PROYECTO FINAL DE CARRERA

MODELOS VHDL SINTETIZABLES DE LOS MICROPROCESADORES 8088/8086 DE INTEL

ALUMNO: Rubén Martín Torres
ESPECIALIDAD: Electrónica Industrial
DIRECTORES: José Ignacio García Nicolás y Tomás Pollán Santamaría

CONVOCATORIA: Septiembre del 2004

si no ansías la Luna, nunca
sabrás lo que podías haber
conseguido... ...para PPA

Índice

A.enunciado y características

1.- Enunciado..... 1
2.- Análisis del Enunciado..... 1
3.- Características del Modelo Desarrollado..... 2

B. características de los micros 8088/8086

4.- Breve Historia del 8088/8086..... 3
5.- Funcionamiento General..... 4
5.1.- Organización de la Memoria..... 4
5.2.- Modos Máximo y Mínimo..... 6
5.3.- Funcionamiento del Bus..... 7
5.4.- Direccionamiento de las Entradas/Salidas..... 8
5.5. Comunicación con el Exterior..... 8
5.5.1 Inicialización y Reseteado del Microprocesador..... 8
5.5.2.- Operaciones de Interrupción..... 8
5.5.3.- Interrupciones no Enmascarables (NMI)..... 9
5.5.4.- Interrupciones Enmascarables (INTR)..... 9
5.6.- HALT..... 9
5.6.1.- Operaciones Utilizando LOCK (Semáforo)..... 10
5.7.- Sincronización Externa Utilizando la Señal TESTn..... 10
5.8.- Temporización Básica del Sistema..... 10
5.8.1.- Temporización del Sistema Mínimo..... 10
5.8.2.- Temporización del Sistema Máximo..... 11
5.9.- El 8088 Comparado con el 8086..... 11

C. estructura del modelo

6.- Diagrama de Bloques General..... 13
7.- Descripción de los Modelos..... 14
7.1.- Diagrama Jerárquico de Bloques..... 14
7.1.1.- Arquitectura Interna Básica..... 14
7.1.1.1.- Nivel 0 de la Jerarquía..... 16
7.1.1.2.- Nivel 1 de la Jerarquía..... 17
7.1.1.3.- Nivel 2 de la Jerarquía..... 17
7.1.1.4.- Nivel 3 de la Jerarquía..... 18
7.1.1.5.- Nivel 4 de la Jerarquía..... 18
7.2.- Subtipos de Datos Definidos en el Package..... 18
7.3.- NIVEL 0..... 19
7.4.- Descripción de los Pines..... 20
7.4.1.- Terminales Comunes..... 20
7.4.2.- Terminales en Modo Mínimo..... 22
7.4.3- Terminales en Modo Máximo..... 23
7.5.- NIVEL 1..... 24
7.5.1.- Diferencias Entre up8088mn-up8088mx..... 24
7.5.2.- Diferencias enltre up8086mn-up8086mx..... 24
7.5.3.- Diferencias entre up8088mn/up8088mx y up8086mn/up8086mx..... 24

D. componentes de ámbito común

8.- Explicación General de los Grafos de Estado..... 27
8.1.- Estados..... 27
8.2.- Transiciones..... 27
8.3.- General..... 27
8.4.- Código VHDL..... 28
8.4.1.- Definiciones..... 28
8.4.2.- Cuerpo de la Arquitectura..... 28
9.- Unidad de Reset e Inicialización (n1/r3)..... 29
9.1.- Entradas y Salidas Internas..... 29
9.1.1.- Algunas Cuestiones Sobre el Reloj..... 29
9.1.2.- Verticalidad, Simultaneidad, Fuerza y Pureza de CK..... 29
9.2.- Explicación del Grafo de Estados..... 30
9.2.1.- Comportamiento..... 30
9.2.2.- Estados..... 30
9.2.1.- Caminos..... 30
10.- Registros de uso General (n1/r1)..... 31
10.1.- Descripción del Fichero..... 31
10.2.- Entradas y Salidas..... 31
10.3.- Procesos..... 32
11.- Registros de Segmento (n1/r2)..... 33
11.1.- Entradas y Salidas..... 33
11.2.- Funcionamiento..... 33
11.3.- Procesos..... 34
12.- Puertos (n1/r6)..... 35
12.1.- Entradas y Salidas..... 36
12.2.- Tipos de Puertos..... 36
12.2.1.- Puerto de Salida..... 37
12.2.2.- Puerto de Salida Triestado..... 37
12.2.3.- Puerto de Entrada-Salida Triestado..... 37
12.2.4.- Puerto de Entrada-Salida con Resistencia Pull-UP..... 37
12.2.5.- Puerto de Entrada..... 38
12.2.6.- Puerto de Entrada NMI..... 38
12.2.7.- Declaración de Tipos Enumerados..... 38
13.- Cola de Instrucciones..... 39
13.1.- Entradas y Salidas..... 39
13.2.- Cola de Instrucciones en el 8088..... 39
13.3.- Procesoso..... 40
13.3.- Cola de Instrucciones en el 8086..... 41

E. unidad de ejecución, EU

14.- Unidad de Ejecución, EU (n1/r4) 42
14.1.- Entradas y Salidas 43
15.- Unidad Lógico Aritmética, ALU (n2/r6) 44
15.1.- Entradas y Salidas 44
15.2.- Operaciones 44
15.2.1.- Operaciones Cotextuales 44
15.2.2.- Operaciones Artméticas que Utilizan el Proceso Sumador/Restador 44
15.2.3.- Operaciones Aritméticas y Lógicas 44
15.3 Procesos 45
16.- Unidad EXE (n2/r2) 46
16.2.- Explicación del Grafo de Estados 46
16.3.- Entradas y Salidas 46
16.4.- Comportamiento 47
16.5.- Estados 47
17.- Contador de Programa, PC (n2/r3) 49
17.1.- Funcionamiento 49
17.2.- Entradas y Salidas 49
17.3.- Procesos 49
18 Registros Especiales, SPW (n2/r4) 50
18.1 Entradas y Salidas 50
18.2.- Funcionamiento 50
18.2.1 Registros de Comunicación con la BIU 50
18.2.2.- Registro de Banderas 50
18.3.- Procesos 51
19.- ROM de Microinstrucciones, CGROM (n2/r1) 52
19.1.- Procesos 52
19.1.1.- Proceso Code_Generatos_ROM 52
19.1.2.- Proceso Demux_OP 52
19.2.- Entradas y Salidas 53
19.3.- Ejemplos 54
19.3.4.- MOC Inmediato a Registro 54
19.3.5.- INC/DEC Registro 54
20.- Decodificador, DECO (n2/r5) 55
20.1.- Explicación del Grafo de Estados 55
20.1.1.- Prefijos 56
20.1.1.1.- Prefijo de Segmento, SS:,ES:,DS: y SS: 56
20.1.1.2.- Prefijo de Bloqueo LOCK: 56
20.2.- Entradas y Salidas 56
20.3.- Estados 57
20.4.- Caminos 57

F. unidad de interface con el Bus, BIU

21.- Unidad de Interface con el Bus, BIU (n1/r7) 58
21.1.- Entradas y Salidas 59
22.- Temporización del Bus Externo, TIMING (n2/r8) 60
22.1.- Entradas y Salidas 61
23.- Máquina de Estados del Bus, TSM (n3/r8) 62
23.1.- Explicación 62
23.1.1.- Fase de Direcciones/Estado 63
23.1.2.- Fase de Datos 63
23.2. Estados 63
23.2.1 Estados de Espera, Tw 63
23.2.2.-Estados Ociosos, Ti 64
23.3.- Caminos 64
24.- Unidad Generadora de Ciclos de Bus, BCG (n3/r9) 64
24.1.- Entradas y Salidas 64
24.2.- Explicación 65
24.2.1.- Ciclos de Bus 65
24.2.1.1.- Ciclos de Bus de Lectura 65
24.2.1.2.- Ciclos de Bus de Escritura 65
24.2.1.3.- Ciclos de Bus de Reconocimiento de Interrupción 65
24.2.1.4.- Ciclos de Bus HALT 66
24.3.- Procesos 66
25.- Unidad de Interface con el Bus, INTERF (n2/r7) 67
25.1.- Entradas y Salidas 68
26.- Árbitro de Canal, ARBITER (n3/r3) 69
26.1.- Entradas y Salidas 69
26.2.- Funcionamiento 70
26.2.1.- Máquina de Estados 1, Arbiter_Logic 70
26.2.2.- Máquina de Estados 2, ifu_dfu_Cycle 71
27.- Unidad de Multibus, MU (n3/r6) 72
27.1.- Entradas y Salidas 72
27.2.- Explicación 72
27.2.1.- Secuencia de Petición-Cesión (Request-Grant) del Bus Local 73
27.2.2.- Situaciones Especiales 74
27.2.2.1.- Aparece una Segunda Petición Cuando ya Está el Bus Cedido 74
27.2.2.2.- Aparecen Dos Peticiones a la Vez 74
28.- Unidad Generadora de Direcciones, GAU (n3/r7) 75
28.1.- Entradas y Salidas 75
29.- Sumados de la BIU, ADDER (n4/r6) 76
30.- Grafo de Control de la GAU, GAU_CTRL (n4/r7) 77
30.1.- Entradas y Salidas 77
30.2. Explicación 78
30.2.1.- Comunicación con la Unidad IFU 78
30.2.2.- Comunicación con la Unidad DFU 78
31.- Unidad de Acceso a Datos, DFU (n3/r4) 79
31.1.- Entradas y Salidas 80
32.- Registros de DFU, DFU_REG (n4/r1) 81
32.1.- Registros de Comunicación Interna 82
32.1.1.- Registros de Comunicación en el 8088 82
32.1.2.- Regisros de Comunicación en el 8086 82
33. Grafo de Control de DFU, DFU_CTRL (n4/r2) 83
33.1.- Entradas y Salidas 84
33.2.- Funcionamiento 84
33.2.1.- Bus de Datos 84

33.2.1.1.- Bus de 16 Bits..... 84
33.2.1.2.- Bus de 8 Bits..... 85
33.2.2.- interfaces con la Memoria de Entrada/Salida..... 85
33.2.4.- Lógica del Grafo de Estados..... 86
34.- Unidad de Acceso a Instrucciones, IFU (n3/r5)..... 87
34.1.- Entradas y Salidas..... 88
35.- Puntero de la IFU, IFU_POINTER (n4/r5) 88
35.1.- Actualización de ifu_pointer en el 8088..... 88
35.2.- Actualización de ifu_pointer en el 8086..... 89
35.2.1.- Cambio en el Puntero de Instrucciones IP..... 99
35.2.2.- Alineamiento del Puntero ifu_pointer..... 90
35.3.- Registro de pa_instr..... 90
35.1.- Delimitación del Puntero ifu_pointer..... 90
36.- Unidad de Sincronización del Flush, FLUSH (n4/r3) 91
36.1.- Entradas y Salidas..... 91
36.2.- Eatados..... 91
37.- Unidad de Control de la IFU, IFU_CTRL (n4/r4)..... 92
37.1.- Entradas y Salidas..... 92
37.2.- Explicación..... 93
37.2.1.- Lógica del Grafo..... 93

G. modelo software de los microprocesadores

38.- Modelo Software de los Microprocesadores..... 94
38.1.- Registros de Propósito General..... 94
38.1.1.- Registros Base e Índices..... 94
38.1.2.- Registro de Banderas..... 94
38.1.3.- Registros de Segmento..... 95
38.2.- Registros de Segmento y Desplazamiento Implícitos..... 95
38.3.- Memoria Segmentada..... 96
38.4.- Relocalización Dinámica del Código..... 96
38.5.- Tipos de Datos Soportados Directamente por los Micros..... 97
38.6.- Modos de direccionamiento de las CPU's 8086/8088..... 97
38.6.1.- Algunos Ejemplos..... 99

H. simulación

39.- Simulación..... 100
40.- Modelado del Entorno..... 101
40.1.- Bancos de Test..... 101
40.2.- Bancos de Test Para la Simulación. Esquemas. 102
41.- Generador de Vectores de Test, SYSTEM (n0/r2) 103
41.1.- Generador de Reloj..... 104
41.2.- Generador de RESET..... 104
41.3.- Generador de Estados de Espera..... 105
41.4.- Desacoplo de los Buses..... 105
41.4.1.- Desacoplo de los Buses del 8088..... 105
41.4.2.- Desacoplo de los Buses del 8086..... 106
41.5.- Mapa de Memoria..... 107
41.6.- Generador de a señal TESTn..... 107
41.7.- Generador de Interrupciones..... 108
41.7.1.- Generador de Interrupciones NMI..... 108
41.7.2.- Generador de Interrupciones INTR..... 110
41.8.- Controlador de Bus..... 110
41.9.- Generador de Peticiones de Cesión de Bus..... 111
41.9.1.- Cesión de Bus en Modo Mínimo, HOLD/HOLDA..... 111
41.9.2.- Cesión de Bus en Modo Máximo, REQUEST/GRANT..... 111
41.10.- Periféricos de I/O del Sistema de Simulación..... 113
42.- Modelado de la Memoria..... 114
42.1.- Modelo de Memoria del 8088..... 114
42.2.- Modelo de Memoria del 8086..... 115
42.3.- Cargador de memoria..... 115
42.3.1.- Formato..... 115
42.4.- Fichero de Salida..... 116
42.4.1.- Formato del Fichero LOG en el 8088..... 116
42.4.1.- Formato del Fichero LOG en el 8086..... 116
43.- Modelo de un Sistema Operativo Mínimo..... 117
44.- Programación en Ensamblador..... 119
44.1.- EJEMPLO 1: Conversión de Binario a ASCII..... 119
44.1.2.- Código del Programa en Ensamblador..... 119
44.1.3.- Compilación del Fichero ASM..... 121
44.1.3.1.- Compilación a Mano..... 121
44.1.3.2.- Compilación Automatizada..... 121
44.1.3.3.- Fichero de Listado LST..... 122
44.1.4.- Generación del Archivo de la Simulación "download_filename"..... 125
44.1.4.1.- Localización del Programa. Definición de los Segmentos..... 125
44.1.4.2.- Cambio de los Octetos..... 126
44.1.4.3.- Eliminación de la Líneas no Útiles..... 126
44.1.5.- Lanzamiento de la Simulación..... 129
44.1.5.1.- Pasos Para Lanzar la Simulación..... 129
44.1.5.2.- Lanzamiento de la Simulación de Forma Automatizada..... 130
44.1.6.- Resultados de la Simulación..... 132
44.1.7.- Análisis de la Simulación..... 137
44.1.8.- Fichero Histórico LOG..... 137
44.2.- EJEMPLO 2: Conversión de Binario a 7 Segmentos..... 138
44.2.1.- Fichero ASM..... 138
44.2.2.- Compilación del Fichero ASM..... 138
44.2.3.- Generación del Archivo de Simulación..... 138
44.2.3.1.- Definición de los Segmentos..... 138
44.2.3.2.- Cambio de los Octetos..... 138
44.2.3.3.- Eliminación de Comentarios..... 138
44.2.3.4.- Fichero "prog2.hex" Listo para la Simulación..... 139
44.2.4.- Lanzamiento de la Simulación..... 140
44.2.5.- Análisis de la Simulación..... 143
44.2.6.- Fichero Histórico LOG..... 143
45.- Programación en Turbo C, C++ EJEMPLO 3: Multiplicación de Matrices..... 144
45.1.- Fichero Matriz.c..... 144

45.2.- Generación del Archivo de Simulación..... 145
45.2.1.- Compilación y Linkado del Programa..... 145
45.2.2.- Generación del Archivo Matriz.asm a Partir del Fichero Matriz.c..... 145
45.2.3.- Fichero Matriz.hex Listo Para la Simulación..... 147
45.3.- Lanzamiento de la Simulación..... 149
45.4.- Análisis de la Simulación..... 149
45.5.- Resultados se la Simulación..... 150

I. síntesis (*)

46.1.- Resultados de la Síntesis..... -
46.2.-Programa Utilizado..... -
46.3.- Mejora del Reloj..... -
46.3.1.- Aumento del Número de Fases..... -
46.3.2.- Cambio del Sumador Serie por una Lógica de Acarreo Adelantado..... -
46.4.- Comparativa..... -

APÉNDICE A. juego de instrucciones

A1.- Códigos OP del Juego de Instrucciones de los Micros 8088/8086..... 150
A2.- Cambios de Bit en el Registro de Banderas..... 155
A3.- Juego de Instrucciones Completo..... 155
A3.1.- Instrucciones de Transferencia de Datos..... 156
A3.2.- Cálculo de la Dirección Efectiva..... 159
A3.3.- Instrucciones Aritméticas..... 160
A3.4.- Instrucciones Lógicas..... 166
A3.5.- Instrucciones con Cadenas de Datos..... 171
A3.6.- Instrucciones de Control de Fuljo..... 175
A3.7.- Instrucciones de Control del Microprocesador..... 178

APÉNDICE B. programa debug

B.- Utilización de DEBUG..... 179
B1.- Ejemplo 1..... 180
B2.- Ejemplo 2..... 180

J. cuestiones generales

C.- Página Web del Proyecto..... 181
C1.1.- Contratación de una Licencia GNU..... 181
C1.2.- Secciones del Menú..... 182
C1.3.- Exploración de la Librerías..... 183
C2.- Licencia de Documentación Libre GNU..... 184
D.- Contenido del CD-ROM del Proyecto..... 188

E.- Referencias Bibliográficas..... 188
E1.- Libros Utilizados..... 188
E2.- Documentación en Papel..... 188
E3.- Documentos en PDF Consultados..... 188
E4.- Páginas de Internet Visitadas..... 189
E5.- Programas Utilizados..... 189

* Los datos de la Síntesis, los daré a conocer durante la presentación del proyecto.

enunciado y características

1. ENUNCIADO

Modelos VHDL Sintetizables de los microprocesadores de 16-Bits de Intel® 8088/8086 en los modos Máximo y Mínimo

2. ANÁLISIS DEL ENUNCIADO

Este proyecto presenta el diseño a nivel estructural (Regisdter Transfer Level, RTL) de los microprocesadores "padre" de la familia más potente de Intel®, la familia 8088/8086 que evolucionaría a 80186, 80286...80586 o Pentium.

El medio utilizado para el diseño ha sido el lenguaje de descripción circuital VHDL (encuadrado dentro de Hardware Description Logic, HDL) estándar dentro del IEEE (Institute of Electrical and Electronic Engineers) que aumenta la portabilidad y legibilidad del diseño en tanto en cuanto:

- Permite definir la jerarquía del microprocesador descomponiéndola en subdiseños y definir la forma en la que estos subloques se interconectan.
- La definición de los componentes se realiza como si de un lenguaje de programación se tratara.
- Nos permite la simulación del diseño sin tenerlo físicamente implementado lo que permite ahorrar costos en la fabricación de prototipos.
- Implementar cambios y corregir errores de una forma rápida hace que el diseño de circuitos integrados requiera poco tiempo de diseño en comparación con la técnica clásica de esquema-prototipo-simulación.
- El diseño deja de depender de la tecnología siendo el sintetizador el que se encarga de embeber el diseño en el dispositivo indicado pudiendo establecer "síntesis a medida" según requerimientos de velocidad, área...

El 8088/8086 es un micro potente con una estructura interna de 16 bits y un bus externo de 8 bits (en el 8088) o de 16 bits (en el 8086) capaz de acceder a 1 Mbyte de memoria (bus de direcciones de 20 bits). El modelo diseñado se comporta como estos micros en los modos Máximo y Mínimo cumpliendo escrupulosamente con los tiempos de bus y mejorando en el 99% de las instrucciones el tiempo de ejecución.

La potencia de estos micros junto con el potencial de su juego de instrucciones y el software disponible para programarlos (podemos utilizar cualquier programa diseñado para generar código Intel hexadecimal: C, Ensamblador, Basic..), hacen que este modelo sea válido en aplicaciones dónde se requiera más potencia que la suministrada por un clásico microcontrolador de 8 bits (como el 8051, o un PIC); aplicaciones como móviles de última generación, PDA's, controladores gráficos...

Si el modelo no va a ser implementado en un sistema diseñado originariamente para los micros i8088/i8086 (ej. Sustituyendo físicamente un micro de uno de los antiguos ordenadores IBM) que será lo normal, no es necesario utilizar sólo 40 pines de la FPGA, sino que podemos utilizar muchos más terminales de salida evitando de esta forma la multiplexación en el tiempo (bus de direcciones-datos, bus de direcciones-sistema...) y la multiplexación en el modo de funcionamiento (HOLD en modo mínimo se convierte en RQn/GT0n en modo máximo). Con esto conseguimos una eficacia mucho mayor, manteniendo el potencial del juego de instrucciones pero reduciendo la lógica combinacional de la multiplexación de la señales así como el tiempo de acceso a estas.

3. CARACTERÍSTICAS DEL MODELO DESARROLLADO

A continuación podemos ver algunas de las características principales los modelos VHDL desarrollados:

- El diseño del modelo es completamente síncrono.
- La estructura interna elimina los buses triestado que tiene el microprocesador original pasando de tener una estructura interna Harvard a una estructura Von Newman que se caracteriza, entre otras cosas, por comunicarse con la unidad de procesamiento lógico aritmético a través de tres buses (como se puede ver en la figura). Las consecuencias inmediatas del cambio de estructura interna son el aumento de la velocidad en muchas instrucciones y la estabilidad obtenida al eliminar los buses internos triestado.

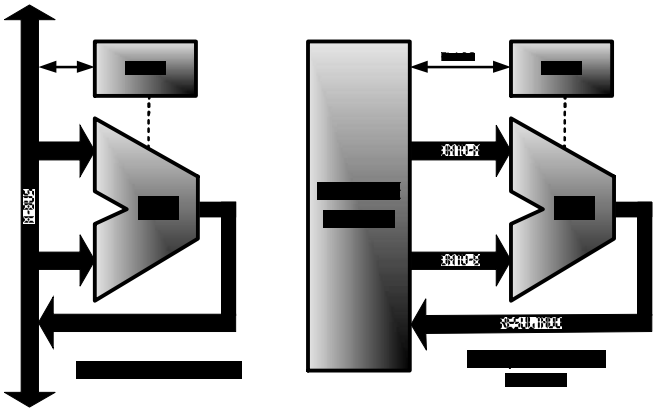


Fig 1.- Arquitecturas Internas

- El microprocesador gana ciclos de reloj en la ejecución del 98% de las instrucciones y en algunos casos ejecuta instrucciones un 200% más rápido como es el caso de las instrucciones de multiplicación y división con y sin signo. Esto junto con la mejora del la frecuencia de reloj (las versiones originales fueron diseñadas para frecuencias de 5 y 8 MHz, mientras que el modelo VHDL admite una frecuencia de unos 10 MHz, ver resultados de la síntesis) hacen que el modelo supere con creces al micro original e incluso a alguna de sus evoluciones. Ver la siguiente gráfica.
- El modelo soporta el 100% de las instrucciones del microprocesador original así como toda la lógica de interrupción y cesión del bus local (soporta Multibus ®).
- Es parametrizable, podemos cambiar la configuración interna del microprocesador modificando unas constantes globales del package del modelo. Podemos indicar la profundidad de la cola de instrucciones "queue_size" y podemos indicar el segmento de inicio "segment_init" entre otras cosas.

queue_size	Incremento en t (base 800us)
1	+200 us
2	+68 us
3	+19 us
4	0 us
5	-10 us
7	-18 us
11	-25 us

Tab 1.- Tiempo de ejecución en función de queue_size

- Es un modelo sintetizable. Los datos de la síntesis se encuentran en el capítulo de Síntesis.

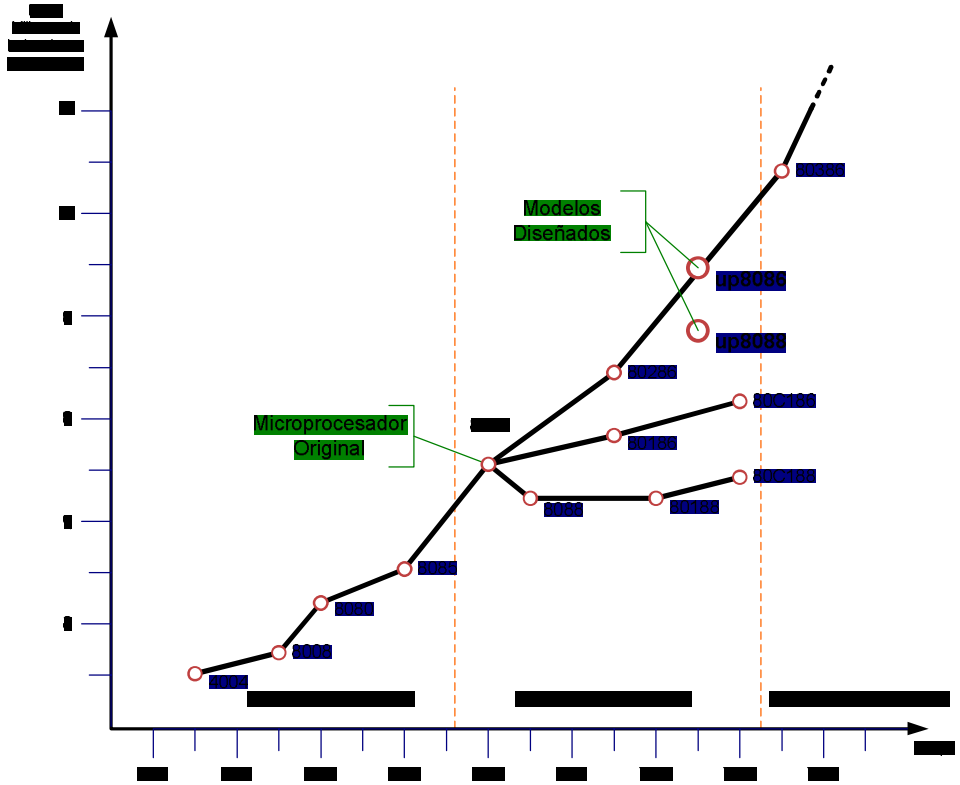


Fig 2.- Comparativa de velocidad entre la evolución de Intel ® y los modelos desarrollados.

características de los microprocesadores 8088/8086

4. BREVE HISTORIA DEL 8086/8088

Antes de empezar a desarrollar el proyecto, creo conveniente una breve introducción histórica al microprocesador 8086/8088 y al entorno que hizo que este micro saltara a la fama. La estrategia empresarial de Intel tanto en aspectos de dirección como de diseño en hardware o software, hizo que este micro se alzara a la cabeza del mundo informático, sobretodo el doméstico, casi 30 años de supremacía. La elección de modelar este micro en VHDL, por lo tanto es obvia; tenemos muchos años de hardware y programas que pueden correr en este micro.

En junio de 1978 Intel lanzó al mercado el primer microprocesador de 16 bits: el 8086. En junio de 1979 apareció el 8088 (internamente igual que el 8086 pero con bus de datos de 8 bits) y en 1980 los coprocesadores 8087 (matemático) y 8089 (de entrada y salida). El primer fabricante que desarrolló software y hardware para estos chips fue la propia Intel. Reconociendo la necesidad de dar soporte a estos circuitos integrados, la empresa invirtió gran cantidad de dinero en un gran y moderno edificio en Santa Clara, California, dedicado al diseño, fabricación y venta de sus sistemas de desarrollo que, como se explicó anteriormente, son computadoras autosuficientes con el hardware y software necesario para desarrollar software de microprocesadores.

Los sistemas de desarrollo son factores clave para asegurar las ventas de una empresa fabricantes de chips. La inmensa mayoría de ventas son a otras empresas, las cuales usan estos chips en aparatos electrónicos, diseñados, fabricados y comercializados por ellas mismas. A estas empresas se las llama" fabricantes de equipo original", o en inglés, OEM (Original Equipment Manufacturer). El disminuir el tiempo de desarrollo de hardware y software para las OEM es esencial, ya que el mercado de estos productos es muy competitivo. Necesitan soporte pues los meses que les puede llevar el desarrollo de las herramientas apropiadas les puede significar pérdidas por millones de dólares. Además quieren ser los primeros fabricantes en el mercado, con lo cual pueden asegurarse las ventas en dos áreas importantes: a corto plazo, ya que al principio la demanda es mucho mayor que la oferta, y a largo plazo, ya que el primer producto marca a menudo los estándares.

De esta manera la empresa Intel había desarrollado una serie completa de software que se ejecutaba en una microcomputadora basada en el 8085 llamada "Intellec Microcomputer Development System". Los programas incluían ensambladores cruzados (éstos son programas que se ejecutan en un microprocesador y generan código de máquina que se ejecuta en otro), compiladores de PL/M, Fortran y Pascal y varios programas de ayuda. Además había un programa traductor llamado CON V86 que convertía código fuente 8080/8085 a código fuente 8086/8088. Si se observan de cerca ambos conjuntos de instrucciones, queda claro que la transformación es sencilla si los registros se traducen así: A -> AL, B -> CH, C -> CL, D -> DH, E -> DL, H -> BH y L -> BL. Puede parecer complicado traducir LDAX B (por ejemplo) ya que el 8088 no puede utilizar el registro CX para direccionamiento indirecto, sin embargo, se puede hacer con la siguiente secuencia: MOV SI, CX; MOV AL, [SI]. Esto aprovecha el hecho que no se utiliza el registro SI. Por supuesto el programa resultante es más largo (en cantidad de bytes) y a veces más lento de correr que en su antecesor 8085. Este programa de conversión sólo servía para no tener que volver a escribir los programas en una primera etapa. Luego debería reescribirse el código fuente en assembler para poder obtener las ventajas de velocidad ofrecidas por el 8088. Luego debía correr el programa en la iSBC 86/12 Single Board Computer basado en el 8086. Debido al engorro que resultaba tener dos plaquetas diferentes, la empresa Godbout Electronics (también de California) desarrolló una placa donde estaban el 8085 y el 8088, donde se utilizaba un ensamblador cruzado provisto por la compañía Microsoft. Bajo control de software, podían conmutarse los microprocesadores. El sistema operativo utilizado era el CP/M (de Digital Research).

El desarrollo más notable para la familia 8086/8088 fue la elección de la CPU 8088 por parte de IBM (International Business Machines) cuando en 1981 entró en el campo de las computadoras personales. Esta computadora se desarrolló bajo un proyecto con el nombre "Acorn" (Proyecto "Bellota") pero se vendió bajo un nombre menos imaginativo, pero más correcto: "Computadora Personal IBM", con un precio inicial entre 1260 dólares y 3830 dólares según la configuración (con 48KB de memoria RAM y una unidad de discos flexibles con capacidad de 160KB costaba 2235 dólares). Esta computadora entró en competencia directa con las ofrecidas por Apple (basado en el 6502 de Notorola) y por Radio Shack (basado en el Z-80 de Zilog).

Fig 3.- Flamante IBM 5150 Original.

El primer IBM PC apareció en el mercado en 1981. Las dos disqueteras que aparecen en la imagen eran opcionales y costaban 400€., por defecto incluía una unidad de cinta para almacenamiento de datos. Internamente disponía de 5 slots de expansión tipo ISA (Industry Standard Architecture, 8 bits). El microprocesador de esta máquina era el 8088 funcionando en modo máximo y el coprocesador matemático, el 8087 costaba unos 600€. No tenía disco duro y su memoria RAM por defecto era de 256 Kbytes.

5. MICROPROCESADOR ORIGINAL

Los microprocesadores originales están fabricados con una tecnología HMOS-II encapsulados en una carcasa de 40 pins Dual In Line cerámica, CERDIP. El microprocesador 8088 es compatible con el software desarrollado para el microprocesador 8086 y con el hardware diseñado para los micros 8080/8085 (Microprocesadores de 8 bits de Intel ®). La necesidad de compatibilidad con el hardware diseñado para la familia de micros 8080/8085 hizo a Intel lanzar el 8088 (año 1979) cuando el 8086 llevaba un año en el mercado. El 8088 difiere en pocos aspectos del 8086, la diferencia más importante es que el bus de datos externo es de 8 bits en lugar 16 bits (en el 8086) ver diferencias entre el 8088 y el 8086.

FUNCIONAMIENTO GENERAL

Los microprocesadores 8088/8086 están divididos internamente en dos unidades independientes. La unidad de interface con el bus (Bus Interface Unit, BIU) y la unidad de ejecución (Execution Unit, EU).

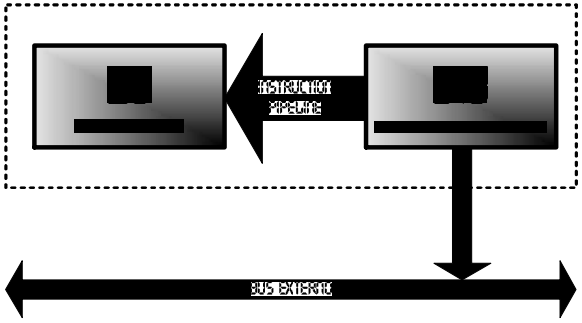


Fig 4.-Arquitectura "Pipeline" de los micros 8088/8086.

Estas unidades pueden comunicarse entre ellas aunque su operación habitual es trabajar individualmente en paralelo de forma independiente. La unidad de interface con el bus se encarga de ir a la memoria en busca de instrucciones y cargarlas en la cola de instrucciones. Esta unidad también se encarga de generar las señales de control del bus local (bus de control externo). El ir en busca de instrucciones a la memoria y almacenarlas en una memoria interna (cola de instrucciones), permite al microprocesador ejecutar instrucciones más rápido ya que la siguiente instrucción no es necesario ir a buscarla a memoria. Con esta estrategia, mejoramos el tiempo de ejecución y la utilización del bus local, ver figura.5.

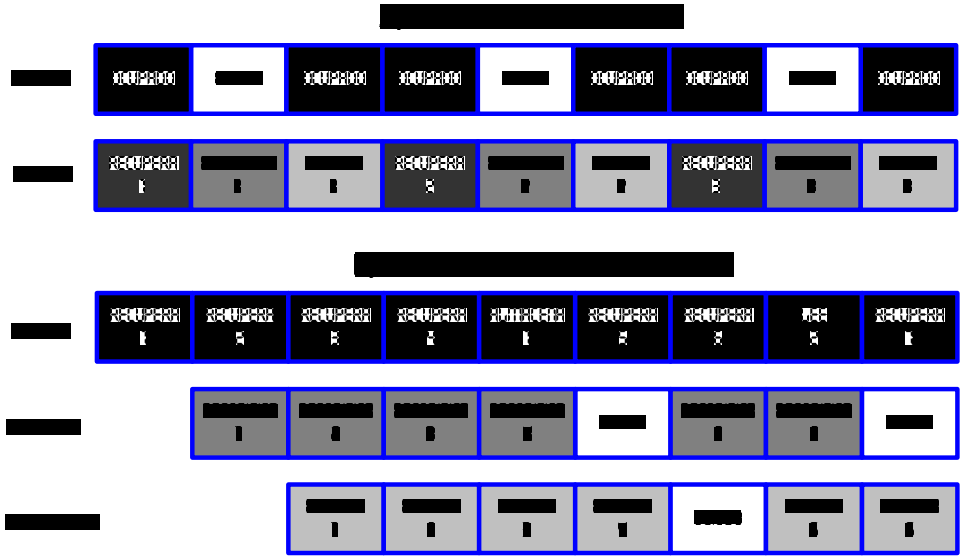


Fig 5.- a) Funcionamiento de un micro antiguo como el 8085A; b) Funcionamiento de los micros 8088/8086 con arquitectura paralela (pipeline).

En el 8086 la cola de instrucciones puede almacenar 6 bytes, mientras que el 8088 almacena 4 bytes de la memoria (ver diferencias entre el 8088 y el 8086]

Ir en busca de una secuencia de instrucciones (la BIU se encarga de ello) hace que utilicemos la memoria de una forma mucho más eficiente de lo que lo hacían los anteriores micros de Intel. Siempre que en la cola de instrucciones hay un espacio para almacenar un dato, la BIU lanza un ciclo de búsqueda de instrucciones que recoge el dato siguiente de la memoria y lo carga en la cola de instrucciones que actúa como un buffer intermedio con una estructura tipo FIFO (Frist-In-Frist-Out) del cual la EU extrae los datos cuando el decodificador los requiera. La cola de instrucciones es inicializada siempre que el microprocesador ejecuta una instrucción de salto, por eso el rendimiento de la cola de instrucciones no es muy bueno en códigos con muchas instrucciones de salto.

5.1 ORGANIZACIÓN DE LA MEMORIA

El procesador dispone de un bus de direcciones de 20 bits por lo que puede direccional un total de 2 ^ 20=1 Millón de posiciones de memoria (de 00000h a FFFFFh). La memoria está dividida lógicamente en 4 segmentos (segmento de Datos, Código, Pila y segmento Extra, ver figura. 8) de 64K bytes cada uno. Para acceder a 64K bytes, son necesarias 16 líneas por lo que los segmentos son almacenados en registros de 16 bits.

Todas las referencias a memoria, son calculadas de forma relativa a un registro base de acceso rápido, los registros de segmento. Este tipo de direccionamiento aumenta enormemente el potencial del software desarrollado. El registro de segmento es seleccionado automáticamente de forma implícita por las instrucciones [ver tabla 2]. Toda la información en un mismo tipo de segmento coparte unos ciertas características comunes (ej. DS contiene Datos, CS contiene Instrucciones...). Estructurando la memoria total del sistema en áreas con estas mismas características y simplemente cargando los registros internos de segmento con las direcciones iniciales

donde vamos a cargar los datos, el código... del programa, damos una estructura muy definida al software permitiendo realizar programas más cortos, rápidos y legibles.

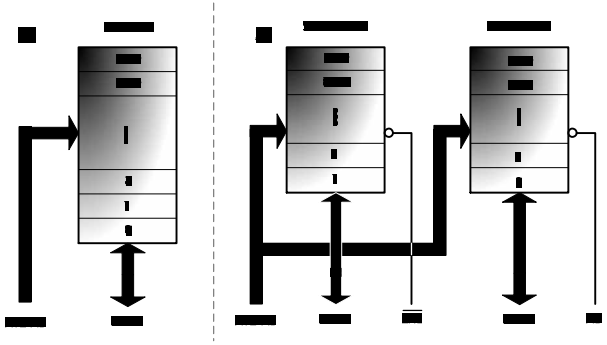


Fig 6.- a) Banco de memoria (1M x 8) en el 8088; b) Banco de memoria alto y bajo del 8086.

Los operandos de 16 bits pueden estar localizados en posiciones de memoria pares o impares, de manera que el byte menos significativo queda almacenado en la posición de memoria más baja y el byte más significativo en la posición de memoria inmediatamente superior. La BIU detecta automáticamente si es necesario uno o dos ciclos de bus para acceder o grabar el dato de 16 bits en memoria.

Datos Almacenados	Registro de Segmento	Reglas de Selección del Segmento
Instrucciones	CODE (Code Segment, CS)	La BIU lo utiliza automáticamente cuando accede en busca de datos para cargarlos en la cola de instrucciones.
Pila	STACK (Stack Segment, SS)	Todas las instrucciones que almacenan datos en la cola de instrucciones, utiliza este segmento como segmento por defecto. Todos los accesos a memoria que utilizan el desplazamiento relativo a BP, excepto en transferencia de datos.
Datos Locales	DATA (Data Segment, DS)	En transferencias de datos cundo son relativas a la pila, como destino en operaciones de cadenas o cundo se indica explícitamente por un prefijo de segmento.
Datos Externos o Globales	EXTRA (Extra Segment, ES)	Es utilizado por defecto como segmento destino en operaciones que tratan cadenas de datos o cuando son explícitamente indicados en el código mediante un prefijo de segmento.

Tab 2.- Tabla de segmentos.

En el microprocesador Intel ® 8088 (i8088) la memoria está organizada en un banco de 1 Megabyte que almacena bytes (bus de datos de 8 bits) por lo que no tenemos problemas de datos "desalineados", es decir, de acceso a datos de 16 bits en los que el byte de la parte alta del mismo se encuentra en la parte baja del dato recogido ya que recoge bytes. Este concepto cobra otra dimensión en el Microprocesador Intel ® 8086 (i8086), en los que la memoria física está organizada en dos bloques en paralelo de 512 Kbytes, uno para la parte alta del dato (D15-D8) y otro para la parte baja del mismo (D7-D0) conformando un bus de datos de 16 bits (D15-D0). Ver la figura 6. El micro direcciona estos 512 Kbytes con las 19 líneas del bus de direcciones (A19-A1). Las transferencias de byte a direcciones pares se realizan en D7-D0, mientras que la transferencia de bytes a direcciones impares (A0='1') se realizan posicionando el dato en D15-D8. En el caso de palabras de 16 bits (words) las transferencias a direcciones pares se realizan en un solo ciclo de bus poniendo/recogiendo el dato en D15-D0, la transferencia de words a direcciones impares requiere dos ciclos, el 1º recibe/escribe el dato menos significativo de D15-D8 y el segundo (y consecutivo) en D7-D0. El microprocesador dispone de dos líneas de habilitación de los bancos, BHEn y A0, con las que podemos controlar la escritura en uno de los bancos o en ambos. En el i8086 los ciclos de acceso a datos a memoria para carga de la cola de instrucciones leen palabras de 16 bits mientras en la cola de instrucciones quedan almacenados en bytes separados para que el decodificador pueda acceder a ellos correctamente.

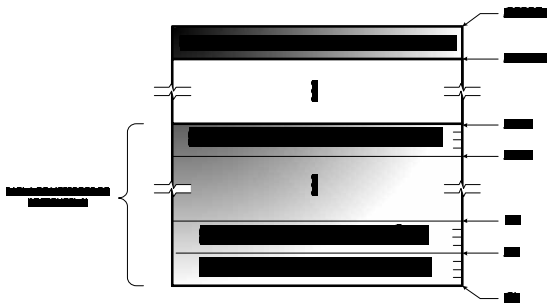


Fig 7.- Posiciones de Memoria Reservadas.

Ciertas posiciones de la memoria, tienen asignado un propósito específico en el comportamiento del microprocesador, ver la figura 7. El Espacio de memoria comprendido entre FFFF0h y FFFFFh está reservado para las instrucciones que se ejecutan nada más arrancar el micro entre las que se tiene que encontrar el salto al programa principal. Después de ejecutar un RESET del sistema, el microprocesador original "arranca" en la posición FFFF0h donde debe encontrarse el salto al programa principal o de inicialización. El rango de memoria comprendido entre 00000h y 003FFh está reservado para los vectores de interrupción. Cuatro posiciones de memoria guardan el segmento (16 bits) y el desplazamiento (16 bits) del vector que accede al servicio de una de las 256 interrupciones que el microprocesador puede atender. Los punteros de interrupción deben ser almacenados en su posición respectiva de memoria antes de activar el servicio de interrupción de dicha interrupción. [ver explicación de los servicios de interrupción].

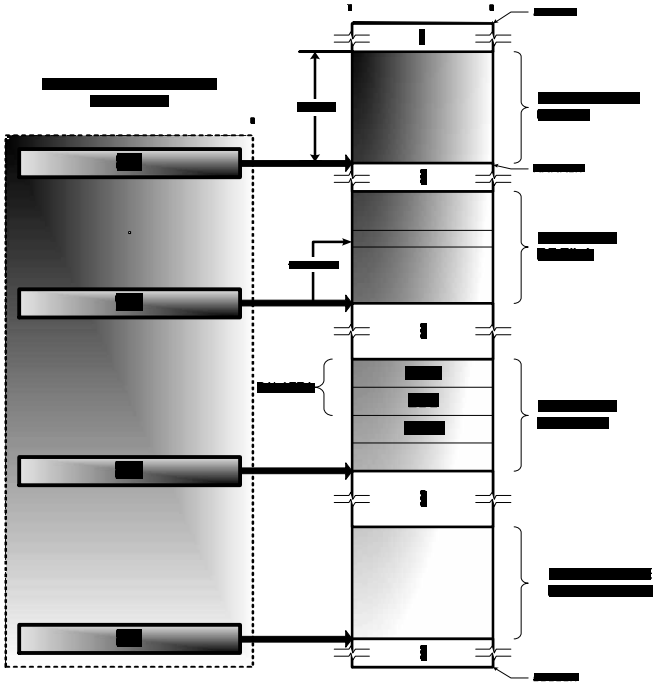


Fig 8.- Organización de la Memoria.

5.2 MODOS MÁXIMO Y MÍNIMO

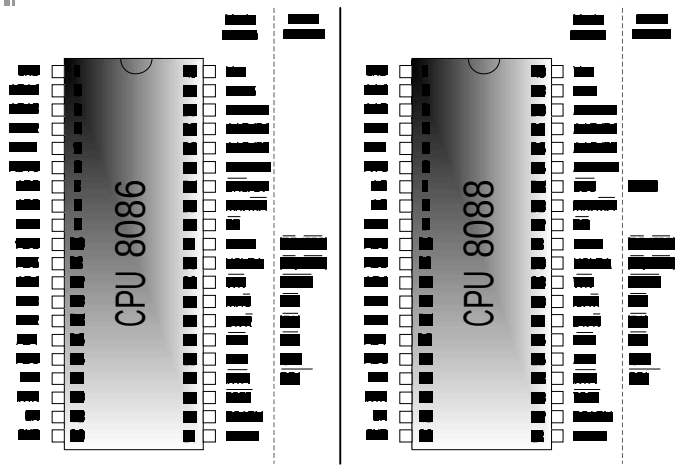


Fig 9.- Disposición de los pines.

Los requisitos de señales para que los micros soporten sistemas en modo máximo y mínimo son muy elevados para las carcasas de 40 pines utilizadas por los micros originales, de ahí que Intel ® decidiera multiplexar (mediante el pin de configuración MN/MXn) las señales internas para sacarlas al exterior por los pines del 24-31 en el i8086 y 24-31,34 en el i8088 (ver la figura 9) en función del sistema donde se va a acoplar el microprocesador (Sistema en Modo Máximo MN/MXn=GND, Sistema en Modo Mínimo MN/MXn= VCC). En los micros i8086/i8088 en modo máximo un controlador de bus externo (el 8288 de Intel ®, i8288) interpreta la información en las señales S0n, S1n y S3n para generar las señales del bus de control compatible con la arquitectura Multibus diseñada por Intel ®. En el modo mínimo las señales de control del bus son generadas por el propio microprocesador, de manera que no es necesario el controlador de bus externo i8288. El modo mínimo del i8088 es compatible con los periféricos MCS-85 lo que lo hace compatible con el anterior hardware desarrollado para el i8085 (Microprocesador de 8 bits desarrollado por Intel ®). La demultiplexación de las señales requiere un latch para memorizar las 8 líneas del bus de datos direcciones (DA7-DA0) en el i8088 y de un latch de 16 bits para memorizar las 16 líneas (DA15-DA0) del i8086, o bien un match de 20 bits si queremos enclavar el bus de direcciones completamente, la señal ALE se encarga de enclavar estos latches. Las señales DENn y DT/Rn proporcionan el soporte para añadir un transceptor bidireccional al bus de datos siendo éste de 8 o 16 bits para los micros i8088 e i8086 respectivamente. Acoplando las señales del micro al bus local por medio de latches y transcivers aislamos el micro del sistema y mejoramos los requisitos temporales de propagación de señales.

El modo máximo utiliza el controlador de bus i8288 que decodifica las líneas de estado S0n, S1n y S2n y proporciona un bus de control con todas las señales necesarias. La utilización del i8288 permite, además de mejorar las características de corriente y por lo tanto de velocidad de las señales del bus de control, liberar pines de los microprocesadores de manera que podemos utilizarlos para implementar nuevas funciones en el sistema en modo máximo. El modo

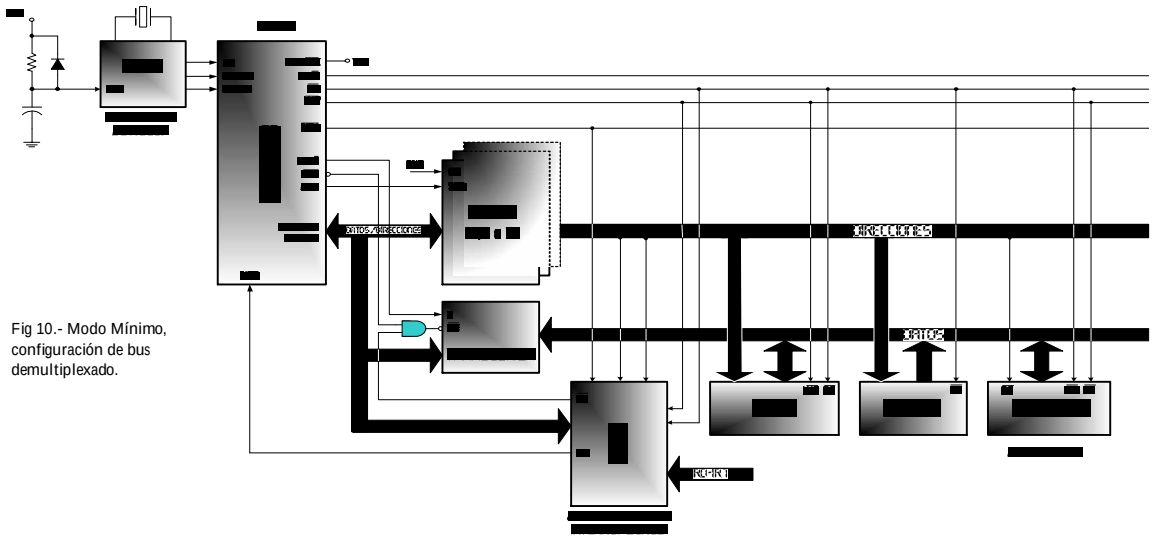


Fig 10.- Modo Mínimo, configuración de bus demultiplexado.

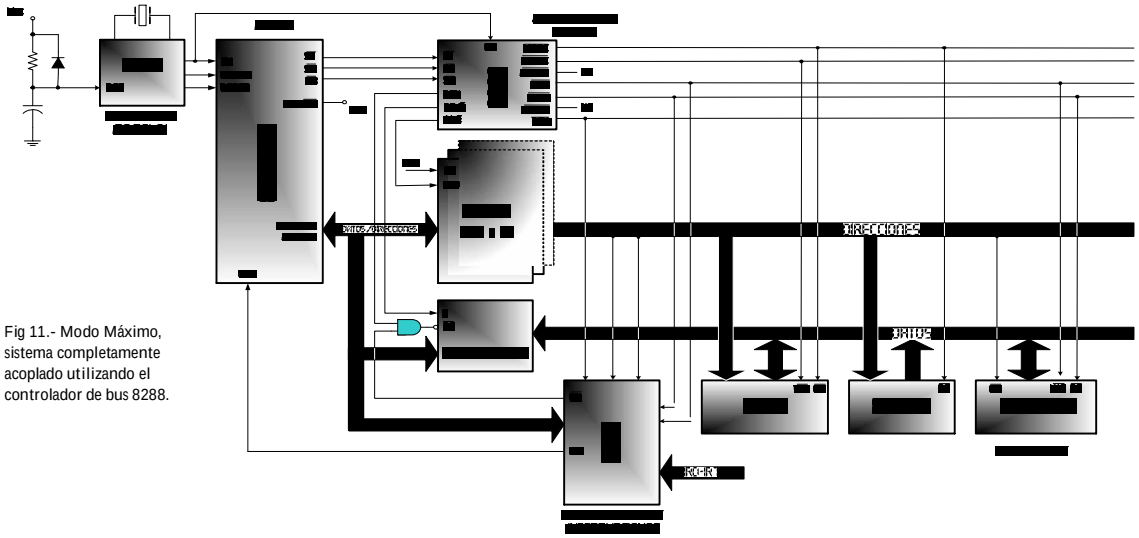


Fig 11.- Modo Máximo, sistema completamente acoplado utilizando el controlador de bus 8288.

máximo de los microprocesadores admite el bloqueo físico de hardware (activando el pin LOCK), ver el estado de la cola de instrucciones (señales QS0, QS1) y amplia el interface de requerimiento/cesión del bus local a dos canales (RQ0n/GT0n, canal 0 y RQ1n/GT1n, canal 1). Estas nuevas características permiten, entre otras cosas, soportar el coprocesador matemático i8087 y añadir lógica de control de bus local y buses externos (soporte Multibus). Ver las figuras 10 y 11.

5.3 FUNCIONAMIENTO DEL BUS

El bus de datos y direcciones está multiplexado en el tiempo, lo que significa que los pines AD? contienen, a veces, Datos y, a veces, Direcciones. Con esta técnica conseguimos aprovechar de la mejor forma posible los pines pudiendo insertar en microprocesador en una carcasa estándar de 40 pines. (esto no tiene mucho sentido en nuestro modelo ya que rara vez lo veremos en un sistema original, ver la figura 12. Este bus local puede ser utilizado directamente o a través de registros de enclavamiento que memoricen la dirección enviada a la memoria o a los dispositivos de entrada salida, I/O. Si deseamos tener un bus demultiplexado, podemos hacerlo fácilmente con la ayuda de las señales del micro y una serie de registros de enclavamiento.

El bus de direcciones/datos del i8088 está dividido en tres partes, el byte menos significativo (AD7-AD0), el segundo byte del bus de direcciones (A15-A8), y la parte alta del bus de direcciones (A19-A15), mientras que el del i8086 está dividido sólo en dos partes, la palabra menos significativa (AD15-AD0) y la parte alta del bus de direcciones (A19-A16).

Cada ciclo de bus dura al menos cuatro ciclos de reloj, CK. Estos cuatro ciclos podemos nombrarlos como T1, T2, T3 y T4. Durante T1 aparece una dirección válida y durante T3, T4 se realiza la transferencia del dato, T2 se utiliza en principio para cambiar el sentido del bus de datos durante operaciones de lectura. Si el dispositivo de memoria no está listo para realizar la transferencia, se insertan ciclos de espera (Tw) entre los ciclos T3 y T4. Cada ciclo de espera dura 1 CK. Entre dos ciclos de bus consecutivos, pueden aparecer ciclos en los que el bus

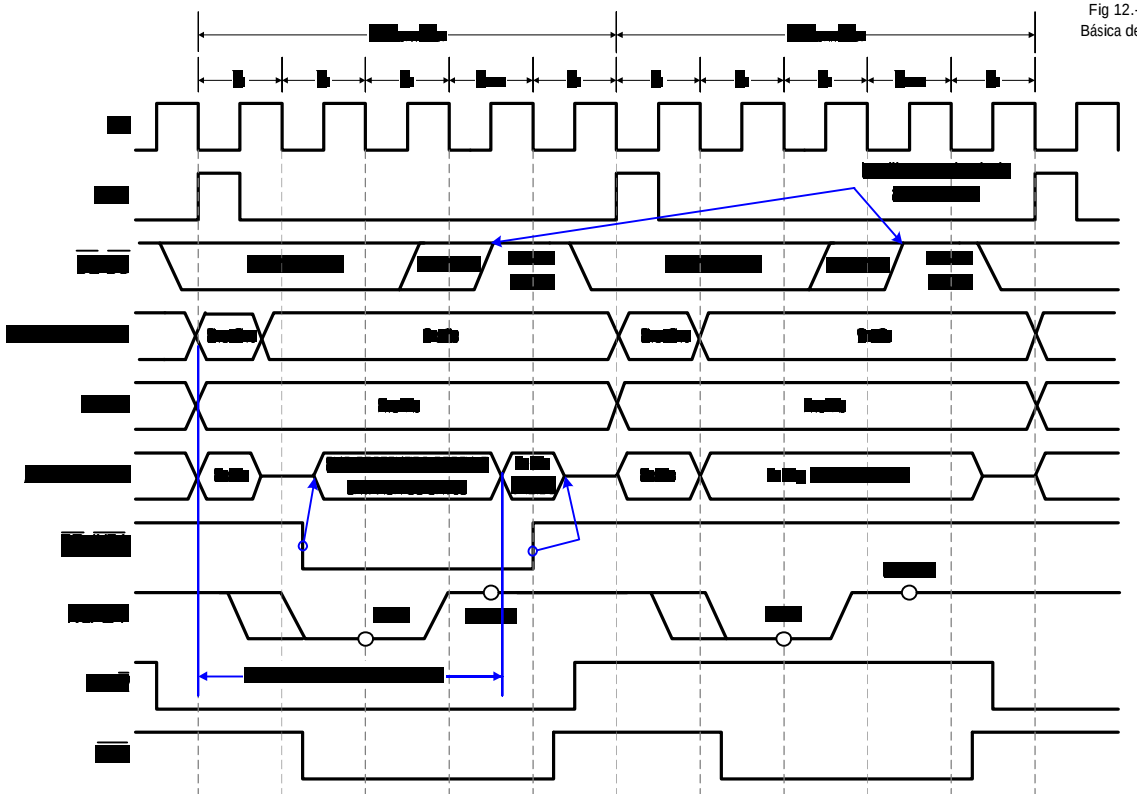


Fig 12.- Temporización Básica del Sistema (8088).

está inactivo ya que el microprocesador está realizando operaciones internas, estos ciclos podemos llamarlos ciclos ociosos “idle” y denominarlos Ti.

Durante el ciclo T1 de cualquier ciclo de bus, aparece la señal ALE (Address Latch Enable). Esta señal es generada por el microprocesador cuando estamos trabajando en modo mínimo o por el controlador de bus externo i8288 cuando estamos utilizando el microprocesador en un sistema estructurado en modo máximo. El flanco de bajada de esta señal enclava una dirección válida y ciertos bits del sistema en los registros de enclavamiento “latches” externos.

Las señales de estado S2n, S1n y S0n son utilizadas por el controlador de bus (en modo máximo) para identificar el tipo de ciclo de bus que se va a ejecutar. Ver la siguiente tabla:

S2n	S1n	S0n	Características
0	0	0	Reconocimiento de interrupción
0	0	1	Lectura de I/O
0	1	0	Escritura de I/O
0	1	1	Halt
1	0	0	Búsqueda de instrucción
1	0	1	Lectura de datos desde la memoria
1	1	0	Escritura de datos en la memoria
1	1	1	Pasivo (No hay ciclo de Bus)

Tab 3.- Decodificación de S0n-S2n.

Los bits de estado S3-S6 en el i8088 y S3-S7 en el i8086 están multiplexados en el tiempo (válidos en T2-T4) con los bits de la parte alta del bus de direcciones (AS19-AS16, bus Address/System). S3 y S4 indican el registro interno de segmento que fue utilizado en la generación de la última dirección válida. Ver la siguiente tabla:

S4	S3	Características
0	0	Segmento Extra
0	1	Segmento de Pila
1	0	Código o Ninguno
1	1	Segmento de Datos

Tab 4.- Decodificación de S4-S3. Segmento Utilizado.

El bit S5 es un "espejo" del bit del registro de banderas que indica si están habilitadas las interrupciones (Bandera de Interrupción "Interrupt Flag", IF). Los bits S6 y S7 tienen una función específica que se comenta más adelante.

5.4 DIRECCIONAMIENTO DE I/O

En los microprocesadores, las operaciones de entrada/salida pueden efectuarse en un rango de direccionamiento de 64Kbytes (16 líneas de direcciones) o 32K de palabras si accedemos a puertos de 16 bits en el i8086. El bus de direcciones de los dispositivos I/O comparten las mismas líneas que el del direccionamiento de la memoria (A15-A8, AD7-AD0 en el i8088 y AD15-AD0 en el i8086) las líneas A19-A16 permanecen a cero durante las transferencias a I/O. Hay dos tipos de instrucciones para acceder a los dispositivos I/O, las de puntero variable y las de acceso directo, las primeras pueden acceder a cualquiera de los 64K de los dispositivos I/O especificando el puntero en el registro DX, y las segundas sólo pueden acceder a los 256 puertos (8 bits de direcciones). En contra, las instrucciones de acceso directo son más rápidas y ocupan menos al especificar el puerto como dato inmediato en el propio código de la instrucción. En el i8086 hay que tener en cuenta que las transferencias a puertos situados en posiciones pares que contienen dispositivos de 8 bits se realizan por las líneas D7-D0 mientras que las transferencias a dispositivos que manejan bytes localizados en posiciones de memoria impares se transmiten por las líneas D15-D8. Al transferir registros internos a dispositivos I/O de 8 bits debemos tener en cuenta que si el puerto está localizado en una dirección par 244h (por ejemplo) y mandamos el dato 56F4h (por ejemplo) el dispositivo recibirá el dato F4h, mientras que si está localizado en una posición impar 245h el dato que recibirá será 56h.

5.5 COMUNICACIÓN CON EL EXTERIOR

INICIALIZACIÓN Y RESETEADO DEL MICROPROCESADOR.

La inicialización del microprocesador comienza con la activación (a nivel alto) de la patilla RESET. El pin RESET tiene que estar activado por lo menos durante 4 ciclos de reloj, de lo contrario el micro no se inicializa, con esta lógica evitamos resetear el sistema accidentalmente por estados espurios o "glich" que pueden aparecer por sobrecorrientes, ruido... en este terminal, ver la figura 13. Una vez que el RESET ha sido aceptado, el micro permanece dormido mientras RESET esta activado, finalmente en el flanco de bajada de RESET se lanza una secuencia de inicialización que dura unos 7 ciclos de reloj en el i8088 y 10 CK en el i8086 tras la cual el microprocesador empieza a ejecutar instrucciones desde la posición de memoria FFFF0h (ver la figura 7). La señal de RESET está sincronizada internamente. Al conectar la alimentación del sistema, se debe inicializar el microprocesador manteniendo a nivel alto la señal de RESET por lo menos durante 50 microsegundos. Se consigue fácilmente con una red Condensador-Resistencia, RC.

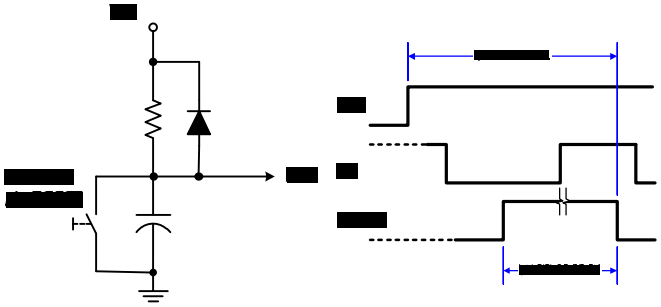


Fig 13.-Temporización de RESET.

Todos los terminales triestado permanecen en alta impedancia durante la inicialización del microprocesador. Una vez que RESET ha sido aceptado el estado del sistema pasa a ser Ti (estado ocioso) y permanece en este estado mientras RESET está activo. Durante la secuencia de inicialización, los pines de estado del sistema están en alta impedancia. Las señales ALE y HLDA están a nivel 0 lógico durante la inicialización.

OPERACIONES DE INTERRUPCIÓN

Las Interrupciones pueden ser de procedencia software o hardware. Las interrupciones software son una forma de llamada a partes específicas de un programa (son interrupciones en el flujo lógico de un programa, aspectos relativos a la programación) mientras que las interrupciones hardware interfieren en el flujo de funcionamiento normal del microprocesador para atender una petición realizada por un dispositivo externo. A estas interrupciones hardware también suele acompañarles unas instrucciones específicas que debe realizar el microprocesador, por lo que también tienen una componente software (el llamado servicio de interrupción). Podemos dividir las interrupciones hardware en Interrupciones Enmascarables (Maskable INTeRrupts, INTR) e Interrupciones No-Enmascarables (Non-Maskable Interrupts, NMI).

Las interrupciones provocan un salto a otro programa, el "servicio de interrupción". Una tabla con 256 elementos en el espacio de memoria comprendido entre 00000h y 003FFh (Ver la figura 7) contiene la posición a la que hay que saltar para atender la interrupción, esta posición de memoria se denomina "puntero o vector del servicio de interrupción" y la tabla "tabla de vectores o punteros de interrupción" cada puntero de servio de interrupción está formado por 2 palabras (4h posiciones consecutivas de memoria) de 16 bits (notar que 256 elementos equivale a FFh en hexadecimal y que FFh*4h=3FFh que es el espacio total que ocupa la tabla de vectores de interrupción), la primera indica el segmento donde se encuentra el servicio y la segunda el desplazamiento dentro de este segmento. La posición dentro de la tabla de vectores se denomina "tipo de interrupción" y

en las interrupciones hardware este dato es proporcionado por el dispositivo que solicita la interrupción, el microprocesador lee este dato y calcula el salto (tipo*4dirección) a la tabla de vectores de interrupción donde conseguirá los datos (segmento y desplazamiento) para saltar al servicio de interrupción. En las interrupciones software el tipo de interrupción es indicado directamente en el código.

INTERRUPCIONES NO ENMASCARABLES, (NON-MASKERABLE INTERRUPTS, NMI)

El microprocesador proporciona un pin para la solicitud de interrupciones no enmascarables (NMI). Este pin tiene una prioridad de ejecución mayor que el de solicitud de interrupción mascarable (INTR). Una aplicación típica de NMI es lanzar una rutina cuando se detecta un fallo en la alimentación del sistema. Esta interrupción es activada por el flanco de subida y provoca implícitamente una interrupción de tipo 2.

La señal NMI tiene que permanecer por lo menos dos ciclos en estado alto para ser validada, el sincronismo se realiza internamente. Cualquier transición de cero a uno lógicos de NMI es memorizado internamente. El microprocesador atiende la petición al final de la ejecución de la instrucción actual o, en el caso de estar ejecutando instrucciones con cadenas, cuando se completa la transferencia actual. El peor caso en la respuesta a una petición de NMI se da cuando se están ejecutando las instrucciones de Multiplicación y División con o sin signo o en el caso de estar ejecutándose una instrucción de desplazamiento con contador, en todos los casos hay que esperar a que la instrucción se complete (que puede ser de 1 a 256 ciclos). La transición a nivel bajo de NMI puede ocurrir antes, durante o después de que haya sido atendida. Si aparece otro flanco de subida en NMI mientras el micro está atendiendo otra NMI anterior, se memoriza la solicitud y es atendida al final del servicio de interrupción. Es importante que NMI sea lo más limpia posible evitando estados espurios eliminando de esta manera el disparo accidental.

INTERRUPCIONES ENMASCARABLES (MASKABLE INTERRUPTS, INTR)

Estos micros tienen una única entrada de solicitud de interrupciones enmascarables, el pin INTR. El estado de éste terminal puede ser ignorado (enmascarado) si el bit de habilitación de interrupción interno está a cero (Interrupt Flag, IF='0'). INTR a diferencia de NMI es activado por nivel y está sincronizado con el flanco de subida de CK. Para que INTR sea atendida, debe permanecer activada (a '1' lógico) hasta el final de la ejecución de la instrucción en curso o hasta que se complete una instrucción de transferencia de bloques. En la secuencia de pasos que realiza el coprocesador para atender la petición de INTR, se borra la bandera IF, como consecuencia otras interrupciones mascarables no serán, en principio, atendidas. Y digo en principio, porque en el servicio de interrupción podemos activar IF, llegamos al concepto de anidamiento, "nesting". IF no sólo se borra en la atención a INTR, sino que también se borra cuando el microprocesador atiende NMI, interrupciones software o la interrupción "paso a paso". Aunque la atención de la interrupción modifique IF, al retornar del servicio de interrupción recuperamos el valor anterior de IF ya que antes de borrar IF el registro de banderas es salvaguardado en la pila del sistema junto con otros registros. Ver la explicación de las interrupciones en la unidad de interrupciones.

El microprocesador ejecuta dos ciclos de bus consecutivos durante la secuencia de respuesta a la petición de INTR. Ver la figura 14. El micro activa la señal de bloqueo, LOCK (sólo en el modo máximo) desde T2 del primer ciclo de reconocimiento de interrupción hasta T2 del segundo ciclo. Las peticiones de cesión del bus local no son atendidas hasta el final del segundo ciclo de reconocimiento de interrupción. Durante el segundo ciclo de reconocimiento de interrupción, se recoge un byte del bus de datos, este byte es el tipo de interrupción requerido por el dispositivo solicitante. Normalmente se utiliza el controlador externo de interrupciones diseñado por Intel ®, el PIC 8259ª. Este byte es multiplicado por cuatro y lo utilizamos como puntero en la tabla de vectores de interrupción (como explicábamos anteriormente en "Operaciones de Interrupción"). Si mantenemos siempre activada INTR, la interrupción es atendida continua e ilimitadamente. En la instrucción de retorno de interrupción, se recupera el valor previo (previo a la atención de la interrupción) del registro de banderas por lo que restituimos el valor de IF y del resto de las banderas.

5.6 HALT

Cuando se ejecuta la instrucción HALT, el microprocesador indica que está entrando en estado HALT, dependiendo del modo de funcionamiento, de

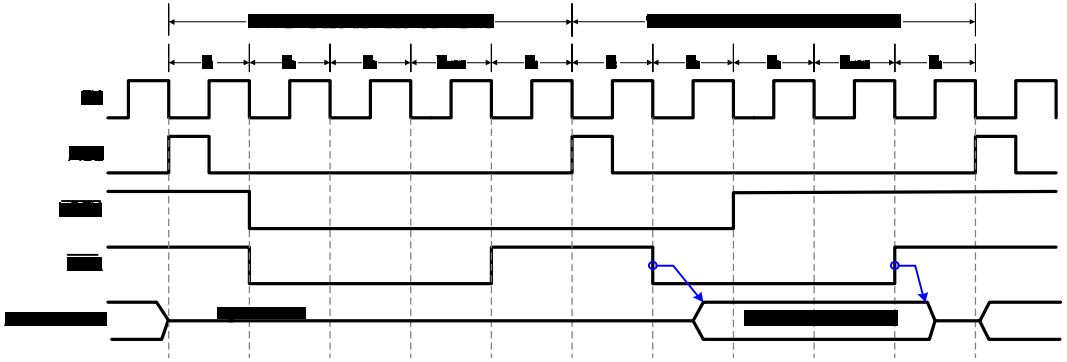


Fig 14.-Secuencia de Reconocimiento de Interrupción.

una de las siguientes maneras:

- En Modo Mínimo, el procesador activa la señal ALE para permitir enclavar el nuevo estado "halt". Este estado está presente en los terminales IO/Mn, DT/Rn y SS0n.
- En el Modo Máximo, el procesador saca el estado "halt" por los pines S2n, S1n y S0n y el controlador de bus externo i8288 genera la correspondiente señal ALE.

El microprocesador no abandona el estado "halt" cuando un dispositivo externo requiere el bus local (activación de la señal HOLD), el micro cede el bus al dispositivo, pero cuando éste termina de utilizar el bus local, el microprocesador vuelve al estado "halt". El requerimiento de una interrupción mascarable o no, así como la activación de RESET (que en definitiva es la interrupción de mayor prioridad) fuerzan al microprocesador a abandonar el estado "halt". Por esta última razón la instrucción HALT suele ser utilizada para sincronizar hardware externo con el software.

OPERACIONES UTILIZANDO LOCK. LECTURA/CAMBIO/ESCRITURA "SEMÁFORO"

La señal de bloqueo (LOCK) es generada por el micro cuando la ejecución de una instrucción requiere dos ciclos de bus consecutivos. Esto permite al microprocesador realizar operaciones de lectura/cambio/escritura en memoria sin que pueda intervenir otro maestro de bus en esta secuencia. Esta característica es utilizada en sistemas multiprocesador en los que se realizan operaciones de "test y bloqueo" de manera que un arbitro de canal conceda los buses a uno y sólo uno de los microprocesadores. La señal LOCKn (que es activada a nivel bajo) también puede ser activada por el programador mediante el prefijo de instrucción (o instrucción LOCK), en este caso la señal se desactiva cuando termina el último ciclo de bus de la instrucción que acompaña al prefijo LOCK en la secuencia de instrucciones. Si mientras la señal LOCKn está activada, aparece un requerimiento de bus en RQ/GT, éste es memorizado y atendido al final de la instrucción LOCK.

5.7 SINCRONIZACIÓN EXTERNA UTILIZANDO LA SEÑAL TESTn

Como alternativa a las interrupciones, los microprocesadores i8088/i8086 disponen de una entrada testeable por software (el pin TESTn). La instrucción WAIT se repite continuamente hasta que el valor del pin TESTn es cero. Cuando aparece una instrucción WAIT, la BIU sigue ejecutando ciclos de búsqueda de instrucciones hasta llenar la cola de instrucciones, momento en el que el bus pasa al estado ocioso, Ti.

Si durante la ejecución de WAIT aparece una petición de bus local, todas las salidas triestado de los microprocesadores pasan a estado de alta impedancia. Si aparece una petición de interrupción en estado de espera "wait", la procesa y al retornar del servicio de interrupción vuelve a ejecutar la instrucción WAIT.

5.8 TEMPORIZACIÓN BÁSICA DEL SISTEMA

En el modo de funcionamiento mínimo, el pin MN/MX está conectado a Vcc ('1' lógico) y el microprocesador emite señales por el bus de control compatibles con la estructura de los sistemas desarrollados para el 8085. En modo máximo (MN/MXn=GND, '0' lógico) el microprocesador saca al exterior información sobre el estado interno. Esta información permite, entre otras cosas, al controlador de bus i8288 generar las señales de control necesarias para implementar sistemas con soporte MULTIBUS.

5.8.1 TEMPORIZACIÓN DEL SISTEMA – SISTEMA MÍNIMO

Como podemos ver en la figura 12, los ciclos de lectura comienzan en T1 con la activación de la señal de activación de los registros de enclavamiento (Address Latch Enable, ALE). El flanco de bajada de ALE es utilizado para registrar la información de la dirección en los registros transparentes, esta información aparece en los terminales AD7-AD0 en el i8088 y en AD15-AD0 en el i8086 en el estado T1. Las líneas del bus de direcciones A15-A8 en el i8088 no es necesario enclavarlas ya que se mantienen con el mismo valor durante todo el ciclo de bus. De T1 a T4 la señal IO/Mn en el i8088 o M/IOn en el i8086 indica si va a ser una transferencia a memoria o a un dispositivo de entrada-salida I/O. En T2 desaparece la dirección del bus direcciones-datos (AD?) y pasa al estado de alta impedancia. La señal de lectura (RDn) del bus de control, también aparece en T2 una vez que el bus de datos está en alta impedancia. Esta señal permite al dispositivo direccionado activar sus "drivers" de datos y conectarse al bus local. Algún tiempo después, habrá un dato válido en el bus de datos y la señal de dato válido en el bus (READY) será activada. Cuando el procesador devuelve la señal de lectura a su nivel inactivo (RDn='1'), el dispositivo direccionado pone sus terminales en estado de alta impedancia, por lo que se desconecta del bus local. Si colocamos un transceptor bidireccional en el sistema, utilizaremos las señales DT/Rn y DENn proporcionadas por el microprocesador para controlarlo.

Un ciclo de escritura, también comienza activando la señal ALE y sacando la dirección. La señal IO/Mn en el i8088 o M/IOn en el i8086 indica si la transferencia es a memoria o a un dispositivo de entrada-salida. En T2, inmediatamente después de las direcciones, aparece el dato a transferir. Este dato permanece válido hasta la mitad de T4. Durante T2, T3 y Tw, el procesador activa la señal de escritura (WRn='0'). Esta señal, a diferencia de RDn que aparece al final de T2 para permitir al bus de datos-direcciones entrar en alta impedancia, es activada al comienzo de T2.

En el i8086, las señales BHEn y A0 son utilizadas para mandar el dato de forma adecuada al dispositivo I/O o Memoria direccionado. Ver siguiente tabla:

BHEn	A0	Características.
0	0	Palabra completa.
0	1	Byte Superior hacia/desde una dirección impar.
1	0	Byte Bajo hacia/desde una dirección par.
1	1	Sin Dato (Nada)

Tab 5.- Señales de control de los bancos de memoria.

Los puertos I/O son direccionados de la misma forma que la memoria. Los bytes transmitidos a direcciones pares se encuentran en D7-D0, mientras que las transferencias a direcciones impares se efectúan en D15-D8.

La diferencia básica entre el ciclo de reconocimiento de interrupción y el ciclo de lectura en que en el reconocimiento de una interrupción se activa la señal INTAn en lugar de RDn. Además en el ciclo de reconocimiento de interrupción el bus de direcciones permanece en estado de alta impedancia. Ver la figura 14. En el segundo ciclo de reconocimiento de interrupción (se ejecutan dos ciclos consecutivos), se lee un dato del bus, este dato suele ser proporcionado por el dispositivo externo de control de interrupciones (el Priority Interrupt Controller PIC 8259A). Este dato de 8 bits indica el tipo de interrupción que es multiplicado por cuatro y utilizado como puntero en la tabla de vectores de interrupción (como se explicó anteriormente).

5.8.2 TEMPORIZACIÓN DEL SISTEMA – SISTEMA MÁXIMO

Para conectar los micros a sistemas estructurados en modo máximo, el pin MN/MXn se conecta a GND ('0' lógico), se añade el controlador de bus i8288 y registros de enclavamiento para memorizar la dirección. También se puede acoplar un transceptor bidireccional al sistema para mejorar las características de carga de los pines del microprocesador. Las señales ALE, DENn y DT/Rn son generadas por el i8288 en lugar de ser generadas por el microprocesador, pese a todo, los tiempos en los que aparecen estas señales, se mantiene con respecto al funcionamiento en un sistema mínimo. Las salidas S2n, S1n y S0n del bus de estado, indican al controlador de bus i8288 el ciclo de bus que se está ejecutando, si se trata de un ciclo de lectura (de código, de datos o de una entrada salida I/O), de escritura (de datos o hacia un dispositivo I/O), un ciclo de reconocimiento de interrupción o una parada del sistema por software (halt). El i8288 genera las señales correspondientes en función del ciclo de bus indicado, generando señales de lectura o escritura de la memoria o de un dispositivo I/O o bien indicando que se trata de un reconocimiento de interrupción.

El puntero de la tabla de vectores de interrupción que es leído durante el segundo ciclo del reconocimiento de interrupción, puede ser dado por el controlador de interrupciones 8259A o por otro dispositivo, por ejemplo, el propio periférico que solicita la interrupción.

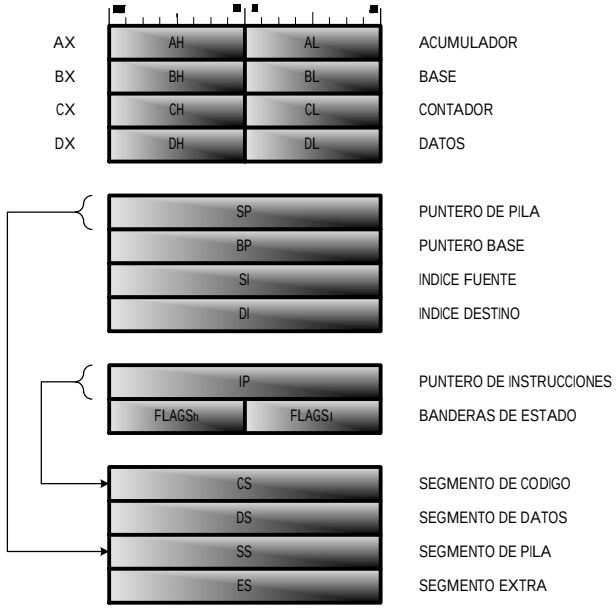


Fig 15.- Modelo de los Registros Internos

5.9 EL 8088 COMPARADO CON EN 8086

El microprocesador i8088 (bus de datos externo de 16 bits) está diseñado en base al a la estructura interna del i8086 (bus de datos externo de 16 bits). La mayoría de las funciones internas son iguales en ambos microprocesadores. Los microprocesadores manejan el bus externo de la misma manera, con la salvedad de que el bus de datos del i8088 es de 8 bits. Los datos de 16 bits en el i8088 son transferidos en dos ciclos de bus consecutivos. Los dos microprocesadores utilizan el mismo juego de instrucciones por lo que el programa que "corre" en el i8088, lo hace también en el i8086 (solo que más rápido). La estructura de registros interna es igual en ambos microprocesadores (ver la figura 15) y las instrucciones tienen el mismo resultado. Abajo se muestran las diferencias entre los micros i8088 e i8086 originales. Internamente hay tres diferencias básicas (todas tienen que ver con el cambio del bus de datos externo a 8 bits) entre el i8088 y el i8086:

- La cola de instrucciones es de 4 bytes en el i8088, mientras que el i8086 tiene 6 bytes, o tres palabras. La disminución del número de bytes de la cola de instrucciones en el i8088 previene la sobre utilización del bus local cuando el micro (concretamente la BIU) accede a memoria en búsqueda de instrucciones. El tiempo que tarda el i8088 en acceder a una palabra de memoria ahora es mayor (requiere dos ciclos consecutivos) por lo que la utilización del bus para tareas de alimentación de la cola de instrucciones interna era muy elevado aumentando el tiempo respuesta (latency) en la atención a interrupciones y peticiones de bus local. La solución: disminuir el tamaño de la cola de instrucciones.
- Para aumentar el rendimiento de la cola de instrucciones, el algoritmo de control de la unidad encargada de ir en busca de instrucciones a la memoria (Instruction Fetch Unit, IFU), ha cambiado. La IFU del i8088 irá en búsqueda de una instrucción para cargarla en la cola de instrucciones siempre que haya un byte libre en ésta. El i8086 espera a que haya un segundo byte libre y así ir a por una palabra a memoria. El i8086 llena la cola de instrucciones de dos en dos bytes, mientras que el i8088 lo hace de uno en uno.
- El tiempo de ejecución queda alterado por el retraso intrínseco introducido por el cambio de interface (de 16 bis a 8 bits). Todas las transferencias de 16 bis hacia/desde la memoria o I/O requieren cuatro ciclos de reloj (por lo menos) más que los utilizados en el i8086. La velocidad del microprocesador está limitada por el tiempo de acceso a datos en memoria en dos sentidos: 1º hay instrucciones en las que el micro no puede continuar hasta tener un dato de memoria en sus registros internos. 2º el decodificador de instrucciones no puede decodificar instrucciones largas hasta que la IFU no traiga todos los datos

del código de operación (Operación Code, OP). El micro espera a la decodificación completa. El segundo problema aparece en programas en los que se ejecutan instrucciones simples (tardan pocos ciclos en ejecutarse), con instrucciones que tardan en ejecutarse más tiempo, la IFU puede operar cargando la cola de instrucciones de manera que las decodificaciones siguientes cogen datos de la cola de instrucciones (mientras sea posible) y no es necesario esperar a recibir la instrucción de la memoria.

El i8088 y el i8086 son compatibles 100% a nivel de software debido a que disponen de la misma unidad de ejecución (Execution Unit, EU). El software que tiene dependencias hardware no es completamente inter-cambiable si ambos sistemas no son compatibles, es decir, si en un programa hacemos referencia a un puerto I/O de 16 bits que sólo existe en el sistema desarrollado para un i8086, ese programa sólo podrá ejecutarse en ese sistema; como es lógico. Si el programa no contiene ningún tipo de dependencia hardware, “correrá” igualmente en un i8086/i8088.

El interface con el exterior es dónde se encuentran las mayores diferencias entre ambos micros. La asignación de pines de los microprocesadores es prácticamente idéntica (ver el patillaje en la figura 9) a excepción de los siguientes cambios:

- A15-A8. Estos pines contienen durante todo el ciclo de bus, el segundo byte de la dirección (en el i8088). Estas líneas son registradas internamente y permanecen válidas durante todo el ciclo de bus como se comentó antes y de manera similar a lo que ocurre con el segundo byte del bus de direcciones del microprocesador i8085. El i8088 nació de la necesidad de reutilización del hardware ya diseñado para el i8085.
- BHEn no tiene sentido alguno en el micro i8088 y es eliminada.
- SS0n es el bit de estado S0n en modo mínimo. Esta salida aparece en el pin 34 sólo en el modo mínimo. DT/Rn, IO/Mn y SS0n proporcionan la información completa del bus de estado en modo mínimo.
- IO/Mn has sido invertida (M/IOn en el i8086) para hacerla compatible con la estructura de bus desarrollada en la familia antecesora, MCS-85.
- ALE es retrasada un ciclo en el modo mínimo al entrar en el estado HALT. Esto permite enclavar el estado con la señal ALE.

estructura de los modelos

6. DIAGRAMA DE BLOQUES GENERAL

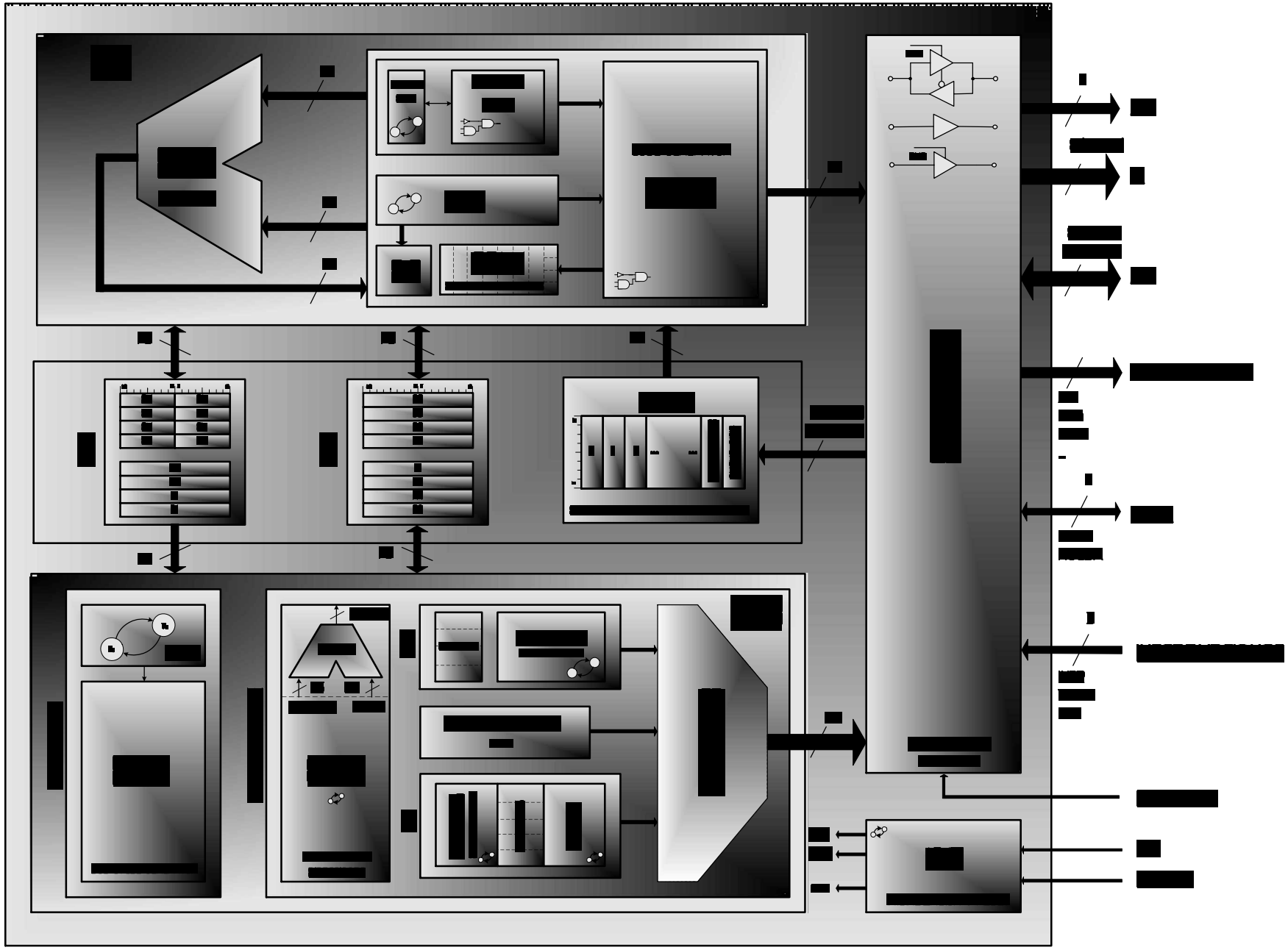


Fig 16.- Diagrama de Bolques General de los Modelos Desarrollados en este Proyecto

7. DESCRIPCIÓN PARTICULAR

7.1 DIAGRMA JERÁRQUICO DE BLOQUES

ARQUITECTURA INTERNA BÁSICA:

El microprocesador diseñado, el i8088/i8086 significó un salto en la evolución de los microprocesadores al implementar una estructura basada en múltiples unidades trabajando en paralelo, nace el concepto de "pipeline". Cada una de estas unidades ejecuta una función específica dentro del microprocesador, en realidad estas unidades son microprocesadores independientes diseñados para un propósito específico.

En la figura 5 se puede ver el funcionamiento normal del antecesor del los micros diseñados, el i8085 (arquitectura de 8 bits). Nótese que las instrucciones se recuperan de memoria con una operación de lectura, por lo que mientras el i8085 ejecuta la instrucción, el bus externo está ocioso.

Por otro lado los micros i8088/i8086 aprovechan este tiempo ocioso del canal para ir en busca de instrucciones por lo que el canal está casi siempre utilizado aumentando la eficiencia del sistema..

En el diagrama de bloques general, se puede observar la jerarquía del modelo. También se puede observar el fichero fuente que define cada uno de los módulos del microprocesador.

Los ficheros fuente están nombrados jerárquicamente, lo que permite una fácil organización alfanumérica, ejecutando, por ejemplo, el comando "dir" de MS-DOS (c:\>dir) en el directorio donde se encuentra el código fuente, podemos ver a simple vista la estructura del modelo.

Por el nobre del fichero se puede detectar la procedencia completa, es decir, en el nombre del archivo fuente está descrito el descenso en la ruta jerárquica hasta llegar a él (ver esquema Jerárquico u poco mán adelante). Por ejemplo: "up8088_biu_interf _dfu_ctrl.vhd" podemos leer literalmente el nombre y decir que define el bloque de control (ctrl) de la unidad encargada de ir en busca de datos (Data Fetch Unit,

```
C:\up8088\>dir
 El volumen de la unidad E es Dame Datos!
 El n£mero de serie del volumen es: 541F-5CC8

 Directorio de E C:\up8088

01/07/2004  16:28    <DIR>          .
01/07/2004  16:28    <DIR>          ..
12/05/2004  14:52            20.190 pack8088_pkg.vhd
21/04/2004  14:25            12.769 up8088_biu.vhd
21/04/2004  14:25            16.174 up8088_biu_interf.vhd
21/04/2004  14:25            13.476 up808x_biu_interf_arbiter.vhd
21/04/2004  14:25             7.067 up8088_biu_interf_dfu.vhd
21/04/2004  14:25            11.018 up8088_biu_interf_dfu_ctrl.vhd
12/05/2004  14:52             4.442 up8088_biu_interf_dfu_reg.vhd
21/04/2004  14:25             7.361 up808x_biu_interf_gau.vhd
21/04/2004  14:25             4.041 up808x_biu_interf_gau_add.vhd
21/04/2004  14:25            16.705 up808x_biu_interf_gau_ctrl.vhd
21/04/2004  14:25             7.032 up8088_biu_interf_ifu.vhd
21/04/2004  14:25             8.697 up8088_biu_interf_ifu_ctrl.vhd
21/04/2004  14:25             6.180 up808x_biu_interf_ifu_flush.vhd
12/05/2004  14:52             3.730 up8088_biu_interf_ifu_pointer.vhd
21/04/2004  14:25            13.392 up808x_biu_interf_mu.vhd
21/04/2004  14:25             7.247 up8088_biu_timing.vhd
12/05/2004  14:52            22.358 up8088_biu_timing_bcg.vhd
21/04/2004  14:25             4.917 up808x_biu_timing_tsm.vhd
21/04/2004  14:25            25.054 up808x_eu.vhd
21/04/2004  14:25            15.706 up808x_eu_alu.vhd
21/04/2004  14:24            20.174 up808x_eu_decode_ctrl.vhd
21/04/2004  14:24            13.779 up808x_eu_decode_drom.vhd
21/04/2004  14:24             7.797 up808x_eu_exe.vhd
17/05/2004  18:29           284.862 up808x_eu_exe_cgrom.vhd
21/04/2004  14:24             5.727 up808x_eu_exe_pc.vhd
21/04/2004  14:25             7.094 up808x_eu_exe_spw.vhd
21/04/2004  14:25             6.602 up808x_init.vhd
21/04/2004  14:25             9.396 up808x_rgu.vhd
21/04/2004  14:25             5.887 up808x_sru.vhd
               30 archivos        588.874 bytes
                2 dirs   5.917.908.992 bytes libres
```

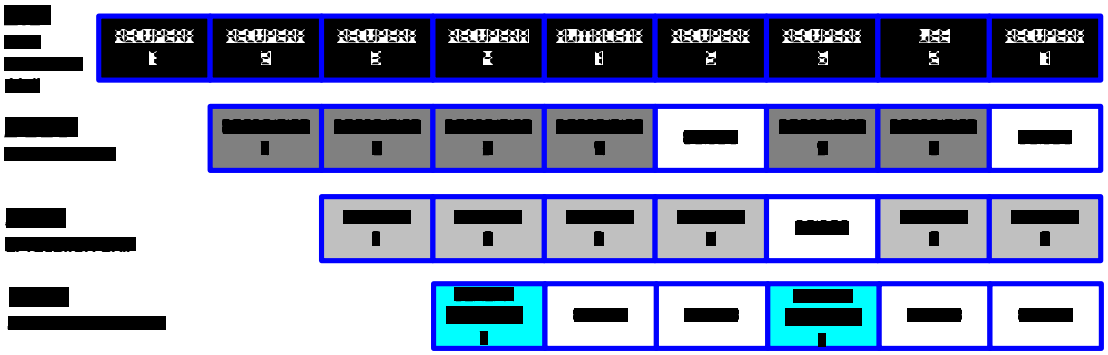


Fig 17.- Funcionamiento de la arquitectura paralela interna (pipeline) de modelo de las CPU 8088/8086

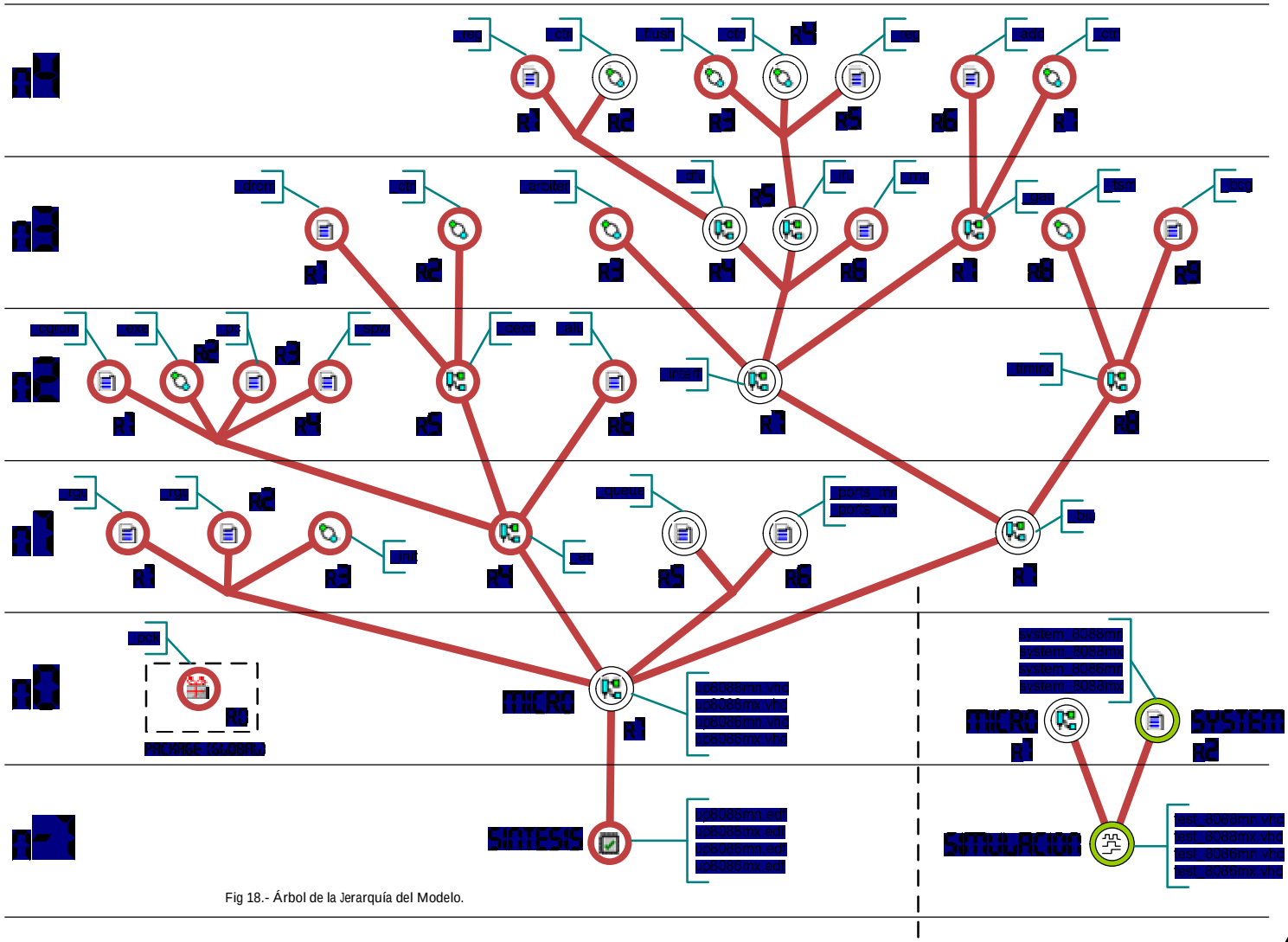


Fig 18.- Árbol de la Jerarquía del Modelo.

Icono	Significado	Icono	Significado
	Pertenece a la Librería Común, up808x.		Indica que el fichero contiene una descripción circuital.
			Indica que el fichero modela el comportamiento lógico de una unidad.
	Existe este componente en la librería del modelo 8088 (up8088) y en la del modelo 8086 (up8086).		El fichero define un grafo de estados.
			El archivo contiene un package (paquete de definiciones globales para todos los modelos).
	Estos componentes existen en la librería diseñada para la simulación de los modelos, up808x_tb. No sintetizable.		Se trata de un fichero resultado de la síntesis
			Son ficheros utilizados en el Banco de Test de los modelos.

Tab 7.- Significado de los Iconos.

dfu) que forma parte del interface (interf) de la unidad de interface con el bus (Bus Unit Interface, biu) en la librería del microprocesador 8086 (up8086)

Podemos organizar la jerarquía en nudos (o niveles de profundidad) y ramas (o elementos por nivel), así en el ejemplo anterior:

Nombre	up8086_biu_interf_dfu_ctrl.vhd				
Jerarquía	up8086	biu	interf	dfu	ctrl
Nudo/Rama	n0/r1	n1/r7	n2/r7	n3/r4	n4/r2
Unidad	Librería	Bus Unit Interface	Interface	Data Fetch Unit	Control Unit

Tab 6.- Ejemplo de Fichero.

Aquí (figura 18) se puede ver el esquema Jerárquico de la estructura del microprocesador.

En el se pueden distinguir los distintos nudos (o niveles) y las ramas (o elementos en cada nivel) de una forma gráfica. Además del nudo/rama, se pueden ver otros datos de interés:

- El tipo de fichero VHDL: es un icono que indica cómo está estructurado el fichero, es decir, si el código de éste define una estructura, grafo,

bloque... En la tabla 7 se puede ver el icono y su significado.

- Librería: el formato del circulo de cada rama, indica la librería a la que pertenece (ver la tabla 7). Es esencial tener una buena organización de los ficheros en un modelo tan complejo como éste, que no sólo incluye muchas unidades, sino que también incluye varios modelos y para más inrri, en varios modos de funcionamiento. Así, el modelo está dividido en cuatro Librerias (como se puede ver en la siguiente tabla), a grandes rasgos, la librería up808x, contiene los archivos que son comunes en todos los modelos, la librería up8088, contiene los ficheros que son utilizados para generar el modelo del micro 8088 en los modos máximo o mínimo.(up8088mn/up8088mx), la librería up8086 que contiene los ficheros necesarios para generar los modelos up8086mn/up8086mx, y por último la librería que contiene los ficheros (no sintetizables) para la simulación de todos los modelos.

En el desarrollo del proyecto iremos "podando" el árbol jerárquico y explicando en detalle cada una de las unidades, pero antes vamos a hacer una breve descripción de la función de cada unidad:

NIVEL 0:

- n0/r0, Package: Contiene la definición de tipos de datos, constantes... que son utilizadas de forma global por todo el sistema.
- n0/r1, Microprocesador: Librería up8088: (up8088mn,up8088mx); Librería up8086: (up8086mn,up8086mx), es el "tronco", modelo completo del microprocesador en sí. Existen varios modelos, aunque up8088mn y up8086mx parezcan microprocesadores distintos, en realidad son el mismo solo que con el nombre de los pines cambiado. Lo mismo podemos decir de los modelos up8086mn y up8086mx.
- n0/r2, Sistema: Librería up808x_tb (system_8088mn,...,system_8086mx): Esta rama no forma parte del código sintetizable, simplemente es utilizada para la simulación de los modelos. Todo esto se explica en detalle en el capítulo de simulación (ver el capítulo de simulación).

LIBRERÍA	FICHEROS	CONTENIDO
up808x (sintetizable)	*pack808x_pkg.vhd* *up808x_biu_interf_arbiter.vhd* *up808x_biu_interf_gau.vhd* *up808x_biu_interf_gau_add.vhd* *up808x_biu_interf_gau_ctrl.vhd* *up808x_biu_interf_ifu_flush.vhd* *up808x_biu_interf_mu.vhd* *up808x_biu_timing_tsm.vhd* *up808x_eu_alu.vhd* *up808x_eu.vhd* *up808x_eu_decode.vhd* *up808x_eu_decode_drom.vhd* *up808x_eu_exe.vhd* *up808x_eu_exe_cgrom.vhd* *up808x_eu_exe_pc.vhd* *up808x_eu_exe_spw.vhd* *up808x_init.vhd* *up808x_rgu.vhd* *up808x_sru.vhd*	Esta librería contiene las unidades comunes para los dos microprocesadores. En particular la Unidad de Ejecución (UE) y algunas partes de la Unidad de Interface con el Bus (BIU). Así como el modulo de inicialización (init), los registros de uso general (rgu), de segmento (sru) y el package (pkg) que contiene las declaraciones de tipos de datos utilizadas por todo el sistema. Se puede observar que UE es común para todos los micros, por lo que nosotros como hicieron los ingenieros de Intel ® después de haber diseñado el 8086, sólo tenemos que rediseñar una pequeña parete de la BIU para generar el nuevo microprocesador, el 8088. En la síntesis, utilizaré como micro base el 8088 en modo mínimo y asumiré resultados parecidos en la supuesta síntesis del micro 8086. Es lógico si tenemos en cuenta que la EU es la misma y ocupa el 80% del espacio en la FPGA y que la BIU es muy parecida en ambos micros (el 95% de la lógica se mantiene).
up8088 (sintetizable)	*up8088mn.vhd* *up8088mx.vhd* *up8088_biu.vhd* *up8088_biu_interf.vhd* *up8088_biu_interf_dfu.vhd* *up8088_biu_interf_dfu_ctrl.vhd* *up8088_biu_interf_dfu_reg.vhd* *up8088_biu_interf_ifu.vhd* *up8088_biu_interf_ifu_ctrl.vhd* *up8088_biu_interf_ifu_pointer.vhd* *up8088_biu_timing.vhd* *up8088_biu_timing_bcg.vhd* *up8088_ports_mn.vhd* *up8088_ports_mx.vhd* *up8088_queue.vhd*	Contiene los módulos específicos para el microprocesador 8088, es decir, aquellos que sufren alguna pequeña variación respecto a los utilizados en el 8086. También contienen las carcasas del microprocesador en modo máximo y mínimo up8088mn y up8088mx. Entre estos dos microprocesadores no hay ninguna diferencia salvo el nombre de los puertos que cambia dependiendo del valor del pin MN_MXn, de ahí que sea necesario cambiar el nombre de los pines a la carcasa del microprocesador y a sus respectivos puertos de salida (ports_mn y ports_mx). Esto último atiende, no sólo a cuestiones estéticas, también hace el código un poco más legible; se podría haber utilizado una sola carcasa para los dos modos, pero aparecerían señales con nombres como RQ1nGT1n_HOLDA, cuando lo cierto es que sólo estará activada una de las dos (RQ1nGT1n ó HOLDA) en un sistema real.
up8086 (sintetizable)	*up8086mn.vhd* *up8086mx.vhd* *up8086_biu.vhd* *up8086_biu_interf.vhd* *up8086_biu_interf_dfu.vhd* *up8086_biu_interf_dfu_ctrl.vhd* *up8086_biu_interf_dfu_reg.vhd* *up8086_biu_interf_ifu.vhd* *up8086_biu_interf_ifu_ctrl.vhd* *up8086_biu_interf_ifu_pointer.vhd* *up8086_biu_timing.vhd* *up8086_biu_timing_bcg.vhd* *up8086_ports_mn.vhd* *up8086_ports_mx.vhd* *up8086_queue.vhd*	Contiene los módulos sintetizables utilizados exclusivamente para el modelo de microprocesador 8086 en los modos máximo (up8086mx) y mínimo (up8086mn). Esta librería contiene casi en exclusiva la Unidad de Interface con el Bus (BIU) y aunque casi en su 95% es la misma que la del microprocesador 8088 (librería up8088), es necesario separarla para la síntesis ya que el nombre y la lógica de algún puerto cambia.
up808x_tb (no sintetizable)	*system_8086mn.vhd* *system_8086mx.vhd* *system_8088mn.vhd* *system_8088mx.vhd* *test_8086mn.vhd* *test_8086mx.vhd* *test_8088mn.vhd* *test_8088mx.vhd*	Contiene los ficheros necesarios para el Banco de Test (Test Bench, "_tb") de los micros 8088/8086 en los modos MX/MN. Estos módulos no son sintetizables ya que son utilizados solamente para la generación automática de los vectores de test, de ello se encargan los sistemas emulados para cada micro en cada modo, así system_8088mn genera los vectores para el test del microprocesador 8088 en modo mínimo. La prueba se realiza en el banco de test (test_8088mn) que aplica al microprocesador up8088mn los vectores generados por system_8088mn. Así con todos los microprocesadores.

Tab 8.- Contenido y Función de las Librerías del Proyecto.

NIVEL 1:

- n1/r1: Registros de Uso General, Register General Use (RGU): Librería up808x (_rgu): Son los registros de uso general, forman parte del modelo software (ver modelo software del microprocesador) de los micros, lo que significa que son transparentes para el usuario, por lo que éste puede verlos, modificarlos u operar con ellos. Sin entrar en detalles, podemos decir que son 8 registros de 16 bits (AX, CX, DX, BX, SP, BP, SI y DI), a 4 de ellos se puede acceder, independiente-mente, al byte alto (Most Significant Byte, MSB) o al byte bajo (Less Significant Byte, LSB), los registros en cuestión son AX (AH y AL), BX (BH y BL), CX (CH y CL) y DX (DH y DL). Son registros de acceso rápido y la unidad de ejecución los utiliza de forma implícita en algunas instrucciones, por lo que estos registros adquieren una función específica e implícita al ejecutar dicha instrucción.

- n1/r2, Registros de Segmento, Segment Register Unit (SRU): Son 8 Registros de 16 bits cada uno, 5 de ellos son transparentes para el usuario y forman parte del modelo software (ver modelo software), 4 de estos registros transparentes, son los registros de segmento CS, DS, SS y ES, el otro registro visto desde el exterior, es el puntero de instrucciones IP. Los 3 registros restantes son registros auxiliares de uso interno, (A0,A1 y A2). Estos registros son utilizados por la unidad de generación de direcciones (Gen Address Unit, GAU n3/r7) para que dependiendo del modo de direccionamiento, generar la dirección física (Phisical Address, PA).

- n1/r3, Unidad de Inicialización (init): Es la unidad encargada de inicializar el sistema cuando se conecta por primera vez el micro a la alimentación o cuando aparece la señal de RESET. También se encarga de separar la señal de reloj CK en dos fases no solapadas, ph1 y ph2.

- n1/r5, Cola de Instrucciones (queue): Es una memoria Tipo FIFO cuya función es memorizar las siguientes instrucciones en ser ejecutadas. Con esta estrategia conseguimos mejorar los tiempos de ejecución ya que el microprocesador no tiene que acceder a la memoria en busca de la instrucción, sino que la recoge directamente de la cola de instrucciones que es una memoria interna y por lo tanto mucho más rápida (concepto de memoria caché).

- n1/r6, Puertos (ports): Ésta unidad se encarga de multiplexar, en función de la entrada MN/MXn (modo de funcionamiento), los terminales del microprocesador.

- n1/r4, Unidad de Ejecución (Execution Unit, EU): La unidad de ejecución es la encargada, como su nombre indica, de ejecutar las instrucciones. Es idéntica en los micros 8088/8086. Las instrucciones de la cola, son decodificadas por la unidad de decodificación (deco n2/r5), los datos decodificados son utilizados, junto con el estado de la unidad de ejecución (exe n2/r2) y la fase de ejecución o contador de programa, Program Counter (pc n2/r3), como vector de entrada en una Code Generator ROM (cgrom n2/r1) que programa todo el sistema. La CGROM contiene la secuencia de pasos (o microinstrucciones) necesarios para llevar a cavo la lógica (de la instrucción en ejecución) completamente. La unidad lógico aritmética (Arithmetic Logic Unit, alu n2/r6) se encarga de realizar las operaciones de origen lógico-aritmético indicadas por la microinstrucción en curso.

- n1/r7, Unidad de Interface con el Bus (Bus Interface Unit, BIU): La BIU se encarga de establecer una comunicación (interface) bidireccional entre el bus loca (bus externo) y la unidad de ejecución interna. Internamente la BIU está dividida en dos bloques funcionales, (1) la unidad de sincronismo del bus (bus_timing n2/r8), que trata aspectos relacionados con la temporización del bus: generación del estado de bus, generación de las señales de los distintos tipos de ciclos de bus, generación de las señales del bus de control.... Y (2) la unidad de comunicación con la memoria (mem_interface n2/r7) dedicada a aspectos de comunicación con el bus como el arbitraje interno del canal, la atención de peticiones del bus local, la generación de la direcci-ón física...

NIVEL 2

- n2/r1 Code Generator ROM (CGROM): como se dijo anteriormente, contiene la lógica combinacional que, en función de los datos suministrados por el decodificador, la unidad de ejecución y el contador de programa, genera las señales que programan las unidades internas para realizar una determinada operación. Una secuencia consecutiva de estas operaciones intermedias (o microinstrucciones) culmina con la ejecución de la instrucción decodificada.

- n2/r2 Máquina de Ejecución (exe): Es una máquina de estados que determina cómo se encuentra la unidad de ejecución. Más adelante (ver la explicación de la unidad exe) se explica detalladamente el funcionamiento de esta unidad que se puede decir que es la que "comanda" la unidad de ejecución EU.

- n2/r3 Contador de Programa, Program Counter (PC): Esta unidad incrementa el vector de entrada a la CGROM que se encarga de saltar a la siguiente microinstrucción dentro de la secuencia de la instrucción en ejecución.

- n2/r4 Registros Especiales, Special Words (SPW): Contiene registros con un significado "especial" o concreto en la lógica de ejecución de UE. Uno de estos registros es el famoso registro de banderas (que forma parte del modelo software ver modelo software). También incluye una serie de registros que dan soporte a la lógica de interrupciones.

- n2/r5 Unidad de Decodificación, (deco): Encargada de interpretar el código de operación OP y proporcionarle la información a la CGROM. La decodificación lee los datos directamente de la cola de instrucciones (n1/r3), decodifica el formato de la instrucción en cuestión y accede a los bytes siguientes si éstos existen.

- n2/r6 Unidad Lógico Aritmética, Arithmetic Logic Unit (ALU): Contiene la lógica combinacional necesaria para realizar operaciones lógico y aritméticas con datos de 8 o 16 bits. Es utilizada

exclusivamente por EU y programada desde la CGROM.

- n2/r7 Unidad de Interface con la Memoria (mem_interface): Encargada de las operaciones relacionadas con la transferencia a memoria. Desde el código de microinstrucción de una instrucción que realiza transferencias, se programa el comportamiento de esta unidad que normalmente trabaja en paralelo con la ejecución de la instrucción accediendo a memoria en busca de instrucciones.
- n2/r8 Unidad de Sincronización con el Bus (bus_timing): Genera las señales de los ciclos de bus y el estado de éste. Los ciclos de bus son utilizados como mecanismo de comunicación con dispositivos externos (Memoria, I/O...) existiendo un ciclo de bus específico para cada operación que el micro realice con el bus. Las señales del Bus de Control son, en su mayoría, generadas por esta unidad.

NIVEL 3:

- n3/r1 ROM del Decodificador, Decode ROM (DROM): Esta unidad contiene la lógica combinacional que utiliza el decodificador para interpretar el OP. En el apéndice A se puede observar el formato de los OP Code.
- n3/r2 Control del Decodificador: (deco_ctrl): Es el corazón del decodificador, contiene el grafo de estados con las transiciones necesarias para decodificar toda la información.
- n3/r3 Árbitro de canal (arbiter): Internamente existen varias unidades que acceden al bus local, por lo qu es necesario un árbitro interno que establezca las prioridades y las condiciones de acceso. De ello se encarga esta unidad.
- n3/r4 Unidad de Acceso a Datos, Data Fetch Unit (DFU): Se encarga de las operaciones de transferncia a memoria o dispositivos de entrada salida, así como de la petición de los ciclos de reconocimiento de interrupción.
- n3/r5 Unidad que va en Búsqueda de Instrucciones, Instruction Fetch Unit (IFU): Encargada de realizar los ciclos que llenan la cola de instrucciones. Siempre que no exista una petición de ciclo de bus específico por parte de la UE, IFU genera la petición al árbitro hasta que llena la cola de instrucciones. En estas condiciones, UE y BIU trabajan en paralelo.
- n3/r6 Unidad de Multibus (Multibus): Esta unidad es la encargada de dar soporte a las peticiones de bus local por parte de otros maestros de canal. La sincronización de la petición con el micro se hace a través del árbitro interno (n3/r3).
- n3/r7 Unidad de Generación de Direcciones, Gen Address Unit (GAU): Esta unidad se encarga de generar la dirección de 20 bits (Phisical Address, PA=Segmento&"0000"b+Desplazamiento) con la que IFU o DFU van a realizar la transferencia. La programación (tipo de direccionamiento, existe o no prefijo de segmento, cual es..) la realiza la unidad de ejecución UE.
- n3/r8 Estado del Bus Local, T State Machine (TSM): Es el grafo general del bus que controla en todo momento el estado del bus local.
- n3/r9 Generador de Ciclos de Bus, Bus Cycle Generator (BCG): Es la unidad encargada de generar las señales de control externas del ciclo de bus en ejecución, el ciclo de bus que determina la lógica de Arbiter (n3/r3).

NIVEL 4

- n4/r1 Registros de DFU (dfu_reg): Son los registros que utiliza DFU para comunicarse con el exterior y pasar información a la EU. También contiene el registro de 20 bits que memoriza PA.
- n4/r2 Grafo de estados de DFU (dfu_ctrl): Es el Automata de Meal que genera el estado de DFU.
- n4/r3 Unidad de Flush (ifu_flush): Cuando aparece un salto en el flujo del programa, es necesario sacar todos los bytes de la cola de instrucciones (flush) e inicializar IFU. Esta unidad se encaga de realizar correctamente estas operaciones.
- n4/r4 Lógica de Control de IFU (ifu_ctrl): Contiene el grafo de conducción de la unidad IFU, desde aquí se "lanza" el ciclo de bus que va en busca de instrucciones (instruction_fetch) para llenar la cola de instrucciones.
- n4/r5 Registros de la IFU (ifu_reg): Contiene el Puntero de la unidad IFU (ifu_pointer) que es a partir del cual se genera la dirección física para acceder a la siguiente instrucción a descargar de memoria. La dirección física (PA) también es memorizada en un registro de 20 bits interno.
- n4/r6 Sumador de 20 bits (gau_adder): Es el sumador utilizado por GAU (unidad encargada de generar la dirección física, n3/r7) para sumar el segmento (desplazado cuatro posiciones hacia la derecha) y el desplazamiento.
- n4/r7 Lógica de Control de GAU (gau_ctrl): Contiene la secuencia de pasos necesarios para generar la dirección física. La dirección física puede tardar en generarse, dependiendo del modo de direccionamiento, de 6 a 12 ciclos en el microprocesador original y de 2 a 4 en el modelo desarrollado.

7.2 SUBTIPOS DE DATOS DEFINIDOS EN PACKAGE

El fichero "pack808x_pck.vhd" de la librería común up808x contiene el "paquete" de declaraciones comunes para todos los ficheros "vhdl" del proyecto. En este sentido, he definido unos subtipos de datos que son utilizados en todos los ficheros "vhdl":

```
--|sUBTiPOS dE Std_Logic_Vector                          |--
--Vector de 4 bits
SUBTYPE Nibble_Vector  IS Std_Logic_Vector(3 DOWNTO 0);
--Vector de 8 bits
SUBTYPE Byte_Vector    IS Std_Logic_Vector(7 DOWNTO 0);
--Vector de 16 bits
SUBTYPE Word_Vector    IS Std_Logic_Vector(15 DOWNTO 0);
--Vector de 20 bits
SUBTYPE Address_Vector IS Std_Logic_Vector(19 DOWNTO 0);
--|.....................................................|--
```

Hay cuatro subtipos del tipo estándar Std_Logic_Vector, de manera que a partir de ahora para definir un vector de 8 bits podré utilizar el subtipo "Byte_Vector", o para definir una señal de 16 bits, "Word_Vector". A medida que en la explicación de los componentes vayan apareciendo declaraciones definidas en el package las iré comentando. De todas

maneras el código, tanto del package como del resto de los ficheros “vhdl” esta ampliamente comentado.

El package debe aparecer en la declaración de librerías en todos los ficheros que utilizan alguna de las definiciones establecidas en éste, así por ejemplo en el fichero “up808x_alu.vhd” aparece:

```
--Librerías Estandar
LIBRARY IEEE;
USE IEEE.Std_Logic_1164.ALL;
USE IEEE.Std_Logic_Arith.ALL;
USE IEEE.Std_Logic_Unsigned.ALL;
--Librería del Modelo
LIBRARY up808x;
USE up808x.pack808x.ALL;
```

7.3 NIVEL 0

El nivel 0 tiene 2 ramas o componentes, los microprocesadores en si (modelos sintetizables, n0/r1) y los emuladores de los sistema físicos (no sintetizables, n0/r2) utilizados en la simulación (se explica en detalle en el apartado de simulación Ver el capítulo de simulación..

También podemos considerar en el nivel n0/r0 un componente de índole global que contiene las declaraciones de tipos de datos, constantes, funciones... utilizados en todos los ficheros del modelo; estoy hablando del fichero “pack808x_pck.vhd” de la librería común up808x.

El nivel n0/r1 es el microprocesador. Existen dos componentes por librería en éste nudo: las carcas en modo máximo y mínimo del modelo 8088 (up8088mx, up8088mn) y las del modelo 8086 (up8086mx, up8086mn). En la figura 19, podemos ver las 4 carcasas y sus diferencias:

Existen 2 componentes por librería para un mismo modelo, la razón es que decidí crear un componenete distinto por cada modo de funcionamiento con la intención de hacer más didáctico y entendible el código.

El modo de funcionamiento, está directamente relacionado con el hardware externo del sistema en el que implementamos el modelo, y éste no cambia por lo que el micro implementado en ese sistema no cambiará su modo de funcionamiento y por lo tanto es inútili dar a un pin un nombre de una función que no utilizará nunca en un sistema real. Esta puede ser otra explicación de la decisión adoptada.

Podríamos haber utilizado una sola carcasa por modelo, pero como se indica en la tabla 8 hubieramos tenido pines con nombres como “RQ1nGT1n_HOLDA” cuando en un sistema real o bien está activada la lógica de RQ1nGT1n (MN_MXn=’0’) o HOLDA (MN_MXn=’1’).

Fig 19.- Carcasas de los Modelos.

Los ficheros del nivel n0/r1, definen una estructura o esquema en VHDL, lo que significa que interconectan componentes o bloques por medio de señales y en su código (por lo general) no se añade lógica. La lógica es programada dentro de las unidades definidas como modelos comportamentales en VHDL o como grafos de estados.

Eliminando lógica entre componentes (en esquemas diseñados a nivel de transferencia de registros, Register Transfer Level RTL) e insertandola en las propias unidades, obtenemos códigos más limpios.

En el siguiente nivel se puede ver la estructura definida por el nivel n0/r1.

A continuación podemos ver las tablas (tablas 9, 10 y 11) donde se describen todos los terminales de los microprocesadores y su funcionamiento. Como referencia incluyo el pin que ocupa en el microprocesador original (ver la disposición de los pines en la figura 9), si bien, no tienen poqrqué corresponder con los que asignemos a la FPGA.

También indico el tipo de puerto de cada terminal, Entrada (I), Salida (O), si dispone de buffer triestado (Z). También podemos observar otro tipo de información como si la entrada está internamente sincronizada, si la señal es activada por nivel o por flanco y si dispone de resistenca pull-up.

,

7.4- DESCRIPCIÓN DE LOS PINES

7.4.1 TERMINALES COMUNES: La siguiente tabla muestra los pines comunes en los modelos 8088/8086 en cualquiera de los modos de funcionamiento.

	SÍMBOLO	PIN (EN MICRO ORIGINAL)	TIPO	NOMBRE Y FUNCIÓN
8088	AD7-AD0 (AD)	9-16	I/O-Z	Bus de Datos-Direcciones, Address/Data (AD): Estas líneas forman el bus de datos (aparece en T2, T3, Tw y T4) y direcciones de memoria/IO (aparece en T1) multiplexado en el tiempo. Se activan a nivel alto y flotan en estado de alta impedancia durante el reconocimiento de una interrupción, durante la cesión del bus local "hold acknowledge" o RESET. En el i8086 la señal A0 realiza la misma función que BHEn pero con el byte bajo del bus de datos, los pines D7-D0. Durante T1 permanece a nivel bajo cuando se transfiere un byte por las 8 líneas menos significativas del bus de datos a la memoria o a un dispositivo de Esntrada/Salida. Para el correcto funcionamiento de dispositivos de 8 bits, se deben posicionar en la parte baja del bus de datos y utilizar A0 como señal de habilitación del dispositivo.
8086	AD15-AD0 (AD)	2-16,39		
8088	A15-A8 (A)	2-8,39	O-Z	Bus de Direcciones, Address (A): Contienen el segundo byte de la dirección (A15-A8) durante todos los estados de bus (T1-T4) por lo que no es necesario que sean enclavadas por la señal ALE. Se activan a nivel alto y flotan en estado de alta impedancia durante el reconocimiento de una interrupción, durante la cesión del bus local "hold acknowledge" o RESET.
8088/8086	A19/S6, A18/S5, A17/S4, A16/S3 (AS)	35-38	O-Z	Bus de Estado-Direcciones, Address/Satus (AS): Durante T1 y en operaciones con la memoria, estas líneas son la parte más significativa del bus de direcciones. En operaciones con I/O, estas señales permanecen a nivel bajo. En operaciones a memoria o I/O, podemos encontrar durante T2, T3, Tw y T4 la información relativa al estado. S6 se encuentra siempre a nivel bajo. El bit S5 indica el estado del bit de habilitación de interrupciones "Interrupt Flag" (IF), este bit es actualizado al comienzo de cada ciclo de reloj. En la tabla de abajo, podemos ver el significado de S4 y S3. Esta información indica cual es el registro de segmento que ha sido utilizado para generar la dirección que se encuentra en el bus de direcciones. Se activan a nivel alto y flotan en estado de alta impedancia durante la cesión del bus local "hold acknowledge" o RESET

S4	S3	Segmento Utilizado
0	0	Extra (Extra Segment, ES)
0	1	Pila (Stack Segment, SS)
1	0	Codigo o Ninguno (CS)
1	1	Datos (Data Segment, DS)
0 (LOW), 1 (HIGH), S6 siempre 0		

8086	BHEn/S7	34	O-Z	Habilitación de la pare alta del Bus/Estado, Bus High Enable/Status (BHEn/S7): BHEn suele ser activada durante T1 para habilitar el byte más significativo del bus de datos (pines D15-D8). Los dispositivos de 8 bits colocados en la parte alta del bus, deberán utilizar BHEn como habilitación (Chip Enable, CEn). BHEn es activa a nivel bajo durante T1 en ciclos de bus de escritura, lectura y reconocimiento de interrupción cuando se está transfiriendo un byte por la parte alta del bus. El bit de estado S7 aparece durante T2, T3 y T4. Esta señal permanece a nivel bajo durante T1 del primer ciclo de un reconocimiento de interrupción y flota en estado de alta impedancia durante la cesión del bus local "hold acknowledge" o RESET.

continua...

8088/8086	RDn	32	O-Z	Lectura, ReaD (RD): Esta señal indica que el microprocesador está realizando una operación de lectura en I/O o Memoria, dependiendo del valor de la señal IO/Mn o S2. La señal RDn es utilizada para leer dispositivos que se encuentran en el bus local. Aparece durante T2, T3 y Tw de cualquier ciclo de lectura una vez que el bus local a pasado a estado de alta impedancia. Esta señal se activa a nivel bajo y flota en estado de alta impedancia durante la cesión del bus local "hold acknowledge" o RESET.
8088/8086	READY	22	I-Sin Sincronizar	Preparado, READY: Es la señal que indica que el dispositivo direccionado (dispositivo de Memoria o I/O) ha completado la transferencia del dato. La señal RDY (señal asíncrona) desde la Memoria o I/O es sincronizada por el circuito integrado 8284 (generador de reloj) obteniendo la señal READY. Esta señal se activa a nivel alto. La entrada no está internamente sincronizadapor lo que los tiempos de preparación "set up" y de mantenimiento "hold" de la señal tienen que cumplirse. El generador de reloj i8284 se encarga de hacerlo.
8088/8086	INTR	18	I-Activada por nivel	Petición de Interrupción, INTeRupt rquest (INTR): Es una entrada registrada por nivel y muestreada en el último ciclo de ejecución de cada instrucción. Si está activa, el microprocesador atenderá la petición. En el servicio de la interrupción, se salta a una subrutina cuya dirección de entrada está especificada en una tabla de vectores de interrupción situada en memoria. La petición será ignorada si el bit de habilitación de interrupción (modificable por software) está a cero. Está internamente sincronizada y se activa a nivel alto.
8088/ 8086	TESTn	23	I-Sincronizada cada CK	TESTn: La instrucción WAIT examina esta patilla, si la entrada TESTn está a nivel bajo, la ejecución continua, de lo contrario, el microprocesador espera (Ti). Esta señal está sincronizada con el flanco de subida de CK.
8088/8086	NMI	17	I-Activada por Falnco de subida	Interrupción no enmascarable, Non-Maskable Interrupt (NMI): Esta entrada, activada por el flanco de subida, lanza una interrupción de tipo 2. El microprocesador salta a un subprograma (servicio de interrupción) cuya dirección de entrada (vector de interrupción) se encuentra en una tabla situada en memoria (tabla de vectores de interrupción). Esta interrupción no puede ser ignorada por software. Una transición del nivel bajo al alto, genera una petición que será atendida al final de la ejecución de la instrucción actual. La entrada está internamente sincronizada.
8088/8086	RESET	21	I-Sincronizada	RESET: Provoca la inmediata inicialización del sistema. Esta señal tiene que estar, por lo menos, 4 ciclos de reloj a nivel alto para ser aceptada. La inicialización del sistema se produce cuando la señal RESET pasa a cero. La entrada está internamente sincronizada.
8088/8086	CLK	19	I-Clock	Señal de Reloj, ClocK (CK): Es la señal utilizada para la sincronización básica del procesador y del controlador de bus. Es una señal asimétrica con un ciclo de trabajo del 33% lo que permite tener una fase larga para las partes del micro en las que hay mucha lógica intermedia.
8088/8086	Vcc	40	Power	Vcc: Es el terminal de alimentación, su tensión es de +5V (+/-)10%
	GND	1, 20	Power	GND: Son los pines de masa.
	MN/MXn	33	I-No Sincronizada, Resistencia Pull-up	Selector de Modo, MiNimun/MaXimun (MN/MXn): Indica el modo de funcionamiento del microprocesador.

Tab 9.- Terminales Comunes en los Modelos para los Modos de Funcionamiento Máximo y Mínimo.

7.4.2 TERMINALES EN MODO MÍNIMO. El funcionamiento en modo mínimo de los 8088 y 8086, se obtiene al conectar el terminal MN/MXn directamente a Vcc.

	SÍMBOLO	PIN (EN MICRO ORIGINAL)	TIPO	NOMBRE Y FUNCIÓN
8088	IO/Mn	28	O-Z	Línea de estado: Esta señal es equivalente a S2n en modo máximo solo que invertida. Su función es distinguir entre un acceso a Memoria (IO/Mn='0'en el8088 o M/IOn='1' en el 8086) o a un dispositivo I/O (IO/Mn='1' en el 8088 o M/IOn='0' en el 8086). Aparece en el estado T4 que precede a un ciclo de bus y permanece válido hasta el final (T4) de este ciclo. Esta señal flota en estado de alta impedancia durante la cesión del bus local "hold acknowledge" o RESET.
8086	M/IOn			
8088/8086	WRn	29	O-Z	Escritura, WRite (WRn): Indica que se está escribiendo en la Memoria o en un dispositivo de I/O dependiendo del valor de la señal IO/Mn, en el 8088 o M/IOn en el 8086. WR aparece en T2, T3 y Tw en cualquier ciclo de escritura. Se activa a nivel bajo y flota en estado de alta impedancia durante la cesión del bus local "hold acknowledge" o RESET.
	INTAn	24	O	Reconocimiento de interrupción, INTerrupt Acknowledge (INTAn): Es utilizada como señal de habilitación de lectura en el ciclo de reconocimiento de interrupción. Se activa a nivel bajo durante los estados T2, T3 y Tw del ciclo de reconocimiento de interrupción.
	ALE	25	O	Habilitación de la dirección, Address Latch Enable (ALE): El micro genera esta señal para enclavar la dirección. Se activa a nivel alto y aparece durante la fase uno (ph1 es cuando CK='0') de T1 de cualquier ciclo de bus. Este puerto no tiene buffer triestado.
	DT/Rn	27	O-Z	Transmisión o Recepción de datos, Data Transmit/ Recive (DT/Rn): Utilizada para activar el tranciver del bus de datos en un sistema en modo mínimo. Controla el sentido de los datos en las transmisiones. Es equivalente a la señal S1n existente en modo máximo y aparece en los mismos intervalos de tiempo que la señal IO/Mn. Flota en estado de alta impedancia durante la cesión del bus local "hold acknowledge" o RESET. Junto con SS0n y DT/Rn, permite saber el ciclo de bus actual:

IO/Mn	DT/Rn	SS0n	Ciclo de Bus
1	0	0	Reconocimiento de Interrupción
1	0	1	Lectura de un Puerto I/O
1	1	0	Escritura de un Puerto I/O
1	1	1	Halt
0	0	0	Acceso a Código
0	0	1	Lectura de Memoria
0	1	0	Escritura de Memoria
0	1	1	Pasivo

8088/8086	DENn	26	O-Z	Habilitación de datos, Data ENable (DENn): Habilita la transferencia de datos a través del "transciver" del bus de datos en un sistema en modo mínimo. DENn es activada a nivel bajo durante las transferencias a Memoria o I/O o en ciclos INTAn. Para una lectura o un ciclo INTAn, aparece en la fase 2 (ph2, fase en la que CK='1') hasta el ph2 de T4, mientras que en ciclos de escritura aparece al comienzo de T2 permaneciendo hasta ph2 de T4. Flota en estado de alta impedancia durante la cesión del bus local "hold acknowledge" o RESET.
8088/8086	HOLD, HLDA	31,30	I,O-Sin sincronizar y con resistencia pull-up	HOLD indica que otro maestro del bus local necesita el bus, por lo que el micro debe dejarlo libre "hold". Para que HOLD sea reconocida debe permanecer a nivel alto, el micro actúa en consecuencia activando la señal de reconocimiento HLDA (activada a nivel alto) que aparece en ph2 de T4 o Ti. A la vez que el microprocesador genera HLDA, pone en estado de alta impedancia el bus local. Cuando la señal HOLD pasa a cero (el maestro de bus ya no necesita el bus local), el microprocesador desactiva HLDA y recupera el control sobre el bus local y las señales de control. Los terminales de HOLD y HLDA están conectadas internamente a Vcc a través de resistencias pull-up. La estrada de HOLD no está sincronizada internamente, por lo que es necesario garantizar los tiempos de "set-up" y mantenimiento "hold" externamente.
8088	SS0n	34	O-Z	Línea de Estado equivalente a S0n en modo máximo. Junto con IO/Mn y DT/Rn, permite saber el ciclo de bus actual.

Tab 10.- Terminales Existentes en Solamente en el Modo de funcionamiento Mínimo.

7.4.3 TERMINALES EN MODO MÁXIMO: A continuación podemos ver la descripción de los terminales para un sistema en modo máximo (MN/MXn=GND).

