PLATAFORMA PARA LA EMULACIÓN Y RECONFIGURACIÓN DE ARQUITECTURAS CISC Y RISC

ALFREDO GUALDRÓN GAMARRA
JOSE PABLO PINILLA GOMEZ

UNIVERSIDAD PONTIFICIA BOLIVARIANA
ESCUELA DE INGENIERÍA ELECTRÓNICA
FACULTAD DE INGENIERÍAS
BUCARAMANGA
2012

PLATAFORMA PARA LA EMULACIÓN Y RECONFIGURACIÓN DE ARQUITECTURAS CISC Y RISC

ALFREDO GUALDRÓN GAMARRA
JOSE PABLO PINILLA GOMEZ

TESIS DE GRADO

DIRECTOR:
ALONSO RETAMOSO LLAMAS

UNIVERSIDAD PONTIFICIA BOLIVARIANA
ESCUELA DE INGENIERÍA ELECTRÓNICA
FACULTAD DE INGENIERÍAS
BUCARAMANGA
2012

AGRADECIMIENTOS

Queremos extender un sincero agradecimiento a quienes hicieron parte en el desarrollo de este proyecto. Al MSc. Alonso de Jesús Retamoso por su dirección, enseñanza y entusiasmo por las líneas de investigación que fundamentan este proyecto y demás áreas de la Ingeniería Electrónica, tanto digital como analógica. Al MSc. Alex Monclou y PhD. Sayra Cristancho por su apoyo y compromiso en la construcción académica de la facultad de Ingeniería Electrónica con mira a crear y mantener enlaces internacionales. A ellos debemos nuestra participación como estudiantes de intercambio en la Universidad de Western Ontario y la experiencia adquirida durante nuestro paso por el grupo CSTAR.

Al PhD Yair Linn, quien nos aportó su conocimiento y experiencia sobre diseño digital avanzado, fundando en nosotros el interés por un área de investigación de gran campo como es la implementación de procesadores en FPGA.

Especialmente al cuerpo docente de la Universidad Pontificia Bolivariana le agradecemos la formación de calidad en los distintos campos de la Ingeniería Electrónica. A la MSc Claudia Rueda y al MSc Raúl Restrepo, a quienes debemos la formación en grupos de investigación. A todos aquellos que están involucrados en el desarrollo del Semillero ADT, docentes y estudiantes que conocen la importancia de este tipo de proyectos y quiénes serán los primeros beneficiados del mismo.

CONTENIDO

pág

LISTA DE FIGURAS

pág.

LISTA DE TABLAS

pág.

LISTA DE ANEXOS

pág.

RESUMEN GENERAL DE TRABAJO DE GRADO

TITULO:	PLATAFORMA PARA LA EMULACIÓN Y RECONFIGURACIÓN DE ARQUITECTURAS CISC Y RISC
AUTOR(ES):	Alfredo Gualdron Gamarra Jose Pablo Pinilla
FACULTAD:	Facultad de Ingenier&#65533;a Electr&#65533;nica
DIRECTOR(A):	Alonso Retamoso Llamas

RESUMEN

Este proyecto ilustra la estructura y fundamentos operacionales de la unidades centrales de procesamiento (CPU), mediante la implementación de un sistema configurable con dos procesadores, uno de arquitectura RISC (Reduced Instruction Set Computing) y otro de arquitectura CISC (Complex Instruction Set Computing), en una FPGA ((Field Programmable Gate Array) en compañía de un programa interfaz de usuario para la programación y monitoreo de cada procesador.

PALABRAS CLAVES:

Arquitectura de computadores, RISC, CISC, FPGA, Procesadores Embebidos

V° B° DIRECTOR DE TRABAJO DE GRADO

GENERAL SUMMARY OF WORK OF GRADE

TITLE: RECONFIGURABLE PLATFORM FOR THE EMULATION OF RISC AND CISC ARCHITECTURES

AUTHOR(S): Alfredo Gualdron Gamarra
Jose Pablo Pinilla Gomez

FACULTY: Facultad de Ingenier&#65533;a Electr&#65533;nica

DIRECTOR: Alonso Retamoso Llamas

ABSTRACT

This is a project planned to illustrate the structure and operational foundations of Central Processing Units, through the implementation of a configurable system with two processors, one RISC (Reduced Instruction Set Computing) and one CISC (Complex Instruction Set Computing), on an FPGA (Field Programmable Gate Array), along with a programming and monitoring user interface software.

KEYWORDS:

Computer Architecture, RISC, CISC, FPGA, Embedded Processors.

V° B° DIRECTOR OF GRADUATE WORK

INTRODUCCIÓN

Los sistemas de cómputo han alcanzado un nivel de sofisticación tan alto que el usuario promedio no necesita familiarizarse con los detalles técnicos de operación para utilizarlos eficientemente. Sin embargo, ingenieros, científicos de la computación y desarrolladores de aplicaciones necesitan conocer los principios de funcionamiento, capacidades y limitaciones de estos sistemas. Al área de la ingeniería electrónica y las ciencias de la computación encargada del estudio y diseño de procesadores se le conoce con el nombre de arquitectura de computadores.

La arquitectura de computadores abarca los tres aspectos del diseño de un procesador: el conjunto de instrucciones, la organización funcional y la implementación del hardware. A nivel de pregrado se estudian las arquitecturas básicas RISC (*Reduced Instruction Set Computing*) y CISC (*Complex Instruction Set Computing*), que ofrecen una base conceptual sobre la cual están construidos procesadores de uso comercial como: Intel IA-32 (x86), Freescale HC08, Motorola 68k, PowerPC, ARM (*Advanced RISC Machine*) y MIPS (*Microprocessor without Interlocked Pipeline Stages*).

Para proveer material de apoyo en la enseñanza de estos temas, el IEEE (*Institute of Electrical and Electronic Engineering*) organiza anualmente el concurso *Simulator Design Competition* que invita a estudiantes miembros de la sociedad a diseñar simuladores que ilustren el funcionamiento de procesadores de las arquitecturas mencionadas.

En contraste, este proyecto propone la descripción, en HDL *(Hardware Description Language)*, de un procesador de arquitectura RISC y uno CISC como parte de una plataforma implementada en FPGA (*Field Programmable Gate Array*) que permite al usuario su programación, supervisión y reconfiguración, así el usuario puede ejecutar en uno de los procesadores propuestos un programa escrito en lenguaje ensamblador con la posibilidad de observar en todo momento el estado del mismo. La arquitectura propuesta puede ser modificada para agregar nuevas funciones, optimizar procesos internos u orientarla a aplicaciones específicas ya que la descripción del hardware es abierta. Esto permite a usuarios con suficientes bases conceptuales, la emulación de prototipos con características de tecnologías como: DSP (*Digital Signal Processing*) ó Multi-núcleo, entre otros.

El contenido de este documento inicia con la formulación del problema y la

justificación de la solución de acuerdo a los objetivos trazados. El marco referencial contiene aspectos teóricos, históricos y del estado del arte que forman la base conceptual del proyecto en el área de arquitectura de computadores. Los criterios utilizados para el diseño de cada una de las arquitecturas se especifican en el diseño metodológico y por último se discuten los resultados obtenidos que validan el desarrollo de la plataforma.

1. DEFINIFICÓN DEL PROBLEMA

Los cursos de diseño y organización de computadores generalmente se ayudan de microcontroladores comerciales que facilitan el desarrollo de proyectos de aplicación, pero la información que brinda cada fabricante de su arquitectura y organización interna no cubre todos los niveles de análisis que se pueden estudiar en clase. También existen simuladores con ayudas visuales de procesadores que ilustran su funcionamiento interno pero no son útiles en aplicaciones prácticas.

Por otra parte las arquitecturas de los procesadores actuales han acumulado funciones a través de los años para preservar la compatibilidad con versiones anteriores, lo cual dificulta su comprensión; como afirman Patterson y Hennessy: “Nos ha conducido a una arquitectura que es difícil de entender e imposible de amar”[1]. Al mismo tiempo, los microcontroladores cuentan con características especiales que facilitan tareas específicas, pero su manejo requiere de bases conceptuales previas.

Una herramienta de apoyo a los cursos de diseño y organización de computadores debe contar con procesadores completamente documentados, revelando detalles de su organización y arquitectura, que además puedan implementarse fácilmente en proyectos de aplicación.

---

[1] PATTERSON, A. David; HENNESSY , John L. Computer Organization and Design. The Hardware/Software Interface. 3rd Edición. Morgan Kaufmann Publishers, Elsevier, 2005. p. 136

2. JUSTIFICACIÓN

A falta de una herramienta que cumpla con las necesidades de aplicabilidad e información detallada, se propone el diseño de dos procesadores descritos en Verilog HDL, un software con funciones de ensamblador, un sistema de comunicación para ver el estado del procesador que se implemente y una guía de usuario que explique en detalle su organización interna.

De esta forma es posible aprovechar tanto el funcionamiento práctico que tienen los microcontroladores al momento de desarrollar proyectos y aplicaciones sin la necesidad de comprar circuitos integrados distintos para cada aplicación ni instrumentos de programación; como la capacidad de observar el estado interno del procesador en cada instrucción del modo que lo hacen los simuladores. Incluso aquellos que ya tienen un dominio conceptual en el tema pueden rediseñar o modificar cada procesador, gracias a la disponibilidad de la descripción en Verilog y la documentación de cada uno de ellos, lo que lo hace un sistema portable y de fácil implementación en distintas FPGA.

En contraste con desarrollos similares, comerciales o académicos, los objetivos de este proyecto resaltan la necesidad de diferenciar distintas arquitecturas, de forma que el usuario sea quien reconozca las ventajas y desventajas de cada una con el fin de reforzar criterios clave al momento de diseñar una arquitectura propia o trabajar eficientemente con una existente.

3. OBJETIVOS

3.1. OBJETIVO GENERAL

Diseñar un emulador de procesadores con arquitecturas tipo RISC y CISC descritos en Verilog para FPGA y un software en Visual Basic para supervisar el estado del procesador que esté implementado.

3.2. OBJETIVOS ESPECÍFICOS

- Diseñar e implementar un procesador de arquitectura RISC en circuitos de lógica programable.
- Diseñar e implementar un procesador de arquitectura CISC en circuitos de lógica programable.
- Programar una Interfaz Gráfica de Usuario (GUI) que permita el seguimiento del procesador implementado y la traducción de programas en lenguaje ensamblador a código de máquina.
- Elaborar una guía de usuario que facilite la comprensión y manejo de la plataforma.

4. ESTADO DEL ARTE

Actualmente existen aplicaciones exitosas de este tipo de plataformas en el ámbito de la investigación y como recursos académicos de entre las cuales se resaltan los siguientes proyectos:

La publicación "*Processor, Assembler, and Compiler Design Education using an FPGA*" desarrollado por Nakano K de la Universidad de Hiroshima en el 2008, hace un reporte del diseño de dos cursos ofrecidos en esa universidad utilizando una plataforma conformada por un procesador de 16 bits llamado TINYCPU, un ensamblador TINYASM y un compilador del lenguaje C llamado TINYC. Su diseño es sencillo y compacto para que los estudiantes puedan entender todo el diseño fácilmente y así aprender los conceptos básicos de sistemas embebidos, que incluyen: arquitectura de computadores, programación, diseño de ensambladores y compiladores.

De la Universidad de Guanajuato la publicación "*8-bit CISC Microprocessor Core for Teaching Applications in the Digital Systems Laboratory*" desarrollado por Romero Troncoso en el 2006. En esta se resalta la importancia del diseño de "*IP cores*" en un nuevo enfoque para implementación de los mismos en dispositivos *System on a Chip* (SoC), con el desarrollo de un microprocesadores CISC de 8 bits. Este diseño es totalmente abierto lo que permite la configuración de su estructura para distintas aplicaciones y su implementación en casi cualquier familia de dispositivos FPGA. La plataforma diseñada con este proyecto contiene también un ensamblador para generar los programas en archivos VHDL que describen memorias ROM.

"*Design and Implementation of a 32bit RISC Processor on Xilinx FPGA*" de la Universidad de Fayoum en Egipto desarrollado por Waelm Elmedany y Khalida Alkooheji en el 2008, documenta el diseño e implementación de un procesador RISC de 32 bits para FPGAs Xilinx, demostrando la estructura interna y funcionamiento de su unidad de control, ruta de datos y memoria de instrucciones; con simulaciones de cada módulo de hardware.

El *paper* de la Universidad de York en el Reino Unido, "*A Minimal CISC Processor Architecture for Field Programmable Gate Arrays*" desarrollado por Michael Freeman en el 2004, contempla las ventajas de describir un procesador CISC para FPGA como alternativa a los *IP cores* ofrecidos por el fabricante XIlinx que son

procesadores de arquitectura RISC. En él se describe el funcionamiento de la unidad de control implementada y se realiza una comparación entre las distintas arquitecturas.

5. MARCO TEÓRICO

5.1. PROCESADORES

El procesador o CPU (*Central Processing Unit*) es la parte activa de un computador digital. Tiene como función principal producir resultados a partir de datos de entrada, realizando operaciones aritméticas, lógicas, de transferencia y de control en interfaces de entrada y salida; con datos almacenados en registros internos o memoria. Inicialmente la construcción de estos dispositivos en circuitos integrados no incluía la unidad de memoria. En 1971 Intel desarrolló el dispositivo 4004, la primera CPU que incluía memoria en el mismo chip y que tomó el nombre de microprocesador.[2]

Los microcontroladores se encuentran en dispositivos digitales de uso cotidiano como teclados, televisores, equipos de sonido y demás sistemas digitales; su costo y rendimiento son relativamente bajos, comparados con procesadores de aplicación general (Computadores personales), pero adecuados para las condiciones de trabajo y la demanda de las tareas que llevan a cabo. Desde el momento de su implementación utilizan un programa fijo que no cambia a lo largo de su ejecución, el programador lo debe modificar para cumplir nuevas tareas o corregir su comportamiento. Existe una gran variedad de microprocesadores, cada uno diseñado especialmente para una tarea específica[3].

5.1.1. Unidades de un microprocesador

Un microprocesador está compuesto por cuatro subsistemas o unidades básicas: Unidad de Control (CU), Unidad Lógico-Aritmética (ALU), Unidad de Memoria (MU) y Unidad de Entradas/Salidas (IOU)[4], cómo se ilustra en la Figura 1**.** A la interconexión entre la unidad de control y los demás subsistemas se le conoce

---

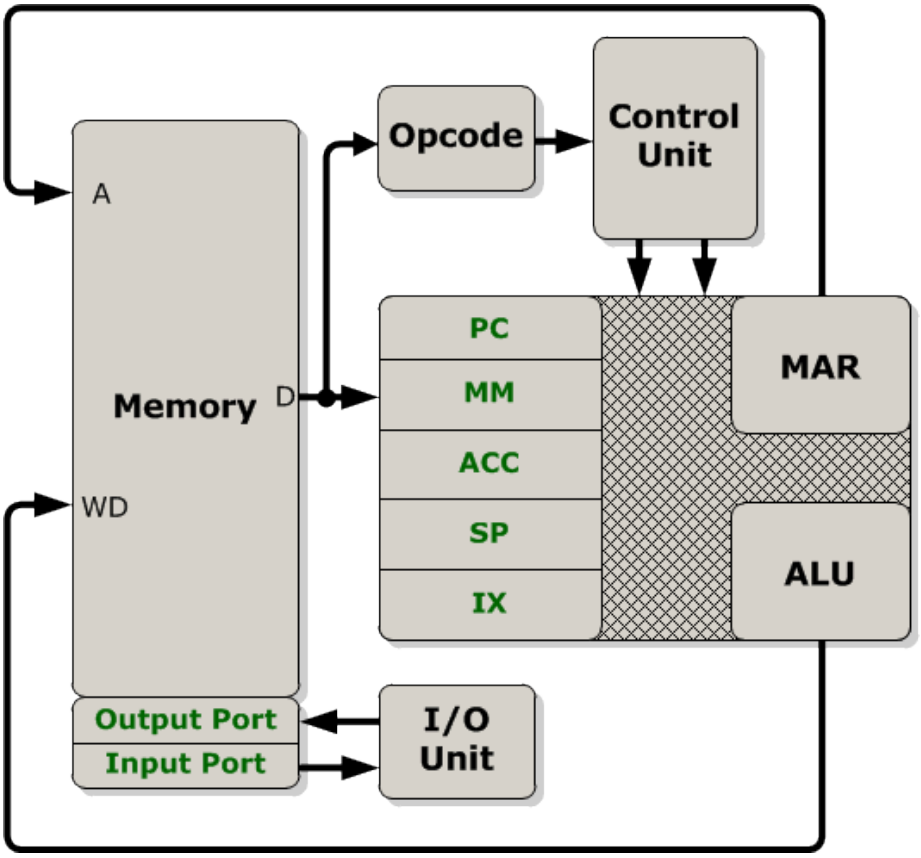
[2] PATTERSON. Op. cit., p. 20.

[3] SHIVA ,Sajjan G. Computer Organization, Design, and Architecture. 4a Edición. CRC Press. 2008. p. 705

[4] SHIVA ,Sajjan G. Computer Organization, Design, and Architecture. 4a Edición. CRC Press. 2008. p. 51

como Ruta de Datos (*Data Path*) y está compuesta por multiplexores y registros, los cuales transportan las señales de control o datos a las demás unidades de acuerdo a las instrucciones previamente almacenadas en la unidad de memoria. Cada instrucción leída secuencialmente de la Unidad de Memoria indica la operación que se va a llevar a cabo, ya sea de asignación (lógica o aritmética) o de control de flujo de programa (saltos). Estas instrucciones se componen de al menos dos campos: Código de Operación (*Opcode*) y dirección o representación del Operando.

Figura 1. Diagrama de bloques de un procesador



5.1.1.1. Unidad de control

La unidad de control es una máquina de estados que genera las señales de control en el orden apropiado, que requieren los demás subsistemas del procesador para ejecutar las operaciones de cada instrucción. Estas señales dependen del *Opcode* y del estado actual del procesador.

La unidad de control pasa por al menos tres fases principales: *Fetch*, en la cual se lee la instrucción de la unidad de memoria; *Decode*, en la que se decodifican los campos de la instrucción leída y *Execute*, dónde se efectúa la función decodificada. Cada fase puede estar compuesta de una secuencia de micro-instrucciones las cuales pueden ser: Una transferencia de registro a registro, una transferencia de ALU a registro o lectura/escritura de memoria.

Existen dos tipos de Unidades de control[5]:

- Unidad de Control por Hardware HCU (*Hardwired Control Unit*): Las señales de control son generadas por la implementación de una lógica circuital a base de compuertas y *flip-flops*.

- Unidad de Control Microprogramada MCU (Microprogrammed Control Unit): Las señales de control son generadas a partir de la decodificación de microoperaciones. Las secuencias de micro-instrucciones para cada operación se guardan en una memoria de sólo lectura (ROM), conformando el microprograma.

La siguiente tabla es una comparación realizada por Michael J. Flynn en la que se muestran las diferencias más importantes entre las unidades de control descritas.

[5] SHIVA. Op. cit., p. 316

Tabla 1. Tabla comparativa entre HCU y MCU

	HCU	MCU
Velocidad	Rápido	Lento
Eficiencia en el área del chip	Requiere menos área	Requiere más área
Implicaciones al modificar o agregar funciones	Cambios en Hardware	Actualizar Micro-código
Manejo de instrucciones complejas-extensas	Hardware Dedicado	Secuencia de Micro-Instrucciones
Dispositivos que los usan	En su mayoría procesadores RISC	Mainframes, x86
Tamaño del set de instrucciones	Menos de 100	Más de 100
Tamaño de la ROM	N/A	2k a 10k

Fuente: FLYNN, Michael J. Computer Architecture: Pipelined and Parallel Processor Design. Jones & Bartlett Publishers, Inc. 1995. p. 9

5.1.1.2. Unidad lógico-aritmética

La Unidad Lógico-Aritmética es la encargada de realizar todas las operaciones del procesador. Las operaciones básicas que debe ejecutar una ALU son aritméticas: suma y resta; y lógicas AND, OR, XOR. Estas operaciones pueden ser complementadas con funciones de corrimiento o más complejas, como multiplicación y división.

5.1.1.3. Unidad de memoria

La unidad de memoria es la que contiene los dispositivos de almacenamiento en los que se guardan las instrucciones y los datos, la estructura de esta unidad depende de la arquitectura de memoria que se utilice: La arquitectura Von Neumann, que organiza el programa y los datos de acceso aleatorio (RAM) en una sola memoria; o la arquitectura Harvard, que los divide en memoria de solo lectura (ROM) para las instrucciones y RAM para los datos. Actualmente la arquitectura

de la unidad de memoria tipo Harvard no utiliza memorias independientes para datos e instrucciones pero si tiene acceso simultaneo a ellas utilizando caminos y buffers separados.[6]

Una unidad de memoria se separa de manera virtual, principalmente para los siguientes usos:

- Programa: El código de programa del microprocesador tendrá reservado un espacio en memoria y la dirección inicial debe corresponder con la de inicialización o reinicio del sistema.

- Memoria de Acceso Aleatorio (RAM): Esta unidad de memoria es utilizada por el procesador para almacenar datos de uso frecuente y aleatorio durante la ejecución de programas. Algunas arquitecturas usan como RAM las localidades de memoria que pueden ser direccionadas con ocho bits.

- Stack ó Pila: La pila corresponde a un rango en memoria dedicado a guardar la información de registros de uso general de manera que permita su reutilización en subrutinas. Para esto, la pila se debe leer y escribir de manera FILO (*First In Last Out*) y utilizando un Puntero de Pila o *Stack Pointer* (SP) que conserva la dirección del último dato disponible para lectura.

- Registros Mapeados en Memoria (MM): Son registros que contienen banderas y datos que cumplen funciones de configuración para funciones externas al procesador como puertos y temporizadores, entre otros. Estos registros se direccionan de la misma forma en que se leen y escriben datos de memoria.

- Reservado: Son espacios de memoria que se reservan para almacenar

[6] SHIVA. Op. cit., p. 57

datos necesarios para el funcionamiento interno del microprocesador, tales como direcciones constantes y subrutinas de control de interrupciones.

5.1.1.4. Unidad de entradas y salidas

Es una interfaz de hardware que permite la entrada y salida de datos hacia y desde la ruta de datos del microprocesador. Esta unidad está conformada por registros paralelos del mismo tamaño del bus de datos y se utiliza para comunicar al microprocesador con otros dispositivos o periféricos como teclados y pantallas.

Este tipo de sistemas se comunican por interrupciones o por requerimientos. A modo de ejemplo, al presionar la tecla de un carácter en el teclado se debe ejecutar la parte del programa que identifica la tecla que fue oprimida, esto se hace interrumpiendo la ejecución actual de código o en el momento que el mismo programa consulte el estado de esa señal de entrada. Cuando el programa se encarga de revisar el estado de las señales se dice que ese evento se va a ejecutar de acuerdo a un requerimiento.

5.1.1.5. Interrupciones y temporizadores

Una interrupción es un evento aleatorio o periódico que necesita una acción rápida. Cuando una interrupción ocurre el flujo normal de programa se altera para atender la subrutina de interrupción inmediatamente despúes de que se ejecute la instrucción actual. Una interrupción o excepción ocurre generalmente por tres condiciones: un estado ilegal de la CPU, un resultado de una operación aritmética o cambios en el puerto de entrada; aunque también es posible generar interrupciones desde el software. La implementación de una interrupción o un requerimiento depende de la prioridad con la que se quiere reconocer un evento dentro del algoritmo.

Cuando se genera una interrupción, el microprocesador debe determinar la fuente de la interrupción, guardar el estado actual del procesador y cargar el contador de programa con la dirección que apunta a la subrutina de la interrupción adecuada.

Una vez la interrupción ha sido atendida el procesador debe retornar al estado anterior a la interrupción. Esto se logra ejecutando una Rutina de Servicio de Interrupción (RSI) que lleva a cabo las siguientes operaciones[7]:

1. Deshabilitar temporalmente futuras interrupciones.
2. Guardar en memoria el contador de programa y registros que se modifican dentro de la interrupción.
3. Habilitar futuras interrupciones.
4. Determinar la causa de la interrupción.
5. Ejecutar la subrutina de interrupción.
6. Deshabilitar temporalmente futuras interrupciones.
7. Recuperar los valores de los registros que se guardaron en memoria.
8. Habilitar futuras interrupciones.
9. Cargar el contador de programa con el valor guardado.

En el caso de estados ilegales y excepciones imprevistas en la ejecución del código, se reinicia el contador de programa y los demás registros que conforman el estado del procesador.

Internamente, los microprocesadores pueden generar interrupciones periódicas gracias a la configuración de contadores programables llamados temporizadores. Estos contadores pueden sincronizarse con una fuente de reloj distinta a la de la CPU, siempre y cuando se mantenga la capacidad de contar con precisión un período de tiempo. Este periodo es definido por un valor que se compara con el contador para generar una bandera que lo reinicia y que puede generar eventos de interrupción. Un preescalador divide la frecuencia del reloj de funcionamiento del temporizador de tal manera que se necesiten más pulsos del reloj del sistema para incrementar en uno el conteo y de esta forma obtener periodos de tiempo más extensos.

[7] SHIVA ,Sajjan G., p. 359

5.2. ARQUITECTURA DE COMPUTADORES

El reto que enfrenta un arquitecto de computadores en el diseño de nuevos microprocesadores es el de determinar qué atributos son importantes para maximizar su rendimiento y disminuir el costo y consumo de energía. Este proceso incluye los tres niveles a diseñar en un procesador: el conjunto de instrucciones, la organización funcional, y la implementación en hardware[8].

La arquitectura del conjunto de instrucciones o ISA (*Instruction Set Architecture*) es el nivel más alto de abstracción en la descripción del hardware y software de un sistema de procesamiento. Se encarga de dar una interfaz al programador, de modo tal que pueda referirse a funciones mediante mnemónicos, independiente del hardware que las realiza. Por ejemplo: hacer una resta aritmética, sin ir al nivel de las compuertas que cumplen esa función (una suma con acarreo y un operando negado)[9].

La organización de computadores desarrolla desde el punto de vista funcional las especificaciones de rendimiento de los diferentes componentes y la interconexión entre ellos, como la estructura de las memorias, la ALU y los archivos de registros.

La decisión de implementar o no una instrucción de multiplicación hace parte de la ISA, pero es una labor de diseño organizacional decidir si la instrucción será implementada por una unidad especial de multiplicación ó por un mecanismo que haga uso repetido de un sumador; esto se determina de acuerdo a una aproximación a la frecuencia de uso de la instrucción, la velocidad de ejecución, costo y espacio físico de las dos opciones[10].

---

[8] HENNESSY, John L.; PATTERSON, David A. Computer Architecture. A Quantitative Approach. San Francisco. Morgan Kaufmann Publishers, Elsevier, 2007. p. 8

[9] PATTERSON, David A.; HENNESSY, John L. Computer Organization and Design. The Hardware/Software Interface. San Francisco. Morgan Kaufmann Publishers, Elsevier, 2005.

[10] STALLINGS. Op. cit., p. 9.

Dos procesadores con el mismo set de instrucciones pero diferente organización son el AMD *Opteron* 64 y el Intel *Pentium* 4. Para el programador ambos dispositivos realizan las mismas funciones, pero no las ejecutan de la misma forma[11].

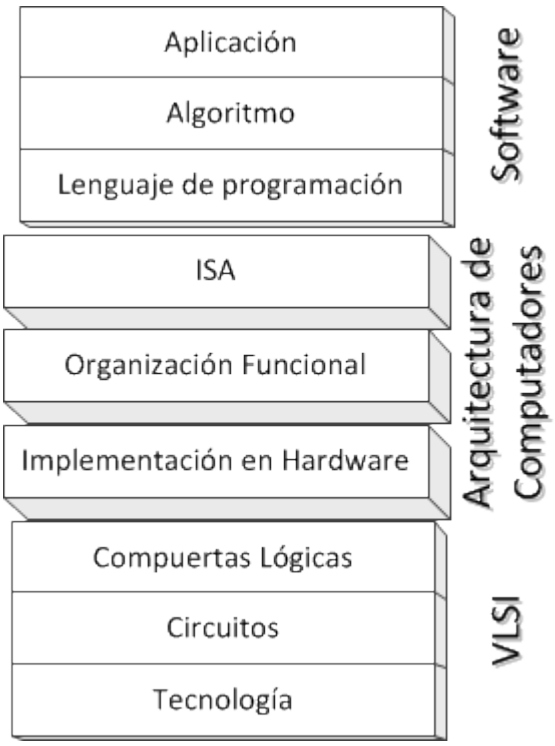
La implementación en hardware es el nivel más bajo de abstracción y se encarga del diseño circuital de la lógica del microprocesador. Determina la tecnología electrónica empleada y distintos aspectos físicos del microprocesador (frecuencias de reloj, sistemas de memoria y consumo de energía) a partir de las especificaciones de funcionamiento. Por consiguiente la implementación del hardware de un dispositivo para aplicaciones móviles difiere de uno para uso casero o de un servidor[12].

La Figura 2 ilustra todos los niveles de abstracción involucrados en el diseño de procesadores. Los niveles de más abstracción son cercanos al software, mientras que los más bajos son cercanos a las características físicas de los componentes, tema que se trata en procesos de *Very Large Scale Integration (VLSI).* La arquitectura de computares se encuentra entre estos dos extremos.

[11] HENNESSY, John L.; PATTERSON, David A. Computer Architecture. A Quantitative Approach. Op. cit. p. 12

[12] SHIVA. Op. cit., p. 59

Figura 2. Niveles de abstracción



A nivel de ISA existen dos modelos de arquitecturas principales: RISC y CISC.

5.2.1. Arquitectura CISC

5.2.1.1. Reseña histórica

Un avance notable en la industria de los microprocesadores tuvo lugar en 1973 cuando Motorola desarrolla el modelo 6800 e Intel el 8080, ambos de 8-bits, considerados posteriormente de arquitectura CISC. Este tipo de arquitectura ejecuta instrucciones complejas para minimizar la cantidad de memoria requerida, ya que en los años 70 las memorias eran costosas y de muy poca capacidad

(16KBytes tenían un costo de $500USD)[13]. Sin embargo, instrucciones complejas también significan hardware complejo y por lo tanto costoso y es por esto que desde 1951 se optó por unidades de control microprogramadas.

En 1978 Intel anuncia el microprocesador 8086 de 16-bits, que IBM adopta para sus computadores personales dos años antes de que Motorola lanzara la serie 68K, conocida por hacer parte del Apple Macintosh. Debido a la popularidad de los equipos IBM, la siguiente versión del 8086 producida por Intel en 1985, el 80386 de 32-bits, ofrece compatibilidad con la versión anterior, conservando el liderazgo en el mercado. Desde entonces todas las versiones de microprocesadores llamados de arquitectura IA-32 ó x86 son compatibles[14].

5.2.1.2. Características principales

Un procesador de arquitectura CISC dispone principalmente de una gran cantidad de instrucciones de distintos tamaños (el *Opcode* o encabezado de la instrucción indica el tipo de operación y cuantas localidades de memoria constituyen la instrucción completa), pocos registros con funciones específicas (Acumulador, Contador de programa, Registro Índice, Puntero de la Pila y Registro de estado) y una gran variedad de modos de direccionamiento que permiten operaciones entre registros, valores inmediatos y datos en memoria. Estas características dan la ventaja a los programadores de escribir códigos cortos ahorrando recursos en memoria de instrucciones, pero requieren una unidad de control extensa y más ciclos de reloj para completar las instrucciones.

1.1.1.1.1. Modos de direccionamiento

Al aumentar el tamaño de la memoria de programa y datos, el campo disponible en la instrucción para especificar la dirección dónde se encuentra el operando que

---

[13] DANDAMUDI Sivarama P., Fundamentals of computers organization and desing. Springer. 2002. p. 20

[14] HARRIS, David Money; HARRIS, Sarah L. Digital design and computer arquitecture. San Francisco. Morgan Kaufmann Publishers, Elsevier, 2007. p. 341

se quiere leer se vuelve insuficiente, es por esto que existen los modos o representaciones de direccionamiento.

- Inherente: El modo de direccionamiento inherente se refiere a aquellas instrucciones que no requieren datos adicionales o direcciones de memoria para lograr su ejecución. Esto no significa que la instrucción se ejecuta en un solo ciclo o que no tiene operandos, pero sí que toda la instrucción está contenida en el *Opcode*. Los operandos de este tipo de instrucciones pueden ser los registros del procesador.

- Inmediato: Las instrucciones de direccionamiento inmediato contienen el operando explícito en la instrucción. De esta forma se logran instrucciones de asignación con constantes. Pero a diferencia de una constante guardada en memoria de datos, este inmediato solo podrá ser utilizado en el momento de ejecutar la instrucción ya que se guarda en la localidad de memoria contigua al *Opcode*.

- Directo: El operando está especificado por una dirección absoluta contenida en la instrucción.

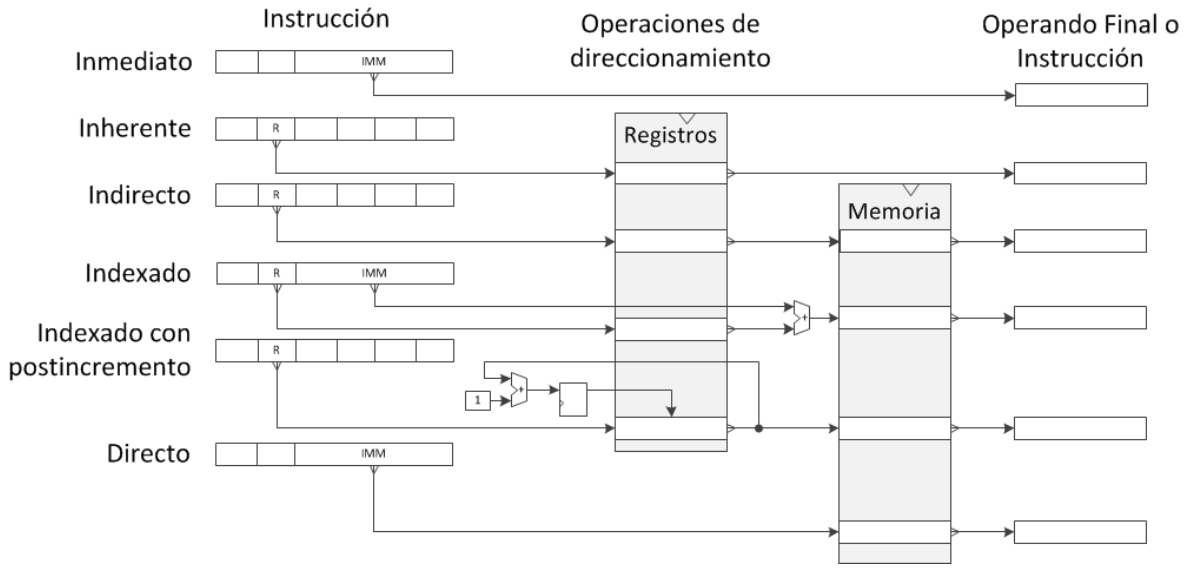
- Indirecto: El operando se lee de una dirección contenida en un registro del procesador, la instrucción en este caso especifica cuál es el registro que contiene la dirección.

- Indexado: Este modo de direccionamiento utiliza un registro comúnmente llamado IX (*Index Register*), para apuntar a una localidad de memoria base y a partir de ésta desplazarse adicionando o sustrayendo un desfase. Este desfase debe estar especificado dentro de la instrucción.

- Relativo: Este direccionamiento se utiliza en las instrucciones de salto condicionado o "*Branch*". La dirección a la que se va a saltar, si la condición se cumple, es un valor relativo a la dirección actual, es decir: la dirección siguiente es un desplazamiento, ya sea positivo o negativo, de la dirección a la que apunta el registro contador de programa. Si la condición no se cumple, se continúa la ejecución del programa con la instrucción inmediatamente siguiente.

La Figura 3 ilustra cada uno de los modos de direccionamiento explicados con anterioridad.

Figura 3. Modos de direccionamiento CISC



Fuente: NURMI, Jari. Processor Design. System-on-Chip Computing for ASICs and FPGAs. Springer. 2007. p. 9.

1.1.1.1.2. Instrucciones de tamaño variable

Una instrucción CISC está compuesta por *Opcode* y operandos. El *Opcode* tiene un ancho fijo que es decodificado por la unidad de control mientras que el número de operandos que debe tener una instrucción depende del la operación en sí y su modo de direccionamiento. Esta característica permite aprovechar la capacidad de la memoria utilizando sólo el número de localidades de memoria necesarios para llevar a cabo la operación especificada por el *Opcode*, agregando un ciclo de lectura de memoria por cada operando adicional.

5.2.1.3. Estado del arte

El ejemplo de arquitectura CISC más común es la x86 de Intel debido a su presencia en computadores personales desde 1978. Los últimos lanzamientos de Intel, la serie *Core*, ofrece una arquitectura llamada Intel 64 (de 64-bits) y múltiples procesadores para procesamiento paralelo. Por otra parte, la arquitectura 6800 también está vigente en el área de microcontroladores como el 68HC08, 68HC11, 68HC12 y 68HC16 a través de la firma *Freescale* (antes Motorola). Ambos

continúan el desarrollo de sus diseños y mantienen compatibilidad con versiones anteriores debido a las necesidades del mercado tanto en computadores personales como en la industria.

5.2.2. Arquitectura RISC

5.2.2.1. Reseña histórica

A mediados de los años 70 y comienzos de los 80 un número de estudios identificó que un programa típico contiene un 80% de instrucciones de asignación, saltos condicionales y llamados a procedimientos y que el 50% de esas operaciones son de asignaciones. Esto significó un cambio en la filosofía de diseño que implicaba optimizar la velocidad de ejecución de instrucciones frecuentes y al mismo tiempo reducir su complejidad y acceso a memoria. Entre las arquitecturas RISC comerciales están: MIPS, ARM y PowerPC; que presentan un gran potencial en dispositivos móviles gracias a su alto desempeño a menores frecuencias[15].

5.2.2.2. Características principales

Un procesador RISC dispone principalmente de un número reducido de instrucciones de tamaño fijo, pocos modos de direccionamiento y gran cantidad de registros de propósito general. Estas características permiten una decodificación sencilla de las instrucciones por la unidad de control y un uso reducido del acceso a memoria al poder realizar más operaciones registro a registro. Esto se traduce en una rápida ejecución de instrucciones y la reducción del área del circuito integrado dedicado a la unidad de control[16].

---

[15] ABD-EL-BARR, Mostafa. Fundamentals of computer organization and architecture. Wiley-Interscience. 2005. p. 22

[16] SHIVA. Op. cit., p. 439

5.2.2.3. Los cuatro principios[17]

La selección de un set de instrucciones tipo RISC requiere un balance entre el número de instrucciones y de ciclos que necesita al ejecutarse, para lograrlo David A. Patterson y John L. Hennessy articularon cuatro principios en el desarrollo de la ISA MIPS:

1. La simplicidad favorece a la regularidad.
2. El caso común debe ser rápido.
3. Lo más pequeño es más rápido.
4. El buen diseño requiere buen compromiso entre simplicidad y flexibilidad.

5.2.2.4. Estado del arte

ARM (*Advanced RISC Machine*) es una familia de microprocesadores y microcontroladores diseñados por ARM Inc. Esta compañía no fabrica los circuitos integrados pero diseña los microprocesadores y arquitecturas de múltiples núcleos y entrega estas licencias a fabricantes. Estos chips se destacan por su tamaño reducido y bajo consumo de potencia y es la arquitectura de procesador embebido más utilizada en el mundo ya que se encuentra en teléfonos inteligentes, reproductores de música y otros sistemas de requerimientos similares[18].

5.2.3. Comparación entre CISC y RISC

La Tabla 2 muestra las características más importantes que podrían presentar procesadores de cada arquitectura.

