Implementación de una microarquitectura para el ISA RISCV

Jeaustin Calderón Quesada, Gabriel Campos Oviedo, Esteban Chavarría Chaves, Diana Cerdas Vargas, Emmanuel Muñoz Peña
jeauscq@estudiantec.cr gabrielitooviedito@estudiantec.cr echavarriach@estudiantec.cr, diana@estudiantec.cr, emmanuel111232@estudiantec.cr,
Área académica de Ingeniería Mecatrónica
Instituto Tecnológico de Costa Rica

***Resumen*—El conjunto de instrucciones RISC-V ha ganado gran popularidad en los últimos años debido a que presenta una arquitectura de código abierto que permite una gran flexibilidad para su implementación en microprocesadores. Se tiene que en este proyecto desarolla el diseño y simulación de un microprocesador capaz de soportar las instrucciones de tipo memoria, aritmético lógicas, branch, jump y algunas pseudoinstrucciones como por ejemplo "li". Esto mediante el lenguaje de descripción de hardware Verilog y la herramienta de síntesis y simulación de circuitos Vivado.**

***Palabras clave*—RISCV, microprocesadores, arquitectura, Verilog**

I. Introducción

El conjunto de instrucciones RISC-V, se ha ganado rápidamente una gran aceptación en la comunidad de diseño de microprocesadores debido a su simplicidad, flexibilidad y capacidad para ser implementado en una amplia gama de aplicaciones. Se tiene que como lo define [1] RISC-V es una arquitectura de conjunto de instrucciones (ISA) de código abierto basada en los principios de computadora con conjunto de instrucciones reducido (RISC).

Por su parte, cabe destacar que RISC se describe como una arquitectura caracterizada por un conjunto de instrucciones reducido y simple, instrucciones de tamaño fijo, uso de registros de propósito general, acceso a memoria explícito, diseño de pipeline y una arquitectura de registro-almacenamiento. Estas características logran minimizar la complejidad del hardware y la codificación de instrucciones necesaria al mantener pequeño el conjunto de instrucciones pequeño. [2].

Estos conjuntos de instrucciones son implementados en una tecnología específica, a la cual se le llama procesador. Como lo menciona [2] este va a estar compuesto por varios componentes esenciales, como la unidad de control, la unidad aritmético lógica, los registros y la unidad de memoria, logra la ejecución de cada una de las instrucciones de la arquitectura.

En este proyecto se realizó la implementación de la arquitectura RISC-V 32I que consistiría en la versión de 32 bits y en las operaciones de tipo memoria, salto condicional e incondicional y aritmético lógicas, mediante la creación de un procesador uniciclo, que como lo define [3] es una arquitectura que ejecuta una instrucción por ciclo de reloj, logrando evitar el uso de unidades de control de riesgo y otros componentes necesarios para procesadores pipeline, por ejemplo. Dicha implementación se logró mediante el lenguaje de descripción de hardware Verilog y la herramienta computacional Vivado.

Se tiene que en este proyecto se evaluaron diferentes tipos de operaciones, primeramente, las tipo S, las cuales consistían en instrucciones como store y load, que se encargarían de guardar o cargar objetos de memoria ya sea desde o hacia registros, respectivamente. Luego de esto, se tiene que se abarcaron las instrucciones de tipo salto, ya sea incondicional (jump) o condicionales (branch) las cuales darían un redireccionamiento de la instrucción leída en funciones con ciclos o condicionales. Por último, se tiene las funciones aritmético lógicas que consistirían en operaciones con la ALU como la función addi, asimismo, se tiene el soporte de pseudoinstrucciones.

II. DESCRIPCIÓN DE LA SOLUCIÓN

II-A. Diseño de la microarquitectura

El diseño de microarquitectura plantea una lógica digital capaz de soportar instrucciones del tipo I, S, J, B y U, en concreto, las instrucciones requeridas por este proyecto son: add, sw, li, lui, j, lw y blt, contenidas en RV32I Base Integer Instruction Set, Version 2.1". En otras palabras, la microarquitectura, respeta la arquitectura de código abierto RISC-V. El único tipo de instrucción no soportado por este diseño, es el tipo R, ya que ninguna de las instrucciones requeridas tiene esa codificación. Los elementos básicos que componen esta microarquitectura son: registro para el PC, memoria de instrucciones, banco de registros, ALU, memoria y unidad de extensión de signo. En la figura 4 se muestra la implementación final. Para describir el diseño de esta implementación, es importante tener claras las diferencias y similitudes entre las codificaciones de cada uno de los tipos de instrucción. Estas se pueden observar en la figura 5 donde se observan las distintas codificaciones para la arquitectura RISC-V. Las instrucciones, indiferentemente del tipo, se componen de 32 bits y estos bits se agrupan dependiendo del tipo de codificación. A continuación, se discuten cada una de las agrupaciones y que impacto tuvieron las diferencias entre instrucciones en la toma de decisiones para implementar la microarquitectura.

II-A1. opcode: Estos bits brindan información sobre el tipo de instrucción que se está enviando, es de uso exclusivo de la unidad de control. La arquitectura exige que el opcode

corresponda a los primeros 7 bits. Sin embargo, para esta implementación en específico, por las instrucciones que amerita soportar, únicamente se utilizaron los 4 bits de mayor valor del opcode, esto porque todas las instrucciones tienen los dos bits de menor valor igual a "11". Se puede observar en la figura 4 que de la memoria de instrucciones a la UC, únicamente van los bit [6-3].