	SÍMBOLO	PIN (EN MICRO ORIGINAL)	TIPO	NOMBRE Y FUNCIÓN
8088/8086	S2n, S1n, S0n	26-28	O-Z	Estado, aparece en la fase ph2 de T4 y se mantiene durante T1 y T2. Retorna al estado pasivo (S2n, S1n y S0n a '1', '1' y '1' respectivamente) durante T3 o en Tw cuando READY está a nivel alto. Este estado es utilizado por el controlador de bus i8288 para generar todas las señales de control de Memoria o I/O.. Cualquier cambio en estas señales durante T4 , indica el comienzo de un ciclo de bus, y la transición al estado pasivo en T3 o Tw indica el final del mismo. Estas señales flotan en estado de alta impedancia durante el reconocimiento de "hold". En el primer ciclo de reloj después de RESET, estas señales indican el estado pasivo (todas a nivel alto), al ciclo siguiente pasan a estado de alta impedancia. S2n S1n S0n Ciclo de bus 0 0 0 Reconocimiento de Interrupción 0 0 1 Lectura de I/O 0 1 0 Escritura de I/O 0 1 1 Halt 1 0 0 Accedo a Código 1 0 1 Lectura de Memoria 1 1 0 Escritura de Memoria 1 1 1 Pasivo
	RQn/GT0n, RQn/GT1n	30, 31	I/O-Sin Sincronizar y con resistencia pull-pu	Petición/Cesión, RQuest/GranT (RQnGT?n): Estos terminales son utilizados por otros maestros de canal para forzar al microprocesador a dejar libre el bus al final del ciclo de bus actual. Son pines bidireccionales teniendo RQn/GT0n mayor prioridad que RQn/GT1n. RQn/GTn está conectado internamente a Vcc a través de una resistencia "pull-up" por lo que nunca adquiere un valor indeterminado. El cronograma de la secuencia request/grant se comenta más adelante en esta memoria [insertar referencia a la secuencia RQ/GT]: 1. Un pulso (pulso 1) de un ciclo de reloj de otro maestro de canal indica que otro dispositivo requiere el bus local para realizar alguna operación. 2. Durante T4 o Ti, el i8088 manda un pulso (pulso 2) de un ciclo de bus al maestro de canal que hizo la petición indicándole que ha puesto el dus local en estado de alta impedancia y que al siguiente ciclo de reloj entrará en el estado de reconocimiento de hold "hold acknowledge". La unidad de comunicación con el bus (BIU) permanece desconectada a nivel lógico del bus local durante "hold acknowledge". 3. Un tercer pulso de un ciclo de duración, indica al i8088 que la operación con el bus va a terminar y que puede recuperar el control sobre el bus local en el siguiente ciclo de reloj. El microprocesador entra en T4. Cada intercambio del control del bus local entre maestro-maestro de bus es una secuencia de tres pulsos. Debe haber un estado ocioso después de cada intercambio de control del bus local. Los pulsos se activan a nivel bajo. Si aparece una petición de bus mientras le micro está realizando una transferencia, cederá el bus en T4 después del ciclo actual, si se cumplen las siguientes condiciones: 1. La petición aparece antes o durante T2. 2. El ciclo de bus actual, no está transfiriendo el primer byte de una transferencia de 2 bytes. 3. El ciclo actual no es el primero de un ciclo de reconocimiento de interrupción. 4. No se está ejecutando una instrucción de bloqueo (prefijo LOCK). Si el bus local está ocioso cuando aparece una petición, pueden ocurrir dos cosas: 1. El bus local será cedido durante el siguiente ciclo de reloj. 2. Un ciclo de memoria dará comienzo dentro de 3 ciclos de reloj. Luego, se siguen las 4 reglas anteriores y la primera se cumple siempre..
	LOCKn	29	O-Z	Bloqueo, LOCKn: Mientras está activa (nivel bajo) el micro indica que otro maestro de bus no puede ganar el control del bus local. El prefijo del juego de instrucciones (LOCK) activa esta señal hasta que se completa la ejecución de la instrucción que le precede. El terminal flota en estado de alta impedancia durante el ciclo de reconocimiento de petición de bus local "hold acknowledge".
	QS1, QS0	24,25	O	Estado de la Cola de Instrucciones, Queue Status (QS): Proporcionan el estado interno de la cola de instrucciones, lo que permite realizar un seguimiento externo de la misma. Estas líneas son válidas durante el ciclo de reloj siguienete a la realización de la operación en cola: QS1 QS0 Operación 0 0 Sin Operación 0 1 Primer Byte de un Códogo de Operación. 1 0 Cola de Instrucciones Borrada. 1 1 Siguiente Byte de un Código.
8088	-	34	O	El Pin 34 está siempre a nivel alto en el modo máximo.

Tab 11.- Terminales Existentes en Solamente en el Modo de funcionamiento Máximo.

7.5 NIVEL 1

El nivel 1 está compuesto por seis componentes (ver la figura 19 y 20), dos de ellos, (BIU y EU) están ampliamente jerarquizados lo que implica una mayor estructuración de su lógica de lo que se puede intuir su mayor grado de complejidad. El resto de los componentes de este nivel (rgu, sru, init, queue y ports) no tienen más profundidad.

Se puede decir que este es el esquema general del microprocesador en el que se pueden distinguir los componentes que forman los bloques funcionales establecidos en el diagrama de bloques general (ver esquema de bloques general)

Existen cuatro esquemas correspondientes a los cuatro ficheros RTL definidos en el nivel anterior (nivel padre o troncal, n0/r1), así "up8088mn.vhd" define el esquema que podemos ver en la figura 21.

El esquema es descriptivo de la estructura interna, de ahí que haya sacado de EU la ALU (en realidad la alu está implementada en el nivel n2/r6), de esta manera podemos ver a simple vista los componentes principales del modelo. La línea discontinua indica que no existe en realidad en este nivel.

En los esquemas podemos ver que los componentes están identificados por 3 campos, el primero es el nombre de la librería donde se encuentra, el segundo es el nombre del fichrero vhdl que define el componente y el tercero, es el nombre asignado al componente.

Las líneas de conexión están a dos grosores indicando el número de líneas que contiene cada señal, así la fina indica que se trata de una señal de un bit mientras que las gordas nos indican que la señal está compuesta de dos o mas bits.

A simple vista puede verse la estructura Von Newman del microprocesador. Puede verse que no existe un bus interno (triestado) común para la comunicación entre las unidades, sino que estas se comunican por buses independientes. En el micro original existian 3 buses internos (Ver figura 1 DS8088) dos de ellos (C-BUS y A-BUS) triestado. La eliminación de estos buses en el diseño confiere muchas ventajas al modelo: por un lado aumenta la estabilidad al eliminar transiciones "pesadas" en los buses internos, disminuye el consumo y por lo tanto el calentamiento que pueden provocan los buses en estado de alta impedanzia; por otro lado realizamos las operaciones más rápido, en un ciclo de reloj podemos enviar a la alu los dos operandos (por los buses bus_a y bus_b) y recoger el resultado (en el bus result). Mientras que en el micro original para realizar una operación con la alu, primero hay que enviar un operando por A-BUS, registrarlo y al siguiente ciclo de reloj mandar el segundo operando por el A-BUS) finalmente en un tercer ciclo de reloj, recoger el resultado en A-BUS. Esto podría ir perfectamente en las ventajas del microprocesdor. El problema de la estructura Von Newman utilizada en el modelo es que aumenta considerablemente el número de líneas; este problema, no es tal si consideramos que la tecnología acual de las FPGA's trabaja perfectamente con diseños con muchas señales.

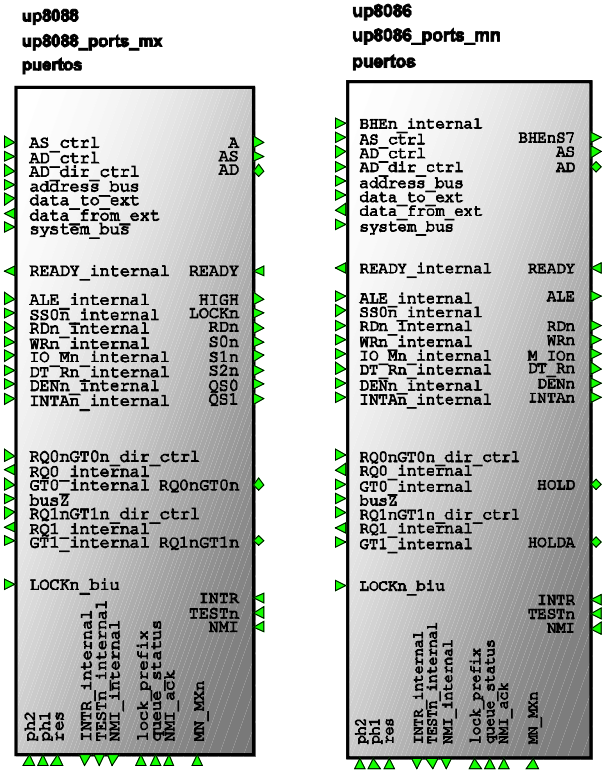


Fig 23.- Pueros para los Esquemas up8088mx y up8086mn.

DIFERENCIAS ENTRE UP8088MN-UP8088MX

La única diferencia entre el esquema up8088mn (modelo 8088 en modo Mínimo, figura 21) y up8088mx (modelo 8088 en modo Mínimo) es la unidad de puertos. Mientras que en el up8088mn utilizamos el fichero "up8088_ports_mn.vhd" en el up8088mx utilizamos el "up8088_ports_mn.vhd". La lógica de estos ficheros es la misma, lo único que cambia es el nombre de los puertos.

Cambiando el ports_mn por ports_mx (ver figura 21) obenemos el esquema del modelo 8088 en modo Máximo.

DIFERENCIAS ENTRE UP8086MN-UP8086MX

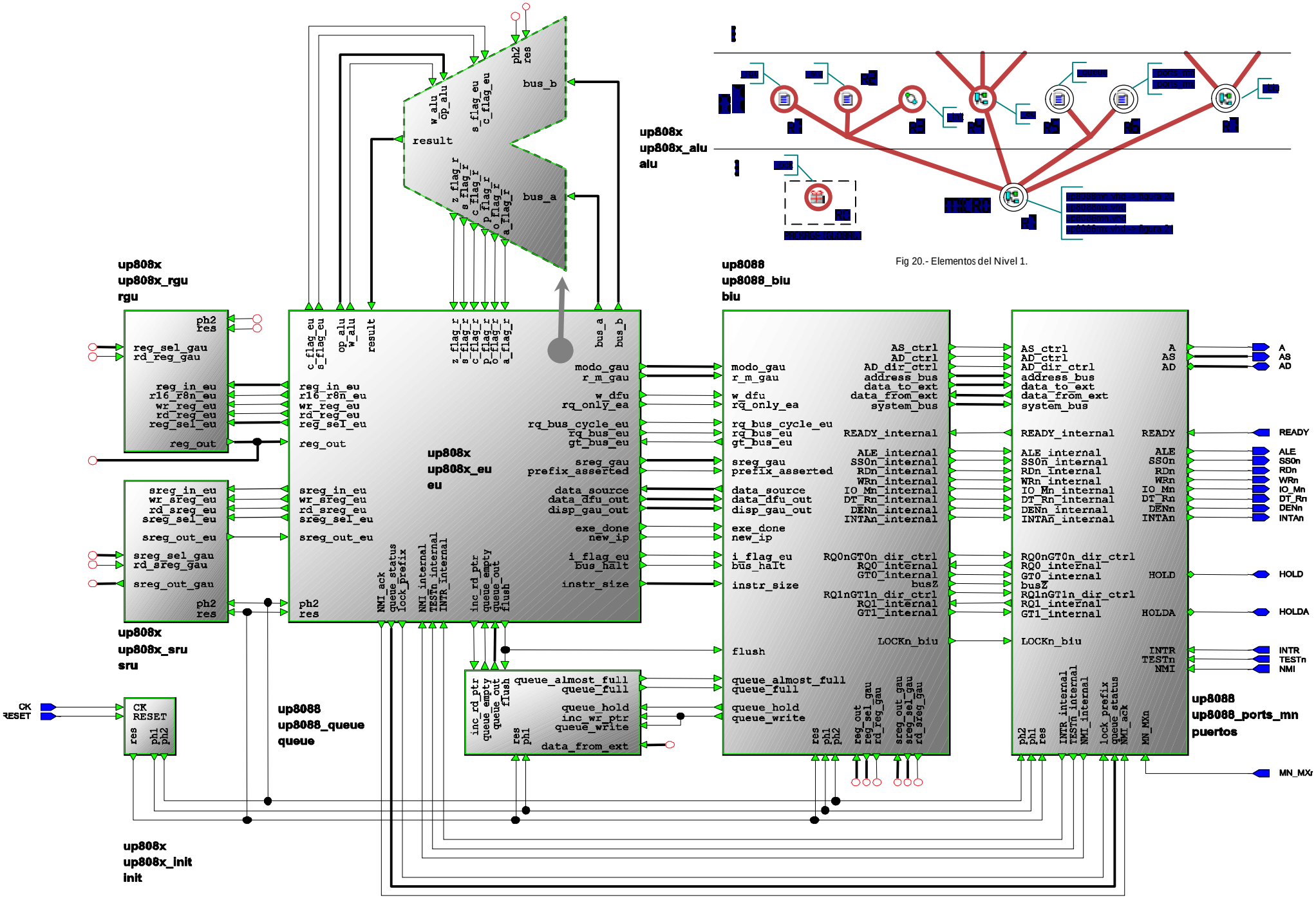
Las diferncias entre el up8086mn y el up8086mx, como en el caso anterior, es la unidad de puertos, de manera que tenemos la unidad "up8086_ports_mn.vhd" para el funcionamiento en modo mínimo y "up8086_ports_mx.vhd" para el modo máximo. Internamente no hay ninguna diferncia lógica entre los dos componentes, lo único que cambia es el nombre de los pines.

Cambiando el ports_mx por ports_mn (ver figura 22) obenemos el esquema del modelo 8086 en modo Mínimo.

DIFERENCIAS ENTRE UP8088MN/UP8088MX Y UP8086MN/UP8086MX

Entre los esquemas up8086?? (figura 22) y up8088?? (figura 21) hay varias diferencias; para empezar en queue, biu y puertos hay algunos cambios por lo que están definidos en dos librerías distintas, la up8088 (para los componentes del 8088) y la up8086 (para los componentes del modelo 8086). Aparece alguna señal nueva en el up8086?? como puede ser "w_queue" y "BHEn_internal", estas señales no son necesarias en el up8088??.

Fig 20.- Elementos del Nivel 1.



21.- MODELO 8088 EN MODO MÍNIMO. Esquema Definido en n0/r1, arcivo “up8088mn.vhd”

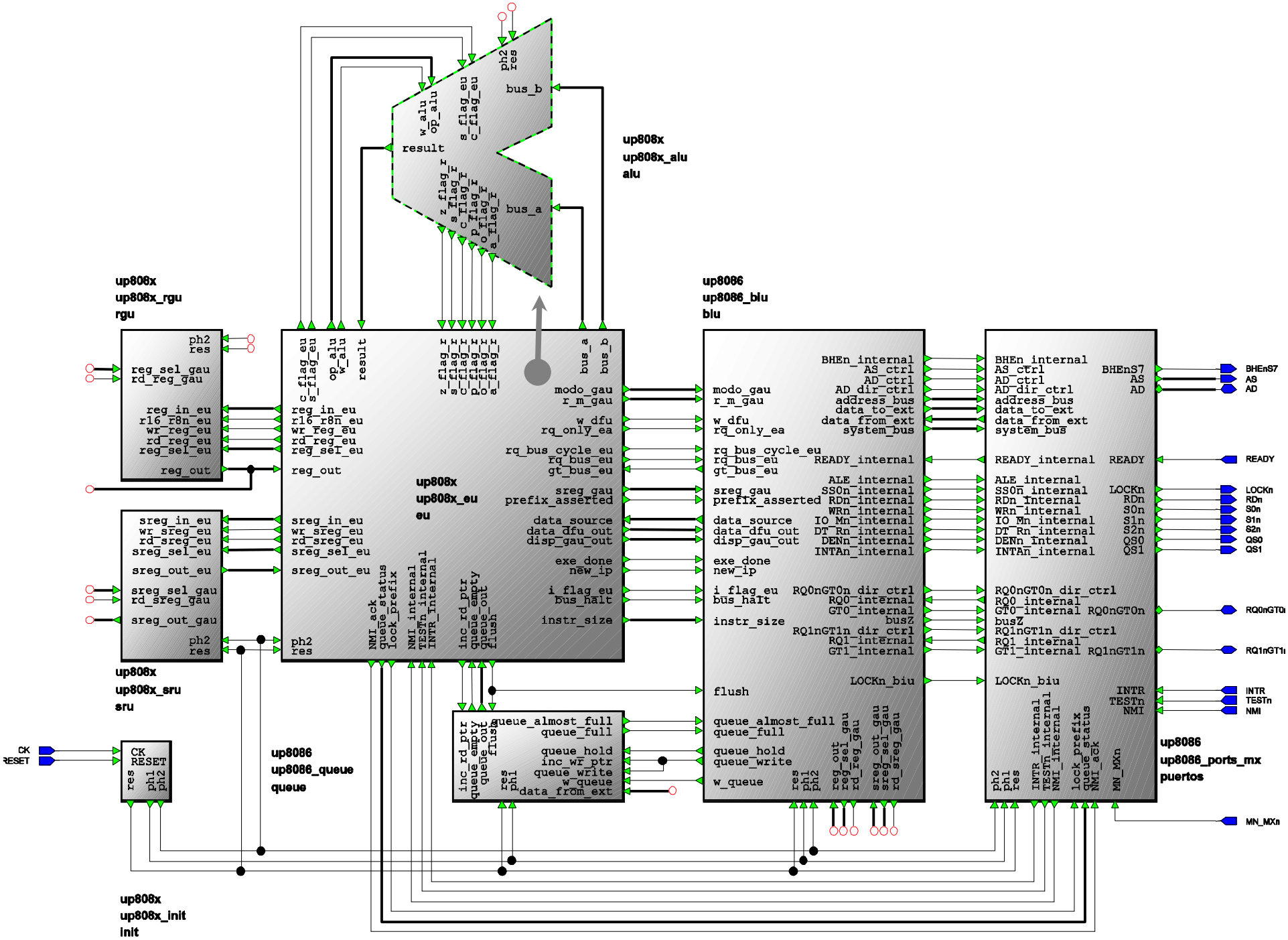


Fig 22.- MODELO 8086 EN MODO MÁXIMO. Esquema Definido en n0/r1, arcivo “up8086mx.vhd”

componentes de ámbito común

8. EXPLICACIÓN GENERAL DE LOS GRAFOS DE ESTADO:

En el desarrollo VHDL de algunas unidades he utilizado grafos de estado que, de una forma gráfica, permiten ver el comportamiento funcional de la unidad. A continución podremos ver el formato y significado de los elementos empleados en el grafo y el código en VHDL que lo modela.

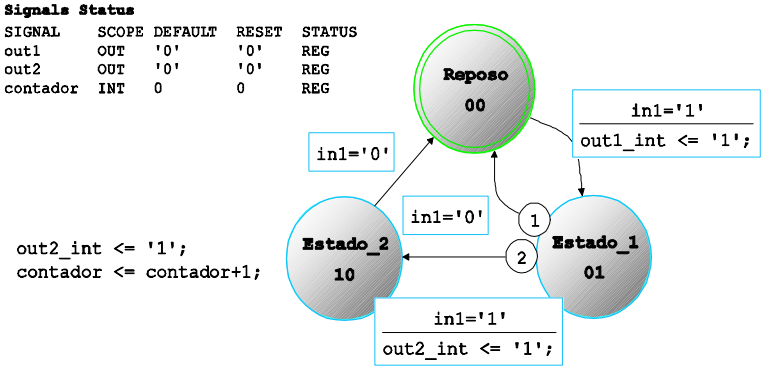


Fig 27.- Grafo de Estados de Ejemplo.

8.1 ESTADOS:

Nomenclatura: A cada uno de los estados y macroestados les asignaré un nombre descriptivo acorde con la situación funcional en la que se encuentra la unidad. Intrínsecamente y aunque no aparezca en el grafo, asignaré a cada estado un nombre que comience por "e" seguido de un número (e2, e5...). Utilizaré las letras "t" y "a" para hacer referencia a las transiciones y acciones respectivamente. Así, por ejemplo, cuando veamos e2/t1, estamos hablando de la transición 1 del estado 2; si vemos e3/t1/a, se trata de la acción de la transición 1 del estado 3. Recordar que pueden aparecer acciones en los estados y en las transiciones, en el caso de e2/a se trata de las acciones del estado 2.

Fig 28.- Formato del Estado Inicial, Estado y Macroestado.

El nombre funcional asignado a cada estado aparece en el interior del estado al que hace referencia (en el ejemplo, Reposo, Estado_1...). Si hemos asignado un código a cada estado, éste aparece justo debajo del nombre. Si no asignamos ningún código a los estados, el sintetizador se encarga de hacerlo según le convenga.

Las acciones de los estados aparecen al lado del estado. En la mayoría de los grafos, cada estado realiza muchas asignaciones por lo que resulta difícil componer una pantalla en la que aparezca tanta información; en estos casos no se visualizan las acciones (pero existen)

- Estados, ver la figura 27. El estado es cada una de las situaciones en las que se puede encontrar la unidad que estamos diseñando.
- Macroestado. Se distingue por tener dos circulos internos. El macroestado aglutina una serie de estados. Este macroestado no es codificado, simplemente sirve para simplificar visualmente el grafo general. En la codificación se codifican los estados internos. Los macroestados pueden estar anidados.
- Estado Inicial. Se distingue por tener un círculo Interno. Es el estado inicial en el que nos encontramos al resetear la unidad.

8.2 TRANSICIONES

Las Transiciones son las distintas condiciones que pueden provocar un cambio de estado en la lógica funcional de la unidad que se está modelando. Cada estado puede tener muchas transiciones y cada una de estas tiene una prioridad, por lo que en el caso de ser ciertas varias condiciones de distintas transiciones, se salta a la de mayor prioridad. Cada transición puede tener asignada una acción.

El modelo visual de las transiciones, son las flechas. Al lado o lo más cercano posible a la flecha está la condición de transición y debajo de esta, separado por una línea horizontal, la acción que realiza al cumplirse la condición. Siempre que existe una acción de transición aparece la línea horizontal, aún en el caso de que, como en las acciones de estado, el número de acciones de transición sea tan elevado que no podamos visualizarlas correctamente. Cada transición tiene asignada una prioridad de ejecución que aparece en un círculo a la salida de la transición del estado. Si el estado tiene una transición, no se visualiza la prioridad porque sólo hay una, t1.

8.3 GNERAL

En todos los grafos aparece una tabla con las señales de salida e internas y el tipo de cada una. Las señales de salida (OUT), al contrario que las internas (INT), no pueden ser leídas directamente. Para leer una señal de salida tenemos que crear una variable interna "fantasma o dummy" y asignar la salida o bien de forma concurrente, o bien de forma síncrona. Digamos que tenemos una salida llamada "action", internamente podemos crear la señal action_dummy (señal que podremos leer en cualquier momento) y luego, fuera del proceso de manera concurrente asignar la salida action<=action_dummy.

Cuando definimos una señal externa, esta puede ser combinacional, registrada o síncrona (el tipo se puede ver en la columna Status, ver la figura 27). Cuando definimos una señal externa como registrada o síncrona, el conversor genera una señal con el mismo nombre añadiendo la terminación _int o _cld respectivamente, de esta manera podemos leer el puerto de salida. La diferencia entre las señales registradas y síncronas es que las síncronas son registradas en el proceso síncrono y asignadas concurrentemente al final de la arquitectura, mientras que las señales registradas son asignadas en el proceso combinacional y registradas en el proceso síncrono. Si hacemos una señal externa combinacional, , no podemos leerla al no ser que generemos una señal _dummy como indicábamos en el párrafo anterior.

Si la señal es interna, podemos definirla como combinacional o síncrona. En este caso, al ser una señal interna, no es necesario crear una señal auxiliar para leerla.

Tanto las señales síncronas como las registradas tienen un valor de inicialización (RESET) y un valor por defecto (DEFAULT), ver la tabla de la figura 27.

Cada unidad puede estar compuesta por una o varias máquinas de estado o grafos generando dos procesos más por máquina (el proceso síncrono y el que define el grafo)..

El RESET puede ser asíncrono o síncrono. Teniendo en cuenta que "res" está sincronizado en INIT (n1/r3), res es asíncrono en todos los grafos de estado.

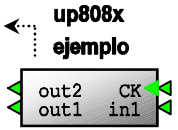
8.4 CODIGO VHDL

Si echamos un vistazo al código VHDL generado a partir del grafo, podemos distinguir las siguientes partes:

DEFINICIÓN DE LA ENTIDAD:

Define los puetos del componente.

```
ENTITY ejemplo IS
    PORT(
        CK   : IN     Std_Logic;
        in1  : IN     Std_Logic;
        out1 : OUT    Std_Logic;
        out2 : OUT    Std_Logic
    );
```



Después de la definición de los puertos, aparece la definición de la arquitectura interna del grafo, en ella se encuentran las definiciones, declaraciones, procesos y asignaciones concurrentes que implementan el comportamiento del Grafo.

DEFINICIONES

- Todas las señales que definamos en el grafo son insertadas en esta parte de la arquitectura.
- El programa utilizado también inserta en esta parte las señales necesarias para la codificación del grafo o las señales de salida registradas (_int) o síncronas (_cld).
- En esta parte de definiciones, también encontramos la definición del estado y su codificación (si existe)

```
    -- Architecture Declarations
    --Contador de Módulo 2
    SIGNAL contador : Integer RANGE 0 TO 1;

    TYPE STATE_TYPE IS (
        Reposo,
        Estado_2,
        Estado_1
    );

    -- State vector declaration
    ATTRIBUTE state_vector : string;
    ATTRIBUTE state_vector OF fsm : ARCHITECTURE IS
"current_state" ;

    -- Enum encoding
    ATTRIBUTE ENUM_ENCODING : string;
    ATTRIBUTE ENUM_ENCODING OF STATE_TYPE: TYPE IS
        "00 " &  -- Reposo
        "10 " &  -- Estado_2
        "01";  -- Estado_1

    -- Declare current and next state signals
    SIGNAL current_state : STATE_TYPE ;
    SIGNAL next_state : STATE_TYPE ;

    -- Declare any pre-registered internal signals
    SIGNAL out1_int : Std_Logic  ;
    SIGNAL out2_int : Std_Logic  ;
```

CUERPO DE LA ARQUITECTURA. PROCESOS

En el cuerpo de la arquitectura se encuentran los procesos que modelan el grafo, existe un proceso síncrono y uno o dos procesos combinacionales por máquina de estados. Se puede decir que existen dos posibles formatos, el de tres procesos y el de dos procesos.

- Proceso Síncrono "clocked": En el proceso síncrono (clocked) es donde se registran las salidas definidas como registro (int), donde se actualizan las salidas definidas como síncronas (_cld), donde se actualiza el estado del grafo o donde se registran las señales internas (como contador). El ejemplo no dispone de habilitación de chip (Chip Enable, CE) y reset.
- Proceso Combinacional "next_state": Se encarga de generar el siguiente estado y asignar las salidas del grafo, todo en el mismo proceso (como el ejemplo) cuando utilizamos el formato de dos procesos. Este proceso pude ser dividido en dos, "output" y "next_state", en este caso, estamos utilizando el formato de tres procesos.

```
-----------------------------------------------
clocked : PROCESS(
    CK)
-----------------------------------------------
BEGIN
    IF (CK'EVENT AND CK = '1') THEN
        current_state <= next_state;
        -- Registered output assignments
        out1 <= out1_int;
        out2 <= out2_int;
        -- Default Assignment To Internals
        contador <= 0;
      -- Combined Actions for internal signals only
        CASE current_state IS
        WHEN Estado_2 =>
            contador <= contador+1;
        WHEN OTHERS =>
            NULL;
        END CASE;
    END IF;
END PROCESS clocked;
```

```
-----------------------------------------------
nextstate : PROCESS (
    current_state,
    in1)
-----------------------------------------------
BEGIN
    -- Default Assignment
    out1_int <= '0';
    out2_int <= '0';
    -- Default Assignment To Internals
    -- Default state assignment
    next_state <= current_state;
    -- Combined Actions
    CASE current_state IS
    WHEN Reposo =>
        IF (in1='1') THEN
            out1_int <= '1';
            next_state <= Estado_1;
        ELSE
            next_state <= Reposo;
        END IF;
    WHEN Estado_2 =>
        out2_int <= '1';
        IF (in1='0') THEN
            next_state <= Reposo;
        ELSE
            next_state <= Estado_2;
        END IF;
    WHEN Estado_1 =>
        IF (in1='0') THEN
            next_state <= Reposo;
        ELSIF (in1='1') THEN
            out2_int <= '1';
            next_state <= Estado_2;
        ELSE
            next_state <= Estado_1;
        END IF;
    WHEN OTHERS =>
    next_state <= Reposo;
    END CASE;
END PROCESS nextstate;
```

9. UNIDAD DE RESET INICIALIZACIÓN (n1/r3)

Esta es la unidad encargada de generar las dos fases de reloj y de atender la señal RESET.

Fig 24.- Componente de Inicialización

Es un elemento común para los dos modelos por lo que esta integrado en la librería "up808x" concretamente en el fichero "up808x_init.vhd". Como se puede ver en la jerarquía, este fichero define un autómata de Meal o lo que es lo mismo, un grafo de estados cuyas salidas están en función del estado actual y las entradas. A continuación podemos ver el grafo de estados:

9.1 ENTRADAS SALIDAS Y SEÑALES INTERNAS

Este componente es junto con ports los únicos que reciben directamente entradas desde el exterior, las entradas son:

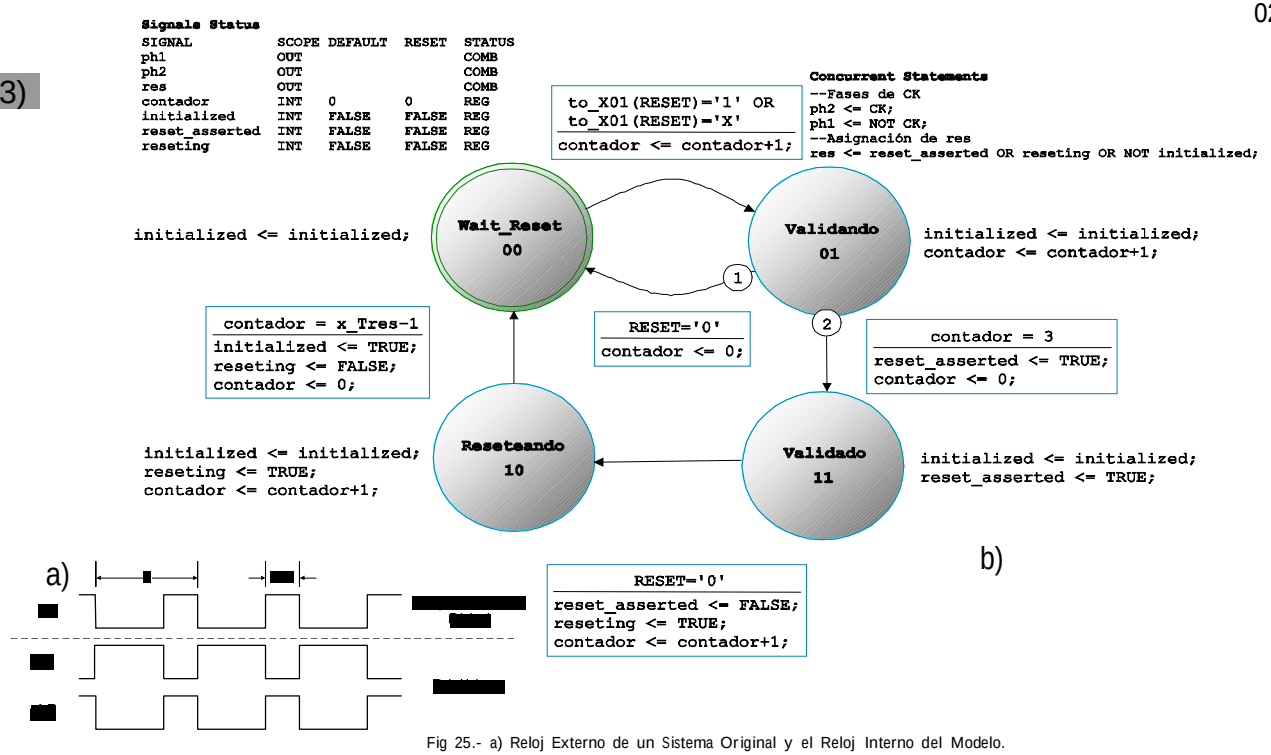


Fig 25.- a) Reloj Externo de un Sistema Original y el Reloj Interno del Modelo.
b) Grafo de Estados de la Unidad INIT (ni/r3).

ALGUNAS CUESTIONES SOBRE EL RELOJ

Ph1: es la fase 1 de la señal externa CK, es decir la fase en la que CK='0'. Ver la figura 25. Esta señal, siempre que hagamos funcionar el modelo en un sistema diseñado originariamente para un 8088/8086, tendrá un ciclo de trabajo del 33%, lo que significa que ph1 permanece 2/3 del tiempo de ciclo de reloj a nivel '0'. Esta asimetría en el ciclo de trabajo la utilizan los micros originales para optimizar los tiempos de propagación y mantenimiento, utilizando la fase larga para las unidades que tienen mucha lógica combinacional intermedia.

Ph2: es la fase 2 de la señal de CK, fase en la que CK='1'. En un sistema original, ph1='1' durante 1/3 del periodo.

Señal	Tipo	Valor Defecto	Valor de Reset	Estado	Función
CK	IN Port	-	-	-	Reloj externo
RESET	IN Port	-	-	-	Señal externa con la que reseteamos el sistema
res	OUT	-	-	COMB	Es la señal interna que resetea todos los componentes del modelo
ph1	OUT	-	-	COMB	Fase 1 de la señal externa CK, es decir, la fase en la que CK='0'.
ph2	OUT	-	-	COMB	Fase 2 de la señal de CK, fase en la que CK='1'.
contador	INT	0	0	REG	Contador de ciclos.
initialized	INT	FALSE	FALSE	REG	Primera inicialización.
reset_asserted	INT	FALSE	FALSE	REG	RESET aceptado
reseting	INT	FALSE	FALSE	REG	Reseteando el Modelo

Fig 12.- Señales Utilizadas en el modelado del componente INIT.

VERTICALIDAD, SIMULTANEIDAD, FUERZA Y PUREZA DE CK.

El reloj debe llegar directa y en coincidencia a todos los biestables del modelo para que estos registren sus datos a la vez (SIMULTANEIDAD), además el reloj tiene que ser lo más vertical posible (VERTICALIDAD). Estas requisitos en la señal de reloj están fuertemente relacionados con la FUERZA de esta señal, si la capacidad de suministrar carga (intensidad de la señal) de los drivers es alta, el tiempo empleado en transmitir carga a las entradas de reloj de los biestables es muy rápido y simultáneo (si no hay mucha diferencia en las longitudes de las pistas de reloj hasta los biestables). Bajo ningún concepto he utilizado lógica con el reloj, de manera que la señal llega lo más limpia y pura posible a los biestables (PUREZA)

El diseño de los modelos orientado a eliminar problemas de ruido e inestabilidades es una de las explicaciones de la decisión de crear dos fases en el modelo.

9.2 EXPLICACIÓN DEL GRAFO DE ESTADOS

El grafo está compuesto por cuatro estados lo que implica que se van a codificar en dos biestables. El código asignado a cada estado aparece debajo de su nombre.

COMPORTAMIENTO:

En condiciones iniciales, nada más conectar la alimentación, la señal "initialized=FALSE" por lo que el modelo permanece reseteado (res <= reset_asserted OR reseting OR NOT initialized).

Cuando aparece RESET tenemos que validar la petición antes de resetear el modelo, si la señal no se mantiene a nivel alto durante al menos 4 ciclos de reloj, se considera que ha sido un RESET fallido y no se resetea el sistema. Si se trata de un RESET válido (reset_asserted=TRUE), saltamos a un nuevo estado en espera de que RESET='0'. Cuado RESET='0', saltamos a un estado en el que seguimos reseteando el modelo hasta que el contador alcalce el núero de ciclos indicado en la constante de configuración x_Tres declarada en el package. Ver el código de abajo.

La constante x_Tres es necesaria para mantener la compatibilidad con sistemas originales. En los micros originales la secuencia de reset (desde que éste, una vez validado, pasa a ser '0') tarda 7 ciclos de reloj; por lo que en el modelo, si queremos emular el comportamiento del original tenemos que hacer que la secuencia de reset también dure 7 ciclos de reloj. Esto lo conseguimos con un contador interno de módulo x_Tres que mantiene reseteando el sistema el número de ciclos indicado en esta constante.

ESTADOS:

- Wait_Reset (e0) "00": Estado inicial, permanecemos en este estado hasta que aparece un '1' o valor indeterminado 'X' en la entrada RESET (transición 1, t1). La condición RESET='X', hace referencia al caso en el que el valor de la señal es indeterminado. Si tenemos en cuenta que RESET suele estar conectado a través de una red RC y un pulsador en paralelo con el condensador, (ver la figura 26), el caso de valor indeterminado correspondería a los instantes iniciales (nada más conectar la alimentación) en los que al micro le llega tensión pero el condensador no está totalmente cargado, el valor lógico de esta situación equivale a RESET='1'.

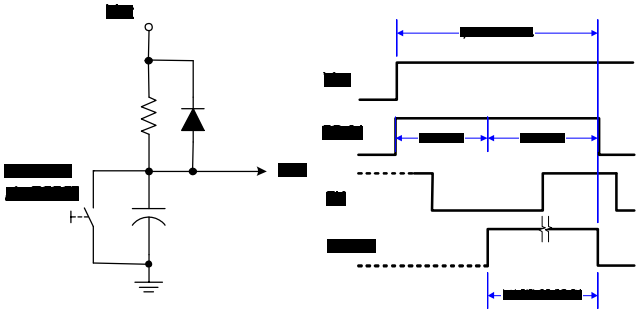


Fig 26.- Circuito RC de RESET y el Valor de la Transición e0/t1.

- Validando (e1) "01": A este estado se le puede considerar como un antirebotes hardware, es decir, discrimina si el pulso en RESET es un pulso válido (t2) o hay un "glich" o estado espurio (t1) en el terminal (estos estados pueden aparecer al intrducirse ruido en la señal (ver microelectrónica de Pollán capítulo de ruidos) o en los microrebotes provocados al activar el pulsador de reset, por ejemplo). Se considera que RESET es válido cuando el contador (que se incrementa cada ciclo que permanecemos en e1, contador<=contador+1) alcanza el valor 3 (4 ciclos de reloj en total), entonces reseteamos el contador y activamos la bandera de, reset_asserted<= TRUE.

  Notar que t1 tiene mayor prioridad que t2 por lo que en el caso de que contador = 3 y RESET='0' saltamos a e0 quedando anulada la secuencia de reset. Caso en el que RESET='1' durante 3 CK.

- Validado (e2) "11": Permanecemos en este estado hasta que RESET='0' (t1). Podemos decir que en éste estado, esperamos que RESET deje de ser 1 para lanzar la secuencia final de reset. De no existir este estado, podríamos encontrarnos con la situación de ciclo infinito, al autoiniciarse múltiples veces el grafo ante una señal de RESET prolongada. En acciones de la transición, activamos la banderita de sistema reseteándose (reseting <= TRUE) y lanzamos el contaje de ciclos de dicha secuencia.

- Reseteando (e3) "10": Es el estado en el que reseteamos el sistema durante el número de ciclos indicado en la constante de configuración x_Tres. Como se comentó anteriormente es necesario para la compatibilidad cuando el modelo se va a implantar en sistemas hardware diseñados originalmente para los microprocesadores 8088/8086 en los que este estado se prolonga durante 7 ciclos. Si no vamos a implementar el modelo en un sistema original, no es necesario implementar este retardo y podemos establecer la constante de configaura-ción en pack808x_pck.vhd a 1 (x_Tres=1).

CAMINOS

Este es un grafo simple que sólo tiene dos caminos:

CAMINO	SECUENCIA	DESCRIPCIÓN
c1	Wait_Reset-Validando-Validado-Reseteando-Wait_Reset	Camino de ejecución normal o habitual. Secuendia de reset completada. Reset válido
c2	Wait_Reset-Validando-Wait_Reset	Reset no válido.

Tab 13.- Posibles Caminos del Grafo de Estados.

```
--|-------------------------------------------------------------------------|--
--|                        cONSTANTES dE CONFiGURACióN                      |--
--|.........................................................................|--
--|                                                                         |--
--|cONSTANTES dE cONFiGURACióN dE lOS mODELOS                               |--
--Segmento inicial
CONSTANT segment_ini : Std_Logic_Vector(15 DOWNTO 0) := x"FFFF";
--Número de ciclos de reset (mayor de 3)
CONSTANT x_Tres : Integer := 7;
--Número de bytes de la cola de intruccines (mayor de 2 para el up8086)
CONSTANT queue_size : Integer := 4; --Por defecto 4 para 8088, 6 para 8086
--|.........................................................................|--
```

10. REGISTROS DE USO GENERAL (n1/r1)

Registros de Uso General, Register General Use (RGU): Son los registros de uso general, forman parte del modelo software (ver modelo software del microprocesador) de los micros, lo que significa que son transparentes para el usuario, por lo que éste puede verlos, modificarlos u operar con ellos. Son 8 registros de 16 bits, a 4 de ellos se puede acceder, independientemente, al byte alto o al byte bajo. Son registros de acceso rápido y la unidad de ejecución los utiliza de forma implícita en algunas intrucciones, por lo que estos registros adquieren una función específica e implícita al ejecutar dicha instrucción.

Cualquiera de estos registros puede utilizarse como fuente o destino en operaciones aritméticas y lógicas, lo que no se puede hacer con ninguno de los seis registros de segmento de la unidad sru (n1/r2) como se verá más adelante.

Además de las funciones indicadas en la tabla 14, cada registro tiene usos especiales:

10.1 DESCRIPCIÓN DEL FICHERO

En la siguiente figura, podemos ver las entradas y salidas de este componente:

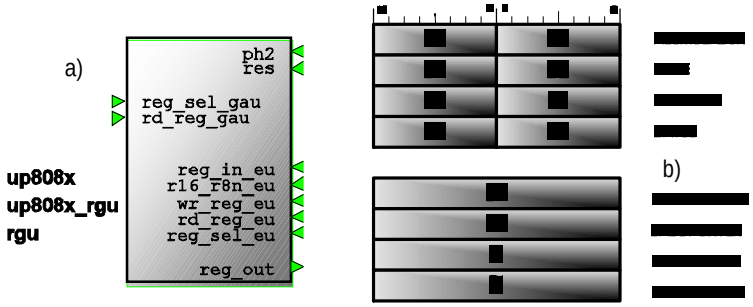


Fig 19.- a) Carcasa de RGU. b) Registros Internos de RGU.

Es un elemento común para los dos modelos por lo que esta integrado en la librería “up808x” concretamente en el fichero “up808x_rgu.vhd”. Como se puede ver en la jerarquía, este fichero define un modelo comportamental de los registros escrito directamente en VHDL.

REGISTRO	DESCRIPCIÓN
AX (Accumulator)	Registro acumulador, dividido en AH y AL (8 bits cada uno). Usándolo se produce (en general) una instrucción que ocupa un byte menos que si se utilizaran otros registros de uso general. Su parte más baja, AL, también tiene esta propiedad. El último registro mencionado es el equivalente al acumulador de los procesadores anteriores (8080 y 8085). Además hay instrucciones como DAA; DAS; AAA; AAS; AAM; AAD; LAHF; SAHF; CBW; IN y OUT que trabajan con AX o con uno de sus dos bytes (AH o AL). También se utiliza este registro (junto con DX a veces) en multiplicaciones y divisiones.
BX (Base)	Registro base, dividido en BH y BL. Es el registro utilizado como base en algunas operaciones del microprocesador (se usa para direccionamiento indirecto) y es una versión más potente del par de registros HL de los procesadores anteriores.
CX (Counter)	Registro contador, dividido en CH y CL. Se utiliza como contador en bucles (instrucción LOOP), en operaciones con cadenas (usando el prefijo REP) y en desplazamientos y rotaciones (usando el registro CL en los dos últimos casos).
DX (Data)	Registro de datos, dividido en DH y DL.. Se utiliza junto con el registro AX en multiplicaciones y divisiones, en la instrucción CWD y en IN y OUT para direccionamiento indirecto de puertos (el registro DX indica el número de puerto de entrada/salida)
SP (Stack Pointer)	Puntero de pila (no se puede subdividir). Aunque es un registro de uso general, debe utilizarse sólo como puntero de pila, la cual sirve para almacenar las direcciones de retorno de subrutinas y los datos temporales (mediante las instrucciones PUSH y POP). Al introducir (push) un valor en la pila a este registro se le resta dos, mientras que al extraer (pop) un valor de la a pila este registro se le suma dos.
BP (Base Pointer)	Puntero base (no se puede subdividir). Generalmente se utiliza para realizar direccionamiento indirecto dentro de la pila.
SI (Source Index)	Puntero índice (no se puede subdividir). Sirve como puntero fuente para las operaciones con cadenas. También sirve para realizar direccionamiento indirecto.
DI (Destin Index)	Puntero destino (no se puede subdividir). Sirve como puntero destino para las operaciones con cadenas. También sirve para realizar direccionamiento indirecto.

Fig 14.- Significado y Función de los Registros de Uso General.

10.2 ENTRADAS Y SALIDAS

Señal	Tipo	Función
ph2	IN	Reloj interno
res	IN	Es la señal interna que resetea todos los componentes del modelo
reg_in_eu	IN	Dato de 16 bits a escribir en el registro
r16_r8n_eu	IN	Indica si EU va a operar sobre un registro de 16 bits (r16_r8n_eu='1') o en un registro de 8 bits (r16_r8n_eu='0').
wr_reg_eu	IN	Indica que EU va a escribir en un registro.
reg_sel_eu	IN	Indica el registro con el que se va a operar.
rd_reg_gau	IN	Indica que GAU va a leer un registro.
reg_sel_gau	IN	Indica el registro gau (n3/r7) va a leer.
reg_out	OUT	Salida de los registros, EU y GAU leen en esta salida el registro deseado.

Tab 15.- Descripción de la función de los pines de RGU.

10.3 PROCESOS

Tabla de procesos en el fichero:

Proceso	Tipo	Función
Priority	Combinacional	Determina la unidad que accederá antes a los registros cuando ambas lo requieran a la vez
write_rgu	Síncrono	Enclava el registro direccionado en el flnco de subida de ph2.
read_rgu	Combinacional	Es el proceso encargado de sacar el registro seleccionado.

Fig 16.- Procesos en RGU.

A RGU acceden, para la lectura de un determinado registro de 16 bits, tanto la EU como la BIU (concretamente la parte de la BIU que genera la dirección física, Gen Address Unit, gau n3/r7). Como hay dos unidades que acceden a la lectura, es necesario establecer un prioridad, el encargado de resolver la prioridad es el proceso Priority:

```
--|PriORiTY rESOLVER (Proceso Combinacional)                          |--
Priority : PROCESS (rd_reg_eu,wr_reg_eu,r16_r8n_eu,reg_sel_eu,
                    rd_reg_gau,reg_sel_gau) IS
BEGIN
    --Valores por defecto
    rd_reg     <= FALSE;
    r16_r8n    <= '0';
    rd_reg_sel <= "000";
    wr_reg_sel <= "000";
    --Establece preferencias en las unidades que acceden a los registros
    IF wr_reg_eu THEN
    --Sólo EU escribe
        r16_r8n    <= r16_r8n_eu;
        wr_reg     <= TRUE;
        wr_reg_sel <= reg_sel_eu;
    END IF;

    IF rd_reg_eu THEN
    --EU tiene mayor prioridad
        r16_r8n <= r16_r8n_eu;
        rd_reg  <= TRUE;
        rd_reg_sel <= reg_sel_eu;
    ELSIF  rd_reg_gau THEN
    --GAU tiene prioridad baja
        r16_r8n <= '1';            --GAU sólo lee registros de 16 bits
        rd_reg  <= TRUE;
        rd_reg_sel <= reg_sel_gau; --GAU sólo lee bases e índices
    END IF;
END PROCESS Priority;
```

Se puede observar cómo sólo EU escribe en RGU y cómo los registros a los que accede GAU son siempre registros de 16 bits. Otra cuestión que hay que destacar es la independencia en las operaciones de escritura y lectura. Si aparecen simultáneamente una operación de escritura (por EU) y una lectura (por GAU), ambas son atendidas correctamente. Esto es debido a que internamente están separadas la selección del registro en el que vamos a escribir (“wr_reg_sel”) de la selección del registro que vamos a leer (“rd_reg_sel”).

Existe un proceso síncrono (write_rgu) encargado de la escritura de los registros:

```
--|wRiTE PrOCESS (Proceso Síncrono)                                    |--
write_rgu : PROCESS (ph2,res) IS
BEGIN
    IF res THEN --res asíncrono
        --Registros internos de uso general
        CX <= X"0000";  AX <= X"0000";  BX <= X"0000";  DX <= X"0000";
        SP <= X"0000";  BP <= X"0000";  SI <= X"0000";  DI <= X"0000";
    ELSIF ph2'EVENT AND ph2='1' THEN
        IF wr_reg AND r16_r8n = '0' THEN --Registros de 8 bits
            CASE wr_reg_sel IS
            WHEN "000" => AL <= reg8_in;
            WHEN "001" => CL <= reg8_in;
            WHEN "010" => DL <= reg8_in;
            WHEN "011" => BL <= reg8_in;
            WHEN "100" => AH <= reg8_in;
            WHEN "101" => CH <= reg8_in;
            WHEN "110" => DH <= reg8_in;
            WHEN "111" => BH <= reg8_in;
            WHEN OTHERS => NULL;
            END CASE;
        ELSIF wr_reg AND r16_r8n = '1' THEN --Registros de 16 bits
            CASE wr_reg_sel IS
            WHEN "000" => AX <= reg16_in;
            WHEN "001" => CX <= reg16_in;
            WHEN "010" => DX <= reg16_in;
            WHEN "011" => BX <= reg16_in;
            WHEN "100" => SP <= reg16_in;
            WHEN "101" => BP <= reg16_in;
            WHEN "110" => SI <= reg16_in;
            WHEN "111" => DI <= reg16_in;
            WHEN OTHERS => NULL;
            END CASE;
        END IF;
    END IF;
END PROCESS write_rgu;
```

En la tabla 17 podemos ver en forma de tabla decodificación del campo de selección de registro:

Registro ≡REG	r16_r8n=’0’ ≡W=’0’	r16_r8n=’1’ ≡W=’1’
000	AL	AX
001	CL	CX
010	DL	DX
011	BL	BX
100	AH	SP
101	CH	BP
110	DH	SI
111	BH	DI

Tab 17.- Decodificación del campo reg.

Existe un tercer proceso, el proceso de lectura (read_rgu), que es el encargado de poner el registro indicado por “rd_reg_sel” y “r16_r8n” en la salida.

```
--|rEAD PrOCESS (Proceso Combinacional)      |--

read_rgu: PROCESS (rd_reg,r16_r8n,rd_reg_sel,
                   AL,CL,DL,BL,AH,CH,DH,BH,
                   AX,CX,DX,BX,SP,BP,SI,DI) IS
BEGIN
    reg_out <= X"0000";
    --Lectura de los registros de 8 bits
    IF rd_reg AND r16_r8n = '0' THEN
        --Registros de 8 bits
        CASE rd_reg_sel IS
        WHEN "000" => reg8_out <= AL;
        WHEN "001" => reg8_out <= CL;
        WHEN "010" => reg8_out <= DL;
        WHEN "011" => reg8_out <= BL;
        WHEN "100" => reg8_out <= AH;
        WHEN "101" => reg8_out <= CH;
        WHEN "110" => reg8_out <= DH;
        WHEN "111" => reg8_out <= BH;
        WHEN OTHERS => NULL;
        END CASE;
    --Lectura de los registros de 16 bits
    ELSIF rd_reg AND r16_r8n = '1' THEN
        --Registros de 16 bits
        CASE rd_reg_sel IS
        WHEN "000" => reg16_out <= AX;
        WHEN "001" => reg16_out <= CX;
        WHEN "010" => reg16_out <= DX;
        WHEN "011" => reg16_out <= BX;
        WHEN "100" => reg16_out <= SP;
        WHEN "101" => reg16_out <= BP;
        WHEN "110" => reg16_out <= SI;
        WHEN "111" => reg16_out <= DI;
        WHEN OTHERS => NULL;
        END CASE;
    END IF;
END PROCESS read_rgu;
```

11. REGISTROS DE SEGMENTO (n1/r2)

n1/r2, Registros de Segmento, Segment Register Unit (SRU): Son 8 Registros de 16 bits cada uno, 5 de ellos son transparentes para el usuario y forman parte del modelo software (ver modelo software), 4 de estos registros transparentes, son los registros de segmento CS, DS, SS y ES, el otro registro visto desde el exterior, es el puntero de instrucciones IP. Los 3 registros restantes son registros auxiliares de uso interno, (A0, A1 y A2). Estos registros son utilizados por la unidad de generación de direcciones (Gen Address Unit, GAU n3/r7) para que dependiendo del modo de direccionamiento, generar la dirección física (Phisical Address, PA).

Estos registros no pueden ser utilizados por instrucciones aritméticas y lógicas. Solamente pueden ser modificados por la instrucción MOV e implícitamente por instrucciones que producen un salto en el flujo normal del programa (CALL/JNZ/...).

REGISTRO DE SEGMENTO	DESCRIPCIÓN
CS (Segmento de Código)	Apunta al segmento de código actual que contiene las instrucciones que va a ser ejecutada.
SS (Segmento de Pila)	Este registro apunta a el segmento actual de pila que es utlilizado por todas las instrucciones que salvan datos en la pila.
DS (Segmento de Datos)	Apunta al segmento donde se encuentran los datos y las variables del programa.
ES (Segmento Extra)	Este registro apunta a la dirección inicial del segmento de extra que es normalmente utilizado para almacenar datos.
IP (Puntero de Instrucciones)	Indica siempre la posición relativa a CS de la siguiente instrucción que se va a ejecutar.
A0,A1 y A2 (Registros Auxiliares)	Estos Registros internos, son utilizados por la unidad de ejecución en ciertas instrucciones. Solamente la unidad de ejecución tiene acceso a ellos.

Tab 18.- Registros de la unidad SRU.

11.1 ENTRADAS Y SALIDAS

SEÑAL	TIPO	FUNCIÓN
ph2	IN	Reloj interno
res	IN	Es la señal interna que resetea todos los componentes del modelo
sreg_in_eu	IN	Dato de 16 bits a escribir en el registro
wr_sreg_eu	IN	Indica que la EU va a escribir en un registro.
rd_sreg_eu	IN	Indica que la EU va a leer un registro.
sreg_sel_eu	IN	Indica el registro con el que se va a operar.
rd_sreg_gau	IN	Indica que GAU va a leer un registro.
sreg_sel_gau	IN	Indica el registro que gau (n3/r7) va a leer.
sreg_out_eu	OUT	Salida de los registros hacia la unidad de ejecución EU..
sreg_out_gau	OUT	Salida de los registros, hacia GAU.

Tab 19.- Descripción de la función de los pines de SRU.

11.2 FUNCIONAMIENTO

En la siguiente figura, podemos ver las entradas y salidas del componente y los registros internos de SRU:

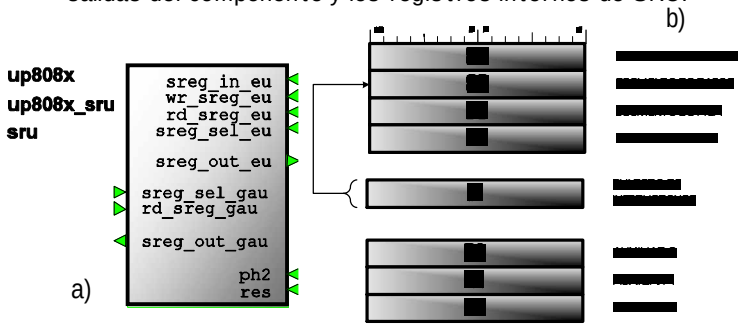


Fig 30.- a) Pines de SRU. b) Registros Internos de SRU.

Como los registros de uso general, estos registros son un elemento común para todos los modelos por lo que pertenecen a la librería “up808x” concretamente está definido en el fichero “up808x_srg.vhd”. Este fichero define un modelo comportamental de los registros escrito en VHDL.

11.3 PROCESOS

En este caso el fichero está compuesto por un proceso, el proceso de escritura “write_sru”, y dos asignaciones concurrentes de las salidas (que equivaldrían a dos procesos combinacionales).

A simple vista llama la atención las dos salidas independientes de los registros, una salida hacia la unidad de ejecución “sreg_out_eu” y otra hacia la unidad de generación de direcciones “sreg_out_gau”. Como en rgu hay dos unidades que acceden a los registros (EU y GAU) pero a diferencia de lo que ocurría con éste, los accesos a sru por parte de estas dos unidades puede coincidir en el tiempo (caso en el que GAU está generando una dirección para ir en busca de una instrucción y EU está trabajando con registros de segmento). Ante esta situación se puede optar por varias soluciones:

- 1.- Diseñar un árbitro de sru que establezca la prioridad de acceso y mantenga en espera a la otra unidad o
- 2.- Hacer un registro de doble salida, salidas independientes para las dos unidades que acceden a ellos.

Está claro que elegí la segunda por ser la que mayores prestaciones de velocidad, simplicidad ofrece. Mejora de velocidad por tener ambas unidades acceso inmediato a la lectura de los registros de segmento (sru) y simplicidad por no añadir lógica de estados a EU y GAU (los estados de espera a que la otra unidad termine de utilizar sru), de esta manera simplemente añadimos un multiplexor más a la lógica de sru. En contra aparecen 16 bits más en el componente, pero como se ha comentado en otras ocasiones, las FPGA no suelen estar limitadas en este aspecto.

Los registros de sru se definen en pocas líneas de VHDL como podemos vea a continuación: ➤

```
--|.....................................................................|--
--|wRiTE PrOCESS (Proceso Síncrono)                                     |--

write_sru: PROCESS (ph2,res) IS

BEGIN
    IF res THEN --res asincrono

    --Registros de segmento
    DS <= X"0000";  ES <= X"0000";  SS <= X"0000";  CS <= Segment_Ini;
    --IP y Registros Auxiliares
    IP <= X"0000";  A1 <= X"0000";  A2 <= X"0000";  A3 <= X"0000";

    ELSIF ph2'EVENT AND ph2='1' THEN
        IF wr_sreg_eu THEN
            CASE sreg_sel_eu IS
            WHEN "000" => ES <= sreg_in_eu; --Extra Segment
            WHEN "001" => CS <= sreg_in_eu; --Code Segment
            WHEN "010" => SS <= sreg_in_eu; --Stack Segment
            WHEN "011" => DS <= sreg_in_eu; --Data Segment
            WHEN "100" => IP <= sreg_in_eu; --Instruction Pointe
            WHEN "101" => A1 <= sreg_in_eu; --Auxiliar Register 1
            WHEN "110" => A2 <= sreg_in_eu; --Auxiliar Register 2
            WHEN "111" => A3 <= sreg_in_eu; --Auxiliar Register 3
            WHEN OTHERS => NULL;
            END CASE;
            END IF;
    END IF;
END PROCESS write_sru;

--|.....................................................................|--
--|aSiGNACiONES cONCURRENTES (Proceso Combinacional)                    |--

--Lectura de los registros de segmento
sreg_out_eu <= ES WHEN sreg_sel_eu="000" AND rd_sreg_eu ELSE
               CS WHEN sreg_sel_eu="001" AND rd_sreg_eu ELSE
               SS WHEN sreg_sel_eu="010" AND rd_sreg_eu ELSE
               DS WHEN sreg_sel_eu="011" AND rd_sreg_eu ELSE
               IP WHEN sreg_sel_eu="100" AND rd_sreg_eu ELSE
               A1 WHEN sreg_sel_eu="101" AND rd_sreg_eu ELSE
               A2 WHEN sreg_sel_eu="110" AND rd_sreg_eu ELSE
               A3 WHEN sreg_sel_eu="111" AND rd_sreg_eu ELSE
               (OTHERS => '0'); --Valor por defecto

sreg_out_gau <= ES WHEN sreg_sel_gau="000" AND rd_sreg_gau ELSE
                CS WHEN sreg_sel_gau="001" AND rd_sreg_gau ELSE
                SS WHEN sreg_sel_gau="010" AND rd_sreg_gau ELSE
                DS WHEN sreg_sel_gau="011" AND rd_sreg_gau ELSE
                IP WHEN sreg_sel_gau="100" AND rd_sreg_gau ELSE
                A1 WHEN sreg_sel_gau="101" AND rd_sreg_gau ELSE
                A2 WHEN sreg_sel_gau="110" AND rd_sreg_gau ELSE
                A3 WHEN sreg_sel_gau="111" AND rd_sreg_gau ELSE
                (OTHERS => '0'); --Valor por defecto
--|.....................................................................|--
```

En la siguiente tabla, podemos ver la decodificación del campo de selección.

sreg_sel(_eu/_gau)	Sreg_out(_eu/_sru)	Valor de RESET
000	ES	X"0000"
001	CS	segment_ini
010	SS	X"0000"
011	DS	X"0000"
100	IP	X"0000"
101	A0	X"0000"
110	A1	X"0000"
111	A0	X"0000"

Tab 20.- Decodificación del campo de selección.

El valor de incialización del registro que contiene el segmento de código está definido en la constante de configuración "segment_ini" declarada en el package del sistema. Ver el código un poco más abajo. Este valor es muy importante porque determina la posición de memoria a partir de la cual el microprocesador empezará a ejecutar instrucciones. En los microprocesadores originales "segment_ini=x"FFFF" por lo que el micro comienza en la dirección física PA=X"FFFF0". En nuestros modelos podemos definir el segmento de inicio en la posición que queramos, simplemente tenemos que modificar la constante de configuaración "segment_ini".

```
--|-------------------------------------------------------------------------|--
--|                          cONSTANTES dE CONFiGURACióN                    |--
--|.........................................................................|--
--|                                                                         |--
--|cONSTANTES dE cONFiGURACióN dE lOS mODELOS                               |--
--Segmento inicial
CONSTANT segment_ini : Std_Logic_Vector(15 DOWNTO 0) := x"FFFF";
--Número de ciclos de reset (mayor de 3)
CONSTANT x_Tres : Integer := 7;
--Número de bytes de la cola de intruccines (mayor de 2 para el up8086)
CONSTANT queue_size : Integer := 4; --Por defecto 4 para 8088, 6 para 8086
--|.........................................................................|--
```

12. PUERTOS (n1/r6)

n1/r6, Puertos (ports): Ésta unidad se encarga de multiplexar, en función de la entrada MN/MXn (modo de funcionamiento), los terminales del microprocesador.

Esta unidad contiene casi todos los puertos del microprocesador (excepto los terminales RESET y CK que pertenecen a la unidad de inicialización n1/r3). En las tablas 9, 10 y 11 puede verse el tipo de puerto y la función de cada pin. En el esquema jerárquico (figura 16), podemos ver que existen dos componentes por librería y que existen diferentes versiones para los modelos 8088/8086. Ver las carcasas en la figura 31 y los archivos en la tabla siguiente.

LIBRERÍA	ARCHIVO	DESCRIPCIÓN
up8088	up8088_ports_mn	Contiene el modelo para el 8088 en modo mínimo
up8088	up8088_ports_mx	Contiene el modelo para el 8088 en modo máximo
up8086	up8086_ports_mn	Contiene el modelo para el 8088 en modo mínimo
up8086	up8088_ports_mx	Contiene el modelo para el 8088 en modo máximo

Tab 21.- Todos los Archivos del Componente ports (n1/r6).

En la siguiente figura podemos cotejar las diferencias entre las distintas carcasas en las dos librerías:

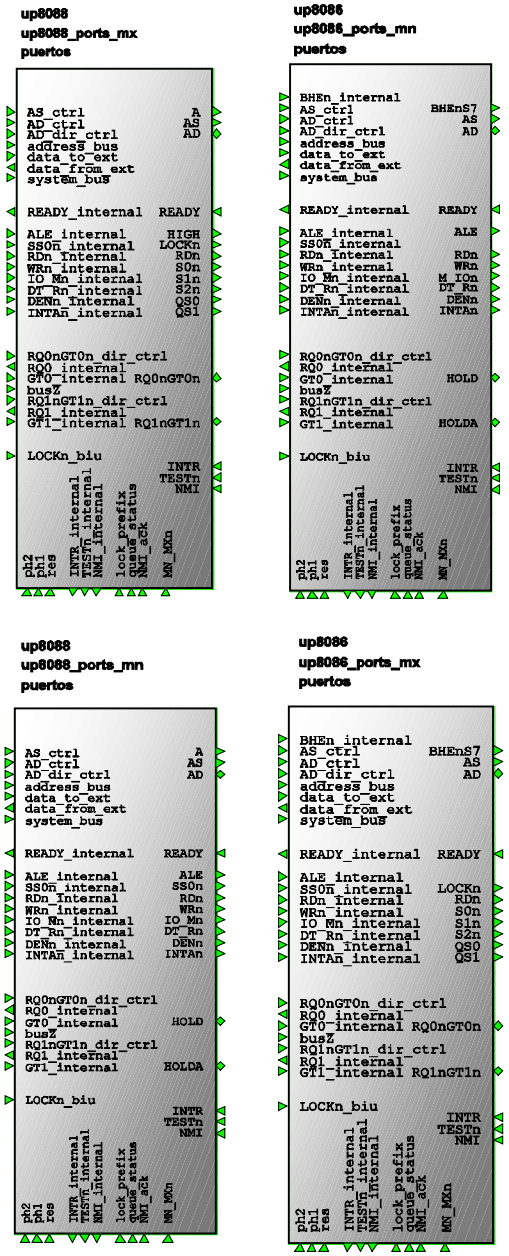


Fig 31.- Todas las Carcasas del Componente ports (n1/r6).

SEÑAL	TIPO	FUNCIÓN
ph1, ph2	IN	Reloj interno
res	IN	Es la señal interna que resetea todos los componentes del modelo
AS_ctrl	IN	Control del multiplexor del puerto AS
AD_ctrl	IN	Control del Multiplexor del puerto AD
AD_dir_ctrl	IN	Controsa si AD es de salida o de entrada.
Address_bus	IN	Dirección que va a salir por los puertos A, A?
Data_to_ext	IN	Datos que saldrán por el puero AD.
Dato_from_ext	OUT	Datos recogidos en el puerto AD
System_bus	IN	Bus de sistema, saldrá por AS, y (S0n-S2n/DT_Rn-IO_Mn-SS0)
BHEn_internal	IN	Señal de activación del bloque alto de la memoria.
READY_internal	OUT	Señal de READY sincronizada.
ALE_internal	IN	Señales internas de sus respectivos puertos creadas por el Generador de Ciclos de Bus, bcg (n3/r9)
SS0n_internal		
RDn_internal		
WRn_internal		
IO_Mn_internal		
DT_Rn_internal		
DENn_internal		
INTAn_internal		
RQ0nGT0n_dir_ctrl	IN	Control de dirección del puerto RQ0nGT0n
RQ0_internal	OUT	Señales RQ0n y GT0n internas del puerto RQ0nGT0n. Generada por la unidad de control Multibus, mu (n3/r6)
GT0_internal	IN	
RQ1nGT1n_dir_ctrl	IN	Control de dirección del puerto RQ1nGT1n
RQ1_internal	OUT	Señales RQ1n y GT1n internas del puerto RQ1nGT1n. Generadas por la unidad de control Multibus, mu (n3/r6)
GT1_internal	IN	
busZ	IN	Pone todos los puertos con bufer triestado en alta impedancia.
LOCKn_biu	IN	Señal interna del puero LOCKn generada por la biu (n1/r7)
INTR_internal	OUT	Señal de interrupción
TESTn_internal	OUT	Señal sincronizada del terminal TESTn
NMI_internal	OUT	Interrupción No Enmascarable reconocida.
Lock_prefix	IN	La instrucción actual tiene activado el prefijo de Bloqueo
Queue_status	IN	Bus de estado de la cola de instrucciones, queue (n1/r5)
NMI_ack	IN	Reconocimiento de la interrupción, ya puede aparecer otra.

Tab 22.- Breve descripción de los pines del componente ports.

SEÑAL	TIPO	FUNCIÓN
A	OUT	Bus de Direcciones
AD	IN/OUT	Bus de Direcciones y Datos
AS	OUT	Bus de Direcciones y Sistema
BHEnS7	OUT	Enable del banco mas significativo de la memoria
READY	IN	Para sincronizar periféricos y memorias lentas
ALE	OUT	Cuando está uno indica que salen direcciones por AD, en caso contrario, es el bus de datos
LOCKn	OUT	Cuando vale cero indica a otros controladores del bus (otros microprocesadores o un dispositivo de DMA) que no deben ganar el control del bus. Se activa poniéndose a cero cuando una instrucción tiene el prefijo LOCK
RDn	OUT	Cuando vale cero hay una lectura
S0n-S2n	OUT	Bits de Estado
QS0-QS1	OUT	Estado de la cola de Instrucciones
SS0n	OUT	con IO/M y DT/R esta salida sirve para determinar estados del 8088/8086
WRn	OUT	Cuando vale cero hay una escritura
IO_Mn	OUT	Si vale 1: operaciones con ports, si vale 0: operaciones con la memoria
DT_Rn	OUT	Data transmit/receive: se conecta al pin de dirección de los chips recién indicados
DENn	OUT	Data enable: cuando vale cero debe habilitar los transceptores externos, sirve para que no se mezclen los datos y las direcciones
INTAn	OUT	Indica que reconoció la interrupción
INTR	IN	Petición de una interrupción Mascarable.
RQ1nGT1n	IN/OUT	Es bidireccional y tiene la misma función que HOLD/HLDA en modo mínimo
RQ1nGT1n	IN/OUT	Lo mismo que RQ1nGT1n pero con mayor prioridad
HOLD	IN/OUT	Indica que otro integrado quiere adueñarse del control de los buses, generalmente se usa para DMA.
HOLDA	IN/OUT	Reconocimiento de cesión del bus. Es una respuesta a HOLD
TESTn	IN	Entrada testeada por la instrucción WAIT
NMI	IN	Petición de una interrupción no enmascarable.
MN_MXn	IN	Selector de la señal a sacar por un puerto, MN_MXn='1' modo Mínimo; MN_MXn='0' modo Máximo.

Tab 23.- Resumen de los Puertos del componente ports. En las tablas 9, 10 y 11 podemos ver una Información más detallada.

12.1 ENTRADAS Y SALIDAS

En las tablas anteriores, podemos ver una breve descripción de los terminales del componente ports en todas sus variaciones. En el propio fichero VHDL hay una información mucho más detallada del funcionamiento de cada señal.

Para cada puerto existen unas señales de control (dependiendo del tipo de puerto, pueden variar, como veremos más adelante) y unas señales internas (_internal) que tienen el mismo significado que sus correspondientes señales externas, solo que sincronizadas. Por ejemplo, la señal "READY_internal" es la misma que la señal "READY" externa pero sincronizada, (mantenemos el principio de sincronismo registrando todas las señales de entrada).

12.2 TIPOS DE PUERTOS

En esta sección vamos a estudiar los tipos de puertos implementados en ports y sus señales de control.

Para ahorrar pines y poder insertar el microprocesador en una carcasa de 40 pines, los micros originales tienen multiplexadas las señales (a) en función del modo de funcionamiento y (b) en el tiempo.. Ver la figura 32. La multiplexación en el modo de funcionamiento está determinada por la configuración del pin MN_MXn, algunos pines multiplexados en el modo son WRn/LOCKn, IO_Mn/S2n, ALE/QS0...

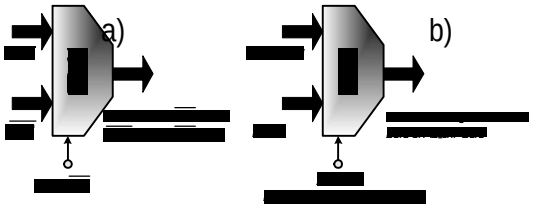


Fig 32.- a) Multiplexación en el Modo de Funcionamiento. b) Multiplexación en el Tiempo.

La multiplexación en el tiempo (ver la figura 32) es controlada por la unidad generadora de ciclos de bus (Bus Cycle Generador, bcg n3/r9) que se encarga de generar la señal de control para sacar distinta información por el mismo puerto en función des estado del bus externo (que está en función del tiempo, ver la temporización del sistema en biu_timing, n2/r8). Algunos puertos multiplexados en el tiempo son AS, AD, RQ0n/GT0n,...

Los terminales multiplexados en el modo de funcionamiento, no lo están en el tiempo y viceversa.

Esta técnica de multiplexación es útil sólo para desminuir el número de pines externos del microprocesador, por lo contrario, requiere mucho tiempo y aumenta la lógica necesaria (tanto externa como interna) para hacer funcionar un sistema. Perdemos tiempo en la multiplexación de las señales internas que luego (externamente) tenemos que demultiplexar, y aumentamos la lógica interna al necesitar un multiplexor por pin y la lógica discreta externa para demultiplexarlos.

Los modelos VHDL diseñados mantienen esta lógica de multiplexación de los pines por compatibilidad con los microprocesadores originales. Si no los vamos a utilizar en un sistema original ni con hardware diseñado por Intel ® o compatible, no es necesario mantener esta compatibilidad en los modelos pudiendo utilizar muchos más pines que los micros originales (las FPGA suelen tener muchos pines). De esta manera hacemos un micro más rápido y con menos lógica aumentando la eficacia de los modelos y manteniendo el potencial del juego de instrucciones y la lógica de bus.

PUERTO DE SALIDA:

Este puerto de salida lo único que hace es ir aumentado la capacidad de carga de la señal interna (_internal) mediante una serie de inversores de canal cada vez más ancho de manera que el puerto de salida pueda suministrar una intensidad unas 100 veces mayor que la de la señal interna. No voy a entrar mucho en detalle en cuestiones de corrientes, protecciones, configuraciones, etc de los puertos, porque son muy dependientes de la tecnología utilizada. Para más información sobre los puertos ver los Apuntes de Microelectrónica de Tomás Pollán (Universidad de Zaragoza).

En el micro solamente hay dos terminales con esta configuración, ALE e INTAn. Estos, no pueden desacoplarse del bus de control local ya que no poseen estado de alta impedancia. Su valor por defecto es '0'. En la siguiente figura, podemos ver un símbolo lógico que podría representar este tipo de puerto.

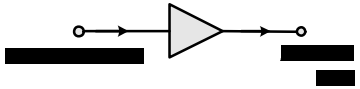


Fig 33.- Puerto de Salida

Si vemos el código del pin 25 (ALE/QS0) podemos ver la multiplexación en el modo de funcionamiento y la ausencia de buffer triestado.

```
--|bUS eXTERNO dE cONTROL (multiplexados en el modo
de fto)|--
--La señal ALE, nunca está en alta impedancia
ALE <= ALE_internal  WHEN to_X01(MN_MXn)='1' ELSE
       QS0_internal  WHEN to_X01(MN_MXn)='0' ELSE
       '0'; --Valor por defecto (MN_MXn='X' caso de
pin "al aire")
```

PUERTO DE SALIDA TRIESTADO:

Este puerto tiene las mismas características que el anterior sólo que con estado de alta impedancia. El estado de alta impedancia permite desacoplar el puerto del sistema al que está conectado. Los pines A, AS, (SS0n, RDn, WRIO_Mn, DT_Rn y DENn) modo mínimo y (LOCKn,S2n-S0n, QS0-QS1) modo máximo (bus de control del sistema) tienen este tipo de puertos, lo que les permite suministrar una corriente relativamente elevada al exterior, y lo más importante desacoplarse, quitarse, aislarse de las pistas externas del bus de control al entrar en alta impedancia. Otro maestro de canal podrá utilizar estas líneas para comandar el bus local como si fuera el único maestro del sistema.

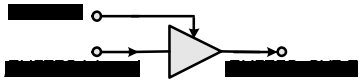


Fig 34.- Puerto Triestado de Salida

Si echamos un vistazo a la asignación concurrente que modela el pin M_Ion/S2n, podemos ver cómo está multiplexado en el modo de funcionamiento y cómo dos señales hacen entrar el pin en estado de alta impedancia.

```
M_IOn <= 'Z'   WHEN busZ='1' OR res ELSE
M_IOn_internal WHEN to_X01(MN_MXn)='1' ELSE
S2n_internal   WHEN to_X01(MN_MXn)='0' ELSE
'0'; --Valor por defecto
```

Una es la señal de res (reset interno del microprocesador) y la otra es busZ generada por la BIU y que pone todos los puertos con buffer triestado en alta impedancia

PUERTO DE ENTRADA-SALIDA TRIESTADO:

Este tipo de puerto puede mandar o recibir datos hacia o desde el exterior, además posee un buffer triestado que permite desacoplar los pines del exterior. En la figura 35 podemos ver un esquema lógico de este tipo de puerto. Este puerto tiene dos señales de control, una para el buffer triestado (representado en la figura 35 por una puerta de transmisión) y otra que controla la dirección del dato.

También puede observarse como hay señales independientes para el dato de entrada y el de salida. Recordar que no existen buses triestado internos.

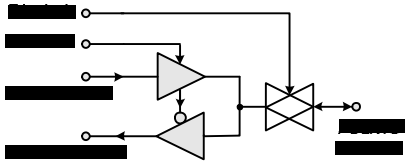


Fig 35.- Puerto Triestado de Entrada-Salida

En los microprocesadores sólo los pines AD tienen este tipo de puerto. Los puertos (HOLD/HOLDA o RQ0n_GT0n/RQ1n_GT1n) también son de entrada salida pero tienen unas características especiales como se puede ver a continuación.

PUERTO DE ENTRADA SALIDA CON RESITENCIA PULL-UP

Puede mandar/recibir datos hacia/desde el exterior. En este caso el puerto no tiene buffer triestado (como en el caso anterior), en su defecto dispone de una resistencia interna conectada a la tensión positiva de

la alimentación (Vcc) que fija un valor de entrada alto (nivel lógico '1') cuando el puerto está configurado como entrada. Ver la figura 36. Para más información sobre este tipo de puerto, ver la página 11 del pdf "80C88 (FN2949).pdf"

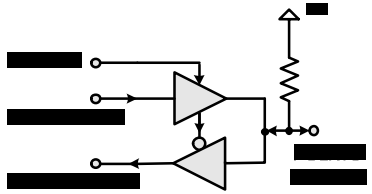


Fig 36.- Puerto Bidireccional con Resistencia Pull-up.

Solamente hay dos pines en el microprocesador que disponen de este tipo de puerto, HODL/RQ0n_GT0n y HOLDA/RQ1n_GT1n.

Cuando estudiemos la lógica de Multibus (n3/r3) veremos que para solucionar las diferencias entre la lógica de HOLD/HOLDA (modo mínimo, un solo canal) y RQ0n_GT0n (canal 0)/RQ1n_GT1n (canal 1), se interpreta la aparición de HOLD como un pulso en RQ0n_internal (petición en el canal 0), por lo que hay que generar un pulso cuando aparece la señal HOLD. Con el circuito de la figura 37 (detector de flancos de subida) conseguimos el propósito deseado.

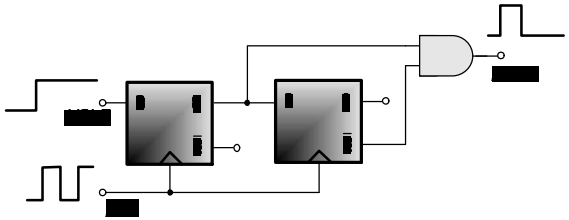


Fig 37.- Detector de Flancos en Hold.

Este esquema está implementado en la unidad de puertos mediante el siguiente proceso:

```
flancos : PROCESS (ph1,res) --Detector de flancos en la
señal HOLD
BEGIN
    IF (res=TRUE) THEN
        flanco <= '0';
        last_HOLD <= '0';
    ELSIF ph1'EVENT AND ph1='1' THEN
        --Valores por defecto
        flanco <= '0';
        last_HOLD <= to_X01(HOLD);
        IF (NOT last_HOLD AND to_X01(HOLD))='1' THEN
            flanco <= '1';
        END IF;
    END IF;
END PROCESS:
```

Puertos de Entrada

Todas las entradas desde el exterior son sincronizadas para evitar problemas con el sincronismo del sistema (problemas con el tiempo de mantenimiento o preparación de la señal). El siguiente esquema muestra como la señal de entrada es registrada en disponiendo de una señal interna síncrona, que es la que utilizaremos.

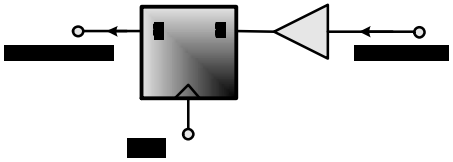


Fig 38.- Registro de la señal de Entrada.

El triangulo que aparece junto al registro representa al buffer de entrada que normalmente está formado por una resistencia (una larga pista de polisilicio) y dos diodos de protección formados por la unión PN entre drenaje y substrato de dos transistores (NMOS y PMOS) con la puerta y el drenaje cortocircuitados. Para más información ver los apuntes de Microelectrónica de T.Pollán.

Los terminales INTR, TESTn y RESET disponen de este tipo de puerto.

```
--INTR, es la entrada de solicitud de interrupción --
Mascarable, es activada por nivel y es muestreada --
en el flanco de subida de cada ciclo de reloj. La --
interrupt_unit se encarga de atenderla o no en    --
función de I_flag.

INTR_internal <= '0' WHEN res ELSE
to_X01(INTR) WHEN ph2'EVENT AND ph2='1';
```

Entrada NMI

El reconocimiento de NMI requiere un tratamiento especial. En principio la señal interna (NMI_internal) se tiene que activar en el flanco de subida del puerto NMI. Pero buscando en la literatura vemos que NMI tiene que estar por lo menos 2 ciclos de reloj a nivel alto para ser aceptada. Teniendo en cuenta que no puede haber otra petición en NMI hasta que la unidad de ejecución termine de atender el servicio de interrupción (indicado por la señal NMI_ack), el proceso de reconocimiento de NMI queda de la siguiente manera:

```
NMI_process : PROCESS (ph2,res,NMI_ack)
--NMI_internal es disparada cuando hay un flanco de
--subida en el pin NMI y aceptada cuando está este
--pin, al menos 2 ciclos a nievel alto. NMI es
--desactivada cuando EU ha saltado a la rutina de
--servicio de NMI, mediante la activación de la señal
--NMI_ack.
VARIABLE trigger:Boolean;
VARIABLE count:Std_Logic;
BEGIN
    IF res OR NMI_ack THEN
        NMI_internal <= '0';
        count := '0';
        trigger := FALSE;
    ELSIF ph2'EVENT AND ph2='1' THEN
        IF trigger AND count='1' THEN
            NMI_internal <= '1';
            count := count;
        ELSIF trigger AND to_X01(NMI)='1' THEN
            count := '1';
        ELSIF to_X01(NMI)='1' THEN
            trigger := TRUE;
            count := '0';
        ELSE
            trigger := FALSE;
            count := '0';
        END IF;
    END IF;
END PROCESS;
```

Declaraciones De Tipos Enumerados

Para manejar los puertos se crearon unos tipos de datos enumerados para hacer más sencillo comprender el funcionamiento de los puertos.

```
--|TiPOS dEFiNiDOS PaRA eL cONTROL dE lOS PuERTOS dE eNTRADA-sALiDA|--
TYPE Direction_Typ IS (dir_output,dir_input); --Sentido del puerto
TYPE AD_Ctrl_Typ   IS (address,data);         --Control del puerto AD
TYPE AS_Ctrl_Typ   IS (address,system);       --Control del puerto AS
--Codificación de los tipos enumerados
ATTRIBUTE enum_encoding OF Direction_Typ : TYPE IS "0 1";
ATTRIBUTE enum_encoding OF AD_Ctrl_Typ   : TYPE IS "0 1";
ATTRIBUTE enum_encoding OF AS_Ctrl_Typ   : TYPE IS "0 1";
```

En las líneas anteriores, podemos ver el código de los tipos de datos enumerados junto a su codificación.

Nota:

Al final del fichero de declaración de los puertos, aparece un proceso no sintetizables (proceso ciclos). Este proceso simplemente sirve para visualizar en la simulación una cadena de caracteres indicando el estado del sistema, es decir, en ver de ver S2n-S0n="101", tendremos una señal que nos indique que el ciclo de bus actual es "read_mem" (lectura de la memoria". Todo esto se explica más detalladamente en la sección de simulación.

13. COLA DE INSTRUCCIONES

n1/r5, Cola de Instrucciones (queue): Es una memoria Tipo FIFO cuya función es memorizar las siguientes instrucciones en ser ejecutadas. Con esta estrategia conseguimos mejorar los tiempos de ejecución ya que el microprocesador no tiene que acceder a la memoria en busca de la instrucción, sino que la recoge directamente de la cola de instrucciones que es una memoria interna y por lo tanto mucho más rápida (concepto de memoria caché).

En el árbol jerárquico de la figura 16, podemos ver que este componente existe en la librería del modelo 8088 y en la del 8086, concretamente en los archivos "up8088_queue.vhd" y "up8086_queue.vhd", y que estos archivos definen una modelo comportamental.

En las siguientes figuras, podemos ver las diferentes carcasas y la estructura interna de una memoria FIFO.

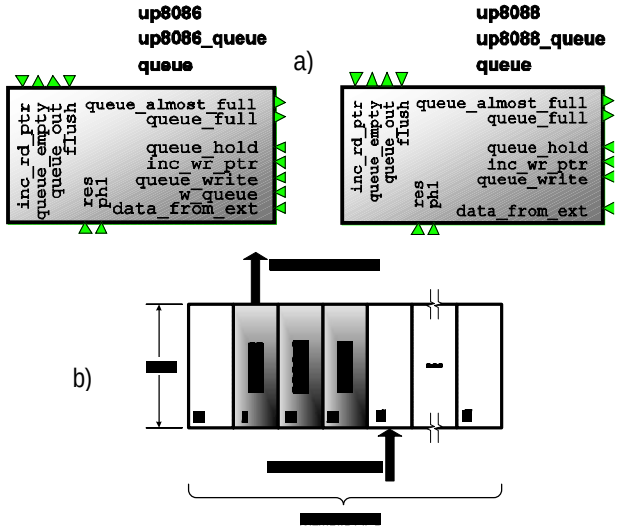


Fig 39.- a) Carcasas de la Cola de Instrucciones. b) Estado de una memoria FIFO de 8 Bytes Después de 4 Escrituras y 1 Lectura.

Existen diferencias sustanciales a nivel interno entre la cola de instrucciones para el microprocesador 8088 y la del 8086, como se verá más adelante. A nivel externo hay pocas diferencias, las únicas diferencias son:

- Aparece (en up8086_queuel la señal w_queue que permite escribir datos de 16 bits
- También en esta misma carcasa los datos recibidos desde el exterior son de 16 bits.

13.1 ENTRADAS Y SALIDAS

SEÑAL	TIPO	FUNCIÓN
ph1	IN	Reloj interno
res	IN	Es la señal interna que resetea todos los componentes del modelo
inc_rd_ptr	IN	Incrementa el puntero de lectura
inc_wr_ptr	IN	Incrementa el puntero de escritura
queue_write	IN	Escribe un dato en el registro de la cola de instrucciones indicado por el puntero de escritura.
queue_hold	IN	Deshabilita el reloj de la cola ("congela la cola")
flush	IN	Borra todos los datos.
queue_full	OUT	Indica que la cola está llena
queue_almost_full	OUT	Indica que la cola está casi llena
queue_empty	OUT	Indica que no hay ningún dato en la cola
data_from_ext	IN	Dato de entrada (8/16 bits en 8088/8086 respectivamente.
queue_out	OUT	Salida del dato (8 bits)
w_queue	IN	Indica si se escriben 8 o 16 bits.

Tab 24.- Breve Descripción de la Función de las Señales

13.2 COLA DE INSTRUCCIONES EN EL 8088

Para implementar una memoria FIFO (Frist In, Frist Out), es necesario manejar dos punteros (uno de escritura, wr_ptr, y otro de lectura, rd_ptr). Estos punteros se encargan de apuntar a la siguiente posición de la memoria en la que se escribirán los datos de entrada (wr_ptr) y a la posición de memoria de la que vamos a leer el dato (rd_ptr). El puntero de lectura se incrementa cuando la entrada inc_rd_ptr vale '1', de la misma manera que el puntero de escritura se incrementa al aparecer un '1' lógico en la entrada inc_wr_ptr.

Cuando llegamos al "techo" de los punteros, otro incremento provocará un desbordamiento o volcado del puntero. En la figura 40 podemos ver las dos situaciones de desbordamiento que pueden aparecer en la cola de instrucciones del 8088.

El 8088 original tiene una cola de instrucciones de 4 bytes, en los modelos podemos variar el ancho de la cola de instrucciones modificando la constante de configuración "queue_size" en el package del sistema (n0/r0):

```
--Número de bytes de la cola de intruccines (mayor de 2
--para el up8086). Por defecto 4 para    8088, 6 para
--8086

CONSTANT queue_size : Integer := 4;
```

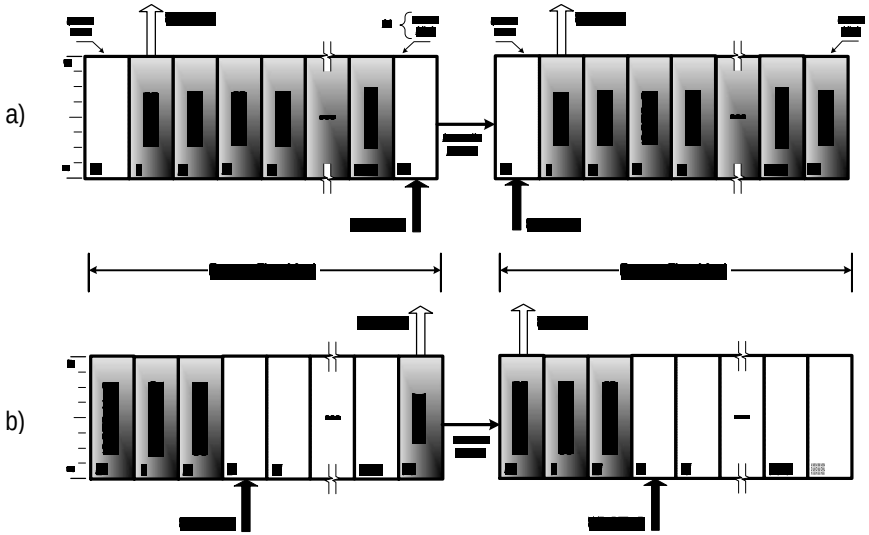
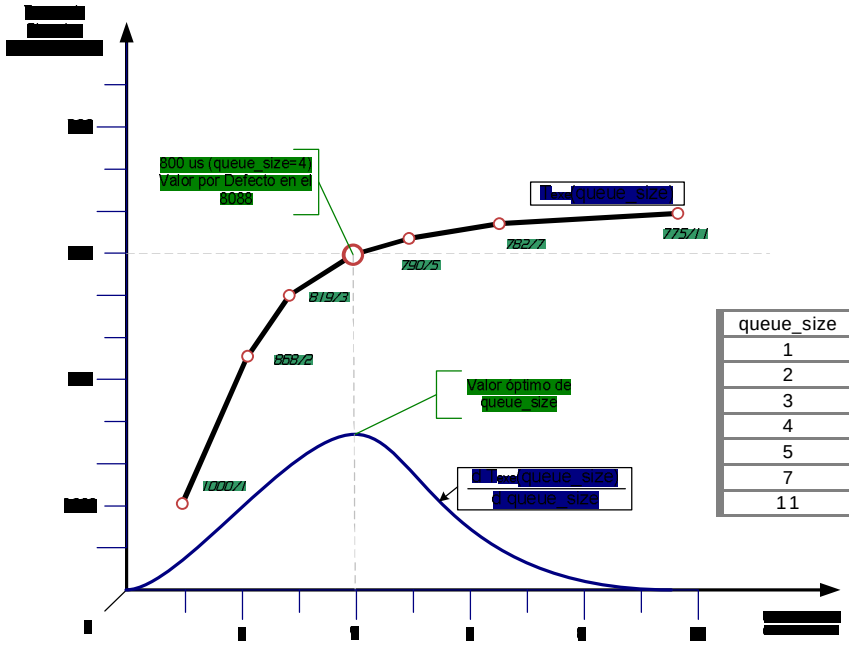


Fig 40.- a) Desbordamiento del Puntero de Escritura
b) Desbordamiento del Puntero de Lectura. Cola de Instrucciones del 8088



queue_size	Incremento en t
1	+200 us
2	+68 us
3	+19 us
4	0 us
5	-10 us
7	-18 us
11	-25 us

Fig 41.- Gráfica de Velocidad en Función de queue_size. Testeado en el modelo 8088.

Modificando esta constante de configuración podemos variar la eficacia de la cola de instrucciones. Como se puede ver en la gráfica de la figura 41, el tiempo de ejecución de un programa varía exponencialmente en función del valor de queue_size. Haciendo la primera derivada de esta exponencial podemos ver el momento de mayor pendiente (mayor incremento de tiempo por variación de queue_size) que corresponderá al valor óptimo de queue_size (4).

Esta gráfica ha sido generada con datos de la simulación, utilizando como programa de ejecución el programa de test utilizado para depurar los modelos de los microprocesadores Este programa contiene todas las instrucciones del microprocesador, por lo que lo hace un buen modelo para obtener datos aplicables a nivel general.

A nivel particular, la curva exponencial es muy dependiente del las instrucciones utilizadas en el programa ya que en códigos con muchas instrucciones de salto (por ejemplo), la eficacia de la cola de instrucciones es mucho menor (la curva exponencial se convierte, más bien, en una recta horizontal).

PROCESOS

Siguiendo un poco con la explicación del funcionamiento, internamente el fichero contiene dos procesos síncronos:

- Queue: Este es el proceso principal, se encarga de incrementar los punteros y de actualizar el estado de la cola de instrucciones. Como puede verse en el código del proceso, la señal queue_hold habilita todo el proceso. La señal flush tiene mayor prioridad que las de petición de incremento de puntero (pueden aparecer inc_wr_ptr e inc_rd_ptr a la vez). En el incremento de los punteros se puede apreciar como se actúa en caso de desbordamiento de registro al incrementar un puntero que "ha tocado techo" (queue_rd_ptr=queue_high).

Las señales queue_empty, queue_almost_full y queue_full, indican al exterior el estado interno de la cola de instrucciones. Ver la siguiente tabla:

ESTADO	SIGNIFICADO
queue_empty	La cola está vacía.
queue_almost_full	Le falta un byte para ser llenada del todo.
queue_full	La cola de Instrucciones está llena

Tab 25.- Estado de la Cola de Instrucciones

```
--|PrOCESO PriNCiPAL, iNCREMENTO dE PuNTEROS i aCTUALiZACióN dEL
eSTADO|--
queue : PROCESS (ph1,res) IS
    VARIABLE bytes_queued : INTEGER RANGE queue_low TO queue_size;
BEGIN

--|iNiCiALiZACióN dE lA QuEUE
    IF res THEN
        bytes_queued      := 0;     --Bytes en cola
        queue_wr_ptr      <= 0;     --Puntero de escritura
        queue_rd_ptr      <= 0;     --Puntero de lectura
        queue_empty       <= TRUE;  --Cola vacía
        queue_full        <= FALSE; --Cola llena
        queue_almost_full <= FALSE; --Cola casi llena
    ELSIF ph1'EVENT AND ph1='1' THEN
--|iNCREMENTA lOS PuNTEROS dE lA cOLA                                |--
        IF NOT queue_hold THEN --queue_hold=clk not enable
            IF flush THEN  --Inicializa punteros y estado
                queue_wr_ptr <= 0;
                queue_rd_ptr <= 0;
                queue_empty  <= TRUE;
                queue_full   <= FALSE;
                queue_almost_full <= FALSE;
                bytes_queued := 0;
            ELSE --Se puede incrementar el puntero de lectura y
                 --escritura a la vez
                IF inc_wr_ptr THEN --Incrementa puntero de escritura
                    IF bytes_queued < queue_size THEN
                        IF    queue_wr_ptr < queue_high THEN
                            queue_wr_ptr <= queue_wr_ptr + 1;
                        ELSIF queue_wr_ptr = queue_high THEN
                            queue_wr_ptr <= 0;
                        END IF;
                        bytes_queued := bytes_queued + 1;
                    END IF;
                END IF;
                IF inc_rd_ptr THEN --Incrementa puntero de lectura
                    IF  bytes_queued > 0 THEN
                        IF  queue_rd_ptr < queue_high   THEN
                            queue_rd_ptr <= queue_rd_ptr + 1;
                        ELSIF queue_rd_ptr = queue_high THEN
                            queue_rd_ptr <= 0;
                        END IF;
                        bytes_queued := bytes_queued - 1;
                    END IF;
                END IF;
            END IF;
--|aCTUALiZACióN dEL eSTADO dE lA cOLA                                |--
            --Valores por defecto
            queue_empty <= FALSE;
            queue_full <= FALSE;
            queue_almost_full <= FALSE;

            IF bytes_queued = 0 THEN
                queue_empty  <= TRUE;
            END IF;
            IF bytes_queued = queue_size-1 THEN
                queue_almost_full <= TRUE;
            END IF;
            IF bytes_queued = queue_size THEN
               queue_full <= TRUE;
            END IF;
        END IF; --hold
    END IF;--clk
END PROCESS queue;
```

- write_queue (proceso): Este proceso (cuyo código podemos ver bajo éste párrafo) es el encargado de escribir el dato de entrada (data_from_ext, de 8 bits en el 8088) en la memoria FIFO.

```
--|...................................................|--
--|PrOCESO dE eSCiTURA                                |--

write_queue : PROCESS (ph1,res)
BEGIN

--|iNiCiALiZACióN dE lOS rEGiSTROS                    |--
IF res THEN --res asincrono
    FOR index IN queue_high DOWNTO 0 LOOP
    --Inicialización a 0
        queue_array(index) <= (OTHERS => '0');
    END LOOP;

ELSIF ph1'EVENT AND ph1='1' THEN --escritura síncrona

--|eSCRiTURA eN eL aRRAY                              |--
    IF NOT queue_hold THEN --queue_hold=clk not enable
        IF queue_write THEN
        --Graba dato
            queue_array(queue_wr_ptr) <= data_from_ext;
        END IF;
    END IF;
END IF;
END PROCESS write_queue;
```

13.3 COLA DE INSTRUCCIONES EN EL 8086

La principal diferencia entre las dos colas de instrucciones es la longitud del dato de entrada (data_from_ext). Mientras que en el 8088 es de 8 bits, en el 8086 es de 16 bits, sin embargo, la salida (queue_out) es de 8 bits en ambos modelos (de esta manera mantenemos el mismo decodificador)

```
--|...................................................|--
--|TiPOS dE dATOS dE lA cOLA dE iNSTRUCCiONES         |--

--Ancho de la palabra, los micros 8088 y 8086 almacenan
--siempre bytes
    CONSTANT queue_width : INTEGER := 8;
    --Límites alto y bajo de la pila
    CONSTANT queue_high : INTEGER := queue_size-1;
    CONSTANT queue_low  : INTEGER := 0;

TYPE fifo_array_type IS ARRAY (INTEGER RANGE
   queue_high DOWNTO queue_low) OF
   Std_Logic_Vector (queue_width-1 DOWNTO queue_low);
```

El código anterior muestra como defino el tipo de dato de la memoria fifo interna utilizando constantes.

Al tener que salvar una palabra en dos bytes consecutivos del "fifo_array", son necesarios dos punteros de escritura (wr_ptr0 y wr_ptr1) el primero apunta al byte menos significativo y el segundo al más significativo. La lógica de control del desbordamiento de los punteros de escritura, ahora es más compleja al tener que controlar dos punteros cuyo incremento normalmente es de 2 en 2 (si w_queue='1') aunque en el caso de tener que alinear la cola de instrucciones con el bus externo, es de 1 en 1 (w_queue='0')

También necesitamos un puntero de lectura (rd_ptr) cuya lógica de control de desbordamiento es la misma que la del 8088, la lectura se realiza de byte en byte por lo que el incremento del puntero de lectura es siempre de 1 en 1. Ver la figura 41.

La lógica de estado de la cola de instrucciones sufre pequeños cambios, ahora la cola está llena aunque quede un byte libre en el array y está casi llena si quedan 3 o 2 bytes libres.

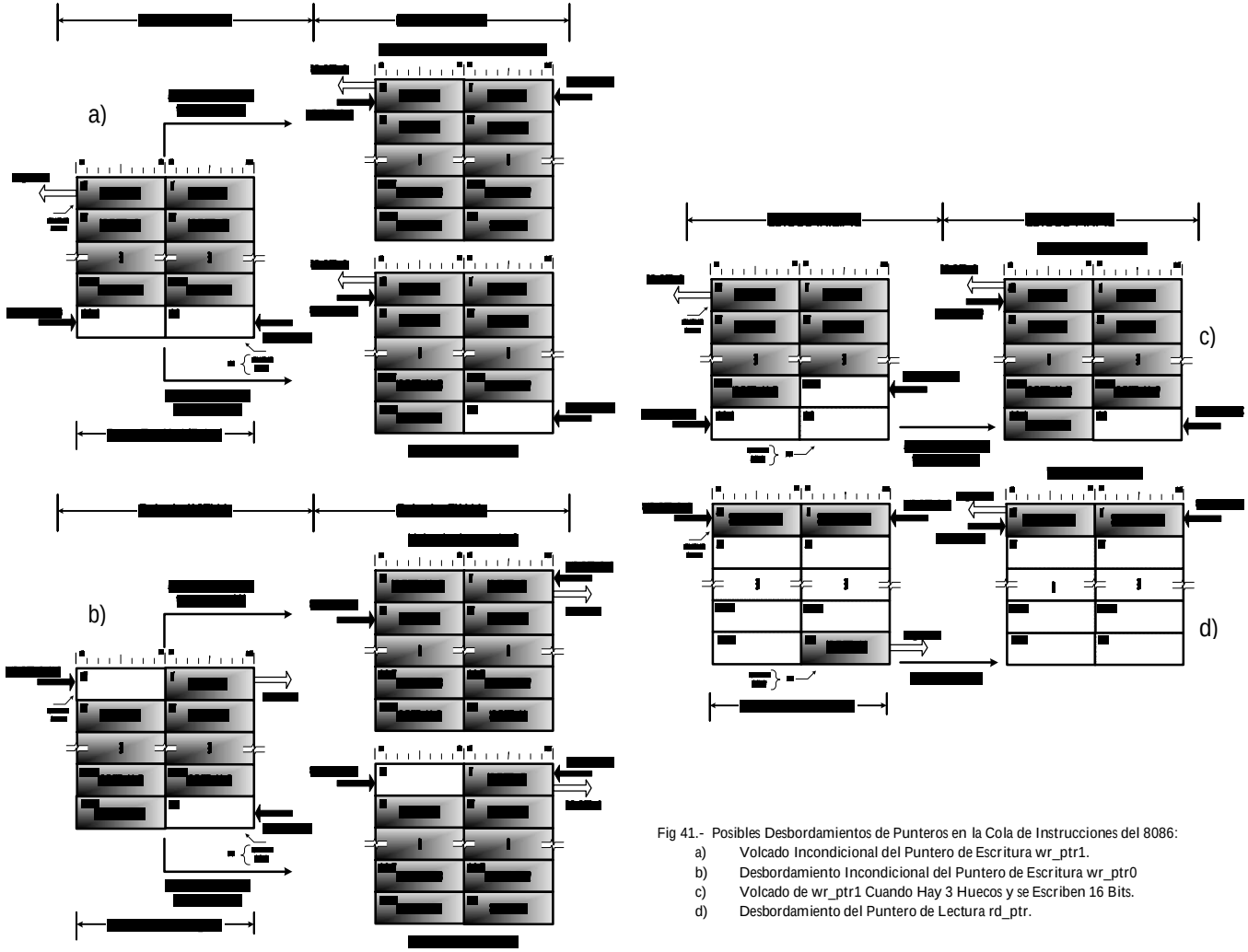


Fig 41.- Posibles Desbordamientos de Punteros en la Cola de Instrucciones del 8086:
a) Volcado Incondicional del Puntero de Escritura wr_ptr1.
b) Desbordamiento Incondicional del Puntero de Escritura wr_ptr0
c) Volcado de wr_ptr1 Cuando Hay 3 Huecos y se Escriben 16 Bits.
d) Desbordamiento del Puntero de Lectura rd_ptr.

EU

Unidad de Ejecución

14. UNIDAD DE EJECUCIÓN, EU (n1/r4)

En el Nivel 1 sólo quedan por explicar dos unidades, la unidad de ejecución (EU) y la unidad de interface con el bus (BIU) que, por su importancia (son las dos ramas principales) voy a tratar a parte en los siguientes capítulos. Iré "podando" las ramas del árbol hasta llegar a la copa explicando en detalle cada una de las unidades en cada uno de los niveles. No obstante, todos los ficheros VHDL de este proyecto están ampliamente comentados, por lo que echando un vistazo a estos podremos aclarar muchas de las dudas que "asalten" al lector durante el estudio de los modelos.

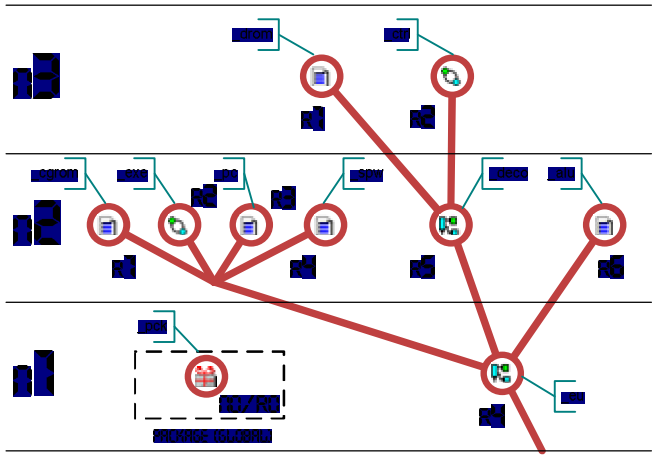
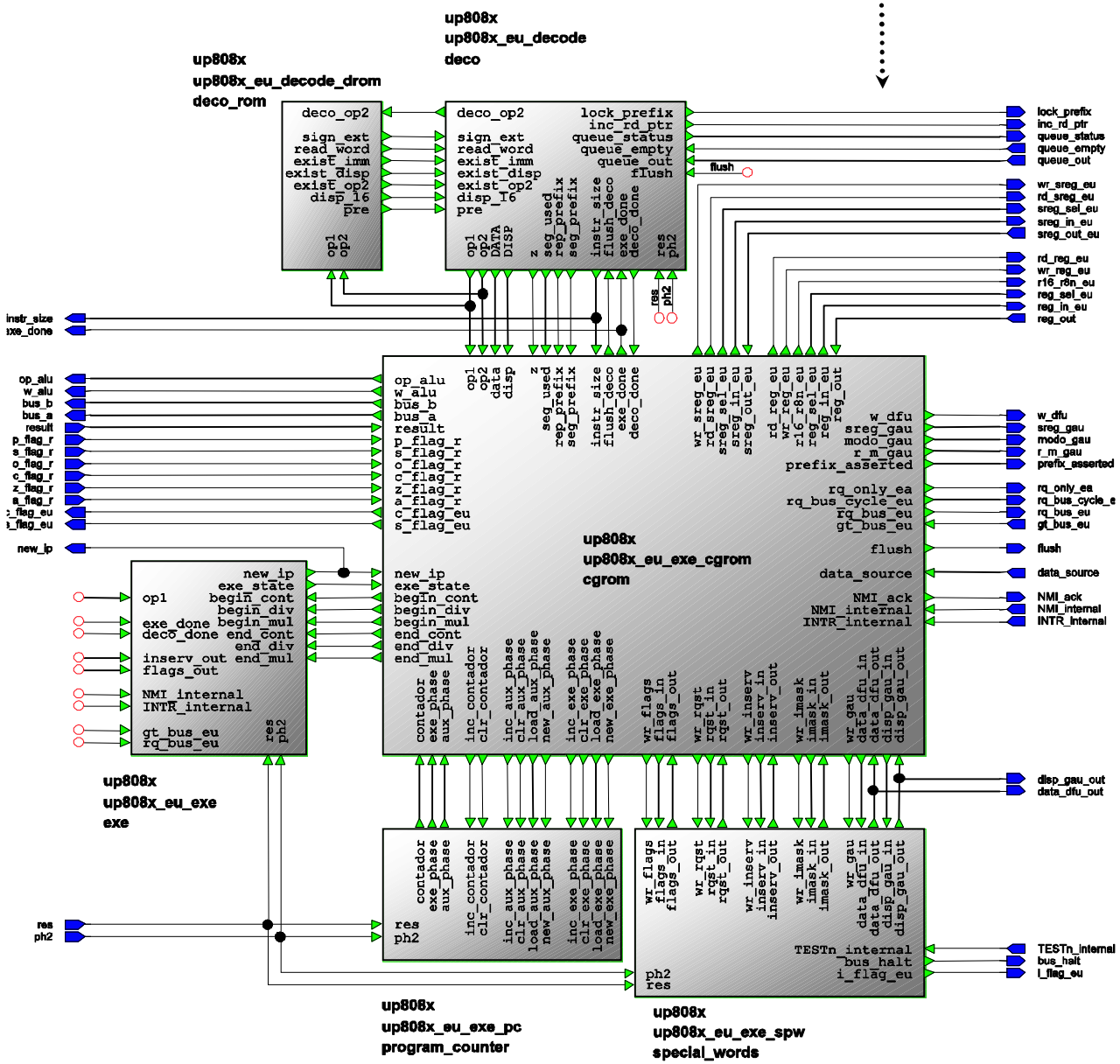


Fig 42.- Jerarquía de la Unidad de Ejecución. Detalle de la Figura 17.

n1/r4, Unidad de Ejecución (Execution Unit, EU): La unidad de ejecución es la encargada, como su nombre indica, de ejecutar las instrucciones. Es idéntica en los micros 8088/8086. Las instrucciones de la cola, son decodificadas por la unidad de decodificación (deco n2/r5), los datos decodificados son utilizados, junto con el estado de la unidad de ejecución (exe n2/r2) y la fase de ejecución o contador de programa, Program Counter (pc n2/r3), como vector de entrada en la Code Generator ROM (cgrom n2/r1) que programa todo el sistema. La CGROM contiene la secuencia de pasos (o microinstrucciones) necesarios para llevar a cavo la lógica (de la instrucción en ejecución) completamente. La unidad lógico aritmética (Arithmetic Logic Unit, alu n2/r6) se encarga de realizar las operaciones de origen lógico-arithmético indicadas por la microinstrucción en curso.

Como se puede ver en la rama de la EU en la figura 42, la unidad de ejecución es íntegramente la misma para los modelos 8088 y 8086 por lo que se encuentra en la librería común up808x. El archivo "up808x_eu.vhd" define un esquema (como indica el icono en el interior del círculo), esquema que podemos ver en la siguiente figura:



A simple vista podemos apreciar todas las unidades que componen la unidad de ejecución a excepción de la ALU que como se comentó anteriormente, aunque pertenece a la unidad de ejecución, la bajé de nivel para que el esquema del microprocesador (n0/r1) reflejara de una forma más real la estructura interna, ver el diagrama de bloques general en la figura 16.

A grandes rasgos, podemos decir que la unidad de decodificación (deco_rom+deco_ctrl) que estudiaremos más adelante. Decodifica los códigos que obtiene de la cola de instrucciones y le pasa la información a la cgrom que junto con el estado generado por la unidad de ejecución (exe) y utilizando los registros especiales (spw) donde guardamos información del estado interno, y teniendo en cuenta la fase actual dentro de la ejecución de la instrucción (controlada por la unidad pc), programa todo el sistema efectuando las micro operaciones con la alu y comunicándose con la biu en la secuencia programada en la cgrom. No se si ha quedado claro el funcionamiento de la eu, pero a medida que nos adentremos en el funcionamiento de las unidades, despejaremos algunas dudas.

La carcasa de la unidad de ejecución, la podemos ver en la siguiente figura:

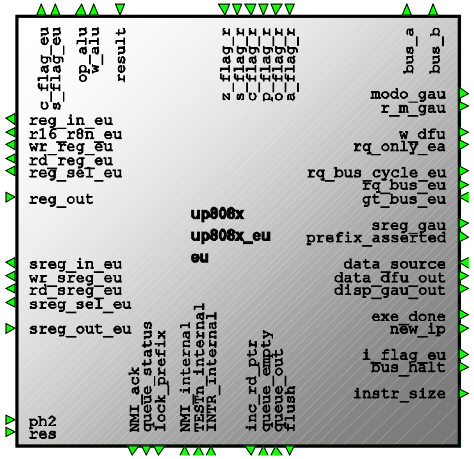


Fig 44.- Carcasa de la Unidad de Ejecución.

14.1 ENTRADAS Y SALIDAS

SEÑAL	TIPO	FUNCIÓN
ph1, ph2	IN	Reloj interno
res	IN	Es la señal interna que resetea todos los componentes del modelo
flush	OUT	Cuando hay un salto la cgrom activa esta señal.
deco->puertos		
lock_prefix	OUT	Indica a la unida ports que la instrucción actual tiene activado el prefijo de bloqueo (LOCK)
deco<->queue		
inc_rd_ptr	OUT	Incrementa el puntero de lectura de la cola de instrucciones.
queue_status	OUT	Información del dato que acaba de ser leído de la cola de instrucciones.
queue_empty	IN	La cola de instrucciones indica al decodificador que no tiene bytes disponibles.
deco -> biu		
instr_size	OUT	Indica a la unidad ifu (n3/r5) el número de bytes de la instrucción decodificada.
cgrom->biu		
exe_done	OUT	Indica que la unidad de ejecución ha terminado secuencia de microinstrucciones del OP en ejecución
W_dfu	OUT	Indica a la unidad dfu (n3/r4) si la transferencia va a ser de byte (si es '0') o palabra (si es'1').
sreg_gau	OUT	Transmite el prefijo utilizado para la generación de la dirección física
modo_gau	OUT	Programa el modificador me la unidad generadora de direcciones.
R_m_gau	OUT	Transmite este campo necesario para que gau que junto con modo indican el modo de direccionamiento.
prefix_asserted	OUT	Indica a la gau que no utilice el segmento implícito porque hay activado un prefijo de segmento en la instrucción.
rq_only_ea	OUT	EU pide a la biu que le pase la dirección efectiva EA.
rq_bus_cycle_eu	OUT	La EU pide a la unidad que va en busca de datos (dfu, n3/r4) que realice el ciclo de bus indicado en esta señal.
rq_bus_eu	OUT	EU solicita el control del bus al árbitro de canal
gt_bus_eu	IN	El árbitro concede el bus a la unidad de ejecución.
data_source	IN	Datos fuente recibidos desde el exterior tras una petición.
data_dfu_out	OUT	Datos que mandamos a la dfu para que los saque cuando corresponda.
disp_gau_out	OUT	Indicamos a la gau el desplazamiento con el tiene que calcular la dirección física (PA).
cgrom<->puertos		
NMI_ack	OUT	Borra el registro de detección de interrupción no mascarable que se encuentra en ports.
NMI_internal	IN	Cuando ports detecta una petición de interrupción válida, activa esta señal.
INTR_internal	IN	Cuando el pin INTR de ports está activo, esta señal también se activa indicando a la eu que existe una petición de interrupción mascarable.
cgrom <-> sreg		
wr_sreg_eu	OUT	Indica que se va a escribir en los registros de segmento, sru (n1/r2)
rd_sreg_eu	OUT	Se va a escribir en sru.
sreg_sel_eu	OUT	Selección del registro con el que se va a operar,
sreg_in_eu	OUT	Dato que vamos a escribir en sru
Sreg_out_eu	IN	Dato que obtenemos de sru.
cgrom <-> rgu		
R16_r8n_eu	OUT	Indica a la unidad de registros de uso general, rgu n1/r1.
Wr_reg_eu	OUT	Indica que se va a escribir en los registros de uso general, rgu (n1/r1)
rd_reg_eu	OUT	Se va a escribir en rgu.
reg_sel_eu	OUT	Selección del registro con el que se va a operar,
reg_in_eu	OUT	Dato que vamos a escribir en rgu
reg_out	IN	Dato que obtenemos de rgu.
exe->biu		
new_ip	OUT	Indica a la unidad ifu (n3/r5) que ha cambiado el puntero de Instrucciones, (que lo tenga en cuenta para calcular el puntero real)

Tab 26.- Descripción de los Pines de la Unidad de Ejecución.

15. UNIDAD LÓGICO ARTIMÉTICA, ALU (n2/r6)

n2/r6 Unidad Lógico Aritmética, Arithmetic Logic Unit (ALU): Contiene la lógica combinacional necesaria para realizar operaciones lógico y aritméticas con datos de 8 o 16 bits. Es utilizada exclusivamente por EU y programada desde la CGROM.

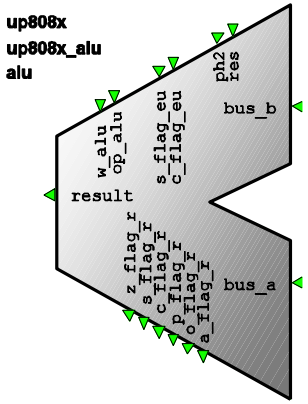


Fig 44.- Carcasa de la Unidad de la ALU.

15.1 ENTRADAS Y SALIDAS

En la figura anterior podemos ver el símbolo, en el que se pueden distinguen tres tipos de señales:

- Los buses de datos (bus_a y bus_b) y resultado (result), ambos de 16 bits. Dependiendo del tipo de instrucción serán utilizados uno (bus_a) o los dos buses de datos que contendrán o contendrá el operando en cuestión. Si w_alu='0' lo que implica operaciones de 8 bits, el byte más significativo de los datos de entrada es ignorado.
- Los Flags. Todas las operaciones de la ALU modifican el estado de estos flags. La unidad de ejecución se encarga de salvarlos en spw (n2/r4) según le convenga o los utiliza en la lógica interna de ejecución de la instrucción actual. También la UE debe programar el estado inicial de ciertos flags (s_flag_eu/c_flag_eu) en operaciones de desplazamientos aritméticos de 16 bits o en sumas con carry corridas (o acumulativas) de datos de 16 bis.
- Selector de operación. Las señales op_alu y w_alu controlan el tipo de operación que va a realizar la ALU `(op_alu) y si se va a operar con palabras (w_alu='1') o bytes (w_alu='0').

Internamente la ALU tiene un registro de 16 bits que memoriza el resultado.

15.2 OPERACIONES

La señal op_code determina el tipo de operación a realizar por la ALU. Este modelo puede realizar 20 operaciones con 8 o 16 bits. Para simplificar la programación de la cgrom he utilizado un tipo de dato enumerado definido en el package del sistema:

```
--Tipo de opreación que realiza la ALU:
TYPE OP_Alu_Typ IS (no_operation,
                    pasar_a,
                   --Operaciones Aritméticas (Sumador)
                    sumar,
                    sumar_carry,
                    restar,
                    restar_borrow,
                    incrementar,
                    decrementar,
                    invertir,
                   --Operaciones Aritméticas
                    negar,
                    desplazar_der_arit,
                   --Operaciones Lógicas
                    desplazar_der_logic,
                    desplazar_izda,
                    rotar_der,
                    rotar_der_carry,
                    rotar_izda,
                    rotar_izda_carry,
                    and_logica,
                    or_logica,
                    xor_logica);
```

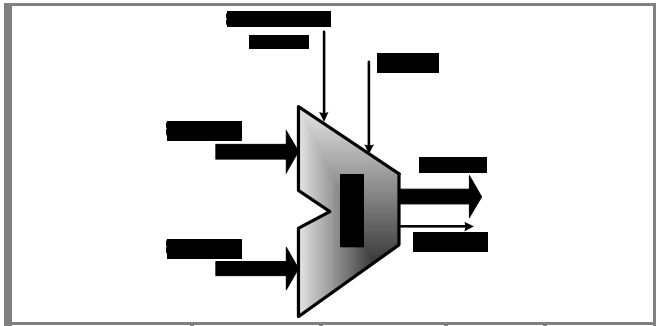
Operaciones Contextuales

Estas operaciones son lógicas dentro del contexto de programación de las microinstrucciones en la cgrom. El nombre es bastante descriptivo de la operación que realiza, así "no_operation" (valor por defecto) no realiza ninguna operación.

Pasar_a, sin embargo es muy utilizada en la microprogramación, lo único que hace es pasar el dato del bus al bus de resultado registrándolo en el registro de salida de la ALU. Podemos utilizar este registro como un registro acumulador (o registro auxiliar) alternativo pudiendo salvar cualquier dato en él.

Operaciones Aritméticas que utilizan el Proceso Sumador/Restador

Internamente la ALU está formada por cuatro procesos (como veremos un poco más adelante), uno de ellos es un Sumador/Restador que es programado desde el proceso principal. Estas instrucciones programan el proceso sumador como podemos ver en la siguiente tabla:



Operación	Bus_a_add	Bus_b_add	C a 2	Carry_IN
Sumar	Bus_a	Bus_b	'0'	'0'
Sumar_carry	Bus_a	Bus_b	'0'	'1'
Restar	Bus_a	Bus_b	'1'	'0'
Restar_borrow	Bus_a	Bus_b	'1'	'1'
Incrementar	Bus_a	X"0001"	'0'	'0'
Decrementar	Bus_a	X"0001"	'1'	'0'
Invertir	X"0000"	Bus_a	'1'	'0'

Tab 27.- Tabla de Programación del Sumador Interno de la Alu.

Operaciones Aritméticas y Lógicas

Estas operaciones pese a la naturaleza aritmética de alguna de ellas, no utilizan el sumador, simplemente utilizan lógica combinacional y de desplazamiento.

Las instrucciones lógicas and_logica, or_logica y xor_logica junto con la operación artmética de negación, realizan las operaciones lógicas AND, OR, XOR y NOT respectivamente.

En la siguiente tabla podemos ver el funcionamiento de las instrucciones lógicas que realiza la ALU:

OPERACIÓN	LÓGICA
Desplazar_der_arit	
Desplaza el dato a hacia la derecha propagando el signo de entrada. Mantiene el signo aritmético.	
Desplazar_der_logic	
Desplaza el dato en el bus_a hacia la derecha insertando '0' por la izquierda.	
Desplazar_Izda	cf Bus_a 0
Desplaza el contenido del bus_a hacia la Izquierda insertando '0' por la derecha. Notar que no existe desplazamiento lógico hacia la izquierda.	
Rotar_der	
Rota el dato en el bus_a hacia la derecha salvando el bit que insertamos por la derecha en el carry de salida.	
Rotar_der_carry	
Desplaza el contenido del dato en el bus_a hacia la derecha e inserta en el huecho libre el contenido del acarreo de entrada. El bit desplazado es salvado en el carry de salida.	
Rotar_Izda	
Rota el dato en el bus_a hacia la izquierda salvando el bit que insertamos por la izquierda en el carry de salida.	
Rotar_Izda_Carry	
Desplaza el contenido del dato en el bus_a hacia la izquierda e inserta en el hueco libre el contenido del acarreo de entrada. El bit desplazado es salvado en el carry de salida.	

Tab 28.- Tabla de la Lógica Utilizada por las Operaciones Lógicas de la ALU.

15.3 PROCESOS

Como puede verse en el árbol Jerárquico la alu está diseñada en VHDL y el fichero que la contiene es de la librería común, "up808x_alu.vhd".

En este fichero podemos encontrar los cuatro procesos que modelan la ALU, tres de ellos son combinacionales y uno (el que registra el resultado de salida) síncrono.

En la siguiente tabla podemos ver una relación de los procesos y su función:

PROCESO	TIPO	FUNCIÓN
ALU	Combinacional	Proceso Principal, en el se realiza la lógica de las operaciones, se programa el sumador y se asigna el resultado.
Adder	Combinacional	Es el Sumador de 16 bits de la ALU. Programado desde el proceso principal, realiza las operaciones aritméticas.
Flags	Combinacional	Proceso de actualización de los flags de salida.
Result_reg	Síncrono	Registra el resultado de una operación realizada por la ALU

Tab 29.- Procesos de la Unidad Lógico Artmética (fichero "up808x_alu.vhd").

Quizás lo más interesante de todo el código esté en el procedimiento parametrizable que encontramos en el proceso adder. Bajo estas líneas podemos ver su código.

Este procedimiento es utilizado repetidas veces dentro del proceso adder para generar las sumas intermedias que necesitamos para calcular todos los acarreos Hay que calcular los acarreos en los bits 4, 7, 8, 15 y 16 para actualizar correctamente los flags de salida de carry y overflow en operaciones con 8 o 16 bits.

Este procedimiento modela un sumador clásico (cascada de sumadores de 1 bit con propagación de acarreo). Este sumador es bastante lento lo que hace que la ALU sea una de las unidades (junto con cgrom) que más retardo introduce en el sistema debido a su gran número de puertas (o lógica combinacional) intermedia.

Para solucionar este problema, es posible sustituir este procedimiento por otro que modele un sumador de acarreo adelantado (Ver los Apuntes de Microelectrónica de Tomás Pollán). La lógica combinacional intermedia se reduce al modelar sumadores de dos bits con una lógica de acarreo más rápida. Ver el Capítulo de Mejora de Velocidad de los Modelos.

Volviendo al procedimiento utilizado, podemos ver como se suman los datos si la entrada sub (o complemento a dos) es '0' o como estos se restan (cuando sub='1') sumando a data_a el complemento a dos de data_b. Se crean 3 variables de 1 bit más que los datos de entrada, b_i que contiene el dato de entrada b incrementado, full_carry que contiene la lógica del carry de entrada con la operación y res_i que es el resultado incrementado. •••••••••••••••••

16. UNIDAD EXE (n2/r2)

n2/r2 Máquina de Ejecución (exe): Es una máquina de estados que determina cómo se encuentra la unidad de ejecución. Se puede decir que es la que "comanda" la unidad de ejecución EU.

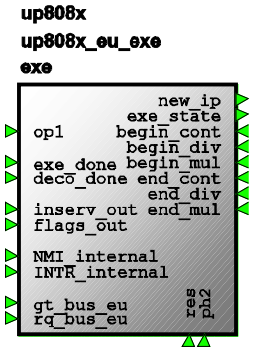


Fig 46.- Señales de Externas de la Unidad exe.

Es un elemento común para todos los modelos y por lo tanto lo encuadramos en la librería común (up808x), en concreto, este autómata está definido en el archivo "up808x_eu_exe.vhd"

16.2 EXPLICACIÓN DEL GRAFO DE ESTADOS

Este grafo de estados está compuesto por 7 estados (como podemos ver en la figura 47), cuatro de ellos están agrupados en un macroestado. Se puede observar que no he asignado codificación a los estados, en esta ocasión, será el sintetizador el que asigne la codificación que le convenga.

Sin embargo, el tipo de estado de este grafo es utilizado por otras unidades (cgrom), por lo que es necesario definirla en el package. De esta manera podemos crear variables con el tipo de estado de la unidad de exe. A continuación podemos ver la definición del tipo de dato en el package:

```
--|eSTADO dE lA uNiDAD dE dECODiFICACióN i eJECUCióN|--
TYPE Exe_State_Typ IS
    (DECO,          --Estado de decodificación del OP
     EXE,           --Fase de ejecución
     TRANSFER,      --Estado de cimunicación con la BIU
     Interrup,      --Estado de Interrupción
     Multiplicar,   --Unidad de Multiplicación
     Dividir,       --Unidad de División
     Cont);         --Contador
```

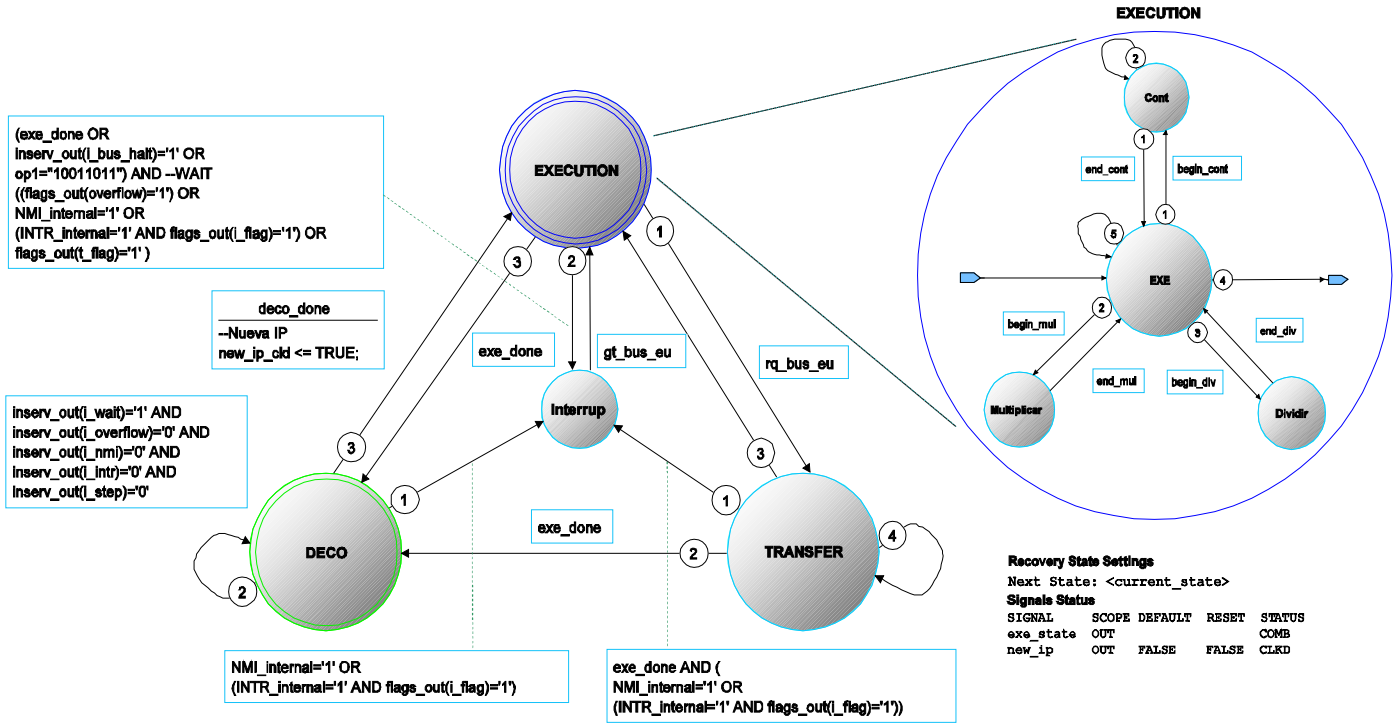


Fig 47.- Grafo de Estados de la Unidad exe.

16.3 ENTRADAS Y SALIDAS

Señal	Tipo	Comunicación	Función
ph2	IN	init -> exe	Reloj interno
res	IN		Señal interna con la que reseteamos el modelo
op1	IN	deco ->exe	Código de Operación (OP) de la instrucción actual.
deco_done	IN		Indica que la decodificación ha terminado.
exe_done	IN	cgrom <-> exe	Indica que la cgrom ha llegado a la última microinstrucción de la secuencia de la operación en curso.
begin_cont	IN		Activa o desactiva el contador Interno..
end_cont			
begin_div	IN		Activa el módulo de división
end_div			
begin_mul	IN		Activa la secuencia de multiplicación.
end_mul			
rq_bus_eu	IN		Petición del bus externo
gt_bus_eu	IN		Ha terminado la operación con el bus. (generada por dfu, n3/r4)
new_IP	OUT		Indica que se ha incrementado el Puntero de Instrucciones
exe_state	OUT		Estado de la unidad de ejecución
liserv_out	IN	spw -> exe	Registro de las interrupciones que están en servicio
flags_out	IN		Registro de banderas.
NMI_internal	IN	ports -> exe	Petición de Interrupción.
INTR_internal	IN		Petición de INTR

Fig 30.- Tabla de Entradas y Salidas de la Unidad exe..

16.4 COMPORTAMIENTO

En el estado inicial (deco), la unidad de decodificación i(deco, n2/r5) interpreta los datos que recoge de la cola de instrucciones. Cuando ha terminado de decodificar la instrucción (señal deco_done=TRUE) y siempre que no haya una petición de interrupción mascarable o no, saltamos al macroestado de ejecución, que indica a la cgrom que puede efectuar la secuencia de microinstrucciones de la instrucción decodificada.

En esta secuencia de microinstrucciones pueden darse 4 situaciones especiales. El más común es que la instrucción requiera una transferencia de datos (necesita un dato de la memoria o I/O o lo tiene que escribir). Es necesario establecer una comunicación con la biu para comunicar la petición, esta petición se realiza a través de las señales rq_bus_eu (generada en el microcódigo) y gt_bus_eu (generada desde la unidad de acceso a datos (dfu, n3/r5). En la gráfica de abajo podemos ver como se sincronizan estas señales.

