DESCRIPCIÓN VHDL DE UNA ARQUITECTURA RISC

Alumnos: Martinez Belot, Luis José Javier
Leyes, Daniel Alejandro
Director: Ing. Horacio Villagarcía Wanza

Trabajo de Grado Licenciatura en Informática
Facultad de Informática - Universidad Nacional de La Plata - Año 2007

AGRADECIMIENTOS

A mi familia, amigos y a todos
aquellos que me dieron su aliento
para continuar durante todo este tiempo
Luis

A mi mujer, padres y amigos
Alejandro

INDICE

Motivación y objetivo del trabajo

En este trabajo de investigación se realizará la descripción de un procesador RISC elemental existente en el mercado en lenguaje VHDL (Very High Speed Integrated Circuit Hardware Description Language), realizando un estudio de tiempo de ejecución de las instrucciones del procesador, análisis del comportamiento y capacidades del mismo mediante la simulación de los módulos descriptos.

Una vez obtenida la descripción del microprocesador se efectuará la compilación y síntesis restringida del procesador descrito en un dispositivo de lógica programable de la familia FLEX 10K de ALTERA incluidos en el University Program Design Laboratory Package.

Además, se realizará un análisis de tiempos de respuesta del procesador, espacio físico utilizado en el dispositivo y eficiencia del mismo, que permita obtener una medición de la fidelidad del procesador descrito.

Desarrollos propuestos

- Descripción de un procesador RISC en lenguaje VHDL.
- Efectuar simulaciones del procesador descrito analizando su funcionamiento y comportamiento.
- Realizar una compilación y síntesis del procesador en un dispositivo de lógica programable.
- Analizar los resultados obtenidos en la simulación y en la síntesis del procesador.

Resultado esperado

Lograr la síntesis en un dispositivo de lógica programable de un procesador elemental descrito en un lenguaje de descripción de hardware.

Conceptos Básicos

INTRODUCCIÓN

¿Qué es una computadora? Mucha gente respondería a esta pregunta diciendo que se trata de una herramienta que sirve para facilitar el trabajo en determinados campos de estudio.

Otros en cambio dirían que se trata de una maquina estupenda para perder el tiempo. Lo cierto es que estos sistemas están en auge en la actualidad. Sin embargo bajo esa apariencia de caja rectangular que ofrece infinidad de posibilidades, se esconde el elemento fundamental e imprescindible que hace la función de "cerebro" del sistema y determina sus características más importantes. Este elemento se denomina PROCESADOR. Físicamente, se trata de un microchip que gobierna el funcionamiento de ese sistema conocido con el nombre de computadora (por ejemplo los microprocesadores de INTEL Pentium).

En este proyecto se pretende diseñar un procesador, elemento principal de las computadoras actuales. Más concretamente, se trata de la implementación en VHDL (Hardware Description Language, un lenguaje específico de descripción de circuitos) de la familia de Microcontroladores denominados PIC de la marca Microchip. Para entender el porqué de esta elección y los objetivos a cubrir en este proyecto, es necesario realizar un estudio teórico de estos sistemas. Gracias a este estudio, será posible definir varios conceptos indispensables, como son las diferentes arquitecturas que se suelen utilizar a la hora de diseñar un procesador, así como otras características y las problemáticas que surgen en el desarrollo de los mismos.

Como se esta viendo los conceptos de computadora y procesador están íntimamente ligados. Una de las vías posibles para comprender qué es un procesador es mediante el desarrollo del concepto computadora. Por ello, en primer lugar se definirá qué es una computadora.

Esto servirá para poder comprender la función que desempeña un procesador en dicho sistema. Se explicarán brevemente los diversos modelos y niveles que se utilizarán para exponer la filosofía que se sigue en el estudio y diseño de los mismos. Después se verá, a grandes rasgos, los dos tipos de arquitectura/modelo usados en el estudio y diseño de procesadores:

- El modelo de Von Neumann.
- La arquitectura Harvard.

A continuación se discutirá qué se entiende por RISC y CISC, y sus principales ventajas e inconvenientes. Por último se estudiará brevemente las diferencias y funcionalidades de un microcontrolador frente a un microprocesador.

Modelos y jerarquías de abstracción para describir a las computadoras

"Una computadora es un sistema complejo que puede describirse mediante diferentes modelos en distintos niveles de una jerarquía de abstracción". Se intentara a continuación explicar esta definición, que servirá para tener una idea general del problema.

Como dice la definición, una computadora se puede modelar de diferentes maneras. Existen modelos funcionales (o de caja negra), estructurales (o de arquitectura) y procesales (estados y transiciones). En este proyecto se hará uso de estos modelos indistintamente.

Por otro lado, la complejidad de las computadoras como sistemas obliga a describirlos en distintos niveles de abstracción (Ver figura 1). En el nivel más bajo (el de menor abstracción) es el de dispositivo. En el se estudian los sistemas electrónicos básicos (transistores, resistencias, condensadores, etc.) que luego serán los componentes del siguiente nivel. En este nivel los componentes son los materiales semiconductores, y el lenguaje para expresar las interrelaciones y los procesos es el de la física del estado sólido.

Los componentes del nivel de circuito electrónico son los sistemas del anterior. Es decir, lo que en terminología electrónica se llaman justamente "componentes": resistencias, transistores, etc. Se utilizan aquí lenguajes gráficos para indicar las interrelaciones o conexiones entre componentes, y para expresar el comportamiento de estos componentes y de los sistemas de este nivel (curvas Voltaje-Intensidad, cronogramas de las evoluciones de las variables, etc.), así como el lenguaje del álgebra para expresar las leyes eléctricas.

En el nivel de circuito lógico se abstrae la estructura de ciertos sistemas construidos en el nivel anterior: las puertas lógicas. Además, y paralelamente se abstrae también la naturaleza física de las variables: ya no se trabaja con valores de voltaje o de corriente, expresadas en voltios o amperios, sino con niveles lógicos. Este nivel puede descomponerse en dos subniveles: el de circuito combinacional y el de circuito secuencial. En efecto, para el diseño de circuitos secuenciales se toman como componentes, sistemas como los biestables, diseñados con puertas. Aparte de lenguajes gráficos para representar los modelos estructurales, se utiliza aquí como lenguaje formal el álgebra de Boole.

El siguiente es ya más característico de las computadoras. Se trata del nivel de micromáquina. Sus componentes son registros (elementos en los que se puede registrar y recuperar una información binaria de longitud fija), operadores aritméticos y lógicos (que permiten transformar una información o combinar varias para dar un resultado predeterminado), unidades de memoria (que almacenan un conjunto de informaciones), secuenciadores (que generan señales secuenciadas en el tiempo para gobernar a otros componentes) y otros circuitos combinacionales y

secuenciales considerados como cajas negras, así como los buses, que permiten combinar a los elementos para formar estructuras.

Las estructuras de este nivel son las llamadas ruta de datos (datapath). Los lenguajes utilizados para descripciones informales son esencialmente gráficos: diagramas y cronogramas, aunque existen también lenguajes formalizados (como por ejemplo VHDL).

En el nivel de máquina convencional, el modelo funcional de una computadora está compuesto por su repertorio de instrucciones y los convenios de representación de la información (instrucciones y datos). La representación es siempre en binario (cadenas de unos y ceros), lenguaje predominante de este nivel. Combinando instrucciones (es decir, programando) se construyen sistemas de niveles superiores: los modelos estructurales aquí se llaman configuraciones; sus componentes son los mismos de las rutas de datos pero a un mayor nivel de abstracción (por ejemplo, la "unidad central de proceso" que engloba a la unidad aritmético-lógica y a la unidad de control y hace una abstracción de los registros par que sean "invisibles para el programador").

El nivel de máquina operativa (se corresponde con una computadora acompañada de un "sistema operativo") surge como necesidad de "arropar" a la "computadora desnuda" (maquina convencional) con programas de uso general que lo hagan más fácil de utilizar y que optimicen su funcionamiento. El modelo funcional de una computadora en esta máquina incluye al modelo funcional del nivel de máquina y lo amplia con "instrucciones virtuales" (esencialmente con llamadas al sistema operativo). Este nivel y el siguiente son ya niveles de maquinas virtuales, en el sentido de que la funcionalidad del sistema se obtiene mediante una combinación de hardware y de software.

Si solo dispusiésemos de los niveles anteriores la programación de las computadoras sería una ardua tarea, porque el lenguaje es binario. Por ello, se inventaron los procesadores de lenguajes (ensamblador (traductor), intérprete de Basic, compilador de C, etc.), y con ellos apareció un nuevo nivel: el de máquina simbólica.

Figura 1

El modelo de Von Neumann

Un hito histórico importante en la historia de las computadoras fue la introducción del concepto de programa almacenado: previamente a su ejecución, las instrucciones que forman el programa deben estar almacenadas en memoria. Ello condiciona el funcionamiento y la estructura de estas máquinas.

El modelo estructural básico formado por unidades de memoria de acceso directo (o acceso aleatorio, RAM), procesamiento, control y entrada/salida, junto con la idea central de "programa almacenado" se debe a John Von Neumann. Básicamente el modelo habla de las partes de que debe constar una máquina computadora de propósito general y qué funciones desempeña cada una. Von Neumann distingue las siguientes partes (Ver figura 2):

- Memoria Principal (M.P.)
- Unidad Central de Proceso (C.P.U.), que se compone de:
  - Unidad Aritmético Lógica (A.L.U)
  - Unidad de Control (U.C)
- Unidad/es de Entrada-Salida (U-E/S)

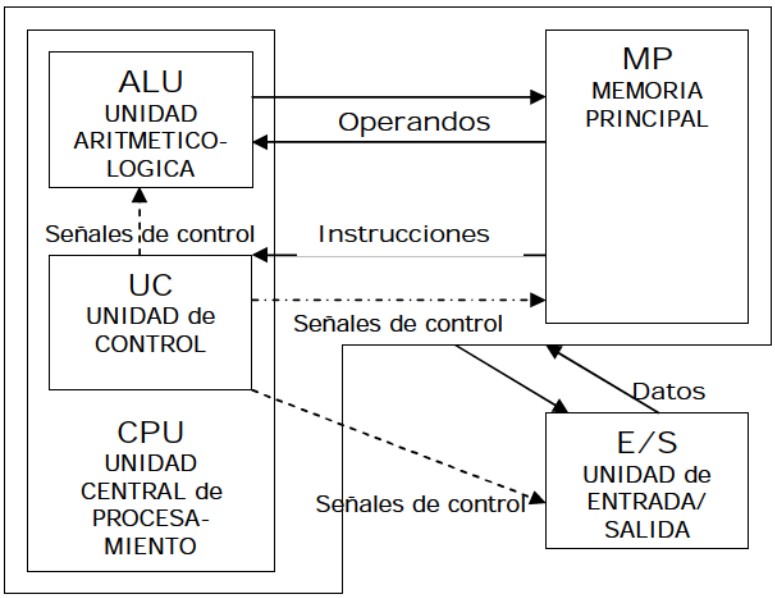


Figura 2

A todo esto, hay que sumar el hecho de que todas las máquinas computadoras actuales son esencialmente sincrónicas. Este hecho hace que cualquier acción o conjunto de acciones (u órdenes) en el sistema, se realizará en uno o varios ciclos de los que se denomina ciclo de reloj del sistema (a partir de ahora lo denominaremos CLK).

Memoria principal: Programa almacenado

La máquina debe ser capaz de almacenar no solamente la información digital necesaria en una determinada computación, sino también las instrucciones que gobiernen la rutina que realmente hay que llevar a cabo con los datos numéricos. Se hace necesario por tanto distinguir entre dos tipos de memoria en el diseño:

- Memoria de programa: Donde se almacenarán las instrucciones (u órdenes) que conforman el programa.
- Memoria de datos: En la que se almacenarán los resultados, datos para la ejecución del programa, etc.

Von Neumann agrupa estos dos tipos de memoria en lo que se conoce como MEMORIA PRINCIPAL. Como se verá mas adelante esta es la principal diferencia entre esta arquitectura y la arquitectura de Harvard.

Por otro lado la memoria debe ser de acceso directo para lectura y escritura. Los tiempos de acceso a la memoria deben ser lo más pequeño posible. El elemento mínimo de almacenamiento en memoria es el bit (0 ó 1). Pero éstos se agrupan formando lo que se conoce con el nombre de "palabra" (word), de modo que en cada acceso, lo que escribiremos (ciclo de escritura) o leeremos (ciclo de lectura) en la Memoria Principal será una palabra de N bits. Para acceder a una palabra de la memoria debemos conocer su posición. La "dirección" identifica la posición de la palabra de memoria. Será un número comprendido entre 0 y M-1, donde M es la capacidad de la memoria (número de palabras). Ver figura 3.

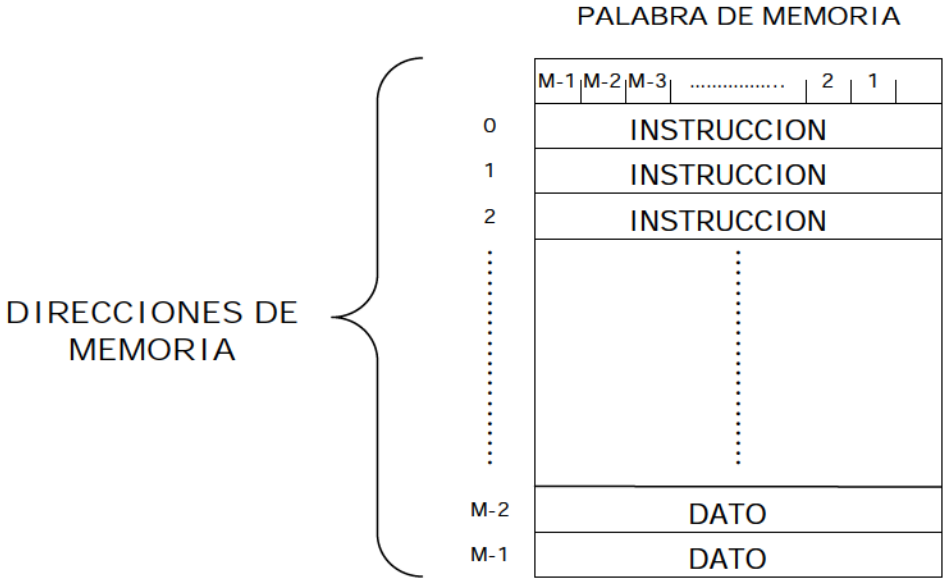


Figura 3

Para extraer una palabra de la memoria (ciclo de lectura) se introduce su dirección y, tras un tiempo de acceso para la lectura, se podrá

tener como salida el contenido de esa posición Para introducir una palabra en una posición (ciclo de escritura), se indicará también la dirección y, tras un tiempo de acceso para la grabación, tendremos la información de entrada registrada. Ver figura 4.

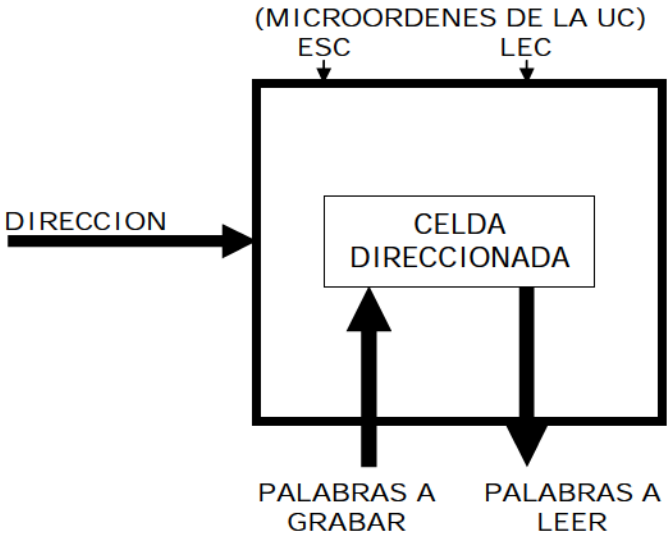


Figura 4

Unidad central de proceso

Es el "cerebro" de lo que hemos definido como máquina computadora. Como se ha dicho consta de:

Unidad aritmético-lógica (ALU)

Puesto que el dispositivo va a ser una máquina calculadora, tiene que contener un órgano que pueda realizar operaciones aritméticas elementales (suma, resta, multiplicación, división). También serán necesarias operaciones de tipo lógico (AND, OR, NOT, etc.).

La ALU es un subsistema que puede tomar dos operandos (o solo uno, en el caso de "NOT", por ejemplo) y generar el resultado correspondiente a la operación que se le indique, de entre un conjunto de operaciones previstas en su diseño.

Por otra parte, hay que hacer mención, del "compromiso Hardware/Software", muy importante a la hora de hacer realidad cualquier máquina computadora, como la llama Von Neumann. Muchas funciones pueden realizarse de manera "cableada" (por "Hardware") o de manera "programada" (por "Software"). La elección depende de que predomine el deseo de reducir el costo (soluciones basadas en software) o de conseguir una maquina más rápida (soluciones basadas en hardware). Existe también una solución intermedia: la realización microprogramada (o microcódigo).

Unidad de control

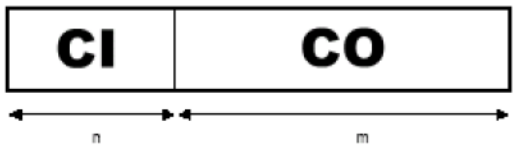
La unidad de control examina las instrucciones ("ordenes") almacenadas en la memoria y genera las señales ("micro órdenes") precisas para que las otras unidades ejecuten lo que indica la instrucción.
Normalmente, al terminar con una instrucción, la siguiente a ejecutar está almacenada en la memoria en la dirección siguiente a ella (las excepciones son las instrucciones de bifurcación, que comentaremos enseguida).

Unidades de entrada/salida

Tiene que haber unos dispositivos que constituyen el sistema de entrada y de salida, mediante los cuáles el usuario y la máquina puedan comunicarse entre sí. Este sistema puede considerarse como una forma secundaria de memoria automática. Las unidades de E/S son realmente bastante complicadas, porque incluyen no sólo los llamados "periféricos" (teclado, pantalla, terminales remotos, impresoras, etc.), sino también las memorias secundarias (cintas, discos, etc.), así como subsistema de control específicos para ellos.

Formato de instrucción

Se hace mención al formato de instrucción propuesto por Von Neumann, que es una buena manera de introducir el concepto de instrucción. La Figura muestra el formato de instrucción que propuso. Posee dos partes o campos:



- El código de instrucción (CI), que indica la instrucción de que se trata. Si usamos n bits podremos tener 2n instrucciones posibles.
- El código de operando (CO), la dirección de la Memoria Principal a la que hace referencia la instrucción. Si tenemos M bits para este campo. Podremos direccionar 2^M posiciones de memoria.

Programa

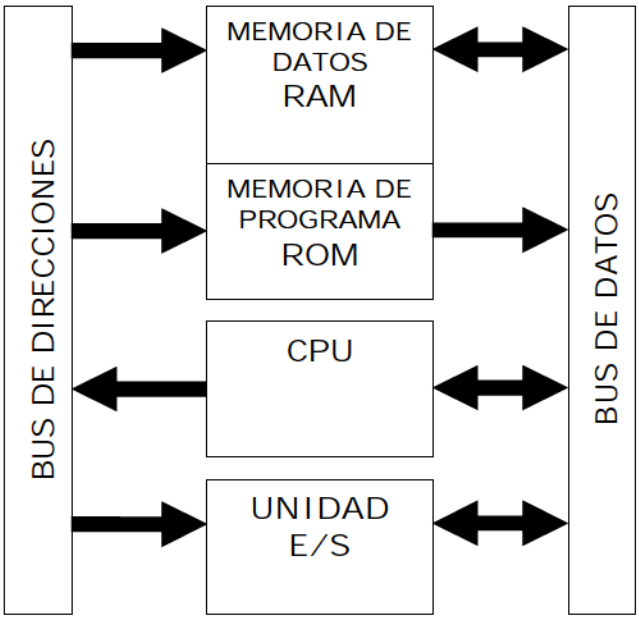
La "programación" es una actividad que se realiza en el nivel de máquina convencional y en el nivel de máquina simbólica y que consiste en hacer uso del modelo funcional para "construir" programas que permiten definir una maquina de nivel superior.

El modelo funcional de una máquina convencional o simbólica esta definida por su "lenguaje", que es en este nivel es lenguaje máquina.

Los programas constan de una secuencia de instrucciones que se almacenan consecutivamente en la memoria. Normalmente, tras la ejecución de una instrucción se pasa a ejecutar la siguiente. Pero en ocasiones hay que pasar no a la instrucción siguiente, sino a otra, almacenada en otra dirección. Para poder hacer tal cosa están las "instrucciones de bifurcación" o "instrucciones de salto".

La arquitectura Harvard

Una vez expuesta el modelo de Von Neumann, cabe ahora destacar en primer lugar que se entiende por arquitectura de Von Neumann, para poder así compararla con la arquitectura de Harvard (Ver figura 5).



Arquitectura simplificada de un procesador de tipo Von Neumann.
Figura 5

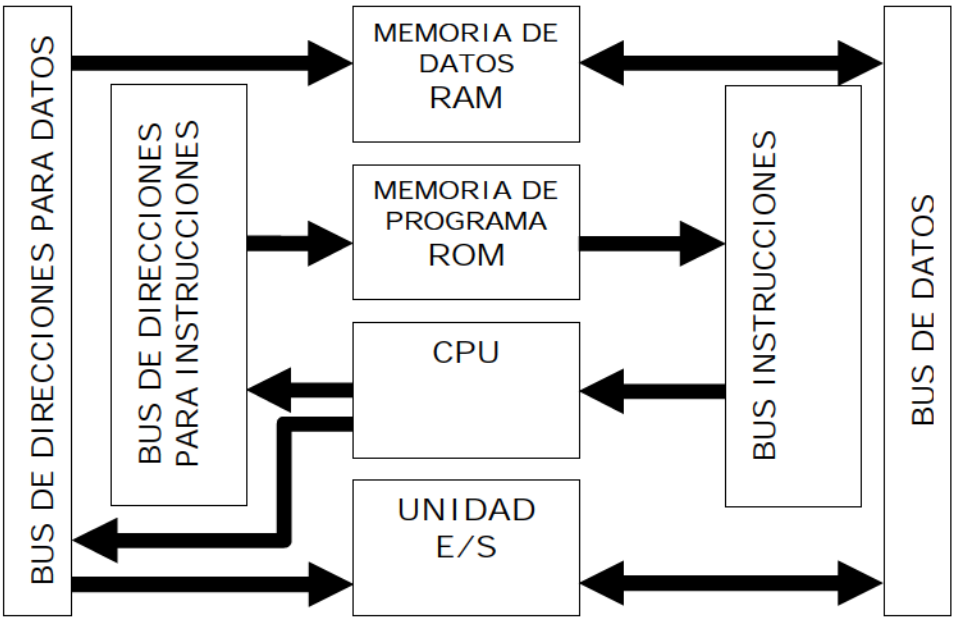
En las arquitecturas de tipo Von Neumann, las instrucciones y los datos son almacenadas en memorias (de tipo RAM o de tipo ROM) a las que accede secuencialmente la CPU a través de un único bus de direcciones y de datos. La CPU lee las instrucciones de la memoria mediante el bus de datos y seguidamente ejecuta las instrucciones leídas previamente. El sistema procesador puede además leer y escribir datos en dispositivos externos mediante un buffer de entrada/salida, de tal forma que el acceso a periféricos externos es similar al acceso a la memoria.

Normalmente, en éste tipo de estructuras microprocesadores, programas y datos residen en la misma memoria. La memoria de programa y la de datos es accedida por el mismo bus de direcciones de forma que las

instrucciones de programa y los datos fluyen por un mismo camino, el bus de datos. Estos sistemas acceden a la memoria para recoger la instrucción a ejecutar.

Posteriormente decodifica la instrucción para determinar la secuencia de operaciones que ésta lleva asociada y vuelve a acceder a la memoria de datos para leer los operandos asociados a la instrucción leída. Finalmente se ejecuta y comienza un nuevo ciclo.

En la arquitectura tipo Harvard el bus de direcciones de acceso a la memoria de programa es diferente al bus de direcciones de acceso a datos. Las instrucciones son recibidas por la CPU a través de un bus de datos reservado para las instrucciones, separado del bus de datos propiamente dicho. La CPU lee las instrucciones de la memoria de programa mediante el bus de datos para instrucciones y paralelamente ejecuta las instrucciones leídas previamente (esta estructura de acceso a datos y ejecución de instrucciones en paralelo, se denomina en terminología inglesa como pipe-lining). Se realiza un acceso simultáneo a la memoria de programa y de datos (posible por la duplicidad de recursos, buses, que existen en el sistema). Ver figura 6.



Arquitectura simplificada de un procesador de tipo Harvard.
Figura 6

Este tipo de estructura es más compleja en el hardware que las de tipo Von Neumann, pero permiten acelerar el tiempo efectivo de ejecución de la instrucción: la CPU prepara (fetch) instrucciones de programa mientras y simultáneamente realiza la manipulación de datos de las instrucciones previamente recogidas. La ejecución real de una instrucción puede que dure varios ciclos de reloj, pero mientras una instrucción se esta ejecutando, otra instrucción/es están en proceso de preparación para

ejecutarse. Esto es posible gracias a que con la estructura Harvard, cada memoria tiene su respectivo bus, lo que permite que la CPU pueda acceder de forma independiente y simultánea a la memoria de datos y a la de instrucciones. Como los buses son independientes éstos pueden tener distintos contenidos en la misma dirección.

Ventajas de esta arquitectura:

1°. El tamaño de las instrucciones no esta relacionado con el de los datos, y por lo tanto puede ser optimizado para que cualquier instrucción ocupe una sola posición de memoria de programa, logrando así mayor velocidad y menor longitud de programa.

2°. El tiempo de acceso a las instrucciones puede superponerse con el de los datos, logrando una mayor velocidad en cada operación.

Una pequeña desventaja de los procesadores con arquitectura Harvard, es que deben poseer instrucciones especiales para acceder a tablas de valores constantes que pueda ser necesario incluir en los programas, ya que estas tablas se encontraran físicamente en la memoria de programa (por ejemplo en la EPROM de un microprocesador).

RISC y CISC

El diseño de un repertorio de instrucciones es una tarea difícil, especialmente en un entorno industrial competitivo, en el que los fabricantes tienen que resolver múltiples compromisos sobre costos de desarrollo, plazos de fabricación, costo y eficacia del producto.

Este proyecto pretende estudiar un caso concreto ya existente, en el que este diseño está ya cubierto, por lo que no es lugar para entrar sobre consideraciones de diseño, pero si se va a exponer una tendencia arquitectónica reciente. Para entenderla, hay que verla con una perspectiva histórica.

Durante los años 60, 70 y 80 la evolución de las arquitecturas se caracterizó por un enriquecimiento del repertorio de instrucciones, tanto en cantidad (lo normal es que una computadora tenga más de 100) como en modos de direccionamiento. Esto obedecía a razones prácticas: cuantas más posibilidades ofrezca el nivel de maquina convencional más fácil será desarrollar niveles superiores (sistemas operativos, compiladores, aplicaciones, etc.). Y ello a pesar de que muchos estudios sobre estadística de uso de instrucciones durante la ejecución de los programas demostraba que gran parte de tales instrucciones apenas se utilizaban.

Esto condujo a algunos investigadores a plantear una alternativa de diseño diferente: si decidimos reducir a un tamaño mínimo (eliminar todas las que tengan un porcentaje muy bajo de utilización) el número de instrucciones tendremos programas más largos y, en principio, más lentos en su ejecución. Pero la unidad de control podrá ser mucho más sencilla, y

su implementación en los niveles inferiores (desde el de micromáquina hasta el de dispositivo) podrá, para el mismo espacio físico (o sea, superficie de circuito integrado), ser mucho más eficaz. O sea, que quizás el resultado final es que la máquina no resulte tan "lenta". De acuerdo con esta idea, a una computadora cuyo diseño sigue la tendencia "tradicional se le llama "CISC" (Complex Instruction Set Computer), por contraposición a "RISC" (Reduced Instruction Set Computer).

Otra razón que justifica el auge inicial que tuvo los diseños CISC´s, es que antiguamente la memoria era escasa (muy cara). Por lo que este tipo de sistemas primaba las instrucciones que reducían el número de accesos a memoria.

En realidad, los RISC's se caracterizan por más detalles, entre ellos los siguientes:

- Relativamente pocas instrucciones y modos de direccionamiento.
- Arquitectura "load/store". Esto quiere decir que los accesos a la Memoria Principal son para extraer instrucciones y datos y para almacenar datos. Todas las operaciones de procesamiento se realizan en registros de la CPU.
- Formato de instrucciones similar: todas las instrucciones tienen la misma longitud (normalmente, 32 bits) y todos los bits tienen el mismo significado en todas las instrucciones. Esto simplifica extraordinariamente la unidad de control.
- Proporcionalmente suelen poseer gran cantidad de registros de propósito general, en comparación con sistemas basados en CISC.

Microprocesador y microcontrolador

La principal diferencia entre microcontrolador y microprocesador se encuentra en la funcionalidad de los mismos, ya que las características que coinciden en ambos superan a las diferencias. Su estructura interna es similar, por no decir idéntica. Ambos siguen los modelos antes descriptos y poseen los componentes típicos de un procesador (CPU, Memoria Principal, Unidades de E/S, y Buses).

