INSTITUTO TECNOLÓGICO DE COSTA RICA

ÁREA ACADEMICA DE INGENIERÍA
EN COMPUTADORES

CE-4302 ARQUITECTURA DE COMPUATADORES II

Diseño de un procesador Vectorial

Creado por :

Esteban Alonso Sanabria Villalobos 2015070913

I Semestre, 2019

***Resumen*—El presente documento tiene el objetivo de informar de manera detallada al lector sobre el diseño e implementación de un ISA y su micro-arquitectura para el procesamiento y encriptación de imágenes por medio de la utilización del concepto de paralelismo a nivel de datos, específicamente en procesadores SIMD del tipo vectorial. De tal forma que lo que a continuación se presenta las especificaciones del diseño, el ISA elaborado con sus diferentes instrucciones, tipos de direccionamientos y componentes de hardware desarrollados para el manejo e implementación de dicha arquitectura. Así mismo, se explicara la construcción y funcionalidad del software desarrollado para la simulación y comprobación del sistema.**

***Palabras Clave*—Benchmark, Desencriptación, Diseño, Encriptación, Imágenes, ISA, Micro-arquitectura, Paralelismo de datos, Procesador, SIMD, Vectorial**

I. Introduction

El presente proyecto tiene como objetivo que mediante la aplicación de los conceptos de paralelismo a nivel de datos, específicamente procesadores SIMD del tipo vectorial, el desarrollador sea capaz de diseñar e implementar en software una arquitectura vectorial propia para el tratamiento de imágenes, en algoritmos de encriptación, como una posible aplicación de seguridad informática. Adicionalmente, pretende ejercitar en el desarrollador los atributos de diseño en ingeniería y habilidades de comunicación.

Primeramente, recalcar que el paralelismo a nivel de datos surge a partir de la taxonomía de Flynn, en la cual se pueden clasificar los procesadores basados en los criterios de la cantidad de instrucciones o datos procesados. La categoría SIMD es parte de la clasificación denotada por Flynn y hace referencia a un sistema donde un único conjunto de instrucciones (programa) se ejecuta sobre múltiples conjuntos de datos, es decir a un conjunto de datos se le aplica una misma operación en un solo ciclo de reloj ahorrando múltiples ciclos de ejecución. Por ejemplo, una sola instrucción SIMD suma 64 números, el hardware envía 64 datos a 64 ALUs para obtener 64 sumas en el mismo ciclo de reloj. Este tipo de procesamiento tiene aplicaciones en el mundo del procesamiento de imágenes, audio, gráfico y aceleración de computación genérica.

Seguidamente, el paralelismo a nivel de datos ha tenido históricamente un gran campo de aplicación. Desde los años 70's, el diseño e implementación de arquitecturas vectoriales ha tenido un desarrollo continuo, siendo los procesadores vectoriales la referencia para otros tipos de arquitecturas que utilizan el concepto de Single-Instruction Multiple-Data (SIMD), en una gran cantidad de aplicaciones. El desarrollo de arquitecturas heterogéneas, en las que se combinan diferentes tipos de paralelismo, entre ellos el paralelismo a nivel de datos, ha tenido un papel fundamental en los sistemas modernos. Los dispositivos móviles, por ejemplo, hacen uso de arquitecturas SIMD para favorecer el desempeño en ejecución de tareas relacionadas a multimedia, en las que el procesamiento paralelo es fundamental.

A demás, tiene como objetivo desarrollar la capacidad de diseñar soluciones a problemas complejos de ingeniería, con final abierto y diseñar sistemas, componentes o procesos que cumplan con necesidades especificas, considerando la salud pública, seguridad, estándares pertinentes, así como los aspectos culturales, sociales y económicos.

II. Listado de requerimientos del sistema

A continuación, se presenta a manera de listado el conjunto de requerimientos capturados e identificados, necesarios para la construcción e implementación del proyecto planteado y su correcta completitud:

- Diseño propio del set de instrucciones (ISA):
  - Número de instrucción.
  - Tipo de instrucción.
  - Formato de las operaciones.
  - Tamaño.
  - Encodificación.
  - Modos de Direccionamiento.
- Implementación de software de un modelo (simulación) del procesador como tal.
- Diseño de una aplicación con cuatro métodos de encriptación de imágenes.
- Procesar 8 píxeles (datos) de forma simultáneamente.
- Definir la interfaz con memoria.
- Contar con instrucciones Vectoriales:
  - Aritméticas.
  - Lógicas.
  - Carga.
  - Almacenamiento.
  - Desplazamientos Regulares en ambas direcciones.
  - Desplazamientos Circulares en ambas direcciones.
- Manejar Vectores con datos (números enteros) de 8-bits.

- Manejar tamaños de vectores de al menos 8-bytes.
- Incluir al menos 12 instrucciones en el ISA:
  - Operaciones Vector - Vector.
  - Operaciones Vector - Escalar.
- El procesador debe implementar la técnica de segmentación (pipeline).
- Lógica de Detección de Riesgos (de ser necesario).
- Poseer al menos 4 ”lanes”para la ejecución paralela del hardware.
- El modelo de software debe emular el comportamiento paralelo del hardware, por lo que las abstracciones utilizadas deben soportar concurrencia en cierta medida.
- Diseñar una aplicación que a partir de una imagen en escala de grises (pre-cargada en memoria, o cargada por algún método) aplique cada uno de los algoritmos de encriptación de imágenes, y muestre la imagen original, la imagen encriptada y la imagen desencriptada en una ventana.
- Algoritmos de encriptación:
  - XOR con clave privada.
  - Desplazamiento circular.
  - Suma simple.
  - Algoritmo de encriptación propio.
- Diseñar una aplicación/script que convierta de lenguaje ensamblador propio a código máquina (binario).
- Crear un programa que demuestre de manera inequívoca el funcionamiento de todas las instrucciones del set.