---

[17] PATTERSON, David A.; HENNESSY, John L. Computer Organization and Design. Op. cit., p. 145
[18] STALLINGS. Op. cit., p. 46

Tabla 2. Tabla de comparación entre arquitecturas CISC y RISC.

Característica	**CISC**	**RISC**
Tamaño de la Instrucción	Variable	Fijo
Registros disponibles	Acumuladores, PC, Index, *Stack Pointer*, CCR	Mayor cantidad de registros
Modos de direccionamiento	Inmediato Directo Extendido Indirecto Inherente Indexado	Inmediato Indirecto Directo Desplazado
Set de Instrucciones	Gran cantidad de instrucciones	Un número reducido de instrucciones
Tipo de operaciones	Registro - Memoria	Registro - Registro
Instrucciones de Acceso a Memoria	Mayor cantidad de Instrucciones	Guardar en memoria Cargar de memoria
Manejo de la Pila	Hardware	Software

5.3. ENSAMBLADOR

Un ensamblador es un programa que decodifica un algoritmo que haya sido escrito utilizando los mnemónicos de un set de instrucciones y etiquetas de datos de acuerdo a un formato preestablecido, para convertirlos en un arreglo de números binarios, ya que conoce los *Opcodes* y direcciones absolutas que van a tener los datos en memoria. Al código en este nivel se le llama ”Lenguaje de máquina”. El archivo generado durante el proceso de ensamble puede ser directamente implementando en el procesador escribiendo este arreglo de instrucciones en la memoria de programa.

El lenguaje de maquina es inherente a todos los microprocesadores, sin embargo la escritura de programas se hace en “lenguaje ensamblador” o de alto nivel como: C, C++, Basic, Python entre otros, para facilitar su escritura y depuración; y son

traducidos por programas de análisis de texto (compiladores y ensambladores) a lenguaje de máquina acorde a cada procesador.

5.3.1. Lenguaje ensamblador

El lenguaje ensamblador es el lenguaje de programación de más bajo nivel y está directamente relacionado con el set de instrucciones y la arquitectura de un microprocesador específico. Consiste de una secuencia de instrucciones codificadas en mnemónicos y direcciones simbólicas, donde cada instrucción está compuesta de cuatro campos principales: etiqueta, operación, operando(s) y comentarios. Estos campos simbolizan los códigos de operación, registros, y demás datos de la instrucción del procesador originalmente representada en números binarios (Lenguaje de Máquina).

Etiqueta: Es un campo opcional que sirve para representar un número constante, o una dirección específica en la memoria de programa.

Operación: Es un campo requerido para todas las instrucciones e identifica el propósito o función de la misma.

Operandos: Especifica el dato sobre el cual tendrá efecto la operación, por lo tanto este campo depende de la operación misma.

Comentarios: Es un campo opcional con el fin de brindar mejor comprensión del funcionamiento del programa.

Figura 4. Ejemplo de código en lenguaje ensamblador

```
* Delay routine:
* Delay = N x (153.6+0.36)uS for 60nS CPU clock
* For example, delay=10mS for N=$41 and 60nS CPU clock
*
*        Entry: COUNT = 0
*        Exit:  COUNT = 0; A = N
*
Label           Operation       Operand         Comments
N               EQU             $41             ;Loop constant for 10mS delay
*
                ORG             $50             ;RAM address space
COUNT           RMB             1               ;Loop counter
*
                ORG             $6E00           ;ROM/EPROM address space
DELAY           LDA             #N              ;Set delay constant
LOOPY           DBNZ            COUNT,LOOPY     ;Inner loop (5x256 cycles)
                DBNZA           LOOPY           ;Outer loop (3 cycles)
```

Fuente: FREESCALE SEMICONDUCTOR. CPU08 Central Processor Unit. 2006. p. 134

Para identificar cada campo generalmente se utilizan caracteres especiales. En la Figura 4, el símbolo ";" indica el inicio del campo comentario, los símbolo "$" y "#" indican el campo operando como una dirección o un valor numérico respectivamente, mientras que el campo operación está separado por espacios o tabulaciones del campo etiqueta.

Además del set de instrucciones disponible para un procesador, el lenguaje ensamblador cuenta con una clase de comandos especiales llamados directivas, que ayudan a facilitar la depuración y la labor de ensamble. Las directivas no son traducidas a código de máquina ya que no hacen parte del set de instrucciones del procesador, pero si pueden por ejemplo, configurar la dirección en que inicia el código de programa en la memoria (ORG), reservar bytes de memoria para una variable (RMB), asignarle un nombre a una dirección de memoria o un número (EQU), entre otras. En la imagen 4.4 observamos las directivas EQU, ORG, y RMB.

5.3.2. Proceso de ensamble[19]

El proceso de ensamble es aquel en el cual, a partir del código en lenguaje ensamblador, se genera el código de máquina que necesita el procesador para ejecutar cada una de las instrucciones escritas. El proceso de ensamble debe llevar a cabo dos funciones. La primera es generar una dirección absoluta para cada nombre simbólico (etiqueta) del programa y la segunda retornar el equivalente binario de cada instrucción. Como es proceso de dos etapas, se le conoce como ensamblador de dos "pasadas", del inglés *"Two pass assembler"*.

En la primera pasada se deben llevar a cabo las siguientes tareas:

1. Guardar las etiquetas dentro de la tabla de símbolos
2. Validar los mnemónicos
3. Interpretar directivas
4. Operar el contador de posición

La tabla de símbolos es una matriz que contiene la representación simbólica de cada etiqueta y su correspondiente valor en lenguaje de máquina.

Una instrucción en lenguaje ensamblador puede ser ejecutable o directiva, las instrucciones ejecutables son las que ejecuta el procesador, las instrucciones directivas determinan características del proceso de ensamble y no son traducidas a lenguaje de máquina.

Como una arquitectura puede tener instrucciones de tamaño variable es importante calcular cuantas localidades de memoria consume cada instrucción y en cual localidad de memoria se escribirá la siguiente instrucción, para ello se implementa un contador de posición que se actualiza cada vez que una instrucción es interpretada.

---

[19] SHIVA. Op. cit. p. 281

En la segunda pasada se llevan a cabo las siguientes tareas:

1. Evaluar los operandos de la instrucción
2. Interpretar *Opcode*
3. Resolver las referencias y direcciones

En instrucciones de tamaño variable los operandos se escriben en las localidades de memoria continuas al *Opcode*, se debe determinar de qué tamaño es el operando y qué formato tiene. El mnemónico que representa el *Opcode* se reemplaza por su equivalente binario y se resuelven las direcciones de salto.

Como resultado del proceso de ensamble se obtiene un archivo en código de máquina, con el formato necesario para ser cargado en la memoria de instrucciones del procesador.

5.4. ARREGLO DE COMPUERTAS PROGRAMABLES EN CAMPO (FPGA)

Son circuitos integrados digitales de lógica programable, conformados por: bloques de lógica configurable (CLB), Macro-celdas y matrices de interruptores programables (PSM). Los CLB pueden configurarse con cualquier lógica digital (secuencial o combinacional), las Macro-celdas son componentes embebidos, diseñados a nivel de transistores para cumplir funciones específicas, como PLL (*Phase Locked Loop*) ó multiplicadores; y las PSM realizan la interconexión entre ellos. Estos dispositivos pueden emular el comportamiento de toda clase de circuitos de lógica digital incluyendo procesadores[20].

La implementación de estos circuitos se realiza en lenguajes de descripción de hardware como: Verilog, VHDL (*Very high speed integrated circuit-Hardware Description Language*) y SystemC, ó mediante herramientas gráficas. Inicialmente los FPGA se utilizaron solo para desarrollar prototipos a partir de los cuales se construyen ASICs (*Application-Specific Integrated Circuit*).

[20] CHU, Pong P. FPGA Prototyping by Verilog examples. Wiley, 2008. p. 15

5.4.1. Procesadores embebidos en FPGA

El crecimiento en la densidad de componentes en un circuito integrado ha permitido aumentar la cantidad de bloques de lógica configurable en FPGA a un nivel (2 millones para la FPGA Virtex-7 2000T) que permite la implementación de múltiples procesadores embebidos como parte del circuito y soportar frecuencias de funcionamiento comparables a las de un ASIC. Esto le permite a los desarrolladores de firmware seguir utilizando sus programas con la ventaja de poder acelerar instrucciones por hardware o incluir periféricos a un microprocesador, formando lo que se conoce como *System on Chip* (SoC).

Actualmente existe la descripción en HDL de varios microprocesadores para implementar en FPGA. El fabricante Xilinx ofrece su diseño MicroBlaze que presenta una arquitectura RISC de 32-bits Harvard y una versión gratuita de 8-bits llamada PicoBlaze para implementar en sus FPGA.

Altera ha enfocado gran parte de su desarrollo en el procesador Nios II, con una arquitectura RISC de 32-bits y un set de herramientas avanzadas para su configuración y programación, como lo son el IDE (*Integrated Development Tool*) para compilar y depurar proyectos en lenguajes de alto nivel y el SOPC(*System On a Programmable Chip*) ó Qsys como herramienta gráfica de integración de sistemas, es decir, que permite integrar sistemas con o sin procesadores a partir de módulos de su IP (Intellectual Property). Altera también ofrece en su sitio web un portafolio de IP con procesadores de 8, 16 y 32-bits entre los cuales se encuentran el ARM Cortex-M1, V1 Coldfire, MP32 y el C68000

Igualmente existe la descripción de OpenRISC, un microprocesador enfocado a aplicaciones de redes de datos, sistemas embebidos y móviles, de código abierto, manejado por la comunidad de OpenCores. Esto permite una rápida depuración de errores y un amplio conocimiento interno del dispositivo. Las implementaciones del OpenRISC en FPGA han sido gracias a la plataforma ORSoC (*OpenRISC System on Chip*) pero la comunidad tiene como objetivo la fabricación y distribución comercial del dispositivo.

6. DESARROLLO METODOLÓGICO

Durante la ejecución de este proyecto se diseñaron e implementaron dos microprocesadores en dispositivos de lógica digital programable, uno de arquitectura RISC y otro CISC, como parte de una plataforma con fines educativos que se nombró X-ISCKER (RISC/CISC *Instruction Set Computing Key Educational Resource*). Ésta plataforma cumple tres funciones principales:

Emulación: Permite observar el comportamiento del procesador implementado en FPGA, mediante comunicación serial con un computador personal por puerto USB, al mismo tiempo que ejecuta algoritmos programados por el usuario. Ambas tareas, programación y observación se hacen a través del software X-ISCKER IDE que cuenta con una interfaz de ensamblador y las herramientas X-ISCKER *Programmer* y *Observer*.

Aplicación: Enfoca la plataforma a proyectos de aplicación donde el programador ya tiene conocimiento sobre el funcionamiento de los procesadores. Esto se logra utilizando el software X-ISCKER IDE como ensamblador de los dos lenguajes correspondientes y la herramienta *Programmer* con una descripción de hardware que carece de las funciones de emulación, permitiendo utilizar frecuencias de funcionamiento más altas.

Reconfiguración: La descripción del hardware de cada procesador y código fuente de la plataforma están abiertas y documentadas al detalle, con el propósito de que puedan ser modificadas. De manera que es posible agregarle nuevas funciones a cada procesador (Instrucciones, puertos, periféricos, etc.), y al software ensamblador (directivas, librerías, etc.).

El procedimiento para lograr la construcción de esta plataforma se ejecutó en cuatro etapas: Diseño de un procesador RISC (RISCKER), diseño de un procesador CISC (CISCKER), diseño del entorno de desarrollo integrado (X-ISCKER IDE) y documentación.

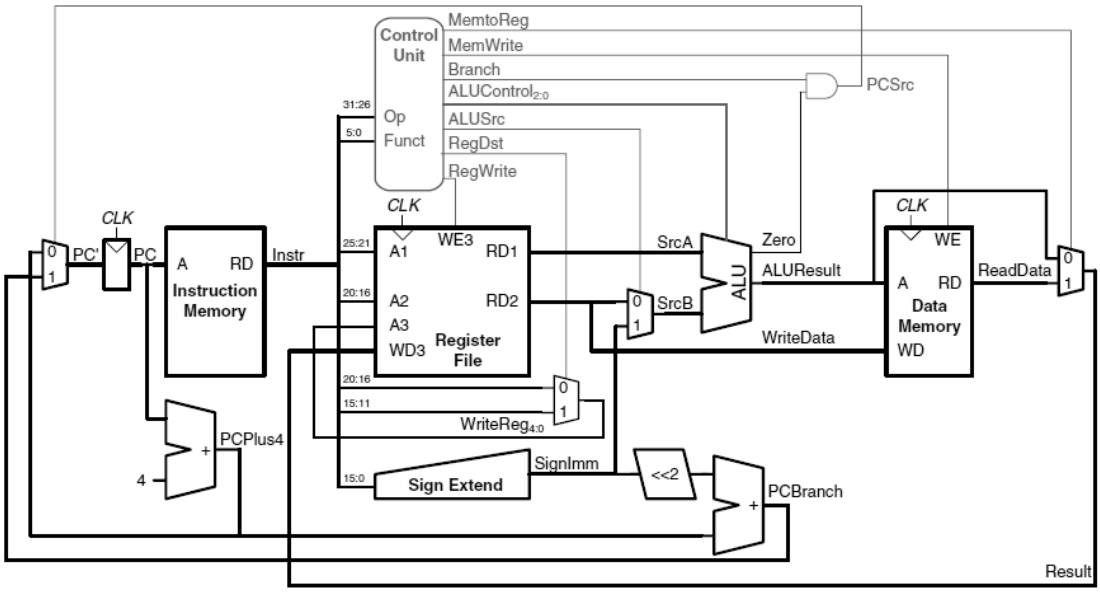
6.1. DISEÑO DE UN PROCESADOR RISC: “RISCKER”

El diseño del microprocesador RISCKER *(Reduced Instruction Set Computing Key Educational Resource*) está basado en la arquitectura MIPS. Se parte de la ruta de datos elaborada en el séptimo capítulo del libro *Digital Design and Computer Architecture* de David Money Harris y Sarah Harris, como también de un diseño similar realizado en el capitulo cinco del libro *Computer Organization and Design* de David A. Patterson y John L. Hennessy.

Los diagramas expuestos en estos textos representan una arquitectura capaz de ejecutar instrucciones de asignación entre la ALU, el archivo de registros y las memorias de programa y datos. Este diseño interconecta cada subsistema mediante lógica combinacional controlada por una HCU que genera las señales apropiadas de acuerdo a la instrucción leída de memoria. Estas señales determinan los operandos y operación de la ALU, registro de destino, dato de escritura en el archivo de registros y siguiente dirección de lectura de la memoria de instrucciones. El diagrama de esta arquitectura se muestra en la Figura *5*.

En la arquitectura que se muestra en la Figura *5* el cálculo de la siguiente dirección de lectura corresponde a la dirección actual del contador de programa (PC) más cuatro, esto se debe a que el diseño cuenta con memorias direccionables por byte y palabras de treinta y dos bits (cuatro bytes). Cuando se trata de saltos relativos la siguiente dirección es la suma del valor inmediato leído de memoria multiplicado por cuatro (corrimiento a la izquierda de dos bits) más la dirección actual del contador de programa. La condición de salto es únicamente el valor cero en el resultado de la ALU.

Figura 5. Ruta de datos MIPS[21]



Este microprocesador cuenta con una memoria de instrucciones de 32 bits, 32 registros en el archivo de registros, una unidad lógico-aritmética de 32 bits con 3 bits de control y una memoria de datos de 32 bits. Las instrucciones pueden contener números inmediatos de hasta 16 bits.

Este diseño se modifica para lograr un mayor número de instrucciones de asignación y llamados a procedimientos, logrando flexibilidad en el circuito sin sacrificar la simplicidad existente.

El diseño de la arquitectura RISCKER comienza con la implementación de un set de instrucciones completo, destinando todas las posibles combinaciones de códigos de operación a nuevas operaciones de asignación entre registros e inmediato, se implementa un salto condicional por la bandera de signo y se agregan tres operaciones de flujo de programa.
Los datos y buses del microprocesador RISCKER son de 16bits, reduciendo

[21] HARRIS, David Money; HARRIS, Sarah L. Digital design and computer arquitecture. San Francisco. Morgan Kaufmann Publishers, Elsevier, 2007. p. 375

considerablemente el tamaño de las memorias y del archivo de registros. Este último representa gran parte de los recursos usados en el FPGA en el momento de implementación ya que cada bloque ó elemento de lógica configurable solo puede aportar hasta 2 registros en los dispositivos seleccionados. La memoria de instrucciones se mantiene de 32 bits destacando la utilidad de una unidad de memoria Harvard y permitiendo la programación de valores inmediatos de 16bits dentro del código que pueden ser cargados directamente a registros.

Las modificaciones hechas a la ruta de datos aprovechan el hardware existente para implementar nuevas instrucciones teniendo en cuenta los cuatro principios de la filosofía RISC y la simplicidad representada por la arquitectura MIPS.

Las operaciones de asignación entre registros comparten el mismo *Opcode* y su decodificación depende directamente de diez bits contenidos dentro de la instrucción, denominados "*Function*". Mientras que las operaciones con valores inmediatos contienen en su *Opcode* la selección directa de la operación de la ALU. Además, el caso común de llamado y retorno de procedimientos se estructura con instrucciones de salto a registro y salto con valor de retorno, independientes y de rápida ejecución.

El archivo de registros se aumenta a 64 posiciones gracias al cambio del formato de la instrucción y los distintos tipos detallados en la ISA RISCKER. Esto permite un mayor número de registros de uso general como también la asignación de más registros de uso especial como, valores de retorno, argumentos de subrutina y direcciones de retorno, entre otros.

6.1.1. Arquitectura del set de instrucciones RISCKER[22]

Al igual que las instrucciones correspondientes a la arquitectura MIPS, la ISA RISCKER puede dividirse en tres tipos.

---

[22] HARRIS, David Money; HARRIS, Sarah L. Digital design and computer arquitecture. San Francisco. Morgan Kaufmann Publishers, Elsevier, 2007. p. 299

6.1.1.1. Instrucciones tipo R

Las instrucciones de tipo Registro o “R” utilizan hasta tres registros como operandos, dos como fuente de los datos (Rs y Rt) y uno como destino (Rd), seguidos por un campo de función y uno de cantidad de corrimiento o “*shift amount”* (*Shamt*). Estas funciones se identifican por el *Opcode* 0000 y la asignación correspondiente está determinada por el campo función.

0000	Rd	Rs	Rt	*Function* [ALU *operation* & *Shift amount*]

6.1.1.2. Instrucciones tipo I

Las instrucciones de tipo Inmediato o “I” utilizan dos registros como operandos, destino y fuente; que se operan con un inmediato contenido en los dos bytes menos significativos de la instrucción. La operación está determinada únicamente por el *Opcode* y puede corresponder a funciones de asignación, salto condicional o de acceso a memoria.

En el caso de las operaciones de salto condicional, la instrucción compara los dos registros indicados en cada campo. Si la condición declarada en la instrucción se cumple, carga el registro contador de programa con la dirección contenida en el campo inmediato.

En las instrucciones de acceso a memoria la dirección de lectura o escritura es el resultado de la suma entre el valor inmediato y el registro fuente. El registro de destino recibe el dato de la memoria en la operación de lectura, o contiene el dato que se almacena en memoria al finalizar la operación de escritura. Este formato permite tres modos de direccionamiento:

- Directo: La dirección está especificada en el inmediato y el registro fuente es cero.
- Indirecto: La dirección está contenida en el registro fuente y el inmediato es cero.
- Desplazado: La dirección es el resultado de la suma de los datos de cada campo.

Opcode	Rd	Rs	Inmediato

6.1.1.3. Instrucciones tipo J

Las instrucciones de salto (*Jump*) afectan el registro contador de programa haciendo que apunte a una dirección especificada en la instrucción. La arquitectura RISCKER cuenta con tres clases de saltos:

- Salto Directo: Carga el contador de programa con la dirección absoluta especificada en la instrucción.

Opcode	Inmediato

- Salto Indirecto con Carga a Registro: Carga PC con el valor almacenado en el registro fuente y actualiza el valor del registro destino con el especificado en el inmediato.

Opcode	Rd	Rs	Inmediato

- Salto Directo y Carga de Registro con Dirección de Retorno: Carga el registro destino con el valor de PC y actualiza PC con la dirección especificada en el campo Inmediato.

Opcode	Rd	Inmediato

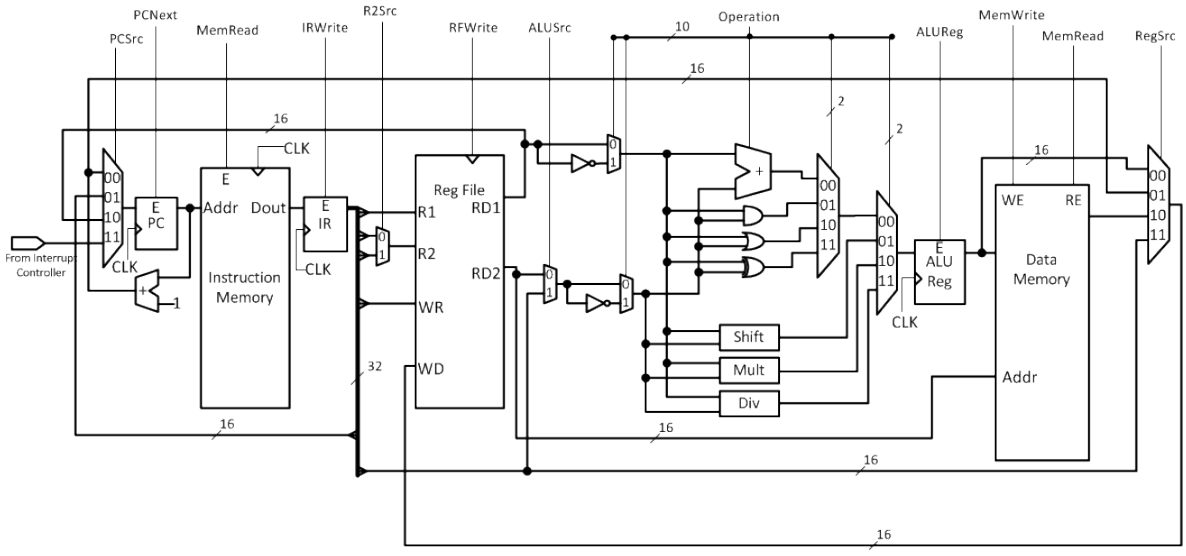
6.1.2. Organización del hardware RISCKER

6.1.2.1. Ruta de datos

La Figura 6 muestra la ruta de datos funcional RISCKER y las señales que la Unidad de Control tiene disponibles para seleccionar y modificar operandos y datos. Teniendo en cuenta que la unidad de control es una máquina de estados que modifica señales de la forma necesaria para la ejecución de cada instrucción.

Adicional a los módulos, señales y componentes especificados en la Figura 6 se debe recalcar la existencia de registros de uso especial dentro del archivo de registros especificados en el numeral 6.1.2.5. Registros y la existencia del bus proveniente del módulo de control de interrupciones encargado de entregar la dirección donde se encuentra la rutina de servicio de interrupción.

Figura 6. Ruta de datos RISCKER



6.1.2.2. Unidad de control

La unidad de control RISCKER es una máquina de diez estados que genera las señales de control necesarias para ejecutar cada instrucción. Es una unidad controlada por hardware (HCU) ya que las señales de control provienen directamente de la instrucción y de un circuito de lógica.

Estados de la unidad de control:

IDLE: Estado de no operación.
FETCH: Se registra la instrucción.
EXECUTE_OP: Se selecciona la operación a realizar en la ALU.
EXECUTE_JRLW: Carga PC con el valor de un registro y retorna un dato en otro.
WRITE_OP: Operación de escritura de la memoria de datos.
WRITE_SW: Operación de lectura de la memoria de datos.
JUMP: Carga PC con un valor especificado en la instrucción.
INTERRUPT: Revisa si hay interrupciones pendientes.
INTJUMP: Estado de interrupción, salta a la subrutina de interrupción.
RETFI: Regresa de la subrutina de interrupción.

Las señales de control que se generan en esta unidad son las siguientes:

PCsrc: Se encarga de seleccionar la fuente de carga del registro PC.
PCNext: Flanco de reloj del registro PC.
IRwrite: Flanco de reloj del registro de instrucción.
R2src: Fuente de la dirección del registro 2 del archivo de registros.
RFwrite: Flanco de reloj de escritura del archivo de registros.
ALUsrc: Fuente del operando B de la unidad de operaciones.
Operation: Bus de control que selecciona la operación indicada.
ALUreg: Flanco de reloj que registra el resultado de la ALU.
MemWrite: Habilitador de escritura de la memoria de datos.
MemRead: Habilitador de lectura de la memoria de datos.
RegSrc: Fuente de datos a escribir en el archivo de registros.
IPCWrite: Flanco de reloj del registro PC de interrupción.
INToggle: Señal que indica el inicio o final de servicio de interrupción.

La Tabla 3 muestra la relación entre los estados de la HCU y las señales de

control.

Tabla 3. Señales de control HCU-RISCKER

Estado\Señal	ipc_write	intoggle	pc_next	ir_write	rf_write	alu_reg	mem_write
idle	0	0	0	0	0	0	0
fetch	0	0	0	1	0	0	0
execute_op	0	0	1	0	0	1	0
execute_jrlw	0	0	1	0	1	0	0
write_op	0	0	0	0	1	0	0
write_sw	0	0	0	0	0	0	1
jump	0	0	1	0	0	0	0
interrupt	1	1	0	0	0	0	0
intjump	1	0	1	0	0	0	0
retfi	0	1	0	0	0	0	0

Hay otras señales que dependen directamente del código de operación de la instrucción y no cambian durante los estados de su ejecución, estas señales y su correspondencia para cada instrucción se encuentran en la Tabla 4.

En la Figura 7 se muestra el diagrama de estados de la unidad de control, especificando las señales modificadas en cada transición.

En el momento de energizar el sistema, la máquina de estados se va a encontrar en un estado inactivo o *IDLE*, a donde solo regresará en el caso de reiniciar el procesador. De este estado irá al estado *FETCH* donde la señal *IRWrite* registra la instrucción leída de memoria en el flanco de reloj anterior. La ejecución continúa con la etapa *DECODE* o decodificación de la instrucción donde, dependiendo de la instrucción se ejecutaran los estados siguientes, *OPERATION* para instrucciones de asignación, *JUMP* para instrucciones de salto; después de ejecutar el último

estado de la instrucción, la máquina regresa hacia el estado *FETCH* a través de una condición "*Interruption*" que se cumple si hay una interrupción esperando ser atendida.

Figura 7. Diagrama de estados RISCKER

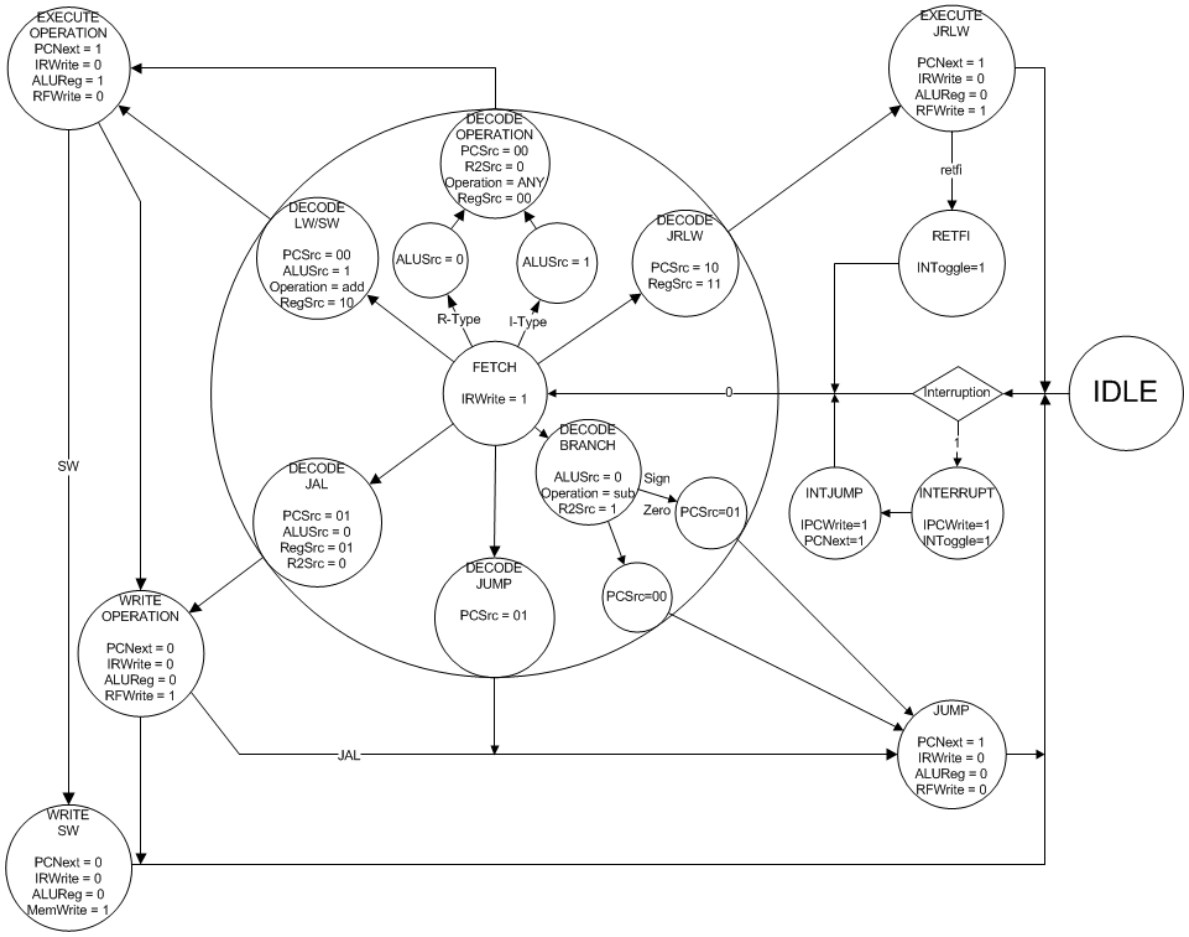


Tabla 4. Señales de control independientes de la máquina de estados

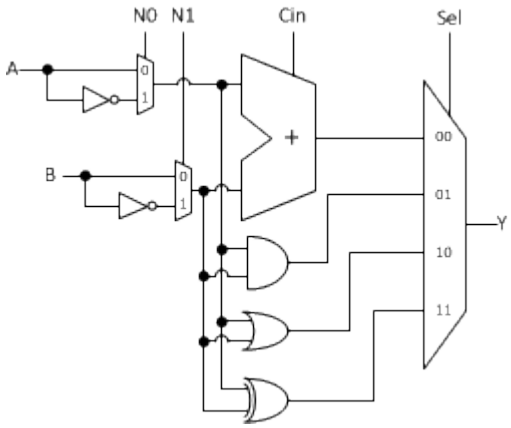
Instruction	**PCSrc**	**R2Src**	**ALUSrc**	**Operation (11 bits)**				**RegSrc**
				aluop	**shamt**	**shift_mode**	**select**	
add	00	0	0	000	XXXX	XX	00	00
and	00	0	0	001	XXXX	XX	00	00
or	00	0	0	010	XXXX	XX	00	00
xor	00	0	0	011	XXXX	XX	00	00
nand	00	0	0	100	XXXX	XX	00	00
nor	00	0	0	101	XXXX	XX	00	00
sub	00	0	0	110	XXXX	XX	00	00
neg	00	0	0	110	XXXX	XX	00	00
inc	00	0	0	111	XXXX	XX	00	00
mult	00	0	0	XXX	XXXX	XX	10	00
div	00	0	0	XXX	XXXX	XX	11	00
rot	00	0	0	XXX	SHAM	00	01	00
sll	00	0	0	XXX	SHAM	01	01	00
srl	00	0	0	XXX	SHAM	10	01	00
sra	00	0	0	XXX	SHAM	11	01	00
addi	00	X	1	000	XXXX	XX	00	00
andi	00	X	1	001	XXXX	XX	00	00
ori	00	X	1	010	XXXX	XX	00	00
xori	00	X	1	011	XXXX	XX	00	00
nandi	00	X	1	100	XXXX	XX	00	00
nori	00	X	1	101	XXXX	XX	00	00
multi	00	X	1	XXX	XXXX	XX	10	00
divi	00	X	1	XXX	XXXX	XX	11	00
lw	00	X	1	000	XXXX	XX	00	10
sw	00	X	1	000	XXXX	XX	00	XX
beq	01	1	0	110	XXXX	XX	00	XX
blt	01	1	0	110	XXXX	XX	00	XX
j	01	1	0	XXX	XXXX	XX	XX	01
jrlw	10	0	X	XXX	XXXX	XX	XX	11
jal	01	X	X	XXX	XXXX	XX	XX	01

6.1.2.3. Unidad lógico-aritmética

Este módulo se encuentra dentro de la unidad llamada "*operations*" que además de la ALU, contiene el hardware para las instrucciones de corrimiento de bits, de multiplicación y de división. Los bits que seleccionan la operación que se almacenará en el registro de salida de la ALU son aquellos especificados en el campo "*function*" de las instrucciones tipo R, o es decodificado del *Opcode* de las instrucciones tipo I.

Como ilustra la Figura 8, La unidad lógico-aritmética RISCKER incluye las compuertas necesarias para las operaciones suma, AND, OR y XOR, cuyos operandos pueden ser negados para conseguir las operaciones NAND y NOR, basándose en las leyes de Morgan, como se muestra en la Ecuación 1 y Ecuación 2.

Figura 8. ALU RISCKER



Ecuación 1. Operación NAND

$$\bar{A} + \bar{B} = (\overline{A \cdot B})$$

Ecuación 2. Operación NOR

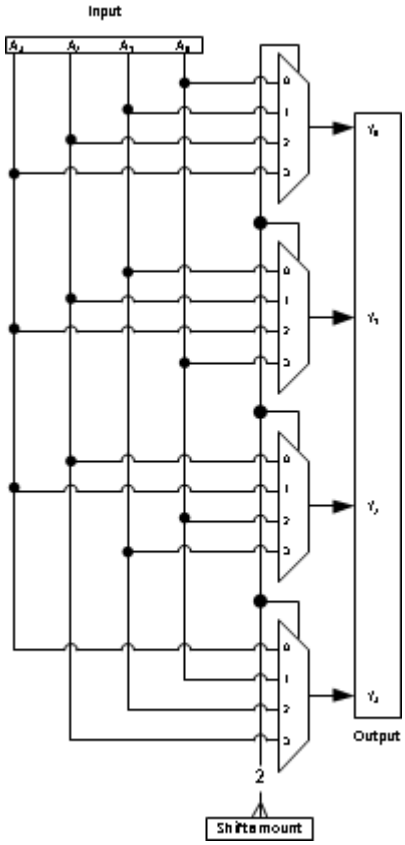
$$\bar{A} \cdot \bar{B} = (\overline{A + B})$$

Los negadores también son de utilidad cuando se ejecuta una instrucción de resta,

la cual está implementada como una suma con un operando negado más la señal de acarreo, obteniendo así el complemento a dos de ese operando.

El funcionamiento de las instrucciones de corrimiento de bits es de tipo barril, es decir, la instrucción especifica el número de bits que se requiere en el corrimiento mediante los bits "*shift amount*" o "*shamt*", y el código de operación indica la dirección o rotación. Esto se logra con la implementación de series de multiplexores conectados de tal forma que el resultado dependa únicamente de los selectores, la Figura *9* muestra un arreglo de multiplexores que ejecutan la instrucción rotar.

Figura 9. Arreglo de multiplexores para la función rotar



La Tabla 5 muestra las señales de control de la ALU que permiten la ejecución de las diferentes operaciones.

Tabla 5. Señales de control de operaciones.

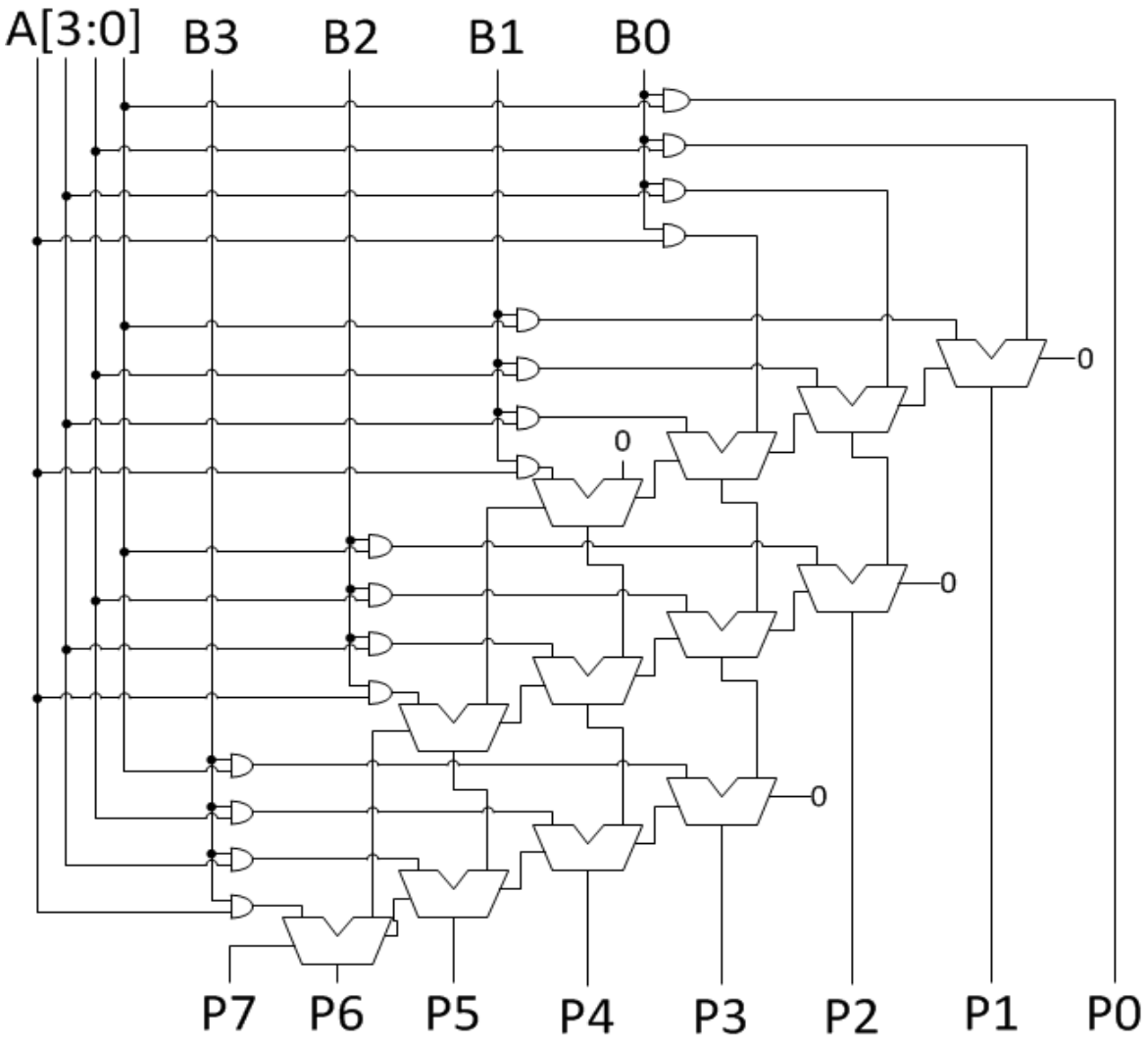
Instrucción	Operation (11bits)			
	aluop	shamt	shift_mode	select
ADD	000	XXXX	XX	00
AND	001	XXXX	XX	00
OR	010	XXXX	XX	00
XOR	011	XXXX	XX	00
NAND	100	XXXX	XX	00
NOR	101	XXXX	XX	00
SUB	110	XXXX	XX	00
NEG	110	XXXX	XX	00
INC	111	XXXX	XX	00
MULT	XXX	XXXX	XX	10
DIV	XXX	XXXX	XX	11
ROT	XXX	Cantidad	00	01
SLL	XXX	Cantidad	01	01
SRL	XXX	Cantidad	10	01
SRA	XXX	Cantidad	11	01

1.1.1.1.3. Multiplicador paralelo

Los bloques de procesamiento digital de señales (DSP) que proveen las FPGA Altera (Cyclone IV) y Xilinx (Spartan 3A), contienen multiplicadores en paralelo para acelerar aquellas aplicaciones que lo requieran. Para la arquitectura RISCKER se utilizan estos módulos de multiplicación paralela pero también se ofrece la descripción de un módulo de multiplicación de tamaño configurable para demostrar su funcionamiento.

