Implementación de CPU RISC-V

Autor: Díaz Gerardo Agustín

Carrera: Ingeniería Electrónica

Materia: Electrónica II

Año: 2025

Resumen

Este informe presenta el diseño e implementación de un microprocesador de 32 bits basado en la arquitectura de conjunto de instrucciones de código abierto RISC-V, específicamente el subconjunto de enteros RV32I. El proyecto aborda el desarrollo completo de la microarquitectura, diferenciando los conceptos teóricos de arquitectura y organización de computadoras. La implementación se realiza mediante la descripción de hardware en lenguaje VHDL, construyendo un Datapath (camino de datos) interconectado con una Unidad de Control basada en una Máquina de Estados Finitos (FSM) para gestionar los ciclos de búsqueda, decodificación y ejecución. El sistema integra una memoria SRAM de 512x32 bits y una interfaz de periféricos de entrada/salida de 8 bits. Finalmente, se exponen los resultados de la validación mediante simulación funcional y la síntesis del diseño en la plataforma de hardware EDU-CIAA-FPGA, verificando el correcto procesamiento de instrucciones aritméticas, lógicas y de control de flujo.

Introducción

Arquitectura

La arquitectura es la perspectiva que tiene el programador de una computadora. Se define por el conjunto de instrucciones (lenguaje) y las ubicaciones de los operandos (registros y memoria). Existen muchas arquitecturas diferentes, tales como RISC-V, ARM, x86, MIPS, SPARC y PowerPC. (Harris & Harris, 2019).

Microarquitectura

Es la implementación específica de esa arquitectura en hardware. Incluye el diseño del datapath (camino de datos), la organización de la memoria y, crucialmente, la Unidad de Control (FSM). Diferentes microarquitecturas (monociclo, multiciclo, segmentada) pueden ejecutar el mismo conjunto de instrucciones RISC-V con diferente rendimiento y costo. (Harris & Harris, 2019)

Desarrollo

Arquitectura RISC-V y Conjunto de Instrucciones RV32I

RISC-V se distingue por ser la primera arquitectura de conjunto de instrucciones (ISA) de código abierto que cuenta con un amplio soporte comercial. Esta arquitectura se clasifica como de tipo Load/Store, lo que implica que el acceso a la memoria se realiza exclusivamente mediante instrucciones de carga y almacenamiento, mientras que las operaciones aritméticas ocurren únicamente dentro de los registros.

En el diseño de RISC-V, cada instrucción se representa como una palabra de 32 bits. Para este proyecto, nos centramos en el conjunto de instrucciones de enteros de 32 bits conocido como RV32I (versión 2.2), el cual constituye el núcleo fundamental de la arquitectura. Este análisis se basa en la definición oficial estipulada en el Manual del Conjunto de Instrucciones RISC-V. (Harris & Harris, 2019)

Conjunto de Registros de Arquitectura RISC-V

En el Harris & Harris, 2019 especifica que RV32I tiene 32 registros de propósito general de 32 bits (x0 a x31), los cuales se detallan a continuación en la Tabla 1.

Registro	Nombre	Descripción/Uso convencional
x0	zero	Constante de valor 0 (cero). Siempre devuelve el valor cero
x1	ra	Dirección de retorno. Usado por jal y jalr para saber dónde volver después de una subrutina.
x2	sp	Puntero de pila. Apunta al tope de la pila en memoria
x3	gp	Puntero global. Apunta a datos estáticos globales para acceso rápido.
x4	tp	Puntero de hilo. Usados en sistemas operativos para gestionar hilos.
x5 – x7	t0-t2	Temporales. Registros para cálculos intermedios. No se preservan tras llamar a una función.
x8	s0 / fp	Registro guardado / puntero de marco. A veces usado para apuntar a la base del marco de la pila.
x9	s1	Registro guardado. Debe ser preservado por la función llamada.
x10 – x11	a0-a1	Argumentos / Valor de retorno. Se usan para pasar los primeros 2 argumentos a una función y para devolver resultados.
x12 – x17	a2-a7	Argumentos de función. Se usan para pasar los argumentos restantes (del 3 al 8)
x18 – x27	s2 – s11	Registros guardados. Variables de larga duración que deben ser restauradas si una función las modifica.
x28 – x31	t3 – t6	Temporales. Más registros para cálculos intermedios que no necesitan conservarse.

