

Arquitectura de Computadoras con RISC-V



José Alfredo Jaramillo Villegas
Hernán Mauricio Zuluaga Bucheli
Camilo Sepúlveda Caviedes

José Alfredo Jaramillo Villegas,
(Pereira, Risaralda, Colombia, 1980).

Doctorado por la Purdue University, West Lafayette, Indiana, Maestría en Física por la Universidad Tecnológica de Pereira e Ingeniería Electrónica por la Pontificia Universidad Javeriana.

Docente Titular en la Facultad de Ingenierías. Autor de los libros: "Computación Reconfigurable: Caso de Estudio BLAST-n" (2012) y "Silicon Nitride Microring Resonator: Classical and Quantum Applications" (2019).

Ha publicado artículos científicos en revistas nacionales e internacionales especializadas en su área. Ha recibido múltiples reconocimientos nacionales e internacionales, entre ellos la beca Francisco José de Caldas de Colciencias y Fulbright Scholarship del Departamento de Estado de U.S.A. para sus estudios de doctorado, la beca Bilsland Scholarship por su tesis doctoral de Purdue University, el premio al mejor emprendimiento social en Global Talent and Innovation Competition en Taiwan y Panamá, Global Young Social Entrepreneurs' Competition en Malaysia, Agentes de Cambio del Banco Interamericano de Desarrollo y la Cruz de los Fundadores de Pereira, por su participación en Proyecto Iris.

Miembro del Grupo de Investigación Sirius.

jiv@utp.edu.co

Hernán Mauricio Zuluaga Bucheli,
(Manizales, Caldas, Colombia 1994).

Ingeniero de Sistemas y Computación y estudiante de la Maestría en Ingeniería de Sistemas y Computación de la Universidad Tecnológica de Pereira

Docente Catedrático en la Facultad de Ingenierías.

Ha publicado artículos científicos en revistas nacionales e internacionales especializadas.

Integrante del Grupo de Investigación Sirius, Categoría A1 Minciencias.

herzulu@utp.edu.co

Camilo Sepúlveda Caviedes,
(Cartago, Valle, Colombia 2000).

Ingeniero de Sistemas y Computación de la Universidad Tecnológica de Pereira

Integrante del Grupo de Investigación Sirius, Categoría A1 Minciencias.

c.sepulveda@utp.edu.co

Arquitectura de Computadoras con RISC-V

José Alfredo Jaramillo Villegas, Ph.D.
Hernán Mauricio Zuluaga Bucheli
Camilo Sepúlveda Caviedes



Facultad de Ingenierías
Colección Textos Académicos
2022

Jaramillo Villegas, José Alfredo
Arquitectura de computadoras con RISC-V / José Alfredo Jaramillo
Villegas, Hernán Mauricio Zuluaga Bucheli y Camilo Sepúlveda
Caviedes. – Pereira : Universidad Tecnológica de Pereira, 2022.
151 páginas. – (Colección Textos académicos).

e-ISBN: 978-958-722-795-6

1. Historia de la computadora 2. Arquitectura de computadoras -
Fundamentos 3. Estructuras de datos y algoritmos 4. Procesamiento de
señales 5. Redes de computadoras y comunicaciones 6. Electrónica
digital

CDD. 004.22

Arquitectura de Computadoras con RISC-V

© José Alfredo Jaramillo Villegas, Ph.D.

© Hernán Mauricio Zuluaga Bucheli

© Camilo Sepúlveda Caviedes

© Universidad Tecnológica de Pereira

eISBN: 978-958-722-795-6

Imagen de cubierta: Freepik

Universidad Tecnológica de Pereira
Vicerrectoría de Investigaciones, Innovación y Extensión
Editorial Universidad Tecnológica de Pereira
Pereira, Colombia

Coordinador editorial:

Luis Miguel Vargas Valencia

luismvargas@utp.edu.co

Teléfono 313 7381

Edificio 9, Biblioteca Central “Jorge Roa Martínez”

Cra. 27 No. 10-02 Los Álamos, Pereira, Colombia

www.utp.edu.co

Montaje y producción:

Maria Alejandra Henao Jiménez

Universidad Tecnológica de Pereira

Pereira

Contenido

CAPÍTULO UNO

Introducción a RISK-V	15
1.1. Historia de la computadora	20
1.1.1. El quipu	21
1.1.2. La yupana	21
1.1.3. El ábaco chino	22
1.1.4. La pascalina	23
1.1.5. Rueda de Leibniz	24
1.1.6. Máquina analítica	27
1.1.7. Colossus	27
1.1.8. Z3	31
1.1.9. ABC	31
1.1.10. ENIAC	33
1.2. Criterios de diseño de computadoras	38
1.2.1. Diseñar teniendo en cuenta la ley de Moore	38
1.2.2. Usar la abstracción para simplificar el diseño	38
1.2.3. Reducir el tiempo de ejecución para los casos más comunes	39
1.2.4. Mejorar rendimiento a través del paralelismo	40
1.2.5. Mejorar rendimiento a través de la segmentación	40
1.2.6. Mejorar rendimiento a través de la predicción de saltos	41
1.2.7. Jerarquía de memorias	42
1.2.8. Mejorar fiabilidad a través de redundancia	43
1.3. Evaluación de rendimiento de la computadora	44
1.4. Ejercicios Capítulo 1	46

CAPÍTULO DOS

Instrucciones	51
2.1. Instrucciones aritmético lógicas	54
2.1.1. Instrucciones tipo R	54
2.1.2. Instrucciones tipo I	58
2.2. Instrucciones de memoria de datos	61
2.2.1. Instrucciones de lectura de memoria tipo I	61
2.2.2. Instrucciones de escritura en memoria tipo S	63
2.3. Instrucciones de saltos condicionados	65
2.4. Instrucciones de saltos incondicionados	72
2.4.1. jal tipo J	72
2.4.2. jalr tipo I	73
2.4.3. Funciones	73
2.5. Instrucciones auxiliares	80
2.6. Pseudo instrucciones	81
2.7. Ejercicios Capítulo 2	83

CAPÍTULO TRES

Procesador	87
3.1. Procesador Monociclo	87
3.1.1. Memoria de instrucciones	89
3.1.2. Unidad de control	89

3.1.3. Unidad de registros.....	92
3.1.4. Unidad de inmediatos.....	93
3.1.5. Unidad aritmética lógica	94
3.1.6. Unidad de saltos	96
3.1.7. Memoria de datos	96
3.1.8. Flujo de datos	98
3.2. Procesador segmentado.....	103
3.2.1. Procesador segmentado con solución a los problemas de segmentación por software	105
3.2.2. Segmentos del procesador.....	108
3.2.3. Procesador segmentado con solución a los problemas de segmentación por hardware	114
3.3. Ejercicios Capítulo 3	123

CAPÍTULO CUATRO

Sistemas de Entrada y Salida	129
4.1. Espacio de direccionamiento.....	129
4.1.1. Controlador de video.....	131
4.1.2. Controlador de teclado	133
4.2. Ejercicios Capítulo 4	134

Apéndice A

Estándar internacional para unidades.....	135
A.1. Estándar ISO/IEC 80000.....	136
A.1.1. Estándar para ciencia de la información y tecnología ISO/IEC 80000-13	137

Apéndice B	139
-------------------------	------------

Glosario	139
-----------------------	------------

Bibliografía	147
---------------------------	------------

Figuras

Figura 1.1: Formato de instrucciones de RISC-V en lenguaje de máquina.....	19
Figura 1.2: Ejemplo de instrucción tipo R.....	19
Figura 1.3: Inca - quipu.....	22
Figura 1.4: La yupana	23
Figura 1.5: El Ábaco chino	24
Figura 1.6: La pascalina.....	25
Figura 1.7: Blaise Pascal (1623-1662).....	25
Figura 1.8: Gottfried Wilhelm von Leibniz (1646-1716)	26
Figura 1.9: Rueda de Leibniz.....	26
Figura 1.10: Charles Babbage (1791-1871).....	28
Figura 1.11: Máquina Analítica	28
Figura 1.12: Alan Turing (1912-1954).....	30
Figura 1.13: Colossus mark 2	30
Figura 1.14: Konrad Zuse (1910-1995)	32
Figura 1.15: Z3.....	33
Figura 1.16: Jhon V. Atanasoff (1903-1995).....	34
Figura 1.17: ABC	35
Figura 1.18: ENIAC Oficials	37
Figura 1.19: Ley de Moore	39
Figura 1.20: Tarea no optimizada.	41
Figura 1.21: Tarea segmentada.	42
Figura 1.22: Jerarquía de Memorias.	43
Figura 1.23: RAID Creative Commons 2.0	44
Figura 2.1: Tipos de instrucciones.	52
Figura 2.2: Ejemplo código de instrucción tipo R	57
Figura 2.3: Ejemplo código de instrucción tipo I	61
Figura 2.4: Ejemplo código de instrucción tipo S.....	65
Figura 2.5: Ejemplo código de instrucción tipo B	71
Figura 2.6: Ejemplo código de instrucción tipo J	72
Figura 2.7: Ejemplo código de instrucción tipo U.....	81
Figura 3.1: Procesador monociclo.	88
Figura 3.2: Contador de programa.....	89
Figura 3.3: Memoria de instrucciones.....	90
Figura 3.4: Unidad de control	90
Figura 3.5: Unidad de registros.....	93
Figura 3.6: Unidad de inmediatos.....	94
Figura 3.7: Unidad aritmético lógica.....	95
Figura 3.8: Unidad de saltos	96
Figura 3.9: Memoria de datos	98
Figura 3.10: Flujo de datos tipo R	99
Figura 3.11: Flujo de datos tipo I aritmético lógico.....	99
Figura 3.12: Flujo de datos tipo I de carga.	100
Figura 3.13: Flujo de datos tipo I de salto.	100
Figura 3.14: Flujo de datos tipo J.....	101
Figura 3.15: Flujo de datos tipo S.....	101
Figura 3.16: Flujo de datos tipo B.	102
Figura 3.17: Proceso de segmentación.....	103
Figura 3.18: Conjunto de instrucciones procesadas de forma totalmente secuencial.....	104

Figura 3.19: Conjunto de instrucciones procesadas de forma segmentada.	105
Figura 3.20: Procesador segmentado con solución a los problemas de la segmentación por software.	107
Figura 3.21: Etapa de búsqueda de instrucción.	109
Figura 3.22: Etapa de decodificación y lectura de operandos.	110
Figura 3.23: Etapa de ejecución.....	112
Figura 3.24: Etapa de acceso a memoria.	113
Figura 3.25: Etapa de escritura de resultado.....	114
Figura 3.26: Procesador segmentado con solución a los problemas de segmentación por hardware.	115
Figura 3.27: Dependencia de registro fuente uno.	116
Figura 3.28: Dependencia de registro fuente dos.....	117
Figura 3.29: Dependencia de registro fuente uno.	117
Figura 3.30: Dependencia de registro fuente dos.....	118
Figura 3.31: Dependencia doble.	119
Figura 4.1: Diagrama del controlador de video.	132
Figura 4.2: Tiempos de Porch	132
Figura 4.3: Horizontal Timing Specification.	132
Figura 4.4: Vertical Timing Specification.....	133
Figura 4.5: Diagrama controlador teclado	133
Figura 4.6: Diagrama de tiempos PS2.	134

Tablas

Tabla 2.1: Tabla de registros	53
Tabla 2.2: Códigos de una instrucción tipo R	55
Tabla 2.3: Códigos de una instrucción tipo I	59
Tabla 2.4: Códigos las instrucciones de lectura de memoria tipo I.....	62
Tabla 2.5: Códigos para las instrucciones de escritura en memoria tipo S.....	64
Tabla 2.6: Códigos de una instrucción tipo B	66
Tabla 2.7: Código de una instrucción tipo J.....	72
Tabla 2.8: Código de una instrucción tipo I para salto incondicional.....	73
Tabla 2.9: Código de una instrucción tipo U	80
Tabla 3.1: Señales de control de los formatos: R, I, I-Carga.	91
Tabla 3.2: Señales de control de los formatos: B, J, S	91
Tabla 3.3: Señales de control de los formatos: I-Salto, U.....	92
Tabla 3.4: Fuente del inmediato	94
Tabla 3.5: Opciones de la ALU.....	95
Tabla 3.6: Opciones de salto	97
Tabla 3.7: Control de la memoria de datos.	97
Tabla 4.1: Selección de Base Address.....	130
Tabla A.1: Estándares ISO/IEC 80000.....	136
Tabla A.2: Nomenclatura en base a los prefijos, símbolos del prefijo y.....	137
símbolos del múltiplo del byte.....	137
Tabla A.3: Nomenclatura de los bytes según el estándar ISO/IEC 80000-13.....	138

La formación en ingeniería de computadoras cumple una función fundamental en la educación en la ciencia de la computación. Esta capacitación proporciona a los estudiantes los conocimientos que necesitan para diseñar, configurar y maximizar las capacidades de los sistemas computacionales. Históricamente, esta área ha sido enseñada con arquitecturas de conjuntos de instrucciones de tipo reducido (RISC) dada la dificultad que conlleva el entendimiento de las arquitecturas de tipo complejo (CISC), tal como su nombre lo indica. En las últimas tres décadas, esta área ha pasado por varias arquitecturas RISC, tales como DLX, MIPS, SPARC y ARM [24]. Sin embargo, ninguna ha permitido una integración vertical totalmente abierta, de libre uso y sin restricciones por licenciamiento [51]. Esta arquitectura RISC-V pretende brindar un estándar moderno que fácilmente pueda cumplir las necesidades pedagógicas en el aula de clase. De igual forma, RISC-V logra brindar la robustez necesaria para cumplir con los requerimientos para su implementación en procesadores para teléfonos inteligentes, tabletas y sistemas embebidos, en donde actualmente domina la arquitectura ARM; y en computadoras portátiles y de escritorio, servidores y supercomputadoras, en donde domina la arquitectura X86-64. Podría proyectarse que en los próximos años el impacto de RISC-V en la arquitectura de computadoras será equivalente al rol que ha jugado Linux en los sistemas operativos.

El origen del interés por los estudios sobre el diseño de sistemas computacionales para realizar operaciones automatizadas y de manera óptima se ha gestado mediante el transcurso del tiempo y de la creciente tendencia del ser humano por realizar cada vez cálculos más exigentes. En este libro se presenta de manera original una perspectiva de los elementos básicos de la computación, amparados en el contexto histórico, además cada sección viene acompañada de un conjunto de ejercicios inéditos y originales de los autores, donde el lector podrá profundizar y consolidar el conocimiento presentado de manera teórica y práctica.

En cuanto a la estructura del libro, está dividido en las siguientes cuatro grandes secciones:

- En el capítulo uno se presenta una introducción a la arquitectura del conjunto de instrucciones reducidas RISC-V, también se presenta una perspectiva histórica de la computadora, brindando un análisis de los elementos más importantes en este contexto, posteriormente se presenta un conjunto de 8 ideas básicas que han sido desarrolladas por grandes personajes históricos, estas permiten un marco teórico amplio en cuanto a la mejora de procesos en la ingeniería, finalmente se realiza un análisis matemático de conceptos relacionados con las métricas del rendimiento computacional.
- En el capítulo dos se presenta el conjunto de instrucciones reducidas RISC-V de la siguiente manera:
 - Conjunto de instrucciones para realizar operaciones aritmético lógicas
 - Conjunto de instrucciones para realizar operaciones sobre la memoria
 - Conjunto de instrucciones para realizar saltos condicionados
 - Conjunto de instrucciones para realizar operaciones de saltos incondicionados
 - Conjunto de instrucciones auxiliares
 - Conjunto de pseudo instrucciones

Finalmente se presenta un conjunto de ejercicios teórico-prácticos para que el lector consolide la información contenida en este capítulo.

- En el capítulo tres se introduce los conceptos de procesador, iniciando con la explicación del procesador monociclo, del cual se definen los módulos necesarios para su funcionamiento, entre ellos se definirán los siguientes:
 - Unidad de control

- Unidad de inmediatos
- Unidad aritmética lógica
- Unidad de saltos
- Memoria de datos
- Unidad de registros
- Memoria de instrucciones

Posteriormente, se presenta un conjunto de técnicas para mejorar el rendimiento del procesador monociclo, entre ellas el uso de segmentación y paralelismo, también se estudian las dificultades de estas técnicas y se proponen diseños de arquitecturas originales por los autores para solucionar los problemas derivados de las técnicas aplicadas, entre ellas el uso de herramientas como software y otras con hardware.

- El capítulo cuatro se explica el acoplamiento del procesador para interactuar con elementos de entrada y salida, esto a partir de la generación de módulos de control, entre ellos:

- Controlador de video
- Controlador de teclado PS2
- Controlador de Switch
- Controlador de LED

1

CAPÍTULO UNO

Introducción a RISC-V

La ingeniería de computadoras es la rama de la ingeniería que integra la ciencia de la computación con la ingeniería electrónica. Esta última tiene como principal objetivo el diseño, configuración, dimensionamiento, mantenimiento y aprovechamiento de equipos de procesamiento de datos basados en sistemas digitales.

En este libro se presenta la adopción de la arquitectura de conjunto de instrucciones (Instruction Set Architecture ISA) RISC-V, pronunciado [rɪsk faɪv], para la enseñanza de la arquitectura de computadoras. Esta es libre de costo, abierta y sin restricciones de licenciamiento [47].

