

**Universidad de Costa Rica
Facultad de Ingeniería
Escuela de Ingeniería Eléctrica**

IE – 0502 Proyecto Eléctrico

**Generación de un modelo sintetizable en compuertas
estándar para la CPUCR**

Por:

**Warren Alvarado Pacheco A00190
Esteban Alonso Ortiz Cubero A12626**

**Ciudad Universitaria Rodrigo Facio
Julio del 2005**

Generación de un modelo sintetizable en compuertas estándar para la CPUCR

Por:

Warren Alvarado Pacheco A00190
Esteban Alonso Ortiz Cubero A12626

Sometido a la Escuela de Ingeniería Eléctrica
de la Facultad de Ingeniería
de la Universidad de Costa Rica
como requisito parcial para optar por el grado de:

BACHILLER EN INGENIERÍA ELÉCTRICA

Aprobado por el Tribunal:

Ing. Roberto Rodríguez Rodríguez
Profesor Guía

Ing. Federico Ruiz Ugalde
Profesor lector

Ing. Enrique Coen Alfaro
Profesor lector

DEDICATORIA

A mis padres, a mi hermano Jose,
a mi hermana Laura y a mi abuela,
por su incondicional apoyo durante
toda mi carrera. A mi novia Cristina,
por estar a mi lado durante la
realización de este proyecto.

Esteban Ortiz Cubero

A mis padres y a mi hermana,
por su incondicional apoyo durante
toda mi carrera y el desarrollo
del proyecto

Warren Alvarado

RECONOCIMIENTOS

A profesor Ing. Roberto Rodríguez, por la ayuda y el apoyo brindado durante toda la realización del proyecto, por sus numerosos consejos y por guiarnos en la dirección correcta.

A los estudiantes que trabajaron anteriormente con la descripción de la CPUCR, por brindarnos una base de dónde partir para realizar nuestro trabajo.

A nuestras familias, por la paciencia y comprensión que nos han demostrado durante toda nuestra carrera universitaria, y en especial, durante la realización del proyecto.

ÍNDICE GENERAL

ÍNDICE DE FIGURAS	vii
ÍNDICE DE TABLAS	viii
RESUMEN.....	ix
CAPÍTULO 1: Introducción.....	1
1.1 Objetivos	2
1.1.1 Objetivos específicos.....	2
1.2 Metodología.....	3
 CAPÍTULO 2: Desarrollo teórico	 4
2.1 La CPUCR.....	4
2.2 Memoria principal.....	6
2.3 Registros internos de la CPUCR.....	7
2.3.1 El contador de programa.....	7
2.3.2 El acumulador	7
2.3.3 El registro de estado	9
2.3.4 El puntero de pila	11
2.4 Señales de control de la CPUCR.	12
2.4.1 Bus de direcciones (A0 ... A15).....	13
2.4.2 Bus de datos (A0 ... A7)	13
2.4.3 Línea de reloj (CLK)	13
2.4.4 Línea de reposición (RPS)	13
2.4.5 Línea de lectura/escritura (L/E)	14
2.4.6 Línea de referencia a memoria (M).....	14
2.4.7 Línea de ciclo de búsqueda (CB)	14
2.4.8 Línea de ciclo de memoria (CM)	15
2.4.9 Línea de indicación de detenido (HLT).....	15
2.4.10 Línea de solicitud de interrupción ($\overline{\text{INT}}$).....	16
2.4.11 Línea de aceptación de interrupción (INTOK)	16
2.4.12 Línea de solicitud de acceso directo a memoria ($\overline{\text{SDMA}}$).....	17
2.4.13 Línea de indicación de buses disponibles (BD).....	18
 CAPÍTULO 3: Programación orientada a síntesis con las herramientas de Synopsys.....	 19

3.1 Lineamientos generales para la síntesis de circuitos usando Synopsys.....	19
3.2 Síntesis de tareas del sistema y funciones	22
3.3 Directivas de compilador.	22
3.4 Instanciación de Módulos y Jerarquía de Módulos.	22
3.5 Tipos de Datos.....	23
3.6 Memorias.	24
3.8 Controles de Tiempo.....	25
3.9 Asignaciones bloqueantes y no bloqueantes.	25
3.10 Síntesis de bloques CASE.....	26
3.11 Buses de tercer estado y asignaciones continuas.....	27

CAPÍTULO 4: Síntesis de la CPUCR con las herramientas de Synopsys y la librería AMI 0.5um. 28

4.1 Estado inicial del proyecto.....	28
4.2 Reorganización del código.....	29
4.2 Síntesis del Registro de Instrucción (RI)	33
4.3 Síntesis del Registro de Estado Presente	35
4.4 Síntesis del módulo de Ciclo de Búsqueda.....	36
4.5 Síntesis del módulo de Próximo Estado	38
4.6 Síntesis del acumulador.	40
4.7 Síntesis del contador de programa.....	41
4.8 Síntesis de los buses de tercer estado	43
4.9 Pruebas realizadas al modelo sintetizado.....	44
4.10 Instrucciones para la creación de nuevas instrucciones de la CPUCR.....	44

BIBLIOGRAFÍA..... 49

APÉNDICES 50

Apéndice 1 Manual introductorio para el uso de VCS.....	50
---	----

ANEXOS..... 66

A.1 Script para para la simulación del modelo sintetizado	66
A.2 Script para la simulación del modelo en Verilog®	66
A.3 Script para la síntesis desde una consola	66
A.4 Script para la síntesis desde una consola gráfica.....	67
A.5 Circuitos esquemáticos sintetizados para la CPUCR.	68
A.6 Código del modelo conductual de la CPUCR en Verilog®.....	79

ÍNDICE DE FIGURAS

Figura 2.1 Detalle de los buses de control de la CPUCR	5
Figura 2.2 Registro status de la CPUCR	9
Figura 3.1 Flujo de diseño para un circuito integrado.....	21
Figura 4.1 Módulos en que se encuentra dividido la CPUCR.	30
Figura 4.2 Diagrama de interacción entre los módulos de la CPUCR	31
Figura 4.4 Circuito sintetizado del registro de instrucción.....	35
Figura 4.5 Circuito sintetizado del registro de estado presente	36
Figura 4.6 Circuito sintetizado para las líneas CB, CM y BD	38
Figura 4.6 Circuito sintetizado del registro de próximo estado.	39
Figura A.5.1 Circuito sintetizado del acumulador	68
Figura A.5.2 Circuito Sintetizado del módulo ciclo de búsqueda.....	69
Figura A.5.3 Circuito Sintetizado del módulo de estado presente	70
Figura A.5.4 Circuito Sintetizado del módulo de HLT	71
Figura A.5.5 Circuito Sintetizado del módulo de lectura/escritura.....	72
Figura A.5.6 Circuito Sintetizado del módulo de la línea M	73
Figura A.5.7 Circuito Sintetizado del puntero de pila.....	74
Figura A.5.8 Circuito Sintetizado del contador de programa	75
Figura A.5.9 Circuito Sintetizado del módulo de próximo estado.....	76
Figura A.5.10 Circuito Sintetizado del módulo de registro de instrucción	77
Figura A.5.11 Circuito Sintetizado de la CPUCR completa.....	78

ÍNDICE DE TABLAS

Tabla 1.1 Tabla de instrucciones de la CPU CR de 8 bits.....	6
--	---

RESUMEN

El presente proyecto trata el proceso de creación de un modelo sintetizable de la CPUCR, un microprocesador didáctico desarrollado enteramente en la Universidad de Costa Rica. Se utiliza el lenguaje de descripción de hardware Verilog® para la realización de dicho modelo.

Se explica el subconjunto de comandos de Verilog® que son sintetizables usando las herramientas de Synopsys®, como el simulador lógico VCS®. Por último. Se explica la forma en que se utilizan dichas herramientas durante todo el proceso de diseño.

Por último se describen los principales obstáculos que se presentaron durante todo el desarrollo del proyecto, la forma en que se superaron y la forma de evitarlos en futuras implementaciones.

CAPÍTULO 1: Introducción

La carrera de la humanidad en busca de nuevos y mayores avances tecnológicos no se ha detenido, y con el advenimiento de la era digital, esto no ha cambiado, las grandes universidades del mundo se han convertido en los grandes centros de investigación, aportando cientos de nuevos descubrimientos cada año. Por esto, las universidades que quieran afrontar los retos del nuevo siglo, deben de modernizarse y plantearse nuevas metas.

La Universidad de Costa Rica siempre se ha caracterizado por ser la universidad líder en investigación en nuestro país, por lo tanto, no se escapa a esta etapa de modernización. Con este fin, el Departamento de Automática y Sistemas Digitales de la Escuela de Ingeniería Eléctrica, se ha planteado el reto de crear el primer microprocesador completamente diseñado en Costa Rica: la CPUCR.

Este microprocesador didáctico, diseñado por profesores de la Escuela, y depurado y mejorado por los mismos estudiantes, fue elegido por su simplicidad teórica y por haber sido estudiado a lo largo de los años por varias generaciones de ingenieros eléctricos.

Una de las etapas finales del proceso de diseño del microprocesador, es el proceso de síntesis, mediante el cual, se toma un modelo del procesador descrito en algún lenguaje de descripción de *hardware*¹ (en este caso Verilog®) y se lleva a una descripción física del circuito, siguiendo ciertas reglas de diseño determinadas por un proceso de fabricación que fue escogido con anterioridad.

¹ *Hardware*: Dispositivo.

El proyecto se basa enteramente en lograr un proceso de síntesis limpio, utilizando herramientas de diseño asistido por computadora, con el fin de poder fabricar el primer circuito integrado de la CPUCR.

Al lograr completar la síntesis del circuito de la CPUCR, el camino hacia la creación del primer microprocesador completamente diseñado en Costa Rica, por fin estará libre de sus mayores obstáculos.

1.1 Objetivos

Obtener un modelo sintetizable de la CPUCR en celdas estándar Ami 500nm.

1.1.1 Objetivos específicos

1. Generar una síntesis correcta del modelo de la CPUCR en las celdas estándar Ami 0.5.
2. Optimizar el modelo en Verilog® de la CPUCR para su síntesis a través de las herramientas de Synopsys.
3. Crear un manual detallado del compilador de Verilog® VCS.
4. Documentar el subconjunto de comandos de Verilog® sintetizable en Synopsys.

1.2 Metodología

El proyecto fue dividido en dos partes esenciales que se complementan, como se aprecia en los objetivos planteados para el mismo. La primera parte consistió en la creación de un manual de uso del compilador de Verilog® VCS². Para esto se utilizó más que todo la experiencia de trabajo con la herramienta, así como la lectura continua de los manuales dados por el fabricante, entonces, por la repetición continua y la experimentación con las herramientas, se determinó cuál era el mejor método para completar una síntesis de un circuito utilizando el compilador VCS.

Para la segunda parte del proyecto, que consistió en la síntesis de la CPUCR, se utilizaron principalmente materiales bibliográficos creados por estudiantes que con anterioridad trataron de sintetizar el circuito de la CPUCR, asimismo se utilizaron los resultados de la primera parte, con lo cual se logró una utilización óptima de los recursos brindados por las herramientas de Synopsys.

Cabe recordar que en diferentes tramos del desarrollo del proyecto fue necesaria la consulta con diferentes expertos en el área de diseño digital, como los profesores Enrique Coen y Roberto Rodríguez con el fin de obtener información reciente y detallada de cómo lograr una síntesis exitosa.

² VCS: Verilog Compiled Source, código de Verilog compilado.

CAPÍTULO 2: Desarrollo teórico

Para poder referirse a la implementación en Verilog® de un circuito integrado, resulta indispensable describir la arquitectura del mismo. La CPUCR original fue descrita por los ingenieros Dr. Randolph Steinvorth y el Ing. Marco Vásquez. En su documento de 1997, mencionaron con lujo de detalles todas las partes de las que se componía el primer circuito integrado completamente diseñado en Costa Rica, que era, esencialmente de 6 bits.

Cuando se define el número de bits que tiene cada uno de los registros internos del procesador se obtiene también el ancho de los buses, el número máximo de direcciones de memoria a las que puede acceder, el ancho de la palabra de código, etc. Por ende, cuando se realiza una nueva descripción en 8 bits (como de la que se ocupó este proyecto), es necesario variar algunas de las definiciones de la arquitectura original.

Se procedió entonces a iniciar la descripción de la CPUCR de 8 bits, tomando como base el documento original de Steinvorth y Vásquez, alterando sólo aquellas partes en las que, por los motivos antes mencionados, las arquitecturas difieren.

2.1 La CPUCR

La CPUCR es una unidad central de procesamiento que dispone de un bus de direcciones de 16 líneas, un bus de datos bidireccional de 8 bits y un bus de control con 11 líneas. Inicialmente, el bus de control poseía 12 líneas, pero debido a los requerimientos de espacio en silicio, se eliminó la señal de WAIT de la descripción final.

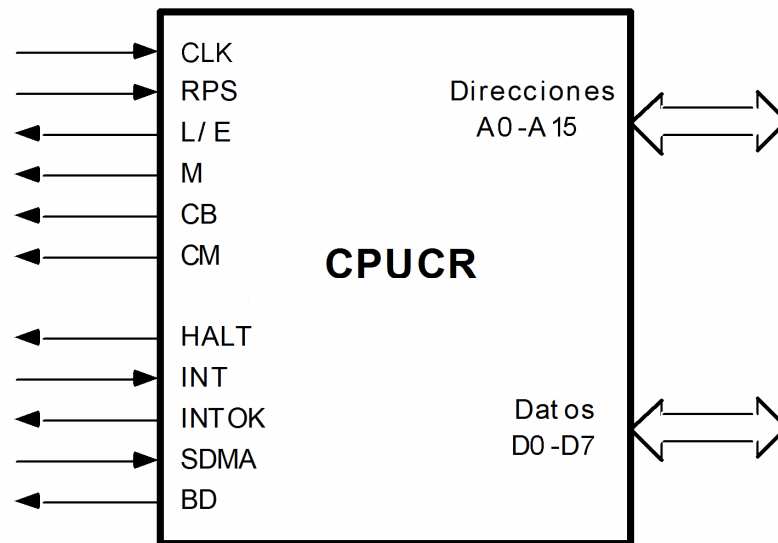


Figura 2.1 Detalle de los buses de control de la CPUCR

El detalle de sus líneas externas se presenta en la Figura 2.1. La línea CLK es la que recibe el reloj externo para sincronizar todas las acciones de la CPUCR. RPS sirve para reiniciar el funcionamiento de la máquina en cualquier momento de ejecución y llevarla a un estado conocido. Las líneas L/E, M, CB y CM controlan las transacciones en el bus de datos. HALT sirve para indicar cuando la CPUCR ha ejecutado una instrucción HLT. INT e INTOK, sirven para manejar el mecanismo de interrupciones, con el cual, ciertos periféricos se comunican con la CPU. Y finalmente, SDMA y BD se usan para que un dispositivo externo pueda controlar directamente las transferencias de datos entre memoria y periféricos sin intervención de la CPUCR.

Bits D7..D4		Bits D3..D0															
		Inmediato	Absoluto	Relativo	Indirecto	Implícito	Acumulado r	Control	E/S								
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	.	LDA	BEQ	LDA	.	CLA	TPA	INP
1	.	STA	BNE	STA	.	CPA	TAP	OUT
2	ADD	ADD	BCS	ADD	SEC	INA	RTI
3	SUB	SUB	BCC	SUB	CLC	DCA	RTS
4	AND	AND	BMI	AND	SEI	ROL	HLT
5	ORA	ORA	BPL	ORA	CLI	ROR	NOP
6	.	JMP	BVS	JMP	.	PLA	PLS
7	.	JSR	BVC	JSR	.	PHA	PHS
8
9
A
B
C
D
E
F

Tabla 1.1 Tabla de instrucciones de la CPUCR de 8 bits

2.2 Memoria principal

La memoria principal está compuesta de un gran número de palabras o localizaciones. Cada localización de memoria tiene una dirección. Si la memoria tiene 4096 palabras, sus direcciones serán 0, 1, 2, ..., 4095. Una palabra o localización puede usarse para almacenar un número binario (dato) o una instrucción del programa, codificada en forma binaria. Las instrucciones de un programa se almacenan en la memoria en localizaciones sucesivas. El computador lee una instrucción de la memoria y la ejecuta. Entonces, lee la próxima instrucción y realiza otra operación y así sucesivamente. En algunos computadores la memoria es dividida en bloques iguales llamados páginas. El tamaño de la página varía de un sistema a otro y está la mayor parte de las veces condicionado a las necesidades de la CPU, para la CPUCR se tienen

páginas de 256 posiciones de memoria. Para un bus de datos de 16 bits, es posible direccional hasta 65536 posiciones de memoria, cada una con 8 bits.

2.3 Registros internos de la CPU.

2.3.1 El contador de programa

Este registro sigue la ejecución del programa. Al final de la ejecución de cada instrucción, el contador de programa PC (del inglés "Program Counter"), contiene la dirección de la localización de memoria que almacena la próxima instrucción del programa. El contador de programa se va incrementando durante la ejecución de la instrucción. Finalmente, el PC contiene la dirección que apunta a la siguiente instrucción a ejecutarse. En algunos casos, el contenido de PC puede ser cambiado por el mismo programa. En esta forma se transfiere la ejecución a otra sección del programa.

Para tener acceso a un espacio total de memoria de 64k palabras es necesario un total de 16 bits en el registro PC. De esta forma se pueden generar direcciones del 0 al 65535 inclusive.

Internamente, existe otro registro muy similar al PC, este se denomina RDR, y es el que determina en realidad la próxima dirección a la que apunta el bus de datos.

2.3.2 El acumulador

El acumulador A, es el componente principal para la manipulación de datos. Este registro tiene la función de almacenar el resultado de operaciones lógicas y aritméticas, calculadas en la ALU, que involucran uno o dos operandos a la vez. En

muchas de las arquitecturas de computador, uno de los operandos proviene del acumulador y el segundo de la localización de memoria especificada por la instrucción. El resultado de la operación se mantiene usualmente en A. El computador puede probar el resultado en A para determinar si contiene alguna condición especial, por ejemplo si es cero, negativo, positivo etc. y tomar diferentes decisiones basado en el resultado de la prueba.

El tamaño del acumulador no se determina tan fácilmente como en el caso del PC, sino que depende de una serie de factores relacionados entre si. Como regla general, cuanto mayor sea su tamaño, se puede procesar en forma simultanea más información de cualquier tipo. Por lo tanto, se evidencia que el tamaño de A condiciona características del computador tales como su velocidad de ejecución, la precisión de sus operandos y en forma indirecta el formato de sus instrucciones. Sin embargo un acumulador grande requiere de mucha circuitería lo cual afecta considerablemente el costo total del computador, lo que lleva al eterno dilema de los ingenieros de ajustar convenientemente rendimiento y costo.

El tamaño en bits del acumulador, determina el ancho de la memoria principal y esta es una de las características más importantes de una CPU. Los tamaños de palabra más frecuentes que se encuentran son 4, 8, 16, 32, 64 y 128 bits. La CPUCR ha sido rediseñada para tener 8 bits en el acumulador.

Existen otras configuraciones de CPU que contienen un grupo de registros de uso general que duplican las funciones del acumulador. En estos computadores, se pueden realizar operaciones aritméticas no solamente entre los registros y alguna localización de memoria, sino entre los mismos registros. Esto permite gran flexibilidad

a la hora de hacer un programa y en ciertas ocasiones hasta se pueden mantener todas las variables del programa dentro de los registros de la CPU aumentando así la velocidad de ejecución del mismo.

2.3.3 El registro de estado

Este registro tiene la función de almacenar información importante para la programación, generalmente referente al contenido del acumulador o al estado de la CPU.



Figura 2.2 Registro status de la CPUCR

En la CPUCR, el registro de estado S (del inglés *status*), es un registro formado por ocho *flip flops*, cuatro de los cuales retienen información acerca del resultado de la última operación con el acumulador, y el restante se utiliza para memorizar una acción de control pertinente a la CPU. Los primeros cuatro son, según la Figura 2.2, los *flip flops* de: signo, N; rebase, V; cero, Z y acarreo, C. El último se llama *flip-flop* de inhibición de interrupción I.

Comúnmente se les refiere también como BANDERAS DE CONDICIÓN y se utilizan para tomar decisiones durante la ejecución del programa.

La bandera de signo (N)

El contenido de esta bandera es copia fiel del bit más significativo del acumulador. Este bit representa el signo del número en el acumulador si se interpreta como un número en complemento a dos.

La bandera de rebase (V)

Esta bandera sirve para detectar condiciones de rebase en operaciones aritméticas con signo. En forma general, el bit más significativo de una palabra, bit 7, representa el signo de la magnitud contenida en los bits restantes, bits 6 a 0. Un 1 en el bit más significativo, equivale a un signo negativo y un cero a un signo positivo. Los números con signo se representan en notación complemento a dos.

La condición de rebase se presenta cuando el resultado de una operación tiene un valor mayor que el que se puede contener en ocho bits, de -127 a +128. La bandera V se pone en 1 cuando se genera un llevo de la posición A6 a la posición A7, o de la posición A7 a la bandera de llevo C exclusivamente.

La bandera de cero (Z)

Este *flip-flop* sirve para detectar la condición de resultado cero en el acumulador. Cuando todos los bits del acumulador se encuentran en cero, esta bandera se pone en 1.

La bandera de acarreo (C)

El *flip-flop* C, se utiliza como una extensión del acumulador para almacenar el acarreo o debo que genera la posición A7 en las operaciones que superan su capacidad de números positivos de 8 bits.

También, se utiliza para conectar en lazo, el dígito más significativo y el menos significativo de A. Este lazo se puede utilizar para desplazar el contenido de A hacia la izquierda o a la derecha, bajo control de programa.

La bandera de interrupción (I)

Sirve para permitir o prohibir la interrupción externa del programa que se está ejecutando.

En situaciones especiales, es necesario que un dispositivo externo interrumpa la ejecución del presente programa para que se le brinde atención inmediata. Como un ejemplo se podría citar una unidad de discos que está buscando los datos pedidos por la CPU; en el momento que se encuentran, la CPU debe tomarlos o de lo contrario se perderán, por causa de la rotación que lleva el disco y no volverán a estar disponibles sino hasta la siguiente vuelta. La interrupción permite a la CPU dejar el curso normal de sus labores para atender una situación especial.

También se puede dar el caso contrario: el computador está realizando una labor especial y no desea ser interrumpido. La bandera de interrupción I sirve para habilitar o inhibir las interrupciones a la CPU.

2.3.4 El puntero de pila

La pila es un conjunto bien definido de posiciones de memoria, en la cual los datos entran (se apilan) y salen (se desapilan), siguiendo la regla EL ULTIMO QUE ENTRA ES EL PRIMERO QUE SALE. En inglés se le conoce como “stack” y la regla es “Last In First Out” o LIFO. Su función primordial es la de servir como medio para almacenar la dirección de retorno después de la ejecución de una subrutina.