Tabla 1: Conjunto de registros RISC-V

Conjunto de instrucciones RV32I

El libro Harris & Harris, 2019 detalla 6 formatos principales de instrucciones de 32 bits que se muestran en la Tabla 2.

Tipo	31-25	24-20	19-15	14-12	11-7	6-0	Descripción
R	funct7	rs2	rs1	func3	rd	opcode	Operaciones entre dos registros fuente (rs1, rs2) y un destino (rd).
I	imm[11:0]		rs1	func3	rd	opcode	Operaciones con un registro fuente (rs1), un inmediato de 12 bits y un destino (rd). También cargas (lw).
S	imm[11:5]	rs2	rs1	func3	imm[4:0]	opcode	Guarda un registro (rs2) en memoria usando una dirección base (rs1) más un offset inmediato.
B	imm[12, 10:5]	rs2	rs1	func3	imm[4:1, 11]	opcode	Saltos condicionales. Compara dos registros.
U	imm[31:12]				rd	opcode	Carga inmediatos de 20 bits en la parte alta del registro.
J	imm[20, 10:1, 19:12]				rd	opcode	Saltos incondicionales largos con enlace.

Tabla 2: Conjunto de Instrucciones RV32I

- **opcode (6:0)**: Código de operación básico. Determina el tipo general de instrucción.
- **rd (11:7)**: Register Destination. Registro donde se guarda el resultado (excepto en S y B, que no escriben).

- **funct3 (14:12)**: Función de 3 bits. Refina la operación (ej. diferencia entre ADD y SUB, o BEQ y BNE).
- **funct7 (31:25)**: Función de 7 bits. Usado en tipo R para distinguir operaciones (ej. bit 30 distingue ADD/SUB o SRL/SRA).
- **rs1 (19:15)**: Register Source 1. Primer operando fuente.
- **rs2 (24:20)**: Register Source 2. Segundo operando fuente (solo para tipos R, S, B).
- **imm**: Valor inmediato (constante numérica). Se observa que en los tipos S y B está dividido, y en B y J está "desordenado" para mantener fijos los bits de signo.

Cálculo del Valor Inmediato (Tipos I, S, B, U, J)

El Harris & Harris, 2019 explica que el hardware debe "Extender el Signo" de los inmediatos para convertirlos a 32 bits.

- Tipo I: Simplemente toma los 12 bits superiores de la instrucción (inst[31:20]) y extiende el bit 31 (signo) hasta llenar los 32 bits.
- Tipo S: El inmediato está dividido. Se concatenan inst[31:25] y inst[11:7] y luego se extiende el signo.
- Tipo B: Es complejo para mantener la posición del bit de signo fija. El inmediato se reconstruye "barajando" bits: inst[31], inst[7], inst[30:25], inst[11:8]. El bit 0 siempre se asume 0 (saltos pares).

Análisis de Opcodes: 19 y 51 (ALU)

Opcode 19 (Decimal) → 0010011 (Binario)

Tipo de Instrucción: I-Type (Aritmética con Inmediato).

Ejemplos: addi, slti, xori, ori, andi.

Cuando la Unidad de Control ve que los últimos 7 bits son 0010011, sabe lo siguiente:

1. Es una operación que involucra un registro y un número constante (inmediato).
2. Debe habilitar la extensión de signo del inmediato.
3. El segundo operando de la ALU (SrcB) será el Inmediato, no un registro.

¿Cómo sabe la ALU qué operación hacer?

Como el Opcode es el mismo para todas estas instrucciones (sumar inmediato, and inmediato, etc.), el control debe mirar el campo funct3 (bits 14-12 de la instrucción).

