TDA 1819: Descripción y simulación de una computadora de 32 bits

Autor: Scorza, Facundo Ricardo

Cátedra: Taller de Arquitectura (Adscripción a la Docencia)

Contacto: zorito1994@yahoo.es

Facultad de Informática, UNLP, 50 y 120, La Plata

Tutor: Villagarcía Wanza, Horacio Alfredo

Índice

1. Introducción 1

2. Entorno de trabajo 5

2.1. Lenguaje de programación 5

2.2. Simulador 7

2.2.1. Instalación 12

2.3. Diseño de la computadora original 21

3. TDA 1819 36

3.1. Arquitectura 44

3.1.1. Atributos generales 44

3.1.2. Directivas y repertorio de instrucciones 52

3.1.3. Códigos de operación e identificadores 59

3.1.4. Formatos de instrucciones 64

3.1.5. Estructura de la memoria principal 69

3.1.5.1. Programa de prueba 69

3.1.6. Ciclo de ejecución: segmentación del cauce 82

3.1.7. Arquitecturas von Neumann y Harvard 94

3.1.8. Riesgos de la segmentación y atascos del cauce 100

1. Introducción

Aunque actualmente la arquitectura de una computadora sea uno de los campos de estudio más relevantes de la informática, no existen en la práctica demasiadas implementaciones que describan y simulen efectivamente el comportamiento de un procesador de forma tal que pueda ser comprendido y utilizado por los estudiantes y/o profesionales del área para llevar a cabo distintos experimentos científicos y/o académicos, desde comprobar la respuesta ofrecida por la CPU durante la ejecución de diversos programas escritos en el lenguaje Assembler hasta incorporar nuevas funcionalidades a la PC que aún se encuentren pendientes de ser incluidas, ya sea algún periférico de entrada/salida o bien alguna mejora que pueda repercutir favorablemente sobre el rendimiento del procesador (segmentación, unidad de punto flotante, etc.).

Como un ejemplo de simulador existente en este campo y utilizados en el ámbito académico con fines educativos se pueden mencionar el MSX88, concebido en el año 1985 en la Escuela Universitaria de Ingeniería Técnica de Telecomunicación de la Universidad Politécnica de Madrid como respuesta a las necesidades docentes en el área de la Arquitectura de Ordenadores. El procesador utilizado consiste en una versión simplificada de la CPU 8088 de Intel: arquitectura interna de 16 bits y externa de 8 bits, bus de direcciones de 16 bits, arquitectura de tipo CISC (Repertorio Complejo de Instrucciones). Otro caso podría ser el WinMIPS64, el cual posee una finalidad muy similar al anterior pero presenta una drástica diferencia respecto a la arquitectura del procesador utilizado para la simulación, ya que aquí la CPU descripta es un MIPS64 con una arquitectura de 64 bits y de tipo RISC (Repertorio Reducido de Instrucciones). No cabe ninguna duda de que, con sus contrastes, ambos simuladores han cumplido su propósito exitosamente ya que, con algunas actualizaciones y modificaciones, aún al día de la fecha siguen siendo empleados por profesores universitarios de esta rama para introducir a los alumnos a la programación de una CPU a bajo nivel.

A modo recordatorio se presentan a continuación en modo sintético las principales diferencias entre una arquitectura CISC y una RISC. Se debe tener en cuenta que no existen procesadores cuyo diseño represente de manera fiel y pura ninguno de los dos tipos de arquitectura, ya que la mayoría es en realidad una mezcla de ambos, aunque siempre con más características de uno que de otro:

Tabla 1. Comparación entre las arquitecturas CISC (Repertorio Complejo de Instrucciones) y RISC (Repertorio Reducido de Instrucciones).

Arquitectura / **Característica**	**CISC**	**RISC**
Ciclos de ejecución	Cada instrucción puede requerir un número variable de ciclos de reloj para ejecutarse	Cada instrucción requiere exactamente un único ciclo de reloj para ejecutarse
Tipo de diseño	Diseño centrado en el hardware: la unidad de control realiza la mayor parte del trabajo	Diseño centrado en el software: los compiladores de alto nivel realizan la mayor parte del trabajo
Uso de la memoria RAM	Utilización eficiente del espacio disponible en la memoria RAM (menor cantidad de instrucciones por programa)	Ineficiente utilización del espacio disponible en la memoria RAM (pueden producirse cuellos de botella si la RAM es limitada)
Interfaz de Entrada/Salida	E/S independiente: espacios de direcciones y repertorios de instrucciones separados para memoria y E/S.	E/S mapeada en memoria: memoria y E/S comparten un único espacio de direcciones y repertorio de instrucciones.
Formato y longitud de las instrucciones	Instrucciones complejas de longitud variable	Instrucciones sencillas y estandarizadas de longitud fija
Microprogramación	Puede soportar microcódigo (microprogramación: las instrucciones complejas son tratadas como pequeños programas)	Sólo se permite una única capa de instrucciones
Cantidad de instrucciones	Gran número de instrucciones	Pequeña cantidad de instrucciones
Modos de direccionamiento	Modos de direccionamiento complejos	Número reducido de modos de direccionamiento
Cantidad de registros	Cantidad reducida de registros de almacenamiento	Elevado número de registros de almacenamiento
Segmentación	La implementación de la	La implementación de la

	segmentación es complicada	segmentación es sencilla
Programación	La codificación de los programas es más sencilla	Los programas requieren un mayor número de líneas de instrucciones
Almacenamiento en memoria	Little Endian: los bytes se almacenan desde el menos significativo en la dirección más baja hacia el más significativo en la más alta	Big Endian: los bytes se almacenan desde el más significativo en la dirección más baja hacia el menos significativo en la más alta

Ambos simuladores mencionados anteriormente resultan extremadamente útiles para introducir a los estudiantes que están en las etapas iniciales de su carrera a la programación de una CPU y al funcionamiento de esta última durante la ejecución de diversos programas. No obstante, cuando el docente se encuentra en la presencia de alumnos más avanzados, su objetivo ahora es que los alumnos no se comporten como meros espectadores del comportamiento del procesador sino que participen aún más activamente en él, conociendo interiormente el código que se encuentra detrás del funcionamiento de cada uno de los componentes y de su interacción entre sí a fin de poder modificarlo para mejorar el rendimiento de la CPU incorporando nuevas técnicas aún no implementadas o bien agregar más funcionalidades a la PC con la inclusión de periféricos de E/S adicionales todavía faltantes en la misma.

En este apartado es donde debe destacarse la importancia de los lenguajes de descripción de hardware (HDLs, Hardware Description Languages), utilizados desde los años 70 en los ciclos de diseño de sistemas digitales asistidos por herramientas de CAD electrónico. En los años 80 aparecen los lenguajes Verilog y VHDL que, aprovechando la disponibilidad de herramientas hardware y software cada vez más potentes y asequibles y los adelantos en las tecnologías de fabricación de circuitos integrados, logran imponerse como herramientas imprescindibles en el desarrollo de nuevos sistemas. Estos lenguajes son sintácticamente similares a los de programación de alto nivel –Verilog tiene una sintaxis similar al C y VHDL a ADA– y se diferencian de éstos en que su semántica está orientada al modelado del hardware. Su capacidad para permitir distintos enfoques en el modelado de los circuitos y su independencia de la tecnología y metodología de diseño permiten extender su uso a los distintos ciclos de diseño que puedan utilizarse. Por ello, para los profesionales

relacionados de alguna manera con el diseño o mantenimiento de sistemas digitales resulta hoy en día imprescindible su conocimiento.

A partir de la información anterior se puede concluir que, al tratarse efectivamente la PC de un sistema digital, su diseño puede ser perfectamente descripto mediante cualquiera de estos lenguajes de manera tal de obtener una representación de la misma tan fiel como se desee sin requerir en un principio ningún tipo de hardware en particular para comprobar su funcionamiento. Por lo tanto, sería extremadamente útil contar con un simulador que posea características similares a aquéllos mencionados anteriormente pero que se encuentre escrito en algún lenguaje de descripción de hardware de manera tal que, además de permitir a los alumnos codificar programas en Assembler para ejecutarlos en una CPU y comprobar el funcionamiento de esta última, no los limite a contemplar la actividad del procesador de forma externa y pasiva a través de una interfaz gráfica sino que les ofrezca la oportunidad de analizar internamente todos los componentes y señales involucrados para que puedan afianzar todos los conocimientos adquiridos en cátedras anteriores vinculadas con este campo y comprender su relevancia si se desea llevar a cabo un diseño para describir una PC o CPU absolutamente original o bien mejorar el desempeño de una ya existente.

A pesar de que existen en la actualidad diversas propuestas de descripciones e implementaciones para distintos procesadores escritas en alguno de estos lenguajes de descripción de hardware, aún ninguna de ellas es tan completa ni fiable como para ser utilizada de manera universal en el ámbito académico de la manera en que sí lo son los dos simuladores previamente mencionados.

Por lo tanto, se propuso como objetivo para este proyecto desarrollar el diseño de una PC a través de un HDL basado en las descripciones ofrecidas por dichos simuladores de forma tal que los estudiantes más avanzados puedan implementar programas en el lenguaje Assembler para ejecutarlos en la CPU tal como lo hacían en cátedras anteriores en el MSX88 y el WinMIPS64 pero además sean capaces de escribir código en un lenguaje de descripción de hardware para mejorar el rendimiento del procesador o incorporar nuevas funcionalidades tanto a la CPU como a la computadora que la contiene.

La denominación seleccionada tanto para la PC como para la CPU diseñadas es “TDA 1819”: TDA por la cátedra para la cual este proyecto fue inicialmente concebido (Taller de Arquitectura) y 1819 por el período durante el cual esta primera versión fue desarrollada (2018-2019).

Figura 2. Logo diseñado para la PC y la CPU del proyecto: denominación “TDA 1819 (MDCCCXIX en números romanos)”.

2. Entorno de trabajo

En esta sección se describirán tanto el lenguaje de programación como el simulador utilizados para describir el diseño desarrollado para la PC, junto con la descripción que había sido originalmente seleccionada como punto de partida para el mismo, así como también los motivos que llevaron al autor a determinar tales decisiones. Este contenido es simplemente de carácter introductorio en caso de que el usuario aún no se encuentre familiarizado con el uso de lenguajes de descripción de hardware ni con el simulador en particular elegido para este proyecto.

2.1. Lenguaje de programación

Como cabría esperarse a partir de todos los conceptos desarrollados anteriormente se optó por utilizar un lenguaje de descripción de hardware para describir el diseño de la PC. El lenguaje seleccionado en particular para este proyecto es aquél conocido como VHDL (VHSIC Hardware Description Language; VHSIC = Very High Speed Integrated Circuit).

Los estudios para la creación de este lenguaje comenzaron en el año 1981, bajo la cobertura de un programa para el desarrollo de Circuitos Integrados de Muy Alta Velocidad (VHSIC), del Departamento de Defensa de los Estados Unidos. En 1983 las compañías Intermetrics, IBM y Texas Instruments obtuvieron la concesión de un proyecto para la realización del lenguaje y de un conjunto de herramientas auxiliares para su aplicación. Finalmente, en el año 1987, el lenguaje VHDL se convierte en la norma IEEE-1076 –como

todas las normas IEEE, se somete a revisión periódica, por lo que en 1993 sufrió algunas leves modificaciones–.

El lenguaje VHDL fue creado con el propósito de especificar y documentar circuitos y sistemas digitales utilizando un lenguaje formal. En la práctica se ha convertido, en un gran número de entornos de CAD, en el HDL de referencia para realizar modelos sintetizables automáticamente. Las principales características del lenguaje VHDL se explican en los siguientes puntos:

- Descripción textual normalizada: El lenguaje VHDL es un lenguaje de descripción que especifica los circuitos electrónicos en un formato adecuado para ser interpretado tanto por máquinas como por personas. Se trata además de un lenguaje formal, es decir, no resulta ambiguo a la hora de expresar el comportamiento o representar la estructura de un circuito. Está, como ya se ha dicho, normalizado, o sea, existe un único modelo para el lenguaje, cuya utilización está abierta a cualquier grupo que quiera desarrollar herramientas basadas en dicho modelo, garantizando su compatibilidad con cualquier otra herramienta que respete las indicaciones especificadas en la norma oficial. Es, por último, un lenguaje ejecutable, lo que permite que la descripción textual del hardware se materialice en una representación del mismo utilizable por herramientas auxiliares tales como simuladores y sintetizadores lógicos, compiladores de silicio, simuladores de tiempo, de cobertura de fallos, herramientas de diseño físico, etc.
- Amplio rango de capacidad descriptiva: El lenguaje VHDL posibilita la descripción del hardware con distintos niveles de abstracción, pudiendo adaptarse a distintos propósitos y utilizarse en las sucesivas fases que se dan en el desarrollo de los diseños. Además es un lenguaje adaptable a distintas metodologías de diseño y es independiente de la tecnología, lo que permite, en el primer caso, cubrir el tipo de necesidades de los distintos géneros de instituciones, compañías y organizaciones relacionadas con el mundo de la electrónica digital; y, en el segundo, facilita la actualización y adaptación de los diseños a los avances de la tecnología en cada momento.
- Otras ventajas: Además de las ventajas ya reseñadas también es destacable la capacidad del lenguaje para el manejo de proyectos de grandes dimensiones, las garantías que comporta su uso cuando, durante el ciclo de mantenimiento del proyecto, hay que sustituir componentes o realizar modificaciones en los circuitos, y el hecho de que, para muchas organizaciones contratantes, sea parte indispensable de la documentación de los sistemas.

Se puede mencionar adicionalmente que se posee el conocimiento de que los alumnos de la cátedra para la cual fue inicialmente desarrollado este proyecto ya han programado en

este lenguaje para diseñar circuitos digitales en asignaturas anteriores de la carrera. Por lo tanto, cuentan con la capacidad para ponerse a estudiar y trabajar directamente sobre el código ya disponible para describir la PC sin tener que atravesar una etapa previa de aprendizaje que sería imprescindible en caso contrario al tratarse de un diseño tan complejo como éste, aprovechándose así más eficientemente el tiempo disponible de las clases para que los estudiantes puedan alcanzar los objetivos propuestos para la cátedra.

```
 1 library ieee;
 2 use ieee.std_logic_1164.all;
 3 use ieee.numeric_std.all;
 4
 5 entity signed_adder is
 6   port
 7   (
 8     aclr : in   std_logic;
 9     clk  : in   std_logic;
10     a    : in   std_logic_vector;
11     b    : in   std_logic_vector;
12     q    : out  std_logic_vector
13   );
14 end signed_adder;
15
16 architecture signed_adder_arch of signed_adder is
17   signal q_s : signed(a'high+1 downto 0); -- extra bit wide
18
19 begin -- architecture
20   assert(a'length >= b'length)
21     report "Port A must be the longer vector if different sizes!"
22     severity FAILURE;
23   q <= std_logic_vector(q_s);
24
25   adding_proc:
26   process (aclr, clk)
27     begin
28       if (aclr = '1') then
29         q_s <= (others => '0');
30       elsif rising_edge(clk) then
31         q_s <= ('0'&signed(a)) + ('0'&signed(b));
32       end if; -- clk'd
33     end process;
34
35 end signed_adder_arch;
```

Figura 3. Código escrito en el lenguaje VHDL para describir el comportamiento de un sumador con signo.

2.2. Simulador

El simulador elegido para comprobar el funcionamiento del diseño desarrollado para la PC es el Active-HDL, una solución integrada de creación y simulación de diseños desarrollada por la compañía Aldec para FPGA (Matrices de Compuertas Programables) orientada a entornos de trabajo en equipo y concebida para ser instalada sólo en el sistema operativo Microsoft Windows (XP/2003/Vista/7/8/10 32/64 bits). El IDE (Entorno de Diseño

Integrado) incorporado en el Active-HDL incluye un conjunto completo de herramientas dedicadas para los diseños gráficos y la programación en lenguajes de descripción de hardware, junto con un simulador de lenguaje mixto de conexiones de compuertas lógicas para un rápido despliegue y verificación de diseños para FPGA. El gestor para el flujo del diseño invoca más de ciento veinte herramientas para EDA (Automatización de Diseños Electrónicos) y FPGA durante los flujos de diseño de entrada, simulación, síntesis e implementación y les permite a los distintos equipos permanecer dentro de una única plataforma común a través de la totalidad del proceso de desarrollo para la FPGA. El Active-HDL ofrece soporte para los dispositivos FPGA punteros de las compañías Altera, Atmel, Lattice, Microsemi (Actel), Quicklogic y Xilinx, entre otras.

Algunas de sus principales ventajas incluyen:

- Gestor de diseño unificado para entornos de trabajo en equipo.
- Despliegue ágil y veloz de diseños con editores de texto, esquemático y máquina de estados.
- Simulador potente de lenguaje mixto de kernel común (VHDL, Verilog, SystemVerilog (Diseño), SystemC).
- Depuración (debugging) avanzada y cobertura de código.
- Verificación basada en aseveraciones: SVA (SystemVerilog Assertions), PSL (Property Specification Language), OVA (OpenVera Assertions), etc.
- Co-simulación DSP (Procesamiento Digital de Señales) con interfaz MATLAB/Simulink.
- Posibilidad de compartir diseños de una manera rápida con la herramienta de Autogeneración de Documentación del Diseño en los formatos HTML y PDF.

Figura 4. Logo utilizado por el simulador Active-HDL.

Con respecto a la tarea del diseño, el Active-HDL utiliza métodos gráficos y textuales para el Design Entry (Entrada del Diseño) e integra más de ciento veinte herramientas para EDA en una plataforma única. Las herramientas para la gestión de diseños colaboran para eliminar los problemas enfrentados por los entornos basados en trabajo de equipo durante el proceso de desarrollo del diseño para la FPGA.

Este simulador también incorpora un simulador de lenguaje mixto de kernel común con herramientas interactivas que le permite a los diseñadores una rápida depuración del código escrito para el diseño. Las herramientas de depuración como el Advanced Data Flow (Flujo de Datos Avanzado) y el Xtrace proveen a los usuarios una representación gráfica de las señales internas del sistema, incrementando así la visibilidad y asistiendo en la depuración de diseños complejos. El Active-HDL además incluye herramientas para el análisis y la cobertura de código, permitiendo a los diseñadores incorporar una verificación orientada a métricas dentro del proceso de diseño.

Finalmente, el Active-HDL le posibilita a los diseñadores documentar de una manera veloz todos los aspectos del espacio de trabajo de su diseño para su posterior revisión, reutilización y preservación. Esto a su vez facilita la capacidad para mantener una documentación apropiada de todas las etapas del proceso de desarrollo, eliminándose así muchos inconvenientes surgidos por los entornos de diseño que involucran el trabajo simultáneo de múltiples equipos.

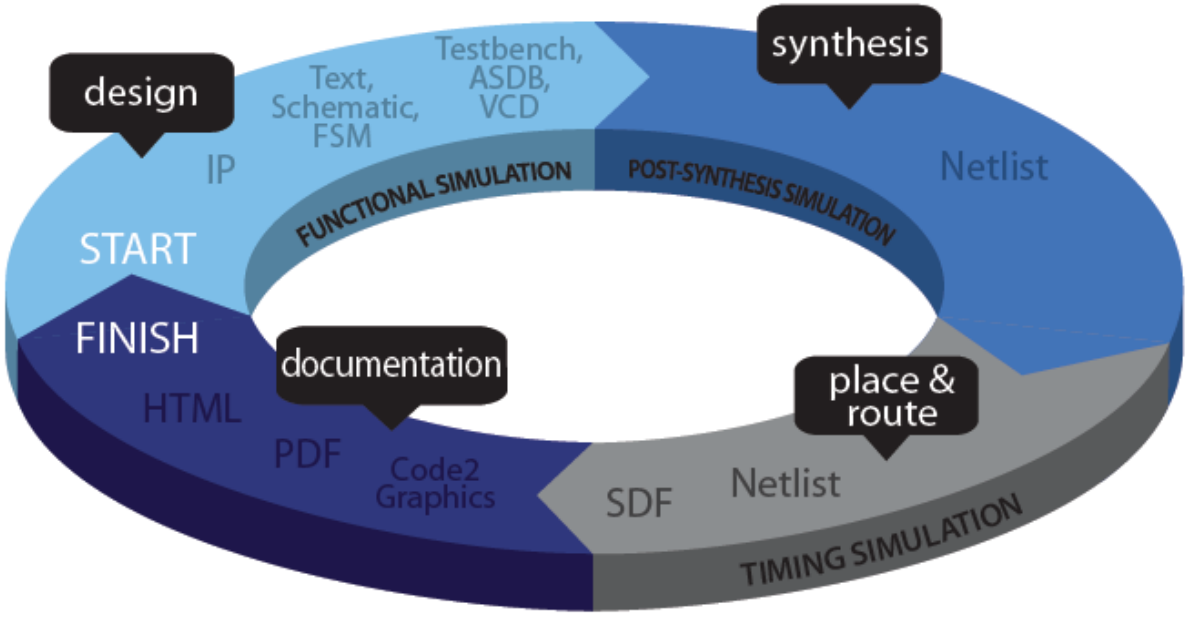

Figura 5. Flujo de Diseño para FPGA.

A continuación se incluyen algunas definiciones para facilitar la comprensión del gráfico anterior:

- IP (Intellectual Property): Un núcleo o bloque IP (Intellectual Property) es un plan de diseño reutilizable de una unidad lógica, celda o circuito integrado (comúnmente denominado “chip”) cuya propiedad intelectual pertenece a sólo un ente. Los núcleos IP pueden ser utilizados como bloques fundacionales dentro de diseños para circuitos integrados para aplicaciones específicas (ASICs) o FPGAs.
- FSM (Finite State Machine): Se denomina máquina de estados a un modelo de comportamiento de un sistema con entradas y salidas, en donde las salidas dependen no sólo de las señales de entradas actuales sino también de las anteriores. Las máquinas de estados se definen como un conjunto de estados que sirve de intermediario en esta relación de entradas y salidas, haciendo que el historial de señales de entrada determine, para cada instante, un estado para la máquina, de forma tal que la salida depende únicamente del estado y las entradas actuales. Una máquina de estados se denomina máquina de estados finitos (FSM por finite state machine) si el conjunto de estados de la máquina es finito, este es el único tipo de máquinas de estados que podemos modelar en un computador en la actualidad; debido a esto se suelen utilizar los términos máquina de estados y máquina de estados finitos de forma intercambiable.
- ASDB (A Simulation Database): El Active-HDL almacena datos de las simulaciones en un archivo ASDB. Este archivo contiene el historial de los valores de algunas señales particulares y jerarquías del diseño cuyo comportamiento fue seleccionado para ser estudiado durante el transcurso de la simulación. Para ver el contenido del archivo ASDB debe encontrarse vinculado con un archivo AWC (A Waveform Configuration), el cual a su vez contiene información respecto a la lista de los objetos que son representados, el color, el sistema numérico de representación, el orden, etc.
- VCD (Value Change Dump): Es un formato de archivo especificado en el estándar IEEE 1364-1995. El archivo VCD consiste en realidad de un archivo ASCII que contiene información de cabecera, definiciones de variables y cambios en los valores de dichas variables. Los archivos VCD almacenan información sobre cambios de valores durante las simulaciones realizadas para redes y registros. Pueden exportarse formas de onda (“waveforms”) conteniendo señales VHDL a un archivo VCD.
- Netlist: Es una descripción de la conectividad de un circuito electrónico. En su forma más simple, consiste en una lista de los componentes electrónicos de un circuito junto con otra

lista de los nodos a los cuales se encuentran conectados. Una red ("net") es una colección de dos o más componentes interconectados. La estructura, complejidad y representación de las netlists puede variar considerablemente, pero el próposito fundamental de cada netlist es expresar información sobre conectividad. Las netlists usualmente proveen únicamente instancias, nodos y quizás algunos atributos de los componentes involucrados.

- SDF (Standard Delay File): En las etapas iniciales del diseño se llevarán a cabo simulaciones "funcionales" para comprobar que la lógica en el circuito desarrollado funcione correctamente. Después de que las herramientas FPGA seleccionadas logren generar una disposición propiamente dicha para el diseño de las compuertas lógicas, podría ser recomendable llevar a cabo una simulación final con información sobre retardos de propagación generada durante el proceso de disposición de las compuertas. Esta información es utilizada frecuentemente como una verificación final para comprobar que los retardos inesperados generados durante el proceso de disposición no creen violaciones de tiempo en el diseño. Las herramientas de disposición crearán un archivo SDF que incluya esta información. Al incluir esta última, el modelo puede ser puesto a prueba basándolo en estos retardos de propagación.
- Code2 Graphics: CODE-2 (Computador Didáctico Elemental, versión 2), y su predecesor ODE son dos procesadores ideados por los profesores Alberto Prieto y Antonio Lloris, de la Universidad de Granada, para facilitar la compresión del funcionamiento y el diseño hardware de un computador. Ambos procesadores son de tipo RISC, con un formato de instrucciones completamente regular, y con 16 instrucciones máquina. La longitud de palabra (tanto de la ALU como de la memoria) de ODE es de 12 bits, mientras que la de CODE-2 es de 16 bits. La concepción y diseño de estos ordenadores se realizó con el objetivo de que el modelo pudiese utilizarse para enseñanza del lenguaje máquina, del lenguaje ensamblador y realizar con los alumnos su diseño completo a nivel de puertas lógicas y circuitos integrados de media escala (decodificadores, multiplexores, etc.), así como con circuitos de muy gran escala (con una FPGA, por ejemplo). Existe una abundante documentación sobre CODE-2, así como material didáctico (ensamblador cruzado, emulador, etc.) definido como software libre. Además, hay una versión comercializada del mismo.

2.2.1. Instalación

La versión del simulador utilizada para este proyecto fue el Active-HDL Student Edition 2018, una alternativa sin costo alguno ofrecida por Aldec para que los estudiantes puedan utilizarla durante su aprendizaje y que también incluye una licencia que les permite comenzar a usar el simulador inmediatamente después de instalarlo, sin necesidad de ningún tipo de activación previa. La misma se encuentra disponible para su descarga en el sitio web oficial de Aldec: https://www.aldec.com/students/student.php?id=9; como se mencionó anteriormente, en forma absolutamente gratuita.

El único requerimiento exigido por Aldec para poder instalar el simulador consiste en que el usuario ingrese sus datos personales en un formulario de solicitud incluido en el sitio de descarga del mismo. Si el proceso fue exitoso, el sitio web mostrará un mensaje agradeciendo su registración e informando que un enlace de descarga para el instalador del simulador está siendo enviado a la dirección de correo ingresada en el formulario. Una vez sucedido esto, el usuario accederá a su casilla de correo electrónico personal, en la cual debería encontrar un nuevo mensaje en la bandeja de entrada de parte de "Aldec University Program (university@aldec.com)" con el asunto "Aldec Active-HDL Student Edition Download Information", el cual incluirá enlaces de descarga para las cinco partes que constituyen el instalador, cada una de las cuales deberá ser descargada para poder instalar el simulador:

Tabla 6. Nombres de los archivos que se deben descargar para instalar el simulador, junto con su tamaño y la función que cada uno de ellos desempeña en el proceso de instalación.

Nombre	Función	Tamaño
Active-HDL-Student-Edition-2018_7z_sfx.exe	Autodescompresor	190 KB
Active-HDL-Student-Edition-2018_7z_sfx.7z.001	Parte 1	800 MB
Active-HDL-Student-Edition-2018_7z_sfx.7z.002	Parte 2	800 MB
Active-HDL-Student-Edition-2018_7z_sfx.7z.003	Parte 3	800 MB
Active-HDL-Student-Edition-2018_7z_sfx.7z.004	Parte 4	392 MB

Todos los archivos descargados deberán ser preservados en una carpeta única con su nombre y extensión originales, y además los enlaces de descarga expirarán veinticuatro horas a partir del instante en el cual el mensaje fue enviado. Por lo tanto, si el usuario no lograra

descargar los ficheros durante el tiempo permitido, debería repetir todo el proceso de registración desde el principio.

Una vez descargadas las cinco partes del simulador, ya no es necesario poseer una conexión a Internet para continuar con los pasos restantes del proceso de instalación. Ahora deberá ejecutarse el archivo "Active-HDL-Student-Edition-2018_7z_sfx.exe", un autoextractor encargado de descomprimir las otras cuatro partes y combinarlas en un único archivo final: el instalador propiamente dicho del simulador. Una vez abierto el descompresor, se le ofrecerá al usuario la posibilidad de seleccionar un directorio destino para la extracción de las partes y la creación del instalador. Una vez elegido, la descompresión se realizará automáticamente y finalizará sin mostrarle al usuario ningún tipo de aviso aun si el proceso fue exitoso; la única comprobación posible consiste en verificar la existencia de un nuevo archivo en el directorio seleccionado para la extracción, cuyo nombre debería ser "Active-HDL-Student-Edition-2018.exe" y su tamaño, 2,75 GB. Éste será el instalador definitivo para el simulador.

El próximo paso será ejecutar el instalador creado anteriormente para llevar a cabo la instalación definitiva del simulador. Antes de comenzar, el asistente de instalación le ofrecerá al usuario la posibilidad de seleccionar tanto el directorio destino para la instalación como la carpeta en la cual se ubicará el espacio de trabajo para almacenar los proyectos desarrollados durante el transcurso de la utilización del Active-HDL, recomendándose en esta oportunidad mantener en ambos casos las opciones por defecto ofrecidas por el asistente ("C:\Aldec\Active-HDL-Student-Edition" y "C:\My_Designs" respectivamente). También deberán elegirse los componentes del simulador que se desean incluir en la instalación, así como los tipos o extensiones de archivos que deberían ser asociados al Active-HDL; nuevamente se aconseja conservar todas las opciones seleccionadas por defecto.

Ahora el instalador llevará a cabo el proceso de instalación del simulador. El mismo se realiza de manera automática y demorará algunos minutos. Al finalizar, se le mostrará un mensaje al usuario informando la exitosa instalación del Active-HDL Student Edition y ofreciéndole las posibilidades de leer las últimas modificaciones realizadas a la versión más reciente del programa (versión 10.5) y de agregar a PATH el directorio de instalación del simulador (PATH es una variable de entorno de los sistemas operativos POSIX y los sistemas de Microsoft, en ella se especifican las rutas en las cuales el intérprete de comandos debe buscar los programas a ejecutar). Se recomienda mantener seleccionada únicamente esta última opción.

Si el usuario ha obedecido correctamente todas las instrucciones indicadas previamente, el simulador ya debería encontrarse instalado en el ordenador. La comprobación más fácil de la exitosa finalización del proceso consiste en acceder al Escritorio de la PC y verificar que se haya creado un nuevo acceso directo con el nombre "Aldec Active-HDL Student Edition". Éste archivo representa uno de los muchos caminos válidos que existen para comenzar a utilizar el simulador. No obstante, aquí es donde termina este tutorial, ya que tanto la configuración como el uso del Active-HDL variarán en función del diseño que el usuario desee desarrollar. Se ofrecerá más información sobre este apartado en capítulos posteriores, una vez que se haya descripto más detalladamente el diseño planificado para este proyecto en particular.