Los ISAs han jugado un papel fundamental en la historia de la computación [46]. Estas arquitecturas proveen una interfaz entre el software y el hardware; definen como los algoritmos escritos en lenguajes de alto nivel (ej. Lenguaje C) son traducidos a una serie de instrucciones simples, que pueden ser decodificadas y procesadas por un sistema digital denominado la unidad central de procesamiento (Central Processing Unit CPU) o simplemente procesador.

A principios de la década de los noventa, los profesores John Hennessy y David Patterson de la Universidad de Stanford y la Universidad de California en Berkeley, respectivamente, crearon un ISA denominado DLX, que luego evolucionó a MIPS (Microprocessor without Interlocked Pipelined Stages)

[22, 31]. MIPS ha jugado un papel fundamental en los cursos de arquitectura de computadoras dado el amplio uso en las aulas de clase de los libros escritos por ambos profesores [46, 30].

Dos de las arquitecturas más predominantes actualmente son x86-64 (también conocida como x64, AMD64 o Intel 64) y ARM (v7/v8). El ISA x86-64 domina el mercado de los servidores, computadoras de escritorio y portátiles, mientras que el ISA ARM (v7/v8) domina el segmento de los teléfonos inteligentes, las tabletas y los sistemas de propósito específico o sistemas embebidos.

Tanto la arquitectura x86-64 como la ARM son propietarias, lo cual significa que su uso está restringido y no pueden ser libremente aplicadas en el contexto académico y de investigación [60]. Adicionalmente, la complejidad de estas arquitecturas hace inviable su uso pedagógico en el aula de clase.

En respuesta a esta problemática, dos estudiantes de doctorado, Andrew Waterman y Yunsup Lee, junto con su profesor guía Krste Asanovic y asesorados por el profesor David Paterson en la Universidad de California en Berkeley, emprenden el diseño de RISC-V en 2010. La concepción del diseño de RISC-V se basó en los siguientes criterios:

- simple y fácil de enseñar en el contexto académico;
- abierto y libre de costos de licenciamiento;
- modular y extensible, de tal forma que se permita su ampliación a diferentes contextos;
- de amplio espectro con la posibilidad de implementar desde micro-controladores hasta supercomputadoras.

Actualmente, RISC-V es mantenido por la RISC-V Foundation [59] formada en 2015, siguiendo la misma exitosa filosofía apropiada por la comunidad de software libre con la Free Software Foundation. A 2021,

RISC-V Foundation cuenta con más de 200 miembros, entre ellos 13 universidades y grandes empresas tales como Qualcomm, Google, Microsoft, IBM, AMD, NVIDIA e Intel. De hecho, para gran sorpresa de muchos especialistas en el área, en febrero de 2021 el Director Ejecutivo de Intel, Patrick P. Gelsinger, anunció la posibilidad de ofrecer servicios a terceros que quieran usar su infraestructura de manufactura de chips para fabricar procesadores de arquitectura RISC-V. Esto abre una puerta importante para la masificación y reducción de costos de estos procesadores.

RISC-V es soportado por una amplia gama de herramientas de desarrollo, entre ellas, compiladores (gcc), depuradores (gdb) y emuladores (qemu). Inclusive, recientemente se realizó el porte completo del sistema operativo Ubuntu en su versión 20.04 LTS. Aparte de brindar un amplio panorama de la adopción de RISC-V en la comunidad de software libre, este amplio soporte de software brinda las herramientas necesarias para el desarrollo de un plan de estudios integral de la línea de ingeniería de computadoras en el Programa de Ingeniería de Sistemas y Computación. La articulación del contenido de la asignatura de Arquitectura de computadoras podría extrapolarse a las asignaturas de Compiladores y Sistemas Operativos, unificados bajo un mismo marco de trabajo.

RISC-V, como su nombre lo indica, está basado en los principios fundamentales del diseño de un ISA tipo RISC. El concepto fundamental de RISC es que los programas no usan la mayoría de instrucciones de procesador, lo cual genera un desaprovechamiento del recurso de memoria en cuanto a la cantidad de bits que pueden reducirse, en consecuencia resultan transistores que no se usan dentro del procesador. En un famoso artículo de Andrew Tanenbaum, demostró que un programa de alto nivel de más de 10000 líneas, puede ser representado por un conjunto de hasta 256 instrucciones simples, usando 8 bits para la codificación de las operaciones [56]. Este estudio sugiere que se puede desarrollar un procesador más simple, con la capacidad de ejecutar cualquier código común usando una arquitectura de instrucciones reducida.

En RISC-V se definen 32 registros que pueden ser de propósito general,

pero algunos de ellos cumplen algunas funciones específicas. RISC-V está dividido en diferentes categorías que dependen del número de bits de los registros. Por ejemplo, el conjunto base se denomina RV32I, que hace referencia a RISC-V de procesamiento de enteros de 32 bits, lo que implica que el tamaño de los registros es también de 32 bits. También están los conjuntos base RV64I y RV128I en los cuales se adiciona compatibilidad con procesamiento de enteros de 64 y 128 bits, respectivamente y así mismo se aumenta el tamaño de los registros. Aparte de estos conjuntos base, se definen conjuntos de extensiones que complementan la funcionalidad:

- I instrucciones base de enteros
- M multiplicación y división de enteros
- A instrucciones atómicas
- F punto flotante de precisión simple
- D punto flotante de precisión doble
- G propósito general que incluye las extensiones IMAFD
- C instrucciones comprimidas de 16 bits

Para efectos pedagógicos es suficiente con realizar la implementación del conjunto base RV32I. Este último incluye 40 instrucciones que se encuentran categorizadas en 6 posibles formatos en lenguaje de máquina, como se describe en la figura (ver Figura: 1.1). En esta se puede observar como los campos de los identificadores de registros son coherentes en los diferentes formatos de instrucciones, disminuyendo así los multiplexores necesarios para el flujo de datos a través del procesador. Como ejemplo práctico, en la figura (ver Figura: 1.2) se presenta la instrucción add rd, rs1, rs2, la cual toma el contenido de dos registros fuente, rs1 y rs2, realiza la suma y el resultado es guardado en un registro destino rd.

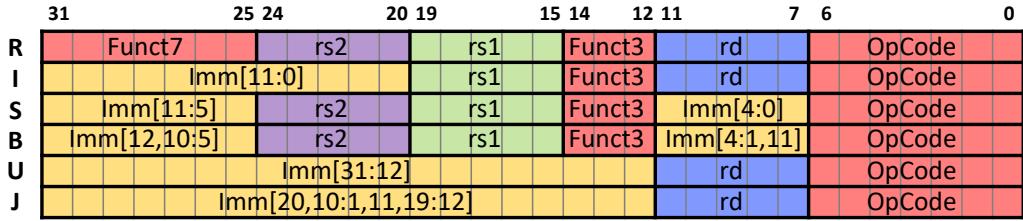


Figura 1.1: Formato de instrucciones de RISC-V en lenguaje de máquina.

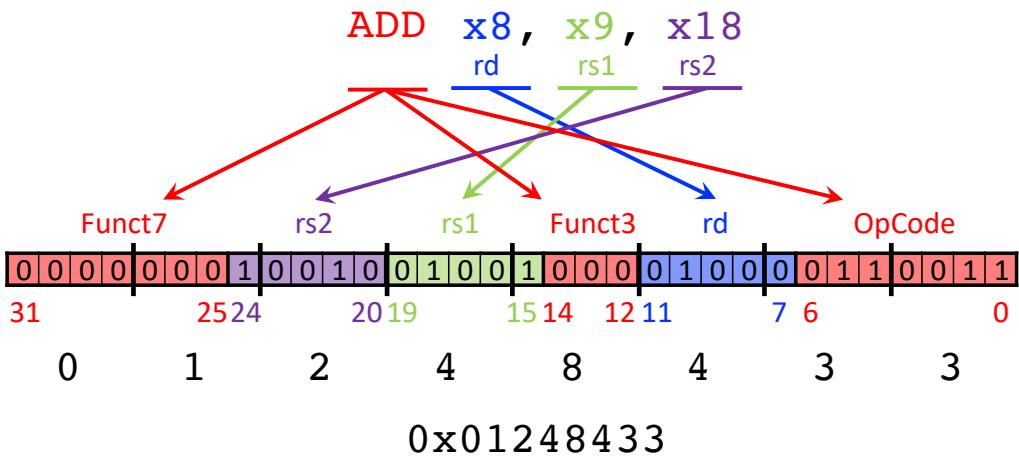


Figura 1.2: Ejemplo de instrucción tipo R.

El hecho de que la arquitectura RISC-V es modular y abierta, permite el desarrollo de módulos avanzados para aceleración de aplicaciones específicas. Un ejemplo de esto, es la definición de extensiones del conjunto de instrucciones especializadas en procesamiento vectorial con aplicación en aceleración de algoritmos de aprendizaje de máquina o aprendizaje profundo [37].

En los últimos años, ha habido un constante aumento de los requerimientos de la capacidad de cómputo dada la cantidad de datos que se procesa en los entrenamientos de las redes neuronales profundas. Esto brinda una nueva perspectiva de investigación en donde se pueda generar extensiones específicas de RISC-V que mejoren el rendimiento de estos algoritmos disminuyendo simultáneamente el consumo energético [25].

Otra área de investigación que se puede visualizar en el corto plazo es la simulación de algoritmos de computación cuántica en aceleradores por hardware. A pesar de los recientes grandes avances en el desarrollo de computadoras cuánticas, todavía hace falta aumentar drásticamente la cantidad de qubits y disminuir el ruido. Mientras se logra este objetivo, se requiere la simulación, depuración y evaluación de rendimiento de nuevos algoritmos cuánticos con aplicaciones en múltiples áreas de la ciencia. Para lograr esto se pueden visualizar extensiones de la arquitectura RISC-V que estén en capacidad de modelar de forma eficiente los fenómenos cuánticos o realizar entrenamiento de redes neuronales cuánticas [9]. Inclusive con las limitaciones que tiene los sistemas de cómputo clásico se pueden visualizar aplicaciones que permitan ser ejecutadas en un futuro en procesadores cuánticos.

1.1. Historia de la computadora

El reconocimiento del contexto histórico en un área brinda la oportunidad para entender cómo se habilitaron las condiciones para que se desarrollara una tecnología y la computadora no es la excepción. En este

capítulo se analiza el contexto geográfico e histórico del desarrollo de la computadora, haciendo un reconocimiento especial al desarrollo de sistemas de almacenamiento de información y procesamiento numérico de las culturas precolombinas, comúnmente ignoradas en la literatura de este tema. Luego se estudia el desarrollo de las ocho grandes ideas que dieron lugar a los actuales conceptos de la arquitectura de computadores y establece principios para la evaluación del rendimiento del ordenador mediante métricas unificadas [39].

1.1.1. El quipu

El quipu [6] (ver Figura: 1.3), es un mecanismo o artificio de conteo desarrollado por el imperio Inca. Este instrumento está basado en una cuerda horizontal, típicamente de lana o algodón, que sostiene otras cuerdas verticales con múltiples nudos. Según el historiador *William Burns Glyn* [27], en el quipu se realiza el proceso de conteo utilizando el número de cuerdas para representar las decenas y la cantidad de nudos para las unidades. Los colores permitían la identificación de diferentes grupos o sectores de cuerdas. Incluso, algunos especialistas contemporáneos piensan que los colores y la forma de la cuerda podrían representar un tipo de objetos, mientras que los nudos harían referencia a las cantidades de estos. Se han descubierto quipus simples con unos pocos nudos hasta unos de muy alta complejidad que llegan a tener varios miles de cuerdas. Este elemento permite almacenar números positivos y negativos, simplemente implementando una cuerda que representa el valor 0.

1.1.2. La yupana

Otro instrumento usado por los Incas fue la yupana o ábaco inca [4] (ver Figura: 1.4), el cual permite la realización de operaciones aritméticas de forma ágil, acumulando granos de maíz en celdas con unas cantidades máximas específicas que facilitaban el conteo y el acarreo. Los diferentes



Figura 1.3: Inca - quipu Creative Commons 2.0

niveles que posee la yupana permitían realizar cálculos matemáticos con números grandes [48].

1.1.3. El ábaco chino

El ábaco chino [35] (ver Figura: 1.5), es un elemento computacional de madera con barras de metal paralelas que contienen esferas móviles. La cantidad de estas esferas representan unidades, decenas, centenas, unidades de mil, decenas de miles. En este instrumento se pueden realizar operaciones básicas como sumar, restar, multiplicar y dividir, e inclusive algunas operaciones de mayor complejidad tal como las raíces, su historia se remonta a la época comprendida entre 300 a.C y 500 a.C [11].



Figura 1.4: La yupana Creative Commons 2.0

1.1.4. La pascalina

En la época moderna se introduce un elemento computacional desarrollado por el filósofo y matemático francés Blaise Pascal (1623-1662) (ver Figura: 1.7) [3], su invento más famoso: la pascalina (ver Figura: 1.6), un antepasado remoto relacionado con la computadora actual. La pascalina [44] tiene una forma parecida a una caja larga de zapatos y dentro de esta se encuentran ocho ruedas con diez dientes conectadas entre sí, generando un sistema de transmisión, donde las primeras seis ruedas representaban números enteros y las dos últimas los números decimales, formando así el sistema decimal de numeración y acarreo, de tal manera que cuando una rueda giraba completamente hacía girar un grado a la siguiente. Usando esta notación, la Pascalina podía representar valores desde 0.01 y 99999999

La pascalina no permite realizar sumas y restas de manera directa; para estas operaciones se debe girar la manivela hacia el sentido apropiado hasta encontrar la cantidad de pasos necesarios. Para realizar restas en la pascalina se utiliza el principio de complemento 9; las multiplicaciones son realizadas a través de adiciones sucesivas y las divisiones por restas sucesivas.

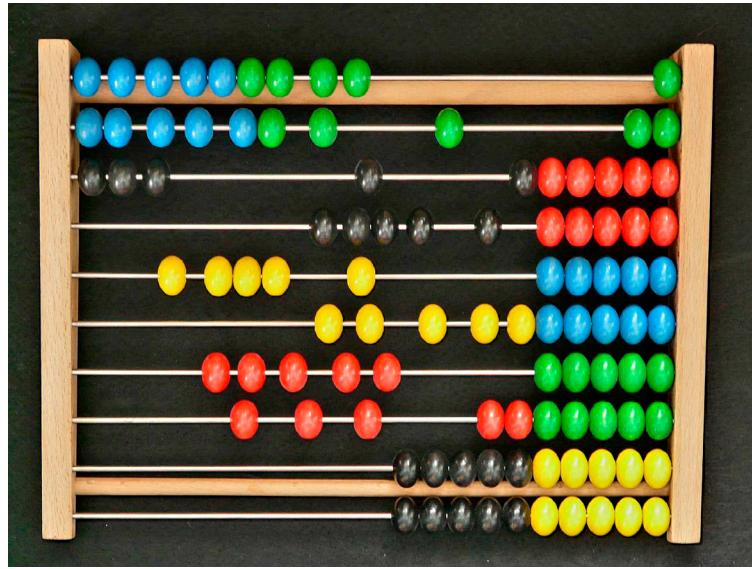


Figura 1.5: El Ábaco chino Creative Commons 2.0

1.1.5. Rueda de Leibniz

El alemán Gottfried Wilhelm von Leibniz (1646-1716) (ver Figura: 1.8) [28], matemático, filosofo, lógico, jurista, teólogo, bibliotecario, conocido como el último genio universal, inventó la rueda de Leibniz o cilindro de Leibniz (ver Figura: 1.9) [13]. Este cilindro es un tambor con un conjunto de dientes de longitud creciente, el cual se acopla a una rueda de conteo. La rueda de Leibniz fue inventada en 1672, se usó los siguientes tres siglos y fue utilizada como inspiración para una calculadora mecánica.

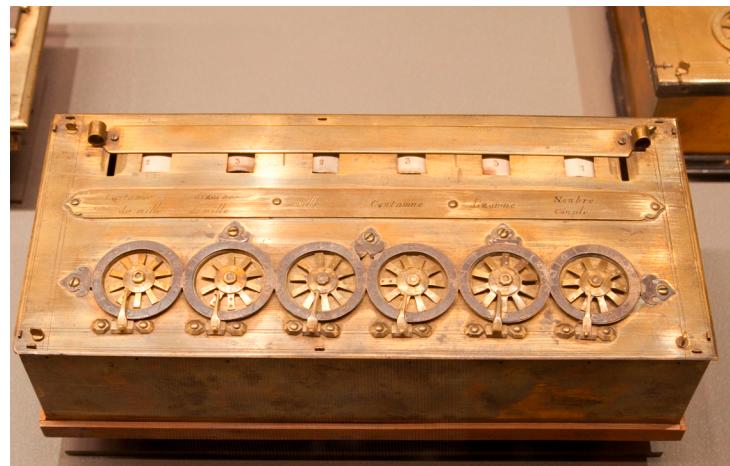


Figura 1.6: La pascalina Creative Commons 2.0



Figura 1.7: Blaise Pascal (1623-1662) Creative Commons 2.0



Figura 1.8: Gottfried Wilhelm von Leibniz (1646-1716)
Creative Commons 2.0

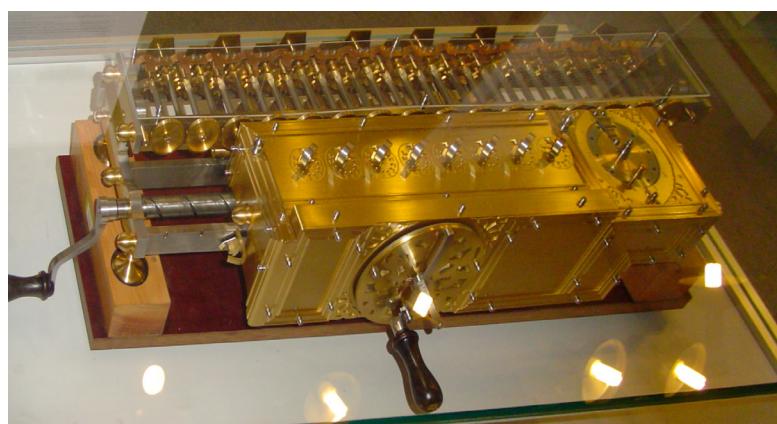


Figura 1.9: Rueda de Leibniz Creative Commons 2.0

1.1.6. Máquina analítica

El matemático Charles Babbage (1791-1871) (ver Figura: 1.10) [8], nació en Inglaterra, diseñó y desarrolló una calculadora mecánica que podía realizar cálculos de tablas sobre funciones numéricas por el método de diferencias. A Babbage se le conoce como el padre de la computadora por diseñar la primera máquina que fuese programable: la máquina analítica (ver Figura: 1.11). La cual estaba en capacidad de resolver desde funciones polinómicas hasta cálculos de tablas analíticas. El funcionamiento de la máquina analítica se basaba en el principio de las tarjetas perforadas de Jacquard (tarjetas perforadas que se usaban para crear diseños sobre telares). Esta se considera como la primera computadora de la historia y su diseño totalmente funcional fue terminado en 1835. Por falta de inversión, el proyecto de la máquina analítica no pudo continuar.

