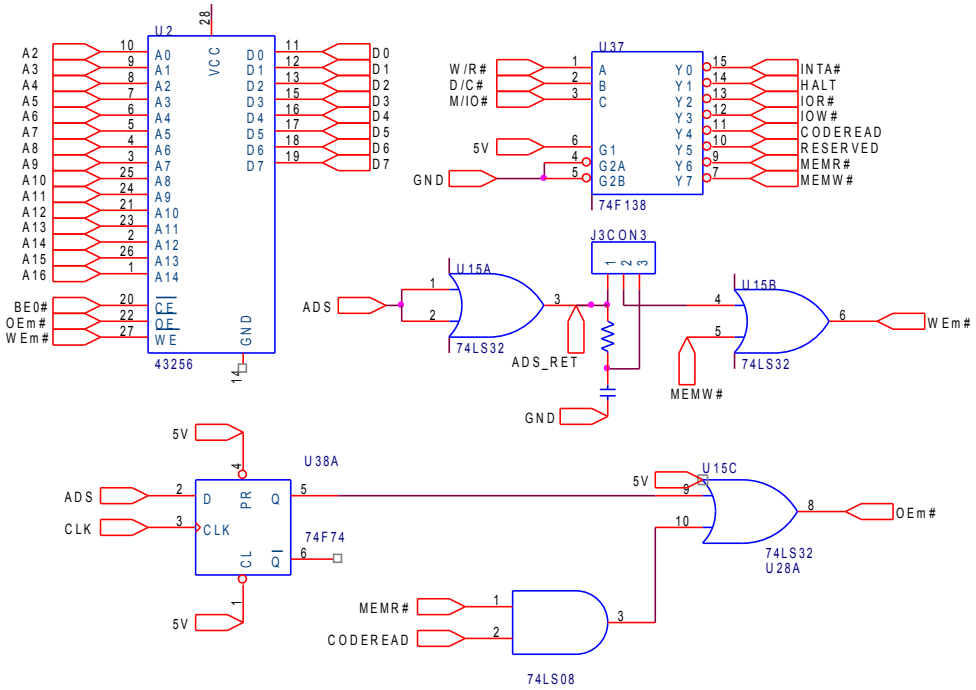

IMAGEN 13 Integrados relacionados con el acceso a la memoria.

En la imagen 13 se muestran los integrados relacionados con la interfaz entre el procesador y la memoria (en la imagen se muestra uno solo de los cuatro chips de 32kb de memoria estática que se incluyen en el circuito final, ya que son todos similares salvo por el hecho de que la entrada CE# esta conectada a BE0# en el primer chip, BE1# en el siguiente, etc y el bus de datos está conectado de forma tal que cada chip almacena uno de los cuatro bytes que conforman el bus de 32 bits).

La decodificación del tipo de ciclo de bus se realiza mediante el integrado 74F138 (U37), que se encuentra habilitado permanentemente (las entradas de habilitación del integrado están colocadas a valores fijos). Este chip tiene ocho salidas, de las cuales para la memoria se utilizan MEMR#, MEMW# y CODEREAD#. El integrado es de la familia lógica Fast ya que la velocidad de decodificación de familias mas lentas no es suficiente para cumplir con una velocidad de bus de 33 Mhz (si bien el 74LS138 podría por si solo decodificar la entrada en menos de 60 ns, el tiempo que consumen el resto de las compuertas usadas, mas los tiempos de hold y setup de la memoria hacen imposible el funcionamiento).

Junto con las señales de identificación del tipo de ciclo de bus, se debe prestar atención a la señal ADS#, que es la que indica cuando existe un ciclo válido, ya que bien puede el procesador poner (o dejar de un ciclo anterior) una señal MEMR# en el bus, sin que por ello esté pretendiendo leer de memoria, y si no se presta atención a ADS#, bien puede la memoria pasar sus salidas a baja impedancia, y asi interferir con el funcionamiento del procesador.

La señal ADS# sigue el comportamiento citado anteriormente. Durante el primer pulso de clock, pasa a nivel bajo (cero), y en el siguiente pasa a nivel alto (y sigue en ese valor si el ciclo durase mas de dos pulsos de reloj). Por lo tanto, en un ciclo válido podemos contar con un flanco positivo y uno negativo de ADS#, separados por un periodo de clock. En un sistema que funciona siempre con cero wait-states (como el implementado), estos flancos de ADS se puede aprovechar convenientemente. En sistemas que mezclan memorias y puertos de distintas velocidades (y que pueden necesitar o no insertar estados de espera según la memoria accedida), la circuitería es mas compleja, y no puede resolverse solamente usando el flanco de ADS#.

Un claro ejemplo de la simplicidad con que puede resolverse el circuito en sistemas sin estados de espera lo da la sección que genera la señal WEm# (Write Enable Mem). Esta sección simplemente efectúa una OR entre la salida decodificada MEMW# y una copia retardada de ADS#. De esta manera, cuando el ciclo válido comienza, ADS# pasa a cero, y si se trata de una escritura en memoria, MEMW# también pasa a cero, con lo cual WEm# pasa a cero (se activa la escritura en la memoria). Sin embargo, el procesador no pone datos válidos en su bus de datos hasta el segundo clock del ciclo, por lo cual no se puede generar el flanco de escritura hasta que esto suceda, y la memoria no almacena los datos presentes en su entrada hasta que la señal WEm# pase a valer uno. En esta situación, resulta conveniente usar una copia retardada temporalmente de ADS#, para generar el flanco de escritura (que es cuando WEm# pasa de cero a uno). El retardo (para el cual se usa una compuerta tipo LS, que ofrece un retardo típico de 10 a 20 nanosegundos) es necesario para asegurar que se cumpla el tsetup de la memoria estática (10 nanosegundos como mucho para una memoria del tipo utilizado), ya que el procesador pone los datos en el bus al mismo tiempo que cambia el valor de ADS#, pero estas señales tienen todas una dispersión en el retardo que consumen para cambiar de valor desde el flanco del reloj que va entre los 3 y los 11 nanosegundos. En caso de que el retardo provisto por la compuerta LS no sea suficiente, se provee un puente que conecta la señal a un filtro RC, que agrega un retardo extra de acuerdo a los valores de sus componentes (en el diseño práctico el uso del filtro RC no fue necesario).

El circuito que activa la salida de datos desde la memoria hacia el procesador es un poco mas complejo. Si bien la solución mas simple en este caso podría ser conectar la señal decodificada MEMR# directamente a la entrada OE# de la memoria, esto ignoraría la validación que debe hacerse mediante la señal ADS#. Por ello es que se agrega un flip flop tipo D (74F74), configurado como puede verse en la imagen 13, con las entradas de set y clear deshabilitadas, la señal ADS# en la entrada de datos, y la señal CLK en la entrada de reloj. La única función que cumple el FF es retardar un pulso de reloj el valor de ADS#. Esto se puede hacer ya que cuando llega el flanco positivo del reloj (CLK) el procesador tarda al menos 3 nanosegundos en cambiar la señal ADS#. Si se usa un FF lo suficientemente rápido (el 74F74 tiene un thold de 1 nanosegundo típicamente), se puede almacenar el valor de ADS# correspondiente al pulso de reloj anterior. Por otro lado, se combinan las señales CODEREAD# y MEMR# mediante una compuerta AND, ya que ambas son equivalentes, y las genera el procesador cuando quiere leer código o datos respectivamente de la memoria (este es un punto

fundamental, ya que en la hoja de datos no se aclara en ningún momento la funcionalidad de CODEREAD#, y el utilizado por Intel no es un comportamiento claramente estándar, ya que uno podría suponer que la señal MEMR# es la única necesaria para leer de la memoria, tanto datos como código). Esta señal combinada se ingresa a una OR junto con la copia retardada de ADS#, para generar finalmente la señal OEm# que habilita la memoria para la lectura. En resumen, la memoria se habilita para la lectura recién en el segundo flanco positivo de clock del ciclo (cuando el flip flop lee el valor de ADS# igual a cero que es propio del primer clock del ciclo). A partir de ahí, la señal OEm# pasa a cero, y se mantiene allí hasta el próximo flanco de clock, cuando el flip flop refleja el valor alto que hay en su entrada (propio del segundo clock del ciclo). La suma de retardos (de salida del procesador, de entrada a salida del FF, de la memoria para pasar a alta impedancia) hacen que thold del procesador se respete sin dificultades.

Por último, las memorias tienen conectadas en su señal de CE# las señales BEx#. Como no existen otras memorias, cada chip maneja un cuarto del bus de 32 bits, y no se necesita decodificación de direcciones. Si existiesen mas integrados en la memoria, seria necesario combinar lógicamente las señales BEx# con las de direcciones que correspondan.

DIAGRAMA DE TIEMPOS DE LA MEMORIA!!!!

Puertos de Salida

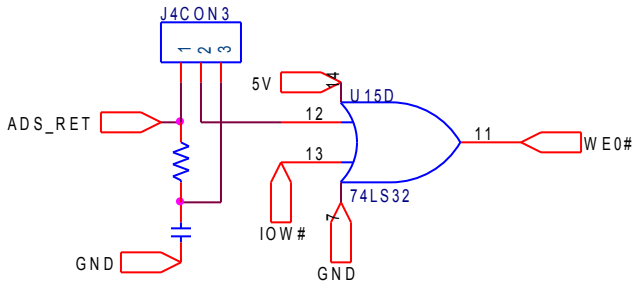

IMAGEN 14 Circuito para la escritura a los puertos.

En el diseño utilizado no se realizan lecturas de los puertos, solamente escrituras, por lo cual el circuito de la imagen 14, similar al de escritura en memoria salvo por el uso de la señal IOW# (que proviene del 74F138) en lugar de MEMW#, resuelve todas las necesidades del caso.

Los diferentes puertos de salida (tres en total, de 8 bits cada uno), no requieren decodificación de direcciones, ya que se conectan a cada uno de los bytes que componen los 32 bits del bus de datos (y de hecho, queda un byte sin utilizar). En caso de utilizarse mas de 4 puertos de 8 bits, seria necesario implementar una decodificación de direcciones.

La funcionalidad de cada uno de los puertos se detalla en los apartados donde son utilizados.

Circuito de interrupciones

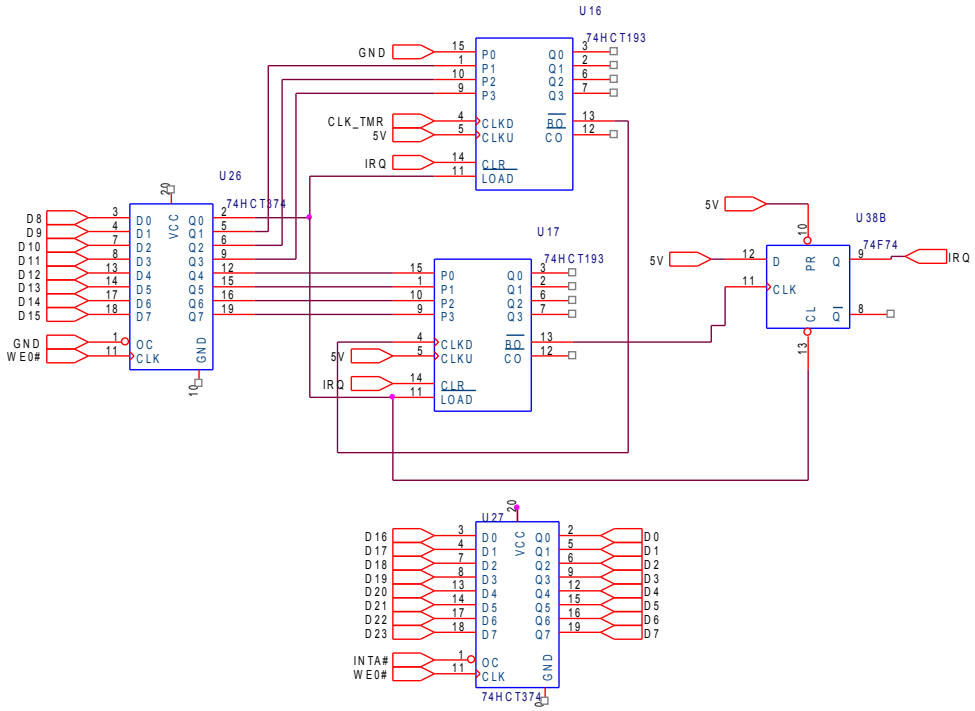

IMAGEN 15 Circuito para la generación de interrupciones y lectura del vector de interrupción.

En un diseño típico, la generación de interrupciones esta administrada por un integrado específicamente creado a tal efecto. Este integrado tiene la capacidad de identificar entre distintas fuentes de interrupcion, priorizarlas y transferir al procesador toda esta información en forma simple y directa. Como en el diseño del reproductor de MP3 hay solo una fuente de interrupción, no hay un sistema operativo formal, y por lo tanto casi todos los vectores de interrupcion están libres, se puede implementar una solución como la de la imagen 15.

En primer término tenemos el buffer 74HCT374 conectado al segundo byte del bus de 32 bits (D8-D15). Este es uno de los puertos de salida de la placa. La información allí grabada se usa para programar y controlar los dos contadores 74HCT193, que dividen una señal de reloj de 3,57Mhz (Un cristal muy común, por ser propio de la portadora de color en sistemas PAL-N). Como se observa en el circuito, los 7 bits mas significativos del puerto se conectan directamente al contador de 8 bits que forman los dos 193 en cadena (cada 193 tiene un contador de 4 bits). El bit menos significativo se usa para activar la señal de carga del contador y para resetear el flip flop que genera la señal de interrupción.

El funcionamiento completo es el que sigue: En primer término, se elige la división que se quiere hacer de la señal de 3,57 Mhz. Esta división debe ser un número par (ya que el bit menos significativo se usa para otra función). Luego, si se aplica la mínima división (por 2, a lo cual hay que sumarle 16 como se verá mas adelante) se obtiene una señal de 198,8 khz y si se aplica la máxima (por 256+16) se logra una frecuencia de interrupciones de 13160 por segundo. El ideal es generar la menor cantidad posible de interrupciones por segundo (ya que cada interrupción

trae aparejado un movimiento de stack del procesador, lo cual termina por hacer mas lenta la ejecución normal del programa), pero se debe tener en cuenta la velocidad de muestreo del audio (suponiendo que se envía una muestra por cada interrupción la frecuencia de interrupciones mínima debe ser igual a la frecuencia de muestreo).

El valor de división deseado se escribe en los bits D8 a D15 del bus de datos, mediante una escritura de puerto de salida (instrucción OUT en assembler x86).. El bit menos significativo del puerto de salida (que sería D8 en el bus de datos) se coloca en cero para que el contador active la carga paralela de datos y lea el valor programado en el puerto. Luego, se escribe nuevamente el mismo valor al puerto, pero poniendo el bit menos significativo en 1, con lo cual el contador comienza a contar de acuerdo a los pulsos de reloj entrantes, y el flip flop queda libre para cambiar su salida (ya que CL#=1).