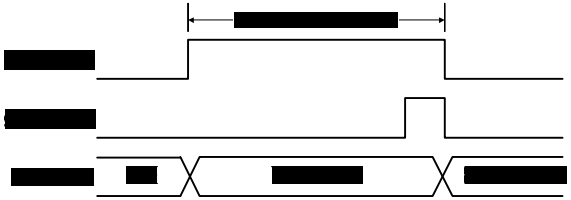


Fig 48.- Diagrama de Tiempos para Realizar una Transferencia. Comunicación EU-DFU.

Permanecemos en el estado TRASNFER hasta que la dfu nos indique que ya ha realizado la transferencia. Si con la operación de transferencia termina la ejecución de la instrucción, saltamos directamente al estado DECO, de esta manera no perdemos un ciclo en ir al estado de EXECUTION y de éste al de Decodificación.

Otra situación especial se da cuando en la lógica de microinstrucciones (estado EXE) nos encontramos con la activación de la señal begin_cont. Esta señal indica que se va a utilizar el contador interno (también microprogramado en la cgrom), por lo que pasamos al estado CONT. Las instrucciones con cadenas, el prefijo REP, la instrucción LOOP e instrucciones de múltiples desplazamientos, utilizan este contador. Salimos del estado de CONT cuando el conteo llega a su fin (end_cont=TRUE).

Los dos estados de MULTIPLICAR y DIVIDIR son activados por las señales begin_mul y begin_div respectivamente y lanzan el algoritmo de multiplicación o división de datos de 8 o 16 bits con o sin signo. Estos algoritmos son utilizados por las instrucciones DIV/IDIV/MUL/IMUL y las de ajuste decimal después de estas AAM/AAD y están microprogramados (por lo que no generan hardware adicional, a excepción de la necesaria para codificar las microinstrucciones) en la cgrom. Los algoritmos de multiplicación y división terminan con la activación de las señales end_mul y end_div respectivamente que devuelven es sistema al estado de ejecución (EXE)

Desde el estado EXE saltamos al estado inicial DECO cuando la cgrom termina la secuencia de microinstrucciones activando la señal exe_done y siempre que la instrucción no termine con una escritura en memoria.

En esta unidad está embebida la unidad de Interrupciones representada por el estado INTERRUP en el que se actualizan los registros de interrupción en servicio (registro de servicios activados inserv_out y registro de servicios con petición de ejecución rqst_out, en spw n2/r4).

Las Interrupciones son atendidas directamente cuando estamos en el estado DECO y condicionadas por la señal exe_done (tenemos que esperar a que la instrucción actual termine de ejecutarse) cuando nos encontramos en los estados EXE o TRANSFER.

16.5 ESTADOS

DEDO (e0): Es el estado inicial, indica que la UE está decodificando una instrucción. Tiene tres transiciones, como podemos ver en el grafo de la figura 47, e0/t1 tiene la condición:

```
NMI_internal='1' OR
(INTR_internal='1' AND flags_out(i_flag)='1')
```

Esta transición hace referencia a la aparición de una interrupción no mascarable (NMI) cuando la UE está decodificando una instrucción. En el caso de aparecer una petición de interrupción mascarable (INTR) en este estado, saltaremos el estado INTERRUP si el flag de interrupción (flags_out(i_flag)) es '1'.

Podemos apreciar como se han declarado constantes para acceder de una forma más fácil a los bits del registro de banderas. Estas declaraciones son utilizadas por varias unidades, de manera que las encontramos definidas en el package del sistema (n0/r0) como podemos ver a continuación:

```
--Flags de Uso General (vistos externamente)
CONSTANT c_flag : Integer := 0; --Flag de Acarreo
CONSTANT p_flag : Integer := 2; --Flag de Paridad
CONSTANT a_flag : Integer := 4; --Flag de CY Auxiliar
CONSTANT z_flag : Integer := 6; --Flag de Zero
CONSTANT s_flag : Integer := 7; --Flag de Signo
CONSTANT t_flag : Integer := 8; --Flag de Trampa
CONSTANT i_flag : Integer := 9; --Flag de Interrupción
CONSTANT d_flag : Integer := 10;--Flag de Dirección
CONSTANT o_flag : Integer := 11;--Flag de Overflow
```

La transición 2 hace que el sistema se pare sólo cuando está en servicio la instrucción WAIT. Recordar que el sistema permanece parado cuando se ejecuta la instrucción WAIT en espera de que TESTn sea '0', pero si aparece alguna interrupción, esta tiene que ser atendida. De esto se encarga a0/t2.

También accedemos a los bits del registro de interrupción en servicio (iserv_out) mediante constantes; éstas se declaran en el package con el código siguiente:

```
 --Flags de Sistema (uso interno)
--Flags de la unidad de interrupción
CONSTANT i_overflow          : Integer := 0;
CONSTANT i_nmi               : Integer := 1;
CONSTANT i_step              : Integer := 2;
CONSTANT i_intr              : Integer := 3;
CONSTANT i_wait              : Integer := 4;
CONSTANT i_pending_bus_halt  : Integer := 5;
CONSTANT i_bus_halt          : Integer := 6;
```

La tercera transición de este estado e0/t3 (la de menor prioridad) salta al estado de ejecución (estado EXE) cuando el decodificador ha decodificado (valga la redundancia) la siguiente instrucción a ejecutar. La señal "deco_done" indica que el decodificador ha terminado.

EXECUTION Macroestado (m1): Es un estado ficticio, en realidad no existe como tal, simplemente sirve para simplificar el grafo aglutinando ciertos

estados reales. En la figura 47 podemos ver como en macroestado m1 “encierra” los estados:

- EXE (e1): Estado que indica a la cgrom que inicie la secuencia de microinstrucciones.
- CONT (e2): Estado activado por el propio microcódigo de la instrucción que activa la secuencia de microinstrucciones que realizan un conteo con el registro CX. Algunas instrucciones con cadenas o de repetición, programan este contador interno en su microcódigo.
- MULTIPLICAR (e3). Como en el caso anterior, este estado “despierta” la secuencia de microinstrucciones que implementa el algoritmo de multiplicación. Las instrucciones MUL/IMUL y AAM programan este algoritmo para realizar multiplicaciones con o sin signo de números de 8 o 16 bits. He implementado en microinstrucciones (sin utilizar hardware específico) una variación del algoritmo de Booth. Más adelante y si dispongo de tiempo explicaré en detalle este algoritmo porque merece la pena.
- DIVIDIR (e4): Cuando nos encontramos en este estado, se activa una secuencia de microinstrucciones en la cgrom que implementa un algoritmo de división. Las instrucciones DIV/IDIV y AAD programan este algoritmo para realizar divisiones con o sin signo de números de 32 o 16 bits. En este caso el algoritmo ha sido íntegramente microprogramado por el diseñador. Sin añadir nuevo hardware al sistema y utilizando solamente la ALU (desplazamientos y restas) he implementado un algoritmo clásico de división que explicaré más adelante si dispongo de tiempo.

Aunque ahora no entremos en detalle en estos algoritmos, si que diremos que utilizan ciertos bits de los registros de uso especial spw, n2/r4, y que para acceder a ello de una forma más sencilla, hay definidos en el package una serie de constantes que utilizamos:

Aunque ahora casi nos de risa, estas instrucciones de multiplicación y división con o sin signo y la posibilidad de hacer un ajuste decimal (AAM/AAD) fueron una razón de fuerza para llevar a los microprocesadores 8088/8086 a la fama y a derrocar la competencia. Ninguno de los competidores de la época implementaba estas instrucciones... ...hoy, no se utilizan.

Desde EXE hay una transición hacia el resto de los estados internos del macroestado, m1. La condición de estas transiciones viene dada desde la cgrom, así e1/t1, e1/t2 y e1/t3; begin_cont, begin_mul y bebin_div respectivamente, son controladas desde las microinstrucciones de la cgrom. El retorno de cont (e2), Multiplicar (e3) y Dividir (e4) a exe (e1) lo controla sus respectiva transición t1, así e2/t1 (end_cont), e3/t1 (end_mul) y e4/t1 (end_div) controlan el retorno a exe.

La transición e2/t4 se desdobla en 3 transiciones (m1/t1, m1/t2 y m1/t3) en el macroestado.

- m1/t1 (rq_bus_eu) es la transición hacia el estado TRANSFER. Como se comentó anteriormente, es la señal que genera la cgrom para pedir una transferencia a la unidad dfu (n3/r4). En la figura 48 podemos apreciar como se comunican estas dos unidades.
- m1/t2 es la transición hacia es estado de interrupción. Cuando durante la secuencia de microinstrucciones aparece una de las condiciones de la transición m1/t2, saltamos al estado de interrupción que se encarga de actualizar el registro de interrupción en servicio (inserv_out), según la petición, el estado actual de las interrupciones y la prioridad. Analizando la transición m1/t2, cuyo código podemos ver bajo este párrafo, podemos ver que son atendidas al final de la secuencia de microinstrucciones de la ejecución en proceso, momento en el que exe_done=TRUE; o cuando el sistema esta parado (inserv_out(i_bus_halt), activado por la instrucción HLT; o cuando la instrucción en ejecución es la instrucción WAIT. En estas condiciones, si hay un desbordamiento en las instrucciones de división flags_out(overflow), una interrupción no mascarable NMI_internal, una interrupción mascarable estando activado el flag de interrupción (INTR_internal=’1’ AND flags_out(i_flag)=’1’) o está activado el flag de Trampa flags_out(t_flag), en estas condiciones se salta a INTERRUP.

```
(Exe_done
OR inserv_out(i_bus_halt)=’1’
OR op1=”10011011”) --WAIT
  AND
((flags_out(overflow)=’1’)
OR MNI_internal=’1’
OR (INTR_internal=’1’ AND flags_out(i_flag)=’1’)
OR flags_out(t_flag)=’1’)
```

- m2/t3, es la transición hacia una nueva decodificación. Cuando la secuencia ha terminado (exe_done=TRUE) y no se da ninguna de las transiciones m1/t1 y m1/t2, saltamos al estado e0.

TRANSFER (e5): Es el estado en el que permanecemos hasta que se realiza la transferencia con la memoria a través de dfu (ver la figura 48). Mientras permanecemos en este estado, la cgrom sigue operando siempre que pueda y con un máximo de 5 ciclos de reloj (que es el mínimo tiempo de transferencia), de esta manera adelantamos ciclos en la secuencia de ejecución. Para diferenciar entre el “paso” en el estado EXE (e2) y la secuencia en este estado, necesitamos 2 fases o contadores de programa, los he llamado exe_phase (controla la fase o paso de e2) y aux_phase que controla la fase del estados auxiliares (CONT, MULTIPLICAR, DIVIDIR y TRANSFER). Todo esto se explica con más detalle en la sección del contador de programa (PC, n2/r3)

e5/t2 representa las instrucciones que terminan con una transferencia. En este caso no es necesario pasar por EXE para ir a DECO.

e5/t1 cuando aparece una interrupción mascarable o no yl la ejecución ha terminado, hay que saltar a e6.

INTERRUP (e6): Es el estado que actualiza el servicio de interrupción, sólo tiene una transición, e6/t1 que es siempre TRUE, por lo que saltamos inmediatamente a EXE para atender la Interrupción en servicio.

```
--Utilizado en DIV/IDIV/AAD, estado e4
CONSTANT s_Y      : Integer := 2;  --Signo del Dividendo (Y)
CONSTANT s_D      : Integer := 4;  --Signo del Divisor   (D)
CONSTANT overflow : Integer := 5;  --Desbordamiento del cociente (Q)
CONSTANT first    : Integer := 11; --Primera pasada
--Utilizado en Instrucciones de String y de Desplazamiento. Estado e2.
CONSTANT zero       : Integer := 13; --Indica que CX='0'
--Utilizados en MUL/IMUL/AAM. Estado e3.
--Podemos Utilizar los Flags de Uso General. Al final de mul/imul quedarán indeterminados.
CONSTANT lsb_q      : Integer := 0; --Bit Menos Significativo del Multiplicador.
CONSTANT last_lsb_q : Integer := 2; --Último Bit Menos Significativo del Multiplicador.
CONSTANT high_data  : Integer := 4; --Existen Datos en la Parte Alta del Resultado.
```

17. CONTADOR DE PROGRAMA, PC (n2/r3)

n2/r3 Contador de Programa, Program Counter (PC): Esta unidad incrementa el vector de entrada a la CGROM que se encarga de saltar a la siguiente microinstrucción dentro de la secuencia de la instrucción en ejecución.

17.1 FUNCIONAMIENTO

En realidad dentro de este contador de programa existen tres contadores como podemos ver en la siguiente tabla:

NOMBRE	FUNCIÓN	TIPO DE CONTADOR
contador	La secuencia de microinstrucciones que implementan los algoritmos de multiplicación/división con/sin signo utilizan este contador para realizar el "barrido" del bucle principal.	
exe_phase	Es la fase de ejecución principal de la unidad de ejecución.	
aux_phase	Fase de ejecución auxiliar, utilizada cuando salimos del estado de Ejecución EXE (ver n2/r2).	
CARCASA DEL COMPONENTE		
up808x **up808x_eu_exe_pc** **program_counter**	contador exe_phase aux_phase inc_contador clr_contador inc_aux_phase clr_aux_phase load_aux_phase new_aux_phase inc_exe_phase clr_exe_phase load_exe_phase new_exe_phase res ph2	

Tab 31.- Fases de Control de las Secuencias de Microinstrucciones.

Estos contadores actúan como vector de entrada a la cgrom, que junto con el op code de la instrucción actual, apuntan a la microinstrucción que se está ejecutando en ese momento.

Se trata de un componente común a los dos modelos que podemos encontrar en el fichero "up808x_eu_pc.vhd". Los contadores están escritos directamente en VHDL como indica la figura 42..

17.2 ENTRADAS Y SALIDAS

SEÑAL	TIPO	FUNCIÓN
ph2	IN	Reloj interno
res	IN	Es la señal interna que resetea todos los componentes del modelo
Inc_contador	IN	Incrementa el valor de contador.
Clr_contador	IN	Pone a cero el valor de contador
Contador	OUT	Salida de contador utilizado por microcódigo en secuencias auxiliares.
Inc_aux_phase	IN	Incrementa el valor del contador de la fase auxiliar
Clr_aux_phase	IN	Borra el contenido del contador de la fase auxiliar
Load_aux_phase	IN	Hace que el valor en la señal new_aux_phase pase a la salida
New_aux_phase	IN	Contiene el nuevo valor a cargar en el contador de la fase auxiliar
Aux_phase	OUT	Volor del contador de la fase auxiliar, es utilizado como puntero en la secuencia de microinstrucciones en procesos auxiliares
Inc_exe_phase	IN	Incrementa el valor del contador de la fase de ejecución
Clr_exe_phase	IN	Borra el contenido del contador de la fase de ejecución
Load_exe_phase	IN	Hace que el valor en la señal new_exe_phase pase a la salida
New_exe_phase	IN	Contiene el nuevo valor a cargar en el contador de la fase de ejecución
exe_phase	OUT	Volor del contador de la fase de ejecución, es utilizado como puntero en la secuencia de microinstrucciones-

Tab 32.- Explicación de los Terminales de Componente PC.

17.3 PROCESOS

Esta unidad (cuyo código podemos ver a la derecha) no es muy complicada, simplemente tiene dos procesos. Uno para implementar el contador con borrado síncrono (proceso cont) y un segundo proceso (phase) que modela el comportamiento de dos contadores síncronos con borrado y precarga síncrona, los contadores de las fases

```
--|...............................................|--
--|cONTADOR dE bUCLE eN eL aLGORiTMO dE
MUL/IMUL/DIV/IDIV
    cont : PROCESS (ph2,res) IS
    BEGIN
        IF res THEN --res asincrono
        --Valores de Reset
            contador_cld <= 0;
        ELSIF ph2'EVENT AND ph2='1' THEN
            IF clr_contador THEN
                contador_cld <= 0;
            ELSIF inc_contador THEN
                contador_cld <= contador_cld+1;
            ELSE
                contador_cld <= contador_cld;
            END IF;
        END IF;
    END PROCESS cont;

    --Asignación de las salidas
    contador <= contador_cld;

--|...............................................|--
--|fASES dE eJECUCióN, cONTADOR dE MiCROiNSTRUCCiONES|--
    phase : PROCESS (ph2,res) IS
    BEGIN
        IF res THEN --res asincrono
        --Valores de Reset
            aux_phase_cld <= 0;
            exe_phase_cld <= 0;
        ELSIF ph2'EVENT AND ph2='1' THEN
            --Contador de la Fase Auxiliar
            IF clr_aux_phase THEN
                aux_phase_cld <= 0;
            ELSIF load_aux_phase THEN
                aux_phase_cld <= new_aux_phase;
            ELSIF inc_aux_phase THEN
                aux_phase_cld <= aux_phase_cld+1;
            ELSE
                aux_phase_cld <= aux_phase_cld;
            END IF;

            --Contador de la Fase de Ejecución
            IF clr_exe_phase THEN
                exe_phase_cld <= 0;
            ELSIF load_exe_phase THEN
                exe_phase_cld <= new_exe_phase;
            ELSIF inc_exe_phase THEN
                exe_phase_cld <= exe_phase_cld+1;
            ELSE
                exe_phase_cld <= exe_phase_cld;
            END IF;
        END IF;
    END PROCESS phase;

    --Asignación de las salidas
    aux_phase <= aux_phase_cld;
    exe_phase <= exe_phase_cld;
```

18. REGISTROS ESPECIALES, SPW (n2/r4)

n2/r4 Registros Especiales, SPecial Words (SPW): Contiene registros con un significado "especial" o concreto en la lógica de ejecución de UE. Uno de estos registros es el famoso registro de banderas (que forma parte del modelo software (ver modelo software). También incluye una serie de registros que dan soporte a la lógica de interrupciones, además de (l/d)os registros de comunicación con la BIU.

18.1 ENTRADAS Y SALIDAS

SEÑAL	TIPO	FUNCIÓN
ph2	IN	Reloj interno
res	IN	Es la señal interna que resetea todos los componentes del modelo
wr_flags	IN	Escribe el registro de banderas
flags_in	IN	Valor a escribir en el registro de banderas.
flags_out	OUT	Salida del registro de Banderas
wr_rqst	IN	Escribe en el registro de petición de Interrupción
rqst_in	IN	Dato a escribir en el registro de petición de interrupción.
rqst_out	OUT	Salida del registro de petición de interrupción.
wr_inserv	IN	Escribe en el registro de interrupción en servicio
Inserv_in	IN	Dato a escribir en el registro de interrupción en servicio.
inserv_out	OUT	Salida del registro de interrupción en servicio.
wr_imask	IN	Escribe en el registro de máscara de interrupciones
mask_in	IN	Dato a escribir en el registro de máscara de interrupciones
imask_out	OUT	Salida del registro de máscara
wr_gau	IN	Escribe en los registros de comunicación con la BIU
data_dfu_in disp_gau	IN	Datos a escribir en data_dfu y disp_gau respectivamente.
data_dfu_out disp_gau	OUT	Salidas de los registros data_dfu.y disp_gau respectivamente
TESTn_internal	IN	Esta entrada borra directamente el bit de "en servicio WAIT" del registro inserv.
bus_halt	OUT	Es el bit del registro de interrupción en servicio que indica que el sistema está parado (halt)
i_flag_eu	OUT	Es una imagen del fit de "interrupción habilitada" del registro de banderas.

Tab 34.- Explicación de los Terminales de Componente SPW.

18.2 FUNCIONAMIENTO

Este componente , único para los dos modelos de microprocesadores, programado en el archivo de la librería común "up808x_eu_spw.vhd". Contiene internamente tres tipos de registros, como podemos ver en la figura 49:

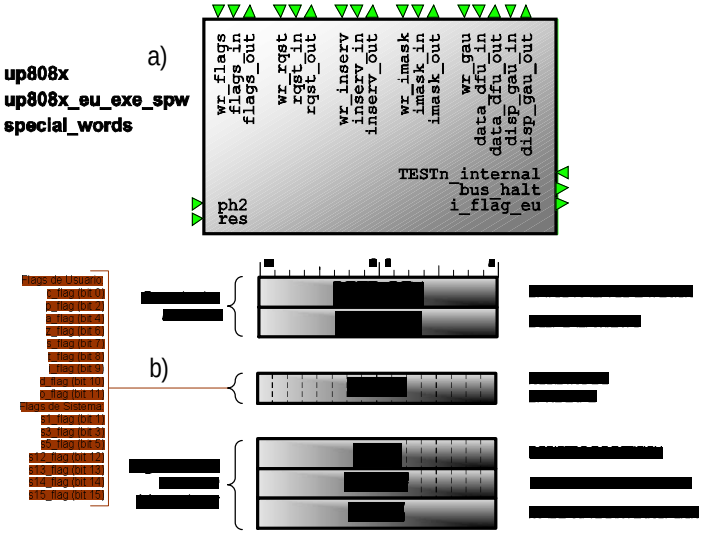


Fig 49.- a) Carcasa del componente SPW definida en "up808x_eu_spw.vhd" y b) Registros Internos Especiales .

La función de los registros es:

NOMBRE	FUNCIÓN
data_dfu	Registra los datos (16 u 8 bits) que la unidad dfu (n3/r4) pondrá en el bus externo.
disp_gau	Memoriza el desplazamiento que la unidad generadora de direcciones, gau (n3/r7), utilizará para crear la dirección física PA.
flags	Registro de banderas que contiene los bits de información del sistema (ver la figura 49) algunos son vistos por el usuario y forman parte del modelo software (ver modelo software) y otros son utilizados por el propio micro para tareas internas.
rqst	Este registro que forma parte de un unidad de interrupción, activa uno de sus bits cuando es aceptada una petición de interrupción.
inserv	Este registro tiene asignado un bit para cada interrupción en servicio.
mask	Registro de Máscara de las interrupciones. Permite enmascarar las interrupciones

Tab 33.- Registros Especiales en SPW.

Registros De Comunicación Con La BIU

Existen dos registros de comunicación con la BIU, el resto de la información es enviada por la CGROM:

- Data_dfu. Este registro de 16 bits se comunica directamente con la unidad encargada de ir en busca de datos a la memoria, dfu (n3/r4) perteneciente a la BIU, y contiene el dato que va a ser enviado. Si la transferencia es de 1 byte (w_dfu='1'), la unidad dfu genera un solo ciclo de bus y obvia el byte alto de este registro.
- Disp_gau. Este registro se comunica con la unidad de la BIU encargada de generar la dirección física, gau (n3/r7). El contenido de este registro de 16 bits indica a gau el desplazamiento que debe utilizar para la generación de la dirección física (Phisical Address, PA). Si el desplazamiento es de 8 bits (indicado por el modificador mod) la unidad gau ignora la parte alta de este registro.

Estos registros se escriben cuando la cgrom activa la señal wr_gau.

Registro de Banderas

Este registro de 2 bytes, cuyos bits son accesibles uno a uno para escritura y lectura, contiene dos tipos de bits:

- Bits de Usuario. Estos son los bits con significado externo (e interno claro está), el usuario puede interpretarlos y modificarlos actuando sobre el comportamiento del microprocesador. Forman parte del modelo software que el usuario tiene del microprocesador y, por lo tanto, se explican en detalle en la sección que explica el modelo software.
- Bits de Sistema. En el flujo de microinstrucciones, alguna instrucción puede modificar estos bits para almacenar información del estado interno. Ocupan los huecos dejados por los bits de bandera de usuario y aunque algunos no son utilizados en ninguna instrucción, cabe la posibilidad de utilizarlos en futuras evoluciones del modelo.

Un ejemplo de utilización de un bit de sistema se encuentra en el código de microinstrucciones de las instrucciones que utilizan el contador. En estas tenemos que detectar cuando el registro contador

(CX) es ‘0’ y no podemos utilizar ningún flag de usuario (como podemos ver en el apéndice del juego de instrucciones), en este caso utilizamos el flag de sistema número 13, que en el package se ha definido con la constante “zero”, (ver el código de abajo):

```
--Utilizado en Instrucciones de String y de Desplazamiento
CONSTANT zero : Integer := 13; --Indica que CX='0'
```

La señal de salida i_flag_eu es el bit de usuario i_flag. Este bit es reflejado externamente vía hardware cada ciclo de bus en la señal S5. La unidad ports (n1/r6) junto con el control proporcionado por la unidad generadora de ciclos de bus, bcg (n3/r)), se encargan de multiplexarla en el tiempo. Én el código al final de este capítulo, podemos ver como es asignada esta salida.

Registros de la Unidad de Interrupciones

La unidad de interrupciones que implementan los microprocesadores 8088/8086, es un poco precaria. No permite controlar el tipo de activación de la interrupción y sólo dispone de dos interrupciones externas (INTR activada por nivel y NMI activada por Flancos).

En previsión de futuras evoluciones de los modelos, he implementado la lógica de registros diseñada para los microprocesador 80188/80186. (ver “80C186EA-80C 188EA Microprocessor User's Manual.pdf”)

Los registros de la unidad de interrupciones son:

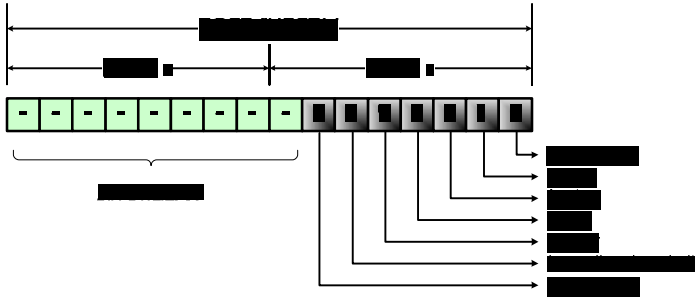


Fig 50.- Significado de los Bits de los Registros de Interrupción..

- Rqst. (Request). Este registro almacena las peticiones de interrupción generadas en la propia instrucción (excepciones internas, como el desbordamiento en una división, la activación del flag de trampa t_flag), las instrucciones software (WAIT, HALT o INT), y las instrucciones externas (NMI e INTR). En la figura 50 podemos ver cómo están organizadas en los registros.

- Inserv. Registro que nos informa de la interrupción que actualmente está en servicio. Este registro es actualizado cuando saltamos al estado INTERUPT (ver la explicación de la unidad exe, n2/r2). Par actualizarlo se tiene en cuenta las peticiones activadas (registro rqst pasado por la máscara de interrupciones, registro mask), las interrupciones que actualmente están en ejecución, si las hay (registro inserv), y una lógica de prioridad. La lógica de prioridad podríamos enuciarla como “No se atenderán interrupciones de prioridad menor o igual que la mayor de las prioridades de las interrupciones en servicio”. La prioridad de las interrupciones las podemos ver en la siguiente figura:

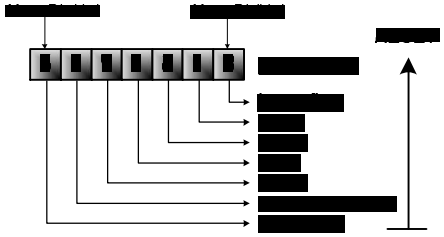


Fig 51.- Prioridades de la Interrupciones y Excepciones.

- Mask. Este registro sería la máscara o registro de habilitación de las interrupciones mascarables. En los microprocesadores 8088/8086 sólo hay una interrupción mascarable (INTR), por lo que este registro sólo utiliza el bit 3, que en realidad es el flag de habilitación de interrupción i_flag..

Para manejar de una forma más simple los registros de la unidad de interrupciones, se han creado unas constantes en el package del sistema:

```
--Flags de Sistema (uso interno)
--Flags de la unidad de interrupción
CONSTANT i_overflow         : Integer := 0;
CONSTANT i_nmi              : Integer := 1;
CONSTANT i_step             : Integer := 2;
CONSTANT i_intr             : Integer := 3;
CONSTANT i_wait             : Integer := 4;
CONSTANT i_pending_bus_halt : Integer := 5;
CONSTANT i_bus_halt         : Integer := 6;
```

18.3 PROCESOS

Es una unidad muy sencilla, sólo tiene un proceso síncrono que registra las señales de entrada y una serie de asignaciones concurrentes que asignan las salidas.

```
--|...................................................|--
--|PrOCEDiMiENTO SíNCRONO dE rEGiSTRO                 |--

    special_word: PROCESS (ph2,res) IS

    BEGIN
        IF res THEN --res asincrono

            --Registros de Comunicación Interna
            data_dfu <= X"0000";
            disp_gau <= X"0000";

            --Registros especiales
            rqst <= X"0000";
            inserv <= X"0000";
            imask <= X"0000";

            --Registro de banderas
            flags <= X"0000";

        ELSIF ph2'EVENT AND ph2='1' THEN
        IF wr_gau  THEN data_dfu <= data_dfu_in;
                        disp_gau <= disp_gau_in; END IF;
        IF wr_flags  THEN flags  <= flags_in;    END IF;
        IF wr_rqst   THEN rqst   <= rqst_in;     END IF;
        IF wr_inserv THEN
            inserv   <= inserv_in;
            IF TESTn_internal='0' THEN
               inserv(i_wait) <= '0';
            END IF;
        END IF;
        IF wr_imask  THEN imask  <= imask_in;    END IF;
        END IF;
    END PROCESS special_word;

--|...................................................|--
--|aSiGANACiONES dE lAS sALiDAS                       |--

    --Registros hacia la BIU (Bus Interface Unit)
    data_dfu_out <= data_dfu;
    disp_gau_out <= disp_gau;

    --Registros de la unidad de interrupción
    rqst_out   <= rqst;   --Registro de la interrupción
    inserv_out <= inserv; --Interrupción en servicio
    imask_out  <= imask;  --Máscara de interrupción

    --Vector de flags
    flags_out <= flags;

    --Asignación de bus_halt
    bus_halt <= TRUE   WHEN inserv(i_bus_halt)='1'  ELSE
                FALSE;

    --Saco el estado del flag de interrupción
    i_flag_eu <= flags(i_flag);
```

n2/r1 Code Generator ROM (CGROM): Contiene la lógica combinacional que, en función de los datos suministrados por el decodificador, la unidad de ejecución y el contador de programa, genera las señales que programan las unidades internas para realizar una determinada operación. Una secuencia consecutiva de estas operaciones intermedias (o microinstrucciones) culmina con la ejecución de la instrucción decodificada.

En realidad la cgrom es la lógica combinacional del grado de estados de la unidad exe (n2/r2), decidí aislarlo para "limpiar" un poco la unidad de ejecución dejando de esta manera la parte síncrona en exe y llevando a cgrom toda la lógica combinacional.

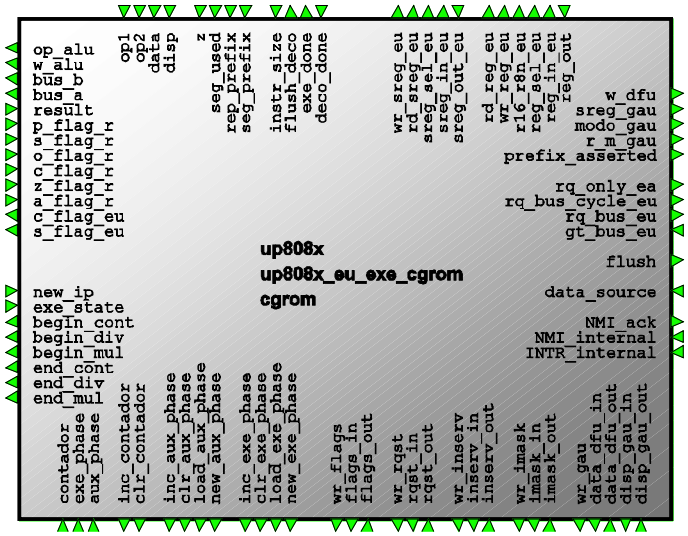


Fig 52.- Terminales de la CGROM (fichero "up808x_ue_cgrom.vhd").

El fichero de la cgrom común a los dos modelos "up808x_eu_exe_cgrom.vhd", contiene unas 7000 líneas de código. Este código, ocuparía unas 100 páginas como ésta, de manera que no lo voy a insertar en este documento, el fichero siempre está ahí para consultarlo. Lo que si voy a hacer, es insertar algún ejemplo de instrucción microprogramada

19.1 PROCESOS

Existen dos únicos procesos en el fichero que define la cgrom:

PROCESO	TIPO	FUNCIÓN
Code_Generator _ROM	Combinacional	Es el proceso principal que implementa el grueso de las microinstrucciones .
Demux_OP	Combinacional	Demultiplexa la información contenida en los bytes de Operation Code (OP) proporcionados por el decodificador

Tab 35.- Procesos de la Code Generador ROM (fichero "up808x_eu_cgrom.vhd").

PROCESO CODE_GENERATOR_ROM

El Proceso Code_Generator_ROM es el proceso principal. En función del OP de la instrucción a ejecutar y del estado de la unidad de ejecución (exe_state), apuntamos una parte específica de la cgrom, se trata de la secuencia de microins-trucciones que hay que "recorrer" para ejecutar la instrucción decodificada.

En el ejemplo genérico de la figura 53, podemos ver como el OP de la instrucción apunta a una secuencia de microinstrucciones, que están dentro de estado exe (después de la decodificación saltamos al estado de ejecución si no hay petición de interrupción). Dentro de este estado, se van ejecutando acciones y a cada ciclo de reloj, se incrementa la fase de ejecución (exe_phase). Cuando llegamos a la fase de ejecución 2, nos encontramos con una transferencia, por lo que saltamos al estado de transferencia (TRANSFER). En este estado, y también con el puntero OP señalando al microcódigo de la instrucción decodificada y utilizando la fase auxiliar "aux_phase", ejecutamos las microi nstrucciones (acción3 y acción 4) en paralelo con la transferencia, (estamos ejecutando micro opera-ciones mientras la BIU está accediendo a la memoria, este trabajo en paralelo no lo implementan los micros originales y es una de las causas de que estos modelos sean mucho más rápidos). La comunicación con la memoria dura un número de ciclos indeterminado debido a la característica de sincronización del bus con dispositivos más lentos (no se termina una transferencia hasta que la señal READY='1'). Cuando la transferencia ha terminado, (ckX) aparece la señal "gt_bus_eu" y volvemos al estado de ejecución EXE dónde continuamos con la fase de ejecución 3 hasta el final de la secuencia (exe_done) momento en el que pasamos al estado de decodificación DECO..

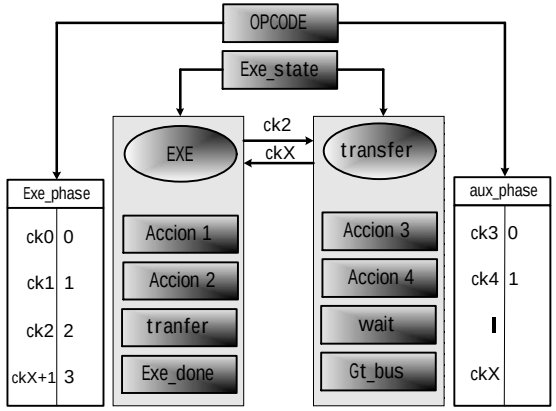


Fig 53.- Ejemplo Genérico de una Secuencia de Microinstrucciones.

PROCESO DEMUX_OP

Este proceso, lo único que hace es "sacar" la información contenida en el código de operación (OP) de la instrucción para poder utilizarlo de una forma más sencilla en la lógica de las microinstrucciones.

La información la obtenemos de las tablas del Juego de Instrucciones de los modelos que podemos encontrar en el apéndice A. Así por ejemplo la instrucción "MOV Inmediato a Registro":

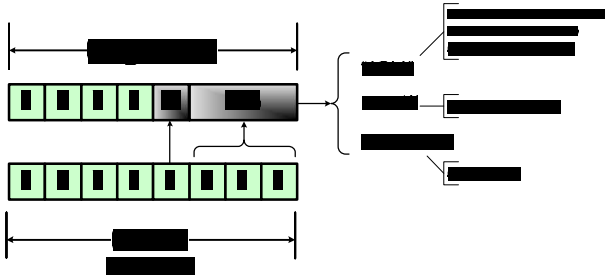


Fig 54.- La Información del OP es Tratada de Forma Genérica gracias a demux_op.

19.2 ENTRADAS Y SALIDAS

SEÑAL	TIPO	FUNCIÓN
ph1, ph2	IN	Reloj interno
res	IN	Es la señal interna que resetea todos los componentes del modelo
flush	OUT	Cuando hay un salto la cgrom activa esta señal.
cgrom<->biu		
exe_done	OUT	Indica que la unidad de ejecución ha terminado secuencia de microinstrucciones del OP en ejecución
w_dfu	OUT	Indica a la unidad dfu (n3/r4) si la transferencia va a ser de byte (si es '0') o palabra (si es'1').
sreg_gau	OUT	Transmite el prefijo utilizado para la generación de la dirección física
modo_gau	OUT	Programa el modificador me la unidad generadora de direcciones.
r_m_gau	OUT	Transmite este campo necesario para que gau que junto con modo indican el modo de direccionamiento.
prefix_asserted	OUT	Indica a la gau que no utilice el segmento implícito porque hay activado un prefijo de segmento en la instrucción.
rq_only_ea	OUT	EU pide a la biu que le pase la dirección efectiva EA.
rq_bus_cycle_eu	OUT	La EU pide a la unidad que va en busca de datos (dfu, n3/r4) que realice el ciclo de bus indicado en esta señal.
rq_bus_eu	OUT	EU solicita el control del bus al árbitro de canal
gt_bus_eu	IN	El árbitro concede el bus a la unidad de ejecución.
data_source	IN	Datos fuente recibidos desde el exterior tras una petición.
data_dfu_out	OUT	Datos que mandamos a la dfu para que los saque cuando corresponda.
disp_gau_out	OUT	Indicamos a la gau el desplazamiento con el tiene que calcular la dirección física (PA).
cgrom<->puertos		
NMI_ack	OUT	Borra el registro de detección de interrupción no mascarable que se encuentra en ports.
NMI_internal	IN	Cuando ports detecta una petición de interrupción válida, activa esta señal.
INTR_internal	IN	Cuando el pin INTR de ports está activo, esta señal también se activa indicando a la eu que existe una petición de interrupción mascarable.
cgrom <-> sreg		
wr_sreg_eu	OUT	Indica que se va a escribir en los registros de segmento, sru (n1/r2)
rd_sreg_eu	OUT	Se va a escribir en sru.
sreg_sel_eu	OUT	Selección del registro con el que se va a operar,
sreg_in_eu	OUT	Dato que vamos a escribir en sru
sreg_out_eu	IN	Dato que obtenemos de sru.
cgrom <-> rgu		
r16_r8n_eu	OUT	Selecciona los registros de 16 u 8 bits de la unidad de registros de uso general, rgu n1/r1.
wr_reg_eu	OUT	Indica que se va a escribir en los registros de uso general, rgu (n1/r1)
rd_reg_eu	OUT	Se va a escribir en rgu.
reg_sel_eu	OUT	Selección del registro con el que se va a operar,
reg_in_eu	OUT	Dato que vamos a escribir en rgu
reg_out	IN	Dato que obtenemos de rgu.
cgrom <-> decode		
op1	IN	Código de Operación 1
op2	IN	Código de operación 2
data	IN	Datos inmediatos en la instrucción
disp	IN	Desplazamiento en el código de la instrucción
rep_prefix	IN	El prefijo REP precede a una instrucción
z	IN	Activado el bit z
seg_prefix	IN	Indica que hay activado un prefijo de segmento en la instrucción
seg_used	IN	Indica el segmento de prefijo
instr_size	IN	Nos dice el número de bytes que tiene la instrucción decodificada
deco_done	IN	Indica que ha terminado con la decodificación
exe_done	OUT	La cgrom nos informa que ha terminado la ejecución de la instrucción
flush_deco	OUT	La cgrom borra la decodificación en curso.

SEÑAL	TIPO	FUNCIÓN
cg_rom <-> exe		
new_ip	IN	Indica que se ha calculado una nueva IP
exe_state	IN	Es el estado de la unidad de ejecución
begin_cont	OUT	Indica a la unidad exe que entre en el estado CONT.
end_cont	OUT	Indica a la unidad exe que vuelva al estado de ejecución
begin_mul	OUT	Indica a la unidad exe que entre en el estado MULTIPLICAR.
end_mul	OUT	Indica a la unidad exe que vuelva al estado de ejecución
begin_div	OUT	Indica a la unidad exe que entre en el estado DIVIDIR
end_div	OUT	Indica a la unidad exe que vuelva al estado de ejecución
cgrom <-> alu		
op_alu	OUT	Indica a la ALU la operación que tiene que realizar
w_alu	OUT	Operción de 16 u 8 bits.
bus_a	OUT	Operando A
bus_b	OUT	Operando B
c_flag_eu	OUT	Flag de carry de entrada a la ALU
s_flag_eu	OUT	Flag de signo de entrada a la ALU
p_flag_r	IN	Flags de Paridad, Signo, Carry, Overflow, Cero y Acarreo Auxiliar generados por la ALU resultado de la operación realizada.
s_flag_r		
c_flag_r		
o_flag_r		
z_flag_r		
a_flag_r		
cgrom <-> pc		
inc_contador	OUT	Incrementa el valor de contador.
clr_contador	OUT	Pone a cero el valor de contador
contador	IN	Salida de contador utilizado por microcódigo en secuencias auxiliares.
inc_aux_phase	OUT	Incrementa el valor del contador de la fase auxiliar
clr_aux_phase	OUT	Borra el contenido del contador de la fase auxiliar
load_aux_phase	OUT	Hace que el valor en la señal new_aux_phase pase a la salida
new_aux_phase	OUT	Contiene el nuevo valor a cargar en el contador de la fase auxiliar
aux_phase	IN	Volor del contador de la fase auxiliar, es utilizado como puntero en la secuencia de microinstrucciones en procesos auxiliares
inc_exe_phase	OUT	Incrementa el valor del contador de la fase de ejecución
clr_exe_phase	OUT	Borra el contenido del contador de la fase de ejecución
load_exe_phase	OUT	Hace que el valor en la señal new_exe_phase pase a la salida
new_exe_phase	OUT	Contiene el nuevo valor a cargar en el contador de la fase de ejecución
exe_phase	IN	Volor del contador de la fase de ejecución, es utilizado como puntero en la secuencia de microinstrucciones-
cgrom <->spw		
wr_flags	OUT	Escribe el registro de banderas
flags_in	OUT	Valor a escribir en el registro de banderas.
flags_out	IN	Salida del registro de Banderas
wr_rqst	OUT	Escribe en el registro de petición de Interrupción
rqst_in	OUT	Dato a escribir en el registro de petición de interrupción.
rqst_out	IN	Salida del registro de petición de interrupción.
wr_inserv	OUT	Escribe en el registro de interrupción en servicio
inserv_in	OUT	Dato a escribir en el registro de interrupción en servicio.
inserv_out	IN	Salida del registro de interrupción en servicio.
wr_imask	OUT	Escribe en el registro de máscara de interrupciones
mask_in	OUT	Dato a escribir en el registro de máscara de interrupciones
imask_out	IN	Salida del registro de máscara
wr_gau	OUT	Escribe en los registros de comunicación con la BIU
data_dfu_in disp_gau	OUT	Datos a escribir en data_dfu y disp_gau respectivamente.
data_dfu_out disp_gau	IN	Salidas de los registros data_dfu.y disp_gau respectivamente

Tab 35.- Entradas y Salidas de Code Generador ROM (fichero "up808x_eu_cgrom.vhd").

Esta tabla resultará, sin duda, de gran ayuda a aquel que quiera seguir una secuencia de microinstrucciones.

19.3 EJEMPLOS

La cgrom, cuyo componente podemos ver en la figura 52, es el "corazón" de los modelos. Como se comentó anteriormente, su lógica combinacional programa cada una de las unidades del modelo para que realice una función. Una secuencia de pasos (microinstrucciones), hace que las unidades del modelo realicen una serie de operaciones consecutivas, que correctamente encadenadas, culminarán con la ejecución de una instrucción específica. Estos ejemplos hacen referencia a una determinada secuencia de microinstrucciones,

MOV INMEDIATO A REGISTRO

Empecemos por una instrucción muy sencilla, La instrucción "Move Inmediate to Register", cuyo formato de OP podemos ver en la figura 54, sólo tiene una fase de ejecución como podemos ver en el código de microinstrucción:

```
    CASE OP1(7 DOWNTO 4) IS
    WHEN "1011" => --MOV Move Immediate to Register
        --RGU write
        r16_r8n_eu <= w;
        reg_sel_EU <= reg;
        WR_reg_EU  <= TRUE;
        reg_in_eu <= data;
        --Ejecución terminada
        exe_done_dummy <= TRUE;
    …
    WHEN OTHERS => NULL;
    END CASE;
```

En la única microinstrucción se programa la unidad de los registros de uso general con los valores obtenidos de la demultiplexación del OP de la instrucción. En el siguiente código podemos ver como se demultiplexa la información del OP:

```
-------------------------DEMULTIPLEXOR DE OP Code

--(cccc,w reg)
CASE OP1(7 DOWNTO 4) IS
    WHEN "1011" => --MOV Move Immediate to Register
        w <= OP1(3);
        reg <= OP1(2 DOWNTO 0);
    …
    WHEN OTHERS => NULL;
END CASE;
```

Podemos utilizar la tabla 35 par seguir las asignaciones realizadas en el código de microinstrucción.

INC/DEC REGISTRO

Una operación un poco más complicada es la de incrementar (INC) y decrementar (DEC) un registro, su código de operación lo podemos ver en el juego de instrucciones del apéndice A.

En esta ocasión hay dos microinstrucciones como se puede observar en el código de un poco más abajo y en la figura 55. En la primera microinstrucción leo en los registros de uso general el registro a incrementar (reg) dato que paso inmediatamente a la unidad lógica aritmética que programada para incrementar (OP1(3)='0') o para decrementar (OP1(3)='1') realiza la operación. También en esta microinstrucción se actualiza el registro de banderas (dentro de la unidad de registros especiales, spw), con el resultado de los flags de la alu.

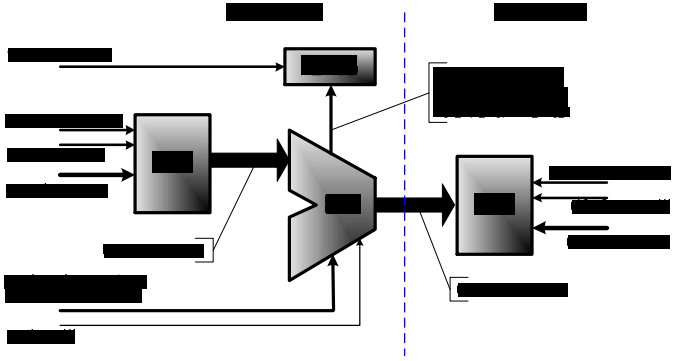


Fig 55.- Representación Gráfica de las Microinstrucciones.

La segunda microinstrucción, lo único que hace es salvar el resultado de la alu en los registros rgu.

```
WHEN "0100" => --INC (OP1(3)='0') Register
               --DEC (OP1(3)='1') Register
    IF exe_phase = 0 THEN
    --Leo los operandos y realizo la operación
        --RGU read r_m
        r16_r8n_eu <= '1'; --Sólo registros de 16 bits
        reg_sel_EU <= reg;
        RD_reg_EU  <= TRUE;
        --Programo Alu
        bus_a <= reg_out;
        --Operación de la Alu
        w_alu <= '1'; --Operandos de 16 bits
        CASE OP1(3) IS--OP_Alu                Flags: O D I T S Z A P C
            WHEN '0' => op_alu <= Incrementar; --INC  *       * * * *
            WHEN '1' => op_alu <= Decrementar; --DEC  *       * * * *
            WHEN OTHERS => NULL;
        END CASE;
        --Flags. Actualizo los flags
        wr_flags <= TRUE;
        flags_in(p_flag) <= p_flag_r; --P_Flag
        flags_in(a_flag) <= a_flag_r; --A_Flag
        flags_in(z_flag) <= Z_Flag_r; --Z_Flag
        flags_in(s_flag) <= S_Flag_r; --S_Flag
        flags_in(o_flag) <= O_Flag_r; --O_Flag
    ELSIF exe_phase = 1 THEN
    --Salvo el resultado en el destino
            --RGU write r_m
            r16_r8n_eu <= '1'; --Sólo registros de 16 bits
            reg_sel_EU <= reg;
            WR_reg_EU  <= TRUE;
            reg_in_EU <= result;
            --Ejecución terminada
            exe_done_dummy <= TRUE;
    END IF;
WHEN OTHERS => NULL;
END CASE;
```

20. DECODIFICADOR, DECO (n2/r5)

n2/r5 Unidad de Decodificación, (deco): Encargada de interpretar el código de operación OP y proporcionarle la información a la CGROM. La decodificación lee los datos directamente de la cola de instrucciones (n1/r3), decodifica el formato de la instrucción en cuestión y accede a los bytes siguientes si éstos existen.

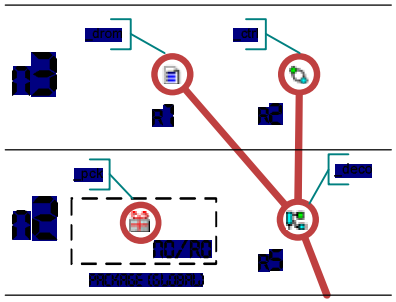


Fig 56.-Detalle de la Jerarquía de la Unidad de Ejecución.

La unidad de decodificación está compuesta por dos unidades, como se puede observar en las figura 56 y 57. Las unidades son:

- n3/r1 ROM del Decodificador, Decode ROM (DROM): Esta unidad contiene la lógica combinacional que utiliza el decodificador para interpretar el OP. En el apéndice A se puede observar el formato de los OP Code.
- n3/r2 Control del Decodificador: (deco_ctrl): Es el corazón del decodificador, contiene el grafo de estados con las transiciones necesarias para decodificar toda la información.

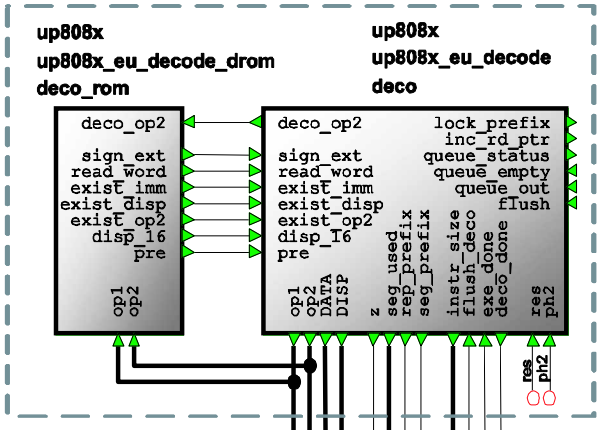


Fig 57.-Esquema de la Unidad de Decodificación

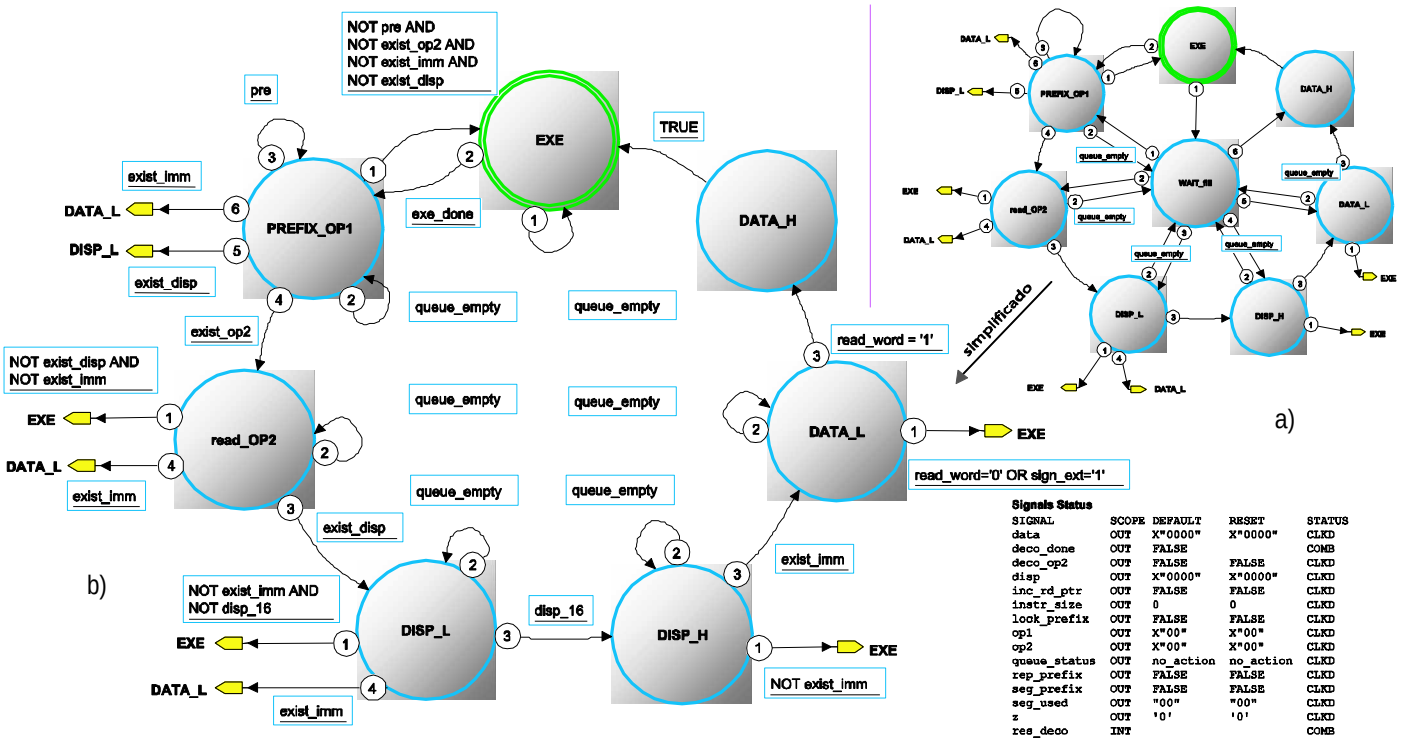


Fig 58.-Grafo de Estados de la Unidad de Ejecución. a) Sin Simplificar. b) Simplificado.

EXPLICACIÓN DEL GRAFO DE ESTADOS

El la figura 58 podemos ver el grafo de estados simplificado de la unidad de decodificación y el grafo sin simplificar. En el grafo sin simplificar, se puede apreciar mejor cómo la unidad de decodificación entra en un estado de espera (WAIT) siempre que la cola de instrucciones se queda vacía (queue_empty) y permanece es este estado hasta que la unidad ifu (n3/r5) accede a una instrucción en memoria y la coloca en la cola de instrucciones.

En el grafo simplificado se ve claramente la secuencia de decodificación de una instrucción. Dependiendo del formato de instrucción cuya información relativa a la decodificación se encuentra en la ROM del decodificador (d_rom, n3/r1) confeccionada a base de la información que encontramos en la tabla de "formato OP del juego de instrucciones" en el apéndice A y del valor que adoptan los campos (de existir) mod, r/m, w y s, la secuencia de estados que seguiremos para decodificar la instrucción, será distinta. En la Tabla 37 podemos ver los caminos que puede adoptar la decodificación de una instrucción suponiendo que no existen prefijos.

Cuando al decodificar un instrucción, el decodificador se encuentra con un prefijo de instrucción, se activa la salida correspondiente y se salva la información de segmento (si el prefijo activado ha sido un prefijo de segmento) y permanecemos en el estado PREFIX_OP1 hasta que la instrucción decodificada no sea un prefijo. Una instrucción puede tener hasta tres prefijos. Como se explica en la siguiente sección.

PREFIJOS

Pueden aparecer hasta tres prefijos delante de una instrucción, aunque no tienen sentido en todas las instrucciones. Por ejemplo, un prefijo de repetición REP, no tiene sentido delante de una instrucción CALL. El decodificador permanece en PREFIX_OP1 hasta encontrar un código de instrucción que no sea un prefijo. Existen tres tipos de prefijos:

PREFIJO DE SEGMENTO, CS:/ES:/DS:/SS:

Estos prefijos, cuyo formato podemos ver en la figura 59, indican a la cgrom que programe la biu para que no calcule la dirección física de memoria (PA) con el segmento por defecto, sino que utilice el registro de segmento indicado

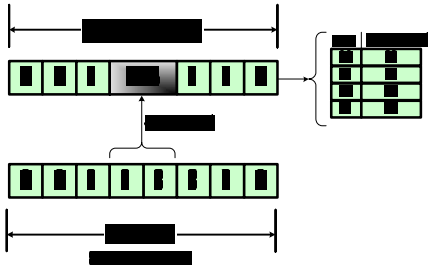


Fig 59.-Formato del Prefijo de Segmento

Deco activa la señal "seg_prefix" indicando que existe un prefijo de segmento, y asigna a la salida "seg_used" el segmento indicado en el OP real.

PREFIJO DE REPETICIÓN, REP:

El prefijo de repetición sólo tiene sentido delante de las instrucciones de manipulación de cadenas de datos. Su formato lo podemos ver en la siguiente figura:

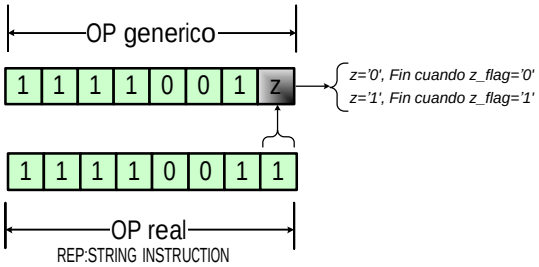


Fig 60.-Formato del Prefijo de Repetición REP

Cuando el decodificador encuentra un prefijo de repetición, activa la señal rep_prefix y saca por z el valor a comparar con el flag "z_flag" de SPW para salir del bucle de repetición.

PREFIJO DE BLOQUEO, LOCK:

El prefijo de bloqueo puede aparecer delante de cualquier instrucción e indica al bus exterior que ningún maestro de canal puede utilizar el bus mientras se ejecuta la instrucción que sigue a LOCK. Es muy importante utilizarlo con instrucciones de cadenas para impedir que otro maestro acceda a las posiciones de memoria con las que la instrucción de cadenas está operando..

Es un OP "puro" y la unidad decode activa la señal "lock_prefix" cuando encuentra ese OP "puro".

20.2 ENTRADAS Y SALIDAS

Como de pude ver en las figuras 56 y 57, la unidad de decodificación está formada internamente por dos componentes, cuyas señales de entrada y salida podemos ver en la siguiente tabla:

- La unidad de decodificación deco_rom, cuyo fichero ("up808x_eu_decode_drom.vhd") está definido en VHDL.
- La unidad de control del decodificador "deco" contenida en el fichero "up808x_eu_decode.vhd" que contiene la definición del grafo que podemos ver en la figura 58 b.

Ambos componentes, son comunes para todos los mosdelos y por lo tanto, están definidos en la librería común "up808x".

SEÑAL	TIPO	FUNCIÓN
ph2	IN	Reloj interno.
res	IN	Señal Interna de reset.
deco_op2	Int	Indica a la DROM que decodificque el OP2
deco_rom		
sign_ext	int	Indica que hay una extensión de signo.
read_word	Int	Hay que leer una 16 bits de datos inmediatos
exist_imm	Int	Activada cuando existen datos inmediatos en la instrucción
exist_disp	Int	Activada cuando existe desplazamiento
exist_op2	Int	Activada cuando existe un op2
disp_16	Int	Indica que el desplazamiento es de 16 bits
pre	int	Indica que el OP es un prefijo.
deco		
OP1	OUT	Código de Operación 1.
Op2	OUT	Código de Operación 2
DATA	OUT	Datos inmediatos
DISP	OUT	Desplazamiento
rep_prefix	OUT	Existe un prefijo de repetición
seg_prefix	OUT	Existe un prefijo de segmento
lock_prefix	OUT	Existe un prefijo de bloqueo
Z	OUT	Valor a comparar con z_flag (obtenido de REP)
seg_used	OUT	Segmento a utilizar para generar la dirección (obtenido del prefijo de segmento)
listr_size	OU	Longitud (en bytes) de la instrucción decodificada
deco_done	OUT	La decodificación ha terminado
flush_deco	IN	Cgrom indica que "aborte" la decodificación.
exe_done	IN	Indica que la ejecución de la instrucción ha terminado.
lic_rd_ptr	OUT	Indica a la cola de instrucciones que incremente el puntero de lectura
queue_status	OUT	Última operación realizada con la cola de instrucciones
queue_empty	IN	La cola de instrucciones está vacía
queue_out	IN	Byte obtenido de la cola de instrucciones.
flush	IN	Aborta la decodificación (hay un salto en el programa).

Tab 37.-Entradas y Salidas de la Unidad de Decodificación

20.3 ESTADOS

Esta unidad dispone de 7 estados como podemos apreciar en la figura 58:

- EXE, Estado Inicial (e0). En este estado, está trabajando la unidad de ejecución, el decodificador está a la espera de que la cgrom active la señal "exe_done" indicando que ha terminado la ejecución de la instrucción actual. En este momento, si la cola de instrucciones no está vacía (transición 1, t1), saltamos (transición 2, t2) al primier estado de la decodificación, PREFIX_OP1

- PREFIX_OP1, (e1). En este estado, la parte de control del decodificador, pregunta a la drom (n3/r1) por el tipo de instrucción. Dependiendo de la información en la drom saltamos al estado EXE, read_OP2, DISP_L o DATA_L (transiciones t1, t4, t5 o t6 respectivamente). Si la cola no tiene datos, no se salta a ningún estado que no sea e0. Si resulta que la instrucción que estamos decodificando es un prefijo de instrucción, pedimos otro dato a la cola de instrucciones y activamos las salidas pertinentes. Como se indicó en la sección anterior. En este estado, se memoriza el código del OP1, siempre que no sea un prefijo.

- Read_OP2, (e2): En este estado, salvamos el valor desde la cola de instrucciones en el registro OP2 y preguntamos a la drom por la información que este contiene. La drom detectará los campos reg, r_m, mod, s y w (si existen) y nos indicará si existen datos inmediatos (exist_imm) o si al código de operación le sigue un desplazamiento (exist_disp), en caso afirmativo, saltamos a DATA_L (t4) o a DISP_L (t3) respectivamente. En caso de no existir ninguno (t1), la instrucción está completamente decodificada y saltamos a la fase de ejecución. Salvo esta última transición, en la que no nos importa si la cola de instrucciones está vacía o no, no pasamos al estado siguiente si no hay al menos un byte en la cola de instrucciones (permanecemos en el mismo estado si la cola de instrucciones está vacía, t2).

- DISP_L, (e3). El dato obtenido de la cola de instrucciones (8 bits) lo salvamos en el registro bajo de la señal de desplazamiento DISP (de 16 bits) extendiendo o no el signo de la parte baja a la paret alta de DISP en función del campo "mod". La información de la drom, nos indicará si existe un desplazamiento de 16 bits, en cuyo caso saltaremos a DISP_H (transición 3, t3), o si existen datos inmediatos después del desplazamiento (t4). Si el desplazamiento es de 8 bits y no existen datos inmediatos (t1), saltamos a la fase de ejecución (e0). Salvo en este caso y, como en el resto de los estados, no saltaremos al siguiente estado si no hay bytes en la cola de instrucciones.

- DISP_H, (e4). Al saltar a este estado, el byte recibido desde la cola de instrucciones, es salvado en la parte alta del registro de desplazamiento DISP (de 16 bits). Si la drom son dice que existen datos inmediatos que siguen al desplazamiento y la cola de instrucciones tiene datos, saltamos al estado DATA_L (t3), de lo contrario, la decodificación ha terminado y saltamos a la fase de ejecución (e0)

- DATL_L, (e5). El dato (8 bits) de la cola de instrucciones es salvado en la parte baja del registro de datos DATA (de 16 bits). Algunas instrucciones aritméticas con datos inmediatos, codifican datos de 16 bits en un byte (siempre que se pueda), en estas situaciones, en el código de operación, se activa la señal "s" que indica que hay que hacer una extensión de signo. En este caso, la acción e5/a rellena el byte alto del regitro DATA con el signo del byte bajo. En el caso de que s='1' o el dato inmediato sea de un solo byte (t1) saltamos a la fase de ejecución, de lo contrario pasamos al estado DATA_H si la cola de instrucciones no está vacía.

- DATA_H, (e6). El dato que recibimos desde la cola de instrucciones es directamente salvado en la parte alta del registro de datos inmediatos DATA. En este estado, sólo hay una transición posible, hacia la fase de ejecución (e0).

Como se puede observar en el grafo sin simplificar (figura 52 a), cuando la cola está vacía, no permanecemos en el estado actual, sino que saltamos a un estado (WAIT_fill) en espera de que la cola de instrucciones reciba algún valor desde el exterior. Gráficamente resulta más real, pero al saltar a este estado, tenemos que memorizar el estado anterior y las condiciones anteriores, lo que implica más registros.

El grafo simplificado pese a ser menos "gráfico" ocupa bastantes registros menos.

20.4 CAMINOS

Este grafo de estados es, junto con el de ejecución (n2/r2) y el de la unidad DFU (n3/r4) uno de los más complicados del modelo. Sin duda es debido al gran número de transiciones y estados junto con la complejidad de su lógica.

En la siguiente tabla (tabla 38), podemos ver una relación de los posibles caminos del grafo si no tenemos en cuenta los prefijos de instrucción que, recordemos, pueden ser 1, 2 o 3 bytes más. Las instrucciones pueden tener hasta tres prefijos.

CAMINO	TIPO DE INSTRUCCIÓN DECODIFICADA
exe-Prefix_op1-exe	Instrucción de 1 Byte
exe-prefix_op1-data_l-exe	Instrucción de 1 Byte con dato Inmediato de 8 bits
exe-prefix_op1-data_l-dat_h-exe	Instrucción de 1 Byte con dato Inmediato de 16 bits
exe-prefix_op1-disp_l-exe	Instrucción de 1 Byte con desplazamiento de 8 bits
exe-prefix_op1-disp_l-disp_h-exe	Instrucción de 1 Byte con desplazamiento de 16 bits
exe-prefix_op1-disp_l-data_l-exe	Instrucción de 1 Byte con dato Inmediato y desplazamiento de 8 bits
exe-prefix_op1-disp_l-data_l-dat_h-exe	Instrucción de 1 Byte con dato Inmediato de 16 bits y desplazamiento de 8 bits.
exe-prefix_op1-disp_l-disp_h-data_l-exe	Instrucción de 1 Byte con dato Inmediato de 8 bits y desplazamiento de 16 bits
exe-prefix_op1-disp_l-disp_h-data_l-dat_h-exe	Instrucción de 1 Byte con dato Inmediato de y desplazamiento de 16 bits.
exe-Prefix_op1-op2-exe	Instrucción de 2 Bytes
exe-prefix_op1-op2-data_l-exe	Instrucción de 2 Bytes con dato Inmediato de 8 bits
exe-prefix_op1-op2-data_l-dat_h-exe	Instrucción de 2 Bytes con dato Inmediato de 16 bits
exe-prefix_op1-op2-disp_l-exe	Instrucción de 2 Bytes con desplazamiento de 8 bits
exe-prefix_op1-op2-disp_l-disp_h-exe	Instrucción de 2 Bytes con desplazamiento de 16 bits
exe-prefix_op1-op2-disp_l-data_l-exe	Instrucción de 2 Bytes con dato Inmediato y desplazamiento de 8 bits
exe-prefix_op1-op2-disp_l-data_l-dat_h-exe	Instrucción de 2 Bytes con dato Inmediato de 16 bits y desplazamiento de 8 bits.
exe-prefix_op1-op2-disp_l-disp_h-data_l-exe	Instrucción de 2 Bytes con dato Inmediato de 8 bits y desplazamiento de 16 bits
exe-prefix_op1-op2-disp_l-disp_h-data_l-dat_h-exe	Instrucción de 2 Bytes con dato Inmediato de y desplazamiento de 16 bits.

Fig 38.- Posibles Caminos del Grado, si NO Tenemos en Cuenta que Pueden Aparecer Prefijos.

BIU

Unidad de Interface con el Bus

21. UNIDAD DE INTERFACE CON EL BUS, BIU (n1/r7)

Esta es la última unidad del nivel 1 que nos queda por explicar, por su importancia, como en el caso de la UE, le dedicamos un capítulo a parte en el que iremos adentrándonos en la jerarquía de la unidad y estudiando en detalle cada una de las unidades que la componen.

En el detalle de la jerarquía de esta unidad (ver la figura 61), podemos observar el alto nivel estructuración de su código. A medida que vayamos explicando las distintas unidades, iremos "podando" ramas.

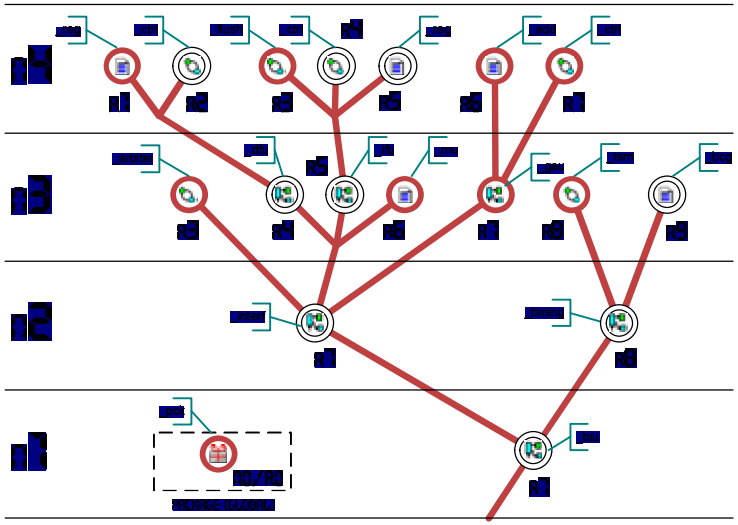


Fig 61.-Detalle de la Jerarquía de la BIU.

Podemos observar en la figura 61 que existen componentes comunes a ambos modelos (librería up808x) y otros que no lo son, y por lo tanto pertenecen a la librería del 8086 (up8086) o a la librería del 8088 (up8088). Un componente o unidad no común, propaga esta característica a la rama madre, esta propagación llega hasta el tronco. Así pues, una unidad no común en una "yema" del árbol, hace que todas las ramas antecesoras sean "no comunes". Esto tiene su lógica esquemática si tenemos en cuenta que si aparece una señal es esta unidad que tiene que llegar hasta el exterior (hata la "raíz"), esta tiene que aparecer en todos los esquemas de la jerarquía.

Los archivos del nivel n1/r7, definen un esquema. En la siguiente figura, podemos ver el esquema que definen los archivos "up8088_biu.vdh" (para el modelo 8088) y el "up8086_biu.vhd" (para el modelo 8086). Las pocas diferencias entre ambos están resaltadas en la figura 62 b).

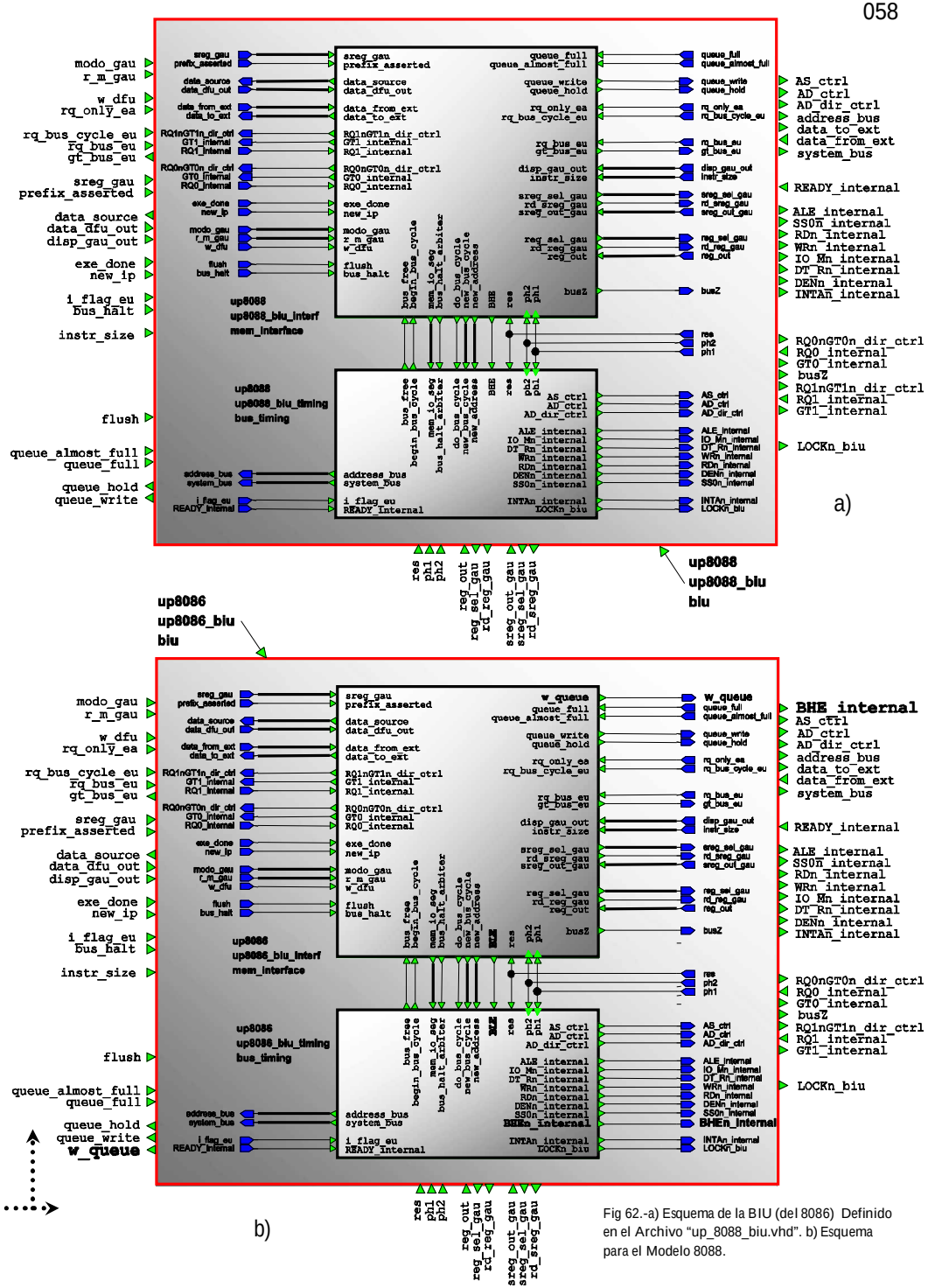


Fig 62.-a) Esquema de la BIU (del 8086) Definido en el Archivo "up_8088_biu.vhd". b) Esquema para el Modelo 8088.

Estas diferencias son (8086 respecto al 8088):

- Aparece la señal “w_queue” para indicar a la cola de instrucciones (n1/r5) que escriba 2 bytes en sus registros.
- La nueva señal BHE_internal, activa el banco del byte alto del sistema de memoria.
- La señal interna BLE, indica a la unidad bcg (n3/r9) cuando tiene que activar BHE.

n1/r7, Unidad de Interface con el Bus (Bus Interface Unit, BIU): La BIU se encarga de establecer una comunicación (interface) bidireccional entre el bus loca (bus externo) y la unidad de ejecución interna. Internamente la BIU está dividida en dos bloques funcionales (ver figura 62),

- la unidad de sincronismo del bus (bus_timing n2/r8), que trata aspectos relacionados con la temporización del bus: generación del estado de bus, generación de las señales de los distintos tipos de ciclos de bus, generación de laseñales del bus de control....
- y (2) la unidad de comunicación con la memoria (mem_interface n2/r7) dedicada a aspectos de comunicació con el bus como el arbitraje interno del canal, la atención de peticiones del bus local, la generación de la dirección física...

Las unidades de la biu tienen muchos ficheros que definen esquemas (como se puede apreciar en la figura 61) y todos ellos se repiten (con algunos cambios) en las librerías de los modelos 8088/8086. Los cambios entre estos esquemas, son mínimos (alguna señal) como en los archivos de este capítulo, que sólo se diferencian en tres señales. Las diferencias más sustanciales se encuentran en la lógica de algunas de las unidades, sobretodo de las que acceden la bus externo (la unidad que va en busca de instrucciones, ifu y la unidad que va en busca de datos, dfu). Más adelante se detalla el comportamiento de estas unidades.

21.1 ENTRADAS Y SALIDAS

SEÑAL	TIPO	FUNCIÓN
ph1, ph2	IN	Reloj interno
res	IN	Es la señal interna que resetea todos los componentes del modelo
biu <-> ports		
AS_ctrl	OUT	Control del multiplexor del puerto AS
AD_ctrl	OUT	Control del Multiplexor del puerto AD
AD_dir_ctrl	OUT	Controsa si AD es de salida o de entrada.
address_bus	IN	Dirección que va a salir por los puertos A, A?
data_to_ext	OUT	Datos que saldrán por el puero AD.
dato_from_ext	IN	Datos recogidos en el puerto AD
system_bus	OUT	Bus de sistema, saldrá por AS, y (S0n-S2n/DT_Rn-IO_Mn-SS0)
BHEn_internal	OUT	Señal de activación del bloque alto de la memoria.
READY_internal	IN	Señal de READY sincronizada.
ALE_internal	OUT	Señales internas de sus respectivos puertos creadas por el Generador de Ciclos de Bus, bcg (n3/r9)
SS0n_internal		
RDn_internal		
WRn_internal		
IO_Mn_internal		
DT_Rn_internal		
DENn_internal		
INTAn_internal		
BHEn_internal	OUT	Habilita la parte alta del banco de memoria
RQ0nGT0n_dir_ctrl	OUT	Control de dirección del puerto RQ0nGT0n
RQ0_internal	IN	Señales RQ0n y GT0n internas del puerto RQ0nGT0n. Generada por la unidad de control Multibus, mu (n3/r6)
GT0_internal	OUT	
RQ1nGT1n_dir_ctrl	OUT	Control de dirección del puerto RQ1nGT1n
RQ1_internal	IN	Señales RQ1n y GT1n internas del puerto RQ1nGT1n. Generadas por la unidad de control Multibus, mu (n3/r6)
GT1_internal	OUT	
busZ	OUT	Pone todos los puertos con bufer triestado en alta impedancia.
LOCKn_biu	OUT	Señal interna del puero LOCKn generada por la biu (n1/r7)
biu <-> queue		
queue_almost_full	IN	Indica que la cola está casi llena
queue_full	IN	La cola de instrucciones está llena
queue_hold	OUT	Congela las operaciones con la cola de instrucciones.
queue_write	OUT	Escribe en la cola de instrucciones el dato en data_from_ext
w_queue	OUT	Indica que escriba una palabra de datos en los registros de la cola.
biu <-> reg/sru		
reg_out	IN	Dato recibido desde rgu.
reg_sel_gau	OUT	Selección del registro de rgu a leer
rd_reg_gau	OUT	La gau indica que va a leer un registro de rgu
sreg_out_gau	IN	Dato recibido desde sru
sreg_sel_gau	OUT	Selección del registro que se va a leer de sru.
rd_sreg_gau	OUT	Indica que se va a leer un registro de sru.
biu <-> cgrom		
flush	IN	Hay un salto en el programa y tenemos que resincronizar.
exe_done	IN	Indica que la unidad de ejecución ha terminado secuencia de microinstrucciones del OP en ejecución
w_dfu	IN	Indica a la unidad dfu (n3/r4) si la transferencia va a ser de byte (si es ‘0’) o palabra (si es’1’).
sreg_gau	IN	Transmite el prefijo utilizado para la generación de la dirección física
modo_gau	IN	Programa el modificador me la unidad generadora de direcciones.
R_m_gau	IN	Transmite este campo necesario para que gau que junto con modo indican el modo de direccionamiento.
prefix_asserted	IN	Indica a la gau que no utilice el segmento implícito porque hay activado un prefijo de segmento en la instrucción.
rq_only_ea	IN	EU pide a la biu que le pase la dirección efectiva EA.
rq_bus_cycle_eu	IN	La EU pide a la unidad que va en busca de datos (dfu, n3/r4) que realice el ciclo de bus indicado en esta señal.
rq_bus_eu	IN	EU solicita el control del bus al árbitro de canal
gt_bus_eu	OUT	El árbitro concede el bus a la unidad de ejecución.
data_source	OUT	Datos fuente recibidos desde el exterior tras una petición.
data_dfu_out	IN	Datos que mandamos a la dfu para que los saque cuando corresponda.
disp_gau_out	IN	Indicamos a la gau el desplazamiento con el tiene que calcular la dirección física (PA).

Fig 39.- Entradas y Salidas de la BIU, Ficheros "up8088_biu.vhd" y "up8086_biu.vhd".

22. TEMPORIZACIÓN DEL BUS EXTERNO, TIMING (n2/r8)

n2/r8 Unidad de Sincronización con el Bus (bus_timing): Genera las señales de los ciclos de bus y el estado de éste. Los ciclos de bus son utilizados como mecanismo de comunicación con dispositivos externos (Memoria, I/O...) existiendo un ciclo de bus específico para cada operación que el micro realice con el bus. Las señales del Bus de Control son, en su mayoría, generadas por esta unidad.

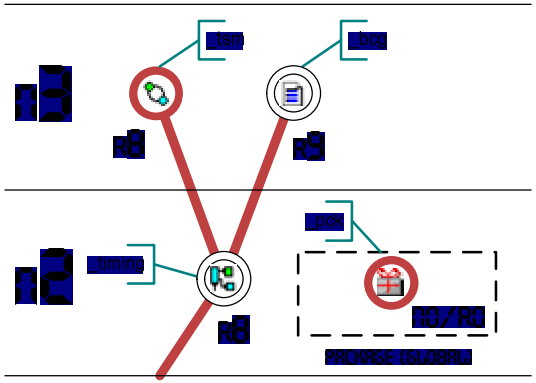


Fig 63.- Detalle de la Jerarquía de la Unidad de Temporización.

Como se observa en las figuras 63 y 64, está compuesta por dos unidades, la unidad tsm (n3/r8) común a ambos modelos y que como se explicará más adelante, genera el estado de la BIU, y la unidad generadora de ciclos de bus, bcg (n3/r9). Esta última unidad, explicada en detalle un poco más adelante, se encarga de generar todas las señales de control necesarias para realizar un ciclo de bus.

Se puede observar en el nivel n2/r8 que ese nudo es un esquema y que es distinto en ambos modelos, por lo que existe un fichero en la librería del modelo 8088 ("up8088_biu_timing.vhd") y otro en la librería del modelo 8086 ("up8086_biu_timing.vhd") que definen este esquema.

En los esquemas de la figura 64, se pueden ver como se interconectan las dos unidades que forman la unidad "timing", y las diferencias que existen entre los modelos 8088 y 8086. Las diferencias del modelo 8086 respecto al 8088, están marcadas en el esquema de la figura 64 a), y son las siguientes:

- La señal BLE, además de indicar a la unidad generadora de ciclos de bus, cuando debe activar la señal BHEn_internal, le indica cuando tiene que realizar el primer ciclo de bus de un reconocimiento de interrupción.
- La señal BHEn_internal, activa el banco del byte alto en el sistema de memoria.

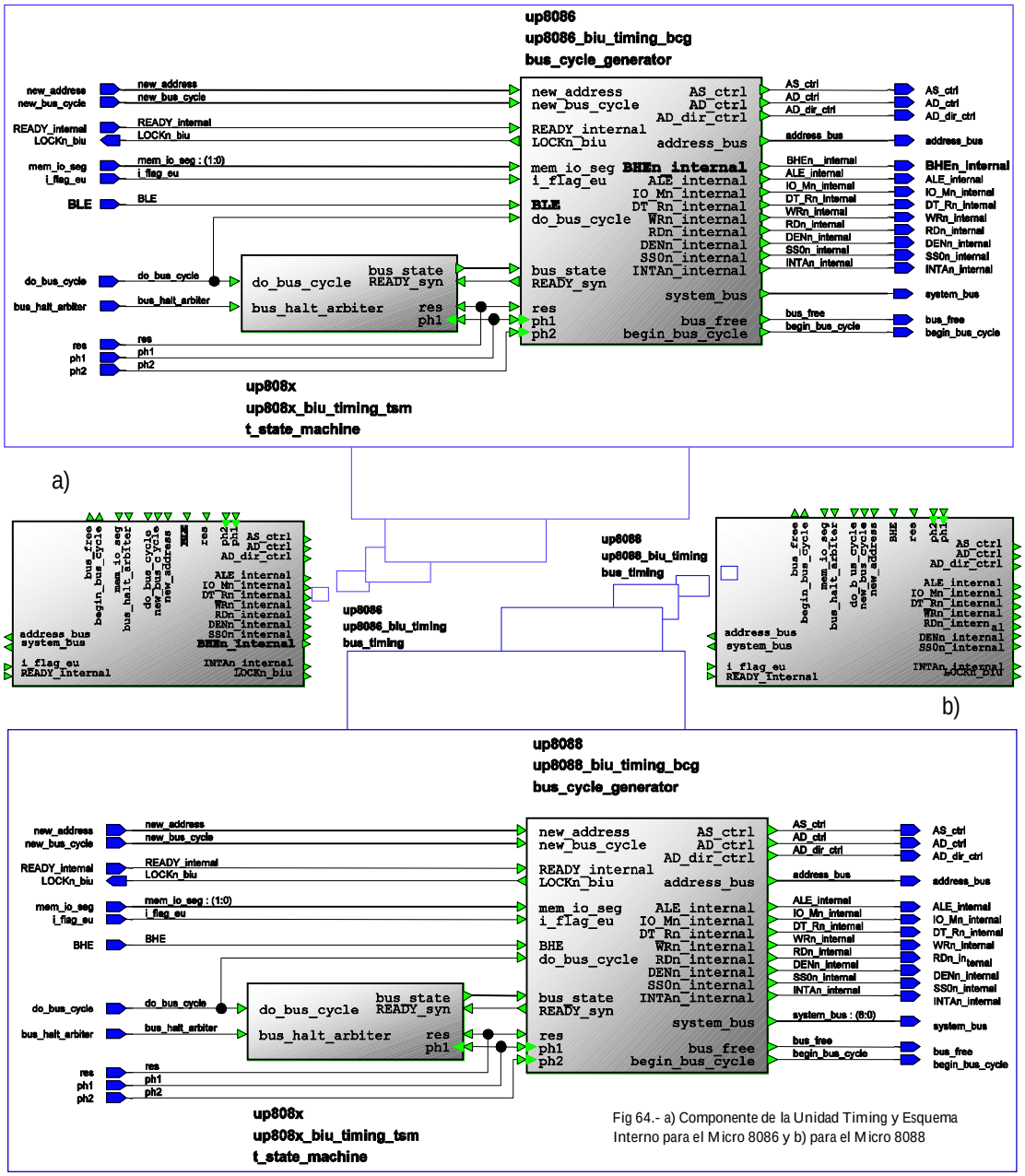


Fig 64.- a) Componente de la Unidad Timing y Esquema Interno para el Micro 8086 y b) para el Micro 8088

22.1 ENTRADAS Y SALIDAS

Aunque la información de estas tablas parezca redundante por haber sido indicada la función de las señales en alguna otra parte de la memoria y, por esta razón, de la sensación de que sólo están para llenar páginas, no es cierto (al contrario de lo que ocurre con este párrafo).

Estas tablas son de gran ayuda para seguir el código de las unidades de los modelos. Estos códigos están ampliamente comentados y para entender el funcionamiento de los modelos, es necesario analizarlo a la vez que leemos la memoria. No se ha incluido en las páginas de la memoria los códigos íntegros de las unidades por el gran volumen que esta adquiriría (eso si que sería llenar páginas).

A continuación, podemos ver las entradas y salidas de la unidad de sincronización del bus externo.

Una información mucho más detallada sobre las señales internas del bus de control externo, la podemos encontrar en las tablas 9, 10 y 11, donde podemos encontrar el significado de los puertos externos e incluso la lógica que siguen.

Las señales con el sufijo "_internal", son las señales internas de los respectivos puertos de comunicación con el exterior. Las señales con sufijo "_ctrl" son señales de control de los puertos de salida (como se comenta en la explicación de la unidad de puertos, n1/r6.

En el estudio de las siguientes unidades (las unidades de generación del estado de bus, tsm n3/r8 y la de generación de ciclos de bus, bcg n3/r9), veremos en detalle todas estas señales y analizaremos el funcionamiento del bus externo.

SEÑAL	TIPO	FUNCIÓN
ph1, ph2	IN	Reloj interno
res	IN	Es la señal interna que resetea todos los componentes del modelo
timing <-> ports		
AS_ctrl	IN	Control del multiplexor del puerto AS
AD_ctrl	IN	Control del Multiplexor del puerto AD
AD_dir_ctrl	IN	Controsa si AD es de salida o de entrada.
address_bus	OUT	Dirección que va a salir por los puertos A, A?
system_bus	OUT	Bus de sistema, saldrá por AS, y (S0n-S2n/DT_Rn-IO_Mn-SS0)
READY_internal	IN	Señal de READY sincronizada.
ALE_internal, SS0n_internal, RDn_internal, WRn_internal, IO_Mn_internal, DT_Rn_internal, DENn_internal, INTAn_internal	OUT	Señales del bus de control generadas por la unidad de ciclos de bus
BHEn_internal	OUT	Habilita la parte alta del banco de memoria
LOCKn_biu	OUT	Señal interna del puero LOCKn generada por la biu (n1/r7)
timing <-> interface		
bus_free	OUT	Indica a la unidad de interface que el bus está libre
begin_bus_cycle	OUT	Indica que ha comenzado un ciclo de bus
mem_io_seg	IN	Se trata de un segmento de memoria o la memoria de entrada salida.
bus_halt_arbiter	IN	Indica que pare el bus en cuanto pueda.
do_bus_cycle	IN	Indica que lance un ciclo de bus.
new_bus_cycle	IN	Le indica el ciclo de bus que tiene que lanzar.
new_address	IN	Le pasa la dirección que tiene que poner en el bus de direcciones.
BHE	IN	Transferencia del byte alto.
BLE	IN	Transferencia del byte bajo.
timing <- cgrom		
i_flag_eu	IN	Dato a sacar por el bus de sistema.

Tab 40.- Tabla de Entradas y Salidas de la Unidad de Temporización del Bus Externo.

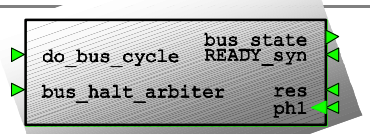
23. MÁQUINA DE ESTADOS DEL BUS, TSM (n3/r8)

n3/r8 Estado del Bus Local, T State Machine (TSM): Es el grafo general del bus que controla en todo momento el estado del bus local.

Podemos ver en la figura 63 que la máquina de estados del bus, es común para todos los modelos y por lo tanto está en la librería up808x (librería común) concretamente en el archivo "up808x_biu_timing_tsm.vhd", en la siguiente tabla, podemos ver el componente y la función de sus entradas y salidas.

23.1 EXPLICACIÓN

Esta unidad se encarga de generar los estados T atendiendo a la lógica impuesta p[illegible] grafo de estados que podemos ver en la figu[illegible]



up808x
up808x_biu_timing_tsm
t_state_machine

SEÑAL	TIPO	FUNCIÓN
ph1	*IN*	*Reloj interno*
res	*IN*	*Es la señal interna que resetea todos los componentes del modelo*
do_bus_cycle	*IN*	*Indica que tiene que "lanzar" otro ciclo de bus porque hay una petición pendiente*
bus_halt_arbiter	*IN*	*El árbitro de bus le indica que entre en estado HALT.*
bus_state	*OUT*	*Saca el estado "T" actual del bus.*
READY_syn	*IN*	*Es la señal de READY interna y sincronizada.*

Tab 41.- Entradas y Salidas de la unidad TSM (n3/r8).

La BIU ejecuta un ciclo de bus para transferir un dato entre las unidades internas y la memoria o dispositivos de estrada salida (en la figura 65 a, se puede observar dos ciclos de bus consecutivos). Un ciclo de bus contiene por lo menos 4 ciclos de reloj o "estados T". Un estado T está delimitado entre el flanco de bajada de reloj externo CK, y el siguiente flanco de bajada de CK. (ver la figura 65 c). La fase 1 (ph1), como se indicó en la explicación de la unidad de inicialización (init, n1/r3), representa el

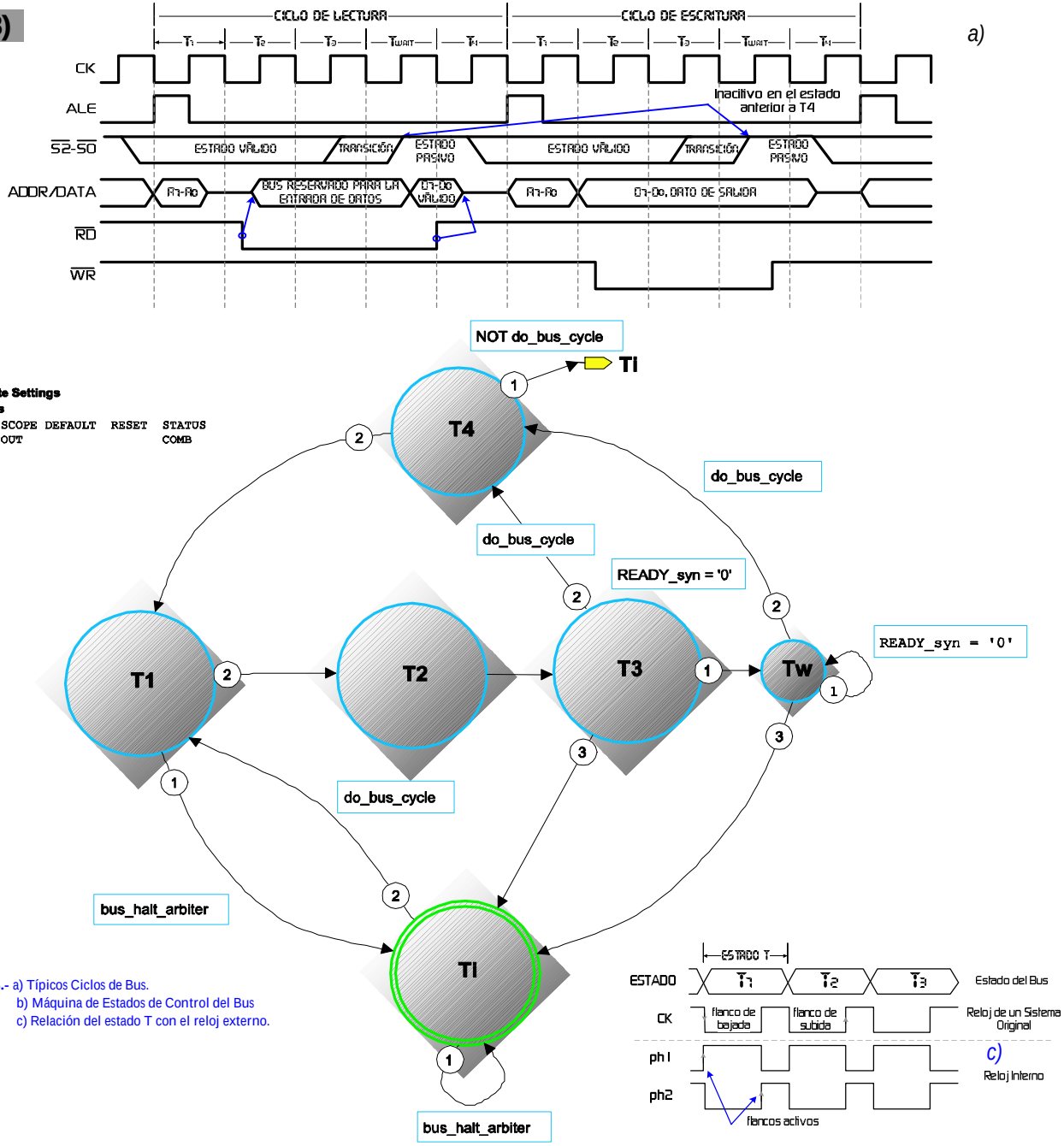


Fig 65.- a) Típicos Ciclos de Bus.
b) Máquina de Estados de Control del Bus
c) Relación del estado T con el reloj externo.

tiempo que CK está a nivel bajo, mientras que la fase 2 (ph2) representa el tiempo que CK está a nivel alto. Las señales de direcciones, datos y de control son generadas por la unidad timing (n2/r8) en los flancos de subida de las diferentes fases dentro de los estados "T". Concretamente, se encarga de ello la unidad generadora de ciclos de bus (bcg, n3/r9) que estudiaremos a continuación.

En la Figura 65 a, podemos ver dos ciclos de bus consecutivos, el primero de lectura y el segundo de escritura. Se puede observar que existe un estado de espera. El estado de espera es una extensión del estado T3 que permite al microprocesador sincronizarse con dispositivos más lentos a través de la señal READY.

Un ciclo de bus puede dividirse en dos fases:

- Fase de Direcciones y Estado. Empieza un poco antes T1 y dura todo T1.
- Fase de Datos. Empieza en T2 y dura hasta T4. En la siguiente figura podemos ver estas fases.

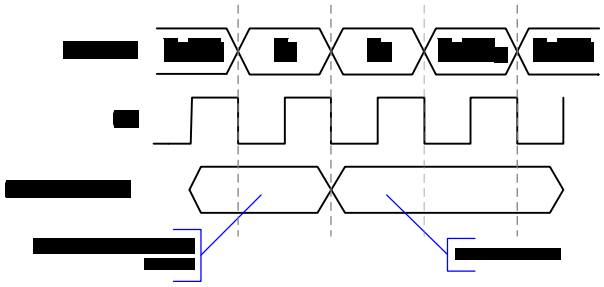


Fig 66.- Fases de un Ciclo de Bus.

Fase de Direcciones/Estado

En la figura 67 a, podemos ver la fase de estado y direcciones. Un ciclo de bus, comienza con una transición de la señal ALE o S2n-S0n. Estas señales aparecen durante la fase dos (ph2) del estado T anterior a T1 (Ti o T4 dependiendo de la operación anterior del bus).

La señal ALE, es utilizada para almacenar la dirección física en los latch externos, esta dirección está presente durante T1 en el bus multiplexado de direcciones y datos (bus A/D). El flanco de bajada de la señal ALE aparece en la mitad de T1 por lo que efectúa el enclavamiento de la dirección cuando está en el bus, y por lo tanto es válida..

Las señales de estado S2n-S0n indican el tipo de ciclo de bus (como se indica en la tabla 3) y adquieren un valor determinado hasta la fase 1 de T3 (o del último Tw, cuando aparecen estados de espera)

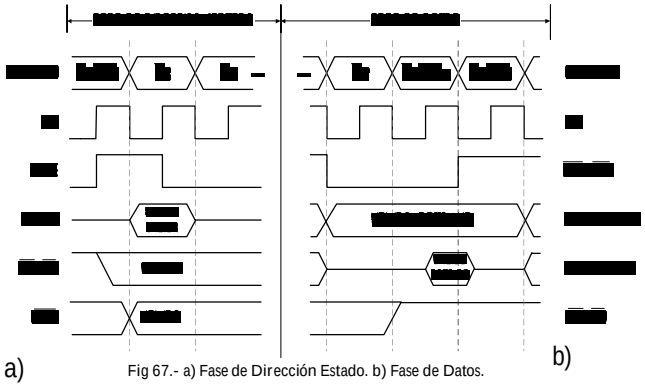


Fig 67.- a) Fase de Dirección Estado. b) Fase de Datos.

Fase de Datos

En la figura 67 b, podemos ver la relación entre la fase de datos y el estado del bus local (estado T). El único ciclo de bus que no dispone de fase de datos, es halt. Durante la fase de datos, el bus transfiere información entre las unidades internas y la memoria o los dispositivos de entrada y salida (durante la fase de direcciones-estado, se indica cual de estos dispositivos va a ser utilizado). La unidad generadora de ciclos de bus (bcg), se encarga de generar las señales de control adecuadas para coordinar la transferencia de datos.

La fase de datos, comienza en la fase 1 de T2 y continua hasta la fase 2 de T4 o Ti. La longitud de la fase de datos varia en función del número de estados de espera (Tw) que aparecen después de T3 y antes de T4 o Ti.

23.2 ESTADOS

El grafo de estados del bus, es muy sencillo. Consta de 6 estados, dos de ellos (T3 y Tw) se pueden simplificar fácilmente, no los simplifico porque así me parece un grafo más didáctico.

Estos estados son utilizados en la unidad de generación de ciclos de bus (bcg) por lo que están definidos en el package del sistema (n0/r0). A continuación podemos ver el código:

```
                                --Estado del bus externo:
TYPE Bus_State_Typ IS (Ti,  --Estado Ocioso
                       T1,  --Estado T1 - fase Address
                       T2,  --Estado T2 - fase Data
                       Tw,  --Estado de espera
                       T3,  --Estado T3
                       T4); --Estado T4
```

Del estado inicial (Ti), salimos cuando el árbitro (arbiter, n3/r3) le indica que hay una petición de ciclo de bus (do_bus_cycle), momento en el que entramos en la fase de direcciones-estado (T1). Al ciclo siguiente, pasamos a la fase de datos (entramos en T2) y al siguiente ciclo, pasamos al estado T3 donde comprobamos la señal READY correctamente sincronizada (READY_syn). Si es '0', alargamos la fase de datos insertando estados de espera (saltamos a Tw), si es '1' y no hay otra petición de ciclo de bus, saltamos al estado de espera Ti; pero si hay otra petición de ciclo, saltamos a T4 para "lanzar" al ciclo siguiente (estado T1) otro ciclo de bus.

Estados de Espera, Tw.

Los estados de espera, expanden la fase de datos del ciclo de bus. Los dispositivos de entrada/salida o las memorias que no pueden proporcionar los datos en el tiempo mínimo (cuatro ciclos de reloj), requieren ciclos de espera. La figura 68 muestra un ciclo de bus típico con estados de espera.

La señal READY es la "responsable" de que tsm inserte ciclos de espera en la fase de datos. Esta señal es utilizada por los dispositivos externos con los que el microprocesador va a intercambiar datos para indicar cuado el dato está accesible.

La fase de datos se puede alargar hasta el infinito y más allá, si la señal READY está siempre a '0', sin embargo, ciclos de bus con muchos estados de espera limitan el potencial de los modelos y de los periféricos del sistema. El potencial se ve limitado porque la unidad ifu no puede ir en busca de instrucciones (en consecuencia la cola de instrucciones no se llena) y limita el rendimiento de los periféricos porque el ancho de banda del bus local se reduce considerablemente.

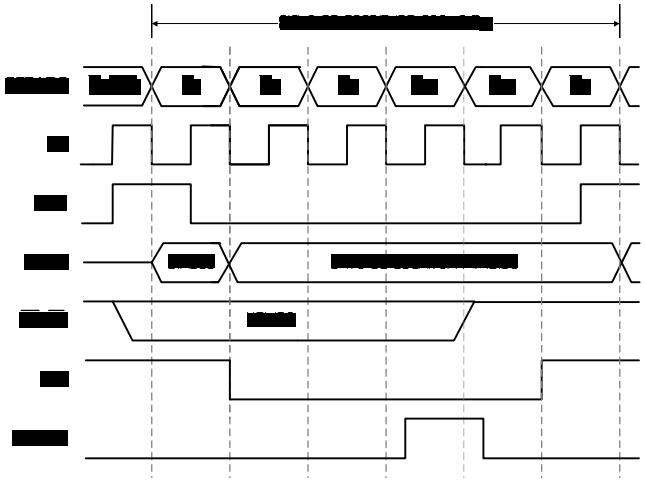


Fig 68.- Típico Ciclo de Bus con Estados de Espeta Tw.

ESTADOS DE ESPERA, TI

En la mayor parte de las operaciones, la BIU ejecuta ciclos de bus consecutivos, sin embargo, algunas condiciones causan que la BIU esté ociosa (idle). Un estado ocioso aparece entre ciclos de bus y el sistema puede permanecer en este estado un tiempo indeterminado.

Las condiciones que pueden hacer que el bus esté ocioso, son las siguientes:

- La cola de instrucciones está llena.
- Se está calculando una dirección.
- El ciclo de bus requiere inherentemente estados de espera.
- La ejecución de la instrucción WAIT, fuerza al bus a entrar en estado ocioso.

23.3 CAMINOS

Los caminos típicos del grafo de la figura 65 b, son:

CAMINO	DESCRIPCIÓN
Ti-Ti	Ciclo de bus ocioso.
Ti-T1-T2-T3-T4-Ti	Un sólo ciclo de bus típico
Ti-T1-T2-T3-T4-x*[T1-T2-T3-T4)-Ti	x ciclos de bus consecutivos
Ti-T1-T2-T3-z*Tw-T4-Ti	Un sólo ciclo de bus con z estados de espera.
Ti-T1-T2-T3-z*Tw-T4-n*[T1-T2-T3-T4)-Ti	n ciclos de bus consecutivos cin z estados de espera.
Ti-T1-Ti	Ciclo de HALT.

Tab 42.- Posibles Caminos de la Unidad TSM.

24. UNIDAD GENERADORA DE CICLOS DE BUS, BCG (n3/r9)

n3/r9 Generador de Ciclos de Bus, Bus Cycle Generator (BCG): Es la unidad encargada de generar las señales de control externas del ciclo de bus en ejecución; el ciclo de bus que determina la lógica de Arbiter (n3/r3).

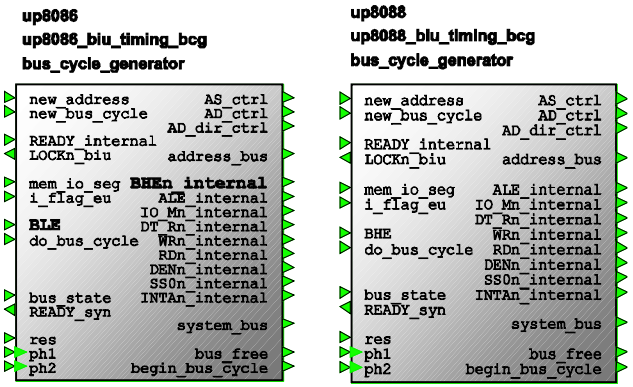


Fig 68.- Carcasas de la BCG para el 8088 y el 8086.

Es una unidad completamente síncrona, que como se indicó en la sección anterior, a partir del estado de bus (estado T) generado por la unidad tsm (n3/r8) y las fases del reloj (ph1 y ph2), genera las señales de control necesarias para controlar las transferencias con las unidades externas. Cada tipo de transferencia con el exterior, tiene un ciclo de bus específico:

S2n	S1n	S0n	TIPO DE BUS
0	0	0	Reconocimiento de interrupción
0	0	1	Lectura de I/O
0	1	0	Escritura de I/O
0	1	1	Halt
1	0	0	Búsqueda de instucción
1	0	1	Lectura de datos desde la memoria
1	1	0	Escritura de datos en la memoria
1	1	1	Pasivo (No hay ciclo de Bus)

Tab 43.- Tipos de Ciclos de Bus.

24.1 ENTRADAS Y SALIDAS

SEÑAL	TIPO	FUNCIÓN
ph1, ph2	IN	Reloj interno
res	IN	Es la señal interna que resetea todos los componentes del modelo
bcg <-> ports		
AS_ctrl	IN	Control del multiplexor del puerto AS
AD_ctrl	IN	Control del Multiplexor del puerto AD
AD_dir_ctrl	IN	Controsa si AD es de salida o de entrada.
address_bus	OUT	Dirección que va a salir por los puertos A, A?
system_bus	OUT	Bus de sistema, saldrá por AS, y (S0n-S2n/DT_Rn-IO_Mn-SS0)
READY_internal	IN	Señal de READY sincronizada.
ALE_internal, SS0n_internal, RDn_internal, WRn_internal, IO_Mn_internal, DT_Rn_internal, DENn_internal, INTAn_internal	OUT	Señales del bus de control generadas por la unidad de ciclos de bus
BHEn_internal	OUT	Habilita la parte alta del banco de memoria
LOCKn_biu	OUT	Señal interna del puero LOCKn generada por la biu (n1/r7)
bcg <-> interface		
bus_free	OUT	Indica a la unidad de interface que el bus está libre
begin_bus_cycle	OUT	Indica que ha comenzado un ciclo de bus
mem_io_seg	IN	Se trata de un segmento de memoria o la memoria de entrada salida.
new_bus_cycle	IN	Le indica el ciclo de bus que tiene que lanzar.
new_address	IN	Le pasa la dirección que tiene que poner en el bus de direcciones.
BHE	IN	Transferencia del byte alto.
BLE	IN	Transferencia del byte bajo.
bcg <- cgrom		
i_flag_eu	IN	Dato a sacar por el bus de sistema.
bcg <-> tsm		
bus_state	IN	Estado T del bus
READY_syn	OUT	Señal de READY convenientemente sincronizada

Tab 44.- Tabla de Entradas y Salidas de la Unidad Generadora de Ciclos de Bus, BCG.

24.2 EXPLICACIÓN

Comenzaremos explicando los distintos tipos de ciclos de bus y el momento, en relación al estado de bus (bus_state) y a las fases de reloj (ph1 y ph2), en el que aparecen las señales Para ello, he creado para cada ciclo tipo de bus un cronograma en el que se puede apreciar la fase en la que aparece la señal en el bus de control.

Finalmente, veremos los procesos que componen las unidades bcg de ambos modelos y las diferencias que existen entre estas.

24.2.1 CICLOS DE BUS

Existen cuatro tipos básicos de ciclos de bus: de lectura, de escritura, el de reconocimiento de interrupción y el de halt.

Los ciclos de reconocimiento de interrupción y halt definen operaciones especiales por lo que las trataremos por separado un poco más adelante.

Los ciclos de lectura, incluyen el ciclo de lectura de memoria y el de dispositivos de entrada-salida, además del lanzado por la unidad ifu (n3/r7) para ir en busca de instrucciones.

Los ciclos de escritura incluyen los ciclos de escritura de memoria y dispositivos de entrada y salida. Los ciclos de lectura y escritura, tienen un formato básico similar, de ahí que en el código de bcg sean tratados en conjunto, ver los procesos internos de los ficheros [up8088/up8086]_biu_timing_bcg.vhd.

Las siguientes secciones muestran la sincronización de los ciclos de bus.

Para determinar los peores tiempos, recomiendo analizar los cronogramas del "data sheet" de los micros originales en los archivos "8086 16-bit HMOS Microprocessor DataSheet.pdf" y "8088 8-bit HMOS Microprocessor DataSheet.pdf".

CICLOS DE BUS DE LECTURA

La figura 69 muestra un ciclo típico de lectura. La siguiente tabla (tabla 45), muestra los tres tipos de ciclos de bus de escritura y una breve descripción.

ESTADO			TIPO DE CICLO DE BUS
S2n	S1n	S0n	
0	0	1	Lectura de I/O. Iniciado por la unidad de ejecución (n1/r4) por la instrucción IN. Las líneas A19-A16 son puestas a '0'.
1	0	0	Instruction Prefetch. Iniciado por la BIU, los datos obtenidos, van a parar a la cola de instrucciones.
1	0	1	Lectura de Memoria. Las íneas A19-A0 seleccionan el byte o la palabra de memoria.

Tab 45.- Ciclos de Bus de Lectura

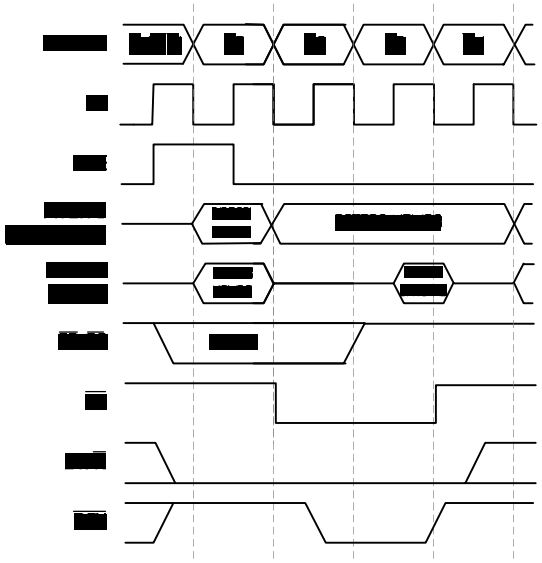


Fig 69.- Ciclo Típico de Lectura

CICLOS DE BUS DE ESCRITURA

La figura 70 muestra un ciclo de escritura típico. El ciclo de bus comienza con la activación de la señal ALE y los bits de sistema S2n-S0n. Termina con la transición a nivel alto de la señal WRn (señal inactiva), aunque los datos permanecen válidos durante un ciclo más. La tabla 46 muestra los dos tipos de ciclos de bus de escritura

ESTADO			TIPO DE CICLO DE BUS
S2n	S1n	S0n	
0	1	0	Escritura I/O. Iniciado por la unidad de ejecución (n1/r4) por la instrucción OUT. Las líneas A15-A0 seleccionan el puerto.
1	1	0	Escritura en Memoria. Iniciado por alguna instrucción para la transferencia de Byte o Word

Tab 46.- Ciclos de Bus de Escritura.

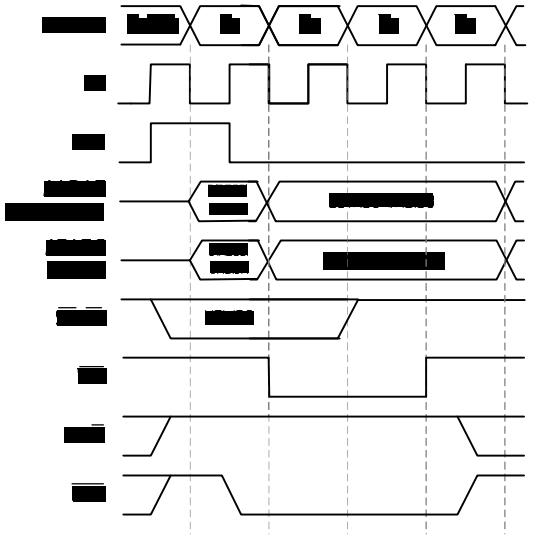


Fig 70.- Ciclo Típico de Escritura

CICLOS DE BUS DE RECONOCIMIENTO DE INTERRUPCIÓN

La expansión de las interrupciones (el microprocesador sólo dispone de dos interrupciones externas), la proporciona una unidad externa de control de interrupciones como puede ser el 82C59A de Intel. La BIU controla los ciclos de bus requeridos para ir en busca del vector de interrupción que proporciona (normalmente) el controlador de interrupciones externo. Estos ciclos de bus (que en conjunto llamamos ciclos de reconocimiento de interrupción), operan de forma similar a los ciclos de lectura. Sin embargo, en lugar de sacar la señal de lectura (RDn), utiliza la señal INTAn.

La figura 71 en la página siguiente, muestra un ciclo típico de reconocimiento de interrupción (o INTA).

Un reconocimiento de interrupción está compuesto de dos ciclos de bus consecutivos. La BIU activa la señal de bloqueo (LOCK_biu) para indicar que se va a realizar una operación secuencial que no puede ser interrumpida por otro maestro e canal.

En el segundo ciclo de bus, se recoge el vector de interrupción presente en la parte baja del bus AD (AD7-AD0). La parte alta del bus AD [A] permanece en estado de alta impedancia durante este ciclo.

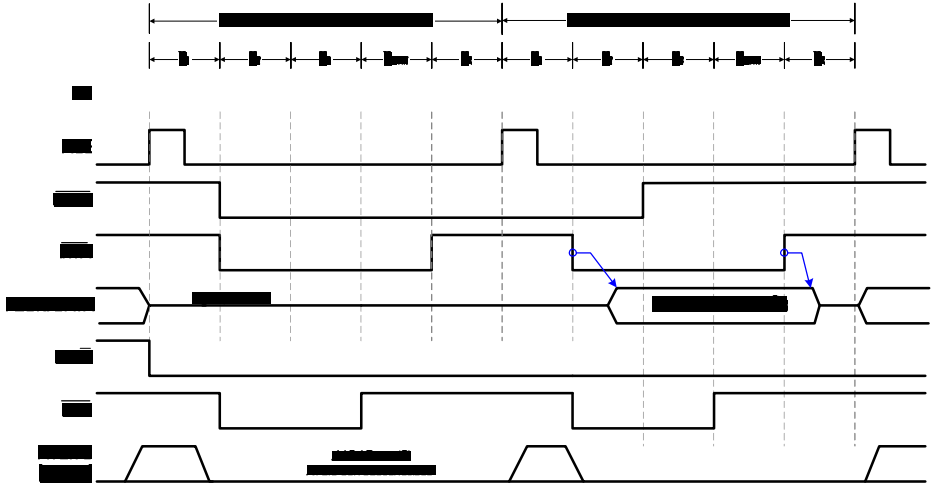


Fig 71.- Ciclo Típico de Reconocimiento de Interrupción.

CICLO DE BUS HALT

Suspendiendo el funcionamiento de los modelos haciéndolos entrar en estado "ocioso" conseguimos, por un lado, reducir el consumo dinámico de la FPGA (las unidades entran en modo de espera y no hay cambios en su estado interno), y por otro lado, reducimos la latencia del sistema ante la aparición de interrupciones (el sistema responde más rápido a una petición de interrupción).

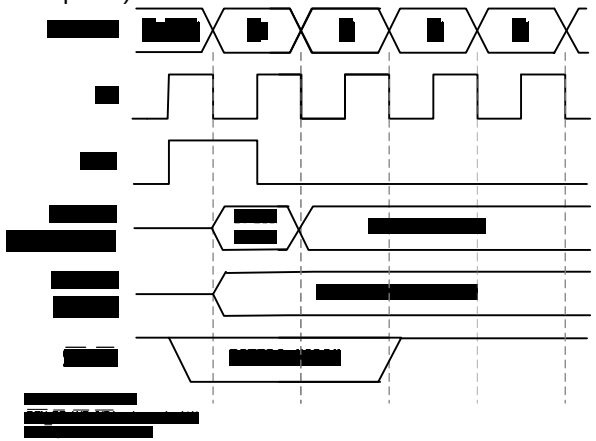


Fig 72.- Temporización de un Ciclo de Bus HALT.

La instrucción HLT lanza dos eventos en el sistema:

- Suspende la ejecución de instrucciones en la unidad de control EU (n1/r4) y
- "induce" a la unidad de control del Bus BIU (n1/r7) a ejecutar el ciclo de bus HALT.

Después de ejecutar un ciclo HALT, la BIU suspende su funcionamiento hasta que ocurre alguno de los siguientes eventos.

- Aparece una Interrupción (NMI_internal o INTR_internal e I_flag='1', ver la unidad exe n2/r2).
- Aparece una petición de bus local (HOLD).

En la figura 72 podemos ver un ciclo HALT. El bus de direcciones/datos y la señal DT/Rn adquieren un valor indeterminado. Las señales del bus de control DENn, RDn, WRn e INTAn son '1' (inactivas) mientras dure el estado HALT..

24.3 PROCESOS

Esta unidad, tiene dos procesos síncronos, el proceso clk_ph1, que activa las señales que aparecen el la fase ph2 y el proceso clk_ph2, que se encarga de las señales que aparecen en la fase ph2.

Ambos procesos, utilizan estructuras lo más paralelas posibles para evitar retardos en la activación de las señales. En lugar de utilizar secuencias IF, utilizo CASE que como se observa en la figura 73 genera hardware más rápido.

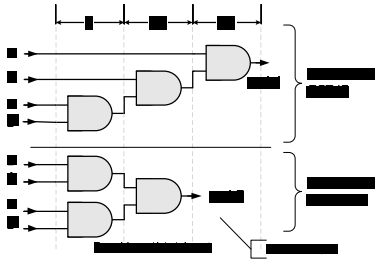


Fig 73.- Los Estamentos CASE son más Rápidos. que los IF.

El siguiente código, muestra parte del proceso clk_ph1. Concretamente muestra como los ciclos de bus de escritura, activan la señal WRn_internal en los estados de bus T1 (registra fase 1 de T2) y la desactiva WRn_internal='1' en la fase 1 de Ti o T4 cuando la señal de READY vale '1'.

```
--|CiCLOS dE eSCRiTURA                                    |--
--Utilizo estructuras lo más paralelas posibles para
--reducir los tiempos de propagación de las señales.
CASE actual_cycle IS
WHEN write_io | write_mem => --Ciclos de escritura
    CASE bus_state IS
    WHEN T1 =>
        WRn_internal_cld <='0';       --Reg en T2 ph1
    WHEN T2 =>
        --Mantengo los valores del bus
        WRn_internal_cld <= WRn_internal_cld;
    WHEN T3 | Tw =>
        --Mantengo los valores del bus
         WRn_internal_cld <= WRn_internal_cld;
        IF READY_syn_cld ='1' THEN
            WRn_internal_cld <= '1';  --Reg en T4|Ti ph1
        END IF;
    WHEN OTHERS => NULL;
    END CASE; --END CASE bus_setate
[...]
WHEN OTHERS => NULL;
END CASE; --END CASE bus_setate
```

25. UNIDAD DE INTERFACE CON EL BUS, INTERF (n2/r7)

n2/r7 Unidad de Interface con la Memoria (mem_interface): Encargada de las operaciones relacionadas con la transferencia a memoria. Desde el código de microinstrucción de una instrucción que realiza transferencias, se programa el comportamiento de esta unidad que normalmente trabaja en paralelo con la ejecución de la instrucción accediendo a memoria en busca de instrucciones.

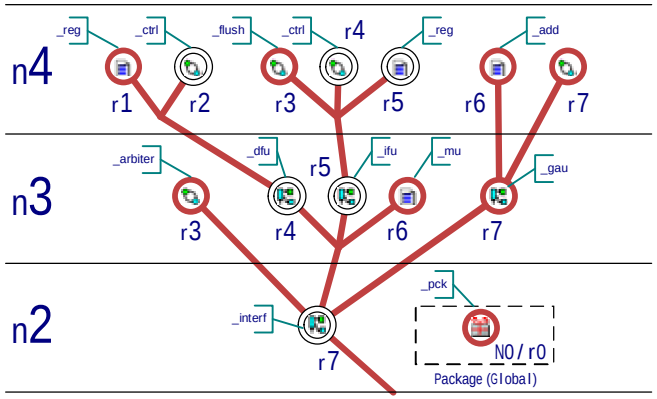


Fig 74.- Detalle de la Jerarquía de la Unidad INTERF.

Es sin duda, la unidad con más niveles jerárquicos de los modelos (como podemos apreciar en el detalle de la jerarquía en la figura 74). El nivel n2/r7 define un esquema que es diferente en los modelos del 8088 y el 8086. En la figura 75 a, podemos ver el esquema definido para el modelo 8088 (fichero "up8088_biu _interf.vhd"). En la figura 75 b, podemos ver la carcasa del modelo 8086 y resaltadas las diferencias con la del modelo 8088 que, como podemos apreciar, son muy pocas, razón por la cual no incluyo el esquema del la unidad de interface con el bus para el modelo 8086 definido en el archivo "up8086_biu_interf.vhd".

Las diferencias entre la unidad de interface con el bus local del modelo 8088 y del modelo 8086 son (referidas al modelo 8088):

- Desaparece la señal BHE y en su lugar aparece la señal BLE. Es un cambio de lógica en la unidad dfu propiciado por la necesidad de alinear la transferencia con memoria cuando escribimos palabras o bytes en

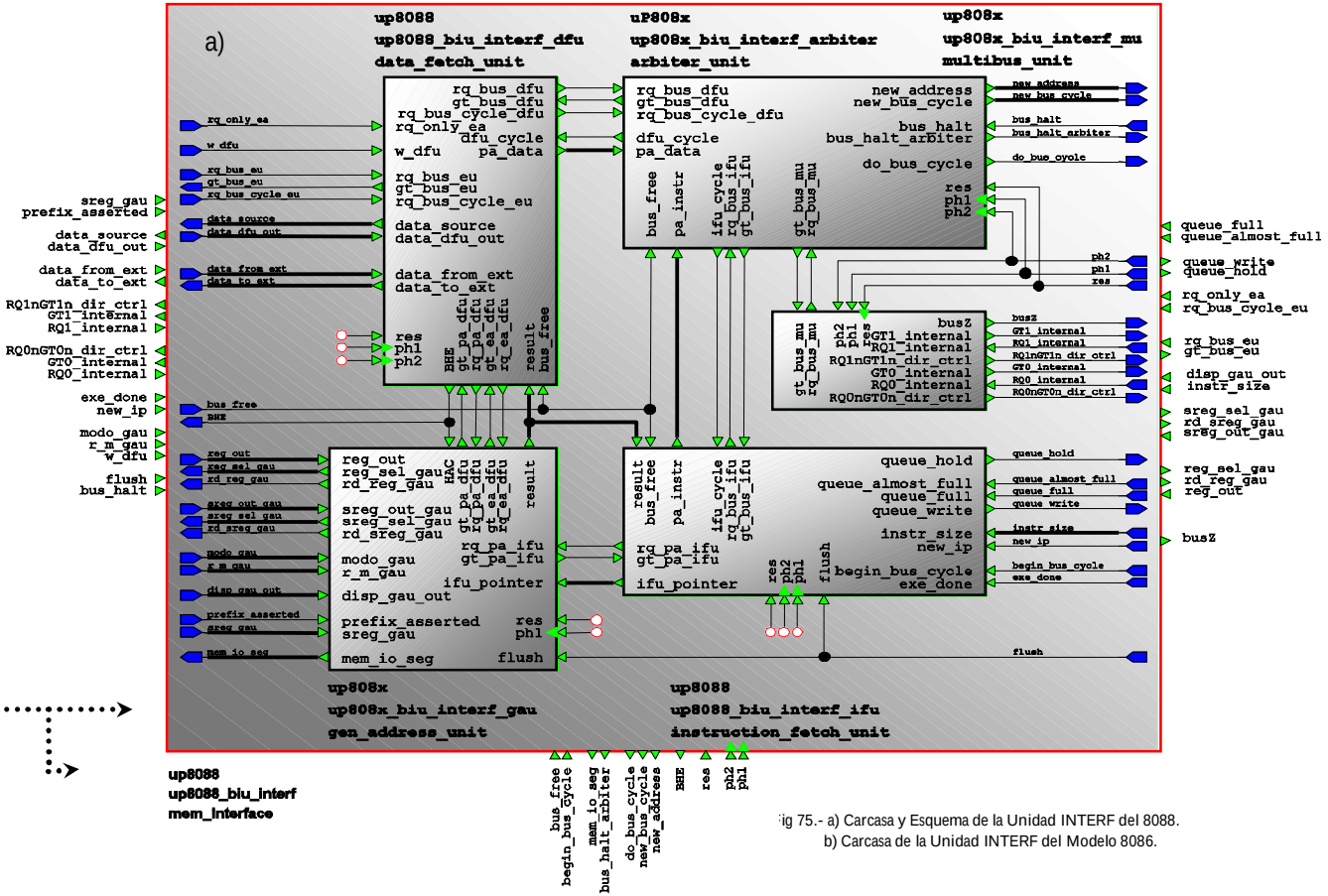
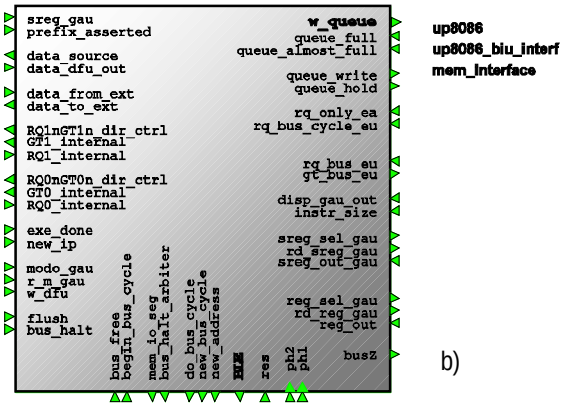


Fig 75.- a) Carcasa y Esquema de la Unidad INTERF del 8088.
b) Carcasa de la Unidad INTERF del Modelo 8086.

direcciones impares. Esto se explica con mayor detalle en la unidad de acceso a datos, dfu (n3/r4).

- La otra diferencia importante es la señal w_queue, necesaria sólo en el modelo del 8086 para indicar a la cola de instrucciones cuando debe escribir un byte o una palabra. En este caso, la lógica de la unidad que va a por instrucciones (ifu, n3/r5), ha sido modificada para alinear la cola de instrucciones con la memoria. Todo esto se explica en detalle un poco más adelante en la sección de la unidad ifu.



25.1 ENTRADAS Y SALIDAS

SEÑAL	TIPO	FUNCIÓN
ph1, ph2	IN	Reloj interno
res	IN	Es la señal interna que resetea todos los componentes del modelo
interf <-> ports		
RQ0nGT0n_dir_ctrl	OUT	Control de dirección del puerto RQ0nGT0n
RQ0_internal	IN	Señales RQ0n y GT0n internas del puerto RQ0nGT0n. Generada por la unidad de control Multibus, mu (n3/r6)
GT0_internal	OUT	
RQ1nGT1n_dir_ctrl	OUT	Control de dirección del puerto RQ1nGT1n
RQ1_internal	IN	Señales RQ1n y GT1n internas del puerto RQ1nGT1n. Generadas por la unidad de control Multibus, mu (n3/r6)
GT1_internal	OUT	
busZ	OUT	Pone todos los puertos con bufer triestado en alta impedancia.
interf <-> queue		
queue_almost_full	IN	Indica que la cola está casi llena
queue_full	IN	La cola de instrucciones está llena
queue_hold	OUT	Congela las operaciones con la cola de instrucciones.
queue_write	OUT	Escribe en la cola de instrucciones el dato en data_from_ext
w_queue	OUT	Indica que escriba una palabra de datos en los registros de la cola.

interf <-> reg/sru		
reg_out	IN	Dato recibido desde rgu.
reg_sel_gau	OUT	Selección del registro de rgu a leer
rd_reg_gau	OUT	La gau indica que va a leer un registro de rgu
sreg_out_gau	IN	Dato recibido desde sru
sreg_sel_gau	OUT	Selección del registro que se va a leer de sru.
rd_sreg_gau	OUT	Indica que se va a leer un registro de sru.
biu <-> cgrom		
flush	IN	Hay un salto en el programa y tenemos que resincronizar.
exe_done	IN	Indica que la unidad de ejecución ha terminado secuencia de microinstrucciones del OP en ejecución
w_dfu	IN	Indica a la unidad dfu (n3/r4) si la transferencia va a ser de byte (si es '0') o palabra (si es'1').
sreg_gau	IN	Transmite el prefijo utilizado para la generación de la dirección física
modo_gau	IN	Programa el modificador me la unidad generadora de direcciones.
r_m_gau	IN	Transmite este campo necesario para que gau que junto con modo indican el modo de direccionamiento.
prefix_asserted	IN	Indica a la gau que no utilice el segmento implícito porque hay activado un prefijo de segmento en la instrucción.

rq_only_ea	IN	EU pide a la biu que le pase la dirección efectiva EA.
rq_bus_cycle_eu	IN	La EU pide a la unidad que va en busca de datos (dfu, n3/r4) que realice el ciclo de bus indicado en esta señal.
rq_bus_eu	IN	EU solicita el control del bus al árbitro de canal
gt_bus_eu	OUT	El árbitro concede el bus a la unidad de ejecución.
data_source	OUT	Datos fuente recibidos desde el exterior tras una petición.
data_dfu_out	IN	Datos que mandamos a la dfu para que los saque cuando corresponda.
disp_gau_out	IN	Indicamos a la gau el desplazamiento con el tiene que calcular la dirección física (PA).
interface <-> timing		
bus_free	IN	Indica a la unidad de interface que el bus está libre
begin_bus_cycle	IN	Indica que ha comenzado un ciclo de bus
mem_io_seg	OUT	Se trata de un segmento de memoria o la memoria de entrada salida.
bus_halt_arbiter	OUT	Indica que pare el bus en cuanto pueda.
do_bus_cycle	OUT	Indica que lance un ciclo de bus.
new_bus_cycle	OUT	Le indica el ciclo de bus que tiene que lanzar.
new_address	OUT	Le pasa la dirección que tiene que poner en el bus de direcciones.
BHE	OUT	Transferencia del byte alto.
BLE	OUT	Transferencia del byte bajo.

Tab 47.- Entradas y Salidas de la Unidad INTERF.

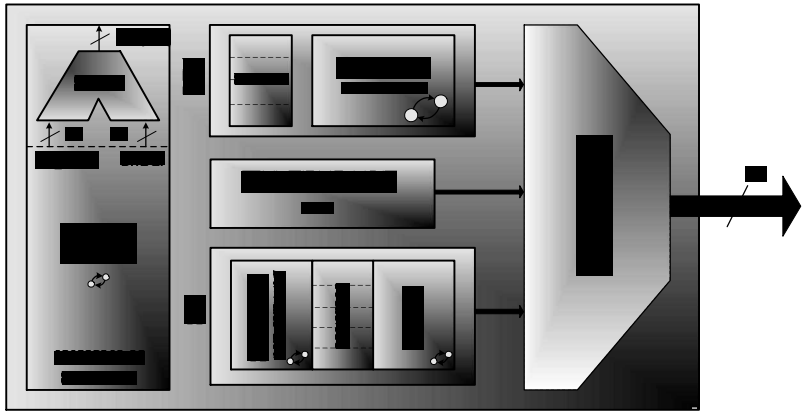


Fig 76.- Unidades de la Unidad INTERF.