II-A2. rd: Corresponde a 5 bits que se encuentran del bit 7 al bit 11. Este es la dirección del registro de destino, únicamente está implementado en las instrucciones I, U y J. Esta dirección indica cuál registro se va a modificar, en caso de que se vaya a requerir. Para poder controlar cuando se va a escribir o no en este registro, se usa la señal de control RegWriteEn, esto porque en los casos de las instrucciones B y S, estos bits no corresponden a un registro de destino y por esto no tendría sentido modificar el banco de registros.

II-A3. funct3: Para esta implementación no fue necesario tomar en cuenta los bits que componen funct3 que están localizados en los bits del 12 al 14 en las codificaciones tipo I, S y B. Es una señal de control, pero en este caso, el opcode brinda información suficiente.

II-A4. rs1 y rs2: Corresponden a las dos agrupaciones que indican cuáles son las direcciones de registros que se desean leer, corresponden a los bits del 15 al 19 y del 20 al 24 respectivamente. El dato rs1 se usa en los tipos I, S y B, mientras que rs2 se usa en los tipos S y B. En las instrucciones tipo I usa rs1, pero no rs2, esto se da porque en vez de la información almacenada en un segundo registro, utiliza un inmediato y para poder escoger este inmediato, se implementó un multiplexor controlado por la señal ALUSrc

II-A5. imm: Este corresponde al inmediato y particularmente en la arquitectura RISC-V presenta muchas variaciones en su posición dependiendo de la codificación. El inmediato se trabaja como valores de 32 bits, sin embargo, según la codificación los inmediatos se componen de doce o veinte bits, para resolver esto se utiliza la unidad de extensión de signo, la cuál recibe una señal de control denominada "ImmSel" que le indica el tipo de codificación que se está utilizando, permitiendo realizar el acomodo pertinente al inmediato. A continuación, se van a discutir distintas situaciones que implicaron consideraciones relevantes en la microarquitectura.

II-A6. PC: PC en la arquitectura usada corresponde al registro 32 y en él se almacena el valor de la dirección de memoria que contiene la instrucción que se debe ejecutar. Para almacenar dicho valor y manejar el flujo entre ciclos, se usa un registro implementado como una flip-flop tipo D. Bajo condiciones "normales" el PC va aumentado de cuatro en cuatro y así lee las instrucciones una tras otra, este aumento se realiza con un sumador exclusivo para este fin, ya que se usa en la gran mayoría de los casos, por lo que no se puede depender de la ALU. Sin embargo, algunas instrucciones modifican este valor sumándole un inmediato. Esta suma se realiza mediante un sumador exclusivo, esto se hace porque las instrucciones tipo B, son condicionales y utilizan la ALU para calcular dicha condición por lo que se necesita este sumador extra que produce "NewPC" que corresponde a la dirección alternativa. Un multiplexor conectado al FF tipo D se encarga de cuál dirección va a ser la próxima y este a su vez es controlado por PCSrc que en los casos de las instrucciones tipo J siempre es 1 y en los casos de las instrucciones tipo B, corrobora que la condición se cumpla mediante la flag "Negative" proveniente de la ALU, esta lógica se puede observar en la parte superior derecha de la figura 4. Asimismo, en la figura 1 se muestra la lógica discutida.

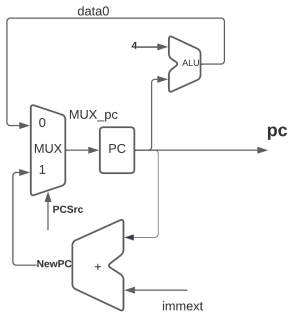

Figura 1. Lógica de PC.

II-A7. Leer en memoria: La única operación que realizar una lectura en memoria corresponde a lw, la dirección de memoria siempre proviene de la ALU y la señal se llama "ALUResult". Esta dirección es el resultado de una operación entre la información que proviene de dos registros o de un registro y una inmediato, esto es controlado por la señal ALUSrc como se discutió anteriormente.

II-A8. Escribir en memoria: Para escribir en memoria, la dirección proviene de una operación realizar en la ALU entre el valor almacenado en un registro (rs1) y el inmediato, mientras que el dato a guardar viene de lo que estaba almacenado en un registro (rs2). Cuando se presentan este tipo de instrucciones, la señal de control denominada "MemWrite" se activa para que la escritura sea posible.

II-A9. Escribir en un registro: Para escribir en un registro, la dirección proviene de la codificación (rd). Los datos a guardar pueden venir desde la unidad de extensión de signo o puede ser la señal llamada "Result" que puede ser datos provenientes de la ALU o de la memoria. Para escoger entre el inmediato y "Result" se usa la señal de control llamada "WDSrc" y en caso de que se use la segunda, para definir si se usan los datos provenientes de la ALU o de memoria se usa otra señal de control llamada "Mem2Reg". Asimismo, es necesario activar la señal RegWriteEn como fue discutido anteriormente.

II-A10. LUI: La LUI corresponde a una instrucción que toma un inmediato de 12 bits y lo convierte en uno de 32, pero en contraste con lo realizado normalmente por la unidad de extensión de signo, esta rellena con ceros a la derecha de tal inmediato. Se tomó la decisión de realizar esta operación dentro de la misma unidad de extensión de signo, esto con el fin de aprovechar que ya como entrada se tiene la instrucción completa y que el inmediato no se va a usar con otro fin. Para poder realizar esta operación la UC usa la señal "LUIOP".

