Centro de Investigación y de Estudios Avanzados
del Instituto Politécnico Nacional

Unidad Zacatenco

Departamento de Computación

Diseño de un Microcontrolador Educativo en VHDL

Tesis que presenta

Alejandro Juárez Arellano

para obtener el Grado de

Maestro en Ciencias

en Computación

Director de la Tesis

Dr. Luis Gerardo de la Fraga

México, D. F. **Octubre 2013**

Resumen

En este trabajo se presenta el diseño de un microcontrolador de 16 bits, el diseño del lenguaje ensamblador para programarlo y un banco de siete pruebas para demostrar su funcionamiento. El diseño fue realizado en el lenguaje de descripción de hardware (VHDL), lo que permite probar el diseño en dispositivos lógicos programables, con el objetivo de optimizarlo y usarlo en distintas aplicaciones, permitiendo experimentar con diversas arquitecturas y organización de computadoras. La última prueba realizada es la aplicación no trivial de los esquemas CCM y GCM de Autenticación Encriptada.

Los microcontroladores forman parte de un gran número de aparatos electrónicos hoy en día. En los relojes que tienen una pantalla de LED o LCD, dentro del automóvil puede llegar a tener al menos cuarenta de estos componentes para controlar el motor, los frenos, el aire acondicionado y así sucesivamente. Cualquier sistema que cuente con un control remoto es seguro que tenga un microcontrolador: televisiones, reproductores de vídeo, ratón y teclado inalámbrico, impresoras, teléfonos, cámaras digitales, ..., etc. Los microcontroladores se usan para controlar una sola tarea con recursos limitados integrados dentro del circuito integrado, por lo regular se encuentra incrustado en la tarjeta del dispositivo que controla.

Las características principales del microcontrolador diseñado son: Bus de 16 bits para datos y direcciones y bus de instrucciones de 29 bits. La memoria de datos almacena palabras de 16 bits, el archivo de registros cuenta con 16 registros de propósito general de 16 bits. Cuenta con cuatro puertos paralelos de 16 bits, cada pin de los puertos puede configurarse de entrada o salida, los bits de entrada se actualizan en cada ciclo del reloj. Un puerto serial que transmite y recibe un byte de forma asíncrona. Tiene un manejador de interrupciones que atiende las señales por prioridad para controlar componentes internos y externos. También tiene cuatro temporizadores atendidos por interrupciones, además de un temporizador guardián como opción de seguridad en cualquier aplicación. Además cuenta con un contador de programa organizado con una pila de registros para las subrutinas. La arquitectura es de un conjunto reducido de instrucciones RISC con una organización de memoria Harvard. La memoria de programas es lineal y la memoria de datos está organizada por páginas. El ciclo de instrucción se ejecuta en un ciclo del reloj.

Abstract

This work presents the design of a 16-bit microcontroller, the design of its assembly language for programming, and a bank of seven tests to prove their functionality. It was designed in hardware description language (VHDL), which allows testing the design with programmable logic devices, in order to optimize it and use it in different applications, allowing to experiment with different architectures and computer organization. The last test performed was the nontrivial application of CCM and GCN schemes of Encrypted Authentication.

The microcontrollers are part of a large number of electronic devices today. They are used in watches that have LED or LCD screen, inside cars where at least forty of these components are used to control the engine, brakes, air conditioning and so on. Any system that has a remote control is sure to have a microcontroller: TVs, VCRs, wireless keyboard and mouse, printers, phones, digital cameras, etc. The microcontrollers are used to control a single task with limited resources embedded within an integrated circuit, and it is usually embedded within the card of the device it controls.

The main features of the designed microcontroller are: Bus 16-bit data and address and instruction bus of 29 bits. The data memory stores 16-bit words, the log file has 16 general purpose registers of 16 bits. It has four 16-bit parallel ports, each port pin can be configured for input or output, the input bits are updated at each clock cycle. A serial port that transmits and receives a byte asynchronously. It has an interrupt handler which handles priority signals to control internal and external components. It also has four timers attended by interruptions, and a guard or watch dog timer as security option in any application. Furthermore, it count with a program counter organized as a stack of records for subroutines. The architecture is a RISC reduced instruction set with a Harvard memory organization. The program memory is linear and the data memory is organized by pages. The instruction cycle is executed in one clock cycle.

A mi familia...
Mis mentores y héroes.

Agradecimientos

Agradezco a mi familia por su amor incondicional, por su apoyo y consejos brindados en cada una de mis decisiones.

Quiero agradecer al CONACyT por el apoyo dado con la beca para realizar los estudios de maestría y al CINVESTAV-IPN por permitirme formar parte de un ambiente que promueve el conocimiento científico y tecnológico.

Quiero agradecer en especial en memoria del Dr. Adriano de Luca Pennacchia que me apoyo para la selección del tema de tesis, quien con sus conocimientos, su experiencia, su paciencia y su motivación me ayudo a comenzar este trabajo.

Quiero agradecer a mi director de tesis el Dr. Luis Gerardo de la Fraga por su apoyo y confianza en mi trabajo, cuya guía ha sido invaluable.

También quiero agradecer a todas las personas que de alguna manera me apoyaron, a mis profesores y compañeros porque todos han aportado con un granito de arena a mi formación, en especial: Dra. Sonia G. Mendoza, Sra. Sofía Reza Cruz, Dr. Amilcar Meneses Viveros, Cuauhtémoc Mancillas López, Jesús Salvador Martínez Delgado y César David Corona Arzola.

Agradezco al CONACyT por el apoyo recibido del proyecto CB2011/168357.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

Una computadora como una *laptop*, *tablet* o de escritorio es un dispositivo de propósito general, esta máquina puede resolver problemas, cuyo elemento principal es el procesador el cual ejecuta las instrucciones que recibe de las personas, atendiendo a un formato estricto y tareas limitadas que este dispositivo realiza. Una secuencia de instrucciones que describe cómo realizar cierta tarea se llama **programa**. Los circuitos electrónicos de una computadora pueden reconocer y ejecutar directamente un conjunto limitado de instrucciones sencillas, a consecuencia de esto todos los programas tienen que convertirse en una serie de estas instrucciones, para que la computadora pueda ejecutarlos. Dichas instrucciones básicas casi nunca son más complicadas, ej.: sumar dos números, verificar si un número es cero, copiar un dato de una parte de la memoria a otra, etc.

El conjunto de instrucciones primitivas de una computadora constituye un lenguaje que permite a las personas comunicarse con la computadora, dicho lenguaje se llama **lenguaje de máquina** o **código máquina**.

El procesador denominado como *controlador incrustado* [1] o *microcontrolador* se emplea para controlar el funcionamiento de una sola tarea determinada, es de un tamaño reducido y se incorpora por lo regular en el dispositivo que gobiernan. Los dispositivos denominados como Controlador de Interfaz de Periféricos (PIC) o *Peripheral Interface Controller*, por sus siglas en inglés, fabricados por *Microchip Thecnology Inc* son un ejemplo de microcontroladores. Aunque su diseño permite ser de propósito general, está limitado a realizar una sola función, por razones de eficiencia o simplemente por conveniencia.

Los microcontroladores pueden utilizarse para distintas aplicaciones como controlar un motor eléctrico, desplegar mensajes e imágenes en una marquesina de LEDs, reproducir música MP3, en cronómetros o relojes digitales, junto con distintos transductores que convierten alguna señal física en eléctrica y sean convertidas a señales digitales, se pueden medir la intensidad de luz, temperatura, voltaje, pulso cardiaco para fines médicos, industriales o de uso doméstico, etc.

Los microprocesadores comerciales existentes se suministran como unidades similares a una caja negra, ilustrada en la figura 1.1. Se toma en cuenta lo que entra y lo que sale, sin

entender profundamente el funcionamiento interno ya que es un secreto comercial. Al describirse los modelos y arquitecturas de forma abierta se permite un mayor entendimiento para los estudiantes y futuros trabajos en este campo, cuyas aplicaciones comerciales y académicas son variadas.

Las utilidades de los procesadores como controladores incrustados, son muy requeridos en los aparatos electrónicos modernos. La tecnología de los dispositivos lógicos programables, en especial los FPGA permiten profundizar en el diseño de este tipo de sistemas digitales.

Figura 1.1: Caja negra

En 1971 Intel y Marcian E. Hoff construyeron el primer microprocesador: el 4004, con un ancho de palabra de 4 bits. Este componente fue un controlador integrado, programable en un solo encapsulado. Disponía de 4096 localidades en la memoria, el repertorio de instrucciones consistía de 45 instrucciones distintas. Se empleaba solamente en aplicaciones limitadas, algunas fueron las primeras versiones de vídeo juegos, o aquellos sistemas en los que hoy se utilizan los microcontroladores.

En el mismo año, al observar la posibilidad de comercialización del microprocesador como producto viable, Intel produjo el 8008, que ya era considerado una computadora de propósito general. Las dimensiones de la memoria eran de 16K de 8 bits, con 48 instrucciones adicionales, sus capacidades permitían realizar aplicaciones más complicadas. Aunque el 8008, permitía realizar más tareas, aún estaba limitada su utilidad. En 1973 Intel presento el 8080 que es considerado el primer microprocesador moderno. Muchas empresas sacaron a la venta sus propias versiones, las cuales se muestran en la tabla 1.1.

Fabricante	Producto	Año
Motorola	MC6800	1974
Burroughs	Mini-D	1973
Fairchild	F-8	1975
MOS Technology	6502	1975
National Semiconductor	ImP-8	1974
Rockwell International	PPS-8	1975
Signetics	2650	1975

Tabla 1.1: Microprocesadores de 8 bits

En 1976 Intel lanzó el 8086, y en 1979 el 8088. Ambos microprocesadores de 16 bits pueden direccionar 1MB de 8 bits o 512KB de 16 bits. La necesidad de realizar operaciones de multiplicación y división por hardware aceleró el diseño de procesadores de 16 bits, además permite un espacio de direccionable más grande que uno de solo de 8 bits [2].

Las generaciones de la computadora están directamente ligadas a la evolución del procesador, se puede establecer en la siguiente división:

* Instrumentos mecánicos de 1642 a 1945.
* Máquinas con bulbos de 1945 a 1955.
* Máquinas con transistores de 1955 a 1965.
* Máquinas con circuitos integrados de 1965 a 1980.
* Máquinas con integración a muy grande escala de 1980 hasta la fecha actual.

Algunos de los antecedentes de las distintas generaciones mencionadas son mostrados en la tabla 1.2.

Año	Nombre	Creador	Descripción
1640	Pascalina	Pascal	Calculadora mecánica (+,-)
1671	Rueda escalada	Leibniz	Calculadora mecánica (+,-,/)
1834	Máquina analítica	Babbage	Primer intento de computadora digital
1936	Z1	Zuse	Primera computadora de relevadores
1943	COLOSSUS	Flowers	Solo descifraba mensajes
1944	Mark 1	Aiken	Primera computadora de propósito general
1946	ENIAC	Eckert Mauchley	Inicia la computadora moderna
1949	EDSAC	Wilkes	Computadora con programa almacenado
1951	Whielwind 1	M.I.T	Computadora de tiempo lineal
1952	IAS	Von Neumann	Diseño de las computadoras actuales
1960	PDP-1	DEC	Primera minicomputadora.
1961	1401	IBM	Máquina pequeña para negocios, muy popular
1962	7094	IBM	Dominó la computación científica en los sesenta
1963	B5000	Burroughs	Diseñada para un lenguaje de alto nivel
1964	360	IBM	Diseñada como la primer familia de productos
1964	6600	CDC	Primera supercomputadora científica
1965	PDP-8	DEC	Primera minicomputadora con mercado masivo
1970	PDP-11	DEC	Domino las minicomputadoras en los 1970
1974	CRAY-1	Cray	Primera supercomputadora vectorial
1978	VAX	DEC	Primera superminicomputadora de 32 bits
1981	IBM PC	IBM	Inicia la computadora personal moderna
1985	MIPS	MIPS	Primera máquina RISC comercial
1987	SPARC	Sun	Primera estación de trabajo RISC-SPARC
1990	RSC6000	IBM	Primera máquina superescalar

Tabla 1.2: Antecedentes del procesador y de la computadora

1.1. Planteamiento del problema

Aunque están disponibles en el mercado decenas de microcontroladores, estos diseños ya están fijos y no pueden modificarse.

En este trabajo de tesis se pretende realizar un diseño de un microcontrolador de propósito general, cuyo diseño sea simple, completo y abierto en VHDL. Usar VHDL permite la simulación del diseño y es muy flexible.

Se intenta proporcionar una herramienta didáctica para poder ser usada en cursos relacionados con el lenguaje ensamblador, organización y arquitectura de computadoras. El diseño propuesto podría bajarse a una tarjeta de trabajo y podría así utilizarse en diversas aplicaciones, como robots móviles siguelíneas o en adquisición de señales con la adaptación de un convertidor analógico digital.

El diseño ejecutará una instrucción en un solo ciclo de reloj, sin *pipeline*. El fin de la propuesta es proponer un conjunto de instrucciones y una organización simple de un microcontrolador para poder controlar un conjunto reducido de recursos: contadores, puerto serial, puerto paralelo, modulación por ancho de pulso, manejador de interrupciones y un perro guardián.

El diseño tendrá una organización de memoria tipo Harvard, este es el más usual en los microcontroladores, para poder manejar por separado las memoria de instrucciones tipo ROM (del inglés *Read Only Memory*, memoria de solo lectura) y la memoria de datos tipo RAM (del inglés *Random Access Memory*, memoria de acceso aleatorio). También se plantea acceder a los registros para configurar los periféricos a través de localidades mapeadas de la memoria de datos.

El lenguaje ensamblador para el diseño de cualquier microcontrolador debe ser entendido por un ensamblador para que a partir de un programa en lenguaje ensamblador se traduzcan a su equivalente en lenguaje de máquina. Por ejemplo la instrucción aritmética de suma:

$$add\ \$r_1,\ \$r_2,\ \$r_3$$

debe intercambiarse a una cadena de ceros y unos, como:

$$\begin{gathered}[\text{CO}][\text{RD}][\text{RF0}][\text{RF1}][\text{CF}] \\ \downarrow \\ [0001][0001][0010][0011][0000] \\ \downarrow \\ [ALU][r_1][r_2][r_3][suma]\end{gathered}$$

donde cada conjunto de bits significa:

* Código de operación (CO): indica el tipo de instrucción.
* Registro destino (RD) y registros fuentes (RF0 y RF1): indica el origen de los operandos y el destino del resultado. El modo de direccionamiento de una instrucción indica el origen de estos argumentos, en el ejemplo anterior se usa el direccionamiento por registro.
* Código de función (CF): junto con el de operación indican que tarea realizar, por lo general se emplea en las instrucciones relacionadas con la ALU.

1.2. Objetivos

El objetivo general de este trabajo es el diseño digital de un microcontrolador con el lenguaje VHDL, para que pueda ser configurado sobre tecnología FPGA e incluido en algún sistema de control digital.

Los objetivos particulares son los siguientes:

1. Establecer prioridad en el diseño de la unidad central de procesamiento CPU, para que sea ocupado como un microcontrolador incrustado.
2. Realizar simulaciones del funcionamiento integral e individual de los componentes que forman al procesador descrito en lenguaje VHDL.
3. Desarrollar un ensamblador básico para realizar la traducción de lenguaje ensamblador en su equivalente a lenguaje máquina.
4. Aprender a diseñar y desarrollar diseños digitales con el lenguaje de descripción de hardware VHDL.
5. Estudiar el nivel de abstracción desde el código fuente en lenguaje ensamblador, hasta la interpretación de las instrucciones del lenguaje máquina por el procesador.
6. Probar el diseño propuesto con un conjunto de programas realizados en su mismo lenguaje ensamblador.

1.3. Resultados obtenidos

Los resultados en términos generales solamente se reducen a tres puntos principales:

* Un diseño propio de un microcontrolador en VHDL, que tiene que ser *simple* para ser personalizado a los requerimientos de alguna aplicación, *completo* al contar con el mínimo de periféricos necesarios para que sea práctico su uso y *abierto* para continuar su desarrollo en proyectos futuros.
* Diseño del lenguaje ensamblador y los programas ensamblador y desensamblador. Éstas facilitan las herramientas de programación para disponer del microcontrolador como un elemento práctico en alguna aplicación académica.
* Una propuesta de un conjunto de pruebas de simulación, con el fin de validar y demostrar el funcionamiento del microcontrolador. Esto también demuestra que el código fuente en ensamblador puede ser el punto de partida para una futura colección o biblioteca de aplicaciones disponibles para este diseño de microcontrolador.

1.4. Organización de la tesis

La presente tesis está organizada en siete capítulos, la descripción de cada uno se presenta a continuación.

* En el capítulo 2 se definen algunos de los conceptos, es decir, el marco teórico necesarios para diseñar el microcontrolador, se indican los componentes y comportamientos de cada elemento para realizar su funcionamiento en el procesador.

* En el capítulo 3 se mencionan algunos trabajos relacionados sobre procesadores y ensambladores. Estos trabajos se toman como modelos en la fase del diseño de la unidad de procesamiento y control (CPU) del procesador y de la elaboración del ensamblador.

* En el capítulo 4 se detallará los requerimientos y la propuesta de diseño del microcontrolador, se define el conjunto de instrucciones y la organización del procesador. Se especifica cada bloque que conforma al dispositivo describiendo su comportamiento y la configuración necesaria para conformar al microcontrolador.

* En el capítulo 5 se menciona las especificaciones del lenguaje ensamblador que servirá como marco para ser traducido a lenguaje de máquina. También en este capítulo se plantea el funcionamiento que este tipo de programa realiza, se especifican los servicios ofrecidos por el ensamblador, el desensamblador y la generación de la memoria de programa en una plantilla que describe una memoria ROM en lenguaje VHDL.

* En el capítulo 6 se mencionan las aplicaciones que validan el diseño del microcontrolador, entre las que se muestran el funcionamiento de los periféricos, así como su uso en encriptación autenticada. Así mismo se describen los resultados conseguidos con las simulaciones obtenidas por medio del uso de la herramienta disponible por XILINX el ISE WebPack.

* Finalmente en el capítulo 7 se describen las conclusiones de este trabajo, además del trabajo a futuro que se podría realizar para mejorar este trabajo.

Capítulo 2

Marco teórico

En este capítulo se abordan los conceptos que especifican las características esenciales de los procesadores, se proporcionan las nociones básicas necesarias para la realización del diseño del microcontrolador incrustado. Se describe como se conforma un procesador, la organización de memoria, así como el tipo de arquitectura, además se define lo que es una instrucción, y también se da una descripción más completa de los modos de direccionamiento. Para el manejo de las interrupciones se describe tanto el manejo por prioridad como el de sondeo.

2.1. Diseño de computadoras

La **arquitectura de computadoras** [3] constituye un área de estudio que se refiere a la interfaz entre hardware y software para diseñar computadoras digitales, es decir, abarca un conjunto de ideas centrales aplicables al diseño o comprensión de virtualmente cualquier computadora digital, desde los más pequeños sistemas anidados hasta las más grandes supercomputadoras. Los subcampos en el diseño de computadoras se ilustran en la figura 2.1, la descripción de cada campo se describe a continuación:

* **Circuitos**: abarca el nivel más bajo, el de los fenómenos físicos que hacen que el hardware de la computadora pueda realizar sus tareas.
* **Lógico**: afronta modelos como compuertas o flip-flops, se apoya de herramientas para el diseño de circuitos que se pueda mostrar mediante la abstracción.
* **Computadoras**: trata principalmente con principios lógicos digitales (ej.: sumadores y registros), debe tener una visión a nivel lógico, debe contar con nociones del área del diseño de sistemas. Proporciona una capa de software que facilita el diseño y desarrollo de aplicaciones.
* **Sistemas operativos**: aborda el hardware en bruto, con componentes claves de software que proteja al usuario de detalles de la operación del hardware, ofreciendo una interfaz con la máquina de uso fácil.

* **Aplicaciones**: es el nivel más alto, utiliza las facilidades ofrecidas por el hardware y el software de nivel inferior para generar soluciones a problemas de aplicación, que interesan a algún usuario en particular.

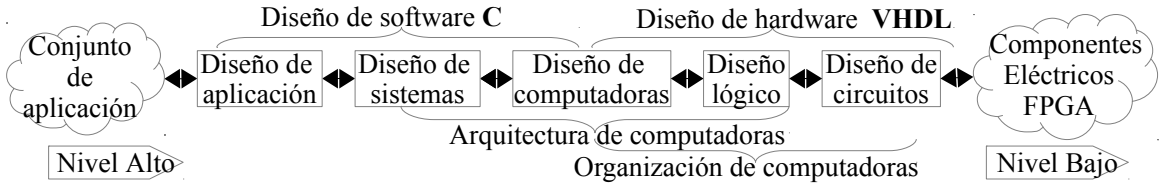


Figura 2.1: Etapas de diseño de un sistema de computadoras

Es importante mencionar aquí que las etapas más bajas, la de diseño de computadoras, diseño lógico y diseño de circuitos, como se ve en la figura 2.1 tratan de diseño de hardware (las otras etapas se diseñan en software) pero en esta tesis se desarrollan también en el software de VHDL. Esto permite una gran ventaja al poder probar los diseños en dispositivos lógicos programables antes de fabricarlos en un chip.

2.2. Dispositivo lógico programable

Hoy en día es posible tener sistemas completos dentro de un solo circuito integrado, que son conocidos como sistemas sobre un chip (SOC por sus siglas en inglés *System On Chip*), lo cual ha aumentado la velocidad, confiabilidad, y sobre todo el área de diseño. En la actualidad, el diseño ASIC (por sus siglas en inglés de *Application-Specific Integrated Circuit*, circuitos integrados desarrollados para aplicaciones específicas) es el nivel de desarrollo adecuado en aplicaciones que requieren un alto volumen de producción. Los dispositivos lógicos programables (PLD del inglés *Programmable Logic Devices*) se tratan de dispositivos fabricados y revisados que se pueden personalizar desde el exterior mediante diversas técnicas de programación.

El diseño se basa en bibliotecas y mecanismos específicos de mapeado de funciones, mientras que su implementación tan solo requiere una fase de programación del dispositivo que el diseñador suele realizar en unos segundos. Los PLD sustituyen circuitos con distinto nivel de integración (con distintos números de componentes), y así los circuitos integrados se pueden clasificar según el número de componentes o la familia de componentes con el que está constituido. Las tablas 2.1 y 2.2 muestran las clasificaciones existentes de los circuitos integrados.

Familia	Siglas
Lógica de transistores y resistores	RTL
Lógica de diodos y resistores	DTL
Lógica de transistores y transistores	TTL
Lógica de transistores de efecto de campo complementario de óxido metal	IGFET
Complentario metal oxido semiconductores	CMOS
Lógica de emisor acoplado	ECL
Lógica de tres estados	TSL

Tabla 2.1: Familias lógicas de los circuitos integrados.

Tipo	Siglas	Número de transistores
Pequeña escala de integración	SSI	de 10 a 100
Mediana escala de integración	MSI	de 101 a 1,000
Alta escala de integración	LSI	de 1,001 a 10,000
Muy alta escala de integración	VLSI	de 10,001 a 100,000
Ultra alta escala de integración	ULSI	de 100,001 a 1,000,000
Giga alta escala de integración	GLSI	más de un millón

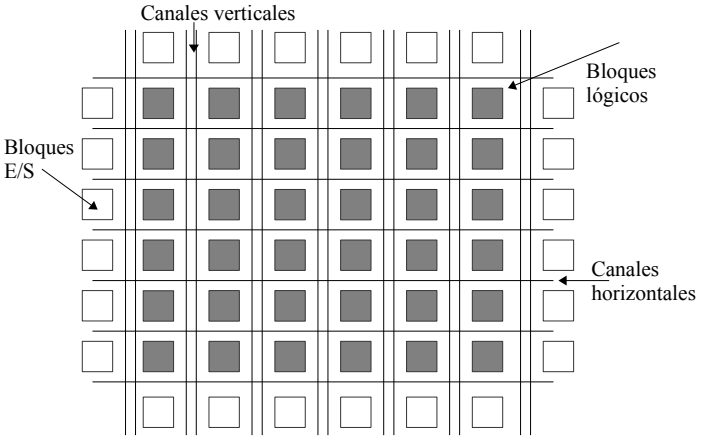
Tabla 2.2: Escalas de integración de los circuitos integrados.

La arquitectura básica de un PLD está formada por un arreglo de compuertas AND y OR conectada a las entradas y salidas del dispositivo. En ambas configuraciones de arreglos las compuertas están interconectadas a través de alambres, los cuales cuentan con un fusible en cada punto de intersección. En esencia la programación consiste en fundir o apagar los fusibles. Una vez fundidos no pueden volver a programarse. Existen varios tipos de PLD, se listan a continuación:

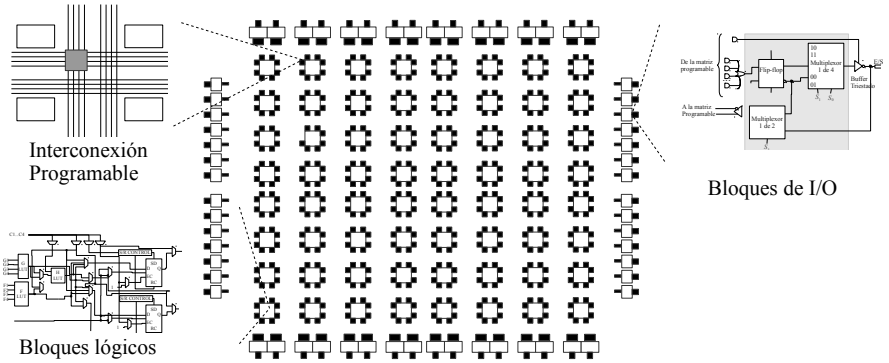
* **PROM**: memoria programable de solo lectura, solo se utilizan como memoria direccionable.
* **PLA**: arreglo lógico programables. En estos dispositivos el fabricante es quien lo programa.
* **PAL**: lógica de arreglos programables; programables por el usuario, sustituye circuitos combinacionales y secuenciales SSI y MSI en un circuito.
* **GAL**: arreglo lógico genérico, está formada por celdas que pueden ser programadas las veces que sean necesario. Se conforma de arreglos OR, AND y macroceldas lógicas de salida. Utiliza E^2CMOS CMOS borrables eléctricamente, en lugar de tecnología bipolar y de fusibles.
* **CPLD o EPLD**: dispositivos lógicos programables complejos o mejorados, consiste en un arreglo múltiple de PLD agrupados como bloque en un chip. Su capacidad equivalente a 50 PLDs sencillos. Su interconexión es programable (IP), conectando los bloques lógicos con los bloques de E/S del dispositivo.
* **FPGA**: arreglo de compuertas programables de campo, que son los que se usarán en este trabajo de tesis.

FPGA

Los dispositivos FPGA se basan en lo que se conoce como arreglos de compuertas, la arquitectura contiene tres elementos configurables: bloques lógicos configurables (CLB), bloque de entrada y salida (IOB) y canales de comunicación. Los CLB se comunican a las terminales de E/S por medio de los canales de comunicación. El diseño lógico se implementa mediante bloques conocidos como generadores de funciones o tablas de búsqueda (*lookup table* o LUT), que permiten almacenar la lógica requerida, ya que cuentan con una memoria interna. Los CLB están ordenados en arreglos de matrices programables (PSM).



(a) Arquitectura básica.



(b) Arquitectura XILINX.

Figura 2.2: Arquitectura del FPGA

Las empresas Xilinx, Altera, y QuickLogic son sólo algunas de las que fabrican FPGAs. Todos los fabricantes comparten el mismo concepto básico de arquitectura: interfaces de entrada/salida (E/S), los bloques básicos de construcción y las interconexiones. La Figura 2.2 ilustra la arquitectura general de un FPGA.

2.3. Lenguajes de descripción de hardware

La necesidad de integrar un mayor número de dispositivos en un solo circuito integrado, obligó la construcción de nuevas herramientas de diseño que auxilian al ingeniero a integrar sistemas de mayor complejidad. Esto permitió que aparecieran lenguajes de descripción de hardware (HDL por sus siglas del inglés *Hardware Description Language*) como una opción de diseño para el desarrollo de sistemas electrónicos elaborados. Los primeros de estos lenguajes eran propietarios, restringidos para las empresas que los crearon.

Los siguientes lenguajes no tuvieron mantenimiento ni soporte para ser utilizados en la industria. En los ochenta apareció **VHDL**, Verilog, ABEL 5.0, y AHDL, considerados lenguajes de descripción de hardware por que permitieron abordar un problema lógico a nivel funcional (describir un problema solamente conociendo las entradas y salidas), facilitando la evaluación de alternativas antes de iniciar un diseño detallado.

La principal característica de estos lenguajes se encuentra en su capacidad para describir distintos niveles de abstracción (funcional, transferencia de registros y lógica o nivel de compuertas). Estos niveles de abstracción son descritos a continuación:

* **Funcional**: es la relación funcional entre las entradas y salidas del circuito o sistema, sin hacer referencia a la realización final.
* **Transferencia de registros**: consiste en la partición del sistema en bloques funcionales sin considerar a detalle la realización final de cada bloque.
* **Lógico o de compuerta**: el circuito se expresa en términos de ecuaciones lógicas o de compuertas.

VHDL

VHDL es un lenguaje estándar, capaz de soportar el proceso de diseño de sistemas electrónicos complejos, con propiedades para reducir el tiempo de diseño y los recursos tecnológicos requeridos. El departamento de Defensa de Estados Unidos creo el lenguaje VHDL como parte del programa "*Very High Speed Integrated Circuits*" (VHSIC), que se trata del diseño rapido de circuitos integrados a una alta escala de integración. Una pieza independiente de código VHDL está compuesta de al menos tres secciones fundamentales:

* **Declaraciones de biblioteca**: contiene una lista de todas las bibliotecas que se utilizarán en el diseño.
* **Entidades**: especifica los pines de E/S del circuito.
* **Arquitectura** (función): contiene el código adecuado, que describe cómo debe comportarse el circuito.

Una biblioteca es una colección de piezas de uso común en el código fuente de un programa en VHDL. La colocación de nuevas piezas dentro de una biblioteca le permite al programador

reutilizar o compartir su funcionalidad en otros diseños. El código se escribe generalmente en forma de funciones, procedimientos o componentes, que se colocan en el interior de los paquetes (conjunto de constantes subprogramas y declaraciones con la intención de implementar algún servicio) y luego se compila para generar una biblioteca.

La combinación del lenguaje VHDL y dispositivos FPGA permiten a los diseñadores desarrollar rápidamente y simular un circuito digital sofisticado, cuenta con un dispositivo de creación de prototipos para verificar el funcionamiento de la implementación física. A medida que estas tecnologías maduran, se han convertido en una práctica corriente. Ahora podemos utilizar una computadora personal y una placa de prototipo FPGA de bajo costo para construir un sistema digital complejo y sofisticado.

En 1987, el Instituto de Ingenieros Eléctricos y Electrónicos o IEEE (*Institute of Electrical and Electronics Engineers*) adoptó al VHDL como un estándar que fue lanzado como IEEE Standard (Std) 1076-1987 o VHDL-87. Más o menos cada cinco años, el comité de estándares de la IEEE se reúne para revisar, mejorar y realizar otras modificaciones al lenguaje. VHDL también está disponible como VHDL-93, VHDL-2000, VHDL-2002 y VHDL-2008.

2.4. Estructura del procesador

Un **procesador** está compuesto por tres bloques fundamentales: la unidad central de control y de procesamiento (Unidad Central de Procesamiento **CPU**), la memoria, las entradas y salidas (Unidad de Entrada y Salida, I/O o **E/S**) que comunican con dispositivos externos e internos del procesador. Los bloques se conectan entre sí mediante grupos de líneas eléctricas denominadas buses. Los buses pueden ser de direcciones (identificando la celda de memoria o localidad de memoria), de datos o de control (cada línea del bus es una microinstrucción). La CPU es el cerebro del procesador, es el componente clave de todos los sistemas de cómputo.

Figura 2.3: Organización genérica de una computadora.

La computadora consiste en un sistema basado en procesadores. La organización de una computadora se muestra en la Figura 2.3. Técnicamente el procesador contiene los elementos que constituyen la CPU que por sí solo no es operativo y precisa la colaboración de la memoria, así como de los módulos de entrada y salida.

Elementos internos del procesador

Existe una gran variedad de componentes internos que constituyen al procesador, algunos de estos elementos más significativos son:

* Registros: almacenan datos de 8, 16, 32 o 64 bits. En general se pueden dividir como registros de propósito general (RPG) y los de función especial (RFE). Los RPG son los

acumuladores, que en conjunto forman los archivos de registros, encargados de almacenar valores de las instrucciones o resultados de la ALU, y forman una pequeña memoria de datos disponible para el libre uso del usuario. Los RFE son a través de los cuales se controla el procesador, pueden ser de: configuración, estado o de datos. El acceso a los RFE es por medio de localidades mapeadas en la memoria de datos.

a) Acumuladores ACC: almacenan los datos para operar por la ALU.

b) Punteros a pila SP o PP: puntero que almacena la dirección actual de la pila del programa.

c) Instrucción RI: almacena la instrucción que será decodificada por la UC.

d) Memoria RM: almacena la dirección de memoria para ser leída o almacenada. También se conoce como *Registro Auxiliar de Memoria* MAR.

e) Contadores de programa CP: retiene la dirección de memoria de programa que referencia la instrucción atendida por el ciclo de instrucción.

f) Temporales RT o auxiliadores RA: almacena datos o direcciones temporalmente.

g) Banderas o bits de status: almacena el estado actual del procesador, cada bit tiene asociado una bandera o indicador que puede estar activada o desactivada dependiendo de su significado, algunas de estas banderas son:

- Z: si el resultado de la ALU fue cero o no.
- C: si el resultado de la ALU al sumar o restar tiene acarreo o no.
- S: si el resultado de la ALU es positivo o negativo.
- O: si el resultado de la ALU genera desbordamiento.

h) Control: configura el funcionamiento que puede realizar el procesador; habilita interrupciones, configurar la entrada y salida, etc.

i) Archivo de registros AR: es una colección de registros de uso general. Por lo general pueden ser de 8, 16, o 32 registros.

* Multiplexores: orienta las conexiones de los diferentes tipos de BUS durante la ejecución de alguna instrucción.
* Memoria: almacena datos en RAM (memoria de datos MD) o almacena programas en ROM (memoria de programa MP).
* ALU: realiza las operaciones aritméticas (suma, resta, multiplicación, división) corrimientos de bits a la izquierda o derecha, lógicas (and, or y not), etc.
* Unidad de Control (UC): analiza la instrucción y por medio de las señales de control ejecuta la tarea asociada. Para procesadores sencillos la UC puede ser implementada como una memoria que decodifica el tipo de instrucción y de operación, produciendo el efecto deseado al manipular las señales de control para todos los componentes del procesador. Al aumentar la complejidad de las instrucciones esta unidad de control puede implementarse como una máquina de estado finito, donde algunos estados son para una ejecución normal y otros estados adicionales para las excepciones o interrupciones.

2.4.1. Unidad de E/S

El procesador y la memoria interna ofrecen mayor rapidez que los dispositivos de E/S que se comunican de forma externa al procesador, por medio del controlador del dispositivo y estos a su vez al BUS del sistema. La figura 2.4 muestra la estructura en bloques de la unidades de entrada y salida comunicada por medio de un bus compartido. Para atender los dispositivos de E/S, se destacan dos posibles casos que son:

* Caso 1: sondear el registro de estado de todas los dispositivos, si el estado indica que se puede acceder a datos o ejecutar una tarea entonces se cede el control a la rutina encargada a este fin. El sondeo es realizado por el programa, se trata de una rutina para establecer que dispositivo está disponible, la desventaja de esto es que consume mucho tiempo.
* Caso 2: se hacen uso de interrupciones, que son señales producidas por el dispositivo de E/S, indicando que está listo para que el procesador lea o escriba datos en el dispositivo. Estas señales interrumpen el estado normal del procesador, son atendidas por la UC en conjunto con el manejador de interrupciones, cediendo el control a la rutina necesaria para atender la interrupción.

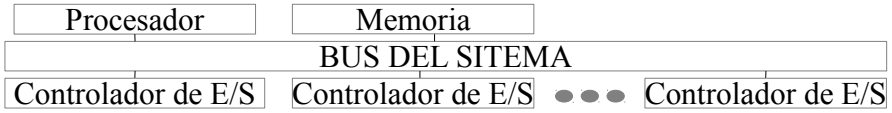


Figura 2.4: Conexiones con dispositivos de E/S

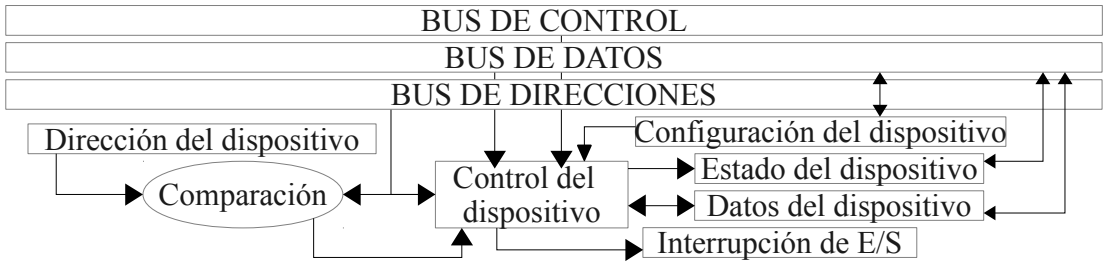


Figura 2.5: Diagrama a bloques del control de un dispositivo de E/S

La figura 2.5 ilustra el mecanismo necesario para cada dispositivo de entrada y salida: el registro de configuración del dispositivo establece el funcionamiento del mismo, si sirve de entrada o de salida, o ambos; el registro de datos del dispositivo es la información que lee o escribe el procesador; el registro de estado del dispositivo indica si está listo, continúa trabajando, está en espera o se encuentra bloqueado; la dirección del dispositivo es una dirección mapeada en memoria de datos que permite el acceso al dispositivo; el control del dispositivo, es el mecanismo que controla las tareas del dispositivo de E/S; la interrupción es el indicador para avisar a la CPU que el dispositivo de E/S está listo para leer o escribir, o cualquier operación que deba ser atendida por el procesador.

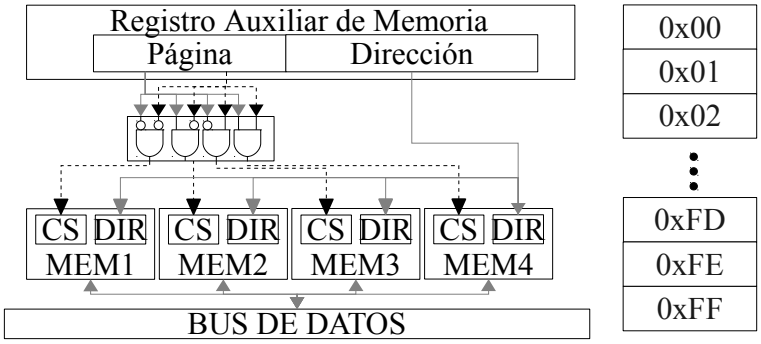
Además de controlar periféricos o dispositivos de entrada y salida (impresora, teclado, ratón, pantalla, etc.) la unidad de E/S también se utiliza para la comunicación entre máquinas, que puede ser serial, o paralela. Algunos ejemplos de periféricos pueden ser:

* UART (del inglés *Universal Asynchronous Receiver-Transmitter*, *Universal Asíncrono Receptor Transmisor*): es la interfaz de comunicación serial para transmitir o recibir un byte de forma asíncrona.
* Puertos de entrada y salida (I/O): es la entrada o salida de datos en forma paralela, la lectura o escritura de bits. Puede existir una interrupción al actualizarse un puerto, es decir, al modificar el valor que tenía.
* Temporizador: se trata de un contador que al llegar a un límite establecido genera una interrupción, es un evento que tiene asociado una subrutina de atención a este tipo de componente.

2.4.2. Organización de la memoria

Se destacan dos formas de manejar la memoria, lineal o por páginas. En la forma lineal, las celdas de memoria se tratan como una sucesión consecutiva, y cada celda se identifica con su dirección. Una página es una porción de memoria de tamaño fijo, una sucesión de varias páginas se organizan de forma consecutiva, y la dirección de una celda se compone del número de página y su dirección.

Algunas localidades de memoria referencian registros que se encargan de configurar al procesador. La figura 2.6 muestra los tipos de organización para la memoria.



(a) Por páginas (b) Lineal

Figura 2.6: Organización de la memoria.

2.4.3. Interrupciones

Son peticiones o avisos que debe atender el procesador, interrumpe el estado normal de la CPU, direccionando el flujo del programa a las rutinas que atienden dichas peticiones. Las interrupciones pueden ser manejadas por orden de importancia o en el orden al sondear las señales. La atención a interrupciones puede activarse por algún registro de control del procesador. La figura 2.7 muestra el mecanismo necesario para las interrupciones atendidas por prioridad, estas son recibidas por el demultiplexor que identifica la mayor prioridad. El

manejo de interrupción se habilita con una bandera y esta bandera se establece en software mediante un registro de función especial. Al activarse la bandera de interrupción puede que una o más peticiones necesiten servicios del procesador.

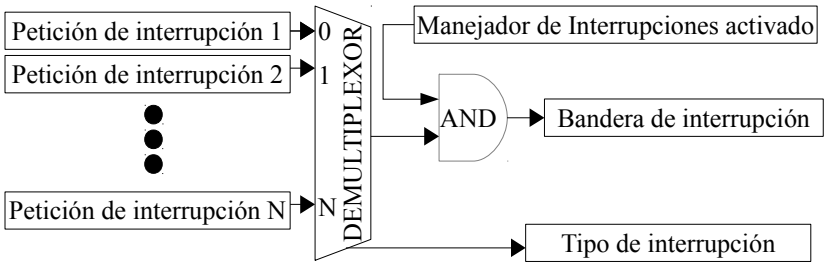


Figura 2.7: Demultiplexor de prioridad de interrupciones

El control de interrupciones se puede abordar teniendo en cuenta una jerarquía de prioridad dada a las interrupciones o por sondeo atendiendo a las interrupciones en el orden de barrido que verifica las señales, con el fin de atender al conjunto de dispositivos de E/S. El *contexto del programa* es el conjunto de información almacenada en los registros (RPG, CP, SP, etc.), se tiene que salvar y restaurar cuando se utilizan interrupciones.

El manejo de interrupciones **por prioridad** se trata de anidar las interrupciones, en caso de llegar una interrupción de mayor prioridad a la que se está ejecutado, se salva el contexto de la interrupción de menor prioridad y se inicia el procedimiento de manejo de interrupción de la que tiene mayor prioridad. Una vez que se termine un procedimiento de manejo de interrupción se pasa a la siguiente rutina de interrupción que le sigue en cuanto a su prioridad de interrupción, restaurando su contexto que fue interrumpido, hasta llegar a la ejecución del programa de forma normal.

El manejo de interrupciones **por sondeo** o *sin prioridad* responde secuencialmente en el orden en que el manejador sondea las interrupciones ejecutando cada procedimiento de manejo de interrupciones hasta que se hayan atendido todas las peticiones de interrupción, retornando de forma normal al programa que se estaba ejecutando y que fue detenido.

El **vector de interrupciones** es una estructura que tiene varias direcciones de memoria de programa asociados a periféricos o componente que genera una señal de interrupción, cada dirección de memoria apunta a la instrucción de inicio de la rutina de manejo de interrupción correspondiente a cada señal. Cuando la interrupción se genera, el CPU termina la instrucción que está actualmente ejecutándose y el contador de programa se actualiza con la dirección de programa relacionada con la interrupción. Este proceso es lo contrario a tener solo una dirección de memoria para iniciar el procedimiento de interrupción y verificar todas las posibles fuentes de la señal.