Consideraciones Generales

- La elección del lenguaje de programación y demás herramientas de software a utilizar queda a criterio de cada estudiante.
- El modelo a implementar, en software, debe ser una representación funcional correcta de un organización en hardware. En todo momento se podrá visualizar las señales y registros internos del procesador, los contenidos de la memoria, las etapas, datos y señales del pipeline, entre otros componentes de la organización.
- Se recomienda el uso del paradigma de programación orientada a objetos, para dar una mejor representación de concurrencia de los módulos de hardware.
- La herramienta debe mostrar un diagrama de la organización del sistema.
- El modelo diseñado debe permitir las mediciones de tiempo de ejecución, en ciclos de reloj del procesador, así como un estimado de tiempo de ejecución a una frecuencia de 1GHz.
- El modelo debe permitir una ejecución por ciclo, así como una ejecución total (sin pausas). En la ejecución por ciclo se debe poder visualizar todas las señales y contenido de la organización, así como dar seguimiento de los datos en las etapas del pipeline.
- El desarrollo de software deberá realizarse utilizando un repositorio en línea como sistema de control de versiones.
- Para la codificación del software deberá establecerse explícitamente (en la sección de diseño) y seguirse adecuadamente algún código, norma o estándar establecido.

III. Diseño del ISA y Explicación

En está sección se procederá a explicar el proceso de diseño e implementación del ISA creado para el procesador vectorial a ser desarrollado en este proyecto; así mismo, se procederá a explicar cada uno de los tipos de instrucciones que posee, la codificación de cada una de estas, su funcionamiento y demás aspectos relacionados con la construcción y correcto funcionamiento del conjunto de instrucciones del procesador elaborado.

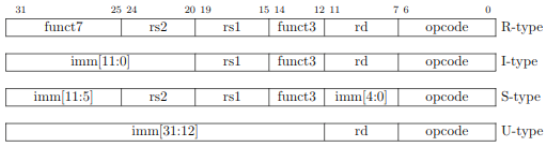
III-A. Sistema Base - RC32I v2.0

Primeramente, para el diseño del ISA que se explicara a continuación se hizo utilización de un sistema base, como comúnmente se hace al realizar un sistema de este tipo; por lo tanto este proyecto no fue la excepción y se hizo utilización del set de instrucciones de la arquitectura RISC-V para el diseño base.

Para este sistema se utilizo el modelo de codificación de RISC-V RV32I en su versión 2.0, la cual hace referencia a un sistema base de 32-bits con manejo de datos de tipo entero (nada de operaciones de punto flotante o datos tipo doble). Esta versión cuenta con un conjunto de 31 registros de propósito general (x1 -x31) para el almacenamientos de datos enteros de 32 bits, cuenta con un registro x0, el cual almacena el valor constante çeroz cuenta con un registro extra ”PC”, el cual mantiene la dirección actual a la cual apunta el Program Counter. Para la implementación de nuestro ISA se utilizaran este mismo conjunto de registros, pero con sus posibles variaciones

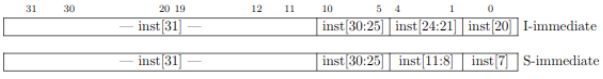
en cuanto al largo de palabra, cantidad de registros y posibles usos específicos. Las variaciones se presentaran en las propuestas planteadas en la próxima sección del documento.

RV32I define un conjunto de cuatro formatos base para la codificación de las instrucciones que recibirá el procesador, todas tiene un largo de 32-bits de largo y deben estar alineadas en 4-bytes de memoria. Para esta versión de RISC se define los formatos base R, I, S y U



En nuestro caso utilizaremos los formatos R, I y S para nuestra arquitectura, con el cambio de que el formato R lo llamaremos formato V, al tratarse de operaciones entre vectores y no entre registros; de forma que V será para operaciones vector-vector, I será para operaciones vector-escalar / escalar-vector y el formato S será para operaciones de almacenamiento en memoria.

Para el caso de aquellas instrucciones que hacen uso de valores inmediatos, tal como los formatos I y S, se le aplicara una extensión de signo a los valores inmediatos siempre, pero en aquellos casos donde se haga uso de este como operando como valor escalar, se recortara a un valore entero de 8-bits para que este concuerde con las operaciones y datos de los vectores a procesar. Para dicho proceso de extensión de signo se seguirá lo establecido en la versión 2.0 de RV32I para este caso, lo cual se muestra en la tabla a continuación.



Seguidamente, se tiene que la memoria en un sistema basado en RISC-V en su versión RV32I se trabaja con el formato de almacenamiento de ”Little Endians”, de forma que los datos serán almacenados con los bits mas bajos a la derecha y los más altos a la izquierda, y a la hora de transmitir esta infracción se mantendrá este formato, de forma que se mantenga la uniformidad y el estándar de procesamiento de los datos a lo largo del sistema.