Cada vez que el primer contador de 4 bits finaliza su cuenta descendiente, activa la señal BORROW# (BO#), lo cual disminuye en uno la cuenta del segundo contador, conformando asi un contador descendiente de 8 bits. Cuando la cuenta programada termina, la señal de BORROW# del contador mas significativo genera un flanco negativo. Esto no sirve para que el flip flop grabe su entrada en su salida (necesita un flanco positivo). Sin embargo, cuando el contador menos significativo reciba 16 pulsos de reloj adicionales, generará un nuevo pulso de clock para el contador mas significativo, el cual cambiará nuevamente su señal BORROW# a 1, y ahí se producirá el flanco de clock que permitirá la escritura del flip flop, y por lo tanto producirá la interrupción. Es por esto que la división mínima que puede realizarse es por 18 (2+16) y la máxima es por 272 (256+16)

Cuando se genera la interrupción, se activa la señal de CLR# de los contadores, con lo cual quedan detenidos, y la única forma de reactivarlos es escribiendo un cero en el bit menos significativo del puerto de salida usado para programar los contadores, con lo cual se limpia la señal IRQ, y se recomienza el proceso (este "reseteo" de la lógica de interrupción debería implementarse en el software dentro de la función que sea llamada por el procesador para administrar la interrupción).

Lo que resta resolver es el ciclo de bus Interrupt Acknowledge, generado por el procesador cuando acepta una interrupción enmascarable y desea conocer el vector de interrupción correspondiente (este número es un valor entre 0 y 255, que actúa como un índice dentro de una tabla). Este ciclo se puede ver en la imagen 9, y a primera vista parece complejo, ya que el bus de direcciones cambia de valor, el ciclo dura ocho ciclos de reloj o mas, y al final de todo esto el procesador espera en sus bits D0-D7 el vector de interrupción correspondiente.

Circuitalmente, se resuelve con un solo integrado, un 74HCT374, configurado como puerto de salida, con su entrada conectada a D16-D23 y su salida a D0-D7. La señal de escritura se conecta a la señal de grabación de puertos (WE0#), y la de lectura se conecta a la salida decodificada INTA# del 74F138. Para generar el vector de interrupción, lo único que debe hacerse es grabar un valor en el buffer (los valores reservados para el procesador son de 0 a 20h, el resto son utilizables), y cuando el procesador genere la señal INTA#, el buffer pondrá en el bus el vector

durante todo el ciclo (si bien estrictamente el procesador lo necesita solo durante el último pulso de reloj).

Aquí se ha realizado un compromiso de tipo "experimental". Como podrá observarse, no se ha incorporado la señal ADS# para la lectura del vector de interrupción, aunque estrictamente debería haberse usado un circuito similar al de la lectura de memoria, cambiando MEMR# por INTA#. En la práctica este compromiso no presentó problemas (en cuyo caso hubiese sido necesario cambiar el diseño), por lo cual no resultó necesaria ninguna modificación. Sin embargo la técnica usada no tiene garantías, y en cada caso debería evaluarse si es conveniente aplicarla o usar un método mas seguro (como el que se usa para leer las memorias).

Conversor Digital-Analógico

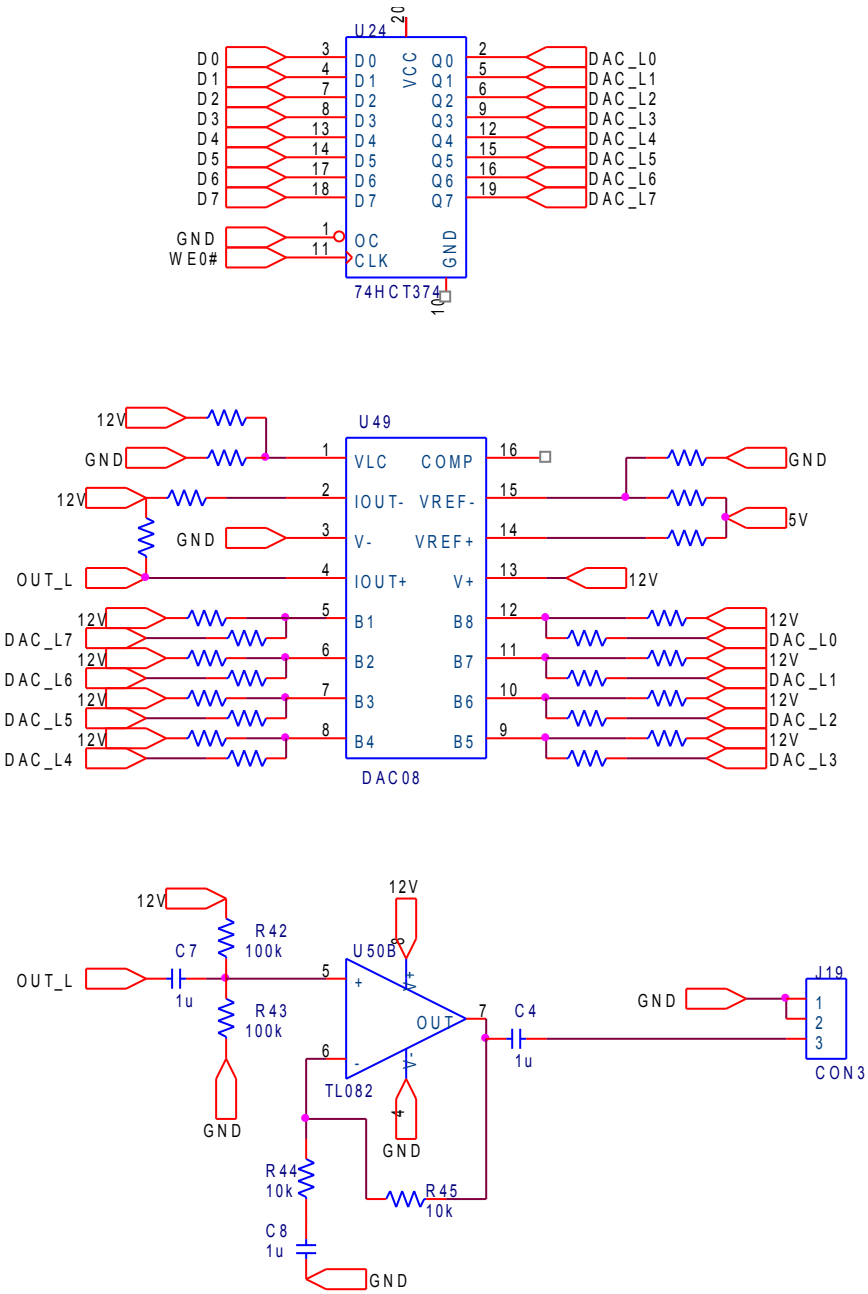

IMAGEN 16 Circuito de conversión digital-analógica, mas amplificador de salida.

El circuito de salida se compone de un buffer 74HCT374 configurado como puerto de salida, donde se escriben las muestras digitales. Esas muestras ingresan al DAC08, que convierte la información digital en analógica, y luego esta se lleva a un TL082 que aísla y amplifica la señal.

El puerto 74HCT374 está conectado a los bits D0-D8 del bus de datos del procesador. Estos bits se trasladan al DAC, el cual posee un gran número de resistencias, debido a que el integrado puede conectarse a distintas familias lógicas, y el ajuste de los niveles de tensión eléctricos se realiza mediante estas resistencias. **La configuración del DAC** no se detallará, ya que es estándar según se indica en

la hoja de datos del mismo y en una nota de aplicación de Analog Devices[7], donde se indica como configurar el dispositivo para operar con una fuente unipolar.

Finalmente, el TL082 actúa como aislador del circuito interno hacia la conexión externa, esta configurado con realimentación solamente en alterna (ya que opera con una sola fuente, su salida se configura a un valor de continua de 6V), la cual puede usarse para amplificar o no la señal. Este integrado se incluye solo por seguridad, pero con el cuidado suficiente, puede tomarse la señal directamente del DAC.

Interfaz con la PC

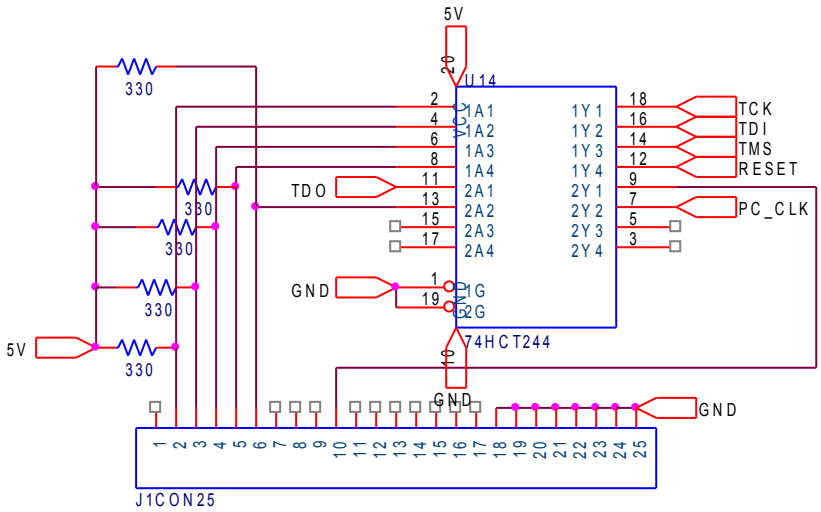

IMAGEN 17 Circuito de interfaz con la PC.

Por no poseer memoria ROM, ni otro tipo de medio de almacenamiento permanente, la carga de datos hacia el procesador depende de un dispositivo externo, en este caso una PC corriendo un software diseñado especialmente. La interfaz hacia la PC se realiza mediante el puerto paralelo de la misma, el cual a su vez esta conectado a un buffer que maneja varias señales dentro de la placa. Estas señales son las del controlador TAP (TCK, TDO, TDI y TMS), mas la de RESET del procesador y una conexión que permite ingresar una señal de clock externa a la placa.

El manejo de estas señales no requiere mayor explicación desde el punto de vista del hardware, aunque si se hará un detalle extensivo en la sección de software, ya que esta conexión es clave para el funcionamiento del reproductor, por no existir otro canal para conocer lo que ocurre en la placa, ni para actuar sobre ella.

7 Analog Devices AB-6, obtenible en www.analog.com

Generadores de reloj

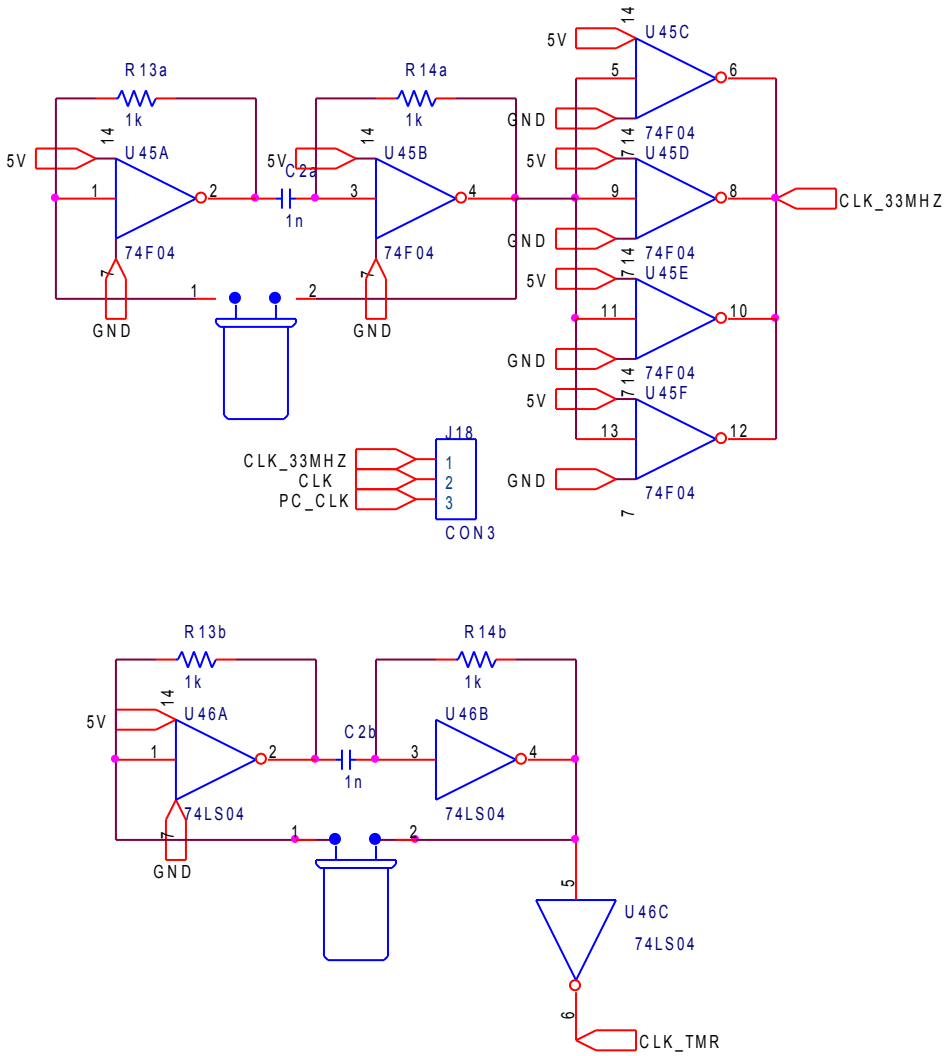

IMAGEN 18 Osciladores a cristal, generadores de las señales de reloj.

En circuitos de mayor complejidad, la generación de la señal de reloj no es una tarea menor. El ruteo de la señal, y la calidad que esta debe tener en todas sus conexiones, mas la variedad de frecuencias que se debe poder generar, hacen que en un sistema complejo, como un motherboard de PC, se deban usar integrados especialmente diseñados para la función. En un diseño como el del reproductor de MP3 (que funciona a una frecuencia fija de bus de 33 Mhz), la cantidad de integrados conectados a esta señal de reloj es mínima, por lo cual un oscilador construido con una compuerta 74F04 cumple perfectamente la función, aunque la señal generada vista en el osciloscopio no es todo lo buena que podría ser. El otro circuito generador de reloj funciona a una frecuencia mucho menor (3,57 Mhz) y se puede armar con una compuerta similar, pero de tecnología LS.

Capítulo

Software

Luego de observar en detalle el hardware, no puede encontrarse allí nada que indique claramente que se trata de un reproductor de MP3. La razón de esto es que el circuito utilizado puede ser empleado en una gran variedad de aplicaciones, y lo que realmente le da el poder de reproducir este formato de audio es el software que ejecuta.

Los elementos que componen el software del reproductor de MP3 son varios. Por un lado esta la aplicación de mantenimiento, que permite utilizar la interfaz JTAG del procesador para programar la memoria y controlar la ejecución de programas en la placa. Por otro lado, esta el software en si que ejecuta el procesador, una aplicación sin sistema operativo, que se ejecuta en modo protegido de 32 bits, y que esta basada en la librería de código abierto MAD[8], la cual es gratuita y capaz de reproducir archivos de audio comprimidos con cualquiera de los layers de la especificación MPEG I.

De igual manera que se hizo con el software, este capítulo comenzará detallando los aspectos mas importantes del procesador i486!, desde el punto de vista del software.

¡486!

Modo protegido

Todos los procesadores de la línea Intel x86 son compatibles "hacia atrás", lo cual implica que pueden ejecutar sin cambios código compilado para versiones anteriores de procesadores x86. Esto a su vez conlleva a que todos los procesadores x86 arrancan su funcionamiento (al momento de ser encendidos, o luego de ser reseteados) en lo que se conoce como "modo real", en el cual se

8 Copyright Robert Leslie (2000), la página web se encuentra en http://www.mars.org/home/rob/proj/mpeg/

comportan como un 8086, aunque con una velocidad de funcionamiento ciertamente muy superior al padre de todos los procesadores x86.