A continuación se incluyen capturas de algunos de los pasos más importantes que constituyen el proceso de instalación del simulador:

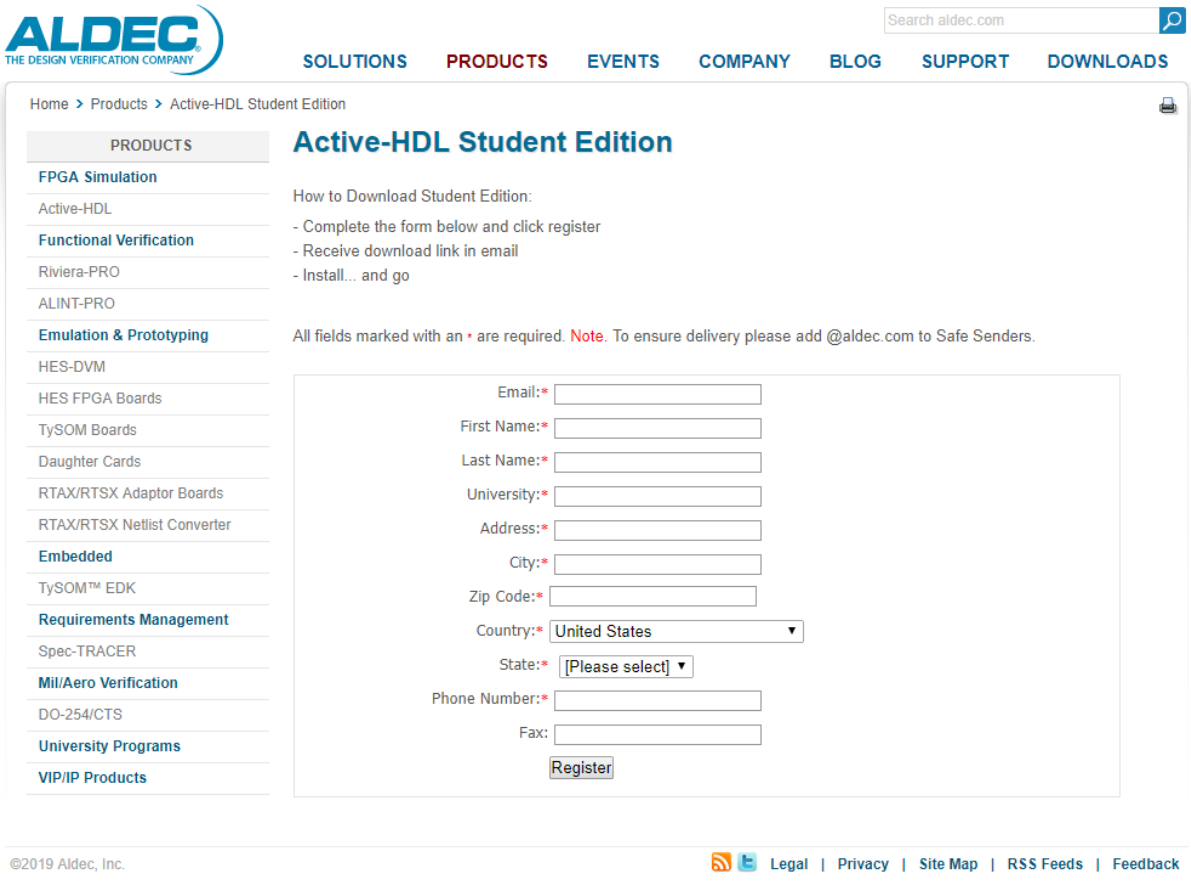

Figura 7. Formulario de registración para la descarga del Active-HDL Student Edition 2018 desde el sitio web oficial de Aldec (Zip Code = Código Postal).

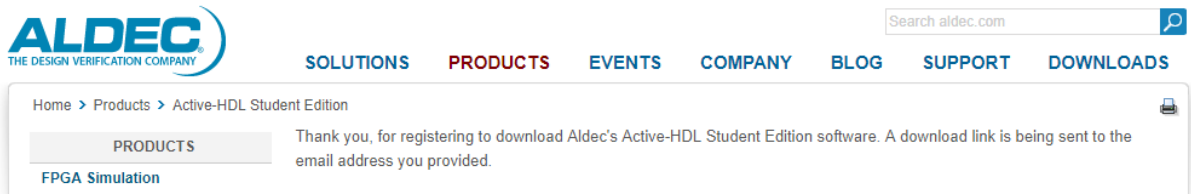

Figura 8. Mensaje informado al usuario por el sitio web oficial de Aldec luego de que el mismo haya completado y enviado exitosamente el formulario de la figura anterior.

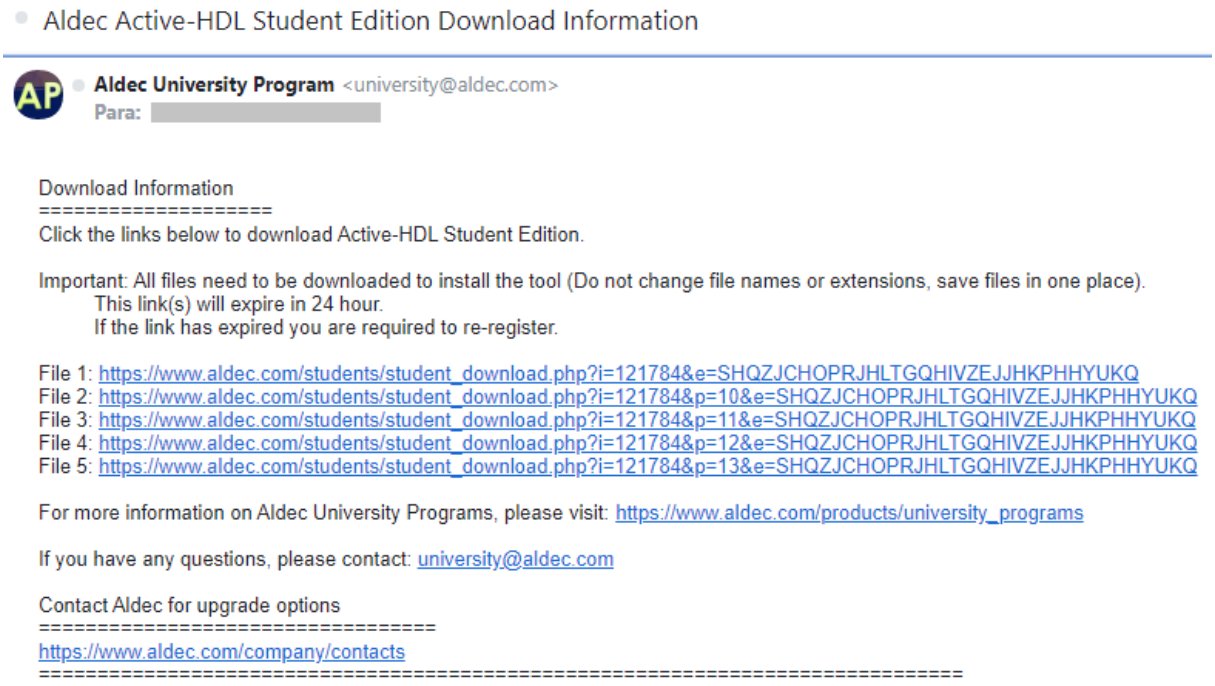

Figura 9. Mensaje enviado por Aldec al correo electrónico indicado por el usuario en su formulario de registración con los cinco enlaces de descarga para el instalador del simulador.

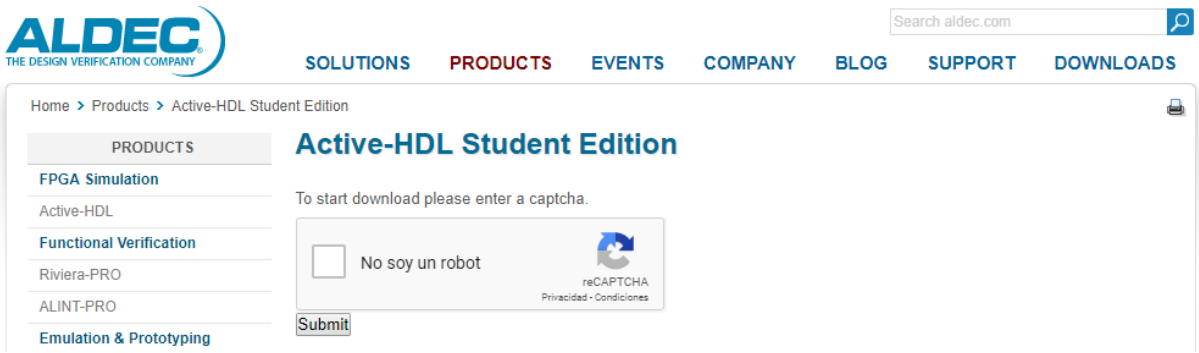

Figura 10. Pantalla de descarga para cada una de las cinco partes del instalador del simulador desde el sitio web oficial de Aldec.

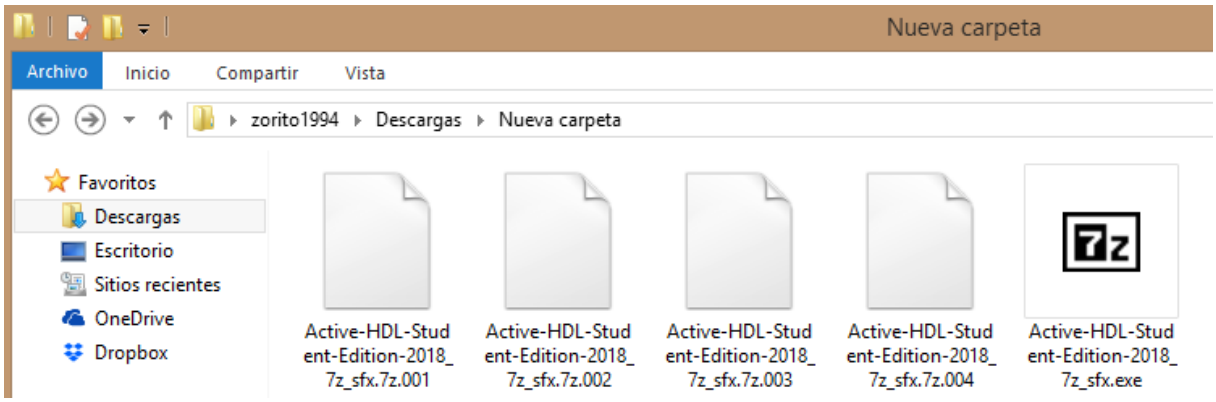

Figura 11. Archivos descargados desde los enlaces de la figura 9, necesarios para generar el instalador del Active-HDL.

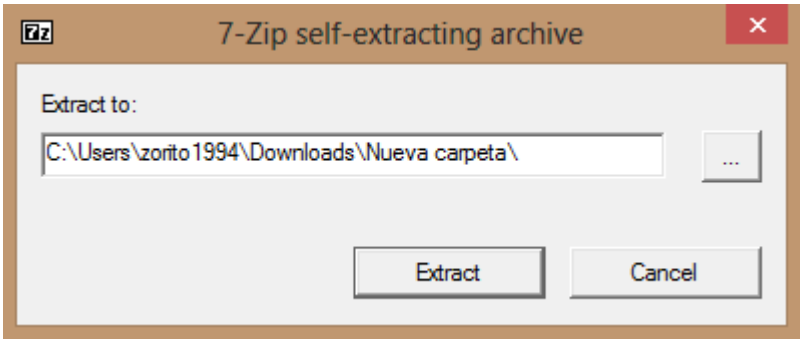

Figura 12. Selección del directorio destino para la generación del instalador del Active-HDL a partir de los archivos de la figura anterior.

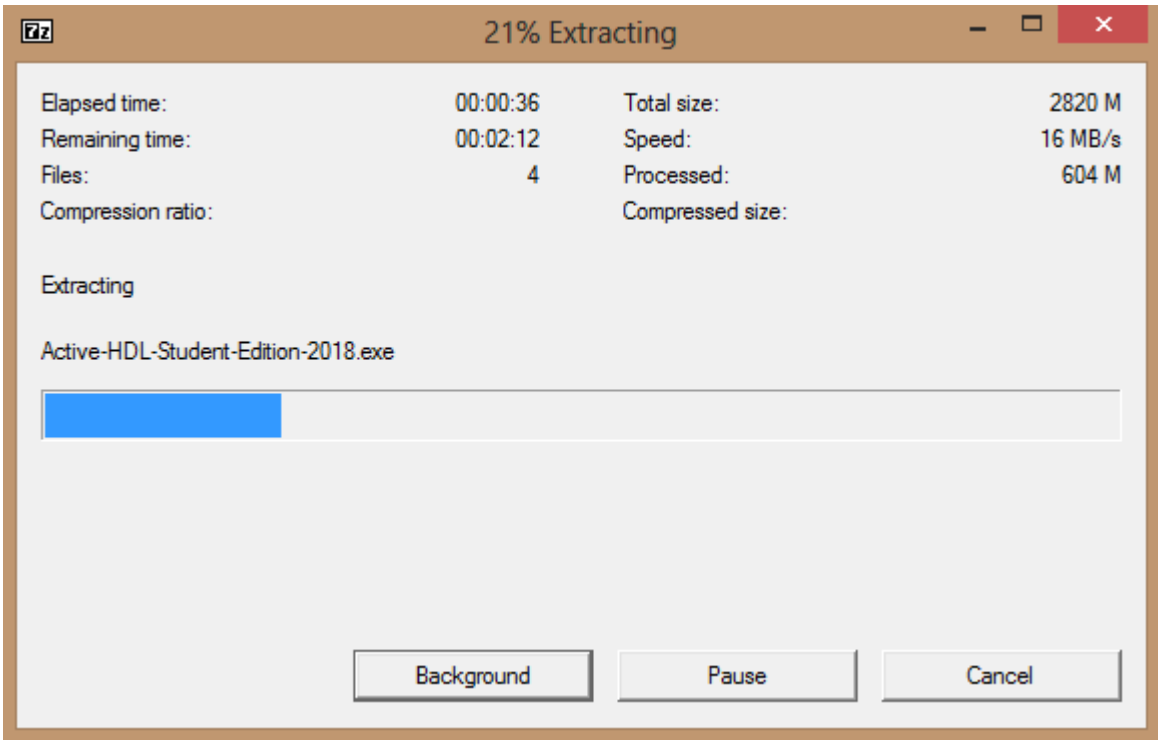

Figura 13. Ejecución del proceso de extracción de los archivos de la figura 11 para generar el instalador del simulador.

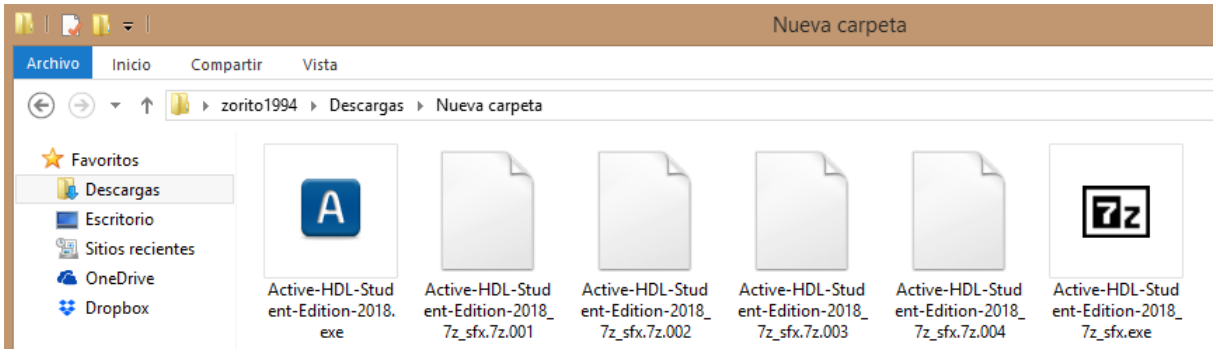

Figura 14. Generación exitosa del instalador del simulador (“Active-HDL-Student-Edition-2018.exe”) en la misma carpeta que los archivos descomprimidos.

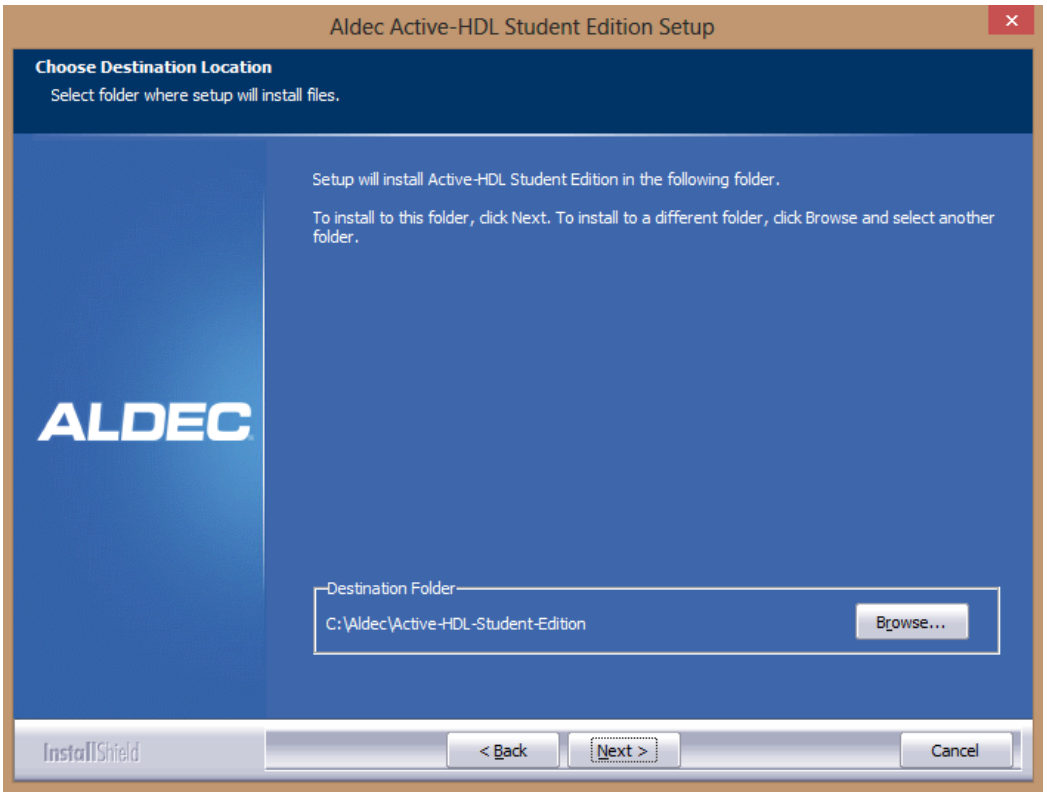

Figura 15. Selección del directorio destino para la instalación definitiva del Active-HDL mediante la ejecución del instalador de la figura anterior.

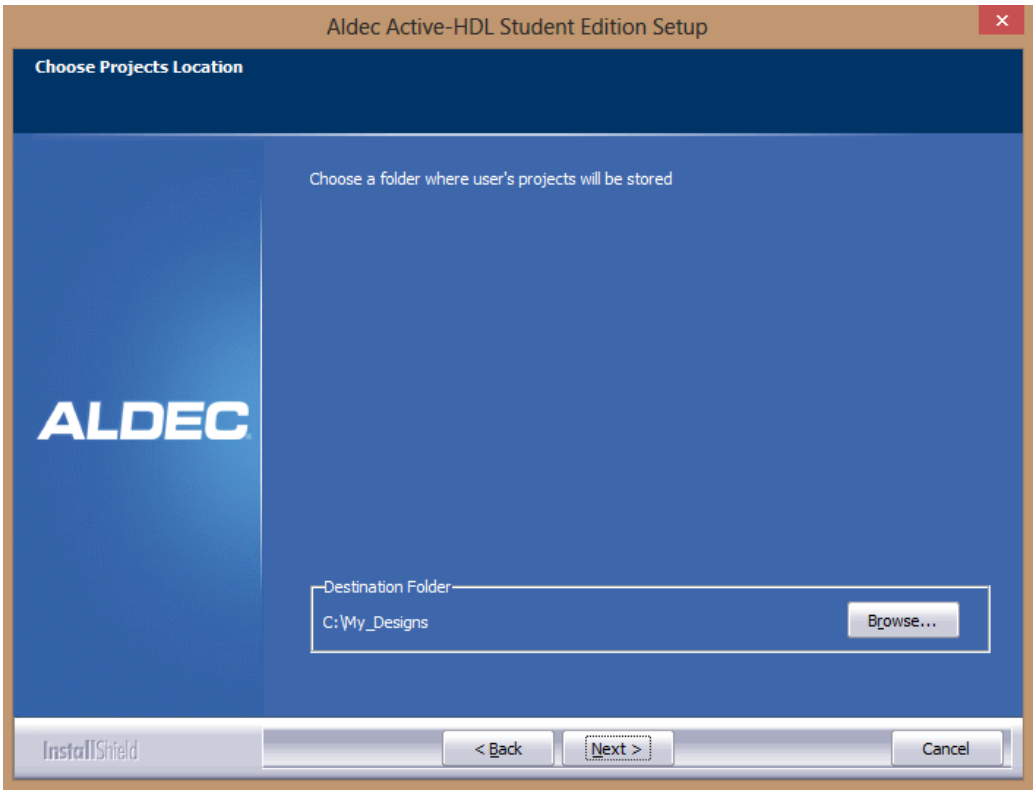

Figura 16. Selección del directorio para el espacio de trabajo destinado a almacenar los proyectos desarrollados durante el transcurso de la utilización del Active-HDL.

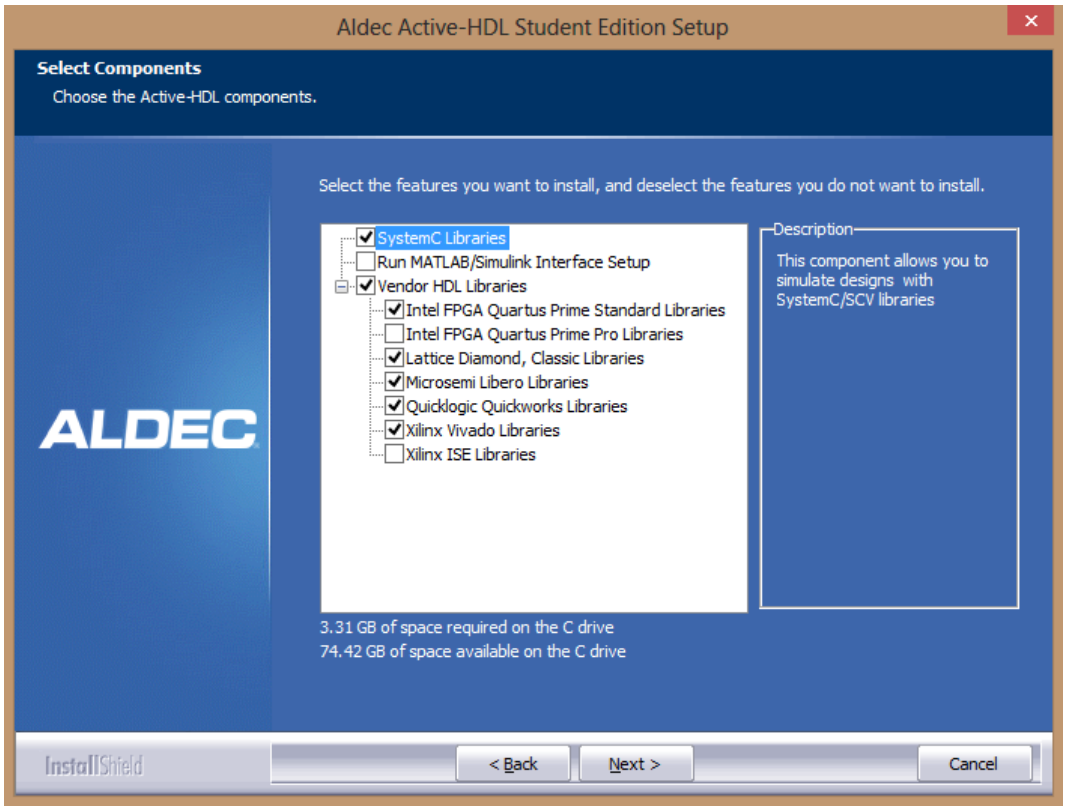

Figura 17. Selección de los componentes a incluir en la instalación del simulador.

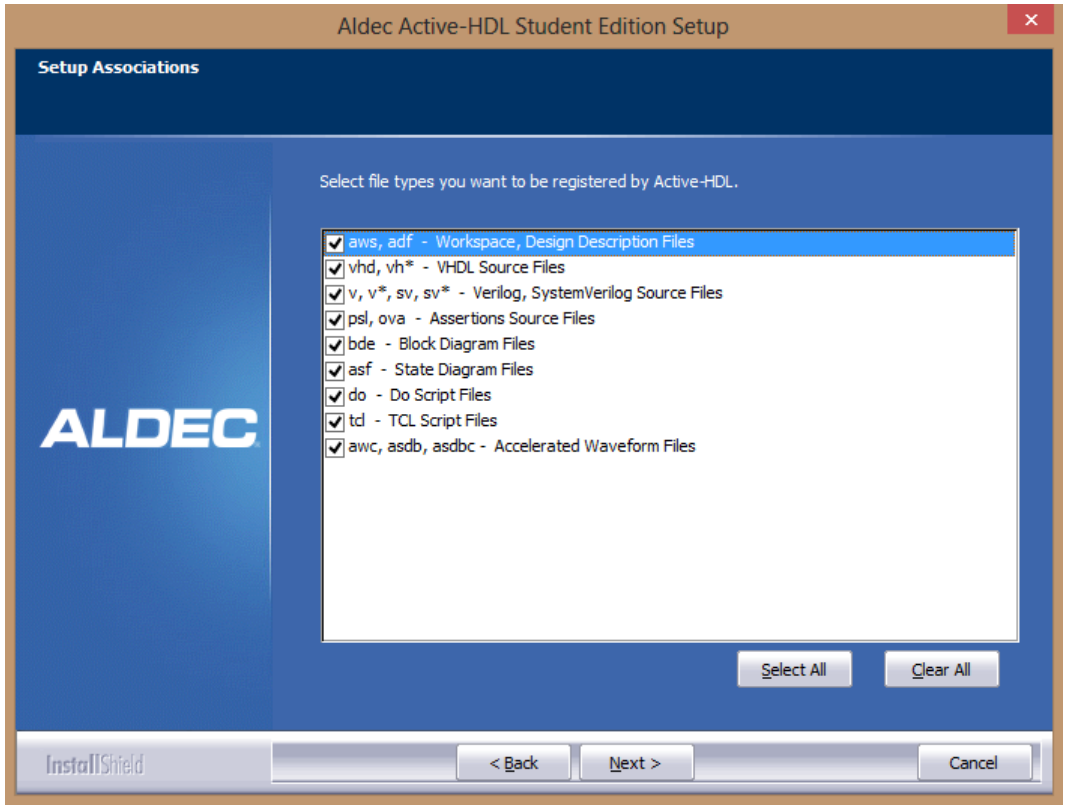

Figura 18. Selección de los tipos o extensiones de archivos que deberían ser asociados al Active-HDL.

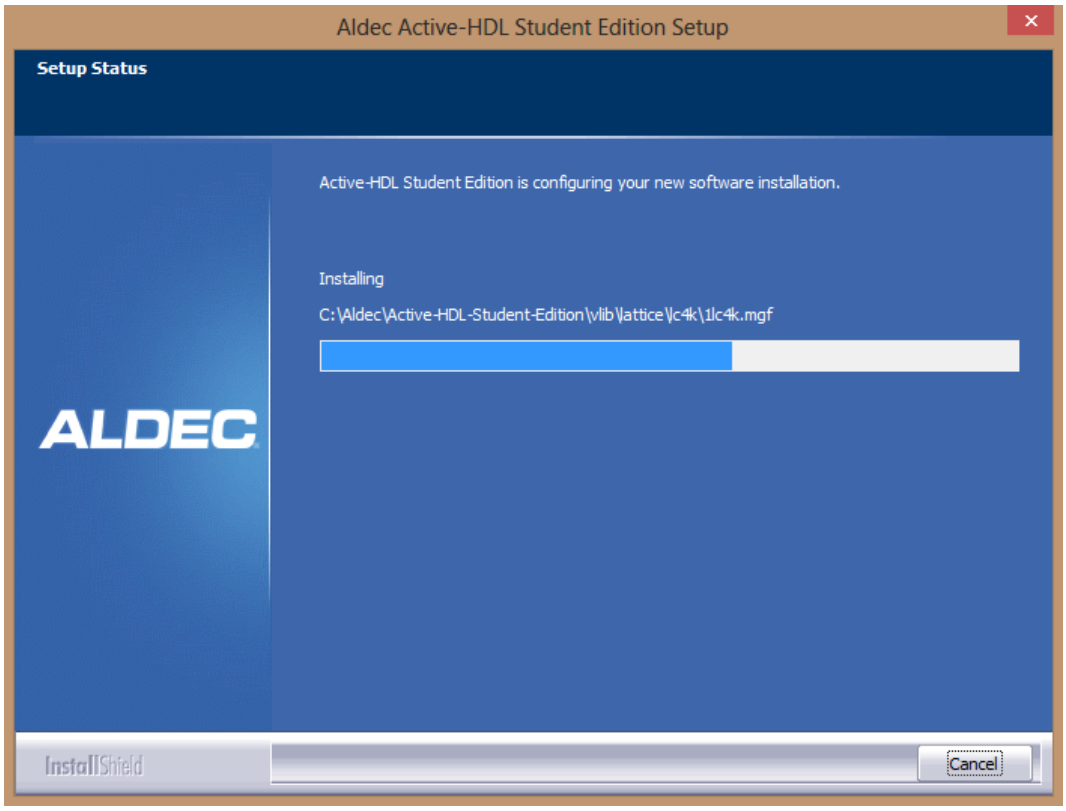

Figura 19. Ejecución del proceso de instalación definitiva del simulador.

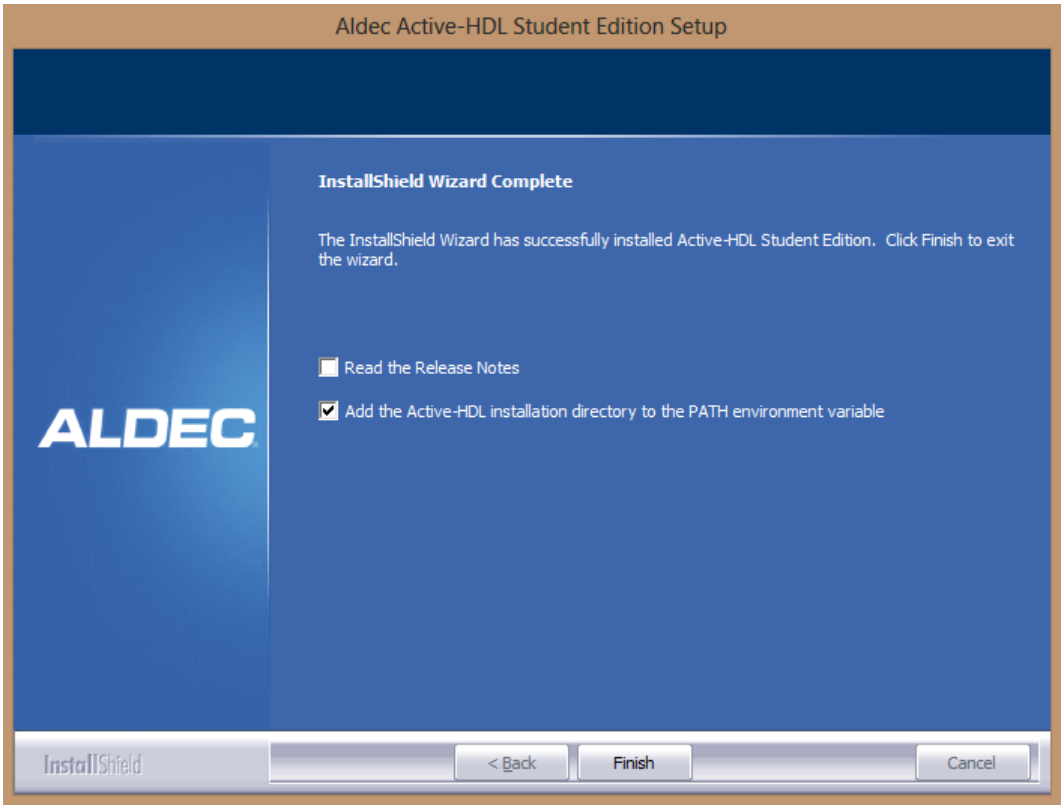

Figura 20. Selección de acciones a realizar una vez finalizado exitosamente el proceso de instalación del Active-HDL.