Las posiciones de memoria de la pila son direccionadas a partir de un registro especial generador de direcciones llamado el apuntador de pila o puntero de la pila (P). Este registro es controlado automáticamente por la CPU para que el acceso a la pila se realice según la regla antes mencionada.

Durante los accesos a la pila, la CPU se encarga de incrementar el puntero de la pila después de cada escritura y lo decrementa antes de cada lectura. Este proceso, causa el efecto de “último que entra, primero que sale”. Sin embargo, el mismo efecto se

obtiene si la CPU decrementa el apuntador después de cada escritura y lo incrementa antes de cada lectura. La diferencia radica en el hecho de que la pila “crece” hacia posiciones de memoria más altas (positivamente) en un caso, y en sentido opuesto (negativamente) en el otro. Después de un acceso a la pila, el puntero contiene una dirección que “apunta” hacia la próxima localización disponible para almacenar un dato.

La pila de la CPUCR tiene un crecimiento positivo, y después de accederla, el apuntador contiene la dirección de la próxima localización disponible para apilar un dato. Además, el puntero de pila, P, es de 8 bits y limita la pila a solo 256 posiciones, ubicadas automáticamente por la CPUCR en la última página de la memoria (página \$FF). La ubicación de la base de la pila, es decir, la primera localización disponible en la pila, no es necesariamente la primera posición de memoria de la página \$FF, ésta se puede escoger usando la instrucción TAP, cuando se ha cargado previamente A con el valor deseado. En otras CPUs la pila puede ubicarse en cualquier parte de la memoria principal y el programador puede asignarle el tamaño que considere necesario para la aplicación que está desarrollando.

2.4 Señales de control de la CPUCR.

La CPUCR cuenta con un bus de direcciones de 16 líneas, A0 a A15, un bus de datos de 8 líneas, D0 a D7 y un bus de control formado por las señales CLK, RPS , L/E, M, CB, CM, HALT, INT, INTOK, SDMA y BD; las cuales aparecen en la Figura 2.1. Además de las líneas antes mencionadas, la CPUCR recibe su alimentación de potencia a través de otras dos llamadas VCC y REF.

2.4.1 Bus de direcciones (A0 ... A15)

Las 16 líneas del bus de direcciones se usan para seleccionar cualquiera de las 65536 posiciones de memoria de la CPUCR. Normalmente estas líneas las maneja la CPUCR; pero durante el acceso directo a la memoria, la CPUCR cede su control al dispositivo que va a realizar la transferencia.

2.4.2 Bus de datos (A0 ... A7)

El bus de datos consta de 8 líneas y sirve para trasladar instrucciones y datos entre la CPUCR, la memoria y los puertos de E/S. El sentido que llevan los datos en el bus lo determina la línea de lectura/escritura L/E, cuando la línea se encuentra en alto, se estarán leyendo los datos de la memoria, si por el contrario la línea se encuentra en bajo, se escribirán los datos en la memoria. El bus de datos también es cedido a un dispositivo externo durante la operación de acceso directo a la memoria.

2.4.3 Línea de reloj (CLK)

Esta línea se conecta a una base de tiempos externa que utiliza la CPUCR para sincronizar y temporizar las acciones que se llevan a cabo durante la ejecución de las instrucciones. El reloj es una simple onda cuadrada de una frecuencia determinada.

Se define un CICLO DE RELOJ como el equivalente a un período básico de la señal de reloj. Todas las acciones en la CPUCR se completan en el borde decreciente del ciclo de reloj.

2.4.4 Línea de reposición (RPS)

Esta línea de entrada se activa por nivel bajo. Su función es la de llevar la CPUCR a un estado inicial conocido a partir del cual comienza su operación.

Cuando la entrada RPS es llevada a cero la CPUCR cancela inmediatamente todas sus actividades. Seguidamente reinicia el contador de programa a cero y la bandera de inhibición de interrupción, I, a uno. Apenas la línea RPS vuelve a uno, la CPUCR comienza a ejecutar su primera instrucción en la posición de memoria \$0000. Puesto que $I=1$, la CPUCR no aceptará interrupciones en ese momento, permitiendo correr un programa que coloque al sistema computador en un estado inicial conocido.

Generalmente se busca que la línea RPS se active automáticamente al encenderse el computador, o manualmente, por medio de un interruptor de contacto momentáneo que permita reponerlo después del encendido, en cualquier momento.

2.4.5 Línea de lectura/escritura (L/E)

La línea L/E es el medio de que dispone la CPUCR para indicarle a la memoria principal y a los puertos de E/S, el sentido en que se realizan las transferencias de datos. Un valor de 1 lógico indica lectura, lo cual significa que el dato en cuestión se traslada de la memoria o un puerto de entrada hacia la CPUCR. Cuando L/E está en cero ocurre una escritura, es decir, el dato sale de la CPUCR.

2.4.6 Línea de referencia a memoria (M)

La CPUCR utiliza la línea M con el fin de indicar si el valor en el bus de direcciones corresponde a una dirección de memoria ($M=1$) o de puertos de E/S ($M=0$). Ya que la mayoría de las transacciones se llevan a cabo con la memoria, la línea M se mantiene generalmente en uno. Sólo cuando se ejecuta una instrucción INP o una instrucción OUT es que se realiza una transacción con el espacio de E/S, lo cual pone M en cero.

2.4.7 Línea de ciclo de búsqueda (CB)

Durante el ciclo de búsqueda la CPUCR va a memoria para leer el código de la próxima instrucción a ejecutar. La línea CB se pone en uno para indicar el inicio del ciclo de memoria en que esto se va a realizar. Con esta señal es posible detectar, externamente esta situación.

Una situación donde se puede utilizar CB es para sincronizar la lectura de los códigos de instrucción de uno o varios coprocesadores trabajando en conjunto con la CPUCR. El ciclo de búsqueda es fácilmente detectado por los coprocesadores y éstos pueden entonces observar en el bus de datos si la instrucción que viene es de ellos o pertenece a la CPUCR.

2.4.8 Línea de ciclo de memoria (CM)

Todas las transacciones, ya sean instrucciones o datos, involucran un acceso al espacio memoria o un acceso al espacio de E/S. Este proceso se le generaliza con el nombre de ciclo de memoria. La línea CM le indica a los dispositivos externos a la CPUCR el inicio de un ciclo de memoria. Todos los ciclos de memoria contienen dos ciclos de reloj, CM identifica al primero tomando un valor lógico de uno.

Esta señal es útil para realizar transacciones de datos con dispositivos lentos, que no pueden completar la operación dentro del microsegundo que se dispone en el ciclo de memoria.

2.4.9 Línea de indicación de detenido (HLT)

Cuando la línea de salida HALT es igual a uno la CPUCR está indicando que se acaba de ejecutar una instrucción HLT. En esta situación la CPUCR esta totalmente detenida, es decir, no se ejecuta ninguna instrucción más y no responde a ninguna petición externa a través de las líneas $\overline{\text{INT}}$ y $\overline{\text{SDMA}}$. Todas las otras líneas, incluyendo

los buses de datos y direcciones, permanecen en su estado al ejecutarse la instrucción HLT. La única forma de iniciar la operación de la CPUCR es mediante una operación de reposición. Esta línea se conectaría directamente a una luz indicadora para que el usuario de la CPUCR se percate de la situación.

El mecanismo que provee la instrucción HLT con la línea de salida HALT resulta útil para que el procesador aborte todo tipo de actividad una vez que detecta una condición de error fatal en el sistema. Además, se le ha agregado a esta señal, una habilidad de oscilación en el momento en que se encuentre una instrucción no reconocida, con lo que se mejoran las posibilidades de detección de errores.

2.4.10 Línea de solicitud de interrupción ($\overline{\text{INT}}$)

Esta línea de entrada se activa por nivel bajo. Su función es la de recibir las solicitudes de interrupción de los periféricos que operan con este tipo de transferencia.

La línea $\overline{\text{INT}}$ puede ponerse a cero en cualquier momento. La CPUCR reconocerá la solicitud únicamente después de terminar de ejecutar la presente instrucción y si la bandera I se encuentra en cero. El estado $\text{INT} = 0$ entonces debe permanecer hasta que la CPUCR reconozca la solicitud. La aceptación se verifica por medio de la señal INTOK.

2.4.11 Línea de aceptación de interrupción (INTOK)

La línea de salida INTOK indica la aceptación de la solicitud de interrupción y marca los dos ciclos de memoria, donde la CPUCR está accediendo el vector de interrupción. Cuando INTOK se hace uno el periférico que pidió la interrupción debe remover su solicitud. Esto sirve para evitar que la subrutina de atención a interrupciones reconozca la solicitud que está procesando como una solicitud nueva.

Si el dispositivo no está diseñado para eliminar su solicitud de esa manera, entonces le corresponde, a la subrutina de servicio de interrupción, el borrar la solicitud antes de admitir nuevas interrupciones.

INTOK se puede usar también para tener un sistema de interrupciones vectorialmente dirigido. Normalmente el único vector de interrupciones que tiene la CPUCR se adquiere de las posiciones de memoria \$FF00 y \$FF01. Mediante un esquema de asignación de prioridades, el dispositivo que genera la interrupción puede suministrar directamente el vector necesario, aligerando considerablemente el tiempo de respuesta, en situaciones donde varios dispositivos hacen uso del mecanismo de interrupción. La sincronización para esta operación se establece con la señal INTOK.

2.4.12 Línea de solicitud de acceso directo a memoria ($\overline{\text{SDMA}}$)

Esta entrada se activa con un nivel lógico de cero. Su propósito es solicitarle a la CPUCR que libere el bus de direcciones, el bus de datos y las líneas L/E, M y CM para establecer una transferencia de datos directa entre algún dispositivo periférico y la memoria.

La solicitud de DMA (*Direct Memory Access*) es reconocida hasta que se termina la ejecución de la presente instrucción y si no hay una solicitud de interrupción ($\text{INT} = 0$) pendiente. Si existe una solicitud de interrupción, esta es atendida primero, saltando a la rutina de servicio de interrupción y ejecutando la primera instrucción. Una vez ejecutada la primera instrucción se atiende la solicitud de DMA.

Cuando la solicitud de DMA es reconocida, la línea BD se pone en uno al mismo tiempo que la CPUCR pone en tercer estado sus salidas del bus de direcciones, bus de datos, L/E, M y CM. El dispositivo puede entonces hacer uso de esas líneas para

iniciar el acceso directo a la memoria. Mientras el dispositivo mantenga SDMA en cero, éste tendrá el control de transferencias en el computador y la CPUCR estará detenida. Una vez que $SDMA = 1$ la CPUCR tomará control de nuevo y continuará su operación normal.

Debido a este modo de funcionamiento, los dispositivos que soliciten DMA no deben prolongar su dominio del bus por más de unos cuantos ciclos de memoria. Esto con el propósito de lograr que el dispositivo que solicitó el servicio de interrupción sea atendido oportunamente.

2.4.13 Línea de indicación de buses disponibles (BD)

La salida BD es activa en nivel alto y sirve para indicar a los dispositivos que solicitaron DMA que su solicitud ha sido atendida. Una vez que $BD = 1$ el bus de direcciones, el bus de datos y las líneas L/E, M y CM son puestos en tercer estado por la CPUCR. El dispositivo que solicitó el acceso directo a la memoria debe entonces hacer uso de las líneas cedidas para iniciar sus transacciones con la memoria.

Mientras el dispositivo mantenga $SDMA = 0$ la CPUCR mantendrá su condición de buses disponibles.

CAPÍTULO 3: Programación orientada a síntesis con las herramientas de Synopsys.

Para crear un circuito integrado, descrito con Verilog® y sintetizado con las herramientas de Synopsys, es necesario seguir una serie de reglas y convenciones que aseguren una síntesis fiel a la descripción del circuito.

Para cada diferente lenguaje de descripción de hardware (HDL, Verilog®, etc.), existen diferentes reglas de descripción, en el presente capítulo se tratarán las concernientes a Verilog®, que fue el lenguaje elegido para crear la CPU CR.

3.1 Lineamientos generales para la síntesis de circuitos usando Synopsys.

Comúnmente, cuando el estudiante comienza el desarrollo de circuitos utilizando un lenguaje de descripción de hardware como Verilog®, aprende a utilizar una serie de comandos y herramientas que le permiten desarrollar con mayor rapidez y facilidad el proyecto que realiza. Lamentablemente, no todos estos comandos son sintetizables por las herramientas de Synopsys, puesto que algunos de ellos describen comportamientos irreales o irrealizables en la práctica.

A la serie de instrucciones que son sintetizables por las herramientas de Synopsys, se les conoce como un *Verilog® subset*³. Estas son las instrucciones y comandos que el fabricante del software asegura que serán sintetizados si se usan adecuadamente.

³ *Verilog Subset*: Subconjunto de Verilog

El diseñador del circuito integrado es responsable de conocer las diferentes instrucciones que son sintetizables, para así, saber cuáles deberá utilizar para reemplazarlas, o inclusive, cuáles cambios al comportamiento del circuito deberá implementar.

No es necesario que el diseñador elimine por completo estas instrucciones de su modelo de programación, porque, aunque no sean sintetizables, son muy útiles a la hora de simular el comportamiento de los circuitos. Por ejemplo, el programador puede utilizar durante las etapas de diseño del circuito, instanciación por jerarquía de módulos, pero siempre debe tener en cuenta que en la etapa de preproducción o síntesis, todas las referencias de este tipo deben ser eliminadas. La instanciación por jerarquía de módulos se explicará más adelante.

En las etapas de planeamiento de un diseño, puede resultar de gran ayuda utilizar un esquema de trabajo similar al que se presenta en la Figura 3.1, en el cual se pueden distinguir las diferentes etapas que conlleva la creación de un nuevo circuito integrado. Siguiendo como base dicho esquema, en la etapa de preparación para la síntesis, todas aquellas partes del circuito que contengan instrucciones no sintetizables deberán ser modificadas, con el fin de tener una descripción completamente sintetizable.

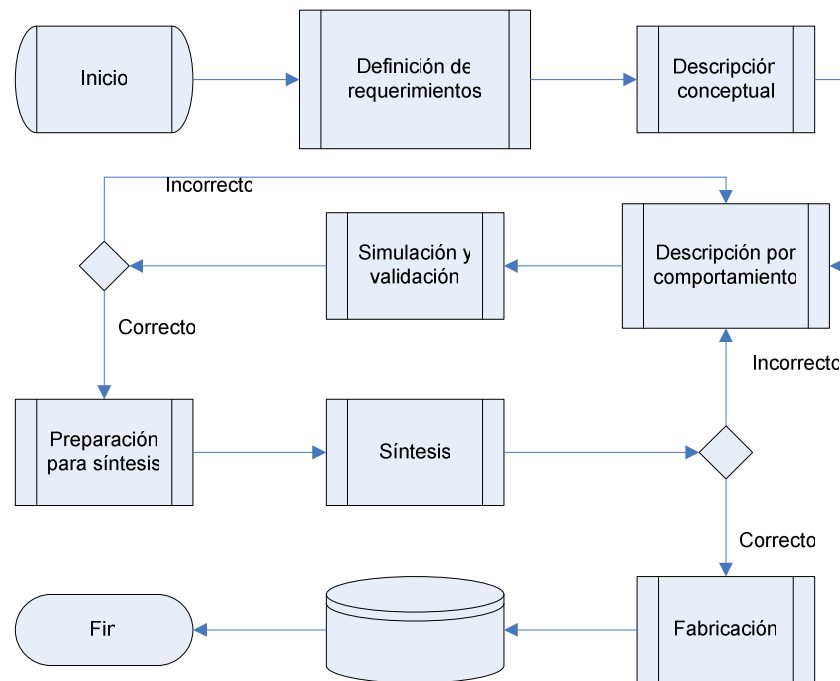


Figura 3.1 Flujo de diseño para un circuito integrado

También resulta conveniente que el diseñador conozca de antemano la librería que se utilizará para la creación del circuito. Este conocimiento de la librería, permitiría al programador, utilizar celdas prediseñadas y probadas por el fabricante del circuito integrado, reduciendo así el tiempo de diseño e implementación.

Siempre resulta importante un buen estilo de programación, con comentarios en el código y, sobre todo, un orden lógico en las instrucciones. La capacidad de Verilog® de trabajar con módulos, resulta una poderosa herramienta para crear código reutilizable y a su vez más fácil de entender.

Por último, el diseñador debe tener siempre en cuenta, que lo que se trata de describir es un circuito físico, compuesto de partes discretas como compuertas, multiplexores, etc. Entonces, resulta indispensable pensar en una implementación física

de los diseños, y no de una simple implementación en software. Es necesario siempre pensar igual que si fuésemos a armar el circuito con componentes individuales, y no dejar todo el trabajo de síntesis al compilador.

3.2 Síntesis de tareas del sistema y funciones

Distintas tareas del sistema pueden ser definidas por el fabricante del ambiente de simulación para ser usadas como parte del diseño. Dichas tareas abarcan comandos como *\$monitor*, *\$display*, *\$time* y *\$finish*. Todos los anteriores son ignorados por las herramientas de síntesis.

3.3 Directivas de compilador.

Las directivas de compilador causan que el compilador de Verilog® tome acciones especiales como definir macros e incluir archivos externos. Dichas directivas se mantienen en efecto hasta que sean modificadas o eliminadas; están activas entre módulos y archivos. Algunas de ellas son: *`define*, *`include*, *`ifdef*, *`else*, *`endif*, *`ifndef*, *`elsif*, *`endif*. Todas son sintetizables si se agrega el modificador *+v2k* en la línea de comandos de VCS.

3.4 Instanciación de Módulos y Jerarquía de Módulos.

La instanciación predeterminada de Verilog® es completamente sintetizable. Cualquier módulo puede ser instanciado dentro de otro módulo y pueden existir múltiples instancias del mismo módulo. Por otro lado, la instanciación por jerarquía de

módulos, es decir, de la forma *nombre_del_módulo.nombre_de_la_instancia*, no pueden ser sintetizados por el compilador de Synopsys. Por el momento no existe ningún equivalente sintetizable de esta forma de instanciación.

3.5 Tipos de Datos.

Existen tres tipos de datos: las redes, los registros y los parámetros. Los primeros especifican la conexión física entre los componentes, los registros indican variables y dispositivos de almacenamiento y por último, los parámetros, son en general constantes.

- Redes.

Las redes son constantemente manejadas por los dispositivos que conectan, y los valores son propagados automáticamente cuando el dispositivo cambia de estado. Los de tipo *wire*, *tri*, *wor*, *wand*, son sintetizables y reconocidos por VCS. Los dispositivos de tercer estado como *tri0* y *trireg* no son sintetizables, a no ser que existan dentro de la librería del fabricante.

- Registros.

Los registros pueden ser utilizados como variables y representar datos abstractos o reales. El más común es el *reg*, que modela los diferentes tipos de *flip-flops*. Hay que tener en cuenta que la iniciación de los registros no es sintetizable, es decir, su valor inicial siempre será desconocido.

- Parámetros.

Los parámetros son herramientas útiles cuando se requieren diseños escalables con longitudes de registros y buses que cambian con los requerimientos de la aplicación.

3.6 Memorias.

Las memorias son declaradas como arreglos bidimensionales de registros y sirven para modelar dispositivos de almacenamiento como memorias RAM o *Flash*. A pesar de que son simulables y sintetizables con las herramientas de VCS, se recomienda utilizar celdas de memoria creadas por el fabricante del chip, puesto que, el sintetizador podría generar resultados que distan de la realidad.

3.7 Bloques de Procedimiento.

Existen dos tipos de bloques de procedimientos, los bloques *initial* y los bloques *always*. Los primeros no son sintetizables en ninguna versión de Verilog®, puesto que implican una inicialización de variables que es irrealizable en la práctica. El orden de ejecución de los bloques de procedimiento es indeterminado y no es posible discernir cual de los bloques se ejecuta primero para un tiempo t .

La mayoría del código de Verilog® que es sintetizable debe estar contenido dentro de un *always*, cuya lista de sensibilidad responde a un flanco (flanco negativo: cuando una señal cambie de uno a cero, únicamente) o al cambio de estado de una señal (cuando pase de cero a uno o de uno a cero), pero nunca a una combinación de ambos.

3.8 Controles de Tiempo.

Los controles de tiempo ayudan al programador a simular las condiciones reales de operación de circuito en términos de su velocidad de procesamiento. El ejemplo clásico son los *delay*, que agregan un retardo de tiempo a una señal en la simulación, pero son ignorados a la hora de la síntesis. De no existir estos controles en Verilog se pondría estimar la velocidad máxima del circuito calculando el mayor tiempo que le toma a una señal recorrer la lógica combinacional más larga y tomando así el periodo de tiempo que ocupa la señal para tener el valor definido a la salida.

Otro control de tiempo es el símbolo @ que especifica una transición de señal, estos son más comunes en conjunto con bloques *always*, y la cual si es sintetizable, no presenta problema alguno.

Finalmente, el otro tipo de control de tiempo es la expresión *wait*, que como su nombre lo indica, espera a que ocurra un evento para continuar con la ejecución del resto de instrucciones.

Resulta evidente que la instrucción *wait* no es sintetizable, puesto que implicaría detener la ejecución y asignación de nuevos valores para todo el circuito, incluyendo a la lógica combinacional.

3.9 Asignaciones bloqueantes y no bloqueantes.

Las asignaciones bloqueantes a registros son aquellas que interrumpen la ejecución del programa hasta que el registro toma el valor deseado. Las asignaciones no bloqueantes, por el contrario, permiten la ejecución en paralelo de distintos pasos de proceso. Los dos tipos son sintetizables en Verilog®, pero no es posible combinar dos tipos de asignaciones dentro de un mismo módulo para un registro dado.

Por ejemplo, no es posible poner $V \leq 1$ (no bloqueante) y $V = A$ (no bloqueante) en un mismo módulo de Verilog®. El sintetizador de Synopsys lanzará un error fatal si se mezclan los tipos de asignaciones.

3.10 Síntesis de bloques CASE.

Uno de los principales obstáculos cuando se sintetizan circuitos lógicos secuenciales, radica en obtener una síntesis fiel al modelo que se trata de describir. Un error común sucede cuando se tratan de sintetizar multiplexores.

Como se sabe, los multiplexores tienen una salida determinada por sus bits de selección. Este tipo de comportamiento tiende a ser descrito intuitivamente con una sucesión de *if* anidados, pero, a la hora de sintetizar dichos grupos de código, estos se interpretan como codificadores de prioridad, que son considerablemente más lentos en ejecución que un multiplexor.

Entonces, para resolver este problema de síntesis, es necesario que se utilicen los bloques *case*, eso sí, tomando en cuenta los requerimientos de descripción de los mismos. Los bloques *case* deben estar completos o *full*, esto es, todas las posibles combinaciones de los bits de selección deben de ser tomados en cuenta, o en su defecto, poseer una cláusula *default*, de lo contrario, el *case* también se sintetizará como decodificador de prioridad.

