Implementación de una microarquitectura para el ISA RISCV

Andres David Apuy Garro *, Diego Brenes Poveda*, Fátima Cárdenas Obando *,
Brandy Juliana Jiménez Delgado *, Bruno Ramses Mora Villalobos *, Ana Victoria Rojas Lazo *

*Escuela de Ingeniería Mecatrónica, Instituto Tecnológico de Costa Rica (ITCR), 30101 Cartago, Costa Rica,
andresapuyg@estudiantec.cr, diegobrenes21@estudiantec.cr, fatimaco13@estudiantec.cr,
brandyjd03@estudiantec.cr, brunosft00@estudiantec.cr, vrojasl@estudiantec.cr

***Resumen*—Este trabajo de investigación aborda el desarrollo, diseño e implementación de una microarquitectura uniciclo para un conjunto reducido de instrucciones del ISA RISC-V32I. El proceso de diseño involucró la creación de múltiples módulos, incluyendo la Unidad Aritmético-Lógica (ALU), el contador de programas, el multiplexor (MUX), la memoria de instrucciones, la memoria de datos, la Unidad de control, el archivo de registros y la extensión de signo. La implementación de esta microarquitectura se llevó a cabo utilizando el lenguaje de descripción de hardware Verilog, lo que permitió una representación precisa y eficiente de la arquitectura del procesador.**

***Palabras Clave*—ALU, control, multiplexor, PC, uniciclo.**

***Resumen*—This research work addresses the development, design and implementation of a multicycle microarchitecture for a reduced set of ISA RISC-V32I instructions. The design process involved the creation of multiple modules, including the Arithmetic-Logic Unit (ALU), program counter, multiplexer (MUX), instruction memory, data memory, Control Unit, register file and sign extension. The implementation of this microarchitecture was carried out using the Verilog hardware description language, which allowed an accurate and efficient representation of the processor architecture.**

***Palabras Clave*—ALU, control, multiplexer, PC, unicycle.**

I. Introducción y Generalidades

Para el desarrollo del proyecto se planteó el problema de dar soporte a un conjunto de instrucciones las cuales fueron extraídas de un código en c de referencia, el cual con ayuda del crosscompile para el ISA RISC-V se descompuso en instrucciones en ensamblador para el ISA mencionado, a partir de las cuales se observó el tipo de instrucciones que debía soportar la arquitectura implementada para su desarrollo, por lo cual con las características necesarias se procedió con el diseño del procesador en el sistema de Xilin Vivado para su construcción y simulación con el fin de verificar su funcionamiento.

Para el diseño de un procesador se necesita, en primer lugar, contar con la definición de la Arquitectura de Conjunto de Instrucciones o ISA (Instruction Set Architecture). La arquitectura de computadoras es la implementación de múltiples diseños que describen los métodos y reglas utilizados en un sistema informático, al inicio de una nueva era de la electrónica a finales de los años 50, cada diseño de computadora es único entre sí, los programas solo se crean y utilizan para una función específica. [1]

Los microprocesadores y microcontroladores suelen desarrollarse siguiendo dos arquitecturas informáticas principales: la arquitectura CISC (Computación con Conjunto de Instrucciones Complejo) y la arquitectura RISC (Computación con Conjunto de Instrucciones Reducido). La arquitectura CISC se enfoca en diseñar un conjunto de instrucciones (ISA) que mejora el rendimiento al ofrecer múltiples instrucciones, cada una con diferentes operandos y varios modos de direccionamiento. Sin embargo, esto provoca que las instrucciones tengan duraciones y tiempos de ejecución variables, lo que requiere una unidad de control más compleja y que ocupa una porción considerable del chip. En cambio, los procesadores RISC se caracterizan por utilizar un conjunto de instrucciones más reducido y simplificado en comparación con CISC. [1]

Una de las principales características de RISC-V radica en la filosofía que refleja su nombre. Un ISA RISC se enfoca en definir instrucciones orientadas a realizar tareas simples y de propósito general. En contraste, la arquitectura CISC (Complex Instruction Set Computing) establece instrucciones diseñadas para llevar a cabo operaciones más específicas. El término ”reducido” hace referencia a que cada instrucción en RISC ejecuta una carga de trabajo menor en comparación con las instrucciones de un conjunto CISC. Esta simplicidad permite que los procesadores RISC ejecuten más instrucciones por ciclo de reloj que los CISC. Es importante señalar que algunas características comunes entre los diferentes ISAs RISC se deben más a similitudes entre los conjuntos de instrucciones que a la filosofía de diseño en sí misma [1]