Figura 21. Acceso directo para la ejecución del Active-HDL, generado automáticamente en el Escritorio por el instalador del simulador una vez finalizada correctamente la instalación.

2.3. Diseño de la computadora original

Originalmente se había optado por tomar como punto de partida para el diseño del proyecto un código disponible para su descarga en la biblioteca digital oficial VHDL de la Universidad de Hamburgo, ubicada en el sitio web https://tams.informatik.uni-hamburg.de/research/vlsi/vhdl/index.php?content=06-models, el cual describe un modelo comportamental simple e incompleto del microprocesador Intel 80386.

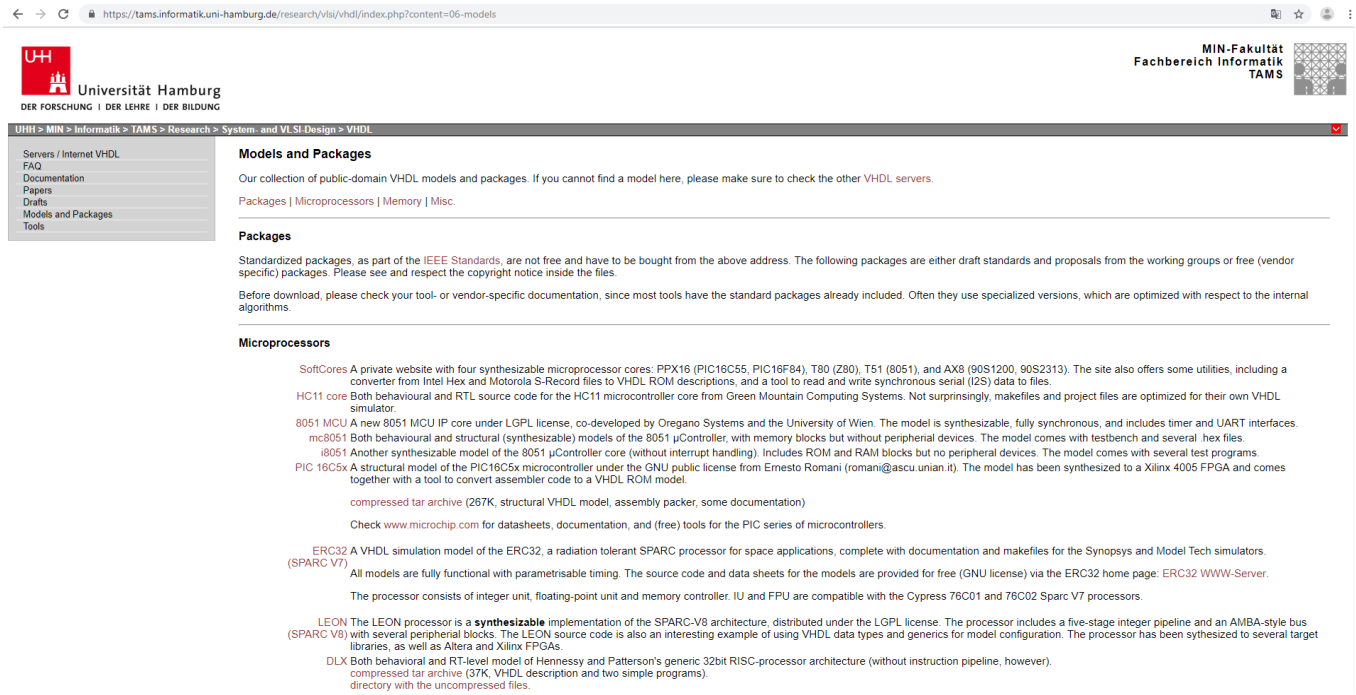

Figura 22. Biblioteca digital oficial VHDL de la Universidad de Hamburgo.

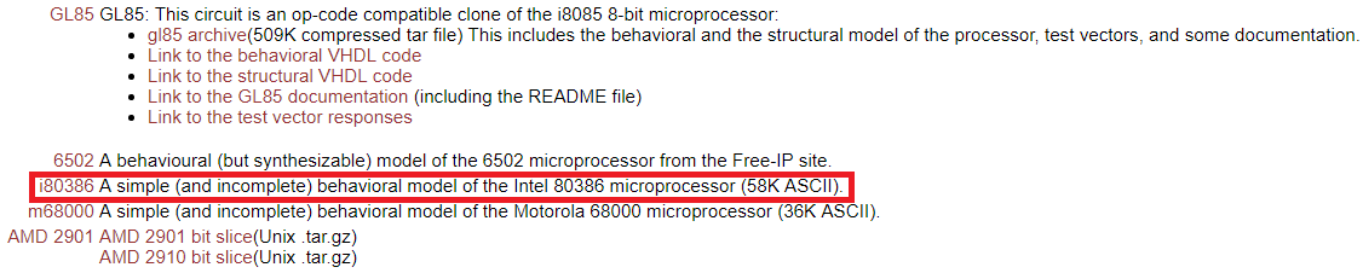

Figura 23. Enlace de descarga para el código VHDL del modelo del microprocesador originalmente seleccionado para comenzar el diseño de este proyecto.

Al no haberse podido localizar ningún tipo de manual que definiera y explicara las características principales de este modelo original, se decidió llevar a cabo un proceso de ingeniería inversa para, a partir de un análisis minucioso y exhaustivo del código, establecer cuáles habían sido la idea y el objetivo originales del diseñador y, por lo tanto, el grado de avance que tenía el desarrollo de la CPU hasta la fecha. A partir de dicho estudio, se logró extraer la información que se detalla a continuación (sólo se mencionan las funcionalidades del procesador que ya se encuentran implementadas):

Tabla 24. Atributos generales del diseño de la computadora original.

Tipo de arquitectura	CISC
Ciclos de ejecución	Cada instrucción puede requerir uno o dos ciclos de reloj para ejecutarse en función del tamaño de su operando
Interfaz de Entrada/Salida	E/S independiente
Formato y longitud de las instrucciones	Instrucciones de formato y longitud variables: uno, dos, tres o cinco bytes en función del tamaño de su operando
Frecuencia del reloj	A determinar por el usuario
Bus de datos	32 bits
Bus de direcciones	32 bits
Modos de direccionamiento	Inmediato
	Directo de Registro
	Directo de Memoria
	Indirecto con Registro
	Indirecto con Desplazamiento Relativo al IP
Registros de uso general	EAX
	EBX
	ECX
	EDX
Almacenamiento en memoria/cola	Little Endian

Tabla 25. Repertorio de instrucciones de la CPU original.

Tipo de instrucción	Instrucción	Comentario	Operación	Observación
Transferencia de Datos	MOV *dest, fuente*	Copia *fuente* en *dest*	(*dest*) ← (*fuente*)	1
	IN *dest, fuente*	Carga el valor en el puerto *fuente* en *dest*	(*dest*) ← (*fuente*)	2
	OUT *dest, fuente*	Carga en el puerto *dest* el valor en *fuente*	(*dest*) ← (*fuente*)	3
Aritmética	INC *dest*	Incrementa *dest* en una unidad	(*dest*) ← (*dest*) + 1	4
Transferencia de Control	JMP *desp*	Salta incondicionalmente a *desp* direcciones de la próxima instrucción	(IP) ← (IP) + *desp*	5
Control	NOP	Operación nula (no realiza ninguna acción)		

Observaciones:

1. Las posibilidades para *dest/fuente* son: *reg/mem*, *reg/op.inm*, *mem/reg*. *mem* sólo puede ser [BX], siendo (BX) una dirección de memoria (direccionamiento indirecto).
2. Las posibilidades para *dest*/*fuente* son: AL/*mem*, AL/DX. *mem* debe ser una dirección entre 0 y 255. Puede ser un operando inmediato o una etiqueta.
3. Las posibilidades para dest/fuente son: *mem*/AL, DX/AL. *mem* debe ser una dirección entre 0 y 255. Puede ser un operando inmediato o una etiqueta.
4. *dest* sólo puede ser *reg*.
5. *desp* sólo puede ser un operando inmediato entre -2.147.483.648 y 2.147.483.647 (valor con signo de 32 bits).

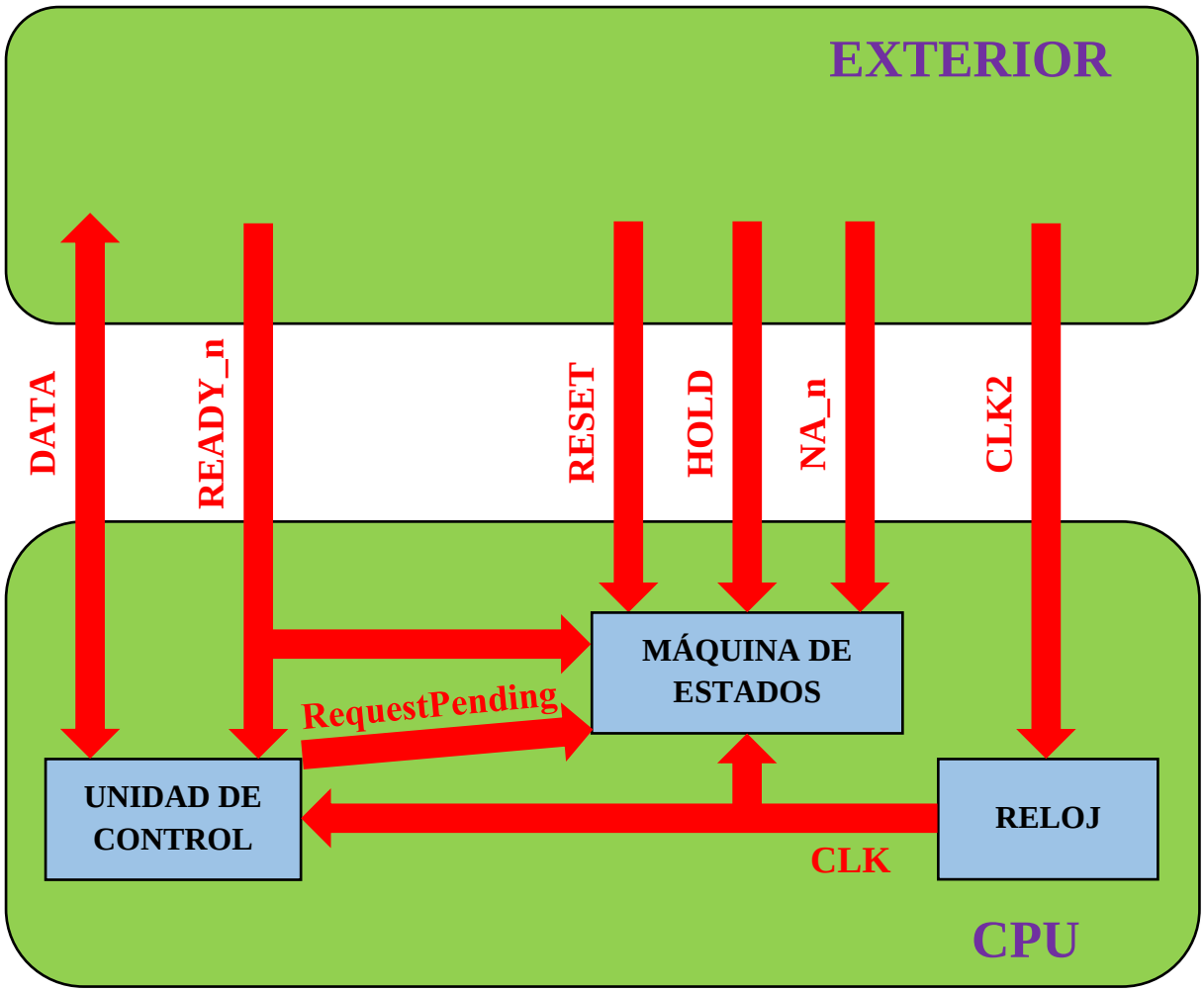

Figura 26. Diseño de los componentes y señales y su interacción para la CPU original.

Observaciones:

- Puertos de entrada/salida:
  - ❖ Data: permite al usuario indicarle al procesador el código de operación de la instrucción a ejecutar junto con el operando necesario para llevarla a cabo. Dicho operando variará en función de la instrucción ingresada (dirección de memoria a leer/escribir, valor inmediato, dirección de salto, etc.). También le informa al usuario el dato resultante una vez ejecutada la instrucción (dato leído/escrito, resultado de la operación aritmética, dirección de salto efectiva, etc.).
- Puertos de entrada:
  - ❖ CLK2: señal de reloj del usuario. La frecuencia de esta señal será el doble de aquélla correspondiente al reloj interno del procesador. Esta señal se activa por flanco ascendente.

- NA_n: permite al usuario segmentar el cauce del procesador. Esta señal se activa con un nivel bajo.
- READY_n: cumple una función similar al puerto CLK2, habilitando la ejecución de la instrucción que se encuentre pendiente de ser atendida siempre y cuando su respectivo dato haya sido cargado en el puerto Data. Esta señal se activa por flanco descendente.
- HOLD: señal de espera de reconocimiento. Esta señal se activa con un nivel alto.
- RESET: permite al usuario reiniciar el procesador. Esta señal se activa por flanco ascendente, pero sólo se desactiva por flanco descendente.

- Señales internas:
  - RequestPending: su valor es '1' (Pending) si existe alguna instrucción pendiente de ser atendida y '0' (NotPending) en caso contrario.
  - CLK: reloj interno del procesador. Su frecuencia es la mitad de aquélla correspondiente a la señal de reloj del usuario. Esta señal se activa por flanco ascendente.

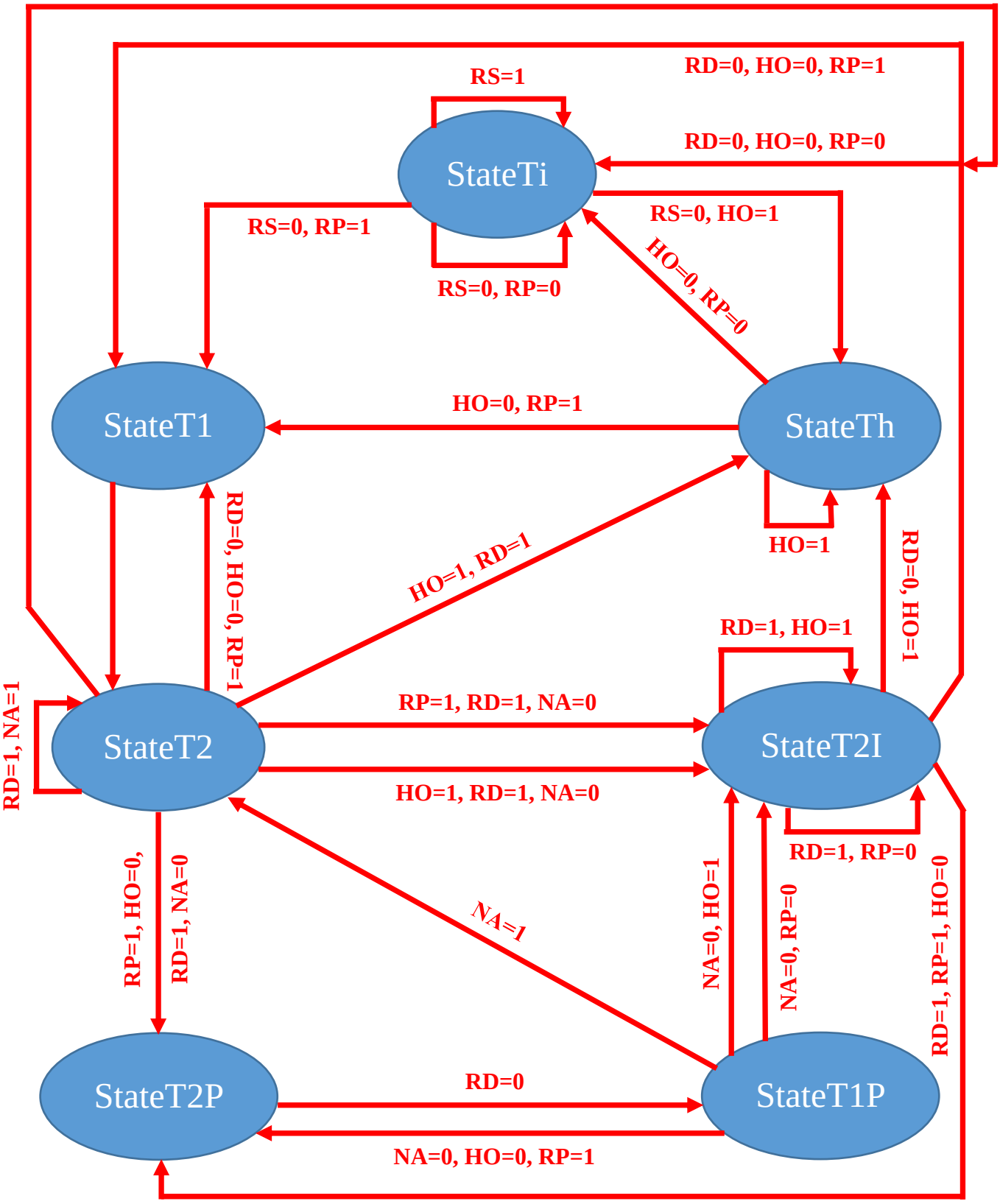

Figura 27. Funcionamiento de la CPU representado a través de una máquina de estados (RS = RESET; RP = RequestPending; HO = HOLD; RD = READY_n; NA = NA_n).

Observaciones:

- StateTi: Estado de Reinicio (Reset State). Todos los registros del procesador pasan a poseer valores de alta impedancia ('Z') y no se realiza ninguna otra acción.
- StateT1: Primer estado de un cauce no segmentado. Aquí se lleva a cabo la totalidad de la ejecución de cada instrucción.
- StateT2: Estado de un cauce no segmentado durante el cual el procesador permanece ocioso.
- StateT1P: Primer estado de un cauce segmentado.
- StateT2P: Estado subsecuente de un cauce segmentado.
- StateT2I: Estado subsecuente de un cauce potencialmente segmentado.

Para cerrar esta sección del informe, se expondrán las conclusiones finales obtenidas sobre el estado actual del proyecto original a partir de las tablas y figuras anteriores:

- Como habrá podido notarse, el diseño original sólo contempla la funcionalidad de la CPU, sin haber desarrollado aún ningún tipo de planificación general para el resto de los componentes de la PC con los cuales debería interactuar durante el desempeño de sus tareas. Es decir, tanto la memoria principal como los periféricos de entrada/salida no existen en absoluto en el proyecto llevado a cabo hasta la fecha, por lo cual cualquier tipo de instrucción que requiera realizar tanto accesos a memoria (transferencias de datos desde o hacia memoria, operaciones aritmético/lógicas o transferencias de control con direccionamiento directo de memoria o indirecto con registro, etc.) como operaciones de entrada/salida (lectura/escritura sobre puertos de periféricos de entrada/salida para realizar impresiones o activar barras de leds o microconmutadores, similar al MSX88, o bien leer datos ingresados por el usuario a través de la pantalla de una terminal o escribir información en ella para que pueda observarlos, al igual que en el WinMIPS64) no podrá ser actualmente implementada por el programador en el repertorio de instrucciones de la CPU hasta no haber descrito previamente de alguna manera el comportamiento de dichos componentes tanto internamente como formando parte de una PC que los contenga y vincule y en la cual deberán transmitir y recibir información del procesador ya desarrollado según corresponda.
- La ausencia de un ensamblador compatible con este procesador implica que resulta imposible para un programador de un nivel de abstracción más alto escribir programas en

lenguaje Assembler que puedan ser interpretados y ejecutados por esta CPU, por lo cual la única alternativa viable sería estudiar detalladamente el código de operación (opcode) pensado por el diseñador para cada instrucción del repertorio de manera tal que la misma resulte inteligible para la unidad de control encargada de ejecutarla. Dicho en otras palabras, para utilizar este procesador habría que escribir instrucciones directamente en código máquina (binario), sin ningún tipo de abstracción que le facilite la tarea al programador, por lo cual la labor de programar sería considerablemente más tediosa y complicada. Además, al no poseer el diseño ningún banco de pruebas (testbench), no se cuenta con un ejemplo claro tanto de la configuración previa del procesador requerida para ejecutar una instrucción como el formato de esta última y el método adecuado para que el usuario/programador pueda transmitirla a la CPU.

- A pesar de que tanto la máquina de estados de la figura 27 como la presencia de la señal NA_n sugieren la existencia de una segmentación de dos etapas ya implementada y plenamente operativa en el cauce del procesador, esto en realidad no es así, ya que la activación de la segmentación sólo implica un cambio en los estados de la máquina pero no en la ejecución propiamente dicha de las instrucciones. Para empezar, no se ha implementado ninguna división en etapas para el proceso de ejecución de las mismas, por lo cual resultará imposible poseer más de una instrucción ejecutándose simultáneamente: cada instrucción se ejecuta en una única etapa que requiere un ciclo de reloj para completar sus acciones, por lo cual cada una deberá esperar a que finalice la anterior para comenzar a ejecutarse. En otras palabras, la ejecución de las instrucciones por parte de este procesador es estrictamente secuencial. Se concluye en este apartado que el diseño desarrollado hasta el momento propone a través de la máquina de estados una “interfaz” o “idea” para incluir la alternativa de segmentar el cauce de ejecución de las instrucciones en dos etapas para reducir el tiempo de ejecución total del programa permitiendo la ejecución de dos instrucciones al mismo tiempo. Sin embargo, la implementación de esta interfaz aún se encuentra pendiente, tarea que deberá ser llevada a cabo por el diseñador que continúe con este proyecto de manera tal que la máquina de estados propuesta refleje genuinamente en todo momento la situación actual de la CPU.
- La descripción seleccionada para el diseño original de la CPU es la comportamental: en ella se define la forma en la cual se comporta el circuito digital, teniendo en cuenta sólo las características del mismo respecto al comportamiento de las entradas y las salidas. Ésta es la forma que más se parece a los lenguajes de software ya que la descripción puede ser secuencial, además de combinar características concurrentes. No obstante, este método no

es el más recomendable cuando el diseño digital se vuelve complejo o está conformado por múltiples bloques de hardware como sería el caso de este procesador: la alternativa más adecuada podría ser una descripción estructural, en la cual se define el circuito a partir de instancias de componentes, representando cada componente a un bloque de hardware distinto. Dichas instancias conforman un diseño de jerarquías, al conectar los puertos de las mismas con las señales internas del circuito o bien con puertos del circuito de jerarquía superior o inferior. Al encontrarse actualmente la CPU diseñada en un estado tan incompleto es posible que la descripción comportamental aún se adapte al funcionamiento esperado por parte del procesador, pero a medida que este último se vuelva más complejo y se le incorporen más funcionalidades, se tornará cada vez más complicada la tarea de programar código adicional manteniendo esta forma de descripción. Por lo tanto, se sugiere replantear el diseño de la CPU con la mayor prontitud posible de manera tal que su descripción sea más estructural, analizando las funciones esperadas del procesador y procurando asignar cada una a un componente individual, el cual debería ser tan independiente como sea posible de la CPU en sí y de los restantes componentes. Esto es lo que se conoce en la programación de alto nivel como "modularización", pero aquí también será sumamente útil para evitar que el diseñador se vea abrumado y superado por la complejidad del diseño a desarrollar.

- El repertorio de instrucciones ofrecido por el diseñador se encuentra actualmente muy limitado: además de las restricciones ya mencionadas respecto a las operaciones de entrada/salida y que requieran acceder a memoria, se puede observar que aún resta implementar casi la totalidad de las instrucciones aritmético-lógicas (suma, resta, multiplicación, división, or, and, not, etc.), de desplazamiento de bits (a izquierda o derecha) y de transferencia de control (salto condicional). De todas maneras, este no sería un inconveniente relevante ya que la implementación e inclusión de las instrucciones faltantes no debería en un principio implicar modificaciones radicales en el diseño original: simplemente habría que agregar los códigos de operación asociados a las nuevas instrucciones a la unidad de control de la CPU para que pueda interpretarlos y determinar las próximas acciones a realizar.
- La ausencia en este diseño de una memoria principal a la cual la CPU pueda acceder para leer o escribir datos obviamente conlleva que la sección de memoria conocida como pila tampoco se encuentre complementada ni mucho menos implementada, al igual que su manejo y acceso por parte de instrucciones específicas para apilar o desapilar datos de ella. De igual manera, hasta el momento no ha sido tenida en cuenta la posibilidad de que

el procesador reciba pedidos de interrupción tanto por hardware como por software, por lo cual todas las instrucciones asociadas a interrupciones tampoco han sido implementadas. A diferencia de la situación mencionada en el párrafo anterior, la situación aquí se agrava porque no sólo es necesario incluir los nuevos códigos de operación en la unidad de control sino que además tanto el manejo de la pila como de las interrupciones involucra una lógica adicional que deberá ser planificada y diseñada detenidamente antes de proceder a implementar las instrucciones propiamente dichas.

- Como consecuencia de la metodología seleccionada para la descripción del diseño de la CPU (comportamental), no se encuentra específica ni individualmente definida una unidad aritmético-lógica (ALU) para realizar operaciones aritmético-lógicas ni un banco de registros integrado en el procesador para leer y escribir datos temporalmente en él a una mayor velocidad que si se viera obligado en cada operación a acceder a la memoria principal de la PC. No obstante, esto no significa que sus funcionalidades no estén ya implementadas en este diseño, considerando que actualmente hay definidas algunas operaciones de suma junto con algunos registros de propósito general que son accedidos directamente durante la ejecución de las instrucciones que así lo requieran. El principal inconveniente que aquí se presenta consiste en que técnicamente no se está respetando la arquitectura von Neumann, el pilar fundamental para describir los diseños de todas las computadoras actuales. En dicha arquitectura, la CPU debe poseer tres secciones claramente diferenciables: la unidad de control, la ALU y el banco de registros. En el código desarrollado para el procesador de este proyecto original podría argumentarse que se ha considerado la existencia de la unidad de control, encargada de decodificar el código de operación de la instrucción actual y, a partir de él, determinar las acciones a realizar para ejecutarla. Aun así, este diseño determina que la ejecución de cada instrucción debe llevarse a cabo en una única etapa y que la unidad de control deberá llevar a cabo por sí sola la totalidad de las tareas requeridas: buscar la siguiente instrucción, decodificarla e interpretarla, ejecutarla y almacenarla en memoria o en un registro según corresponda. Esto refleja la problemática mencionada anteriormente respecto a la descripción comportamental y la necesidad de estructurar el diseño y modularizar el código: a este ritmo, si se desean incorporar nuevas funcionalidades la unidad de control se complejizará hasta tal punto que se volverá imposible de estudiar, manejar y mejorar para cualquier programador, por lo cual se recomienda como primera acción asignar a la unidad de control sólo las funciones que le corresponde desempeñar y definir una ALU y un banco

de registros separados para realizar el resto de las tareas, tal y como lo determina la arquitectura von Neumann.

- Teniendo en cuenta el objetivo inicialmente propuesto en la introducción de este informe, se puede apreciar que existen algunas funcionalidades planteadas en el diseño original que resultan superficiales y complejizan innecesariamente el funcionamiento del procesador, por lo que podrían ser eliminadas sin mayores inconvenientes para simplificar el modelo y focalizarlo en las tareas que sí se esperan que el mismo pueda cumplir. Se debe recordar que en este proyecto se está realizando una simulación, por lo tanto no necesariamente el diseño debe representar fielmente la CPU: alcanza con que se contemplen los aspectos indispensables para el estudio de la misma en función de las metas estipuladas al comenzar el proceso de diseño. Por ejemplo, aquí no sería necesario que el procesador posea la capacidad de ser reiniciado ni que aguarde una señal de reconocimiento por parte del usuario ya que, aunque estas prestaciones asemejen el comportamiento de la CPU al que tendría en el mundo real, no tienen ninguna relación con el objetivo determinado para este proyecto en particular.
- Para finalizar, se describirá en términos generales el ciclo de ejecución definido en este diseño para cada instrucción:
    - En primer lugar, se activa la señal “RequestPending” para indicar que existe una instrucción pendiente de ser atendida.
    - Luego, permanece ociosa hasta el próximo flanco descendente en la señal “READY_n”. Una vez producido éste, desactiva la señal “RequestPending” y aguarda hasta el siguiente flanco descendente que ocurra en la señal “CLK”.
    - Una vez producido el flanco descendente en el reloj interno del procesador, se recibe en la señal “Data” la información enviada desde el exterior correspondiente a la próxima instrucción a ejecutar (código de operación y operando) y se la almacena en la cola interna de la CPU (señal “InstQueue”).
        - Recordar que, al no encontrarse implementado en este modelo la memoria principal de la PC, “exterior” es una expresión ambigua cuya representación varía en función de la intención del diseñador: puede ser la memoria principal ya efectivamente definida y descripta en el modelo o bien directamente el usuario final quien le esté transmitiendo directamente instrucciones individuales al procesador para que éste las ejecute.
        - La cola interna a la que se hizo referencia anteriormente consiste en una estructura auxiliar que utiliza temporalmente el procesador para almacenar

temporalmente la instrucción recibida del usuario hasta el momento de ejecutarla. Aunque esta "cola" auxiliar integrada en la CPU no debería estar incluida en el diseño bajo ningún punto de vista ya que las únicas unidades de almacenamiento permitidas en el procesador son los registros, podría justificarse su existencia en la necesidad de reemplazar la memoria principal ausente en este modelo por una estructura de almacenamiento que desempeñe una función similar. Cada dirección en la cola puede almacenar un byte de información.

- Los formatos adecuados para enviar una instrucción a esta CPU a fin de que esta última pueda ejecutarla son los siguientes:

			Código de operación
Byte 3 (bits 31-24)	Byte 2 (bits 23-16)	Byte 1 (bits 15-8)	Byte 0 (bits 7-0)

Figura 28. Formato de instrucción sin operandos para la CPU original. Adecuado para instrucciones de control (NOP).

		Operando	**Código de operación**
Byte 3 (bits 31-24)	Byte 2 (bits 23-16)	Byte 1 (bits 15-8)	Byte 0 (bits 7-0)

Figura 29. Formato de instrucción con un operando de 8 bits para la CPU original. Adecuado para instrucciones de transferencia de datos y aritméticas.

	Operando (byte 1)	**Operando (byte 0)**	**Código de operación**
Byte 3 (bits 31-24)	Byte 2 (bits 23-16)	Byte 1 (bits 15-8)	Byte 0 (bits 7-0)

Figura 30. Formato de instrucción con un operando de 16 bits para la CPU original. Adecuado para instrucciones de transferencia de datos y aritméticas.

Operando (byte 2)	Operando (byte 1)	Operando (byte 0)	Código de operación
Byte 3 (bits 31-24)	Byte 2 (bits 23-16)	Byte 1 (bits 15-8)	Byte 0 (bits 7-0)

Figura 31. Formato de instrucción con un operando de 32 bits para la CPU original (primera transmisión). Adecuado para instrucciones de transferencia de datos, aritméticas y de transferencia de control.