II-B. Desarrollo en ISA

A continuación, se realiza una explicación sobre cada módulo necesario para la implementación de la microarquitectura en

Verilog. Se especifica su funcionalidad, sus entradas, salidas y la razón de porque se introdujo el módulo.

II-B1. FF D: El módulo corresponde a un flip-flop tipo D con accionamiento de flanco positivo que tiene como principal función almacenar la nueva dirección de memoria de PC para la ejecución de la próxima instrucción, ya sea definida por PC+4 o la dirección de memoria calculada para los casos de salto condicional o incondicional. La nueva dirección, se almacena esperando a que se termine de ejecutar la instrucción actual y cuando se dé un nuevo flanco positivo en el ciclo del reloj, actualiza la dirección que estuvo guardando para que pueda ser leída en el fetch.

Tiene tres entradas y una salida:

- **Entrada Clock:** El ciclo del reloj, necesario para actualizar la nueva dirección almacenada.
- **Entrada Reset:** La entrada del reset, en caso de que se ponga en alto, la nueva dirección se definirá por defecto como la dirección cero.
- **Entrada D:** Es la entrada de 32 bits, dónde almacena la nueva dirección de memoria a leer para la siguiente instrucción.
- **Salida Q:** Es la salida de 32 bits que se actualiza al valor que contenía la entrada D en cada ciclo de reloj.

La principal razón por la que se introdujo el módulo, es para evitar que exista un conflicto para leer la dirección de memoria de la instrucción que se ejecuta en el actual ciclo con la dirección de memoria de la instrucción que se ejecutará en el siguiente ciclo.

II-B2. Fetch: El módulo fetch tiene como principal función recuperar de la memoria, la decodificación para la nueva instrucción a ejecutar durante el ciclo, esto según la dirección de memoria que se haya definido anteriormente. El módulo lee todo el archivo.txt que contiene la decodificación de instrucciones y brinda la decodificación de la instrucción que coincida con la dirección de memoria deseada.

Tiene dos entradas y una salida:

- **Entrada Clock:** El ciclo del reloj, necesario para que se actualice la lectura en memoria y brinde la decodificación de la nueva instrucción.
- **Entrada PC:** Es la entrada de 32 bits, que contiene la dirección de memoria que se quiere leer para obtener la decodificación de la nueva instrucción.
- **Salida data_out:** Es la salida de 32 bits que contiene la decodificación de la nueva instrucción.

La principal razón por la que se introdujo el módulo es para la lectura de la compilación de las instrucciones de ensamblador necesarias para ejecutar el código c propuesto.

II-B3. PCPlus4: El módulo PCPlus4 es reduce a ser un simple sumador, que tiene como propósito calcular la siguiente dirección de memoria de PC, es decir, al valor actual de PC le suma el número cuatro como inmediato para obtener PC+4, que contiene la decodificación de la nueva instrucción siempre y cuando la instrucción actual no sea un salto.

Tiene una entrada y una salida

- **Entrada PC:** Es la entrada de 32 bits, que contiene la dirección de memoria actual que se quiere leer para calcular PC+4 de la siguiente instrucción a ejecutar del próximo ciclo.
- **Salida PC+4:** Es la salida de 32 bits, que contiene la dirección de memoria calculada PC+4, para la siguiente instrucción a ejecutar del próximo ciclo.

Este módulo se introdujo ante la necesidad de realizar un cálculo, la nueva dirección PC+4 sin generar conflictos con la ALU que se podría necesitar para otras operaciones de la instrucción que se está ejecutando.

II-B4. UC: El módulo UC, es la unidad de control de la microarquitectura, tiene como principal función colocar valores lógicos a las diferentes banderas que habilitan, deshabilitan o definen selectores para las funciones de los módulos; esto según la decodificación de la instrucción que se desea ejecutar en dicho ciclo.

Tiene 2 entradas y 11 salidas:

- **Entrada Selector:** Es la entrada de 5 bits, que contiene los bits [6:2] de la decodificación, estos bits corresponden al opcode que identifica cual es el tipo de instrucción (I,S,J,B y U) que se desea ejecutar.
- **Entrada Negative:** Es una bandera que proviene de la ALU, la cual indica que la operación que se calculó en la ALU dio como resultado un número negativo. Es necesario para definir si se ejecuta o no el salto condicional.
- **Salida Branch:** En caso de que la instrucción sea tipo B, se pone en alto y para cualquier otro caso es bajo.
- **Salida Jump:** En caso de que la instrucción sea tipo J, se pone en alto y para cualquier otro caso es bajo.
- **Salida PCSrc:** La bandera que indica al multiplexor si debe seleccionar el nuevo valor de PC del módulo PCPlus4 al ser bajo o del módulo NewPC al ser alto. Para saltos condicionales, se debe cumplir una AND entre la bandera Negative y la bandera Branch. Para saltos incondicionales solamente depende de la bandera Jump.
- **Salida ImmSel:** La salida de dos bits que funciona como selector para definirle al módulo al módulo Extend cuanto debe expandir el inmediato según sea necesario para cada instrucción.
- **Salida LUIOP:** En caso de que la instrucción sea tipo U, se pone en alto para indicarlo al módulo Extend y para cualquier otro caso es bajo.
- **Salida WDSrc:** La bandera que indica al multiplexor si debe seleccionar el valor del inmediato extendido del módulo Extend al ser alto o al ser bajo del valor recuperado de memoria o calculado de la ALU.
- **Salida ALUSrc:** La bandera que indica al multiplexor si debe seleccionar el valor del inmediato extendido del módulo Extend al ser alto o del valor del segundo registro, definido por el módulo Banco de registros.
- **Salida ALUOP:** La bandera que indica a la ALU, según lo que se necesite en la instrucción, si debe realizar una operación de suma en caso de ser bajo o una operación

de resta en caso de ser alto.
- **Salida Mem2Reg:** La bandera que indica al multiplexor si debe seleccionar el valor calculado por el módulo ALU al ser alto o al ser bajo del valor recuperado de memoria por el módulo Data Memory.
- **Salida MemWrite:** La bandera que indica al módulo Data Memory si debe o no escribir a memoria.
- **Salida RegWriteEn:** La bandera que indica al módulo Banco de registro si debe o no escribir a un registro.