Los microprocesadores son unos sistemas que suelen formar parte a su vez en otros sistemas denominados computadoras (con teclado, monitor, ratón CD-ROM, placa base). Sobre el se ejecutan programas. Y sobre estos programas se ejecutan otros programas, así sucesivamente hasta llegar a conformar lo que se conoce como sistema operativo.

Este S.O. convierte a la computadora en una máquina de propósito general donde se pueden ejecutar diversas aplicaciones de una manera cómoda para el usuario (procesadores de texto, compiladores de C, gestores de tráfico de red, programas de dibujo, etc.).

Sin embargo un microcontrolador como su nombre indica está destinado únicamente a controlar un sistema determinado, o una parte del mismo. Una vez que se carga un programa en su memoria que se suele hacer a través de una computadora (suele ser un programa pequeño que requiere pocos registros para su funcionamiento), el microcontrolador tiene un único objetivo, ejecutarlo para controlar así el sistema donde este incluido. Se intenta siempre reducir el número de componentes en las soluciones microcontroladas. En determinados casos aparece únicamente el microcontrolador, y a lo sumo algún componente más. Por todo esto, normalmente poseen un bus de direcciones menor e incluye en el mismo chip algunos periféricos que en un microprocesador estarían fuera del mismo.

Un microcontrolador es un dispositivo complejo, formado por otros más sencillos. A continuación se analizan los más importantes.

Procesador

Es la parte encargada del procesamiento de las instrucciones. Debido a la necesidad de conseguir elevados rendimientos en este proceso, se ha desembocado en el empleo generalizado de procesadores de arquitectura Harvard frente a los tradicionales que seguían la arquitectura de Von Neumann.

El procesador de los modernos microcontroladores responde a la arquitectura RISC (Computadores de Juego de Instrucciones Reducido), que se identifica por poseer un repertorio de instrucciones máquina pequeño y simple, de forma que la mayor parte de las instrucciones se ejecutan en un ciclo de instrucción.

Otra aportación frecuente que aumenta el rendimiento del computador es el fomento del paralelismo implícito, que consiste en la segmentación del procesador (pipe-line), descomponiéndolo en etapas para poder procesar una instrucción diferente en cada una de ellas y trabajar con varias a la vez.

Memoria de programa

El microcontrolador está diseñado para que en su memoria de programa se almacenen todas las instrucciones del programa de control. Como éste siempre es el mismo, debe estar grabado de forma permanente. Existen algunos tipos de memoria adecuados para soportar estas funciones, de las cuales se citan las siguientes:

- ROM con máscara: se graba mediante el uso de máscaras. Sólo es recomendable para series muy grandes debido a su elevado coste.

- EPROM: se graba eléctricamente con un programador controlado por una PC. Disponen de una ventana en la parte superior para someterla a luz ultravioleta, lo que permite su borrado. Puede usarse en fase de diseño, aunque su costo unitario es elevado.

- OTP: su proceso de grabación es similiar al anterior, pero éstas no pueden borrarse. Su bajo costo las hacen idóneas para productos finales.

- EEPROM: también se graba eléctricamente, pero su borrado es mucho más sencillo, ya que también es eléctrico. No se pueden conseguir grandes capacidades y su tiempo de escritura y su consumo son elevados.

- FLASH: se trata de una memoria no volátil, de bajo consumo, que se puede escribir y borrar en circuito al igual que las EEPROM, pero que suelen disponer de mayor capacidad que estas últimas. Son recomendables para aplicaciones en las que es necesario modificar el programa a lo largo de la vida del producto. Por sus mejores prestaciones, está sustituyendo a la memoria EEPROM para contener instrucciones.

Memoria de datos

Los datos que manejan los programas varían continuamente, y esto exige que la memoria que los contiene debe ser de lectura y escritura, por lo que la memoria RAM estática (SRAM) es la más adecuada, aunque sea volátil.

Hay microcontroladores que disponen como memoria de datos una de lectura y escritura no volátil, del tipo EEPROM. De esta forma, un corte en el suministro de la alimentación no ocasiona la pérdida de la información, que está disponible al reiniciarse el programa.

Línea de E/S

A excepción de dos patitas destinadas a recibir la alimentación, otras dos para el cristal de cuarzo, que regula la frecuencia de trabajo, y una más para provocar el Reset, las restantes patitas de un microcontrolador sirven para soportar su comunicación con los periféricos externos que controla.

Las líneas de E/S que se adaptan con los periféricos manejan información en paralelo y se agrupan en conjuntos de ocho, que reciben el nombre de Puertas. Hay modelos con líneas que soportan la comunicación en serie; otros disponen de conjuntos de líneas que implementan puertas de comunicación para diversos protocolos, como el I2C, el USB, etc.

Recursos auxiliares

Según las aplicaciones a las que orienta el fabricante cada modelo de microcontrolador, incorpora una diversidad de complementos que refuerzan la potencia y la flexibilidad del dispositivo. Entre los recursos más comunes se citan los siguientes:

- Circuito de reloj: se encarga de generar los impulsos que sincronizan el funcionamiento de todo el sistema.

- Temporizadores, orientados a controlar tiempos.

- Perro Guardián o WatchDog: se emplea para provocar una reinicialización cuando el programa queda bloqueado.

- Conversores AD y DA, para poder recibir y enviar señales analógicas.

- Sistema de protección ante fallos de alimentación

- Estados de reposos, gracias a los cuales el sistema queda congelado y el consumo de energía se reduce al mínimo.

Programación de microcontroladores

La utilización de los lenguajes más cercanos a la máquina (de bajo nivel) representan un considerable ahorro de código en la confección de los programas, lo que es muy importante dada la estricta limitación de la capacidad de la memoria de instrucciones. Los programas bien realizados en lenguaje Assembler optimizan el tamaño de la memoria que ocupan y su ejecución es muy rápida.

Los lenguajes de alto nivel más empleados con microcontroladores son el C y el BASIC de los que existen varias empresas que comercializan versiones de compiladores e intérpretes para diversas familias de microcontroladores.

Hay versiones de intérpretes de BASIC que permiten la ejecución del programa línea a línea, y en ocasiones, residen en la memoria del propio microcontrolador. Con ellos se puede escribir una parte del código, ejecutarlo y comprobar el resultado antes de proseguir.

Qué es un PLD (Dispositivo de Lógica Programable)

Un dispositivo lógico programable es un circuito integrado, formado por una matriz de puertas lógicas y flip-flops, que proporcionan una solución al diseño de formas análogas, a las soluciones de suma de productos, productos de sumas y multiplexores.

La estructura básica de una PLD permite realizar cualquier tipo de circuito conbinacional basándose en una matriz formada por puertas AND, seguida de una matriz de puertas OR. Tres son los tipos más extendidos de PLD's, la PROM, PLA, y la PAL.

- PROM (Programmable Read Only Memory)

Este tipo de dispositivo se basa en la utilización de una matriz AND fija, seguida de una matriz OR programable. La matriz programable esta formada por líneas distribuidas en filas y columnas en las cuales los puntos de cruce quedaran fijos por unos diodos en serie con unos fusibles que serán los encargados de aislar las uniones donde no se requiera la función lógica.

La fase de programación se realiza haciendo circular una corriente capaz de fundir el fusible en aquellas uniones donde no se desee continuidad. Por otra parte, para cada combinación de las señales de

entrada, el codificador activa una única fila y a su vez activa aquella columna a las que esta todavía unida a través del diodo.

- PLA (Programmable Logic Array)

Parecido en la dispositivo a la PROM, difiere de esta, en que aquí en la PLD, ambas matrices, la de puertas And, así como la de puertas Or es programable, por lo que nos vemos habilitados a incrementar el número de entradas disponibles, sin aumentar el tamaño de la matriz. Esta estructura permite una mejor utilización de los recursos disponibles en el circuito integrado, de tal forma que se genera el mínimo número de términos necesarios para generar una función lógica.

- PAL (Programmable array Logic)

Una PAL es diferente de una PROM a causa de que tiene una red Y programable y una red O fija. Con un programador Prom podemos obtener los productos fundamentales deseados quemando los eslabones y luego conseguir la suma lógica de dichos productos mediante las conexiones fijas de salida.

Tendencia actual

En la actualidad los sistemas a desarrollar han aumentado su complejidad y cada vez se utilizan con mayor frecuencia procesadores o microcontroladores en hardware para realizar tareas específicas o donde el tiempo de respuesta es crítico. Por otro lado los tiempos de desarrollo se han visto reducidos lo que ha conducido a la búsqueda de soluciones que permitan responder a los tiempos impuestos por el mercado.

Dentro del marco de la lógica programable la existencia de dispositivos programables cada vez más poderosos ha llevado a las empresas líderes a programar soluciones SOC (System On Chip) existiendo para ello diferentes lenguajes que permiten la descripción de dispositivos y su posterior compilación y síntesis en chips programables. Además ante la posibilidad de tener librerías con bloques prediseñados y verificados que pueden ser reutilizados se logra una reducción considerable en el tiempo de desarrollo de un sistema.

La complejidad del hardware ha crecido de modo sostenido en los últimos años a punto tal que es posible diseñar -y fabricar- computadoras digitales completas (CPU, Memoria y E/S) o colocar múltiples procesadores en un único chip de silicio con diversas soluciones tecnológicas. Así mismo, los dispositivos de Lógica Programable de alta densidad de integración que se puede adquirir en el mercado, permiten buscar soluciones integradas en un dispositivo SOPC (System On a Programmable Chip).

Cuando se desarrolla un dispositivo comercial se busca la mejora en la calidad de los diseños obtenidos y el incremento de la productividad. Además la necesidad de integrar sistemas cada vez más complejos en dimensiones más reducidas, y con tiempos de desarrollo menores ha

llevado a tener que desarrollar técnicas y metodologías de trabajo que permitan obtener una mayor productividad.

Este objetivo necesario ha dado lugar a un desarrollo cada vez más orientado al diseño de módulos que puedan ser reutilizados, con el fin de alcanzar las metas, no para diseñar más rápido sino para diseñar menos.

Esto ha motivado al mercado a la utilización de SOC (System On a Chip), soluciones que integren en un único chip todo un sistema. Las nuevas metodologías de diseño basadas en SOC se fundamentan en la existencia de librerías con bloques pre-diseñados y pre-verificados denominados "bloques IP (Intellectual Property)". La reusabilidad de los bloques IP es uno de los tópicos que facilita la creación de un nuevo SOC.

Los criterios de diseño utilizados para definir la arquitectura de un sistema, dependen del grado de conocimiento que se tenga de los problemas a resolver:

- Si las tareas son desconocidas, y pueden ser de tipo muy variado, la solución mas razonable es el empleo de un procesar de propósito general.

- Si las tareas son de un tipo especifico pero aún indefinido (por ejemplo, tratamiento de imágenes), dicha elección suele orientarse a un procesar especializado (tal como un DSP) con amplios recursos adicionales (memoria, I/O, periféricos).

- Si la aplicación es totalmente conocida, el diseño puede hacerse "a medida", usando los recursos de hardware mas apropiados para esa aplicación, y según el caso, es posible pensar en el diseño de un ASIC (Application Specific Integrated Circuit).

Para poder encarar soluciones programables por software, existen varias empresas que han comenzado a ofrecer productos donde dentro de un chip conviven un único procesador, un sistema operativo de tiempo real (RTOS) con capacidad de multitarea, y un conjunto de recursos programables aptos para distintos aplicaciones:

ATMEL ofrece un procesador RISC de 8 bits (AVR), con abundante RAM y ROM de programa y bloque programable de entre 10K y 40K compuertas.

TRISCEND ofrece un ARM7DMI de 32 bits, con memorias cache internas, interfase a memorias externas, periféricos (timers, UARTs, interrupts) y una conexión a una matriz programable de complejidad equivalente a 40K compuertas.

ALTERA ofrece una alternativa Softcore llamada NIOS, configurable a 16 o 32 bits de ancho de datos, que puede ser sintetizado en chips de la familias APEX, FLEX o ACEX. Como alternativa Hardcore, dentro de la

familia Excalibur, se ofrecen también tres modelos de ARM922T y 3 modelos de MIPS32 4Kc, que difieren entre sí por la capacidad de memoria y de lógica programable interna.

XILINX posee una alternativa Softcore llamada MicroBLAZE, de 32 bits, que puede ser sintetizado en la familia VirtexII, con UART, timer, entrada/salida paralela, controlador de interrupciones, árbitro multimaster, e interfaces memoria FLASH, y distintos tipos de RAM.

Todas estas soluciones se basan en un único procesador muy poderoso (salvo el AVR, todos los otros casos son de 32 bits), periféricos propios, y recursos de interconexión con la lógica programable; internamente los procesadores poseen buses de alta performance (100 MHz o superiores) usando formato estándar tipo AMBA (ARM), CoreConnect (IBM), Wishbone (SILICORE), o soluciones propias. Para el desarrollo de sistemas todas las empresas ofrecen paquetes de diseño que combinan las herramientas EDA propias del diseño lógico con compiladores y depuradores del software, y poderosos sistemas operativos multitareas de tiempo real (RTOS).

El concepto de un procesador RISC esta basado en la idea de un conjunto de instrucciones básico y pequeño en conjunción con un compilador inteligente que puede entregar una performance superior a la otorgada por un procesador CISC, basado en un conjunto complejo de instrucciones, con un número grande de instrucciones especializadas. La simplicidad de las operaciones permite que cada instrucción sea ejecutada eficientemente en un ciclo de procesador.

PORQUE UNA SOLUCIÓN SOPC

En la actualidad los dispositivos Lógica Programable superan el millón de compuertas, por lo tanto las soluciones integradas System on Programmable Chip (SOPC) pueden utilizar estos dispositivos como una excelente opción para su implementación.

Algunas compañías están ofreciendo soluciones soft-core para procesadores propietarios (NIOS) o estándar (8051 y otros); también se ofrecen soluciones hard-code para el caso de aplicaciones intensivas, con procesadores de alta performance pregrabados (como MIPS32_4Kc o ARM922) embebidos junto con grandes bloques de memoria y lógica programable.

Un core FPL (Field Programmable logic) es una lógica de fabricación flexible que puede ser customizada para implementar cualquier circuito digital luego de la fabricación. FPL conduce a un nuevo paradigma de diseño, agregando gran flexibilidad al proceso de diseño:

- Selección del procesador: cuando se utilizan soluciones soft-core, el procesador puede ser seleccionado acorde a las necesidades de la aplicación.

- Recursos del procesador: el diseñador puede definir que memoria y periféricos deben ser embebidos. Esta característica provee un uso eficiente del silicio, y la posibilidad de agregar periféricos estándar o especiales si fuera necesario.

- IP (Intellectual Property): el reuso de un módulo existente validado, que puede ser considerado como una biblioteca de bloques con una implementación dada que puede reducir dramáticamente el ciclo de diseño.

- Re-configuración: Usando lógica programable, un SOPC puede ser desarrollado antes de que algunas especificaciones sean definidas, o modificado a último momento. En algunos casos, la re-configuración puede ser usada como una actualización post-venta o una característica de customización.

Esto es importante para entender que cualquier diseño SOPC involucra un conjunto complejo de decisiones de diseño hardware-software, teniendo en cuenta el costo del producto, la flexibilidad del mismo y las restricciones que puedan existir dependiendo del escenario donde debe desarrollarse el proyecto.

VHDL

Introducción

El lenguaje de descripción de hardware VHDL (VHSIC HDL, Very High Speed Integrated Circuit Hardware Description Language) permite diseñar, modelar y comprobar un sistema digital con distintos niveles de abstracción. Los módulos creados se podrían re-utilizar en diversos diseños, incluso una misma descripción en diferentes tecnologías de implementación. En estos casos se requiere analizar que pautas específicas se deben aplicar para que el encapsulamiento de los módulos no modifique sus propiedades ni comportamientos.

Reseña histórica

En 1983, IBM, Intermetrics y Texas Instruments empezaron a trabajar en el desarrollo de un lenguaje de diseño que permitiera la estandarización, facilitando con ello, el mantenimiento de los diseños y la depuración de los algoritmos, para ello el IEEE propuso su estándar en 1984.

Tras varias versiones llevadas a cabo con la colaboración de la industria y de las universidades, que constituyeron a posteriori etapas intermedias en el desarrollo del lenguaje, el IEEE publicó en diciembre de 1987 el estándar IEEE std 1076-1987 que constituyó el punto firme de partida de lo que después de cinco años sería ratificado como VHDL.

Esta doble influencia, tanto de la empresa como de la universidad, hizo que el estándar asumido fuera un compromiso intermedio entre los lenguajes que ya habían desarrollado previamente los fabricantes, de manera que éste quedó como ensamblado y por consiguiente un tanto limitado en su facilidad de utilización haciendo dificultosa su total comprensión. Este hecho se ha visto incluso ahondado en su revisión de 1993.

Pero esta deficiencia se ve altamente recompensada por la disponibilidad pública, y la seguridad que le otorga el verse revisada y sometida a mantenimiento por el IEEE.

La independencia en la metodología de diseño, su capacidad descriptiva en múltiples dominios y niveles de abstracción, su versatilidad para la descripción de sistemas complejos, su posibilidad de reutilización y en definitiva la independencia de que goza con respecto de los fabricantes, han hecho que VHDL se convierta con el paso del tiempo en el lenguaje de descripción de hardware por excelencia

Conociendo el lenguaje VHDL

El lenguaje VHDL está creado específicamente para el diseño de hardware, es decir, podremos implementar con él multitud de circuitos lógicos, tanto combinacionales como secuenciales. Éste lenguaje también nos permite describir elementos más complejos, como CPU's (Unidad Central de Procesamiento), manejar ficheros, retrasos en el tiempo, etc. pero no siempre se puede implementarlos; tan sólo, y en según que casos, se llegará a la simulación.

VHDL fue desarrollado como un lenguaje para el modelado y simulación lógica dirigida por eventos de sistemas digitales, y actualmente se lo utiliza también para la síntesis automática de circuitos. El VHDL fue desarrollado de forma muy parecida al ADA debido a que el ADA fue también propuesto como un lenguaje puro pero que tuviera estructuras y elementos sintácticos que permitieran la programación de cualquier sistema hardware sin limitación de la arquitectura. El ADA tenía una orientación hacia sistemas en tiempo real y al hardware en general, por lo que se lo escogió como modelo para desarrollar el VHDL.

VHDL es un lenguaje con una sintaxis amplia y flexible que permite el modelado estructural, el flujo de datos y de comportamiento hardware. VHDL permite el modelado preciso, en distintos estilos, del comportamiento de un sistema digital conocido y el desarrollo de modelos de simulación.

Uno de los objetivos del lenguaje VHDL es el modelado. Modelado es el desarrollo de un modelo para simulación de un circuito o sistema previamente implementado cuyo comportamiento, por tanto, se conoce. El objetivo del modelado es la simulación. Otro de los usos de este lenguaje es la síntesis automática de circuitos. En el proceso de síntesis, se parte de una especificación de entrada con un determinado nivel de abstracción, y se llega a una implementación más detallada, menos abstracta. Por tanto, la síntesis es una tarea vertical entre niveles de abstracción, del nivel más alto en la jerarquía de diseño se va hacia el más bajo nivel de la jerarquía.

El VHDL es un lenguaje que fue diseñado inicialmente para ser usado en el modelado de sistemas digitales. Es por esta razón que su utilización en síntesis no es inmediata, aunque lo cierto es que la sofisticación de las actuales herramientas de síntesis es tal que permiten implementar diseños especificados en un alto nivel de abstracción.

La síntesis a partir de VHDL constituye hoy en día una de las principales aplicaciones del lenguaje con una gran demanda de uso. Las herramientas de síntesis basadas en el lenguaje permiten en la actualidad ganancias importantes en la productividad de diseño.

Algunas ventajas del uso de VHDL para la descripción de hardware son:

- VHDL permite diseñar, modelar, y comprobar un sistema desde un alto nivel de abstracción bajando hasta el nivel de definición estructural de puertas.
- Circuitos descritos utilizando VHDL, siguiendo unas guías para síntesis, pueden ser utilizados por herramientas de síntesis para crear implementaciones de diseños a nivel de puertas.
- Al estar basado en un estandar (IEEE Std 1076-1987) los ingenieros de toda la industria de diseño pueden usar este lenguaje para minimizar errores de comunicación y problemas de compatibilidad.
- VHDL permite diseño Top-Down, esto es, permite describir (modelado) el comportamiento de los bloques de alto nivel, analizándolos (simulación), y refinar la funcionalidad de alto nivel requerida antes de llegar a niveles más bajos de abstracción de la implementación del diseño.
- Modularidad: VHDL permite dividir o descomponer un diseño hardware y su descripción VHDL en unidades más pequeñas.

VHDL describe estructura y comportamiento

Existen dos formas de describir un circuito. Por un lado se puede describir un circuito indicando los diferentes componentes que lo forman y su interconexión, de esta manera tenemos especificado un circuito y sabemos como funciona; esta es la forma habitual en que se han venido describiendo circuitos y las herramientas utilizadas para ello han sido las de captura de esquemas y las descripciones netlist.

La segunda forma consiste en describir un circuito indicando lo que hace o como funciona, es decir, describiendo su comportamiento. Naturalmente esta forma de describir un circuito es mucho mejor para un diseñador puesto que, lo que realmente interesa es el funcionamiento del circuito más que sus componentes. Por otro lado, al encontrarse lejos de lo que un circuito es realmente puede plantear algunos problemas a la hora de realizar un circuito a partir de la descripción de su comportamiento.

El VHDL va a ser interesante puesto que va a permitir los dos tipos de descripciones:

Estructura: VHDL puede ser usado como un lenguaje de Netlist normal y corriente, donde se especifican por un lado los componentes del sistema y por otro sus interconexiones.

Comportamiento: VHDL también se puede utilizar para la descripción comportamental o funcional de un circuito. Esto es lo que lo distingue de un lenguaje de Netlist. Sin necesidad de conocer la estructura interna de un circuito es posible describirlo explicando su funcionalidad. Esto es especialmente útil en simulación ya que permite simular un sistema sin conocer su estructura interna, pero este tipo de descripción se está

volviendo cada día más importante porque las actuales herramientas de síntesis permiten la creación automática de circuitos a partir de una descripción de su funcionamiento.

Un programa en VHDL consta de dos partes. La primera, la entidad, nos sirve para relacionar nuestro diseño con el mundo exterior, es decir, analizamos lo que tratamos de crear como una "caja negra", de la que sólo conocemos sus entradas, sus salidas y la disposición de las mismas. La segunda parte, la arquitectura, describe como trata el circuito la información correspondiente a las entradas para obtener las salidas.

Sentencias concurrentes y secuenciales

Para iniciarse correctamente en el aprendizaje y manejo de VHDL es importante comprender desde un principio la diferencia entre concurrente y secuencial.

El concepto de concurrencia, se ve claramente graficado en los circuitos electrónicos donde los componentes se encuentran siempre activos, existiendo una asociación intrínseca, entre todos los eventos del circuito; ello hace posible el hecho de que si se da algún cambio en una parte del mismo, se produce una variación (en algunos casos casi instantánea) de otras señales.

Este comportamiento de los circuitos reales obliga a que VHDL soporte estructuras específicas para el modelado y diseño de este tipo de especificaciones de tiempos y concurrencias en el cambio de las distintas señales digitales de los diseños.

Por el contrario, las asignaciones secuenciales, son más bien propios de los SDL (soft design lenguage) en los que la programación tiene un flujo natural secuencializado, siendo propio de este tipo de eventos las sentencias case, if, while, loop, etc más propias de estas sintaxis.

Las construcciones concurrentes del lenguaje son usadas dentro de estructuras concurrentes, por ejemplo una arquitectura tiene una naturaleza eminentemente concurrente (es decir que está activo todo el tiempo), mientras que el cuerpo de un process es en principio eminentemente secuencial. La asignación de eventos secuenciales dentro de una estructura concurrente se ejecutará de forma concurrente, es decir, al mismo tiempo que las demás sentencias.

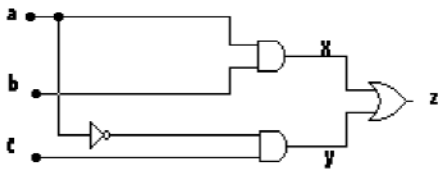
VHDL soporta con este motivo, tres tipos de objetos, las variables, las constantes y las señales. Como las variables y las señales pueden variar su valor mientras ejecutamos un programa, serán éstas las encargadas de almacenar dichos datos, asimismo serán los portadores de la información. Únicamente las señales pueden tener la connotación de globalidad dentro de un programa, es decir, que pueden ser empleadas dentro de cualquier parte del programa a diferencia de las variables que solo tienen sentido en el interior de un process .

Los process son estructuras concurrentes constituidas por sentencias de ejecución secuencial, esto provocará que dentro de un process nos encontremos con sentencias similares a las de los SDL (lenguajes de descripción de software) que nos llevan a emplear VHDL como si de otro lenguaje común se tratara. Dentro de un process nos podemos encontrar con la declaración y utilización de las variables como parámetros locales al process.

En la ejecución secuencial, las variables evalúan su valor dentro del cuerpo del proceso de forma inmediata, sin consumir tiempo de ejecución, pero como están dentro de un process, que es una estructura concurrente, este valor no será asumido, sino hasta el final de la ejecución de dicho process.

Veamos el siguiente ejemplo de asignación concurrente para señales:

```
w<= not a;
x <= a and b;
y <= c and w;
z <= x or y;
```



Al producirse un cambio en la parte derecha de la estructura de asignación (<= NOT a) de alguna de las sentencias, la expresión es evaluada de nuevo en su totalidad, con la siguiente asignación del nuevo valor a la señal de la izquierda. Esto puede provocar que múltiples asignaciones en el cuerpo de una arquitectura se activen simultáneamente, como por ejemplo, en la imagen superior, la cual se corresponde al código situado a su izquierda.

PIC

Microcontroladores PIC17C4X

El PIC17C4X es una familia de microcontroladores de gama alta, de 8 bits con memoria de programa EPROM/ROM, basada en tecnología CMOS. Se hará hincapié en la CPU, los registros, y los puertos de E/S, ya que es lo que se va a implementar, aunque hay que citar que esta familia también posee un temporizador (timer), además de un perro guardián (WatchDog), componentes típicos en los microcontroladores.

La familia del PIC17C4X, la componen básicamente tres modelos:

· El PIC17C42, con 2K de memoria de programa EPROM/ROM y 232 registros de propósito general (memoria de datos). No incluye multiplicador hardware (aunque existe una variación, modelo PIC17C42A, que si lo incluye).

· El PIC17C43, con 4K de memoria de programa EPROM/ROM y 454 registros de propósito general. Incluye multiplicador hardware.

· El PIC17C44, con 8K de memoria de programa EPROM/ROM y 454 registros de propósito general. Incluye multiplicador hardware.

Características

Entre las características de esta familia de microcontroladores cabe destacar:

- Posee 58 instrucciones simples (RISC). Las instrucciones se pueden agrupar en 3 tipos:
  - Orientadas a byte
  - Orientadas a bit
  - Literales y operaciones de control

- La arquitectura del procesador sigue el modelo Harvard.
  La arquitectura Harvard permite a la CPU acceder simultáneamente a las dos memorias (Ver figura 7). Además, propicia numerosas ventajas al funcionamiento del sistema, tal como se explicó en capítulos anteriores.

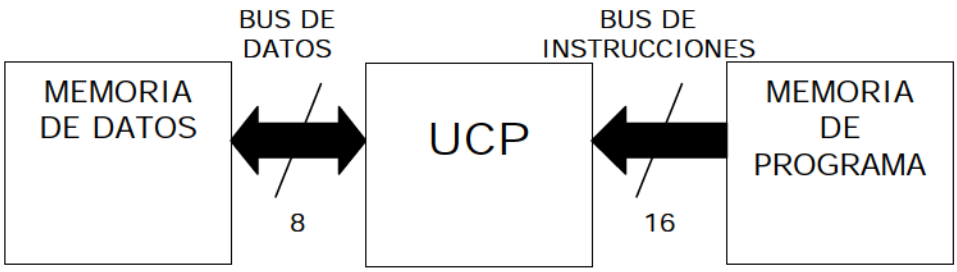


Figura 7

- Se aplica la técnica de segmentación ("pipe-line") en la ejecución de las instrucciones.
- Todas las instrucciones son ortogonales.
- El contador de programa tiene una pila con profundidad de 16 niveles.
- Permite direccionamiento directo, indirecto e inmediato para datos e instrucciones.
- Puede tener hasta 454 registros de propósito general (memoria de datos).
- Puede direccionar hasta 64Kx16 de memoria de programa.
- Controlador de interrupciones.
- Multiplicador 8x8 Hardware.

