Tarea #3 CPU

Danny Gutiérrez Campos
C33566
Sistemas Digitales II

13/10/2025

1. Resumen

Esta tarea implementa un controlador de CPU basado en una máquina de estados finita (FSM) que ejecuta un conjunto de 7 instrucciones (LOAD, STORE, ADD, SUB, OR, AND, EQUAL) operando sobre dos registros de 8 bits (A y B) y accediendo a memorias ROM y RAM. El controlador fue diseñado en Verilog con una arquitectura de 7 estados, el cual fue sintetizado con Yosys usando la biblioteca cmos_cells.lib, y verificado mediante un plan de pruebas que incluye la ejecución de un programa de ejemplo con datos inicializados usando los últimos 4 dígitos del carné (C33566: 3, 5, 6, 6). Las pruebas demostraron el correcto funcionamiento tanto del diseño original como del netlist sintetizado, verificando todas las operaciones aritméticas, lógicas, de memoria y la condición de terminación mediante la instrucción EQUAL, obteniendo los resultados esperados y confirmando que el programa se detiene correctamente cuando los registros son iguales.

2. Descripción Arquitectónica

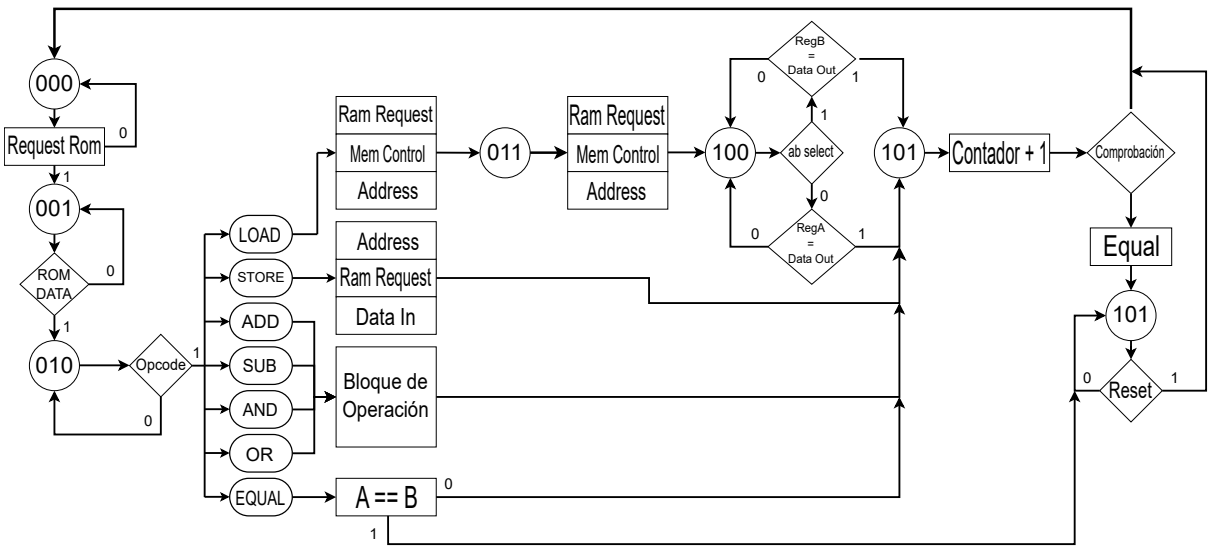

Figura 1: Diagrama de bloques del sistema

El base a la figura 1 el proceso comienza en el estado **BUSCAR (000)**, donde el controlador activa la señal `Request_Rom` para solicitar una instrucción desde la memoria ROM utilizando el contador de programa como dirección. La ROM devuelve una instrucción (`ROM_DATA`) que contiene el OPCODE, el selector A/B y la dirección del operando. En el estado **DECODIFICAR (001)**, el controlador extrae los campos de la instrucción: OPCODE[3:0] para identificar la operación, A/B SELECT para determinar qué registro (A o B) se utilizará, y OP_ADDR[7:0] para las operaciones de memoria.