La razón por la que se introdujo el módulo es para que toda la microarquitectura pueda ser aprovechada para cualquier tipo de instrucción y que no existan conflictos entre los módulos según lo que se requiera.

II-B5. Banco de registros: El módulo de banco de registros tiene dos principales funciones: leer y escribir registros. Se encarga, mediante cases, de recibir desde la decodificación la cadena que indica de cuál de los registros debe leer los datos que contenga para las instrucciones que lo requieran, pueden funcionar para contener temporalmente direcciones de memoria o inmediatos que se deseen utilizar. Por otro lado, puede habilitarse para definirle a un determinado registro lo que se haya calculado en la ALU o lo que se haya recuperado de la memoria.

Tiene 7 entradas y 2 salidas:
- **Entrada Clock:** El ciclo del reloj, necesario para actualizar al nuevo registro que se desea escribir.
- **Entrada RegWriteEn:** Es la bandera definida por el módulo UC que habilita o deshabilita si se desea escribir a un determinado registro.
- **Entrada read_r1:** Es la entrada de 5 bits que proviene de la decodificación de la instrucción [19:15] que indica cuál es el número del primer registro que se desea leer.
- **Entrada read_r2:** Es la entrada de 5 bits que proviene de la decodificación de la instrucción [24:20] que indica cuál es el número del segundo registro que se desea leer.
- **Entrada rd:** Es la entrada de 5 bits que proviene de la decodificación de la instrucción [11:7] que indica cuál es el número del registro al que se le desea escribir.
- **Entrada data:** Es la entrada de 32 bits que contiene los datos que sea desean escribir en el registro dado por rd. Los datos pueden ser definidos por la memoria, por lo calculado en la ALU o por la extensión del inmediato.
- **Entrada Reset:** La entrada del reset, si se pone en alto inicializarán todos los datos de los registros en sus valores por defecto, es decir, cero(exceptuando para el x2 que es 100).
- **Salida data_r1:** La salida de 32 bits de los datos temporales que almacenaba el primer registro que se definió por leer.
- **Salida data_r2:** La salida de 32 bits de los datos temporales que almacenaba el segundo registro que se definió por leer.

Este módulo se introdujo para poder tener espacios de almacenamiento temporales de datos que se requieran durante la ejecución de las instrucciones y no verse obligado a escribir en memoria datos intermedios de la operación.

II-B6. Extend: El módulo Extend cumple como funcionalidad, extender el valor del inmediatos datos por la decodificación y definirle un tamaño de 32 bits y prepararlo para su utilización en las diferentes instrucciones que requieren de un inmediato. La extensión depende de la decodificación de la instrucción, ya que la cantidad de bits con la que cuenta el inmediato inicialmente es diferente para cada una. También se encarga del acomodo del inmediato en la pseudo operación Lui.

Tiene 3 entradas y 1 salida:
- **Entrada instr:** Es la entrada de 32 bits que contiene toda la decodificación de la instrucción que se desea ejecutar.
- **Entrada LuiOP:** Es la bandera definida por el módulo UC que habilita o deshabilita si se desea realizar el acomodo de la operación Lui.
- **Entrada LuiOP:** Es el selector de 2 bits definida por el módulo UC que define cuáles bits de la cadena de la instrucción se necesitan leer y extender para obtener el inmediato.
- **Salida immext:** La salida de 32 bits del inmediato de la decodificación requerido ya extendido.

La razón por la que se introdujo el módulo es para evitar conflictos y que se puedan soportarse todas las instrucciones que requieren de inmediatos y tienen decodificaciones diferentes y con sus inmediatos definidos en variados bits.

II-B7. NewPC: El módulo NewPC es reduce a ser un simple sumador, que tiene como propósito calcular la siguiente dirección de memoria de PC para instrucciones tipo B y J, es decir, al valor actual de PC le suma el valor del inmediato extendido que contiene la decodificación.

Tiene dos entradas y una salida:
- **Entrada PC:** Es la entrada de 32 bits, que contiene la dirección de memoria actual que se quiere leer para calcular la nueva dirección de memoria del salto.
- **Entrada immExt:** Es la entrada de 32 bits, que contiene al inmediato extendido para sumarlo al PC y definir la nueva dirección.
- **Salida newpc:** Es la salida de 32 bits, que contiene la dirección de memoria calculada para el salto.

Este módulo se introdujo ante la necesidad de realizar un cálculo, la dirección de memoria que se requiera en salto; sin generar conflictos con la ALU que se podría necesitar para otras operaciones de la instrucción que se está ejecutando. Específicamente para la instrucción de salto condicional, es totalmente requerido puesto que la ALU se encarga de realizar el cálculo que determina si el salto se debe hacer o no, mientras que este módulo debe calcular la nueva dirección en caso de que sí se haga.