Luego, todas las ramas de *case*, deben de ser paralelas, es decir, que el valor que tome el multiplexor al final de la ejecución, no dependa de varios valores de los bits de selección.

En distintos tipos de sintetizadores, como Synopsys, resulta posible utilizar directivas como el *synopsys full case* o *parallel case*, para evitar la necesidad de describir todas las combinaciones de entrada, pero su uso no se recomienda, ya que con frecuencia genera diferencias entre el circuito descrito y el circuito sintetizado.

Deben evitarse en cualquier bloque *case*, todas aquellas asignaciones complicadas, que puedan resultar en lógica no paralela, como *if* anidados dentro de operaciones algebraicas.

3.11 Buses de tercer estado y asignaciones continuas.

Para describir buses de tercer estado, existe actualmente sólo una posibilidad dentro de las herramientas de Synopsys. Esto es, declarar los buses como un registro *inout*, lo que nos permitirá leer del bus toda vez que se encuentre en tercer estado, y a su vez permitirá escribir datos en el bus, cuando sea necesario.

Para manejar las compuertas de tercer estado, debe utilizarse una asignación continua, mediante el uso de *assign*. Esta instrucción conecta una señal permanentemente a los componentes requeridos. En general, los *assign* se utilizarán para manejar combinatorialmente los pines de habilitación de los *buffer* de tercer estado. Debe evitarse el uso de asignaciones continuas excesivas (la menor cantidad posible para evitar redundancia de instrucciones que controlen el bus de datos), puesto que podrían darse diferencias entre el código descrito y el sintetizado.

CAPÍTULO 4: Síntesis de la CPUCR con las herramientas de Synopsys y la librería AMI 0.5um.

El principal objetivo de este proyecto de graduación, fue desde un principio, el generar un modelo en Verilog® para la CPUCR que se pudiera sintetizar con las herramientas de Synopsys. En el presente capítulo, se describe el proceso que se siguió para obtener el modelo final de la CPUCR, cuáles fueron los principales problemas y sus soluciones.

4.1 Estado inicial del proyecto.

En el inicio del proyecto se pensó en continuar los esfuerzos hechos por otros grupos de estudiantes en semestres anteriores, con el fin de crear una descripción de la CPUCR que fuera robusta y que cumpliera con los estándares requeridos por las herramientas de Synopsys para su síntesis.

Cuando se dio inicio al proyecto, se pensaba que el trabajo para obtener un modelo sintetizable iba a ser bastante sencillo, puesto que se tenía un modelo bastante funcional, pero que sintetizaba con errores. El enfoque inicial, sería entonces, tomar el diseño hecho por los grupos anteriores, entender qué se estaba haciendo mal y corregirlo.

Luego de una lectura intensa del código en Verilog® que se tenía, fue posible encontrar errores serios en la forma en que estaba descrito el circuito. Los errores no eran por la forma en que se usaba Verilog®, ni por la forma en que funcionaba el circuito, puesto que todas las simulaciones del modelo de Verilog® eran correctas.

Los errores radicaban en que se usaba un modelo de programación que no tomaba en cuenta las capacidades del sintetizador. Esto es, no todo lo que estaba descrito en el modelo era sintetizado correctamente.

También se encontraron problemas con la forma de programar, puesto que se tenía un único módulo que abarcaba todos los registros y funciones de la CPUCR. Encontrar una falla en el código se convertía en una tarea tediosa y poco efectiva, puesto que había que buscar dentro de una maraña de máquinas de estados y asignaciones de registros.

Debido a todo lo anterior, y en conjunto con los profesores a cargo del proyecto, se tomó la decisión de reescribir por completo la CPUCR. Utilizando como referencia los distintos documentos que se encuentran en la Escuela de Ingeniería Eléctrica acerca del tema de diseño para síntesis.

Comenzó así la investigación exhaustiva de los distintos tipos de programación, para encontrar la que más se adecuaba a nuestras necesidades.

4.2 Reorganización del código.

Para comenzar la nueva CPUCR fue necesario reorganizar el código y la forma en que se programaba el mismo.

Primero se dividió la descripción de la CPUCR en sus respectivos registros. Esto es, se crearon diferentes módulos para cada uno de los registros principales del procesador: el acumulador, el puntero de pila, el contador de programa y el estado. También se crearon módulos para los registros de la máquina de estados: estado presente y próximo estado. Finalmente, se crearon módulos individuales para describir señales menores, como la línea M, la línea L/E y la línea de HLT. A continuación se

presenta un esquema de los módulos que están presentes en el código de la CPUCR, luego se muestra un diagrama de sus interacciones, donde la flecha indica el sentido de la interacción.

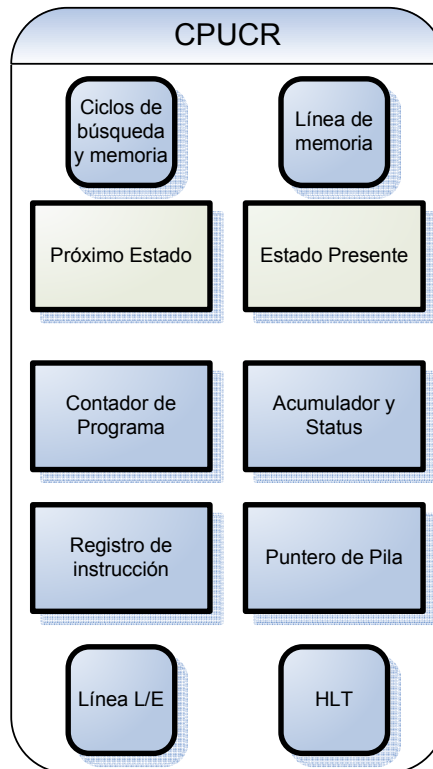


Figura 4.1 Módulos en que se encuentra dividido la CPUCR.

La división en módulos trae además una ventaja muy importante, si se requiere cambiar la forma en que funciona alguno de los registros, solamente será necesario reescribir el módulo que maneja dicho registro. O si, por ejemplo, se desea crear una *ALU*⁴, optimizada a mano, sin herramientas de síntesis, se puede crear en un módulo aparte, y ser incorporada al grueso del programa.

⁴ *ALU*: Unidad lógica aritmética por sus siglas en inglés.

Con respecto a la forma que se programa cada uno de los módulos, todos tienen una estructura similar, y sólo varía su complejidad dependiendo de la funcionalidad del módulo. Por ejemplo, para el contador de programa, es necesario describir lo que sucede en casi todos los estados de ejecución, mientras que en módulos más sencillos como el que maneja la línea M, sólo es necesario describir unos pocos estados, donde es relevante el funcionamiento del módulo.

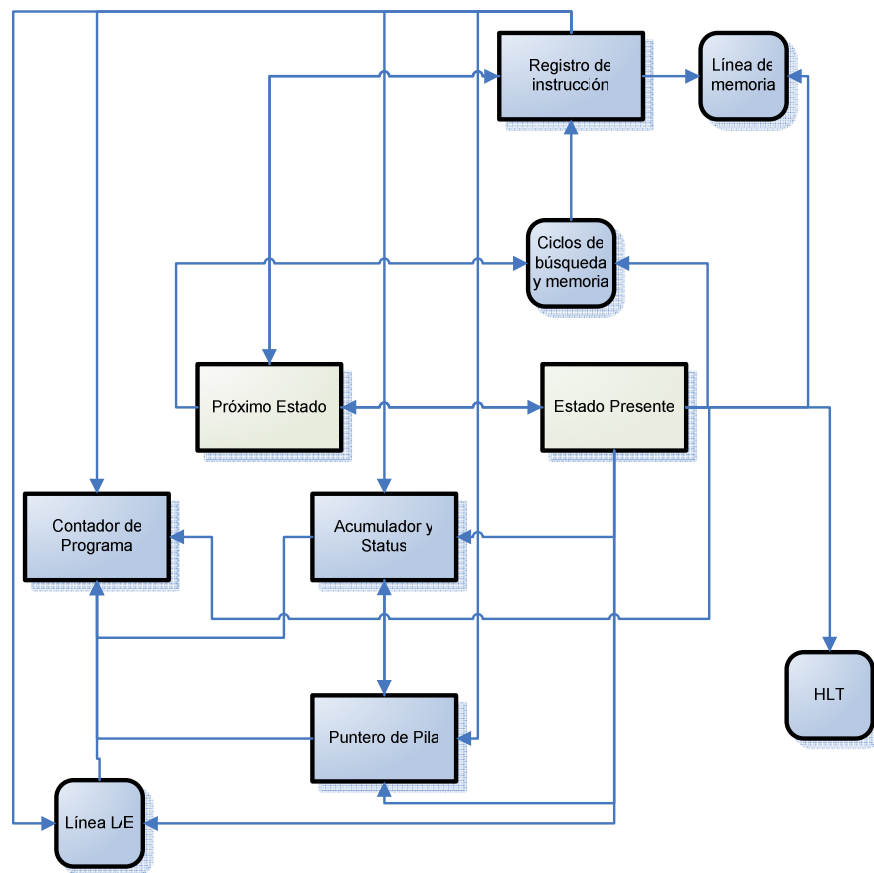


Figura 4.2 Diagrama de interacción entre los módulos de la CPUCR

Para ilustrar la forma de programación que se utilizó a lo largo de todos los módulos, se presenta el código del módulo M:

```

module reg_M(CLK, EstPresente, RI, M, RPS);
  input CLK, RPS;
  input [`MSB8:0] EstPresente, RI;

```



```

output M;
reg M;
`include "estados.v"
`include "Dec_Inst.v"

always @(EstPresente) begin
    if(!RPS) M<=1'b1;
    else begin
        case(EstPresente)
            Estado_0: M<=1'b1;
            Estado_1: M<=1'b1;
            Estado_5:case(RI)
                INP,OUT:M<=1'b0;
                default: M<=1'b1;
            endcase
            Estado_6:case(RI)
                INP,OUT:M<=1'b0;
                default: M<=1'b1;
            endcase
            default: M<=1'b1;
        endcase
    end
end
endmodule

```

Como se observa, el módulo está compuesto por una sección de declaración de variables, donde se define cuáles son los datos que entran y salen del módulo. En este caso específico, entran el reloj, la línea de reposición, el estado presente y el registro de instrucción, se tiene una única salida que es la línea M.

Posteriormente aparecen dos líneas de *include*⁵, estas líneas se repiten a lo largo de todos los módulos, puesto que permiten el acceso al archivo donde están definidos los códigos de instrucción y los estados de la máquina de estados.

Seguidamente se encuentra lo que quizás sea la parte más importante del módulo, su cuerpo en sí. El cuerpo de cada uno de los módulos se inicia con un *always* @. El tipo de dependencia que se asigna al *always* cambia de módulo a módulo, según se requiera actualizar los registros con el cambio del estado o con el flanco de reloj.

⁵ *Include*: directiva de compilador para incluir un archivo externo en formato verilog.

Luego viene la prueba de la línea de reposición (RPS), con esta se detecta una condición de reposición y se llevan a cabo las acciones requeridas por la misma. No todos los módulos poseen esta comprobación, puesto que algunos registros como el acumulador, no requieren ser inicializados a un estado conocido después de una reposición.

Por último viene el bloque *case* que se repite a lo largo de todos los módulos. Este bloque es el que representa la diferencia entre una síntesis de decodificadores de prioridad y un multiplexor. En descripciones anteriores de la CPUCR, se realizaban múltiples bloques anidados de *if-else*, los que, por definición, implican decodificación de prioridad. Al crearse bloques *case* paralelos y completos (*full case*), el programador se asegura que el circuito se sintetizará como multiplexores paralelos y no se producirán discrepancias entre el modelo sintetizado y el modelo en Verilog®.

Esta forma de programación antes descrita, se repite a través de todos los módulos, lo que hace que el código sea más fácil de entender, puesto que si se conoce el funcionamiento de un módulo, se conocerá el comportamiento general de todos.

4.2 Síntesis del Registro de Instrucción (RI)

Este registro interno se encarga de guardar el código de la instrucción que se va a ejecutar a continuación. Al observar cualquier diagrama de temporización de la CPUCR, se puede apreciar que el registro de instrucción se debería actualizar durante la primera mitad del ciclo de memoria y cuando la señal del ciclo de búsqueda se encuentra en estado alto. Este registro, además, está sujeto a la aparición de una señal de reposición, con lo cual es necesario llevar al registro nuevamente a cero.

Este es un comportamiento muy fácil de describir en Verilog®, como se muestra a continuación:

```
module reg_RI (CLK, CB, BUSDAT, RI, RPS);
    input CLK, CB, RPS;
    input [`MSB8:0] BUSDAT;
    output [`MSB8:0] RI;
    reg [`MSB8:0] RI;

    always @(negedge CLK)
    begin
        if (CB) RI<=BUSDAT;
        else RI <= RI;
    end
endmodule
```

Debe de recordarse que el registro de instrucción juega un papel vital en la ejecución de las instrucciones, puesto que constituye la forma de saber qué clase de operación debe de utilizar, sin embargo, no se tuvo ningún problema con la síntesis de este módulo debido a su simplicidad.

A continuación se muestra el circuito sintetizado del registro de instrucción y su vista en esquemático:

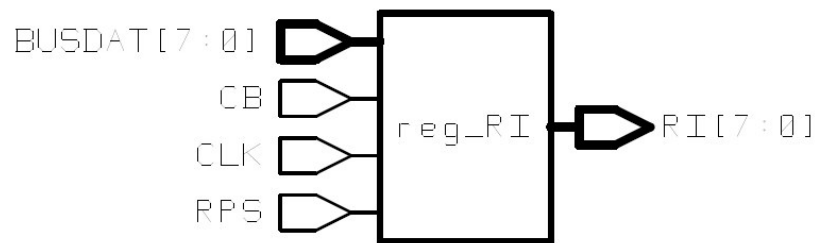


Figura 4.3 Vista esquemática del registro de instrucción.

4.3 Síntesis del Registro de Estado Presente

La síntesis de este registro resultó ser bastante fácil, puesto que simplemente toma los valores del registro de próximo estado y los almacena en sus *flip flop*. El código que describe este comportamiento se presenta a continuación, junto con los esquemas resultantes.

```

module reg_Est_Presente(CLK, EstPresente, ProxEst, RPS);
    input CLK, RPS;
    input [`MSB8:0] ProxEst;
    output [`MSB8:0] EstPresente;
    reg [`MSB8:0] EstPresente;
    always @(negedge CLK)
        begin
            if(RPS) EstPresente<=ProxEst;
            else EstPresente<=8'b0;
        end
endmodule

```

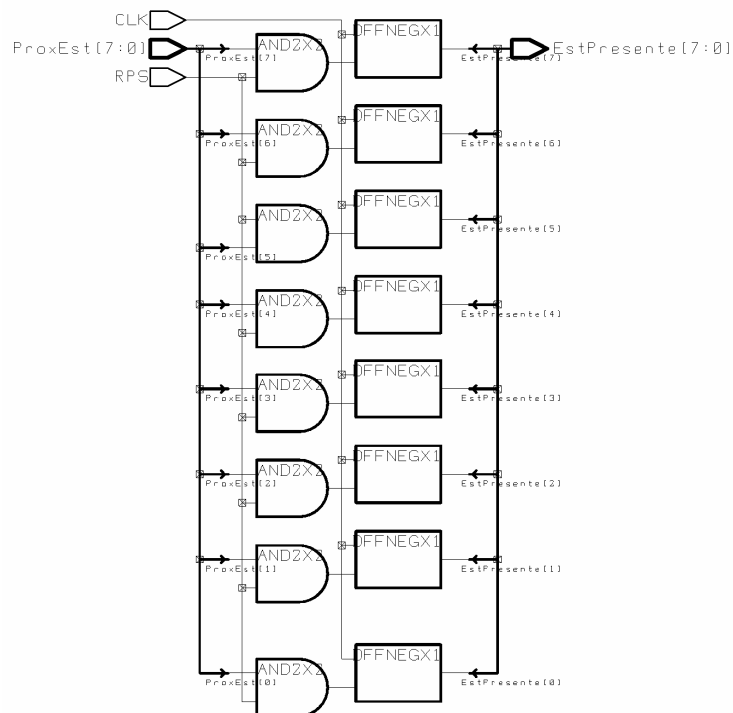


Figura 4.5 Circuito sintetizado del registro de estado presente

4.4 Síntesis del módulo de Ciclo de Búsqueda

El módulo de ciclo de búsqueda se utilizó para agrupar tres líneas menores de la CPU CR, el CB, el CM y la línea de Buses disponibles BD.

Inicialmente, el ciclo de memoria (CM) resultó un poco problemático, puesto que se requería que cambiara con cada flanco del reloj. Esto es trivial si se cuenta con

flip flops toggle, pero la librería de AMI que se posee, no cuenta con este tipo de componente. También fue necesario agregarle sensibilidad a la línea de reposición, para que la línea estuviera en un estado conocido al reiniciar el computador.

El código utilizado para la descripción de este módulo es el siguiente:

```
module reg_CB(CLK, EstPresente, RPS, CB, CM, BD, ProxEst, INTOK, SDMA);
    input CLK, RPS, SDMA;
    input [`MSB8:0] EstPresente, ProxEst;
    output CB, CM, BD, INTOK;
    reg CB, CM, BD, INTOK;
    `include "estados.v"
    always @(negedge CLK)
        begin
            if(RPS)
                begin
                    if(ProxEst==Estado_1) CB<=1'b1;
                    else CB<=1'b0;
                end
            else CB<=1'b0;
        end
        always @(negedge CLK) begin
if(!SDMA&&((ProxEst==Estado_15)|| (ProxEst==Estado_16))) CM<=1'bz;
            else if (!RPS) CM<=1'b0;
            else if ((ProxEst==Estado_1)&&(EstPresente==Estado_16))
CM<=1'b1;
            else CM<=~(CM);
            if ((ProxEst==Estado_15)|| (ProxEst==Estado_16)) BD<=1'b1;
            else BD<=1'b0;
            if
((EstPresente==Estado_21)|| (EstPresente==Estado_22)|| (EstPresente==Est
ado_23)|| (EstPresente==Estado_24)) INTOK<=1'b1;
                else INTOK<=1'b0;
            end
        end
endmodule
```

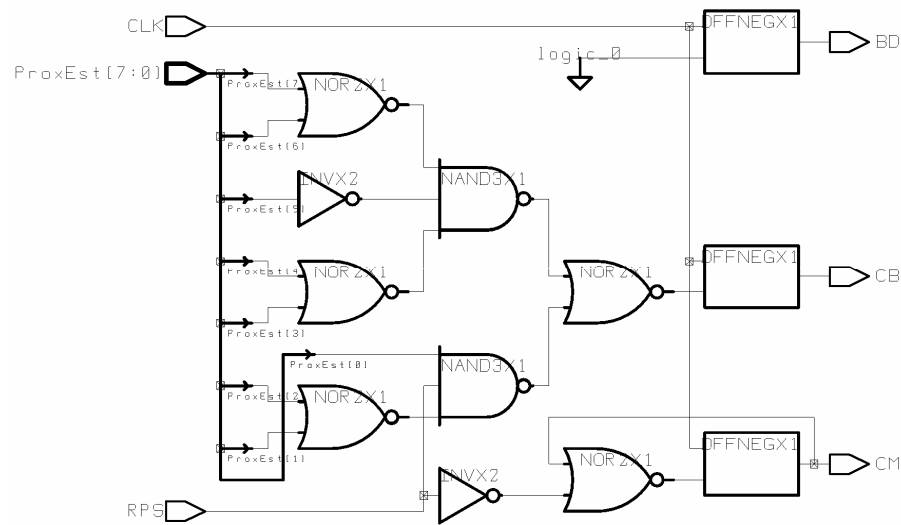


Figura 4.6 Circuito sintetizado para las líneas CB, CM y BD

4.5 Síntesis del módulo de Próximo Estado

A partir de este módulo, la complejidad de la descripción de los módulos fue aumentando considerablemente. Lo primero que hay que hacer notar, es que en este módulo se controla la ejecución de todas las instrucciones del procesador. Es una máquina de estados pura, que además tiene que monitorear las líneas de atención a interrupción y la línea de acceso directo a memoria, para llevar a cabo las acciones necesarias.

Este módulo también es sensible a la reposición, puesto que es necesario volver a estado cero cuando se dé una señal de reposición. La verificación de SDMA e INT se da en el estado 1. En el estado dos se dan los primeros cambios de flujo. Las instrucciones de sólo un ciclo de memoria saltarán a estado 1 después del mismo, todas las demás deberán saltar hacia estado 3 para continuar con su ejecución. Si la

instrucción no se reconoce, la ejecución saltará a estado 31, el mismo caso sucede si la instrucción es un HLT.

Cabe mencionar que para futuras mejoras o implementaciones de la CPUCR, es necesaria la creación de un registro de próximo estado más grande, puesto que en esta versión, se llegaron a utilizar todos los 32 estados. El código se encuentra en el Anexo

A.6

El circuito sintetizado es el siguiente:

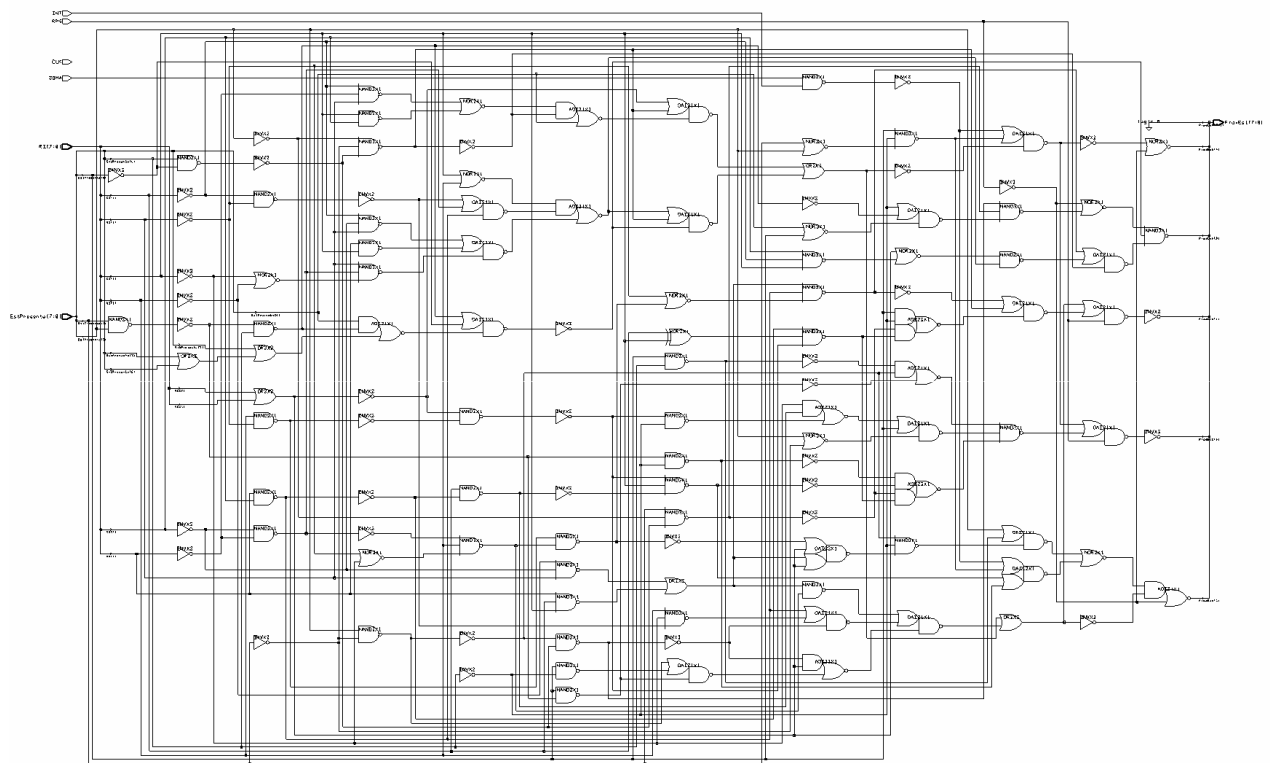


Figura 4.6 Circuito sintetizado del registro de próximo estado.