Actualmente la ISA RISC-V continúa en desarrollo con la participación de ingenieros de compañías privadas en conjunto con académicos de diversas instituciones. Según se muestra en las especificaciones oficiales, RISC-V es en realidad un conjunto de varias ISA. A la más sencilla de éstas

se le denomina “base entera RV32I” y se emplea para procesar datos enteros de 32 bits. En la versión de especificaciones 20191213 esta ISA cuenta con 40 instrucciones incluyendo operaciones de memoria, comparaciones, ramificaciones, operaciones aritméticas y lógicas, entre otras [2]. Podemos considerar esta base entera como el conjunto mínimo de instrucciones necesarias para el diseño de procesadores o microcontroladores sencillos.

II. Descripción del Problema

Para el desarrollo de un procesador basado en el conjunto de instrucciones RISC-V, se decidió utilizar una arquitectura uniciclo. Dicha elección se basó en la simplicidad que presenta este diseño, ya que, en una arquitectura uniciclo cada instrucción completa su ciclo de ejecución en un sólo ciclo de reloj.

Asimismo, a partir del código C suministrado (figura 1), se realizó una compilación para obtener el conjunto de instrucciones en ensamblador a través de RISC-V (figura 2), con el propósito de diseñar la arquitectura uniciclo. Al realizar esto, se determinó que la arquitectura debía soportar instrucciones tipo R, I, J, U y S, además de algunas pseudointrucciones extendidas de estos tipos.

```c
#include <stdio.h>

int main(){
  int *r = 0xC7DF;
  char a = 'c';
  int b = 33;

  for (int i=0; i<2; i++) {
    if (i==0){
      a = 'b';
    }
    else{
      b = b<<1;
      b = b&0xF;
    }

  }

  // printf("Result is %d and string is %c\n", b, a);
}
```

Figura 1. Código C base para la solución.

```
Desensamblado de la sección .text:

00000000 <main>:
#include <stdio.h>

int main(){
   0:   fe010113                addi    sp,sp,-32
   4:   00812e23                sw      s0,28(sp)
   8:   02010413                addi    s0,sp,32
  int *r = 0xC7DF;
   c:   0000c7b7                lui     a5,0xc
  10:   7df78793                addi    a5,a5,2015 # c7df <.LASF13+0xc745>
  14:   fef42223                sw      a5,-28(s0)
  char a = 'c';
  18:   06300793                li      a5,99
  1c:   fef401a3                sb      a5,-29(s0)
  int b = 33;
  20:   02100793                li      a5,33
  24:   fef42623                sw      a5,-20(s0)

00000028 <.LBB2>:

  for (int i=0; i<2; i++) {
  28:   fe042423                sw      zero,-24(s0)
  2c:   03c0006f                j       68 <.L2>

00000030 <.L5>:
    if (i==0){
  30:   fe842783                lw      a5,-24(s0)
  34:   00079863                bnez    a5,44 <.L3>
      a = 'b';
  38:   06200793                li      a5,98
  3c:   fef401a3                sb      a5,-29(s0)
  40:   01c0006f                j       5c <.L4>

00000044 <.L3>:
    }
    else{
      b = b<<1;
  44:   fec42783                lw      a5,-20(s0)
  48:   00179793                slli    a5,a5,0x1
  4c:   fef42623                sw      a5,-20(s0)
      b = b&0xF;
  50:   fec42783                lw      a5,-20(s0)
  54:   00f7f793                andi    a5,a5,15
  58:   fef42623                sw      a5,-20(s0)

0000005c <.L4>:
  for (int i=0; i<2; i++) {
  5c:   fe842783                lw      a5,-24(s0)
```

Figura 2. Conjunto de instrucciones en ensamblador generadas a través de RISC-V.

Además, el diseño requiere de una Unidad de Control que genere señales específicas a la hora de ejecutar cada instrucción. Una vez desarrolladas y probadas las instrucciones, se generó un diagrama final de cómo debería ser la estructura de esta arquitectura, el cual se puede observar en el anexo 1.

III. Diseño de Solución

La **arquitectura uniciclo** ejecuta cada instrucción en un solo ciclo de reloj, permitiendo un flujo continuo de datos a través del procesador. Sin embargo, el ciclo de reloj debe ajustarse a la instrucción más lenta, lo cual puede limitar el rendimiento en algunas operaciones. Esta arquitectura se basa en una **Unidad de Control (UC)** que coordina la ejecución de instrucciones y en un **datapath** que maneja la transferencia y manipulación de datos.