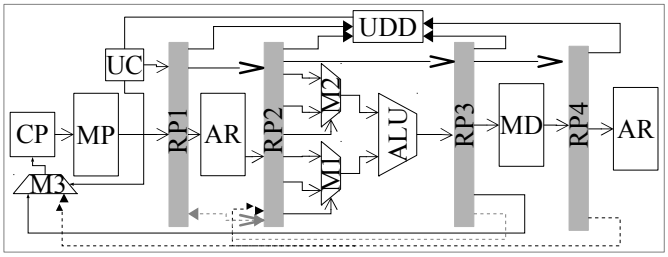
2.4.4. Segmentación o *pipeline*

La segmentación consiste en traslapar la ejecución de varias instrucciones en el diseño de ciclos sencillos, comenzando la siguiente instrucción antes de que la previa haya concluido; lo

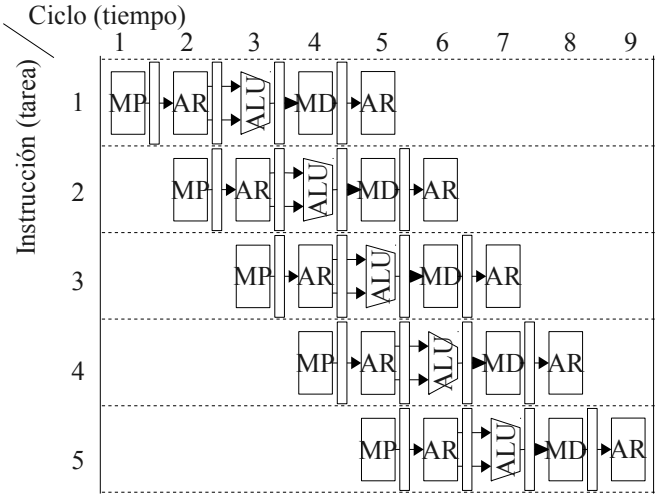
anterior conduce a la organización encauzada (*pipelined*) o superencauzada (*superpipelined*). La Figura 2.8(a) muestra un procesador segmentado por cuatro registros de *pipeline* (RP_1, RP_2, RP_3 y RP_4) que almacenan las señales de control así como el tipo de instrucción, el tipo de operación, los selectores de registros origen y destino, los datos que almacenan los registros, el resultado de la ALU, etc.

En la Figura 2.8(b) se observa la paralelización que se logra a nivel del ciclo de instrucción, en el ciclo cinco del reloj todas las etapas están trabajando. Los saltos (*jump*), llamadas (*call*) y retornos (*return*) a subrutinas son caso especiales, cuando se actualiza el CP a la nueva dirección las instrucciones anteriores al salto o llamada son anulados, se inicia de nuevo el pipeline en el ciclo uno de la figura 2.8(b).

La unidad de dependencia de datos (UDD) se encarga de revisar los registros de segmentación, si la información contenida en un registro de *pipeline* ya no está actualizada, es decir, que los datos fuentes para realizar una operación se han modificado. Se necesita realizar una revisión en los registros de segmentación que tienen el dato necesario, por medio de la interconexión de multiplexores se corrige este problema cargando los datos correctos, de tal forma que no se vea afectado el flujo del programa con resultados erróneos.



(a) Procesador segmentado.

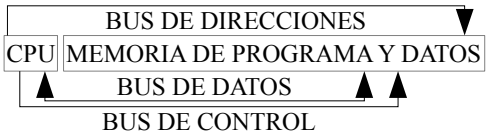


(b) Diagrama tarea-tiempo

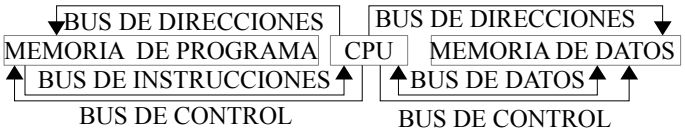
Figura 2.8: Segmentación

2.4.5. Arquitectura de computadoras

La memoria de una computadora almacena instrucciones y datos. Las instrucciones deben pasar secuencialmente a la CPU para su decodificación y ejecución, en tanto que algunos datos en memoria son leídos por la CPU y otros son escritos en la memoria desde la CPU. La organización de la memoria y su comunicación con la CPU son dos aspectos que influyen en el nivel de prestaciones de la computadora. La arquitectura Von Neumann (AN) utiliza una memoria única para instrucciones y datos. La arquitectura Harvard (AH) utiliza memoria separadas para instrucciones y datos. Ambos modelos generales de hardware son ilustrados en las figura 2.9. La AN requiere menos líneas que la AH para conectar la CPU con la memoria, lo cual supone menos buses.

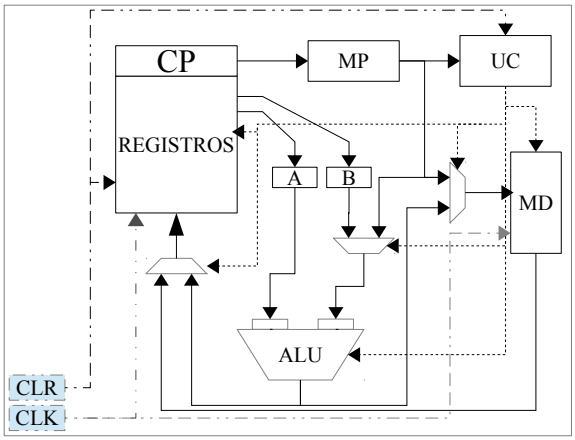


(a) Modelo Von Neumann.

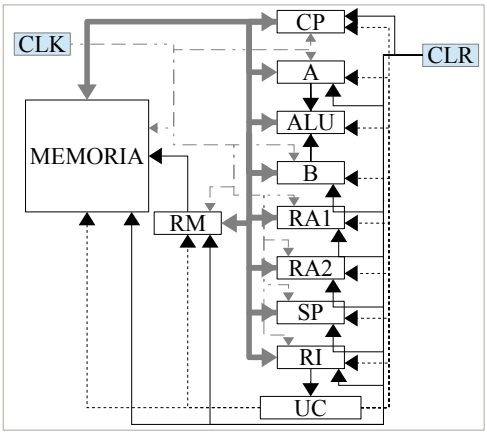


(b) Modelo Harvard.

Figura 2.9: Modelos de memoria para microcontroladores



(a) RISC y modelo Harvard



(b) CISC y modelo Von Neuman

Figura 2.10: Ejemplo de arquitecturas

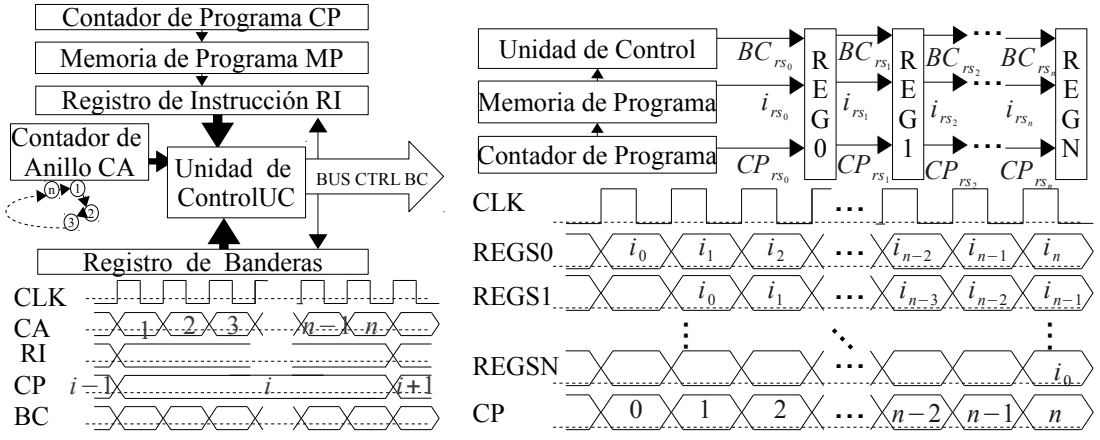
Tomando en cuenta el repertorio de instrucciones, lo cual afecta la arquitectura del CPU, se encuentran dos modelos de computadoras. La *Computadora con un Conjunto Complejo de Instrucciones* (del inglés *Complex Instruction Set Computer*, CISC) y *Computadora con un Conjunto Reducido de Instrucciones* (del inglés *Reduced Instruction Set Computer*, RISC). El RISC cuenta con pocas instrucciones que son simples, toma menor tiempo en ejecutarse

cada una, por lo regular las instrucciones de carga y almacenamiento tienen acceso a la memoria. Los tipo CISC cuenta con un amplio rango de instrucciones, además de contar con instrucciones sencillas cuenta con algunas que permiten operaciones complejas, permitiendo en muchos casos acceder a la memoria más de una vez.

En la figura 2.10 se muestra un ejemplo de los dos modelos de arquitectura. La configuración del bus puede ser de dos maneras: la primera forma puede compartir un mismo bus para datos, instrucciones o direcciones; la segunda forma pueden usar varias líneas direccionadas con multiplexores para datos, direcciones o instrucciones. Por lo general la primera tendencia del bus es aplicada en los CISC y la segunda en los RISC. La arquitectura RISC presenta frecuentemente una organización tipo Harvard, cuenta con una Memoria de Datos (MD) y una Memoria de Programa (MP). La arquitectura CISC presenta frecuentemente una organización Von Neumann, con los datos y el programa en una misma memoria; requiere de un registro auxiliar de memoria y el ciclo de instrucción emplea un contador de anillo que selecciona el conjunto de microinstrucciones para cada etapa del ciclo de instrucción.

2.4.6. Unidad de control

Las unidades de control (UC) son muy parecidas a una caja negra; entra un conjunto de señales; el bus de instrucción, el valor de un registro o el contador de anillo; y la salida se carga en el bus de control. El contador de anillo se utiliza en la arquitectura CISC, este referencía la microinstrucción y las señales de control asociadas a una etapa del ciclo de instrucción. El contador de programa indica la localidad de instrucción en la memoria de programa. La memoria de programa regresa la instrucción referenciada por el contador de programa depositándola en el bus de instrucción.



(a) Unidad de control en un CISC. (b) Unidad de control en un RISC.

Figura 2.11: Configuración de la unidad de control.

La figura 2.11(a) muestra la configuración de la unidad de control en una arquitectura CISC, el bus de control (BC) se actualiza constantemente, manejado directamente sin intermediarios por la unidad de control. El contador de anillo (CA) indica a la UC que señales

(microinstrucciones) asociadas al código de operación de la instrucción deben ponerse en el bus de control.

La figura 2.11(b) muestra la configuración de la unidad de control en una arquitectura RISC que usa el *pipeline* o segmentación, el desplazamiento de la información se almacenada en los registro REG_i cada uno asociado a la etapa i del ciclo de instrucción (CI). La información del bus de control (todas las microinstrucciones de todas las etapas del CI asociadas a una instrucción) obtenidas de la UC, el contador de programa (CP), el contenido del bus de instrucción (BI) e información adicional se almacena en los registros segmentados. No es necesario el contador de anillo porque se tiene en su lugar los registros de segmentación que se actualizan en cada ciclo de reloj. Se necesitan $n+1$ ciclos de reloj para ejecutar una instrucción. Existen otros diseños en el RISC en los que no se ocupan los registros de segmentación, donde se intenta ejecutar en un solo ciclo de reloj toda la instrucción.

2.5. Instrucción

Dependiendo del conjunto de aplicaciones para los que un procesador está construido, se define un repertorio de instrucciones que cubre con las necesidades de un diseñador de aplicaciones. Las instrucciones son las tareas que puede realizar un procesador.

El ciclo de instrucción es el conjunto de etapas necesarias para atender una instrucción. Las etapas realizadas en el ciclo de instrucción son:

1. **Buscar** la instrucción en memoria de programa, obteniendo la instrucción que tiene que ejecutarse.
2. **Decodificar** las instrucciones, donde la CPU examina la instrucción que debe ser atendida.
3. **Ejecutar la instrucción**: la unidad de control configura todos los elementos por medio del BUS de control para realizar la operación requerida por la instrucción.
4. **Leer o escribir en la MD**: almacenar o leer datos de la memoria de datos.
5. **Leer o escribir en el AR**: almacenar o leer datos del archivo de registros.

Los modos de direccionamiento sirven para especificar donde encontrar los argumentos que una instrucción necesita al ejecutarse. Algunos de estos modos de direccionamiento son:

- Implícito: no tiene argumentos, sólo es el código de operación. Se conoce toda la información necesaria para ejecutar la instrucción. Ej.: *nop*; CP←CP+1
- Registro: los datos que necesita la instrucción están contenidos en los registros. Ej.: $jump\ \$r1$; CP← $\$r_1$.
- Inmediato: uno de los argumentos es una constante, es el valor que utiliza la instrucción. Ej.: $load\ \$r1,\ \#A2h$; $\$r1 \leftarrow A2h$.
- Directo: un argumento dado es una dirección cuya localidad de memoria es el origen o destino necesario para la instrucción. Ej.: $load\ \$r1,\ @A2h$; $\$r1 \leftarrow \text{MD}[A2h]$.

- Indirecto por registro: el contenido de un registro contiene la dirección de memoria, que indica el origen o destino de un argumento de la instrucción. Ej.:
  load $\$r1,\ @\$r2;\ \$r1 \leftarrow \mathrm{MD}[\$r2]$.
- Base indexado: utiliza dos registros, uno tiene la dirección de memoria base y otro un desplazamiento o índice, cuya suma resultante es la dirección de memoria en donde está almacenado un argumento de la instrucción. Ej.:
  load $\$r1,\ @\$r2,\ *\$r3;\ \$r1 \leftarrow \mathrm{MD}[\$r2 + \$r3]$.

La unidad de control utiliza las señales de control o **microinstrucción** para cubrir con las etapas antes descritas. No necesariamente cada etapa debe realizarse en un ciclo de reloj, pueden combinarse dos o más en un ciclo de reloj, depende mucho de la organización del hardware.

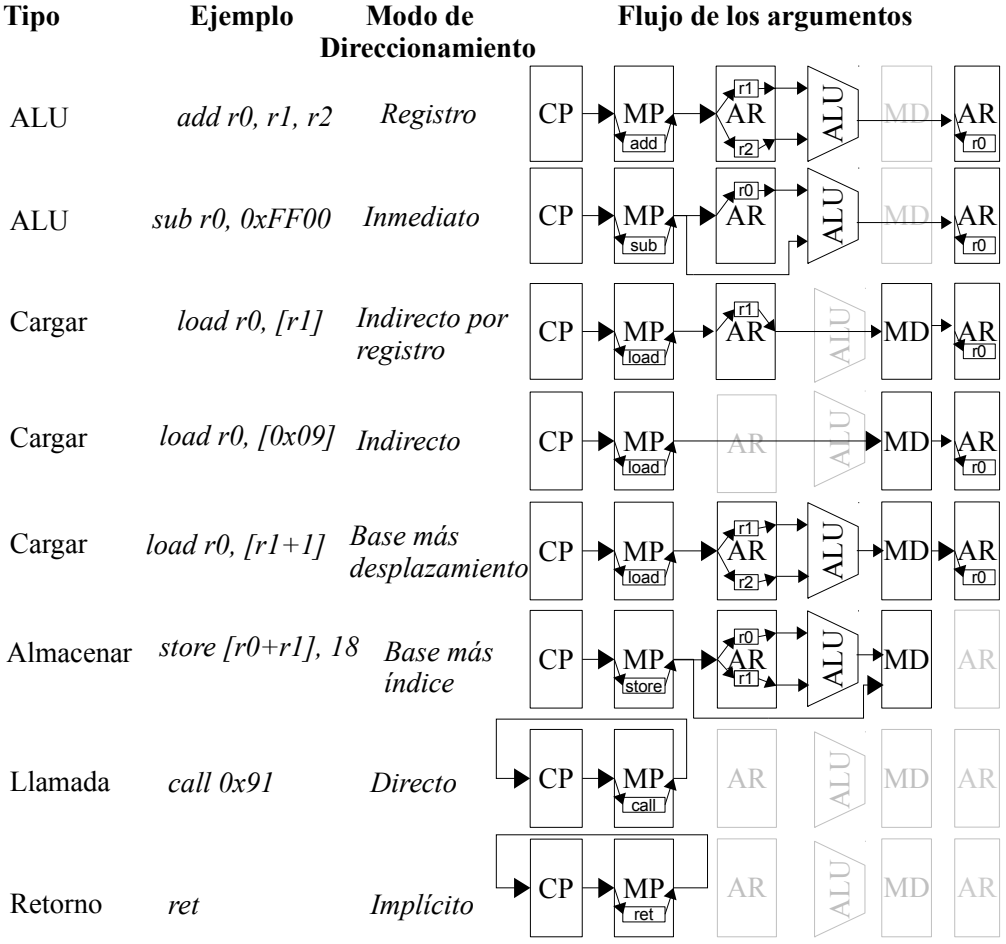


Figura 2.12: Tipos de instrucción y modos de direccionamiento

La figura 2.12 ilustra algunos de los modos de direccionamiento, además de indicar el tipo de instrucción de cada ejemplo. Las figuras ilustran los componentes involucrados, contador de programa (CP), memoria de programa (MP), archivo de registros (AR), ALU y memoria de datos (MD).

Una clasificación posible para las instrucciones se enlista a continuación:

* Acceso a memoria:
  - Load: cargar un dato de la memoria a registro.
  - Store: almacenamiento de registro a memoria.
  - Pila: para manejar la estructura datos tipo pila (LIFO último en entrar, primero en salir) se utiliza un registro como puntero, para almacenar un dato se obtiene de forma inmediata o por registro, al recuperar el dato solamente se almacena en un registro.
* Transferencia de control:
  - Saltos condicionales: de acuerdo al estado de una bandera salta a una dirección de memoria de programa.
  - Saltos incondicionales: salta directo a una dirección de memoria de programa.
* Subrutina:
  - Llamada a subrutina: se actualiza el contador del programa con una nueva dirección donde comienza una rutina. Se utiliza algún método para salvar la dirección del contador de programa antes de la actualización.
  - Retorno de subrutina: se actualiza al contador de programa con la dirección de la instrucción siguiente a la llamada de la subrutina, para restaurar el flujo normal del programa.
* Interrupciones:
  - Retorno de interrupciones: es parecido al anterior pero con la diferencia de que se tiene que notificar mediante esta instrucción al manejador de interrupciones que la interrupción ha terminado y que continue con el flujo de ejecución, bajo el criterio que éste tome.
* Utilizando la ALU:
  - Aritmética: suma, resta, multiplica, división, etc.
  - Corrimiento o desplazamiento: realiza un desplazamiento de n bits a la izquierda o la derecha, de acuerdo a los valores almacenados en los acumuladores o seleccionados del archivo de registros.
  - Lógica: realiza las operaciones and, or, xor, not, nand, nor y xnor.
  - Comparación: es una resta pero no guarda el resultado, y modifica los bits del registro de banderas.
* Auxiliares:
  - Reiniciar el temporizador guardián, poner en cero el contador de este componente evitando el reinicio del procesador.
  - La instrucción de no operación que soló consume tiempo durante la ejecución del programa.
  - Las instrucciones que detienen de forma definitiva el flujo del programa. Ej.: la instrucción *halt*.
  - Las instrucciones que esperan un evento externo o interno, pasa a un estado de bajo rendimiento o de bajo consumo de energía hasta que suceda el evento deseado. Ej.: la instrucción *sleep*.

Capítulo 3

Trabajos relacionados

En este capítulo se proporciona una pequeña descripción de trabajos relacionados con los procesadores, microcontroladores y ensambladores. Se exponen algunas de las características del software y hardware, resaltando las capacidades para su uso en aplicaciones.

3.1. Arquitectura de 32 bits

3.1.1. UAM RISC-II

El diseño propuesto por Zamudio [4] es el «UAM RISC-II» es la versión mejorada de la arquitectura de 32 bits desarrollada originalmente en el Departamento de Ingeniería Eléctrica de la UAM, las etapas segmentadas del ciclo de instrucción coinciden con el diseño de Santana [5]. La unidad de dependencia de datos es el *Forward UNIT*. No considera el manejo de interrupciones, para las llamadas a subrutina utiliza 5 instancias de contadores. Hace una comparación del procesador en los FPGA XC400XL y XCV1000.

3.1.2. LEON 2

LEON 2 [6] es un procesador de 32 bits, su arquitectura se basa en los SPARC V8 conforme al estándar de la IEEE-1754. Dicho estándar define el conjunto de instrucciones, el modelo de registro, tipo de datos, el código de operación de las instrucciones, la interfaz del coprocesador para esta arquitectura, describe meras sugerencias de la sintaxis del lenguaje ensamblador y da una idea para extender la arquitectura.

Este procesador cuenta con unidad de punto flotante FPU que realiza operaciones de números reales con el formato de precisión simple (32 bits) del estándar IEEE 754. La figura 3.1 muestra los bloques que conforman la arquitectura LEON 2. El núcleo del procesador se basa en operaciones realizadas por la unidad de enteros UI (*Integer Unit*), es decir, las operaciones de la ALU soló operan con la representación de números enteros, cuyo núcleo RISC cuenta con 5 etapas segmentadas que conforma el *pipeline*.

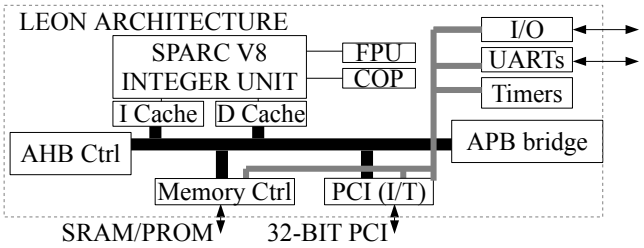


Figura 3.1: Diagrama a bloques de la arquitectura LEON.

Su diseño está dirigido para aplicaciones incrustadas sobre un chip. Cuenta con temporizadores de 24 bits, un temporizador de reinicio o "perro guardián", para la comunicación serial utiliza UARTs con transferencia de datos de 8 bits, implementa el estándar de comunicación de redes Ethernet MAC, además de contar con un interfaz para el bus PCI (interconexión de componentes periféricos), puertos paralelos de entrada y salida de 16 bits. VHDL LEON tiene la licencia libre LGPL. Implementa un *pipeline* de 5 etapas: búsqueda de la instrucción, decodificación, ejecución, memoria y escritura.

Tiene un controlador de interrupciones que es utilizado para priorizar y propagar solicitudes de interrupciones, maneja 15 interrupciones internas y externas, el controlador secundario conectado en cascada permite un máximo de 32 interrupciones, separa la memoria de dato y el de instrucciones. Las excepciones manejadas son: reinicio, error de memoria, error en la etapa de la búsqueda de la instrucción, ejecución de instrucciones en privilegio en modo usuario, ejecución de instrucciones en punto flotante cuando la unidad de punto flotante o FPU esta inhabilitada, etc.

El número de ciclos por instrucciones son: de 1, 2, 3, 4 ciclos y 35 para la multiplicación y división, con y sin signo. Implementa la especificación de arquitectura de bus avanzada para microcontroladores (*Advanced Microcontroller Bus Architecture* AMBA) que define una red de comunicación sobre un chip: Bus de alta velocidad avanzado (*Advanced High-speed Bus* AHB) y bus de periféricos avanzados (*Advanced Peripheral Bus* APB).

3.1.3. RISC de un ciclo de reloj por instrucción

En el artículo [7] se menciona la implementación de un procesador MIPS (millones de instrucciones por segundo) RISC de 32 bits que realiza la ejecución de una instrucción en un único ciclo de reloj, su diseño se basa en la rapidez. El único ciclo de reloj se divide en cinco etapas: la búsqueda de la instrucción en la memoria del programa, decodificar la instrucción con ayuda de la unidad de control, realizar una operación con la ALU, acceder y almacenar en la memoria de datos y archivo de registros. Se describe el diseño y los componentes que integran este procesador RISC para poder ejecutar cada instrucción en un solo ciclo de reloj.

Todos los módulos en el diseño se codifican en VHDL, ya que es una herramienta muy útil, con su concepto de simultaneidad para hacer frente con el paralelismo de hardware digital. En una máquina RISC, el conjunto de instrucciones se basa en un enfoque de carga

y almacenamiento. Sólo las instrucciones de cargar *load* y almacenar *store* tienen acceso a la memoria RAM. En las demás instrucciones sus argumentos están en los registros o viene incluida en la palabra de la instrucción, esta es la clave para un solo ciclo en la ejecución de las instrucciones. Maneja tres modos de direccionamiento: por registro, directo e inmediato.

3.1.4. Núcleo del procesador de 32 bits

El RISC descrito en [8] es un diseño que utiliza la arquitectura *pipeline*, a través de este se puede mejorar la velocidad de la operación, cuenta con cinco ciclos de reloj para cada ciclo de instrucción y cuatro registros de pipeline que segmentan las cinco etapas que conforman al ciclo de instrucción que son: fetch, decodificación, ejecución, actualizar memoria y registros.

Destaca la importancia que debe darse a la dependencia de datos descrita en el diseño del procesador. Cuenta con una unidad de detección de peligros, con el fin de asegurarse que las instrucciones se ejecutan con el conjunto de datos correctos. Al tomar las medidas adecuadas en la dependencia de datos, se puede retrasar la ejecución de cualquier instrucción tantas veces como sea necesario para garantizar la correcta ejecución de las instrucciones.

3.2. Arquitectura de 16 bits

3.2.1. RISC-1oo2

La arquitectura propuesta por Ummar [9] propone la integración de dos procesadores RISC de 16 bits en un FPGA, proponiendo un nivel alto de integración (ISL), el estándar aplicado es el IEC 61508 el cual establece métodos completos para el análisis y determinación de requisitos de seguridad para ser aplicables con sistemas electrónicos programables.

El propósito de los dos controladores conectados es minimizar el efecto de los fallos peligrosos, cada uno cuenta con unidades de entrada y salida independientes. La finalidad de una arquitectura doble (1oo2) es que proporcione una integridad de alta seguridad. La arquitectura 1oo2 consta de dos procesadores RISC independientes, ambos procesadores están conectados entre sí de manera que la salida está disponible, sólo si las señales son idénticas ambos procesadores están funcionando correctamente. La figura 3.2 muestra las conexiones con el control que genera una alarma, si el valor de los buses de datos es distinto en caso contrario el funcionamiento es el correcto.

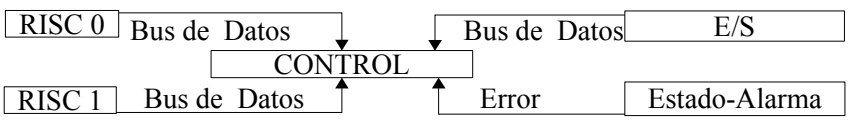


Figura 3.2: Configuración del control de la arquitectura *RISC-ioo2*

3.2.2. Procesador CISC de 16-Bits

En el artículo de Tieju [10] muestra un diseño del 16-Bit microprocesador didáctico (*Teaching Microprocessor*) que ha sido desarrollado para propósitos escolares, su repertorio cuenta con 47 instrucciones con la posibilidad de ser expandido; tiene un registro que funciona como puntero de pila, soporta siete modos de direccionamiento, es un CISC con tres ciclos de reloj para ejecutar cada instrucción. Este diseño describe el diseño de un CISC detallando su funcionamiento en forma general para poder ser recreado. El objetivo de este procesador es enseñar a partir del punto de vista del diseño, la estructura y el principio de la computadora de las partes con el todo. Al ser abierto beneficia a los estudiantes con un conocimiento en profundidad de la estructura interior del microprocesador y la experiencia práctica en el diseño de microprocesadores.

3.2.3. Diseño de un procesador con HDL

El diseño propuesto por Santana [5] es un RISC de 16 bits. La segmentación divide al ciclo de instrucción en las cinco etapas de búsqueda, decodificación, ejecución, actualización de memoria y actualización de registros. Es utilizando el lenguaje de descripción de hardware Verilog, sobre un FPGA SPARTAN 2 de XILINX. Cuenta con cuatro registros de *pipeline*, conectados a la unidad que controla y a la unidad de dependencia de datos que evalúa constantemente el contenido de los registros de segmentación atendiendo correctamente a cada instrucción. Presenta una propuesta de la arquitectura de un sistema para el procesamiento del algoritmos de detección de bordes con el método *Canny*, ya que es un algoritmo muy utilizado en el procesamiento de imágenes.

3.2.4. CPU86

CPU86 8088 FPGA IPCORE descrito en [11] se implementa un procesador 8088, el núcleo es compatible con un procesador iAPX8088 y es posible implementarse en cualquier FPGA. Algunas frecuencias de reloj utilizadas son de 5 MHz (0.33 MIPS), 8 MHz (0.66 MIPS) y 10 MHz (0.75 MIPS). Un ancho de bus de 16 bits, direcciona memoria de 1 MB, el CPU86 es ideal para sistemas incrustados. Tiene un modelo de 256 bytes ROM, 256 KB SRAM, cuenta con una unidad asíncrona receptor-transmisor (UART) para realizar la comunicación serial. Está disponible con una licencia GPL.

Las recomendaciones para construir un sistema con la CPU86 se ilustran en la figura 3.3 se visualizan las conexiones del procesador con algunos periféricos. Las interrupciones son imprescindibles en cualquier sistema, el controlador programable de interrupciones es el circuito integrado CI con la etiqueta 8259A, vectoriza las interrupciones por prioridad y pueden conectarse en cascada para extender el número de interrupciones. Para la comunicación serial se tienen dos puertos UART utilizando el CI 16550. Para la comunicación paralela de entrada y salida el puerto es un CI PPI8255. La unidad de temporizadores programable es el CI

PIT8254, es usado por las interrupciones periódicas del sistema operativo, como refrescar la memoria, etc. El núcleo del procesador al igual que los componentes que describen diversos CI está disponible en lenguaje VHDL.

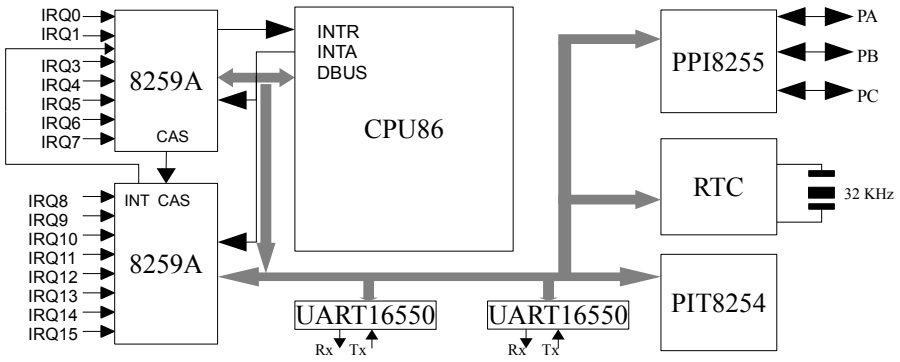


Figura 3.3: Diagrama a bloques del CPU86-8088

Interrupción	Descripción
INT10	Salida de teletipo, el carácter escrito en la UART.
INT21	Leer carácter de la UART.
INT1A	Obtener/establecer la hora del sistema.
INT16	Leer el teclado

Tabla 3.1: Servicio de interrupciones

MON88 es el software de depuración utilizada para este CPU86, es similar al debug.exe para MSDOS, es flexible para adaptares a otros procesadores de la familia x86, el programa cuenta con un cargador, desensamblador y algunos servicios de interrupciones INT 21/10/16/1A, la descripción de las interrupciones se ven en la tabla 3.1. El desensamblador está basado en la versión 0.1 para la arquitectura x86 realizada por David Moore "disasm.c" [12].

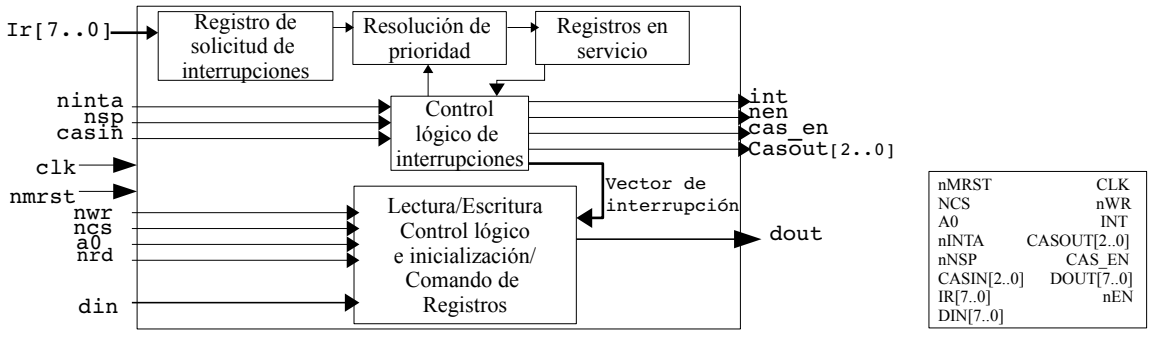


Figura 3.4: Diagrama a bloques del controlador programable CI 8259

El CI 8259 es un controlador de interrupciones descrito en VHDL disponible para la CPU86 compatible a nivel binario del 8088/8086, la mayoría del software del 8086 debería

funcionar sin problemas. Existe una versión comercial HTL80186, que fue hecha por la misma empresa que proporciona la CPU86. Para usar el lenguaje C se pueden utilizar los compiladores OpenWatcom C, Turbo-C y muchos otros.

El manejador de interrupciones programable CI 8259 [13] desarrollado por Intel ofrece ocho niveles de interrupciones individuales enmascarables. Ampliable hasta 64 interrupciones. Ofrece un esquema de resolución flexible. Ofrece modos programables de interrupción y un vector de direcciones. Ocho líneas para cada bus de datos: de entrada *din*[7..0] y de salida *dout*[7..0]; sus señales de control son *ncs*, *nrd*, *nwr*, *int* y *ninta* cuya función es configurar y operar este componente. Las figura 3.4 muestra el símbolo y el diagrama de bloques interno del CI 8259. Este componente puede adquirirse por un precio con ALTERA o HT-Lab. Maneja un bus para tener varios 8259 en cascada, es una configuración maestro esclavo que permite extenderlo. El control de este componente es una máquina de estados que considera el modo de operaciones por prioridad o sondeo, incluye internamente los registros necesarios para su vector de interrupción y de control de configuración.

3.2.5. DCPU-16

Existen muchos programas de vídeo juegos antiguos que siguen perdurando, gracias a los emuladores estos viejos programas de este tipo, pueden seguir funcionando en las computadoras personales actuales. Un moderno juego de conquista y exploración espacial "0x10c" hace uso en su trama de un tipo de máquina denominada DCPU-16 como controlador de naves espaciales, donde son virtualizadas, permitiendo al jugador añadir nuevas soluciones dentro de la trama del juego en lenguaje ensamblador.

En [14] se proporcionan los códigos fuentes de la máquina virtual, es decir, su emulador, el ensamblador y desensamblador, en varios lenguajes de programación: python, java, perl, ruby, go, javascript, etc. La DCPU-16 es totalmente programable por los jugadores, la empresa de este videojuego proporciona documentación [15], la comunidad de jugadores ha creado toda una biblioteca de programas informáticos para esta arquitectura. La figura 3.5 muestra un ejemplo de aplicación para la máquina DCPU-16.

Figura 3.5: Ejemplo de aplicación para la máquina DCPU-16.

El CPU virtual de microordenador DCPU-16 maneja 16 bits por palabra, 0x10000 (65536) palabras de RAM, ocho registros de propósito general A, B, C, X, Y, Z, I, J, puntero a pila (SP), contador de programa (PC), extra (EX), dirección de interrupción (IA). Los modos de direccionamiento son directo, absoluto, relativo al CP, indirecto e indexado. Cinco bits

de código de operación ($2^5 = 32$). El DCPU-16 llevará a cabo a lo sumo una interrupción a la vez. Si las interrupciones se activan de forma múltiple al mismo tiempo, se añaden a una estructura tipo cola (FIFO). La estructura cuenta con un límite de 256 interrupciones. Las tablas 3.2 y3.3 muestran algunas instrucciones que esta máquina virtual ejecuta.

Mnemónico	Condición	Mnemónico	Condición	Mnemónico	Condición
IFB b,a	$b \wedge a)! = 0$	IFC b,a	$(b \wedge a) == 0$	IFE b,a	$b == a$
IFN b,a	$b! = a$	IFE b,a	$b == a$	IFN b,a	$b! = a$
IFA b,a	$b > a$	IFG b,a	$\|b\| > \|a\|$	IFL b,a	$\|b\| < \|a\|$
IFU b,a	$b < a$				

Tabla 3.2: Instrucciones de salto condicional de la máquina DCPU-16. Cuando no se cumple la condición CP+=1 y cuando se cumple CP+=2.

Mnemónico	Descripción	Mnemónico	Descripción
MUL b,a	$ex, b \leftarrow \|b\| * a$	MULI b, a	$ex, b \leftarrow b * a$
DIV b, a	$ex, b \leftarrow \|b\|/a$	DIVI b, a	$ex, b \leftarrow b/a$
MOD b, a	$b \leftarrow \|b\| \%\|a\|$	M0DI b,a	$b \leftarrow b\%a, [a = 0] \rightarrow [b \leftarrow 0]$
AND b, a	$b \leftarrow b \wedge a$	BOR b, a	$b \leftarrow b\|a$
XOR b, a	$b \leftarrow b \oplus a$	SET b,a	$b \leftarrow a$
ADD b,a	$b \leftarrow b + a$	SUB b,a	$b \leftarrow b - a$
SHR b,a	$b \leftarrow b >>> a$	SHL b,a	$b \leftarrow b <<< a$

Tabla 3.3: Instrucciones de suma, resta, desplazamiento de bits e intercambio de datos, multiplicación, división, módulo y lógicas de la máquina DCPU-16.

3.3. Arquitectura de 8 bits

3.3.1. RISC incrustado de 8 bits

El artículo [16] propone un RISC de 8 bits con la finalidad de ser para uso industrial, este puede ser programado a través de una interfaz de comunicación serial, cuenta con tres puertos de propósito general de entrada y salida, temporizadores, y una interfaz de comunicación serial. Además incorpora en su arquitectura un vector de interrupciones, pude recibir interrupciones externas, además de las que provienen de los temporizadores y la comunicación serial. La figura 3.6 muestra una configuración completa de un microcontrolador incrustado para ser practico en sistemas de control incrustado.

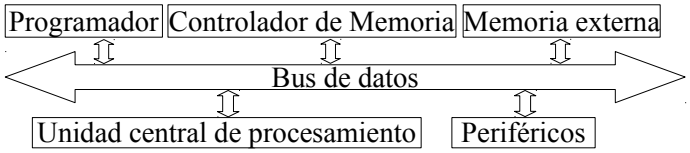


Figura 3.6: Componentes de un microcontrolador RISC incrustado.

El principal objetivo de este trabajo es presentar la estructura del microcontrolador de 8 bits y sus bloques funcionales; el trabajo está centrado en la construcción del núcleo del con-

trolador. La filosofía de la arquitectura tipo RISC trata de reducir el número de instrucciones y unificar el número de conexiones entre los bloques funcionales. Además, el decodificador de instrucciones se puede simplificar dependiendo del criterio de su implementación, convirtiéndolo en un procesador pequeño y rápido.

3.3.2. Procesador de 8 bits

El microprocesador basado en FPGA debe ser flexible, programable y confiable. Los PLD (dispositivos lógicos programables) facilitan la creación de prototipos para complejos diseños electrónicos. En el trabajo descrito en [17] aborda una discusión de cómo obtener un alto rendimiento en los sistemas incrustados y la computación en tiempo real, asegurando que se puede conseguir a través del uso de la tecnología FPGA.

Un procesador incrustado se destaca por su velocidad y capacidad de programación que son las principales características que determinan su rendimiento. El procesador desarrollado es una arquitectura tipo RISC con un modelo de memoria Von Neumann. Maneja tres formatos de instrucciones cada uno con un propósito: uso de los registros, instrucciones de salto, acceso a las unidades de entrada y salida y la instrucción de alto *halt*. Cuenta con cuatro registros de 4 bits, una memoria que almacena palabras de 8 bits direccionada por 16 bits. La ALU es un circuito digital que calcula operaciones aritméticas como: suma, resta, desplazamiento de bits y o exclusiva. Se compone de un sumador completo de 4 bits y una unidad utilizada para obtener el complemento a dos de números para realizar sustracciones simples.

3.3.3. PicoBlaze

El microcontrolador PicoBlaze^{TM} [18] es un CPU tipo RISC de 8-bits, utilizado especialmente para su implementación en FPGAs de XILINX. La versión KCPSM3 está optimizada para Spartan 3, la versión KCPSM6 está optimizada para Virtex 6 y Spartan 6. El microcontrolador PicoBlaze es extremadamente flexible. La funcionalidad básica es fácilmente ampliable y reforzada por su comunicación con el exterior a través de sus puertos de entrada y salida. Soporta hasta 256 puertos de entrada y 256 puertos de salida o una combinación de los puertos de entrada/salida. Cuenta con una pila en hardware de hasta 31 contadores de subrutina. Permite la opción de interrupciones para que el microcontrolador maneje eventos asíncronos externos.

La empresa se encarga de poder garantizar el uso de este microcontrolador con diferentes periféricos como: VGA, puerto serial, PS/2 (teclado y mouse), pantallas LCD. Los dispositivos lógicos programables en los que puede trabajar son en FPGA y CPLD que produce la empresa XILINX. La ventaja de esta clase de CPU incrustado cubren las necesidades de tener un procesador sin añadir hardware adicional, todo los recursos se implementan en un solo PLD. El código en VHDL está bajo la licencia BSD.

3.4. Software

3.4.1. SC123

Silverman [19] describe que para cualquier lenguaje de programación para propósitos didáctico, incluido el lenguaje ensamblador, debe contar con tres elementos que son: simplicidad (lo más simple que sea posible), regular (reglas regulares, sin excepción, sean fáciles de aprender, describir e implementar), y ortogonalidad (funciones independientes que deben ser controladas por mecanismos independientes). El proyecto da una tabla comparativa de varios entornos de trabajo entre los cuales destaca la máquina virtual SC123, el cual tiene su propia arquitectura de conjunto de instrucciones (ISA), que es usado para aprender a usar el lenguaje ensamblador y está disponible en [20].

Esta herramienta se extiende a los cursos de arquitectura de computadoras, lenguajes de programación y lenguaje ensamblador. La finalidad es que el estudiante distinga entre la variedad de ensambladores así como el entorno de desarrollo y tengan en cuenta el nivel de comprensión de esta clase de sistemas, pero lo suficientemente simple para ser entendido dentro de los límites de un curso en este campo de estudio.

El entorno de trabajo del SC123 consiste en un editor para facilitar la creación del código fuente, un ensamblador, desensamblador y emulador para poder generar código en lenguaje de máquina y verificar el resultado en las instrucciones, por ultimo un depurador para localizar errores de programación. Este sistema es empleado como material didáctico para comprender los principios de la arquitectura de computadoras. Cuenta con una documentación del entorno de trabajo, un manual de lenguaje ensamblador y el software está desarrollado en Java para su uso en múltiples plataformas.

3.4.2. Framework para FPGA

Al utilizar un FPGA para acelerar un programa, es necesario primero identificar un conjunto de operaciones que deben realizarse en hardware. Estos deben ser implementados como circuitos digitales, llamados unidades funcionales. Koltes en [21] propone un framework que está constituido por un circuito controlador genérico definido en VHDL que puede ser configurado por el usuario, de acuerdo a las necesidades de las unidades funcionales y el canal de entrada y salida, todo pensando para que el interesado construya una CPU bajo su criterio.

El objetivo propuesto es acelerar un programa que se ejecuta en uno o más procesadores, mediante el aumento de los procesadores con un conjunto de unidades funcionales. Para los cálculos altamente repetitivos, esto puede hacer al hardware significativamente más rápido que un programa correspondiente. Una unidad funcional es un circuito que realiza algún cálculo significativamente más rápido que se puede realizar en software. El núcleo de la interfaz es una máquina de transferencia de registro (RTM). Se trata de un microcontrolador con una arquitectura de estilo RISC, con base en archivos de registro y las instrucciones que actúan sobre los registros. Utilizando marcos de trabajo como este, pueden implementarse

procesadores que trabajen en paralelo, implementados en sistemas que usan FPGA.

3.4.3. Aplicación criptográfica en un microcontrolador MSP430X

En [22] se implementa una versión en alta velocidad de varios modos de encriptación autenticada (EA) en un microcontrolador MSP430X de 16 bits de Texas Instruments. Los autores programaron los seis modos CCM, GCM, SGCM, OCB3, Hummingbird-2 y MASHA. La encriptación autenticada (EA) es un esquema de criptografía simétrica que provee al mismo tiempo los servicios de confidencialidad y autenticación. Algunos esquemas para realizar EA usan un cifrador de bloque. Y uno de los cifradores de bloque estándar es el AES.

AES son las siglas de *Advanced Encryption Standard*. Este es un esquema de cifrado por bloques adoptado como estándar por el gobierno de los Estados Unidos de América. El tamaño del bloque es fijo a 128 bits y los tamaños de la llave pueden ser de 128, 192 o 256 bits. El microcontrolador MSP430X cuenta con el AES en hardware y los autores de [22] demuestran cómo se acelera el cómputo de los modos de EA que usan AES.