En 1840, Babbage viajó a Turín, Italia, a presentar los conceptos de diseño y operación de su máquina analítica. A esta presentación asistió Luigi Menabrea, un matemático italiano quien posteriormente fue primer ministro. Menabrea publicó en 1842 un artículo en donde describe con detalle la máquina analítica de Babbage [38]. En 1843, Lady Ada Lovelace [2], una matemática hija del poeta Lord Byron y la también matemática Lady Byron, tradujo al inglés el escrito de Menabrea y en ese proceso agregó al documento varios algoritmos que podían ser ejecutados por la máquina analítica. Esto hizo que la señora Lovelace pasará a la historia como la primera programadora de computadoras [2].

1.1.7. Colossus

El matemático británico Alan Turing (1912-1954) (ver Figura: 1.12) [17] y su estudio sobre los números y la criptografía [58], fuentes fundamentales que lo llevaron a escribir su más famoso artículo, sobre el reto de la lógica simbólica *Entscheidungsproblem* [57].

En este artículo trata de un reto de lógica simbólica en el cual se parte de un algoritmo el cual clasifica si una fórmula de cálculo lógico

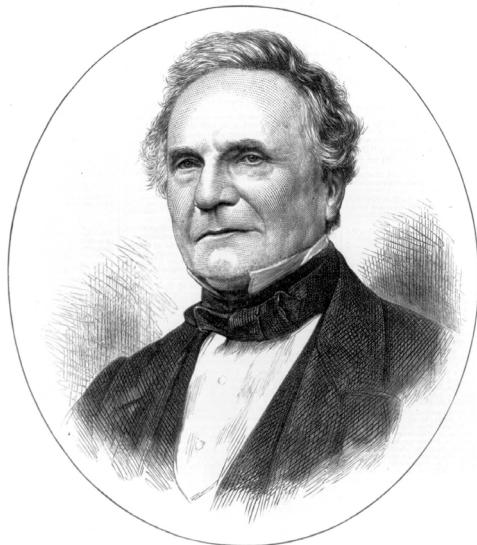


Figura 1.10: Charles Babbage (1791-1871) Creative Commons 2.0

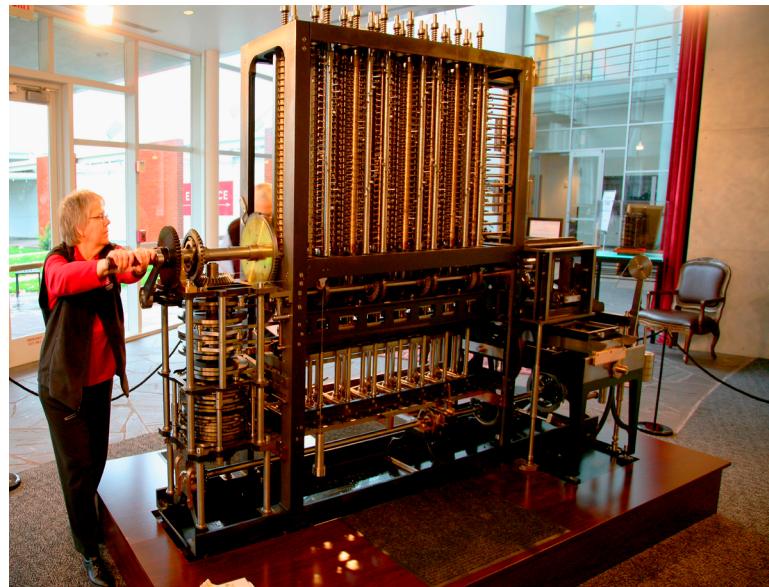


Figura 1.11: Máquina Analítica Creative Commons 2.0

puede ser un teorema o un cálculo de primer orden [10], dentro de los elementos que diseño Turing están el Colossus [15] y la Bombe [12], sistemas computacionales que posteriormente se usaron en la carrera para descifrar mensajes encriptados, el principio de funcionamiento de estos dispositivos computacionales es con base en la teoría de las máquinas de Turing.

Una máquina de Turing [26], es un dispositivo que permite entender las limitaciones de que puede ser computado. Estos dispositivos tienen una cinta mecánica, la cual contiene símbolos y además un conjunto de reglas, con las cuales podía representar cualquier tipo de lógica booleana, inicialmente esta concepción permite realizar un cálculo de entradas infinitas, ya que la cinta era circular, esto también permite asumir que la memoria de la máquina es potencialmente infinita. Además, una máquina de Turing capaz de simular cualquier otra máquina de Turing es llamada una máquina Universal.

Las máquinas colossus [49, 16, 34], fueron las primeras calculadoras electrónicas usadas por los británicos para decodificar las comunicaciones cifradas alemanas, garantizando una ventaja táctica durante la Segunda Guerra Mundial. De estas máquinas, colossus mark 2 (ver Figura: 1.13) es actualmente considerada la primera computadora electrónica digital capaz de cumplir los requisitos para ser clasificada como una máquina de Turing completa [42]. Cabe resaltar que las máquinas colossus no fueron utilizadas para descifrar los mensajes codificados por la máquina alemana enigma. Realmente, la máquina que realizó esta labor fue bombe, diseñada en 1939 por Turing, su función fue decodificar los ajustes diarios de las máquinas enigma, esto se logró a partir de generar la rotación de un conjunto de motores con los símbolos del alfabeto simulando el comportamiento de múltiples máquinas de Turing.



Figura 1.12: Alan Turing (1912-1954) Creative Commons 2.0

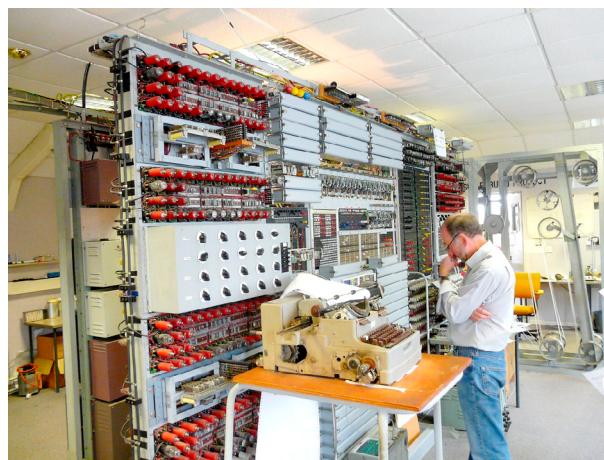


Figura 1.13: Colossus mark 2 Creative Commons 2.0

1.1.8. Z3

El ingeniero alemán Konrad Zuse (1910-1995) [62, 63], diseñó y fabricó una serie de computadoras desde 1936. La computadora Z1 (1938) y Z2 (1940) [54], fueron ensambladas en casa de sus padres, con el apoyo financiero de sus amigos. La versión Z3 [52], terminada en Berlín en 1941 y financiada por el gobierno alemán Nazi, es clasificada como la primera computadora digital electromecánica que cumplía las condiciones para ser considerada una máquina de Turing completa [51]. En esta se podían realizar operaciones básicas como sumas, restas, multiplicación, división, calcular la raíz cuadrada y además almacenar cálculos y resultados en la memoria. Su funcionamiento se da a partir de la conexión de 2300 relés telefónicos y a la reutilización de cintas cinematográficas que al ser perforadas servían como memoria para las instrucciones. En 1943, la Z3 fue destruida en un bombardeo de los aliados sobre la ciudad de Berlín, posteriormente se hizo una réplica que reposa en el Museo Alemán de esta misma ciudad. Otro aporte importante de Zuse a la computación fue la concepción del primer lenguaje de programación de alto nivel denominado el Plankalkül [53].

1.1.9. ABC

La Atanasoff Berry Computer (ABC) [7, 29, 43], fue desarrollada por el estadounidense Jhon V. Atanasoff (1903-1995) (ver Figura: 1.16), mientras cursaba su doctorado en física en la Universidad de Wisconsin. A mediados de los años 20, realizó una investigación sobre como automatizar los procesos de cálculo, haciendo uso de un sistema computacional [61].

El primer prototipo de computadora fue construido en 1939, que puso a prueba las ideas de automatizar procesos de cálculo por medio de un sistema electrónico. El segundo prototipo fue el Atanasoff-Berry Computer (ABC) (ver Figura:1.17) [14], el cual contó con la colaboración de Clifford E. Berry, un discípulo de Atanasoff y colaborador desde 1939 hasta 1942.

El ABC cumplía con un propósito específico: la resolución de sistemas



Figura 1.14: Konrad Zuse (1910-1995) Creative Commons 2.0

de ecuaciones lineales, por lo cual no era de propósito general y por tal no podía ser considerado como la primera computadora electrónica digital. Tenía funciones de cálculo en las cuales se realizaban las operaciones bit a bit y esta era independiente de la función de memoria, además innovó realizando los cálculos aritméticos en conmutadores electrónicos en vez de mecánicos. El ABC manipula números binarios, los cuales son almacenados en condensadores. Esto representaba un problema, ya que los condensadores se descargaban de manera natural, perdiendo así los datos que se guardaban. Una solución que ideó Atanasoff sobre el módulo de memoria fue hacer que mientras se realiza la lectura, también ir realizando



Figura 1.15: Z3 Creative Commons 2.0

la escritura. A esta técnica le llamo circuito refresco, que es una de las bases teóricas del funcionamiento de las memorias RAM (Random Access Memory) [19].

Uno de los mayores logros conseguidos por el ABC fue el desarrollo del circuito lógico sumador restador, el cual se llamó caja negra. Este circuito realizaba sumas y restas por medio de las reglas lógicas y estaba compuesto por válvulas termoiónicas o tubos de vacío, que permitían amplificar, conmutar, o modificar la corriente que pasaba por su interior. El sistema de entrada y salida estaba implementado con tarjetas perforadas.

1.1.10. ENIAC

John Adam Presper Eckert Jr. (1919 – 1995) [1, 20] y Jhon Mauchly (1907-1980), pioneros de la computación, desarrollaron la primera computadora electrónica de propósito general ENIAC [19, 18, 55], la cual inicio su construcción en 1943 y finalizó en 1946 financiado por el ejército de los Estados Unidos. A este contrato se le denominó Proyecto PX y tuvo una inversión de, 500000 dólares. En 1944 se unió al proyecto Jhon von Neumann, en el cual su papel principal fue generar mensajes cifrados, el grupo

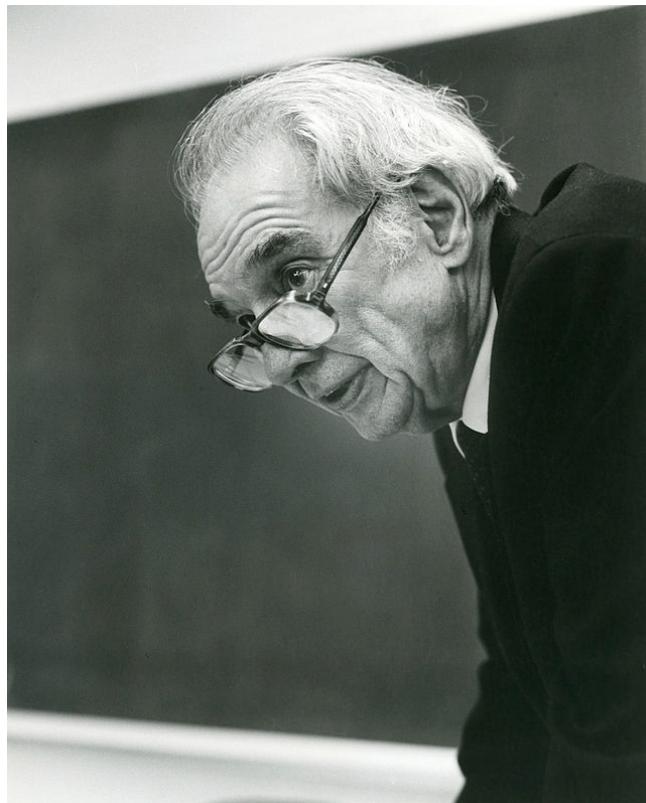


Figura 1.16: Jhon V. Atanasoff (1903-1995) Creative Commons 2.0

estaba conformado por oficiales de estados unidos, además de un grupo de científicos, este grupo se llamó los oficiales de la ENIAC (ver Figura: 1.18), en la cual se observa de izquierda a derecha J. Presper Eckert, Jr., Chief Engineer; Professor J. G. Brainerd, Supervisor; Sam Feltman, Chief Engineer for Ballistics, Ordnance Department; Captain H. H. Goldstine, Liaison Officer; Dr. J. W. Mauchly, Consulting Engineer; Dean Harold Pender, Moore School of Electrical Engineering, University of Pennsylvania; General G. M. Barnes, Chief of the Ordnance Research and Development Service; Colonel Paul N. Gillon, Chief..

La ENIAC fue una computadora electrónica digital con fines generales a gran escala y en su época fue la máquina más grande del mundo. Estaba compuesta por más de, 17000 tubos de vacío, lo cual permitía mayor velocidad de commutación en comparación con sus análogos, los relés

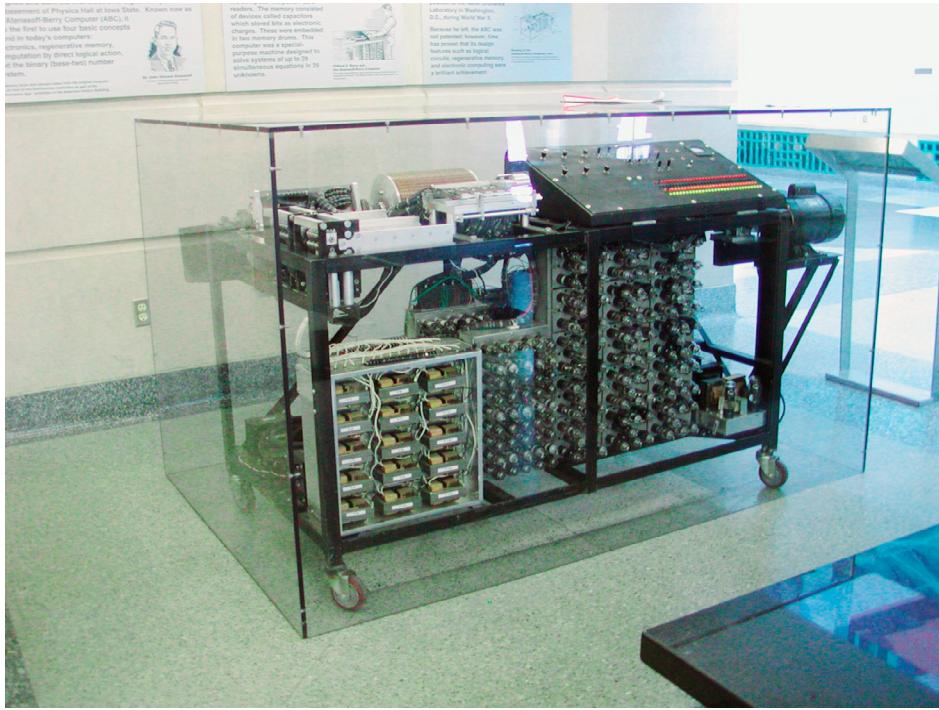


Figura 1.17: ABC Creative Commons 2.0

electromecánicos. Sin embargo, estos tubos tenían una vida útil de aproximadamente 3000 horas y ante una falla de estos se hacía muy complejo identificar el origen.

La ENIAC [21], tenía 30 unidades autónomas, de las cuales 20 se llamaban acumuladores, que podían sumar 10 dígitos a gran velocidad y almacenar sus propios cálculos. El contenido del acumulador podía visualizarse en unas pequeñas lámparas externas. El sistema utilizaba números decimales y para acelerar las operaciones aritméticas usaba un multiplicador y un divisor. El multiplicador usaba una matriz de resistencias para ejecutar las multiplicaciones de un dígito, además de un circuito de control adicional para multiplicar dígitos sucesivos y usaba acumuladores para el almacenamiento del multiplicando y los resultados parciales multiplicador. La lectura de datos se hacía por medio de una lectora de tarjetas perforadas y la salida de datos se hacía por medio de una perforadora de tarjetas.