III-A. Ruta de Datos Uniciclo

- **Datapath:** Maneja el flujo de datos y realiza operaciones aritméticas y lógicas mediante los siguientes elementos:
  - **ALU (Unidad Aritmético-Lógica):** Ejecuta operaciones matemáticas y lógicas definidas por la ISA. La ALU calcula la dirección de memoria para instrucciones de acceso a memoria y realiza operaciones con los registros de datos.
  - **Archivo de Registros (RegFile):** Este bloque contiene 32 registros de propósito general que se utilizan

para almacenar datos temporales, direcciones y resultados de la ALU. Se pueden acceder a los registros de manera flexible, lo que permite su uso en diversas instrucciones.
    - **Extensión de Signo:** Amplía los operandos inmediatos a 32 bits, conservando el signo, lo cual es crucial en operaciones de salto y acceso a memoria.
- **Unidad de Control (UC):** La UC genera las señales de control necesarias (RegWrite, MemWrite, ALUSrc, etc.) en función del tipo de instrucción. También gestiona condiciones de salto y banderas para asegurar que las instrucciones se ejecuten correctamente en un solo ciclo. Su lógica de control se basa en tablas de verdad y decodificación de iniopcode.

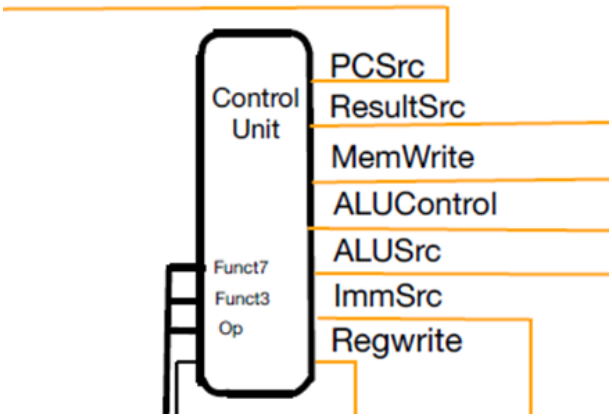

Figura 3. Unidad de Control.

- **Memoria de Instrucciones (InstructionROM):** Este bloque contiene el programa y proporciona la siguiente instrucción a ejecutar, se conecta al contador de programa (PC), que mantiene la dirección de la instrucción actual. En cada ciclo, el PC se incrementa (PC + 4) para la instrucción siguiente, o cambia en caso de una instrucción de salto.
- **Contador de programa (PC):** es un registro que guarda la dirección de la instrucción que se ejecutará en el siguiente ciclo, en el cual el PC se actualiza para dirigir a la próxima instrucción, lo que permite un flujo continuo en la ejecución del procesador.
- **PC_plus4:** este módulo se encarga de calcular la dirección de la siguiente instrucción sumando 4 al valor actual del PC. Esto es esencial en arquitecturas RISC, ya que cada instrucción tiene un tamaño fijo de 4 bytes y PC_plus4 asegura que el procesador avance a la siguiente instrucción de forma secuencial.
- **Memoria de Datos y Puertos (DataRAM & Ports):** Incluye tanto la memoria de datos como los puertos de entrada y salida, permitiendo el almacenamiento y recuperación de datos, así como la interacción con periféricos externos.
- **DataRAM:** Permite la lectura y escritura de datos en memoria en las instrucciones de carga y almacenamiento (por ejemplo, LDR y STR). La UC controla las operaciones de acceso mediante señales como MemWrite para escribir datos.
- **Control de saltos (Branch_Control):** Este módulo decide si se debe realizar un salto en función del tipo de instrucción y de los resultados de la ALU. Es importante mencionar que en esta instrucción entra en juego también PC_plus4 pues si la condición se cumple, Branch Control envía una señal para que el valor del PC se actualice con una dirección de salto calculada, en lugar de utilizar el valor de PC_plus4; esto ya que, si se realiza un salto, el procesador no debe continuar con la siguiente instrucción secuencial (PC + 4), sino que debe ir a la dirección especificada en la instrucción de salto. [3]
- **Decodificador de inmediatos (Immediate_Decoder):** Este módulo toma los bits de inmediato de una instrucción y los extiende a 32, conservando el signo cuando es necesario, permitiendo que las instrucciones que usan valores inmediatos puedan operar correctamente en la ALU y otros módulos.
- **Write_back:** Este módulo es responsable de escribir los resultados de la ALU o de la memoria en el RegisterFile al final del ciclo de ejecución de una instrucción para que el valor obtenido esté disponible para las siguientes instrucciones que lo necesiten.