Los autores de [22] también ponen a disposición pública en [23] el software programado y le llamaron *biblioteca RELIC*, una biblioteca eficiente para criptografía. La biblioteca está en C y en el ensamblador del MSP430X.

De los seis modos de EA programados en [22], dos de ellos, CCM y GCM se implementaron en el ensamblador diseñado en este trabajo de tesis y se describirán muy brevemente aquí. Cada modo conta de dos algoritmos, uno para cifrar u otro para descifrar. Estos dos modos fueron estandarizados por el NIST (National Institute of Standardsand Technology) de los Estados Unidos de América. En el modo CCM (modo contador con CBC-MAC) para cada bloque del mensaje, un contador se encripta con el cifrador de bloque y al resultador se le aplica un xor con el mensaje para producir el mensaje cifrado; y entonces se incrementa el contador. Al mensaje también se le aplica un xor junto a un "acumulador" que también se encripta; este acumulador se vuelve la etiqueta (tag) de autenticación después de que se procesan todos los bloques. El modo GCM (modo contador de Galois) emplea la aritmética en el campo finito $\mathbb{F}_{2^{128}}$ para autenticación y el modo CTR para encriptar. Para cada bloque del mensaje, GCM encripta el contador y al resultado le aplica un xor con el mensaje para producir el texto cifrado; después se incrementa el contador. Al texto cifrado se le aplica un xor en un acumulador, y éste se multiplica en el campo finito con una constante dependiente de la llave. Este acumulador se usa para generar la etiqueta de autenticación.

Capítulo 4

Diseño del procesador

En este capítulo se considera la descripción completa en la que funciona todos los componentes que conforman a un microcontrolador y también se da una descripción de las microinstrucciones involucradas para cada una de las instrucciones.

El tipo de procesador que se ha seleccionado para el proyecto es el controlador incrustado o microcontrolador, se considera como un computador dedicado. Se caracteriza por tener almacenado dentro de la memoria un solo programa con el fin de gobernar un dispositivo. Las unidades internas que conforman al procesador se encargan de ejecutar un conjunto de instrucciones. Estos componentes se dividen en cuatro bloques:

* Proceso: unidad de control, unidad aritmética y lógica, archivo de registros, contador de programa, líneas de conexión y manejador de interrupciones.
* Memoria: almacenamiento de programa y datos.
* Periféricos: temporizadores, modulación de ancho de pulso, puerto paralelo y serie.
* Recursos auxiliares: temporizador perro guardián, reinicialización o reset.

Comúnmente todos los componentes mencionados se montan en un mismo circuito integrado CI de dimensiones reducidas. Este CI se encuentra montado en el dispositivo que controla, es por eso que es clasificado como microcontrolador incrustado.

No se realizan operaciones de punto flotante por lo que se descarta el uso de palabras de datos de 32, 64 o más bits. El uso de 8 bits maneja un máximo de 256 localidades de direccionamiento que en sí es muy poco y hace necesario el uso de registros que completen la dirección para direccionar más de 256 localidades. Descartando el tamaño de palabras anteriores se decide manejar en este diseño palabras de 16 bits para los datos y direcciones. Otros requerimientos para el diseño se enlistan a continuación:

- Realizar operaciones con representación de números enteros.
- Manejo de periféricos por medio del acceso a memoria.
- Atención de periféricos por interrupciones con vector de interrupciones programable.
- Memoria separada de datos y programa.
- Memoria de datos organizada por páginas.
- Manejo de subrutinas con una pila de contadores de programa.

- Manejo de subrutinas con una pila en memoria de datos.
- Palabras de 16 bits para datos y direcciones.
- Comunicación por medio de puerto serial y paralelo.
- Temporizadores de 16 bits.
- Manejador de interrupciones con enmascaramiento y por prioridad.

4.1. Arquitectura del conjunto de instrucciones

En esta sección se define el diseño del microcontrolador por medio de la arquitectura del conjunto de instrucciones (ISA pos sus siglas en inglés). Al definir el conjunto de instrucciones se tiene que considerar el propósito o alcance que se pretende alcanzar de forma anticipada [24]. La tabla 4.1 establece la notación y el uso de abreviaciones para referirse a los componentes usados y el tipo de comportamiento que estos realizan.

La primer instrucción considerada fue por direccionamiento inmediato donde un registro se inicializa con una constante, el formato utilizado influyo en el formato de todas las demás instrucciones del lenguaje máquina, está representada por el código de operación *opcode*, el registro de trabajo r_t, una valor inmediato *cte* y el modo de direccionamiento *modo*.

Las dimensiones de cada conjunto de bits es definida como: *opcode* de 6 bits (64 tipos de instrucciones), selector de registro r_t de 4 bits (16 registros de propósito general), *cte* de 16 bits (para datos o direcciones en un rango de 0 a 65535) y por último 3 bits para el *modo*. El conjunto definido por el r_t y el *cte* suman 20 bits que son distintos para cada formato de instrucciones, es la lista de $argumentos$. El número de bits utilizados para expresar una instrucción es de 29 bits. Ej.:

$$load\ r_t,\ cte$$
$$\Downarrow$$
$$[oo\ oooo]\ [rrrr]\ [cccc\ cccc\ cccc\ cccc]\ [mmm]$$
$$\Downarrow$$
$$[oc]\ [r_t]\ [cte]\ [md]$$
$$[28..23]\ [22..19][18..3]\ [2..0]$$
$$\Downarrow$$
$$[opcode][argumentos][modo]$$
$$[28..23]\ [22..3]\ [2..0]$$

La notación de pre-incremento, post-incremento, pre-decremento y post-decremento, aumentan o reducen en uno el contenido de algún registro antes o después de utilizarse. Los seleccionadores de registros r_x, r_t, r_f y r_a, se encargan de seleccionar algún registro utilizado en la instrucción y se obtiene del bus de instrucciones. La memoria de datos se abrevia como MD a diferencia del modo de direccionamiento que es *md* o *modo*.

El archivo de registros cuenta con 16 registros para referirse a un registro especifico se puede emplear como un arreglo, utilizando los corchetes para encerrar el índice, el primer registro se puede expresar como AR[0] (r_0) y el último registro se puede representar como AR[15] (r_{15}). El contador de programa y la memoria de datos también pueden utilizar el mismo comportamiento de un arreglo. Si se quiere acceder a un bit se utiliza como arreglo de arreglos, por ej.: acceder al bit 5 del registro 3 seria AR[3][5].

Siglas	Descripción
CP	Contador de programa
AR	Archivo de registros
MP	Memoria de programa
MD	Memoria de datos
BD	Bus de datos
BI	Bus de instrucción
BI.RX, r_x	Registro de evaluación (8 bits del BI)
BI.RT, r_t	Registro de trabajo (4 bits del BI)
BI.RF, r_f	Registro fuente (4 bits del BI)
BI.RA, r_a	Registro auxiliar (4 bits del BI)
BI.CTE, *cte*	Constante de dato (16 bits del BI)
BI.CTE, *dir*	Constante de dirección (16 bits del BI)
BI.N, n	Constante asociada a los bits de un registro (5 bits del BI)
i_{CP}	Puntero de pila de los registros del CP
X←Y	Asignación X← Y
X+=Y	Suma X← X+ Y
X-=Y	Resta X← X+ Y
X++	Post-incremento de X
++X	Pre-incremento de X
X--	Post-decremento de X
--X	Pre-decremento de X
(+|-)	Elegir suma o resta
clr, reset	Reinicio del CP, AR y MD
EOI	Fin de la interrupción (*End Of Interrupt*)
opcode, oc	Código de operación
modo, *md*	Modo de direccionamiento
status	Registro de banderas (Z,C,OV,S,...)

Tabla 4.1: Lista de notación usada para definir el conjunto de instrucciones

Salto

Mnemónico	Formato	Direccionamiento
$jclr\ r_x,\ n$	$[jclr][r_x][su][n][i]$	inmediato
$jset\ r_x,\ n$	$[jset][r_x][su][n][i]$	inmediato
$jmp\ r_t$	$[jmp][r_t][su][r]$	registro
$jmp\ dir$	$[jmp][su][dir][d]$	directo
jmp	$[jmp][su][m]$	implícito

Tabla 4.2: Instrucción tipo salto

Las instrucciones de salto se ven en la tabla 4.2, a continuación se describe su comportamiento. Salto condicional si el bit de un registro es cero o uno (JCLR y JSET):

- Las condiciones para los saltos condicionales son:
  AR[BI.RX][BI.N]==0 o status[BI.N]==0 para $jclr\ r_x,\ n$.
  AR[BI.RX][BI.N]==1 o status[BI.N]==1 para $jset\ r_x,\ n$.

- Si la condición es cierta se incrementar en dos el contador de programa:
  $jclr\ r_x,\ n \Leftrightarrow$ (AR[BI.RX][BI.N]==0 y 0 ≤BI.RX≤ 15) → CP[i_{CP}]+=2
  $jset\ r_x,\ n \Leftrightarrow$ (AR[BI.RX][BI.N]==1 y 0 ≤BI.RX≤ 15) → CP[i_{CP}]+=2
  $jclr\ r_x,\ n \Leftrightarrow$ (status[BI.N]==0 y BI.RX=255)→ CP[i_{CP}]+=2
  $jset\ r_x,\ n \Leftrightarrow$(status[BI.N]==1 y BI.RX=255)→ CP[i_{CP}]+=2
- Si la condición es falsa se incrementar en uno el contador de programa:
  $jclr\ r_x,\ n \Leftrightarrow$ (AR[BI.RX][BI.N]==1 y 0 ≤BI.RX≤ 15) → CP[i_{CP}]+=1
  $jset\ r_x, n \Leftrightarrow$ (AR[BI.RX][BI.N]==0 y 0 ≤BI.RX≤ 15) → CP[i_{CP}]+=1
  $jclr\ r_x,\ n \Leftrightarrow$ (status[BI.N]==1 y BI.RX=255) → CP[i_{CP}]+=1
  $jset\ r_x,\ n \Leftrightarrow$ (status[BI.N]==0 y BI.RX=255) → CP[i_{CP}]+=1

Salto incondicional (JMP):

- Saltar a la dirección de programa de forma directa: $jmp\ dir \Leftrightarrow$ CP[i_{CP}]← dir
- Saltar a la dirección de programa por registro: $jmp\ rt \Leftrightarrow$ CP[i_{CP}]← AR[BI.RT]
- Saltar de forma implícita: $jmp \Leftrightarrow$ CP[i_{CP}]+=2

Operaciones con la ALU

Mnemónico	Formato	Direccionamiento	Instrucciones ALU
$opcode_{alu_{binario}}\ r_t,\ r_f,\ r_a$	$[oc][r_t][r_f][r_a][su][r]$	registro	Binarias
$opcode_{alu_{binario}}\ r_t,\ r_f,\ n$	$[oc][r_t][r_f][su][n][i]$	inmediato	Desplazamiento
$opcode_{alu_{binario}}\ r_t,\ cte$	$[oc][r_t][cte][i]$	inmediato	Lógicas y aritméticas
$opcode_{alu_{unario}}\ r_t,\ r_f$	$[oc][r_t][r_f][su][r]$	registro	Unarias
$opcode_{alu_{unario}}\ r_t,\ cte$	$[oc][r_t][cte][i]$	inmediato	Unarias

Tabla 4.3: Instrucción tipo ALU, con dos operandos (binarias): lógicas (and, nand, or, nor, xor y xnor), aritméticas (add y sub), desplazamiento (srl, sra, sll, sla, ror, rol, rorc y rolc); e instrucciones con un operando (unarias): not e inv.

Se almacena el resultado en el registro r_t ($r_t \leftarrow resultado$). En la tabla 4.3 se muestran las instrucciones que utilizan un solo operando inv y not ($OP\ r_f$ o $OP\ cte$) representadas por $opcode_{alu_{unario}}$, también con dos operandos ($r_t\ OP\ cte$, $r_f\ OP\ n$ o $r_f\ OP\ r_a$) representadas por $opcode_{alu_{binaria}}$.

Instrucciones lógicas y aritméticas (AND, OR, XOR, NAND, NOR, XNOR, ADD y SUB):

- Seleccionar un operando de forma inmediata:
  $opcode_{alu_{binario}}\ r_t,\ cte \Leftrightarrow$AR[BI.RT]←AR[BI.RT] OP BI.CTE
- Seleccionar todos los operandos por registro:
  $opcode_{alu_{binario}}\ r_t,\ r_f,\ r_a \Leftrightarrow$AR[BI.RT]←AR[BI.RF] OP AR[BI.RA]

Las instrucciones de desplazamiento lógico y circular (ROR, ROL, RORC, ROLC, SRL, SRA, SLL y SLA):

- Seleccionar el número de bits de desplazamiento n de forma inmediato:
  $opcode_{alu_{binario}}\ r_t,\ r_f,\ n$ ⇔AR[BI.RT]←AR[BI.RF] OP BI.N
- Seleccionar el número de bits de desplazamiento n por registro:
  $opcode_{alu_{binario}}\ r_t,\ r_f,\ r_a$ ⇔AR[BI.RT]←AR[BI.RF] OP AR[BI.RA][4..0]
- Las instrucciones RORC y ROLC, modifican el accarreo: $carry, r_t \leftarrow [carry,\ r_f]$ OP n

La instrucción de negación lógica y revertir el orden de bits de una palabra de 16 bits (NOT e INV), manejan un solo operando:

- Seleccionar un operando de forma inmediata:
  $opcode_{alu_{unario}}\ r_t,\ cte$ ⇔ AR[BI.RT]← OP BI.CTE
- Seleccionar todos los operandos por registro:
  $opcode_{alu_{unario}}\ r_t,\ r_f$ ⇔ AR[BI.RT]← OP AR[BI.RF]

Mnemónico	Formato	Direccionamiento
$cmp\ r_t,\ r_f$	$[cmp][r_t][r_f][su][r]$	registro
$cmp\ r_t,\ cte$	$[cmp][r_t][cte][i]$	inmediato

Tabla 4.4: Instrucción tipo comparación

La tabla 4.4 describe la instrucción de comparar dos enteros (CMP):

- Cuando el segundo valor es por registro: $cmp\ r_t,\ r_f$ ⇔ AR[BI.RT] - AR[BI.RF]
- Cuando el segundo valor es inmediato: $cmp\ r_t,\ cte$ ⇔ AR[BI.RT] - BI.CTE

Mnemónico	Formato	Direccionamiento
$clr\ r_x$	$[oc][r_x][su][r]$	registro
$set\ r_x$	$[oc][r_x][su][r]$	registro
$clrb\ r_x, n$	$[oc][r_x][su][n][i]$	inmediato
$setb\ r_x, n$	$[oc][r_x][su][n][i]$	inmediato

Tabla 4.5: Instrucción para modificar un bit o todo un registro.

Las instrucciones de la tabla 4.5 inicializan con cero o uno un bit o todos los bits de un registro utilizando la ALU (CLRB, SETB, CLR y SET):

* Reiniciar todo un registro poniendo en cero todos los bits:
  $clr\ r_t$ ⇔AR[BI.RT]←0x0000
  $clr\ status$ ⇔status←0x0000
* Llenar todo un registro poniendo en uno todos los bits:
  $set\ r_t$ ⇔AR[BI.RT]←0xFFFF
  $set\ status$ ⇔status←0xFFFF
* Poner en cero un bit:
  $clrb\ r_t,\ n$ ⇔AR[BI.RT][BI.N]←0
  $clrb\ status,\ n$ ⇔status[BI.N]←0

* Poner en uno un bit:
  setb $r_t,\ n$ ⇔AR[BI.RT][BI.N]←1
  setb status, n ⇔status[BI.N]←1

Todas las instrucciones tipo ALU incrementan el contador de programa en uno: CP[i_{CP}]+=1

Subrutinas

Mnemónico	Formato	Direccionamiento
call dir	[*call*][*su*][*dir*][*d*]	directo
calla r_t, *dir*	[*calla*][r_t][*dir*][*d*]	directo
call r_t	[*call*][r_t][*su*][*ri*]	registro indirecto
calla r_t, r_f	[*calla*][r_t][r_f][*su*][*ri*]	registro indirecto
ret	[*ret*][*su*][*m*]	implícito

Tabla 4.6: Instrucciones tipo subrutina

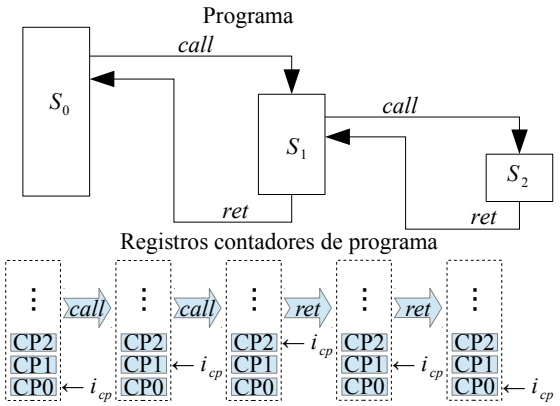


Figura 4.1: Llamada a subrutinas con *call* y *ret*

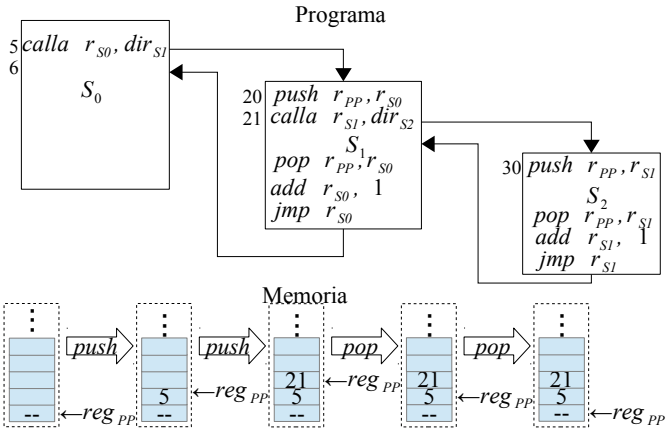


Figura 4.2: Llamada a subrutinas con *calla* y *jmp*

Las instrucciones necesarias para las subrutinas se muestran en la tabla 4.6, la figuras 4.1 y 4.2 ilustran el mecanismo al utilizar las instrucciones *call* con *ret* y *calla* con *jmp*. El registro de puntero de pila reg_{PP} es cualquier registro del AR. El comportamiento de estas instrucciones es:

* Llamar a una subrutina, utilizando la pila del contador de programa (CALL)
  - Obtener la dirección de programa de forma directa: *call dir*⇔ CP[++i_{CP}]←BI.CTE
  - Obtener la dirección de programa por registro: *call rt* ⇔ CP[++i_{CP}]←AR[BI.RT]
* Retornar de una subrutina, utilizando la pila del contador de programa (RET)
  - Pre-decrementar el puntero de la pila del contador de programa y post-incrementar el contador de programa en uno: *ret* ⇔ CP[--i_{CP}]++
* Llamar a una subrutina, para salvar el CP en registro del AR (CALLA)
  - Almacena el contador de programa en un registro: AR[BI.RT] ← CP.
  - Actualizar el contador de programa con un valor directo:
    calla r_t, *dir* ⇔ AR[BI.RT]←CP, CP←BI.CTE
  - Actualizar el contador de programa con un registro:
    calla r_t, r_f ⇔ AR[BI.RT]←CP, CP←AR[BI.RF]

La instrucción de llamada a subrutina con *calla* consiste en salvar el CP en un registro para ser posteriormente almacenado en memoria de datos con una instrucción *push*, para retornar se recupera el CP que está almacenado en memoria de datos con una instrucción *pop*, se incrementa en uno el valor del CP recuperado y se utiliza un salto incondicional *jmp* r_t.

Almacenar en archivo de registros

Mnemónico	Formato	Direccionamiento
load r_t, [*dir*]	[*load*][r_t][*dir*][*d*]	directo
load r_t, *cte*	[*load*][*rt*][*cte*][*i*]	inmediato
load r_t, r_f	[*load*][r_t][r_f][*su*][*r*]	registro
load r_t, [r_f]	[*load*][r_t][r_f][*su*][*ri*]	registro indirecto
load r_t, [$r_f(+\|-)r_a$]	[*load*][r_t][r_f][r_a][*su*][*bi*]	base índice

Tabla 4.7: Instrucción tipo carga *load*

La instrucción encargada de carga una palabra de 16 bits en un registro (LOAD) se muestra en la tabla 4.7, a continuación se describe el comportamiento de esta instrucción:

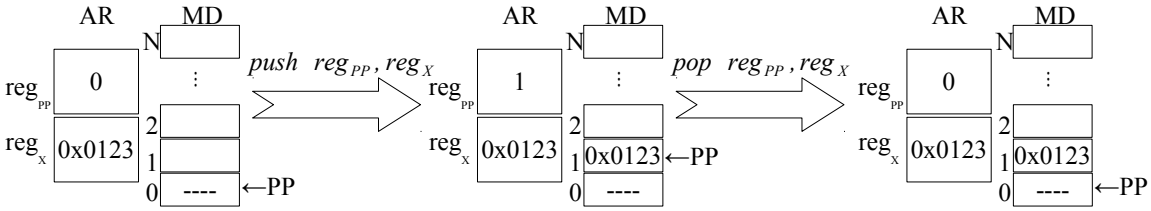
- Se obtiene el destino seleccionando por el bus de instrucción: AR[BI.RT]← Valor
- Obtener la palabra de forma inmediata: *load rt*, *cte* ⇔ AR[BI.RT]← BI.CTE
- Obtener la palabra de forma directa: *load rt*, [*dir*] ⇔AR[BI.RT]← MD[BI.CTE]

- Obtener la palabra por registro: $load\ rt,\ rf \Leftrightarrow$ AR[BI.RT]$\leftarrow$ AR[BI.RF]
- Obtener la palabra por base indexada:
  $load\ rt,\ [rf(+|\text{-})ra] \Leftrightarrow$AR[BI.RT]$\leftarrow$AR[BI.RF(+|-)BI.RA]
- Se incrementan en uno el contador de programa: CP[i_{CP}]+=1.

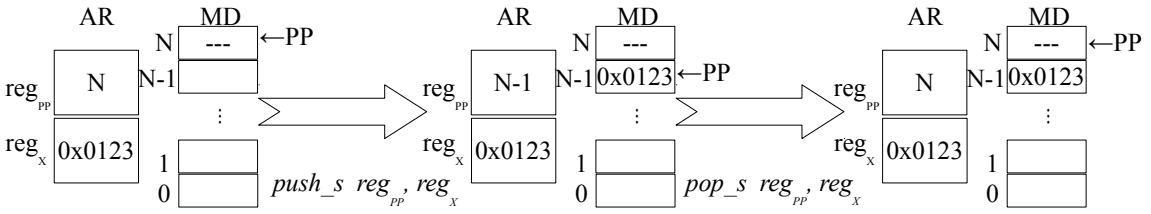
Pila en memoria de datos

Mnemónico	Formato	Direccionamiento
$opcode_{push}\ r_t,\ cte$	$[oc][rt][cte][i]$	inmediato
$opcode_{push}\ r_t,\ r_f$	$[oc][r_t][r_f][su][r]$	registro
$opcode_{pop}\ r_t,\ r_f$	$[oc][r_t][r_f][su][in]$	indexado

Tabla 4.8: Instrucción tipo pila



(a) Utilizando un puntero al inicia de la memoria



(b) Utilizando un puntero en el tope de la memoria

Figura 4.3: Instrucciones de la pila en memoria de datos.

La figura 4.3 ilustra el funcionamiento al usar la pila en memoria de datos. Las instrucciones que manipulan una pila en la memoria de datos se muestran en la tabla 4.8, a continuación se describa el funcionamiento de estas instrucciones.

Poner datos en la pila (PUSH):

- Se realiza un pre-incremento en el puntero de pila: ++AR[BI.RT]
- Almacenar el contenido de un registro: $push\ rt,\ rf \Leftrightarrow$ MD[++AR[BI.RT]]$\leftarrow$AR[BI.RF]
- Almacenar una palabra de forma inmediata: $push\ rt,\ cte \Leftrightarrow$ MD[++AR[BI.RT]]$\leftarrow$BI.CTE

Quitar datos de la pila (POP):

- Poner el tope de la pila en un registro: $pop\ rt,\ rf \Leftrightarrow$ AR[BI.RF] $\leftarrow$ MD[AR[BI.RT]--]
- Se realiza un post-decremento del puntero de pila: AR[BI.RT]--

Poner datos en la pila (PUSH_S):

- Pre-decrementa el puntero de pila: --AR[BI.RT]
- Almacenar el contenido de un registro en memoria:
  $push\_s\ rt,\ rf \Leftrightarrow$ MD[–AR[BI.RT]]$\leftarrow$AR[BI.RF]
- Almacenar una palabra de forma inmediata: $push\_s\ rt,\ cte \Leftrightarrow$ MD[–AR[BI.RT]]$\leftarrow$BI.CTE

Quitar datos de la pila (POP_S):

- Poner el tope de la pila en un registro: $pop\_s\ rt,\ rf \Leftrightarrow$ AR[BI.RF] $\leftarrow$ MD[AR[BI.RT]++]
- Post-incrementa el puntero de pila: AR[BI.RT]+ +

Todas las instrucciones que manipulan la pila tienen que incrementar en uno el contador de programa: CP[i_{CP}]+=1.

Almacenar en memoria de datos

Mnemónico	Formato	Direccionamiento
$store\ [dir], r_t$	$[store][r_t][dir][d]$	directo
$store\ [r_t], cte$	$[store][r_t][cte][i]$	inmediato
$store[r_t], r_f$	$[store][r_t][r_f][su][r]$	registro indirecto
$store[r_t(+\|-)r_f], ra$	$[store][r_t][r_f][r_a][su][bi]$	base índice

Tabla 4.9: Instrucción tipo almacenamiento

La instrucciones principal que almacenan una palabra de 16 bits en memoria de datos (STORE) se muestra en la tabla 4.9, su comportamiento se describen a continuación:

- Obtener la dirección de memoria de forma directa:
  $store\ [dir],\ rt \Leftrightarrow$MD[BI.CTE]$\leftarrow$AR[BI.RT]
- Obtener la dirección de memoria por registro indirecto:
  $store\ [rt],\ rf \Leftrightarrow$ MD[AR[BI.RT]] $\leftarrow$ AR[BI.RF]
- Obtener la dirección de memoria por registro base indexado:
  $store\ [rt(+|\text{-})rf],\ ra \Leftrightarrow$ MD[AR[BI.RT] (+|-) AR[BI.RF]] $\leftarrow$ MD[AR.RA]
- Inicializar una localidad de memoria de forma inmediata:
  $store\ [rt],\ cte \Leftrightarrow$ MD[AR[BI.RT]]$\leftarrow$ BI.CTE
- Se incrementa en uno el contador de programa: CP[i_{CP}]+=1.

Otras operaciones

Mnemónico	Formato	Direccionamiento
nop	[*nop*][*su*][*m*]	implícito
reti	[*reti*][*su*][*m*]	implícito
halt	[*halt*][*su*][*m*]	implícito
clrwdg	[*clrwd*][*su*][*m*]	implícito

Tabla 4.10: Instrucciones auxiliares

Las operaciones implícitas de la tabla 4.10 son descritas a continuación

- NOP: la no operación soló mata tiempo: $nop \Leftrightarrow \text{CP}[i_{CP}]+=1$

- RETI: el retorno de interrupción avisa del fin de la interrupción: $reti \Leftrightarrow \text{EOI} \leftarrow 1$

- HALT: detiene el flujo del programa: $halt \Leftrightarrow \text{CP} \leftarrow \text{CP}$

- CLRWDG: reinicia el temporizador guardián: $clrwdg \Leftrightarrow \text{timer}_{WDG} \leftarrow 0$ y $\text{CP}[i_{CP}]+=1$

El formato de todas las instrucciones se resume en la tabla 4.11, los bits sin uso se representan con [*su*]. Los bits del bus de instrucción que selecciona un registro para evaluar r_x consta de 8 bits donde las primeras 16 combinaciones (de 0x00 a 0x0F) corresponden a un registro del AR, y la combinación $0xFF$ se refiere al registro de banderas, dejando a las demás combinaciones sin uso. En la tabla 4.1 se describe muchas de las abreviaciones utilizadas.

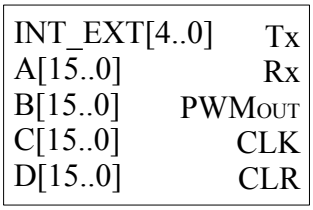
Formato de la instrucción	Bloques de Bits
[co][rt][rf][su][n][md]	[28..23][22..19][18..15][14..8][7..3][2..0]
[co][rt][rf][ra][su][md]	[28..23][22..19][18..15][14..11][10..3][2..0]
[co][rt][rf][su][md]	[28..23][22..19][18..15][14..3][2..0]
[co][rt][cte][md]	[28..23][22..19][18..3][2..0]
[co][rt][dir][md]	[28..23][22..19][18..3][2..0]
[co][su][md]	[28..23][22..3][2..0]
[co][rt][su][md]	[28..23][22..19][18..3][2..0]
[co][rt][label][md]	[28..23][22..19][18..3][2..0]
[co][rx][su][md]	[28..23][22..15][18..3][2..0]
[co][rx][su][n][md]	[28..23][22..15][14..8][7..3][2..0]

Tabla 4.11: Formato de las instrucciones en lenguaje de máquina

4.1.1. Microcontrolador

El procesador diseñado servirá en aplicaciones donde se utiliza un controlador incrustado, no se necesita recursos complejos, basta con una CPU que sea capaz de configurar y controlar un conjunto limitado de periféricos. No es suficiente controlar los periféricos solamente por

software, se pierde tiempo al conocer el estado de todos las unidades de E/S, los manejadores de interrupciones reducen drásticamente el tiempo desperdiciado en esta tarea. La figura 4.4 muestra la interfaz de entrada y salida del microcontrolador diseñado, se enlista el nombre de cada señal con su descripción correspondiente.



Señal	E/S	Bits	Descripción
Tx	S	1	Transmisión serial de datos
Rx	E	1	Recepción serial de datos
PWM_{out}	S	1	Modulación de ancho de pulso
CLR	E	1	Señal de reinicio externo
CLK	E	1	Señal de reloj
A	E-S	16	Puerto paralelo A
B	E-S	16	Puerto paralelo B
C	E-S	16	Puerto paralelo C
D	E-S	16	Puerto paralelo D
INT_{EXT}	E	5	Aviso de interrupción externa

Figura 4.4: Símbolo del procesador

El ciclo de instrucción del microcontrolador se ejecutara en un ciclo de reloj, se describe como: en el tiempo x_i se realizara las etapas de búsqueda, decodificación y ejecución, es decir, todos los componentes que funcionan de forma combinacional incluidas la lectura del AR, FLAGS, RFE, CP, MP y de la MD; las etapas que actualizan la memoria de datos y los registros se realizan en la pendiente positiva o franco de subida y_i, es decir, los componentes que funcionan de forma secuenciales incluida la escritura del AR, FLAGS, RFE, CP y la MD. La figura 4.5 muestra la señal de reloj, el contador de programa, resaltando la ejecución de distintas instrucciones i_i.

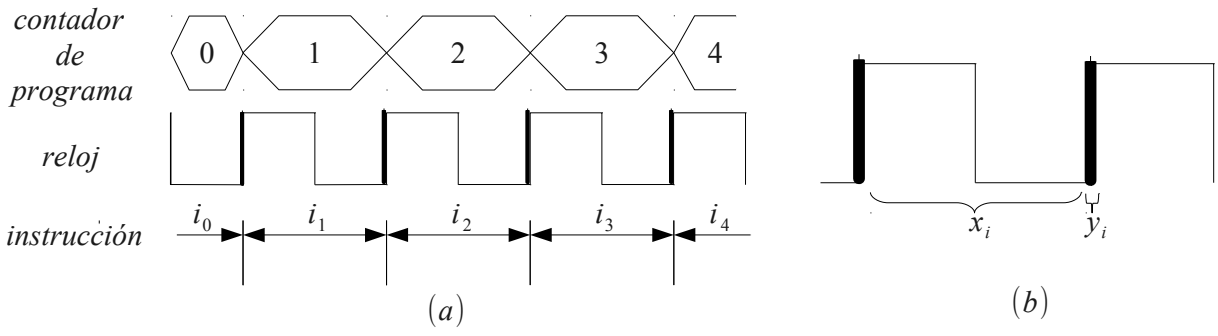


Figura 4.5: Ciclo de instrucción

La figura 4.6 muestra la conexión general de todos los componentes que conforman al procesador. A lo largo de todo el capítulo se describe el funcionamiento de estos elementos.

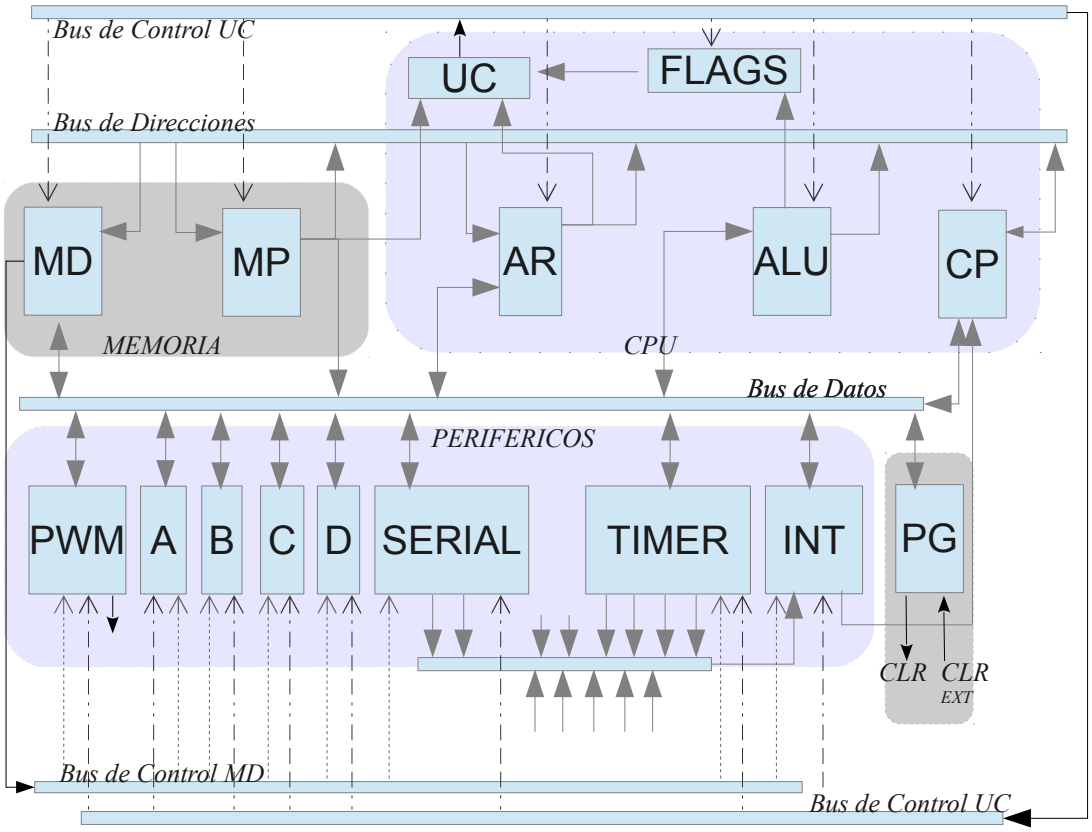


Figura 4.6: Diagrama a bloques del procesador

4.2. Proceso

Los componentes que caen en esta clasificación traducen, decodifican, almacenan y ejecutan las operaciones principales, son el núcleo del procesador. El comportamiento de estos componentes es definido por el conjunto de instrucciones.

4.2.1. Unidad aritmética lógica

La unidad aritmética y lógica se encarga de realizar una operación a la vez, estas operaciones son lógicas, aritméticas, desplazamiento lógico o desplazamiento circular. Las operaciones se describen en las tablas 4.12 a la 4.18. Los componentes que conforman a la ALU se ilustran en la figura 4.7 donde un multiplexor selecciona el resultado que la ALU devolverá al bus de datos o direcciones [25].

Símbolos	Descripción	Operación	Ejemplo
+	Suma	$a_i + b_i$	$1010 + 0101 = 1111, carry = 0$
-	Resta	$a_i - b_i$	$1010–0101 = 1010 + 1011 = 0101, carry = 1$

Tabla 4.12: Operaciones aritméticas

Símbolos	Operación	Ejemplo
$\ll$	sll	$0110 \ll 2 = 1000$
$\ll\sim$	sla	$0110 \ll\sim 2 = 1011$
$\gg$	srl	$0101 \gg 2 = 0001$
$\gg\sim$	sra	$0101 \gg\sim 2 = 1101$

Tabla 4.13: Desplazamiento lógico

$r \leftarrow a \ll n,\ r \leftarrow a \ll\sim n$	$r \leftarrow a \gg n, r \leftarrow\ a \gg\sim n$
$(i + n < N) \rightarrow (r_i \leftarrow a_{i+n})$	$(i - n \geq N) \rightarrow (r_i \leftarrow a_{i-n})$
$(i + n \geq N) \rightarrow (r_i \leftarrow X)$	$(i - n < N) \rightarrow (r_i \leftarrow X)$

Tabla 4.14: Descripción del desplazamiento lógico, donde $X \in \{0, 1\}$, $a \leftarrow [a_0, a_1, \ldots, a_{N-1}]$, $r \leftarrow [r_0, r_1, \ldots, r_{N-1}]$, len(a)=N, $i \in \{0, \ldots,$ N-1$\}$ y $n \in \{0, \ldots, N\}$

$r \leftarrow a\ ror\ n$	$r \leftarrow a\ rol\ n$
$(i - n \geq 0) \rightarrow (r_i \leftarrow a_{i-n})$	$(i + n < N) \rightarrow (r_i \leftarrow a_{i+n})$
$(i - n < 0) \rightarrow (r_i \leftarrow a_{N+(i-n)})$	$(i + n \geq N) \rightarrow (r_i \leftarrow a_{(i+n)-N})$

Tabla 4.15: Descripción del desplazamiento circular, donde $a \leftarrow [a_0, a_1, \ldots, a_{N-1}]$, $r \leftarrow [r_0, r_1, \ldots, r_{N-1}]$, len(a)=N, $i \in \{0, \ldots,$ N-1$\}$ y $n \in \{0, \ldots, N\}$

Operación	Ejemplo
ror	$1100\ ror\ 3 = 1001$
rol	$1100\ rol\ 3 = 0110$
rorc	$[carry \leftarrow 0]1100 rorc 3 = [carry \leftarrow 1]0001$
rolc	$[carry \leftarrow 0]1100 rolc 3 = [carry \leftarrow 0]0011$

Tabla 4.16: Desplazamiento circular

Símbolos	Descripción	Operación	Ejemplo
$\wedge$	$a_i \wedge b_i$	and	$0011 \wedge 0110 = 0010$
$\neg, \wedge$	$\neg(a_i \wedge b_i)$	nand	$\neg(0011 \wedge 0110) = 1101$
$\vee$	$a_i \vee b_i$	or	$0011 \vee 0110 = 0111$
$\neg, \vee$	$\neg(a_i \vee b_i)$	or	$\neg(0011 \vee 0110) = 1000$
$\oplus$	$a_i \oplus b_i$	xor	$0011 \oplus 0110 = 0101$
$\neg, \oplus$	$\neg(a_i \oplus b_i)$	xnor	$\neg(0011 \oplus 0110) = 1010$
$\neg$	$\neg\ a_i$	not	$\neg\ 0011 = 1100$

Tabla 4.17: Operaciones lógicas

Operación	Descripción	Ejemplo
clr	$rx_i = 0$	$a \leftarrow$ 0x0000
set	$rx_i = 1$	$a \leftarrow$ 0xFFFF
clrb	$rx_n = 0$	$(a =$ 0xABCD y $n = 3) \rightarrow (a \leftarrow$ 0xABC5)
setb	$rx_n = 1$	$(a =$ 0xABCD y $n = 1) \rightarrow (a \leftarrow$ 0xABCF)

Tabla 4.18: Operación de *clr* o *set*

La ALU cuenta con diferentes señales los operandos A y B que son dos señales de 16 bits, N es el número de bits que se desplazan hacia la izquierda o a la derecha, siendo un entero de 5 bits cuando se toma en cuenta el acarreo y 4 bits en otro caso, OP selecciona el tipo de operación (con un máximo de 32 operaciones), CIN es el acarreo de entrada señal de un solo bit. La tabla 4.19 muestra la descripción de las banderas que son almacenadas en el registro de status o FLAGS, el sumador completo de la figura 4.10 muestra la conexión de estas banderas. Cada operación realizada modifica ciertas banderas.

Símbolo	Bandera	Descripción	Ecuación
Z	Cero	Indica si el resultado es cero	$\bigvee_{i=0}^{n} s_i$
C	Acarreo	Indica si el resultado tiene acarreo	C_{n+1}
S	Signo	Indica si el resultado es negativo	s_n
O	Desbordamiento	Descripción	$c_{n+1} \oplus c_n$

Tabla 4.19: Banderas de la ALU durante la suma

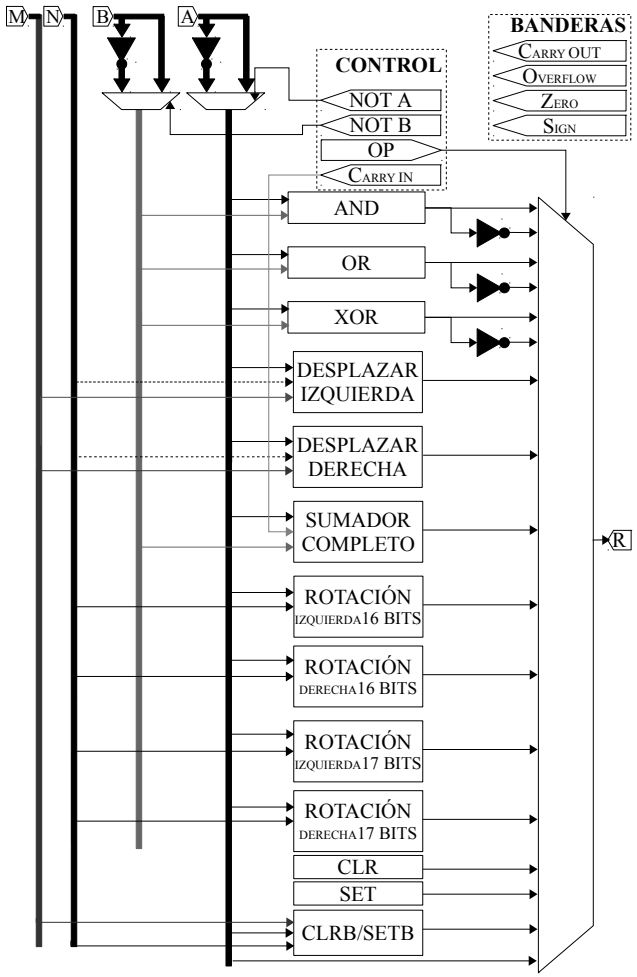


Figura 4.7: Diagrama a bloques de la ALU

Positiva					Negativo				
s	b3	b2	b1		s	b3	b2	b1	
0	0	0	0	+0	1	0	0	0	-0
0	0	0	1	+1	1	0	0	1	-1
0	0	1	0	+2	1	0	1	0	-2
0	0	1	1	+3	1	0	1	1	-3
0	1	0	0	+4	1	1	0	0	-4
0	1	0	1	+5	1	1	0	1	-5
0	1	1	0	+6	1	1	1	0	-6
0	1	1	1	+7	1	1	1	1	-7

(a)

Número positivo		Número negativo			
		Complemento a uno		Complemento a dos	
0000	0	1111	-0	1111	-1
0001	1	1110	-1	1110	-2
0010	2	1101	-2	1101	-3
0011	3	1100	-3	1100	-4
0100	4	1011	-4	1011	-5
0101	5	1010	-5	1010	-6
0110	6	1001	-6	1001	-7
0111	7	1000	-7	1000	-8

(b)

Figura 4.8: Representación de números enteros

La representación de enteros con signo más simple es interpretar al bit más significativo como positivo con 0 y negativo con 1, esto se ilustra en la figura 4.8(a). Otras forma de representar a los enteros negativos es usando el complemento a uno o a dos, se ilustra en la tabla 4.8(b).

