CENTRO DE INVESTIGACIÓN Y DE ESTUDIOS AVANZADOS
DEL INSTITUTO POLITÉCNICO NACIONAL
DEPARTAMENTO DE INGENIERÍA ELÉCTRICA
SECCIÓN DE COMPUTACIÓN

Captura de Múltiples Eventos MIDI en Tiempo de Ejecución

Tesis que presenta

Maximino Peña Guerrero

Para Obtener el Grado de

Doctor en Ciencias

En la Especialidad de

Ingeniería Eléctrica

Opción Computación

Director de la Tesis: Dr. Adriano De Luca Pennacchia.
Codirector: Dr. Jorge Buenabad Chávez.

México D.F. Enero 2005

RESUMEN

El protocolo de comunicaciones entre instrumentos musicales MIDI (*Musical Instrument Digital Interface*) fue introducido en 1983, y desde entonces se ha desarrollado un gran número de aplicaciones musicales. Existen sistemas de software con los cuales se puede editar una partitura musical tal como se hace con un texto, transportar las notas a otro tono, construir secuencias de acompañamiento, insertar la voz de un cantante, o bien, simular una orquesta real. La captura de los eventos MIDI de un instrumento actualmente está limitada en fidelidad, y la de varios instrumentos MIDI no es posible debido a que el sistema MIDI no fue diseñado para capturar datos en paralelo. Este tipo de captura haría posible *arreglos* con sonidos de distintos instrumentos y la extracción precisa y automática de las partituras de cada instrumento.

Esta tesis presenta las alternativas de diseño de un sistema de grabación de eventos MIDI de varias fuentes y propone dos diseños específicos: *MIDI Capture System Linear-memory* (MCS-L) y *MIDI Capture System Segmented-Memory* (MCS-S). MCS-L utiliza una memoria lineal convencional, y es adecuado para desarrollar un prototipo en poco tiempo. MCS-S utiliza un diseño especial de memoria segmentada autocompactante (MSA) que facilita su producción masiva y muy posiblemente reduce su costo ya que sólo utiliza tres circuitos integrados de propósito específico. MSA inserta los datos MIDI de cada instrumento en un segmento distinto de tamaño variable y autocompactante, no desperdiciando memoria si uno de los instrumentos no genera datos.

Para probar MCS-L desarrollamos un prototipo con un sistema mínimo basado en el microcontrolador 8051, y el software de control del mismo. Capturamos datos MIDI de varios instrumentos, y con un compilador que desarrollamos, llamado KL (K*ernel for music* L*anguage*), obtuvimos sus partituras con una calidad aceptable. Para probar MCS-S sólo evaluamos su MSA, el único componente distinto de los de MCS-L. Nuestra evaluación, en base al sistema de desarrollo de sistemas digitales Xilinx, muestra que MSA es mucho más rápida que una memoria convencional, garantizando la captura fiel de los eventos MIDI.

ABSTRACT

The protocol of communication among musical instruments MIDI (Musical Instrument Digital Interface) it was introduced in 1983, and from then on a great number of musical applications it has been developed. Software systems exist with those which one can edit a musical score just as it is made with a text, to transport the notes to another tone, to build accompaniment sequences, to insert the voice of a singer, or to simulate a real orchestra. The capture of the events MIDI of an instrument at the moment it is limited in fidelity and that of several instruments is not possible because the MIDI system it was not designed to capture data in parallel. This capture type would make possible arrangements with sounds of different instruments and the precise and automatic extraction of the scores of each instrument.

This thesis presents the design alternatives of a MIDI events recording system of several sources and it proposes two specific designs: MIDI Capture System Linear-Memory (MCS-L) and MIDI Capture System Segmented-Memory (MCS-S). MCS-L uses a conventional lineal memory, and it is adapted to develop a prototype in little time. MCS-S uses a special design of self-compacting segmented memory (MSA) that it facilitates their massive production and very possibly it reduces their cost since alone it uses three integrated circuits of specific purpose. MSA inserts the data MIDI of each instrument in a different segment from variable size and self-compacted, not wasting memory if one of the instruments doesn't generate data.

In order to prove MCS-L we develop a prototype with a minimum system based on the 8051 microcontroller, and its software of control of the one. We capture data MIDI of several instruments, and with a compiler that we have developed, named KL (Kernel for music Language), we obtained their scores with acceptable quality. In order to prove alone MCS-S we evaluate their MSA, the only component different from those of MCS-L. Our evaluation, based on the development system of digital systems Xilinx, it shows that MSA is much quicker that a conventional memory, guaranteeing the faithful capture of MIDI events.

Xóchitl Peña Navarro,

Ma. del Rosario Peña Navarro,

Ma. Teresa Navarro Serrano,

Elena Guerrero Badillo,

Alberto Peña Ramírez,

Instituto Politécnico Nacional...

Dios:

Gracias.

Agradecimientos

Agradezco a mi director de tesis, Dr. Adriano De Luca Pennacchia la guía, asesoría y la oportunidad que me dio al formar parte de su grupo de investigadores, con lo cual fue posible desarrollar el presente proyecto de tesis doctoral.

De la misma forma, también quiero hacer patente mi deuda y agradecimientos al Dr. Jorge Buenabad Chávez, codirector de esta tesis, cuyas sugerencias, ideas, y señalamientos enriquecieron el contenido del proyecto.

De una manera muy especial, agradezco al Dr. Manuel González Hernández los consejos que recibí como amigo, maestro, y compañero de trabajo durante aquellos momentos difíciles de mi vida; asimismo las revisiones, sugerencias y correcciones que hizo a este documento. Muchas gracias *Manix*.

Estoy agradecido con: Dr. Oscar Olmedo Aguirre, cuyas recomendaciones ayudaron a mejorar la estructura del texto; Dr. Miguel Ángel León Chávez, invirtió su valioso tiempo en revisar el documento final, sus observaciones y sugerencias enriquecieron esta tesis; Dra. Graciela Roman Alonzo, amablemente hizo revisiones y recomendaciones que también enriquecieron la tesis; y, Dr. Luis Gerardo de La Fraga, hizo las primeras observaciones que ayudaron a bien organizar nuestro proyecto.

Reconozco mi deuda con el Dr. Sergio V. Chapa V., por el apoyo y facilidades que permitieron iniciar, desarrollar, y profundizar en el tema durante su administración en la Sección de Computación del Departamento de Ingeniería Eléctrica del CINVESTAV; agradezco sus sugerencias cuando presentamos por primera vez este tópico en el Centro de Investigación en Inteligencia Artificial de Barcelona España.

También debo dar las gracias a: Dr. Raúl Cortés Mateos, que destacó la importancia de lograr objetivos precisos, además me dio el apoyo que necesitaba para la realización del trabajo; Dr. Adolfo Guzmán Arenas, Dr. Marvin Minsky, y Dr. Harold V. McIntosh, que me ayudaron a reforzar la convicción para desarrollar y profundizar en el tema.

Gracias también: Ing. Luis Alcaraz Ugalde, las facilidades que permitieron terminar el proyecto; Ing. Jorge Suárez Díaz, que facilitó tiempo de trabajo y las instalaciones donde surgieron las primeras ideas; M.C. Armando Jiménez Flores, indicó revisiones y cambios que mejoraron el comportamiento de los dispositivos; Ing. Leodegario Wrvieta Cortés, la disposición de sus sistemas de desarrollo; Ing. José de Jesús Negrete Redondo, la revisión final del manuscrito; y Lic. José Urbano Cruz Cedillo, el acceso a sus recursos computacionales.

Muchas gracias al Prof. Ignacio Rubén Pérez Escamilla que invirtió su valioso tiempo en ayudarme a comprender aquellos conceptos matemáticos que *me* permitieron seguir adelante. En este mismo contexto, deseo expresar profundos agradecimientos a todos los maestros que me formaron en la vida, desde el primero que recuerda mi infancia, Prof. Camilo Torres, maestro rural de Sn. José Zoquital Hgo., y que me enseñó los primeros "*palitos*", hasta los que he tenido en estos últimos años, y que me han permitido lograr mis objetivos, sin olvidar para nada a los profesores de música que me ayudaron a comprender su lenguaje. Muchas gracias a todos mis alumnos que participaron programando muchos conceptos sin relación alguna aparente, pero que comprobaron en los laboratorios y los reunieron en un todo congruente.

Agradezco con profunda emoción las facilidades otorgadas por las autoridades de las instituciones adonde he sido alumno, docente e investigador, y donde surgieron aquellas ideas que me permitieron desarrollar esta investigación, resultado de la cual, he dedicado gran parte de mi vida: Escuela Superior de Ingeniería Mecánica y Eléctrica (ESIME) del IPN; Centro de Investigación y de Estudios Avanzados (CINVESTAV) del IPN, principalmente, la Sección de Computación del Departamento de Ingeniería Eléctrica, y la Sección de Proyectos de Ingeniería, del mismo departamento; Centro Nacional de Cálculo (CeNaC) del IPN; Centro de Investigación en Computación (CIC) del IPN; Dirección de Cómputo y Comunicaciones (DCyC) del IPN; y por supuesto, mi *Alma Mater*: Instituto Politécnico Nacional.

Maximino Peña Guerrero

... Levántate, pues, y vence tu flaqueza con el ánimo que triunfa en los combates [...] *... y al levantarse mira alrededor, desvanecido por la grande angustia por* [la] *que ha pasado...*

DANTE ALIGHIERI
(1265-1321)

Índice general

1. Introducción **7**

2. Música Digital **15**

2.1. Composición 16

2.1.1. Creación de una composición 16

2.1.2. Creación de una composición con una máquina 17

2.2. Escritura de una composición 19

2.2.1. Lenguajes digitales orientados a la música 20

2.2.2. Partituras en TEX 22

2.2.3. Transportación del guión a los instrumentos 23

2.3. Interpretación 24

2.3.1. El estándar MIDI 24

2.3.2. Sintetizadores 25

2.3.3. Secuenciadores 26

2.4. Arreglos 27

2.5. Dictado de partituras 29

2.6. Resumen 30

3. Captura de Eventos MIDI en Tiempo de Ejecución: Espacio de Diseño **31**

3.1. Introducción 31

3.2. Funcionamiento conceptual 33

3.3. Diseño con productos de propósito general 33

3.4. Diseño a la medida con componentes de propósito general 35

3.5. Diseño de componentes a la medida 36

3.6. Resumen 38

4. Nuestros diseños de un Sistema de Captura de Múltiple Eventos MIDI **39**
4.1. Sistema MCS-L 40
4.1.1. Hardware 41
4.1.2. Software: captura y almacenamiento en memoria 42
4.2. Sistema MCS-S 43
4.2.1. Memoria Segmentada Autocompactante (MSA): funcionamiento 45
4.2.2. Identificadores de datos y listas 47
4.3. Discusión 49
4.4. Resumen 50

5. Diseño de la Memoria Segmentada Autocompactante **51**
5.1. Terminología 51
5.2. Estados Escritura 54
5.2.1. Estado 1. 55
5.2.2. Estado 2 57
5.2.3. Estado 3 58
5.2.4. Estado 4 59
5.2.5. Estado 5 60
5.2.6. Estado 6 61
5.2.7. Estado 7 63
5.2.8. Estado 8 63
5.2.9. Estado 9 64
5.2.10. Estado 10 65
5.3. Estados de lectura 66
5.3.1. Estado 11 66
5.3.2. Estado 12 68
5.3.3. Estado 13 68
5.3.4. Estado 14 69
5.3.5. Estado 15 70
5.3.6. Estado 16 71
5.4. Resumen 71

6. Evaluación de nuestras propuestas: MCS-L y MCS-S **73**
6.1. Evaluación de MCS-L 73
6.1.1. Prototipo 73
6.1.2. Resultados 79

6.1.3. Otros resultados: nuestro compilador KL 80
6.2. Evaluación de MCS-S 82
6.2.1. Xilinx 82
6.2.2. Simulación de las celdas de control de la MSA 83
6.2.3. Simulación de los componentes restantes de la MSA 87
6.2.4. Resultados 90
6.2.5. Discusión 92
6.3. Resumen 93

7. Conclusiones y Trabajos Futuros **95**
7.1. Limitaciones y trabajos futuros 99

A. El sistema MIDI **101**
A.1. Formato de tiempo de duración de nota *Delta Time* 101
A.2. La interfaz MIDI 102

B. Códigos fuente de componentes VHDL **107**
B.1. Código de MEMSEG 107
B.2. Código de CELCTL. 120
B.3. Código de FMUXSCH. 124
B.4. Código de FMUXM8. 125
B.5. Código de DLATCHA8. 126
B.6. Código de DLATCHB8. 127
B.7. Código de BUF8STAT. 128
B.8. Programa del control de prueba MAQUINA 129
B.9. Código del decodificador de memoria lineal DECODER. 133

C. Notas, tonos y acordes **135**
C.1. Nota musical 135
C.2. La octava MIDI 136
C.3. El tono MIDI 137
C.4. Las escalas MIDI 138
C.4.1. Escala cromática MIDI 138
C.4.2. Escala MIDI en modo mayor 139
C.4.3. Escala MIDI en modo menor 139
C.4.4. Acorde MIDI 140
C.5. Archivo fuente `poli.kl` para la partitura de la figura 6.4 141

Índice de figuras

3.1. Interacción de un sistema de captura en tiempo de ejecución de múltiples eventos MIDI. 33
3.2. Sistema de captura utilizando computadoras personales estándar interconectadas a través de una red local. 34
3.3. Eventos que se producen cuando se toca una nota 36
3.4. Parámetros de una nota completa para su impresión 36

4.1. CMIDIP con memoria lineal MCS-L. 40
4.2. Varias listas FIFO independientes para cada instrumento 42
4.3. Una sola lista FIFO para todos los instrumentos. 43
4.4. CMIDIP con memoria segmentada MCS-S. 44
4.5. Segmentos FIFO de tamaño variable concatenados dentro del espacio de la memoria segmentada; i apunta al inicio de un segmento, y j al final. 44
4.6. Sistema de Memoria Segmentada Autocompactante MSA. 45

5.1. Celdas de memoria y *buses*. 52
5.2. Estados de escritura de las celdas de memoria. 55
5.3. Estados de lectura de las celdas de memoria. 67

6.1. Componentes del CMIDIP con memoria lineal MCS-L. 74
6.2. Prototipos MCS-L: foto izquierda, de izquierda a derecha, MCE8051 y DSP TMS320FC240, abajo HC11; foto derecha: captura de datos MIDI. 75
6.3. Desplegado experimental de datos MIDI. 79
6.4. Salida KL del archivo `poli.kl` compilado (listado: Anexo C.5). . . . 81
6.5. Diagrama esquemático de una celda de control de la MSA. 83
6.6. Diagrama esquemático de una celda de memoria MSA con sus componentes básicos. 88
6.7. Acoplamiento esquemático de MAQUINA y MSA (MEMSEG). 90
6.8. Formas de onda de la operación de MSA (MEMSEG). 91

A.1. Especificación hardware MIDI. 104

A.2. Especificación software MIDI. 105
A.3. Interfaz hardware del MCS con los instrumentos MIDI. 106

B.1. Componente final de memoria segmentada MEMSEG. 119
B.2. Componente memoria segmentada de prueba MEMSEG. 120
B.3. Componente CELCTL: celda de control de memoria. 124
B.4. Componente selector y generador de direcciones FMUXSCH8. 125
B.5. Componente selector de datos de entrada a la memoria FMUXM8. . . 126
B.6. Componente auxiliar de memoria DLATCHA8. 127
B.7. Componente principal de memoria DLATCHB8. 128
B.8. Componente *buffer* de tercer estado BUF8STAT. 129
B.9. Componente MAQUINA que genera los estados de MEMSEG para pruebas. 132
B.10. Decodificador de direcciones DECODER. 133

C.1. Nombres, símbolos, variables, y teclas de un piano (octava 4). 138
C.2. Acorde de DO mayor (C_M). 140
C.3. Partitura generada con KL. 145
C.4. Descripción Backus-Naur Form de KL. 146
C.5. Programa fuente para el compilador KL. 147

Índice de tablas

5.1. Tabla de valores fijos para direcciones y apuntadores. 53
5.2. Tabla de estados y comandos para las celdas de control. 56

A.1. Tablas de formato natural y formato MIDI de *Delta Time*. 102
A.2. Otros estándares y protocolos de comunicación. 103
A.3. Medios físicos de transmisión de datos. 104
A.4. Ancho de banda: Voz, TV y microondas. 104

B.1. Señales del componente final memoria segmentada MEMSEG. 119
B.2. Señales de MEMSEG con ventana de direcciones y datos de prueba. . 120
B.3. Señales de una celda de control CELCTLM. 123
B.4. Estados de FMUXSCH8 para generar direcciones y desplazamientos. . 125
B.5. Estado de FMUXM8 para seleccionar la fuente de datos. 126
B.6. Señales de MAQUINA. 132

C.1. Correspondencia entre: frecuencia (*hertz*) de nota(i), posición en octava(j), y tono. 136
C.2. Relación n entre octavas(j), notas(i) y tonos(n), dado por $n = j(12) + i$.137

Capítulo 1

Introducción

Tal vez la historia de la música electrónica y digital comienza a escribirse con la introducción, a principios de los años 80s, de la computadora personal (PC) y el sistema MIDI (*Musical Instrument Digital Interface*). Este último implementa un protocolo de comunicaciones que permite conectar un instrumento musical a una computadora. Hoy en día existen los sistemas CAC (*Computer Assisted Composer*), poderosas herramientas que ayudan a los músicos (y aun aquellos que no lo son) a realizar una composición musical, interpretarla o estudiarla [45][77].

Cuando un músico hace una composición, primero escribe sobre pentagramas un guión general donde describe con precisión todos los eventos musicales que interpretarán los músicos de una orquesta. Después, un transportista extraerá una partitura para cada instrumento a partir de este guión, las cuales finalmente se interpretan y se tocan. Este proceso de crear, escribir, transportar y ejecutar una composición, puede ser un trabajo difícil y tardado. Pero los sistemas CAC pueden hacer la mayor parte de este trabajo de una manera automática. Su hardware y software permite hacer edición de partituras, escucharlas y corregirlas de una manera interactiva; además, permiten la simulación de una orquesta real que ejecuta dicha composición.

Con todo esto, actualmente los sistemas CAC no permiten realizar la captura precisa de múltiples eventos MIDI en el momento que se generan. Los CAC tienen restricciones en cuanto a la entrada de datos, debido a que el protocolo MIDI, en el

cual se basan, no fue diseñado para capturar datos, sino para la ejecución de música. Sin embargo, la captura de múltiples eventos MIDI, los cuales son generados por muchos instrumentos musicales cuando están tocando, es deseable por varias razones. Por ejemplo, en el aprendizaje musical, un sistema CAC con tal capacidad ayudaría a ejecutantes de música a comparar sus ejecuciones con las de versiones "perfectas". Tales ejecutantes capturarían su ejecución, y generarían las partituras correspondientes y las compararían con aquellas versiones "perfectas".

También haría posible obtener las partituras de improvisaciones, las cuales son ejecuciones únicas e irrepetibles para su posterior estudio. Actualmente, la única forma de obtener las partituras de improvisaciones es grabando dichas improvisaciones y escuchándolas repetidamente para discernir las notas correspondientes, y escribirlas a mano en un pentagrama *"sacando la partitura de un disco"*. Normalmente este proceso es muy difícil hasta para un compositor.

Actualmente la captura de eventos MIDI no es eficiente. Varios eventos MIDI se pueden perder porque los bits que los codifican viajan en serie a través de la interfaz MIDI. Cada evento MIDI corresponde a una nota. El tiempo que tarda una nota, cuando ésta se dispara y empieza a escucharse, es de casi un milisegundo. Pero cuando se tocan acordes (varias notas simultáneas), con un piano o una guitarra, este retardo es más relevante. Por ejemplo, un acorde de diez notas disparadas al mismo tiempo tiene un retardo de casi 10 milisegundos. La situación es aún más crítica si son dos o más instrumentos tocando al mismo tiempo. El ancho de banda del MIDI se saturaría si se producen demasiadas notas en paralelo.

Esta restricción de hardware hace que los sistemas CAC no capturen con suficiente precisión y fidelidad todos los eventos MIDI, lo que se traduciría en partituras que no describirían fielmente la música original. Para capturar los eventos MIDI de varios instrumentos es necesario manejar una interfaz MIDI por cada instrumento, configuración que no es común en computadoras personales. Además, el software capaz de manejar tal configuración tampoco está disponible.

Esta tesis explora el espacio de diseño de sistemas de captura y grabación de múltiples eventos MIDI, los cuales, se producen en tiempo de ejecución musical, y provienen de varias fuentes en paralelo. Se presenta un análisis de las posibles organizaciones de hardware y software para capturar los datos de eventos MIDI que generan uno o varios instrumentos musicales. Se hacen dos diseños específicos completos para lograr la captura de eventos MIDI musicales (MCS-L y MCS-S). Se estudia el diseño, la simulación, y la verificación de una Memoria Segmentada Autocompactante (MSA). Y se presenta el diseño de un compilador (KL) para producir partituras musicales a partir de datos MIDI, o del lenguaje natural de la música (solfeo).

Es posible organizar uno de tales sistemas con computadoras personales completas, con componentes de propósito general o con componentes diseñados a la medida. Se proponen dos diseños específicos: MCS-L (*MIDI Capture System Linear-Memory*) y MCS-S (*MIDI Capture System Segmented-Memory*). MCS-L utiliza un tipo de memoria estándar y es adecuado para desarrollar un prototipo en poco tiempo, pero su fabricación masiva sería relativamente costosa por requerir alrededor de 20 circuitos integrados convencionales. En cambio, MCS-S utiliza un diseño especial de Memoria Segmentada Autocompactante [19], y requiere a lo más 3 circuitos integrados de propósito específico; lo que reduciría su costo en una producción masiva.

Cada interfaz MIDI de MCS-L es un puerto serial que recibe los comandos MIDI de un instrumento. Cada puerto está asociado a un temporizador de tiempo de notas. Cuando una interfaz MIDI recibe un comando de activación de nota ($NoteON$), se dispara un contador que se detiene cuando el puerto recibe el comando ($NoteOff$) que apaga esta misma nota. Tanto los datos MIDI, como los del contador de tiempo de notas, se almacenan en memoria junto con un encabezado, el cual identifica la fuente que los generó. El procesador de MCS-L hace un sondeo para revisar los registros de cada interfaz MIDI y verifica que se haya recibido algún evento. Si éste es el caso, se almacena el dato en la memoria; de lo contrario el procesador continúa su sondeo.

En cambio, en MCS-S no son necesarios los encabezados para identificar las fuentes. Su Memoria Segmentada Autocompactante (MSA) es un diseño especial que maneja internamente varias listas FIFO (*First In First Out*) independientes, una por cada interfaz MIDI. En todo momento las listas están contiguas en memoria, lo que permite ahorrar memoria cuando un instrumento no genera eventos MIDI. También contiene un mecanismo especial que mueve en paralelo, durante un solo ciclo de reloj, todos los datos de todas las listas hacia la derecha cuando se inserta un dato en cualquier lista, o hacia la izquierda cuando se extrae un dato de cualquier lista.

MSA utiliza sólo dos comandos de operación: *inserción* y *extracción* de datos en, o de, una lista, los cuales corresponden a la *escritura* y *lectura* de datos, respectivamente. La escritura se usa sólo en el modo de captura de eventos MIDI, y la lectura se usa sólo para descargar los datos capturados a una PC para su posterior procesamiento. MSA contiene un control general distribuido entre las celdas de memoria que se encarga de enviar, tanto las señales de escritura/lectura, como los apuntadores que indican en que segmento se ha de realizar una operación. MSA utiliza celdas de memoria consecutivas que constituyen un tipo de memoria particionada en segmentos (listas FIFO) donde se almacenan los datos, y cada celda de memoria es controlada por una celda de control asociada que indica cuál operación debería realizar.

Debido a que dentro de la memoria MSA la mayor parte del trabajo de movimiento de datos se hace por hardware y en paralelo, el software de captura sólo coloca el dato y el apuntador de una lista específica sobre sus buses respectivos, el hardware de la memoria hace el resto, pues muchas de las funciones de captura se han delegado al hardware (el software realiza poco trabajo). Todos los movimientos de datos se realizan durante un periodo de reloj, el cual depende de la tecnología vigente impuesta en la fabricación de circuitos integrados [12][14]. Se dispone de software hecho a la medida de MCS-L y MCS-S que nos permite una captura aceptable y un post-procesamiento confiable de partituras.

Para evaluar el diseño de MCS-L implementamos un prototipo con base en el microcontrolador de Intel 8051, y un software de sistema operativo KLM (K*ernel for a smal*L M*onitor*) de control. Las funciones de KLM exploran los puertos de cada interfaz MIDI y extraen los datos de sus registros. Otras almacenan los eventos MIDI en memoria, controlan el reloj interno de tiempo real del prototipo, controlan el manejo de interrupciones del reloj, entre otras funciones. Una vez que se han capturado y almacenado dichos eventos, estos se transfieren a una PC para que sean procesados y así obtener las partituras.

Este post-procesamiento se hace con KL (K*ernel for a music* L*anguage*), un compilador diseñado para traducir eventos MIDI en partituras. Su entrada es un archivo que contiene dichos eventos, o la descripción textual de una melodía como la cadena de sílabas "do do mi sol do do♯ re♭", las cuales describen un pequeño fragmento de música. Acepta instrucciones de control gráfico para indicar otros elementos musicales como: ligaduras, corcheas, silencios, tresillos, y otras que tienen un fin estético. Toda instrucción de entrada será convertida en comandos gráficos para los formatos: PCL (*Printer Command Language*), HPGL (*Hewlett Packard Graphics Language*), PostScript, TeX, o MIDI, con los cuales se imprimirán las partituras.

En cuanto a la evaluación del diseño de MCS-S se hizo una simulación completa de la Memoria Segmentada Autocompactante (MSA), y de cada uno de los componentes internos que constituyen una sola celda de memoria. Dichos componentes constituyen la diferencia entre MCS-S y MCS-L, pues difieren sólo en la memoria donde se almacenan los eventos MIDI de cada instrumento; todos los demás componentes pueden ser iguales en una implementación.

Una descripción funcional en lenguaje VHDL de cada uno de los componentes de la MSA permite evaluar interactivamente el comportamiento de los mismos, y por lo tanto de la memoria en su totalidad [10][3]. Se puede observar que al escribir o leer un dato de la memoria, cada celda de control genera señales internas y externas para que los datos y direcciones de toda la memoria se desplacen una posición a la

derecha o la izquierda, respectivamente. Esto se debe a que la MSA está formada por una sucesión consecutiva de celdas de memoria, cada una de las cuales es controlada por una celda de control. La celda de control es el elemento más complejo de la MSA, y describiendo cómo se comporta una ellas, de acuerdo con una tabla de estados que se ha diseñado ex profeso, se explica la mayor parte del funcionamiento de la MSA.

Nuestros resultados corresponden a la compilación software de VHDL de una memoria MSA con 30 celdas, de las cuales 16 corresponden a las cabeceras de los segmentos, por lo que nuestra simulación sólo maneja 14 localidades para datos, y por lo tanto está limitada. Observamos que este tipo de simulación es muy demandante de los recursos de hardware y software de nuestra computadora de trabajo y sobre la cual hicimos la simulación.

Cabe señalar que compilando código VHDL para 18 celdas de memoria, se lleva un poco más de tres horas, mientras que la compilación de 100 celdas incurría fallas del sistema operativo Windows al punto de bloquear la máquina. Desafortunadamente no teníamos la versión Xilinx para Unix/Linux. Sin embargo, a pesar de las limitaciones de nuestra simulación, nuestros resultados de todos los componentes básicos de una memoria MSA son una prueba de concepto de que dicha memoria es posible, y así mismo nuestro sistema MCS-S basado en la misma.

La organización de la tesis está estructurada de la siguiente manera: El capítulo 2 presenta los principales desarrollos en ciencias de la computación y el procesamiento de música con métodos computacionales, todo lo cual constituye hoy en día los sistemas CAC, los cuales facilitan enormemente la composición de música.

En el capítulo 3 se presenta el espacio de diseño de sistemas de captura múltiple de eventos MIDI de varias fuentes en tiempo de ejecución, referidos como CMIDIP (Captura de eventos MIDI en Paralelo). Se describe la posibilidad de diseñar un sistema CMIDIP, con PCs, con componentes de propósito general, o con componentes especiales y a la medida.

En el capítulo 4 se presenta el diseño específico de los dos sistemas CMIDIP: MCS-L y MCS-S. Ambos son pequeños, portátiles y fáciles de usar. Consideramos que estas son características esenciales para que estos sistemas ganen una aceptación general. MCS-L utiliza componentes comunes y una memoria convencional estándar, mientras que MCS-S utiliza un tipo de memoria especial que lo hace único: la Memoria Segmentada Autocompactante (MSA).

En el capítulo 5 se describe en detalle el diseño de la memoria segmentada autocompactante MSA del sistema MCS-S; aquí se describen los 16 estados en que puede encontrarse cada una de las celdas de la memoria. Se muestra que este diseño puede ser directamente implementado en un circuito integrado ASIC de propósito específico. Para cada estado en que se puede encontrar una celda, antes de una operación de lectura o escritura, se describe su operación, la función lógica que determina su comportamiento, y una matriz que muestra el flujo de datos cuando se realiza la inserción o extracción de un dato específico.

En el capítulo 6 se hace la evaluación de MCS-L y MCS-S, y se muestran los resultados que se obtuvieron en cada uno de ellos. Se describe la evaluación de MCS-L con base en un prototipo que utiliza el microcontrolador Intel 8051. También se describe el software que se desarrolló ex profeso para el mismo: un núcleo de sistema operativo (KLM), y otro (KL) que realiza, después de la captura, la extracción de datos musicales requeridos para procesar la impresión gráfica de las partituras. En la evaluación de MCS-S sólo se considera la simulación de la memoria segmentada autocompactante MSA por ser éste el componente primordialmente diferente en ambos diseños. Mostramos resultados de dicha simulación con base en el medio ambiente de desarrollo de sistemas digitales Xilinx.

Por último, en el capitulo 7 se presentan nuestras conclusiones y las limitaciones de nuestras propuestas, asimismo, se presentan algunas aplicaciones de nuestros resultados, tanto en proyectos inmediatos, como a largo plazo.

Φ

Capítulo 2

Música Digital

En este capítulo se presentan los principales desarrollos en las ciencias de la computación que han enriquecido las diferentes actividades musicales. Hoy en día, muchos de estos avances han sido integrados en los sistemas de *Composición Asistida por Computadora*, o sistemas CAC (*Computer Aided Composer*) [6], los cuales consisten de programas de computadora y de dispositivos de procesamiento de señales digitales orientados al procesamiento de la música y la composición.

Desde que aparecieron las primeras computadoras digitales, el desarrollo de estos sistemas ha sido lento, sin embargo casi de inmediato fue acuñado el concepto como una potente herramienta de ayuda para la creación de música y análisis de partituras [42]. Pero fue cuando aparecieron las primeras computadoras personales (PC´s), que una nueva generación de músicos le ha dado una gran importancia a este nuevo concepto. Primero lo adoptaron los estudios de grabación y luego estos músicos profesionales lo han ido utilizando paulatinamente, a pesar de la resistencia natural de los viejos compositores a componer con medios automáticos [62].

Los conceptos que se presentan en este capítulo son esenciales para el entendimiento de los capítulos siguientes.

2.1. Composición

Una composición musical es la escritura de símbolos gráficos que representan el sonido de una pieza musical sobre un *guión musical*, el cual consiste de renglones de pentagramas (cinco líneas paralelas y equidistantes) y conocido simplemente como *la partitura*. Esta puede ser el resultado de una obra musical compuesta para varias voces e instrumentos. El término composición se refiere a "... la parte de la música que enseña las reglas para la formación del canto y del acompañamiento"[87], y también al "... arte de combinar varias partes y elementos para formar un todo unificado"[104]. Nosotros utilizaremos el término composición en el primer sentido mencionado.

Muchos compositores necesitan un instrumento musical como herramienta para componer. El piano y la guitarra, por ejemplo, son instrumentos polifónicos (muchas voces) con los cuales es posible tocar al mismo tiempo una melodía y su acompañamiento. Pero aunque sólo se toque una nota a la vez, lo que hace más lento el proceso de componer, también es posible utilizar un instrumento monofónico (una sola voz) como una trompeta o un saxofón. De cualquier forma se ha observado que el proceso de hacer una composición incluye cinco etapas: *creación*, *escritura*, *transportación*, *interpretación*, y *retroalimentación* (evaluación) [68][69].

2.1.1. Creación de una composición

¡Y surge la idea de una composición! Es el resultado de una capacidad humana muy difícil de definir debido a la gran complejidad inherente. Simplemente se manifiesta. Tal vez los únicos factores que la caracterizan sea una emoción intensa y un poco de conocimiento y ejecución musical. Sin embargo, es una actividad con una parte puramente creativa y otra meramente mecánica o de rutina. La primera es el arte de combinar sonidos y tiempos; la segunda involucra cálculos matemáticos con los objetos musicales de una composición [27]. Generalmente un compositor conoce de teoría musical.

Sin embargo, hay quienes no han recibido esta formación, pero tienen una habilidad natural para hacerlo. Pueden "escuchar"en su mente la música de una nueva composición, y sólo deben escribirla de inmediato para no perder los detalles [35].

2.1.2. Creación de una composición con una máquina

Ya desde tiempos remotos se hicieron muchos intentos para lograr que una máquina pudiera hacer una composición. Pero no fue hasta que aparecieron las primeras computadoras digitales que estas ideas tuvieron una mayor consideración. Hoy en día la creación de una composición musical forma parte de una de las ramas de las ciencias de la computación: la Inteligencia Artificial o AI (*Artificial Intelligence*) [17][33].

A este respecto, Ada Augusta (1815-1852), considerada como la primera programadora de computadoras, y refiriéndose a la máquina analítica de C. Babbage (1821-1894), dijo: "...[si] las relaciones fundamentales de los sonidos emitidos en la ciencia de la armonía y la composición musical fueran susceptibles de [implementarse con...] expresiones [de lenguaje natural] y adaptaciones [mecánicas], la máquina debería elaborar composiciones y fragmentos científicos de música de cualquier grado de complejidad y extensión"[90].

A pesar de que dicha máquina no se pudo construir en su momento, la idea de Ada fue reconsiderada cuando aparecieron las primeras computadoras digitales. Una de ellas, la ILLIAC, fue programada por Hiller e Isaacson para crear una composición que llamaron *Illiac Suite*, la cual, ha sido considerada como la primera composición hecha por una máquina de una manera automática [37]. A partir de este momento el sueño de Ada Augusta poco a poco se iría haciendo una realidad.

Pensamos que este acontecimiento histórico no habría sido posible si J. S. Bach (1685-1750) no hubiera hecho una sistematización formal de la música y la hubiera puesto en práctica en su obra el *Clavesin Bien Temperado*. El maestro de la fuga utilizó los logaritmos de J. Napier (1550-1617) para dividir una *octava*, o intervalo en-

tre un tono y el mismo tono pero con doble frecuencia de vibración, en 12 semitonos de acuerdo con la ecuación $f_n = f_{ref}(2^{\frac{n}{12}})$, donde: f es la frecuencia de vibración de uno de esos 12 semitonos; f_{ref} es una frecuencia de referencia (p.e. $\text{LA}_4 = 440$ hz); n $(1 - 12)$ es el número de semitono a partir de dicha referencia. En consecuencia, cada nota del instrumento estará separada en vibraciones por segundo por una constante numérica igual a $\sqrt[12]{2} = 1{,}059\ldots$. Esta característica nos permite realizar fácilmente cálculos numéricos entre los símbolos que conforman el lenguaje de la música [65].

Hoy en día es posible crear una composición con medios artificiales: *agentes* y *redes neuronales* [7][103]. Un agente es un proceso dedicado a resolver un problema único y además puede convivir en sociedad con otros agentes a través de una red de interconexiones. De acuerdo con Minsky, un agente debería reconocer el compás de una melodía (*measure-takers*), otro podría construir una estructura musical (*structure-builders*), y otro más sería capaz de reconocer diferencias dentro de una estructura melódica (*difference-finders*) [60].

Por otro lado, una red neuronal es un arreglo masivamente paralelo de unidades simples llamadas neuronas que sirven para modelar algunas de las funciones que tiene el sistema nervioso [18][35]. Surgió de un movimiento filosófico llamado *coneccionismo*, el cual constituye un enfoque computacional que sirve para modelar el cerebro conectando muchas unidades simples y producir un comportamiento complejo. Una de las aplicaciones más relevantes de este comportamiento es una memoria de red neuronal llamada SRN (*Simple Recurrent Network*) que sirvió para construir algunas frases musicales aprendidas durante un entrenamiento previo. También sirvió para modelar los aspectos psicológicos que tienen que ver con la representación del tono, la duración de las notas, y algunas estructuras armónicas utilizadas durante una composición. Fue desarrollada a principios de la década de los 90s por Michael C. Mozer modificando una red coneccionista llamada CONCERT [35].

Otra red importante fue EBM (*Effective Boltzmann Machine*) que sirvió para armonizar automáticamente una pieza musical durante su evolución. Matthew I. Bell-

gard y C. P. Tsang desarrollaron y entrenaron esta red con las reglas sintácticas de la música de acuerdo con el modelo probabilístico de una distribución Boltsmann. Y para cuerdas, para hacer el acompañamiento de una melodía con una guitarra en tiempo real, Garrison W. Cottrell construyó una red llamada CAG (*Connectionist Air Guitar*), la cual aprende a adivinar las siguientes tonalidades que vienen en una melodía. Fue desarrollada como una representación conceptual de un instrumento de cuerdas tocando en sincronía con la música.

2.2. Escritura de una composición

Una vez que se ha desarrollado el tema y los detalles de una composición, lo que sigue es poner los símbolos correspondientes sobre pentagramas para la partitura completa.

Mucho antes de que apareciera la primera partitura no había forma de que el sonido de la música quedara registrado de una manera permanente. Este problema quedaría resuelto cuando un monje benedictino, llamado Guido D'Arezzo (*ca.* 1000 d.C.), inventa el pentagrama y da nombre a cada una de las notas musicales (*do, re, mi, fa, sol, la, si*) tomando las primeras sílabas de un fragmento de S. Juan: "*...ut queant laxis resonare fibris mira gestorum fa muti tuorum solye poluti labii reatum sancte Ioannes* [el subrallado es nuestro]"[65, p:472]. Desde entonces el pentagrama y la partitura se han convertido en el principal medio de comunicación entre los músicos.

Sin embargo, a esta partitura solamente podían tener acceso unos cuantos privilegiados. Si alguien más la quería, ésta tendría que copiarse a mano utilizando pluma, tinta y papel. Pero, desde que Gutenberg (1400-1468) pudo imprimir la primera partitura, la música ya no sería reproducida por monjes letrados dentro de claustros aislados, sino impresa con mecanismos cada vez más automatizados y sofisticados, hasta llegar a los que se disponen ahora. Hoy en día los sistemas CAC facilitan la edición de partituras, ahorrando mucho tiempo a través de una estrecha interacción compositor-máquina.

2.2.1. Lenguajes digitales orientados a la música

La edición de una partitura con computadora, al igual que otros procesos sobre la misma, por ejemplo, su ejecución, requiere la traducción del lenguaje musical a un lenguaje de números que pueda manejar la computadora [13][9]. Ya en la antigüedad se hizo la primera traducción [54]. Fue Pitágoras (580?-497 a. C.), matemático y filósofo griego, creador del teorema que lleva su nombre, quien estudió las cuerdas vibrantes y descubrió un método para afinar una escala (del latín *scala*, "escalera") musical en las cuerdas mismas. A la primera nota le asignó un valor de frecuencia arbitrario, el número total de vibraciones por unidad de tiempo, a la siguiente nota le asignó una frecuencia igual al valor de la anterior (primera) pero multiplicado por 3/2, y así sucesivamente, hasta completar una cadena de notas separadas por quintas perfectas.

Pero era necesario desarrollar un sistema mucho más preciso que el del filósofo para describir con números el mundo complejo de los sonidos. Era preciso hacer un análisis formal de los componentes de onda, armónicos, que contiene una nota musical o cualquier otro fenómeno natural que se encuentre en estado de vibración. Este análisis fue realizado por J. Fourier (1768-1830), lo cual dio como resultando la transformada que lleva su nombre, y que sirve para cuantificar el contenido armónico de una onda compuesta, como es la de cualquier sonido [2].