- ✓ Nota: Si el operando es de 32 bits, se requerirán dos transmisiones para enviar la totalidad de la instrucción (en el primer paquete se podrán transmitir únicamente los tres bytes menos significativos del operando). A continuación se define el formato para realizar la segunda transmisión (sólo restaría para enviar el byte más significativo del operando):

			Operando (byte 3)
Byte 3 (bits 31-24)	Byte 2 (bits 23-16)	Byte 1 (bits 15-8)	Byte 0 (bits 7-0)

Figura 32. Formato de instrucción con un operando de 32 bits para la CPU original (segunda transmisión).

Dirección	Contenido
1	Código de operación
2	Operando (byte 0)
3	Operando (byte 1)
4	Operando (byte 2)
5	Operando (byte 3)
...	...

Figura 33. Instrucción que incluye un operando de 32 bits, almacenada en la cola interna de la CPU (formato de almacenamiento: little endian).

- A continuación se realiza la decodificación de la próxima instrucción a ejecutar: se obtiene de la cola de la CPU su código de operación (aunque sin desencolarlo ni borrarlo de la misma) y se llevan a cabo las acciones que se requieran para ejecutarlo según corresponda: lectura/escritura en registros, operaciones aritméticas, operaciones de entrada/salida (las cuales implicarán leer o escribir nuevamente sobre la señal "Data"), etc. Si es necesario, se realizará un segundo acceso a la cola para leer el operando asociado a la instrucción actual.
- En el caso excepcional de que el operando asociado a la instrucción sea de 32 bits, se deberá diferir la ejecución de la instrucción hasta el próximo ciclo de reloj, durante el cual se recibirá el byte restante del operando y se ejecutará efectivamente la instrucción demorada.
- Antes de proceder a la próxima sección del informe, se representará en la siguiente figura un ejemplo teórico sobre el progreso esperado a través del tiempo para la ejecución de algunas de las instrucciones que conforman el repertorio ya implementado del procesador para que el lector pueda ver aplicados los conceptos explicados en los incisos anteriores y así terminar de comprender el ciclo de ejecución de esta CPU y la variación en la cantidad de ciclos de reloj necesarios para ejecutar cada instrucción en función del tamaño de su operando (cada columna representa el transcurso de un ciclo de reloj):

Tiempo / **Instrucción**	t1	t2	t3	t4	t5	t6	t7	t8	t9	t10	...
MOV EAX, 00000F00H	■	■									
INC EAX			■								
MOV [EBX], EAX				■							
MOV EBX, 01020304H					■	■					
IN AL, 0ABH							■				

Figura 34. Progreso de la ejecución de algunas instrucciones en la CPU original a través del tiempo.

3. TDA 1819

A partir de todas las consideraciones previamente mencionadas, se ha determinado que el diseño inicialmente seleccionado no era apto para los objetivos planteados para este proyecto. Por lo tanto, se ha optado por la alternativa de realizar un nuevo modelo absolutamente original que no obstante conservara algunos de los aspectos de la arquitectura inicial, como el tamaño del bus de datos, el formato de almacenamiento en memoria y la variabilidad en el formato y el tamaño de las instrucciones, pero con la drástica determinación de cambiar el tipo de la arquitectura de CISC a RISC. Como cabría esperarse, al mantener algunas características del diseño CISC original el resultado final no será un procesador RISC puro sino un "híbrido", aunque como se demostrará a continuación definitivamente presentará más atributos de una arquitectura RISC que de una CISC.

La razón fundamental para llevar a cabo esta migración en el tipo de arquitectura se encuentra basada en la motivación a futuro de implementar una arquitectura "multi-core" a partir de esta CPU, donde cada núcleo pueda ejecutar instrucciones en forma paralela con el resto de los núcleos del conjunto. Un primer paso orientado hacia este paralelismo podría ser la incorporación de una segmentación para el cauce del procesador, a través de la cual se divide el ciclo de ejecución de cada instrucción en etapas con funciones específicas e independientes de manera tal de poder ejecutar múltiples instrucciones al mismo tiempo, aunque cada una en una etapa distinta. Idealmente, se podrían tener tantas instrucciones en ejecución como etapas en las cuales se divida el cauce. Sin embargo, existen diversas situaciones en las cuales dicho cauce se verá interrumpido debido a atascos que se presenten debido a conflictos por los recursos de hardware (memoria de datos, registro de propósito general), los datos (RAW, WAR, WAW) o la próxima instrucción a ejecutar (dependencia de control). Aunque esta técnica en términos estrictos no implique un paralelismo en la ejecución de las instrucciones (ninguna de las etapas podrá ser realizada simultáneamente), podrá comprobarse que su aplicación reducirá el tiempo de ejecución total del programa, por lo que siempre es recomendable que cualquier procesador posea incorporado este mecanismo.

Como se recordará, la ejecución de las instrucciones en el diseño original de la CPU era estrictamente secuencial, sin que existiera ningún tipo de división de la misma en etapas para implementar la segmentación en el cauce. Ahora bien, al intentar planificar para su arquitectura CISC la manera de subdividir el proceso de ejecución de cada instrucción en tareas más sencillas e independientes entre sí se determinó, en primer lugar, que la implementación más tradicional para la segmentación del cauce (cinco etapas: búsqueda,

decodificación, ejecución, acceso a memoria y almacenamiento en registro) era imposible de aplicar en este caso debido a la inexistencia de una instrucción específica para acceder a la memoria de datos ya sea para leer o escribir un valor, lo cual significa que básicamente cualquier tipo de instrucción puede acceder a la misma para buscar un operando antes de proceder con su ejecución. Esta situación es incompatible con la segmentación tradicional, según la cual sólo puede accederse a la memoria de datos después de la etapa de ejecución. Además, la presencia de un número tan elevado de modos de direccionamiento, sumado a la complejidad que poseían algunos de ellos, volvía muy complicada la tarea de plantear un flujo de ejecución general para cualquier instrucción que contemplara todas las alternativas posibles. A su vez, esta adversidad dificultaba considerablemente la labor de diseñar una unidad de control para el nuevo procesador capaz de determinar todos los pasos a seguir para ejecutar una instrucción una vez decodificado e interpretado su código de operación. Finalmente, todo esto permite concluir que resulta sumamente arduo para este tipo de arquitectura llevar a cabo una división del ciclo en ejecución en etapas para segmentar el cauce de la CPU, tal como se había expuesto previamente en la sección introductoria de este informe.

Por su parte, la arquitectura RISC presenta en su ISA (instruction set architecture – repertorio de instrucciones) instrucciones particulares para acceder a la memoria de datos y modos de direccionamiento sencillos y limitados en número. La primera consecuencia de este diseño consiste en que aquí en ningún caso podría existir la posibilidad de que la CPU requiera acceder a la memoria de datos para obtener un operando antes de proceder con la ejecución de la instrucción en cuestión. Asimismo se reduce considerablemente el abanico de alternativas posibles de ejecución para una instrucción, por lo cual la tarea de predecir y determinar un flujo de ejecución general para todas las instrucciones se vuelve mucho más fácil y sencilla. Como cabría esperarse, el diseño de la unidad de control encargada de decodificar la próxima instrucción a ejecutar y dictaminar el resto de los pasos para finalizar su ejecución también será más asequible, ya que la mayor parte del trabajo recaerá sobre el ensamblador: el número de instrucciones del programa a traducir a código máquina será superior al requerido para una arquitectura CISC debido a que en esta última una única instrucción puede desempeñar la misma tarea que varias instrucciones RISC, con la desventaja ya previamente mencionada de que su ejecución será considerablemente más compleja y el mayor peso de la labor de llevarla a cabo deberá ser soportado directamente por la unidad de control, dificultando su diseño para el desarrollador del mismo. Por último, como ya se habrá podido adivinar, este tipo de arquitectura es plenamente compatible con la

implementación tradicional de cinco etapas para la segmentación del cauce del procesador. De hecho, el simulador WinMIPS64 mencionado en la introducción de este informe representa una CPU cuya arquitectura es de tipo RISC y posee implementada exactamente esta misma metodología de segmentación para acelerar la ejecución de sus instrucciones. Por lo tanto, ya se posee un modelo completamente funcional y operativo como punto de partida para incluir esta técnica en el nuevo procesador a diseñar. Aunque dicho modelo no se encuentre descripto en ningún lenguaje de descripción de hardware, ofrece una perspectiva general, si bien abstracta, sobre el camino a seguir para aplicar efectivamente esta segmentación, sin necesidad de pensar en una división alternativa en etapas para la ejecución de las instrucciones ni en una manera para implementarla en la nueva CPU.

Teniendo en cuenta que el modelo original únicamente ofrecía una descripción para la CPU pero no para la PC de la cual debería formar parte, ni tampoco incluía un ensamblador para traducir las instrucciones escritas por el programador en el lenguaje Assembler en código máquina inteligible para el procesador, otro apartado cuya consideración se consideró prioritaria fue la inclusión tanto de una memoria principal para almacenar las instrucciones y los datos necesarios para la ejecución del programa como de un ensamblador que fuera capaz de leer un archivo creado por el usuario que contuviera un programa Assembler y convertirlo a código máquina. De esta manera podría ofrecerse un diseño tan similar como sea posible al representado en los simuladores MSX88 y WinMIPS64, los cuales presentan tanto la totalidad de los componentes de la PC (CPU, memoria principal y periféricos de E/S) como el software necesario para traducir código Assembler a código binario y así el usuario pueda programar en un lenguaje más abstracto y cercano a su idioma natural que el código máquina puro. Así también se podría prescindir de la estructura interna incluida en el diseño original para suplantar la ausencia de la memoria principal, la cual funcionaba como una cola auxiliar integrada directamente en el procesador para almacenar temporalmente las instrucciones y los datos transmitidos directamente por el usuario en código máquina hasta que la CPU requiriera hacer uso de ellos. Como se recordará, la existencia de esta dicha estructura no respetaba los principios de la arquitectura von Neumann, según la cual las únicas unidades de almacenamiento permitidas en el procesador son los registros, y además no poseía ningún correlato con la disposición y el funcionamiento real de una CPU, por lo que se recomendaba eliminarla del diseño con la mayor prontitud posible.

Desafortunadamente, considerando que el objetivo planteado para este proyecto no posee inicialmente ninguna relación con las operaciones que puedan involucrar alguno de los periféricos de E/S de la PC (impresora, microconmutadores, leds, pantalla de la terminal,

etc.), se ha determinado omitir por el momento la inclusión de estos últimos en el nuevo diseño por considerarse la tarea demasiado laboriosa y compleja aún sabiendo que la computadora permanecería incompleta si nos basamos en los principios de la arquitectura von Neumann, en la cual toda PC debe estar conformada tanto por la CPU y la memoria principal como así también por los dispositivos de E/S para poder interactuar con el usuario. Una situación similar ocurre con las interrupciones de hardware y software: al no haberse juzgado como indispensables para lograr el objetivo del proyecto, su implementación ha sido pospuesta en esta versión inicial del diseño de la PC. Por lo tanto, la modelización de estos atributos quedará como desafío para algún diseñador que se encuentre interesado en optimizar el modelo actual en alguno de estos apartados.

Uno de los atributos cuya implementación se consideró prioritaria fue la incorporación de la mayor cantidad posible de instrucciones del repertorio del procesador, siempre teniendo en cuenta que la migración de la arquitectura CISC a una RISC podría acarrear modificaciones considerables en el formato de dichas instrucciones debido a la limitación y la simplificación en los modos de direccionamiento y en el funcionamiento de las mismas hasta tal punto de tener que redefinirlas por completo desde el comienzo a partir de los principios de la arquitectura RISC. Afortunadamente ya se dispone del simulador WinMIPS64 con un repertorio de instrucciones plenamente definido, operativo y compatible con una CPU de tipo RISC, el cual obviamente fue utilizado como base para determinar el repertorio para el nuevo procesador. Partiendo de la premisa de que resultaría imposible obtener una implementación total del nuevo repertorio debido a su elevado número de instrucciones y la considerable complejidad que presentan algunas de ellas, se decidió prestarles especial atención a aquellas instrucciones vinculadas a las operaciones de transferencia de datos y de control, aritmético-lógicas, control y de desplazamiento de bits, ya que éstas son las que el programador suele utilizar con mayor asiduidad durante la escritura de sus programas y al mismo tiempo resultan ser para el diseñador las más sencillas de incorporar al diseño dado que no requieren alterar drásticamente el diseño de la CPU ni modelar nuevos componentes sino simplemente realizar algunas modificaciones a aquéllos ya existentes para que puedan buscar, decodificar y ejecutar las nuevas instrucciones. En cambio, la incorporación de otro tipo de instrucciones relacionados con el manejo de pila y subrutinas, interrupciones y periféricos de E/S requeriría, además de los cambios previamente mencionados, no sólo diseñar nuevos componentes para representar y gestionar el comportamiento de los nuevos dispositivos sino también remodelar el diseño original para poder adaptarlo a ellos, por lo cual la consideración de estos tópicos se ha omitido por completo para esta versión inicial de la CPU. Aún así, se asume que la

cantidad y variedad de las instrucciones implementadas hasta el momento resultan más que suficientes para que el usuario pueda desarrollar distintos programas con el nivel de complejidad necesario para comprobar con un alto grado de detalle y profundidad el comportamiento del procesador ante las diversas tareas que se le asignen a través de dichos programas.

Con respecto al diseño de los componentes internos de la nueva CPU, se optó por tomar como referencia la arquitectura von Neumann, según la cual el procesador debe presentar tres secciones principales: la unidad de control, la unidad aritmético-lógica (ALU) y el banco de registros, cada una de las cuales encargada de cumplir una función particular e independiente de las demás. En la sección anterior ya se había descripto la disposición interna planificada para la CPU original, la cual consistía en realidad de un único componente central que concentraba en su lógica interna todas las funcionalidades de las tres secciones del procesador previamente mencionadas: decodificar la próxima instrucción a ejecutar y determinar el flujo restante de su ciclo de ejecución (unidad de control); y, en caso de ser necesario, realizar la operación aritmético/lógica requerida para ejecutarla (ALU) y/o almacenar un valor final en uno de los registros de propósito general del procesador (banco de registros). También se encargaba de llevar a cabo la búsqueda de la próxima instrucción a ejecutar y de gestionar los accesos a la memoria principal (en este caso, la cola interna auxiliar del procesador) para realizar operaciones de lectura o escritura sobre ella. Como se recordará, se había mencionado como principal inconveniente de este diseño la posibilidad casi segura de que, cuando el diseñador decidiera comenzar a incorporar nuevas prestaciones al procesador (más instrucciones del repertorio, segmentación del cauce, memoria principal y periféricos de E/S, operaciones en punto flotante, etc.), el modelo se volvería tan complejo que la implementación de dichas mejoras se volvería prácticamente inviable a menos que se dividiera y asignara cada una de las tareas de la CPU a un componente separado. Por lo tanto, al planificar el nuevo diseño se planteó inicialmente la labor de modelar tres componentes individuales para representar las tres secciones principales de la CPU, cada una con su propia tarea particular y exclusiva; así como también se consideraron las cinco etapas que constituyen el ciclo de ejecución de la implementación tradicional de la segmentación del cauce ya que éste será al fin y al cabo el que terminará rigiendo el comportamiento general de la CPU, concluyéndose así la necesidad de incorporar dos componentes adicionales, uno encargado exclusivamente de realizar la búsqueda de la siguiente instrucción y actualizar el Puntero de Instrucción (etapa de búsqueda) y otro para gestionar el acceso a la memoria principal desde la CPU para operaciones de lectura o escritura (etapa de acceso a memoria).

Por su parte, las restantes etapas del ciclo de ejecución ya se encuentran asociadas y representadas cada una por una de las secciones principales de la CPU: decodificación (unidad de control), ejecución (ALU) y almacenamiento en registro (banco de registros), por lo que resulta innecesario diseñar componentes adicionales para modelar sus respectivos comportamientos. De esta manera, mientras que en el modelo original cualquier modificación podía tener repercusiones sobre el funcionamiento de la totalidad de la CPU, aquí resulta posible realizar alteraciones sobre uno de los componentes sin que los demás se vean afectados. Por ejemplo, si se desean implementar nuevas instrucciones del repertorio, en un principio sólo sería necesario modificar la unidad de control para que reconozca e interprete los nuevos códigos de operación, sin necesidad de alterar el resto de los componentes ni que su funcionalidad se vea comprometida en caso de que los cambios no hayan sido correctamente incorporados, lo cual sería imposible en el modelo original. Además, al dividir el trabajo total de la CPU en subtareas más pequeñas el diseño final se simplifica considerablemente, resultando el mismo más fácil de entender y modificar principalmente para un diseñador que no se encuentre familiarizado con él y desee realizarle cualquier tipo de optimizaciones.

Se entendió que la máquina de estados desempeña un rol esencial en el diseño de la PC en primer lugar por su capacidad para administrar el comportamiento global de la CPU gestionando la transición entre las distintas etapas posibles del ciclo de ejecución del procesador a través del tiempo en función de la etapa ejecutada anteriormente, el tipo de la instrucción actualmente en ejecución y, en caso de encontrarse activada la segmentación del cauce, la presencia de posibles atascos que difieran temporalmente la ejecución tanto de la instrucción actual como de las subsiguientes. La máquina se ha diseñado de manera tal que cada estado estaría representando una etapa distinta del ciclo de ejecución, aunque la acción a realizar en cada uno de ellos podría variar según sea requerido por la instrucción ejecutada. La tarea de la máquina de estados se vuelve especialmente vital y compleja si el cauce se encuentra segmentado ya que la máquina de estados deberá ser capaz de gestionar simultáneamente las transiciones entre etapas de múltiples instrucciones ejecutándose en forma paralela, aunque, como se recordará, siempre lo harán en etapas diferentes debido a las obvias limitaciones en el hardware de la CPU que impiden que la ejecución de las mismas sea estrictamente paralela. Por otra parte, esta máquina representa para el usuario la interfaz más intuitiva y eficaz para analizar y comprender externamente el funcionamiento del procesador, manteniéndose detalladamente informado y actualizado a través de ella sobre la actividad del mismo en todo momento. Dicha información le permitirá identificar unívocamente la/s

instrucciones del programa que están siendo ejecutadas en un instante dado y la etapa exacta del ciclo de ejecución en la cual se encuentra cada una de ellas. Sin esta interfaz, el único método del cual podría disponer el usuario para interiorizarse sobre el estado de la CPU sería estudiar las variaciones a través del tiempo en los valores contenidos en cada una de las unidades de almacenamiento de la PC (memoria principal y banco de registros de propósito general), así como también los cambios en las señales transmitidas entre las distintas etapas conforme progresa la ejecución del programa. Por lo tanto, se ha realizado e invertido todo el esfuerzo necesario para que en este diseño la máquina de estados refleje fielmente la actividad del procesador de una manera fácil de interpretar para el usuario, utilizando como modelo la interfaz gráfica provista por el simulador WinMIPS64 para representar en el tiempo el avance de la ejecución de las instrucciones a través de las etapas en las que se encuentra segmentado el cauce del procesador, las cuales resultan coincidir con aquéllas elegidas para segmentar el cauce de la nueva CPU, con algunas ligeras modificaciones que serán explicadas en la próximamente en esta misma sección.

En consonancia con las conclusiones ofrecidas en la sección anterior sobre las desventajas presentadas por el diseño original del procesador, se ha optado por simplificar el nuevo modelo tanto como fuera posible, eliminando por completo cualquier tipo de señales prescindibles cuya tarea no se encontrara plenamente vinculada con el objetivo principal de este proyecto. Por lo tanto, esta CPU no poseerá la posibilidad de reiniciarse a menos que se reinicie la tototalidad de la simulación, además de que su flujo de ejecución también se ha modificado considerablemente respecto al original: ahora el usuario se limitará simplemente a impartir a la PC la orden de ejecutar el programa y se abstraerá del resto de la ejecución, encargándose la CPU por sí sola se ejecutar secuencialmente las instrucciones a medida que se vayan completando las anteriores sin esperar ninguna clase de señal de reconocimiento por parte del usuario. De esta manera, se puede concentrar el diseño exclusivamente en la implementación de la segmentación del cauce, la cual se asume que será la parte más compleja y delicada del modelo, sin mencionar la ALU, la unidad de control y el banco de registros, e incluso la memoria principal y el ensamblador si se considera que el diseño final consistirá en una computadora que sea capaz de recibir programas escritos por el usuario/programador en Assembler, traducirlos a código máquina y ejecutarlos en su procesador para alcanzar el o los objetivos por el mismo. Si a todos estos componentes se les agregaran las funcionalidades del modelo original, se estaría dificultando innecesariamente una descripción ya de por sí sumamente compleja para el diseñador, quien agredecerá cualquier tipo de facilitación que se le ofrezca para desarrollar su tarea.

Como se recordará, se había mencionado como una de las principales desventajas del diseño original la metodología seleccionada para realizar la descripción del mismo, conocida como comportamental. Aunque la utilización de la misma en lugar de otra de las técnicas disponibles pudo haberle resultado en un principio más fácil y atractiva al diseñador por ser la estructura de su código la más semejante a la de los lenguajes de programación de alto nivel, precisamente por esta última razón el resultado obtenido será demasiado abstracto y alejado de la representación real de un circuito digital. Esto se debe a que, al estar esta metodología más orientada al software que al hardware, a pesar de que existe la posibilidad de que a través de ella se pueda describir correctamente el comportamiento esperado por parte de dicho circuito, el diseño terminará concentrando toda su lógica en un único componente extremadamente complejo encargado de ejecutar por sí sólo todas las tareas necesarias para cumplir los objetivos planteados para el modelo. A medida que el circuito a representar se complejice, aun más acentuadamente se presentará este fenómeno y, en el caso de este proyecto en particular, al tratarse el procesador precisamente del circuito digital más complicado posible de todos los que existen, conforme se le incorporen nuevas funcionalidades y mejoras eventualmente el modelo se volverá prácticamente ininteligible e inmanejable para cualquier diseñador, especialmente si nunca antes había trabajado sobre él. Por lo tanto, antes de alcanzar esta situación límite, se aprovechó la oportunidad que presentaba la determinación de replantear todo el diseño de la CPU desde el principio para proceder directamente a cambiar por completo la metodología utilizada para llevar a cabo su descripción: en lugar del método comportamental se recurrirá al estructural, el cual permite definir el circuito a partir de instancias de múltiples componentes, representando cada componente a una sección de hardware distinta dentro de la PC en general (CPU, memoria principal y periféricos de E/S cuando estos últimos se implementen en el diseño) y del procesador en particular (unidad de control, ALU, banco de registros, etc.). Se observa que, al menos inicialmente, se deberá diseñar una jerarquía de tres niveles para obtener una descripción fiel de la computadora, ubicándose la PC en el nivel superior, sus tres partes fundamentales en el nivel intermedio y, finalmente, los componentes de la CPU en el nivel inferior del diseño. Mientras que en la metodología comportamental la totalidad de las actividades a realizar por todos los dispositivos previamente mencionados debería ser llevada a cabo por un único componente con la consecuencia de que su lógica interna sería extremadamente compleja; en la descripción estructural, al dividir el problema general en tareas menores entre distintos componentes independientes entre sí, cada uno responsable de ejecutar una de ellas, se obtendrá un diseño más fácil de comprender y de modificar para

cualquier diseñador ya que, al ser su funcionalidad más sencilla, ninguno de estos componentes individuales podrá poseer una lógica tan compleja como la del componente central que resultaría de utilizar el método comportamental.

3.1. Arquitectura

A continuación se describirán las características principales que definen la arquitectura de la nueva computadora. Como se explicó en la introducción de esta sección del informe (“3. TDA 1819”), se observarán notorias diferencias respecto al diseño de la PC original, las cuales incluyen tanto atributos que ya se encontraban implementados como funcionalidades adicionales que se han incorporado en esta versión.

3.1.1. Atributos generales

Se comenzará presentando una tabla con los atributos generales del nuevo diseño tal y como se hizo en el caso del procesador original. En ella se podrá observar la presencia de funcionalidades adicionales, como la implementación de la segmentación en el cauce, de la memoria principal y de registros de uso particular, resultando la incorporación de estos dos últimos componentes esencial para que la descripción de la PC sea lo más fiel posible a la estructura y el comportamiento de una computadora real. También se ha elevado considerablemente la cantidad de registros de uso general tanto enteros como de punto flotante y se han modificado algunos de los modos de direccionamiento para que las instrucciones para que ahora requieran un único ciclo de reloj para ejecutarse, tal y como debería ser para una arquitectura de tipo RISC. Sin embargo, ya sea para conservar algunos de los atributos del diseño original que ya se encontraban implementados o bien por decisión personal del autor del informe por considerarlas más fáciles de incorporar se decidió incluir en el procesador características propias de una arquitectura de tipo CISC, como una interfaz de entrada/salida independiente de la memoria principal, formatos y longitudes variables para las instrucciones del repertorio y modos de direccionamiento adicionales, entre otras. Por lo tanto, tal y como se había mencionado en la introducción de esta sección del informe (“3. TDA 1819”), el resultado final no será una arquitectura RISC pura sino un híbrido, es decir, una combinación entre ambos tipos de arquitectura, aunque, como se apreciará a

continuación, presentando un mayor número de características que la inclinan hacia el tipo RISC.

Tabla 35. Atributos generales del nuevo diseño de la computadora.

Tipo de arquitectura	RISC
Ciclos de ejecución	Cada instrucción requiere exactamente un único ciclo de reloj para ejecutarse
Interfaz de Entrada/Salida	E/S independiente (*) (***)
Formato y longitud de las instrucciones	Instrucciones de formato y longitud variables: dos, cuatro, cinco, seis u ocho bytes en función de su cantidad y tamaño de operandos (***)
Frecuencia del reloj	20 MHz
Bus de datos	32 bits
Bus de direcciones	16 bits
Bus de control	2 bits: lectura (read) y escritura (write)
Mínima unidad de memoria direccionable	1 byte
Tamaño total de memoria	64 kilobytes
Modos de direccionamiento	Implícito (*) (***)
	Inmediato
	Directo de Registro
	Directo de Memoria
	Indirecto con Desplazamiento Relativo al SP (*) (***)
Ejecución de operaciones aritmético-lógicas	1 ALU (Arithmetic Logic Unit o Unidad Aritmético-Lógica) para operandos enteros
	1 FPU (Floating-Point Unit o Unidad de Punto Flotante) para operandos en punto flotante
Registros de uso general	16 registros enteros: r0, r1, r2, r3, r4, r5, r6, r7, r8, r9, r10, r11, r12, r13, r14 y r15
	16 registros de punto flotante: f0, f1, f2, f3, f4, f5, f6, f7, f8, f9, f10, f11, f12, f13, f14 y f15
Registros de uso particular	IR (Instruction Register o Registro de Instrucción, 8 bits)
	IP (Instruction Pointer o Puntero de Instrucción, 16 bits)
	FLAGS (Banderas, 8 bits) (**) (***)

	FPFLAGS (Banderas para Punto Flotante, 8 bits) (**) (***)
	BP (Base Pointer o Puntero Base, 32 bits) (*) (***)
	SP (Stack Pointer o Puntero de Pila, 32 bits) (*) (***)
	RA (Return Address o Dirección de Retorno, 32 bits) (*) (***)
Segmentación	Cinco etapas: búsqueda, decodificación, ejecución, acceso a memoria y almacenamiento en registro
Almacenamiento en memoria	Little Endian (***)

Observaciones:

- (*): Atributo sin implementar.
- (**): Atributo parcialmente implementado: se explicará con mayor detenimiento en una observación posterior y en la próxima tabla de este informe.
- (***): Atributo incompatible con una arquitectura RISC:
    - ❖ Interfaz de Entrada/Salida: RISC propone que tanto la memoria principal como los periféricos de E/S compartan el mismo espacio de direcciones y las mismas instrucciones. Sin embargo, aquí se optó por distanciar ambas secciones, por lo que existirá un espacio de direcciones separado e instrucciones particulares para realizar operaciones de E/S, aunque éstas aún no se encuentren implementadas en el diseño.
    - ❖ Formato y longitud de las instrucciones: RISC establece que todas las instrucciones deben poseer un formato único y fijo. No obstante, en este diseño se determinó que la longitud de cada instrucción pueda variar en función de la cantidad y el tamaño de sus operandos para evitar desperdiciar espacio en memoria con direcciones sin utilizar por reservar más lugar del estrictamente necesario para almacenar la instrucción en cuestión.
    - ❖ Modos de direccionamiento: en la arquitectura RISC los modos de direccionamiento deben reducirse y limitarse en número y complejidad tanto como sea posible. Sin embargo, en este caso se incluyeron un par de modos adicionales que no serían imprescindibles para implementar todas las instrucciones del repertorio. Esto se debe a la decisión de incorporar un conjunto de instrucciones particulares para el manejo de la pila de la PC, también contradiciendo los principios del tipo RISC, según los cuales no debería existir ningún motivo para implementar instrucciones separadas para el acceso a la pila ya que el usuario tendría que ser capaz de utilizarla manipulando directamente el SP (Puntero de

Pila) como si se tratara de uno de los registros de uso general de la CPU. De todas maneras, dichas instrucciones aún no se encuentran implementadas, por lo cual el programador no debería prestar atención a esta cuestión por el momento.

- Registros de uso particular:
  - Registros FLAGS y FPFLAGS: La arquitectura CISC presenta un registro de uso particular llamado FLAGS, cuyos bits representan banderas asociadas al estado actual del procesador, principalmente respecto a su última operación aritmético-lógica ejecutada y la posibilidad de atender o ignorar peticiones vinculadas a interrupciones de hardware enmascarables. Esto se debe a que el repertorio de un procesador CISC posee instrucciones explícitas para realizar saltos condicionales en función del valor de la bandera que se desee (por ejemplo: “jz” y “jnz” para la bandera “Z”, “js” y “jns” para la bandera “S”, etc.). A partir de esto, la idea sería que el usuario realice una operación aritmético-lógica (suma, resta, comparación, etc.) y, a partir del estado posterior a su ejecución y del objetivo buscado, determine el tipo de bandera y, en consecuencia, del salto condicional a utilizar. En cambio, el repertorio asociado a la arquitectura RISC cuenta con instrucciones (“beq”, “bne”, “beqz” y “bnez”) para realizar saltos condicionales que efectúan ambos pasos a la vez: llevan a cabo la operación aritmético-lógica y la comparación necesaria para determinar si se cumple o no la condición de salto establecida, sin necesidad de utilizar la mayoría de las banderas del registro FLAGS de la arquitectura CISC, por lo que suele descartarse la incorporación de dicho registro en los procesadores RISC. No obstante, en el caso de la nueva CPU se determinó que la presencia tanto de interrupciones de hardware enmascarables (aún no implementadas) como de instrucciones de transferencia de control especiales (“bfpt” y “bfpf”) que se comportan de una manera similar a los saltos condicionales en CISC, imponía la necesidad de preservar la existencia del registro en el nuevo diseño con la misma funcionalidad que tendría en una arquitectura CISC. Además, al incorporarse adicionalmente la unidad de punto flotante o FPU junto con la unidad aritmético-lógica o ALU, se juzgó conveniente crear un nuevo registro de banderas llamado FPFLAGS con un contenido y comportamiento similar al registro FLAGS original aunque, mientras FLAGS reflejará el estado de la última operación

ejecutada en la ALU, FPFLAGS se asociará a las operaciones ejecutadas en la FPU. De esta manera, ambas unidades pueden trabajar en forma paralela libre e independientemente sin que ninguna tenga que preocuparse por la posibilidad de intentar actualizar simultáneamente su estado sobre el mismo registro. No obstante, la elección de la arquitectura RISC para esta computadora implica que la mayoría de los flags de ambos registros quedarán inutilizados dado que sus valores no presentan ningún interés para la ejecución de las instrucciones de salto condicionales. Por lo tanto, la implementación de los mismos en este diseño se encuentra parcialmente finalizada, por lo que no puede garantizarse su correcta actualización posterior a la ejecución de algunas de las operaciones aritmético-lógicas más complejas como la multiplicación y la división tanto en la ALU como en la FPU.