A	B	Cin	S	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$S \leftarrow \overline{A}\,\overline{B}\,Cin + \overline{A}\,B\,\overline{Cin} + A\,\overline{B}\,\overline{Cin} + A\,B\,Cin$

$S \leftarrow A \oplus B \oplus Cin$

$Cout \leftarrow \overline{A}\,B\,Cin + A\,\overline{B}\,Cin + A\,B\,\overline{Cin} + A\,B\,C$

$Cout \leftarrow (AB) + ((A+B)\,Cin)$

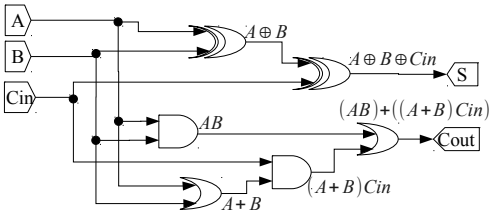


Figura 4.9: Tabla de verdad, ecuación booleana y diagrama de compuertas del sumador completo de un bit

En la figura 4.9 se observa la tabla de verdad, las ecuaciones booleanas y el diagrama de compuertas correspondiente a un sumador completo de un solo bit. Al conectar varios sumadores completos de un bit en cascada se obtiene un módulo para la ALU que realiza la suma, la configuración se observa en las figura 4.10.

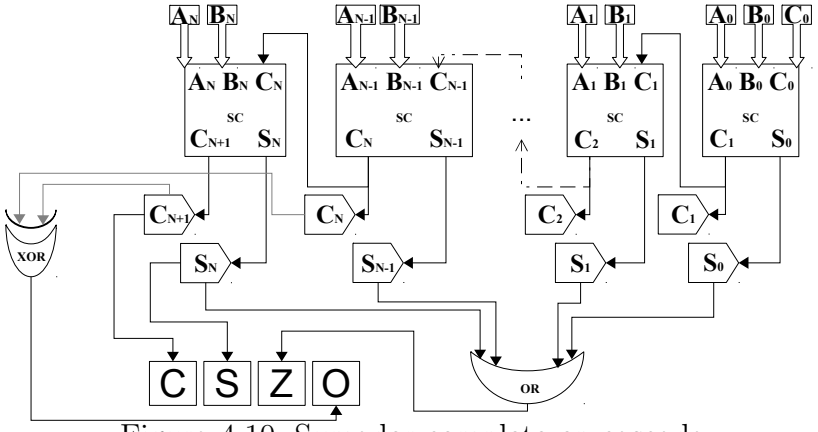
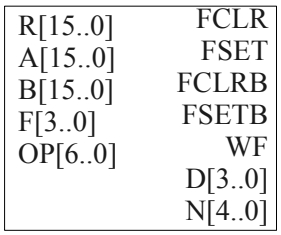


Figura 4.10: Sumador completo en cascada

Para realizar una resta $A - B$ se convierte B a complemento a dos B'' y con el sumador completo en cascada se suma $A + B''$, si el bit más significativo es 1 el resultado está en complemento a dos. La ALU está configurada para realizar esta operación $A - B = A + (\neg B + 1)$. El complemento a uno de A se obtiene $A' = \neg A$ y para recuperarlo el valor $A = \neg A'$. El complemento a dos de A se obtiene sumando el complemento uno de A más uno $A'' = A' + 1$ para recuperar A se realiza el complemento a dos de A'' [26].

El rango de enteros representados con complemento a uno es $2^{n-1} - 1$ para positivos, $2^{n-1} - 1$ para negativos y dos representaciones para el cero. Con el complemento a dos el rango de enteros representados es $2^{n-1} - 1$ para positivos, para negativos 2^{n-1} y una sola forma de representar al cero.

La interfaz de la unidad aritmética y lógica se distingue en figura 4.11, este componente integra al registro de banderas.

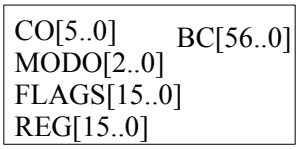


Señal	E/S	Bits	Descripción
R	S	16	Resultado
F	S	4	Banderas
A	E	16	Operando A
B	E	16	Operando B
N	E	5	Operando N
D	E	4	Bus de datos, actualiza la banderas.
OP	E	7	Operación A op B
FCLR	E	1	Inicializa las banderas en cero
FSET	E	1	Inicializa las banderas en uno
FCLRB	E	1	Inicializa la bandera[N] en cero
FSETB	E	1	Inicializa la bandera[N] en uno
WF	E	1	Habilita la escritura en las banderas

Figura 4.11: Símbolo de la ALU.

4.2.2. Unidad de control

Este componente recibe información del bus de instrucción, las entradas se evalúan con una serie de condiciones para activar el bus de control en el orden necesario para poder ejecutar todas las etapas del ciclo de instrucción, podría verse como un demultiplexor o una memoria ROM en ciertos casos cuando las instrucciones no tiene mucha complejidad. Evalúa las banderas del registro FLAGS o status en los saltos condicionales. Este componente traduce el lenguaje de máquina a microinstrucciones. Lee el bus de instrucción y algún registro para decidir que señales de control activar con 1 o desactivar con 0.



Señal	E/S	Bits	Descripción
CO	E	6	Código de operación
MODO	E	3	Modo de direccionamiento
FLAGS	E	16	Banderas del registro status
REG	E	16	Valor de un registro del AR
BC	S	57	Bus de control

Figura 4.12: Símbolo de la unidad de control

El bus de control incluye las señales de lectura R y escritura W, las operación que realiza la ALU OP_{ALU}, el control de los multiplexores que dirige la interconexión de los componentes, etc. La figura 4.12 muestra el símbolo que define a la unidad de control, el flujo de los datos se muestra en la figura 4.13.

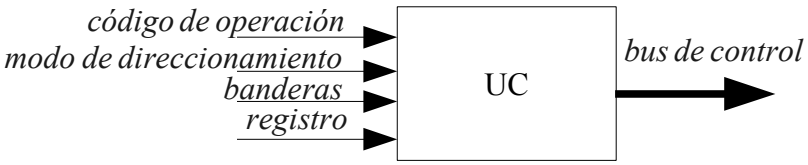


Figura 4.13: Unidad de control

4.2.3. Archivo de registros

El archivo de registros AR, es una colección de registros de propósito general, se leen tres registros de forma simultánea y se escribe en uno solamente, es decir, la lectura es combinacional y la escritura secuencial. La imagen 4.14 muestra los componentes que integran al AR. Se cuenta con un máximo de 2^4 registros direccionables con S_0, S_1, S_2 y W_0 estas señales cuentan con 4 bits, las primeras tres señales seleccionan los registros de lectura y la otra señal referencia al registro que se desea escribir. El símbolo de este componente se ilustra en la figura 4.15

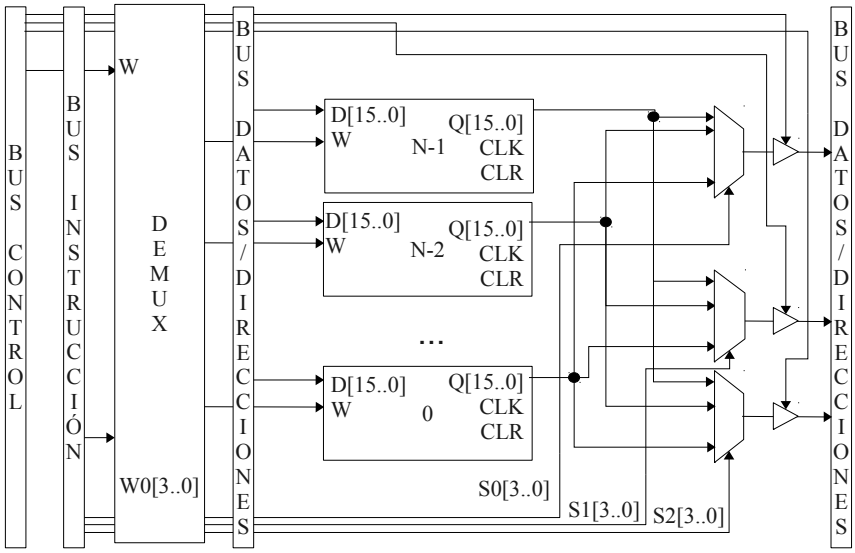
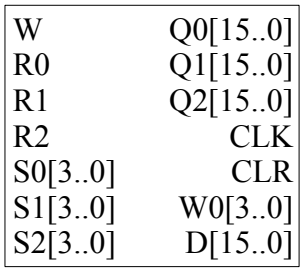


Figura 4.14: Diagrama a bloques del AR

Las señale de lectura R_0, R_1 y R_2 controlan cada una un buffer triestado. La señal selectora W_0 establece que registro actualizar, se activa la escritura con el bit W.

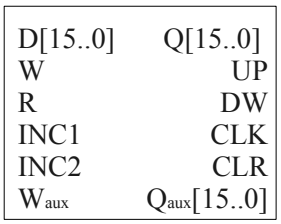


Señal	E/S	Bits	Descripción
D	E	16	Bus de datos
Q_0	S	16	Bus de datos 0
Q_1	S	16	Bus de datos 1
Q_2	S	16	Bus de datos 2
S_0	E	4	Selector de registro lectura 0
S_1	E	4	Selector de registro lectura 1
S_2	E	4	Selector de registro lectura 2
W_0	E	4	Selector de registro de escritura
W	E	1	Habilita la escritura de datos
R_0	E	1	Habilita la lectura de datos
R_1	E	1	Habilita la lectura de datos
R_2	E	1	Habilita la lectura de datos
CLR	E	1	Reinicio de los registros
CLK	E	1	Reloj

Figura 4.15: Símbolo del AR

4.2.4. Contador de programa

El contador de programa CP, es el registro encargado de referenciar la instrucción a ejecutar en el ciclo de instrucción. El valor almacenado se incrementa en uno, casi siempre. En instrucciones de salto, llamadas a subrutinas, retorno de subrutina, llamadas y retornos de subrutinas de interrupciones modifican el CP para seguir el flujo del programa. Se modifica este registro con una dirección que se puede obtener directamente del bus de instrucción, del archivo de registros o de la ALU, o se puede incrementando en uno o dos el CP.



Señal	E/S	Bits	Descripción
D	E	16	Bus de direcciones
Q	S	16	Bus de direcciones
Q_{aux}	S	16	Bus de direcciones
R	E	1	Habilita la lectura de datos
W	E	1	Habilita la escritura del CP
W_{aux}	E	1	Mayor prioridad en la escritura del CP
UP	E	1	Incrementa en uno el puntero del CP
DW	E	1	Decrementa en uno el puntero del CP
CLR	S	1	Reinicio de los registros
CLK	E	1	Reloj
INC1	E	1	Incrementa en uno el CP
INC2	E	1	Incrementa en dos el CP

Figura 4.16: Símbolo del CP

Para una llamada a subrutina se utiliza una pila interna de registros que almacenan direcciones en cada CP_i. Otra forma de resolver el uso de subrutinas es salvar el CP en un registro de propósito general y ponerlo en la pila de la memoria de datos, para retornar se recupera la dirección de la pila incrementando en uno y realizar una instrucción de salto incondicional por registro. En las interrupciones el manejador programable debe salvar el CP en una pila de registros y debe contar con un vector de interrupciones para obtener la

subrutina asociada con la señal de interrupción. La conexión interna del contador de programa se observa en la figura 4.17, la interfaz de las señales de este componente se muestra en la figura 4.16.

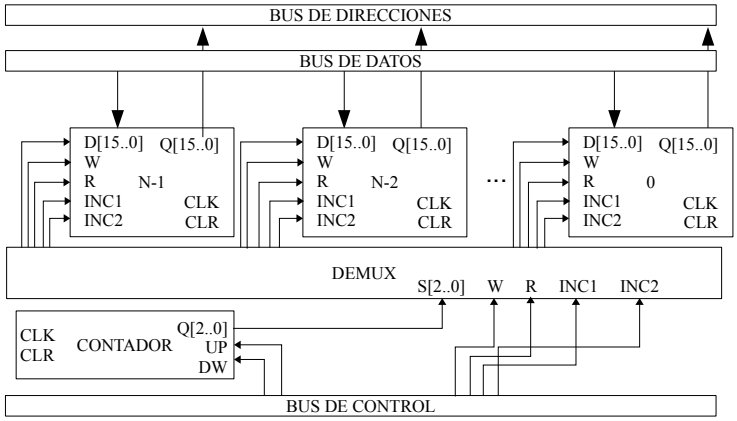


Figura 4.17: Diagrama a bloques del CP

La figura 4.17 muestra los componentes que integran al CP, el CONTADOR selecciona el registro a utilizar del CP, el demultiplexor selecciona que registro tendrá acceso a las señales de control. El número de registros utilizados es de 2^3 o de 2^4, cada registro es posible seleccionarlo por medio del contador que es un puntero. Al configurar al CP con el control de interrupciones se tiene otra señal de escritura W_{aux} que tiene mayor prioridad que las señales que recibe de la unidad de control, además se cuenta con otra señal de lectura Q_{aux} que manda la posición donde el programa se detiene por una interrupción, este valor se deposita en una pila en hardware que está incluido en el controlador de interrupciones. Cuando se termina una subrutina de interrupción se recupera el valor del CP que se modificó en la última instrucción sea el de una llamada a subrutina, salto condicional o incondicional, o el simple incremento en uno del CP.

4.2.5. Control de interrupciones

La figura 4.18 muestra un ejemplo en la que cuatro interrupciones requieren ser atendidas por el microcontrolador, las señales $interrupciones_i$ tiene asociado un valor que define el nivel de privilegio que está tiene. El identificador de prioridad se distingue en la señal $Prioridad$. El programa se ejecuta de forma normal, en el momento en que una interrupción esta en alto, se interrumpe el programa salvando el contenido del contador de programa. Cuando ya se esta atendiendo alguna interrupción y llega una nueva se revisa su prioridad, si es mayor se anida la interrupción actual salvando el contenido del contador de programa y cargando la nueva dirección de subrutina que ha entrado en escena, en caso contrario de que su prioridad sea menor o igual se ignora y espera hasta que se termine la ejecución de la que tiene mayor privilegio y cuya importancias es más significativa. Cuando se termine de atender a todas las interrupciones se reincorpora el valor original del contador de programa así como el estado

en el que se encontraba los registros de propósito general y cualquier otra información que se tenía antes de ser interrumpido, es decir, se restaura el contexto del programa.

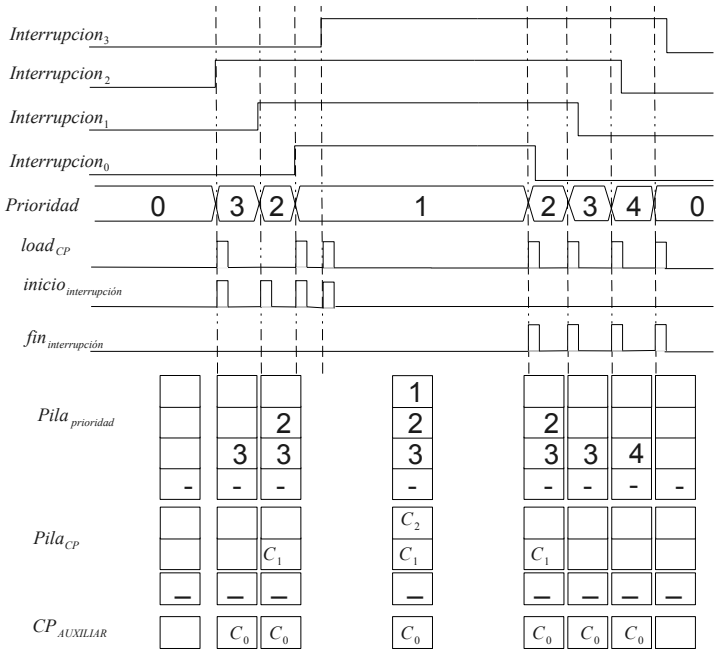


Figura 4.18: Ejemplo de atención a señales de interrupción

La señal $inicio_{\text{interrupción}}$ muestra el momento en que aparece una interrupción y es cuando el mecanismo de la figura 4.19 debe comportarse como se describió en el párrafo anterior, la señal $fin_{\text{interrupción}}$ (EOI) es generada por la instrucción *reti* que indica que la interrupción ya fue atendida y queda bajo el manejador de interrupciones el criterio para continuar, la señal $load_{CP}$ (WCP) indica la dirección que debe cargar en el contador de programa, siendo estas el valor original del mismo, las direcciones interrumpidas al ser anidadas o las mismas direcciones que fueron configuradas en el vector de interrupciones. Es obligación del programa configurar y cargar todos los registros del manejador de interrupciones, incluido las direcciones depositadas en el vector de interrupciones, el correcto enmascaramiento que es un filtro que especifique que interrupciones se deben manejar. Se debe garantizar que la señal de interrupción esta en bajo $INTR[i] = 0$ para no quedarse atascado en un ciclo infinito atendiendo las mismas interrupciones. La tabla 4.20 muestra el orden de prioridad en la que están asociadas las interrupciones con la señal INTR, la figura 4.20 describe la interfaz disponible para este componente.

Nombre	Pin	Nombre	Pin	Nombre	Pin	Nombre	Pin
PortRx	0	PortTx	1	Timer1	2	timer0	3
Timer2	4	Timer3	5	SU0	6	SU1	7
SU2	8	SU3	9	Ext1	10	Ext0	11
Ext2	12	Ext3	13	Ext4	14	*	*

Tabla 4.20: Interrupciones asociadas a la señal INTR[14..0]

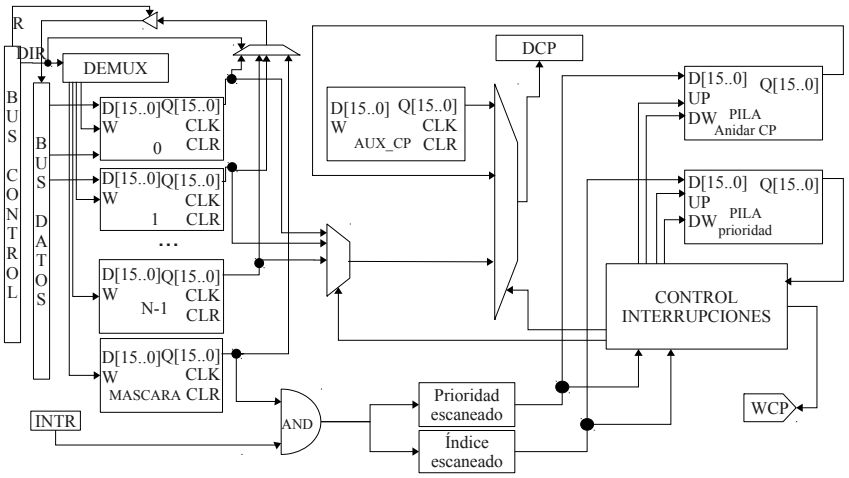
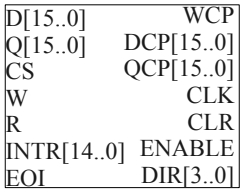


Figura 4.19: Diagrama a bloques del manejador de interrupciones



Señal	E/S	Bits	Descripción
D	E	16	Bus de datos
Q	S	16	Bus de datos
DCP	E	16	Bus de direcciones
QCP	S	16	Bus de direcciones
R	E	1	Habilita la lectura de datos
W	E	1	Habilita la escritura de datos
WCP	S	1	Habilita la escritura del CP
ENABLE	S	1	Habilita el control de interrupciones
CLR	E	1	Reinicio de los registros
CLK	E	1	Reloj
CS	E	1	Habilita lectura y escritura de datos
DIR	E	3	Selector de registro
INTR	E	15	Interrupción internas y externas
EOI	E	1	Fin de la interrupción

Figura 4.20: Símbolo del manejador de interrupciones

4.3. Periféricos

La mayoría de estos componentes son constituidos por puertos de comunicación que se encargan de transmitir o recibir datos desde o hacia otros dispositivos [27]. La figura 4.21 muestra el comportamiento del divisor de frecuencia ocupado frecuentemente.

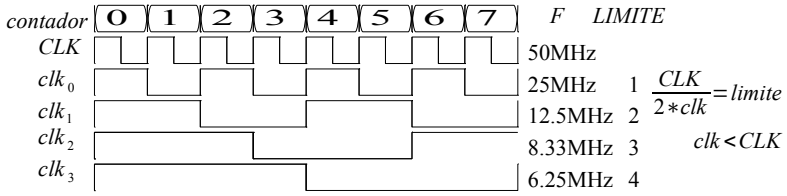


Figura 4.21: Divisor de frecuencia

4.3.1. Modulación por ancho de pulso

La modulación por ancho de pulso o PWM por sus siglas en inglés. Este componente recibe el contenido de un puerto y lo compara con una señal sierra, si es mayor el resultado es uno de lo contrario es cero, también se puede negar esta señal para requerimientos de cualquier aplicación. Este componente sirve para controlar un motor eléctrico, o para controlar la intensidad de luz de un LED.

En la figura 4.22 la señal (a) es generada internamente, es una onda de sierra, el programador establece el rango en el que trabaja de 0 a un límite establecido CONTLIM. La señal (b) es obtenida directamente por un puerto paralelo, configurado como pines entrada. La señal (c) es la onda cuadrada generada al comparar las señales (a) y (b). La configuración de los componentes que conforman al componente PWM se ven en la figura 4.23. La interfaz de este componente es ilustrada en la figura 4.24.

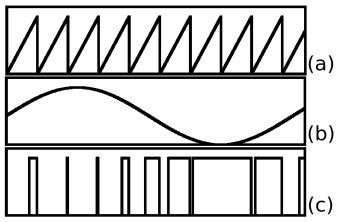


Figura 4.22: Señales involucradas en el PWM

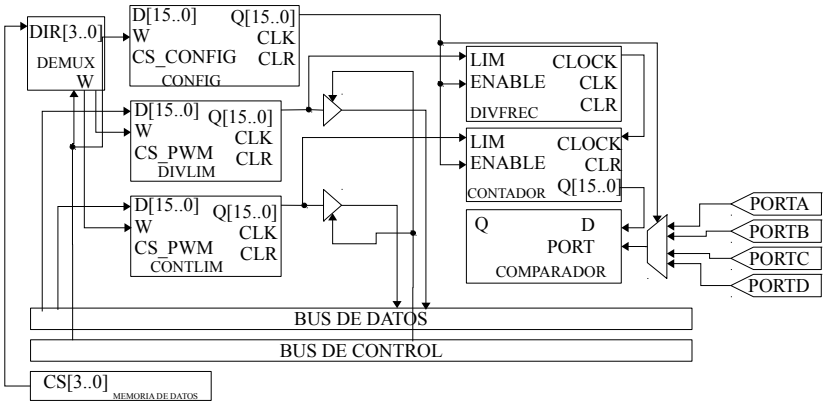
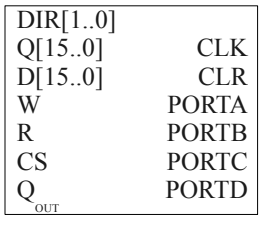


Figura 4.23: Diagrama a bloques del PWM



Señal	E/S	Bits	Descripción
DIR	E	2	Dirección de memoria
D	E	16	Bus de datos
Q	S	16	Bus de datos
CS	S	3	Selección de registros mapeados en la MD
R	E	1	Habilita la lectura de datos
W	E	1	Habilita la escritura de datos
CLR	E	1	Reinicio interno
CLK	E	1	Reloj
Q_{out}	S	1	Resultado del PWM
$PORT_{A,\ B,\ C\ o\ D}$	E	16	Puerto A, B, C o D

Figura 4.24: Símbolo del PWM

4.3.2. Puerto serie

El puerto serie consiste en un pin que transmite asíncronamente 8 bits de información, cada vez que se actualiza el registro de transmisión. Para recibir se habilita un pin de recepción que actualiza un registro, recibiendo 8 bits de datos. La velocidad de transmisión y recepción se establece al fijar un límite al contador de divisor de frecuencia cada vez que se repite el intervalo de cero al límite cambia de estado el reloj de la comunicación serial.

La comunicación serial transmite y recibe 1 byte, es deber del programador calcular el límite para el contador del divisor de frecuencia. De esta forma se establece cuantos bits por segundos se transmiten por medio del puerto serie. La forma en que este puerto funciona se ilustra en la figura 4.25. La configuración interna del transmisor y receptor se ilustra en la figura 4.27, la interfaz para activar estos componentes se muestra en la figura 4.26.

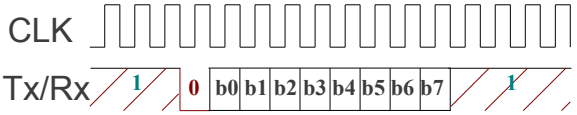
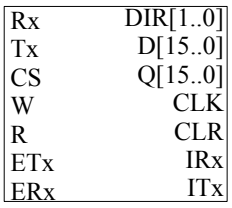
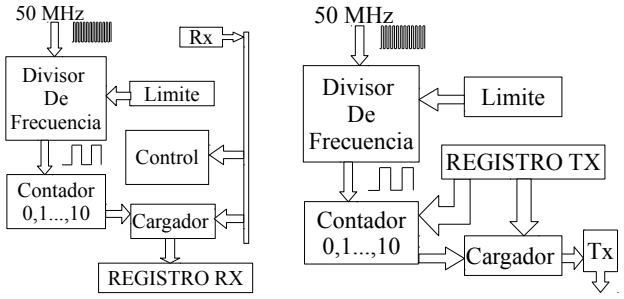


Figura 4.25: Comunicación con el puerto serie



Señal	E/S	Bits	Descripción
D	E	16	Bus de datos de entrada
Q	S	16	Bus de datos de salida
R	E	1	Habilita la lectura de datos
W	E	1	Habilita la escritura de datos
CLR	E	1	Reinicio de los registros
CLK	E	1	Reloj
CS	E	1	Habilita lectura y escritura de datos
Rx	E	1	Recepción de la comunicación serial
Tx	E	1	Transmisión de la comunicación serial
DIR	E	2	Selector de registro
ERx	E	1	Habilita la transmisión
ETx	E	1	Habilita la recepción
IRx	S	1	Aviso de interrupción de la recepción
ITx	S	1	Aviso de interrupción de la transmisión

Figura 4.26: Símbolo del puerto serie



(a) (b)

Figura 4.27: Diagrama a bloques del transmisor y receptor serial

4.3.3. Puerto paralelo

El puerto paralelo, es el encargado de comunicarse con el mundo exterior a través de 16 pines, estos pueden ser configurados de lectura o escritura, por defecto son de lectura para evitar averías eléctricas, el programa configura uno o varios pines de salida, es decisión del programador establecer las entradas y salidas. La configuración de los componentes que integran al puerto paralelo se ven en la figura 4.28, el símbolo que define al componente se ve en la figura 4.29.

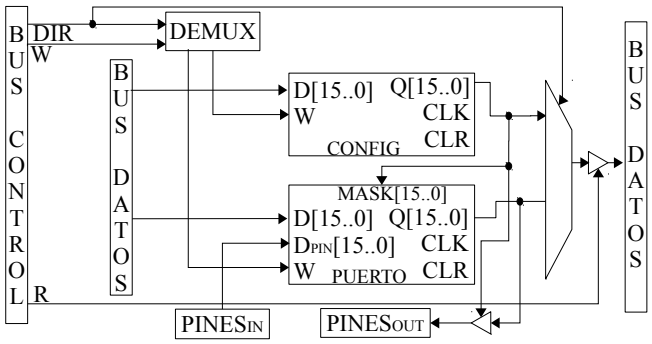
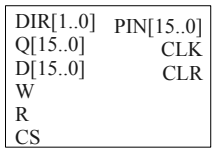


Figura 4.28: Diagrama a bloques del puerto paralelo

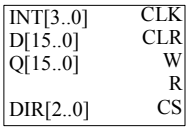


Señal	E/S	Bits	Descripción
D	E	16	Bus de datos de entrada
Q	S	16	Bus de datos de salida
R	E	1	Habilita la lectura de datos
W	E	1	Habilita la escritura de datos
PIN	E/S	1	Conexión con los pines del puerto
CLR	E	1	Reinicio de los registros
CLK	E	1	Reloj
CS	E	1	Habilita lectura y escritura de datos
DIR	E	2	Selector de registro

Figura 4.29: Símbolo del puerto paralelo

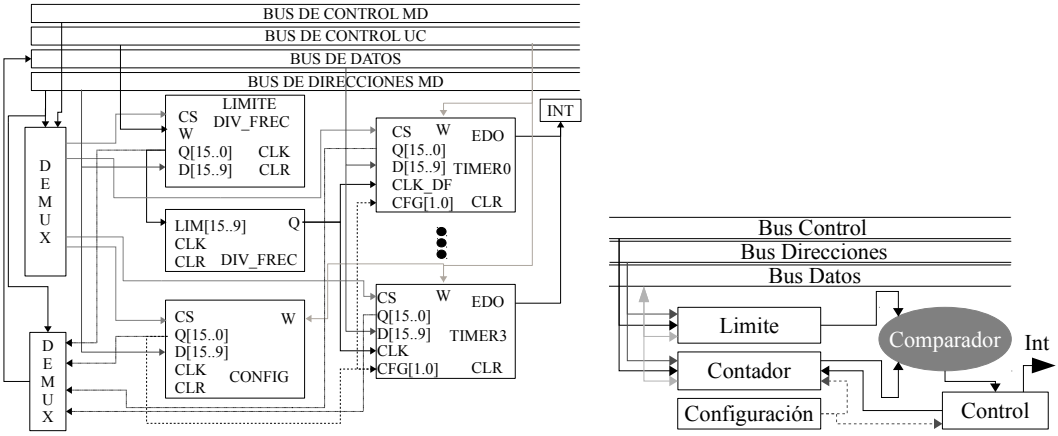
4.3.4. Temporizador

Se establecen cuatro temporizadores todos conectados a un divisor de frecuencia en común, el limite del divisor de frecuencia establece el tiempo que demora en cambiar el reloj de los temporizadores de cero a uno. Los registros de configuración y limites están mapeados en memoria, para poder ser programado. Cada temporizador tiene un limite establecido por un registro, este se compara con un contador de cero al limite establecido, cuando se cumple un ciclo el control activa la interrupción y se reinicia el contador. Con el registro de configuración se habilita a los cuatro temporizadores y también desactiva la señal de interrupción para continuar con el proceso. Las figuras 4.30 y 4.31 describen al temporizador



Señal	E/S	Bits	Descripción
D	E	16	Bus de datos de entrada
Q	S	16	Bus de datos de salida
R	E	1	Habilita la lectura de datos
W	E	1	Habilita la escritura de datos
CLR	E	1	Reinicio de los registros
CLK	E	1	Reloj
CS	E	1	Habilita lectura y escritura de datos
DIR	E	3	Selector de registro

Figura 4.30: Símbolo del temporizador



(a) Diagrama a bloques (b) Estructura de un temporizador

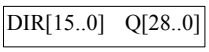
Figura 4.31: Temporizador

4.4. Memoria

La memoria para los microcontroladores es muy limitada, con 256 localidades de RAM para los datos y 1024 localidades de ROM para las instrucciones es más que suficiente para realizar un conjunto limitado de aplicaciones utilizando los microcontroladores. En esta sección se muestra la organización de memoria manejado por páginas para la memoria de datos, utilizando un modelo de memoria Harvard que separa los datos de las instrucciones.

4.4.1. Memoria de programa

Es una memoria lineal que cuenta con 16 bits para direccionar la memoria de programa MP, el programa en ensamblador que es a su vez traducido a lenguaje máquina limita el número de localidades disponibles. Se tiene como máximo 2^{16} instrucciones almacenadas en este componente, mientras que el dispositivo lógico programable lo soporte. El tamaño de palabra que almacena es de 29 bits, el formato de la instrucciones es ilustrado en la tabla 4.11, el símbolo de este componente se ve en la figura 4.32.

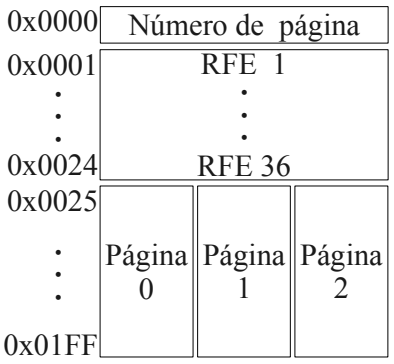


Señal	E/S	Bits	Descripción
Q	S	29	Bus de instrucciones
DIR	E	16	Dirección de memoria

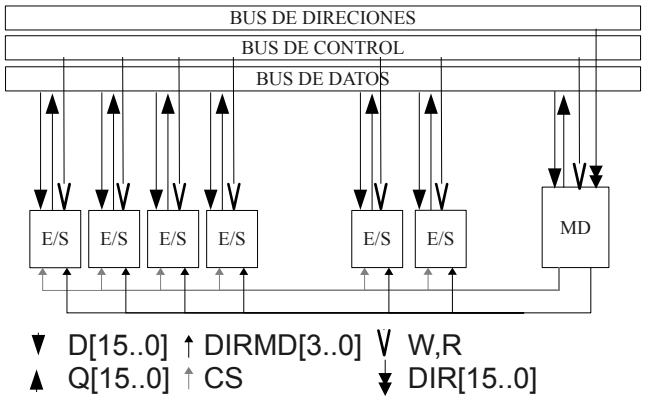
Figura 4.32: Símbolo de la memoria de programa

4.4.2. Memoria de datos

La memoria de datos, almacena la información que se va generando con el programa, se reservan las primeras 48 localidades de memoria para el mapeado de registros de función especial que configuran a los distintos periféricos, cada página puede manejar un límite variado de localidades válidas. La primer localidad de memoria modifica la página seleccionada, por lo que debe forzosamente estar mapeada en las tres páginas, de esta forma no importa en qué página se encuentre en la ejecución del programa se puede cambiar con solo actualizar la primer localidad de memoria con el número de página, es decir: MD[0] ← 0x0000 para pasar a la página cero, MD[0] ← 0x0001 para pasar a la página uno y MD[0] ← 0x0002 para pasar a la página dos.

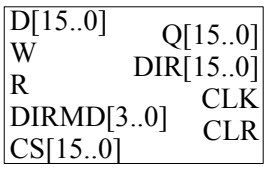


(a) Paginado en la MD



(b) Mapeo de periféricos en la MD

Figura 4.33: Comportamiento de la memoria de datos



Señal	E/S	Bits	Descripción
D	E	16	Bus de datos de entrada
Q	E-S	16	Bus de datos de salida
R	E	1	Habilita la lectura de datos
W	E	1	Habilita la escritura de datos
CLR	E	1	Reinicio interno
CLK	E	1	Reloj
CS	S	16	Selección de periférico mapeado
DIRMD	S	4	Selección de registro mapeado
DIR	E	16	Dirección de memoria

Figura 4.34: Símbolo de la memoria de datos

La figura 4.33(a) muestra un ejemplo donde las primeras 37 localidades comparten los mismos registros mapeados bajo las tres páginas, las tres páginas tienen cada una 475 localidades de memoria RAM. La figura 4.33(b) muestra como la memoria de datos tiene control de los periféricos, todos los componentes reciben las señales de control de lectura (R) y escritura (W), la memoria de datos se encarga de establecer que periférico o que página de memoria debe ser seleccionada, esto lo realiza con la señal de control selector de chip (CS). La señal DIRMD establece un máximo de 15 registros para acceder en cada periférico. Internamente la dirección de una localidad de memoria se calcula restando la dirección de entrada DIR menos el número de registros mapeados en una página. En el ejemplo cuando DIR[15..0] es igual a 50 se le resta 37 dando como resultado la dirección 13 que es la dirección correcta. La figura 4.35 ilustra la configuración interna con tres páginas de memoria y el bus de control de la memoria de datos que mapea algunos registros, la figura 4.34 detalla la interfaz de este componente.

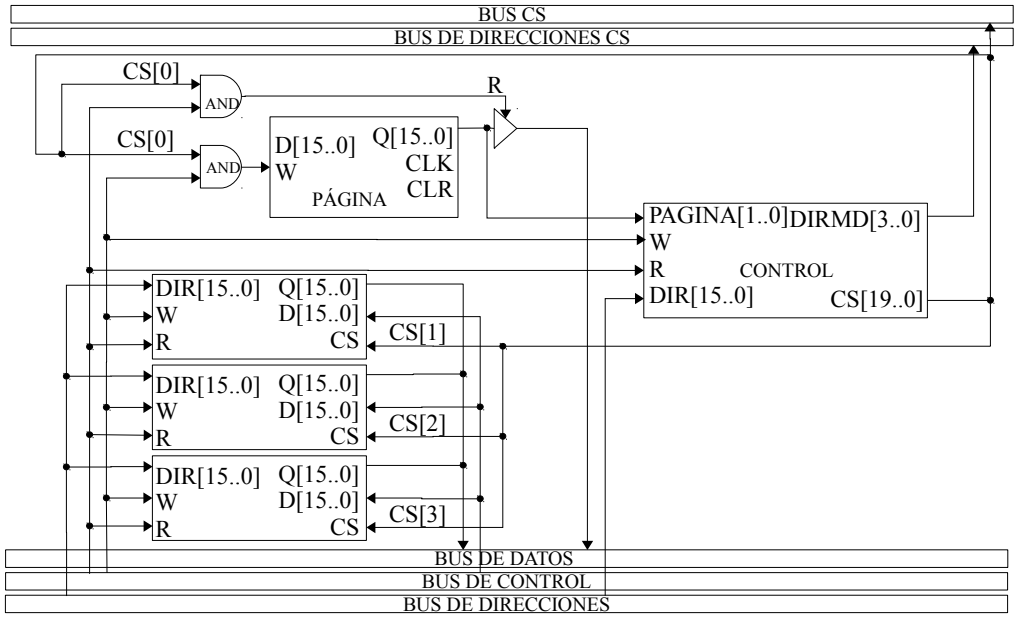
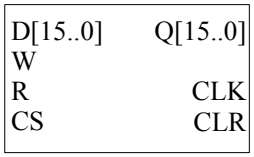


Figura 4.35: Diagrama a bloques de la memoria de datos

4.4.3. Registro de función especial

Los registros de función especial RFE se encarga de configurar al procesador, de acuerdo a las especificaciones de la aplicación en la que se use el procesador estos adquieren cierto valor específico y activan o desactivan las funciones de los componentes que conforman al microcontrolador.



Señal	E/S	Bits	Descripción
D	E	16	Bus de datos de entrada
Q	S	16	Bus de datos de salida
R	E	1	Habilita la lectura de datos
W	E	1	Habilita la escritura de datos
CLR	E	1	Reinicio interno
CLK	E	1	Reloj
CS	E	1	Habilita lectura y escritura de datos

Figura 4.36: Símbolo del registro de función especial

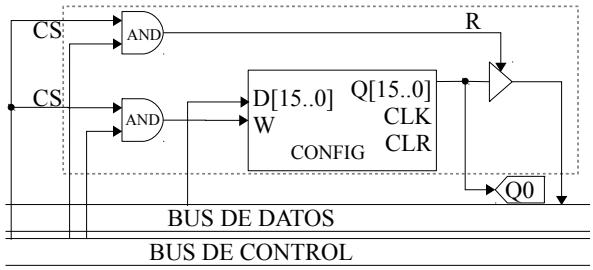


Figura 4.37: Diagrama a bloques de un registro de función especial

El símbolo y diagrama a bloques se ilustran en las imágenes 4.36 y 4.37 respectivamente.

4.5. Recursos auxiliares

La señal global que reinicia a todo el procesador es el reset o CLR, inicializa todos los registros y localidades de la memoria de datos con 0x0000, el contador de programa apunta a la dirección 0x0000 de la memoria de programa, se deshabilita las interrupciones, se desactivan los cuatro temporizadores, con el perro guardián y el módulo PWM. Los puertos paralelos son de entrada, el programa se reinicia y se encarga de habilitarlos como salida.

4.5.1. Perro guardián

El perro guardián es un temporizador que activa un divisor de frecuencia y un contador que va del intervalo de 0 a CONTLIM. La señal de reloj (CLOCK) sirve para coordinar las acciones de varios circuitos combinacionales, según su aplicación la señal se puede repetir con una frecuencia definida se representa con un bit (0 y 1), esta es generada con un divisor de frecuencia cuyo contador interno es establecido por el registro DIVLIM.

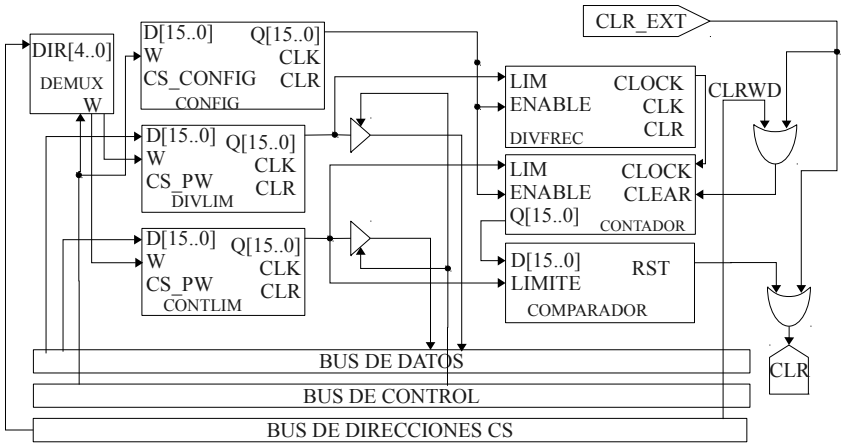


Figura 4.38: Diagrama a bloques del perro guardián

El evento que se genera cuando el contador hace un conteo de 0 a DIVLIM es reiniciar de forma interna todos los componentes del microcontrolador (CLR_{int}). Para evitar este evento se debe reiniciar periódicamente este contador para que nunca llegue al límite, se debe poner en alto la señal CLRWD, esto se logra al utilizar la instrucción *clrwdg*. En la figura 4.39 se muestra la interfaz de señales de este componente y el diagrama a bloques se ve en la figura 4.38.

D[15..0]	Q[15..0]
W	DIR[1..0]
R	CLK
CS	CLR
CLR	ENABLE
CLR_EXT	CLRWD

Señal	E/S	Bits	Descripción
D	E	16	Bus de datos de entrada
Q	S	16	Bus de datos de salida
R	E	1	Habilita la lectura de datos
W	E	1	Habilita la escritura de datos
CLR	S	1	Señal de reinicio interno
CLK	E	1	Señal de reloj
CS	E	1	Habilita lectura y escritura de datos
CLR_{EXT}	E	1	Señal de reinicio externo
DIR	E	2	Selector de registro
ENABLE	E	1	Habilita el perro guardián
CLRWD	E	1	Reinicia el contador del perro guardián

Figura 4.39: Símbolo del perro guardián

Capítulo 5

Ensamblador

Los lenguajes formales como los lenguajes de programación obedecen a reglas preestablecidas y por tanto, se ajustan a ellas, no evolucionan y han sido creados para un fin específico. Se define lenguaje como un conjunto de palabras que están compuestos de símbolos de un alfabeto. Una gramática da cuenta de la estructura de un lenguaje, es decir, de las sentencias que lo forman, proporcionando las formas válidas en que se pueden combinar los símbolos del alfabeto.

Una máquina abstracta o autómata es un dispositivo teórico capaz de recibir y transmitir información. Para realizar esta labor manipula cadenas de símbolos que se suministran en la entrada, produciendo como salida otras cadenas de símbolos en cada momento. Para realizar esto es necesario un conjunto de estados internos requeridos para poder deducir a partir de la entrada una salida de información. La figura 5.1 describe la relación de la terna lenguaje-gramática-máquina.

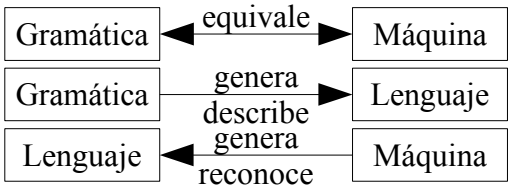


Figura 5.1: Relación de una gramática, lenguaje y la máquina abstracta.

Los autómatas finitos son reconocedores que solo dicen *si* o *no* en relación con cada posible cadena de entrada. Los autómatas finitos puede ser de dos tipos: autómata finito determinista [AFD] que tiene para cada estado y para cada símbolo de su alfabeto de entrada, exactamente una línea (transición) con ese símbolo que sale de esa transición; o autómata finito no determinista [AFN] donde puede existir más de una transición o ninguna por cada tupla (estado, entrada), se tienen varias opciones o ninguna. Tanto los autómatas AFN como los AFD son capaces de reconocer los mismos lenguajes. De hecho los lenguajes son exactamente los mismos lenguajes, conocidos como lenguajes regulares, que pueden describir las expresiones regulares.