Ahora bien, como requerimiento del sistema tenemos que el procesador debe ser y cumplir con el paralelismo a nivel de datos por medio de la implementación de registros vectoriales y unidades funcionales capaces de llevar a cabo una instrucción sobre múltiples datos, por lo cual también nos basaremos en la .[ex]tensión estándar para operaciones vectoriales, versión 0.2”(V) de RISC-V. Este estándar consiste de un conjunto de 32 registros vectoriales de datos (v0-v31), 8 vectores de predicado (vp0-vp7) y otro conjunto de registros de configuración que no serán utilizados en la implementación que plantea este documento.

III-B. Listado y Especificaciones de Instrucciones

Seguidamente, y tomando en cuenta cada uno de los puntos tomados como base para la construcción del presente ISA, se explicaran y mencionaran cada uno de las instrucciones presentes en el set, junto con sus aspectos relevantes (opcode, tipo, formato de operandos, tamaño, codificación, modo de direccionamiento, etc)

III-B1. Instrucciones de tipo V: Formato de codificación

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
funct7	rs2	rs1	funct3	rd	opcode

Instrucciones tipo V

- ADDVV vxd, vx1, vx2

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
funct7	rs2	rs1	funct3	rd	opcode
0000000	vx2	vx1	000	vxd	1000000

  - OPcode: 1000000
  - Operandos: vx1 (src1), vx2 (src1), vxd (dest)
  - Expresión: $vxd[i] = vx1[i] + vx2[i]$
  - Instrucción que suma los elementos de 2 vectores y coloca el resultado en un vector resultante

- SUBVV vxd, vx1, vx2

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
funct7	rs2	rs1	funct3	rd	opcode
0000000	vx2	vx1	000	vxd	1000001

- OPcode: 1000001
- Operandos: vx1 (src1), vx2 (src1), vxd (dest)
- Expresión: $vxd[i] = vx1[i] - vx2[i]$
- Instrucción que resta los elementos de 2 vectores y coloca el resultado en un vector resultante

- MULVV vxd, vx1, vx2

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
funct7	rs2	rs1	funct3	rd	opcode
0000000	vx2	vx1	000	vxd	1000010

- OPcode: 1000010
- Operandos: vx1 (src1), vx2 (src1), vxd (dest)
- Expresión: $vxd[i] = vx1[i] * vx2[i]$
- Instrucción que multiplica los elementos de 2 vectores y coloca el resultado en un vector resultante

- DIVVV vxd, vx1, vx2

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
funct7	rs2	rs1	funct3	rd	opcode
0000000	vx2	vx1	000	vxd	1000011

- OPcode: 1000011
- Operandos: vx1 (src1), vx2 (src1), vxd (dest)
- Expresión: $vxd[i] = vx1[i]/vx2[i]$
- Instrucción que divide los elementos de 2 vectores y coloca el resultado en un vector resultante

Instrucciones de Encriptación

- SLEVV vxd, vx1, vx2

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
funct7	rs2	rs1	funct3	rd	opcode
0000000	vx2	vx1	001	vxd	0000000

- OPcode: 0000000
- Operandos: vx1 (src1), vx2 (src1), vxd (dest)
- Expresión: $vxd[i] = vx1[i] << vx2[i]$
- Instrucción que corre (shift) los bits del vector vx1[i] hacia la izquierda en vx2[i] veces y lo coloca en vxd[i]

- SREVV vxd, vx1, vx2

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
funct7	rs2	rs1	funct3	rd	opcode
0000000	vx2	vx1	001	vxd	0000010

- OPcode: 0000010
- Operandos: vx1 (src1), vx2 (src1), vxd (dest)
- Expresión: $vxd[i] = vx1[i] >> vx2[i]$
- Instrucción que corre (shift) los bits del vector vx1[i] hacia la derecha en vx2[i] veces y lo coloca en vxd[i]

- XOREVV vxd, vx1, vx2

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
funct7	rs2	rs1	funct3	rd	opcode
0000000	vx2	vx1	001	vxd	0000100

- OPcode: 0000100
- Operandos: vx1 (src1), vx2 (src1), vxd (dest)
- Expresión: $vxd[i] = vx1[i]^v x2[i]$
- Instrucción que aplica una operación XOR a los bits del vector vx1[i] con el vector vx2[i] y lo coloca el resultado en vxd[i]

- OWNEP vxd, vx1, vx2

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
funct7	rs2	rs1	funct3	rd	opcode
0000000	vx2	vx1	001	vxd	0000110

- OPcode: 0000110
- Operandos: vx1 (src1), vx2 (src1), vxd (dest)
- Expresión: $vxd[i] = vx1[i] * *vx2[i]$
- Instrucción eleva a la vx2[i] el valor de vx1[i] y lo coloca en vxd[i]

Instrucciones de Desencriptación

- SLDVV vxd, vx1, vx2

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
funct7	rs2	rs1	funct3	rd	opcode
0000000	vx2	vx1	010	vxd	0000000

- OPcode: 0000000
- Operandos: vx1 (src1), vx2 (src1), vxd (dest)
- Expresión: $vxd[i] = vx1[i] << vx2[i]$

- Instrucción que corre (shift) los bits del vector vx1[i] hacia la izquierda en vx2[i] veces y lo coloca en vxd[i]