- Registros BP, SP y RA: En consonancia con la observación realizada anteriormente respecto a los modos de direccionamiento existentes en la arquitectura de la nueva computadora, en este diseño la presencia de instrucciones explícitas para apilar y desapilar valores en la pila implica que tanto el BP (Puntero Base) como el SP (Puntero de Pila) pertenecen al conjunto de registros de uso particular del procesador, es decir, no pueden ser accedidos ni modificador por el usuario bajo ningún concepto, mientras que normalmente en una arquitectura RISC esto no suele ser así, ya que, al no existir instrucciones particulares para el manejo de la pila el programador obligatoriamente debe manipular estos registros para poder leer o almacenar datos en ella. Algo similar ocurre con el registro RA (Dirección de Retorno): a diferencia de una arquitectura RISC tradicional, en este caso sí existe un manejo implícito de la pila en la invocación a las subrutinas a través de la ejecución de instrucciones especiales del repertorio (“call” y “ret”) que se encargan automáticamente de apilar y desapilar la dirección de retorno según corresponda sin intervención del usuario, por lo cual éste no tendría ningún motivo para acceder al RA, por lo tanto este registro también se incluyó dentro del conjunto de registros de uso particular, mientras que en una RISC normal suele formar parte de los registros de uso general. De todas maneras, dichas instrucciones aún no se

encuentran implementadas, ergo, el programador no debería prestar atención a esta cuestión por el momento.

- Almacenamiento en memoria: por último, mientras que los procesadores cuya arquitectura es de tipo RISC suelen almacenar los bytes en la memoria principal de la PC desde el más significativo en la dirección más baja hacia el menos significativo en la más alta (Big Endian), en este modelo se determinó e implementó el formato y orden inversos: los bytes se guardarán desde el menos significativo en la dirección más baja hacia el más significativo en la más alta (Little Endian). Esta decisión se debió simplemente a una preferencia personal del autor de este informe ya que esta última manera de almacenar la información le pareció más lógica y natural que la primera. Cabe aclarar que esta modificación actúa sobre un nivel de abstracción tan bajo que el usuario que escriba programas para la CPU en el lenguaje Assembler ni siquiera debería percatarse de la diferencia, por lo cual su manera de programar no tendría que sufrir ningún cambio a pesar de que el formato de almacenamiento en memoria elegido por el diseñador sea incompatible con el de los procesadores RISC tradicionales.
- Para más detalles, comparar con la tabla 1 presentada en la introducción de este informe.

Como se acaba de mencionar en una de las observaciones de la tabla anterior, en este diseño se encuentran incluidos dos registros de banderas, FLAGS y FPFLAGS, asociados a las operaciones aritmético-lógicas realizadas en la ALU y la FPU respectivamente. Ambos poseen las mismas banderas, pero su valor no será el mismo ya que representan el estado de dos unidades de ejecución distintas. A continuación se especifican la disposición y ubicación de dichas banderas en los dos registros y el propósito de cada una de ellas:

Tabla 36. Banderas de los registros FLAGS y FPFLAGS del nuevo procesador.

	Bit 7	**Bit 6**	**Bit 5**	**Bit 4**	**Bit 3**	**Bit 2**	**Bit 1**	**Bit 0**
Bandera	F	I	Z	S	O	C	A	P

Observaciones:

- Bandera F (Punto Flotante): su valor se actualiza en el registro FPFLAGS sólo si en la FPU se ejecuta alguna de las instrucciones de comparación especiales incorporadas en el repertorio a tal efecto. Esta bandera es utilizada por algunas instrucciones de transferencia de control condicional (saltos condicionales) en función del resultado obtenido en la última comparación realizada en la FPU. Por su parte, en el registro FLAGS la bandera permanece inutilizada y su valor, irrelevante.
- Bandera I (Interrupción): se asocia al manejo de interrupciones de hardware enmascarables, es decir, aquellas solicitudes por parte de los dispositivos de hardware cuyas prioridades son lo suficientemente bajas como para que su atención pueda ser ignorada por la CPU si el usuario así lo deseara. Si el valor de la bandera es un uno, las peticiones serán atendidas; en cambio, si es un cero, serán ignoradas. Esta bandera sólo posee relevancia en el registro FLAGS, mientras que en el registro FPFLAGS permanece inutilizada.
- Bandera Z (Cero): ofrece información asociada al estado de la última operación aritmético-lógica ejecutada en la ALU (registro FLAGS) o la FPU (registro FPFLAGS). Si el valor de la bandera es un uno, el resultado obtenido fue igual a cero; en cambio, si es un cero, fue distinto de cero. En una arquitectura CISC esta bandera suele utilizarse para la evaluación de saltos condicionales, pero aquí, al tratarse de una arquitectura RISC, su utilidad es prácticamente nula, por lo que su implementación se encuentra incompleta y su correcto comportamiento y funcionamiento no está plenamente garantizado para todas las operaciones aritmético-lógicas ejecutadas tanto en la ALU como en la FPU.
- Bandera S (Signo): ofrece información asociada al estado de la última operación aritmético-lógica ejecutada en la ALU (registro FLAGS) o la FPU (registro FPFLAGS). El valor de esta bandera coincide exactamente con el del bit más significativo del resultado obtenido. De esta manera, si se interpretara a dicho resultado en términos del sistema de representación conocido como complemento a dos, podría decirse que, si el valor de la bandera es un uno, el resultado de la última operación aritmético-lógica fue negativo; en cambio, si es un cero, fue un valor natural. En una arquitectura CISC esta bandera suele utilizarse para la evaluación de saltos condicionales, pero aquí, al tratarse de una arquitectura RISC, su utilidad es prácticamente nula, por lo que su implementación se encuentra incompleta y su correcto comportamiento y funcionamiento no está plenamente

garantizado para todas las operaciones aritmético-lógicas ejecutadas tanto en la ALU como en la FPU.

- Bandera O (Overflow o Desbordamiento): ofrece información asociada al estado de la última operación aritmética ejecutada en la ALU (registro FLAGS) o la FPU (registro FPFLAGS). Más precisamente, indica si durante la operación se produjo un desbordamiento aritmético, es decir, si el número de bits utilizados para realizar dicha operación (ancho de la ALU) no alcanza para representar el resultado en complemento a dos. Si el valor de la bandera es un uno, el desbordamiento ocurrió; en cambio, si es un cero, no se produjo. Esta bandera sólo debe ser tenida en cuenta si los operandos de la operación eran valores con signo, en caso contrario no posee ninguna utilidad. En una arquitectura CISC esta bandera suele utilizarse para la evaluación de saltos condicionales, pero aquí, al tratarse de una arquitectura RISC, su utilidad es prácticamente nula, por lo que su implementación se encuentra incompleta y su correcto comportamiento y funcionamiento no está plenamente garantizado para todas las operaciones aritméticas ejecutadas tanto en la ALU como en la FPU.
- Bandera C (Carry o Acarreo): ofrece información asociada al estado de la última operación aritmética ejecutada en la ALU (registro FLAGS) o la FPU (registro FPFLAGS). Más precisamente, indica si durante la operación se produjo un acarreo aritmético desde el bit más significativo del resultado (sólo si la operación realizada se hubiera tratado de una suma) o bien si existió un préstamo aritmético hacia el bit más significativo del primero de los dos operandos (sólo si la operación realizada se hubiera tratado de una resta). Si el valor de la bandera es un uno, el acarreo o préstamo ocurrió; en cambio, si es un cero, no se produjo. Esta bandera permite llevar a cabo sumas o restas entre números que requieren una mayor cantidad de bits para ser representados que el ancho de la ALU o la FPU, acarreando o sumando un dígito binario de una suma o resta parcial al bit menos significativo de una palabra o conjunto de bits más significativos. Además, en una arquitectura CISC esta bandera suele emplearse para la evaluación de saltos condicionales, pero aquí, al tratarse de una arquitectura RISC, su utilidad se reduce considerablemente, por lo que su implementación se encuentra incompleta y su correcto comportamiento y funcionamiento no está plenamente garantizado para todas las operaciones aritméticas ejecutadas tanto en la ALU como en la FPU.
- Bandera A (Acarreo Auxiliar): ofrece información asociada al estado de la última operación aritmética ejecutada en la ALU (registro FLAGS) o la FPU (registro FPFLAGS). Más precisamente, indica si durante la operación se produjo un acarreo

aritmético desde el cuarto bit menos significativo del resultado (sólo si la operación realizada se hubiera tratado de una suma) o bien si existió un préstamo aritmético hacia el cuarto bit menos significativo del primero de los dos operandos (sólo si la operación realizada se hubiera tratado de una resta). Si el valor de la bandera es un uno, el acarreo o préstamo ocurrió; en cambio, si es un cero, no se produjo. Esta bandera suele utilizarse principalmente a fin de ofrecer apoyo para operaciones aritméticas cuyos operandos se encuentran expresados en el sistema de representación BCD (Binary-Coded Decimal o Decimal Codificado en Binario), en el cual cada dígito decimal es codificado con una secuencia de cuatro bits. No obstante, al considerarse altamente improbable que el usuario realice operaciones aritméticas en dicho sistema de representación, esta bandera no se juzgó como imprescindible para este diseño, por lo que su implementación se encuentra incompleta y su correcto comportamiento y funcionamiento no está plenamente garantizado para todas las operaciones aritméticas ejecutadas tanto en la ALU como en la FPU.

- Bandera P (Paridad): ofrece información asociada al estado de la última operación aritmética ejecutada en la ALU (registro FLAGS) o la FPU (registro FPFLAGS). Más precisamente indica si, en la representación binaria del resultado obtenido (en el sistema binario sin signo o complemento a dos según corresponda), el número total de bits cuyo valor es igual a uno es par o impar. Si el valor de la bandera es un uno, el número de bits es par; en cambio, si es un cero, es impar. En una arquitectura CISC esta bandera suele utilizarse para la evaluación de saltos condicionales, pero aquí, al tratarse de una arquitectura RISC, su utilidad es prácticamente nula, por lo que su implementación se encuentra incompleta y su correcto comportamiento y funcionamiento no está plenamente garantizado para todas las operaciones aritméticas ejecutadas tanto en la ALU como en la FPU.

3.1.2. Directivas y repertorio de instrucciones

A continuación se incluye una tabla que contiene las directivas reconocidas por el ensamblador diseñado e incorporado a la PC con el objetivo de traducir las instrucciones escritas por el usuario en un lenguaje de programación de bajo nivel (Assembler) a un código máquina que pueda resultar inteligible para la unidad de control de la nueva CPU, la cual a su vez se encargará de coordinar su posterior ejecución conforme logre interpretarlas. Dichas

directivas deberán ser especialmente tenidas en cuenta por cualquier individuo que desee escribir programas en el lenguaje Assembler para ser ejecutados por el procesador ya que a través de las mismas será capaz de definir las distintas secciones del programa (datos, subrutinas y código del programa principal), así como también los diversos tipos de datos que pueden adoptar las variables a declarar. Se puede observar que la elección de las directivas por parte del diseñador se encuentra basada en aquéllas ya ofrecidas en el simulador WinMIPS64 para su propio ensamblador.

Tabla 37. Directivas válidas para el ensamblador de la nueva computadora.

Directiva	**Comentario**
.data	Comienzo de la sección de datos
.subr	Comienzo de la sección de subrutinas (*)
.text	Comienzo de la sección de código del programa principal
.code	Comienzo de la sección de código del programa principal (igual que .text)
.ascii *s1, s2, ...*	Almacena cadena/s ASCII
.asciiz *s1, s2, ...*	Almacena cadena/s ASCII terminada/s en cero
.byte *n1, n2, ...*	Almacena número/s de 8 bits con signo
.ubyte *n1, n2, ...*	Almacena número/s de 8 bits sin signo
.hword *n1, n2, ...*	Almacena número/s de 16 bits con signo
.uhword *n1, n2, ...*	Almacena número/s de 16 bits sin signo
.word *n1, n2, ...*	Almacena número/s de 32 bits con signo
.uword *n1, n2, ...*	Almacena número/s de 32 bits sin signo
.float *n1, n2, ...*	Almacena número/s en punto flotante (32 bits)

Observaciones:

- (*): Directiva sin implementar.
- *s1, s2, ...*: Denota una sucesión de cadenas de caracteres ASCII como “hola”, “bienvenido”, “lunes”, etc.
- *n1, n2, ...*: Denota una sucesión de números enteros o en punto flotante según corresponda como “1”, “25”, “-5.6”, etc.

Otro de los pilares fundamentales que el usuario debería considerar, estudiar y aprender si posee la intención de escribir programas en el lenguaje Assembler que resulten compatibles con la nueva CPU es el repertorio de instrucciones que pueden ser ejecutadas por la misma. En la siguiente tabla se incluyen tanto las instrucciones que ya se encuentran implementadas como las que aún no lo están, en caso de que el lector tenga el interés de incorporarlas por sus propios medios al modelo actual o al menos desee conocer en términos generales cuál sería la manera que el diseñador tiene pensada para añadir al procesador las funcionalidades pendientes. Como podrá deducirse rápidamente, los repertorios del simulador WinMIPS64 y en menor medida del MSX88 han influenciado de algún modo sobre la forma y la disposición elegidas para denominar todas las nuevas instrucciones.

Tabla 38. Repertorio de instrucciones de la nueva CPU.

Tipo de instrucción	Instrucción	Comentario
Transferencia de Datos	lb r_d, $Inm(r_i)$	Copia en r_d un byte (8 bits) desde la dirección ($Inm + r_i$)
	sb r_f, $Inm(r_i)$	Guarda los 8 bits menos significativos de r_f en la dirección ($Inm + r_i$)
	lh r_d, $Inm(r_i)$	Copia en r_d un half-word (16 bits) desde la dirección ($Inm + r_i$)
	sh r_f, $Inm(r_i)$	Guarda los 16 bits menos significativos de r_f a partir de la dirección ($Inm + r_i$)
	lw r_d, $Inm(r_i)$	Copia en r_d un word (32 bits) desde la dirección ($Inm + r_i$)
	sw r_f, $Inm(r_i)$	Guarda los 32 bits de r_f a partir de la dirección ($Inm + r_i$)
	lf f_d, $Inm(r_i)$	Copia en r_d un valor en punto flotante (32 bits) desde la dirección ($Inm + r_i$)
	sf f_f, $Inm(r_i)$	Guarda los 32 bits de f_f a partir de la dirección ($Inm + r_i$)
	mff f_d, f_f	Copia el valor del registro f_f (32 bits) al registro f_d
	mfr f_d, r_f	Copia los 32 bits del registro r_f entero al

		registro f_d de punto flotante
	mrf r_d, f_f	Copia los 32 bits del registro f_f de punto flotante al registro r_d entero
	tf f_d, f_f	Convierte a punto flotante el valor entero copiado al registro f_f (32 bits), dejándolo en f_d
	ti f_d, f_f	Convierte a entero el valor en punto flotante contenido en f_f (32 bits), dejándolo en f_d
Aritmética	dadd r_d, r_f, r_g	Suma r_f con r_g, dejando el resultado en r_d (valores con signo)
	daddi r_d, r_f, N	Suma r_f con el valor inmediato N, dejando el resultado en r_d (valores con signo)
	daddu r_d, r_f, r_g	Suma r_f con r_g, dejando el resultado en r_d (valores sin signo)
	daddui r_d, r_f, N	Suma r_f con el valor inmediato N, dejando el resultado en r_d (valores sin signo)
	addf f_d, f_f, f_g	Suma f_f con f_g, dejando el resultado en f_d (en punto flotante)
	dsub r_d, r_f, r_g	Resta r_g a r_f, dejando el resultado en r_d (valores con signo)
	dsubu r_d, r_f, r_g	Resta r_g a r_f, dejando el resultado en r_d (valores sin signo)
	subf f_d, f_f, f_g	Resta f_g a f_f, dejando el resultado en f_d (en punto flotante)
	dmul r_d, r_f, r_g	Mutiplica r_f con r_g, dejando el resultado en r_d (valores con signo)
	dmulu r_d, r_f, r_g	Mutiplica r_f con r_g, dejando el resultado en r_d (valores sin signo)
	mulf f_d, f_f, f_g	Multiplica f_f con f_g, dejando el resultado en f_d (en punto flotante)
	ddiv r_d, r_f, r_g	Divide r_f por r_g, dejando el resultado en r_d (valores con signo)
	ddivu r_d, r_f, r_g	Divide r_f por r_g, dejando el resultado en r_d (valores sin signo)
	divf f_d, f_f, f_g	Divide f_f por f_g, dejando el resultado en f_d (en

		punto flotante)
	slt r_d, r_f, r_g	Compara r_f con r_g, dejando r_d =1 si r_f es menor que r_g (valores con signo)
	slti r_d, r_f, N	Compara r_f con el valor inmediato N, dejando r_d =1 si r_f es menor que N (valores con signo)
	ltf f_f, f_g	Compara f_f con f_g, dejando el flag FP=1 si f_f es menor que f_g (en punto flotante)
	lef f_f, f_g	Compara f_f con f_g, dejando el flag FP=1 si f_f es menor o igual que f_g (en punto flotante)
	eqf f_f, f_g	Compara f_f con f_g, dejando el flag FP=1 si f_f es igual que f_g (en punto flotante)
	negr r_d, r_f	Calcula el opuesto de r_f, dejando el resultado en r_d (valores con signo)
Lógica	andr r_d, r_f, r_g	Realiza un AND entre r_f y r_g (bit a bit), dejando el resultado en r_d
	andi r_d, r_f, N	Realiza un AND entre r_f y el valor inmediato N (bit a bit), dejando el resultado en r_d
	orr r_d, r_f, r_g	Realiza un OR entre r_f y r_g (bit a bit), dejando el resultado en r_d
	ori r_d, r_f, N	Realiza un OR entre r_f y el valor inmediato N (bit a bit), dejando el resultado en r_d
	xorr r_d, r_f, r_g	Realiza un XOR entre r_f y r_g (bit a bit), dejando el resultado en r_d
	xori r_d, r_f, N	Realiza un XOR entre r_f y el valor inmediato N (bit a bit), dejando el resultado en r_d
	notr r_d, r_f	Realiza un NOT sobre r_f (bit a bit), dejando el resultado en r_d
Desplazamiento de Bits	dsl r_d, r_f, r_N	Desplaza a izquierda r_N veces los bits del registro r_f, dejando el resultado en r_d
	dsli r_d, r_f, N	Desplaza a izquierda N veces los bits del registro r_f, dejando el resultado en r_d
	dsr r_d, r_f, r_N	Desplaza a derecha r_N veces los bits del registro r_f, dejando el resultado en r_d
	dsri r_d, r_f, N	Desplaza a derecha N veces los bits del

		registro r_f, dejando el resultado en r_d
	dsls r_d, r_f, r_N	Igual que la instrucción **dsl** pero mantiene el signo del valor desplazado
	dslsi r_d, r_f, N	Igual que la instrucción **dsli** pero mantiene el signo del valor desplazado
	dsrs r_d, r_f, r_N	Igual que la instrucción **dsr** pero mantiene el signo del valor desplazado
	dsrsi r_d, r_f, N	Igual que la instrucción **dsri** pero mantiene el signo del valor desplazado
Transferencia de Control	jmp $offN_I$	Salta incondicionalmente a la instrucción etiquetada $offN_I$
	beq r_f, r_g, $offN_I$	Si r_f es igual a r_g, salta a la instrucción etiquetada $offN_I$
	bne r_f, r_g, $offN_I$	Si r_f no es igual a r_g, salta a la instrucción etiquetada $offN_I$
	beqz r_f, $offN_I$	Si r_f es igual a 0, salta a la instrucción etiquetada $offN_I$
	bnez r_f, $offN_I$	Si r_f no es igual a 0, salta a la instrucción etiquetada $offN_I$
	bfpt $offN_I$	Salta a la instrucción etiquetada $offN_I$ si el flag F=1
	bfpf $offN_I$	Salta a la instrucción etiquetada $offN_I$ si el flag F=0
Subrutinas	call $offN_S$	Llama a subrutina cuyo inicio es la instrucción etiquetada $offN_S$ (*)
	ret	Retorna de la subrutina (*)
Manejo de Interrupciones	int N_I	Salva los flags y ejecuta la interrupción por software N_I (*)
	iret	Retorna de la interrupción y restablece los flags (*)
	cli	Inhabilita interrupciones enmascarables (*)
	sti	Habilita interrupciones enmascarables (*)
Manejo de la Pila	pushb r_f	Apila los 8 bits menos significativos de r_f (*)
	popb r_d	Desapila un byte (8 bits) y lo carga en r_d (*)

	pushh r_f	Apila los 16 bits menos significativos de r_f (*)
	poph r_d	Desapila un half-word (16 bits) y lo carga en r_d (*)
	pushw r_f	Apila los 32 bits de r_f (*)
	popw r_d	Desapila un word (32 bits) y lo carga en r_d (*)
	pushfl	Apila los 8 bits del registro FLAGS (*)
	popfl	Desapila un byte (8 bits) y lo carga en el registro FLAGS (*)
	pushfp	Apila los 8 bits del registro FLAGS (punto flotante) (*)
	popfp	Desapila un byte (8 bits) y lo carga en el registro FLAGS (punto flotante) (*)
	pushra	Apila los 32 bits del registro RA (*)
	popra	Desapila un word (32 bits) y lo carga en el registro RA (*)
Entrada/Salida	inb r_d, $offN_P$	Carga el valor del puerto etiquetado $offN_P$ en los 8 bits menos significativos de r_d (*)
	outb r_f, $offN_P$	Carga los 8 bits menos significativos de r_f en el puerto etiquetado $offN_P$ (*)
	inh r_d, $offN_P$	Carga el valor del puerto etiquetado $offN_P$ en los 16 bits menos significativos de r_d (*)
	outh r_f, $offN_P$	Carga los 16 bits menos significativos de r_f en el puerto etiquetado $offN_P$ (*)
	inw r_d, $offN_P$	Carga el valor del puerto etiquetado $offN_P$ en los 32 bits de r_d (*)
	outw r_f, $offN_P$	Carga los 32 bits de r_f en el puerto etiquetado $offN_P$ (*)
Control	nop	Operación nula (no realiza ninguna acción)
	halt	Detiene la ejecución del procesador

Observaciones:

- (*): Instrucción sin implementar.

3.1.3. Códigos de operación e identificadores

Ahora se procederá a tratar asuntos que describen la lógica del procesador a un nivel de abstracción mucho más inferior que aquél en el cual se ubica el lenguaje Assembler y que, por lo tanto, el usuario que esté simplemente interesado en escribir programas a través de dicho lenguaje no tiene la obligación de conocerlos. No obstante, permiten comprender algunas cuestiones que pueden resultar sumamente útiles para comprender el funcionamiento interno y el progreso de la CPU a un nivel más detallado que la información ofrecida por la máquina de estados durante la ejecución de los distintos programas que se propongan para las simulaciones que se lleven a cabo. Por lo tanto, la lectura de los siguientes tópicos, aunque optativa, no deja de ser recomendable.

En primer lugar, se presentarán los códigos de operación asignados por el diseñador para cada una de las instrucciones que constituyen el repertorio del procesador. Cada instrucción posee un código único, el cual resulta fundamental para que la unidad de control de la CPU pueda interpretar e identificar a partir de él la denominación de la próxima instrucción a ejecutar y determinar a continuación los pasos a seguir para poder llevarla a cabo. Asimismo, el ensamblador deberá tener en cuenta estos identificadores para asociar cada instrucción a traducir al lenguaje máquina con su respectivo código de operación antes de almacenar la misma en la memoria principal de la PC para su posterior interpretación y ejecución.

Tabla 39. Códigos de operación de las instrucciones para la nueva CPU.

Tipo de instrucción	Instrucción	Código de operación
Transferencia de Datos	lb r_d, $Inm(r_i)$	00000100
	sb r_f, $Inm(r_i)$	00000101
	lh r_d, $Inm(r_i)$	00000110
	sh r_f, $Inm(r_i)$	00000111
	lw r_d, $Inm(r_i)$	00001000
	sw r_f, $Inm(r_i)$	00001001
	lf f_d, $Inm(r_i)$	00001010
	sf f_f, $Inm(r_i)$	00001011
	mff f_d, f_f	00001100
	mfr f_d, r_f	00001101

	mrf r_d, f_f	00001110
	tf f_d, f_f	00001111
	ti f_d, f_f	00010000
Aritmética	dadd r_d, r_f, r_g	00011000
	daddi r_d, r_f, N	00011001
	daddu r_d, r_f, r_g	00011010
	daddui r_d, r_f, N	00011011
	addf f_d, f_f, f_g	00011100
	dsub r_d, r_f, r_g	00011101
	dsubu r_d, r_f, r_g	00011110
	subf f_d, f_f, f_g	00011111
	dmul r_d, r_f, r_g	00100000
	dmulu r_d, r_f, r_g	00100001
	mulf f_d, f_f, f_g	00100010
	ddiv r_d, r_f, r_g	00100011
	ddivu r_d, r_f, r_g	00100100
	divf f_d, f_f, f_g	00100101
	slt r_d, r_f, r_g	00100110
	slti r_d, r_f, N	00100111
	ltf f_f, f_g	00101000
	lef f_f, f_g	00101001
	eqf f_f, f_g	00101010
	negr r_d, r_f	00101011
Lógica	andr r_d, r_f, r_g	00110000
	andi r_d, r_f, N	00110001
	orr r_d, r_f, r_g	00110010
	ori r_d, r_f, N	00110011
	xorr r_d, r_f, r_g	00110100
	xori r_d, r_f, N	00110101
	notr r_d, r_f	00110110
Desplazamiento de Bits	dsl r_d, r_f, r_N	01000000
	dsli r_d, r_f, N	01000001
	dsr r_d, r_f, r_N	01000010
	dsri r_d, r_f, N	01000011

	dsls r_d, r_f, r_N	01000100
	dslsi r_d, r_f, N	01000101
	dsrs r_d, r_f, r_N	01000110
	dsrsi r_d, r_f, N	01000111
Transferencia de Control	jmp $offN_I$	01001000
	beq r_f, r_g, $offN_I$	01001001
	bne r_f, r_g, $offN_I$	01001010
	beqz r_f, $offN_I$	01001011
	bnez r_f, $offN_I$	01001100
	bfpt $offN_I$	01001101
	bfpf $offN_I$	01001110
Subrutinas	call $offN_S$	01010000
	ret	01010001
Manejo de Interrupciones	int N_I	01010100
	iret	01010101
	cli	01010110
	sti	01010111
Manejo de la Pila	pushb r_f	01100000
	popb r_d	01100001
	pushh r_f	01100010
	poph r_d	01100011
	pushw r_f	01100100
	popw r_d	01100101
	pushfl	01100110
	popfl	01100111
	pushfp	01101000
	popfp	01101001
	pushra	01101010
	popra	01101011
Entrada/Salida	inb r_d, $offN_P$	01110000
	outb r_f, $offN_P$	01110001
	inh r_d, $offN_P$	01110010
	outh r_f, $offN_P$	01110011
	inw r_d, $offN_P$	01110100

	outw r_f, $offN_P$	01110101
Control	nop	10000000
	halt	10000001

Observaciones:

- Cada código de operación ocupa exactamente un byte u ocho bits.
- Los códigos de operación se encuentran representados en el sistema de numeración binario.

El diseñador ha determinado que cada uno de todos los registros del procesador, tanto de uso general como de propósito particular, debe poseer asociado un identificador único. Éstos identificadores deberán ser especialmente tenidos en cuenta por el ensamblador ya que los utilizará para representar y traducir en código máquina los registros de uso general que pueden ser utilizados por el usuario en cualquiera de las instrucciones del repertorio. Asimismo, cumplirán una función similar a la de los códigos de operación para las instrucciones: ofrecer a todos los componentes de la CPU que lo requieran (unidad de búsqueda, unidad de control, ALU, unidad de almacenamiento en registro) una manera de interpretar unívocamente el registro al cual se debe acceder ya sea para realizar una operación de lectura o escritura sobre él sin cometer ningún tipo de errores.

En particular, se han asignado los siguientes identificadores para los registros del nuevo procesador:

Tabla 40. Identificadores asignados para todos los registros de la nueva CPU.

Tipo de registro	Denominación	Identificador
Uso general	r0	0
	r1	1
	r2	2
	r3	3
	r4	4
	r5	5
	r6	6

	r7	7
	r8	8
	r9	9
	r10	10
	r11	11
	r12	12
	r13	13
	r14	14
	r15	15
	f0	16
	f1	17
	f2	18
	f3	19
	f4	20
	f5	21
	f6	22
	f7	23
	f8	24
	f9	25
	f10	26
	f11	27
	f12	28
	f13	29
	f14	30
	f15	31
Uso particular	IR	32
	IP	33
	FLAGS	34
	FPFLAGS	35
	BP	36
	SP	37
	RA	38

3.1.4. Formatos de instrucciones

Por otra parte, a continuación se muestran los formatos elegidos para representar las instrucciones del repertorio de manera tal de poder ser almacenadas por el ensamblador en la memoria principal de la PC y posteriormente interpretadas y ejecutadas por la CPU. Todas las instrucciones se encuentran conformadas por dos campos cuya presencia es obligatoria: una cabecera, cuyo propósito se explicará próximamente, y uno de los códigos de operación incluidos en la tabla anterior el cual, obviamente, variará en función de la instrucción. Los siguientes campos del formato, si los hubiere, dependerán del tamaño y la cantidad de operandos que posea la instrucción.

Aunque se admite que la opción seleccionada no es la más eficiente dado que desperdicia espacio en memoria (en varios de los campos quedan algunos bits sin utilizar), se justifica la elección en la priorización de una mayor agilidad y facilidad en el proceso de envío de los campos de las instrucciones traducidas por el ensamblador para guardarlos en la memoria principal. Esto se fundamenta en uno de los atributos definidos en la tabla 35 para la arquitectura de esta computadora: la mínima unidad de memoria direccionable, la cual, como se recordará, consiste en un byte. Por lo tanto, si se pretendía asignar un formato para cada uno de los campos de las instrucciones utilizando como base una unidad de información más pequeña que el byte como, por ejemplo, el bit, el ensamblador presentaba algunos inconvenientes de incompatibilidad para transmitir el campo traducido en código binario a fin de ser almacenado en una o varias direcciones de la memoria principal según correspondiera ya que, en un principio, existía la posibilidad de que no encajara correctamente en las mismas y dejara un hueco vacío que no se completaría por el siguiente campo. De este modo, el formato de la instrucción terminaría siendo incorrecto y la unidad de control tendría serios problemas para interpretar el código de operación y/o recuperar los operandos necesarios para ejecutar dicha instrucción. Además, si el usuario estuviera interesado en analizar el contenido de la memoria principal durante las distintas simulaciones que se realicen, le resultará mucho más sencilla la tarea de identificar cada uno de los campos de las instrucciones del programa si conoce al menos que jamás podrán coexistir dos campos diferentes en una misma dirección, ahorrándose así el esfuerzo de llevar a cabo un estudio pormenorizado bit a bit para diferenciar un campo del otro.

Tabla 41. Formatos de las instrucciones para la nueva CPU.