Noah Chomsky definió cuatro gramáticas formales, que se diferencian en los tipos de

producciones de la gramática. A lo largo del capítulo se presenta la notación de la *gramática libre de contexto* que se utiliza para describir la sintaxis de un lenguaje de programación. Una gramática describe de forma natural la estructura jerárquica de la mayoría de las instrucciones de un lenguaje de programación.

A lo largo de este capítulo se describe el lenguaje ensamblador, así como el programa ensamblador que se encarga de transformar el programa en esté lenguaje al correspondiente en lenguaje de máquina.

5.1. Notación

Para describir un lenguaje se utilizan diversas herramientas que muestra la estructura léxica (el conjunto de tokens) y sintáctica (el conjunto de reglas gramaticales encargadas de verificar la validez de las expresiones de tokens). Estos elementos son: las expresiones regulares, las reglas gramaticales y los diagramas de sintaxis.

5.1.1. Gramática libre de contexto

Para definir un lenguaje de programación [28] se utiliza una gramática **libre de contexto** que es conformada por la cuádrupla: $G \leftarrow (T, N, P, S)$, donde cada elemento se define como:

- T un conjunto de símbolos terminales a los que se les conoce como tokens. Los terminales son los símbolos elementales del lenguaje definido por una gramática.
- N un conjunto de símbolos no terminales (disjunto de T), se les conoce como variables sintácticas. Cada no terminal representa un conjunto de cadenas o terminales.
- P un conjunto de reglas gramaticales (de la forma $B \to \beta$, donde $B \in N$ y $\beta \in (T \cup N)^*$). $(T \cup N)^*$ significa "todos los símbolos posibles que resultan de la unión de T y N, incluyendo la palabra vacía".
- S un símbolo inicial, donde $S \in N$.

Un lenguaje L generado por la gramática G denotado como

$$L(G) = \{w \in T, \text{ existe una derivación } S \Longrightarrow {}^*w\}.$$

Una regla gramatical [29] es conocida también como producción, una derivación es una secuencia de producciones que parte del símbolo inicial S hasta una cadena de símbolos determinada obtenida a través de las reglas gramaticales. El conjunto de las cadenas de símbolos terminales forman el lenguaje $L(G)$. La derivación [30] se denota como: $\alpha_0 \Longrightarrow \alpha_1 \Longrightarrow \ldots \Longrightarrow \alpha_n$, que se simplifica como $\alpha_0 \Longrightarrow {}^*\alpha_n$, donde $n \geq 0$.

El uso de una regla se representan por la flecha $\Longrightarrow$. Esta gramática también es conocida como independientes del contexto: la parte izquierda de la producción sólo puede tener un símbolo no terminal, es decir:

$$P \leftarrow \{(S \to \varepsilon) \text{ ó } (A \to v) | A \in N, v \in T^+\}$$

donde la expresión T^+ consiste en todas las palabras generadas con todas las combinaciones posibles concatenando símbolos del alfabeto terminal (Ej.: dado $T = \{0, 1\}$, T^+ es igual a {0,1,00,01,10,11,000,0001,...}), excluyendo la palabra vacía ε. Con la diferencia de que $(T \cup N)^*$ incluye la palabra vacía y la concatenación de las palabras que se obtienen de la unión de T y N.

Esta gramática se define de contexto libre, porque a la hora de transformar una palabra en otra, el símbolo no terminal que se transforma no depende de los que estén a la izquierda o a la derecha. Así cuando se realicen derivaciones para transformar el símbolo A, no hace falta saber que hay alrededor de él.

5.1.2. Expresiones regulares

Las expresiones regulares representan patrones de cadenas de caracteres. Una expresión regular se define mediante el conjunto de cadenas con las que concuerda, tal conjunto se llama lenguaje generado por la expresión regular. Sea r una expresión regular y $L(r)$ el lenguaje generado por la expresión regular [28]. El lenguaje depende de caracteres o símbolos cuyo conjunto se conoce como alfabeto, representado por la letra griega Σ.

En una expresión regular todos los símbolos o caracteres indican patrones, existen algunos caracteres especiales llamados metacaracteres o metasímbolos y no deben ser caracteres legales en el alfabeto. El carácter de escape desactiva el significado especial de un metacaracter. Las cadenas de caracteres o palabra es una secuencia de longitud arbitraria de elementos del alfabeto. La cadena que no tienen ningún carácter cuya longitud es cero se llama cadena vacía y se representa por ε . El conjunto que no tiene ninguna cadena se representa por $\boldsymbol{\phi}$.

Un carácter por si solo puede representar una expresión regular, al utilizarlo en las expresiones regulares se escribe en negritas. Por ejemplo $\boldsymbol{a} = L(\boldsymbol{a}) = \{a\}$ significa que $\boldsymbol{a}$ es el carácter a usado como patrón. También se cuenta con los siguientes patrones: $\boldsymbol{\varepsilon} = L(\varepsilon) = \{\varepsilon\}$ y $\boldsymbol{\phi} = L(\phi) = \{\}$.

La tilde es el metacaracter que sirve para evitar el uso de un patrón dentro de un expresión regular. Ej.: $\sim (\boldsymbol{a}|\boldsymbol{b}|\boldsymbol{c})$ son todos los patrones que no sean los caracteres a ni b ni c. El metacaracter ^ al principio entre los corchetes igual sirve para evitar uno o varios patrones $[\hat{}\boldsymbol{abc}]$.

Operación	Metacaracter	Expresión regular	Lenguaje
Selección	|	**0 | 1**	{0, 1}
Concatenación		**01**	{01}
Agrupación	()	**01**(**0 | 1** | $\boldsymbol{\varepsilon}$)	{010, 011, 01}
Repetición	$*$	$\mathbf{01}^*$	{0, 01, 011, 0111, ...}
	$+$	$\mathbf{01}^+$	{01, 011, 0111, ...}
Opcional	?	(**0**|**1**)?	{ε, 0, 1}

Tabla 5.1: Ejemplo de metacaracteres

La tabla 5.1 enlista los metacaracteres más utilizados en las expresiones regulares, en los ejemplos se utiliza el alfabeto $\Sigma = \{0, 1\}$.

Para largas secuencias de selecciones se utilizan los puntos $\mathbf{0} \mid \mathbf{1} \mid \cdots \mid \mathbf{99}$. La precedencia de los metacaracteres toma a la repetición como la mayor, seguido por la concatenación y por último la selección. Los paréntesis sirven para indicar una precedencia diferente. Ej.: $(\mathbf{01})^* = \{\varepsilon, 01, 0101, \dots\}$, sin paréntesis $\mathbf{01}^* = \{0, 01, 011, \dots\}$ o $\boldsymbol{a} \mid \boldsymbol{bc} = \{a, bc\}$, con paréntesis $(\boldsymbol{a} \mid \boldsymbol{b})\boldsymbol{c} = \{ac, bc\}$.

Asignar un nombre a una expresión regular larga simplifica la notación evitando escribir el patrón de caracteres más de una vez. Ej.: digito = ($\mathbf{0} \mid \mathbf{1} \mid \mathbf{2} \mid ... \mid \mathbf{9}$)

A menudo es necesario escribir un intervalo de caracteres, por ejemplo las letras minúsculas del alfabeto $\boldsymbol{a} \mid \boldsymbol{b} \mid \cdots \mid \boldsymbol{z}$ una alternativa a esta notación es utilizar los metacaractes corchetes y guion, el equivalente es $[\boldsymbol{a}-\boldsymbol{z}]$, para representar un carácter hexadecimal el patrón es [**a-fA-F0-9**]. El metacaracter punto (.) sirve para asociarlo con cualquier carácter sin incluir los saltos de línea (\n), Ej.: donde $\Sigma = \{0, 1\}$ se tiene que $(.^*) = \{\varepsilon, 0, 1, 00, 01, 10, 11, \dots\}$.

El patrón para un número entero base 16 y 10 se tiene como sigue:

$$\text{digito}_{16} = [\mathbf{0\text{-}9a\text{-}fA\text{-}F}] \tag{5.1}$$

$$\text{digito}_{10} = [\mathbf{0\text{-}9}] \tag{5.2}$$

$$\text{int}_{16} = \text{digito}_{16}{}^{+} \tag{5.3}$$

$$\text{int}_{10} = \text{digito}_{10}{}^{+} \tag{5.4}$$

El patrón para el ensamblador diseñado, para el tipo de instrucción es:

$$\begin{aligned}\text{tipo}_{\text{instrucción}} = &\mathbf{load}|\mathbf{store}|\mathbf{jmp}|\mathbf{jset}|\mathbf{jclr}|\mathbf{add}|\mathbf{sub}|\mathbf{cmp}|\mathbf{not}|\mathbf{reti}|\mathbf{clrwdg}|\\ &\mathbf{halt}|\mathbf{srl}|\mathbf{sra}|\mathbf{sll}|\mathbf{sla}|\mathbf{rorc}|\mathbf{rolc}|\mathbf{inv}|\mathbf{ror}|\mathbf{rol}|\mathbf{xor}|\mathbf{xnor}|\mathbf{or}|\mathbf{nor}|\\ &\mathbf{and}|\mathbf{nand}|\mathbf{calla}|\mathbf{ret}|\mathbf{call}|\mathbf{clrb}|\mathbf{setb}|\mathbf{clr}|\mathbf{set}|\mathbf{nop}|\mathbf{push}|\mathbf{pop}\end{aligned}$$

El patrón para indicar una directiva es:

$$\text{directiva} = \#(\mathbf{include}|\mathbf{end}|\mathbf{equ})$$

El patrón general de todas las instrucciones y directivas es:

$$op\ arg_1,\ arg_2,\ \dots,\ arg_n$$

donde *op* es el tipo instrucción o directiva, seguida de una lista de argumentos arg_i.

5.1.3. Reglas de una gramática libre de contexto

La estructura sintáctica de un lenguaje de programación se especifica mediante reglas, o dicho de otra manera: podemos especificar un lenguaje de programación mediante reglas recursivas.

El lenguaje de la jerarquía de Chomsky que puede ser reconocido por un autómata de pila se le llama *lenguaje libre de contexto*, que se especifica por una *gramática libre de contexto*. La notación más usada para describir la gramática mediante reglas recursivas, es la forma Backus-Naur (BNF) creada por John Backus [31].

El conjunto de *tokens* o componentes léxicos son las cadenas de caracteres que tienen un significado para un lenguaje de programación. Para el lenguaje ensamblador el conjunto de tokens es definido por:

$$\begin{aligned} Palabras_{Reservadas} \rightarrow & load|store|jmp|jset|jclr|add|sub|cmp|not|reti|clrwdg|srl|sra|sll|sla| \\ & rorc|rolc|inv|ror|rol|xor|nor|or|nor|and|nand|calla|ret|call|clrb|setb| \\ & clr|set|nop|push|pop|halt|include|end|equ|status|reg0|reg1|reg2|reg3| \\ & reg4|reg5|reg6|reg7|reg9|reg8|reg10|reg11|reg12|reg13|reg14|reg15 \end{aligned} \tag{5.5}$$

$$Caracteres_{Especiales} \rightarrow [\ |\ ]\ |,|+|-|;|\sharp \tag{5.6}$$

$$Tokens \rightarrow Caracteres_{Especiales}|Palabras_{Reservadas} \tag{5.7}$$

Dado el alfabeto, una regla se componen de una cadena de símbolos, el primer símbolo es el nombre de la estructura; el segundo símbolo es el metasímbolo $\rightarrow$, seguido por una cadena de símbolos del alfabeto: sea el nombre de una estructura o el meta símbolo | para separar las opciones. Un ejemplo sencillo para la instrucción **or**:

$$\begin{aligned} instrucción_{or} &\rightarrow or\ reg_i, arg_{fuente} \\ arg_{fuente} &\rightarrow int_i|(reg_i,\ reg_i) \\ int_i &\rightarrow int_{10}|int_{16} \\ reg_i &\rightarrow reg0|\ldots|reg15 \end{aligned}$$

Describir la instrucción or en un modo de direccionamiento inmediato es $or\ reg_i, int_i$ ($reg_i \leftarrow reg_i \bigvee cte$). Para el direccionamiento por registro es $or\ reg_i,\ reg_i,\ reg_i$ ($reg_i \leftarrow reg_i \bigvee reg_i$).

La notación BNF extendida, o EBNF simplifica la tarea de describir un lenguaje. La repetición se expresa utilizando la recursión por la izquierda $A \rightarrow A\alpha|\beta$ o por la derecha $A \rightarrow \alpha A|\beta$, donde α y β son cadenas arbitrarias de terminales y no terminales (nombres de otras reglas), teniendo en cuenta que en la primera β no comienza con A y la segunda β no finaliza con A . En lugar de la recursión se puede emplear el asterisco * (conocida como cerradura de Kleene en expresiones regulares) la representación de las reglas anteriores quedarían como: $A \rightarrow \beta\alpha^*$ y $A \rightarrow \alpha^*\beta$. Con EBNF se prefiere usar las llaves: $A \rightarrow \beta\{\alpha\}$ y $A \rightarrow \{\alpha\}\beta$. Para las instrucciones que tienen al menos un argumento puede expresarse al utilizar la recursividad por la izquierda:

$$instrucción \rightarrow opcode\ arg\ \{, arg\}$$

Las estructuras opcionales en EBNF se indican encerrándolas entre corchetes [...], es el equivalente al signo de interrogación en las expresiones regulares. En las instrucciones donde los argumentos son opcionales se utilizan los corchetes:

$$instrucción \rightarrow opcode\ [arg\ \{, arg\}]$$

Las regla expresadas por EBNF se puede mostrar en un diagrama de sintaxis, donde los círculos representan cadenas de caracteres o símbolos terminales y las cajas distinguen a otras reglas o símbolos no terminales. Las flechas indican la selección y la secuencia. El diagrama

de sintaxis que representa a la regla *instrucción* con argumentos opcionales se ilustra en la figura 5.2. La figura 5.3 ilustra el diagrama de sintaxis equivalente cuando se emplean llaves o corchetes.

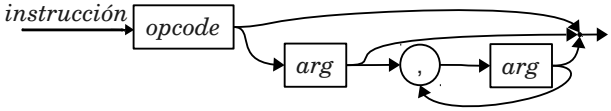


Figura 5.2: Ejemplo del diagrama de sintaxis

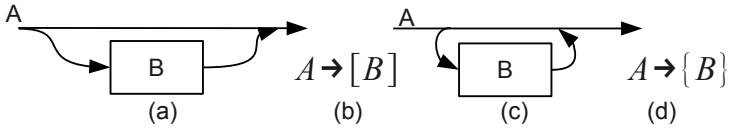


Figura 5.3: Regla EBNF para la regla de repetición (b) y para la opcional (d). Diagrama de sintaxis para la repetición (a) y para la opcional (c).

5.2. Descripción del lenguaje ensamblador

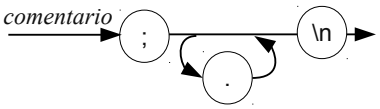
Las constantes numéricas se representan con números según las expresiones (5.3) y (5.4) en la página 66. Es decir, solo se representan números enteros positivos para base decimal y hexadecimal (de 0 hasta $2^{16} - 1$). Cuando se utiliza la instrucción *sub* y el resultado es negativo se obtiene el resultado representado en complemento a dos.

Expresión	Modo	Recurso	Identificador	Ejemplo
$[id\|cte]$	Directo	MD	#equ PG1, 1	store [0], PG1
id	Directo	MP	:loop	jmp loop
$id\|cte$	Inmediato	BI	#equ UNO, 1	load reg1, UNO
reg_i	Registro	AR	-	load reg1, reg0
$[reg_i(+\|-)reg_j]$	Base indexada	MD	-	load reg0, [reg1-reg2]

Tabla 5.2: Ejemplos de algunos modos de direccionamiento, cuyos recursos relacionados pueden ser el archivo de registros AR, la memoria de datos MD, la memoria de programa MP o el bus de instrucción BI

Las palabras reservadas se definen en la expresión (5.5) junto con los caracteres especiales (5.6). La coma es un separador de argumentos, los corchetes dan acceso a la memoria de datos, la suma y resta pueden servir para el direccionamiento base indexada que toma un registro base con un registro índice entre corchetes, la tabla 5.2 muestra algunos ejemplos.

Los comentarios se encargan de documentar un archivo de código fuente para facilitar su lectura. Estos son ignorados por el ensamblador por que no afectan en nada el proceso de traducción a lenguaje máquina, pero son muy útiles para el programador. Un comentario comienza con punto y coma, toda la línea es ignorada después de este carácter, la figura 5.4 muestra su descripción.

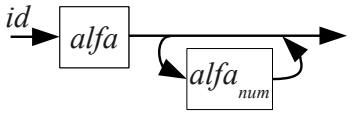


;(.)*\n *comentario* → ; {.}\n

(a) (b) (c)

Figura 5.4: El comentario, en (a) su expresión regular, en (b) su regla EBNF y en (c) su diagrama de sintaxis

Los identificadores son cadenas de símbolos que nombran alguna entidad, son elementos textuales para denotar constantes numéricas o etiquetas que indican el flujo del programa en las instrucciones de salto y las llamadas a subrutinas. La figura 5.5 muestra la descripción de un identificador.



$alfa \leftarrow$[a-zA-Z]

$alfa_{num} \leftarrow$[a-zA-Z0-9]

$id \leftarrow alfa^{+}\ (\_ \mid alfa_{num})^{*}$

$alfa \rightarrow$ a|...|z|A|...|Z

$alfa_{num} \rightarrow$ 0|...|9|$alfa$

$aux_{id} \rightarrow \_ \mid alfa_{num}$

$id \rightarrow alfa\ \{aux_{id}\}$

(a) (b) (c)

Figura 5.5: El identificador, en (a) su diagrama de sintaxis, y en (b) su expresión regular y en (c) su regla EBNF.

Las directivas del ensamblador no son instrucciones del procesador, son utilizadas como herramientas durante la traducción al lenguaje de máquina. Facilita el desarrollo que se realiza con el lenguaje ensamblador. La figura 5.6 muestra los diagramas de sintaxis y la representación en EBNF de las directivas y constantes.

$directiva_{equ} \rightarrow \sharp equ\ id, cte$

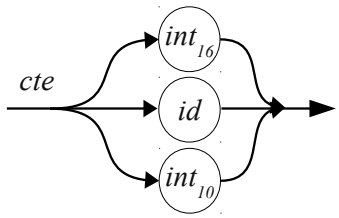
$cte \rightarrow id|int_{10}|int16$

$directiva_{include} \rightarrow \sharp include\ \text{“}alfa_{num}\ \{cadena_{caracteres}\}\text{”}$

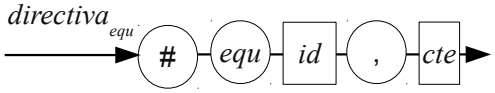
$cadena_{caracteres} \rightarrow alfa_{num} \mid \backslash. \mid \_$

$directiva_{end} \rightarrow \sharp end$

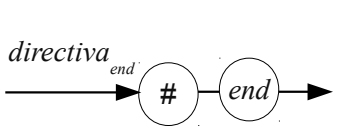
(a) Regla EBNF

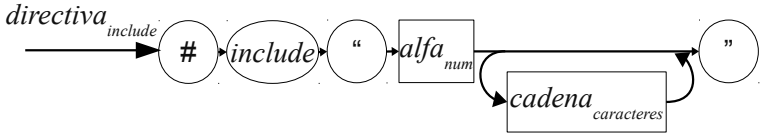


(b) Constantes



(c) Asignación





(d) Finalizar (e) Incluir

Figura 5.6: Las directivas y constantes, en (a) sus reglas EBNF y en (b), (c), (d) y (e) sus diagramas de sintaxis.

Para modificar el flujo del programa se necesita conocer la dirección de memoria de la siguiente instrucción a ejecutar, al crear el código fuente no es posible conocer que dirección

es asignada para cualquier instrucción, para facilitar este trabajo se utilizan las etiquetas dando al argumento de la instrucción el identificador adecuado que se definió en algún lugar del código fuente. La figura 5.7 describe a las etiquetas.

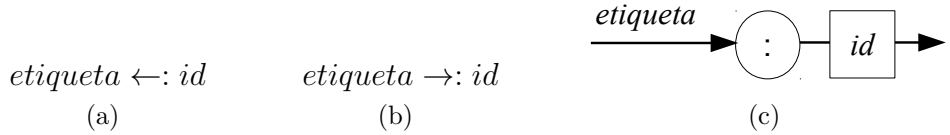
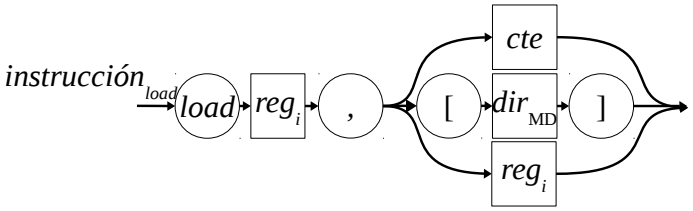
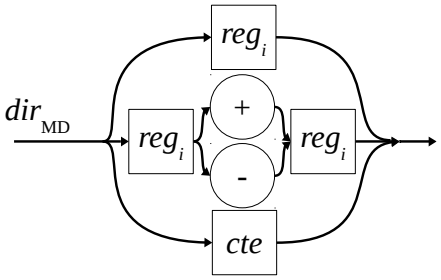


Figura 5.7: Etiqueta expresión regular en (a), en (b) su regla EBNF y en (c) su diagrama de sintaxis.

Las instrucción *load* carga una palabra (entero de 16 bits) en un registro, el origen puede ser: un valor inmediato (cte), copiarlo de otro registro (reg_i) o de alguna localidad de memoria de datos (directo $[cte]$, registro indirecto$[reg_i]$ o base indexado $[reg_i(+|-)reg_i]$). La figura figura 5.8 muestra la descripción de la instrucción *load*.



(a) Instrucción *load*



(b) Dirección de memoria de datos

$$
\begin{aligned}
instrucción_{load} &\rightarrow destino_{load},\ fuente_{load} \\
destino_{load} &\rightarrow reg_i \\
fuente_{load} &\rightarrow cte\ |reg_i \mid fuente_{\mathrm{MD}} \\
fuente_{\mathrm{MD}} &\rightarrow [dir_{\mathrm{MD}}] \\
dir_{\mathrm{MD}} &\rightarrow reg_i|cte|base_{indice} \\
base_{indice} &\rightarrow reg_i\ (+|-)\ reg_i
\end{aligned}
$$

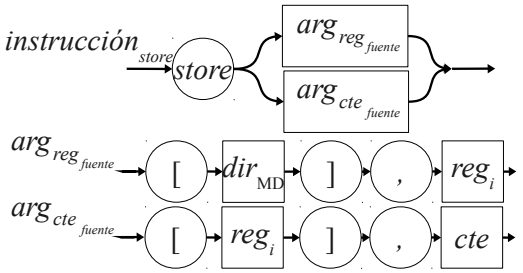
(c) Reglas EBNF

Figura 5.8: La instrucción *load* en (a) y en (b) su diagrama de sintaxis, y en (c) sus reglas EBNF.

La instrucción *store* almacena una palabra en una localidad de memoria. Cuando la dirección de memoria de datos es directo ($[cte]$) el valor para almacenar se obtiene de un registro, cuando la localidad de memoria se obtiene por registro indirecto ($[reg_i]$) el valor puede ser inmediato (cte) o por registro (reg_i), por ultimó cuando la dirección se obtiene por base indexada $[reg_i(+|-)reg_i]$ el dato solo está en un registro (reg_i). La figura 5.9 describe la instrucción *store*.

$$
\begin{aligned}
instrucción_{store} &\rightarrow store\ arg_{store} \\
arg_{store} &\rightarrow arg_{reg_{fuente}} | arg_{cte_{fue}} \\
arg_{reg_{fuente}} &\rightarrow [dir_{MD}],\ reg_i \\
arg_{cte_{fuente}} &\rightarrow [reg_i],\ cte
\end{aligned}
$$

(a)



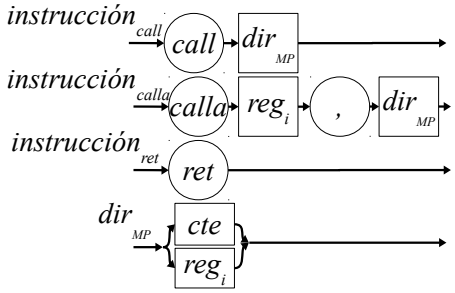
(b)

Figura 5.9: Instrucción *store*, en (a) sus reglas EBNF y en (b) su diagrama de sintaxis.

Las instrucciones para subrutinas: la llamada a subrutina *call* recibe la dirección de la subrutina de forma inmediata *cte* o dentro de un registro reg_i, tiene asociada a la instrucción de retorno de subrutina *ret*; la otra llamada de subrutina es *calla*, que almacena el actual valor del contador de programa en un registro, el primer argumento es el registro auxiliar para el contador de programa (reg_i) y el segundo argumento es la dirección de memoria de la subrutina (reg_i o *cte*). La figura 5.10 describe las instrucciones relacionadas a las subrutinas.

$$
\begin{aligned}
instrucción_{call} &\rightarrow call\ dir_{\mathrm{MD}} \\
instrucción_{calla} &\rightarrow calla\ reg_i, dir_{\mathrm{MD}} \\
dir_{\mathrm{MD}} &\rightarrow cte \mid reg_i \\
instrucción_{ret} &\rightarrow ret
\end{aligned}
$$

(a)



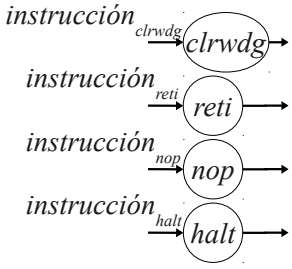
(b)

Figura 5.10: Instrucciones para las subrutinas, sus reglas EBNF en (a) y su diagrama de sintaxis en (b).

Las instrucciones implícitas solo manejan el código de operación sin ningún argumento, la descripción de estas instrucciones se muestra en la figura 5.11.

$$
\begin{aligned}
instrucción_{clrwdg} &\rightarrow clrwdg \\
instrucción_{reti} &\rightarrow reti \\
instrucción_{nop} &\rightarrow nop \\
instrucción_{halt} &\rightarrow halt
\end{aligned}
$$

(a)



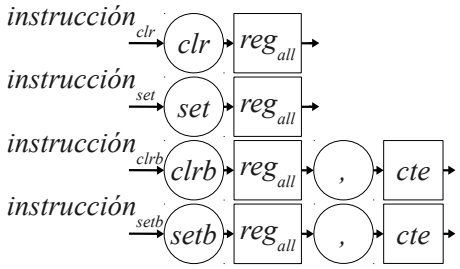
(b)

Figura 5.11: Instrucciones implícitas para el reinicio del perro guardián (*clrwdg*), retorno de interrupción (*reti*), no operación (*nop*) y detener (*halt*): en (a) sus reglas EBNF asociadas y en (b) sus diagramas de sintaxis.

Las instrucciones que limpian un registro o un solo bit poniendo en alto o bajo su valor (bit← 0 o 1, registro← 0x0000 o 0xFFFF), son las instrucciones *clr*, *set*, *clrb* o *setb*. La descripción de estas instrucciones se muestra en la figura 5.12.

$$instrucción_{clr} \rightarrow clr\ reg_{all}$$
$$instrucción_{set} \rightarrow set\ reg_{all}$$
$$instrucción_{clrb} \rightarrow clrb\ reg_{all},\ cte$$
$$instrucción_{clrb} \rightarrow setb\ reg_{all},\ cte$$
$$reg_{all} \rightarrow reg_i \mid status$$

(a)



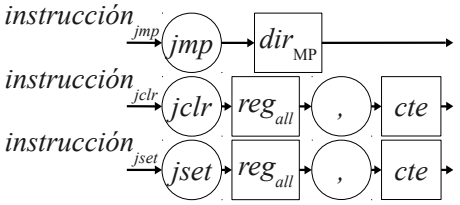
(b)

Figura 5.12: Instrucciones que reinician un registro de propósito general o el registro de status se muestra: en (a) sus reglas EBNF y en (b) sus diagramas de sintaxis.

Las instrucciones de salto alteran el contador de programa se necesita proveer una dirección de memoria de programa, las instrucciones condicionales ($jclr$ y $jset$) incrementa el contador de programa en dos (CP+2) si se cumple la condición de que un bit de registro evaluado sea igual a cero para $jclr$ o uno para $jset$ ($reg_{all}[cte] = x, x \in \{0, 1\}$). El salto incondicional acepta la dirección de forma explícita (dir_{MP}). La figura 5.13 describe la instrucción de salto.

$$instrucción_{jmp} \rightarrow jmp\ dir_{\text{MP}}$$
$$instrucción_{jclr} \rightarrow jclr\ reg_{all},\ cte$$
$$instrucción_{jset} \rightarrow jset\ reg_{all},\ cte$$

(a)



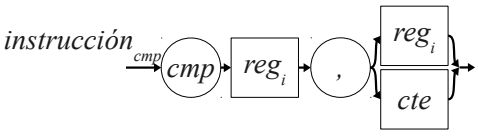
(b)

Figura 5.13: Instrucciones de salto incondicional (jmp) y condicional (jset y jclr), en (a) sus reglas EBNF y en (b) sus diagramas de sintaxis.

La instrucción de comparación resta dos valores, el primer valor es obtenido por un registro (reg_i) y el segundo puede ser un valor inmediato (cte) u otro registro (reg_i). Las instrucciones de salto se asocian con esta instrucción, *cmp* modifica los bits del registro *status*. La figura 5.14 describe la instrucción de comparación.

$$instrucción_{cmp} \rightarrow cmp\ reg_i,\ op_{cmp}$$
$$op_{cmp} \rightarrow reg_i | cte$$

(a)



(b)

Figura 5.14: Instrucción *cmp*, en (a) sus reglas EBNF y en (b) su diagrama de sintaxis.

Las instrucciones que utilizan la unidad aritmética y lógica modifican el contenido de un registro reg_i, necesitan de otros operandos que pueden ser obtenidos de forma inmediata *cte* o de registros reg_i. El tipo de operación se obtiene del nemónico ($add|sub|\ldots|inv$). Las reglas EBNF asociadas a estas instrucciones son:

$$
\begin{aligned}
opcode_{ALU_{\text{otras}}} &\rightarrow not|inv \\
opcode_{ALU_{\text{lógicas}}} &\rightarrow and|or|xor|nand|nor|xnor \\
opcode_{ALU_{\text{aritméticas}}} &\rightarrow add|sub \\
opcode_{ALU_{\text{shift}}} &\rightarrow srl|sra|sll|sla|ror|rol|rorc|rolc \\
opcode_{ALU_{registro}} &\rightarrow opcode_{ALU_{\text{lógicas}}}|opcode_{ALU_{\text{aritméticas}}}|opcode_{ALU_{\text{shift}}} \\
opcode_{ALU_{cte}} &\rightarrow opcode_{ALU_{\text{lógicas}}}|opcode_{ALU_{\text{aritméticas}}}|opcode_{ALU_{\text{otras}}} \\
instrucción_{ALU_{registro}} &\rightarrow opcode_{ALU_{registro}}\ reg_i,\ reg_i,\ reg_i \\
instrucción_{ALU_{inmediato}} &\rightarrow opcode_{ALU_{cte}}\ reg_i,\ cte \\
instrucción_{ALU_{shift}} &\rightarrow opcode_{ALU_{shift}}\ reg_i,\ reg_i,\ cte \\
instrucción_{ALU_{otras}} &\rightarrow opcode_{ALU_{otras}}\ reg_i,\ reg_i
\end{aligned}
$$

La pila es un tipo de estructura de datos con organización en la que el último en entrar es el primero en salir LIFO (*Lash In First Out*). Las reglas EBNF que describen la sintaxis de las instrucciones *pop* y *push* que controlan la pila son:

$$
\begin{aligned}
instrucción_{pop\_s} &\rightarrow pop\_s\ punter_{pila},\ fuente_{pop} \\
instrucción_{push\_s} &\rightarrow push\_s\ puntero_{pila},\ fuente_{push} \\
instrucción_{pop} &\rightarrow pop\ punter_{pila},\ fuente_{pop} \\
instrucción_{push} &\rightarrow push\ puntero_{pila},\ fuente_{push} \\
fuente_{push} &\rightarrow cte|reg_i \\
fuente_{pop} &\rightarrow reg_i \\
puntero_{pila} &\rightarrow reg_i
\end{aligned}
$$

Las diferencia de las instrucciones *push_s* y *pop_s*, contra las instrucciones *push* y *pop*, consiste en que las primeras dos toman al puntero como límite superior, decrementándolo al poner un elemento e incrementando al quitarlo; el segundo par de instrucciones toman al puntero como límite inferior, lo incrementan al poner un elemento y se decrementa al quitarlo. La sintaxis de ambos pares de instrucciones difieren con el uso del sufijo *_s*.

5.3. Funcionamiento del ensamblador, desensamblador y la descripción de una memoria ROM

El software es el conjunto de programas que reciben el código fuente en lenguaje ensamblador y retorna tres archivos: un archivo con el equivalente del programa en lenguaje máquina, una memoria ROM descrita en VHDL, que almacena el programa en lenguaje máquina, y un archivo resultante de traducir el primer archivo a lenguaje ensamblador (desensamblado).

Las instrucciones en código máquina son cadenas de símbolos donde Σ={0,1,-}, el guion se utiliza para especificar que no importa su valor (puede ser cero o uno). Por ejemplo, en las instrucciones implícitas solo nos interesa el código de operación y su modo de direccionamiento:

[00 0000][- - - - - - - - - - - - - - - - - - -][000] $\Leftrightarrow$ Instrucción *nop*

Para explicar el funcionamiento del ensamblador se detalla con un ejemplo tomando en cuenta el código fuente de la figura 5.15.

```
00    #include "lib.asm"
01    #include "cte.asm"
02    call main
03    halt
04 :main
05    load reg0, 0x0
06 :loop
07    call inc
08    cmp reg0, limite
09    jclr status, ZERO
10    jmp loop
11    ret
12  #end
```

(a) *main.asm*

```
00    #include "cte.asm"
01 :inc  ;subrutina r0++
02    add reg0, step
03    ret
04  #end
```

(b) *lib.asm*

```
00    #equ step  ,   1
01    #equ limite, 0xA
02    #equ ZERO  ,   1
03    #end
04    ;opcional la
05    ;directiva #end
```

(c) *cte.asm*

Figura 5.15: Ejemplo de código fuente de un programa sencillo en lenguaje ensamblador

El archivo principal del programa en ensamblador es *main.asm* se ve en la figura 5.15(a): las líneas 0 y 1 incluyen los archivos de *lib.asm* y *cte.asm*. La línea 2 llama a la subrutina *main* cargando la dirección en el CP (CP $\leftarrow$*main*). La línea 3 detiene el flujo del programa, se ejecuta al retornar de la subrutina *main*. La línea 4 declara la etiqueta donde comienza la subrutina *main* finalizando en la línea 11 en la instrucción de retorno. La línea 5 inicializa en cero al registro cero, es el contador del bucle. La línea 6 establece la etiqueta de salto para permanecer en el bucle. La línea 7 llama a una subrutina *inc* que suma al registro contador el valor de la constante *step*. La línea 8 compara la constante límite con el contenido del registro contador, modificando las banderas del registro *status*. La línea 9 evalúa la bandera *ZERO* del registro *status*, si el contador es distinto al límite se realiza un salto a la dirección indicada por la etiqueta *loop*, si son iguales salta a la instrucción *ret* donde finaliza la subrutina *main*. La línea 12 finaliza el código fuente del archivo por medio de la directiva *#end*, es decir, todas instrucciones o directivas después de esta línea son ignoradas.

El archivo *cte.asm* se ve en la figura 5.15(c), define la constantes por medio de la directiva *#equ* del ensamblador, que tiene que sustituirse por su valor numérico al generar el código en lenguaje de máquina, es decir, establece los valores numéricos asociados a una etiqueta: *step* $\leftarrow$ 1, *limite* $\leftarrow$ 10 y *ZERO* $\leftarrow$ 1.

Por último el archivo *lib.asm* se ve en la figura 5.15(b), declara la subrutina *inc* que suma el contenido del registro cero más la constante *step* y lo almacena en el mismo registro. Todos los comentarios son ignorados.

La figura 5.16(a) muestra el archivo xml que describe el proyecto, los directorios donde se crearan el archivo binario, desensamblado y la descripción de la memoria ROM en VHDL. Para poder ejecutar el software se utiliza la terminal de comandos. El archivo de la figura 5.16(b) crea las carpetas en el directorio de trabajo y ejecuta un *script* en lenguaje de programación *python* que realiza el ensamblado y desensamblado.

```
<proyecto nombre="Test"
            ruta="C:\src\Ejemplo\"
                  version="1.0">
   <asm paso="ensamblador">
       <nombre>main</nombre>
       <ruta>asm\</ruta>
       <extension tipo="asm"/>
   </asm>
   <dasm paso="desensamblador">
      <nombre>main</nombre>
      <ruta>dasm\</ruta>
      <extension tipo="dasm"/>
   </dasm>
   <bin paso="binario">
      <nombre>main</nombre>
      <ruta>bin\</ruta>
      <extension tipo="bin"/>
   </bin>
   <rom paso="meoria">
<nombre>mp</nombre>
      <ruta>rom\</ruta>
      <extension tipo="vhd"/>
   </rom>
</proyecto>
```

(a) Descripcion del proyecto en un archivo XML.

```
mkdir bin
mkdir dasm
mkdir rom
"C:\asm\console.py" proyecto.xml>out.txt
pause
```

(b) Archivo de procesamiento por lotes que se ejecuta en la línea de comandos.

Figura 5.16: Archivos adicionales para generar el programa en lenguaje de máquina.

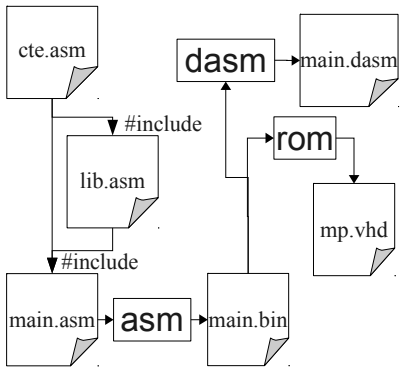


Figura 5.17: Flujo de trabajo del software

La figura 5.17 muestra el flujo de los archivos leídos y los archivos generados. Los pasos seguidos durante el proceso de ensamblado *asm* son:

1. Lee el archivo principal en lenguaje ensamblador y todos los archivos incluidos con la directiva *#include*.
2. Extrae los tokens de cada línea leída, calculando los valores inmediatos o registros involucrados (int_{10}, int_{16}, i de reg_i o status).

3. Intercambia los identificadores de las constantes definidas con la directiva *#equ* por sus valores numéricos correspondientes.
4. Intercambia los identificadores definidos por las etiquetas por su valor numérico (se suma el offset correspondiente).

   El valor de cada etiqueta toma el número de instrucción que tiene a continuación. El valor numérico de la etiqueta dentro del programa en lenguaje máquina se obtiene sumando:

   $$No.\ de\ instrucción + Offset.$$

   El archivo *main.asm* tiene dos etiquetas asociadas: *main* que toma el valor de dos que referencia la posición de la instrucción *load* $reg0,\ 0x0$ y *loop* con valor de tres que referencia la posición de la instrucción *call inc*. El archivo *lib.asm* tiene asociada la etiqueta *inc* con el valor de cero que referencia la instrucción *add* $reg0,\ step$.

   Las instrucciones se acomodan iniciando por las del archivo principal seguidas de las demás. Cada archivo inicializa una etiqueta con la posición de instrucción que referencia, cuando se incorporan todas las instrucciones:

   [instrucciones *main.asm*][instrucciones *lib.asm*][instrucciones *cte.asm*],

   se suma el número de instrucciones anteriores a cada bloque más la dirección que ya tenía asociada la etiqueta. A las etiquetas del archivo *main.asm* no precede ningún bloque su offset es cero dejando el valor de las etiquetas *main* y *loop* intactos, para las etiquetas del archivo *lib.asm* le preceden ocho instrucciones del bloque del archivo *main.asm* por lo que el valor de la etiqueta *inc* es ocho y por último el archivo *cte.asm* no tiene instrucciones por lo que no se toma en cuenta el offset de diez. El archivo *main.asm* tiene asociada un offset de cero y ocho instrucciones, *lib.asm* tiene asociado un offset de ocho y dos instrucciones, el último archivo *cte.asm* tiene asociado un offset de 10 y cero instrucciones.
5. Teniendo la información de todas las instrucciones, es decir, el conjunto de tokens y sus valores numéricos se obtienen todas las líneas en lenguaje de máquina dentro de un archivo de salida. Se puede asociar los identificadores de las etiquetas adelante de la instrucción máquina, para su futuro desensamblado. Ej.:

   loop: $nop \rightarrow 0000 - 000\#loop$

Los pseudocódigos del 1 al 5 describen el funcionamiento del ensamblador descrito en los cinco puntos anteriores. Las listas vacías son definidas al iniciarlas usando corchetes [ ] y pueden acceder a cualquier elemento usando si índice $lista[i]$. Las tablas hash (o diccionarios) se inicializan con las llaves {} su estructura consiste en una lista de duplas valor y clave $diccionario \leftarrow \{key : valor, ...\}$, para tener acceso a un valor se utiliza la clave como su índice. Las variables, listas o diccionarios que son definidas como globales son accesibles en cualquier función o procedimiento.

Los diccionarios se pueden utilizar como tablas de búsqueda, entre las cuales encontramos el *código de operación* que tiene el nemónico y su valor numérico. El código de operación tiene un límite de 6 bits para un rango binario de b000000 hasta 0b111111 o en hexadecimal de 0x00 a 0x3F; otra tabla de búsqueda es el *modo de direccionamiento*, definido con solo 3 bits, con un rango binario de 0b000 hasta 0b111, o en hexadecimal de 0x0 a 0x7.

En el pseudocódigo 1 se definen variables globales como *Error* que toma el valor de verdad cuando existe un error y se verifica para continuar o salir. El código de operación CO y el modo de direccionamiento MODO son diccionarios que definen un mnemónico con un valor numérico. El nombre del archivo de salida *archivo_out* y el nombre del archivo principal del código fuente *archivo_in*. La lista que almacenara el nombre de todos los archivos del código fuente es *Archivos* ← [main.asm, lib.asm, cte.asm] y el diccionario que almacenara todas las instrucciones asociadas a un archivo es *Programa* ← {main.asm:[*instrucción*$_0, \ldots$],$\ldots$}, existe un solo diccionario para las constantes *Cte* ← {*id*$_0$:*valor*,$\ldots$}, las etiquetas almacena las etiquetas asociadas a un archivo *Labels* ←{main.asm:{*id*$_0$:*dirección*$_0$,$\ldots$},$\ldots$}.

Pseudocódigo 1 Función principal.

```
1: procedimiento PRINCIPAL(archivo_in) ▷ El archivo de entrada es el archivo principal del código fuente.
2:     Archivos ←[archivo_in]                          ▷ Lista global de archivos en lenguaje ensamblador.
3:     Programa ← { }      ▷ Diccionario global que asociando cada archivo con una listas de instrucciones
   representadas por conjuntos de tokens.
4:     Constantes ← { }  ▷ Diccionario global que almacena todas las constantes definidas con la directiva
   #equ.
5:     Etiquetas ← {}      ▷ Diccionario global que asocia cada archivo con un diccionario que almacena la
   tupla (identificador : dirección de la instrucción).
6:     Error← Falso                                    ▷ Inicializada la variable global de error.
7:     CO ←{'nop':0, 'load':1,...}                     ▷ Inicializar el diccionario global de código de operación.
8:     MODO ← {'implícito':0,'inmediato':1,...}        ▷ Inicializar el diccionario global de modos de
   direccionamientos.
9:     archivo_out ← "main.bin"                        ▷ El nombre del archivo del programa en lenguaje de máquina
10:    ENSAMBLADOR()                                   ▷ Llamada al procedimiento del ensamblador
11: fin procedimiento
```

El pseudocódigo 2 describe el procedimiento ENSAMBLADOR() que divide el proceso en dos etapas un análisis con la llamada a la función ANALIZAR() y la generación del programa en código máquina con la llamada a la función GENERAR_BINARIO().