Además, era también necesario estudiar el proceso psicológico que existe cuando el cerebro escucha la música. A través de las investigaciones sobre fisiología acústica que hizo H. L. F. Helmoltz (1821-1894) se establecieron los fundamentos científicos de la teoría física, y se tendió un puente formal entre la ciencia y la música [36]. Sin embargo, hasta la fecha tales acontecimientos no han bastado para comprender como funciona el entendimiento humano. A este respecto Alan M. Turing (1912-1954) dijo: "Al tratar de construir estas máquinas no estaríamos usurpando irreverentemente [Su] poder de crear almas, como tampoco lo hacemos al procrear niños: en cada caso somos más bien instrumentos de [Su] voluntad, al proporcionar mansiones para

las almas que Dios crea” [102].

Durante la construcción de las primeras computadoras digitales, John von Neumann (1903-1957) desarrollaría la teoría de autómatas complejos que permitió diseñar los primeros lenguajes de programación de computadoras [67]. Estos lenguajes utilizarían palabras de lenguaje natural que indican una acción específica a ser realizada por una computadora. Estas palabras y otros símbolos se traducen entonces al lenguaje específico que entiende una máquina [32][85].

Poco después se empezaron a escribir los primeros lenguajes orientados al procesamiento de sonidos musicales. Uno de los pioneros, llamado DARMS (*Digital Alternate Representation of Musical Scores*), se desarrolló para realizar la primera notación digital de pentagramas. Fue escrito por Stefan Bauer-Mengelberg, y por un tiempo fue el paradigma o modelo a seguir en este tipo de lenguajes. Para poder ser utilizado en computadoras personales, las cuales estaban limitadas en su capacidad de almacenamiento, se desarrollaron varios de sus dialectos: “Note-Processor DARMS”de J. S. Dydo, “A-R DARMS”de T. Hall, y “COMUS DARMS”de J. Dunn [92].

Posteriormente, con base en las características de FORTRAN (*Formula Translate*) como lenguaje científico orientado al procesamiento de fórmulas matemáticas, Michel Kassler escribió su lenguaje MIR (*Music Information Retrieval*) para estudiar las primeras representaciones digitales de la música. Sus instrucciones son una colección de macros de otro lenguaje llamado FAP (*FORTRAN Assembly Program*) que se había escrito para la máquina IBM-7094. MIR fue parte de un proyecto piloto que ayudó a responder algunas de las preguntas hechas por los musicólogos, las cuales estaban relacionadas con el análisis de ciertas frases musicales de la partitura *Misas de Josquin des Prez* [47].

En 1986, y aprovechando la programación orientada a objetos, Miller Puckette escribió el lenguaje MAX para controlar con objetos gráficos una complicada síntesis de sonido (sintetizador 4X), y también para controlar una estación de procesamiento de señales digitales con MIDI llamada ISPW (*IRCAM Signal Processing Worksta-*

tion). MAX sigue siendo útil para desarrollar sistemas interactivos computacionales de procesamiento musical en tiempo real [104].

2.2.2. Partituras en TEX

A mediados de la década de los 70's, Donald E. Knuth desarrolló el programa TEX para ayudar a preparar documentos científicos de alta calidad tipográfica [48]. También escribió un programa llamado METAFONT con base en ecuaciones cúbicas para dar forma gráfica a los tipos de caracteres (*fonts*), tal como se hace con los tipos de plomo de una imprenta, pero ahora hechos con software. El maestro fusionó ingeniería, ciencia, arte y tecnología con la palabra inglesa "*technology*"cuyas primeras letras forman el vocablo griego $\tau\epsilon\chi$ (*tau epsilon chi*) que significa "arte", pero que en mayúsculas son los caracteres de TEX [48].

Con base en el estudio y análisis de este compilador de tipografía, se fueron creando los primeros sistemas de edición e impresión de partituras. El primero de ellos, llamado MuTEX, fue construido por Steinbach y Schofer (1988) con macros de instrucciones de TeX. Pero estaba limitado en cuanto al conjunto de símbolos musicales que podía manejar. Sin embargo, estableció el principio básico que permitiría la construcción de sistemas de tipografía de impresión musical [28].

Tratando de resolver sus limitaciones, Daniel Taupin escribió una versión mejorada que llamó MusicTEX, la cual permitía especificar la polifonía (partituras con varios instrumentos), y tenía la ventaja de ser compatible con LATEX de Leslie Lamport [100]. No obstante que estaba limitado en cuanto a la transportación de notas, cambios de tonalidad, ligaduras, y tresillos, ya era considerado una buena herramienta de edición de partituras. Sus limitaciones se resolvieron con la siguiente versión que Taupin, junto con Mitchell y Egler, llamaron MusiXTEX, que además permite la escritura de tablaturas de guitarra [34].

Otros desarrollos relacionados con TEX incluyen MIDI2TEX y nuestro compilador KL. MIDI2TEX, de Kuykens y Verbruggen, convierte un archivo MIDI (Mu-

sical Instrument Digital Interface) del tipo 1 (con muchas pistas) a instrucciones de MusicTEX para obtener una partitura gráfica. Pero su procesamiento es muy elemental, no genera partituras estéticamente legibles: omite símbolos como ligaduras, puntos de extensión, entre otros.

Es preciso señalar que, con base en las ideas del profesor Knuth, hemos diseñado un compilador, que llamamos KL (*Kernel for music Language*), ex profeso para traducir el solfeo de la música como lenguaje natural (`do`, `mi`, `sol`, `la`$\sharp$, `re`$\flat$, etc.) a: sonido musical, comandos TEX, PostScript, MIDI, símbolos musicales en pantalla, o bien, comandos de impresión estándar como PCL (*Printer Command Language*) o HPGL (*Hewlett-Packard Graphics Language*) [38][39]. KL está formado por un núcleo (*kernel*) escrito en lenguaje C [78].

2.2.3. Transportación del guión a los instrumentos

Después que el compositor ha escrito los detalles de una composición, lo que sigue es extraer del guión musical una partitura independiente para cada instrumento que participa en la composición. Las notas originales deben transportarse (transferirse) hacia otras escalas, debido a que muchos instrumentos tienen tonalidades diferentes a la escala del instrumento que se utilizó para componer. Normalmente este trabajo lo hace un auxiliar llamado *transportista*.

La transportación de tonos requiere de varias operaciones aritméticas simples entre las notas musicales [79]. Consiste en mover la tonalidad, hacerla más aguda o más grave, de cada una de las notas en una partitura en la misma proporción. La transportación es relativamente sencilla de explicar en relación con un piano.

Imaginemos que cada tecla está marcada con un número entero, y los números son consecutivos y ascendentes de izquierda a derecha, por ejemplo: 0, 1, 2, ..., hasta el 87. Si la primera nota de una composición empieza en la tecla 50, entonces, para transportar la composición a la tecla 60, sólo hay que sumar 10 a todas las notas musicales. Como se observa, la transportación es una operación puramente mecánica

que una computadora puede hacer mucho mejor y más rápido que aquellos músicos de carne y hueso.

2.3. Interpretación

Una vez que se obtienen las partituras de una composición para cada instrumento, cada partitura será interpretada por los músicos de carne y hueso de una orquesta en vivo, o bien, por los "músicos digitales" de una *orquesta virtual*. Hoy día los sistemas CAC permiten simular el trabajo de los músicos de una orquesta real, lo cual es particularmente útil, porque se reducen los costos de producción musical, y además, se reduce el tiempo que tarda el compositor en hacer su trabajo. Escuchando la música, nuestro compositor corrige las partituras hasta que queda satisfecho. Si utiliza una orquesta real, las partituras deberían modificarse a mano de manera que puedan tocarse nuevamente, pero si el compositor utiliza una orquesta virtual, esas pequeñas imperfecciones pueden corregirse interactivamente de forma inmediata hasta que se cumplan los resultados musicales que se esperan.

A continuación describimos las investigaciones y desarrollos computacionales que dieron como resultado la funcionalidad de las orquestas virtuales en los sistemas CAC.

2.3.1. El estándar MIDI

Los primeros instrumentos musicales electrónicos no podían intercambiar datos entre ellos debido a que eran construidos con base en la electrónica analógica [66]. Si bien, cuando aparecieron los primeros microprocesadores casi de inmediato se fueron integrando a dichos instrumentos, tampoco existía una forma de comunicación que permitiera su interconexión [93]. Sin embargo, en 1981 D. Smith y C. Wood desarrollaron un protocolo para instrumentos electrónicos digitales llamado USI (*Universal Synthesizer Interface*), el cual propusieron a la *Audio Engineering Society* como un estándar industrial, pero éste no fue aceptado tan fácilmente.

No obstante, los fabricantes de instrumentos musicales acordaron un nuevo protocolo estándar de comunicaciones con base en el USI a fin de conectar hardware y software entre los sistemas de música electrónica. Este protocolo llamado MIDI (*Musical Instrument Digital Interface*) es un lenguaje digital de comandos que sirve para interconectar instrumentos musicales: entre sí, con computadoras digitales, y con otros dispositivos como son los sintetizadores y los secuenciadores [64].

Fabricantes como *Roland* y *Secuential Circuits* acordaron definir en 1983 la especificación OSI MIDI 1.0. de software y hardware [4]. La parte de software es un lenguaje (Figura A.2) programado para convertir acciones musicales en: código binario digital, lenguajes de comandos, o formatos de archivos reconocidos por los sistemas operativos actuales [11].

Por otro lado, la parte de hardware (Figura A.1) consiste de conectores DIN de 5 terminales eléctricamente conectados a un circuito integrado UART de comunicaciones (6850), el cual es programado a una velocidad de 32 250 bauds. El receptor de datos (MIDI IN) se encuentra aislado eléctricamente a través de un optoacoplador (PC-900). De la misma forma, el transmisor de datos (MIDI OUT) también se encuentra aislado a través de dos inversores ($buffers$). Además se hace una copia de datos del receptor para que sean retransmitidos hacia otros instrumentos (MIDI TRUE). Esto permite configurar una red tipo *daisy chain* entre los instrumentos musicales.

2.3.2. Sintetizadores

Aparecido a mediados de los 60, el *sintetizador* es un dispositivo electrónico digital que puede ser utilizado como un instrumento musical [63]. Contiene muchos generadores de funciones periódicas, los cuales crean formas de ondas complejas que sintetizan las características particulares del sonido de un instrumento musical. Sus osciladores electrónicos pueden cambiar la frecuencia, la amplitud, el timbre, y el envolvente de dichas ondas, a través de la activación/desactivación de un sistema de interruptores (teclas de piano) y potenciómetros. Robert Moog construyó el primer

sintetizador analógico que aún lleva su nombre, el clásico MOOG.

Hoy en día los sintetizadores son digitales y algunos utilizan muestras reales de un sonido natural grabado en memoria para tocar una nota (Korg M1). Algunos que se conocen como *generadores de tono* no contienen teclas, pero sí tienen las conexiones MIDI y normalmente son externos a una computadora (Yamaha MU50). Otro tipo de sintetizadores son las tarjetas de sonido que vienen con las computadoras personales (SB AWE32); utilizan circuitos integrados de alta escala (YM3810) para hacer la síntesis y tienen por lo menos un circuito de comunicación serial (6850), el cual permite transmitir y/o recibir mensajes MIDI con otros sintetizadores o dispositivos MIDI externos. Finalmente, otros más reciben el nombre de *sintetizadores software*, programas de computadora que tienen programado los algoritmos que hacen la síntesis de sonido y que controlan uno o varios convertidores digital-analógico [49].

2.3.3. Secuenciadores

Cuando se hace una grabación profesional es necesario contar con herramientas de hardware y software que permitan controlar: el sistema de grabación, las diversas fuentes que producen el sonido, y algunas otras actividades que intervienen en el proceso de registrar el sonido. Una de ellas es el *secuenciador hardware*, dispositivo analógico digital que sirve para controlar la edición del sonido de una grabadora de cinta multicanal. Esta edición consiste en insertar, modificar, o borrar la información secuencial que han producido los instrumentos musicales, inclusive la voz.

Sin embargo, gradualmente los secuenciadores de hardware han quedado obsoletos y se han ido sustituyendo por programas computacionales que reciben el nombre de *secuenciadores software* [16]. Ellos contienen algoritmos programados que permiten simular las funciones que antes eran realizadas por los secuenciadores hechos con dispositivos originales de hardware. Actualmente un secuenciador por software permite editar, borrar, insertar, grabar, transmitir y recibir de manera secuencial, los mensajes musicales en un formato puramente digital [29]. Esto quiere decir, por ejemplo,

que una secuencia de dígitos binarios puede representar una melodía musical, un reloj en tiempo real, o la conversión analógica-digital de la voz de un cantante [40][89].

2.4. Arreglos

Una composición musical se hace pensando en aprovechar las características acústicas de los instrumentos y los cantantes. Sin embargo, a veces es necesario hacer adaptaciones, o *arreglos,* a una composición hacia otras formas de interpretación, ya sea porque aparecen nuevos instrumentos, o se tienen pocos y no son los originales, o simplemente porque se quiere saber cómo sonaría de otra manera. De esta forma, un *arreglo* es una partitura escrita por un músico *arreglista* con base en una composición ya hecha. Su trabajo es similar al del transportista, pero requiere de mayor creatividad, pues debe hacer adaptaciones a instrumentos que generalmente no son los originales.

A veces un arreglista no cuenta con las partituras originales que le permitan realizar su trabajo. Pero en algunos casos, él sí puede escuchar el sonido que produce un amplificador de sonido que se encuentra reproduciendo la música original. Para escribir la música que escucha, él debe utilizar todas sus habilidades para reconocer el sonido de cualquier instrumento, sin más ayuda que la de su propia experiencia. A esta actividad se conoce popularmente como *"sacar la partitura del disco"*, la cual es muy reconocida debido a la pérdida accidental de muchas de las partituras originales. Sacar la partitura de un disco es una actividad muy compleja que aún no se ha podido automatizar debido a la gran cantidad de frecuencias armónicas que contiene el sonido de la música.

Sin embargo, algunos investigadores han logrado reconocer de manera automática patrones de sonido de distintos instrumentos musicales y de voz, utilizando técnicas de Inteligencia Artificial [86]. De acuerdo con las leyes físicas de Fourier, un sonido musical contiene una verdadera mezcla de todos los instrumentos y voces humanas involucrados, los cuales son muy difíciles de identificar con una máquina. Pero,

con base en la teoría de la resonancia adaptiva, la cual describe un proceso biológico humano, y utilizando la información tonal que permite separar y rastrear muchos sonidos distintos, se pudieron construir dos modelos de redes neuronales que permitieron identificar algunos de dichos sonidos.

Con el primer modelo desarrollado por S. Grossberg, es posible escuchar y reconocer una melodía con muchos sonidos superpuestos y producidos dentro de un recinto. Con el segundo modelo (ARTMAP) fue posible reconocer una nota y la tonalidad en que ésta se encuentra. Su creador, Nial Griffithi, dice que dicho proceso de reconocimiento se puede modelar parcialmente con mecanismos rastreadores de intervalos y tonos, dentro del contexto de una melodía. Sin embargo, estos mecanismos, al igual que los agentes de Minsky, requieren de un procesamiento masivamente paralelo.

Para implementar estos modelos neuronales es necesario utilizar sistemas digitales en paralelo que permitan la simulación de los procesos neuronales [41][52]. Las investigaciones del paralelismo intrínseco en dichos modelos, y de la misma forma en las actividades musicales, se vieron fortalecidas con la llegada de los *transputers* a finales de los 80s, los cuales podían interconectarse con líneas de comunicación punto a punto. Desafortunadamente, los transputers no prosperaron, en poco más de un año, casi de inmediato quedaron obsoletos, debido a la llegada de las redes de área local que eran de propósito general, y que se empezaron a utilizar en los prototipos Mockinbird de Xerox y el de Neil Gershenfeld [57][30].

No obstante, se pudo construir un órgano de 88 voces con 160 transputers T800 INMOS controlados con MIDI. Cada transputer simulaba 8 osciladores, los cuales estaban programados como generadores de audio de acuerdo con lo que marca la síntesis aditiva de Fourier. Sus creadores, Itagaki, Manning, y Purvis, empezaron a construirlo en 1987 y tardaron diez años en terminarlo [44]. También se construyó un sistema para hacer improvisaciones de jazz, pero utilizando transputers T-805. Para este sistema, Bruce y Pennycook escribieron un software de medio ambiente paralelo, llamado T-MAX, utilizando el lenguaje CYPHER de R. Rowe [88].

2.5. Dictado de partituras

Con una grabación analógica (o su conversión a digital) el sonido de todos los instrumentos se encuentran mezclados en uno o varios canales de audio. Esto hace difícil, si no imposible, la discriminación musical de cada instrumento para poder obtener sus partituras de una manera automática. Pero, con la llegada de los sistemas MIDI, ahora es posible realizar el *dictado de una partitura* tocando un instrumento MIDI, o bien, se puede obtener la partitura de una ejecución grabada que fue tocada con un instrumento MIDI. Esto es posible porque, como vimos anteriormente, los comandos MIDI identifican sin ambigüedad cada nota producida por un instrumento. Sin embargo, la mayoría de los sistemas CAC ofrecen tales funciones de una manera muy restringida.

La velocidad de grabación de música MIDI permitida actualmente es relativamente baja, 32 250 bits/seg. A esta velocidad, es necesario manejar un retardo entre nota y nota de 1 ms, debido a que la comunicación es serial. Cuando se tocan varias notas en paralelo, formando un acorde, este retardo provoca que se pierdan algunas notas durante la captura, por lo que la grabación no es fiel. Cuando se toca un acorde de doce notas se introduce un retardo de diez milisegundos (un retardo entre notas consecutivas). Si además se suma el flujo continuo de datos que generan algunos controles especiales como la rueda y el vibrato, entonces este retardo es mucho mayor, ocasionando problemas en los algoritmos de sincronización y de ejecución musical.

Algunos investigadores han mencionado que "... el scherzo *Cuarteto para Piano No. 8* de Schubert requiere movimientos repetitivos de mano y de dedos de aproximadamente 8 Hz."[51]. Esta cifra se aproxima a la velocidad máxima que han logrado algunos pianistas profesionales. También se ha observado que un músico es incapaz de efectuar movimientos con más de once pulsaciones por segundo. MIDI se ha diseñado pensando en satisfacer esta velocidad. Actualmente se está investigando el uso de otros protocolos de comunicación para MIDI.

2.6. Resumen

Hemos presentado los principales desarrollos computacionales que han enriquecido las diferentes actividades musicales, y que hoy día se han integrado en sistemas que reciben el nombre de *Composición Asistida por Computadora*, o simplemente CAC. Estos sistemas facilitan enormemente la composición musical, la escritura de la música, la interpretación de una composición y los arreglos musicales.

El diseño de sistemas CAC incluye técnicas de varias áreas computacionales. De Inteligencia Artificial como son las redes neuronales y los agentes, cada uno de los cuales podrían facilitar la producción de arreglos musicales y la generación automática de música. De reconocimiento de patrones, lo cual haría posible el reconocimiento de un sonido específico que se produce dentro de un auditorio con mucho ruido. Del paralelismo que permite simular muchos eventos simultáneos que se generan cuando se realiza una producción y ejecución con muchos instrumentos tocando en paralelo. De los protocolos de comunicación, los cuales permiten conectar en red varios instrumentos MIDI, ya sea para generar música o para interpretarla. Y por último, de los sistema gráficos, con los cuales es posible crear partituras impresas con una calidad profesional, o simplemente visualizar estas partituras en la pantalla.

Capítulo 3

Captura de Eventos MIDI en Tiempo de Ejecución: Espacio de Diseño

3.1. Introducción

Hoy día los sistemas CAC (*Computer Assisted Composer*) son herramientas indispensables en la mayoría de las actividades musicales [8][94]. La composición, la escritura e incluso la interpretación musical son mucho más fáciles y creativas gracias a estos sistemas. Tal vez su única función que actualmente está limitada sea la captura de eventos MIDI en paralelo y en tiempo de ejecución, ya que, como se mencionó en el capítulo anterior, el protocolo de comunicación está limitado.

Esta limitación se debe a que el protocolo no fue diseñado para captura, sino para ejecución. La ejecución de un evento MIDI puede ser múltiple, involucrando varios instrumentos/dispositivos, simplemente por medio de una conexión común de la fuente MIDI a los varios instrumentos/dispositivos. Más el proceso inverso actualmente no es posible con suficiente fidelidad. No es posible que los eventos MIDI producidos por varios instrumentos/dispositivos sean recibidos/capturados simultáneamente, para su posterior reproducción, por ejemplo.

La captura eficaz de múltiples eventos MIDI mientras ocurren, de una o más fuentes, es deseable porque ofrece varias ventajas. Facilitaría la separación de la música de cada instrumento, para *arreglarla* posteriormente, posiblemente con otro sonido distinto al original. Haría posible la extracción precisa y automática de las

partituras de cada instrumento. Particularmente, permitiría contar con las partituras de improvisaciones musicales, las cuales son una manifestación única e irrepetible. La improvisación musical es tal vez la actividad musical más enriquecedora para un músico, y contar con las partituras de improvisaciones haría posible un análisis de las mismas para facilitar su aprendizaje. Algunos investigadores han analizado partituras de las fugas de Bach y han encontrado algunas representaciones de autómatas celulares [46]. Sin embargo, el análisis formal de improvisaciones es un campo virgen de investigación.

En el capítulo anterior vimos que es posible obtener las partituras de una composición a partir de una grabación analógica, por el método de *sacar las partituras del disco*. Pero esto es un trabajo que muy pocos músicos pueden hacer y está sujeto a errores. Actualmente se está trabajando en obtener las partituras de múltiples grabaciones digitales, una por cada instrumento [59]. Este enfoque se basa en el procesamiento de señales digitales, filtrando las frecuencias que corresponden a cada tono. Este enfoque tiene la desventaja de requerir mucho espacio en disco. Requiere un poco más de cinco millones de datos de ocho bits para guardar un solo minuto de audio digital, y si son muchos los canales, esta cantidad será considerablemente mucho mayor [84].

La captura eficaz de eventos MIDI no requiere de tanto espacio [97]. Los archivos que se generarían serían relativamente pequeños (menos del 80 % de su contraparte de conversión digital), pues cada comando MIDI especifica, además de una nota y su intensidad, la duración de la misma. Para garantizar la precisión de la captura, se pueden utilizar otros protocolos de comunicación con mayor ancho de banda que el del estándar MIDI, como el protocolo FireWire IEEE-1394b, el cual permite una relación de transferencia de poco más de tres mil millones de bits por segundo, comparado con las decenas de miles del MIDI [5].

Por estas razones y las anteriormente dadas, los sistemas de captura de múltiples eventos MIDI en tiempo de ejecución, de una o más fuentes, son deseables. En

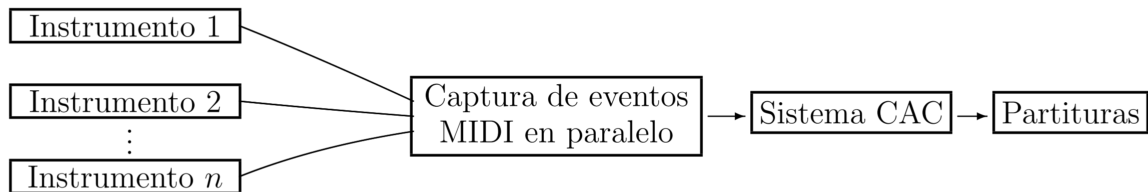


Figura 3.1: Interacción de un sistema de captura en tiempo de ejecución de múltiples eventos MIDI.

lo que sigue de este capítulo presentamos el espacio de diseño de tales sistemas.

3.2. Funcionamiento conceptual

La Figura 3.1 muestra el funcionamiento conceptual de un sistema de captura de eventos MIDI en tiempo de ejecución de múltiples fuentes. Dado que la captura de dos o más fuentes diferentes puede ocurrir en paralelo, nos referiremos a estos sistemas como CMIDIP, de: *Captura de eventos MIDI en Paralelo.*

Los instrumentos musicales (u otros dispositivos como sintetizadores) se conectan al sistema de captura a través de un cable cuyos extremos terminan ambos en una interfaz MIDI. El sistema de captura tendrá una cierta capacidad de almacenamiento. Una vez terminada la captura, el post-procesamiento de la información capturada, por ejemplo, para generar las partituras correspondientes, se realizará fuera de línea, y se hará a través de un sistema CAC.

3.3. Diseño con productos de propósito general

La manera más rápida y económica de construir un sistema CMIDIP es en base a un diseño con componentes de propósito general. Las computadoras personales (PCs) son, por su popularidad, obviamente más fáciles de usar y económicas, además ya existen sistemas CAC para ellas bajo los sistemas operativos Windows, Unix y Linux [58] [83] [101]. Así, una PC puede estar a cargo de la captura de los eventos MIDI generados por un instrumento. La captura de los eventos MIDI de varios instrumentos utilizaría tantas PCs como instrumentos se encuentren involucrados.

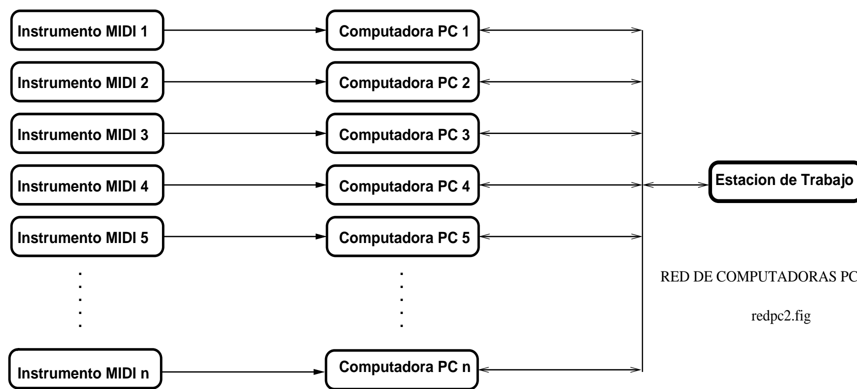


Figura 3.2: Sistema de captura utilizando computadoras personales estándar interconectadas a través de una red local.

Después de la grabación, es posible realizar post-procesamiento de los eventos MIDI de un instrumento individualmente. Pero si se desea realizar post-procesamiento conjunto de los eventos MIDI de varios instrumentos (por ejemplo, su interpretación por una orquesta virtual), es necesario coordinar el principio de la grabación de cada instrumento. Con los sistemas CAC actuales, como ProTools, Finale, MusicXML (dialecto de XML que describe notación musical) se pueden realizar estas y otras tareas de edición musical [50][96].

Esta sincronización posterior con base en un sistema CAC, se puede evitar si las PCs se interconectan a través de una red local y se maneja un reloj global de sincronización [91]. Para implementar este reloj es necesario un programa que correrá en todos las PCs; y una copia del mismo enviará a las otras copias, en las otras PCs, un mensaje global (broadcast) indicándoles el inicio de la grabación. Obviamente, el costo del sistema se incrementará por la red y el programa de captura. La Figura 3.2 muestra esta configuración.

3.4. Diseño a la medida con componentes de propósito general

Un componente de software es un programa diseñado específicamente para realizar una acción predeterminada como puede ser, por ejemplo, el control de la recepción de los datos MIDI dentro de uno de los circuitos UART (*Unasynchronous Asynchronous Reciver Transmitter*) de comunicaciones, o también una función que calcula el tiempo que tarda una nota en producir sonido. Por otro lado, un componente de hardware puede ser cualquiera de los circuitos integrados que contiene nuestro CMIDIP, el motherboard de una computadora PC, o el mismo procesador.

Aunque el diseño con computadoras personales puede desarrollarse rápida y económicamente, es impráctico [75]. Las PCs deben transportarse, conectarse y configurarse cada vez que se realice una grabación. Su conexión y configuración en particular, requiere de conocimientos de computación elementales [74][76]. Este CMIDIP es adecuado si su transportación no es frecuente, por ejemplo, en estudios de grabación.

Para añadir flexibilidad al diseño anterior basado en una interconexión de red, se puede utilizar sólo un monitor y un teclado por todas las PCs. De cada PC se utilizará sólo todo lo que está en el gabinete: la fuente de alimentación, el *motherboard* y los periféricos. Con este diseño aun es posible utilizar software de propósito general, como Windows y Linux [53].

Se podría diseñar también un gabinete especial que contendría todos los componentes de propósito general (fuente de alimentación, motherboard y los periféricos) para un sistema CMIDIP con un cierto número de interfases MIDI. Pero este CMIDIP sería más grande que una PC y su desarrollo requiere de conocimientos de electrónica y computación para realizar su configuración [99].

Para reducir aun más el tamaño, es posible utilizar una sola fuente de alimentación, un solo disco, y tantos motherboards como interfases MIDI se requieran. Pero además de requerir el diseño del gabinete, su fuente de alimentación, y los

conocimientos de electrónica y computación para realizar su configuración, es necesario diseñar el software específico para controlar el acceso de los varios procesadores e interfases a un solo disco.

3.5. Diseño de componentes a la medida

Por medio de diseñar a la medida algunos componentes, todavía es posible reducir aun más el tamaño de un CMIDIP. Utilizando un solo motherboard con un procesador relativamente rápido, se conectarían al motherboard las interfases MIDI requeridas. Cada interfaz MIDI es en sí un puerto serial. Por lo tanto, se debe diseñar la interconexión de los puertos seriales al motherboard, y para cada puerto, un sistema de referencia de tiempo musical, explicado a continuación.

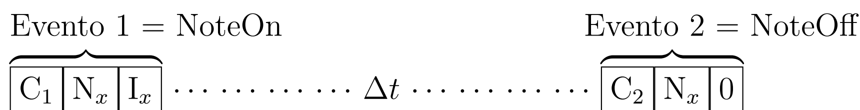


Figura 3.3: Eventos que se producen cuando se toca una nota

Típicamente un evento MIDI contiene información relevante de una nota musical: un código de comando (C_n), el nombre de la nota (N_x), y su intensidad (I_x). Tal y como se muestra en la Figura 3.3, cuando un instrumento toca una nota se producen dos eventos importantes. El primero se produce cuando se inicia la nota, mientras que el segundo se produce cuando se termina de tocar la nota. El primero arranca uno de los contadores de tiempo y el segundo lo detiene. Como se ha mencionado, el tiempo de duración (Δt) de la nota se calcula automáticamente y su resultado ($T_{n_0} \ldots T_{n_3}$) es utilizado para formar una estructura, como la que se muestra en la Figura 3.4, en la cual se describen la características de una nota y dan su aspecto en la partitura.

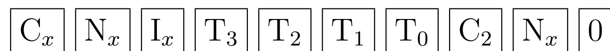


Figura 3.4: Parámetros de una nota completa para su impresión

Esta referencia de tiempo se puede implementar en hardware o software. La implementación en hardware sería un circuito integrado contador, al que también se debe diseñar su interconexión y control. Por ejemplo, después de recibir por un puerto serial un evento MIDI de activación de nota, este será grabado en memoria y se arrancará el contador correspondiente. Este contador corresponde al tiempo de duración de la nota, el cual se incrementa por hardware con base en las interrupciones del reloj del sistema. El contador se detiene hasta que se reciba el evento de apagado/desactivación de la nota. El tiempo de duración de la nota en el contador, representado en ticks de reloj, será entonces transformado a formato MIDI y grabado en la memoria, por ejemplo, siguiendo al evento MIDI de activación de la nota correspondiente. (El formato MIDI de tiempo de duración se explica en el Apéndice A.1).

Generalmente los sistemas CAC implementan el sistema de referencia de tiempo musical en software. La única diferencia con la versión de hardware anterior es que, por cada interrupción de reloj, el procesador ejecutaría la rutina de interrupción correspondiente, la cual incrementa las variables contador que hayan sido activadas. Sin embargo, será necesario diseñar la(s) estructura(s) de datos y su manejo para grabar los eventos recibidos de diferentes fuentes MIDI. Se puede utilizar sondeo (*polling*) o interrupciones en la detección de un evento MIDI. Entonces, se identifica la fuente y se inicia la captura y almacenamiento del evento MIDI en memoria.

La captura y almacenamiento en memoria de un evento MIDI se pueden realizar por software; lo cual es lo más común y conocido. Sin embargo, también es posible realizar la captura y almacenamiento completamente por hardware. En esta opción, el costo de diseño sería mayor, pero se obtendría mayor velocidad, lo que ayudaría a incrementar la precisión de la captura. Finalmente, es posible diseñar en un ASIC (*Aplication Specific Integrated Circuit*) todas las funciones de un dispositivo CMIDIP. El costo de diseño sería mayor, pero sería aún más pequeño y más rápido que las opciones anteriores [24][95].

3.6. Resumen

Presentamos el espacio de diseño y el funcionamiento conceptual de los sistemas de captura múltiple de eventos MIDI de varias fuentes en tiempo de ejecución musical. Los referimos como CMIDIP *Captura de eventos MIDI en Paralelo*, porque la captura de dos o más fuentes diferentes pueden ocurrir al mismo tiempo. Sus interfases MIDI son puertos seriales, los cuales están conectados a un sistema de referencia de tiempo que calcula cuanto tardan las notas en producir su sonido.

Es posible diseñar un sistema CMIDIP con componentes de propósito general, o diseñados especialmente a la medida. Se podría desarrollar rápida y económicamente un CMIDIP con PCs, pero este sistema es inflexible. Las PCs deben transportarse, conectarse y configurarse cada vez que se haga una grabación. Además, se requieren conocimientos que permitan su operación. También se podría configurar una red *cluster* con muchas PCs, o bien, utilizando muchas microcomputadoras. Estos sistemas CMIDIP son adecuados si su transportación no es frecuente (estudios de grabación). Sin embargo, en todos los casos existen limitaciones referente a la transferencia de datos.

Capítulo 4

Nuestros diseños de un Sistema de Captura de Múltiple Eventos MIDI

En el capítulo anterior presentamos el espacio de diseño de sistemas de Captura de eventos MIDI en Paralelo (CMIDIP). Vimos que es posible organizar un sistema CMIDIP con computadoras personales completas, con componentes de propósito general, o con componentes diseñados a la medida. Aunque las dos primeras alternativas se pueden desarrollar en un tiempo relativamente corto y a un precio accesible, no son fáciles de usar. Las conexiones entre los instrumentos y el CMIDIP serían estorbosas y complejas, además el usuario debe saber de software.

Creemos que un CMIDIP debe ser fácil de usar para ganar una aceptación general. Está dirigido a músicos que no necesitan saber de detalles técnicos complejos. Su operación debería requerir solamente conectar los instrumentos y presionar unos pocos botones. Cuando los músicos terminan de tocar se presionará el botón de paro. Entonces se podrá apagar el CMIDIP. Los datos permanecerán en su memoria gracias a un mecanismo electrónico que impida su desaparición. Posteriormente, estos datos serán descargados a la memoria de una PC conectando el CMIDIP a través de sus puertos seriales. Software corriendo en esta PC se encargará de procesar la información capturada, por ejemplo, para extraer las partituras.

En este capítulo presentamos dos diseños de sistemas CMIDIP que son portátiles y fáciles de usar. Cada uno consiste de circuitos integrados de hardware y software de

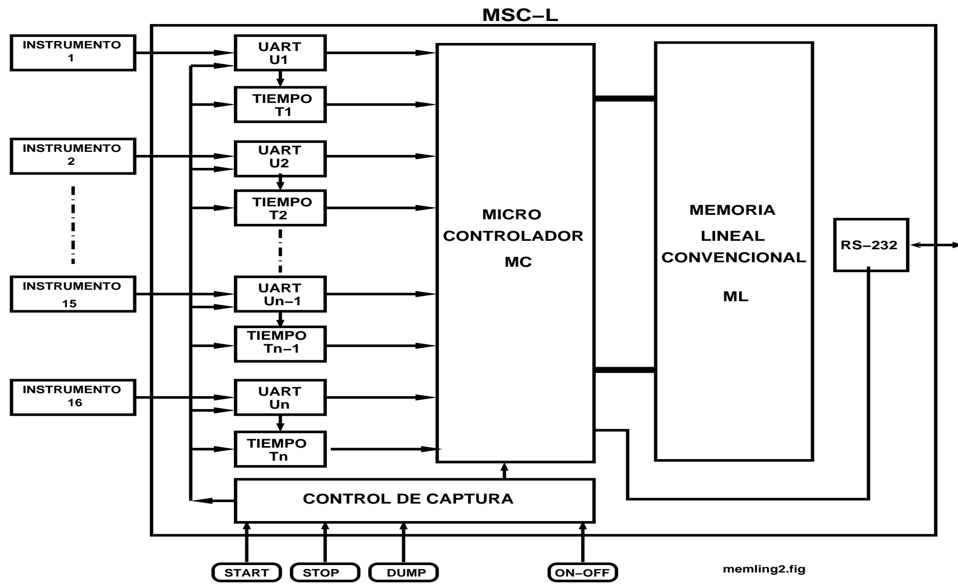


Figura 4.1: CMIDIP con memoria lineal MCS-L.

control que pueden ser integrados en un espacio relativamente pequeño. Un diseño, llamado MCS-L (MIDI *Capture System-Lineal*), utiliza componentes de propósito general, y por esta razón se puede desarrollar un prototipo en muy poco tiempo. Sin embargo, su fabricación masiva sería relativamente costosa debido a que requiere alrededor de 20 circuitos integrados convencionales. El otro diseño, llamado MCS-S (MIDI *Capture System-Segmented*), utiliza un diseño especial de memoria segmentada por hardware. Su ventaja es que sólo utiliza tres circuitos integrados de propósito específico (ASICs: *Application Specific Integrated Circuit*), por lo que en producción masiva su costo sería considerablemente menor que el de MCS-L.

4.1. Sistema MCS-L

La motivación de nuestro primer diseño de un sistema CMIDIP es un tiempo de desarrollo relativamente corto y un costo no muy elevado. Utiliza hardware de propósito general, y software especialmente diseñado para su control.

4.1.1. Hardware

La Figura 4.1 muestra la organización del hardware de nuestra primera propuesta, a la cual llamamos MCS-L (*MIDI Capture System-Lineal*) porque su memoria es del tipo lineal, que normalmente utilizan los sistemas computacionales. Cada interfaz MIDI, compuesta de un puerto serial (UART: *Unasynchronous Asynchronous Receiver Transmitter*) y un temporizador, recibe los comandos MIDI de un instrumento.

Cuando una interfaz MIDI recibe un comando de un instrumento, por ejemplo $NoteON$ (90H), en este momento se dispara un contador hardware que cuantifica el tiempo que tarda la nota produciendo sonido. Este contador se detiene hasta que se recibe el comando $NoteOff$ (80H), el cual indica que ya se ha apagado el sonido de la nota. Una vez que se han recibido los comandos y se ha cuantificado el tiempo, el microcontrolador (MC) almacena esta información en memoria junto con una etiqueta que identifica su fuente. La cuantificación del tiempo puede hacerse por software, pero es conveniente hacerlo por hardware con el fin de quitarle trabajo al procesador, y así asegurar que no se pierdan datos.

Una vez terminada la captura, los datos quedan en la memoria de MCS-L de manera permanente aun si se apaga MCS-L. Entonces, MCS-L puede conectarse, través de su puerto serial, a cualquier computadora con el fin de extraer la información, para procesarla, por ejemplo extraer la partitura impresa de cada instrumento que estuvo involucrado en la captura.

Para operar MCS-L se tienen algunos botones que ponen al sistema en cualquiera de los siguientes estados exclusivos: ON-OFF, apagado y encendido; START, inicia la captura de datos; STOP, detener la captura de datos; DUMP, descarga de datos en una computadora digital para su procesamiento posterior. Con DUMP se arranca un programa que permite interactuar con una terminal de consola (en una computadora personal) con el fin de escribir los comandos que controlan el vaciado de la memoria.