- SRDVV vxd, vx1, vx2

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
funct7	rs2	rs1	funct3	rd	opcode
0000000	vx2	vx1	010	vxd	0000010

- OPcode: 0000010
- Operandos: vx1 (src1), vx2 (src1), vxd (dest)
- Expresión: $vxd[i] = vx1[i] >> vx2[i]$
- Instrucción que corre (shift) los bits del vector vx1[i] hacia la derecha en vx2[i] veces y lo coloca en vxd[i]

- XORDVV vxd, vx1, vx2

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
funct7	rs2	rs1	funct3	rd	opcode
0000000	vx2	vx1	010	vxd	0000100

- OPcode: 0000100
- Operandos: vx1 (src1), vx2 (src1), vxd (dest)
- Expresión: $vxd[i] = vx1[i]^v x2[i]$
- Instrucción que aplica la operación XOR a los bits del vector vx1[i] contra vx2[i] y coloca el resultado en vxd[i]

- OWNDP vxd, vx1, vx2

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
funct7	rs2	rs1	funct3	rd	opcode
0000000	vx2	vx1	010	vxd	0000110

- OPcode: 0000110
- Operandos: vx1 (src1), vx2 (src1), vxd (dest)
- Expresión: $vxd[i] = sqrt(vx1[i], vx2[i])$
- Instrucción que saca la raíz vx2[i] del valor de vx1[i] y lo coloca en vxd[i]

III-B2. Instrucciones de tipo I: Formato de codificación

31 - 20	19 - 15	14 - 12	11 - 7	6 - 0
imm	rs1	funct3	rd	opcode

Instrucciones tipo I

- ADDVS vxd, vx1, imm

31 - 20	19 - 15	14 - 12	11 - 7	6 - 0
imm	rs1	funct3	rd	opcode
imm	vx1	000	vxd	1000100

- OPcode: 1000100
- Operandos: vx1 (src1), vxd (dest), inmediato
- Expresión: $vxd[i] = vx1[i] + imm$
- Instrucción que suma vx1[i] con un número inmediato y coloca el resultado en vxd[i]

- SUBVS vxd, vx1, imm

31 - 20	19 - 15	14 - 12	11 - 7	6 - 0
imm	rs1	funct3	rd	opcode
imm	vx1	000	vxd	1000101

- OPcode: 1000101
- Operandos: vx1 (src1), vxd (dest), inmediato
- Expresión: $vxd[i] = vx1[i] - imm$
- Instrucción que resta vx1[i] con un número inmediato y coloca el resultado en vxd[i]

- SUBSV vxd, vx1, imm

31 - 20	19 - 15	14 - 12	11 - 7	6 - 0
imm	rs1	funct3	rd	opcode
imm	vx1	000	vxd	1000110

- OPcode: 1000110
- Operandos: vx1 (src1), vxd (dest), inmediato
- Expresión: $vxd[i] = imm - vx1[i]$
- Instrucción que resta vx1[i] con un número inmediato y coloca el resultado en vxd[i]

- MULVS vxd, vx1, imm

31 - 20	19 - 15	14 - 12	11 - 7	6 - 0
imm	rs1	funct3	rd	opcode
imm	vx1	000	vxd	1000111

- OPcode: 1000111
- Operandos: vx1 (src1), vxd (dest), inmediato
- Expresión: $vxd[i] = vx1[i] * imm$
- Instrucción que multiplica vx1[i] con un número inmediato y coloca el resultado en vxd[i]

- DIVVS vxd, vx1, imm

31 - 20	19 - 15	14 - 12	11 - 7	6 - 0
imm	rs1	funct3	rd	opcode
imm	vx1	000	vxd	1001000

- OPcode: 1001000
- Operandos: vx1 (src1), vxd (dest), inmediato
- Expresión: $vxd[i] = vx1[i]/imm$
- Instrucción que divide vx1[i] con un número inmediato y coloca el resultado en vxd[i]

- DIVSV vxd, vx1, imm

31 - 20	19 - 15	14 - 12	11 - 7	6 - 0
imm	rs1	funct3	rd	opcode
imm	vx1	000	vxd	1001001

- OPcode: 1001001
- Operandos: vx1 (src1), vxd (dest), inmediato
- Expresión: $vxd[i] = imm/vx1[i]$
- Instrucción que divide un número inmediato con vx1[i] y coloca el resultado en vxd[i]

- LV vxd, sx1

31 - 20	19 - 15	14 - 12	11 - 7	6 - 0
imm	rs1	funct3	rd	opcode
imm	sx1	000	vxd	1001010

- OPcode: 1001010
- Operandos: sx1 (src1), vxd (dest)
- Direccionamiento: Absoluto
- Expresión: $vxd = MEM[sx1]$
- Instrucción que carga el contenido de la memoria en la dirección almacenada en sx1 en el registro vectorial vxd

- LSI sxd, sx1

31 - 20	19 - 15	14 - 12	11 - 7	6 - 0
imm	rs1	funct3	rd	opcode
imm	sx1	000	sxd	1001011

- OPcode: 1001011
- Operandos: sxd (dest), inmediato
- Expresión: $sxd = imm$
- Instrucción que carga un inmediato en el registro escalar sxd