Este modo real tiene muchas restricciones, propias de una arquitectura que ya tiene mas de dos décadas de antigüedad. Por empezar, es un modo inherentemente de 16 bits (aunque para ser precisos, los procesadores x86 desde el 386 en adelante pueden ejecutar instrucciones de 32 bits aun estando en modo real), lo cual limita el espacio de memoria accesible (1 Megabyte de memoria, gracias al uso de registros de segmento y desplazamiento). Tampoco se posee protección entre aplicaciones, ni herramientas para paginación, ni ninguna de las ventajas que ofrece el modo protegido.

Sin embargo, a pesar de tantas restricciones, el modo real debe ser usado al menos una vez en este proyecto, ya que es el modo "por defecto" cuando se inicializa el procesador. Luego, el único segmento de software escrito para modo real que tendrá el proyecto es el código que se encarga de "saltar" a modo protegido.

Nuevamente es oportuno aclarar que la mejor forma de comprender acabadamente el funcionamiento del procesador es leyendo la documentación del fabricante, la cual es extensa y muy explicativa. En este informe solo se detallarán las partes específicas que sean necesarias para la comprensión del funcionamiento del sistema. Un ejemplo de esto es todo el sistema de paginación, el cual no es requerido en el reproductor de MP3, pero que sin embargo puede ser muy útil para cualquier diseñador, y es muy importante entenderlo.

Una de las principales características del modo protegido es una tabla conocida como GDT (Global Descriptor Table). En el modo real el procesador usa para acceder a la memoria una serie de registros de 16 bits (llamados registros de segmento), que combinados con otros registros generales (usados para el desplazamiento), permiten acceder de forma rápida a un total de un megabyte de espacio de memoria, con una tecnología de 16 bits. Los registros de segmento se denominan CS (Code Segment), DS (Data Segment), SS (Stack Segment), y los registros adicionales para datos ES, FS y GS. Luego, si el procesador en modo real debe acceder a un dato en el stack, combina el SS con SP (registro puntero de stack) desplazando el primero cuatro bits a la izquierda y sumándole el segundo, para conformar la dirección de 20 bits final.

Hasta aquí, el comportamiento del 486 es igual al de todos los procesadores que funcionan en modo real. Sin embargo, en modo protegido los registros de segmento dejan de tener un significado como el que tienen en modo real, y pasan a ser índices en una tabla. Dicha tabla es precisamente la GDT (o la LDT, que es una variante de la GDT localizada para cada aplicación, de la cual no nos ocuparemos por no ser necesaria para el desarrollo del reproductor que tiene una sola aplicación ejecutándose).

El procesador posee un registro de modo protegido llamado GDTR de 48 bits (6 bytes). Este registro, junto con LDTR (que se usa para la LDT), son los únicos que almacenan una dirección física para uso directo del procesador (el resto de las direcciones son lógicas, y se transforman en direcciones físicas luego de ser

afectadas por la información de la GDT, y del sistema de paginación). Dicha información se carga desde una zona de memoria de 6 bytes, donde el sistema operativo (o en este caso el programa que inicializa el reproductor) debe colocar un double word (o dword igual a 4 bytes) seguido de un word (2 bytes). El dword especifica la dirección física en memoria donde esta la base (el primer byte) de la GDT. El word especifica el tamaño de la GDT menos uno (que se expresa en bytes), es decir, si la tabla ocupa 24 bytes, el valor que se debe almacenar allí es 23. **El registro GDTR se carga** con la instrucción de assembler LGDT, y se puede volcar a una dirección de memoria usando la instrucción SGDT.

La estructura de una entrada de la GDT (llamadas descriptores de segmento) se muestra a continuación.

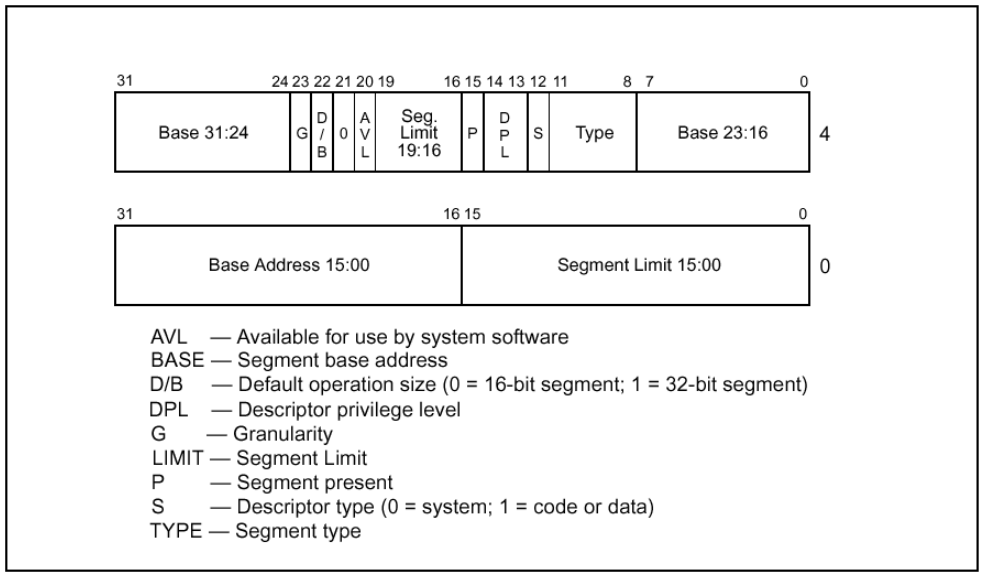

IMAGEN 19 Estructura de cada una de las entradas de la GDT, llamadas descriptores de segmento

La primer entrada, el índice cero, de la GDT debe llenarse con ceros (es decir, los primeros ocho bytes). Esta entrada se usa como "null", y se asigna a los registros de segmento no utilizados. Si en algún momento el programa intenta usar alguno de estos registros cargados con el valor null, se provoca una excepción. El resto de las entradas (hasta un máximo de 8192) se pueden configurar libremente por parte del usuario, pero se recomienda que haya al menos un segmento asignado para datos y otro para código.

La explicación de cada una de las banderas que hay en cada descriptor de segmento no se explicará aquí. Para mas detalles, se puede consultar la documentación de Intel (Intel Architecture Software Developer's Manual, Vol 3, Cap 3: Protected Mode Memory Management). De forma simplificada, puede decirse que cada entrada de la GDT tiene un valor base de 32 bits (la dirección donde comienza el segmento), mas un valor de límite de segmento de 16 bits (el cual puede especificar valores con una granularidad de un byte o de 4KB, alcanzándose en el último caso el límite de 4GB), mas banderas que indican si se trata de un segmento de solo escritura, de lectura, o bien de datos o código, y

otras que fijan el nivel de privilegio que tiene el segmento, y si este se encuentra disponible o no (en el caso de sistemas con paginación), mientras que otra bandera indica si el segmento es de 32 o 16 bits (ya que el modo protegido no implica si o si utilización de código de 32 bits, como el modo real no implica tampoco utilización de código de 16 bits).

Si bien para el programador los registros de segmentos siguen siendo de 16 bits tanto en modo real como en modo protegido, su funcionalidad como ya se explicó es diferente. Aún mas, en modo protegido los registros de segmentos tienen una parte oculta (inaccesible para cualquier programa), que no es otra cosa que un buffer de la GDT. Entonces, cada vez que se cambia un registro de segmento, el procesador copia la información correspondiente desde la GDT hacia la parte oculta del registro, de forma tal de no tener que consultar la GDT desde la memoria cada vez que usan estos registros (que en el caso de CS por ejemplo es cada vez que se carga una nueva instrucción). Cuando se desea cargar un nuevo descriptor de segmento, se debe buscar el índice que tiene en la GDT, y conformar el selector de segmento de la siguiente manera:

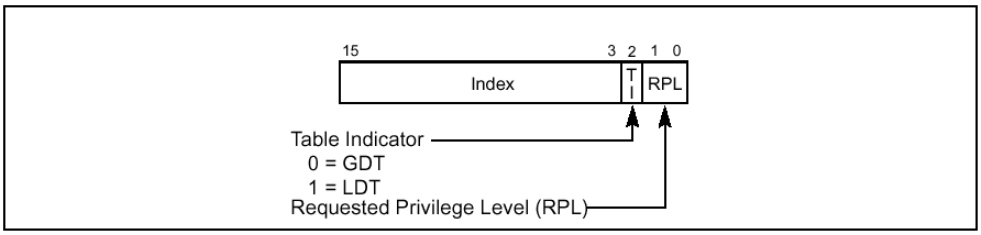

IMAGEN 20 Selector de segmento

En la imagen 20, se ve que el índice de la tabla se desplaza 3 bits hacia la izquierda, en los dos bits menos significativos se escribe el nivel de privilegio requerido, y en el tercer bit se indica si el selector pertenece a la GDT o a la LDT. Luego, para acceder a la segunda entrada de la GDT (índice igual a 1), con un RPL (nivel de privilegio requerido) de cero, el valor del selector debe ser de 8.

Dada toda la información precedente, resulta necesario para pasar de modo real a modo protegido configurar una GDT con al menos dos entradas (tres en rigor, ya que la primer entrada debe dejarse en blanco), una para un segmento de datos y otra para un segmento de código (en el reproductor de MP3, ambos segmentos se configuran al máximo de su tamaño, 4 GB). Se debe luego configurar el registro CS, el DS y el SS para que apunten a estas entradas de la GDT (CS debe apuntar al segmento de código, mientras que DS y SS al de datos), y recién ahí generar el salto de modo real a protegido. Otro tipo de configuraciones pueden resultar necesarias (como configurar la tabla de interrupciones para modo protegido), pero no son indispensables. El código especifico que realiza todas estas tareas se detalla mas adelante.

En un párrafo anterior se menciona que GDTR y LDTR son los únicos registros que almacenan información de direcciones físicas en modo protegido. Esto es así por la naturaleza misma de este modo, que aísla mediante la GDT al programa de la ubicación real en memoria donde se encuentra. Luego, el único momento en que

se necesita inevitablemente una dirección física en forma directa es cuando se debe cargar la tabla GDT. Esta traslación entre direcciones físicas y lógicas se realiza en dos pasos: la conversión de la dirección lógica mediante la GDT (o LDT), y la paginación del espacio lineal de memoria.

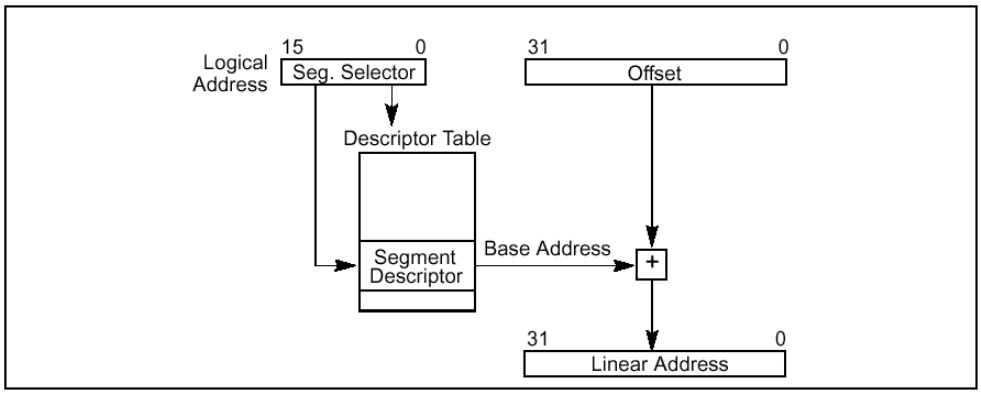

IMAGEN 21 Conversión entre direcciones lógicas y lineales

En la imagen precedente se observa que una dirección lógica compuesta de un segmento y un desplazamiento (segment selector y offset), se combinan mediante la GDT (o la LDT) en una dirección lineal. Esto se hace tomando el valor de base que aparece en el descriptor de segmento elegido por el selector de segmento, y sumándole el offset (el cual puede ser de 32, 16 u 8 bits), previa verificación de que el offset no supere el límite indicado en el descriptor. Esta operación da como resultado una dirección lineal. Si no se está usando paginación, esta dirección lineal es directamente la dirección física en memoria.

El sistema de paginación hace uso de otra tabla (y una serie de buffers para acelerar el proceso de conversión), que le permite trasladar la dirección lineal en una dirección física, a su vez controlando que dicha dirección lineal tenga una correspondencia física, ya que puede darse el caso de que la posición de memoria a la que quiere accederse no este cargada (puede estar almacenada en un disco rígido por ejemplo), momento en el cual se genera una excepción no fatal, que da la oportunidad al programa de cargar la información necesaria en memoria, y luego continuar la ejecución de forma transparente para el usuario. Un ejemplo puede clarificar esto: Supongamos que el sistema tiene solo 64kb de memoria, mapeados físicamente desde 0h hasta FFFFh, pero tenemos un disco rígido de 128kb (un disco ridículamente pequeño). Podemos configurar el sistema de paginación de forma tal de mapear dos bloques de direcciones lineales (0h hasta FFFFh y 10000h hasta 1FFFFh) en la misma dirección física (0h hasta FFFFh). Luego, tenemos dos páginas, que se cargan alternativamente en memoria, dependiendo de cual es la región lineal a la que se accede. Cada vez que aparezca una dirección lineal que no esta cargada en memoria, el sistema de paginación generará una excepción gracias a la cual nuestro programa podrá cargar la información del disco rígido hacia la memoria.

Otro tema que queda por aclarar es el de los niveles de privilegio. Existen cuatro niveles de privilegio en modo protegido (0, 1, 2 y 3). El nivel mas privilegiado es el

cero, y cada descriptor de segmento tiene un nivel de privilegio asociado. Estos niveles de privilegio permiten controlar el funcionamiento del procesador al máximo. En un sistema típico, el sistema operativo corre en el nivel cero, mientras que las aplicaciones corren en niveles superiores. Luego, solo el sistema operativo tiene acceso a ciertas instrucciones de assembler (como la que carga la GDT, o el acceso a los puertos) por estar en el nivel cero, y si alguna aplicación desea acceder a alguna de estas funciones, debe hacerlo mediante una "puerta de llamada", la cual es administrada también por el sistema operativo, ya que si se intenta un salto directo de un segmento con nivel de privilegio menor hacia uno mayor (por ejemplo, del nivel 3 al 0), se genera una excepción por parte del procesador.

En el diseño del reproductor de MP3 no hay sistema operativo, y los niveles de privilegio no tienen aplicación, ya que desde que se inicia en modo real, y durante toda la ejecución, la aplicación corre en el nivel mas privilegiado.

Comprendidos ahora los sistemas de segmentación y de niveles de privilegio, no es difícil entender las ventajas que ofrece el modo protegido. En primer término, permite aislar los diferentes programas entre si, e independizarlos de la dirección física real en la que se encuentran (ya que los programas solo "saben" de offsets, no conocen el segmento en el que están, ni que dirección base tiene), además de controlar que no exista interferencia indeseada entre programas (por ejemplo, si un programa contiene un error, e intenta escribir información fuera de los límites del segmento). Esto junto con el control de privilegios, permite armar sistemas muy robustos, cuya fortaleza depende exclusivamente del hardware, y no de software. Nuevamente, muy pocas de estas posibilidades se usan en el reproductor de MP3, pero es importante conocer su existencia y funcionamiento.