El módulo consta de series de sumadores en paralelo de acuerdo al tamaño máximo de los operandos, configurado para el módulo. La Figura 10 muestra un multiplicador con operandos de cuatro bits. El algoritmo se resume en la utilización de compuertas AND para la multiplicación individual de bits de manera correspondiente entre los operandos A y B. Estos resultados se suman de forma acumulativa con los demás bits y la señal de acarreo proveniente del índice anterior, este valor es 0 para los bits menos significativos.

Figura 10. Multiplicador paralelo



6.1.2.3.1. Divisor paralelo

El procesador RISCKER cuenta con un módulo de división dedicado, que se construyó con base en los módulos expuestos en los libros *Computer Organización*[23] *y Digital Design*[24].
Consiste de un arreglo de sumadores conectados en cascada que efectúan la operación de división mediante el algoritmo descrito en el diagrama de flujo de la

[23] PATTERSON, David A.; HENNESSY, John L. Computer Organization and Design. Op. cit., p. 183
[24] HARRIS, David Money; HARRIS, Sarah L. Digital design and computer arquitecture. San Francisco. Morgan Kaufmann Publishers, Elsevier, 2007. p. 248

Figura *11*.

Acorde con la

Figura *11*, para llevar a cabo la división se inicializa el residuo (R) con el valor del dividendo (A). Este residuo parcial se le resta al divisor (B) repetidamente para determinar cuántas veces cabe. Si la diferencia entre residuo y divisor (D) es negativa, se asigna cero al cociente (Q) y se descarta el resultado de la resta. Si por el contrario la diferencia es positiva, se asigna uno al cociente y el residuo recibe la diferencia. En cualquier caso el residuo se multiplica por dos, es decir se hace un corrimiento a la derecha, y se repite el proceso N+1 veces, Dónde N es el número de bits de los operandos. El diagrama de bloques de la Figura 12 ilustra la implementación en hardware de este algoritmo para N igual a cuatro bits.

Figura 11. Algoritmo de división binaria

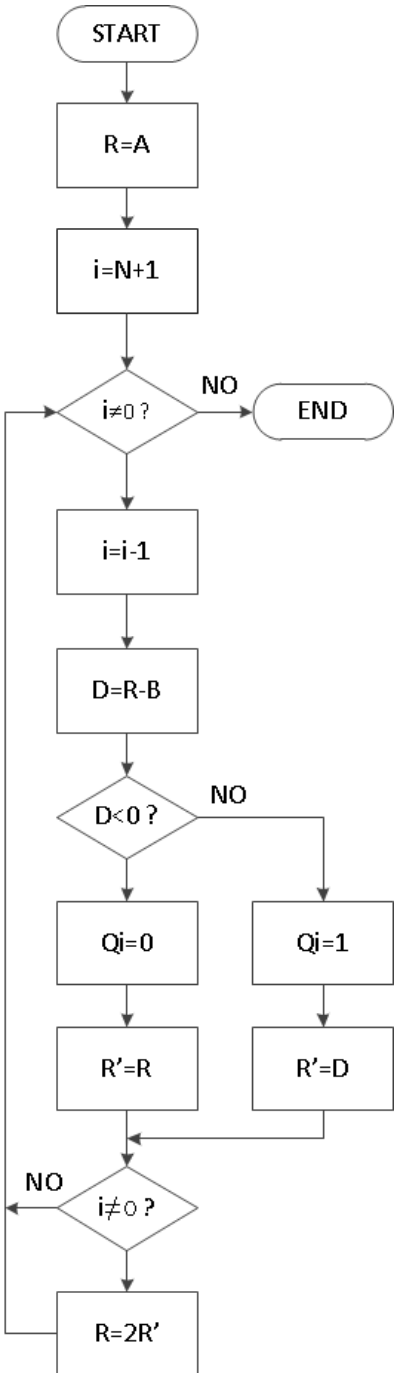
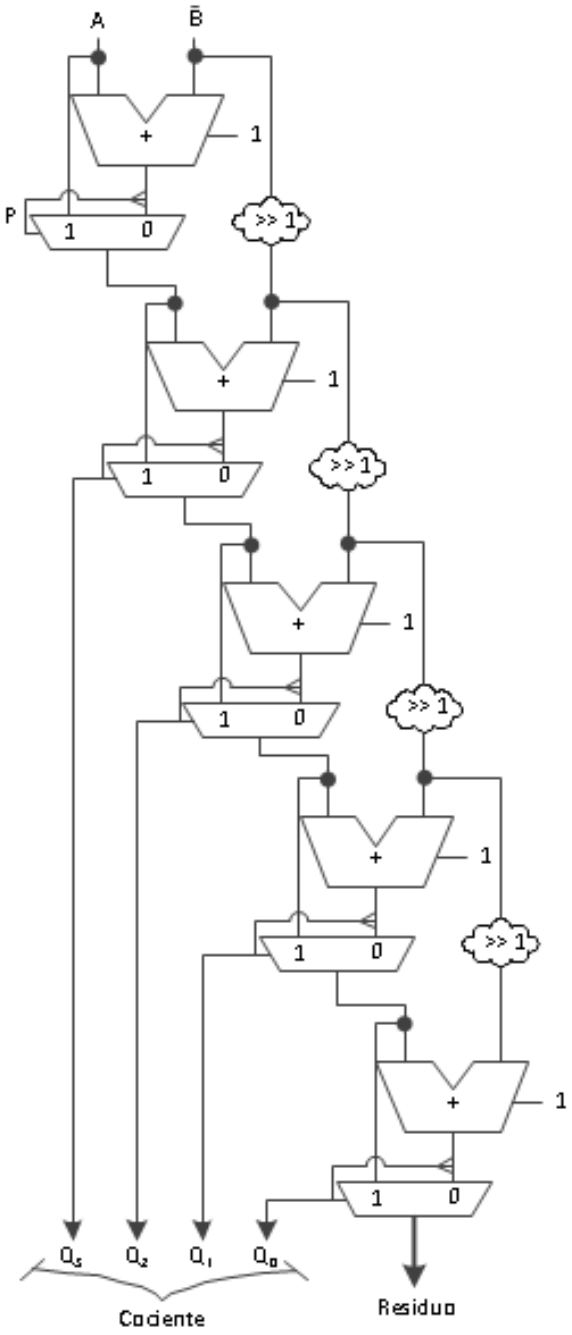


Figura 12. Diagrama del modulo de división paralela



6.1.2.4. Unidad de memoria

Para el diseño de este microprocesador se optó por una unidad de memoria tipo Harvard, con una memoria ROM de instrucciones de 32 bits y una memoria RAM para datos de 16 bits direccionadas de forma independiente por el contador de programa y el resultado de la ALU respectivamente.

El acceso a la memoria de datos en la arquitectura RISCKER se hace únicamente a través de las instrucciones de escritura y lectura y sus tres modos de direccionamiento (Directo, indirecto y desplazado). Un cuarto modo de direccionamiento corresponde al de las instrucciones tipo “I” (Inmediato), en el cual los datos están contenidos en la memoria ROM y son leídos únicamente durante la ejecución de la instrucción que lo contenga.

6.1.2.5. Registros

El procesador RISCKER cuenta con un archivo de 64 registros de 16 bits organizados de la siguiente forma:

6.1.2.5.1. Registros Modificables

Estos registros permiten escritura y lectura en todas las instrucciones. Están divididos para cumplir distintas funciones de programa pero su implementación en hardware es la misma. El buen uso de estos registros permite la reutilización sencilla de código.

- Valores de Retorno (rv0-rv7): Registros destinados para retornar valores de subrutinas.

- Argumentos de subrutina (rp0-rp7): Registros destinados para llevar valores como argumentos de subrutinas.

- Registros de uso Temporal (rt0-rt7): Registros para uso temporal como intermediario en algoritmos complejos o de subrutinas.

- Registros Globales (rg0-rg21): Registros para definir constantes y variables de programa.

- Dirección de Retorno (ra0-ra3): Registros para direcciones de retorno de procedimientos. Se permiten hasta cuatro procedimientos en cadena teniendo en cuenta la dirección de retorno correspondiente.

- Puntero de la Pila (sp): Registro que debe apuntar a la última localidad ocupada de la pila. Su utilización es completamente por software, "decrementando" su valor antes de una escritura e incrementándolo después de una lectura.

6.1.2.5.2. Registros de Sólo Lectura

Son registros especiales que contienen información relacionada al funcionamiento del microprocesador.

- Registro 0 (r0): Constante cero mapeada en el archivo de registros.

- Temporizador (tmr1, tmr2): Contiene el valor del contador de cada temporizador.

- Puerto de Entrada (ip): Contiene la información del puerto de entrada.

- *Interrupt* PC (ipc): Contiene el valor del contador de programa antes de ejecutar una interrupción.

6.1.2.5.3. Registros de propósito especial

Son registros que configuran características del procesador.

- Periodo del Temporizador (pr1, pr2): Contiene el valor en el cual ocurre una interrupción por desbordamiento de un temporizador, por defecto es 65535.

- Direcciones de Interrupción por desbordamiento de Temporizador (ti1, ti2): Contiene la dirección en la que se encuentra el servicio de interrupción por desbordamiento de cada Temporizador.

- Direcciones de Interrupción Externa (eir1, eir2): Contiene la dirección en la que se encuentra el servicio de cada interrupción externa.

- Puerto de Salida (op): Registro conectado como salida del microprocesador.

- Registro de control (rc): Registro de banderas de configuración de interrupciones y temporizadores.

La

Tabla *6* muestra la distribución del archivo de registros del procesador RISCKER.

6.1.2.6. Interrupciones y temporizadores

La arquitectura RISCKER cuenta con cuatro fuentes de interrupción, dos interrupciones por hardware, que provienen de la IOU y dos interrupciones por desbordamiento de temporizador. Estas señales van al módulo de control de interrupciones que se muestra en la

Figura *13*, el cual activa una única bandera que indica si reconoce o no la interrupción de acuerdo a los registros de configuración de interrupciones.

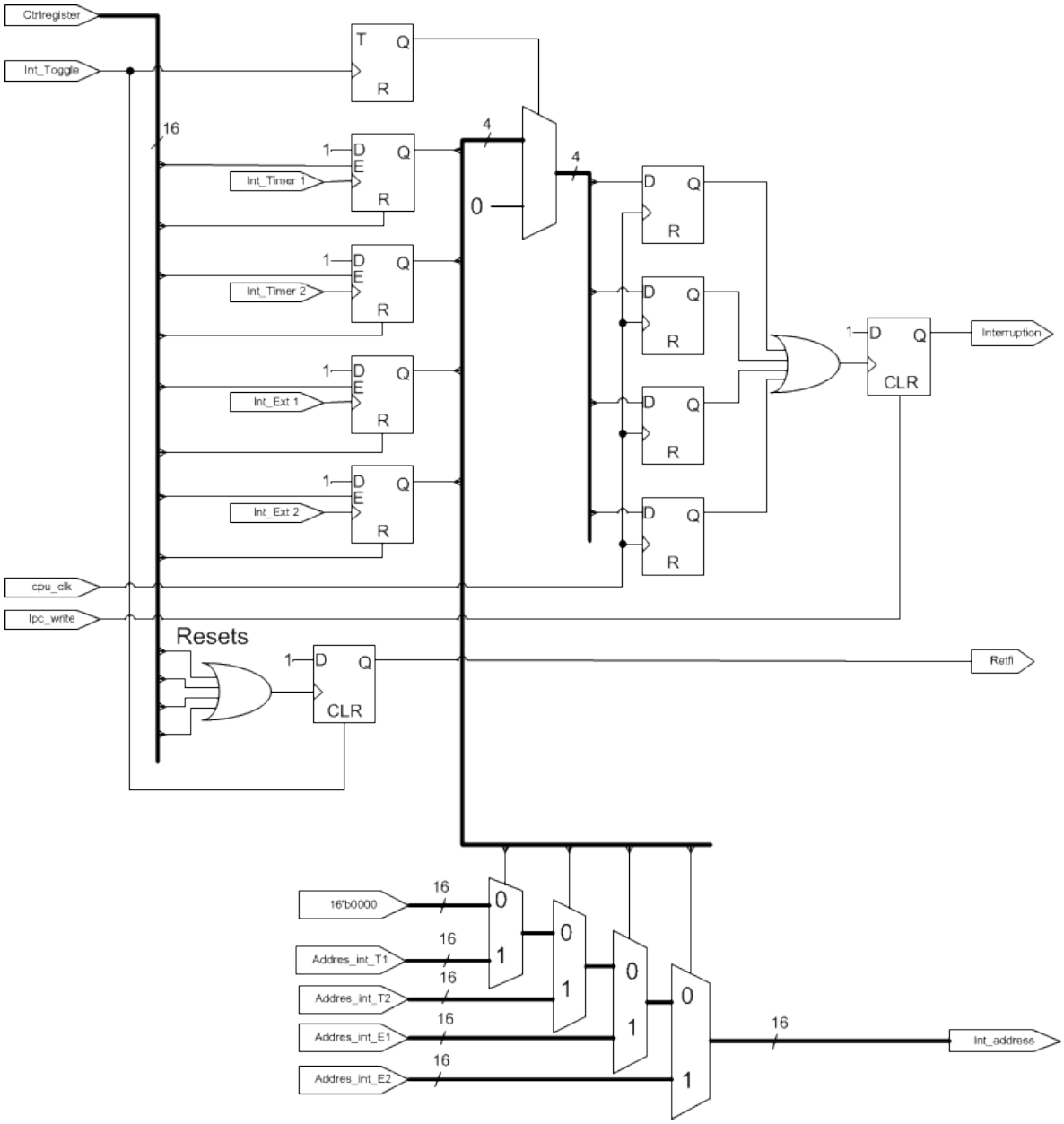
Este módulo se encarga de enviar directamente al contador de programa la dirección donde se encuentra la RSI de mayor prioridad que haya ocurrido y se comunica con la máquina de estados para controlar su ejecución. Cada una de las banderas de interrupción pueden deshabilitarse o habilitarse y deben ser borradas al momento de salir de la RSI, las señales para habilitar o borrar cada uno de los registros hacen parte del registro de control en el archivos de registros.

Cuando la señal “*interruption*” del controlador de interrupciones se activa, le indica a la máquina de estados de la HCU que debe salir de la ejecución normal del programa para entrar a la RSI. Esto lo hace en el ciclo de máquina que precede a la ejecución de cualquier instrucción, como se observa en la Figura *7*, los siguientes estados que se ejecutan, almacenan el valor actual del contador de programa y reemplazan su contenido por la dirección de la RSI.

Tabla 6. Archivo de registros RISCKER

Número de Registros	Nombre	Índice	Descripción
1	r0	0	Valor 0
8	rv0-rv7	1:8	Valores de retorno
8	rp0-rp7	9:16	Argumentos de subrutina
8	rt0-rt7	17:24	Registros de uso temporal
22	rg0-rg21	25-46	Registros globales
4	ra0-ra3	47-50	Dirección de retorno
1	sp	51	Puntero de la pila
2	pr1-pr2	52-53	Periodo de temporizador
2	ti1-ti2	54-55	Direcciones de interrupción por temporizador
2	eir1-eir2	56-57	Dirección de interrupciones Externas
1	op	58	Registro de salida
1	rc	59	Registro de control
2	tmr1-tmr2	60-61	Registros de temporizador
1	ip	62	Registro de entrada
1	ipc	63	PC de la interrupción

Figura 13. Módulo de control de interrupciones RISCKER



Finalmente, cuando el contador de programa se encuentra ubicado en la dirección de la RSI, es necesario que las instrucciones que se ejecuten a continuación cumplan con unos requisitos de ejecución. El programador debe crear un algoritmo que cumpla con el almacenamiento de aquellos registros que se vayan a utilizar durante la RSI, en la pila de memoria. Además, el algoritmo debe borrar la señal de interrupción antes de retornar a la ejecución regular del programa. La máquina de estados reconoce el cambio de la señal "*return from interruption"* o "retfi" al momento de saltar al registro donde se encontraba almacenado el valor del contador de programa en el momento de la interrupción.

6.2. DISEÑO DE UN PROCESADOR CISC: "CISCKER"

El diseño de la ISA y organización del microprocesador CISCKER *(Complex Instruction Set Computing Key Educational Resource*) se realiza con base en la documentación de las series de microcontroladores HC08 y HC11 ofrecida por la compañía Freescale. La arquitectura diseñada presenta un set de instrucciones y funcionamiento similar en cuanto al número de ciclos necesarios para la ejecución de las instrucciones por la MCU.

Las series de microcontroladores CISC de Freescale cuentan con una unidad de control que utiliza siete tipos de micro-instrucciones, ejecutadas en cuatro ciclos de reloj[25].

- *Program fetch*(p): Realiza una lectura de la siguiente localidad de memoria de programa.
- *Read* (r): Realiza la lectura de un operando de 8bits.
- *Push* (s): Escribe ocho bits en la pila.
- *Pop* (u): Lee ocho bits de la pila.

[25] FREESCALE SEMICONDUCTOR. CPU08 Central Processor Unit. 2006. p. 19

- *Write* (w): Realiza la escritura de un operando de 8bits.

- *Vector Read* (v): Lee un vector como dirección de una subrutina de servicio de interrupción.

- *Dummy* (d): Siempre es un ciclo de lectura direccionado por la micro-instrucción anterior.

Las instrucciones están codificadas en un byte, dando la posibilidad de 256 instrucciones distintas, sin embargo, el diseño de un set de instrucciones para esta cantidad de posibilidades también debe contemplar la compatibilidad entre versiones de la arquitectura y asegurar una unidad de decodificación y de control reducida. Esto se logra, organizando las instrucciones de manera que los bits que codifiquen cada una de ellas sean comunes para instrucciones de ejecución similar. La mejor representación de esta organización son los mapas de *Opcode*.

La Figura 14, muestra el mapa de *Opcode*s de la arquitectura HC08, que a diferencia de la arquitectura HC11 está representado en una sola “página” y muestra en la parte superior la organización por tipo de instrucción y por modo de direccionamiento. Cuando un mapa de *Opcode*s cuenta con más de una “página”, debe existir un byte que le indique a la unidad de control que debe leer la instrucción del siguiente byte en memoria. La decodificación de *Opcode*s en más de una página agrega un retardo en las ejecuciones porque exige una lectura adicional de memoria para decodificar la instrucción.

En cuanto a la organización de hardware de las dos arquitecturas analizadas, la diferencia está en que la arquitectura HC11 permite la operación de datos de diez y seis bits aún contando con una unidad de memoria de ocho bits. La arquitectura HC08 ofrece un único acumulador de ocho bits a diferencia de la HC11 en la cual existen dos acumuladores que concatenados por hardware hacen un registro de diez y seis bits; para el cual existen instrucciones dedicadas de operaciones de asignación, carga y comparación. HC11 también agrega un segundo registro de índice y es por estos dos cambios que su set de instrucciones aumenta en número y se hacen necesarias más páginas de *Opcode* para distribuirlas eficientemente.

Figura 14. Mapa de *Opcode*s HC08

	Bit-Manipulation		Branch	Read-Modify-Write						Control		Register/Memory							
	DIR	DIR	REL	DIR	INH	INH	IX1	SP1	IX	INH	INH	IMM	DIR	EXT	IX2	SP2	IX1	SP1	IX
HIGH / LOW	0	1	2	3	4	5	6	9E6	7	8	9	A	B	C	D	9ED	E	9EE	F
0	BRSET0 3 DIR	BSET0 2 DIR	BRA 2 REL	NEG 2 DIR	NEGA 1 INH	NEGX 1 INH	NEG 2 IX1	NEG 3 SP1	NEG 1 IX	RTI 1 INH	BGE 2 REL	SUB 2 IMM	SUB 2 DIR	SUB 3 EXT	SUB 3 IX2	SUB 4 SP2	SUB 2 IX1	SUB 3 SP1	SUB 1 IX
1	BRCLR0 3 DIR	BCLR0 2 DIR	BRN 2 REL	CBEQ 3 DIR	CBEQA 3 IMM	CBEQX 3 IMM	CBEQ 3 IX1+	CBEQ 4 SP1	CBEQ 2 IX+	RTS 1 INH	BLT 2 REL	CMP 2 IMM	CMP 2 DIR	CMP 3 EXT	CMP 3 IX2	CMP 4 SP2	CMP 2 IX1	CMP 3 SP1	CMP 1 IX
2	BRSET1 3 DIR	BSET1 2 DIR	BHI 2 REL		MUL 1 INH	DIV 1 INH	NSA 1 INH		DAA 1 INH		BGT 2 REL	SBC 2 IMM	SBC 2 DIR	SBC 3 EXT	SBC 3 IX2	SBC 4 SP2	SBC 2 IX1	SBC 3 SP1	SBC 1 IX
3	BRCLR1 3 DIR	BCLR1 2 DIR	BLS 2 REL	COM 2 DIR	COMA 1 INH	COMX 1 INH	COM 2 IX1	COM 3 SP1	COM 1 IX	SWI 1 INH	BLE 2 REL	CPX 2 IMM	CPX 2 DIR	CPX 3 EXT	CPX 3 IX2	CPX 4 SP2	CPX 2 IX1	CPX 3 SP1	CPX 1 IX
4	BRSET2 3 DIR	BSET2 2 DIR	BCC 2 REL	LSR 2 DIR	LSRA 1 INH	LSRX 1 INH	LSR 2 IX1	LSR 3 SP1	LSR 1 IX	TAP 1 INH	TXS 1 INH	AND 2 IMM	AND 2 DIR	AND 3 EXT	AND 3 IX2	AND 4 SP2	AND 2 IX1	AND 3 SP1	AND 1 IX
5	BRCLR2 3 DIR	BCLR2 2 DIR	BCS 2 REL	STHX 2 DIR	LDHX 3 IMM	LDHX 2 DIR	CPHX 3 IMM		CPHX 2 DIR	TPA 1 INH	TSX 1 INH	BIT 2 IMM	BIT 2 DIR	BIT 3 EXT	BIT 3 IX2	BIT 4 SP2	BIT 2 IX1	BIT 3 SP1	BIT 1 IX
6	BRSET3 3 DIR	BSET3 2 DIR	BNE 2 REL	ROR 2 DIR	RORA 1 INH	RORX 1 INH	ROR 2 IX1	ROR 3 SP1	ROR 1 IX	PULA 1 INH		LDA 2 IMM	LDA 2 DIR	LDA 3 EXT	LDA 3 IX2	LDA 4 SP2	LDA 2 IX1	LDA 3 SP1	LDA 1 IX
7	BRCLR3 3 DIR	BCLR3 2 DIR	BEQ 2 REL	ASR 2 DIR	ASRA 1 INH	ASRX 1 INH	ASR 2 IX1	ASR 3 SP1	ASR 1 IX	PSHA 1 INH	TAX 1 INH	AIS 2 IMM	STA 2 DIR	STA 3 EXT	STA 3 IX2	STA 4 SP2	STA 2 IX1	STA 3 SP1	STA 1 IX
8	BRSET4 3 DIR	BSET4 2 DIR	BHCC 2 REL	LSL 2 DIR	LSLA 1 INH	LSLX 1 INH	LSL 2 IX1	LSL 3 SP1	LSL 1 IX	PULX 1 INH	CLC 1 INH	EOR 2 IMM	EOR 2 DIR	EOR 3 EXT	EOR 3 IX2	EOR 4 SP2	EOR 2 IX1	EOR 3 SP1	EOR 1 IX
9	BRCLR4 3 DIR	BCLR4 2 DIR	BHCS 2 REL	ROL 2 DIR	ROLA 1 INH	ROLX 1 INH	ROL 2 IX1	ROL 3 SP1	ROL 1 IX	PSHX 1 INH	SEC 1 INH	ADC 2 IMM	ADC 2 DIR	ADC 3 EXT	ADC 3 IX2	ADC 4 SP2	ADC 2 IX1	ADC 3 SP1	ADC 1 IX
A	BRSET5 3 DIR	BSET5 2 DIR	BPL 2 REL	DEC 2 DIR	DECA 1 INH	DECX 1 INH	DEC 2 IX1	DEC 3 SP1	DEC 1 IX	PULH 1 INH	CLI 1 INH	ORA 2 IMM	ORA 2 DIR	ORA 3 EXT	ORA 3 IX2	ORA 4 SP2	ORA 2 IX1	ORA 3 SP1	ORA 1 IX
B	BRCLR5 3 DIR	BCLR5 2 DIR	BMI 2 REL	DBNZ 3 DIR	DBNZA 2 INH	DBNZX 2 INH	DBNZ 3 IX1	DBNZ 4 SP1	DBNZ 2 IX	PSHH 1 INH	SEI 1 INH	ADD 2 IMM	ADD 2 DIR	ADD 3 EXT	ADD 3 IX2	ADD 4 SP2	ADD 2 IX1	ADD 3 SP1	ADD 1 IX
C	BRSET6 3 DIR	BSET6 2 DIR	BMC 2 REL	INC 2 DIR	INCA 1 INH	INCX 1 INH	INC 2 IX1	INC 3 SP1	INC 1 IX	CLRH 1 INH	RSP 1 INH		JMP 2 DIR	JMP 3 EXT	JMP 3 IX2		JMP 2 IX1		JMP 1 IX
D	BRCLR6 3 DIR	BCLR6 2 DIR	BMS 2 REL	TST 2 DIR	TSTA 1 INH	TSTX 1 INH	TST 2 IX1	TST 3 SP1	TST 1 IX		NOP 1 INH	BSR 2 REL	JSR 2 DIR	JSR 3 EXT	JSR 3 IX2		JSR 2 IX1		JSR 1 IX
E	BRSET7 3 DIR	BSET7 2 DIR	BIL 2 REL		MOV 3 DD	MOV 2 DIX+	MOV 3 IMD		MOV 2 IX+D	STOP 1 INH	*	LDX 2 IMM	LDX 2 DIR	LDX 3 EXT	LDX 3 IX2	LDX 4 SP2	LDX 2 IX1	LDX 3 SP1	LDX 1 IX
F	BRCLR7 3 DIR	BCLR7 2 DIR	BIH 2 REL	CLR 2 DIR	CLRA 1 INH	CLRX 1 INH	CLR 2 IX1	CLR 3 SP1	CLR 1 IX	WAIT 1 INH	TXA 1 INH	AIX 2 IMM	STX 2 DIR	STX 3 EXT	STX 3 IX2	STX 4 SP2	STX 2 IX1	STX 3 SP1	STX 1 IX

INH Inherent
IMM Immediate
DIR Direct
EXT Extended
DD DIR/DIR
IX+D IX+/DIR
REL Relative
IX Indexed, No Offset
IX1 Indexed, 8-Bit Offset
IX2 Indexed, 16-Bit Offset
IMD IMM/DIR
DIX+ DIR/IX+
SP1 Stack Pointer, 8-Bit Offset
SP2 Stack Pointer, 16-Bit Offset
IX+ Indexed, No Offset with Post Increment
IX1+ Indexed, 1-Byte Offset with Post Increment
*Pre-byte for stack pointer indexed instructions

High Byte of Opcode in Hexadecimal: F
Low Byte of Opcode in Hexadecimal: 0
SUB 2 IX 1 — HC08 Cycles / Opcode Mnemonic / Number of Bytes / Addressing Mode

Fuente: FREESCALE SEMICONDUCTOR. CPU08 Central Processor Unit. 2006.

6.2.1. Arquitectura del set de instrucciones CISCKER

Al igual que la arquitectura de los microprocesadores HC08 y HC11, las instrucciones del CISCKER se organizan en un mapa de *Opcode*s, como lo muestra la Tabla 7 y se implementan teniendo en cuenta los modos de direccionamiento. Una misma operación con varios *Opcode*s será ejecutada con el modo de direccionamiento especificado en el lenguaje de ensamblador, esto significa que el orden y número de micro-instrucciones varía para realizar la lectura y escritura de datos.

Tabla 7. Mapa de *Opcode*s CISCKER

	Bit	Branch	Read Modify Write					Register Memory								
	INH	REL	INH	INH	INH	IX	EXT	IX+	IMM	DIR	IX	EXT	IMM	DIR	IX	EXT
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NOP	BHCC	TSX	NEGA	NEGB	NEG	NEG	SUBA	SUBA	SUBA	SUBA	SUBA	SUBB	SUBB	SUBB	SUBB
1	STOP	BHCS	TXS	CLRA	CLRB	CLR	CLR	SBCA	SBCA	SBCA	SBCA	SBCA	SBCB	SBCB	SBCB	SBCB
2	DABNZ	BHI	INS	INCA	INCB	INC	INC	ADDA	ADDA	ADDA	ADDA	ADDA	ADDB	ADDB	ADDB	ADDB
3	DBBNZ	BLS	DES	DECA	DECB	DEC	DEC	ANDA	ANDA	ANDA	ANDA	ANDA	ANDB	ANDB	ANDB	ANDB
4	BRN	BCC	INX	LSRA	LSRB	LSR	LSR	LDA	LDA	LDA	LDA	LDA	LDB	LDB	LDB	LDB
5	BRA	BCS	DEX	RORA	RORB	ROR	ROR	ORAA	ORA	ORA	ORA	ORA	ORB	ORB	ORB	ORB
6	BIL	BNE	PULX	ASRA	ASRB	ASR	ASR	EORA	EORA	EORA	EORA	EORA	EORB	EORB	EORB	EORB
7	BIH	BEQ	PSHX	LSLA	LSLB	LSL	LSL	ADCA	ADCA	ADCA	ADCA	ADCA	ADCB	ADCB	ADCB	ADCB
8	IDIV	BVC	PULA	ROLA	ROLB	ROL	ROL	CMPA	CMPA	CMPA	CMPA	CMPA	CMPB	CMPB	CMPB	CMPB
9	MUL	BVS	PULB	TSTA	TSTB	TST	TST	BITA	BITA	BITA	BITA	BITA	BSR		JSR	JSR
A	CLV	BPL	PSHA	TDX		BRSET	BRSET	STAA		STA	STA	STA		STB	STB	STB
B	SEV	BMI	PSHB			BRCLR	BRCLR		AIS	STS	STS	STS	AIX	STX	STX	STX
C	CLC	BGE	RTS	TBA	TAB	BCLR	BCLR			STD	STD	STD	CPX	CPX	CPX	CPX
D	SEC	BLT	RTI			BSET	BSET	CPD	LDS	LDS	LDS	LDS	LDX	LDX	LDX	LDX
E	CLI	BGT	WAIT	SBA	CBA	MOV	MOV	SUBD	SUBD	SUBD	SUBD	SUBD	CPD	CPD	CPD	CPD
F	SEI	BLE	SWI	ABA		JMP	JMP	ADDD	ADDD	ADDD	ADDD	ADDD	LDD	LDD	LDD	LDD
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
	INH	REL	INH	INH	INH	IX	EXT	IX+	IMM	DIR	IX	EXT	IMM	DIR	IX	EXT

6.2.1.1. Modos de direccionamiento

El set de instrucciones CISCKER utiliza siete modos de direccionamiento diferentes, incluyendo todos aquellos mencionados anteriormente: Inherente, Inmediato, Directo, Indirecto (utilizando el registro IX), Indexado y Relativo. Adicionalmente se incluyen las modalidades: Indexado con post-incremento (IX+) y Extendido.

- Indexado con post-incremento (IX+): A diferencia del direccionamiento indexado, este tipo de acceso a memoria no necesita de más datos. El valor a leer será el seleccionado por el registro IX y al finalizar la instrucción ese registro IX habrá incrementado en uno su valor. La utilización de este direccionamiento aumenta la eficiencia en el procesamiento secuencial de tablas.

- Extendido (EXT): Su funcionamiento es similar al direccionamiento Directo pero duplica el rango de direcciones que puede acceder al utilizar dos bytes de memoria después del *Opcode* como dirección.

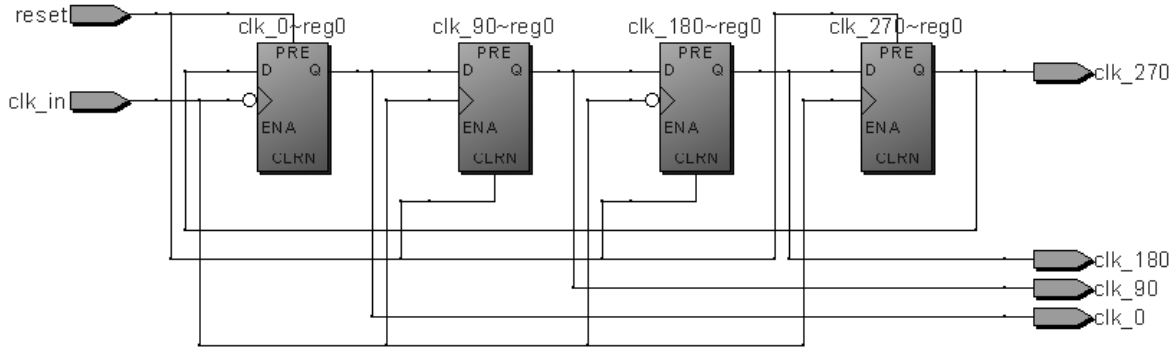
La implementación de cada modo de direccionamiento se logra en la distribución de micro-instrucciones para cada instrucción. Las señales derivadas de cada micro-instrucción seleccionan los registros que determinan la dirección de memoria de donde va a ser leído el operando y realizan la lectura de bytes de memoria subsiguientes cuando la instrucción lo requiera, por ejemplo, en direccionamiento Directo y Extendido.

6.2.2. Organización del hardware CISCKER

6.2.2.1. Reloj de cuatro fases

La arquitectura CISCKER funciona con cuatro pulsos desfasados noventa grados entre sí. Para generar estos cuatro flancos se utiliza el circuito basado en registros que se ilustra en la Figura 15. El circuito de desfase o *phaser* debe su funcionamiento a la alternación del flanco de reloj (descendente y ascendente) que rige el comportamiento de los cuatro registros en serie y a los valores de inicialización de los mismos.

Figura 15. Reloj de cuatro fases (*Phaser*)



Fuente: Altera Quartus II web edition, RTL Viewer.

6.2.2.2. Unidad de control

La arquitectura CISCKER permite instrucciones de alta complejidad, las cuales a partir de un solo *Opcode* modifican la ruta de datos de acuerdo a la función requerida, por ejemplo: las instrucciones de asignación a datos en memoria deben leer de una dirección, modificar el valor leído y finalmente re-escribirlo en la misma dirección. Para lograr esto, cada instrucción se decodifica en micro-instrucciones por un módulo llamado "*Instruction Decoder*". Una instrucción puede estar compuesta por una o más micro-instrucciones.

Las micro-instrucciones de la unidad de control CISCKER son las que se muestran en la siguiente tabla.

Tabla 8. micro-instrucciones CISCKER

	Nombre	Código	Descripción
p	Program Fetch	00	Lectura del siguiente byte de memoria de programa
r	Read Byte	01	Lectura de un byte de memoria
d	Dummy	10	Direccionamiento de siguiente byte de lectura
w	Write	11	Escritura de byte en memoria

Adicional al código de cada micro-instrucción existen cuatro bits de control que modifican funciones especiales que se deben realizar durante su ejecución:

- *Lookahead*: En esta micro-instrucción se registra el *Opcode* que está siendo leído de memoria.

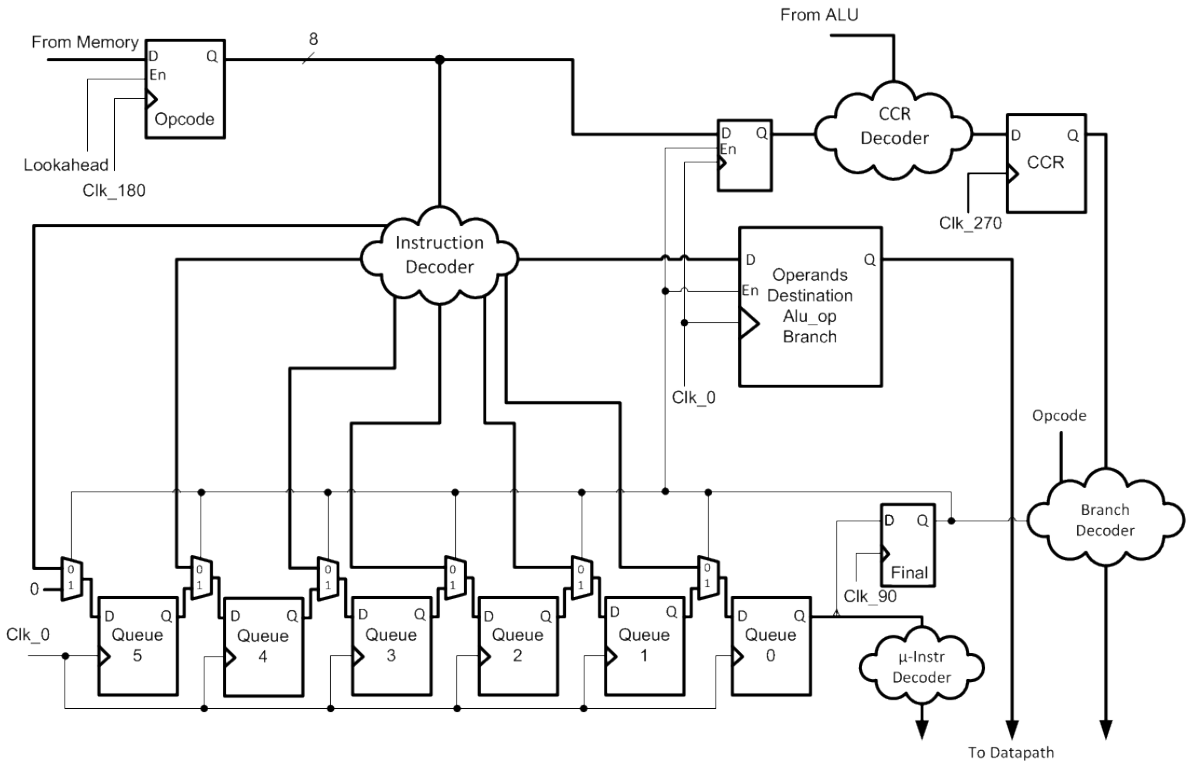
- Final: Determina cual es la última micro-instrucción para indicarle a la MCU que debe comenzar el registro de la nueva trama de micro-instrucciones.

- *Operation*: En esta micro-instrucción se registra el resultado de la operación de la ALU y el CCR.

- *Complement*: Bit de complemento a cada micro-instrucción. Este bit determina el comportamiento de una señal de control específica por cada micro-instrucción, permitiendo una mayor flexibilidad de configuración.

La Figura 16 es una representación de la unidad de control incluyendo el registro donde se almacena el *Opcode* leído de memoria, los módulos de decodificación y el circuito que administra la ejecución secuencial de las micro-instrucciones CISCKER. Este circuito está basado en un arreglo de registros de corrimiento de seis etapas llamado *Queue*. Cada una de las etapas almacena y permite el corrimiento del código y bits de control de las micro-instrucciones procedentes del decodificador del *Opcode* y se carga de forma paralela cada vez que la señal *final* lo indique.