4.6 Síntesis del acumulador.

El acumulador es el segundo registro más complejo, luego del contador de programa. Su síntesis implicó una descripción detallada de sus funciones.

Su complejidad radica principalmente en el alto número de líneas que convergen en el acumulador. En ocasiones se interactúa con el puntero de pila, o con el bus de datos, o inclusive con los puertos de entrada y salida.

El registro de estado resultó ser el escollo más grande para la síntesis de este módulo. Inicialmente, se había definido un módulo exclusivo para este registro, pero era casi imposible actualizar las banderas que dependen directamente del acumulador, puesto que no se tenía una interfaz con los valores directos del acumulador. Se producían retardos inmanejables y una serie de problemas que desembocaron en la solución de incorporar este registro al módulo acumulador.

En el código se nota que las banderas meramente combinacionales como el signo y el cero, se ponen fuera del bloque *case*, para asegurarnos de una síntesis combinacional pura, sin *latches* inferidos.

Otras banderas resultaron de mayor complejidad, como la bandera de rebase. Se optó por un esquema en el cual, la bandera se actualiza hasta el primer estado de la siguiente instrucción, esto no corresponde con la descripción arquitectónica inicial, pero ahorra una gran cantidad de lógica y registros extra en el circuito.

Vale la pena recordar, que en el registro estatus aún sobran 3 bits que están siendo utilizados, y pueden servir en el futuro para crear nuevas banderas, como banderas de *borrow* y habilitación de puertos.

El código fuente del módulo acumulador, y el circuito sintetizado, no se muestran debido a su extensión, pero pueden ser estudiados en los anexos. A.5 y A.6

4.7 Síntesis del contador de programa

El contador de programa resultó ser el módulo más complejo y difícil de sintetizar. La complejidad radica en los diferentes tipos de saltos que deben realizarse en la ejecución de un programa.

Los saltos condicionales resultaron ser un verdadero reto a la hora de la síntesis, puesto que producían incoherencias a la hora de sintetizar el modelo. Todas estas incoherencias se subsanaron cuando se cambió la forma en que se describen los saltos condicionales.

Presentamos a continuación el código del estado 4 para un salto condicional, en el que es posible apreciar la nueva forma en que se asignarán los saltos.

```
BCC: begin
  case (S[0])
    1'b1: begin
      PC<=PC;
      RDR<=RDR+16'b1;
    end
    1'b0: begin
      PC<=BUSDAT[7] ? PC + {8'hFF,BUSDAT} : PC +
{8'b0,BUSDAT};
      RDR<=BUSDAT[7] ? PC + {8'hFF,BUSDAT} : PC +
{8'b0,BUSDAT};
    end
  endcase
end
```

Cada *salto condicional* tiene en su estructura, un *case* que interroga la bandera de estado que se requiere probar. Esto con el fin de obtener un multiplexor dos a uno, con puertos de 16 bits. Para este caso específico, cuando la bandera de *acarreo* se encuentra en 1, se debe de aumentar el registro de dirección en uno, para avanzar a la siguiente posición de memoria, el contador de programa puede mantenerse en su estado

anterior, puesto que en el estado 3 se había aumentado en uno, con el fin de calcular la siguiente posición de salto.

Si el caso contrario se da, es decir la bandera se encuentra en cero, es necesario hacer un salto de posición. La asignación se hace de una forma muy sencilla, extendiendo el signo del número de saltos. Como se mencionó anteriormente el *PC* se había aumentado anteriormente, permitiéndonos un cálculo rápido de la posición del salto, sin necesidad de registros extras.

Luego de resuelto el problema de síntesis con los saltos condicionales se procedió a probarlos, para encontrar que los programas de prueba que se habían diseñado, no probaban exhaustivamente las condiciones de salto y no salto. Cuando se corrieron los programas que sí probaban ambas condiciones, se encontró un error en el cálculo del salto, puesto que inicialmente no se extendía el signo, resuelto este problema, se reprodujo el bloque de código para cada uno de los diferentes tipos de saltos condicionales que posee la CPUCR.

Los saltos a subrutina y los *JMP*, no presentaron mayor problema con respecto a su programación, sólo que, por su mayor longitud y número de estados requeridos, fue necesario ir creando nuevos estados sobre la marcha, que en algunos casos, simplemente aumenta el puntero de pila y el registro de instrucción.

Las diferentes instrucciones que requerían del direccionamiento basado en la pila, fueron relativamente fáciles de crear. Se usaron una serie de concatenaciones de bits, para crear la nueva dirección deseada, a continuación se presenta un ejemplo de código:

```

PLA: begin
    PC<=PC+16'b1;
    RDR<={8'hFF,P-8'b1};
end

```

En el mismo se aprecia que por tratarse de un direccionamiento a la pila, se agregan los primeros ocho bits en uno, y luego se le agrega el puntero de pila. En este caso, como se trata de una instrucción *PLA*, fue necesario que el puntero de pila se redujera en uno antes de la asignación.

El código fuente y el circuito sintetizado se pueden apreciar en los anexos A.5 y A.6

4.8 Síntesis de los buses de tercer estado

La síntesis de los buses de tercer estado resultó ser un problema bastante grande y complicado, principalmente por el hecho de que las variables que son declaradas como *input*, no pueden ser asignadas como registros, y resulta necesario asignarlas como *inout*.

El primer acercamiento fue el de manejar la asignación de tercer estado dentro de cada módulo que requería un manejo de los buses. Esto resultó ser impráctico y la mayoría de las veces, incorrecto.

La solución final, fue la de crear varias asignaciones continuas para los buses, que creaban en la síntesis, los *buffer* de tercer estado que se requerían, un ejemplo de estas asignaciones continuas se presenta a continuación:

```

assign BUSDAT=(!LE&&(RI==PHS)&&Estado_4) ? S : 8'bz;

```

Se observa que la condición de prueba de la izquierda será la que maneje el estado de los buses.

Debe de tenerse especial cuidado a la hora de asignar las condiciones de los buses, puesto que si se crean asignaciones NO excluyentes, pueden darse serios choques de datos, especialmente cuando se trata de buses que interaccionan con periféricos y la memoria.

La síntesis final resultó ser exitosa y pasó todas las pruebas de asignación de tercer estado.

4.9 Pruebas realizadas al modelo sintetizado

Para probar que el modelo sintetizado estaba funcionando de la forma que se requería, se corrieron pruebas exhaustivas, creadas por estudiantes que habían descrito la CPUCR con anterioridad. Si cuando se ejecutaban los programas, se obtenían los mismos resultados que los del modelo arquitectónico requería, se asumía un funcionamiento correcto.

Es necesario aclarar una cosa, para poder simular el modelo sintetizado, es necesario especificar al compilador de VCS, la librería donde se encuentran descritas las distintas partes que se van a simular, en nuestro caso específico, las librerías de *AMI*. También es necesario agregar una directiva de *timescale* en el archivo resultante de la síntesis.

Los *scripts* utilizados durante todo el proyecto para simular el modelo sintetizado, se encuentran en los anexos.

4.10 Instrucciones para la creación de nuevas instrucciones de la CPUCR.

Como se mencionó anteriormente, el registro de instrucción de la CPUCR tiene 8 bits, lo que permite un máximo de 256 instrucciones diferentes, pero la CPUCR solamente tiene 54 instrucciones en su definición original.

Es posible crear nuevas instrucciones para el microprocesador, como una XOR o una multiplicación o división, si se utilizan ciertos lineamientos y pasos de diseño sencillos.

Esta es una muy breve guía de cómo hacerlo.

- 1- Defina la instrucción a realizar, su tipo de direccionamiento y los registros necesarios para su realización.
- 2- Asígnele un código de instrucción y agréguelo en el archivo Dec_Inst.v.
- 3- Cree un diagrama de temporización, tomando en cuenta todas líneas que pueden ser afectadas por la instrucción.
- 4- Determine el número de estados necesarios para realizar la instrucción, de ser necesario, aumente el ancho del registro de estado presente, para tener más de 32 estados.
- 5- Comience a modificar el código de la CPUUCR, primero comenzando con el módulo de cálculo del próximo estado.
- 6- Modifique luego el acumulador si es necesario, siguiendo el estilo de programación que se presenta en este trabajo.
- 7- Continúe modificando los módulos que estén involucrados en ejecución de la instrucción, asegúrese de inspeccionar cada diferente módulo, registros distintos pueden encontrarse en el mismo módulo, como el estado y el acumulador.

- 8- Si requiere crear nuevos registros internos, preferiblemente deberá describir su funcionamiento en un módulo aparte, e instanciar dicho módulo en el módulo principal.
- 9- Cree un programa de prueba para la instrucción.
- 10- Compile el código en Verilog®, con la ayuda de *VCS* y los *scripts* que se presentan en los anexos.
- 11- Si la simulación es exitosa, sintetice el módulo, usando los *scripts* para ese fin.
- 12- Simule el modelo sintetizado, con el mismo programa de prueba.
- 13- Si todo el proceso es correcto, podrá seguir agregando nuevas instrucciones.
- 14- Si encuentra alguna incoherencia, deberá volver al código en Verilog® para revisarlo.

Si se sigue este procedimiento general, con tiempo y calma, resulta relativamente fácil crear nuevas instrucciones. Utilice siempre las instrucciones ya creadas y probadas como ejemplo.

CAPÍTULO 5: Conclusiones y recomendaciones.

Luego de seis meses de arduo trabajo y una serie de dificultades y aciertos, resulta indispensable que se enumeren las principales conclusiones que se extraen de la realización del trabajo.

- 1- Para obtener una síntesis de un circuito correcta, que se comporte como se espera, es necesario conocer el lenguaje de descripción de hardware que se va a utilizar, la tecnología de fabricación, y tipo de instrucciones que son sintetizables para una determinada herramienta de síntesis, ya sea Synopsys o algún otro CAS.
- 2- La nueva descripción de la CPUCR en ocho bits, resulta flexible y más moderna que su predecesora, tiene una gran capacidad de expansión futura, sin perder de vista el propósito con que fue creada: la educación de futuros ingenieros.
- 3- El lenguaje de descripción de hardware Verilog® es un lenguaje sencillo y a su vez poderoso para la descripción de toda clase de circuitos digitales. En combinación con otras herramientas, puede convertirse en una herramienta de simulación y prueba.
- 4- Las herramientas de síntesis asistida por computadora, CAS; son de vital importancia en la creación y diseño de proyectos de microelectrónica muy grandes, que por su naturaleza no pueden ser sintetizados manualmente.

No es posible finalizar este proyecto, sin antes enumerar, las que son, a nuestro criterio, las principales recomendaciones para futuros proyectos de síntesis de circuitos, tanto de la CPUCR como de otra índole.

- 1- La CPUCR de ocho bits puede ser expandida en gran medida, deben agregarse nuevas instrucciones que la hagan más flexible.
- 2- La creación de un buen grupo de programas de prueba para los circuitos creados, ayudará en gran medida a la realización del proyecto, deben de crearse, si es posible, antes de que se tenga la descripción del circuito, para ir probándolo conforme se avanza.
- 3- El conocimiento previo de la tecnología de fabricación del circuito, resulta indispensable, para ahorrar tiempo en optimización y creación de estructuras de diseño.
- 4- Ciertas partes de la CPUCR deben ser optimizadas para una menor área y una mayor velocidad, principalmente en lo concerniente a la ALU y al cálculo de la próxima dirección.
- 5- Se requiere un mayor entendimiento de la herramientas de Synopsys, para poder crear un diseño desde cero y llevarlo durante todo el proceso de creación a través de una serie de herramientas compatibles.

BIBLIOGRAFÍA

Libros:

1. Arellano, R. y Cruz, R. **“Modelado y síntesis de la CPUCR”**, Sin editorial, Costa Rica, 2003.
2. Smith, D. **“HDL Chip Design”**, Doone Publications, Estados Unidos, 2000.
3. Steinvorth, R. y Vásquez, M. **“Estructuras de Computadoras Digitales”**, Sin editorial, Costa Rica, 1997.
4. Synopsys Inc. **“Chip Synthesis Workshop”**, Synopsys Educational Resources, Estados Unidos, 2002.
5. Thomas, D y Moorby Philip. **“The Verilog® Hardware Description Language”**, 3era Edición, Kluwer Academic Publishers, Estados Unidos, 1996.
6. WillametteHDL, **“Basic Verilog® with VCS”**, Sin editorial, Estados Unidos, 2002.

APÉNDICES

Apéndice 1 Manual introductorio para el uso de VCS

Manual Básico de Uso de VCS®

Por: Esteban Ortiz

Warren Alvarado

Introducción

El siguiente tutorial pretende introducir al lector al uso básico del simulador VCS®, parte de las herramientas de Synopsys® para el diseño de circuitos integrados.

Se basa en la experiencia adquirida durante la etapa de diseño de la CPU CR, en la que se utilizó esta poderosa herramienta para la simulación y diseño del microprocesador.

Este documento no pretende ser una guía exhaustiva de uso del programa, si no más bien, un punto de inicio para que el diseñador pueda comenzar a obtener resultados rápidamente.

Se supone durante todo el desarrollo del mismo, que el lector está familiarizado con el lenguaje de descripción de hardware Verilog®, y que además tiene un conocimiento básico de los sistemas operativos similares a UNIX.

¿Qué es VCS®?

VCS® es un sistema integrado de simulación de lenguajes de descripción de hardware, especialmente Verilog®. Comprende un compilador de HDL, llamado *Presto* y un ambiente de simulación llamado VirSim®.

Tiene la capacidad de trabajar con distintos tipos de lenguajes, como *System Verilog®*, *System C®* y *VHDL®*.

VCS® significa simulador compilado de Verilog® (*Verilog® Compiled Simulator*), puesto que cuando se simula el circuito, se crea un archivo en lenguaje C, que es compilado en lenguaje de máquinas, acelerando la velocidad de simulación del diseño.

Tutorial paso a paso para utilizar VCS®

A continuación se desarrollará un ejemplo de diseño en Verilog® que implica la creación de un multiplexor de 8 entradas y una salida. Así como su banco de pruebas. El circuito lógico sintetizado con las herramientas de Synopsys® se muestra a continuación.



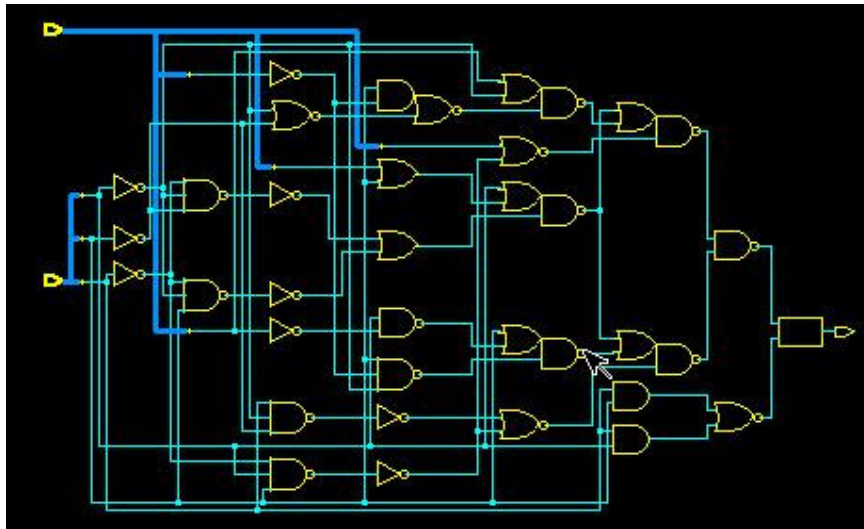


Figura 1 Circuito sintetizado del multiplexor 8 a 1

Paso uno

Copie el siguiente programa de Verilog® (ejemplo1.v) en su carpeta de trabajo, exactamente como se muestra:

```
module mux8a1(dir,datos_in,dato_out);
input [2:0] dir;
input [3:0] datos_in;
output dato_out;
reg dato_out
always @( dir) begin
    case(dir)
        0: dato_out<=datos_in[dir];
        1: dato_out<=datos_in[dir];
        2: dato_out<=1'b0;
        3: dato_out<=1'b1;
        4: dato_out<=datos_in[3];
        default: dato_out<=1'bz;
    endcase
end
endmodule
```

Paso dos

En una consola de texto digite `vcs ejemplo1.v`. La línea anterior comenzara a compilar el archivo de Verilog® para determinar si cumple con el estándar de Verilog® y luego llevar a cabo la simulación, de haber errores o advertencias aparecerán líneas indicando el archivo de origen y la línea donde está la falla.

Por ejemplo, en el archivo de `ejemplo1.v` se omitió un punto y coma (;) en la línea 5, lo que es interpretado como un error al ejecutar la línea seis. Corrija el error agregando un punto y coma (;) al final de la línea cinco, la cual define el registro `dato_out`.

Compile nuevamente el archivo de Verilog® con la línea de comando `vcs ejemplo1.v`. Deberá aparecer la línea de texto *Top Level Modules* indicando los módulos de Verilog® que fueron leídos correctamente

Paso tres.

Copie el siguiente texto en un archivo llamado `banco1.v` en su carpeta de trabajo. El archivo consiste en un banco de pruebas para ejercitar el multiplexor que se describe en `ejemplo1.v`.

```
module banco1();
    wire out;
    reg [2:0] dir;
    reg [3:0] datos;
    mux8a1 mux(dir, datos,out);
    initial begin
        #5 datos=4'hE;
        #5 dir=0;
        #5 dir=1;
        #5 dir=2;
        #5 dir=3;
        #5 dir=4;
        #5 dir=5;
    end
endmodule
```

```

        #20 $finish;
    end
endmodule

```

Paso cuatro

En una consola de texto digite la linea de comando `vcs ejemplo1.v banco1.v`. Asegúrese de que la compilación sea correcta y de que no halla errores.

Paso cinco

Con el paso anterior se puede observar que cada vez que se ejecuta VCS® de esta forma se recompilan todos los módulos, aumentando el tiempo de prueba y simulación. Si a la línea de comando usada para compilar se le agrega `-Mupdate`, se compilaran solo los módulos que hallan sido modificados desde la ultima compilación.

Paso seis

La línea de comando que se ha utilizado para compilar no ejecuta los archivos de Verilog®, solo revisa que todas las reglas de Verilog® sean cumplidas. Para revisar el funcionamiento del circuito se le debe agregar a la linea de comando `-R`, que viene de la palabra en ingles *run*, con lo cual se simulará el circuito.

Ejecute la linea `vcs ejemplo1.v banco1.v -Mupdate -R`. Con lo cual obtendrá un reporte de simulación de VCS®.

Paso siete.

Por lo general, lo que se busca al simular un circuito es ver las transiciones de los estados y las líneas de una forma gráfica, lo cual no se obtiene con el comando anterior. Para una simulación interactiva debe agregar a la línea de comando *-RI* del inglés *Run Interactive*. Esto abrirá la interfaz gráfica llamada *VirSim®*, donde podrá realizar diferentes tareas de simulación. Su ventana principal puede observarse a continuación.

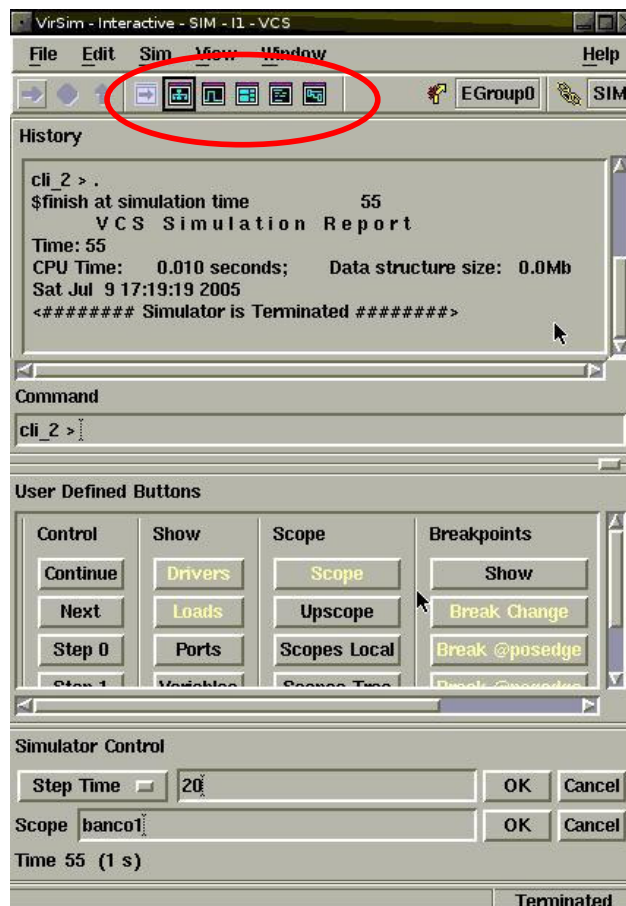




Figura 2 Ventana principal de VirSim®, se muestra la ubicación de los botones principales.

En la Figura 2 se muestra la ubicación de los principales botones que se usan durante la síntesis.

Paso ocho.

Para poder ver los resultados de la simulación es necesario abrir dos nuevas ventanas, la de jerarquía  y la de formas de onda . La de jerarquía muestra la relación entre los módulos y la de formas de onda es como tener un osciloscopio que se conecta al circuito y muestra el comportamiento de las señales. Haga clic sobre los botones correspondientes para abrir dichas ventanas. Las ventanas que debe de obtener se muestran a continuación.