El estado **EJECUTAR (010)** bifurca el flujo según el tipo de instrucción. Para instrucciones de memoria (LOAD y STORE), el controlador activa `Ram_Request`, configura `Mem_Control` (1 para lectura, 0 para escritura) y establece la dirección en `Address`. La instrucción LOAD requiere dos estados adicionales: **ESPERAR_RAM (011)** mantiene las señales activas mientras la RAM procesa la solicitud, y **ESCRIBIR (100)** captura el dato de `Data_Out` y lo almacena en el registro A o B según `ab_select`. La instrucción STORE escribe directamente el contenido del registro seleccionado en RAM mediante `Data_In`. Para operaciones aritméticas y lógicas (ADD, SUB, OR, AND), el **Bloque de Operación** ejecuta la operación correspondiente entre los registros A y B, almacenando el resultado en el registro B. La instrucción EQUAL compara ambos registros y actualiza la bandera `Equal`.

El estado **INCREMENTAR_CONTADOR (101)** realiza dos funciones críticas: incrementa el contador de programa para apuntar a la siguiente instrucción y verifica la condición de terminación. Si la última instrucción fue EQUAL y la bandera `Equal` está activa (regA == regB), el sistema transiciona al estado **DETENER (101)**, donde permanece indefinidamente con todas las señales de control desactivadas. Si la condición no se cumple, el ciclo regresa a BUSCAR para ejecutar la siguiente instrucción. La señal `Reset` permite reiniciar el sistema en cualquier momento, retornando al estado BUSCAR con el contador de programa en 0 y todos los registros inicializados.

3. Plan de Pruebas

A continuación se detallan las pruebas realizadas al diseño del controlador de CPU. El plan de pruebas se implementa mediante dos módulos principales: el tester que genera las señales de reloj y reset, la ROM que contiene el programa de prueba diseñado para validar todas las instrucciones del conjunto de instrucciones, y la RAM que posee la memoria a acceder.

3.1. Módulo Probador (tester.v)

El módulo probador es responsable de generar las señales fundamentales para la operación del sistema: el reloj (clk) y el reset (rst). El reloj se genera con un período de 10 unidades temporales mediante una inversión cada 5 unidades, proporcionando un flanco activo creciente y estable para la sincronización del controlador de CPU. La secuencia de prueba inicia con la activación de la señal de reset durante 20 unidades temporales, garantizando que todos los registros internos del CPU, el contador de programa y las señales de control se inicialicen correctamente en sus valores por defecto. Específicamente, el reset asegura que el contador de programa retorne a la posición cero, permitiendo que la ejecución del programa comience desde la primera instrucción almacenada en ROM.

Después de desactivar el reset, el probador permite que el sistema ejecute libremente durante 900 unidades temporales. Este tiempo fue calculado para permitir la ejecución completa del programa de prueba almacenado en ROM, que consiste en 17 instrucciones, considerando que cada instrucción requiere múltiples ciclos de reloj para completarse (especialmente las operaciones LOAD que necesitan estados adicionales para esperar la respuesta de la RAM).

3.2. Programa de Prueba (rom.v)

El módulo ROM implementa la memoria de instrucciones del sistema y contiene un programa de prueba diseñado específicamente para validar todas las instrucciones del conjunto de instrucciones. El programa consta de 17 instrucciones organizadas en seis bloques de prueba, cada uno enfocado en verificar aspectos específicos del funcionamiento del controlador.

3.2.1. Bloque 1: Validación de LOAD y ADD (Direcciones 0-3)

Las primeras cuatro instrucciones validan las operaciones de carga desde memoria y suma aritmética. La instrucción en la dirección 0 carga el valor 3 desde la posición 0 de RAM en el registro A, utilizando el formato de instrucción con OPCODE 0011, selector A/B en 0 (para registro A) y dirección de operando 00000000. La instrucción en la dirección 1 carga el valor 5 desde la posición 1 de RAM en el registro B, con selector A/B en 1. Estas dos instrucciones LOAD verifican que el controlador puede acceder correctamente a diferentes posiciones de memoria y almacenar los datos en registros distintos según el campo A/B SELECT.

La instrucción en la dirección 2 ejecuta una operación ADD con OPCODE 1001, que debe sumar los contenidos de los registros A y B (3 + 5) y almacenar el resultado (8) en el registro B. Esta instrucción valida que las operaciones aritméticas se ejecutan correctamente sin acceder a memoria. Finalmente, la instrucción en la dirección 3 ejecuta STORE con OPCODE 0100 para guardar el resultado (8) en la posición 15 de RAM, verificando que el controlador puede escribir correctamente en memoria y que el resultado de operaciones aritméticas puede ser persistido.