Figura 16. Unidad de control CISCKER



El decodificador de micro-instrucciones convierte los códigos de dos bits en las señales de control de la ruta de datos CISCKER como lo muestra la Tabla 9, permitiendo la configuración de las señales indicadas con asterisco con el bit *Complement*.

Tabla 9. Decodificación de micro-instrucciones CISCKER

	µ-Instrucción	**pass_adder**	**pc_en**	**adder_cin**	**h_reset**	**mah_en**	**we**	**mdr_en**
00	**p**	1	1	1	*	1	0	1
01	**r**	1	*	1	0	0	0	1
10	**d**	*	0	0	1	0	0	1
11	**w**	1	*	1	0	0	1	0

Adicionalmente existen señales de control que dependen directamente del *Opcode* y no se ven afectadas por las micro-instrucciones durante la ejecución de la instrucción. Estas señales corresponden a los operandos de la instrucción, el destino de las instrucciones de asignación, la operación de la ALU y la condición de salto (en el caso de instrucciones *Branch*). Son decodificadas por los submódulos "*ALU operation decoder*", "*Branch Decoder*" y "*Main Decoder*" contenidos en el módulo "*Instruction Decoder*".

Además de la ejecución de las seis micro-instrucciones permitidas por la *Queue* de la unidad de control CISCKER, se desarrolló un diseño especial para la ejecución de otras instrucciones que requieren de mayor número de micro-instrucciones, como lo son: Multiplicación (MUL) y División (DIV) por iteraciones, e instrucciones de retorno y salto a la RSI (SWI y RTI). Una característica que se aprovecha de estas instrucciones es que su ejecución es iterativa, es decir, una misma trama de micro-instrucciones se repite sobre los mismos operandos un número de veces. Por último, se decodifica una trama que finaliza la instrucción, ya sea almacenando resultados o cargando el PC con una dirección de salto.

El circuito que logra esto se encarga de reconocer las instrucciones que requieren iteraciones de tramas y decodifica una primera trama teniendo en cuenta que es la primera vez que se ejecuta. Al mismo tiempo comienza un conteo de la aparición de la señal final, cargando nuevamente el registro *Queue* con una nueva o la misma (iterativa) trama de micro-instrucciones dependiendo del conteo. Una última trama permitirá que el registro donde se almacena el *Opcode* se llene con un nuevo valor leído de memoria, usando la señal de control *Lookahead* en cualquiera de sus micro-instrucciones, lo cual permitirá la continuación de la

ejecución de programa.

6.2.2.3. Unidad lógico-aritmética

El diseño de la ALU CISCKER cumple con las operaciones de asignación básicas (Suma, resta, AND, OR, XOR) y adiciona la ejecución de funciones de corrimiento de un bit en distintas variaciones. Como parte del diseño también se tiene en cuenta una función que no afecte el operando para permitir su uso directo. Para su control se utilizan tres señales, N0, N1 y Cin, donde las dos primeras niegan los operandos A y B respectivamente y Cin es la señal de acarreo. Con estas tres señales se realiza la selección de las operaciones lógicas y aritméticas básicas. Al igual que en la arquitectura RISCKER, la ALU CISCKER hace parte de una unidad llamada *operation* que contiene el hardware encargado de las operaciones de corrimiento. De esta forma las tres señales de control utilizadas para la ALU se aprovechan en los módulos adicionales como lo muestra la Tabla 10.

Tabla 10. Tabla de operaciones CISCKER.

Operación	N0	N1	Cin	Selector
Resta	0	1	1	000
Suma	0	0	0	000
AND	0	0	0	001
OR	0	0	0	010
XOR	0	0	0	011
NAND	1	1	0	001
NOR	1	1	0	011
Incrementar	0	0	1	000
Decrementar	0	1	0	000
Suma con *carry*	0	0	c	000
Resta con *carry*	0	1	~c	000
Corrimiento aritmético a la derecha	x	x	x	000
Corrimiento lógico a la derecha	x	1	x	101
Corrimiento lógico a la izquierda	x	0	x	100
Rotar a la derecha	x	1	x	110
Rotar a la izquierda	x	0	x	110
No afecta el operando	x	x	x	111
Negación lógica	1	0	1	000

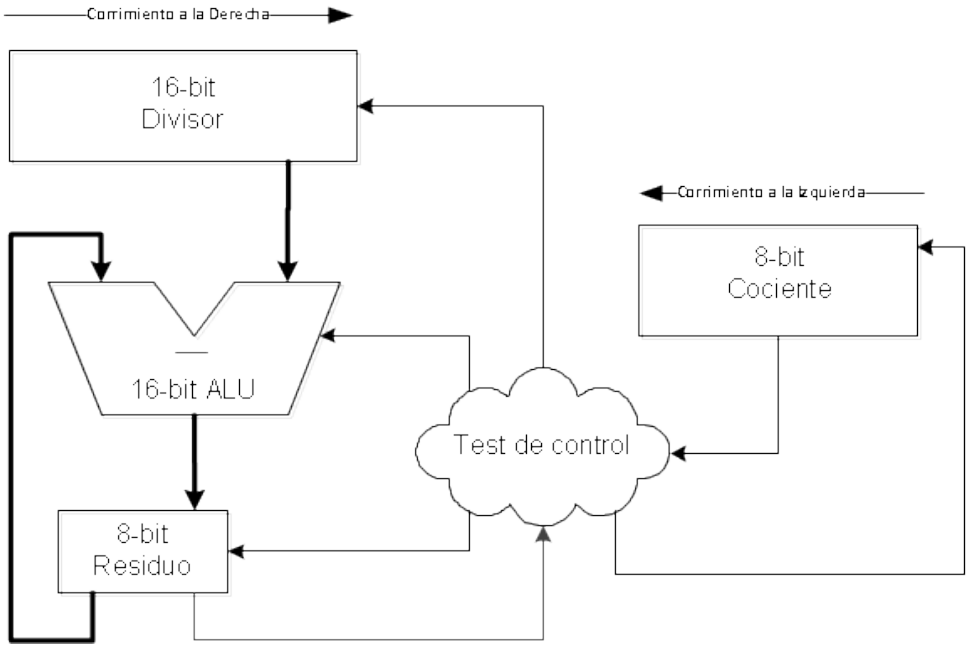
6.2.2.3.1. Multiplicador-divisor por iteraciones

Las funciones de multiplicación y división en la arquitectura CISCKER se implementaron mediante un circuito iterativo externo al módulo *operations*, a diferencia del hardware paralelo utilizado en la arquitectura RISCKER. Para hacerlo se diseñó un módulo de multiplicación y división conectado a los registros ACCA, ACCB y *Hidden* del procesador. La descripción de estos registros se encuentra en el numeral 6.2.2.5.

El circuito hace uso de un único sumador cuyas entradas varían dependiendo de los operandos y la operación.

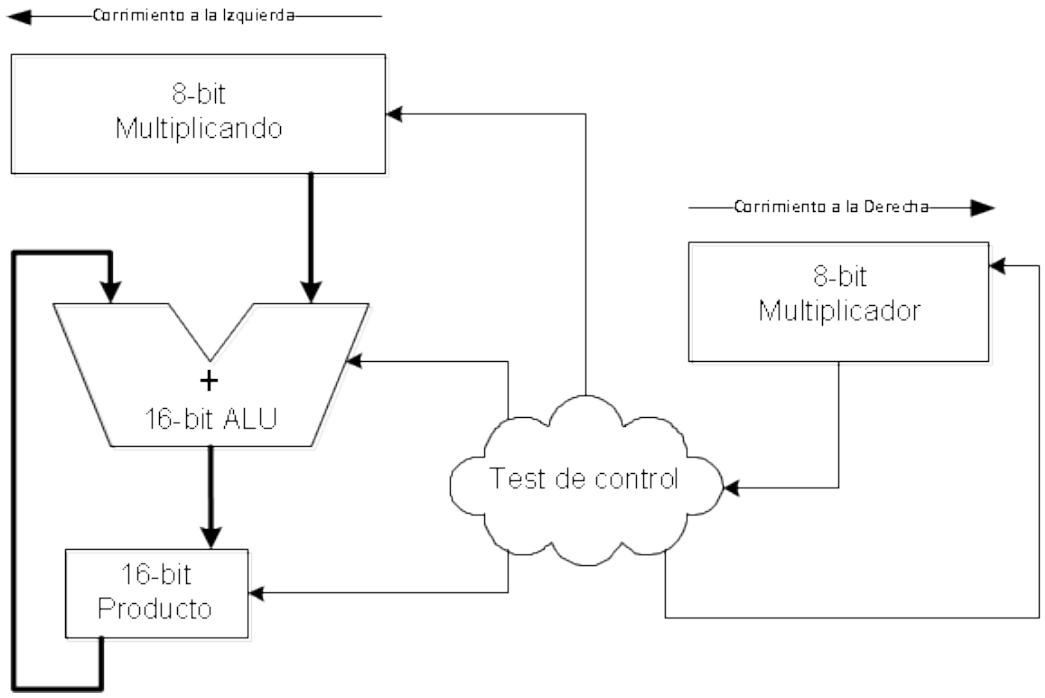
En la Figura 17 y Figura 18 se encuentran los diagramas de bloques de los algoritmos de división y multiplicación de ocho bits.

Figura 17. Diagrama de bloques de división por ciclos



Fuente: PATTERSON, A. David; HENNESSY , John L. Computer Organization and Design. Elsevier, 2005. p. 184

Figura 18. Diagrama de bloques de multiplicación por ciclos



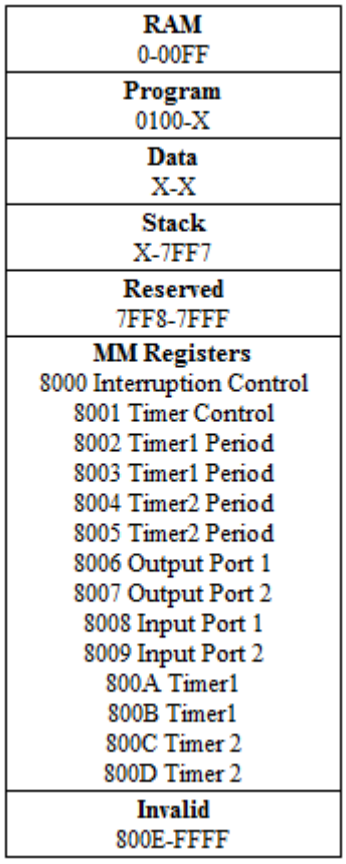
Fuente: PATTERSON, A. David; HENNESSY , John L. Computer Organization and Design. Elsevier, 2005. p. 177

Las instrucciones de multiplicación y división hacen uso de la función de ejecución cíclica de la unidad de control CISCKER, precedida por una iteración de micro-instrucciones que almacenan el multiplicador/divisor en el registro *Hidden* permitiendo almacenar el resultado en los registros ACCA y ACCB, formando así un resultado de 16 bits al utilizar el registro ACCD en una próxima instrucción.

6.2.2.4. Unidad de memoria

La unidad de memoria Von Newmann que hace parte de la arquitectura CISCKER requiere una subdivisión de las direcciones para preestablecer distintas funciones. La Figura 19 muestra la distribución de memoria CISCKER.

Figura 19. Diagrama de distribución de memoria CISCKER



Para esto se designan las primeras doscientas cincuenta y seis direcciones como RAM o memoria de acceso aleatorio rápido. El rango de 0000 a 00FF puede ser accedido con el modo de direccionamiento Directo, utilizando únicamente un byte como dato de dirección, permitiendo ejecutar la lectura de estas direcciones más rápidamente.

Por defecto, el código de programa comienza en la dirección 0100, esto hace parte de la configuración inicial del procesador y del estado de retorno en caso de reinicio, el registro PC se carga con la dirección 0100 para dar comienzo a la ejecución del código.

La parte de memoria que se destina al *Stack* o Pila son los valores inferiores a la

dirección 7FF7. El funcionamiento de la Pila automatiza el uso de estas direcciones facilitando la resta en escritura y el incremento en lectura a la dirección del puntero de pila SP.

Los campos que se encuentran entre la última dirección del código de programa y el límite mínimo por debajo de la dirección inicial de la Pila están disponibles. Al usuario se le permite el uso de estas localidades de memoria pero el control del rango de direcciones debe ser predefinido para evitar la escritura indeseada de datos almacenados durante la ejecución de programas.

Las direcciones reservadas desde 7FF8 a 7FFF son utilizadas en el algoritmo de reconocimiento de interrupciones. Cuando la instrucción SWI se ejecuta hace uso de una de estas cuatro direcciones de acuerdo a la interrupción ocurrida para obtener la dirección en la cual se encuentra la RSI. El usuario debe modificar los valores en estas localidades de memoria antes de habilitar cualquiera de las interrupciones para evitar la ubicación incorrecta del contador de programa.

Adjunto a la memoria, desde la dirección 8000 se encuentran los registros mapeados en memoria para configuración e interfaz al exterior del procesador. Las direcciones mayores a las de estos registros son inválidas.

6.2.2.4.1. Registros mapeados en memoria

La arquitectura del procesador CISCKER requiere la implementación de registros MM (*Memory Mapped*), es decir, registros de configuración, interconectados no solo para su escritura y lectura, sino también a módulos externos como: Temporizadores, Banderas de Excepciones e I/O; que se encuentran “Mapeados” en memoria. Esto significa que el acceso a estos registros se hace a través del mismo direccionamiento que el de cualquier dato en localidades de memoria, pero físicamente son registros adjuntos a la memoria y habilitados por el bit más significativo de la señal de dirección en el registro MAR (*Memory Address Register*).

En el microprocesador CISCKER se encuentran los siguientes registros MM:

- Puertos I/O: Cuenta con dos registros de salida conectados directamente a salidas del microcontrolador y dos registros de entrada, que tienen función únicamente de lectura, la entrada de estos registros está conectada a través de sincronizadores para evitar meta-estabilidad en estas señales.

- Registros para temporizadores: Para el control y uso de los temporizadores son necesarios dos registros acumuladores por temporizador y un registro de configuración. En la descripción de dos temporizadores, cada uno tiene dos registros de lectura y escritura que conforman un valor de diez y seis bits con el periodo de desbordamiento del mismo y dos registros de solo lectura que es el conteo actual de diez y seis bits. El registro de configuración contiene las siguientes variables partiendo de los bits más significativos: dos bits de *prescaler*, un habilitador y un *reset* para cada temporizador. Las funciones de estos registros se profundizan en el numeral 6.2.2.6. Interrupciones y temporizadores

- Configuración de interrupciones: Para el manejo de interrupciones se cuenta con un módulo de control por banderas explicado con detalle en el numeral 6.2.2.6. Interrupciones y temporizadores. Estas banderas se rigen por un comportamiento donde el usuario puede borrarlas o deshabilitarlas, según la configuración de las señales contenidas en este registro. Para la configuración predeterminada del CISCKER se cuenta con cuatro señales de *reset* de bandera de interrupción en el siguiente orden, temporizador1, temporizador2, interrupción externa 1, interrupción externa 2; y con cuatro habilitadores activos altos de las mismas interrupciones.

6.2.2.5. Registros

A continuación se describen los registros de propósito general y especial de la arquitectura CISCKER.

- ACCA: Acumulador A de propósito general como operando de instrucciones de asignación.

- ACCB: Acumulador B de propósito general.

- ACCD: La unión entre los acumuladores A y B forman el acumulador D, direccionable como operando en operaciones de asignación de 16bits.

- IX: Registro Índice, puntero para direccionamientos indexados.

- SP: Puntero de la pila. Este registro es modificado automáticamente por el hardware en el reconocimiento de interrupciones.

- *Condition Code Register* (CCR): Registro de estado que contiene las banderas modificadas por instrucciones de asignación.

    - *Carry* (C): Representa un acarreo en operaciones aritméticas o de corrimiento.

    - *Overflow* (V): Represeta un desbordamiento en operaciones aritméticas.

    - *Zero* (Z): El resultado de la ALU es igual a cero.

    - *Negative* (N): El resultado de la ALU es negativo.

    - *Interrupt Mask* (I): Máscara de Interrupción.

    - *Half-Carry* (H): Representa un acarreo en operaciones aritméticas de la mitad del tamaño del dato. Para uso en operaciones en BCD.

- *Hidden*[26]: Registro escondido utilizado en instrucciones de paso de datos (MOV) o de multiplicación y división para mantener un operando durante la ejecución de las iteraciones. Este registro no se incluye dentro de la documentación al programador pero hace parte de la estructura funcional interna del microprocesador.

---

[26] LIPOVSKI, G. Jack. Introduction to Microcontrollers. United States of America. Academic Press, 1999. p. 32

- *Memory Address Register* (MAR) (16bits): El registro de dirección de memoria retiene la dirección que va a ser leída de la memoria. Su función es mantener este valor durante el procesamiento de la siguiente dirección para evitar lecturas inadecuadas.

- *Memory Data Register* (MDR) (16bits): El registro de dato de memoria almacena el resultado de la ALU y lo ubica en el bus de datos de escritura de la memoria.

- *Program Counte*r (PC) (16bits): El contador de programa contiene la dirección de la instrucción de programa que se está ejecutando.

- Operation Code (*Opcode*) (8bits): El registro de código de operación solo almacena los bytes leídos de memoria que son códigos de operación.

- *Memory Data* (MM) (16bits): Esta señal presenta los bytes leídos de memoria. Los ocho bits más significativos se almacenan en el registro MH (*Memory High*) asimilando el funcionamiento de un registro de corrimiento al leer el segundo byte de un dato o dirección de memoria que se mantendrá en el registro de salida de la memoria, ML (*Memory Low*). MH es borrado cuando la lectura que se realiza es de un byte.

- *Memory Address Holder* (MAH) (16bits): El registro de retención de dirección de memoria almacena los bytes leídos de memoria que son utilizados en instrucciones de direccionamiento Indexado, Relativo, Directo ó Extendido.

6.2.2.6. Ruta de datos

La Figura 20 es una representación esquemática de la ruta de datos del microprocesador CISCKER. La estructura del diagrama permite identificar las unidades lógico-aritmética, de memoria (Von-Neumann), control y de entradas y salidas, como también la sección encargada de determinar la dirección de lectura de memoria. A esta última unidad se le llama sumador de direcciones ya que el componente principal es un sumador y tiene una única salida conectada al registro MAR, determina la siguiente localidad de memoria a leer.

El funcionamiento por defecto de esta unidad es el de incrementar el valor del contador de programa. Sin embargo, el valor de dirección en esta arquitectura debe variar durante la ejecución de la mayoría de las instrucciones para llevar a cabo la lectura de bytes de acuerdo al modo de direccionamiento de la instrucción. Es por esto que la unidad de suma de direcciones permite que un dato de memoria (MAH), el registro índice (IX) o el puntero de la pila sean operandos para determinar la dirección, como también están las opciones de incrementar (+1), disminuir (-1) y sostener (0).

En la Figura 20 también están numeradas las señales de control provenientes de la MCU y se muestran las señales de reloj, provenientes del circuito *phaser*, que controlan cada registro. La Tabla 11. Señales de control CISCKER describe las señales de control que habilitan los registros de la ruta de datos o configuran los multiplexores y demás bloques lógicos para las distintas instrucciones.

Figura 20. Ruta de datos CISCKER

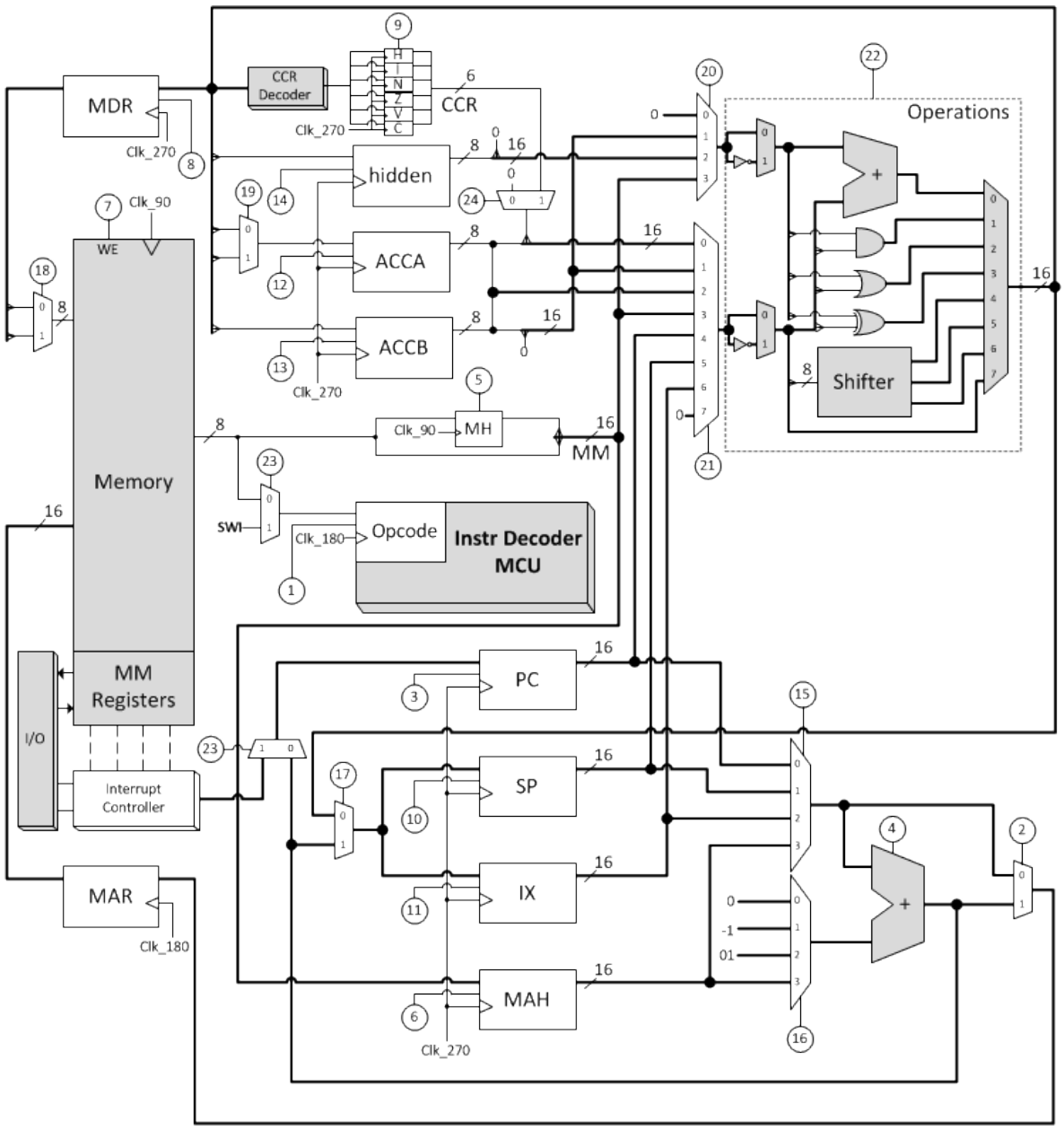


Tabla 11. Señales de control CISCKER

Número	Nombre	Descripción
1	opcode_en	Habilitador del registro de opcode
2	pass_adder	Selecciona la fuente del registro MAR
3	pc_en	Habilitador del registro PC
4	adder_cin	Acarreo de entrada al sumador de direcciones
5	h_reset	Reset del registro MH
6	mah_en	Habilitador del registro de memoria para direcciones
7	we	Habilitador de escritura de la memoria
8	mdr_en	Habilitador del registro MDR
9	ccr_en	Habilitador del registro CCR
10	sp_en	Habilitador del registro SP
11	ix_en	Habilitador del registro IX
12	acca_en	Habilitador del ACCA
13	accb_en	Habilitador del ACCB
14	hidden_en	Habilitador del registro HIDDEN
15	adder_src_a[1:0]	Selecciona el operando A del sumador de direcciones
16	adder_src_b[1:0]	Selecciona el operando B del sumador de direcciones
17	adder_alu	Selecciona la fuente de los registros IX,SP
18	wd_src[1:0]	Selecciona el dato que se va a escribir en memoria
19	acca_src	Selecciona la fuente del siguiente valor de ACCA
20	alu_src_a[2:0]	Selecciona el operando A de la ALU
21	alu_src_b[1:0]	Selecciona el operando B de la ALU
22	alu_op[5:0]	Determina la operación de la ALU
23	Interruption	Señal de interrupción
24	wd_ccr	Selecciona CCR como operando de la ALU para pasar a MDR

6.2.2.7. Interrupciones y temporizadores

Para la implementación de interrupciones en una arquitectura CISC se deben plantear dos módulos de hardware encargados del reconocimiento y ejecución completa de la RSI, de manera que el usuario no esté encargado de almacenar el estado del procesador al momento de la interrupción, sino que el hardware automáticamente utilice la pila y modifique las banderas apropiadas.

El primer módulo es el de control ó manejo de interrupciones cuyo objetivo es reconocer cuando y qué interrupción hubo, este módulo le entrega al procesador una señal de interrupción y una dirección donde se encuentra el vector de interrupción, siendo este vector la dirección extendida donde se encuentra la RSI.

Una característica notable de este circuito es la facilidad de escalabilidad y configuración, el CISCKER predeterminado cuenta con cuatro interrupciones, dos externas y dos por temporizador, pero fácilmente se podría cambiar esta configuración por un número distinto y por fuentes distintas de interrupción únicamente determinando la dirección del vector correspondiente.

La segunda parte correspondiente a la ejecución de la interrupción hace parte de la ruta de datos donde la interrupción opera únicamente forzando la instrucción SWI (*Software Interruption)* en el registro de *Opcode* en vez de la siguiente instrucción, al igual que la arquitectura HC08[27].

La instrucción SWI tiene un funcionamiento especial comparado con las instrucciones comunes, no solo porque es forzada en el registro *Opcode* sino porque su ejecución es cíclica al igual que la de las instrucciones MUL y DIV, durante las primeras etapas cíclicas de esta instrucción se almacenan los registros PC, IX, ACCA y CCR, en ese orden, (a diferencia de la arquitectura HC08 que almacena únicamente la parte menos significativa del registro IX) con las micro-instrucciones d, w y w, posteriormente las micro-instrucciones ejecutadas serán de lectura de memoria, donde se encuentra el vector correspondiente, y el salto a la RSI. La dirección donde se encuentra el vector es fija, proviene de hardware, por lo que es posible direccionarla en la ruta de datos sin necesidad de ciclos adicionales.

6.3. DISEÑO DEL IDE X-ISCKER

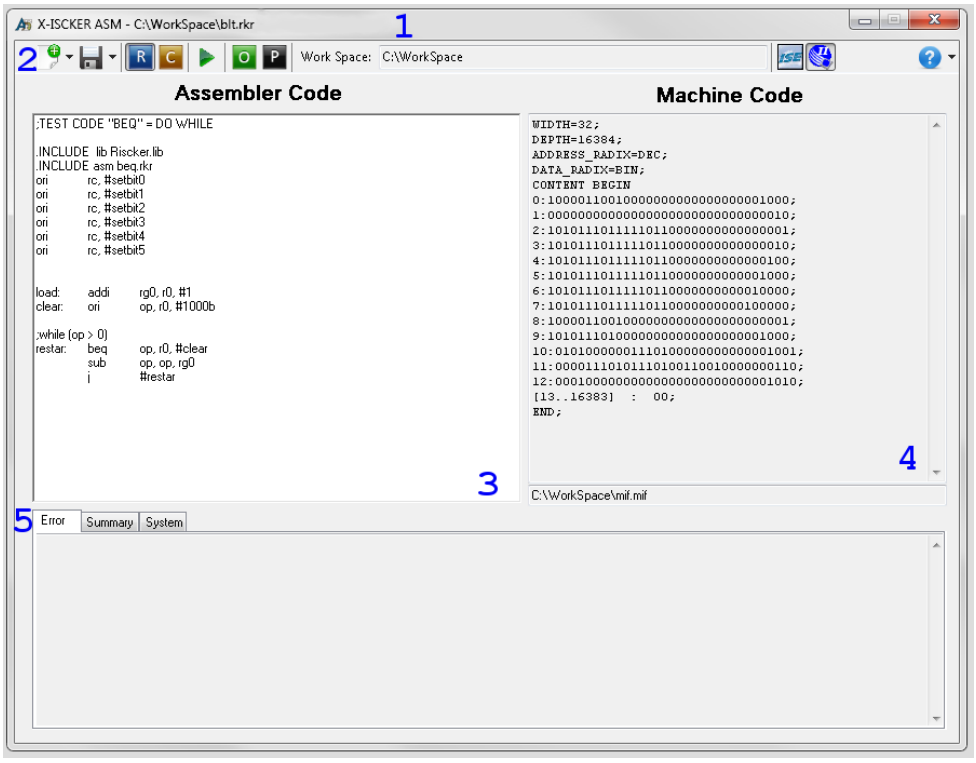
La plataforma X-ISCKER facilita la programación y uso de los microprocesadores RISCKER y CISCKER mediante un entorno de desarrollo integrado que lo conforman un ensamblador (X-ISCKER ASM), programador (X-ISCKER *Programmer*) y la herramienta de emulación (X-ISCKER *Observer*).

6.3.1. X-ISCKER ASM

[27] FREESCALE SEMICONDUCTOR. CPU08 Central Processor Unit. 2006. p. 23

La interfaz principal del X-ISCKER IDE presenta los campos y funciones correspondientes a la labor de ensamblador X-ISCKER ASM. La Figura 21 muestra la ventana principal del programa. A continuación se describen sus funciones básicas.

Figura 21. X-ISCKER IDE



1. Barra de Título: Muestra la dirección del archivo “*.ckr” o “*.rkr” que se está modificando.

2. Barra de Herramientas:

a. Abrir: El botón abre un nuevo archivo en blanco para empezar a editar en la caja de código. Ofrece un menú con tres opciones.

→ RISCKER ASM: Abre un archivo de extensión .rkr y selecciona la arquitectura para el IDE como RISCKER.

→ CISCKER ASM: Abre un archivo de extensión .ckr y selecciona la arquitectura para el IDE como CISCKER.

→ *New blank file*: Crea un archivo vacío, es la acción por defecto del botón.

b. Guardar: El botón guarda el contenido de la caja de código y de memoria en archivos existentes o consulta la ubicación para crearlos.

→ *Assembly File*: Guarda el archivo contenido en la caja de código con extensión .ckr o .rkr según sea la arquitectura seleccionada.

→ *Memory File*: Guarda el contenido de la caja de memoria en un archivo de inicialización de memoria *.mif o *.mem.

c. Selector de Arquitectura RISCKER CISCKER: Selección del algoritmo de ensamble para la arquitectura de set de instrucciones RISCKER/CISCKER. Esta selección también modifica el ambiente de desarrollo XISCKER IDE de manera que el código se guarde en un archivo "*.rkr" o "*.ckr" y para identificar la interfaz con la que se va a lanzar el XISCKER *Observer* (detallado en el numeral 6.3.3).

d. Ejecutar ensamblador: Inicia la interpretación del programa en lenguaje de ensamblador que se encuentra en la caja de código (*Assembler Code*), genera el archivo de inicialización de memoria de instrucciones y lo muestra en la caja de memoria (*Machine Code*).

e. *Programmer*: Lanza una ventana adicional correspondiente a la herramienta XISCKER *Programmer* (detallada en el numeral 6.3.2).

f. *Observer*: Lanza una ventana adicional correspondiente a la herramienta X-ISCKER *Observer* (detallada en el numeral 6.3.3).

g. *Workspace*: Muestra la ruta del espacio de trabajo actual.

h. Xilinx Altera: Al elegir uno de los dos fabricantes se modifican los parámetros del XISCKER *Programmer* y el formato en que se crea el archivo de memoria.

i. Ayuda: Este botón lanza una ventana con información acerca del XISCKER IDE.

→ *User Guide*: Esta guía de usuario.

→ Wiki: Enlace de la wiki XISCKER, contiene los archivos y avances más recientes del proyecto.

→ *About*: Lanza la ventana con información acerca de la versión y desarrollo del software XISCKER IDE. Es la opción por defecto del botón.

3. Caja de Código: Caja de texto para editar el código en lenguaje ensamblador de la arquitectura seleccionada.

4. Caja de memoria: Caja de texto que muestra el archivo de inicialización de memoria generado por el proceso de ensamble. En la parte inferior se muestra la ruta del archivo que se esté mostrando.

5. Pestañas de Procesos: Pestañas que muestran información de los procesos de ensamble y configuración del dispositivo.

a. Error: Muestra las líneas de código con errores generados durante el proceso de ensamble o la programación del dispositivo. El número de la línea se especifica en la pestaña *Summary*.

b. *Summary*: Muestra un resumen de las diferentes etapas del proceso de ensamble. Asigna un número a cada línea de código para facilitar su depuración.

c. *System*: Muestra información sobre los comandos para generar el archivo de configuración del dispositivo, los errores y las advertencias que estos generan.

6.3.1.1. Sintaxis del RISCKER ASM

Una Instrucción RISCKER - ASM está compuesta de hasta cuatro campos:

[Etiqueta] : [Operación] [Operandos] ; [Comentario]

[Etiqueta]: Nombre simbólico de la localidad de memoria en que se encuentra la instrucción.

[Operación]: Indica la operación que se va a ejecutar

[Operandos]: El campo operandos tiene hasta cuatro sub-campos dependiendo del tipo de operación.

Instrucción Completa:

[Etiqueta] : [Operación] [Rd] , [Rs] , [Rt] , #[Imm/Dir] ; [Comentario]

[Rd] Registro Destino: Indica el registro en el cual se guardará el resultado de la operación.

[Rs] Registro Fuente (*Source*) Indica el registro en el que se encuentra un dato a operar.

[Rt][28] Registro Fuente Indica un segundo registro con un dato a operar, sólo si la operación lo requiere.

#[Imm/Dir] [29] Valor Inmediato o Dirección Indica un número o dirección.

---

[28] Este campo dependen del tipo de operación.

[29] Este campo depende del tipo de operación.

[Comentario] Se escribe al final de la instrucción precedido por un “;”.

Ejemplos:

```
beq    rg1, rg0, #16 ;Salto condicional, si rg1 es igual a rg0 a la dirección 16

lw     rg3, r0, #0   ;Cargar el registro rg3 con el dato Memoria[r0+0]

inc    rg0,rg1       ;Incrementar rg1 y guardar el resultado en rg0

add    rg0,rg1,rg2   ;Sumar rg1 con rg3 y guardar el resultado en rg0
```

6.3.1.2. Sintaxis del CISCKER ASM

Una Instrucción CISCKER - ASM está compuesta de hasta cinco campos:

[Etiqueta] : [Operación] [Operando] , [MD] ; [Comentarios]

[Etiqueta]: Nombre simbólico de la localidad de memoria en que se encuentra la instrucción.

[Operación]: Indica la operación que se va a ejecutar

[Operando]: Indica el operando con el que se va a ejecutar la operación.
Se precede de “#” si el operando es un número.
Se precede de “$” si el operando es una dirección.

[Modo de Direccionamiento]: Se precede de “,”. Indica el modo de direccionamiento de la instrucción de forma explícita: IX o IX+. Los demás modos de direccionamientos se infieren de la operación y los operandos.

[Comentarios]: Se escribe al final de la instrucción y precedido por un “;”.

Ejemplos:

```
ADDA #10, IX      ; Sumar ACCA con Memoria[IX+10]

INCA              ; Incrementar ACCA
```

6.3.1.3. Algoritmo de ensamble

Para convertir los programas escritos en X-ISCKER ASM a su respectivo lenguaje de máquina, se desarrolló un algoritmo de ensamble para cada arquitectura que lleva a cabo las tareas básicas de un programa ensamblador, como son la interpretación de los distintos códigos de operación, operandos, etiquetas y la resolución de direcciones. A continuación se hará una descripción de las características más importantes de este desarrollo.

Previo al proceso de ensamble de los lenguajes X-ISCKER ASM se realiza la interpretación de las directivas que definen características de dicho proceso. A continuación se muestra la sintaxis de las directivas para el ensamblador X-ISCKER:

.[DIRECTIVA] [Operando] ; [Comentario]

.[DIRECTIVA]: Mnemónico de la directiva precedido de un punto. Se escribe en mayúscula sostenida.

[Operando] El campo operando tiene hasta cuatro sub-campos dependiendo de la directiva.

[Comentarios]: Se escribe al final de la instrucción y precedido por un ";".

El ensamblador X-ISCKER cuenta con tres directivas:

EQU: Asigna un nombre simbólico a una constante.
Sintaxis: .EQU valor1 1010101b
Descripción: Asigna la etiqueta "valor1" al número binario "1010101".

ORG: Configura la dirección de inicio del código de programa.
Sintaxis: .ORG 100h
Descripción inicializa el contador de direcciones con el valor hexadecimal 100.

INCLUDE: Importa líneas de código desde un archivo externo. Este archivo no debe contener líneas vacías o directivas.

La directiva INCLUDE puede importar archivos *.rkr y *.ckr siempre y cuando estos no contengan líneas vacías, comentarios o directivas.

Ejemplo: .INCLUDE asm rutina1.rkr

También puede importar un archivo de declaración de constantes el cual se compone únicamente de directivas EQU, sin comentarios, ni líneas vacías.
Ejemplo: .INCLUDE lib Riscker.lib

La resolución de directivas, así como la eliminación de los comentarios y líneas vacías se lleva a cabo en la función *TrimLinesWithoutOperation*, la cual retorna un vector con cada línea de operación detectada. Esta función se ejecuta en la traducción de ambas arquitecturas.

La función principal del algoritmo de ensamble es generar las instrucciones binarias de un determinado código escrito en lenguaje ensamblador, para ello se ejecutan dos pasadas al vector generado por la función *TrimLinesWhitoutOperation*, y se decodifica línea por línea cada representación simbólica, construyendo su equivalente binario.

6.3.1.3.1. CISCKER ASM

El primer paso para generar las instrucciones binarias partiendo del CISCKER ASM es obtener cada campo de la instrucción simbólica como cadenas de caracteres o *strings* separados. Para ello, se utiliza la función *split* la cual busca un símbolo en un *string* y almacena lo que está a la izquierda del símbolo en la posición cero de un vector y lo que está en la derecha en la posición uno del vector. Primero se descartan los comentarios de línea dividiendo el *string* por el símbolo “;” y desechando la posición uno. Luego se divide por el símbolo “,” y se obtiene el direccionamiento explícito, se divide por el símbolo “:” y se obtiene la etiqueta, finalmente se divide por el símbolo “ “ (espacio) y se obtienen operación y operando.

Una vez detectados cada campo de la instrucción se procede a la interpretación de la instrucción. Primero se convierte el operando a base hexadecimal y se calcula el direccionamiento implícito, la Tabla 12 muestra las condiciones para determinar el modo direccionamiento de forma implícita. Luego se hace la validación de mnemónicos y administración del contador de direcciones, ya que las instrucciones en esta arquitectura son de tamaño variable. Al final de la primera pasada se convierten operandos y operaciones conocidas a lenguaje de máquina y se dejan las etiquetas intactas.

Tabla 12. Determinación de modos de direccionamiento implícito CISCKER

Modo de Direccionamiento	Indicador Léxico	Condición
Inmediato	#	Si el operando se antecede por un “**#**”
Extendido	$	Si el operando se antecede por un “$“ y es superior a 8 bits
	Opcode	Si es una instrucción de salto extendido: JUMP
Directo	$	Si el operando se antecede por un “$“ y es igual o inferior a 8 bits
Relativo	*Opcode*	Si es una instrucción de salto condicionado

Finalmente se hace la segunda pasada para encontrar o calcular las direcciones de las etiquetas para ser convertidas a la dirección absoluta correspondiente, en lenguaje de máquina. La Figura 22 muestra un diagrama de flujo que ilustra el proceso descrito.