- LSM sxd, sx1, imm

31 - 20	19 - 15	14 - 12	11 - 7	6 - 0
imm	rs1	funct3	rd	opcode
imm	sx1	000	sxd	1001100

- OPcode: 1001100
- Operandos: sx1 (src1), sxd (dest), inmediato
- Direccionamiento: Absoluto
- Expresión: $sxd = MEM[sx1]$
- Instrucción que carga el contenido de la memoria en la dirección almacenada en sx1 en el registro escalar sxd

- LVWS vxd, sx1, imm

31 - 20	19 - 15	14 - 12	11 - 7	6 - 0
imm	rs1	funct3	rd	opcode
imm	sx1	000	vxd	1001101

- OPcode: 1001101
- Operandos: sx1 (src1), vxd (dest), inmediato
- Direccionamiento: Relativo
- Expresión: $vxd = MEM[sx1 + imm]$
- Instrucción que carga el contenido de la memoria en la dirección almacenada en sx1 en el registro vectorial vxd

Instrucciones de Encriptación

- SLEVS vxd, vx1, imm

31 - 20	19 - 15	14 - 12	11 - 7	6 - 0
imm	rs1	funct3	rd	opcode
imm	vx1	001	vxd	0000001

- OPcode: 0000001
- Operandos: vx1 (src1), inmediato, vxd (dest)
- Expresión: $vxd[i] = vx1[i] << imm$
- Instrucción que corre (shift) los bits del vector vx1[i] hacia la izquierda en imm veces y lo coloca en vxd[i]

- SREVS vxd, vx1, imm

31 - 20	19 - 15	14 - 12	11 - 7	6 - 0
imm	rs1	funct3	rd	opcode
imm	vx1	001	vxd	0000011

- OPcode: 0000011
- Operandos: vx1 (src1), inmediato, vxd (dest)

- Expresión: $vxd[i] = vx1[i] >> imm$
- Instrucción que corre (shift) los bits del vector vx1[i] hacia la derecha en imm veces y lo coloca en vxd[i]

- XOREVS vxd, vx1, imm

31 - 20	19 - 15	14 - 12	11 - 7	6 - 0
imm	rs1	funct3	rd	opcode
imm	vx1	001	vxd	0000101

- OPcode: 0000101
- Operandos: vx1 (src1), inmediato, vxd (dest)
- Expresión: $vxd[i] = vx1[i]^v x2[i]$
- Instrucción que aplica una operación XOR a los bits del vector vx1[i] con el inmediato y lo coloca el resultado en vxd[i]

Instrucciones de Desencriptación

- SLDVS vxd, vx1, imm

31 - 20	19 - 15	14 - 12	11 - 7	6 - 0
imm	rs1	funct3	rd	opcode
imm	vx1	010	vxd	0000001

- OPcode: 0000001
- Operandos: vx1 (src1), inmediato), vxd (dest)
- Expresión: $vxd[i] = vx1[i] << imm$
- Instrucción que corre (shift) los bits del vector vx1[i] hacia la izquierda en imm veces y lo coloca en vxd[i]

- SRDVS vxd, vx1, imm

31 - 20	19 - 15	14 - 12	11 - 7	6 - 0
imm	rs1	funct3	rd	opcode
imm	vx1	010	vxd	0000011

- OPcode: 0000011
- Operandos: vx1 (src1), inmediato, vxd (dest)
- Expresión: $vxd[i] = vx1[i] >> imm$
- Instrucción que corre (shift) los bits del vector vx1[i] hacia la derecha en imm veces y lo coloca en vxd[i]

- XORDVS vxd, vx1, imm

31 - 20	19 - 15	14 - 12	11 - 7	6 - 0
imm	rs1	funct3	rd	opcode
imm	vx1	010	vxd	0000101

- OPcode: 0000101
- Operandos: vx1 (src1), inmediato, vxd (dest)
- Expresión: $vxd[i] = vx1[i]^i mm$
- Instrucción que aplica la operación XOR a los bits del vector vx1[i] contra imm y coloca el resultado en vxd[i]

III-B3. Instrucciones de tipo S: Formato de codificación

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
imm[7]	rs2	rs1	funct3	imm[5]	opcode

Instrucciones tipo S

- SV vx1, imm

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
imm[7]	rs2	rs1	funct3	imm[5]	opcode
imm[7]	00000	vx1	000	imm[5]	1001110

- OPcode: 1001110
- Operandos: vx1 (dest), inmediato
- Direccionamiento: Absoluto
- Expresión: $MEM[imm] = vx1$
- Instrucción que guarda en la dirección de memoria dada por imm el contenido registro vectorial vx1

- SVWS vx1, sx1, imm

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
imm[7]	rs2	rs1	funct3	imm[5]	opcode
imm[7]	sx1	vx1	000	imm[5]	1001111

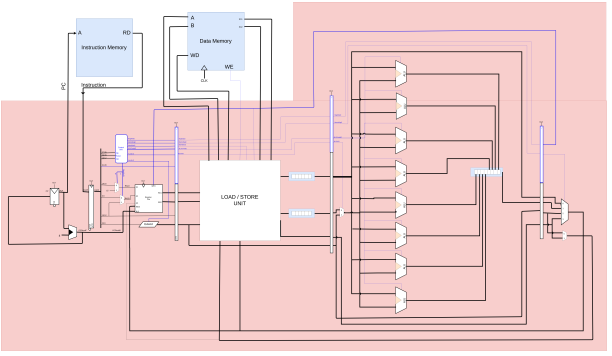
- OPcode: 1001111
- Operandos: vx1 (dest), inmediato, sx1 (src2)
- Direccionamiento: Relativo
- Expresión: $MEM[sx1 + imm] = vx1$
- Instrucción que guarda en la dirección de memoria dada por sx1 + imm el contenido registro vectorial vx1