Arquitectura

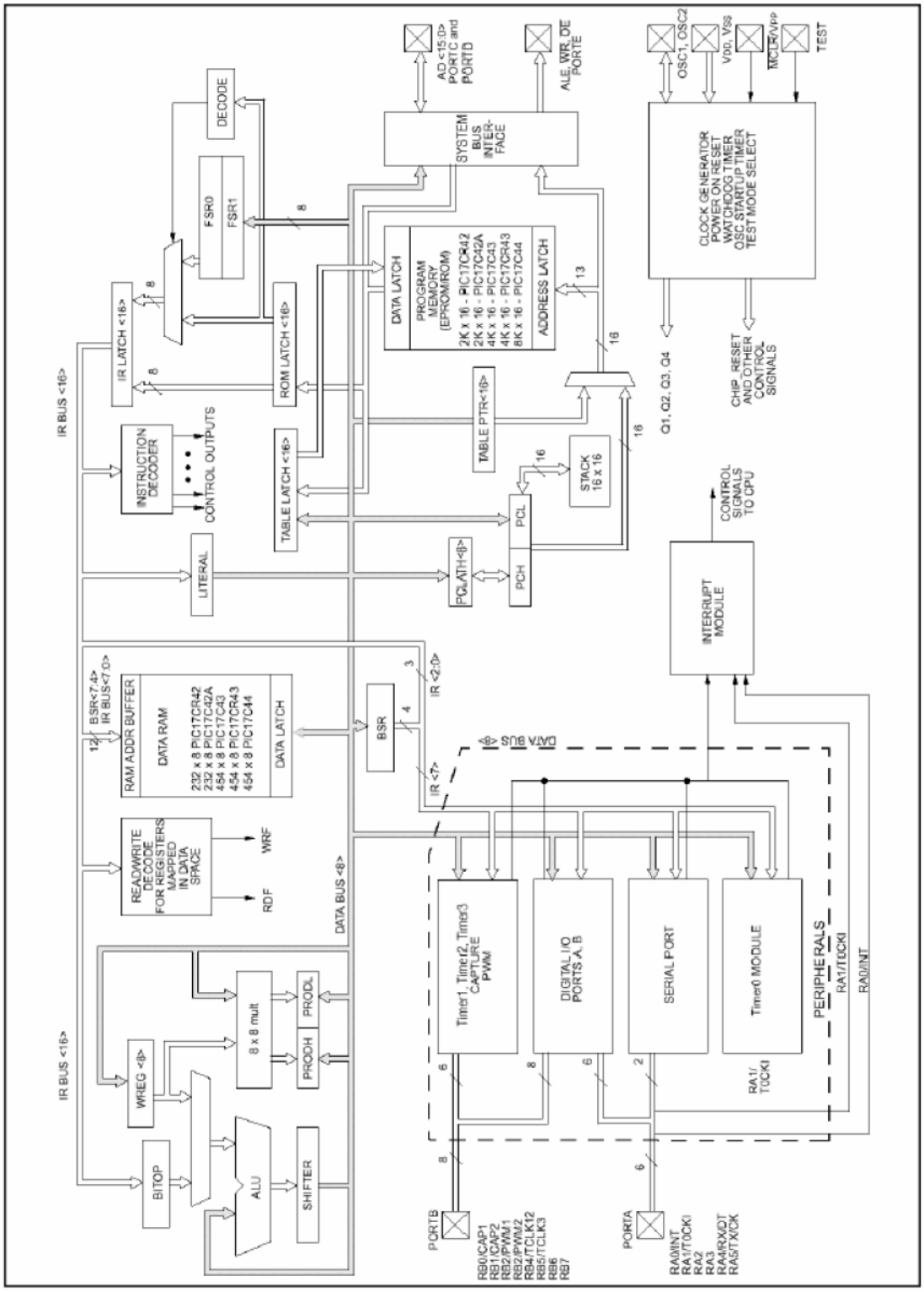
La arquitectura se mantiene como RISC y tipo Harvard. Las instrucciones tienen una longitud de 16 bits, la misma que la de las posiciones de la memoria de programa. Esta es independiente de la memoria de datos, cuyas posiciones son de 8 bits (byte). La memoria de programa alcanza internamente 8Kx16 bits, sin embargo es factible ampliarla externamente y alcanzar los 64Kx16 bits.

Los registros especiales (SFR) y los de propósito general (GPR) están englobados en la memoria de datos y se pueden direccionar directa e indirectamente, más delante de detallará en que se especializan cada uno.

La ALU de 8 bits permite realizar operaciones aritmética y lógicas sobre un operando o 2 operandos. Todas las operaciones de un simple operando se realizan sobre el WREG o 'registro de trabajo' y para las operaciones de 2 operandos, uno de los operandos es el WREG y el del otro puede ser un registro o una constante de 8 bits.

Existen instrucciones (MOVPF y MOVFP) que permiten la transferencia entre dos posiciones del área de datos, sin tener que pasar a través del registro de trabajo WREG.

El multiplicador hardware permite, valga la redundancia, multiplicar dos operandos de 8 bits generando un resultado de 16 bits en un solo ciclo. Este resultado se guardará en los registros especiales PRODH (byte alto de la multiplicación) y PRODL (byte bajo).


AD <15:0>
PORTC and
PORTD
ALE, WR, OE
PORTE
OSC1, OSC2
VDD, VSS
MCLR/VPP
TEST
SYSTEM
BUS
INTER-
FACE
CLOCK GENERATOR
POWER ON RESET
WATCHDOG TIMER
OSC STARTUP TIMER
TEST MODE SELECT
DECODE
FSR0
FSR1
DATA LATCH
PROGRAM
MEMORY
(EPROM/ROM)
2K x 16 - PIC17CR42
2K x 16 - PIC17C42A
4K x 16 - PIC17C43
4K x 16 - PIC17CR43
8K x 16 - PIC17C44
ADDRESS LATCH
IR LATCH <16>
ROM LATCH <16>
IR BUS <16>
INSTRUCTION
DECODER
CONTROL OUTPUTS
TABLE PTR<16>
TABLE LATCH <16>
STACK
16 x 16
Q1, Q2, Q3, Q4
CHIP RESET
AND OTHER
CONTROL
SIGNALS
PCL
PCH
PCLATH<8>
LITERAL
CONTROL
SIGNALS
TO CPU
INTERRUPT
MODULE
BSR<7:4>
IR BUS<7:0>
RAM ADDR BUFFER
DATA RAM
232 x 8 PIC17CR42
232 x 8 PIC17C42A
454 x 8 PIC17C43
454 x 8 PIC17CR43
454 x 8 PIC17C44
DATA LATCH
BSR
IR <2:0>
IR <7>
DATA BUS
READ/WRITE
DECODE
FOR REGISTERS
MAPPED
IN DATA
SPACE
WRF
RDF
DATA BUS <8>
WREG <8>
8 x 8 mult
PRODL
PRODH
BITOP
ALU
SHIFTER
Timer1, Timer2, Timer3
CAPTURE
PWM
DIGITAL I/O
PORTS A, B
SERIAL PORT
Timer0 MODULE
RA1/
T0CKI
PERIPHERALS
RA1/T0CKI
RA0/INT
PORTB
RB0/CAP1
RB1/CAP2
RB2/PWM1
RB3/PWM2
RB4/TCLK12
RB5/TCLK3
RB6
RB7
PORTA
RA0/INT
RA1/T0CKI
RA2
RA3
RA4/RX/DT
RA5/TX/CK

Interrupciones

Una interrupción consiste en una detención del programa en curso para realizar una determinada rutina que atienda la causa que ha provocado la interrupción. Funciona como una llamada a subrutina, que no se origina por la instrucción CALL. Tras la finalización de la rutina de interrupción, se retorna al programa principal en el punto en que se abandonó.

Los dispositivos PIC17C4X tienen 11 fuentes de interrupción diferentes agrupadas en 4 vectores de interrupción. Como solo se implementarán dos de ellas, nos centraremos solamente en:

- Interrupción externa desde el pin RA0/INT (patita 0 del puerto A) que coincide con el vector 1.
- Cambio en los pines RB7:RB0 (patitas del puerto B) que coincide con uno de las 8 fuentes de interrupción que incluye el vector 4.

Asociados a las interrupciones, existen 5 registros especiales usados para controlar y conocer el estado de las mismas:

- CPUSTA
- INTSTA
- TOSTA
- PIE
- PIR

El registro CPUSTA contiene el bit GLINTD. Este es el bit de Deshabilitación Global de Interrupciones. Cuando este bit vale 1, todas las interrupciones están deshabilitadas. Este bit, forma parte de diseño del controlador y esta descrito en la descripción de la memoria de datos.

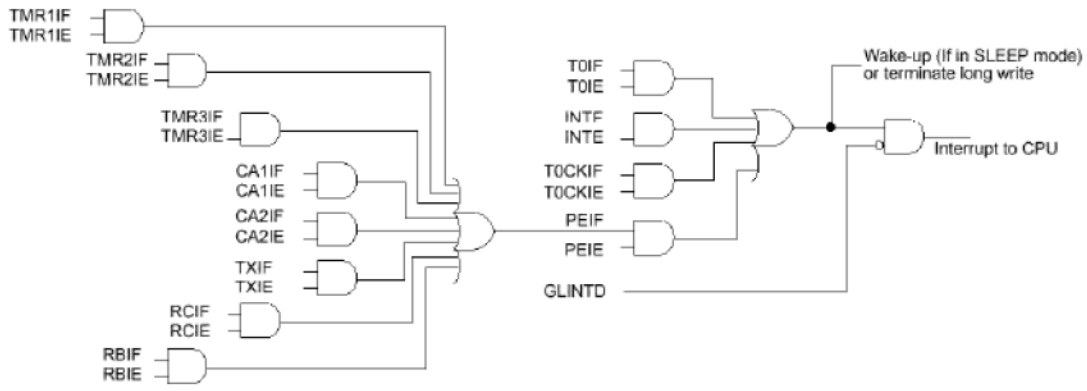


Figura 8

Cuando una interrupción es atendida, el bit GLINTD se pone automáticamente en 1 para impedir cualquier posible nueva interrupción. La dirección de retorno se introduce en la pila del sistema, y se carga en el PC la dirección del vector de interrupción correspondiente. Existen cuatro vectores de interrupción. Cada dirección del vector esta asociada a una

única fuente de interrupción (excepto las interrupciones debidas a los periféricos, que tienen asociado un mismo vector). Estas fuentes son:

- Interrupción externa desde el pin RA0/INT (patita 0 del puerto A), que llamaremos vector 1.
- Desbordamiento del temporizador 0 (TMR0), que llamaremos vector 2.
- Suceso de flanco (de subida o de bajada) de reloj externo (TOCKL), y que llamaremos vector 3.
- Una interrupción producida por los periféricos, y que llamaremos vector 4.

Cuando se atienda alguna de estás causas de interrupción (excepto el caso de los periféricos), el bit de bandera (flag) de interrupción se pondrá automáticamente en 0. En el caso de que la interrupción se deba a los periféricos, habrá que determinar por software, de que instrucción se trata. Para ello, la rutina de interrupción deberá comprobar una a una las 8 posibles causas (comprobando el valor de los registros PIE y PIR). Por lo que el bit de bandera correspondiente deberá ponerse en 0 por software, antes de rehabilitar las interrupciones.

Cada uno de los bits "banderas" de interrupción, deben ser comprobados con la mascara de habilitación de las interrupciones correspondientes y el resultado de todas ellas con el bit GLINTD.

Existe una instrucción exclusiva para el "retorno de interrupción", RETFIE, situada siempre al final de cada una de las rutinas de interrupción asociadas a los vectores de interrupción. Cuando la instrucción se ejecuta, se saca de la pila el valor de PC y GLINTD se pone en 0 (rehabilitando las interrupciones).

Registro de estado de interrupciones (INTSTA)

Este registro esta dividido en 2 grupos. Los 4 bits menos significativos configuran los valores de las mascaras de las 4 fuentes de interrupción. Un 1 en estos bits habilita la fuente de interrupción correspondiente (solo interesa INTF que corresponde al vector de interrupción 1 y PEIF que corresponde al vector de interrupción 4).

Los 4 bits más significativos están orientados a "latchear" (capturar y conservar) la posible interrupción de la fuente correspondiente. Un 1 en estos bits significa por tanto que se ha producido una interrupción causada por la fuente que corresponda a esos bits (solo interesa INTE que corresponde al vector de interrupción 1 y PEIE que corresponde al vector de interrupción 4)

Registro TOSTA

De este registro solo nos interesa el bit 7 denominado INTEDG (selector de flanco de interrupción). Si en este bit vale 1, la fuente de interrupción que tiene asociado el vector de interrupción 1 (es decir el pin RA0/INT), se activara cuando se produzca un flanco de subida de la señal

que ataque al pin RA0/INT. En cambio si el citado bit vale 0 estará configurado por flanco de bajada.

Organización de Memoria

Tal como lo exige la arquitectura Harvard, los PIC17C4X disponen de dos bloques de memoria independientes: uno destinado al programa y otro para los datos.

Memoria de programa

El contador de programa de los dispositivos PIC17C4X tiene un ancho de 16bits, capaz de direccionar 64Kx16 instrucciones. El vector de reset esta en la dirección 0000h, y los vectores de interrupción se encuentran en las direcciones 0008h, 0010h, 0018h, y 0020h. En la figura 9 se muestra el esquema del mapa de la memoria de programa.

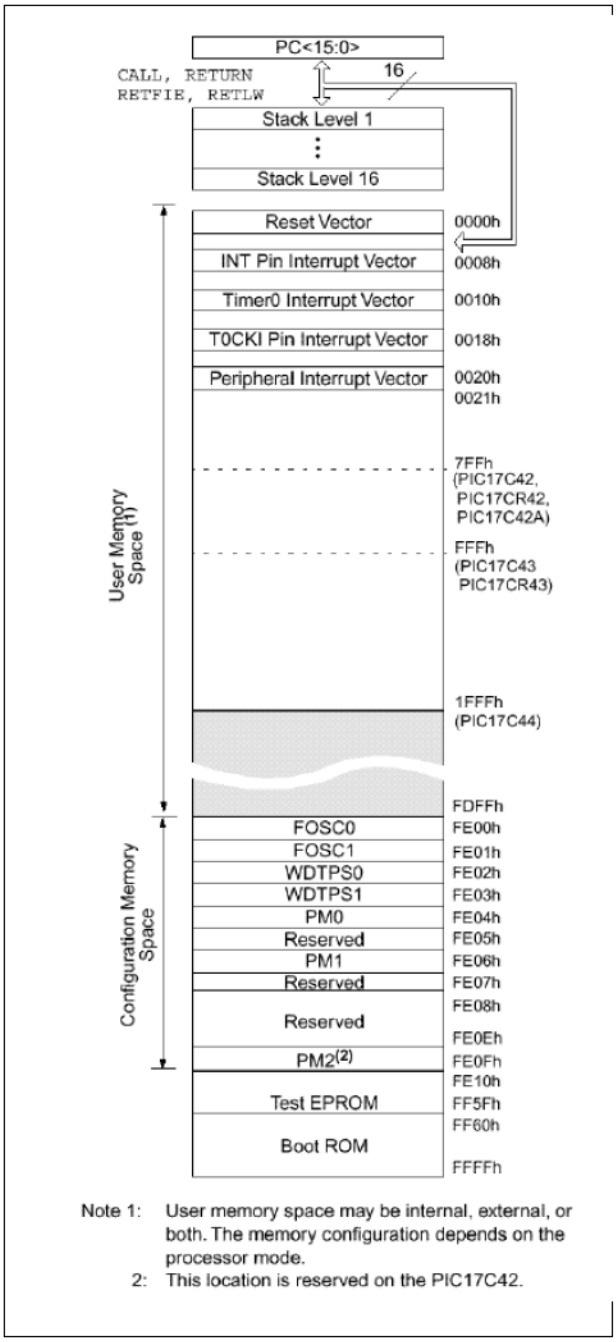


Figura 9

El contador de programa

Como ya ha dicho el contador de programa (PC) es un registro de 16 bits. PCL, el byte bajo de PC, es un registro accesible de la memoria de datos. Se puede leer y escribir como cualquier otro registro. PCH, el byte alto de PC. Sin embargo no es accesible directamente (no es un registro de

la memoria de datos). Por ello el registro PCLATCH se usa para mantener "guardado" PCH. PCLATCH es accesible y permite conocer el valor de PCH a través de él.

PC se incrementa después de ejecutar cualquier instrucción, excepto:

- Que se modifique a través de las instrucciones GOTO, CALL, LCALL, RETURN, RETLW.
- Que se modifique debido a la atención de una interrupción.
- Se modifique directamente PCL.

En estos casos la instrucción siguiente a ejecutar será NOP (No operar). Ya que la instrucción cargado durante la ejecución actual, no se va a ejecutar.

A continuación se estudia por separado las modificaciones que sufre PC y PCLATCH para las siguientes instrucciones:

- Instrucción LCALL: PCL da la dirección del salto a subrutina. Los 8 bits bajos de ese salto vienen dado como operando en la instrucción (CODIGO OP). PCLATCH no se modifica.

  PCLATCH -> PCH
  CODIGO OP<7:0> -> PCL

- Instrucción que lea PCL: Cualquier instrucción que lea PCL.

  PCL -> BUS de DATOS -> ALU o Destino
  PCH -> PCLATCH

- Instrucción que escriba PCL: Cualquier instrucción que escriba PCL.

  DATO de 8 bits -> BUS de DATOS -> PCL
  PCLATCH -> PCH

- Instrucción que lea, escriba y modifique PCL: Cualquier instrucción que lea, escriba y modifique PCL como por ejemplo ADDWF PCL.

  LECTURA: PCL -> BUS de DATOS -> ALU
  ESCRITURA: RESULTADO de 8 bits -> BUS de DATOS -> PCL
  PCLATCH -> PCH

- Instrucción RETURN:

  PCH -> PCLATCH
  PILA<TOS> -> PC <15:0>

- Instrucción GOTO y CALL: La primera es un salto incondicional y la segunda es un salto a subrutina (por lo que en este caso también se guarda PC en la pila).

  CODIGO OP<12:0> -> PC <12:0>

PC<15:13> -> PCLATCH<7:5>
CODIGO<12:8> -> PCLATCH <4:0>

Reset

Cuando se aplica un reset el PC apunta a la dirección 0x0000 (Vector de reset). Ahí habrá normalmente una instrucción de salto incondicional (GOTO) que apuntara a la dirección de comienzo del programa a ejecutar por el PIC.

Memoria de datos

La memoria de datos está dividida en dos partes.

La primera la componen los registros de propósito general (GPR), mientras que la segunda corresponde a los registros especiales (SFR). Los SFR´s controlan el funcionamiento del dispositivo. Las dos áreas están repartidas en bancos, que se seleccionan mediante ciertos bits destinados a ese propósito que se hallan en este caso en el Registro de Selección de Banco (BSR). Cuando se realiza un acceso a una posición situada fuera de los bancos, se ignoran los bits de BSR.

Se describen a continuación algunas peculiaridades de los registros:

a) Los registro de indirección INDF0 e IND1

Son especiales en el sentido de que no tienen existencia física, por lo que no se podrá acceder a ellos. Sirven únicamente para especificar la utilización del direccionamiento indirecto.

b) Los registros FSR0 y FSR1.

Se utilizan para el direccionamiento indirecto. En este caso se disponen de dos punteros para direccionamiento indirecto. Si estos registros no se utilizan para el direccionamiento indirecto, se pueden emplear como registros de propósito general.

c) Los registros PCL y PCLATCH (PC)

Fueron descriptos en la sección de Contador de programa.

d) El registro de estado de la ALU (ALUSTA)

Contiene los flags (señalizadores) C, DC, Z y OV. Este último se usa en operaciones aritméticas con números con signo y advierte del desbordamiento en el bit más significativo (bit 7). Este hecho origina el cambio del estado del octavo bit que es el de signo. Los cuatro bits más significativos seleccionan el modo de direccionamiento indirecto. Se pueden configurar en:

- Modo normal, bits FS1:FS01=1X para FSR0, FS3:FS2=1X para FSR1

- Post-auto-decremento, bits FS1:FS01=00 para FSR0, FS3:FS2=00 para FSR1
- Post-auto-incremento, bits FS1:FS01=01 para FSR0, FS3:FS2=01 para FSR1

e) El registro de estado de la CPU (CPUSTA)

Contiene los bits de estado y control de la CPU. De los cuatro bits que posee este registro, destaca el bit STKAV que indica si la pila se ha desbordado y el bit GLINTD que habilita (0) / deshabilita globalmente las interrupciones.

f) Los registros TOSTA e INTSTA.

Relacionados con las interrupciones.

g) Los registros TBLPTRL/TBLPTRH (Puntero de las operaciones Tabla)

Estos registros forman un puntero de 16 bits que configuran la dirección de la memoria de programa de 64K. Este puntero lo utilizan las instrucciones TABLWT Y TABLRD que sirven para transferir información entre los espacios de la memoria de programa y de datos.

h) Los registros TBLATH/TBLATL (Valor del traspaso en las operaciones Tabla)

Estos dos registros forman uno de 16 bits que se emplea para guardar temporalmente la información que se transfiere entre las memorias.

Los PIC17C4X disponen de cuatro instrucciones especiales que permiten mover información entre las memorias. Como la anchura de la memoria de programa es de 16 bits y los datos sólo de 8 bits, hay que realizar dos operaciones para completar la transferencia.

Las instrucciones TLWT y TABLWT se usan para escribir información desde la memoria de datos a la de programa. Las instrucciones TLRD y TABLRD pasan información desde la memoria de programa a la de datos.

h) El registro BSR (Selector de Banco)

Se emplea en la conmutación de bancos en el área de la memoria de datos. En el PIC17C42 sólo están implementados los 4 bits de menor peso para seleccionar los registros asociados a los periféricos. Solo son necesarios 2 de los 4 bits para seleccionar el banco. Los otros no tienen utilidad para el usuario.

Repertorio de instrucciones

El PIC17C4X dispone de un repertorio de 58 instrucciones de 16 bits. Los bits de la instrucción se dividen en varios campos. En la se muestras los distintos formatos de instrucción. Ver figura 10.

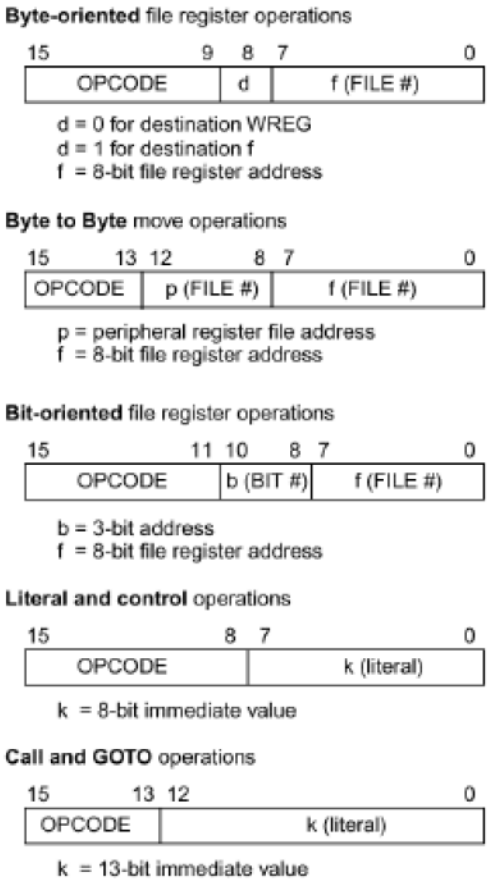


Figura 10

Todas las instrucciones se ejecutan en un ciclo de reloj excepto las:

- Instrucciones condicionales
- Instrucciones que modifican el valor de PC
- Instrucciones de tablas.

A continuación se expone un resumen de las instrucciones que poseen estos microcontroladores:

Operación y operandos	Descripción	Ciclos	Estado afectado	Notas
ADDWF f,d	Suma WREG a f	1	OV,C,DC,Z	
ADDWFC f,d	Suma WREG y el bit de Carry a f	1	OV,C,DC,Z	
ANDWF f,d	AND WREG con f	1	Z	
CLRF f,s	Clear f, o clear f y Clear WREG	1	None	3
COMF f,d	Complementa f	1	Z	
CPFSEQ f	Compara f con WREG, salta si f = WREG	1(2)	None	6,8
CPFSGT f	Compara f con WREG, salta si f > WREG	1(2)	None	2,6,8
CPFSLT f	Compara f con WREG, salta si f < WREG	1(2)	None	2,6,8
DAW f,s	Ajuste decimal WREG	1	C	3
DECF f,d	Decrementa f	1	OV,C,DC,Z	
DECFSZ f,d	Decrementa f, salta si es 0	1(2)	None	6,8
DECSNZ f,d	Decrementa f, salta si no es 0	1(2)	None	6,8
INCF f,d	Incrementa f	1	OV,C,DC,Z	
INCFSZ f,d	Incrementa f, salta si es 0	1(2)	None	6,8
INCSNZ f,d	Incrementa f, salta si no es 0	1(2)	None	6,8
IORWF f,d	IOR WREG con f	1	Z	
MOVFP f,p	Mueve f a p	1	None	
MOVPF p,f	Mueve p a f	1	Z	
MOVWF f	Mueve WREG a f	1	None	
MULWF f	Multiplica WREG con f	1	None	9
NEGW f,s	Niega WREG	1	OV,C,DC,Z	1,3
NOP	NO operación	1	None	
RLCF f,d	Rota a izquierda f con Carry	1	C	
RLNCF f,d	Rota a izquierda f sin carry	1	None	
RRCF f,d	Rota a derecha f con carry	1	C	
RRNCF f,d	Rota a derecha f sin carry	1	None	
SETF f,s	Setea f	1	None	3
SUBWF f,d	Resta WREG de f	1	OV,C,DC,Z	1
SUBWFB f,d	Resta WREG de f con Borrow	1	OV,C,DC,Z	1
SWAPF f,d	Swap f	1	None	
TABLRD t,i,f	Lee tabla	2(3)	None	7
TABLWT t,i,f	Escribe tabla	2	None	5
TLRD t,f	Lee tabla latch	1	None	
TLWT t,f	Escribe tabla latch	1	None	
TSTFSZ f	Test f, salta si es 0	1(2)	None	6,8
XORWF f,d	XOR WREG con f	1	Z	
BCF f,b	Limpia bit en f	1	None	
BSF f,b	Setea bit en f	1	None	
BTFSC f,b	Testea bit, salta si esta limpio	1(2)	None	6,8
BTFSS f,b	Testea bit, salta si esta seteado	1(2)	None	6,8
BTG	Bit toggle f	1	None	
ADDLW k	Suma literal a WREG	1	OV,C,DC,Z	
ANDLW k	AND literal con WREG	1	Z	
CALL k	Llamada a subrutina	2	None	7
CLRWDT	Limpia el Timer del WatchDog	1		
GOTO k	Salto incondicional	2	None	7
IORLW k	IOR literal con WREG	1	Z	
LCALL k	Long Call	2	None	4,7
MOVLB k	Mueve literal a la parte baja de BSR	1	None	
MOVLR k	Mueve literal a la parte alta de BSR	1	None	9
MOVLW k	Mueve literal a WREG	1	None	
MULLW k	Multiplica literal con WREG	1	None	9
RETFIE	Retorno de interrupción	2	GLINTD	7

RETLW k	Retorna literal a WREG	2	None	7
RETURN	Retorno de subrutina	2	None	7
SLEEP	Ingresa en modo sleep	1		
SUBLW k	Resta WREG del literal	1	OV,C,DC,Z	
XORLW k	XOR literal con WREG	1	Z	

Notas:

1. Segundo método de complemento
2. Aritmética sin signo
3. Si s='1', solo el file es afectado; si s='0' tanto el WREG como el file son afectados. Si se requiere que solo el WREG sea afectado, entonces se debe setear f=WREG.
4. Durante un LCALL, el contenido de PCLATCH es cargado en el MSB del PC y kkkk kkkk es cargado en el LSB del PC (PCL).
5. Instrucción de múltiples ciclos cuando el puntero elige un EPROM interno. La instrucción es terminada con una interrupción. Cuando escribe en la memoria de un programa externo, es una instrucción de 2 ciclos.
6. Es una instrucción de 2 ciclos cuando es true verdadero y una instrucción simple cuando el falso.
7. Instrucción de 2 ciclos excepto para TABLRD al PCL (parte baja del contador) en cuyo caso toma 3 ciclos.
8. Un salto significa que la instrucción cargada durante la actual ejecución no es ejecutada, en su lugar se ejecuta un NOP.
9. Estas instrucciones no están disponible en el PIC17C42.

Descripción en VHDL

A continuación se realizará una descripción de los pasos adoptados para llegar a la implementación en VHDL del microcontrolador de la familia del PIC17C4X.

EVOLUCIÓN CRONOLOGICA DEL MODELO

En base a la hoja de datos provista por el fabricante del microcontrolador, en la cual se describen los diferentes bloques que conforman el modelo real del PIC, se comenzó con un diseño inicial de componentes que describen en VHDL las principales funciones del microcontrolador.