II-B8. ALU: El módulo ALU, como su nombre lo indica, se encarga de realizar las operaciones aritmético-lógicas necesarias para las instrucciones. En este caso, solamente se necesita realizar sumas y restas. Adicionalmente, puede generar una bandera en caso de que su resultado en resta dé negativo.

Tiene tres entradas y dos salidas:

- **Entrada data_r1:** Es la entrada de 32 bits, que contiene los datos temporales que almacenaba el primer registro.
- **Entrada data_r1:** Es la entrada de 32 bits, que contiene dos posibles valores: los datos temporales que almacenaba el segundo registro o el inmediato extendido.
- **Entrada ALUOP:** Es la bandera que indica a la ALU si debe realizar una operación de suma o de resta, proviene del módulo de la UC.
- **Salida ALUResult:** Es la salida de 32 bits, que contiene el resultado del cálculo realizado.
- **Salida Negative:** Bandera que indica que la operación que se calculó dio como resultado un número negativo.

Este módulo se introdujo porque es necesario realizar operaciones que no tengan relación directamente con la dirección de PC, sino que se necesita para operaciones intermedias en las diferentes instrucciones.

II-B9. Data Memory: Este módulo consiste en la memoria de datos la cual se se utiliza para almacenar instrucciones, datos y direcciones durante la ejecución de un programa. La memoria consiste en un array de datos de acceso aleatorio, en cualquier momento se puede acceder a cualquier dirección de memoria. Puede tanto leer como escribir a memoria.

Tiene 5 entradas y 1 salida:

- **Entrada data:** Es la entrada de 32 bits, que contiene los datos temporales que almacenaba el segundo registro y lo que se desean escribir en memoria.
- **Entrada addr:** Es la entrada de 32 bits, que contiene la dirección de memoria a la que se desea leer o escribir.
- **Entrada WE:** Es la bandera que indica al módulo si la escritura a memoria se encuentra habilitada o no.
- **Entrada Reset:** La entrada del reset, si se pone en alto se reiniciarán todos los datos almacenados en la memoria, es decir, se pondrán en cero.
- **Entrada Clock:** El ciclo del reloj, necesario para actualizar a la nueva dirección de memoria en la que se deaea escibrir.
- **Salida Q:** Es la salida de 32 bits, que contiene los datos leídos en la dirección de memoria indicada.

II-B10. Mux: El módulo multiplexor, simplemente se encarga de dar en su salida los datos que se encuentran conectados a la entrada 0 o a la entrada 1, según lo que defina su selector lógico. Este módulo se utiliza para seleccionar diferentes valores de entrada o de salida en otros módulos.

Tiene 3 entradas y 1 salida:

- **Entrada data0:** Es la entrada de 32 bits, que contiene los datos conectados a la entrada lógica 0.
- **Entrada data1:** Es la entrada de 32 bits, que contiene los datos conectados a la entrada lógica 1.
- **Entrada sel:** Es el selector que indica al módulo si debe de definir en la salida los datos que se encuentran conectados a la entrada 0 o a la entrada 1.
- **Salida out:** Es la salida de 32 bits definida por lo el valor del selector y los datos de la entrada 0 o 1.

III. ANÁLISIS DEL DISEÑO

El objetivo de este proyecto era que la microarquitectura propuesta fuera capas de soportar y simular un código de lenguaje C, a partir de su compilación en el ISA de RISCV. En la figura 2 se muestra el código de C que se debía soportar para la solución.

```
#include <stdio.h>

int main(){
  int *k = 0xABCD;
  int a = 2;
  int b = 0;

  for (int i=0; i<a; i++) {
    b = b+3;
  }
}
```

Figura 2. Código en C que se va a implementar en la microarquitectura.

En este caso como se puede observar, el código crea y utiliza solamente 4 variables (un puntero de tipo entero k y 3 valores enteros: a, b e i). Siguiendo la lógica y la secuencia del código al final las variables deben tener los siguientes valores:

- a = 2
- b = 3
- i = 2
- k = 0xABCD

Ahora, para la microarquitectura propuesta en esta solución, en primer lugar se analizó manualmente el datapath de cada tipo de instrucción. En las figuras 6, 7, 8, 9 y 10 de los anexos se muestra cómo en el diagrama final se soporta cada tipo de instrucción sin ningún problema.

Seguidamente, cuando se implementó la solución en Verilog, se notó que las direcciones de memoria a las que se escribían las operaciones ”SW”, eran demasiado altas y sobrepasaban la capacidad de la memoria diseñada. La solución a esto fue asignarle un valor inicial de 100 (decimal) al registro ”sp”, el cual, según la lista de las instrucciones en ensamblador se vio que era el que podía alterar las direcciones de memoria. Una vez hecho esto, se resolvieron las operaciones que escriben en memoria a mano, para así poder comparar los resultados con los obtenidos en la simulación de Verilog y el reporte de memoria final dado por el programa (archivo dump de memoria.txt).

En la figura 11 de los anexos se muestra la tabla en Excel con los resultados manuales, como se puede notar, el código inicia creando un puntero k, al cual le asigna el valor de ”0xABCD”. En la compilación esta línea de código C se traduce en tres instrucciones de ensamblador, las primeras dos se encargan de generar el valor del inmediato ”0xABCD”, mientras que el ”SW”debe guardar dicho inmediato en un espacio de la memoria RAM designado por la dirección 0x48. Esto se puede comprobar con la simulación de Verilog, en la figura 3 se muestra una captura de la simulación, donde el valor de PC indica que se está ejecutando la instrucción 14 y si se compara el valor del data_out con el hexadecimal de la instrucción de la tabla, se puede comprobar que efectivamente se trata de la operación ”SW.ª la que se hace referencia.