Instrucción	Byte 7	Byte 6	Byte 5	Byte 4	Byte 3	Byte 2	Byte 1	Byte 0
lb **lh** **lw**			ri	Inm (byte 1)	Inm (byte 0)	rd	Código de operación	Cabecera
sb **sh** **sw**			ri	Inm (byte 1)	Inm (byte 0)	rf	Código de operación	Cabecera
lf			ri	Inm (byte 1)	Inm (byte 0)	fd	Código de operación	Cabecera
sf			ri	Inm (byte 1)	Inm (byte 0)	ff	Código de operación	Cabecera
mff **tf** **ti**					ff	fd	Código de operación	Cabecera
mfr					rf	fd	Código de operación	Cabecera
mrf					ff	rd	Código de operación	Cabecera
dadd **daddu** **dsub** **dsubu** **dmul** **dmulu** **ddiv** **ddivu** **slt** **andr** **orr**				rg	rf	rd	Código de operación	Cabecera

xorr								
daddi **daddui** **slti** **andi** **ori** **xori** **dsli** **dsri** **dslsi** **dsrsi**	N (byte 3)	N (byte 2)	N (byte 1)	N (byte 0)	rf	rd	Código de operación	Cabecera
addf **subf** **mulf** **divf**				fg	ff	fd	Código de operación	Cabecera
ltf **lef** **eqf**					fg	ff	Código de operación	Cabecera
negr **notr**					rf	rd	Código de operación	Cabecera
dsl **dsr** **dsls** **dsrs**				rN	rf	rd	Código de operación	Cabecera
jmp **bfpt** **bfpf**					offNI (byte 1)	offNI (byte 0)	Código de operación	Cabecera
beq **bne**			offNI (byte 1)	offNI (byte 0)	rg	rf	Código de operación	Cabecera
beqz **bnez**				offNI (byte 1)	offNI (byte 0)	rf	Código de operación	Cabecera
call (*)					offNS (byte 1)	offNS (byte 0)	Código de operación	Cabecera

ret **iret** **cli** **sti** **pushfl** **popfl** **pushfp** **popfp** **pushra** **popra** **nop** **halt**							Código de operación	Cabecera
int (*)						NI	Código de operación	Cabecera
pushb (*) **pushh (*)** **pushw (*)**						rf	Código de operación	Cabecera
popb (*) **poph (*)** **popw (*)**						rd	Código de operación	Cabecera
inb (*) **inh (*)** **inw (*)**				offNP (byte 1)	offNP (byte 0)	rd	Código de operación	Cabecera
outb (*) **outh (*)** **outw (*)**				offNP (byte 1)	offNP (byte 0)	rf	Código de operación	Cabecera

Observaciones:

- (*): Formato sin implementar.
- Cabecera: byte inicial que contiene el tamaño en bytes de la instrucción (incluyendo la cabecera) para ayudar a la unidad de búsqueda de la CPU a determinar el incremento que deberá tener el registro IP para que pueda apuntar a la próxima instrucción a ejecutar.

- Código de operación: identificador único para cada instrucción a fin de que la unidad de control pueda interpretarlo y determinar los próximos pasos a seguir para completar la ejecución de dicha instrucción.
- rd: identificador del registro destino de la instrucción. Puede variar entre 0 (r0) y 15 (r15).
- rf: identificador del primer registro fuente de la instrucción. Puede variar entre 0 (r0) y 15 (r15).
- rg: identificador del segundo registro fuente de la instrucción. Puede variar entre 0 (r0) y 15 (r15).
- Inm: dirección base para determinar la dirección de memoria de datos a acceder para leer o escribir un valor en ella. Puede variar entre 4096 (1000H) y 8191 (1FFFH).
- ri: identificador del registro índice de la instrucción que contiene el desplazamiento a partir de la dirección base "Inm" para determinar la dirección de memoria de datos a acceder. Puede variar entre 0 (r0) y 15 (r15).
- fd: identificador del registro destino de la instrucción (punto flotante). Puede variar entre 16 (f0) y 31 (f15).
- ff: identificador del primer registro fuente de la instrucción (punto flotante). Puede variar entre 16 (f0) y 31 (f15).
- fg: identificador del segundo registro fuente de la instrucción (punto flotante). Puede variar entre 16 (f0) y 31 (f15).
- N: valor inmediato del tercer operando de la instrucción (segundo operando fuente). Puede variar entre 0 y 4.294.967.295 (sin signo) o bien –2.147.483.648 y 2.147.483.647 (con signo) según el tipo de instrucción.
- offNI: dirección de salto de la instrucción dentro de la memoria de instrucciones. Puede variar entre 8192 (2000H) y 12287 (2FFFH).
- offNS: dirección de inicio de la subrutina invocada por la instrucción. Puede variar entre 12288 (3000H) y 28671 (6FFFH).
- offNP: dirección de memoria asociada al puerto de E/S a acceder para leer o escribir un valor.
- NI: número de la instrucción por software a ejecutar. Puede variar entre 0 y 255.

3.1.5. Estructura de la memoria principal

Si el usuario estuviera interesado en estudiar el contenido de la memoria principal en tiempo de ejecución durante las pruebas que realice, también debería conocer la estructura de la misma, en particular las distintas secciones que la conforman, la naturaleza de la información que almacenan, su ubicación y sus delimitaciones (dirección inicial y final). De este modo, podrá determinar rápidamente qué dirección de memoria debería tomar como punto de referencia para localizar la información deseada en función de su tipo (dato, instrucción, subrutina, etc.). La estructura fue diseñada a partir del modelo ofrecido por el simulador MSX88 para la memoria principal de su propia computadora aunque, como se observará, la amplia mayoría de las secciones aún no se encuentran implementadas en este diseño.

Tabla 42. Estructura de la memoria principal diseñada para la nueva computadora.

Sección de memoria	Dirección inicial	Dirección final	Tamaño total
Interrupciones y Entrada/Salida (*)	0000	0FFF	4 kilobytes
Datos	1000	1FFF	4 kilobytes
Instrucciones	2000	2FFF	4 kilobytes
Subrutinas (*)	3000	6FFF	16 kilobytes
Pila (*)	7000	7FFF	4 kilobytes
Sistema Operativo (*)	8000	FFFF	32 kilobytes
Total	**0000**	**FFFF**	**64 kilobytes**

Observaciones:

- (*): Sección de memoria sin implementar.
- Las direcciones de memoria iniciales y finales se encuentran representadas en el sistema de numeración hexadecimal.

3.1.5.1. Programa de prueba

Una vez asimilada toda la información anterior el lector ya debería encontrarse en condiciones de determinar a partir de cualquier programa escrito en el lenguaje Assembler la disposición de cada uno de sus datos e instrucciones en la memoria principal, desde la/s

direcciones que ocuparían hasta el orden particular en el cual se guardarían cada uno de sus campos en ellas teniendo en cuenta principalmente las distintas secciones de la estructura de la memoria, el formato de las instrucciones y el mecanismo de almacenamiento utilizado por la computadora (little endian).

Se propone el siguiente programa de prueba para su análisis:

```
        .data
num1:   .word   –20
num2:   .hword  80
        .code
        lw r1, num1(r0)
        daddi r3, r2, –5
        addf f3, f1, f2
        halt
```

Figura 43. Programa de prueba propuesto para analizar la disposición de sus datos e instrucciones en la memoria principal de la nueva computadora.

Como se habrá podido observar, la sintaxis definida para la programación de código en el lenguaje Assembler que resulte compatible con este nuevo procesador es muy similar a la utilizada por el simulador WinMIPS64 para su propia CPU, con algunas pequeñas modificaciones en la denominación de algunas instrucciones y tipos de datos. Se profundizará más en este apartado en la subsección “3.3. Programación Assembler” de este informe.

Ahora mismo se prestará una especial importancia a explicar paso a paso el proceso deductivo realizado para determinar la disposición del programa de la figura anterior en la memoria principal:

1. Sección de datos (directiva .data):
    a. Dirección inicial: 1000H.
    b. Primer dato (num1):
        i. Se trata de un número de 32 bits con signo (directiva .word).
        ii. Su dirección inicial coincidirá con la dirección inicial de la sección de datos, es decir, la número 1000H.

iii. Su longitud es de 32 bits, es decir, 4 bytes. Al contener cada dirección de memoria exactamente 1 byte de información, este dato ocupará 4 direcciones: desde la 1000H hasta la 1003H inclusive.

iv. La computadora utiliza el sistema de numeración conocido como complemento a dos para representar valores con signo (para números sin signo usa el sistema binario sin signo). Por lo tanto, para determinar la representación correcta del valor a almacenar deberá realizarse la conversión a dicho sistema, siempre recordando que la longitud de la variable es de 32 bits. El resultado de dicha conversión tendría que ser el siguiente: $-20_{10} = 11111111111111111111111111101100_{Ca2}$.

c. Segundo dato (num2):

i. Se trata de un número de 16 bits con signo (directiva .hword).

ii. Su dirección inicial será aquélla ubicada inmediatamente a continuación de la dirección final del dato anterior, es decir, la número 1004H.

iii. Su longitud es de 16 bits, es decir, 2 bytes. Al contener cada dirección de memoria exactamente 1 byte de información, este dato ocupará 2 direcciones: desde la 1004H hasta la 1005H inclusive.

iv. La computadora utiliza el sistema de numeración conocido como complemento a dos para representar valores con signo (para números sin signo usa el sistema binario sin signo). Por lo tanto, para determinar la representación correcta del valor a almacenar deberá realizarse la conversión a dicho sistema, siempre recordando que la longitud de la variable es de 16 bits. El resultado de dicha conversión tendría que ser el siguiente: $80_{10} = 0000000001010000_{Ca2}$.

2. Sección de código del programa principal (directiva .code):

a. Dirección inicial: 2000H.

b. Primera instrucción (lw):

i. La dirección inicial de esta instrucción coincidirá con la dirección inicial de la sección de código del programa principal, es decir, la número 2000H.

ii. Cabecera:

- Su dirección inicial coincidirá con la dirección inicial de esta instrucción, es decir, la número 2000H.
- Su longitud es de 8 bits, es decir, 1 byte. Al contener cada dirección de memoria exactamente 1 byte de información, este campo ocupará 1 dirección: la número 2000H.
- Tamaño de la instrucción:
  - ❖ Cabecera: 1 byte.
  - ❖ Código de operación: 1 byte.
  - ❖ Registro destino (rd): 1 byte.
  - ❖ Dirección base de memoria de datos (Inm): 2 bytes.
  - ❖ Registro índice (ri): 1 byte.
  - ❖ Total: 6 bytes.
- Como se recordará, el campo cabecera contiene el tamaño total en bytes de la instrucción almacenada (incluyendo dicha cabecera). Por lo tanto, el valor almacenado será la representación del total obtenido en el inciso anterior en el sistema de numeración binario sin signo (al ser siempre el tamaño de la instrucción un valor positivo, la computadora utiliza este sistema para representarlo). Una vez realizada la conversión, sin olvidar que este campo ocupa 8 bits, puede obtenerse el siguiente resultado: $6_{10} = 00000110_2$.

iii. Código de operación:

- Su dirección inicial será aquélla ubicada inmediatamente a continuación de la dirección final del campo anterior, es decir, la número 2001H.
- Su longitud es de 8 bits, es decir, 1 byte. Al contener cada dirección de memoria exactamente 1 byte de información, este campo ocupará 1 dirección: la número 2001H.
- El código de operación a almacenar para esta instrucción se obtiene de la tabla 38 de este informe: 00001000.

iv. Registro destino (rd):

- Su dirección inicial será aquélla ubicada inmediatamente a continuación de la dirección final del campo anterior, es decir, la número 2002H.
- Su longitud es de 8 bits, es decir, 1 byte. Al contener cada dirección de memoria exactamente 1 byte de información, este campo ocupará 1 dirección: la número 2002H.
- Al tratarse r1 del registro destino de esta instrucción, se debe calcular la representación de su identificador de registro, 1, en el sistema de numeración binario sin signo (al ser siempre el identificador de registro un valor natural, la computadora utiliza este sistema para representarlo). Una vez realizada la conversión, sin olvidar que este campo ocupa 8 bits, puede obtenerse el siguiente resultado: $1_{10} = 00000001_2$.

v. Dirección base de memoria de datos (Inm):

- Su dirección inicial será aquélla ubicada inmediatamente a continuación de la dirección final del campo anterior, es decir, la número 2003H.
- Su longitud es de 16 bits, es decir, 2 bytes. Al contener cada dirección de memoria exactamente 1 byte de información, este dato ocupará 2 direcciones: desde la 2003H hasta la 2004H inclusive.
- Al hacerse referencia en este campo a la etiqueta o nombre del primer dato del programa (num1), el valor a almacenar será la dirección inicial de dicho dato en memoria, 1000H, representada en el sistema de numeración binario sin signo (al ser siempre la dirección de memoria un valor natural, la computadora utiliza este sistema para representarla). Una vez realizada la conversión, sin olvidar que este campo ocupa 16 bits, puede obtenerse el siguiente resultado: $1000_{16} = 0001000000000000_2$.

vi. Registro índice (ri):

- Su dirección inicial será aquélla ubicada inmediatamente a continuación de la dirección final del campo anterior, es decir, la número 2005H.
- Su longitud es de 8 bits, es decir, 1 byte. Al contener cada dirección de memoria exactamente 1 byte de información, este campo ocupará 1 dirección: la número 2005H.
- Al tratarse r0 del registro índice de esta instrucción, se debe calcular la representación de su identificador de registro, 0, en el sistema de numeración binario sin signo (al ser siempre el identificador de registro un valor natural, la computadora utiliza este sistema para representarlo). Una vez realizada la conversión, sin olvidar que este campo ocupa 8 bits, puede obtenerse el siguiente resultado: $0_{10} = 00000000_2$.

vii. La longitud total de esta instrucción es de 6 bytes. Al contener cada dirección de memoria exactamente 1 byte de información, dicha instrucción ocupará 6 direcciones: desde la 2000H hasta la 2005H inclusive.

c. Segunda instrucción (daddi):

i. La dirección inicial de esta instrucción será aquélla ubicada inmediatamente a continuación de la dirección final de la instrucción anterior, es decir, la número 2006H.

ii. Cabecera:

- Su dirección inicial coincidirá con la dirección inicial de esta instrucción, es decir, la número 2006H.
- Su longitud es de 8 bits, es decir, 1 byte. Al contener cada dirección de memoria exactamente 1 byte de información, este campo ocupará 1 dirección: la número 2006H.
- Tamaño de la instrucción:
  - ❖ Cabecera: 1 byte.
  - ❖ Código de operación: 1 byte.
  - ❖ Registro destino (rd): 1 byte.
  - ❖ Primer registro fuente (rf): 1 byte.

- ❖ Segundo operando fuente (N): 4 bytes.
- ❖ Total: 8 bytes.

- Como se recordará, el campo cabecera contiene el tamaño total en bytes de la instrucción almacenada (incluyendo dicha cabecera). Por lo tanto, el valor almacenado será la representación del total obtenido en el inciso anterior en el sistema de numeración binario sin signo (al ser siempre el tamaño de la instrucción un valor positivo, la computadora utiliza este sistema para representarlo). Una vez realizada la conversión, sin olvidar que este campo ocupa 8 bits, puede obtenerse el siguiente resultado: $8_{10} = 00001000_2$.

iii. Código de operación:

- Su dirección inicial será aquélla ubicada inmediatamente a continuación de la dirección final del campo anterior, es decir, la número 2007H.
- Su longitud es de 8 bits, es decir, 1 byte. Al contener cada dirección de memoria exactamente 1 byte de información, este campo ocupará 1 dirección: la número 2007H.
- El código de operación a almacenar para esta instrucción se obtiene de la tabla 38 de este informe: 00011001.

iv. Registro destino (rd):

- Su dirección inicial será aquélla ubicada inmediatamente a continuación de la dirección final del campo anterior, es decir, la número 2008H.
- Su longitud es de 8 bits, es decir, 1 byte. Al contener cada dirección de memoria exactamente 1 byte de información, este campo ocupará 1 dirección: la número 2008H.
- Al tratarse r3 del registro destino de esta instrucción, se debe calcular la representación de su identificador de registro, 3, en el sistema de numeración binario sin signo (al ser siempre el identificador de registro un valor natural, la computadora utiliza este sistema para representarlo). Una vez realizada la conversión, sin olvidar que este campo

ocupa 8 bits, puede obtenerse el siguiente resultado: 3_{10} = 00000011_2.

v. Primer registro fuente (rf):
    - Su dirección inicial será aquélla ubicada inmediatamente a continuación de la dirección final del campo anterior, es decir, la número 2009H.
    - Su longitud es de 8 bits, es decir, 1 byte. Al contener cada dirección de memoria exactamente 1 byte de información, este campo ocupará 1 dirección: la número 2009H.
    - Al tratarse r2 del primer registro fuente de esta instrucción, se debe calcular la representación de su identificador de registro, 2, en el sistema de numeración binario sin signo (al ser siempre el identificador de registro un valor natural, la computadora utiliza este sistema para representarlo). Una vez realizada la conversión, sin olvidar que este campo ocupa 8 bits, puede obtenerse el siguiente resultado: 2_{10} = 00000010_2.

vi. Segundo operando fuente (N):
    - Su dirección inicial será aquélla ubicada inmediatamente a continuación de la dirección final del campo anterior, es decir, la número 200AH.
    - Su longitud es de 32 bits, es decir, 4 bytes. Al contener cada dirección de memoria exactamente 1 byte de información, este campo ocupará 4 direcciones: desde la 200AH hasta la 200DH inclusive.
    - Al tratarse –5 del valor inmediato del segundo operando fuente de esta instrucción, se debe calcular la representación del mismo en el sistema de numeración conocido como complemento a dos (al existir posibilidades de que el valor inmediato sea negativo, la computadora utiliza este sistema para representarlo). Una vez realizada la conversión, sin olvidar que este campo ocupa 32 bits, puede obtenerse el

siguiente resultado: -5_{10} = $11111111111111111111111111111011_{Ca2}$.

vii. La longitud total de esta instrucción es de 8 bytes. Al contener cada dirección de memoria exactamente 1 byte de información, dicha instrucción ocupará 8 direcciones: desde la 2006H hasta la 200DH inclusive.

d. Tercera instrucción (addf):

i. La dirección inicial de esta instrucción será aquélla ubicada inmediatamente a continuación de la dirección final de la instrucción anterior, es decir, la número 200EH.

ii. Cabecera:

- Su dirección inicial coincidirá con la dirección inicial de esta instrucción, es decir, la número 200EH.
- Su longitud es de 8 bits, es decir, 1 byte. Al contener cada dirección de memoria exactamente 1 byte de información, este campo ocupará 1 dirección: la número 200EH.
- Tamaño de la instrucción:
  - ❖ Cabecera: 1 byte.
  - ❖ Código de operación: 1 byte.
  - ❖ Registro destino (fd): 1 byte.
  - ❖ Primer registro fuente (ff): 1 byte.
  - ❖ Segundo registro fuente (fg): 1 byte.
  - ❖ Total: 5 bytes.
- Como se recordará, el campo cabecera contiene el tamaño total en bytes de la instrucción almacenada (incluyendo dicha cabecera). Por lo tanto, el valor almacenado será la representación del total obtenido en el inciso anterior en el sistema de numeración binario sin signo (al ser siempre el tamaño de la instrucción un valor positivo, la computadora utiliza este sistema para representarlo). Una vez realizada la conversión, sin olvidar que este campo ocupa 8 bits, puede obtenerse el siguiente resultado: $5_{10} = 00000101_2$.

iii. Código de operación:

- Su dirección inicial será aquélla ubicada inmediatamente a continuación de la dirección final del campo anterior, es decir, la número 200FH.
- Su longitud es de 8 bits, es decir, 1 byte. Al contener cada dirección de memoria exactamente 1 byte de información, este campo ocupará 1 dirección: la número 200FH.
- El código de operación a almacenar para esta instrucción se obtiene de la tabla 38 de este informe: 00011100.

iv. Registro destino (fd):

- Su dirección inicial será aquélla ubicada inmediatamente a continuación de la dirección final del campo anterior, es decir, la número 2010H.
- Su longitud es de 8 bits, es decir, 1 byte. Al contener cada dirección de memoria exactamente 1 byte de información, este campo ocupará 1 dirección: la número 2010H.
- Al tratarse f3 del registro destino de esta instrucción, se debe calcular la representación de su identificador de registro, 19, en el sistema de numeración binario sin signo (al ser siempre el identificador de registro un valor natural, la computadora utiliza este sistema para representarlo). Una vez realizada la conversión, sin olvidar que este campo ocupa 8 bits, puede obtenerse el siguiente resultado: $19_{10} = 00010011_2$.

v. Primer registro fuente (ff):

- Su dirección inicial será aquélla ubicada inmediatamente a continuación de la dirección final del campo anterior, es decir, la número 2011H.
- Su longitud es de 8 bits, es decir, 1 byte. Al contener cada dirección de memoria exactamente 1 byte de información, este campo ocupará 1 dirección: la número 2011H.
- Al tratarse f1 del primer registro fuente de esta instrucción, se debe calcular la representación de su identificador de registro, 17, en el sistema de numeración binario sin signo

(al ser siempre el identificador de registro un valor natural, la computadora utiliza este sistema para representarlo). Una vez realizada la conversión, sin olvidar que este campo ocupa 8 bits, puede obtenerse el siguiente resultado: $17_{10} = 00010001_2$.

vi. Segundo registro fuente (fg):
    - Su dirección inicial será aquélla ubicada inmediatamente a continuación de la dirección final del campo anterior, es decir, la número 2012H.
    - Su longitud es de 8 bits, es decir, 1 byte. Al contener cada dirección de memoria exactamente 1 byte de información, este campo ocupará 1 dirección: la número 2012H.
    - Al tratarse f2 del segundo registro fuente de esta instrucción, se debe calcular la representación de su identificador de registro, 18, en el sistema de numeración binario sin signo (al ser siempre el identificador de registro un valor natural, la computadora utiliza este sistema para representarlo). Una vez realizada la conversión, sin olvidar que este campo ocupa 8 bits, puede obtenerse el siguiente resultado: $18_{10} = 00010010_2$.

vii. La longitud total de esta instrucción es de 5 bytes. Al contener cada dirección de memoria exactamente 1 byte de información, dicha instrucción ocupará 5 direcciones: desde la 200EH hasta la 2012H inclusive.

e. Cuarta y última instrucción (halt):
    i. La dirección inicial de esta instrucción será aquélla ubicada inmediatamente a continuación de la dirección final de la instrucción anterior, es decir, la número 2013H.
    ii. Cabecera:
        - Su dirección inicial coincidirá con la dirección inicial de esta instrucción, es decir, la número 2013H.

- Su longitud es de 8 bits, es decir, 1 byte. Al contener cada dirección de memoria exactamente 1 byte de información, este campo ocupará 1 dirección: la número 2013H.
- Tamaño de la instrucción:
  - ❖ Cabecera: 1 byte.
  - ❖ Código de operación: 1 byte.
  - ❖ Total: 2 bytes.
- Como se recordará, el campo cabecera contiene el tamaño total en bytes de la instrucción almacenada (incluyendo dicha cabecera). Por lo tanto, el valor almacenado será la representación del total obtenido en el inciso anterior en el sistema de numeración binario sin signo (al ser siempre el tamaño de la instrucción un valor positivo, la computadora utiliza este sistema para representarlo). Una vez realizada la conversión, sin olvidar que este campo ocupa 8 bits, puede obtenerse el siguiente resultado: $2_{10} = 00000010_2$.

iii. Código de operación:

- Su dirección inicial será aquélla ubicada inmediatamente a continuación de la dirección final del campo anterior, es decir, la número 2014H.
- Su longitud es de 8 bits, es decir, 1 byte. Al contener cada dirección de memoria exactamente 1 byte de información, este campo ocupará 1 dirección: la número 2014H.
- El código de operación a almacenar para esta instrucción se obtiene de la tabla 38 de este informe: 10000001.

iv. La longitud total de esta instrucción es de 2 bytes. Al contener cada dirección de memoria exactamente 1 byte de información, dicha instrucción ocupará 2 direcciones: desde la 2013H hasta la 2014H inclusive.

Tabla 44. Disposición de los datos e instrucciones del programa de prueba de la figura anterior en la memoria principal de la nueva computadora.

Dirección de memoria	Contenido	Campo del dato o instrucción
1000	11101100	n1 (byte 0)
1001	11111111	n1 (byte 1)
1002	11111111	n1 (byte 2)
1003	11111111	n1 (byte 3)
1004	01010000	n1 (byte 0)
1005	00000000	n1 (byte 1)
1006	...	...
1007	...	...
...	...	...
2000	00000110	Cabecera
2001	00001000	Código de operación
2002	00000001	rd
2003	00000000	Inm (byte 0)
2004	00010000	Inm (byte 1)
2005	00000000	ri
2006	00001000	Cabecera
2007	00011001	Código de operación
2008	00000011	rd
2009	00000010	rf
200A	11111011	N (byte 0)
200B	11111111	N (byte 1)
200C	11111111	N (byte 2)
200D	11111111	N (byte 3)
200E	00000101	Cabecera
200F	00011100	Código de operación
2010	00000011	fd
2011	00000001	ff
2012	00000010	fg
2013	00000010	Cabecera

2014	10000001	Código de operación
2015	...	...
2016	...	...
...	...	...

Observaciones:

- Las direcciones de memoria se encuentran representadas en el sistema de numeración hexadecimal.

3.1.6. Ciclo de ejecución: segmentación del cauce

El ciclo de ejecución propuesto para las instrucciones en este procesador se encuentra basado en el tipo de arquitectura elegido para la nueva computadora (RISC) y la compatibilidad con una posible implementación de la segmentación en el cauce sin que existan inconvenientes. Para ello, se determinó, en primer lugar, en consonancia con la arquitectura seleccionada, que el flujo de ejecución debía ser fijo para cualquier instrucción sin importar su tipo ni su denominación particular (recordar que en el diseño original podía variar en función del tamaño del operando) ya que, en caso contrario, sería imposible realizar la división del ciclo de ejecución en etapas para segmentar el cauce si las mismas pudieran ejecutarse o no según la instrucción en cuestión.

Las etapas en las cuales fue fraccionado el ciclo de ejecución con el objetivo de implementar la segmentación en el cauce del procesador coinciden exactamente con la división ofrecida por el simulador WinMIPS64, siendo ésta también la más comúnmente mencionada al momento de introducir esta técnica en las explicaciones teóricas sobre este tema. No debe olvidarse que, incluso si ya se encuentra plenamente incorporada y operativa la segmentación en la CPU, las etapas siempre se ejecutarán de manera secuencial dentro de una misma instrucción. A continuación se describe en términos generales cada una de ellas:

Tabla 45. Etapas del ciclo de ejecución de las instrucciones en el nuevo procesador.

Etapa	Tareas a ejecutar (en orden secuencial)
1. Búsqueda	Acceder al registro IP y obtener a partir de él la dirección inicial en memoria principal de la próxima instrucción a ejecutar.
	Acceder a la dirección de memoria adquirida en el paso anterior y obtener a partir de ella y las contiguas todos los campos de la próxima instrucción.
	Acceder al registro IR y actualizarlo con el código de operación de la nueva instrucción obtenida en el paso anterior.
	Transmitir todos los campos de la nueva instrucción, excepto la cabecera y el código de operación, a la unidad de control de la CPU para que ésta pueda continuar con la siguiente etapa del ciclo de ejecución.
	Incrementar el registro IP a partir del valor contenido en el campo cabecera de la instrucción adquirida en el paso anterior para que apunte a la próxima instrucción a ejecutar.
2. Decodificación	Acceder al registro IR y obtener a partir de él el código de operación de la nueva instrucción a ejecutar, actualizado en la etapa anterior del ciclo de ejecución.
	Decodificar el código de operación obtenido en el paso anterior e interpretar a partir de él la denominación de la nueva instrucción a ejecutar.
	En caso de ser necesario, acceder a los registros de uso general de la CPU asociados con los identificadores recibidos de la etapa anterior del ciclo de ejecución y obtener a partir de ellos los operandos de la nueva instrucción.
	Enviar los operandos obtenidos en el paso anterior a las unidades de la CPU encargadas de ejecutar las etapas restantes del ciclo de ejecución junto con la información necesaria para indicarle a cada una de ellas, en caso de corresponder, la acción a realizar cuando le llegue el turno de ejecutar su etapa asignada del ciclo de ejecución de la nueva instrucción.
3.1. Ejecución	Determinar a partir de la información recibida de la etapa de

	decodificación, en caso de corresponder, la operación aritmético-lógica requerida por la nueva instrucción para ser ejecutada en esta etapa del ciclo de ejecución.
	Ejecutar la operación aritmético-lógica determinada en el paso anterior.
	Acceder al registro FLAGS y actualizar las banderas que deban ser modificadas como consecuencia de la ejecución de la operación aritmético-lógica en el paso anterior.
	En caso de tratarse la nueva instrucción de un salto condicional, acceder nuevamente al registro FLAGS y obtener a partir de él el valor actual de cada una de sus banderas.
	A partir de las banderas obtenidas en el paso anterior, analizar aquélla que corresponda para determinar si el salto debe ser tomado o no.
	Si la respuesta determinada en el paso anterior fue afirmativa, actualizar el registro IP con la dirección de salto indicada en la nueva instrucción.
	Enviar el resultado de la operación aritmético-lógica obtenido en esta etapa a las unidades de la CPU encargadas de ejecutar las etapas restantes del ciclo de ejecución para que puedan utilizarlo en caso de requerirlo para llevar a cabo sus respectivas tareas.
3.2. Ejecución en punto flotante	Determinar a partir de la información recibida de la etapa de decodificación, en caso de corresponder, la operación aritmético-lógica en punto flotante requerida por la nueva instrucción para ser ejecutada en esta etapa del ciclo de ejecución.
	Ejecutar la operación aritmético-lógica en punto flotante determinada en el paso anterior.
	Acceder al registro FPFLAGS y actualizar las banderas que deban ser modificadas como consecuencia de la ejecución de la operación aritmético-lógica en punto flotante en el paso anterior.
	En caso de tratarse la nueva instrucción de una comparación, acceder nuevamente al registro FPFLAGS y obtener a partir de él el valor actual de cada una de sus banderas.
	A partir de las banderas obtenidas en el paso anterior, analizar aquélla que corresponda para determinar el valor con el cual deberá

	actualizarse la bandera F.
	Accceder una vez más al registro FPFLAGS y actualizar la bandera F con el valor determinado en el paso anterior.
	Enviar el resultado de la operación aritmético-lógica en punto flotante obtenido en esta etapa a las unidades de la CPU encargadas de ejecutar las etapas restantes del ciclo de ejecución para que puedan utilizarlo en caso de requerirlo para llevar a cabo sus respectivas tareas.
4. Acceso a memoria	Determinar a partir de la información recibida de la etapa de decodificación, en caso de corresponder, el tipo de acceso a memoria de la computadora requerido por la nueva instrucción para ser llevado a cabo en esta etapa del ciclo de ejecución.
	Acceder a la dirección de memoria de datos o el puerto del periférico de E/S determinado en el paso anterior para realizar sobre él una operación de lectura o escritura según corresponda.
	En caso de haberse tratado de una operación de lectura, enviar el valor obtenido a la unidad de almacenamiento en registro para que pueda utilizarlo en caso de requerirlo para ejecutar la última etapa del ciclo de ejecución de la nueva instrucción.
5. Almacenamiento en registro	Determinar a partir de la información recibida de la etapa de decodificación, en caso de corresponder, el tipo de acceso a registro de uso general de la CPU requerido por la nueva instrucción para ser ejecutada en esta etapa del ciclo de ejecución.
	Acceder al registro de uso general determinado en el paso anterior y actualizarlo con el valor que corresponda según lo indicado en la instrucción actual. Dicho valor habrá sido recibido de alguna de las etapas anteriores del ciclo de ejecución: decodificación, ejecución/ejecución en punto flotante o acceso a memoria en función de la instrucción ejecutada.