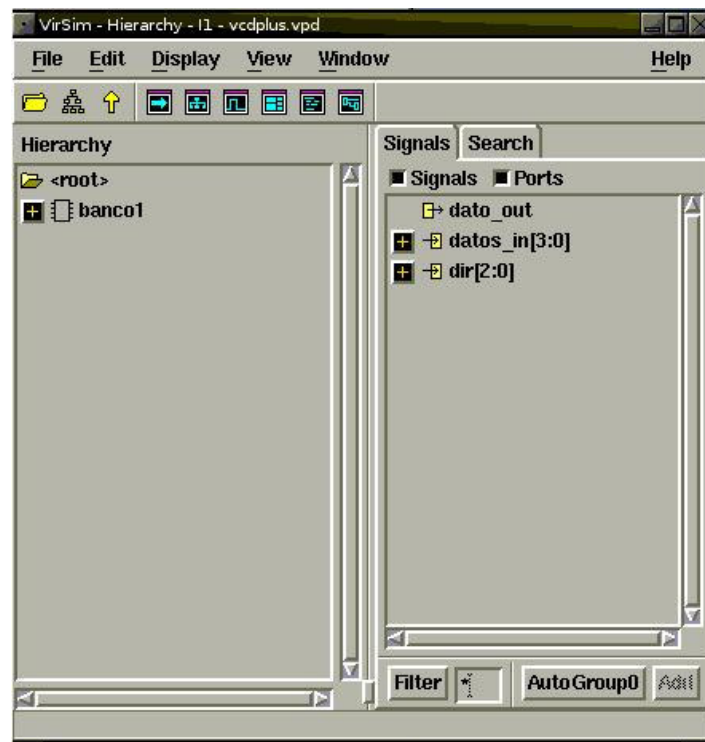


Figura 3 Ventana de jerarquía de módulos

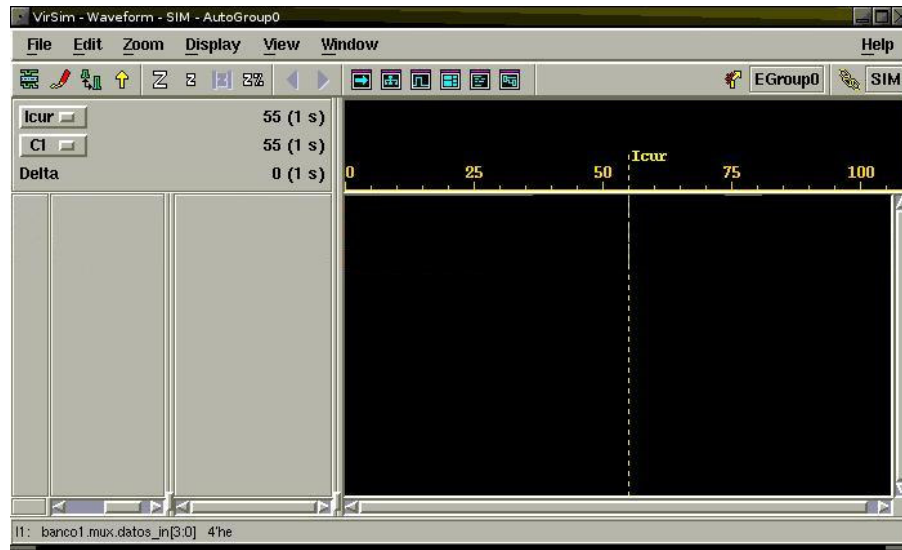



Figura 4 Venta de formas de onda

Ahora es necesario agregar las líneas que se quieran observar a la ventana de forma de onda. Vaya a la ventana de jerarquía y haga clic sobre el signo más a la izquierda de banco uno. Esto expande el árbol de jerarquía del módulo de mayor nivel, y se muestran los módulos que se encuentran en la misma.

Haga clic sobre el módulo mux, para observar las señales que entran y salen del mismo. Seleccione cada una de ellas (`dato_out`, `datos_in` y `dir`) y haga clic sobre el botón Add en la parte inferior, o arrástrelas utilizando el botón central del Mouse. Aparecerán los nombres de las líneas en la ventana de formas de onda, junto con una serie de barras grises.

Paso nueve

Ahora procederá a correr la simulación, haciendo clic sobre la flecha verde  que se encuentra en la ventana principal de VirSim®. Deberá obtener la simulación de la figura siguiente

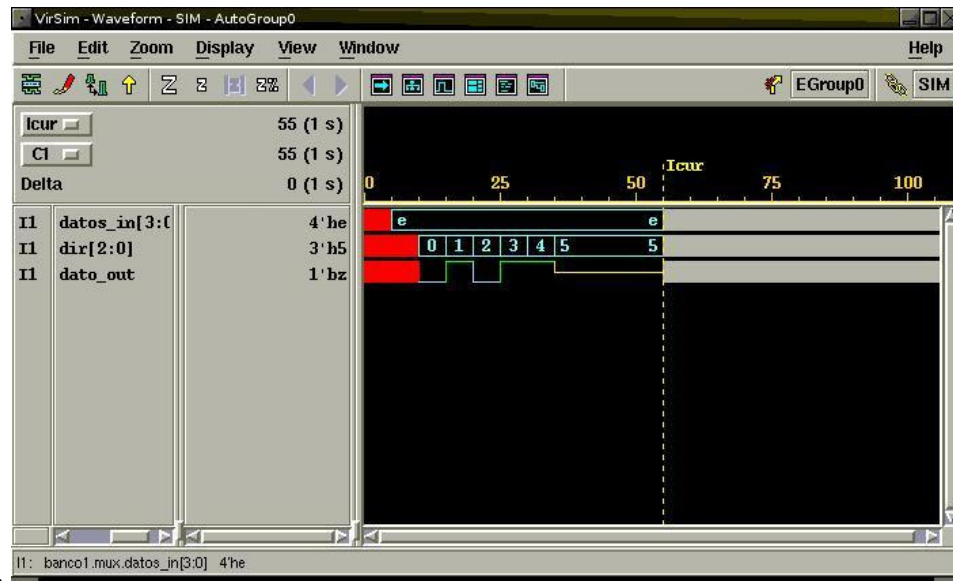





Figura 5 Simulación realizada en VirSim®

Las barras sólidas en color rojo representan valores indeterminados de las señales, las barras grises representan que la simulación para esos espacios de tiempo no se ha realizado. Para disminuir o ampliar el rango visible de tiempo de la simulación, puede hacer clic sobre los botones ,  y  que se encuentran en la barra de herramientas de la ventana de forma de onda.

Para cambiar la base con que se representan los valores de los buses, puede hacer clic sobre la columna a la derecha del nombre del bus, aparecerá el menú de cambio de base que se muestra en la figura siguiente.

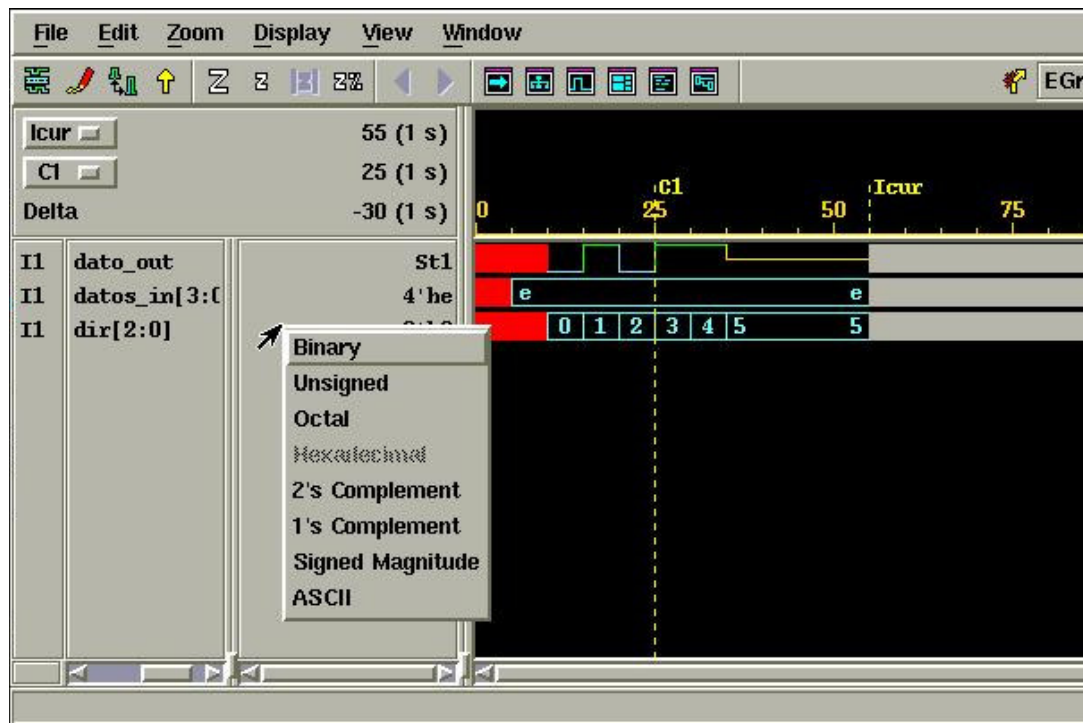




Figura 6 Menú de cambio de base en el simulador

Paso diez

A menudo será necesario observar nuevas líneas que no se simularon al principio. Estas pueden agregarse de la misma forma que se mencionó anteriormente, pero encontrará que las nuevas líneas aparecerán como barras grises sin simular. Para simular esas nuevas líneas, sin necesidad de salir de VirSim®, sólo tiene que ir al menú *Sim* de la ventana principal, y hacer clic sobre *Re-Exec*. Esto reiniciará la simulación y la llevará al punto inicial, puede hacer clic nuevamente en la flecha verde , para simular las nuevas líneas, junto con las anteriores.

Paso once.

Otra funcionalidad importante de VCS®, es la facilidad que brinda para ver el código en Verilog® que se está simulando. Deberá agregar la opción *-line* a la línea de comandos, por ejemplo, `vcs ejemplo1.v banco1.v -Mupdate -RI -line`. Para ver el código, basta con hacer clic en el botón , con el fin de abrir la ventana de código que se muestra en la siguiente figura. Note que esto se debe hacer ANTES de iniciar la simulación.

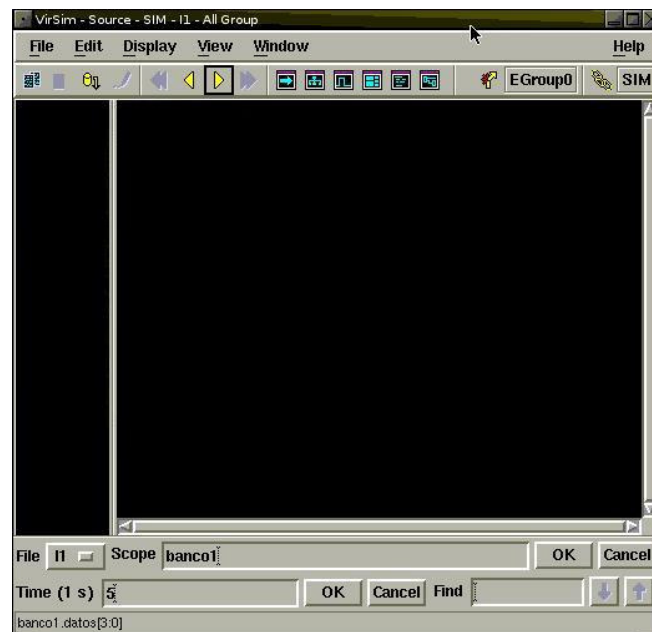





Figura 7 Ventana de código fuente

Ahora deberá de cargar los archivos que contienen el código fuente, haciendo clic en el menú *File*, y sobre la opción *Load Sources*. Haga clic en *Ok*, puesto que la mayoría de las veces los archivos aparecerán en la lista de archivos a cargar. Ahora haga clic en el botón , que se encuentra en la barra de herramientas de la ventana de código. Elija la opción por defecto. Con esto se activarán los botones  y , que le

servirán para correr la simulación paso a paso, además, el archivo que contiene la primera instrucción a ejecutar se mostrará en pantalla, como en la siguiente figura.

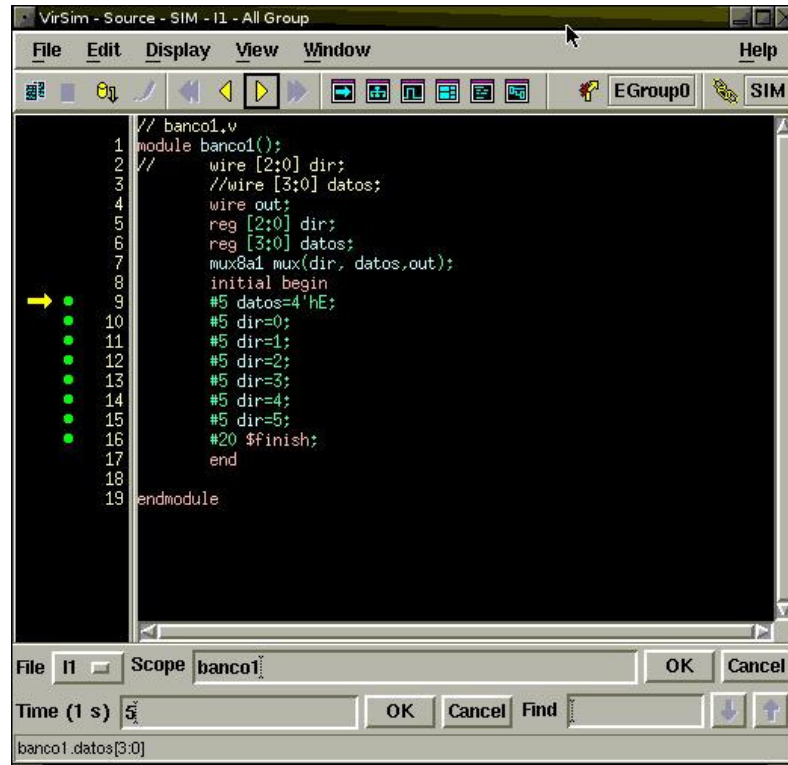


Figura 8 Código del programa, como aparece en la ventana de código.

Paso doce.

Por la facilidad que brinda Linux de tener varias consolas abiertas, existirán ocasiones en las cuales usted modifique el código del circuito mientras se encuentra en una simulación. Para no tener que salir de VirSim® y volver a compilar el código, VCS®, brinda una opción en el menú *Sim*, llamada *Rebuild and Re-exec* que vuelve a compilar todos los archivos que fueron invocados en la compilación inicial, y además reinicia la simulación al tiempo cero.

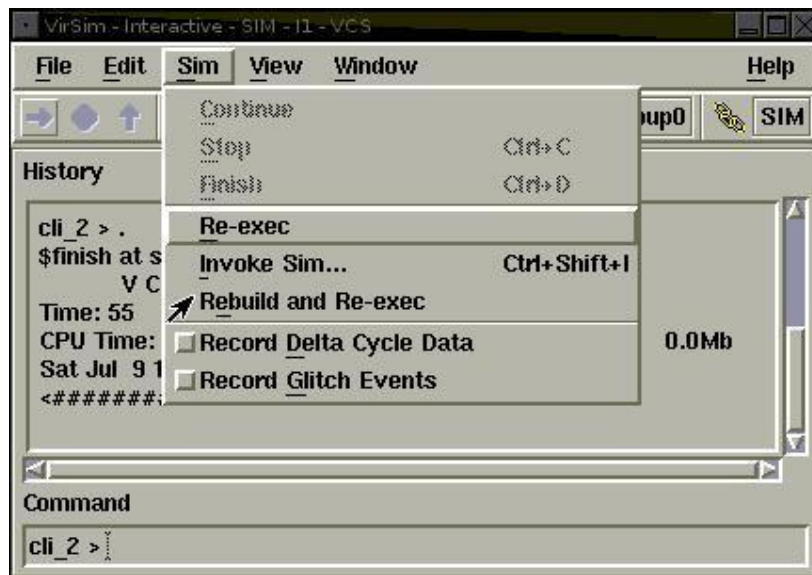


Figura 9 Menú Sim, que muestra la opción *Rebuild and Re-exec*

Paso trece.

Por último, cuando se simula un circuito complejo o muy grande, es probable que tenga que abrir y cerrar el simulador en muchas ocasiones, a lo largo de todas las etapas del diseño. Cada vez que se vuelve a abrir VCS®, se comienza con un ambiente de trabajo desde cero, es decir, deberá volver a agregar las líneas que quiere simular a la ventana de formas de onda, deberá volver a cargar los archivos fuente, etc. Para evitar este inconveniente, y aumentar la velocidad del proceso de diseño, es posible guardar la configuración que se está usando, y además la disposición de las ventanas en la pantalla. Cuando tenga un ambiente de trabajo completo, y a su gusto, haga clic sobre el menú *file* de cualquiera de las ventanas de *VirSim*®, ahí encontrará la opción *Save Configuration*, la cual le permitirá guardar su espacio de trabajo actual. Cuando reinicie *VirSim*® y quiera volver a dicha configuración, bastará con ir al menú *file* y seleccionar *load configuration*, y seleccionar la configuración deseada, en un instante tendrá su

escritorio de la misma forma en que lo dejó. Puede tener múltiples configuraciones, dependiendo del tipo de prueba que realice el circuito, y las líneas que quiera observar en una determinada simulación.

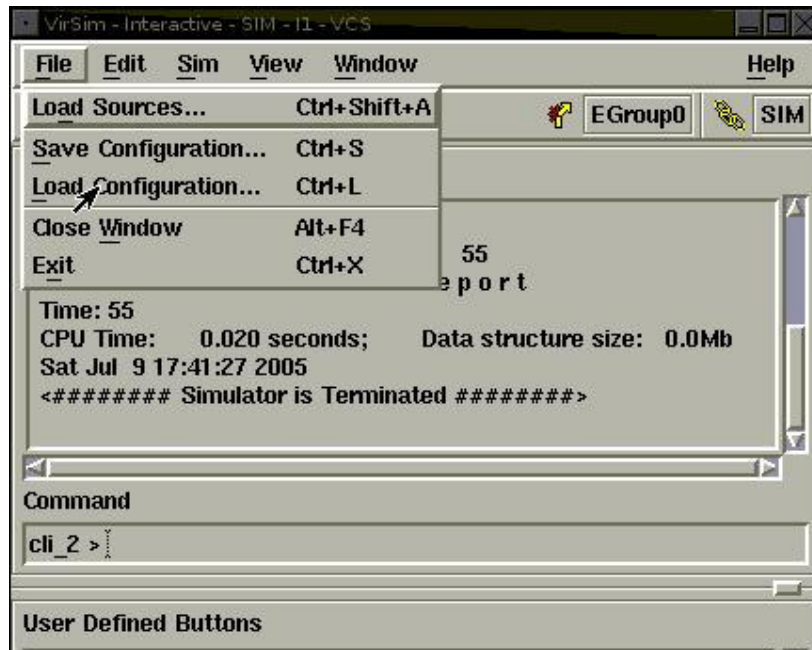


Figura 10 La opción *Load Configuration* del menú *file*.

Scripts para simulación en VCS®

A menudo resulta útil crear un *script* en algún interpretador de comandos, con el fin de evitar digitar la línea de comandos de VCS® cada vez que se quiere simular un diseño. Esto se logra fácilmente con la ayuda de un editor de texto. A continuación mostramos un ejemplo, que puede servir para simular el diseño ejemplo1.v que se estudió en este documento, guárdelo con el nombre de *compilar.scr*

```
#!/bin/bash
# Script para simular ejemplo1.v en VCS®
# Por Esteban Ortiz y Warren Alavarado
# Con # se hacen comentarios en el script
vcs ejemplo1.v bancol.v -RI -Mupdate -line
```


Ahora sólo basta cambiar los permisos de ejecución del archivo, para comenzar a utilizarlo, con el comando *chmod a+x compile.scr*.

Finalmente, para ejecutar el *script*, sólo hace falta digitar *./compile.scr* en la línea de comandos, en el directorio donde se guardó el *script* y se encuentran los archivos de Verilog®.

Algunas opciones de compilación útiles.

A continuación se enumeran algunas opciones de compilación de VCS® que se utilizaron durante la simulación de la CPU CR, pero que no son necesariamente requeridas para simular otros circuitos.

+sysvcs: Esta opción es necesaria si se quiere poder mezclar la sintaxis de Verilog® clásica, con la sintaxis de *System Verilog*®, un lenguaje de descripción de hardware que permite además, la descripción de circuitos analógicos.

+v2k: Esta es una de las más importantes opciones que se tienen, puesto que permite simular código de Verilog® que contenga declaraciones e instrucciones que cumplen con el estándar de Verilog® 2000, de lo contrario, estas declaraciones serán detectadas como errores de sintaxis.

-v: A menudo es necesario simular un circuito descrito en Verilog® después de sintetizarlo con las herramientas de *Synopsys*®. También es común que los archivos sintetizados requieran el uso de librerías de componentes específicos, proporcionadas por los fabricantes de *ASIC*. Esta opción especifica la librería a utilizar para la simulación del circuito. Su sintaxis es la siguiente: *vcs <Archivo_a_simular> -v <Librería> -Opciones_adicionales*

-notice: Esta opción brinda información adicional sobre distintos errores encontrados por VCS®, aparte de la información básica.

Bibliografía

- Willamette HDL. *Basic Verilog® with VCS®*. Willamette HDL, 2002.