- Si funct3 = 000 → La ALU debe SUMAR (Instrucción addi).
- Si funct3 = 111 → La ALU debe hacer AND (Instrucción andi).
- Si funct3 = 110 → La ALU debe hacer OR (Instrucción ori).

Para Opcode 19: La operación de la ALU está determinada únicamente por el campo **funct3**. (Nota: En este opcode NO existe la instrucción de "restar inmediato", por lo que no necesitamos funct7 para distinguir suma de resta). (Harris & Harris, 2019).

Opcode 51 (Decimal) → 0110011 (Binario)

Tipo de Instrucción: R-Type (Aritmética entre Registros).

Ejemplos: add, sub, sll, xor, srl, or, and.

Cuando el control ve 0110011, sabe:

1. Es una operación entre dos registros.
2. El segundo operando de la ALU (SrcB) proviene del Banco de Registros (Read Data 2).

¿Cómo sabe la ALU qué operación hacer? (El problema de ADD vs SUB)

Aquí la situación es más compleja. La ALU mira el campo funct3, pero esto no es suficiente.

- Para add (sumar), el funct3 es 000.
- Para sub (restar), el funct3 TAMBIÉN es 000.

¿Cómo los distingue el hardware? Aquí entra en juego el campo funct7 (bits 31-25), y específicamente el bit 30 de la instrucción.

- **Instrucción add:**
    - Opcode: 51
    - funct3: 000
    - funct7: 0000000 (El bit 30 es **0**).
    - Acción ALU: Sumar (A + B).
- **Instrucción sub:**
    - Opcode: 51
    - funct3: 000
    - funct7: 0100000 (El bit 30 es **1**).
    - Acción ALU: Restar (A - B).

Para Opcode 51: La operación de la ALU depende de funct3 y del bit 30 del funct7. (Harris & Harris, 2019).

Análisis del Opcode 99: Saltos Condicionales (Branch)

Según el Harris & Harris, 2019 y el mapa de Opcodes, el valor decimal 99 corresponde al binario 1100011, que identifica a las instrucciones de tipo Branch (Tipo B).

Objetivo: Determinar cómo la ALU decide si se toma el salto o no.

A. Operación de la ALU

Para todas las instrucciones de salto (beq, bne, blt, bge), la ALU realiza una RESTA (SUB).

- **Operación:** Resultado = Registro[rs1] - Registro[rs2]
- La ALU no guarda este resultado numérico en ningún registro destino (en instrucciones tipo B, no se escribe en rd). Solo se utiliza para activar las "banderas" (flags) de estado.

B. La Condición de Cero (Zero Flag)

El requerimiento específico de tu actividad es explicar la "condición de cero".

La ALU tiene una salida de 1 bit llamada Zero.

- Si el resultado de la resta es 0 (es decir, rs1 = rs2), la señal Zero se pone en 1 (Valor Alto).
- Si el resultado de la resta es distinto de 0 (es decir, rs1 ≠ rs2), la señal Zero se pone en 0 (Valor Bajo).

C. Lógica de Decisión

La Unidad de Control evalúa el funct3 y la señal Zero para decidir si escribe el PC con la nueva dirección (tomar el salto) o simplemente pasa a la siguiente instrucción (PC + 4).

En el Opcode 99, la ALU se configura para realizar una operación de sustracción entre los operandos fuente. La decisión de tomar el salto depende de la señal 'Zero' generada por la ALU. En el caso de beq, el salto se ejecuta si la señal Zero está activa (alto); en el caso de bne, se ejecuta si la señal Zero está inactiva (bajo), lo cual puede verse más claro en la Tabla 3.

Instrucción	func3	Significado	Condición para tomar el salto
beq (Branch if Equal)	000	Saltar si son iguales	Si Zero = 1
bne (Branch not equal)	001	Saltar si son distintos	Si Zero = 0

Tabla 3: Condiciones de salto

Resultados

En la Figura 1 se muestra el datapath implementado de la CPU con cada uno de sus bloques funcionales (RAM, ALU, Control, etc).