- SS vx1, imm

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
imm[7]	rs2	rs1	funct3	imm[45]	opcode
imm[7]	00000	sx1	000	imm[5]	1010000

- OPcode: 1010000
- Operandos: sx1 (dest), inmediato
- Direccionamiento: Absoluto
- Expresión: $MEM[imm] = sx1$
- Instrucción que guarda en la dirección de memoria dada por imm el contenido registro escalar sx1

IV-A. Solución A - RV32VH8



IV. Opciones de solución al problema

A continuación, se presentan las dos propuestas de soluciones planteadas para el modelo de la microarquitectura que cumpliría y ejecutara cada una de las condiciones e instrucciones planteadas en la previa explicación del ISA elaborado para este problema de paralelismo a nivel de datos. Las soluciones a continuación planteadas presentan una interfaz de memoria basada en el esquema de Hardvard, es decir, presentan una memoria para instrucciones y para datos de forma separada y ubicadas fuera del procesador en cuestión, esto con el fin de modular el diseño y disminuir el área del procesador al colocar por fuera de este las memorias; la separación entre instrucciones y datos, fue seleccionada con el objetivo de disminuir la complejida del direccionamiento a la hora de leer o escribir en dicha memoria. También, ambas propuestas implementan una arquitectura segmentada (pipeline) para aumentar el desempeño del dispositivo y disminuir la latencia.

Cabe resaltar que la propuestas a continuación fueron elaboradas bajo la metodología de diseño modular, des la cual se planteo un sistema base con entradas y salidas, para seguidamente proceder a una especificación más detallada del sistema. Esta metodología en conjunto con el set de instrucciones creado como solución del problema dieron como resultado las propuestas que se muestran a continuación.

Esta propuesta cuenta con una arquitectura segmenta, se diseño con un pipeline de 5 etapas, entre ellas: fetch, decode, mem, exe y write back. A diferencia de las arquitecturas convencionales segmentadas, para el caso de esta arquitectura vectorial se decidió colocar primeramente la etapa de memoria y posterior a esta la etapa de ejecución de cálculos u operaciones, esto con el fin de procesar los datos una vez estos sean cargados. La arquitectura esta diseñada de forma que no existen saltos condicionales, tal que la ejecución completa del programa se lleva a cabo de forma secuencial. En la etapa de decode se cuenta con un banco de 16 registros (s0-s15), donde s0 almacena la constante del valor cero en todo momento y el registro s15 esta destinado para el almacenamiento del pc. En esta etapa también se cuenta con una unidad de extensión de signo y la unidad de control especializada en la generación de las señales de control correspondiente para la correcta ejecución del sistema.

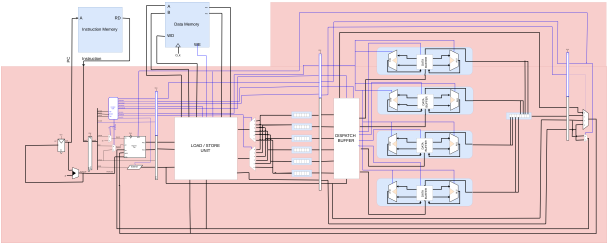
En la etapa de memoria, se tiene una unidad de Load/Store Unit, la cual es la encargada de llevar el control del proceso de almacenamiento de datos en memoria y en los registros vectoriales, así como de llevar el control de las posiciones y cantidades de elementos a ser cargadas en los diferentes procesos vectoriales; así mismo, es capaz de cargar datos escalares de memoria y direccionar esta de forma absoluta o relativa y realizar las cargas de datos por palabras completas, media palabra o por datos de 1-byte cada uno.

Seguidamente, este diseño solo cuenta con dos registros vectoriales para la ejecución de las operaciones, por lo cual que a cargo del programador la lógica de carga de

vectores de forma secuencial para evitar la mezcla de los datos, de igual forma el compilador a diseñar para el ISA diseñado tomara parte en este control.

Finalmente, para cumplir con la especificación de procesar al menos 8 píxeles al mismo tiempo, la etapa de ejecución se diseño bajo el principio de ALUs paralelas, de forma que se cuenta con 8 ALUs completamente idénticas que comparten señales de control y que toman los datos de los buses provenientes de los registros vectoriales, de forma que procesan cada uno de los 8 datos contenidos en los buses vectoriales y colocan el resultado de la operación en el vector de salida resultante, para finalmente proceder a la etapa de write back y realizar la operación de almacenamiento ya sea a registro o a memoria.

IV-B. Solución B - RV32VH4



En esta segunda propuesta se mantiene el mismo principio de arquitectura segmentada con 5 etapas en el pipeline y con el mismo orden de ejecución. Las principales diferencias de esta propuesta radican en:

Se añadieron 4 registros vectoriales más, dejando un total de 6 registros vectoriales para mayor manejo de datos y de operaciones, junto con estos registros vectoriales se hace la adición de dos decodificadores de 1 - 6, que son los encargados de direccionar los datos a su vector correspondiente.