6.3.1.3.2. RISCKER ASM

El conjunto de instrucciones del procesador RISCKER, además de ser reducido, es de tamaño fijo, esto significa que cada instrucción equivale a una localidad en la memoria de instrucciones. Esta característica de la arquitectura facilita el proceso de ensamble. El proceso se lleva a cabo en dos pasadas. En la primera se buscan las etiquetas y se llena la tabla de símbolos, esta tabla contiene cada etiqueta y su valor binario correspondiente, en direcciones absolutas.

Figura 22. Diagrama de flujo del ensamble CISCKER-ASM

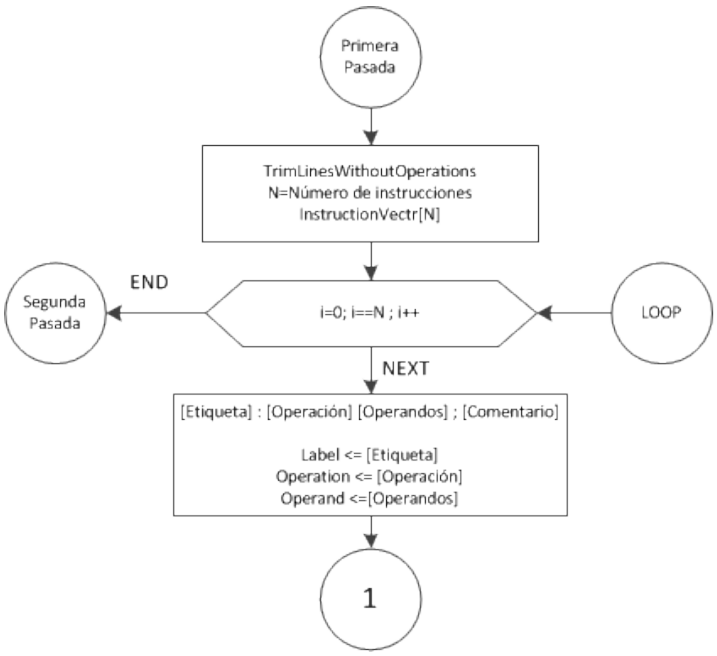


Figura 22. (Continuación)

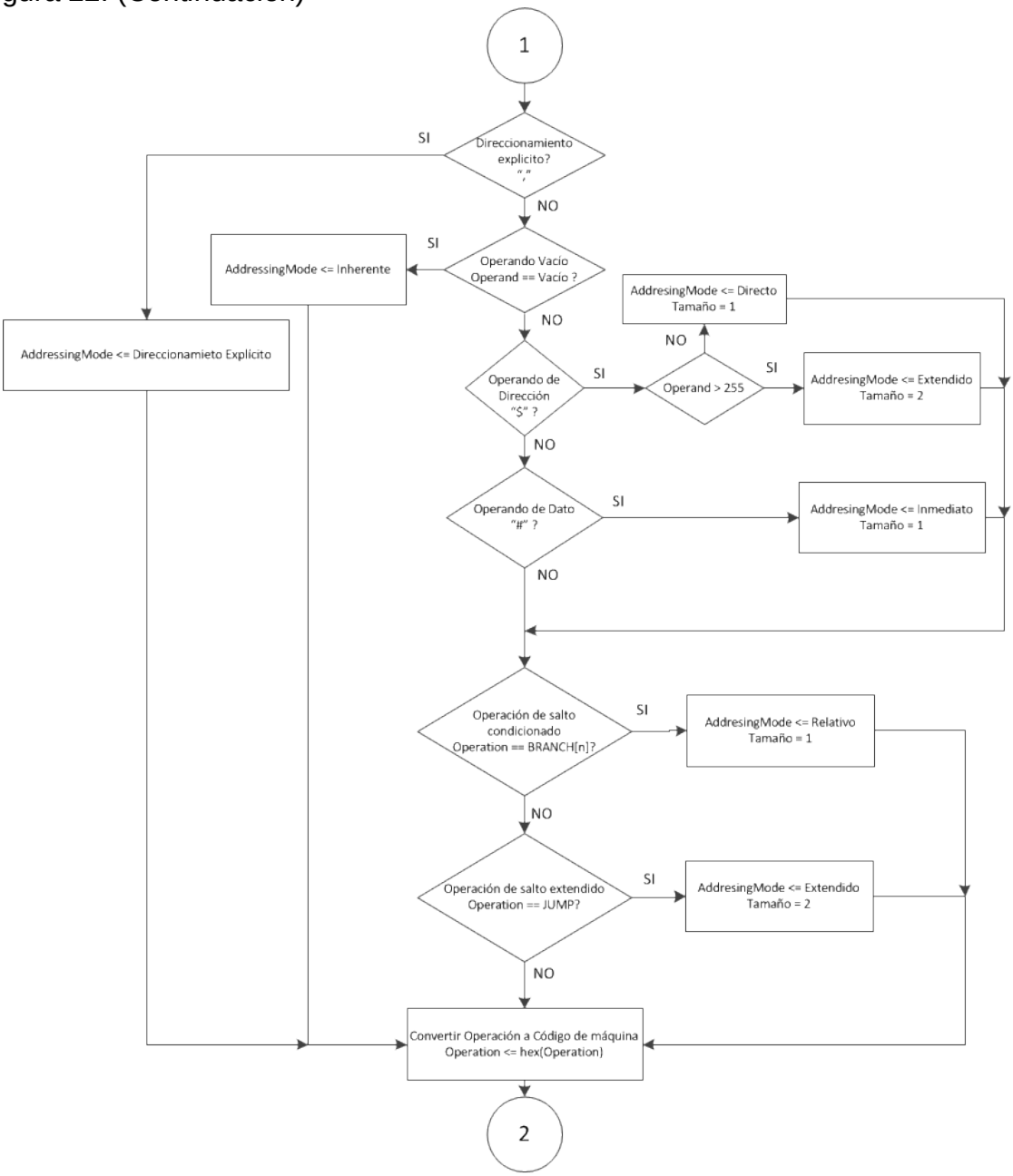


Figura 22. (Continuación)

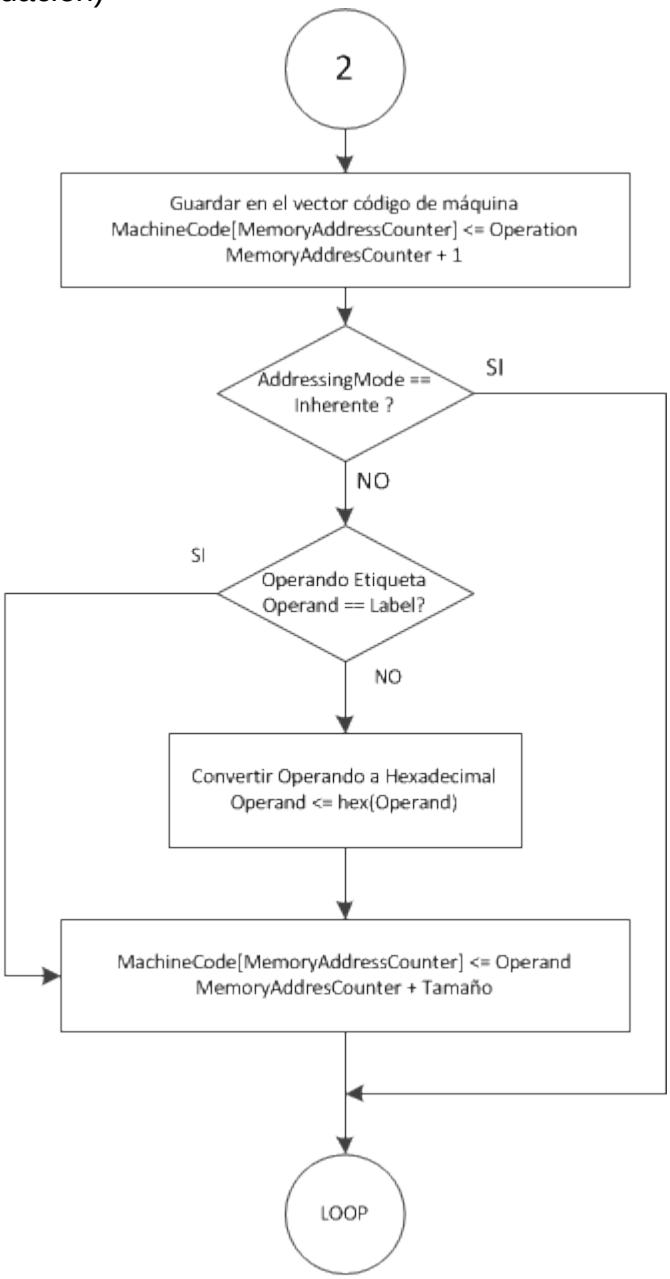
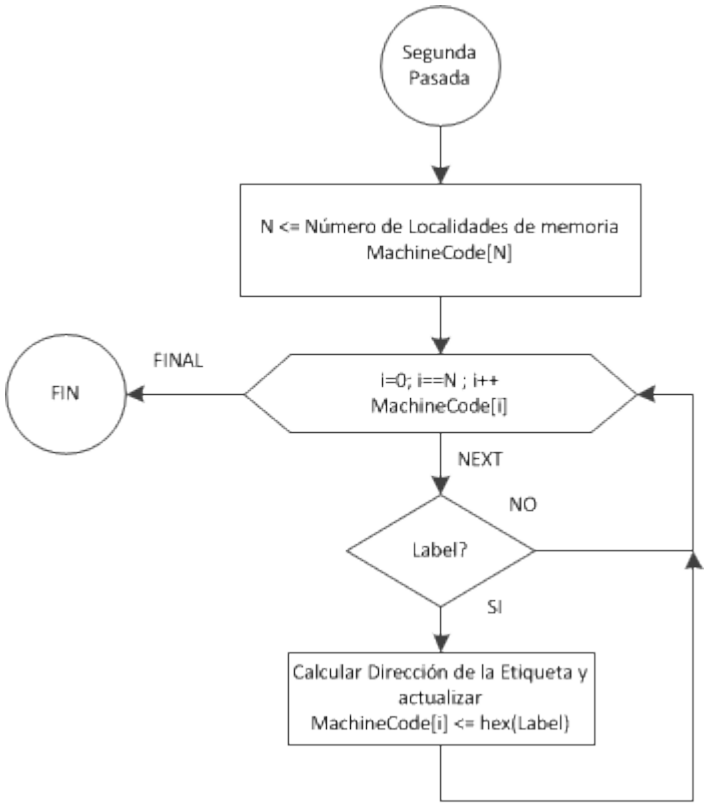


Figura 22. (Continuación)



En la segunda pasada se separan los comentarios mediante la función *split*, buscando el símbolo “;”, luego se separa cada uno de los campos de la instrucción: Inmediato y Operandos con el símbolo “#” y Operación con el símbolo “ “ (espacio). Se resuelven los operandos Rd, Rt, Rs. Se valida e interpreta el *Opcode* y finalmente se reemplaza cada símbolo con su valor binario correspondiente. La Figura 23 muestra un diagrama de flujo que ilustra el proceso descrito.

Figura 23. Diagrama de flujo del ensamble RISCKER ASM

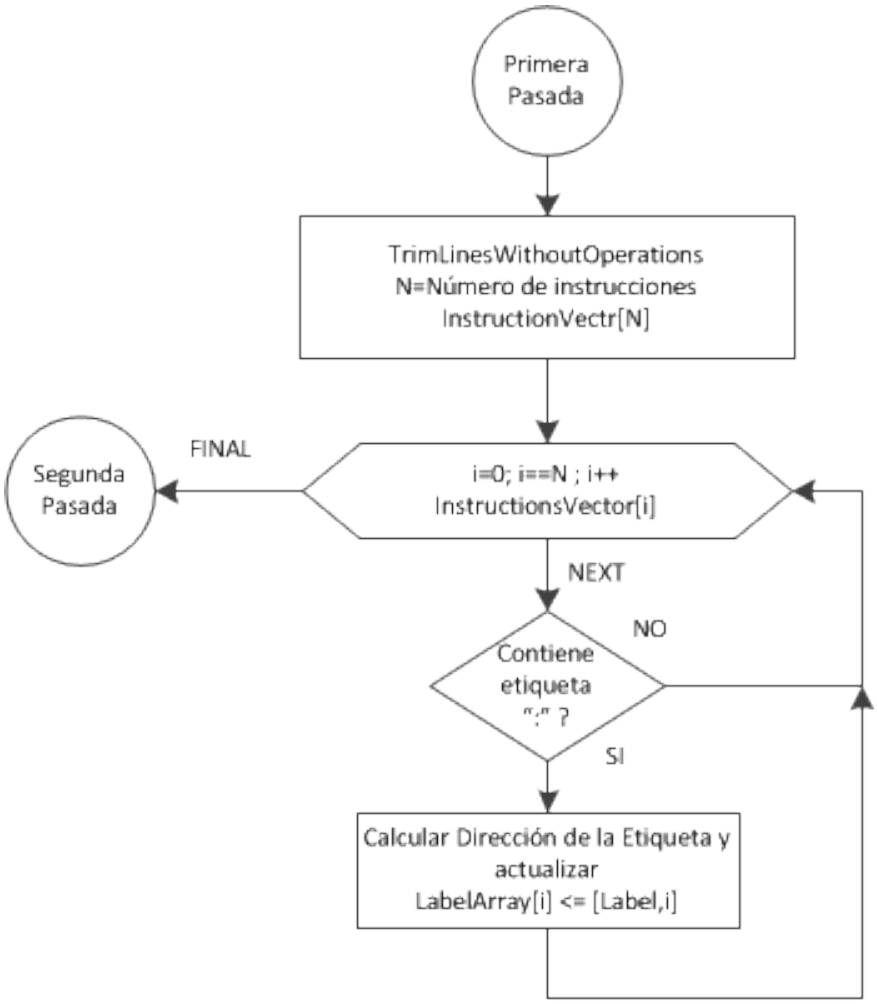
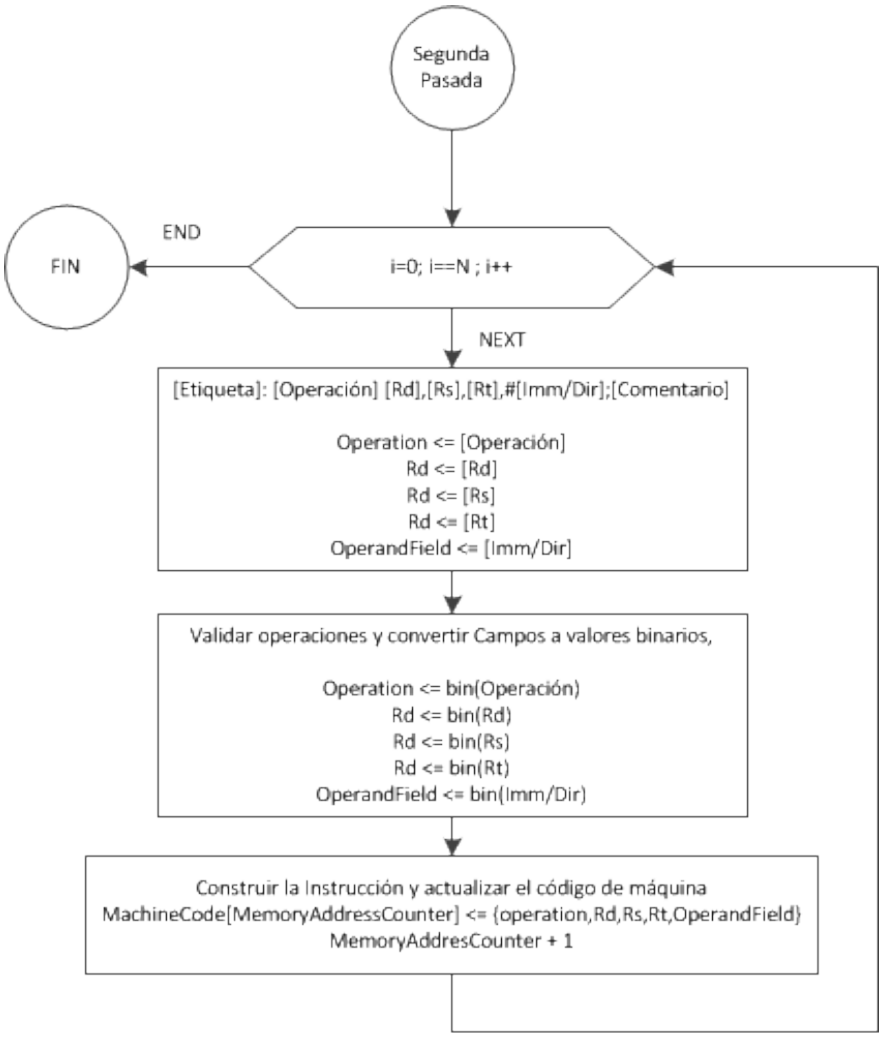


Figura 24. (Continuación)



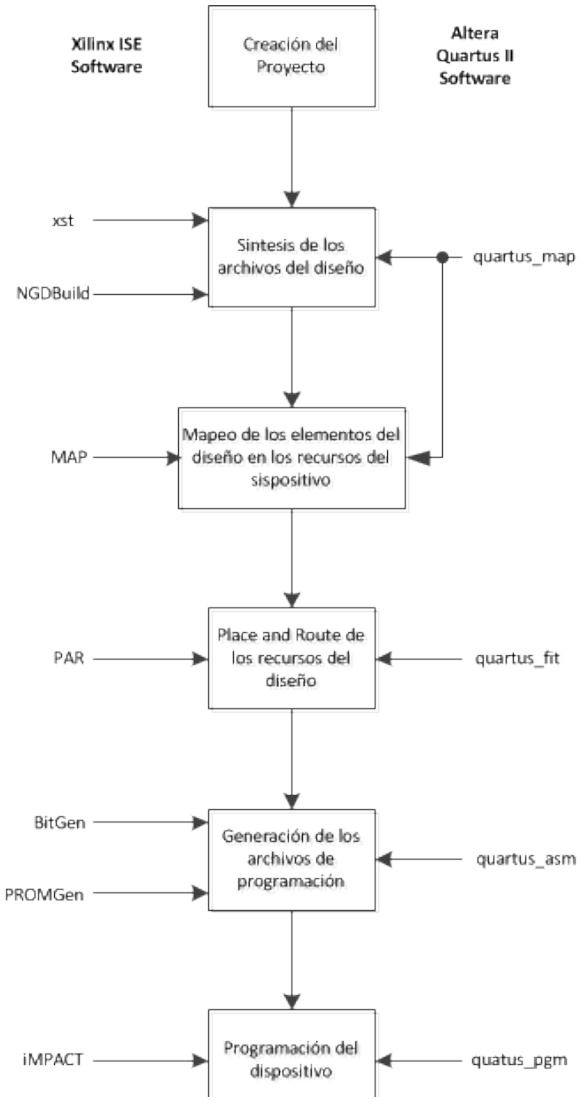
6.3.2. X-ISCKER *Programmer*

Esta herramienta permite programar el FPGA con el microprocesador de la arquitectura seleccionada. Se utilizan los programas ejecutables por línea de comandos que hacen parte del paquete de instalación de Altera Quartus II® y Xilinx ISE®, siguiendo el diagrama de flujo de diseño con FPGA señalado por cada fabricante. Es necesario tener instalados los IDE de acuerdo al fabricante del dispositivo que se va a utilizar.

La Figura 25 muestra el Diagrama de Flujo de diseño con FPGA. En el lado

izquierdo se muestran los programas del fabricante Xilinx y en el derecho los del fabricante Altera, estos programas ejecutables por líneas de comando llevan a cabo cada una de las etapas de diseño con FPGA.

Figura 25. Diagrama de flujo de implementación en FPGA Xilinx y Altera[30]



[30] ALTERA. AN 307: Altera Design Flow for Xilinx Users. 2009. p. 2

A continuación se describen los programas ejecutables por línea de comandos de Altera Quartus II® que se utilizaron en esta aplicación:

Tabla 13. Programas ejecutables por línea de comandos Altera

Ejecutable	Descripción	Requerimientos
"*Analysis and Synthesis*" quartus_map.exe	Integra los archivos de diseño, realiza una síntesis lógica y mapea ese resultado en los recursos del dispositivo seleccionado.	Archivo del proyecto (.qpf) y todos los archivos del diseño.
"*Fitter*" quartus_fit.exe	Ejecuta el "*Place and Route*" ajustando la lógica de un diseño en el dispositivo, selecciona las rutas de interconexión apropiadas, asigna pines y celdas lógicas.	"*Analysis and Synthesis*"
"*Assembler*" quartus_asm.exe	Genera un archivo imagen de la programación del dispositivo, en los formatos .pof, .sof	"*Fitter*"
"*Compiler Database* Interface" quartus_cdb.exe	Actualiza la inicialización de los bloques de memorias RAM y ROM	"*Analysis and Synthesis*" Archivo de inicialización de memoria MIF
quartus_pgm.exe	Programa los dispositivos de altera	.sof .cdf

Fuente: ALTERA. Quartus II Version 12.0. 2012. p. 803

En el caso de utilizar dispositivos Xilinx, la Tabla 14 muestras los programas ejecutables por línea de comandos que se utilizaron en esta aplicación:

Tabla 14. Programas ejecutables por línea de comandos Xilinx

Ejecutable	Descripción	Entrada	Salida
map.exe	Hace el mapeo del diseño lógico en la FPGA	NGD (*Native Generic Database*)	NCD (*Native Circuit Description*), Mapped
par.exe	Ejecuta el *Place and Route*	NCD, *Mapped*	NCD), Placed and Routed
trce.exe	Reporte de tiempos, y analizador de circuitos	NCD, Placed and Routed	TWR (*Timing report File*)
bitgen.exe	Genera el archivo de configuración del dispositivo.	NCD, *Placed and Routed*	BIT (*Binary File*)
iMPACT.exe	Programa el FPGA	BIT (*Binary File*)	

Fuente: XILINX. UG628: Command Line Tools User Guide. 2009. p. 33

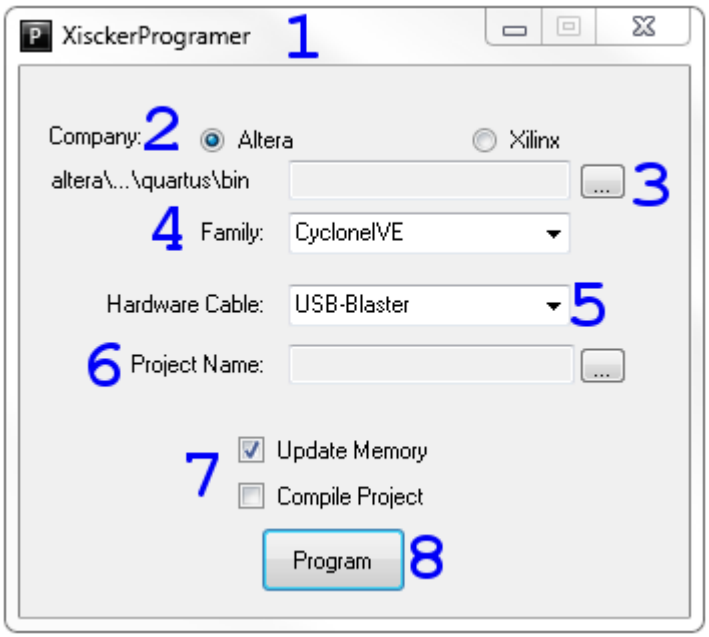
El X-ISCKER *Programmer* tiene dos modos de funcionamiento:

- *Compile Project*: Compila el proyecto seleccionado (*Synthesis, Assembly, Place and Route*) y programa el FPGA. Sólo es necesaria cuando el proyecto se compila por primera vez después de una modificación en el hardware. Es la opción más lenta.

- *Update Memory*: Actualiza únicamente los bloques relacionados a la memoria de instrucciones del procesador y programa el FPGA. Es la opción más rápida. Sólo disponible para dispositivos Altera.

La Figura 26 muestra la interfaz del X-ISCKER *Programmer*.

Figura 26. Interfaz X-ISCKER *Programmer*



A continuación se explica el funcionamiento de cada una de las partes de la interfaz en orden de aparición:

1. *Company*: Permite seleccionar entre dispositivos fabricados por Altera o Xilinx.

2. altera\...\quartus\bin: Se activa al seleccionar el fabricante Altera. Muestra la ruta en la que se encuentran los ejecutables. Se debe seleccionar la carpeta adecuada para el funcionamiento de la aplicación.

    a. Xilinx\...\ISE\bin\nt: Se activa al seleccionar el fabricante Xilinx. Muestra la ruta en la que se encuentran los ejecutables. Se debe seleccionar la carpeta adecuada para el funcionamiento de la aplicación.

3. *Family*: Especifica la familia del dispositivo que se desea programar.

4. *Hardware* Cable: Para dispositivos Altera. Especifica el protocólo de

configuración. (USB-Blaster, Ethernet-Blaster...)

5. *Project* Name: Se activa al seleccionar el fabricante Altera. Especifica la dirección en la que se encuentra el archivo del proyecto que se quiere programar.

    a. *Top Module*: Se activa al seleccionar el fabricante Xilinx. Especifica la dirección en la que se encuentra el archivo módulo principal del proyecto que se quiere programar.

6. *Update Memory*: Selecciona el modo de funcionamiento “Actualización de memoria”.

7. *Compile Project*: Selecciona el modo de funcionamiento que compila todo el proyecto.

6.3.3. X-ISCKER *Observer*

El X-ISCKER *Observer* es una interfaz hardware-software que permite la visualización de los registros más relevantes del microprocesador programado en el FPGA. La

Tabla *15* contiene los registros que se pueden observar de cada arquitectura, estos registros conforman el estado del procesador y pueden ser observados paso a paso a diferentes frecuencias. Los valores de cada registro se muestran en etiquetas sobrepuestas en los diagramas de la Figura 27. RISCKER *Observer* y la Figura 28. CISCKER *Observer*.

El componente en hardware de la interfaz consta de una variación en la descripción de cada microprocesador que incluye un módulo de comunicación serial por protocolo UART (*Universal Asynchronous Receiver Transmitter*) conectado a los buses de los registros que se desean visualizar. Para enviar estos datos a un computador personal se puede utilizar un puerto RS-232 o un puerto COM virtual (VCP) a través de módulos que soportan este servicio sobre protocolo USB u otros, ya que se especifica la ubicación de las conexiones Tx (Transmisor) y Rx (Receptor). Para el desarrollo del proyecto se utilizo un módulo “FTDI *Basic*

Breakout" de Sparkfun[31].

Tabla 15. Señales transmitidas RISCKER y CISCKER

RISCKER	**CISCKER**
Program Counter (PC)	Program Counter (PC)
Instruction (*Opcode* Operands Function)	Memory Address Register (MAR)
Arithmetic-Logic Unit Register (ALU Result)	Operation Code (*Opcode*)
Memory Data (MemData)	Memory Data (MM)
Write Memory Data (WriteData)	Arithmetic-Logic Unit Result (ALU Result)
Register General purpose 0 (RG 0)	Accumulator D (ACCD)
Register General purpose 1 (RG 1)	Index Register (IX)
Stack Pointer (SP)	Condition Code Register (CCR)
Output Port	Stack Pointer (SP)
Control Register (RC)	Output Port (I/O Unit)
Temporizador 1 y 2 (TMR1, TMR2)	Temporizador 1 y 2 (TMR1, TMR2)

La plataforma soporta distintas configuraciones de frecuencia de transmisión y la selección del puerto COM a utilizar. El registro de las señales adquiridas puede almacenarse en un archivo *.csv para su visualización y análisis.

[31] SPARKFUN ELECTRONICS. FTDI Basic Breakout. http://www.sparkfun.com/products/8772

Figura 27. RISCKER *Observer*

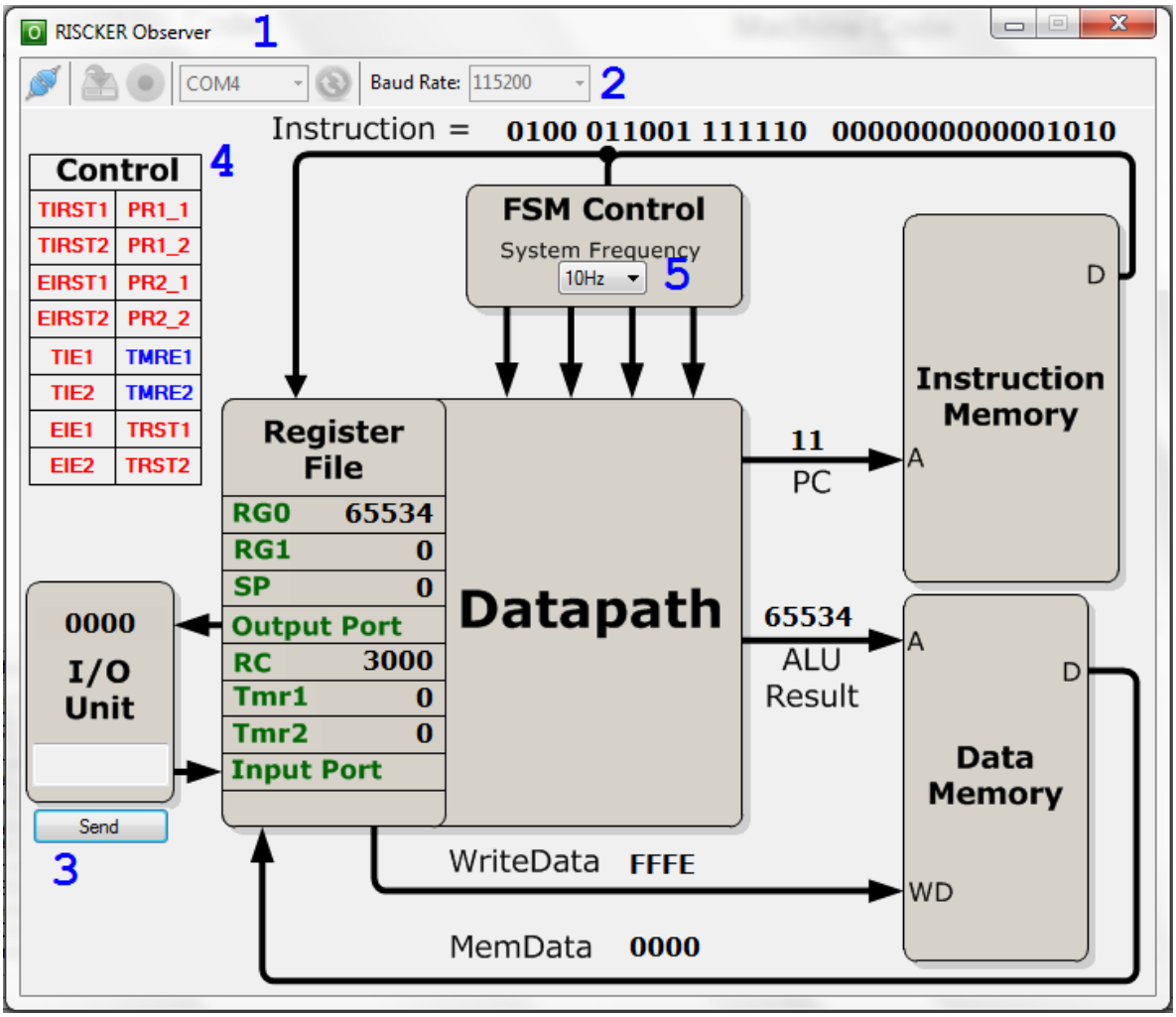
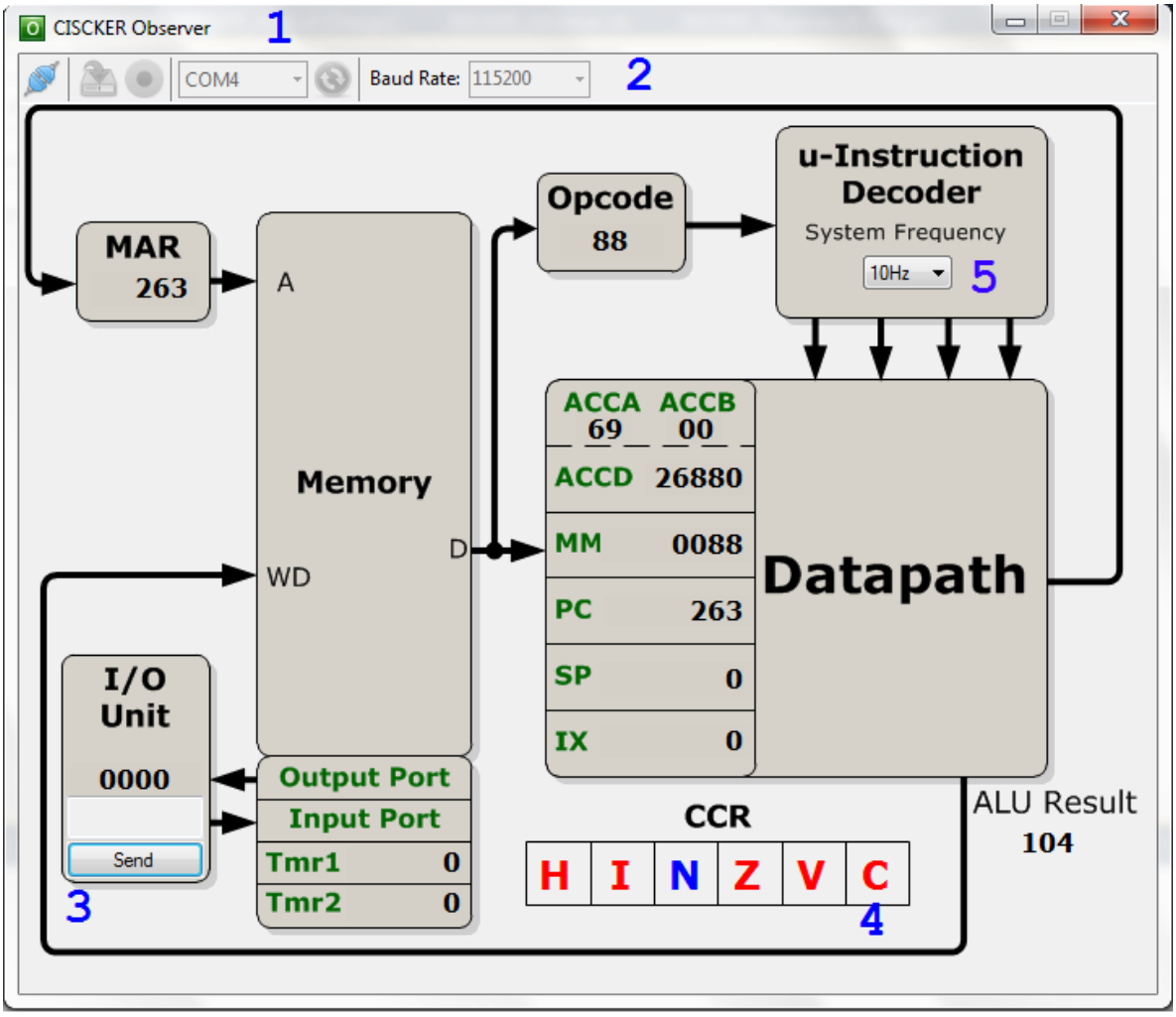


Figura 28. CISCKER *Observer*



La comunicación entre el software y el FPGA es bidireccional ya que al usuario se le permite enviar, por el mismo medio, tres bytes de información: Dos bytes que se conectan al puerto de entrada del microprocesador y un byte que configura un circuito divisor de reloj que controla la frecuencia de funcionamiento del diseño dentro de un rango preconfigurado.

A continuación se explican las distintas funciones presentes en la GUI del *Observer* RISCKER y CISCKER de acuerdo a la enumeración correspondiente la Figura 27 y la Figura 28.

1. Barra de título: Muestra el nombre del programa y la arquitectura actual.

2. Barra de Herramientas: Para ambas configuraciones es la misma. En esta barra se hace la configuración del puerto VCP y las opciones de conexión y registro de sesión. La barra de herramientas contiene los siguientes ítems:

    a. Conectar/Desconectar: Botón de conexión y desconexión del puerto COM. El icono cambia indicando el estado del puerto. Al iniciar la comunicación se deshabilitan las opciones de configuración del puerto.

    b. Definir archivo de registro: Botón para definir la ubicación del archivo de registro de sesión. Al momento de seleccionar la ubicación, el programa creará e inicializará una fila de un documento *.csv.

    c. Grabar/Pausa: Botón para activar o desactivar el registro de la sesión que va a ser o está siendo ejecutada. El icono cambia indicando la opción de pausar el registro actual o reanudarlo.

    d. Listado de puertos COM: Muestra una lista de puertos COM disponibles en el sistema para seleccionar como medio de comunicación con la plataforma.

    e. Actualizar puertos COM: Botón para actualizar el listado de puertos COM en caso de conectar el dispositivo después de la inicialización del programa.

    f. Tasa de Baudios: Muestra una lista de selección de tasa de baudios predeterminados. Permite edición para tasas de baudios personalizadas.

3. Puerto de Entrada: XISCKER *Observer* permite controlar e interactuar con el microprocesador implementado. Este campo envía la palabra de 16 bits escrita en hexadecimal (FFFFh), binario (1010101010101010b), octal (177777o) o decimal (65535) a un registro conectado directamente al puerto de entrada del microprocesador implementado.

4. CCR (*Condition Code Register*) / Registro de Control: Las casillas muestran el registro de códigos de condición (CCR) para el microprocesador CISCKER o el registro de control (RC) en el microprocesador RISCKER. El color rojo corresponde al valor cero y el azul al valor uno.

5. Configuración de frecuencia: Esta lista muestra distintas frecuencias a las cuales se puede operar el microprocesador implementado y la opción de detenerlo. Las frecuencias que se despliegan en la lista aseguran un rango de observabilidad y de comunicación fiable entre el FPGA y el computador personal

7. RESULTADOS Y DISCUCIÓN

Se construyó la plataforma X-ISCKER conformada en parte por un desarrollo en Verilog HDL para FPGAs, con bases conceptuales en el área de arquitectura de computadores y diseño avanzado de hardware. Esta implementación en FPGA se complementa con el diseño y programación de un software IDE mediante la herramienta Visual Studio 2010 en un proyecto de Visual Basic que utiliza los fundamentos de los procesos de ensamble y análisis léxico. Por último, se generó un conjunto de documentos (Guía de usuario y Hojas de datos) con información detallada de la arquitectura y funcionamiento de cada módulo de la plataforma.

Se diseñó e implementó, en dispositivos de lógica programable, dos procesadores, uno de arquitectura RISC (RISCKER) basado en MIPS, y otro de arquitectura CISC (CISCKER) basado en las arquitecturas HC08 y HC11, los cuales poseen las características descritas en la

Tabla 16 resaltando las diferencias más importantes entre las dos.

Tabla 16. Comparación RISCKER-CISCKER

Característica	RISCKER	CISCKER
# de Instrucciones	30	244
# de Registros	64	4
ALU	16 bits	16bits
Arquitectura de memoria	Harvard	Von Neumann
Tamaño de las instrucciones	Fijo: 16-bits	Variable: 8 a 24-bits
Multiplicación y División	Unidad de hardware dedicada	Iteraciones
Operaciones “Shift”	Logica, Aritmetica y Rotación de 1 a 16 bits por instrucción.	Logica, Aritmetica y Rotación de 1 bit por instrucción.
Modos de Direccionamiento	Inmediato,Directo, Indirecto,Desplazado	INH,REL,IMM,DIR,EXT,IX,IX+
Tamaño máximo de memoria	65536x32bits 65536x16bits	65536x16bits
# de condiciones de salto	2	14
Temporizadores	2 de 16 bits	2 de 16 bits
Interrupciones	2 por Temporizador 2 Externas	2 por Temporizador 2 Externas
Unidad de Control	HCU	MCU

Arquitectura similar	MIPS	HC08 y HC11 Freescale

La Tabla 17 establece una comparación en el consumo de recursos que requiere la implementación de cada uno de los procesadores. El parámetro de medición es el uso de Elementos Lógicos (LE) en una FPGA *Cyclone* IV de Altera.