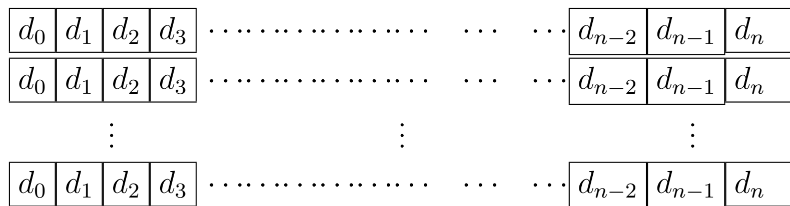


Figura 4.2: Varias listas FIFO independientes para cada instrumento

4.1.2. Software: captura y almacenamiento en memoria

En el modo de captura, MCS-L realiza por software la misma tarea básica, captura y almacenamiento de eventos MIDI, para cada instrumento. Esta tarea se hace en base a sondeo, el procesador revisa los registros de cada interfaz MIDI para capturar los eventos MIDI que pudieran enviar los instrumentos. Se pueden utilizar interrupciones, pero el hardware y el software se complica, ya que es necesario utilizar varios controladores de interrupción en cascada. Una vez que se ha capturado un evento MIDI, sigue su almacenamiento en memoria. Este almacenamiento debe facilitar la recuperación de los eventos MIDI en el orden en que fueron generados por cada interfaz: FIFO (*First In First Out*). A continuación describimos tres maneras de manejar la memoria con este propósito.

La memoria se puede particionar en tantas partes como interfaces MIDI hayan, y cada parte manejarla como una lista consecutiva de los eventos MIDI de cada interfaz, tal y como se muestra en la Figura 4.2. Esta estructura tiene la ventaja de poder enviar los eventos MIDI de cada instrumento no sólo en el orden en que fueron generados, sino también en el formato que cualquier sistema MIDI los puede identificar. Se puede manejar una lista *dinámica* por cada instrumento, en la que cada lista crece conforme se reciben eventos MIDI. Así se ahorraría memoria si un instrumento no genera eventos o genera muy pocos. Pero se gastaría memoria en el apuntador de cada nodo para ligarlo al siguiente nodo y el manejo se complica un poco.

También es posible, como se muestra en la Figura 4.3, manejar sólo una lista global compartida por todos los instrumentos y adicionando a cada evento un en-

Figura 4.3: Una sola lista FIFO para todos los instrumentos.

cabezado h que identifica la interfaz MIDI fuente. (Cada nodo en una lista global incluye el tiempo de duración, si éste último aplica). La ventaja de utilizar una lista global es que no se desperdicia memoria si un instrumento no genera eventos; pero tiene la desventaja de utilizar más memoria para el encabezado. A diferencia de la estructura anterior, el procesamiento de esta lista requiere de software especial que interprete los encabezados correctamente.

4.2. Sistema MCS-S

La motivación de nuestro segundo diseño de un sistema CMIDIP, es una mayor precisión de la captura de eventos MIDI con base en una mayor velocidad de procesamiento de los mismos y costo menor en caso de ser fabricado masivamente. En este diseño, al cual llamamos MCS-S (*MIDI Capture System-Segmented*), la memoria y la función de almacenamiento en la misma de los eventos de cada interfaz MIDI se encuentra dentro de un ASIC (*Application Specific Integrated Circuit*). Los puertos seriales y los temporizadores de todas las interfaces MIDI se encuentran también en un ASIC. La Figura 4.4 muestra nuestro MCS-S.

Junto con el microcontrolador (MC) y los dos ASICs que se han mencionado, el MCS-S consiste de tres circuitos integrados principales. Debido a este número reducido de componentes, MCS-S tendrá un costo menor si se produce masivamente, además el manejo de la memoria por hardware es más rápido. El ASIC que contempla los puertos seriales y temporizadores de las interfaces MIDI contiene registros de control para cada interfaz, y su control es igual al ya descrito para MCS-L. El procesador sondea cada interfaz para ver si se ha recibido un evento MIDI (aunque se pueden usar interrupciones). Por otro lado, el ASIC de la memoria es un diseño es-

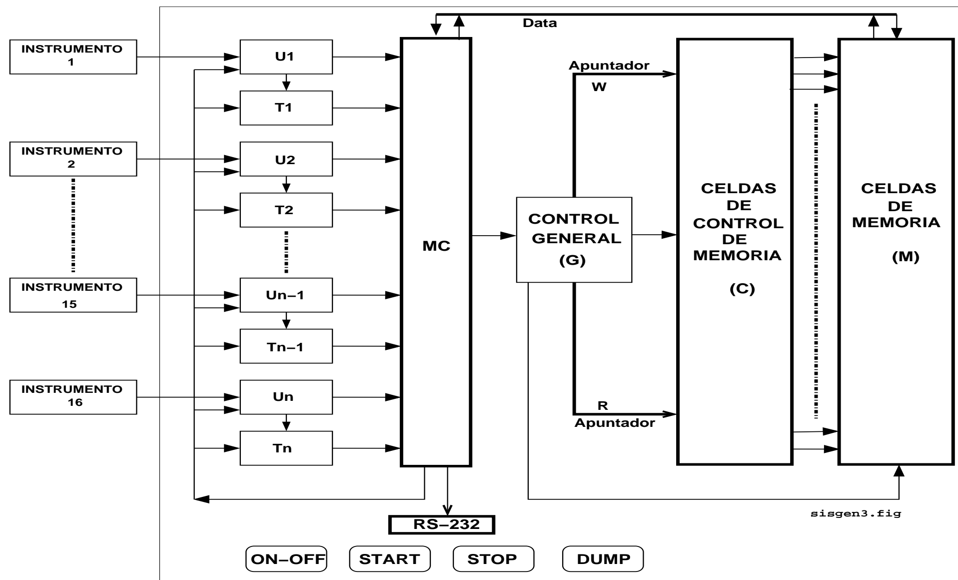


Figura 4.4: CMIDIP con memoria segmentada MCS-S.

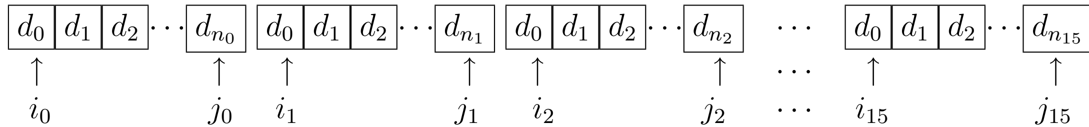


Figura 4.5: Segmentos FIFO de tamaño variable concatenados dentro del espacio de la memoria segmentada; i apunta al inicio de un segmento, y j al final.

pecial que maneja internamente varias listas FIFO independientes, una por cada interfaz MIDI, tal y como se muestra en la Figura 4.5.

Como se ve, al inicio y al final de cada lista se encuentra un par de registros i_n y j_n, donde n identifica a la interfaz MIDI. Cada lista constituye un segmento de tamaño *variable* [23]. Además, en todo momento las listas están contiguas en memoria. Esto permite ahorrar memoria cuando un instrumento no genera eventos MIDI. Pero requiere de un mecanismo especial que mueve, en un solo ciclo de reloj, todos los datos de todas las listas simultáneamente, hacia la derecha cuando se inserta un dato en cualquier lista, o hacia la izquierda cuando se extrae un dato de cualquier lista [20]. Por esta funcionalidad, nos referimos a esta memoria como Memoria Segmentada Autocompactante, o MSA.

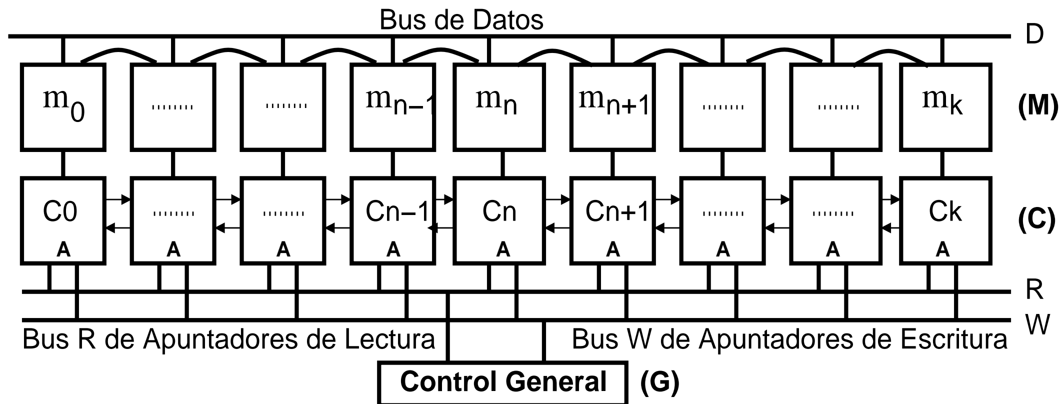


Figura 4.6: Sistema de Memoria Segmentada Autocompactante MSA.

4.2.1. Memoria Segmentada Autocompactante (MSA): funcionamiento

Como se recordará, el MCS-S utiliza una memoria segmentada con espacios de direcciones que se compactan de manera automática. Esta MSA tiene sólo dos comandos de operación: *inserción* y *extracción* de datos en, o de, una lista. Estas dos operaciones corresponden a la *escritura* y *lectura* de datos, respectivamente. La escritura se usa sólo en el modo de captura de eventos MIDI, mientras que la lectura se usa sólo para descargar los datos capturados a una PC para su posterior procesamiento. La MSA consiste de los siguientes componentes o subsistemas (ver Figura 4.6): un control general (G), celdas de memoria (M) y sus celdas de control (C), los cuales describimos a continuación.

Primero veremos que la unidad de control general (G) es la encargada de enviar las señales de carácter global a los otros subsistemas. Por ejemplo, cuando el microcontrolador (MC) ha leído un evento de una interfaz MIDI para grabarlo en memoria, este tipo de procesador le indica a G en que lista se ha de guardar y entonces escribe el evento MIDI en el *bus* de la MSA. Entonces, G genera las señales de control que son comunes en las terminales de los componentes que constituyen el sistema de memoria como son sus registros, sus celdas de control, y sus buses de dirección y datos.

Ahora veamos el sistema de memoria (M). Ésta se encuentra particionada en segmentos de celdas de memoria; cada segmento implementa una lista FIFO. Cada celda puede almacenar un solo dato dentro de su registro. De esta forma, el sistema de memoria segmentada autocompactante M está definida mediante la expresión 4.1.

$$M = \{m_0, m_1, m_2, m_3, \ldots, m_{n-1}, m_n, m_{n+1}, \ldots, m_k\} \tag{4.1}$$

Dada una celda de memoria m_n determinada, entonces m_{n-1} es la celda de memoria vecina inmediata anterior, y m_{n+1} es la celda de memoria vecina inmediata posterior; k es el tamaño máximo de la memoria. Cabe recordar que las celdas de memoria son consecutivas, y una celda m_n puede, además de almacenar el dato de un evento MIDI, leer (almacenar) el dato que tiene una de sus celdas vecinas.

Consideremos ahora las celdas de control (C). Estas provocan, con base en la lectura o escritura de una celda de memoria, el corrimiento a la izquierda o a la derecha, respectivamente, del contenido de todas las celdas en memoria. Existe una celda de control por cada celda de memoria, y en conjunto se pueden denotar por la expresión 4.2, Dada una celda de control c_n determinada, entonces c_{n-1} es la celda vecina inmediata anterior, y c_{n+1} es la celda vecina inmediata posterior; k es el tamaño de la memoria.

$$C = \{c_0, c_1, c_2, c_3, \ldots, c_{n-1}, c_n, c_{n+1}, \ldots, c_k\}. \tag{4.2}$$

Cada celda c_n puede interactuar con sus vecinas, c_{n-1} y c_{n+1}, intercambiando señales de control entre ellas con base en las señales del control general y otros dispositivos. Las celdas de control, y consecuentemente las de memoria, operan en paralelo en cada pulso de reloj. De esta manera el control de las operaciones de escritura o lectura se hace en forma concurrente, logrando con ello una máxima velocidad de operación, pero sin perder datos.

Si se escribe un dato en la celda de memoria m_n, las celdas $c_{n+1}, c_{n+2}, \ldots$ generan señales de control para desplazar hacia la derecha una posición el contenido de las celdas de memoria $m_{n+1}, m_{n+2}, \ldots$, dejando un hueco para colocar el nuevo da-

to; todo esto sucede en un solo pulso de reloj. De la misma forma, Si se lee un dato en la celda de memoria m_n, las celdas c_{n+1}, $c_{n+2}\ldots$ generan señales de control para desplazar hacia la izquierda una posición el contenido de las celdas de memoria $m_{n+1}, m_{n+2}, \ldots$, llenando el hueco dejado por el dato leído.

4.2.2. Identificadores de datos y listas

Recordamos que la memoria MSA está particionada en listas, y que para grabar un dato, o leer un dato, de la misma, es necesario especificar la lista destino/fuente. Por ejemplo, cuando el microcontrolador (MC) lee un evento de una interfaz MIDI para grabarlo en memoria, indica al control general (G) en que lista se ha de guardar. G escribe la identificación de la lista en el bus de *apuntadores de escritura* de la MSA, e indica a MC que puede escribir el dato MIDI en el bus de datos de la MSA. Finalmente, MC escribe el dato MIDI. El proceso de lectura es similar, pero G escribe la identificación de la lista en el bus de *apuntadores de lectura*. El bus de apuntadores de escritura y el bus de apuntadores de lectura se representan en la Figura 4.4, como *apuntador W* y *apuntador R*, respectivamente.

El bus de apuntadores de escritura y el bus de apuntadores de lectura alcanzan a cada una de las celdas de control. Cada celda de control tiene un registro donde se almacena la identificación del final de una lista, del inicio de una lista o de un dato intermedio, almacenado en la celda de memoria correspondiente. Esta identificación es comparada con el valor puesto en cualquiera de los buses de apuntadores. Cuando la identificación en una celda de control corresponde al valor puesto en uno de los buses, la operación de escritura o lectura afecta a esa celda de control y a las subsecuentes, así como a las celdas de memoria correspondientes.

Se notará que, debido a los corrimientos que ocurren cuando se inserta o extrae un dato, la identificación en las celdas de control varia dinámicamente. Al recorrer los datos a la izquierda o a la derecha en varias celdas de memoria, la identificación en las celdas de control correspondientes se recorren acordemente. Se notará también

que, por el manejo FIFO de cada lista, los identificadores de los finales de listas están asociados con el bus de apuntadores de escritura, mientras que los identificadores de los inicios de listas están asociados con el bus de apuntadores de lectura. Por esta razón, a los identificadores de los finales de listas les llamaremos *direcciones de escritura* W_i, y a los identificadores de inicios de listas *direcciones de lectura* R_i. También, usaremos *direcciones de dato* D_i para referirnos a los identificadores de datos.

Un ejemplo ayudará a entender estos conceptos. Al iniciar el sistema, y suponiendo que éste puede manejar 16 listas, las primeras 16 celdas de control tendrán las direcciones que se muestran en la expresión 4.3, cuyos valores numéricos constantes se muestran en la expresión 4.4, las cuales identifican a cada una de las 16 listas:

$$W_1W_2W_3W_4W_5W_6W_7W_8W_9W_{10}W_{11}W_{12}W_{13}W_{14}W_{15}W_{16}\Phi, \tag{4.3}$$

$$02_h0A_h12_h1A_h22_h2A_h32_h3A_h42_h4A_h52_h5A_h62_h6A_h72_h7A_h00_h. \tag{4.4}$$

Ahora, supóngase que se escriben/insertan tres datos en la lista 1, en 3 pulsos de reloj sucesivos: t_1, t_2 y t_3. El arreglo 4.5 muestra las direcciones que se generan y su corrimiento a través de las celdas de control: (t_0), es el estado inicial; (t_1), inserción del primer dato; (t_2), inserción del segundo dato; (t_3), inserción del tercer dato, etc.

$$\begin{bmatrix} t_0 \\ t_1 \\ t_2 \\ t_3 \\ \vdots \\ t_n \end{bmatrix} \begin{bmatrix} W_1W_2W_3W_4W_5W_6W_7W_8W_9W_{10}W_{11}W_{12}W_{13}W_{14}W_{15}W_{16}\Phi \\ W_1R_1W_2W_3W_4W_5W_6W_7W_8W_9W_{10}W_{11}W_{12}W_{13}W_{14}W_{15}W_{16}\Phi \\ W_1D_1R_1W_2W_3W_4W_5W_6W_7W_8W_9W_{10}W_{11}W_{12}W_{13}W_{14}W_{15}W_{16}\Phi \\ W_1D_1D_1R_1W_2W_3W_4W_5W_6W_7W_8W_9W_{10}W_{11}W_{12}W_{13}W_{14}W_{15}W_{16}\Phi \\ \vdots \\ W_1D_1\ldots D_1D_1R_1W_2\ldots \end{bmatrix} \tag{4.5}$$

Cuando se inserta un dato en una lista vacía, se crea una dirección de lectura (R) enseguida de la dirección de escritura (W); la segunda inserción crea una dirección de dato (D) que empuja a una de lectura (R), las siguientes operaciones sucesivas crean direcciones de dato (D) que desplazan a las anteriores, y así sucesivamente, hasta la capacidad de la memoria.

Ahora consideremos la lectura/extracción de tres datos. El arreglo (4.6) muestra el estado de direcciones que toma la memoria: (t_n), existen 3 datos; (t_{n+1}) se ex-

trae el primer dato; (t_{n+2}), se extrae el segundo dato; (t_{n+3}), se extrae el tercer, último dato y desaparece R. R y las direcciones que le siguen deben hacer un desplazamiento simultáneo para llenar el hueco. R desaparece cuando se extrae el último dato, y por lo tanto el segmento queda vacío, quedando solo la dirección de escritura.

$$\begin{bmatrix} t_0 \\ \vdots \\ t_n \\ t_{n+1} \\ t_{n+2} \\ t_{n+3} \end{bmatrix} \begin{bmatrix} W_1D_1 \dots D_1D_1R_1W_2 \dots \\ \vdots \\ W_1D_1D_1R_1W_2W_3W_4W_5W_6W_7W_8W_9W_{10}W_{11}W_{12}W_{13}W_{14}W_{15}W_{16}\Phi \\ W_1D_1R_1W_2W_3W_4W_5W_6W_7W_8W_9W_{10}W_{11}W_{12}W_{13}W_{14}W_{15}W_{16}\Phi \\ W_1R_1W_2W_3W_4W_5W_6W_7W_8W_9W_{10}W_{11}W_{12}W_{13}W_{14}W_{15}W_{16}\Phi \\ W_1W_2W_3W_4W_5W_6W_7W_8W_9W_{10}W_{11}W_{12}W_{13}W_{14}W_{15}W_{16}\Phi \end{bmatrix} \tag{4.6}$$

Finalmente, cabe recordar que para manejar la memoria segmentada, el software de control sólo requiere escribir comandos en los registros de control de la MSA con el propósito de activar diferentes funciones en la unidad de control general (G). Debido a que, dentro de la memoria, la mayor parte del trabajo de movimiento de datos se hace por hardware y de manera paralela, el software de captura sólo coloca los datos sobre el bus de la MSA, y el hardware de la memoria hace el resto. De esta forma, muchas de las funciones de captura se han delegado al hardware, de tal manera que el software realiza poco trabajo.

4.3. Discusión

Nuestra primera propuesta, MCS-L, está constituido por un procesador y componentes auxiliares de propósito general. Éste es un dispositivo pequeño y ligero y por lo tanto manejable. La grabación de los datos MIDI de varios instrumentos utiliza un manejo de memoria con el cual la información de los instrumentos queda separada utilizando varias estructuras FIFO, una para cada instrumento, o bien una sola estructura, pero adicionando un encabezado en cada dato que se captura a fin de identificar al instrumento que lo genera.

MCS-L utiliza varios componentes de propósito general (más de 20 en nuestro prototipo) que se deben ensamblar. Por el número de componentes, su fabricación

masiva es probable que sea relativamente costosa y propensa a errores. Sin embargo, es conveniente como una prueba de concepto, con fines académicos, por ejemplo. Por otro lado, MCS-S utiliza sólo tres componentes principales en nuestro diseño, uno de los cuales es un diseño especial de memoria segmentada autocompactante. Aunque se requiere un comportamiento especial de la misma, el diseño de MCS-S se simplifica, además, por utilizar menos componentes, este último tiene la ventaja de ser mucho más compacto, económico, y menos propenso a errores en producción masiva.

Ambas propuestas utilizan el mismo software que administra la captura de los datos y el vaciado de la memoria hacia una computadora *host*. Pero en MCS-S no se requiere que el software maneje la memoria debido a que el acceso a sus datos lo realiza por hardware.

4.4. Resumen

Hemos presentado el diseño de dos sistemas CMIDIP, los cuales se pensaron para formar un sistema de captura que sea pequeño y portátil. MCS-L utiliza componentes comunes y una memoria convencional estándar. Por otro lado, MCS-S utiliza una memoria segmentada autocompactante no convencional y de diseño especial. Pensando en su fabricación masiva, su costo comercial del sistema completo podría ser lo más reducido posible, pues MCS-S sólo utiliza a lo más tres circuitos integrados de propósito específico. Las especificaciones de la memoria segmentada de MCS-S permiten la separación automática de las melodías que tocan en paralelo varios instrumentos musicales. Los datos se insertan en espacios diferentes de tamaño variable, de tal manera que si un instrumento musical no genera datos, dicho espacio es aprovechado por otro que sí esté generando datos. En este capítulo sólo presentamos el funcionamiento conceptual de la memoria segmentada autocompactante; en el siguiente capítulo presentamos su diseño.

Capítulo 5

Diseño de la Memoria Segmentada Autocompactante

En el capítulo anterior presentamos nuestros dos diseños para un sistema de captura de múltiples eventos MIDI: MCS-L utiliza una memoria convencional, y MCS-S utiliza una memoria segmentada y autocompactante por hardware. La memoria de MCS-S, a la cual nos referimos como *Memoria Segmentada Autocompactante* (MSA), es un diseño especial y en el capítulo anterior sólo describimos su funcionamiento. En este capítulo presentaremos su diseño en detalle.

5.1. Terminología

Recordamos que MSA está formada por una sucesión consecutiva de celdas de memoria (Figura 4.6), cada una de las cuales es controlada por una celda (unidad) de control asociada a cada una de ellas [21]. Físicamente, cada celda de memoria (c_n) está conectada eléctricamente con su inmediata anterior (c_{n-1}) y su inmediata posterior (c_{n+1}). En la Figura 5.1 se muestra de una manera general, las celdas de memoria; donde: c_0 es la primera celda, c_k es la última celda, R es el *bus* de lectura, W es el *bus* de escritura, D es el *bus* de datos, y A es la dirección de celda.

Para escribir y leer datos en/desde la memoria se utilizan *Apuntadores de Memoria*, y *Direcciones de Memoria.* Los apuntadores son valores fijos y externos a la MSA, mientras que las direcciones también son valores fijos, pero se encuentran dentro de

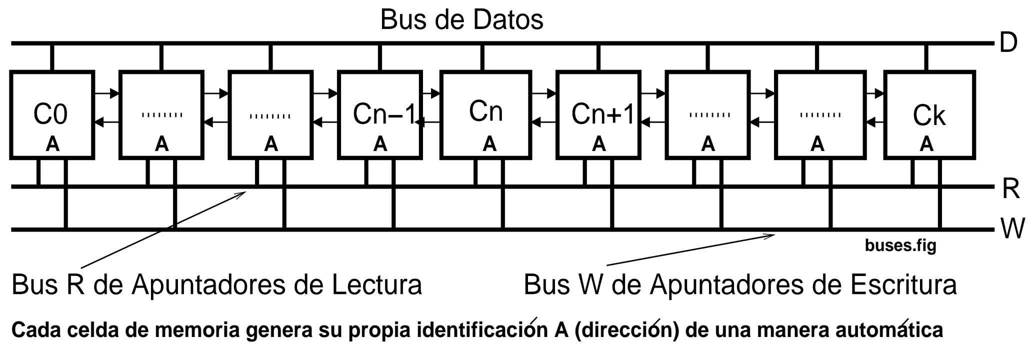


Figura 5.1: Celdas de memoria y *buses*.

la MSA. Los apuntadores a memoria P se dividen en dos grupos (Expresión 5.1) llamados *Apuntadores de Escritura*, denotados por W_{p_1}, ..., W_{p_n}, y los *Apuntadores de Lectura*, denotados por R_{p_1}, ..., R_{p_n}.

Para hacer más claro nuestro diseño, consideraremos una memoria con 16 segmentos. Cuando hay una operación de escritura/lectura el valor numérico de dichos apuntadores es generado de manera automática por el control general, y depositado en uno de los *buses* comunes a cada una de las celdas de memoria.

$$P = \{W_{p_1}, \dots, W_{p_{16}}, R_{p_1}, \dots, R_{p_{16}}\} \tag{5.1}$$

Por otra parte, las direcciones (identificadores) A de los datos en la memoria también están divididas, pero en tres grupos (Expresión 5.2) llamados *Direcciones de Escritura*, y denotadas por W_{a_1}, ..., W_{a_n}; *Direcciones de Lectura*, denotadas por R_{a_1}, ..., R_{a_n}; y *Direcciones de Dato*, denotadas por D_{a_1}, ..., D_{a_n}. Además, hay una *Dirección Vacía* denotada por Φ que sirve para mejorar el rendimiento del control de hardware.

$$A = \{W_{a_1}, \dots, W_{a_{16}}, R_{a_1}, \dots, R_{a_{16}}, D_{a_1}, \dots, D_{a_{16}}, \Phi\} \tag{5.2}$$

En la Tabla 5.1 se muestran cada uno de los valores que toman dichos apuntadores y direcciones. Nótese que sólo se utiliza una dirección (identificador) de dato D_{a_n} para identificar todos los datos de un segmento. Esto es posible porque sólo se pueden escribir datos al final de una cola, y leer datos del inicio de una cola. Por

Tabla 5.1: Tabla de valores fijos para direcciones y apuntadores.

ESTRUCTURA DE DIRECCIONES Y APUNTADORES [direcs.tex]						
NOMBRE	APUNTADOR	DIRECCION	BIN	HEX	DEC	SECUENCIA
Escritura	Wp_1	W_1	000 0010	02	2	1
Dato		D_1	000 0011	03	3	2
Lectura	Rp_1	R_1	000 0100	04	4	3
Escritura	Wp_2	W_2	000 1010	0A	10	4
Dato		D_2	000 1011	0B	11	5
Lectura	Rp_2	R_2	000 1100	0C	12	6
Escritura	Wp_3	W_3	001 0010	12	18	7
Dato		D_3	001 0011	13	19	8
Lectura	Rp_3	R_3	001 0100	14	20	9
Escritura	Wp_4	W_4	001 1010	1A	26	10
Dato		D_4	001 1011	1B	27	11
Lectura	Rp_4	R_4	001 1100	1C	28	12
Escritura	Wp_5	W_5	010 0010	22	34	13
Dato		D_5	010 0011	23	35	14
Lectura	Rp_5	R_5	010 0100	24	36	15
Escritura	Wp_6	W_6	010 1010	2A	42	16
Dato		D_6	010 1011	2B	43	17
Lectura	Rp_6	R_6	010 1100	2C	44	18
Escritura	Wp_7	W_7	011 0010	32	50	19
Dato		D_7	011 0011	33	51	20
Lectura	Rp_7	R_7	011 0100	34	52	21
Escritura	Wp_8	W_8	011 1010	3A	58	22
Dato		D_8	011 1011	3B	59	23
Lectura	Rp_8	R_8	011 1100	3C	60	24
Escritura	Wp_9	W_9	100 0010	42	66	25
Dato		D_9	100 0011	43	67	26
Lectura	Rp_9	R_9	100 0100	44	68	27
Escritura	Wp_{10}	W_{10}	100 1010	4A	74	28
Dato		D_{10}	100 1011	4B	75	29
Lectura	Rp_{10}	R_{10}	100 1100	4C	76	30
Escritura	Wp_{11}	W_{11}	101 0010	52	82	31
Dato		D_{11}	101 0011	53	83	32
Lectura	Rp_{11}	R_{11}	101 0100	54	84	33
Escritura	Wp_{12}	W_{12}	101 1010	5A	90	34
Dato		D_{12}	101 1011	5B	91	35
Lectura	Rp_{12}	R_{12}	101 1100	5C	92	36
Escritura	Wp_{13}	W_{13}	110 0010	62	98	37
Dato		D_{13}	110 0011	63	99	38
Lectura	Rp_{13}	R_{13}	110 0100	64	100	39
Escritura	Wp_{14}	W_{14}	110 1010	6A	106	40
Dato		D_{14}	110 1011	6B	107	41
Lectura	Rp_{14}	R_{14}	110 1100	6C	108	42
Escritura	Wp_{15}	W_{15}	111 0010	72	114	43
Dato		D_{15}	111 0011	73	115	44
Lectura	Rp_{15}	R_{15}	111 0100	74	116	45
Escritura	Wp_{16}	W_{16}	111 1010	7A	122	46
Dato		D_{16}	111 1011	7B	123	47
Lectura	Rp_{16}	R_{16}	111 1100	7C	124	48
Nulo		Φ	0000-0	00	0	49

lo tanto no hay necesidad de manejar *direcciones distintas* para cada dato, sólo para el inicio y final de cada segmento al que pertenecen.

Recordemos que una operación de escritura es el proceso de insertar un dato dentro de un segmento de la memoria. La primera vez que se inserta un dato, el control crea una dirección de lectura R_{a_n} y la coloca enseguida de una de escritura W_{a_n}; la segunda inserción crea una dirección de dato D_{a_n} que empuja a una de lectura R_{a_n}, las siguientes operaciones sucesivas crearán direcciones de dato D_{a_n}, las cuales desplazarán a las subsecuentes por la derecha.

De la misma forma, una operación de lectura es el proceso de extraer un dato desde un segmento de memoria, el cual se encuentra identificado con dirección R_{a_n}. Esta R_{a_n} y los identificadores de direcciones que se encuentran después, hacen un desplazamiento simultáneo hacia la izquierda para llenar el hueco que se ha dejado. Por último, R_{a_n} desaparece cuando se extrae el último dato, y por lo tanto el segmento ha quedado vacío.

El funcionamiento de la memoria MSA se puede explicar en base a los diferentes estados por los que pasan las celdas de memoria durante las operaciones de escritura y lectura sobre las mismas.

5.2. Estados Escritura

Una escritura ocurre en el pulso de reloj que sigue a la colocación, por el sistema operativo, de uno de los apuntadores W_{p_n} en el *bus* de escritura. Este bus está conectado a todas las celdas de memoria; y dicho apuntador selecciona a uno de los 16 segmentos que almacenará la información [81].

Cada celda de memoria puede estar en uno de 10 estados antes de que ocurra una escritura; su estado junto con varias *señales de entrada* determinan su transición a otro estado al concluir la escritura. Estos estados se representan en la Figura 5.2 por los números enteros encerrados en un círculo. En esta figura, W_p apunta al inicio del segmento W_a sobre el cual se desea insertar un dato. El valor A en algunas cel-

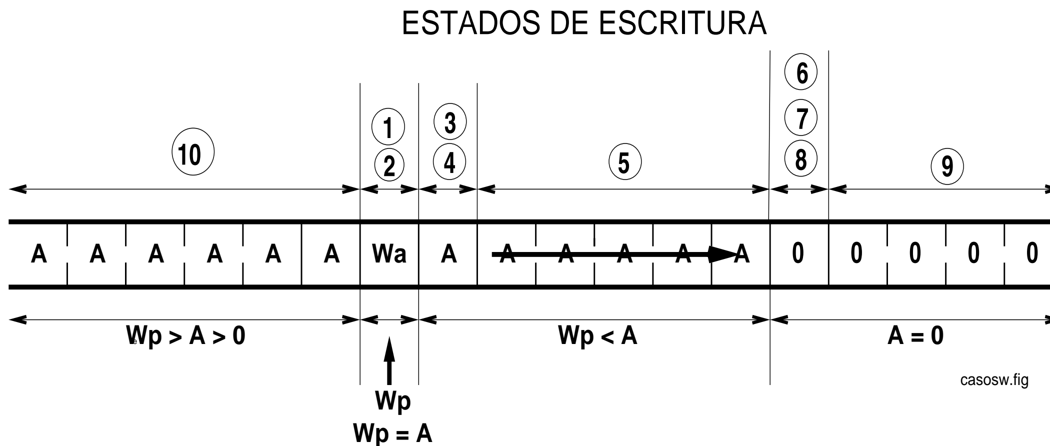


Figura 5.2: Estados de escritura de las celdas de memoria.

das representa un valor arbitrario que puede ser otra dirección de escritura W_a, una dirección de lectura R_a, una dirección de dato D_a o la dirección vacía Φ (0).

Para cada estado que tiene una celda se presenta su descripción operativa, una función lógica que determina su comportamiento, y un arreglo matricial que muestra el flujo de datos cuando hay actividad en dicha celda. La Figura 5.2, muestra los estados en que pueden estar, antes de escribir un dato, cada una de las celdas a lo largo de la memoria segmentada, donde: A es la dirección que identifica a una celda, W_p es el apuntador de escritura, y los dígitos $1, 2, \ldots, 9, 10$, son los estados (sólo de escritura) que se han definido de acuerdo con lo que se muestra en la Tabla 5.2; la primera columna contiene el número de estado, la parte izquierda presenta las condiciones de entrada y el lado derecho es la salida que genera la función de estados.

De la misma forma, y para todos los estados que siguen, la parte izquierda de la función lógica es una expresión booleana de las señales de entrada, y la parte derecha son las señales producidas. Cada función se ha diseñado de acuerdo con la Tabla 5.2.

5.2.1. Estado 1.

En el estado 1 un segmento W_a está vacío y este estado es indicado con señales de salida (descritas a continuación) a las celdas vecinas. Este estado se presenta cuan-

Tabla 5.2: Tabla de estados y comandos para las celdas de control.

ESTADOS DEL CIRCUITO DE LA CELDA DE CONTROL DE MEMORIA

CONDICIONES DE ENTRADA

	12	11	10	9								8	7	6	5	4	3	**2**	1	0
ESTADO	CW	CR	E1C (W<R)	E2C (W>R)	AR=0	AR=WW	AR=RR	AR>WW	AR>RR	AR<WW	AR<RR	WI	RI	WL	SS	AUWI	AURI	ADWI	USSI	DSSI
1	1					1									0					
2	1					1									1					
3	1							1										1		0
4	1							1										1		1
5	1				0			1										0		
6	1				1							1								
7	1				1								1							
8	1				1													1		0
9	1				1							0	0							
10	1				0					1										
11		1					1													
12		1			0						1						0			
13		1			0						1			1			1			
14		1			0						1			0			1			
15		1							1											
16		1			1									0			0			

COMANDOS DE SALIDA

	16	15	14	13	12	11	10	9	8	7	6	5	4	3	**2**	1	0
ESTADO	EAR	EM	WO	RO	MS	MR	X0C	X1C	X2C	X3C	X4C	ATRO	AUWO	AURO	ADWO	USSO	DSSO
1		1			1										1		0
2															1		1
3	1	1					0	0	1	1	0						
4	1	1					0	1	0	1	0						
5	1	1					0	0	0	0	0						
6	1	1					0	0	0	0	0						
7	1	1					0	0	0	0	0						
8	1	1					0	0	1	1	0						
9																	
10																	
11	1	1					1			1	1	1		1			
12																	
13		1				1											
14	1						1										
15	1	1					1			1	1						
16																	

CW: Escritura.
CR: Lectura.
E1C: Lectura Escritura (pw<ar).
E2C: Lectura Escritura (wp>ar).
AR: Registro de direcciones.
WW: Dir. de escritura.
RR: Dir. de lectura.
WI: La celda anterior Cn−1 es W.

RI: Celda anterior Cn−1 es R.
WL: Celda actual Cn es W.
SS: Estado de segmento (vacio SS=0, ocupado SS=1).
AUWI: La celda siguiente Cn+1 es W activo.
AURI: La celda siguiente Cn+1 es R activo.
ADWI: La celda anterior Cn−1 es W activo.
USSI: Estado del segmento de la celda siguiente Cn+1.
DSSI: Estado del segmento de la celda anterior Cn−1.

EAR: Habilitar Regitro de Direcciones A.
EM: Habilitar celda de memoriia Mn.
RO: La celda actual Cn es R.
MS: Fijar la memoria en 1 (segmento ocupado).
MR: Reestablecer celda de memoria en 0 (Segmento vacio).
X0C: Desplazamiento de AR (derecha =0, izaquierda=1).
X1C: AR crea direccion de datos D.
X2C: AR crea direccion de lectura R.

X3C: Control de la memoria.
X4C: Control de la memoria.
ATRO: Habilitar salida de memoria.
AUWO: Indicar a Cn−1 que Cn es W.
AURO: Indicar a Cn−1 que Cn es R.
ADWO: Indicar a Cn+1 que Cn es W.
USSO: Indicar a Cn−1 estado de Cn.
DSSO: Indicar a Cn+1 estado de Cn.

SISTEMA CON MEMORIA SEGMENTADA

TABLA DE ESTADOS DE LA CELDA DE CONTROL

tabnew.fig

do se inserta el primer dato en W_a.

Las señales de entrada son: CW habilita escribir dato; WW alimenta al apuntador de segmento W_p; AR alimenta la dirección A de la celda (en esta caso W_a); y $\overline{SS}$ indica *status* de segmento vacío.

Las señales de salida son: EM habilita la celda de memoria para la operación de escritura; MS escribe FF_H en un registro de control para indicar a las celdas vecinas(c_{n+1} y c_{n-1}) que este segmento ya no está vacío; SS es una copia del bit más significativo de este registro e indica lo mismo a la celda W_a (c_n); $ADWO$ dice a la celda siguiente (c_{n+1}) que aquí hay una dirección de escritura W activada; y $\overline{DSSO}$ es una copia del *status* anterior $\overline{SS}$ indicando a la celda siguiente (c_{n+1}) que esta región de memoria no contenía información.

Función lógica:

$$(CW \cdot (WW = AR) \cdot \overline{SS}) \Longrightarrow (EM, MS, ADWO, \overline{DSSO})$$

Flujo de datos:

$$\begin{array}{cccccc} & & \overbrace{\textit{vacio} \leftarrow \textit{ocupado}} & \overbrace{\textit{nuevo dato}} & \\ \cdots & m_{n-1} & m_n & m_{n+1} & \cdots \\ \cdots & c_{n-1} & c_n & c_{n+1} & \cdots \\ \cdots & A & W & A \leftarrow R & \cdots \\ & \underbrace{W_p > A > 0} & \underbrace{W_p = A} & \underbrace{W_p < A} & \end{array}$$

En esta definición del flujo de datos (y las de otros estados), las relaciones entre W_p y A toman en cuenta los valores de direcciones de la Tabla 5.1, donde una celda vacía Φ tiene una dirección de identificación 0 (cero).

5.2.2. Estado 2

Una celda W_a en el estado 1 pasa al estado 2 una vez que se completa la escritura del primer dato en el segmento que identifica. En el estado 2, dicha celda indica a las celdas vecinas que este segmento tiene por lo menos un dato. Este estado corresponde a la condición para insertar más datos en el segmento. En el estado 2, y próxima a ocurrir una escritura, las señales de entrada y salida son las siguientes.

Las señales de entrada son: CW habilita escritura; WW alimenta el apuntador W_p; AR alimenta la dirección W_a; y SS señala, así misma (c_n), *status* de segmento ocupado.

Las señales de salida son: $ADWO$ dice a la siguiente celda (c_{n+1}) que esta celda (c_n) tiene una dirección W activada; y $DSSO$ es una copia del *status* anterior SS indicando a la celda siguiente (c_{n+1}) que esta región tiene por lo menos un dato.

Función lógica:

$$(CW \cdot (WW = AR) \cdot SS) \Longrightarrow (ADWO,\, DSSO)$$

Flujo de datos:

$$\begin{array}{cccccc} & & \overbrace{}^{\textit{status ocpado}} & \overbrace{\phantom{m_{n+1}}}^{\textit{nuevo dato}} & \overbrace{\phantom{m_{n+2}}}^{\textit{dato}} & \\ \cdots & m_{n-1} & m_n & m_{n+1} & m_{n+2} & \cdots \\ \cdots & c_{n-1} & c_n & c_{n+1} & c_{n+2} & \cdots \\ \cdots & \underbrace{A}_{W_p > A > 0} & \underbrace{W}_{W_p = A} & \underbrace{R \leftarrow D}_{W_p < A} & \underbrace{A \leftarrow R}_{W_p < A} & \cdots \end{array}$$

Nótese en la Figura 5.2, que sólo una celda W_a puede pasar por los estados 1 y 2. Sin embargo, se notará que, debido a los desplazamientos automáticos de los datos, una celda con una dirección de datos D_a o con una dirección de lectura R_a se puede convertir en una celda con una dirección de escritura W_a y viceversa.