Observaciones:

- Las etapas 3.1. (ejecución) y 3.2. (ejecución en punto flotante) son mutuamente excluyentes, es decir, no pueden pertenecer simultáneamente al ciclo de ejecución de una misma instrucción. Esto se debe a que, como su nombre lo indica, la etapa de ejecución en

punto flotante está orientada únicamente a instrucciones aritméticas o de transferencia de datos que involucran la realización de operaciones aritméticas en punto flotante en la FPU (unidad de punto flotante) de la nueva CPU; mientras que la etapa de ejecución forma parte del ciclo de ejecución de la totalidad del resto de las instrucciones del repertorio del procesador, incluyendo a aquéllas que no requieren realizar ningún tipo de acción en esta etapa.

- Para realizar la descripción ofrecida en la tabla anterior se consideró un escenario de ejecución considerablemente más simplificado con respecto al que se presentaría en una simulación real de este procesador. Más concretamente, se omitió la presencia de un fenómeno adverso asociado a la implementación de la segmentación en el cauce: posibles atascos en la ejecución de las instrucciones debido a diversos conflictos ya sea por el acceso y utilización de los limitados recursos de hardware de la computadora, el orden de lectura y escritura de algún dato contenido en uno de los registros de uso general de la CPU o la determinación de la próxima instrucción a ejecutar debido a la existencia de una instrucción de transferencia de control que podría alterar el normal flujo de ejecución del programa. Cada uno de estos tipos de atascos debe ser detectado con anticipación para detener temporalmente la ejecución del procesador durante uno o varios ciclos de reloj hasta que el conflicto sea resuelto y de esta manera prevenir que el programa en ejecución ofrezca un comportamiento y/o resultados incorrectos debido al inadecuado manejo de estos conflictos. Obviamente, dicha labor de detección tiene que llevarse a cabo durante el ciclo de ejecución de cada una de las instrucciones del programa, lo cual involucrará ciertas modificaciones en algunas etapas de la segmentación para que, además de todas las tareas más conocidas y tradicionales mencionadas en la tabla, posean la capacidad de conocer algunos datos puntuales sobre el estado actual de la ejecución del programa (datos pendientes de escritura, futuros accesos a memoria de datos, instrucciones de transferencia de control actualmente en ejecución), junto con la lógica necesaria para determinar si puede producirse un atasco y, de ser así, tomar todas las medidas necesarias para gestionarlo. Los detalles más técnicos sobre las metodologías seleccionadas para la incorporación de estas tareas no serán ofrecidos en este informe por considerarse demasiado complejos e intrincados y ultimadamente irrelevantes para el usuario interesado simplemente en programar la CPU desde un nivel de abstracción más elevado como lo es el del lenguaje Assembler. Debería bastar con informar que la mayoría de estas labores de detección son realizadas por componentes de la CPU completamente ajenos a las unidades encargadas de ejecutar las tareas más conocidas de cada una de las etapas del

ciclo de ejecución, enunciadas en la tabla anterior. No obstante, huelga decir que, si el usuario continúa ávido de interiorizarse aún más sobre los pormenores del funcionamiento de estos "detectores", es libre de estudiar directamente el código utilizado por el diseñador para la descripción de la computadora y obtener por sí mismo sus propias conclusiones.

A continuación se definen las unidades de la CPU encargadas de ejecutar cada una de las etapas del ciclo de ejecución mencionadas y descriptas en la tabla precedente a ésta. Como se detalló en una de las observaciones anteriores, aquí se está considerando un escenario simplificado en el cual no intervienen ninguno de las distintos tipos de atascos asociados a la implementación de la segmentación en el cauce debido a potenciales conflictos respecto a recursos de hardware, dependencias de datos o transferencias de control. Por lo tanto, sólo se tienen en cuenta las tareas enunciadas en la tabla anterior omitiendo cualquier labor vinculada a los mecanismos de detección de atascos utilizados en la mayoría de las etapas, los cuales son gestionados y ejecutados en su mayor parte por unidades del procesador independientes y separadas de aquéllas definidas en la próxima tabla.

Tabla 46. Unidad encargada de ejecutar cada etapa del ciclo de ejecución de las instrucciones en el nuevo procesador.

Etapa	Unidad de la CPU encargada de su ejecución
1. Búsqueda	Unidad de búsqueda
2. Decodificación	Unidad de control
3.1. Ejecución	Unidad aritmético-lógica (ALU)
3.2. Ejecución en punto flotante	Unidad de punto flotante (FPU)
4. Acceso a memoria	Unidad de acceso a memoria
5. Almacenamiento en registro	Unidad de almacenamiento en registro

Aunque ya se hizo referencia a este apartado durante la definición de las tareas a ejecutar en las distintas etapas que constituyen el ciclo de ejecución de las instrucciones en el nuevo procesador (tabla 45), sería conveniente incluir una tabla adicional en la cual se enuncien sintéticamente los dispositivos de almacenamiento que pueden ser accedidos en cada

una de dichas etapas, diferenciando el tipo de operación con el cual podría encontrarse asociado cada posible acceso: lectura y/o escritura de información (ya sea datos o instrucciones, la distinción entre ambos tipos de información resulta irrelevante para el nivel de este caso de estudio) según corresponda. Esto será útil más adelante cuando se necesite detallar una importante consideración que se debió tener en cuenta para describir el diseño de la arquitectura para la nueva computadora.

Tabla 47. Dispositivos de almacenamiento que pueden ser accedidos en cada etapa del ciclo de ejecución de las instrucciones en la nueva computadora.

Dispositivo de almacenamiento		**Etapa**	**Tipo de acceso**
Banco de registros de uso general		2. Decodificación	Lectura
		5. Almacenamiento en registro	Escritura
Registros de uso particular	Registro IR	1. Búsqueda	Escritura
		2. Decodificación	Lectura
	Registro IP	1. Búsqueda	Lectura/Escritura
		2. Decodificación	Escritura
		3.1. Ejecución	Escritura
	Registro FLAGS	3.1. Ejecución	Lectura/Escritura
	Registro FPFLAGS	2. Decodificación	Lectura
		3.2. Ejecución en punto flotante	Lectura/Escritura
Memoria principal	Sección de datos	4. Acceso a memoria	Lectura/Escritura
	Sección de instrucciones	1. Búsqueda	Lectura

Ahora bien, como podrá recordarse, en la primera tabla de este informe se mencionó como uno de los principios fundamentales de la arquitectura RISC la imperiosa necesidad de que cada instrucción requiera exactamente un único ciclo de reloj para ejecutarse, sin importar su denominación o complejidad exigida para ser llevada a cabo. Asimismo, el lector se acordará que, tal y como se refleja en la tabla 35, el procesador de este nuevo diseño, al haberse inclinado el diseñador por una arquitectura para él más cercana al tipo RISC, se

mantuvo fiel a dicho principio. No obstante, se debe considerar que, al implementar la segmentación en el cauce, cada instrucción ya no podrá ser tenida en cuenta como una unidad compacta e inseparable sino que ahora cada una de las etapas en las cuales fue dividido su ciclo de ejecución conformará su propia entidad independiente, por lo cual el principio original de la arquitectura RISC en lugar de aplicarse sobre cada instrucción deberá destinarse a cada etapa particular del ciclo, es decir, a partir de este momento serán estas últimas las que deberán requerir, cada una, exactamente un único ciclo de reloj para ser ejecutada. La metodología que suele utilizarse a fin de cumplir esta regla es determinar el tiempo de ejecución necesario para ejecutar cada etapa y optar por el más lento de todos para que sea el nuevo ciclo de reloj, de manera tal de garantizar que todas las etapas puedan completar su ejecución durante su espacio de tiempo asignado (algunas de las etapas más veloces permanecerán ociosas durante un pequeño intervalo de tiempo hasta que finalice el período de reloj).

Una particularidad a tener en cuenta sería la situación de la etapa 3.2. del ciclo de ejecución de las instrucciones, denominada “ejecución en punto flotante”. En el párrafo anterior se definió uno de los principios fundamentales de la arquitectura RISC, el cual determina que cada instrucción o, en caso de implementarse la segmentación en el cauce, cada etapa del ciclo de ejecución debe requerir exactamente un único ciclo de reloj para ser ejecutada. Sin embargo, al diseñarse el ciclo de ejecución para el nuevo procesador, se determinó que la etapa 3.2., al encontrarse destinada a la ejecución de operaciones aritméticas en punto flotante en la FPU (unidad de punto flotante), y al considerarse ésta como una labor considerablemente más compleja y, por lo tanto, más lenta que cualquiera de las demás tareas del ciclo de ejecución de una instrucción, necesitará cuatro ciclos de reloj para completarse en lugar de uno como el resto de las etapas, procurando así que la descripción de la CPU sea tan cercana a la realidad como resulte posible.

Una alternativa para evitar esta anomalía podría haber sido utilizar el tiempo de ejecución de la etapa 3.2. como el nuevo ciclo de reloj para la ejecución del cauce segmentado del procesador. Sin embargo, una importante desventaja presentada como consecuencia de esta elección sería que, al existir una diferencia tan elevada respecto a los tiempos de ejecución del resto de las etapas, estas últimas permanecerían ociosas en cada período durante un intervalo de tiempo demasiado considerable como para ser ignorado, repercutiendo así muy negativamente sobre el rendimiento general del procesador.

En cambio, se optó por subdividir esta etapa en cuatro etapas distintas, las cuales, tanto ellas como la FPU encargada de ejecutarlas, están definidas y configuradas de manera

tal de permitirle al procesador seguir cumpliendo con los principios fundamentales tanto de la arquitectura RISC como de la segmentación del cauce: cada una de estas cuatro etapas requerirá exactamente un único ciclo de reloj para ser ejecutada y, además, múltiples instrucciones podrán ejecutarse simultáneamente en un mismo ciclo de reloj, siempre y cuando se trate, obviamente, de etapas distintas del ciclo de ejecución (recordar que no se debe confundir segmentación con paralelismo: en esta CPU jamás podrá ejecutarse al mismo tiempo una misma etapa para múltiples instrucciones bajo ninguna circunstancia).

A continuación se presentarán por separado los dos posibles ciclos de ejecución que pueden presentar las instrucciones del repertorio del nuevo procesador en función de su necesidad de utilizar la FPU para realizar operaciones aritméticas en punto flotante.

Tabla 48. Ciclo de ejecución de las instrucciones que no requieren realizar operaciones aritméticas en punto flotante en la FPU del nuevo procesador.

Etapa	Denominación
1.	Búsqueda
2.	Decodificación
3.1.	Ejecución
4.	Acceso a memoria
5.	Almacenamiento en registro

Tabla 49. Ciclo de ejecución de las instrucciones que requieren realizar operaciones aritméticas en punto flotante en la FPU del nuevo procesador.

Etapa	Denominación
1.	Búsqueda
2.	Decodificación
3.2.1.	Ejecución en punto flotante (Etapa 1)
3.2.2.	Ejecución en punto flotante (Etapa 2)
3.2.3.	Ejecución en punto flotante (Etapa 3)
3.2.4.	Ejecución en punto flotante (Etapa 4)
4.	Acceso a memoria
5.	Almacenamiento en registro

Observaciones:

- Recordar que las etapas 3.2.1., 3.2.2., 3.2.3. y 3.2.4. surgen simplemente de la subdivisión de la etapa 3.2. (ejecución en punto flotante) en cuatro etapas independientes a fin de preservar el cumplimiento por parte de la CPU de los principios fundamentales tanto de la arquitectura RISC como de de la segmentación del cauce.
- Huelga decir que una misma instrucción del repertorio jamás podrá exhibir un tipo de ciclo durante una de sus ejecuciones y el otro tipo en una instancia distinta, ya que en todas las ocasiones su función y sus tareas permanecerán inalterables.
- Como habrá podido adivinarse, al encontrarse su etapa de ejecución subdividida en cuatro etapas distintas, cada una con un tiempo de ejecución igual a un ciclo de reloj, el ciclo de ejecución de una instrucción que necesite utilizar la FPU del procesador para realizar una operación aritmética en punto flotante será considerablemente más lento, extenso y complejo que el de una que no requiera hacer uso de ella. Más precisamente, la diferencia entre los tiempos de ejecución totales de ambos ciclos de ejecución será exactamente igual a tres ciclos de reloj (ocho vs. cinco ciclos de reloj).

Aquí se definirán específicamente las instrucciones del repertorio del nuevo procesador que requieren realizar operaciones aritméticas en punto flotante en la FPU de la CPU y que, por lo tanto, presentarán su etapa de ejecución subdividida en cuatro etapas distintas, cada una de ellas con un tiempo de ejecución igual a un ciclo de reloj, tal como lo indica la segunda de las dos tablas anteriores, siendo el tiempo de ejecución total de esta etapa igual a cuatro ciclos de reloj. Asimismo, se incluyen el resto de las instrucciones que no necesitan realizar ningún tipo de operación aritmético-lógica o, en caso de requerirlo, sólo intervendrían en ella operandos enteros, por lo que únicamente tendrían que utilizar la ALU del procesador. En consecuencia, tal como lo muestra la primera de las dos tablas anteriores, su etapa de ejecución no presentaría subdivisión alguna y el tiempo de ejecución de la misma sería igual a un solo ciclo de reloj.

Tabla 50. Tipo de etapa de ejecución perteneciente al ciclo de ejecución de cada instrucción del repertorio del nuevo procesador.

Tipo de instrucción	Instrucción	Etapa de ejecución
Transferencia de Datos	lb r_d, $Inm(r_i)$	3.1. Ejecución
	sb r_f, $Inm(r_i)$	3.1. Ejecución
	lh r_d, $Inm(r_i)$	3.1. Ejecución
	sh r_f, $Inm(r_i)$	3.1. Ejecución
	lw r_d, $Inm(r_i)$	3.1. Ejecución
	sw r_f, $Inm(r_i)$	3.1. Ejecución
	lf f_d, $Inm(r_i)$	3.1. Ejecución
	sf f_f, $Inm(r_i)$	3.1. Ejecución
	mff f_d, f_f	3.1. Ejecución
	mfr f_d, r_f	3.1. Ejecución
	mrf r_d, f_f	3.1. Ejecución
	tf f_d, f_f	3.2. Ejecución en punto flotante
	ti f_d, f_f	3.2. Ejecución en punto flotante
Aritmética	dadd r_d, r_f, r_g	3.1. Ejecución
	daddi r_d, r_f, N	3.1. Ejecución
	daddu r_d, r_f, r_g	3.1. Ejecución
	daddui r_d, r_f, N	3.1. Ejecución
	addf f_d, f_f, f_g	3.2. Ejecución en punto flotante
	dsub r_d, r_f, r_g	3.1. Ejecución
	dsubu r_d, r_f, r_g	3.1. Ejecución
	subf f_d, f_f, f_g	3.2. Ejecución en punto flotante
	dmul r_d, r_f, r_g	3.1. Ejecución
	dmulu r_d, r_f, r_g	3.1. Ejecución
	mulf f_d, f_f, f_g	3.2. Ejecución en punto flotante
	ddiv r_d, r_f, r_g	3.1. Ejecución
	ddivu r_d, r_f, r_g	3.1. Ejecución
	divf f_d, f_f, f_g	3.2. Ejecución en punto flotante
	slt r_d, r_f, r_g	3.1. Ejecución
	slti r_d, r_f, N	3.1. Ejecución
	ltf f_f, f_g	3.2. Ejecución en punto flotante

	lef f_f, f_g	3.2. Ejecución en punto flotante
	eqf f_f, f_g	3.2. Ejecución en punto flotante
	negr r_d, r_f	3.1. Ejecución
Lógica	andr r_d, r_f, r_g	3.1. Ejecución
	andi r_d, r_f, N	3.1. Ejecución
	orr r_d, r_f, r_g	3.1. Ejecución
	ori r_d, r_f, N	3.1. Ejecución
	xorr r_d, r_f, r_g	3.1. Ejecución
	xori r_d, r_f, N	3.1. Ejecución
	notr r_d, r_f	3.1. Ejecución
Desplazamiento de Bits	dsl r_d, r_f, r_N	3.1. Ejecución
	dsli r_d, r_f, N	3.1. Ejecución
	dsr r_d, r_f, r_N	3.1. Ejecución
	dsri r_d, r_f, N	3.1. Ejecución
	dsls r_d, r_f, r_N	3.1. Ejecución
	dslsi r_d, r_f, N	3.1. Ejecución
	dsrs r_d, r_f, r_N	3.1. Ejecución
	dsrsi r_d, r_f, N	3.1. Ejecución
Transferencia de Control	jmp $offN_I$	3.1. Ejecución
	beq r_f, r_g, $offN_I$	3.1. Ejecución
	bne r_f, r_g, $offN_I$	3.1. Ejecución
	beqz r_f, $offN_I$	3.1. Ejecución
	bnez r_f, $offN_I$	3.1. Ejecución
	bfpt $offN_I$	3.1. Ejecución
	bfpf $offN_I$	3.1. Ejecución
Subrutinas	call $offN_S$	3.1. Ejecución
	ret	3.1. Ejecución
Manejo de Interrupciones	int N_I	3.1. Ejecución
	iret	3.1. Ejecución
	cli	3.1. Ejecución
	sti	3.1. Ejecución
Manejo de la Pila	pushb r_f	3.1. Ejecución
	popb r_d	3.1. Ejecución
	pushh r_f	3.1. Ejecución

	poph r_d	3.1. Ejecución
	pushw r_f	3.1. Ejecución
	popw r_d	3.1. Ejecución
	pushfl	3.1. Ejecución
	popfl	3.1. Ejecución
	pushfp	3.1. Ejecución
	popfp	3.1. Ejecución
	pushra	3.1. Ejecución
	popra	3.1. Ejecución
Entrada/Salida	inb r_d, $offN_P$	3.1. Ejecución
	outb r_f, $offN_P$	3.1. Ejecución
	inh r_d, $offN_P$	3.1. Ejecución
	outh r_f, $offN_P$	3.1. Ejecución
	inw r_d, $offN_P$	3.1. Ejecución
	outw r_f, $offN_P$	3.1. Ejecución
Control	nop	3.1. Ejecución
	halt	3.1. Ejecución

3.1.7. Arquitecturas von Neumann y Harvard

Ahora supóngase que se ejecuta un programa de prueba conformado por instrucciones que en ningún caso requieren utilizar la FPU del procesador para realizar operaciones aritméticas en punto flotante. Por lo tanto, todas presentarán un ciclo de ejecución conformado por cinco etapas, cada una con una duración de un ciclo de reloj cada una: la etapa de ejecución no presenta a su vez ninguna clase de subdivisión en etapas, por lo cual sólo necesita un ciclo de ejecución para ejecutarse. A continuación se describirá la ejecución de dicho programa en forma segmentada a través del tiempo, es decir, aprovechando la división del ciclo de ejecución en etapas separadas e independientes originada a partir de la implementación de la segmentación en el cauce del procesador de manera tal de poder ejecutar múltiples instrucciones simultáneamente, siempre tratándose de distintas etapas del ciclo de ejecución de cada una de ellas.

Tiempo / Instrucción	t1	t2	t3	t4	t5	t6	t7	t8	t9	t10	...
I_1	F	D	X	M	W						
I_2		F	D	X	M	W					
I_3			F	D	X	M	W				
I_4				F	D	X	M	W			
I_5					F	D	X	M	W		
...											

Figura 51. Progreso de la ejecución segmentada de una serie de instrucciones en la nueva CPU a través del tiempo que no requieren utilizar la FPU del procesador para realizar operaciones aritméticas en punto flotante.

Tabla 52. Referencias para poder interpretar los ciclos de ejecución de las instrucciones representados en la figura anterior.

Etapa	Letra	Color
1. Búsqueda	F (Fetch)	Rojo
2. Decodificación	D (Decode)	Violeta
3.1. Ejecución	X (Execute)	Verde
4. Acceso a memoria	M (Memory Access)	Azul
5. Almacenamiento en registro	W (Writeback)	Rosa

Aquí es donde desempeña un papel crucial la información proporcionada por la tabla 47, en particular aquélla referida a los posibles accesos a la memoria principal de la computadora ya que, como se recordará, originalmente se había propuesto la arquitectura von Neumann como plantilla para describir el diseño de la arquitectura de la nueva computadora. A continuación se ofrece un diagrama de la arquitectura von Neumann en el cual se incluyen los componentes principales que plantea para el diseño de la computadora junto con el sistema de interconexión dispuesto para transmitir datos entre los mismos (para este diseño de arquitectura no existe distinción alguna entre datos e instrucciones a este nivel de abstracción) a través de canales de comunicación conocidos como buses.

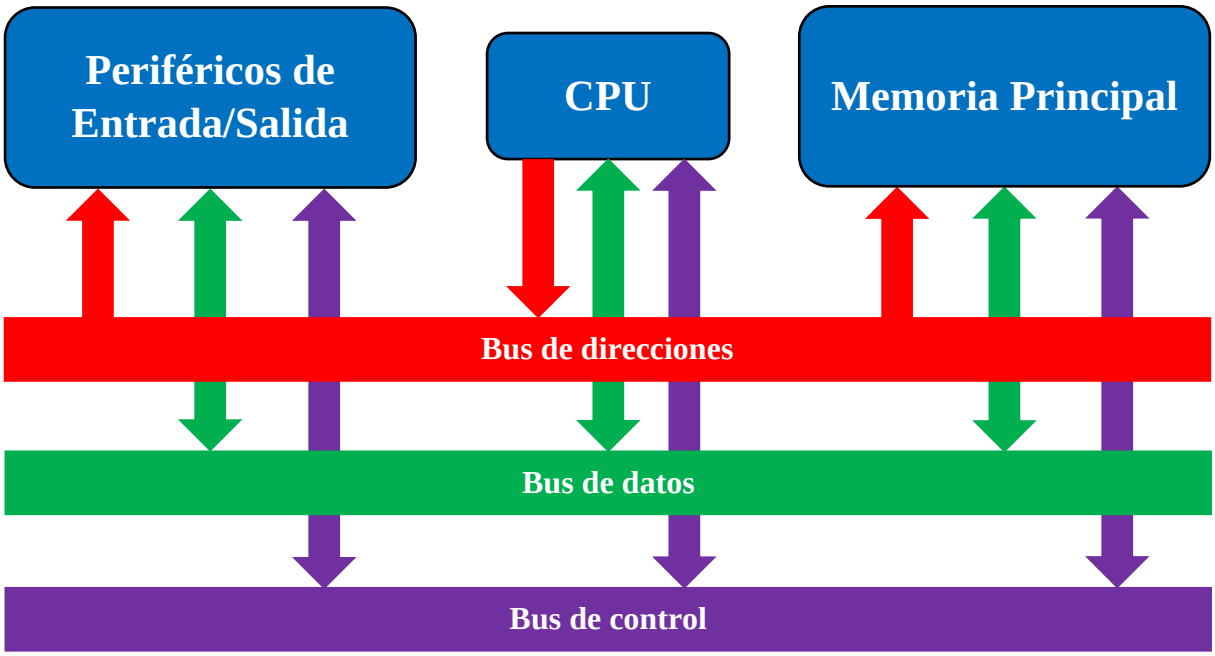

Figura 53. Diagrama de la arquitectura von Neumann.

Como se observará, en la arquitectura von Neumann la memoria principal de la computadora se considera como un único dispositivo de almacenamiento compacto sin distinción alguna entre sus distintas secciones. Por consiguiente, tanto la sección de datos de la misma como la sección de instrucciones tendrán en común el mismo sistema compartido de buses de direcciones, datos y control para poder comunicarse con la CPU para transmitir o recibir información según corresponda.

Si se recuerda la información brindada en la tabla 47, existen dos etapas distintas del ciclo de ejecución de las instrucciones que podrían requerir acceso a la memoria principal de la computadora para realizar operaciones de lectura o escritura: la etapa de búsqueda (memoria de instrucciones) y la de acceso a memoria (memoria de datos). Sin embargo, al establecer la arquitectura von Neumann que las secciones de datos y de instrucciones de la memoria principal deben compartir el mismo sistema de buses, la lógica consecuencia es la imposibilidad de que las dos etapas previamente mencionadas puedan ejecutarse simultáneamente en un mismo ciclo de reloj ya que una de ellas deberá esperar a que la otra libere la utilización del sistema de buses compartido, sin importar que ambas etapas estén intentando acceder a dos secciones distintas e independientes de memoria.

La siguiente figura representa la ejecución de la misma secuencia de instrucciones que en el caso de la figura 51, pero ahora la CPU encargada de ejecutarlas formará parte de una

arquitectura von Neumann, por lo cual la restricción establecida en el párrafo anterior no podrá ser ignorada.

Tiempo / Instrucción	t1	t2	t3	t4	t5	t6	t7	t8	t9	t10	t11	t12	...
I_1	F	D	X	M	W								
I_2		F	D	X	M	W							
I_3			F	D	X	M	W						
I_4				~~X~~	~~M~~	~~W~~	F	D	X	M	W		
I_5								F	D	X	M	W	
...													

Figura 54. Progreso de la ejecución segmentada de una serie de instrucciones idéntica a aquélla ejecutada en la figura 51, ahora en la CPU de una computadora cuya arquitectura sea von Neumann.

Observaciones:

- Las celdas tachadas representan la suspensión temporal de la ejecución de la etapa de búsqueda de la instrucción I_4 debido a que el sistema de buses compartido que debería utilizar para acceder a la memoria de instrucciones de la computadora ya se encuentra actualmente reservado por la etapa de acceso a memoria de las instrucciones I_1, I_2 e I_3 respectivamente para poder acceder a la memoria de datos (independientemente de si la instrucción en cuestión efectivamente requiera o no realizar el acceso a memoria de datos para leer o escribir en ella). Únicamente cuando no queden por ejecutar más etapas de acceso a memoria por parte de instrucciones anteriores podrá la instrucción I_4 comenzar la ejecución de su etapa de búsqueda.

Tabla 55. Referencias para poder interpretar los ciclos de ejecución de las instrucciones representados en la figura anterior.

Etapa	Letra	Color
1. Búsqueda	F (Fetch)	Rojo
2. Decodificación	D (Decode)	Violeta
3.1. Ejecución	X (Execute)	Verde
4. Acceso a memoria	M (Memory Access)	Azul
5. Almacenamiento en registro	W (Writeback)	Rosa

A partir de la situación representada en la figura anterior, claramente puede determinarse que la arquitectura von Neumann por sí sola, debido a la limitación en su sistema de buses previamente decripta, no puede ofrecer el soporte necesario para ejecutar las instrucciones en forma segmentada tal y como se presentó en la figura 51, ya que eventualmente siempre el cauce del procesador se verá temporalmente suspendido hasta que todas las etapas de acceso a memoria pendientes de ser ejecutadas se completen. Además, como se podrá adivinar, esta situación ocurrirá muy probablemente en reiteradas ocasiones durante la ejecución del programa, por lo cual el rendimiento final del procesador se verá seriamente afectado y será muy inferior al esperado cuando se decidió implementar originalmente la segmentación en el cauce.

La solución a este problema se encuentra en incorporar una ligera modificación al diseño de la arquitectura originalmente planteado, utilizando como plantilla la arquitectura Harvard, la cual consiste básicamente en una versión moderna de la arquitectura von Neumann con algunas pequeñas alteraciones orientadas precisamente a resolver el inconveniente descripto anteriormente.

Ahora se mostrará un diagrama de la arquitectura Harvard de una manera similar a la representación de la arquitectura von Neumann ofrecida en la figura 53 para que el lector pueda comparar ambas versiones y así comprender el motivo por el cual esta nueva versión sí es capaz de ofrecer el soporte necesario para que la ejecución segmentada del cauce del procesador sea idéntica a la inicialmente esperada (figura 51): múltiples instrucciones pueden ejecutarse simultáneamente en cualquiera de sus etapas, siempre y cuando estas últimas no coincidan entre sí.

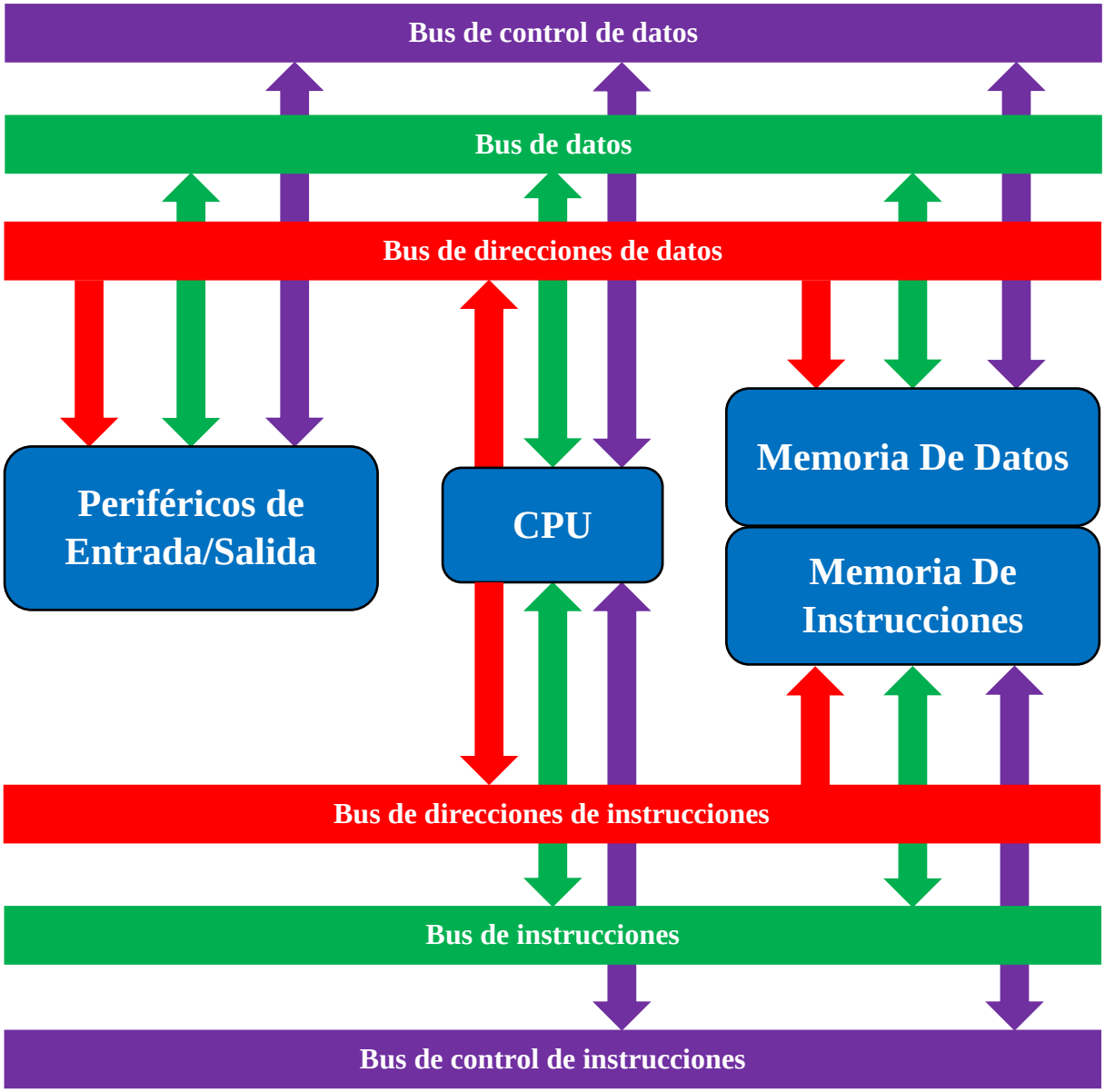

Figura 56. Diagrama de la arquitectura Harvard.

La figura anterior refleja claramente que la mayor diferencia entre las arquitecturas von Neumann y Harvard consiste en la división propuesta por esta última para la memoria principal en dos secciones independientes: una para almacenar datos y otra para preservar instrucciones. Al plantear esta distinción, también determina la necesidad de implementar un sistema de buses de direcciones, control y datos (o instrucciones según corresponda, para este diseño de arquitectura sí existe una diferenciación apreciable entre datos e instrucciones a este nivel de abstracción) separado para cada sección a fin de que la CPU sea capaz de acceder a ambas simultáneamente para realizar operaciones de lectura o escritura.