La ENIAC se controlaba a partir de un tren de pulsos eléctrico y también era capaz de generarlos para que unidades diferentes realizaran alguna tarea, de tal manera que los programas de la ENIAC consistían en unir de forma manual los cables de las distintas unidades para que realizaran las secuencias deseadas [23]. Este proceso hacía que el cambio de programa fuera complejo y lento. Por tal motivo, en una actualización de la misma se incluyó un módulo de tabla de funciones que facilitaba la programación. Dado que en la ENIAC las unidades podían trabajar simultáneamente, los cálculos se podían realizar en forma paralela. Esta realizaba una suma en 0.2 ms, una multiplicación de dos números de 10 dígitos en 2.8 ms y una división podía tardar hasta 24 ms.

La ENIAC no funcionaba las 24 horas todos los días, al contrario, se calculaba dos veces el mismo cómputo para comprobar los resultados y se ejecutaba periódicamente cálculos cuyos resultados eran conocidos para comprobar el correcto funcionamiento de la máquina, esto permitía afinar componentes y detectar fallos. A pesar de que fue construida para fines militares, terminó siendo de gran ayuda en las investigaciones científicas. Se considera que durante su funcionamiento hasta 1955, realizó más cálculos matemáticos que todos los realizados por la humanidad anteriormente. Después de su construcción, los autores se dieron cuenta de las limitaciones a nivel estructural y de la programación, por tal motivo en forma simultánea desarrollaron nuevas ideas que dieron el origen a la estructura lógica que caracteriza las computadoras actuales.



Figura 1.18: ENIAC Officials Creative Commons 2.0

1.2. Criterios de diseño de computadoras

En esta sección se presentan ocho grandes ideas que han dado forma a la arquitectura actual de las computadoras. Estas han sido desarrolladas por diferentes científicos y personajes históricos, durante los últimos 60 años y constituyen una gran ayuda para entender, analizar, estudiar y aplicar las estrategias que se tomaran en las siguientes secciones.

1.2.1. Diseñar teniendo en cuenta la ley de Moore

Gordon Moore, visionario y cofundador de Intel, afirmó que la escala de integración de los circuitos encapsulados se duplicaría aproximadamente entre cada 18 a 24 meses. Esta escala de integración se refiere a la cantidad de transistores por unidad de área con la que podía fabricarse un circuito integrado (ver Figura: 1.19) [40]. A este concepto posteriormente se le denominó la ley de Moore y logró predecir durante muchos años de forma muy acertada el crecimiento de la industria de los semiconductores. El diseño de un procesador puede tomar varios años, es por tal motivo que los ingenieros deben anticipar cuál es el contexto de la tecnología (escala de integración) de acuerdo a la ley de Moore para cuando el proceso de diseño termine. La única constante para los diseñadores de computadoras es el cambio rápido [41, 45] .

1.2.2. Usar la abstracción para simplificar el diseño

La esencia de la abstracción es preservar la información que es relevante en un contexto dado y descartar la información que es irrelevante [36].

Los ingenieros y arquitectos de hardware y software han desarrollado diferentes estrategias y técnicas para ser más productivos para facilitar y reducir el tiempo de diseño. Una de las técnicas para simplificar los diseños es encapsular los diferentes niveles de un determinado sistema computacional; esto quiere decir que entre los niveles de abstracción se crea una jerarquía, en donde el nivel más bajo hace referencia a estar más

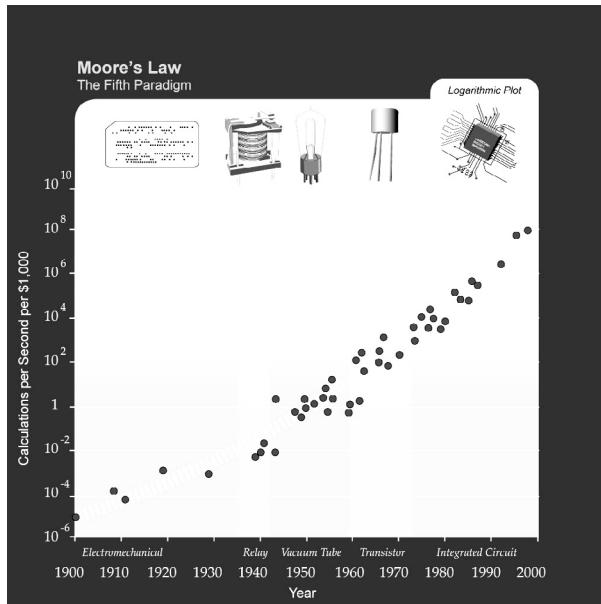


Figura 1.19: Ley de Moore Creative Commons 2.0

cerca del hardware. Las características son los diferentes elementos que lo componen y en la medida en que el nivel de abstracción sea alto, dichos detalles se hacen irrelevantes [45].

1.2.3. Reducir el tiempo de ejecución para los casos más comunes

Supongamos que una tarea está compuesta por dos procesos. El A es ejecutado mil veces y el B diez veces y ambos toman aproximadamente el mismo tiempo de ejecución. Tiene un mayor impacto en el rendimiento general tomar decisiones de arquitectura que tiendan a reducir el tiempo de ejecución del proceso A inclusive si esas decisiones aumentan un poco el tiempo de ejecución del proceso B. A lo largo de los próximos capítulos se podrán observar ejemplos de estas decisiones de arquitectura.

1.2.4. Mejorar rendimiento a través del paralelismo

El concepto de ejecución en paralelo hace referencia a múltiples unidades de procesamiento ejecutando una tarea sobre diferentes datos de forma simultánea. Como ejemplo básico se toma la suma de dos vectores de tamaño n . Se asume que esta tarea tiene un tiempo de ejecución de t en una sola unidad de procesamiento. En principio y sin entrar en detalles de paralelización, se puede dividir esta tarea en dos procesos que se ejecutan de forma simultánea por diferentes unidades de procesamiento, cada uno de ellos realizado la suma de la mitad de los elementos de los vectores. Bajo este esquema, en el entorno ideal el tiempo de ejecución sería $t/2$ s. En general, si hay m unidades de procesamiento y sin tener en cuenta sobre costos del paralelismo, se dice que la tarea tarda en ejecutarse t/m s.

1.2.5. Mejorar rendimiento a través de la segmentación

Se asume que se tiene una línea de producción de objetos dividida en tres etapas. En la primera etapa E_1 se traen los componentes para el ensamblado. En la segunda etapa E_2 se realiza el ensamblado del objeto. Finalmente, en tercera etapa E_3 se realiza el acabado externo del objeto y su almacenamiento. Un primer enfoque de este proceso para la producción de un objeto O_1 será primero ejecutar la etapa E_1 , luego la etapa E_2 y se finaliza con la etapa E_3 . A partir de este momento se inicia la producción de un objeto O_2 (ver Figura: 1.20).

En un enfoque más eficiente, un enfoque segmentado, se pretende disminuir el tiempo en espera de las etapas. Usando el ejemplo previo, esto se logra iniciando la producción del objeto O_2 cuando el objeto O_1 esté iniciando la etapa E_2 . Posteriormente, se puede iniciar la producción de un objeto O_3 cuando el objeto O_2 haya llegado a E_2 y el objeto O_1 a la etapa E_3 (ver Figura: 1.21).

Si en el primer escenario se logra producir un objeto cada t segundos y suponiendo que las etapas que se implementan en la segmentación toman aproximadamente el mismo tiempo, se logra que en el segundo escenario se

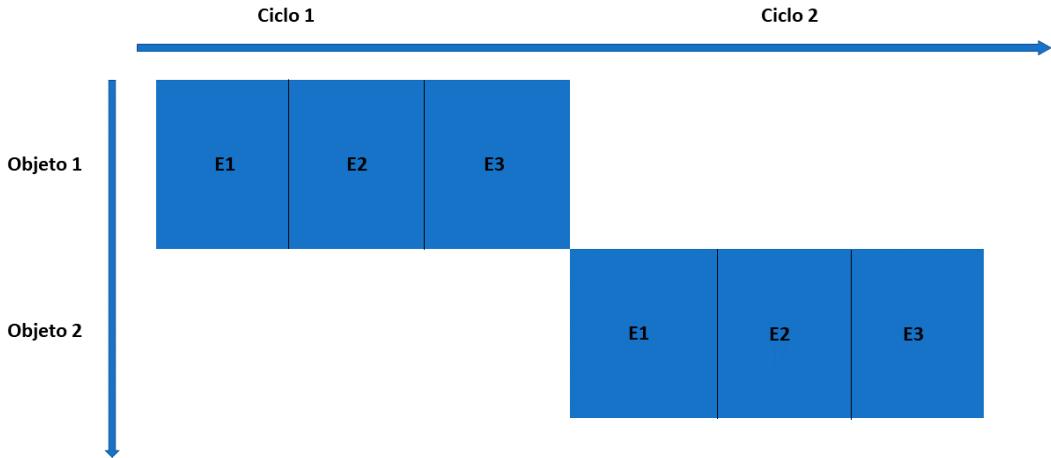


Figura 1.20: Tarea no optimizada.

produzca un objeto cada $t/3$ segundos. Un proceso equivalente se realiza en la optimización de sistemas digitales, dividiéndolo en etapas aproximadamente balanceadas y realizando diferentes partes del proceso sobre diferentes datos.

1.2.6. Mejorar rendimiento a través de la predicción de saltos

Los saltos o bifurcaciones algorítmicas son procesos fundamentales en la ejecución de programas para tomar decisiones basados en una determinada condición. Adicionalmente, Estas bifurcaciones son esenciales para crear ciclos que se repiten siempre y cuando un estado específico esté presente. Poder predecir el resultado de la evaluación de estas condiciones permite que se pueda anticipar los procesos que continúan después en los posibles escenarios que surgen de la bifurcación. Estas predicciones permiten mejorar el rendimiento disminuyendo el tiempo de ejecución total

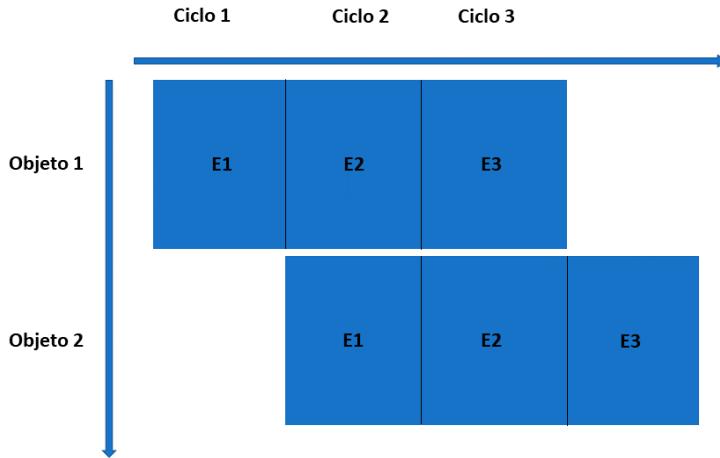


Figura 1.21: Tarea segmentada.

de un programa.

1.2.7. Jerarquía de memorias

La memoria es el componente fundamental en la organización de un computador en donde se almacenan los datos y los programas. El tiempo de respuesta, la capacidad y el costo son algunas de las características principales asociadas a la memoria. Algunas de estas características son inversamente proporcionales, por ejemplo, mientras más bajo es el tiempo de respuesta, más alto el costo de la memoria y en consecuencia no se puede tener mucha cantidad. Esto lleva a que la memoria deba ser organizada en una jerarquía en donde se llegue a un balance entre costo y rendimiento, involucrando diferentes tipos de tecnología. En esta jerarquía se busca que la memoria de menor tiempo de respuesta sea la más cercana a las unidades de procesamiento, mientras que la más lenta y de menor precio esté más alejado, pero permita almacenar gran cantidad de datos a bajo costo.

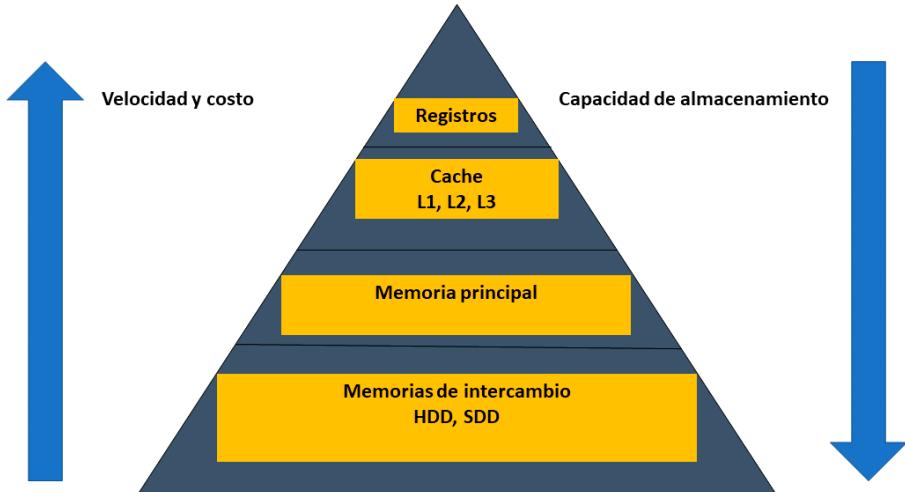


Figura 1.22: Jerarquía de Memorias.

1.2.8. Mejorar fiabilidad a través de redundancia

Todos los sistemas presentan una probabilidad de fallar y un sistema de cómputo no es la excepción. La fiabilidad está definida como la cualidad de un sistema de comportarse en la forma requerida o esperada bajo unas condiciones específicas y durante un tiempo determinado de operación. Para mejorar la fiabilidad de los sistemas de comunicación, almacenamiento o procesamiento de datos se usa el concepto de redundancia. La redundancia hace referencia a tener múltiples sistemas operando de forma simultánea sobre los mismos datos, de tal forma que si alguno o algunos de los sistemas fallan se puede garantizar que la tarea sea completada sin dificultades por los demás que se conservan en correcto funcionamiento. Un ejemplo de esto se puede apreciar en el almacenamiento de bits en una unidad de disco duro en donde se puede evitar perdida de información o errores de lectura guardando la misma información en múltiples unidades, a este tipo de grupo de discos se le denomina comúnmente como RAID (Redundant Array of Independent Disks) (ver Figura: 1.23).



Figura 1.23: RAID Creative Commons 2.0

1.3. Evaluación de rendimiento de la computadora

El rendimiento de una computadora está asociado a que tan eficiente es ejecutando una tarea. Una caracterización holística de rendimiento de una computadora debe incluir tiempo, energía y recursos utilizados. Sin embargo, bajo el contexto de los próximos capítulos se definirá el rendimiento específicamente en términos del tiempo de ejecución de un programa X en una computadora A . Es importante tener en cuenta un programa específico X dado que el rendimiento de esa misma computadora A puede ser diferente para un programa Y . Realizamos entonces las siguientes definiciones para establecer el rendimiento de una computadora:

$$R(X, A) = \frac{1}{T_{\text{ex}}(X, A)} \quad (1.1)$$

en donde R es el rendimiento para el programa X en la computadora A y T_{ex} es el tiempo de ejecución efectivo que tarda el programa en ser totalmente ejecutado.

A partir de esto podemos establecer también la ecuación básica de

rendimiento:

$$T_{\text{ex}}(X, A) = I_{\text{eff}}(X) \cdot CPI(X, A) \cdot T_{\text{clk}}(A) \quad (1.2)$$

en donde $I_{\text{eff}}(X)$ es el número efectivo de instrucciones que se requieren ejecutar para completar el programa, X incluyendo todas las instrucciones que se ejecutan múltiples veces en los ciclos, $CPI(X, A)$ es el número promedio de ciclos que le toma a una instrucción del programa X en ser ejecutada en la computadora A y $T_{\text{clk}}(A)$ es el periodo de la señal de reloj de la computadora A el cual es el inverso de la frecuencia

$$T_{\text{clk}}(A) = \frac{1}{f_{\text{clk}}(A)} \quad (1.3)$$

Otra forma de analizar el número total de ciclos de reloj requeridos por un programa se puede realizar con la siguiente ecuación

$$C(X, A) = \sum_i CPI_i(X, A) \cdot I_i(X) \quad (1.4)$$

En donde i hace referencia a un tipo de instrucción, $CPI_i(X, A)$ es el número de ciclos de reloj que le toma a las instrucciones tipo i en ser ejecutadas en la computadora A y $I_i(X)$ es el número total de instrucciones ejecutadas efectivas tipo i para el programa X .

La comparación del rendimiento de dos computadoras A y B se debe realizar para un programa X de la siguiente forma

$$R(X, A) > R(X, B) \quad (1.5)$$

$$\frac{1}{T_{\text{ex}}(X, A)} > \frac{1}{T_{\text{ex}}(X, B)} \quad (1.6)$$

$$T_{\text{ex}}(X, B) > T_{\text{ex}}(X, A) \quad (1.7)$$

Adicionalmente, se puede hallar la relación entre ambos rendimientos para obtener la aceleración del programa X por la computadora A teniendo como base la computadora B

$$\text{Aceleración} = \frac{R(X, A)}{R(X, B)} = \frac{T_{\text{ex}}(X, B)}{T_{\text{ex}}(X, A)} \quad (1.8)$$

1.4. Ejercicios Capítulo 1

1. Mencione cinco ventajas que brinda la arquitectura de instrucciones reducida RISC-V.
2. ¿Cuáles son los aspectos más importantes de la historia de la computación?
3. Realice una tabla con los años de invención de cada elemento computacional, resaltando las características relevantes de cada invento.
4. Realizar un análisis de la tabla realizada anteriormente.
5. ¿De qué manera ve reflejada la ley de Moore en la actualidad?
6. Dar tres ejemplos de cómo la abstracción permite simplificar diseños?
7. Planteé un modelo que sea resultado de la unión del paradigma del paralelismo y la segmentación.
8. Se asume que se tiene un problema en el cual se debe almacenar una gran cantidad de información. ¿Qué memoria es la más apropiada para abordarlo?
9. Dar cinco ejemplos de redundancia en sistemas computacionales.
10. Cuál es el rendimiento de una computadora con un procesador de 2500 Hz ejecutando un programa de 900 instrucciones.