III-B. Sección de Control:

Para el caso del control, este varía según la instrucción cargada para procesar, por ello toma el opcode de la instrucción y determina a partir de esto que señales de control debe aplicar dado que todo se realiza en un único ciclo, se pueden considerar caso según el tipo de instrucción:

1. **Tipo I:** Para este caso, se mantiene tanto MemRead es variable según la operación, ya que no siempre se lee esta, luego el Branch en 0, dado que no realiza salto, el ALUSrc y el RegWrite en 1, considerando que la fuente 2 es siempre el inmediato y siempre se escribe en un registro, el MemToRead también puede ser variable y el MemWrite es siempre 0 al no escribir nunca en memoria.
2. **Tipo S:**Para este tipo de instrucciones el MemToReg es 0 siempre, ya que no escribe en registro desde memoria nunca, el MemRead, RegWrite y Branch son siempre 0, dado que no se lee memoria, no se escribe en un registro ni se realiza un salto, y el ALUSrc y MemWrite son 1 siempre, dado que se trabaja con un inmediato y se va a escribir en memoria, en este caso el ALUOp da siempre el valor del selector de suma en la ALU.
3. **Tipo B:** Como el resultado de la comparación viene de la operación de la ALU, por lo cual ALUOp es el valor de la operación resta (SUB) y el Branch varía entre 0 y 1 si se toma o no el salto, las demás son siempre 0 en todo momento.

4. **Tipo J:** Como en estas no se utiliza la ALU, ALUOp y ALUSrc son irrelevantes, RegWrite, MemRead y MemWrite son 0, esto porque no se escribe en registro, no se lee de memoria ni se escribe en esta, únicamente el Branch es 1 indicando que se va a saltar, dado que es para saltos incondicionales.
5. **Tipo U:** En el caso de MemRead y MemWrite se mantienen en 0, dado que no se lee la memoria ni se escribe en memoria, el ALUSrc y RegWrite son 1, para utilizar el inmediato como fuente 2 y hacer la escritura en el registro destino, para el ALUOp se tiene el valor para la operación Passthrough. Por ende, la unidad maneja múltiples salidas de control y la única entrada es la instrucción de la memoria de instrucciones, de la cual el opcode, el funct3, y el funct7 los extrae para determinar las salidas necesarias para la ejecución correcta de la instrucción y así varía de instrucción a instrucción.

Por ende, la unidad maneja múltiples salidas de control y la única entrada es la instrucción de la memoria de instrucciones, de la cual el opcode, el funct3, y el funct7 los extrae para determinar las salidas necesarias para la ejecución correcta de la instrucción y así varía de instrucción a instrucción.

IV. Resultados

IV-A. Consumo de recursos y potencia

El principal consumo se encuentra en entradas y salidas, esto debido a la cantidad de datos que se mueve entre los diferentes módulos, considerando que son buses de datos en su mayoría. La potencia utilizada es de 65.479 W, que es sumamente alta, lo cual podría indicar que la sintetización puede estar realizada de forma que no es óptima para el rendimiento, o que este diseño no está correctamente optimizado para su funcionamiento, lo cual se contrasta con la temperatura de la unión que llega hasta los 125.0 °C, cuyo valor supera por mucho los valores de operación segura, siendo necesario reducirlo para no comprometer vida útil.

Por ende, es importante tomar estos parámetros como referencia para buscar un diseño óptimo que permita que la microarquitectura trabaje correctamente en los márgenes de potencia y temperatura; más que deseados, seguros para garantizar una larga vida útil.

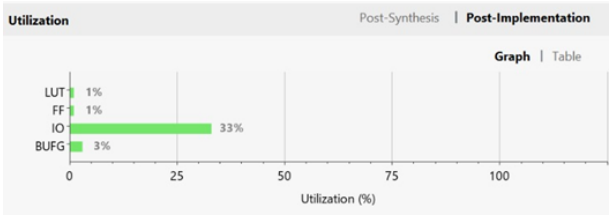

Figura 4. Utilización de recursos en Vivado.