ANEXOS

A.1 Script para para la simulación del modelo sintetizado

```
#!/bin/bash
#Script para simular el modelo sintetizado de la CPUCR
#Sustituir los archivos *.v por los del modelo a simular
#Sustituir la librería por la librería a utilizar.
#Creado por Esteban Ortiz 2005

vcs CPUCR_out.v Reloj.v Simulador.v Memoria.v Puertos.v Periferico.v -RI -
Mupdate -line +sysvcs +v2k -v iit06_stdcells_pads.v
```

A.2 Script para la simulación del modelo en Verilog®

```
#!/bin/bash
#Script para simular el modelo en Verilog® de la CPUCR
#Sustituir los archivos *.v por los del modelo a simular
#Creado por Esteban Ortiz 2005
vcs CPUCR_nueva.v Reloj.v Simulador.v Memoria.v Puertos.v Periferico.v -
RI -Mupdate -line +sysvcs +v2k -notice
```

A.3 Script para la síntesis desde una consola

```
#Script para sintetizar el modelo en Verilog® de la CPUCR en una consola
#Ejecutar dc_shell -f <nombre_del_script>
# Creado por Johannes Grad
# Modificado por Esteban Ortiz 2005
link_library=target_library={iit06_stdcells_pads.db}#Librería objetivo
define_design_lib work -path . #Define el directorio de trabajo
read -f Verilog CPUCR_nueva.v #Lectura del archivo de diseño
set_flatten true -effort low #Esfuerzo de mapeado
Verilog out_show_unconnected_pins = "true";
max_area 0.0 #Restricción de area a utilizar
current_design CPUCR #Nombre del diseño principal
set_max_fanout 8.0 CPUCR #Fanout por componente
create_clock CLK -name RELOJ -period 8 #Crea el reloj
set_dont_touch_network RELOJ
compile -ungroup_all -incremental_mapping -map_effort low #compilación
check_design #chequea el diseño
write -f Verilog® -output CPUCR_out.v #archivo destino
write -hier -output CPUCR_out.db #archivo destino en .db
report_timing > TEMP.rep #Reporte de tiempo
report_area > AREA.rep #Reporte de área
quit
```

A.4 Script para la síntesis desde una consola gráfica

```
#Script para sintetizar el modelo en Verilog® de la CPUCR en una consola
#gráfica
#Ejecutar design_analyzer -f <nombre_del_script>
# Creado por Johannes Grad
# Modificado por Esteban Ortiz 2005

link_library=target_library={iit06_stdcells_pads.db}
define_design_lib work -path .
read -f Verilog® CPUCR_nueva.v
set_flatten true -effort high
Verilog®out_show_unconnected_pins = "true";
max_area 0.0
current_design CPUCR
set_max_fanout 8.0 CPUCR
create_clock CLK -name RELOJ -period 6
set_dont_touch_network RELOJ
compile -ungroup_all -incremental_mapping -map_effort low
check_design
write -f Verilog® -output CPUCR_out.v
write -hier -output CPUCR_out.db
report_timing > TEMP.rep
report_area > AREA.rep
designer=Ortiz/Alvarado    #Nombre del diseñador
company="U.C.R"          #Compañía
```

A.5 Circuitos esquemáticos sintetizados para la CPUCR.

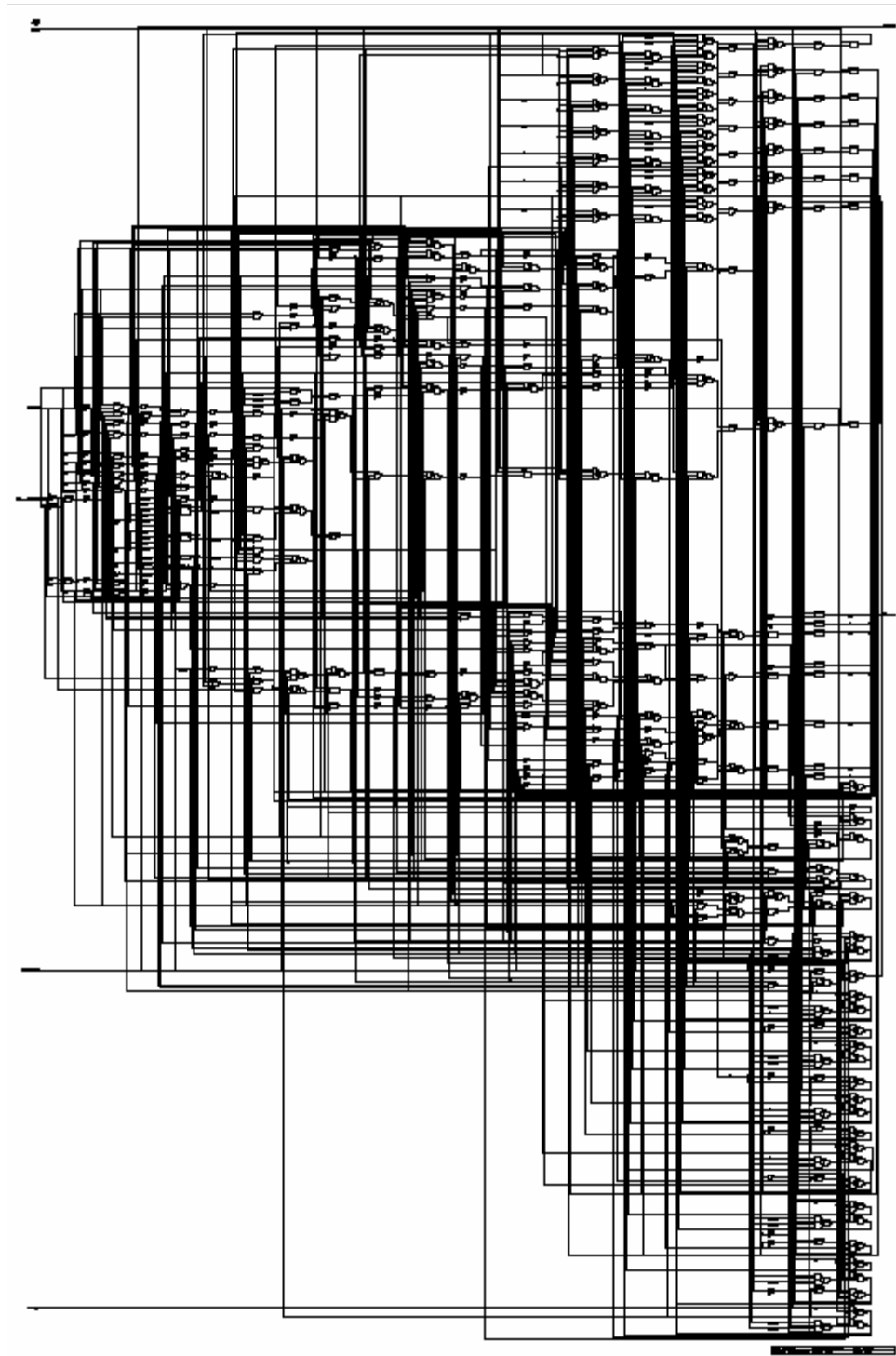


Figura A.5.1 Circuito sintetizado del acumulador

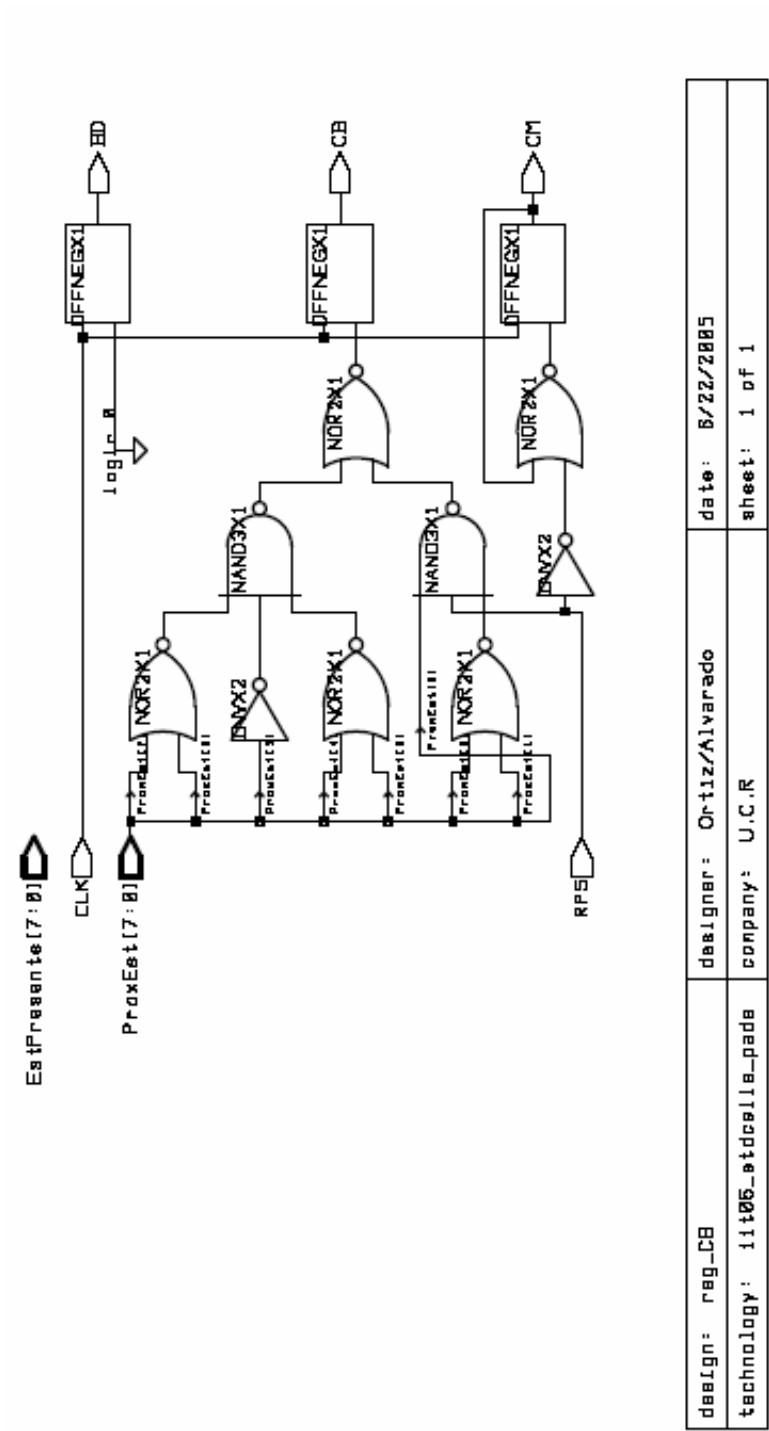


Figura A.5.2 Circuito Sintetizado del módulo ciclo de búsqueda

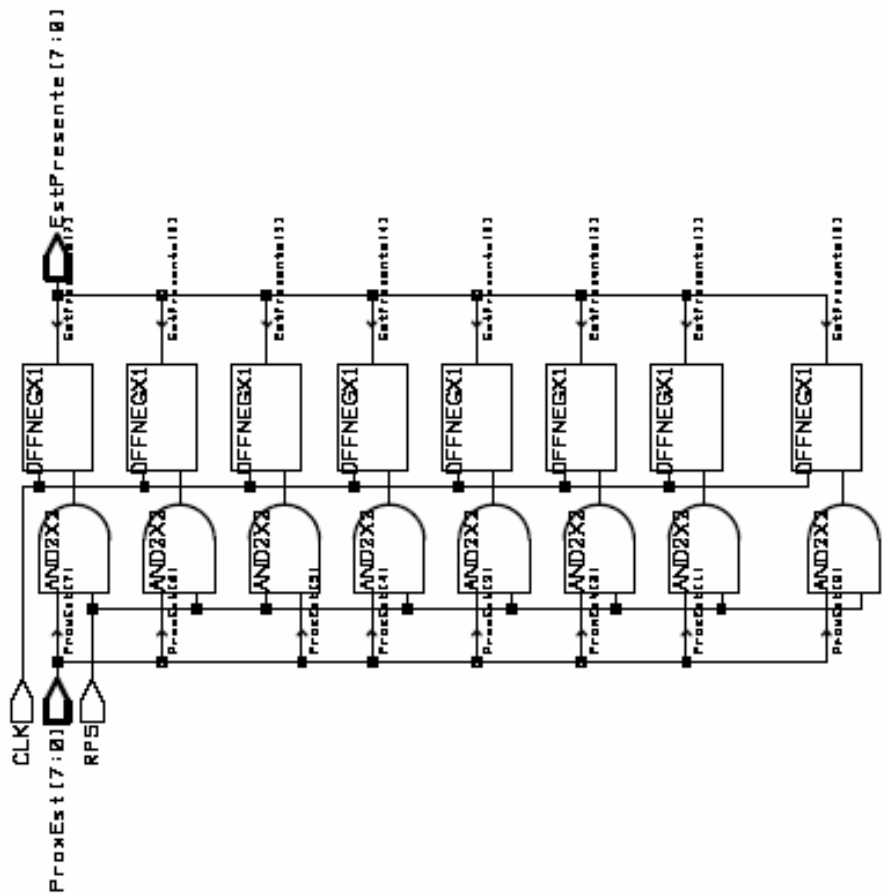


Figura A.5.3 Circuito Sintetizado del módulo de estado presente

design: reg_Est_Presente	designer: Ortiz/Alvarado	date: 6/22/2005
technology: 11t06_stdcells_pada	company: U.C.R	sheet: 1 of 1

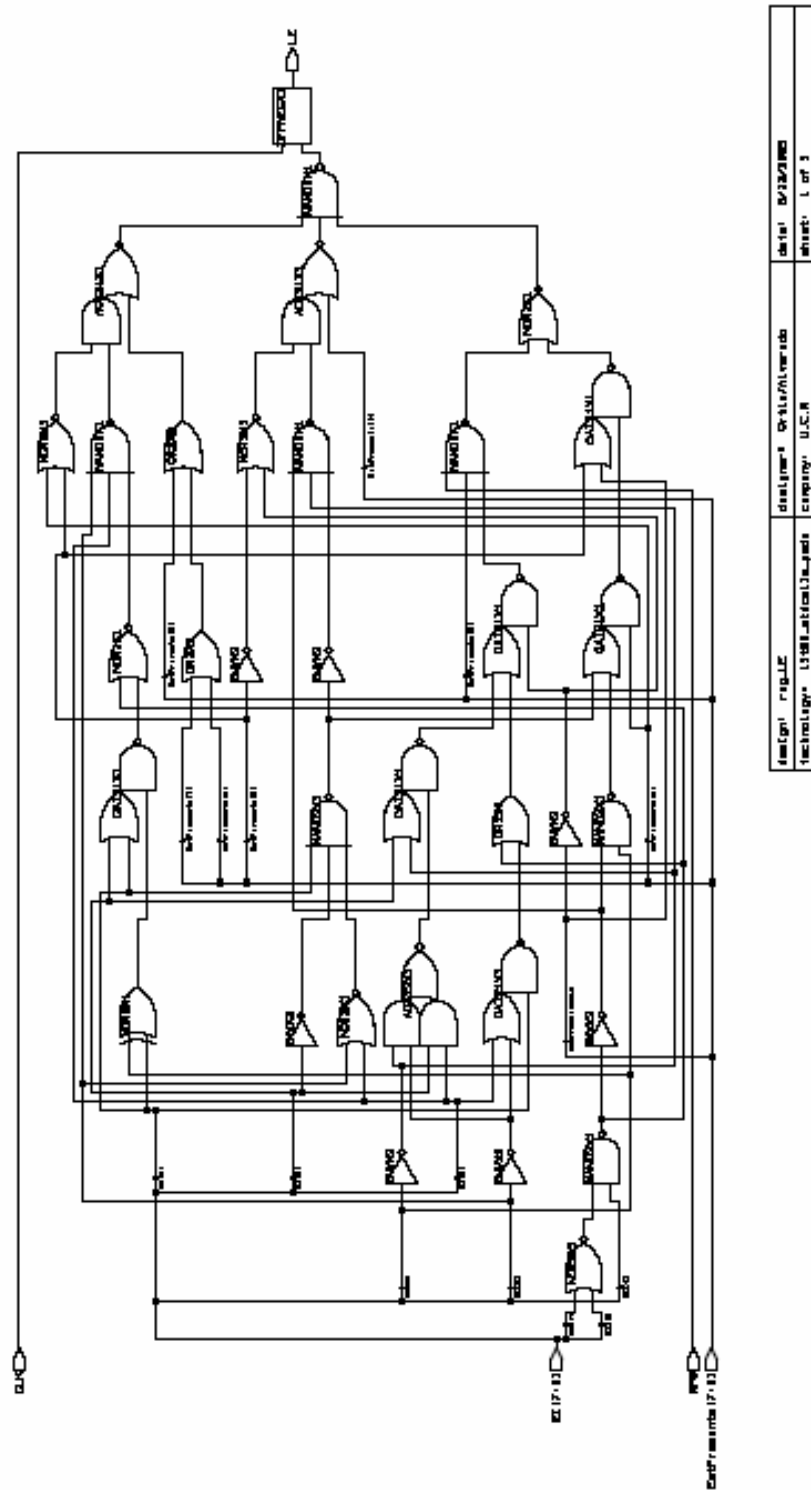


Figura A.5.5 Circuito Sintetizado del módulo de lectura/escritura

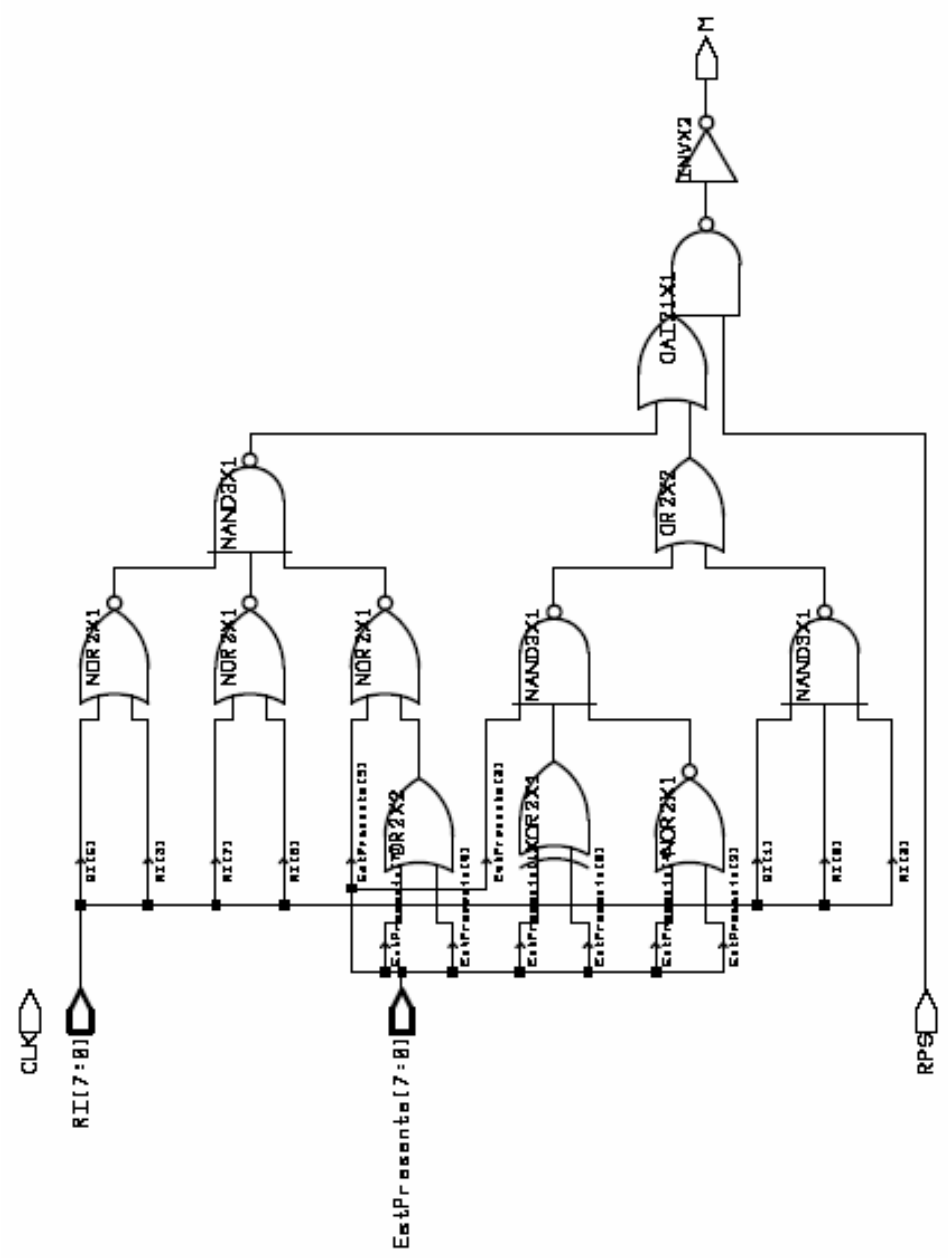


Figura A.5.6 Circuito Sintetizado del módulo de la línea M

design: reg_M	designer: Ortiz/Alvarado	date: 6/22/2005
technology: iit06_stdcells_pads	company: U.C.R	sheet: 1 of 1