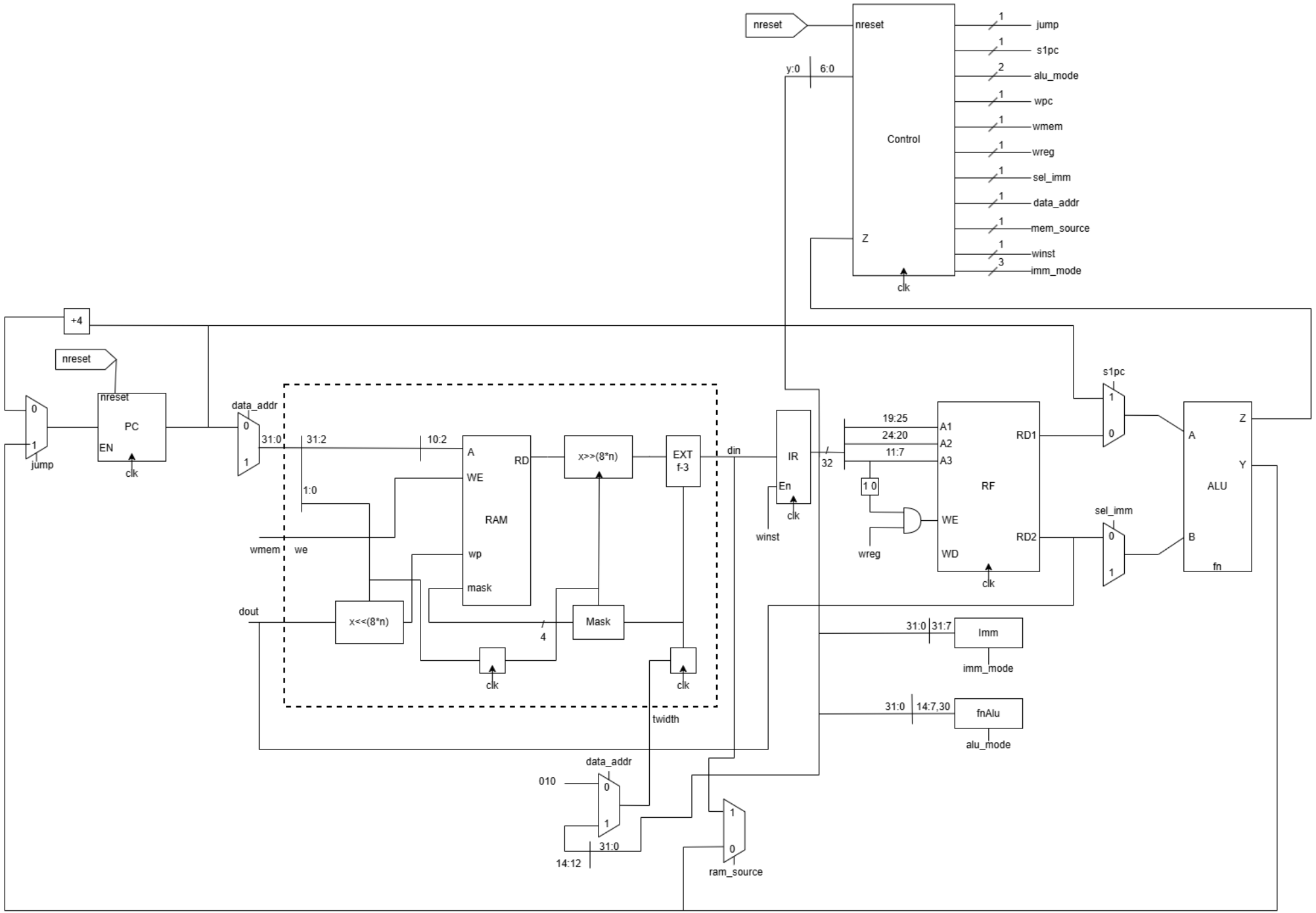

Figura 1: Datapath completo CPU RISC-V

En la Figura 2 se muestra el diagrama de estados implementado para el bloque que controla la CPU.