En la figura 76, podemos observar que la unidad de interface con el bus está formada por cinco unidades, tres de las cuales (dfu, mu e ifu) acceden al bus local. Para evitar los conflictos que podrían aparecer cuando dos o más unidades accedieran a la vez al bus local, se diseñó una unidad de arbitraje (arbiter, n3/r3) que controlara el acceso al bus. La unidad generadora de direcciones (gau, n3/r7) es una unidad "esclava" de ifu y dfu, se encarga de generar la dirección de memoria de 20 bits con la que estas unidades van a realizar la transferencia. A grandes rasgos, podemos decir que:

- DFU, es la unidad que se encarga de ir a por datos a memoria cuando la unidad de ejecución los necesita.
- IFU, es una unidad independiente de la UE que aprovecha los tiempos ociosos del bus local para ir a por instrucciones a la memoria y colocarlas en la cola de instrucciones.
- MU, es la unidad que controla la cesión del bus local, lógica MULTIBUS.
- ARBITER, es la unidad que arbitra el acceso al bus local.
- GAU, es la unidad encargada de generar la dirección física (PA) que vamos a utilizar.

26 ÁRBITRO DE CANAL, ARBITER (n3/r3)

En el nivel tres (n3), hay dos finales de rama o "yemas". El árbitro de canal (n3/r3) y la unidad de multibus (n3/r6), ambas son comunes para los dos modelos por lo que se encuentran en la librería up808x.

n3/r3 Árbitro de canal (arbiter): Internamente existen varias unidades que acceden al bus local, por lo que es necesario un árbitro interno que establezca las prioridades y las condiciones de acceso. De ello se encarga esta unidad como estudiaremos a continuación.

26.1 ENTRADAS Y SALIDAS

SEÑAL	TIPO	FUNCIÓN
ph1, ph2	IN	Reloj interno
res	IN	Es la señal interna que resetea todos los componentes del modelo
arbiter <-> timing		
new_address	OUT	Es la dirección que la unidad bcg tiene que sacar por el bus de direcciones.
new_bus_cycle	OUT	Le indica a bcg el tipo de ciclo de bus que tiene que realizar.
bus_halt_arbiter	OUT	Le indica a tsm que entre en estado HALT.
do_bus_cycle	OUT	le indica a tsm que "lance" un nuevo ciclo de bus.
bus_free	IN	Indica al árbitro que el bus está libre.
arbiter <-> dfu		
rq_bus_dfu	IN	La unidad dfu pide el control del bus.
gt_bus_dfu	OUT	Indica que le ha concedido el bus a dfu
rq_bus_cycle_dfu	IN	Indica el tipo de ciclo de bus
dfu_cycle	OUT	Indica a dfu que el ciclo de bus actual (y por lo tanto el bus) es suyo
pa_data	IN	Es la dirección física con la que se va a realizar la transferencia.
arbiter <-> ifu		
rq_bus_ifu	IN	La unidad ifu pide el control del bus.
gt_bus_iful	OUT	Indica que le ha concedido el bus a ifu
ifu_cycle	OUT	Indica a ifu que el ciclo de bus actual (y por lo tanto el bus) es suyo
pa_instr	IN	Es la dirección física con la que se va a realizar la transferencia.
arbiter <-> mu		
rq_bus_mu	IN	La unidad mu pide el control del bus.
gt_bus_mul	OUT	Indica que le ha concedido el bus a mu
arbiter <-> eu		
bus_halt	IN	La cgrom indica que hay que entrar en bus HALT (instrucción HLT)

Tab 48.- Entradas y salidas de ARBITER (archivo " up808x_biu_interf_arbiter.vhd").

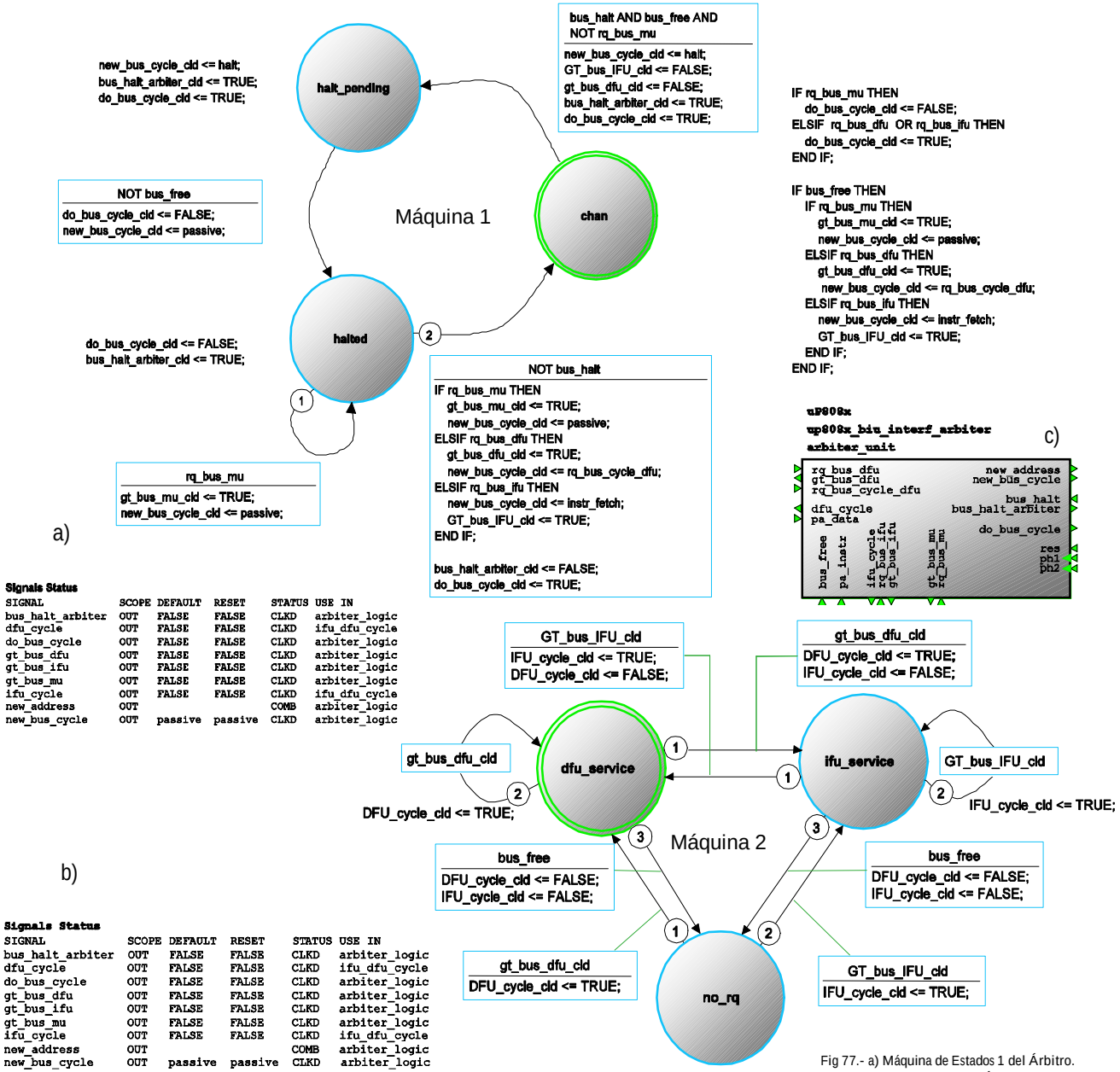


Fig 77.- a) Máquina de Estados 1 del Árbitro.
b) Máquina de Estados 2 del Árbitro.
c) Carcasa de la Unidad

26.2 FUNCIONAMIENTO

El árbitro está compuesto internamente por dos máquinas de estados:

- Máquina de Estados 1 (arbiter_logic), este grafo que podemos ver en la figura 77 a, es el principal de esta unidad. Es el encargado de atender las peticiones de las unidades en función de la prioridad y el estado de la señal bus_halt.
- La Máquina de Estados 2 (ifu_dfu_logic) figura 77 b, es una unidad secundaria encargada del control de las señales ifu_cycle y dfu_cycle que indican a la unidad ifu y dfu respectivamente si tienen o no el control sobre el bus local.

MÁQUINA DE ESTADOS 1, ARBITER_LOGIC

Esta es la máquina de estados encargada de pedir a la unidad tsm (n3/r8) que lance un nuevo ciclo de bus (señal do_bus_cycle). El ciclo de bus que tiene que ejecutar la unidad bcg (n3/r9) es el indicado en la señal "new_bus_cycle" y la dirección que tiene que sacar es la que le pasa el árbitro en "new_address". Estas dos señales (new_address y new_bus_cycle), están multiplexadas, como podemos ver en la siguiente figura 78 b.

La lógica de control de estos multiplexores, está implementada en el estado CHAN:

```
IF rq_bus_mu THEN
    do_bus_cycle_cld <= FALSE;
ELSIF  rq_bus_dfu  OR rq_bus_ifu THEN
    do_bus_cycle_cld <= TRUE;
END IF;

IF bus_free THEN
    IF rq_bus_mu THEN
        gt_bus_mu_cld <= TRUE;
        new_bus_cycle_cld <= passive;
    ELSIF rq_bus_dfu THEN
        gt_bus_dfu_cld <= TRUE;
        new_bus_cycle_cld <= rq_bus_cycle_dfu;
    ELSIF rq_bus_ifu THEN
        new_bus_cycle_cld <= instr_fetch;
        GT_bus_IFU_cld <= TRUE;
    END IF;
END IF;
```

```
--Multiplexor de new_address
new_address <= pa_data WHEN  gt_bus_dfu_cld ELSE
               pa_instr WHEN gt_bus_ifu_cld ELSE
               X"00000";
```

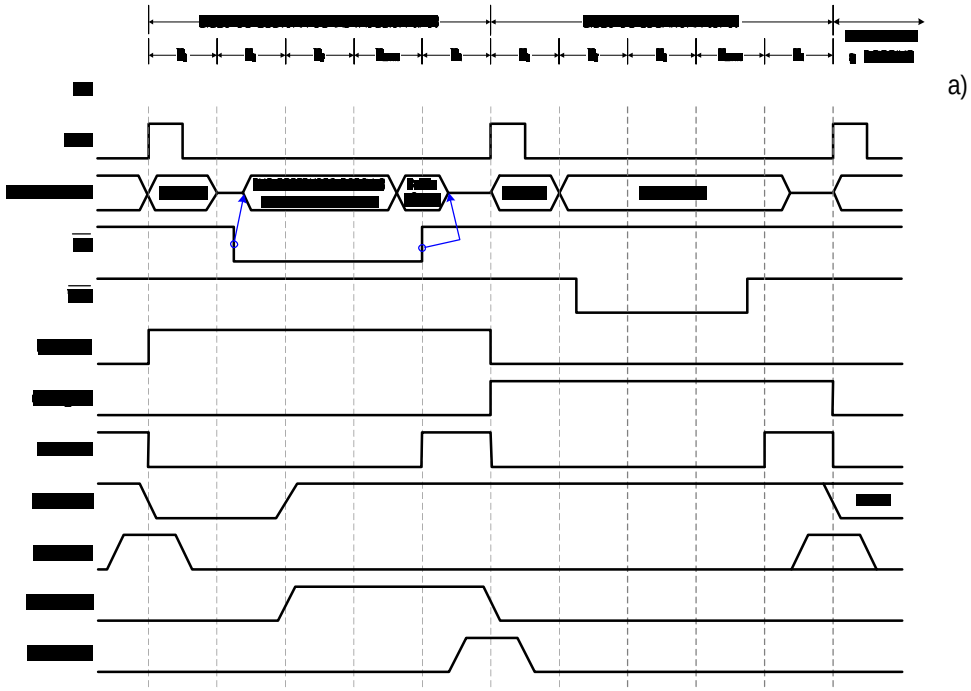
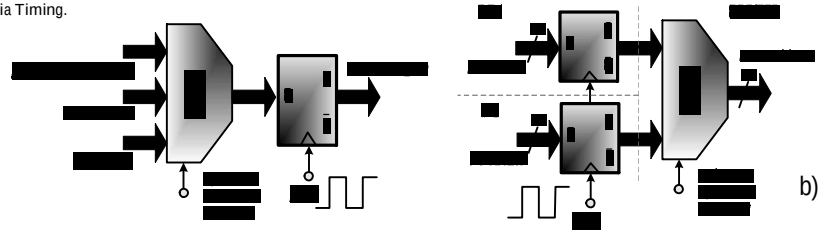


Fig 78.- a) Cronograma de la Unidad Arbiter.
b) Multiplexación de las Señales hacia Timing.



La multiplexación de la dirección física PA_inst y PA_data en new_address, se realiza de manera combinacional con las siguientes líneas de código

Los otros estados se encargan de sincronizar la petición de bus HALT (intrucción HLT) con el bus externo.

Estado inicial CHAN (e0). Es el estado natural (típico) para el cambio del tipo de ciclo de bus. En el código acción de este estado, podemos ver la multiplexación de la figura 74 b.

También podemos observar como el árbitro pide un ciclo de bus (do_bus_cycle) cuando hay una petición y cómo responde a las peticiones activando las señales de concesión gt_bus_mu, gt_bus_dfu y gt_bus_ifu. La prioridad queda establecida por la secuencia IF siendo la unidad MU la de mayor prioridad e ifu la que menor.

halt_pending (e1). Cuando aparece una petición de halt, el árbitro ignora cualquier otra petición y salta a este estado en espera de sincronizar el bus. Una petición de HALT tiene mayor prioridad que cualquiera de las peticiones de las unidades MU, IFU o DFU. En este estado el árbitro pide a tsm y bcg (unidad de

temporización del bus, Timing n2/r8) que ejecute un ciclo de bus HALT, que como vimos en la sección anterior sólo tiene fase de direcciones-estado. Cuando el bus deja de estar libre (e1/t1), podemos asegurar que hemos entrado en estado ocioso y por lo tanto el sistema está parado (HALTED, saltamos al siguiente estado).

halted (e2). Con el sistema parado (halted), el bus está ocioso. Sólo podemos salir de este estado de dos formas:

- si en este estado, aparece una petición del bus local (e2/t1, rq_bus_mu), cedemos inmediatamente el bus (situación de menor latencia del bus o tiempo de respuesta a una petición de bus) ya que el bus está irremediablemente ocioso (Ti). Concedemos el bus, pero el sistema no abandona el estado HALT.
- si lo que aparece es una interrupción (mascarable y aceptada o no mascarable), la unidad de ejecución deshabilita la bandera bus_halt (e2/t2) y, consecuentemente, salimos del estado HALTED para saltar al estado de acción normal CHAN.

En ambas situaciones, el tiempo de respuesta a una petición de bus o a una interrupción, es de 1 ciclo. Estamos en las condiciones de menor latencia del sistema. De hecho, la instrucción HLT se utiliza en muchas ocasiones para sincronizar interrupciones hardware con el software

MÁQUINA DE ESTADOS 2, IFU_DFU_CYCLE

Es una máquina de estados secundaria, simplemente indica que unidad está en la actualidad en disposición del control del bus local (dfu_cycle si está en control de la unidad dfu y ifu_cycle si está en control de ifu). Consta de tres estados, de los cuales dfu_service e ifu_service pueden ser simplificados. El funcionamiento es sencillo, alternamos entre los estados dfu_service y ifu_service en funión de la unidad a la que la máquina de estados 1 ha cedido el bus (gt_bus_ifu o gt_bus_dfu). Si el bus está libre y no hay ninguna concesión de bus, el bus está ocioso sin ser controlado por ninguna de las unidades internas.

Estado inicial DFU_SEVICE (e0). Encargado de activar la señal dfu_cycle cuando la máquina de estados 1 ha cedido el bus a dfu (gt_bus_dfu).

IFU_SEVICE (e1). Encargado de activar la señal ifu_cycle cuando la máquina de estados 1 ha cedido el bus a ifu (gt_bus_ifu).

NO_RQ (e2). Estado en el que ninguna de las unidades tiene el control sobre el bus local.

En la figura 78 a podemos ver un cronograma donde aparecen varias señales de esta unidad en sincronismo con las señales externas para ver su relación.

La señal "bus_free" generada por la unidad de ciclos de bus (bcg, n3/r9) indica al árbitro cuando tiene que analizar las entradas de solicitud de bus (en la gráfica aparecen las señales de petición por parte de la unidad dfu , señal rq_bus_dfu, y la de petición por parte de la unidad ifu, rq_ifu_dfu). Esta señal (la señal bus_free) es la "puerta de entrada" al bus, y establece la máxima de todo sistema síncrono que podríamos enunciarla como: "en un sistema síncrono, el cambio en las características del proceso modelado, debe hacerse en un solo punto", punto que denominamos punto o "puerta" de entrada. Estableciendo un único periodo de tiempo en el que puede cambiarse el tipo de bus en ejecución, y estableciéndolo en el instante adecuado (bus_free) aumentamos la estabilidad de la unidad, y por ser esta unidad la que se comunica con el exterior, aumentamos la estabilidad de todo el sistema. Se pueden establecer más puntos de entrada, pero el sistema tiene que controlar muchas posibles combinaciones y establecer una lógica de acción en función del estado actual y el momento en el que aparece la condición . El sistema es menos estable.

Cuado el bus está libre, analiza las peticiones, y en función de la prioridad, concede o no el bus. En la gráfica 78 a, podemos ver como cuando el bus está libre, sólo hay una petición por parte de la unidad ifu, petición que es concedida (gt_bus_ifu) y el siguiente ciclo de bus es el indicado por la unidad arbiter en la señal new_bus_cycle (en el caso de petición por parte de la unidad ifu, es siempre instr_fetch), la dirección que aparece en el bus externo, es la que pasa el árbitro en new_address que al conceder el bus a la unidad ifu es PA_intr.

La segunda vez que el bus local pasa a estar libre, hay dos peticiones, entra en juego la prioridad y el árbitro concede el bus (activando la señal gt_bus_dfu) a la unidad dfu (de mayor prioridad). DFU detecta que le ha sido concedido el bus, y borra su petición. El siguiente ciclo de bus, controlado por dfu, es el indicado en la señal rq_bus_cycle_dfu (en este caso un ciclo de escritura). La dirección con la que se realiza la transferencia es la que dfu pasa al árbitro en la señal PA_data. El árbitro se encarga de multiplexar estos datos el las salidas new_address y new_bus_cycle, que pasa a la unidad timing (n2/r8) de la biu.

La tercera vez que aparece la señal bus_free, podemos encontrarnos con dos situaciones:

- La cola de instrucciones está llena, en respuesta, la unidad ifu no pide el bus local (rq_bus_ifu inactivo); no necesita ir a por más instrucciones a memoria. En este caso, el siguiente ciclo de bus será un ciclo de pasivo y el bus entrará en estado Ti. En estas condiciones, como cuando estamos en HALT, nos encontramos en situación de mínima latencia (o respuesta más rápida) ante posibles interrupciones o peticiones del bus local.
- En caso de que la cola de instrucciones no esté llena, la señal rq_bus_ifu estará activa, por lo que el siguiente ciclo de bus será un ciclo de "intr_fetch". Múltiples ciclos "instr_fetch" se repiten hasta llenar la cola de instrucciones.

Todo esto se explica mejor en la sección de la unidad IFU un poco más adelante.

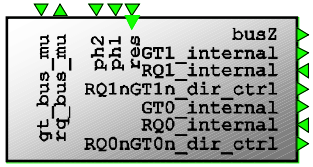
• • • •

27 UNIDAD DE MULTIBUS, MU (n3/r6)

Esta es la última "yema" del nivel 3 que nos resta por explicar

27.1 ENTRADAS Y SALIDAS

SEÑAL	TIPO	FUNCIÓN
ph1, ph2	IN	Reloj interno
res	IN	Es la señal interna que resetea todos los componentes del modelo
multibus_unit <-> puertos		
bus_z	OUT	Pune en estado de alta impedancia los buses A, AD y de control.
RQ0_internal RQ1_internal	IN	Entradas de peticiónde bus local de los canales 0 y 1. ch 0 tiene mayor prioridad que ch1
GT0_internal GT1_internal	OUT	Sacan el pulso que indica que el bus está cedido.
RQ0nGT0n_dir_ctrl RQ1nGT1n_dir_ctrl	OUT	Indica la dirección de los puertos de entrada salida RQ0nGT0n y RQ1nGT1n.
multibus_unit <-> arbitro		
rq_bus_mu	OUT	La unidad mu pide el control del bus.
gt_bus_mul	IN	Indica que le ha concedido el bus a mu



up808x

up808x_biu_interf_mu

multibus_unit

Tab 49.- Entradas y salidas de MU (archivo " up808x_biu_interf_mu.vhd").

n3/r6 Unidad de Multibus (Multibus): Esta unidad es la encargada de dar soporte a las peticiones de bus local por parte de otros maestros de canal. La sincronización de la petición con el micro se hace a través del árbitro interno (n3/r3).

Aunque, este fichero esté escrito en VHDL (como podemos observar en la figura 74), en realidad es un grafo de estados que codifiqué a mano.

```
-----------------------------------------------------------------------------
nextstate : PROCESS (gt_bus_mu,RQ0_internal,RQ1_internal,inserv_ch0,
    inserv_ch1,rq_ch1,rq_ch0,current_state)
-----------------------------------------------------------------------------
BEGIN
    CASE current_state IS
    WHEN Reposo =>
    --1. Primer Pulso. Pulso de petición de bus desde dispositivo externo.
        IF (RQ0_internal='1' OR RQ1_internal='1') THEN
            next_state <= RQ_Bus;
        ELSE
            next_state <= Reposo;
        END IF;
    WHEN RQ_Bus =>
        --Requiero el control del bus local para cederlo al dispositivo externo.
        IF (gt_bus_mu) THEN
            next_state <= GT_bus;
        ELSE
            next_state <= RQ_Bus;
        END IF;
    WHEN GT_Bus =>
    --2. Segundo Pulso. Indica al dispositivo externo que ya puede utilizar el bus
        next_state <= Multibus_Ack;
    WHEN Multibus_Ack =>
    --3. Tercer pulso. Pulso de "operación en bus terminada"
        IF ((inserv_ch0 AND RQ0_internal='1' AND (RQ1_internal='1' OR rq_ch1)) OR
            (inserv_ch1 AND RQ1_internal='1' AND (RQ0_internal='1' OR rq_ch0))) THEN
            --No recupero el bus, porque hay otra petición
            next_state <= GT_Bus;
        ELSIF (inserv_ch0 AND RQ0_internal='1') OR
              (inserv_ch1 AND RQ1_internal='1') THEN
            --Recupero el bus
            next_state <= Recover_Bus;
        ELSE
            next_state <= Multibus_Ack;
        END IF;
    WHEN Recover_Bus =>
        --Aseguro que no apazca otra petición en el canal actual hasta por lo
        --menos un ciclo de reloj después del pulso de recuperación del bus
        --(pulso 3, ver pág 25 Datasheet). Si aparece, la memorizo en rq.
        IF (inserv_ch0 AND RQ0_internal='0') OR
           (inserv_ch1 AND RQ1_internal='0') THEN
            next_state <= Reposo;
        ELSE
            next_state <= Recover_Bus;
        END IF;
    WHEN OTHERS =>
        next_state <= Reposo;
    END CASE;
END PROCESS nextstate;
```

27.2 EXPLICACIÓN

En el código de arriba, podemos observar la definición del grafo (sus estados y transiciones) en VHDL. A simple vista, podemos darnos cuenta de que existen 5 estados:

- REPOSO, estado inicial (e0). En este estado permanecemos a la espera de alguna petición de bus por parte de dispositivos externos en alguno de los canales (RQ0_internal, RQ1_internal; transición 1 t1).
- RQ_Bus, (e1). En este estado, pedimos el bus local al árbitro de canal (arbiter, n3/r3) activando la señal "rq_bus_mu". Cuando el bus es concedido (gt_bus_mu, t2) saltamos al estado de cesión del bus local, estado GT_Bus (e2).
- GT_Bus, (e2). Cuando MU tiene el control del bus, se lo cede al maestro que hizo la petición

llevando todas las señales a estado de alta impedancia (mediante la activación de bus_z) e indicando que puede utilizar el bus, para lo cual activa la señal GT(0/1)_internal. En este estado sólo permanecemos durante un cilo de reloj. Inmediatamente pasamos a Multibus_Ack

- Multibus_Ack (e3). En este estado mantenemos el bus en alta impedancia y estamos a la espera de recibir el pulso que nos indique que el maestro de canal ha terminado de utilizar el bus local y que podemos recuperarlo. Las señales [RQ0/RQ1]_ internal son las que nos lo indican. En este estado hay dos posibles transiciones: t1 si en el momento de recibir el tercer pulso, en el otro canal aparece un pulso de petición o existe una petición anterior, no "soltamos el canal" y saltamos al estado e2 atendiendo la petición. t2 si aparece el tercer pulso y no existe otra petición, saltamos al estado de recuperación del bus Recover_Bus (e4).
- Recover_Bus, (e4). En este estado recupero el estado normal del bus local dejando este de estar en estado de alta impedancia. Hay que asegurar que no aparece otra petición de bus hasta, por lo menos, un ciclo después de haber recuperado el bus local. De esta manera evitamos múltiples transiciones "pesadas" en los puertos de salida. Una vez que hemos asegurado esta condición, saltamos al estado inicial e0.

27.2.1 SECUENCIA DE PETICIÓN-CESIÓN (REQEST-GRANT) DEL BUS LOCAL

Petición/Cesión, RQuest/GranT (RQ?nGT?n): Estos terminales son utilizados por otros maestros de canal para forzar al microprocesador a dejar libre el bus al final del ciclo de bus actual. Son pines bidireccionales teniendo RQ0n/GT0n mayor prioridad que RQ1n/GT1n. RQn/GTn. En la figura 79 podemos ver un cronograma típico de la cesión del bus local con la secuencia de pulsos:

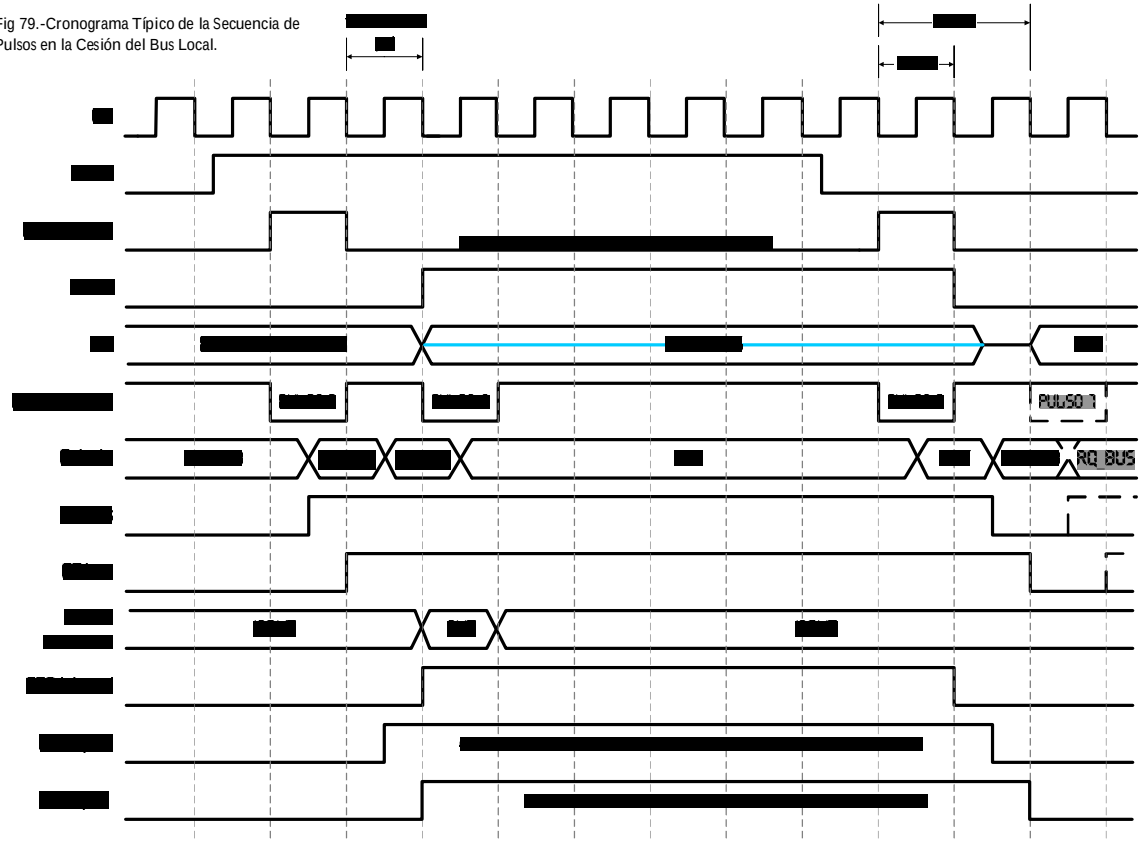
1. Un pulso (pulso 1) de un ciclo de reloj de otro maestro de canal indica que otro dispositivo requiere el bus local para realizar alguna operación.
2. Durante T4 o Ti, la unidad de multibus MU manda un pulso (pulso 2) de un ciclo de reloj al maestro de canal que hizo la petición indicándole que ha puesto el bus local en estado de alta impedancia y que al siguiente ciclo de reloj entrará en el estado de reconocimiento de hold "hold acknowledge". La unidad de comunicación con el bus (BIU) permanece desconectada a nivel lógico del bus local durante "hold acknowledge".
3. Un tercer pulso de un ciclo de duración, indica a la unidad MU que la operación con el bus va a terminar y que puede recuperar el control sobre el bus local en el siguiente ciclo de reloj.

Cada intercambio del control del bus local entre maestro-maestro de bus es una secuencia de tres pulsos. Debe haber un estado ocioso después de cada intercambio de control del bus local. Los pulsos se activan a nivel bajo.

Si aparece una petición de bus mientras le micro está realizando una transferencia, cederá el bus en T4 después del ciclo actual, si se cumplen las siguientes condiciones:

1. La petición aparece antes o durante T2.
2. El ciclo de bus actual, no está transfiriendo el primer byte de una transferencia de 2 bytes.
3. El ciclo actual no es el primero de un ciclo de reconocimiento de interrupción.
4. No se está ejecutando una instrucción de bloqueo (prefijo LOCK).

Fig 79.-Cronograma Típico de la Secuencia de Pulsos en la Cesión del Bus Local.



27.2.2 SITUACIONES ESPECIALES

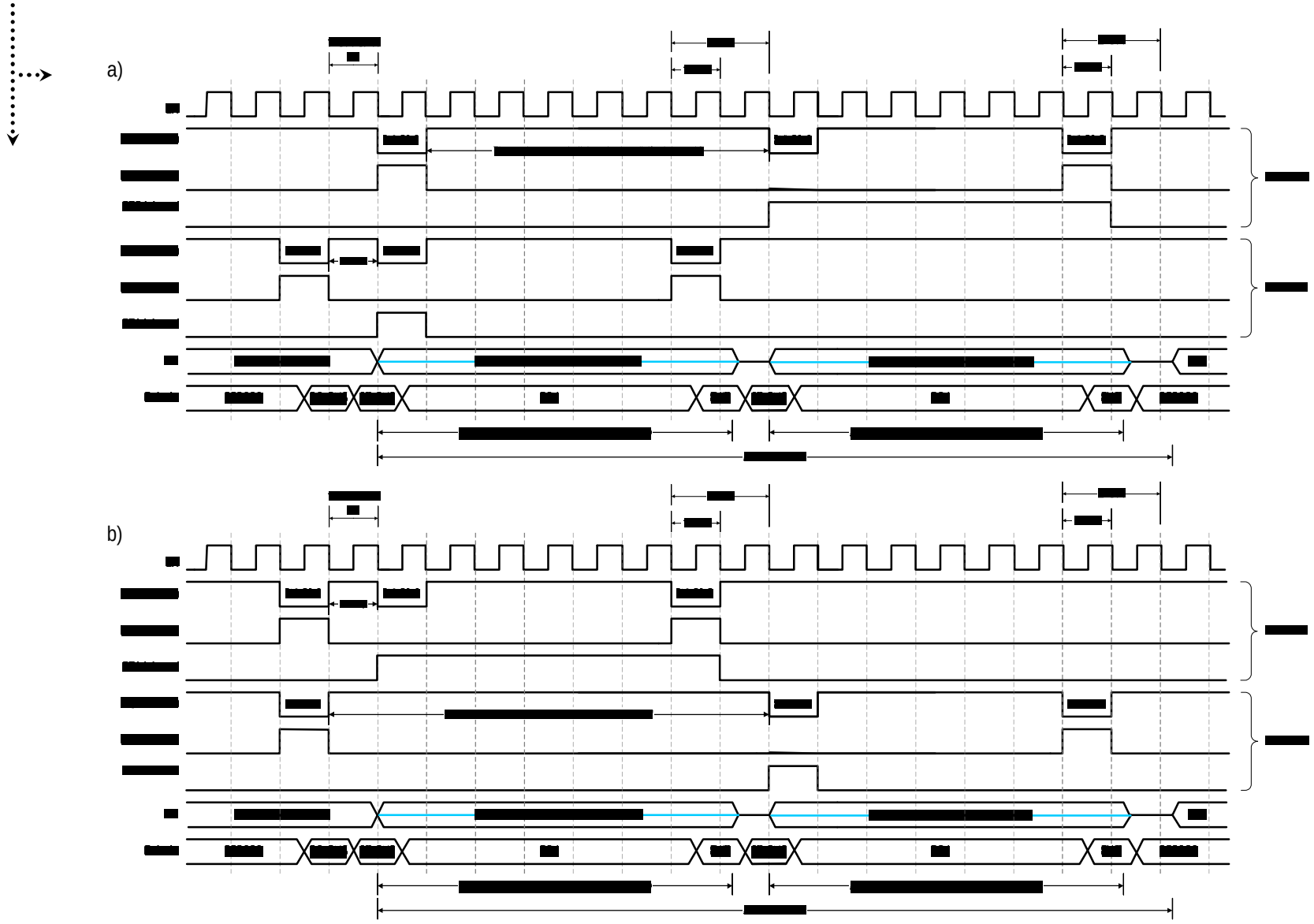
En la cesión del bus local cuando trabajamos en modo máximo, pueden darse algunas situaciones especiales a las que tenemos que dar solución. Estas situaciones "especiales" son causadas por la aparición de peticiones en ambos canales (sólo están activos los dos canales en modo máximo). A continuación podemos ver dos de estas situaciones especiales.

Aparece Una Segunda Petición Cuando ya Está Cedido el Bus.

Como podemos ver en la figura 80 a, la petición del canal ch0 es memorizada y atendida cuando finaliza la cesión del bus local al canal ch0. Lo mismo ocurriría si el canal está cedido a ch0 y aparece una petición en ch1. Observar cómo la MU no pierde el control del canal (no salta al estado REPOSO).

Aparecen Dos Peticiones a la Vez

En la Figura 80 b podemos apreciar esta situación. En este caso, entra en juego la prioridad de canal, ch0 tiene mayor prioridad que ch1, por lo que el bus es cedido al dispositivo del canal ch0 quedando memorizada la petición del canal ch1. La petición de ch1 es atendida cuando el maestro de bus del canal ch0 termina de utilizar el bus (pulso 3). ••••••••••



28 UNIDAD GENERADORA DE DIRECCIONES, GAU (n3/r7)

En el nivel 3 sólo nos queda por explicar la unidad generadora de direcciones, que está formada por dos unidades internas.

n3/r7 Unidad de Generación de Direcciones, Gen Address Unit (GAU): Esta unidad se encarga de generar la dirección de 20 bits (Phisical Address, PA=Segmento&"0000"b+Desplazamiento) con la que IFU o DFU van a realizar la transferencia. La programación (tipo de direccionamiento, existe o no prefijo de segmento, cual es..) la realiza la unidad de ejecución EU a través de la cgrom (n2/r1).

Como podemos apreciar en el detalle de la figura 82 , esta unidad está compuesta por dos unidades ambas comunes para los modelos y por lo tanto, localizadas en la librería común "up808x". El nivel n3/r7 define un esquema que podemos ver en la figura 81 junto al componente de la unidad generadora de direcciones.

La unidad del sumador (addrer, add n4/r6), es la encargada de realizar las sumas necesarias para calcular la dirección física. Es un sumador de 20 bits. Un poco más adelante, se explicará el funcionamiento completo de esta unidad meramente combinacional.

La unidad de control (ctrl n4/r7) ejecuta (a petición de las unidades dfu e ifu) la secuencia de pasos necesarios para calcular la dirección física.

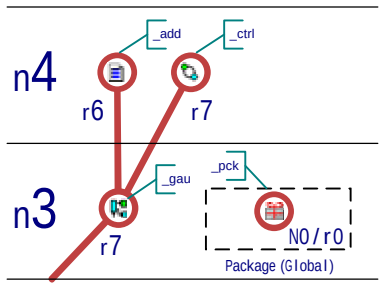


Fig 82.- Detalle de la Jerarquía interna de la unidad GAU.

En la secciones siguientes, explicaremos en detalle estas dos unidades . Analizaremos el grafo de control y la forma en la que las unidades ifu y dfu se comunican con gau para pedirle que calcule una dirección. ••••••

28.1 ENTRADAS Y SALIDAS

SEÑAL	TIPO	FUNCIÓN
ph1	IN	Reloj interno
res	IN	Es la señal interna que resetea todos los componentes del modelo
gau <-> rgu/sru		
reg_out sreg_out_gau	IN	Dato de salida de rgu/srg del registro seleccionado
reg_sel_gau sreg_sel_gau	OUT	Selector del registro de rgu/sru.
rd_reg_gau rd_sreg_gau	OUT	Indica a rgu/sreg que va a leer un registro.
gau <- eu		
modo_gau r_m_gau	IN	Le indica a gau los parámetros mod y r_m.
prefix_asserted sreg_gau	IN	Le indican si existe un prefijo de segmento y en su caso cual es.
disp_gau_out	IN	Le indica a la gau que utilice este desplazamiento.
flush	IN	Le indica que aborte el actual.
gau -> timing		
mem_io_seg	OUT	Le indica a la unidad de temporización del bus si se trata de un segmento de memoria o del segmento de I/O.
gau <-> dfu		
rq_pa_dfu	IN	dfu le pide que genere una dirección física, PA.
gt_pa_dfu	OUT	Le indica a dfu que la dirección ha sido generada.
rq_ea_dfu	IN	dfu le pide que genere sólo la direcciónefectiva EA.
gt_ea_dfu	OUT	Le indica a dfu que EA ha sido generada.
HAC	IN	Le indica a gau que calcule la parte alta de la dirección.
gau <-> ifu		
rq_pa_ifu	IN	La ifu le pide que calcule la dirección física de la siguiente instrucción.
gt_pa_ifu	OUT	Le indica a ifu que ha calculado la dirección física PA.
ifu_pointer	IN	Le pasa el puntero de instrucción para que calcule la PA.
result	OUT	Cuando la gau termina de calcular la dirección, contiene la PA.

Tab 50.- Entradas y Salidas de GAU (archivo " up808x_biu_interf_gau.vhd").

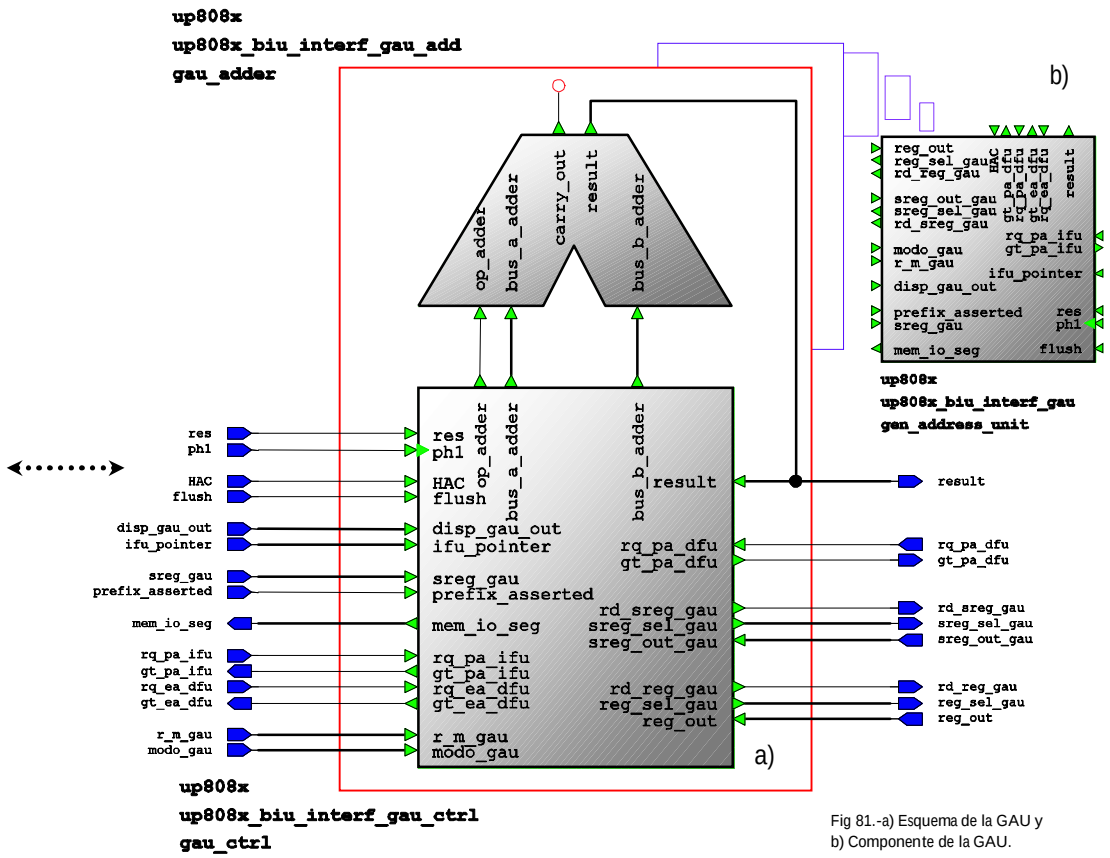
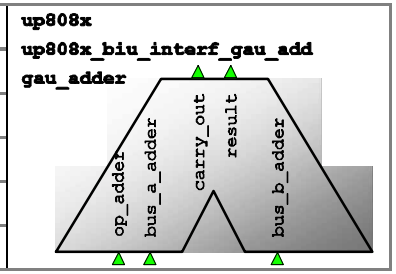


Fig 81.-a) Esquema de la GAU y b) Componente de la GAU.

29. SUMADOR DE LA BIU, ADDER (n4/r6)

n4/r6 Sumador de 20 bits (gau_adder): Es el sumador utilizado por GAU (unidad encargada de generar la dirección física, n3/r7) para sumar el segmento (desplazado cuatro posiciones hacia la izquierda) y el desplazamiento.

SEÑAL	TIPO	FUNCIÓN
bus_a_adder	IN	Operando A.
bus_b_adder	IN	Operando B.
op_adder	IN	Operación a Realizar.
reslult	OUT	Resultado de la Suma
carry_out	OUT	Acarreo de Salida



Tab 51.- Entradas y Salidas de la Unidad ADDER (archivo "up808x_biu_interf_gau_add.vhd").

Este componente tiene una estructura parecida a la utilizada para implementar la unidad lógico aritmética ALU (n2/r6). Como podemos ver en la figura 83, todos los buses son de 20 bits. Los buses bus_a_adder (operando a) y bus_b_adder (operando b) contienen los valores a sumar. Normalmente el operando a es el segmento utilizado para el cálculo de la dirección desplazado 4 bits hacia la izquierda y el operando b contiene el off-set (o dirección efectiva, EA) previamente calculada.

En el siguiente código podemos ver la arquitectura íntegra de esta unidad, que como podemos observar contiene pocas líneas.

La unidad generadora de direcciones GAU, programa el sumador indicándole que realice una determinada operación. Esta indicación se realiza en la variable op_adder que es una señal de tipo enumerado que al ser utilizada también en la gau, está definida en el paquete de declaraciones del proyecto (package, n0/r0) mediante el siguiete código.

```
--Tipo de opreación que realiza el Sumador de la BIU:
TYPE OP_Adder_Typ IS (no_operation,
                      pasar_a,
                      sumar);
```

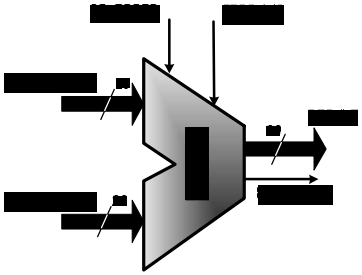


Fig 83.- Sumador de la Unidad Generadora de Direcciones

Cuando se activa la señal carry_out, indica que ha habido un desbordamiento en el cálculo de la dirección. Por ejemplo si CS=X"FF00" y la dirección efectiva previamente calculada para esta transferencia es EA=X"2FFF", el cálculo de la dirección física será PA=CS&X"0"+EA=X"FF000"+X"02FFF=X"01FFF" y el Acarreo de salida será '1'.Esto constituye un fallo grave en la lógica de programa ya que accederemos a una posición de memoria que no es la deseada(PA=01FFFh cuando debería ser PA=101FFFh), perdiendo el control del programa. Los micros originales, no contemplan el desbordamiento de dirección física por hardware, de manera que tampoco lo he implementado, de todas maneras mantengo la salida de acarreo para utilizarla en futuras evoluciones del microprocesador.

```
    --=============================================================================--
    --|                  -GeNERADOR dE lA DiRECCióN FíSiCA, Adder-                 |--
    --|                                                                            |--
    --| Esta unidad se enacarga de generar la dirección física de 20 bits sumando |--
    --|al segmento utilizado (16 bits) desplazado 4 bits hacia la izquierda (20   |--
    --|bists) el desplazamiento (16 bits) calculado en la Unidad Genearadora de   |--
    --|Direcciones (GAU):                                                         |--
    --|                     Phisical Address=Segment&"0000"+Disp                  |--
    --|---------------------------------------------------------------------------|--
    --|                                                                            |--

ARCHITECTURE beh_simpl OF up808x_biu_interf_gau_add IS

BEGIN -- architecture

adder : PROCESS (op_adder, bus_a_adder, bus_b_adder) IS
    --|...........................................................................|--
    --|PrOCEDiMiENTO dE sUMA                                                      |--
    PROCEDURE add (data_a,data_b    : IN Std_Logic_Vector;
                   SIGNAL resu      : OUT Std_Logic_Vector;
                   SIGNAL carry_out : OUT Std_Logic) IS

        VARIABLE b_i    : Std_Logic_Vector(data_a'Length DOWNTO 0);
        VARIABLE resu_i : Std_Logic_Vector(data_a'Length DOWNTO 0);
    BEGIN
        b_i := "0" & (data_b);
        resu_i := ("0" & data_a) + (b_i);
        carry_out <= resu_i(data_a'Length);
        resu <= resu_i(data_a'Length - 1 DOWNTO 0);
    END PROCEDURE add;

BEGIN
    --|...........................................................................|--
    --|PrOCESO PriNCiPAL                                                          |--

    --Valores por defecto
    result <= X"00000";
    carry_out <= '0';

    --Operaciones con la ALU
    CASE op_adder IS
    WHEN no_operation => NULL; --No realiza ninguna operación
    WHEN pasar_a => result <= bus_a_adder; --Pasa e dato del bus_A
    WHEN sumar =>
        --Llamada al procedimiento de Suma
        --Sumador de 20 bits
        add(bus_a_adder,bus_b_adder,result,carry_out);
    WHEN OTHERS => NULL;
    END CASE;
END PROCESS adder;


END ARCHITECTURE beh_simpl;
    --|                                                                            |--
    --|---------------------------------------------------------------------------|--
    --|                        FINAL DEL SUMADOR DE 20 bits                       |--
    --=============================================================================--
```

30. GRAFO DE CONTROL DE GAU, GAU_CTRL (n4/r7)

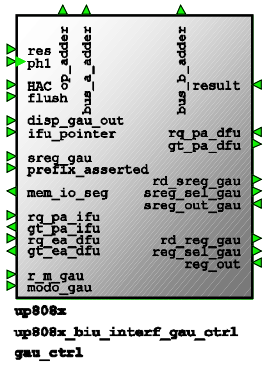
n4/r7 Lógica de Control de GAU (gau_ctrl): Contiene la secuencia de pasos necesarios para generar la dirección física. La dirección física puede tardar en generarse, dependiendo del modo de direccionamiento, de 6 a 12 ciclos en el microprocesador original y de 2 a 4 en el modelo desarrollado. Esta es una de las razones por las que los modelos diseñados son más rápidos que los originales.

La unidad GAU completa y por lo tanto este grafo de estados, es común para ambos microprocesadores (como podemos ver en la figura 82), por lo tanto, podemos encontrar su código en la librería común ("u808x") y concretamente en el archivo " up808x_biu _interf_gau_ctrl.vhd".

En la sección del modelo software de los modelos 8088 y 8086 diseñados, podemos ver los diferentes tipos de direccionamientos que esta unidad soporta. También en la figura 3 de esta sección nos encontramos con un esquema descriptivo del funcionamiento general de la unidad de generación de direcciones que bien podríamos insertar en esta sección.

Signals Status

SIGNAL	SCOPE	DEFAULT	RESET	STATUS
Mem_IO_seg	OUT	"01"	"01"	CLKD
bus_a_adder	OUT	X"00000"		COMB
bus_b_adder	OUT	X"00000"		COMB
gt_ea_dfu	OUT	FALSE		COMB
gt_pa_dfu	OUT	FALSE		COMB
gt_pa_ifu	OUT	FALSE		COMB
op_adder	OUT	no_operation		COMB
rd_reg_gau	OUT	FALSE		COMB
rd_sreg_gau	OUT	FALSE		COMB
reg_sel_gau	OUT	"000"		COMB
sreg_sel_gau	OUT	"000"		COMB
EA	INT	X"00000"	X"00000"	REG
EA_generated	INT	FALSE		COMB
cont	INT	0	0	REG
ifu_dfun_gen	INT	FALSE	FALSE	REG
res_gau	INT			COMB
result_sincr	INT	X"00000"	X"00000"	REG
seg_default	INT	"01"		COMB
seg_used	INT	"01"		COMB



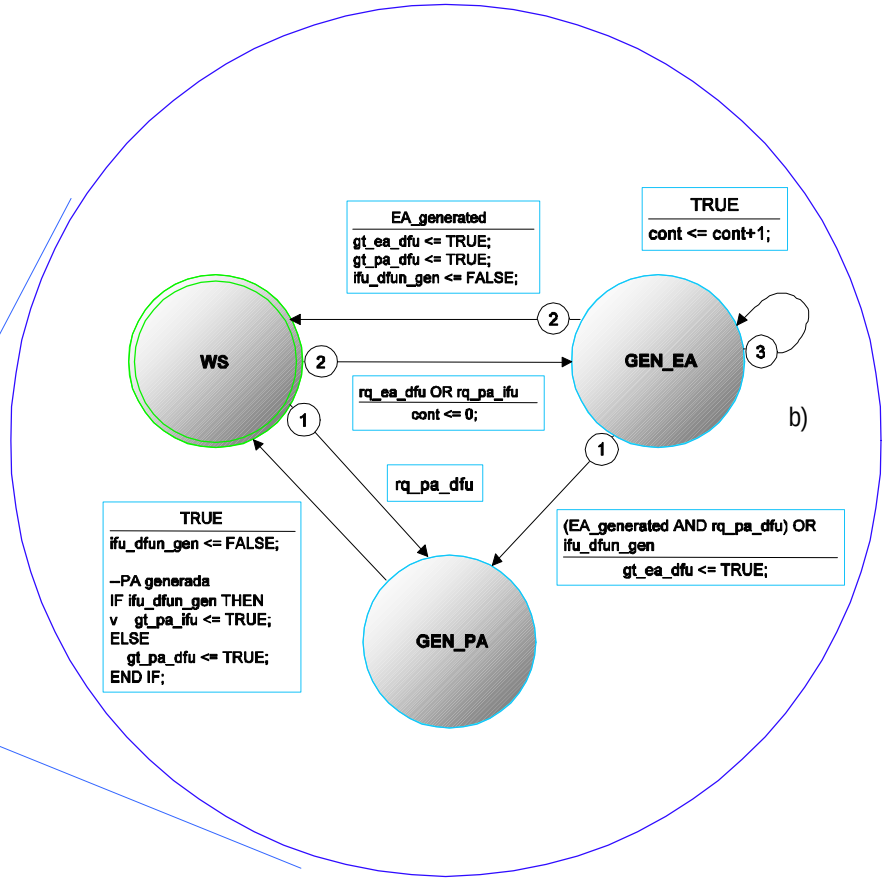


Fig 84.- a) Componente de la Unidad de GAU y
b) Grafo de Estados de la Unidad de Control de GAU.

30.1 ENTRADAS Y SALIDAS

En la siguiente tabla podemos ver las entradas y salidas de la unidad generadora de direcciones cuyo componente podemos ver en la figura 84 a.

SEÑAL	TIPO	FUNCIÓN
ph1, ph2	IN	Reloj interno
res	IN	Es la señal interna que resetea todos los componentes del modelo
gau_ctrl <-> rgu/sru		
reg_out sreg_out_gau	IN	Dato de sºalida de rgu/srg del registro seleccionado
reg_sel_gau sreg_sel_gau	OUT	Selector del registro de rgu/sru.
rd_reg_gau rd_sreg_gau	OUT	Indica a rgu/sreg que va a leer un registro.
gau_ctrl <- eu		
modo_gau r_m_gau	IN	Le indica a gau los parámetros mod y r_m.
prefix_asserted sreg_gau	IN	Le indican si existe un prefijo de segmento y en su caso cual es.
disp_gau_out	IN	Le indica a la gau que utilice este desplazamiento.
flush	IN	Le indica que aborte el actual.
gau_ctrl -> timing		
mem_io_seg	OUT	Le indica a la unidad de temporización del bus si se trata de un segmento de memoria o del segmento de I/O.
gau_ctrl <-> dfu		
rq_pa_dfu	IN	dfu le pide que genere una dirección física, PA.
gt_pa_dfu	OUT	Le indica a dfu que la dirección ha sido generada.
rq_ea_dfu	IN	dfu le pide que genere la dirección efectiva EA.
gt_ea_dfu	OUT	Le indica a dfu que EA ha sido generada.
HAC	IN	Le indica a gau que calcule la parte alta de la dirección.
gau_ctrl <-> ifu		
rq_pa_ifu	IN	La ifu le pide que calcule la dirección física de la siguiente instrucción.
gt_pa_ifu	OUT	Le indica a ifu que ha calculado la dirección física PA.
ifu_pointer	IN	Le pasa el puntero de instrucción para que calcule la PA.
result	OUT	Cuando la gau termina de calcular la dirección, contiene la PA.

Tab 52.- Entradas y Salidas de GAU_CTRL (archivo " up808x_biu_interf_gau_ctrl.vhd").

30.2 EXPLICACIÓN

Esta unidad es la encargada de calcular las direcciones físicas que vamos a utilizar en la transferencia de datos con la memoria o dispositivos de entrada salida. Estos cálculos se hacen a petición de las unidades que realizan transferencias con la memoria, es decir, a petición de IFU o DFU. Podemos considerar que la GAU responde a estas unidades de la siguiente manera.

30.2.1 COMUNICACIÓN CON LA UNIDAD IFU

Cuando la unidad que accede a la memoria en busca de instrucciones IFU necesita una dirección, la pide a la GAU activando la señal de petición (request) "rq_pa_ifu" que calcule la dirección física (PA) de la siguiente instrucción. Para ello la única información que necesita la unidad GAU es el puntero de la unidad IFU (ifu_pointer). Este puntero, sumado al puntero de instrucciones general IP, genera el puntero real que tenemos que utilizar para acceder a la siguiente instrucción de memoria a llevar a la cola de instrucciones. Es conveniente tener claro esto, el puntero de instrucciones IP apunta a la siguiente instrucción en ejecutarse de memoria e ifu_pointer contiene el desplazamiento a la última instrucción que ha "bajado" de la memoria. Más adelante en la sección de la unidad IFU se explica todo esto con más detalle.

La unidad GAU responde a la petición, sumando a CS desplazado 4 posiciones a la izquierda el puntero IP+ifu_pointer, como podemos ver en la siguiente figura:

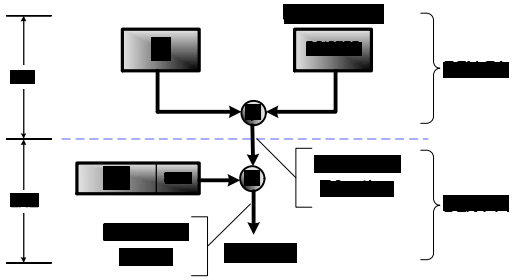


Fig 85.- Generación de PA_instr.

Este cálculo (que requiere 2 sumas), lo realiza la GAU en dos ciclos de reloj. La GAU responde a IFU activando la señal de petición concedida (grant) "gr_pa_ifu", momento en el que IFU recoge el dato de la señal "result" y lo salva en el registro interno "pa_instr". Ver la figura 86.

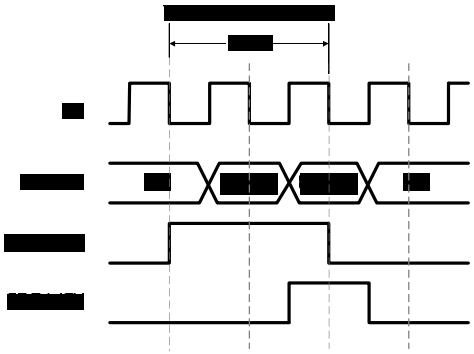


Fig 86.- Comunicación entre IFU-GAU

30.2.2 COMUNICACIÓN CON LA UNIDAD DFU

En este caso la comunicación es un poco más compleja, debido esencialmente a que para la generación de la dirección física que va a utilizar la unidad DFU, existen muchos modos de direccionamiento y no solo uno como ocurría en el caso anterior. Los distintos modos de direccionamiento que puede indicarle la unidad DFU a GAU, son los modos de direccionamiento de las instrucciones que podemos ver en el modelo software de los microprocesadores y apreciar de forma resumida en la figura 87.

Para que GAU puede seleccionar los registros a utilizar para generar la dirección (tipo de direccionamiento), deben estar programados los campos "modo" y "r_m".

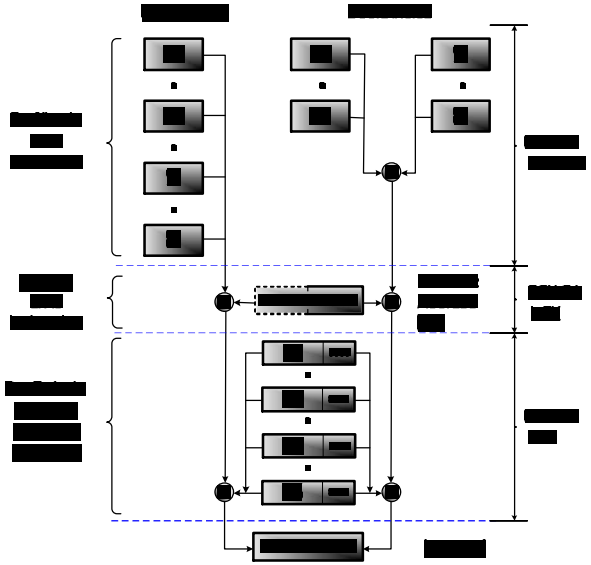


Fig 87.- Generación de PA_instr.

Si no está activa la señal de prefijo activado ("prefix_asserted"), la GAU se encarga de seleccionar el segmento por defecto asignado para el tipo de direccionamiento elegido, de lo contrario, la GAU utiliza el registro de segmento indicado en la señal seg_gau.

En las siguientes figura, podemos ver dos ejemplos de generación de "PA_data".

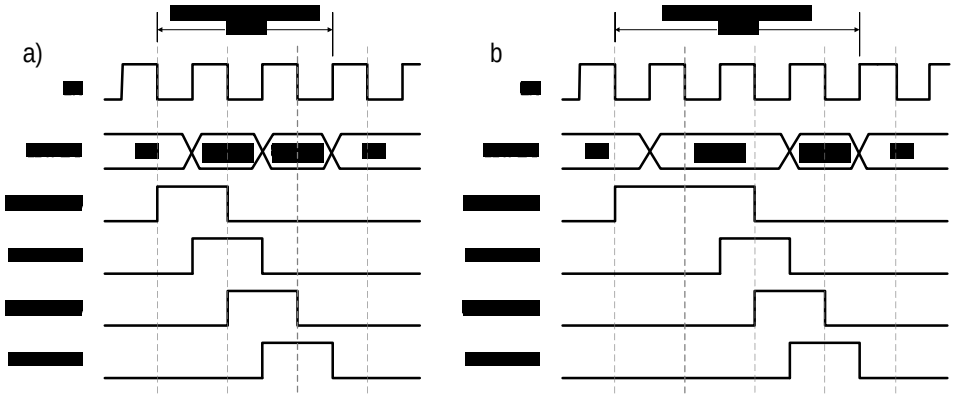


Fig 88.- a) Generación de PA_data Utilizando Direccionamientos de una Sola Suma.
b)Generación de PA_data Utilizando Direccionamientos de dos Sumas.

31. UIDAD DE ACCESO A DATOS, DFU (n3/r4)

n3/r4 Unidad de Acceso a Datos, Data Fetch Unit (DFU): Se encarga de las operaciones de transferencia a memoria o dispositivos de entrada salida, así como de la petición de los ciclos de reconocimiento de interrupción. Esta unidad hace de enlace entre la unidad de ejecución y el bus exterior. Cuando la EU necesita salvar/recoger un dato de la memoria o dispositivo de I/O, se lo comunica a esta unidad que se encarga de efectuar las operaciones necesarias para que la EU satisfaga sus necesidades. De esta forma la unidad de ejecución delega la comunicación con el exterior en esa unidad.

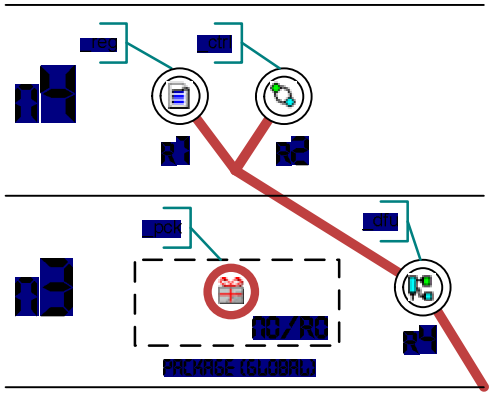
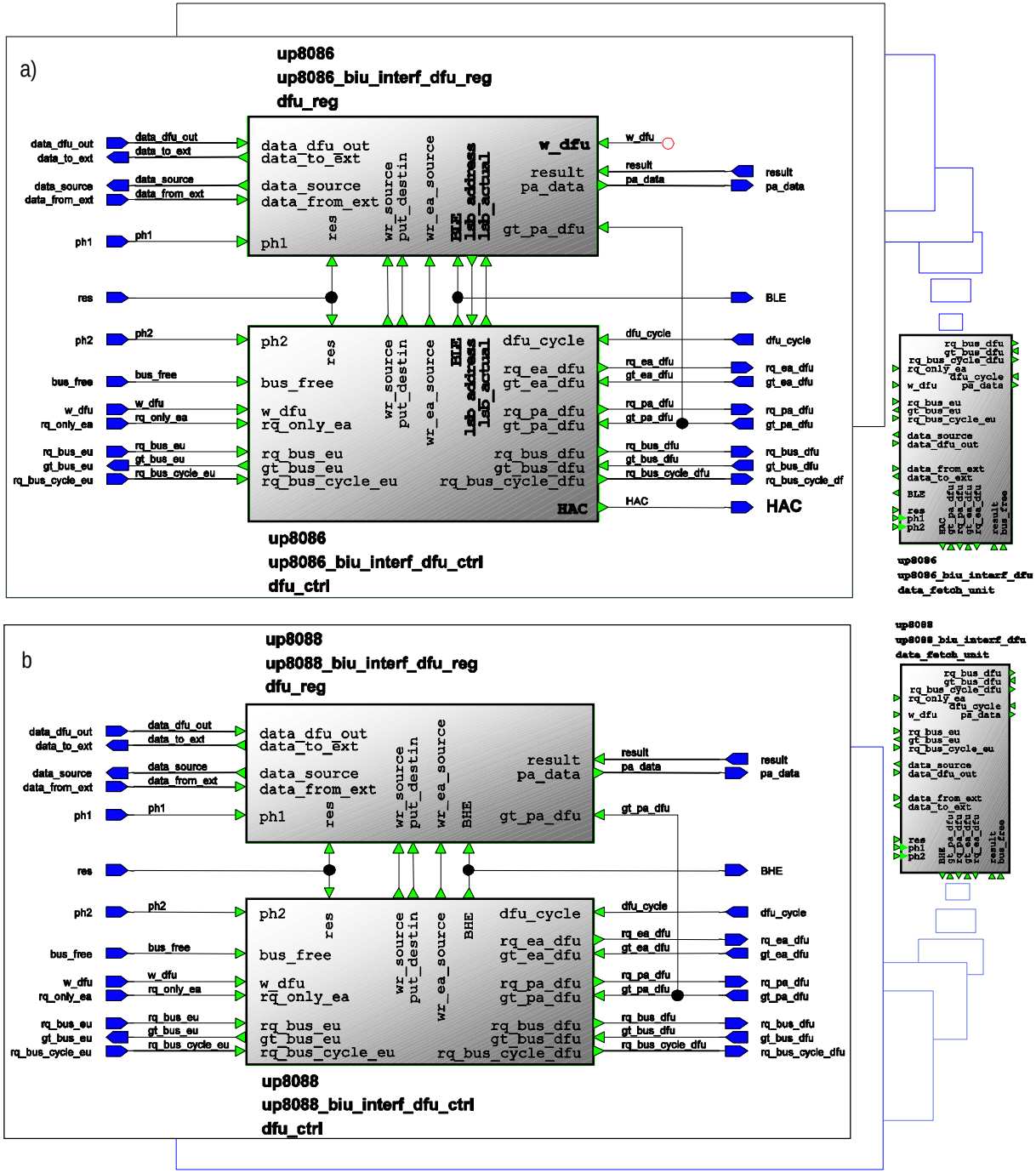


Fig 89.- Rama de la Unidad DFU.

Esta unida, junto con la cola de instrucciones QUEUE (n1/r5) e IFU (n3/r5) que estudiaremos más adelante, son las dos unidades con más cambios en función del modelo de microprocesador. Esto es lógico si tenemos en cuenta (como se explicó en la sección de diferencias entre el microprocesador 8088 y 8086) que el bus de datos externo es distinto en ambos micros (ver la figura 6), y por lo tanto esta unidad, que utiliza como medio este bus, tienen que seguir un lógica algo distinta. Sobretodo, las diferencias en la lógica, son provocadas por la necesidad de "alineamiento" de la memoria con el bus de datos. Esto se estudia un poco más adelante.

En la detalle del esquema jerárquico de la figura 89, podemos ver que en este nivel (n3/r4) se define un esquema y que la unidad DFU al completo es distinta en ambos modelos, por lo que tenemos una definición del esquema para el modelo 8088 (archivo " up8088_biu_interf_dfu.vhd") que podemos ver en la figura 90 b, y otra para el modelo 8086 (archivo " up8086_biu_interf_dfu.vhd") ver la figura 90 a, ambos localizados en la librería propia de cada modelo.

Fig 90.- a) Componente y Esquema Interno de la Unidad DFU del Modelo 8086
b) Componente y Esquema Interno de la Unidad DFU Modelo 8088



El la figura 90 a, podemos observar en forma resaltada las diferencias existentes entre los esquemas de ambos modelos. Estas diferencias son (el 8088 como referencia):

- BHE se convierte en BLE (Byte Low Enable), esta señal es un simple cambio de lógica de la señal BHE, BLE=NOT BHE, con este cambio, hacemos el código un poco más entendible. Hay que tener en cuenta que la señal externa es BHEn=NOT BHE, con lo que podemos relacionar directamente BLE con la función del pin BHEn (ojo, sólo en significado).
- lsb_actual, esta nueva señal, le indica a los registros el valor del bit bajo del bus de direcciones. Esta señal es la que habilita el banco bajo de la memoria y la necesitan los registros para saber por qué líneas le llega información.
- lsb_address. le indica a la parte de control el bit bajo de la última dirección calculada.
- w_dfu, le indica a los registros que va a recibir datos de 16 bits.(en 1 o dos ciclos de bus)
- HAC, High Address Calculation. Le indica a la unidad GAU (n3/r7) que calcule la dirección la parte alta del dato de 16 bits.

Internamente está compuesta por dos unidades, como podemos apreciar en las figuras de la página anerior.

- La unidad DFU_REG (n4/r1), contiene los registros de comunicación con el exterior, así como el registro de la dirección física PA (de 20 bits). En la siguiente sección explicaremos en detalle los regisros de esta unidad.
- La unidad de control DFU_CTRL que es el "corazón" de la unidad de acceso a datos. Esta unidad define el grafo de estados que implementa la lógica de control de la unidad DFU, comunicándose con el árbitro para pedir el control sobre el bus local y con la unidad GAU para pedirle la generación de la dirección física. Todo esto lo veremos en detalle más adelante.

31.1 ENTRADAS Y SALIDAS

SEÑAL	TIPO	FUNCIÓN
ph1,ph2	N	Reoj interno
res	I	E la señal interna que resetea todos los componentes del modelo
dfu <-> eu		
rq_only_ea	IN	UE indica que sólo quiere la dirección efectiva
w_dfu	IN	Le indica a DFU que tansfiera un byte de datos.
rq_bus_eu	IN	La UE pide una operación con el bus.
gt_bus_eu	OUT	DFU le indica que la operación con el bus ha sido realizada.
rq_bus_cycle_eu	IN	Le indica el tipo de ciclo de bus que quiere realizar la UE.
data_source	OUT	Contiene los datos recibidos desde el exterior
data_dfu_out	IN	Datos que tiene que sacar por el bus.
bus_free	IN	Le indica que el bus externo está libre
dfu <-> puertos		
data_from_ext	IN	Datos directamente desde el exterior.
data_to_ext	OUT	Datos a sacar al exterior
dfu -> bcg		
BLE	OUT	Habilitación de la parte baja del bus
BHE	OUT	Habilitación de la parte alta del bus
HAC	OUT	Cálculo de la dirección del segundo Byte,
dfu <-> gau		
rq_ea_dfu	OUT	Pide a la GAU que genere la dirección efectiva.
gt_ea_dfu	IN	Le comunica que la EA ha sido calculada
rq_pa_dfu	OUT	Pide la generación de la dirección física PA
gt_pa_dfu	IN	Le indica que la dirección física ha sido generada
result	IN	Contiene el resultado de las operaciones del sumador de la GAU.
dfu <-> arbiter		
rq_bus_dfu	OUT	Pide el control sobre el bus local
gt_bus_dfu	IN	Le indica que se le ha concedido el control sobre el bus local
rq_bus_cycle_dfu	OUT	Le indica al árbitro el tipo de bus que tiene que realizar la unidad TIMING.
dfu_cycle	IN	Le indica que el ciclo actual es de la unidad DFU
pa_data	OUT	Dirección física que hay que utilizar en el ciclo de bus.

Tab 52.- Entradas y Salidas de DFU (archivos " (up8088/8086)_biu_interf_dful.vhd").

32. REGISTROS DE DFU, DFU_REG (n4/r1)

n4/r1 Registros de DFU (dfu_reg): Son los registros que utiliza DFU para comunicarse con el exterior y pasar información a la EU. También contiene el registro de 20 bits que memoriza la dirección física con la que vamos a realizar la transferencia PA_idata.

Como podemos ver en la figura 89, este componente está definido en las librerías particulares de los dos modelos. Así tenemos el fichero " up8088_biu_interf_dfu_reg.vhd" (figura 91 a) que define los registros de la unidad DFU del modelo 8088 y el fichero " up8086_biu_interf_dfu_reg.vhd" (figura 91 b) que define el comportamiento de estos registros en el modelo 8086. En la figura 91 b podemos ver resaltadas las diferencias entre los dos componentes.

Como podemos observar en los códigos del proceso "reg_dfu" de ambos ficheros, el registro de las señales "data_source" y "data_to_ext" es diferente en ambos. En el modelo 8088 el bus externo es de 8 bits por lo que tenemos que descomponer el dato que tenemos que mandar ("data_to_ext" de 16 bits) en dos bytes, sacamos el byte_alto cuando está activada la señal BHE y el byte bajo cuando no está activada.

En el 8086, el bus de datos externo es de 16 bits, por lo que no es necesario descomponer (demultiplexar) el dato a enviar.

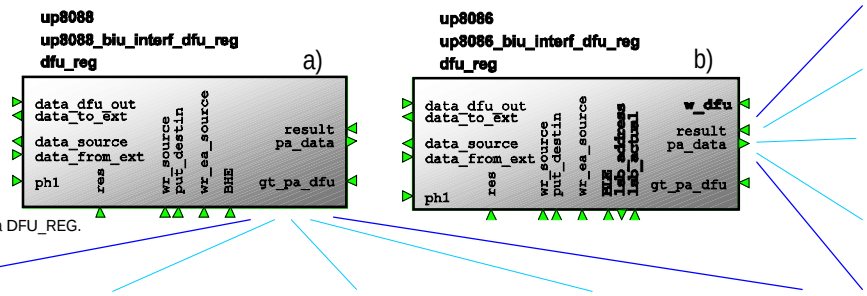


Fig 91.- Carcasas para DFU_REG.
a) Para el 8088
b)Para el 8086

```
    ------------------------------------------------------------------------------
    reg_dfu : PROCESS(ph1,res)
    ------------------------------------------------------------------------------
    BEGIN
        IF (res = true) THEN
            -- Reset Values
            data_source_cld <= X"0000";
            data_to_ext_cld <= X"00";
        ELSIF (ph1'EVENT AND ph1 = '1') THEN
            -- Default Assignment To Internals
            data_source_cld <= data_source_cld;
            data_to_ext_cld <= data_to_ext_cld;
            --wr_source, wr_ea_source y put_destin no cionciden
            --nunca en el tiempo, por eso no establezco prioridades

            --Escribir ea en dato fuente hacia exe
            IF wr_ea_source THEN
                data_source_cld <= result(15 DOWNTO 0);
            END IF;
            --Escribir (componer) dato desde memoria en un reg de 16 bits
            IF wr_source AND BHE='1' THEN
                data_source_cld(15 DOWNTO 0) <= SXT(data_from_ext,16);
            ELSIF wr_source AND BHE='0' THEN
                data_source_cld(15 DOWNTO 8) <= data_from_ext;
            END IF;

            --Escribir (descomponer) dato desde exe (16 bits) en registros
            --de 8 bits, para el 8088; para 8086 data_to_ext_cld es de 16 bits
            IF put_destin AND BHE='1' THEN
                data_to_ext_cld <= data_dfu_out(7 DOWNTO 0);
            ELSIF put_destin AND BHE='0' THEN
                data_to_ext_cld <= data_dfu_out(15 DOWNTO 8);
            END IF;

        END IF;
    END PROCESS reg_dfu;
```

```
    compost <= w_dfu&lsb_actual&ble;

    ------------------------------------------------------------------------------
    reg_dfu : PROCESS(ph1,res)
    ------------------------------------------------------------------------------
    BEGIN
        IF (res = true) THEN
            -- Reset Values
            data_source_cld <= X"0000";
            data_to_ext_cld <= X"0000";
        ELSIF (ph1'EVENT AND ph1 = '1') THEN
            -- Default Assignment To Internals
            data_source_cld <= data_source_cld;
            data_to_ext_cld <= data_to_ext_cld;
            --wr_source, wr_ea_source y put_destin no cionciden
            --nunca en el tiempo, por eso no establezco prioridades

            --Escribir ea en dato fuente hacia exe
            IF wr_ea_source THEN
                data_source_cld <= result(15 DOWNTO 0);
            END IF;
            --Escribir (componer) dato desde memoria en un reg de 16 bits
            IF wr_source THEN
                CASE compost IS
                WHEN "001" =>
                --Lectura de la parte baja (8 bits)
                --con extensión de signo para extender desplazamientos cortos
                --o datos enteros de byte a word
                    data_source_cld(15 DOWNTO 0) <= sxt(data_from_ext(7 DOWNTO 0),16);
                WHEN "010"|"110" =>
                --Lectura de la parte alta (8 bits), Desalineado
                --con extensión de signo para extender desplazamientos cortos
                --o datos enteros de byte a word
                    data_source_cld(15 DOWNTO 0) <= sxt(data_from_ext(15 DOWNTO 8),16);
                WHEN "100" =>
                --Lectura de (16 bits), Alineado
                    data_source_cld(15 DOWNTO 0) <= data_from_ext(15 DOWNTO 0);
                WHEN "111" =>
                --Lectura de (8 bits), Desalineado
                    data_source_cld(15 DOWNTO 8) <= data_from_ext(7 DOWNTO 0);
                WHEN OTHERS => NULL;
                END CASE;
            END IF;

            --Ahora no es necesario descomposer el dato
            IF put_destin THEN
                data_to_ext_cld <= data_dfu_out;
            END IF;

        END IF;
    END PROCESS reg_dfu;
```

32.1 REGISTROSDE COMUNICACIÓN INTERNA

El registro de la dirección física para la transferencia de datos, está en un proceso independiente y es común para los dos modelos. El proceso "pa_data_reg" que podemos ver a continuación, modela un registro clásico de reset asíncrono activado en fase ph1. La señal que enclava el dato es la señal "gt_pa_dfu" (generada por GAU, n3/r7) que indica que en el bus result está la dirección física.

```
-----------------------------------------------------
    pa_data_reg : PROCESS(
        ph1,
        res
    )
-----------------------------------------------------
    BEGIN
        IF (res = true) THEN
            -- Reset Values
            pa_data_cld <= X"00000";
        ELSIF (ph1'EVENT AND ph1 = '1') THEN
            -- Default Assignment To Internals
            pa_data_cld <= pa_data_cld;

            IF gt_pa_dfu THEN
                pa_data_cld <= result;
            END IF;

        END IF;
    END PROCESS pa_data_reg;
```

Existe otra acción común a ambos modelos y es el registro de la dirección efectiva EA, en la señal hacia la unidad de ejecución ("data_source") cuando aparece la señal de control "wr_ea_source" generada por DFU_CTRL (n4/r2). La instrucción LEA utiliza la dirección efectiva como operando, la BIU se comunica con la EU a través de "data_source" donde le pasa el operando.

A continuación, vamos a ver el comportamiento de los registros de comunicación entre el bus externo y la EU en los diferentes modelos.

32.1.1 REGISTROS DE COMUNICACIÓN EN EL 8088

El 8088 tiene un bus de datos externo de 8 bits, sin embargo, la estructura interna es de 16 bits de manera que para realizar transferencias de 16 bits, necesitamos multiplexar el dato interno hacia el bus en "bloques" de 8 bits. De la misma manera, debemos demultiplexar el dato de 8 bits externo para componer el registro interno de 16 bits que se comunica con la EU (registro "data_source"). En la siguiente figura (figura 92 a), podemos ver esta conversión entre el bus externo de 8 bits y la lógica de registros interna de 16 bits.

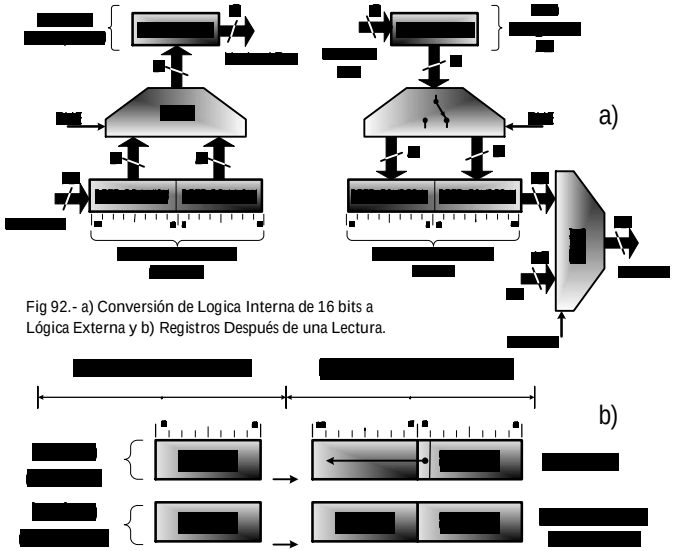


Fig 92.- a) Conversión de Logica Interna de 16 bits a Lógica Externa y b) Registros Después de una Lectura.

En estos casos es la señal BHE la que determina que parte del registro de 16 bits va a ser escrita o leída. En la Figura 92 b, podemos ver la lógica seguida.

Para la lectura de 16 bits, se ejecutan dos ciclos consecutivos. En la primera lectura, se extiende siempre el signo del bit alto del dato recogido para dar soporte a desplazamientos cortos y a números enteros en bytes.

32.1.2 REGISTROS DE COMUNICACIÓN EN EL 8086

El 8086, tiene un bus de datos de 16 bits lo que permite una comunicación directa entre la estructura interna de 16 bits y el bus local en operaciones de escritura. La situación cambia en la lectura, pese a ser ambos sistemas de 16 bits, puede haber un "desalineamiento" en una transferencia entre estas debida esencialmente a la localización de datos de 16 bits en posiciones de memoria impares, todo esto se explica con más detalle en la siguiente sección.

Para solventar este problema, los micros "mapean" la memoria en dos bancos de 8 bits (bus de datos de 8 bits), uno de ellos contiene el byte más significativo MSB, y el otro el menos significativo LSB. El banco MSB es controlado por la señal BHEn (BLE internamente) y el banco LSB por la señal A0 (control en lógica inversa, activo con '0'). En la gráfica 6 podemos ver la organización de la memoria en el 8086.

Las señales que "alinean" la memoria con la estructura interna son. ble, lsb_actual y w_dfu como se puede ver en el código de la página anterior. La lógica de alineamiento de los registros es la siguiente:

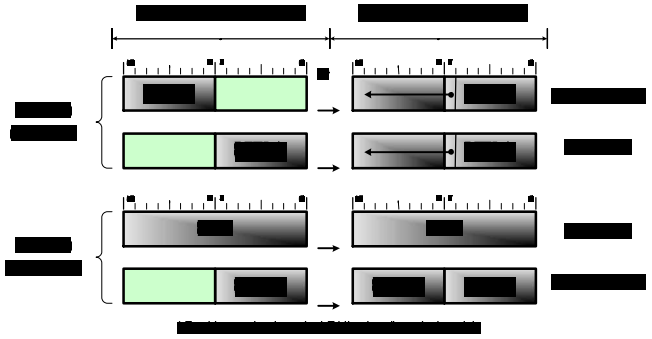


Fig 93.- Alineamiento en los Microprocesadores 8086.

Cuando una accedemos a la lectura de un dato 16 bits en memoria que no está alineado, el microprocesador precisa dos ciclos de lectura para leer el dato en cuestión. En el primer ciclo, el sistema de memoria, coloca en la parte alta del bus de datos el byte menos significativo del dato; nosotros debemos colocar este byte en la parte baja del registro de comunicación interna e ignorar los posibles datos que pudieran aparecer en el byte menos significativo del bus de datos. En un segundo ciclo, el sistema de memoria coloca el byte alto del dato en la parte baja del bus de datos; de nuevo, debemos alinear este dato componiéndolo correctamente en el registro de comunicación interna ("data_source"), el lugar correcto del byte válido en el bus de datos es como byte más significativo del registro de comunicación interna.

33. GRAFO DE CONTROL DE DFU, DFU_CTRL (n4/r2)

n4/r2 Grafo de Control de DFU. Este es el grafo de lleva a cabo los ciclos de transferencia de datos entre la memoria y dispositivos de entrada/salida y la unidad de ejecución.

La UE indica que quiere ejecutar un ciclo de bus a esta unidad activando la señal "rq_bus_eu" e indicándole el ciclo de bus que tiene que ejecutar en la señal "rq_bus_cycle". Para manejar de una forma más fácil los ciclos de bus, se creó en el package del sistema (n0/r0) un subtipo de vector Std_Logic cuya codificación se realiza a base de constantes. Aquí podemos ver el código: .....➤

```
--Codificación de Bus_Cycle_Type como Std_Logic_Vector
SUBTYPE Bus_Cycle_Typ IS Std_Logic_Vector(2 DOWNTO 0);
-- Codificación Hardware de Bus_Cycle_Typ        --Tipo de ciclo de bus:
CONSTANT interrupt_ack: Bus_Cycle_Typ := "000"; --Ciclo de reconocimineto de Interrupción
CONSTANT read_io      : Bus_Cycle_Typ := "001"; --Ciclo de lectura de un puerto externo
CONSTANT write_io     : Bus_Cycle_Typ := "010"; --Ciclo de escritura de un puerto
CONSTANT halt         : Bus_Cycle_Typ := "011"; --Ciclo de parada
CONSTANT instr_fetch  : Bus_Cycle_Typ := "100"; --Ciclo de ir en busca de intrucciones
CONSTANT read_mem     : Bus_Cycle_Typ := "101"; --Ciclo de lectura en memoria
CONSTANT write_mem    : Bus_Cycle_Typ := "110"; --Ciclo de escritura en memoria
CONSTANT passive      : Bus_Cycle_Typ := "111"; --Ciclo de bus pasivo
```

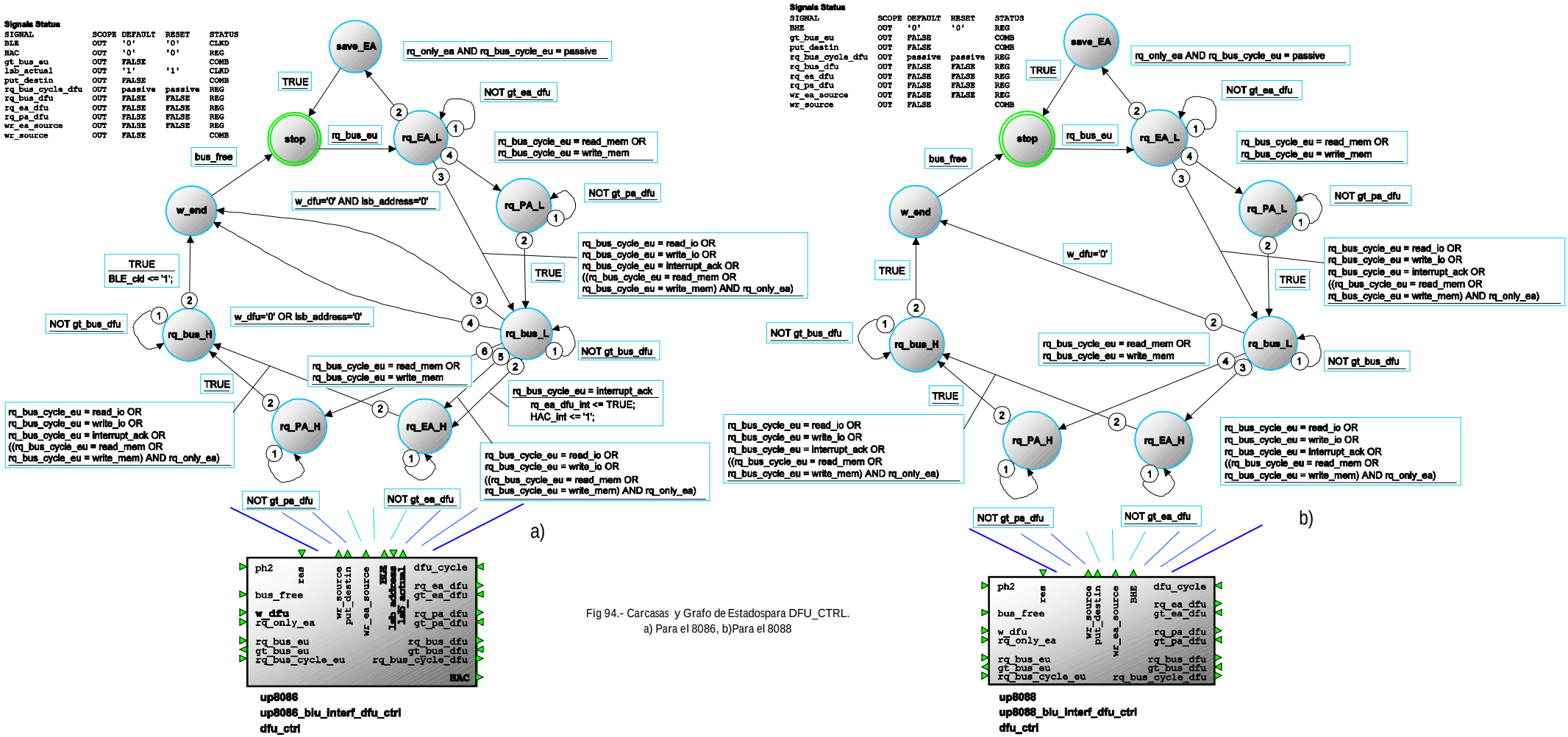


Fig 94.- Carcasas y Grafo de Estadospara DFU_CTRL.
a) Para el 8086, b)Para el 8088

33.1 ENTRADAS Y SALIDAS

En la siguiente tabla, podemos ver las señales de entrada y salida del componente de la figura 94. Las señales que son diferentes en ambos modelos, aparecen resaltadas en la figura 94 a.

SEÑAL	TIPO	FUNCIÓN
ph1,ph2	N	Reoj interno
res	IN	E la señal interna que resetea todos los componentes del modelo
dfu_ctrl <-> eu		
rq_only_ea	IN	UE indica que sólo quiere la dirección efectiva
w_dfu	IN	Le indica a DFU que tansfiera un byte de datos.
rq_bus_eu	IN	La UE pide una operación con el bus.
gt_bus_eu	OUT	DFU le indica que la operación con el bus ha sido realizada.
rq_bus_cycle_eu	IN	Le indica el tipo de ciclo de bus que quiere realizar la UE.
bus_free	IN	Le indica que el bus externo está libre
dfu_ctrl -> bcg		
BLE	OUT	Habilitación de la parte baja del bus
BHE	OUT	Habilitación de la parte alta del bus
HAC	OUT	Cálculo de la dirección del segundo Byte,
dfu_ctrl <-> gau		
rq_ea_dfu	OUT	Pide a la GAU que genere la dirección efectiva.
gt_ea_dfu	IN	Le comunica que la EA ha sido calculada
rq_pa_dfu	OUT	Pide la generación de la dirección física PA
gt_pa_dfu	IN	Le indica que la dirección física ha sido generada
dfu <-> arbiter		
rq_bus_dfu	OUT	Pide el control sobre el bus local
gt_bus_dfu	IN	Le indica que se le ha concedido el control sobre el bus local
rq_bus_cycle_dfu	OUT	Le indica al árbitro el tipo de bus que tiene que realizar la unidad TIMING.
dfu_cycle	IN	Le indica que el ciclo actual es de la unidad DFU
dfu_ctrl <-> dfu_reg		
wr_source	OUT	Le indica que escriba el dato desde e exterior en el registro data_source.
put_destin	OUT	Saca por la señal "data_to ext" el dato que aparecerá en el bus de datos.
wr_ea_source	OUT	Escribe en data_source result (que contiene la dirección efectiva).
lsb_actual	OUT	Contiene el bit A0 de la dirección que hay actualmente en el bus de direcciones.
lsb_address	IN	Le pasa el bit A0 de la dirección calculada.

Tab 54.- Entradas y salidas de DFU_CTRL.

33.2 FUNCIONAMIENTO

Para entender el funcionamiento de esta unidad, es preciso comprender perfectamente cómo se comporta el bus de datos externo del microprocesador. Que adelanto, no es sencillo. Además de toda la lógica desarrollada por Intel ® para dar soporte a sus famosos sistemas MULTIBUS, estos micros soportan sistemas de memoria de 8 o 16 bits (en el 8086) y de 8 bits (en el 8088) y transferencias de 16 bits (en ambos modelos). Para implementar esta característica de forma transparente para el usuario, el sistema de memoria de 16 bits se divide en dos bancos. Estas y otras características de bus, hacen a un sistema muy potente, y nuestros modelos (como los originales) las soportan.

A continuación explicaré en detalle el funcionamiento del bus de datos local utilizado para comunicar la UE con la memoria o periféricos, esta comunicación se realiza siempre a través de la unidad DFU.

33.2.1 BUS DE DATOS

Podemos ver la memoria o los dispositivos I/O como una secuencia de bytes. La memoria tiene una profundidad de 1 Mbyte, mientras que el espacio para los puertos de entrada-salida es de 64 Kbytes. Dos posiciones consecutivas de memoria, forman una palabra.

BUS DE 16 BITS

El 8086 tiene un bus de datos de 16, el bus de datos de la memoria está físicamente dividido en dos bancos de 512K cada uno (ver la figura 95 a)

Un banco para la parte baja del bus de datos qu contiene todas las direcciones pares (A0='0') y otro banco para la parte alta del bus de direcciones que contiene las direcciones impares (A0='1'). Las líneas de direcciones A19:A1 seleccionan una de las posibles 512K posiciones de memoria y las líneas (BHEn y A0) determinan que banco está activo (uno de ellos o ambos)

Las transferencias de Byte a direcciones pares, transmiten la información al banco bajo (ver la figura 96). A0='0" (en direcciones pares) activa el banco LSB mientras que BHEn='1' deshabilita el banco MSB. Cualquier dato en la parte alta del bus de direcciones en una lectura de este tipo es ignorado, como se comentó en la sección anterior. En las operaciones de escritura, BHEn='0',impide que escribamos un dato no válido en el banco MSB.

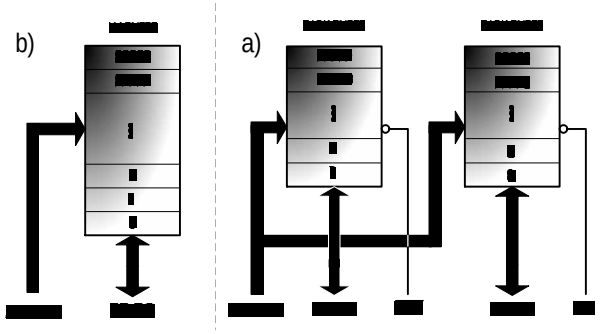


Fig 95.- a) Bancos de Memoria de un Sistema de 16 bits y b) para un Sistema de 8 bits.

Transferencia de Byte a direcciones impares, estas operaciones, transfieren el dato en la parte alta del bus de direcciones (ver la figura 97). La activación de BHEn ('0'), activa el banco MSB, mientras que A0='1' deshabilita el banco LSB. Los datos en la parte baja son ignorados en ciclos de lectura y la activación de A0='1' (en todas las direcciones impares A0='1') previene la escritura de datos en el banco LSB.

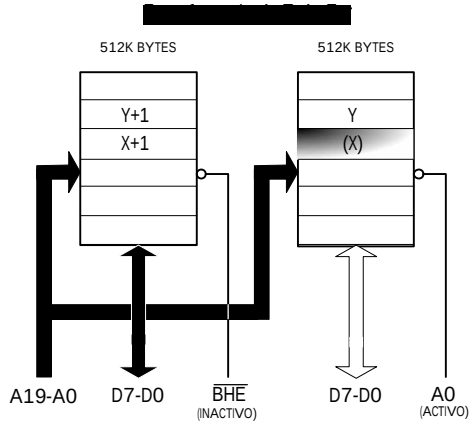


Fig 96.- Transferencia de un Byte a una Dirección Par en un Sistema de 16 Bits.

Para acceder a direcciones pares que contienen datos de 16 bits (dos bytes consecutivos con el byte menos significativo en la dirección par, dato alineado),

la información es transferida en ambas partes del bus de datos (ver la siguiente figura). A19:1 seleccionan el byte apropiado dentro de cada banco. A0 y BHEn, son ambas '0' habilitando de esta manera los dos bancos de memoria. Ver la Figura 98

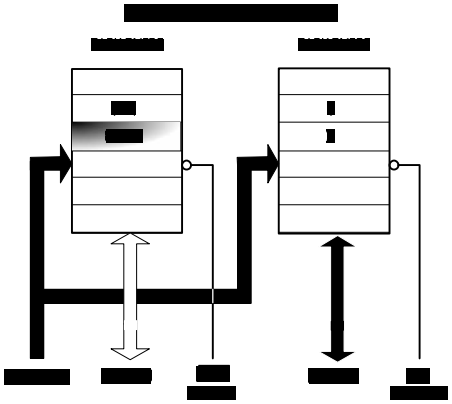


Fig 97.- Transferencia de un Byte a una Dirección Impar en un Sistema de 16 Bits.

Los datos de 16 bits en direcciones impares (desalineados), requieren dos ciclos de bus consecutivos como se puede ver en la figura 99. El primer ciclo de bus transfiere en la parte alta del bus de datos, mientras que el segundo ciclo de bus lo hace el la parte baja del bus de datos. La unidad DFU se encarga de "lanzar" estos dos ciclos de bus consecutivos y de alinear los datos de bus con los registros de la estructura interna.

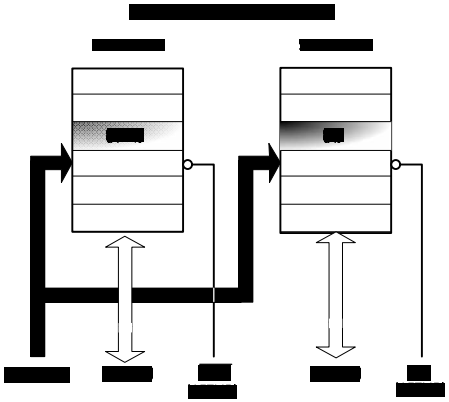


Fig 98.- Transferencia de una Palabra a una Dirección Par en un Sistema de 16 Bits.

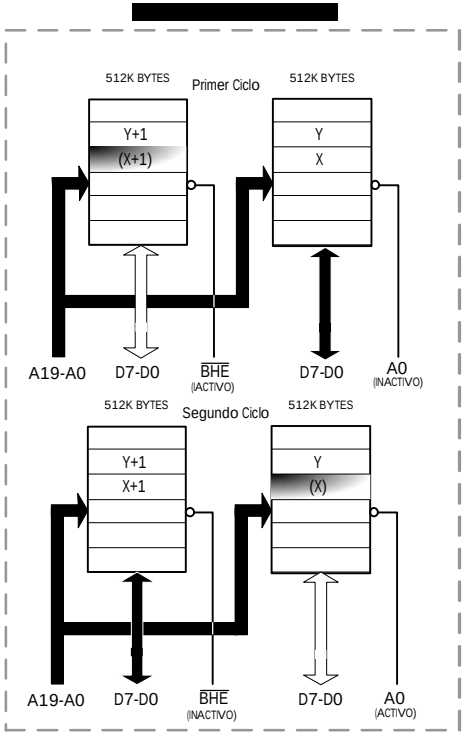


Fig 99.- Transferencia de una Palabra a una Dirección Impar en un Sistema de 16 Bits.

Durante una operación de lectura de un byte, la unidad BCG, pone en estado de alta impedancia el bus de datos, pero la transferencia sólo se realiza o bien el el byte bajo o bien en el alto. Esta acción simplifica los requerimientos de decodificación para los dispositivos de "solo lectura" (ROM, PROM, Flash). Durante el ciclo de lectura, un dispositivo externo puede poner datos en ambas partes de bus de datos, pero la unidad DFU sólo accede al byte correcto del bus de datos. Durante operaciones de escritura de un byte, la unidad DFU utiliza el bus de datos completo pero los dispositivos externos ignoran el byte del bus de datos cuyo banco no esté activo.

BUS DE 8 BITS

El espacio de memoria de un sistema con un bus de datos de 8 bits, está formado por un banco de 1 Mbyte (ver la figura 95). Las líneas de direcciones A19:0 seleccionan un byte específico dentro del espacio de memoria. A diferencia de las transferencias en el 8086 (bus de datos de 16 bits), en el 8088 se realizan siempre a posiciones pares, ya sean transferencia de bytes o de palabras utilizando los 8 bits del bus de datos.

Las transferencias de byte a direcciones pares o impares se realizan en in ciclo de bus. Las transferencias de palabras a direcciones pares o impares se siempre en dos ciclos de bus. Esta simplicidad en las transferencias y la ausencia de datos desalineados en la memoria, hace que la lógica de la unidad DFU para el 8088 sea más simple que la del 8086. La unidad DFU para el 8088, asigna automáticamente todas las transferencias de 16 secuencias de dos ciclos de bus. Para transferencias de 16 bits, la dirección del dato es en la que se encuentra el byte bajo (la primera), la dirección del byte alto es la del byte bajo más uno.

En la siguiente figura, podemos ver una transferencia de 16 bits en un sistema con bus de 8 bits.

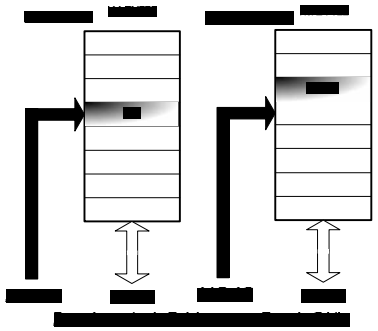


Fig 100.- Transferencia de una Palabra en un Sistema de 8 Bits.

33.2.2 INTERFACES CON LA MEMORIA E I/O

Esto microprocesadores pueden tener buses de datos de 8 o 16 bits para comunicarse con la memoria o dispositivos de entrada y salida. La memoria puede intercambiar información con los micros en los ciclos de lectura de memoria, escritura de memoria y ciclos de búsqueda de instrucciones. Los periféricos de entrada u y salida intercambian información con el microprocesador durante los ciclos de lectura de lectura de Memoria, escritura de memoria (si los dispositivos están mapeados) e I/O,

escritura de I/O y ciclos de reconocimiento de interrupción de forma natural. Los dispositivos de I/O Mapeados, permiten utilizar toda la potencia del juego de instrucciones para intercambiar información con el periférico en cuestión. Los ciclos de bus de lectura/escritura de I/O, utilizan un espacio de memoria separado al que solamente se puede acceder con las instrucciones IN u OUT. La información que le pasamos debe estar en el registro AX. A los primeros 256 bytes del espacio reservado a I/O, podemos acceder directamente con la instrucciones anteriores, pero para acceder a los 64Kbytes de espacio reservado para I/O, sólo podemos hacerlo indirectamente a través del registro DX. Las instrucciones de entrada y salida hacen que el nibble alto del bus de direcciones sea cero. Los ciclos de reconocimiento de interrupción, se comunican con un dispositivo de entrada salida que le indica el vector de interrupción.

33.2.3 LÓGICA DEL GRAFO DE ESTADOS

Se puede ver en los grafos de la figura 94 que las unidades de ambos modelos siguen una lógica parecida, simplemente existe algún cambio en las transiciones. Los estados son:, seguir en la figura 101:

- Stop, estado inicial (e0). En este estado permanecemos a la espera de una petición por parte de la EU ("rq_bus_eu", transición 1), momento en el que pasamos al estado "rq_EA_L".
- rq_EA_L pide la generación de EA baja,(e1). En este estado, la unidad de control de DFU le indica a GAU que calcule la dirección efectiva de menor peso. Para ello activa la señal "rq_es_dfu" y pone a nivel bajo la señal BHE. Permanecemos en este estado (t1 activa) hasta que la GAU nos indica que ha calculado la dirección efectiva. En ese momento, dependiendo del tipo de ciclo de bus que la UE requiere, saltamos al estado "rq_PA_L" (si EU requiere un ciclo de lectura o escritura de memoria, t4), al estado "rq_bus_L" (si EU requiere un ciclo de lectura/escritura de un periférico, un ciclo de lectura/escritura de memoria en el segmento 0, es decir que no hace falta calcular PA, utilizado para el acceso a los vectores de interrupción, o un ciclo de reconocimiento de interrupción, t3). En el caso de que EU requiera sólo el cálculo de EA, y no un ciclo debus (t2), saltamos al estado "save_EA.
- save_EA, salva el valor de EA en el registro "data_source" (e2). Cuando la EU ejecuta la instrucción LEA, pide a la GAU (a través de DFU) que le mande la EA que utilizaría para calculare la PA con el tipo de direccionamiento que le indica la EU en "modo" y "r_m". Esto es muy útil para cargar registros con el puntero a un dato específico en tiempo de ejecución, que es el único momento en el que sabemos dónde está posicionado un dato (ver la teoría de la relocalización dinámica del código en el capítulo del modelo software). De este estado saltamos a e0.
- rq_PA_L, requiere el cáculo de la parte baja de PA (e3). Manteniendo la señal BHE a nivel bajo y activando la señal "rq_pa_dfu", le pide a GAU que calcule la dirección física del primer byte. Permanecemos en este estado hasta que PA_L ha sido calculada (t1).
- rq_bus_L, requiere el bus para la transferencia del byte bajo, (e4). Una vez calculada PA, pedimos el control sobre el bus local activando la señal "rq_bus_dfu". Cuando el árbitro de bus (n3/r3) estime oportuno, concederá el bus a DFU y mandará a BCG (n3/r9) la dirección física ("PA_idata) y el tipo de ciclo de bus ("rq_bus_cycle_dfu") que tiene que lanzar. Permanecemos en este estado hasta que el bus nos es concedido (t1). Con el bus concedido, si la transferencia era de un soó byte (w_dfu='0'), saltamos al estado "w_end" en espera de que finalice el ciclo de bus. Si w_dfu='1', hay que realizar otro ciclo de bus (siempre en el 8088 y si el dato está desalineado, en el 8086) por lo que saltamos a "rq_EA_H" si necesitamos calcular EA_H (transición 3) o a "rq_PA_H" si la transferencia es a memoria (t4). Observar como de no requerir el cálculo de EA específicamente, no saltamos al estado e5 y por lo tanto ahorramos un ciclo de reloj.
- rq_EA_H, requiere el cálculo de la dirección del byte alto de EA, (e5). A partir de este estado, BHE='1' con lo que indicamos a GAU que utilice el mismo modo de direccionamiento anterior, pero que le sume 1. La petición como en el estado e1 la hacemos a través de la señal "rq_ea_dfu" y GAU nos responde en "gt_ea_dfu" cuando la ha calculado, momento en el que saltamos a "rq_bus_H".
- rq_PA_H, requiere el cálculo de la dirección para acceder al byte más significativo del dato, (e6).Manteniendo activa la señal BHE y activando "rq_pa_dfu", la GAU suma a la dirección calculada anteriormente 1, de manera que accedemos al byte de mayor peso. Permanecemos en este estado (t1) hacta que GAU nos indica que ha calculado PA_H. momento en el que saltamos a "rq_bus_H".
- rq_bus_H, requiere el bus para transferir el byte más significativo (e7). Este estado, hace lo mismo que "rq_bus_L". Solo que los datos que mandamos a la BCG (n3/r9) esta vez son los del byte más significativo. Cuando el bus es concedido (t2), saltamos a w_end.
- w_end, espera a que termine el ciclo de bus, (e8). Esperamos a que el bus esté libre "bus_free" (t1) momento en el que ha terminado la transferencia, hecho que india a la EU activando la señal "gt_bus_eu".

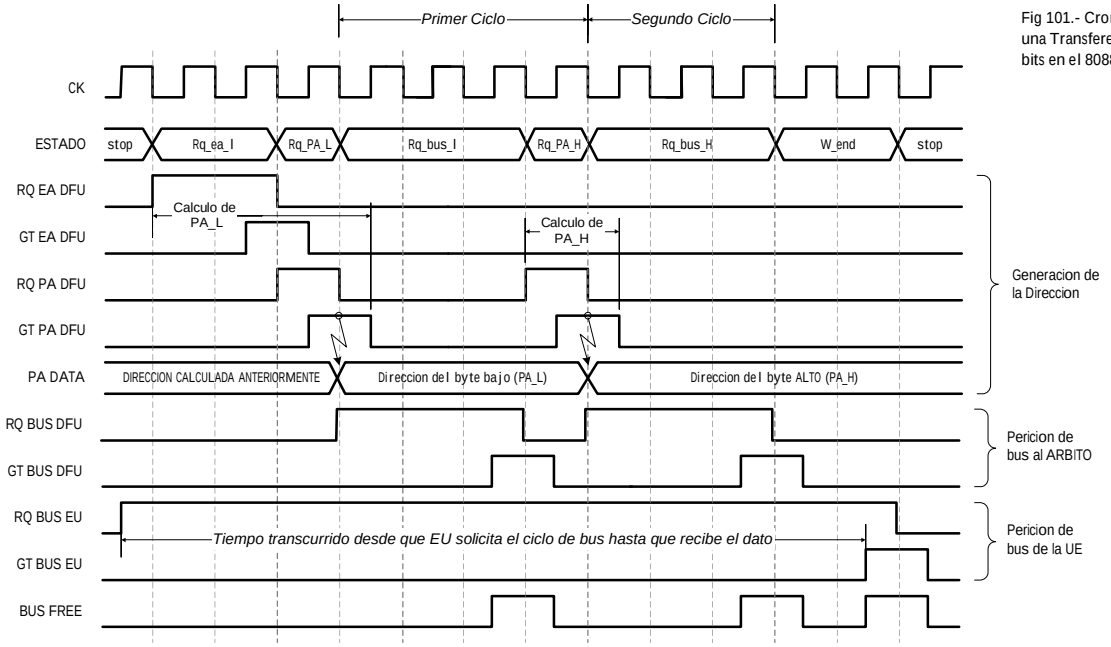


Fig 101.- Cronograma de una Transferencia de 16 bits en el 8088.

34. UNIDAD DE ACCESO A INSTRUCCIONES, IFU (n3/r5)

La última rama que nos queda por "podar", es la rama de la Intruction Fetch Unit, IFU, como podemos ver en el siguiente detalle del esquema jerárquico de la figura 102.

n3/r5 Unidad que va en Búsqueda de Instrucciones, Instruction Fetch Unit (IFU): Encargada de realizar los ciclos que llenan la cola de instrucciones. Siempre que no exista una petición de ciclo de bus específico por parte de la UE, IFU genera la petición al árbitro hasta que llena la cola de instrucciones. En estas condiciones, UE y BIU trabajan en paralelo.

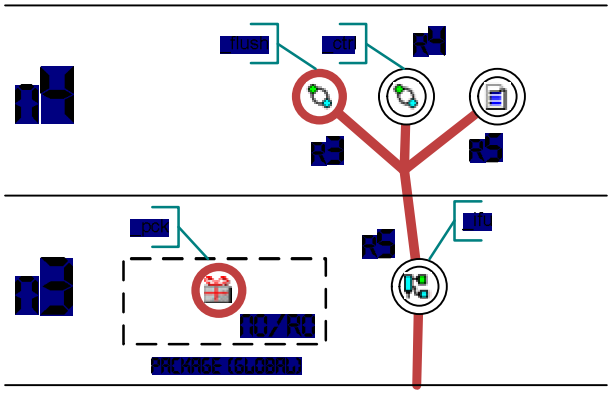


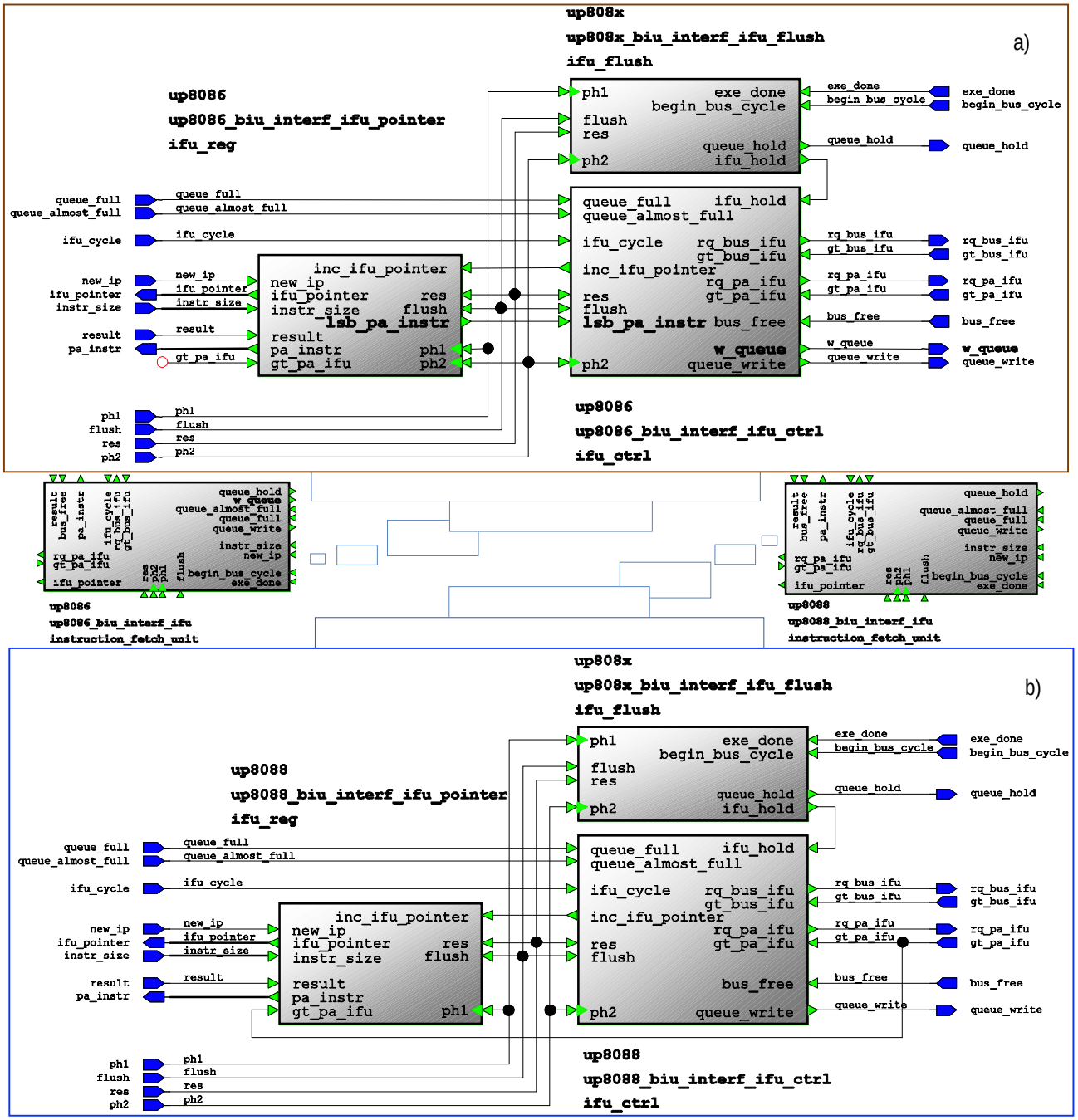
Fig 102- Podado de la Rama de la Unidad IFU.

Como se puede ver en los esquemas de la figura 103, este nivel (n3/r5) contiene un esquema con tres unidades interconectadas entre si. Una de ellas, la unidad de sincronización de flush (n4/r3), es común para ambos modelos por lo que podemos encontrarla en la librería común up808x (archivo " up808x_biu_interf_ifu_flush .vhd"), mientras que las otras dos unidades (n4/r4 y n4/r5), son distintas en ambos modelos por lo que sus archivos se encuentran en la librería propia de cada modelo. Ver la figura 102.

Podemos ver el esquema definido en el nivel n3/r5 para el modelo 8086 (archivo " up8086_biu_interf_dfu.vhd") el la figura 103 a, donde podemos apreciar de forma resaltada las diferencias que existen entre este esquema y el que define este mismo nivel en el archivo "up8088_biu_interf_dfu.vhd" para el modelo 8088, figura 103 b. Estas diferencias son dos (y aparecen en el 8086):

- lsb_pa _intr, bit bajo (A0) de la dirección de memoria donde vamos a buscar la siguiente instrucción. Utilizado por la unidad de control de la cola de instrucciones "ifu_ctrl" (n4/r3) para alinear la transferencia en el 8086.
- w_queue, escribir palabra en la cola de instrucciones. Cuando IFU baja una palabra de la mimoria, se indica a la cola de instrucciones que escriba dos bytes.

Fig 103.- a) Componente y Esquema Interno de la Unidad IFU para del 8086
a) Componente y Esquema Interno de la Unidad IFU para del 8088



34.1 ENTRADAS Y SALIDAS

En la siguiente tabla, podemos ver las señales de entrada y salida de los componentes de la figura 103. Las señales que son diferentes en ambos modelos, aparecen resaltadas en la figura 103 a.

SEÑAL	TIPO	FUNCIÓN
ph2	N	Reoj interno
res	IN	E la señal interna que resetea todos los componentes del modelo
ifu <-> queue		
queue_full	IN	Indica que la cola de instrucciones está llena.
queue_almost_full	IN	Activa cuando la cola está casi llena.
w_queue	OUT	Le indica que escriba 2 bytes.
queue_write	OUT	Señal de escritura de la cola de instrucciones.
queue_hold	OUT	Congela la cola de instrucciones.
ifu -> arbitro		
ifu_cycle	IN	Le indica que el ciclo de bus actual es de IFU.
pa_instr	OUT	Contiene la dirección del próximo ciclo de bus "instr_fetch"
rq_bus_eu	OUT	Le pide al árbitro el control sobre el bus
gt_bus_ifu	IN	Indica que le han concedido el bus.
ifu <- eu		
new_ip	IN	Le indica que ha cambiado el registro IP.
instr_size	IN	El decodificador nos pasa el número de bytes de la instrucción actual.
flush	IN	Señal que indica que hay un salto en el programa.
exe_done	IN	EU indica que ha terminado la ejecución de la instrucción.
ifu <-> gau		
ifu_pointer	OUT	Le pasa el puntero de IFU
rq_pa_ifu	OUT	Pide el cálculo de la siguiente dirección.
gt_pa_dfu	IN	El cálculo de la dirección ha terminado.
result	OUT	Contiene el resultado de la ultima operación con adder (la dirección)
ifu <-> timing		
begin_bus_cycle	IN	Indica el comienzo de un nuevo ciclo de bus.
bus_free	IN	El bus local está libre.

Tab 55.- Entradas y salidas de DFU.

35. PUNTERO DE LA IFU, IFU_POINTER (n4/r5)

Esta unidad contiene dos registros, el puntero de la unidad IFU (ifu_pointer) y el registro de la dirección física de la siguiente instrucción que la unidad va a descargar de memoria.

Existen algunas diferencias entre la lógica de actualización del puntero de IFU en los distintos modelos, podemos ver las señales que los diferencian resaltadas en la figura 104. Seguidamente las estudiaremos por separado.

ACTUALIZACIÓN DE IFU_POINTER EN EL 8088

El puntero de la unidad IFU, es el que determina cual va a ser la siguiente posición de memoria a la que accedamos para descargar su contenido en la cola de instrucciones. Este puntero, se lo pasamos a la unidad GAU para que calcule la dirección física de la instrucción (PA_instr). La GAU realiza la siguiente cálculo: PA_instr=CS&"0000"+IP+IFU_pointer.

Ese puntero lo actualizan dos señales inc_ifu_pointer y new_ip:

- La activación de la señal new_IP, nos indica que se ha actualizado el registro IP sumándole el número de bytes de la siguiente instrucción. new_ip aparece en la fase de ejecución 0, lo que implica que

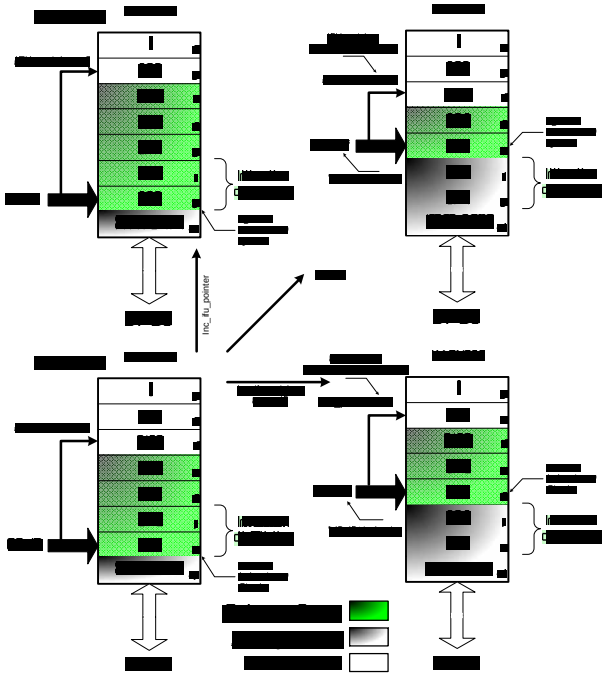


Fig 105.- Actualización del Puntero de IFU en el 8088.

se va a ejecutar la instrucción inmediatamente. Como el puntero de la unidad IFU tiene que seguir

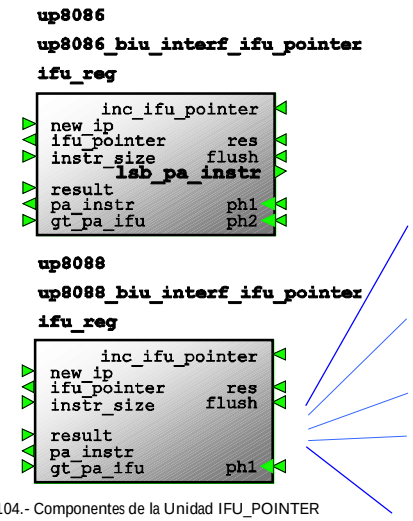


Fig 104.- Componentes de la Unidad IFU_POINTER

```
-----------------------------------------------------------------
ifu_pointer_reg : PROCESS(ph1,res)
-----------------------------------------------------------------
BEGIN
    IF (res = true) THEN
        -- Reset Values
        ifu_pointer_cld <= 0;
    ELSIF (ph1'EVENT AND ph1 = '1') THEN
        -- Default Assignment To Internals
        ifu_pointer_cld <= ifu_pointer_cld;
        -- Combined Actions for internal signals only
        IF flush THEN
            ifu_pointer_cld <= 0;
        ELSIF inc_ifu_pointer AND new_IP THEN
            ifu_pointer_cld <= (ifu_pointer_cld+1) - instr_size;
        ELSIF inc_ifu_pointer THEN
            ifu_pointer_cld <= ifu_pointer_cld +1;
        ELSIF new_ip THEN
            ifu_pointer_cld <= ifu_pointer_cld - instr_size;
        END IF;
    END IF;
END PROCESS ifu_pointer_reg;
-- Asignación de la salida del registro
ifu_pointer <= ifu_pointer_cld;
```

apuntando a la misma dirección de memoria que lo hacía anteriormente (antes de que IP cambiara), hay que restarle el mismo valor que hemos sumado a IP, el valor del número de bytes de la instrucción decodificada (instr_size)

$$PA_instr|_{ANTERIOR} = (CS : IP) + IFU_puntero;$$
$$PA_instr|_{NUEVA} = [CS : (IP + instr_size)] + IFU_puntero;$$
$$Como : PA_instr|_{ANTERIOR} = PA_instr|_{NUEVA} \Rightarrow$$
$$IFU_puntero|_{NUEVO} = IFU_puntero|_{VIEJO} - instr_size$$

- La otra señal que actualiza el puntero IFU_pointer, es la señal "inc_ifu_pointer" generada por la unidad de control de la IFU (IFU_CTRL, n4/r4) que indica que ha sido salvado una instrucción en la cola de instrucciones y hay que incrementa el puntero para generar la dirección física de la siguiente instrucción a descargar de memoria.

Una situación especial, se da cuando aparecen ambas señales a la vez (caso en el que coinciden en el tiempo la escritura de un dato en la cola de instrucciones y la fase de ejecución cero (exe_phase=0), de la instrucción en curso). En este caso:

$$\{PA_instr|_{ANTERIOR} + 1\} = PA_instr|_{NUEVA} \Rightarrow$$
$$IFU_puntero|_{NUEVO} = \{IFU_puntero|_{VIEJO} + 1\} - instr_size$$

Todas las posibles situaciones, aparecen representadas gráficamente en la figura 105 de la página anterior, donte también podemos ver el código VHDL.

35.2 ACTUALIZACIÓN DE IFU_POINTER EN EL 8086

En este modelo, la lógica de control del puntero de la unidad IFU, se complica un poco debido a la posibilidad de que la dirección física generada para acceder a la siguiente instrucción, no sea par (desalineamiento). En este caso, no utilizamos los 16 bits del bus de direcciones en la primera transferencia, sólo utilizamos los 8 bits del banco alto. El puntero IFU_pointeres modificado en las siguientes situaciones:

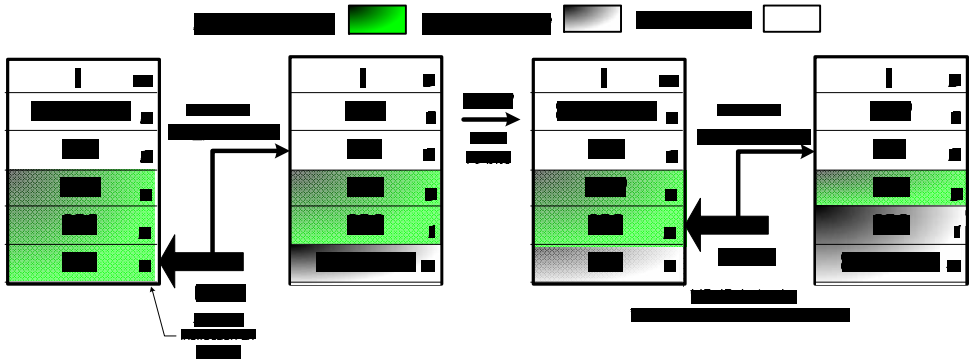


Fig 106.- Actualización de IFU_pointer al aparecer una nueva IP

```
-----------------------------------------------------------------------------
ifu_pointer_reg : PROCESS(ph1,res
-----------------------------------------------------------------------------
BEGIN
    IF (res = true) THEN
        -- Reset Values
        ifu_pointer_cld <= 0;
    ELSIF (ph1'EVENT AND ph1 = '1') THEN
        -- Default Assignment To Internals
        ifu_pointer_cld <= ifu_pointer_cld;
        -- Combined Actions for internal signals only
        IF flush THEN
            ifu_pointer_cld <= 0;
        ELSIF inc_ifu_pointer AND new_IP THEN
        --Caso en el que incrementamos y deco ha terminado
            IF pa_instr_cld(0)='0' THEN
            --Dirección alineada con el bus (PA_instr(0)='0')
                --Escribo una palabra en cola e incremento en 2 el puntero
                ifu_pointer_cld <= (ifu_pointer_cld+2)-instr_size;
            ELSE
            --Dirección desalineada (Misaligned) (PA_instr(0)='1')
                --Escribo un byte en cola e incremento en 1 el puntero
                ifu_pointer_cld <= (ifu_pointer_cld+1)-instr_size;
            --Después de este alineamiento con el bus, se transfieren palabras
            END IF;
        ELSIF inc_ifu_pointer THEN
        --Caso en el que incrementamos el puntero
            IF pa_instr_cld(0)='0' THEN
            --Dirección alineada con el bus (PA_instr(0)='0')
                --Escribo una palabra en cola e incremento en 2 el puntero
                ifu_pointer_cld <= ifu_pointer_cld+2;
            ELSE
            --Dirección desalineada (Misaligned) (PA_instr(0)='1')
                --Escribo un byte en cola e incremento en 1 el puntero
                ifu_pointer_cld <= ifu_pointer_cld+1;
            --Después de este alineamiento con el bus, se transfieren palabras
            END IF;
        ELSIF new_ip THEN
        --En este caso, sólo deco pide actualización del puntero
            ifu_pointer_cld <= ifu_pointer_cld-instr_size;
        END IF;
    END IF;
END PROCESS ifu_pointer_reg;

-- Asignación de la salida del registro
ifu_pointer <= ifu_pointer_cld;
```

CAMBIO EN EL PUNTERO DE INSTRUCCIONES IP

Si la decodificación de la instrucción ha terminado, y por lo tanto entramos en la fase de ejecución 0 (exe_phase=0), la EU actualiza el valor de la IP sumándole el número de bytes de la instrucción que va a ejecutar (IP=IP+instr_size) y activa la señal "new_IP". La activación de esta señal, hace que el puntero "IFU_pointer" se decremente en la misma cantidad, de manera que la generación de la siguiente dirección física para acceder a por instrucciones, sea correcta. Esto se puede ver gráficamente en la figura 106. La actualización de la IP, no provoca un desalineamiento del puntero, ya que el "techo" lo marca el puntero de la IFU. Podemos decir que este puntero se comporta como una gominola, podemos aplastarla con los dedos, pero estos nunca llegan a tocarse.

ALINEAMIENTO DEL PUNTERO IFU_POINTER

Un desalineamiento del puntero IFU_pointer, solamente lo pueden provocar instrucciones que ejecutan un salto en el flujo del programa, es decir, las instrucciones que hacen "flush". No en todos los casos en los que hay un salto en el programa crean un desalineamiento, sólo los saltos a posiciones impares lo hacen. Cuando se hace "flush", se borra el contenido del puntero (IFU_pointer=0), de manera que la dirección física de la siguiente instrucción la determina IP que puede adquirir un valor impar.

Al ser una dirección impar, el único banco del sistema de memoria que estará activo, será el banco alto y sólo tendremos un byte válido para salvar en la cola de instrucciones. El valor del bit PA_instr(0), indica si se va a leer una posición par de memoria, si PA_instr(0)='1', al recibir la señal de incremento del puntero "inc_ifu_pointer", incrementará en 1 el valor del puntero con lo que el puntero queda de nuevo alineado con la memoria. A partir de ahora PA_instr(0), será siempre '0' y por lo tanto la transferencia de 16 bits.

En condiciones normales de alineamiento, este puntero aumenta de 2 en 2, ya que leemos 2 bytes de la memoria

En la figura 107 y en el código de la página anterior, podemos seguir el comportamiento anteriormente explicado.

En la unidad, también se contempla el caso en el que coinciden en el tiempo las señales "new_ip" e "inc_ifu_pointer". En ese caso, se resta al puntero de IFU el valor de "instr_size" y se incrementa el resultado en 1 o dos unidades dependiendo de si el puntero está alineado o no.

35.3 REGISTRO DE PA_instr

Esta unidad, que podemos encontrar en los archivos "up8088_biu_interf_ifu_pointer.vhd" para el modelo 8088 y "up8086_biu_interf_ifu_pointer.vhd" para el 8086, también hay un proceso síncrono que se encarga de registrar la dirección de 20 bits generada por la unidad GAU que utilizaremos para acceder a la siguiente instrucción en memoria. El código lo podemos ver en las siguientes líneas de código:

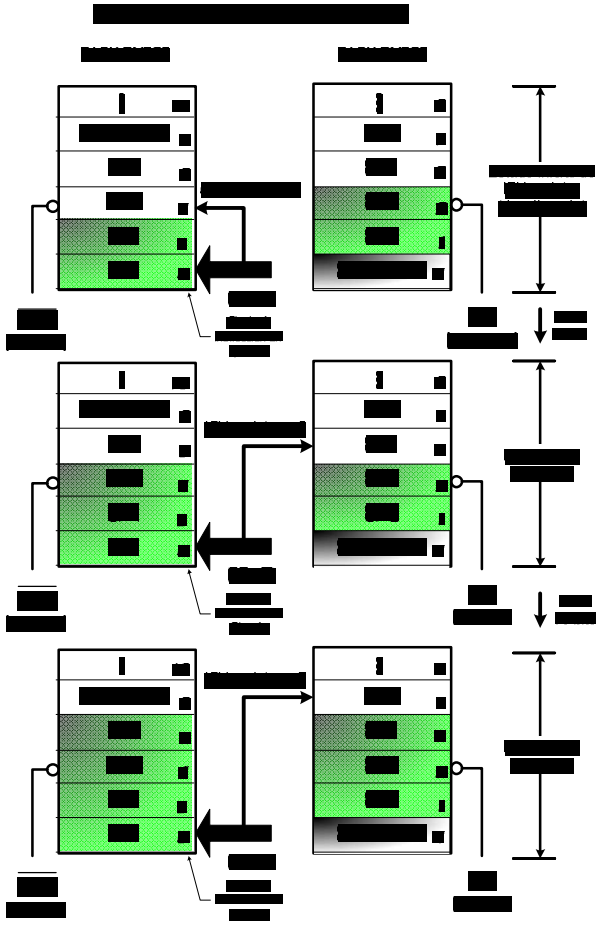


Fig 107.- Alineamiento del Puntero IFU_pointer en el 8086.

```
-----------------------------------------------
pa_instr_reg : PROCESS(ph2,res)
-----------------------------------------------
BEGIN
    IF res OR flush  THEN
        -- Valores de Reset
        pa_instr_cld <= X"00000";
    ELSIF (ph2'EVENT AND ph2 = '1') THEN
        -- Valores por defecto
        pa_instr_cld <= pa_instr_cld;
        IF gt_pa_ifu THEN
            pa_instr_cld <= result;
        END IF;
    END IF;
END PROCESS pa_instr_reg;
-- Asignación de la salida del registro
pa_instr <= pa_instr_cld;
lsb_pa_instr <= pa_instr_cld(0);
```

DELIMITACIÓN DEL PUNTERO IFU_POINTER

El rango (o valores entre los que oscila) el puntero IFU_pointer, está en función de la longitud de la cola de instrucciones. Como de dijo en la características generales de los modelos, la longitud de la cola de instrucciones es parametrizable (por lo tanto variable) y en consecuencia, el rango de "IFU_pointer" también es variable.