Seguidamente, se pasa de un modelo de 8 lanes, a uno de 4 lanes, cada uno con 2 ALUs y un buffer de datos interno para el procesamiento de los datos. El aumento de los registros vectoriales y el cambio en la organización de los lanes, obliga a crear una logica de reparto de datos entre los diferentes lanes, para que estos realicen la ejecución de las operaciones; así mismo, este modelo podría permitir la ejecución de múltiples operaciones sobre diferentes conjuntos de datos, al separar las señales de control de las ALUs y del control de los buffers internos de los lanes, los cuales una el procesado un conjunto de datos escriben su salida al registro vectorial de resultado para su posterior escritura a memoria o bien a alguno de los 6 registros vectoriales de la arquitectura para permitir una ejecución en cadena de los datos.

Inclusive a esta propuesta podría reemplazarse el vector de resultados por un buffer de reordenamiento de resultados, el cual podría encargarse de reorganizar todos y cada uno de los datos procesados en los lanes y preparar los para su almacenamiento.

V. Comparación de las soluciones

En esta sección realizaremos un análisis detallado de las dos propuestas anteriores y a partir de este análisis seremos capaces de determinar cual es la opción más viable en términos de consumo de potencia, complejidad, costo, área, cumplimiento de objetivos, entre otras.

V-A. Consumo de Potencia

En cuanto a consumo de potencia se tiene que ambas arquitecturas comparten el diseño de las etapas de fetch, decode y write back, por lo cual la diferencia en consumo de potencia radica en las secciones de memoria y de ejecución. En la etapa de memoria la similitud es grande, con la diferencia de que la propuesta RV32VH4 cuenta con 4 registros vectoriales más que la RV32VH8, aunque el consumo de energía de estas unidades extra podría crear una diferencia mínima. La mayor diferencia en cuanto a consumo se localiza en la etapa de ejecución en la cual la versión RV32VH8 cuenta con un mux, 8 ALUs y un registro vectorial para el resultado de las operaciones; en cambio, la versión RV32VH4 cuenta con las mismas 8 ALUs, el mismo registro vectorial de salida, pero cambia el mux por una logica de entrega y organización de datos, componente que internamente estaría compuesto por un gran conjunto de decodificadores, muxes y registros, además al simplificar el modelo a 4 lanes, se le añadió un buffer interno a cada lane para mantener los operandos, lo cual aumentaría el consumo de potencia de la versión RV32VH4 en gran proporción con respecto a la versión RV32VH8.

V-B. *Complejidad del Diseño*

En cuanto a complejidad del diseño, la versión RV32VH8 tiene una versión más simplificada del manejo de las operaciones vectoriales y de la logica de ordenamiento de los datos en la etapa de ejecución con respecto a la versión RV32VH4, la cual cuenta con la 4 registros vectoriales más que por ende añaden la complejidad de cumplir con la correcta calendarización de las operaciones en los diferentes lanes en el orden correcto y de obtener la salida en el orden correcto, tal que la logica de la unidad de calendarización (dispatch) complica el diseño de la micro-arquitectura.

V-C. *Costo*

El costo de producción en el caso de la versión RV32VH4 sería relativamente mayor a de la versión RV32VH8 respecto a la composición interna de los lanes, la red de control más compleja y amplia, y debido al costo de producción de la logica de calendarización de los datos en la etapa de ejecución, sin mencionar que se añaden 4 registros vectoriales más, los cuales suelen ser más costosos al ser memoria pequeña y de rápido acceso.

V-D. *Área*

El área de ambas propuestas es similar, a simple vista y basándonos en los diagramas de la micro-arquitectura y sus diferencias, podría decirse que la propuesta RV32VH4 es la que tiene la mayor área debido a la adición de los 4 registros vectoriales extra, la logica de calendarización y los 4 buffers internos en los lanes, elementos que por si solos no representan una gran área, pero al estar en conjunto con todo el resto de la solución hace que este incremento sea relevante. Especialmente, porque se desconoce el tamaño y complejidad que podría alcanzar el modulo de calendarización y una posible adición de un modulo de reordenamiento.

V-E. *Cumplimiento de Requerimientos*

En cuanto al cumplimiento de los requerimientos ambas micro-arquitecturas fueron planteadas para cumplir con el diseño de ISA inicial, y dicho ISA fue elaborado para cumplimiento de cada uno de los requerimientos solicitados para la solución de este problema, por lo tanto las dos propuestas cumplen con esta cualidad.

VI. Selección de la propuesta final

Seguidamente, y de acuerdo a los criterios de evaluación mencionados y evaluados en la sección anterior se tiene que la propuesta final a ser realizada sera la propuesta RV32VH8ƨa que según los criterios de comparación y las suposiciones realizadas en la etapa anterior propone, está propuesta es la mejor opción en cuanto a consumo de potencia, menor área, menor costo de producción, cumplimiento de requisitos y la de menor complejidad para ser desarrollada en el tiempo que se estableció para el desarrollo del presente proyecto.