3.2.2. Bloque 2: Validación de SUB (Direcciones 4-7)

Este bloque verifica la operación de resta mediante cuatro instrucciones. Las instrucciones en las direcciones 4 y 5 cargan los valores 6 y 6 desde las posiciones 2 y 3 de RAM en los registros A y B respectivamente. La instrucción en la dirección 6 ejecuta SUB con OPCODE 1010, que debe restar B de A (6 - 6) produciendo resultado 0. Este caso particular es importante porque valida el manejo correcto de resultados nulos. La instrucción en la dirección 7 almacena el resultado en la posición 16 de RAM mediante STORE, confirmando que valores cero pueden ser escritos y almacenados correctamente en memoria.

3.2.3. Bloque 3: Validación de OR (Direcciones 8-11)

Las instrucciones 8 y 9 recargan los valores 3 y 5 en los registros A y B respectivamente, reutilizando las posiciones 0 y 1 de RAM. Esto también valida que el sistema puede leer múltiples veces desde las mismas posiciones de memoria sin degradación de datos. La instrucción en la dirección 10 ejecuta OR con OPCODE 0001, realizando una operación OR bit a bit entre los registros. El resultado esperado es 7 (0b00000011 OR 0b00000101 = 0b00000111), validando que las operaciones lógicas se ejecutan correctamente a nivel de bits individuales. La instrucción 11 almacena este resultado en la posición 17 de RAM.

3.2.4. Bloque 4: Validación de AND (Direcciones 12-13)

La instrucción en la dirección 12 ejecuta AND con OPCODE 0010 sobre los valores actuales de los registros. Esta prueba es crucial porque valida que las operaciones pueden encadenarse, utilizando el resultado de una operación previa como operando de la siguiente. El resultado esperado es 1 (0b00000011 AND 0b00000111 = 0b00000001). La instrucción 13 almacena el resultado en la posición 18 de RAM, completando la validación de todas las operaciones lógicas del conjunto de instrucciones.

3.2.5. Bloque 5: Validación de EQUAL y Terminación (Direcciones 14-16)

Este bloque final valida la instrucción de comparación y la condición de terminación del programa. Las instrucciones 14 y 15 cargan ambos registros con el valor 7 desde las posición 17 de RAM. La instrucción 16 ejecuta EQUAL con OPCODE 0101, que debe comparar el contenido de ambos registros. Como ambos contienen el mismo valor, la bandera equal debe activarse y el controlador debe transicionar al estado DETENER. Este es el caso crítico del programa porque valida que el sistema puede detectar correctamente la condición de terminación y detenerse sin ejecutar instrucciones adicionales, permaneciendo indefinidamente en el estado DETENER.

3.3. Modulo RAM (ram.v)

El módulo `ram.v` implementa la memoria de datos del sistema y tiene como función principal almacenar y proveer información al controlador de CPU durante la ejecución del programa. Esta memoria es de tipo síncrona, con lectura y escritura controladas por flanco positivo de reloj, e incluye un mecanismo de inicialización basado en los últimos cuatro dígitos del carné. El diseño cuenta con una memoria de 256 posiciones de 8 bits cada una, direccionadas mediante un bus de 12 bits para compatibilidad con la ROM y el CPU. Al activarse la señal de `reset`, la memoria se limpia completamente y se cargan los cuatro primeros valores de la RAM con los dígitos del carné. En el caso del carné `C33566`, las posiciones `RAM[0]` a `RAM[3]` contienen los valores 3, 5, 6 y 6 respectivamente.

- `clk`: Señal de reloj que sincroniza las operaciones de lectura y escritura.
- `rst`: Señal de reinicio que limpia e inicializa la memoria.
- `request`: Indica que la CPU solicita acceso a memoria.
- `mem_control`: Controla el tipo de operación, donde 0 corresponde a escritura y 1 a lectura.
- `address`: Dirección de 12 bits que selecciona la posición de memoria a acceder.
- `data_in` y `data_out`: Buses de 8 bits para transferencia de datos entre la CPU y la memoria.

El bloque principal del módulo opera según la siguiente lógica:

- Durante el **reset**, la memoria se limpia y se cargan los valores iniciales del carné.
- Cuando `request` está activo y `mem_control` = 1, se realiza una lectura desde la dirección especificada.
- Cuando `request` está activo y `mem_control` = 0, se escribe en la dirección seleccionada.