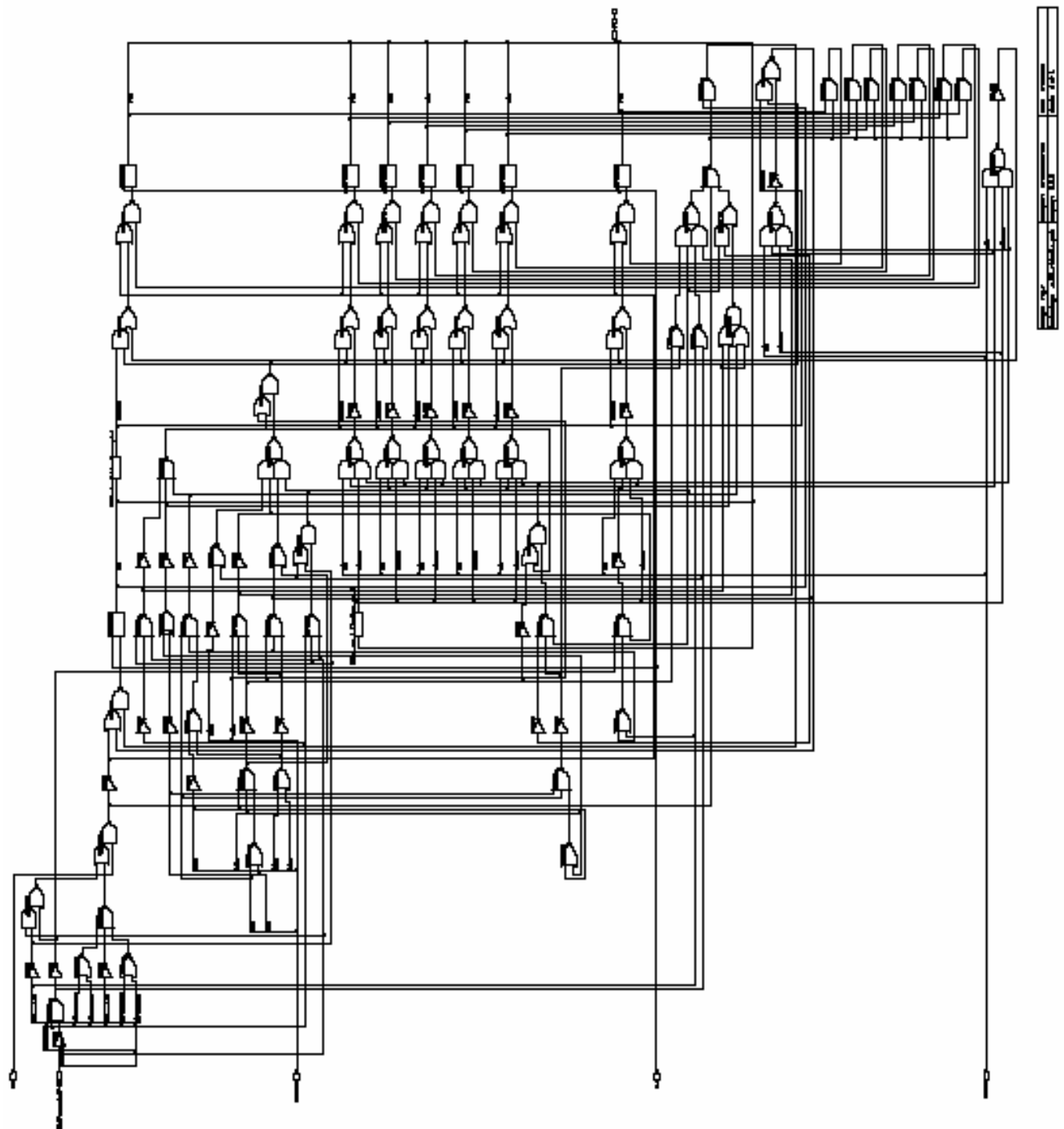


Figura A.5.7 Circuito Sintetizado del puntero de pila

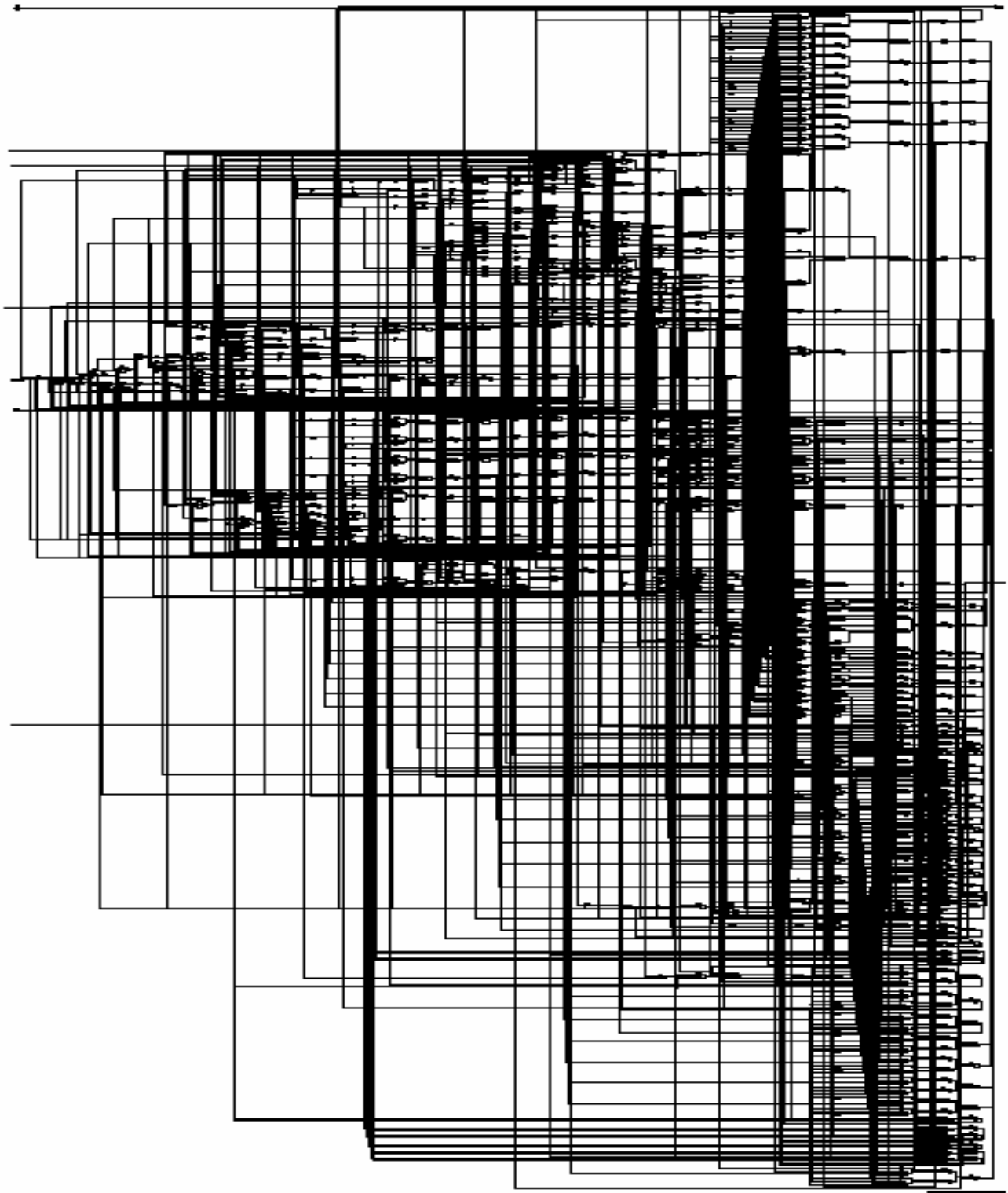


Figura A.5.8 Circuito Sintetizado del contador de programa

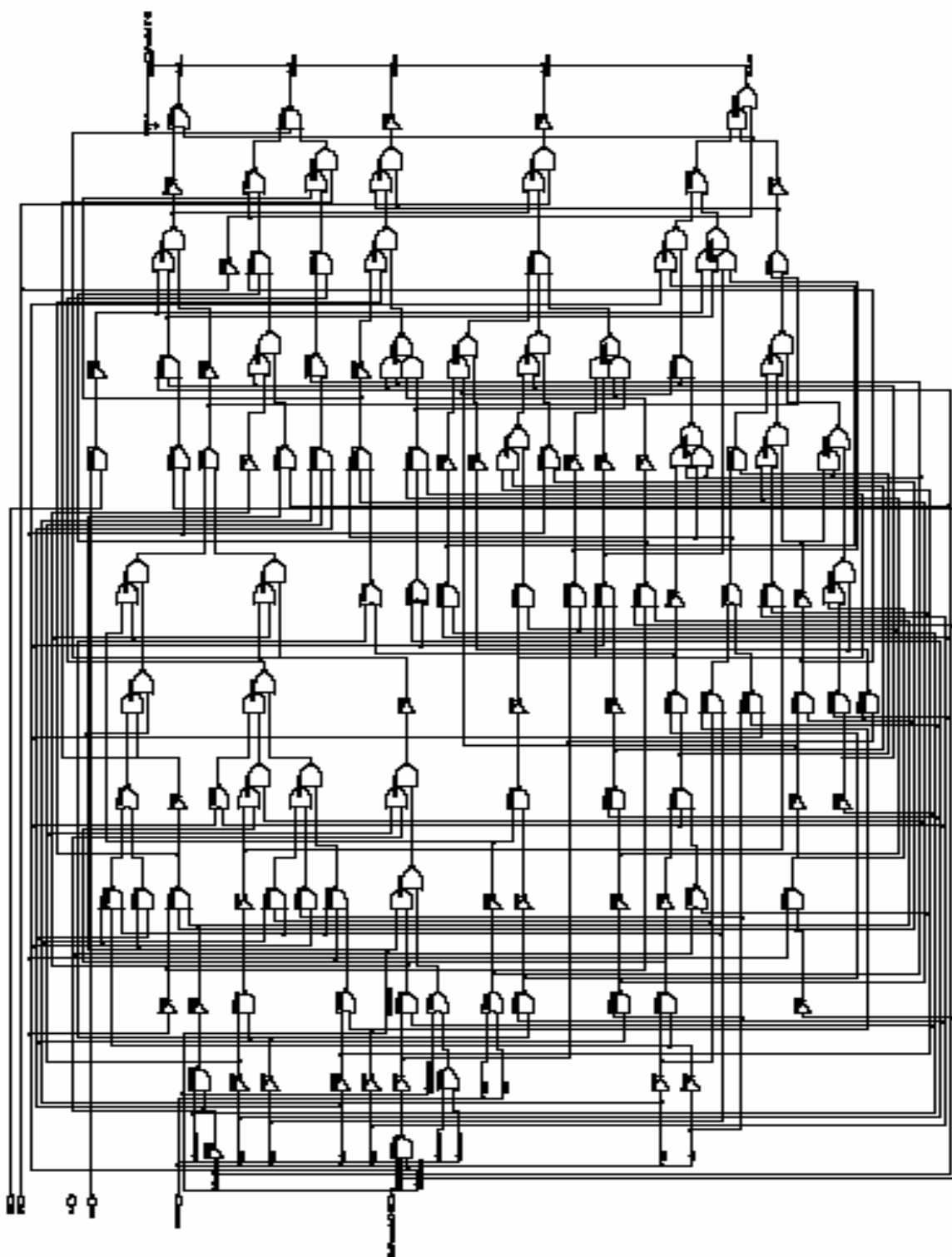
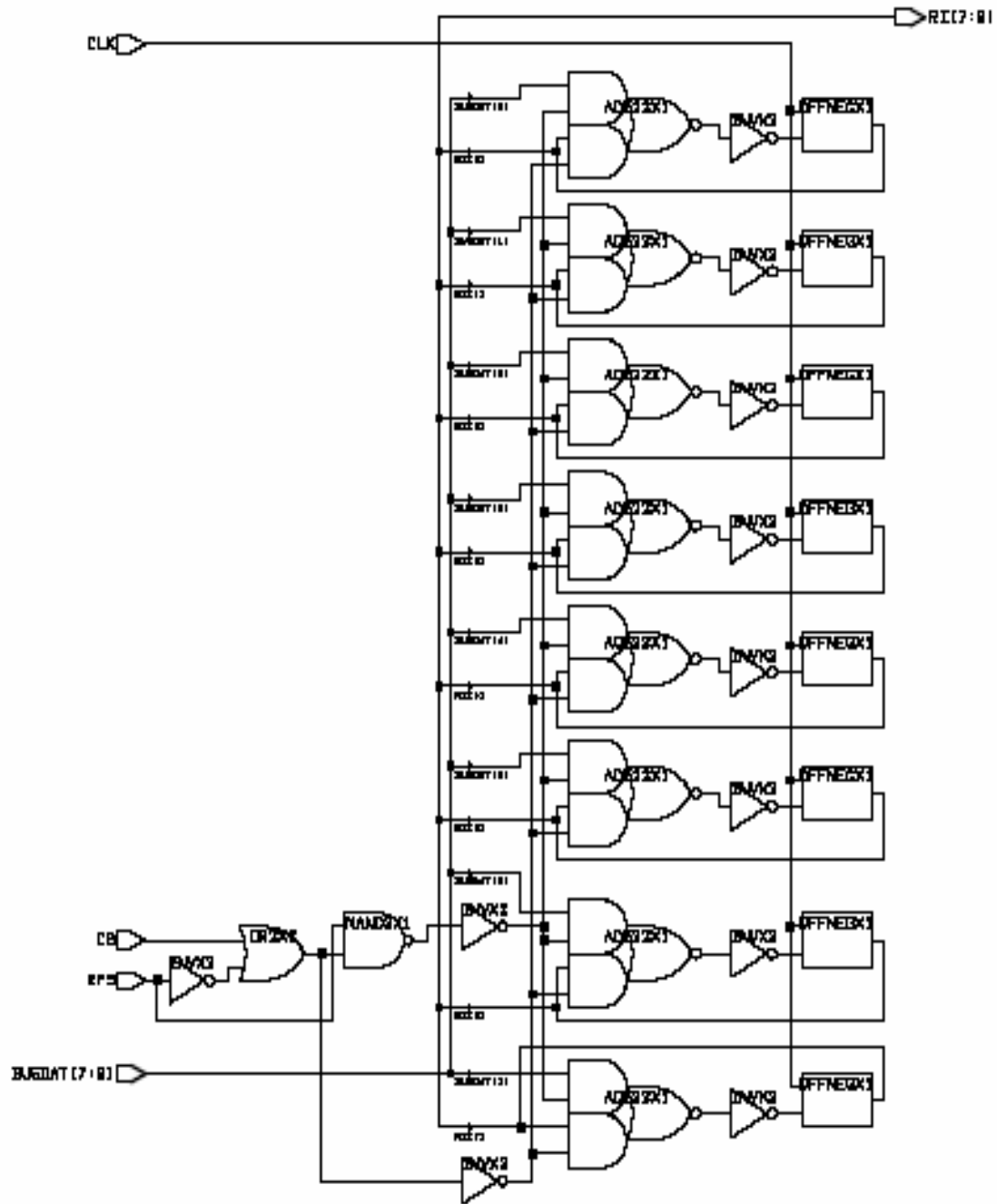


Figura A.5.9 Circuito Sintetizado del módulo de próximo estado



design: reg_RI	designer: Ortiz/Alvarado	date: 6/22/2003
technology: itt06_midcella_pads	company: U.C.R	sheet: 1 of 1

Figura A.5.10 Circuito Sintetizado del módulo de registro de instrucción

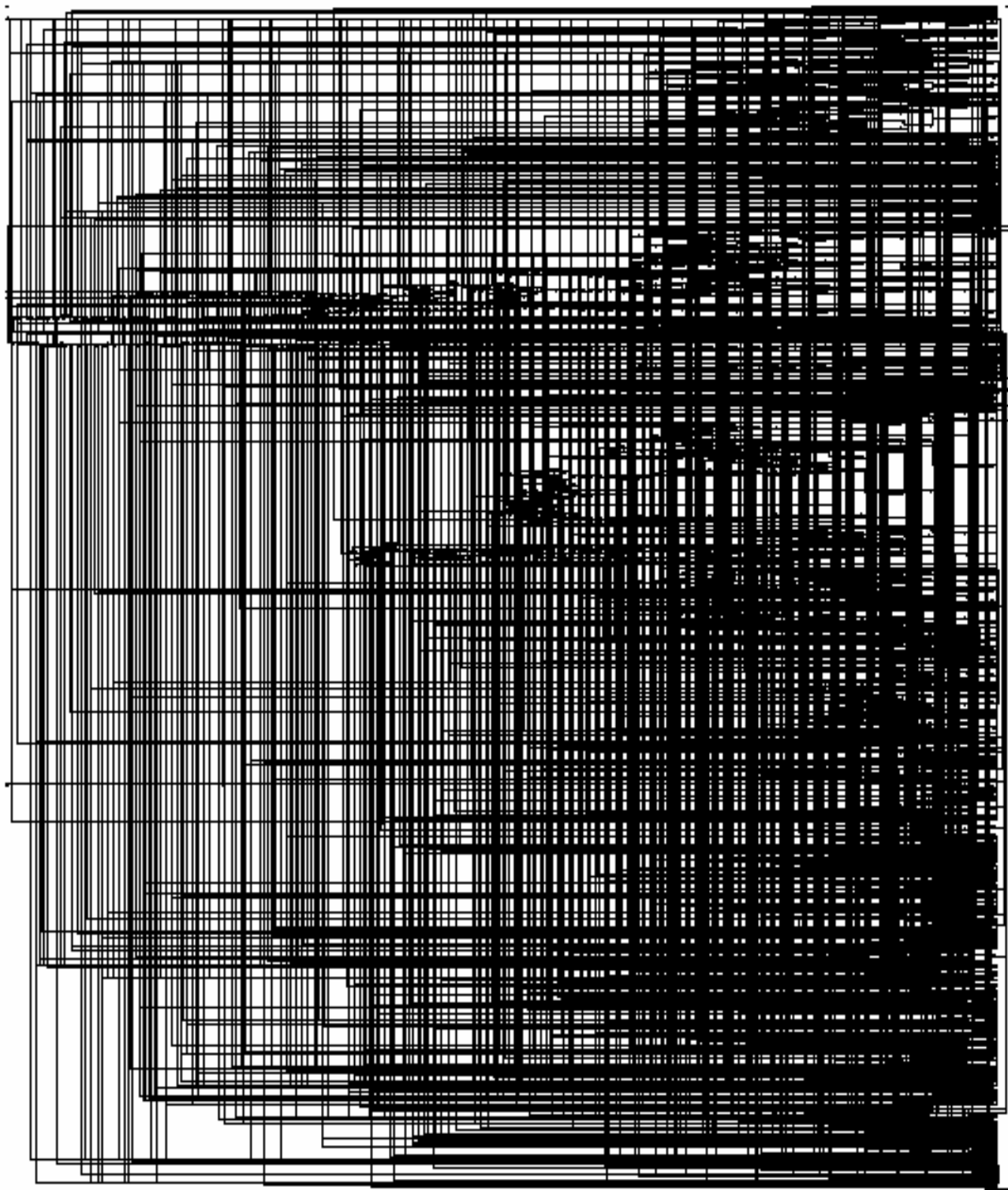


Figura A.5.11 Circuito Sintetizado de la CPUCR completa

A.6 Código del modelo conductual de la CPUCR en Verilog®.

```

`timescale 1ns/10ps
`define MSB8 7 //Buses de 8bits
`define MSB16 15 //Buses de 16bits

module CPUCR(CLK,RPS,LE,M,CB,CM,HALT,INT,INTOK,SDMA,BD,BUSDIR,BUSDAT);
    input CLK,INT,SDMA,RPS; //Reloj;Interrupcion;Acceso directo a
    memoria;Reposicion;Espere
    output BD,CB,CM,LE,HALT,INTOK,M; //Bus de datos;Ciclo de busqueda;Ciclo de
    memoria;Lectura/escritura;Halt;INTOK;memoria

    //*****BUSES*****
    output [`MSB16:0] BUSDIR;
    inout [`MSB8:0] BUSDAT;

    wire CLK,RPS,HALT;
    wire [`MSB8:0] BUSDAT,A,P,S;
    wire [`MSB8:0] RI,PB;
    wire [`MSB8:0] EstPresente, ProxEst;
    wire [`MSB16:0] PC,BUSDIR;
    `include "estados.v"
    `include "Dec_Inst.v"

    reg_RI RI1(CLK,CB,BUSDAT,RI,RPS);
    reg_Est_Presente EP1(CLK,EstPresente,ProxEst,RPS);
    reg_CB CB1(CLK,EstPresente,RPS,CB,CM,BD,ProxEst,INTOK,SDMA);
    reg_ProxEst PE1(RI,EstPresente,ProxEst,RPS,SDMA,INT);
    acumulador A1(CLK,RI,EstPresente,A,BUSDAT,S,P);
    reg_PC PC1(CLK,RI,RPS,EstPresente,PC,BUSDAT,S,BUSDIR,P,PB,SDMA,INT);
    reg_M M1(CLK,ProxEst,RI,M,RPS,SDMA);
    reg_LE LE1(CLK,EstPresente,RI,LE,RPS,SDMA);
    reg_P PP(CLK,EstPresente,RI,RPS,P,A);
    reg_HLT HL1(EstPresente,RPS,CLK,HALT);
    assign BUSDAT=(!LE&&INT&&((RI==STaInd)|| (RI==PHA)|| (RI==STA))) ? A :8'bz;
    assign BUSDAT=(!LE&&(RI==PHS)) ? S :8'bz;
    assign BUSDAT=(!LE&&((RI==JSR)|| (RI==JSRind))) ? PB:8'bz;
    assign BUSDAT=(!LE&&(RI==OUT)) ? A:8'bz;
    assign BUSDAT=(!LE&&!INT) ? PB:8'bz;

endmodule

//*****Registro de instruccion*****

module reg_RI(CLK,CB,BUSDAT,RI,RPS);
    input CLK,CB,RPS;
    input [`MSB8:0] BUSDAT;
    output [`MSB8:0] RI;
    reg [`MSB8:0] RI;
    `include "estados.v"

    always @(negedge CLK)
        begin
            if (RPS)
                begin
                    if (CB) RI<=BUSDAT;
                    else RI <= RI;
                end
            else RI<=8'b0;
        end
endmodule

```



```
//*****Registro Estado Presente*****
module reg_Est_Presente(CLK,EstPresente,ProxEst,RPS);
    input CLK,RPS;
    input [`MSB8:0] ProxEst;
    output [`MSB8:0] EstPresente;
    reg [`MSB8:0] EstPresente;
    always @(negedge CLK)
        begin
            if(RPS) EstPresente<=ProxEst;
            else EstPresente<=8'b0;
        end
endmodule

//*****Registro Ciclo de Búsqueda*****

module reg_CB(CLK, EstPresente,RPS,CB,CM,BD,ProxEst,INTOK,SDMA);
    input CLK,RPS,SDMA;
    input [`MSB8:0] EstPresente,ProxEst;
    output CB,CM,BD,INTOK;
    reg CB,CM,BD,INTOK;
    `include "estados.v"
    always @(negedge CLK)
        begin
            if(RPS)
                begin
                    if(ProxEst==Estado_1) CB<=1'b1;
                    else CB<=1'b0;
                end
            else CB<=1'b0;
        end
    always @(negedge CLK) begin
        if(!SDMA&&((ProxEst==Estado_15)|| (ProxEst==Estado_16))) CM<=1'bz;
        else if (!RPS) CM<=1'b0;
        else if ((ProxEst==Estado_1)&&(EstPresente==Estado_16)) CM<=1'b1;
        else CM<=~(CM);
        if ((ProxEst==Estado_15)|| (ProxEst==Estado_16)) BD<=1'b1;
        else BD<=1'b0;
        if
            ((EstPresente==Estado_21)|| (EstPresente==Estado_22)|| (EstPresente==Estado_23)|| (EstPresente==Estado_24)) INTOK<=1'b1;
        else INTOK<=1'b0;
    end
endmodule

//*****Registro de Proximo Estado *****

module reg_ProxEst(RI,EstPresente,ProxEst,RPS,SDMA,INT);
    input RPS,INT,SDMA;
    input [`MSB8:0] RI,EstPresente;
    output [`MSB8:0] ProxEst;
    reg [`MSB8:0] ProxEst;
    `include "estados.v"
    `include "Dec_Inst.v"
    always @(EstPresente)
        begin
            if(!RPS) ProxEst<=8'b1;
            else
                begin
                    case(EstPresente)
                        Estado_0: ProxEst<=Estado_1;
                    endcase
                end
            end
endmodule
```

```

Estado_1: begin
    if (!INT) ProxEst<=Estado_17;
    else if (!SDMA) ProxEst<=Estado_15;
    else ProxEst<=Estado_2;
    end
Estado_2: begin
    case (RI)

LDAinm, ADDinm, SUBinm, ANDinm, ORAinm, BCC, BCS, BEQ, BNE, BMI, BPL, PHA, PLS, BVC, BVS:
ProxEst<=Estado_3;
        STA, LDA, AND, SUB, ORA, ADD, JMP, PLA, PHS, JSR, JSRind, INP, OUT:
ProxEst<=Estado_3;
        TAP, TPA, CLA, INA, DCA, ROL, ROR, CLC, SEC, SEI, CLI, NOP, CPA:
ProxEst<=Estado_1;
        LDAind, ADDind, SUBind, ANDind, ORAind, STAind, JMPind, RTS, RTI:
ProxEst<=Estado_3;
        HLT: ProxEst<=Estado_30;
        default: ProxEst<=Estado_31;
    endcase
end
Estado_3: begin
    ProxEst<=Estado_4;
end
Estado_4: begin
    case (RI)
        STA, LDA, AND, SUB, ORA, ADD, JMP, JSR, JSRind, RTS, INP, OUT:
ProxEst<=Estado_5;
        LDAind, ADDind, SUBind, ANDind, ORAind, STAind, JMPind, RTI:
ProxEst<=Estado_5;
        default: ProxEst<=Estado_1;
    endcase
end
Estado_5: begin
    ProxEst<=Estado_6;
end
Estado_6: begin
    case (RI)
        JMP, RTS, INP, OUT, RTI: ProxEst<=Estado_1;
        default: ProxEst<=Estado_7;
    endcase
end
Estado_7: ProxEst<=Estado_8;
Estado_8: begin
    case (RI)
        LDAind, ADDind, SUBind, ANDind, ORAind, STAind, JMPind:
ProxEst<=Estado_9;
        JSR, JSRind: ProxEst<=Estado_9;
        default: ProxEst<=Estado_1;
    endcase
end
Estado_9: begin
    ProxEst<=Estado_10;
end
Estado_10: begin
    case (RI)
        JMPind, JSR: ProxEst<=Estado_1;
        default: ProxEst<=Estado_11;
    endcase
end
Estado_11: begin
    ProxEst<=Estado_12;
end
Estado_12: begin