Inicialización del procesador

Como ya se indicó, el procesador inicia su funcionamiento cuando es encendido, o luego de ser reseteado, en modo real, y con un estado predeterminado en todos sus registros.

La siguiente tabla resume el estado inicial del procesador:

Register	P6 Family Processors	Pentium® Processor	Intel486™ Processor
EFLAGS[1]	00000002H	00000002H	00000002H
EIP	0000FFF0H	0000FFF0H	0000FFF0H
CR0	60000010H[2]	60000010H[2]	60000010H[2]
CR2, CR3, CR4	00000000H	00000000H	00000000H
MXCSR	Pentium® III processor only- Pwr up or Reset: 1F80H FINIT/FNINIT: Unchanged	NA	NA
CS	Selector = F000H Base = FFFF0000H Limit = FFFFH AR = Present, R/W, Accessed	Selector = F000H Base = FFFF0000H Limit = FFFFH AR = Present, R/W, Accessed	Selector = F000H Base = FFFF0000H Limit = FFFFH AR = Present, R/W, Accessed
SS, DS, ES, FS, GS	Selector = 0000H Base = 00000000H Limit = FFFFH AR = Present, R/W, Accessed	Selector = 0000H Base = 00000000H Limit = FFFFH AR = Present, R/W, Accessed	Selector = 0000H Base = 00000000H Limit = FFFFH AR = Present, R/W, Accessed
EDX	000006xxH	000005xxH	000004xxH
EAX	0[3]	0[3]	0[3]
EBX, ECX, ESI, EDI, EBP, ESP	00000000H	00000000H	00000000H
FPU Data Operand and Inst. Pointers[4] GDTR,IDTR	Pwr up or Reset: 00000000H FINIT/FNINIT: 00000000H Base = 00000000H Limit = FFFFH AR = Present, R/W	Pwr up or Reset: 00000000H FINIT/FNINIT: 00000000H Base = 00000000H Limit = FFFFH AR = Present, R/W	Pwr up or Reset: 00000000H FINIT/FNINIT: 00000000H Base = 00000000H Limit = FFFFH AR = Present, R/W
LDTR, Task Register	Selector = 0000H Base = 00000000H Limit = FFFFH AR = Present, R/W	Selector = 0000H Base = 00000000H Limit = FFFFH AR = Present, R/W	Selector = 0000H Base = 00000000H Limit = FFFFH AR = Present, R/W
DR0, DR1, DR2, DR3	00000000H	00000000H	00000000H
DR6	FFFF0FF0H	FFFF0FF0H	FFFF1FF0H
DR7	00000400H	00000400H	00000000H

IMAGEN 22 Estado inicial de los procesadores Intel.

El procesador lee su primer instrucción de la dirección física/lineal FFFFFFF0h, es decir 16 bytes por debajo del límite máximo. Esta dirección esta fuera del alcance del modo real (que solo puede acceder al primer megabyte de memoria), pero los valores con los que se inicializan los registros permiten acceder a esta posición de memoria.

El registro CS se carga con el valor F000h, y su parte oculta (la que en modo protegido actúa como "buffer" de la GDT, almacenando el último descriptor de segmento utilizado) se carga con una valor de base "ficticio" de FFFF0000h. El puntero de instrucciones (EIP) se carga con el valor FFF0h. Luego, el procesador (a pesar de estar en modo real), hace la suma entre la base del descriptor almacenado y el registro EIP, para leer su primer instrucción (resultando la dirección FFFFFFF0h). Nótese que si se ejecuta la regla normal de modo real (desplazamiento*4+offset), el resultado es FFFF0h, lo cual también representa 16 bytes por debajo del límite máximo del modo real. En cuanto el código ejecute un salto de segmento (que modifique el valor de CS, e invalide la información que se

halla en la parte oculta del registro, la cual no puede ser restaurada ya que no existe una GDT ni algo similar en modo real), el procesador quedará confinado al primer megabyte de la memoria mientras permanezca en modo real. Es importante entonces no ejecutar "saltos largos" (far jumps, o saltos de un segmento a otro), si el código de inicialización está en la parte mas alta de la memoria (se deben usar saltos relativos, lo cual limita el campo a los 64 kb superiores de los 4GB totales de memoria física). Si el código de inicialización puede ser colocado dentro del primer megabyte físico de memoria, no hay inconvenientes en realizar un far jump. Estos problemas son mas complejos de resolver en sistemas que deben aceptar cantidades variadas de memoria RAM (como un motherboard). En el diseño del reproductor hay 128 kb de memoria, por lo cual la complejidad se reduce.

Otro registro de importancia es CR0, que almacena una serie de banderas que controlan el funcionamiento del procesador. Su significado se puede observar en el siguiente cuadro:

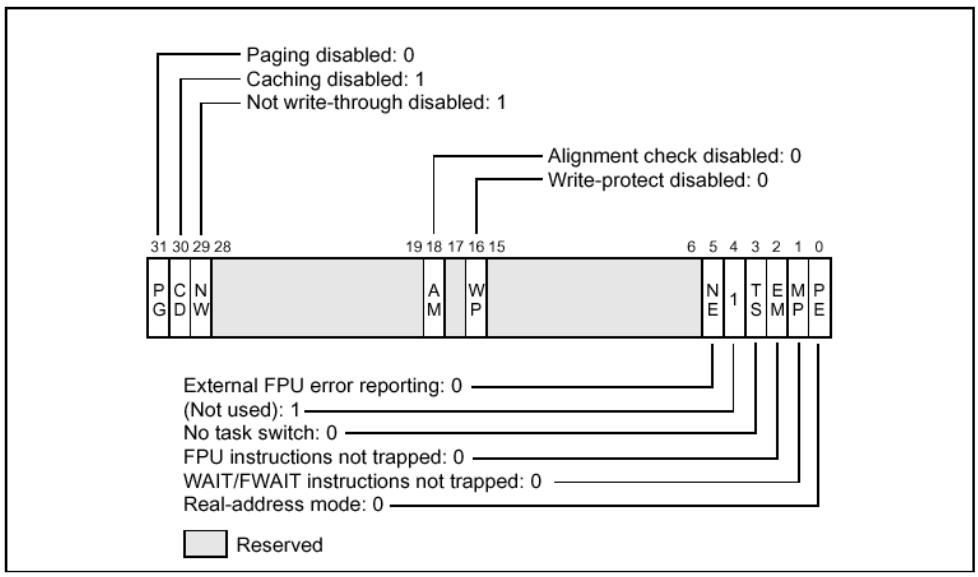

IMAGEN 23 Contenido de CR0 al iniciarse el procesador.

Como puede observarse, el estado de CR0 pone al procesador en modo real, con el cache deshabilitado, sin paginación y con presencia del coprocesador matemático.

La tabla de interrupciones de modo real se encuentra por defecto en las posiciones mas bajas de memoria (desde 0h en adelante), como es típico del modo real, aunque esto puede cambiarse, ya sea para el modo real como el protegido.

Existen más registros que los aquí detallados cuya inicialización puede ser importante en algún diseño particular. En especial, existe un protocolo para sistemas multiprocesador, hay registros MMX, controladores de cache y de interrupciones dentro del chip (en procesadores más avanzados que el 486).

Interrupciones y excepciones

Las interrupciones y las excepciones son transferencias forzadas de la ejecución desde el programa hacia una función o procedimiento conocido como "manejador" o "administrador" (handler).

Las interrupciones pueden ocurrir por un requerimiento del hardware externo (si este activa las señales IRQ o NMI), o por un requerimiento del software (que puede generar una interrupción mediante la instrucción INT). Las excepciones las genera el procesador frente a una amplia gama de situaciones, como ser un error (división por cero por ejemplo), una necesidad relacionada al sistema de paginación, o por el sistema de "debugging", que puede generar excepciones luego de cada instrucción o en puntos específicos, para poder investigar el estado del programa en cada paso.

El manejo de excepciones e interrupciones es diferente en el modo real y en el protegido. En ambos casos se utiliza una tabla de vectores de interrupción, que en el modo protegido se denomina Tabla de Descriptores de Interrupción (IDT). Si bien en ambos casos se usan vectores de interrupción (que indican donde se encuentra alojado el handler que corresponde a cada interrupción), en el modo protegido se agregan otros datos en cada entrada de la IDT, que especifican diversas opciones propias del modo protegido. En ambos modos el total de vectores de interrupción es de 256, de los cuales los primeros 32 estan reservados para el procesador (se usan para los distintos tipos de excepciones que pueden generarse).

Las excepciones son muy prácticas para entender el funcionamiento de un programa, ya que estas se generan para prácticamente cualquier error en que el programa incurra. Además, se pueden generar excepciones luego de cada instrucción, o en puntos específicos del programa, las cuales se pueden usar para evaluar el estado del procesador en cada paso. Por motivos de simplicidad no se han implementado en el software del reproductor administradores de excepciones, aunque en un proyecto con una base de software mas grande, y mas difícil de controlar y analizar, puede resultar muy beneficioso implementar uno o varios de estos controladores.

Lo que si se ha implementado es un handler de interrupción, para administrar las interrupciones por hardware que genera el divisor programable externo. Para hacer esto, hay que construir una IDT, en la cual se dejan todas las entradas en blanco salvo por una (la que administrará la interrupción deseada). Cabe destacar que si por algún motivo se genera otra interrupción con un vector distinto al definido por el diseño previo, o una excepción, el procesador intentará interpretar la información de la tabla, que se ha dejado en blanco, y lo mas probable es que el procesador entre el modo "halt" al producirse tres interrupciones en cadena. Este es un riesgo que se asume, y que no representó dificultades en la práctica.

Las entradas de la IDT se dividen en tres grupos: Task Gate, Interrupt Gate y Trap Gate. Las tres tienen un tamaño de 8 bytes cada entrada. Las que se usan para administrar las excepciones e interrupciones son generalmente las Interrupt Gate cuya estructura se detalla en la siguiente imagen.

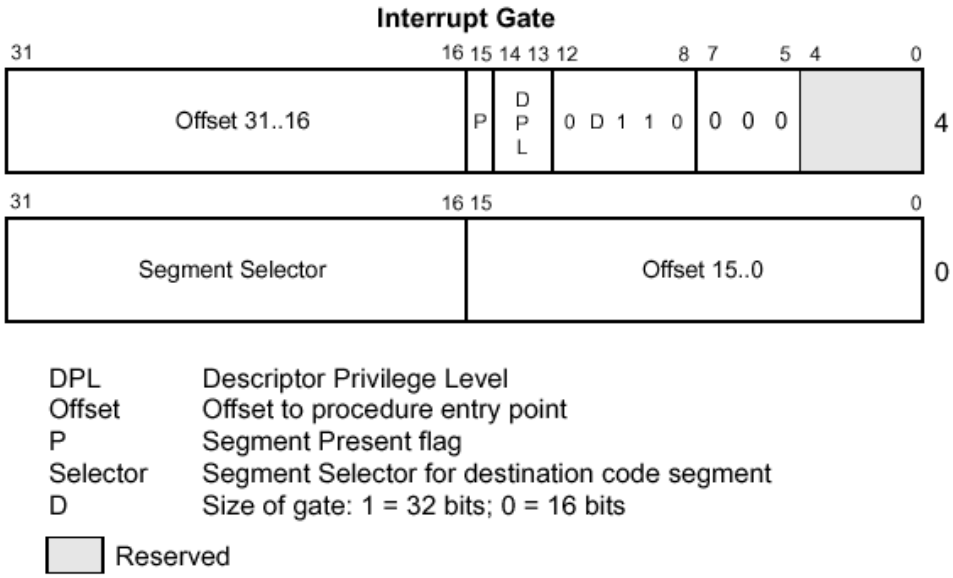

IMAGEN 24 Descriptor del tipo Interrupt Gate.

Como puede observarse, en el descriptor se incluye un selector de segmento (que es un índice en la GDT), mas un offset, y tres banderas (DPL, P y D) que se usan para controlar el nivel de privilegios, y su presencia física (en caso de usarse paginación), y el tipo de direccionamiento a usarse (32 o 16 bits).

Cache

El procesador 486 posee un cache "on chip" de 16 kb de memoria estática de alta velocidad. Dada la poca cantidad de memoria que tiene el reproductor (128 kb), y su gran velocidad (que permite el acceso a la misma sin estados de espera), la memoria cache se configura como memoria estática (es decir, no actúa como cache, sino como una parte más de la memoria, con la única diferencia de que no puede ser accedida externamente), ya que lo que se necesita en este caso es aumentar la cantidad, y no tanto la velocidad.

La configuración del cache en un procesador 486 DX4 es levemente distinta al resto de los 486, ya que este posee el doble de memoria cache. Los 16 KB se dividen en 256 sets, de cuatro líneas cada uno, conteniendo cada línea 16 bytes. A su vez, cada línea tiene asociada un tag de 20 bits, y dos banderas, una que sirve para arbitrar el recambio de las entradas del cache, y otra que indica la validez de los datos. El tag mencionado no es otra cosa que los 20 bits mas significativos de la dirección cuya información está almacenada en el cache. Es importante notar que estos 20 bits no pueden identificar unívocamente una dirección de memoria, ya que estas se identifican con 32 bits. Si se tiene en cuenta que cada línea de cache tiene 16 bytes (lo cual son cuatro bits de direcciones), tenemos que cada tag representa 256 posiciones de memoria que son indistinguibles entre si (32 bits en total, menos los 24 recién mencionados, dan un total de 8 bits, o 256 posiciones). Para completar la información faltante, se usa el número de set asignado. Luego, como hay 256 sets, se completa la información necesaria para determinar a que dirección corresponde cada entrada de cache.

Esta forma de direccionamiento conlleva a que cualquier dirección de memoria no puede ir a parar a cualquier punto del cache. Por ejemplo, una dirección como FFFFh, no puede ubicarse en el tercer set del cache, sino que debe colocarse en el último set (FFh=255). Una dirección como AABB33h, se coloca en el set 33h del cache. Resulta inmediato entonces que si el cache debe almacenar cinco direcciones como las siguientes: AAA55h, BBB55h, CCC55h, DDD55h y EEE55h, no puede hacerlo, ya que cada set tiene cuatro líneas, y estas cinco direcciones irían a parar todas al mismo set (el mecanismo de cache descarta la línea de mayor antigüedad para dar lugar a la mas nueva).

En el diseño utilizado se configuran las líneas de la siguiente manera: como la memoria física es de 128 kb, esta se extiende entre las direcciones físicas 0h hasta 1FFFFh. Por lo tanto, en los 256 sets los tags de las cuatro líneas se graban con el valor 20h para la primera línea, 21h para la segunda, 22h para la tercera y 23h para la cuarta. Entonces, por ejemplo la primer línea del set cero representa las direcciones 20000h hasta 2000Fh y la cuarta línea del set 256 representa la dirección 23FF0h hasta la dirección 23FFFh.

Para configurar el cache como memoria estática, se debe hacer uso de tres registros que son específicos del 486 (no están en generaciones previas o posteriores de procesadores x86). Estos registros son TR3, TR4 y TR5 y se accede a ellos mediante las instrucciones de assembler (exclusivas del 486) mov TREG,reg y mov reg,TREG. Su funcionalidad esta pensada para comprobar la integridad del cache, y como función accesoria para configurar el mismo como memoria estática (el funcionamiento completo de estos registros se explica en el apartado 2 del apéndice B del Embedded Intel 486 Processor Family Developer's Manual).