Esta estructura permite validar completamente las instrucciones del conjunto definido en el plan de pruebas:

1. Las instrucciones `LOAD` leen desde las posiciones `RAM[0]`–`RAM[3]` los valores del carné.
2. Las instrucciones `ADD`, `SUB`, `OR` y `AND` procesan estos valores en los registros internos de la CPU.
3. Las instrucciones `STORE` escriben los resultados en posiciones superiores de la memoria (`RAM[15]`–`RAM[18]`).
4. Finalmente, la instrucción `EQUAL` compara los resultados almacenados y activa la señal de terminación.

4. Instrucciones de Utilización de la Simulación

Este inciso describe los pasos para ejecutar las simulaciones de la tarea #3. **Recordar que el paso final es solo correr el comando `make` en la terminal dentro del directorio que contenga los siguientes archivos.**

4.1. Paso #1: Archivos Necesarios

Para la correcta utilización de la simulación, es importante organizar todos los archivos en un único directorio. Dicho directorio debe contener exclusivamente los siguientes archivos:

- `cpu.ys`
- `cpu_synth.v`
- `cpu.v`
- `rom.v`
- `ram.v`
- `cmos_cells.lib`
- `cmos_cells.v`
- `Completo.gtkw`
- `Makefile`
- `testbench.v`
- `tester.v`

4.2. Paso #2: Explicación de uso del Makefile

Dentro del archivo Makefile, la primera parte corresponde a 2 líneas de código comentadas con el fin de que el usuario elija qué archivo simular.

```
1 # Archivos fuente del proyecto // Comente y descomente en base a la versión que
  ↪  quiera observar
2 #SRC = rom.v ram.v tester.v testbench.v cpu.v
3 SRC = rom.v ram.v tester.v testbench.v cpu_synth.v
```

Listing 1: Opciones de selección de archivos fuente en la variable `SRC`

En este caso, la variable `SRC` define los archivos fuente que se utilizarán en la simulación. Se puede escoger la opción normal (cpu.v) y la opción sintetizada (cpu_synth.v).