Así mismo, en la figura 3, se evidencia que el ”we”se encuentra en 1 (lo que indica que se va a escribir en memoria) y el valor del ”addr” (address) sería 48 (HEX). Por último, como la escritura es síncrona con el reloj, el valor del inmediato 0xABCD se va a escribir en memoria hasta que se de el siguiente flanco de reloj, tal como se aprecia en la simulación.

De manera similar se pueden verificar las instrucciones 1C, 34 y 40 con las figuras 12, 13 y 14 de los anexos.

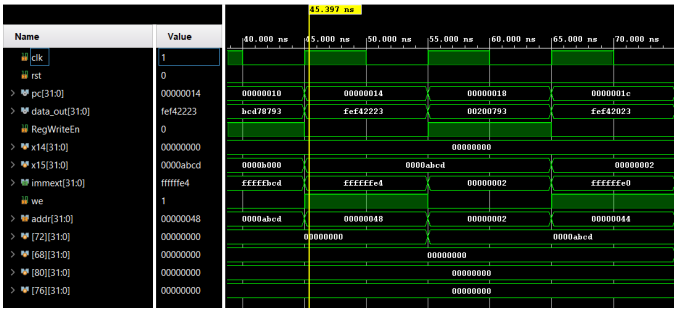

Figura 3. Simulación en Verilog de la microarquitectura propuesta ejecutando la instrucción 14.

Adicionalmente, al final de la ejecución de la simulación se debía generar un archivo de texto con todas las direcciones de la memoria y su respectivo valor. En la figura 14 de los anexos se muestra una parte de este documento en el cual se evidencian los espacios de memoria que son diferentes de cero. Como se puede notar, las direcciones de memorias y sus valores coinciden tanto con la solución manual (figura 11) como con los valores de la simulación (figuras 3, 12, 13 y 14).

Por otro lado, un aspecto importante de mencionar es que durante la implementación se notó que la microarquitectura leía las instrucciones de acuerdo con el valor de PC, como este inicia en cero y va aumentando de 4 en 4, se estaban saltando 3 instrucciones del archivo de texto. Como solución a este problema, se insertaron tres líneas de 32 bits con valor de 0. No obstante, este cambio tiene implicaciones en toda la microarquitectura.

Cada vez que se da una instrucción nueva, el bloque del Instruction Memory lee todo el archivo de texto que contiene el set de instrucciones, las almacena en una memoria y de acuerdo con el valor de PC obtiene las instrucciones. Entonces, una implicación de colocar estos espacios con ceros es que se va a desperdiciar un gran porcentaje del Instruction Memory, pues por cada instrucción habrán 3 líneas de ceros; es decir, se triplica el tamaño necesario para la memoria. La segunda implicación está relacionada con el tiempo de ejecución, ya que al ser una memoria más grande tarda más tiempo en obtener el dato deseado. Esto a su vez aumenta el tiempo del ciclo de reloj y reduce la eficiencia de la microarquitectura.

Por último, een la simulación de Verilog se ejecutan en total 33 instrucciones, lo cual indica que la microarquitectura requiere de 33 ciclos de reloj para completar todas las instrucciones. Así mismo el clock debe durar lo que tarda en ejecutarse la instrucción más larga, que en este caso es la instrucción ”LW”. En el caso de la solución realizada se utilizó un ciclo de reloj de 5 ns, este ciclo fue suficiente para que todas las instrucciones pudieran ejecutarse. No obstante un punto de mejora sería buscar el momento en el que el valor de la memoria llega al multiplexor controlado por la señal Mem2Reg, ya que es cuando dato estaría listo para escribirlo en el banco de registros.

IV. CONCLUSIONES

1. Por medio de la compilación de código para RISCV es posible generar un objdump que permita relacionar un código C con las líneas en ensamblador, el cual determinará las instrucciones del set de enteros del ISA RISCV: RISCV32I necesarias para la ejecución del mismo.
2. Para determinar la duración del clock se debe tomar el tiempo que dura la instrucción LW en completarse, puesto que es el tipo de instrucción que más tarda y todas las demás pueden acomodarse a dicho tiempo.
3. Con cada actualización en el valor de PC se carga el archivo en el que se realiza el dump en la memoria y se busca la próxima instrucción.

V. RECOMENDACIONES

1. Es posible ahorrar gran cantidad de memoria si en el archivo al que realiza el dump no estuviera cargado de ceros.
2. A la hora de implementar los módulos en el lenguaje de descripción de hardware Verilog se recomienda realizarlo de forma gradual por partes y realizando pruebas indivuales.
3. En caso de requerir que la microarquitectura soprte códigos C diferentes al facilitado en este proyecto es necesario analizarlo detalladamente y asegurarse de que no incluya ninguna instrucción fuera de las permitidas.

VI. CITAS Y/O REFERENCIAS

REFERENCIAS

[1] L. Hunter, N. Kazakov, K. Ostrolenk y D. Hood. .Adding RISC-V Vector Cryptography Extension support to QEMU”. Codethink.
[2] S. Harris & D. Harris. *Digital design and computer architecture*. Morgan Kaufmann, 2015.
[3] S. Harris & D. Harris. *Digital design and computer architecture: RISC-V Edition*. Morgan Kaufmann, 2021.