5.2.3. Estado 3

Una celda se encuentra en el estado 3 cuando sigue por la derecha a una celda W_a en el estado 1 (ver Figura 5.2). En el estado 3, una celda transforma su dirección anterior a una dirección de lectura R_a, indicando que es el primer dato en el segmento. (Su dirección anterior sólo puede ser escritura: cuando hay dos segmentos vacíos consecutivos W_aW_{a+1}). En el estado 3, una celda recibe el primer dato en un segmento.

Nótese que en la celda identificada W_a no se guarda el primer dato, sino en la celda siguiente. Este aspecto es para facilitar el diseño (tal como se facilita el manejo de colas en software por medio de manejar una localidad de administración extra).

Las señales de entrada son: CW habilita escritura; WW alimenta el apuntador W_p; AR alimenta la dirección A de esta celda; $ADWI$ señala que la celda anterior (c_{n-1}) tiene la dirección de escritura W activada; y $\overline{DSSI}$ manifiesta *status* vacío, es decir, este segmento no contiene datos.

Las señales de salida son: EAR habilita registro de direcciones; EM habilita la celda de memoria para escritura; $X2C$ provoca que el registro de direcciones genera la dirección de dato R; y $X3C - \overline{X4C}$ hace que el multiplexor (ver Figura 6.6) de entrada a la memoria seleccione la fuente de datos MIDI que transmite un instrumento musical. Nótese que esta celda es la primera localizada después de aquella que, tiene una dirección W y está apuntada por W_p, esto último hace que el segmento esté activado.

Función lógica:

$$(CW \cdot (WW < AR) \cdot ADWI \cdot \overline{DSSI}) \Longrightarrow (EAR, EM, X2C, X3C, \overline{X4C})$$

Flujo de datos:

$$\begin{array}{ccccc} & \overbrace{\textit{status vacio}} & \overbrace{\textit{primer dato}} & & \\ \cdots & m_{n-1} & m_n & m_{n+1} & \cdots \\ \cdots & c_{n-1} & c_n & c_{n+1} & \cdots \\ \cdots & W & A \leftarrow R & A & \cdots \\ & \underbrace{W_p = A} & \underbrace{W_p < A} & \underbrace{W_p < A} & \end{array}$$

5.2.4. Estado 4

Una celda se encuentra en el estado 4 cuando sigue por la derecha a una celda W_a en el estado 2 (ver Figura 5.2). En el estado 4, una celda transforma su dirección de lectura R_a a una dirección de dato D_a, al ocurrir una escritura en el segmento W_a, indicando que es un dato en el segmento, pero no el primero (para lectura, el cual es identificado por la dirección R_a que se ha desplazado a la derecha). El estado 4 es la condición para recibir el segundo dato y subsecuentes en un segmento mientras haya espacio disponible en la memoria. Esto se indicaría (a través de la señal MEMRDY) mientras no sea accesada la última celda (función no implementada en esta versión).

Las señales de entrada son: CW habilita escritura; WW alimenta al apuntador W_p; AR alimenta la dirección A de esta celda; $ADWI$ indica que la celda anterior (c_{n-1}) tiene la dirección de escritura W activada; y $DSSI$ indica *status* ocupado, es decir, este segmento tiene por lo menos un dato.

Las señales de salida son: EAR habilita registro de direcciones; EM habilita el registro de memoria; $X1C$ provoca que el registro de direcciones genere la dirección de dato D; y $X3C - \overline{X4C}$ hace que el multiplexor de entrada a la memoria seleccione la fuente de datos MIDI que transmiten los instrumentos musicales. Obsérvese también que, esta celda es la primera localizada después de aquella que tiene una dirección W y está apuntada por W_p, esto último hace que el segmento esté activado y que la celda siguiente (c_{n+1}) deba copiar la dirección R que tenía esta celda (c_n) efectuando un desplazamiento: es decir, el nuevo dato empuja la información que está a la derecha.

Función lógica:

$$(CW \cdot (WW < AR) \cdot ADWI \cdot DSSI) \Longrightarrow (EAR, EM, X1C, X3C, \overline{X4C})$$

Flujo de datos:

$$\begin{array}{ccccc}
 & \overbrace{\hspace{2cm}}^{\textit{status ocupado}} & \overbrace{\hspace{1.5cm}}^{\textit{nuevo dato}} & \overbrace{\hspace{1.5cm}}^{\textit{primer dato}} & \\
\cdots & m_{n-1} & m_n & m_{n+1} & \cdots \\
\cdots & c_{n-1} & c_n & c_{n+1} & \cdots \\
\cdots & W & A \leftarrow D & A \leftarrow R & \cdots \\
 & \underbrace{W_p = A} & \underbrace{W_p < A} & \underbrace{W_p < A} &
\end{array}$$

5.2.5. Estado 5

Una celda c_n que está en el estado 5 hace una copia exacta de los valores de dirección y dato que tiene una celda c_{n-1} que le sigue por la izquierda. Para que se cumpla la condición del estado 5 es necesario, que la dirección A que identifica a esta celda c_n, sea menor que el apuntador de escritura W_p del segmento al que se desea escribir el dato. Esto significa que las celdas que tienen el estado 5 se encuentran a partir de le tercera celda de un segmento y terminan con la última celda de este mis-

mo segmento. El efecto resultante en dichas celdas es un desplazamiento de información hacia la derecha.

Las señales de entrada son: CW habilita escritura; WW alimenta al apuntador W_p; AR alimenta a la dirección A de esta celda; y $\overline{ADWI}$ indica que la celda anterior (c_{n-1}) es dirección de escritura W inactivada.

Las señales de salida son: EAR habilita registro de direcciones; EM habilita registro de memoria; $\overline{X0C}$ provoca que el registro de direcciones haga desplazamiento hacia adelante copiando la dirección que tiene la celda anterior (c_{n-1}); y $\overline{X3C}$-$\overline{X4C}$ hace que el multiplexor de entrada a la memoria seleccione el dato que tiene la celda precedente (c_{n-1}) para absorberlo.

Función lógica:

$$(CW \cdot (WW < AR) \cdot \overline{ADWI}) \Longrightarrow (EAR, EM, \overline{X0C}, \overline{X3C}, \overline{X4C})$$

Flujo de datos:

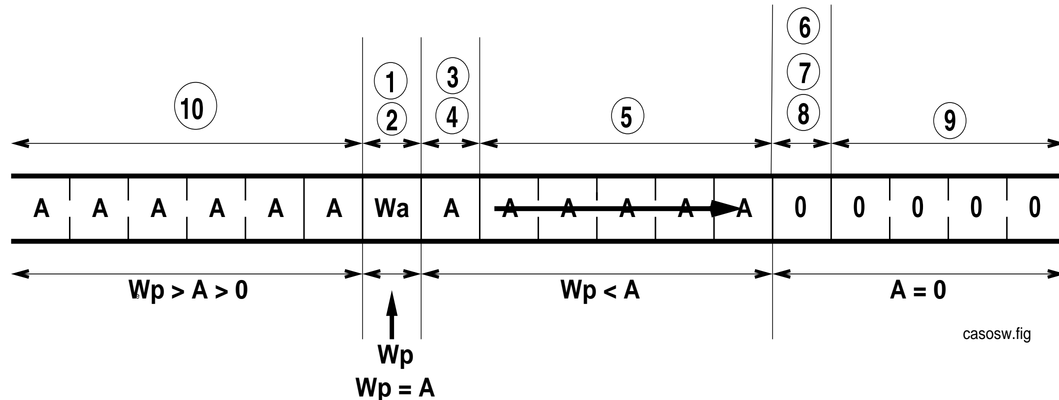
$$\begin{array}{ccccc} & \underbrace{d(m_{n-1}) \leftarrow d(m_{n-2})} & \underbrace{d(m_n) \leftarrow d(m_{n-1})} & \underbrace{d(m_{n+1}) \leftarrow d(m_n)} & \\ \cdots & m_{n-1} & m_n & m_{n+1} & \cdots \\ \cdots & c_{n-1} & c_n & c_{n+1} & \cdots \\ \cdots & A(c_{n-1}) \leftarrow A(c_{n-2}) & A(c_n) \leftarrow A(c_{n-1}) & A(c_{n+1}) \leftarrow A(c_n) & \cdots \\ & \overbrace{W_p < A} & \overbrace{W_p < A} & \overbrace{W_p < A} & \end{array}$$

5.2.6. Estado 6

Continuando con nuestra descripción de los estados de escritura, y para una mejor aclaración de las ideas que presentamos, se muestra nuevamente la Figura 5.2.

Cuando una celda c_n se encuentra en el estado 6, significa que debe realizar una copia exacta de los valores de dirección y dato que tiene la celda de la izquierda c_{n-1}. La celda c_n que cumple esta condición es la primera celda vacía que se encuentra al final del último segmento. Esto quiere decir que la dirección A de esta celda c_n debe ser igual a cero y además, la dirección A de la al celda c_{n-1} que está a su izquierda, debe ser la primera dirección W_a del último segmento. Cabe señalar que en este seg-

mento no se recibirá el dato, pues el resultado de este estado sólo será un efecto de desplazamiento de información hacia la derecha.



Las señales de entrada son: CW habilita escritura; AR alimenta la dirección A de esta celda (cero); y WI indica que la celda anterior c_{n-1} tiene dirección de escritura W inactiva.

Las señales de salida son: EAR habilita registro de direcciones; EM habilita registro de memoria; $\overline{X0C}$ provoca que el registro de direcciones haga desplazamiento hacia adelante copiando la dirección que tiene la celda anterior c_{n-1}; y $\overline{X3C}$-$\overline{X4C}$ hace que el multiplexor de entrada a la memoria seleccione el dato que tiene la celda precedente c_{n-1} para absorberlo.

Función lógica:

$$(CW \cdot (AR = 0) \cdot \overline{WI}) \Longrightarrow (EAR,\, EM,\, \overline{X0C},\, \overline{X3C},\, \overline{X4C})$$

Flujo de datos:

$$\begin{array}{ccccc}
 & \underbrace{d(m_{n-1}) \leftarrow d(m_{n-2})} & \underbrace{d(m_n) \leftarrow d(m_{n-1})} & \underbrace{d(m_{n+1}) \leftarrow d(m_n)} & \\
\cdots & m_{n-1} & m_n & m_{n+1} & \cdots \\
\cdots & c_{n-1} & c_n & c_{n+1} & \cdots \\
\cdots & W & A \leftarrow W & A \leftarrow 0 & \cdots \\
 & \overbrace{W_p < A} & \overbrace{A = 0} & \overbrace{A = 0} &
\end{array}$$

5.2.7. Estado 7

Una celda c_n que está en el estado 7 hace una copia exacta de los valores de dirección y dato de la celda c_{n-1} que está a su izquierda. La celda c_n que cumple con esta condición es la primera celda vacía que se encuentra al final del último segmento. Esto significa que la dirección A de esta celda c_n es igual a cero, y además la dirección A de la celda c_{n-1} que está a su izquierda debe tener el valor de la última dirección de dato R_a del último segmento. Ya sea que este segmento (el último) reciba o no el dato, sólo se tendrá el efecto de desplazamiento de información hacia la derecha.

Las señales de entrada son: CW habilita escritura; AR alimenta la dirección A de esta celda (cero); y RI indica que la celda precedente c_{n-1} tiene dirección de lectura R.

Las señales de salida son: EAR habilita registro de direcciones; EM habilita registro de memoria; $\overline{X0C}$ provoca que el registro de direcciones haga desplazamiento hacia adelante copiando la dirección que tiene la celda anterior c_{n-1}; y $\overline{X3C}$-$\overline{X4C}$ hace que el multiplexor de entrada a la memoria seleccione el dato que tiene la celda precedente c_{n-1} para absorberlo.

Función lógica:

$$(CW \cdot (AR = 0) \cdot RI) \Longrightarrow (EAR,\, EM,\, \overline{X0C},\, \overline{X3C},\, \overline{X4C})$$

Flujo de datos:

$$\begin{array}{ccccc}
 & \underbrace{d(m_{n-1}) \leftarrow d(m_{n-2})} & \underbrace{d(m_n) \leftarrow d(m_{n-1})} & \underbrace{d(m_{n+1}) \leftarrow d(m_n)} & \\
\cdots & m_{n-1} & m_n & m_{n+1} & \cdots \\
\cdots & c_{n-1} & c_n & c_{n+1} & \cdots \\
\cdots & R & A \leftarrow R & A \leftarrow 0 & \cdots \\
 & \overbrace{W_p < A} & \overbrace{A = 0} & \overbrace{A = 0} &
\end{array}$$

5.2.8. Estado 8

El estado 8 se presenta cuando se escribe el primer dato en el último segmento de la memoria. La celda que cumple esta condición se encuentra al final del último segmento y es la primera celda vacía de la memoria. Esta celda cambia automáticamente su dirección vacía (cero) generando ella misma una dirección de dato R_a. Para

que el estado 8 se cumpla, la dirección A que identifica esta celda c_n debe ser igual a cero, y además la celda de la izquierda c_{n-1} debe tener una dirección de escritura W_a que pertenece al último segmento de la memoria.

Las señales de entrada son: CW habilita escritura; AR alimenta la dirección A de esta celda c_n; $ADWI$ señala que la celda anterior (c_{n-1}) tiene su dirección de escritura W activada; y $\overline{DSSI}$ manifiesta *status* vacío, es decir, este segmento no contiene datos.

Las señales de salida son: EAR habilita registro de direcciones; EM habilita registro de memoria; $X2C$ provoca que el registro de direcciones genere la dirección de dato R; y $X3C - \overline{X4C}$ hace que el multiplexor de entrada a la memoria seleccione la fuente de datos MIDI que transmite un instrumento musical. Nótese que esta celda es la primera localizada después de aquella que, tiene una dirección W y está apuntada por W_p, esto último hace que el segmento esté activado.

Función lógica:

$$(CW \cdot (AR = 0) \cdot ADWI \cdot \overline{DSSI}) \Longrightarrow (EAR, EM, X2C, X3C, \overline{X4C})$$

Flujo de datos:

$$\begin{array}{cccccc} & \overbrace{\qquad}^{status\ vacio} & \overbrace{\qquad}^{primer\ dato} & & \\ \cdots & m_{n-1} & m_n & m_{n+1} & \cdots \\ \cdots & c_{n-1} & c_n & c_{n+1} & \cdots \\ \cdots & \underbrace{W}_{W_p = A} & \underbrace{0 \leftarrow R}_{W_p < A} & \underbrace{0}_{W_p < A} & \cdots \end{array}$$

5.2.9. Estado 9

Cuando una celda se encuentra en el estado 9, ésta permanece estática o quieta, es decir, no hace ningún movimiento. A fin de que se cumpla esta condición, el valor de la dirección A que identifica a esta celda c_n debe ser igual a cero, además no debe haber a la izquierda una celda c_{n-1} que se encuentre identificada con una dirección A de escritura o de lectura (W_a o R_a). Las celdas que cumplen esta condición

se encuentran localizadas después (a la derecha) de la primera celda vacía del último segmento que recibirá el dato.

Las señales de entrada son: CW habilita escritura; AR alimenta la dirección A de esta celda (cero); WI señala que la celda anterior c_{n-1} no tiene dirección de escritura W; y RI indica que la celda anterior c_{n-1} no tiene dirección de lectura R.

Las señales de salida son: Ninguna, no hay movimiento, las celdas permanecen estáticas. Todos los valores de salida son iguales a cero.

Función lógica:

$$(CW \cdot (AR = 0) \cdot \overline{WI} \cdot \overline{RI}) \Longrightarrow (0)$$

Flujo de datos:

$$\begin{array}{ccccc} & \underbrace{estatica} & \underbrace{estatica} & \underbrace{estatica} & \\ \cdots & m_{n-1} & m_n & m_{n+1} & \cdots \\ \cdots & c_{n-1} & c_n & c_{n+1} & \cdots \\ \cdots & 0 & 0 & 0 & \cdots \\ & \overbrace{W_p > A} & \overbrace{W_p > A} & \overbrace{W_p > A} & \end{array}$$

5.2.10. Estado 10

Cuando una celda se encuentra en el estado 10, ésta permanece estática o quieta, es decir, no hace ningún movimiento. Para que esto se cumpla, el valor de A que identifica cualquier dirección (W_a, D_a, o R_a) de una celda c_n, debe ser mayor que cero y menor al valor del apuntador de escritura W_p. Las celdas c_n que cumplen esta condición se encuentran localizadas antes (a la izquierda) de la primera celda del segmento que recibirá el dato.

Las señales de entrada son: CW habilita escritura; AR alimenta la dirección A de esta celda (no cero); y WW alimenta el apuntador W_p.

Las señales de salida son: Ninguna, no hay movimiento, las celdas permanecen inmóviles. Todos los valores de salida son iguales a cero.

Función lógica:

$$(CW \cdot (\overline{AR = 0}) \cdot (AR < WW)) \Longrightarrow (0)$$

Flujo de datos:

$$
\begin{array}{ccccc}
 & \underbrace{estatica} & \underbrace{estatica} & \underbrace{estatica} & \\
\cdots & m_{n-1} & m_n & m_{n+1} & \cdots \\
\cdots & c_{n-1} & c_n & c_{n+1} & \cdots \\
\cdots & A & A & A & \cdots \\
 & \overbrace{W_p > A > 0} & \overbrace{W_p > A > 0} & \overbrace{W_p > A > 0} &
\end{array}
$$

5.3. Estados de lectura

En estos casos el control general le dice a la memoria MSA que se va a leer un dato activando la señal $CR = 1$. Además, también coloca el valor de uno de los apuntadores de lectura R_p sobre el *bus* de apuntadores a la memoria [22]. Al igual que en los estados de escritura, para cada estado se describe en detalle: una operación de lectura de cada celda de memoria, una función lógica que determina su comportamiento, y también una matriz que muestra el flujo de datos.

En la Figura 5.3 se muestran los estados de lectura en que pueden estar cada una de las celdas. Recordemos que A es la dirección que identifica a una celda c_n, R_p es uno de los apuntadores de lectura, y que los dígitos $11, 12, \ldots, 15, 16$ encerrados en círculos son los estados de lectura que se han definido de acuerdo con la Tabla 5.2.

De la misma forma, y para los estados que restan, la parte izquierda de la función lógica es una expresión booleana de las señales de entrada, y la parte derecha son las señales producidas. Cada función se ha diseñado de acuerdo con la Tabla 5.2.

5.3.1. Estado 11

La celda que está en el estado 11 contiene almacenado el primer dato de un segmento. El control de esta celda c_n habilita una señal que permite la salida del dato colocándolo sobre el *bus* de datos externo para que sea leído por el procesador. La celda que cumple con esta condición es la última de un segmento que contiene datos. Además de habilitar la salida del dato, esta celda c_n hace una copia exacta de los valores de dirección y dato que tiene la celda de la derecha c_{n+1}. El resultado es un efecto

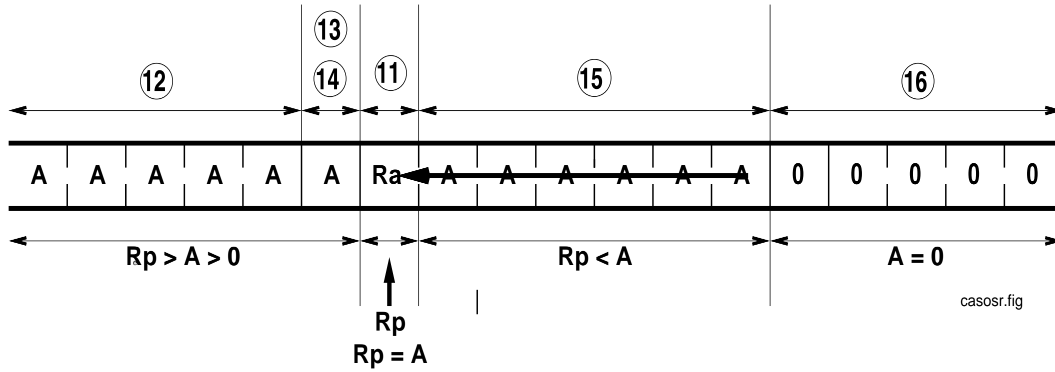


Figura 5.3: Estados de lectura de las celdas de memoria.

de desplazamiento de información hacia la izquierda después que se haya leído el dato.

Las señales de entrada son: CR habilita lectura; AR alimenta la dirección A de esta celda; y RR alimenta el apuntador de lectura R_p.

Las señales de salida son: EAR habilita el registro de direcciones; EM habilita la celda de memoria; $X0C$ provoca desplazamiento hacia atrás copiando la dirección que tiene la celda siguiente; $X3C$-$X4C$ provoca desplazamiento hacia atrás copiando el dato que tiene la celda siguiente; $ATRO$ habilita la salida del dato; y $AURO$ indica a la celda anterior (c_{n-1}) que esta celda tiene una dirección de lectura R activada.

Función lógica:

$$(CR \cdot (AR = RR)) \Longrightarrow (EAR,\ EM,\ X0C,\ X3C,\ X4C,\ ATRO,\ AURO)$$

Flujo de datos:

$$\begin{array}{ccccc} & & \overbrace{\underbrace{d(c_n) \leftarrow d(c_{n+1})}}^{\textit{dato disponible}} & \underbrace{\textit{dato}} & \\ \cdots & m_{n-1} & m_n & m_{n+1} & \cdots \\ \cdots & c_{n-1} & c_n & c_{n+1} & \cdots \\ \cdots & \overbrace{A}_{R_p > A > 0} & \overbrace{R \leftarrow A(c_{n+1})}_{R_p = R} & \overbrace{A}_{R_p < A} & \cdots \end{array}$$

5.3.2. Estado 12

Cuando una celda c_n se encuentra en el estado 12, ésta permanece estática o quieta, es decir, no hace ningún movimiento. Para que esto se cumpla, el valor de la dirección A (W_a, D_a, o R_a) de una celda, debe ser menor que el valor del apuntador de escritura R_p, y también mayor que cero. Cualquier celda que se encuentre a la derecha de una celda c_n debe tener una dirección D_n. Todas las celdas que cumplen con este estado se encuentra localizadas una celda después (hacia la izquierda) de la celda que contiene el dato a leer.

Las señales de entrada son: CR habilita lectura; AR alimenta la dirección A de esta celda; RR alimenta el apuntador R_p; y $AURI$ indica que la celda posterior c_{n+1} no es R activada.

Las señales de salida son: iguales a cero, no hay movimiento.

Función lógica:

$$(CR \cdot (\overline{AR = 0}) \cdot (AR < RR) \cdot \overline{AURI}) \Longrightarrow (0)$$

Flujo de datos:

$$\begin{array}{ccccc} & \overbrace{}^{estatica} & \overbrace{}^{estatica} & \overbrace{}^{estatica} & \\ \cdots & m_{n-1} & m_n & m_{n+1} & \cdots \\ \cdots & c_{n-1} & c_n & c_{n+1} & \cdots \\ \cdots & A & A & A & \cdots \\ & \underbrace{R_p > A > 0} & \underbrace{R_p > A > 0} & \underbrace{R_p > A > 0} & \end{array}$$

5.3.3. Estado 13

En este estado, una celda c_n cambia su estado *no vacío* a un estado *vacío*. Esta condición se cumple cuando el procesador lee el último dato de un segmento. La celda c_n en cuestión se reestablecerá generando una dirección W_a de escritura, y colocando también, la bandera de *status* que indica al sistema de control que este segmento de memoria ha quedado desocupado. Esta celda es la primera que se encuentra a la izquierda de la celda que contiene el dato que se va a leer.

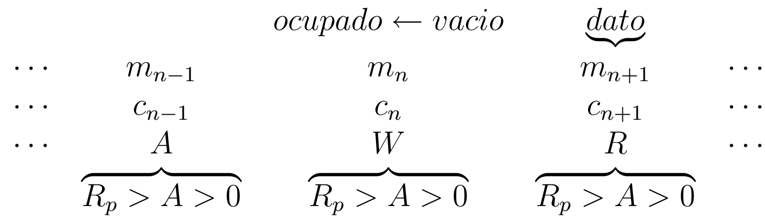
Las señales de entrada son: CR habilita lectura; AR alimenta la dirección A diferente de cero de esta celda; RR alimenta el apuntador R_p; WL indica que esta celda puede tener una dirección W; $AURI$ indica que la celda siguiente (c_{n+1}) contiene una dirección R_a activa.

Las señales de salida son: EM habilita la memoria; y MR hace que se coloque un valor de dato igual a 00_H en la celda de memoria.

Función lógica:

$$(CR \cdot (\overline{AR = 0}) \cdot (AR < RR) \cdot WL \cdot AURI) \Longrightarrow (EM, MR)$$

Flujo de datos:



5.3.4. Estado 14

La celda que se encuentra en este estado convierte su dirección actual en una dirección R, y también realiza una copia del dato que se encuentra almacenado en la celda de la derecha siguiente. Esta operación tiene el efecto de hacer un desplazamiento de direcciones y datos hacia la izquierda. La celda que cumple con este estado es la primera que está a la izquierda de la celda que contiene el dato que se va a leer. El valor de su dirección debe ser menor que el valor del apuntador de escritura R_p y mayor que cero.

Las señales de entrada son: CR habilita la lectura; AR alimenta la dirección A de esta celda; RR alimenta el apuntador R_p; $\overline{WL}$ indica que esta celda no contiene una dirección W; y $AURI$ indica que la celda siguiente (c_{n+1}) contiene una dirección R activa.

Las señales de salida son: EAR habilita el registro de direcciones; y $X0C$ hace desplazamiento izquierdo.

Función lógica:

$$(CR \cdot (\overline{AR = 0}) \cdot (AR < RR) \cdot \overline{WL} \cdot AURI) \Longrightarrow (EAR,\, X0C)$$

Flujo de datos:

$$\begin{array}{ccccc} & & \overbrace{d(m_n) \leftarrow d(m_{n+1})} & \overbrace{dato} & \\ \cdots & m_{n-1} & m_n & m_{n+1} & \cdots \\ \cdots & c_{n-1} & c_n & c_{n+1} & \cdots \\ \cdots & A & A \leftarrow R & R & \cdots \\ & \underbrace{R_p > A > 0} & \underbrace{R_p > A > 0} & \underbrace{R_p > A > 0} & \end{array}$$

5.3.5. Estado 15

Todas las celdas que cumplen con esta condición solamente hacen desplazamiento de direcciones y de datos hacia la izquierda. Para que esto se cumpla, el valor de la dirección de una celda c_n debe ser mayor que el valor de uno de los apuntadores de lectura R_p. Las celdas en dicho estado están inmediatamente después a la derecha de la celda que contiene el dato que se va a leer.

Las señales de entrada son: CR habilita la lectura; AR alimenta la dirección A de esta celda; y RR alimenta el apuntador R_p.

Las señales de salida son: EAR habilita el registro de direcciones; $X0C$ provoca el desplazamiento de una dirección hacia la izquierda; y $X3C - X4C$ activa el multiplexor de entrada que contiene una celda de memoria con el fin de leer el dato que se encuentra almacenado en la celda de la derecha siguiente.

Función lógica:

$$(CR \cdot (AR > RR)) \Longrightarrow (EAR,\, EM,\, X0C,\, X3C,\, X4C)$$

Flujo de datos:

$$\begin{array}{ccccc} & \overbrace{d(m_{n-1}) \leftarrow d(m_n)} & \overbrace{d(m_n) \leftarrow d(m_{n+1})} & \overbrace{d(m_{n+1}) \leftarrow d(m_{n+2})} & \\ \cdots & m_{n-1} & m_n & m_{n+1} & \cdots \\ \cdots & c_{n-1} & c_n & c_{n+1} & \cdots \\ \cdots & A(c_{n-1}) \leftarrow A(c_n) & A(c_n) \leftarrow A(c_{n+1}) & A(c_{n+1}) \leftarrow A(c_{n+2}) & \cdots \\ & \underbrace{R_p < A} & \underbrace{R_p < A} & \underbrace{R_p < A} & \end{array}$$

5.3.6. Estado 16

Cuando una celda se encuentra en el estado 16, ésta no hace nada, permanece inmóvil. En este caso, la celda c_n es una celda vacía, es decir, contiene una dirección con valor de cero (Φ) y el dato que contiene almacenado es irrelevante. Todas las celdas en este estado pertenecen al espacio de memoria donde no existe información válida. Recordemos que la memoria se autocompacta con el fin de no dejar huecos entre los datos.

Las señales de entrada son: CR habilita la lectura; AR alimenta la dirección A de esta celda; $\overline{WL}$ indica que esta celda no contiene una dirección de escritura W; y $\overline{AURI}$ indica que la celda siguiente (c_{n+1}) no contiene una dirección R activa.

Las señales de salida son: Ninguna porque no hay movimiento; todas son iguales a cero.

Función lógica:

$$(CR \cdot (AR = 0) \cdot \overline{WL} \cdot \overline{AURI}) \Longrightarrow (0)$$

Flujo de datos:

$$\begin{array}{ccccc} & \overbrace{estatica} & \overbrace{estatica} & \overbrace{estatica} & \\ \cdots & m_{n-1} & m_n & m_{n+1} & \cdots \\ \cdots & c_{n-1} & c_n & c_{n+1} & \cdots \\ \cdots & 0 & 0 & 0 & \cdots \\ & \underbrace{A=0} & \underbrace{A=0} & \underbrace{A=0} & \end{array}$$

5.4. Resumen

Hemos presentado el detalle de diseño la memoria segmentada autocompactante; este diseño puede ser directamente implementado en un circuito integrado ASIC de propósito específico. Describimos cada uno de los estados por los que debe pasar una celda de memoria antes de que se haga una operación de escritura o lectura de un dato. Además, para cada estado en que se puede encontrar una celda antes de una operación de lectura o escritura, se presentó su descripción operativa, su función lógica

que determina su comportamiento, y una matriz gráfica que muestra el flujo de datos cuando se realiza la inserción o extracción de un dato específico.

Capítulo 6

Evaluación de nuestras propuestas: MCS-L y MCS-S

En los dos capítulos anteriores presentamos dos diseños para un sistema de captura de eventos MIDI en paralelo portátil y fácil de usar: MCS-L y MCS-S. En este capítulo presentaremos una evaluación de cada diseño.

6.1. Evaluación de MCS-L

Para evaluar el diseño MCS-L diseñamos e implementamos un prototipo del mismo. Este prototipo ha sido invaluable para entender mejor los requerimientos de velocidad de los componentes de una implementación en serie. Con nuestro prototipo hemos realizado pruebas de captura de datos, desarrollado las primitivas de comunicación, de manejo de memoria, y las funciones de mayor nivel que controlan otros componentes de hardware y software.

6.1.1. Prototipo

Nuestro prototipo de MCS-L lo implementamos sobre un sistema mínimo de computación, basado en un microcontrolador comercial de 8 bits, y al cual añadimos otros componentes. Para controlar el hardware de nuestro prototipo, desarrollamos un software ex profeso para ello. A continuación describimos el hardware y software de nuestro prototipo.

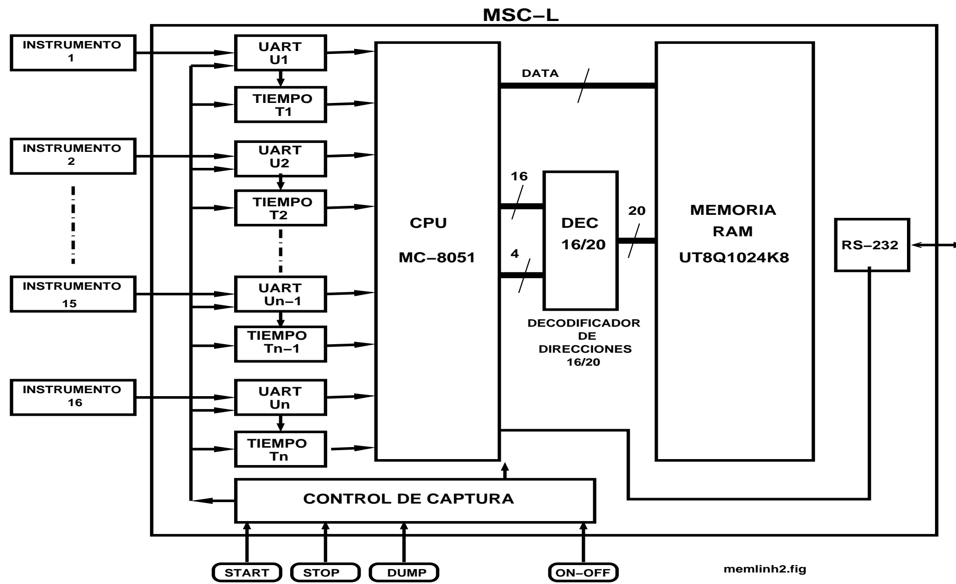


Figura 6.1: Componentes del CMIDIP con memoria lineal MCS-L.

Hardware

La Figura 6.1 muestra los principales componentes de nuestro prototipo MCS-L. Contiene un microcontrolador Intel 8051, con un *bus* de datos de 8 bits y un *bus* de direcciones de 16 bits [43]. Con direcciones de 16 bits se puede direccionar un máximo de 64 K localidades de memoria. Pero con el circuito decodificador de direcciones DECODER, es posible direccionar todo el espacio de direcciones de la memoria estática RAM UT8Q1024K8 de Aeroflex con capacidad de 1Mb de almacenamiento [25]. Nótese en la Figura 6.1, que del procesador al decodificador salen las 16 líneas del bus de direcciones y 4 líneas más, a través de las cuales se identifica uno de 16 bloques de 64 KB de memoria. Es decir, la dirección de 16 bits identifica una localidad dentro de un bloque seleccionado con aquellos 4 bits.

Con un 1 MB de memoria, MCS-L puede grabar continuamente 16 instrumentos por un periodo de hasta 60 minutos aproximadamente. Es un instrumento por cada uno de los 16 bloques de memoria disponibles. Experimentalmente hemos observado que una canción MIDI de 3:40 minutos de una orquesta con 15 instrumentos musicales ocupa más/menos 47KB [26]. Esto quiere decir, que si la misma canción tu-

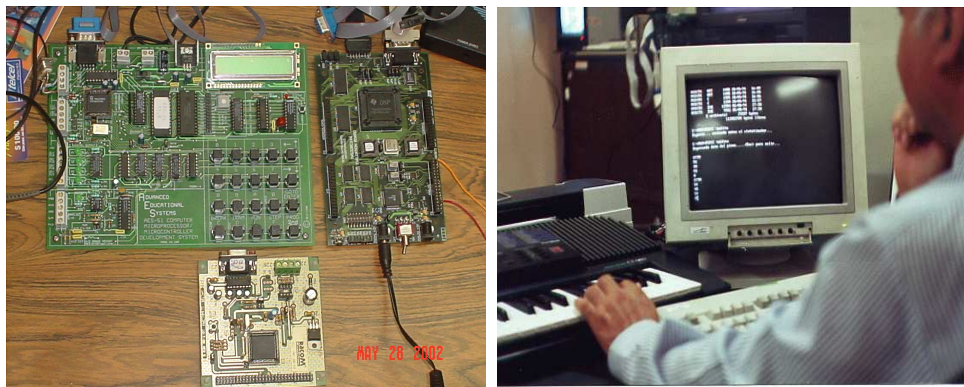


Figura 6.2: Prototipos MCS-L: foto izquierda, de izquierda a derecha, MCE8051 y DSP TMS320FC240, abajo HC11; foto derecha: captura de datos MIDI.

viera 60 minutos, ésta ocuparía entonces (60/3,4) 47 = 830 KB aproximadamente.

Las interfaces de hardware MIDI son 16, y consisten de 16 circuitos integrados UART 6850 de Motorola. Con cada UART se maneja un cuantificador de tiempo de duración de las notas MIDI que se han de recibir. Cada cuantificador de tiempo consiste de circuitos integrados discretos que funcionan como un contador digital. Cuando un UART recibe el comando de activación de una nota ($NoteOn$), se inicia el contador de tiempo correspondiente. Cuando el UART recibe el comando de desactivación de nota ($NoteOff$), el contador se detiene.

Además de nuestro prototipo descrito anteriormente, basado en el microcontrolador 8051, experimentamos con otros 2 prototipos, uno basado en el DSP (*Digital Signal Processor*) TMS 320FC240, de Texas Instrument, y otro basado en el microcontrolador HC11, de Motorola. Los 3 prototipos se muestran en la Figura 6.2, en la foto izquierda; la foto derecha muestra algunas pruebas de captura de datos MIDI con los mismos [15][80].

Los prototipos basados en el DSP TMS 320FC240 y en el HC11 pronto fueron abandonados por sus limitaciones de almacenamiento y escasos recursos de software de desarrollo. Ambos manejaban en ese entonces muy poca memoria, alrededor de 2 KB, y dicho software de desarrollo, además de ser costoso, generaría código objeto de

gran tamaño.

Software

Para controlar nuestro prototipo desarrollamos KLM (K*ernel for a smal*L M*onitor*), un pequeño sistema operativo escrito en lenguaje C y ensamblador que administra el hardware del MCS-L. KLM se compone de funciones que exploran los puertos de cada UART y extraen los datos MIDI de sus registros. Además, otras funciones de KLM almacenan los eventos MIDI en memoria, otras controlan el reloj interno de tiempo real del prototipo, otras controlan el manejo de interrupciones del reloj, etc. El desarrollo de estas funciones requirió la programación de funciones más básicas de: manejo de cadenas de caracteres, cálculo matemático, comparación de cadenas, y concatenación. Esto fue así porque el compilador C para el 8051 carecía de dichas funciones elementales [1][55][56].

Escritura y lectura de la memoria

El manejo de memoria se compone básicamente de dos primitivas, `poke(`*address*`,`*data*`)` y `peek(`*address*`)`, para escribir y leer un dato en memoria, respectivamente. Ambas controlan la interfaz de hardware que expande el espacio de direcciones natural del microcontrolador de 64 KB a 16 bloques de 64 KB cada uno. A través de estas primitivas, las demás funciones de KLM observan la memoria como si estuviera continua (virtualmente lineal) y con un tamaño de 1 MB. Ambas poseen un argumento común: *address*, variable de 20 bits codificada para seleccionar una de las direcciones de memoria (0,1,2, ..., 1048575).

El siguiente seudocódigo de `poke()` muestra cómo, con una dirección de 20 bits (almacenados en una variable de 32 bits), se obtiene el bloque de memoria respectivo (4 bits) y el desplazamiento dentro del mismo (16 bits).

función poke(*adreess*: entero_20_bits, *data*: byte_8_bits): entero
    inicio
    n: entero p: entero
        Extraer número de bloque: $n \leftarrow adreess(a_{16}, \ldots, a_{19})$.