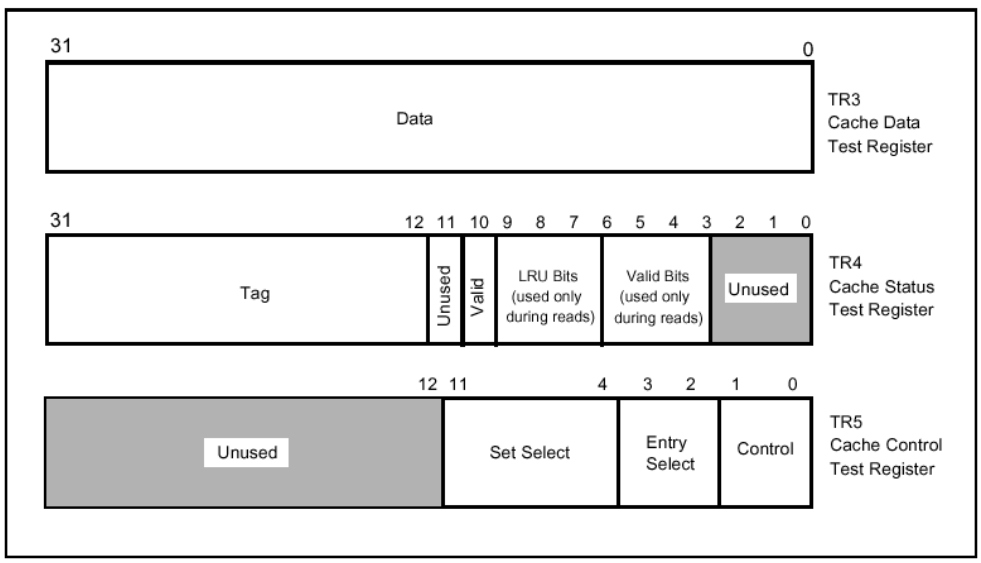

IMAGEN 25 Descripción de los tres registros para pruebas en el cache interno.

El procedimiento que se sigue para configurar el cache como memoria estática es el siguiente: lo primero es configurar los bits que controlan el cache (CD y NW en el registro CR0), de forma tal de deshabilitarlo. Esto le dice al procesador que no puede sacar la información que se encuentra en el cache hacia la memoria externa,

ni actualizar el cache interno con información del exterior. Lo siguiente, es recorrer los índices del cache, y marcar cada entrada como válida, asignándole a la vez una dirección de memoria que representan mediante el tag. El procesador, a pesar de tener el uso del cache deshabilitado, recorrerá la información del mismo, y si encuentra tags válidos, utilizará la información allí almacenada (aunque no la intentará actualizar desde el exterior, o reportar los cambios a la memoria externa, y esto es justamente lo que queremos lograr).

Antes de explicar como se efectúa el procedimiento para programar las entradas del cache, conviene conocer cual es la funcionalidad de cada uno de los tres registros.

TR3 es el Cache Data Test Register, y se usa para acceder al buffer intermedio de lectura del cache y al buffer intermedio de escritura del cache. Este registro es de 4 bytes (32 bits), mientras que una línea de cache (y también el buffer intermedio) tiene 16 bytes, por lo cual hay que escribir (o leer) cuatro veces a este registro para ingresar una línea completa. El control de que grupo de cuatro bytes dentro de una línea es accedido se realiza mediante dos bits en TR5.

TR4 es el Cache Status Test Register, en el cual se escribe (o lee) la información correspondiente al tag de cada línea, junto con el bit de validez. En el modo de lectura se puede obtener de este registro el valor del LRU (Last Recently Used), un índice de utilización de las entradas de cache, con el cual el procesador controla el reemplazo de las líneas de cache.

TR5 es el Cache Control Test Register, el cual se usa para seleccionar las operaciones que se realizan sobre el cache (lectura o escritura al buffer intermedio, o al cache mediante el buffer intermedio), y para elegir a que set se escribe, a que línea dentro de cada set, y a que grupo de 32 bits se accede dentro de los 16 bytes que componen una línea.

Luego, para programar el cache debe realizarse para todas las cuatro líneas de los 256 sets que conforman la memoria interna la siguiente operación: primero se debe llenar el buffer intermedio del cache, lo cual se hace en cuatro iteraciones del proceso de escribir un valor en TR3, y luego escribir el comando "00" en los bits de control de TR5, seleccionando en cada iteración un grupo de bytes diferente. Una vez lleno este buffer intermedio (de 16 bytes), se lo debe trasladar al cache. Previamente se debe escribir en TR4 el tag correspondiente, y el bit de validez, para luego escribir en TR5 el comando "01", junto con el número de set deseado, y la línea correspondiente dentro de dicho set. Para un ejemplo mas claro de esto, ver más adelante el código fuente del reproductor.

Durante esta operación es importante que no se permitan accesos a la memoria externa, ya que los buffers TRx son usados por el procesador en su funcionamiento normal, y pueden llegar a verse alterados sin que el programa efectúe operaciones explícitas sobre ellos. Si existe una única aplicación (como es el caso del reproductor de MP3), la única precaución que debe tomarse es no permitir interrupciones, y mantener cierto cuidado en el código para que no se acceda a una variable ubicada en memoria en ciertos pasos clave de la inicialización.

Coprocesador Matemático

El procesador 486 DX4 posee un coprocesador **"**on chip", que permite realizar operaciones de punto flotante con 80 bits de precisión. Si bien la librería MAD utilizada en la reproducción de MP3 es una librería de punto fijo (con la idea de hacerla mas portable a diferentes procesadores que pueden no tener unidad de punto flotante), se le han efectuado modificaciones con el objeto de reducir su tamaño para poder incluirla en la memoria del sistema, junto con un tema musical comprimido con MP3 (lo cual no es una tarea sencilla con solo 128 kb de memoria). Estas modificaciones consisten en la eliminación de varias tablas con las cuales se realizaban operaciones matemáticas (como el elevar una muestra a la potencia ¾), y reemplazarlas por operaciones de punto flotante, mas lentas que la resolución por tablas, pero infinitamente mas pequeñas (a modo de ejemplo, la librería con todas las tablas ocupaba 115kb, mientras que sin las tablas se reducía a menos de 60 kb).

Para utilizar el coprocesador no se necesita prácticamente inicialización, salvo por ejecutar la instrucción "FINIT" antes de ejecutar instrucciones de punto flotante, y cambiar la resolución de trabajo si asi se lo desea (puede variarse entre 32, 64 y 80 bits, siendo la segunda resolución la que se usa por defecto).

La unidad de punto flotante se maneja mediante un "stack" de ocho registros internos de 80 bits mas una serie de registros de control. Los valores para las funciones se ingresan en este stack mediante funciones del tipo push y pop, y desde allí son tomadas por las instrucciones (esto es diferente al funcionamiento normal de la unidad de punto fijo del procesador , donde se puede acceder a cualquier registro de uso general en cualquier orden).

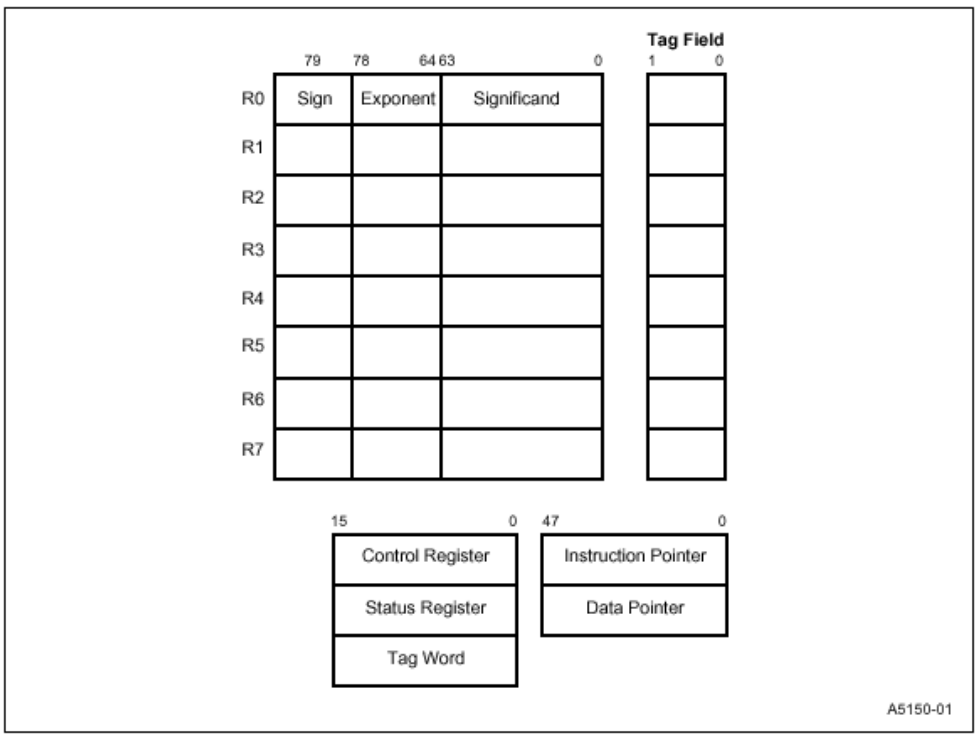

IMAGEN 26 Registros del coprocesador matemático.

Los detalles de la utilización del coprocesador pueden verse mas adelante en este informe en la sección donde se explica el código fuente del reproductor.

MAD

La librería MAD es la encargada de la reproducción de los archivos MP3 en este reproductor. El resto del software actúa meramente como soporte de esta librería. Puede obtenerse gratuitamente en http://www.mars.org/home/rob/proj/mpeg/. La versión utilizada en el proyecto es la 0.14.2b

Características

La librería creada por Robert Leslie y un amplio grupo de colaboradores trabajando en la modalidad "open-source" vía internet, permite la reproducción de archivos de audio comprimidos según el estándar MPEG-I, en sus tres "layers". Está planeado para futuras versiones ampliar esta funcionalidad a estandars de compresión de audio mas avanzados, como AAC , AC-3 (Dolby Digital), o MPEG-II.

MAD fue escrita en lenguaje C, sin instrucciones en assembler, y sin hacer uso de instrucciones de punto flotante., para mantenerla portable a la mayor cantidad de sistemas y tipos de procesadores posibles, muchos de los cuales pueden no tener unidad de punto flotante. En el paquete de software vienen incluidos "makefiles" para una amplia variedad de plataformas, entre ellas para el compilador Visual C++ de Microsoft, con lo cual la incorporación de la librería al proyecto es directa.

La falta de instrucciones de punto flotante se suple generalmente con el uso de tablas para realizar los cálculos con la mayor precisión. Dicha precisión es siempre de 32 bits, lo cual es también una característica que la distingue por sobre otras librerías (ya que todo el procesamiento se hace con una precisión de 32 bits, y recién al final de la conversión se realiza un proceso de "dithering" para reducir la resolución a los 16 bits finales, o a la resolución que sea necesaria en cada caso).

La librería MAD utiliza para su operación enteros de 32 bits, los cuales se interpretan como si estuviesen multiplicados por 2 elevado a la potencia -28. Luego, el valor 1000-0000h equivale a 1.0 en el entorno de MAD (esto es importante, ya que como luego se verá se han hecho modificaciones a la librería, y se debe respetar esta convención).

Utilización

Existen dos variantes para incorporar MAD a un programa, ya que la librería puede operar en modo sincrónico o asincrónico. En el último caso, se crea un "thread" independiente para la librería, y la aplicación se comunica con este mediante mensajes. Esta forma de funcionamiento es la mas simple, pero requiere de un sistema operativo multitarea (como Linux o Windows) para funcionar, lo cual esta fuera del alcance de este proyecto. Por lo tanto, la alternativa lógica a utilizar es la variante sincrónica, donde el programa y la librería corren sobre un mismo "thread", y el programa debe ir llamando a las subrutinas de la librería a medida que desea que estas vayan ejecutándose.

Para entender como se hace esto, la librería incluye un programa de ejemplo minimad.c, que implementa un reproductor de MP3 sincrónico con el mínimo de elementos. La función más importante es la llamada "decode":

```
static
int decode(unsigned char const *start, unsigned long length)
{
  struct buffer buffer;
  struct mad_decoder decoder;
  int result;

  buffer.start  = start;
  buffer.length = length;
  mad_decoder_init(&decoder, &buffer,
                   input, 0 /* header */, 0 /* filter */, output,
                   error, 0 /* message */);
  result = mad_decoder_run(&decoder, MAD_DECODER_MODE_SYNC);
  mad_decoder_finish(&decoder);
  return result;
}
```

Esta función inicializa una estructura privada del programa, llamada buffer, donde guarda la dirección de memoria donde está almacenado el archivo comprimido, junto con su tamaño. Luego llama a la función mad_decoder_init, proporcionándole como parámetros tres funciones implementadas en otro apartado del código, input, output y error. Como puede verse, hay otras funciones como header, filter y message que no se usan, aunque bien podrían implementarse de ser necesarias. Luego se llama a la función mad_decoder_run, indicándose que el modo de funcionamiento será el sincrónico. Esto implica que esta función no retornará hasta que termine la reproducción del archivo. Finalmente la función mad_decoder_finish se encarga de liberar la memoria no utilizada. Las tres funciones que necesita mad_decoder_init son las que siguen:

```
static enum mad_flow input(void *data, struct mad_stream *stream)
{
  struct buffer *buffer = data;
  if (!buffer->length) return MAD_FLOW_STOP;
  mad_stream_buffer(stream, buffer->start, buffer->length);
  buffer->length = 0;
  return MAD_FLOW_CONTINUE;
}

static enum mad_flow output(void *data, struct mad_header const
*header, struct mad_pcm *pcm)
{
  unsigned int nchannels, nsamples;
  mad_fixed_t const *left_ch, *right_ch;
  nchannels = pcm->channels;
  nsamples  = pcm->length;
  left_ch   = pcm->samples[0];
  right_ch  = pcm->samples[1];
```

```
  while (nsamples--) {
    signed int sample;
    sample = scale(*left_ch++);
    putchar((sample >> 0) & 0xff);
    putchar((sample >> 8) & 0xff);

    if (nchannels == 2) {
      sample = scale(*right_ch++);
      putchar((sample >> 0) & 0xff);
      putchar((sample >> 8) & 0xff);
    }
  }

  return MAD_FLOW_CONTINUE;
}

static enum mad_flow error(void *data, struct mad_stream *stream,
                           struct mad_frame *frame)
{
  struct buffer *buffer = data;
  fprintf(stderr, "decoding error 0x%04x (%s) at byte offset %u\n",
          stream->error, mad_stream_errorstr(stream),
          stream->this_frame - buffer->start);
  return MAD_FLOW_BREAK;
}
```

El trabajo de estas tres funciones es muy intuitivo. La función input es llamada por la librería cuando necesita datos de entrada, y lo que debe hacer es retornar un puntero a la dirección de memoria donde esta el MP3 y su duración (o bien informarle a la librería de que ha finalizado el archivo y se debe terminar la reproducción). La función output recibe un "buffer" con los datos ya descomprimidos, y a partir de allí puede realizar un dithering, y enviar las muestras al conversor digital-analógico. Por último, la función error recibe los posibles errores que puedan darse durante la reproducción y los comunica al usuario.

Mas adelante, cuando se vea el código del reproductor de MP3, podrán observarse muchas similitudes con este ejemplo, ya que el código esta basado directamente en el archivo minimad.c

Modificaciones

Como ya se mencionó, la librería MAD no utiliza funciones de punto flotante. Esto es asi para mantener la mayor compatibilidad entre diversas plataformas, muchas de las cuales pueden no tener un coprocesador matemático, como si ocurre con los procesadores de Intel desde el 386DX en adelante.