En la arquitectura de componentes VHDL de este diagrama se pueden distinguir la Unidad de Control, la unidad aritmético-lógica, memoria de datos y de programa y el registro de trabajo. Todos estos componentes se encuentran intercomunicados a través de la distintas señales y por buses de datos que permiten transferir la información hacia la unidad aritmético lógica y luego almacenar el resultado de la operación en el componente correspondiente. Este diagrama puede observarse en la siguiente figura:

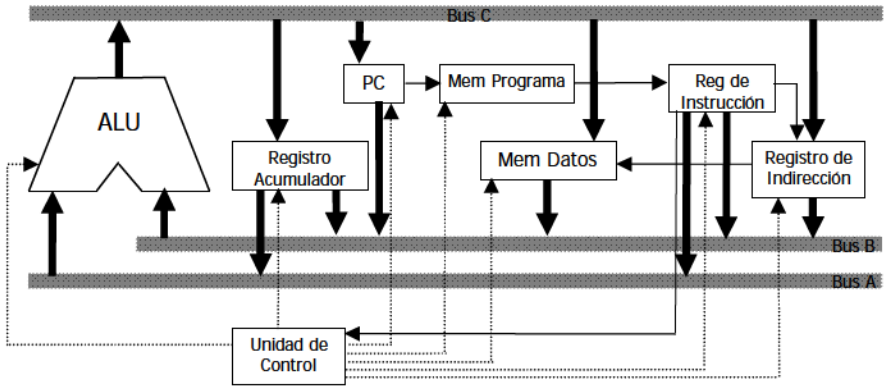


Diagrama inicial del Microcontrolador
Fig. 11

Puede observarse que los buses de datos A y B sirven como entrada a la ALU y transportan el resultado obtenido en la salida de la misma hacia el correspondiente componente según lo indica la señal activada por la Unidad de Control.

Esta arquitectura de componentes se encuentra comandada por una señal de reloj (Clock), la cual toma valores que pueden ser 0 o 1 y que se repiten continuamente. Esta señal permite tener controlados y sincronizados todos los componentes del modelo.

Una vez obtenida la descripción en VHDL según la arquitectura de componentes explicada en el diagrama inicial, se utilizó la herramienta que provee Altera mediante el University Program Design Laboratory Package, y se generó un nuevo proyecto y se procedió a realizar el procesamiento del mismo para llegar a la síntesis del modelo.

Una vez obtenido el modelo sintetizado, se cargo el mismo en el simulador que provee esta herramienta para poder hacer un análisis de su funcionamiento.

Luego de los análisis realizados sobre los valores obtenidos durante la simulación se comprobó que no se estaban obteniendo los resultados esperados, ya que debido a la arquitectura diseñada no se estaba pudiendo implementar correctamente el pipeline de instrucciones como lo requiere un procesador de este tipo para su correcto funcionamiento.

La investigación en el modelo descrito y en los resultados obtenidos luego de diversas etapas de simulación permitieron detectar que el error se producía por la utilización de los buses de datos que comunican los componentes con las entradas de la unidad aritmético lógica, pues existe una interferencia entre los valores asignados por cada componente en cada ciclo de ejecución.

Entre las diferentes mejoras propuestas para solucionar este inconveniente se realizaron distintas tareas:

- En una primer etapa se realizaron ajustes introduciendo tiempos de espera (denominados de SETUP y HOLD) en los componentes del modelo VHDL que describían el comportamiento de los Bus A y B. Una vez sintetizado este modelo se realizaron nuevamente simulaciones y los resultados obtenidos nos permitieron ver que estas mejoras no fueron suficientes pues los inconvenientes se seguían sucediendo.

- En una etapa posterior se decidió adaptar la arquitectura de componentes y se reemplazaron los buses de datos (A y B) de entrada a la ALU por componentes del tipo LATCH. Estos componentes tienen la particularidad de almacenar el valor que se ingreso en la entrada del mismo y mantenerlo por un cierto periodo de tiempo.

- En nuestro caso cada uno de los componentes que se desarrollaron poseen varias posibles entradas y una única salida, que sirve como entrada a la ALU. El componente selecciona el valor de entrada correspondiente según lo indicado por las señales de control generadas en la Unidad de Control. El período de tiempo que debe permanecer el valor en el registro esta marcado por el ciclo de clock.
  Por ello se desarrollaron las descripciones en VHDL de estos componentes para el reemplazo de los buses de datos los cuales tienen un número de entradas distinto que se corresponde con cada uno de los componentes de la arquitectura que conectan a cada bus de datos y que pueden escribir un valor en cada una de las entradas de la ALU. Estos componentes fueron denominados MuIA y MuIB. La siguiente figura muestra el diagrama con esta arquitectura de componentes:

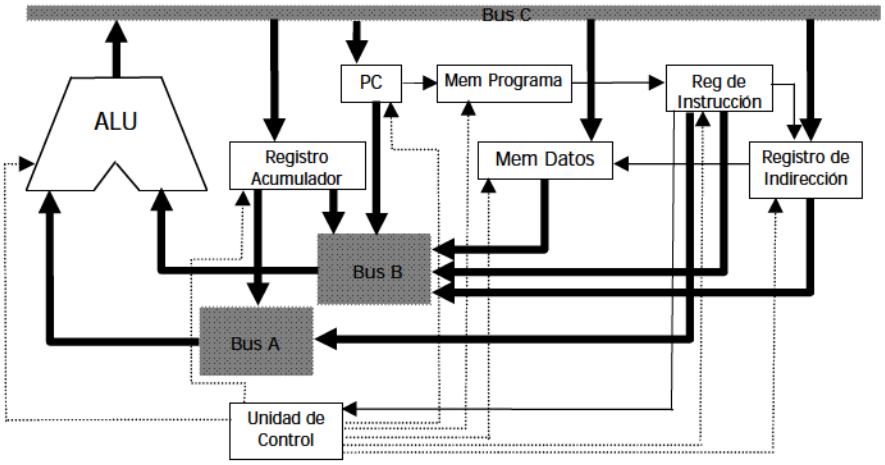


Diagrama modificado del Microcontrolador
Fig. 12

Con estos nuevos componentes se hicieron las adaptaciones al modelo diseñado para que pudiese ser sintetizado en la herramienta de desarrollo. Una vez verificado el correcto funcionamiento del modelo siempre mediante el análisis de resultados obtenidos en la simulación, se fueron sumando a la arquitectura otros componentes que permitieron reflejar más fielmente el comportamiento del microcontrolador.

Se agregaron componentes que describen el funcionamiento de los registro de memoria de indirección, generador de constantes, registro de estado de la ALU.

Cuando se genera un nuevo proyecto en la herramienta de desarrollo, Max +Plus II, se deben definir un conjunto de parámetros entre los que se encuentra el modelo de dispositivo de celdas lógicas que se utilizará para sintetizar finalmente el modelo descrito en VHDL.

Al momento de realizar la compilación de un proyecto desde la herramienta se informa el número de celdas lógicas ocupadas en el dispositivo parametrizado.

A medida que se fueron agregando componentes al proyecto que contiene la descripción del microcontrolador luego de la compilación se fueron visualizando mensajes que informan que el proyecto ocupa demasiadas celdas para el dispositivo seleccionado, por lo que se realizaron ajustes en los parámetros de los componentes para permitir contener la mayor cantidad de componentes posibles, se decidió no incluir alguno de ellos en la arquitectura del mismo o bien acotar su espacio utilizado en el dispositivo de hardware.

Descripción del Modelo:

Como explicamos en el punto anterior, se tomo como base la hoja de datos del microcontrolador provista por el fabricante, con la cual se fue desarrollando la arquitectura de componentes a utilizar para describir en VHDL.

Algunos modelos de microcontroladores de la familia PIC17C4X, poseen un multiplicador por hardware que no se ha implementado ya que se requiere una gran cantidad de celdas lógicas a utilizar en el chip para lograr que efectúe las operaciones en un único ciclo de reloj como las demás operaciones.

Otros componentes, no se han descritos en este trabajo, como son el Perro Guardián (Watch Dog), los timers adicionales que poseen este tipo de microcontroladores, el registro de estado de la CPU y la pila. La inclusión de estos componentes a la descripción del microcontrolador, no representan cambios sustanciales en el diseño actual.

A continuación se estudiarán los rasgos más importantes a la hora de implementar en VHDL cada uno de los subsistemas o bloques que describen el modelo del microcontrolador.

Unidad aritmético-lógica (alu.vhd)

La ALU, es un sistema combinacional e independiente del clock. Y esta provista de:

Entrada:

- 1 bit de reset.
- 2 líneas de 8 bits que serán el BUS A y el BUS B (los operandos).
- 1 línea multiplexada de 5 bits que seleccionará la operación a realizar.
- Varias señales binarias entre las que se destacan los bits de estado (carry, digital carry, overflow y zero) y los bits que especifican si la operación a realizar modifica alguno de estos bits de estado.

Salida:

- 1 bus de 8 bits que será el resultado, que se conectará al BUS OUT.
- 4 señales binarias que reflejarán si en la operación llevada a cabo por la ALU, el resultado a sido 0, z_Out (Z=1), ha habido acarreo, c_Out (C=1), acarreo de dígito, dc_Out (DC=1) o desbordamiento, ov_Out. (OV=1)

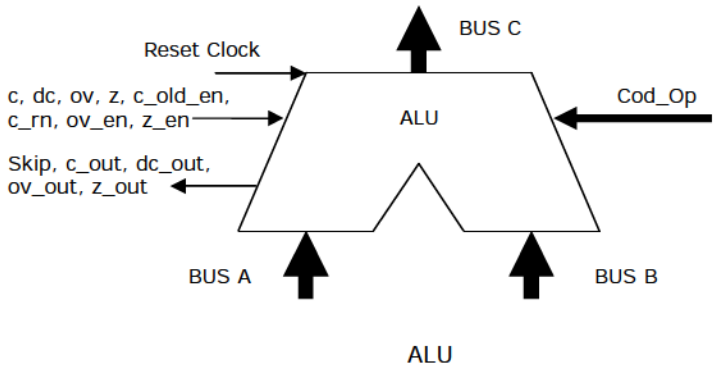


Figura 13

La unidad aritmética-lógica que implementamos puede realizar las siguientes operaciones, que se dividen por tipos o grupos.

- Operaciones aritméticas: suma y resta.
- Operaciones lógicas: not, and, or y xor.
- Operaciones de desplazamiento: rotación izquierda, rotación derecha y SWAP.
- Operaciones orientadas a bit.
- Operaciones de transferencia.
- Registro de estado de la ALU

El proceso principal toma los operandos de los buses A y B y el resultado lo saca por el bus C.

La forma que utilizamos para realizar esto, es usando un valor temporal de 9 bits donde ponemos el resultado de la operación y luego dependiendo del tipo de operación, analizamos de una forma u otro el noveno bit.

Un ejemplo puede ser:

```
when MC_SUMA  => if (c_old_en = '0')
                  then busTemp <= ('0' & busA) + ('0' & busB);
                  else busTemp <= ('0' & busA) + ('0' & busB) + c;
                 end if;
```

Cuando el código de operación tiene el valor que representa a la suma, entonces, si no es una suma con carry, realiza la suma directamente y si es una suma con carry, le suma al resultado el carry que venia arrastrando. Observar que tanto busA como busB tiene 8 bit y busTemp tiene 9 bit.

Además fuera de un proceso VHDL, se setea el estado, es decir carry, overflow, zero y digital carry.

Registro de estado (ALUSTA)

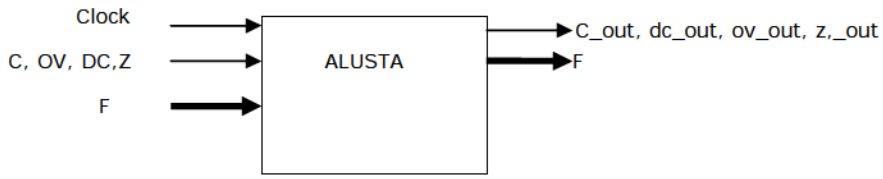
En este registro se guarda la configuración de la ALU. Es un registro de 8 bits de los cuales se puede mencionar, los bits C (Carry), Z (Zero), DC (Digit Carry) y OV (Overflow). Además los cuatro bits más significativos de este registro están dedicados a seleccionar el modo de direccionamiento indirecto.

Entrada:

- Los 4 bits de estado.
- El conjunto de 4 bits más significativo.

Salida:

- Los 4 bits de estado.
- El conjunto de 4 bits más significativo.



Estado de la ALU

Figura 14

Unidad de Control (UC)

Es el bloque que implementa la unidad de control de microcontrolador, encargado de decodificar las instrucciones y ejecutarlas generando una secuencia de ordenes (activación de señales). Recibe la señal CodOP[15:0] proveniente del registro de instrucción RegPc, en función de esta señal de entrada, genera las micro-instrucciones necesarias para que se ejecute la instrucción. Es un bloque combinacional al 100%.

La estructura de este bloque en VHDL la podemos dividir en dos partes bien diferenciadas, que actúan en paralelo (cada parte es un "PROCESS" de VHDL en el cual, las líneas de código se ejecutan secuencialmente).

La primera parte determina cual es el registro involucrado en la instrucción, tanto en la entrada como en la salida. Genera las señales que corresponden en cada caso. Además para evaluar si los registros se usan como operando o como destino se necesitan dos señales internas al diseño, que genera el segundo PROCESS.

En la segunda parte del bloque se describe el decodificador de instrucciones, en el cual se decodifica cada una de las instrucciones y se generan las señales de control que gobiernan el funcionamiento del diseño.

El código VHDL de este último proceso tiene la siguiente estructura de procesamiento de las sentencias, en la cual se utiliza una estructura de VHDL IF-THEN-ELSIF para verificar que instrucción debe decodificarse:

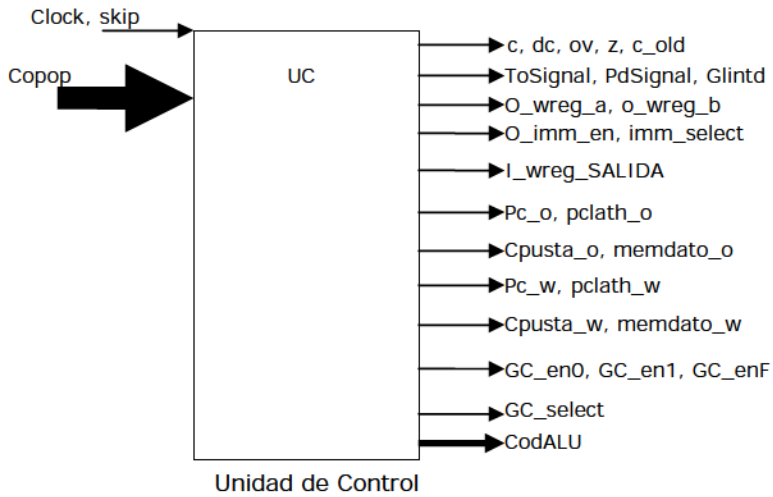
```
SI (Instrucción = ADD)
 ENTONCES Activación de las señales que ejecuten ADD.
SINO-SI (Instrucción = SUB)
 ENTONCES Activación de las señales que ejecuten SUB
      ....
```

Entrada:

- Clock.
- Código de operación (16 bits).

Salida:

- Código de operación para la alu (5 bits).
- Las señales de especifican que la operación a realizar afecta los flags de estado.
- Las señales que indican a los distintos registros que habiliten la lectura o escritura sobre los buses de datos.
- Señales para habilitar el generador de constantes.



Unidad de Control

Figura 15

Un extracto del código que encontramos en este componente sería:

```
 ELSIF CodOp(15 DOWNTO 9) = ADDWFC THEN
      OV <= '1';
      DC <= '1';
      C  <= '1';
      Z  <= '1';
      CodALU <= MC_SUMA;
c_old <= '1';
```

```
     o_wreg_b <='1';
IF(CodOp(7 DOWNTO 0) = WREG) THEN
   o_wreg_a <='1';
   i_wreg <= '1';
ELSE
    esRegFile_out <='1';
   IF CodOp(8) = '0' THEN
          i_wreg <='1';
         ELSE
         o_regfile_en <='1';
   END IF;
END IF;
```

Contador de programa (RegPC)

En este bloque se describe la unidad que se encarga de sumar uno cada vez que se ejecuta una instrucción, cuya salida es tomada por el bloque de la memoria de programa, ROM, para distinguir que instrucción debe enviar a la unidad de control para su ejecución.

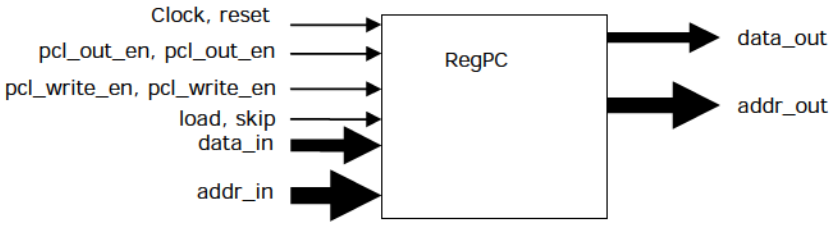
Además puede tener como salida la parte alta o baja del PC, para se utilizado como dato, así como tomar un valor de entrada para reemplazar la parte alta o baja de PC.

Entrada:

- Señales que habilitan la entrada o salida de datos tanto para la parte alta como para la parte baja del PC.
- Datos (8 bits) para el caso en el que se tenga que escribir en el PC.

Salida:

- Código de operación (16 bits).
- Datos (8 bits).



Registro PC
Figura 16

Registro de trabajo (RegAcum)

Este registro de 8 bits únicamente tiene 3 señales de control (entradas). Una para habilitar la salida al BUSA, otra similar para el BUSB, y por último, una señal para habilitar la escritura del registro procedente del BUSC.

Es un modulo muy sencillo, el cuerpo principal seria:

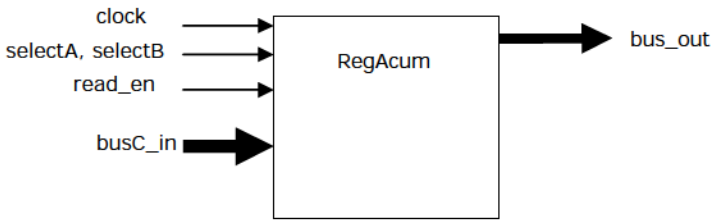
```
Process (clock, read_en, busC_in)
 begin
    if clock'event and clock='1' then
         t <='0';
    end if;
   if  read_en='1' and t='0' then
          W_Reg <= busC_in;
         t <= '1';
    end if;
end process;

Bus_out <= W_Reg when selectA='1' or selectB='1';
```

Entrada:

- Clock.
- Señales que determinan sobre que bus se trabaja y si se habilita la lectura desde los buses de datos.
- Datos (8 bits).

Salida:

- Datos (8 bits).



Registro Acumulador
Figura 17

Registro de memoria de datos (RAM)

En este bloque se describen los registros de propósito general pertenecientes a la memoria de datos. Será por tanto un bloque constituido por registros. Además, tiene señales de control que indican al bloque cuando debe realizar una acción de escritura y/o de lectura; y canales de comunicación que permitirán seleccionar el registro dentro del bloque con el cual se opera.

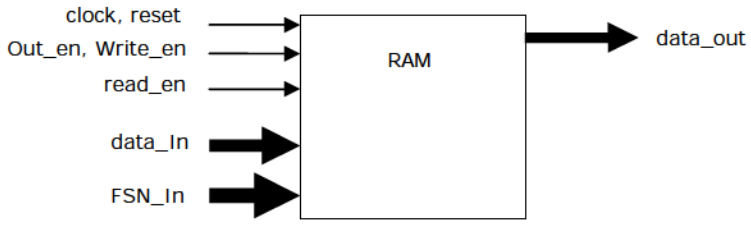
Tiene un formato similar al registro acumulador con la diferencia de que son varios registros en un solo modulo.

Entrada:

- Clock.
- Señales que habilitan la lectura y escritura desde los buses de datos.
- La dirección de memoria con la cual se va a trabajar.
- Datos (8 bits).

Salida:

- Datos (8 bits).



Memoria
Figura 18

Memoria de programa (ROM)

Simplemente, este bloque emula a una memoria ROM convencional, donde se incluye el programa a ejecutar. Recibe la dirección de la instrucción generada por el PC (PC_in<15:0>) y devuelve la instrucción (inst_out<15:0>) que va al registro de instrucción.

Ejemplo:

```
ARCHITECTURE Rtl OF Rom IS
 SIGNAL int: integer;
BEGIN
 int <= conv_integer(pc_in);
 inst_out <=
          "0000000000000000" when int = 0 else
          "1011000001100000" when int = 1 else
          "0000000100010010" when int = 2 else
          "0000011100010010" when int = 3 else
          "0000011100001010" when int = 4 else
          "0000000000000000" when int = 5 else
          "0000000000000000" when int = 6 else
          "0001010100010010" when int = 7 else
          "0001010100001010" when int = 8 else
          "0000011100010010" when int = 9 else
          "0000011100001010" when int = 10 else
          "0001010100010010" when int = 11 else
```

```
        "0001010100001010" when int = 12 else
        "0000000000000000" when int = 13;
end Rtl;
```

En el archivo se evalúa una condición en base al valor de la señal de Contador de Programa (PC) de entrada al componente y se obtiene la instrucción de salida.

La generación de este componente demanda cierta dificultad al momento de transcribir cada una de las instrucciones assembler con la cual se escriben los programas al código de máquina en el cual deben expresarse para poder ser interpretadas por la UC.

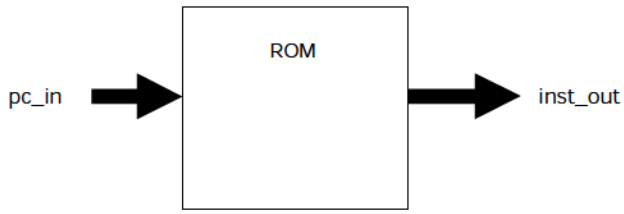
Luego de un análisis del conjunto de instrucciones que soporta el microcontrolador, se desarrollo una herramienta que permitiera realizar esta tediosa tarea de una forma más amigable al programador. Las características de esta herramienta son descritas en el capitulo de Pruebas y Simulaciones.

Entrada:

- PC.

Salida:

- Código de operación (16 bits).



Memoria de programa
Figura 19

Generador de constantes (GENCONS)

Este bloque genera valores constantes de acuerdo a las señales que recibe de la unidad de control.
Ejemplo:

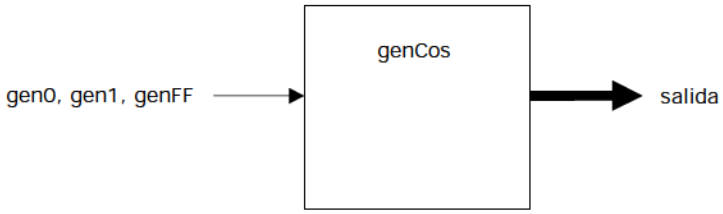
```
begin
  salida <=  "00000000" when gen0  = '1' else
             "00000001" when gen1  = '1' else
             "11111111" when genff  = '1';
end RTL;
```

Entrada:

- Señales que habilitan a la generación de constantes (0, 1 o FF).

Salida:

- La constante (8 bits).



Generador de constantes
Figura 20

Multiplexor (muxBusA y muxBusB)

Estos bloques generan como salida el dato que es tomado como entrada por la ALU, la salida la genera dependiendo de la señal de control que haya sido habilitada por la UC. Son dos bloques muy similares que se diferencian en el origen de los datos de entrada, y que el canal de salida de cada uno de estos se encuentra conectado a un canal de entrada distinto en la ALU.

Código del multiplexor A:

```
ARCHITECTURE RTL OF MuxBusA IS
SIGNAL TEMPA: STD_LOGIC_VECTOR(7 downto 0);
BEGIN

 PROCESS
   BEGIN
        IF    WREG_en= '1' THEN TEMPA  <=  WREG_data;
        ELSIF RAM_en = '1' THEN TEMPA  <=  Ram_data;
        ELSIF PC_en  = '1' THEN TEMPA  <=  PC_data ;
        ELSIF GC_en  = '1' THEN TEMPA  <=  GC_data ;
        ELSIF RI_en  = '1' THEN TEMPA  <=  RI_data;
        ELSE TEMPA  <=  TEMPA;
        END IF;
        DATA_OUTA <= TEMPA;
END PROCESS;
END RTL;
```

Entrada:

- Clock.
- Datos (8 bits) de los distintos registros (Acumulador, memoria RAM, PC o generador de constantes).
- Señales que habilitan la salida de cada registro.

Salida:

- Datos (8 bits).

Pruebas y simulaciones

Se realizará una descripción de los procesos que se llevaron a cabo durante la etapa de verificación y validación de los componentes desarrollados en VHDL que se mencionaron en la arquitectura propuesta para la descripción del microcontrolador PIC17C42.

Una vez finalizada la etapa del diseño de la arquitectura de componentes que se utilizaría para la implementación del microcontrolador, se comenzó la etapa de diseño y desarrollo de cada uno de los componentes.

A medida que se desarrollaron los diferentes componentes, estos se fueron integrando al proyecto donde se fue implementando el microcontrolador. Además una vez realizada la descripción de cada componente y con el motivo de comprobar su correcto funcionamiento se fueron realizando distintos tipos de pruebas parciales que afectaban a cada uno de los componentes diseñados.

Se diseñaron diferentes tipos de pruebas que permitieron verificar el funcionamiento de cada uno de los componentes desarrollados (pruebas de unidad) como así también la sincronización y comunicación que debe existir entre todos los componentes de la arquitectura del microcontrolador (pruebas de integración).

Para cada una de las pruebas, se diseñaron diferentes programas (secuencia de instrucciones) que formaban el conjunto de casos de pruebas que se verificaban cada vez que efectuaban los testeos del microcontrolador.

Los casos de prueba se diseñaron de modo que fuesen programas que agrupaban distintos tipos de instrucciones dependiendo de los módulos que se intentaban verificar.

En cada uno de estos casos se generaba una secuencia de instrucciones a ejecutar que se cargaban en el componente que cumple las funciones de memoria de programa (ROM) en el procesador, la cual se actualizaba en el proyecto para luego poder sintetizar el código VHDL generado y de esta manera poder verificar, mediante la simulación, los resultados obtenidos.

Lograr la descripción del componente que describe el comportamiento de la memoria de programa del microcontrolador debe ser una tarea muy detallada y que puede demandar mucho tiempo y cuidado por parte de diseñador ya que cada una de las instrucciones escritas en lenguaje assembler deben ser traducidas a su código de máquina correspondiente para que puedan ser interpretadas por la unidad de control del dispositivo. Debido a lo expresado anteriormente nos vimos ante la necesidad de desarrollar una herramienta que permitiera realizar la traducción del

programa escrito en base a las instrucciones del lenguaje assembler a las instrucciones de código máquina.

Una vez desarrollada esta aplicación, Generador de ROM, obtuvimos una herramienta que nos permite llegar a describir un programa (secuencia de instrucciones a ejecutar) de una forma sencilla y rápida, lo cual redunda en la disminución de los tiempos de prueba de los componentes.

GENERADOR DE ROM:

Como se mencionó anteriormente, para poder realizar una descripción más eficiente de la memoria de programa del microcontrolador se desarrollo una aplicación para la conversión de instrucciones en assembler al código de máquina.

Esta aplicación permite generar la descripción en VHDL del componente de memoria de programa del microcontrolador de una forma rápida y sencilla; garantizando que la traducción de las instrucciones escritas en assembler a su respectivo código de máquina se realiza en forma correcta por lo que el programador debe centrarse únicamente en la lógica del algoritmo que se cargará en la ROM y desentenderse del problema de la traducción de cada una de las instrucciones.

Esta herramienta, posee varias funcionalidades que ayudan al diseñador a especificar el programa y a generar el componente de memoria de programa del microcontrolador.

En la pantalla principal de la aplicación (Figura 17) se visualiza una barra de herramientas que permite el acceso a las funciones más utilizadas y un editor de instrucciones donde se pueden especificar la secuencia de instrucciones que conformarán el programa que luego de la síntesis se cargará en la memoria ROM del microcontrolador.

En el editor de instrucciones se permiten el ingreso de instrucciones por cada línea, las cuales serán controladas durante el proceso de comprobación para verificar que su sintaxis sea correcta. Luego mediante la compilación se generará el componente VHDL que contiene la descripción del comportamiento de la memoria de programa del microcontrolador.

Una vez obtenida la descripción en VHDL de la memoria de programa del microcontrolador se debe incluir dicho componente en el proyecto en el cual se describe el microcontrolador y se realizará la síntesis.

Debido que la estructura del componente generado para la memoria de programa es muy similar en cada caso y se genera automáticamente mediante la aplicación, se puede asegurar que la descripción obtenida no contiene errores en su sintaxis por lo que facilita la síntesis del mismo.

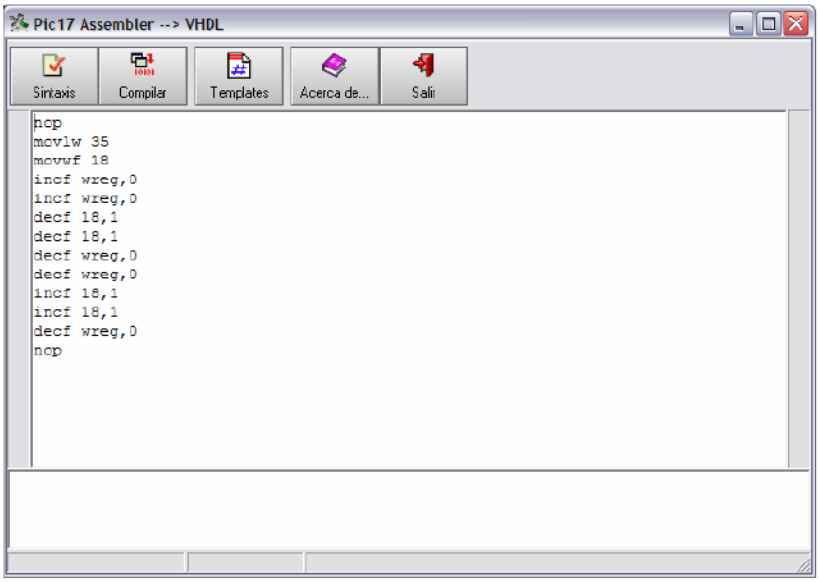


Figura 21. Pantalla principal del editor de instrucciones del ConversorPIC17.