Extraer desplazamiento: $p \leftarrow adreess(a_{15}, \ldots, a_0)$.
Seleccionar el bloque: $puerto \leftarrow n$.
Escribir en memoria: $p \leftarrow data$.
regresa(1)
fin

De la misma manera, el seudocódigo de `peek()` sirve para leer y regresar un dato localizado en una dirección de memoria (*address*). A partir de una dirección lineal, separa el número de bloque y el desplazamiento.

función peek(*adreess*: entero_20_bits): byte_8_bits
inicio
n: entero, *p*: entero, *data*: byte_8_bits;
Extraer número de bloque: $n \leftarrow adreess(a_{16}, \ldots, a_{19})$;
Extraer desplazamiento: $p \leftarrow adreess(a_{15}, \ldots, a_0)$;
Seleccionar bloque: $puerto \leftarrow n$;
Leer de memoria: $data \leftarrow p$;
regresa(*data*);
fin

Manejo de la memoria

El procedimiento principal `main()` de KLM, una vez que ha terminado con varias funciones de inicialización (de puertos, variables, etc.) se dedica a la captura y almacenamiento de eventos MIDI. Vimos anteriormente que el almacenamiento de eventos MIDI requiere un manejo de memoria tal que la información de los instrumentos queda separada utilizando varias estructuras FIFO, una para cada instrumento, o bien una sola estructura, pero adicionando una etiqueta (8 bits) a cada dato que se captura a fin de identificar al instrumento que lo genera. KLM implementa este último; porque no se desperdicia memoria si un instrumento no genera eventos MIDI, aunque se desperdicia un poco por la etiqueta que requiere cada evento MIDI para identificarlo en la lista global. A continuación mostramos el seudocódigo del `main()` relacionado con la captura y almacenamiento de eventos MIDI; en el mismo, $[ID_i][DATA_i]$ representa un evento MIDI del instrumento i precedido por su etiqueta.

function main(): integer
begin

```
    n, DATA: integer;
    while 1
    for n = 0; n < NUM_INSTRUMENTOS; n++
    if midready(n) {
       DATA_n=midiread(n)
       if DATA_n=NOTEON_n then
          Arrancar contador TIMER_n de duración de nota;
          Almacenar identificador y comando:
          MEM ← ID_n, MEM ← NOTEON_n;
          Almacenar identificador y dato (nota o volumen):
          MEM ← ID_n, MEM ← DATA_n;
       else
       if DATA_n=NOTEOFF_n then
          Detener contador TIMER_n de duración de nota;
          Almacenar identificador y comando:
          MEM ← ID_n, MEM ← NOTEOFF_n;
          Almacenar identificador y dato (nota o volumen):
          MEM ← ID_n, MEM ← DATA_n;
          Calcular duración de la nota DELTATIME_n;
          Almacenar identificador y duración de nota:
          MEM ← ID_n, MEM ← DELTATIME_n;
       else continue;
       }
    regresa(1);
end main
```

KLM utiliza exploración de puertos (*polling*) para capturar eventos MIDI. (El uso de interrupciones es más complicado.) El procedimiento `midready()` detecta si un instrumento ha enviado un dato, en cuyo caso `midiread()` realiza su lectura y lo regresa en el valor de la función.

```
function midready(n:integer): integer
begin
   n: integer;
   Seleccionar puerto MIDI n y leer bandera del registro de control,
   verifica que exista un dato listo para leer:
   FLAG:=RX_CTL_MIDIPORT_n;
   if FLAG=TRUE return TRUE;
   else         return FALSE;
   endif;
end midready;
```

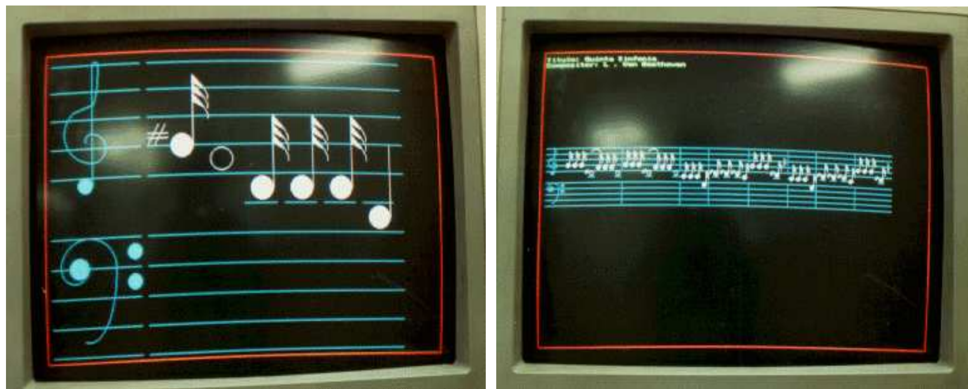


Figura 6.3: Desplegado experimental de datos MIDI.

function midiread(n:integer): integer
begin
  n: integer;
  Seleccionar y leer el registro de recepción
  de datos que tiene el puerto $MIDI\ n$:
  $DATA$:=$RX_MIDIPORT_n$;
  return($DATA$);
end midiread;

6.1.2. Resultados

Con nuestro prototipo de MCS-L realizamos la captura de los eventos MIDI generados por varios instrumentos. La datos así capturados los bajamos entonces a una PC, en la cual post-procesamos dichos datos para *generar las partituras de la música* tocada por los instrumentos. Para realizar este post-procesamiento diseñamos un compilador que traduce tales eventos MIDI, con base a otra información recibida por el usuario, en partituras con calidad profesional. Este compilador, llamado KL (K*ernel for a music* L*anguage*), lo describiremos en la siguiente subsección. La Figura 6.3 muestra las primeras partituras gráficas experimentales que fueron producidas por KL a partir de los datos capturados con nuestro prototipo de MCS-L.

6.1.3. Otros resultados: nuestro compilador KL

La Figura 6.4 muestra la partitura *politécnico* [82], la cual fue generada con nuestro compilador KL. Su archivo fuente (poli.kl) fue escrito con un editor de propósito general. Desarrollamos KL para poder escribir partituras de calidad profesional. KL está escrito en lenguaje C; en el anexo C.5, página 146, se describe KL de manera formal. Su entrada puede ser un archivo de eventos MIDI, un archivo de comandos del lenguaje natural de solfeo, por ejemplo, *do*, *re*, *mi*, *fa*, *sol la*, *si* (el Anexo C.5, página 141, muestra la entrada correspondiente a la partitura mostrada en la Figura 6.4). Todas las entradas que recibe KL son convertidas en comandos (de símbolos gráficos de partituras) para PCL (*Printer Command Language*), HPGL (*Hewlett-Packard Graphics Language*), PostScript, TEX, y MIDI [107].

Además, KL acepta instrucciones de control que le dicen qué instrumento produjo una serie de eventos MIDI, e instrucciones de control gráfico para indicar elementos musicales como: ligaduras, corcheas, silencios, tresillos, y otras que tienen un fin estético. Por ejemplo, para describir el segmento de una melodía, se escribe y se edita un texto normal de caracteres ASCII como la siguiente cadena de sílabas: "`do do mi sol do do♯ re♭`".

Cabe recordar que MIDI2TEX puede convertir un archivo MIDI (con muchas pistas) a instrucciones de MusicTEX para obtener una partitura gráfica [61][70]. Sin embargo, a diferencia de nuestro KL, MIDI2TEX no genera partituras estéticamente legibles, ya que omite símbolos como ligaduras, puntos de extensión, entre otros. Esto se debe a la poca información que contiene un conjunto de eventos MIDI. En particular, no se sabe que instrumento los generó. En el Anexo C, página 145, la Figura C.3 muestra otra partitura (*minena*) procesada con KL.

POLITÉCNICO

D. Pérez-Prado

2-12-76

Sax Alto

Figura 6.4: Salida KL del archivo `poli.kl` compilado (listado: Anexo C.5).

6.2. Evaluación de MCS-S

Para evaluar el diseño de MCS-S diseñamos e implementamos una simulación de sus componentes que son distintos a los componentes de MCS-L. MCS-S y MCS-L difieren solamente en la memoria donde se almacenan los eventos MIDI de cada instrumento; todos los demás componentes pueden ser iguales en una implementación. Se recordará que MCS-S utiliza una memoria segmentada autocompactante (MSA) que facilita el manejo de las listas FIFO donde se almacenan los eventos MIDI de varios instrumentos. El procesador sólo tiene que escribir los datos de un evento MIDI especificando la lista del instrumento correspondiente. Por esta razón, nuestra evaluación de MCS-S se concentra en verificar el funcionamiento correcto de la memoria MSA.

Hoy en día el diseño e incluso la fabricación de un circuito integrado se realiza con paquetes de software especializado. Con estos paquetes es posible diseñar un *chip* que contendrá millones de transistores dentro en una pequeña área física: VLSI (*Very Large Scale Integration*), probar la correctitud del diseño y finalmente descargar el diseño en un circuito integrado de aplicación específica ASIC (*Application Specific Integrated Circuit*) [98]. Existen diferentes tipos de ASICs. Algunos FPGAs (*Field Programmable Gate Array*) pueden ser reprogramados/grabados más de una vez; pero la mayoría de los VLSIs sólo pueden ser grabados solo una vez. Para verificar el funcionamiento correcto de la MSA, utilizamos el paquete de desarrollo de circuitos integrados Xilinx.

6.2.1. Xilinx

Xilinx es un paquete que permite el diseño, simulación (verificación) e implementación (descarga) de circuitos integrados. Xilinx es un medio ambiente interactivo que puede recibir como entrada la especificación de un circuito en lenguaje VHDL, el cual es muy parecido al lenguaje Pascal. Xilinx también ofrece un editor gráfico de componentes esquemáticos digitales (así como una biblioteca VHDL de los mismos)

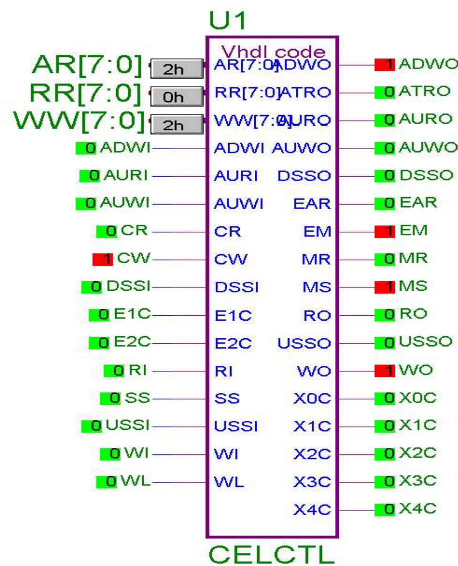


Figura 6.5: Diagrama esquemático de una celda de control de la MSA.

con los cuales el usuario puede "ensamblar"el diseño de su circuito digital. Una vez que se ha leído o editado un circuito digital, es posible simular su comportamiento. Finalmente, el circuito o sistema digital se puede descargar a un ASIC [105][106].

6.2.2. Simulación de las celdas de control de la MSA

Se recordará que MSA está formada por una sucesión consecutiva de celdas de memoria m_i, cada una de las cuales es controlada por una celda de control c_i. La celda de control es el elemento más complejo de la MSA. Al escribir o leer un dato de la memoria, cada celda de control genera señales internas y externas para que los datos y direcciones de toda la memoria se desplacen una posición a la derecha o la izquierda, respectivamente. Describiendo el funcionamiento de las celdas de control se describe la mayor parte del funcionamiento de la MSA. En esta subsección describimos sólo la simulación de las celdas de control.

La Figura 6.5 muestra el diagrama esquemático de una celda de control c_i de la memoria MSA. La especificación funcional de la misma se escribió en lenguaje VHDL; después se simuló para verificar su funcionamiento visualizando la forma de onda de

sus salidas. Esta figura muestra también las terminales de entrada y salida por las que una celda de control recibe señales de control de, y produce señales de control para, celdas de control vecinas. Las señales de control producidas por una celda dependen del estado en que se encuentra, y en el capítulo 5 vimos que existen 10 estados de escritura y 6 de lectura.

El estado 1, de escritura (pág. 55), se presenta cuando se inserta un dato en un segmento (vacío); su función lógica es:

$$(CW \cdot (WW = AR) \cdot \overline{SS}) \Longrightarrow (EM,\, MS,\, ADWO,\, \overline{DSSO})$$

Las señales de entrada son: CW habilita escribir dato; WW alimenta al apuntador de segmento W_p; AR alimenta la dirección A de la celda (en esta caso W_a); y $\overline{SS}$ indica *status* de segmento vacío.

Las señales de salida son: EM habilita la celda de memoria para la operación de escritura; MS escribe "FFH"en un registro de control para indicar a las celdas vecinas(c_{n+1} y c_{n-1}) que este segmento ya no está vacío; SS es una copia del bit más significativo de este registro e indica lo mismo a la celda W_a (c_n); $ADWO$ dice a la celda siguiente (c_{n+1}) que aquí hay una dirección de escritura W activada; y $\overline{DSSO}$ es una copia del *status* anterior $\overline{SS}$ indicando a la celda siguiente (c_{n+1}) que esta región de memoria no contenía información. El código VHDL de una celda control, correspondiente al estado 1 de escritura, se muestra a continuación:

```
library ieee;
use ieee.std_logic_1164.all;

entity CELCTL is
port(
 CW,CR,E1C,E2C,WI,RI,WL,SS: in  std_logic;
  AUWI,AURI,ADWI,USSI,DSSI: in  std_logic;
                  WW,RR,AR: in  std_logic_vector(7 downto 0);
        EAR,EM,WO,RO,MS,MR: out std_logic;
       X0C,X1C,X2C,X3C,X4C: out std_logic;
       ATRO,AUWO,AURO,ADWO: out std_logic;
 USSO,DSSO: out std_logic
);
end CELCTL;
```

```
architecture operacion of CELCTL is
signal vsal: std_logic_vector(16 downto 0);
begin  process (CW,CR,WW,AR,RR,SS,ADWI,AURI,DSSI,WI,RI,vsal) begin
       --
       -- estado 1 de escritura.
  if  (CW='1' and AR=WW    and SS='0') then
       vsal<="01001000000000100";
       -- estado 2 de escritura
  .
  .
  .
       -- estado 16 de escritura
  .
  .
  .
```

Las primeras dos líneas incluyen varias definiciones estándar de componentes en el diseño de circuitos eléctricos. La línea 3 declara una celda de control CELCTL como un circuito integrado. De la línea `port(` a la línea `);` se declaran las terminales de entrada y salida de la celda de control. Nótese que estas corresponden a las señales de entrada y salida utilizadas en la descripción de la función lógica del estado descrito anteriormente. Las siguientes 3 líneas anuncian que sigue la especificación de la operación de la celda de control; las líneas que empiezan con "–" son comentarios.

La línea "if (CW='1' and AR=WW and SS='0') ..." y la siguiente corresponden a la especificación funcional en VHDL del estado 1 de escritura. Nótese que la expresión lógica de la instrucción `if` corresponde a la parte izquierda de $\Longrightarrow$ en la función lógica de dicho estado, la cual repetimos a continuación por conveniencia:

$$(CW \cdot (WW = AR) \cdot \overline{SS}) \Longrightarrow (EM, MS, ADWO, \overline{DSSO})$$

La parte derecha o producción de esta función lógica corresponde al bloque `then` de la instrucción `if`:

```
vsal<="01001000000000100";
```

La cadena de dígitos binarios (0 ó 1) corresponde a la activación o desactivación de la terminales de salida especificadas con la declaración `port(... out ... out ...)`. El primer dígito binario a la izquierda corresponde a

la primera terminal de salida declarada (EAR), el segundo dígito corresponde a la terminal de salida EM. Se notará que, de acuerdo con la producción de la función lógica, se están activando EM, MS, $ADWO$ y $\overline{DSSO}$.

El código VHDL de una celda de control, correspondiente a los estados escritura y lectura restantes, se muestra a continuación. Nótese que los estados 11–16 son estados de lectura:

```
       -- estado 1.
  if  (CW='1' and AR=WW    and SS='0') then
       vsal<="0100100000000100";
       -- estado 2.
elsif (CW='1' and AR=WW    and SS='1') then
       vsal<="0000000000000101";
       -- estado 3.
elsif (CW='1' and AR>WW    and ADWI='1' and DSSI='0') then
       vsal<="1100000110000000";
       -- estado 4.
elsif (CW='1' and AR>WW    and ADWI='1' and DSSI='1') then
      vsal<="1100001010000000";
       -- estado 5.
elsif (CW='1' and AR/=X"00" and AR>WW   and ADWI='0') then
       vsal<="1100000000000000";
       -- estado 6, no se da WI siempre 0.
elsif (CW='1' and AR=X"00" and WI='1') then
       vsal<="1100000000000000";
       -- estado 7 no se da RI siempre 0.
elsif (CW='1' and AR=X"00" and RI='1') then
       vsal<="1100000000000000";
       -- estado 8.
elsif (CW='1' and AR=X"00" and ADWI='1' and DSSI='0') then
       vsal<="1100000110000000";
       -- estado 9.
elsif (CW='1' and AR=X"00" and WI='0'   and RI='0') then
       vsal<="0000000000000000";
       -- estado 10.
elsif (CW='1' and AR/=X"00" and AR<WW) then
       vsal<="0000000000000000";
       -- estado 11.
elsif (CR='1' and AR=RR) then
       vsal<="1100001001110100";
       -- estadao 12.
elsif (CR='1' and AR/=X"00" and AR<RR and AURI='0') then
       vsal<="0000000000000000";
       -- estado 13 falta WL=1 ?
--elsif (CR='1' and AR/=X"00" and AR<RR and AURI='1') then
       -- vsal<="0100010000000000";
       -- estado 14 falta WL=0?
elsif (CR='1' and AR/=X"00" and AR<RR and AURI='1') then
       vsal<="1000001000000000";
```

```
       -- estado 15.
elsif (CR='1' and AR>RR) then
       vsal<="11000010011000000";
       -- estado 16.
elsif (CR='1' and AR=X"00" and AURI='0') then
       vsal<="00000000000000000";
else
       vsal<="00000000000000000";
end if;
    --Calcula WO
if (AR(1)='1' and AR(0)='0') then
    WO<='1'; else WO<='0';
end if;
    --Calcular RO
if (AR(2)='1' and AR(1)='0' and AR(0)='0') then
    RO<='1'; else RO<='0';
end if;
-- Salida de variables.
   EAR <=vsal(16);
   EM <=vsal(15);
-- WO <=vsal(14);
-- RO <=vsal(13);
   MS <=vsal(12);
   MR <=vsal(11);
   X0C <=vsal(10);
   X1C <=vsal(9);
   X2C <=vsal(8);
   X3C <=vsal(7);
   X4C <=vsal(6);
  ATRO <=vsal(5);
  AUWO <=vsal(4);
  AURO <=vsal(3);
  ADWO <=vsal(2);
  USSO <=vsal(1);
  DSSO <=vsal(0);
end process;
end operacion;
```

6.2.3. Simulación de los componentes restantes de la MSA

La Figura 6.6 muestra todos los componentes esquemáticos de una celda de memoria. El primer bloque a la izquierda, CELCTL, corresponde a la celda de control que ya describimos. El quinto bloque de izquierda a derecha es el buffer mismo de almacenamiento de la celda, es decir m_n, donde se guarda un dato MIDI. Todos los otros componentes proveen una función auxiliar a la celda de control y al buffer de almacenamiento, por esto no los habíamos mencionando antes. Su código VHDL, con relación a la definición de sus señales de entrada y salida, es muy similar a la

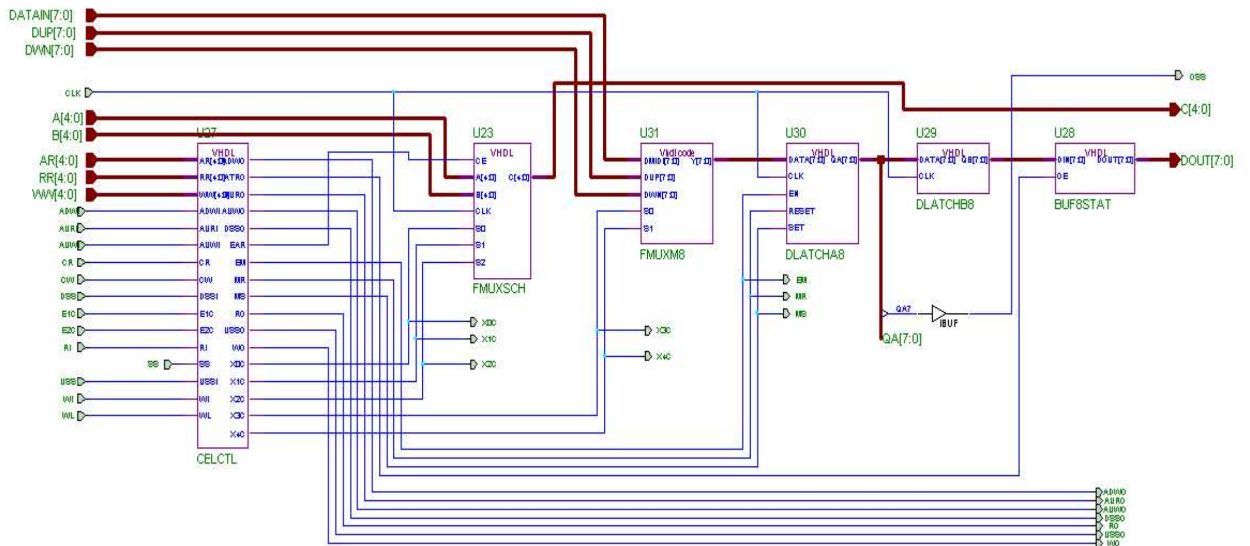


Figura 6.6: Diagrama esquemático de una celda de memoria MSA con sus componentes básicos.

definición correspondiente de la celda de control mostrada antes. Su código VHDL con relación al procesamiento de sus señales, es más simple y pequeño; se muestra y explica con más detalle en el Anexo B, página 107.

En lo que resta de esta subsección, sólo describimos su función de soporte a las funciones principales de la celda de control y del buffer de almacenamiento.

El segundo bloque de izquierda a derecha, FMUXSCH8, corresponde a un generador-multiplexor de direcciones. Su función depende del estado de escritura o lectura de la celda de control. Se recordará que inicialmente la memoria MSA está vacía y sólo tiene celdas con direcciones de escritura W_a consecutivas (ver Sección 4.2.2, página 48) las cuales no almacenan un dato sino sirven sólo como cabecera de la lista para inserción.

Al escribir el primer dato en un segmento vacío, éste se almacena en la celda vecina derecha, la cual genera una dirección de lectura R_a desplazando (los datos y) las direcciones en las celdas vecinas derechas. Al escribir el segundo dato en un segmento, la dirección R_a del primer dato y el dato mismo desalojan su celda para des-

plazarse a la siguiente celda derecha. En la celda así desalojada se escribe el segundo dato y se genera una dirección de dato D_a. Es en estos casos que el multiplexor FMUXSCH8 genera direcciones. En los otros casos de lectura y escritura selecciona la dirección (W_a, R_a o D_a) de la celda vecina derecha o izquierda, respectivamente.

El tercer bloque de izquierda a derecha, FMUXM8, es un multiplexor de datos. Su función es seleccionar qué dato se va a almacenar en su celda: el de la celda izquierda, el de la celda derecha, o el dato MIDI que se ha capturado. El cuarto bloque de izquierda a derecha, DLATCHA8, es un buffer que guarda una bandera (00 o FF) para indicar que el segmento está vacío u ocupado. El quinto bloque de izquierda a derecha, DLATCHA8, es el buffer de almacenamiento para un dato MIDI. El sexto bloque de izquierda a derecha, DLATCHB8, es otro buffer de tercer estado que habilita la salida del dato en DLATCHA8 para su lectura.

Una vez que diseñamos los componentes individuales de una celda de memoria, los conectamos por software (con un programa VHDL principal) como se mostró en la Figura 6.6 para verificar su funcionamiento. Finalmente se conectaron y verificaron por software varias celdas de memoria constituyendo una memoria MSA. Con este propósito diseñamos un componente de pruebas, llamado MAQUINA, que envía a la MSA las señales de control y datos correspondientes a la escritura y lectura de datos. De esta manera no es necesario introducir lenta y manualmente entradas a MSA a través del teclado, reduciendo el tiempo de desarrollo. El código VHDL de MÁQUINA se encuentra en el Apéndice B, página 129.

La Figura 6.7 muestra el acoplamiento esquemático de MAQUINA y la MSA, mostrando las varias conexiones entre las mismas. Por la conexión MIDI[7:0], de 8 bits, la MSA recibe un dato MIDI a guardar en la el segmento especificado por la conexión SEGMENT[4:0], de 5 bits. Recibe también una señal de reloj, CLOCK, la señal para leer datos, READ, y para escribir datos, WRITE. Las salidas de la MSA son QQQQ[7:0], de 8 bits, por donde se leen datos; DTARDY indica que hay un dato disponible para lectura y MEMRDY indica que MSA está en una operación de lec-

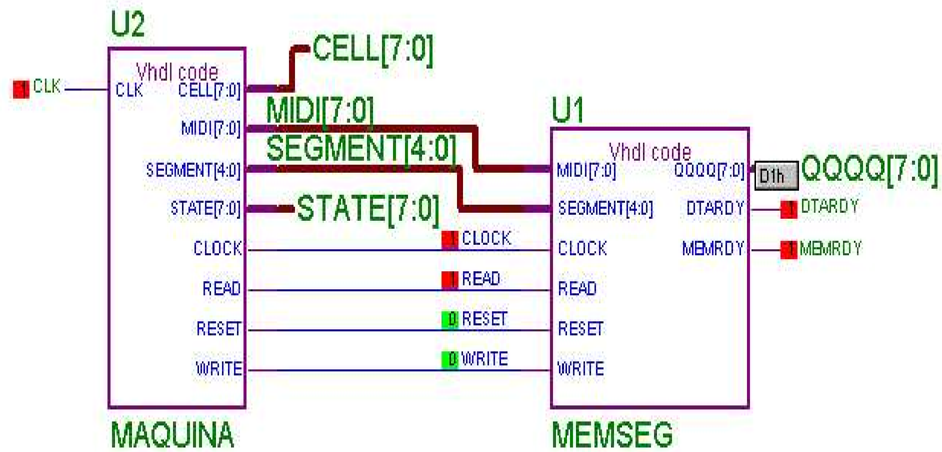


Figura 6.7: Acoplamiento esquemático de MAQUINA y MSA (MEMSEG).

tura o escritura.

6.2.4. Resultados

A continuación mostramos los resultados de la simulación de las celdas de memoria de una MSA con 16 segmentos. Mostraremos los formas de onda de la MSA al realizar varias operaciones de escritura y lectura de datos.

La Figura 6.8 muestra las formas de onda correspondientes a 10 escrituras de datos consecutivas seguidas de 3 lecturas de datos consecutivas. En esta figura, las señales de entrada y salida de la MSA en la Figura 6.7 son filas. El nombre de cada señal se encuentra en la segunda columna. La fila llamada MIDI7 muestra en la ventana principal la secuencia de escrituras, a partir de los pulsos/estados del reloj (llamados STATE7) 16_H, 17_H, 18_H, 19_H, $1A_H$, ..., $1F_H$. El valor $D1_H$ del primer dato se escribe en el segmento 1, $D2_H$ y $D3_H$ se escriben en el segmento 2, $D4_H$ y $D5_H$ en el segmento 3, $D6_H$ y $D7_H$ en el segmento 4, y $D8_H$, $D9_H$ y DA_H se escriben en el segmento 5. La fila SEGMENT4 indica sobre qué segmento se realiza la escritura o lectura de un dato.

Nótese que durante las escrituras la señal WRITE se encuentra activada, alta,

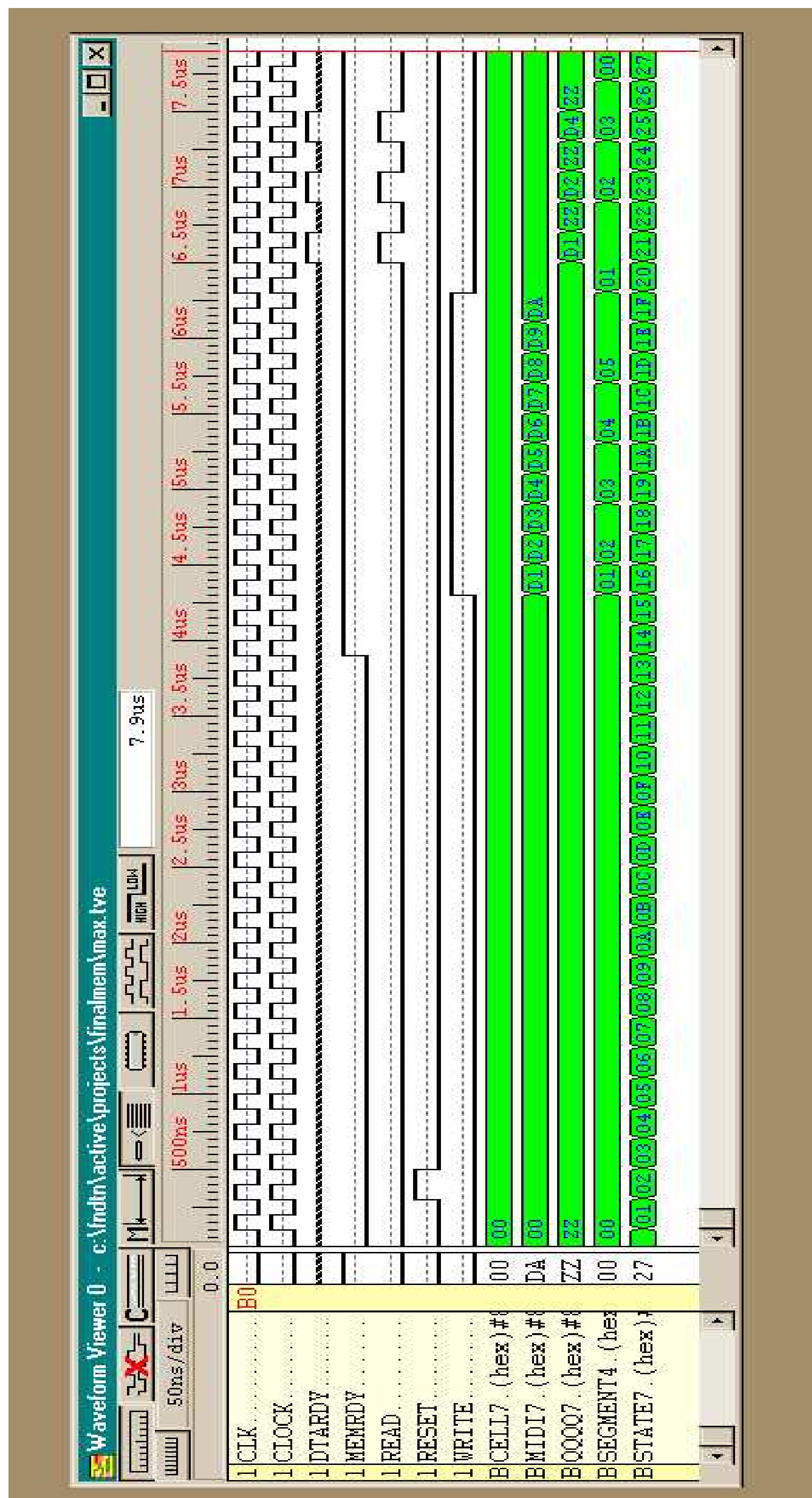


Figura 6.8: Formas de onda de la operación de MSA (MEMSEG).

en la ventana principal, desde el pulso 16_H hasta el pulso $1F_H$ de STATE7.

La fila llamada QQQQ7 muestra en la ventana principal la secuencia de lecturas, a partir de los pulsos STATE7 21_H,..., 25_H. Se lee $D1_H$ del segmento 1, luego se lee $D2_H$ del segmento 2 y finalmente se lee $D4_H$ del segmento 3. Los estados ZZ mostrados en la fila QQQQ7 entre los datos MIDI que se están leyendo, corresponden a estados de alta impedancia de la salida de la memoria. Durante la lectura de un dato la señal READ y la señal DTARDY están activadas/altas alternativamente, en los pulsos STATE7 21_H, 23_H y 25_H, debido a los estados de alta impedancia intermedios.

Finalmente, cabe señalar que las operaciones de escritura y lectura de datos empiezan hasta el pulso 16_H de la fila STATE7, ya que los primeros 21 pulsos (00_H-15_H) corresponden a la inicialización de los segmentos: de sus direcciones de escritura W_a.

6.2.5. Discusión

Nuestros resultados corresponden a una memoria de 30 celdas, de las cuales 16 corresponden a las W_a de los segmentos, los cuales, se recordará, son sólo la cabecera de los segmentos. Así que nuestra simulación sólo maneja 14 localidades para datos, y por lo tanto está limitada.

Desafortunadamente, este tipo de simulaciones son muy demandantes tanto de tiempo de procesador como de memoria. La compilación del código VHDL para 18 celdas de memoria se lleva un poco más de tres horas, mientras que la compilación de 100 celdas incurría fallas del sistema operativo Windows al punto de bloquear la máquina. Desafortunadamente no contábamos con la versión Xilinx para Unix/Linux. Por esta razón surge la necesidad de disponer de mayor potencia computacional, y software de desarrollo y simulación con mayor capacidad y confiabilidad, similar al que se utiliza hoy en día en la industria de fabricación de circuitos integrados de muy alta escala de integración VLSI.

A pesar de las limitaciones de nuestra simulación, nuestros resultados de todos los componentes básicos de una memoria MSA son una prueba de concepto que dicha

memoria es posible, y así mismo nuestro sistema MCS-S basado en la misma.

6.3. Resumen

Hemos presentado la evaluación de nuestros dos diseños de sistemas CMIDIP: MCS-L y MCS-S. La diferencia que presentan es su memoria RAM. Nuestra evaluación de MCS-L se basó principalmente en un prototipo construido con base en un sistema de desarrollo con el microcontrolador Intel 8051, y al cual añadimos otros componentes. El software que controla la captura son dos programas que hemos desarrollado de manera independiente: un núcleo de sistema operativo KLM, y un programa KL que realiza, después de la captura, la extracción de datos musicales que se requieren para procesar la impresión gráfica de las partituras.

Por otro lado, en nuestra evaluación de MCS-S sólo se consideró la simulación de la memoria segmentada autocompactante MSA, por ser éste el componente primordialmente diferente en ambos diseños. Nuestra simulación se hizo con Xilinx, un paquete que permite, a través de un medio ambiente interactivo de software, construir y simular un sistema digital completo. Primeramente presentamos, el componente principal de la MSA que es la celda de control, y luego presentamos sus componentes auxiliares. Para propósitos de prueba de la MSA, simulamos a través de un componente VHDL auxiliar, llamado MAQUINA, las condiciones de operación de escritura y lectura de la MSA y presentamos las formas de onda que nos muestran su comportamiento.

Φ

Capítulo 7

Conclusiones y Trabajos Futuros

Esta tesis presentó los aspectos de diseño de un sistema de captura, en tiempo de ejecución, de eventos MIDI provenientes de múltiples fuentes en paralelo (CMIDIP). Este tipo de sistemas son deseables para separar la música de varios instrumentos y así facilitar su post-procesamiento. Lo diferente, lo nuevo, es que nuestro diseño hace posible la extracción precisa y automática de las partituras de cada instrumento. Permite contar con las partituras de improvisaciones musicales, las cuales son una manifestación única e irrepetible. Facilita la tarea del arreglista. Y ayuda al estudio y desarrollo de la ejecución musical, al permitir obtener las partituras de ejecuciones, las cuales se pueden comparar con las partituras de versiones "perfectas".

Es posible organizar un sistema CMIDIP con computadoras personales completas, con componentes de propósito general, o con componentes diseñados a la medida. Aunque las dos primeras alternativas se pueden desarrollar en un tiempo relativamente corto y a un precio accesible, no son prácticos de usar. Para ganar una aceptación general, creemos que un CMIDIP debe ser fácil de usar por músicos que no necesitan saber de detalles técnicos complejos.

Presentamos dos diseños de sistemas CMIDIP, con componentes diseñados a la medida, que son portátiles y fáciles de usar. Su operación sólo requiere conectar los instrumentos y presionar unos pocos botones.

Uno de nuestros diseños, MCS-L (*MIDI Capture System-Lineal*), utiliza com-

ponentes comunes en el mercado y una memoria convencional estándar cuyas localidades se manejan en varias listas con un algoritmo convencional. El uso de componentes comunes permite el desarrollo de un prototipo en muy poco tiempo. Sin embargo, creemos que su fabricación masiva sería relativamente costosa debido a que requiere alrededor de 20 circuitos integrados convencionales.

El otro diseño, llamado MCS-S (MIDI *Capture System-Segmented*), utiliza un diseño especial de memoria segmentada por hardware. Su ventaja es que sólo utiliza 3 circuitos integrados de propósito específico ASIC: (*Application Specific Integrated Circuit*), por lo que en una producción masiva su costo sería considerablemente menor que el de MCS-L. Además, el uso de menos circuitos reduce la complejidad del hardware y el software de control del sistema, y por lo tanto el riesgo de errores de implementación. Las especificaciones de la memoria de MCS-S, llamada memoria segmentada autocompactante (MSA), permite la separación automática de los eventos MIDI que producen en paralelo varios instrumentos musicales. Los datos se insertan en espacios diferentes de tamaño variable, de tal manera que si un instrumento musical no genera datos, dicho espacio es aprovechado por otro que sí esté generando datos. Internamente, en cada operación de lectura o escritura, todos los datos en memoria se mueven en paralelo durante un periodo de reloj, cuya velocidad depende de la tecnología vigente impuesta en la fabricación de circuitos integrados.

Para evaluar nuestros diseños de MCS-L y MCS-S procedimos de la siguiente manera. Con base en el diseño de MCS-L desarrollamos un prototipo utilizando un sistema mínimo con el microcontrolador Intel 8051, y desarrollamos software para controlarlo. Nuestro prototipo maneja 1 MB de memoria y tiene 16 interfases MIDI para conectar el mismo número de instrumentos. Cada interfaz MIDI consiste de un UART 6850 de Motorola y un temporizador de duración de las notas MIDI que se reciben. Nuestro software de control KLM, es un pequeño sistema operativo básico escrito en lenguaje C y ensamblador que, explora los puertos de cada UART, extrae

los datos MIDI de sus registros, almacena los eventos MIDI en memoria, e inicializa y apaga los temporizadores. El manejo de memoria es una estructura (cola) FIFO global, en la que se almacena cada evento MIDI con una etiqueta que identifica al instrumento que lo generó.

Con nuestro prototipo de MCS-L realizamos una captura de eventos MIDI generados por varios instrumentos. Los datos así capturados los bajamos entonces a una PC, en la cual post-procesamos dichos datos para *generar las partituras de la música* tocada por los instrumentos. Este post-procesamiento fue realizado con KL, un compilador que desarrollamos para producir partituras con calidad profesional, escrito también en lenguaje C. Su entrada puede ser un archivo de eventos MIDI o un archivo de comandos de lenguaje natural de solfeo: por ejemplo, *do*, *re*, *mi*, *fa*, *sol la*, *si*. Ambos tipos de entrada son convertidas en comandos de símbolos gráficos musicales para diferente formatos de impresión de partituras.

Para evaluar el diseño de MCS-S diseñamos e implementamos una simulación de sus componentes que son distintos a los componentes de MCS-L. MCS-L y MCS-S difieren solamente en la memoria donde se almacenan los eventos MIDI de cada instrumento; todos los demás componentes pueden ser iguales en una implementación.

Para verificar el funcionamiento correcto de la MSA utilizamos Xilinx, un sistema (CAD *Computer Aided Design*) de software que ayuda en el diseño y desarrollo de sistemas digitales. Su medio ambiente interactivo permite el diseño, la simulación (verificación), e implementación (descarga) de circuitos integrados depositándolos en un ASIC. La especificación de un circuito se realiza en lenguaje VHDL (similar a PASCAL). Realizamos la especificación funcional en VHDL de los seis componentes que tiene cada celda de memoria: celda de control, multiplexor de direcciones, multiplexor de datos, registro de memoria auxiliar, registro de memoria principal, y un *buffer* de tercer estado para almacenamiento temporal. Se compilaron, y se observaron las formas de onda producidas en la simulación de los 16 estados que tendría

cada celda, antes de escribir o leer cualquier dato de un evento MIDI.

La MSA está formada por una sucesión consecutiva de celdas de memoria. Cuando se escribe o se lee un dato en/desde la memoria, cada celda de control genera señales internas y externas para que los datos y direcciones de todas las celdas de memoria se desplacen en paralelo una posición a la derecha o a la izquierda, respectivamente. Nuestros resultados corresponden a una memoria MSA con 30 celdas, de las cuales 16 corresponden a las cabeceras de los segmentos, por lo que nuestra simulación solo maneja 14 localidades para datos. Esta es una severa limitación debido a que la simulación de la MSA demanda demasiados recursos de hardware y software.

La compilación del código VHDL para 18 celdas de memoria tarda más de tres horas, y en la compilación de 100 celdas el sistema operativo Windows generó una falla y paró el sistema. Cabe aclarar que este problema no se debe a nuestro diseño de la MSA, sino a las limitaciones, tanto de memoria RAM operativa como de espacio en disco duro, que imponen Windows y Xilinx. Este problema se resuelve utilizando versiones profesionales de Xilinx y estaciones de trabajo con sistemas operativos tipo UNIX. No obstante las limitaciones de nuestra simulación, nuestros resultados de los componentes básicos de una memoria MSA son una prueba de concepto de que dicha memoria es posible, y así mismo nuestro sistema MCS-S basado en la misma.

En resumen, las contribuciones principales de esta tesis son las siguientes:

- Un análisis de las posibles organizaciones de hardware y software para capturar los datos de eventos MIDI que generan uno o varios instrumentos musicales.

- Dos diseños específicos completos, MCS-L y MCS-S, para lograr la captura de eventos MIDI musicales. MCS-L utiliza componentes estándar, y MCS-S utiliza una Memoria Segmentada y Autocompactante (MSA) cuyos datos internos se mueven en paralelo en cada ciclo de reloj, y cuyo periodo es dependiente de la tecnología VLSI del momento.

- Una implementación de MCS-L.