Tabla 17. Consumo de recursos de cada procesador

Plataforma	Recursos	Utilizados	Disponibles	Porcentaje
CISCKER	**Total LE**	1,436	22,320	6%
	Funciones combinacionales	1,388		6%
	Registros	362		2%
RISCKER	**Total LE**	2,750		12%
	Funciones combinacionales	2,428		11%
	Registros	1,089		5%
CISCKER Observer	**Total LE**	2,173		10%
	Funciones combinacionales	2,062		9%
	Registros	832		4%
RISCKER Observer	**Total LE**	3,436		15%
	Funciones combinacionales	3,140		14%
	Registros	1,643		7%

Se desarrolló una interfaz en software que permite la programación y depuración de los procesadores descritos. Esta interfaz con tres aplicaciones: ASM, *Observer* y *Programmer*, conforma el entorno de desarrollo X-ISCKER IDE. Para hacer la conexión entre el software y hardware se realizó una variación de los proyectos en Verilog HDL que incluyen un módulo de comunicación bidireccional de protocolo UART.

Se generó documentación adjunta al proyecto en forma de Hojas de Datos y Guía de Usuario, con información necesaria para la puesta en funcionamiento de toda la plataforma y conocimiento de la organización funcional de los procesadores. Además se deja a disposición del usuario un sitio *web* para dar soporte y desarrollo de nuevas ideas.

En comparación con otros proyectos de implementación de procesadores embebidos en FPGA enfocados a trabajar los fundamentos básicos de la arquitectura de computadores, la plataforma X-ISCKER ofrece los dos enfoques

que se reflejan en las arquitecturas RISC y CISC en vez de limitar las opciones de diseño a una sola arquitectura. Esto tiene el objetivo de generar un criterio de selección de las características más convenientes para una aplicación específica y fomentar el diseño de nuevas arquitecturas, ya sea modificando las que son provistas o mediante la creación de modelos alternativos.

Todos los desarrollos futuros de este proyecto tienen un punto de encuentro en el sitio web de desarrollo colectivo que funcionará como una "*Wiki*", de manera que sean los mismos visitantes del sitio quienes participen en su desarrollo. La administración del sitio *web* y la adición de nuevo contenido estará a cargo del Semillero ADT (*Advanced Digital Technologies*) de la Universidad Pontificia Bolivariana seccional Bucaramanga.

Esta modalidad de trabajo es común en el desarrollo de software con la implementación de licencias de código abierto y el uso de repositorios y control de versiones. También es ahora, en descripción de hardware para FPGAs, una tendencia de bastante interés para los desarrolladores de sistemas embebidos con dispositivos de lógica programable. Como ejemplo, existe el proyecto OpenRISC mencionado anteriormente, y las plataformas COFFEE y MOLEN, entre otros.

Durante la ejecución de este proyecto se planteó la inclusión de derivaciones X-ISCKER, como alternativa a microcontroladores comerciales, en proyectos que se están llevando a cabo en la Corporación para la Investigación de la Corrosión (CIC) y en la Fundación Cardiovascular de Colombia (FCV). Estas derivaciones estarían enfocadas a aplicaciones específicas por lo cual sería necesaria la modificación de Hardware y creación de un ISA enfocado en las tareas de lectura/escritura de memorias Flash de protocolo ONFI para el caso de la CIC, y la interfaz con controladores gráficos en la FCV.

8. CONCLUSIONES Y RECOMENDACIONES

La plataforma X-ISCKER es una herramienta de aplicación en el área de procesadores embebidos en FPGA que se basa en los conceptos tratados en los cursos de arquitectura de computadores. Esta herramienta ilustra con detalle los principios de funcionamiento de las dos arquitecturas básicas, CISC y RISC, dejando claras las diferencias, ventajas y desventajas de cada arquitectura, sobre las cuales se basan los diseños de las arquitecturas actuales. Esto permite un nivel de generalización de los fundamentos de funcionamiento e implementación, aplicables a procesadores de distintas gamas.

La arquitectura CISCKER ofrece un amplio set de instrucciones con ciclos de ejecución variable: instrucciones de un solo ciclo de máquina (INX, INS), con estados de lectura y escritura de memoria implícito (NEG, DEC), de múltiples iteraciones (MUL, DIV) e instrucciones complejas que llevan a cabo varias tareas (SWI). En contraste la arquitectura RISCKER ofrece un set de instrucciones reducido basado en operaciones sencillas de hasta tres ciclos de ejecución, con solo dos instrucciones de acceso a memoria. Aunque una instrucción RISCKER generalmente se ejecuta con mayor rapidez que una instrucción CISCKER, las tareas que cumple una instrucción CISCKER pueden tomar un número mayor de instrucciones RISCKER. A su vez, el uso inadecuado y reducido de las instrucciones CISCKER de mayor complejidad pueden afectar el rendimiento del programa o dejar inutilizado el hardware adicional que existe para cumplirlas.

Este proyecto no contempla entre sus objetivos las pruebas de rendimiento que pueden medir las capacidades de ejecución de estos diseños, como son el número máximo de instrucciones por segundo o velocidad de respuesta a una interrupción, entre otras. Se documentaron las especificaciones técnicas de cada arquitectura de manera que el rendimiento depende de los algoritmos que se quieran implementar. Un proyecto futuro podría proponer la medición de rendimiento como parte de una estrategia de selección de una arquitectura para llevar a cabo una tarea determinada.

Se recomienda la utilización de esta plataforma en la demostración de las bases conceptuales que permiten el desarrollo de otras aplicaciones en el área de arquitectura de computadores y ciencias de la computación, como lo son la implementación de etapas de pipeline a las arquitecturas, diseño de una unidad de punto flotante, creación y control de memoria caché, aplicaciones multi-núcleo y arquitecturas VLIW (*Very Long Instruction Word*).

Por otro lado, la plataforma X-ISCKER brinda el entorno necesario para el desarrollo de un Compilador de lenguaje C de los lenguajes ensamblador de cada arquitectura para lograr una mayor competencia en la elaboración de software embebido y la implementación de librerías existentes. Este desarrollo ubicaría las arquitecturas X-ISCKER en un nivel más cercano al de microcontroladores comerciales haciéndolas más atractivas para aplicaciones posteriores.

Adicionalmente, la plataforma X-ISCKER es un inicio para la elaboración de procesadores reconfigurables en campo, un tema de investigación de la ciencia de computación y arquitectura de computadores, cuyo desarrollo es joven y brinda muchas oportunidades de investigación. Estos sistemas hacen uso de la propiedad programable en campo de los FPGA o dispositivos similares, para adecuar el hardware e instrucciones de un procesador embebido a las necesidades del entorno. Algunos temas afines a este área de investigación son los procesadores de Set de Instrucciones Dinámico (DISC) y las FPGA con hardware parcialmente configurable.

BIBLIOGRAFÍA

ARTICULOS CIENTÍFICOS

ELMEDANY, WAEL M y ALKOOHEJI, KHALID A. Design and Implementation of a 32bit RISC Processor on Xilinx FPGA. Egypt : Fayoum University. Department of Communications and Electrical Engineering, 2008.

FREEMAN, Michael. A Minimal CISC Processor Architecture for Field Programmable Gate Arrays. York, UK : IEEE. 2006.

JOSEPH, N.; SABARINATH, S.; SANKARAPANDIAMMAL, K. FPGA Based Implementation of High Performance Architectural Level Low Power 32-bit RISC Core. s.l. : IEEE International Conference, 2009. Vol. Advances in Recent Technologies in Communication and Computing.

NAKANO, K.; Ito, Y. Processor, Assembler, and Compiler Design Education Using an FPGA. s.l. : 14th IEEE International Conference, 2008.

ROMERO-TRONCOSO, R. de J. 8-bit CISC Microprocessor Core for Teaching Applications in the Digital Systems Laboratory. 2006. s.l. : IEEE International Conference. Vol. Reconfigurable Computing and FPGA's. 2004.

WIRTHLIN, Michael J.; HUTCHINGS, Brand L. DISC: The dynamic instruction architecture. Brigham Young University, Provo, Utah 84032.

REFERENCIAS BIBLIOGRÁFICAS

HARRIS, David Money; HARRIS, Sarah L. Digital design and computer arquitecture. San Francisco: Morgan Kaufmann Publishers, Elsevier, 2007. 592 p.

HENNESSY, John L.; PATTERSON, David A. Computer Architecture. A Quantitative Approach. 3 ed. San Francisco: Morgan Kaufmann Publishers, Elsevier, 2007. 1136 p.

MORRIS MANO, M. Fundamentos De Diseño Lógico Y De Computadoras. 3 ed. México: Prentice Hall, 2005. 536 p.

PATTERSON, David A.; HENNESSY, John L. Computer Organization and Design. The Hardware/Software Interface. 3 ed. San Francisco: Morgan Kaufmann Publishers, Elsevier, 2005. 914 p.

SHIVA ,Sajjan G. Computer Organization, Design, and Architecture. 4 ed. United States of America: CRC Press, 2008. 784 p.

STALLINGS, William. Computer Organization and Architecture. 6a Edition. United States of America: Pearson Education, 2003. 750 p.

FLYNN, Michael J. Computer Architecture: Pipelined and Parallel Processor Design. United States of America: Jones & Bartlett Publishers Inc, 1995. 788 p.

DANDAMUDI Sivarama P., Fundamentals of computer organization and design. United States of America: Springer, 2002. 1092 p.

ABD-EL-BARR, Mostafa. Fundamentals of computer organization and architecture. United States of America: Wiley-Interscience, 2005. 288 p.

CHU, Pong P. FPGA Prototyping by Verilog examples: Xilinx Spartan 3 Version. United States of America: Wiley, 2008. 518 p.

LIPOVSKI, G. Jack. Introduction to Microcontrollers: Architecture, Programming, and Interfacing of the Motorola 68Hc12. United States of America: Academic Press, 1999. 488 p.

NURMI, Jari. Processor Desing. System-on-Chip Computing for ASICs and FPGAs. United States of America: Springer, 2007. 548 p.

DOCUMENTOS WEB

ALTERA. Embedded Processors. [Artículo en Internet]. <http://www.altera.com/products/ip/processors/ipm-index.jsp> [Consulta: 1 de Julio de 2012]

KHOLODOV, Igor. CIS-77 Introduction to Computer Systems. [Artículo en internet]. <http://www.c-jump.com/CIS77/CIS77syllabus.htm> [Consulta: 1 Julio de 2012]

SCHMALZ. Organization of Computer Systems: § 3: Computer Arithmetic. [Artículo de internet]. <http://www.cise.ufl.edu/~mssz/CompOrg/CDA-arith.html>. [Consulta: 1 Julio 2012]

TEN EYCK, James. CMSC 415 & MSCS 521 Computer Architecture. [Articulo en internet]. <http://www.academic.marist.edu/~jzbv/architecture/>. [Consultado 1 Julio 2012]

WANG, Ruye. Multiplication and Division. [Artículo en internet]. <http://fourier.eng.hmc.edu/e85/lectures/arithmetic_html/node8.html>. [Consultado 1 Julio de 2012]

BIBLIOGRAFÍA COMPLEMENTARIA

BRITTON, Robert L. MIPS Assembly Language Programming. California: Pearson, Prentice Hall. 2004. 168 p.

COMER, Douglas E. Essentials of Computer Architecture. New Jersy: Pearson Prentice Hall, 2005. 400 p.

DANDAMUDI, Sivarama P. Guide to RISC Processors for Programmers and Engineers. United States of America: Springer, 2005. 404 p.

FOOK LEE, Weng. VLIW Microprocessor Hardware Design: On ASIC and FPGA. United States of America: McGraw-Hill, 2008. 219 p.

FREESCALE SEMICONDUCTOR. M68HC16 Family CPU16 Reference Manual. 1997.

FREESCALE SEMICONDUCTOR. S12CPUV2 Reference Manual. Freescale Semiconductor Inc. 2006.

LOUDEN, KENNETH C. Construcción de compiladores principios y práctica. Thomson Editores, 2004

MAZIDI, Ali Muhammad.; MAZIDI, Janice Gillispie. The 80x86 IBM PC and Compatible Computers. Volumes I & II. Cuarta Edición. Pearson Education, Inc. United States of America. 2003.

McFARLAND, Grant. Microprocessor Design. A Practical Guide From Design Planning to Manufacturing. McGraw-Hill. United States of America. 2006.

MIPS TECHNOLOGIES INC. MIPS32 Architecture for programmers. Volumes I,II,III. Mips Technologies Inc. Mountain View. 2005.

MOTOROLA SEMICONDUCTOR PRODUCTS INC. M6800 Application Manual. MOTOROLA INC. 1975.

MOTOROLA SEMICONDUCTOR PRODUCTS INC. M6809-MC6809E Microprocessor Programming Manual. MOTOROLA INC. 1983.

SWEETMAN, Dominic. See MIPS Run. Morgan and Kufmann. San Francisco. 1999.

XILINX. Spartan-3A FPGA Family: Data Sheet. 2010

ALTERA. Cyclone IV Device Handbook. Volumen I, II, III. 2011

FREESCALE SEMICONDUCTOR. CPU08 Central Processor Unit. 2006.

ALTERA. AN 307: Altera Design Flow for Xilinx Users. 2009.

ALTERA. Quartus II Version 12.0. 2012.

XILINX. UG628: Command Line Tools User Guide. 2009.

ANEXO A

GUÍA DE USUARIO XISCKER

2. Introducción

Esta guía de usuario hace parte del desarrollo de una “Plataforma para la Emulación y Reconfiguración de Arquitecturas RISC y CISC” en FPGA (Field-Programmable Gate Arrays) como herramienta académica clave en el área de arquitectura y organización de computadores, llamada X-ISCKER por sus siglas en inglés, “Reduced/Complex Instruction Set Computing Key Educational Resource”; elaborada como tésis de pregrado en la Universidad Pontificia Bolivariana Seccional (UPB) Bucaramanga por Alfredo Gualdrón y Jose Pablo Pinilla.

La plataforma X-ISCKER nace de la necesidad de una herramienta de apoyo en la enseñanza de arquitectura y organización de computadores, que ayude a la comprención de la materia y al mismo tiempo sea implementada en el desarrollo de aplicaciones tangibles en campos como la robótica, el control digital, instrumentación, comunicaciones, entre otros.

La plataforma X-ISCKER está compuesta por la descripción de hardware en lenguaje Verilog de dos microprocesadores, una interfáz de comunicación UART y un software IDE (Integrated Development Environment) que cumplen tres funciones principales: Programación, Emulación y Reconfiguración, con el fin de facilitar el desarrollo de proyectos basados en la tecnología de procesadores embebidos en FPGA, conformando una metodología de diseño que integra el desarrollo de software, la aplicación de una herramienta de depuración y el diseño avanzado de hardware.

3. Objetivos

- Utilizar la plataforma X-ISCKER IDE con el fin de observar el comportamiento de un procesador de arquitectura CISC y uno RISC.
- Utilizar la plataforma como un sistema de procesamiento independiente para aplicaciones en el campo de la electrónica digital.
- Reconfigurar la plataforma para implementar instrucciones y funciones nuevas a cualquiera de las dos arquitecturas.

4. Recursos Disponibles

Plataforma para la Emulación y Reconfiguración de Arquitecturas RISC y CISC: Libro principal del proyecto. Contiene la descripción de los objetivos, el soporte teórico de cada arqutiectura y la metodología de diseño.

Hojas de Datos: Contiene información detallada de la estructura y funcionamiento de los procesadores RISCKER y CISCKER.

Wiki: Sitio web diseñado para el soporte y continuo desarrollo de la herramienta. Contiene toda la información relacionada con el proyecto.

5. Arquitectura de los Microprocesadores

El siguientes es un cuadro comparativo de los dos microprocesadores disponibles para implementar en la plataforma.

Tabla 1. Comparación de las arquitecturas RISCKER y CISCKER

Característica	RISCKER	CISCKER
# de Instrucciones	30	244
# de Registros	64	4
ALU	16 bits	16bits
Arquitectura de memoria	Harvard	Von Neumann
Tamaño de las instrucciones	Fijo: 16-bits	Variable: 8 a 24-bits
Multiplicación y División	Unidad de hardware dedicada	Iteraciones
Operaciones “Shift”	Logica, Aritmetica y Rotación de 1 a 16 bits por instrucción.	Logica, Aritmetica y Rotación de 1 bit por instrucción.
Modos de Direccionamiento	Inmediato,Directo, Indirecto,Desplazado	INH,REL,IMM,DIR,EXT,IX,IX+
Tamaño máximo de memoria	65536x32bits 65536x16bits	65536x16bits
# de condiciones de salto	2	14
Temporizadores	2 de 16 bits	2 de 16 bits
Interrupciones	2 por Temporizador 2 Externas	2 por Temporizador 2 Externas
Unidad de Control	HCU	MCU
Arquitectura similar	MIPS	HC08 y HC11 Freescale

En las hojas de datos se presentan el set de instrucciones, diagramas de bloques y diagramas esquemáticos del diseño digital de los módulos de cada uno de los microprocesadores, contienen toda la información, de arquitectura y organización, necesaria para entender el funcionamiento general del hardware de la plataforma descrito en modulos Verilog.

6. Xilinx Spartan 3A & Altera DE0-Nano

El desarrollo de esta plataforma se llevo a cabo utilizando dos kits con FPGA: el primero es una tarjeta “Spartan 3A Starter Kit” que posee varios periféricos integrados y pocos pines GPIO (General Purpose Input/Ouput), ideal para utilizar dispositivos como: Display LCD, encoder, ADC, DAC, entre otros. La segunda

tarjeta es el kit de desarrollo DE0-Nano, la cual ofrece mayor flexibilidad gracias a la gran cantidad de GPIO disponibles.

La plataforma ha sido desarrollada para soportar ambos kits simplemente seleccionando el dispositivo con el cual se está trabajando, esto corresponde inicialmente a los dispositivos Xilinx Spartan 3A XC3S700A y Altera Cyclone IV EPC22F17C6, de manera que es posible que la plataforma no funcione en dispositivos de los mismos fabricantes pero de otras familias. Para lograr compatibilidad con otros dispositivos se recomienda primero, revisar las necesidades de hardware del sistema como capacidad del FPGA, memoria y bloques DSP y GPIO. Segundo, se debe modificar el software XISCKER Programmer o utilizar la herramienta de programación proporcionada por el fabricante del dispositivo.

7. Arquitectura de la Plataforma

La plataforma consta de los siguientes dispositivos y software:

- Spartan 3A Starter Kit o DE0-Nano Development board.
- Dispositivo de comunicación UART y Drivers.
- Módulos Verilog de los microprocesadores RISCKER y CISCKER.
- Módulos Verilog RISCKER y CISCKER con interfáz de comunicación UART-Observer.
- Software de Altera Quartus II 11.0 o superior ó Software de Xilinx ISE 13.0 o superior.
- Software X-ISCKER IDE (Integrated Development Environment)
- PC Windows con conexión USB y .NET Frameworks 4.0 o superior.

7.1. XISCKER IDE

Para instalar el software X-ISCKER IDE se debe descargar la última versión disponible del sitio: semilleroadt.upbbga.edu.co/XISCKER, ejecutar la aplicación y seguir los pasos del software de instalación.

Durante la instalación se crea la carpeta C:/XISCKER en la cual se almacena la dirección del último espacio de trabajo utilizado. Al ejecutar el programa por

primera vez aparecerá un dialogo para crear la carpeta que contendrá toda la información de la última sesión de trabajo: como son la arquitectura seleccionada, el código assembler, el archivo de inicialización de memoria y la configuración de programación. A esta carpeta se le llama *Workspace*.

Al crear un nuevo workspace se genera automáticamente una subcarpeta llamada “sys” con los siguiente archivos:

- AlteraCableItems.item: Contiene un listado de dispositivos de conexión para la programación del FPGA.
- AlteraFamilyItems.item: Contiene un listado con las posibles familias de FPGA Altera para utilizar en el XISCKER Programmer.
- LastAsmSession.bak: Contiene las direcciones del último archivo fuente en lenguaje ensamblador, el código de máquina, tipo de arquitectura y la dirección del espacio de trabajo.
- LastPrgmSession.bak: Contiene la informacion de cada uno de los campos elegidos en la última sesión del XISCKER programmer.
- XilinxFamilyItem.item: Contiene un listado con las posibles familias de FPGA Xilinx para utilizar en el XISCKER Programmer.

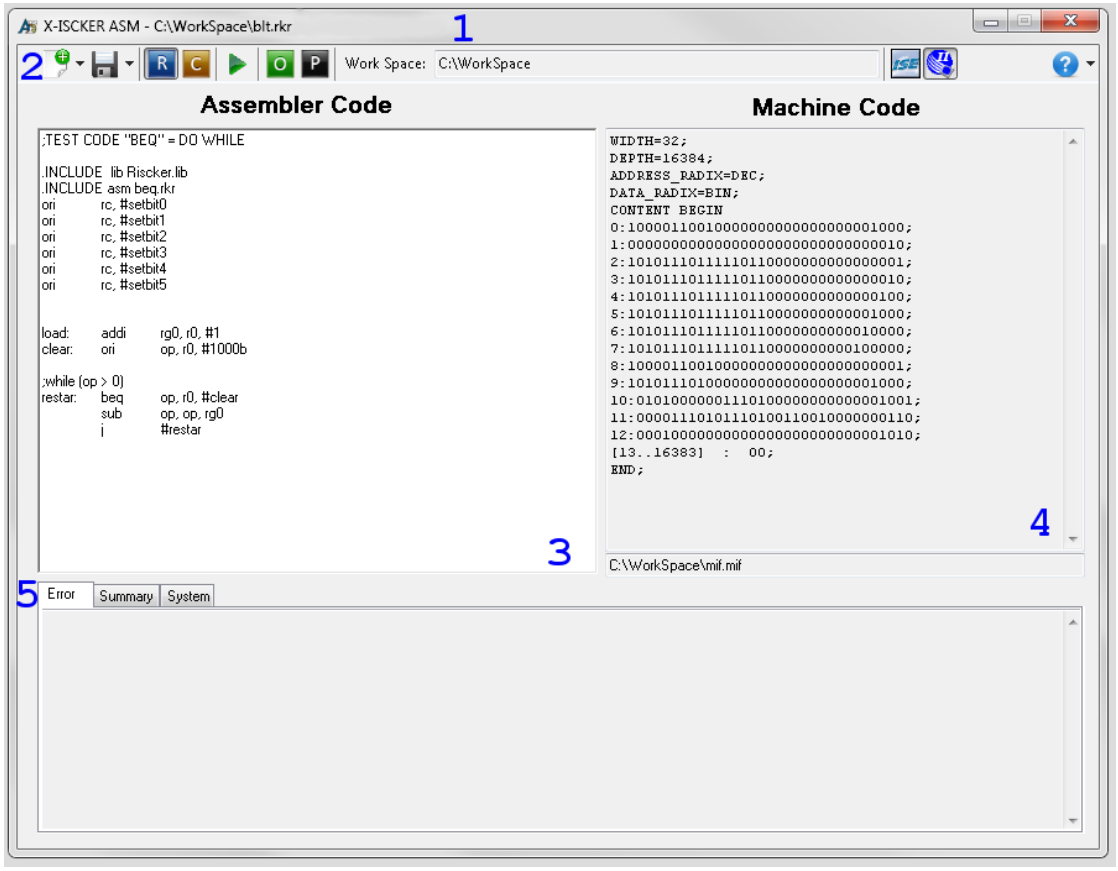
Los demás archivos correspondientes a cada proyecto que se trabaje en el XISCKER IDE son:

- *.rkr: Archivo de texto plano con código en lenguaje RISCKER ASM.
- *.ckr: Archivo de texto plano con código en lenguaje CISCKER ASM.
- *.mif , *.mem o *.dat: Archivos de inicialización de memoria. Contienen datos en binario o en hexadecimal que se escriben directamente a las memorias ROM o RAM del diseño en FPGA.

Cada uno de estos archivos puede tener una ubicación distinta, ya que pertenecen al proyecto actual del usuario y no directamente al software, sin embargo, se recomienda utilizar el espacio de trabajo actual.

A continuación se muestra una imagen con la GUI de la ventana principal del X-ISCKER IDE y una enumeración de cada una de las funciones que ofrece:

Figura 1. GUI de la ventana principal del software X-ISCKER IDE.



1. **Barra de Título:** Muestra la dirección del archivo “*.ckr” o “*.rkr” que se está modificando.
2. **Barra de Herramientas:**
   a.  Abrir: El botón abre un nuevo archivo en blanco para empezar a editar en la caja de código. Ofrece un menú con tres opciones.
      → RISCKER ASM: Abre un archivo de extención .rkr y selecciona la

arquitectura para el IDE como RISCKER.

→ CISCKER ASM: Abre un archivo de extención .ckr y selecciona la arquitectura para el IDE como CISCKER.

→ New blank file: Crea un archivo vacío, es la acción por defecto del botón.

b.  Guardar: El botón guarda el contenido de la caja de código y de memoria en archivos existentes o consulta la ubicación para crearlos.

→ Assembly File: Guarda el archivo contenido en la caja de código con extensión .ckr o .rkr según sea la arquitectura seleccionada.

→ Memory File: Guarda el contanido de la caja de memoria en un archivo de inicialización de memoria *.mif o *.mem.

c. Selector de Arquitectura  RISCKER  CISCKER: Selección del algoritmo de ensamble para la arquitectura de set de instrucciones RISCKER/CISCKER. Esta selección también modifica el ambiente de desarrollo XISCKER IDE de manera que el código se guarde en un archivo “*.rkr” o “*.ckr” y para identificar la interfaz con la que se va a lanzar el XISCKER Observer (detallado en el numeral 3.3).

d.  Ejecutar ensamblador: Inicia la interpretación del programa en lenguaje de ensamblador que se encuentra en la caja de código (Assembler Code), genera el archivo de inicialización de memoria de instrucciones y lo muestra en la caja de memoria (Machine Code).

e.  Programmer: Lanza una ventana adicional correspondiente a la herramienta X-ISCKER Programmer (detallada en el numeral 3.2).

f.  Observer: Lanza una ventana adicional correspondiente a la herramienta X-ISCKER Observer (detallada en el numeral 3.3).

g. Workspace: Muestra la ruta del espacio de trabajo actual.

h.  Xilinx  Altera: Al elegir uno de los dos fabricantes se modifican los parametros del XISCKER Programmer y el formato en que se crea el archivo de memoria.

i.  Ayuda: Este botón lanza una ventana con información acerca del XISCKER IDE.

→ User Guide: Esta guía de usuario.

→ Wiki: Enlace de la wiki XISCKER, contiene los archivos y avances más recientes del proyecto.

→ About: Lanza la ventana con información acerca de la versión y desarrollo del software XISCKER IDE. Es la opción por defecto del botón.

3. **Caja de Código:** Caja de texto para editar el código en lenguaje ensamblador de la arquitectura seleccionada.
4. **Caja de memoria:** Caja de texto que muestra el archivo de inicialización de memoria generado por el proceso de ensamble. En la parte inferior se muestra la ruta del archivo que se esté mostrando.
5. **Pestañas de Procesos:** Pestañas que muestran información de los procesos de ensamble y configuración del dispositivo.
   a. Error: Muestra las líneas de código con errores generados durante el proceso de ensamble o la programación del dispositivo. El número de la línea se especifica en la pestaña Summary.
   b. Summary: Muestra un resumen de las diferentes etapas del proceso de ensamble. Asigna un número a cada línea de código para facilitar su depuración.
   c. System: Muestra información sobre los comandos para generar el archivo de configuración del dispositivo, los errores y las advertencias que estos generan.

7.2. XISCKER Programmer

Es la herramienta de XISCKER IDE que permite programar el FPGA con el microprocesador de la arquitectura seleccionada. Es necesario tener instalado el IDE de la tecnología del dispositivo que se va programar (Quartus-Altera, ISE-Xilinx). Ésta herramienta tiene dos modos de funcionamiento:

Update Memory: Actualiza únicamente los bloques relacionados a la memoria de instrucciones del procesador y programa el FPGA. Es la opción más rápida.

Compile Project: Compila el proyecto seleccionado (Synthesis, Assembly, Place and Route) y programa el FPGA. Sólo es necesaria cuando el proyecto se compila

por primera vez después de una modificación en el hardware. Es la opción más lenta.

A continuación se muestra una imagen de la ventana de diálogo del X-ISCKER Programmer y una enumeración de cada una de las funciones que ofrece:

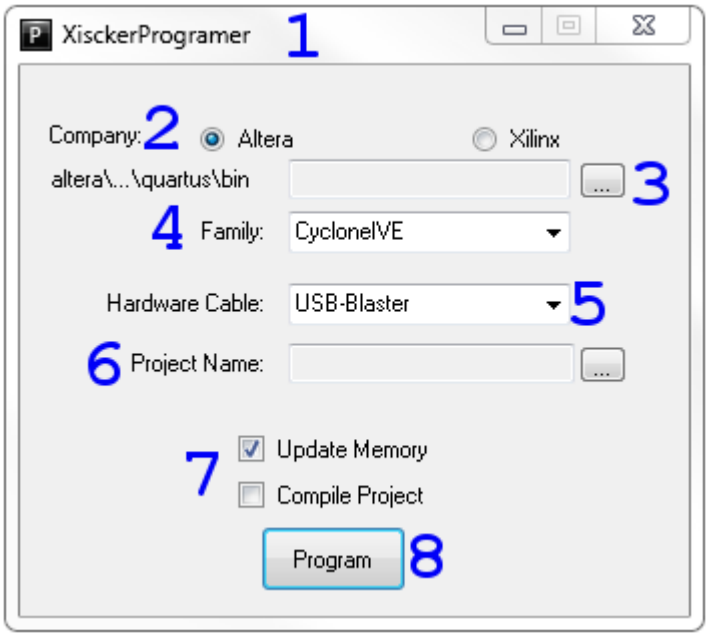


Figura 2. GUI de la herramienta de software X-ISCKER Programmer

1. **Barra de título:** Muestra el nombre del programa.
2. **Company:** Selección de la marca del dispositivo que desea configurar.
3. **Folder:** Selección de la carpeta “bin”, en la cual se encuentran los ejecutables necesarios para configurar el dispositivo (Carpeta de instalación de Quartus o Xilinx ISE, según la tarjeta de desarrollo).
4. **Family:** Selección de la familia o nombre del dispositivo.
5. **Hardware cable:** Selección del driver de configuración del dispositivo. Para Xilinx esta selección es automática.
6. **Project Name:**

   → Para implementación con dispositivos Altera: Selección de la ubicación del proyecto (Archivo de extención “.qpf”).

   → Para implementación con dispositivos Xilinx: Selección de la

ubicación del módulo principal (Archivo Verilog o VHDL) del proyecto.

7. **Update Memory/Compile Project:** Selección de los procesos que se desean llevar a cabo.
8. **Program:** Botón para iniciar la secuencia de programación.

7.3. XISCKER Observer

El XISCKER Observer es una interfáz que permite la visualización periódica de las señales más relevantes del microprocesador programado en el FPGA mediante etiquetas sobrepuestas a un diagrama de la arquitectura seleccionada, como lo muestran las figuras 3 y 4.
El registro de las señales adquiridas puede almacenarse en un archivo *.csv para una visualización y análisis posterior. A continuación se explican los distintos componentes presentes en la GUI del Observer RISCKER y CISCKER.

1. **Barra de título:** Muesta el nombre del programa y la arquitectura actual.
2. **Barra de Herramientas:** Para ambas configuraciones es la misma. En esta barra se hace la configuración del puerto VCP que desea utilizar para comunicarse con el microprocesador seleccionado y las opciones de conexión y registro de sesión. La barra de herramientas contiene los siguientes items:
   a.  Conectar  Desconectar: Botón de conexión y desconexión del puerto COM. El icono cambia indicando el estado del puerto. Al iniciar la communicación se deshabilitan las opciones de configuración del puerto.
   b.  Definir archivo de registro: Botón para definir la ubicación del archivo de registro de sesión. Al momento de seleccionar la ubicación, el programa creará e inicializará una fila de un documento *.csv.

Figura 3. Imagen de la GUI del Observer RISCKER.

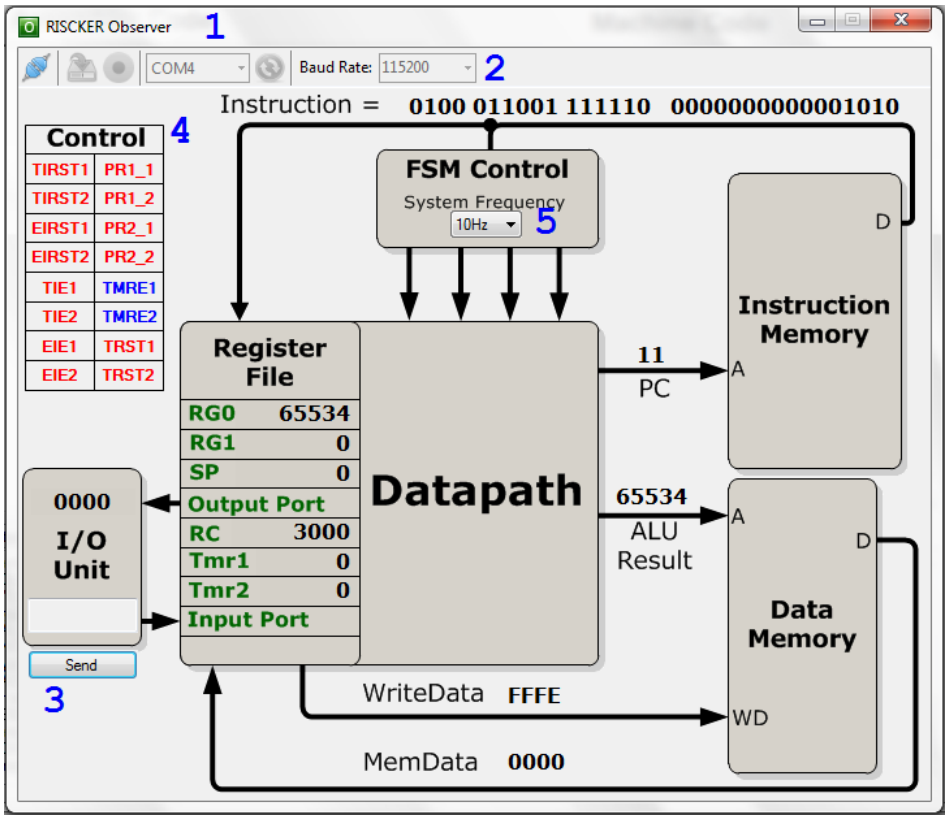
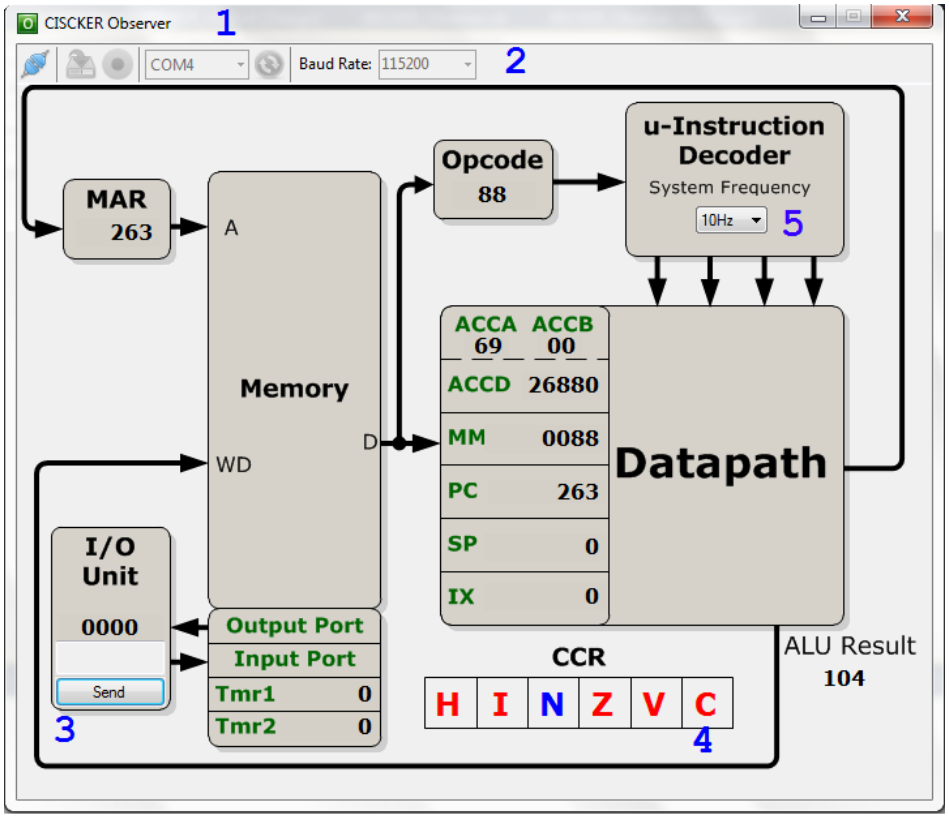

RISCKER Observer
1
COM4
Baud Rate: 115200
2
Instruction = 0100 011001 111110 0000000000001010
Control
4
TIRST1 PR1_1
TIRST2 PR1_2
EIRST1 PR2_1
EIRST2 PR2_2
TIE1 TMRE1
TIE2 TMRE2
EIE1 TRST1
EIE2 TRST2
FSM Control
System Frequency
10Hz
5
Register File
RG0 65534
RG1 0
SP 0
Output Port
RC 3000
Tmr1 0
Tmr2 0
Input Port
Datapath
11
PC
Instruction Memory
D
A
0000
I/O Unit
Send
3
65534
ALU Result
A
D
Data Memory
WD
WriteData FFFE
MemData 0000

Figura 4. Imagen de la GUI del Observer CISCKER.



c. Grabar Pausa: Botón para activar o desactivar el registro de la sesión que va a ser o está siendo ejecutada. El icono cambia indicando la opción de pausar el registro actual o reanudarlo.

d. Listado de puertos COM: Muestra una lista de puertos COM disponibles en su sistema para seleccionar como medio de comunicación con la plataforma.

e. Actualizar puertos COM: Botón para actualizar el listado de puertos COM en caso de conectar el dispositivo después de la inicialización del programa.

f. Tasa de Baudios: Muestra una lista de selección de tasa de baudios predeterminados. Permite edición para tasas de baudios personalizadas.

3. **Puerto de Entrada:** XISCKER Observer permite controlar e interactuar con el microprocesador implementado. Este campo envía la palabra de 16 bits escrita en hexadecimal (FFFFh), binario (1010101010101010b), octal (177777o) o decimal (65535) a un registro conectado directamente al puerto de entrada del microprocesador implementado.
4. **CCR (Condition Code Register) / Registro de Control:** Las casillas muestran el registo de códigos de condición (CCR) para el microprocesador CISCKER o el registro de control (RC) en el microprocesador RISCKER. El color rojo corresponde al valor zero y el azul al valor uno. En las hojas de datos de cada procesador encontrará más información sobre cada una de las señales que aparecen en las casillas.
5. **Configuración de frecuencia:** Esta lista muestra distintas frecuencias a las cuales se puede operar el microprocesador implementado y la opción de detenerlo. Las frecuencias que se despliegan en la lista aseguran un rango de observabilidad y de comunicación fiable entre el FPGA y el computador personal.

A continuación se muestra una tabla con las señales recibidas de cada microprocesador:

Tabla 2. Señales transmitidas por el hardware UART desde el FPGA para cada arquitectura.