Concretamente el valor de IFU_pointer varía desde 0 hasta el valor del número de bytes de la cola de instrucciones "queue_size" más el número de bytes de la instrucción más larga del juego de instrucciones.

Para simplificar la definición de este rango, se creó una constante en el package del sistema (n0/r0) que indica el numero de bytes máximo que puede tener una instrucción.

```
--|                                                    |--
--|cONSTANTES GloBALES                                 |--
--|"max_length_code" es una constante que indica el    |--
--|número de bytes de la Instrucción más larga, sirve|--
--|para establecer el rage de ifu_pionter. ej.:        |--
--|la instrucción "mover un dato inmediato de 16 bits|--
--|a memoria en el segmento extra impidiendo la ce-    |--
--|sión del bus", sería:                               |--
--|          lock mov es:word ptr result2,0F0F0h       |--
--|F0 26 C7 06 0F00 F0F0                               |--
--||  |  |  |  |    |--Dato Inmediato                  |--
--||  |  |  |  |-------Desplazamiento de result2       |--
--||  |  |  |----------2º Byte de OP code              |--
--||  |  |-------------1º Byte de OP code              |--
--||  |----------------Prefijo de Segmento Extra(es:)|--
--||------------------Prefijo de Bloqueo (lock)       |--

    CONSTANT max_length_code : Integer := 8;
```

Utilizando esta constante podemos definir IFU_pointer con

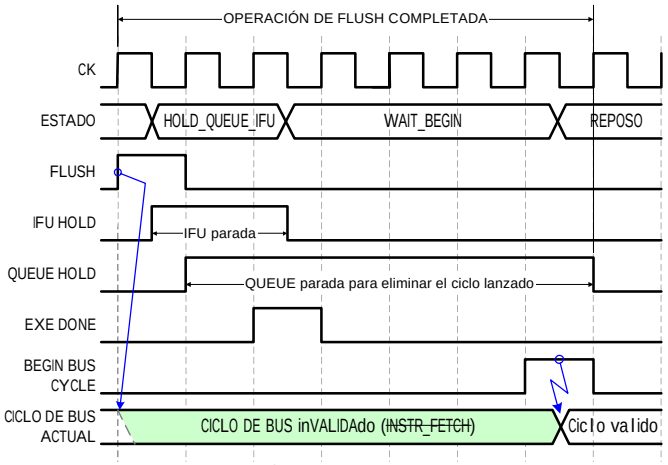
... RANGE 0 TO queue_size+max_lenght_code;

Establecer los márgenes de las señales es de vital importancia para el sintetizador, ni no indicamos el margen de valores que puede adquirir una variable, el sintetizador suele codificarla en registros de 32 bits.

36. UNIDAD DE SINCRONIZACIÓN DE FLUSH, FLUSH (n4/r3)

n4/r3 Unidad de Flush (ifu_flush): Cuando aparece un salto en el flujo del programa, es necesario sacar todos los bytes de la cola de instrucciones (flush) e inicializar IFU. Esta unidad es la encargada de sincronizar estas operaciones.

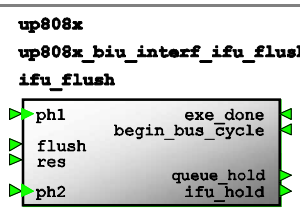
La señal de flush, es generada por la cgrom (n2/r1), cuado la lógica de la instrucción que está ejecutando, determina un salto a otra parte del programa. Este salto, provoca un cambio inesperado en el contenido de IP que hace que las instrucciones descargadas en la cola de instrucciones, no sean las siguientes en ejecutarse y en consecuencia hay que borrarlas y empezar de nuevo a descargar bytes de instrucciones de la memoria a partir de la nueva dirección donde continúa el programa (CS:IP). La señal de flush, por lo tanto, inicializa la cola de instrucciones, pero también es necesario inicializar la unidad que opera sobre esta (la unidad IFU). Es necesario inicializar el puntero IFU_pointer para que la siguiente instrucción sea la indicada en CS:IP, y es necesario inicializar la unidad de control IFU_CRL (n4/r4) para que aborte la petición de bus y recalcule la dirección "IP_instr" acorde con la nueva situación. La inicialización de la cola de instrucciones y la unidad IFU, requiere una sincronización con el bus externo ya que tenemos que anular un posible ciclo de bus lanzado y esperar a que este concluya. En la siguiente figura, podemos ver el sincronismo entre las unidades inicializadas y el bus (secuencia de flush):



Tab 108.- Típica Secuencia de Flush.

36.1 ENTRADAS Y SALIDAS

Como podemos observar en la figura 102, este componente es el único común a los dos modelos de la unidad IFU. Define un grafo de estados y lo podemos encontrar en el archivo "up808x_biu_interf_ifu_flush .vhd" de la librería común. En la siguiente tabla, podemos ver el componente y la función de las señales de la unidad de flush:



SEÑAL	TIPO	FUNCIÓN
ph1,ph2	IN	Reoj interno
res	IN	E la señal interna que resetea todos los componentes del modelo
flush -> queue		
queue_hold	OUT	Congela la cola de instrucciones.
flush <- eu		
flush	IN	Señal que indica que hay un salto en el programa.
exe_done	IN	EU indica que ha terminado la ejecución de la instrucción.
flush <-> ifu		
ifu_hold	OUT	Paraliza el funcionamiento de ls unidad IFU.
ifu <- timing		
begin_bus_cycle	IN	Indica el comienzo de un nuevo ciclo de bus.

Tab 56.- Entradas y Salidas de IFU_FLUSH (archivo "up808x_biu_interf_ifu_flush.vhd").

36.2 ESTADOS

Esta es una unidad muy simple, se encarga de establecer la secuencia que podemos ver en la figura 108, para ello, necesitamos tres estados (como podemos ver en la figura 109):

- REPOSO, estado inicial (e0). Permanecemos en este estado hasta que la unidad de ejecución nos indica que hay que realizar un flush del sistema. La EU activa la señal "flush" (transición 1, t1) y saltamos al estado e1 iniciando de esta manera la secuencia de flush.
- HOLD_QUEUE_IFU, paraliza las operaciones con la cola de instrucciones (e1). En este estado, la unidad de flush activa la señal "queue_hold" que bloquea toda operación con los registros de la cola de instrucciones, impidiendo que el ciclo de bus lanzado por la IFU (antes de aparecer la señal "flush"), grabe datos en estos registros. En la fase dos (ph2) de este estado, aparece la señal "IFU_hold" que al igual que hace "queue_hold" con la cola de instrucciones, "ifu_hold" paraliza el funcionamiento de la unidad IFU_CTRL en este caso para evitar el lanzamiento de otro ciclo de bus. Permanecemos en este estado hasta la finalización de ejecución de la instrucción actual exe_done (t1).
- WAIT_BEGIN, esperamos a que comience un ciclo de bus válido, (e3). Al entrar en este estado la señal "ifu_hold" deja de bloquear la cola de instrucciones y esta empieza a operar (pidiendo la generación de PA_instr), sin embargo, la cola de instrucciones permanece bloqueada hasta que comience un ciclo de bus válido, es decir, un ciclo de bus que va a por instrucciones a la dirección del salto.

En la siguiente figura, podemos ver el grafo de estados:

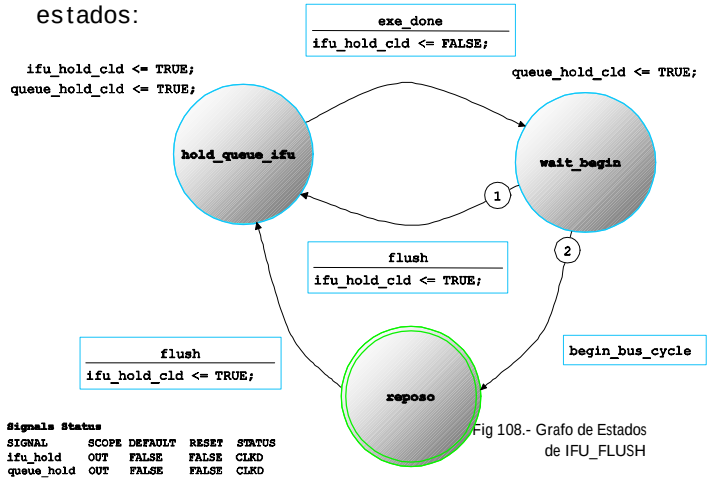


Fig 108.- Grafo de Estados de IFU_FLUSH

SECUENCIA	DESCRIPCIÓN
reposo-hold_queue_ifu-wait_begin-reposo	Secuencia de flush típica
reposo-hold_queue_ifu-wait_begin-hold_queue_ifu-wait_begin-reposo	Aparece una interrupción después de una instrucción de salto

Tab 56.- Posibles Caminos del Grafo de Estados.

37. UNIDAD DE CONTROL DE LA IFU, IFU_CTRL (n4/r4)

Esta es la última unidad de los modelos que nos queda por analizar, junto con las unidades ifu_flush y ifu_reg, forman la unidad IFU.

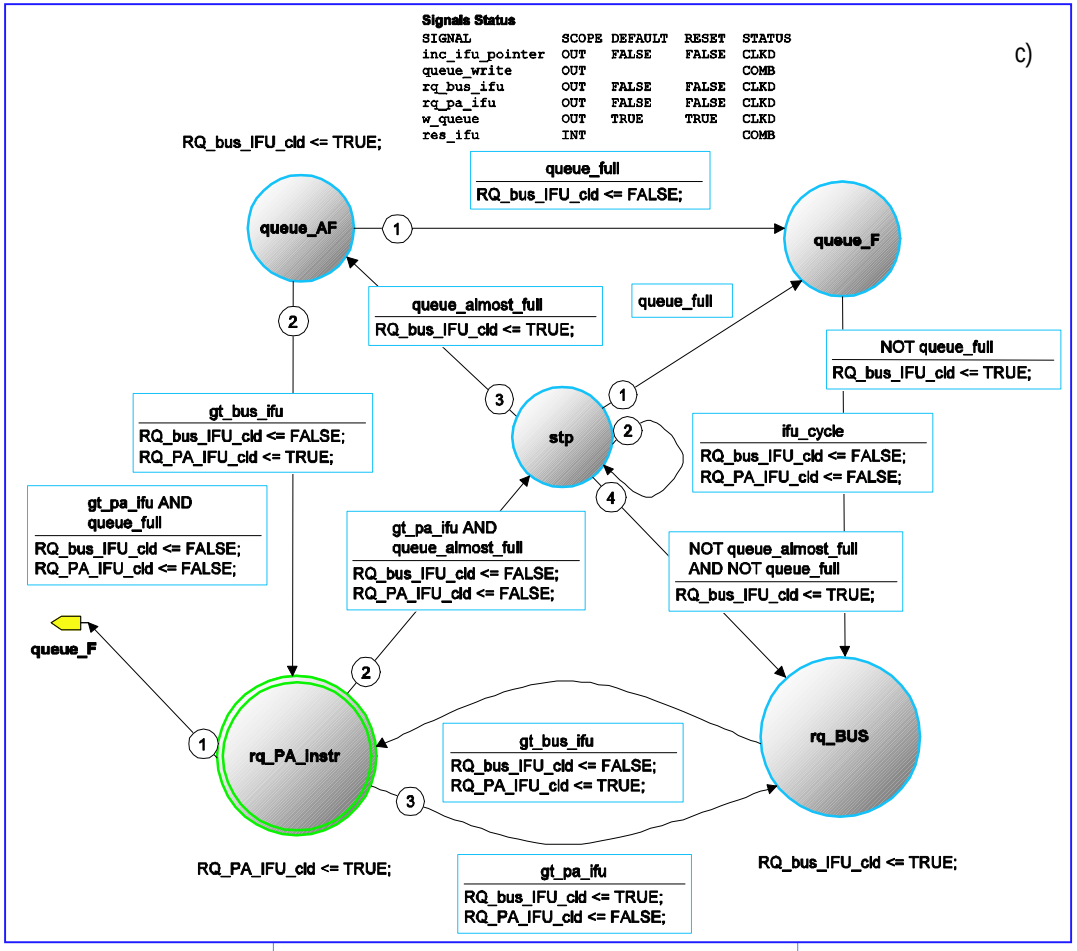
n4/r4 Lógica de Control de IFU (ifu_ctrl): Contiene el grafo de conducción de la unidad IFU, desde aquí se "lanza" el ciclo de bus que va en busca de instrucciones (instruction_fetch) para llenar la cola de instrucciones.

Esta unidad trabaja de forma independiente de la EU, la única comunicación que entablan estas dos unidades es al realizar un flush del sistema, y lo hacen a través de la unidad ifu_flush (n4/r3) estudiada anteriormente. Junto con las unidades DFU (n3/r4) y MU (n3/r6), son las tres unidades que acceden al bus local, a través del árbitro de bus (n3/r3). Esta unidad, accede al bus local exclusivamente para ir a la memoria en búsqueda de instrucciones que salvar en QUEUE (n1/r5). A diferencia de la unidad DFU, sólo pide la ejecución del ciclo de bus "instr_fetch". Para acceder a la posición de la siguiente instrucción a descargar (que no a ejecutar) de la memoria, esta unidad le pide a GAU (n3/r7) que calcule la dirección física de la siguiente instrucción (PA_instr) a través de las señales de Petición/Cesión (ReQuest/GranT) "rq_PA_instr" y "gt_PA_instr". Una vez que la instrucción ha sido calculada y siempre que en la cola de instrucciones quepan más datos, pide al árbitro de bus el control sobre este; la comunicación se realiza a través de las señales "rq_bus_ifu" y "gt_bus_ifu". Para comunicarse con la cola de instrucciones, utiliza las señales "queue_almost_full" y "queue_full" que le indican el estado de la cola de instrucciones.

37.1 ENTRADAS Y SALIDAS

SEÑAL	TIPO	FUNCIÓN
ph2	N	Reoj interno
res	IN	E la señal interna que resetea todos los componentes del modelo
ifu_queue <-> queue		
queue_full	IN	Indica que la cola de instrucciones está llena.
queue_almost_full	IN	Activa cuando la cola está casi llena.
w_queue	OUT	Le indica que escriba 2 bytes.
queue_write	OUT	Señal de escritura de la cola de instucciones.
ifu_ctrl <-> arbitro		
ifu_cycle	IN	Le indica que el ciclo de bus actual es de IFU.
rq_bus_eu	OUT	Le pide al árbitro el control sobre el bus
gt_bus_ifu	IN	Indica que le han concedido el bus.
ifu_ctrl <- ifu_flush		
ifu_hold	IN	Congela el estado de la unidad
ifu_ctrl <- eu		
flush	IN	Señal que indica que hay un salto en el programa.
ifu_ctrl <-> gau		
rq_pa_ifu	OUT	Pide el cálculo de la siguiente dirección.
gt_pa_dfu	IN	El cálculo de la dirección ha terminado.
ifu <-> timing		
bus_free	IN	El bus local está libre.

Tab 57.- Entradas y Salidas de IFU_CTRL (archivos "[up8088/up8086]_biu_interf_ifu_ctrl.vhd").



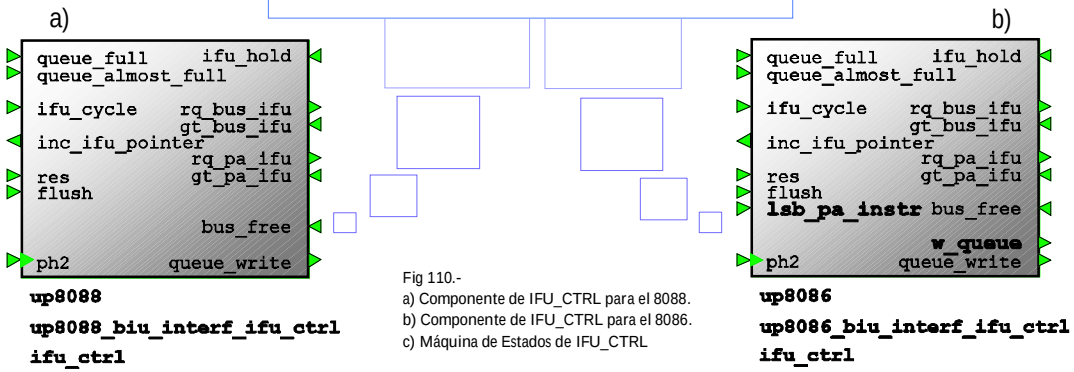


Fig 110.-
a) Componente de IFU_CTRL para el 8088.
b) Componente de IFU_CTRL para el 8086.
c) Máquina de Estados de IFU_CTRL

37.2 EXPLICACIÓN

En la figura 110, se puede observar que ambos componentes contienen un grafo de estados similar. En realidad y aunque en la figura 102 observemos que una unidad distinta para cada modelo, las diferencias son tan mínimas que podemos considerarla como una unidad común. De hecho, en las librerías más recientes, es un elemento común. La única diferencia que podemos encontrar entre las dos versiones es la acción global que podemos ver a continuación::

```
w_queue_cld <= w_queue_cld;
IF gt_bus_ifu THEN
    IF lsb_pa_instr='0' THEN
    --Transferencia Alineada, leo 16 bits
        w_queue_cld <= TRUE;
    ELSE
    --Transferencia desalineada, leo 8 bits y alíneo
        w_queue_cld <= FALSE;
    END IF;
END IF;
```

Esta acción ha sido llevada a IFU_POINTER de manera que las diferencias entre ambas carcasas (resaltadas en la figura 102 b) y los grafos, desaparecen y hacemos este componente común a ambos modelos.

LÓGICA DEL GRAFO

En la figura 102 a, podemos ver el grafo de esta unidad, formado por 5 estados, los que podemos ver de mayor diámetro, hacen referencia la secuencia normal (generación de PA-petición de bus-generación de PA). Los otros tres estados implementan la lógica de "parada" en función del estado de la cola de instrucciones. Los Estados que nos encontramos son:

- ra_PA_instr, pide la generación de PA_instr, (e0). Es el estado inicial, en este estado, la unidad IFU_CTRL activa la señal de petición de PA (rq_pa_ifu) que le indica a la unidad generadora de direcciones que calcule la dirección física de la siguiente instrucción que IFU tiene que bajar de memoria. Como se indicó en la explicación de IFU_POINTER, este calculo lo realiza sumándole a CS:IP el desplazamiento en la señal "ifu_pointer". A diferencia de DFU_CTRL, esta unidad siempre utiliza el mismo tipo de direccionamiento, de ahí que GAU cuando recibe la señal de petición ya sabe que direccionamiento utilizar por lo que nos ahorramos indicárselo. Por lo anterior, sabemos el tiempo que la GAU va a tardar en calcular la PA_instr, el tipo de direccionamiento realiza dos sumas, por lo que en dos ciclos tendremos el resultado (indicado por la activación de la señal "gr_PA_ifu"). De e0 lo normal es saltar a rq_BUS (t3), pero en caso de estar la cola de instrucciones llena ("queue_full, t1) o casi llena (queue_almost_full, t2), saltamos a los estados queue_Full o stp y anulamos la petición de bus.

- rq_BUS, petición de bus, (e1). En este estado (y con la dirección PA_instr calculada), la unidad establece una comunicación con el árbitro de bus a través de las señales "rq_bus_ifu" y "gt_bus_ifu" con la intención de conseguir el control del bus y ejecutar un ciclo de búsqueda de instrucciones. Cuando el bus nos es concedido ("gt_bus_ifu" activada) saltamos a e0 (t1) para calcular la dirección de la siguiente instrucción.

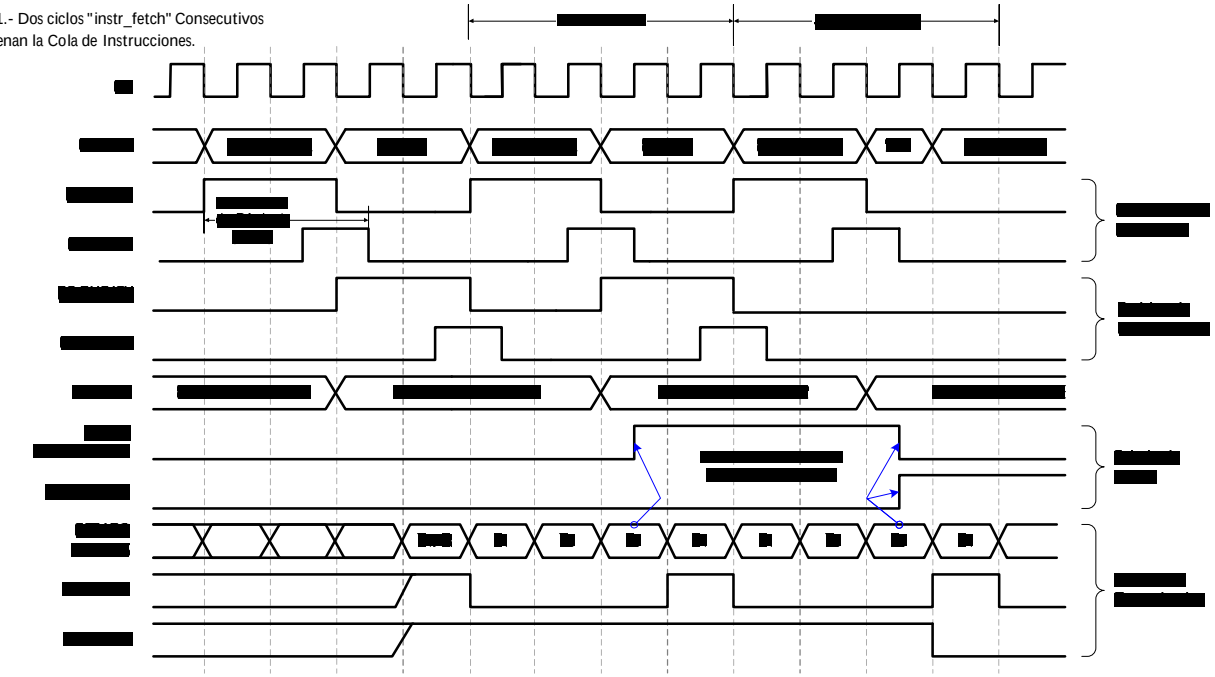
- stp, SToP o parada de IFU, (e2). Si la cola de instrucciones está casi llena (queue_almost_full), y el ciclo de bus actual es de la unidad IFU ("ifu_cycle" activada), hay que parar la petición de bus por parte de IFU_CTRL ya que al finalizar el ciclo de bus actual, la cola de instrucciones estará llena. De esto se encarga este estado, en el que permanecemos hasta que finaliza el ciclo de bus actual ("ifu_cycle", t2) y la cola de instrucciones está llena ("queue_full", t1) o permanece casi llena ("queue_almost _full", t3) o deja de estar casi llena al lanzar una decodificación (t4).

- queue_F, cola de instrucciones llena, e(3). En este estado, permanece la unidad parada hasta que no haya algún hueco en la cola de instrucciones (NOT queue_full, t1).

- queue_AF, cola de instrucciones casi llena, (e4). En este estado permanecemos cuando la cola de instrucciones está casi llena. En este estado, y habiéndonos asegurado de que el ciclo de bus actual ha terminado, pedimos el control sobre el bus para realizar otra transferencia y llenar la cola. Si tras esta nueva lectura la cola está llena, saltamos al estado e3, mientras que si pese a la lectura la cola sigue estando casi llena o hay más de 1 espacio (lo que indica que el decodificador ha cogido datos), saltamos al estado inicial (e0) para calcular la nueva dirección..

En el Cronograma de la figura 111, podemos ver como dos ciclos de búsqueda de instrucciones llenan la cola de instrucciones y la unidad de control IFU_CTRL interrumpe su actividad con el cálculo de la siguiente dirección. En el momento en el que el decodificador lee algún dato de la cola, DFU_CTRL abandona el reposo y lanza el siguiente ciclo de bus sin necesidad se calcular la siguiente PA_instr.

Fig 111.- Dos ciclos "instr_fetch" Consecutivos que Llenan la Cola de Instrucciones.



modelo software de los microprocesadores

38. MODELO SOFTWARE DEL MICROPROCESADOR

La programación requiere un claro entendimiento de la estructura de los registros internos de los modelos diseñados (los mismos que los de la familia 8088/8086 de Intel ®). A continuación podremos ver los registros internos visibles por el programador y la forma en que se direcciona la memoria por medio de los registros de segmento y desplazamientos de dirección.

En la figura 1 podemos ver los registros internos del microprocesador. Se pueden distinguir cuatro tipos de registros:

- Registros de Uso General (punteros).
- Registros Base/Índice.
- Registros de Segmento.
- Registro de Banderas.

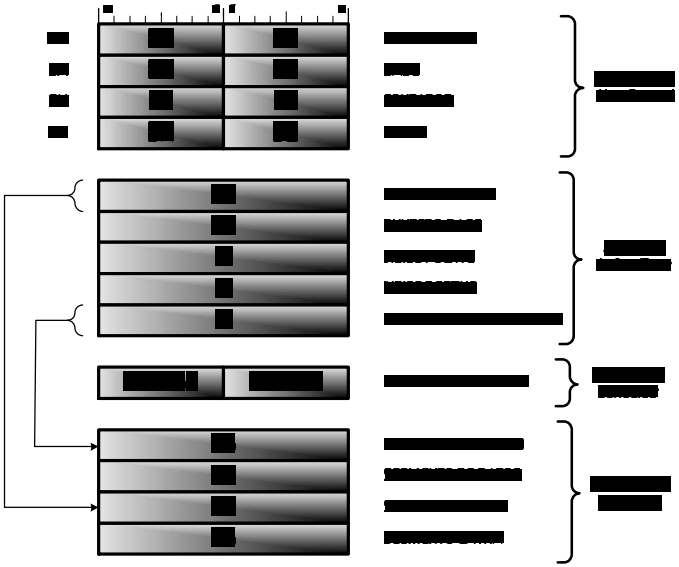


Fig 1.- Registros del Modelo Software.

38.1 REGISTROS DE PROPÓSITO GENERAL

Los registros de uso general pueden ser utilizados por el programador como fuente o destino en las instrucciones. Pueden ser direccionados como registros de 16 u 8 bits. Estos registros suelen utilizarse por algunas instrucciones para tareas específicas, de ahí que cada registro tenga un nombre específico.

- AX, AH/AL (Acumulador): suele conservar el resultado temporal después de una operación aritmética o lógica.
- BX, BH/BL (Base): Suele tener la dirección base (desplazamiento) de los datos que hay en la memoria o la dirección base de una tabla de datos referenciados por la instrucción para convertir (ver XLAT en el juego de instrucciones).
- CX, CH/CL (Contador): contiene el número de desplazamientos en instrucciones de rotación (en CL) y el número de interacciones en la instrucción de bucles (LOOP).
- DX, DH/DL (Datos): En las instrucciones de multiplicación contiene la parte mas significativa del resultado y la parte más significativa del dividendo antes de la división. También contiene el puerto de 16 bits en las instrucciones IN/OUT.

38.1.1 REGISTROS BASE E ÍNDICES

Aunque los registros base e índices son también de uso general, se utilizan más a menudo para apuntar a la posición de memoria que contiene los datos del operando en muchas instrucciones.

- SP (Puntero de Pila). Se emplea para direccional datos en una pila de memoria con un comportamiento de memoria LIFO (Last In Frist Out, último en entrar primero en salir). Las Instrucciones PUSH y POP utilizan la pila de instrucciones al igual que, implícitamente, también lo hacen las instrucciones CALL/RET para salvar/recuperar los valores de ciertos registros al atender una subrutina.
- BP (Puntero Base). Es un puntero de uso general que se utiliza a menudo para direccionar una matriz de datos en una pila de memoria.
- SI (Índice Fuente). Se emplea para direccional datos fuente en forma indirecta, estos son utilizarlos con las instrucciones de cadenas.
- DI (Índice Destino). Se suele emplear para direccionar datos destino de forma indirecta cuando son utilizados con las instrucciones de manipulación de cadenas de datos.
- IP (Puntero de Instrucciones). Se utiliza siempre para direccional la siguiente instrucción que va a ejecutar el microprocesador. Para formar la localidad real de la siguiente instrucción, se suma el contenido de IP a CS multiplicado por 10h.

Suele hacerse direccionamiento indirecto de datos por medio de cuatro de estos cinco registros de 16 bits, pero nunca con IP.

38.1.2 REGISTRO DE BANDERAS

Las “banderas” indican la condición actual del microprocesador a la vez que controlan su funcionamiento en algunos aspectos.

Los bits de bandera cambian después de ejecutar muchas de las instrucciones aritméticas y lógicas. Algunas de las banderas se utilizan para controlar ciertas características del microprocesador.

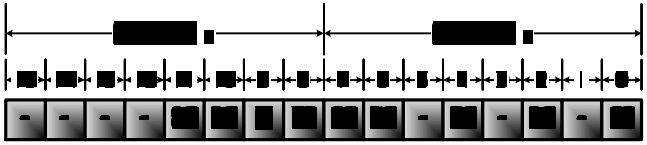


Fig 2.- Posicionamiento de los flags en el registro de banderas.

Los bits de bandera están posicionados en el registro de banderas como se muestra en la figura 2. A continuación podemos ver su significado:

- CF Acarreo (Carry Flag, bit 0): Indica que hubo acarreo o "arrastre" (en caso de suma) hacia, o un borrow o "préstamo" (en caso de resta) desde el bit de orden más significativo del resultado. Este indicador es usado por instrucciones que suman o restan números que ocupan varios bytes. Las instrucciones de rotación pueden “aislar” un bit de la memoria o de un registro poniéndolo en el CF. Algunos programadores utilizan esta bandera para detectar condiciones de error en ciertos procedimientos o programas.
- PF Paridad (Parity Flag, PF, bit 2). Activado si hay un número par de 1 lógicos en el resultado. Borrado si el número de unos es impar. A veces es utilizado en programas de comunicación o como una simple comprobación de seguridad.

- AF Acarreo Auxiliar (Auxiliary Carry, AF, bit 4). Se activa cuando hay un acarreo en la mitad baja de un número de 8 o 16 bits tras operaciones incrementales, o cuando hay un "me llevo una" (borrow a partir de ahora) desde la mitad alta hacia la baja en operaciones de resta o decrementales. Principalmente es actualizado en instrucciones aritméticas. Es un indicador muy especializado, utilizado en exclusiva por las instrucciones DAA y DAS para ajustar el valor de AL después de una suma o resta BCD.
- ZF Cero (Zero Flag, ZF, bit 6). Se activa cuando el resultado de una operación lógico o aritmética es cero.
- SF Signo (Sign Flag, SF, bit 7). Se activa cuando el bit de mayor orden del resultado es 1. En otras palabras, SF = 0 para números positivos y SF = 1 para números negativos.
- TF Trampa (Trap Flag, TF, bit 8). Este flag es utilizado por los depuradores para realizar un simple paso (ejecutar una instrucción) en el programa. Cuando éste flag está activado, se genera una INT 3 (interrupción de tipo 3) después de cada instrucción.
- IF Habilitación de Interrupción (Interrupt enable Flag, IF, bit 9). Las interrupciones están habilitadas cuando ésta bandera está activada. Cuando no, las interrupciones (excepto las no enmascarables), están deshabilitadas. Habilita el pin externo INTR. Las instrucciones STI/CLI activan/desactivan respectivamente esta bandera.
- DF Dirección (Direction, DF, bit 10). Activado/ Desactivado, establece un autodecremento/ autoincremento de DI/SI en instrucciones de cadenas. Las Instrucciones STD/CLD activan/desactivan esta bandera.
- OF Desbordamiento (Overflow Flag, OF, bit 11). Se activa cuando ocurre un desbordamiento aritmético. Un desbordamiento aritmético ocurre cuando el resultado de un cómputo excede la capacidad del destino. Por ejemplo, si se suma un 7Fh (+127) a 01h (+1) el resultado es 80h (-128). Este resultado representa un desbordamiento en la suma con signo, señalado por OF. Para operaciones con números sin signo, no se tiene en cuenta esta bandera. 80h en números sin signo es 128.

38.1.3 REGISTROS DE SEGMENTO

Los registros de segmento (junto con otros registros del microprocesador) generan direcciones en la memoria. Existen 4 registros de segmento y 1 puntero de instrucciones, todos ellos son de 16 bits y a diferencia algunos de los registros de propósito general, no podemos acceder al byte alto y bajo por separado.

A continuación podemos ver los nombres de los registros de segmento y su función:

- CS Segmento de Código, Code Segment: El segmento de código es una sección de memoria que tiene el código de las instrucciones y procedimientos utilizados por los programas. El registro de segmento de código define la dirección inicial de la sección de memoria que tiene el código. Todos los segmentos están limitados a un máximo de 64K bytes de datos.
- DS Segmento de Datos, Data Segment. El segmento de datos en una sección de la memoria que contiene la mayor parte de los datos utilizados por un programa. Se accede a ellos a través de éste y un desplazamiento contenido en otro registro (un registro de uso general).
- ES Segmento Extra, Extra Segment. Es un segmento adicional para almacenar datos, algunas instrucciones de cadenas lo utilizan implícitamente.
- SS Segmento de Pila, Stack Segment. Define las posiciones de memoria reservadas para la pila de instrucciones. La ubicación del punto inicial de entrada a la pila, se determina por el registro apuntador SP. Alternativamente, también podemos utilizar el registro BP para direccionar los datos de la pila.

38.2 REGISTROS DE SEGMENTO Y DESPLAZAMIENTO IMPLÍCITOS

Los microprocesadores tienen un grupo de reglas que se aplican siempre que se direcciona a la memoria. Estas reglas definen la combinación de registro de segmento y de desplazamiento que se utilizan en ciertos modos de direccionamiento. Por ejemplo, el registro de segmento de código (CS) se emplea siempre con el puntero de instrucciones para acceder la siguiente instrucción a ejecutar en el programa. Esta combinación es CS:IP. El registro de segmento de código define el principio de un segmento de código y el puntero de instrucciones apunta a la siguiente instrucción, dentro del segmento de código, a ejecutar por el microprocesador. Por ejemplo, si CS=1400h e IP=1200h, el micro busca y lee la siguiente instrucción en la localidad 14000h+1200h o sea 1520h en la memoria.

Otro direccionamiento implícito es el de la pila. Se hace referencia a los datos en la pila por medio del segmento de pila (SS) y por la localidad de la memoria a la cual direcciona el puntero de pila (SP) o puntero base (BP). Haremos referencia a estas combinaciones mediante SS:SP o SS:BP. Por Ejemplo, si SS=2000h y BP=3000h, el microprocesador direcciona la localidad 23000h en la memoria del segmento de pila direccionada por el puntero base BP.

En la siguiente tabla se pueden ver otros direccionamiento implícitos:

SEGMENTO	DESPLAZAMIENTO
CS	IP
SS	SP oBP
DS	BX, DI, SI o número de 16 bits
ES	DI para instrucciones con cadenas

Tab 1.- Segmentos y Desplazamientos Implícitos.

38.3 DIRECCIONAMIENTO FÍSIC0 Y LÓGIC0

Diferencias entre memoria física (PA) y Lógica (Segmento:OffSet). Ver la siguiente figura:

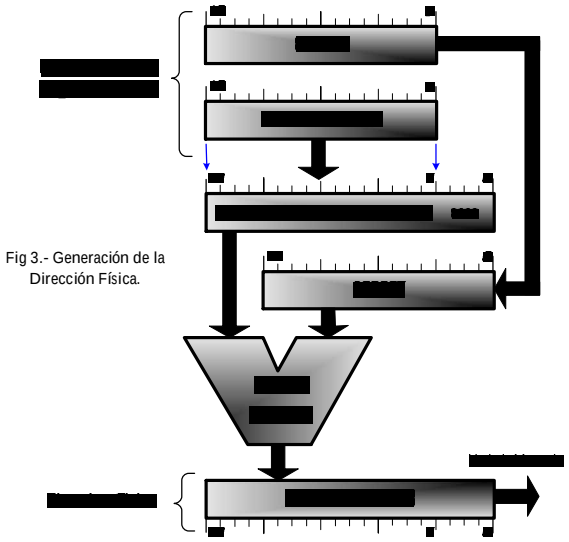


Fig 3.- Generación de la Dirección Física.

Es bueno que el programador piense que existen dos tipos de direcciones, la física y la lógica. La dirección física es de 20 bits e identifica un byte en el espacio de memoria (el rango es de 0h-0FFFFFh). Todas las transferencias entre el micro y la memoria utilizan este direccionamiento.

En cuestiones de programación, es recomendable asumir el direccionamiento lógico que "aisle" al programador de la dirección física real donde se va a posicionar el código. Una dirección lógica consta de un segmento base y un Offset. Para una posición de memoria cualquiera, el segmento base indica la posición inicial del segmento y el Offset el desplazamiento en bytes desde esa posición inicial hasta la posición de memoria indicada. Ver la figura 4. Los segmentos base y el Offset se representan por valores de 16 bits (por ejemplo (0100h:0020h indica la posición física 01020h)..

Una posición física puede ser representada lógicamente de varias formas. La figura siguiente muestra como se puede acceder a la dirección física 2C3h en base a dos segmentos solapados, uno comienza en 2B0h y el otro en 2C0h. La dirección física 002C3h puede representarse lógicamente como 02B0h:0013h o 02C0h:0003h. Ver la siguiente figura:

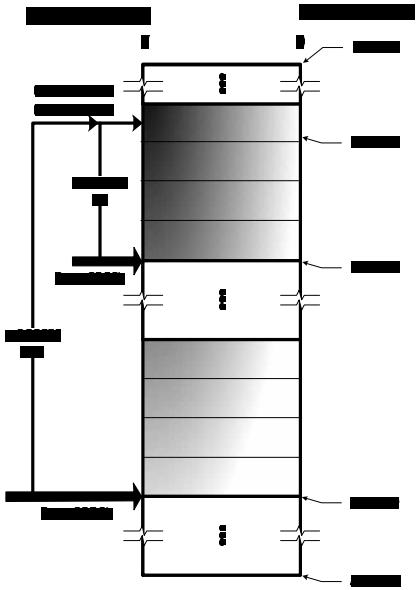


Fig 4.- Direccionamiento Lógico y Físico.

38.3 MEMORIA SEGMENTADA

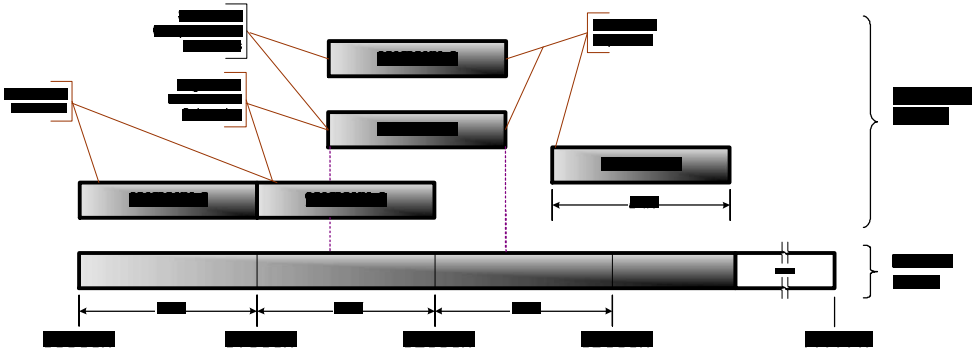


Fig 5.- Posicionamiento de los Segmentos en la Memoria Física.

Los microprocesadores permiten tener cuatro segmentos de memoria. Se debe tener en cuenta que los segmentos de memoria pueden tocarse o incluso solaparse si no se requieren 64K bytes de memoria para un segmento. Se puede decir que los segmentos son como ventanas que se pueden mover a cualquier parte de la memoria. Ver la figura 5.

Supongamos que un programa de aplicación requiere 1000h bytes de memoria para su código, 190h bytes para sus datos y 200h bytes para su pila. Esta aplicación no requiere un segmento adicional. El sistema operativo se encarga de posicionarlo en la memoria. En la figura 6 se muestra la forma en la que esta aplicación es almacenada en la memoria. Los registros muestran una superposición o solapamiento debido a que la cantidad de datos que hay en ellos, no requiere 64K bytes de memoria. Una supuesta vista lateral de los segmentos muestra como se montan los segmentos en la memoria física.

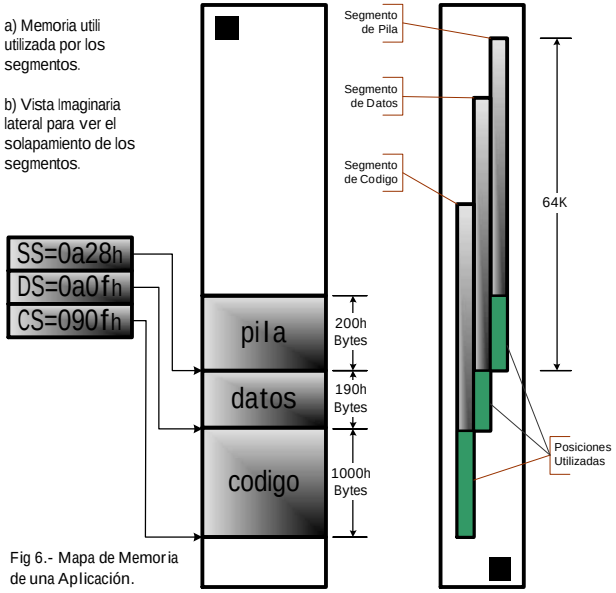


Fig 6.- Mapa de Memoria de una Aplicación.

38.4 EL DIRECCIONAMIENTO DE SEGMENTO Y DESPLAZAMIENTO PERMITE LA RELOCALIZACIÓN DINÁMICA DEL CÓDIGO

El sistema de direccionamiento de segmento desplazamiento parece ser demasiado complicado. Si, lo es, pero también ofrece una gran ventaja al sistema. La forma un tanto compleja de direccionamiento, permite el cambio de lugar de los programas en el sistema de memoria. Un programa relocalizable es el que se puede poner en cualquier zona de la memoria y ejecutarlo sin cambio. Los datos relocalizables son los que se pueden colocar en cualquier zona de la memoria y utilizarlos sin ningún cambio en el programa. El sistema de direccionamiento de estos micros, permite relocalizar los programas y los datos sin cambiar nada, absolutamente, en un programa o en los datos.

Esto es perfecto para utilizarlo en un sistema de computadora de uso general, cuando no todas las máquinas contienen las mismas zonas de memoria. La estructura de la memoria de la computadora personal es diferente entre una máquina y otra, por lo cual requieren software y datos que se puedan relocalizar.

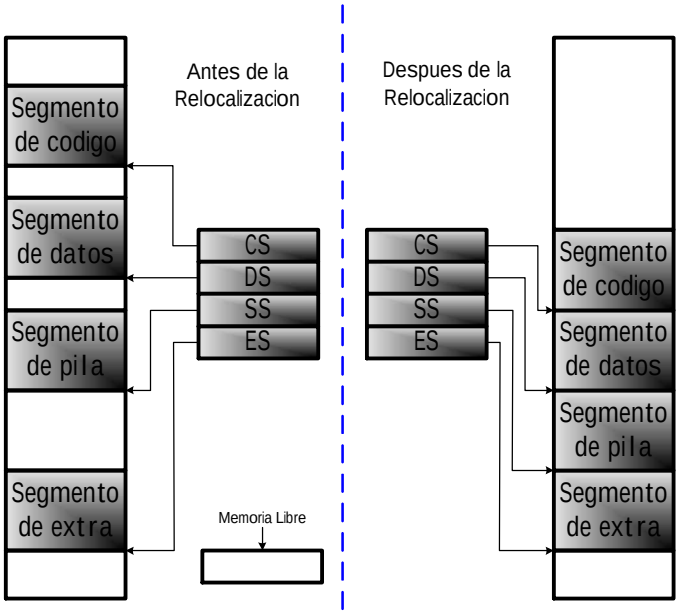


Fig 7.- Relocalización Dinámica del Código.

Debido a que a las localidades de memoria se las direcciona dentro de un segmento con una dirección de desplazamiento, el segmento se puede cambiar sin cambiar ninguna de las direcciones de desplazamiento. Para lograrlo, se mueve todo el programa, como bloque, a una nueva zona y sólo se cambia el contenido de los registros de segmento. Si una instrucción está 4 bytes arriba del inicio de segmento, su dirección de desplazamiento es 4 por lo que si movemos todo el programa a una nueva zona, esta dirección de desplazamiento 4 todavía apunta a los 4 bytes que están arriba del inicio de segmento. Lo único que se necesita cambiar es el contenido del registro de segmento para relocalizar el programa en una nueva zona de memoria. Esta facilidad de relocalización ha hecho que las computadoras personales basadas en los microprocesadores de Intel ® sean muy potentes. Si no se tuviera esta característica, habría que volver a escribir o alterar un programa antes de moverlo. Esto necesitaría tiempo adicional o requeriría un programa para cada una de las muchas configuraciones diferentes de los sistemas de computadoras.

38.5 TIPOS DE DATOS SOPORTADOS DIRECTAMENTE POR LOS

TIPO	DESCRIPCIÓN
Integer	Un valor numérico con signo de 8 o 16 bits (signed byte or word). Todas las operaciones asumen una representación en complemento a dos.
Ordinal	Un valor numérico binario sin signo de 8 o 16 bits (unsigned byte or word).
BCD	Una representación de un dígito decimal (0-9) contenido en un byte (sin empaquetar).
ASCII	Una representación de un carácter alfanumérico o de control en un byte utilizando el estándar ASCII.
BCD Packed	Una representación de dos dígitos decimales (00-99) en un byte (BCD empaquetado, packed BCD)
String	Una secuencia contínua de bytes o words. Una cadena de datos puede contener de 1 byte a 64Kbytes.
Pointer	Una cantidad de 16 o 32 bits. Un puntero de 16 bits está formado por un Offset de 16 bits, mientras que un puntero de 32 bits está formado por una componente base de 16 bits (selector) más una componenete de Offset, también de 16 bits.

Tab 2.- Tipos de Datos Soportados por los Micros.

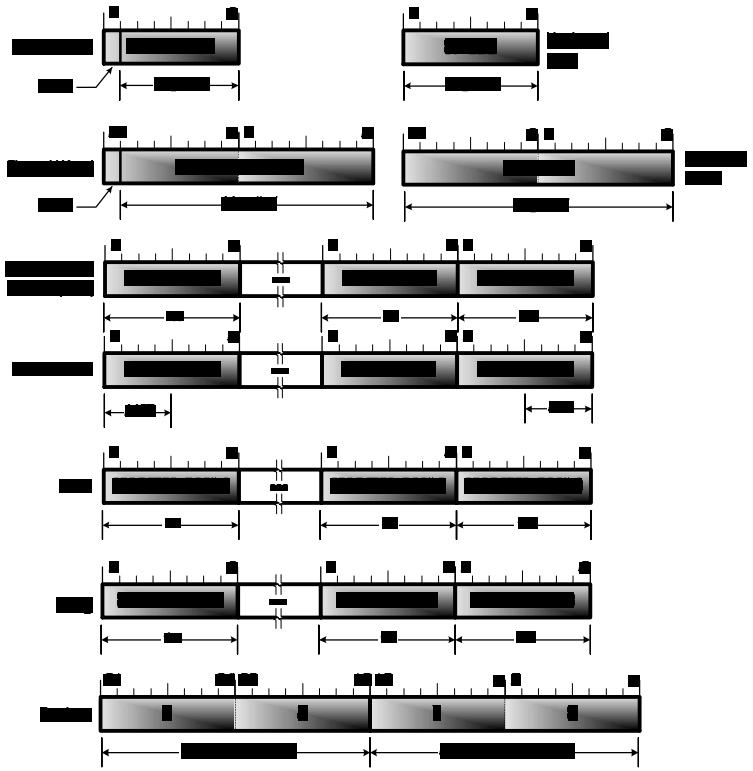


Fig 8.- Representación Gráfica de los Tipos de Datos Soportados Directamente.

38.6 MODOS DE DIRECCIONAMIENTO DEL 8086/8088

Estos procesadores tienen 27 modos de direccionamiento (una cantidad bastante más grande que los microprocesadores anteriores) o reglas para localizar un operando de una instrucción. Tres de ellos son comunes a microprocesadores anteriores:

- DIRECCIONAMIENTO INMEDIATO (el operando es un número que se encuentra en la misma instrucción),
- DIRECCIONAMIENTO A REGISTRO (el operando es un registro del microprocesador)
- DIRECCIONAMIENTO INHERENTE (el operando está implícito en la instrucción, por ejemplo, en la multiplicación uno de los operandos siempre es el acumulador).

Estos modos de direccionamiento suelen ser los más rápidos y compactos debido a que el registro de generación de dirección está codificado en unos pocos bits de la propia instrucción, además no requieren ciclos de bus ya que la la transferencia se realiza internamente en la CPU.

El resto de los modos sirve para localizar un operando en memoria. Para facilitar la explicación de estos modos, se pueden resumir de la siguiente manera:

Deben sumarse cuatro cantidades:

1) dirección de segmento,
2) dirección base,
3) una cantidad índice y
4) un desplazamiento.

La dirección de segmento se almacena en el registro de segmento (DS, ES, SS o CS).. El contenido del registro de segmento se multiplica por 16 antes de utilizarse para obtener la dirección física real (ver la figura 3). El registro de segmentación siempre se usa para referenciar a memoria.

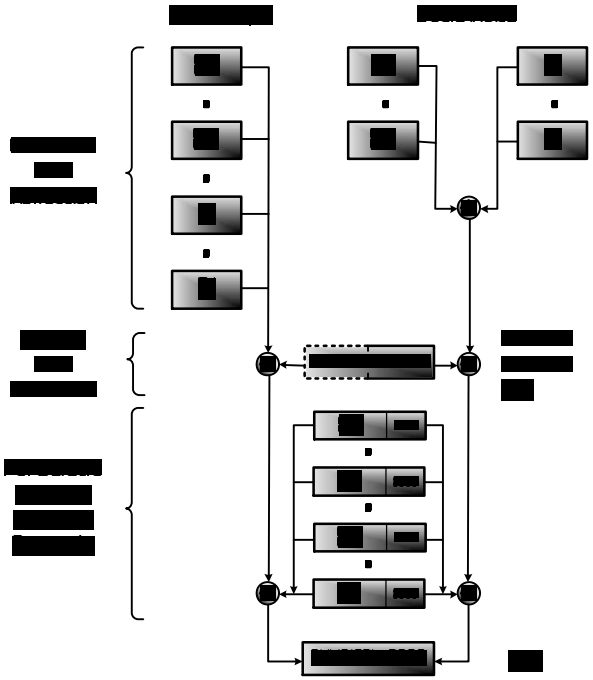


Fig 9.- Cálculo de la Dirección de Memoria.

La base se almacena en el registro base (BX o BP). El índice se almacena en el registro índice (SI o DI). Cualquiera de estas dos cantidades, la suma de las dos o ninguna, pueden utilizarse para calcular la dirección real, pero no pueden sumarse dos bases o dos índices, ver la figura 8. Los registros restantes (AX, CX, DX y SP) no pueden utilizarse para direccionamiento indirecto. El programador puede utilizar tanto la base como el índice para gestionar ciertas cosas, tales como matrices de dos dimensiones, o estructuras internas a otras estructuras, esquemas que se utilizan en las prácticas comunes de programación. La base y el índice son variables dinámicas, ya que están almacenadas en registros de la CPU. Es decir, pueden modificarse fácilmente mientras se ejecuta un programa.

Además del segmento, base e índice, se usa un desplazamiento de 16 bits, 8 bits o 0 bits (sin desplazamiento). Ésta es una cantidad estática que se fija al ensamblar el programa (paso de código fuente a código máquina) y no puede cambiarse durante la ejecución del programa (a menos que el programa se escriba sobre sí mismo, lo que constituye una práctica no aconsejable).

Todo esto genera los 24 modos de direccionamiento a memoria como se puede ver a continuación:

- REGISTRO INDIRECTO: 1) [BX], 2) [DI]. 3) [SI].

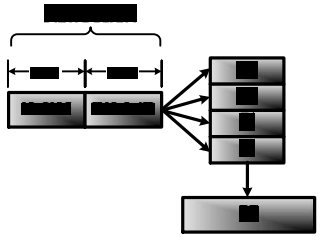


Fig 10.- Modo de Direccionamiento Registro Indirecto.

- BASADO: 4) desp8[BX], 5) desp8[BP], 6) desp16[BX], 7) desp16[BP].

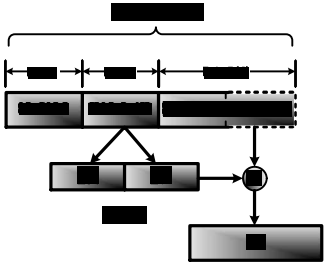


Fig 11.- Modo de Direccionamiento Basado.

- INDEXADO: 8) desp8[SI], 9) desp8[DI], 10) desp16[SI], 11) desp16[DI].

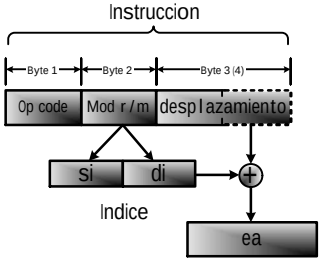


Fig 12.- Modo de Direccionamiento Indexado.

- BASADO-INDEXADO:

-Sin Desplazamiento: 12) [BX+SI], 13) [BX+DI], 14) [BP+SI], 15) [BX+DI].

-Con Desplazamiento: 16) desp8[BX+SI], 17) desp8[BX+DI], 18) DESP8[BP+SI],19) desp8[BX+DI], 20) desp16[BX+SI], 21) desp16[BX+DI], 22) desp16[BP+SI], 23) desp16[BX+DI].

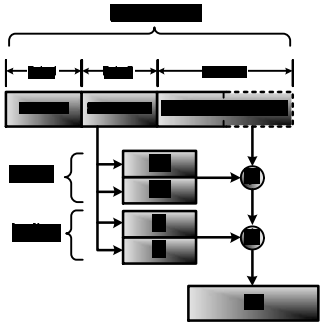


Fig 13.- Modo de Direccionamiento Basado-Indexado.

- DIRECTO: 24) [desp16].

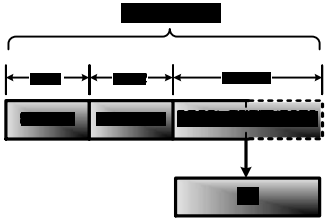


Fig 14.- Modo de Direccionamiento Directo.

Aquí desp8 indica desplazamiento de 8 bits y desp16 indica desplazamiento de 16 bits. Otras combinaciones no están implementadas en la CPU y generarán error al querer ensamblar, por ejemplo, ADD CL,[DX+SI].

El ensamblador genera el tipo de desplazamiento más apropiado (0, 8 ó 16 bits) dependiendo del valor que tenga la constante: si vale cero se utiliza el primer caso, si vale entre -128 y 127 se utiliza el segundo, y en otro caso se utiliza el tercero. Nótese que [BP] sin desplazamiento no existe. Al ensamblar una instrucción como, por ejemplo, MOV AL,[BP], se generará un desplazamiento de 8 bits con valor cero. Esta instrucción ocupa tres bytes, mientras que MOV AL,[SI] ocupa dos, porque no necesita el desplazamiento.

La CPU gasta tiempo calculando una dirección compuesta de varias cantidades. Principalmente esto se debe al hecho de que el cálculo de direcciones está programado en microcódigo (dentro de la CGROM del sistema de control de la unidad de ejecución). En las siguientes versiones (a partir del 80186/80188) estos cálculos están cableados en la máquina y, por lo tanto, cuesta mucho menos tiempo el realizarlos.

Algunos Ejemplos

MOV CX, desp8 [BX]. El operando de la izquierda tiene direccionamiento a registro y el de la derecha es un direccionamiento basado con desplazamiento de 8 bits. Ver la siguiente figura:

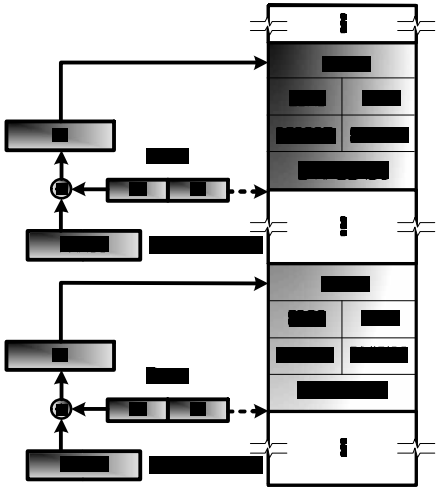


Fig 15.- Estructura de Acceso a Memoria Mediante Direccionamiento Basado.

MOV AL, ES:[BP+SI+4]. En este caso el operando de la izquierda tiene direccionamiento a registro mientras que el de la derecha indica una posición de memoria. Poniendo valores numéricos, supongamos que los valores actuales de los registros sean: ES = 3200h, BP = 200h, SI = 10h. Como se apuntó más arriba la dirección real de memoria será:

ES * 10h + BP + SI + 4 = 3200h * 10h + 200h + 10h + 4h = 32214h, ver la figura 16.

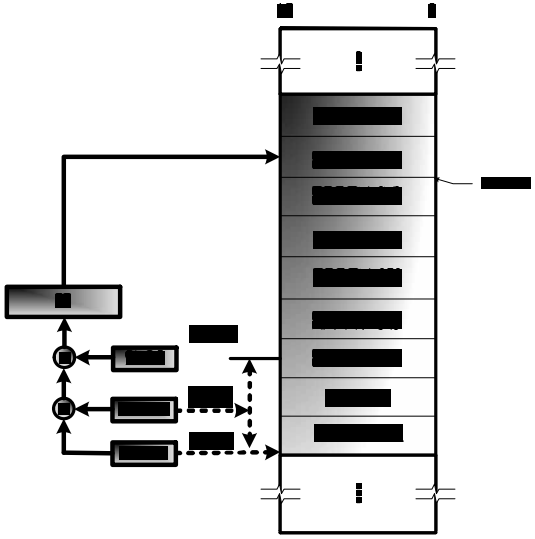


Fig 16.- Acceso a Memoria Mediante Direccionamiento Basado Indexado con desplazamiento de 8 bits.

MOV [SI+220], AH: El operando fuente de la derecha tiene direccionamiento a registro mientras que el de la izquierda direcciona una posición destino de la memoria utilizando el modo Indexado. En la siguiente figura podemos ver dos ejemplos de modo indexado:

- Con SI=6 y Desplazamiento=220h
- Con SI=16 y Desplazamiento=220h

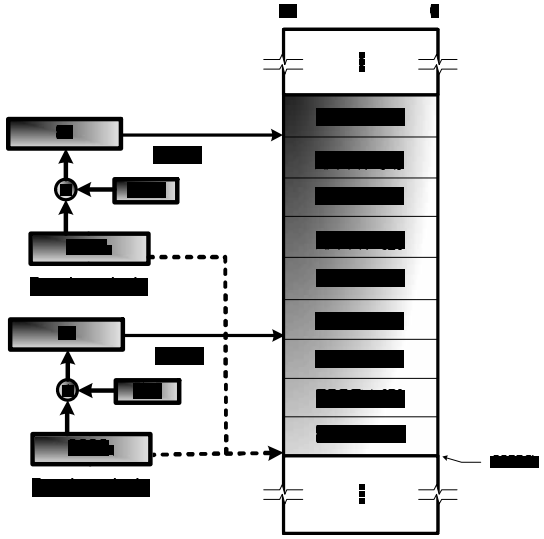


Fig 17.- Acceso a una Array Utilizando Direccionamiento Indexado

simulación

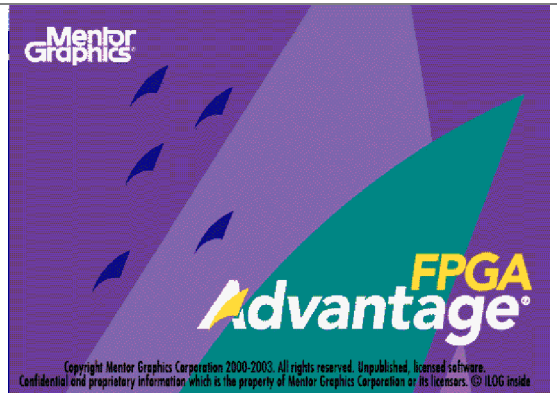
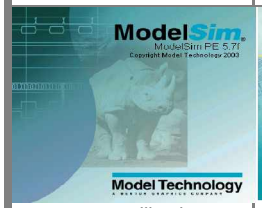
39. SIMULACIÓN

La simulación de los modelos diseñados es esencial durante todas y cada una de las fases del diseño:

En la fase de desarrollo, fueron simuladas de forma intensiva cada una de las unidades de los modelos. A medida que avanzamos en la implementación de cada una de las características de los microprocesadores, vamos analizando las posibles situaciones que pueden aparecer corrigiendo, en su caso, los errores que aparezcan en la lógica y/o añadiendo las señales necesarias para llevarlas a cabo correctamente. Cada instrucción, en el momento de su implementación fue simulada. El soporte para interrupciones, el soporte de cesión de bus (HOLD/HOLDA), la Multiplicación y División con o sin signo cuando provocan desbordamiento, y en general las características más "conflictivas" de los modelos han sido escrupulosamente simuladas incluso en situaciones anómalas como puede ser la aparición de "nesting" en la lógica de interrupciones o apariciones de petición de canal simultáneas.

En la fase de depuración se eliminó lógica recurrente y señales no utilizadas. Se reestructuró el modelo para aislar unidades y eliminar señales no necesarias, se modificaron nombres de señales, se crearon tipos enumerados y se simplificó o rescribió partes del código para hacerlos más entendibles. Después de cada cambio en esta fase, se simulaban los modelos y se comparaban con la simulación anterior (última simulación válida), no deben existir diferencias entre ambas simulaciones, de haberlas, se analizan y si constituyen un error, se descarta la modificación hecha en el modelo. Este es básicamente el "modus operandi" de la fase de depuración.

En la fase de síntesis, se analizó el modelo proporcionado por el sintetizador aplicándole los mismos vectores de entrada (sometiéndolo al mismo programa) que al modelo RTL escrito en VHDL. La comparación de la simulación de ambos modelos, no debe tener más diferencias que las introducidas por los retardos de propagación de la lógica con la que se ha implementado el modelo sintetizable. Estos retardos (consecuencia de la simulación de un modelo más real) no pueden superar los tiempos más restrictivos de los modelos originales y que podemos ver en los Datasheet de los micros 8088/8086 que podemos encontrar en el directorio PDF en el CD-ROM. Por las características de la tecnología utilizada (mucho más avanzada que la tecnología MOS de canal N, HMOS-III con la que se desarrollaron los micros originales), no suelen aparecer problemas en este sentido. De aparecer este tipo de problemas o en el caso de querer mejorar las características de velocidad de los modelos, es necesario acortar el retardo disminuyendo el número de niveles (o puertas lógicas) por los que una señal tiene pasar. Esto se consigue rescribiendo la lógica combinacional que provoca el retardo introduciendo estructuras más paralelas o introduciendo registros "pegar tijeretazo" (ver la figura 73) en el camino que provoca el retardo, si se puede claro está.

ADVANTAGE FPGA		
 Programa utilizado como entorno de diseño VHDL en el desarrollo del proyecto.		
MODELSIM	LEONARDO SPECTRUM	PRECISION SYNTHESIS
 Programa Utilizado en las Simulaciones	Programa Utilizado para la Síntesis	Programa Utilizado para la Síntesis

Tab 1.- Herramientas Utilizadas en la Síntesis y Simulación del Proyecto.

El programa utilizado para las simulaciones del modelo, ha sido "ModelSim SE/PE 5" versiones. 5.5e/5.7f integrado dentro del paquete de diseño HDL de Mentor Graphics ®, "FPGA Advantage" versiones 5.2/6.2 que además de ModelSim incluye las herramientas de síntesis "Leonardo Spectrum v2003b.35" y "Precision Synthesis v2003b.41" como podemos apreciar en la tabla 1. En la síntesis, además de los programas indicados, también se hicieron pruebas con las herramientas de síntesis de Xilinx "Xilinx WebPack" versiones 3.3 y 4.2.

Todas las simulaciones que veamos a partir de esta página, están realizadas con "ModelSim 5.5e", los archivos de las formas de onda *.wlf, así como las macros programadas *.do que podemos encontrar en el CD, también son para la versión 5.5e de ModelSim.

Dada la complejidad de los modelos de los microprocesadores 8088/8086 desarrollados en este proyecto, es imposible programar los vectores de test uno a uno e introducirlos de manera secuencial para recoger y analizar la respuesta del micro. Para cada Instrucción habría que realizar cientos de vectores para las muchas combinaciones o situaciones en las que se puede encontrar en microprocesador. Durante la ejecución de cualquier instrucción, puede aparecer una petición de bus, o una interrupción, o ambas a la vez, o estar la cola de instrucciones vacía, o... ...un sinfín de variables.

Para simular los modelos, hay que recurrir a otra estrategia, la estrategia de autogeneración de vectores de test a través de un sistema que interactúe con el microprocesador. El sistema que genera los vectores tiene que ser programable o parametrizable por nosotros, de manera que podamos indicarle las situaciones en las que queremos que el modelo se encuentre. El análisis meticuloso de la respuesta del microprocesador ante esas situaciones, nos indicará si el funcionamiento del microprocesador es correcto o no. En definitiva, lo que buscamos es un modelo que emule las situaciones en las que se puede encontrar el microprocesador en un sistema real, es decir, un modelo de un sistema real. En las siguientes páginas se explica detalladamente los ficheros utilizados para la simulación.

40. MODELADO DEL ENTORNO

Como se habrán dado cuenta, en el árbol jerárquico de la figura 16, quedan por analizar las "raíces". Hasta ahora, en los capítulos anteriores de la descripción particular, hemos analizado todas las unidades o ramas del árbol, pero en la esquina inferior derecha de la figura 16, se puede observar unos nudos que todavía no hemos explicado, se trata de la simulación del proyecto. En la siguiente figura, podemos ver este detalle.

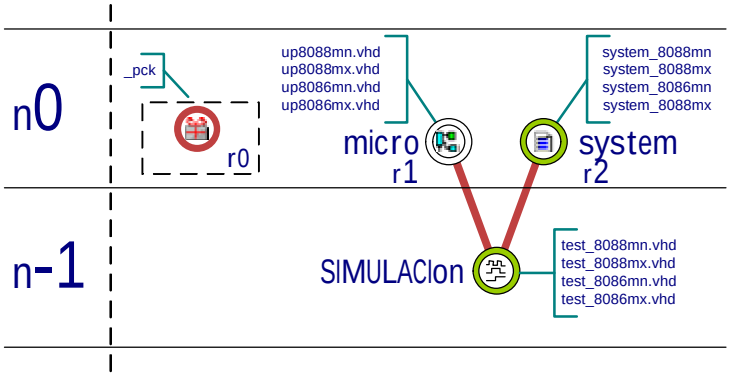


Fig 1.- Detalle de la Jerarquía de la Simulación de los Modelos

Bajo tierra, en las raíces (nivel -1, n-1), podemos encontrar los archivos del banco de test (Test Bench, _tb). Estos archivos definen la conexión entre el microprocesador "micro" (n0/r1) y el sistema que genera los vectores de test "system" (n0/r2). Existen cuatro bancos de test (uno por cada modelo en cada modo de funcionamiento).

FICHEROS	FUNCIÓN
system_8088mn test_8088mn	Ficheros utilizados en la simulación del modelo 8088 en modo mínimo
system_8088mx test_8088mx	Ficheros utilizados en la simulación del modelo 8088 en modo máximo
system_8086mn test_8086mn	Ficheros utilizados en la simulación del modelo 8086 en modo mínimo
system_8086mx test_8086mx	Ficheros utilizados en la simulación del modelo 8086 en modo máximo

Tab 2.- Función de los Ficheros de la Librería de Simulación, "up808x_tb".

Aunque se multiplique por 4 el trabajo de la simulación, es necesario establecer estos 4 bancos de test para comprobar el correcto funcionamiento de los modelos 8088/8086 en los modos Máximo y Mínimo. Como es lógico, existen diferencias entre los sistemas reales configurados para un 8088 en modo máximo y para ese mismo micro en modo mínimo, al igual que ocurre con el 8086, de manera que los ficheros de generación de vectores, también son distintos. Los bancos de test, son sistemas cerrados, es decir, no tienen ni entradas ni salidas (carece de entity), simplemente conecta las entradas del micro a las salidas del vector de test (salidas de system) y las salidas del micro a las entradas de verificación del sistema generador de vectores que analiza la respuesta y en su caso genera vectores en tiempo de ejecución, es decir, lanza otra secuencia de vectores en función de la respuesta del micro (interactuación).

Se puede observar en la figura 1 que tanto los 4 ficheros que definen los bancos de test (test_*.*) como los que modelan el sistema al que se conectan los micros (system_*.*), forman parte de una librería que no habíamos tratado hasta ahora, la librería de simulación "up808x_tb", (el sufijo _tb significa banco de test) que podemos considerarla como una librería común a ambos micros (contiene los archivos necesarios para simular los dos micros, ver la tabla 2). Se trata de una librería NO SINTETIZABLE, lo que significa que utiliza funciones que no pueden ser sintetizadas, funciones como las de retardos en señales, o las de tratamiento de ficheros externos. Los archivos en la librería de simulación tienen en común dos cosas, no son sintetizables, y se utilizan sólo y exclusivamente para la generación de vectores de test.

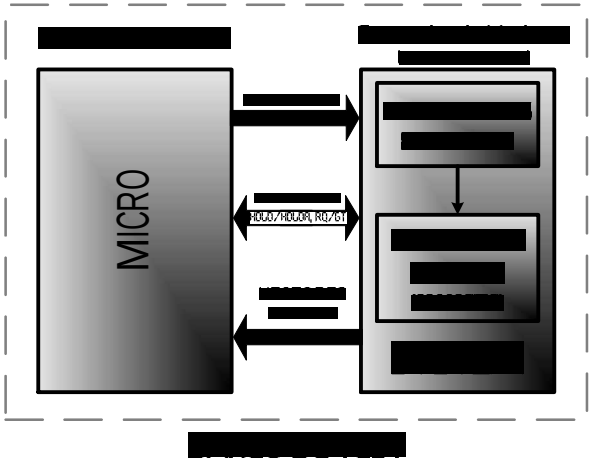


Fig 2.- Diagrama de Bloques de los Bancos de Test.

40.1 BANCOS DE TEST

El las figuras 2 y 3, podemos ver los bancos de test utilizados para la simulación de los modelos. Los esquemas de la figura 3 son muy explícitos en cuanto al conexionado, mientras que la figura 2 muestra la lógica de comunicación entre los dos elementos que forman el banco de test. En un test, siempre hay un dispositivo que se somete al test (como es lógico), en este caso es el microprocesador, y otro elemento que realiza las preguntas, en ese caso el sistema. Como se comentó antes, este es un caso especial de banco de test debido principalmente a que el que hace las preguntas, no tiene las respuestas. Las respuestas, como las preguntas, están en función del programa que introducimos. System solamente se encarga de establecer una comunicación, de enviar la secuencia de vectores (programa) y de memorizar las respuestas externas. Somos nosotros, los que en función del programa y el análisis de las respuestas internas (o valor de los registros del modelo software del microprocesador) y externas (o valores almacenados en memoria o dispositivos de entrada y salida), establecemos si el funcionamiento del modelo es correcto o no.

En la figura 3, llama la atención a simple vista el gran número de constantes GENERIC de configuración del SYSTEM. En la siguiente tabla podemos ver una breve descripción:

GENERIC	FUCIÓN
log_filename	Nombre del fichero histórico que contiene las operaciones realizadas con la memoria
download_filename	Contiene el nombre del archivo que va a ser volcado en memoria (el programa)
size, addr_width, width	Configura la Memoria
x_Tw	Indica el número de estados de espera que se van a generar.
TRYLCL,TCHRYX	Peores tiempos de setup y hold de la señal READY
x_Tres	Indica el ancho del pulso de RESET
TDVCL,TCLDX	Peores tiempod de RESET.
x_Thold	Ancho de pulso de la señal HOLD
THVCH	Tiempo de Setup de HOLD
TCLCH, TCHCL, TCH1CH2, TCL2CH1	Tiempos para la generación de CK
pointer_INTR	Indica el tipo de interrupción que aparecerá en los ciclos intr_ack

Tab 3.- Tabla de GENERICS del Generador de Vectores.

40.2 BANCOS DE TEST PARA LA SIMULACIÓN

Fig 3.- a) Banco de Test para el 8088 en Modo Mínimo, "test_8088mn.vhd".
b) Banco de Test para el 8088 en Modo Máximo, "test_8088mx.vhd".
a) Banco de Test para el 8086 en Modo Mínimo, "test_8086mn.vhd".
a) Banco de Test para el 8086 en Modo Máximo, "test_8086mx.vhd".

Package List

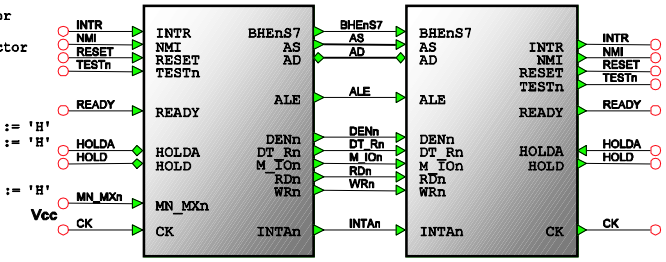
```
LIBRARY IEEE;
USE IEEE.Std_Logic_1164.ALL;
LIBRARY up808x;
USE up808x.pack808x.ALL;
USE ieee.Std_Logic_arith.ALL;
USE ieee.Std_Logic_textio.ALL;
USE ieee.Std_Logic_unsigned.ALL;
LIBRARY std;
USE std.textio.ALL;
```

```
log_filename       = ".\memoria.log"    ( STRING       )
download_filename  = ".\prog27.hex"     ( STRING       )
size               = 1048576            ( INTEGER      )
addr_width         = 20                 ( INTEGER      )
width              = 8                  ( INTEGER      )
x_Tw               = 0                  ( INTEGER      )
TRYLCL             = -8 ns              ( TIME         )
TCHRYX             = 20 ns              ( TIME         )
x_Tres             = 7                  ( INTEGER      )
TDVCL              = 20 ns              ( TIME         )
TCLDX              = 10 ns              ( TIME         )
x_Thold            = 7                  ( INTEGER      )
THVCH              = 20 ns              ( TIME         )
TCLCH              = 60 ns              ( Time         )
TCHCL              = 40 ns              ( Time         )
TCH1CH2            = 0 ns               ( Time         )
TCL2CL1            = 0 ns               ( Time         )
pointer_INTR       = X"41"              ( Byte_Vector  )
```

a)

Declarations
Ports:
Diagram Signals:

```
SIGNAL AD     : Word_Vector
SIGNAL ALE    : Std_Logic
SIGNAL AS     : Nibble_Vector
SIGNAL BHEnS7 : Std_Logic
SIGNAL CK     : Std_Logic
SIGNAL DENn   : Std_Logic
SIGNAL DT_Rn  : Std_Logic
SIGNAL HOLD   : Std_Logic := 'H'
SIGNAL HOLDA  : Std_Logic := 'H'
SIGNAL INTAn  : Std_Logic
SIGNAL INTR   : Std_Logic
SIGNAL MN_MXn : Std_Logic := 'H'
SIGNAL M_IOn  : Std_Logic
SIGNAL NMI    : Std_Logic
SIGNAL RDn    : Std_Logic
SIGNAL READY  : Std_Logic
SIGNAL RESET  : Std_Logic
SIGNAL TESTn  : Std_Logic
SIGNAL WRn    : Std_Logic
```



up8086
up8086mn
micro

up808x_tb
system_8086mn
system

Package List

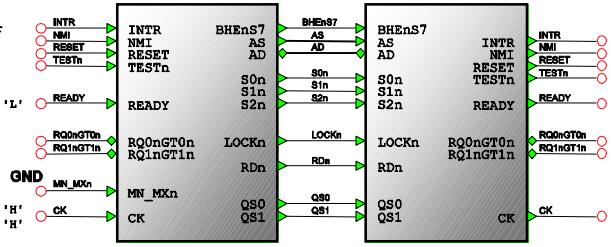
```
LIBRARY IEEE;
USE IEEE.Std_Logic_1164.ALL;
LIBRARY up808x;
USE up808x.pack808x.ALL;
USE ieee.Std_Logic_arith.ALL;
USE ieee.Std_Logic_textio.ALL;
USE ieee.Std_Logic_unsigned.ALL;
LIBRARY std;
USE std.textio.ALL;
```

```
log_filename       = ".\memoria.log"    ( STRING       )
download_filename  = ".\prog27.hex"     ( STRING       )
size               = 1048576            ( INTEGER      )
addr_width         = 20                 ( INTEGER      )
width              = 8                  ( INTEGER      )
x_Tw               = 0                  ( INTEGER      )
TRYLCL             = -8 ns              ( TIME         )
TCHRYX             = 20 ns              ( TIME         )
x_Tres             = 7                  ( INTEGER      )
TDVCL              = 20 ns              ( TIME         )
TCLDX              = 10 ns              ( TIME         )
x_Thold            = 7                  ( INTEGER      )
THVCH              = 20 ns              ( TIME         )
TCLCH              = 60 ns              ( Time         )
TCHCL              = 40 ns              ( Time         )
TCH1CH2            = 0 ns               ( Time         )
TCL2CL1            = 0 ns               ( Time         )
pointer_INTR       = X"41"              ( Byte_Vector  )
```

b)

Declarations
Ports:
Diagram Signals:

```
SIGNAL AD       : Word_Vector
SIGNAL AS       : Nibble_Vector
SIGNAL BHEnS7   : Std_Logic
SIGNAL CK       : Std_Logic
SIGNAL INTR     : Std_Logic
SIGNAL LOCKn    : Std_Logic
SIGNAL MN_MXn   : Std_Logic := 'L'
SIGNAL NMI      : Std_Logic
SIGNAL QS0      : Std_Logic
SIGNAL QS1      : Std_Logic
SIGNAL RDn      : Std_Logic
SIGNAL READY    : Std_Logic
SIGNAL RESET    : Std_Logic
SIGNAL RQ0nGT0n : Std_Logic := 'H'
SIGNAL RQ1nGT1n : Std_Logic := 'H'
SIGNAL S0n      : Std_Logic
SIGNAL S1n      : Std_Logic
SIGNAL S2n      : Std_Logic
SIGNAL TESTn    : Std_Logic
```



up8086
up8086mx
micro

up808x_tb
system_8086mx
system

Package List

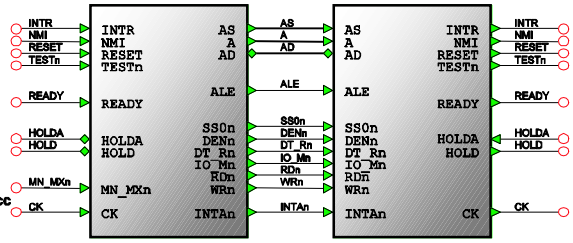
```
LIBRARY IEEE;
USE IEEE.Std_Logic_1164.ALL;
LIBRARY up808x;
USE up808x.pack808x.ALL;
USE ieee.Std_Logic_arith.ALL;
USE ieee.Std_Logic_textio.ALL;
USE ieee.Std_Logic_unsigned.ALL;
LIBRARY std;
USE std.textio.ALL;
```

```
log_filename       = ".\memoria.log"    ( STRING       )
download_filename  = ".\prog27.hex"     ( STRING       )
size               = 1048576            ( INTEGER      )
addr_width         = 20                 ( INTEGER      )
width              = 8                  ( INTEGER      )
x_Tw               = 0                  ( INTEGER      )
TRYLCL             = -8 ns              ( TIME         )
TCHRYX             = 20 ns              ( TIME         )
x_Tres             = 7                  ( INTEGER      )
TDVCL              = 20 ns              ( TIME         )
TCLDX              = 10 ns              ( TIME         )
x_Thold            = 7                  ( INTEGER      )
THVCH              = 20 ns              ( TIME         )
TCLCH              = 60 ns              ( Time         )
TCHCL              = 40 ns              ( Time         )
TCH1CH2            = 0 ns               ( Time         )
TCL2CL1            = 0 ns               ( Time         )
pointer_INTR       = X"41"              ( Byte_Vector  )
```

c)

Declarations
Ports:
Pre User:
Diagram Signals:

```
SIGNAL A      : Byte_Vector
SIGNAL AD     : Byte_Vector
SIGNAL ALE    : Std_Logic
SIGNAL AS     : Nibble_Vector
SIGNAL CK     : Std_Logic
SIGNAL DENn   : Std_Logic
SIGNAL DT_Rn  : Std_Logic
SIGNAL HOLD   : Std_Logic := 'H'
SIGNAL HOLDA  : Std_Logic := 'H'
SIGNAL INTAn  : Std_Logic
SIGNAL INTR   : Std_Logic
SIGNAL IO_Mn  : Std_Logic
SIGNAL MN_MXn : Std_Logic := 'H'
SIGNAL NMI    : Std_Logic
SIGNAL RDn    : Std_Logic
SIGNAL READY  : Std_Logic
SIGNAL RESET  : Std_Logic
SIGNAL SS0n   : Std_Logic
SIGNAL TESTn  : Std_Logic
SIGNAL WRn    : Std_Logic
Post User:
```



up8088
up8088mn
micro

up808x_tb
system_8088mn
system

Package List

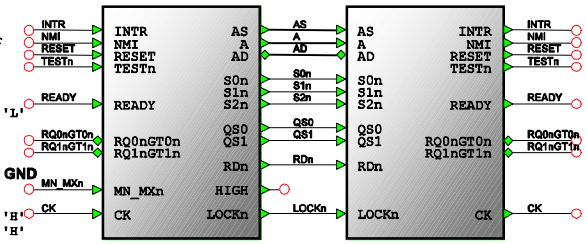
```
LIBRARY IEEE;
USE IEEE.Std_Logic_1164.ALL;
LIBRARY up808x;
USE up808x.pack808x.ALL;
USE ieee.Std_Logic_arith.ALL;
USE ieee.Std_Logic_textio.ALL;
USE ieee.Std_Logic_unsigned.ALL;
LIBRARY std;
USE std.textio.ALL;
```

```
log_filename       = ".\memoria.log"    ( STRING       )
download_filename  = ".\prog27.hex"     ( STRING       )
size               = 1048576            ( INTEGER      )
addr_width         = 20                 ( INTEGER      )
width              = 8                  ( INTEGER      )
x_Tw               = 0                  ( INTEGER      )
TRYLCL             = -8 ns              ( TIME         )
TCHRYX             = 20 ns              ( TIME         )
x_Tres             = 7                  ( INTEGER      )
TDVCL              = 20 ns              ( TIME         )
TCLDX              = 10 ns              ( TIME         )
x_Thold            = 7                  ( INTEGER      )
THVCH              = 20 ns              ( TIME         )
TCLCH              = 60 ns              ( Time         )
TCHCL              = 40 ns              ( Time         )
TCH1CH2            = 0 ns               ( Time         )
TCL2CL1            = 0 ns               ( Time         )
pointer_INTR       = X"41"              ( Byte_Vector  )
```

d)

Declarations
Ports:
Pre User:
Diagram Signals:

```
SIGNAL A      : Byte_Vector
SIGNAL AD     : Byte_Vector
SIGNAL AS     : Nibble_Vector
SIGNAL CK     : Std_Logic
SIGNAL HIGH   : Std_Logic
SIGNAL INTR   : Std_Logic
SIGNAL LOCKn  : Std_Logic
SIGNAL MN_MXn : Std_Logic := 'L'
SIGNAL NMI    : Std_Logic
SIGNAL QS0    : Std_Logic
SIGNAL QS1    : Std_Logic
SIGNAL RDn    : Std_Logic
SIGNAL READY  : Std_Logic
SIGNAL RESET  : Std_Logic
SIGNAL RQ0nGT0n : Std_Logic := 'H'
SIGNAL RQ1nGT1n : Std_Logic := 'H'
SIGNAL S0n    : Std_Logic
SIGNAL S1n    : Std_Logic
SIGNAL S2n    : Std_Logic
SIGNAL TESTn  : Std_Logic
Post User:
```



up8088
up8088mx
micro

up808x_tb
system_8088mx
system

41. GENERADOR DE VECTORES DE TEST, SYSTEM (n0/r2)

Este nivel como se comentó anteriormente, genera los vectores de test para los modelos diseñados. Como se puede apreciar en la figura 1, pertenece a la librería de simulación (up808x_tb) y están definidos en los archivos siguientes.

FICHEROS	FUNCIÓN
system_8088mn.vhd	Contiene la definición de un sistema mínimo para el 8088.
system_8088mx.vhd	Contiene la definición de un sistema máximo para el micro 8088.
system_8086mn.vhd	Contiene la definición de un sistema mínimo para el microprocesador 8088.
system_8086mx.vhd	Contiene la definición de un sistema máximo para la CPU 8086.

Tab 4.- Ficheros de Declaración del Sistema para la Simulación.

Estos ficheros, tratan de emular el sistema real donde se implantan los microprocesadores. En las siguientes figuras, podemos ver una aproximación de estos sistemas para los microprocesadores en modo máximo y modo mínimo. En las siguientes páginas se explicará cada proceso (representado por un bloque en las figuras 4a y 4b) que compone el fichero de sistema detalladamente y las diferencias entre los distintos sistemas.

Las diferencias más importantes son las existentes entre los sistemas en modo máximo y los sistemas en modo mínimo. Las diferencias entre los sistemas para el mismo modo de funcionamiento en los micros 8088 y 8086 se limitan básicamente a la longitud del bus de datos y la estructura de la memoria, por lo que los únicos bloques que cambian son los de "desacoplo de los buses" y el emulador de la memoria.

Existen elementos que son comunes a todos los sistemas como el "generador de reloj", el "generador de reset", el "generador de estados de espera" el "generador de test" el "generador de interrupciones", el "mapa de memoria" y los "periféricos". Otros elementos sufren unas pequeñas modificaciones (como un cambio del número de bits en una señal) en función del tipo de procesador (8088 u 8086); el "desacoplo de los buses" y el "cargador de memoria" sufren pequeños cambios. Finalmente, existen otros elementos que son propios del tipo de funcionamiento del sistema como el "generador de DMA" (o generador de petición del bus local) que es completamente distinto en el modo máximo al utilizado en el modo mínimo. El "controlador de bus", también es propio del modo máximo y genera las señales de control que desaparecen al hacer funcionar el microprocesador en modo máximo. Ver la figura 4.

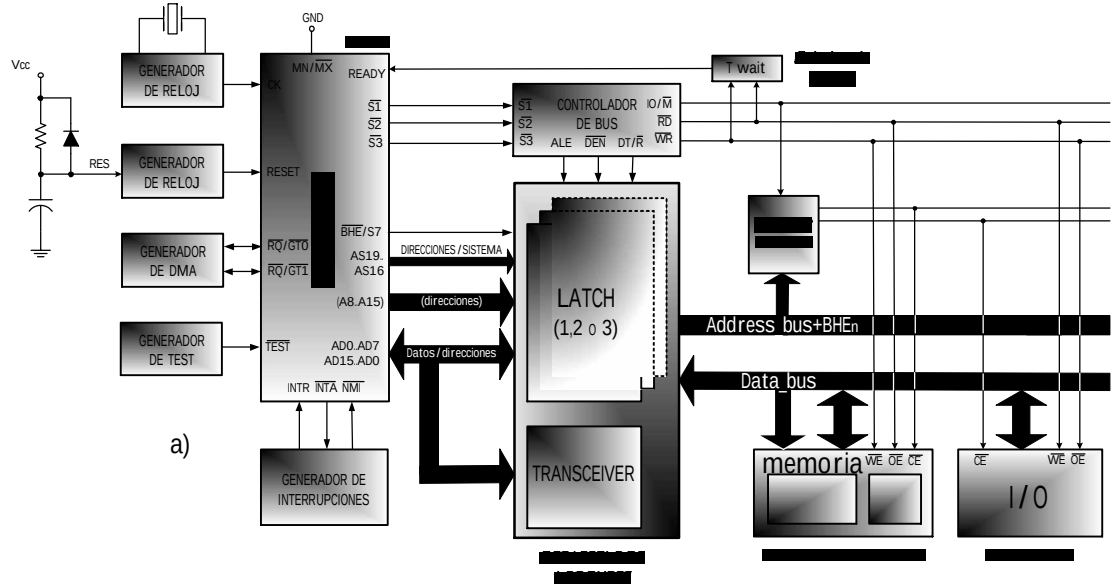
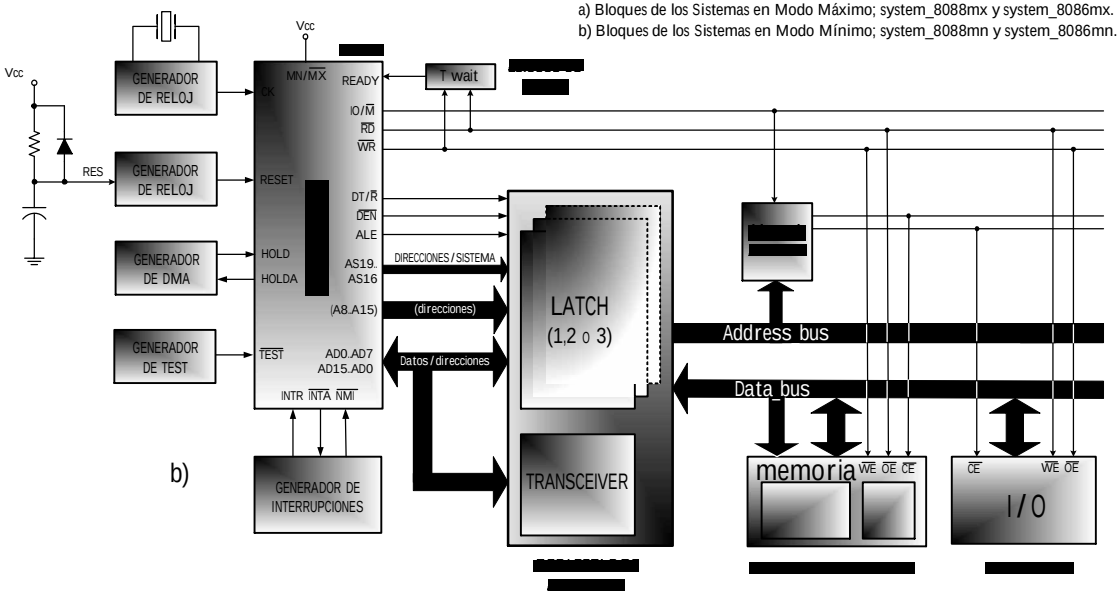


Fig 4.-
a) Bloques de los Sistemas en Modo Máximo; system_8088mx y system_8086mx.
b) Bloques de los Sistemas en Modo Mínimo; system_8088mn y system_8086mn.



41.1 GENERADOR DE RELOJ

Este es un sencillo proceso que se encarga de generar el reloj que el microprocesador y el sistema utilizarán como base de sincronización. Como se puede observar en la tabla 3, la señal de reloj se define con 4 constantes de configuración (GENERICS), como se puede ver en la siguiente figura

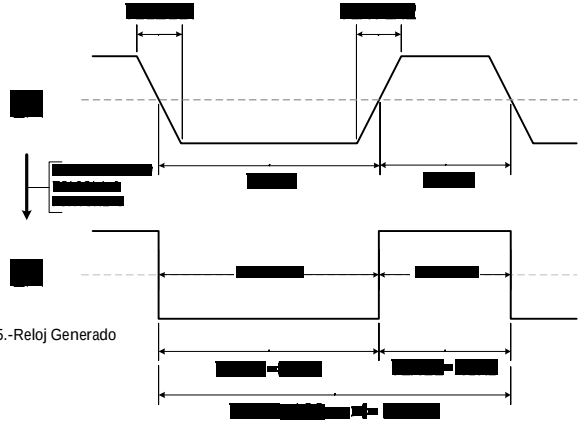


Fig 5.-Reloj Generado

Los tiempos están sacados del Datasheet del 8088 que podemos encontrar en el CD del proyecto. La siguiente tabla Muestra el significado de cada constante de configuración:

GENERIC	SIGNIFICADO
TCLCH	Tiempo en estado bajo de CK
TCHCL	Tiempo en estado alto de CK
TCH1CH2	Tiempo del flanco de subida
TCH2CH1	Tiempo del flanco de bajada

Tab 5.-Constantes GENERIC utilizadas. en el Generador de Reloj

Para simplificar la simulación he asumido que los tiempos de subida y de bajada son despreciables. TCLCL (Periodo del Reloj) es la suma de TCLCH+TCHC. En los archivos de sistema, TCLCL está definido con una constante:

```
--Ciclo de la señal de CK, se han ignorado los tiempos
--de falling y rising de la señal
    CONSTANT TCLCL : Time:=TCLCH+TCHCL;  -- TCLCL=1/Fck
```

El sencillo proceso definido en los archivos system que genera el reloj, lo pos ver en el siguiente código. Se puede observar que este proceso también tiene un contador de ciclos definido en la señal n_ciclo (de tipo entero) que es utilizado en el registro histórico de acciones en la memoria que estudiaremos un poco más adelante.

```
--=================================================--
--|                -GENERADOR DE RELOJ-             |--
--|                                                 |--
--|Genera el Reloj de todo el sistema incluída la   |--
--|CPU. La señal ck_i es necesaria, ya que CK al ser|--
--|un buffer de salida no puede ser leído directa-  |--
--|mente desde dentro. El ciclo de trabajo del reloj|--
--|no es 1/2 sino 2/3                               |--
--|-------------------------------------------------|--
--|                                                 |--
    CK <= ck_i;
    CK_gen : PROCESS
    BEGIN
        ck_i <= '0', '1' AFTER TCLCH;
        WAIT FOR TCLCL;
        --Contador del número de ciclos
        n_ciclo <= n_ciclo+1;
    END PROCESS CK_gen;
--|                                                 |--
--|-------------------------------------------------|--
--|           FINAL DEL GENERADOR DE RELOJ          |--
--=================================================--
```

41.2 GENERADOR DE RESET

Este proceso, también es idéntico en todos los sistemas generadores de vectores (n0/r2). se encarga de generar la señal de reset cuando se lo indicamos. Las características de sincronización de la señal de RESET las podemos ver en la figura 6.

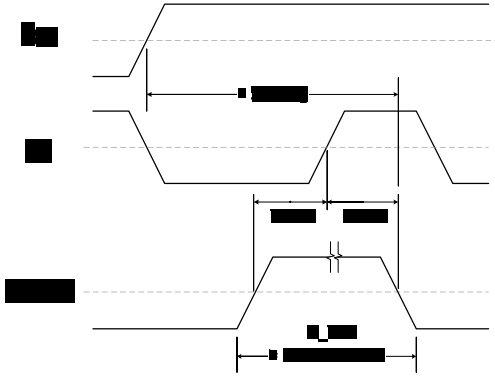


Fig 6.-Temporización de RESET..

GENERIC	SIGNIFICADO
TDVCL	Tiempo de Setup de RESET
TCLDX	Tiempo en HOLD de RESET
x_Tres	Ancho del pulso de RESET

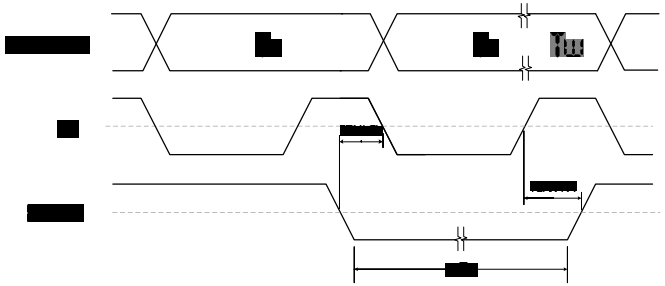
Las peticiones de generación de RESET, las programamos produciendo un flanco de subida en la señal "do_RESET", este flanco despierta el proceso "RESET_gen" que se encarga de generar un pulso de RESET con el ancho que le indicamos en la constante GENERIC "x_Tres". En la tabla de la figura 6 podemos ver el significado de las GENERICS utilizadas en este proceso y en las líneas siguientes el código.

```
--=================================================--
--|      -GENERADOR DE lA sECUENCiA dE RESET-       |--
--|                                                 |--
--|El proceso principal mantiene RESET='1' durante |--
--|x_Tres ciclos. Las asignaciones concurrentes,    |--
--|despiertan el proceso en los tiempos especifica-|--
--|dos. A los 4us se emula un RESET no válido (los |--
--|RESET's menores de 4 ciclos no nos reconocidos   |--
--|por la CPU).                                     |--
--|-------------------------------------------------|--
--|                                                 |--

--|TiEMPOS eN lOS QuE aPARECE uN RESET              |--
--Lanza a los 10ns y
--Lanza a los 22us
do_RESET <= '1' AFTER 10 ns, '0' AFTER 10 ns + TCLCL,
            '1' AFTER  22 us, '0' AFTER  22 us + TCLCL;

--|PrOCESO PrAL                                     |--
    RESET_gen: PROCESS
    VARIABLE i:integer;

    BEGIN
        --Secuencia de RESET
        WAIT UNTIL rising_edge(do_reset);
        WAIT UNTIL falling_edge(ck_i);

        i:=0;
        WHILE i<=x_Tres LOOP
        --Inserta x estados de RESET (Tres)
        RESET <= '1' AFTER TCLCH-TDVCL;
        i:=i+1;
        WAIT UNTIL rising_edge(ck_i);
        END LOOP;
        RESET <= '0' AFTER TCLDX;

--Éstas líneas hacen lo mismo que las anteriores
--RESET <='1' AFTER TCLCH-TDVCL, '0' AFTER TCLCH-
--TDVCL+x_Tres*TCLCL+TCLDX;
--WAIT FOR TCLCH-TDVCL+x_Tres*TCLCL+TCLDX;
    END PROCESS RESET_gen;
--|                                                 |--
--|-------------------------------------------------|--
--|         FINAL DE LA SECUENCIA DE RESET          |--
--=================================================--
```

Si queremos que durante la ejecución de un programa aparezca un RESET, podemos programarlo en esta sección. Por defecto los sistemas lanzan un RESET a los 10 ns (RESET inicial) y otro a los 22 us. El ancho de RESET predeterminado es de 7 ciclos, x_Tres=7.

41.3 GENERADOR DE ESTADOS DE ESPERA

También es un proceso que aparece en todos los sistemas, en este caso se encarga de generar el número de estados de espera indicados en la constante GENERIC "x_Tw". En la figura 7, podemos ver la sincronización de la señal READY.



Tab 7.-Temporización de la señal READY.

En la siguiente tabla, podemos ver el significado de los tiempos que podemos configurar desde las GENERICS del sistema.

GENERIC	SIGNIFICADO
TRYHCH	Tiempo de Mantenimiento de READY
TRYLCL	Tiempo de READY inactivo hasta CK
x_Tw	Ancho del pulso de READY

Tab 6.-Constantes GENERIC utilizadas. en el Generador de Ciclos de Espera.

Los estados de espera son generados por memorias o dispositivos de entrada/salida (periféricos) lentos, es decir, por aquellos dispositivos que no pueden realizar la transferencia de un dato en el tiempo mínimo de un ciclo de bus (4 estados). Normalmente, son estos dispositivos "lentos" los que generan la señal READY a través de las entradas asíncronas RDY del generador de reloj 8284 de Intel. Este generador de reloj se encarga de generar una señal READY correctamente sincronizada, cumpliendo los tiempos de "setup" y "hold" tal y como podemos apreciar en la figura 7.

La GENERIC x_Tw, hace que la memoria sea un dispositivo lento cuando su valor es distinto de 0. Todas las transferencias con la memoria generarán x_Tw estados de bus adicionales (o de espera, Tw).

En el siguiente código, podemos que este proceso funciona de una forma muy similar al proceso generador de RESET estudiado anteriormente, la única diferencia es que entramos en el proceso "READY_gen" cuando hay una operación de lectura o escritura en la memoria. momento en el que el flanco de subida de "do_READY" despierta el proceso que genera un pulso de READY inactivo durante "x_Tw" ciclos.

```
--=================================================--
--|       -GENERADOR DE ESTADOS DE ESPERA-         |--
--|                                                |--
--|El siguiente proceso se encarga de generar x_Tw |--
--|estados de espera cuando se detecta una opera-  |--
--|ción de lectura o escritura en la RAM del siste-|--
--|ma.                                             |--
--|------------------------------------------------|--
--|                                                |--

--|aSIGNACiONES cONCURRENTES                       |--
    do_READY <= '1' WHEN (RDn='0' OR WRn='0')
                    AND RAM_CE='1' ELSE '0';

--|PrOCESO PrAL                                    |--
    READY_gen : PROCESS
    VARIABLE i : Integer;
    BEGIN
        WAIT UNTIL rising_edge(do_READY);
        WAIT UNTIL rising_edge(ck_i);
    --Inserta los estados de espera indicados en x_Tw
        i := 0;
        WHILE i < x_Tw LOOP
        READY <= '0';       --AFTER TCHCL-TRYLCL;
        i:=i+1;
        WAIT UNTIL rising_edge(ck_i);
        END LOOP;
        READY <= '1' AFTER TCHRYX;

    --Éstas líneas hacen lo mismo que las anteriores
        --READY <='0' AFTER TCHCL-TRYLCL, '1' AFTER
        --TCHCL-TRYLCL+x_Tw*TCLCL+TCHRYX;
        --WAIT FOR TCHCL-TRYLCL+x_Tw*TCLCL+TCHRYX
    END PROCESS READY_gen;
--|                                               |--
--|- ---------------------------------------------|--
--|   FINAL DEL GENERADOR DE ESTADOS DE ESPETA    |--
--=================================================--
```

41.4 DESACOPLO DE LOS BUSES

Los buses de direcciones, datos y sistema, están acoplados (o multiplexados) en los microprocesadores, es decir, comparten los mismos pines. La ventaja de la reducción de terminales del microprocesador, repercute en su lado negativo en la necesidad de desacoplar (o demultiplexar) los buses para acceder a dispositivos que no admiten estos buses "acoplados" directamente. Para ello, se necesitan una serie de registros de enclavamiento y un buffer bidireccional. Como ventaja, conseguimos aislar los terminales del microprocesador y mejoramos el fan-out de las señales.

41.4.1 DESACOPLO DE LOS BUSES DEL 8088

El 8088 tiene un bus de datos de 8 bits que está multiplexado en el tiempo en los terminales AD(7:0) del microprocesador 8088. Los archivos de sistema "syatem_8088mn" y "syatem_8088mx" implementan la configuración que podemos ver en la figura 8 para desacoplar los buses.

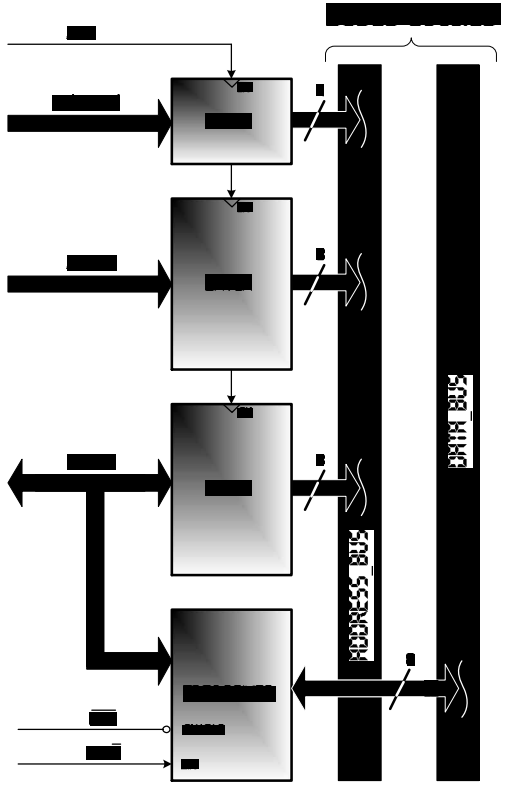


Fig 8.-Desacoplo total de los buses del 8088.

A la izquierda se pueden observar los buses acoplados en el tiempo, mientras que a la derecha nos encontramos con los buses locales totalmente desacoplados. Los buses locales son los que serán utilizados por el resto de las unidades para comunicarse con el microprocesador y entre ellas si otro maestro de canal adquiere el control sobre estos.

En la parte baja de esta columna, podemos ver el código del archivo de sistema "system_8088mn.vhd" que desacopla los buses.

Son una serie de asignaciones concurrentes, las señales "latched_", enclavan en el flanco de bajada de la señal ALE las direcciones que posteriormente se encadenan para formar el bus de direcciones "address_bus". De igual manera se encadenan todos los bits del sistema en el bus de sistema "system_bus". En cuanto al bus bidireccional triestado de datos y direcciones AD(7:0), la señal DENn activa los transceptores, mientras que la señal de control DT/Rn controla la dirección permitiendo la salida de datos desde el microprocesador a la memoria cuando DT/Rn='1" y el envío de datos desde la memoria o un dispositivo periférico mapeado en memoria cuando DT/Rn='0'.

La lógica de desacoplo en el 8088 en modo máximo es idéntica, con la salvedad de que las señales de control ALE, DENn y DT/Rn son generadas por el controlador de bus en lugar de la CPU.

```
--==============================================================================--
--|           -BUFFERS, LATCHEDS i TRANSCIVERS (DESACOPLO EN EL 8088)-        |--
--|                                                                           |--
--|Para Demultiplexar y acoplar las señales de la CPU en el sistema mínimo,   |--
--|son necesarios una serie de registros transparentes o latcheds (74LS373)   |--
--|para demultiplexar los buses de ADS, registros bidireccionales (74LS245)   |--
--|para el bus de datos y transceptores 74LS244 para amplificar y aislar las  |--
--|señales de control.                                                        |--
--|---------------------------------------------------------------------------|--
--|                        bUS dE ADDRESS-DATA-SYSTEM                         |--
--|...........................................................................|--
--|                                                                           |--

--|dEMULTiPLEXACióN dE lOS bUSES ADS                                          |--

--Enclava la parte baja de la dirección (latch)
latched_AD <= to_X01(AD) WHEN to_X01(ALE) = '1' ELSE UNAFFECTED;

--Enclava el nibble alto de la dirección (latch)
latched_AS <= to_X01(AS) WHEN to_X01(ALE) = '1' ELSE UNAFFECTED;

--Enclava los bits S6-S3 en el bus de sistema en el flanco de subida de T2
--T2 comienza con la activación de alguna de las señales RDn, WRn o INTAn
System_Bus(6 downto 3) <= to_X01(AS) WHEN (to_X01(RDn)='0' OR to_X01(WRn)='0'
            OR to_X01(INTAn)='0') AND rising_edge(ck_i) ELSE UNAFFECTED;

Address_Bus <= latched_AS & to_X01(A) & latched_AD ;--Genera la derección completa
System_Bus <= System_Bus(6 downto 3) & "00" & to_X01(SS0n);
--Genera el bus del sistema

--|tANSCEPTOR BiDiRECCiONAL                                                   |--
data_Bus <= (OTHERS =>'Z') WHEN to_X01(DENn)='1' ELSE
to_X01Z(AD) WHEN to_X01(DT_Rn)='1' ELSE
UNAFFECTED;

AD <=       (OTHERS =>'Z') WHEN to_X01(DENn)='1' ELSE
data_Bus WHEN to_X01(DT_Rn)='0' ELSE
UNAFFECTED;

--|                                                                           |--
--|---------------------------------------------------------------------------|--
--|                     FINAL DEL DESACOPLO EN EL 8088                        |--
--==============================================================================--
```

41.4.2 DESACOPLO DE LOS BUSES DEL 8086

El 8086 tiene un bus de datos de 16 bits, por lo que para desacoplarlo, deberemos utilizar un transciver de 16 bits (o dos de 8 bits como el 74F245). En la figura 8 podemos ver un esque básico del desacoplo completo de los buses del 8086.

Bajo esta columna, podemos observar el código de los archivos "system_8086mn" y "system_8086mx" que modelan la demultiplexación de los buses. Podemos observar que existen muy pocas diferencias con el explicado anteriormente, la más importante es que el transceptor bidireccional y latched _AD, ahora son de 16 bits. Las señales ALE, DENn y DT/Rn son generadas por el controlador de bus.

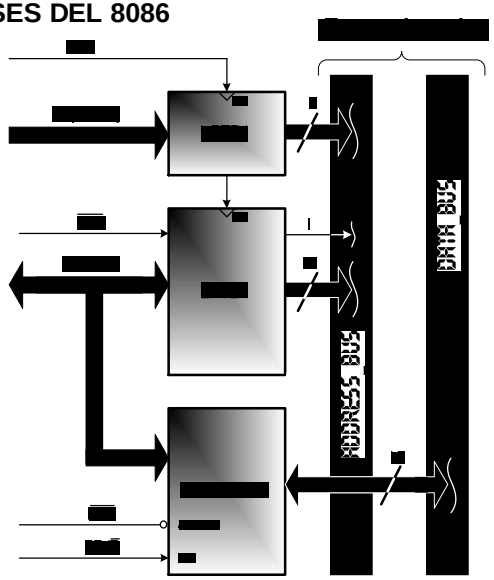


Fig 9.-Desacoplo total de los buses del 8086.

```
--==============================================================================--
--|           -BUFFERS, LATCHEDS i TRANSCIVERS (DESACOPLO EN EL 8086)-        |--
--|                                                                           |--

--|dEMULTiPLEXACióN dE lOS bUSES ADS                                          |--

--Enclava los 16 bits de la dirección (latch) y la señal BHEn
latched_AD   <= to_X01(AD)     WHEN to_X01(ALE) = '1' ELSE UNAFFECTED;
latched_BHEn <= to_X01(BHEnS7) WHEN to_X01(ALE) = '1' ELSE UNAFFECTED;

--Enclava el nibble alto de la dirección (latch)
latched_AS <= to_X01(AS) WHEN to_X01(ALE) = '1' ELSE UNAFFECTED;

--Enclava los bits S6-S3 en el bus de sistema en el flanco de subida de T2
--T2 comienza con la activación de alguna de las señales RDn, WRn o INTAn
System_Bus(6 downto 3) <= to_X01(AS) WHEN (to_X01(RDn)='0' OR to_X01(WRn)='0'
            OR to_X01(INTAn)='0') AND rising_edge(ck_i) ELSE UNAFFECTED;

Address_Bus <= latched_AS & latched_AD ;--Genera la derección completa
System_Bus <= System_Bus(6 downto 3) & "000";--Genera el bus del sistema

--|tANSCEPTOR BiDiRECCiONAL                                                   |--
data_Bus <= (OTHERS =>'Z') WHEN to_X01(DENn)='1' ELSE
to_X01Z(AD) WHEN to_X01(DT_Rn)='1' ELSE
UNAFFECTED;


AD <=       (OTHERS =>'Z') WHEN to_X01(DENn)='1' ELSE
data_Bus WHEN to_X01(DT_Rn)='0' ELSE
(OTHERS =>'Z');
--|                                                                           |--
--|---------------------------------------------------------------------------|--
--|                     FINAL DEL DESACOPLO EN EL 8088                        |--
--==============================================================================--
```

41.5 MAPA DE MEMORIA

Es un elemento común para todos los sistemas que define como están las repartidas las direcciones a las que el microprocesador puede acceder.

Los microprocesadores tienen un bus de direcciones de 20 líneas, lo que significa que podemos acceder a 2 ^ 20 (1Megabyte) posiciones de memoria. En el caso de acceso a dispositivos de entrada/salida, el microprocesador utiliza sólo las 16 primeras líneas del bus de direcciones, por lo que podemos acceder a 2 ^ 16 (64K) posiciones dentro de la memoria dedicada a los dispositivos I/O. La manera en la que repartimos estas direcciones entre los dispositivos que vamos a colocar dentro del rango de posiciones indicado (mapear el dispositivo), queda definido en el mapa de memoria. Este mapa de memoria, al detectar una transferencia, comprueba la dirección que sitúa dentro del espacio reservado al dispositivo determinado con el que se va a realizar la transferencia, dispositivo que el mapa de memoria habilita para que pueda comunicarse y realizar la operación. Lo mismo ocurre con las direcciones reservadas para dispositivos I/O.

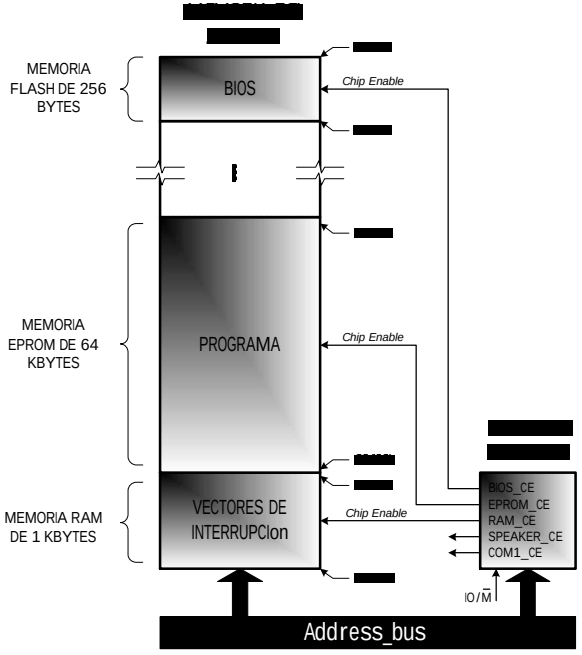


Fig 10.-Ejemplo de Configuración de la Memoria del Sistema.

Como ejemplo, implementemos el mapa de memoria necesario para configurar la memoria del sistema como o podemos ver en la figura 10.

En la memoria hay "mapeados" tres dispositivos de almacenamiento, una memoria RAM de 1 Kbyte que contiene los vectores de interrupción, a continuación una memoria EPROM de 64 Kbytes que contiene el programa y en la parte alta una memoria Flash de 256 bytes que contiene la BIOS del sistema. El reparto de direcciones lo podemos ver en la siguiente tabla:

MAPEADO DE LAS MEMORIAS (IO_Mn='0')		
DISPOSITIVO/ENABLE	CAPACIDAD	MAPEADO
RAM/RAM_CE	1K (10 lineas)	00000h-003FFh
EPROM/EPROM_CE	64K (16 líneas)	00400h-0F3FFh
FLASH/FLASH_CE	256 Bytes (8 líneas)	FFF00h-FFFFFh
DISPOSITIVOS I/O (IO_Mn='1')		
SPEAKER/SPEAKER_CE	1 posición	-0200h
COM1/COM1_CE	8 posiciones (3 líneas)	-03F8h/-03FFh

Tab 7.-Tabla de Dispositivos Mapeados en el Espacio de Memoria e I/O.

Como se puede observar también he mapeado en la memoria reservada para dispositivos I/O, un altavoz (speaker) que solo utiliza una dirección y un dispositivo de comunicación serie, COM1.

El código que modelaría este ejemplo de mapa de memoria, sería:

```
--|--Memoria de los vectores de interrupción       |--
--|RAM_CE <= '1' WHEN to_X01(IO_Mn)='0' AND        |--
--|ADRRESS <= X"003FF" AND ADRRESS >= X"00000" ELSE|--
--|'0';                                            |--
--|--Memoria EPROM de 64K a continuación           |--
--|EPROM_CE <= '1' WHEN to_X01(IO_Mn)='0' AND      |--
--|ADRRESS <= X"0F3FF" AND ADRRESS >= X"00400" ELSE|--
--|'0';                                            |--
--|--Memoria 256 palabras de memoria BIOS(al final)|--
--|BIOS_CE <= '1' WHEN to_X01(IO_Mn)='0' AND       |--
--|ADRRESS <= X"FFFFF" AND ADRRESS >= X"FFF00" ELSE|--
--|'0';                                            |--
--|                                                |--
--|--Dirección de un altavoz                       |--
--|SPEAKER_CE <= '1' WHEN to_X01(IO_Mn)='1' AND    |--
--|           ADRRESS = X"-0020" ELSE '0';         |--
--|--Direcciones en las que se encuentra COM1      |--
--|COM1_CE <= '1' WHEN to_X01(IO_Mn)='0' AND       |--
--|ADRRESS <= X"-03F8" AND ADRRESS >= X"-03FF" ELSE|--
--|'0';                                            |--
```

Para no complicar más el modelo, he supuesto que los sistemas utilizados para la simulación, sólo tienen un dispositivo de almacenamiento, una memoria de 1 Mbyte que ocupa todas las posiciones de memoria, de manera que se habilita siempre que el microprocesador ejecuta una operación de transferencia. En el siguiente código podemos ver el código real implementado en todos los archivos "system_*.vhd".

```
--|-------------------------------------------------|--
--|                mAPA dE mEMORiA                  |--
--|.................................................|--
--|                                                 |--

--RAM Chip Enable
RAM_CE <= '1' WHEN to_X01(IO_Mn)='0' ELSE '0';
--IO Chip Enable
IO_CE  <= '1' WHEN to_X01(IO_Mn)='1' ELSE '0';

--|-------------------------------------------------|--
--|           FINAL DEL MAPA DE MEMORIA             |--
--=================================================--
```

También se puede observar que sólo existe un dispositivo de I/O que ocupa todas las posiciones reservadas para los periféricos, por lo que se activa siempre que desde el micro accedemos a estos (instrucciones IN/OUT -> IO_Mn='1').

41.6 GENERADOR DE LA SEÑAL TESTn

Este es un proceso común para todos los sistemas.

Cuando el microprocesador ejecuta la instrucción WAIT, paraliza la ejecución del programa a la espera de que aparezca un '0' en el terminal TESTn. Este comportamiento suele ser utilizado para sincronizar software con el hardware del sistema. Por ejemplo, para comunicarse con el coprocesador matemático, después de la instrucción ESC que le indica al coprocesador la operación que tiene que realizar, se ejecuta una instrucción WAIT que para la ejecución hasta que el coprocesador matemática pone un '0' en la señal TESTn indicando que ha terminado la operación. Este tipo de sincronización, también es utilizado típicamente para actualizar la memoria de vídeo (en los sistemas que dispongan de tarjeta gráfica) en el momento en el que se produce el retrazo vertical en el tubo de imagen de la pantalla.

Si queremos simular este tipo de sincronización hardware, es necesario que incluyamos la instrucción WAIT en el programa para parar la ejecución y generemos la señal TESTn cuando queramos reanudar

la ejecución. Para generar la señal TESTn, tenemos que crear un flanco de subida en la señal "do_TEST", este flanco de subida, despierta el proceso "TEST_gen" que pone a nivel bajo la señal TESTn durante tres ciclos de reloj, tiempo más que suficiente para que el micro reconozca esta señal. El código de system (n0/r2) que implementa este simple proceso es el siguiente:

```
--=====================================================--
--|           -GENERADOR DE SEÑAL DE TEST-            |--
--|                                                   |--
--|---------------------------------------------------|--
--|                                                   |--

--|TiEMPOS eN lOS QuE aPARECE tEST           N        |--
    do_TEST <= '0','1'AFTER 840 us;

--|PrOCESO PrAL                                       |--
    TEST_gen: PROCESS
    BEGIN
        WAIT UNTIL rising_edge(do_TEST);
            TESTn <='0', '1' AFTER 3*TCLCL;
    END PROCESS TEST_gen;
--|                                                   |--
--|---------------------------------------------------|--
--|             GENERADOR DE SEÑAL DE TEST            |--
--=====================================================--
```

Se puede observar que aparece una generación de TESTn a los 840 microsegundos de la ejecución del programa. La aparición de un '0' en el terminal de test, sólo tiene sentido cuando el microprocesador está parado por una instrucción WAIT, en otro caso es una acción completamente transparente para el sistema y el miicroprocesador.

Si queremos que lanzar más de una generación de TESTn, simplemente tenemos que modificar la asignación concurrente de "do_TEST", por ejemplo:

```
--|TiEMPOS eN lOS QuE aPARECE uNA iNTERRUPCióN       |--
    do_TEST <= '0','1'AFTER 840 us,
--              '0','1'AFTER 853 us;
```

Este código genera dos veces la señal TESTn, una a los 840 microsegundos de la ejecución del programa y otra a los 853 microsegundos.

La lógica que siguen todos estos procesos, es muy parecida, hay una asignación (do_"SEÑAL") donde programamos los tiempos en los que aparece la señal y un proceso ("SEÑAL"_gen) que genera la señal con los peores tiempos que se pueden dar. Los tiempos se pueden configurar desde las GENERICS de los sistemas.

41.7 GENERADOR DE INTERRUPCIONES

Existen tres tipos de interrupciones externas (o hardware) que debemos simular:

- RESET, interrupción de mayor prioridad que inicializa el sistema.
- NMI, interrupciones no enmascarables, lanzan irremediablemente una interrupción de tipo 2 al terminal la ejecución de la instrucción actual.
- INTR, interrupciones mascarables, la activación de esta señal, lanza la secuencia de servicio de interrupción si están activadas. Para activar las interrupciones mascarables, el flag I debe estas a '1'. En la secuencia del servicio de interrupción, se lanzan dos ciclos de bus de reconocimiento de interrupción en los que el microprocesador recoge del bus de datos el tipo de interrupción que tiene que ejecutar.

Salvo el primer tipo que por sus características especiales explicamos anteriormente en el "generador de RESET", los generadores de las interrupciones NMI e INTR se explicarán a continuación.

41.7.1 GENERADOR DE INTERRUPCIONES NMI

Este generador, común para todos los sistemas, sigue la misma lógica que los anteriores. Una asignación concurrente de la señal "do_NMI", lanza el proceso "NMI_gen" cuando hay una transición de '0' a '1'. El proceso "NMI_gen" activa la señal de petición de

```
--=====================================================--
--|           -GENERADOR DE LA SEÑAL NMI-             |--
--|                                                   |--
--|---------------------------------------------------|--
--|                                                   |--
--|TiEMPOS eN lOS QuE aPARECE uNA iNTERRUPCióN        |--
    do_NMI <= '0'              ,'1' AFTER 790 us,
              '0' AFTER 800 us,'1' AFTER 818 us;
--|PrOCESO PrAL                                       |--
    NMI_gen: PROCESS
    BEGIN
        WAIT UNTIL rising_edge(do_NMI);
            --Genera NMI fallida (menor de dos ciclos)
            NMI <='1' AFTER TCLCL, '0' AFTER 2.5*TCLCL,
            --Genera NMI aceptada
                 '1' AFTER 5*TCLCL, '0' AFTER 8*TCLCL;
    END PROCESS NMI_gen;
--|                                                   |--
--|---------------------------------------------------|--
--|             GENERADOR DE LA SEÑAL NMI             |--
--=====================================================--
```

interrupción no mascarable NMI durante un intervalo de tiempo. En la columna anterior podemos ver el código de este generador.

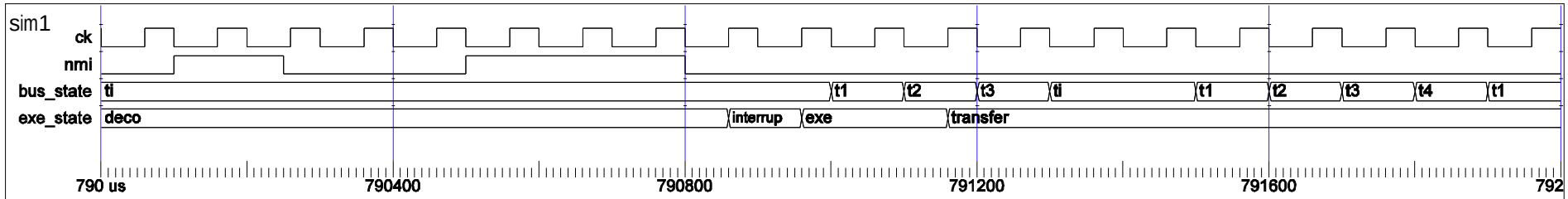
Se observa que aparecen dos peticiones de NMI, una a los 790 microsegundos y otra a los 818. En las simulaciones 1 y 2 de la página siguiente, podemos ver estas peticiones de NMI.

Puede apreciarse en ambas simulaciones (simulaciones 1 y 2) y en el código anterior, que en la secuencia de generación de NMI primero se genera una petición errónea. Una petición errónea es aquella en la que NMI no permanece por lo menos 2 ciclos de reloj a nivel alto (como se explicó en el desarrollo del proyecto). Seguidamente, aparece una petición válida que es inmediatamente atendida.

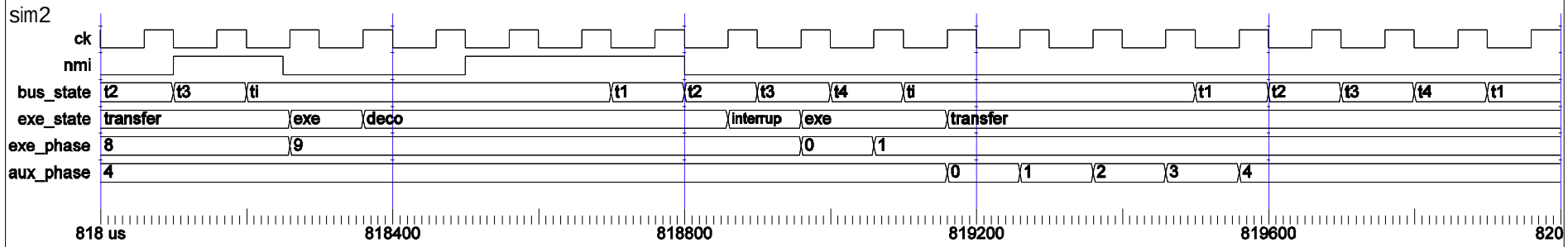
En la simulación 1, NMI aparece con el bus parado mediante la instrucción HLT. En estas condiciones, el bus permanece ocioso (estado Ti) hasta la aparición de una interrupción o una petición de bus. En este caso la respuesta a la petición es la más rápida (condiciones de menor latencia) y la unidad de ejecución comienza la secuencia de servicio de NMI un ciclo después de ser aceptada la señal NMI.

En la segunda simulación, aparece la petición de interrupción durante la ejecución normal de un programa. En este caso coincide que la petición aparece durante la decodificación de una instrucción por lo que esta decodificación es inmediatamente interrumpida para atender la interrupción no mascarable (saltamos del estado deco al de interrupción). La latencia en este caso también es mínima.

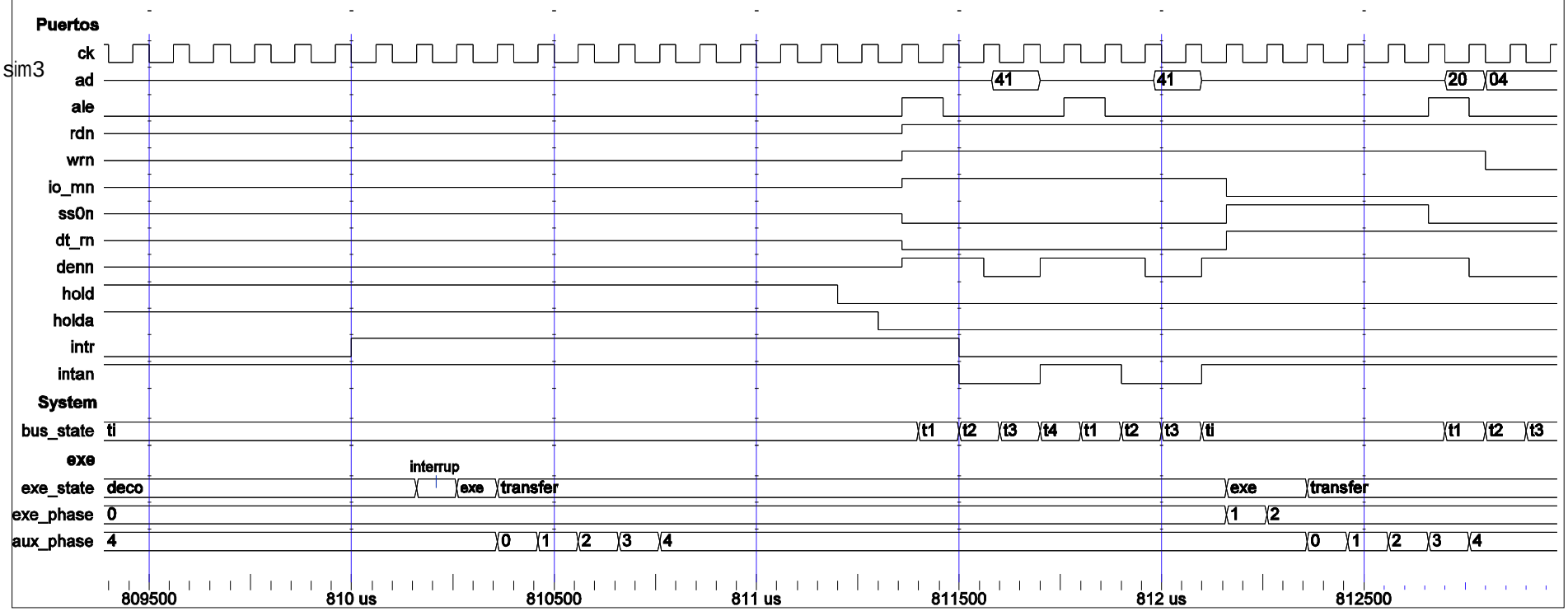
Estas simulaciones se hicieron "corriendo" el programa de depuración "prog27.hex" que es un programa que "empalma" una tras otra todas las instrucciones de los microprocesadores. El programa en si carece de sentido, pero es un buen programa para realizar pruebas por ejecutar todas las instrucciones utilizando la estrategia de "error acumulativo"; el resultado de luna instrucción es utilizado como operando de entrada de la siguiente. Comprobando el resultado final podemos casi asegurar que no ha habido fallos durante la ejecución del programa.



Sim 1.- Aparece una Petición de Interrupción no Mascarable, NMI.



Sim 2.- Aparece una Petición de Interrupción no Mascarable, NMI Estando en .Servicio una INTR.



Sim 3.- Petición de Interrupción INTR Cuando el Bus Local está Cedido.

41.7.2 GENERADOR DE INTERRUPCIONES INTR

Es un proceso utilizado en todos los sistemas de generación de vectores, sigue el esquema clásico. Una transición de '0' a '1' en "do_INTR", despierta el proceso "INTR_gen" que genera la señal INTR indicándole al microprocesador que hay una petición de interrupción enmascarable. El microprocesador responde activando la señal "INTA" que indica que la petición ha sido aceptada. en ese momento el generador de INTR desactiva la petición de interrupción llevando INTR a '0'. Este "handsake", lo podemos ver en el siguiente código:

```
--=================================================--
--|           -GENERADOR DE LA SEÑAL INTR-          |--
--|                                                 |--
--|-------------------------------------------------|--
--|                                                 |--
--|TiEMPOS eN lOS QuE aPARECE uNA iNTERRUPCióN      |--
    do_INTR <= '0'             ,'1' AFTER 810 us,
               '0' AFTER 815 us,'1' AFTER 840 us;
--|PrOCESO PrAL                                     |--
    INTR_gen: PROCESS
    BEGIN
        WAIT UNTIL rising_edge(do_INTR);
            INTR <= '1';
        WAIT UNTIL falling_edge(INTAn);
            --Desactivo la petición de interrupción
            INTR <= '0';
    END PROCESS INTR_gen;
--|                                                 |--
--|-------------------------------------------------|--
--|            GENERADOR DE LA SEÑAL INTR           |--
--=================================================--
```

En la secuencia de operaciones que realiza el microprocesador para atender una INTR, está la generación del ciclo de reconocimiento de interrupción "intr_ack" mediante el cual recoge del bus de datos el tipo de interrupción que tiene que atender. El tipo de interrupción (o puntero a la tabla de vectores de interupción), podemos modificarlo cambiando la constante de configuración "pointer_INTR" en las GENERICS del sistema (ver la figura 3). El tipo de interrupción aparece en el bus al ser INTA='0', como se puede ver en el código que lo modela:

```
--Saca los datos al bus con los peores tiempos
Data_Bus<=pointer_INTR AFTER TCLCL-TDVCL WHEN INTAn='0'
ELSE (OTHERS => 'Z') AFTER TCLDX;
```

Como podemos observar en el código, aparecen dos peticiones de INTR, una a los 810 microsegundos y otra a los 840. La segunda es una INTR que aparece durante la ejecución normal de un programa. La primera (que podemos ver en la simulación 3), aparece cuando el sistema ha cedido el bus local (observar el estado de HOLD y HOLDA). Es estas condiciones, INTR no es atendida hasta que el maestro de canal ha terminado de utilizar el bus y devuelve el control al micro (indicado por la transición a nivel bajo de la señal HOLD). En la simulación se puede apreciar perfectamente como la unidad de ejecución se queda enclavada en el estado "transfer" (el bus local no está libre y el árbitro de bus interno n3/r3, no cede el control del bus a DFU n3/r4). En el momento en el que el bus queda libre (HOLD='0'), se lanzan los dos ciclos consecutivos de reconocimiento de interrupción "intr_ack" donde recogemos el tipo de interrupción que la CPU tiene que atender (41h en la simulación que es el valor de la GENERIC "pointer_INTR"). En la fase de ejecución 1 (exe_phase=1), la unidad de ejecución (n1/r4) tiene el tipo de interrupción y está lista para acceder, en la fase de ejecución 2, a la tabla de vectores de interrupción. Debido a que INTR apareció cuando el bus no estaba libre, el tiempo de respuesta del sistema a la petición (latency) es mayor.