VII. ANEXOS

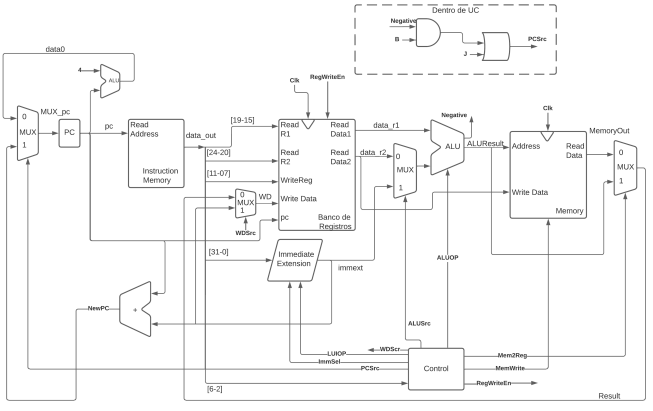

Figura 4. Diseño microarquitectura.

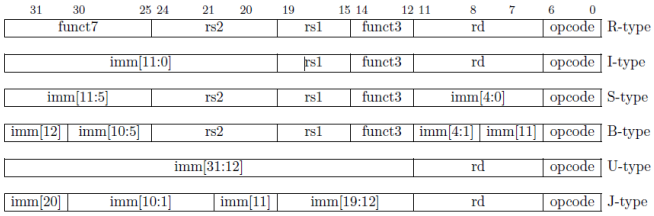

Figura 5. Tipos de codificación en RISC-V.

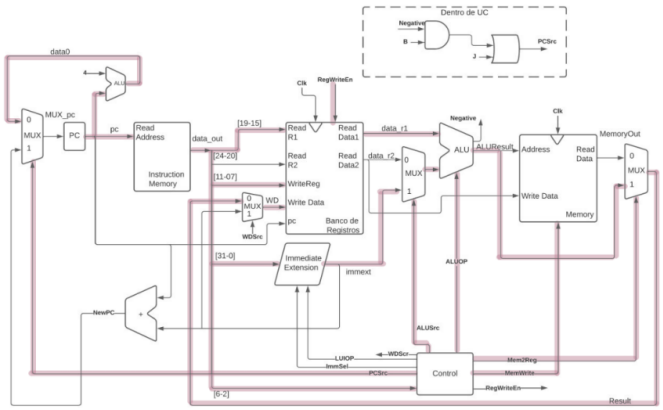

Figura 6. Datapath para las instrucciones de tipo I.

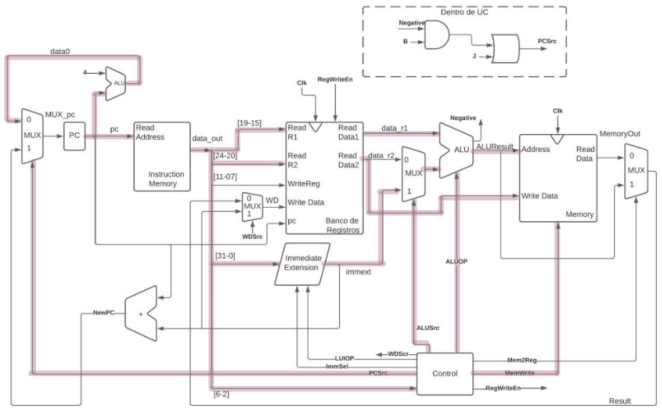

Figura 7. Datapath para las instrucciones de tipo S.

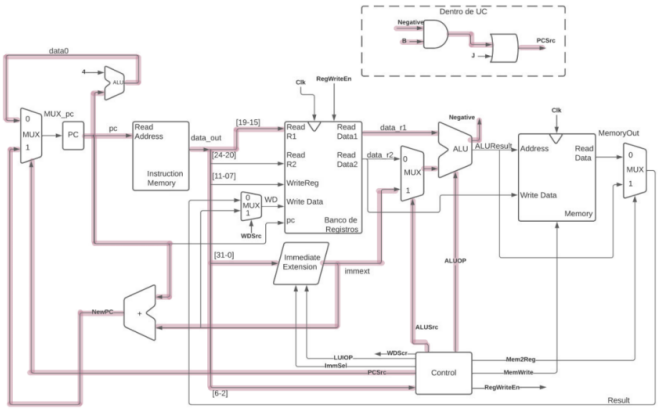

Figura 8. Datapath para las instrucciones de tipo B.

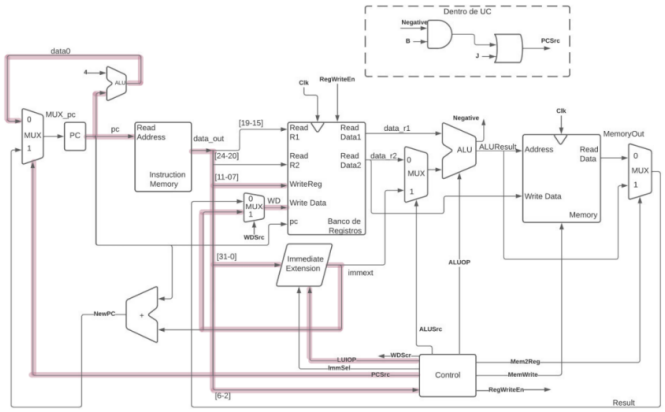

Figura 9. Datapath para las instrucciones de tipo U.