- Una diseño y verificación sólo de la memoria MSA, que es el único componente de MCS-S distinto a los componentes de MCS-L.
- Un compilador (KL) que produce partituras musicales a partir de datos MIDI o de lenguaje natural de solfeo musical.

7.1. Limitaciones y trabajos futuros

La evaluación de nuestro diseño de MCS-L y de la memoria MSA de MCS-S estuvo un tanto restringida, y es conveniente corroborar ambos diseños por medio de una evaluación más completa.

El prototipo de MCS-L sólo permite realizar la captura de eventos MIDI de 2 instrumentos. Este prototipo utiliza exploración de puertos (*polling*) para capturar los eventos MIDI. Es conveniente extender nuestro prototipo para que sea capaz de capturar los eventos MIDI de más instrumentos (16 sería ideal, pues es el número de los integrantes de una banda). En esta extensión tal vez sea necesario utilizar interrupciones en lugar del sondeo de puertos (*polling*), con el fin de mejorar el tiempo del procesador. Alternativamente, para garantizar la captura de todos los eventos MIDI de 16 instrumentos, se podría utilizar un prototipo con un procesador y memoria más rápidos. Su control, basado en KLM, debe ser relativamente fácil, ya que KLM está en lenguaje C.

Nuestro software KL permitió evaluar las partituras obtenidas a partir de los datos experimentales que se capturaron. Sin embargo, estos datos sólo describen los parámetros básicos de una nota: nota, duración, y volumen. Parámetros como ligaduras, glissandos o barras, no se pudieron capturar, se editaron después en el post-procesamiento escribiéndolos manualmente. Esto puede hacerse de manera automática aumentando la funcionalidad de KL.

La memoria MSA de MCS-S sólo fue simulada en un medio ambiente de software Xilinx cuyas limitaciones sólo permitieron simular una pequeña cantidad (30)

de celdas de memoria. Estas limitaciones se debieron a la cantidad de memoria RAM y de disco duro disponibles en nuestra computadora personal con Windows NT. Con una estación de trabajo bajo UNIX y con una versión de Xilinx profesional, sería posible corroborar un diseño de MSA con un número de celdas de memoria como la requerida para una implementación real.

Hoy en día existen paquetes de software llamados *núcleos* que proveen la funcionalidad de los circuitos integrados más comunes. Estos núcleos se escriben generalmente en lenguaje VHDL, y hay núcleos de UART (6820, 8250), de memoria RAM tradicionales, microcontroladores (8051), procesadores (68000, 80486), puertos paralelos (8255), entre muchos otros. Estos núcleos se desarrollan con herramientas como Xilinx, con la cual desarrollamos la MSA.

Encadenando nuestro diseño de MSA con otros núcleos se podría tener un diseño de MCS-S totalmente encapsulado en un solo circuito integrado ASIC cuya densidad dependerá de la tecnología VLSI que se utilice. El resultado será: mayor rapidez de captura, menor tamaño, y una mayor tolerancia a fallas.

En resumen, esta tesis presento el diseño y evaluación preliminar de sistemas de captura en tiempo de ejecución de eventos MIDI provenientes de varias fuentes en paralelo. Nuestros resultados sugieren que estos sistemas son factibles de ser construidos. Pueden también aprovecharse en la implementación de unidades terminales remotas, las cuales podrían formar parte de un sistema de mantenimiento preventivo basado en ruido, o de un sistema de supervisión y control de datos (SCADA: *Supervisory Control And Data Acquisition*).

Apéndice A

El sistema MIDI

A.1. Formato de tiempo de duración de nota *Delta Time*

Delta Time (DT) es un número entero que representa el número de *ticks* de reloj que han pasado desde el último evento ocurrido (p.e. tiempo de duración de una nota). Puede ser un valor entre 0000_0000_H y $7FFF_FFFF_H$, en un formato de longitud variable (largo) como se muestra en la Tabla A.1. DT se coloca en un registro de longitud variable codificado como sigue:

Este valor de tiempo de retardo se divide en paquetes de 7 bits, los cuales son almacenados en bytes individuales. Debido a que su longitud (en bytes) varía de acuerdo con el tiempo de retardo, se usa el bit 7 como marca final. En los primeros bytes el bit 7 se fija en "1". Solamente el último byte tiene el bit 7 en "0". De esta forma, DT se puede construir con varios bytes codificados. Se ignoran los bits más significativos, el resto de los bits de cada byte deben estar en primer plano.

Por ejemplo, un dato entero normal 2000_H se convierte a $C000_H$ en formato largo MIDI; $3FFF_H$ a $FF7F_H$; y 4000_H a 818000_H.

Tabla A.1: Tablas de formato natural y formato MIDI de *Delta Time*.

Delta Time natural (8 bits/byte)							
2^{31}	2^{30}	2^{29}	2^{28}	2^{27}	2^{26}	2^{25}	2^{24}
d_7	d_6	d_5	d_4	d_3	d_2	d_1	d_0
2^{23}	2^{22}	2^{21}	2^{20}	2^{19}	2^{18}	2^{17}	2^{16}
d_7	d_6	d_5	d_4	d_3	d_2	d_1	d_0
2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^{9}	2^{8}
d_7	d_6	d_5	d_4	d_3	d_2	d_1	d_0
2^{7}	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}
d_7	d_6	d_5	d_4	d_3	d_2	d_1	d_0

Delta Time MIDI (7 bits/byte)							
	2^{27}	2^{26}	2^{25}	2^{24}	2^{23}	2^{22}	2^{21}
1	d_6	d_5	d_4	d_3	d_2	d_1	d_0
	2^{20}	2^{19}	2^{18}	2^{17}	2^{16}	2^{15}	2^{14}
1	d_6	d_5	d_4	d_3	d_2	d_1	d_0
	2^{13}	2^{12}	2^{11}	2^{10}	2^{9}	2^{8}	2^{7}
1	d_6	d_5	d_4	d_3	d_2	d_1	d_0
	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}
0	d_6	d_5	d_4	d_3	d_2	d_1	d_0

A.2. La interfaz MIDI

A partir de 1983 la interfaz MIDI (*Musical Instrument Digital Interface*) se inició con la especificación OSI MIDI (1.0). MIDI es un protocolo computacional de hardware y software que sirve para interconectar instrumentos musicales electrónicos. La parte de software es un lenguaje programado para convertir acciones musicales en código binario digital, lenguajes de comandos, o formatos de archivos computacionales. La Figura A.2 muestra la descripción formal (Backus Naur Form) del lenguaje que definieron los mismos fabricantes de instrumentos como *Roland*, *Secuential Circuits*, entre otros.

Por otro lado, la parte de hardware consiste de conectores DIN de 5 terminales eléctricamente conectados a un circuito integrado UART de comunicaciones (6850), el cual es programado a una velocidad de 32 250 bauds. El receptor de datos (MIDI IN) se encuentra aislado eléctricamente a través de un optoacoplador (PC-900). De la misma forma, el transmisor de datos (MIDI OUT) también se encuentra aislado a través de dos inversores (*buffers*). Se hace una copia de datos del receptor para que sean retransmitidos hacia otros instrumentos (MIDI TRUE). Esto permite configurar una red tipo *daisy chain* entre los instrumentos musicales. La Figura A.1 muestra el diagrama esquemático que definieron dichos fabricantes.

Tabla A.2: Otros estándares y protocolos de comunicación.

Interfaz	Formato	Max. disps.	Long. max. (pies)	bits/seg max.	uso típico
USB	Serial asíncrono	127	16 (o hasta 96 pies con 5 hubs)	1.5 M, 12 M 480 M	Ratón, teclado, modem, audio
RS-232 (EIA/TIA 232)	Serial asíncrono	2	50-100	20k (115k con algún hardware)	Modem, ratón, instrum.
RS-285 (TIA/EIA 485)	Serial asíncrono	32 unidades de carga (hasta 256 dispositivos con algún hardware	4000	10M	Adquisición de datos y, control de sistemas
IrDA	infrarojo serial asíncrono	2	6	115k	Impresoras, $hand-held$ $computers$
Microwire	serial asíncrono	8	10	2M	comunicación con micro-controladores
SPI	serial asíncrono	8	10	2.1M	comunicación con micro-controladores
I^2C	serial asíncrono	40	18	3.4M	comunicación con micro-controladores
IEEE-1394 (FireWire)	serial	64	15	400 M incrementa a 3.2G con IEEE-1394b	Video, almacena-miento masivo
IEEE-488 (GPIB)	paralelo	15	60	8M	Instrmenta-ción
Ethernet	serial	1024	1600	10M/100M/1G	Redes PC
MIDI	serial, lazo de corriente	2 (más con el modo $flow-$ $through$)	50	31.5k	Música, control de escenarios
Puerto paralelo impresión	paralelo	2 (8 con soporte $daisy-$ $chain$)	10-30	8M	Impresoras, $scanners$, controladores de disco

Fuente: Axelson Jan,*USB Complete: Everything You Need to Develop Custom USB Peripherals,* Second Edition, Lakview Research, p: 4. USA 2001. 523 pgs.

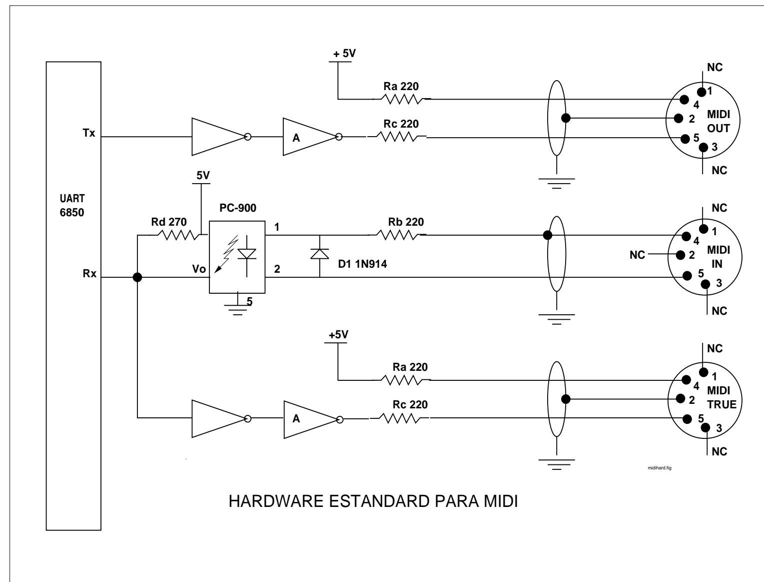


Figura A.1: Especificación hardware MIDI.

Tabla A.3: Medios físicos de transmisión de datos.

MEDIO	VELOCIDAD
Cable de cobre	30-300 Khz
Par trenzado	Hasta 4 Mbps
Coaxial	Hasta 500 Mbps
Fibra óptica	Hasta 2000 Mbps
Fuente: *Edificios Inteligentes: Arquitectura Ingeniería Construcción,* Fundación casa de la arquitectura, México. p:126	

Tabla A.4: Ancho de banda: Voz, TV y microondas.

MEDIO	VELOCIDAD
Voz	300-3300 Hz
TV cable	54 Mhz a 750 Mhz
Torres de microondas	2 Ghz a 12 Ghz
Fuente: Alesso Peter H., *e-Video: Producing Video as Broadband Technologies Converge,* Addison-Wesley, USA 2000, p:14. 289 pgs.	

1

Descripción Backus Naur Form de MIDI

```
 1. <MIDIstream>::=<MIDImsg>< MIDIstream>
 2. <MIDImsg>::=<SisMsg>|<ChanMsg>
 3. <ChanMsg>::=<Chan1ByteMsg>|<Chan2ByteMsg>
 4. <Chan1ByteMsg>::=<ChanStat1Byte><DataSinglet>
                     <RunningSinglets>
 5. <Chan2ByteMsg>::=<ChanStat2Byte><DataPair><RunningPairs>
 6. <ChanStat1Byte>::=<ChanVoiceStat1Nibble><HexNyble>
 7. <ChanStat2Byte>::=<ChanVoiceStat2Nibble><HexNyble>
 8. <ChanVoiceStat1Nyble>::=C|D
 9. <ChanVoiceStat2Nyble>::=8|9|A|B|E
10. <HexNyble>::=0|1|2|3|4|5|6|7|8|9|A|B|C|D|E|F
11. <DataPair>::=<DataSinglet><DataSinglet>
12. <DataSinglet>::=<RealTimeByte><DataSinglet>|<DataByte>
13. <RunningPairs>::=<Empty>|<DataPair><RunningPairs>
14. <RunningSinglets>::=<Empty>|<DataSinglet><RunningSinglets>
15. <DataByte>::=<DataMSD><HexNyble>
16. <DataMSD>::=0|1|2|3|4|5|6|7
17. <RealTimeByte>::=F8|FA|FB|FC|FE|FF
18. <SisMsg>::=<SysCommonMsg>|<SysExMsg>|<SysRealTimeMsg>
19. <SysRealTimeMsg>::=<RealTimeByte>
20. <SysExMsg>::=<SysExDataByte><DataSinglet><RunningSinglets>
                <EoxByte>
21. <SysExStatByte>::=F0
22. <EoxByte>::=F7
23. <SysCommonMsg>::=<SongPositionMsg>|<SongSelectMsg>
                                       |<TuneRequest>
24. <TuneRequest>::=F6
25. <SongPositionMsg>::=<SongPositionStatByte><DataPair>
26. <SongSelectMsg>::=<SongSelectStatByte><DataSinglet>
27. <SongPositionStatByte>::=F2
28. <SongSelectStatByte>::=F3
```

Figura A.2: Especificación software MIDI.

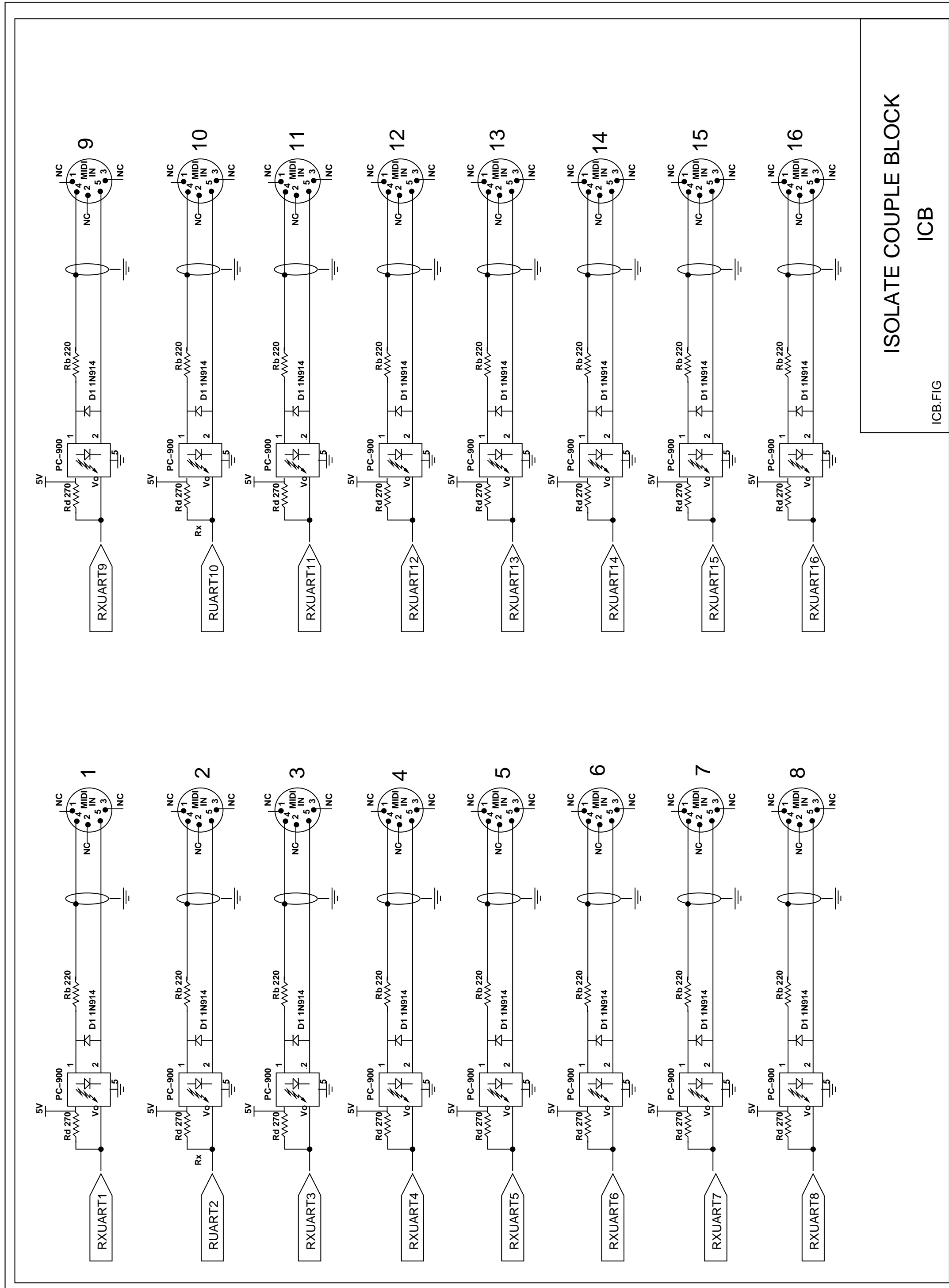


Figura A.3: Interfaz hardware del MCS con los instrumentos MIDI.

Apéndice B

Códigos fuente de componentes VHDL

B.1. Código de MEMSEG

```
-----------------------------------------------------
-- memseg() - memoria segmentada.
----------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
entity MEMSEG is
generic (constant NCELDAS: integer:=5); -- numero de celdas.
port(
     MIDI: in  std_logic_vector(7 downto 0);
  SEGMENT: in  integer range 1 to 16;
    RESET: in  std_logic;
    WRITE: in  std_logic;
     READ: in  std_logic;
    CLOCK: in  std_logic;

     MEMRDY: out  std_logic;
     DTARDY: out  std_logic;
       QQQQ: out std_logic_vector(7 downto 0)
    );
end MEMSEG;

library ieee;
use ieee.std_logic_1164.all;
--------------------------------------------
-- celctl() -control de la celda de memoria.
--------------------------------------------
entity CELCTL is
port(
 CW,CR,E1C,E2C,WI,RI,WL,SS: in  std_logic;
  AUWI,AURI,ADWI,USSI,DSSI: in  std_logic;
  WW,RR,AR: in  std_logic_vector(7 downto 0);
EAR,EM,WO,RO,MS,MR: out std_logic;
       X0C,X1C,X2C,X3C,X4C: out std_logic;
       ATRO,AUWO,AURO,ADWO: out std_logic;
```

```
 USSO,DSSO: out std_logic
);
end CELCTL;
architecture operacion of CELCTL is
signal vsal: std_logic_vector(16 downto 0);

begin  process (CW,CR,WW,AR,RR,SS,ADWI,AURI,DSSI,WI,RI,vsal) begin
--65432109876543210
-- estado 1.
  if  (CW='1' and AR=WW    and SS='0') then
       vsal<="01001000000000100";
-- estado 2.
elsif (CW='1' and AR=WW    and SS='1') then
       vsal<="00000000000000101";
-- estado 3.
elsif (CW='1' and AR>WW    and ADWI='1' and DSSI='0') then
       vsal<="11000000110000000";
-- estado 4.
elsif (CW='1' and AR>WW    and ADWI='1' and DSSI='1') then
      vsal<="11000001010000000";
-- estado 5.
elsif (CW='1' and AR/=X"00" and AR>WW   and ADWI='0') then
       vsal<="11000000000000000";
-- estado 6, no se da WI siempre 0.
elsif (CW='1' and AR=X"00" and WI='1') then
       vsal<="11000000000000000";
-- estado 7 no se da RI siempre 0.
elsif (CW='1' and AR=X"00" and RI='1') then
       vsal<="11000000000000000";
-- estado 8.
elsif (CW='1' and AR=X"00" and ADWI='1' and DSSI='0') then
       vsal<="11000000110000000";
-- estado 9.
elsif (CW='1' and AR=X"00" and WI='0'   and RI='0') then
       vsal<="00000000000000000";
-- estado 10.
elsif (CW='1' and AR/=X"00" and AR<WW) then
       vsal<="00000000000000000";
-- estado 11.
elsif (CR='1' and AR=RR) then
       vsal<="11000010011101000";
-- estadao 12.
elsif (CR='1' and AR/=X"00" and AR<RR and AURI='0') then
       vsal<="00000000000000000";
-- estado 13 falta WL=1 ?
--elsif (CR='1' and AR/=X"00" and AR<RR and AURI='1') then
 --vsal<="01000100000000000";
-- estado 14 falta WL=0?
elsif (CR='1' and AR/=X"00" and AR<RR and AURI='1') then
       vsal<="10000010000000000";
-- estado 15.
elsif (CR='1' and AR>RR) then
       vsal<="11000010011000000";
-- estado 16.
elsif (CR='1' and AR=X"00" and AURI='0') then
```

```
       vsal<="00000000000000000";
else
       vsal<="00000000000000000";
end if;

--Calcula WO
if (AR(1)='1' and AR(0)='0') then
    WO<='1'; else WO<='0';
    end if; --WO

--Calcular RO
if (AR(2)='1' and AR(1)='0' and AR(0)='0') then
    RO<='1'; else RO<='0';
    end if; --RO

   EAR <=vsal(16);
    EM <=vsal(15);
--  WO <=vsal(14);
--  RO <=vsal(13);
    MS <=vsal(12);
    MR <=vsal(11);
   X0C <=vsal(10);
   X1C <=vsal(9);
   X2C <=vsal(8);
   X3C <=vsal(7);
   X4C <=vsal(6);
  ATRO <=vsal(5);
  AUWO <=vsal(4);
  AURO <=vsal(3);
  ADWO <=vsal(2);
  USSO <=vsal(1);
  DSSO <=vsal(0);

end process;
end operacion;
----------------------------------------------------------------
-- fmuxsch() - crear direcciones, desplazar datos.
---------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
entity fmuxsch is port
(
     CLK,CE,s0,s1,s2: in  std_logic;
         A,B: in  std_logic_vector(7 downto 0);
   C: out std_logic_vector(7 downto 0)
);
end fmuxsch;
architecture rtlfmuxsch of fmuxsch is
signal sel: std_logic_vector(2 downto 0);
signal TC: std_logic_vector(7 downto 0);

begin process (CLK,CE,S0,S1,S2,sel,A,B) begin
sel(0)<=s0; --X0C
sel(1)<=s1; --X1C
sel(2)<=s2; --X2C
```

```
case sel is
     when "000" => TC<=A; -- shift right, recibe direccion celda izquierda.
     when "001" => TC<=B; -- shift left, recibe direccion celda deracha.
     when "010" => TC(2 downto 0) <="011";
   TC(7 downto 3) <= A(7 downto 3); --direccion dato.
     when "100" => TC(2 downto 0) <="100";
   TC(7 downto 3) <= A(7 downto 3); --direccion lectura
     when others => TC<=A;
end case;
end process;

process(CLK) begin
 if (CLK'event and CLK='1') then
     if (CE='1') then C <= TC; end if;
 end if;
end process;
end rtlfmuxsch;

----------------------------------------------------------
-- fmuxm8() -multiplexor de entrada a memoria (8 bits).
----------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
entity fmuxm8 is port
(
      S0,S1: in  std_logic;
      DUP,DMIDI,DWN: in  std_logic_vector(7 downto 0);
          Y: out std_logic_vector(7 downto 0)
);
end fmuxm8;

architecture RTL of fmuxm8 is
signal selector: std_logic_vector(1 downto 0);
begin process (S0,S1,DUP,DMIDI,DWN,selector) begin
selector(0)<=S0; --X3C
selector(1)<=S1; --X4C
case (selector) is
     when "00" => Y <= DUP;
     when "01" => Y <= DMIDI;
     when "11" => Y <= DWN;
     when others => Y <=X"00";
end case;
end process;
end RTL;
-----------------------------------------------
-- dlatchb8() -flipflop tipo D sincrono 8 bits.
-----------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
entity DLATCHB is  port
(
      DB: in  std_logic_vector(7 downto 0);
     CLK: in  std_logic;
      QB: out std_logic_vector(7 downto 0)
);
```

```
end DLATCHB;
architecture RTL of DLATCHB is
begin process(CLK)begin
 if (CLK'event and CLK='0') then
     QB <= DB;
 end if;
end process;
end RTL;
------------------------------------------------------------------
-- dlatcha8() -flipflop tipo D con set reset enable sincrono 8 bits.
------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;

entity DLATCHA is  port
(
   DA: in  std_logic_vector(7 downto 0);
     CLK,SET,RESET,EN: in  std_logic;
   QA: out std_logic_vector(7 downto 0)
);
end DLATCHA;
architecture RTL of DLATCHA is
begin process(CLK)begin
 if (CLK'event and CLK='1') then
    if (SET='1') then
QA <= X"FF";
    elsif (RESET='1') then
QA <= X"00";
    elsif (EN='1') then
QA <= DA;
    end if;
 end if;
end process;
end RTL;
----------------------------------------------------------
-- buf8stat() - buffer de salida tercer estado 8bits.
----------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;

entity BUFFERO is port
(
     BIN: in  std_logic_vector(7 downto 0);
      OE: in  std_logic;
    BOUT: out std_logic_vector(7 downto 0)
);
end BUFFERO;
architecture RTL of BUFFERO is
  signal temporal: std_logic_vector (7 downto 0);
  begin process(BIN,OE,temporal) begin
        temporal<=BIN;
if (OE='1') then
    BOUT<=temporal;
else
    BOUT<="ZZZZZZZZ";
```

```
end if;
  end process;
end RTL;
--------------------------
-- Arquitectura principal.
--------------------------
architecture RTL of MEMSEG is

type   CABLE   is array (0 to NCELDAS -1) of std_logic_vector(7 downto 0);
type   ALAMBRE is array (0 to NCELDAS -1) of std_logic;
signal tq,td,ta,tb,tcc,tqa,tqb,tdup,tdmidi,tdwn,ty,tww,trr,tar: CABLE;

signal datareset,tareset,tbusww,tbusrr: std_logic_vector(7 downto 0);
signal bitss: std_logic_vector(7 downto 0);
signal databus: CABLE;

signal tcr,te1c,te2c,twi,tri,twl,
       tauwi,tauri,tadwi,tussi,tdssi,
       tear,tem,two,tro,tms,tmr,
       tx0c,tx1c,tx2c,tx3c,tx4c,
       tatro,tauwo,tauro,tadwo,
       tusso,tdsso,
       tschce,tmce,HabilitarSCH,tdtardy: ALAMBRE;
signal ContadorReset: integer range 0 to 16;
signal tbcr,tbcw:std_logic;
signal ResetGeneral: BOOLEAN;

signal tcw:  ALAMBRE;
signal tce:  ALAMBRE;
signal tss:  ALAMBRE;

signal ttcw: ALAMBRE;
signal ttcr: ALAMBRE;
signal ttce: ALAMBRE;
signal ttss: ALAMBRE;
---------------------------------
-- interfase celda de control.
---------------------------------
component CELCTL port(
 CW,CR,E1C,E2C,WI,RI,WL,SS: in  std_logic;
  AUWI,AURI,ADWI,USSI,DSSI: in  std_logic;
  WW,RR,AR: in  std_logic_vector(7 downto 0);
EAR,EM,WO,RO,MS,MR: out std_logic;
       X0C,X1C,X2C,X3C,X4C: out std_logic;
       ATRO,AUWO,AURO,ADWO: out std_logic;
 USSO,DSSO: out std_logic);
end component;
-----------------------------------------------------
-- interfase multiplexor crear/desplazar direcciones.
-----------------------------------------------------
component FMUXSCH port(
     CLK,CE,s0,s1,s2: in  std_logic;
         A,B: in  std_logic_vector(7 downto 0);
   C: out std_logic_vector(7 downto 0));
end component;
```

```
------------------------------------
-- interfase multiplexor de memoria.
------------------------------------
component FMUXM8 port(
   S0,S1: in  std_logic;
          DUP,DMIDI,DWN: in  std_logic_vector(7 downto 0);
       Y: out std_logic_vector(7 downto 0));
end component;
---------------------------
-- interfase flip flop A.
---------------------------
component DLATCHA port(
      DA: in  std_logic_vector(7 downto 0);
          CLK,SET,RESET,EN: in  std_logic;
      QA: out std_logic_vector(7 downto 0));
end component;
---------------------------
-- interfase flip flop B.
---------------------------
component DLATCHB port(
      DB: in  std_logic_vector(7 downto 0);
     CLK: in  std_logic;
      QB: out std_logic_vector(7 downto 0));
end component;
---------------------------
-- interfase buffer.
---------------------------
component  BUFFERO port(
     BIN: in  std_logic_vector(7 downto 0);
      OE: in  std_logic;
    BOUT: out std_logic_vector(7 downto 0));
end component;

begin
ResetearSistema: process (CLOCK) begin
     if (CLOCK'event and CLOCK='1') then
         if (RESET='1')        then
             ResetGeneral <= TRUE;
             MEMRDY <='0';
         end if;
         if (ContadorReset=16) then
             ResetGeneral <= FALSE;
             MEMRDY <='1';
         end if;
     end if;
end process ResetearSistema;

IniciarApuntadores: process(CLOCK) begin
     if (CLOCK'event and CLOCK='1') then
 if (ResetGeneral=TRUE) then
     case ContadorReset is
  when 0 => tareset<=X"7A";
  when 1 => tareset<=X"72";
  when 2 => tareset<=X"6A";
  when 3 => tareset<=X"62";
```

```
  when 4 => tareset<=X"5A";
  when 5 => tareset<=X"52";
  when 6 => tareset<=X"4A";
  when 7 => tareset<=X"42";
  when 8 =>  tareset<=X"3A";
  when 9 =>  tareset<=X"32";
  when 10 => tareset<=X"2A";
  when 11 => tareset<=X"22";
  when 12 => tareset<=X"1A";
  when 13 => tareset<=X"12";
  when 14 => tareset<=X"0A";
  when 15 => tareset<=X"02";
  when others => tareset<=X"00";
  end case;
     ContadorReset <= ContadorReset+1;
   else ContadorReset<=0;
 end if;
         end if;
end process IniciarApuntadores;

SeleccionarApuntador: process(SEGMENT) begin
     case (SEGMENT) is
   when  1 => tbusww <= X"02"; tbusrr <=X"04";
   when  2 => tbusww <= X"0A"; tbusrr <=X"0C";
   when  3 => tbusww <= X"12"; tbusrr <=X"14";
   when  4 => tbusww <= X"1A"; tbusrr <=X"1C";
   when  5 => tbusww <= X"22"; tbusrr <=X"24";
   when  6 => tbusww <= X"2A"; tbusrr <=X"2C";
   when  7 => tbusww <= X"32"; tbusrr <=X"34";
   when  8 => tbusww <= X"3A"; tbusrr <=X"3C";

   when  9 => tbusww <= X"42"; tbusrr <=X"44";
   when  10 => tbusww <= X"4A"; tbusrr <=X"4C";
   when  11 => tbusww <= X"52"; tbusrr <=X"54";
   when  12 => tbusww <= X"5A"; tbusrr <=X"5C";
   when  13 => tbusww <= X"62"; tbusrr <=X"64";
   when  14 => tbusww <= X"6A"; tbusrr <=X"6C";
   when  15 => tbusww <= X"72"; tbusrr <=X"74";
   when  16 => tbusww <= X"7A"; tbusrr <=X"7C";
           when others => tbusww <= X"00";tbusrr <=X"00";
          end case;
end process SeleccionarApuntador;

ConectarCeldas:for J in 0 to NCELDAS-1 generate
--
-- Conectar la primera celda.
--
PrimeraCelda: if J=0 generate
HabilitarCE: process(CLOCK) begin
    tce(J)  <= '1'; -- Chip 1 Siempre habilitado
    ttce(J) <= '1';
end process;

HabilitarCW: process(WRITE)begin
     tcw(J)  <=WRITE; -- Celda 1 siempre activa.
```

```
     ttcw(J) <=WRITE;
end process;

HabilitarSS: process(tqa)begin
     if (tqa(J)=X"FF") then -- dato de celda
         tss(J)  <= '1';    -- Bit de status bit 7
         ttss(J) <= '1';
     else
         tss(J)  <= '0';    -- Bit de status bit 7
         tss(J) <= '0';
     end if;
end process;

HabilitarDirecciones: process(ResetGeneral,tareset,tcc)begin
     if (ResetGeneral=TRUE) then
         ta(J)     <= tareset;  --direccion anterior en reset.
         tar(J)    <= tcc(J);   --direccion actual.
         tb(J)     <= tcc(J+1); --direccion posterior.
     else
         ta(J)     <= X"02";    --La primera seiempre es WW1.
         tar(J)    <= tcc(J);   --direccion actual
         tb(J)     <= tcc(J+1); --direccion posterior
     end if;
end process;

    tadwi(J) <= '0';
    tauri(J) <= '0';
    tauwi(J) <= '0';
    tcr(J)   <= '0';
    tdssi(J) <= '0';
    te1c(J)  <= '0';
    te2c(J)  <= '0';
    tri(J)   <= '0';
    tussi(J) <= '0';
    twi(J)   <= '0';
    twl(J)   <= '0';
end generate PrimeraCelda;
--
-- Conectar las celdas intermedias
--
CeldasIntermedias: if (J>0 and J< NCELDAS-1) generate
     -------
     --FMUXM
     -------
     tdup(J)   <=tqa(J-1);
     tdmidi(J) <=MIDI;
     tdwn(J)   <=tqa(J+1);
     ---------
     --FMUXSCH
     ---------
HabilitarCE: process(ResetGeneral,tear) begin
     if (ResetGeneral=TRUE) then
         tce(J)  <= '1';
         ttce(J) <= '1';
     else
```

```
          tce(J)  <= tear(J);
          ttce(J) <= tear(J);
     end if;
end process;

HabilitarCW: process(ResetGeneral,WRITE) begin
     if (ResetGeneral=TRUE) then
 tcw(J)  <='1';
 ttcw(J) <='1';
     else
 tcw(J)  <=WRITE;
 ttcw(J) <=WRITE;
     end if;
end process;

HabilitarCR: process(READ)begin
     tcr(J)  <=READ; -- Celda 1 siempre activa.
     ttcr(J) <=READ;
end process;

HabilitarSS: process(tqa)begin
     if (tqa(J)=X"FF") then -- dato de celda
 tss(J)  <= '1';    -- Bit de status bit 7
 ttss(J) <= '1';
     else
 tss(J)  <= '0';    -- Bit de status bit 7
 ttss(J) <= '0';
     end if;
end process;

ConectarQQQQ: process(tq) begin
     QQQQ<=tq(j);
end process;

ConectarATRO: process(tatro,tdtardy) begin
     if (tatro(J)='1') then
 tdtardy(J)<='1';
     else
 tdtardy(J)<='Z';
     end if;
 DTARDY<=tdtardy(J);
end process;

HabilitarDirecciones: process(tcc) begin
     ta(J)   <= tcc(J-1);  -- direccion anterior
     tar(J)  <= tcc(J);    -- direccion actual
     tb(J)   <= tcc(J+1);  -- direccion posterior
end process;

     tadwi(J)   <= tadwo(J-1);
     tauri(J)   <= tauro(J+1);
     tauwi(J)   <= tauwo(j+1);
     tdssi(J)   <= tdsso(J-1);
     twi(J)     <= two(J-1);
     tri(J)     <= tro(J-1);
```

```
     tussi(J)   <= tusso(J+1);
     te1c(J)    <= '0';
     te2c(J)    <= '0';
     twl(J)     <= '0';

end generate CeldasIntermedias;
--
-- Conectar la última celda
--
    UltimaCelda: if J=NCELDAS-1 generate

                    tdup(J)   <= x"00"; -- dato anterior.
                    tdmidi(J) <= x"00"; -- dato midi.
                    tdwn(J)   <= x"00"; -- dato posterior.

                    ta(J)     <= x"00"; -- direccion anterior en reset.
                    tar(J)    <= x"00"; -- direccion actual.
                    tb(J)     <= x"00"; -- direccion posterior.

                    tadwi(J) <= '0';
                    tauri(J) <= '0';
                    tauwi(J) <= '0';
                    tcr(J)   <= '0';
                    tdssi(J) <= '0';
                    te1c(J)  <= '0';
                    te2c(J)  <= '0';
                    tri(J)   <= '0';
                    tussi(J) <= '0';
                    twi(J)   <= '0';
                    twl(J)   <= '0';

     end generate UltimaCelda;

end generate ConectarCeldas;

GenerarCeldas: for I in 0 to NCELDAS-1 generate
 U1: CELCTL port map(
     CW   => tcw(I),
     CR   => tcr(I),
     E1C  => te1c(I),
     E2C  => te2c(I),
     WI   => twi(I),
     RI   => tri(I),
     WL   => twl(I),
     SS   => tss(I),
     AUWI => tauwi(I),
     AURI => tauri(I),
     ADWI => tadwi(I),
     USSI => tussi(I),
     DSSI => tdssi(I),
     WW   => tbusww,
     RR   => tbusrr,
     AR   => tar(I),

     EAR  => tear(I),
```

```
    EM   => tem(I),
    WO   => two(I),
    RO   => tro(I),
    MS   => tms(I),
    MR   => tmr(I),
    X0C  => tx0c(I),
    X1C  => tx1c(I),
    X2C  => tx2c(I),
    X3C  => tx3c(I),
    X4C  => tx4c(I),
    ATRO => tatro(I),
    AUWO => tauwo(I),
    AURO => tauro(I),
    ADWO => tadwo(I),
    USSO => tusso(I),
    DSSO => tdsso(I)
   );
U2: FMUXSCH port map (
    CLK => CLOCK,
    CE  => tce(I),
    S0  => tx0c(I),
    S1  => tx1c(I),
    S2  => tx2c(I),
    A   => ta(I),
    B   => tb(I),

    C   => tcc(I)
   );
U3: FMUXM8  port map(
    S0    => tx3c(I),
    S1    => tx4c(I),
    DUP   => tdup(I),
    DMIDI => tdmidi(I),
    DWN   => tdwn(I),

    Y     => ty(I)
    );
U4: DLATCHA port map(
    DA    => ty(I),
    CLK   => CLOCK,
    SET   => tms(I),
    RESET => tmr(I),
    EN    => tem(I),
    QA    => tqa(I)
    );
U5: DLATCHB port map(
    DB    => tqa(I),
    CLK   => CLOCK,
    QB    => tqb(I)
    );
U6: BUFFERO port map (
    BIN   => tqb(I),
    OE    => tatro(I),
    BOUT  => tq(I)
    );
```

```
 end generate GenerarCeldas;
end RTL;
```

Tabla B.1: Señales del componente final memoria segmentada MEMSEG.

TERMINAL	E/S	FUNCIÓN
MIDI	Entrada	Dato de 8 bits.
SEGMENT	Entrada	No. de segmento (1-16).
CLOCK	Entrada	Reloj (copia de CLK).
READ	Entrada	"1", lectura.
RESET	Entrada	"1", iniciar la memoria.
WRITE	Entrada	"1", escritura.
DTARDY	Salida	"1", dato listo para leer.
MEMRDY	Salida	"1", memoria lista para recibir datos.
QQQQ	Salida	Dato de 8 bits disponible.

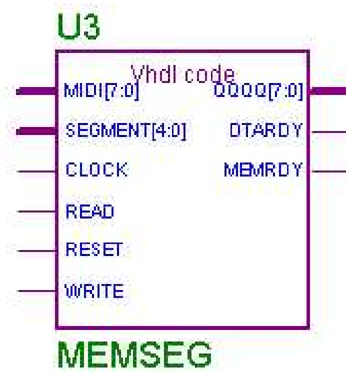


Figura B.1: Componente final de memoria segmentada MEMSEG.

Tabla B.2: Señales de MEMSEG con ventana de direcciones y datos de prueba.

TERMINAL	E/S	FUNCIÓN
CELL	Entrada	Dirección base de ventana (1-255).
MIDI	Entrada	Dato de 8 bits
SEGMENT	Entrada	No. de segmento (1-16).
CLOCK	Entrada	Reloj (copia de CLK).
READ	Entrada	”1”, lectura.
RESET	Entrada	”1”, iniciar la memoria.
WRITE	Entrada	”1”, escritura.
DTARDY	Salida	”1”, dato listo para leer.
MEMRDY	Salida	”1”, memoria lista para recibir datos.
QQQQ	Salida	Dato de 8 bits disponible.
A00A-A15A	Salida	Ventana de 16 direcciones.
D00D-D15D	Salida	Ventana de 16 datos.