11. Investigue los diferentes tipos de lenguajes de alto nivel según sus formas de interpretación. (lenguaje compilado, interpretado).
12. ¿Por qué instancia pasa el lenguaje de alto nivel para llegar al lenguaje máquina?
13. ¿Cuáles son las características más importantes de la máquina analítica?
14. ¿Cuáles son los tipos de instrucciones definidas en la arquitectura RISC-V?
15. ¿De qué manera permite la Pascalina realizar sumas y restas?
16. Describir las funcionalidades del Bombe. pp
17. ¿Qué característica permite medir la máquina de Turing?
18. ¿Por qué razón el ABC no puede ser considerada la primera computadora electrónica digital?
19. ¿Cuáles fueron los elementos de los que estaba compuesta la caja negra del ABC?
20. ¿Qué operaciones permitía realizar la ENIAC?
21. En el caso de la máquina ENIAC, ¿cuál es la cantidad de operaciones como sumas, multiplicaciones, divisiones y raíces cuadradas que realizaba por segundo?
22. ¿Cuál es el nombre del primer lenguaje de programación y por quien fue creado?

2

CAPÍTULO DOS

Instrucciones

La arquitectura RISC-V, una arquitectura de conjunto de instrucciones (ISA) de código abierto, ha ganado una atención significativa en la comunidad de la informática y la electrónica debido a su simplicidad, eficiencia y flexibilidad. RISC-V es una arquitectura de tipo RISC (Reduced Instruction Set Computer), lo que significa que se basa en un conjunto reducido de instrucciones simples que se pueden combinar para realizar operaciones complejas. Este enfoque contrasta con las arquitecturas de tipo CISC (Complex Instruction Set Computer), que incluyen un gran número de instrucciones complejas en su conjunto de instrucciones.

El conjunto de instrucciones RV32 de RISC-V es una de las variantes de esta arquitectura, diseñada para sistemas con registros de 32 bits. Este conjunto de instrucciones proporciona las operaciones básicas que necesita un procesador para ejecutar programas de alto nivel, incluyendo operaciones aritméticas, operaciones lógicas, operaciones de carga y almacenamiento, y operaciones de control de flujo.

Un conjunto de instrucciones es el resultado de un proceso de compilación de un programa en alto nivel como C, las cuales están representadas en dos niveles de lenguaje, que son el lenguaje ensamblador y el lenguaje de máquina. El primero se compone de un conjunto de instrucciones reducidas y el segundo es la representación binaria de cada instrucción,

es en este nivel donde el procesador la ejecuta. Además, las instrucciones son el elemento que brinda la dimensión de la arquitectura, al punto que actualmente las computadoras tienen instrucciones de 32 bits o 64 bits. Cada instrucción está dividida en diferentes formatos, los cuales son (ver Figura: 2.1):

- Instrucciones aritmético lógicas
 - Tipo Register
 - Tipo Immediate
- Instrucciones saltos condicionados
- Instrucciones de memoria
 - Tipo I lectura
 - Tipo S Escritura
- Instrucciones saltos incondicionales.
 - Tipo I jal
 - Tipo J jalr
- Instrucciones auxiliares
- Pseudo instrucciones

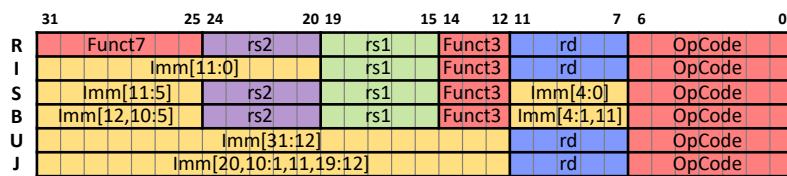


Figura 2.1: Tipos de instrucciones.

Registros: RISC-V al ser un set de instrucciones reducido, posee 31 registros (ver 2.1), que van desde el x1 hasta el x31. Dentro de este conjunto

de registros cuenta con algunos de propósito específico como **zero** (x0), el cual siempre tiene como valor un 0 lógico y no se puede modificar, lo cual es útil para inicializar otros registros. Se tiene el registro **ra** (x1), que es utilizado para almacenar la dirección de retorno después de una llamada función, el registro **sp** (x2) donde se almacena la dirección base del segmento de memoria separado para la pila. cada registro tiene una longitud de 32 bits, por lo cual se tiene una matriz de 32x32, donde cada registro puede ser accedido por medio de una dirección específica de 5 bits relacionada con el índice de registro, siendo 0 el valor para llamar al registro x0, y 31 el valor para identificar el registro x31.

Se asume que se tiene un elemento de 32 bits llamado registro:

registro [31:0]

Sí, se desea tomar los 7 bits menos significativos del registro:

registro [6:0]

Esta nomenclatura será usada en el resto del libro.

Registro	Nemónico	Descripción	Preservado
x0	zero	Siempre Cero	—
x1	ra	Dirección de retorno	<i>Caller</i>
x2	sp	Puntero a la pila	<i>Callee</i>
x3	gp	Puntero global	—
x4	tp	Puntero hilo	—
x5	t0	Registro temporal/alternativo	<i>Caller</i>
x6-x7	t1-t2	Temporales	<i>Caller</i>
x8	s0 / fp	Registro de guardado/Puntero de cuadro	<i>Callee</i>
x9	s1	Registro de guardado	<i>Callee</i>
x10-x11	a0-a1	Argumentos de función/valores de retorno	<i>Caller</i>
x12-x17	a2-a7	Argumentos de función	<i>Caller</i>
x18-x27	s2-s11	Registros de guardado	<i>Callee</i>
x28-x31	t3-t6	Temporales	<i>Caller</i>

Tabla 2.1: Tabla de registros.

2.1. Instrucciones aritmético lógicas

2.1.1. Instrucciones tipo R

Las instrucciones tipo R son aquellas en las que se realiza una operación aritmética o lógica sobre el contenido de dos registros y el resultado de esta operación es almacenado en un tercer registro.

Las operaciones aritméticas son la suma y la resta, y las operaciones lógicas son la and, or y xor, donde se realizan bit a bit sobre los contenidos de los operandos.

Está instrucción está formada por diferentes campos, los cuales son las direcciones de dos registros fuente (`rs1` y `rs2`) de 5 bits cada uno, la dirección de un registro destino (`rd`) de 5 bits, un código de operación de 7 bits y dos campos de descripción de operación llamados `Funct3` y `Funct7` de 3 y 7 bits respectivamente que se combinan, formando un campo de 10 bits. A continuación se presenta una lista detallada de los campos de las instrucciones tipo R.

- **OpCode (Inst[6:0]):** Codifica el Tipo Instrucción.
- **rd (Inst[11:7]):** Recibe la dirección del registro donde va a ser almacenado el resultado de la operación entre los registros fuente uno y dos.
- **Funct3 (Inst[14:12]), Funct7 (Inst[31:25]):** Estos campos combinados con el OpCode describen la operación a realizar (ver tabla: 2.2).
- **rs1 (Inst[19:15]):** Dirección del registro fuente 1 u operando A.
- **rs2 (Inst[24:20]):** Dirección del registro fuente 2 u operando B.

Las operaciones que permite realizar este formato están especificadas en la figura (ver tabla: 2.2). De esta manera funciona cada instrucción:

	Tipo R		
	OpCode	Funct3	Funct7
add	0110011	000	0000000
sub	0110011	000	0100000
sll	0110011	001	0000000
slt	0110011	010	0000000
sltu	0110011	011	0000000
xor	0110011	100	0000000
srl	0110011	101	0000000
sra	0110011	101	0100000
or	0110011	110	0000000
and	0110011	111	0000000

Tabla 2.2: Códigos de una instrucción tipo R.

- **add:** Suma (Addition)

$$RU[rd] = RU[rs1] + RU[rs2]$$

- **sub:** Resta (Subtraction)

$$RU[rd] = RU[rs1] - RU[rs2]$$

- **slt:** Activar si menor qué (Set on less than)

$$RU[rd] = RU[rs1] < RU[rs2]$$

- **sltu:** Activar si menor qué sin signo (Set on less than unsigned)

$$RU[rd] = RU[rs1] < RU[rs2]$$

- **sll:** Corrimiento lógico a la izquierda (Shif left logical)

$$RU[rd] = RU[rs1] \ll RU[rs2]$$

- **srl:** Corrimiento lógico a la derecha (Shif right logical):

$$RU[rd] = RU[rs1] \gg RU[rs2]$$

- **sra:** Corrimiento aritmético a la derecha (Shif right arithmetic)

$$RU[rd] = RU[rs1] \gg RU[rs2]$$

- **and:** Operación lógica bit a bit AND

$$\text{RU}[rd] = \text{RU}[rs1] \ \& \ \text{RU}[rs2]$$

- **or:** Operación lógica bit a bit OR

$$\text{RU}[rd] = \text{RU}[rs1] \mid \text{RU}[rs2]$$

- **xor:** Operación lógica bit a bit XOR

$$\text{RU}[rd] = \text{RU}[rs1] \oplus \text{RU}[rs2]$$

Ejemplo instrucciones tipo R

Se asume que se va a realizar la suma de dos variables enteras g , h de 32 bits cada una, donde el resultado se guarda en la variable entera f .

A continuación se presenta un código en C:

$$f = g + h;$$

Se asume que las variables enteras f , g , h serán almacenados en los registros de guardado (ver Tabla: 2.1) $x8$, $x9$, $x18$. Se asignan los registros de la siguiente manera:

$$\begin{aligned} x8 &\leftarrow f \\ x9 &\leftarrow g \\ x18 &\leftarrow h \end{aligned}$$

A continuación se presenta el código en lenguaje ensamblador RISC-V:

add x8, x9, x18

En el siguiente ejemplo se realizan dos restas y una suma en una sola sentencia. Se asume que todas las variables han sido previamente declaradas como enteros de 32 bits **int**.

$$f = (g - h) + (i - j);$$

Se realiza la asignación de las variables **f**, **g**, **h**, **i** y **j** a registros asociados a variables de la siguiente manera:

```
x8 ← f
x9 ← g
x18 ← h
x19 ← i
x20 ← j
```

En el proceso de pasar a lenguaje de bajo nivel (Low Level Language LLL), se debe almacenar la resta de **g** + **h** y de **i** - **j** en registros temporales (Temporaries. Ver 2.1), para posteriormente realizar la suma.

A continuación se presenta el código en lenguaje ensamblador RISC-V:

```
sub x5, x9, x18
sub x6, x19, x20
add x8, x5, x6
```

Lenguaje de bajo nivel

En la figura (ver Figura: 2.2), se observa una instrucción en formato tipo R, en lenguaje ensamblador y su traducción en formato binario. También el proceso de redacción, desde ensamblador hasta binario para esta instrucción.

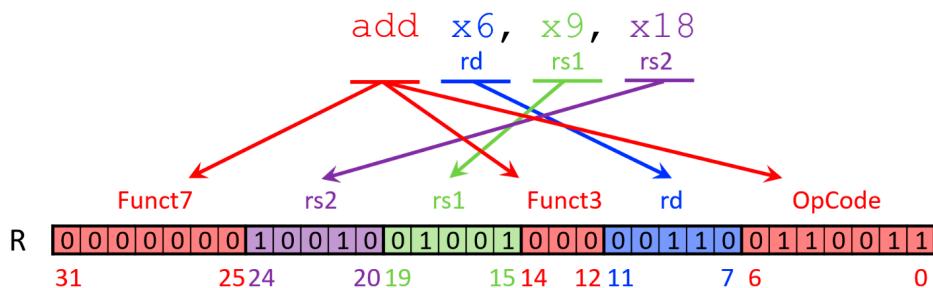


Figura 2.2: Ejemplo código de instrucción tipo R.

2.1.2. Instrucciones tipo I

Este tipo de instrucciones permite realizar una operación aritmética o lógica sobre el contenido de un registro y un valor entero, llamado inmediato, el cual tiene un dominio entre $[-2^{11}, 2^{11}]$. las instrucciones tipo I se usan para realizar corrimientos lógicos (**slli**, **slti**), con los cinco bits menos significativos del inmediato. El formato está conformado por las direcciones de un registro fuente (**rs1**) de 5 bits, una dirección de un registro destino (**rd**) de 5 bits, un código de operación de 7 bits, un campo **Funct3** de 3 bits y un campo de 11 bits, que representa el inmediato.

- **OpCode** (**Inst[6:0]**): Este campo codifica el formato de la instrucción que se ejecuta. Varía entre el formato-I aritmético-lógico y el formato-I para carga.
- **rd** (**Inst[11:7]**): Este campo recibe la dirección del registro donde va a ser almacenado el resultado de la operación.
- **Funct3** (**Inst[14:12]**): Este campo combinado con el **OpCode** describen la operación a realizar (ver Figura: 2.3).
- **rs1** (**Inst[19:15]**): Dirección del registro fuente 1.
- **Imm** (**Inst[31:20]**): Campo que almacena el valor inmediato, que puede representar valores entre el intervalo de: $[-2^{11}, 2^{11}]$.

- **addi**: Suma con inmediato (Addition immediate)

$$\text{RU}[rd] = \text{RU}[rs1] + \text{Imm}$$
- **slti**: Activar menor qué inmediato (Set less than immediate)

$$\text{RU}[rd] = \text{RU}[rs1] < \text{Imm}$$
- **sltiu**: Activar menor qué sin signo inmediato (Set less than immediate unsigned)

$$\text{RU}[rd] = \text{RU}[rs1] < \text{Imm}$$

Tipo I		
	OpCode	Funct3
addi	0010011	000
slli	0010011	001
slti	0010011	010
sltiu	0010011	011
xori	0010011	100
srl	0010011	101
srai	0010011	101
ori	0010011	110
andi	0010011	111

Tabla 2.3: Códigos de una instrucción tipo I.

- **slli**: Corrimiento lógico a la izquierda inmediato (Shift left logical immediate)

$$RU[rd] = RU[rs1] \ll Imm$$
- **srl**: Corrimiento lógico a la derecha inmediato (Shift right logical immediate)

$$RU[rd] = RU[rs1] \gg Imm$$
- **srai**: Corrimiento aritmético a la derecha inmediato (Shift right arithmetic immediate)

$$RU[rd] = RU[rs1] \gg Imm$$
- **andi**: Operación lógica bit a bit AND inmediato (AND immediate)

$$RU[rd] = RU[rs1] \& Imm$$
- **ori**: Operación lógica bit a bit OR inmediato (OR immediate)

$$RU[rd] = RU[rs1] \mid Imm$$
- **xori**: Operación lógica bit a bit XOR inmediato (XOR immediate)

$$RU[rd] = RU[rs1] \text{ XOR } Imm$$

Ejemplo instrucciones tipo I

Para demostrar el uso de inmediatos se asume que las variables enteras f y g son registros de longitud 32 bits. Despu s, se realiza la suma del contenido del registro uno m s un inmediato de valor decimal 10, el resultado se almacena en f .

Adicionalmente, para la demostraci n de los corrimientos l gicos con inmediatos se realiza una multiplicaci n por 2, teniendo en cuenta que los corrimientos a la izquierda son multiplicaciones por potencias de 2.

Un corrimiento tambi n es definido como la cantidad de bits que se desplaza hacia la derecha o a la izquierda en una palabra, se debe tener en cuenta si es un corrimiento aritm tico, ya que en esta operaci n los bits de corrimiento en la palabra se asignan dependiendo del valor del bit m s significativo de la palabra.

Se asignan los registros de la siguiente manera:

```
x8 ← f
x9 ← g
```

A continuaci n se presenta un c digo en C:

```
f = g + 10; // Se realiza la suma de g + 10.
f = g * 8; // Multiplicaci n con potencia de 2.
```

Se asume que las variables enteras f , g ser n almacenadas en los registros $x8$, $x9$, donde $x8$ almacena la suma de $i + 10$. Adem s, se realiza el corrimiento a la izquierda de 3 bits, de acuerdo a la regla de corrimientos a la izquierda $8 = 2^3$.

A continuaci n se presenta el c digo en lenguaje ensamblador RISC-V:

```
addi x8, x9, 10; x8 almacena la suma de x9 + 10
slli s3, S3, 3 ; Corrimiento de 3 bits a la izquierda
```

Las operaciones de corrimiento se realizan sobre el registro fuente 1 con la cantidad que representan los 5 bits menos significativos del inmediato.

Lenguaje de bajo nivel

En la figura (ver Figura: 2.3), se observa una instrucción en formato I, lenguaje ensamblador y su traducción en formato binario.