Pseudocódigo 2 Procedimiento del ensamblador.

```
1: procedimiento ENSAMBLADOR()
2:     ANALIZAR(Archivos[0])          ▷ Llamada al procedimiento del analizador léxico y sintáctico.
3:     GENERAR_BINARIO()  ▷ Llamada al procedimiento que genera el programa en lenguaje de máquina.
4: fin procedimiento
```

Pseudocódigo 3 Función que obtiene el conjunto de tokens asociados a una línea del lenguaje ensamblador

```
 1: procedimiento OBTENER_TOKENS(linea, numlinea)   ▷ Cada línea en lenguaje ensamblador generara
    un diccionario de tokens: Línea←{}
 2:     si línea = r': id' entonces                                              ▷ Etiqueta
 3:         Linea['etiqueta']← id          ▷ Un token está compuesto por el tipo y su valor (o lexema).
 4:     fin si                                          ▷ Ej.: token←(etiqueta,identificador).
 5:     si línea = r"load reg_I, int_10" entonces
 6:         Linea['opcode']← 'load'
 7:         Linea['rt']← ENTERO(I,10)                  ▷ Convertir la expresión a un entero de base 10
 8:         Linea['cte']← ENTERO(cte,10)               ▷ Convertir la expresión a un entero de base 10
 9:         Linea['modo']← 'inmediato'
10: ...
11:     en otro caso si línea = r'#equ id, int_16' entonces
12:         Linea['directiva']← 'equ'
13:         Linea['id']← id
14:         Linea['cte']← ENTERO(cte,16)
15:     en otro caso si línea = r'#include "Cadena_caracteres"' entonces
16:         Linea['archivo']← Cadena_caracteres
17:         Linea['directiva']← 'include'
18:     en otro caso si línea = r'#end' entonces
19:         Linea['directiva']← 'end'
20:     en otro caso
21:         Error ← Cierto
22:     fin si
23:     devolver Linea
24: fin procedimiento
```

El pseudocódigo 3 recibe una línea en lenguaje ensamblador y obtiene un conjunto de tokens. Los siguientes ejemplos se consiguieron aplicando el pseudocódigo 3 por cada línea en lenguaje ensamblador mostrado en la figura 5.15 (en la página 74):

```
#include "lib.asm" ->{directiva:'include','cadena:'lib.asm'}
#include "cte.asm" ->{directiva:'include','cadena:'cte.asm'}
call main          ->{opcode:'call',id:main',modo:'directo'}
halt               ->{opcode:'halt',modo:'implicito'}
:main              ->{label:'main'}
load reg0, 0x0     ->{opcode:'load',rt:0,cte:0,modo:'inmediato'}
:loop              ->{label:'loop'}
call inc           ->{opcode:'call',id:'inc',modo:'directo'}
cmp reg0, limite   ->{opcode:'cmp',rt:0,id:'limite',modo:'inmediato'}
jclr status, ZERO  ->{opcode:'jclr',rx:'status',id:'ZERO',modo:'inmediato'}
jmp loop           ->{opcode:'jmp',id:'loop',modo:'directo'}
ret                ->{opcode:'ret',modo:'implicito'}
#end               ->{directiva:'end'}
#include \cte.asm" ->{directiva:'include','cadena:'cte.asm'}
:inc               ->{label:'inc'}
add reg0, step     ->{opcode:'add',rt:0,id:'step',modo:'inmediato'}
```

```
ret                ->{opcode:'ret',modo:'implicito'}
#end               ->{directiva:'end'}
#equ step  ,   1  ->{directiva:'equ',id:'step',cte:1}
#equ limite,  10  ->{directiva:'equ',id:'limite',cte:10}
#equ ZERO  ,   1  ->{directiva:'equ',id:'ZERO',cte:1}
#end              ->{directiva:'end'}
```

Pseudocódigo 4 Procedimiento que realiza el análisis léxico y sintáctico.

```
1: procedimiento ANALIZAR(archivo)
2:     si Programa[archivo] != Nulo entonces        ▷ verifica si el diccionario asociado al archivo es nulo
3:         Programa ← []                                                    ▷ Inicializa la lista vacia.
4:         num_Instrucción ← 0
5:         para linea, num_linea en archivo hacer                  ▷ Itera todas las lineas del archivo.
6:             Linea_Tokens ← OBTENER_TOKENS(linea, num_linea)
7:             si Error = True entonces                          ▷ Error al generar el conjunto de tokens.
8:                 PRINT("Error")
9:                 FIN_PROGRAMA                                  ▷ Terminar la ejecución del ensamblador
10:            en otro caso si Linea_Tokens['etiqueta'] no es Nulo entonces
11:                Etiquetas[archivo][Linea_Tokens['etiqueta']] ← num_Instrucción
12:            en otro caso si Linea_Tokens['directiva'] no es Nulo entonces
13:                si Linea_Tokens['directiva'] = 'include' entonces
14:                    AGREGAR(Archivos, Linea_Tokens['include'])      ▷ Incluir un elemento en la lista.
15:                en otro caso si Linea_Tokens['directiva'] ='equ' entonces
16:                    Constantes[Linea_Tokens['id']] ← Linea_Tokens['cte']
17:                en otro caso si Linea_Tokens['directiva'] ='end' entonces
18:                    SALIR_DEL_BUCLE                                ▷ Sale del bucle para
19:                fin si
20:            en otro caso
21:                Linea_Tokens['nl']← num_linea
22:                Linea_Tokens['ni']← num_Instrucción
23:                AGREGAR(Programa[archivo],Linea_Tokens) ▷ Agregar cada uno de los conjuntos de tokens
       de una instrucción a una lista asociada a un archivo.
24:            fin si
25:        fin para
26:    fin si
27: fin procedimiento
```

El pseudocódigo 4 lee los archivos en lenguaje ensamblador y regresa la siguiente estructura en la que se almacena todas las instrucciones, etiquetas y constantes. Esta función equivale a los puntos uno y dos que describen el proceso del ensamblador. El resultado obtenido es la siguiente estructura:

```
Constantes={'step':1,'limite':10,'ZERO:1}
Etiquetas ={'main.asm':{'loop':2,'main':3},
            'lib.asm':{'inc':0},'cte.asm':{}}
Programa=  { 'cte.asm':[],
'main.asm':[{opcode:'call',id:'main',md:'directo',ni=0,nl=2},
```

```
            {opcode:'halt',md:'implicito', ni=1,nl=3},
            {opcode:'load',rt:0,cte:0,md:'inmediato', ni=2,nl=5},
            {opcode:'call',id:'inc',md:'directo', ni=3, nl=7},
            {opcode:'cmp',rt:0,id:'limite',md:'inmediato', ni=4,nl=8},
            {opcode:'jclr',rx:status,id:'ZERO',md:'inmediato',ni=5,nl=9},
            {opcode:'jmp',id:'loop',md:'inmediato',ni=6, nl=10},
            {opcode:'ret',md:'implicito', ni=7,nl=11}],
'lib.asm':[ {opcode:'add',md:'inmediato',ni=8, nl=2},
            {opcode:'ret',md:'implicito',ni=9, nl=3}] }
```

Pseudocódigo 5 Cambiar el conjunto de tokens de una instrucción por su equivalente en lenguaje máquina

```
 1: procedimiento OBTENER_LINEA_BINARIA(linea_Tokens, archivo)
 2:     linea_Lenguaje máquina ← BINARIA(CO[linea_Tokens['opcode']], 6)   ▷ La función binaria devuelve la
    cadena de caracteres {0,1}^6, el diccionario CO es una tabla de consulta que devuelve el valor numérico
    del modo de direccionamiento.
 3:     si linea_Tokens['opcode']='load' y linea_Tokens['modo']='inmediato' entonces
 4:         linea_Lenguaje máquina +← BINARIO(linea_Tokens['rt'],4)
 5:         linea_Lenguaje máquina +← BINARIO(linea_Tokens['cte'],16)
 6:     en otro caso si linea_Tokens['opcode']='call' y linea_Tokens['modo']='directo' entonces
 7:         linea_Lenguaje máquina +← '0000'
 8:         si linea_Tokens['cte'] no es Nulo entonces
 9:             linea_Lenguaje máquina +← BINARIO(linea_Tokens['cte'],16)
10:         en otro caso
11:             linea_Lenguaje máquina +← BINARIO(Etiquetas[archivo][linea_Tokens['id']],16)
12:         fin si
13:     en otro caso si linea_Tokens['opcode']='add' y linea_Tokens['modo']='inmediato' entonces
14:         linea_Lenguaje máquina +← BINARIO(linea_Tokens['rt'],4)
15:         si linea_Tokens['cte'] no es Nulo entonces
16:             linea_Lenguaje máquina +← BINARIO(linea_Tokens['cte'],16)
17:         en otro caso
18:             linea_Lenguaje máquina +← BINARIO(Constantes[linea_Tokens['id']],16)
19:         fin si
20: ...
21:     en otro caso
22:         Error ← Cierto
23:     fin si
24:     linea_Lenguaje máquina +← BINARIO(MODO[linea_Tokens['modo']],3)
25:     devolver linea_Lenguaje de máquina
26: fin procedimiento
```

El pseudocódigo 5 modifica un conjunto de tokens a su equivalente a una cadena de caracteres binarios, se utilizan los corchetes para separar los bloques de caracteres [opcode][argumento][modo], el guion es indistinto 0 o 1, por ejemplo:

[011111] [- - - -][0000000000000010] [010] ⇔ {op:call,dir:2,...}

El pseudocódigo 6 almacena todos las líneas en lenguaje máquina en un archivo de salida, los puntos 3, 4 y 5 que describen el proceso del ensamblador corresponden a esta función.

Usando este pseudocódigo se obtienen cada una de las líneas del programa en lenguaje de máquina:

```
[011111] [----0000000000000010] [010] :{op:call,dir:2,...}
[100100] [--------------------] [000] :{op:halt,...}
[000001] [0000000000000000000] [001] :{op:load,rt:0,cte:0...}
[011111] [----0000000000001000] [010] :{op:call,dir:8,...}
[000110] [0000000000000001010] [001] :{op:cmp,rt:0,cte:10,...}
[000101] [11111111-------00001] [001] :{op:jclr,rt:256,n:1,...}
[000011] [----0000000000000011] [010] :{op:jmp,dir:3,...}
[100000] [--------------------] [000] :{op:ret,...}
[000111] [0000000000000000001] [001] :{op:add,rt:0,cte:1...}
[100000] [--------------------] [000] :{op:ret,...}
```

Pseudocódigo 6 Función que retorna el programa en lenguaje máquina

```
 1: procedimiento GENERAR_BINARIO()
 2:     OffSet ← 0
 3:     para archivo en Archivos hacer
 4:         para linea_i en archivo hacer
 5:             linea_m ← OBTENER_LINEA_BINARIA(linea_i)
 6:             ESCRIBIR(archivo_out, linea_m)
 7:             si Error = True entonces          ▷ Error al generar el programa en lenguaje máquina.
 8:                 PRINT("Error")
 9:                 FIN_PROGRAMA                  ▷ Terminar la ejecución del ensamblador
10:             fin si
11:         fin para
12:         OffSet ← OffSet + LONGITUD(Programa[archivo])
13:     fin para
14: fin procedimiento
```

Los pasos seguidos por el desensamblador son:

1. Leer línea por línea del archivo que contiene las instrucciones en lenguaje máquina.
2. En cada línea se tiene una instrucción máquina o cadena binaria que se divide en un conjunto de bits y cada conjunto se intercambia a su equivalente mnemónico, formando las instrucciones en lenguaje ensamblador.
3. Los valores numéricos que representan etiquetas se almacenan en una colección, para generar las etiquetas correspondientes.
4. Todas las líneas en lenguaje ensamblador se ponen en un archivo con extensión $*.dasm$.

Los pasos para generar una memoria ROM descrita en VHDL son:

1. Leer línea por línea del archivo que contiene las instrucciones en lenguaje máquina, para verificar la longitud fija de 29 bits de la instrucción.

2. En el orden que fue leída cada instrucción, se asocia cada una con una localidad de memoria y se rellena la estructura WHEN-ELSE:

$$
\begin{aligned}
salida \leftarrow & instrucción_0 \textbf{ WHEN } && dir = dirección_0 \textbf{ ELSE} \\
& instrucción_1 \textbf{ WHEN } && dir = dirección_1 \textbf{ ELSE} \\
& \ldots \\
& instrucción_{k-1} \textbf{ WHEN } && dir = dirección_{k-1} \textbf{ ELSE} \\
& insrucción_k;
\end{aligned}
$$

3. Poner en la arquitectura (comportamiento) de la ROM la estructura WHEN ELSE obtenida en el paso anterior.

Finalmente, la memoria de programa generada se muestra en el código 5.1, la figura 5.18(b) muestra el código fuente desensamblado del archivo binario de la figura 5.18(a), los valores binarios representados por un guión son indiferentes, no tienen uso.

Código 5.1: Memoria de programa

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity memoria_programa is
    port( dir:    in std_logic_vector( 15 downto 0);
          q   :   out std_logic_vector( 28 downto 0));
end entity memoria_programa;
architecture arq_memoria_programa of memoria_programa is begin
q<="011111----00000000000000010010" when dir=x"0000" else
   "100100--------------------000" when dir=x"0001" else
   "00000100000000000000000000001" when dir=x"0002" else
   "011111----00000000000001000010" when dir=x"0003" else
   "00011000000000000000001010001" when dir=x"0004" else
   "00010111111111-------00001001" when dir=x"0005" else
   "000011----00000000000000011010" when dir=x"0006" else
   "100000--------------------000" when dir=x"0007" else
   "00011100000000000000000001001" when dir=x"0008" else
   "100000--------------------000" when dir=x"0009" else
   "00000000000000000000000000000";
end architecture arq_memoria_programa;
```

```
0 011111----00000000000000010010
1 100100--------------------000
2 00000100000000000000000000001# label: main
3 011111----00000000000001000010# label: loop
4 00011000000000000000001010001
5 00010111111111-------00001001
6 000011----00000000000000011010
7 100000--------------------000
8 00011100000000000000000001001# label: inc
9 100000--------------------000
```

(a)

```
00      call main
01      halt
02:main load reg0  , 0
03:loop call inc
04      cmp reg0   ,10
05      jclr status, 1
06      jmp loop
07      ret
08:inc  add reg0   , 1
09      ret
10      #end
```

(b)

Figura 5.18: Programa en lenguaje de máquina y desensamblado.

Capítulo 6

Simulaciones y Resultados

A lo largo de este capítulo se muestran los pseudocódigos asociados a cada prueba, los cuales son equivalentes a los programas en lenguaje ensamblador utilizados para el banco de pruebas que valida el diseño del microcontrolador. Con el ensamblador explicado en el capítulo anterior, se crearon los programas para generar: (1) un reloj, (2) un modulador por ancho de pulso, (3) un puerto serial, (4) un ejemplo del perro guardián, (5) un multiplicador Karatsuba de dos números de 128 bits, (6) el estándar de cifrado por bloques AES, y (7) dos modos de operación CCM y GCM de encriptación autenticada. A continuación se describirán en detalle cada uno de estos programas, con los resultados de uso de memoria, reporte de tiempos y número de ciclos por byte para el cifrado y descifrado para la aplicación (7).

Todos los banco de pruebas presentados realizan una serie de pasos repetitivos que siguen un orden secuencial, cada paso se describe como: (a) inicializar las localidades de memoria, arreglos y variables; (b) configurar los periféricos con o sin el manejador de interrupciones; (c) ejecución de un bucle infinito o llamada a una subrutina que se desea probar; y (d) todas las subrutinas que auxilian el flujo del programa. A continuación cada una de las siete pruebas se describierán en detalle.

6.1. Reloj

Esta aplicación consiste en un reloj que despliega horas, minutos y segundos en formato binario. Usa los puertos paralelos B, C y D para desplegar los datos de horas, minutos y segundos, respectivamente; y se configura a través del puerto paralelo A. Las pseudocódigos 7 y 8 describen el programa realizado en ensamblador para esta prueba. La subrutina Principal() desarrolla la aplicación, primero realiza una llamada a la subrutina Inicializar() que configura al microcontrolador como reloj; luego inicia un bucle infinito para desplegar los contadores; en cada ciclo del bucle se verifica si el bit dos del puerto A esta habilitado para reiniciar a cero los contadores.

La variable 'flag' controla el estado del reloj: en estado alto (1) funciona la aplicación en

modo normal, en estado bajo (0) está en modo edición. Las variables que realizan el conteo de los segundos ('seg'), minutos ('min') y horas ('hrs') se inicializa en cero.

El temporizador es el componente del microcontrolador que en este caso se usa para contar segundos. Los registros del temporizador son: 'dfTimer' que establece la velocidad del reloj que alimentara al temporizador, 'timer0' establece el límite superior del contador (bucle infinito de cero a $timer0 - 1$) y el último registro 'cfgTimer' habilita el temporizador cero, la tabla 6.1 describe el uso de los bits para este registro, los primeros cuatro bits en alto (1) habilitan los temporizadores, los siguientes cuatro bits apagan la llamada de interrupción con el bit en alto (1).

Bit	Descripción	Bit	Descripción	Temporizador
0	Habilita	4	Apaga la interrupción	0
1	Habilita	5	Apaga la interrupción	1
2	Habilita	6	Apaga la interrupción	2
3	Habilita	7	Apaga la interrupción	3

Tabla 6.1: Descripción del registro 'cfgTimer' que afecta el funcionamiento de los distintos temporizadores.

Los registros 'TrisX', en la subrutina Inicializar configura el puerto paralelo de entrada o salida. Por defecto es de entrada al reiniciarse por lo que toma el valor de 0x0000. Para cambiar un pin como de salida se tiene que cambiar a '1' el valor del registro 'TrisX'. Los puertos B, C y D ('TrisB'←'TrisC'←'TrisD'←x0000) son de salida, el byte menos significativo del puerto A es de entrada y el byte más significativo de salida (esto se configura cuando 'TrisA' toma el valor 0xFF00).

La tabla 4.20, en la página 52, indica el número de bit que también se asocia en el registro de enmascaramiento 'MSKINT'. Se configura el vector de interrupciones almacenando en los registros 'dirintT0', 'dirintE0', ... y 'dirintE3' las subrutinas de atención a interrupciones. Cada una de estas subrutinas son las que incrementa los segundos, minutos y horas. Al final se inicializa el registro de configuración 'cfg0' con 0x0001 que habilita la bandera de interrupciones, permitiendo la ejecución de las subrutinas asociadas a las interrupciones.

La subrutina ModificarX atienden las interrupciones externas que modifican a los contadores 'hrs', 'min' y 'seg', incrementando o decrementando su valor por medio de la subrutina ModificarCtr. La interrupción externa cero llama a la función ModificarFlag que habilita el modo normal o el modo edición del reloj con el fin de modificar los contadores, la variable 'flag' permite saber en qué modo se encuentra el reloj. El funcionamiento normal del reloj se ejecuta con la subrutina Reloj, evalúa el contador de segundos y minutos en el intervalo [0-59], y horas en el intervalo [0-23].

La tabla 6.2 muestra la especificación de la interfaz del microcontrolador con el banco de prueba del reloj. La interfaz se muestra en la figura 4.4, mostrada en la página 43.

La gráficas de onda se muestra en la figura 6.1. Para esta simulación se establece el ciclo de reloj de 1 ns para la señal CLK del microcontrolador.

PIN	Descripción	PIN	Descripción
a[0]	incrementa/decrementa	a[1]	reiniciar reloj
a[2..7]	sin uso	a[8]	apagar int_ext[0]
a[9]	apagar int_ext[1]	a[10]	apagar int_ext[2]
a[11]	apagar int_ext[3]	a[12..15]	sin uso
b[0..15]	muestra los segundos	c[0..15]	muestra los minutos
d[0..15]	muestra las horas	int_ext[0]	habilita la bandera 'flag'
int_ext[1]	modifica segundos	int_ext[2]	modifica minutos
int_ext[3]	modifica horas	int_ext[4..5]	sin uso

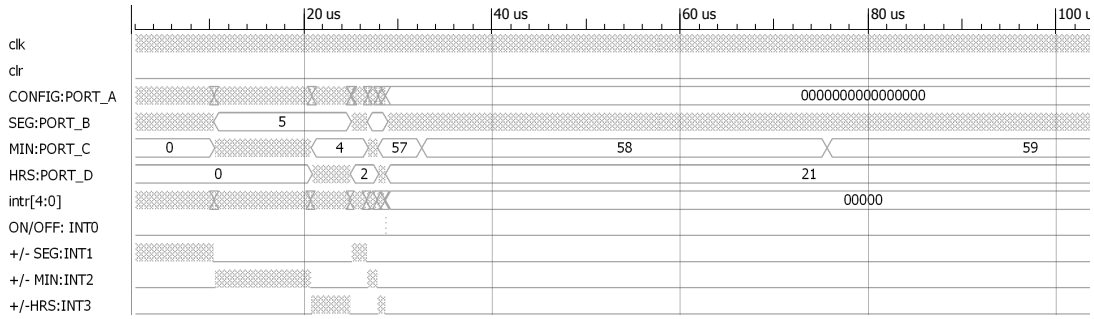
Tabla 6.2: Especificación de la interfaz del microcontrolador para el reloj.

Pseudocódigo 7 Principal, inicializar y modificar contador

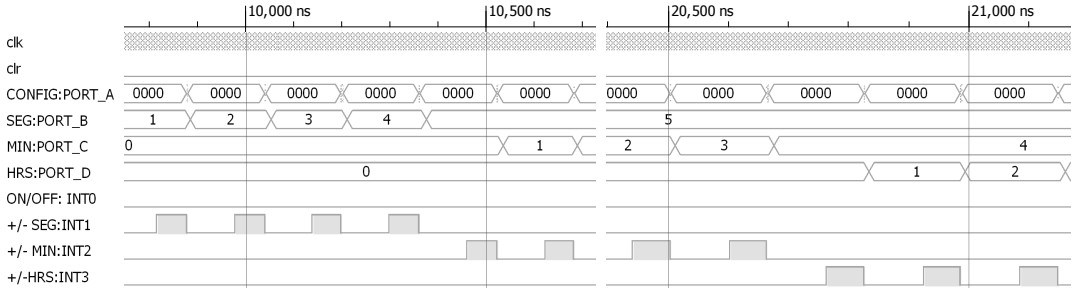
```
 1: procedimiento PRINCIPAL
 2:     INICIALIZAR( )
 3:     mientras Cierto hacer
 4:         si (PuertoA&0x0002)!=0) entonces
 5:             hrs←min←seg← 0,
 6:         fin si
 7:         PortB←seg, PortC←min, PortD←hrs
 8:     fin mientras
 9: fin procedimiento
10: procedimiento INICIALIZAR(a)
11:     flag ← Falso
12:     hrs ← min ← seg ← 0
13:     dfTimer ← 0x0002
14:     timer0 ← 0x0120
15:     cfgTimer ←0x0001
16:     TrisA ← 0xFF00
17:     TrisB ←TrisC←TrisD←0x0000
18:     dirintT0 ←RELOJ
19:     dirintE0 ←MODIFICARFLAG
20:     dirintE1 ←MODIFICARSEG
21:     dirintE2 ←MODIFICARMIN
22:     dirintE3 ←MODIFICARHRS
23:     MSKINT ← 0x3C04
24:     cfg0 ← 0x0001
25: fin procedimiento
26: procedimiento MODIFICARCTR(var,lim)
27:     si flag = Cierto entonces
28:         si (PortA & 0x0001) =0 entonces
29:             var←var+1
30:             var ← (var≥lim)? 0 : var
31:         en otro caso
32:             var←var-1
33:             var ←(var<0)? lim-1: var
34:         fin si
35:     fin si
36: fin procedimiento
```

Pseudocódigo 8 Atencion a interrupcines

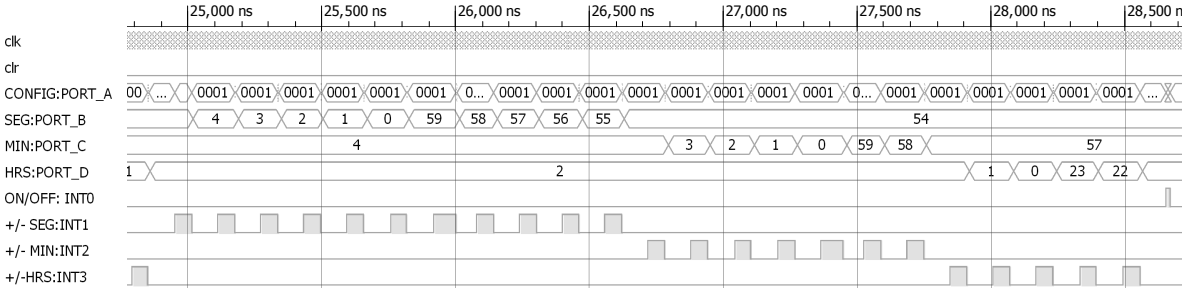
```
 1: procedimiento MODIFICARX()  ▷ MODIFICARSEG(), MODIFICARMIN() y MODIFICARHRS()
 2:     var, lim ← (hrs,60) o (min,60) o (seg,24)  ▷ Depende de la subrutina
 3:     MODIFICARCTR(var, lim)
 4:     tmp0 ← PortA      ▷ Apagar interrupción
 5:     tmp1 ← PortA | MASCARAX
 6:     PortA← tmp1
 7:     PortA← tmp0
 8: fin procedimiento
 9: procedimiento MODIFICARFLAG()
10:     si flag = Falso entonces
11:         flag ← Cierto
12:     en otro caso
13:         flag ← Falso
14:     fin si
15:     tmp0 ← PortA      ▷ Apagar interrupción
16:     tmp1 ← PortA | 0x0100
17:     PortA← tmp1
18:     PortA← tmp0
19: fin procedimiento
20: procedimiento RELOJ()
21:     si flag = Falso entonces
22:         seg←seg+1
23:         si seg≥60 entonces
24:             seg←0, min←min+1
25:             si seg≥60 entonces
26:                 min←0, hrs←hrs+1
27:                 si hrs≥24 entonces
28:                     hrs←0
29:                 fin si
30:             fin si
31:         fin si
32:     fin si
33: fin procedimiento
```



(a) Se configura la hora (21:57:54) y se activa el modo normal del reloj.



(b) Modificar la hora incrementando los contadores



(c) Modificar la hora decrementando los contadores

Figura 6.1: Gráficas de ondas del reloj.

6.2. Modulación de ancho de pulso

Esta prueba consiste en comparar dos señales, una de entrada contra una señal tipo sierra, cuando la señal de entrada sea mayor a la de diente de sierra pone en alto la salida.

El pseudocódigo 9 selecciona uno de los cuatro puertos paralelos para la modulación de ancho de pulso, la subrutina del pseudocódigo 10 inicializa este módulo. El reloj de la señal sierra se establece con el registro 'limdivPwm'. La señal sierra toma los valores enteros en el rango de 0x0000 a 0x00FF; el límite se establece con el registro 'limSierraPwm'. La configuración del registro 'configPwm' se describe en la tabla 6.3, la configuración asociada para seleccionar los puertos se muestra en la tabla 6.4.

El bucle de la subrutina PRINCIPAL() realiza un retardo con la subrutina ESPERAR(), cuando se termina esta demora se modifica el puerto con la variable 'Npuerto'. La interfaz del microcontrolador se muestra en la tabla 6.5. Las gráficas de ondas del banco de prueba del PWM se muestran en las figuras 6.2 y 6.3 .

BIT	Descripción
0	Habilita PWM
1	Salida negada
2,3	Selector de puerto

Tabla 6.3: Registro 'configPwm'

Selector	Puerto
00	A
01	B
10	C
11	D

Tabla 6.4: Selector de puerto paralelo

PIN	Descripción	E-S
A[1..15]	Señal 0	E
B[1..15]	Señal 1	E
C[1..15]	Señal 2	E
D[1..15]	Señal 3	E
PWM_{out}	PWM	S

Tabla 6.5: Interfaz del microcontrolador

Pseudocódigo 9 Principal

```
 1: procedimiento PRINCIPAL()
 2:     INICIALIZAR()
 3:     mientras Cierto hacer
 4:         ESPERAR()
 5:         CAMBIARPUERTO()
 6:     fin mientras
 7: fin procedimiento
 8: procedimiento CAMBIARPUERTO()
 9:     Npuerto ← Npuerto+1
10:     si Npuerto ≥ 4 entonces
11:         Npuerto←0
12:     fin si
13:     tmp0 ← Npuerto ≪2
14:     tmp1← configPwm ∧ 0xFFF3
15:     configPwm ← tmp0 | tmp1
16: fin procedimiento
```

Pseudocódigo 10 Inicializar y esperar

```
 1: procedimiento INICIALIZAR()
 2:     Npuerto←3
 3:     Contador←0
 4:     trisA←trisB← trisC←trisD←0x0000
 5:     limdivPwm ← 0x0007
 6:     limSierraPwm ← 0x000FF
 7:     configPwm ← 0x000D
 8: fin procedimiento
 9: procedimiento ESPERAR()
10:     mientras contador≥32 hacer
11:         NOP
12:     fin mientras
13: fin procedimiento
```

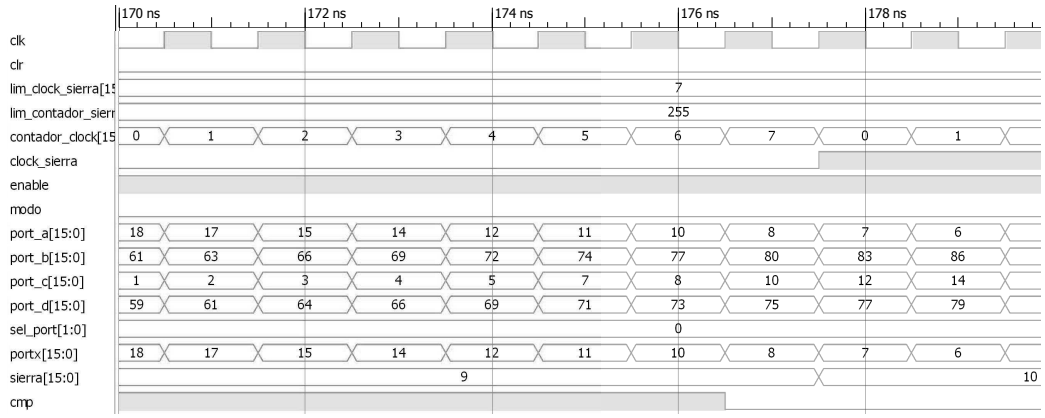


Figura 6.2: Gráfica de onda de las señales del componente PWM

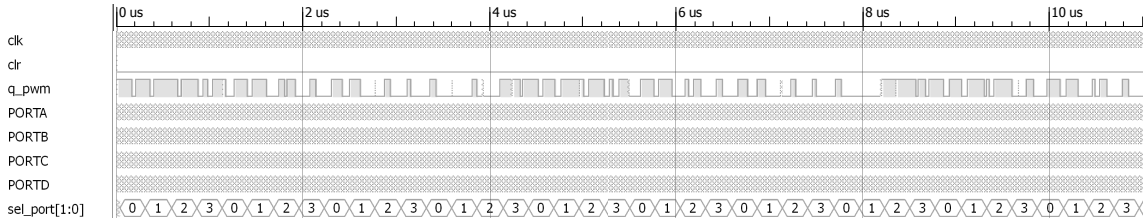


Figura 6.3: Gráfica de onda de la interfaz del microcontrolador.

6.3. Puerto serial

La prueba que se realizó fue conectar a dos microcontroladores a través del puerto serial, ambos microcontroladores comparten la misma interfaz que se ve en la tabla 6.6, los pines de la comunicación son: Tx1→Rx0 y Tx0→Rx1.

PIN	Descripción	E-S
A[15..0]	Configuración	E
C[15..0]	Dato transmitido	S
D[15..0]	Dato recibido	S
Tx	Transmisor serial	S
Rx	Receptor serial	E

Tabla 6.6: Interfaz de microcontrolador para la comunicación serial

El puerto paralelo A selecciona uno de los dos mensajes de salida, el puerto C muestra el byte transmitido y el puerto D muestra el byte recibido. El identificador del mensaje 'IDMsg' se evalúa para seleccionar que información transmitir, la subrutina descrita en el pseudocódigo 13 realiza la carga del mensaje correcto. La función PRINCIPAL() del pseudocódigo 11 verifica que mensaje se debe cargar en el arreglo 'MsgOut'.

Pseudocódigo 11 Principal

```
1: procedimiento PRINCIPAL()
2:     INICIALIZAR()
3:     ESTABLECERMSG()
4:     mientras Cierto hacer
5:         si IDMsg != portA entonces
6:             IDMsg ← portA
7:             ESTABLECERMSG()
8:         fin si
9:     fin mientras
10: fin procedimiento
```

Pseudocódigo 12 Inicializar

```
1: procedimiento INICIALIZAR()
2:     i_out ← i_in ← 0
3:     trisA ← 0x0000
4:     IDmsg ← portA
5:     trisC ← trisD ← 0xFFFF
6:     mskint ← 0x0003
7:     dirintTx ← TRANSMISOR
8:     dirintRx ← RECEPTOR
9:     cfgUart ← 100
10:    cfg0 ← 0x0007
11: fin procedimiento
```

Pseudocódigo 13 Cargar mensaje

```
1: procedimiento ESTABLECERMSG()
2:     MsgOut←((IDmsg & 0x0001) = 0)? ['Hola-mundo-cruel'] : ['-Hi-cruel-world-']
3:     IDmsg ← ((IDmsg & 0x0001) = 0)? 0 : 1          ▷ Operador ternario ?:
4: fin procedimiento
```

Pseudocódigo 14 Transmisión

```
1: procedimiento TRANSMISOR
2:     mientras Cierto hacer
3:         si (status&0x1000)!=0 entonces
4:             portC←txUart←Msg[i_out]
5:             i_out ← i_out+1
6:             i_out ←(i_out≥ 16)? 0 : i_out
7:             Salir del bucle
8:         fin si
9:     fin mientras
10: fin procedimiento
```

Pseudocódigo 15 Recepción

```
1: procedimiento RECEPTOR
2:     mientras Cierto hacer
3:         si (status&0x2000)!=0 entonces
4:             Msg[i_in]←portD← rxUart
5:             i_in ← i_in+1
6:             i_in ←(i_in≥ 16)? 0 : i_in
7:             Salir del bucle
8:         fin si
9:     fin mientras
10: fin procedimiento
```

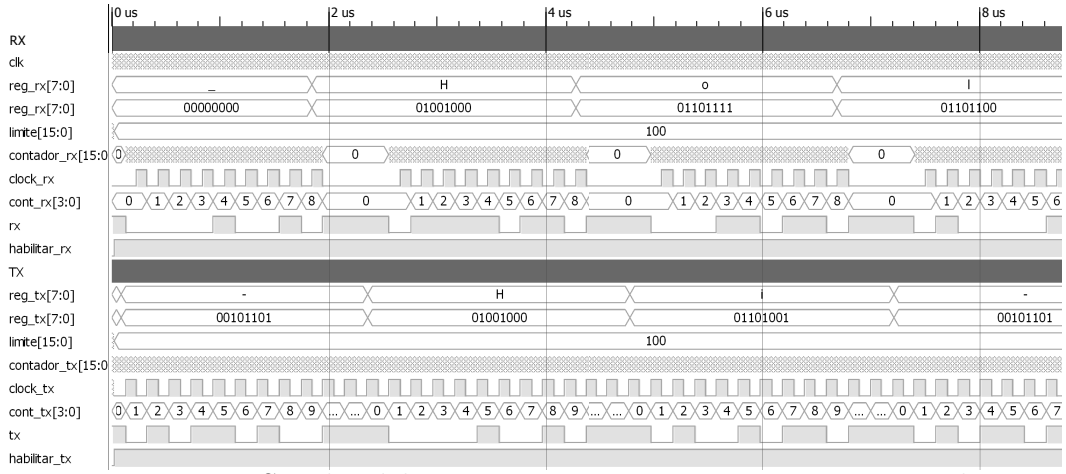
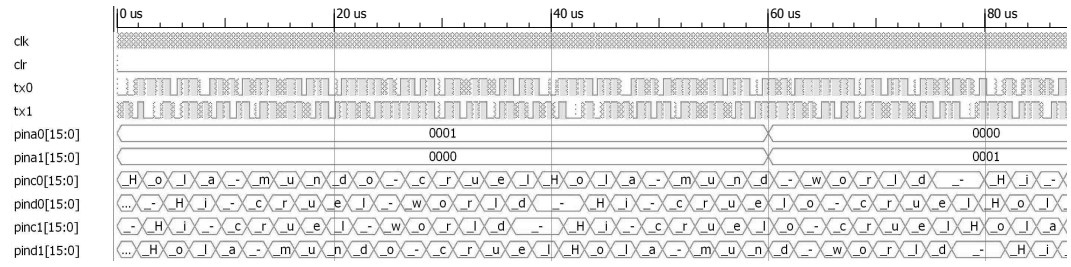
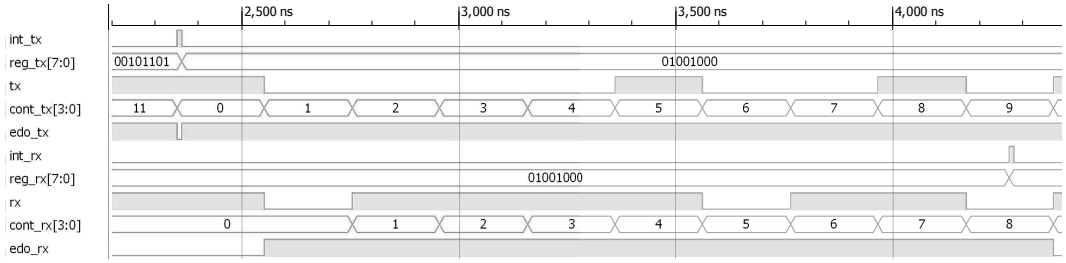


Figura 6.4: Señales del componente transmisor y receptor serial.



(a) Interfaz del microcontrolador



(b) Resumen de señales asociadas a Tx y Rx

Figura 6.5: Gráfica de ondas de la comunicación serial

La subrutina que inicializa todos los componentes se muestran en el pseudocódigo 12, el reloj de la comunicación serial se configura con el registro 'cfgUart' que establece el límite del contador del divisor de frecuencia. Para habilitar la transmisión, recepción e interrupciones se establece 0x0007 en el registro 'cfg0'. La subrutinas que atienden las interrupciones de transmisión y recepción se ven en el pseudocódigo 14 y 15. Las figuras 6.4 y 6.5 muestran las señales de onda del banco de prueba de la comunicación serial.

6.4. Perro guardián

El temporizador guardián, o perro guardián, es un contador que se debe reiniciar cada determinado tiempo, cuando está activado. En el pseudocódigo 16 dentro del bucle se reinicia el contador con la instrucción CLRWDG; cuando se sale del bucle no hay manera de evitar que el perro guardián reinicie todos los componentes del microcontrolador. La instrucción que inicializa el perro guardián se ve en el pseudocódigo 17. La tabla 6.7 describe la interfaz del microcontrolador, los puertos muestran el valor de las variables y el estado de la prueba.

Pseudocódigo 16 Principal

```
1: procedimiento PRINCIPAL()
2:     INICIALIZAR()
3:     PortC← contador ← 0
4:     PortA← 0xFFFF
5:     PortB← 10
6:     mientras contador< 10 hacer
7:         NOP, NOP, NOP, NOP
8:         CLRWDG
9:         PortC←contador ← contador+1
10:    fin mientras
11:    PortA←PortC← 0x0000
12: fin procedimiento
```

Pseudocódigo 17 Inicializar