Durante el proceso de codificación de un archivo de audio a formato MP3, las muestras son elevadas a la potencia ¾. Por lo tanto, para recuperar su valor original, estas deben ser elevadas nuevamente a la potencia 4/3, y a su vez ser multiplicadas por un factor de escala que se obtiene durante el proceso de codificación. La librería MAD resuelve esta operación con el uso de tablas (las

cuales se almacenan en los archivos *.dat), lo cual es muy beneficioso, porque permite convertir las muestras manteniendo la máxima precisión (32 bits) y a una gran velocidad (ya que todo el proceso consiste en ingresar en una tabla, y desplazar el resultado a izquierda o derecha tantas posiciones como indique un exponente que se calcula con anticipación).

El problema con el reproductor de MP3 que se desarrolló es que posee muy poca memoria (128 kb + 16kb de cache), y allí se debe poder incluir el tema musical a reproducir, mas el código que lo reproduce y los datos dinámicos que se generen. Si se observa el archivo rq_table.dat, podrá verse que el mismo tiene 8207 entradas de 32 bits cada una. Esta es precisamente la tabla con la cual se resuelve la operación de potencia antes citada. Como resultado de esta tabla, el tamaño final compilado de la librería ronda los 150kb, mientras que el programa de reproducción de MP3 ronda los 115kb (porque no incluye todos los módulos de la librería). Esto no deja espacio para nada, ni para un archivo MP3 ni para los datos dinámicos que genera el programa (mediante la función malloc), por lo tanto es necesario modificar la librería en este punto.

Para hacer esta modificación, se remueve la tabla, y se reemplaza su funcionalidad por instrucciones de punto flotante que cumplan la misma tarea. Con esto se pierde en velocidad, pero se gana en espacio, y la precisión no se ve afectada (recordemos que la FPU del 486 puede trabajar con 80 bits de precisión).

La función original de la librería MAD, tal como se encuentra en el archivo layer3.c es la siguiente:

```
/*
 * table for requantization
 *
 * rq_table[x].mantissa * 2^(rq_table[x].exponent) = x^(4/3)
 */
static
struct fixedfloat {
  unsigned long mantissa  : 27;
  unsigned short exponent :  5;
} const rq_table[8207] = {
# include "rq_table.dat"
};


/*
 * NAME:       III_requantize()
 * DESCRIPTION:      requantize one (positive) value
 */
static
mad_fixed_t III_requantize(unsigned int value, signed int exp)
{
  mad_fixed_t requantized;
  signed int frac;
  struct fixedfloat const *power;

  frac = exp % 4;  /* assumes sign(frac) == sign(exp) */
```

```
  exp /= 4;

  power = &rq_table[value];
  requantized = power->mantissa;
  exp += power->exponent;

  if (exp < 0) {
    if (-exp >= sizeof(mad_fixed_t) * CHAR_BIT) {
      /* underflow */
      requantized = 0;
    }
    else {
      requantized += 1L << (-exp - 1);
      requantized >>= -exp;
    }
  }
  else {
    if (exp >= 5) {
      /* overflow */
# if defined(DEBUG)
      fprintf(stderr, "requantize overflow (%f * 2^%d)\n",
              mad_f_todouble(requantized), exp);
# endif
      requantized = MAD_F_MAX;
    }
    else
      requantized <<= exp;
  }

  return frac ? mad_f_mul(requantized, root_table[3 + frac]) :
requantized;
}
```

En primer término se reproduce un segmento de la primer parte del archivo, donde se incluye la tabla rq_table.dat. Luego se reproduce la función III_requantize, donde se hace uso de esta tabla.

Como puede verse en el primer comentario del código, los valores de la tabla rq_table son tales que para un dado valor x, se busca la entrada correspondiente en la tabla, se la desplaza de forma binaria según el exponente que también figura allí, y se obtiene el resultado equivalente a elevar x a la 4/3. La función III_requantize hace exactamente esto, teniendo en cuenta posibles desbordes, y dando lugar a un exponente que llega en forma de parámetro (el cual proviene del proceso de codificación y se calcula en otra parte de la librería), y que se incorpora a la cuenta total. Por ultimo, la función contempla la posibilidad de que el exponente total no sea un entero, y resuelve esta situación multiplicando el resultado final por una constante sacada de otra tabla (de menor tamaño, con solo siete entradas) que almacena las potencias fraccionarias de dos (entre -1 y 1, a intervalos de un cuarto).

La modificación realizada consiste en remover la tabla rq_table.dat, y cambiar la función III_requantize a lo siguiente:

```
static
mad_fixed_t III_requantize(unsigned int value, signed int exp)
{
	//Calcular directamente value ^ 4/3, sin tablas.
	mad_fixed_t requantized;
	signed int frac;
	//Es static para que el calculo sea hecho desde la compilacion, y
		no antes.
	static double escala=1.333;
	unsigned uno=1;
	int word_power;

	frac = exp % 4;  // assumes sign(frac) == sign(exp)
	exp/=4;

	_asm
	{
		push eax;
		mov eax,exp;
		add eax,28;
		fld escala;
		fild value;
		fyl2x;
		fist word_power;
		fisub word_power;
		f2xm1;
		fiadd uno;
		add word_power,eax;
		fild word_power;
		fxch;
		fscale;
		fistp requantized;
		ffree st(0);
		pop eax;
	}

	return frac ? mad_f_mul(requantized, root_table[3 + frac]) :
requantized;

}
```

Lo primero que se hace en la función es definir la constante escala igual a 4/3 como del tipo double y static. La razón de declararla static es para que el valor que se almacena en la constante se calcule al momento de la compilación del código, y no al momento de la ejecución. Si la constante no se definiese como static, el código generado por el compilador crearía el valor al momento de usarse la constante, y para esto utilizaría las librerias estándar de punto flotante, las cuales no es seguro utilizar en un entorno como el que tenemos, sin sistema operativo, y sin una inicialización "estandar" de la unidad de punto flotante. Al definirse las constante como "static" esta pasa a ocupar una posición fija de memoria, y su valor es inicializado cuando se "baja" la imagen del código a la memoria del reproductor.

En lo que sigue, se calcula primero si el exponente es divisible por cuatro, y cual es el resto de dicha división. A partir de allí comienzan las instrucciones en assembler, donde el objetivo es calcular la potencia de x por el proceso de logaritmo, multiplicación y antilogaritmo (las operaciones logarítmicas son todas en base 2). Se salva el contenido del registro eax (push eax), y se copia allí el valor del exponente dividido por cuatro (mov eax,exp). A este valor se le suma 28, que es el factor de escala que usa toda la librería MAD (recuérdese que los enteros de 32 bits se interpretan dentro de MAD como multiplicados por 2^-28). A continuación se cargan dos valores en el stack del FPU, el factor de escala (que se carga en ST(1) por haber sido empujado primero en el stack) y el valor pasado como parámetro a la función (que aparecerá en ST(0)). Es oportuno notar que el stack del FPU se organiza en ocho posiciones, ST(0) a ST(7), siendo ST(0) la más recientemente accedida.

La instrucción que sigue es fyl2x, la cual hace el cálculo ST(1)=ST(1) * log2(ST(0)) y realiza un pop del stack (con lo cual el resultado queda en ST(0)). Hecha esta operación, lo único que se necesita para obtener el valor deseado es realizar el antilogaritmo en base dos del numero obtenido, pero esto no se puede hacer directamente ya que la instrucción del procesador que hace esta operación (f2xm1) no toma un valor de x cuyo módulo sea mayor a 1 (es decir, calcula potencias fraccionarias de dos, con módulo menor a uno). Luego, se debe dividir el valor que se encuentra en ST(0) en su parte entera y decimal, utilizando la primera para una multiplicación por desplazamiento hacia derecha o izquierda, y la segunda como parámetro de la instrucción f2xm1. Luego se multiplican los dos resultados, obteniéndose el valor final.

La instrucción fist word_power almacena la parte entera de ST(0) en la variable word_power (sin hacer pop del stack), mientras que la instrucción fisub word_power resta el valor de dicha variable al que está en ST(0) y lo almacena allí mismo, con lo cual se obtiene la parte decimal del resultado de la operación fyl2x realizada previamente. Con este valor en el stack del FPU, se ejecuta la instrucción f2xm1, que eleva 2 al valor almacenado en ST(0) y le resta 1. Para compensar esta resta está la instrucción "fiadd uno". Falta ahora elevar a 2 a la parte entera del exponente, y multiplicar los dos resultados.

El valor almacenado en eax (el exponente enviado como parámetro a la función) se suma a word_power (la parte entera antes obtenida) mediante la instrucción add word_power,eax. Este valor resultante se carga en el stack del FPU (fild word_power), y se realiza un cambio de variables para la operación que sigue (ST(0) y ST(1) se intercambian). Para terminar, se ejecuta la instrucción fscale que multiplica el valor en ST(0) por 2 elevado a la potencia almacenada en ST(1) (que es el valor word_power) y se almacena el resultado. La instrucción ffree st(0) (similar al pop tradicional) se incluye para compensar a la función fscale (que no realiza un pop del stack), y dejar el stack en el mismo estado que al principio de la función (de lo contrario, al ingresarse varias veces a la función se desborda el stack del FPU y se cuelga el programa).

La ultima operación que se ejecuta es similar a la de la función original, y contempla el caso de potencias fraccionarias. Por resolverse la cuenta con una tabla

mínima, se ha decidido no modificarla por motivos de simplicidad. Lo mismo se ha hecho con otras operaciones que realiza la librería MAD con tablas de tamaño reducido en diversas funciones.

Como resultado de la modificación citada, el tamaño de la librería se redujo a alrededor de 70kb, y el tamaño del ejecutable a 55kb, dejando espacio suficiente para archivos musicales de calidad moderada y los datos dinámicos que resulten necesarios.

El reproductor de MP3

Todos los elementos descriptos hasta aquí, se combinan en el software de reproducción de MP3, cuya descripción pormenorizada se brinda a continuación.

Generación de la imagen

El compilador utilizado para trabajar en este proyecto es el Visual C++ de Microsoft. Este compilador puede generar código para el sistema operativo Windows 95 en adelante (tanto para ventanas como en modo consola). Es un compilador creado especialmente para el modo protegido y de 32 bits de los procesadores x86, y no puede generar código para otro tipo de configuración.

El modo consola del sistema operativo Windows se utiliza para brindar compatibilidad con las aplicaciones de DOS. Pero es mucho más que un entorno donde correr aplicaciones viejas (de modo real y de 16 bits), ya que brinda la posibilidad de crear aplicaciones simples, pero que ejecuten en modo protegido de 32 bits. Una aplicación de modo consola es en su estructura similar a una de DOS (no necesita del "Message Pump" típico de las aplicaciones con ventanas, y corre a partir de una función main), aunque bien puede tener multithreading, y hacer uso de todas las ventajas del modo protegido.

En el caso del reproductor de MP3, el Visual C++ se configuró para crear una aplicación de modo consola para Windows, aunque la intención final no sea crear un programa para este sistema operativo, sino uno que pueda funcionar sin sistema alguno. Esta configuración permite que el compilador genere el código especialmente preparado para modo protegido de 32 bits. Lo que resta resolver es la inicialización del procesador en este modo, y pasarle luego el control a la aplicación.

Un punto importante a solucionar es que el compilador genera un ejecutable tal que pueda ser interpretado por Windows. Esto incluye un encabezado antes del código ejecutable en si, que le brinda varios datos al sistema operativo que no tienen sentido en una aplicación como la del reproductor de MP3. Afortunadamente, en http://www.exposecorp.com se puede hallar una aplicación que convierte un ejecutable para modo consola de Windows en una imagen de memoria lista para ser cargada en una ROM y ejecutada, con los valores de las variables estáticas ya resueltos y en posición (el programa se llama Exe2Img).

En la dirección citada se puede encontrar también una serie de artículos que el autor del sitio web , Jean L. Gareau, ha publicado en diversas revistas, y que hablan de cómo crear código para aplicaciones "embedded" (como la de este reproductor

de MP3), y de cómo pasar el procesador desde su estado inicial en modo real al modo protegido. Gran parte del código y las técnicas creadas por Gareau son aplicadas directamente en el reproductor de MP3, por lo cual se recomienda la lectura de todo el material mencionado (el cual se reproduce parcialmente en el apéndice A).

El compilador se configura, como ya se dijo, para crear una aplicación de modo consola, sin utilizar ninguna de las librerías estándar de C (que pueden utilizar parte del hardware típico de una PC, que aquí no está presente). Se desactiva la información de Debug, y se coloca la base del programa en cero (esta base del programa es un offset que se puede agregar). Este número de base es tenido en cuenta al crear las entradas de la GDT del sistema (si bien se deja en cero, puede ser colocado en otro valor de ser necesario).

Hecho todo esto, y agregada la librería MAD al proyecto, se procede a la compilación. Esta compilación resulta exitosa, pero durante el proceso de enlazado (link) de los módulos, se generan varios errores por falta de módulos. Estos módulos que faltan corresponden a algunas funciones estándar de la librería de C que son necesarias para el funcionamiento del programa, pero que el compilador no puede encontrarlas, ya que activamos la opción de no incluirlas automáticamente (ya que queremos tener control sobre todos y cada uno de los módulos que se incluyen en nuestro código). Concretamente las funciones que resultan faltantes son malloc, free, calloc, chkstk y chkesp.

Las ultimas dos funciones, chkstk y chkesp son para el control del stack, y se pueden extraer de las librerías estándar de C (en forma de un archivo .obj) e incluirlas sin problemas (a pesar de que si realmente se produce un desborde del stack, difícilmente estas funciones puedan detectarlo en el ambiente en que se van a ejecutar). Las otras tres funciones deben ser escritas especialmente para el proyecto, ya que las funciones estándar de C que manejan la asignación dinámica de memoria dependen del sistema operativo. El código de dichas funciones se ve mas adelante.

Una vez desarrolladas todas las funciones accesorias necesarias, se compila el código y se obtiene una imagen (gracias al programa Exe2Img) lista para ser subida a la memoria del reproductor, a partir de la posición cero de memoria (ya que la base del programa se especificó como cero, y lo mismo se hizo en la GDT). Sin embargo, por una característica propia del programa Exe2Img y de la forma en que enlaza los módulos el "linker" de Microsoft, la primer instrucción se encuentra en la posición 1000h (esto simplemente significa que la primer instrucción del programa no está en 0h sino en 1000h, y todo el espacio por debajo de 1000h queda disponible para usarse como memoria dinámica).

El código de inicialización

La primer parte del código del reproductor de MP3 es un segmento de assembler tomado del trabajo de Jean Gareau, que inicializa el procesador desde el modo real en el que arranca, hasta el modo protegido, para luego pasarle el control a la aplicación principal, escrita en lenguaje C.