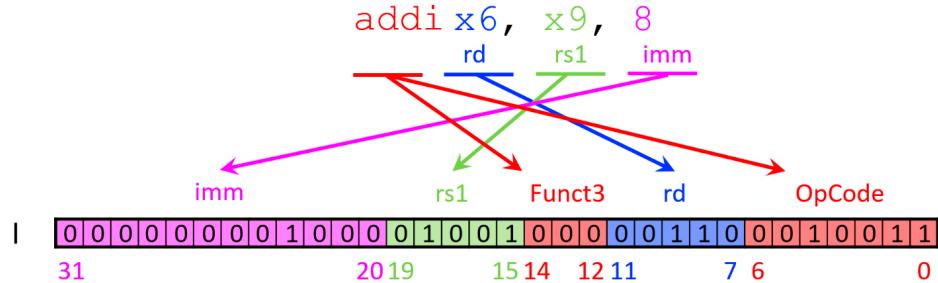


Figura 2.3: Ejemplo código de instrucción tipo I.

Nota: Los corrimientos se denotan con los símbolos ($>>$), para desplazamientos a la derecha y para desplazamientos a la izquierda el símbolo ($<<$).

2.2. Instrucciones de memoria de datos

2.2.1. Instrucciones de lectura de memoria tipo I

Las instrucciones tipo I también sirven para cargar registros en memoria. El inmediato se suma a la dirección base del registro fuente 1 (rs1) y se realiza la carga sobre el registro destino [5].

- **lw:** Leer palabra de memoria (Load word)

$$RU[rd] = RU[rs1 + imm](31:0)$$

- **lb:** Leer byte (Load byte)

$$RU[rd] = RU[rs1 + imm](7:0)$$

- **lh:** Leer media palabra (Load halfword)

$$RU[rd] = RU[rs1 + imm](15:0)$$

- **lbu**: Leer byte sin signo (Load byte unsigned)

$$RU[rd] = RU[rs1 + imm](7:0)$$
- **lhu**: Leer media palabra sin signo (Load halfword unsigned)

$$RU[rd] = RU[rs1 + imm](15:0)$$

Lectura de memoria tipo I

	OpCode	Funct3
lb	0000011	000
lh	0000011	001
lw	0000011	010
lbu	0000011	100
lhu	0000011	101

Tabla 2.4: Códigos las instrucciones de lectura de memoria tipo I.

En la tabla 2.4 se muestran los diferentes valores de los registros para cada instrucción tipo I de lectura.

Ejemplo instrucciones tipo I

Se asume que se tiene un puntero a la dirección de un vector de enteros llamado V del cual se requiere acceder al elemento 10. La asignación de variables queda de la siguiente manera:

```
x8 ← x
x9 ← V
```

A continuación se presenta un código en C:

```
x = V[10]; // x es igual al elemento 10 de V
```

Se asume que la dirección donde inicia el vector de enteros V está almacenada en el registro **x9** y la variable entera que almacenará el valor está almacenada en el registro **x8**.

A continuación se presenta el código en lenguaje ensamblador RISC-V:

```
lw x8, 40(x9); Instrucción Load Word  
; Carga 4 bytes
```

El valor inmediato en este caso es 10, valor que será usado para sumarse a la dirección que es contenida en el registro **x9**, para encontrar el contenido efectivo que será cargado en **x8**.

Para llevar esta instrucción a bajo nivel se hace directamente como es definido en la sección de LLL para instrucciones tipo I aritmético lógicas 2.1.2.

2.2.2. Instrucciones de escritura en memoria tipo S

Estas instrucciones permiten modificar el contenido de la memoria, con el objetivo de almacenar información en ella. Los datos que se necesitan son, el registro fuente 1 (**rs1**), que tiene la dirección donde se va a almacenar en la memoria de datos, un registro fuente 2 (**rs2**) que contiene la información que se desea almacenar en la memoria de datos y finalmente posee un inmediato que permite desplazar la dirección del registro fuente 2 (**rs2**). Se pueden realizar diferentes tipos de almacenamiento. Dentro de los tipos de almacenamiento se tiene almacenar un byte, almacenar media palabra (dos bytes) y almacenar una palabra (4 bytes).

- **sb**: Almacenar un byte (Store byte)
RU[**rs1** + **imm**] (7:0)
- **sh**: Almacenar media palabra (Store halfword)
RU[**rs1** + **imm**] (15:0)
- **sw**: Almacenar una palabra (Store word)
RU[**rs1** + **imm**] (31:0)

Escritura en memoria tipo S

	OpCode	Funct3
sb	0100011	000
sh	0100011	001
sw	0100011	010

Tabla 2.5: Códigos para las instrucciones de escritura en memoria tipo S.

En la tabla 2.5 se muestran los diferentes valores de los registros para cada instrucción tipo S.

Ejemplo instrucciones formato S

Se asume que se tiene un puntero a la dirección de un vector de enteros llamado V, en el cual se almacenará un valor entero x, los registros son asignados así:

```
x8 ← x
x9 ← V
```

A continuación se presenta un algoritmo en C:

```
V[12] = x; // V[12] almacena el valor del entero x
```

Se asume que la dirección base del vector de enteros V está almacenada en el registro x9 y la variable entera x, representa dato que se almacenará, está en el registro x8.

A continuación se presenta el código en lenguaje ensamblador RISC-V:

```
sw x8, 48(x9); Instrucción Store Word
; almacena en la memoria de datos
```

El valor de desplazamiento en este caso es 12, valor que será usado para sumarse a la dirección de x9 y almacenar el contenido efectivo que está guardado en el registro x8.

Lenguaje de bajo nivel

En la figura (ver Figura: 2.4), se observa una instrucción de formato tipo S en lenguaje ensamblador y su traducción en formato binario.

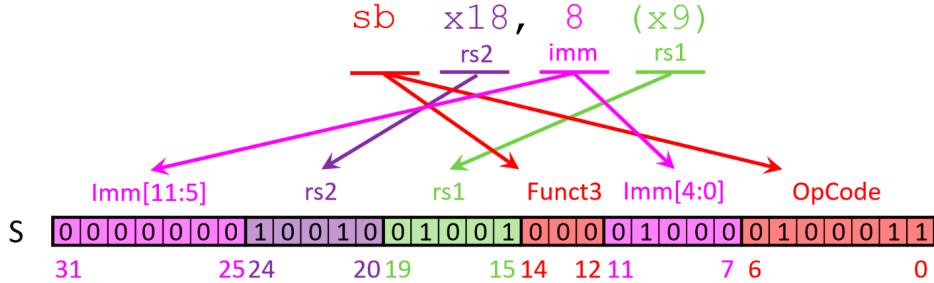


Figura 2.4: Ejemplo código de instrucción tipo S.

2.3. Instrucciones de saltos condicionados

Este tipo de formato permite realizar comparaciones lógicas condicionales entre dos registros. Cuando se cumple la condición, el valor del Program Counter (PC) toma la dirección del desplazamiento, lo cual es un salto simple a otra dirección de la memoria de instrucciones; de no cumplirse, la ejecución continua su flujo secuencial. En ejercicios prácticos de alto nivel, se describe la dirección de una instrucción con una etiqueta alfanumérica, también llamada label.

Las operaciones tipo B que pueden usarse son:

- **beq**: Salto si es igual (Branch if equal)
 $\text{if}(\text{RU}[rs1] == \text{RU}[rs2]) \text{ then } \text{PC} = \text{PC} + \text{imm}$
- **bne**: Salto si no es igual (Branch if not equal)
 $\text{if}(\text{RU}[rs1] != \text{RU}[rs2]) \text{ then } \text{PC} = \text{PC} + \text{imm}$
- **blt**: Salto si es menor qué (Branch if less than)
 $\text{if}(\text{RU}[rs1] < \text{RU}[rs2]) \text{ then } \text{PC} = \text{PC} + \text{imm}$

- **bge**: Salto si es mayor o igual (Branch if greater than or equal)
 $\text{if}(\text{RU}[rs1] \geq \text{RU}[rs2]) \text{ then } \text{PC} = \text{PC} + \text{imm}$
- **bltu**: Salto si es menor que sin signo (Branch if less than unsigned)
 $\text{if}(\text{RU}[rs1] < \text{RU}[rs2]) \text{ then } \text{PC} = \text{PC} + \text{imm}$
- **bgeu**: Salto si es mayor o igual sin signo (Branch if greater than or equal)
 $\text{if}(\text{RU}[rs1] \geq \text{RU}[rs2]) \text{ then } \text{PC} = \text{PC} + \text{imm}$

Tipo B		
	OpCode	Funct3
beq	1100011	000
bne	1100011	001
blt	1100011	100
bge	1100011	101
bltu	1100011	110
bgeu	1100011	111

Tabla 2.6: Códigos de una instrucción tipo B.

En la tabla (ver Tabla: 2.6), se muestran los diferentes valores de los registros para cada instrucción de salto condicional.

Ejemplo instrucciones tipo B

Se comparará si la variable $i == 0$, si se cumple se realizará la suma de g, h de no cumplirse se realizará una resta entre g, h , la asignación de registros queda de la siguiente manera:

```
x8 ← f
x9 ← i
x18 ← g
x19 ← h
```

A continuación se presenta un código en C:

```
if (i == 0) {  
    f = g + h;  
}  
else {  
    f = g - h;  
}
```

En este ejercicio se usará la estructura de las instrucciones condicionales, lo cual se logra con la instrucción de salto incondicional, donde se compara si los registros fuentes rs1 y rs2 son iguales, esto genera que la dirección del program counter (PC) cambie a la dirección de la etiqueta de la instrucción, generando así la ejecución de otra instrucción, además, por convención, en lenguaje de bajo nivel, se realiza la comparación inversa, esto quiere decir que si en alto nivel se tiene la comparación == en bajo nivel es !=.

A continuación se presenta el código en lenguaje ensamblador RISC-V:

```
if:  
    bne x9, x0, endif  
    sub x8, x18, x19  
endif:  
    add x8,x18, x19
```

Ahora se realizará un bucle while, en el cual se iterara un índice i, se asigna los registros de la siguiente manera:

```
x8 ← i  
x9 ← tope
```

A continuación se presenta un código en C:

```
int i = 0;  
int tope = 10;  
  
while (i < tope) {  
    i = i + 1;  
}
```

Para realizar un bucle, se debe realizar un salto incondicional que esté comparándose mientras se cumple la condición, estos saltos se generan realizando la comparación donde los registros fuente uno (rs1) y dos (rs2) con el registro x0.

A continuación se presenta el código en lenguaje ensamblador RISC-V:

```
add x8, x0, x0  
addi x9, x0, 10  
while:  
    bge x8, x9, endwhile  
    addi x8, x8, 1  
    beq x0,x0, while  
endwhile:
```

Ahora se realizará un ciclo **for**, mientras i sea menor que 10, la asignación de variables queda de la siguiente manera:

```
x8 ← i  
x9 ← f  
x18 ← g  
x19 ← tope
```

A continuación se presenta un código en C:

```
for (i = 0; i < n; i++) {  
    f = g + 1;  
}
```

Se inicializa el registro x8 en 0 y se crea la etiqueta for, se realiza la comparación, se realizan las operaciones internas, se actualiza el contenido del registro x8 y finalmente se genera el bucle con un salto incondicional.

A continuación se presenta el código en lenguaje ensamblador RISC-V:

```
add x8, x0, x0  
addi x19, x0, 10  
for:  
  
    bge x8, x19, endfor  
    addi x9, x18, 1  
    addi x8, x8, 1  
    beq x0, x0, for  
endfor:
```

Para probar el funcionamiento en conjunto de las instrucciones de carga y las instrucciones, se asume que se tiene un vector V, con 10 elementos. A continuación se desarrolla un algoritmo en c que permita conocer la suma de sus elementos.

```
i=0;  
acc = 0;  
size=10;  
for (i; i < size; i++) {  
    acc += V[i];  
}
```

En este ejercicio se itera sobre el vector acumulando los valores de cada uno. En lenguaje ensamblador, para inicializar una variable de iteración dentro de un vector se debe realizar el cálculo para conocer la dirección efectiva que debe cargarse, para esto se cuenta con la ecuación (ver Ecación: (2.3))

$$4.i + \&V, \quad (2.1)$$

Donde **i**, es la variable de iteración que es multiplicada por 4 bytes y **&V** representa la dirección base del vector **V**, que estará almacenada en **x19** se supondrá que la dirección inicial de **V** es en la posición 10.

La variable **&V**, representa dirección base del vector de elementos **V**. La variable **i** es una variable de iteración que será usada para calcular el índice de cada elemento, esta debe ser multiplicada por un factor de 4, esto debido a que como se verá más adelante, la memoria que contiene esta información está diseñada en conjuntos de bytes, donde la suma de 4 bytes representa un elemento de 32 bits. Al finalizar un ciclo **for** es imprescindible introducir el salto incondicional, que permite efectuar el bucle, obligando al sistema a ejecutar las veces que sean necesarias las instrucciones del cuerpo del ciclo **for**. Finalmente, se necesitará una variable **size** que almacene la cantidad de elementos que tiene el vector **V**.

Los registros que se usarán son:

```
x8 ← i
x9 ← acc
x18 ← size
x18 ← &V
```

A continuación se presenta el código en lenguaje ensamblador RISC-V:

```

addi x8, x0, 0
addi x9, x0, 0
addi x18, x0, 10
for:
    bge x8, x18, endfor
    slli x8, x8, 2
    add x7,x8, x18
    lw x5, 0(x7)
    add x9, x9, x5
    addi x8,x8, 1
    beq x0,x0, for
endfor:

```

Lenguaje de bajo nivel

En la figura (ver Figura: 2.5), se observa el formato de una instrucción tipo B en lenguaje ensamblador y su traducción en formato binario.

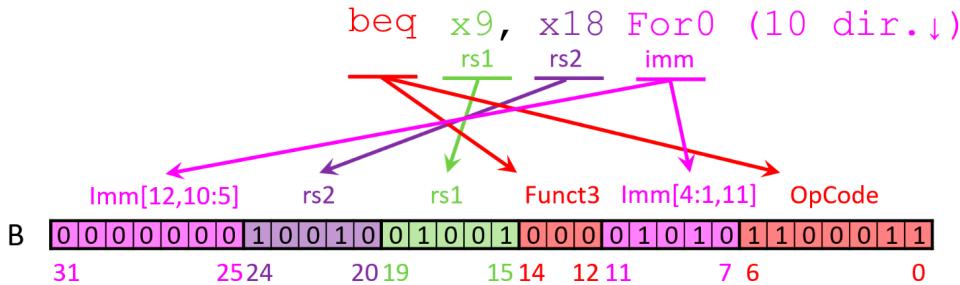


Figura 2.5: Ejemplo código de instrucción tipo B.

2.4. Instrucciones de saltos incondicionados

2.4.1. jal tipo J

Este tipo de instrucciones se utilizan para realizar saltos desde la siguiente instrucción del contador de programa (PC) hasta el inmediato especificado. La dirección de la siguiente instrucción es almacenada en el registro especificado por la entrada rd, donde generalmente se utiliza el registro ra (return address).

- **jal**: Salto y guardado (Jump and link)
 $PC = PC + imm; RU[rd] = PC + 4.$

Tipo J	
OpCode	
jal	1101111

Tabla 2.7: Código de una instrucción tipo J.

En la tabla (ver Tabla: 2.7), se muestra el valor del OpCode para la instrucción jal.

Lenguaje de bajo nivel

En la figura (ver Figura: 2.6), se observa una instrucción en formato tipo J, lenguaje ensamblador y su traducción en formato binario.

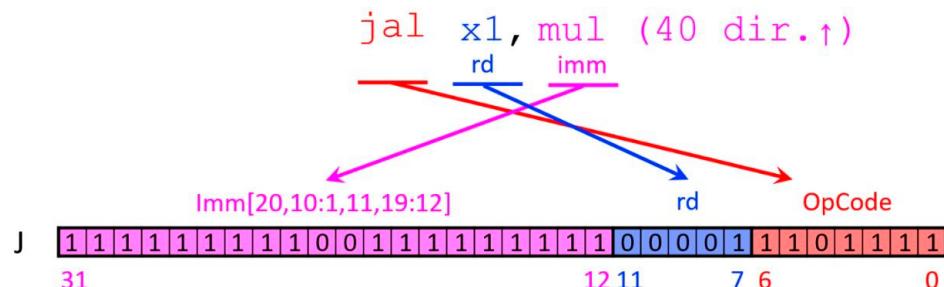


Figura 2.6: Ejemplo código de instrucción tipo J.

2.4.2. jalr tipo I

Entre las instrucciones tipo I se encuentra el *jalr*, la cual sirve para ejecutar un salto a una instrucción que tiene su dirección almacenada dentro de un registro. A esta dirección se le suma el desplazamiento especificado en el valor del inmediato. A su vez se guarda en el registro **rd** la dirección de la siguiente instrucción antes del salto. Suele ser almacenado en el registro de uso específico **ra** (x1).

- **jalr**: Salto a un registro y guardado (Jump And Link Register)
 $PC = RU[rs1] + imm; RU[rd] = PC + 4$

Tipo I salto		
OpCode	Funct3	
jalr	1100111	000

Tabla 2.8: Código de una instrucción tipo I para salto incondicional.

En la tabla (ver Tabla: 2.8), se muestran los valores de los registros para la instrucción *jalr*.

Para llevar esta instrucción a bajo nivel se hace directamente como es definido en la sección de instrucciones tipo I aritmético lógicas 2.1.2.