```
1: procedimiento INICIALIZAR()
2:     TrisA←TrisB←TrisC←0x0000
3:     PortA←PortB←PortC←0x0000
4:     limpg←15
5:     cfg0←0x0008 ∨ cfg
6: fin procedimiento
```

PIN	Descripción	E-S
A[15..0]	Duración del bucle	S
B[15..0]	Limite del contador	S
C[15..0]	Contador	S

Tabla 6.7: Interfaz del microcontrolador para la prueba del perro guardián.

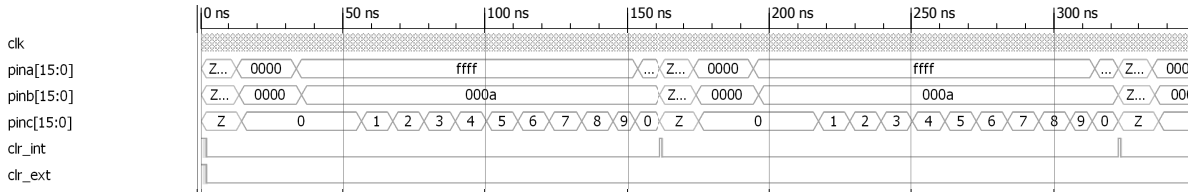


Figura 6.6: Gráfica de onda de la interfaz del microcontrolador

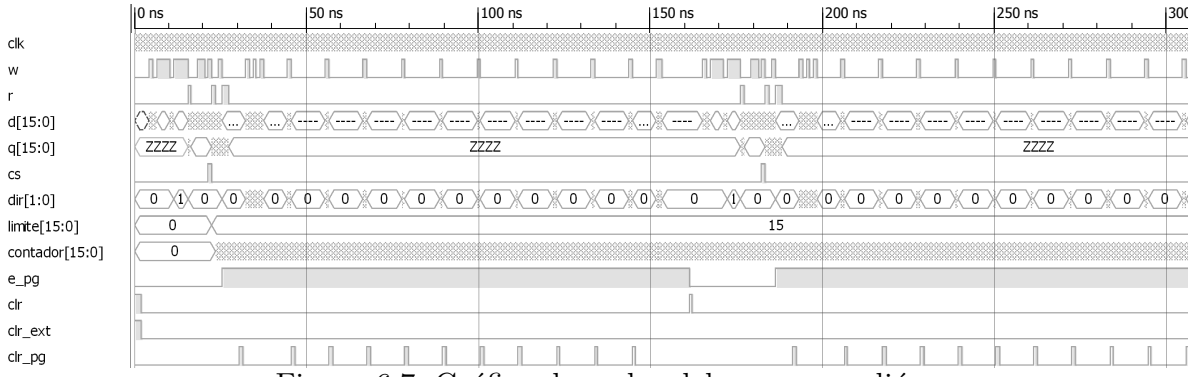


Figura 6.7: Gráfica de ondas del perro guardián

Las gráfica de ondas del banco de pruebas del perro guardián se ve en las figuras 6.6 y 6.7.

6.5. Multiplicador Karatsuba

La operación básica es la multiplicación polinomial de dos enteros de 16 bits. Como un ejemplo en la figura 6.9 se muestra la multiplicación de dos enteros de 4 bits (a[3..0] y b[3..0]); el resultado es un entero de siete bits (c[6..0]), se necesita 16 celdas para realizar la operación. La celda para este ejemplo se distingue en la figura 6.8.

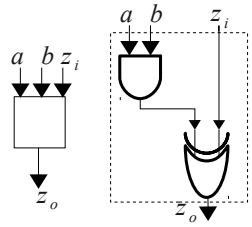


Figura 6.8: Celda del multiplicador, $z_o \leftarrow (a \text{ AND } b) \text{ XOR } z_i$.

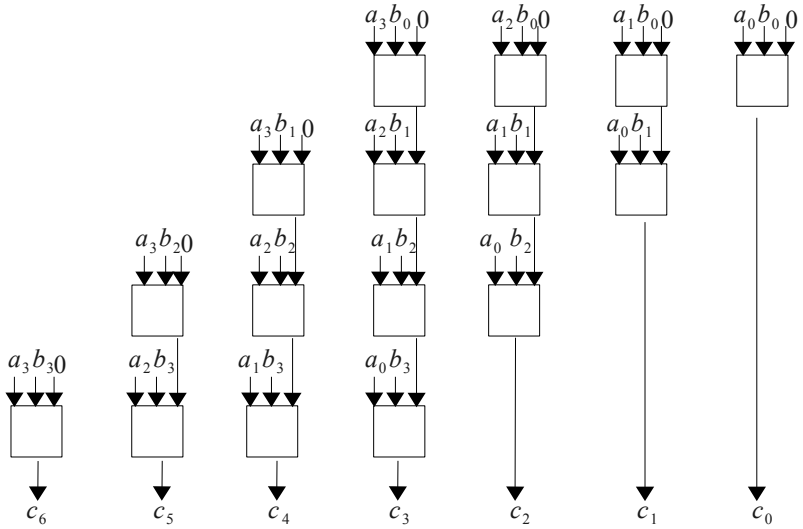


Figura 6.9: Multiplicación (n^2 operaciones) con $n = 4$

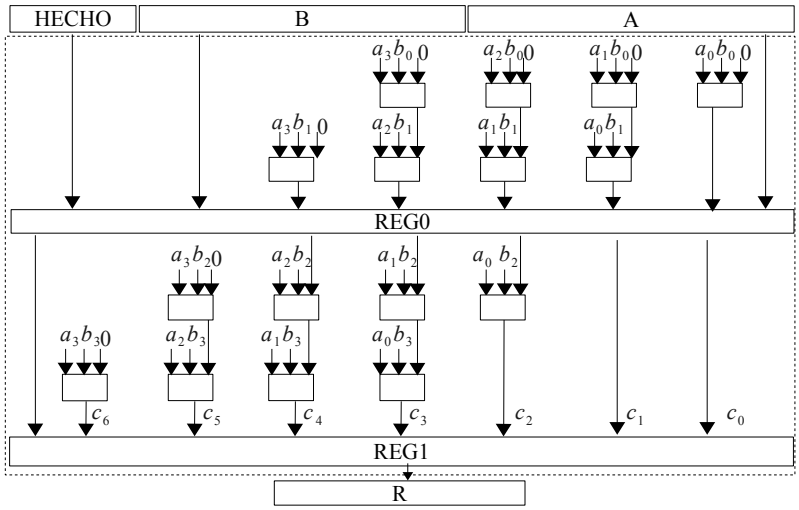


Figura 6.10: Multiplicación con *pipeline*.

Una posible manera de reducir la frecuencia máxima de la multiplicación es realizar las operaciones por segmentos o pipeline, la figura 6.10 muestra la operación con dos segmentos, la primer parte operan los primeros dos bits del entero $b[1..0]$ y la segunda parte opera con los bits restante $b[3..2]$. El registro de configuración activa un bit 'HECH0' que habilita la escritura de los registros de *pipeline* 'REG0', 'REG1' y 'R', mientras sea uno, el flujo de los datos se almacena, en otro caso conserva el dato por defecto o el de una operación anterior.

El algoritmo Karatsuba-Ofman es un método recursivo con el paradigma de divide y vencerás, permite calcular el producto de dos números grandes usando tres multiplicaciones más pequeñas, más algunas sumas y desplazamientos. El procedimiento estándar para multiplicar dos números de n dígitos requiere n^2 operaciones, en contraposición a Karatsuba que requiere a como máximo $3n^{log_2 3}$ operaciones.

Sea A(x) y B(x) dos elementos en $F(2^m)$, donde nos interesa encontrar el producto polinomial D(x)=A(x)B(X) con el grado $\leq 2m$-2. Ambos enteros pueden ser representados en su forma polinomial:

$A \leftarrow x^{\frac{m}{2}}(x^{\frac{m}{2-1}}a_{m-1} + \cdots + a_{\frac{m}{2}}) + (x^{\frac{m}{2-1}}a_{\frac{m}{2-1}} + \cdots + a_0) \leftarrow x^{\frac{m}{2}}A_H + A_L$

$B \leftarrow x^{\frac{m}{2}}(x^{\frac{m}{2-1}}b_{m-1} + \cdots + b_{\frac{m}{2}}) + (x^{\frac{m}{2-1}}b_{\frac{m}{2-1}} + \cdots + b_0) \leftarrow x^{\frac{m}{2}}B_H + B_L$

Los pasos para realizar la multiplicación de dos enteros con un número par de bits y obtener el resultado D por el método Karatsuba [32] son:

$$A \rightarrow (A_H \ll \frac{m}{2}) + A_L \tag{6.1}$$

$$B \rightarrow (B_H \ll \frac{m}{2}) + B_L \tag{6.2}$$

$$m_0 \leftarrow A_L \times B_L \tag{6.3}$$

$$m_1 \leftarrow [A_L \oplus A_H] \times [B_L \oplus B_H] \tag{6.4}$$

$$m_2 \leftarrow A_H \times B_H \tag{6.5}$$

$$D \leftarrow (m_2 \ll n) \oplus [(m_2 \oplus m_1 \oplus m_0) \ll (\frac{n}{2})] \oplus m_0 \tag{6.6}$$

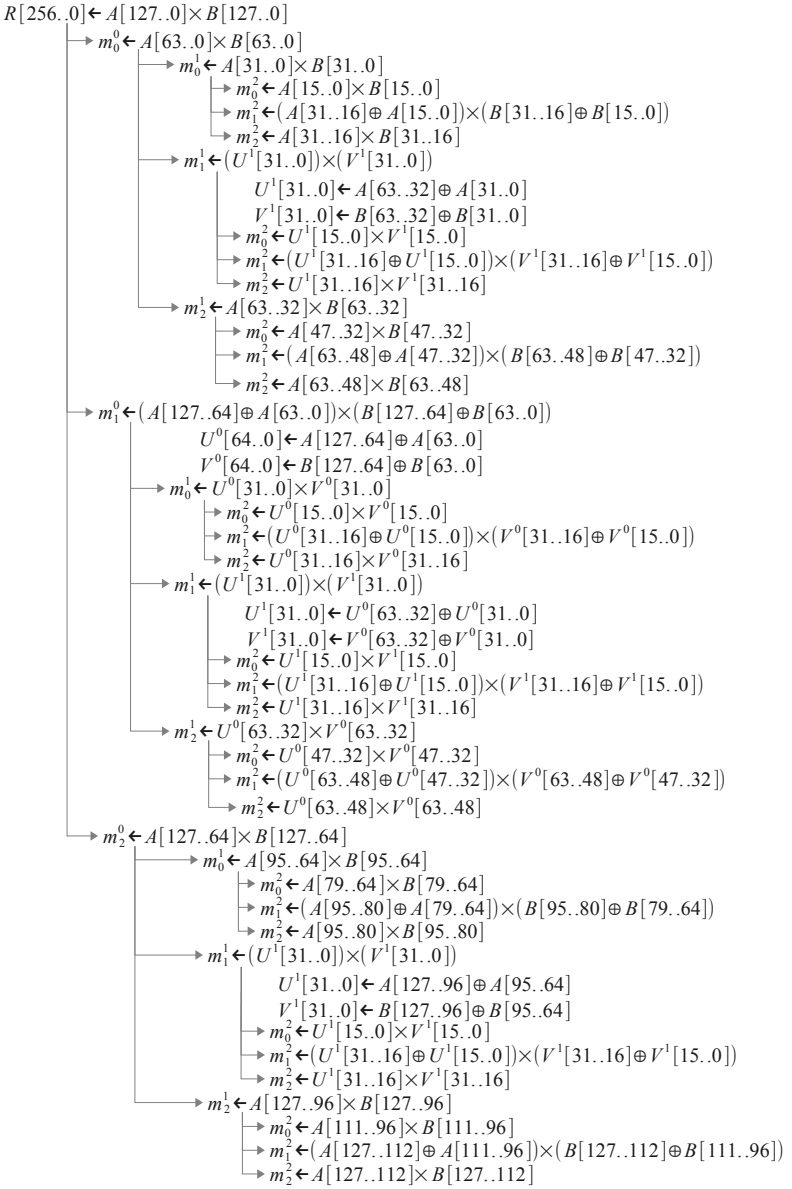


Figura 6.11: Arbol de llamadas para la multiplicación ($\times$), $mul_{256} : R$, $mul_{128} : m_x^0$, $mul_{64} : m_x^1$ y $mul_{32} : m_x^2$. Se omite la llamada a mul_{16}.

El pseudocódigo 18 utiliza claramente la recursividad, para evitar la recursividad se necesita una subrutina que maneje la multiplicación de 128 por 128 bits $mul128$ que llame a otra subrutina que multiplique enteros de 64 por 64 bits $mul64$, que a su vez llame a otra subrutina que multiplique enteros de 32 por 32 bits $mul32$, que al final llame a una subrutina que multiplique enteros de 16 por 16 bits $mul16$. La subrutina $mul16$ hará uso de un componente del microcontrolador que realice la multiplicación polinomial con *pipeline* el resultado será un entero de 32 bits, el último bit por defecto es cero. La figura 6.11 muestra las llamadas de las subrutinas para obtener el resultado de la multiplicación usando Karatsuba.

Pseudocódigo 18 Multiplicación utilizando Karatsuba

Entrada: A$[a_0, a_1, ..., a_{n-1}]$,B$[b_0, b_1, ..., b_{n-1}]$ y n es par
Salida: D$[d_0, d_1, ..., d_{2*n-1}] \leftarrow$ A $\times$ B

función MUL(A,B,n)
  $w \leftarrow \frac{n}{2}$
  A_H ,$A_L \leftarrow$ A; B_H ,$B_L \leftarrow$ B
  $U \leftarrow A_H \oplus A_L$; $V \leftarrow B_H \oplus B_L$
  $m_0 \leftarrow$ MUL(A_L,B_L,w); $m_1 \leftarrow$ MUL(U,V,w); $m_2 \leftarrow$ MUL(A_H,B_H,w)
  $midle \leftarrow m_0 \oplus m_1 \oplus m_2$
  $D \leftarrow (m_2 \ll n) \oplus (midle \ll w) \oplus m_0$
  devolver D
fin función

La reducción modular se describe en [33] transforma el entero de 256 bits obtenido de multiplicar dos enteros de 128 bits, a uno de 128 bits. Esta reducción modular $F(2^{128})$ utiliza el pentanomio irreducible $p(x) = x^{128}+x^7+x^2+x+1$. El pseudocódigo 19 muestra la función realizada para conseguir un entero de 128 bits.

Pseudocódigo 19 Reducción de $D \in F(2^{256}) \rightarrow Q \in F(2^{128})$

Entrada: Di entero de 256 bits
Salida: Do entero de 128 bits

función REDUCCIÓN(D)
  $Q_{127:14} \leftarrow D_{127:14} \oplus D_{255:142} \oplus D_{254:141} \oplus D_{253:140} \oplus D_{248:135}$
  $Q_{13:9} \leftarrow D_{13:9} \oplus D_{141:137} \oplus D_{140:136} \oplus D_{139:135} \oplus D_{134:130} \oplus D_{255:251}$
  $Q_{8:7} \leftarrow D_{8:7} \oplus Di_{136:135} \oplus D_{135:134} \oplus D_{134:133} \oplus D_{129:128} \oplus D_{250:249}$
  $Q_{6:4} \leftarrow D_{6:4} \oplus D_{134:132} \oplus D_{133:131} \oplus D_{132:130} \oplus D_{255:253} \oplus D_{254:252} \oplus D_{253:251}$
  $Q_{3:2} \leftarrow D_{3:2} \oplus D_{131:130} \oplus D_{130:129} \oplus D_{129:128} \oplus D_{252:251} \oplus D_{251:250} \oplus D_{250:249} \oplus D_{255:254}$
  $Q_1 \leftarrow D_1 \oplus D_{129} \oplus D_{128} \oplus D_{250} \oplus D_{249} \oplus D_{254}$
  $Q_0 \leftarrow D_0 \oplus D_{128} \oplus D_{249} \oplus D_{254} \oplus D_{255}$
  devolver Q
fin función

Los 128 bytes menos significativos	Los 128 bytes mas significativos
$D_0 \leftarrow$MD$[dir_D + 0] \leftarrow$D[15..00]	$D_8 \leftarrow$MD$[dir_D + 8] \leftarrow$D[143..128]
$D_1 \leftarrow$MD$[dir_D + 1] \leftarrow$D[31..16]	$D_9 \leftarrow$MD$[dir_D + 9] \leftarrow$D[159..144]
$D_2 \leftarrow$MD$[dir_D + 2] \leftarrow$D[47..32]	$D_{10} \leftarrow$MD$[dir_D + 10] \leftarrow$D[175..160]
$D_3 \leftarrow$MD$[dir_D + 3] \leftarrow$D[63..48]	$D_{11} \leftarrow$MD$[dir_D + 11] \leftarrow$D[191..176]
$D_4 \leftarrow$MD$[dir_D + 4] \leftarrow$D[31..16]	$D_{12} \leftarrow$MD$[dir_D + 12] \leftarrow$D[207..192]
$D_5 \leftarrow$MD$[dir_D + 5] \leftarrow$D[95..80]	$D_{13} \leftarrow$MD$[dir_D + 13] \leftarrow$D[223..108]
$D_6 \leftarrow$MD$[dir_D + 6] \leftarrow$D[111..96]	$D_{14} \leftarrow$MD$[dir_D + 14] \leftarrow$D[239..224]
$D_7 \leftarrow$MD$[dir_D + 7] \leftarrow$D[127..112]	$D_{15} \leftarrow$MD$[dir_D + 15] \leftarrow$D[255..240]

Tabla 6.8: Almacenar un entero de 256 bits en localidades de memoria de 16 bits

Los 64 bytes más significativos	Los 64 bytes menos significativos
$Q_0 \leftarrow$MD$[dir_Q + 0] \leftarrow$Q[15..00]	$Q_4 \leftarrow$MD$[dir_Q + 4] \leftarrow$Q[31..16]
$Q_1 \leftarrow$MD$[dir_Q + 1] \leftarrow$Q[31..16]	$Q_5 \leftarrow$MD$[dir_Q + 5] \leftarrow$Q[95..80]
$Q_2 \leftarrow$MD$[dir_Q + 2] \leftarrow$Q[47..32]	$Q_6 \leftarrow$MD$[dir_Q + 6] \leftarrow$Q[111..96]
$Q_3 \leftarrow$MD$[dir_Q + 3] \leftarrow$Q[63..48]	$Q_7 \leftarrow$MD$[dir_Q + 7] \leftarrow$Q[127..112]

Tabla 6.9: Almacenar un entero de 128 bits en localidades de memoria de 16 bits

La tabla 6.8 especifica el uso de memoria para leer un entero de 256 bits y la tabla 6.9 para un entero de 128 bits. Para obtener los primeros 16 bits se resuelve la ecuación $Q_0 \leftarrow q_0 \vee q_1 \vee q_2 \vee q_3 \vee q_4 \vee q_5 \vee q_6$, las operaciones necesarias son:

$$q_0 \leftarrow (D_0 \oplus D_8 \oplus (D_8 \ll 1) \oplus (D_8 \ll 2) \oplus (D_8 \ll 7)) \wedge \text{0xC000} \tag{6.7}$$

$$q_1 \leftarrow (D_0 \oplus D_8 \oplus (D_8 \ll 1) \oplus (D_8 \ll 2) \oplus (D_8 \ll 7) \oplus (D_{15} \gg 2)) \wedge \text{0x3E00} \tag{6.8}$$

$$q_2 \leftarrow (D_0 \oplus D_8 \oplus (D_8 \ll 1) \oplus (D_8 \ll 2) \oplus (D_8 \ll 7) \oplus (D_{15} \gg 2)) \wedge \text{0x0180} \tag{6.9}$$

$$q_3 \leftarrow (D_0 \oplus D_8 \oplus (D_8 \ll 1) \oplus (D_8 \ll 2) \oplus (D_{15} \ll 9) \oplus (D_{15} \gg 8) \oplus (D_{15} \gg 7)) \wedge \text{0x0070} \tag{6.10}$$

$$q_4 \leftarrow (D_0 \oplus D_8 \oplus (D_8 \ll 1) \oplus (D_8 \ll 2) \oplus (D_{15} \gg 9) \oplus (D_{15} \gg 8) \oplus (D_{15} \gg 7) \oplus (D_{15} \gg 12)) \wedge \text{0x000C} \tag{6.11}$$

$$q_5 \leftarrow (D_0 \oplus D_8 \oplus (D_8 \ll 1) \oplus (D_{15} \gg 9) \oplus (D_{15} \gg 8) \oplus (D_{15} \gg 13)) \wedge \text{0x0002} \tag{6.12}$$

$$q_6 \leftarrow (D_0 \oplus D_8 \oplus (D_{15}] \gg 9) \oplus (D_{15} \gg 14) \oplus (D_{15} \gg 15)) \wedge \text{0x0001} \tag{6.13}$$

Realizadas las operaciones anteriores se puede obtener Q_0, para el resto de los bits se realizan las siguientes operaciones:

$$Q_1 \leftarrow D_1 \oplus D_9 \ \oplus (D_9 \ \ll 1 \vee D_8 \ \gg 15) \oplus (D_9 \ \ll 2 \vee D_8 \ \gg 14) \oplus (D_9 \ \ll 7 \vee D_8 \ \gg 9) \tag{6.14}$$

$$Q_2 \leftarrow D_2 \oplus D_{10} \oplus (D_{10} \ll 1 \vee D_9 \ \gg 15) \oplus (D_{10} \ll 2 \vee D_9 \ \gg 14) \oplus (D_{10} \ll 7 \vee D_9 \ \gg 9) \tag{6.15}$$

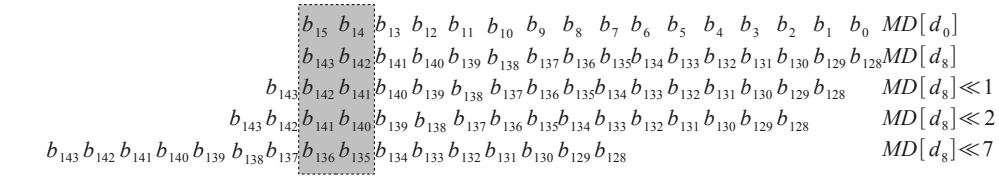
$$Q_3 \leftarrow D_3 \oplus D_{11} \oplus (D_{11} \ll 1 \vee D_{10} \gg 15) \oplus (D_{11} \ll 2 \vee D_{10} \gg 14) \oplus (D_{11} \ll 7 \vee D_{10} \gg 9) \tag{6.16}$$

$$Q_4 \leftarrow D_4 \oplus D_{12} \oplus (D_{12} \ll 1 \vee D_{11} \gg 15) \oplus (D_{12} \ll 2 \vee D_{11} \gg 14) \oplus (D_{12} \ll 7 \vee D_{11} \gg 9) \tag{6.17}$$

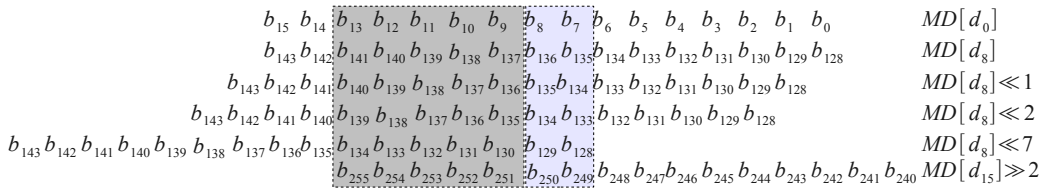
$$Q_5 \leftarrow D_5 \oplus D_{13} \oplus (D_{13} \ll 1 \vee D_{12} \gg 15) \oplus (D_{13} \ll 2 \vee D_{12} \gg 14) \oplus (D_{13} \ll 7 \vee D_{12} \gg 9) \tag{6.18}$$

$$Q_6 \leftarrow D_6 \oplus D_{14} \oplus (D_{14} \ll 1 \vee D_{13} \gg 15) \oplus (D_{14} \ll 2 \vee D_{13} \gg 14) \oplus (D_{14} \ll 7 \vee D_{13} \gg 9) \tag{6.19}$$

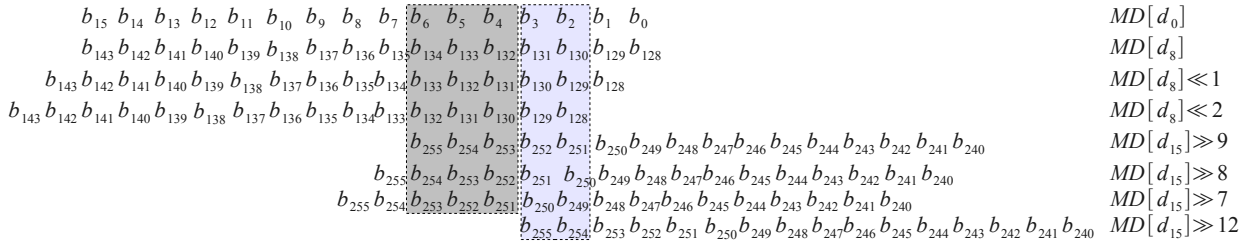
$$Q_7 \leftarrow D_7 \oplus D_{15} \oplus (D_{15} \ll 1 \vee D_{14} \gg 15) \oplus (D_{15} \ll 2 \vee D_{14} \gg 14) \oplus (D_{15} \ll 7 \vee D_{14} \gg 9) \tag{6.20}$$



(a) q_0

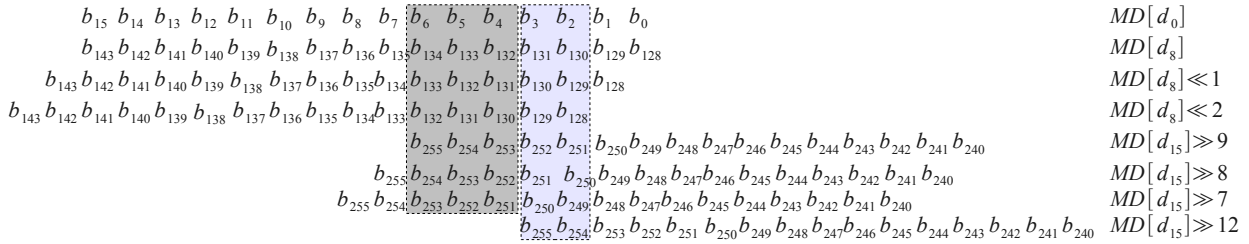


(b) q_1 y q_2

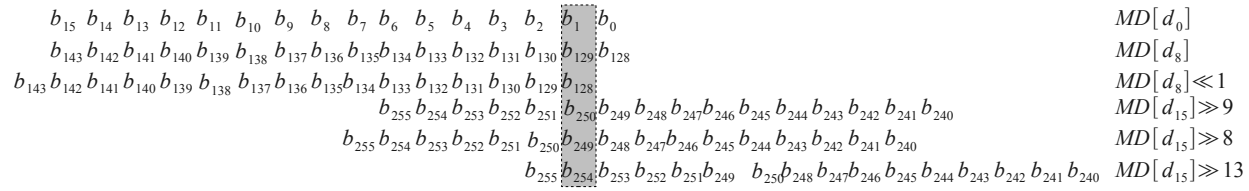


(c) q_3 y q_4

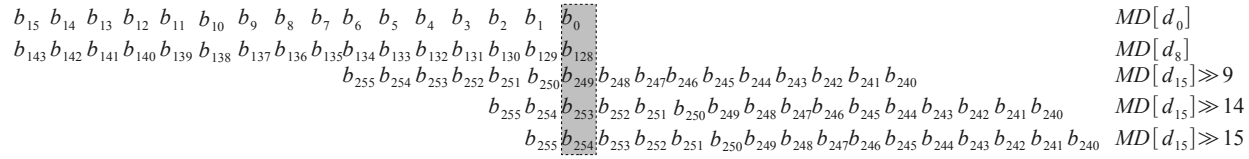
Figura 6.12: Desplazamientos para calcular q_i.



(a) q_3 y q_4



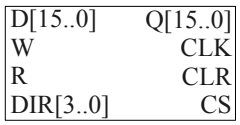
(b) q_5



(c) q_6

Figura 6.13: Desplazamientos para calcular q_i, continuación.

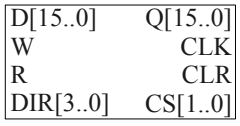
Las figura 6.12 y 6.13 muestra los desplazamientos de los datos almacenados en memoria para obtener los dos bytes menos significativos por medio de la reducción.



(a)

Señal	E/S	Bits	Descripción
D	E	16	Bus de datos
Q	S	16	Bus de datos
R	E	1	Habilita la lectura de datos
W	E	1	Habilita la escritura de datos
CS	E	1	Habilita la lectura y escritura
DIR	E	4	Selector de registro
CLK	E	1	Reloj del sistema
CLR	E	1	Reinicio del componente

(b) Interfaz del multiplicador de 16 por 16 bits



(c)

CS[0]	E	1	Accede a los 128 bits (R)
CS[1]	E	1	Accede a los 256 bits (W/R)

(d) Interfaz del componente que realiza la reducción

Figura 6.14: Símbolo y descripción de señales: (a) y (b) del multiplicador de 16x16 bits, (c) y (d) de la reducción .

Para realizar la multiplicación y la reducción se agregaron dos componentes más al microcontrolador, la figura 6.14 muestra la interfaz para acceder a estos componentes, cuyos registros se encuentran mapeados en memoria de datos. La interfaz de ambos componentes

difiere en el selector del chip, para la multiplicación solo es necesaria una señal. Para la reducción se necesita más de 16 registros el CS[0] accede a los 16 registros del entero de 256 bits y el CS[1] al entero de 128 bits. La figura 6.15 muestra los arreglos que se utilizan en la multiplicación de 128 por 128 bits. Las gráfica de ondas para la reducción se ve en las figuras 6.15 y 6.16.

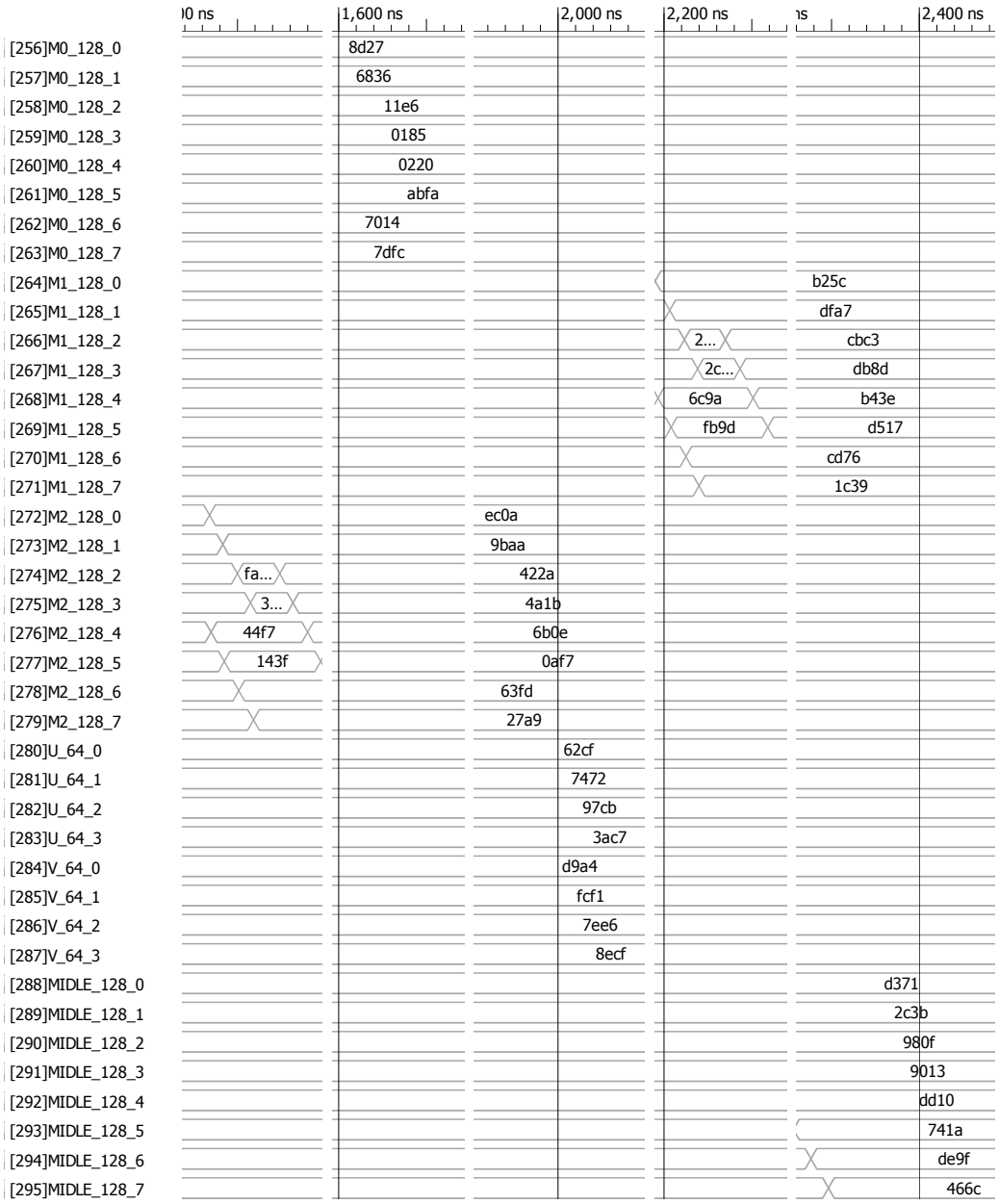
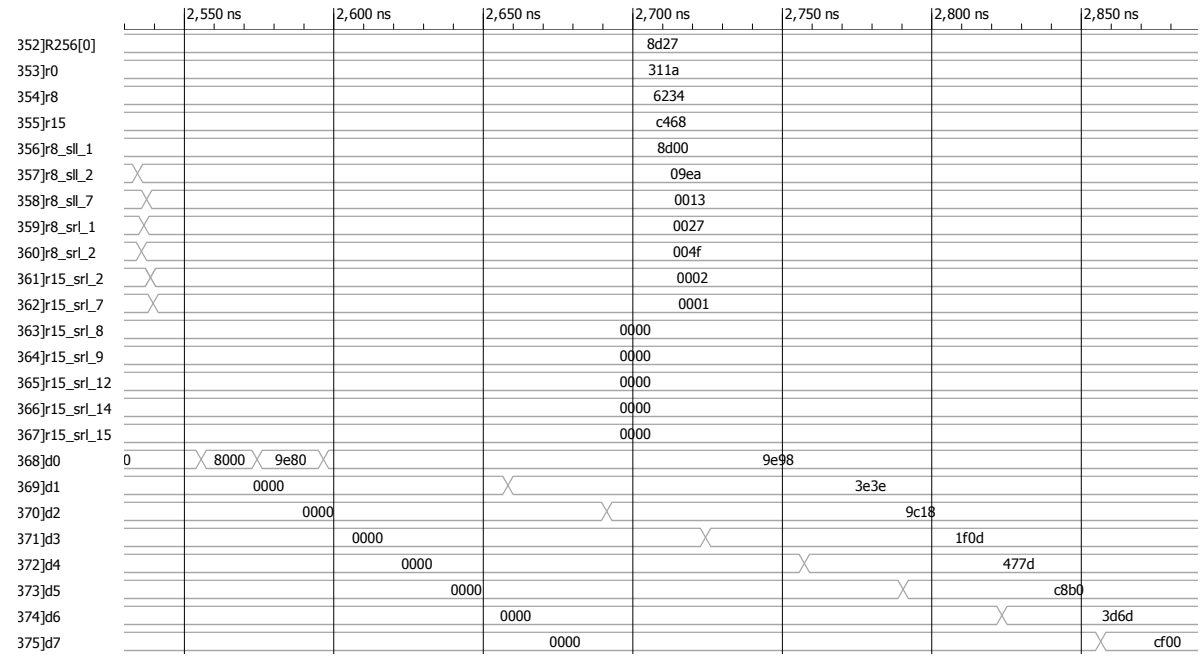
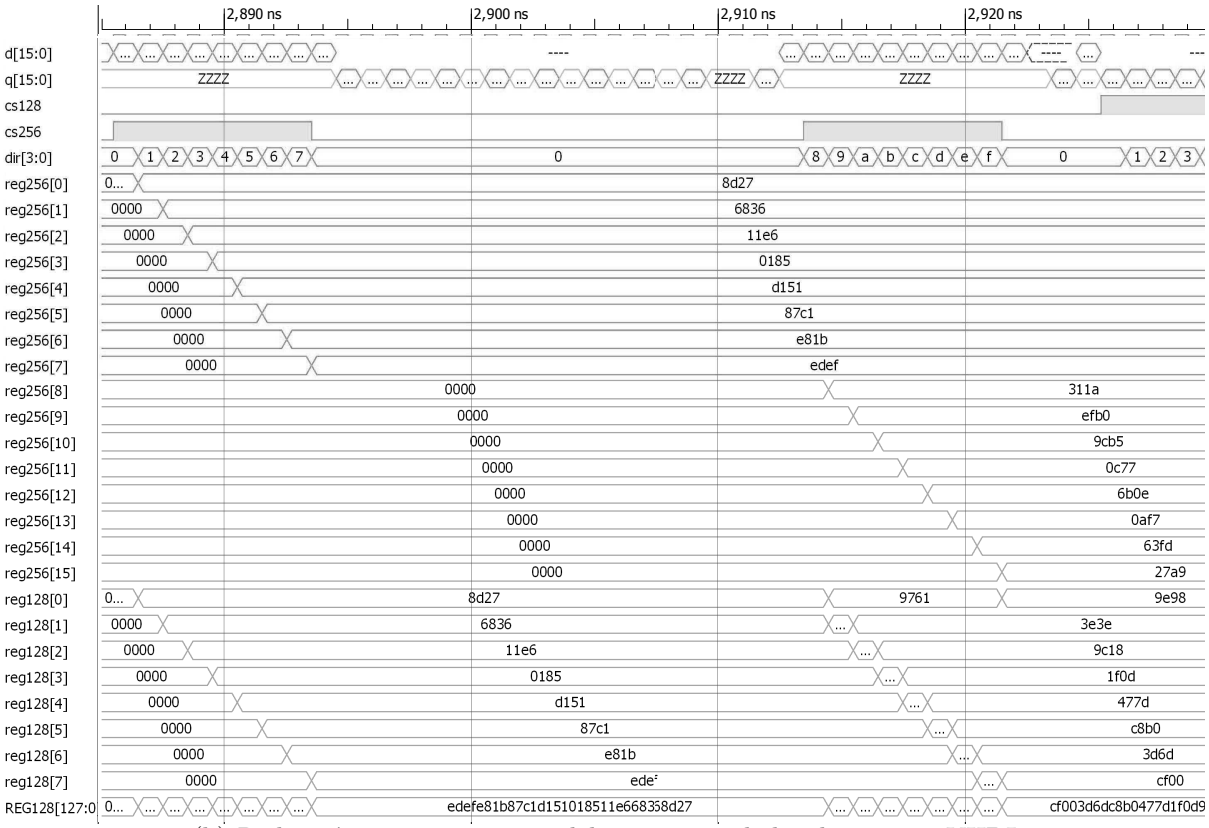


Figura 6.15: Variables utilizadas para la multiplicación de 128 por 128 bits, porción de la memoria de datos.



(a) Reducción con subrutina en ensamblador ASM



(b) Reducción con componente del microcontrolador descrita en VHDL

Figura 6.16: Gráfica de ondas de la reducción

6.6. Cifrado en bloque

Este sistema cifra el mensaje original agrupando en los símbolos del mensaje en grupos o bloques. Se denotará por símbolo a un carácter estándar de cualquier alfabeto, todos los caracteres usados en comunicaciones digitales se codifican mediante una sucesión de bits, se entenderá que un símbolo es un bit. El cifrado de bloque pertenece a los cifrados de llave simétrica, conocido como llave secreta. La llave es única y se utiliza para cifrar o descifrar.

Propiedades del cifrado en bloque:

* Dependencia entre bits: en cada bloque cada bit del texto cifrado depende de los bits de la llave y todos los bits del bloque de texto claro.
* Cambio de los bits de entrada: un bit modificado en un bloque de texto claro o de la llave produce un cambio de los bits del bloque de texto cifrado (debería ser un 50 % de cambios).

Elementos que conforman los cifrados en bloque:

* Una transformación inicial: consiste en aleatorizar simplemente los datos de entrada (ocultar bloques de datos), sirve para entorpecer ataques por análisis lineal o diferencial (en función de la llave).
* Función criptográfica iterada n veces: función no lineal complicada de los datos y la llave. La función no lineal puede estar formada por una sola operación muy compleja o por la sucesión de varias transformaciones simples. Durante las iteraciones con subllaves diferentes correspondientes no son equivalentes a una pasada única con una subllave diferente, lo que sería un desastre.
* Una transformación final: consiste en invertir la transformación inicial.
* Una función de expansión de llave: consiste en convertir la llave del usuario, con una longitud entre 56 y 256 bits, en un conjunto de subllaves que puedan estar constituidas por varios cientos de bits en total. Conviene que sea unidireccional y que el conocimiento de una o varias subllaves intermedias no permite deducir las subllaves anteriores.

Modo contador CTR

Crea una serie cifrante bloque a bloque cifrado con el cifrador a bloque que se use, que luego se suma módulo 2, bit a bit, con los sucesivos bloques del texto claro o del cifrado. La longitud de la palabra del contador ha de ser igual al tamaño de b bloques del cifrador en bloques que esté usando, es decir, 128 bits para el AES.

Para cada llave distinta que se use, el contenido de cada contador debe ser diferente y no debe reutilizarse, es decir, si se usa una llave de sesión y cada sesión incluye el cifrado de varios documentos que se cifre. Una forma sencilla de conseguir este fin es construir de forma aleatoria un número de uso único (nonce, *number used once*) como cabecera del mensaje cifrado. El contenido inicial del primer contador es el *nonce*, el valor de los contadores seria el sucesivo incremento del primer contador.

Las propiedades del modo contador son:

- Cada bloque cifrado es función del *nonce*, del incremento del contador, de la llave y del correspondiente bloque de texto claro.
- Cada bloque descifrado es función del *nonce*, del incremento del contador, de la llave y del correspondiente bloque de texto cifrado.
- Los errores de bits no se propagan: un bit erróneo de transmisión produce un solo bit erróneo en el texto claro.
- Se puede hacer que cifre mensajes de forma diferente con sólo cambiar cada vez el *nonce*.
- No cambia el espacio de llaves.
- Se puede cifrar y descifrar en paralelo.

La figura 6.17 muestra el cifrado y descifrado utilizando el modo contador o CTR.

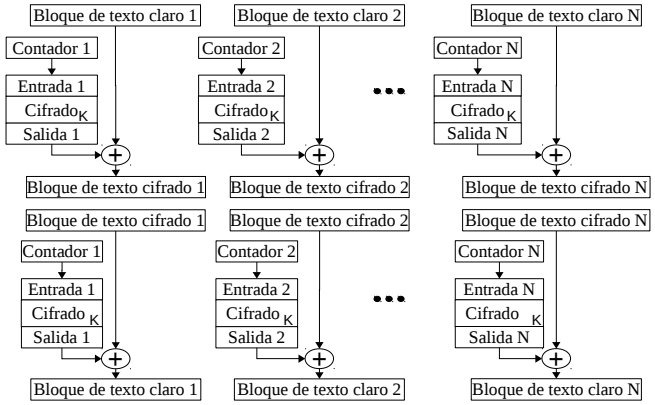


Figura 6.17: Cifrado y descifrado en bloque en modo contador.

6.6.1. Estándar avanzado de cifrado AES

El AES según [34] es un cifrado iterativo, que emplea funciones invertibles y opera con bloques enteros. El resultado obtenido en cada paso se le denomina *estado* y consiste en un conjunto de tantos bits como la longitud del bloque. Los bits adyacentes se agrupan de 8 en 8 formando bytes y estos en una tabla cuadrada de cuatro renglones y cuatro columnas, entonces el bloque es de 128 bits, o 16 bytes, esto es ilustrado en la figura 6.18.

$$\begin{pmatrix} i_0 & i_4 & i_8 & i_{12} \\ i_1 & i_5 & i_9 & i_{13} \\ i_2 & i_6 & i_{10} & i_{14} \\ i_3 & i_7 & i_{11} & i_{15} \end{pmatrix}; \qquad \begin{pmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{pmatrix}$$

Figura 6.18: Representación de un estado del AES, con bloque de entrada i_x, el índice x indica el byte x del bloque; estado $s_{i,j}$ donde los índices i, j indican el byte dentro de la matriz de estado.

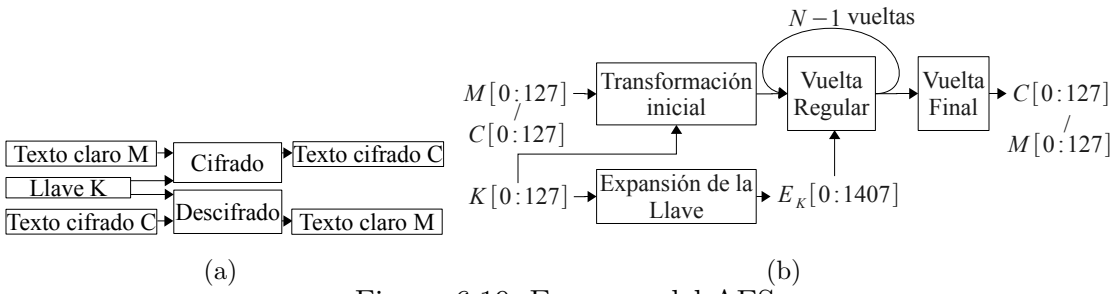


Figura 6.19: Esquema del AES

El esquema de cifrado del AES se ilustra en la figura 6.19. Las N rondas están determinadas por el tamaño de la llave: para un tamaño de 128, N es igual a 10; para un tamaño de 192, $N = 12$; y para un tamaño de llave de 256, $N = 14$. El proceso de cifrado consta de tres fases:

1. Transformación inicial: es una suma módulo 2 ($\oplus : XOR$) con la llave K .

2. $N - 1$ rondas regulares que constan de cuatro transformaciones:

   a) SubBytes: sustitución no lineal de bytes.

   b) ShiftRow: desplazamiento circular de los renglones del estado.

   c) MixColumns: mezcla de columnas.

   d) AddRoundKey: es una suma módulo 2 con la subllave de ronda correspondiente ($E_K[i : i + 15], i \in \{16, 32, \ldots 144\}$ donde K[0:127])

3. Una ronda final: se realizan tres de las transformaciones anteriores ignorando $MixColumns$ y se utiliza la ultima subllave.

SubBytes

La sustitución consiste en tomar un byte, por ejemplo 0x9d, y con ayuda de la tabla 6.10 se obtine un nuevo byte seleccionado del renglón 9 y la columna d: resultando de 0x9d el valor 0x5e. Los autores del AES proporcionan una fórmula matemática para evitar la sospecha de una posible trampa, cumpliendo con los siguientes criterios:

* Minimizar la correlación de la entrada con la salida.

* Minimizar la probabilidad de propagación de diferencias.

* Maximizar la complejidad de la expresión de transformación.

Para el proceso de cifrado y descifrado nunca se usan las fórmulas (transformaciones), se realiza una consulta en las tablas de búsqueda. Para el cifrado la tabla asociada es la S-Box, mostrada en la tabla 6.10, y para el descifrado es la tabla inversa del S-Box.

xy	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f
00	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
10	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
20	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
30	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
40	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
50	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
60	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
70	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
80	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
90	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	[**5e**]	0b	db
a0	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
b0	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
c0	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d0	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e0	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f0	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Tabla 6.10: Tabla de búsqueda S-Box para el AES, el byte xy determina la columnas y y el renglón x para retornar un nuevo valor.

ShiftRow

En esta operación el primer renglón no se modifica, los siguientes renglones se rotan una, dos y tres veces los bytes a la izquierda, respectivamente. Esta operación modifica el estado inicial para generar otro estado como se muestra en las siguientes matrices:

$$\begin{pmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{pmatrix} \rightarrow \begin{pmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,1} & s_{1,2} & s_{1,3} & s_{1,0} \\ s_{2,2} & s_{2,3} & s_{2,0} & s_{2,1} \\ s_{3,3} & s_{3,0} & s_{3,1} & s_{3,2} \end{pmatrix}$$

MixColumns

Esta transformación consiste en multiplicar cada columna por una matriz, la matriz para el cifrado es:

$$\begin{pmatrix} s'_{0,j} \\ s'_{1,j} \\ s'_{2,j} \\ s'_{3,j} \end{pmatrix} = \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \begin{pmatrix} s_{0,j} \\ s_{1,j} \\ s_{2,j} \\ s_{3,j} \end{pmatrix}$$

y la matriz para el descifrado es:

$$\begin{pmatrix} s_{0,j} \\ s_{1,j} \\ s_{2,j} \\ s_{3,j} \end{pmatrix} = \begin{pmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{pmatrix} \begin{pmatrix} s'_{0,j} \\ s'_{1,j} \\ s'_{2,j} \\ s'_{3,j} \end{pmatrix}$$

AddRoundKey

En esta transformación, en cada ronda se realiza una operación XOR ($\oplus$) de la subllave con el estado de la forma siguiente:

$$\begin{pmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{pmatrix} = \begin{pmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{pmatrix} \oplus \begin{pmatrix} E_{k[i][0]} & E_{K[i][4]} & E_{K[i][8]} & E_{K[i][12]} \\ E_{K[i][1]} & E_{K[i][8]} & E_{K[i][9]} & E_{K[i][13]} \\ E_{K[i][2]} & E_{K[i][6]} & E_{K[i][10]} & E_{K[i][14]} \\ E_{K[i][3]} & E_{K[i][7]} & E_{K[i][11]} & E_{K[i][15]} \end{pmatrix}$$

Los pseudocódigos son basados en la implementación del AES de la biblioteca RELIC disponible en [23] que se probó en un microcontrolador MSP430X de *Texas Instrument*, que está basada en [35] cuya licencia es GNU GPL v2.1. Los pseudocódigos 20 y 21 realizan una consultas en varias tablas de búsqueda, los pseudocódigos 22 y 23 muestran los desplazamientos, el pseudocódigos 24 genera las subllaves, el pseudocódigo 25 muestra el método de cifrado y el pseudocódigo 26 muestra el método de descifrado. La figura 6.20 muestra la gráfica de onda para obtener los ciclos por byte.

Todas las tablas de búsqueda almacenan 256 elementos, la tabla *isbox* almacena solo enteros de 1 byte, las demás de 2 bytes. La descripción de cada tabla de búsqueda se enlista a continuación, considerando el campo finito del AES:

* *isbox* es la inversa S-Box del AES.

* *smul*21 combina la tabla de la S-Box multiplicando por x (byte inferior).

* *smul*13 combina la tabla de la S-Box multiplicando por $x + 1$ (byte superior).

* *smul*32 combina la tabla de la S-Box multiplicando por $x+1$ (byte superior) y por x (byte inferior).

* *smul*11 carga la tabla de consulta de la S-Box (byte superior e inferior).

* *mule*9 carga $x^3 + x^2 + x$ (byte inferior) y $x^3 + 1$ (byte superior).

* *muldb* carga $x^3 + x^2 + 1$ (byte inferior) y $x^3 + x + 1$ (byte superior).

* *mulbe* carga $x^3 + x + 1$ (byte inferior) y $x^3 + x^2 + x$ (byte superior).

* *mul*9*d* carga $x^3 + 1$ (byte inferior) y $x^3 + x^2 + 1$ (byte superior).

M		C	E_{key}
68 6f	h o	a9 f6	2a 6e ae 9a 9a c8 be 34 4b 86 a0 e5 c7 b9 9d 45
6c 61	l a	e1 de	7d 31 c1 5d e7 f9 7f 69 ac 7f df 8c 6b c6 42 c9
2d 6d	- m	da f9	cb 1f 1e 20 2c e6 61 49 80 99 be c5 eb 5f fc c
75 6e	u n	d3 a7	0 ab e4 cd 2c 4d 85 84 ac d4 3b 41 47 8b c7 4d
64 6f	d o	c2 aa	35 65 f 65 19 28 8a e1 b5 fc b1 a0 f2 77 76 ed
2d 63	- c	28 d5	d0 4d 4a fc c9 65 c0 1d 7c 99 71 bd 8e ee 7 50
72 75	r u	e2 40	d8 a8 39 c5 11 cd f9 d8 6d 54 88 65 e3 ba 8f 35
65 6c	e l	54 e8	6c 9b ef 94 7d 56 16 4c 10 2 9e 29 f3 b8 11 1c
			80 99 f3 19 fd cf e5 55 ed cd 7b 7c 1e 75 6a 60
			6 80 38 70 fb 4f dd 25 16 82 a6 59 8 f7 cc 39
			58 fd 1c 76 a3 b2 c1 53 b5 30 67 a bd c7 ab 33

Tabla 6.11: Crifrado y descifrado de un mensaje usando AES. Texto claro M, texto cifrado C y subllaves generadas E_{key}

Pseudocódigo 20 Función auxiliar del descifrado

función AUX_DECIFRADO(S_0, S_1, S_2, S_3)
  $A \leftarrow 0\text{xFF} \wedge S_0, B \leftarrow 0\text{xFF} \wedge S_1, C \leftarrow 0\text{xFF} \wedge S_2, D \leftarrow 0\text{xFF} \wedge S_3$
  $r_0 \leftarrow mule9[A] \oplus mulbe[B] \oplus muldb[C] \oplus mul9d[D]$
  $r_1 \leftarrow muldb[A]^m ul9d[B]^m ule9[C]^m ulbe[D]$
  $t_0 \leftarrow isbox[0\text{xFF} \wedge r_0]$
  $t_1 \leftarrow isbox[0\text{xFF} \wedge (r_0 >> 8)]$
  $t_2 \leftarrow isbox[(0\text{xFF} \wedge r_1]$
  $t_3 \leftarrow= isbox[(0\text{xFF} \wedge (r_1 >> 8)]$;
  devolver t_0, t_1, t_2, t_3
fin función

Pseudocódigo 21 Función auxiliar de cifrado

función AUX_CIFRADO(S_0, S_1, S_2, S_3)
  $A \leftarrow 0\text{xFF} \wedge S_0, B \leftarrow 0\text{xFF} \wedge S_1, C \leftarrow 0\text{xFF} \wedge S_4, D \leftarrow 0\text{xFF} \wedge S_3$
  $r0 \leftarrow smul21[A] \oplus smul32[B] \oplus smul13[C] \oplus smul11[D]$
  $r1 \leftarrow smul13[A] \oplus smul11[B] \oplus smul21[C] \oplus smul32[D]$
  devolver $r0 \wedge 0\text{x00FF}, (r0 \gg 8) \wedge 0\text{x00FF}, r1 \wedge 0\text{x00FF}, (r1 \gg 8) \wedge 0\text{x00FF}$
fin función

Pseudocódigo 22 Funcione de desplazamiento del descifrado

procedimiento INV_SHIFT_SUB(j,T,S)
  T[4*j+0]← 0x00FF∧ isbox[s[4*j]];
  T[4*j+1]← 0x00FF∧ isbox[s[4*((j+3) %4)+1]];
  T[4*j+2]← 0x00FF∧ isbox[s[4*((j+2) %4)+2]];
  T[4*j+3]← 0x00FF∧ isbox[s[4*((j+1) %4)+3]];
fin procedimiento

Pseudocódigo 23 Funcion de desplazamiento de cifrado