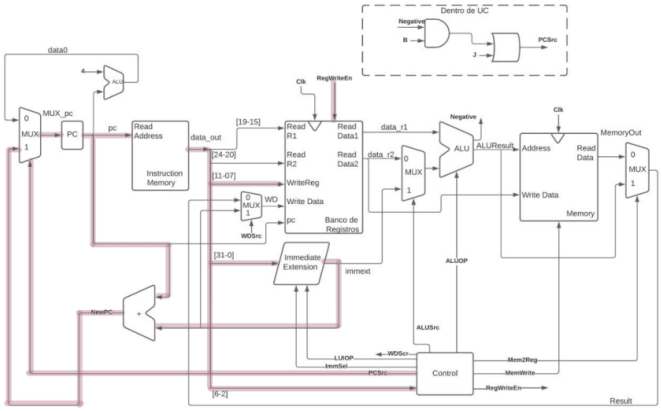

Figura 10. Datapath para las instrucciones de tipo J.

Número de instrucción	Instrucción en hexadecimal	Instrucción en ensamblador	Address en decimal	Significado en la memoria
--	int main(){		sp = 100 (valor inicial)	
0	fe010113	add sp, sp, -32		
4	00812e23	sw s0, 28(sp)	128	**M[0x80] = 0 (s0)**
8	02010413	add s0, sp, 32	s0 = 100	s0 = 100
--	int *k = 0xABCD;			
c	0000b7b7	lui a5, 0xb		
10	bcd78793	add a5, a5, -1075		
14	fef42223	sw a5, -28(s0)	72	**M[0x48] = 0xABCD (a5)**
--	int a = 2;			
18	00200793	li a5, 2		
1c	fef42023	sw a5, -32(s0)	68	**M[0x44] = 0x2 (a5)**
--	int b = 0;			
20	fe042623	sw zero, -20(s0)	80	**M[0x50] = 0x0**
--	for (int i=0; i<a; i++) {			
24	fe042423	sw zero, -24(s0)	76	**M[0x4C] = 0x0**
28	01c0006f	j 44		
--	b = b + 3;			
2c	fec42783	lw a5, -20(s0)		
30	00378793	add a5, a5, 3		
34	fef42623	sw a5, -20(s0)	80	**M[0x50] = 0x6 (a5)**
--	for (int i=0; i<a; i++) {			
38	fe842783	lw a5, -24(s0)		
3c	00178793	add a5, a5, 1		
40	fef42423	sw a5, -24(s0)	76	**M[0x4C] = 0x2 (a5)**
44	fe842703	lw a4, -24(s0)		
48	fe042783	lw a5, -32(s0)		
4c	fef740e3	blt a4, a5, 2c		
50	00000793	li a5, 0		

Figura 11. Comprobación manual de los resultados que escriben en memoria.

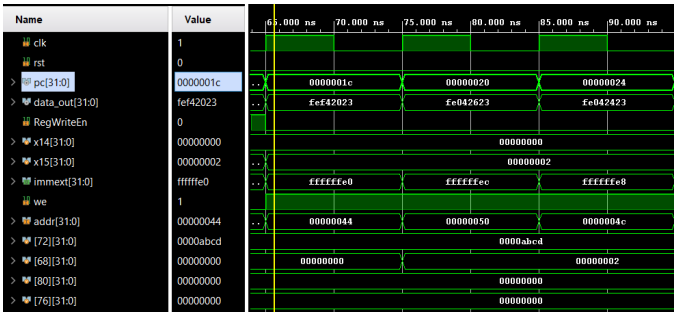

Figura 12. Simulación en Verilog de la microarquitectura propuesta ejecutando la instrucción 1C.

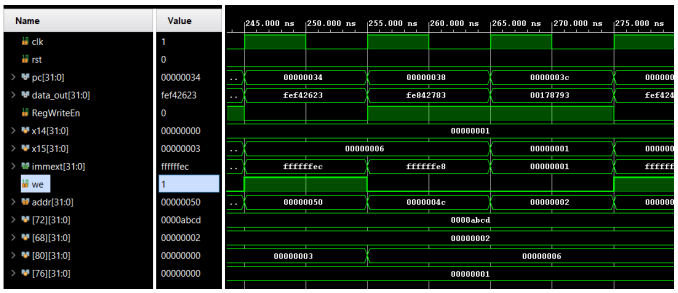

Figura 13. Simulación en Verilog de la microarquitectura propuesta ejecutando la instrucción 34.

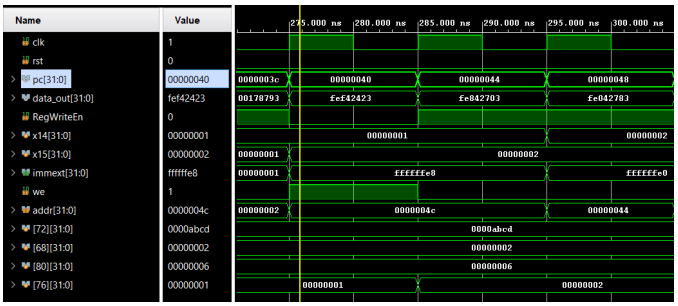

Figura 14. Simulación en Verilog de la microarquitectura propuesta ejecutando la instrucción 40.

Dirección	Valor
0x00000044	0x00000002
0x00000048	0x0000abcd
0x0000004c	0x00000002
0x00000050	0x00000006

Figura 15. Dump de memoria realizado al final de la simulación en Verilog.