2.4.3. Funciones

Las instrucciones *jal* y *jalr* trabajan en conjunto para el llamado y retorno a funciones. Este proceso se ve evidenciado con el siguiente ejemplo:

Se asume que se tiene una función en C llamada multiplicación por medio de sumas y adicionalmente una función *main*, desde la cual se realiza el llamado a la función multiplicación.

```

int Multiplicacion(int a, int b) {
    int i = 0;
    int acc = 0;

    for(i; i < a; i++) {
        acc = acc + b;
    }
    return acc;
}

```

También se asume que se tiene una función que calcula la potencia por medio de multiplicaciones sucesivas.

```

int Potencia(int x, int y) {
    int acc = 1;
    int i = 0;
    for(i; i < y; i++) {
        acc = Multiplicacion(acc,x);
    }
    return acc;
}

```

Para este ejercicio es importante introducir la tabla de direcciones, a fin de que pueda verse más clara y detalladamente en qué consisten los llamados a función. Para el paso de separar espacio en la pila para los elementos de la función multiplicación, se usará la cantidad de registros, más el apuntador a la dirección de retorno (ra), en este caso se debe separar 5 registros, como las direcciones de la memoria están en bytes para poder guardar los registros es necesario multiplicar el valor antes mencionado por 4, que es la cantidad de bytes de un registro.

El almacenamiento en la **pila** es una parte del proceso de llamado a funciones debido a que permite guardar en memoria temporalmente los valores dentro de los registros que se va a utilizar en la función, dado que

no hay certeza si los valores que contienen en ese instante de tiempo hacen parte de otro proceso. El registro **sp** (**x2**) contiene la dirección inicial de la pila en la memoria. Con base en esta dirección se reserva un espacio en la memoria, restando el valor del **sp** (**x2**) con la cantidad de bytes que se quieren guardar temporalmente, seguido de una instrucción de escritura de memoria en esa posición.

De una manera global, los pasos que se deben seguir para realizar una función son:

- Definir los registros asociados a las variables
- Reservar espacio en la pila
- Almacenar los registros en la pila
- Realizar instrucciones intermedias
- Cargar los nuevos valores de la pila
- Retornar los resultados por el registro de retorno **x10**
- Recuperar los registros de la pila
- Liberar espacio en la pila
- Retornar a la dirección de llamado **ra** ó **x1**

Los registros que se van a usar para recibir los argumentos son **x10** y **x11** y los registros **x8**, **x9**, **x18**, **x20**, para las variables **a**, **b**, **i**, **acc**. En la asignación de registros se tiene en cuenta la dirección de las flechas, por ejemplo, para la primera asignación, la variable entra a la función desde un registro de tipo argumento, por lo cual la dirección de la flecha es hacia la izquierda, indicando el registro que almacena esta información.

Asignación de variables a registros para la función multiplicación

```
x8 ← x10 ← a  
x9 ← x11 ← b  
x18 ← i  
x20 ← acc
```

Asignación de variables a registros para la función potencia:

```
x8 ← x10 ← x  
x9 ← x11 ← y  
x18 ← i  
x19 ← acc
```

Para hacer el llamado a funciones es necesario almacenar la dirección de retorno que es la siguiente instrucción a ejecutarse (PC + 4), esta dirección se almacena en el registro **ra** o Return Address. Para que el salto sea efectivo se usa la instrucción jump and link (**jal**), con los argumentos de dirección de retorno y la etiqueta.

Para mostrar el llamado a funciones, se desarrolla una función en C, que llama a la función a la función potencia y retorna el resultado por **x10**.

```
int main () {  
    int a = 10;  
    int b = 5;  
    int respuesta = Potencia(a,b);
```

Dirección	Etiqueta	Instrucción
0x00002004	multi:	addi sp, sp, -20
0x00002008		sw x1, 16(sp)
0x0000200C		sw x8, 12(sp)
0x00002010		sw x9, 8(sp)
0x00002014		sw x18, 4(sp)
0x00002018		sw x20, 0(sp)
0x0000201C		add x8, x10, x0
0x00002020		add x9, x11, x0
0x00002024	for:	bge x18, x8, endfor
0x00002028		add x20, x20, x9
0x0000202C		addi x18, x18, 1
0x00002030		beq x0, x0, for
0x00002034	endfor:	addi x10, x20, 0
0x00002038		lw x1, 16(sp)
0x0000203C		lw x8, 12(sp)
0x00002040		lw x9, 8(sp)
0x00002044		lw x18, 4(sp)
0x00002048		lw x20, 0(sp)
0x0000204C		addi sp, sp, 20
0x00002050		jalr x0, x1, 0

Dirección	Etiqueta	Instrucción
0x00004008	pot:	addi sp, sp, -20
0x0000400C		sw x8, 16(sp)
0x00004010		sw x9, 12(sp)
0x00004014		sw x18, 8(sp)
0x00004018		sw x19, 4(sp)
0x0000401C		sw x1, 0(sp)
0x00004020		add x8, x10, x0
0x00004024		add x9, x11, x0
0x00004028		addi x18, x0, 0
0x0000402C		addi x19, x0, 1
0x00004030	for:	bge x18, x9, endfor
0x00004034		addi x10, x19, 0
0x00004038		addi x11, x8, 0
0x0000403C		jal x1, multi
0x00004040		add x19 x0, x10
0x00004044		addi x18 x18, 1
0x00004048		beq x0, x0, for
0x0000404C	endfor:	addi x10, x19, 0
0x00004050		lw x8, 16(sp)
0x00004054		lw x9, 12(sp)
0x00004058		lw x18, 8(sp)
0x0000405C		lw x19, 4(sp)
0x00004060		lw x1, 0(sp)
0x00004064		addi sp, sp, 20
0x00004068		jalr x0, x1, 0

```
    return respuesta;  
}
```

Los pasos para realizar un llamado a función son los siguientes:

- Definir los registros asociados a las variables
- Almacenar los registros en argumentos de función
- Realizar el llamado con `jal` almacenando la dirección de retorno
- Tomar el resultado del retorno que está en el registro de retorno `x10`

Los registros asociados a la llamada de función son `x8`, `x9`, `x18` para `a`, `b`, `respuesta` y después deben pasar a registros de argumentos de función: `x10` y `x11` respectivamente, para después hacer el llamado a la función Multiplicación.

```
x10 ← x8 ← a  
x11 ← x9 ← b  
x10 ← x18 ← respuesta
```

En el código ensamblador se ve reflejado el uso del formato `J`, para realizar un llamado a la función Potencia.

Dirección	Etiqueta	Instrucción
0x00003000	Main:	addi x8, x0, 10
0x00003004		addi x9, x0, 5
0x00003008		add x18, x0, x0
0x0000300C		addi x10, x8, 0
0x00003010		addi x11, x9, 0
0x00003014		jal ra, pot
0x00003018		addi x18, x10, 0
0x0000301C		addi x10, x18, 0

2.5. Instrucciones auxiliares

Este tipo de instrucciones es útil en caso de necesitar cargar inmediatos de 20 bits en un registro destino o en el PC. En este formato debe especificar la dirección del registro (rd) en el cual se desea almacenar el valor y el inmediato, que será cargado en los 20 bits más significativos del registro destino y borra los 12 menos significativos.

- **lui:** Cargar un inmediato superior (Load Upper Immediate)

$$RU[rd] = 16'b'imm[20], \text{ imm}, 12'b0$$

- **auipc:** Cargar un inmediato superior a PC (Add Upper Immediate to PC)

$$RU[rd] = PC + imm, 12'b0$$

Tipo U	
	OpCode
lui	0110111
auipc	0010111

Tabla 2.9: Código de una instrucción tipo U.

En la tabla 2.8 se muestran los valores del OpCode para las instrucciones tipo U.

Ejemplo instrucciones formato U, para almacenar un gran inmediato

Para almacenar un valor mayor que 2^{12} en una variable, el campo del inmediato no es suficiente para codificarlos. Llevar este valor a un registro requiere de 2 instrucciones: lui para escribir los 20 bits más significativos y addi para escribir los 12 bits menos significativos.

x = 1000000;

A continuación se presenta la asignación de variable a registro:

`x8 ← x`

El código correspondiente en el lenguaje de bajo nivel sería:

```
lui x8, 244  
addi x8, x8, 576
```

Existe un caso especial en el cual la instrucción `addi` le suma un 1 bit a los 20 bits más significativos del registro destino. Esto se debe a la extensión de signo cuando el bit más significativo de los 12 bits del inmediato es 1. Para corregir esto se le debe restar 1 al valor que se carga con la instrucción `lui`.

Lenguaje de bajo nivel

En la figura (ver Figura: 2.7), se observa el formato de las instrucciones tipo U en lenguaje ensamblador y su traducción en formato binario.

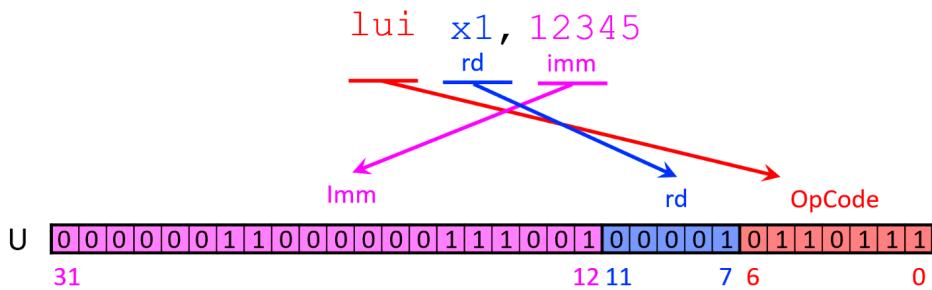


Figura 2.7: Ejemplo código de instrucción tipo U.

2.6. Pseudo instrucciones

Este conjunto de instrucciones hace referencia a una representación para hacer abreviaciones a casos específicos que se pueden presentar en

una ejecución. Estos casos son relacionados en la siguiente tabla.

- **beqz:** Salto si es igual a 0 (Branch = 0)
 $\text{if}(\text{RU}[\text{rs1}] == 0) \text{ PC} = \text{PC} + \text{imm}, 1'b0$
- **bnez:** Salto si es diferente a 0 (Branch != 0)
 $\text{if}(\text{RU}[\text{rs1}] \neq 0) \text{ PC} = \text{PC} + \text{imm}, 1'b0$
- **j:** Salto (Jump)
 $\text{PC} = \text{imm}, 1'b0$
- **jr:** Salto a registro (Jump register)
 $\text{PC} = \text{RU}[\text{rs1}]$
- **la:** Carga de dirección (Load address)
 $\text{RU}[\text{rd}] = \text{address}$
- **li:** Carga inmediato (Load immediate)
 $\text{RU}[\text{rd}] = \text{imm}$
- **mv:** Mover (Move)
 $\text{RU}[\text{rd}] = \text{RU}[\text{rs1}]$
- **neg:** Negación (Negate)
 $\text{RU}[\text{rd}] = -\text{RU}[\text{rs1}]$
- **nop:** Sin operación (No operation)
 $\text{RU}[0] = \text{RU}[0]$
- **not:** No (Not)
 $\text{RU}[\text{rd}] = \sim \text{RU}[\text{rs1}]$
- **ret:** Retornar (Return)
 $\text{PC} = \text{RU}[1]$
- **seqz:** Activar si es igual a 0 (Set = 0)
 $\text{RU}[\text{rd}] = (\text{RU}[\text{rs1}] == 0) ? 1 : 0$

- **snez:** Activar si es igual a 0 (Set != 0)

$\text{RU[rd]} = (\text{RU[rs1]} \neq 0) ? 1 : 0$

2.7. Ejercicios Capítulo 2

1. Desarrollar una función en alto nivel que ordene un vector de enteros.
2. Desarrollar una función en alto nivel que calcule el promedio de los elementos de un vector.
3. Desarrollar una función en alto nivel que realice polinomios por medio de sumas de potencias sucesivas.
4. Desarrollar una función en alto nivel que realice la suma de los elementos de un vector.
5. Desarrollar una función en alto nivel que calcule el número Factorial.
6. Desarrollar una función en alto nivel que calcule el número Fibonacci.
7. Desarrollar las anteriores funciones en lenguaje de máquina y lenguaje ensamblador.
8. Si se tienen las siguientes operaciones entre variables:
 $a = b - c$
 $d = (a + c) \text{ XOR } (b - c)$
 $e = (d \& a) - ((a + b) < b)$

Escriba su representación secuencial en lenguaje ensamblador y lenguaje de máquina.

9. Si se tienen las siguientes instrucciones en formato ensamblador RISC-V:

```
addi x8, x0, 45
addi x9, x8, 16
sw x8, 0(x7)
sw x9 1(x7),
lw x8, 1(x7)
```

Traducirlas al lenguaje de alto nivel C y señalar las variables que son enteros y las que son direcciones de memoria.

10. Del ejercicio anterior escriba alguna de las instrucciones de RISC-V en lenguaje de máquina.
11. Si se tiene el siguiente código en lenguaje C:

```
int a = 6;
int acc = 0;
int V[] = {1,2,3,4,5,6};
for(int i = 0; i < a; i++){
    int num = V[i];
    V[i] = num * 2;
}
```

Escriba su representación secuencial en lenguaje ensamblador y lenguaje binario.

12. Del ejercicio anterior escriba alguna de las instrucciones de RISC-V en formato binario.
13. De los anteriores ejercicios, escoja uno en el cual se puedan implementar pseudo instrucciones y hágalo, de manera que el código RISC-V quede más compacto y entendible.

3

CAPÍTULO TRES

Procesador

El procesador es el nombre o identificador que se le da a la unión de un conjunto de módulos que permiten realizar la ejecución de instrucciones en lenguaje de máquina. Dentro de estos procesadores se encuentran 4 arquitecturas, la primera y menos compleja es la arquitectura del procesador monociclo, que permite ejecutar una instrucción por ciclo de procesador. Además, se estudiarán un conjunto de procesadores de arquitectura segmentada, la cual permite reducir tiempos de ejecución y mejora el rendimiento, algunos de los cuales destacan el procesador segmentado con solución por software, el procesador segmentado con solución por Hardware y finalmente el procesador segmentado con detección de riesgos.

3.1. Procesador Monociclo

El procesador monociclo (ver Figura: 3.1), realiza ejecución de instrucciones en forma secuencial, una por ciclo de procesador. Este procesador cuenta con diferentes módulos que permiten ejecutar diferentes tareas, como por ejemplo la lectura de registros o la suma entre dos operandos, los diferentes módulos y sus caminos de datos, serán analizados a profundidad más adelante en esta sección.

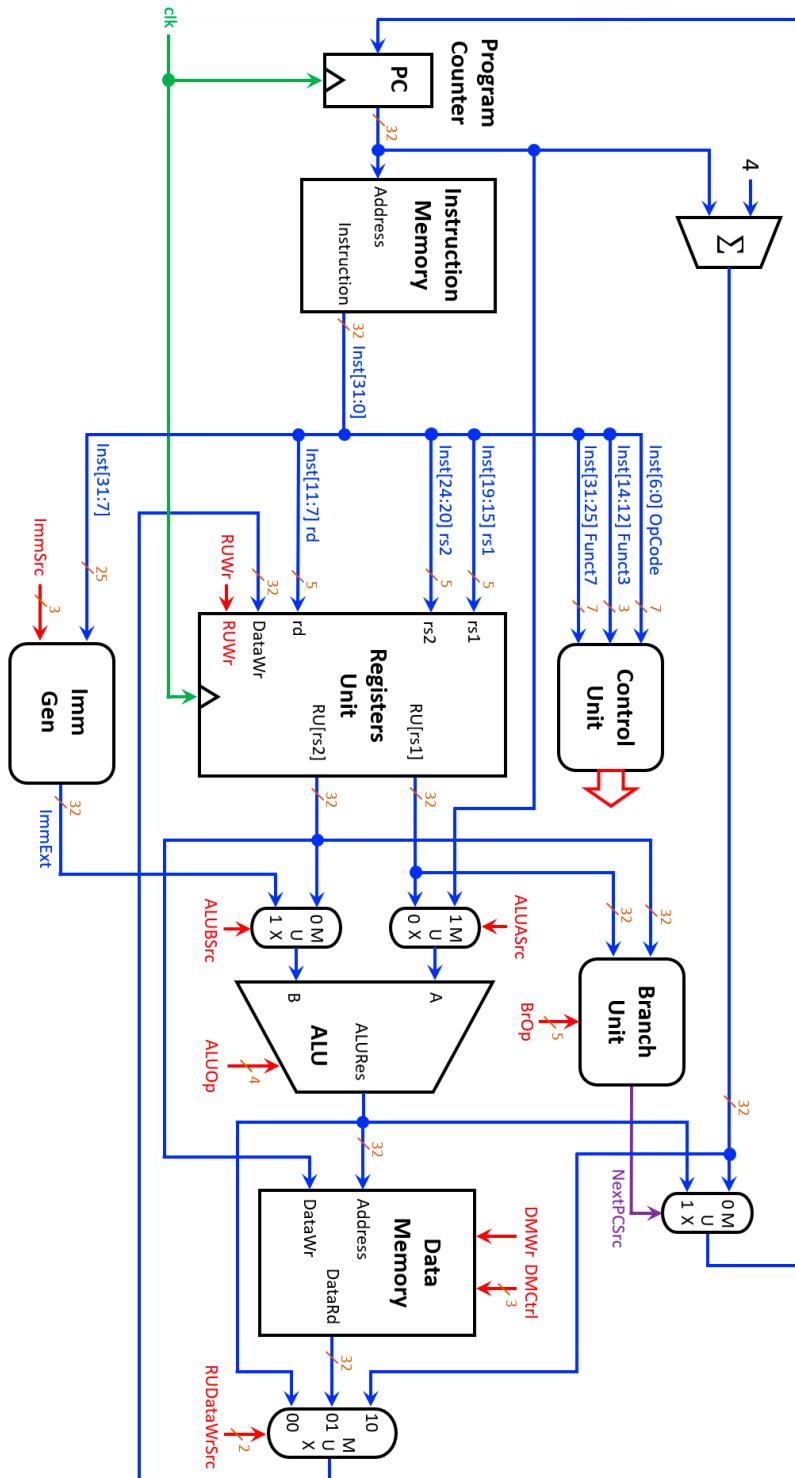


Figura 3.1: Procesador monociclo.