Las principales funciones de la aplicación son:

- Chequeador de sintaxis: Cada una de las instrucciones ingresadas en el editor de instrucciones puede ser controlada para verificar que su sintaxis sea correcta antes de realizar la compilación del programa escrito a los códigos de máquina correspondientes.

  Una vez ejecutado el chequeador de sintaxis, en caso de encontrar errores en la sintaxis de las instrucciones ingresadas como programa, se indicarán en la lista de instrucciones incorrectas. En esta lista se detalla la línea de programa donde se encontró el problema y además se muestra la sintaxis que se espera para la instrucción.

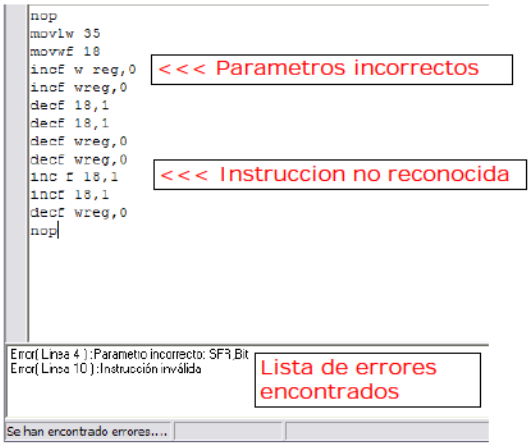


Figura 22. Eventos encontrados por el chequeador de sintaxis

- Compilador de memoria de programa: una vez que se obtiene un programa con sintaxis correcta, el compilador genera el componente de memoria de programa del microcontrolador en código VHDL, en el cual se especifica para cada ciclo de reloj cuales son las instrucciones (en código máquina) que se ejecutarán.

Debido a esto el proceso de conversión que realiza esta traducción debe tomar cada una de las instrucciones escritas en código assembler y pasarlas al código de máquina correspondiente. Una vez finalizado este procesamiento, se generará el componente VHDL que representa la descripción de la memoria de programa, generando un archivo (ROM.VHD) que puede ser guardado donde el usuario lo especifique y que luego será tomado por una herramienta que no permita realizar la síntesis del componente.

- Templates de instrucciones: para acelerar los tiempos de desarrollo de un programa escrito en código assembler del microcontrolador de la familia del PIC17, el editor permite seleccionar las posibles instrucciones que se encuentran el repertorio de instrucciones de esta familia de microcontroladores (Tabla instrucciones).

En la pantalla de templates de instrucciones (Figura 23) se permite seleccionar el template de una instrucción y automáticamente se agregada en el editor de instrucciones del programa.
Esto permite que el usuario pueda tener a mano las instrucciones posibles conjuntamente con los tipos de parámetros que se deben ingresar para dicha instrucción.

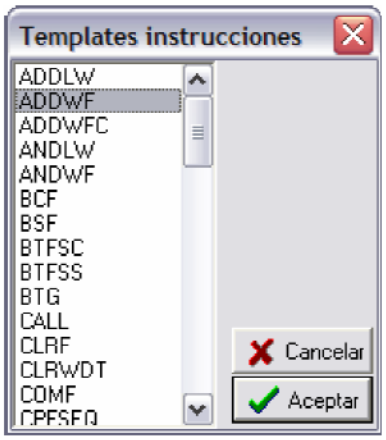


Figura 23. Templates de instrucciones del microcontrolador.

En base a cada uno de los casos de prueba diseñados se generaron distintos componentes, que representaban la memoria de programa del microcontrolador, descritos en VHDL (archivos ROM.VHD).

Luego desde el editor de la herramienta de desarrollo para VHDL, Max Plus II, se procedió a verificar el resultado de las pruebas. Para ello se deben realizar la siguiente secuencia de pasos:

- En el proyecto VHDL del microcontrolador (PIC17.vhdl) se debe seleccionar como componente ROM según el caso de prueba a verificar.

- Se compilar el proyecto, traduciendo el código VHDL a nivel de compuertas lógicas (proceso que puede demandar varios minutos).

- Una vez obtenido este modelo se realiza la ejecución del modelo en el simulador. En este paso se deben indicar (o simular) las señales que son externas al modelo tales como el la señal de clock, reset, etc; como así también se deben seleccionar que señales se van a controlar en cada caso.

- Esta simulación produce una salida grafica por pantalla, la cual muestra para cada unidad de tiempo cual era el valor de cada una de las señales que fueron seleccionadas para ser visualizadas.

- En base a los datos provistos por el simulador se pueden verificar y validar cada uno de los componentes.

Casos de prueba y resultados obtenidos

Con el modelo VHDL en el cual se describe el comportamiento del microcontrolador se realizaron distintos casos de pruebas para verificar el funcionamiento del modelo y luego se realizaron las correspondientes ejecuciones en el simulador la ejecución para los distintos procesos.

Para el desarrollo, síntesis y simulación del modelo VHDL desarrollado se utilizó el software provisto por ALTERA, denominado Max +Plus II. Esta aplicación posee varias herramientas que nos permiten ejecutar los procesos para llegar a la síntesis del modelo.

A continuación se muestran los resultados obtenidos luego de las simulaciones efectuadas con las diferentes configuraciones del microcontrolador. En cada caso se efectuó un análisis en base al resultado obtenido y el resultado esperado, como así también de los errores detectados.

Para una mayor compresión de los resultados se describen cada una de las señales que se visualizan en los distintos gráficos:

```
Reset          -> Señal de Reset (Activa Baja)
clk            -> Señal de Clock
EntradaRAM     -> Entrada de la RAM
SalidaRAM      -> Salida de la RAM
EntradaWREG    -> Entrada de WREG
SalidaWREG     -> Salida de WREG

BusA           -> Entrada A de la ALU
BusB           -> Entrada B de la ALU
BusC           -> Salida de la ALU
BusC_in        -> Salida de la ALU
PC             -> Contador de Programa
Data_in        -> Entrada de la RAM
CodOp          -> Código de Operación


ledORam        -> Habilita a leer la RAM
ledWRam        -> Habilita a escribir la RAM
ledOWreg       -> Habilita a leer el registro de trabajo
ledWWreg       -> Habilita a escribir el registro de trabajo
```

CASOS DE PRUEBA:

CASO DE PRUEBA 1:

Este caso de prueba se basa en instrucciones que ejecutan sobre el registro de trabajo (WREG), por ello en el componente de memoria de programa se cargaron las instrucciones a ejecutar, pero en cada uno de los casos el valor con el cual se modifica dicho registro son valores constantes.

; wreg con literales
nop
movlw 50
addlw 50
sublw 20
andlw 30
iorlw 40
xorlw 100
nop

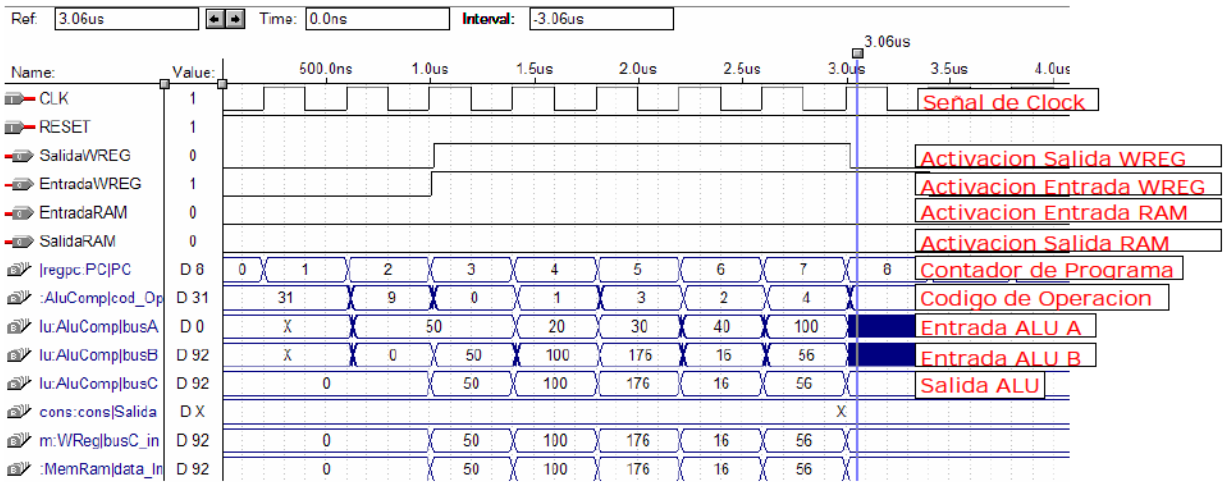


Figura 24. Resultado caso de prueba 1

En este caso los resultados obtenidos son correctos, en el cual se puede observar que los valores obtenidos se corresponden con los valores esperados como salida.

Además las señales correspondientes a la activación para lectura y escritura de los registros de memoria de datos (EntradaRAM y SalidaRAM) nunca son asignados, por ello que su valor se mantiene siempre en 0.

CASO DE PRUEBA 2:

En este caso de prueba se verifica el funcionamiento de la Unidad Aritmético Lógica, registro de estado de la Alu, registro de trabajo y generador de constantes. Las sentencias que se ejecutan, realizan sucesivas sumas sobre el registro de trabajo, obteniendo el dato desde el propio registro y reutilizando el valor obtenido en la siguiente operación.

Como primer paso se carga el registro (WREG) con un valor constante y luego este valor es utilizado para realizar sumas con acarreo y sin acarreo. Como ultima operación se limpia el valor del registro de trabajo.

```
; REGISTRO con operaciones a REGISTRO
;add
nop
movlw 50
addwf wreg,0
addwfc wreg,0
addwfc wreg,0
addwfc wreg,0
clrf wreg,0
nop
```

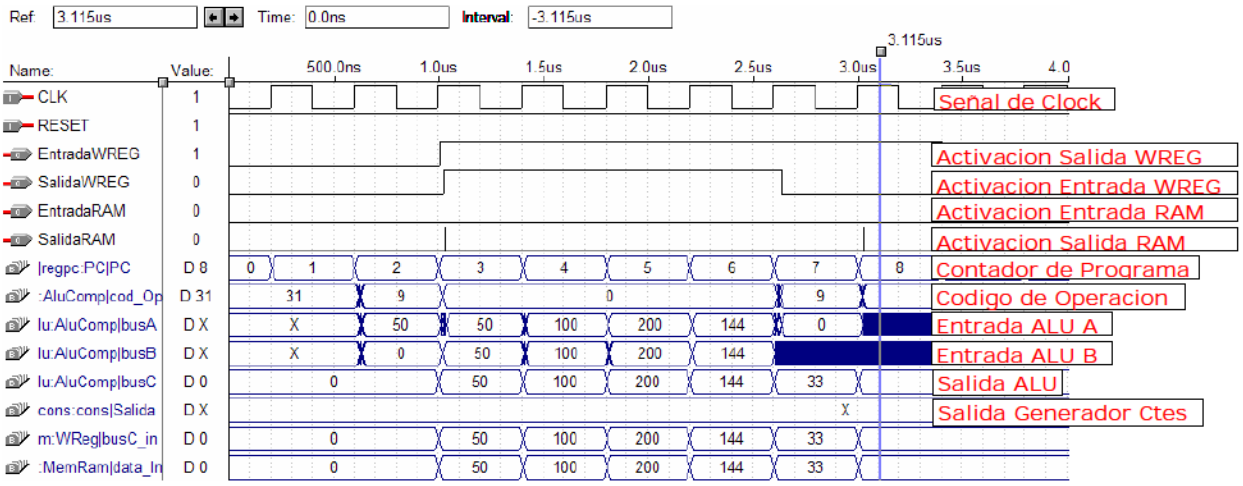


Figura 25. Resultado caso de prueba 2

El resultado es correcto, en el cual se observa que los resultados se ven acotados por el valor máximo del registro de trabajo (255) y generando el acarreo correspondiente para la próxima operación.

CASO DE PRUEBA 3:

En el siguiente caso de prueba se verifica el funcionamiento de la unidad de control, para verificar la activación de las distintas señales dependiendo de los componentes que intervienen, como así también se prueba el funcionamiento de la ALU con un conjunto de operaciones variadas.

Las operaciones se realizan entre registros, utilizando siempre el registro acumulador (WREG) para almacenar los valores intermedios.

```
;resta, set, inc, dec, complemento
nop
movlw 100
subwf wreg,0
setf wreg,0
incf wreg,0
decf wreg,0
comf wreg,0
nop
```

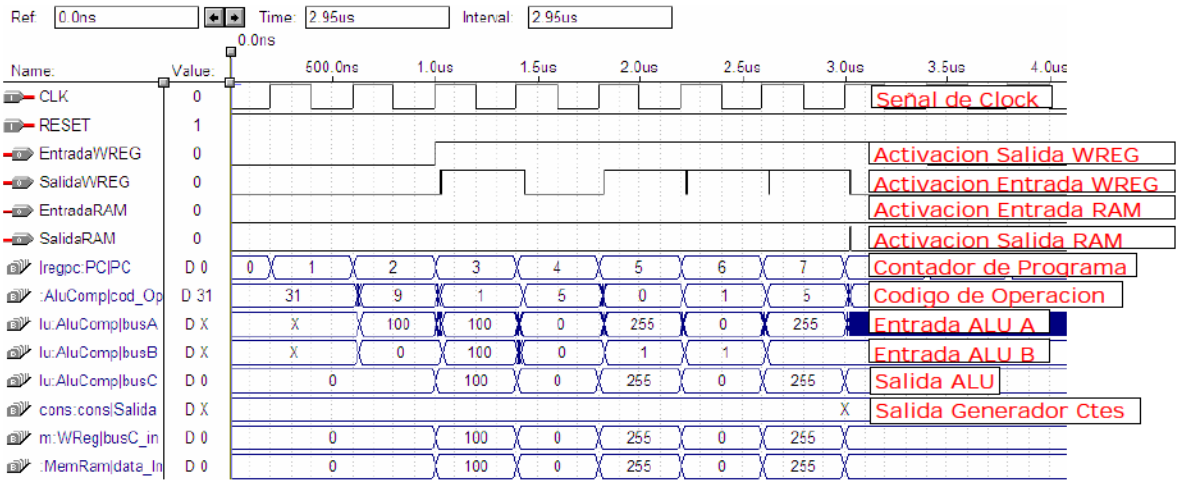


Figura 26. Resultado caso de prueba 3

Los valores obtenidos son correctos y el comportamiento de los distintos componentes fue el esperado. La activación de las distintas señales de control fue el correcto, donde se puede verificar las señales SalidaWREG y EntradaWREG que activan/desactivan la lectura/escritura del registro de trabajo dependiendo de la operación.

En la señal SalidaWREG puede observarse entre las operaciones 5 y 7 que haya cambios en la señal pero afectan el resultado final puesto los tiempos de Setup y Hold aseguran que en el momento que el WREG asigna internamente el valor de la señal su valor es el correcto.

CASO DE PRUEBA 4:

En esta oportunidad se verifica el funcionamiento de las operaciones de rotación en la unidad aritmético lógica, también se verifica el funcionamiento del registro de estado de la ALU, conjuntamente con el registro de trabajo.

En esta prueba se evalúan los valores almacenados en el registro de trabajo ejecutando operaciones de desplazamiento a izquierda y derecha del valor del registro teniendo. A su vez las operaciones de rotación pueden necesitar tener en cuenta en ciertos casos el bit de acarreo y por último se verifica también el funcionamiento de la condición lógica XOR.

; Instrucciones de Rotación

```
nop
movlw 180
rlcf wreg,0
rrcf wreg,0
rlncf wreg,0
rrncf wreg,0
xorwf wreg,0
nop
```

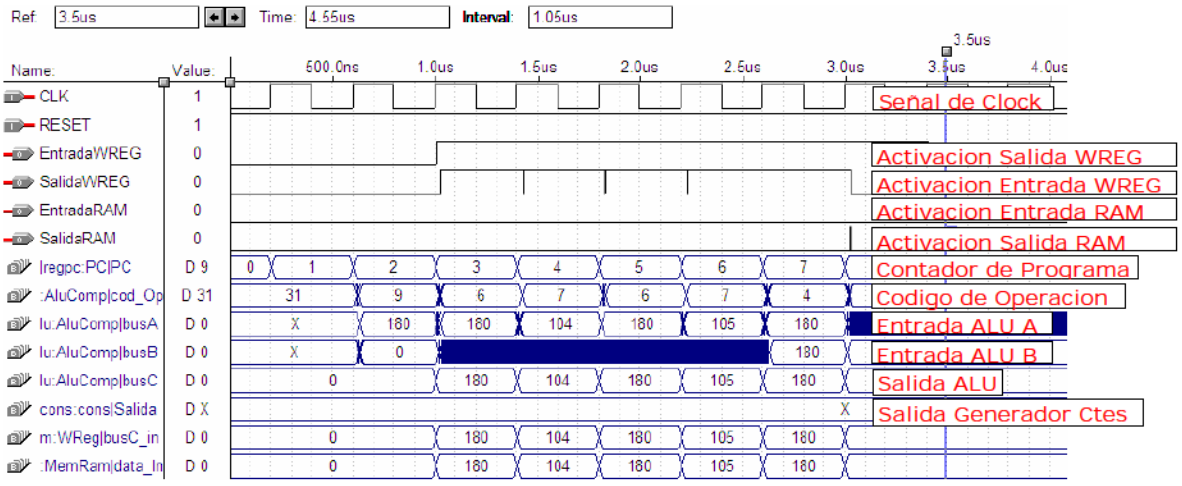


Figura 27. Resultado caso de prueba 4

El resultado obtenido fue el esperado pudiéndose verificar el funcionamiento de las operaciones de rotación y el funcionamiento del registro de estado de la ALU (ALUSTA) mediante la utilización del Bit de Carry para las operaciones de rotación con acarreo. Además, se pudo comprobar que estas operaciones utilizan solo el bus A de entrada a ALU. Se puede observar que el bus B únicamente tuvo datos válidos cuando se ejecutó la operación XOR.

CASO DE PRUEBA 5:

En este caso de prueba se verifica el funcionamiento de instrucciones que involucran una posición de memoria de datos y el registro de trabajo en una misma operación. Las operaciones trabajan sobre una misma posición de memoria con el fin de optimizar el espacio utilizado en la síntesis. Las operaciones que se utilizan son operaciones de movimiento de valores entre registro y posiciones de memoria y operaciones que afectan solo a la memoria de datos.

; RAM con operaciones a WReg
; Add
nop
movlw 98
movwf 18
addwf 18,1
addwfc 18,1
addwfc 18,1
addwfc 18,1
clrf 18,1
nop

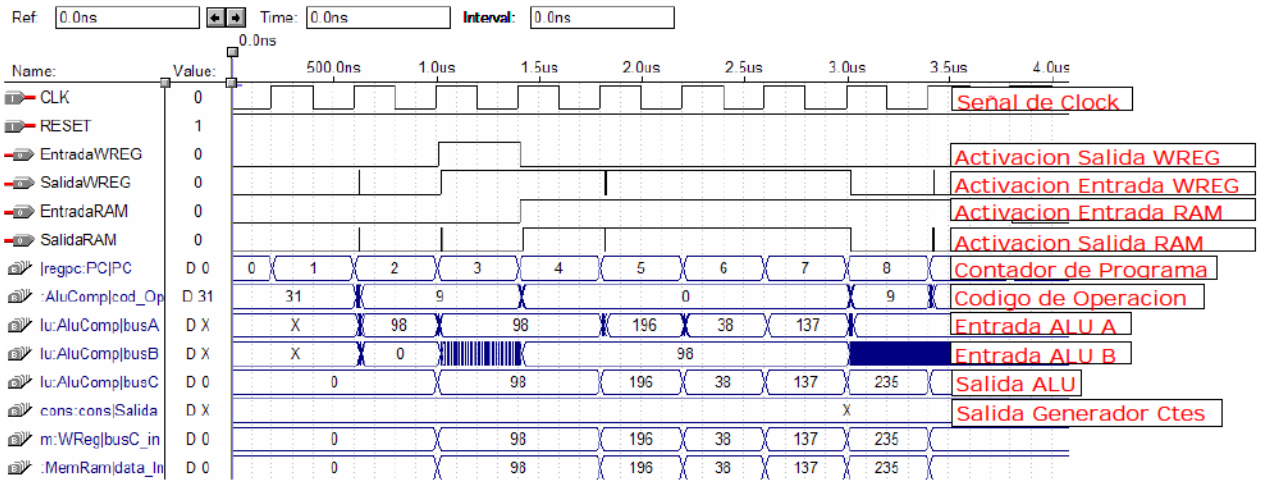


Figura 28. Resultado caso de prueba 5

El resultado se corresponde con los valores esperados, en el gráfico se puede observar que existe una sincronización entre el registro de trabajo y la memoria de datos para intercambiar los valores para poder realizar cada operación y que el valor obtenido en una operación puede ser utilizado en la instrucción siguiente.

Además permite comprobar el funcionamiento del registro de estado de la Alu cuando se involucran operaciones aritméticas que requieren la utilización del bit de acarreo.

Esta prueba también permite verificar el pipeline de instrucciones y la sincronización que efectúa la unidad de control con cada uno de los componentes

CASO DE PRUEBA 6:

En el siguiente caso de prueba, la memoria de programa se cargan instrucciones que permitan verificar el funcionamiento de las operaciones de resta, incremento y decremento de valores que trabajan con la memoria de datos y el registro de trabajo.

```
;resta, set, inc, dec, complemento
nop
movlw 98
movwf 18
subwf 18,1
setf 18,1
incf 18,1
decf 18,1
comf 18,1
nop
```

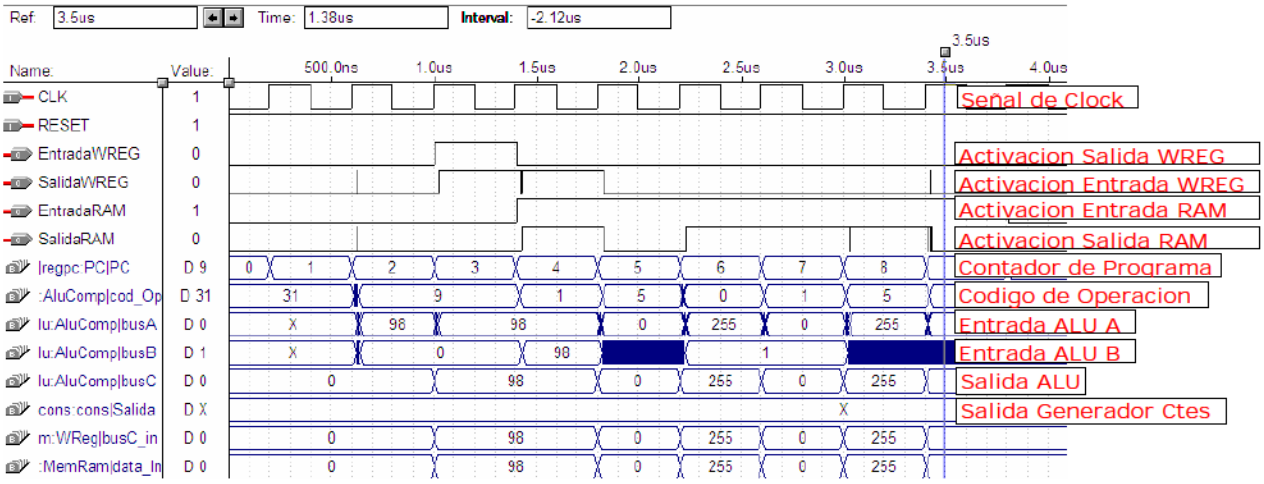


Figura 29. Resultado caso de prueba 6

Luego de la síntesis y simulación del modelo con estas instrucciones cargadas en la memoria de programa, se puede observar que se obtuvieron los resultados esperados.

Además puede observarse que las señales de control asociadas a la lectura y escritura del registro de trabajo se activaron únicamente cuando fue requerido trabajar con dicho registro. De forma similar ocurrió cuando se requirió trabajar con la memoria de datos.

CASO DE PRUEBA 7:

En este caso de prueba se intenta verificar el funcionamiento de los componentes que se ven afectados por las operaciones de rotación de datos que se encuentran almacenados en posiciones de memoria RAM. Para ello se carga el registro de trabajo con un valor constante y se transfiere dicho valor a una posición de la memoria RAM y luego se trabaja con esta última posición de la memoria de datos volviendo a utilizar la misma en cada una de las instrucciones.

; rotaciones con registros de RAM
nop
movlw 98
movwf 18
rlcf 18,1
rrcf 18,1
rlncf 18,1
rrncf 18,1
xorwf 18,1
nop

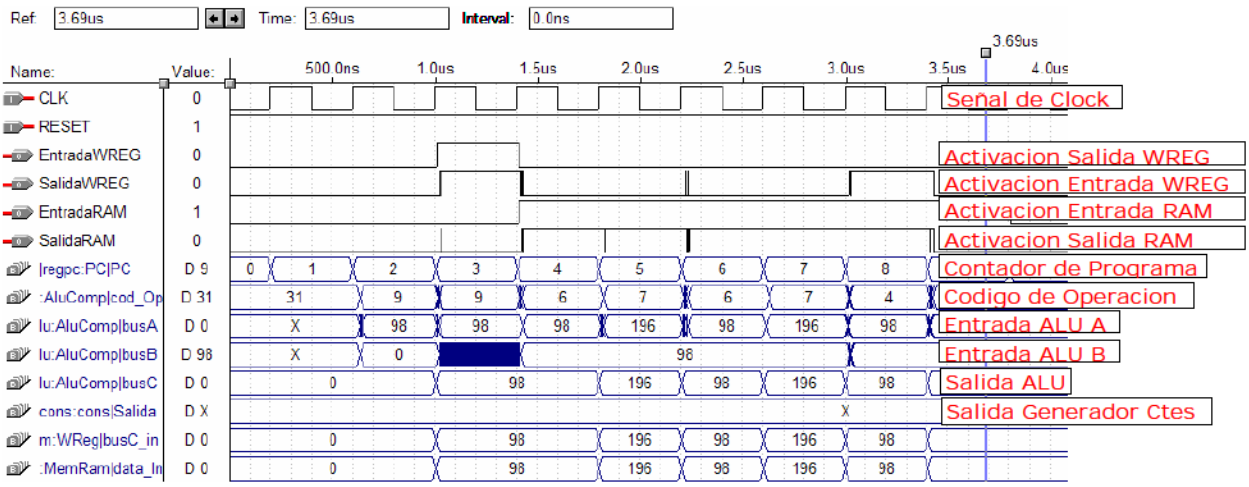


Figura 30. Resultado caso de prueba 7

Los valores obtenidos en esta etapa concuerdan con los objetivos deseados, lográndose una sincronización en la ejecución de las distintas instrucciones, pudiéndose utilizar el valor obtenido en una operación como entrada de la siguiente, verificándose de esta forma el funcionamiento del pipeline de instrucciones y la operación de las instrucciones de rotación sobre posiciones de memoria de datos.

CASO DE PRUEBA 8:

En este caso de prueba se intentará demostrar el funcionamiento de la memoria de datos y el registro de acumulador trabajando conjuntamente con operaciones lógicas. Para ello se ejecutarán instrucciones que permitirán cargar los registros con diferentes valores y luego realizar operaciones lógicas que requieran de ambos registros.

```
; Operaciones lógicas
nop
movlw 98
movwf 18
movlw 97
andwf 18,0
andwf 18,1
iorwf 18,0
iorwf 18,1
xorwf 18,0
xorwf 18,1
nop
```

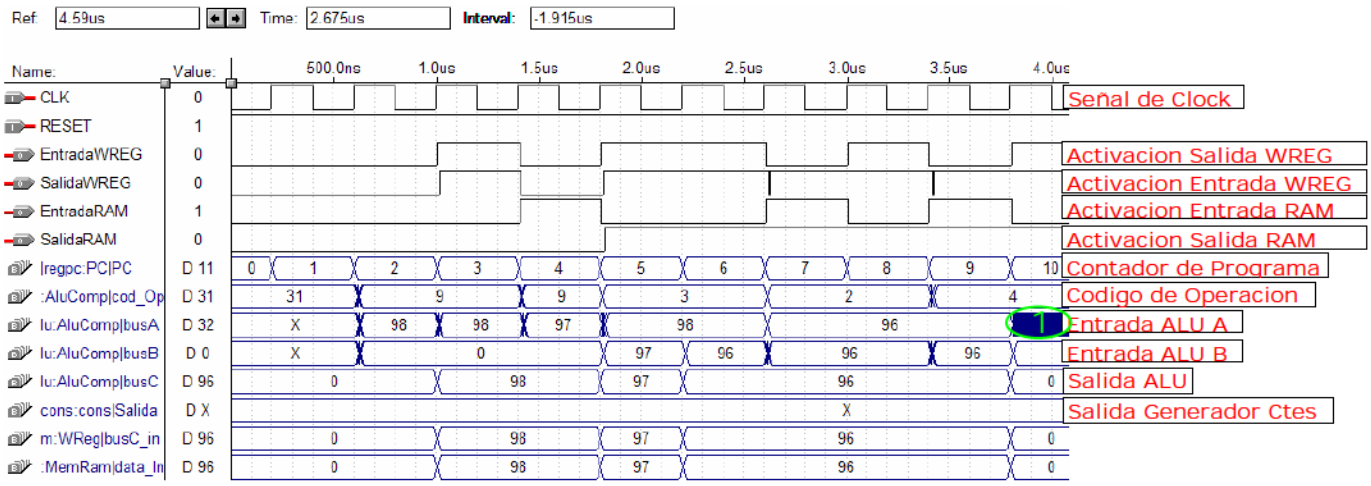


Figura 31. Resultado caso de prueba 8

Si observamos en el grafico superior, que muestra los resultados obtenidos durante la simulación del caso de prueba, se puede verificar que los valores obtenidos hacia el final de la ejecución no son correctos (1) debido a una desincronización de las señales de control de los diferentes componentes.