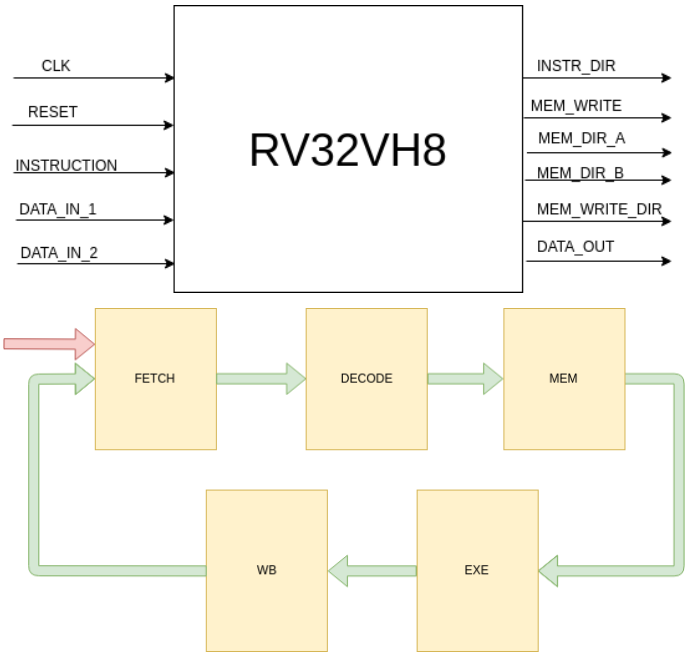
VII. Implementación del diseño

A continuación, se describe el proceso de implementación de la propuesta RV32VH8, seleccionada como la propuesta final.

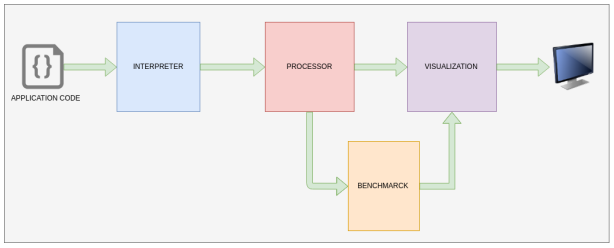
VII-A. *Descripción del ISA*

La descripción del set de instrucciones utilizado e implementado para la implementación de esta solución es el mismo que se explico y detallo en la sección III. Diseño del ISA y Explicación (pág. 3)

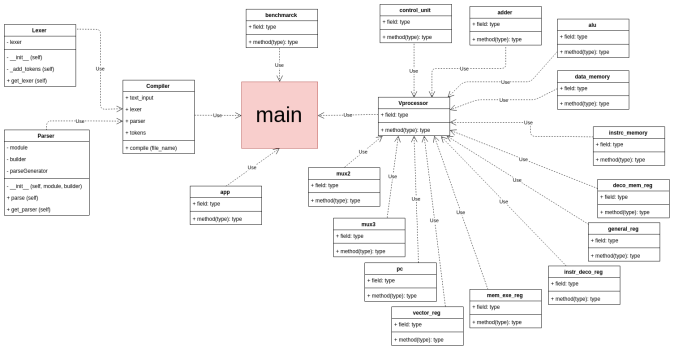
VII-B. *Diagrama de bloques del modelo del procesador*



VII-C. Diagrama de bloques del computador



VII-D. Diseño de software



VII-E. Descripción de algoritmo propuesto

Para el desarrollo de la aplicación y simulación del software aquí planteado se hizo utilización del lenguaje de programación de python debido a su versatilidad y facilidad de desarrollo, así mismo se siguió el estándar de código en python PEP-8, que establece los estándares de código a seguir a la hora de realizar programación en este lenguaje.

Así mismo, se hizo utilización de las bibliotecas de OpenMP para la simulación del sistema concurrente al hacer uso de esta biblioteca que permite el multiprocesamiento, dando la apariencia de estar corriendo el hardware simulado de forma simultanea como en la vida real.

El código planteado consta de 3 paquetes principales, el paquete de .ªpp", el cual contiene todo el código desarrollado para la aplicación de visualización y de encriptación de las imágenes. El paquete de "benchmark", el cual posee las rutinas de evaluación del sistema que confirman el funcionamiento de la arquitectura y de todas y cada una de las instrucciones. El paquete de "hw", el cual contiene todo el código base de cada uno de los componentes que forman la arquitectura de la propuesta RV32VH8 planteados como objetos y que por medio de la utilización de todos estos en el top "vector_processor.py"se logra la simulación del hardware propuesto, y finalmente, el paquete de ïnterpreter", el cual contiene el código del interprete diseñado para comprender y transformar a lenguaje maquina cada una de las instrucciones del ISA desarrollado, colocándolas en un archivo de salida que posteriormente sera cargado al sistema simulado de hardware para su ejecución.

REFERENCIAS

[1] onzáles G., J. (2019). Introducción al paralelismo a nivel de datos Lección 7.

[2] onzáles G., J. (2019). Arquitectura Vectorial Lección 8.

[3] Hennesy and David Patterson (2012) Computer Architecture: A Quantitative Approach. 5th Edition. Elsevier - Morgan Kaufmann.

[4] illiam Stallings (2010) Computer organization and architecture: designing for performance. Pearson Education India

[5] he RISC-V Instruction Set Manual. (2017). 2nd ed. [ebook] California: Andrew Waterman, Krste Asanović, pp.9-26 93 - 100 103 - 104. Available at: https:content.riscv.orgwp-contentuploads201705riscv-spec-v2.2.pdf [Accessed 8 May 2019].