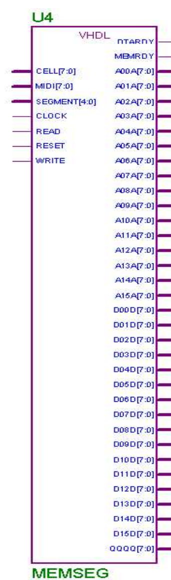


Figura B.2: Componente memoria segmentada de prueba MEMSEG.

B.2. Código de CELCTL.

```
library ieee;
use ieee.std_logic_1164.all;

entity CELCTL is
port(
 CW,CR,E1C,E2C,WI,RI,WL,SS: in  std_logic;
  AUWI,AURI,ADWI,USSI,DSSI: in  std_logic;
  WW,RR,AR: in  std_logic_vector(7 downto 0);
```

```
EAR,EM,WO,RO,MS,MR: out std_logic;
       X0C,X1C,X2C,X3C,X4C: out std_logic;
       ATRO,AUWO,AURO,ADWO: out std_logic;
 USSO,DSSO: out std_logic
);
end CELCTL;

architecture operacion of CELCTL is
signal vsal: std_logic_vector(16 downto 0);
begin  process (CW,CR,WW,AR,RR,SS,ADWI,AURI,DSSI,WI,RI,vsal) begin
       --65432109876543210
       -- estado 1.
  if  (CW='1' and AR=WW    and SS='0') then
       vsal<="01001000000000100";
       -- estado 2.
elsif (CW='1' and AR=WW    and SS='1') then
       vsal<="00000000000000101";
       -- estado 3.
elsif (CW='1' and AR>WW    and ADWI='1' and DSSI='0') then
       vsal<="11000000110000000";
       -- estado 4.
elsif (CW='1' and AR>WW    and ADWI='1' and DSSI='1') then
      vsal<="11000001010000000";
       -- estado 5.
elsif (CW='1' and AR/=X"00" and AR>WW   and ADWI='0') then
       vsal<="11000000000000000";
       -- estado 6, no se da WI siempre 0.
elsif (CW='1' and AR=X"00" and WI='1') then
       vsal<="11000000000000000";
       -- estado 7 no se da RI siempre 0.
elsif (CW='1' and AR=X"00" and RI='1') then
       vsal<="11000000000000000";
       -- estado 8.
elsif (CW='1' and AR=X"00" and ADWI='1' and DSSI='0') then
       vsal<="11000000110000000";
       -- estado 9.
elsif (CW='1' and AR=X"00" and WI='0'   and RI='0') then
       vsal<="00000000000000000";
       -- estado 10.
elsif (CW='1' and AR/=X"00" and AR<WW) then
       vsal<="00000000000000000";
       -- estado 11.
elsif (CR='1' and AR=RR) then
       vsal<="11000010011101000";
       -- estadao 12.
elsif (CR='1' and AR/=X"00" and AR<RR and AURI='0') then
       vsal<="00000000000000000";
       -- estado 13 falta WL=1 ?
--elsif (CR='1' and AR/=X"00" and AR<RR and AURI='1') then
       -- vsal<="01000100000000000";
       -- estado 14 falta WL=0?
elsif (CR='1' and AR/=X"00" and AR<RR and AURI='1') then
       vsal<="10000010000000000";
       -- estado 15.
elsif (CR='1' and AR>RR) then
```

```
       vsal<="11000010011000000";
       -- estado 16.
elsif (CR='1' and AR=X"00" and AURI='0') then
       vsal<="00000000000000000";
else
       vsal<="00000000000000000";
end if;
    --Calcula WO
if (AR(1)='1' and AR(0)='0') then
    WO<='1'; else WO<='0';
end if;
    --Calcular RO
if (AR(2)='1' and AR(1)='0' and AR(0)='0') then
    RO<='1'; else RO<='0';
end if;
-- Salida de variables.
   EAR <=vsal(16);
   EM <=vsal(15);
-- WO <=vsal(14);
-- RO <=vsal(13);
   MS <=vsal(12);
   MR <=vsal(11);
   X0C <=vsal(10);
   X1C <=vsal(9);
   X2C <=vsal(8);
   X3C <=vsal(7);
   X4C <=vsal(6);
  ATRO <=vsal(5);
  AUWO <=vsal(4);
  AURO <=vsal(3);
  ADWO <=vsal(2);
  USSO <=vsal(1);
  DSSO <=vsal(0);
end process;
end operacion;
```

Tabla B.3: Señales de una celda de control CELCTLM.

TERMINAL	E/S	FUNCIÓN
AR	Entrada	Dirección de celda (8 bits).
RR	Entrada	Apuntador de lectura (8 bits).
WW	Entrada	Apuntador de escritura (8 bits).
ADWI	Entrada	Celda anterior (c_{n+1}) es W activa.
AURI	Entrada	Celda siguiente (c_{n+1}) es R activa.
AUWI	Entrada	Celda siguiente (c_{n+1}) es W activa.
CR	Entrada	"1", habilitar operación de lectura.
CW	Entrada	"1", habilitar operación de escritura.
DSSI	Entrada	*Status* de segmento celda anterior (c_{n-1}): "0", vacío; "1", ocupado.
E1C	Entrada	Reservada.
E2C,	Entrada	Reservada.
RI	Entrada	Celda anterior (c_{n-1}) es R.
SS	Entrada	*Status* de segmento de esta celda (c_n): "0", vacío; "1", ocupado.
USSI	Entrada	*Status* de segmento celda siguiente (c_{n+1}): "0", vacío; "1", ocupado.
WI	Entrada	Celda anterior (c_{n-1}) es W.
WL	Entrada	Esta celda (c_n) es dirección de escritura W.
ADWO	Salida	Indicar a celda siguiente (c_{n+1}) que esta celda es W.
ATRO	Salida	Habilitar salida de dato.
AURO	Salida	Indicar a celda anterior (c_{n-1}) que esta celda es R.
AUWO	Salida	Indicar a celda anterior (c_{n-1}) que esta celda es W.
DSSO	Salida	Indicar a celda siguiente (c_{n+1}) *status* de esta celda.
EAR	Salida	Habilitar registro de direcciones.
EM	Salida	Habilitar celda de memoria.
MR	Salida	Restablecer la memoria a 00H.
MS	Salida	Establecer la memoria a FFH.
RO	Salida	Esta celda (c_n) es dirección de escritura R.
DSSO	Salida	Indicar a celda anterior (c_{n-1}) *status* de esta celda.
WO	Salida	Reservada.
X0C	Salida	Desplazamiento de direcciones; "1", izquierda; "0", derecha.
X1C	Salida	"1", crea dirección de lectura D.
X2C	Salida	"1", crea dirección de lectura R.
X3C-X4C	Salida	Selecciona datos de:
		"00", celda anterior (c_{n-1})
		"01", dato MIDI
		"10", reservada
		"11", celda siguiente (c_{n+1}).

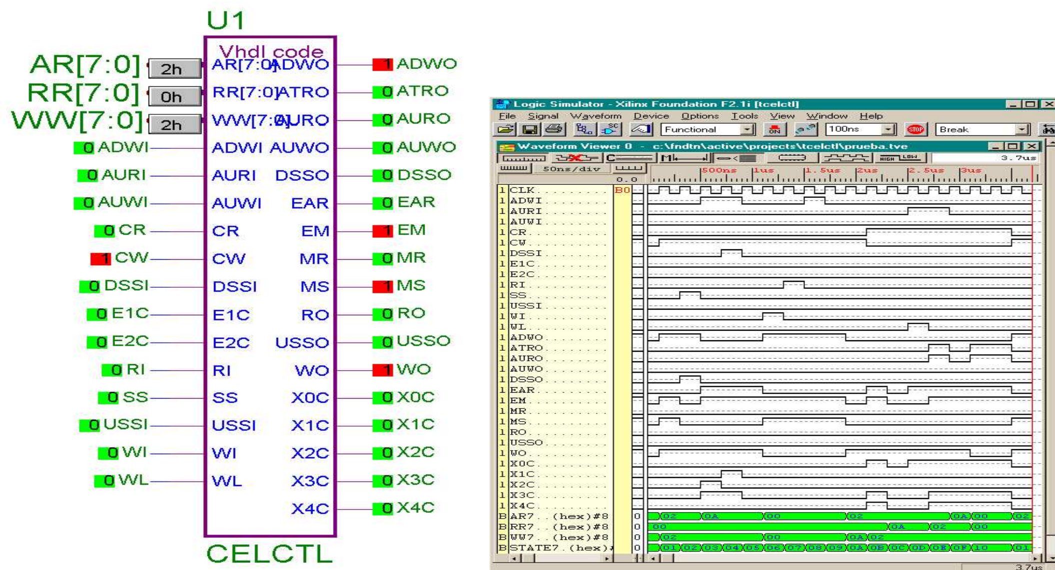


Figura B.3: Componente CELCTL: celda de control de memoria.

B.3. Código de FMUXSCH.

```
library ieee;
use ieee.std_logic_1164.all;
entity FMUXSCH is port
(
     CLK,CE,s0,s1,s2: in  std_logic;
 A,B: in  std_logic_vector(7 downto 0);
   C: out std_logic_vector(7 downto 0)
);
end FMUXSCH;

architecture rtlfmuxsch of FMUXSCH is
signal sel: std_logic_vector(2 downto 0);
signal TC: std_logic_vector(7 downto 0);

begin process (CLK,CE,S0,S1,S2,sel,A,B) begin
sel(0)<=s0; --X0C
sel(1)<=s1; --X1C
sel(2)<=s2; --X2C
case sel is
     when "000" => TC<=A; -- shift right, recibe direccion celda izquierda.
     when "001" => TC<=B; -- shift left, recibe direccion celda deracha.
     when "010" => TC(2 downto 0) <="011";
   TC(7 downto 3) <= A(7 downto 3); --direccion dato.
     when "100" => TC(2 downto 0) <="100";
   TC(7 downto 3) <= A(7 downto 3); --direccion lectura
     when others =>TC<=A;
```

```
end case;
end process;

process(CLK) begin
 if (CLK'event and CLK='1') then
     if (CE='1') then C <= TC; end if;
 end if;
end process;
end rtlfmuxsch;
```

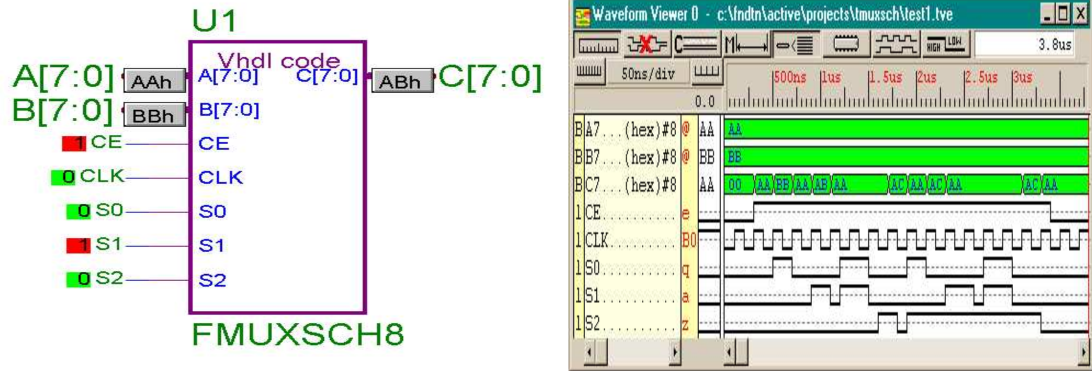


Figura B.4: Componente selector y generador de direcciones FMUXSCH8.

Tabla B.4: Estados de FMUXSCH8 para generar direcciones y desplazamientos.

$S2$	$S1$	$S0$	FUNCIÓN	
0	0	0	Recibir dirección de celda posterior,	C = A
0	0	1	Recibir dirección de celda anterior,	C = B
0	1	1	Crear dirección de dato D,	C$\leftarrow D$
1	0	0	Crear dirección de lectura R,	C$\leftarrow R$

B.4. Código de FMUXM8.

```
library ieee;
use ieee.std_logic_1164.all;
entity FMUXM8 is port
(
       S0,S1: in  std_logic;
       DUP,DMIDI,DWN: in  std_logic_vector(7 downto 0);
   Y: out std_logic_vector(7 downto 0)
);
end FMUXM8;

architecture RTL of FMUXM8 is
```

```
signal selector: std_logic_vector(1 downto 0);

begin process (S0,S1,DUP,DMIDI,DWN,selector) begin
      selector(0)<=S0; --X3C
      selector(1)<=S1; --X4C
      case (selector) is
    when "00" => Y <= DUP;
    when "01" => Y <= DMIDI;
    when "11" => Y <= DWN;
    when others => Y <=X"00";
      end case;
end process;
end RTL;
```

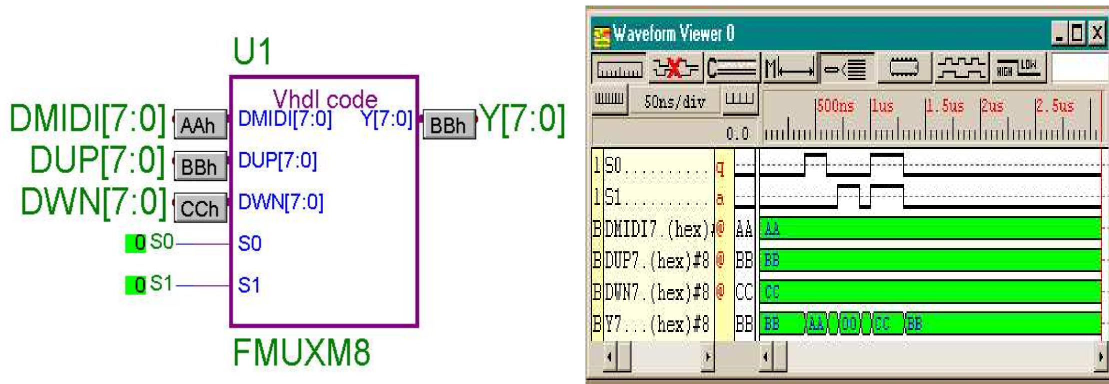


Figura B.5: Componente selector de datos de entrada a la memoria FMUXM8.

Tabla B.5: Estado de FMUXM8 para seleccionar la fuente de datos.

S1	S0	FUNCIÓN	
0	0	Recibir dato de celda anterior	Y←DUP
0	1	Recibir dato MIDI	Y←DMIDI
1	0	No se usa	
1	1	Recibir dato de celda posterior	Y← DWN

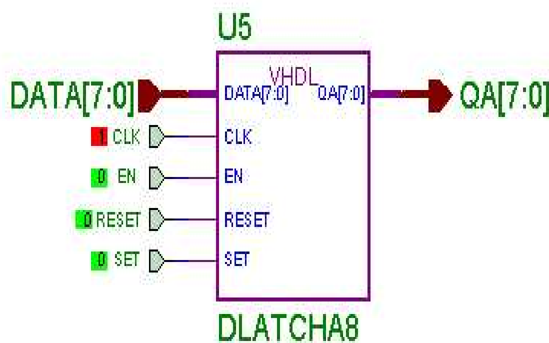
B.5. Código de DLATCHA8.

```
library ieee;
use ieee.std_logic_1164.all;

entity DLATCHA is  port
(
     DA: in  std_logic_vector(7 downto 0);
```

```
        CLK,SET,RESET,EN: in  std_logic;
      QA: out std_logic_vector(7 downto 0)
);
end DLATCHA;

architecture RTL of DLATCHA is
begin process(CLK)begin
      if (CLK'event and CLK='1') then
  if (SET='1') then
      QA <= X"FF";
  elsif (RESET='1') then
      QA <= X"00";
  elsif (EN='1') then
      QA <= DA;
  end if;
      end if;
end process;
end RTL;
```



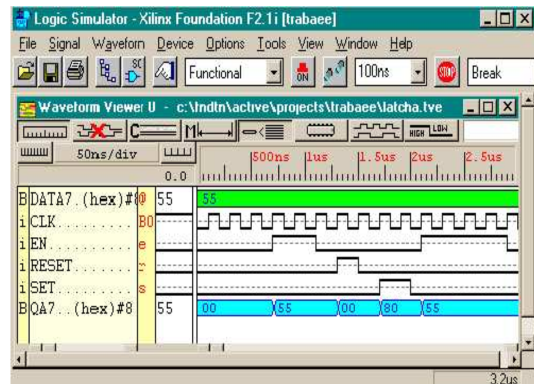


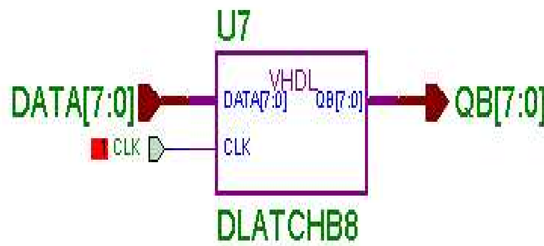
Figura B.6: Componente auxiliar de memoria DLATCHA8.

B.6. Código de DLATCHB8.

```
library ieee;
use ieee.std_logic_1164.all;
entity DLATCHB is  port
(
      DB: in  std_logic_vector(7 downto 0);
     CLK: in  std_logic;
      QB: out std_logic_vector(7 downto 0)
);
end DLATCHB;

architecture RTL of DLATCHB is
begin process(CLK)begin
      if (CLK'event and CLK='0') then
```

```
  QB <= DB;
      end if;
end process;
end RTL;
```



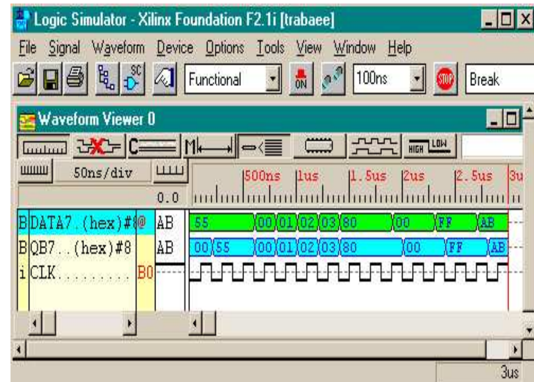


Figura B.7: Componente principal de memoria DLATCHB8.

B.7. Código de BUF8STAT.

```
library ieee;
use ieee.std_logic_1164.all;

entity BUFFERO is port
(
     BIN: in  std_logic_vector(7 downto 0);
      OE: in  std_logic;
    BOUT: out std_logic_vector(7 downto 0)
);
end BUFFERO;

architecture RTL of BUFFERO is
      signal temporal: std_logic_vector (7 downto 0);
begin process(BIN,OE,temporal) begin
      temporal<=BIN;
      if (OE='1') then
  BOUT<=temporal;
      else
  BOUT<="ZZZZZZZZ";
      end if;
end process;
end RTL;
```

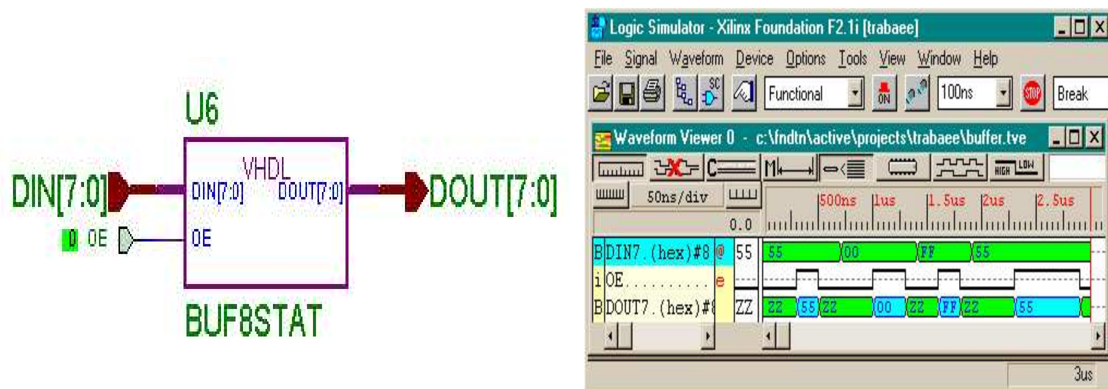


Figura B.8: Componente *buffer* de tercer estado BUF8STAT.

B.8. Programa del control de prueba MAQUINA

```
----------------------------------------------------------
-- maquina - maquina de estados de prueba.
----------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;

entity MAQUINA is
port(
      CLK: IN   std_logic;
     MIDI: OUT  std_logic_vector(7 downto 0);
     CELL: OUT  integer range 0 to 255;
  SEGMENT: OUT  integer range 0 to 16;
    RESET: OUT  std_logic;
    WRITE: OUT  std_logic;
     READ: OUT  std_logic;
    CLOCK: OUT  std_logic;
    STATE: OUT  integer range 0 to 255
);
end MAQUINA;

architecture RTL of MAQUINA is
signal estado: integer range 0 to 255;
begin process (CLK) begin

if (CLK'event AND CLK='1') then
    case estado is
     when 0   => estado <= 1;
                MIDI     <= X"00";
                CELL     <=  0;
                SEGMENT  <=  0;
                RESET    <= '0';
                WRITE    <= '0';
                READ     <= '0';
```

```
      when 1  => estado <= 2;
                 RESET  <='1';
      when 2  => estado <= 3;
                 RESET  <='0';
      when 3  => estado <= 4;
      when 4  => estado <= 5;
      when 5  => estado <= 6;
      when 6  => estado <= 7;
      when 7  => estado <= 8;
      when 8  => estado <= 9;
      when 9  => estado <= 10;
      when 10 => estado <= 11;
      when 11 => estado <= 12;
      when 12 => estado <= 13;
      when 13 => estado <= 14;
      when 14 => estado <= 15;
      when 15 => estado <= 16;
      when 16 => estado <= 17;
      when 17 => estado <= 18;
      when 18 => estado <= 19;
      when 19 => estado <= 20;
      when 20 => estado <= 21;
-- Escribir 2 datos al segmento 1
      when 21 => WRITE   <='1';
                 SEGMENT <= 1;
                 MIDI    <= X"D1";
                 estado  <= 22;
-- Escribir 2 datos al segmento 2
      when 22 => WRITE   <='1';
                 SEGMENT <= 2;
                 MIDI    <=X"D2";
                 estado  <= 23;

      when 23 => WRITE   <='1';
                 SEGMENT <= 2;
                 MIDI    <=X"D3";
                 estado  <= 24;
-- Escribir 2 datos al segmento 3
      when 24 => WRITE   <='1';
                 SEGMENT <= 3;
                 MIDI    <=X"D4";
                 estado  <= 25;

      when 25 => WRITE   <='1';
                 SEGMENT <= 3;
                 MIDI    <=X"D5";
                 estado  <= 26;
-- Escribir 2 datos al segmento 4
      when 26 => WRITE   <='1';
                 SEGMENT <= 4;
                 MIDI    <=X"D6";
                 estado  <= 27;

      when 27 => WRITE   <='1';
                 SEGMENT <= 4;
```

```
                MIDI    <=X"D7";
                estado  <= 28;
-- Escribir 3 datos al segmento 5
     when 28 => WRITE   <='1';
                SEGMENT <= 5;
                MIDI    <=X"D8";
                estado  <= 29;

     when 29 => WRITE   <='1';
                SEGMENT <= 5;
                MIDI    <=X"D9";
                estado  <= 30;

     when 30 => WRITE   <='1';
                SEGMENT <= 5;
                MIDI    <=X"DA";
                estado  <= 31;
-- Apagar WRITE
     when 31 => WRITE   <='0';
                SEGMENT <= 1;
                estado  <= 32;
-- Leer 1 dato del segmento 1.
     when 32 => READ    <='1';
                SEGMENT <= 1;
                estado  <= 33;

     when 33 => READ    <='0';
                SEGMENT <= 1;
                estado  <= 34;
-- Leer 1 dato del segmento 2.
     when 34 => READ    <='1';
                SEGMENT <= 2;
                estado  <= 35;

     when 35 => READ    <='0';
                SEGMENT <= 2;
                estado  <= 36;
-- Leer 1 datos del segmento 3.
     when 36 => READ    <='1';
                SEGMENT <= 3;
                estado  <= 37;

     when 37 => READ    <='0';
                SEGMENT <= 3;
                estado  <= 38;


-- Apagar READ
     when 38 => READ    <='0';
                SEGMENT <= 0;
                estado  <= 39;
-- Estado final.
     when 39 => estado <= 39;

     when others => estado <= 0;
```

```
    end case;
 end if;


end process;
process(CLK,estado) begin
  if CLK='1' then  CLOCK <= '1'; else CLOCK<='0'; end if;
  STATE<=estado;
end process;
end RTL;
```

Tabla B.6: Señales de MAQUINA.

TERMINAL	E/S	FUNCIÓN
CLK	Entrada	Reloj del sistema.
CELL	Salida	Dirección base de ventana (1-255).
MIDI	Salida	Dato de 8 bits.
SEGMENT	Salida	No. de segmento (1-16).
STATE	Salida	Estado del autómata
CLOCK	Salida	Reloj (copia de CLK).
READ	Salida	"1", lectura.
RESET	Salida	"1", reset de memoria.
WRITE	Salida	"1", escritura.

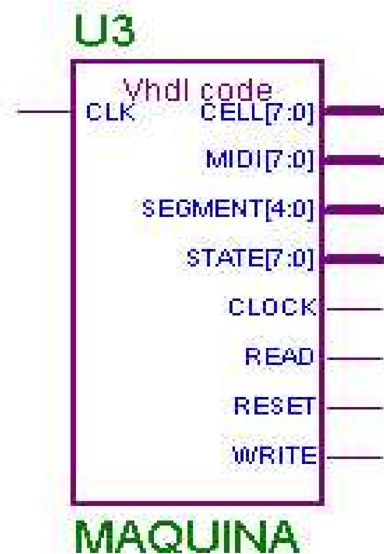
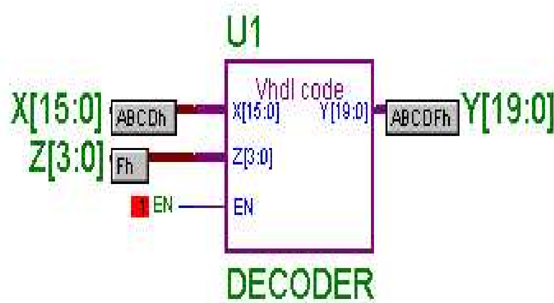


Figura B.9: Componente MAQUINA que genera los estados de MEMSEG para pruebas.

B.9. Código del decodificador de memoria lineal DECODER.

```
library ieee;
use ieee.std_logic_1164.all;
entity DECODER is  port
(
       X: in  std_logic_vector(15 downto 0);
       Z: in  std_logic_vector(3  downto 0);
      EN: in  std_logic;
       Y: out std_logic_vector(19 downto 0)
);
end DECODER;

architecture rtl of DECODER is
--signal tmp: std_logic_vector(19 downto 0);
begin process (X,Z,EN) begin
   if (EN='1') then
       Y(19 downto 4) <=X(15 downto 0);
       Y( 3 downto 0) <=Z( 3 downto 0);
   else
       Y<="ZZZZZZZZZZZZZZZZZZZZ";
   end if;
end process;
end rtl;
```



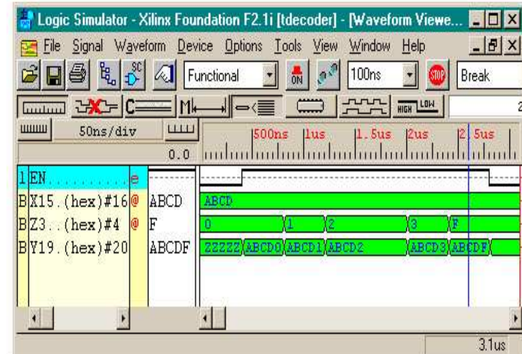


Figura B.10: Decodificador de direcciones DECODER.

Φ

Apéndice C

Notas, tonos y acordes

C.1. Nota musical

Tradicionalmente, una nota musical está definida por la función $N(f, r, m)$ cuyos parámetros son f, r y m, donde f es la frecuencia (tono) de la nota, r es el tiempo de duración de dicha nota (ritmo) y m es la melodía asociada a la frecuencia de la nota, tal que permite tocar varias en paralelo. El nombre de las notas son:

Notación europea con sostenidos: DO, $DO\sharp$, RE, $RE\sharp$, MI, FA, $FA\sharp$, SOL, $SOL\sharp$, LA, $LA\sharp$ y SI. Con bemoles: DO, $RE\flat$, RE, MI$\flat$, MI, FA, $SOL\flat$, SOL, $LA\flat$, LA, $SI\flat$ y DO; Notación americana con sostenidos: C, $C\sharp$, D, $D\sharp$, E, F, $F\sharp$, G, $G\sharp$, $A\sharp$ y B. Con bemoles: C, $D\flat$, D, $E\flat$, E, F, $G\flat$, G, $A\flat$, A, $B\flat$, B y C.

Los tonos de 2 notas están separadas por la constante $2^{\frac{1}{12}} = 1{,}059463094\ldots$, en general, la frecuencia f_n de una nota musical está dada por la expresión C.1

$$f_n = 2^{\frac{n}{12}} f_{ref} \tag{C.1}$$

donde, f_{ref} es la frecuencia de referencia, y n es el número de semitonos a partir de dicha referencia [71][72]. Por ejemplo, si $f_{ref} = 440$, la frecuencia de $DO_5\sharp$ ($RE_5\flat$) es de $440(2^{\frac{4}{12}}) = 554{,}365...Hz.$, debido a que está separada hacia arriba por una tercera (+4 semitonos); la nota DO_4 que está una sexta mayor hacia abajo (-9 semitonos) tiene una frecuencia de $440(2^{\frac{-9}{12}}) = 261{,}6255...Hz$. Por acuerdo internacional, la frecuencia de audio de 440Hz. está asociada a la nota LA_4 (A_4) situada en la octava

cuatro (subíndice) de un piano de referencia [73]. La Tabla C.1 muestra la frecuencia de cada tono, la nota, y su octava.

Tabla C.1: Correspondencia entre: frecuencia (*hertz*) de nota(i), posición en octava(j), y tono.

	C	D♭	D	E♭	E	F	G♭	G	A♭	A	B♭	B
	C	C♯	D	D♯	E	F	F♯	G	G♯	A	A♯	B
	DO	RE♭	RE	MI♭	MI	FA	SOL♭	SOL	LA♭	LA	SI♭	SI
	DO	DO♯	RE	RE♯	MI	FA	FA♯	SOL	SOL♯	LA	LA♯	SI
	(i)											
(j)	0	1	2	3	4	5	6	7	8	9	10	11
0	16.35	17.32	18.35	19.44	20.60	21.82	23.12	24.49	25.95	27.50	29.13	30.86
1	32.70	34.64	36.70	38.89	41.20	43.65	46.24	48.99	51.91	55.00	58.27	61.73
2	65.40	69.29	73.41	77.78	82.40	87.30	92.49	97.99	103.82	110.00	116.54	123.47
3	130.81	138.59	146.83	155.56	164.81	174.61	184.99	195.99	207.65	220.00	233.08	246.94
4	261.62	277.18	293.66	311.12	329.62	349.22	369.99	391.99	415.30	440.00	466..16	493.88
5	523.25	554.36	587.32	622.25	659.25	698.45	739.98	783.99	830.60	879.99	932.32	987.76
6	1046.50	1108.73	1174.65	1244.50	1318.51	1396.91	1479.97	1567.98	1661.21	1759.99	1864.65	1975.53
7	2093.00	2217.46	2349.31	2489.01	2637.02	2793.82	2959.95	3135.96	3322.43	3519.99	3729.31	3951.06

C.2. La octava MIDI

Una octava MIDI, denotada por O u O', es un conjunto de doce notas asociadas a una sección del teclado de un sintetizador; con sostenidos (♯), expresión C.2:

$$O = \{C, C\sharp, D, D\sharp, E, F, F\sharp, G, G\sharp, A, A\sharp, B\}, \quad (C.2)$$

o bien con bemoles (♭), expresión C.3:

$$O' = \{C, D\flat, D, E\flat, E, F, G\flat, G, A\flat, A, B\flat, B\}. \quad (C.3)$$

El teclado de un sintetizador, denotado por P, es un conjunto de octavas O_n concatenadas dentro del espacio del teclado, expresión C.4,

$$P = \{O_0, O_1, O_2, \ldots, O_n\}, \quad (C.4)$$

para un sintetizador MIDI de concierto con ocho octavas, $n = 7$; donde n es el número de octava. Los tonos de la notas de O_7 (octava siete) tienen el doble de frecuencia de los tonos de las notas que pertenecen a O_6 (octava seis); asimismo, las de O_5 tienen el doble de O_4, y así sucesivamente.

C.3. El tono MIDI

Un tono MIDI es el elemento X_{ij} de la matriz $\mathbf{X}$ cuyo número entero $n = X_{ij}$ es asignado a cada tecla de un sintetizador MIDI o piano electrónico. Los elementos de la matriz están agrupados por notas y octavas; donde: $i = 0, 1, 2, \ldots, 11$ es la nota de la escala musical; $j = 0, 1, 2, \ldots, 7$ es la octava que contiene la nota; y $n = 0, 1, 2, \ldots, 95$ es el tono X_{ij} de la nota, tal como se muestra en la expresión (C.5),

$$\mathbf{X} = \begin{bmatrix} X_{0,0} & X_{1,0} & \ldots & X_{11,0} \\ X_{0,1} & X_{1,1} & \ldots & X_{11,1} \\ \vdots & \vdots & X_{i,j} & \vdots \\ X_{0,7} & \ldots & \ldots & X_{11,7} \end{bmatrix}; \tag{C.5}$$

con base en octavas y notas, la notación X_{ij} permite a los sistemas MIDI transmitir y recibir tonos y notas sin importar la octava [31]. Aplicando el teorema del residuo, y dado que $0 \leq n \leq 95$, se obtiene que, el tono n de la nota de una octava está dada por la expresión (C.6)

$$n = j(12) + i, \tag{C.6}$$

donde n es el tono, el cociente es la octava (j), y el residuo es la nota (i). La Tabla C.2 y la Figura C.1, muestran esta relación.

Tabla C.2: Relación n entre octavas(j), notas(i) y tonos(n), dado por $n = j(12) + i$.

	C	D♭	D	E♭	E	F	G♭	G	A♭	A	B♭	B
	C	C♯	D	D♯	E	F	F♯	G	G♯	A	A♯	B
	DO	RE♭	RE	MI♭	MI	FA	SOL♭	SOL	LA♭	LA	SI♭	SI
	DO	DO♯	RE	RE♯	MI	FA	FA♯	SOL	SOL♯	LA	LA♯	SI
	Nota(i)											
Octava(j)	0	1	2	3	4	5	6	7	8	9	10	11
0	0	1	2	3	4	5	6	7	8	9	10	11
1	12	13	14	15	16	17	18	19	20	21	22	23
2	24	25	26	27	28	29	30	31	32	33	34	35
3	36	37	38	39	40	41	42	43	44	45	46	47
4	48	49	50	51	52	53	54	55	56	57	58	59
5	60	61	62	63	64	65	66	67	68	69	70	71
6	72	73	74	75	76	77	78	79	80	81	82	83
7	84	85	86	87	88	89	90	91	92	93	94	95

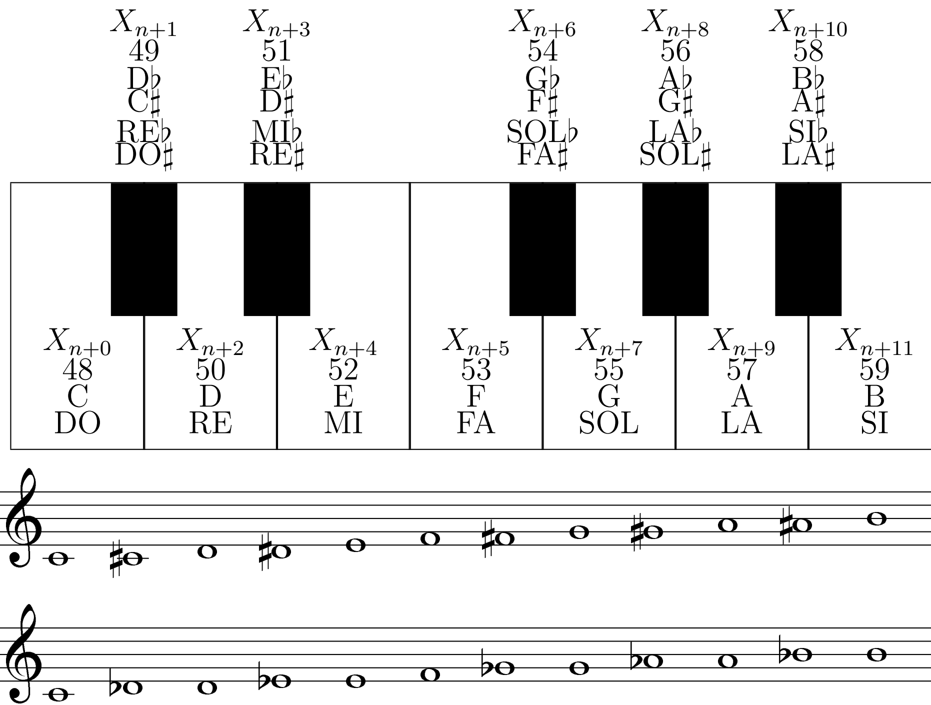


Figura C.1: Nombres, símbolos, variables, y teclas de un piano (octava 4).

C.4. Las escalas MIDI

De acuerdo con la Figura C.1, y teniendo en cuenta la notación que corresponde a cada una de las teclas (blancas y negras) de un segmento del teclado de piano, considérese lo siguiente:

C.4.1. Escala cromática MIDI

Una escala cromática MIDI está definida por el conjunto E_c cuyos elementos son tonos consecutivos n, $n+1$, $n+2$, ..., $n+12$, tal que n es llamado *tono raíz* donde comienza la escala (Expresión C.7),

$$E_c = \{n, n+1, n+2, n+3, n+4, n+5, n+6, n+7, n+8, n+9, n+10, n+11, n+12\}, \tag{C.7}$$

por ejemplo, para el caso específico del segmento de teclado mostrado en la Figura C.1; si $i = 0$, $j = 4$, entonces el *tono raíz* es $n = i + j(12) = 48$, por lo que, la escala

comienza con la nota C_4 (DO_4) de la octava 4. Es decir, con sostenidos: C_4, $C_4\sharp$, D_4, $D_4\sharp$, E_4, F_4, $F_4\sharp$, G_4, $G_4\sharp$, A_4, $A_4\sharp$, B_4, C_5. Con bemoles: C_4, $D_4\flat$, D_4, $E_4\flat$, E_4, F_4, $G_4\flat$, G_4, $A_4\flat$, A_4, $B_4\flat$, B_4, C_5. Los tonos X_{ij} de la matriz son: 48, 49, 50, 51, 52, 53, 54 55, 56, 57, 58, 59, y 60. Para este caso se consideran todas las teclas (blancas y negras).

C.4.2. Escala MIDI en modo mayor

Asimismo, una escala MIDI en modo mayor está definida por el conjunto E_M cuyos elementos son tonos consecutivos n, $n+2$, $n+4$, $n+5$, $n+7$ $n+9$, $n+11$, y $n+12$, tal que n es el primer elemento llamado *tono raíz* donde comienza la escala (expresión C.8),

$$E_M = \{n+0, n+2, n+4, n+5, n+7, n+9, n+11, n+12\}, \qquad \text{(C.8)}$$

por ejemplo, si $i = 0$, $j = 4$, el *tono raíz* es $n = i + j(12) = 48$ entonces, la escala empieza con la nota C_4 (DO_4). Es decir, C_4, D_4, E_4, F_4, G_4, A_4, B_4, C_5. Los tonos X_{ij} de la matriz son: 48, 50, 52, 53, 55, 57, 59 y 60. Las teclas negras la Figura C.1 no están consideradas para este caso.