La desincronización se produce en el valor de entrada A a la Alu, por lo que se hicieron ajustes en el componente mulBusA que actúa como selector del valor entrada a la Alu, pero no se llego a un resultado correcto.

CASO DE PRUEBA 9:

En este punto se verificó el funcionamiento del microcontrolador durante la ejecución de operaciones que involucran a registros que guardan su valor en la memoria de datos y el registro de trabajo.

Las instrucciones que se ejecutan son operaciones de suma y resta alternadas sobre los registros anteriormente descriptos.

```
; mezcla RAM con wreg
nop
movlw 96
movwf 18
incf 18,1
decf wreg,1
incf 18,1
decf wreg,1
incf 18,1
decf wreg,1
nop
```

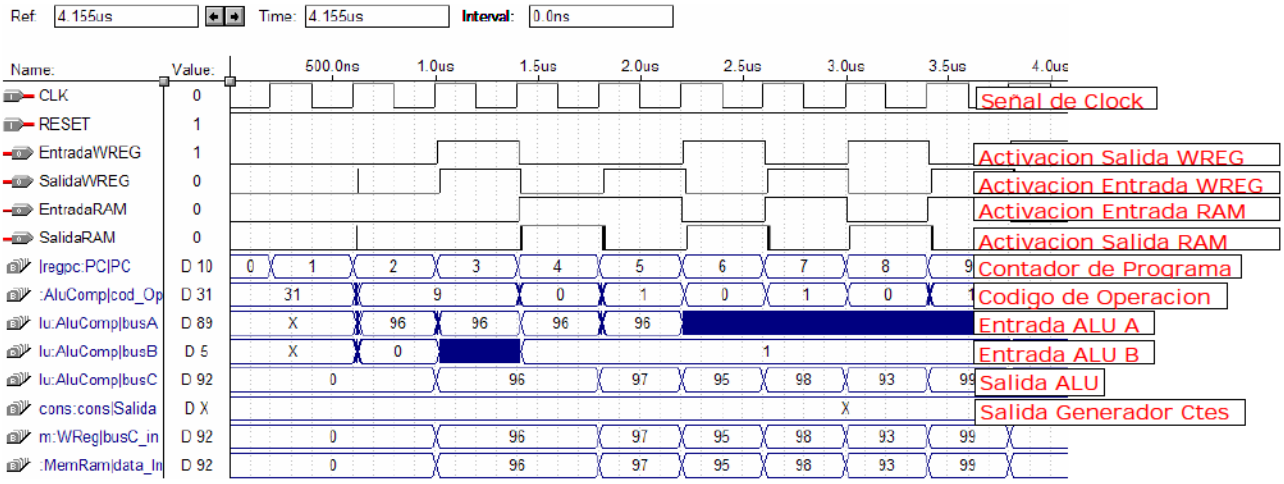


Figura 32. Resultado caso de prueba 9

Con características similares al caso de prueba anterior, durante la simulación del caso de prueba actual se verifico la existencia de interferencias entre distintas señales de activación en varios componentes, lo que produjo una salida errónea en los valores finales de la operación.

Debido a las dificultades que se describieron en los últimos casos de pruebas nos llevaron a realizar diferentes modificaciones en el diseño de distintos componentes, para intentar lograr que los valores se mantuvieran mas estables durante un periodo de tiempo asignados en las señales sea en la salida o entrada de un componente, pero en las pruebas realizadas nunca se llego a poder comprobar que el estas esperas (tiempos de HOLD y SETUP) se realizaran efectivamente.

Se realizaron una gran variedad de ajustes, para lo cual se realizaron las siguientes acciones:

- Se introdujeron tiempos de retardo en la activación y duración de las señales de control.

- Se diseñaron nuevos componentes que permitieran almacenar valores temporales para utilizar como buffers para los valores de las señales entre los distintos componentes.

- Se implementaron nuevas versiones de algunos componentes con diferentes arquitecturas internas.

Aunque se llevaron a cabo todos estos ajustes no se lograron eliminar todos los errores descriptos anteriormente debido a que surgieron nuevas complicaciones que se encuentran fuera del alcance de este trabajo debido que:

- Los tiempos de retardo introducidos en el modelo VHDL no se transportan al modelo simulado.

- Debido a que los retardos no se tienen en cuenta en el simulador se decidió realizar modificaciones en la arquitectura de ciertos componentes donde se producen interferencias entre señales pero aún así no se pudo eliminar la desincronización existente.

- Durante la simulación de los distintos casos de prueba ejecutados se encontraron algunos inconvenientes que se no corresponden con el modelo descripto que se esta sintetizando. Teniendo cargado en la memoria de programa un conjunto de instrucciones se realiza la síntesis y posterior simulación y se obtienen ciertos valores; a continuación, se vuelve a ejecutar una nueva síntesis y simulación del modelo VHDL con la única diferencia que en la salida visual del simulador se agrega una nueva señal para que esta pueda ser graficada y esperando obtener los mismos valores de salida, pero en el conjunto de datos obtenidos se observan errores, correspondiente a la interferencia entre señales del modelo.

Este tipo de inconvenientes han consumido gran parte del tiempo total de elaboración de este trabajo ya que para cada caso que se encontraba se realizaba una investigación para detectar el punto donde se producía el mismo y luego hacían numerosas pruebas para lograr solucionarlo y verificar su funcionamiento.

Conclusión

A lo largo del desarrollo de éste trabajo se logró realizar la descripción VHDL del microcontrolador PIC17 propuesta con todo lo que ello concierne, realizándose:

- Una jerarquía de componentes que conforman la estructura del microcontrolador descripto.

- La descripción de cada uno de los componentes incluidos en la arquitectura.

- La sincronización de cada componente según el tipo de instrucción a ejecutar.

Además se logró efectuar la síntesis del modelo VHDL mediante la herramienta Max Plus II provista en el University Program Design Laboratory Package de Altera, obteniéndose resultados dispares, tal como se puede comprobar en los casos de prueba mencionados.

En los casos donde se encontraron diferencias se hicieron ajustes al modelo de componentes para llegar a los resultados esperados.

Estas últimas tareas de ajustes y simulaciones han sido las que más tiempo consumieron en la realización de este trabajo debido a que los distintos cambios que se realizaban en la descripción para cada una de las correcciones que se intentaban hacer no se proyectaban luego en el modelo sintetizado.

Por otro lado, al momento de ejecutar cada una de las simulaciones, se observo que la carga de las instrucciones en la memoria de programa requerían de tiempo extra y era muy proclive a errores por lo que se desarrollo una herramienta que permite de una forma rápida la generación automática del módulo de ROM a partir de código assembler de las instrucciones a ejecutar.

Los resultados obtenidos durante el desarrollo del presente trabajo han sido satisfactorios y concuerdan con los resultados esperados en cuanto a poder realizar una descripción en VHDL de un microcontrolador existente en el mercado. Sin embargo, sería conveniente profundizar y hacer desarrollos en las siguientes tareas para trabajos a futuro:

- Multiplicador por hardware: algunos modelos pertenecientes a la gama de microcontroladores descritos en este trabajo poseen un multiplicador por hardware. Este desarrollo sería de mucha utilidad ya que simplifica muchos de las operaciones que debe ejecutar la unidad aritmético lógica. Este componente debería ser

descripto de forma tal que pueda ejecutar una instrucción por ciclo de reloj, algo indispensable en este tipo de microcontroladores.

- Conexión a periféricos: sería deseable que el modelo pueda interactuar con el mundo exterior o con otros dispositivos, para lo cual se debe profundizar en el desarrollo de módulos descriptos en VHDL que permitan establecer esta comunicación.

- Debido a los inconvenientes que surgieron en la síntesis y simulación del modelo con las herramientas descriptas, un trabajo de interés sería evaluar los ajustes necesarios para llevar a cabo la síntesis de esta descripción VHDL con otras herramientas de simulación.

Bibliografía

1. Keating M., Bricaud P., "Reuse Methodology Manual For System-On-A-Chip Designs, Second Edition", Kluwer Academic Publishers 1999, USA, ISBN 0-7923-8558-6.
2. Meerwein M. et al, "Linking Codesign and Reuse in Embedded Systems Design", Proc. Of the 8th. Intl. Workshop on Hardware/Software Codesign. San Diego, USA, May 2000, pp.93-97.
3. Seepold R, Martinez Madrid N. (Editores), "Virtual Components Design and Reuse", Kluwer Academic Publishers 2000, USA, ISBN 0-7923-7261-1.
4. Villagarcía H., Bria O., "Diseño de bloques IP: Programabilidad y Reutilización". WICC2001, San Luis, Argentina, May 2001, pp.2-5.
5. Pollard L.H., "Computer Design and Architecture", Prentice Hall 1990, USA, ISBN 0-13- 167255-X.
6. Smith M., "Application Specific Integrated Circuits", Addison Wesley 1997, USA, ISBN 0- 201-50022-1.
7. Wolf W., "Computer as Components: Principles of Embedded Computer Systems Design", Morgan Kaufmann 2000, USA, ISBN 1-55860-541-X.
8. ALTERA Corp., "NIOS Soft core Embedded Processor Data Sheet. Version 1". San José, CA, USA, 2000.
9. ALTERA Corp., "ARM-based Embedded Processor Device Overrview. Version 1.1", "MIPSbased Embedded Processor Device Overrview. Version 1.1". San José, CA, USA, 2000.
10. Usselmann R., "Open Cores SoC Bus Review", Rev.1.0, http://www.opencores.org, Jan. 2001.
11. ALTERA Corp., "Intellectual Property Catalog". San José, CA, USA, 1999.
12. Wolf W. "Computers as Componentes: Principles of Embedded Computer Systems Design", Morgan Kaufmann Publishers 2000, USA. ISBN 155860541X
13. Villar E. et al., "VHDL Lenguaje estándar de diseño electrónico", McGraw-Hill 1998, ESPAÑA. ISBN 84-481-1196-6.
14. Pardo, F. y Boluda J., "VHDL Lenguaje para síntesis y modelado de circuitos", Alfaomega 2000, MEXICO. ISBN 970-15-0443-7.