A continuación se reproduce el contenido del archivo entry.asm (las palabras clave DB, DW y DD indican valores constantes que se incluyen directamente en el código, cuyo tamaño es de un byte, dos o cuatro respectivamente):

```
                .386P                   ; Use 386+ privileged instructions

LGDT32 MACRO Addr                       ; 32-bit LGDT Macro in 16-bit
                DB      66h             ; 32-bit operand override
                DB      8Dh             ; lea (e)bx,Addr
                DB      1Eh
                DD      Addr
                DB      0Fh             ; lgdt fword ptr [bx]
                DB      01h
                DB      17h
ENDM

FJMP32 MACRO Selector,Offset            ; 32-bit Far Jump Macro in 16-bit
                DB      66h             ; 32-bit operand override
                DB      0EAh            ; far jump
                DD      Offset          ; 32-bit offset
                DW      Selector        ; 16-bit selector
ENDM

PUBLIC          EntryPoint              ; The linker needs it.
PUBLIC          _Gdt
EXTERN          _main:NEAR

_TEXT           SEGMENT PARA USE32 PUBLIC 'CODE'
                ASSUME      CS:_TEXT, DS:_TEXT, ES:_TEXT, SS:_TEXT

; Entry Point. The CPU is executing in 16-bit real mode.                ;

EntryPoint      PROC  NEAR
                cli
                LGDT32 GdtDesc          ; Load GDT descriptor
                mov     eax,cr0         ; Get control register 0
                or      ax,1            ; Set PE bit (bit #0) in (e)ax
                mov     cr0,eax         ; Activate protected mode!
                jmp     $+2             ; To flush the instruction queue.

; The CPU is now executing in 16-bit protected mode.
; Make a far jump in order to load CS with a selector to a 32-bit
; executable code descriptor.


                FJMP32          08h,Start32     ; Jump to EntryPoint32

; This point is never reached.

EntryPoint      ENDP

; The CPU is now executing in 32-bit protected mode.

Start32         PROC  NEAR
```

```
; Initialize all segment registers to 10h (entry #2 in the GDT)

                mov     ax,10h              ; entry #2 in GDT
                mov     ds,ax               ; ds = 10h
                mov     es,ax               ; es = 10h
                mov     fs,ax               ; fs = 10h
                mov     gs,ax               ; gs = 10h
                mov     ss,ax               ; ss = 10h

; Set the top of stack to allow stack operations.
                mov     esp,1ffech          ; empieza arriba del todo
; Call main(), which is not expected to return.

                call    _main

; In case main() returns, enter an infinite loop.

IdleLoop:
                hlt
                jmp     IdleLoop

; This point is never reached.

Start32         ENDP

;-----------------------------------------------------------------------------
; Tables Descriptors (to use with LGDT32 & LIDT32)
;-----------------------------------------------------------------------------
                ALIGN 4
GdtDesc:                                    ; GDT descriptor
                DW      GDT_SIZE - 1        ; GDT limit
                DD      _Gdt                ; GDT base address (below)

; Global Descriptor Table (GDT)
_Gdt:

; GDT[0]: Null entry, never used.

        DD      0
        DD      0

; GDT[1]: Executable, read-only code, base address of 0, limit of
FFFFFh,  granularity bit (G) set (making the limit 4GB)

        DW      0FFFFh          ; Limit[15..0]
        DW      0000h           ; Base[15..0]
        DB      00h             ; Base[23..16]
        DB      10011010b       ; P(1) DPL(00) S(1) 1 C(0) R(1) A(0)
        DB      11001111b       ; G(1) D(1) 0 0 Limit[19..16]
        DB      00h             ; Base[31..24]

; GDT[2]: Writable data segment, covering the save address space than
; GDT[1].
```

```
        DW      0FFFFh          ; Limit[15..0]
        DW      0000h           ; Base[15..0]
        DB      00h             ; Base[23..16]
        DB      10010010b       ; P(1) DPL(00) S(1) 0 E(0) W(1) A(0)
        DB      11001111b       ; G(1) B(1) 0 0 Limit[19..16]
        DB      00h             ; Base[31..24]

GDT_SIZE    EQU   $ - _Gdt              ; Size, in bytes

_TEXT       ENDS
            END
```

Lo primero que aparece en el código son dos macros, LGDT32 y FARJMP32. Estos macros contienen instrucciones en assembler codificadas directamente con los opcodes de 16 bits del procesador. Esto se hace asi porque el compilador de assembler usado es de 32 bits, pero cuando el procesador inicia su funcionamiento esta ejecutando en modo 16 bits. Si bien hay ensambladores que pueden manejar segmentos de 16 y 32 bits sin problemas (como es el caso del programa MASM utilizado), el linker del Microsoft Visual C++ no acepta esta "mezcla", por lo cual se lo "engaña" haciendo aparecer todo el código como de 32 bits.

Aquí me permito realizar un apartado para destacar una particularidad del código presentado. Si bien asi como se lo presentó este código escrito por Jean Gareau funciona perfectamente, en lo anterior se incluye un error, el cual fue descubierto por mera casualidad, y no se ha corregido para preservar lo más intacto posible el material original. El error radica en la utilización del opcode 66h (operand size override), que sirve para hacer que el procesador cambie el tamaño de los operandos que espera para la instrucción (si esta en modo 16 bits, esperará un operando de 32 bits en la próxima instrucción), para cargar en EBX el valor de la base del descriptor de la GDT. La falla está en no haber incluido también el opcode 67h (address size override), el cual le indica al procesador que la dirección que a continuación se expone debe ser interpretada como de 32 bits (y no de 16 como es el funcionamiento por defecto en el estado inicial del procesador), que seria lo correcto (ya que el parámetro Addr es de 32 bits). En resúmen, el 66h hace que BX se transforme en EBX, y el 67h hace que la dirección esperable pase de ser de 16 bits a 32.

Es decir que según como se encuentra el código, el parámetro Addr es interpretado como de 16 bits. La pregunta es por que esto, a pesar del error, puede funcionar. Y la respuesta es que el parámetro Addr generalmente es un valor menor a FFFFh (sobre todo si la aplicación posee dirección base cero), por lo cual la carga en ebx se ejecuta igual, y Addr no se ve recortado (ya que sus 16 bits mas significativos son ignorados pero no importa porque están en cero). Los 16 bits ignorados por la instrucción "lea" quedan en memoria, y son interpretados como la instrucción siguiente. Si ambos bytes están en cero, la instrucción que resulta interpretada no causa problemas, pero si se tratase de otro valor (F4h por ejemplo es la instrucción HALT) podrían producirse errores. Esta situación puede presentarse en aplicaciones donde la base del programa no es cero (cosa que se da en todos los ejemplos de Gareau, y por eso nunca se le ha presentado el error), o

donde por algún motivo se desea colocar la GDT en alguna posición de memoria alta (más allá de los primeros 64kb). Lo mismo se puede argumentar para la instrucción siguiente "lgdt [bx]" que sufre del mismo problema.

El código comienza en la etiqueta Entrypoint, deshabilitando las interrupciones, y llamando al macro LGDT32, con la dirección del descriptor de la GDT como parámetro (que es una posición de memoria de 6 bytes bajo la etiqueta GdtDesc, que contiene la dirección base de la GDT y su tamaño en bytes menos uno). Cargada la GDT, a continuación se modifica el flag del registro CR0 que controla el paso a modo protegido (el bit menos significativo de CR0), y se ejecuta un salto relativo hacia la siguiente instrucción para limpiar los registros internos y la cadena de prefetch del procesador (lo cual es obligatorio de realizar para asegurar un comportamiento determinístico del procesador). A partir de ese salto, el procesador está ejecutando en modo protegido, de 16 bits. Un nuevo salto, esta vez intersegmento, pone al procesador en modo 32 bits. (dato el cual es extraido por el procesador de la información almacenada en la GDT) Este salto intersegmento se hace nuevamente mediante un macro (FARJMP32), que guarda los opcodes necesarios para hacer ese salto con instrucciones de 16 bits.

El salto se realiza al segmento 08h (que apunta a la primer entrada válida de la GDT con un nivel de privilegio cero, la cual se ha asignado a un segmento de código de 32 bits, cuyo tamaño es de 4GB con base cero), y con un offset tal que continúe la ejecución en la etiqueta Start32. Recuérdese que los selectores de segmento en modo protegido se desplazan tres bits a la izquierda, ya que los tres bits menos significativos tienen otra función asignada. Desde esta etiqueta en adelante no es necesario usar mas opcodes en su forma directa, ya que el procesador está ejecutando en modo protegido de 32 bits.

Una vez alcanzado el modo protegido de 32 bits, se inicializan todos los registros de segmentos destinados a datos con la entrada 10h de la GDT (es decir, la segunda entrada válida de la GDT), incluido el segmento de stack (SS). El puntero de stack a su vez se coloca en la posición más alta de memoria (luego será corrido mas arriba aun cuando se habilite el cache como memoria estática), pero por debajo del vector de reset del procesador (que se encuentra en 1FFF0h). Por último se llama a la función main, que es el punto de entrada al código escrito en lenguaje C.

Al final del código se puede ver la GDT, construida de forma estática, con una entrada vacía (la primera, lo cual es obligatorio), una entrada para un segmento de código y una para un segmento de datos, ambos ocupando todo el espacio de memoria disponible (4GB con base en cero). También puede verse el descriptor de la GDT (los 6 bytes mencionados) usados como parámetro para el macro LGDT32.

Como puede notarse, el paso a modo protegido tarda mas en explicarse que en realizarse. Con unas pocas instrucciones ya se ha desatado todo el poder de procesamiento del 486, y se está listo para ejecutar la aplicación principal.

La aplicación principal

Este es el código completo del reproductor de MP3, salvo la sección de inicialización ya detallada (los comentarios del código han sido removidos).

player.h:

```
#define IDT_VECTORS     0x40
#define TMR_IRQ         0x20
#define MP3_BASE                57344
#define MP3_SIZE                19240
#define MEM_BASE                57344+MP3_SIZE
#define MAX_STACK               1024
#define MAX_MEM         0x4000 + (128*1024-MEM_BASE) - MAX_STACK
#define CACHE_BASE      0x20000
#define NMALLOC_ORG     64
#define CLOCKIRQ                3579545
#define CLOCK_PER               2793658*3

struct IDT_DESC
{
        short unsigned offl;
        short unsigned seg;
        short unsigned flags;
        short unsigned offh;
};

struct buffer {
        unsigned char const *start;
        unsigned long length;
};

struct malloc_org {
        unsigned base;
        unsigned size;
 };

static IDT_DESC idt[IDT_VECTORS];
static malloc_org morg[NMALLOC_ORG];

static struct {
        unsigned char bytes[6];
} IDT_load;

static union
{
        unsigned outdw;
        struct {
                unsigned char dac;
                unsigned char tmr;
                unsigned char irqcode;
                unsigned char unused;
        } port;
};

static struct
{
        bool rdy, wait;
        unsigned samples;
        unsigned div,divinc;
        unsigned char rdbuf, wrbuf;
        unsigned char buf[2][1152];
} out_ctrl;
void out(unsigned val, short unsigned port=0);
void timer(void);

#include "mad\mad.h"
```

```
static enum mad_flow input(void *data,struct mad_stream *stream);
static enum mad_flow output(void *data, struct mad_header const *header, struct
mad_pcm *pcm);
static enum mad_flow error(void *data, struct mad_stream *stream, struct mad_frame
*frame);
unsigned char scale(mad_fixed_t sample);

extern "C"
{
        void *malloc(unsigned);
        void *calloc(unsigned,unsigned);
        void free(void *);
        void reportar (unsigned line);
}
```