RISCKER	**CISCKER**
Program Counter (PC)	Program Counter (PC)
Instruction (Opcode Operands Function)	Memory Address Register (MAR)
Arithmetic-Logic Unit Register (ALU Result)	Operation Code (Opcode)
Memory Data (MemData)	Memory Data (MM)
Write Memory Data (WriteData)	Arithmetic-Logic Unit Result (ALU Result)
Register General purpose 0 (RG 0)	Accumulator D (ACCD)
Register General purpose 1 (RG 1)	Index Register (IX)
Stack Pointer (SP)	Condition Code Register (CCR)
Output Port	Stack Pointer (SP)
Control Register (RC)	Output Port (I/O Unit)
Timers 1 y 2 (TMR1, TMR2)	Timers 1 y 2 (TMR1, TMR2)

7.4. Programación en XISCKER ASM

Para programar un procesador de la plataforma, una vez escritos los algoritmos en el lenguaje ensamblador apropiado para cada arquitectura, es necesario generar el archivo de inicialización de memoria (*.mif o *.mem) dando click en el botón ejecutar ensamblador (Play), o presionar la tecla F5. Una vez generado el código de máquina, éste debe ser guardado en la carpeta del proyecto en el que se encuentre instanciado el procesador y reemplazar el archivo de la misma extención. En caso que se quiera modificar la ubicación de este archivo, se debe actualizar ese cambio en el proyecto de hardware *.qpf o *.ise para que el módulo de memoria de programa reconozca la nueva ubicación del código de máquina.

Los recursos correspondientes a esta etapa son el software X-ISCKER IDE y las

hojas de datos de cada microprocesador, en las cuales se encuentra la información de la ISA (Instruction Set Architecture) completa desde el punto de vista del programador que le permite conocer todas las instrucciones y módulos internos para utilizar, como puertos, temporizadores e interrupciones.

7.4.1. Procedimiento

1. Ejecute la aplicación X-ISCKER IDE
2. Si es la primera vez que ejecuta el programa, seleccione una carpeta, en la cual se crearán los archivos relacionados con el proyecto que vaya a realizar (Workspace).
3. Seleccione la arquitectura con la que va trabajar
4. Escriba el código en lenguaje ensamblador
5. Haga click en  "Run" (F5), Para generar el archivo de inicialización de memoria.
6. Guarde el archivo de inicialización de memoria en la carpeta que contiene el proyecto del microprocesador (.qpf ó .ise).
7. Haga click en  "Programmer" (F7). En la nueva ventana:
   a. Seleccione la dirección de la carpeta bin, según sea Altera o Xilinx.
   b. Seleccione la Familia de dispositivo que desea configurar.
   c. Seleccione el Cable de la tarjeta correspondiente (Xilinx es Automático).
   d. Seleccione la dirección del proyecto donde está instanciado el procesador.
      → Para dispositivos Altera: Seleccione el archivo del proyecto (*.qpf).
      → Para dispositivos Xilinx: Seleccione el archivo del módulo principal del proyecto (Verilog o VHDL).
   e. Haga click en "Program"

Nota: Para dispositivos Altera asegurese de haber generado el archivo con extención ".cdf" desde la herramienta "Programmer" del software Quartus en la opción "Save" del menú "File".

   f. En la pestaña "System" aparecerá el resultado de los comandos

ejecutados y en la pestaña “Error” las líneas que indican algún error ocurrido en el proceso.

7.5. Emulación en XISCKER ASM

El objetivo de esta etapa es la observación del comportamiento de un microprocesador, es decir, el usuario programa una rutina utilizando los recursos ofrecidos, implementa el microprocesador seleccionado en el FPGA, comunica este sistema con un computador personal y visualiza el estado real de las señales más relevantes del circuito.

A pesar de que esta configuración es totalmente funcional en el sentido de ejecución de algoritmos de cualquier complejidad, el usuario se va a ver limitado en frecuencia de funcionamiento del sistema y en accesibilidad a los puertos I/O de los microprocesadores, ya que estos parámetros están sujetos al software XISCKER Observer. Esto da la oportunidad de comprender internamente la plataforma, observando detalladamente el comportamiento de las señales dispuestas en cada arquitectura, antes de continuar con una utilización más avanzada.

Para realizar la emulación del microprocesador RISCKER ó CISCKER se requiere el software X-ISCKER IDE y los archivos de descripción de hardware correspondientes a esta etapa, aquellos que contienen la descripción del microprocesador seleccionado junto a un circuito de comunicación UART para ser utilizado con cualquier interfáz física de interconexión bidireccional con un computador personal (VCP ó RS232).

Esta etapa es la más importante en el reconocimiento de la plataforma, ya que se utilizarán todas las herramientas que dispone el X-ISCKER IDE y requiere de un conocimiento básico previo de la arquitectura y set de instrucciones expuestas en las hojas de datos de los microprocesadores.

7.5.1. Procedimiento

1. Haga click en  “Observer” para abrir la herramienta desde el XISCKER IDE.
2. Asegúrese que el diagrama de hardware corresponda a la arquitectura que está trabajando.
3. Conecte el la tarjeta de desarrollo con FPGA al computador personal en modo serial. Para esto se disponen los siguientes pines:
   a. Terasic DE0-Nano: Rx=PIN_A3 Tx=PIN_C3
   b. Spartan 3A Starter Kit: Rx=V14 Tx=V15
4. Configure la conexión serial en la barra de herramientas del XISCKER Observer.
   a. Puerto COM
   b. Tasa de Baudios
5. (Opcional) Elija el archivo *.csv en que desea tener el registro de los datos adquiridos ().
6. (Opcional) Active el almacenamiento de datos del puerto serial con el botón de grabar ().
7. Inicie la adquisición haciendo click en el botón de conexión ()

Nota: Si elije almacenar los datos recibidos y el hardware implementado en el FPGA ya contiene una comunicación serial, estos primeros datos se verán reflejados en las primeras líneas del registro *.csv.

8. Complete el procedimiento de programación de cualquier procesador XISCKER.
9. Utilice el menú que se encuentra en la unidad de control del diagrama para cambiar la frecuencia de funcionamiento del procesador o para detenerlo.
10. Utilice el campo en la unidad I/O del diagrama para enviar datos de 16 bits al puerto de entrada del procesador.
11. Una vez terminada la sesión, desconecte la comunicación serial ().
12. Para abrir el archivo *.csv en un visor/editor de hojas de calculo es importante definir las columnas de los datos binarios como texto, de lo contrario el programa elimina los ceros a la izquierda afectando la lectura inicial.

7.6. Reconfiguración del Hardware XISCKER

El conocimiento de la plataforma en esta etapa requiere una profundización en la descripción en Verilog que está disponible en la Wiki o en el sitio del Semillero ADT del microprocesador seleccionado. Los esquemas que también hacen parte de las hojas de datos y documentación de este proyecto son una descripción detallada de la organización de hardware de los microprocesadores y los módulos anexos, es por esto que se recomienda estudiarlos junto a la descripción de hardware para tener una visión estructurada del diseño y funcionamiento de todo el sistema.

Esta etapa no tiene un objetivo específico pero sí implica reconfigurar la plataforma para implementar nuevas instrucciones o funciones. Por ejemplo: Agregar un número cualquiera de módulos PWM al procesador para hacer un controlador de servomotores de forma paralela ó acelerar una instrucción en hardware que obtenga el valor medio entre dos variables.

8. BIBLIOGRAFÍA

Wiki: Sitio web con los archivos necesarios para trabajar con la plataforma XISCKER, documentación y evolución del proyecto.
semilleroadt.upbbga.edu.co/XISCKER.

Sitio web del Semillero de Tecnologías Avanzadas ADT de la Universidad Pontificia Bolivariana seccional Bucaramanga. https://semilleroadt.upbbga.edu.co

FREESCALE SEMICONDUCTOR. CPU08 Central Processor Unit. 2006.

FREESCALE SEMICONDUCTOR. 68HC11 Technical Data. 2001.

HARRIS, David Money; HARRIS, Sarah L. Digital design and computer arquitecture. San Francisco. Morgan Kaufmann Publishers, Elsevier, 2007.

SHIVA, Sajjan G. Computer Organization, Design, and Architecture. 4a Edición.

ANEXO B

HOJA DE DATOS RISCKER

9. SET DE INSTRUCCIONES RISCKER

9.1. Sintaxis de la instrucción RISKCER

Una Instrucción RISCKER - ASM está compuesta de hasta cuatro campos:

[Etiqueta] ***:*** *[Operación] [Operandos]* ***;*** *[Comentario]*

[Etiqueta]: Nombre simbólico de la localidad de memoria en que se encuentra la instrucción.

[Operación]: Indica la operación que se va a ejecutar *(Ver Tabla 1)*

[Operandos]: El campo operandos tiene hasta cuatro sub-campos dependiendo del tipo de operación.

Instrucción Completa:

```
[Etiqueta] : [Operación] [Rd],  [Rs], [Rt], #[Imm/Dir]  ;[Comentario]
```

[Rd] Registro Destino: Indica el registro en el cual se guardará el resultado de la operación.
[Rs] Registro Fuente (Source): Indica el registro en el que se encuentra un primer operando.
[Rt]* Registro Fuente: Indica un segundo registro con un dato a operar, sólo si la operación lo requiere.
#[Imm/Dir]* Valor Inmediatio o Dirección: Indica un número o dirección de 16bits.

(*) Estos campos dependen del tipo de operación.

[Comentario]: Se escribe al final de la instrucción precedido por un ";".

Ejemplos:

```
beq     rg1, rg0, #16    ;Salto condicional, si rg1 es igual a rg0 a la dirección 16
lw      rg3, r0, #0      ;Cargar el registro rg3 con el dato Memoria[r0+0]
```

```
inc     op              ;Incrementar el puerto de salida (op) y almacenar el resulta en op
```

La Tabla 2 muestra el formato de las distintas instrucciones RISCKER diviendo los campos que harán la función de operandos, código de operación (*Opcode*) y en el caso de instrucciones de asignación, los bits que identifian la función a realizar por la ALU.

Tabla 1. Listado de instrucciones RISCKER

Operación	Operandos	Descripción	
R-Type			
add	Rd, Rs, Rt	Rd = Rs + Rt	Rd = Rd + Rt
and		Rd = Rs & Rt	Rd = Rd & Rt
or		Rd = Rs | Rt	Rd = Rd | Rs
xor		Rd = Rs ^ Rt	Rd = Rd ^ Rs
nand	Rd, Rt	Rd = ~(Rs & Rt)	Rd = ~(Rd & Rs)
nor		Rd = ~(Rs | Rt)	Rd = ~(Rd | Rs)
sub		Rd = Rs - Rt	Rd = Rd - Rs
neg	Rd,Rs	Rd = -Rs	Rd = -Rd
inc	Rd	Rd = Rs + 1	Rd = Rd+1
mult	Rd,Rs,Rt	Rd[15:0] = Rs[7:0] * Rt[7:0] Rd[15:0] = Rd[7:0] * Rs[7:0]	
div	Rd,Rt	Rd = {Rs[15:0] / Rt[15:0],Rs[15:0] % Rt[15:0]} Rd = {Rd[15:0] / Rs[15:0],Rd[15:0] % Rs[15:0]}	
rot	Rd, Rt, shamt Rd,shamt	Rd = Rt rotates left by shamt	Rd = Rd rotates left by shamt
sll		Rd = Rt << Shift Amount	Rd = Rd << Shift Amount
srl		Rd = Rt >> Shift Amount	Rd = Rd >> Shift Amount
sra		Rd = Rt>>Arithmetic by shamt	Rd = Rd>>Arithmetic by shamt
I-Type			
addi	Rd,Rs,imm Rd,imm	Rd = Rs + imm	Rd = Rd + imm
andi		Rd = Rs & imm	Rd = Rd & imm
ori		Rd = Rs | imm	Rd = Rd | imm
xori		Rd = Rs ^ imm	Rd = Rd ^ imm
nandi		Rd = ~(Rs & imm)	Rd = ~(Rd & imm)
nori		Rd = ~(Rs | imm)	Rd = ~(Rd | imm)
multi		Rd = Rs[15:0] * imm	Rd = Rd[15:0] * imm
divi	Rd,Rs,imm Rd, imm	Rd = {Rs[15:0] / imm, Rs[15:0] % imm} Rd = {Rd[15:0] / imm, Rd[15:0] % imm}	
I-Type			
sw	Rd, Rs, imm Rd, imm	Memory[Rs+imm] = Rd Memory[Rd+imm] = Rd	
lw	Rd, Rs, imm Rd, imm	Rd = Memory[Rs+imm] Rd = Memory[Rd+imm]	
blt	Rd, Rs, target	((Rd-Rs)<0) ? PC = target	
beq	Rd, Rs, target Rd, target	((Rd-Rs)=0) ? PC = target ((Rd-Rd)=0) ? PC = target	

Tabla 1. Listado de instrucciones RISCKER (continuación)

J-Type		
j	target	PC = target
jrlw	Rs, Rd, imm Rd, imm	PC = Rs ; Rd = imm PC = Rs ; Rt0 = 0
jal	Rd, target	Rd = PC+1; PC = target

Tabla 2. Formato de los distintos tipos de instrucciones de la arqutiectura RISCKER

R-Type					
Assembler	**Opcode 4bits**	**Rd 6bits**	**Rs 6bits**	**Rt 6bits**	**Function 10bits**
add	0000	Rx	Rx	Rx	0000000000
and	0000	Rx	Rx	Rx	0000000001
or	0000	Rx	Rx	Rx	0000000010
xor	0000	Rx	Rx	Rx	0000000011
nand	0000	Rx	Rx	Rx	0000000100
nor	0000	Rx	Rx	Rx	0000000101
sub	0000	Rx	Rx	Rx	0000000110
neg	0000	Rx	Rx		0000000110
inc	0000	Rx	Rx		0000000111
mult	0000	Rx	Rx	Rx	0010001000
div	0000	Rx	Rx	Rx	0011001001
rot	0000	Rx		Rx	000100sham
sll	0000	Rx		Rx	000101sham
srl	0000	Rx		Rx	000110sham
sra	0000	Rx		Rx	000111sham
I-Type Operations					
Assembler	**Opcode 4bits**	**Rd 6bits**	**Rs 6bits**	**Immediate (16 bits) Two's Complement**	
addi	1000	Rx	Rx	imm	
andi	1001	Rx	Rx	imm	
ori	1010	Rx	Rx	imm	
xori	1011	Rx	Rx	imm	
nandi	1100	Rx	Rx	imm	
nori	1101	Rx	Rx	imm	
multi	1110	Rx	Rx	imm	
divi	1111	Rx	Rx	imm	

Tabla 2. Formato de los distintos tipos de instrucciones de la arquitectura RISCKER (continuación)

I-Type				
sw	0010	Rx	Rx	imm
lw	0011	Rx	Rx	imm
blt	0100	Rx	Rx	target
beq	0101	Rx	Rx	target
J-Type				
Assembler	**Opcode 4bits**	**Rd 6bits**	**Rs 6bits**	**Immediate (16 bits) Two's Complement**
j	0001	target		
jrlw	0110	Rx		
jal	0111	Rx	target	

9.2. Características principales del Hardware

9.2.1. Registros

RISCKER cuenta con 64 registros organizados como se muestran en la tabla 3, divididos en tres tipos.

9.2.1.1. Registros Modificables

Estos registros permiten escritura y lectura en todas las instrucciones. Están divididos para cumplir distintas funciones de programa pero su implementación en hardware es la misma. El buen uso de estos registros permite la reutilización sencilla de código.

Valores de Retorno (rv0-rv7)
Registros destinados para retornar valores de subrutinas.

Argumentos de subrutina (rp0-rp7)
Registros destinados para llevar valores como argumentos de subrutinas.

Registros de uso Temporal (rt0-rt7)
Registros para uso temporal como intermediario en algoritmos complejos o de subrutinas.

Registros Globales (rg0-rg21)

Registros para definir constantes y variables de programa.

Tabla 3. Archivo de registros RISCKER

# of Registers	Name	Index	Descripción
1	r0	0	Valor 0
8	rv0-rv7	1:8	Valores de Retorno
8	rp0-rp7	9:16	Argumentos de Subrutina
8	rt0-rt7	17:24	Registros de uso Temporal
22	rg0-rg21	25-46	Registros Globales
4	ra0-ra3	47-50	Dirección de Retorno
1	sp	51	Puntero de la Pila
2	pr1-pr2	52-53	Periodo de Timers
2	ti1-ti2	54-55	Direcciones de Interrupciones por Timers
2	eir1-eir2	56-57	Dirección de Interrupciones Externas
1	op	58	Registro de Salida
1	rc	59	Registro de Control
2	tmr1-tmr2	60-61	Registros de Timers
1	ip	62	Registro de Entrada
1	ipc	63	PC en el momento de la interrupción
64			

Modificables
Proposito Especial
Sólo Lectura

Dirección de Retorno (ra0-ra3)
Registros para direcciones de retorno de procedimientos. Se permiten hasta cuatro procedimientos en cadena teniendo en cuenta la dirección de retorno correspondiente.

Puntero de la Pila (sp)
Registro que debe apuntar a la última localidad ocupada de la pila. Su utilización es completamente por software, decrementando su valor antes de una escritura e incrementándolo después de una lectura.

9.2.1.2. Registros de Sólo Lectura

Son registros especiales que contienen información relacionada al funcionamiento del microprocesador. Su entrada está conectada directamente a señales externas. Y por lo tanto sus valores no pueden ser modificados por software.

Registro 0 (r0)
Constante cero mapeada en el archivo de registros.

Timers (tmr1, tmr2)
Contiene el valor del contador de cada temporizador.

Puerto de Entrada (ip)
Contiene la información del puerto de entrada.

Interrupt PC (ipc)
Contiene el valor del Contador de programa antes de ejecutar una interrupción.

9.2.1.3. Registros de propósito especial

Son registros que configuran características del procesador. Su contenido puedo ser leído pero estas señales también están conectadas a componentes externos.

Periodo del Timer (pr1, pr2)
Contiene el valor en el cual ocurre el desbordamiento de cada temporizador con lo cual genera una señal de interrupción.

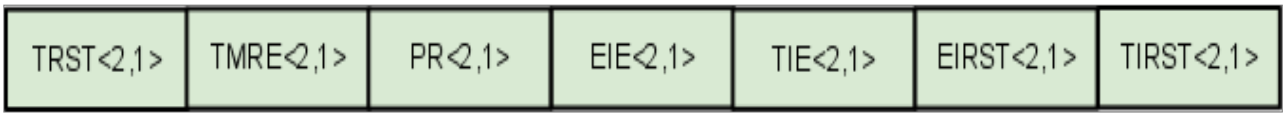
Direcciones de Interrupción por desbordamiento de Timer (ti1, ti2)
Contiene la dirección en la que se encuentra el servicio de interrupción por desbordamiento de cada timer. Esta señal está conectada al circuito controlador de interrupciones.

Direcciones de Interrupción Externa (eir1, eir2)
Contiene la dirección en la que se encuentra el servicio de cada interrupción externa. Esta señal está conectada al circuito controlador de interrupciones.

Puerto de Salida (op)
Registro conectado como salida del microprocesador.

Registro de control (rc)
Registro de banderas de configuración de interrupciones y temporizadores:

Figura 1. Organización de señales en el registro de control RISCKER



[0:1] TIRST<1,2> = Timer Interrupt Reset Flag
Borra la bandera de interrupción por termporizador 1 o 2

[2:3] EIRST<1,2> = External Interrupt Reset Flag
Borra la bandera de interrupción externa 1 o 2

[4:5] TIE<1,2> = Timer Interrupt Enable
Habilita la Interrupción por temporizador 1 o 2

[6:7] EIE<1,2> = External Interrupt Enable
Habilita la Interrupción externa 1 o 2

[8:9] PR1<1,2> = Timer1 Prescaler

[10:11] PR2<1,2> = Timer2 Prescaler

[12:13] TMRE<1,2>= Timer Enable
Habilita el temporizador 1 o 2

[14:15] TRST<1,2> = Timer Reset
Borra el contenido del temporizador 1 o 2

9.2.2. Ruta de Datos y Unidad de Control

La figura 2 es el diagrama de la ruta de datos RISCKER con las señales de control provenientes de la HCU (Hardware Controlled Unit), en esta imagen se pueden observar el archivo de registros, la ALU (Arithmetic Logic Unit) y la unidad de memoria. Las señales de control están descritas en la tabla 4 y su comportamiento está determinado por los estados descritos en la figura 3 y las tablas 5 y 6.

La tabla 5 muestra las señales que dependen de los estados de la unidad de control. La tabla 6 muestra las señales que dependen únicamente del opcode de la instrucción y por lo tanto no cambian durante los ciclos de ejecución de la misma.

La figura 3 representa la máquina de estados de la HCU RISCKER. Esta contempla todas las instrucciones en los estados *FETCH, DECODE* y *EXECUTE*, como también los estados de reconocimiento y retorno de interrupción.

Figura 2. Ruta de Datos RISCKER

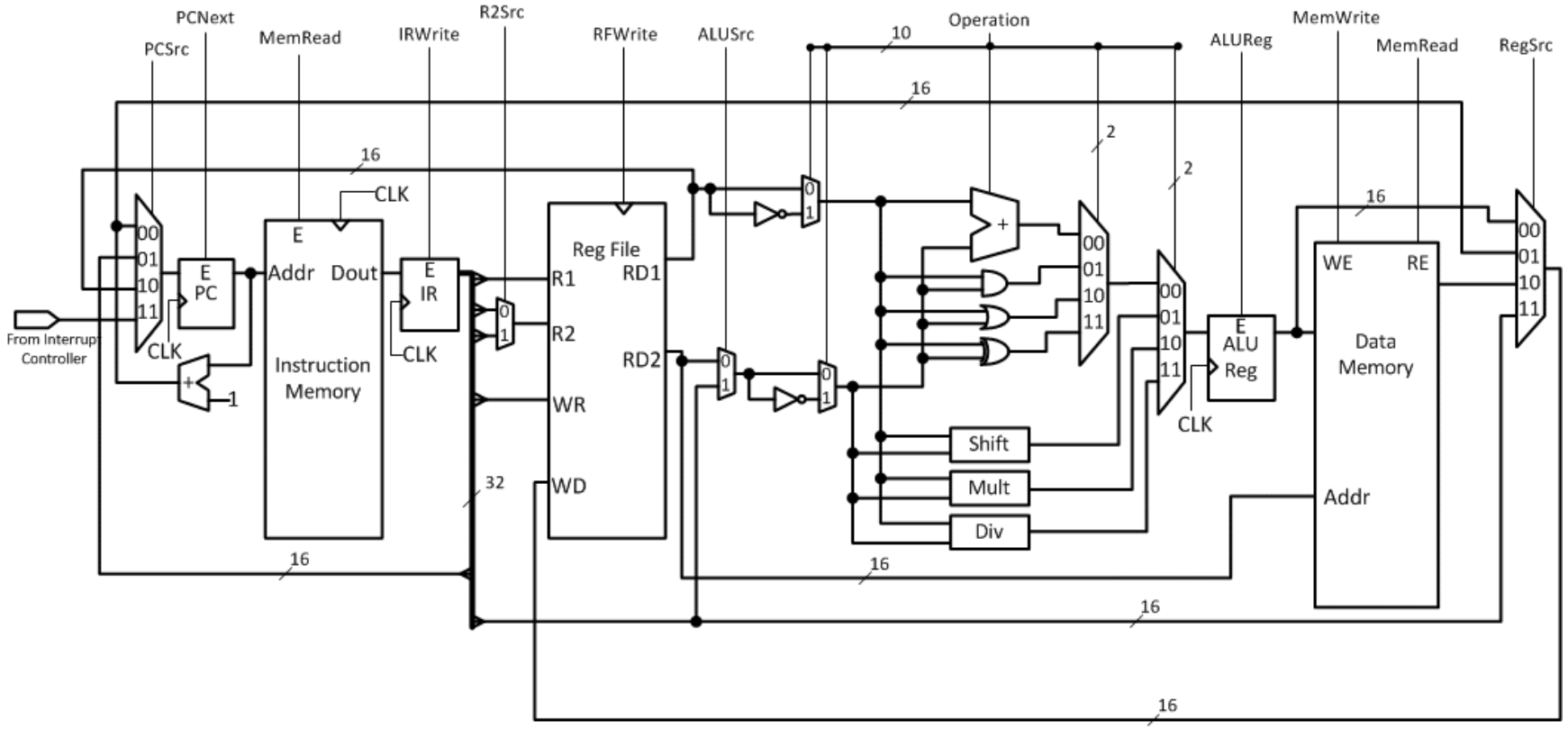


Tabla 4. Señales de Control

PCsrc	Se encarga de seleccionar la fuente de carga del registro PC
PCNext	Flanco de reloj del registro PC
IRwrite	Flanco de reloj del registro de instrucción
R2src	Fuente de la dirección del registro 2 del archivo de registros
RFwrite	Flanco de reloj de escritura del archivo de registros
ALUsrc	Fuente del operando B de la unidad de operaciones
Operation	Bus de control que selecciona la operación indicada
ALUreg	Flanco de reloj que registra el resultado de la ALU
MemWrite	Habilitador de escritura de la memoria de datos
MemRead	Habilitador de lectura de la memoria de datos
RegSrc	Fuente de datos a escribir en el archivo de registros
IPCWrite	Flanco de reloj del registro PC' de interrupción
INToggle	Señal que indica el inicio o final de servicio de interrupción.

Tabla 5. Señales de control de la máquina de estados

Estado	ipc_write	intoggle	pc_next	ir_write	rf_write	alu_reg	mem_write
idle	0	0	0	0	0	0	0
fetch	0	0	0	1	0	0	0
execute_op	0	0	1	0	0	1	0
execute_jrlw	0	0	1	0	1	0	0
write_op	0	0	0	0	1	0	0
write_sw	0	0	0	0	0	0	1
jump	0	0	1	0	0	0	0
interrupt	1	1	0	0	0	0	0
intjump	1	0	1	0	0	0	0
retfi	0	1	0	0	0	0	0

Figura 3. Diagrama de estados RISCKER

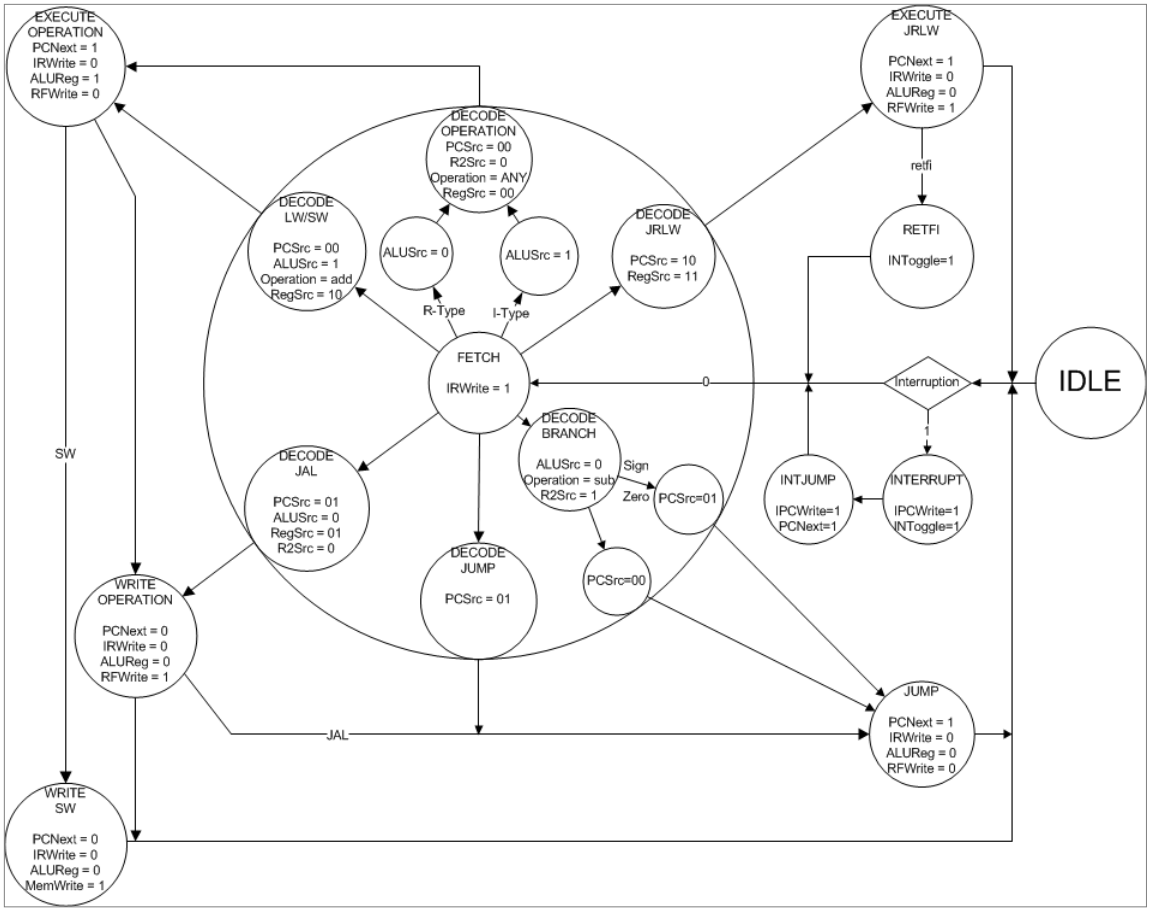

EXECUTE
OPERATION
PCNext = 1
IRWrite = 0
ALUReg = 1
RFWrite = 0
DECODE
OPERATION
PCSrc = 00
R2Src = 0
Operation = ANY
RegSrc = 00
DECODE
LW/SW
PCSrc = 00
ALUSrc = 1
Operation = add
RegSrc = 10
ALUSrc = 0
ALUSrc = 1
R-Type
I-Type
DECODE
JRLW
PCSrc = 10
RegSrc = 11
EXECUTE
JRLW
PCNext = 1
IRWrite = 0
ALUReg = 0
RFWrite = 1
retfi
RETFI
INToggle=1
FETCH
IRWrite = 1
0
Interruption
IDLE
1
SW
DECODE
JAL
PCSrc = 01
ALUSrc = 0
RegSrc = 01
R2Src = 0
DECODE
BRANCH
ALUSrc = 0
Operation = sub
R2Src = 1
Sign
Zero
PCSrc=01
INTJUMP
IPCWrite=1
PCNext=1
INTERRUPT
IPCWrite=1
INToggle=1
DECODE
JUMP
PCSrc = 01
PCSrc=00
WRITE
OPERATION
PCNext = 0
IRWrite = 0
ALUReg = 0
RFWrite = 1
JAL
JUMP
PCNext = 1
IRWrite = 0
ALUReg = 0
RFWrite = 0
WRITE
SW
PCNext = 0
IRWrite = 0
ALUReg = 0
MemWrite = 1

Tabla 6. Señales de control independientes de la máquina de estados

Instruction	PCSrc	R2Src	ALUSrc	Operation (11 bits)				RegSrc
				aluop	shamt	shift_mode	select	
add	00	0	0	000	XXXX	XX	00	00
and	00	0	0	001	XXXX	XX	00	00
or	00	0	0	010	XXXX	XX	00	00
xor	00	0	0	011	XXXX	XX	00	00
nand	00	0	0	100	XXXX	XX	00	00
nor	00	0	0	101	XXXX	XX	00	00
sub	00	0	0	110	XXXX	XX	00	00
neg	00	0	0	110	XXXX	XX	00	00
inc	00	0	0	111	XXXX	XX	00	00
mult	00	0	0	XXX	XXXX	XX	10	00
div	00	0	0	XXX	XXXX	XX	11	00
rot	00	0	0	XXX	SHAM	00	01	00
sll	00	0	0	XXX	SHAM	01	01	00
srl	00	0	0	XXX	SHAM	10	01	00
sra	00	0	0	XXX	SHAM	11	01	00
Instruction	**PCSrc**	**R2Src**	**ALUSrc**	**aluop**	**shamt**	**shift_mode**	**select**	**RegSrc**
addi	00	X	1	000	XXXX	XX	00	00
andi	00	X	1	001	XXXX	XX	00	00
ori	00	X	1	010	XXXX	XX	00	00
xori	00	X	1	011	XXXX	XX	00	00
nandi	00	X	1	100	XXXX	XX	00	00
nori	00	X	1	101	XXXX	XX	00	00
multi	00	X	1	XXX	XXXX	XX	10	00
divi	00	X	1	XXX	XXXX	XX	11	00
Instruction	**PCSrc**	**R2Src**	**ALUSrc**	**aluop**	**shamt**	**shift_mode**	**select**	**RegSrc**
lw	00	X	1	000	XXXX	XX	00	10
sw	00	X	1	000	XXXX	XX	00	XX
beq	01	1	0	110	XXXX	XX	00	XX
blt	01	1	0	110	XXXX	XX	00	XX
jrlw	10	0	X	XXX	XXXX	XX	XX	11

Tabla 6. Señales de control independientes de la máquina de estados

(continuación)

Instruction	PCSrc	R2Src	ALUSrc	aluop	shamt	shift_mode	select	RegSrc
jal	01	X	X	XXX	XXXX	XX	XX	01
j	01	1	0	XXX	XXXX	XX	XX	01

9.2.3. Controlador de interrupciones

El manejo de las interrupciones en el microprocesador RISCKER se logra con el circuito expuesto en la figura 4. Dispone de cuatro fuentes de interrupción implementadas por defecto como dos interrupciones por temporizadores y dos interrupciones externas. El funcionamiento de este circuito está determinado por las siguientes señales.

Control Register: Registro de control del archivo de registros. En él se encuentran las señales “Enable” y “Reset” de cada interrupción.

Interruption Toggle: Señal proveniente de la unidad de control como bandera de entrada y salida de la rutina de servicio de interrupción.

Interruption: Bandera de entrada a la máquina de estados para reconocimiento de interrupciones.

IPC Write: Señal de escritura del contador de programa en el registro de almacenamiento para retorno de interrupciones. Esta señal borra automáticamente la señal “Interruption”.

Return from Interruption (Retfi): Señal que indica el retorno de la interrupción al programa mediante el “Reset” de cualquiera de las banderas.

Address Interruption X: Registros del archivo de registros que contienen la dirección de la rutina de servicio de interrupción correspondiente a la interrupción reconocida. Estas direcciones tienen un orden de prioridad configurable en hardware.

Interruption Address: Dirección de la rutina de servicio de interrupción correspondiente a la interrupción reconocida cuando se activa la señal *Interruption*.

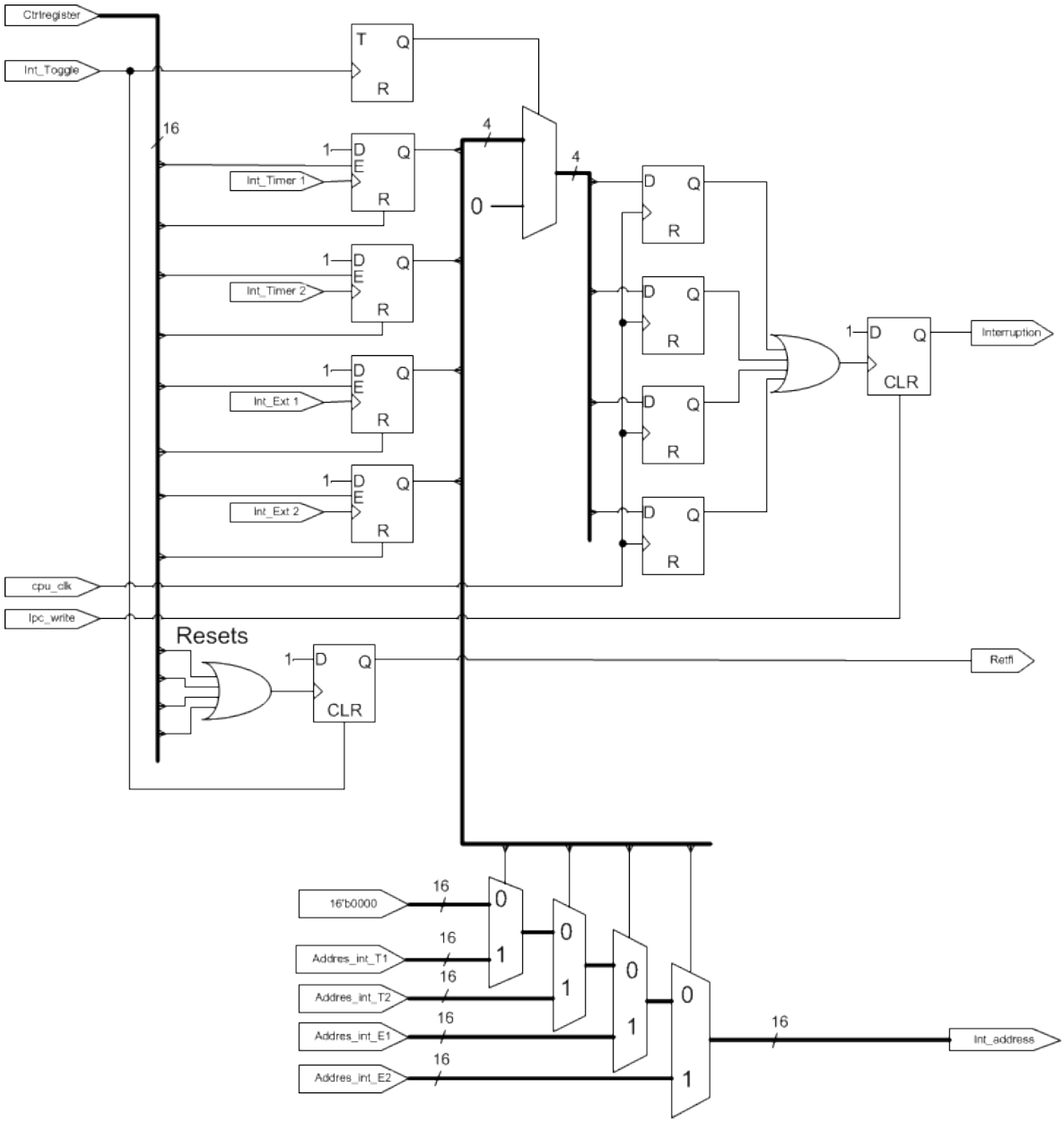
Algoritmo para el servicio de interrupciones:

Deshabilitar temporalmente las interrupciones.
Guardar en memoria los registros que se modifican dentro de la interrupción.
Ejecutar la subrutina de interrupción.
Recuperar los valores de los registros que se guardaron en memoria.
Habilitar las interrupciones.
Hacer reset a la bandera de la interrupción que se generó.
Cargar el contador de programa con el valor guardado en el registro ipc.

Manejo de la pila:

```
addi sp, sp, -3       ;Dejar el espacio en la pila para almacenar registros
sw rg0, sp, 2         ;Guardar rg0 de último en la pila
sw rg1, sp, 1         ;Guardar rg1 de segundo en la pila
sw rg2, sp, 0         ;Guardar rg2 de primero en la pila
...
;Subrutina de servicio de interrupción
;que utiliza los registros rg0,rg1 y rg2
...
lw rg2, sp, 0         ;Restaurar rg2 desde la pila
lw rg1, sp, 1         ;Restaurar rg1 desde la pila
lw rg0, sp, 2         ;Restaurar rg0 desde la pila
addi sp, sp, 3        ;Retornar el puntero de la pila
jrlw ra               ;Retornar a la rutina que llamo este procedimiento
```

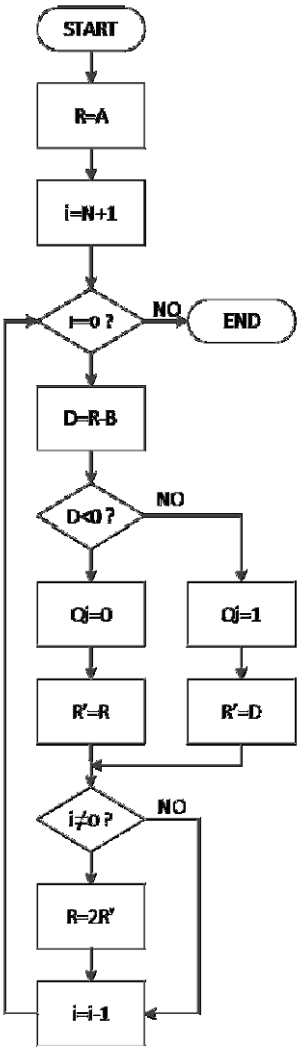
Figura 4. Controlador de Interrupciones RISCKER



9.2.4. Divisor

El procesador RISCKER cuenta con un módulo de división por hardware que consiste en la implementación del algoritmo para dividir números enteros sin signo que se muestra en el siguiente diagrama de flujo. Dónde N es número de bits, A es dividendo, B el divisor, Q el cociente y R el residuo.