De este modo puede lograr evitarse la restricción impuesta por la arquitectura von Neumann respecto al acceso simultáneo por parte de la CPU a la memoria de datos y de instrucciones de la computadora. Por lo tanto, naturalmente se arribó a la conclusión de que la alternativa más recomentable y por la cual se optó para describir el diseño de la nueva PC era la arquitectura Harvard; cuya implementación implicó que ambas secciones de memoria, al poseer cada una un sistema de buses independiente asociado a ella, podrán ser accedidas al mismo tiempo por la CPU sin inconvenientes y libremente, sin que ninguna operación a una de las secciones deba esperar a que otra operación actualmente siendo llevada a cabo en la otra sección sea completada para poder comenzar a realizarse. Así, las etapas de búsqueda y de acceso a memoria del ciclo de ejecución del cauce segmentado podrán ser ejecutadas al mismo tiempo sin inconvenientes, por lo que la ejecución del cauce será idéntica a la esperada inicialmente, representada en la figura 51.

Por su parte, se recordará que durante la etapa de acceso a memoria la instrucción puede requerir acceder a la memoria de datos o bien a un puerto de memoria asociado a alguno de los periféricos de entrada/salida de la computadora. De esta manera, puede concluirse que los accesos a ambos dispositivos de almacenamiento son mutuamente excluyentes: al poder ser realizados únicamente en la misma etapa del ciclo de ejecución, y teniendo en cuenta que en una ejecución segmentada del cauce del procesador jamás podrán ejecutarse múltiples instrucciones simultáneamente en la misma etapa del ciclo, bajo ninguna circunstancia la CPU podrá intentar acceder al mismo tiempo a la memoria de datos y a algún periférico de entrada/salida para leer o escribir datos. Por lo tanto, en el diseño de la arquitectura Harvard se ha determinado que tanto la memoria de datos como los periféricos de entrada/salida pueden compartir el mismo sistema de buses de interconexión sin ningún tipo de conflictos por el uso de dicho sistema que puedan demorar temporalmente la ejecución de las instrucciones del programa.

3.1.8. Riesgos de la segmentación y atascos del cauce

El último concepto teórico que deberá ser presentado en esta subsección del informe (“3.1. Arquitectura”) antes de proceder a realizar algunas demostraciones con programas de prueba escritos para ser ejecutados por el nuevo procesador está vinculado a un tópico que ya fue mencionado previamente en reiteradas ocasiones pero sin profundizar demasiado sobre el tema hasta este momento.

Si el lector ya se encuentra familiarizado con algunos conceptos básicos del área de arquitectura de computadoras, entonces conocerá que la ejecución segmentada de un programa en el procesador implicará que existirán múltiples instrucciones ejecutándose en forma simultánea en el mismo, aunque siempre en etapas diferentes. La primera conclusión que se puede extraer de esta definición es que el número de instrucciones en ejecución en cualquier instante dado se encontrará siempre limitado por la cantidad de etapas en las cuales se haya optado por dividir el ciclo de ejecución de las instrucciones. Aquí se plantea la diferencia más notoria entre segmentación y paralelismo: en este último sí pueden ejecutarse múltiples instrucciones simultáneamente en la misma etapa, por lo cual no existiría en un principio restricción alguna para el número de instrucciones en ejecución en un instante dado (obviamente, en la práctica siempre se presentará alguna limitación vinculada principalmente a los recursos de los cuales dispone la computadora para ejecutar varias instrucciones al mismo tiempo).

Por otra parte, la ejecución simultánea de múltiples instrucciones en el procesador, aunque repercutirá favorablemente sobre el rendimiento del mismo ya que el tiempo de ejecución del programa debería disminuir considerablemente, también provocará posibles conflictos entre las distintas instrucciones que se encuentren en ejecución en un instante dado por distintos motivos que pueden estar vinculados tanto a los recursos de hardware de la computadora como a la lógica de ejecución (software) del programa ejecutado. Estos conflictos, si no son atendidos y resueltos de alguna manera, probablemente desencadenen un comportamiento no deseado por parte del programa que, a su vez, provocará que sus resultados finales sean incorrectos. Por lo tanto, es fundamental aprender los tipos de conflictos o riesgos conocidos que pueden presentarse durante la ejecución segmentada del procesador ya que su detección y resolución será tan importante como lo es la implementación de la propia segmentación para que esta última funcione correctamente.

En cualquiera de los casos que se describirán a continuación, las demoras que se vean obligadas a presentar en su ejecución las instrucciones afectadas por conflictos con instrucciones anteriores del programa, ya sea a raíz de la utilización de un recurso de hardware, una dependencia de datos o el resultado de la evaluación de una condición de salto, se conocen como atascos en el cauce del procesador. Cada atasco representará un retardo de un ciclo de reloj tanto en la instrucción conflictiva como en todas las que se encuentran a continuación de ella en la ejecución del programa:

- Riesgos estructurales: están vinculados a conflictos por los recursos de hardware de la computadora. Como éstos son limitados en número y capacidad, únicamente podrán ser

accedidos por un número reducido de instrucciones al mismo tiempo. No obstante, sólo en casos muy especiales se presentarán este tipo de riesgos, ya que, al encontrarse cada etapa del ciclo de ejecución de las instrucciones asociada a un recurso distinto de la PC (ver tabla 46 del informe), para que existan conflictos estructurales debería suceder que no se cumpliera la secuencia de ejecución natural del programa, es decir, que no todas las instrucciones terminen en el orden en el cual comenzaron a ejecutarse. Ahora bien, para determinar los casos en los cuales podría no mantenerse el orden de ejecución natural del programa, se debería tomar como punto de partida la división en etapas del ciclo de ejecución optada para implementar la segmentación del cauce en este procesador en particular (ver tablas 48 y 49 del informe). Entonces, puede plantearse la situación de una instrucción actualmente en ejecución que necesita realizar una operación aritmética en punto flotante en la FPU de la CPU. A continuación de ella, el programa presenta una serie de instrucciones que no requieren llevar a cabo ningún tipo de operación aritmética en punto flotante. Por lo tanto, cabría esperar que, si no existiera inconveniente alguno, la primera instrucción, al encontrarse dividida en más etapas y, por lo tanto, requerir un mayor número de ciclos de reloj para ejecutarse que el resto de las instrucciones del programa, finalice con posterioridad a algunas de las instrucciones que no hagan ningún uso de la FPU de la CPU y estén ubicadas en sus proximidades más inmediatas. Éste sería claramente un caso en el cual no se cumple el orden natural del programa: la primera instrucción se estaría completando después de que terminen de ejecutarse una o varias instrucciones ubicadas a continuación de ella y cuyas ejecuciones comenzarían con posterioridad al inicio de la suya propia. Por lo tanto, aquí existiría un alto riesgo de que surjan conflictos entre la primera instrucción y alguna de las demás instrucciones del programa ya que en algún momento ambas podrían intentar acceder simultáneamente a alguno de los recursos de hardware de la computadora, por lo cual una de las dos debería esperar a que la otra complete su acceso y utilización del recurso en conflicto para proceder a realizar su propia operación sobre él. Después de llevar a cabo numerosas pruebas, se ha determinado que los recursos de hardware de esta computadora en particular que pueden ser objeto de conflictos de acceso simultáneo por parte de múltiples instrucciones son los siguientes:

- Unidad de acceso a memoria / Memoria de datos: éstas son respectivamente la unidad de la CPU encargada de gestionar la operación de acceso y el recurso de hardware de la computadora en conflicto. Como se recordará, estos dos dispositivos se encuentran asociados a la etapa de acceso a memoria del ciclo de

ejecución de las instrucciones: si dos de estas últimas pretenden acceder a la memoria de datos simultáneamente, entonces existirá un conflicto entre ambas: una de ellas se verá obligada a detener su ejecución hasta que la otra haya completado su acceso y utilización del recurso en conflicto (memoria de datos). En la figura 57 se presenta un ejemplo del progreso de la ejecución de un programa a través del tiempo cuando existe este tipo de atascos en el cauce del procesador.

- ❖ Unidad de almacenamiento en registro / Banco de registros de uso general: éstas son respectivamente la unidad de la CPU encargada de gestionar la operación de acceso y el recurso de hardware de la computadora en conflicto. Como se recordará, estos dos dispositivos se encuentran asociados a la etapa de almacenamiento en registro del ciclo de ejecución de las instrucciones: si dos de estas últimas pretenden acceder al banco de registros de uso general simultáneamente para realizar operaciones de escritura sobre él, entonces existirá un conflicto entre ambas: una de ellas se verá obligada a detener su ejecución hasta que la otra haya completado su acceso y utilización del recurso en conflicto (banco de registros de uso general). En la figura 58 se presenta un ejemplo del progreso de la ejecución de un programa a través del tiempo cuando existe este tipo de atascos en el cauce del procesador.

- Riesgos por dependencias de datos: otra de las consecuencias no deseadas de la implementación de la segmentación en el cauce del procesador es una posible alteración en el orden de acceso a los registros de la CPU, ya sea de uso general o particular, con el objetivo de realizar operaciones de lectura o bien escritura sobre él, respecto a la secuencia de acceso que se desarrollaría si el programa se ejecutara en forma secuencial. No se debe confundir esta situación con el orden de ejecución natural del programa: en este caso incluso si las instrucciones finalizaran en el mismo orden en el cual comenzaron a ejecutarse, podría ocurrir el riesgo de que una instrucción intentara acceder a un registro antes de que una instrucción anterior a ella haya podido llevar a cabo su respectivo acceso a él para realizar la operación que necesite. Si en ambas instrucciones la operación en cuestión se tratara de una lectura, al no encontrarse involucrada una actualización del registro, entonces indudablemente no existiría bajo ningún concepto la posibilidad de un comportamiento inesperado y/o una respuesta incorrecta del programa por inconsistencias en el estado del valor de dicho registro. Sin embargo, si alguna de las operaciones consistiera en una escritura, podrían presentarse algunos problemas debido a varias circunstancias distintas, las cuales se discutirán a continuación (tener en cuenta en todo

momento la división en etapas del ciclo de ejecución de las instrucciones propuesta en las tablas 48 y 49 del informe):

- Lectura Después de Escritura (Read After Write – RAW): en este caso, el conflicto se debe a la existencia de una instrucción en el programa que está intentando acceder a un registro del procesador para leer un dato que una instrucción anterior a ella todavía no ha podido almacenar en él. Por lo tanto, el dato que estaría leyendo en el momento actual no sería el correcto ya que no coincidiría con aquél que ya debería haber sido escrito por la instrucción anterior pero aún no lo fue, lo cual a su vez provocaría un comportamiento inesperado por parte del programa ejecutado. En consecuencia, la única solución posible para este riesgo es establecer una demora en la ejecución de la instrucción afectada hasta que la instrucción anterior alcance y finalice la etapa en la cual accederá al registro conflictivo para escribir en él el dato que corresponda. Con respecto a la identificación particular de este registro, se detectaron dos situaciones particulares:
  - Registro de uso general: ésta sería la situación más común, en la cual una instrucción se encuentra en la segunda etapa de su ciclo de ejecución (decodificación) intentando acceder al banco de registros de uso general del procesador para leer un registro que todavía posee pendiente una operación de escritura por parte de una instrucción anterior. Por lo tanto, deberá demorarse la ejecución tanto de la instrucción que pretende realizar la lectura como de todas las que le siguen en el programa hasta que la instrucción anterior haya completado su escritura pendiente, la cual, como se recordará, debería llevarse a cabo en la quinta y última etapa de su ciclo de ejecución (almacenamiento en registro). En la figura 59 se presenta un ejemplo del progreso de la ejecución de un programa a través del tiempo cuando existe este tipo de atascos en el cauce del procesador.
  - Registro de uso particular: aquí básicamente se presenta el mismo fenómeno que en el caso anterior, con la peculiaridad de que el registro en conflicto no pertenece al banco de registros de uso general del procesador, sino que se trata de un registro de uso particular: el registro de banderas FPFLAGS, asociado al estado de las operaciones aritméticas en punto flotante realizadas en la FPU. Más precisamente, la bandera conflictiva sería la de Punto Flotante (bandera F), ya que éste sería un caso excepcional, pero no por ello menos importante, en el cual una instrucción

de transferencia de control condicional necesitaría obtener en su segunda etapa (decodificación) el estado actual de la bandera F para evaluar una condición y determinar así si el salto planteado debe ser tomado o no. Ahora bien, si se recuerda el repertorio de instrucciones del procesador, las únicas instrucciones que podrían tener esta función son exactamente dos en particular: “bfpt” y “bfpf”. Por su parte, todas las instrucciones del repertorio que deban realizar operaciones aritméticas en punto flotante en la FPU tendrán que actualizar el registro FPFLAGS una vez completada su respectiva operación. Esta actualización puede o no incluir a la bandera F en función de la tarea de la instrucción ejecutada. Sin embargo, como la dependencia de datos se establece sobre la totalidad del registro FPFLAGS sin importar la bandera en particular que deba ser leída, si existiera una actualización pendiente de dicho registro por cualquier instrucción anterior, sin importar su naturaleza o las banderas específicas a sobrescribir, entonces la instrucción de transferencia de control condicional tendría que detener temporalmente su ejecución hasta que la operación de actualización del registro de banderas se haya completado. Asimismo, si se tiene en cuenta la división en etapas del ciclo de ejecución propuesta para segmentar el cauce de este procesador, esta sobrescritura del registro FPFLAGS se llevaría a cabo en el cuarto ciclo de la tercera etapa (ejecución en punto flotante), por lo cual, si se lo compara con la situación anterior, en la cual el registro de uso general en conflicto se actualizaba recién en la última etapa del ciclo de ejecución de la instrucción (almacenamiento en registro), aquí el número de ciclos demorados sería inferior, siempre y cuando las instrucciones en cuestión realicen alguna operación aritmética en punto flotante en la FPU y su ciclo de ejecución requiera una cantidad de ciclos mayor para completarse que una instrucción que no involucre de ningún modo el uso de la FPU. En la figura 60 se presenta un ejemplo del progreso de la ejecución de un programa a través del tiempo cuando existe este tipo de atascos en el cauce del procesador.

- Escritura Después de Lectura (Write After Read – WAR): este tipo de conflicto se presentaría únicamente en caso de que una instrucción del programa pretendiera actualizar uno de los registros de uso general del procesador que aún mantuviera

una operación de lectura pendiente por parte de una instrucción anterior. Si existiera esta situación, entonces dicha instrucción anterior al realizar su lectura no estaría obteniendo el dato esperado debido a que el mismo ya habría sido sobrescrito previa e incorrectamente por la instrucción posterior, desencadenando así una respuesta inadecuada por parte del programa en ejecución. Así como para los riesgos ya descriptos anteriormente, la solución más adecuada para resolver este conflicto sería una demora en la ejecución de la instrucción que pretende almacenar un nuevo valor en el registro conflictivo hasta que la instrucción anterior haya logrado completar su operación de lectura sobre él. De este modo, todas las instrucciones realizan sus accesos a los registros respetando su posición dentro del programa, tal y como lo harían si éste se ejecutara de manera secuencial, garantizándose así la consistencia del estado de los datos de los registros en cualquier instante de la ejecución del programa. Por otra parte, puede determinarse fácilmente a partir del estudio de la división en etapas del ciclo de ejecución propuesta para segmentar el cauce de este procesador que las etapas específicas en las cuales se presentaría el conflicto serían la quinta y última (almacenamiento en registro) de la instrucción que pretendería realizar la escritura y la segunda (decodificación) de la instrucción anterior que necesitaría llevar a cabo la lectura. Por lo tanto, al encontrarse ambas etapas tan alejadas en el tiempo (la segunda etapa del ciclo de ejecución de una instrucción se llevará a cabo muy anteriormente a la realización de la quinta y última etapa del ciclo de una instrucción posterior en el programa), podrá arribarse a la conclusión de que esta clase de riesgos es la más inusual de todas, incluyendo todos los tipos conocidos y aquí presentados (estructurales, datos, control y fin), ya que únicamente podrían existir conflictos desde este tipo en caso de que se alterara el orden de ejecución de las instrucciones en el programa. No se está haciendo referencia aquí al orden de ejecución natural del programa, sino a un reordenamiento y reubicación de las instrucciones llevados a cabo deliberadamente por el ensamblador. Un posible objetivo buscado con este reposicionamiento podría ser mejorar el rendimiento del procesador a través de la implementación de una de las técnicas más conocidas para resolver riesgos de control, conocida como salto retardado. En ella, esencialmente, el ensamblador comienza seleccionando algunas instrucciones del programa que no posean relación o dependencia alguna con la condición de salto evaluada en la instrucción de transferencia de control que provoca el riesgo de

control a resolver. Luego, en función del tipo de instrucción, es decir, si el salto es condicional o incondicional, el ensamblador determina la cantidad de ciclos que demorará en evaluarse la condición de salto a partir de que la misma es detectada por la unidad de control del procesador en la segunda etapa del ciclo de ejecución de la instrucción (decodificación). Cada uno de estos ciclos de demora se conoce en esta situación como hueco de retardo o delay slot, y es aquí donde el ensamblador procurará introducir las instrucciones seleccionadas al principio, una en cada hueco a fin de que el procesador se mantenga la mayor cantidad de tiempo posible realizando trabajo útil en lugar de permanecer ocioso hasta que logre resolverse la condición de salto, sin comprometer en ningún momento la correcta ejecución del programa ya que, como se recordará, las instrucciones elegidas eran independientes de la condición evaluada. No obstante, al no poseer este procesador soporte para esta técnica de solución de riesgos de control, a pesar de las numerosas pruebas realizadas no logró detectarse en ninguna situación la presencia de un conflicto de tipo WAR, por lo que se decidió directamente omitir la implementación de este último para esta versión del proyecto. En la figura 61 se presenta un ejemplo del progreso de la ejecución de un programa a través del tiempo cuando existe este tipo de atascos en el cauce del procesador.

- Escritura Después de Escritura (Write After Write – WAW): la situación representada en este tipo de riesgo involucra, de una forma similar a los casos previos, dos instrucciones actualmente en ejecución en un programa: una instrucción posterior cuya intención es realizar una actualición del valor de un registro de uso general del procesador, y una anterior que todavía mantiene pendiente una solicitud de escritura sobre el mismo registro. Por lo tanto, si la instrucción posterior llevara a cabo la actualización que en el presente momento está pretendiendo efectuar, entonces en algún instante posterior la instrucción anterior concretaría eventualmente su operación de escritura pendiente, por lo cual el valor final del registro durante el resto de la ejecución del programa sería el correspondiente a esta última operación. Esta situación tendrá como consecuencia un estado incorrecto en el registro involucrado ya que, en una ejecución secuencial del programa, el valor actual de un registro siempre se encontrará determinado única y exclusivamente por la última instrucción que, siguiendo el orden de ejecución natural (todas las instrucciones finalizan en el mismo orden en el cual comenzaron a ejecutarse), haya realizado una actualización sobre él. En esta última

sentencia se encuentra la clave de la cuestión: como se recordará, en la ejecución segmentada de un programa, existe la posibilidad de que no todos los ciclos de ejecución de las instrucciones coincidan entre sí, ya que algunos, más largos y lentos y divididos en un mayor número de etapas, requieren realizar operaciones aritméticas en punto flotante en la FPU del procesador y otros, más cortos y rápidos, no precisan de ninguna manera utilizar la FPU para sus operaciones. De esta manera, un programa segmentado podría no seguir su orden de ejecución natural. Supóngase la situación de un programa actualmente en ejecución que en un determinado momento presenta una serie de dos instrucciones: una de ellas con un ciclo más largo de ejecución, que requiere realizar una operación aritmética en punto flotante en la FPU, e inmediatamente a continuación de ella otra instrucción con un ciclo corto de ejecución, que no necesita utilizar la FPU para llevar a cabo ninguna tarea. Entonces, asumiendo que no se encuentra involucrado ningún otro tipo de riesgo de segmentación, la primera instrucción, a pesar de haber comenzado su ejecución antes, finalizará después de haberse completado la ejecución de la segunda instrucción debido a la diferencia en el número de etapas en ambos ciclos y, en consecuencia, en el tiempo de ejecución total de ambas instrucciones. Ésta sería una clásica situación en la cual podría ocurrir este conflicto: si ambas instrucciones pretendieran realizar una operación de escritura sobre el mismo registro de uso general del procesador, entonces, si no se llevara a cabo ninguna acción preventiva, la segunda instrucción actualizaría el registro antes que la primera, provocando el error en el estado del registro ya mencionado anteriormente. Por lo tanto, para impedir este riesgo, esta segunda instrucción, aunque ya se encuentre lista para realizar su escritura, se verá obligada a detener temporalmente su ejecución y la del todo el cauce del procesador hasta que la primera instrucción se haya completado. Esto se debe a que, como se recordará, es recién en la última etapa del ciclo de ejecución (almacenamiento en registro) cuando las instrucciones llevan a cabo, si correspondiera, la actualización del registro de uso general del procesador que deban realizar. Finalmente, podrá observarse que éste es el único riesgo por dependencia de datos en el cual las dos instrucciones en conflicto presentan sus problemas en la misma etapa (almacenamiento en registro), ya que únicamente aquí ambas estarían intentando realizar la misma operación sobre el registro conflictivo: una escritura de datos. En

la figura 62 se presenta un ejemplo del progreso de la ejecución de un programa a través del tiempo cuando existe este tipo de atascos en el cauce del procesador.

- Riesgos de control: como sin duda se sabrá, al ejecutarse una instrucción de transferencia de control, el resultado de la evaluación de la condición de salto determinará si se debe modificar el flujo de ejecución natural del programa, es decir, si será necesario realizar alguna modificación adicional en el registro de uso particular del procesador conocido como Puntero de Instrucción (IP) para que, en lugar de apuntar a la instrucción cuya ubicación física se encuentra inmediatamente después de la instrucción de transferencia de control, contenga la dirección de salto a partir de la cual se almacena la nueva instrucción que deberá ejecutarse a continuación de la instrucción de salto en reemplazo de aquélla que estaba siendo previamente apuntada por el IP. Bajo ningún concepto el salto se encuentra obligado a llevarse a cabo hacia adelante, es decir, la nueva instrucción a ejecutar puede localizarse antes o después de la instrucción de transferencia de control que originó el salto. Ahora bien, si no se cumpliera la condición evaluada en el salto, entonces no existiría ningún inconveniente ya que se estaría preservando el flujo de ejecución natural del programa, por lo que el procesador procedería con la ejecución de la instrucción ubicada a continuación de la instrucción de salto como en cualquier otra situación normal. Sin embargo, si la condición de salto llegara a cumplirse, entonces sería incorrecto continuar con la ejecución de la instrucción localizada a continuación de la instrucción de transferencia de control ya que ésta no pertenece más al flujo de ejecución del programa, encontrándose sustituida por la nueva instrucción contenida a partir de la dirección de salto que deberá ser actualizada en el IP para que el procesador pueda ejecutarla. Por su parte, al ejecutarse ahora el programa en forma segmentada en lugar de secuencial, se posee la certeza de que, en caso de alterarse el flujo de ejecución del programa debido al cumplimiento de la condición de salto planteada en una instrucción de transferencia de control, no se dispondrá del tiempo suficiente para evitar que la instrucción originalmente establecida para ejecutarse a continuación de la instrucción de salto comience a ser ejecutada antes de haber logrado determinar la nueva instrucción a ejecutar en su reemplazo de acuerdo con el flamante flujo de ejecución definido para el programa. En función del tipo de instrucción de control involucrado en esta situación, se pueden distinguir dos casos conocidos:
    - ❖ Salto incondicional: las instrucciones que presentan este tipo de salto básicamente no poseen condición de salto alguna o, en otras palabras, tienen una condición de salto que siempre se cumple. Por lo tanto, aquí en todas las ocasiones que se

ejecuten esta clase de instrucciones se alterará el flujo de ejecución del programa para que la próxima instrucción a ejecutar a continuación del salto sea aquélla almacenada a partir de la dirección de salto. Al no requerirse la intervención de la ALU del procesador para realizar alguna operación aritmética orientada a evaluar la condición de salto de la instrucción, el cambio en el flujo de ejecución del programa puede ser llevado a cabo directamente por la unidad de control de la CPU sin ningún inconvenientes. Esta acción se llevaría a cabo en la segunda etapa del ciclo de ejecución de la instrucción de transferencia de control (decodificación), inmediatamente luego de que la unidad de control hubiera logrado decodificar tanto la identificación de dicha instrucción como la naturaleza incondicional de su salto asociado. Por lo tanto, la instrucción que originalmente formaba parte del flujo de ejecución natural del programa y cuya ejecución ya había comenzado a efectuarse sería prontamente cancelada, procediendo en su lugar a iniciar la ejecución de la nueva instrucción apuntada por el IP como consecuencia del salto acontecido. En la figura 63 se presenta un ejemplo del progreso de la ejecución de un programa a través del tiempo cuando existe este tipo de atascos en el cauce del procesador.

- Salto condicional: aquí la situación es ligeramente más complicada que en el caso anterior ya que ahora sí existe una condición de salto asociada a la instrucción de salto que deberá ser evaluada a fin de determinar si efectivamente debe alterarse el flujo de ejecución natural del programa. En consecuencia, si se compara este escenario con el previo, inevitablemente deberá demorarse la resolución de la evaluación, ya que la unidad de control no posee las prestaciones requeridas para llevar a cabo las operaciones aritméticas que sí es capaz de realizar la ALU y que resultarán imprescindibles para determinar el cumplimiento o incumplimiento de la condición de salto. La primera conclusión que puede obtenerse de este retraso indeseado es que, en cualquier caso, tanto si se modifica el flujo de ejecución del programa como si se mantiene inalterable, la instrucción ubicada a continuación de la instrucción de transferencia de control, que forma parte del flujo natural y ya había comenzado con su ejecución antes de que la unidad de control lograra identificar la instrucción de salto, deberá demorar su ejecución hasta que la ALU del procesador haya realizado la operación aritmética necesaria para evaluar la condición de salto y finalizado la evaluación efectiva de dicha condición. Dicha evaluación, como cabe esperarse, será llevada a cabo en la tercera etapa del ciclo

de ejecución de la instrucción de transferencia de control (ejecución), en lugar de en la segunda (decodificación) como ocurría en el caso de los saltos incondicionales. Finalmente, si la ALU determinó que la condición se cumple, entonces la instrucción que se encontraba demorada reanudará su ejecución sin ningún otro tipo de inconvenientes. No obstante, en caso contrario la instrucción temporalmente detenida deberá cancelar definitivamente su ejecución y será sustituida por la nueva instrucción según lo indique la dirección de salto indicada en la instrucción de transferencia de control que cambiará el flujo de ejecución natural del programa. En las figuras 64 y 65 se exhiben respectivamente un ejemplo del progreso de la ejecución de un programa a través del tiempo para cada una de las dos alternativas previamente explicadas que pueden presentarse cuando existe este tipo de atascos en el cauce del procesador.

- Riesgos de fin del programa: a través de la ejecución de los distintos programas de prueba diseñados para comprobar el funcionamiento de la computadora se detectó una situación extremadamente particular y anómala que no había sido tenida en cuenta anteriormente ni tampoco pudo ser encuadrada dentro de ninguno de los riesgos previamente explicados. El escenario es el siguiente: se dispone de un programa que actualmente se encuentra en las instancias finales de su ejecución. El procesador comienza a ejecutar sus últimas dos instrucciones, las cuales consisten en una que realiza una operación aritmética en punto flotante en la FPU de la CPU, y otra que simplemente se trata de la instrucción HALT, encargada de detener definitivamente la ejecución del procesador. Ahora bien, como se recordará, las instrucciones que deben llevar a cabo operaciones aritméticas en punto flotante en la FPU poseen un ciclo de ejecución más largo y lento que aquéllas que no necesitan realizar esta tarea. Por su parte, la orden de detención de la ejecución de la CPU es emitida por la unidad de control del procesador durante la segunda etapa del ciclo de ejecución (decodificación) de la instrucción HALT, la cual a su vez presentará un ciclo de ejecución corto, como el de cualquier instrucción cuya ejecución no involucra la utilización de la FPU. Esto significaría que, en condiciones normales, sólo quedarían por ejecutarse una etapa de ejecución de un ciclo único (la etapa de ejecución de la propia instrucción HALT), dos etapas de acceso a memoria y tres etapas de almacenamiento en registro (ver figura 66). Precisamente el procesador de este proyecto fue diseñado para, una vez decodificada e identificada la instrucción HALT, detener inmediatamente su ejecución a partir del escenario descripto en la sentencia anterior. Sin embargo, la presencia de una instrucción que realice una operación aritmética en punto flotante en la

FPU, al ser su ciclo de ejecución más largo que el del resto de las instrucciones del repertorio del procesador, representaría una anomalía en este escenario ya que en él sólo se había considerado un único tipo de ciclo de ejecución: aquél conformado por cinco etapas de un ciclo de reloj cada una y asociado a las instrucciones que no requieren utilizar la FPU para efectuar sus operaciones. Esta alteración en el contexto de ejecución podría provocar consecuencias indeseadas sobre la correcta finalización del programa ya que una instrucción que incluya una operación en punto flotante no sólo tardará más tiempo en completar su ejecución que el resto de las instrucciones sino que además podría provocar cualquier tipo de los otros riesgos de segmentación ya mencionados que demorarían la ejecución de las instrucciones ubicadas a continuación de ella en el programa. De cualquier manera, el efecto indeseado de esta alteración inesperada del contexto sería que la unidad de control, al no haber sido inicialmente diseñada para detectar este cambio en las condiciones de ejecución, al identificar que actualmente la instrucción HALT se encuentra en ejecución, emita prematuramente la orden de detención para el procesador, lo cual a su vez tendría como consecuencia la temprana e incorrecta finalización de la instrucción que involucra el uso de la FPU y/o de las instrucciones que se encuentran a continuación de ella en el programa y también puedan verse afectadas por posibles riesgos durante su ejecución. Esto se debe a que, al requerir un mayor número de ciclos de reloj para completarse que el esperado, la CPU se estaría deteniendo antes de que las instrucciones alcancen a ejecutar sus últimas etapas (ver figura 67). La única solución posible para evitar este riesgo consiste en demorar la ejecución de la instrucción HALT hasta que sólo una o ninguna instrucción posea pendiente de ejecución un único ciclo en su etapa de ejecución (tercera etapa del ciclo de ejecución de las instrucciones). Este ciclo puede incluir dos alternativas: el último ciclo de la etapa de ejecución de una instrucción que necesita realizar una operación aritmética en punto flotante en la FPU, o bien una etapa de ejecución de ciclo único de cualquiera de las demás instrucciones que conforman el repertorio del procesador. De este modo, pueden garantizarse condiciones de ejecución similares al escenario planteado inicialmente como “normal”, en el cual la única etapa de ejecución pendiente sería aquélla asociada a la propia instrucción HALT. Así podrá asegurarse sin ningún tipo de duda que la orden de detención de la CPU no será emitida prematuramente por la unidad de control y que el programa no finalizará su ejecución hasta que absolutamente todas sus instrucciones se hayan completado hasta sus últimas etapas. Finalmente, por si es necesario, cabe aclarar que la etapa de la instrucción HALT directamente afectada y demorada por la presencia de este riesgo es la segunda de

su ciclo de ejecución (decodificación), ya que sería precisamente aquí el momento en el cual la unidad de control decodificaría la identificación de la instrucción e intentaría emitir la orden de detención a todos los componentes del procesador. En la figura 68 se presenta un ejemplo del progreso de la ejecución de un programa a través del tiempo cuando existe este tipo de atascos en el cauce del procesador.