5. Resultados Obtenidos

5.1. Prueba con cpu.v

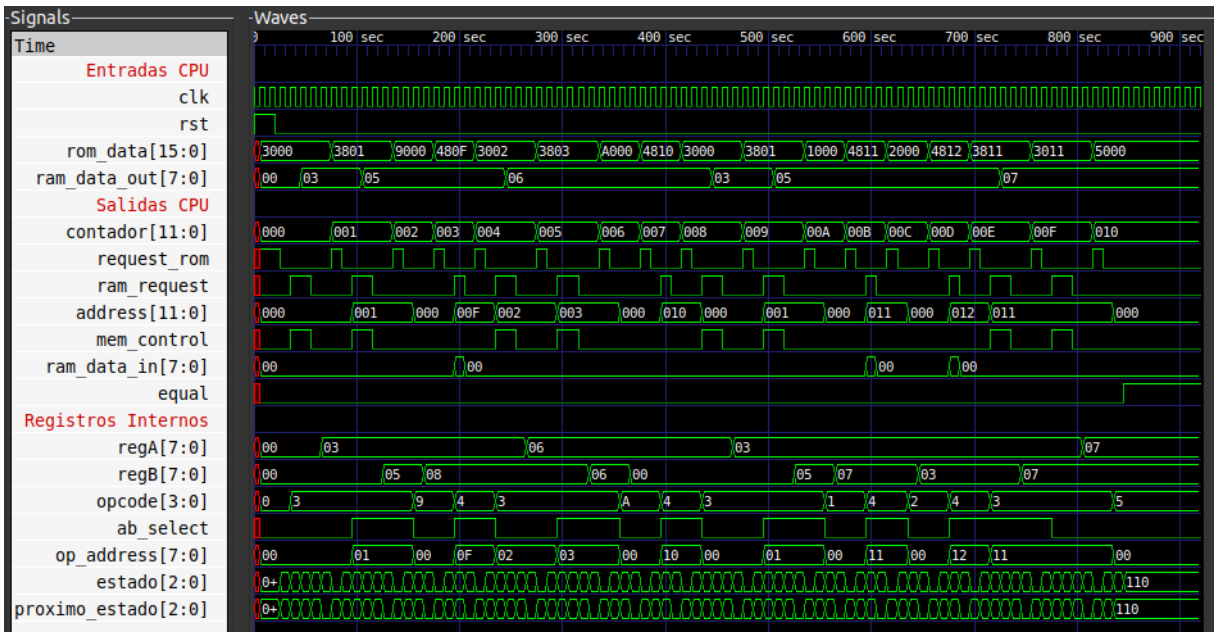

Figura 2: Resultado de la simulación original

Como se observa en la figura 2, esta muestra la simulación completa del cpu. Este resultado muestra cómo se ve la máquina de estados original sin haber pasado por ninguna síntesis. Este es el resultado obtenido al correr el Makefile con `SRC = rom.v ram.v tester.v testbench.v cpu.v`. En la figura se muestra como el registro A y B van cambiando su valor con el tiempo, con respecto a como se efectuaron las instrucciones en la rom. Note que la señal `contador` comienza en 0 y termina en la posición 16, es decir toda la ROM se leyó. Al final es notable como ambos registros toman el valor de 7 y al aplicar la instrucción de EQUAL, el sistema se detiene.

Como se observa en la figura 2, esta muestra la simulación completa del controlador de CPU en su implementación original. Este resultado corresponde a la ejecución del diseño conductual en Verilog sin haber pasado por ningún proceso de síntesis, es decir, representa el comportamiento ideal del sistema tal como fue especificado en el código fuente. Este es el resultado obtenido al ejecutar el Makefile configurado con `SRC = rom.v ram.v tester.v testbench.v cpu.v`, donde se incluye el módulo de CPU original sin sintetizar.

En la figura se puede apreciar claramente cómo los registros A y B van modificando sus valores a lo largo del tiempo, reflejando la ejecución secuencial de las instrucciones almacenadas en la ROM. El registro A comienza en 0 debido al reset inicial, luego toma el valor 3 al ejecutarse la primera instrucción LOAD desde la posición 0 de RAM, posteriormente cambia a 6 cuando se ejecuta otra instrucción LOAD desde la posición 2, y finalmente vuelve a tomar el valor 3 en las operaciones subsecuentes que recargan datos desde las posiciones iniciales de memoria, y finalmente llega a 7 al cargarse la posición 17 de la ram. De manera similar, el registro B también experimenta múltiples transiciones: inicia en 0, carga el valor 5 desde RAM[1], se actualiza a 8 después de la operación ADD (3+5), en total finalmente retorna a 7 en preparación para la instrucción EQUAL.

La señal `contador`, que representa el contador de programa (PROG_ADDR), inicia correctamente en la posición 0 inmediatamente después de la desactivación del reset, y se incrementa

de manera secuencial después de completar cada instrucción. La progresión del contador es evidente en la figura: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, hasta finalizar en la posición 16, indicando que todas las 17 instrucciones programadas en la ROM fueron leídas y ejecutadas exitosamente.

Al final de la simulación, es particularmente notable observar cómo ambos registros convergen al mismo valor. Específicamente, las últimas instrucciones LOAD cargan el valor 7 tanto en el registro A como en el registro B desde la posición 17 de RAM respectivamente. Cuando se ejecuta la instrucción EQUAL en la posición 16 de la ROM, el controlador detecta correctamente que ambos registros contienen valores idénticos, activando la bandera `equal` que se puede observar transicionando a 1 en la figura. Esta activación desencadena la transición de la máquina de estados al estado DETENER (110 en binario), donde el sistema permanece indefinidamente sin ejecutar más instrucciones. Se puede verificar que después de este punto el contador de programa ya no se incrementa, las señales de control (`request_rom` y `ram_request`) permanecen en 0, y los valores de los registros se mantienen estables, confirmando así el correcto funcionamiento de la condición de terminación del programa según las especificaciones de la tarea.