```

```

        case (RI)
            JSRind: ProxEst<=Estado_13;
            default: ProxEst<=Estado_1;
        endcase
    end
Estado_13: begin
    ProxEst<=Estado_14;
end
Estado_14: begin
    ProxEst<=Estado_1;
end
Estado_15: begin //Estados del SDMA
    ProxEst<=Estado_16;
end

Estado_16: begin
    if (!SDMA) ProxEst<=Estado_15;
    else ProxEst<=Estado_1;
end
Estado_17: begin
    ProxEst<=Estado_18;
end
Estado_18: begin
    ProxEst<=Estado_19;
end
Estado_19: begin
    ProxEst<=Estado_20;
end
Estado_20: begin
    ProxEst<=Estado_21;
end
Estado_21: begin
    ProxEst<=Estado_22;
end
Estado_22: begin
    ProxEst<=Estado_23;
end
Estado_23: begin
    ProxEst<=Estado_24;
end
Estado_24: begin
    ProxEst<=Estado_25;
end
Estado_25: begin
    ProxEst<=Estado_1;
end
Estado_30: ProxEst<=Estado_30;
Estado_31: ProxEst<=Estado_31;
    default: ProxEst<=Estado_31;
endcase
end
end
endmodule

//*****Acumulador*****

module acumulador (CLK,RI,EstPresente,A,BUSDAT,S,P);
    input CLK;
    input [`MSB8:0] EstPresente,RI,P;
    input [`MSB8:0] BUSDAT;
    output [`MSB8:0] A;
    output [`MSB8:0] S; //**N*VIZC

```

```

reg [`MSB8:0] A;
reg BC,BN,BZ,BI,BV,Temp,Temp1;
reg [`MSB8:0] S;

`include "estados.v"
`include "Dec_Inst.v"
always @(negedge CLK)
begin
    BZ<= ~|A;
    BN<= A[7];
    case(EstPresente)
        Estado_0: BI<=1'b1;
        Estado_1: begin
            A<=A;
            Temp<=A[7];
            if ((RI == ADD) || (RI == ADDinm) || (RI == ADDind))
                BV <= Temp^Temp1 ? 1'b0:Temp^A[7];
            if (RI == DCA)
                BV <= Temp ? ~A[7]:1'b0;
            if (RI == INA)
                BV <= Temp ? 1'b0:A[7];
            if ((RI == SUB) || (RI == SUBinm) || (RI == SUBind))
                BV <= Temp^Temp1 ? Temp^A[7]:1'b0;
            end
        Estado_2: case(RI)
            CLA: A<=8'b0;
            INA: {BC,A}<=A+1;
            DCA: {BC,A}<=A-1;
            ROL:begin
                BC<=A[7];
                A<={A[6:0],BC};
            end
            ROR:begin
                BC<=A[0];
                A<={BC,A[7:1]};
            end
            CLC: BC<=1'b0;
            SEC: BC<=1'b1;
            SEI: BI<=1'b1;
            CLI: BI<=1'b0;
            TPA: A<=P;
            CPA: A<=~A;
            default:begin
                A<=A;
                BC<=BC;
                BI<=BI;
            end
        endcase
        Estado_3: A<=A;
        Estado_4: begin
            Temp1<=BUSDAT[7];
            case(RI)
                LDainm: A<=BUSDAT;
                ADDinm: {BC,A}<=A+BUSDAT;
                SUBinm: {BC,A}<=A-BUSDAT;
                ANDinm: A<=A&BUSDAT;
                ORainm: A<=A|BUSDAT;
                PLA: A<=BUSDAT;
                PLS: begin
                    A<=A;
                    {BV,BI,BZ,BC}<=BUSDAT[3:0];
                    BN<=BUSDAT[5];
                end
            end
        end
    end
end

```

```

        default: begin
            A<=A;
            BC<=BC;
        end
    endcase
end
Estado_5: A<=A;
Estado_6: case (RI)
    INP: A<=BUSDAT;
    default: A<=A;
endcase
Estado_7: A<=A;
Estado_8: begin
    Temp1<=BUSDAT[7];
    case (RI)
        LDA: A<=BUSDAT;
        AND: A<=A&BUSDAT;
        SUB: {BC,A}<=A-BUSDAT;
        ORA: A<=A|BUSDAT;
        ADD: {BC,A}<=A+BUSDAT;
        default: begin
            A<=A;
            BC<=BC;
        end
    endcase
end
Estado_9: A<=A;
Estado_10: A<=A;
Estado_11: A<=A;
Estado_12: begin
    Temp1<=BUSDAT[7];
    case (RI)
        LDAind: A<=BUSDAT;
        ADDind: A<=A+BUSDAT;
        SUBind: A<=A-BUSDAT;
        ANDind: A<=A&BUSDAT;
        ORAind: A<=A|BUSDAT;
        default: begin
            A<=A;
            BC<=BC;
        end
    endcase
end
default: begin
    A<=A;
    BC<=BC;
    BI<=BI;
    BV<=BV;
end
endcase
S={1'b0,1'b0,BN,1'b0,BV,BI,BZ,BC};
end
endmodule

//***** Contador de programa *****

module reg_PC(CLK,RI,RPS,EstPresente,PC,BUSDAT,S,RDR,P,PB,SDMA,INT);
    input CLK, RPS,SDMA,INT;
    input [`MSB8:0] EstPresente,RI,S,P;
    input [`MSB8:0] BUSDAT;
    output [`MSB16:0] PC, RDR;
    output [`MSB8:0] PB;
    reg [`MSB16:0] PC, RDR; //Registro de direccion

```

```

reg [`MSB8:0] PB; //Parte baja de la direccion

`include "estados.v"
`include "Dec_Inst.v"

always @(negedge CLK)
begin
    if(!RPS) begin
        PC<=16'b0;
        RDR<=16'b0;
        PB<=8'b0;
    end
    else begin
        case(EstPresente)
            Estado_0: begin
                PC<=16'b0;
                RDR<=16'b0;
            end
            Estado_1: begin
                PC<=PC;
                RDR<=(!SDMA&&INT) ? 16'bz:RDR;
            end
            Estado_2: begin
                case(RI)
                    CPA, LDAinm, ADDinm, TAP, TPA, CLA, INA, DCA, SUBinm, ANDinm, ORAinm, INP, OUT:
begin
                                PC<=PC+16'b1;
                                RDR<=RDR+16'b1;
                                end
                    ROL, ROR, BCC, BCS, CLC, SEC, SEI, CLI, NOP, BNE, BEQ, BMI, BPL, STA, LDA, AND, SUB, ORA, ADD, JMP, BVS
                    ,BVC: begin
                                PC<=PC+16'b1;
                                RDR<=RDR+16'b1;
                                end
                    PHA, PHS: begin
                                PC<=PC;
                                RDR<={8'hFF,P};
                                end
                    PLA: begin
                                PC<=PC+16'b1;
                                RDR<={8'hFF,P-8'b1};
                                end
                    PLS:begin
                                PC<=PC;
                                RDR<={8'hFF,P-8'b1};
                                end
                    LDAind, ADDind, SUBind, ANDind, ORAind, STAind, JMPind: begin
                                PC<=PC+16'b1;
                                RDR<=RDR+16'b1;
                                end
                    JSR, JSRind: begin
                                PC<=PC+16'h3;
                                RDR<={8'hFF,P};
                                end
                    RTS, RTI:begin
                                PC<=PC;
                                RDR<={8'hFF,P-8'b1};
                                end

                    default: begin
                                PC<=PC;
                                RDR<=RDR;

```

```

        end
    endcase
    end
Estado_3: begin
    case (RI)

LDAinm, ADDinm, SUBinm, ANDinm, ORainm, STA, LDA, AND, SUB, ORA, ADD, JMP, PLA: begin
        PC<=PC;
        RDR<=RDR;
    end
    BCC, BCS, BPL, BMI, BEQ, BNE, BVC, BVS: begin
        PC<=PC+16'b1;
        RDR<=RDR;
    end
    PHA, PLS, PHS: begin
        PC<=PC+16'b1;
        RDR<=RDR;
    end
    LDAind, ADDind, SUBind, ANDind, ORAind, STAind, JMPind, RTS, INP, OUT, RTI:
begin
        PC<=PC;
        RDR<=RDR;
    end
    JSR, JSRind: begin
        PC<=PC;
        RDR<=RDR;
        PB<=PC[15:8];
    end
    default: begin
        PC<=PC+16'b1;
        RDR<=RDR+16'b1;
        PB<=PB;
    end
    endcase
    end
Estado_4: begin
    case (RI)
    PHA, PHS, PLS: begin
        PC<=PC;
        RDR<=PC;
        PB<=BUSDAT;
    end
    PLA: begin
        PC<=PC;
        RDR<=PC;
    end
    STA, LDA, AND, SUB, ORA, ADD, JMP: begin
        PC<=PC+16'b1;
        RDR<=RDR+16'b1;
        PB<=BUSDAT;
    end
    INP, OUT: begin
        PC<=PC+16'b1;
        RDR<={8'hff, BUSDAT};
    end
    BCC: begin
        case (S[0])
        1'b1: begin
            PC<=PC;
            RDR<=RDR+16'b1;
        end
        1'b0: begin
            PC<=BUSDAT[7] ? PC + {8'hFF, BUSDAT} : PC + {8'b0, BUSDAT};

```

```

        RDR<=BUSDAT[7] ? PC + {8'hFF,BUSDAT} : PC + {8'b0,BUSDAT};
    end
endcase
end
BCS:begin
    case(S[0])
    1'b1:begin
        PC<=BUSDAT[7] ? PC + {8'hFF,BUSDAT} : PC + {8'b0,BUSDAT};
        RDR<=BUSDAT[7] ? PC + {8'hFF,BUSDAT} : PC + {8'b0,BUSDAT};
    end
    1'b0:begin
        PC<=PC;
        RDR<=RDR+16'b1;
    end
    endcase
end
BNE: begin
    case(S[1])
    1'b1: begin
        PC<=PC;
        RDR<=RDR+16'b1;
    end
    1'b0: begin
        PC<=BUSDAT[7] ? PC + {8'hFF,BUSDAT} : PC + {8'b0,BUSDAT};
        RDR<=BUSDAT[7] ? PC + {8'hFF,BUSDAT} : PC + {8'b0,BUSDAT};
    end
    endcase
end
BEQ:begin
    case(S[1])
    1'b1: begin
        PC<=BUSDAT[7] ? PC + {8'hFF,BUSDAT} : PC + {8'b0,BUSDAT};
        RDR<=BUSDAT[7] ? PC + {8'hFF,BUSDAT} : PC + {8'b0,BUSDAT};
    end
    1'b0: begin
        PC<=PC;
        RDR<=RDR+16'b1;
    end
    endcase
end
BMI:begin
    case(S[5])
    1'b1: begin
        PC<=BUSDAT[7] ? PC + {8'hFF,BUSDAT} : PC + {8'h00,BUSDAT};
        RDR<=BUSDAT[7] ? PC + {8'hFF,BUSDAT} : PC + {8'h00,BUSDAT};
    end
    1'b0:begin
        PC<=PC;
        RDR<=RDR+16'b1;
    end
    endcase
end
BPL: begin
    case(S[5])
    1'b1: begin
        PC<=PC;
        RDR<=RDR+16'b1;
    end
    1'b0: begin
        PC<=BUSDAT[7] ? PC + {8'hFF,BUSDAT} : PC + {8'b0,BUSDAT};
        RDR<=BUSDAT[7] ? PC + {8'hFF,BUSDAT} : PC + {8'b0,BUSDAT};
    end
    endcase
end

```



```

end

BVC: begin
  case(S[3])
    1'b1: begin
      PC<=PC;
      RDR<=RDR+16'b1;
    end
    1'b0: begin
      PC<=BUSDAT[7] ? PC + {8'hFF,BUSDAT} : PC + {8'b0,BUSDAT};
      RDR<=BUSDAT[7] ? PC + {8'hFF,BUSDAT} : PC + {8'b0,BUSDAT};
    end
  endcase
end
BVS:begin
  case(S[3])
    1'b1:begin
      PC<=BUSDAT[7] ? PC + {8'hFF,BUSDAT} : PC + {8'b0,BUSDAT};
      RDR<=BUSDAT[7] ? PC + {8'hFF,BUSDAT} : PC + {8'b0,BUSDAT};
    end
    1'b0:begin
      PC<=PC;
      RDR<=RDR+16'b1;
    end
  endcase
end

LDAind,ADDind,SUBind,ANDind,ORAind,STAind,JMPind: begin
  PC<=PC+16'b1;
  RDR<=RDR+16'b1;
  PB<=BUSDAT;
end

JSR,JSRind: begin
  PC<=PC;
  RDR<={8'hFF,P};
  PB<=PB;
end

RTS:begin
  PC<=PC;
  RDR<={8'hFF,P};
  PB<=BUSDAT;
end

RTI:begin
  PC<=PC;
  RDR<={8'hFF,P-8'b1};
  PB<=BUSDAT;
end

default: begin
  PC<=PC+16'b1;
  RDR<=RDR+16'b1;
  PB<=PB;
end
endcase
end

Estado_5: begin
  case(RI)
    STA,LDA,AND,SUB,ORA,ADD,INP,OUT: begin
      PC<=PC;
      RDR<=RDR;
    end
    LDAind,ADDind,SUBind,ANDind,ORAind,STAind,JMPind: begin
      PC<=PC;
      RDR<=RDR;
    end
  end
end

```

```

        JSR, JSRind: begin
            PC<=PC;
            RDR<=RDR;
            PB<=PC[7:0];
        end
        default: begin
            PC<=PC;
            RDR<=RDR;
            PB<=PB;
        end
    endcase
end

Estado_6: begin
    case (RI)
        STA, LDA, AND, SUB, ORA, ADD: begin
            PC<=PC+16'b1;
            RDR<={BUSDAT, PB};
        end
        JMP: begin
            PC<={BUSDAT, PB};
            RDR<={BUSDAT, PB};
        end
        LDAind, ADDind, SUBind, ANDind, ORAind, STAind, JMPind: begin
            PC<=PC+16'b1;
            RDR<={BUSDAT, PB};
        end
        JSR, JSRind: begin
            PC<=PC-16'h2;
            RDR<=PC-16'h2;
        end
        RTS, RTI: begin
            PC<={BUSDAT, PB};
            RDR<={BUSDAT, PB};
        end
        INP, OUT: begin
            PC<=PC;
            RDR<=PC;
        end
        default: begin
            PC<=PC;
            RDR<=RDR;
        end
    endcase
end

Estado_7: begin
    case (RI)
        STA, LDA, AND, SUB, ORA, ADD: begin
            PC<=PC;
            RDR<=RDR;
        end
        default: begin
            PC<=PC;
            RDR<=RDR;
        end
    endcase
end

Estado_8: begin
    case (RI)
        STA, LDA, AND, SUB, ORA, ADD: begin
            PC<=PC;
            RDR<=PC;
        end
    endcase
end

```

```

        end
        LDAind, SUBind, ADDind, ORAind, STAind, JMPind: begin
            PC<=PC;
            RDR<=RDR+16'b1;
            PB<=BUSDAT;
        end
        JSR, JSRind: begin
            PC<=PC+16'b1;
            RDR<=RDR+16'b1;
            PB<=BUSDAT;
        end
        default: begin
            PC<=PC;
            RDR<=RDR;
        end
    endcase
end
Estado_9: begin
    PC<=PC;
    RDR<=RDR;
end
Estado_10: begin
    case (RI)
        JMPind, JSR, JSRind: begin
            PC<={BUSDAT, PB};
            RDR<={BUSDAT, PB};
        end
        default: begin
            PC<=PC;
            RDR<={BUSDAT, PB};
        end
    endcase
end
Estado_11: begin
    PC<=PC;
    RDR<=RDR;
end
Estado_12: begin
    case (RI)
        JSRind: begin
            PB<=BUSDAT;
            PC<=PC;
            RDR<=RDR+16'b1;
        end
        default: begin
            PC<=PC;
            RDR<=PC;
            PB<=PB;
        end
    endcase
end
Estado_13: begin
    PC<=PC;
    RDR<=RDR;
end
Estado_14: begin
    PC<={BUSDAT, PB};
    RDR<={BUSDAT, PB};
    PB<=PB;
end
Estado_15: begin
    PC<=PC;
    RDR<=!SDMA ? 16'bz:PC;

```

```

        PB<=PB;
    end
Estado_16: begin
    PC<=PC;
    RDR<= !SDMA ? 16'bz:PC;
    PB<=PB;
    end
Estado_17: begin
    PC<=PC;
    RDR<={8'hFF,P};
    PB<=PC[15:8];
    end
Estado_18: begin
    PC<=PC;

    PB<=PB;
    RDR<=RDR;
    end
Estado_19: begin
    PC<=PC;
    RDR<={8'hFF,P};
    PB<=PC[7:0];
    end
Estado_20: begin
    PC<=PC;
    RDR<=RDR;
    PB<=PB;
    end
    Estado_21: begin
        PC<=PC;
        RDR<=16'hFF00;
        PB<=PB;
    end
Estado_22: begin
    PC<=PC;
    RDR<=RDR;
    PB<=BUSDAT;
    end
    Estado_23: begin
        PC<=PC;
        RDR<=16'hFF01;
        PB<=PB;
    end
Estado_24: begin
    PC<=PC;
    RDR<=RDR;
    PB<=PB;
    end
Estado_25: begin
    PC<={BUSDAT,PB};
    RDR<={BUSDAT,PB};
    PB<=PB;
    end

default: begin
    PC<=PC;
    RDR<=RDR;
    PB<=PB;
    end
endcase
end
end
endmodule

```

```

//***** Línea M *****
module reg_M(CLK, ProxEst, RI, M, RPS, SDMA);
    input CLK, RPS, SDMA;
    input [`MSB8:0] ProxEst, RI;
    output M;
    reg M;
    `include "estados.v"
    `include "Dec_Inst.v"

    always @(negedge CLK) begin
        if(!RPS) M<=1'b0;
        else begin
            case (ProxEst)
                Estado_0: M<=1'b1;
                Estado_1: M<=1'b1;
                Estado_5: case (RI)
                    INP, OUT: M<=1'b0;
                    default: M<=1'b1;
                endcase
                Estado_6: case (RI)
                    INP, OUT: M<=1'b0;
                    default: M<=1'b1;
                endcase
                Estado_15: M <=1'bz;
                Estado_16: M<= !SDMA ? 1'bz : 1'b1;
                default: M<=1'b1;
            endcase
        end
    end
endmodule

//***** Línea Lectura/Escritura *****
module reg_LE(CLK, EstPresente, RI, LE, RPS, SDMA);
    input CLK, RPS, SDMA;
    input [`MSB8:0] EstPresente, RI;
    output LE;
    reg LE;
    `include "estados.v"
    `include "Dec_Inst.v"
    always @(negedge CLK) begin
        if(!RPS) LE<=1'b1;
        else begin
            case (EstPresente)
                Estado_0: LE<=1'b1;
                Estado_1: LE<=1'b1;
                Estado_3: case (RI)
                    PHA, PHS: LE<=1'b0;
                    JSR: LE<=1'b0;
                    default: LE<=1'b1;
                endcase
                Estado_5: case (RI)
                    JSR, OUT: LE<=1'b0;
                    default: LE<=1'b1;
                endcase
                Estado_7: case (RI)
                    STA: LE<=1'b0;
                    default: LE<=1'b1;
                endcase
                Estado_11: case (RI)
                    STAind: LE<=1'b0;
                    default: LE<=1'b1;
            endcase
        end
    end
endmodule

```

```

        endcase
        Estado_15: LE<=1'bz;
        Estado_16: LE<=!SDMA ? 1'bz:1'b1;
        Estado_18: LE<=1'b0;
        Estado_20: LE<=1'b0;
        default: LE<=1'b1;
    endcase
end
end
endmodule

//***** Puntero de Pila *****
module reg_P(CLK,EstPresente,RI,RPS,P,A);
    input RPS,CLK;
    input [`MSB8:0] EstPresente,RI,A;
    output [`MSB8:0] P;
    reg [`MSB8:0] P;
    `include "estados.v"
    `include "Dec_Inst.v"
    always@(negedge CLK)begin
        if(!RPS) P<=8'bx;
        else
            begin
                case(EstPresente)
                    Estado_1: begin
                        case(RI)
                            TAP: P<=P;
                            default: P<=P;
                        endcase
                    end
                    Estado_2: begin
                        case(RI)
                            TAP: P<=A;
                            RTS,RTI:P<=P-8'b1;
                            default:P<=P;
                        endcase
                    end
                    Estado_3:begin
                        case(RI)
                            JSR: P<=P+8'b1;
                            RTS: P<=P-8'b1;
                            default:P<=P;
                        endcase
                    end
                    Estado_4: begin
                        case(RI)
                            PHA,PHS: P<=P+8'b1;
                            PLS,PLA,RTI: P<=P-8'b1;
                            default: P<=P;
                        endcase
                    end
                    Estado_6: begin
                        case(RI)
                            JSR: P<=P+8'b1;
                            RTS: P<=P-8'b1;
                            default: P<=P;
                        endcase
                    end
                    Estado_18: begin
                        P<=P+8'b1;
                    end
                    Estado_21: begin
                        P<=P+8'b1;

```

```

        end
        default: P<=P;
    endcase
end
end
endmodule

//***** HLT *****
module reg_HLT(EstPresente,RPS,CLK,HALT);
    input RPS,CLK;
    input [`MSB8:0] EstPresente;
    output HALT;
    reg HALT;
    `include "estados.v"
    `include "Dec_Inst.v"
    always@(negedge CLK) begin
        if(!RPS) HALT<=1'b0;
        else begin
            case(EstPresente)
                Estado_30:HALT<=1'b1;
                Estado_31:HALT<=~HALT;
                default: HALT<=1'b0;
            endcase
        end
    end
endmodule

```