IV-B. Resultados del Testbench

Para esto se realizan pruebas individuales en los diferentes módulos, con el objetivo de verificar que funcionen correctamente, por lo cual, se tratan como si recibieran alguna de las instrucciones soportadas, para así ver que las señales de salida de cada uno sea la deseada.

Por lo cual se generan testbench para cada módulo realizado, de esta manera es sencillo encontrar y depurar posibles errores que puedan existir, así se simulan las entradas que debería tener el módulo para una determinada instrucción y posteriormente se aanliza si el comportamiento en salidas es correcto para poder proceder con el desarrollo.

Después de realizar todos los módulos y las pruebas en cada uno de ellos, se realiza el testbench del main mediante la ejecución instrucción por instrucción de la memoria para así verificar que efectivamente la microarquitectura funciona sin problemas y soporta correctamente las instrucciones a ejecutar. Se presentan algunos de los resultados en la Figura 5.

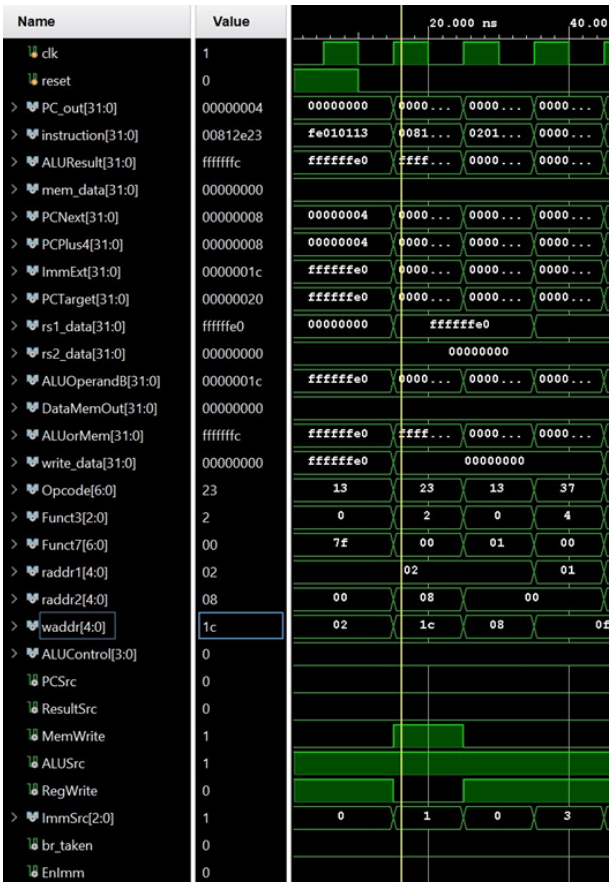

Figura 5. Testbench de Main de microarquitectura uniciclo en Vivado.

V. Conclusiones

- Con el uso de Vivado es posible el desarrollo de una microarquitectura, generando los diferentes módulos ne-

cesarios para el soporte de las instrucciones requeridas.

- Con la ayuda del crosscompile es posible obtener las instrucciones de ensamblador para el ISA deseado, de manera que se pueda diseñar con el ISA deseado sin problema alguno.
- Los testbech son la herramienta que permite probar el funcionamiento de los diferentes casos (tipo de instrucción) para verificar el correcto funcionamiento y realizar también la revisión del proyecto.
- La optimización del diseño es un punto crítico, ya que como se observó, en este caso no se alcanzó una buena optimización por lo cuál la microarquitectura sobrepasa parámetros fundamentales para asegurar su correcto funcionamiento como lo son la potencia y temperatura.

REFERENCIAS

[1] N. Pacheco, *Diseño, implementación y validación de un núcleo de procesador basado en el conjunto de instrucción RISC-V*, 2021.

[2] R.-V. Foundation, *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version: 20191213*, 2019. [Online]. Available: https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf.

[3] J. M. Cuadros, *Diseño monociclo del procesador*, Universidad Complutense de Madrid, España, 2023. [Online]. Available: https://www.fdi.ucm.es/profesor/mendias/FC2/FC2tema5-imprimible.pdf