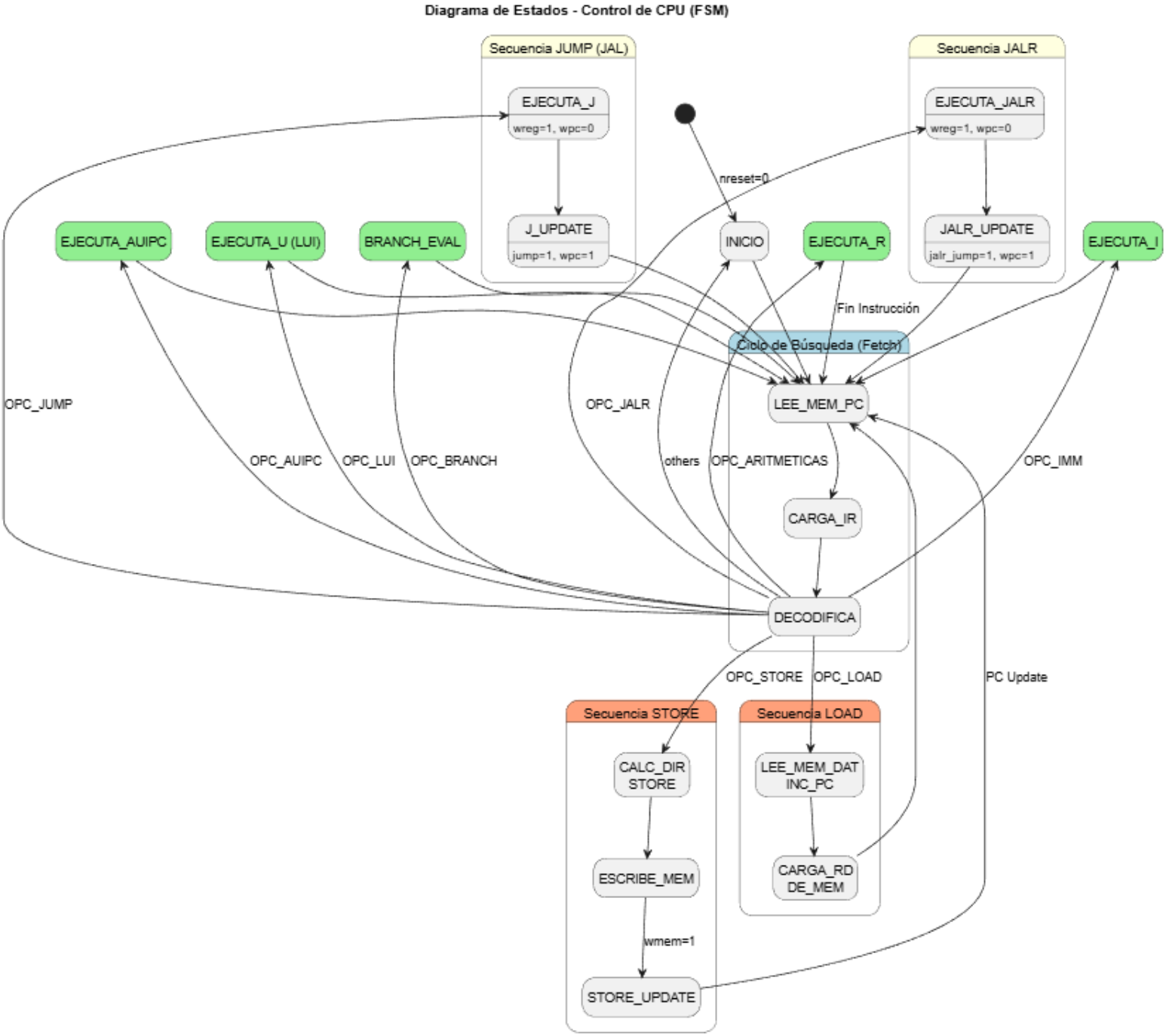

Figura 2: Diagrama FSM del control de CPU

Descripción:

El esquema representa la implementación de la Unidad de Control como una Máquina de Estados Finita. Su funcionamiento se estructura en tres fases cíclicas:

1. **Fase de Búsqueda y Decodificación**:

Todos los ciclos inician con una secuencia fija (LEE_MEM_PC → CARGA_IR → DECODIFICA). En esta etapa, el procesador recupera la instrucción de la memoria, la almacena en el registro de instrucción (IR) y evalúa el código de operación (Opcode) para determinar la ruta a seguir.

2. **Fase de Ejecución:**

Desde el estado de decodificación, el flujo se bifurca según la complejidad de la instrucción:

- **Instrucciones Simples:** Las operaciones aritméticas (Tipo R, I), de carga de constantes o cálculo de direcciones relativas (Tipo U) y saltos condicionales (Branch) se resuelven en un único estado de ejecución antes de retornar.
- **Instrucciones Complejas (Multiciclo):** Las operaciones de acceso a memoria (LOAD, STORE) y saltos incondicionales (JAL, JALR) requieren secuencias de 2 o 3 estados encadenados. Esto se diseñó así para garantizar la estabilidad de las señales de escritura (wmem) y direccionar correctamente los buses antes de actualizar el Contador de Programa (PC).

3. **Retorno:**

Independientemente de la ruta tomada, todos los caminos convergen nuevamente al estado inicial de búsqueda (LEE_MEM_PC), asegurando que el procesador esté listo para ejecutar la siguiente instrucción de forma continua y sincronizada.

Resultados de Simulación y Validación

Para verificar el correcto funcionamiento de la arquitectura diseñada, se llevó a cabo una estrategia de validación incremental utilizando bancos de pruebas y análisis de formas de onda en GTKWave. El proceso se dividió en pruebas unitarias por tipo de instrucción y pruebas de integración con algoritmos completos.

Validación Incremental y Depuración

Se probaron secuencialmente los distintos formatos de instrucción soportados por la arquitectura RISC-V. Durante esta etapa, se detectaron errores de temporización y lógica que requirieron modificaciones en los bloques cpu.vhd y control_cpu.vhd.

A. Instrucciones Aritmético-Lógicas (Tipo R y Tipo I)

- **Prueba**: Se verificaron operaciones básicas como ADD, SUB y ADDI.
- **Observación**: Se monitorearon las señales del Banco de Registros (rf_din, rf_addr_w) para confirmar que los resultados de la ALU se escribían en el registro destino correcto.
- **Corrección**: Inicialmente, se detectó que el Contador de Programa (PC) no avanzaba tras ejecutar estas instrucciones, quedándose estancado en la dirección 0x00. Se corrigió la lógica en control_cpu asegurando la activación de la señal wpc al finalizar el estado de ejecución, permitiendo el flujo secuencial del programa.

B. Instrucciones de Acceso a Memoria (Tipo S y Load)

- **Prueba**: Se simuló la instrucción SW (Store Word) para escribir datos en la memoria RAM.
- **Problema Crítico de Timing**: Durante las primeras simulaciones, se observó un conflicto de tiempos. La señal de escritura en memoria (wmem) permanecía activa mientras el PC intentaba actualizar la dirección para la siguiente instrucción. Esto causaba escrituras erróneas en direcciones no deseadas.
- **Solución en control_cpu**: Se modificó la Máquina de Estados Finita agregando un estado intermedio denominado STORE_UPDATE. Esto separó la operación en dos fases:
    - ESCRIBE_MEM: Mantiene la dirección y datos estables con wmem='1'.
    - STORE_UPDATE: Desactiva la escritura y actualiza el PC. Esta modificación eliminó el error y estabilizó la escritura en memoria.