Figura 5. Algoritmo de división

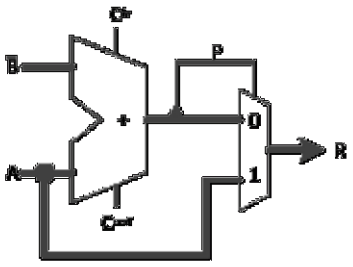


Para llevar a cabo la división se inicializa el residuo con el valor del dividendo.

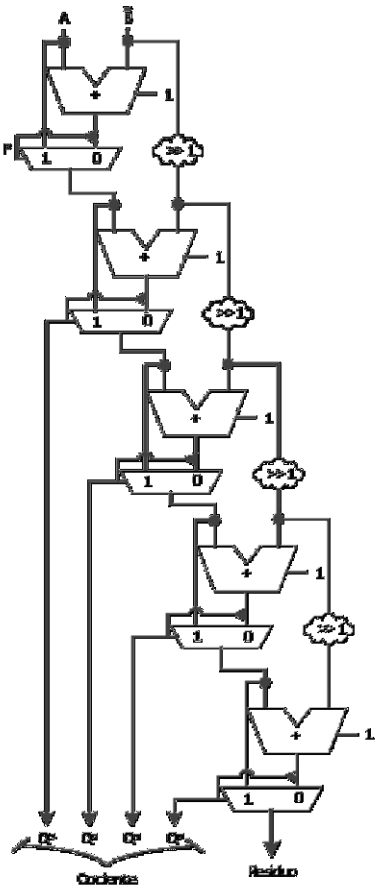
Este residuo parcial se le resta al divisor repetidamente para determinar cuántas veces cabe. Si la diferencia entre residuo y divisor es negativa, se asigna cero al cociente y se descarta el resultado de la resta. Si por el contrario la diferencia es positiva, se asigna uno al cociente y el residuo recibe la diferencia. En cualquier caso el residuo se multiplica por dos y se repite el proceso N+1 veces.

Para implementar en Hardware el algoritmo descrito, se entiende cada iteración como una etapa del circuito, cada etapa lleva a cabo todas las operaciones correspondientes a una repetición del algoritmo, por lo tanto el circuito tendrá N+1 etapas. La Figura 5.a muestra la implementación en hardware de una sola etapa, dónde A y B son números enteros, P el bit de signo del resultado de la resta, Cin el carry de entrada y Cout el de salida y R el residuo parcial de cada etapa.

Figura 6. Diagrama esquemático de un divisor paralelo de 4 bits.
(a) Celda del Divisor paralelo



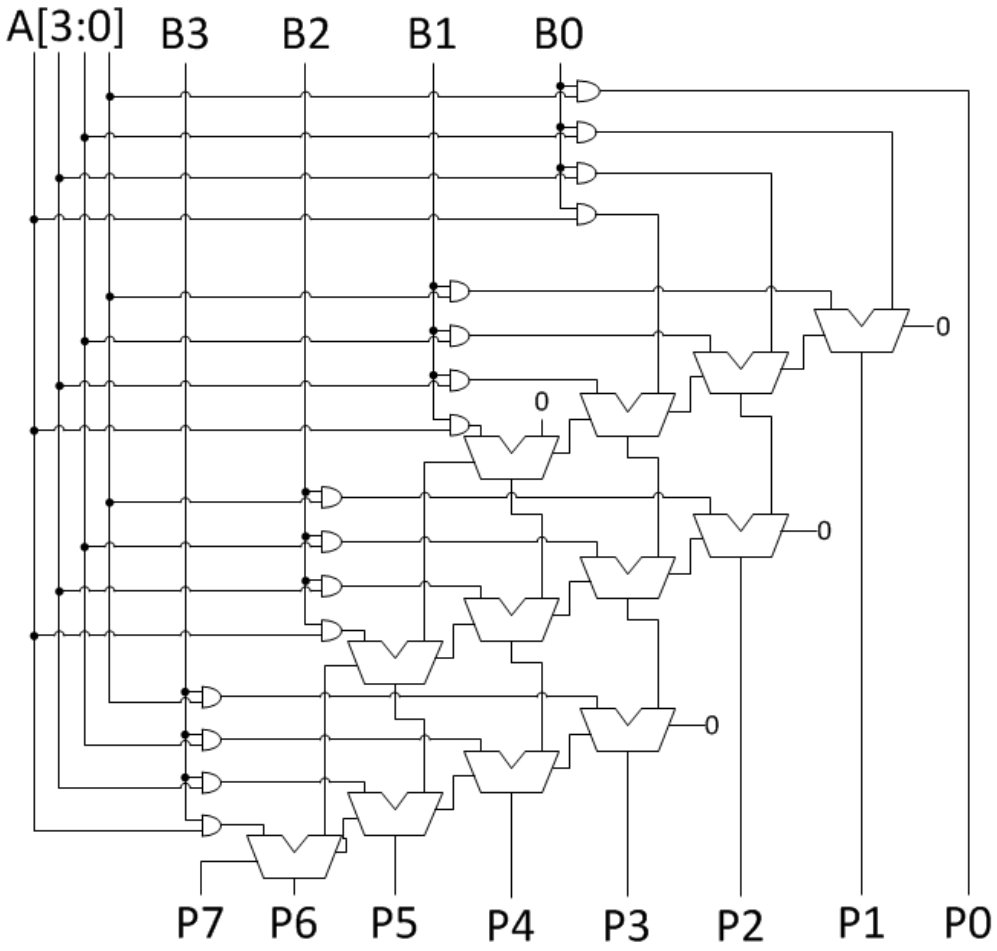
(b) Sistema completo



9.2.5. Multiplicador

Igualmente se implementa un circuito de multiplicación en hardware de 16 bits. Un ejemplo con operandos de 4 bits se muestra en la figura 6. El algoritmo realiza, bit a bit, la multiplicación con la función lógica AND y suma los resultados con un corrimiento.

Figura 7. Diagrama esquemático de un multiplicador paralelo de 4 bits



ANEXO C

HOJA DE DATOS CISCKER

10. Set de Instrucciones CISCKER

10.1. Sintaxis de la instrucción CISKCER

Una Instrucción CISCKER - ASM está compuesta de hasta cinco campos:

[Etiqueta]: [Operación] [Operando], [MD] ;[Comentarios]

[Etiqueta]: Nombre simbólico de la localidad de memoria en que se encuentra la instrucción.

[Operación]: Indica la operación que se va a ejecutar *(Ver Tabla 1)*

[Operando]: Indica el operando con el que se va a ejecutar la operación
Se precede de "**#**" si el operando es un número.
Se precede de "$" si el operando es una dirección.

[MD] Modo de Direccionamiento: Se precede de ",". Indica el modo de direccionamiento de la instrucción de forma explícita: IX o IX+. Los demás modos de direccionamientos se infieren de la operación y los operandos.

[Comentarios]: Se escribe al final de la instrucción y precedido por un ";".

Ejemplos:

```
ADDA    #10,    IX      ;   Sumar   ACCA    con     Memoria[IX+10]
INCA            ; Incrementar ACCA
```

La Tabla 1 es el mapa de *opcodes* que muestra todas las instrucciones implementadas en el microprocesador CISCKER, agrupadas en colores por la trama de micro-instrucciones en que se decodifican. Esto permite diferenciar modos de direccionamiento y tipos de instrucciones: Saltos condicionales, manipulación de bits, control de flujo de programa y aquellas con operandos de 8 o 16bits.

La Tabla 2 divide el mapa por la operación que realiza la ALU con la correlación de colores correspndiente a cada operación, incluyendo aquellas que no utilizan la ALU para su ejecucución como el control de bits del CCR o del flujo de programa.

Las instrucciones también pueden ser divididas como muestra la Tabla 3, por los bits que modifican el registro CCR, esta información facilita la descripción de instrucciones de salto condicional dado por las condiciones lógicas presentadas en la Tabla 4.

Tabla 1. Mapa de OpCodes divido por u-instrucciones

	Bit	Branch	Read Modify Write					Register Memory								
	INH	REL	INH	INH	INH	IX	EXT	IX+	IMM	DIR	IX	EXT	IMM	DIR	IX	EXT
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NOP	BHCC	TSX	NEGA	NEGB	NEG	NEG	SUBA	SUBA	SUBA	SUBA	SUBA	SUBB	SUBB	SUBB	SUBB
1	STOP	BHCS	TXS	CLRA	CLRB	CLR	CLR	SBCA	SBCA	SBCA	SBCA	SBCA	SBCB	SBCB	SBCB	SBCB
2	DABNZ	BHI	INS	INCA	INCB	INC	INC	ADDA	ADDA	ADDA	ADDA	ADDA	ADDB	ADDB	ADDB	ADDB
3	DBBNZ	BLS	DES	DECA	DECB	DEC	DEC	ANDA	ANDA	ANDA	ANDA	ANDA	ANDB	ANDB	ANDB	ANDB
4	BRN	BCC	INX	LSRA	LSRB	LSR	LSR	LDA	LDA	LDA	LDA	LDA	LDB	LDB	LDB	LDB
5	BRA	BCS	DEX	RORA	RORB	ROR	ROR	ORA	ORA	ORA	ORA	ORA	ORB	ORB	ORB	ORB
6	BIL	BNE	PULX	ASRA	ASRB	ASR	ASR	EORA	EORA	EORA	EORA	EORA	EORB	EORB	EORB	EORB
7	BIH	BEQ	PSHX	LSLA	LSLB	LSL	LSL	ADCA	ADCA	ADCA	ADCA	ADCA	ADCB	ADCB	ADCB	ADCB
8	IDIV	BVC	PULA	ROLA	ROLB	ROL	ROL	CMPA	CMPA	CMPA	CMPA	CMPA	CMPB	CMPB	CMPB	CMPB
9	MUL	BVS	PULB	TSTA	TSTB	TST	TST	BITA	BITA	BITA	BITA	BITA	BSR		JSR	JSR
A	CLV	BPL	PSHA	TDX		BRSET	BRSET	STA		STA	STA	STA		STB	STB	STB
B	SEV	BMI	PSHB			BRCLR	BRCLR		AIS	STS	STS	STS	AIX	STX	STX	STX
C	CLC	BGE	RTS	TBA	TAB	BCLR	BCLR			STD	STD	STD	CPX	CPX	CPX	CPX
D	SEC	BLT	RTI			BSET	BSET	CPD	LDS	LDS	LDS	LDS	LDX	LDX	LDX	LDX
E	CLI	BGT	WAIT	SBA	CBA	MOV	MOV	SUBD	SUBD	SUBD	SUBD	SUBD	CPD	CPD	CPD	CPD
F	SEI	BLE	SWI	ABA		JMP	JMP	ADDD	ADDD	ADDD	ADDD	ADDD	LDD	LDD	LDD	LDD
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
	INH	REL	INH	INH	INH	IX	EXT	IX+	IMM	DIR	IX	EXT	IMM	DIR	IX	EXT

Tabla 2. Mapa de OpCodes divido por operaciones

	Bit	Branch	Read Modify Write					Register Memory								
	INH	REL	INH	INH	INH	IX	EXT	IX+	IMM	DIR	IX	EXT	IMM	DIR	IX	EXT
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NOP	BHCC	TSX	NEGA	NEGB	NEG	NEG	SUBA	SUBA	SUBA	SUBA	SUBA	SUBB	SUBB	SUBB	SUBB
1	STOP	BHCS	TXS	CLRA	CLRB	CLR	CLR	SBCA	SBCA	SBCA	SBCA	SBCA	SBCB	SBCB	SBCB	SBCB
2	DABNZ	BHI	INS	INCA	INCB	INC	INC	ADDA	ADDA	ADDA	ADDA	ADDA	ADDB	ADDB	ADDB	ADDB
3	DBBNZ	BLS	DES	DECA	DECB	DEC	DEC	ANDA	ANDA	ANDA	ANDA	ANDA	ANDB	ANDB	ANDB	ANDB
4	BRN	BCC	INX	LSRA	LSRB	LSR	LSR	LDA	LDA	LDA	LDA	LDA	LDB	LDB	LDB	LDB
5	BRA	BCS	DEX	RORA	RORB	ROR	ROR	ORA	ORA	ORA	ORA	ORA	ORB	ORB	ORB	ORB
6	BIL	BNE	PULX	ASRA	ASRB	ASR	ASR	EORA	EORA	EORA	EORA	EORA	EORB	EORB	EORB	EORB
7	BIH	BEQ	PSHX	LSLA	LSLB	LSL	LSL	ADCA	ADCA	ADCA	ADCA	ADCA	ADCB	ADCB	ADCB	ADCB
8	IDIV	BVC	PULA	ROLA	ROLB	ROL	ROL	CMPA	CMPA	CMPA	CMPA	CMPA	CMPB	CMPB	CMPB	CMPB
9	MUL	BVS	PULB	TSTA	TSTB	TST	TST	BITA	BITA	BITA	BITA	BITA	BSR	JSR	JSR	JSR
A	CLV	BPL	PSHA	TDX		BRSET	BRSET	STA		STA	STA	STA		STB	STB	STB
B	SEV	BMI	PSHB			BRCLR	BRCLR		AIS	STS	STS	STS	AIX	STX	STX	STX
C	CLC	BGE	RTS	TBA	TAB	BCLR	BCLR			STD	STD	STD	CPX	CPX	CPX	CPX
D	SEC	BLT	RTI			BSET	BSET	SUBD	LDS	LDS	LDS	LDS	LDX	LDX	LDX	LDX
E	CLI	BGT	WAIT	SBA	CBA	MOV	MOV	CPD	SUBD	SUBD	SUBD	SUBD	CPD	CPD	CPD	CPD
F	SEI	BLE	SWI	ABA		JMP	JMP	ADDD	ADDD	ADDD	ADDD	ADDD	LDD	LDD	LDD	LDD
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
	INH	REL	INH	INH	INH	IX	EXT	IX+	IMM	DIR	IX	EXT	IMM	DIR	IX	EXT

	Sub		NAND		Sub with carry		Pass	
	Add		NOR		Shift		Control	
	AND		Increment		Neg			
	OR		Decrement		cCear			
	XOR		Add with carry		Don't Care			

Tabla 3. Mapa de OpCodes divido por señales modificadas del CCR

	Bit	Branch	Read Modify Write					Register Memory								
	INH	REL	INH	INH	INH	IX	EXT	IX+	IMM	DIR	IX	EXT	IMM	DIR	IX	EXT
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NOP	BHCC	TSX	NEGA	NEGB	NEG	NEG	SUBA	SUBA	SUBA	SUBA	SUBA	SUBB	SUBB	SUBB	SUBB
1	STOP	BHCS	TXS	CLRA	CLRB	CLR	CLR	SBCA	SBCA	SBCA	SBCA	SBCA	SBCB	SBCB	SBCB	SBCB
2	DABNZ	BHI	INS	INCA	INCB	INC	INC	ADDA	ADDA	ADDA	ADDA	ADDA	ADDB	ADDB	ADDB	ADDB
3	DBBNZ	BLS	DES	DECA	DECB	DEC	DEC	ANDA	ANDA	ANDA	ANDA	ANDA	ANDB	ANDB	ANDB	ANDB
4	BRN	BCC	INX	LSRA	LSRB	LSR	LSR	LDA	LDA	LDA	LDA	LDA	LDB	LDB	LDB	LDB
5	BRA	BCS	DEX	RORA	RORB	ROR	ROR	ORA	ORA	ORA	ORA	ORA	ORB	ORB	ORB	ORB
6	BIL	BNE	PULX	ASRA	ASRB	ASR	ASR	EORA	EORA	EORA	EORA	EORA	EORB	EORB	EORB	EORB
7	BIH	BEQ	PSHX	LSLA	LSLB	LSL	LSL	ADCA	ADCA	ADCA	ADCA	ADCA	ADCB	ADCB	ADCB	ADCB
8	IDIV	BVC	PULA	ROLA	ROLB	ROL	ROL	CMPA	CMPA	CMPA	CMPA	CMPA	CMPB	CMPB	CMPB	CMPB
9	MUL	BVS	PULB	TSTA	TSTB	TST	TST	BITA	BITA	BITA	BITA	BITA	BSR	JSR	JSR	JSR
A	CLV	BPL	PSHA	TDX		BRSET	BRSET	STA		STA	STA	STA		STAB	STAB	STAB
B	SEV	BMI	PSHB			BRCLR	BRCLR		AIS	STS	STS	STS	AIX	STX	STX	STX
C	CLC	BGE	RTS	TBA	TAB	BCLR	BCLR			STD	STD	STD	CPX	CPX	CPX	CPX
D	SEC	BLT	RTI			BSET	BSET	SUBD	LDS	LDS	LDS	LDS	LDX	LDX	LDX	LDX
E	CLI	BGT	WAIT	SBA	CBA	MOV	MOV	CPD	SUBD	SUBD	SUBD	SUBD	CPD	CPD	CPD	CPD
F	SEI	BLE	SWI	ABA		JMP	JMP	ADDD	ADDD	ADDD	ADDD	ADDD	LDD	LDD	LDD	LDD
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
	INH	REL	INH	INH	INH	IX	EXT	IX+	IMM	DIR	IX	EXT	IMM	DIR	IX	EXT

8bits Logic	V,N,Z	set or clr ccr bits	CCR
16bits Arithmetic	V,H,N,Z,C	8 bits Aritmetic	V,N,Z,C
16bits logic	V,N,Z	8 bits INC/DEC	V,N,Z
8bits shift	V,N,Z,C	No actualizan CCR	
don´t care		Branch Bit test	Z,C

Tabla 4. Condiciones de las Instrucciones de salto condicional.

BRN	1'b0	BLS	C|Z==1	BPL	N==0	DABNZ	Z==0
BRA	1'b1	BCC	C==0	BMI	N==1	DBBNZ	Z==0
BIL	I==0	BCS	C==1	BGE	N^V==0	BRSET	Z==0
BIH	I==1	BNE	Z==0	BLT	N^V==1	BRCLR	Z==0
BHCC	H==0	BEQ	Z==1	BGT	Z|N^V==1	BRSET	Z==0
BHCC	H==1	BVC	V==0	BLE	Z|N^V==1	BRCLR	Z==0
BHI	C|Z==0	BVS	V==1				

10.2. Ruta de datos

La Figura 1 muestra la ruta de datos CISCKER, en esta se muestran las unidades lógico-aritmética, de memoria (Von-Neumann), control y de entradas y salidas. Los números corresponden a las señales decritas en la Tabla 5.

Figura 1. Diagrama esquematico de la ruta de datos de la *arqutiectura* CISCKER

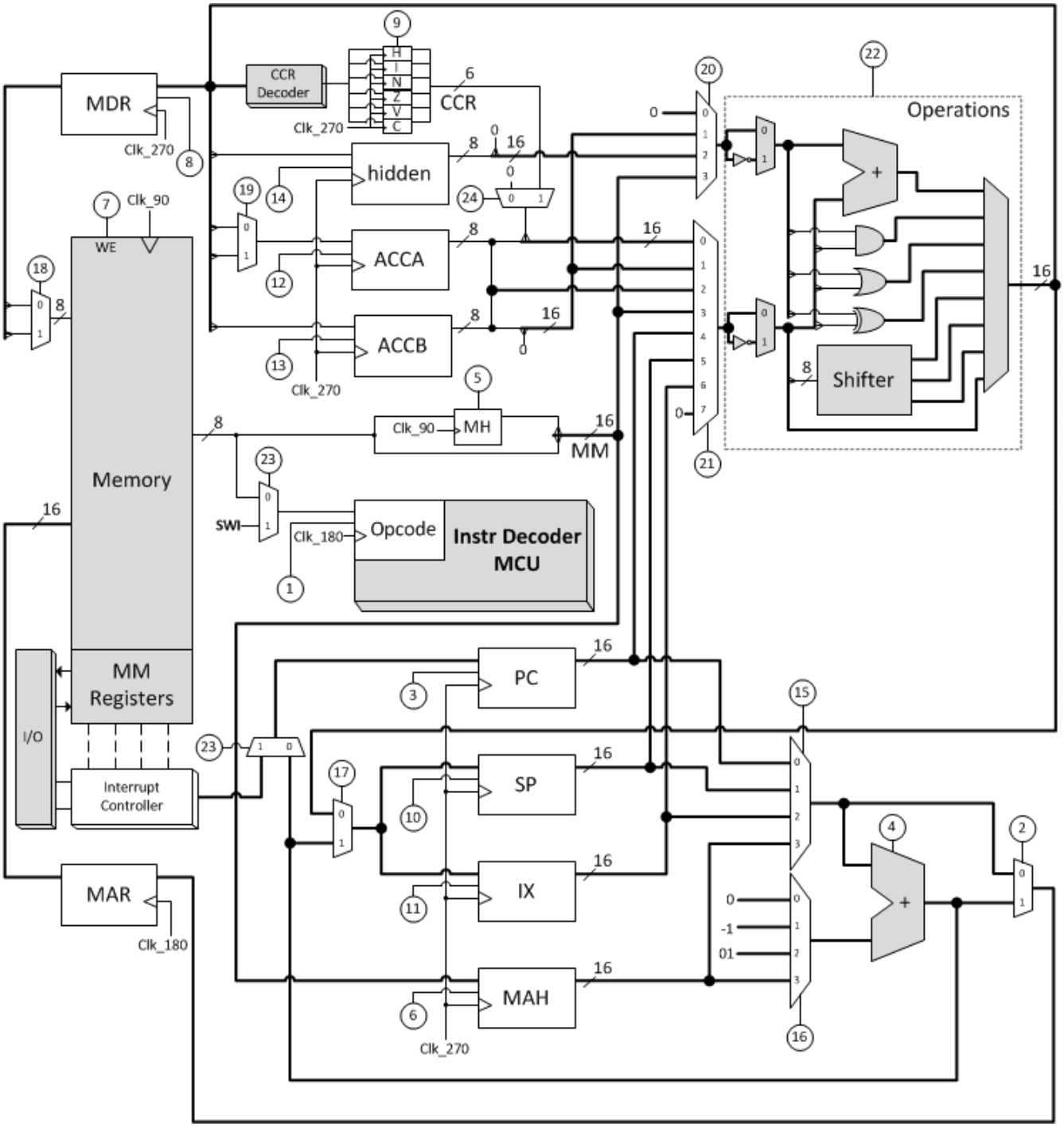


Tabla 5. Señales de control en la ruta de datos CISCKER

Número	Nombre	Descripción
1	opcode_en	Habilitador del registro de opcode
2	pass_adder	Selecciona la fuente del registro MAR
3	pc_en	Habilitador del registro PC
4	adder_cin	Acarreo de entrada al sumador de direcciones
5	h_reset	Reset del registro MH
6	mah_en	Habilitador del registro de memoria para direcciones
7	we	Habilitador de escritura de la memoria
8	mdr_en	Habilitador del registro MDR
9	ccr_en	Habilitador del registro CCR
10	sp_en	Habilitador del registro SP
11	ix_en	Habilitador del registro IX
12	acca_en	Habilitador del ACCA
13	accb_en	Habilitador del ACCB
14	hidden_en	Habilitador del registro HIDDEN
15	adder_src_a[1:0]	Selecciona el operando A del sumador de direcciones
16	adder_src_b[1:0]	Selecciona el operando B del sumador de direcciones
17	adder_alu	Selecciona la fuente de los registros IX,SP
18	wd_src[1:0]	Selecciona el dato que se va a escribir en memoria
19	acca_src	Selecciona la fuente del siguiente valor de ACCA
20	alu_src_a[2:0]	Selecciona el operando A de la ALU
21	alu_src_b[1:0]	Selecciona el operando B de la ALU
22	alu_op[5:0]	Determina la operación de la ALU
23	Interruption	Señal de interrupción
24	wd_ccr	Selecciona CCR como operando de la ALU para pasar a MDR

10.3. Unidad de Control

Las señales de control CISCKER son generadas por la MCU que decodifica los *Opcodes,* leídos de memoria, en las micro-instrucciones descritas en la tabla 6. Cada micro-instrucción es a su vez decodificada generando las señales que se muestran en la tabla 7 para generar las distintas etapas de ejecución de una instrucción..

Tabla 6. Micro-Instrucciones CISCKER

	Nombre	Código	Descripción
p	program fetch	00	Lectura del siguiente byte de memoria.
r	read byte	01	Lectura de un byte de memoria.
d	Dummy	10	Direccionamiento del siguiente byte de lectura.
w	Write	11	Escritura de byte en memoria.

Tabla 7. Decodificación de las micro-instrucciones CISCKER

	µ-Instrucción	pass_adder	pc_en	adder_cin	h_reset	mah_en	we	mdr_en
00	**p**	1	1	1	*	1	0	1
01	**r**	1	*	1	0	0	0	1
10	**d**	*	0	0	1	0	0	1
11	**w**	1	*	1	0	0	1	0

10.3.1. Bits de control en las u-instrucciones:

Adicional al código de cada micro-instrucción existen cuatro bits de control que modifican funciones especiales que se deben realizar durante su ejecución:

lookahead: en esta u-instrucción se registra el opcode (depende del modo de direccionamiento)

final: determina cual es la última u-instrucción.

operation: en esta u-instrucción se registra el resultado de la operación y el CCR.

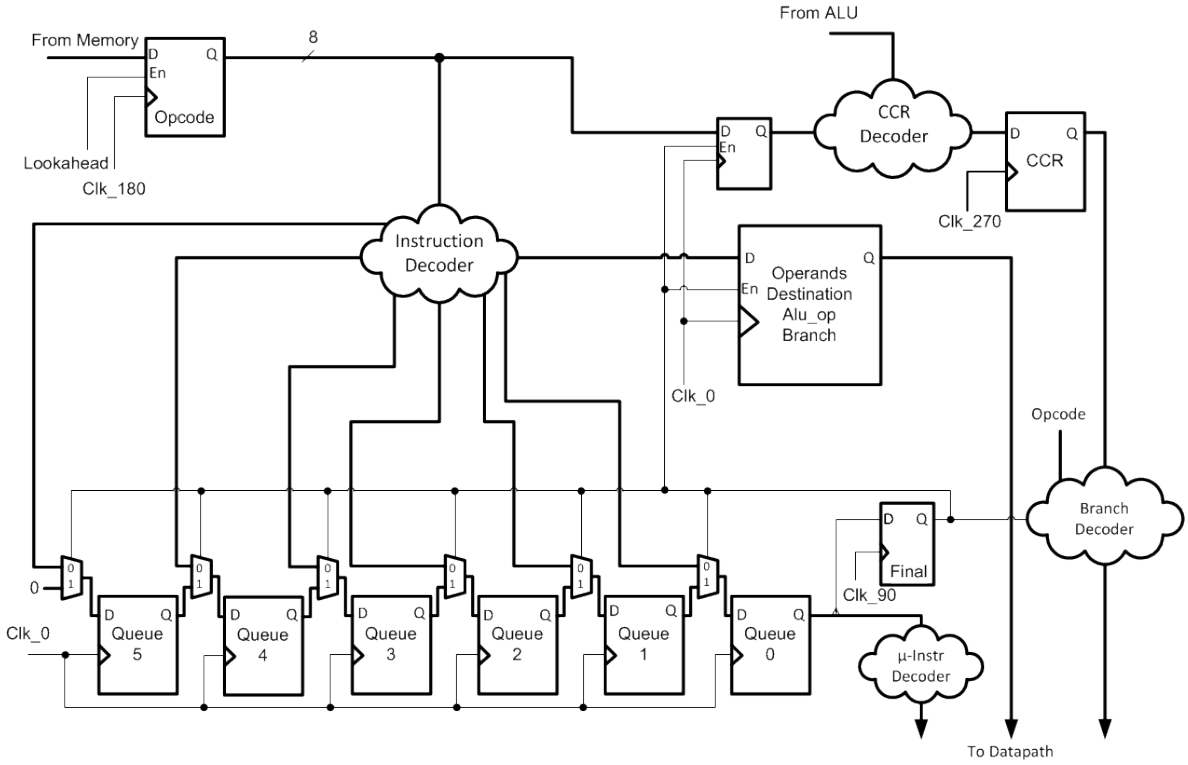
complement: Bit de complemento a la u-instrucción. Determina dos funciones diferentes de una misma u-instrucción. Este bit configura los campos que contienen un asterisco en la tabla 7, permitiendo una mayor versatilidad de las tramas decodificadas.

La Figura 2 es una representación de la MCU CISCKER, en la cual se encuentran los decodificadores de instrucción, CCR, salto condicional y de micro-instrucción. El decodificador de instrucción tiene además el decodificador de la operación de la ALU, el cual retorna los operandos, operación y destino de asignación.

El registro de corrimeinto de seis iteraciones, llamado *Queue*, almacena y permite

la decodificación secuencial de las micro-instrucciones. La señal *Final* indica que se debe cargar una nueva trama de micro-instrucciones cuando termine la ejecución actual teniendo en cuenta que el *Opcode* de la nueva instrucción ya habría sido cargado en el registro gracias a la señal *Lookahead*. En el caso del decodificador del CCR es necesario que el nuevo *Opcode* esté durante la ejecucución de la instrucción, no antes o después como ocurre con la señal *Lookahead (a lo cual debe su nombre)* es por esto que se añade un registro que es habilitado por la señal *Final*.

Figura 2. MCU CISCKER

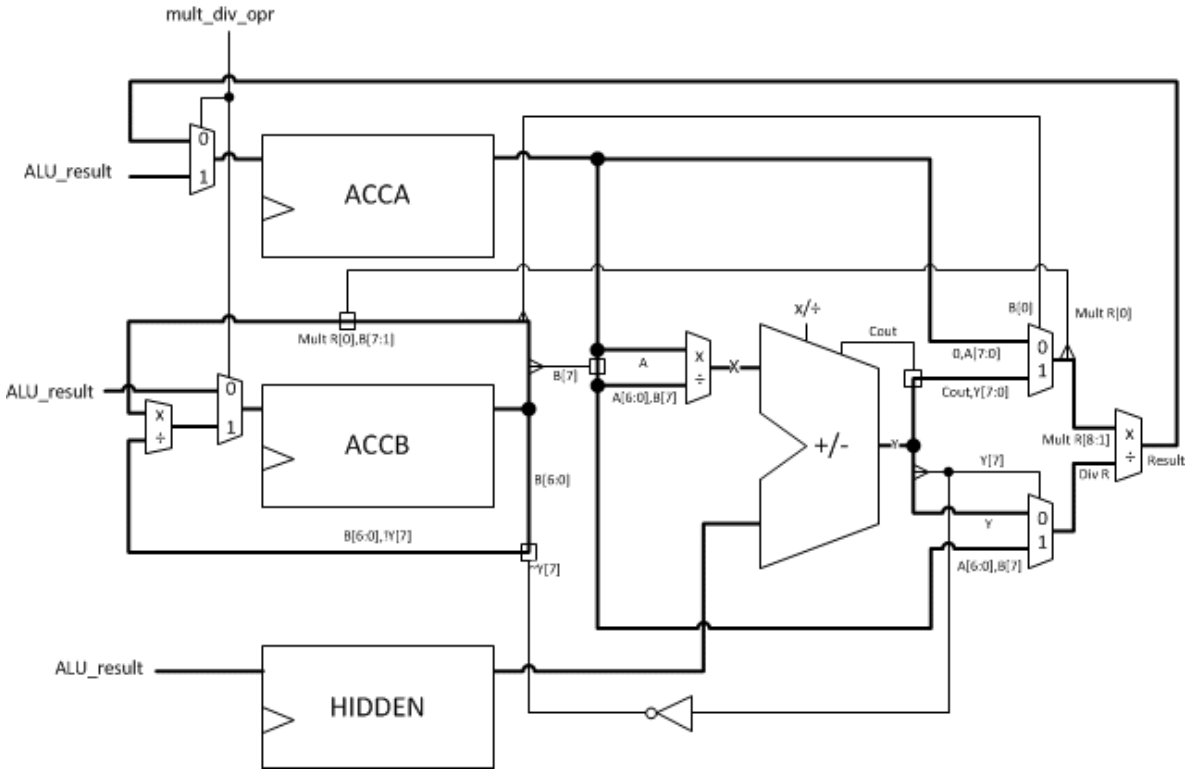


La función de instrucciones iterativas de la MCU CISCKER hace uso de los mismos registros *Queue,* y toma un conteo de la señal *Final* como referencia para conocer cuando y cual iteración ha sido completada. Esto le permite al decodificador cambiar sus salidas sin necesidad de que se haya leído un nuevo *Opcode*. De esta forma es posible ejecutar instrucciones como multiplicación y división, o las de retorno y entrada a la rutina de interrupción.

El circuito que permite la ejecución de instrucciones de multiplicación y división por

iteraciones está representado en el diagrama esquemático de la figura 3. La primera trama de micro-instrucciones se transladar el cociente o el multiplicador, dependiendo de la instrucción, desde el acumulador B al registro *Hidden*, para así poder iniciar la ejecución de las iteraciones que darán como resultado un valor de 16 bits en multiplicación y el cociente y el dividendo de 8 bits en el acumulador D.

Figura 3. Diagrama esquemático del modulo de multiplicación y división



10.4. Unidad de Memoria

En la figura 4 está ilustrada la separación virtual que se hace de la unidad de memoria CISCKER. En esta se indican también los registros mapeados en memoria.

Las direcciones 7FF8 a 7FFF se encuentran reservadas para los vectores de

interrupción. Estos valores deben ser modificados en el caso de habilitar una interrupción para que señale la ubicación de la RSI correspondiente.

7FF6 y 7FF7 vector de la interrupción SWI

7FF8 y 7FF9 vector de la interrupción externa 1

7FFA y 7FFB vector de la interrupción externa 2

7FFC y 7FFD vector de la interrupción por temporizador 1

7FFE y 7FFF vector de la interrupción por temporizador 2

Tabla 4. Distribución de memoria

RAM 0-00FF
Program 0100-X
Data X-X
Stack X-7FF7
Reserved 7FF8-7FFF
MM Registers 8000 Interruption Control 8001 Timer Control 8002 Timer1 Period 8003 Timer1 Period 8004 Timer2 Period 8005 Timer2 Period 8006 Output Port 1 8007 Output Port 2 8008 Input Port 1 8009 Input Port 2 800A Timer1 800B Timer1 800C Timer 2 800D Timer 2
Invalid 800E-FFFF

10.4.1. Registros Mapeados en Memoria (Registros MM)

Los registros MM de la arquitectura CISCKER se describen en la Tabla 6. La función de los bits de los registros de control de interrupciones y temporizadores se encuentra en la Tabla 7. Las direcciones de estos registros comienzan en el valor hexadecimal 8000 donde el bit más significativo los selecciona para escritura y lectura. Es posible agregar nuevos registros MM por encima de la dirección 800D, pero la lectura y escritura de estas direcciones vacías son inválidas.

10.5. Controlador de Interrupciones

Consta de dos partes: La unidad de reconocimiento de interrupciones y el algoritmo que se encarga de almacenar los registros de estado del procesador y saltar a la dirección de la RSI. La figura 5 muestra el circuito de reconocimiento de interrupciones, que entrega a la MCU la señal Interruption y a la ruta de dato, la dirección de donde leer la RSI. En la ruta de datos se carga síncronamente SWI en *Opcode* con la señal Interruption.

SWI pone la máscara de Interrupción en 1, así se deshabilitan las interrupciones y se borra la señal *Interruption* durante la RSI.

Tabla 6. Registros Mapeados en Memoria

Dirección (HEX)	Registro
8000	Control de Interrupciones
8001	Control de Temporizadores
8002	Periodo del Temporizador 1 [15:8]
8003	Periodo del Temporizador 1 [7:0]
8004	Periodo del Temporizador 2 [15:8]
8005	Periodo del Temporizador 2 [7:0]
8006	Puerto de Salida 1
8007	Puerto de Salida 2
8008	Puerto de Entrada 1
8009	Puerto de Entrada 1
800A	Temporizador 1 [15:8]
800B	Temporizador 1 [7:0]
800C	Temporizador 2 [15:8]
800D	Temporizador 2 [7:0]

Tabla 7. Registros MM de Control de temporizadores e interrupciones

Control de Interrupciones		**Control de Temporizadores**	
Bit	**Función**	**Bit**	**Función**
7	Timer1 Prescaler H	**7**	Enable External2 IRQ
6	Timer1 Prescaler L	**6**	Enable External1 IRQ
5	Timer2 Prescaler H	**5**	Enable Timer2 IRQ
4	Timer2 Prescaler L	**4**	Enable Timer1 IRQ
3	Timer1 Enable	**3**	Reset External2 IRQ
2	Timer2 Enable	**2**	Reset External1 IRQ
1	Timer1 Reset	**1**	Reset Timer2 IRQ
0	Timer2 Reset	**0**	Reset Timer1 IRQ

Instrucción SWI: En modo de instrucción de ejecución cíclica, se ejecutan 3 micro-instrucciones 3 veces seguidas por una última trama de 4 micro-instrucciones de objetivo distinto. Estas 3 primeras tramas almacenan en el *Stack los* registros de 16 bits (PC),(X) y (CCR,A). El contador de señales *final* escoge el dato a guardar en la pila correspondientes a las Señales (PC), (X) y (CCR,A) en la ALU con la operación "PASS". La señal (CCR,A) se crea reemplazando la parte alta de "0A" por CCR durante toda la ejecución de la instruccion SWI como se ve en la figura de ruta de datos. Una vez cumplidos los 3 ciclos, la nueva decodificación de SWI dará como resultado 4 micro-instrucciones cuyo objetivo es leer de la dirección donde se encuentra el vector de la interrupción correspondiente y saltar a él. Estas direcciones se encuentran físicamente dentro del procesador, solo podrían ser cambiadas por hardware y son unidas a la ruta de datos mediante un multiplexor a la entrada de PC seleccionado mientras (Opcode==SWI).

Tres micro-instrucciones para almacenar registros:
d: Direcciona SP, y Carga MDR con el valor a guardar (PC, X, CCR, A)
w: Escribe en la memoria el valor L
w: Escribe en la memoria el Valor H y carga PC y MAR con el valor de dirección de la interrupción, esto se hace una vez por cada ciclo de u-instrucciones. (Loop)

Cuatro micro-instrucciones de salto al vector de interrupción:

p: Lee la dirección de interrupción que está en MAR, obteniendo la parte alta del vector de interrupción y la dirección se aumenta para guardarla en PC (El multiplexor de entrada a PC vuelve a su estado normal).
p: En el nuevo valor de dirección en MAR se lee la parte baja del vector de interrupción, completando su valor y guardándolo en MAH.
d: Direcciona MAH para guardar el vector de interrupción en MAR.
r: Lee el Opcode que está en el vector de interrupción, siendo esta la primera instrucción del servicio de interrupción, y deja PC y MAR en el valor siguiente para continuar la ejecución normal del programa.

Figura 5. Controlador de interrupciones CISCKER