player.c:

```
#include "player.h"

#ifdef MYDEBUG
#define report reportar(__LINE__)
#else
#define report
#endif

static unsigned *report_p;

main()
{

        static struct buffer buffer;
        static struct mad_decoder decoder;

        report_p=(unsigned *)0;
        report;

        outdw=0;
        port.irqcode=TMR_IRQ;
        port.tmr=0xfe;
        out(outdw);
        unsigned short *idtsize=(unsigned short *)&IDT_load.bytes[0];
        unsigned *idtbase = (unsigned *) &IDT_load.bytes[2];
        *idtsize=sizeof(IDT_DESC) * IDT_VECTORS - 1;
        *idtbase=(unsigned) idt;

        for(unsigned j=0;j<IDT_VECTORS;j++)
        {
                idt[j].offl=0;
                idt[j].offh=0;
                idt[j].flags=0;
                idt[j].seg=0;
        }

        unsigned timer_ptr=(unsigned) timer;
        idt[TMR_IRQ].offl = timer_ptr & 0xffff;
        idt[TMR_IRQ].offh = timer_ptr >> 16;
        idt[TMR_IRQ].seg = 0x08;
        idt[TMR_IRQ].flags = 0x8E00;
        unsigned char *px3=&IDT_load.bytes[0];
        _asm
        {
                push ebx;
                mov ebx,px3;
                lidt [ebx];
                pop ebx;
```

```
}


unsigned tag=CACHE_BASE >> 12;

_asm
{
        push eax;
        push ebx;
        xor eax,eax;
        mov tr5,eax;
        mov tr3,eax;

        mov eax,0x4;
        mov tr5,eax;
        xor eax,eax;
        mov tr3,eax;
        mov eax,0x8;
        mov tr5,eax;
        xor eax,eax;
        mov tr3,eax;
        mov eax,0xc;
        mov tr5,eax;
        xor eax,eax;
        mov tr3,eax;
}


for(unsigned cache_bank=0;cache_bank<4;cache_bank++)
{
        //Setear los 4 bancos
        for(unsigned cache_set=0;cache_set<256;cache_set++)
        {
                //Setear los 256 sets de datos.
                _asm
                {
                        mov eax,tag;
                        shl eax,12;
                        or eax, 0x400;
                        mov ebx,cache_set;
                shl ebx,2;
                        or ebx,cache_bank;
                        shl ebx,2;
                        or ebx,0x1;

                        mov tr4,eax;
                        mov tr5,ebx;
                }
        }
        tag++;
}
_asm
{
        pop ebx;
        pop eax;
}

report;
unsigned *stack_old, *stack_new;
unsigned *stack_pointer;

_asm mov stack_pointer,esp;

stack_old=(unsigned *) (0x1fff0-4);
stack_new=(unsigned *) (0x24000-4);

while(stack_old!=stack_pointer)
{
```

```
                *stack_new=*stack_old;
                stack_new--;
                stack_old--;
        }

        unsigned esp_new=(unsigned)stack_new;
        _asm mov esp,esp_new;

        _asm finit;
        report;


        for(j=0;j<NMALLOC_ORG;j++)
        {
                morg[j].base=0;
                morg[j].size=0;
        }

        morg[0].size=MAX_MEM;

        out_ctrl.rdy=false;
        out_ctrl.wait=false;
        out_ctrl.div=0;
        out_ctrl.divinc=CLOCK_PER*256;
        out_ctrl.rdbuf=0;
        out_ctrl.wrbuf=0;

        for(j=0;j<1152;j++)
        {
                out_ctrl.buf[0][j]=0;
                out_ctrl.buf[1][j]=0;
        }

        int result;

        report;

        buffer.start  = (unsigned char *) MP3_BASE;
        buffer.length = MP3_SIZE;

        mad_decoder_init(&decoder, &buffer,
                input, 0 /* header */, 0 /* filter */, output,
                error, 0 /* message */);

        report;
        port.tmr=0xff;                                                  out(outdw);
        _asm sti;

        report;
        result = mad_decoder_run(&decoder, MAD_DECODER_MODE_SYNC);
        report;
        reportar(result);

        mad_decoder_finish(&decoder);

        reportar(0xaabbaabb);
        _asm hlt;

        return 0;
}

static
enum mad_flow input(void *data,
                                        struct mad_stream *stream)
{
        struct buffer *buffer = (struct buffer *) data;
        report;
        mad_stream_buffer(stream, buffer->start, buffer->length);
```

```
        report;
        return MAD_FLOW_CONTINUE;
}

static
enum mad_flow output(void *data,
                                        struct mad_header const *header,
                                        struct mad_pcm *pcm)
{
        unsigned int nsamples;
        mad_fixed_t const *audio;
        unsigned char *buf;

        switch(pcm->samplerate)
        {
        case 8000:
                out_ctrl.div=125000000;
                break;
        case 11025:
                out_ctrl.div=90702948;
                break;
        case 22050:
                out_ctrl.div=45351474;
                break;
        case 44100:
                out_ctrl.div=22675737;
                break;
        case 48000:
                out_ctrl.div=20833333;
                break;
        default:
                out_ctrl.div=125000000;
                break;
        }

        nsamples = out_ctrl.samples  = pcm->length;
        audio   = pcm->samples[0];
        buf = out_ctrl.buf[out_ctrl.wrbuf];
        while (nsamples--)          *buf++ = scale(*audio++);
        out_ctrl.rdy=true;
        if(out_ctrl.wrbuf) out_ctrl.wrbuf=0;
        else out_ctrl.wrbuf=1;
        while(out_ctrl.wait) _asm nop;
        out_ctrl.wait=true;
        return MAD_FLOW_CONTINUE;
}

static
enum mad_flow error(void *data,
                                        struct mad_stream *stream,
                                        struct mad_frame *frame)
{
        struct buffer *buffer = (struct buffer *)data;
        report;
        reportar(stream->error);
        reportar(stream->this_frame - buffer->start);
        _asm hlt;
        return MAD_FLOW_BREAK;
}

void *malloc(unsigned size)
{
        report;
        reportar(size);
        if(!size) return 0;

        for(unsigned j=0;j<NMALLOC_ORG;j++)
        {
```

```
                if(!(morg[j].size & 0x80000000) && morg[j].size>=size)
                {
                        for(unsigned k=0;k<NMALLOC_ORG;k++)
                        {
                                if(!morg[k].size) break;
                        }
                        if(k==NMALLOC_ORG) _asm hlt;
                        morg[k].base=morg[j].base;
                        morg[k].size=0x80000000 | size;
                        morg[j].base+=size;
                        morg[j].size-=size;

                        unsigned *ptr=(unsigned *) ( ( (unsigned) morg[k].base ) +
(unsigned) MEM_BASE);
                        unsigned *ptr1=ptr;

                        for(k=0;k<(size>>2);k++)
                        {
                                *ptr1=0;
                                ptr1++;
                        }
                        report;
                        return (void *) ptr;
                }
        }
        if(j==NMALLOC_ORG) _asm hlt;
        return 0;
}

void free(void *ptr)
{
        report;
        if((unsigned)ptr<MEM_BASE) return;
        short unsigned base = (unsigned)ptr - (unsigned) MEM_BASE;

        for(unsigned j=0;j<NMALLOC_ORG;j++)
        {
                if(morg[j].base==base && (morg[j].size & 0x80000000) )
                {
                        morg[j].size&=0x4FFF;
                }
        }

        if(j==NMALLOC_ORG) _asm hlt;

        for(j=0;j<NMALLOC_ORG;j++)
        {
                if(morg[j].size & !(morg[j].size & 0x80000000) )
                {
                        for(unsigned k=j+1;k<NMALLOC_ORG;k++)
                        {
                                if(morg[k].size & !(morg[k].size & 0x80000000) )
                                {
                                        if(morg[j].base+morg[j].size==morg[k].base
                                                ||
morg[k].base+morg[k].size==morg[j].base)
                                        {
                                                if(morg[j].base>morg[k].base)
morg[j].base=morg[k].base;
                                                morg[j].size+=morg[k].size;
                                                morg[k].size=0;
                        morg[k].base=0;
                                                j=0;
                        break;
                                        }
                                }
                        }
                }
```

```
	}
	report;

}

void *calloc(unsigned nitems, unsigned size)
{
	report;
	return malloc(nitems*size);
}


void out(unsigned val, short unsigned port )
{
	_asm
	{
		push eax;
		push dx;
		mov eax,val;
		mov dx,port;
		out dx,eax;
		pop dx;
		pop eax;
	}
}

__declspec ( naked ) void timer(void)
{
	_asm pushad;

	static unsigned sample=0, div=0;

	if(out_ctrl.rdy)
	{
		if(div>out_ctrl.div)
		{
			div-=out_ctrl.div;

			port.dac=out_ctrl.buf[out_ctrl.rdbuf][sample];
			out(outdw);
			sample++;

			if(sample>=out_ctrl.samples)
			{
				sample=0;
				if(out_ctrl.rdbuf) out_ctrl.rdbuf=0;
	else out_ctrl.rdbuf=1;

				out_ctrl.wait=false;
			}
		}
		else div+=out_ctrl.divinc;
	}

	port.tmr=0xfe;
	out(outdw);
	port.tmr=0xff;
	out(outdw);

	_asm
	{
		popad;
		iretd;
	}
}

void reportar (unsigned line)
{
```

```
        *report_p=line;
        report_p++;
        if((unsigned)report_p>=0x100) report_p=0;
}


unsigned char scale(mad_fixed_t sample)
{
 sample += (1L << (MAD_F_FRACBITS - 8));
 if (sample >= MAD_F_ONE)
   sample = MAD_F_ONE - 1;
 else if (sample < -MAD_F_ONE)
   sample = -MAD_F_ONE;
 return (sample >> (MAD_F_FRACBITS + 1 - 8))+127;
}
```

El significado y función de cada una de las constantes y estructuras definidas en player.h se irá explicando a medida que sean utilizadas en el código.

La función main comienza con el siguiente código de inicialización:

```
        static struct buffer buffer;
        static struct mad_decoder decoder;

        /*Inicializacion de la plaqueta*/
        report_p=(unsigned *)0;
        report;
        outdw=0;
        port.irqcode=TMR_IRQ;                    //Setear codigo de IRQ
        port.tmr=0xfe;
        out(outdw);
```

Lo primero que se hace aquí es definir dos estructuras utilizadas por la librería MAD. Luego se asigna el valor cero al puntero report_p, el cual es usado por la función reportar para copiar valores en memoria que pueden ser usados para hacer "debugging" del programa. Luego se llama al macro report (cuya única tarea es la de reportar en memoria que se ha alcanzado la línea de código en que se encuentra), y luego se inicializa la estructura usada para enviar datos a los tres puertos de salida. Esta estructura es una unión entre un entero de 32 bits (outdw), y cuatro enteros de 8 bits (dac, tmr, irqcode y unused), lo cual permite acceder a cada uno de los puertos individualmente, sin tener que estar haciendo operaciones de desplazamiento binarias. La palabra outdw (que representa a los cuatro puertos en conjunto) se inicializa a cero, poniendo en principio todos los puertos en cero (aunque los puertos no son escritos hasta que no se ejecuta la función out, que espera como parámetro un entero de 32 bits). Luego se asigna a port.irqcode el código de interrupción que habrá de recibir el procesador cada vez que genere un ciclo de bus del tipo INTA. Este código se denomina TMR_IRQ, y tiene un valor de 20h. A port.tmr se le asigna el valor FEh, lo cual genera la máxima división que puede obtenerse por parte del circuito de interrupciones, pero este no comienza aun a funcionar (ya que el bit menos significativo de port.tmr está en cero, por lo cual los contadores del circuito están detenidos). Hecho todo esto, se llama a la función out, que escribe el valor de 32 bits en los puertos (en teoría una función como out también necesita de una dirección de puerto, pero como en este diseño solo hay tres puertos de 8 bits, no es necesario usar direccionamiento porque

cualquier dirección que se use da lo mismo). Lo que sigue dentro de main es lo siguiente:

```
unsigned short *idtsize=(unsigned short *)&IDT_load.bytes[0];
unsigned *idtbase = (unsigned *) &IDT_load.bytes[2];
*idtsize=sizeof(IDT_DESC) * IDT_VECTORS - 1;
*idtbase=(unsigned) idt;

/*Inicializacion de la tabla de interrupciones*/
for(unsigned j=0;j<IDT_VECTORS;j++)
{
        idt[j].offl=0;
        idt[j].offh=0;
        idt[j].flags=0;
        idt[j].seg=0;
}

/*Inicializacion de la entrada 20h de la tabla...apunta a nuestra funcion timer*/
unsigned timer_ptr=(unsigned) timer;
idt[TMR_IRQ].offl = timer_ptr & 0xffff;
idt[TMR_IRQ].offh = timer_ptr >> 16;
idt[TMR_IRQ].seg = 0x08;
idt[TMR_IRQ].flags = 0x8E00;      //P=1 DPL=00 0 D=1 1 1 0 0 0 0 x x x x x
```

En esta sección tenemos la inicialización de la IDT, la tabla de descriptores de interrupción. En si la tabla está casi toda vacía, salvo por la entrada correspondiente a la interrupción 20h, que es la que debemos manejar, ya que es la que generará nuestro circuito externo (tal como lo hemos programado en el segmento anterior de código). Las primeras dos líneas declaran dos punteros, idtsize y idtbase, los cuales se asignan a diferentes partes de la estructura IDT_load (bytes 0 y 2 respectivamente). Esta estructura de 6 bytes contendrá la dirección base de la IDT y su tamaño (el uso de los punteros es para facilitar la colocación de estos datos en la estructura), de la misma manera que se hizo con GdtDesc en el archivo entry.asm. A continuación sigue la asignación de los valores citados (base de la IDT y tamaño), más un bucle que inicializa la IDT con ceros.

Por último, se ingresa en la entrada 20h (TMR_IRQ) de la IDT la dirección de la función timer (que será la encargada de procesar las interrupciones), en el formato dado en la documentación del procesador, más los flags que allí también se indican.

```
/*Cargar la tabla*/
unsigned char *px3=&IDT_load.bytes[0];
_asm
{
        push ebx;
        mov ebx,px3;
        lidt [ebx];
        pop ebx;
}
```

La parte final en la carga de la IDT consiste en este segmento de assembler que ejecuta la instrucción lidt con la dirección de la estructura IDT_load como parámetro. Cargada la IDT, el código de inicialización procede a configurar el cache para su funcionamiento como memoria estática.

```
/*Inicialización del cache en la direccion 20000h a 24000h (16kb)*/
/*Se utilizan los registros de test de cache para programarlo*/

unsigned tag=CACHE_BASE >> 12;
```

```
_asm
{
        push eax;
        push ebx;
        //Poner a cero el cache fill buffer de 16 bytes
        xor eax,eax;
        mov tr5,eax;            //TR5= Write fill buffer [0]
        mov tr3,eax;            //TR3=0

        mov eax,0x4;
        mov tr5,eax;            //TR5= Write fill buffer [1]
        xor eax,eax;
        mov tr3,eax;            //TR3=0

        mov eax,0x8;
        mov tr5,eax;            //TR5= Write fill buffer [2]
        xor eax,eax;
        mov tr3,eax;            //TR3=0

        mov eax,0xc;
        mov tr5,eax;            //TR5= Write fill buffer [3]
        xor eax,eax;
        mov tr3,eax;            //TR3=0
}
```

Para comenzar la configuración, se pone a cero el "cache fill buffer", el buffer temporario de cache de 16 bytes, que se accede mediante escrituras a TR3, y utilizando a TR5 como ""indice" para seleccionar que grupo de 4 bytes de los 16 que conforman el cache fill buffer se está accediendo. La variable tag será utilizada a continuación, y se inicializa con un valor igual al de la dirección a partir de la cual se asignará el cache, desplazada 12 bits a la derecha (dejando los 20 bits más significativos). Luego, con cuatro escrituras a TR3 y TR5 se inicializa el cache fill buffer a cero.

```
for(unsigned cache_bank=0;cache_bank<4;cache_bank++)
{
        //Setear los 4 bancos
        for(unsigned cache_set=0;cache_set<256;cache_set++)
        {
                //Setear los 256 sets de datos.
                _asm
                {
                        mov eax,tag;    //Cache Status: TAG + Valid Bit
                        shl eax,12;
                        or eax, 0x400;  //EAX=TR4

                        mov ebx,cache_set;      //EBX=TR5
                        shl ebx,2;
                        or ebx,cache_bank;
                        shl ebx,2;
                        or ebx,0x1;

                        mov tr4,eax;    //Entre estas dos instrucciones no
                        mov tr5,ebx;    //puede haber referencias externas
                                        //a memoria.
                }
        }
        tag++;
}
_asm
{
        pop ebx;
```

```
        pop eax;
}
```

Este bucle recorre cada uno de los 256 sets del cache, y cada una de sus cuatro líneas, asignándoles a todas un tag y un bit de validez, mediante el uso de los registros TR4 y TR5 para grabar en cada línea el contenido del cache fill buffer (y al mismo tiempo el tag y bit de validez). El tag no es otro valor que los 20 bits más significativos de la dirección de memoria que el cache ha de representar (ya se ha mencionado en el apartado del cache como se completa el procesador los 12 bits restantes). Como puede verse, el bucle recorre primero todas las primeras líneas de los 256 sets, asignándoles un tag y bit de validez, para luego incrementar el tag en uno, y repetir el proceso en otras tres oportunidades. Nótese el uso de los registros eax y ebx, lo cual es muy importante, ya que como se comenta en el código, hay dos instrucciones entre las cuales no puede haber accesos a memoria externa, porque se alterarían los contenidos de los registros TRx. Completadas todas las iteraciones, el cache queda inicializado con sus contenidos en cero (gracias a que se puso el cache fill buffer en cero al comienzo de la inicialización), y ocupando una posición de memoria entre los 20000h y 23FFFh. Recuerdese que el stack habia sido colocado temporalmente a partir de 1FFECh, pero ahora puede ser corrido más arriba (ya que el stack en la arquitectura x86 crece "hacia abajo" por defecto).

```
/*Movilizar el stack desde 1ffech a 24000h*/
report;
unsigned *stack_old, *stack_new;
unsigned *stack_pointer;

_asm mov stack_pointer,esp;

stack_old=(unsigned *) (0x1fff0-4);
stack_new=(unsigned *) (0x24000-4);              //4 bytes antes del tope

while(stack_old!=stack_pointer)
{
        *stack_new=*stack_old;
        stack_new--;
        stack_old--;
}

unsigned esp_new=(unsigned)stack_new;
_asm mov esp,esp_new;
```

Puede verse aquí la movilización del stack desde su posición inicial a la nueva, mas el cambio del puntero esp para reflejar esta nueva situación. A partir del cambio de esp, se puede usar la zona donde antes estaba el stack para asignación dinámica de memoria.

```
/*Inicializar el FPU*/
_asm finit;
report;

/*Inicializacion de los bloques para organizar los mallocs*/
for(j=0;j<NMALLOC_ORG;j++)
{
        morg[j].base=0;
        morg[j].size=0;
```

```
}

morg[0].size=MAX_MEM;              //el msb indica que esta libre (=cero).


/*inicializacion de la estructura que se usa para comunicarse con el timer*/
out_ctrl.rdy=false;
out_ctrl.wait=false;
out_ctrl.div=0;
out_ctrl.divinc=CLOCK_PER*256;              //La división es por 256.
out_ctrl.rdbuf=0;
out_ctrl.wrbuf=0;

for(j=0;j<1152;j++)
{
        out_ctrl.buf[0][j]=0;
        out_ctrl.buf[1][j]=0;
}
```

APENDICE A – Jean Gareau!!

Índice