C.4.3. Escala MIDI en modo menor

De la misma forma, una escala MIDI en modo menor está definida por el conjunto E_m cuyos elementos son tonos consecutivos n, $n+2$, $n+3$, $n+5$, $n+7$, $n+8$, $n+10$, $n+12$, tal que n es llamado el *tono raíz* donde comienza la escala (expresión C.9),

$$E_m = \{n+0, n+2, n+3, n+5, n+7, n+8, n+10, n+12\}, \qquad \text{(C.9)}$$

por ejemplo, si $i = 0$, $j = 4$, entonces el *tono raíz* es $n = i + j(12) = 48$, por lo que, la escala comienza con la nota C_4 (DO_4) de la octava 4. Esto es, con sostenidos: C_4, D_4, $D_4\sharp$, F_4, G_4, $G_4\sharp$,$A_4\sharp$, C_5; con bemoles: C_4, D_4, $E_4\flat$, F_4, G_4, $A_4\flat$, $B_4\flat$, C_5. Los tonos X_{ij} de la matriz son: 48, 50, 51, 53, 55, 56, 58, y 60. Para este caso se consideran algunas teclas negras.

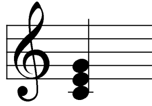


Figura C.2: Acorde de DO mayor (C_M).

C.4.4. Acorde MIDI

Si dos o más notas se tocan al mismo tiempo, entonces se puede construir una estructura de acorde. Así, un acorde mayor está definido como:

$$A_M = \{n + 0, n + 4, n + 7\} \tag{C.10}$$

donde: n es el tono raíz del acorde. Por ejemplo, si n=48, el acorde de DO mayor (C_M) tendrá las notas DO, MI, y SOL (C,E,G) y los números MIDI 48_H, 52_H, y 55_H; en la Figura C.2 se muestra la notación musical para esta estructura.

C.5. Archivo fuente `poli.kl` para la partitura de la figura 6.4

```
titulo(POLIT\'ECNICO)
autor(P\'erez Prado)
date(2-12-76)
inst(Sax Alto)
tempo(c)
tono(fa mayor)
clave ( sol )

--inicio de cancion

--compas 0
sbeam(s, 1, 0, do)
ebeam(1) do

--compas 1
barra(lr)
setbarra(1)
dacapo(up)
re
sbeam(s, 1, 9, mi)
ebeam(1) sol
do(5, dn)
sbeam(s, 1, 0, do)
ebeam(1) do

--compas 2
barra
setbarra(2)
re fa sil
sbeam(s, 1, 0, do)
ebeam(1) do

--compas 3
barra
re mi sil
sbeam(s, 1, 0, do)
ebeam(1) do

--compas 4
barra
re fa sil
sbeam(s, 1, 0, do)
ebeam(1) do

--compas 5
barra(rr)
re fa sil
sbeam(s, 1, 0, do)
ebeam(1) do

--compas 6
barra(lr)
setbarra(6)
re
sbeam(s, 1, 9, mi)
ebeam(1) sol
do(5, dn)
sbeam(s, 1, 0, do)
ebeam(1) do

--compas 7
barra
re fa sil
sbeam(s, 1, 0, do)
ebeam(1) do

--compas 8
barra
re mi sil
sbeam(s, 1, 0, do)
ebeam(1) do

--compas 9
barra
re fa sil
sbeam(s, 1, 0, do)
ebeam(1) do

--compas 10
setplica(dn)
barra(rr)
setbarra(10)
sbeam(s, 1, 2, do(5))
do(5) ebeam(1) re(5)
sil(c) sil(h)

--compas 11
barra
sbeam(s, 1, 0, si) si
si ebeam(1) si
sbeam(s, 1, 2, si) si
```

```
do(5) ebeam(1) do(5)

--compas 12
setplica(up)
barra
re(5,c, dn) sil(c) do
fa sil

--compas 13
barra
do fa
sbeam(s, 1, 0, fa)
ebeam(1) fa sil

--compas 14
barra
sil  sbeam(s, 1, 0, fa)
ebeam(1) fa
sbeam(s, 1, 5, la)
la do(5)
ebeam(1) do(5)

--compas 15
barra(lr)
setbarra(15)
sbeam (s, 1, -7, re(5))

ebeam(1) si sil
sbeam (s, 1, -7, re(5))

ebeam(1) si sil

--compas 16
barra
sbeam(s, 1, 0, mi)
ebeam(1) mi
sbeam(s, 1, 5, sol)
sol
si ebeam(1) si

--compas 17
barra(lr)
setbarra(17)
sbeam (s, 1, -7, re(5))

ebeam(1) si sil
sbeam (s, 1, -7, re(5))

ebeam(1) si sil

--compas 18
barra
sil  sbeam(s, 1, 0, fa)
ebeam(1) fa
sbeam(s, 1, 5, la)
la do(5)
ebeam(1) do(5)

--compas 19
barra
sil(w)

--compas 20
barra
do(5, h) si(h)

--compas 21
barra
do(5) sil(c)
sol(c) si sil

--compas 22
barra
repite

--compas 23
barra
do(5, h) si(h)

--compas 24
barra
do(5) do(5) sil(h)

--compas 25
barra
do(5, h) si(h)

--compas 26
barra
do(5) do(5)  sil
sbeam(s, 1, 0, sol)
ebeam(1) sol

--compas 27
barra
sbeam(s, 1, 0, do(5))

ebeam(1) do(5)
si sil(h)

--compas 28
barra
sil(h) sil
sbeam(s, 1, 0, sol)
ebeam(1) sol

--compas 29
barra
sbeam(s, 1, 0, do(5))
```

```
ebeam(1) do(5)
si sil(h)

--compas 30
barra
do(5) fa(5) sil(h)

--compas 31
barra
ojo(up)
sil(w)

--compas 32
barra
sil(h) sil
sbeam(s, 1, 0, do)
ebeam(1) do

--compas 33
barra
dacapo(md)
ojo(md)

--compas 34
barra
ojo(up)
do(5, h) si(h)

--compas 35
barra
do(5, rp) sol(c)
re(5) sil

--compas 36
barra
repite

--compas 37
barra
do(5, h) si(h)

--compas 38
barra
do(5, rp) sol(c)
re(5) sil

--compas 39
barra
repite

--compas 40
barra
repite

--compas 41
barra
repite

--compas 42
barra
repite

--compas 43
barra
do(5, h) si(h)

--compas 44
barra
do(5, rp) sol(c)
 re(5) sil

--compas 45
barra
si(w) stie(1, si)

--compas 46
barra
etie(1) si(c) cresendo ( 3 )
sil(c) sil sil(h)

--compas 47
barra
sol sbeam(s, 1, 8, sol)
stie (1, do(5))
ebeam(1) do(5) etie(1)
do(5, c)
si stie(1, do(5,c))

--compas 48
barra
etie(1) do(5) si
re(5) si sil(c)

--compas 49
barra
sol sbeam(s, 1, 8, sol)
stie (1, do(5))
ebeam(1) do(5) etie(1)
do(5, c)
si stie(1, do(5,c))

--compas 50
barra
etie(1) do(5) si
re(5) si sil(c)

--compas 51
barra
repite
```

```
--compas 52
barra(lr)
setbarra(52)
sol sbeam(s, 1, 8, sol)
stie (1, do(5))
--do(5)
ebeam(1) do(5) etie(1)
do(5, c)
si stie(1, do(5,c))
--do(5,c)

--compas 53
barra
etie(1) do(5) si
--re(5) si sil(c)

--compas 54
barra
sol sbeam(s, 1, 8, sol)
stie (1, do(5))
--do(5)
ebeam(1) do(5) etie(1)
do(5, c) si do(5,c)

--compas 55
barra
setplica(dn)
sil(c)
sbeam(s, 1, 0, si)
ebeam(1) si sil(c)
sil(c)
sbeam(s, 1, 0, si)
ebeam(1) si sil(c)

--compas 56
barra(rr)
setbarra(56)
mi(5,f)
sbeam(s, 1, 0, mi(5,f))
ebeam(1) mi(5,f)
sbeam(s, 1, 0, re(5,f))
re(5,f) ebeam(1)
mi(5,f)
sil(c)

--compas 57
barra
ojo(up)
sil(w)

--compas 58
barra
mi(5, rp,f)
stie (1, mi(5))
mi(5,c,f)
etie(1) mi(5, f)
la(5, f)

exit
```

MI NENA

M. Peña-Guerrero.

1977

1er. Sax Alto

Figura C.3: Partitura generada con KL.

1

Descripción formal del lenguaje KL

```
<statment>      :: <notas>*|<commands>*
<notes>         :: <notac>
<accidente>     :: b|#
<numero>        :: <-integer>..<+integer>
<digito>        :: 0|1|2|3|4|5|6|7
<tiempo>        :: w|h|n|c|f|sf
<volumen>       :: p|pp|ppp|f|ff|fff
<plica>         :: up|dn
<bcuadro>       :: bc
<flat>          :: tst
<circunf>       :: cir
<puntos>        :: pt|rp
<tbarra>        :: s|d|rl|rr|rlr
<modificadores> :: (<numero>,<tiempo>,<volumen>
                    ,<plica>,<bcuadro>,<flat>
                    ,<circunf>,<puntos>)
<nota>          :: do|re|mi|fa|sol|la|si
<notac>         :: nota[accidente]
                     [modificadores]
<commands>      :: <command>
<command>       :: <autor><barra><casilla>
                   <clave><comment><cresendo>
                   <dacapo><date><decresendo>
                   <dupla><ebeam><etie>
                   <exit><inst><liga>
                   <melodia><newline><ojo>
                   <repite><sil>/<rest>
                   <sbeam><setbarra><setclave>
                   <setplica><stie><tempo>
                   <titulo><tono><triada>
                   <tupla>

<autor>         :: <autor><(><string><)>
<barra>         :: <barra>[<(><tbarra><)>]
<casilla>       :: <casilla><(><1|2><)>
<clave>         :: <clave><(><sol|fa><)>
<comment>       :: <comment><string>
<cresendo>      :: <cresendo><(><1|2|3|4><)>
<dacapo>        :: <dacapo><(><up|md|dn><)>
<date>          :: <date><(><string><)>
<decresendo>    :: <decresendo><(><1|2|3|4><)>
<dupla>         :: <dupla><(><s|d|t><plica>
                   <nota><nota><)>
<ebeam>         :: <ebeam><(><numero><,>
                   <nota><)>
<etie>          :: <etie><(><1|2|3|4|5|6|7|8|
                   9|0><)>
<exit>          :: <exit>
<inst>          :: <inst><(><string><)>
<liga>          :: <liga><(><statment><)>
<melodia>       :: <melodia><(><statment><)>
<newline>       :: <newline>
<ojo>           :: <ojo><(><plica><)>
<repite>        :: <repite>
<sil>/<rest>    :: <sil>/<rest><(><timepo><)>
<sbeam>         :: <sbeam><(><s|d|t><numero>
                   <numero><nota><)>
<setbarra>      :: <setbarra><(><numero><)>
<setclave>      :: <setclave><(><sol|fa><)>
<setplica>      :: <setplica><(><up|dn><)>
<stie>          :: <stie><(><numero><notac><)>
<tempo>         :: <tempo><(><num/num|c|C><)>
<titulo>        :: <titulo><(><string><)>
<tono>          :: <tono><(><nota>[<accidente>]
                   <mayor|memor><)>
<triada>        :: <triada><(><tbarra><notac>
                   <notac><notac><)>
<tupla>         :: <tupla><(><s|d|t><notac>
                   <notac><notac><notac><)>
```

Figura C.4: Descripción Backus-Naur Form de KL.

1

Programa Fuente para KL

```
-- Este es un programa de prueba del sistema KL.
-- Prueba comandos y notas.
-- Septiembre de 1998.
--
-- Encabezado.
--
titulo(Maximino Pena Guerrero)
--
-- Notas
--
do re mi fa sol la si
do(1,w)re(h)mi(7,w)fa(p)sol(pp)la(4,p)si(5)
do#(1,w)re#(h)mib(7,w)fa#(p)sol#(pp)la#(4,p)sib(5)
do(1,w) re(h) mi(7,w) fa(p) sol(pp) la(4,p) si(5)
do#(1,w) re#(h) mib(7,w) fa#(p) sol#(pp) la#(4,p) sib(5)
--
--Comandos
--
barra rest sil
rest(h) barra
barra sil(w) rest(w) barra
sil(w) rest(w) rest(w) rest(w)
--
-- Combinados.
--
barra do#(1,w)re#(h)mib(7,w)fa#(p)barra
barra sol#(pp)rest(w) la#(4,p)sib(5) barra
do(1,w) re(h) mi(7,w)rest(h) barra
barra fa(p) sol(pp) rest la(4,p) sil si(5)
do#(1,w) rest(w) sil(w) re#(h) mib(7,w)barra
barra fa#(p) sol#(pp) la#(4,p) sib(5)barra
```

Figura C.5: Programa fuente para el compilador KL.

Φ

Bibliografía

[1] *AES-51 Language Manual Version 1.3*, Advanced Educational Systems, USA, 1998. (p: 76)

[2] Aho Alfred V., John E. Hopcroft, Jeffrey D. Ullman, *The Design and Analysis of Computer Algoritms*, Addison-Wesley, USA, 1974. 470 pgs. (p: 20)

[3] Ashenden Peter J., *The Designer's Guide to VHDL*, Second Edition, Acadenic Press, MK Morgan Kaufmann Publishers, USA, 2002. 759 pgs. (p: 11)

[4] Anderton Craig, *MIDI for Musicians*, Amasco Publications, New York, London, Sydney, 1986. (p: 25)

[5] Axelson Jan, *USB Complete: Everything you need to develop custom USB peripherals*, Second Edition, Lakview Research. USA 2001. 523 pgs. (p: 32)

[6] Baggi Dennis L., (ed.) *Readings in Computer-Generated Music*, IEEE Computer Society Press tutorial, USA, 1992. 222 pgs. (p: 15)

[7] Baggi Dennis L., "NeurSwing: An Intelligent Workbench for the Investigation of Swing in Jazz", in *Readings in Computer-Generated Music*, IEEE Computer Society Press tutorial, USA, 1992. pp. 79-94. (p: 18)

[8] Baird Kevin C., "Generating Music Notation in Real Time", Linux Journal, December 2004, p. 72. (p: 31)

[9] Bergin Thomas J., Richard G. Gibson, eds. *History of Prgramming Languages*, ACM Press, Addison Wesley, USA, 1996. 864 pgs. (p: 20)

[10] Bhasker J., *A VHDL Primer*, Third Edition, Prentice Hall, USA, 1999. 373 pgs. (p: 11)

[11] Born Günter, *The File Formats Handbook*, International Thompson, Computer Press, 1995. 1274 pgs. (p: 25)

[12] Brown William D., Joe E. Brewer, ed., "Nonvolatile Semiconductor Memory Technology: A Comprehensive Guide to Understanding and Using NVSM Devices", IEEE Press Series on Microelectronic Systems, USA 1998. 589 pgs. (p: 10)

[13] Byrd Donald Alvin, "Music Notation by Computer", Thesis, Doctor of Philosophy, Department of Computer Science, Indiana University, August, 1984. 238 pgs. (p: 20)

[14] Ceppelletti Paolo, et. ed., *Flash Memories*, Academic Publishers, USA 1999, Fourth Printing 2002. 540 pgs. (p: 10)

[15] Chassing Rulph, *Digital Signal Processing: Laboratory Experiments Using C and the TMS320C31DSK*, John Wiley & Sons, USA 1999. 305 pgs. (p: 75)

[16] Conger Jim, *Sequensing in C*, M&T Publishing, Redwood City Calif., 1989. (p: 26)

[17] Cope David, "An Expert System for Computer-Assisted Composition", Computer Music Journal, Vol. 11, No. 4, Winter 1987, MIT, pp. 30-46. (p: 17)

[18] Delgado-Frias José G., Will R. Moore, (eds.), *VLSI for Artificial Intelligence*, Kluwer Academic Publishers, USA, 1989. 274 pgs. (p: 18)

[19] Delgado-Frias José G., Stamatis Vassiliadis, Gerald G. Pechanek, "A Processing Unit for Flexible Multiplexor Machine Organizations", IBM Tech. Rep. TR 01.C 417, 1992. (p: 9)

[20] De Luca P. A., M. Peña G., A. Jiménez F., "Memoria Escalable Multisegmentada Diseñada con VHDL", memorias, II Simposium Internacional, "Las Humanidades en la Educación Técnica Ante el Siglo XXI", ESIQUIE-IPN, memorias, septiembre 2000. (p: 44)

[21] De Luca P. A., M. Peña G., "Memoria de Alto Rendimiento Para Enrutamiento de Mensajes", Sociedad Mexicana de Instrumentación, SOMI XV, Guadalajara Jal. octubre 2000. (p: 51)

[22] De Luca P. A., M. Peña G., "Dispositivo Para Capturar Eventos MIDI en Paralelo", memorias, VII Conferencia de Ingeniería Eléctrica, CINVESTAV, CIE, memorias, septiembre 2001. (p: 66)

[23] De Luca P. A., M. Peña G., "Dispositivo de Captura y Digitalización Paralela MIDI", Congreso Nacional de Ingeniería en Computación y Sistemas Electrónicos, Universidad Autónoma de Tlaxcala, memorias, octubre 5, 2001. (p: 44)

[24] De Luca P. A., M. Peña G., "Captura de Eventos MIDI en Paralelo con FPGA's", Congreso Nacional de Instrumentación, SOMI XVI, Querétaro Qro. México, memorias, octubre 2001. (p: 37)

[25] Ercegovac Milôs D., Tomás Lang, Jaime H. Moreno, *Introduction to Digital Systems*, John Wiley & Sons, Inc, USA, 1999. (p: 74)

[26] Escobedo Pedro, "Zacatlán", Orq. C. Campos, D.R. 33 rpm, s.f. México. Versión MIDI: M. Peña G., 2002. (p: 74)

[27] Estrada Julio, Jorge Gil Mendieta. *Teoría de Música y Grupos Finitos (3 variables boolenas)*, Universidad Autónoma de México, Mex. 1984. (p: 16)

[28] Francois Jalbert, *MuTEX User's Guide*, (version 1.1), Rheinische Friedrich-Wilhelms University, Wegler Strabe 6 5300, Bonn Deutschland, 29 August 1989. (`cdr@stolaf.edu`) (p: 22)

[29] Garvin Mark, "Designing a Music Recorder", Dr. Dobb's Journal, May 1987. pp. 22-49. (p: 26)

[30] Gershenfeld Neil, *When Thinks Start to Think*, Henry Holt & Comp., USA, 2000. (p: 28)

[31] Gil Cepeda I. Fco., "Diseño de un Lenguaje Musical (MUSFOR) y Construcción de su Compilador (MUS80)". Inédita. México. Tesis presentada para aspirar al grado de Licenciado en Físico Matemáticas. Escuela Físico Matemáticas. Universidad Autónoma de Puebla. 1982. (p: 137)

[32] Gourlay John S., "A Language For Music Printing", Communications of the ACM, may 1986, v. 29, No. 5, pp. 388-401. (p: 21)

[33] González H. M., "Desarrollo de un Esqueleto Para Manejar Bases de Conocimiento Estratificado en Temas". Inédita. México. Tesis presentada para aspirar al grado de Doctor en Ciencias en Ingeniería Eléctrica. Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional, 1996. 103 pgs. (p: 17)

[34] Goossens Michel, Sebastian Rahtz, Frank Mittelbach, *LaTeX Graphics Companion: Illustrating Documents with TeX and PostScript*, Addison Wesley, USA, 1997. 556 pgs. (p: 22)

[35] Griffith Niall, Peter M. Todd, *Musical Networks: Parallel Distributed Perception and Performance*, MIT Press, 1999. 385 pgs. (p: 18)

[36] Helmholtz Hermann L.F., *On the Sensations of Tone: As a Physiological Basis for the Theory of Music*, 2th. ed. Ing., Dover Publications, Inc., New York, 1954. 576 pgs. (p: 20)

[37] Hiller Lejaren, Leonard Isaacson (1958), "Musical Composition with a High-Speed Digital Computer", in ed., Stephan M. Schwanauer and David A. Levitt, *Machine Models of Music*, USA, The MIT Press Cambridge, 1993. 544 pgs. (p: 17)

[38] *PCL-5 Printer Language Technical Reference Manual*, 1st. ed., HP Part No. 5961-0509, Hewlett-Packard, USA, 1992. (p: 23)

[39] *The HP-GL/2 and HP RTL Reference Guide: a handbook for program developers*, Adison Wesley, 3th. ed., Hewlett-Packard, USA, 1997. 516 pgs. (p: 23)

[40] Huber David Miles, *The MIDI Manual*, USA, SAMS, 1991 (5a. imp. 1994). 268 pgs. (p: 27)

[41] Hwang Kai, *Advanced Computer Architecture: Parallelism, Scalability, Programmability*, McGraw-Hill, Singapure, 1993. 770 pgs. (p: 28)

[42] IBM, *Technical Reference IBM PC*, IBM Inc., USA, 1981. (p: 15)

[43] Intel, *MCS 51 Microcontroller Family User's Manual*, USA, 1994. (p: 74)

[44] Itagaki Takebumi, Peter D. Manning, and Alan Purvis, "Distributed Parallel Processing: Lessons Learned from a 160 Transputer Network", Computer Music Journal, Volume 21, Number 4, pp. 42-54, Winter 1997. (p: 28)

[45] Jameson David H., "Building Real-Time Music Tools Visually with Sonet", in *Proceedings Real-Time Technology and Applications*, Symposium June 10-12, 1996, Brookline, Massachusetts, IEEE, Computer Society. pp. 11-18. (p: 7)

[46] Janzen Thomas E., “AlgoRhythms: Real-Time Algorithmic Composition for a Microcomputer”, in Denis Baggi, *Readings in Computer-Generated Music*, IEEE Computer Society Press tutorial, USA, 1992. pp. 199-209. (p: 32)

[47] Kassler Michel, “MIR -A Simple Programming Language for Musical Information Retrival”, chapter XVII, in ed. Harry B. Lincoln *The Computer and Music*, Cornell University Press, Ithaca and London, 1970. pp. 300-327. 354 pgs. (p: 21)

[48] Knuth Donald E., *TEX and METAFONT New Directions in Typesetting*, Digital Press, American Mathematical Society, 1979. (p: 22)

[49] Knuth Donald E., *The Art of Computer Programming: Fundamental Algorithms*, Volume 1, Third Edition, Addison Wesley, USA, 1997. 650 pgs. (p: 26)

[50] Leathers David, *Pro Tools Bible: Pro Tools 6.1 and Beyond*, McGraw-Hill, USA, 2004. 348 pgs. (p: 34)

[51] Llinás Rodolfo, *I of The Vortex: From the Neurons to Self*, The MIT Press, USA 2001. (p: 29)

[52] Linding Bos 1 Michel, “Una Arquitectura Para el Procesamiento Paralelo Basada en una Red Crossbar Modificada”. Inédita. México. Tesis presentada para aspirar al grado de Doctor en Ciencias en Ingeniería. Universidad Nacional Autónoma de México. 1996. 118 pgs. (p: 28)

[53] Lombardo John, *Embedded Linux*, Mew Riders Publishing, 2002. (p: 35)

[54] Marsden Alan, Antony Pople, (eds.) *Computer Representations and Models in Music* Academic Press, Great Yarmouth, Norfolk, 1992. 309 pgs. (p: 20)

[55] MCC Microcomputer Control Software Development Tools, *8051 C Cross Compiler MICRO/C-51: Programmer´s Guide Version 1*, Micro Computer Control Corporation, USA, 1993. (p: 76)

[56] MCC Microcomputer Control Software Development Tools, *8051 Macro Cross Assembler MICRO/ASM-51 Version 1*, Micro Computer Control Corporation, USA, 1993. (p: 76)

[57] Maxwell III Turner John, and Severo M. Ornstein, “Mockingbird: A composer's Amanuensis”, BYTE, January, 1984. pp. 384-401. (p: 28)

[58] Messick Paul, *Maximum MIDI: Music Applications in C++*, Manning Publications Co., Canada, 1997. 453 pgs. (p: 33)

[59] Miles Huber David, *Hard Disk Recording for Musicians*, Amasco Publications, USA, 1995. (p: 32)

[60] Minsky Marvin, “Music, Mind, and Meaning” (1981), en Schwanauer and Levitt, *Machine Models of Music*, USA, The MIT Press Cambridge, 1993., pp. 326-354. 544 pgs. (p: 18)

[61] Miranda T. R., M. Peña G., “Intérprete Musical Para Multimedia”, Simposium Internacional, Centro Nacional de Cálculo IPN, México, memorias, noviembre 1993. (p: 80)

[62] Miranda Eduardo Reck, *Composing Music with Computers*, Music technology series, Focal Press, Great Britain, London England, 2001. 238 pgs. (p: 15)

[63] Moog Robert A., “Voltaje-Controlled Electronic Music Modules”, R. A. Moog Co., Trumansburg, New York, Journal of the Audio Engeering Society, january, 1968, vol. 16 No. 1. pp. 200-206. (p: 25)

[64] Moog Bob., “MIDI: Musical Instrument Digital Interface”, Journal of the Audio Engeering Society, may 1986, v. 34 No. 5. pp. 394-404. (p: 25)

[65] Moore F. Richard, *Elements of Computer Music*, Prentice Hall, USA, 1990. (p: 19)

[66] Morgan Cristpher P. (ed.), *The Byte Book of Computer Music*, Byte Books, 1979. 144 pgs. (p: 24)

[67] Neumann John Von, *Theory of Self-Reproducing Automata*, Edited and completed by Arthur W. Burks, University of Illinois Press, Urbana and London, 1966. (p: 21)

[68] Peña G. M., “Mira”, en *Conjunto Marakatumba*, Dir.: M. Peña, M-026 45 rpm, Discos Marhnos, 1977. (p: 16)

[69] Peña G. M., “Mi Nena”, en *Delfines del Trópico*, Dir.: A. Pacheco, stereo L.P.BO-06, Discos Boga 1/3 rmp (Masterworks), siete composiciones, 1981. (p: 16)

[70] Peña G. M., M. González. H., “Producción Musical Para Multimedia”, Congreso Nacional ANIEI, Villa Hermosa Tabasco, memorias, octubre 1992. Simposium Nacional de Computación, Centro Nacional de Cálculo IPN, memorias, 1992. (p: 80)

[71] Peña G. M., “Un lenguaje para MIDI”, IV Semana de la Investigación Científica, Centro Nacional de Cálculo IPN, memorias, 1993. (p: 135)

[72] Peña G. M., “Tecnología MIDI”, Centro de Cálculo Arquitecto Joel Arriaga, Universidad Autónoma de Puebla, memorias, abril 1994. (p: 135)

[73] Peña G. M., “Análisis y Procesamiento Electrónico de la Música”, V Semana de la Investigación Científica, Centro Nacional de Cálculo IPN, memorias, abril 1994. (p: 136)

[74] Peña G. M., “SODM94: Diseño de un Sistema Operativo Simple”, Informe Técnico, serie azul, No.4, Centro Nacional de Cálculo, Centro de Investigación en Computación, IPN, 1994. (p: 35)

[75] Peña G. M., “BIOSMM: Sistema de Entrada Salida de Música para Multimedia”, DEPI 932191, Informe Técnico Serie azul, No. 5, Centro Nacional de Cálculo, Centro de Investigación en Computación, IPN, 1994. (p: 35)

[76] Peña G. M., “MCX: Sistema Operativo para una Máquina Paralela”, Reporte de Proyecto DEPI 951021, Centro Nacional de Cálculo, Centro de Investigación en Computación IPN, 1995. (p: 35)

[77] Peña G. M., "SEEAM: Sistema de Escritura y Ejecución Automática de Música", Reporte de Proyecto DEPI 962961, Centro Nacional de Cálculo, Centro de Investigación en Computación IPN, 1997. (p: 7)

[78] Peña G. M., "Algoritmos Para Simulación de una Orquesta Sinfónica en una Máquina Distribuida o Paralela", informe sabático, CIC-IPN, residencia sabática: CINVESTAV IPN, 1998-99. (p: 23)

[79] Peña G. M., "Algunas Formalidades de la Música", en I Simposium Internacional: Informática y Telecomunicaciones en la Educación y IV Simposium Nacional: La Educación Técnica en México, memorias, ESIQIE-IPN, Zacatenco D.F, septiembre 1999. pp. 141-146. (p: 23)

[80] Peña G. M., M. A. Ortiz M., R. Ocón V., E. Martínez R., "Sistema Operativo para DSP TMS320F/C240", Memorias, IEEE México RVP-AI/2002, Acapulco Gro., Mex., Julio, 2002. (p: 75)

[81] Peña G. M., A. De Lucca P., "Capture of Events MIDI in Parallel With FPGA", Journal of Applied Sciences & Technology, Vol. 1, No. 1, March 2003. (p: 54)

[82] Pérez-Prado D., "El Mambo del Politécnico", Partituras de la *Orquesta D. Pérez Prado*, 1976, Méx. (p: 80)

[83] Phillips, Dave, *Linux Music & Sound: How to Install, Configure, and use Linux Audio Software*, Linux Journal Press, USA, 2000. 408 pgs. (p: 33)

[84] Pohlmann Ken C., "Principles of Digital Audio", McGraw-Hill, 2000. (p: 32)

[85] Rader Gary M., "Creating Printed Music Automatically", Mesa State College, Grand Junction, Colorado, Computer IEEE, june 1996, pp. 61-68. (p: 21)

[86] Ramírez Rafael, "Inductive Logic Programming for Learning Musical Rules", Proceedings International Conference on Applied Artificial Intelligence, Dec. 2003. (p: 27)

[87] Real Academia Española, *Diccionario de la Lengua Española*, Espasa, Vigésima primera edición, España 2000. (p: 16)

[88] Rowe Robert, *Interactive Music Systems: Machine Listening and Composing*, The MIT Press Cambridge, Massachusetts London, England, 1993. (p: 28)

[89] Rumsey Francis, *MIDI Systems and Control*, Focal Press, 2nd. ed. 1994. (p: 27)

[90] Schwanauer Stephan M. and David A. Levitt, *Machine Models of Music*, USA, The MIT Press Cambridge, 1993. 544 pgs. (p: 17)

[91] Shin Kang G., "HARTS: A Distrubuted Real-Time Architecture" in Yann Hang Lee and C.M. Krishna, (eds.) *Readings in Real-Time Systems*, IEEE Computer Society Press, USA, pp. 30-49, 1993. 246 pgs. (p: 34)

[92] Selfridge-Field Eleanor, "DARMS, Its Dialects, and Its Uses", in Eleanor Selfridge-Field, ed., *Beyond MIDI: The Handbook of Musical Codes*, The MIT Press, USA, 1997, pp. 163-174. 630 pgs. (p: 21)

[93] Smith Gina, "Field Guide to CPUs", PC Computing, March 1993, p. 123. (p: 24)

[94] Smith Roderick W., “Tux, can you hear me?”, Linux Magazine, December 2004, p. 34. (p: 31)

[95] Süderberg Denis, “Extraction of Token Based VHDL models from Old ASIC Net Lists”, in Greg Peterson, Philip Wilsey (eds.) *Rapid Systemas Prototyping with VHDL*, Proceedings, VHDL International, IEEE, Computer Society, Fall 1997 Conference. pp. 157-161. 280 pgs. (p: 37)

[96] Stevens Al., “Band-In-A-Box, Finale, & MusicXML”, Dr Dobb´s Journal, September 2004, p. 52. (p: 34)

[97] Stevens Al., “MidiRecorder: An Exercise in Code Reuse”, Dr Dobb´s Journal, January 2005, p. 57. (p: 32)

[98] Stewart James W., Chao-Ying-Wang, *Digital Electronics Laboratory Experiments Using Xilinx XC95108 CPLD with Xilinx Foundation: Design and Simulation Software*, Prentice Hall, USA, 2001. 328 pgs. (p: 82)

[99] Suárez Guerra Sergio, R. Barrón F., M.E. Cruz M., “Drivers de Dispositivos Virtuales VxDs”, Reporte Técnico No. 17, serie verde, Centro de Investigación en Computación, IPN, México, mayo 1999. 48 pgs. (p: 35)

[100] Taupin Daniel, *MusicTEX Using TEX to write polyphonic or instrumental Music*, (version 5.14), Laboratoire de Phisique des Solides (associé au CNRS), batiment 510, Centre Universitaire, F-91405 ORSAY Cedex, Oct. 25, 1995. 67 pgs. (p: 22)

[101] Tranter Jeff, *Linux Multimedia Guide*, O’Reilly & Associates, Inc., USA 1996. 363 pgs. (p: 33)

[102] Turing A. M., “¿Puede pensar una máquina?”, en ed., James R. Newman, *SIGMA: El mundo de las matemáticas*, vol. VI, Grijalbo, décima edición, Barcelona Esp., 1985. p.45. (p: 21)

[103] Ulrich John Wade, “The Analysis and Synthesis of Jazz by Computer”, Computing and Information Science Department, Univeristy of New Mexico Alburquerque, New Mexico, Aplications-1: Ulrich, pp. 865-872. s.f. (p: 18)

[104] Winkler Todd, *Composing Interactive Music: Techniques and Ideas Using Max*, The MIT Press, USA, 1998. 350 pgs. (p: 22)

[105] Xilinx, *VHDL: Reference Guide*, Xilinx Inc., USA, 1999. (p: 83)

[106] Xilinx, *The Programmable Logic Data Book 1999*, Xilinx Inc., USA, 1999. (p: 83)

[107] Yuan Feng, *Windows Graphics Programming: Win32 GDI and DirectDraw*, p. cm. (Hewlett-Packard profesional books), Prentice Hall, USA 2001. 1234 pgs. (p: 80)

Φ

Glosario

ASIC: *Application Specific Integrated Circuit*, circuito integrado de aplicación específica. Circuitos integrados electrónicos que pueden ser diseñados en forma directa por los ingenieros de aplicaciones.

CAC: *Computer Aided Composer*, Composición Asistida con Computadora. Colección de programas de *software* agrupados en un medio ambiente de computadora digital que interactúa con el usuario para estudiar, desarrollar, simular e interpretar música de cualquier tipo, incluyendo toda clase de sonidos.

CPLD: *Complex Programmable Logic Device*, dispositivo de lógica programable compleja. Circuito integrado de alta densidad que puede ser programado de tal forma que se permita introducir sistemas de circuitos lógicos electrónicos dentro de un sólo *chip*, y con lo cual es posible producir circuitos integrados de propósito específico (ASIC).

FPGA: *Field Programmable Gate Array*, arreglo de compuertas programables en campo. Circuito integrado de alta densidad que contiene un arreglo de compuertas lógicas cuyas conexiones son programables de tal forma que sea posible introducir sistemas de circuitos lógicos electrónicos en un sólo *chip* para implementar un circuito integrado de propósito específico (ASIC).

Generador de tonos: Dispositivo electrónico digital que puede generar una gran variedad de sonidos musicales de acuerdo con las notas que recibe. Es un sintetizador de sonidos similares a los producidos por instrumentos musicales. Se conecta, a través de la interfaz MIDI, a computadoras digitales o cualquier otro controlador MIDI, los cuales envían notas musicales en forma de comandos para reproducir la música de acuerdo con la información que tiene un archivo MIDI, u otra forma de almacenamiento digital.

ICB: *Isolate Couple Block*, bloque de acoplamiento aislado. Interfaz de hardware diseñada para el sistema MCS que contiene 16 circuitos de aislamiento optoelectrónico entre los dispositivos de comunicaciones UART y los instrumentos MIDI: optoacopladores, inversores, amplificadores y componentes pasivos.

KL: *Kernel for a musical Language*, núcleo para un lenguaje musical. Compilador de estructuras musicales desarrollado por el autor para estudiar el procesamiento electrónico de la música. La entrada es un programa fuente que tiene cadenas de texto escritas en lenguaje natural (`do re mi`$\flat$ `fa sol sil fa`$\sharp$`...`) de las notas musicales (solfeo). La salida son gráficas, partituras, y/o el sonido de una canción escrita en el programa fuente.

KLM: *Kernel for a smalL Monitor*, núcleo para un monitor pequeño. Conjunto de primitivas y funciones que configuran un sistema operativo básico en tiempo real, desarrollado por el autor, para controlar los recursos de *hardware* y *software* de una microcomputadora con base en microprocesadores y microcontroladores. Supervisa las peticiones de usuario a través de una terminal de computadora y responde a las señales eléctricas de control que emiten los periféricos activando las rutinas de atención asociadas.

MCB: *MIDI Communications Block*, bloque de comunicaciones para interfaz MIDI. Interfaz de hardware diseñada para el sistema MCS, con 16 componentes digitales de recepción de datos UART y de propósito específico ASIC, descritos formalmente con VHDL, para conectar la grabadora digital MCS con instrumentos musicales.

MCS: *MIDI Capture System*, sistema de captura MIDI. Nombre genérico del dispositivo de grabación electrónico digital que se presenta en este trabajo. Diseñado para capturar datos MIDI que transmiten varios instrumentos musicales cuando están tocando, y obtener la partitura de cada uno.

MCS-L: *MIDI Capture System-Lineal*, sistema de captura MIDI versión 1. Sistema MCS que tiene una memoria estática RAM lineal; versión académica.

MCS-S: *MIDI Capture System-Segmented*, sistema de captura MIDI versión 2. Sistema MCS que tiene memoria con segmentos variables; versión industrial que puede fabricarse en grandes cantidades con tecnología VLSI.

MIDI: *Musical Instrument Digital Interface*, interfaz digital para instrumentos musicales. Protocolo de comunicaciones de *hardware* y *software* estándar diseñado por los fabricantes de instrumentos musicales electrónicos.

Pentagrama: Grupo de cinco líneas horizontales para escribir símbolos gráficos que tiene el lenguaje musical y que determina una melodía.

Sintetizador: Dispositivo electrónico digital, similar a un piano, que puede generar una gran variedad de sonidos musicales. Las notas son activadas por el sistema de teclas, o también por aquellas que recibe a través de su interfaz MIDI interconstruida y conectada con computadoras digitales u otros dispositivos similares.

Sistema de tiempo real: Dispositivo que controla una aplicación dedicada la cual tiene restricciones temporales bien definidas.

Solfeo: Lenguaje natural musical, compuesto por las sílabas *do*, *re*, *mi*, fa, *sol*, *la* y *si*, para enseñar música cantando.

Tiempo real: Momento actual de respuesta al procesamiento inmediato de datos que interactúan con procesos físicos.

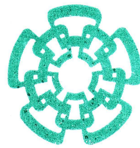
UART: *Unasynchronous Asynchronous Reciver Transmitter*, transmisor y receptor síncrono asíncrono. Dispositivo electrónico digital de comunicaciones serial que transforma datos con formato serie a paralelo para recibir o transmitir información. La velocidad de transferencia, cuando se utiliza en instrumentos musicales electrónicos, es de 32,250 Bauds lo que permite enviar y recibir, en promedio, hasta 142 notas musicales en un segundo.

VHDL: *VHSIC Hardware Description Language* (lenguaje descriptivo de *hardware* para VHSIC). Lenguaje de alto nivel para diseñar circuitos digitales electrónicos.

VHSIC: *Very High Speed Integrated Circuits* (circuitos integrados a velocidad muy elevada).

VLSI: *Very Large Scale Integration* (integración a escala muy elevada). Técnica de fabricación de circuitos integrados electrónicos en espacios bastantes reducidos.

Φ



CENTRO DE INVESTIGACION Y DE ESTUDIOS AVANZADOS DEL I.P.N.

Los abajo firmantes, integrantes del jurado para el examen de grado que sustentará el **M. en C. Maximino Peña Guerrero**, declaramos que hemos revisado la tesis titulada:

"Captura de Múltiples Eventos MIDI en Tiempo de Ejecución"

Y consideramos que cumple con los requisitos para obtener el grado de **Doctor en Ciencias**, en la especialidad de Ingeniería Eléctrica opción Computación.

ATENTAMENTE:

Dra. Graciela Roman Alonso __________

Dr. Miguel Ángel León Chávez __________

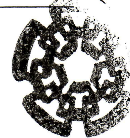
Dr. José Oscar Olmedo Aguirre __________

Dr. Sergio Victor Chapa Vergara __________

Dr. Adriano De Luca Pennacchia __________

Dr. Jorge Buenabad Chávez __________

CINVESTAV - IPN
ING. ELECTRICA
ENE. 4 2005
COMPUTACION



CINVESTAV - I.P.N.
DEPTO DE ING. ELECTRICA
COORDINACION ACADEMICA
SECCION DE COMPUTACION

Av. Instituto Politécnico Nacional 2508 Col. San Pedro Zacatenco México, D.F. C.P. 07360
Tels: 747-70-00 y 747-70-01 Fax: 747-70-03

TGM/1996