41.8 CONTROLADOR DE BUS

El controlador de bus es un elemento que aparece sólo en el modo de funcionamiento máximo, es decir en los archivos "system_8088mx.vhd" y "system_8086mx .vhd" de la librería de simulación "up808x_tb". El controlador de bus es un dispositivo encargado de generar las señales que desaparecen del bus de control del microprocesador al hacerlo funcionar en modo máximo, es decir, a partir de S2n/S1n/S0n, genera las señales WRn, IO_Mn, DT_Rn, DENn, ALE e INTAn. Está basado el el controlador de bus de Intel (i8288, ver 7-6 pág.282 Brey), con la diferencia de que el i8288 genera las señales necesarias para el control de "multibus" y aquí no son necesarias. En realidad este controlador de bus es parte de la lógica interna del microprocesador, unidad "Generador de Ciclos de Bus" BCG n3/r9 en la BIU.

Los procesos que modelan el controlador de bus son un poco largos por lo que no voy a insertarlos en esta memoria. para verlos, podemos acceder a los archivos indicados anteriormente.

El controlador de bus está formado por 3 procesos como podemos ver en la figura 11:

- bc_state, es un proceso síncrono que se encarga de pasar al siguiente estado de bus, existen 4 estados. Es equivalente al estado T que utilizamos en el interior del micro.
- bc_ph1, es un proceso síncrono que en función del estado indicado por bc_state y el ciclo de bus indicado por el microprocesador en las señales S0n:S2n, genera las señales que se activan el la fase 1.
- bc_ph2, es un proceso síncrono, que de manera similar a bc_ph1, genera las señales activas en la fase ph2 del reloj.

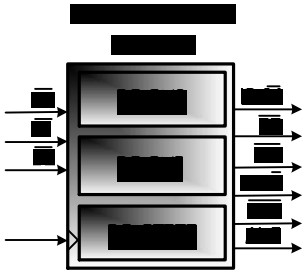


Fig 11.-Procesos que Forman el Controlador de Bus.

La lógica que implementa el controlador de bus es muy parecida, si no la misma, que la utilizada en el interior de los modelos para generar esas mismas señales. La mayor capacidad del modo máximo, hace necesario sacar lógica interna al exterior para poder dejar pines libres con los que implementar esas nuevas características que aumentan la capacidad de los microprocesadores. Los tres pines ganados con esta "chapuza", permiten a los modelos sacar el estado de la cola interna, señales QS0n y QS1n esencial para acoplar al sistema un coprocesador matemático, y la señal de bloqueo de bus LOCKn necesaria también para el afamado sistema MULTIBUS de Intel ®. También los microprocesadores en modo máximo aumentan la capacidad de cesión de bus local a dos canales (RQ/GT0n y RQ/GT1n) necesarios para las unidades de acceso directo a memoria DMA y para realizar el refresco de la memoria.

41.9 GENERADOR DE PETICIONES DE CESIÓN DE BUS

La petición de bus la realizan otros maestros de canal para pedir el control sobre los buses compartidos o buses locales. Esta característica aumenta enormemente las capacidades de un sistema al permitir que varios dispositivos utilicen los buses locales. El compartir el canal es bueno si aseguramos un ancho de banda suficiente para que el microprocesador llene la cola de instrucciones. Dispositivos clásicos que comparten los buses locales, son las unidades de acceso de directo a memoria DMA y los coprocesadores (cuando eran externos).

La lógica de las señales de petición-cesión del bus local, son bastante distintas dependiendo del modo de funcionamiento. El sencillo "handsake" de las señales HOL/HLDA en modo mínimo se convierten en una comunicación basada en tres pulsos a través de los pines bidireccionales RQn/GT[0/1]n en el modo máximo. Los procesos que modelan estos comportamientos en el sistema generador de vectores son muy distintos y los veremos por separado.

41.9.1 CESIÓN DE BUS EN MODO MÍNIMO, HOLD/HLDA

En el modo mínimo, la comunicación es muy sencilla, se realiza a través de las señales HOLD/HLDA como se puede apreciar en la figura 12.

Para simular esta comunicación, se ha creado en las GENERICS del sistema una constante de configuración llamada "x_Thold" que indica el ancho de pulso de la señal de petición HOLD. De nuevo, para lanzar el proceso de generación de HOLD (proceso "HOLD_gen"), debemos producir un flanco de subida en la señal "do_HOLD".

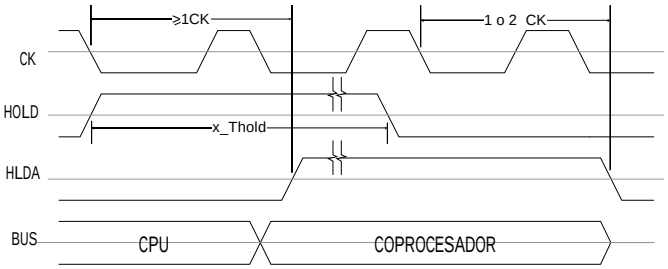


Fig 12.-Sincronización de las Señales HOLD/HLDA. Sólo en Modo Mínimo.

En el siguiente código podemos ver el proceso completo:

```
--=====================================================--
--|           -GENERADOR DE LA SEÑAL HOLD-            |--
--|                                                   |--
--|---------------------------------------------------|--
--|                                                   |--
--|TiEMPOS eN lOS QuE aPARECE PeTiCióN dEL bUS lOCAL|--
    do_HOLD <= '0'             , '1' AFTER 560 us,
               '0' AFTER 561 us, '1' AFTER 565 us,
               '0' AFTER 566 us, '1' AFTER 807 us,
               '0' AFTER 808 us, '1' AFTER 830 us;
--|PrOCESO PrAL                                      |--
    HOLD_gen: PROCESS
    BEGIN
        WAIT UNTIL falling_edge(ck_i);--Genera HOLD
        WAIT UNTIL rising_edge(do_DMA);
            HOLD <= '1';
        WAIT UNTIL rising_edge(HOLDA);
        WAIT FOR x_Thold*TCLCL;
            HOLD <= '0'; --Desactivo la petición HOLD
    END PROCESS HOLD_gen;
--|                                                   |--
--|---------------------------------------------------|--
--|            GENERADOR DE LA SEÑAL HOLD             |--
--=====================================================--
```

Para comprobar el funcionamiento de la lógica de HOLD/HLDA, aparecen 4 peticiones de bus (como se puede observar en el código anterior) durante la ejecución del programa de depuración "prog27.hex" En la simulación 4 podemos ver la petición de HOLD que aparece a los 560 microsegundos, en ella se puede apreciar como pese a estar la unidad de ejecución realizando una división, el bus es cedido (HLDA activada) de manera que las líneas del bus de datos, bus de direcciones y de sistema entran en estado de alta impedancia. Es un claro ejemplo de trabajo en paralelo de la EU y la BIU concretamente la unidad de multibus MU (n3/r6).

HOLD permanece a nivel alto durante 40 ciclos (GENERIC x_Thold=40), Durante tiempo desde la cesión del bus hasta que HOLD es cero, el dispositivo que pidió el bus, utiliza los buses para realizar la una operación específica. En el caso de las unidades DMA, utilizan los buses de datos y direcciones para mover bloques enteros de memoria sin que los datos pasen por el microprocesador, por lo que la transferencia es más rápida. Las tarjetas de sonido, los discos duros... utilizan unidades DMA. Los coprocesadores también piden el bus para realizar transferencias con la memoria, (coger datos, grabar resultados,...).

41.9.2 CESIÓN DE BUS EN MODO MÁXIMO, RQUEST/GRANT

Esta lógica generadora de vectores, solo aparece en los sistemas en modo máximo, es decir, en los archivos "system_8088mx.vhd" y "system_8086mx .vhd". Emula una posible petición de bus por parte de un maestro de canal utilizando la lógica de tres pulsos:

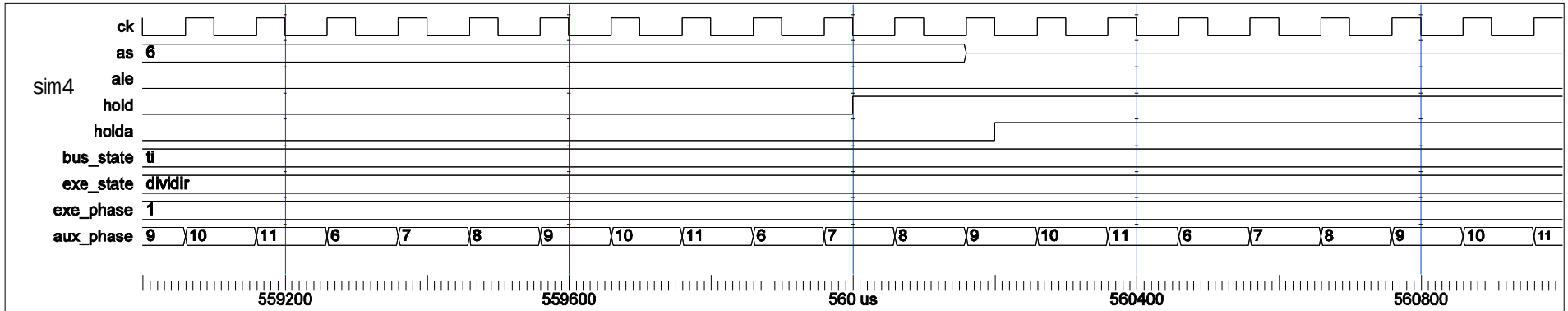
- Pulso 1. Pulso de petición de bus generado por el dispositivo que pide el bus.
- Pulso 2. Pulso de bus concedido generado por el microprocesador cuando resuelve la peticición y pone el bus local en triestado.
- Pulso 3. Pulso de operación terminada generado por el maestro que tiene el control sobre el canal para indicarle al microprocesador que ha terminado la operación y puede recuperar el control sobre el bus local.

En la figura 13 podemos ver una comparativa de la lógica utilizada en modo mínimo y la utilizada en modo máximo.

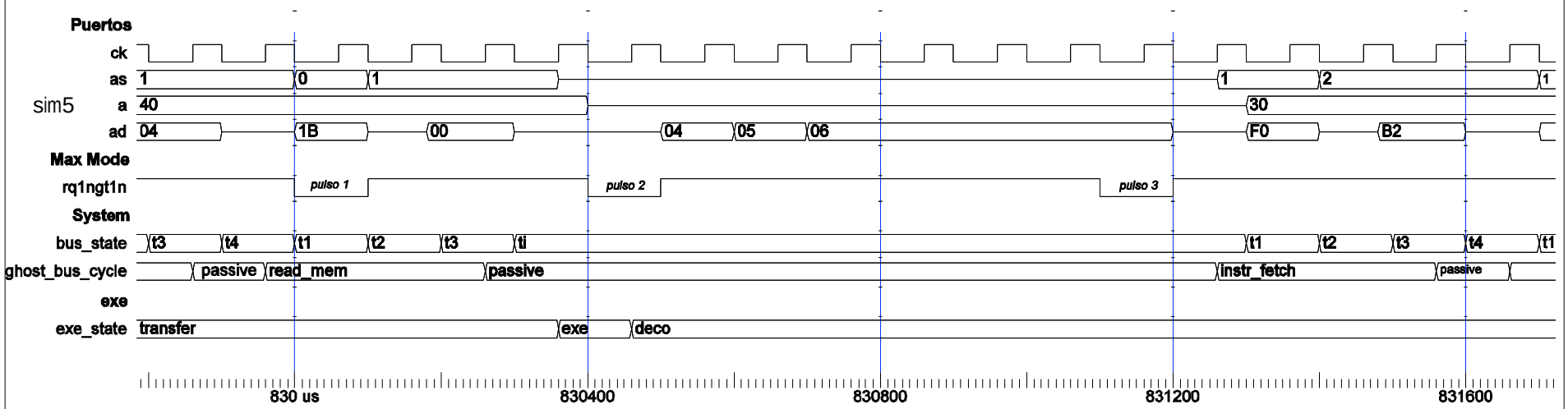
Como existen dos canales, la lógica que estudiaremos a continuación está duplicada en los archivos de sistema.

Los tres pulsos tienen un mismo medio físico, el terminal RQn/GT0n para el canal 0 y RQn/GT1n para el canal 1. El canal 0 tiene mayor prioridad que el canal 1. El primer pulso y el tercero, son generados por este controlador (terminal RQn/GT?n en modo salida) y el segundo pulso es recibido desde el micro (RQn/GT?n en modo estrada). Las líneas de código que cambian la dirección del puerto son las siguientes:

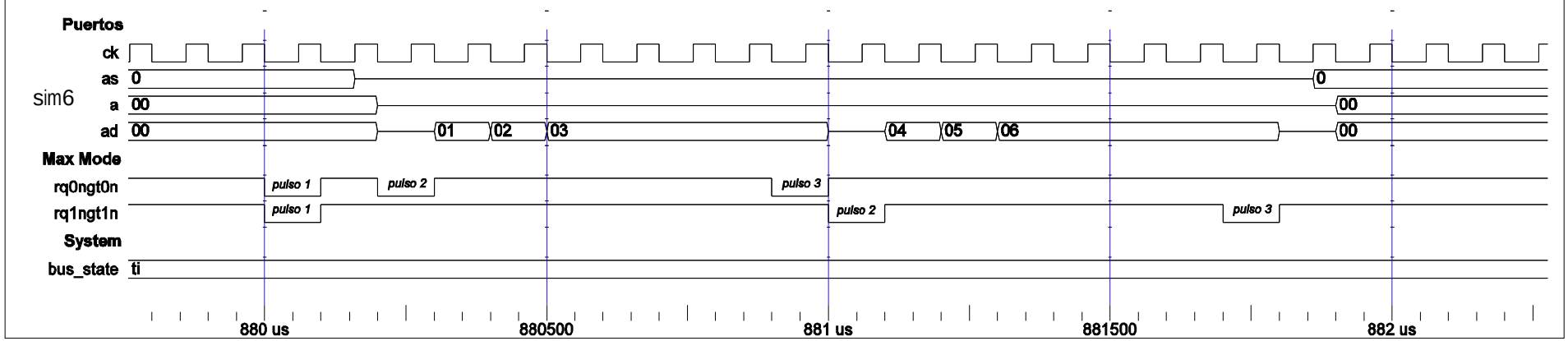
```
--|...................................................|--
--|cANAL 0                                            |--
    RQ0nGT0n <= RQ0n WHEN dma0_pulse=1 OR dma0_pulse=3
                ELSE 'H';
    GT0n <= to_X01(RQ0nGT0n) WHEN dma0_pulse=2
                ELSE 'H';
--|cANAL 1                                            |--
    RQ1nGT1n <= RQ1n WHEN dma1_pulse=1 OR dma1_pulse=3
                ELSE 'H';
    GT1n <= to_X01(RQ1nGT1n) WHEN dma1_pulse=2
                ELSE 'H';
--|Valor por defecto ('H') fijado por la resisten- |--
--|cia pull-up                                        |--
```



Sim 4.- Petición de Bus en Paralelo con la Ejecución de una División. Sistema en Modo Mínimo



Sim 5.- Petición de Bus en un Sistema Máximo Mientras se Ejecuta una Instrucción.



Sim 6.- Aparición de Dos Peticiones de Bus a la Vez. Sistema en Modo Máximo.

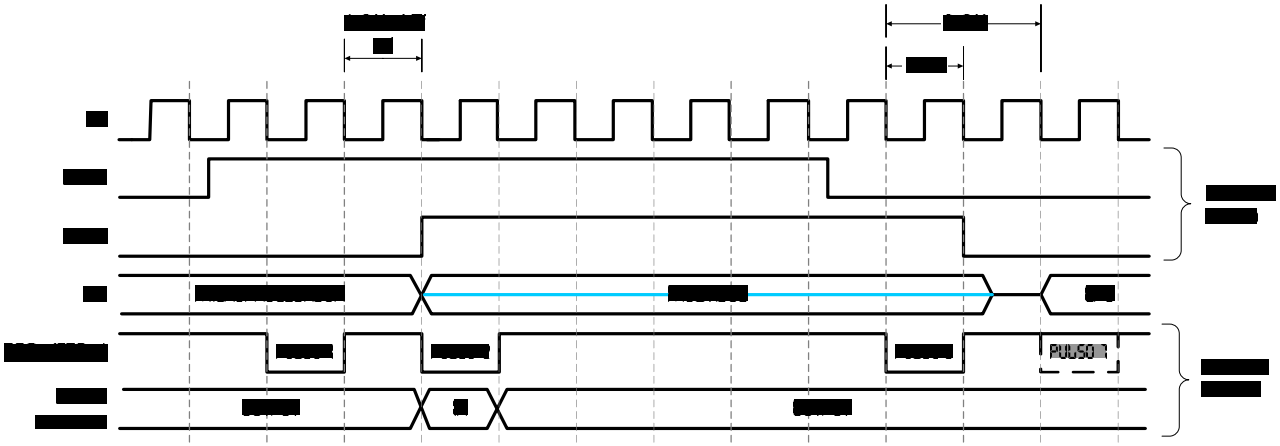


Fig 13.-Sincronización de la Petición de Bus en Modo Mínimo y Máximo

En la figura 13, se puede apreciar como es controlado el puerto bidireccional. Por lo demás sigue una lógica parecida a los anteriores procesos estudiados. el flanco de subida en la señal "do_dma0" o "do_dma1" indica a los procesos "dma1_gen" y "dma1_gen" respectivamente que lancen la secuencia de petición de bus. En las siguientes líneas de código podemos ver el proceso "dma0_gen":

En la simulación 5, podemos ver la secuencia de pulsos en el canal 1 para la petición de bus. En este caso, al recibir el primer pulso, el microprocesador se encuentra ejecutando una instrucción que está leyendo un dato de la memoria (bus_cycle=read_mem), por lo que no le manda el pulso de bus cedido (pulso 2) hasta que termina el ciclo de lectura y el bus entra en estado pasivo. Una vez que tiene el control sobre el canal, manda los datos 04h, 05h y 06h por el bus de datos. Después de enviar el último dato espera 4 ciclos y le indica al micro que ha terminado la operación generando el tercer pulso, momento en el que la CPU retoma el control del bus y lanza inmediatamente un ciclo de búsqueda de instrucción (instr_Fetch).

En la Simulación 6, se observa claramente la forma de actuar en caso de aparecer dos peticiones a la ver. El canal de mayor prioridad es atendido en primer lugar permaneciendo memorizada la petición del canal 1 que será atendida cuando el canal 0 deje libre el bus (pulso3). El canal 0 saca los datos 01h, 02h y 04h por el bus de datos, mientras que el canal 1 saca 04h, 05h y 06h.

```
--|...................................................|--
--|sECUENCiA dEL GeNERADOR dMA eN eL cANAL 0         |--
    dma0_gen: PROCESS
    BEGIN
        dma0_bus <= X"00";
        dma0_master <= FALSE;
        RQ0n <= '1';
        dma0_pulse <= 0;
        WAIT UNTIL rising_edge(do_dma0);
        WAIT UNTIL falling_edge(ck_i);
        --1er Pulso, Petición de bus.
        dma0_pulse <= 1;
        RQ0n <= '0';
        WAIT UNTIL falling_edge(ck_i);
        RQ0n <= '1';
        --2º Pulso, espero la confirmación por parte
        --del procesador
        dma0_pulse <= 2;
        WAIT UNTIL falling_edge(GT0n);
        WAIT UNTIL falling_edge(ck_i);
        WAIT UNTIL rising_edge(GT0n);
        dma0_master <= TRUE;
        --Bus cedido al dispositivo solicitante.
            dma0_bus <= X"01";
            WAIT UNTIL falling_edge(ck_i);
            dma0_bus <= X"02";
            WAIT UNTIL falling_edge(ck_i);
            dma0_bus <= X"03";
        WAIT FOR 4*TCLCL;
        WAIT UNTIL falling_edge(ck_i);
        --3er Pulso, operación completada por parte
        --del solicitante
        dma0_pulse <= 3;
        RQ0n <= '0';
        WAIT UNTIL falling_edge(ck_i);
    END PROCESS dma0_gen
```

41.10 PERIFÉRICOS I/O DEL SISTEMA DE SIMAULACIÓN

Para simular las instrucciones IN/OUT que acceden a la zona de memoria reservada para entradas y salidas (I/O), he diseñado un solo periférico:

Se trata de un simple registro de 8 bits que memoriza el dato que enviamos al bus con la instrucción OUT y lo devuelve cuando leemos cualquier posición reservada a I/O utilizando la instrucción IN. A continuación podemos ver el código.

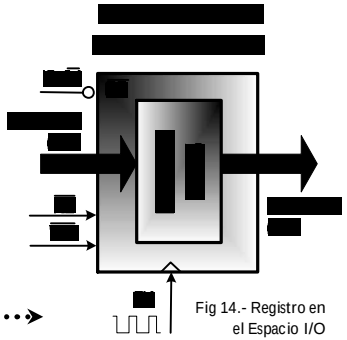


Fig 14.- Registro en el Espacio I/O

```
--==================================================--
--|                -ENTRADAS SALIDAS-               |--
--|                                                 |--
--|Para no complicar mucho el modelo se simula-     |--
--|ción, he diseñado un simple dispositivo de       |--
--|entrada-salida, el funcionamiento es parecido|--
--|al de la memoria, sólo que sin el procedimi-     |--
--|ento de carga. El proceso pral es despertado     |--
--|cuando la CPU direcciona un puerto de IO (se-|--
--|ñal IO_ce='1').                                  |--
--|El puerto actúa como una memoria de 1 byte       |--
--|guardando el dato recibido en operaciones de     |--
--|escritura en IO y devolviéndolo en lecturas      |--
--|de IO.                                           |--
--|-------------------------------------------------|--
--|                                                 |--
--|aSIGNACiONES cONCURRENTES                        |--
    IO_write <= '1' WHEN IO_ce='1' AND
                to_X01(WRn)='0' ELSE '0';
    IO_read  <= '1' WHEN IO_ce='1' AND
                to_X01(RDn)='0' ELSE '0';
    data_Bus <= read_data_IO WHEN IO_read='1' ELSE
                (OTHERS => 'Z');
--|LíSTA dE sENSiBiLiDAD dEL PrOCESO PrAL           |--
    IO: PROCESS (IO_write, IO_read) IS
    VARIABLE device : Std_Logic_Vector(7 DOWNTO 0);
--|cUERPO dEL PrOCESO                               |--
    BEGIN
        IF rising_edge(IO_write)   THEN
        --Escribe en el dispositivo de IO
            device := to_X01(data_bus);
        ELSIF rising_edge(IO_read) THEN
        --Lee un dato proporcionado por IO
            read_data_IO <= device;
        END IF;
    END PROCESS IO;
--|                                                 |--
--|-------------------------------------------------|--
--|          FINAL DE LAS ENTRADAS-SALIDAS          |--
--==================================================--
```

42. MODELADO DE LA MEMORIA

Este dispositivo forma parte del "modelado del entorno" (capítulo anterior) y los procesos que modelan la memoria, están dentro de los sistemas generadores de vectores de test (system, n1/r2), pero por su importancia, le dedico un capítulo aparte.

La memoria está organizada de forma distinta en los microprocesadores 8088 y 8086. Como se puede ver en la figura 1, en el 8088 la memoria está formada por un solo banco de 1 Megabyte de direcciones con un bus de datos de 8 bits, mientras que la memoria del 8086 está dividida en dos bancos de 512K de 8 bits cada uno (formando un bus de datos de 16 bits particionado en octetos).

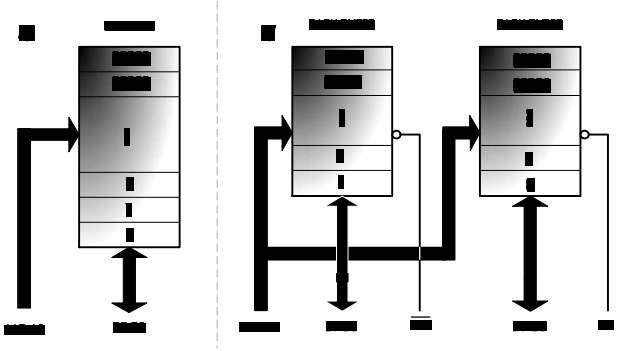


Fig 1.- a) Distribución de la Memoria en el 8088. b) Distribución de la Memoria en el 8086.

Estas diferencias en la estructuración de la memoria, se reflejan en los sistemas modelados para la generación de test. Por ello vamos a estudiar el modelo utilizado para las memorias de ambos micros por separado.

42.1 MODELO DE MEMORIA DEL 8088

La memoria que explicamos a continuación está compuesta por dos procesos que podemos encontrar en los archivos de sistema del microprocesador 8088 (archivos "system_8088mn.vhd" y "system 8088mx.vhd"). Los procesos son:

PROCESO	FUNCIÓN
Memory	Define la Memoria
Text_OUT	Genera un Fichero histórico de las operaciones realizadas sobre la Memoria

Tab 1.-Procesos que Modelan la Memoria

Además de estos procesos, existe un procedimiento dentro del proceso "Memory" que se encarga de inicializar la memoria. Es el cargador del programa, lo estudiaremos un poco más adelante.

Como se puede apreciar en la figura 1a, el 8088 trata la memoria física externa como un banco de 1 millón de posiciones que almacena datos de 8 bits. Para modelar este banco, utilizamos una array definido en la cabecera del proceso (proceso "Memory"). Este array está definido utilizando las GENERICS:

GENERIC	SIGNIFICADO
size	Número de registros de la memoria
addr_width	Número de bits del bus de direcciones
width	Ancho del registro de memoria

Tab 2.-Constantes GENERIC utilizadas. en el Generador de la Memoria.

El código en el fichero de sistema que define el array es:

```
--Constantes para definir el rango de la memoria
CONSTANT low_address  : natural := 0;
CONSTANT high_address : natural := size -1;

--Tipo de memoria definido con las constantes
TYPE memory_array_typ IS
ARRAY (natural RANGE low_address TO high_address) OF
Std_Logic_Vector(width-1  DOWNTO 0);

--Memoria del Sistema.
VARIABLE mem : memory_array_typ := (OTHERS =>
               "00000000"); --Inicializada a 0
```

El proceso "memory" es despertado cuando el microprocesador activa las señales de escritura (WRn) o lectura (RDn) y el mapa de memoria habilita la señal "RAM_CE" (ver "mapa de memoria").

```
M_write <= '1' WHEN RAM_ce='1' AND to_X01(WRn)='0'
           ELSE '0';--Lanza una escritura
M_read  <= '1' WHEN RAM_ce='1' AND to_X01(RDn)='0'
           ELSE '0';--Lanza una lectura
```

Las señales "M_write" y "M_read" son las que están en la lista de sensibilidad del proceso "memory".

En el cuerpo del proceso "memory", cuyo código podemos ver en la columna siguiente, podemos ver como la primera vez que el proceso es despertado, se ejecuta el procedimiento "load_hex" que carga el array de memoria ("mem") con el contenido del fichero indicado en la GENERIC "download_Filename". Esta

```
--|...............................................|--
--|cUERPO dEL PrOCESO                             |--
BEGIN -- Proceso Principal
IF NOT initialized THEN
    --Carga el Archivo en la Memoria
    --Procedimiento de Carga
    load_hex(mem,download_filename);
    initialized := TRUE;
END IF;

IF falling_edge(M_write) THEN
--Escribe en mem en el flanco de subida de WRn(del uP)
   mem(CONV_INTEGER(Address_Bus)) := to_X01(data_Bus);
ELSIF rising_edge(M_read) THEN
--Lee de mem en el flanco de bajada de RDn(del uP)
   read_data_M <= mem(CONV_INTEGER(Address_Bus));
END IF;
END PROCESS memory;
```

procedimiento es común para todos los sistemas y es el que inicializa la memoria cargando en ella el código (o programa) hexadecimal que queremos simular. Este código se encuentra en un fichero que debe tener un formato especial que estudiaremos más adelante.

En el código de arriba, también podemos ver como se almacena el dato que hay en el bus en el flanco de subida de la señal WRn y cómo en el flanco de bajada de la señal RDn, sacamos el dato a la señal "read_data_M" (ojo, no lo sacamos al bus de datos).

El dato sacado de la memoria, es puesto en el bus de datos con el ajuste de tiempos que vemos a continuación:

```
--Saca los datos al bus con los peores tiempos
Data_Bus <= read_data_M AFTER (TCLCL*(x_Tw+1))-TDVCL
            WHEN M_read='1' ELSE
            (OTHERS => 'Z') AFTER TCLDX;
```

Vemos que la memoria tiene en cuenta el número de estados de espera, indicado en la GENERIC "c_Twait", que tienen que esperar para sacar el dato

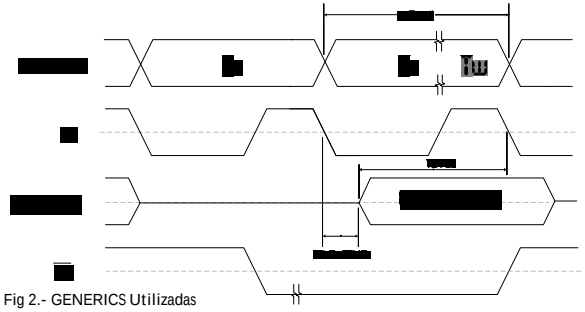


Fig 2.- GENERICS Utilizadas Para Leer un Dato

42.2 MODELO DE MEMORIA DEL 8086

Esta memoria la podemos encontrar en los archivos de sistema de los microprocesadores 8086 (archivos "system_8086mn.vhd" y "system_8086mx.vhd".

Como podemos observar en la figura 1 b, la memoria en los 8086 está dividida en dos bancos de de 512K posiciones de memoria y 16 bits de bus de datos dividido en dos octetos. La implementación sigue la misma lógica utilizada para el modelado de la memoria del 8088, por lo que podemos considerar válido todo lo explicado anteriormente a excepción del cuerpo del proceso "Memory" que cambia sustancialmente para implementar la nueva lógica de acceso a los dos bancos de memoria. En el código siguiente podemos apreciar estos cambios:

```
--|cUERPO dEL PrOCESO                                    |--
BEGIN -- Proceso Principal
    IF NOT initialized THEN
    --Carga el Archivo en la Memoria
        --Procedimiento de Carga
        load_hex(mem,download_filename);
        initialized := TRUE;
    END IF;

IF falling_edge(M_write)  THEN
--Escribe en mem en el flanco de subida de WEn(del uP)
    IF Address_Bus(0)='0' AND latched_BHEn='0' THEN
    --Escritura de word (16 bits)
        mem(CONV_INTEGER(Address_Bus(19 DOWNTO 0)))
        := to_X01(data_Bus)(7 DOWNTO 0);
        mem(CONV_INTEGER(Address_Bus(19 DOWNTO
         0))+1) := to_X01(data_Bus)(15 DOWNTO 8);
   ELSIF Address_Bus(0)='0' AND latched_BHEn='1' THEN
    --Escritura del byte bajo (8 bits)
        mem(CONV_INTEGER(Address_Bus(19 DOWNTO 0)))
        := to_X01(data_Bus(7 DOWNTO 0));
   ELSIF Address_Bus(0)='1' AND latched_BHEn='0' THEN
    --Escritura del byte alto (8 bits)
        mem(CONV_INTEGER(Address_Bus(19 DOWNTO 0)))
        :=  to_X01(data_Bus(15 DOWNTO 8));
   END IF;
ELSIF rising_edge(M_read) THEN
--Lee de mem en el flanco de bajada de RDn(del uP)
    IF Address_Bus(0)='0' AND latched_BHEn='0' THEN
    --Lectura de word (16 bits)
        read_data_M <=
        mem(CONV_INTEGER(Address_Bus(19 DOWNTO 0))+1) &
        mem(CONV_INTEGER(Address_Bus(19 DOWNTO 0)));
    ELSIF Address_Bus(0)='0' AND latched_BHEn='1' THEN
    --Lectura del byte bajo (8 bits)
        read_data_M(7 DOWNTO 0) <=
        mem(CONV_INTEGER(Address_Bus(19 DOWNTO 0)));
    ELSIF Address_Bus(0)='1' AND latched_BHEn='0' THEN
    --Lectura del byte alto (8 bits)
        read_data_M(15 DOWNTO 8) <=
        mem(CONV_INTEGER(Address_Bus(19 DOWNTO 0)));
    END IF;
END IF;
END PROCESS memory;
```

Podemos observar que la inicialización de la memoria es la misma y que la definición del array de memoria (señal "mem") no cambia. Lo que si cambia es la lógica de la lectura y escritura en el array.

En función de las señales A(0) y BHEn o lo que es lo mismo Address_Bus(0) y latched_BHEn, que son sus respectivas señales registradas (ver el desacoplo de los buses del 8086), efectuamos la transferencia con el octeto alto del bus de datos, el bajo, o ambos. Ver la siguiente tabla:

Address _Bus(0)	latched _BHEn	TIPO DE TRANSFERENCIA
0	0	Transferencia de 16 bits, lectura/escritura de las direcciones Address_Bus y Address_Bus+1
0	1	Transferencia de los 8 bits del banco bajo. Dirección Address_Bus
1	0	Transferencia de de 8 bits con el banco alto. Dirección Address_Bus
1	1	No hay tansferencia

Tab 3.-Banco activo en Función de Address_Bus(0) y latched_BHEn, Tipo de Transferencia.

Este modo de funcionamiento de la memoria, permite realizar transferencias de 8 o 16 bits a direcciones pares o impares. Ver los capítulos de las unidades DFU (n3/r4) e IFU (n3/r5) donde se explica detalladamente el alineamiento de las transferencias y la forma en la que opera el microprocesador para acceder a los datos de memoria.

42.3 CARGADOR DE MEMORIA

El cargador de memoria es un procedimiento común a todos los sistemas que "vuelca" el contenido del fichero indicado en la GENERIC "download_filenamen" en la memoria. Este procedimiento está dentro del proceso "memory" de todos los archivos "system_*.vhd" y se ejecuta una sola vez en la simulación del programa. El fichero que este proceso vuelca en memoria, contiene el programa que queremos simular y tiene un formato especial que permite posicionar código en cualquier parte de la memoria.

FORMATO

El formato del fichero que volcamos a memoria es el que nos proporciona directamente el compilador de ensamblador "Marcro Assembler" (MASM) al compilar un fichero escrito en ensamblador con la opción /Fl:

```
c:\> ML.EXE /c /Fl RAM.ASM
```

Este comando genera un archivo RAM.LST con un formato que apenas necesita modificaciones para que pueda ser cargado (y simulado) en la memoria del sistema.

El cargador va barriendo el fichero leyendo líneas y dentro de la línea, el carácter. Cuando encuentra el carácter "%" sabe que los cuatro dígitos siguientes le indican el segmento a partir del cual tiene que posicionar el código. Nosotros podemos indicarle tantos cambios de segmento como queramos y el cargador se encarga de posicionar el código que sigue a la definición de segmento en la posición física adecuada. El cargador también admite comentarios. Cuando encuentra el carácter ";" salta inmediatamente de línea y por lo tanto ignora el comentario.

A continuación podemos ver un ejemplo de fichero:

```
--|Ejemplo:                                                                |--
--|     %FFFF                reset segment       ;Segmento de Reset         |--
--|     0000  B8 BA78            mov ax, 0BA78h                             |--
--|     0003  8E D8              mov ds, ax                                 |--
--|     0005                 reset ends                                     |--
--|                                                                         |--
--|     %0000                interrupt segment   ;Segmento de Interrupciones |--
--|                              org 0004        ;Desp de la Interrupción 1 |--
--|     0004  B8 0600            mov ax, 0600h                              |--
--|     0007  8E D8              mov ds, ax                                 |--
--|     0009                 interrupt ends                                 |--
--|                                                                         |--
--|     %0040                code segment        ;Segmento de Código        |--
--|     0000  B8 0BFE            mov ax, 0BFEh                              |--
--|     0003  8E D8              mov ds, ax                                 |--
--|     0005                 code ends                                      |--
--|                          end                                            |--
```

42.4 FICHERO DE SALIDA

Para seguir la simulación, los sistemas generadores de vectores de test, implementan un proceso llamado "text_out" que genera un fichero historial de las acciones realizadas sobre la memoria. Podemos cambiar el nombre del archivo de salida modificando la GENERIC "log_filename".

El proceso "text_out" difiere un poco dependiendo del microprocesador que estamos simulando. A continuación los veremos estas diferencias

FORMATO DEL FICHERO LOG EN EL 8088

El estándar de VHDL para manejo de ficheros, es un poco primitivo y difiere según la revisión de la norma IE[3] que utilicemos. He utilizada la revisión del 93 por lo que para compilar este fichero, es preciso indicarle al compilador que utilice la sintaxis del 93, como se verá más adelante. VHDL, trata los ficheros de forma secuencial, no permitiendo ir a cualquier parte del fichero. La lectura/escritura se realiza de línea en línea y siempre de forma incremental, de manera que el archivo de salida contiene un registro (en el tiempo) de las operaciones realizadas sobre la memoria.

```
--==============================================================================--
--|                            -FICHERO DE REGISTRO-                             |--
--|------------------------------------------------------------------------------|--
--|                                                                              |--

--|LíSTA dE sENSiBiLiDAD dEL PrOCESO PrAL                                        |--
text_out: PROCESS (M_write,M_read,IO_write, IO_read) IS
    FILE file_out : TEXT OPEN write_mode IS log_filename;
    VARIABLE L : LINE; -- Linea
    -- Strings para componer la línea
    VARIABLE address : STRING(1 TO 8) := "Address=";
    VARIABLE data    : STRING(1 TO 5) := "Data=";
    VARIABLE ciclo   : STRING(1 TO 3) := "CK=";
    VARIABLE espacio : STRING(1 TO 1) := " ";
    VARIABLE rd_m    : STRING(1 TO 7) := "rd_mem:";
    VARIABLE wr_m    : STRING(1 TO 7) := "wr_mem:";

--|cUERPO dEL PrOCESO                                                           |--
BEGIN
--|FORMATO->  rd_mem:CK=23 Address=11111111111111110000 Data=10111011            |--
    IF falling_edge(M_write) THEN --Composición para la operación de escritura
        write(L,wr_m);write(L,ciclo);write(L,n_ciclo);write(L,espacio);
        write(L,address);write(L,Address_Bus);write(L,espacio);
        write(L,data);write(L,Data_Bus);write(L,espacio);
        writeline(file_out,L);
    ELSIF falling_edge(M_read) THEN --Composición para la operación de lectura
        write(L,rd_m);write(L,ciclo);write(L,n_ciclo);write(L,espacio);
        write(L,address);write(L,Address_Bus);write(L,espacio);
        write(L,data);write(L,data_bus);write(L,espacio);
        writeline(file_out,L);
    END IF;

END PROCESS text_out;
--|                                                                              |--
--|------------------------------------------------------------------------------|--
--|                        FINAL DEL FICHERO DE REGISTRO                         |--
--==============================================================================--
```

El fichero de salida está generado a base de líneas. Cada vez que el micro realiza una operación con la memoria, el proceso "text_out" que podemos ver en la columna anterior, graba una línea en el fichero con el siguiente formato:

```
(OPERACIÓN):CK=(nº de ciclo) Address=(Address_Bus) Data=(Data_Bus)
```

FORMATO DEL FICHERO LOG EN EL 8086

El proceso "text_out" del 8086 cuyo código podemos ver bajo estas líneas, es muy parecido al anterior, solo que en este caso, el formato de la línea que escribimos en el fichero es el siguiente:

```
(OPERACIÓN):CK=(nº de ciclo) Address/BHEn=(Address_Bus)/(latched_BHEn) Data=(Data_Bus)
```

El valor número de ciclo, indica el ciclo en el que aparece la operación. Para calcular el tiempo absoluto, hay que multiplicarlo por TCLCL=TCLCH+TCHCL=100ns.

```
--==============================================================================--
--|                            -FICHERO DE REGISTRO-                             |--
--|                                                                              |--
--| Este proceso, genera el fichero de salida indicado en la variable de log_ |--
--|filename. En el podemos encontrar las operaciones realizadas con la memoria|--
--|y el estado de los buses de direcciones y datos. Este fichero puede servir |--
--|para analizar el resultado de los programas.                                |--
--|------------------------------------------------------------------------------|--
--|                                                                              |--

--|LíSTA dE sENSiBiLiDAD dEL PrOCESO PrAL                                        |--
text_out: PROCESS (M_write,M_read,IO_write, IO_read) IS
    FILE file_out : TEXT OPEN write_mode IS log_filename;
    VARIABLE L : LINE; -- Linea
    -- Strings para componer la línea
    VARIABLE address : STRING(1 TO 13) := "Address/BHEn=";
    VARIABLE data    : STRING(1 TO 5)  := "Data=";
    VARIABLE ciclo   : STRING(1 TO 3)  := "CK=";
    VARIABLE separa  : STRING(1 TO 1)  := "/";
    VARIABLE espacio : STRING(1 TO 1)  := " ";
    VARIABLE rd_m    : STRING(1 TO 7)  := "rd_mem:";
    VARIABLE wr_m    : STRING(1 TO 7)  := "wr_mem:";

--|cUERPO dEL PrOCESO                                                           |--
BEGIN
--|FORMATO->                                                                    |--
--|  rd_mem:CK=23 Address/BHEn=1111111111111110000/0 Data=1011101100110110      |--
    IF falling_edge(M_write) THEN --Composición para la operación de escritura
      write(L,wr_m);write(L,ciclo);write(L,n_ciclo);write(L,espacio);
      write(L,address);write(L,Address_Bus);write(L,separa);write(L,latched_BHEn);
      write(L,espacio);write(L,data);write(L,data_bus);write(L,espacio);
      writeline(file_out,L);
    ELSIF falling_edge(M_read) THEN --Composición para la operación de lectura
      write(L,rd_m);write(L,ciclo);write(L,n_ciclo);write(L,espacio);
      write(L,address);write(L,Address_Bus);write(L,separa);write(L,latched_BHEn);
      write(L,espacio);write(L,data);write(L,data_bus);write(L,espacio);
      writeline(file_out,L);
    END IF;

END PROCESS text_out;
--|                                                                              |--
--|------------------------------------------------------------------------------|--
--|                        FINAL DEL FICHERO DE REGISTRO                         |--
--==============================================================================--
```

43. MODELO DE UN SISTEMA OPERATIVO MÍNIMO

Existen unas direcciones de memoria que son utilizadas por el microprocesador de forma intrínseca para dar soporte a las interrupciones y excepciones internas que pueden aparecer en el microprocesador. Estas direcciones las podemos ver en la figura 1. El modelo sistema operativo da cobertura a estas interrupciones programando la tabla de vectores y los servicios de interrupción para los 5 primeros vectores.

En el siguiente código, podemos ver el contenido íntegro del sistema operativo mínimo utilizado para las simulaciones (archivo: "so.asm"):

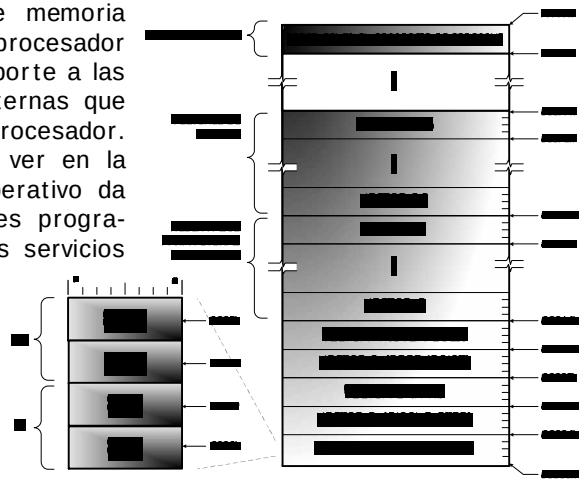


Fig 1.-Direcciones Reservadas

```
;    --=========================================================================--
;    --|PrOGRAMA dE SiSTEMA:       -SISTEMA OPERATIVO BÁSICO-                    |--
;    --|                                                                         |--
;    --| Estas líneas constituyen un sistema operativo básico. Para dar soporte a  |--
;    --|las interrupciones y excepciones internas, es necesario que el programa que|--
;    --|simulemos tenga estas líneas de código básicas para hacer funcionar el sis-|--
;    --|tema.                                                                    |--
;    --|-------------------------------------------------------------------------|--
;    --|sEGMENTO iNiCiAL                                                FFFF:0000 |--
;    --|.........................................................................|--
;    --|                                                                         |--
ini segment                              ;Segmento Incicial FFFF:0000
        mov     bx, code                 ;slto al segmento de código
        mov     cs, bx
ini ends
;    --|-------------------------------------------------------------------------|--
;    --|vECTORES dE iNTERRUPCiÓN                                        0000:0000 |--
;    --|.........................................................................|--

int_vect segment            ;Segmento de vectores de interrupción 0000:0000

        dd  0EFF0000h       ;Vector int 00h => 0EFF:0000 -> Error al Dividir
        dd  0EFF0100h       ;Vector int 01h => 0EFF:0100 -> Paso Único
        dd  0EFF0200h       ;Vector int 02h => 0EFF:0200 -> Terminal MNI
        dd  0EFF0300h       ;Vector int 03h => 0EFF:0300 -> Break Point
        dd  0EFF0400h       ;Vector int 04h => 0EFF:0400 -> Sobreflujo INTO

;    --|.........................................................................|--
;    --|Introducir Aquí los Vector de Interrupción de Usuario                    |--
;        Ejemplos:
;        org 41h*4h
;        dd  0EFF4100h      ;Vector int 41h => 0EFF:4100 -> INTR (recogido del bus)
;        org 21h*4h
;        dd  0EFF2100h      ;Vector int 21h => 0EFF:2100 -> Interrupción de exhibición
;        ...

int_vect ends
```

```
;    --|-------------------------------------------------------------------------|--
;    --|sERViViOS dE iNTERRUPCiÓN                                       0EFF:0000 |--
;    --|.........................................................................|--
int_serv segment                         ;Segmento de servicios de interrupción
0EFF:0000
    assume ds:data, es:extra, ss:stack
;    --|.........................................................................|--
;    --|div0  - int 00h                                                          |--
        org 0000h                        ;Servicio de int 00h
        nop                              ;Acciones (Introducir aquí las acciones)
        iret                             ;Retorno del servicio de interrupción
;    --|.........................................................................|--
;    --|step  - int 01h                                                          |--
        org 0100h                        ;Servicio de int 01h
        nop                              ;Acciones (Introducir aquí las acciones)
        iret                             ;Retorno del servicio de interrupción
;    --|.........................................................................|--
;    --|nmi   - int 02h                                                          |--
        org 0200h                        ;Servicio de int 02h
        nop                              ;Acciones (Introducir aquí las acciones)
        iret                             ;Retorno del servicio de interrupción
;    --|.........................................................................|--
;    --|break - int 03h                                                          |--
        org 0300h                        ;Servicio de int 03h
        nop                              ;Acciones (Introducir aquí las acciones)
        iret                             ;Retorno del servicio de interrupción
;    --|.........................................................................|--
;    --|over  - int 04h                                                          |--
        org 0400h                        ;Servicio de int 04h
        nop                              ;Acciones (Introducir aquí las acciones)
        iret                             ;Retorno del servicio de interrupción
;    --|.........................................................................|--
;    --|Introducie en esta parte los servicios de interrupción de Usuario. Ej:     |--
;    --|intr  - int XXh (XXh recogido del bus, 41h)                              |--
;        org 4100h                        ;Servicio de int 41h
;        nop                              ;Acciones (Introducir aquí las acciones)
;        iret                             ;Retorno del servicio de interrupción
;    --|.........................................................................|--
;    --|int21 - int 21h (MS-DOS)                                                 |--
;        org 2100h                ;Servicio de int 21h (exhivicion device en IO 0CDh)
;        push    ax
;        mov     al,dl
;        out     0CDh,ax          ;Pongo ax en el dispositivo de entrada salida en 41h
;        pop     ax
;        iret                     ;Retorno del servicio de interrupción
;        ...
int_serv ends
;    --|-------------------------------------------------------------------------|--
;    --|sEGMENTO dE PiLA                                                03FF:0000 |--
;    --|.........................................................................|--
stack segment                            ;Segmento de Pila 03FF:0000
  db      50 dup(?)                      ;reserva 50 bytes
stack ends
;    --|-------------------------------------------------------------------------|--
;    --|sEGMENTO dE dATOS                                               0FFF:0000 |--
;    --|.........................................................................|--
data segment                             ;Segmento de datos 0FFF:0000
  data_seg  dw      0FFFh                ;Segmento de Datos
  extra_seg dw      02FFh                ;Segmento de Datos
  stack_seg dw      03FFh                ;Segmento de Datos
;    --|.........................................................................|--
;    --|Introducir en esta parte los datos del programa. Ej:                     |--
;  dato_b    db      62h                ;Byte a convertir
;  dato_w    dw      0100h              ;Word a convertir
;  ...
data ends
```

```
;    --|-----------------------------------------------------------------------|--
;    --|sEGMENTO eXTRA                                                 02FF:0000 |--
;    --|.......................................................................|--
extra segment                          ;Segmento extra de datos 02FF:0000
;    --|.......................................................................|--
;    --|Introducir en esta parte los datos del Segmento extra. Ej:              |--
;  result    db                        ;Resultado
;  ...
extra ends
;    --|-----------------------------------------------------------------------|--
;    --|sEGMENTO dE CóDiGO                                            3000:0000 |--
;    --|.......................................................................|--
code segment                           ;Segmento de Pila 3000:0000
        assume cs:code, ds:data, es:extra, ss:stack
        mov     ax,data                ;Cargo los registros de segmento
        mov     ds,ax
        mov     es,extra_seg
        mov     ss,stack_seg
;    --|.......................................................................|--
;    --|Introducir las Instrucciones y Procedimientos Cercanos del Programa. Ej:   |--
;        mov     ax,word ptr dato_b       ;covierte a ASCII 62h -> 98d
;        call    desp
;        [...]
        wait                           ;Terminar con 1 o dos Instrucciones WAIT
        wait
code ends
end
;    --|-----------------------------------------------------------------------|--
;    --|                            FiNAL dEL PrOGRAMA                         |--
;    --=========================================================================--
```

Aunque el código es un poco largo, merece la pena tenerlo a la vista para estudiarlo. En el podemos observar varias segmentos:

- Un segmento inicial "ini_segment" que contiene el salto al segmento de código donde está cargado e programa. Este segmento inicial, está localizado en el segmento por defecto dónde el microprocesador comienza a ejecutar instrucciones después de la aparición de un RESET (segmento FFFFh). Como comentamos en la explicación de RGU (n1/r1), este segmento podemos modificarlo cambiando la constante de configuración "segment_ini" en el package del sistema (n0/r0), en este caso, el microprocesador comienza a ejecutar instrucciones en otro segmento y deberemos modificar la posición de "ini_segment".

- La definición de los segmentos de interrupción. Como podemos observar en la figura 1, existen 5 vectores a los que el microprocesador accede de forma intrínseca al darse una serie de condiciones internas. Estos vectores deben estar programados, de lo contrario provocaríamos la pérdida de control de flujo de programa cuando el microprocesador reaccionara ante esa situación interna especial que puede aparecer durante el transcurso de cualquier programa. El puntero o vector de interrupción, está formado por 4 bytes que definen el segmento de código (CS) y puntero de instrucciones (IP) donde se encuentra el programa de servicio para esa interrupción (ver la figura 1). Aquí definí un segmento para localizar los servicios de las interrupciones (segmento 0EFFh) y establecí un servicio de interrupción cuyo código no supere los 256 bytes. De esta manera para la interrupción 0, el programa del servicio lo encontraremos en el rango de direcciones lógicas (CS:IP) comprendido entre (0EFFF:0000) y (0EFFF:00FF). en la tabla 1 podemos ver el vector y el rango de direcciones asociado a cada interrupción:

Tipo	Dirección del Vector	Descripción	Direcciones del Servicio
0	0000:0000	Error al Dividir	EFFF:0000- EFFF:00FF
1	0000:0004	Paso Único	EFFF:0100- EFFF:01FF
2	0000:0008	Interrupción NMI	EFFF:0200- EFFF:02FF
3	0000:000C	Break Point	EFFF:0300- EFFF:03FF
4	0000:0010	Desbordamiento	EFFF:0400- EFFF:04FF

Tab 1.- Vectores de Interrupción del Sistema

- El segmento de servicios de interrupción "intr_serv", (0EFFFh), contiene todos los servicios de las interrupciones de sistema y las que nosotros programemos. En la tabla 1 podemos ver el rango de posiciones que puede ocupar cada servicio de interrupción. Como se puede observar en el código, no hay ninguna acción programada para los servicios de las interrupciones 0/4. El diseñador del programa deberá introducir el programa del servicio si quiere que el micro realice alguna operación. Aunque no haya programada ninguna acción, (a parte de la instrucción NOP que no realiza ninguna acción y está puesta como ejemplo) la instrucción IRET es necesaria para que el programa principal recupere el control. Como mínimo el servicio de interrupción tiene que contener la instrucción IRET.

- Segmento de pila "stack_segment". Reservo 50 bytes para la pila del sistema que he colocado en el segmento 03FFh.

- Segmento de datos (data_segment" que he posicionado en la dirección 0FFFh. En las 3 primeras palabras de este segmento, debemos indicar las direcciones con las que el programa principal cargará los segmentos de datos (DS), extra (ES) y pila (SS). En el código de "so.asm", he establecido estos tres segmentos (y el de código) donde me ha parecido bien, el programador puede posicionar los segmentos donde le plazca, pero debe modificar estos datos para que después de las primeras instrucciones, los registros de segmento apunten a las posiciones adecuadas. Después de estos datos obligatorios para que el sistema funcione bien, el programador puede posicionar los datos del programa.

- Segmento extra "extra_segment". Posicionado en la página 02FFh, contiene los datos que el programador quiera posicionar en él.

- Segmento de código "code_segment". Inicializado en el segmento inicial "ini_segment" al valor 3000h. Todo programa que realicemos debemos posicionarlo en esta posición después de las instrucciones de carga de segmentos. Estas instrucciones cogen las direcciones de los segmentos extra y de pila almacenadas en el segmento de datos y las cargan en los registros ES ySS respectivamente. Para acceder al segmento de datos, la posición inicial de este debe estar cargada en el registro DS. De ello se encargan las dos primeras instrucciones.

Al ser estos modelos compatibles al 100% con el juego de instrucciones (y el harwdare) de Intel, aseguramos el correcto funcionamiento de todos los programas diseñados utilizando el código máquina de Intel (el conocido "Intel Hexadecimal OP Code"). Más de 20 años de programación pueden "correr" en estos modelos. Esta gran bentaja se extiende también a la utilización de programas que genran código Intel Hexadecimal, es decir, podemos utilizar cualquier compilador que genere código Intel Hexadecimal para generar nuestros propios programas. De esta manera, utilizando Pascal, C, Ensamblador, Basic,... y sus compiladores para PC, podemos generar código que funcionará corréctamente en los modelos diseñados en este proyecto evitándonos la ardua tarea de gerear código máquina "a mano".

Como lenguaje de programación base he utilizado ENSAMBLADOR. Las razones son obias, este lenguaje tiene un mnemónico o símbolo por instrucción del microprocesador, lo que permite controlar en todo momento el código compilado en Intel Hexadecimal y el comportamiento del microprocesaror en relación con el programa que estamos ejecutando. Esto no ocurre con C, por ejemplo, que como veremos más adelante genera una serie de instrucciones Intel Hexadecimal por insctrucción escrita en C.

El compilador utilizado para generar código Intel Hexadecimal es "Macro Assemble, MASM" v6.14.8444 propiedad de Microsoft ®, Programa que podemos encontrad en el CD-ROM del proyecto y que es uno de los más utilizados, junto con "Turbo Assembler, TASM" para el desarrollo de programas en ensamblador. En esta versión de la memoria, no he podido crear un apéndice con una descripción del ensambador y sus pseudo insrtucciones. No obstante, en el CD hay literatura sobre el tema.

A continuación vamos a explicar paso a paso, desde la programación en ensamblador hasta la simulación del programa, dos ejemplos:

- Ejemplo 1. Programa de conversión de binario a ascii utilizando la instrucción AAM o divisiones por 10 sucesivas.
- Ejemplo 2. Conversor BCD a 7 segmentos, un ejemplo clásico de utilización de tablas.

44.1 EJEMPLO 1. CONVERSIÓN DE BINARIO A ASCII.

Es un programa típico de conversión de datos antes de la exhibición de un resultado. Se pueden utilizar dos técnicas:

- La utilización de la instrucción AAM que divide el contenido de AX por 10 guardando el cociente en AH y el resto en AL. El valor de AX debe ser menor de 100d, de lo contrario se desborda la capacidad de AX en la conversión.
- La utilización de divisiones sucesivas (hasta que el cociente es cero) del dato a convertir guardando el cociente en la pila como dígito más significativo en la conversión. Podemos convertir números de cualquier longitud siempre.

En este ejemplo se han programado las dos técnicas.

44.1.2 CÓDIGO DEL PROGRAMA EN ENSAMBLADOR

Para programar en ensamblador, podemos utilizar cualquier editor de textos que genere código ASCII puro sin ningún tipo de formato. Podemos utilizar los editores EDIT, VI o (GVIM), Worpad,...

Para empezar a generar el programa, debemos cargar como "plantilla" el archivo del sistema operativo mínimo "so.asm" explicado anteriormente e insertar el programa, los datos, los servicios y vectores de interrupción, etc ahí donde corresponda. El sistema operativo mínimo diseñado, es un sistema operativo básico apara hacer funcionar los programas, el programador que se familiarice un poco con el entorno, puede modificarlo a su antojo.

A continuación podemos ver íntegramente el código en ensamblador del programa de ejemplo 1 que podemos encontrar en el archivo "prog1.asm" en el CD del proyecto. Los códigos que voy a insertar a continuación son códigos largos, de varias hojas, todos ellos se encuentran en el CD-ROM del proyecto, pese a ello, veo necesario insertarlos en esta memoria para su correcto seguimiento y análisis. En el CD-ROM hay más ejemplos de programas que los vistos en esta sección.

```
;    --=============================================================================--
;    --|PrOGRAMA eJEMPLO:     -cONVERSióN dE BiNARiO a ASCII-                       |--
;    --|                                                                            |--
;    --| La conversión de binario a ASCII se consigue de dos formas. (1) con la    |--
;    --|instrucción AAM si el número es menor de 100 o (2) con una serie de divi-  |--
;    --|siones decimales (dividir entre 10). Ambas técnicas se muestran en el si-  |--
;    --|guiente código.                                                            |--
;    --|----------------------------------------------------------------------------|--
;    --|sEGMENTO iNiCiAL                                                  FFFF:0000 |--
;    --|............................................................................|--
;    --|                                                                            |--
ini segment                              ;Segmento Incicial FFFF:0000
        mov     bx, code                 ;slto al segmento de código
        mov     cs, bx
ini ends
;    --|----------------------------------------------------------------------------|--
;    --|vECTORES dE iNTERRUPCióN                                          0000:0000 |--
;    --|............................................................................|--
int_vect segment                         ;Segmento de vectores de interrupción
0000:0000
        dd  0EFF0000h       ;Vector int 00h => 0EFF:0000 -> Error al Dividir
        dd  0EFF0100h       ;Vector int 01h => 0EFF:0100 -> Paso Único
        dd  0EFF0200h       ;Vector int 02h => 0EFF:0200 -> Terminal MNI
        dd  0EFF0300h       ;Vector int 03h => 0EFF:0300 -> Break Point
        dd  0EFF0400h       ;Vector int 04h => 0EFF:0400 -> Sobreflujo INTO
        org 41h*4h
        dd  0EFF4100h       ;Vector int 41h => 0EFF:4100 -> INTR (recogido del bus)
        org 21h*4h
        dd  0EFF2100h       ;Vector int 21h => 0EFF:2100 -> Interrupción de exhibición
int_vect ends
;    --|----------------------------------------------------------------------------|--
;    --|sERViViOS dE iNTERRUPCióN                                         0EFF:0000 |--
;    --|............................................................................|--
int_serv segment                         ;Segmento de servicios de interrupción
0EFF:0000
    assume ds:data, es:extra, ss:stack
```

```
    assume ds:data, es:extra, ss:stack
;    --|.....................................................................|--
;    --|div0  - int 00h                                                      |--
        org 0000h                ;Servicio de int 00h
        nop                      ;Acciones
        iret                     ;Retorno del servicio de interrupción
;    --|.....................................................................|--
;    --|step  - int 01h                                                      |--
        org 0100h                ;Servicio de int 01h
        nop                      ;Acciones
        iret                     ;Retorno del servicio de interrupción
;    --|.....................................................................|--
;    --|nmi   - int 02h                                                      |--
        org 0200h                ;Servicio de int 02h
        nop                      ;Acciones
        iret                     ;Retorno del servicio de interrupción
;    --|.....................................................................|--
;    --|break - int 03h                                                      |--
        org 0300h                ;Servicio de int 03h
        nop                      ;Acciones
        iret                     ;Retorno del servicio de interrupción
;    --|.....................................................................|--
;    --|over  - int 04h                                                      |--
        org 0400h                ;Servicio de int 04h
        nop                      ;Acciones
        iret                     ;Retorno del servicio de interrupción
;    --|.....................................................................|--
;    --|intr  - int XXh (XXh recogido del bus, 41h)                          |--
        org 4100h                ;Servicio de int 41h
        nop                      ;Acciones
        iret                     ;Retorno del servicio de interrupción
;    --|.....................................................................|--
;    --|int21 - int 21h (MS-DOS)                                             |--
        org 2100h                ;Servicio de int 21h (exhivicion device en IO 0CDh)
        push    ax
        mov     al,dl
        out     0CDh,ax          ;Pongo ax en el dispositivo de entrada salida en 41h
        pop     ax
        iret                     ;Retorno del servicio de interrupción
int_serv ends
;    --|-----------------------------------------------------------------|--
;    --|sEGMENTO dE PiLA                                          03FF:0000 |--
;    --|.....................................................................|--
stack segment                    ;Segmento de Pila 03FF:0000
  db      50 dup(?)              ;reserva 50 bytes
stack ends
;    --|-----------------------------------------------------------------|--
;    --|sEGMENTO dE dATOS                                         0FFF:0000 |--
;    --|.....................................................................|--
data segment                     ;Segmento de datos 0FFF:0000
  data_seg  dw      0FFFh        ;Segmento de Datos
  extra_seg dw      02FFh        ;Segmento de Datos
  stack_seg dw      03FFh        ;Segmento de Datos
  dato_b    db      62h          ;Byte a convertir
  dato_w    dw      0100h        ;Word a convertir
data ends
;    --|-----------------------------------------------------------------|--
;    --|sEGMENTO eXTRA                                            02FF:0000 |--
;    --|.....................................................................|--
extra segment                          ;Segmento extra de datos 02FF:0000
extra ends
;    --|-----------------------------------------------------------------|--
;    --|sEGMENTO dE CóDiGO                                        3000:0000 |--
;    --|.....................................................................|--
code segment                           ;Segmento de Pila 3000:0000
```

```
        assume cs:code, ds:data, es:extra, ss:stack
        mov     ax,data                  ;Cargo los registros de segmento
        mov     ds,ax
        mov     es,extra_seg
        mov     ss,stack_seg

        mov     ax,word ptr dato_b       ;covierte a ASCII 62h -> 98d
        call    desp
        mov     ax,dato_w                ;covierte a ASCII 100h -> 256d
        call    despx
        wait                             ;wait en espera de TESTn
        wait                             ;paro el micro

;    --|.....................................................................|--
;    --|PrOCEDiMiENTO dE cONVERSióN uTiLiZANDO AAM (1)                       |--
;    --| La instrucción AAM convierte el valor que hay en AX en un número de dos   |--
;    --|digitos en BCD no empacado. Si el número en AX es 0062h (98 decimal) antes |--
;    --|de ejecutar AAM, AX contendrá un 0908h después de ejucutar AAM. No se trata|--
;    --|de un código ASCII, sino que se convierte a ASCII al sumarle 3030h. El si- |--
;    --|guiente procedimiento, procesa el valor binario de AL (0 a 99) y lo exibe  |--
;    --|en pantalla (utilizando la int 21h) como decimal. Este procedimiento borra |--
;    --|un cero precedente, lo cual ocurre para los números 0 a 9 con un código es-|--
;    --|pecial ASCII.                                                          |--
;    --|.....................................................................|--

desp    proc    near

        push    dx                       ;salva DX
        xor     ah,ah                    ;borra AH
        aam                              ;convertir a BCD
        add     ah,20h                   ;sumar 20h
        cmp     ah,20h                   ;probar si hay un cero avanzado
        je      desp1                    ;si hay cero inicial
        add     ah,10h                   ;convertir a ASCII

desp1:
        push    ax
        mov     dl,ah                    ;axhibir primer dígito
        mov     ah,6
        int     21h
        pop     ax
        add     al,30h                   ;convertir a ASCII
        mov     dl,al
        mov     ah,6                     ;exhibir segundo dígito
        int     21h
        pop     dx                       ;recupera DX
        ret

desp    endp

;    --|.....................................................................|--
;    --|PrOCEDiMiENTO dE cONVERSióN uTiLiZANDO DiViSiONES sUCESiVAS (2)      |--
;    --| La razón por la que AAM convierte cualquier número entre 0 y 99 a un núme-|--
;    --|ro BCD sin empacar de dos dígitos, es porque divide AX entre 10. El resul- |--
;    --|tado se deja en AX, con lo cual AH contiene el cociente y AL el residuo.   |--
;    --|Este mismo sistema de dividir entre 10 se puede ampliar o expandir para    |--
;    --|convertir cualquier número completo binario a una cadena de caracteres     |--
;    --|ASCII que se puede exhibir en una supuesta pantalla utilizando la interrup-|--
;    --|ción software 21. El algoritmo para convertir de binario a ASCII es:       |--
;    --|  1. Dividir entre 10 y conservar el cociente en la pila como dígito BCD   |--
;    --|     significativo.                                                        |--
;    --|  2. Repetir el paso 1 hasta que el cociente sea un 0.                     |--
;    --|  3. Recuperar cada residuo y sumarle 30h para convertir a ASCII antes de  |--
;    --|     exhibir o imprimir.                                                   |--
;    --| Este procedimiento muestra la forma en la que el contenido de 16 bits sin |--
```

```
;    --|signo de AX se convierte a ASCII y se exhibe en la pantalla. En este caso  |--
;    --|se divide entre 10 y se salva el residuo en la pila después de cada divisi-|--
;    --|ón, para su posterior conversión a ASCII. Después de haber convertido todos|--
;    --|los dígitos, se extraen los residuos de la pila y son convertidos a código |--
;    --|ASCII para exhibir el resultado en la pantalla. Este procedimiento también |--
;    --|borra todos los ceros precedenes que pudiera haber.                        |--
;    --|...........................................................................|--

despx   proc    near

        push    dx                      ;salva BX,CX y DX
        push    cx
        push    bx

        xor     cx,cx                   ;borra CX
        mov     bx,10                   ;carga 10

despx1:
        xor     dx,dx                   ;borra DX
        div     bx
        push    dx                      ;salva el residuo
        inc     cx                      ;contador de residuo
        or      ax,ax                   ;probar el cociente
        jnz     despx1                  ;si no es cero

despx2:
        pop     dx                      ;exhibir número
        add     dl,30h                  ;convertir a ASCII
        mov     ah,6
        int     21h
        loop    despx2                  ;repetir

        pop     dx                      ;recupera BX,CX y DX
        pop     cx
        pop     bx
        ret

despx   endp

code ends
end
;    --|-------------------------------------------------------------------------|--
;    --|                            FiNAL dEL PrOGRAMA                           |--
;    --=========================================================================--
```

Como podemos observar, el código está ampliamente comentado, casi podríamos decir que se autoexplica. Se aprecia claramente que la estructura base del código es la del fichero de sistema operativo, fichero "so.asm". La estructura de los segmentos "ini", "int_vect", "int_serv", "stack" y "extra", apenas cambia con respecto a la original en "so.asm", simplemente aparecen las interrupciones 21h (que emula a la de exhibición de datos que utiliza el DOS) y la interrupción 41 que es el tipo que el generador de interrupciones pone en el bus en el ciclo de reconocimiento de interrupción "intr_ack" (servicio de INTR, ver el modelado del entorno de simulación). Ambas interrupciones efectúan pequeñas modificaciones en "int_vect" e "int_serv". Vemos que en el segmento de datos aparecen los datos a convertir, "dato_b=62h" y "dato_B=100h" y en el segmento de código "code", aparece el cuerpo del programa y los dos procedimientos que implementan los dos algoritmos. Estos dos procedimientos cercanos (dentro del mismo segmento) son llamados desde sendas instrucciones "CALL" en el cuerpo del programa.

44.1.3 COMPILACIÓN DEL FICHERO ASM

Una vez que tenemos escrito el programa en ensamblador, tenemos que pasarlo Intel Hexadecimal, de ello se encarga el compilador MASM. Nosotros no vamos a utilizar el fichero de salida *.obj, sino que utilizaremos la opción /Fl del compilador que genera un fichero de ayuda (con extensión *.lst) donde podemos ver el código en hexadecimal y a la derecha las instrucciones en ensamblador. Esto nos viene muy bien para seguir la conversión que hace MASM, de hecho, este fichero *.lst convenientemente modificado, va a ser el que utilicemos como fichero de entrada para la simulación.

COMPILACIÓN A MANO

El comando del compilador que genera el fichero *.lst es:

```
c:\> ML.EXE /c /Fl prog1.asm ↵
```

Este comando, genera en el mismo directorio de de "prog1.asm" (programa de ejemplo 1 cuyo código podemos ver en las páginas anteriores) un fichero llamado "prog1.lst" y otro llamado "prog1.obj" si la compilación se ha realizado sin fallos. El fichero *obj sólo es útil si vamos a "linkar" (convertir en *.exe o *.com) el programa.

COMPILACIÓN AUTOMATIZADA

Para simplificar la tarea de compilación y mantener una estructura de directorios más limpia, he creado un archivo por lotes (hex.bat) con ayudas que puede hacer más fácil la compilación. El archivo podemos encontrarlo dentro de directorio "assembler" en el CD. Ejecutándolo nos aparece la ayuda:

```
c:\assembler> hex.bat ↵

 --==========================================================================--
 --| ****ERROR****> TiENES QuE eSPECiFiCAR uN aRCHiVO <****                 |--
 --==========================================================================--
 --|          aRCHiVO PoR lOTES PaRA GeNERAR eL aRCHiVO iNTEL hEX           |--
 --|-------------------------------------------------------------------------|--
 --| fORMATO:                                                                |--
 --|                         hex.bat programa.asm                            |--
 --|                           |          |                                  |--
 --|                           |          |----Programa de entrada           |--
 --|                           |---------------Archivo por Lotes             |--
 --|.........................................................................|--
 --| nOTAS: * Este Compilador (Macro Assembler v6.14) genera, a partir de un |--
 --|          fichero de entrada escrito en ensamblador, un fichero hexadeci-|--
 --|          mal en el formato comprensible por los micos de Intel.         |--
 --|        * Nosotros vamos a utilizar el fichero *.lst en el que podemos   |--
 --|          ver el código generado y el fichero fuente.                    |--
 --|        * Es obligatorio poner la extensión en el fichero de entrada.    |--
 --|        * El fichero de entrada "programa.asm" debe encontrarse en .\asm |--
 --|        * El fichero de salida "programa.lst" aparecerá en .\lst         |--
 --|        * El compilador tiene un bug en los tabuladores que hace que el  |--
 --|          fichero de salida pierda un poco el formato.                   |--
 --|        * Para mas información sobre el Compilador teclear "ML /help"    |--
 --==========================================================================--

c:\assembler>_
```

FICHERO DE LISTADO *.LST

Este fichero, contiene el código Intel hexadecimal y el programa compilado de manera que nos permite ver a simple vista la conversión realizada. Información que nos será de gran utilidad. Además podemos ver los comentarios que escribimos en el fichero *.asm que facilitan bastante el seguimiento del código.

En el directorio .\lst encontraremos el fichero "prog1.lst" cuyo código podemos ver a continuación:

Para generar el archivo de listado de forma automatizada, simplemente tenemos que introducir en la línea de comandos:

```
c:\assembler> hex.bat prog1.asm ↵
Microsoft (R) Macro Assembler Version 6.14.8444
Copyright (C) Microsoft Corp 1981-1997.  All rights reserved.

 Assembling: ..\asm\prog1.asm

--=========================================================--
--|>lOS FiCHEROS dE sALiDA eSTßN eN eL DiRECTORiO .\LST<|--
--=========================================================--
c:\assembler>_
```

```
0.                      Microsoft (R) Macro Assembler Version 6.14.8444         05/17/04 20:34:06
1.                      ..\asm\prog1.asm                          Page 1 - 1
2.                      ;    --=============================================================================--
3.                      ;    --|PrOGRAMA eJEMPLO:     -cONVERSiON dE BiNARiO a ASCII-                      |--
4.                      ;    --|                                                                            |--
5.                      ;    --| La conversión de binario a ASCII se consigue de dos formas. (1) con la    |--
6.                      ;    --|instrucción AAM si el número es menor de 100 o (2) con una serie de divi-  |--
7.                      ;    --|siones decimales (dividir entre 10). Ambas técnicas se muestran en el si-  |--
8.                      ;    --|guiente código.                                                            |--
9.                      ;    --|----------------------------------------------------------------------------|--
10.                     ;    --|sEGMENTO iNiCiAL                                                  FFFF:0000 |--
11.                     ;    --|..........................................................................|--
12.                     ;    --|                                                                            |--
13.     0000                ini segment                              ;Segmento Incicial FFFF:0000
14.     0000  BB ---- R            mov     bx, code                 ;slto al segmento de código
15.     0003  8E CB                mov     cs, bx
16.                       ini ends
17.                     ;    --|----------------------------------------------------------------------------|--
18.                     ;    --|vECTORES dE iNTERRUPCiON                                          0000:0000 |--
19.                     ;    --|..........................................................................|--
20.     0000                int_vect segment               ;Segmento de vectores de interrupción 0000:0000
21.     0000  0EFF0000             dd  0EFF0000h          ;Vector int 00h => 0EFF:0000 -> Error al Dividir
22.     0004  0EFF0100             dd  0EFF0100h          ;Vector int 01h => 0EFF:0100 -> Paso Único
23.     0008  0EFF0200             dd  0EFF0200h          ;Vector int 02h => 0EFF:0200 -> Terminal MNI
24.     000C  0EFF0300             dd  0EFF0300h          ;Vector int 03h => 0EFF:0300 -> Break Point
25.     0010  0EFF0400             dd  0EFF0400h          ;Vector int 04h => 0EFF:0400 -> Sobreflujo INTO
26.                              org 41h*4h
27.     0104  0EFF4100             dd  0EFF4100h          ;Vector int 41h => 0EFF:4100 -> INTR (recogido del bus)
28.                              org 21h*4h
29.     0084  0EFF2100             dd  0EFF2100h          ;Vector int 21h => 0EFF:2100 -> Interrupción de exhibición
30.                       int_vect ends
31.                     ;    --|----------------------------------------------------------------------------|--
32.                     ;    --|sERViViOS dE iNTERRUPCiON                                         0EFF:0000 |--
33.                     ;    --|..........................................................................|--
34.     0000                int_serv segment               ;Segmento de servicios de interrupción 0EFF:0000
35.                              assume ds:data, es:extra, ss:stack
36.                     ;    --|..........................................................................|--
37.                     ;    --|div0  - int 00h                                                           |--
38.                              org 0000h              ;Servicio de int 00h
39.     0000  90                   nop                    ;Acciones
40.     0001  CF                   iret                   ;Retorno del servicio de interrupción
41.                     ;    --|..........................................................................|--
42.                     ;    --|step  - int 01h                                                           |--
43.                              org 0100h              ;Servicio de int 01h
44.     0100  90                   nop                    ;Acciones
45.     0101  CF                   iret                   ;Retorno del servicio de interrupción
46.                     ;    --|..........................................................................|--
47.                     ;    --|nmi   - int 02h                                                           |--
48.                              org 0200h              ;Servicio de int 02h
49.     0200  90                   nop                    ;Acciones
50.     0201  CF                   iret                   ;Retorno del servicio de interrupción
51.                     ;    --|..........................................................................|--
52.                     ;    --|break - int 03h                                                           |--
53.                              org 0300h              ;Servicio de int 03h
54.     0300  90                   nop                    ;Acciones
55.     0301  CF                   iret                   ;Retorno del servicio de interrupción
56.                     ;    --|..........................................................................|--
57.                     ;    --|over  - int 04h                                                           |--
58.                              org 0400h              ;Servicio de int 04h
59.     0400  90                   nop                    ;Acciones
60.     0401  CF                   iret                   ;Retorno del servicio de interrupción
61.                     ;    --|..........................................................................|--
62.                     ;    --|intr  - int XXh (XXh recogido del bus, 41h)                               |--
63.                              org 4100h              ;Servicio de int 41h
64.     4100  90                   nop                    ;Acciones
65.     4101  CF                   iret                   ;Retorno del servicio de interrupción
66.                     ;    --|..........................................................................|--
67.                     ;    --|int21 - int 21h (MS-DOS)                                                  |--
68.                              org 2100h              ;Servicio de int 21h (exhivicion device en IO 0CDh)
69.     2100  50                   push    ax
70.     2101  8A C2                mov     al,dl
71.     2103  E7 CD                out     0CDh,ax        ;Pongo ax en el dispositivo de entrada salida en 41h
72.     2105  58                   pop     ax
73.     2106  CF                   iret                   ;Retorno del servicio de interrupción
74.     int_serv ends
```


...sigue

continua...

```
76.                     ;    --|-----------------------------------------------------------------------|--
77.                     ;    --|sEGMENTO dE PiLA                                                03FF:0000 |--
78.                     ;    --|.......................................................................|--
79.     0000               stack segment                          ;Segmento de Pila 03FF:0000
80.     0000  0032 [         db      50 dup(?)                    ;reserva 50 bytes
81.     00
82.               ]
83.                       stack ends
84.                     ;    --|-----------------------------------------------------------------------|--
85.                     ;    --|sEGMENTO dE dATOS                                                0FFF:0000 |--
86.                     ;    --|.......................................................................|--
87.     0000               data segment                           ;Segmento de datos 0FFF:0000
88.     0000 0FFF            data_seg  dw      0FFFh               ;Segmento de Datos
89.     0002 02FF            extra_seg dw      02FFh               ;Segmento de Datos
90.     0004 03FF            stack_seg dw      03FFh               ;Segmento de Datos
91.     0006 62              dato_b    db      62h                 ;Byte a convertir
92.     0007 0100            dato_w    dw      0100h               ;Word a convertir
93.                       data ends
94.                     ;    --|-----------------------------------------------------------------------|--
95.                     ;    --|sEGMENTO eXTRA                                                   02FF:0000 |--
96.                     ;    --|.......................................................................|--
97.     0000               extra segment                          ;Segmento extra de datos 02FF:0000
98.                       extra ends
99.                     ;    --|-----------------------------------------------------------------------|--
100.                    ;    --|sEGMENTO dE CóDiGO                                               3000:0000 |--
101.                    ;    --|.......................................................................|--
102.    0000               code segment                           ;Segmento de Pila 3000:0000
103.                             assume cs:code, ds:data, es:extra, ss:stack
104.    0000  B8 ---- R            mov     ax,data                 ;Cargo los registros de segmento
105.    0003  8E D8                mov     ds,ax
106.    0005  8E 06 0002 R         mov     es,extra_seg
107.    0009  8E 16 0004 R         mov     ss,stack_seg

108.    000D  A1 0006 R            mov     ax,word ptr dato_b      ;covierte a ASCII 62h -> 98d
109.    0010  E8 0008              call    desp
110.    0013  A1 0007 R            mov     ax,dato_w               ;covierte a ASCII 100h -> 256d
111.    0016  E8 0024              call    despx
112.    0019  9B                   wait                            ;wait en espera de TESTn
113.    001A  9B                   wait                            ;paro el micro

114.                    ;    --|.......................................................................|--
115.                    ;    --|PrOCEDiMiENTO dE cONVERSióN uTiLiZANDO AAM (1)                       |--
116.                    ;    --| La instrucción AAM convierte el valor que hay en AX en un número de dos   |--
117.                    ;    --|digitos en BCD no empacado. Si el número en AX es 0062h (98 decimal) antes |--
118.                    ;    --|de ejecutar AAM, AX contendrá un 0908h después de ejucutar AAM. No se trata|--
119.                    ;    --|de un código ASCII, sino que se convierte a ASCII al sumarle 3030h. El si- |--
120.                    ;    --|guiente procedimiento, procesa el valor binario de AL (0 a 99) y lo exibe  |--
121.                    ;    --|en pantalla (utilizando la int 21h) como decimal. Este procedimiento borra |--
122.                    ;    --|un cero precedente, lo cual ocurre para los números 0 a 9 con un código es-|--
123.                    ;    --|pecial ASCII.                                                           |--
124.                    ;    --|.......................................................................|--

125.    001B               desp    proc    near

126.    001B  52                   push    dx                      ;salva DX
127.    001C  32 E4                xor     ah,ah                   ;borra AH
128.    001E  D4 0A                aam                             ;convertir a BCD
129.    0020  80 C4 20             add     ah,20h                  ;sumar 20h
130.    0023  80 FC 20             cmp     ah,20h                  ;probar si hay un cero avanzado
131.    0026  74 03                je      desp1                   ;si hay cero inicial
132.    0028  80 C4 10             add     ah,10h                  ;convertir a ASCII
133.    002B               desp1:
134.    002B  50                   push    ax
135.    002C  8A D4                mov     dl,ah                   ;axhibir primer dígito
136.    002E  B4 06                mov     ah,6
137.    0030  CD 21                int     21h
138.    0032  58                   pop     ax
139.    0033  04 30                add     al,30h                  ;convertir a ASCII
140.    0035  8A D0                mov     dl,al
141.    0037  B4 06                mov     ah,6                    ;exhibir segundo dígito
142.    0039  CD 21                int     21h
143.    003B  5A                   pop     dx                      ;recupera DX
144.    003C  C3                   ret

145.    003D               desp    endp

146.                    ;    --|.......................................................................|--
147.                    ;    --|PrOCEDiMiENTO dE cONVERSióN uTiLiZANDO DiViSiONES sUCESiVAS (2)        |--
148.                    ;    --| La razón por la que AAM convierte cualquier número entre 0 y 99 a un núme-|--
149.                    ;    --|ro BCD sin empacar de dos dígitos, es porque divide AX entre 10. El resul- |--
150.                    ;    --|tado se deja en AX, con lo cual AH contiene el cociente y AL el residuo.   |--
151.                    ;    --|Este mismo sistema de dividir entre 10 se puede ampliar o expandir para    |--
152.                    ;    --|convertir cualquier número completo binario a una cadena de caracteres     |--
153.                    ;    --|ASCII que se puede exhibir en una supuesta pantalla utilizando la interrup-|--
154.                    ;    --|ción software 21. El algoritmo para convertir de binario a ASCII es:       |--
155.                    ;    --|  1. Dividir entre 10 y conservar el cociente en la pila como dígito BCD   |--
156.                    ;    --|     significativo.                                                       |--
157.                    ;    --|  2. Repetir el paso 1 hasta que el cociente sea un 0.                    |--
158.                    ;    --|  3. Recuperar cada residuo y sumarle 30h para convertir a ASCII antes de  |--
159.                    :    --|     exhibir o imprimir.                                                  |--
```

continua...

```
                    ;    --| Este procedimiento muestra la forma en la que el contenido de 16 bits sin |--
                    ;    --|signo de AX se convierte a ASCII y se exhibe en la pantalla. En este caso  |--
                    ;    --|se divide entre 10 y se salva el residuo en la pila después de cada divisi-|--
                    ;    --|ón, para su posterior conversión a ASCII. Después de haber convertido todos|--
                    ;    --|los dígitos, se extraen los residuos de la pila y son convertidos a código |--
                    ;    --|ASCII para exhibir el resultado en la pantalla. Este procedimiento también |--
                    ;    --|borra todos los ceros precedenes que pudiera haber.                       |--
                    ;    --|.........................................................................|--

003D                despx   proc    near

003D  52                    push    dx                      ;salva BX,CX y DX
003E  51                    push    cx
003F  53                    push    bx

0040  33 C9                 xor     cx,cx                   ;borra CX
0042  BB 000A               mov     bx,10                   ;carga 10

                      despx1:
0045  33 D2                 xor     dx,dx                   ;borra DX
0047  F7 F3                 div     bx
0049  52                    push    dx                      ;salva el residuo
004A  41                    inc     cx                      ;contador de residuo
004B  0B C0                 or      ax,ax                   ;probar el cociente
004D  75 F6                 jnz     despx1                  ;si no es cero

004F                despx2:
004F  5A                    pop     dx                      ;exhibir número
0050  80 C2 30              add     dl,30h                  ;convertir a ASCII
0053  B4 06                 mov     ah,6
0055  CD 21                 int     21h
0057  E2 F6                 loop    despx2                  ;repetir

0059  5A                    pop     dx                      ;recupera BX,CX y DX
005A  59                    pop     cx
005B  5B                    pop     bx
005C  C3                    ret

005D                despx   endp

005D                code ends
                    end
                    ;    --|-----------------------------------------------------------------------|--
                    ;    --|                              FiNAL dEL PrOGRAMA                       |--
                    ;    --=========================================================================--


Microsoft (R) Macro Assembler Version 6.14.8444            05/17/04 20:34:06
..\asm\prog1.asm                        Symbols 2 - 1




Segments and Groups:

N a m e                             Size     Length   Align   Combine Class

code . . . . . . . . . . . . . .    16 Bit   005D     Para    Private
data . . . . . . . . . . . . . .    16 Bit   0009     Para    Private
extra  . . . . . . . . . . . . .    16 Bit   0000     Para    Private
ini  . . . . . . . . . . . . . .    16 Bit   0005     Para    Private
int_serv . . . . . . . . . . . .    16 Bit   4102     Para    Private
int_vect . . . . . . . . . . . .    16 Bit   0108     Para    Private
stack  . . . . . . . . . . . . .    16 Bit   0032     Para    Private


Procedures,  parameters and locals:

N a m e                           Type     Value    Attr

despx  . . . . . . . . . . . . .  P Near   003D     code  Length= 0020 Public
despx1 . . . . . . . . . . . . .  L Near   0045     code
despx2 . . . . . . . . . . . . .  L Near   004F     code
desp . . . . . . . . . . . . . .  P Near   001B     code  Length= 0022 Public
desp1  . . . . . . . . . . . . .  L Near   002B     code


Symbols:

N a m e                             Type     Value    Attr

data_seg . . . . . . . . . . . .    Word     0000     data
dato_b . . . . . . . . . . . . .    Byte     0006     data
dato_w . . . . . . . . . . . . .    Word     0007     data
extra_seg  . . . . . . . . . . .    Word     0002     data
stack_seg  . . . . . . . . . . .    Word     0004     data

0 Warnings
0 Errors
```

41.1.4 GENERACIÓN DEL ARCHIVO DE SIMULACIÓN "download_filename"

El último paso para generar el archivo que utilizaremos para la simulación, es realizar unas pequeñas modificaciones en el archivo *.lst generado por el compilador MASM y que podemos ver en las páginas anteriores. Como se puede observar en ese código, he numerado todas las líneas con la intención de indicar en esta sección los cambios que hay que realizar.

El archivo convenientemente modificado, lo salvaremos como "prog1.hex", que será el archivo que debemos especificar en la GENERIC "download_filename" de configuración de los sistemas de simulación (ver la figura 3 de los bancos de test para la simulación). Este archivo es el que el cargador coloca en la memoria del sistema (ver la explicación del modelado de la memoria).

Básicamente tenemos que realizar tres tipos de cambios que explicaremos a continuación.

1. Localización del Programa. Definición de los Segmentos

Como se explicó en la "relocalización dinámica del código", la segmentación del código, permite posicionarlo en la/s parte/s de memoria que queramos. Esto confiere una gran potencia al sistema, pero tiene el inconveniente de que sólo sabremos dónde está posicionado el código en tiempo de ejecución. Para posicionar el código en la memoria, el sistema operativo tienen una tabla con las posiciones de memoria que están libres (estoy hablando de un sistema operativo real como por ejemplo el MS-DOS). Cuando escribimos el nombre de un archivo ejecutable el la línea de comandos del MS-DOS, el intérprete de comandos ("command.com" junto con el archivo de sistema "io.sys"), buscan en una tabla las posiciones de memoria libres y cargan los segmentos del programa en ellas, luego asigna a los registros de segmento la posición de memoria dónde ha grabado los distintos segmentos del código. Esta asignación de los registros de segmento que tan bien realiza el sistema operativo automáticamente, debemos hacerla nosotros manualmente.

Como se puede observar en el código de "prog1.lst" (líneas 14 y 104) en la parte de código donde tiene que poner un puntero al segmento de código (code) o datos (data), no pone ningún dato salvo "----" para indicar que tiene que haber un dato de 16 bits. Se trata de partes del código que son actualizadas en tiempo de ejecución y por lo tanto no se sabe su valor hasta que el programa está completamente cargado en la memoria. Estos datos actualizados en tiempo de ejecución, son los registros de segmento. En las líneas 200-207 de "prog1.lst", podemos ver todos los "simbolos" utilizados en el programa que se actualizan en tiempo de ejecución.

Pues bien, estos valores los debemos introducir a mano, como se indica en la siguiente tabla:

LÍNEA	OLD(16bits)	NEW(16bits)	SIMBOLO
Posiciones Donde Se Va A Grabar El Contenido Del Segmento			
13	0000	%FFFF	ini
20	0000	%0000	int_vect
34	0000	%0EFF	int_serv
79	0000	%03FF	stack
87	0000	%0FFF	data
97	0000	%02FF	extra
102	0000	%3000	code
Punteros Que Se Actualizan En Tiempo De Ejecución			
14	----	0030	code
104	----	FF0F	data

Tab 2.- Cambios en los símbolos actualizados en tiempo de ejecución

Se puede observar que he definido los segmentos donde me ha dado la gana , de manera que tenemos segmentos dispersos por toda la memoria. Si queremos poner todo el código seguido, podemos utilizar la información que nos proporciona el compilador en la tabla de las líneas 200-207 (campo Length) y poner el código de los segmentos code, data, extra y stack en posiciones de memoria consecutivas.

Por ejemplo (ver la figura 1a), queremos posicionar el programa a partir de la página 100 (segmento 0100h), el primer segmento será (por ejemplo) el de código por lo que al símbolo "code" le asignaremos el valor "0001", seguidamente colocaremos el segmento de datos que como el segmento de código ocupar 5Dh bytes, posicionaremos en la página más cercana (segmento 0160h) asignando al símbolo "data" el valor "6001", finalmente e ignorando "extra" (que no es utilizado), posicionaremos la pila en la página más cercana (segmento 0170) y asignaremos "stack" "7001".

Los segmentos ini y int_vect no podemos cambiarlos de lugar (ini si que podemos si cambiamos la constante de configuración "ini_segment" en el package) y el segmento int_serv está en función de los vectores de interrupción del segmento int_vect.

Existe otra técnica de localización de código (válida para programas que ocupan menos de 64K) que consiste en establecer la posición inicial del segmento de código, de datos y de pila en el mismo lugar y utilizar desplazamientos de 16 bits. De esta manera evitamos dejar posiciones de memoria libres entre páginas. Ver la figura 1 b.

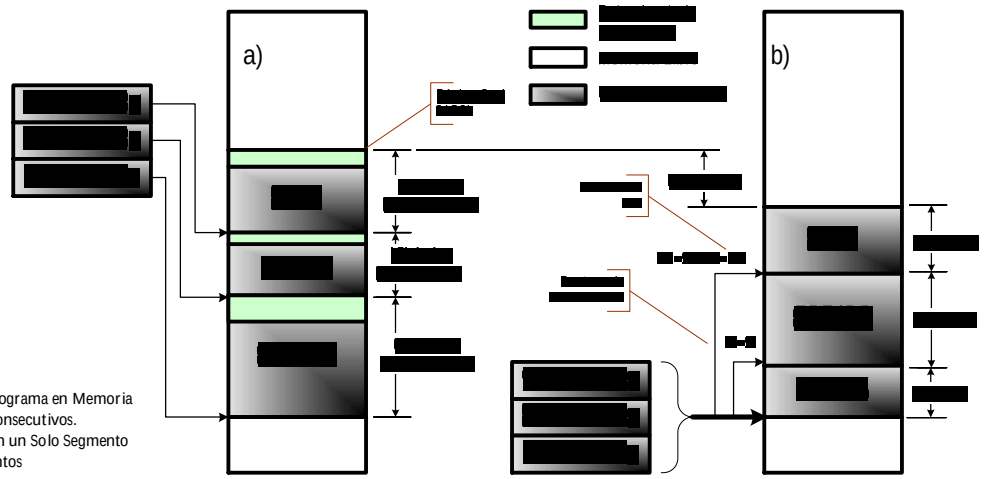


Fig 1.- a) Carga de un Programa en Memoria Utilizando Segmentos Consecutivos.
b) Carga del Programa en un Solo Segmento Utilizando Desplazamientos

2. Cambio de los Octetos

El fichero *.lst generado por MASM, es muy explícito, lo que nos viene muy bien para interpretar el código. En el fichero "prog1.lst", podemos ver el formato del código Intel Hexadecimal que el compilador escribe. Estéticamente es muy bonito, separando el desplazamiento del OP code de la instrucción y éste de los datos inmediatos o desplazamiento o ambos (de existir). Los primeros 4 bytes son el desplazamiento de la instrucción dentro del segmento en el que está dicha instrucción. Los siguientes bytes (1 o 2) separados entre sí definen el código hexadecimal de la instrucción (ver la tabla de códigos op en el apéndice A). Después de estos bytes y separado por un espacio, aparece el desplazamiento (de existir), este desplazamiento puede ser corto (1 byte), cercano (2 bytes) o lejano (4 bytes). En los dos últimos casos el desplazamiento aparece de forma "bonita", es decir, lo pone en la forma que nosotros mejor lo leemos. Lo mismo hace con los datos inmediatos de 2 bytes y los datos que son de más de 1 byte.

Esta forma de presentar el código es muy bonita, pero no es la real, en realidad el byte menos significativo de un dato de más de 1 byte tiene que estar localizado en la menor posición de memoria y el byte de mayor peso en la posición de memoria más alta.

El cargador de memoria (ver el modelado de la memoria) lee secuencialmente los datos de la línea del archivo y coloca el primer byte encontrado en la posición indicada por el offset, el segundo en la posicón del Off-Set+1 y el tercero en Off-Set+3, así sucesivamente. Esta carga secuencial de un dato de más de 8 bits del archivo *.lst, provocaría un error, ver la figura 2. Para solucionar este "bug" en el fichero de ayua generado por MASM, debemos reescribir todos los datos (inmediatos o no) y desplazamientos si ambos son de más de 1 byte. Es una tarea sencilla que no lleva más de 1 minuto en un programa más o menos largo como este.

Los datos rescritos en el programa son los siguientes:

LÍNEA	BYTES	OLD	NEW(LSB..MSB)	SIMBOLO
Segmento de Vecores de Interrupción				
21	4	0EFF0000	0000FF0E	ini
22	4	0EFF0100	0001FF0E	int_vect
23	4	0EFF0200	0002FF0E	int_serv
24	4	0EFF0300	0003FF0E	stack
25	4	0EFF0400	0004FF0E	data
27	4	0EFF4100	0041FF0E	extra
29	4	0EFF2100	0021FF0E	code
Segmento de Datos				
88	2	0FFF	FF0F	data_seg
89	2	02FF	FF02	extra_seg
90	2	03FF	FF03	stack_seg
92	2	0100	0001	dato_w
Desplazamientos y Datos Inmediatos				
106	2	0002	0200	ptr extra_seg
107	2	0004	0400	ptr stack_seg
108	2	0006	0600	dato_b
109	2	0008	0800	ptr desp
110	2	0007	0700	dato_w
111	2	0024	2400	ptr despx
173	2	000A	0A00	imm 10d

Tab 3.- Cambios en los Octetos de los Datos y Desplazamientos.

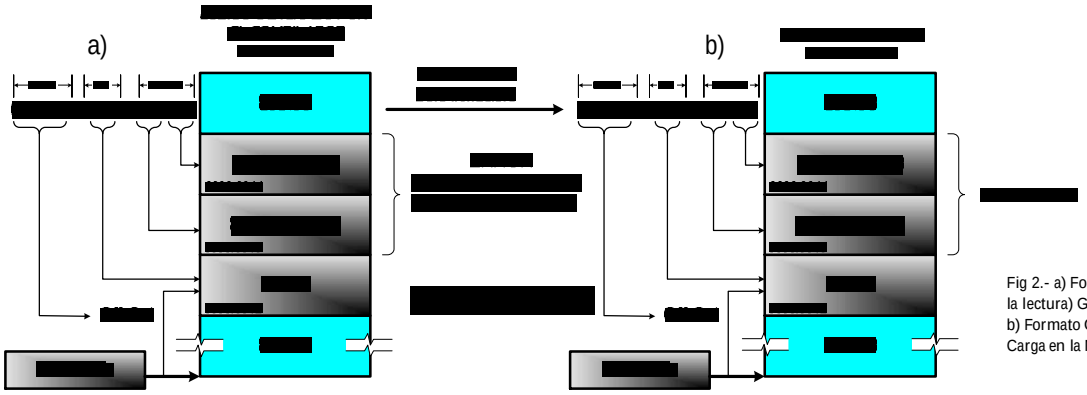


Fig 2.- a) Formato del Código (para la lectura) Generado por MASM
b) Formato OP Corregido para la Carga en la Memoria. Línea 173.

3. Eliminación de Líneas No Útiles

El archivo que genera el compilador MASM, crea en la cabecera y en pie del archivo *.lst una serie de líneas informativas. Esta información, sobretodo la que encontramos al final del fichero, pueden ser muy útiles para seguir el código y la forma en la que el compilador genera el código.

- En las líneas 199-207 aparece una tabla que nos informa de los segmentos definidos en el código, indicándonos el número de octetos que ocupa cada segmentos (campo Length). Podemos percatarnos que compilador por defecto inicia los segmentos en posiciones pares de la memoria física (Align=Para) asegurando de esta manera que la lectura en el modelo 8086 se realizara en un solo ciclo de bus. Ver la explicación de la unidad DFU (n3/r4).
- En las líneas 208-214 podemos ver información sobre los Procedimientos y etiquetas utilizadas. En este programa de ejemplo he programado cada algoritmo de conversión de binario a ASCII en dos procedimientos (desp y despx) que se encuentran en el mismo segmento de código que la llamada (procedimientos cercanos, NEAR). El campo "valor" contiene el desplazamiento (en bytes) al primer byte del proceso desde el inicio del segmento de código. El campo "Length" indica el número de bytes que contiene cada procedimiento.
- Por último, en las líneas 215-221, hay una tabla con los nombres definidos en los segmentos de datos. Estos nombres o símbolos son utilizados para manejar el puntero al dato de una forma más pragmática que utilizando el valor del puntero. En el campo "Value" podemos ver el valor del puntero o desplazamiento desde el inicio del segmento de datos hasta la celda donde se encuentra o hay que posicionar el dato en cuestión. El "Type" define el número de bytes del valor del dato, es importante tener que para cargar una palabra desde una posición que hemos definido como byte, es necesario indicárselo al compilador con la pseudoinstrucción deel compilador "word ptr" (ver la línea 108).

Aunque el cargador ignora estas líneas, recomiendo eliminarlas (lineas 0,1,80-82 y 197-223) del fichero final "prog1.hex" cuyo código final podemos ver en las siguientes páginas. Es necesario eliminar las líneas 80-82 para evitar que el cargador inserte en el segmento de pila el dato 0032. Aunque este dato es sobrescrito por la primera instrucción que utiliza la pila y no afecta al funcionamiento, no es un dato que deba escribirse en la pila.

```
2.                      ;    --=================================================================================--
3.                      ;    --|PrOGRAMA eJEMPLO:     -cONVERSiON dE BiNARiO a ASCII-                        |--
4.                      ;    --|                                                                                |--
5.                      ;    --| La conversión de binario a ASCII se consigue de dos formas. (1) con la        |--
6.                      ;    --|instrucción AAM si el número es menor de 100 o (2) con una serie de divi-      |--
7.                      ;    --|siones decimales (dividir entre 10). Ambas técnicas se muestran en el si-      |--
8.                      ;    --|guiente código.                                                               |--
9.                      ;    --|-------------------------------------------------------------------------------|--
10.                     ;    --|sEGMENTO iNiCiAL                                                   FFFF:0000 |--
11.                     ;    --|...............................................................................|--
12.                     ;    --|                                                                               |--
13.     %FFFF                ini segment                                ;Segmento Incicial FFFF:0000
14.     0000  BB 0030 R              mov     bx, code                 ;slto al segmento de código
15.     0003  8E CB                  mov     cs, bx
16.                          ini ends
17.                     ;    --|-------------------------------------------------------------------------------|--
18.                     ;    --|vECTORES dE iNTERRUPCiON                                           0000:0000 |--
19.                     ;    --|...............................................................................|--
20.     %0000                int_vect segment                ;Segmento de vectores de interrupción 0000:0000
21.     0000  0000FF0E               dd  0EFF0000h           ;Vector int 00h => 0EFF:0000 -> Error al Dividir
22.     0004  0001FF0E               dd  0EFF0100h           ;Vector int 01h => 0EFF:0100 -> Paso Único
23.     0008  0002FF0E               dd  0EFF0200h           ;Vector int 02h => 0EFF:0200 -> Terminal MNI
24.     000C  0003FF0E               dd  0EFF0300h           ;Vector int 03h => 0EFF:0300 -> Break Point
25.     0010  0004FF0E               dd  0EFF0400h           ;Vector int 04h => 0EFF:0400 -> Sobreflujo INTO
26.                                 org 41h*4h
27.     0104  0041FF0E               dd  0EFF4100h           ;Vector int 41h => 0EFF:4100 -> INTR (recogido del bus)
28.                                 org 21h*4h
29.     0084  0021FF0E               dd  0EFF2100h           ;Vector int 21h => 0EFF:2100 -> Interrupción de exhibición
30.                          int_vect ends
31.                     ;    --|-------------------------------------------------------------------------------|--
32.                     ;    --|sERViViOS dE iNTERRUPCiON                                          0EFF:0000 |--
33.                     ;    --|...............................................................................|--
34.     %0EFF                 int_serv segment               ;Segmento de servicios de interrupción 0EFF:0000
35.                                 assume ds:data, es:extra, ss:stack
36.                     ;    --|...............................................................................|--
37.                     ;    --|div0  - int 00h                                                              |--
38.                                 org 0000h              ;Servicio de int 00h
39.     0000  90                     nop                     ;Acciones
40.     0001  CF                     iret                    ;Retorno del servicio de interrupción
41.                     ;    --|...............................................................................|--
42.                     ;    --|step  - int 01h                                                              |--
43.                                 org 0100h              ;Servicio de int 01h
44.     0100  90                     nop                     ;Acciones
45.     0101  CF                     iret                    ;Retorno del servicio de interrupción
46.                     ;    --|...............................................................................|--
47.                     ;    --|nmi   - int 02h                                                              |--
48.                                 org 0200h              ;Servicio de int 02h
49.     0200  90                     nop                     ;Acciones
50.     0201  CF                     iret                    ;Retorno del servicio de interrupción
51.                     ;    --|...............................................................................|--
52.                     ;    --|break - int 03h                                                              |--
53.                                 org 0300h              ;Servicio de int 03h
54.     0300  90                     nop                     ;Acciones
55.     0301  CF                     iret                    ;Retorno del servicio de interrupción
56.                     ;    --|...............................................................................|--
57.                     ;    --|over  - int 04h                                                              |--
58.                                 org 0400h              ;Servicio de int 04h
59.     0400  90                     nop                     ;Acciones
60.     0401  CF                     iret                    ;Retorno del servicio de interrupción
61.                     ;    --|...............................................................................|--
62.                     ;    --|intr  - int XXh (XXh recogido del bus, 41h)                                  |--
63.                                 org 4100h              ;Servicio de int 41h
64.     4100  90                     nop                     ;Acciones
65.     4101  CF                     iret                    ;Retorno del servicio de interrupción
66.                     ;    --|...............................................................................|--
67.                     ;    --|int21 - int 21h (MS-DOS)                                                    |--
68.                                 org 2100h              ;Servicio de int 21h (exhivicion device en IO 0CDh)
69.     2100  50                     push    ax
70.     2101  8A C2                  mov     al,dl
71.     2103  E7 CD                  out     0CDh,ax         ;Pongo ax en el dispositivo de entrada salida en 41h
72.     2105  58                     pop     ax
73.     2106  CF                     iret                    ;Retorno del servicio de interrupción
94.     int_serv ends
76.                     ;    --|-------------------------------------------------------------------------------|--
77.                     ;    --|sEGMENTO dE PiLA                                                   03FF:0000 |--
78.                     ;    --|...............................................................................|--
79.     %03FF                stack segment                          ;Segmento de Pila 03FF:0000
83.                          stack ends
84.                     ;    --|-------------------------------------------------------------------------------|--
85.                     ;    --|sEGMENTO dE dATOS                                                  0FFF:0000 |--
86.                     ;    --|...............................................................................|--
87.     %0FFF                data segment                           ;Segmento de datos 0FFF:0000
88.     0000 0FFF              data_seg  dw      0FFFh              ;Segmento de Datos
89.     0002 02FF              extra_seg dw      02FFh              ;Segmento de Datos
90.     0004 03FF              stack_seg dw      03FFh              ;Segmento de Datos
91.     0006 62                dato_b    db      62h                ;Byte a convertir
92.     0007 0100              dato_w    dw      0100h              ;Word a convertir
93.                          data ends
```


...sigue

continua...

```
                    ;    --|-----------------------------------------------------------------------|--
                    ;    --|sEGMENTO eXTRA                                                   02FF:0000 |--
                    ;    --|.......................................................................|--
%02FF                  extra segment                          ;Segmento extra de datos 02FF:0000
                     extra ends
                    ;    --|-----------------------------------------------------------------------|--
                    ;    --|sEGMENTO dE CóDiGO                                               3000:0000 |--
                    ;    --|.......................................................................|--
%3000                  code segment                           ;Segmento de Pila 3000:0000
                             assume cs:code, ds:data, es:extra, ss:stack
0000  B8 FF0F R          mov     ax,data                      ;Cargo los registros de segmento
0003  8E D8              mov     ds,ax
0005  8E 06 0200 R       mov     es,extra_seg
0009  8E 16 0400 R       mov     ss,stack_seg

000D  A1 0600 R          mov     ax,word ptr dato_b           ;covierte a ASCII 62h -> 98d
0010  E8 0800            call    desp
0013  A1 0700 R          mov     ax,dato_w                    ;covierte a ASCII 100h -> 256d
0016  E8 2400            call    despx
0019  9B                 wait                                 ;wait en espera de TESTn
001A  9B                 wait                                 ;paro el micro

                    ;    --|.......................................................................|--
                    ;    --|PrOCEDiMiENTO dE cONVERSióN uTiLiZANDO AAM (1)                        |--
                    ;    --| La instrucción AAM convierte el valor que hay en AX en un número de dos   |--
                    ;    --|digitos en BCD no empacado. Si el número en AX es 0062h (98 decimal) antes |--
                    ;    --|de ejecutar AAM, AX contendrá un 0908h después de ejucutar AAM. No se trata|--
                    ;    --|de un código ASCII, sino que se convierte a ASCII al sumarle 3030h. El si- |--
                    ;    --|guiente procedimiento, procesa el valor binario de AL (0 a 99) y lo exibe  |--
                    ;    --|en pantalla (utilizando la int 21h) como decimal. Este procedimiento borra |--
                    ;    --|un cero precedente, lo cual ocurre para los números 0 a 9 con un código es-|--
                    ;    --|pecial ASCII.                                                            |--
                    ;    --|.......................................................................|--

001B                   desp    proc    near

001B  52                 push    dx                           ;salva DX
001C  32 E4              xor     ah,ah                        ;borra AH
001E  D4 0A              aam                                  ;convertir a BCD
0020  80 C4 20           add     ah,20h                       ;sumar 20h
0023  80 FC 20           cmp     ah,20h                       ;probar si hay un cero avanzado
0026  74 03              je      desp1                        ;si hay cero inicial
0028  80 C4 10           add     ah,10h                       ;convertir a ASCII
002B                   desp1:
002B  50                 push    ax
002C  8A D4              mov     dl,ah                        ;axhibir primer dígito
002E  B4 06              mov     ah,6
0030  CD 21              int     21h
0032  58                 pop     ax
0033  04 30              add     al,30h                       ;convertir a ASCII
0035  8A D0              mov     dl,al
0037  B4 06              mov     ah,6                         ;exhibir segundo dígito
0039  CD 21              int     21h
003B  5A                 pop     dx                           ;recupera DX
003C  C3                 ret

003D                   desp    endp

                    ;    --|.......................................................................|--
                    ;    --|PrOCEDiMiENTO dE cONVERSióN uTiLiZANDO DiViSiONES sUCESiVAS (2)        |--
                    ;    --| La razón por la que AAM convierte cualquier número entre 0 y 99 a un núme-|--
                    ;    --|ro BCD sin empacar de dos dígitos, es porque divide AX entre 10. El resul- |--
                    ;    --|tado se deja en AX, con lo cual AH contiene el cociente y AL el residuo.   |--
                    ;    --|Este mismo sistema de dividir entre 10 se puede ampliar o expandir para    |--
                    ;    --|convertir cualquier número completo binario a una cadena de caracteres     |--
                    ;    --|ASCII que se puede exhibir en una supuesta pantalla utilizando la interrup-|--
                    ;    --|ción software 21. El algoritmo para convertir de binario a ASCII es:       |--
                    ;    --|  1. Dividir entre 10 y conservar el cociente en la pila como dígito BCD   |--
                    ;    --|     significativo.                                                        |--
                    ;    --|  2. Repetir el paso 1 hasta que el cociente sea un 0.                     |--
                    ;    --|  3. Recuperar cada residuo y sumarle 30h para convertir a ASCII antes de  |--
                    ;    --|     exhibir o imprimir.                                                   |--
                    ;    --| Este procedimiento muestra la forma en la que el contenido de 16 bits sin |--
                    ;    --|signo de AX se convierte a ASCII y se exhibe en la pantalla. En este caso  |--
                    ;    --|se divide entre 10 y se salva el residuo en la pila después de cada divisi-|--
                    ;    --|ón, para su posterior conversión a ASCII. Después de haber convertido todos|--
                    ;    --|los dígitos, se extraen los residuos de la pila y son convertidos a código |--
                    ;    --|ASCII para exhibir el resultado en la pantalla. Este procedimiento también |--
                    ;    --|borra todos los ceros precedenes que pudiera haber.                       |--
                    ;    --|.......................................................................|--

003D                   despx   proc    near

003D  52                 push    dx                           ;salva BX,CX y DX
003E  51                 push    cx
003F  53                 push    bx

0040  33 C9              xor     cx,cx                        ;borra CX
0042  BB 0A00            mov     bx,10                        ;carga 10
```

...sigue

continua...

```
174.    despx1:
175.    0045  33 D2                xor     dx,dx                   ;borra DX
176.    0047  F7 F3                div     bx
177.    0049  52                   push    dx                      ;salva el residuo
178.    004A  41                   inc     cx                      ;contador de residuo
179.    004B  0B C0                or      ax,ax                   ;probar el cociente
180.    004D  75 F6                jnz     despx1                  ;si no es cero

181.    004F               despx2:
182.    004F  5A                   pop     dx                      ;exhibir número
183.    0050  80 C2 30             add     dl,30h                  ;convertir a ASCII
184.    0053  B4 06                mov     ah,6
185.    0055  CD 21                int     21h
186.    0057  E2 F6                loop    despx2                  ;repetir

187.    0059  5A                   pop     dx                      ;recupera BX,CX y DX
188.    005A  59                   pop     cx
189.    005B  5B                   pop     bx
190.    005C  C3                   ret

191.    005D               despx   endp

192.    005D               code ends
193.                    end
194.                    ;    --|-------------------------------------------------------------------------|--
195.                    ;    --|                            FiNAL dEL PrOGRAMA                           |--
196.                    ;    --=========================================================================--
```

44.1.5 LANZAMIENTO DE LA SIMULACIÓN

Para la simulación de los modelos, he utilizado el programa "ModelSim SE/PE v5.5e". Este simulador es el más utilizado para el test en el mundo del diseño VHDL y Verilog, incluso los resultados obtenidos con este simulador son aceptados por algunos fabricantes y programadores de FPGA´s

PASOS PARA LANZAR LA SIMULACIÓN

Para el lanzamiento de la simulación, lógicamente, debemos tener instalado y correctamente configurado ModelSim. Es importante tener las variables de entorno del sistema operativo bien configuradas, sobretodo la de la licencia del programa (LM_LICENSE_FILE) y la de la caminos de búsqueda de ejecutables (PATH). Normalmente el programa de instalación del ModelSim se encarga de configurarlas. No obstante, podemos hacerlo manualmente con el comando "set" desde el prompt del sistema. Ejemplo:

```
c:\> set PHAT=c:\Modeltech\win32;%PHAT% ↵
```

Para más información sobre la configuración de ModelSim, ver los archivos de Ayuda. A continuación veremos paso a paso los comandos que tenemos que introducir en la línea de comandos (ejecutando el intérprete de comandos "cmd" o "command.com" en Windows) para lanzar la simulación. He supuesto que nos encontramos en el directorio donde se encuentran las librerías de los modelos, es decir, haciendo un "dir" veremos los directorios up_808x, up8088, up8086 y up808x_tb. Estas librerías se encuentran en el directorio "sources" del CD.

1.- Una vez que tenemos el programa correctamente instalado y configurado, creamos la librería donde vamos la compilar el modeloen el directorio "sources". Normalmente a esta librería se le pone el nombre: "work". El comando de ModelSim que genera esta librería es:

```
c:\sources\> vlib.exe work ↵
```

Veremos como aparece un nuevo directorio con el nombre "work" y un fichero de información en su interior. Este dichero es utilizado por el simulador para guardar información sobre la compilación.

2.- Tenemos que indicar a ModelSim el nombre de las librerías que vamos a compilar en el directorio "work", lo que se llama "mapear las librerías". Podemos hacerlo de dos maneras:. (a) modificando directamente el fichero "modelsim.ini" en el directorio de instalación de "ModelSim" o (b) utilizando el comando "vmap.exe" de la forma siguiente:

```
c:\sources\>vmap.exe up808x work
c:\sources\>vmap.exe up8088 work
c:\sources\>vmap.exe up8086 work
c:\sources\>vmap.exe up808x_tb work ↵
```

No pretendo dar una clase de manejo de Modelsim, sólo estoy explicando la utilización básica de los comandos para simular el programa. En los documentos que instala el programa hay información detallada sobre el funcionamiento y las opciones de cada comando.

El comando "vmap" lo único que hace es modificar el archivo de inicio de ModelSim "modelsim.ini" de forma automática. Ejecutando "vmap.exe" sin parámetros podemos comprobar como las sibrerías del proyecto se han mapeado en la librería "work".

```
c:\sources\> vmap.exe ↵
Reading: c:\modeltech\win32\..\modelsim.ini
[...]
"up808x" maps to directory work.
"up8088" maps to directory work.
"up8086" maps to directory work.
"up808x_tb" maps to directory work.
```

3.- El siguiente paso es la compilación de las librerías del proyecto en directorio work. Para ello, vamos a utilizar el compilador de ModelSim "vcom.exe". Este comando tiene un montón de opciones de las que nosotros sólo utilizaremos la de soporte para ficheros VHDL escritos utilizando la norma IEEE 1076-93 (opción -93). Esta revisión implementa algunas mejoras respecto a la norma original del 87, mejoras como el control sobre los ficheros y el nombrado de los procesos (entre otras) que he utilizado en la definición del código del proyecto.

Las librerías hay que compilarlas utilizando una orden determinado, concretamente en orden jerárquico inverso, primero las unidades que no tienen ninguna dependencia y luego las unidades que tienen dependencias. Para compilar la unidad GAU n3/r7 (por ejempo), deben estar previamente compiladas las unidades

GAU_ADDER (n4/r6) y GAU_CTRL (n4/r7). Como podemos observar en el diagrama jerárquico de la figura 17, GAU está compuesta por estas dos unidades.

La primera librería que compilaremos, será la librería común es decir, la librería "up808x" y lo haremos con el siguiente comando:

```
c:\sources\>vcom.exe -93
     up808x/pack808x_pkg.vhd
     up808x/up808x_biu_interf_gau_add.vhd
     up808x/up808x_biu_interf_gau_ctrl.vhd
     up808x/up808x_biu_interf_gau.vhd
     up808x/up808x_biu_interf_ifu_flush.vhd
     up808x/up808x_biu_interf_mu.vhd
     up808x/up808x_biu_interf_arbiter.vhd
     up808x/up808x_biu_timing_tsm.vhd
     up808x/up808x_eu_decode.vhd
     up808x/up808x_eu_decode_drom.vhd
     up808x/up808x_eu_exe_cgrom.vhd
     up808x/up808x_eu_exe_pc.vhd
     up808x/up808x_eu_exe_spw.vhd
     up808x/up808x_eu_exe.vhd
     up808x/up808x_eu.vhd
     up808x/up808x_alu.vhd
     up808x/up808x_init.vhd
     up808x/up808x_rgu.vhd
     up808x/up808x_sru.vhd ↵
```

La segunda librería que debemos compilar, es la específica del micro que vamos a simular; la "up8088" si vamos a realizar la simulación con el modelo del micro 8088 en modo máximo o mínimo y la "up8086" si vamos a simular el programa en el 8086 en cualquiera de sus modos de funcionamiento. Suponiendo que vayamos a utilizar el 8088 compilaríamos de la siguiente manera:

```
c:\sources\>vcom.exe -93
     up8088/up8088_biu_interf_dfu_ctrl.vhd
     up8088/up8088_biu_interf_dfu_reg.vhd
     up8088/up8088_biu_interf_dfu.vhd
     up8088/up8088_biu_interf_ifu_ctrl.vhd
     up8088/up8088_biu_interf_ifu_pointer.vhd
     up8088/up8088_biu_interf_ifu.vhd
     up8088/up8088_biu_interf.vhd
     up8088/up8088_biu_timing_bcg.vhd
     up8088/up8088_biu_timing.vhd
     up8088/up8088_biu.vhd
     up8088/up8088_ports_mn.vhd
     up8088/up8088_ports_mx.vhd
     up8088/up8088_queue.vhd
     up8088/up8088mn.vhd
     up8088/up8088mx.vhd ↵
```

Por último, hay que compilar los archivos de la librería del banco de test que vamos a utilizar en la simulación, pero antes debemos configurar las GENERICS del banco de test, y en especial la GENERIC "download_filename" en los archivos "test_*.vhd". Esta variable como se ha explicado en otras ocasiones, contiene el nombre del fichero que vamos a simular. Si grabamos el fichero "prog1.hex" preparado para la simulación, en el directorio "work", deberemos asignar en el PORT MAP (ojo, no de la definición del componente) de las GENERICS, el valor ".\prog1.hex" a la GENERIC "download_filename". También podemos utilizar la táctica de no modificar la GENERIC, asignándole un valor fijo (por ejemplo: download_filename=".\ram.hex") y grabar en el directorio "work" siempre con ese nombre el fichero que queremos simular. De esta manera evitamos tener que compilar el modelo siempre que cambiemos de programa. Si el fichero de simulación lo colocamos en otro directorio distinto a ".\work", deberemos indicar la ruta completa en la GENERIC download_filename.

Una vez que hemos modificado las GENERICS, compilamos los ficheros de simulación. Si vamos a simular el programa en el modelo 8088 en modo mínimo, tendremos que compilar el fichero de generación de vectores "system_8088mn.vhd" y el banco de test "test_8088mn.vhd" con las siguientes líneas:

```
c:\sources\>vcom.exe -93
     up808x_tb/system_8088mn.vhd
     up8088_tb/test_8088mn.vhd ↵
```

4.- Ya tenemos todos los archivos necesarios para la simulación compilados en la librería "work". El último paso es invocar el simulador con el comando "vsim.exe" indicándole el componente de mayor nivel, que en el caso de estar utilizando el banco de test para el 8088 en modo mínimo será "test_8088mn":

```
c:\sources\>vsim.exe test_8088mn ↵
```

Sólo queda cargar las señales que queremos visualizar y lanzar la simulación con el comando "run".

Lanzamiento de la Simulación de Forma Automatizada

Para hacer menos laboriosa la tarea de ejecución de los 4 pasos indicados anteriormente, he programado un fichero por lotes que automatiza todos los pasos de manera que la compilación de los modelos y el lanzamiento de la simulación se hace de una forma muy sencilla. El archivo por lotes que automatiza este proceso, es e archivo "comp.bat" que podemos encontrar en el directorio ".\sourceVHDL" del CD-ROM del proyecto. El archivo tiene una autoayuda que guía al usuario:

```
D:\sourceVHDL>comp.bat ↵

--==============================================================================--
--| ****ERROR****> fALTA eL oPERANDO 1 <****                                   |--
--==============================================================================--
--|  aRCHiVO PoR lOTES PaRA cOMPiLAR i lANZAR lA SiMULACioN dE lOS mODELOS  |--
--|------------------------------------------------------------------------|--
--| fORMATO:                                                               |--
--| comp.bat [8086/8088] [mn/mx] programa.hex                              |--
--|    |          |         |          |                                  |--
--|    |          |         |          |----Programa a simular            |--
--|    |          |         |---------------Modo Maximo/Minimo            |--
--|    |          |-------------------------CPU a compilar 8088/8086       |--
--|    |-----------------------------------Archivo por lotes              |--
--|........................................................................|--
--| nOTAS: * Tienen que Aparecer los 3 parametros obligatoriamente         |--
--|        * Programa.hex es el programa que queremos simular. Tiene que es-|--
--|          tar en el directorio ./programs.                              |--
--|        * Modelsim tiene que estar instalado y las variables de entorno |--
--|          correctamente configuradas.                                   |--
--==============================================================================--

D:\sourceVHDL>_
```

El archivo "comp.bat", compila y lanza el simulador de una forma más limpia y organizada. En el directorio sourceVHDL hay tres subdirectorios:

- programs. Que contiene los archivos de programa que vamos a simular.
- sources. Que contiene los archivos fuente del proyecto.
- modelsim. Que contiene los ficheros con la secuencia para la compilación y unas macros de ModelSim que inicializan la ventana de Formas de Onda..

Para lanzar la simulación del programa de ejemplo 1 ("prog1.hex" que debe encontrarse en el directorio ".\sourceVHDL\programs\") en el microprocesador 8088 en modo mínimo, introduciremos en el prompt el siguiente comando:

```
c:\sources\> comp.bat 8088 mn prog1.hex↵
```

Si todo ha ido bien, aparecerá una pantalla como la que podemos ver a la derecha, donde se puede apreciar que el fichero com.bat lanza el simulador con una macro que activa la pantalla de formas de onda "Wave" y carga unas señales en ella. Se trata de las macros (archivos con extensión *.do) que se encuentran en el directorio ".\modelsim". Estas macros cargan una serie de señales (que podemos ver en la tabla 4) que considero esenciales para seguir el funcionamiento de los programas, dándoles formato y agrupándolas según su función. Existes cuatro macros, wave_88mn.do, wave_88mx.do, wave_86mn.do y wave_86mx.do; según el tipo de sistema que estemos simulando, cargaremos una u otra.

PUERTOS Estas señales son los pines de los microprocesadores. Son el interface con el exterior, y su comportamiento tiene que ser escrupulosamente analizado.	
Señales en Modo Mínimo	Señales en Modo Mínimo
res, reset, ck, as, a, ad, ale, (bhen), rdn, wrn, iomn, ss0n, dt_rn, denn, reary, hold, holda, intr, intan, nmi, testn, mn_mxn	res, reset, ck, as, a, (bhen), ad, rdn, ready, intr, nmi, testn, mn_mxn
	ale,denn, dt_rn, io_mn, wrn, intan
	queue_status, bus_cycle, lockn, rq0ngt0n, rq1ngt1n
SISTEMA Señales de sistema, muestran el estado del bus del sistema y de la BIU, las señales ghost_*, son conversiones de los bits del bus de sistema que nos indican de una forma más didáctica el significado de los bits de sistema	
bus_state, system_bus, ghost_bus_cycle, ghost_segment_used, ghost_queue_status	
REGISTROS Son los registros "transparentes" del microprocesador, a excepción de los registros A?, cualquier programa puede modificar el valor de estos registros. Forman parte del Modelo software de los microprocesadores.	
ax, bx, cx, dx, sp, bp, si, di, cs, ss, ds, es, ip, a1, a2, a3, flags.	
EJECUCIÓN Son las señales más importantes de la unidad de ejecución EU. Permiten saber la situación en la que se encuentra durante toda la simulación del programa	
exe_state, exe_phase, aux_phase, op1, op2	

Tab 4.- Señales cargadas en Wave por la Macros.

En las siguientes páginas veremos el resultado de la simulación.