5.2. Prueba con cpu_synth

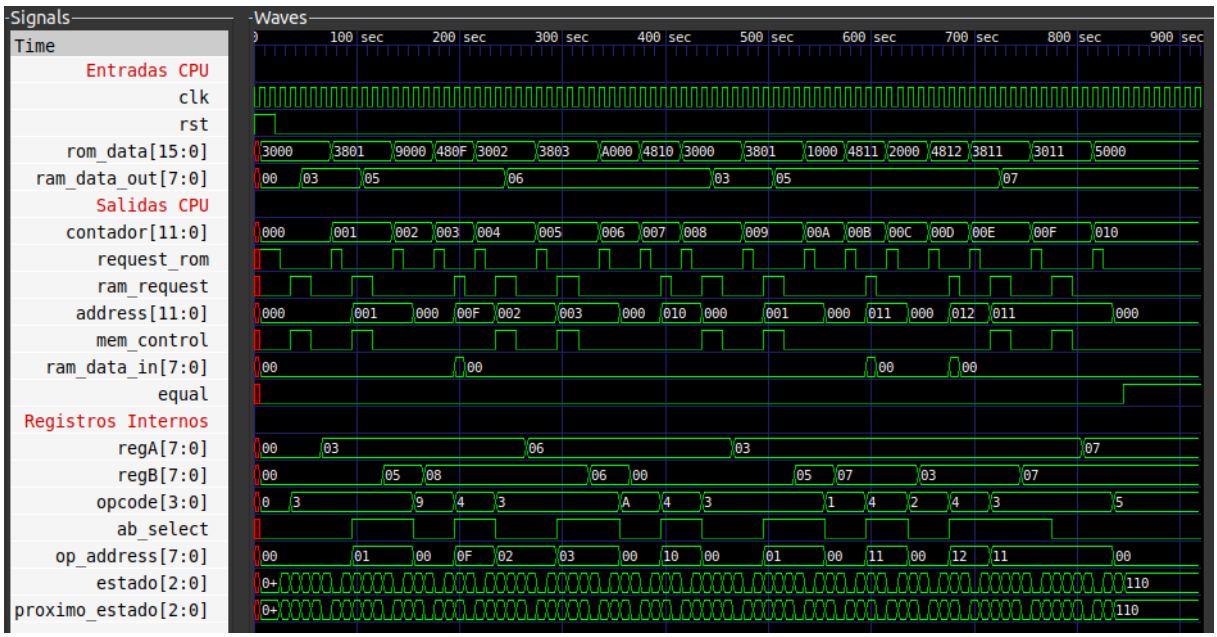

Figura 3: Resultado de la simulación final sin retardos

Como se observa en la figura 3, esta muestra la simulación completa del cpu utilizando el archivo `Cajero_synth.v`, generado tras completar todos los pasos del archivo cpu.ys, es decir, con la síntesis final ya realizada. El resultado mostrado confirma que, como era esperado, la máquina de estados opera de manera idéntica al código original de la figura 2. Por lo tanto, se comprueba que la síntesis fue exitosa. Este resultado se obtuvo ejecutando el `Makefile` con `SRC = rom.v ram.v tester.v testbench.v cpu_synth.v`, utilizando `gtkwave Completo.gtkw` y el `Testbench.v` con el `include cmos_cells.v`.

En el código sintetizado del archivo `cpu_synth.v` generó la siguiente cantidad de elementos lógicos: 202 compuertas `NAND`, 258 compuertas `NOR`, 90 compuertas `NOT` y 45 flip-flops tipo `DFF`. Por lo tanto se comprueba que la sintesis fue realizada de forma correcta.

6. Conclusiones

El diseño e implementación del controlador de CPU demostró ser exitoso, logrando ejecutar correctamente todas las instrucciones de la ROM y validando el funcionamiento completo del sistema mediante pruebas exhaustivas. Uno de los principales retos superados fue el manejo apropiado de la RAM, que requirió la incorporación de un estado adicional (ESPERAR_RAM) en la máquina de estados finita para garantizar que los datos estuvieran estables antes de ser capturados por los registros, evitando así lecturas inválidas. Otro desafío significativo fue la correcta separación entre la lógica secuencial y combinacional del controlador, especialmente en el manejo de las señales de control (request_rom, ram_request, mem_control) que debían activarse y desactivarse en momentos precisos según el estado actual, requiriendo múltiples iteraciones para eliminar latches indeseados y garantizar que todas las señales tuvieran valores definidos en cada ciclo de reloj. La síntesis del diseño con Yosys presentó desafíos adicionales relacionados con el mapeo de flip-flops y compuertas lógicas. Los resultados obtenidos confirman que tanto el diseño original como el sintetizado producen comportamientos idénticos, validando que la metodología de diseño empleada es correcta. Se recomienda para futuros trabajos la implementación de un conjunto de instrucciones más amplio que incluya saltos condicionales y subrutinas, así como la optimización del número de estados para reducir los ciclos de reloj necesarios por instrucción, particularmente para operaciones que no requieren acceso a memoria.