ANEXO I

```
-- ****************************************************************************
-- **   Descripcion en VHDL de una arquitectura RISC                         **
-- **                                                                        **
-- **   Autores: Leyes, Daniel Alejandro                                     **
-- **            Martinez Belot, Luis José Javier                            **
-- **                                                                        **
-- **   Modulo: InstruccionesASM                                             **
-- **                                                                        **
-- **   Modulo de constantes                                                 **
-- **                                                                        **
-- ****************************************************************************

 library ieee;
 use ieee.std_logic_1164.all;

 PACKAGE instruccionesAsm IS

  -- Tipos de datos utilizados para identificar los codigos de operacion
  SUBTYPE ANCHO_16 IS std_logic_vector(15 downto 0);
  SUBTYPE ANCHO_8  IS std_logic_vector(15 downto 8);
  SUBTYPE ANCHO_7  IS std_logic_vector(15 downto 9);
  SUBTYPE ANCHO_6  IS std_logic_vector(15 downto 10);
  SUBTYPE ANCHO_5  IS std_logic_vector(15 downto 11);
  SUBTYPE ANCHO_3  IS std_logic_vector(15 downto 13);

  -- Constantes para identificar los registros especiales
  CONSTANT INDF0    : std_Logic_Vector(7 downto 0) := "00000000";
  CONSTANT FSR0     : std_Logic_Vector(7 downto 0) := "00000001";
  CONSTANT PCL      : std_Logic_Vector(7 downto 0) := "00000010";
  CONSTANT PCLATH   : std_Logic_Vector(7 downto 0) := "00000011";
  CONSTANT ALUSTA   : std_Logic_Vector(7 downto 0) := "00000100";
  CONSTANT TOSTA    : std_Logic_Vector(7 downto 0) := "00000101";
  CONSTANT CPUSTA   : std_Logic_Vector(7 downto 0) := "00000110";
  CONSTANT INTSTA   : std_Logic_Vector(7 downto 0) := "00000111";
  CONSTANT INDF1    : std_Logic_Vector(7 downto 0) := "00001000";
  CONSTANT FSR1     : std_Logic_Vector(7 downto 0) := "00001001";
  CONSTANT WREG     : std_Logic_Vector(7 downto 0) := "00001010";
  CONSTANT TRM0L    : std_Logic_Vector(7 downto 0) := "00001011";
  CONSTANT TRM0H    : std_Logic_Vector(7 downto 0) := "00001100";
  CONSTANT TBLPTRL  : std_Logic_Vector(7 downto 0) := "00001101";
  CONSTANT TBLPTRH  : std_Logic_Vector(7 downto 0) := "00001110";
  CONSTANT BSR      : std_Logic_Vector(7 downto 0) := "00001111";


  -- Constantes para identificar las instrucciones
  CONSTANT NOP       : ANCHO_16 :=  "0000000000000000"; -- No realiza ninguna operacion
  CONSTANT RETURNN   : ANCHO_16 :=  "0000000000000010"; -- Retorno de subrrutina
  CONSTANT SLEEP     : ANCHO_16 :=  "0000000000000011"; -- Entra en modo SLEEP
  CONSTANT CLRWDT    : ANCHO_16 :=  "0000000000000100"; -- Limpia el timer del WatchDog
  CONSTANT RETFIE    : ANCHO_16 :=  "0000000000000101"; -- Retorno de interrupcion ( y
                                                           conecta interrupciones)
  CONSTANT MOVWF     : ANCHO_8  :=  "00000001"; -- Mueve WREG a F
  CONSTANT SUBWFB    : ANCHO_7  :=  "0000001";  -- Resta entre WREG y F con Borrow
  CONSTANT SUBWF     : ANCHO_7  :=  "0000010";  -- Resta entre WREG y F
  CONSTANT DECF      : ANCHO_7  :=  "0000011";  -- Decrementa F
  CONSTANT IORWF     : ANCHO_7  :=  "0000100";  -- O inclusivo entre WREG y F
  CONSTANT ANDWF     : ANCHO_7  :=  "0000101";  -- Suma WREG con F
  CONSTANT XORWF     : ANCHO_7  :=  "0000110";  -- O exclusivo entre WREG y F
  CONSTANT ADDWF     : ANCHO_7  :=  "0000111";  -- Suma WREG a F
  CONSTANT ADDWFC    : ANCHO_7  :=  "0001000";  -- Suma WREG y bit de Carry a F
  CONSTANT COMF      : ANCHO_7  :=  "0001001";  -- Complemento de F
  CONSTANT INCF      : ANCHO_7  :=  "0001010";  -- Incrementa F"
  CONSTANT DECFSZ    : ANCHO_7  :=  "0001011";  -- Decrementa F, salta si es 0"
  CONSTANT RRCF      : ANCHO_7  :=  "0001100";  -- Rota a derecha F, con Carry"
  CONSTANT RLCF      : ANCHO_7  :=  "0001101";  -- Rota a izquierda F, con Carry"
  CONSTANT SWAPF     : ANCHO_7  :=  "0001110";  -- Intercambia F"
  CONSTANT INCFSZ    : ANCHO_7  :=  "0001111";  -- Incrementa F, salta si es 0"
  CONSTANT RRNCF     : ANCHO_7  :=  "0010000";  -- Rota a derecha F, sin Carry"
  CONSTANT RLNCF     : ANCHO_7  :=  "0010001";  -- Rota a izquierda F, sin Carry"
  CONSTANT INFSNZ    : ANCHO_7  :=  "0010010";  -- Incrementa F, salta si no es 0"
  CONSTANT DCFSNZ    : ANCHO_7  :=  "0010011";  -- Decrementa F, salta si no es 0"
  CONSTANT CLRF      : ANCHO_7  :=  "0010100";  -- Limpia F, o limpia F y WREG "
```

```
  CONSTANT SETF      : ANCHO_7  :=  "0010101";  -- Setea F "
  CONSTANT NEGW      : ANCHO_7  :=  "0010110";  -- Niega WREG"
  CONSTANT DAW       : ANCHO_7  :=  "0010111";  -- Ajuste decimal de WREG"
  CONSTANT CPFSLT    : ANCHO_8  :=  "00110000"; -- Compara F con WREG, salta si F <
WREG"
  CONSTANT CPFSEQ    : ANCHO_8  :=  "00110001"; -- Compara F con WREG, salta si F =
WREG"
  CONSTANT CPFSGT    : ANCHO_8  :=  "00110010"; -- Compara F con WREG, salta si F >
WREG"
  CONSTANT TSTFSZ    : ANCHO_8  :=  "00110011"; -- Testea F, salta si es 0"
  CONSTANT MULWF     : ANCHO_8  :=  "00110100"; -- Multiplica WREG con F"
  CONSTANT BTG       : ANCHO_5  :=  "00111";    -- Invierte un Bit en F"
  CONSTANT MOVPF     : ANCHO_3  :=  "010";      -- Mueve P a F"
  CONSTANT MOVFP     : ANCHO_3  :=  "011";      -- Mueve F a P"
  CONSTANT BSF       : ANCHO_5  :=  "10000";    -- Setea un bit de F"
  CONSTANT BCF       : ANCHO_5  :=  "10001";    -- Limpia un bit de F"
  CONSTANT BTFSS     : ANCHO_5  :=  "10010";    -- Testea un Bit, salta si esta en 1"
  CONSTANT BTFSC     : ANCHO_5  :=  "10011";    -- Testea un Bit, salta si esta en 0"
  CONSTANT TLRD      : ANCHO_6  :=  "101000";   -- Lectura con Latch de tabla"
  CONSTANT TLWT      : ANCHO_6  :=  "101001";   -- Escritura con Latch de tabla"
  CONSTANT TABLRD    : ANCHO_6  :=  "101010";   -- Lectura de tabla"
  CONSTANT TABLWT    : ANCHO_6  :=  "101011";   -- Escritura de tabla"
  CONSTANT MOVLW     : ANCHO_8  :=  "10110000"; -- Mueve un valor literal a WREG"
  CONSTANT ADDLW     : ANCHO_8  :=  "10110001"; -- Suma un valor literal a WREG"
  CONSTANT SUBLW     : ANCHO_8  :=  "10110010"; -- Resta entre WREG y un valor literal"
  CONSTANT IORLW     : ANCHO_8  :=  "10110011"; -- O inclusivo entre un valor literal y
WREG"
  CONSTANT XORLW     : ANCHO_8  :=  "10110100"; -- O Exclusivo entre un valor literal y
WREG"
  CONSTANT ANDLW     : ANCHO_8  :=  "10110101"; -- And entre un valor literal y WREG"
  CONSTANT RETLW     : ANCHO_8  :=  "10110110"; -- Retorna valor literal a  WREG"
  CONSTANT LCALL     : ANCHO_8  :=  "10110111"; -- Llamado largo a subrrutina "
  CONSTANT MOVLB     : ANCHO_8  :=  "10111000"; -- Mueve un valor literal al nibble bajo
de BSR"
  CONSTANT MOVLR     : ANCHO_7  :=  "1011101";  -- Mueve un valor literal al nibble alto
de BSR"
  CONSTANT MULLW     : ANCHO_8  :=  "10111100"; -- Multiplica un valor literal con WREG"
  CONSTANT GOTO       : ANCHO_3  :=  "110";      -- Salto incondicional"
  CONSTANT CALL       : ANCHO_3  :=  "111";      -- Llamado a subrrutina"

  CONSTANT MC_SUMA   : ANCHO_5  :=  "00000";
  CONSTANT MC_RESTA  : ANCHO_5  :=  "00001";
  CONSTANT MC_OR     : ANCHO_5  :=  "00010";
  CONSTANT MC_AND    : ANCHO_5  :=  "00011";
  CONSTANT MC_XOR    : ANCHO_5  :=  "00100";
  CONSTANT MC_COMP   : ANCHO_5  :=  "00101";
  CONSTANT MC_ROL    : ANCHO_5  :=  "00110";
  CONSTANT MC_ROR    : ANCHO_5  :=  "00111";
  CONSTANT MC_SWAP   : ANCHO_5  :=  "01000";
  CONSTANT MC_MUEVE  : ANCHO_5  :=  "01001";
  CONSTANT MC_BITC   : ANCHO_5  :=  "01010";
  CONSTANT MC_BITS   : ANCHO_5  :=  "01011";
  CONSTANT MC_NEG    : ANCHO_5  :=  "01100";
  CONSTANT MC_BITT   : ANCHO_5  :=  "01101";
  CONSTANT MC_DAW    : ANCHO_5  :=  "01110";
  CONSTANT MC_SKIPE  : ANCHO_5  :=  "01111";
  CONSTANT MC_SKIPG  : ANCHO_5  :=  "10000";
  CONSTANT MC_SKIPL  : ANCHO_5  :=  "10001";

  CONSTANT MC_NOP    : ANCHO_5  :=  "11111";

  CONSTANT SELECT_A : std_logic := '0';
  CONSTANT SELECT_B : std_logic := '1';

  CONSTANT WAIT_ASSIGN: time := 0 ns;

  END instruccionesAsm;
```

```
-- ********************************************************************************
-- **   Descripcion en VHDL de una arquitectura RISC                             **
-- **                                                                            **
-- **   Autores: Leyes, Daniel Alejandro                                         **
-- **            Martinez Belot, Luis José Javier                                **
-- **                                                                            **
-- **   Modulo: PIC17                                                            **
-- **                                                                            **
-- **   Modulo principal del proyecto que interconecta los                       **
-- **   componentes de la arquitectura PIC17                                     **
-- **                                                                            **
-- ********************************************************************************

 LIBRARY IEEE;
 USE IEEE.STD_LOGIC_1164.all;
 USE work.instruccionesasm.all;

  ENTITY Pic17 IS
  PORT ( clk: in STD_Logic;
         reset: in STD_Logic;
         ledoram: out std_logic;
         ledwram: out std_logic;
                          ledowreg: out std_logic;
                          ledwwreg: out std_logic;
         dataA: out std_logic_vector(7 downto 0);
         dataB: out std_logic_vector(7 downto 0);
         dataC: out std_logic_vector(7 downto 0);
         dataD: out std_logic_vector(7 downto 0)

       );
 END Pic17;

 ARCHITECTURE RTL1 OF Pic17 IS

   COMPONENT UnidadControl
     PORT (Clock                      : IN STD_LOGIC;
           Z, C, DC, OV, C_old        : OUT STD_LOGIC;
           ToSignal, PdSignal, Glintd : OUT STD_LOGIC;
           o_wreg_a, o_wreg_b         : OUT STD_LOGIC;
           o_imm_en, imm_select       : OUT STD_LOGIC;
           i_wreg_SALIDA              : OUT STD_LOGIC;
           pc_o, pclath_o             : OUT STD_LOGIC;
           cpusta_o, memdato_o        : OUT STD_LOGIC;
           pc_w, pclath_w             : OUT STD_LOGIC;
           cpusta_w, memdato_w        : OUT STD_LOGIC;
           GC_en0, GC_en1, GC_enF     : OUT STD_LOGIC;
           GC_select                  : OUT STD_LOGIC;
           skip                       : IN STD_LOGIC;
           CodOp                      : IN STD_LOGIC_VECTOR (15 downto 0);
           CodALU                     : OUT STD_LOGIC_VECTOR (4 downto 0)
      );
   END COMPONENT;

   COMPONENT Alu
     PORT (Reset      : IN STD_LOGIC;
           Clock      : IN STD_LOGIC;
           c          : IN STD_LOGIC;
           z          : IN STD_LOGIC;
           ov         : IN STD_LOGIC;
           dc         : IN STD_LOGIC;
           c_old_en: IN STD_LOGIC;
           c_en       : IN STD_LOGIC;
           z_en       : IN STD_LOGIC;
           ov_en      : IN STD_LOGIC;
           dc_en      : IN STD_LOGIC;
           cod_Op     : IN STD_LOGIC_VECTOR (4 downto 0);
           busA       : IN STD_LOGIC_VECTOR (7 downto 0);
           busB       : IN STD_LOGIC_VECTOR (7 downto 0);
           busC       : OUT STD_LOGIC_VECTOR (7 downto 0);
           skip       : OUT STD_LOGIC;
           c_Out      : OUT STD_LOGIC;
           z_Out      : OUT STD_LOGIC;
           ov_Out     : OUT STD_LOGIC;
           dc_Out     : OUT STD_LOGIC
         );
    END COMPONENT;
```

```
COMPONENT RAM
 PORT(     Reset     : IN STD_LOGIC;
           Clock     : IN std_logic;
           Out_En    : IN STD_LOGIC;
           Write_En  : IN STD_LOGIC;
           FSN_In    : IN  std_logic_vector (15 downto 0);
           data_In   : IN  std_logic_vector (7 downto 0);
           data_Out  : OUT std_logic_vector (7 downto 0)
         );
END COMPONENT;

   COMPONENT ROM
    PORT( pc_in     : IN  STD_LOGIC_VECTOR(15 DOWNTO 0);
       inst_out  : OUT STD_LOGIC_VECTOR(15 DOWNTO 0)
        );
   END COMPONENT;

   COMPONENT RegPC
    PORT( clock        : IN  STD_LOGIC;
          reset        : IN  STD_LOGIC;
          skip         : IN  STD_LOGIC;
          pcl_out_en   : IN  STD_LOGIC;
          pch_out_en   : IN  STD_LOGIC;
          pcl_write_en : IN  STD_LOGIC;
          pch_write_en : IN  STD_LOGIC;
          load         : IN  STD_LOGIC;
          data_in      : IN  STD_LOGIC_VECTOR( 7 DOWNTO 0);
          addr_in      : IN  STD_LOGIC_VECTOR(15 DOWNTO 0);
          data_out     : OUT STD_LOGIC_VECTOR( 7 DOWNTO 0);
          addr_out     : OUT STD_LOGIC_VECTOR(15 DOWNTO 0)
        );
   END COMPONENT;

   COMPONENT regAcum
    PORT (clock       : IN  STD_LOGIC;
                selectA       : IN  STD_LOGIC;
                selectB       : IN  STD_LOGIC;
                read_en     : IN  STD_LOGIC;
                busC_in     : IN  STD_LOGIC_VECTOR( 7 DOWNTO 0 );
                Bus_out     : OUT STD_LOGIC_VECTOR( 7 DOWNTO 0 )
                );
   END COMPONENT;

COMPONENT ALUSTA IS
 PORT ( clock: IN STD_LOGIC;
                           C   : IN STD_LOGIC;
                  OV : IN STD_LOGIC;
                  DC : IN STD_LOGIC;
                  Z  : IN STD_LOGIC;
                  F  : IN STD_LOGIC_VECTOR (3 DOWNTO 0);
                  C_Out  : OUT STD_LOGIC;
                  OV_Out : OUT STD_LOGIC;
                  DC_Out : OUT STD_LOGIC;
                  Z_Out  : OUT STD_LOGIC;
                  F_Out  : OUT STD_LOGIC_VECTOR (3 DOWNTO 0)
                     );
END COMPONENT;

COMPONENT regInst IS
 PORT ( clock        : IN  STD_LOGIC;
        reset        : IN  STD_LOGIC;
        inm_out_en   : IN  STD_LOGIC;
        inm_Select   : IN  STD_LOGIC;
        skip         : IN  STD_LOGIC;
        inst_in      : IN  STD_LOGIC_VECTOR(15 DOWNTO 0);
        inst_out     : OUT STD_LOGIC_VECTOR(15 DOWNTO 0);
        busA_out     : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
        busB_out     : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
      );
END COMPONENT;

COMPONENT MuxBusB IS
 PORT (CLOCK     : IN  STD_LOGIC;
    WREG_data : IN STD_LOGIC_VECTOR(7 downto 0);
    RI_data   : IN STD_LOGIC_VECTOR(7 downto 0);
    GC_data   : IN STD_LOGIC_VECTOR(7 downto 0);
    WREG_en   : IN  STD_LOGIC;
```

```
        RI_en     : IN  STD_LOGIC;
        GC_en     : IN  STD_LOGIC;
                          Data_OutB : inOUT STD_LOGIC_VECTOR(7 downto 0)
                            );
    END COMPONENT;

    COMPONENT MuxBusA
     PORT(
           CLK        : IN STD_LOGIC;
           WREG_data  : IN STD_LOGIC_VECTOR(7 downto 0);
           RI_data    : IN STD_LOGIC_VECTOR(7 downto 0);
           RAM_data   : IN STD_LOGIC_VECTOR(7 downto 0);
           PC_data    : IN STD_LOGIC_VECTOR(7 downto 0);
           GC_data    : IN STD_LOGIC_VECTOR(7 downto 0);
           WREG_en    : IN STD_LOGIC;
           RI_en      : IN STD_LOGIC;
           RAM_en     : IN STD_LOGIC;
           PC_en      : IN STD_LOGIC;
           GC_en      : IN STD_LOGIC;
           Data_OutA  : OUT STD_LOGIC_VECTOR(7 downto 0)
       );
    END COMPONENT;

    COMPONENT genCons
     PORT (
         gen0, gen1, genFF    : IN  STD_LOGIC;
         Salida               : OUT STD_LOGIC_VECTOR(7 downto 0)
         );
     END COMPONENT;


--Señales internas para conectar las entradas del multiplexor del busB con la ALU
signal WREG_MUX  :  std_Logic_vector(7 downto 0);
signal RI_MUXB   :  std_Logic_vector(7 downto 0);
signal MUXB_ALU  :  std_Logic_vector(7 downto 0);
--Señales internas para conectar las entradas del multiplexor del busA con la ALU
signal WREG_MUXA :  std_Logic_vector(7 downto 0);
signal RI_MUXA   :  std_Logic_vector(7 downto 0);
signal PC_MUXA   :  std_Logic_vector(7 downto 0);
signal RAM_MUXA  :  std_Logic_vector(7 downto 0);
signal MUXA_ALU  :  std_Logic_vector(7 downto 0);

--Señales internas para conectar el PC y la ROM
signal PC_ROM_addr : std_Logic_vector(15 downto 0);
--Señales internas para conectar la ROM y el RI
signal ROM_RI_inst : std_Logic_vector(15 downto 0);
--Señales internas para conectar el RI y el PC
signal RI_Inst_Out : std_Logic_vector(15 downto 0);

--Señales internas para conectar la UC con el WREG
signal UC_WREG_write_en: STD_Logic;
signal UC_WREG_read_en : STD_Logic;
signal UC_WREG_selBus  : STD_Logic;
--Señales internas para conectar la UC con el PC
signal UC_PCL_out_en   : STD_LOGIC;
signal UC_pch_out_en   : STD_LOGIC;
signal UC_pcl_write_en : STD_LOGIC;
signal UC_pch_write_en : STD_LOGIC;
signal UC_PC_load          : STD_LOGIC;
--Señales internas para conectar la ALU con el reg de estado de la ALU
signal ALU_ALUSTA_C  : STD_LOGIC;
signal ALU_ALUSTA_OV : STD_LOGIC;
signal ALU_ALUSTA_DC : STD_LOGIC;
signal ALU_ALUSTA_Z  : STD_LOGIC;
--Señales internas para conectar  el reg de estado de la ALU con la ALU
signal ALUSTA_ALU_C  : STD_LOGIC;
signal ALUSTA_ALU_OV : STD_LOGIC;
signal ALUSTA_ALU_DC : STD_LOGIC;
signal ALUSTA_ALU_Z  : STD_LOGIC;
--otras señales
Signal ALU_BusC: std_Logic_vector(7 downto 0);
--Signal ALU_BusC_in: std_Logic_vector(7 downto 0);
Signal UC_ALU_C_OLD:STD_Logic;
Signal UC_ALU_C  :STD_Logic;
Signal UC_ALU_Z  :STD_Logic;
Signal UC_ALU_OV :STD_Logic;
Signal UC_ALU_DC :STD_Logic;
```

```
Signal UC_RAM_W :STD_Logic;
Signal UC_RAM_O :STD_Logic;
Signal UC_RI_imm_out :STD_Logic;
Signal UC_RI_imm_sel :STD_Logic;

Signal cod_PC_ALU: std_Logic_vector(15 downto 0);
Signal cod_UC_ALU: std_Logic_vector(4 downto 0);
signal intFSN : std_Logic_vector(3 downto 0):="0000";

Signal UC_GC_selBusB : std_logic;
Signal UC_GC_selBusA : std_logic;
Signal UC_GC_gen0    : std_logic;
Signal UC_GC_gen1    : std_logic;
Signal UC_GC_genFF   : std_logic;
Signal UC_GC_select  : std_logic;
Signal GC_MUX       : std_logic_vector(7 downto 0);
Signal cero          :std_logic;
Signal UC_WREG_B: std_logic;
Signal UC_RI_write_enB  : std_logic;
Signal UC_WREG_A: std_logic;
Signal UC_RI_write_enA  : std_logic;
Signal UC_PC_write_enA  : std_logic;
signal uc_ledOn: std_logic;
signal ALU_UC_SKIP  : STD_LOGIC;

begin
   cero<='0';

   UC_RI_write_enA   <= '1' WHEN (UC_RI_imm_sel='0')      AND(UC_RI_imm_out='1')
                        ELSE '0';

   UC_GC_selbusA     <= '1' WHEN ((UC_GC_Gen0 = '1') OR (UC_GC_Gen1 = '1') OR
                         (UC_GC_Genff = '1'))  AND (UC_GC_select = SELECT_A)
                      ELSE '0';

  -- Instanciacion del componente de selector de datos para la entrada A de la ALU
  MulBusA  : MuxBusA port map (clk     => clk,
                       WREG_data => WReg_Mux,
                       RI_data   => RI_MuxA,
                       RAM_data  => RAM_MuxA,
                       PC_data   => PC_MuxA,
                       WREG_en   => UC_WREG_A,
                       RI_en    => UC_RI_write_enA,
                       RAM_en    => UC_RAM_O,
                       PC_en    => UC_PCL_OUT_EN,
                       GC_Data   => GC_MUX,
                       GC_en     => UC_GC_selBusA,
                       Data_OutA => MuxA_ALU
                      );

   UC_GC_selbusB     <= '1' WHEN ((UC_GC_Gen0 = '1') OR (UC_GC_Gen1 = '1')
                                 OR (UC_GC_Genff = '1'))AND(UC_GC_select = SELECT_B)
                        ELSE '0';
   UC_RI_write_enB   <= UC_RI_imm_sel      OR UC_RI_imm_out;

   -- Instanciacion del componente de selector de datos para la entrada B de la ALU
   MulBusB  : MuxBusB port map (clock         => clk,
                           WREG_data => WReg_Mux,
                           RI_data   => RI_MuxB,
                           GC_Data   => GC_MUX,
                           GC_en     => UC_GC_selBusB,
                           WREG_en   => UC_WREG_B,
                           RI_en      => UC_RI_write_enB,
                           Data_OutB => MuxB_ALU
                         );



  -- Instanciacion del componente Generador de constantes

  cons : genCons port map ( gen0 => UC_GC_gen0,
                       gen1 => UC_GC_gen1,
                       genFF => UC_GC_genFF,
                       Salida => GC_MUX
                      );
```

```
-- Instanciacion del componente de contador de programa

PC                  : RegPC port map (reset => reset,
                                      clock    => clk,
                                      skip     => ALU_UC_SKIP,
                                      pcl_out_en   => UC_pcl_out_en,
                                      pch_out_en   => UC_pch_out_en,
                                      pcl_write_en => UC_pcl_write_en,
                                      pch_write_en => UC_pch_write_en,
                                      load         => Cero,
                                      data_in      => ALU_BusC,
                                      addr_in      => RI_Inst_Out,
                                      data_out     => PC_MUXA,
                                      addr_out     => PC_ROM_addr
                                    );

-- Instanciacion del componente de memoria de programa

Prog                : ROM port map (pc_in   => PC_ROM_addr,
                                    inst_out=> ROM_RI_inst
                                     );

-- Instanciacion del componente de registro de estado de la alu

EstadoAlu: ALUSTA port map (clock    => clk,
                            C      => ALU_ALUSTA_C,
                            OV     => ALU_ALUSTA_OV,
                            DC     => ALU_ALUSTA_DC,
                            Z      => ALU_ALUSTA_Z,
                            F      => INTfSN,
                            C_Out  => ALUSTA_ALU_C,
                            OV_Out => ALUSTA_ALU_OV,
                            DC_Out => ALUSTA_ALU_DC,
                            Z_Out  => ALUSTA_ALU_Z,
                            F_Out  => INTfSN
                           );

-- Instanciacion del componente de la unidad aritmetico logica

AluComp             : ALU port map (clock  => clk,
                                    reset  => reset,
                                    cod_OP => cod_UC_ALU,
                                    busA   => MuxA_ALU,
                                    busB   => MuxB_ALU,
                                    busC   => ALU_BusC,
                                    c      => ALUSTA_ALU_C,
                                    z      => ALUSTA_ALU_Z,
                                    ov     => ALUSTA_ALU_OV,
                                    dc         => ALUSTA_ALU_DC,
                                    c_old_en=> UC_ALU_C_OLD,
                                    c_en   => UC_ALU_C,
                                    z_en   => UC_ALU_Z,
                                    ov_en  => UC_ALU_OV,
                                    dc_en  => UC_ALU_DC,
                                    c_out  => ALU_ALUSTA_C,
                                    z_out  => ALU_ALUSTA_Z,
                                    ov_out => ALU_ALUSTA_OV,
                                    dc_out => ALU_ALUSTA_DC,
                                    skip   => ALU_UC_SKIP
                                    );

-- Instanciacion del componente de unidad de control

UC     : UnidadControl port map (  clock  => clk,
                                   c  => UC_ALU_C,
                                   z  => UC_ALU_Z,
                                   ov => UC_ALU_OV,
                                   dc => UC_ALU_DC,
                                   c_old => UC_ALU_C_OLD,
                                   memdato_w => UC_RAM_W,
                                   memdato_o => UC_RAM_O,
                                   codOP     => RI_Inst_Out,
                                   codALU    => cod_UC_ALU,
                                   O_wreg_a  => UC_WREG_A,
                                   o_wreg_b  => UC_WREG_B,
                                   i_wreg_SALIDA    => UC_WREG_read_en,
```

```
                              GC_en0     => UC_GC_gen0,
                              GC_en1     => UC_GC_gen1,
                              GC_enF     => UC_GC_genFF,
                              GC_select  => UC_GC_select,
                              pc_o       => UC_pcl_out_en,
                              pclath_o   => UC_pch_out_en,
                              pc_w       => UC_pcl_write_en,
                              pclath_w   => UC_pch_write_en,
                              o_imm_en   => UC_RI_imm_out,
                              imm_select=> UC_RI_imm_sel,
                              skip       => ALU_UC_SKIP
                          );

   -- Instanciacion del componente de memoria de datos

   MemRam : RAM port map (    clock    => clk,
                         reset    => Reset,
                         Out_en   => UC_RAM_O,
                         Write_en => UC_RAM_W,
                         FSN_in   => RI_Inst_Out,
                         Data_IN  => ALU_BusC,
                         data_out => RAM_MuxA
                          );

   -- Instancion del componente de registro de trabajo

   WReg   : RegAcum port map(clock     => clk,
                          selectA   => UC_WREG_A,
                          selectB   => UC_WREG_B,
                          read_en   => UC_WREG_read_en,
                          busC_in   => ALU_BusC,
                          bus_out   => WREG_mux
                         );

   -- Instancion del componente de registro de instruccion

   RI     : RegInst port map (clock      => clk,
                           reset      => Reset,
                           inst_in    => ROM_RI_inst,
                           busA_out     => RI_MuxA,
                           busB_out     => RI_MuxB,
                           inm_out_en => UC_RI_imm_out,
                           inm_Select => UC_RI_imm_sel,
                           skip     => cero,
                           inst_out   => RI_Inst_Out
                          );

   -- Instancion de la señales para poder ser visualizadas en el simulador

   dataA(7 downto 0) <= MuxB_ALU;
   dataD(7 downto 0) <= MuxA_ALU;

   dataB(7 downto 0) <= RAM_MuxA;
   dataC(7 downto 0) <= WREG_MUX;

   ledowreg <= UC_WREG_A;
   ledwwreg <= UC_WREG_read_en;
   ledoram  <= UC_RAM_O;
   ledwram  <= UC_RAM_W;

   end RTL1;


-- ***************************************************************************
-- **   Descripcion en VHDL de una arquitectura RISC                        **
-- **                                                                       **
-- **   Autores: Leyes, Daniel Alejandro                                    **
-- **            Martinez Belot, Luis José Javier                           **
-- **                                                                       **
-- **   Modulo: Unidad Aritmetico Logica                                    **
-- **                                                                       **
-- **   Modulo que describe el comportamiento de la ALU del PIC17           **
-- **                                                                       **
-- ***************************************************************************

LIBRARY ieee;
LIBRARY work;
```

```
USE ieee.std_logic_1164.ALL;
USE IEEE.std_logic_arith.all;
USE IEEE.std_logic_signed.all;
USE work.instruccionesAsm.ALL;

ENTITY ALU IS
  PORT (
        Reset               : IN STD_LOGIC;
        Clock               : IN STD_LOGIC;
        c                   : IN STD_LOGIC;
        z                   : IN STD_LOGIC;
        ov                  : IN STD_LOGIC;
        dc                  : IN STD_LOGIC;
        c_old_en            : IN STD_LOGIC;
        c_en                : IN STD_LOGIC;
        z_en                : IN STD_LOGIC;
        ov_en               : IN STD_LOGIC;
        dc_en               : IN STD_LOGIC;
        cod_Op              : IN  STD_LOGIC_VECTOR (4 downto 0);
        busA                : IN  STD_LOGIC_VECTOR (7 downto 0);
        busB                : IN  STD_LOGIC_VECTOR (7 downto 0);
        busC                : OUT STD_LOGIC_VECTOR (7 downto 0);
        skip                : OUT STD_LOGIC;
        c_Out               : OUT STD_LOGIC;
        z_Out               : OUT STD_LOGIC;
        ov_Out              : OUT STD_LOGIC;
        dc_Out              : OUT STD_LOGIC
        );
end ALU;

Architecture RTL of ALU is
                      signal nroBit  : std_logic_vector (7 downto 0);
                      signal busTemp : std_logic_vector (8 downto 0):= "000000000";
begin

     nroBit <= "00000001" when busB(2 DOWNTO 0) = "000" else
               "00000010" when busB(2 DOWNTO 0) = "001" else
               "00000100" when busB(2 DOWNTO 0) = "010" else
               "00001000" when busB(2 DOWNTO 0) = "011" else
               "00010000" when busB(2 DOWNTO 0) = "100" else
               "00100000" when busB(2 DOWNTO 0) = "101" else
               "01000000" when busB(2 DOWNTO 0) = "110" else
               "10000000" when busB(2 DOWNTO 0) = "111" else
               "00000000";


   process (clock,reset)
   begin
      if reset='0' then
       --limpiar ALU
      elsif clock'event and clock='1' then
       skip <= '0';
      CASE cod_OP is
         when MC_SUMA  => if (c_old_en = '0')
                            then busTemp <= ('0' & busA) + ('0' & busB);
                            else busTemp <= ('0' & busA) + ('0' & busB) + c;
                          end if;
         when MC_RESTA => if (c_old_en = '0')
                             then busTemp <= ('0' & busA) - ('0' & busB);
                             else busTemp <= ('0' & busA) - ('0' & busB) - (not c);
                          end if;
         when MC_NEG   => busTemp <= ('0' & not busB) + "0000000001";
         when MC_MUEVE => busTemp(7 downto 0) <= busA;
         when MC_AND   => busTemp(7 downto 0) <= busA AND busB;
         when MC_OR    => busTemp(7 downto 0) <= busA OR busB;
         when MC_XOR   => busTemp(7 downto 0) <= busA XOR busB;
         when MC_ROR   => if (c_old_en = '0')
                             then busTemp(7 downto 0) <= busA(0) & busA(7 DOWNTO 1);
                             else  busTemp <= busA(0) & c &  busA(7 DOWNTO 1);
                          end if;
         when MC_ROL   => if (c_old_en = '0')
                             then busTemp(7 downto 0) <= busA(6 DOWNTO 0) & busA(7);
                             else  busTemp <= busA(7 DOWNTO 0) & c;
                          end if;
         when MC_DAW   => if (busB(3 downto 0) > "1001") OR (dc = '1')
                             then busTemp(3 downto 0) <= busB(3 downto 0) + 6;
                             else busTemp(3 downto 0) <= busB(3 downto 0);
```

```
                        end if;
                        If (busB(7 downto 4) > "1001") OR (c = '1')
                             then busTemp(8 downto 4) <= ('0' & busB(7 downto 4)) + 6;
                             else busTemp(8 downto 4) <= '0' & busB(7 downto 4);
                         end if;
         when MC_COMP  => busTemp(7 downto 0) <= NOT busA;
         when MC_SWAP  => busTemp(7 downto 0) <= busA(3 DOWNTO 0) & busA(7 DOWNTO 4);
         when MC_BITC  => busTemp(7 downto 0) <= busA AND NOT nroBit;
         when MC_BITS  => busTemp(7 downto 0) <= busA OR nroBit;
         when MC_BITT  => busTemp(7 downto 0) <= busA XOR nroBit;

         when MC_SKIPE => if (busA = busB) then skip <= '1'; end if;
         when MC_SKIPG => if (busA < busB) then skip <= '1'; end if;
         when MC_SKIPL => if (busA > busB) then skip <= '1'; end if;

         when others   => busTemp <=  busTemp;
    END CASE;
end if;

end process;

  busC <= busTemp(7 downto 0);

  c_Out <=  NOT busTemp(8)     when (c_en = '1'      AND cod_Op = MC_RESTA)    else
            busTemp(8)          when (c_en = '1'      AND (cod_Op = MC_SUMA OR
                                      cod_Op = MC_NEG  OR cod_Op = MC_DAW  OR
                                      cod_Op = MC_ROL  OR  cod_Op = MC_ROR))  else

            c;   -- mantiene el carry anterior


  ov_Out <= '1' when ov_en = '1' AND busA(7) = busB(7) AND busA(7) /= busTemp(7) else
            '0' when ov_en = '1' AND not(busA(7)=busB(7) AND busA(7) /= busTemp(7)) else
            ov;  -- mantiene el overflow anterior

  z_Out  <= '1' when z_en = '1' AND busTemp(7 DOWNTO 0) = "00000000" else
            '0' when z_en = '1' AND not (busTemp(7 DOWNTO 0) = "00000000") else
             z;  --mantiene el zero anterior

  dc_Out <= not busTemp(4)    when dc_en = '1' and ( cod_Op = MC_RESTA  ) else
            busTemp(4)        when dc_en = '1' and ( cod_Op /= MC_RESTA ) else
            Dc;  --mantiene el dc anterior

end RTL;


-- ***************************************************************************
-- **   Descripcion en VHDL de una arquitectura RISC                        **
-- **                                                                       **
-- **   Autores: Leyes, Daniel Alejandro                                    **
-- **            Martinez Belot, Luis José Javier                           **
-- **                                                                       **
-- **   Modulo: Registro de Estado de la ALU                                **
-- **                                                                       **
-- **   Modulo que describe el comportamiento del registro ALUSTA           **
-- **   del microcontrolador PIC17                                          **
-- **                                                                       **
-- ***************************************************************************

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.all;

ENTITY ALUSTA IS
  PORT ( clock: IN STD_LOGIC;
         C  : IN STD_LOGIC;
         OV : IN STD_LOGIC;
         DC : IN STD_LOGIC;
         Z  : IN STD_LOGIC;
         F  : IN STD_LOGIC_VECTOR (3 DOWNTO 0);
         C_Out  : OUT STD_LOGIC;
         OV_Out : OUT STD_LOGIC;
         DC_Out : OUT STD_LOGIC;
         Z_Out  : OUT STD_LOGIC;
         F_Out  : OUT STD_LOGIC_VECTOR (3 DOWNTO 0)
    );
END ALUSTA;
```

```
ARCHITECTURE RTL OF ALUSTA IS
Signal ByteEstado : STD_LOGIC_VECTOR (7 DOWNTO 0);

BEGIN
  C_Out  <= ByteEstado(7);
  OV_Out <= ByteEstado(6);
  DC_Out <= ByteEstado(5);
  Z_Out  <= ByteEstado(4);
  F_Out  <= ByteEstado(3 downto 0);


 ByteEstado(7) <= C;
 ByteEstado(6) <= OV;
 ByteEstado(5) <= DC;
 ByteEstado(4) <= Z;
 ByteEstado(3 downto 0) <= F ;

END RTL;
```

```
-- ******************************************************************************
-- **   Descripcion en VHDL de una arquitectura RISC                           **
-- **                                                                          **
-- **   Autores: Leyes, Daniel Alejandro                                       **
-- **            Martinez Belot, Luis José Javier                              **
-- **                                                                          **
-- **   Modulo: Generador de constantes                                        **
-- **                                                                          **
-- **   Modulo que describe el comportamiento de un generador de               **
-- **   constantes (0, 1 y 255)                                                **
-- **                                                                          **
-- ******************************************************************************

Library ieee;
USE IEEE.STD_LOGIC_1164.all;

ENTITY genCons IS

   PORT (gen0, gen1, genFF : IN  std_logic;
         Salida            : OUT std_logic_vector(7 downto 0)
         );
END genCons;


ARCHITECTURE RTL of genCons is

BEGIN
    salida <= "00000000" when gen0   = '1' else
              "00000001" when gen1   = '1' else
              "11111111" when genff  = '1';
END RTL;
```

```
-- ******************************************************************************
-- **   Descripcion en VHDL de una arquitectura RISC                           **
-- **                                                                          **
-- **   Autores: Leyes, Daniel Alejandro                                       **
-- **            Martinez Belot, Luis José Javier                              **
-- **                                                                          **
-- **   Modulo: Selector de la Entrada A de datos a la ALU                     **
-- **                                                                          **
-- **   Modulo que describe el comportamiento del selector del dato            **
-- **   que ingresa a la entrada A a la ALU segun las señales de               **
-- **   activacion indicadas por la unidad de control                          **
-- **                                                                          **
-- ******************************************************************************

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

```
ENTITY MuxBusA IS
   PORT(
        clk       : IN  STD_LOGIC;
        WREG_data : IN STD_LOGIC_VECTOR(7 downto 0);
        RI_data   : IN STD_LOGIC_VECTOR(7 downto 0);
        RAM_data  : IN STD_LOGIC_VECTOR(7 downto 0);
        PC_data   : IN STD_LOGIC_VECTOR(7 downto 0);
        GC_data   : IN STD_LOGIC_VECTOR(7 downto 0);
        WREG_en   : IN  STD_LOGIC;
        RI_en     : IN  STD_LOGIC;
        RAM_en    : IN  STD_LOGIC;
        PC_en     : IN  STD_LOGIC;
        GC_en     : IN  STD_LOGIC;
        Data_OutA : OUT STD_LOGIC_VECTOR(7 downto 0)
       );
END MuxBusA;

ARCHITECTURE RTL OF MuxBusA IS
SIGNAL TEMPA: STD_LOGIC_VECTOR(7 downto 0);
BEGIN

 PROCESS
 BEGIN
      IF WREG_en= '1' THEN TEMPA  <=  WREG_data;
        ELSIF RAM_en = '1' THEN TEMPA  <=  Ram_data;
        ELSIF PC_en  = '1' THEN TEMPA  <=  PC_data ;
        ELSIF GC_en  = '1' THEN TEMPA  <=  GC_data ;
        ELSIF RI_en  = '1' THEN TEMPA  <=  RI_data;
        ELSE TEMPA  <=  TEMPA;
      END IF;

      DATA_OUTA <= TEMPA;
 END PROCESS;

END RTL;



-- ***************************************************************************
-- **   Descripcion en VHDL de una arquitectura RISC                        **
-- **                                                                       **
-- **   Autores: Leyes, Daniel Alejandro                                    **
-- **            Martinez Belot, Luis José Javier                           **
-- **                                                                       **
-- **   Modulo: Selector de la Entrada B de datos a la ALU                  **
-- **                                                                       **
-- **   Modulo que describe el comportamiento del selector del dato         **
-- **   que ingresa a la entrada b a la ALU segun las señales de            **
-- **   activacion indicadas por la unidad de control                       **
-- **                                                                       **
-- ***************************************************************************

library ieee;
use ieee.std_logic_1164.all;

ENTITY MuxBusB IS
   PORT(CLOCK     : IN  STD_LOGIC;
        WREG_data : IN STD_LOGIC_VECTOR(7 downto 0);
        RI_data   : IN STD_LOGIC_VECTOR(7 downto 0);
        GC_data   : IN STD_LOGIC_VECTOR(7 downto 0);
        WREG_en   : IN  STD_LOGIC;
        RI_en     : IN  STD_LOGIC;
        GC_en     : IN  STD_LOGIC;
        Data_OutB : OUT STD_LOGIC_VECTOR(7 downto 0)
       );
END MuxBusB;

ARCHITECTURE RTL OF MuxBusB IS
  SIGNAL TEMPB: STD_LOGIC_VECTOR(7 downto 0);
BEGIN

   PROCESS
   BEGIN
        IF    GC_EN = '1'     THEN TEMPB <= GC_DATA;
          ELSIF WREG_EN = '1'   THEN TEMPB <= WREG_DATA;
          ELSIF RI_EN = '1'     THEN TEMPB <= RI_DATA;
```

```
          ELSE TEMPB <= TEMPB;
        END IF;

       DATA_OUTB<= TEMPB;
   END PROCESS;

END RTL;



-- ************************************************************************
-- **   Descripcion en VHDL de una arquitectura RISC                     **
-- **                                                                    **
-- **   Autores: Leyes, Daniel Alejandro                                 **
-- **            Martinez Belot, Luis José Javier                        **
-- **                                                                    **
-- **   Modulo: Memoria de datos del PIC17                               **
-- **                                                                    **
-- **   Modulo que describe el comportamiento de la RAM del PIC17        **
-- **                                                                    **
-- ************************************************************************

LIBRARY ieee;
LIBRARY work;
USE work.instruccionesAsm.ALL;
USE ieee.std_logic_1164.ALL;

ENTITY RAM IS
  PORT (
         Reset          : IN STD_LOGIC;
         Clock          : IN std_logic;
         Out_En         : IN STD_LOGIC;
         Write_En       : IN STD_LOGIC;
         FSN_In         : IN  std_logic_vector (15 downto 0);
         data_In        : IN  std_logic_vector (7 downto 0);
         data_Out       : OUT std_logic_vector (7 downto 0)
        );
 end RAM;

Architecture RTL of RAM is
--Celdas de memoria RAM, por razones de espacio disponible para la sintesis solo se
--dispone de una celda de memoria RAM
SIGNAL MEM_2: std_logic_vector (7 downto 0);
SIGNAL t: std_logic := '0';

BEGIN
    PROCESS (clock,write_en, data_in)
    BEGIN
      IF clock'event AND clock='1' THEN
         t <='0';
      END IF;

      IF write_en='1' AND  t='0' THEN
         mem_2 <= data_In;
            t <= '1';
      END IF;
    END PROCESS;

    data_out <= mem_2 WHEN out_En = '1';

END RTL;



-- ************************************************************************
-- **   Descripcion en VHDL de una arquitectura RISC                     **
-- **                                                                    **
-- **   Autores: Leyes, Daniel Alejandro                                 **
-- **            Martinez Belot, Luis José Javier                        **
-- **                                                                    **
-- **   Modulo: Registro acumulador                                      **
-- **                                                                    **
-- **   Modulo del registro de intercambio                               **
-- **                                                                    **
-- ************************************************************************
```

```
LIBRARY ieee;
LIBRARY WORK;
USE work.instruccionesAsm.ALL;
USE ieee.std_logic_1164.ALL;

ENTITY regAcum IS

  PORT (
        clock    : IN  STD_LOGIC;
        selectA  : IN  STD_LOGIC;
        selectB  : IN  STD_LOGIC;
        read_en  : IN  STD_LOGIC;
        busC_in  : IN  STD_LOGIC_VECTOR( 7 DOWNTO 0);
        bus_out  : OUT STD_LOGIC_VECTOR( 7 DOWNTO 0)
        );

end regAcum;

Architecture RTL of RegAcum is
   Signal W_Reg : STD_LOGIC_VECTOR( 7 DOWNTO 0 );
   Signal t : STD_LOGIC:= '0';

BEGIN

 PROCESS (clock, read_en, busC_in)
 BEGIN
        IF clock'event AND clock='1' THEN
             t <='0';
        END IF;
        IF  read_en='1' AND t='0' THEN
            W_Reg <= busC_in;
            t <= '1';
        END IF;
 END PROCESS;

 Bus_out <= W_Reg WHEN selectA='1' OR selectB='1';
END RTL;



-- ***************************************************************************
-- **   Descripcion en VHDL de una arquitectura RISC                        **
-- **                                                                       **
-- **   Autores: Leyes, Daniel Alejandro                                    **
-- **            Martinez Belot, Luis José Javier                           **
-- **                                                                       **
-- **   Modulo: FSR                                                         **
-- **                                                                       **
-- **   Modulo para el direccionamiento indirecto                           **
-- **                                                                       **
-- ***************************************************************************