C. **Instrucciones de Control de Flujo (Tipo B y J)**

- **Prueba**: Se validaron los saltos condicionales (BEQ) y saltos incondicionales (JAL).
- **Corrección en cpu.vhd**: Se detectó un error lógico en el cálculo del pc_next. La CPU interpretaba erróneamente señales de la ALU (take_branch) incluso en instrucciones que no eran saltos (como un ADDI con resultado 0). Se refinó el proceso del PC para que solo obedezca a las señales de salto cuando la Unidad de Control active explícitamente la salida jump.
- **Corrección Crítica para saltos indirectos (JALR)**: Al escalar las pruebas para soportar código compilado desde C, se detectó que la instrucción JALR fallaba al calcular la dirección de destino. A diferencia de JAL y los Branches que usan saltos relativos al PC, JALR requiere saltar a una dirección absoluta calculada por la ALU (Registro + Inmediato). Se solucionó este problema arquitectónico introduciendo una nueva señal dedicada (jalr_jump) en la Unidad de Control y modificando el multiplexor del PC en cpu.vhd para que enrute directamente la salida de la ALU hacia el pc_next cuando se ejecuta esta instrucción.

Pruebas de Integración (Programas Completos)

Una vez depurados los componentes individuales, se cargaron programas complejos en la memoria de instrucciones para validar la interacción de todos los módulos.

Descripción: La validación culminante de la arquitectura consistió en la ejecución de un programa escrito en lenguaje C, diseñado para funcionar como un contador de 0 a F (15 en hexadecimal). El valor del conteo debía ser enviado a un módulo periférico de salida (interface_out) mapeado en la dirección de memoria 0x80000000, cuyos 8 bits menos significativos controlan un display de 7 segmentos.

Detección de Errores (Instrucciones Faltantes): Al cargar el firmware compilado con GCC, la simulación reveló que la CPU no lograba ejecutar el programa correctamente. El análisis exhaustivo demostró que el compilador generaba instrucciones avanzadas que la Unidad de Control original no soportaba, específicamente AUIPC y JALR (fundamentales para el cálculo de direcciones relativas y el retorno de funciones en C, respectivamente).

Corrección y Resultado Final: Se procedió a actualizar la Unidad de Control, añadiendo los estados necesarios (EJECUTA_AUIPC, EJECUTA_JALR, JALR_UPDATE) para decodificar estas instrucciones. Además, se implementó una corrección crítica en el datapath (cpu.vhd) agregando la señal jalr_jump para permitir que el PC cargue direcciones absolutas desde la ALU. Tras estas correcciones, el procesador logró ejecutar la secuencia correcta en el display, confirmando que la CPU es 100% funcional.

Conclusiones

Conceptos Aprendidos: A través de este diseño consolidé el funcionamiento de la arquitectura Multiciclo. El aprendizaje más significativo fue sobre la sincronización y el timing digital: la necesidad de diseñar estados intermedios para estabilizar buses y señales de escritura demostró que, en hardware, el control preciso del tiempo es tan importante como la lógica booleana.

Valoración del Diseño: El procesador obtenido es funcional, estable y modular. Ha demostrado capacidad para ejecutar algoritmos complejos que integran aritmética, acceso a memoria y bucles de control de flujo sin presentar errores. Más allá de la validación a nivel ensamblador, el mayor logro del proyecto fue la ejecución exitosa de código de alto nivel (Lenguaje C). La correcta interacción del firmware compilado con herramientas GCC y el periférico mapeado en memoria (control del display de 7 segmentos) demuestra que

la CPU no solo procesa datos internamente, sino que es capaz de interactuar con el mundo físico. Se valora especialmente la robustez en la gestión del bus y la modularidad del código VHDL, lo cual deja una base sólida y escalable para futuras expansiones del set de instrucciones RISC-V.

Referencias

- Harris, S. L., & Harris, D. (2019). Digital design and computer architecture: RISC-V edition. Morgan Kaufmann.