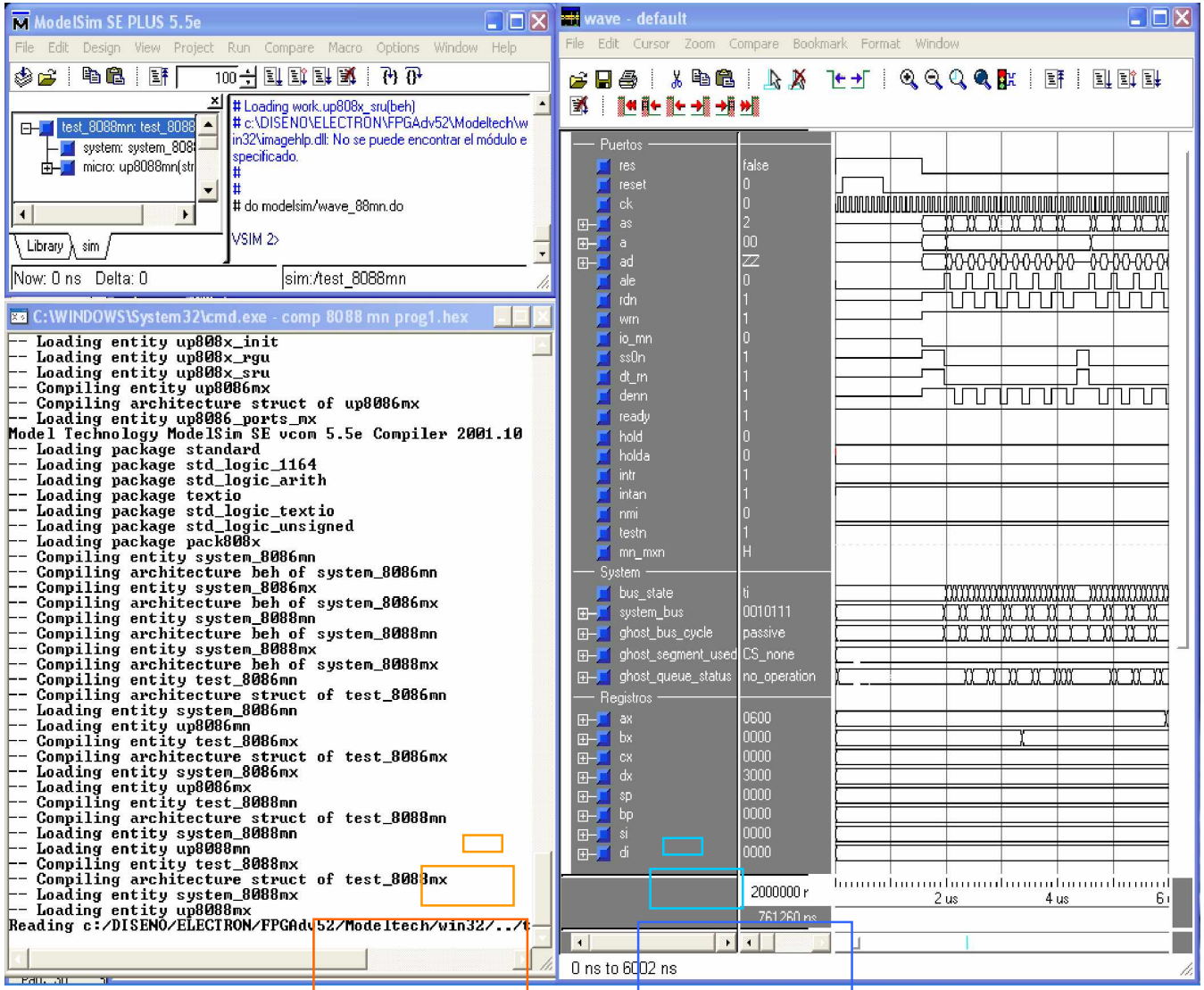


Fig 3.- Pantallazo de ModelSim y la Terminal CMD

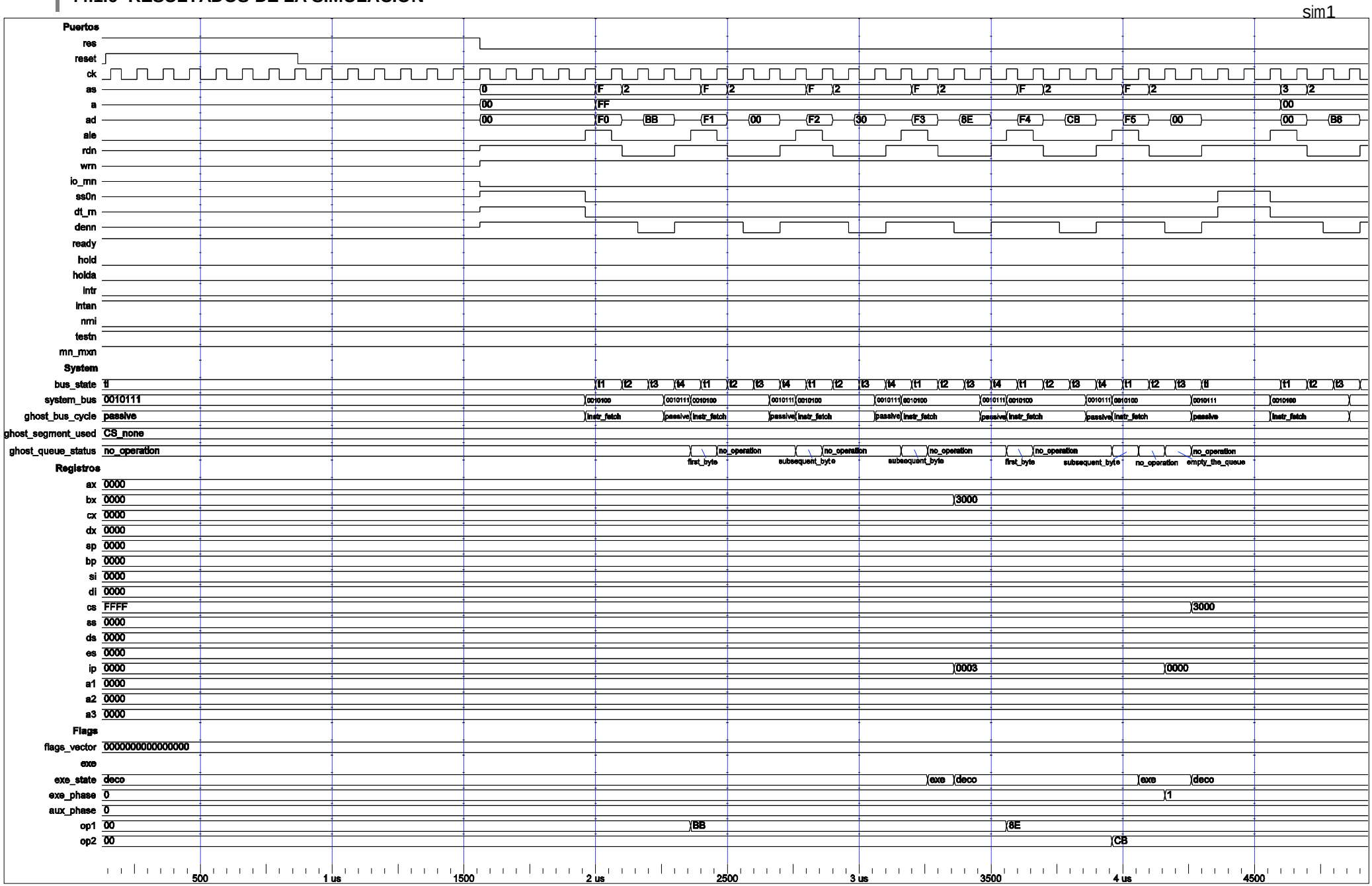
Terminal del sistema Operativo. Podemos ver la parte final del archivo comp.bat que compila los bancos de test y lanza el simulador. El fichero por lotes comp.bat, utiliza la opción "-f" para compilar la secuencia desde un archivo (*.sec) que encontramos en el directorio ".\modelsim\":

```
REM ****|   cOMPiLANDO LiBRERiA cOMúN   |****
vcom -f modelsim/up808x.sec
```

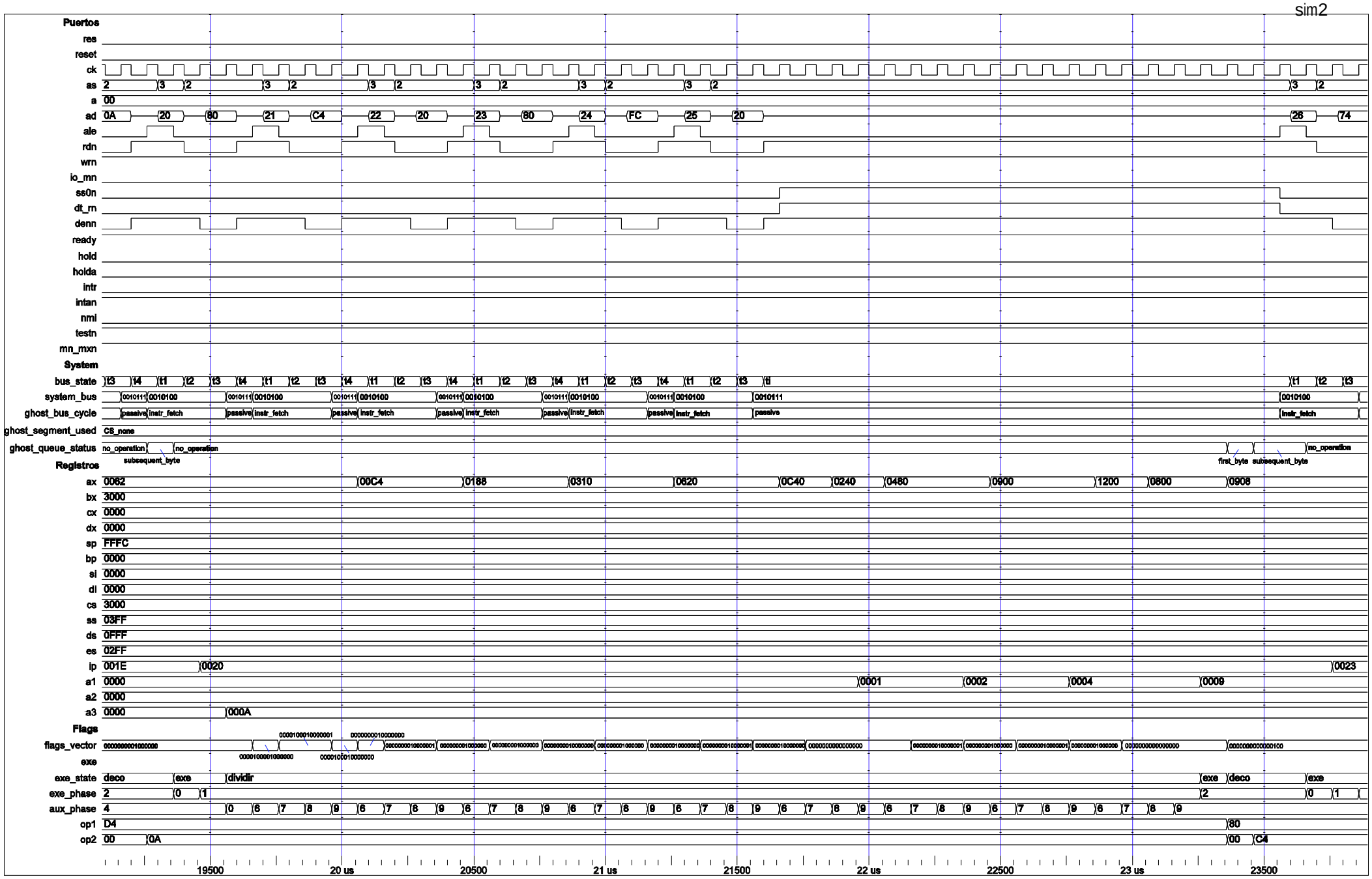
Pantalla de Formas de Onda. Presenta los resultados de la simulación. Esta es la pantalla inicializada por la macro "wave_88mn.do" que se encuentra en el directorio ".\modelsim". Para lanzar el simulador y que automáticamente cargue la macro, el fichero *.bat invoca el simulador utilizando el comando:

```
vsim -do modelsim/wave_88mn.do test_8088mn
```

44.1.6 RESULTADOS DE LA SIMULACIÓN

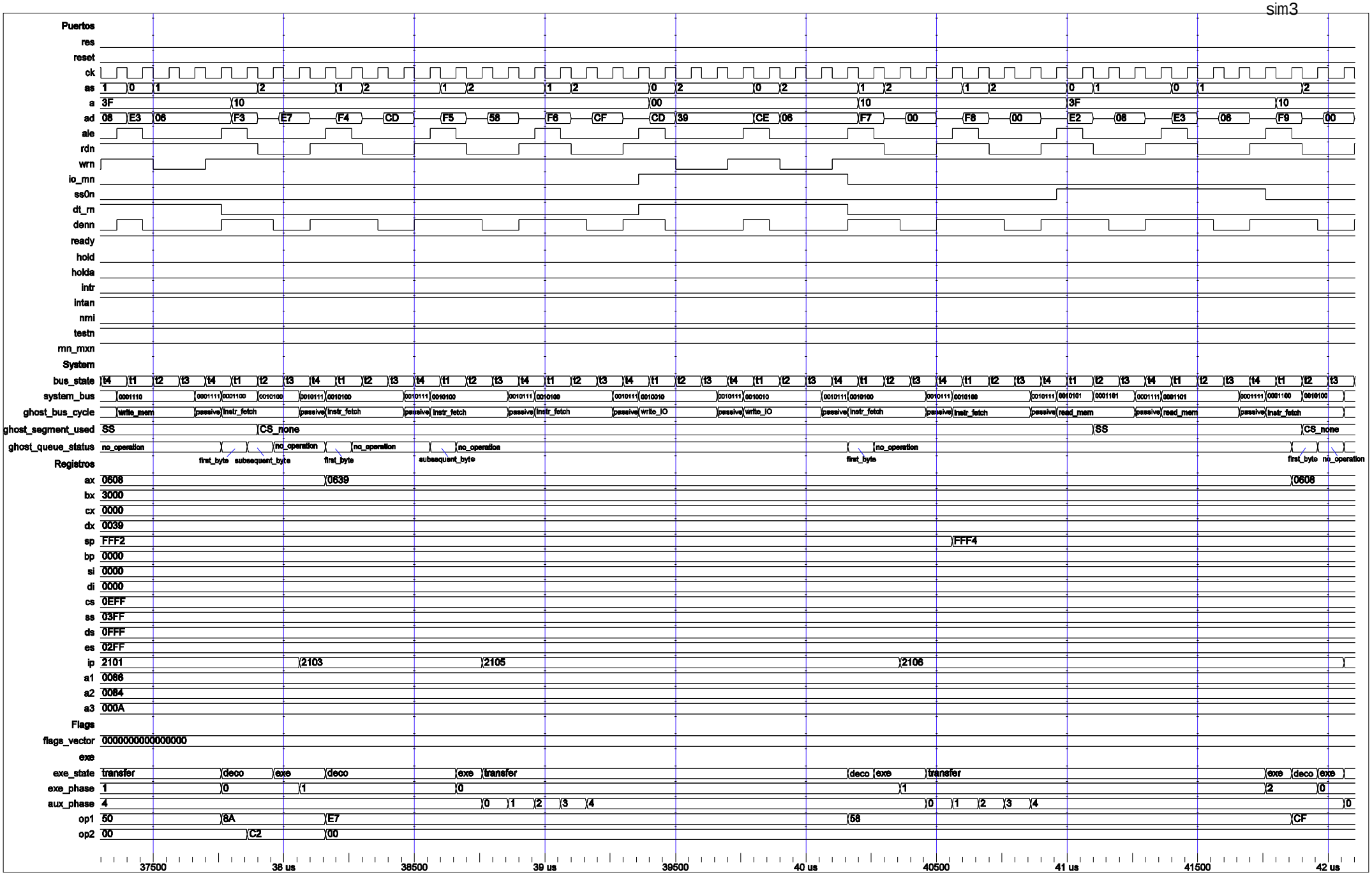


Sim 1.- Podemos Observar el RESET Inicial y Como el Microprocesador 8088 en Modo Mínimo Comienza a Bajar Datos a Partir de la Dirección FFFF0h. Al final Podemos Observar el Salto al Segmento de Código

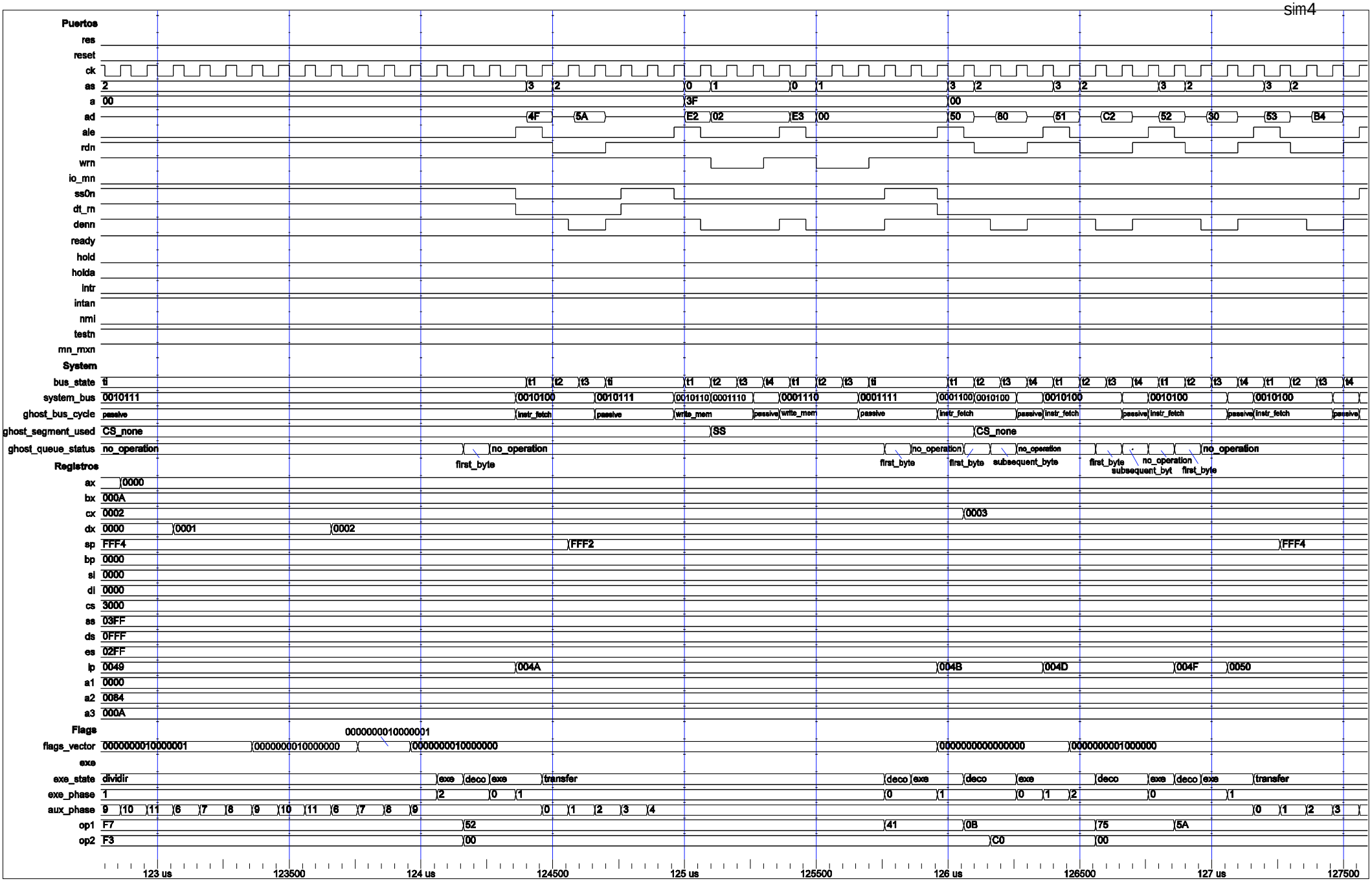


Sim 2.- División del Procedimiento "desp" línea 128. Podemos Observar como el Número en Binario de AX (62h), es Dividido por 10d (A3=0Ah) para convertirlo en BCD sin Empaquetar, AX=0908h Final de la División.

sim3

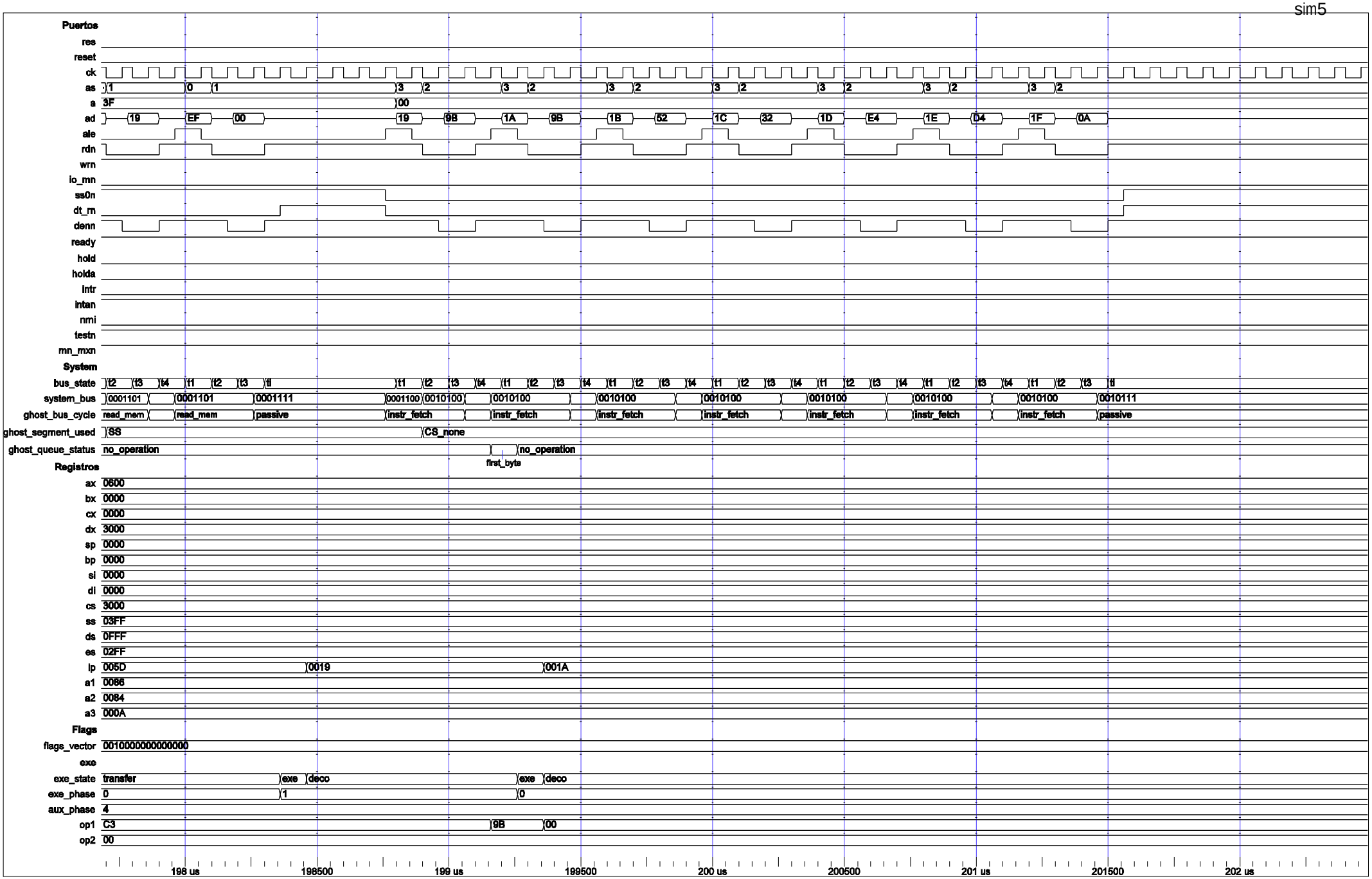


Sim 3- Estamos en el Servicio de la Interrupción 21. Interrupción que Emula la del MS-DOS para la Exhibición de Datos. Mandamos el Caracter ASCII "9" (39h) al Periférico de 8 bits Situado en la Dirección CDh de la Memoria I/O (linea 71)



Sim 4- Nos Encontramos en la Última Interacción del Bucle "despx1", Línea 176. El Resultado del Último Dígito de la Segunda Conversión, se Encuentra en DX=0002h Después de la División. Al Final Podemos Ver Como Abandonamos el Buche Cuando Z_Flag='1'.

sim5



Sim 5- El Programa Finaliza Con la Ejecución de una Instrucción WAIT (línea 113). el bus Permanece en Espera sin Actividad Cuando la Unidad IFU Llena la Cola de Instrucciones. Al Principio Podemos Ver el Final de la Instrucción RET (línea 190)

44.1.7 ANÁLISIS DE LA SIMULACIÓN

Podemos realizar dos tipos de análisis, el intensivo y el análisis funcional. El intensivo consiste en seguir paso a paso la ejecución de cada una de las instrucciones del programa comprobando su correcto funcionamiento en la simulación. Para ello, necesitamos tener en una mano el contenido del fichero "prog1.hex" y en la otra las hojas de la simulación, o mejor aún, la simulación en ModelSim. Es necesario haber comprendido completamente el funcionamiento del programa. El análisis funcional consiste en ver si el resultado final de la simulación coincide con el esperado.

Lamentablemente, simulaciones tan largas y complejas como esta, ocupan demasiado espacio para imprimirlas en papel. El programa termina de ejecutarse en el 8088 en modo mínimo a los 200 us... en papel ocuparía más de 40 páginas. Lo ideal es ver la simulación en el programa ModelSim. He elegido 5 partes de la simulación que me parecen interesantes y pueden servir de modelo para seguir el resto de la simulación.

SIMULACIÓN 1 (SIM1). INICIO.

En esta simulación, podemos apreciar claramente la secuencia de RESET inicial y cómo el microprocesador empieza a descargar instrucciones desde la posición de servicio de RESET (dirección=FFFF0h). Observar que el valor inicial de CS es FFFFh (valor que podemos modificar).

Para el seguimiento del código, recomiendo ignorar las señales que aparecen en la parte alta (señales de los Puertos y Sistema), estas señales son generadas (en su mayoría) por la BIU que trabaja de forma independiente y paralela a la EU. Debemos centrarnos en las señales de la pare baja (Registros y Exe), que nos indican el estado de la UE. Acudiremos a las señales de la parte alta cuando la EU requiera una transferencia (exe_state = transfer).

Alrededor de los 4 us, podemos ver como la instrucción "mov cs,bx" línea 15 carga el registro CX con el segmento donde hemos posicionado el código (3000h). La posición actual del programa la delatan los registros CS e IP (FFFF:0003) -> línea 15 y la instrucción en ejecución podemos verla en op1 y op2 (8Eh, CBh respectivamente).

SIMULACIÓN 2 (SIM2), 1ª CONVERSIÓN.

En esta simulación, podemos ver la ejecución completa de la instrucción AAM de la línea 128. El dato en AX (dato a convertir) es dividido por 10d de manera que al finalizar la parte alta del registro AX contiene el cociente de la división y la parte baja el resto (conversión del dato a BCD sin empaquetar). Podemos ver como la división está formada por un bucle principal de cuatro estados (aux_phase de 6 a 9) que se repite 8 veces para divisiones de 8 bits. También podemos ver como la BIU trabaja en paralelo llenando la cola de instrucciones mientras la EU está ocupada ejecutando la división.

SIMULACIÓN 3 (SIM3), SERVICIO DE INTERRUPCIÓN 21H

Para visualizar los resultados, he programado la interrupción 21h para que emule el comportamiento de la interrupción de visualización del DOS.

Para llamar a la interrupción 21h, desde el procedimiento de conversión se programa la función 06h en AH (AH=06h) y se pone en DL el dato ASCII que queremos visualizar. Ver las líneas135-137 y 183-185. En el servicio de interrupción, sacamos al puerto de entrada-salida donde se supone que está el dispositivo de visualización (puerto CDh), la función y el dato a visualizar (en AH y AL respectivamente).

En la simulación 3, podemos ver la ejecución de la instrucción "out 0CDh, ax" de la línea 71. Cuando la EU entra en el estado "transfer", le pide a la BIU la ejecución de un ciclo de bus de escritura en IO (write_IO). En las señales de "Puertos" vemos como se ejecuta el ciclo de bus de escritura en el periférico de la dirección "CDh", el primer dato (39h) es el carácter ASCII que corresponde al "9" y el segundo dato (06h) le indica al dispositivo de visualización función que tiene que realizar con el dato 39h.

SIMULACIÓN 4 (SIM4), ÚLTIMO BUCLE.

El procedimiento de conversión a base de divisiones, divide el dato a convertir sucesivamente entre 10 (BX=Ah) hasta que el cociente (AX) es cero. En la simulación nos encontramos en la última división de este buche. Líneas 176-180. Al prinicpi, podemos ver la parte final de la división que da como resultado 2 (DX = 0002h). La instrucción push (OP1=52h), lo salva en la pila. El contador CX lleva el conteo del número de dígitos del dato convertido (CX=3). La instrucción "or ax,ax" pone el flag Z_Flag (flags_vector(6)) a 1 cuando el cociente AX es cero. El buche finaliza cuando jnz no salta a despx1.

SIMULACIÓN 5 (SIM5), FINAL DEL PROGRAMA

En esta simulación podemos ver como termina el programa con la ejecución de la instrucción WAIT de la línea 112. Podemos apreciar como la instrucción anterior es la de retorno (RET) del proceso cercano, línea 190. La unidad IFU sigue descargando instrucciones de la memoria hasta llenar la cola de instrucciones mientras la EU está a la espera de que aparezca la señal TESTn para continuar con la ejecución.

44.1.8 FICHERO HISTÓRICO *.LOG

La simulación crea en el directorio ".\programs" un archivo ("memoria.log") que contiene las acciones realizadas sobre la memoria durante la simulación del programa.

```
rd_mem:CK=  23 Address=11111111111111110000 Data=10111011
rd_mem:CK=  27 Address=11111111111111110001 Data=00000000
rd_mem:CK=  31 Address=11111111111111110010 Data=00110000
rd_mem:CK=  35 Address=11111111111111110011 Data=10001110
rd_mem:CK=  39 Address=11111111111111110100 Data=11001011
rd_mem:CK=  43 Address=11111111111111110101 Data=00000000
rd_mem:CK=  49 Address=00110000000000000000 Data=10111000
rd_mem:CK=  53 Address=00110000000000000001 Data=11111111
rd_mem:CK=  57 Address=00110000000000000010 Data=00001111
[...]
rd_mem:CK=1979 Address=00000011111111101110 Data=00011001
rd_mem:CK=1983 Address=00000011111111101111 Data=00000000
rd_mem:CK=1991 Address=00110000000000011001 Data=10011011
rd_mem:CK=1995 Address=00110000000000011010 Data=10011011
rd_mem:CK=1999 Address=00110000000000011011 Data=01010010
rd_mem:CK=2003 Address=00110000000000011100 Data=00110010
rd_mem:CK=2007 Address=00110000000000011101 Data=11100100
rd_mem:CK=2011 Address=00110000000000011110 Data=11010100
rd_mem:CK=2015 Address=00110000000000011111 Data=00001010
```

Este archivo, podemos utilizarlo para analizar el resultado de la simulación. Sobre estas líneas, podemos ver las primeras y últimas operaciones realizadas con la memoria en la simulación de "prog1.hex". Podemos cotejarlas con las simulaciones sim1 y sim5. El campo CK indica el ciclo en el que aparece la acción. Cada CK dura 100 ns (TCLCL).

44.2 EJEMPLO 2. CONVERSIÓN DE BINARIO A 7 SEGMENTOS

En el ejemplo anterior hemos explicado detalladamente todos los pasos necesarios para simular un programa escrito en ensamblador. Este ejemplo sigue exactamente los mismos pasos por lo que no haremos un seguimiento tan detallado.

El ejemplo 2 es un típico programa de manejo de tablas, se trata de un conversor de binario a 7 segmentos. El dato a convertir es utilizado como índice en la tabla. Para la ejecución utilizamos la instrucción XLAT que es específica para este tipo de conversiones. Cargamos BX con la dirección de la tabla (posición Base), en AL introducimos el puntero al byte que codifica el dato dentro de esa misma tabla, el puntero puede ser perfectamente el dato original a convertir, y ejecutamos la instrucción XLAT que devuelve en AL (sobrescribiendo el dato sin codificar) el dato codificado. Ver la lógica de la instrucción XLAT en el juego de instrucciones (apéndice A) para más detalles.

44.2.1 FICHERO PROG2.ASM

El código completo del programa escrito en ensamblador (archivo "prog2.asm") lo podemos encontrar en el directorio ".\asm" dentro de la carpeta "assembler" en el CD-ROM del proyecto. Para escribirlo, como en el caso anterior, hemos utilizado un editor de textos que genere texto ASCII sin formato, como por ejemplo "VIM" VI Mejorado v6.1.

44.2.2 COMPILACIÓN DEL FICHERO ASM

Para compilar el fichero escrito en ensamblador *.asm y generar el archivo *.lst que utilizaremos para la simulación, podemos utilizar el fichero por lotes "hex.bat", en el directorio ".\assembler" del CD-ROM del proyecto, que automatiza la tarea. Par ello debemos introducir el comando:

```
c:\assembler> hex.bat prog2.asm ↵

Microsoft (R) Macro Assembler Version 6.14.8444
Copyright (C) Microsoft Corp 1981-1997.  All rights reserved.
 Assembling: ..\asm\prog2.asm
--=====================================================--
--|lOS FiCHEROS dE sALiDA eSTáN eN eL DiRECTORiO .\LST |--
--=====================================================--

c:\assembler>_
```

El fichero de salida que nos interesa (fichero "prog2.lst"), es generado en el directorio ".\lst" si no hay fallos en el archivo "prog1.asm".

44.2.3 GENERACIÓN DEL ARCHIVO DE SIMULACIÓN

Como base el archivo "prog2.lst" generado por el compilador MASM, crearemos el archivo "prog2.hex" en el directorio ".\programs" una ver realizadas las siguientes modificaciones.

1.- DEFINICIÓN DE LOS SEGMENTOS:

Observar como en la definición de los segmentos (%) y los desplazamientos dentro del segmento (primeros 16 bits de cada línea de código), no es necesario definir primero el byte bajo y después el alto. Estos datos son de información para el cargador de memoria, no es código que se escriba en la memoria como el caso de los punteros en los que si que hay que definir primero el byte bajo y después el alto (ver la tabla 5).

LÍNEA	OLD(16bits)	NEW(16bits)	SIMBOLO
Posiciones Donde Se Va A Grabar El Contenido Del Segmento			
14	0000	%FFFF	ini
21	0000	%0000	int_vect
33	0000	%0EFF	int_serv
69	0000	%03FF	stack
75	0000	%0FFF	data
84	0000	%02FF	extra
89	0000	%3000	code
Punteros Que Se Actualizan En Tiempo De Ejecución			
15	----	0030	code
91	----	FF0F	data

Tab 5.- Cambios en los símbolos actualizados en tiempo de ejecución

2.- CAMBIO DE LOS OCTETOS

En la tabla 5, podemos ver los cambios realizados en los datos, desplazamientos y off-set para solucionar el pequeño bug en el formato que MASM escribe en el archivo "prog2.lst". El lector percatado, se dará cuenta de que los cambios que podemos ver de la mitad de la tabla hacia arriba (Segmento de Vectores y los 3 primeros datos del Segmento de Datos), son siempre los mismos mientras no cambiemos el sistema operativo "so.asm" o introduzcamos soporte/s para nuevas interrupciones. De la Mitad de la tabla hacia abajo (datos propios del programa y los desplazamientos, off-sets y datos inmediatos del segmento de código), cambian siempre dependiendo del programa que simulemos.

Podemos aprovechar que la mitad de la tabla no varía y crear una plantilla del sistema operativo ya modificado en la que solamente tengamos que insertar el código y los datos propios del programa. Con esta técnica adelantaríamos algo de tiempo en la modificación del archivo "*.lst".

LÍNEA	BYTES	OLD	NEW(LSB..MSB)	SIMBOLO
Segmento de Vecores de Interrupción				
22	4	0EFF0000	0000FF0E	ini
23	4	0EFF0100	0001FF0E	int_vect
24	4	0EFF0200	0002FF0E	int_serv
25	4	0EFF0300	0003FF0E	stack
26	4	0EFF0400	0004FF0E	data
28	4	0EFF4100	0041FF0E	extra
Segmento de Datos				
76	2	0FFF	FF0F	data_seg
77	2	02FF	FF02	extra_seg
78	2	03FF	FF03	stack_seg
79	2	0104	0401	dato_w
Desplazamientos y Datos Inmediatos				
93	2	0002	0200	ptr extra_seg
94	2	0004	0400	ptr stack_seg
95	2	0006	0600	dato_1
96	2	0009	0900	ptr1 7seg
98	2	0004	0400	ptr2 7seg
115	2	0024	2400	ptr tabla

Tab 6.- Cambios en los Octetos de los Datos y Desplazamientos.

3.-ELIMINACIÓN DE LOS COMENTARIOS

Se han eliminado los comentarios e información de la cabecera y el final del archivo *.lst. Aunque el cargador ignora estos comentarios, es posible que si aparece algún número en los primeros caracteres de la línea, el cargador lo deposite en la memoria. En cualquier caso, ese dato sería sobrescrito por el código del programa y nunca sería ejecutado.

En las siguientes páginas, podemos ver el resultado final de la modificaciones realizadas en el archivo "prog2.lst". Los números de línea indicados en esta página, hacen referencia a la numeración del código en el archivo final "prog2.hex" (no a la del archivo *.lst como en el ejemplo anterior). Este archivo es el que debemos indicar en la GENERIC "download_filename".

```
1.                      ;    --=================================================================================--
2.                      ;    --|PrOGRAMA eJEMPLO:      -cONVERSiÓN dE BCD a 7 Seg-                            |--
3.                      ;    --|                                                                                |--
4.                      ;    --| Las tablas se utilizan a menudo para convertir datos de una forma a otra. |--
5.                      ;    --|Una tabla está posicionada en memoria como una lista de datos que se con-  |--
6.                      ;    --|sultan por un procedimiento para efectuar una conversión. En muchas ocasio-|--
7.                      ;    --|nes, se puede utilizar la instrucción XLAT para buscar o consultar datos en|--
8.                      ;    --|una tabla, siempre y cuando la tabla contenga datos de 8 bits y menos de   |--
9.                      ;    --|256 elementos.                                                              |--
10.                     ;    --|-----------------------------------------------------------------------|--
11.                     ;    --|sEGMENTO iNiCiAL                                                 FFFF:0000 |--
12.                     ;    --|.......................................................................|--
13.                     ;    --|                                                                           |--
14.     %FFFF                ini segment                                ;Segmento Incicial FFFF:0000
15.     0000  BB 0030 R            mov     bx, code                 ;slto al segmento de código
16.     0003  8E CB                mov     cs, bx
17.                          ini ends
18.                     ;    --|-----------------------------------------------------------------------|--
19.                     ;    --|vECTORES dE iNTERRUPCiÓN                                         0000:0000 |--
20.                     ;    --|.......................................................................|--
21.     %0000                int_vect segment             ;Segmento de vectores de interrupción 0000:0000
22.     0000  0000FF0E             dd  0EFF0000h            ;Vector int 00h => 0EFF:0000 -> Error al Dividir
23.     0004  0001FF0E             dd  0EFF0100h            ;Vector int 01h => 0EFF:0100 -> Paso Único
24.     0008  0002FF0E             dd  0EFF0200h            ;Vector int 02h => 0EFF:0200 -> Terminal MNI
25.     000C  0003FF0E             dd  0EFF0300h            ;Vector int 03h => 0EFF:0300 -> Break Point
26.     0010  0004FF0E             dd  0EFF0400h            ;Vector int 04h => 0EFF:0400 -> Sobreflujo INTO
27.                              org 41h*4h
28.     0104  0041FF0E             dd  0EFF4100h            ;Vector int 41h => 0EFF:4100 -> INTR (recogido del bus)
29.                          int_vect ends
30.                     ;    --|-----------------------------------------------------------------------|--
31.                     ;    --|sERViViOS dE iNTERRUPCiÓN                                       0EFF:0000 |--
32.                     ;    --|.......................................................................|--
33.     %0EFF                int_serv segment             ;Segmento de servicios de interrupción 0EFF:0000
34.                              assume ds:data, es:extra, ss:stack
35.                     ;    --|.......................................................................|--
36.                     ;    --|div0  - int 00h                                                         |--
37.                               org 0000h               ;Servicio de int 00h
38.     0000  90                  nop                     ;Acciones
39.     0001  CF                  iret                    ;Retorno del servicio de interrupción
40.                     ;    --|.......................................................................|--
41.                     ;    --|step  - int 01h                                                         |--
42.                               org 0100h               ;Servicio de int 01h
43.     0100  90                  nop                     ;Acciones
44.     0101  CF                  iret                    ;Retorno del servicio de interrupción
45.                     ;    --|.......................................................................|--
46.                     ;    --|nmi   - int 02h                                                         |--
47.                               org 0200h               ;Servicio de int 02h
48.     0200  90                  nop                     ;Acciones
49.     0201  CF                  iret                    ;Retorno del servicio de interrupción
50.                     ;    --|.......................................................................|--
51.                     ;    --|break - int 03h                                                        |--
52.                               org 0300h               ;Servicio de int 03h
53.     0300  90                  nop                     ;Acciones
54.     0301  CF                  iret                    ;Retorno del servicio de interrupción
55.                     ;    --|.......................................................................|--
56.                     ;    --|over  - int 04h                                                        |--
57.                               org 0400h               ;Servicio de int 04h
58.     0400  90                  nop                     ;Acciones
59.     0401  CF                  iret                    ;Retorno del servicio de interrupción
60.                     ;    --|.......................................................................|--
61.                     ;    --|intr  - int XXh (XXh recogido del bus, 41h)                            |--
62.                               org 4100h               ;Servicio de int 41h
63.     4100  90                  nop                     ;Acciones
64.     4101  CF                  iret                    ;Retorno del servicio de interrupción
65.                       int_serv ends
66.                     ;    --|-----------------------------------------------------------------------|--
67.                     ;    --|sEGMENTO dE PiLA                                                 03FF:0000 |--
68.                     ;    --|.......................................................................|--
69.     %03FF                stack segment                ;Segmento de Pila 03FF:0000
70.     0000                    db      50 dup(?)         ;reserva 50 bytes
71.                       stack ends
72.                     ;    --|-----------------------------------------------------------------------|--
73.                     ;    --|sEGMENTO dE dATOS                                                0FFF:0000 |--
74.                     ;    --|.......................................................................|--
75.     %0FFF                data segment                 ;Segmento de datos 0FFF:0000
76.     0000 FF0F               data_seg  dw      0FFFh   ;Segmento de Datos
77.     0002 FF02               extra_seg dw      02FFh   ;Segmento de Datos
78.     0004 FF03               stack_seg dw      03FFh   ;Segmento de Datos
79.     0006 0401             dato_1    dw      0104h   ;Word a convertir
80.                       data ends
81.                     ;    --|-----------------------------------------------------------------------|--
82.                     ;    --|sEGMENTO eXTRA                                                   02FF:0000 |--
83.                     ;    --|.......................................................................|--
84.     %02FF                extra segment                ;Segmento extra de datos 02FF:0000
85.                       extra ends
```


...sigue

continua...

```
86.                     ;    --|-------------------------------------------------------------------------|--
87.                     ;    --|sEGMENTO dE CóDiGO                                                   3000:0000 |--
88.                     ;    --|.........................................................................|--
89.     %3000               code segment                              ;Segmento de Pila 3000:0000
90.                               assume cs:code, ds:data, es:extra, ss:stack
91.     0000  B8 FF0F R             mov     ax,data                   ;Cargo los registros de segmento
92.     0003  8E D8                 mov     ds,ax
93.     0005  8E 06 0200 R          mov     es,extra_seg
94.     0009  8E 16 0400 R          mov     ss,stack_seg

95.     000D  A1 0600 R             mov     ax,dato_1                 ;convierte dato_1 a 7 segmentos
96.     0010  E8 0900               call    seg7
97.     0013  86 C4                 xchg    al,ah
98.     0015  E8 0400               call    seg7
99.     0018  86 C4                 xchg    al,ah                     ;resultado AX=0666h (14 7_Seg)
100.    001A  9B                            wait                              ;wait en espera de TESTn
101.    001B  9B                            wait                              ;paro el micro

102.                    ;    --|.........................................................................|--
103.                    ;    --|PrOCEDiMiENTO dE cONVERSióN                                              |--
104.                    ;    --| Este procedimiento, sólo tiene dos instrucciones y se supone que AL con-  |--
105.                    ;    --|tiene el dígito que se va a convertir a código de 7 segmentos. Una de las  |--
106.                    ;    --|instrucciones, direcciona la tabla cargando en BX el desplazamiento para   |--
107.                    ;    --|acceder a esta y la otra instrucción realiza la conversión retornando el   |--
108.                    ;    --|código 7 seg en AL.                                                       |--
109.                    ;    --| Debido a que la tabla está en el segmento de código y la instrucción XLAT |--
110.                    ;    --|accede al segmento de datos implícitamente, es necesario establecer un pre-|--
111.                    ;    --|fijo de cambio de segmento, CS:                                            |--
112.                    ;    --|.........................................................................|--

113.    001C                    seg7    proc    near

114.    001C  53                            push    bx
115.    001D  BB 2400 R             mov     bx,offset tabla
116.    0020  2E: D7                xlat    cs:tabla
117.    0022  5B                            pop     bx
118.    0023  C3                            ret
119.                    ;    --|.........................................................................|--
120.                    ;    --|tabla                                                                    |--
121.                    ;    --|            seg:-gfedcba                                                  |--
122.    0024  3F                        tabla   DB      00111111b               ;0 -> 3Fh         -------
123.    0025  06                                DB      00000110b               ;1 -> 06h        |   a   |
124.    0026  5B                                DB      01011011b               ;2 -> 5Bh        |f     b|
125.    0027  4F                                DB      01001111b               ;3 -> 4Fh        |   g   |
126.    0028  66                                DB      01100110b               ;4 -> 66h         -------
127.    0029  6D                                DB      01101101b               ;5 -> 6Dh        |       |
128.    002A  7D                                DB      01111101b               ;6 -> 7Dh        |e     c|
129.    002B  07                                DB      00000111b               ;7 -> 07h        |   d   |
130.    002C  7F                                DB      01111111b               ;8 -> 7Fh         -------
131.    002D  6F                                DB      01101111b               ;9 -> 6Fh    Byte_Ctrl:0gfedcba
132.    002E                    seg7    endp

133.    002E            code ends
134.                    end
135.            ;    --|-------------------------------------------------------------------------|--
136.            ;    --|                            FiNAL dEL PrOGRAMA                           |--
137.            ;    --=========================================================================--
```

44.2.4 LANZAMIENTO DE LA SIMULACIÓN

Para lanzar la simulación, utilizaremos el archivo por lotes "comp.bat" que podemos encontrar en el directorio ".\sourceVHDL". Este archivo simplifica mucho la compilación y lanzamiento de la simulación de cualquier programa en cualquier micro en cualquier modo de funcionamiento.

El fichero por lotes compila todas las librerías y en función del primer y segundo parámetro (ver el formato en la sección "Lanzamiento de la Simulación de Forma Automatizada" en el ejemplo anterior), lanza la simulación con el banco de test que corresponda cargando una macro determinada que inicializa la pantalla "Wave" de ModelSim con las señales necesarias en el mejor formato (hexadecimal, binario) para el seguimiento del programa. Como primer parámetro (microprocesador utilizado) acepta las entradas "8088", "88", "8086" y "86", como segundo parámetro (modo de funcionamiento) acepta .las entradas "mn", "MN", "mx" y "MX. El tercer parámetro es el fichero que vamos a simular, en este caso el fichero "prog2.hex". Este fichero debe encontrarse en el subdirectorio "..\sourceVHDL\programs"".

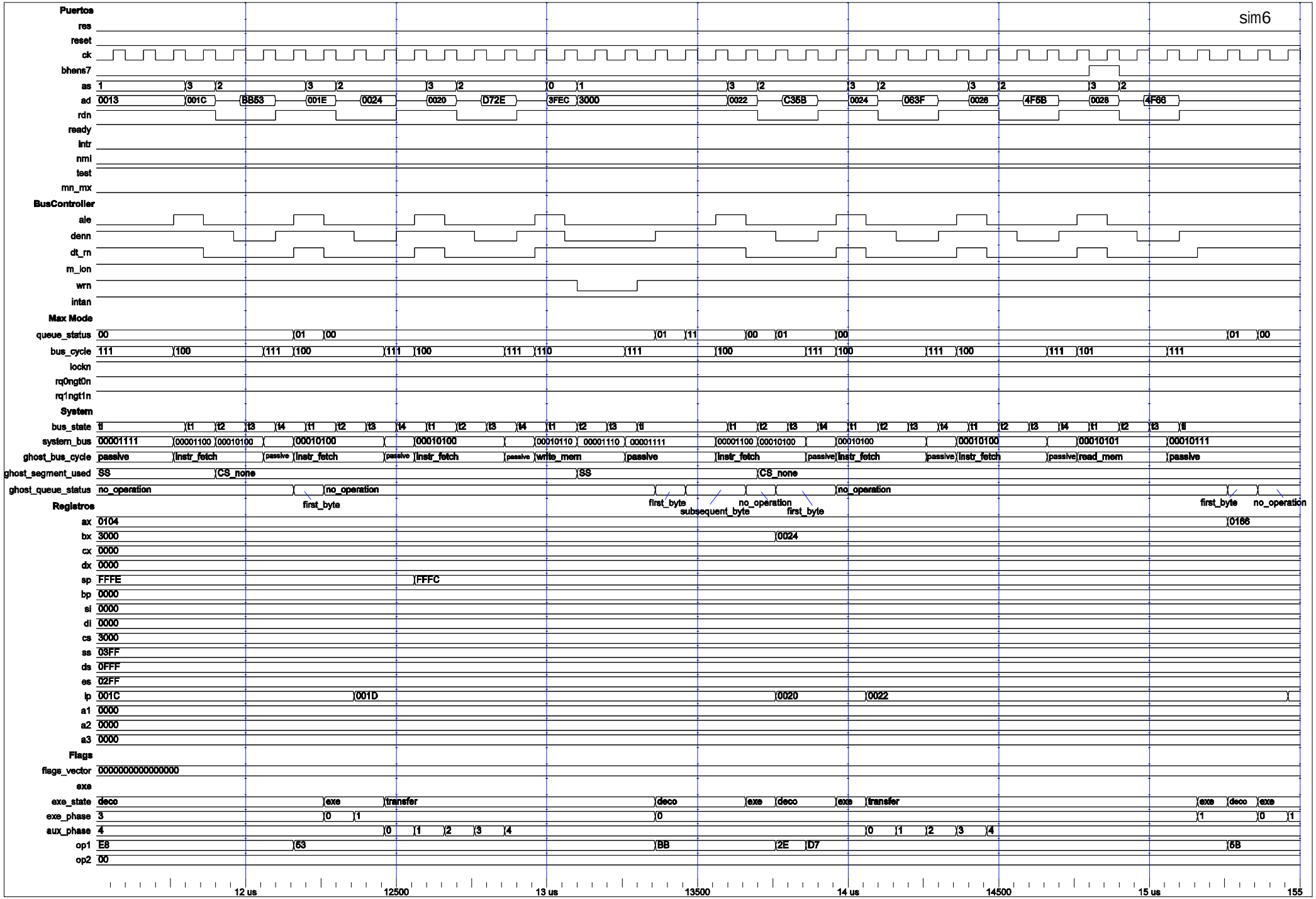
El fichero por lotes, emplea la táctica de simular siempre el mismo fichero (fichero "ram.hex") evitando de esta manera tener que modificar la constante GENERIC "download_filename" cada vez que cambiemos de programa de simulación. Para que la simulación sea la del archivo indicado en el tercer parámetro, el fichero por lotes crea una copia de archivo indicado con el nombre "ram.hex" en el subsirectorio ".\sourceVHDL\programs".

El comando que lanza la simulación del programa "prog2.hex" es el siguiente:

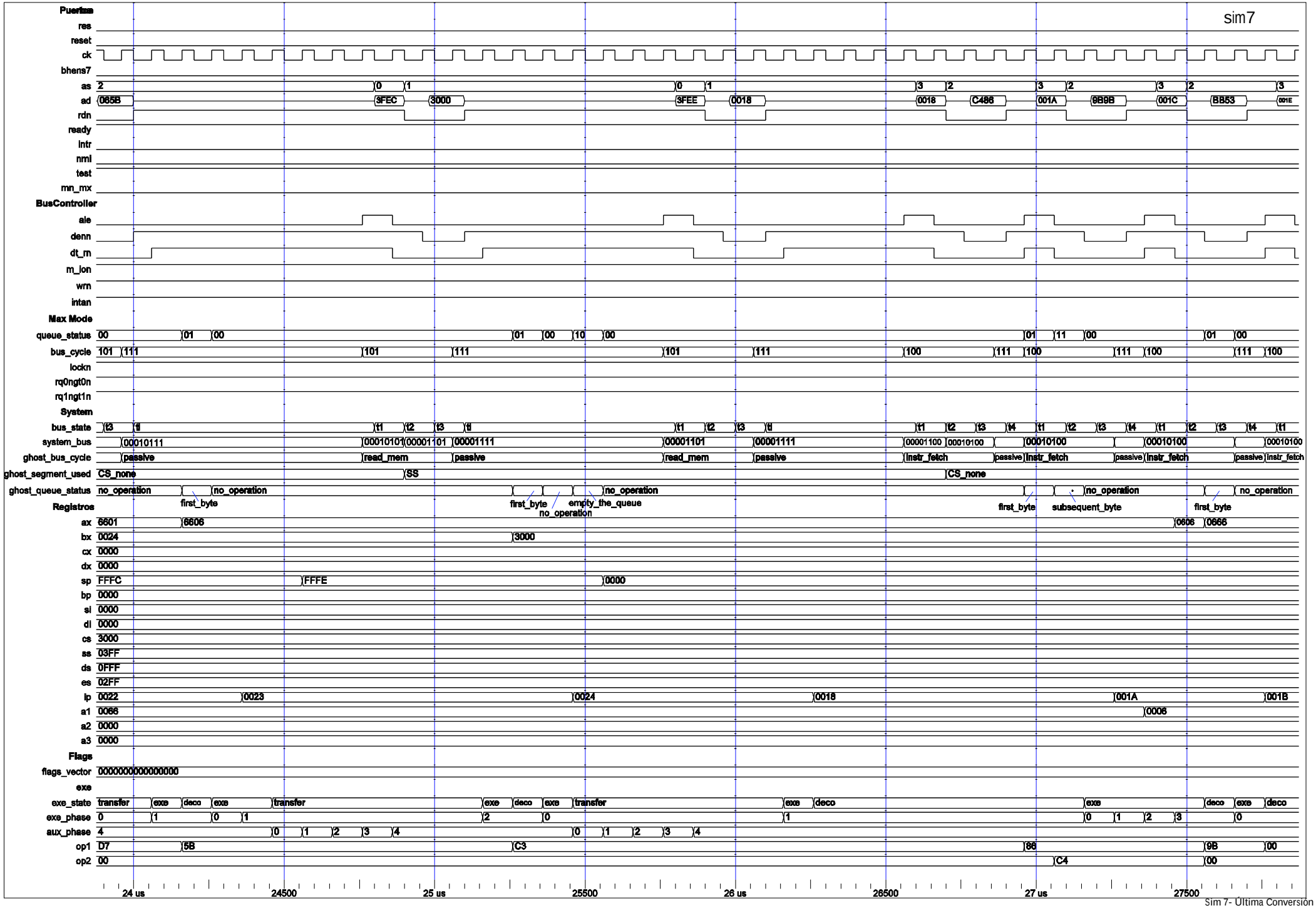
```
c:\sourceVHDL\> comp.bat 86 mx prog2.hex ↵
```

Como podemos observar, para la simulación de este programa vamos a utilizar el modelo de la CPU 8086 en un sistema funcionando en modo máximo. En la simulación anterior, utilizamos el modelo 8088 en modo mínimo. El cambiar el modelo de micro y el modo de funcionamiento resulta tan fácil como cambiar el programa a simular.

En las siguientes gráficas podemos ver los resultados de la simulación del programa de ejemplo 2. Para ejecutar la simulación teclear "run 30 us".



Sim 6- Primera Conversión



Sim 7- Última Conversión y WAIT for TESTn.

44.2.5 ANÁLISIS DE LA SIMULACIÓN

Este programa convierte el dato1 del segmento de datos (0104h en el ejemplo) al valor binario que activa los segmentos de un típico display electrónico de 7 segmentos y dos dígitos. Si hacemos corresponder los bits 7,6,5,4,3,2,1 y 0 de un byte a los segmentos 0,g,f,e,d,c,b y a respectivamente (ver la Figura 4), la activación de los segmentos b y c (06h) visualiza el dígito '1' mientras que la activación de los segmentos b,c,f y g (66h) visualiza el dígito '4'. La satánica combinación de ambos (666h) visualiza el número "14". Luego el dato1=14d es en código de 7 segmentos el valor 666.

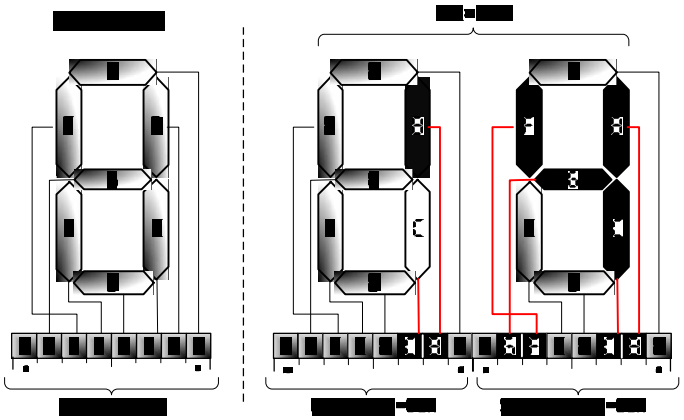


Fig 4.- Codificación de siete Segmentos.

Realizando un análisis funcional, podemos ver en la simulación 7 que el registro AX contiene al final del programa el dígito 14 codificado en 7 segmentos, por lo que podemos decir que la simulación (y el programa) han funcionado.

Un análisis más detallado, instrucción por instrucción, de la simulación constituiría un análisis intensivo. Pese a que este es un programa mucho más corto que el anterior (la simulación termina a los 28 microsegundos, en unas 5 páginas podríamos verla entera), no la voy a insertar de forma íntegra. Se extraído dos pantallas de la simulación que considero interesantes y que comentaré a continuación. En ellas aparecen las dos conversiones y se analiza el comportamiento de la pila de instrucciones del sistema, así como el alineamiento de las transferencias con la memoria.

SIMULACIÓN 6 (SIM6), PRIMERA CONVERSIÓN 4H->66H

Podemos ver como en el 8086 el bus de datos es de 16 bits. Al inicio de la simulación 6, nos encontramos en el primer salto a el procedimiento "seg7" (encargado de codificar el dato que le pasamos en AL (04h). Este salto ha provocado un flush del sistema y la unidad IFU comienza a bajar instrucciones a partir de la dirección CS:IP (PA=3001Ch) dirección que vemos en el primer ciclo "instr_fetch" que aparece en el bus. El dato recogido en este ciclo (0BB53h), lo utiliza directamente el decodificador que está a la espera (exe_state=deco). Podemos observar en el código de "prog2.hex" que estos datos corresponden al OP de la instrucción "push bx" (53h) y al primer byte de la instrucción "mov bx, offset tabla" (0BBh) líneas 114 y 115 respectivamente. En el ciclo de escritura de la instrucción "push bx", podemos ver como en el bus aparece el dato 3000h (que es el contenido de bx) y es escrito en la dirección SS:SP (PA=03FCEh). Al puntero "sp" se le restan las dos posiciones que va a ocupar el dato en la fase 0 del estado de transferencia. Las transferencias con pila de datos estás siempre alineadas (direcciones pares, siempre transferencias de 2 bytes).

Después de salvar el registro base (BX), es cargado con el offset hasta la tabla de datos desde la posición inicial del segmento de código (0024h). Finalmente podemos ver la ejecución de la instrucción "xlat cs:tabla", línea 116. Como se trata de una tabla que se encuentra en el segmento de código y la instrucción XLAT accede de forma implícita al segmento de datos, es preciso indicarle el cambio de segmento con el prefijo "cs:" (op1=2Eh). La instrucción recoge el AL el dato que se encuentra en la posición CS:BX+AL (PA=30028h), siendo AL el puntero dentro de la tabla que coincide con el dígito a convertir "4". Se puede observar en el bus, que la lectura de la tabla es sólo del los bits del banco bajo por lo que BHEn='1'. Al final de la instrucción, el registro AL contiene el dífito codificado (66h).

SIMULACIÓN 7 (SIM7), SEGUNDA CONVERSIÓN 1H->6H

A los 24 us nos encontramos al final de la última ejecución de la instrucción "xlat cs:tabla" de la línea 116. El dato en el bus (esta vez recogido de la parte alta del bus, dirección impar, A0='1' y BHEn='0') es colocado en la parte baja del registro AL (06h). La siguiente instrucción, recupera el valor de BX salvado en la pila de datos, y la siguiente "ret" línea 118, recupera el puntero de instrucciones de la pila para devolver el control del flujo del programa al bucle principal ahí donde se había quedado antes de saltar al procedimiento. El valor de IP es 18h, por lo que la siguiente instrucción en ejecutarse es "xchg al,ah", línea 99 a los 27 us. Esa instrucción intercambia los bytes del registro AX para que el resultado final tenga el orden adecuado AX=0666h. La última instrucción (op1=9Bh), es la instrucción de WAIT for TESTn, línea 100, que para la ejecución del programa.

Al no inicializar el puntero de pila (SP), el valor de este al principio y al final de la simulación, debe ser 0 si todo ha ido bien. Lo normal es inicializar el puntero de pila al número de bytes reservados para la pila de datos. El puntero de pila, como todos los punteros y desplazamientos, son tratados por la unidad GAU (n3/r7) como palabras de números enteros con signo, de manera que si no inicializamos SP y dada la lógica de la pila, que considera el valor inicial de SP como el "techo", todos los valores que adquiere el puntero durante la ejecución del programa son negativos. Esto se puede observar en todas las simulaciones realizadas hasta ahora.

44.2.6 FICHERO HISTÓRICO *.LOG

Como en las simulaciones del ejemplo anterior, esta crea en el directorio ".\programs" un archivo ("memoria.log") que contiene las acciones realizadas sobre la memoria durante la simulación del programa:

```
 [...]
rd_mem:CK=251 Address/BHEn=00000011111111101100/0
Data=0011000000000000
rd_mem:CK=261 Address/BHEn=00000011111111101110/0
Data=0000000000011000
rd_mem:CK=269 Address/BHEn=00110000000000011000/0
Data=1100010010000110
rd_mem:CK=273 Address/BHEn=00110000000000011010/0
Data=1001101110011011
rd_mem:CK=277 Address/BHEn=00110000000000011100/0
Data=1011101101010011
rd_mem:CK=281 Address/BHEn=00110000000000011110/0
Data=0000000000100100
```

En esta ocasión el formato difiere un poco, al aparecer la señal BHEn. Esta información puede ser utilizada para el análisis.

45. PROGRAMACIÓN EN TURBO C (C++)

La compatibilidad de estos modelos no solo con los programas desarrollados en código Intel Hexadecimal, sino también con los programas para el desarrollo de este código, les confiere un gran potencial. Podemos utilizar para la creación de nuestros propios programas cualquier lenguaje de programación cuyo compilador genere código Intel Hexadecimal.

El lenguaje que mejor se amolda al microprocesador es el Ensamblador, de hecho es un lenguaje creado exproceso para manejar el juego de instrucciones de estos micros, a cada instrucción del micro le corresponde un mnemónico en Ensamblador. Pese a la ventaja evidente del Ensamblador, cuando el programa se complica un poco, el número de instrucciones se dispara haciendo códigos ilegibles. Es entonces cuando tenemos que bajar el nivel de abstracción del código implementando el código en un lenguaje de mayor nivel como Pascal, Qbasic, C.... A modo de ejemplo, vamos a utilizar para la programar el Micro un lenguaje de programación ampliamente utilizado, el lenguaje C++, cuyo compilador podemos encontrar en el CD-ROM del proyecto..

EJEMPLO 3. MULTIPLICACIÓN DE MATRICES

El Compilador que vamos a utilizar en este ejemplo, es "Turbo C++ Version 1.01 Copyright (c) 1990 Borland International", es un compilador un poco antiguo, pero nos servirá como ejemplo. Como en el caso del ensamblador, no voy a entrar en detalles sobre este lenguaje de programación.

45.1 FICHERO MATRIZ.C

Para generar el fichero, debemos utilizar un editor que genere texto en código ASCII (texto plano) como VI o EDIT. También podemos utilizar el entorno de programación de Turbo C++. Para arrancar el entorno, debe estar instalado Turbo C++ y las variables de entorno correctamente configuradas. En estas condiciones tecleamos en la línea de comandos.

```
c:\turboC\> tc.exe matriz.c ↵
```

En la siguiente columna podemos ver el entrorno de programación de utiliza turbo C++.

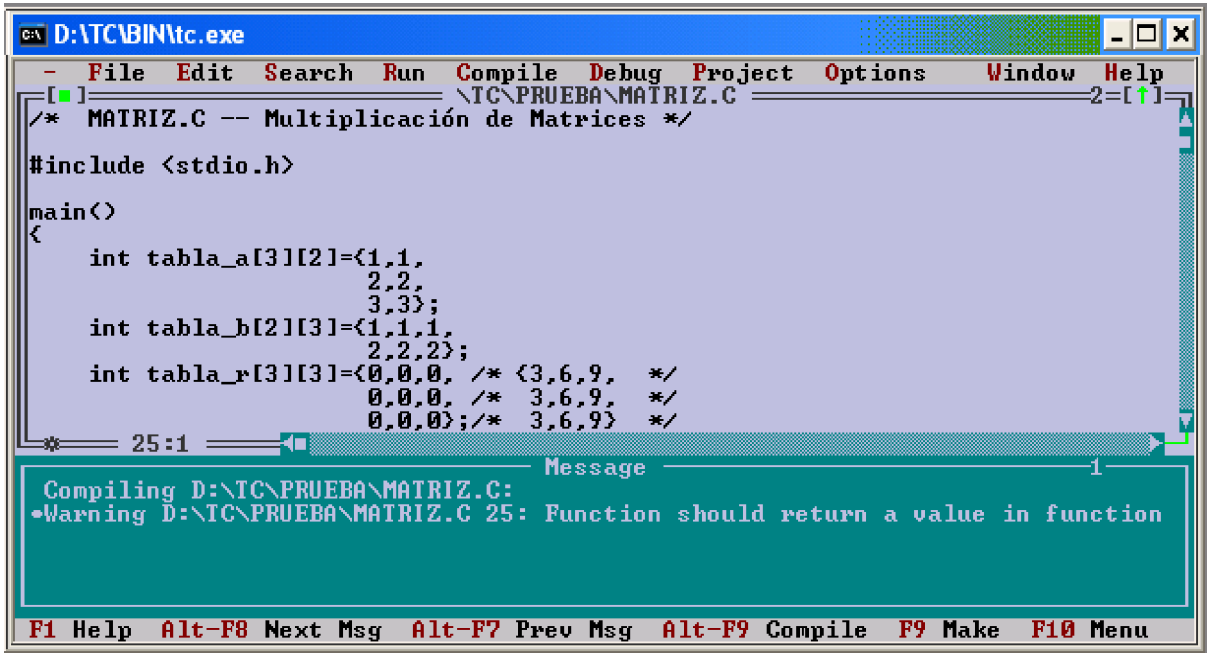


Fig 1.- Pantallazo del Entorno de Programación de Turbo C++.

Desde el entorno de programación podemos escribir, compilar, linkar y depurar el programa que escribamos, además de indicarnos de una forma interactiva los fallos y avisos que genera la compilación. También dispone de una pequeña ayuda sobre C. En el CD-ROM del proyecto, podemos encontrar manuales de programación en C. Este entorno es utilizado genéricamente por muchos lenguajes de programación. Se denomina entorno PWD.

El programa de ejemplo 3, implementa el algoritmo de multiplicación de dos matrices. Para que dos matrices se puedan multiplicar, el número de columnas de la primera matriz (matriz_a, dimensión 3x2), debe ser igual al número de filas de la segunda matriz (matriz_b, dimensión 2x3). El resulado es una matriz de dimensiones 3x3 (matriz_r). El sumatorio de los elementos 1 a 1 de la fila 1 de la primera matriz por la columna 1 de la matriz_b, crea el elemento k-ésimo de la fila 1 de la matriz de resultado.

El programa en C que implementa este algoritmo, ocupa unas pocas líneas como podemos ver en el siguiente código (código del fichero "matriz.c" en el directorio ".\turboC" del CD-ROM.

```
/*      MATRIZ.C -- Multiplicaci¢n de Matrices */

#include <stdio.h>

main()
{
    int tabla_a[3][2]={1,1,
                       2,2,
                       3,3};
    int tabla_b[2][3]={1,1,1,
                       2,2,2};
    int tabla_r[3][3]={0,0,0, /* {3,6,9,  */
                       0,0,0, /*  3,6,9,  */
                       0,0,0};/*  3,6,9}  */
    int i,j,k;

    for (i=1;i<3;i++)
    {
    for (j=1;j<3;j++)
        {
        for (k=1;k<2;k++)
            tabla_r[i][j]+=tabla_a[j][k]*tabla_b[k][i];
        }
    }
}
```

45.2 GENRACIÓN DEL ARCHIVO DE SIMULACIÓN

Una vez que tenemos el programa escrito en C, debemos pasarlo a código Intel Hexadecimal. Existen varias maneras de conseguirlo:

Compilación y Linkado del Programa

La forma más fácil de generar código Intel Hexadecimal, es generar un fichero *.exe o *.com. Para ello tenemos que compilar y linkar el programa escrito en C. En ambos casos, el fichero de salida no es directamente ejecutable por nuestro sistema de simulación por varias razones:

- 1º.El formato de salida no es "ASCII", sino binario. El cargador de la memoria, está preparado para leer el formato del fichero *.lst generado por el compilador MASM y no admite otros formatos.
- 2º, el fichero *.exe, contiene una "cabecera" con información sobre la localización de los segmentos, profundidad de la pila, parámetros de entrada,... que sólo entiende el cargador del sistema operativo, estos datos no forman parte del código del programa. En el caso de generar un archivo *.com, todos los datos en él, son código Intel Hexadecimal, los segmentos de datos, código y pila se encuentran posicionados de forma continua y no pueden ocupar más de 64K. Este formato nos puede venir muy bien para generar nuestros programas ya que las modificaciones que tenemos que realizar en el son mínimas.
- 3º, ambos formatos carecen del sistema operativo mínimo y necesario para hacer funcionar los modelos.

La solución a estos tres problemas es rescribir el cargador de la memoria para el formato de los archivos *.exe y *.com e insertar con un editor hexadecimal el código Intel Hexadecimal diseñado para el sistema operativo. Aunque lo ideal sería tener el sistema operativo en un fichero a parte que se cargara siempre junto al fichero a simular.

Ninguna de las soluciones expuestas está implementada en la revisión actual de los sistemas de simulación. Para simular el programa utilizaremos otra metodología como veremos a continuación.

Generación del Archivo Matriz.ASM a Partir del Fichero Matriz.C

La opción "-s" del compilador de C "TCC.EXE", genera un archivo *.ASM con el listado de instrucciones en Ensamblador del programa escrito en C. Esto nos viene muy bien por dos razones: 1.- Podemos ver como genera código el compilador de Turbo C y estudiar la codificación de las instrucciones en C, y 2.- no es necesario que modifiquemos el cargador de memoria ni el sistema operativo, simplemente tenemos que tomar como fuente el archivo *.ASM generado y seguir el mismo "modus operandi" que en los dos ejemplos anteriores.

El comando que debemos introducir en la terminal (cmd o command) es:

```
c:\turboC\> tcc.exe -S matriz.c ↵
```

En el mismo directorio aparecerá el archivo matriz.asm cuyo código podemos ver a continuación.

```
ifndef  ??version
?debug  macro
        endm
$comm   macro   name,dist,size,count
        comm    dist name:BYTE:count*size
        endm
        else
$comm   macro   name,dist,size,count
        comm    dist name[size]:BYTE:count
        endm
        endif
        ?debug  S "matriz.c"
        ?debug  C E9E6941131086D617472697A2E63
        ?debug  C
E920085C1615643A5C74635C696E636C7564655C737464696F2E68
_TEXT   segment byte public 'CODE'
_TEXT   ends
DGROUP  group   _DATA,_BSS
        assume  cs:_TEXT,ds:DGROUP
_DATA   segment word public 'DATA'
d@      label   byte
d@w     label   word
_DATA   ends
_BSS    segment word public 'BSS'
b@      label   byte
b@w     label   word
_BSS    ends
_DATA   segment word public 'DATA'
        db      1
        db      0
        db      1
        db      0
        db      2
        db      0
        db      2
        db      0
        db      3
        db      0
        db      1
        db      0
        db      1
        db      0
        db      1
        db      0
        db      2
        db      0
        db      2
        db      0
        db      2
        db      0
        db      0
        db      0
        db      0
        db      0
        db      0
        db      0
        db      0
        db      0
        db      0
        db      0
        db      0
        db      0
        db      0
        db      0
        db      0
        db      0
        db      0
        db      0
        db      0
        db      0
        db      0
        db      0
_DATA   ends
_TEXT   segment byte public 'CODE'
   ;
   ;    main()
   ;
        assume  cs:_TEXT
_main   proc    near
        push    bp
        mov     bp,sp
        sub     sp,44
        push    si
        push    di
        push    ss
        lea     ax,word ptr [bp-14]
        push    ax
        push    ds
        mov     ax,offset DGROUP:d@w+0
        push    ax
        mov     cx,12
        call    near ptr N_SCOPY@
        push    ss
        lea     ax,word ptr [bp-26]
        push    ax
        push    ds
        mov     ax,offset DGROUP:d@w+12
        push    ax
        mov     cx,12
        call    near ptr N_SCOPY@
        push    ss
        lea     ax,word ptr [bp-44]
        push    ax
        push    ds
```

```
        mov     ax,offset DGROUP:d@w+24
        push    ax
        mov     cx,18
        call    near ptr N_SCOPY@
   ;
   ;    {
   ;    int tabla_a[3][2]= {1,1,
   ;                        2,2,
   ;                        3,3};
   ;    int tabla_b[2][3]={1,1,1,
   ;                       2,2,2};
   ;    int tabla_r[3][3]= {0,0,0, /* {3,6,9,  */
   ;                        0,0,0, /*  3,6,9,  */
   ;                        0,0,0};/*  3,6,9}  */
   ;            int i,j,k;
   ;
   ;            for (i=1;i<3;i++)
   ;
        mov     word ptr [bp-2],1
        jmp     short @1@242
@1@50:
   ;
   ;            {
   ;            for (j=1;j<3;j++)
   ;
        mov     di,1
        jmp     short @1@194
@1@74:
   ;
   ;            {
   ;                    for (k=1;k<2;k++)
   ;
        mov     si,1
        jmp     short @1@146
@1@98:
   ;
   ;    tabla_r[i][j]+=tabla_a[j][k]*tabla_b[k][i];
   ;
        mov     bx,di
        shl     bx,1
        shl     bx,1
        mov     ax,si
        shl     ax,1
        add     bx,ax
        lea     ax,word ptr [bp-14]
        add     bx,ax
        mov     ax,word ptr [bx]
        push    ax
        mov     ax,si
        mov     dx,6
        imul    dx
        mov     dx,word ptr [bp-2]
        shl     dx,1
        add     ax,dx
        lea     dx,word ptr [bp-26]
        add     ax,dx
        mov     bx,ax
        pop     ax
        imul    word ptr [bx]
        push    ax
        mov     ax,word ptr [bp-2]
        mov     dx,6
        imul    dx
        mov     dx,di
        shl     dx,1
```

```
        add     ax,dx
        lea     dx,word ptr [bp-44]
        add     ax,dx
        mov     bx,ax
        pop     ax
        add     word ptr [bx],ax
        inc     si
@1@146:
        cmp     si,2
        jl      short @1@98
        inc     di
@1@194:
        cmp     di,3
        jl      short @1@74
        inc     word ptr [bp-2]
@1@242:
        cmp     word ptr [bp-2],3
        jl      short @1@50
   ;
   ;            }
   ;        }
   ;    }
   ;
        pop     di
        pop     si
        mov     sp,bp
        pop     bp
        ret
_main   endp
        ?debug  C E9
_TEXT   ends
_DATA   segment word public 'DATA'
s@      label   byte
_DATA   ends
_TEXT   segment byte public 'CODE'
_TEXT   ends
        public  _main
        extrn   N_SCOPY@:far
        end
```

El análisis del código puede resultar muy didáctico, pero no es, en absoluto, sencillo. C, como la mayoría de los lenguajes de programación de alto nivel, utiliza la pila de datos para pasar información a los procedimientos. Se trata de una técnica que al principio resulta complicada y un tanto abstracta. Lo primero que hace C al entrar en el procedimiento principal (_proc), es reservar 44 palabras de la pila, seguidamente salva en ella los punteros (desplazamiento y segmento) de las tablas de datos que van a ser multiplicadas. En las acciones siguientes lee las variables de interacción de los bucles "for" y salta a la rutina que comprueba la condición (casi al final del proceso). Como hay tres bucles "for" anidados, las condiciones y las acciones de estos también lo están . Dentro de los bucles se encuentra el algoritmo recursivo de la multiplicación que de nuevo utiliza los punteros y variables de interacción almacenadas en la pila de datos. Recuerdo que todos los direccionamientos que utilizan el registro BP, acceden al segmento de pila implícitamente. Podemos observar como C utiliza la instrucción LEA para incrementar el puntero al dato en la pila. Los punteros en la pila sólo son conocidos en tiempo de ejecución, la instrucción LEA permite utilizar la dirección de un dato, que no se sabe donde va a estar situado, como operando en el código del programa. Esta es una de las claves de la recolocación del código. Al final del proceso, podemos ver como recupera los datos y restablece el sistema.

A simple vista en este código, llama la atención el gran número de segmentos definidos. No dudo de la eficacia de C, pero desde luego, no es elegante. Los datos definidos como enteros en C "int", son codificados como números de 16 bits, de ahí que cada celda de las tablas en es segmento de datos, tenga dos dígitos y las multiplicaciones se codifiquen como multiplicaciones con signo de 16 bits (instrucción IMUL). En la cabecera y pie del listado, hay una serie de pseudoinstrucciones que dan soporte a la depuración del programa en el entorno de Turbo C. Estas líneas en principio se podrían borrar, pero hay que tener en cuenta que hay que modificar el símbolo (o nombre asociado) en todo el código.

Este código debemos insertarlo en el archivo del sistema operativo mínimo (archivo "so.asm" en el directorio ".\assembler") y salvarlo con otro nombre como "matriz.asm" en ese mismo directorio. A partir de ahora seguimos exactamente los mismos pasos que en los dos ejemplos anteriores.

La compilación la realizaremos con el siguiente comando:

```
c:\assembler> hex.bat matriz.asm ↵

Microsoft (R) Macro Assembler Version 6.14.8444
Copyright (C) Microsoft Corp 1981-1997.  All rights
reserved.
 Assembling: ..\matriz.asm
--=====================================================--
--| el FiCHERO dE sALiDA eSTá eN eL DiRECTORiO.\LST  |--
--=====================================================--

c:\assembler>_
```

El archivo "matriz.hex", preparado para la simulación, lo podemos ver en las siguientes páginas:

```
1.                      ;    --=================================================================================--
2.                      ;    --|PrOGRAMA eJEMPLO:          -MuLTIPLiCACióN dE MaTiCES-                      |--
3.                      ;    --|                                                                                |--
4.                      ;    --| Este ejemplo cuyo código ha sindo importado del fichero *.asm gnerado por |--
5.                      ;    --|Turbo C++ al compilarlo con la opción -s (TCC -s archivo.c), implementa el |--
6.                      ;    --|algoritmo de multiplicación de dos matrices de dimensiones 3x2 (taba_a) y  |--
7.                      ;    --|2x3 (tabla_b), el resultado en almacenado en una tabla de 3x3 (tabla_r) si-|--
8.                      ;    --|tuada en el segmento de datos.                                              |--
9.                      ;    --|-----------------------------------------------------------------------|--
10.                     ;    --|sEGMENTO iNiCiAL                                                 FFFF:0000 |--
11.                     ;    --|.......................................................................|--
12.                     ;    --|                                                                           |--
13.     %FFFF              ini segment                      ;Segmento Incicial FFFF:0000
14.     0000  BB 0030 R                mov     bx, code     ;slto al segmento de código
15.     0003  8E CB                    mov     cs, bx
16.                        ini ends
17.                     ;    --|-----------------------------------------------------------------------|--
18.                     ;    --|vECTORES dE iNTERRUPCióN                                         0000:0000 |--
19.                     ;    --|.......................................................................|--
20.     %0000              int_vect segment                 ;Segmento de vectores de interrupción 0000:0000
21.     0000  0000FF0E                 dd  0EFF0000h        ;Vector int 00h => 0EFF:0000 -> Error al Dividir
22.     0004  0001FF0E                 dd  0EFF0100h        ;Vector int 01h => 0EFF:0100 -> Paso Único
23.     0008  0002FF0E                 dd  0EFF0200h        ;Vector int 02h => 0EFF:0200 -> Terminal MNI
24.     000C  0003FF0E                 dd  0EFF0300h        ;Vector int 03h => 0EFF:0300 -> Break Point
25.     0010  0004FF0E                 dd  0EFF0400h        ;Vector int 04h => 0EFF:0400 -> Sobreflujo INTO
26.                        int_vect ends
27.                     ;    --|-----------------------------------------------------------------------|--
28.                     ;    --|sERViViOS dE iNTERRUPCióN                                       0EFF:0000 |--
29.                     ;    --|.......................................................................|--
30.     %0EFF              int_serv segment                 ;Segmento de servicios de interrupción 0EFF:0000
31.                           assume ds:data, es:extra, ss:stack
32.                     ;    --|.......................................................................|--
33.                     ;    --|div0  - int 00h                                                          |--
34.                                 org 0000h               ;Servicio de int 00h
35.     0000  90                     nop                    ;Acciones (Introducir aquí las acciones)
36.     0001  CF                     iret                   ;Retorno del servicio de interrupción
37.                     ;    --|.......................................................................|--
38.                     ;    --|step  - int 01h                                                          |--
39.                                 org 0100h               ;Servicio de int 01h
40.     0100  90                     nop                    ;Acciones (Introducir aquí las acciones)
41.     0101  CF                     iret                   ;Retorno del servicio de interrupción
42.                     ;    --|.......................................................................|--
43.                     ;    --|nmi   - int 02h                                                          |--
44.                                 org 0200h               ;Servicio de int 02h
45.     0200  90                     nop                    ;Acciones (Introducir aquí las acciones)
46.     0201  CF                     iret                   ;Retorno del servicio de interrupción
47.                     ;    --|.......................................................................|--
48.                     ;    --|break - int 03h                                                          |--
49.                                 org 0300h               ;Servicio de int 03h
50.     0300  90                     nop                    ;Acciones (Introducir aquí las acciones)
51.     0301  CF                     iret                   ;Retorno del servicio de interrupción
52.                     ;    --|.......................................................................|--
53.                     ;    --|over  - int 04h                                                          |--
54.                                 org 0400h               ;Servicio de int 04h
55.     0400  90                     nop                    ;Acciones (Introducir aquí las acciones)
56.     0401  CF                     iret                   ;Retorno del servicio de interrupción
57.                        int_serv ends
58.                     ;    --|-----------------------------------------------------------------------|--
59.                     ;    --|sEGMENTO dE PiLA                                                 03FF:0000 |--
60.                     ;    --|.......................................................................|--
61.     %03FF              stack segment                    ;Segmento de Pila 03FF:0000
62.     stack ends
63.                     ;    --|-----------------------------------------------------------------------|--
64.                     ;    --|sEGMENTO dE dATOS                                                0FFF:0000 |--
65.                     ;    --|.......................................................................|--
66.     DGROUP  group   _DATA,_BSS
67.     %2000              _DATA   segment word public 'DATA'
68.     0000                   d@  label   byte
69.     0000                   d@w label   word
70.     0000               _DATA   ends
71.     %2001              _BSS    segment word public 'BSS'
72.     0000                   b@  label   byte
73.     0000                   b@w label   word
74.     0000               _BSS    ends
75.     %0FFF              data segment                     ;Segmento de datos 0FFF:0000
76.     0000  0100             dw  01                       ;Tabla_a=
77.     0002  0100             dw  01                       ; ((1 1)
78.     0004  0200             dw  02                       ;  (2 2)
79.     0006  0200             dw  02                       ;  (3 3))
80.     0008  0300             dw  03
81.     000A  0300             dw  03
82.     000C  0100             dw  01                       ;Tabla_b
83.     000E  0100             dw  01                       ; ((1 1 1)
84.     0010  0100             dw  01                       ;  (2 2 2))
85.     0012  0200             dw  02
86.     0014  0200             dw  02
87.     0016  0200             dw  02
```


...sigue

continua...

```
88.     0018  0000            dw  00                      ;Tabla_r (resultado final)
89.     001A  0000            dw  00                      ; ((3 6 9)
90.     001C  0000            dw  00                      ;  (3 6 9)
91.     001E  0000            dw  00                      ;  (3 6 9))
92.     0020  0000            dw  00
93.     0022  0000            dw  00
94.     0024  0000            dw  00
95.     0026  0000            dw  00
96.     0028  0000            dw  00
97.     002A FF0F           data_seg  dw      0FFFh       ;Segmento de Datos
98.     002C FF02           extra_seg dw      02FFh       ;Segmento de Datos
99.     002E FF03           stack_seg dw      03FFh       ;Segmento de Datos
100.                    ;    --|.....................................................................|--
101.                    ;    --|Introducir en esta parte los datos del programa. Ej:                |--
102.                    ;  dato_b    db      62h            ;Byte a convertir
103.                    ;  dato_w    dw      0100h          ;Word a convertir
104.                    ;  ...
105.                    data ends
106.                    ;    --|-----------------------------------------------------------------|--
107.                    ;    --|sEGMENTO eXTRA                                          02FF:0000 |--
108.                    ;    --|.....................................................................|--
109.    %02FF              extra segment                          ;Segmento extra de datos 02FF:0000
110.                    ;    --|.....................................................................|--
111.                    ;    --|Introducir en esta parte los datos del Segmento extra. Ej:           |--
112.                    ;  result    db                     ;Resultado
113.                    ;  ...
114.    extra ends
115.                    ;    --|-----------------------------------------------------------------|--
116.                    ;    --|sEGMENTO dE CóDiGO                                      3000:0000 |--
117.                    ;    --|.....................................................................|--
118.    %3000              code segment                    ;Segmento de Pila 3000:0000
119.    assume cs:code, ds:data, es:extra, ss:stack
120.    0000  B8 FF0F R            mov     ax,data          ;Cargo los registros de segmento
121.    0003  8E D8                mov     ds,ax
122.    0005  BC 5000              mov     sp,50h           ;Inicializo la Pila
123.    0008  90                   nop
124.    0009  8E 16 2E00 R         mov     ss,stack_seg
125.                    ;    --|.....................................................................|--
126.                    ;    --|Introducir las Instrucciones y Procedimientos Cercanos del Programa. Ej:   |--
127.    000D  E8 0200              call    _main            ;Salto al procedimeinto de multiplicación
128.    0010  9B                   wait                     ;Terminar con 1 o dos Instrucciones WAIT
129.    0011  9B                   wait

130.    ;--Procedimiento
131.    0012               _main   proc    near                 ;Inicialización de los punteros
132.    0012  55                   push    bp
133.    0013  8B EC                mov bp,sp                    ;
134.    0015  83 EC 2C             sub sp,44                    ;  {
135.    0018  56                   push    si                   ;      int tabla_a[3][2]={1,1,
136.    0019  57                   push    di                   ;                         2,2,
137.    001A  16                   push    ss                   ;                         3,3};
138.    001B  8D 46 F2             lea ax,word ptr [bp-14]      ;      int tabla_b[2][3]={1,1,1,
139.    001E  50                   push    ax                   ;                         2,2,2};
140.    001F  1E                   push    ds                   ;      int tabla_r[3][3]={0,0,0, /* {3,6,9,  */
141.    0020  B8 0000 R            mov ax,offset DGROUP:d@w+0   ;                         0,0,0, /*  3,6,9,  */
142.    0023  50                   push    ax                   ;                         0,0,0};/*  3,6,9}  */
143.    0024  B9 0C00              mov cx,12                    ;      int i,j,k;
144.    0027  E8 0000 E            call    near ptr N_SCOPY@
145.    002A  16                   push    ss
146.    002B  8D 46 E6             lea ax,word ptr [bp-26]
147.    002E  50                   push    ax
148.    002F  1E                   push    ds
149.    0030  B8 0C00 R            mov ax,offset DGROUP:d@w+12
150.    0033  50                   push    ax
151.    0034  B9 000C              mov cx,12
152.    0037  E8 0000 E            call    near ptr N_SCOPY@
153.    003A  16                   push    ss
154.    003B  8D 46 D4             lea ax,word ptr [bp-44]
155.    003E  50                   push    ax
156.    003F  1E                   push    ds
157.    0040  B8 1800 R            mov ax,offset DGROUP:d@w+24
158.    0043  50                   push    ax
159.    0044  B9 1200              mov cx,18
160.    0047  E8 0000 E            call    near ptr N_SCOPY@

161.    004A  C7 46 FE 0100        mov word ptr [bp-2],1      ;for (i=1;i<3;i++)
162.    004F  EB 5E                jmp short @1@242
163.    @1@50:

164.    0051  BF 0100              mov di,1                   ;{
165.    0054  EB 51                jmp short @1@194           ;    for (j=1;j<3;j++)
166.    @1@74:

167.    0056  BE 0100              mov si,1                   ;    {
168.    0059  EB 46                jmp short @1@146           ;        for (k=1;k<2;k++)
169.    005B               @1@98:
170.    ;
171.    ;                tabla_r[i][j]+=tabla_a[j][k]*tabla_b[k][i];
172.    ;
173.    005B  8B DF                mov     bx,di
```


...sigue

continua...

```
174.    005D  D1 E3                shl     bx,1
175.    005F  D1 E3                shl     bx,1
176.    0061  8B C6                mov     ax,si
177.    0063  D1 E0                shl     ax,1
178.    0065  03 D8                add     bx,ax
179.    0067  8D 46 F2             lea     ax,word ptr [bp-14]
180.    006C  8B 07                mov     ax,word ptr [bx]
181.    006E  50                   push    ax
182.    006F  8B C6                mov     ax,si
183.    0071  BA 0600              mov     dx,6
184.    0074  F7 EA                imul    dx
185.    0076  8B 56 FE             mov     dx,word ptr [bp-2]
186.    0079  D1 E2                shl     dx,1
187.    007B  03 C2                add     ax,dx
188.    007D  8D 56 E6             lea     dx,word ptr [bp-26]
189.    0080  03 C2                add     ax,dx
190.    0082  8B D8                mov     bx,ax
191.    0084  58                   pop     ax
192.    0085  F7 2F                imul    word ptr [bx]
193.    0087  50                   push    ax
194.    0088  8B 46 FE             mov     ax,word ptr [bp-2]
195.    008B  BA 0600              mov     dx,6
196.    008E  F7 EA                imul    dx
197.    0090  8B D7                mov     dx,di
198.    0092  D1 E2                shl     dx,1
199.    0094  03 C2                add     ax,dx
200.    0096  8D 56 D4             lea     dx,word ptr [bp-44]
201.    0099  03 C2                add     ax,dx
202.    009B  8B D8                mov     bx,ax
203.    009D  58                   pop     ax
204.    009E  01 07                add     word ptr [bx],ax
205.    00A0  46                   inc     si
206.    00A1                @1@146:                         ;--Cndiciones de Salto
207.    00A1  83 FE 02             cmp     si,2
208.    00A4  7C B5                jl      short @1@98
209.    00A6  47                   inc     di
210.    00A7                @1@194:
211.    00A7  83 FF 03             cmp     di,3            ;
212.    00AA  7C AA                jl      short @1@74     ;           }
213.    00AC  FF 46 FE             inc     word ptr [bp-2] ;       }
214.    00AF                @1@242:                         ;   }
215.    00AF  83 7E FE 03          cmp word ptr [bp-2],3   ;
216.    00B3  7C 9C                jl  short @1@50

217.    00B5  5F                   pop di                  ;Recuperación del entorno
218.    00B6  5E                   pop si
219.    00B7  8B E5                mov sp,bp
220.    00B9  5D                   pop bp
221.    00BA  C3                   ret
222.    00BB                _main   endp

223.    00BB                code ends
224.                      extrn   N_SCOPY@:far
225.                    end
226.                    ;    --|-------------------------------------------------------------------------------|--
227.                    ;    --|                              FiNAL dEL PrOGRAMA                              |--
228.                    ;    --===============================================================================--
```

45.3 LANZAMIENTO DE LA SIMULACIÓN

Para lanzar la simulación, utilizaremos el archivo por lotes "comp.bat" que podemos encontrar en el directorio ".\sourceVHDL". El comando que debemos escribir en el "prompt" del sistema es:

```
c:\sourceVHDL\> comp.bat 86 mn matriz.hex ↵
```

En este caso, y para comprobar otro modelo, lanzaremos la simulación en el 8086 trabajando en modo mínimo. El archivo "matriz.hex" debe encontrarse en el subdirectorio ".\programs" junto al resto de los programas de ejemplo.El fichreo por lotes, lanzará la simulación cargando la macro de modelsim "wave_88mn.do" que se encuentra en el subdirectorio ".\ModelSim"

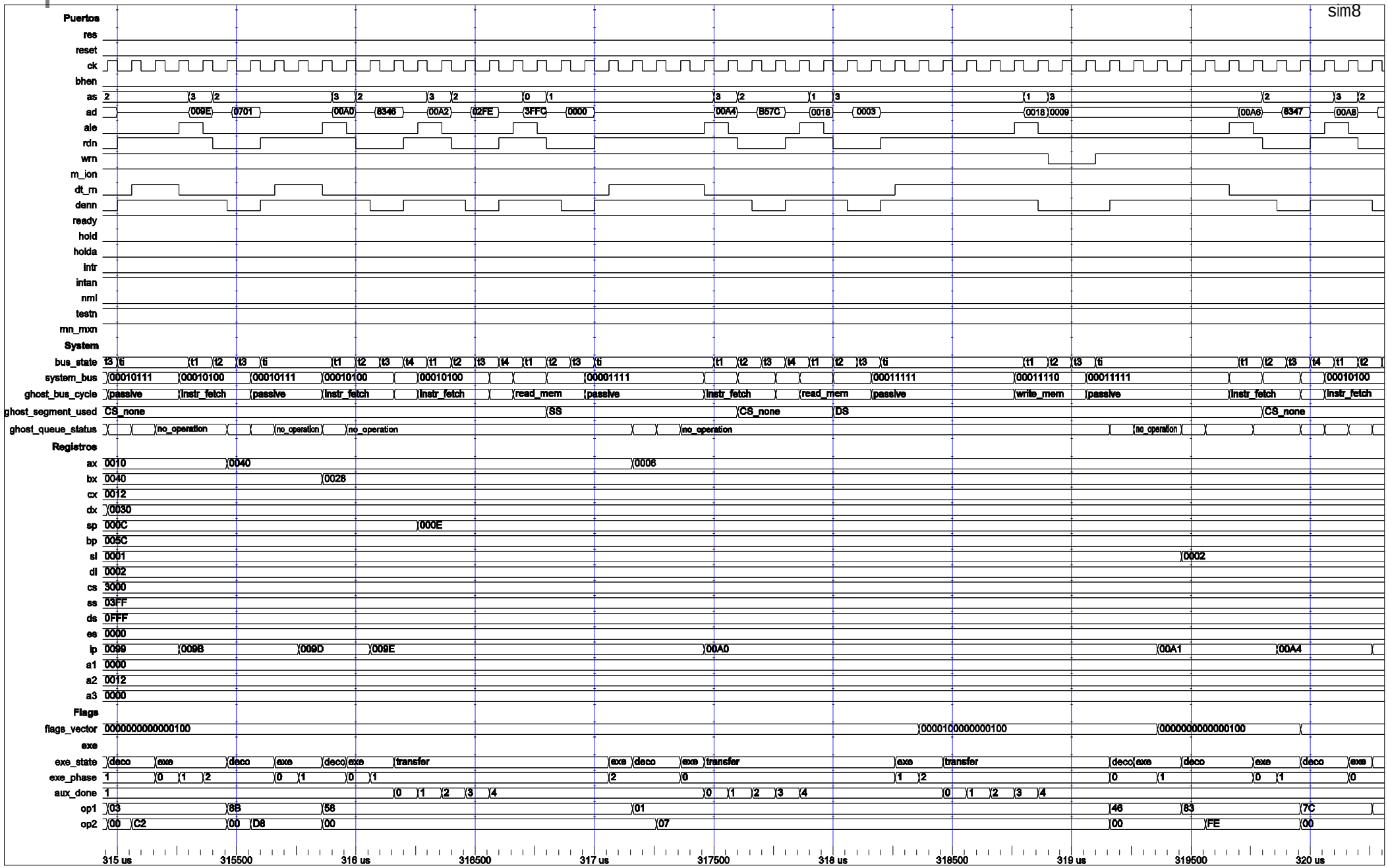
45.4 ANÁLISIS DE LA SIMULACIÓN

En esta simulación, podemos volvernos locos si tratamos de hacer un anális intensivo de la simulación, por lo que nos limitaremos a realizar un análisis funcional comprobando el resultado. En la pagina siguiente podemos ver la simulación 8, que salva el último valor.

Para el análisis funcional, vamos a utilizar el fichero de registro "memory.log" del directorio ".\programs". En el, buscaremos las acciones con las posiciones de memoria donde se almacena el resultado. La tabla de resultado tabla_r se almacena en las posiciones 18/28h del segmento de datos (PA=10008/10018). Si buscamos estas direcciones en el fichreo *.log, encontramos transferencias con estas posiciones. Podemos ver que la suma acumulativa se realiza con la celda en la memoria directamente (línea 284), de monera que para cada posición realiza dos lecturas y dos escrituras. El resultado lo podemos comprobar en la última escritura de cada celda, como podemos ver en el siguiente extracto del fichero "memory.log".

```
[...] rd_mem:CK=1230 Address/BHEn=10008h/0 Data=0000h [...]
[...] wr_mem:CK=1260 Address/BHEn=10008h/0 Data=0001h [...]
[...] rd_mem:CK=1530 Address/BHEn=10008h/0 Data=0001h [...]
[...] wr_mem:CK=1560 Address/BHEn=10008h/0 Data=0003h [...]
[...] wr_mem:CK=1832 Address/BHEn=1000Ah/0 Data=0006h [...]
[...] wr_mem:CK=2024 Address/BHEn=1000Ch/0 Data=0009h [...]
[...] wr_mem:CK=2125 Address/BHEn=1000Eh/0 Data=0003h [...]
[...] wr_mem:CK=2323 Address/BHEn=10010h/0 Data=0006h [...]
[...] wr_mem:CK=2665 Address/BHEn=10012h/0 Data=0009h [...]
[...] wr_mem:CK=2863 Address/BHEn=10014h/0 Data=0003h [...]
[...] wr_mem:CK=3007 Address/BHEn=10016h/0 Data=0006h [...]
[...] wr_mem:CK=3185 Address/BHEn=10018h/0 Data=0009h [...]
```

RESULTADOS DE LA SIMULACIÓN



Sim 8- Última Suma Para el Cálculo de la Celda (3,3) de la Matriz de Resultado. Nos Encontramos en la Última Pasada por Línea 204 Realizando la Suma Aacumulativa 3+6.

APÉNDICE A

juego de instrucciones

A1. CÓDIGOS OP DEL JUEGO DE INSTRUCCIONES DE LOS MICROS 8088/8086

En la siguiente tabla podemos ver el código máquina que entienden los micros 8088/8086 de Inter. ®; por extensión los modelos VHDL diseñados y por compatibilidad todas las evoluciones de los unos, los 80186, 80286,... ..., Pentium, etc y las posibles mejoras en los otros. El código en cuestión es el denominado “Intel Hexadecimal”, que, como es lógico, generan la mayoría de los programas diseñados para la programación. Desde el Ensamblador con sus compiladores más famosos (Turbo y Macro Assembler) hasta las últimas versiones visuales de C++, pasando por Basic, Pascal...

Esta tabla ha sido de gran ayuda para la generación de la unidad de decodificación (deco, n2/r5) y sin duda será de gran ayuda para aquel que quiera generar código “Intel Hexadecimal” a mano. Yo personalmente recomiendo utilizar Ensamblador por tener un Mnemónico (ver la siguiente tabla) por intrucción al contrario que los lenguajes de alto nivel (Pascal, Turbo C,...) que utilizan varias instrucciones en “Intel Hexadecimal” para codificar sus instrucciones.

OPERACIONES DE TRANSFERENCIA DE DATOS																																
Mnemónico	Código De La Instrucción																															
MOV = Mover:	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Registro/Memoria hacia/desde Registro	1	0	0	0	1	0	d	w	mod		reg			r/m																		
Inmediato a Registro/Memoria	1	1	0	0	0	1	1	w	mod		reg			r/m			data L								data H si w = 1							
Inmediato a Registro	1	0	1	1	w	reg			data L								data H si w = 1															
Memoria a Acumulador	1	0	1	0	0	0	0	w	disp L								disp H															
Acumulador a Memoria	1	0	1	0	0	0	1	w	disp L								disp H															
Registro/Memoria a Registro de Segmento	1	0	0	0	1	1	1	0	mod		0	sreg		r/m																		
Registro de Segmento a Registro/Memoria	1	0	0	0	1	1	1	0	mod		0	sreg		r/m																		
PUSH = Poner:																																
Registro/Memoria	1	1	1	1	1	1	1	1	mod		1	1	0	r/m																		
Registro	0	1	0	1	0	reg																										
Registro de segmento	0	0	0	sreg		1	1	0																								
POP = Sacar:																																
Registro/Memoria	1	0	0	0	1	1	1	1	mod		0	0	0	r/m																		
Registro	0	1	0	1	1	reg																										
Registro de segmento	0	0	0	sreg		1	1	1																								
XCHG = Intercambiar:																																
Registro/Memoria con Registro	1	0	0	0	0	1	1	w	mod		reg			r/m																		
Registro con Acumulador	0	1	0	1	1	reg																										
IN = Entrada desde:																																
Puero Dado	1	1	1	0	0	1	0	w	puerto																							
Puerto Variable	1	1	1	0	1	1	0	w																								
OUT = Sacar hacia:																																
Puero Dado	1	1	1	0	0	1	1	w	puerto																							
Puerto Variable	1	1	1	0	1	1	1	w																								
XLAT=Recodifica un byte:																																
AL	1	1	0	1	0	1	1	1																								
LEA = carga EA en::																																
Registro	1	0	0	0	1	1	0	1	mod		reg			r/m																		
LDS = carga puntero en:																																
DS	1	0	0	0	1	1	0	1	mod		reg			r/m																		
LES = carga puntero en::																																
ES	1	0	0	0	1	1	0	0	mod		reg			r/m																		
LAHF=Carga AH con:																																
“Flags”	1	0	0	1	1	1	1	1																								
SAHF=Almacena AH en:																																
“Flags”	1	0	0	1	1	1	1	0																								
PUSHF = Pone Flags en:																																
Pila (Apila los flags)	1	0	0	1	1	1	0	0																								
POPF = Recupera Flags de:																																
Pila	1	0	0	1	1	1	0	1																								

continua...

...sigue

INSTRUCCIONES ARITMÉTICAS																																
Mnemónico	Código De La Instrucción																															
ADD = Sumar:	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Registro/Memoria hacia/desde Registro	0	0	0	0	0	0	d	w	mod		reg			r/m																		
Inmediato a Registro/Memoria	1	0	0	0	0	0	s	w	mod	0	0	0		r/m			data L								data H si s:w = 01							
Inmediato a Acumulador	0	0	0	0	0	1	0	w	data L								data H si w = 1															
ADC = Sumar con Acarreo:																																
Registro/Memoria hacia/desde Registro	0	0	0	0	0	0	d	w	mod		reg			r/m																		
Inmediato a Registro/Memoria	1	0	0	0	0	0	s	w	mod	0	0	0		r/m			data L								data H si s:w = 01							
Inmediato a Acumulador	0	0	0	0	0	1	0	w	data L								data H si w = 1															
INC = Incrementar:																																
Registro/Memoria	1	1	1	1	1	1	1	w	mod	0	0	0		r/m																		
Registro	0	1	0	0	0	reg																										
AAA = Ajuste Ascii para la Suma del:																																
Acumulador	0	0	1	1	0	1	1	1																								
DAA = Ajuste Decimal de la Suma en:																																
Acumulador	0	0	1	0	0	1	1	1																								
SUB = Restar:																																
Registro/Memoria hacia/desde Registro	0	0	1	0	1	0	d	w	mod		reg			r/m																		
Inmediato desde Registro/Memoria	1	0	0	0	0	0	s	w	mod	1	0	1		r/m			data L								data H si s:w = 01							
Inmediato desde Acumulador	0	0	0	0	0	1	0	w	data L								data H si w = 1															
SBB = Restar con Borrow:																																
Registro/Memoria hacia/desde Registro	0	0	0	0	0	0	d	w	mod		reg			r/m																		
Inmediato desde Registro/Memoria	1	0	0	0	0	0	s	w	mod	0	0	0		r/m			data L								data H si s:w = 01							
Inmediato desde Acumulador	0	0	0	0	0	1	0	w	data L								data H si w = 1															
DEC = Decrementar:																																
Registro/Memoria	1	1	1	1	1	1	1	w	mod	0	0	1		r/m																		
Registro	0	1	0	0	1	reg																										
NEG = Cambio de signo:																																
Registro/Memoria	1	1	1	1	0	1	1	w	mod	0	1	1		r/m																		
CMP = Comparar:																																
Registro/Memoria y Registro	0	0	1	1	1	0	d	w	mod		reg			r/m																		
Inmediato con Registro/Memoria	1	0	0	0	0	0	s	w	mod	1	1	1		r/m			data L								data H si s:w = 01							
Inmediato con Acumulador	0	0	1	1	1	1	0	w	data L								data H si w = 1															
AAS = Ajuste Ascii para la Resta del:																																
Acumulador	0	0	1	1	0	1	1	1																								
DAS = Ajuste Decimal de la Resta en:																																
Acumulador	0	0	1	0	1	1	1	1																								
MUL = Multiplicar Sin Signo:																																
Registro/Memoria	1	1	1	1	0	1	1	w	mod	1	0	0		r/m																		
IMUL=Multiplicar (Entera) con signo:																																
Registro/Memoria	1	1	1	1	0	1	1	w	mod	1	0	1		r/m																		
AAM = Ajuste ASCII para la Mul:																																
Accumulador	1	1	0	1	0	1	0	0	0	0	0	0	1	0	1	0																
DIV = Dividir Sin Signo:																																
Registro/Memoria	1	1	1	1	0	1	1	w	mod	1	1	0		r/m																		
IDIV = División (Entera) con signo:																																
Registro/Memoria	1	1	1	1	0	1	1	w	mod	1	1	1		r/m																		
AAD = Ajuste ASCII para la Div:																																
Accumulador	1	1	0	1	0	1	0	1	0	0	0	0	1	0	1	0																
CBW = Convertir Byte en Word de:																																
Acumulador	1	0	0	1	1	0	0	0																								
CWD=Convertir Word en 2-Word de:																																
Acumulador	1	0	0	1	1	0	0	1																								

continua...

...sigue

INSTRUCCIONES LÓGICAS																																
Mnemónico	Código De La Instrucción																															
	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
NOT = Invertir:																																
Registro/Memoria	1	1	1	1	0	1	1	w	mod			0	1	0	r/m																	
SHL/SAL = Desplazamiento Lógico/Aritmético a la Izquierda:																																
Registro/Memoria	1	1	0	1	0	0	v	w	mod			1	0	0	r/m																	
SHR = Desplazamiento Lógico a la Derecha:																																
Registro/Memoria	1	1	0	1	0	0	v	w	mod			1	0	1	r/m																	
SAR = Desplazamiento Aritm a la Derecha:																																
Registro/Memoria	1	1	0	1	0	0	v	w	mod			1	1	1	r/m																	
ROL = Rotar a la izquierda:																																
Registro/Memoria	1	1	0	1	0	0	v	w	mod			0	0	0	r/m																	
ROR = Rotar a la Derecha:																																
Registro/Memoria	1	1	0	1	0	0	v	w	mod			0	0	1	r/m																	
RCL = Rotar a la Izquierda a través del Carry:																																
Registro/Memoria	1	1	0	1	0	0	v	w	mod			0	1	0	r/m																	
RCR = Rotar a la Derecha a través del Carry:																																
Registro/Memoria	1	1	0	1	0	0	v	w	mod			0	1	1	r/m																	
AND = And Lógica:																																
Registro/Memoria con Registro	0	0	1	0	0	0	d	w	mod			reg			r/m																	
Inmediato con Registro/Memoria	1	0	0	0	0	0	0	w	mod			1	0	0	r/m		data L								data H si w = 01							
Inmediato con Acumulador	0	0	1	0	0	1	0	w	data L								data H si w = 1															
TEST = Función AND con los Flags. Sin Resultado::																																
Registro/Memoria y Registro	1	0	1	0	0	1	0	w	mod			reg			r/m																	
Inmediato y Registro/Memoria	1	1	1	1	0	1	1	w	mod			0	0	0	r/m		data L								data H si w = 01							
Inmediato y Acumulador	0	0	1	1	0	1	0	w	data L								data H si w = 1															
OR = Or Lógica:																																
Registro/Memoria con Registro	0	0	0	0	1	0	d	w	mod			reg			r/m																	
Inmediato con Registro/Memoria	1	0	0	0	0	0	0	w	mod			0	0	1	r/m		data L								data H si w = 01							
Inmediato con Acumulador	0	0	1	0	0	1	0	w	data L								data H si w = 1															
XOR = OR exclusiva Lógica:																																
Registro/Memoria con Registro	0	0	1	1	0	0	d	w	mod			reg			r/m																	
Inmediato con Registro/Memoria	1	0	0	0	0	0	0	w	mod			1	1	0	r/m		data L								data H si w = 01							
Inmediato con Acumulador	0	0	1	1	0	1	0	w	data L								data H si w = 1															
INSTRUCCIONES DE CADENAS																																
REP (Prefijo) = Repite:	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
String Instruction	1	1	1	1	0	0	1	z																								
MOVS = Mover Cadena:																																
Byte/Word	1	0	1	0	0	1	0	w																								
CMPS = Comparar una Cadena:																																
Byte/Word	1	0	1	0	0	1	1	w																								
SCAS = Escanéa una Cadena:																																
Byte/Word	1	0	1	0	1	1	1	w																								
LODS = Carga Byte/Word en::																																
AL/AX	1	0	1	0	1	1	0	w																								
STOS = Almacena Byte/Word en::																																
AL/AX	1	0	1	0	1	0	1	w																								

continua...

...sigue

INSTRUCCIONES DE CONTROL DE TRANSFERENCIA

Mnemónico	Código De La Instrucción																															
CALL = Llamada:	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Directo Dentro del Segmento	1	1	1	0	1	0	0	0	disp L								disp H															
Indirecto Dentro del Segmento	1	1	1	1	1	1	1	1	mod	0	1	0	r/m																			
Directo entre Segmentos	1	0	0	1	1	0	1	0	offset-L								offset-H															
									seg-L								seg-H															
Indirecto entre Segmentos	1	1	1	1	1	1	1	1	mod	0	1	1	r/m																			
JMP = Salto Incondicional:																																
Directo Dentro del Segmento (Largo)	1	1	1	0	1	0	0	1	disp L								disp H															
Directo Dentro del Segmento (Corto)	1	1	1	0	1	0	1	1	disp																							
Indirecto Dentro del Segmento	1	1	1	1	1	1	1	1	mod	1	0	0	r/m																			
Directo Entre Segmentos	1	1	1	0	1	0	1	0	offset-L								offset-H															
									seg-L								seg-H															
Indirecto Entre Segmentos	1	1	1	1	1	1	1	1	mod	1	0	1	r/m																			
RET = Retorno de una Llamada:																																
Dentro del Segmento	1	1	0	0	0	0	1	1																								
Dentro del Seg. Sumando a SP imm-data	1	1	0	0	0	0	1	0	data L								data H															
Entre Segmentos	1	1	0	0	1	0	1	1																								
Entre Segs. Sumando a SP imm-data	1	1	0	0	1	0	1	0	data L								data H															

INSTRUCCIONES DE SALTO CONDICIONAL

JXX = Salto Condicional:	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Dentro del Segmento, Salto Corto.	0	1	1	1	c	c	c	c	disp																							

Mnemónico: (JXX)	Código de Condicción (cccc)	Condición de salto (Lógica)	Descripción
JO	0000	OF=1	Saltar si existe desbordamiento.
JNO	0001	OF=0	Saltar si no existe desbordamiento.
JB/JNAE	0010	CF=1	Saltar si está abajo.
JAE/JNB	0011	CF=0	Saltar si está arriba o igual.
JE/JZ	0100	ZF=1	Saltar si es igual/cero.
JNE/JNZ	0101	ZF=0	Saltar si no es igual/no es cero.
JBE/JNA	0110	CF = 1 OR ZF = 1	Saltar si está a bajo o igual.
JA/JNBE	0111	CF=0 AND ZF=0	Saltar si está arriba.
JS	1000	S=1	Saltar si existe signo.
JNS	1001	S=0	Saltar si no existe signo.
JP/JPE	1010	P=1	Saltar si la paridad es par.
JNP/JPO	1011	P=0	Saltar si la paridad es nones.
JL/JNGE	1100	SF <> OF	Saltar si es menor que.
JGE/JNL	1101	SF=OF	Saltar si es mayor o igual.
JLE/JNG	1110	(SF <> OF) OR ZF = 1	Saltar si es menor o igual.
JG/JNLE	1111	ZF=0 AND (SF=OF)	Saltar si en mayor

LOOPING INSTRUCTIONS

LOOPING = Bucles:	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Dentro del Segmento, Salto Corto.	1	1	1	0	0	0	L	L	disp																							

Mnemónico	LL	Descripción
LOOPNZ/LOOPNE	00	Salata a la etiqueta si el contador (CX) es distinto de cero y ZF=0
LOOPZ/LOOPE	01	Salata a la etiqueta si el contador (CX) es distinto de cero y ZF=1
LOOP	10	Salata a la etiqueta si el contador (CX) es distinto de cero.
JCXZ	11	Salta si el contador es cero.

INTERRUPTION INSTRUCTIONS

INTERRUPT = Interrupciones:	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Dentro del Segmento, Salto Corto.	1	1	0	0	1	1	i	i	tipo (si ii=01)																							

Mnemónico	ii	Descripción
INT	00 (Tipo 3)	INT 3. Ejecuta la interrupción 3. (step by step)
	01 (específico)	INT número. Ejecuta la interrupcón “número”.
INTO	10	INT 4. Ejecuta la interrupción 4 (desbordamiento, Overflow handled)
IRET	11	Retorno de una interrupción recuperando los indicadores de la pila

continua...

INSTRUCCIONES DE CONTROL DEL PROCESADOR																															
SetFlags Instructions																															
Mnemonico	Código De La Instrucción																														
SetFlags = Establecer Flags:	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Registro de Banderas	1	1	1	1	1	f	f	t																							

Mnemónico	Código de flag (ff)	New Value (t)	Descripción
CLC	00 (Flag de Carry)	0 Borra	Borra el flag de Carry.
STC		1 Activa	Activa el flag de Carry, no se modifican otros flags
CLI	01 (Flag de Inter.)	0 Borra	Borra el Flag de Habilitación de Interrupción.
STI		1 Activa	Activa el Flag de Habilitación de Interrupción.
CLD	10 (Flag de Dirección)	0 Borra	Borra el Flag de Dirección.
STD		1 Activa	Activa el Flag de Dirección.

HLT = Parar:	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Procesador	1	1	1	1	0	1	0	0																							
WAIT = Espera:																															
Procesador	1	0	0	1	1	0	1	1																							
ESC = Salir de un dispositivo externo:																															
Procesador	1	1	0	1	1	x	x	x	mod		xxx			r/m																	
LOCK (prefijo)= Bloqueo del Bus:																															
Procesador	1	1	1	1	0	0	0	0																							

Tab 1.- Resumen del Formato (Operation Code, OP) del Juego de Instrucciones Implementado en los Modelos (Juego de Instrucciones de los Micros 8088/8086 Intel ®.)

Notas:

AL = Acumulador de 8 bits.
AX = Acumulador de 16 bits.
CX = Registro de Conteo.
DS = Segmento de Datos.
ES = Segmento Extra.

Si d = 1 entonces “hacia” reg; Si d = 0 entonces “desde” reg.
Si w = 1 entonces se trata de una instrucción que opera con palabras (16 bits), si no, se trata de una opración con bytes.

Si mod = 11 entonces r/m es tratado como el campo reg.
Si mod = 00 entonces DISP = 0*, no están presentes disp-low ni disp-high.
Si mod = 01 entonces DISP = disp-low con extensión de signo a 16 bits; disp-high no está presente.
Si mod = 10 entonces DISP = disp-low; disp-high.

Si r/m = 000 entonces EA = (BX)+(SI)+DISP
Si r/m = 001 entonces EA = (BX)+(DI)+DISP
Si r/m = 010 entonces EA = (BP)+(SI)+DISP
Si r/m = 011 entonces EA = (BP)+(DI)+DISP
Si r/m = 100 entonces EA = (SI)+DISP
Si r/m = 101 entonces EA = (DI)+DISP
Si r/m = 110 entonces EA = (BP)+DISP*
Si r/m = 111 entonces EA = (BX)+DISP

DISP sigue al segundo byte del opcode (antes de los datos inmediatos, si están presentes)
*Excepto si mod = 00 y r/m = 100, en cuyo caso EA = disp-high;disp-low.

Si s:w = 11 entonces el byte de datos inmediatos es extendido en signo para utilizarlo como operando de 16 bits.
Si v = 0 entonces el contador es uno; si v = 1 entonces el contador está localizado en CL.
x = no importa.

z es utilizado por las instrucciones de cadenas para compararlo con el Flag ZF.

PREFIJOS DE SEGMENTO

7	6	5	4	3	2	1	0
0	0	1	sreg		1	1	0

Tab 2.- Prefijos de Segmento

REG es asignado como se indica en la siguiente tabla:

reg, 16-Bit (w=1)		reg, 8-Bit (w=0)		Segmento (sreg)	
000	AX	000	AL	00	ES
001	CX	001	CL	01	CS
010	DX	010	DL	10	SS
011	BX	011	BL	11	DS
100	SP	100	AH		
101	BP	101	CH		
110	SI	110	DH		
111	DI	111	BH		

Tab 3.- Decodificación de reg y sreg.

En las instrucciones que hacen referencia a el “flags register”, lo hacen como un registro de 16 bits; podremos considerar la localización de los Flags en éste como se muestra:

Registro de Banderas															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
X	X	X	X	OF	DF	IF	TF	SF	ZF	X	AF	X	PF	X	CF

Tab 4.- Posición de las Banderas en el Registro.

A2. CAMBIOS DE BIT DE BANDERA.

Esta tabla muestra sólo las instrucciones que realmente cambian los bits de bandera. Cualquier instrucción que no esté mencionada no afectará ninguno de los bits de bandera :

	Banderas afectadas								
INSTRUCCÓN	O	D	I	T	S	Z	A	P	C
AAA	?				?	?	*	?	*
AAD	?				*	*	?	*	?
AAM	?				*	*	?	*	?
AAS	?				?	?	*	?	*
ADC	*				*	*	*	*	*
ADD	*				*	*	*	*	*
AND	0				*	*	?	*	0
CLC									0
CLD		0							
CLI			0						
CMC									*
CMP	*				*	*	*	*	*
CMPS	*				*	*	*	*	*
DAA	?				*	*	*	*	*
DAS	?				*	*	*	*	*
DEC	*				*	*	*	*	*
DIV	?				?	?	?	?	?
IDIV	?				?	?	?	?	?
IMUL	*				?	?	?	?	*
INC	*				*	*	*	*	*
IRET	*	*	*	*	*	*	*	*	*
MUL	*				?	?	?	?	*
NEG	*				*	*	*	*	*
OR	0				*	*	?	*	0
POPF	*	*	*	*	*	*	*	*	*
RCL/RCR	*								*
REPE/REPNE						*			
ROL/ROR	*								*
SAHF					*	*	*	*	*
SAL/SAR	*				*	*	?	*	*
SHL/SHR	*				*	*	?	*	*
SBB	*				*	*	*	*	*
SCAS	*				*	*	*	*	*
STC									1
STD		1							
STI			1						
SUB	*				*	*	*	*	*
TEST	0				*	*	?	*	*
XOR	0				*	*	?	*	0

Tab 1.- Instrucciones que Modifican los Flags.

A3. JUEGO DE INSTRUCCIONES.

En las siguientes páginas, podemos ver el juego de instrucciones completo implementado en los modelos diseñados. En las tablas confeccionadas (que agrupan las instrucciones en función de su comportamiento funcional general, “Transferencia de Datos”, “Lógicas”...), podemos distinguir el formato del Código de Instrucción y una información particular del funcionamiento, formato, tiempos, transferencias, flags que modifica, etc de cada instrucción.

Estas tablas han sido fundamentales para la generación de la unidad de ejecución (eu, n1/r4).

La información es una traducción ampliada de la Ayuda de Norton Guides 1987 desarrollada por Peter Norton Computing (en el CD se encuentra la versión original). Los ciclos de reloj hacen referencia a los microprocesadores originales (a partir de ellos se puede descifrar la lógica de ejecución interna), los modelos VHDL ejecutan las instrucciones mucho más rápido.

Las tablas abandonan el formato apaisado del resto de la memoria para adquirir el formato clásico con la intención de disminuir el número de páginas en esta sección. Espero no provocar una tortícolis en el lector.

? Indica que después de la ejecución de la instrucción, el Flag adquiere un valor indeterminado.

* Indica que el Flag es modificado.

0 Indica que el Flag es borrado.

1 Indica que al terminar a ejecución de la instrucción el Flag estará activo.

A3.1 TABLA DE INSTRUCCIONES DE TRANSFERENCIA DE DATOS

OPERACIONES DE TRANSFERENCIA DE DATOS

Mnemónico	Código De La Instrucción																															
MOV = Mover:	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Registro/Memoria hacia/desde Registro	1	0	0	0	1	0	d	w	mod		reg			r/m																		
Inmediato a Registro/Memoria	1	1	0	0	0	1	1	w	mod		reg			r/m			data L								data H si w = 1							
Inmediato a Registro	1	0	1	1	w	reg			data L								data H si w = 1															
Memoria a Acumulador	1	0	1	0	0	0	0	w	disp L								disp H															
Acumulador a Memoria	1	0	1	0	0	0	1	w	disp L								disp H															
Registro/Memoria a Registro de Segmento	1	0	0	0	1	1	1	0	mod		0	sreg		r/m																		
Registro de Segmento a Registro/Memoria	1	0	0	0	1	1	1	0	mod		0	sreg		r/m																		

```
MOV              Move (Byte or Word)                Flags: not altered

MOV destination,source

          Logic:    destination <- source

    MOV copia un byte o palabra desde la fuente al destino.

  ---------------------------------------------------------------------------
   Operands                  Clocks   Transfers  Bytes   Example
                           byte(word)
   register, register          2          -        2     MOV BX,CX
   memory, accumulator       10(14)       1        3     MOV MEM_DEST,AL
   accumulator, memory       10(14)       1        3     MOV AX,MEM_SOURCE
   memory, register        9(13) + EA     1       2-4    MOV MEM_DEST,BX
   register, memory        8(12) + EA     1       2-4    MOV BX,MEM_SOURCE
   register, immediate          4         -       2-3    MOV BX,0F6CDh
   memory, immediate      10(14) + EA     1       3-6    MOV MASK,0F6CDh
   seg-reg, reg16              2          -        2     MOV DS,BX
   seg-reg, mem16          8(12) + EA     1       2-4    MOV DS,SEGMENT_VAL
   reg16, seg-reg              2          -        2     MOV BP,SS
   memory, seg-reg         9(13) + EA     1       2-4    MOV CODE_VAR,CS
  ---------------------------------------------------------------------------
```

PUSH = Poner:

Registro/Memoria	1	1	1	1	1	1	1	1	mod		1	1	0	r/m																		
Registro	0	1	0	1	0	reg																										
Registro de segmento	0	0	0	sreg		1	1	0																								

```
PUSH             Push Word onto Stack               Flags: not altered

PUSH source

          Logic:    SP <- SP - 2
                    (SP) <- source

    PUSH decrementa SP en 2 unidades y copia el operando fuente en el nuevo
    puntero de la pila. El operando fuente no puede ser un registro de 8 bits

  ---------------------------------------------------------------------------
   Operands                  Clocks   Transfers  Bytes   Example
   register                    15         1        1     PUSH BX
   seg-reg (CS illegal)        14         1        1     PUSH ES
   memory                   24 + EA       2       2-4    PUSH PARAMETERS
  ---------------------------------------------------------------------------
```

Notas:
Cuando que el operando fuente se refiera a un byte en la memoria, siempre se pondrá una palabra completa.
El la instrucción PUSH SP, los microprocesadores 8086/8088 incrementan primero SP y luego lo salvan el la pila; sin embargo, los microprocesadores 80286 y 80386 salvan primero SP y luego lo incrementan.

POP = Sacar:

Registro/Memoria	1	0	0	0	1	1	1	1	mod		0	0	0	r/m																		
Registro	0	1	0	1	1	reg																										
Registro de segmento	0	0	0	sreg		1	1	1																								

```
POP              POP a Word from the Stack          Flags: not altered

POP destination

          Logic:    destination <- (SP)
                    SP <- SP + 2

    POP transfiere la palabra que apunta el puntero de la pila al operando
    destino, acto seguido se incrementa SP en 2 para generar el nuevo puntero
    de pila.

  ---------------------------------------------------------------------------
   Operands                  Clocks   Transfers  Bytes   Example
   register                    12         1        1     POP CX
   seg-reg (CS illegal)        12         1        1     POP ES
   memory                   25 + EA       2       2-4    POP VALUE
  ---------------------------------------------------------------------------
```

Notas: *No se puede utilizar CS como registro destino en una instrucción POP.*

continua...

...sigue

XCHG = Intercambiar:

Registro/Memoria con Registro	1	0	0	0	0	1	1	w	mod	reg	r/m
Registro con Acumulador	0	1	0	1	1	reg					

```
XCHG            Intercambia Registros                  Flags: not altered

XCHG destination,source

          Logic:    destination <--> source

    XCHG conmuta los contenidos de sus operandos los cuales pueden ser o
    bytes o palabras.

  ---------------------------------------------------------------------------
   Operands                  Clocks   Transfers  Bytes   Example
                           byte(word)
   accumulator, reg16          3          -        1     XCHG AX,SI
   memory, register        17(25) + EA    2       2-4    LOCK XCHG SEMPHOR, DX
   register, register          4          -        2     XCHG CL,DL
  ---------------------------------------------------------------------------
```

Notas: *Usar en conjunciónconcon el prefijo LOCK, esta instrucción es particularmente útil en la implementación de semáforosd de recursos compartidos.*

IN = Entrada desde:

Puero Dado	1	1	1	0	0	1	0	w	puerto		
Puerto Variable	1	1	1	0	1	1	0	w			

```
IN              Input Byte or Word                     Flags: not altered

IN accumulator,port

          Logic:    accumulator <- (port)

    IN transfiere un byte o una palabra desde un puerto a AL o AX. El puerto
    puede ser especificado de forma inmediata (para puertos de 0 a 255) o
    mediante el registro DX (permite direccionar los 64K puertos)

  ---------------------------------------------------------------------------
   Operands                  Clocks   Transfers  Bytes   Example
                           byte(word)
   accumulator, immed8       10(14)       1        2     IN AL,45h
   accumulator, DX           8(12)        1        1     IN AX,DX
  ---------------------------------------------------------------------------
```

OUT = Sacar hacia:

Puero Dado	1	1	1	0	0	1	1	w	puerto		
Puerto Variable	1	1	1	0	1	1	1	w			

```
OUT             Output to Port                         Flags: not altered

OUT port,accumulator

          Logic:    (port) <- accumulator

    OUT transfiere un byte o una palabra desde AL o AX al puerto especificado.
    El puerto debe darse con una constate como dato inmediato (permite acceder
    a los 256 primeros puertos) o mediante una palabra situada en DX (con lo
    que podremos acceder a todo el rango de puertos, de 0 hasta 65536).

  ---------------------------------------------------------------------------
   Operands                  Clocks   Transfers  Bytes   Example
                           byte(word)
   immed8, accumulator       10(14)       1        2     OUT 254,AX
   DX, accumulator           8(12)        1        1     OUT DX,AL
  ---------------------------------------------------------------------------
```

XLAT=Recodifica un byte:

AL	1	1	0	1	0	1	1	1			

```
XLAT            Translate                              Flags: not altered

XLAT translate-table

          Logic:    AL <- (BX + AL)

    XLAT recodifica bytes mediante una tabla. Un puntero a una tabla de
    256 bytes es cargado en BX previamente. El byte que va a ser recodificado
    se carga en AL; este sirve como índice (ej. un offset) en la tabla de
    translación. Después de ejecutar XLAT, el byte en AL es remplazado por
    el byte localizado AL bytes desde el comienzo de la tabla de translación
    prefijado en BX.

  ---------------------------------------------------------------------------
   Operands                  Clocks   Transfers  Bytes   Example
   translate-table             11         1        1     XLAT SINE_TABLE
  ---------------------------------------------------------------------------

    El siguiente ejemplo translada un valor decimal (de 0 a 15) a su
    correspondiente dígito hexadecimal.

          lea     bx, hex_table          ;pone el puntero de la tabla en BX
          mov     al, decimal_digit      ;dígito a ser recodificado en AL
          xlat    hex_table              ;translate the value in AL
             .                           ;AL ahora contiene el valor ASCII
             .
  hex_table       db      '0123456789ABCDEF'
```

Notas:
La tabla de translación no puede tener más de 256 bytes
El operando, translate-table, es opcional ya que un puntero a la tabla debe ser cargado en BX antes de la ejecución de la operación.

...sigue

continua...

LEA = carga EA en::

Registro	1	0	0	0	1	1	0	1	mod	reg	r/m

```
LEA              Load Effective Address              Flags: not altered

LEA destination,source

          Logic:       destination <- Addr(source)

    LEA transfiere el offset del operando fuente (no su valor) al operando
    destino. La fuente debe debe ser una referencia de memoria, y el destino
    un registro de uso general de 16 bits.

  ---------------------------------------------------------------------------
   Operands                  Clocks   Transfers  Bytes   Example
   reg16, mem16              2 + EA       -       2-4    LEA BX,MEM_ADDR
  ---------------------------------------------------------------------------
```

Notas:
Esta intrucción tiene ventajas sobre la utilización del operador OFFSET con la intruciión MOV.
Con LEA el operando fuente puede ser ignorado a la hora de compilar. Por ejemplo:

```
                    LEA  BX, TABLE[SI]         ; Legal
                                 mientras que
                    MOV  BX, OFFSET TABLE[SI]     ; No es legal
```

Es ilegal, ya que el operador OFFSET realiza su cálculo en tiempo de ensablaje del programa pero su dirección no es conocida hasta que se ejecuta (¿dónde va a poner el cargador del DOS u otro sistema la TABLA?).

LDS = carga puntero en:

DS	1	0	0	0	1	1	0	1	mod	reg	r/m

```
LDS              Load Pointer using DS               Flags: not altered

LDS  destination,source

          Logic:    DS <- (source + 2)
                    destination <- (source)

    LDS carga en dos registros los 32 bits del puntero que se encuentra en
    la posición de memoria indicada en el campo fuente. LDS almacena  el
    valor del segmento (la palabra de mayor peso de la fuente) en DS y el
    valor del offset (la palabra de menor peso) en el registro destino. El
    registro destino debe ser cualquiera de los registros de 16 bits de uso
    general (los registros de segmento, nó son de uso general).

  ---------------------------------------------------------------------------
   Operands                  Clocks   Transfers  Bytes   Example
   reg16, mem32             24 + EA       2       2-4    LDS DI,32_POINTER
  ---------------------------------------------------------------------------
```

Nota: *LES, carga de puntero usando ES, es similar a LDS pero utilizando ES en lugar de DS.*

LES = carga puntero en::

ES	1	0	0	0	1	1	0	0	mod	reg	r/m

```
LES              Load Pointer using ES               Flags: not altered

LES  destination,source

          Logic:    ES <- (source + 2)
                    destination <- (source)

    LES carga en dos registros los 32 bits del puntero que se encuentra en
    la posición de memoria indicada en el campo fuente. LES almacena  el
    valor del segmento (la palabra de mayor peso de la fuente) en ES y el
    valor del offset (la palabra de menor peso) en el registro destino. El
    registro destino debe ser cualquiera de los registros de 16 bits de uso
    general (los registros de segmento, nó son de uso general).

  ---------------------------------------------------------------------------
   Operands                  Clocks   Transfers  Bytes   Example
   reg16, mem32             24 + EA       2       2-4    LES DI,32_POINTER
  ---------------------------------------------------------------------------
```

Nota: *LES, es funcionalmente igual a LDS exceptuando el destino del segmento; en LES el segmento se carga en ES y en LDS en ES.*

LAHF=Carga AH con:

"Flags"	1	0	0	1	1	1	1	1

```
LAHF             Load Register AH from Flags         Flags: not altered

          Logic:     AH bits    <-    Flag-reg bits
                    7 6 4 2 0   <-      S Z A P C

    LAHF copia los cinco 8080/8085 flags (Signo, Cero, Auxiliar Carry, Paridad
    y Carry) en los bits 7, 6, 4, 2, y 0 respectivamente del registro AH.
    Los Flags originales no son alterados por esta intrucción.

  ---------------------------------------------------------------------------
   Operands                  Clocks   Transfers  Bytes   Example
   no operands                 4          -        1     LAHF
  ---------------------------------------------------------------------------
```

Notas:
Esta intrucción es principalmente usada para compatibilidad entre la familia 8080/8085 y la familia 8088/8086.
Después de esta intrucción, los bits 1, 3 y 5 de AH son desconocidos.

...sigue

continua...

```
SAHF=Almacena AH en:
  "Flags"                          | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 |
                 SAHF              Store Register AH into Flags        Flags: O D I T S Z A P C
                                                                                 * * * * *
                          Logic:    Flag-reg bits   <-     AH bits
                                      S Z A P C     <-    7 6 4 2 0

                     SAHF copia los bits 7, 6, 4, 2 y 0 desde el registro AH a el registro de
                     Flags remplazando los anteriores valores de los Flags: Signo, Cero,
                     Auxiliar Carry, Paridad y Carry.

                   ---------------------------------------------------------------------------
                    Operands                  Clocks   Transfers  Bytes   Example
                    no operands                 4          -        1     SAHF
                   ---------------------------------------------------------------------------
                 Notas:
                 Los Flags de Desbordamiento (Overflow), Dirección, Interrupción y Trampa no son modificados
                 por esta instrucción.
                 La  compatibilidad  con  las  antiguas  familias  8080/8085  es  el  justificación  de  esta
                 intrucción.
PUSHF = Pone Flags en:
  Pila (Apila los flags)           | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 |
                 PUSHF             Push Flags onto Stack               Flags: not altered

                          Logic:    SP <- SP - 2
                                    (SP) <- flag-register

                     PUSHF decrementa SP en  y transfiere el registro de Flags a la nueva cabecera
                     de la pila.

                   ---------------------------------------------------------------------------
                    Operands                  Clocks   Transfers  Bytes   Example
                    no operands                14         1        1     PUSHF
                   ---------------------------------------------------------------------------
POPF = Recupera Flags de:
  Pila                             | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 |
                 POPF              POP Flags from the Stack            Flags: O D I T S Z A P C
                                                                              r r r r r r r r r
                          Logic:    flag-register <- (SP)
                                    SP <- SP + 2

                     POPF transfiere la palabra de la parte alta de la pila a el registro de
                     Flags remplazando las antiguas, acto seguido se incrementa SP en 2 unidades
                     para apuntar la nueva cavecera de la pila.

                   ---------------------------------------------------------------------------
                    Operands                  Clocks   Transfers  Bytes   Example
                    no operands                12         1        1     POPF
                   ---------------------------------------------------------------------------
```

Tab 1.- Instrucciones de Transferencia de Datos.

A3.2 CÁLCULO DE LA DIRECCIÓN EFECTIVA (Effective Address, EA):

Esta tabla lista el número de ciclos de reloj que se requieren para calcular la dirección efectiva (EA, a partir de ahora) en los microprocesadores originales de la familia 8088/8086. Los microprocesadores 80x8x al igual que los modelos VHDL, requieren muchos menos ciclos de reloj para calcular EA (debido a que lo hacen mediante lógica cableada y no a través de la ALU), por lo tanto, ésta tabla representa el peor de los casos.

COMPOSICIÓN DE EA	8088 RELOJES	EJEMPLO
Desplazamiento		
Registro indirecto	6	MOV AX,ADDR
BX, BP, SI, DI	5	MOV AX,[BP]
Desplazamiento + Base o Índice		
BX + Disp, BP + Disp	9	MOV AX,ADDR[BP]
SI + Disp, DI + Disp		
Base + Índice		
BP + DI, BX + SI	7	MOV AX,[BP+DI]
BP + SI, BX + DI	8	MOV AX,[BX+DI]
Desplazamiento + Base + Índice		
BP + DI + Disp	11	MOV AX,ADDR[BP+DI]
BX + SI + Disp		
BP + SI + Disp		
BX + DI + Disp	12	MOV AX,ADDR[BP+SI]

Notas:
Sumar 2 ciclos de reloj cuando hay un prefijo de segmento.
Cada referencia a memoria, requiere 4 ciclos de reloj adicionales. El campo "transferencias" en los diagramas de temporización de la instrucción, proporcionan el número de referencias a memoria para cada instrucción.

Tab 2.- Cálculo de la Dirección Efectiva.

A3.3 TABLA DE INSTRUCCIONES ARITMÉTICAS

INSTRUCCIONES ARITMÉTICAS																																
Mnemónico	Código De La Instrucción																															
ADD = Sumar:	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Registro/Memoria hacia/desde Registro	0	0	0	0	0	0	d	w	mod		reg		r/m																			
Inmediato a Registro/Memoria	1	0	0	0	0	0	s	w	mod	0	0	0	r/m				data L								data H si s:w = 01							
Inmediato a Acumulador	0	0	0	0	0	1	0	w	data L								data H si w = 1															

```
ADD              Addition                          Flags: O D I T S Z A P C
                                                          *       * * * * *

ADD destination,source

          Logic:    destination <- destination + source

    ADD suma los operandos y almacena el resultado en el destino. Ambos
    operandos pueden ser bytes o palabras y contener datos biarios con o
    sin signo.

  ---------------------------------------------------------------------------
   Operands                  Clocks   Transfers  Bytes   Example
                           Byte(word)
   register, register          3          -        2     ADD BX,CX
   accumulator, immediate      4          -       2-3    ADD AX,256
   register, immediate         4          -       3-4    ADD BL,4
   register, memory        9(13) + EA     1       2-4    ADD DI,[DX]
   memory, register       16(24) + EA     2       2-4    ADD TOTAL,BL
   memory, immediate      17(25) + EA     2       3-6    ADD RESULT,3
  ---------------------------------------------------------------------------
```

ADC = Sumar con Acarreo:																																
Registro/Memoria hacia/desde Registro	0	0	0	0	0	0	d	w	mod		reg		r/m																			
Inmediato a Registro/Memoria	1	0	0	0	0	0	s	w	mod	0	0	0	r/m				data L								data H si s:w = 01							
Inmediato a Acumulador	0	0	0	0	0	1	0	w	data L								data H si w = 1															

```
ADC              Add with Carry                    Flags: O D I T S Z A P C
                                                          *       * * * * *

ADC destination,source

          Logic:    destination <- destination + source + CF

    ADC suma los operandos, suma 1 si el flag de carry está activo, y pone
    el resultado en destino. Ambos operandos pueden ser bytes o palabras con
    o sin signo.

  ---------------------------------------------------------------------------
   Operands                  Clocks   Transfers  Bytes   Example
                           byte(word)
   register, register          3          -        2     ADC BX,SI
   register, immediate         4          -       3-4    ADC CX,128
   accumulator, immediate      4          -       2-3    ADC AL,10
   register, memory        9(13) + EA     1       2-4    ADC DX,RESULT
   memory, register       16(24) + EA     2       2-4    ADC BETA,DI
   memory, immediate      17(25) + EA     2       3-6    ADC GAMMA,16h
  ---------------------------------------------------------------------------
```

INC = Incrementar:																																
Registro/Memoria	1	1	1	1	1	1	1	w	mod	0	0	0	r/m																			
Registro	0	1	0	0	0	reg																										

```
INC              Increment                         Flags: O D I T S Z A P C
                                                          *       * * * *

INC destination

          Logic:    destination <- destination + 1

    INC suma 1 al destino. El destino, que puede ser byte o palabra, es
    considerado un número binario sin signo.

  ---------------------------------------------------------------------------
   Operands                  Clocks   Transfers  Bytes   Example
                           byte(word)
   reg16                       2          -        1     INC BX
   reg8                        3          -        2     INC BL
   memory                 15(23) + EA     2       2-4    INC THETA[BX]
  ---------------------------------------------------------------------------
```

Notas:
Esta instrucción no afecta al flag de carry. Si necesitamos sumar un 1 a un número que ocupa más de una palabra, entonces, las instrucciones ADD y ADC son las adecuadas.