ENTITY regFSR IS

   PORT (clock       : IN  STD_LOGIC;
      reset          : IN  STD_LOGIC;
      indf0_out_en   : IN  STD_LOGIC;
      indf0_write_en : IN  STD_LOGIC;
      indf1_out_en   : IN  STD_LOGIC;
      indf1_write_en : IN  STD_LOGIC;
      data_in        : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
      inst_in        : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
      data_out       : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
      inst_out       : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
        );

Architecture RTL of RegFSR is
   Signal fsr0: STD_LOGIC_VECTOR(7 DOWNTO 0);
   Signal fsr1: STD_LOGIC_VECTOR(7 DOWNTO 0);

   inst_out <= fsr0 when indf0_out_en='1' ELSE
               fsr1 when indf1_out_en='1' ELSE
               "ZZZZZZZZ";

  PROCESS (clock,reset)
    BEGIN
```

```
        IF reset = '0' THEN
           fsr0 <= "00000000";
           fsr1 <= "00000000";
        ELSIF clock'EVENT AND clock = '1' THEN
            IF indf0_write_en = '1' THEN
                fsr0 <= data_in;
            END IF;
            IF indf1_write_en = '1' THEN
                fsr1 <= data_in;
            END IF;
        END IF;
   END PROCESS;
   PROCESS(inst_in, fsr0, fsr1)
    BEGIN
         CASE inst_in IS
           WHEN INDF0 =>
             inst_out <= fsr0;
           WHEN INDF1 =>
             inst_out <= fsr1;
           WHEN OTHERS =>
             inst_out <= inst_in;
         END CASE;
    END PROCESS;
end
End RTL;



-- ************************************************************************
-- **   Descripcion en VHDL de una arquitectura RISC                     **
-- **                                                                    **
-- **   Autores: Leyes, Daniel Alejandro                                 **
-- **            Martinez Belot, Luis José Javier                        **
-- **                                                                    **
-- **   Modulo: Registro de instruccion                                  **
-- **                                                                    **
-- **   Modulo que describe el comportamiento del registro de            **
-- **   instrucciones del PIC17                                          **
-- **                                                                    **
-- ************************************************************************

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.all;

ENTITY regInst IS

   PORT (
        clock      : IN  STD_LOGIC;
        reset      : IN  STD_LOGIC;
        inm_out_en : IN  STD_LOGIC;
        inm_Select : IN  STD_LOGIC;
        skip       : IN  STD_LOGIC;
        inst_in    : IN  STD_LOGIC_VECTOR(15 DOWNTO 0);
        inst_out   : OUT STD_LOGIC_VECTOR(15 DOWNTO 0);
        busA_out   : OUT STD_LOGIC_VECTOR( 7 DOWNTO 0);
        busB_out   : OUT STD_LOGIC_VECTOR( 7 DOWNTO 0)
        );
end RegInst;

Library work;
USE work.instruccionesAsm.ALL;

Architecture RTL of RegInst is
   Signal tempRI: STD_LOGIC_VECTOR (15 DOWNTO 0);
BEGIN
   inst_Out <= tempRI;

   busA_out <= tempRI(7 downto 0) when inm_out_en='1' AND inm_select=SELECT_A;

   busB_out <= tempRI(15 downto 8) when inm_out_en='1' AND inm_select=SELECT_B;


   PROCESS (clock, reset, skip)
   BEGIN
      IF reset = '0' THEN
         tempRI <= "0000000000000000";
      ELSIF clock'EVENT AND clock = '1'
```

```
        THEN If skip='1' THEN
              TempRI <= NOP;
             ELSE tempRI <= inst_in;
             END IF;
   END IF;
   END PROCESS;

End RTL;



-- ************************************************************************
-- **   Descripcion en VHDL de una arquitectura RISC                     **
-- **                                                                    **
-- **   Autores: Leyes, Daniel Alejandro                                 **
-- **            Martinez Belot, Luis José Javier                        **
-- **                                                                    **
-- **   Modulo: Registro del PC                                          **
-- **                                                                    **
-- **   Modulo contador de programa                                      **
-- **                                                                    **
-- ************************************************************************

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.all;
USE ieee.std_logic_arith.all;
use IEEE.std_logic_signed.all;

ENTITY regPC IS

   PORT (
        Clock        : IN  STD_LOGIC;
        Reset        : IN  STD_LOGIC;
        Pcl_out_en   : IN  STD_LOGIC;
        Pch_out_en   : IN  STD_LOGIC;
        Pcl_write_en : IN  STD_LOGIC;
        Pch_write_en : IN  STD_LOGIC;
        Load         : IN  STD_LOGIC;
        Skip         : IN  STD_LOGIC;
        Data_in      : IN  STD_LOGIC_VECTOR( 7 DOWNTO 0);
        Addr_in      : IN  STD_LOGIC_VECTOR(15 DOWNTO 0);
        Data_out     : OUT STD_LOGIC_VECTOR( 7 DOWNTO 0);
        Addr_out     : OUT STD_LOGIC_VECTOR(15 DOWNTO 0)
        );
END RegPC;

Architecture RTL of RegPC is
   Signal PC: STD_LOGIC_VECTOR( 15 DOWNTO 0);
begin
   Data_Out <= pc(7 downto 0)  when pcl_out_en='1' ELSE
               pc(15 downto 8) when pch_out_en='1' ELSE
               pc(7 downto 0);

   Addr_out <= pc;


   PROCESS (clock, reset)
   BEGIN
      IF clock'EVENT AND clock = '1' THEN
         IF pcl_write_en='1' THEN
           Pc(7 downto 0) <= data_in;
         ELSIF load='1' THEN
            Pc(12 downto 0) <= addr_in(12 downto 0);
         ELSE
            IF pch_write_en='1' THEN
               Pc(15 downto 8) <= data_in;
            END IF;
            PC <= PC + '1';
            IF skip='1' THEN
              PC <= PC + '1';
            END IF;
         END IF;
      END IF;
   END PROCESS;
END RTL;
```

```
-- ********************************************************************
-- **   Descripcion en VHDL de una arquitectura RISC                 **
-- **                                                                **
-- **   Autores: Leyes, Daniel Alejandro                             **
-- **            Martinez Belot, Luis José Javier                    **
-- **                                                                **
-- **   Modulo: Memoria de programa del PIC17                        **
-- **                                                                **
-- **   Modulo que contiene el programa a ejecutar                   **
-- **                                                                **
-- ********************************************************************

Library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

ENTITY Rom IS
GENERIC (size: integer := 256);

PORT ( pc_in    : IN  STD_LOGIC_VECTOR(15 DOWNTO 0);
       inst_out : OUT STD_LOGIC_VECTOR(15 DOWNTO 0)
      );
END Rom;


ARCHITECTURE Rtl OF Rom IS
 SIGNAL int: integer;
BEGIN
 int <= conv_integer(pc_in);
 inst_out <=
            "0000000000000000" when int = 0 else
            "1011000001100010" when int = 1 else
            "0000000100010010" when int = 2 else
            "0000111100010010" when int = 3 else
            "0001000100010010" when int = 4 else
            "0001000100010010" when int = 5 else
            "0001000100010010" when int = 6 else
            "0010100100010010" when int = 7 else
            "0000000000000000" when int = 8;
end Rtl;

--
-- Codigo origen del programa
--
--
-- ; file con operaciones a reg
-- ; add
--
-- nop
-- movlw 98
-- movwf 18
-- addwf 18,1
-- addwfc 18,1
-- addwfc 18,1
-- addwfc 18,1
-- clrf 18,1
-- nop



-- ********************************************************************
-- **   Descripcion en VHDL de una arquitectura RISC                 **
-- **                                                                **
-- **   Autores: Leyes, Daniel Alejandro                             **
-- **            Martinez Belot, Luis José Javier                    **
-- **                                                                **
-- **   Modulo: Unidad de Control del PIC17                          **
-- **                                                                **
-- **   Modulo que describe el comportamiento de la unidad de control **
-- **   del PIC17, activando las señales necesarias dependiendo de   **
-- **   la instruccion a ejecutar                                    **
-- **                                                                **
-- ********************************************************************
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

library work;
use work.instruccionesAsm.ALL;

ENTITY UnidadControl IS
   PORT(clock                        : IN STD_LOGIC;
        Z, C, DC, OV, C_old          : OUT STD_LOGIC;
        ToSignal, PdSignal, Glintd   : OUT STD_LOGIC;
        o_wreg_a, o_wreg_b           : OUT STD_LOGIC;
        o_imm_en, imm_select         : OUT STD_LOGIC;
        i_wreg_SALIDA                : OUT STD_LOGIC;
        pc_o, pclath_o               : OUT STD_LOGIC;
        cpusta_o, memdato_o          : OUT STD_LOGIC;
        pc_w, pclath_w               : OUT STD_LOGIC;
        cpusta_w, memdato_w          : OUT STD_LOGIC;
        GC_en0, GC_en1, GC_enF       : OUT STD_LOGIC;
        GC_select                    : OUT STD_LOGIC;
        skip                         : IN STD_LOGIC;
        CodOp                        : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
        CodALU                       : OUT STD_LOGIC_VECTOR(4 DOWNTO 0)
       );
 END UnidadControl;


ARCHITECTURE RTL1 OF UnidadControl IS
 SIGNAL esRegFile_out  : STD_LOGIC;
 SIGNAL o_regfile_en   : STD_LOGIC;
 SIGNAL I_WREG_ANTERIOR: STD_LOGIC := '0';
 SIGNAL I_WREG: STD_LOGIC                 := '0';
 SIGNAL operando: STD_LOGIC_VECTOR(7 DOWNTO 0);
 SIGNAL o_regfile_en_anterior: STD_LOGIC := '0';
 SIGNAL memdato_w_2: STD_LOGIC            := '0';
 SIGNAL pc_w_2: STD_LOGIC                 := '0';
 SIGNAL pclath_w_2: STD_LOGIC             := '0';
 SIGNAL cpusta_w_2: STD_LOGIC             := '0';
 SIGNAL memdato_o_2: STD_LOGIC            := '0';
 SIGNAL pc_o_2: STD_LOGIC                 := '0';
 SIGNAL pclath_o_2: STD_LOGIC             := '0';
 SIGNAL cpusta_o_2: STD_LOGIC             := '0';

BEGIN
  PROCESS (esRegFile_out, CodOp, clock)
  BEGIN
    memdato_o   <= '0';
    Pc_o        <= '0';
    pclath_o    <= '0';
    cpusta_o    <= '0';

    IF(esRegFile_out = '1') THEN
      CASE CodOp(7 DOWNTO 0) IS
        WHEN PCL    => pc_o       <= '1';
        WHEN PCLATH => pclath_o   <= '1';
        WHEN CPUSTA => cpusta_o   <= '1';
        WHEN OTHERS => memdato_o  <= '1';
      END CASE;
    END IF;
  END PROCESS;

  PROCESS (o_regfile_en , CodOp, clock)
  BEGIN
    IF CLOCK'EVENT AND CLOCK='1' THEN
       memdato_w  <= '0';
       Pc_w            <= '0';
       pclath_w        <= '0';
       cpusta_w        <= '0';
       IF(o_regfile_en = '1') THEN
          CASE CodOp(7 DOWNTO 0) IS
            WHEN PCL    => pc_w        <= '1';
            WHEN PCLATH => pclath_w    <= '1';
            WHEN CPUSTA => cpusta_w    <= '1';
            WHEN OTHERS => memdato_w   <= '1';
          END CASE;
       END IF;
    END IF;
  END PROCESS;
```

```
PROCESS (codOp,clock)
BEGIN
  IF clock'event AND clock='1' THEN
    i_wreg_salida <= i_wreg;
    i_wreg        <='0';
  END IF;

  GC_en0   <= '0';
  GC_en1   <= '0';
  GC_enF   <= '0';
  ToSignal <= '1';
  PdSignal <= '1';
  GlinTd   <= '0';
  OV       <= '0';
  DC       <= '0';
  C        <= '0';
  Z        <= '0';
  c_old    <= '0';
  o_wreg_a <= '0';
  o_wreg_b <='0';
  o_regfile_en  <='0';
  esRegFile_out <='0';
  o_imm_en <='0';
  imm_select   <='0';

IF skip = '0' THEN
  --Analisis y asignacion de señales segun el codigo de operacion
  IF CodOp(15 DOWNTO 9) = ADDWF THEN
       OV <= '1';
       DC <= '1';
       C  <= '1';
       Z  <= '1';
       CodALU <= MC_SUMA;
       o_wreg_b <='1';
       IF(CodOp(7 DOWNTO 0) = WREG) THEN
          o_wreg_a      <='1';
          i_wreg        <= '1';
          esRegFile_out <='0';
          o_regfile_en  <='0';
       ELSE
           esRegFile_out <='1';
           IF CodOp(8) = '0' THEN
             o_regfile_en <='0';
             i_wreg <='1';
           ELSE
             o_regfile_en <='1';
           END IF;
       END IF;
  ELSIF CodOp(15 DOWNTO 9) = ADDWFC THEN
       OV <= '1';
       DC <= '1';
       C  <= '1';
       Z  <= '1';
       CodALU <= MC_SUMA;
       c_old <= '1';
       o_wreg_b <='1';
       IF(CodOp(7 DOWNTO 0) = WREG) THEN
           O_wreg_a <='1';
           i_wreg <= '1';
       ELSE
           esRegFile_out <='1';
           IF CodOp(8) = '0' THEN
             i_wreg <='1';
           ELSE
             o_regfile_en <='1';
           END IF;
       END IF;
  ELSIF CodOp(15 DOWNTO 9) = ANDWF THEN
       Z  <= '1';
       CodALU <= MC_AND;
       o_wreg_b <='1';
       IF(CodOp(7 DOWNTO 0) = WREG) THEN
           o_wreg_a <='1';
           i_wreg <= '1';
       ELSE
           esRegFile_out <='1';
```

```
            IF CodOp(8) = '0' THEN
              i_wreg <='1';
            ELSE
              o_regfile_en <='1';
            END IF;
        END IF;
 ELSIF CodOp = CLRWDT THEN
       ToSignal <= '0';
       PdSignal <= '0';
ELSIF CodOp(15 DOWNTO 9) = COMF THEN
       IF(CodOp(7 DOWNTO 0) = WREG)
       THEN
            o_wreg_a <='1';
            i_wreg <= '1';
       ELSE
            esRegFile_out <='1';
            IF CodOp(8) = '0'
            THEN i_wreg <='1';
            ELSE o_regfile_en <='1';
            END IF;
       END IF;
       Z  <= '1';
       CodALU <= MC_COMP;
ELSIF CodOp(15 DOWNTO 9) = INCF THEN
       OV <= '1';
       DC <= '1';
       C  <= '1';
       Z  <= '1';
       CodALU <= MC_SUMA;
       GC_en1          <= '1';
       GC_select       <= SELECT_B;
       IF(CodOp(7 DOWNTO 0) = WREG) THEN
            o_wreg_a <='1';
            i_wreg <= '1';
       ELSE
            esRegFile_out <='1';
            IF CodOp(8) = '0' THEN
              i_wreg <='1';
            ELSE
              o_regfile_en <='1';
            END IF;
       END IF;
 ELSIF CodOp(15 DOWNTO 9) = DECF THEN
       OV <= '1';
       DC <= '1';
       C  <= '1';
       Z  <= '1';
       CodALU <= MC_RESTA;
       GC_en1          <= '1';
       GC_select       <= SELECT_B;
       IF(CodOp(7 DOWNTO 0) = WREG)
       THEN
            o_wreg_a <='1';
            i_wreg <= '1';
       ELSE
            esRegFile_out <='1';
            IF CodOp(8) = '0' THEN
              i_wreg <='1';
            ELSE
              o_regfile_en <='1';
            END IF;
       END IF;
 ELSIF CodOp(15 DOWNTO 9) = IORWF THEN
       Z  <= '1';
       CodALU <= MC_OR;
       o_wreg_b <='1';
       IF(CodOp(7 DOWNTO 0) = WREG) THEN
            o_wreg_a <='1';
            i_wreg <= '1';
       ELSE
            esRegFile_out <='1';
            IF CodOp(8) = '0' THEN
              i_wreg <='1';
            ELSE
              o_regfile_en <='1';
            END IF;
       END IF;
```

```
ELSIF CodOp(15 DOWNTO 8) = MOVWF THEN
     null;  --no causa ningun efecto en las banderas
     CodALU <= MC_MUEVE;
     o_wreg_a <='1';
     IF(CodOp(7 DOWNTO 0) = WREG)
         THEN i_wreg <= '1';
         ELSE o_regfile_en <='1';
     END IF;
ELSIF CodOp = NOP THEN
     codALU <= MC_NOP;
     null;  --no causa ningun efecto en las banderas
ELSIF CodOp = RETFIE THEN
     GlinTd <= '1';
ELSIF CodOp = RETURNN THEN
     null;  --no causa ningun efecto en las banderas
ELSIF CodOp = SLEEP THEN
     ToSignal <= '0';
     PdSignal <= '0';
ELSIF CodOp(15 DOWNTO 9) = SUBWF THEN
     OV <= '1';
     DC <= '1';
     C  <= '1';
     Z  <= '1';
     CodALU <= MC_RESTA;
     o_wreg_b <='1';
     IF(CodOp(7 DOWNTO 0) = WREG) THEN
         o_wreg_a <='1';
         i_wreg <= '1';
     ELSE
         esRegFile_out <='1';
         IF CodOp(8) = '0'
           THEN i_wreg <='1';
           ELSE o_regfile_en <='1';
         END IF;
      END IF;
ELSIF CodOp(15 DOWNTO 9) = SUBWFB THEN
     OV <= '1';
     DC <= '1';
     C  <= '1';
     Z  <= '1';
     CodALU <= MC_RESTA;
     c_old <= '1';
     o_wreg_b <='1';
     IF(CodOp(7 DOWNTO 0) = WREG) THEN
          o_wreg_a <='1';
          i_wreg <= '1';
     ELSE
          esRegFile_out <='1';
          IF CodOp(8) = '0' THEN
             i_wreg <='1';
          ELSE
             o_regfile_en <='1';
          END IF;
     END IF;
ELSIF CodOp(15 DOWNTO 9) = XORWF THEN
     Z  <= '1';
     CodALU <= MC_XOR;
     o_wreg_b <='1';
     IF(CodOp(7 DOWNTO 0) = WREG) THEN
          o_wreg_a <='1';
          i_wreg <= '1';
     ELSE
          esRegFile_out <='1';
          IF CodOp(8) = '0'
          THEN i_wreg <='1';
          ELSE o_regfile_en <='1';
          END IF;
     END IF;
ELSIF CodOp(15 DOWNTO 9) = DECFSZ THEN
     null;  --no causa ningun efecto en las banderas
ELSIF CodOp(15 DOWNTO 9) = RRCF THEN
     C  <= '1';
     c_old <= '1';
     CodALU <= MC_ROR;
     IF(CodOp(7 DOWNTO 0) = WREG) THEN
         o_wreg_a <='1';
         i_wreg <= '1';
```

```
        ELSE
            esRegFile_out <='1';
            IF CodOp(8) = '0'
            THEN i_wreg <='1';
            ELSE o_regfile_en <='1';
            END IF;
        END IF;
    ELSIF CodOp(15 DOWNTO 9) = RLCF THEN
        C  <= '1';
        c_old <= '1';
        CodALU <= MC_ROL;
        IF(CodOp(7 DOWNTO 0) = WREG) THEN
              o_wreg_a <='1';
              i_wreg <= '1';
        ELSE
              esRegFile_out <='1';
              IF CodOp(8) = '0'
              THEN i_wreg <='1';
              ELSE o_regfile_en <='1';
              END IF;
        END IF;
    ELSIF CodOp(15 DOWNTO 9) = SWAPF THEN
        null;  --no causa ningun efecto en las banderas
        CodALU <= MC_SWAP;
        IF(CodOp(7 DOWNTO 0) = WREG)
        THEN
              o_wreg_a <='1';
              i_wreg <= '1';
        ELSE
              esRegFile_out <='1';
              IF CodOp(8) = '0'
              THEN i_wreg <='1';
              ELSE o_regfile_en <='1';
              END IF;
         END IF;
    ELSIF CodOp(15 DOWNTO 9) = INCFSZ THEN
        null;  --no causa ningun efecto en las banderas
    ELSIF CodOp(15 DOWNTO 9) = RRNCF THEN
        null;  --no causa ningun efecto en las banderas
        CodALU <= MC_ROR;
        IF(CodOp(7 DOWNTO 0) = WREG) THEN
              o_wreg_a <= '1';
              i_wreg   <= '1';
        ELSE
             esRegFile_out <='1';
              IF CodOp(8) = '0'
              THEN i_wreg <='1';
              ELSE o_regfile_en <='1';
              END IF;
        END IF;
    ELSIF CodOp(15 DOWNTO 9) = RLNCF THEN
        null;  --no causa ningun efecto en las banderas
        CodALU <= MC_ROL;
        IF(CodOp(7 DOWNTO 0) = WREG) THEN
              o_wreg_a <='1';
              i_wreg <= '1';
        ELSE
              esRegFile_out <='1';
              IF CodOp(8) = '0'
              THEN i_wreg <='1';
              ELSE o_regfile_en <='1';
              END IF;
        END IF;
    ELSIF CodOp(15 DOWNTO 9) = INFSNZ THEN
        null;  --no causa ningun efecto en las banderas
    ELSIF CodOp(15 DOWNTO 9) = DCFSNZ THEN
        null;  --no causa ningun efecto en las banderas
    ELSIF CodOp(15 DOWNTO 9) = CLRF THEN
        null;  --no causa ningun efecto en las banderas
        CodALU <= MC_MUEVE;
        GC_en0 <= '1';
        GC_select <= SELECT_A;
        IF CodOp(7 DOWNTO 0) /= WREG
        THEN o_RegFile_en  <='1';
        END IF;
        IF CodOp(8) = '0' or CodOp(7 DOWNTO 0) = WREG
        THEN i_wreg <='1';
```

```
     END IF;
ELSIF CodOp(15 DOWNTO 9) = SETF THEN
     null;  --no causa ningun efecto en las banderas
     CodALU <= MC_COMP;
     GC_en0 <= '1';
     GC_select <= SELECT_A;
     IF CodOp(7 DOWNTO 0) /= WREG
     THEN o_RegFile_en  <='1';
     END IF;
     IF CodOp(8) = '0' or CodOp(7 DOWNTO 0) = WREG
     THEN i_wreg <='1';
     END IF;
ELSIF CodOp(15 DOWNTO 9) = NEGW THEN
     OV <= '1';
     DC <= '1';
     C  <= '1';
     Z  <= '1';
     CodALU <= MC_NEG;
     o_wreg_b <='1';
     IF CodOp(7 DOWNTO 0) /= WREG
     THEN o_RegFile_en  <='1';
     END IF;
     IF CodOp(8) = '0' or CodOp(7 DOWNTO 0) = WREG
     THEN i_wreg <='1';
     END IF;
ELSIF CodOp(15 DOWNTO 9) = DAW THEN
     C  <= '1';
     CodALU <= MC_DAW;
     o_wreg_b <='1';
     IF CodOp(7 DOWNTO 0) /= WREG
     THEN o_RegFile_en  <='1';
     END IF;
     IF CodOp(8) = '0' or CodOp(7 DOWNTO 0) = WREG
     THEN i_wreg <='1';
     END IF;
ELSIF CodOp(15 DOWNTO 8) = CPFSEQ THEN
     null;  --no causa ningun efecto en las banderas
     CodALU <= MC_SKIPE;
     IF(CodOp(7 DOWNTO 0) = WREG) THEN
         o_wreg_b <='1';
     ELSE
         esRegFile_out <='1';
     END IF;
     o_wreg_a <= '1';
ELSIF CodOp(15 DOWNTO 8) = CPFSGT THEN
     null;  --no causa ningun efecto en las banderas
ELSIF CodOp(15 DOWNTO 8) = CPFSLT THEN
     null;  --no causa ningun efecto en las banderas
ELSIF CodOp(15 DOWNTO 8) = TSTFSZ       THEN
     null;  --no causa ningun efecto en las banderas
ELSIF CodOp(15 DOWNTO 8) = MULWF THEN
     null;  --no causa ningun efecto en las banderas
ELSIF CodOp(15 DOWNTO 11) = BTG  THEN
     null;  --no causa ningun efecto en las banderas
     CodALU <= MC_BITT;
     o_imm_en <='1';
     imm_select <=SELECT_B;
     IF(CodOp(7 DOWNTO 0) = WREG) THEN
          o_wreg_a <='1';
          i_wreg <= '1';
     ELSE
          esRegFile_out <='1';
          o_regfile_en <='1';
     END IF;
ELSIF CodOp(15 DOWNTO 13) = MOVFP THEN
     null;  --no causa ningun efecto en las banderas
ELSIF CodOp(15 DOWNTO 13) = MOVPF THEN
     Z  <= '1';
ELSIF CodOp(15 DOWNTO 11) = BSF  THEN
     null;  --no causa ningun efecto en las banderas
     CodALU <= MC_BITS;
     o_imm_en <='1';
     imm_select <=SELECT_B;
     IF(CodOp(7 DOWNTO 0) = WREG) THEN
         o_wreg_a <='1';
         i_wreg <= '1';
     ELSE
```

```
            esRegFile_out <='1';
            o_regfile_en <='1';
        END IF;
    ELSIF CodOp(15 DOWNTO 11) = BCF  THEN
        null;  --no causa ningun efecto en las banderas
        CodALU <= MC_BITC;
        o_imm_en <='1';
        imm_select <=SELECT_B;
        IF(CodOp(7 DOWNTO 0) = WREG) THEN
            o_wreg_a <='1';
            i_wreg <= '1';
        ELSE
            esRegFile_out <='1';
            o_regfile_en <='1';
        END IF;
    ELSIF CodOp(15 DOWNTO 11) = BTFSS       THEN
        null;  --no causa ningun efecto en las banderas
    ELSIF CodOp(15 DOWNTO 11) = BTFSC       THEN
        null;  --no causa ningun efecto en las banderas
    ELSIF CodOp(15 DOWNTO 10) = TLRD THEN
        null;  --no causa ningun efecto en las banderas
    ELSIF CodOp(15 DOWNTO 10) = TLWT THEN
        null;  --no causa ningun efecto en las banderas
    ELSIF CodOp(15 DOWNTO 10) = TABLRD      THEN
        null;  --no causa ningun efecto en las banderas
    ELSIF CodOp(15 DOWNTO 10) = TABLWT      THEN
        null;  --no causa ningun efecto en las banderas
    ELSIF CodOp(15 DOWNTO 8) = MOVLW THEN
        null;  --no causa ningun efecto en las banderas
        CodALU <= MC_MUEVE;
        o_imm_en <='1';
        imm_select <=SELECT_A;
        i_wreg <='1';
    ELSIF CodOp(15 DOWNTO 8) = ADDLW THEN
        OV <= '1';
        DC <= '1';
        C  <= '1';
        Z  <= '1';
        CodALU <= MC_SUMA;
        o_imm_en <='1';
        imm_select <=SELECT_A;
        o_wreg_b <= '1';
        i_wreg   <= '1';
    ELSIF CodOp(15 DOWNTO 8) = SUBLW THEN
        OV <= '1';
        DC <= '1';
        C  <= '1';
        Z  <= '1';
        CodALU <= MC_RESTA;
        o_imm_en <='1';
        imm_select <=SELECT_A;
        o_wreg_b <='1';
        i_wreg <='1';
    ELSIF CodOp(15 DOWNTO 8) = IORLW THEN
        Z  <= '1';
        CodALU <= MC_OR;
        o_imm_en <='1';
        imm_select <=SELECT_A;
        o_wreg_b <='1';
        i_wreg <='1';
    ELSIF CodOp(15 DOWNTO 8) = XORLW THEN
        Z  <= '1';
        CodALU <= MC_XOR;
        o_imm_en <='1';
        imm_select <=SELECT_A;
        o_wreg_b <='1';
        i_wreg <='1';
    ELSIF CodOp(15 DOWNTO 8) = ANDLW THEN
        Z  <= '1';
        CodALU <= MC_AND;
        o_imm_en <='1';
        imm_select <=SELECT_A;
        o_wreg_b <='1';
        i_wreg <='1';
    ELSIF CodOp(15 DOWNTO 8) = RETLW THEN
        null;  --no causa ningun efecto en las banderas
        CodALU <= MC_MUEVE;
```

```
          o_imm_en <='1';
          imm_select <=SELECT_A;
          i_wreg <='1';
      ELSIF CodOp(15 DOWNTO 8) = LCALL THEN
          null;  --no causa ningun efecto en las banderas
      ELSIF CodOp(15 DOWNTO 8) = MOVLB THEN
          null;  --no causa ningun efecto en las banderas
      ELSIF CodOp(15 DOWNTO 9) = MOVLR THEN
          null;  --no causa ningun efecto en las banderas
      ELSIF CodOp(15 DOWNTO 8) = MULLW THEN
          null;  --no causa ningun efecto en las banderas
      ELSIF CodOp(15 DOWNTO 13) = GOTO THEN
          null;  --no causa ningun efecto en las banderas
      ELSIF CodOp(15 DOWNTO 13) = CALL   THEN
          null;  --no causa ningun efecto en las banderas
      END IF;
    ELSE
      CodALU <=MC_NOP;
    END IF;


  END PROCESS;

END RTL1;
```