VI. Anexos

VI-A. *Anexo 1*

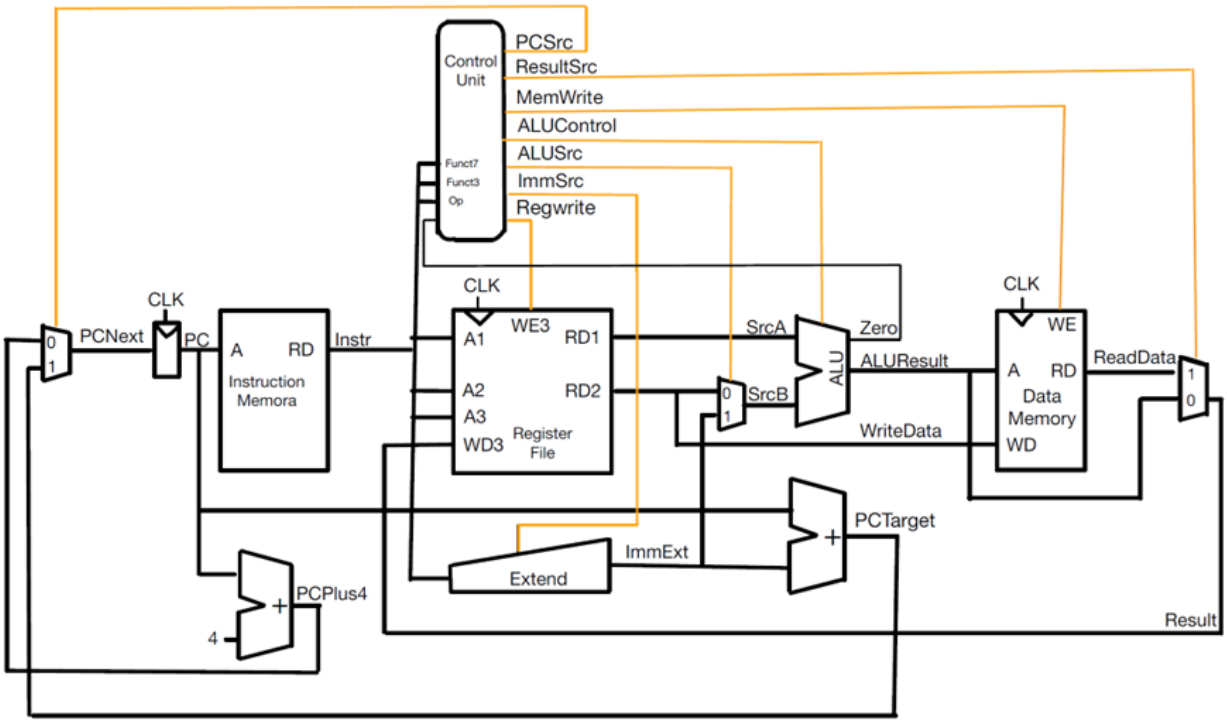

Figura 6. Diagrama de microarquitectura uniciclo implementado.

VI-B. *Anexo 2*

Cuadro I
Análisis de instrucciones utilizadas de base entera de RV32I.

	Instruction	Description	i_{31}: i_{25}	i_{24}: i_{20}	i_{19}: i_{15}	i_{14}: i_{12}	i_{11}: i_7	i_6: i_2	bus inPC	bus inRF	bus Addr	Other	mux A	mux B	Mux inPC	Mux inRF
LOAD	LW	Load Word	[11:5]	[4:0]	rs1	010	rd	00000	PC+4	L	D1+im	load	D1	im	PC4	L
STORE	SB	Store Byte	[11:5]	rs2	rs1	000	[4:0]	01000	PC+4	-	D1+im	store	D1	im	PC4	-
	SW	Store Word	[11:5]	rs2	rs1	010	[4:0]	01000	PC+4	-	D1+im	store	D1	im	PC4	-
SHIFT	SLLI	Shift Left Log. Imm	0000000	shamt	rs1	001	rd	00100	PC+4	ShiftimD1	-	-	D1	im	PC4	aluB
ARITHMETIC	ADDI	ADD Immediate	[11:5]	[4:0]	rs1	000	rd	00100	PC+4	D1+im	-	-	D1	im	PC4	aluAdSb
LOGIC	ANDI	AND Immediate	[11:5]	[4:0]	rs1	111	rd	00100	PC+4	D1&im	-	-	D1	im	PC4	aluAND
Up. Imm	LUI	Load Upper Imm	[31:25]	[24:20]	[19:15]	[14:12]	rd	01101	PC+4	im	-	-	-	-	PC4	im
BRANCH	BGE	Branch if >=	[12|10:5]	rs2	rs1	101	[12|10:5]	11000	PC+4 PC+im	-	-	D1<D2 ?	PC	im	PC4/ aluAdSb	-

Link GitHub (Diagramas completos y códigos en Verilog):

Bitácora