AAA = Ajuste Ascii para la Suma del:																																
Acumulador	0	0	1	1	0	1	1	1																								

```
AAA              ASCII Adjust after Addition       Flags: O D I T S Z A P C
                                                          ?       ? ? * ? *

AAA

          Logic:    If (AL & 0Fh) > 9 or (AF = 1) then
                        AL <- AL + 6
                        AH <- AH + 1
                        AF <- 1;  CF <- 1
                    else
                        AF <- 0;  CF <- 0
                    AL <- AL & 0Fh

    AAA Convierte el numero de los 4 bits bajos (nibble) de AL en un número BCD
    desempaquetado (el nibble alto de AL está lleno de ceros).

  ---------------------------------------------------------------------------
```


continua...

...sigue

Operands	Clocks	Transfers	Bytes	Example
no operands	4	-	1	AAA

Si el nibble bajo de AL es mayor que 9, o el carry auxiliar está activo, esta intrucción convierte AL en su respectivo BCD sin empaquetar sumando 6 (subtrayendo 10) a AL; sumando 1 a AH; y estableciendo activos los flags de acarreo auxiliar y carry. Esta instrucción siempre pone a cero el nibble alto de AL.

Notas:
Los BCD sin empaquetar almacenan un dígito por byte; AH contiene el dígito más significativo y AL el de menor significado.

DAA = Ajuste Decimal de la Suma en:

Acumulador	0	0	1	0	0	1	1	1							

DAA Decimal Adjust after Addition Flags: O D I T S Z A P C
? * * * * *

DAA

```
Logic:    If (AL & 0Fh) > 9 or (AF = 1) then
              AL <- AL + 6
              AF <- 1
          else AF <- 0
          If (AL > 9Fh) or (CF = 1) then
              AL <- AL + 60h
              CF <- 1
          else CF <- 0
```

DAA corrige el resultado de una suma previa y lo convierte en dos operandos decimales empaquetados válidos (el resultado debe estar en AL). Esta instrucción cambia el contenido de AL por lo que contendrá un par de dígitos decimales empaquetados.

Operands	Clocks	Transfers	Bytes	Example
no operands	4	-	1	DAA

Notas:
Los BCD empaquetados almacenan un dígito por nibble (4 bits); el dígito menos significativo está en el nibble bajo.
No es posible aplicar un ajuste después de la división o multiplicación de números BCD empaquetados. Si necesitamos usar multiplicaciones o divisiones, es mejor utilizar números BCD sin empaquetar. Observar el código de ejemplo en la descripción de la instrucción AAM (ASCII Adjust después de la Multiplicación).

SUB = Restar:

Registro/Memoria hacia/desde Registro	0	0	1	0	1	0	d	w	mod	reg	r/m					
Inmediato desde Registro/Memoria	1	0	0	0	0	0	s	w	mod	1 0 1	r/m	data L				data H si s:w = 01
Inmediato desde Acumulador	0	0	0	0	0	1	0	w	data L			data H si w = 1				

SUB Subtract Flags: O D I T S Z A P C
* * * * * *

SUB destination,source

```
Logic:    destination <- destination – source
```

SUB resta el operando fuente del destino y almacena el resultado en el destino. Ambos operandos pueden ser bytes o palabras con o sin signo.

Operands	Clocks byte(word)	Transfers	Bytes	Example
register, register	3	-	2	SUB DX,BX
register, memory	9(13) + EA	1	2-4	SUB DX,TOTAL
memory, register	16(24) + EA	2	2-4	SUB RATE,CL
accumulator, immediate	4	-	2-3	SUB AH,25
register, immediate	4	-	3-4	SUB DX,5280
memory, immediate	17(25) + EA	2	3-6	SUB RESULT,1032

SBB = Restar con Borrow:

Registro/Memoria hacia/desde Registro	0	0	0	0	0	0	d	w	mod	reg	r/m					
Inmediato desde Registro/Memoria	1	0	0	0	0	0	s	w	mod	0 0 0	r/m	data L				data H si s:w = 01
Inmediato desde Acumulador	0	0	0	0	0	1	0	w	data L			data H si w = 1				

SBB Subtract with Borrow Flags: O D I T S Z A P C
* * * * * *

SBB destination,source

```
Logic:    destination <- destination - source - CF
```

SBB resta el operando fuente del destino, resta 1 de este resultado si el Flag de Carry está activo, y almacena el resultado en el destino. Los operandos pueden ser bytes o paleabras con o sin signo.

Operands	Clocks byte(word)	Transfers	Bytes	Example
register, register	3	-	2	SBB DX,AX
register, memory	9(13) + EA	1	2-4	SBB DX,FEE
memory, register	16(24) + EA	2	2-4	SBB SIGH,SI
accumulator, immediate	4	-	2-3	SBB AX,8
register, immediate	4	-	3-4	SBB BH,4
memory, immediate	17(25) + EA	2	3-6	SBB TOTAL,10

Notas:
SBB es útil para restar números que son de más de 16 bits, ya que resta un borrow (en el flag de carry) de la operación pervia.
Puede restarse una cantidad de un byte de lonqitud (dato inmediatamento) de un destino que es una palabra; en este caso , el byte (dato inmediato) es extendido manteniendo el signo para convertirlo en un dato de 16 bits antes de realizar la resta. De ésta forma el código ocupa menos espacio en memoria.

...sigue

continua...

DEC = Decrementar:

Registro/Memoria	1	1	1	1	1	1	1	w	mod	0	0	1	r/m		
Registro	0	1	0	0	1	reg									

```
DEC              Decrement                          Flags: O D I T S Z A P C
                                                           *       * * * *

DEC destination

          Logic:    destination <- destination - 1

    DEC decrementa el operando destino en una unidad. El operando destino,
    que puede ser una palabra o un byte, es tratado como un número sin signo.

  ---------------------------------------------------------------------------
   Operands                  Clocks   Transfers  Bytes   Example
   reg16                       2          -        1     DEC BX
   reg8                        3          -        2     DEC BL
   memory                 15(23) + EA     2       2-4    DEC MATRIX[SI]
  ---------------------------------------------------------------------------
```

Notas:
Esta instrucción no afecta el flag de carry, por eso, si necesitamos decrementar un número de múltiples palabras, es mejor utilizar las instrucciones SUB y SBB.

NEG = Cambio de signo:

Registro/Memoria	1	1	1	1	0	1	1	w	mod	0	1	1	r/m		

```
NEG              Negate                             Flags: O D I T S Z A P C
                                                           *       * * * * *

NEG destination

          Logic:    destination <- -destination      ; Complemento a dos

    NEG resta el operando destino de cero, o lo que es lo mismo, hace el
    complemento a uno al destino y le suma uno almacenando el resultado
    en el operando destino. Esta operación, efectivamente cambia el signo
    del operando que puede ser un byte o una palabra.

  ---------------------------------------------------------------------------
   Operands                  Clocks   Transfers  Bytes   Example
                           byte(word)
   register                    3          -        2     NEG DL
   memory                 16(24) + EA     2       2-4    NEG COEFFICIENT
  ---------------------------------------------------------------------------
```

Notas:
Si el operando es cero, se borra el flag de carry; en los restantes casos, el carry se pone a 1 lógico.
Una negación de un byte que contenga el valor –128 o una palabra que contenga –32,768, no causan cambios en el operando y activan el flag de desbordamiento (Overflow Flag).

CMP = Comparar:

Registro/Memoria y Registro	0	0	1	1	1	0	d	w	mod	reg			r/m		
Inmediato con Registro/Memoria	1	0	0	0	0	0	s	w	mod	1	1	1	r/m	data L	data H si s:w = 01
Inmediato con Acumulador	0	0	1	1	1	1	0	w	data L					data H si w = 1	

```
CMP              Compare                            Flags: O D I T S Z A P C
                                                           *       * * * * *

CMP destination,source

          Logic:    Los Flags cambian de acuerdo con el resultado de:
                       (destination - source)

    CMP compara dos números restándo la fuente del destino y actualizando
    los flags. CMP no modifica ni la fuente ni el destino. Los operandos
    pueden ser bytes o palabras.

  ---------------------------------------------------------------------------
   Operands                  Clocks   Transfers  Bytes   Example
                           byte(word)
   register, register          3          -        2     CMP CX,BX
   register, immediate         4          -       3-4    CMP BL,02h
   accumulator, immediate      4          -       2-3    CMP AL,00010110b
   register, memory        9(13) + EA     1       2-4    CMP DH,ALPHA_BETA
   memory, register        9(13) + EA     1       2-4    CMP TOTAL,SI
   memory, immediate      10(14) + EA     1       3-6    CMP VALUES,3420h
  ---------------------------------------------------------------------------
```

AAS = Ajuste Ascii para la Resta del:

Acumulador	0	0	1	1	0	1	1	1				

```
AAS              ASCII Adjust after Subtraction     Flags: O D I T S Z A P C
                                                            ?       ? ? * ? *

AAS

          Logic:    If (AL & 0Fh) > 9 or AF = 1 then
                        AL <- AL - 6
                        AH <- AH - 1
                        AF <- 1;  CF <- 1
                    else
                        AF <- 0;  CF <- 0
                    AL <- AL & 0Fh

    AAS corrige el resultado de una resta previa de dos operandos BCD
    válidos y sin empaquetar, cambiando el contenido de AL por un número
    BCD válido. El operando destino de la resta debe haber sido especificado
    perviamente en AL. El nibble de mayor orden de AL siempre es cero.
  ---------------------------------------------------------------------------
   Operands                  Clocks   Transfers  Bytes   Example
   no operands                 4          -        1     AAS
  ---------------------------------------------------------------------------
```

Notas:
Un dígito BCD sin empaquetar, almacena un dígito por byte; AH contiene el dígito más significativo y AL el menos significativo.

continua...

...sigue

DAS = Ajuste Decimal de la Resta en:

Acumulador	0	0	1	0	1	1	1	1

```
DAS              Decimal Adjust after Subtraction    Flags: O D I T S Z A P C
                                                            ?       * * * * *
DAS

          Logic:    If  (AL & 0Fh) > 9 or (AF = 1) then
                        AL <- AL - 6
                        AF <- 1
                    else AF <- 0
                    If (AL > 9Fh) or (CF = 1) then
                        AL <- AL - 60h
                        CF <- 1
                    else CF <- 0
```

DAS corrige el resultado de una resta de dos números decimales empaquetados válidos realizada previamente (el resultado debe estar en AL). Esta intrucción cambia el contenido de AL ya que contendrá una pareja de dígitos decimales empaquetados válida.

Operands	Clocks	Transfers	Bytes	Example
no operands	4	-	1	DAS

Notas:
Los BCD empaquetados almacenan un dígito por nibble (4 bits); el dígito menos significativo está en el nibble bajo.
No es posible aplicar un ajuste después de la división o multiplicación de números BCD empaquetados. Si necesitamos usar multiplicaciones o divisiones, es mejor utilizar números BCD sin empaquetar. Observar el código de ejemplo en la descripción de la instrucción AAM (ASCII Adjust después de la Multiplicación).

MUL = Multiplicar Sin Signo:

Registro/Memoria	1	1	1	1	0	1	1	w	mod	1	0	0	r/m

```
MUL              Multiply, Unsigned                  Flags: O D I T S Z A P C
                                                            *       ? ? ? ? *
MUL source

          Logic:    AX <- source * AL             ; if source is a byte
               or
                    DX:AX = source * AX          ; if source is a word
```

MUL realiza multiplicaciones sin signo. Si la fuente es un byte, MUL multiplica la fuente por AL devolviendo el producto en AX. Si la fuente es una palabra, MUL multiplica la fuente por a AX devolviendo el producto en DX:AX. Los Flags de Carry y Overflow, son activados si la mitad alta del resultado (AH para una fuente de byte, DX para un dato fuente de word) contiene algún dígito significativo del producto, si nó, se desactiva.

Operands	Clocks	Transfers	Bytes	Example
reg8	70-77	-	2	MUL CH
reg16	118-133	-	2	MUL BX
mem8	(76-83) + EA	1	2-4	MUL A_BYTE
mem16	(128-143) + EA	1	2-4	MUL A_WORD

IMUL=Multiplicar (Entera) con signo:

Registro/Memoria	1	1	1	1	0	1	1	w	mod	1	0	1	r/m

```
IMUL             Integer Multiply, Signed            Flags: O D I T S Z A P C
                                                            *       ? ? ? ? *
IMUL source

          Logic:    AX    <- AL * source          ; si la fuente es byte
               or
                    DX:AX <- AX * source          ; si la fuente es word
```

IMUL realiza multiplicaciones con signo. Si la fuente es un byte, IMUL multiplica la fuente y AL devolviendo el resultado en AX. Si la fuente es una palabra. IMUL multiplica AX y la fuente devolviendo el producto en DX:AX. Lod Flags de Carry y Overflow son activados si la mitad alta del resultado (AH para una fuente de un byte, DX si la fuente es una palabra) contiene algún dígito significante del producto, de otro modo, es borrado.

Operands	Clocks	Transfers	Bytes	Example
reg8	80-98	-	2	IMUL CL
reg16	128-154	-	2	IMUL BX
mem8	(86-104) + EA	1	2-4	IMUL BITE
mem16	(138-164) + EA	1	2-4	IMUL WORD[BP][DI]

AAM = Ajuste ASCII para la Mul:

Accumulador	1	1	0	1	0	1	0	0	0	0	0	0	1	0	1	0

```
AAM              ASCII Adjust after Multiply         Flags: O D I T S Z A P C
                                                            ?       * * ? * ?
AAM

          Logic:    AH <- AL / 10
                    AL <- AL MOD 10
```

AAM Esta instrucción corrige el resultado de una multiplicación previa de números en BCD sin empaquetar. Un bidígito válido de un número BCD sin empaqutar situado en AX, es ajustado posicionando el resultado en AX. Los nibbles de mayor orden de los operandos que fueron utilizados debían haber sido 0 para que esta intrucción produzca un resultado correcto.

continua...

...sigue

Operands	Clocks	Transfers	Bytes	Example
no operands	83	-	1	AAM

Notas:
Los números BCD sin empaquetar almacenan un dígito por byte; AH contiene el más significativo mientras que AL el de menor significado.

DIV = Dividir Sin Signo:

Registro/Memoria	1	1	1	1	0	1	1	w	mod	1	1	0	r/m

```
DIV              Divide, Unsigned                    Flags: O D I T S Z A P C
                                                            ?       ? ? ? ? ?

DIV source

          Logic:    AL <- AX / source                ; La fuente es byte
                    AH <- remainder
               or
                    AX <- DX:AX / source             ; La fuente es word
                    DX <- remainder
```

DIV esta instrucción realiza divisiones sin signo. Si la fuente es un byte, DIV divide el valor de AX por la fuente devolviendo el coeficiente en AL y el resto en AH. Si la fuente es una palabra, DIV divide la doble palabra situada en DX:AX por la fuente dando el coeficiente en AX y el resultado en DX.

Operands	Clocks	Transfers	Bytes	Example
reg8	0-90	-	2	DIV BL
reg16	4-162	-	2	DIV BX
mem8	(86-96) + EA	1	2-4	DIV VYUP
mem16	(154-172) + EA	1	2-4	DIV NCONQUER[SI]

Notas:
Si el resultado en demasiado largo para rellenar el destino (AL o AX), una INT 0 (División por Cero) es generada, y el coeficiente y resto no adquieren un valor definido.
Cuando una Interrupción 0 (División por Cero) aparece, en la familia 8088/8086, los registros CS:IP apunta a la instrucción siguiente a DIV que es la que ha dado el fallo, en la familia 80286 /80386, CS:IP apunta a DIV.

IDIV = División (Entera) con signo:

Registro/Memoria	1	1	1	1	0	1	1	w	mod	1	1	1	r/m

```
IDIV             Integer Divide, Signed              Flags: O D I T S Z A P C
                                                            ?       ? ? ? ? ?

IDIV source

          Logic:    AL <- AX / source                 ; fuente Byte
                    AH <- remainder
               or
                    AX <- DX:AX / source              ; fuente Word
                    DX <- remainder
```

IDIV esta instrucción realiza divisiones con signo. Si la fuente es un byte, IDIV divide el valor de AX por la fuente devolviendo el coeficiente en AL y el resto en AH. Si la fuente es una palabra, IDIV divide la doble palabra situada en DX:AX por la fuente dando el coeficiente en AX y el resultado en DX.

Operands	Clocks	Transfers	Bytes	Example
reg8	101-112	-	2	IDIV CL
reg16	165-184	-	2	IDIV DX
mem8	(107-118) + EA	1	2-4	IDIV BYTE[SI]
mem16	(175-194) + EA	1	2-4	IDIV [BX].WORD_ARRAY

Notas:
Si el resultado en demasiado largo para rellenar el destino (AL o AX), una INT 0 (División por Cero) es generada, y el coeficiente y resto no adquieren un valor definido.
Cuando una Interrupción 0 (División por Cero) aparece, en la familia 8088/8086, los registros CS:IP apunta a la instrucción siguiente a DIV que es la que ha dado el fallo, en la familia 80286 /80386, CS:IP apunta a DIV.

AAD = Ajuste ASCII para la Div:

Accumulador	1	1	0	1	0	1	0	1	0	0	0	0	1	0	1	0

```
AAD              ASCII Adjust before Division        Flags: O D I T S Z A P C
                                                            ?       * * ? * ?

AAD

          Logic:    AL <- AH * 10 + AL
                    AH <- 0
```

AAD convierte el bidígito de un múmero BCD sin empaquetar situado en AX en otro binario preparado para la división usando DIV o IDIV, intrucciones que requieren operandos binarios en lugar de BCD.

Operands	Clocks	Transfers	Bytes	Example
no operands	60	-	2	AAD

AAD modifica el numerador situado en AL, por esto, el resultado producido por la subsecuente operación de división dará un número BCD sin empaquetar válido. Para que la división se efectúe correctamente, AH debe ser 0. Después de la división, el coeficiente se sitúa en AL, y el resto en AH. Ambos dos tienen sus nibbles altos a cero.

Notas:
Los números BCD sin empaquetar almacenan un dígito por byte; AH contiene el más significativo mientras que AL el de menor significado.

continua...

...sigue

CBW = Convertir Byte en Word de:

Acumulador	1	0	0	1	1	0	0	0			

```
CBW              Convert Byte to Word                Flags: not altered

CBW

          Logic:    if (AL < 80h) then
                        AH <- 0
                    else
                        AH <- FFh

    CBW extiende el bit de signo del regitro AL en AH. Esta intrucción extiende
    el valor de un byte con signo a su equivalente palabra con signo.

  ---------------------------------------------------------------------------
   Operands                  Clocks   Transfers  Bytes   Example
   no operands                 2          -        1     CBW
  ---------------------------------------------------------------------------
```

Notas:
Esta intrucción establecerá AH al valor 0FFh si el bit de signo (bit 7) del registro AL está activo; si el bit 7 de AL es cero, entonces AH es rellenado con ceros. Esta intrucción es útil para generar palabras desde un byte generado anteriornete para realizar una división de byte.

CWD=Convertir Word en 2-Word de:

Acumulador	1	0	0	1	1	0	0	1			

```
CWD              Convert Word to Doubleword          Flags: not altered

CWD

          Logic:    if (AX < 8000h) then
                        DX <- 0
                    else
                        DX <- FFFFh

    CWD extiende el bit de signo del registro AX al registro DX. Esta instrucción
    genera la doble palabra equivalente de el número con signo presente en
    el registro AX.

  ---------------------------------------------------------------------------
   Operands                  Clocks   Transfers  Bytes   Example
   no operands                 5          -        1     CWD
  ---------------------------------------------------------------------------
```

Notas:
Esta intrucción establecerá DX al valor 0FFFFh si el bit de signo (bit 15) del registro AX está activo; si el bit 15 de AX es cero, entonces DX será rellenado con ceros.

Tab 3.- Instrucciones Aritméticas.

A3.4 TABLA DE INSTRUCCIONES LÓGICAS

INSTRUCCIONES LÓGICAS																																
Mnemónico	Código De La Instrucción																															
NOT = Invertir:	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Registro/Memoria	1	1	1	1	0	1	1	w	mod		0	1	0		r/m																	

```
NOT                 Logical NOT                         Flags: not altered

NOT destination

          Logic:    destination <- NOT(destination)      ; Complemento a uno

    NOT invierte cada bit de su operando (crea el complemento a uno)
    que puede ser un byte o una palabra.

                        NOT Instrucción lógica

                        Destination    Result
                             0           1
                             1           0

  ---------------------------------------------------------------------------
   Operands                  Clocks   Transfers  Bytes   Example
                           byte(word)
   register                    3          -        2     NOT DX
   memory                  16(24) + EA     2       2-4    NOT MASK
  ---------------------------------------------------------------------------
```

SHL/SAL = Desplazamiento Lógico/Aritmético a la Izquierda:																																
Registro/Memoria	1	1	0	1	0	0	v	w	mod		1	0	0		r/m																	

```
SHL                 Shift Logical Left                  Flags: O D I T S Z A P C
SAL                 Shift Arithmetic Left                      *       * * ? * *

SHL destination,count
SAL destination,count
```

```
    SHL es la misma instrucción que SAL, Desplazamiento Aritmético a la
    Izquierda. SHL/SAL desplaza la palabra o el byte del destino hacia la
    izquierda el número de bits(posiciones) indicado en el segundo operando,
    el CONTADOR. Como los bits son desplazados hacia la izquierda (hacia el
    orden superior), el espacio vacío dejado en la derecha (menor orden) es
    rellenado con ceros. El Flag de Carry es actualizado para indicar el
    último bit desplazado fuera del destino.

    Si el CONTADOR no es igual a 1, el flag de Overflow no adquiere un valor
    definido. Si el CONTADOR es igual a 1, el Flag de Overflow será borrado
    si los dos bits de mayor peso del operando original son iguales, de lo
    contrario, será activado.

  ---------------------------------------------------------------------------
   Operands                  Clocks   Transfers  Bytes   Example
                           byte(word)
   register, 1                 2          -        2     SHL AL,1
   register, CL            8 + 4/bit      -        2     SHL SI,CL
   memory, 1               15(23) + EA     2       2-4    SHL WORD,1
   memory, CL            20(28)+EA+4/bit   2       2-4    SHL BYTE,CL
  ---------------------------------------------------------------------------
```

Notas:

El CONTADOR normalmente está indicado en el registro CL. Sin embargo, si queremos desplazar solamente una posicióm, debemos sustituir CL por 1, como se puede observar en el primer ejemplo.

La familia 80286/80386 limitan el valor del CONTADOR a 31. Si el contador es mayor de 31, estos microprocesadores usan COUNTADOR MOD 32 para generar un nuevo CONTADOR entre 0 y 31. Este filtro paso bajo existe para limitar la cantidad de tiempo en dar respuesta a una interrupción (tiempo de latencia, latency time), ya que las interrupciones no son atendidas hasta que termina de ejecutarse la instrucción actual. La familia 8088/8086 no posee esta mejora.

Múltiples instrucciones SHL/SAL que usen 1 como CONTADOR, pueden ser más rápidas y requieren menos memoria que una instrucción SHL/SAL que utiliza CL como contador.

SHR = Desplazamiento Lógico a la Derecha:																																
Registro/Memoria	1	1	0	1	0	0	v	w	mod		1	0	1		r/m																	

```
SHR                 Shift Logical Right                 Flags: O D I T S Z A P C
                                                               *       * * ? * *
SHR destination,count
```

```
    SAR desplaza los bits de destino hacia la derecha el número de posiciones
    indicado en el operando contador (o en CL, si no se incluye contador).
    Por la izquierda son inserados ceros. Si el bit de signo mantiene su
    valor original, el flag de desbordamiento es borrado; si el bit de signo
    (el bit más significativo) cambia, se activará el flag de desbordamiento.
    El flag de acarrea es actualizado pata reflejar el último bit desplazado
    fuera por la derecha.

    Si el CONTADOR no es igual a 1, el flag de Overflow queda sin definir,
    por el contrario, el flag de desbordamiento es actualizado con el bit más
    significativo del operando original.
```


continua...

...sigue

Operands	Clocks	Transfers	Bytes	Example
register, 1	2	-	2	SHR SI,1
register, CL	8 + 4/bit	-	2	SHR SI,CL
memory, 1	15 + EA	2	2-4	SHR ID_BYTE[SI][BX],1
memory, CL	20 + EA + 4/bit	2	2-4	SHR INPUT_WORD,CL

Notas:
El CONTADOR normalmente está indicado en el registro CL. Sin embargo, si queremos desplazar solamente una posicióm, debemos sustituir CL por 1, como se puede observar en el primer ejemplo.
La familia 80286/80386 limitan el valor del CONTADOR a 31. Si el contador es mayor de 31, estos microprocesadores usan COUNTADOR MOD 32 para generar un nuevo CONTADOR entre 0 y 31. Este filtro paso bajo existe para limitar la cantidad de tiempo en dar respuesta a una interrupción (tiempo de latencia, latency time), ya que las interrupciones no son atendidas hasta que termina de ejecutarse la instrucción actual. La familia 8088/8086 no posee esta mejora.
Múltiples instrucciones SHR que usen 1 como CONTADOR, pueden ser más rápidas y requieren menos memoria que una instrucción SHR que utiliza CL como contador.
El flag de desbordamiento no adquiere un valor definido cuando el contador es mayor de 1.

SAR = Desplazamiento Aritm a la Derecha:

Registro/Memoria	1 1 0 1 0 0 v w	mod 1 1 1 r/m	

SAR Shift Arithmetic Right Flags: O D I T S Z A P C
* * * ? * *

SAR destination,count

SAR desplaza la palabra o el byte del destino hacia la derecha el número de bits (posiciones) indicado en el segundo operando, el CONTADOR. Como los bits son desplazados hacia la derecha (hacia el orden inferior), el espacio vacío dejado en la izquierda es rellenado con el bit de signo (bit más significativo) del operando original, de esta manera mantenemos el signo original (desplazamiento aritmético). El Flag de Carry es actualizado para indicar el último bit desplazado fuera del destino por la derecha.

Si el CONTADOR no es igual a 1, el flag de Overflow no adquiere un valor definido. Si el CONTADOR es igual a 1, el Flag de Overflow será borrado.

Operands	Clocks byte(word)	Transfers	Bytes	Example
register, 1	2	-	2	SAR DX,1
register, CL	8 + 4/bit	-	2	SAR DI,CL
memory, 1	15(23) + EA	2	2-4	SAR N_BLOCKS,1
memory, CL	20(28)+EA+4/bit	2	2-4	SAR N_BLOCKS,CL

Notas:
El CONTADOR normalmente está indicado en el registro CL. Sin embargo, si queremos desplazar solamente una posicióm, debemos sustituir CL por 1, como se puede observar en el primer ejemplo.
La familia 80286/80386 limitan el valor del CONTADOR a 31. Si el contador es mayor de 31, estos microprocesadores usan COUNTADOR MOD 32 para generar un nuevo CONTADOR entre 0 y 31. Este filtro paso bajo existe para limitar la cantidad de tiempo en dar respuesta a una interrupción (tiempo de latencia, latency time), ya que las interrupciones no son atendidas hasta que termina de ejecutarse la instrucción actual. La familia 8088/8086 no posee esta mejora.
Múltiples instrucciones SHR que usen 1 como CONTADOR, pueden ser más rápidas y requieren menos memoria que una instrucción SHR que utiliza CL como contador.

ROL = Rotar a la izquierda:

Registro/Memoria	1 1 0 1 0 0 v w	mod 0 0 0 r/m	

ROL Rotate Left Flags: O D I T S Z A P C
* *

ROL destination,count

ROL desplaza la palabra o el byte del destino hacia la izquierda el número de bits (posiciones) indicado en el segundo operando, el CONTADOR. El bit desplazado fuera por la izquierda, es realimentado en el espacio vacío que queda en la drecha (bit de menor orden). El Flag de Carry es actualizado para indicar el último bit desplazado fuera del destino por la izquierda.

Si el CONTADOR no es igual a 1, el flag de Overflow queda indefinido. Si el CONTADOR es igual a 1, el Flag de Overflow es actualizado con la operación XOR de los 2 bits más significativos del operando original.

Operands	Clocks byte(word)	Transfers	Bytes	Example
register, 1	2	-	2	ROL DI,1
register, CL	8 + 4/bit	-	2	ROL BX,CL
memory, 1	15(23) + EA	2	2-4	ROL BYTE,1
memory, CL	20(28)+EA+4/bit	2	2-4	ROL WORD,CL

continua...

...sigue

Notas:
El CONTADOR normalmente está indicado en el registro CL. Sin embargo, si queremos desplazar solamente una posicióm, debemos sustituir CL por 1, como se puede observar en el primer ejemplo.
La familia 80286/80386 limitan el valor del CONTADOR a 31. Si el contador es mayor de 31, estos microprocesadores usan COUNTADOR MOD 32 para generar un nuevo CONTADOR entre 0 y 31. Este filtro paso bajo existe para limitar la cantidad de tiempo en dar respuesta a una interrupción (tiempo de latencia, latency time), ya que las interrupciones no son atendidas hasta que termina de ejecutarse la instrucción actual. La familia 8088/8086 no posee esta mejora.
Múltiples instrucciones ROL que usen 1 como CONTADOR, pueden ser más rápidas y requieren menos memoria que una instrucción ROL que utiliza CL como contador.

ROR = Rotar a la Derecha:

Registro/Memoria	1	1	0	1	0	0	v	w	mod	0	0	1	r/m

```
ROR              Rotate Right                    Flags: O D I T S Z A P C
                                                        *             *

ROR destination,count
```

ROR desplaza la palabra o el byte del destino hacia la derecha el número de bits (posiciones) indicado en el segundo operando, el CONTADOR. El bit desplazado fuera por la derecha (bit de menor orden) del destino, es realimentado en el espacio vacío que queda en la izquierda (bit de mayor orden). El Flag de Carry es actualizado para indicar el último bit desplazado fuera del destino por la derecha.

Si el CONTADOR no es igual a 1, el Flag de Overflow queda indefinido, de lo contrario, el Flag de Overflow es actualizado con la operación XOR de los 2 bits más significativos del resultado.

Operands	Clocks byte(word)	Transfers	Bytes	Example
register, 1	2	-	2	ROR BL,1
register, CL	8 + 4/bit	-	2	ROR AX,CL
memory, 1	15(23) + EA	2	2-4	ROR WORD,1
memory, CL	20(28)+EA+4/bit	2	2-4	ROR BYTE,CL

Notas:
El CONTADOR normalmente está indicado en el registro CL. Sin embargo, si queremos desplazar solamente una posicióm, debemos sustituir CL por 1, como se puede observar en el primer ejemplo.
La familia 80286/80386 limitan el valor del CONTADOR a 31. Si el contador es mayor de 31, estos microprocesadores usan COUNTADOR MOD 32 para generar un nuevo CONTADOR entre 0 y 31. Este filtro paso bajo existe para limitar la cantidad de tiempo en dar respuesta a una interrupción (tiempo de latencia, latency time), ya que las interrupciones no son atendidas hasta que termina de ejecutarse la instrucción actual. La familia 8088/8086 no posee esta mejora.
Múltiples instrucciones ROR que usen 1 como CONTADOR, pueden ser más rápidas y requieren menos memoria que una instrucción ROR que utiliza CL como contador.

RCL = Rotar a la Izquierda a través del Carry:

Registro/Memoria	1	1	0	1	0	0	v	w	mod	0	1	0	r/m

```
RCL              Rotate through Carry Left       Flags: O D I T S Z A P C
                                                        *             *

RCL destination,count
```

RCL desplaza la palabra o el byte del destino hacia la izquierda el número de bits (posiciones) indicado en el segundo operando, el CONTADOR. El bit desplazado fuera por la izquierda reemplaza el flag de carry que es realimentado al espacio vacío que queda en la drecha (bit de menor orden). Esta “rotación de bit” se repite el número de veces indicado en el CONTADOR.(Se puede considerar que el flag de carry es el bit más significativo del opreando que va a ser rotado).

Si el CONTADOR no es igual a 1, el flag de Overflow queda indefinido. Si es igual a 1, el Flag de Overflow es actualizado con la operación XOR de los 2 bits más significativos del operando original.

Operands	Clocks byte(word)	Transfers	Bytes	Example
register, 1	2	-	2	RCL CX,1
register, CL	8 + 4/bit	-	2	RCL BL,CL
memory, 1	15(23) + EA	2	2-4	RCL MULTIPLY_X_2,1
memory, CL	20(28)+EA+4/bit	2	2-4	RCL MOVE_AROUND,CL

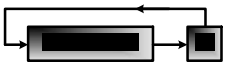
Notas:
El CONTADOR normalmente está indicado en el registro CL. Sin embargo, si queremos desplazar solamente una posicióm, debemos sustituir CL por 1, como se puede observar en el primer ejemplo.
La familia 80286/80386 limitan el valor del CONTADOR a 31. Si el contador es mayor de 31, estos microprocesadores usan COUNTADOR MOD 32 para generar un nuevo CONTADOR entre 0 y 31. Este filtro paso bajo existe para limitar la cantidad de tiempo en dar respuesta a una interrupción (tiempo de latencia, latency time), ya que las interrupciones no son atendidas hasta que termina de ejecutarse la instrucción actual. La familia 8088/8086 no posee esta mejora.
Múltiples instrucciones RCL que usen 1 como CONTADOR, pueden ser más rápidas y requieren menos memoria que una instrucción RCL que utiliza CL como contador.

continua...

RCR = Rotar a la Derecha a través del Carry:

Registro/Memoria	1 1 0 1 0 0 v w	mod 0 1 1 r/m		

```
RCR              Rotate through Carry Right          Flags: O D I T S Z A P C
                                                            *               *

RCR destination,count
```



RCR desplaza la palabra o el byte del destino hacia la derecha el número de bits (posiciones) indicado en el segundo operando, el CONTADOR. El bit desplazado fuera por la derecha (bit de menor orden) del destino, reemplaza el flag de carry que es realimentado en el espacio vacío que queda en la izquierda (bit de mayor orden). Esta "rotación de bit" se repite el número de veces indicado en el CONTADOR.(Se puede considerar que el flag de carry es el bit más significativo del opreando que va a ser rotado).

Si el CONTADOR no es igual a 1, el Flag de Overflow queda indefinido, de lo contrario, el Flag de Overflow es actualizado con la operación XOR de los 2 bits más significativos del resultado.

Operands	Clocks byte(word)	Transfers	Bytes	Example
register, 1	2	-	1	RCR CX,1
register, CL	8 + 4/bit	-	2	RCR DL,CL
memory, 1	15(23) + EA	2	2-4	RCR DIVIDE_BY_2,1
memory, CL	20(28)+EA+4/bit	2	2-4	RCR AROUND_MOVE,CL

Notas:
El CONTADOR normalmente está indicado en el registro CL. Sin embargo, si queremos desplazar solamente una posicióm, debemos sustituir CL por 1, como se puede observar en el primer ejemplo.
La familia 80286/80386 limitan el valor del CONTADOR a 31. Si el contador es mayor de 31, estos microprocesadores usan COUNTADOR MOD 32 para generar un nuevo CONTADOR entre 0 y 31. Este filtro paso bajo existe para limitar la cantidad de tiempo en dar respuesta a una interrupción (tiempo de latencia, latency time), ya que las interrupciones no son atendidas hasta que termina de ejecutarse la instrucción actual. La familia 8088/8086 no posee esta mejora.
Múltiples instrucciones RCR que usen 1 como CONTADOR, pueden ser más rápidas y requieren menos memoria que una instrucción RCR que utiliza CL como contador.

AND = And Lógica:

Registro/Memoria con Registro	0 0 1 0 0 0 d w	mod reg r/m		
Inmediato con Registro/Memoria	1 0 0 0 0 0 0 w	mod 1 0 0 r/m	data L	data H si w = 01
Inmediato con Acumulador	0 0 1 0 0 1 0 w	data L	data H si w = 1	

```
AND              Logical AND                         Flags: O D I T S Z A P C
                                                            0       * * ? * 0

AND destination,source

          Logic:    destination <- destination AND source
```

AND realiza una operación lógica AND bit a bit con sus operandos y almacena el resultado en el destino. Los operandos pueden ser bytes o palabras.

AND Instrucción Lógica

Destination	Source	Result
0	0	0
0	1	0
1	0	0
1	1	1

AND pone a uno cada bit del resultado si los correspondientes bits de los operandos son ambos 1 lógico.

Operands	Clocks byte(word)	Transfers	Bytes	Example
register, register	3	-	2	AND AL,DL
register, immediate	4	-	3-4	AND CX,0FFh
accumulator, immediate	4	-	2-3	AND AX,01000010b
register, memory	9(13) + EA	1	2-4	AND CX,MASK
memory, register	16(24) + EA	2	2-4	AND VALUE,BL
memory, immediate	17(25) + EA	2	3-6	AND STATUS,01h

TEST = Función AND con los Flags. Sin Resultado:

Registro/Memoria y Registro	1 0 1 0 0 1 0 w	mod reg r/m		
Inmediato y Registro/Memoria	1 1 1 1 0 1 1 w	mod 0 0 0 r/m	data L	data H si w = 01
Inmediato y Acumulador	0 0 1 1 0 1 0 w	data L	data H si w = 1	

```
TEST             Test                                Flags: O D I T S Z A P C
                                                            0       * * ? * 0

TEST destination,source

          Logic:    (destination AND source)            ; Set flags only
                    CF <- 0
                    OF <- 0
```

TEST realiza una operación AND lógica entre sus dos operandos y actualiza los flags. Ni la fuente ni el destino modifican su contenido.

continua...

```
---------------------------------------------------------------------------
Operands                  Clocks   Transfers  Bytes   Example
                        byte(word)
register, register          3          -        2     TEST SI,DX
register, memory        9(13) + EA     1       2-4    TEST SI,MASK
accumulator, immediate      4          -       2-3    TEST AL,00000100b
register, immediate         5          -       3-4    TEST CX,1234
memory, immediate        11 + EA       -       3-6    TEST PARAM,1F1Fh
---------------------------------------------------------------------------
```

TEST es útil para examinar el estado de bits individuales. Por ejemplo, el siguiente código transferirá el control a ONE_FIVE_OFF si los bits uno y cinco del registro AL son cero. El estado del resto de los bits será ignorado.

```
          TEST    AL,00100010b    ;Tapa todos los bits excepto 1 y 5
          JZ      ONE_FIVE_OFF    ;Si 1 o 5 son 1 lógico, no se activa Z
NOT_BOTH:         .               ;Uno o ambos bits eran 1 lógico.

ONE_FIVE_OFF:                     ;Los bits 1 y 5 eran 0 lógico.
```

OR = Or Lógica:

Registro/Memoria con Registro	0 0 0 0 1 0 d w	mod reg r/m		
Inmediato con Registro/Memoria	1 0 0 0 0 0 0 w	mod 0 0 1 r/m	data L	data H si w = 01
Inmediato con Acumulador	0 0 1 0 0 1 0 w	data L	data H si w = 1	

```
OR                Logical OR                          Flags: O D I T S Z A P C
                                                                 0       * * ? * 0

OR destination,source

          Logic:    destination <- destination OR source
```

OR realiza una operación lógica OR bit a bit con sus operandos y almacena el resultado en el destino. Los operandos pueden ser bytes o palabras.

```
                    OR Instruction Logic
              Destination  Source    Result
                   0         0         0
                   0         1         1
                   1         0         1
                   1         1         1
```

OR pone a uno cada bit del resultado si cualquiera de los correspondientes bits de los operandos es 1 lógico.

```
---------------------------------------------------------------------------
Operands                  Clocks   Transfers  Bytes   Example
                        byte(word)
register, register          3          -        2     OR CH,DL
register, memory        9(13) + EA     1       2-4    OR BX, MEM_OR_Y
memory, register       16(24) + EA     2       2-4    OR MEM_OR_Y, BX
accumulator, immediate      4          -       2-3    OR AL,01110110b
register,immediate          4          -       3-4    OR CX,00FFh
memory, immediate      17(25) + EA     2       3-6    OR MEM_WORD,76h
---------------------------------------------------------------------------
```

XOR = OR exclusiva Lógica:

Registro/Memoria con Registro	0 0 1 1 0 0 d w	mod reg r/m		
Inmediato con Registro/Memoria	1 0 0 0 0 0 0 w	mod 1 1 0 r/m	data L	data H si w = 01
Inmediato con Acumulador	0 0 1 1 0 1 0 w	data L	data H si w = 1	

```
XOR               Exclusive OR                        Flags: O D I T S Z A P C
                                                                 0       * * * * 0

XOR destination,source

          Logic:    destination <- destination XOR source
```

XOR realiza una operación lógica "OR exclusiva" bit a bit con sus operandos y almacena el resultado en el destino. Los operandos pueden ser bytes o palabras.

```
                    XOR Instruction Logic
              Destination  Source    Result
                   0         0         0
                   0         1         1
                   1         0         1
                   1         1         0
```

XOR pone a uno cada bit del resultado sólo si uno de los correspondientes bits de los operandos es 1 lógico.

```
---------------------------------------------------------------------------
Operands                  Clocks   Transfers  Bytes   Example
register, register          3          -        2     XOR CX,BX
register, memory        9(13) + EA     1       2-4    XOR CL,MASK_BYTE
memory, register       16(24) + EA     2       2-4    XOR ALPHA[SI],DX
accumulator, immediate      4          -       2-3    XOR AL,01000001b
register, immediate         4          -       3-4    XOR SI,00C2h
memory, immediate      17(25) + EA     2       3-6    XOR RETURN_CODE,0D2h
---------------------------------------------------------------------------
```

Tab 4.- Instrucciones Lógicas.

A3.5 TABLA DE INSTRUCCIONES CON CADENAS DE DATOS

INSTRUCCIONES DE CADENAS

REP (Prefijo) = Repite:	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
String Instruction	1	1	1	1	0	0	1	z																								

```
REP              Repeat                                  Flags: not altered

REP  string-instruction

          Logic:    while CX <> 0                        ;para MOVS, LODS o STOS
                        ejecuta la intrucción de cadenas
                        CX <- CX – 1
                    ------------------------------------------
                    while CX <> 0
                        ejecuta la intrucción de cadenas ;para CMPS o SCAS
                        CX <- CX - 1
                        if ZF = 0 terminate loop

    REP es un prefijo que puede ser especificado antes de cualquier intrucción
    de manipulación de cadenas (CMPS, LODS, MOVS, SCAS y STOS). REP hace que la
    instrucción de manipulación de cadenas (string-instruction) siguiente se
    repita el número de veces indicado en CX; CX es decrementado después de cada
    ejecución de la "string instruction". (Para CMPS y SCAS, REP también dejará
    de repetirse si la Bandera de Cero (Zero Flag) es cero antes de la ejecución
    de la "string instruction".

  ---------------------------------------------------------------------------
   Operands                  Clocks   Transfers  Bytes   Example
      -                        2          -        1     REP MOVS TO,FROM
  ---------------------------------------------------------------------------

 ------------------------------- Ejemplo ---------------------------------

  El siguiente ejemplo mueve 100 bytes deste BUFFER1 a BUFFER2:
        CLD                     ;Mueve hacia delante
        LEA     SI, BUFFER1     ;El puntero fuente en SI
        LEA     DI, BUFFER2     ;  ...y el destino en DI
        MOV     CX,100          ;REP usa CX como contador
        REP     MOVSB           ;  ...realiza la operación.
```

Notas:

Si CX es inicialmente 0, la "string struction" es ignorada completamente.
El test de CX igual a 0 es realizado antes de la ejecución de la instrucción. El test para el Flag de Cero (realizado sólo en CMPS y SCAS) es realizado después de que la instrucción sea ejecutada.
REP, REPE (repetir mientras sea igual), y REPZ (repetir mientras sea cero) son iguales para la misma instrucción.
REPNZ (repetir mientras no sea cero) es similar a REP, pero usada con CMPS y SCAS, terminará cuando el flag de cero sea 1 lógico, en lugar de 0 lógico.
REP es usada generalmente con las instrucciones MOVS (mover cadena) y STOD (almacenar cadena); puede ser algo como "repetir hasta que no sea el final de la cadena".
No es necesario inicializar el flag de cero (ZF) antes de usar las instrucciones de manipulación de cadenas.
Una instrucción de repetición que es interrumpida mientras se ejecutaba, puede ser retomada corréctamente desde el retorno de una interrupción. Sin embargo, si otros prefijos son usados en una instrucción simple (por ejemplo, prefijos de segmento o de bloqueo LOCK) junto con REP, todos los prefijos excepto el que precede inmediatamente a la "instruction string" se perderán. De todas formas, si queremos usar más de un perfijo en una instrucción simple, debemos deshabilitar las interrupciones antes de la instrucción, y habilitarlas después. Hay que tener en cuenta que esta precaución no protege ante la aparición de interrupciones no enmascarables, y que operaciones con cadenas largas pueden causar mucho retraso en el proceso de la interrupción (latency time muy elevado).

MOVS = Mover Cadena:

Byte/Word	1	0	1	0	0	1	0	w																								

```
MOVS             Move String (Byte or Word)          Flags: not altered

MOVS destination-string,source-string

          Logic:    (ES:DI) <. (DS:SI)
                    if DF = 0
                        SI <- SI + n         ; n = 1 para byte, 2 para word
                        DI <- DI + n
                    else
                        SI <- SI – n
                        DI <- DI – n

    MOVS esta instrucción copia el byte o palabra del puntero formado por
    DS:SI, a la localización indicada en ES:DI. Después del movimiento, SI
    y DI son incrementados (si el flag de dirección está borrado) o
    decrementados (si está el flag de dirección activado), para apuntar al
    siguiente elemento de la cadena.

  ---------------------------------------------------------------------------
   Operands                  Clocks   Transfers  Bytes   Example
                           byte(word)
   dest,source               18(26)       2        1     MOVS WORD_BUFF,INPUT
   (repeat) dest,source   9+17(25)/rep  2/rep      1     REP MOVSW
  ---------------------------------------------------------------------------

  ------------------------------- Ejemplo ---------------------------------

    Asumiendo que BUFFER1 ha sido definido en algún lugar como:
          BUFFFER1        DB      100 DUP (?)
    el siguiente ejemplo, mueve 100 bytes de BUFFER1 a BUFFER2:

          CLD                     ;Mueve hacia delante
          LEA     SI, BUFFER1     ;Dirección fuente en SI
          LEA     DI, BUFFER2     ;Dirección destino en DI
          MOV     CX,100          ;CX es usado por el prefijo REP
  REP     MOVS    BUFFER2, BUFFER1 ;    ...y efectúa el movimiento.
```

...sigue

continua...

Notas:
Esta instrucción es siempre codificada por el ensamblador como una instrucción MOVSB, Move String Byte; o MOSW, Move String Word, dependiendo de si la cadena fuente se refiere a una cadena de bytes o de palabras. En cualquier caso, debemos cargar los registros SI y DI con los offset de las cadenas de los datos fuente y destino.

CMPS = Comparar una Cadena:

Byte/Word	1	0	1	0	0	1	1	w

```
CMPS            Compare String (Byte or Word)       Flags: O D I T S Z A P C
                                                           *       * * * * *

CMPS destination-string,source-string

          Logic:    CMP (DS:SI), (ES:DI)        ; Afecta sólo a los flags
                    if DF = 0
                        SI <- SI + n            ; n = 1 para byte, 2 para palabra
                        DI <- DI + n
                    else
                        SI <- SI - n
                        DI <- DI - n
```

CMPS esta instrucción compara dos valores restando el byte o palabra apuntado por ES:DI, del byte o palabra apuntado por DS:SI, y activa los flags de acuerdo con el resultado de la operación. Los operandos en sí, no son alterados. Después de la comparación, SI y DI son incrementados (si el flag de dirección es cero) o decrementados (si el flag de dirección es uno), para preparar la comparación del siguiente elemento de la cadena.

Operands	Clocks byte(word)	Transfers	Bytes	Example
dest,source	22(30)	2	1	CMPS STR1,STR2
(repeat) dest,source	9 + 22(30)/rep	2/rep	1	REPE CMPS STR1,STR2

---------- Ejemplo ----------

Asumiendo las definiciones:

```
          buffer1 db      100 dup (?)
          buffer2 db      100 dup (?)
```

el sigiente ejemplo compara buffer1 con buffer2 hasta la primera diferencia.

```
          cld                     ;Escanea hacia delante
          mov     cx, 100         ;Escaneará 100 bytes (CX es usado por REPE)
          lea     si, buffer1     ;Dirección de inicio del primer buffer
          lea     di, buffer2     ;Dirección de inicio del segundo buffer
repe      cmps    buffer1,buffer2 ;...y comparación entre ellos
          jne     mismatch        ;El Flag de Cero estará activado si hay
                                  ;alguna diferencia
match:            .               ;Si estamos aquí, los buffers son iguales
                  .
mismatch:
          dec     si              ;Si estamos aquí, los buffers son diferentes
          dec     di              ;Decrementamos SI y DI para que apunten a la
                  .               ;primera diferencia
                  .
```

Sobre la salida del bucle REPE CMPS, el Flag Cero será activado si se encuentra una diferencia y, de lo contrario, borrado. Si se encuentra una diferencia, DI y SI apuntarán un byte (o palabra) más allá del que (o la que) es diferente; el decremento de los registros DI y SI (DEC DI, DEC SI) hace que estos apunten a los bytes o palaras (si decrementamos otra vez) que no casan.

Notas:
Esta instrucción es siempre codificada por el ensamblador como una instrucción CMPSB, Compare String Byte; o CMPSW, Compare String Word, dependiendo de si la cadena fuente se refiere a una cadena de bytes o de palabras. En cualquier caso, debemos cargar los registros SI y DI con los offset de las cadenas de los datos fuente y destino.

SCAS = Escanéa una Cadena:

Byte/Word	1	0	1	0	1	1	1	w

```
SCAS            Scan String (Byte or Word)          Flags: O D I T S Z A P C
                                                           *       * * * * *

SCAS destination-string

          Logic:    CMP Accumulator, (ES:DI)      ;Eatablece Flags
                    if DF = 0
                        DI <- DI + n                ;n = 1 para byte, 2 para word
                    else
                        DI <- DI - n
```

SCAS ésta instrucción compara el acumulador (AL o AX) con el byte o palabra apuntado por ES:DI y activa los flags de acuerdo con el resultado de la comparación; los operandos en sí, no son alterados. Después de la comparación, DI es incrementado (si el flag de dirección es cero) o decrementado (si el flag de dirección es uno), para preparar la comparación del siguiente elemento de la cadena.

Operands	Clocks byte(word)	Transfers	Bytes	Example
dest-str	15(19)	1	1	SCAS WORD_TABLE
(repeat) dest-str	9 + 15(19)/rep	1/rep	1	REPNE SCAS BYTE_TABLE

...sigue

continua...

```
------------------------------- Example --------------------------------

  Asumiendo la definición:

        LOST_A   DB   100  DUP(?)

  el siguiente ejemplo busca en los 100 primeros bytes del bloque a
  escanear (LOST_A) el caracere ASCII 'A' (65 en decimal).


        MOV     AX, DS
        MOV     ES, AX        ;SCAS utiliza ES:DI, por eso copia DS en ES
        CLD                   ;Escanéa hacia delante
        MOV     AL, 'A'       ;Busca el catacter 'A'
        MOV     CX,100        ;Escanea los 100 bytes (CX usado por REPNE)
        LEA     DI, LOST_A    ;Dirección de comienzo en DI.
REPNE   SCAS    LOST_A        ;   ...y realiza la búsqueda.
        JE      FOUND         ;El Flag Cero será uno si hay alguna
                              ; coincidencia.
NOTFOUND:       .             ;Si estamos aquí, no hay ninguna 'A'
                .
                .
FOUND:  DEC     DI            ;Vuelta atrás para apuntar la 'A'
                .             ;   encontrada 'A'
                .

  Sobre la salida del bucle REPNE CMPS, el Flag Cero será activado si se
  encuentra una coincidencia y, de lo contrario, borrado. Si se encuentra una
  coincidencia, DI apuntarán un byte (o palabra) más allá del que (o la
  que) es diferente; el decremento del registro DI (DEC DI) hace que éste
  apunte al byte o palara (habrá que hacer otro DEC DI) que casa.
```

Notas:
Esta instrucción es siempre codificada por el ensamblador como una instrucción SCASB, Escanéa String Byte; o SCASW, Escanéa String Word, dependiendo de si la cadena fuente se refiere a una cadena de bytes o de palabras. En cualquier caso, debemos cargar los registros SI y DI con los offset de las cadenas de los datos fuente y destino.
SCAS se suele utilizar para buscar en un bloque un determinado byte o palabra. Para comparar dos cadenas o bloques de memoria elemento a elemento, es mejor utilizar la función CMPS (Comparar Cadenas).

LODS = Carga Byte/Word en:

AL/AX	1	0	1	0	1	1	0	w		

```
LODS            Load String (Byte or Word)          Flags: not altered

LODS source-str

        Logic:    Accumulator <- (DS:SI)
                  if DF = 0
                      SI <- SI + n         ; n = 1 para byte, 2 para word
                  else
                      SI <- SI - n

  LODS transfiere el valor (byte o word) apuntado por DS:DI a AL/AX. También
  incrementa o decrementa SI (dependiendo del estado del flag de dirección)
  para apuntar al siguiente elemento de la cadena.

-------------------------------------------------------------------------
 Operands                 Clocks   Transfers  Bytes   Example
                        byte(word)
 source-str               12(16)       -        1     LODS LIST
 (repeat) source-str    9+13(17)/rep  1/rep     1     REP LODS LIST
-------------------------------------------------------------------------

------------------------------- Example --------------------------------

  El siguiente ejemplo envía los ocho bytes de INIT_PORT a el puerto 250.

INIT_PORT:
        DB      '$CMD0000'      ;La cadena que queremos enviar.
                .
                .
        CLD                     ;Mover hacia delante a lo largo de INIT_PORT
        LEA     SI,INIT_PORT    ;SI coge el puntero de inicio de la cadena
        MOV     CX,8            ;CX cuenta el número de bucles
AGAIN:  LODS    INIT_PORT       ;"INIT_PORT" es necesario sólo para el
        OUT     250,AL          ;  ensamblador, para determinar word o byte.
        LOOP    AGAIN
```

Notas:
Esta instrucción es siempre codificada por el ensamblador como una instrucción LODSB, Carga String Byte; o LODSW, Carga String Word, dependiendo de si la cadena fuente se refiere a una cadena de bytes o de palabras. En cualquier caso, debemos cargar el registro SI con el offset de la cadena de los datos fuente.
Aunque es legal repetir esta instrucción, no se hace casi nunca, ya que lo que estamos haciendo es sobreescribir todo el rato el valor del acumulador.

STOS = Almacena Byte/Word en:

AL/AX	1	0	1	0	1	0	1	w		

```
STOS            Store String (Byte or Word)         Flags: not altered

STOS destination-string

        Logic:    (ES:DI) <- Accumulator
                  if DF = 0
                      DI <- DI + n         ; n = 1 para byte, 2 para word
                  else
                      DI <- DI - n
```


...sigue

continua...

STOS copia el valor (byte o word) de AL/AX a la posición apuntada por ES:DI a AL/AX. Acto seguido se incrementa o decrementa DI (dependiendo del estado del flag de dirección) para apuntar el siguiente elemento dónde se va a salvar el acumulador.

Operands	Clocks byte(word)	Transfers	Bytes	Example
dest-string	11(15)	1	1	STOS WORD_ARRAY
(repeat) dest-string	9 + 10(14)/rep	1/rep	1	REP STOS BYTE_ARRAY

---------- Example ----------

Utilizando esta instrucción con el prefijo REP, resulta útil para inicializar un bloque de memoria. Por ejemplo, el siguiente código podría inicializar a 0 los 100 primeros bytes del bloque de memoria BUFFER:

```
MOV     AL,0            ;El valor de inicialización de BUFFER
LEA     DI,BUFFER       ;Dirección de inicio del buffer
MOV     CX,100          ;Longitud del buffer
CLD                     ;Movámonos hacia delante.
REP     STOS BUFFER     ;Inicializa el buffer.
```

Notas:
Esta instrucción es siempre codificada por el ensamblador como una instrucción STOSB, STOre String Byte; o STOSW, STOre String Word, dependiendo de si la cadena fuente se refiere a una cadena de bytes o de palabras. En cualquier caso, debemos cargar el registro DI con el offset de la cadena de los datos destino.

Tab 5.- Instrucciones con Cadenas de Datos.

A3.6 TABLA DE INSTRUCCIONES DE CONTROL DE FLUJO

INSTRUCCIONES DE CONTROL DE TRANSFERENCIA

Mnemónico	Código De La Instrucción			
CALL = Llamada:	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
Directo Dentro del Segmento	1 1 1 0 1 0 0 0	disp L	disp H	
Indirecto Dentro del Segmento	1 1 1 1 1 1 1 1	mod 0 1 0 r/m		
Directo entre Segmentos	1 0 0 1 1 0 1 0	offset-L	offset-H	
		seg-L	seg-H	
Indirecto entre Segmentos	1 1 1 1 1 1 1 1	mod 0 1 1 r/m		

```
CALL            Call Procedure                    Flags: not altered

CALL procedure_name

          Logic:    if FAR CALL (inter-segment)
                        PUSH CS
                        CS <- dest_seg
                    PUSH IP
                    IP <- dest_offset

     CALL transfiere el control a un procedimiento que puede estar dentro
     del segmento actual (un procedimiento cercano, NEAR) o fuera de él (
     un procedimiento lejano, FAR). Los dos tipos de llamadas tienen distinto
     código máquina, y la instrucción de retorno RET que hace que podamos
     volver del procedimiento tiene que ser igual que el tipo de llamada CALL.

  ---------------------------------------------------------------------------
   Operands                  Clocks   Transfers  Bytes   Example
                           byte(word)
   near-proc                 19(23)       1        3     CALL NEAR_PROC
   far-proc                  28(36)       2        5     CALL FAR_PROC
   memptr 16               21(29) + EA    2       2-4    CALL PROC_TABLE[SI]
   regptr 16                 16(24)       1        2     CALL AX
   memptr 32               37(57) + EA    4       2-4    CALL [BX].ROUTINE
  ---------------------------------------------------------------------------
```

Notas:
Para un perocedimiento entre segmentos (procedimiento en un segmento distinto), el procesador primero pone el valor actual del CS en la pila, después el de IP (que apunta a la instrucción que sigue a CALL), y finalmente concede el control al procedimiento.
Para un salto dentro del segmento actual (intra-segment), el procesador primero pone el valor actual de IP (que está apuntando a la siguiente instrucción) en la pila, y transfiere el control al procedimiento.
CALL pueden también leer la dirección del procedimiento desde un registro o una localización de memoria. Esta forma de CALL es una llamada indirecta.

JMP = Salto Incondicional:				
Directo Dentro del Segmento (Largo)	1 1 1 0 1 0 0 1	disp L	disp H	
Directo Dentro del Segmento (Corto)	1 1 1 0 1 0 1 1	disp		
Indirecto Dentro del Segmento	1 1 1 1 1 1 1 1	mod 1 0 0 r/m		
Directo Entre Segmentos	1 1 1 0 1 0 1 0	offset-L	offset-H	
		seg-L	seg-H	
Indirecto Entre Segmentos	1 1 1 1 1 1 1 1	mod 1 0 1 r/m		

```
JMP             Jump Unconditionally              Flags: not altered

JMP target

          Jump Condition:  Jump always

     JMP transfiere siempre el control a la posición apuntada. A diferencia
     de CALL, JMP no salva IP, ya que no se espera un RETorno. An salto JMP en el
     segmento puede hecerse o a través de la memoria o un registro de 16 bits;
     un JMP etre segmentos, puede hacerse sólo a través de memoria.

  ---------------------------------------------------------------------------
   Operands                  Clocks   Transfers  Bytes   Example
   short-label                 15         -        2     JMP ROPE_NEAR
   near-label                  15         -        3     JMP SAME_SEGMENT
   far-label                   15         -        5     JMP FAR_LABEL
   memptr16                18 + EA       -       2-4    JMP SAME_SEG
   regptr16                    11         -        2     JMP BX
   memptr32                24 + EA       -       2-4    JMP NEXT_SEG
  ---------------------------------------------------------------------------
```

Notas:
Si el ensamblador puede determinar que el objetivo del salto está dentro del segmento y que a su vez está dentro de los +- 127 bytes de la posición actual, el ensamblador generará automaticamente una instrucción de salto corto (SHORT que es el que menos ocupa, 2 bytes); de otra forma se generaría instrucción de calto cercano (NEAR JMP, 3 bytes).
Podemos forzar la generación de un SHORT JMP utilizando explícitamente el operador "short": JMP short near_by

RET = Retorno de una Llamada:				
Dentro del Segmento	1 1 0 0 0 0 1 1			
Dentro del Seg. Sumando a SP imm-data	1 1 0 0 0 0 1 0	data L	data H	
Entre Segmentos	1 1 0 0 1 0 1 1			
Entre Segs. Sumando a SP imm-data	1 1 0 0 1 0 1 0	data L	data H	

```
RET             Return from Procedure             Flags: not altered

RET optional-pop-value

          Logic:    POP IP
                    If FAR RETURN (inter-segment)
                        POP CS
                    SP <- SP + optional-pop-value (si se especifica)
```


continua...

...sigue

RET devuelve el control desde un procedimiento a la instrucción siguiente a la de llamada (CALL), haciendo:

1 Recupera el IP de la pila.
2 Si el RETorno es a un procedimiento dentro del segmento: Recuperar también CS de la pila
3 Sumar un valor opcional a SP, si se especifica.

El ensamblador generará un RET dentro del segmento si el procedimiento que contiene RET, estaba designado por el programador como NEAR, y un RET entre segmentos, si estaba designado como FAR. El valor opcional (dato inmediato) especifica un valor a sumar a SP depués de recuperar IP y, en en su caso CS; el efecto de esta suma el sacar (popping) un número específico de bytes de la cabecera de la pila.

Operands	Clocks	Transfers	Bytes	Example
(intrasegment, no pop)	20	1	1	RET
(intrasegment, with pop)	24	1	3	RET 4
(intersegment, no pop)	32	2	1	RET
(intersegment, pop)	31	2	3	RET 2

INSTRUCCIONES DE SALTO CONDICIONAL

JXX = Salto Condicional:	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Dentro del Segmento, Salto Corto.	0	1	1	1	c	c	c	c	disp																							

Mnemónico: (JXX)	Código de Condicción (cccc)	Condición de salto (Lógica)	Descripción
JO	0000	OF=1	Saltar si existe desbordamiento.
JNO	0001	OF=0	Saltar si no existe desbordamiento.
JB/JNAE	0010	CF=1	Saltar si está abajo.
JAE/JNB	0011	CF=0	Saltar si está arriba o igual.
JE/JZ	0100	ZF=1	Saltar si es igual/cero.
JNE/JNZ	0101	ZF=0	Saltar si no es igual/no es cero.
JBE/JNA	0110	CF = 1 OR ZF = 1	Saltar si está a bajo o igual.
JA/JNBE	0111	CF=0 AND ZF=0	Saltar si está arriba.
JS	1000	S=1	Saltar si existe signo.
JNS	1001	S=0	Saltar si no existe signo.
JP/JPE	1010	P=1	Saltar si la paridad es par.
JNP/JPO	1011	P=0	Saltar si la paridad es nones.
JL/JNGE	1100	SF < > OF	Saltar si es menor que.
JGE/JNL	1101	SF=OF	Saltar si es mayor o igual.
JLE/JNG	1110	(SF < > OF) OR ZF = 1	Saltar si es menor o igual.
JG/JNLE	1111	ZF=0 AND (SF=OF)	Saltar si en mayor

JXX Jump Conditional Flags: not altered

JXX short-label

JXX transfire el control a una "etiqueta corta" (el objetivo debe estar dentro de los –128 a +127 bytes de la siguiente instrucción) si se da la condición de salto. Estas instrucciones suelen ser utilizadas después de las instrucciones CMP o SUB; los operandos de CMP o SUB son tratados como números con signo.

Operands	Clocks	Transfers	Bytes	Example
short-label	16 or 4	-	2	JO SIGNED_OVERFLOW
short-label	16 or 4	-	2	JE ZERO
short-label	16 or 4	-	2	JG GREATER

LOOPING INSTRUCTIONS

LOOPING = Bucles:	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Dentro del Segmento, Salto Corto.	1	1	1	0	0	0	L	L	disp																							

Mnemónico	LL	Lógica	Descripción
LOOPNZ/ LOOPNE	00	Salta a la etiqueta si el contador (CX) es distinto de cero y ZF=0 CX <- CX – 1 If CX <> 0 and ZF = 0 JMP short-label	LOOPNE Loop While not Equal Flags: not altered LOOPNE short-label Utilizado después de SUB o CMP, LOOPNE decrementa CX en 1 y transfiere el control a la etiqueta si el primer operando de CMP o SUB no es igual al segundo operando. Operands / Clocks / Transfers / Bytes / Example short-label / 19 or 5 / - / 2 / LOOPNE AGAIN ***Nota:*** *LOOPNZ, Loop Mientras no sea Cero, es la misma instrucción*
LOOPZ/ LOOPE	01	Salta a la etiqueta si el contador (CX) es distinto de cero y ZF=1. CX <- CX - 1 If CX <> 0 and ZF = 1 JMP short-label	LOOPE Loop While Equal Flags: not altered LOOPE short-label Utilizado después de SUB o CMP, LOOPE decrementa CX en 1 y transfiere el control a la etiqueta si el primer operando de CMP o SUB es igual al segundo operando. Operands / Clocks / Transfers / Bytes / Example short-label / 18 or 6 / - / 2 / LOOPE AGAIN ***Nota:*** *LOOPZ, Loop si es Cero, es la misma instrucción*
LOOP	10	Salta a la etiqueta si el contador (CX) es distinto de cero. CX <- CX - 1 If (CX <> 0) JMP short-label	LOOP Loop on Count Flags: not altered LOOP short-label Decrementa en uno CX y transfiere el control a la etiqueta si CX no es cero Operands / Clocks / Transfers / Bytes / Example short-label / 17/5 / - / 2 / LOOP AGAIN

continua...

...sigue

JCXZ	11	Salta si el contador es cero. Jump if CX = 0	JCXZ Jump If CX Register Zero Flags: not altered JCXZ short-label JCXZ transfiere el control a una etiqueta corta si el registro CX es 0. Operands / Clocks / Transfers / Bytes / Example short-label / 18 or 6 / - / 2 / JCXZ COUNT_DONE ***Nota**: Esta instrucción es nortmalmente utilizada al comienzo de un bucle para no ejecutarlo en caso de que la variable del contador (CX) sea cero.*

***Nota**: El objetivo de una etiqueta cota (short-label), debe estar dentro de los –128 a +127 bytes de la siguiente instrucción*

INTERRUPTION INSTRUCTIONS

INTERRUPT = Interrupciones:	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Dentro del Segmento, Salto Corto.	1	1	0	0	1	1	i	i	tipo (si ii=01)																							

Mnemónico	ii	Descripción
INT	00 (Tipo 3) 01 (específico)	INT Interrupt Flags: O D I T S Z A P C 0 0 INT interrupt-num Logic: PUSHF ; Salva los flaga en la pila TF <- 0 ; Borra el flag de Trampa IF <- 0 ; Deshabilita interrupciones CALL FAR (INT*4) ; Llama al procedimiento de inter. INT salva los flags en la pila, borra los Flags de Trampa y Habilitación de Interrupción, salva CS e IP y transfiere el control al manejador de interrupciones especificado en el "interrupt-num". Si el manejador de la interupción retorna utilizando la instrucción IRET, los flags originales son reestablecidos desde la pila. Operands / Clocks byte(word) / Transfers / Bytes / Example immed8 (type = 3) / 52 / 5 / 1 / INT 3 immed8 (type <> 3) / 51 / 5 / 2 / INT 21 ***Notas**: Los flags son reestablecidos de la misma manera que lo hace la intrucción PUSHF.* *La dirección del vector de interrupción es calculada multiplicando el "interrupt-num" por 4. La primera palabra del resultado es cargada en IP, y la segunda en CS.* *Todas los "interrupt-nums" excepto el tipo 3 generan un opcode de dos bytes; la interrupción tipo 3, genera un opcode de un byte llamado Brakpoint (punto de ruptura).*
INTO	10	INTO Interrupt on Overflow Flags: O D I T S Z A P C 0 0 INTO Logic: if (OF = 1) PUSHF ; Salva flags en la pila TF <- 0 ; Borra la bandera de tampa IF <- 0 ; Deshabilita las interrupciones CALL FAR (10h) ; El vector de INTO es 0000:0010h INTO activa la interrupción tipo 4 si el Flag de desbordamiento OF está activado; si no lo está, no hace nada. Esta interrupcióm actúa de la misma manera que "INT 4" si el Flag de Overflow está activado; en cuyo caso INTO salva los flags en la pila, borra los Flags de habilitación de Interrupción (IF) y Trampa (TF), salva CS e IP y finalmente transfiere el control a el manejador de la interrupción 4 que es apuntado por el vector de la posición de memoria 10h. Si el manejador de la interrupción retorna utilizando la instrucción IRET, los flags originales son actualizados. Operands / Clocks byte(word) / Transfers / Bytes / Example no operands / 53 or 4 / 5 / 1 / INTO ***Notas**: Los flags son almacenados de la misma manera que en la instrucción PUSHF.* *INTO puede ser usada después de una operación que puede causar un desbordamiento, para llamar a un procedimiento de recuperación.*
IRET	11	IRET Interrupt Return Flags: O D I T S Z A P C r r r r r r r r IRET Logic: POP IP POP CS POPF ; Saca los flags de la pila IRET devuelve el control de una rutina de interupcción a el punto dónde ocurrió la interrupción recuperndo de la pila los registros IP, CS y las banderas. Operands / Clocks / Transfers / Bytes / Example no operands / 32 / 3 / 1 / IRET

Tab 6.- Instrucciones de Control de Transferencia.

A3.7 TABLA DE INSTRUCCIONES DE CONTROL DEL MICROPROCESADOR

INSTRUCCIONES DE CONTROL DEL PROCESADOR

SetFlags Instructions

Mnemonico	Código De La Instrucción																														
SetFlags = Establecer Flags:	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Registro de Banderas	1	1	1	1	1	f	f	t																								

Mnemónico	Código de flag (ff)	New Value (t)	Descripción	Flags: O D I T S Z A P C
CLC	00 (Flag de Carry)	0 Borra	Borra el flag de Carry.	0
STC		1 Activa	Activa el flag de Carry, no se modifican otros flags	1
CLI	01 (Flag de Inter.)	0 Borra	Borra el Flag de Habilitación de Interrupción.	0
STI		1 Activa	Activa el Flag de Habilitación de Interrupción.	1
CLD	10 (Flag de Dirección)	0 Borra	Borra el Flag de Dirección.	0
STD		1 Activa	Activa el Flag de Dirección.	1

Operands	Clocks	Transfers	Bytes	Example
no operands	2	-	1	CLC, STI, CLD

HLT = Parar:

	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Procesador	1	1	1	1	0	1	0	0																								

HLT Halt the Processor Flags: not altered

HLT

HLT esta instrucción, para la CPU. El procesador se mantiene en espera de una interrupción no enmascarable, una interrupción mascarable con la habilitación permitida o la activación de la señal de RESET.

Operands	Clocks	Transfers	Bytes	Example
no operands	2	-	1	HLT

WAIT = Espera:

Procesador	1	0	0	1	1	0	1	1																								

WAIT Wait Flags: not altered

WAIT hace que el procesador entre en el estado de espera. El procesador se mantendrá inactivo hasta que la entrada de TEST del microprocesador esté activada.

Operands	Clocks	Transfers	Bytes	Example
no operands	3 + 5n	-	1	WAIT

***Notas**: Esta instrucción es utilizada para la sincronización con hardware externo, como un coprocesador.*

ESC = Salir de un dispositivo externo:

Procesador	1	1	0	1	1	x	x	x	mod		xxx		r/m																			

ESC Escape Flags: not altered

ESC opcode del coprocesador,fuente

ESC es utilizado para pasar el control del microprocesador al un coprocesador como el 8087 o el 80287. En respuesta a ESC, el microprocesador accede a un operando en memoria (la instrucción para el coprocesador) y la pone en el bus. El coprocesador está a la espera de los comandos ESC y ejecuta la instrucción posicionada en el bus, utilizando la dirección efectiva (EA) como fuente. XXXXXX es el código de operación para el coprocesador.

Operands	Clocks	Transfers	Bytes	Example
	byte(word)			
immediate, memory	8(12) + EA	1	2-4	ESC 6,ADR[SI]
immediate, register	2	-	2	ESC COPROC-CODE,AH

***Notas**: para sincronizarse con el coprocesador, el programador debe poner explícitamente la instrucción WAIT después de todas las instrucciones ESC. Los microprocesadores 80286 y 80386 tienen una sincronización de intrucción automática, por lo que no es necesario poner instrucciones WAIT.*

LOCK (prefijo)= Bloqueo del Bus:

Procesador	1	1	1	1	0	0	0	0																								

LOCK Lock the Bus Flags: not altered

LOCK

LOCK es un prefijo de un byte que puede precede a cualquier instrucción. LOCK provoca que el procesador active su señal externa de lock mientras la instrucción que sigue al prefijo se está ejecutando. Si el sistema está diseñado teniendo en cuenta la señal lock, cualquier tipo de acceso al bus por parte de un dispositivo externo será ignorado (si lock está activada), incluso interrupciones y transferencias DMA.

Operands	Clocks	Transfers	Bytes	Example
no operands	2	-	1	LOCK XCHG FLAG,AL

***Nota**: Esta instrucción está concevida para el soporte de sistemas multiprocesador con actividades en segundo plano. En cualquier sistema, el acceso a esas actividades es generalmente controlado por un sistema de control, combinación de software-hardware (semáforos o árbitros).*
Esta instrucción debería ser sólo usada para prevenir que otros maestros de bus interrumpan una operación de movimiento de datos. Este prefijo debería ser utilizado solamente con las instrucciones XCHG, MOV y MOVS.

Tab 7.- Instrucciones de Control del Microprocesador.

APÉNDICE B

programa DEBUG

B DEBUG

Voy a hacer una breve descripción de éste programa, enseñando lo esencial sin entrar en profundidad en todas sus posibilidades.

> NOTA: Para más información acerca del programa DEBUG, se puede consultar la pág 97 de Triebel "The DEBUG, a software program for PC" o incluso la propia ayuda de Windows.

DEBUG es un programa utilizado para depurar código que viene como utilidad en todas las versiones de sistema operativo MS-DOS ®, y por compatibilidad en todas las superiores, Windows 98, Windows XP. Lo he utilizado cuando me encontraba con el problema de no saber exactamente cómo funciona una instrucción, los flags que modifica, como quedan afectados los registros de uso general tras su ejecución, etc. Ante este problema y como la gran mayoría de los usuarios tenemos un PC con un juego de instrucciones compatible con el microprocesador 8088/8086 (son los padres de los actuales micros de Intel ®, por lo tanto los actuales micros mantienen su compatibilidad con sus antecesores), podemos utilizar nuestro ordenador para que, con DEBUG, poder ejecutar la instrucción y despejar nuestras dudas acerca de su funcionalmiento.

El programa arranca escribiendo DEBUG en el promp del DOS:

```
C:\>DEGUG.EXE
```

Una vez inicializado, podemos teclear la interrogación y ENTER para ver la lista de los comandos que se pueden utilizar con el programa. En la tabla 1 se muestra el nombre de cada comando en castellano y entre paréntesis en el idioma anglosajón con el fin de cotejarlo con el acrónimo de la sintaxis. También podemos ver la función de cada comando.

Con la carga de DEBUG, el estado del microprocesador es inicializado, los registros AX, BX, CX, DX, BP, SI y DI son puestos a cero, el puntero de instrucciones IP=0100h; el sistema operativo detecta una posición de memoria libre y la carga en los registros de segmento CS, DS, SS y ES; el puntero de pila SP es inicializado al valor FFEEh. Por último, todos los flags excepto IF son borrados. Con el comando R (Register) podemos ver o modificar el valor de los registros internos. La notación de las banderas es un poco particular como podemos ver en la tabla siguiente:

Comando	Sintaxis	Funcion
Registro (Register)	R [Nombre del Registro]	Examina o modifica el contenido de un registro interno
Salir (Quit)	Q	Sale del programa DEBUG
Volcar (Dump)	D [Dirección]	Vuelca el contenido de la memoria a la pantalla.
Entrar (Enter)	E Dirección [Lista]	Examina o modifica el contenido de memoria.
Rellenar (Fill)	F Dirección_Inicial Dirección_Final Lista	Rellena un bloque de memoria con los datos de la lista
Mover (Move)	M Dirección_Inicial Dirección_Final Dirección_Destino	Mueve un bloque de datos desde una localización fuente de memoria a una destino.
Comparar (Compare)	C Dirección_Inicial Dirección_Final Dirección_Destino	Compara dos bloques de datos en memoria y muestrea las posiciones que contienen alguna diferencia
Buscar (Search)	S Dirección_Inicial Dirección_Final Lista	Busca a través de un bloque de datos en memoria y muestra todas las posiciones de memoria que coinciden con los datos en la lista.
Entrada (Input)	I Address	Lee un puerto de entrada
Salida (Output)	O Address, Byte	Escribe el Byte en el puerto de salida
Suma/Resta Hexadecimal	H Número1, Número2	Genera sumas o resas en hexadecimal de los dos números indicados.
Desensamblar (Unanssemble)	U [Dirección_Inicial [Dirección_Final]]	Desensambla el código máquina en su s instrucciones ensabladas equivalentes, es decir, convierte el intervalo indicado de datos en hexadecimal en sus correspondientes mnemónicos en ensamblador.
Nombre (Name)	N Fichero	Asigna el fichero a los datos que van a ser escritos en el disco.
Escribir (Write)	W [Dirección _Inicial [Disco Sector_Inicial Número_de_Sectores]]	Escribe el contenido de memoria en un disco o archivo
Cargar (Load)	L [Dirección _Inicial [Disco Sector_Inicial Número_de_Sectores]]	Carga en memoria el contenido de un disco o archivo.
Ensamblar (Assemble)	A [Direccion_Inicial]	Ensambla la instrucción (convierte a código máquina) y almacena el código en memoria.
Trazado (Trace)	T [=Dirección][Número]	Ejecuta paso a paso el número indicado de instrucciones.
Ejecutar (Go)	G [=Dirección_Inicial[Dirección_de_Parada...]]	Ejecuta las instrucciones hasta que llega a una dirección de parada.

Tab 1.- Tabla de Comandos de Debug.

Bandera	Significado	Estado Activo	Estado Inactivo
OF	Desbordamiento (Overfow)	OV	NV
DF	Dirección (Direction)	DN	UP
IF	Interrupción (Interrupt)	EI	DI
SF	Signo (Sign)	NG	PL
ZF	Cero (Zero)	ZR	NZ
AF	Acarreo Auxiliar (Auxiliar Carry)	AC	NA
PF	Paridad (Parity)	PE	PO
CF	Acarreo (Carry)	CY	NC

Tab 2.- Notación de las Banderas en Debug.

La notación que utiliza DEBUG para los registros internos se muestra a continuación:

Símbolo	Registro
AX	Acumulador
BX	Base
CX	Contador
DX	Datos
SI	Índice Fuente
DI	Índice Destino
SP	Puntero de Pila
BP	Puntero Base
CS	Segmento de Código
DS	Segmento de Datos
SS	Segmento de Pila
ES	Segmento de Datos Extra
F	Registro de Banderas
IP	Puntero de Instrucciones

Tab 3.- Notación de los Registros en Debug.

B1 EJEMPLO 1:

Como ejemplo, digamos que queremos saber el código OP de la instrucción MOV AH,CL (Mover registro de 8 bits a otro registro interno de 8 bits), hay dos soluciones:

1. Utilizar MaroASseMbler (MASM), compilando un programa con dicha instrucción. Más adelante se explica la utilización de este programa para la generación de los programas de ejemplo [insertar referencia a la explicación del MASM]
2. Utilizar el programa genérico de los sistemas operativos de Microsoft, DEBUG. Como se muestra a continuación.

Antes de nada, comentar que éste es un ejemplo real que utilizé para despejar una duda en esta instrucción. Si vemos el formato de este OP (OPeration Code):

OPERACIONES DE TRANSFERENCIA DE DATOS																
Mnemonico	Código de la instrucción															
MOV = Mover:	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Registro/Memoria hacia/desde Registro	1	0	0	0	1	0	d	w	mod		reg			r/m		
Posibles OP de la Instrucción MOV AH,CL																
MOV AH,CL:	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Registro/Memoria hacia/desde Registro	1	0	0	0	1	0	d	w	mod		reg			r/m		
Opción (1): OP= 8Ah E1h	1	0	0	0	1	0	1	0	11		100			001		
Opción (2): OP= 88h CCh	1	0	0	0	1	0	0	0	11		001			100		

Tab 4.- Formato y Posibles Soluciones de la Instrucción MOV AH,CL.

Si tenemos que codificar a mano la instrucción MOV AH,CL [Ver Instruction Set y formato de los código máquina], podemos hacerlo de dos maneras:

No hay duda de que: w='0', transferncia de byte y mod="11", indicando que r/m será tratado como reg (transferencia de registro a registro); ahora viene la duda:

(1) ¿d='1'? lo que significaría que reg="100" (AH) y r/m="001" (CL)

(2) ¿d='0'? y reg="001" (CL) y r/m="100" (AX).

Si Ejecutamos DEBUG y ensamblamos la instrucción con el comando A, veremos el resultado.

C:\> DEBUG.EXE (Enter)	Entramos en el programa
-A (Enter)	Ejecutamos el comando A para Ensamblar la Instrucción
CS:0100 MOV AH,CL (Enter)	Introducimos la instrucción
CS:0102 (Enter)	La Instrucción ha sido Ensamblada y ocupa 2 Bytes.
-U (Enter)	Desensamblamos la memoria con el comando U.
CS:100 88CC MOV AH,CL	Podemos ver que lo ha codificado como 88CCh, Opción (2)
-Q (Enter)	Salimos del Progrma con el comando Quit.
C:\>	Promp del sistema operativo.

Queda claro que el programa DEBUG ensambla la instrucción con d='0', pero la opción (1) también es factible por lo que implementé ambas formas.

B2 EJEMPLO 2

Al Implementar la Instrucción NOP (No OPeration), me di cuenta que no existe como tal en el juego de instrucciones de los microprocesadores i8088/i8086 [insertar refa a Datasheet 8088/8086 pág 26], sin embargo, muchos programas la utilizan para "perder" tiempo.

Introduciéndola en DEBUG, podemos ver el código máquina:

Analizando el resultado y cotejándolo con código máquina del juego de instrucciones, vemos que en realidad se trata de la instrcción XCHG AX,AX

C:\> DEBUG.EXE (Enter)	Entramos en el programa
-A (Enter)	Ejecutamos el comando A para Ensamblar la Instrucción
CS:0100 NOP (Enter)	Introducimos la instrucción
CS:0101 (Enter)	La Instrucción ha sido Ensamblada y ocupa 1 Byte.
-U (Enter)	Desensamblamos la memoria con el comando U.
CS:100 90 NOP	Podemos ver que lo ha codificado como 90h.
-Q (Enter)	Salimos del Progrma con el comando Quit.
C:\>	Promp del sistema operativo.

XCHG AX,AX; reg="000" (AX)						
Mnemonico	Código de la instrucción					
Registro con Acumulador	0	1	0	1	1	reg
Código Máquina= 90h	0	1	0	1	1	000

Tab 5.- Formato y Posibles Soluciones de la Instrucción XCHG AX,AX.

Hay ejemplos mucho más complejos en la literatura. [ver más ejemplos en capitulo 4 trebel]

cuestiones generales

C-. PÁGINA WEB DEL PROYECTO: http://perso.wanadoo.es/odradek

La página Web del proyecto, fue creada con la intención de hacer un seguimiento contínuo del proyecto a través de Internet. La revisión que actualmente se encuentra "colgada" en internet es la 5ª y su código está optimizado para que se cargue rápidamente. Pese a ello, las características de la página (muchos archivos que ocupan poco espacio), hacen necesario un buen ancho de banda. La página está programada en Java y Optimizada para Internet Explorer v5 o superior (si utilizamos otro browser puede que no se visualicen correctamente algunas secciones de la página), incluye algunas animaciones "flash" que podemos desactivar desde la pantalla de presentación activando la casilla "HTML versión", ver la figura 1. Puede resultar útil para aquellos visitantes con una conexión lenta. Por alguna esxtraña razón, el contador de visitas ha fallado, pero las últimas noticias indicaban más de 2000 visitas lo que no está mal si tenemos en cuenta que no he dado ningún tipo de publicidad ni he realizado ninguna acción para darla a conocer.

La página está dividida en varias secciones, que aparecen en forma de libors en el browser (estilo explorador de Windows) utilizado para movernos por todo el contenido de la página Web. En la página siguiente podemos ver la pantalla principal de la página a la que accedemos después de de pulsar con el rator sobre el texto "ENTRAR" de la pantalla de presentación que podemos ver a la derecha.

C1.1 CONTRATACIÓN DE LICENCIA GNU

La página contiene las librerías completas de los modelos, por lo que el visitante puede bajar y/o modificar su contenido. Para evitar que alguien pueda pedirme responsabilidades y a la vez proteger tanto el código como la documentación (incluida esta memoria), he contratado una licencia de documentación libre GNU'. De esta manera evitamos, por ejemplo, que alguien publique este trabajo como suyo sin mencionar la fuente o que la utilización de los modelos produzca algún tipo de pérdida de la cual quieran hacerme responsable.

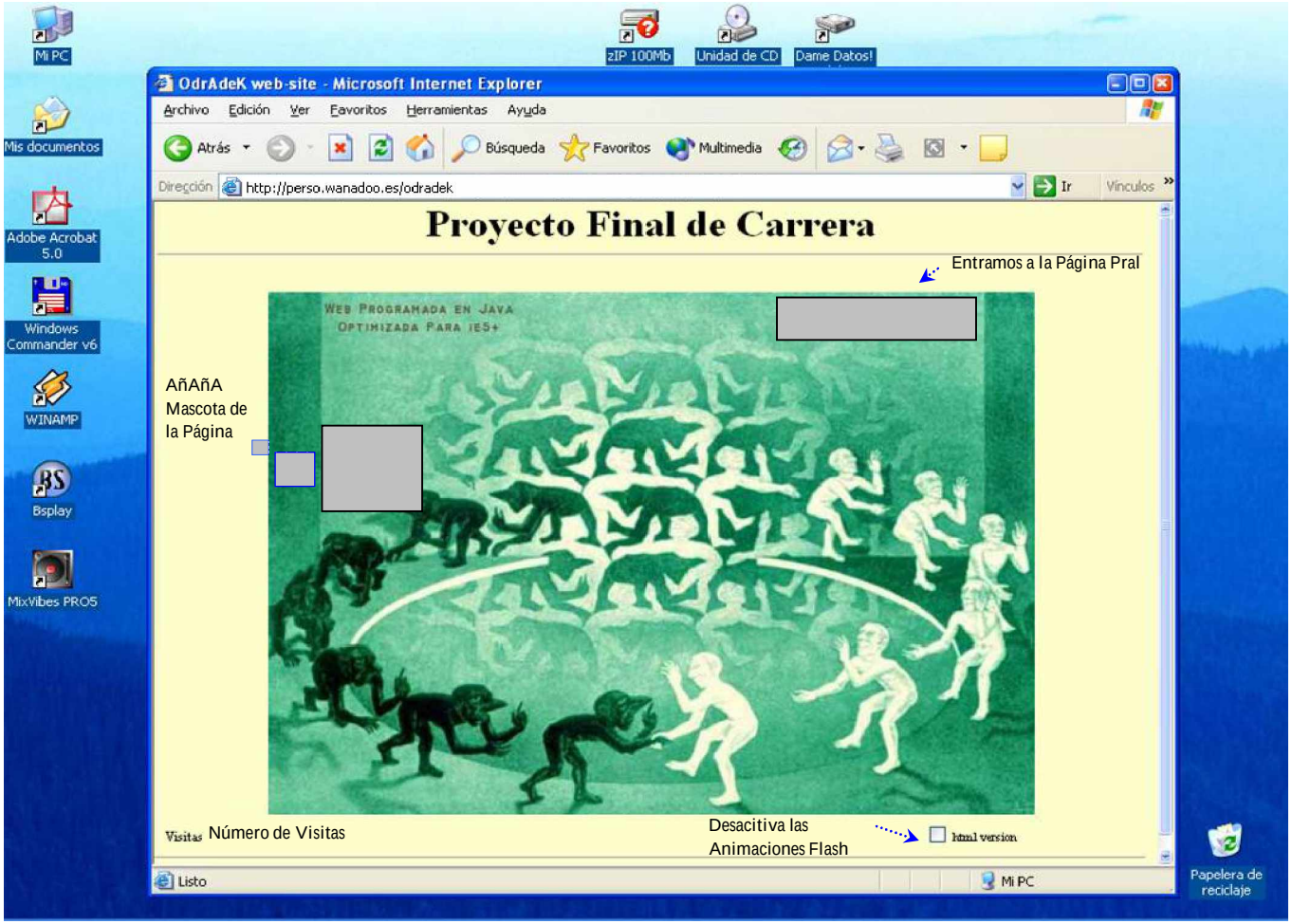


Fig 1.- Pantalla de Presentación de la Web del Proyecto

Derecho de Autor (c) 2004, Rubén Martín.

Se otorga permiso para copiar, distribuir y/o modificar este documento bajo los términos de la Licencia de Documentación Libre GNU, Versión 1.1 o cualquier otra versión posterior publicada por la Free Software Foundation y siempre y cuando no sean utilizados con fines comerciales y/o militares. Una copia de la licencia es incluida en la sección titulada "Licencia de Documentación Libre GNU", al final de este capítulo.

C1.2 SECCIONES DEL MENÚ

LIBRERIAS DE ActiveHDL. Contiene los esquemas y grafos de todos los componentes que implementan los modelos.

CÓDIGO VHDL. Aquí se encuentran todos los archivos fuente VHDL necesarios para grnerar los modelos.

FICHEROS PDF. En esta Sección podemos Ver y Descargar algunos de los Archivos PDF Utilizados para el Diseño.

FICHEROS EXTRA: Incluye algunos archivos de curiosidades y cuestiones generales

GUESTBOOK Contiene un libro de visitas y un Foro con varias temáticas abiertas donde el visitante puede participal

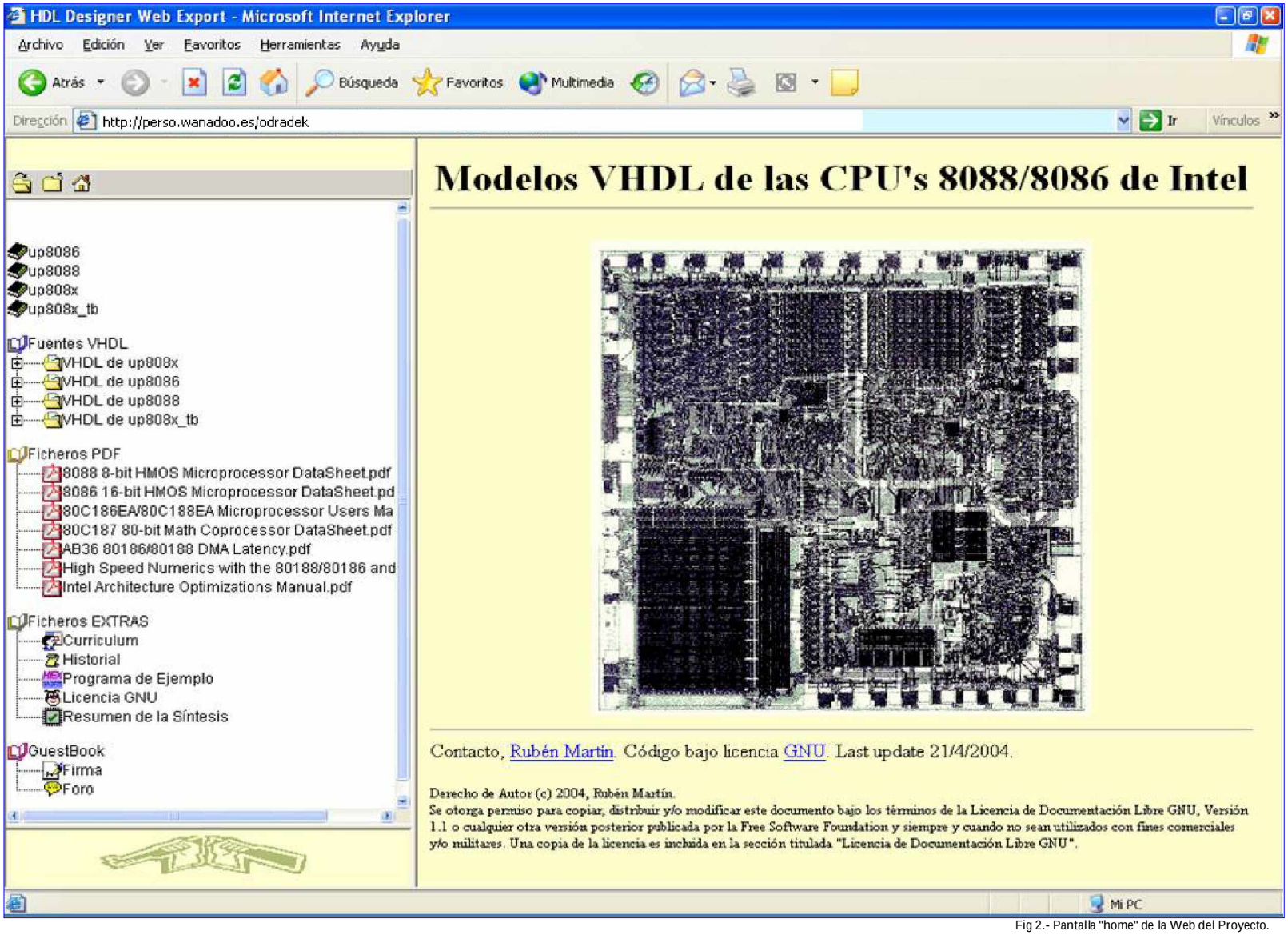


Fig 2.- Pantalla "home" de la Web del Proyecto.

La pantalla principal está dividida en dos partes claramente diferenciadas, la pantalla del browser (a la iizquierda) en la que podemos ver todo el contenido de la Web de forma gráfica., y la pantalla de presentación (a la derecha), donde veremos en contenido del archivo que seleccionemos en el browser.

En el manejador de archivos de la izquierda, aparecen tres iconos en la parte superior. El primero de ellos, permite expandir toda la jerarquía del elemento selecionado. El segundo Contrae la jerarquía hasta el archivo seleccionado, y el tercero, permite acceder en cualquier momento a la pantalla principal (home) que es la que podemos ver en la figura 2.

Todos los archivos (a excepción de los archivos fuente y los PDF) son presentados en la pantalla de la derecha con un formato especial de texto que los hace más legibles y bonitos. Tanto el Foro como el GuestBooK siguen abiertos con varias temáticas aunque los visitantes no participan mucho. En la pantalla de presentación de Home, podemos ver la última actualización de la página.

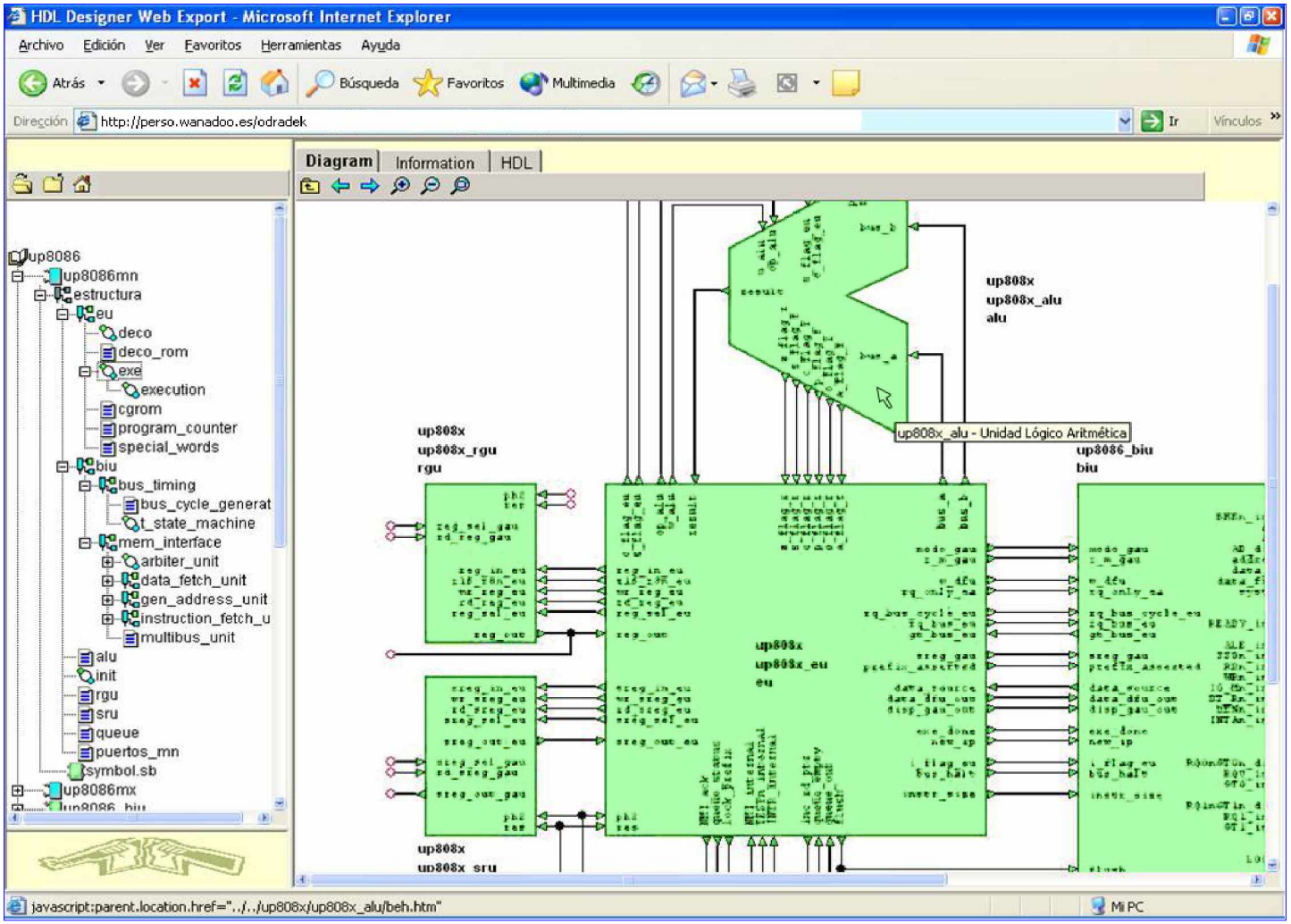


Fig 3.- La Exploración de la Jerarquía se Puede Hacer desde el Browser o la Pantalla de Presentación

C1.3 EXPLORACIÓN DE LAS LIBRERÍAS

La navegación por las librerias es, quizás, la parte más interesante de la página Web. La exploración de la jerarquía de los modelos, se realiza de una forma muy sencilla a través de la pantalla de exploración. Posicionánsonos sobre el componente podemos ver una breve descripción de la función que realiza, y pulsando con el ratón sobre él, entramos en la jerarquía inmediatamente inferior. Si se trata de un esquema, veremos el nuevo esquema en la pantalla, si se trata de un grafo de estados, lo veremo tal y como lo podemos encontrar en esta memoria, y si se trata de una unidad escrita directamente en VHDL, veremos el código VHDL formateado (con colorines) en la pantalla de exploración. Para volver al nivel padre, simplemente tenemos que pulsar sobre el icono que aparece arriba a la izquierda en la pantalla de exploración. Tanto los Grafos como los Esquemas, tienen dos pestañas más, la pestaña de Información que nos indica las señales declaradas y la Entity del componente, y la pestaña HDL que muestra el código VHDL con el que se ha codificado el Grafo o Esquema.

C2. LICENCIA DE DOCUMENTACIÓN LIBRE GNU

La versión original esta disponible en:

http://www.gnu.org/copyleft/fdl.html

Versión 1.1, Marzo de 2000

Derechos de Autor (C) 2000 Free Software Foundation, Inc. 59 Temple Place, Suite 330, Boston, MA 02111-1307 USAS se permite la copia y distribución de copias literales de este documento de licencia, pero no se permiten cambios.

0. PREÁMBULO

El propósito de esta licencia es permitir que un manual, libro de texto, u otro documento escrito sea "libre" en el sentido de libertad: asegurar a todo el mundo la libertad efectiva de copiarlo y redistribuirlo, con o sin modificaciones, de manera no comercial. En segundo término, esta licencia preserva para el autor o para quien publica una manera de obtener reconocimiento por su trabajo, al tiempo que no se considera responsable de las modificaciones realizadas por terceros, lo cual significa que los trabajos derivados del documento deben a su vez ser libres en el mismo sentido. Complementa la Licencia Pública General GNU, que es una licencia de copyleft diseñada para el software libre.

Hemos diseñado esta Licencia para usarla en manuales de software libre, ya que el software libre necesita documentación libre: Un programa libre debe venir con los manuales que ofrezcan la mismas libertades que da el software. Pero esta licencia no se limita a manuales de software; puede ser usada para cualquier trabajo textual, sin tener en cuenta su temática o si se publica como libro impreso. Recomendamos esta licencia principalmente para trabajos cuyo fin sea instructivo o de referencia.

1. APLICABILIDAD Y DEFINICIONES

Esta Licencia se aplica a cualquier manual u otro documento que contenga una nota del propietario de los derechos que indique que puede ser distribuido bajo los términos de la Licencia. El "Documento", en adelante, se refiere a culquiera de dichos manuales o trabajos. Cualquier miembro del público es un como "Usted".

Una "Versión Modificada" del Documento significa cualquier trabajo que contenga el Documento o una porción del mismo, ya sea una copia literal o con modificaciones y/o traducciones a otro idioma.

Una "Sección Secundaria" es un apéndice titulado o una sección preliminar al prólogo del Documento que tiene que ver exclusivamente con la relación de quien publica, o con los autores del Documento, o con el tema general del Documento (o asuntos relacionados) y cuyo contenido no entre directamente en tal tema general. (Por ejemplo, si el Documento es en parte un texto de matemáticas, una Sección Secundaria puede no explicar matemáticas). La relación puede ser un asunto de conexión histórica, o de posición egal, comercial, filosófica, ética o política con el tema o la materia del texto.

Las "Secciones Invariantes" son ciertas Secciones Secundarias cuyos títulos son denominados como Secciones Invariantes en la nota que indica que el documento es liberado bajo esta licencia.

Los "Textos de Cubierta" son ciertos pasajes cortos de texto que se listan, como Textos de Título o Textos al respaldo de la página de tí, en la nota que indica que el documento es liberado bajo esta Licencia.

Una copia "Transparente" del Documento, significa una copia para lectura en máquina, representada en un formato cuya especificación está disponible al público general, cuyos contenidos pueden ser vistos y editados directamente con editores de texto genéricos o (para imágenes compuestas por pixeles) con programas genéricos de dibujo o (para dibujos) con algún editor gráfico ampliamente disponible, y que sea adecuado para exportar a formateadores de texto o para traducción automática a una variedad de formatos adecuados para ingresar a formateadores de texto. Una copia hecha en un formato de un archivo que no sea Transparente, cuyo formato ha sido diseñado para impedir o dificultar subsecuentes modificaciones posteriores por parte de los lectores no es Transparente. Una copia que no es "Transparente" es llamada "Opaca".

Como ejemplos de formatos adecuados para copias Transparentes están el ASCII plano sin formato, formato de Texinfo, formato de LaTeX, SGML o XML usando un DTD disponible publicamente, y HTML simple que siga los estándares, diseñado para modificaciones humanas. Los formatos Opacos incluyen PostScript, PDF, formatos propietarios que pueden ser leídos y editados unicamente en procesadores de palabras propietarios, SGML o XML para los cuáles los DTD y/o herramientas de procesamiento no están disponibles generalmente, y el HTML generado por máquinas, producto de algún procesador de palabras solo para propósitos de salida.

La "Pagina de Título" en un libro impreso significa, la pagina de titulo misma, más las páginas siguientes que sean necesaria para mantener, legiblemente, el material que esta Licencia requiere en la página de titulo. Para trabajos en formatos que no tienen página de título como tal, "Página de Título" significa el texto cercano a la aparición más prominente del título del trabajo, precediendo el comienzo del cuerpo del trabajo.

2. COPIA LITERAL

Puede copiar y distribuir el Documento en cualquier medio, de forma no comercial, siempre y cuando esta Licencia, las notas de derecho de autor, y la nota de licencia que indica que esta Licencia se aplica al Documento se reproduzca en todas las copias, y que usted no adicione ninguna otra condición a las expuestas en en esta Licencia. Usted no puede usar medidas técnicas para obstruir o controlar la lectura o copia posterior de las copias que usted haga o distribuya. Sin embargo, usted puede aceptar compensación a cambio de las copias. Si distribuye un número suficientemente grande de copias también deberá seguir las condiciones de la sección 3.

También puede prestar copias, bajo las mismas condiciones establecidas anteriormente, y puede exhibir copias publicamente.

3. COPIADO EN CANTIDAD

Si publica copias impresas del Documento que sobrepasen las 100, y la nota de Licencia del Documento exige Textos de Cubierta, debe incluir las

copias con cubiertas que lleven en forma clara y legible, todos esos Textos de Cubierta: Textos de título en la portada, y Textos al respaldo de la página de tílo. Ambas cubiertas deben identificarlo a Usted clara y legiblemente como quien publica tales copias. La portada debe mostrar el título completo con todas las palabras igualmente prominentes y visibles. Además puede adicionar otro material en la cubierta. Las copias con cambios limitados en las cubiertas, siempre que preserven el título del Documento y satisfagan estas condiciones, pueden considerarse como copias literales.

Si los textos requeridos para la cubierta son muy voluminosos para que ajusten legiblemente, debe colocar los primeros (tantos como sea razonable colocar) en la cubierta real y continuar el resto en páginas adyacentes.

Si publica o distribuye copias Opacas del Documento cuya cantidad exceda las 100, debe incluir una copia Transparente, que pueda ser leída por una máquina, con cada copia Opaca o entregar en o con cada copia Opaca una dirección en una red de computadores accesible publicamente que contenga una copia completa Transparente del Documento, sin material adicional, a la cual el público en general de la red pueda acceder a bajar anónimamente sin cargo usando protocolos públicos y estandarizados. Si usted hace uso de la última opción, deberá tomar medidas necesarias, cuando comience la distribución de las copias Opacas en cantidad, para asegurar que esta copia Transparente permanecerá accesible en el sitio por lo menos un año después de su última distribución de copias Opacas (directamente o a través de sus agentes o distribuidores) de esa edición al público.

Se solicita, aunque no es requisito, que contacte a los autores del Documento antes de redistribuir cualquier gran número de copias, para darle la oportunidad de que le provean una versión actualizada del Documento.

4. MODIFICACIONES

Puede copiar y distribuir una Versión Modificada del Documento bajo las condiciones de las seccions 2 y 3 anteriores, siempre que usted libere la Versión Modificada bajo esta misma Licencia, con la Versión Modificada haciendo el rol del Documento, por lo tanto licenciando la distribución y modificación de la Versión Modificada a quienquiera posea una copia de esta. Además, debe hacer lo siguiente en la Versión Modificada:

- Usar en la Página de Título (y en las cubiertas, si hay alguna) un título distinto al del Documento, y de versiones anteriores (que deberían, si hay alguna, estar listados en la sección de Historia del Documento). Puede usar el mismo título de versiones anteriores al original siempre y cuando quien publicó originalmente otorge permiso.
- Listar en la Página de Título, como autores, una o más personas o entidades responsables por la autoría o las modificaciones en la Versión Modificada, junto con por lo menos cinco de los autores principales del Documento (Todos sus autores principales, si hay menos de cinco).
- Incluir en la Página de Título el nombre de quién publica la Versión Modificada, como quien publica.
- Preservar todas las notas de derechos de autor del Documento.
- Adicionar una nota de derecho de autor apropiada a sus modificaciones, adyacente a las otras notas de derecho de autor.
- Incluir, immediatamente después de la nota de derecho de autor, una nota de licencia dando el permiso público para usar la Versión Modificada bajo los términos de esta Licencia, de la forma mostrada en el anexo al final de este documento.
- Preservar en esa nota de licencia el listado completo de Secciones Invariantes y de los Textos de Cubiertas que sean requeridos como se especifique en la nota de Licencia del Documento
- Incluir una copia sin modificación de esta Licencia.
- Preservar la sección llamada "Historia", y su título, y adicionar a esta una sección estableciendo al menos el título, el año,los nuevos autores, y quién publique la Versión Modificada como reza en la Página de Título. Si no hay una sección titulada "Historia" en el Documento, crear una estableciendo el título, el año, los autores y quieng publicó el Documento como reza en la Página de Título, añadiendo además una sección describiendo la Versión Modificada como se estableció en la oración anterior.
- Preservar la localización en red, si hay , dada en la Documentación para acceso público a una copia Transparente del Documento, tanto como las otras direcciones de red dadas en el Documento para versiones anteriores en las cuáles estuviese basado. Estas pueden ubicarse en la sección "Historia". Se puede omitir la ubicación en red para un trabajo que sea publicado por lo menos cuatro años antes que el Documento mismo, o si quien publica originalmente la versión da permiso explícitamente.
- En cualquier sección titulada "Agradecimientos" o "Dedicatorias", preservar el título de la sección, y preservar en la sección toda la sustancia y el tono de los agradeimientos y/o dedicatorias de cada contribuyente que estén incluídas.
- Preservar todas las Secciones Invariantes del Documento, sin alterar su texto ni sus títulos. Números de sección o el equivalente no son considerados parte de los títulos de la sección.
- Borrar cualquier sección titulada "Aprobaciones". Tales secciones no pueden estar incluidas en las Versiones Modificadas.
- No retitular ninguna sección existente como "Aprobaciones" o conflictuar con el título de alguna Sección Invariante.

Si la Versión Modificada incluye secciones o apendices nuevos o preliminares al prólogo que califiquen como Secciones Secundarias y contienen material no copiado del Documento, puede opcionalmente designar algunas o todas esas secciones como invariantes. Para hacerlo, adicione sus títulos a la lista de Secciones Invariantes en la nota de licencia de la Versión Modificada. Tales títulos deben ser distintos de cualquier otro título de sección.

Puede adicionar una sección titulada "Aprobaciones", siempre que contenga unicamente aprobaciones de su Versión Modificada por varias fuentes--por ejemplo, observaciones de peritos o que el texto ha sido aprobado por una organización como la definición oficial de un estándar.

Puede adicionar un pasaje de hasta cinco palabras como un Texto de Portada, y un pasaje de hasta 25 palabras al respaldo de la página de título al final de la lista de Textos de Cubierta en la Versión Modificada. Solamente un pasaje de Texto de Portada y uno al respaldo de la página de título puede ser adicionado por (o a manera de arreglos hechos por) una entidad. Si el Documento ya incluye un texto de cubierta para la misma portada o al respaldo de la portada, previamente adicionado por usted o por arreglo hecho por la misma entidad, a nombre de la cual usted está

actuando, usted no puede adicionar otro; pero puede reemplazar el anterior, con permiso explícito de quien publicó previamente y agregó el texto anterior

El(los) autor(es) y quien(es) publica(n) el Documento no dan con esta Licencia permiso para usar sus nombres para publicidad o para asegurar o implicar aprobación de cualquier Versión Modificada.

5. COMBINANDO DOCUMENTOS

Puede combinar el Documento con otros documentos liberados bajo esta Licencia, bajo los términos definidos en la sección 4 anterior para versiones modificadas, siempre que incluya en la combinación todas las Secciones Invariantes de todos los documentos originales, sin modificar, y listadas todas como Secciones Invariantes del trabajo combinado en su nota de licencia.

El trabajo combinado necesita contener solamente una copia de esta Licencia, y múltiples Seccions Invariantes idénticas pueden ser reemplazadas por una sola copia. Si hay múltiples Secciones Invariantes con el mismo nombre pero con contenidos diferentes, haga el título de cada una de estas secciones único adicionándole al final del mismo, entre paréntesis, el nombre del autor o de quien publicó originalmente esa sección, si es conocido, o si no, un número único. Haga el mismo ajuste a los títulos de sección en la lista de Secciones Invariantes en la nota de licencia del trabajo combinado.

En la combinación, debe combinar cualquier sección titulada "Historia" de los varios documentos originales, formando una sección titulada "Historia"; de la misma forma combine cualquier seción titulada "Agradecimientos", y cualquier sección titulada "Dedicatorias". Debe borrar todas las secciones tituladas "Aprobaciones."

6. COLECCIONES DE DOCUMENTOS

Puede hacer una colección consistente del Documento y otros documentos liberados bajo esta Licencia, y reemplazar las copias individuales de esta Licencia en los varios documentos con una sola copia que esté incluida en la colección, siempre que siga las reglas de esta Licencia para cada copia literal de cada uno de los documentos en cualquiera de los demás aspectos.

Puede extraer un solo documento de una de tales colecciones, y distribuirlo individualmente bajo esta Licencia, siempre que inserte una copia de esta Licencia en el documento extraido, y siga esta Licencia en todos los otros aspectos concernientes a la copia literal de tal documento.

7. AGREGACIÓN CON TRABAJOS INDEPENDENTES

Una recopilación del Documento o de sus derivados con otros documentos o trabajos separados o independientes, en cualquier tipo de distribución o medio de almacenamiento, no como un todo, cuenta como una Versión Modificada del Documento, siempre que no se aleguen derechos de autor por la recopilación. Tal recopilación es llamada un "agregado", y esta Licencia no aplica a los otros trabajos auto-contenidos así recopilados con el Documento, o a cuenta de haber sido recopilados, si no son ellos mismos trabajos derivados del Documento.

Si el requerimiento de la sección 3 sobre el Texto de Cubierta es aplicable a estas copias del Documento, entonces si el Documento es menor que un cuarto del agregado entero, los Textos de Cubierta del Documento pueden ser colocados en cubiertas que enmarquen solamente el Documento entre el agregado. De otra forma deben aparecer en cubiertas enmarcando todo el agregado.

8. TRADUCCIÓN

La Traducción es considerada como una clase de modificación, Así que puede distribuir traducciones del Documento bajo los términos de la sección 4. Reemplazar las Secciones Invariantes con traducciones requiere permiso especial de los dueños de derecho de autor, pero usted puede incluir traducciones de algunas o todas las Secciones Invariantes adicionalmente a las versiones originales de tales Secciones Invariantes. Puede incluir una traducción de esta Licencia siempre que incluya también la versión en Inglés de esta Licencia. En caso de un desacuerdo entre la traducción y la versión original en Inglés de esta Licencia, la versión original en Inglés prevalecerá.

9. TERMINACIÓN

No se puede copiar, modificar, sublicenciar, o distribuir el Documento excepto por lo permitido expresamente bajo esta Licencia. Cualquier otro intento de copia, modificación, sublicenciamiento o distribución del Documento es nulo, y serán automáticamente terminados sus derechos bajo esa licencia. Sin embargo, los terceros que hayan recibido copias, o derechos, de su parte bajo esta Licencia no tendrán por terminadas sus licencias siempre que tales personas o entidades se encuentren en total conformidad con la licencia original.

10.REVISIONES FUTURAS DE ESTA LICENCIA

La Free Software Foundation puede publicar versiones nuevas y revisadas de la Licencia de Documentación Libre GNU de tiempo en tiempo. Tales nuevas versiones serán similares en espíritu a la presente versión, pero pueden diferir en detalles para solucionar nuevos problemas o intereses. Vea http://www.gnu.org/copyleft/.

Cada versión de la Licencia tiene un número de versión que la distingue. Si el Documento especifica que se aplica una versión numerada en particular de esta licencia o "cualquier versión posterior", usted tiene la opción de seguir los términos y condiciones de la versión especificada o cualquiera posterior que haya sido publicada (no como un borrador) por la Free Software Foundation. Si el Documento no especifica un número de versión de esta Licencia, puede escoger cualquier versión que haya sido publicada(no como un borrador) por la Free Software Foundation.

1. N. del T. copyleft es un nuevo término acuñado por GNU. Nace de un juego de palabras en Inglés, en vez de "copyright" usan "copyleft", indicando que no se restringe la copia, sino por el contrario se permite mientras que el derecho a seguir copiando no se restrinja.
http://www.gn\u.org/copyleft/copyleft.es.html

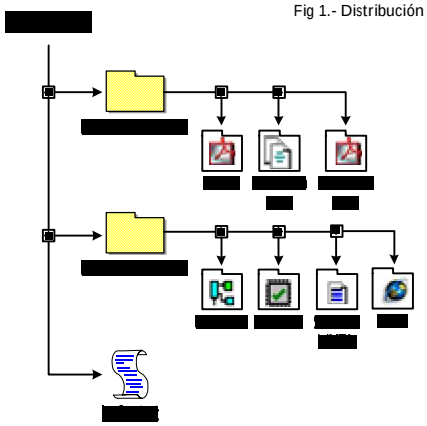
2. N. del T. licenciatario: A quien se le otorga la licencia.

D. CONTENIDO DEL CD-ROM DEL PROYECTO.

El presente proyecto, está compuesto por esta memoria y un CD-ROM que la acompaña. El CD-ROM contiene la información íntegra de la memoria en varios formatos, la página Web y las librerías que modelan los microprocesadores diseñados.

Atendiendo a la normativa de "Presentación de Proyectos en Formato Informático" establecida por la Escuela Universitaria de Ingenieros Técnicos Industriales de Zaragoza, EUITIZ,l CD-ROM del proyecto está estructurado se la siguiente manera:

Fig 1.- Distribución del CD-ROM.



En el directorio Raíz, se pueden encontrar dos directorios y un Archivo:

- DOCUMENTOS.- Contiene la Información relativa a la memoria. Pese a que la normativa sólo exige los archivos en formato PDF (subdirectorio Memoria PDF), he decidido insertar los originales creados con Microsoft Word XP (subdirectorio Memoria DOC). En el subdirectorio PDF's, podemos encontrar una serie de documentos auxiliares, descargados de Internet, que complementan a esta memoria.
- DESARROLLO.- Este directorio contiene todos los archivos generados o utilizados en todas y cada una las fases del desarrollo del proyecto en si. Está dividido en cuatro subdirectorios que explican el la tabla 1.

En el directorio raíz, también aparece un archivo "info.txt" (como indica la norma) que contiene una breve descripción del contenido del CD-ROM.

En la Tabla 1, podemos ver una información un poco más detallada sobre el contenido del CD. En el directorio Software, podemos encontrar algunas versiones de evaluación de los programas utilizados en la generación, simulación y síntesis de los modelos.

Utilizando estas versiones no podremos ni compilar ni sintetizar ni simular completamente las librerías del proyecto. Las versiones de evaluación tienen limitaciones en sus funciones que hacen que sólo se puedan utilizar en diseños de cierta (terando a poca) complejidad. Este proyecto es muy complejo a nivel de programación por lo que si queremos compilar, simular o sintetizar las librerías que se encuentran en el CD-ROM, deberemos hacernos con una licencia que permita utilizar al 100% los programas

DIRECTORIO	DESCRIPCIÓN
[Cd]	CD-ROM del proyecto. acompaña a la Memoria como soporte Informático
[Desarrollo]	Contiene los archivos utilizados en el desarrollo del proyecto.
[Librerias]	Contiene las librerías generadas con el programa Advantage FPGA v5.2
[up8086]	Librería propia del modelo 8086 en los modos máximo y mínimo.
[up8088]	Librería propia del modelo 8088 en los modos máximo y mínimo
[up808x]	Librería común a los dos modelos de los microprocesadores
[up808x_tb]	Librería del Banco de Test de los modelos
[Sintesis]	Contiene la Información Obtenida durante la síntesis de los modelos.
[Software]	En este directorio se encuentran algunas demos de programas utilizados.
[sourceVHDL]	Aquí podemos encontrar el código VHDL de los Modelos.
[modelsim]	Contiene las Macros de Inicialización y las secuencias de compilado.
[programs]	Este Directorio Contiene todos los programas utilizados en la simulación.
[assembler]	Los ficheros escritos en ensamblador se compilan y modifican aquí
[asm]	Contiene los ficheros fuente en ensamblador de los programas a simular.
[lst]	LST guarda el listado generado por Macro Assembler, MASM.
[masm]	En este directorio podemos encontrar el Compilador de MASM.
[turboC]	Contiene los archivos fuente escritos en Turbo C++.
[sources]	Contiene los archivos Fuente VHDL generados. y listos para compilar.
[up8086]	Archivos VHDL del modelo 8086 en los modos máximo y mínimo.
[up8088]	Archivos VHDL del modelo 8088 en los modos máximo y mínimo
[up808x]	Archivos VHDL comunes a los dos modelos de los microprocesadores
[up808x_tb]	Archivos VHDL de la librería del Banco de Test de los modelos
[web]	Página Web del Proyecto "http//:perso.wanadoo.es/odradek"
[Documentos]	Este Directorio contiene esta memoria y algunos PDF's que la complementan
[Historial]	Contiene un Documento "histórico" de los cambios realizados en el proyecto.
[Memoria Doc]	Memoria en el formato original de Office XP (*.doc)
[Memoria PDF]	Memoria guardada en formato PDF (Portable Data File)
[PDFs]	Este directorio, contiene algunos archivos PDF descargados de Internét.
[Apuntes de Computadores]	Apuntes de la asignatura "Computadores" Cursada en 2º Año.
[Booth Multiplier]	Información del algoritmo de Booth serie para la multiplicación de enteros.
[DataSheet]	DataSheet de los micros originales. Cortesía de Intel (r).
[Ensamblador]	Algunas nociones de programación en ensamblador
[Vhdl]	Manuales completos del estándar VHDL.

Tab 1.- Información Detallada Sobre el Contenido del CD-ROM del Proyecto.

E. REFERNCIAS BIBLIOGRÁFICAS:

E.1 Libros Utilizados:

L1.-TRIEBEL SINGH, “The 8088 and 8086 Microprocessors, Programing, Interfacing, Software, Hardware, and Applications”, Third Edition. Prentice Hall.

L2.-BREY, “Los Microprocesadores Intel, 8086/8088/80186/80286/80386 y 80486”, Tercera Edición. Prentice Hall.

L3.-ANGULO, “Electrónica Digital Moderna”. Editorial Paraninfo.

L5.-ANTONIO MATA, "Turbo C, C++. Iniciación y Programación". Editorial Paraninfo.

L6.-JOSÉ Mª ANGULO, "Microprocesadores, Arquitectura Programación y Diseño de Sistemas". Editorial Paraninfo.

L7.-TOCCI, "Sistemas Digitales, Principios y Aplicación", Cuarta Edición. Prentice Hall.

L8.-J.M. ANGULO, "Electrónica Digital Moderna", Editorial Paraninfo.

L9.-TOMÁS POLLÁN SANTAMARÍA, "Electrónica Digital", Prensas Universitarias de Zaragoza, PUZ.

L10.-BONIFACION MARTÍN DEL BRIO, "Sistemas Electrónicos Basados en Microprocesadores y Microcontroladores", Prensas Universitarias de Zaragoza, PUZ.

L11.-S. LANCETA & L.PECIÑA, "Microcontroladores Industriales MC-51", EDEBÉ.

L12 J.M. ANGULO, "Curso Práctico de Microelectrónica y Microinformática Industrial"", Editorial Paraninfo.

L13 Javier Martinez Pérez &Mariano Barrau Ruíz"Prácticas Con Microcontroladores de 8 bits", Mc Graw Gill.

E2. Documentación en Papel:

D1.- TOMÁS POLLÁN SANTAMARÍA "uE, Microelectrónica". Apuntes de la asignatura Microelectrónica, EUITIZ.

D2.- Apuntes de la Asignatura "Microelectrónica, Diseño ASIC", cursada en la facultad de Ciencias.

D3.- SYNOPSYS, “VHDL Compiler, Reference Manual”.

D4.- A.P Paplinnski, "Word-Serial Multiplication Processor, the Booth's Algorithm".

D5. John Kubiatowicz "Computer Architecture Engineering".

D6.- Apuntes de Computadores, EUITIZ.

E3. Documentos en PDF consultados:

Tab 1.- Documentos PDF consultados Durante el Diseño de las CPU's.

P?.-PDF ARCHIVO (DIRECTORIO .\PDFS\)	FUENTE	DESCRIPCIÓN
P1.-80186-80188 High-Integration 16-bit Microprocessors Datasheet.pdf P2.-8086 16-bit HMOS Microprocessor datasheet.pdf P3.-8088 8-bit HMOS microprocessor datasheet.pdf P4.-80C187 80-Bit Math Coprocessor Datasheet.pdf P5.-80C88 (FN2949).pdf P6.-8085ovr.pdf P7.-86016.pdf	Intel ®	DataSeets de Intel. La información de estos archivos PDF, ha sido meticulosamente estuciada al abordar los sucesivos problemas con los que me encontraba al tratar de comprender el funcionamiento de los micros.
P8.-80C186EA-80C188EA Microprocessor User's Manual.pdf P9.-Intel Architecture Optimizations Manual.pdf P10.-Intel Architecture Software Developer's Manual, Volume 1, Basic Architecture.pdf P11.-Intel Architecture Software Developer's Manual, Volume 2, Instruction Set Reference Manual.pdf P12.-Intel Architecture Software Developer's Manual, Volume 3, System Programming Guide.pdf		Manuales de Usuario. El manual de usuario de los micros 8088/8086 no está disponible. Tube que utilizar el del Micro más Próximo, el 80188/80186.
P13.-High Speed Numerics with the 80188-80186 and 8087.pdf P14.-AB-36 80186-80188 DMA Latency.pdf P15.-The 8088 Bus Cycle and Timing (lec07).pdf		Estudios. Estos Estudios y aplicaciones han sido muy útiles para comprender el interface de los Micros.
P16.-Booth Multipier (original name L08.pdf).pdf	A.P.Papliński (11 de Julio de2001)	Explicación Detallada del Algoritmo de Booth para la Multiplicación Con y Sin Signo
P17.-lec06-div-fp.pdf (directorio Arquitectura de Computadores)	John Kubiatowicz (http.cs.berkeley.edu/~kubitron) 11 de Octubre de 1999)	Transparencias de la asignatura de arquitectura de ordenadores de una universidad polaca, la utilicé para desarrollar los algoritmos de Mul DiV)
P18.-Arquitectura de Computadores (Ensamblador).pdf	Eduardo René Rodríguez Ávila	Utilizado para la programación en ensamblador y para el estudio de las Instrucciones
P19.-IEEE Standard VHDL Language Reference Manual	IEEE	Útil Guia de VHDL
P20.-synopsys_vcrm.pdf	Synopsys	Guía utilizada para escribir código sintetizable
P21.-VHDL-Cookbook.pdf	Peter J. Andersen	Una guía muy buena del lenguaje VHDL.

E4. Páginas de Internet Visitadas:

URL VISITADA	CONTENIDO
W1.- http://www.cs.ucr.edu/content/esd/labs/tutorial/VHDL_Page.html	VHDL Online - A Collection of VHDL Related Internet Resources
W2.- http://alumnat.upv.es/pla/visfit/1466/AAABNcAATAAAErSAAQ/bibintel.htm	sandpile.org -- Intel manuals
W3.- http://www.techboss.com/hardware/cpu/cpu.htm	CPU - A History
W4.- http://www.dcd.com.pl/	DCD - Digital Core Design - VHDL and Verilog IP Cores supplier
W5.- http://www.fpgacpu.org/	fpgacpu.org - FPGA CPU News
W6.- http://www.estec.esa.nl/wsmwww/leon/	LEON-1 VHDL model Microprocessor
W7.- http://www.opencores.org/projects?category=microprocessor	OPENCORES
W8.- http://www.ashenden.com.au/	Peter Ashenden
W9.- http://www.pldworld.com/_hdl/1/rassp.aticorp.org/vhdl/models/processor.html	Processor Models
W10.- http://home.teleport.com/~scd/	Sierra Circuit Design VHDL microprocessor DSP 6502 65c02 MELPS740 1802 8085 6800 6805 68705 6809 UART SPI
W11.- http://tech-www.informatik.uni-hamburg.de/vhdl/	The Hamburg VHDL Archive
W12.- http://www.ecsi.org/EARNEST/digests/VHDL_cookbook/default.htm	VHDL Cookbook
W13.- http://www.eda.org/vhdl_intl/vhdltimes/73-flatpanel.html	VHDL International [VI] - VHDL Times
W14.- http://www.csee.umbc.edu/help/VHDL/samples/samples.shtml	VHDL samples Mul_Div
W15.- http://www.quantasm.com/opcode_i.html	80x86 Integer Instruction Set (8088 - Pentium)
W16.- http://research.microsoft.com/~gbell/Designing_Computers_and_Digital_Systems	Algorítmos de Multiplication
W17.- http://w3.mor.itesm.mx/~jfrausto/Algoritmos/introduccion.html	Algorítmo Multiplication
W18.- http://www.vautomation.com/	ARC International's Vautomation product line - Synthesizable BVCI Ethernet, USB 1.1 USB 2.0 Host Device, 8086, 80186, 8-bit RISC microprocessor VHDL Verilog soft IP cores
W19.- http://webster.cs.ucr.edu/	Art of Assembly Language Programming and HLA by Randall Hyde
W20.- http://webster.cs.ucr.edu/Page_AoALinux/HTML/RealArithmetic.html	Chapter Eleven Real Arithmetic 80x87
W21.- http://www.cs.berkeley.edu/~kubitron/courses/cs152-F99/	Fall 1999 CS152: Computer Architecture and Engineering
W22.- http://www.ami.ac.uk/courseware/adveda/ch6/adveda06notes.html	Mul unsigned
W23.- http://www.clipx.net/ng/masm/index.php	Norton Guide
W24.- http://webster.cs.ucr.edu/Page_asm/ArtofAssembly/0_ArtofAsm.html	The Art of Assembly Language Programming

E5. Programas utilizados:

PROGRAMA	VERSIÓN	UTILIDAD
E1.- Active HDL	v5.1, v6.2	Generación del Modelo
E2.- Modelsim	v5.5, v5.7	Simulación del Modelo
E3.- Synopsis		Sintetización del Modelo
E4.- Leonardo Spectrum		Síntesis del Modelo
E5.- Xilinx Webpack	v5.1	Síntesis dek Modelo.
E6.- AutoCAD	2002	Generación de los planos
E7.- Debug		Depuración de los programas de prueba
E8.- MacoAnssembler	v7.0	Generación de los programas de prueba
E9.- Turbo C, C++	v1.01	Compilación de Archivos C
E10.- Win Word	XP	Generación de los documentos
E11.- Visio Pro	XP	Generación de la Figuras de la Memoria.
E12.- The Norton Guides	-	Guia de ayuda TSR
E13.- Norton Commander	v6.03	Manejador de Archivos
E14.- Macromedia Flash	v4.0	Generación de Animaciones flash y Presentación
E15.- Macromedia Dreamweaver	v6.0	Generación de la página Web
E16.- Macromedia Flash	v6.0	Genreración de la Mascota y Animaciones.
E17.- VIM Vi –IMproved	v6.1	Editor de textos.
E18.- GWD Text Editor	v3.2	Utilizado para Expandir los Taburadores
E19.- Acrobat Reader	v5.0	Generación de PDF's
E20.- Adobe Phothoshop	v7.0	Edición de Imágenes.
E21.- Adobe Illustrator	v8.0	Tratamiento de Ficheros de Vectores
E22.- Paint Shop Pro	v7.02	Utilizado para la Captura de Pantallas
E23.- ASCII Pick	v2.0	Generación de Imágenes ASCII
E24.- Nero Burning	v5.5	Creación del CD-ROM del Proyecto
E25.- CD_Label	v3.2	Utilizado para las pegatinas del CD.

Tab 3.- Relación de Programas Utilizados Durante la Realización de Este Proyecto.

Tab 2.- Direcciones De Internet Visitadas, Relacionadas con la Temática del Proyecto.