```
1: procedimiento SHIFT_SUB(j,T,S)
2:     T[4*j+0]← 0x00FF∧ smul11[S[4 * j]];
3:     T[4*j+1]← 0x00FF∧ smul11[S[4*((j+1) %4)+1]];
4:     T[4*j+2]← 0x00FF∧ smul11[S[4*((j+2) %4)+2]];
5:     T[4*j+3]← 0x00FF∧ smul11[S[4*((j+3) %4)+3]];
6: fin procedimiento
```

Pseudocódigo 24 Generar subllaves

Entrada: Llave de 16 bytes K[0:15]
Salida: Llave Extendida 11*16 bytes $E_{key}[0:175]$

```
1: función GENERARLLAVES(K)
2:     rcon ← 1, a ← 16 y t[3:0] ← {0x0000,... }
3:     E_key[0 : 15] ← K[0 : 15]
4:     mientras a < (11 ∗ 16) hacer                                  ▷ a < 176
5:         t[0] ← 0xFF ∧ (smul11[E_key[a − 3]] ⊕ rcon)
6:         t[1] ← 0xFF ∧ smul11[E_key[a − 1]]
7:         t[2] ← 0xFF ∧ smul11[E_key[a − 2]]
8:         t[3] ← 0xFF ∧ smul11[E_key[a − 4]]
9:         rcon ← 0xFF ∧ ((rcon ≪ 1) ⊕ ((rcon ≫ 7) ∗ 0x11b)
10:        para j ← 0 hasta 3 hacer
11:            t[0] ← 0xFF ∧ (t[0] ⊕ E_key[a − 16])
12:            t[1] ← 0xFF ∧ (t[0] ⊕ E_key[a − 15])
13:            t[2] ← 0xFF ∧ (t[0] ⊕ E_key[a − 14])
14:            t[3] ← 0xFF ∧ (t[0] ⊕ E_key[a − 13])
15:            E_key[a : a + 4] ← t[0 : 3]
16:        fin para
17:    fin mientras
18:    devolver E_key
19: fin función
```

Pseudocódigo 25 Cifrar un mensaje de 16 bytes

Entrada: Llave extendida 176 bytes $E_{key}[0:175]$ y mensaje claro 16 bytes $M[0:15]$
Salida: Mensaje Cifrado 16 bytes $C[0:15]$

```
1: función CIFRADO_AES128(E_key,M)
2:     T[0 : 15] ← S[0 : 15] ← {0x00,... }
3:     S ← M[0 : 15] ⊕ E_key[0 : 15]
4:     para i ← 16 hasta i < 160 paso i ← i + 16 hacer
5:         T[0 : 3] ← AUX_0(S[0],S[5],S[10],S[15])
6:         T[4 : 7] ← AUX_0(S[4],S[9],S[14],S[3])
7:         T[8 : 11] ← AUX_0(S[8],S[13],S[2],S[7])
8:         T[12 : 15] ← AUX_0(S[12],S[1],S[6],S[11])
9:         S[0 : 15] ← T[0 : 15] ⊕ E_key[i : i + 15]
10:    fin para
11:    SHIFT_SUB(0,T,S), SHIFT_SUB(1,T,S), SHIFT_SUB(2,T,S), SHIFT_SUB(3,T,S)
12:    C[0 : 15] ← T[0 : 15] ⊕ E_key[i : i + 15]
13:    devolver C
14: fin función
```

Pseudocódigo 26 Descifrar un mensaje de 16 bytes

Entrada: Llave extendida 176 bytes $E_{key}[0:175]$ y mensaje cifrado 16 bytes $C[0:15]$
Salida: Mensaje claro 16 bytes $M[0:15]$

función DECIFRADO_AES128(E_{key},C)
  $S[0:15], T[0:15] \leftarrow \{0x0000, \dots\}$
  $S[0:15] \leftarrow C[0:15] \oplus E_{key}[160:175]$
  INV_SHIFT_SUB(0,T,S),INV_SHIFT_SUB(1,T,S),INV_SHIFT_SUB(2,T,S),INV_SHIFT_SUB(3,T,S)
  para $i \leftarrow 9 * 16$ **hasta** $i > 0$ **paso** $i \leftarrow i - 16$ **hacer**
    $S[0:15] \leftarrow T[0:15] \oplus E_{key}[i:i+15]$
    T[0],T[5],T[10],T[15]←AUX_DECIFRADO(S[0],S[1],S[2],S[3])
    T[4],T[9],T[14],T[3]←AUX_DECIFRADO(S[4],S[5],S[6],S[7])
    T[8],T[13],T[2],T[7]←AUX_DECIFRADO(S[8],S[9],S[10],S[11])
    T[12],T[1],T[6],T[11]←AUX_DECIFRADO(S[12],S[13],S[14],S[15])
  fin para
  $M[0:15] \leftarrow T[0:15] \oplus E_{key}[0:15]$
  devolver M
fin función

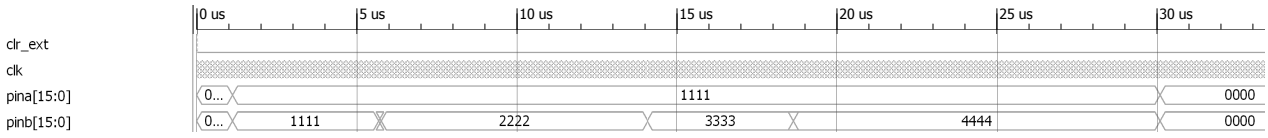


Figura 6.20: Para medir los ciclos de reloj que ocupan las operaciones criptográficas, se desplegaron marcas por los puertos paralelos A y B. El puerto A despliega 0x1111 durante la prueba. El puerto B despliega 0x1111 y 0x3333 cuando se generan las subllaves, 0x2222 en el cifrado y 0x4444 en el descifrado.

6.6.2. Encriptación autenticada

Un esquema de encriptación autenticada (EA) se compone de dos métodos: encriptación autenticada y desencriptación-verificación (de la integridad). El método de encriptación autenticada se denota por la función Cifrado(K, N, M, A) que devuelve (C, T), donde $K \in \{0,1\}^k$ es la llave de $k \leftarrow 128$ bits, $N \in \{0,1\}^n$ es el nonce n bits, $M \in \{0,1\}^*$ es el mensaje, $A \in \{0,1\}^*$ son los datos asociados, $C \in \{0,1\}^*$ es el texto cifrado y $T \in \{0,1\}^t$ es la etiqueta de autenticación. El método de desencriptación-verificación se denota por Descifrado(K, N, C, A, T) que devuelve (M, V) donde K, N, C, A, T, M se explicaron anteriormente y V es un valor booleano que indica si la variable dada es válido (es decir, si el mensaje descifrado y los datos asociados son los únicos auténticos).

Muchos esquemas de EA se construyen usando un cifrado en bloque como el AES. Cifrado(K, B) denota el cifrado del bloque, donde la clave K y $B \in \{0,1\}^b$ es un mensaje de b bits (un bloque). La función inversa (descifrado) se denota Descifrado(K, B) donde B es también un bloque (por lo general del texto cifrado). Los métodos de EA utilizados en las pruebas son el CCM y GCM, la descripción para sus uso con microcontroladores se describe en [22].

Modo CCM

El pseudocódigo 27 describe el funcionamiento de encriptación con CCM, donde la subrutina FORMATO calcula un bloque de encabezado $B0$ (donde codifica la longitud de la etiqueta, la longitud del mensaje y el nonce). Los bloques $A_1, ..., A_a$ (codifica la longitud de los datos asociados junto con los datos en sí) y los bloques $M_1, ..., M_m$ representan el mensaje original. La subrutina INICIALIZAR_CONTADOR devuelve el contador inicial basada en el *nonce*. La función INCREMENTAR incrementa el contador.

Pseudocódigo 27 CCM encriptación

Entrada: Mensaje M, datos adicionales A, número arbitrario (nonce) N y la llave K.
Salida: Texto cifrado C y una etiqueta de autenticado T

función CIFRADO_CCM(M,A,N,K)
    $B_0, A_1, ..., A_a, M_1, ..., M_m \leftarrow$FORMATO(N,A,M)
    $Y \leftarrow$ CIFRADO(K,B0)
    para $i \leftarrow 1$ **hasta** a **hacer**
        Y $\leftarrow$ CIFRADO(K, $A_i \oplus$ Y)
    fin para
    J$\leftarrow$ INICIALIZAR_CONTADOR
    $S_0 \leftarrow$ CIFRADO(K, J)
    J$\leftarrow$ INCREMENTAR(J)
    para $i \leftarrow$**hasta** m **hacer**
        U $\leftarrow$ CIFRAR(K,J)
        J $\leftarrow$ INCREMENTAR(J)
        S $\leftarrow M_i \oplus$ Y
        Y $\leftarrow$ CIFRADO(K,S)
        $C_i \leftarrow M_i \oplus U$
    fin para
    T $\leftarrow$Y[0..t-1]$\oplus S_0$[0..t-1]
    devolver T, C
fin función

Las figuras 6.21 y 6.22 muestran los valores de la interfaz del microcontrolador, el puerto A toma el valor de 0x1111 para el cifrado y 0x2222 para el descifrado. El puerto B toma el valor 0x1111 cuando se ejecuta la subrutina ac_ccm_key, toma el valor 0x2222 para la subrutina ac_ccm_init, toma los valores 0x3333, 0x4444 y 0x5555 para ac_ccm_data, toma los valor 0x6666 y 0x7777 para ac_ccm_enc (o ac_ccm_dec) y por último toma el valor 0x8888 para la subrutina ac_ccm_tag. La figura 6.23 muestra las localidades de memoria que almacena el texto cifrado C y el texto claro M.

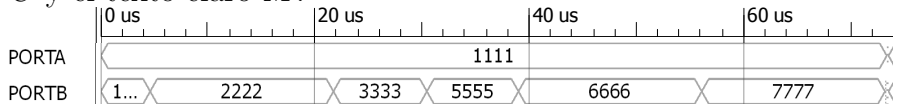


Figura 6.21: CCM cifrado

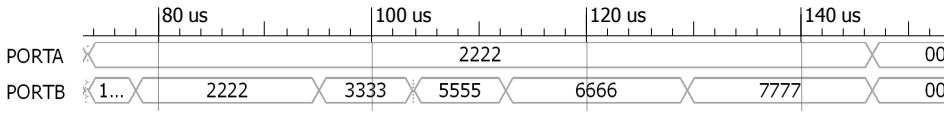


Figura 6.22: CCM descifrado

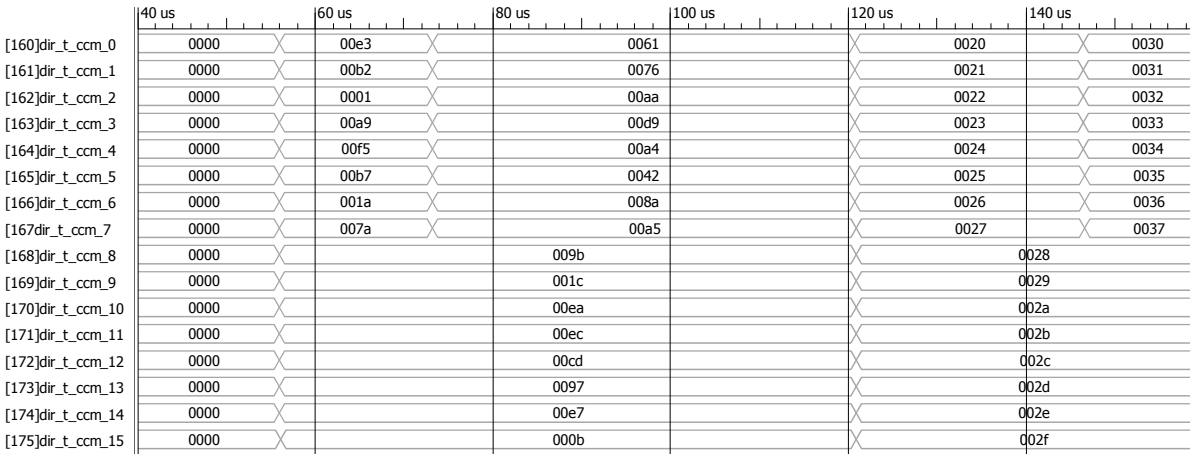


Figura 6.23: Texto claro y cifrado

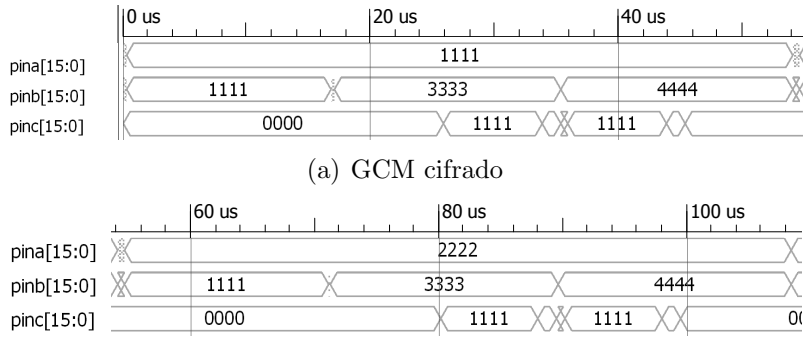
Modo GCM

El pseudocódigo 28 describe la encriptación con GCM, donde la subrutinas INCREMENTAR e INICIALIZAR_CONTADOR incrementa e inicializa el contador. La operación $A \times B$ denota la multiplicación de A y B en $F(2^{128})$. Este modo de EA se beneficia de la búsqueda de tablas precalculadas desde la segundo operación que se fija para todas las multiplicaciones (líneas 6, 15 y 18 del pseudocódigo 28).

Pseudocódigo 28 Cifrado GCM

Entrada: Mensaje M, Datos asociados A, Número arbitrario (nonce) N, y una llave K.
Salida: Texto cifrado C y la etiqueta de autenticado T

1: **función** CIFRADO_GCM(M,A,N,K,T)
2: $A_1, ..., A_a \leftarrow A$ y $M_1, ..., M_m \leftarrow M$ ▷ Se divide en bloques
3: H← CIFRADO(K, 0^{128})
4: Y $\leftarrow 0^{128}$
5: **para** $i \leftarrow 1$ **hasta** a **hacer**
6: Y $\leftarrow (A_i \oplus Y) \times H$
7: **fin para**
8: J← INICIALIZAR_CONTADOR(N)
9: S_0 ←CIFRAR(K,J)
10: J ← INCREMENTAR(J)
11: **para** $i \leftarrow 1$ **hasta** m **hacer**
12: U← CIFRAR(K,J)
13: J ← INCREMENTAR(J)
14: $C_i \leftarrow M_i \oplus U$
15: $Y \leftarrow (C_i \oplus Y) \times H$
16: **fin para**
17: $L \leftarrow [\text{LONGITUD(A)}]_{64} || [\text{LONGITUD(M)}]_{64}$
18: S $\leftarrow (L \oplus Y) \times H$
19: T $\leftarrow (S \oplus S_0)$[0..t-1]
20: **devolver** T, C
21: **fin función**



(a) GCM cifrado

(b) GCM descifrado

Figura 6.24: Gráfica de ondas del GCM

La figura 6.24 muestran los valores de la interfaz del microcontrolador. El puerto A toma el valor 0x1111 para el cifrado y 0x2222 para el descifrado. El puerto B toma los valores 0x1111 cuando se ejecuta la subrutina ac_gcm_key, toma el valor 0x2222 para la subrutina ac_gcm_init, toma el valor 0x3333 para ac_gcm_dec y por ultimo toma el valor 0x4444 para ac_gcm_tag. La figura 6.25 muestra el texto cifrado C y el texto claro M.

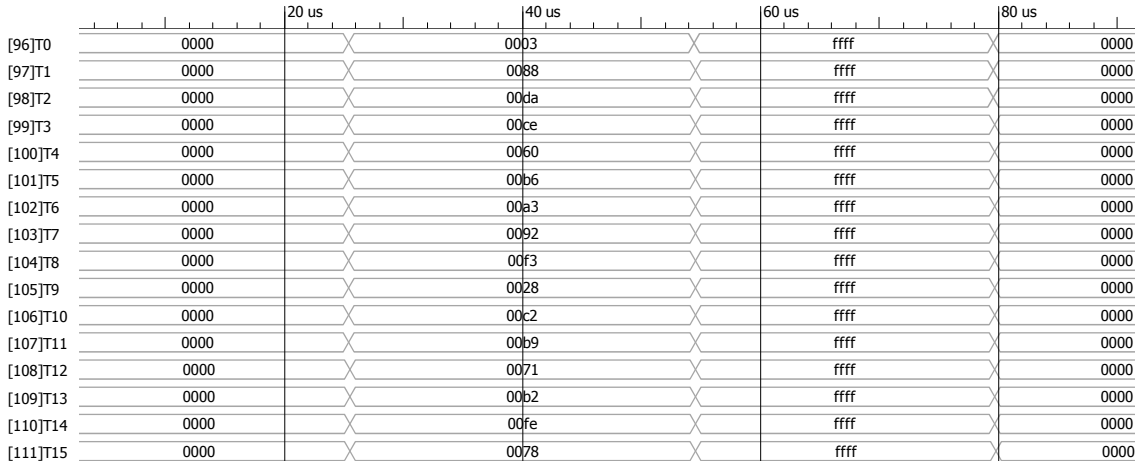


Figura 6.25: Mensaje cifrado y mensaje claro con GCM

6.7. Resultados

Las subrutinas implementadas para la encriptación autenticada basadas en RELIC son: *ac_X_key* genera las subllaves E_{key}, *ac_X_init* inicializa la estructura para todas las demás subrutinas, *ac_X_enc* y *ac_X_dec* realiza el cifrado y descifrado con el método AE seleccionado, *ac_X_tag* obtiene la etiqueta y por ultimo *ac_X_data* realiza las operaciones con los datos asociados.

Las tablas describen el número de ciclos que se demora en cada proceso de cifrado y descifrado: para el GCM 6.13 y 6.14, para el CCM 6.15 y 6.16 y para el AES 6.12

Subrutina	rango de ciclos	ciclos
generar subllaves	1,146.5 – 5,683.5	4,537.0
cifrar	5,792.5- 14.107.5	8,315.0
generar subllaves	14,107.5- 18,644.5	4,537.0
descifrar	18,646.5-30,091.5	11,445.0

Tabla 6.12: Número de ciclos de cifrado y descifrado usando AES

rango de ciclos	ciclos	subrutina	bytes cifrados
362.5 - 16,851.5	16489.0	ac_gcm_key	
16,851.5 - 17,020.5	169.0	ac_gcm_init	
17,020.5 - 35,307.5	18287.0	ac_gcm_enc	16
35,307.5 - 54,150.5	18843.0	ac_gcm_tag	

Tabla 6.13: Cifrar un mensaje de 16 bytes con GCM.

rango ciclos	ciclos	subrutina	bytes descifrados
54,610.5-71,101.5	16491.0	ac_gcm_enc	
71,101.5-71,270.5	169.0	ac_gcm_init	
71,270.5-89,556.5	18286.0	ac_gcm_dec	16
89,556.5-108,399.5	18843.0	ac_gcm_tag	

Tabla 6.14: Descifrar un mensaje de 16 bytes con GCM.

rango de ciclos	ciclos	subrutina	Bytes cifrado
162.5 -4,701.5	4539.0	ac_ccm_key	
4,701.5 -21,709.5	17008.0	ac_ccm_init	
21,709.5-30,406.5	8697.0	ac_ccm_data	
30,406.5-30,503.5	97.0	ac_ccm_data	
30,503.5-39,077.5	8574.0	ac_ccm_data	
39,077.5-56,062.5	16985.0	ac_ccm_enc	16
56,062.5-73,230.5	17168.0	ac_ccm_enc	8
73,230.5–73,335.5	105.0	ac_ccm_tag	

Tabla 6.15: Cifrar un mensaje de 24 bytes con CCM

rango de ciclos	ciclos	subrutina	bytes descifrados
73,482.5 - 78,023.5	4541.0	ac_ccm_key	
78,023.5 - 95,031.5	17008.0	ac_ccm_init	
95,031.5 - 103,728.5	8697.0	ac_ccm_data	
103,728.5 - 103,825.5	97.0	ac_ccm_data	
103,825.5 - 112,399.5	8574.0	ac_ccm_data	
112,399.5 - 129,384.5	16985.0	ac_ccm_dec	16
129,384.5 - 146,552.5	17168.0	ac_ccm_dec	8
146,552.5 - 146,657.5	105.0	ac_ccm_tag	

Tabla 6.16: Descifrar un mensaje de 24 bytes con CCM

El número de ciclos por byte para descifrar y cifrar 16 byte es: para el GCM $\frac{18286}{16} \leftarrow 1142$ ciclos por byte; para el CCM $\frac{16985}{16} \leftarrow 1061.5625$ ciclos por byte ; y por último para el AES se

demora en el cifrado $\frac{8{,}315.0}{16} \leftarrow 519$ ciclos por byte y para descifrar $\frac{11{,}445.0}{16} \leftarrow 715.3125$ ciclos por byte.

Las subrutinas del AES manejan bloques con una longitud de 16 bytes para C, M y K (texto cifrado, texto claro y la llave). La multiplicación en GCM se lleva a cabo en 7876 ciclos y la reducción en 1502 ciclos, estas dos operaciones están definidas en la biblioteca RELIC al igual que las subrutinas ac_ccm_x y ac_gcm_x. Las subrutinas en ensamblador necesarias para realizar el banco de pruebas del AES, CCM y GCM, son basadas en el código fuente de RELIC.

Las operaciones de multiplicación con Karatsuba tarda 2404 ciclos, la reducción en ensamblador dura 356 ciclos y la reducción con VHDL demora 94 ciclos. Estas versiones no son compatibles con las versiones basadas en RELIC.

Las tablas 6.17 y 6.18 muestran cada prueba del GCM: la versión 0 se basa en las funciones de reducción y multiplicación basadas en la biblioteca RELIC, la versión 1 realiza la reducción basada en el pseudocódigo 19 realizado por una subrutina en ensamblador y la multiplicación basada en Karatsuba que utiliza un componente descrito en VHDL que realiza una multiplicación de 16 por 16 bits, la última versión difiere a la anterior al utilizar la reducción del pseudocódigo 19 por medio de un componente descrito en VHDL.

	GCM 0	GCM 1	GCM 2
Ciclos por byte	1142.9375	731.75	715.375

Tabla 6.17: Ciclos por byte que demora las pruebas del GCM para el cifrado y descifrado.

	GCM 0	GCM 1	GCM 2
ac_gcm_key	16490	13003	13003
ac_gcm_init	169	169	169
ac_gcm_enc/dec	18287	11708	11446
ac_gcm_tag	18843	12265	12002

Tabla 6.18: Número de ciclos que dura cada subrutina para las distintas versiones del GCM. Detalles en el texto.

Uso de memoria

La tabla 6.19 resume el uso de memoria para cada banco de pruebas: comunicación serial SERIAL, multiplicación MUL, modulación de ancho de pulso PWM, Perro guardián WATCH DOG, AES, CCM y GCM. Las constantes (ROM) y las variables (RAM) son localidades en memoria de datos con palabras de datos de 16 bits. Las instrucciones (ROM) son palabras de 29 bits.

Pruebas	Variables	Constantes	Instrucciones
SERIAL	46	0	174
RELOJ	4	0	253
MUL	210	0	708
PWM	1	0	85
WATCH DOG	0	0	64
AES	420	2304	1039
CCM	470	2304	1603
GCM	780	2560	1993

Tabla 6.19: Uso de memoria para cada prueba

Para la pila en memoria de datos se establecen 48 localidades extras de memoria de datos (variables) que sirven para la llamada a subrutinas. La prueba del GCM descrita en la tabla anterior es la basada por las funciones de la biblioteca RELIC, las otras dos versiones manejan 2745 instrucciones, 2560 constantes y cerca de 1024 variables.

Reporte de tiempo

Para obtener el reporte Static Timing, se necesitó correr la implementación con el ISE WEB PACK de XILINX versión 10. En resumen la tabla de frecuencia y periodos se ve en la tabla 6.20.

Dispositivo/Paquete/Velocidad	Periodo mínimo	Frecuencia máxima
xc5vlx50t,ff1136,-1	79.530ns	12.574MHz
xc3s1600e,fg320,-4	148.554ns	6.732MHz

Tabla 6.20: Resultados obtenidos durante la implementación

Capítulo 7

Conclusiones

Se diseñó un microcontrolador que puede ser utilizado en aplicaciones donde los recursos sean limitados y solo para atender una única tarea. Se validó un diseño por un conjunto de simulaciones, cada banco de pruebas utiliza uno o varios componentes del dispositivo. Cada banco de pruebas consiste en varios archivos en lenguaje ensamblador que componen un programa, las pruebas realizadas son: (1) reloj binario, (2) modulador de ancho de pulso, (3) comunicación serial, (4) perro guardián, (5) Karatsuba, (6) el estándar de cifrado por bloques AES y (7) dos modos de operación de encriptación autenticada CCM y GCM.

El conjunto de simulaciones de prueba validan su uso en una comunicación serial, entrada y salida de datos por puertos paralelos, con modulación de ancho de pulso, en el uso de temporizador o con un manejo de los periféricos por medio de interrupciones.

La memoria RAM para todos las pruebas fue almacenada en la página cero, las demás páginas almacenaron las constantes que se necesitaron en las aplicaciones de cifrado, la figura 7.1 muestra todas las constantes y variables que inician desde la localidad 0x0030 (48 en base 10). Al tener todos los registros de función especial mapeados en localidades de memoria (0x0000 - 0x002F, es decir, de la 0 hasta la 47) fue posible acceder a los periféricos, la figura 7.2 enlista los registros mapeados en las tres páginas de memoria, la primera y segunda página comparte la misma interfaz con los registros de función especial, mientras que la última es distinta.

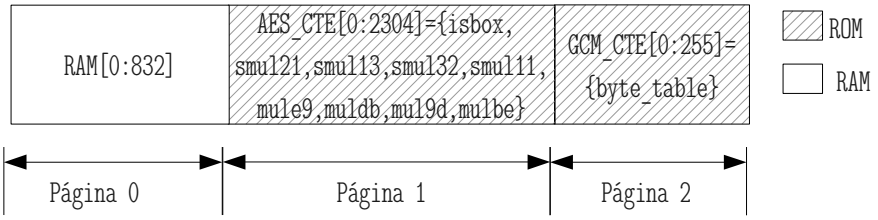


Figura 7.1: Organización de las constantes (ROM) y variables (RAM) de la memoria de datos.

Número de página	
Status	Reg 128[0]
Limite contador del perro guardián	Reg 128[1]
Configuración del perro guardián	Reg 128[2]
Mascara de la interrupción	Reg 128[3]
Dirección de subrutina para la interrupción 0	Reg 128[4]
Dirección de subrutina para la interrupción 1	Reg 128[5]
Dirección de subrutina para la interrupción 2	Reg 128[6]
Dirección de subrutina para la interrupción 3	Reg 128[7]
Dirección de subrutina para la interrupción 4	Reg 256[0]
Dirección de subrutina para la interrupción 5	Reg 256[1]
Dirección de subrutina para la interrupción 6	Reg 256[2]
Dirección de subrutina para la interrupción 7	Reg 256[3]
Dirección de subrutina para la interrupción 8	Reg 256[4]
Dirección de subrutina para la interrupción 9	Reg 256[5]
Dirección de subrutina para la interrupción 10	Reg 256[6]
Dirección de subrutina para la interrupción 11	Reg 256[7]
Dirección de subrutina para la interrupción13	Reg 256[8]
Dirección de subrutina para la interrupción 14	Reg 256[9]
Configuración del puerto paralelo A	Reg 256[10]
Configuración del puerto paralelo B	Reg 256[11]
Configuración del puerto paralelo C	Reg 256[12]
Configuración del puerto paralelo C	Reg 256[13]
Valor del puerto paralelo A	Reg 256[14]
Valor del puerto paralelo B	Reg 256[15]
Valor del puerto paralelo C	Sin uso
Valor del puerto paralelo D	Sin uso
Limite del temporizador 0	Sin uso
Limite del temporizador 1	Sin uso
Limite del temporizador 2	Sin uso
Limite del temporizador 3	Sin uso
Divisor de frecuencia de los temporizadores	Sin uso
Divisor de frecuencia del puerto serie	Sin uso
Byte transmitido por el puerto serie	Sin uso
Byte recibido por el puerto serie	Sin uso
Divisor de frecuencia de la señal sierra del PWM	Sin uso
Limite de la señal sierra del PWM	Sin uso
Configuración del PWM	Sin uso
Registro de configuración 0	Sin uso
Registro de configuración 1	Sin uso
Configuración de los temporizadores	Sin uso
Operando A de la multiplicación polinomial	Sin uso
Operando B de la multiplicación polinomial	Sin uso
Configuración de la multiplicación polinomial	Sin uso
Resultado[0:15] de la multiplicación	Sin uso
Resultado[31:16] de la multiplicación	Sin uso
Sin uso	Sin uso
Página 0 y Página 1	Página 2

Sin uso

Registros de función especial

Figura 7.2: Registros de función especial mapeados en memoria

Uno de los alcances al realizar el diseño es que sea simple, para que llegue a ser configurado en dispositivos lógicos programables como los FPGA donde se puede personalizar añadiendo o eliminando componentes para cualquier aplicación predeterminada.

Las expresiones regulares no son suficientes al utilizarse en la construcción de compiladores de lenguajes de alto nivel, pero para la creación de un lenguaje de bajo nivel como lo es el lenguaje ensamblador es más que suficiente, ya que cada línea de un archivo en lenguaje ensamblador maneja toda la información necesaria para la generación del programas en lenguaje de máquina, la información adicional que emplea el ensamblador se logra con el uso de directivas o etiquetas. El software creado para este proyecto empleó el lenguaje de programación *python* que es un lenguaje interpretado, la biblioteca estándar de este lenguaje ofrece un módulo para el reconocimiento de expresiones regulares, llamado *re*, que fue de gran ayuda para la construcción del ensamblador.

El conjunto de instrucciones no es difícil aprenderlo, es realmente práctico en aplicaciones pequeñas, menores a las quinientas doce instrucciones, pero es frustrante la creación de programas largos, se complica la depuración de errores y el mantenimiento de aplicaciones desarrolladas solamente con el lenguaje ensamblador. Pero aun así este lenguaje es de mejor ayuda que solo codificar directamente en lenguaje de máquina, siendo una tarea sumamente complicada el desarrollo de programas con este lenguaje que es solo comprendido por los procesadores.

El código fuente del proyecto esta disponible en:
https://sites.google.com/site/tesis2013microcontoladorvhdl/

7.1. Trabajo a futuro

Se contempla que aún podrían realizarse muchas mejoras al diseño propuesto. Entre estas tenemos:

Se podrían agregar otros periféricos al diseño, tales como un puerto I2C o USB. También podría anexarse un puerto Ethernet o BlueTooth.

Se podría crear el lenguaje de alto nivel para el diseño, con la finalidad de reducir el tiempo de codificación, que ahora mismo es arduo en ensamblador.

También se podría elaborar un emulador y depurador para el ensamblador.

Ahora mismo el diseño realizado podría usarse en aplicaciones prácticas, como un robot seguilíneas, y para impartir cursos educativos de arquitectura de computadoras. Esta es la contribución principal de este trabajo de tesis.

Una manera de mejorar el rendimiento seria modificar el diseño al utilizar la segmentación donde debe tenerse en cuenta la dependencia de datos, ya se tiene un conjunto de pruebas suficiente para poder comparar en futuros trabajos donde se realicen modificaciones al diseño de éste proyecto.

Para reducir el número de ciclos por byte en el cifrado en bloque se podría realizar un módulo en VHDL del AES, para realizar el cifrado y descifrado en menor tiempo.

Bibliografía

[1] F. E. V. Pérez and R. P. Areny. *Microcontroladores, Fundamentos y Aplicaciones con PIC*. ALFAOMEGA marcombo, 2007.

[2] B. B. Brey. *Los Microprocesadores Intel 8086/8088, 80186, 80286, 80386, 80486 Arquitectura, programación e interfaces 3ra edición*. Prentice Hall, 1995.

[3] B. Parhami. *Arquitectura de computadoras, de los microprocesadores a las supercomputadoras*. Mc Graw-Hill, 2007.

[4] A. Zamudio Vissuet. *Diseño e Implementación de un microprocesador RISC en VHDL*. Tesis de Maestría en Ciencias en Ingeniería Eléctrica, Centro de Investigación y de Estudios Avanzados del IPN, CINVESTAV-IPN, Unidad Zacatenco, Mayo 2003.

[5] G. E. Santana Hernández. *Diseño de un procesador usando el lenguaje de descripción de hardware*. Tesis de Maestría en Ciencias en Ingeniería Cómputo, Centro de Investigación en Computación, CIC-IPN, México DF., Marzo 2004.

[6] University of California Dalton Project. Model LEON 2. http://www.cs.ucr.edu/ dalton/leon/. Consultada el 10 de enero del 2013.

[7] M.B.I. Raez, M. S. Islam, and M. S. Sulaiman. A single clock cycle MIPS RISC processor design using VHDL. In *International Conference on Semiconductor Electronics, 2002. Proceedings. ICSE 2002. IEEE*, pages 199 – 203, Dec. 2002.

[8] N. Joseph and K. Sankarapandiammal. FPGA based Implementation of High Performance Architectural level Low Power 32-bit RISC Core. In *International Conference on Advances in Recent Technologies in Communication and Computing, 2009. ARTCom 09.*, pages 53 – 57, Oct 2009.

[9] J. Borcsok, A. Hayek, and M. Umar. Implementation of a 1oo2-RISC-architecture on FPGA for safety systems. In *Proceedings of the 2008 IEEE/ACS International Conference on Computer Systems and Applications*, AICCSA '08, pages 1046–1051, Washington, DC, USA, 2008. IEEE Computer Society.

[10] X. Tieju and Z. L. Fang. 16-bit Teaching Microprocessor Design and Application. In *International Symposium on Date of Conference IT in Medicine and Education, 2008. ITME 2008. IEEE*, pages 160 – 163, Dec. 2008.

[11] HT-Lab FPGA/VHDL/SystemC/Embedded. 8088 IP in VHDL. http://www.ht-lab.com/freecores/cpu8086/cpu86.html. Consultada el 13 de enero 2013.

[12] ht lab. MON88 Debug Monitor and Tiny Bios for the 8088/8086 Processor. http://www.ht-lab.com/freeutils/mon88/mon88.html. Consultado el 30 de abril del 2013.

[13] ALTERA. a8259 Programmable Interrupt Controller Data Sheet. http://extras.springer.com/2001/978-0-306-47635-8/ds/ds8259.pdf. Versión 1 in Julio 1997.

[14] V. Martí. Take Over The Galaxy with GitHub. https://github.com/blog/1098-take-over-the-galaxy-with-github. Consultada 11 de enero 2013.

[15] Mojang. DCPU-16 Documentation. http://dcpu.com. Consultada 12 de enero 2013.

[16] R. Gal, A. G. Krakow, M. Frankiewicz, and A. Kos. FPGA implementation of 8-bit RISC microcontroller for embedded systems. In *International Conference Date of Conference Mixed Design of Integrated Circuits and Systems (MIXDES), 2011 Proceedings of the 18th*, pages 323 – 328, June 2011.

[17] E. Ayeh, K. Agbedanu, Y., O. Adamo, and P. Guturu. FPGA Implementation of an 8-bit Simple Processor. In *Region 5 Conference, Kansas City, MO, 2008 IEEE*. IEEE Computer Society, 2008.

[18] XILINX. *PicoBlaze 8-bit Microcontroller*. http://www.xilinx.com/products/intellectual-property/picoblaze.htm. Consultada el 9 de octubre del 2012.

[19] R. Silverman and M. J. Melanie. Design of a pedagogical assembly language and classroom experiences. *J. Comput. Sci. Coll.*, 23(4):208–214, apr 2008.

[20] Dr. Robert Silverman. SC123 (tm) Computer System. http://www.cs.csustan.edu/ rrsilver/html/sc123.html. Consultado el 20 de febreo del 2013.

[21] A. Koltes and J.T. O'Donnell. A framework for FPGA functional units in high performance computing. In *IEEE International Symposium on Parallel and Distributed Processing*, 2010. Proceedings ISBN: 9781424465330(c) 2010 IEEE.

[22] Conrado P. L. Gouvêa and Julio López. High speed implementation of authenticated encryption for the msp430x microcontroller. In *Proceedings of the 2nd international*

conference on Cryptology and Information Security in Latin America, LATINCRYPT'12, pages 288–304, Berlin, Heidelberg, 2012. Springer-Verlag.

[23] Conrado P. L. Gouvêa. Authenticated encryption for the msp430. http://conradoplg.cryptoland.net/software/authenticated-encryption-for-the-msp430/. Consultado el 7 de marzo del 2013.

[24] J. L. Dávila A. C. Infante, J. I. H. Pérez and J. L. R. Martín. *Problemas de fundamentos y estructuras de computadores*. Prentice Hall, 2009.

[25] A. S. Tanenbaum. *Organización de computadoras, un enfoque estructurado*. Prentice Hall, 4ta. edition, 2000.

[26] B. A. Forouzan. *Introducción a la ciencia de la computación de la manipulación de datos a la teoría de la computación*. Thomson, 2003.

[27] F. Remiro E. Palacios and L. J. López. *Microcontrolador PIC16F84 Desarrollo de proyectos*. Alfaomega Ra-Ma, 2009.

[28] K.C. Louden. *Construcción de compiladores: principios y práctica*. Ciencias e Ingenierías. Thomson, 2004.

[29] J.R. Catalán. *Compiladores : teoría e implementación*. RC Libros- Alfaomega, 2010.

[30] Pedro Isasi, Paloma Martínez, and Borrajo Daniel. *Lenguajes, gramáticas y autómatas: un enfoque práctico*. Addison-Wesley, 1997.

[31] Aho, Lam, Sethi, and Ullman. *Compiladores principios, técnicas y herramientas Segunda edición*. Pearson Addison-Wesley, 2006.

[32] J.P. Deschamps. *Hardware Implementation of Finite-Field Arithmetic*. McGraw-Hill professional engineering: Electronic engineering. Mcgraw-hill, 2009.

[33] Cuauhtemoc Mancillas López. *Implementación Eficiente en Hardware Reconfigurable de Esquemas de Cifrado Entonados*. Tesis de Maestría en Ciencias en Ingeniería Eléctrica, Centro de Investigación y de Estudios Avanzados del IPN, CINVESTAV-IPN, Unidad Zacatenco, Noviembre 2007.

[34] A. F. Sabater, L. H. Encinas, A. M. Muñoz, F. M. Vitini, and J. M. Masqué. *Criptografía, protcción de datos y aplicaciones Guia para estudiantes y profesionales*. Alfaomega Ra-Ma, 2012.

[35] D. F. Aranha and C. P. L. Gouvêa. Relic-toolkit relic is an efficient library for cryptography. http://code.google.com/p/relic-toolkit/. consultada el 7 de marzo del 2013.

[36] D.G. Maxinez and J. Alcalá. *El arte de programar sistemas digitales*. CECSA, 2002.