Contador de programa

Este módulo (ver Figura: 3.2), se encarga de recibir la dirección de la instrucción a ejecutar en el ciclo del reloj y la mantiene por ese periodo de tiempo.

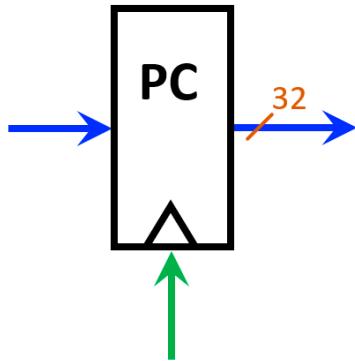


Figura 3.2: Contador de programa.

3.1.1. Memoria de instrucciones

La memoria de instrucciones (ver Figura: 3.3) en un procesador tiene la función de retornar la instrucción que se le solicita para ser ejecutada. Esta memoria recibe como entrada una dirección específica que apunta a la memoria de instrucciones, de la cual se recupera el dato que finalmente es retornado por la salida *instruction*.

3.1.2. Unidad de control

Esta unidad que se observa (ver Figura: 3.4), puede ser comparada con el cerebro humano, debido a que es la encargada de dar las órdenes a los módulos subyacentes. La forma en la que se comunica con los otros módulos y multiplexores es mediante las señales de control. Cada señal se encarga de transmitir un valor binario, el cual representa una determinada acción tal como activar o desactivar el modo de lectura de una memoria de datos. La forma en que asigna cada función se da gracias a los valores

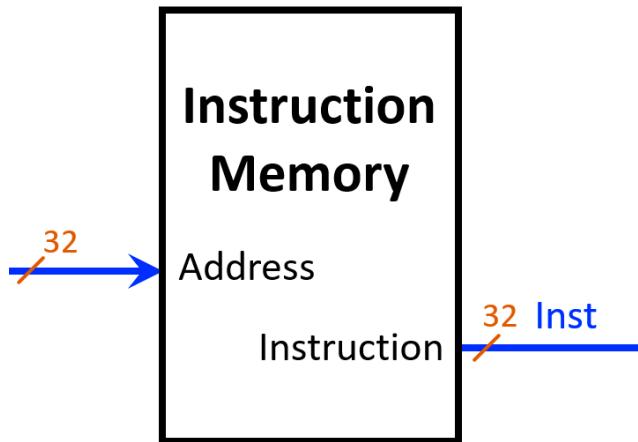


Figura 3.3: Memoria de instrucciones.

que llegan de cada instrucción. Los campos que se tienen en cuenta de cada instrucción para el proceso de asignación de señales son: el OpCode: $\text{Inst}([6:0])$, el cual contiene el formato de instrucción a ejecutar, el Funct3: $\text{Inst}([14:12])$ y Funct7: $\text{Inst}([31:25])$ que generan un Funct10 el cual se usa para representar las respectivas operaciones a ejecutar. En los diagramas de los módulos del procesador las señales de control estarán marcadas con su respectivo nombre.

A continuación se presentan tres tablas que condensan todas las señales de control generadas para los módulos que componen el diseño del procesador monociclo:

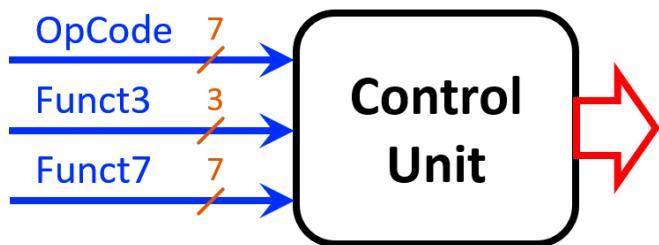


Figura 3.4: Unidad de control.

Señales	R	I	I-Carga
RUWr	1	1	1
ALUOp	{F7[5], F3}	{F7[5], F3}	0000
ImmSrc	XXX	000	000
ALUASrc	0	0	0
ALUBSrc	0	1	1
DMWr	0	0	0
DmCtrl	XXX	XXX	F3
BrOp	00XXX	00XXX	00XXX
RUDataWrSrc	00	00	01

Tabla 3.1: Señales de control de los formatos: R, I, I-Carga.

Señales	B	J	S
RUWr	0	1	0
ALUOp	0000	0000	0000
ImmSrc	101	110	001
ALUASrc	1	1	0
ALUBSrc	1	1	1
DMWr	0	0	1
DmCtrl	XXX	XXX	{2b01, F3}
BrOp	F3	1XXXX	00XXX
RUDataWrSrc	XX	10	XX

Tabla 3.2: Señales de control de los formatos: B, J, S.

Señales	I-Salto	U
RUWr	1	0
ALUOp	0000	0111
ImmSrc	000	010
ALUASrc	0	X
ALUBSrc	1	1
DMWr	0	0
DMCtrl	XXX	XXX
BrOp	1XXXX	00XXX
RUDataWrSrc	10	00

Tabla 3.3: Señales de control de los formatos: I-Salto, U.

3.1.3. Unidad de registros

La Unidad de Registros (ver Figura: 3.5), se compone de 32 registros, en los cuales cada uno almacena 32 bits. esta unidad funciona de dos maneras, en modo de lectura y modo de escritura.

Cada registro posee un uso diferente, existen diferentes tipos, entre ellos: registros temporales y registros almacenados en memoria, registros para interactuar con la pila, entre otros, que se especifican en la tabla (ver Tabla: 2.1).

En el modo de lectura su función es recuperar o leer de la matriz donde están almacenados los registros, utilizando como índice para la lectura las dos entradas **rs1** y **rs2** respectivamente, ambas constan de 5 bits, por lo tanto, su representación o lectura va desde registro x0 al registro x31. Después de recuperar estos dos valores, son almacenados en las salidas **RU[rs1]** y **RU[rs2]** respectivamente, donde cada una consta de 32 bits, siendo el valor recuperado de la unidad de registros.

Cuando se va a trabajar en el modo de escritura, la entrada **RUWr** de un bit debe estar activa o en '1'. El registro **rd** (tamaño de 5 bits) especifica la dirección en que debe ser almacenado el valor ingresado por la entrada **DataWr** (tamaño de 32 bits). Este proceso de escritura se debe hacer en el

respectivo flanco de subida de la señal Clk.

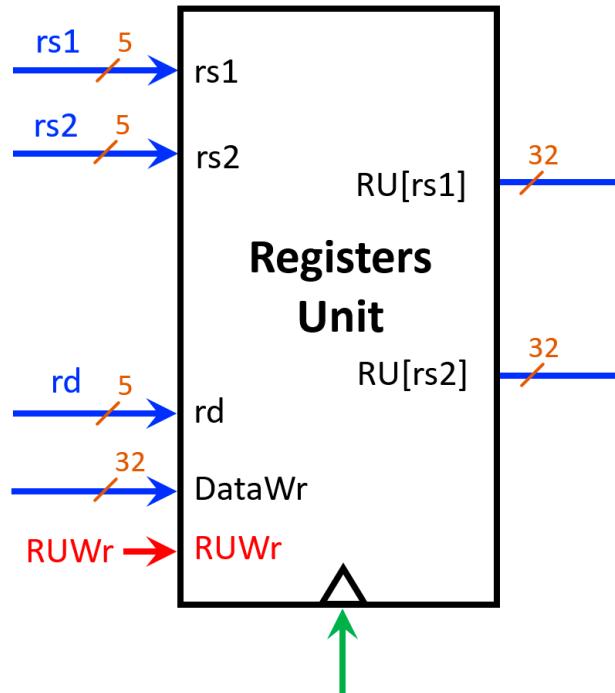


Figura 3.5: Unidad de registros.

3.1.4. Unidad de inmediatos

La unidad de inmediatos (ver Figura: 3.6), es de gran utilidad para extender o llenar los bits que hacen falta para completar un mensaje completo de 32 bits. El cual va a ser usado en la unidad aritmético lógica (ver Figura: 3.1.5), como una de sus entradas.

Este módulo cuenta con una señal de control llamada `ImmSrc` (ver Tabla: 3.4), la cual se encarga de seleccionar el tipo de extensión que se va a aplicar debido al formato de instrucción que se está ejecutando.

El inmmedio a extender o `ImmGen` tiene los valores de los 25 bits más significativos de la instrucción, ya que en ellos se encuentran los diferentes campos donde están ubicados los inmediatos a extender.

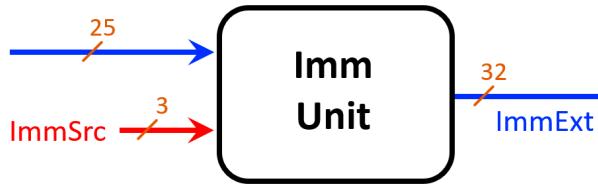


Figura 3.6: Unidad de inmediatos.

ImmSrc
I 000
S 001
B 101
U 010
J 110

Tabla 3.4: Fuente del inmediato

3.1.5. Unidad aritmética lógica

La unidad aritmética lógica (ver Figura: 3.7), es la encargada de realizar operaciones aritméticas tal como sumas, restas, también realiza operaciones lógicas como AND, OR, NOT, XOR, corrimientos a la derecha y a la izquierda.

Su funcionamiento está basado en dos entradas A y B donde cada una tiene 32 bits por las cuales ingresan los valores de los operandos de tipo entero. También tiene una entrada llamada ALUOp de 4 bits, la cual representa las diferentes operaciones que esta unidad puede realizar. Entre ellas se encuentra la suma que se representa con el valor binario 4'b0000 o la resta que es representada con el valor binario 4'b1000. De esta manera se tiene 10 operaciones en total, las cuales se pueden ver en la tabla (ver Tabla: 3.5). Para operaciones lógicas el proceso se hace bit a bit (Bitwise). Cuando se realiza el corrimiento a la izquierda o a la derecha se tiene en cuenta la entrada Shamt (Shift Ammount) que tiene 5 bits para representar

la cantidad de máxima de bits a desplazar. El corrimiento se realiza sobre la entrada A con la cantidad de desplazamiento que la entrada shamt indique. La salida de 32 bits contiene el resultado de la operación.

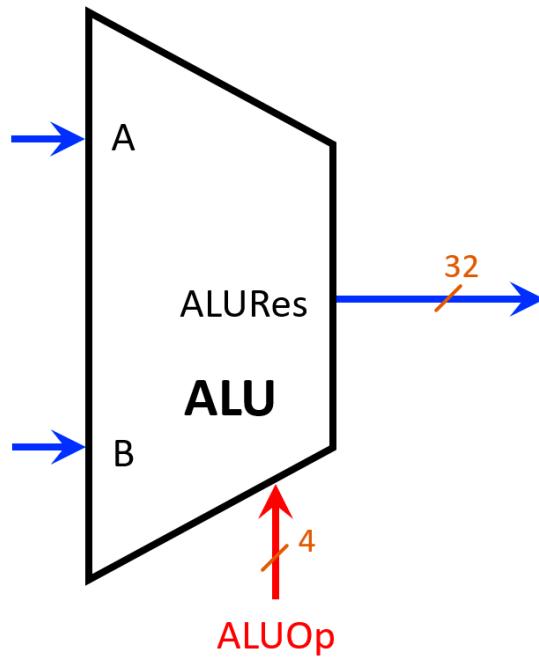


Figura 3.7: Unidad aritmético lógica.

ALUOp			
A + B	0000	A \oplus B	0100
A - B	1000	A $>>$ B	0101
A $<<$ B	0001	A $>>>$ B	1101
A < B	0010	A B	0110
A < B(U)	0011	A & B	0111

Tabla 3.5: Opciones de la ALU.

3.1.6. Unidad de saltos

La unidad de saltos (ver Figura: 3.8), está encargada de realizar comparaciones condicionales entre dos registros de 32 bits. Esta unidad tiene una entrada llamada $\text{BrOp}[4:0]$ (Branch Option de 5 bits) la cual define qué tipo de comparación realizar para evaluar cuál será la siguiente instrucción a ejecutar. El proceso de selección lo hace mediante el tipo de condición de salto dado en la señal BrOp [3:6]. Si el bit más significativo es 1 ($\text{BrOp}[4]$) se ejecuta el salto. Si los dos bits más significativos del Branch Option ($\text{BrOp}[4:3]$) son '01', se evalúa la condición de salto en ($\text{BrOp} [2:0]$) y si los dos bits más significativos del Branch Option ($\text{BrOp} [4:3]$) son '00', no salta y se ejecuta la siguiente instrucción.

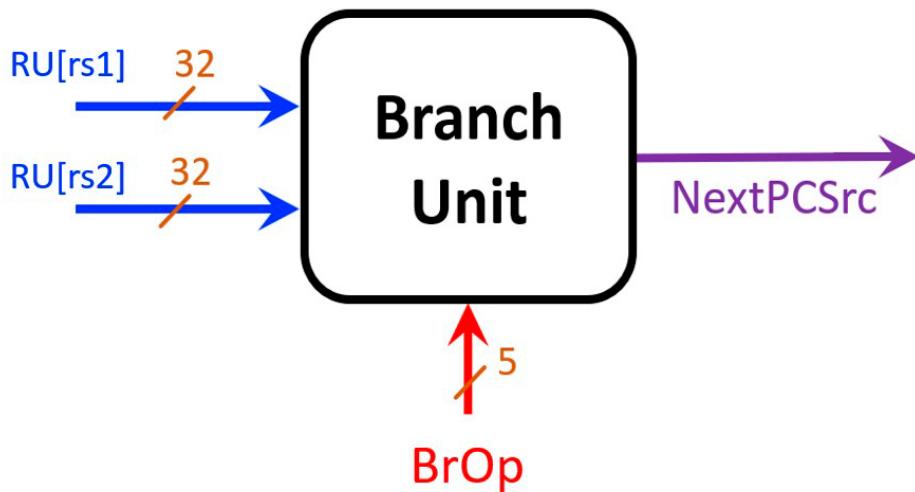


Figura 3.8: Unidad de saltos.

3.1.7. Memoria de datos

La memoria de datos (ver Figura: 3.9), es el lugar del procesador donde son almacenados los valores no volátiles o que se quiere que perduren. Su uso generalmente es para almacenar información.

NextPCSrc	BrOp
0	00XXX
=	01000
≠	01001
<	01100
≥	01101
< (U)	01110
≥ (U)	01111
1	1XXXX

Tabla 3.6: Opciones de salto.

Esta memoria está haciendo una lectura continua. Debido a su señal **DMCtr1** (Data memory control) (ver Tabla: 3.7), la cual con sus 3 bits especifica qué tipo de lectura efectuar, si es un Byte (8 bits), una HalfWord (16 bits) o una Word (32 bits). De esta manera leyendo en la dirección especificada por la entrada **Address** (32 bits). Para la parte de la escritura, está solo se ve aplicada cuando la señal de control **DMWr** está activa, en la cual se desplaza a la memoria en la dirección especificada por la entrada **Address** (32 bits) en el formato especificado en la señal **DMCtr1** guardando el valor dado en la entrada **DataWr** (32 bits) siendo la información a escribir en memoria en esa dirección.

DMCtr1		
B	000	B(U)
H	001	H(U)
W	010	100
		101

Tabla 3.7: Control de la memoria de datos.

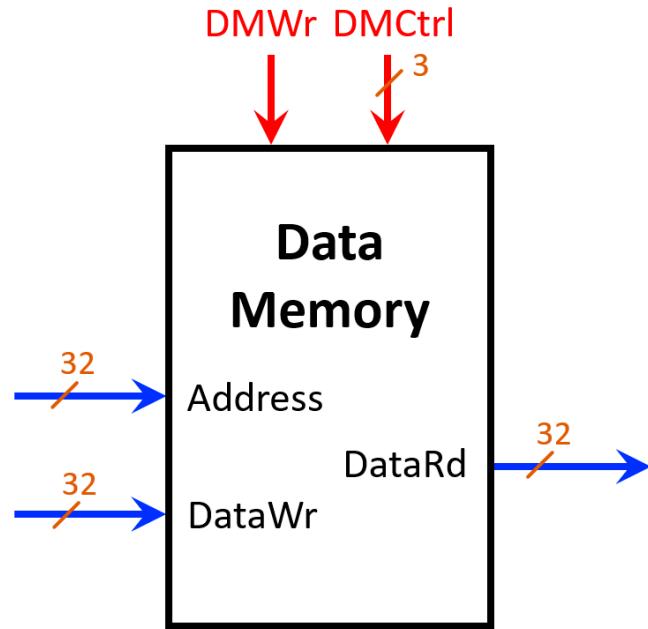


Figura 3.9: Memoria de datos.

3.1.8. Flujo de datos

Para cada tipo de instrucción, hay un flujo de datos específico, donde se tienen en cuenta los valores para las diferentes señales de control. Las señales de control no especificadas en los gráficos se debe a que se asigna un valor dependiendo del comportamiento de la instrucción (ver Figuras: 3.10, 3.11, 3.12, 3.13, 3.14, 3.15, 3.16).

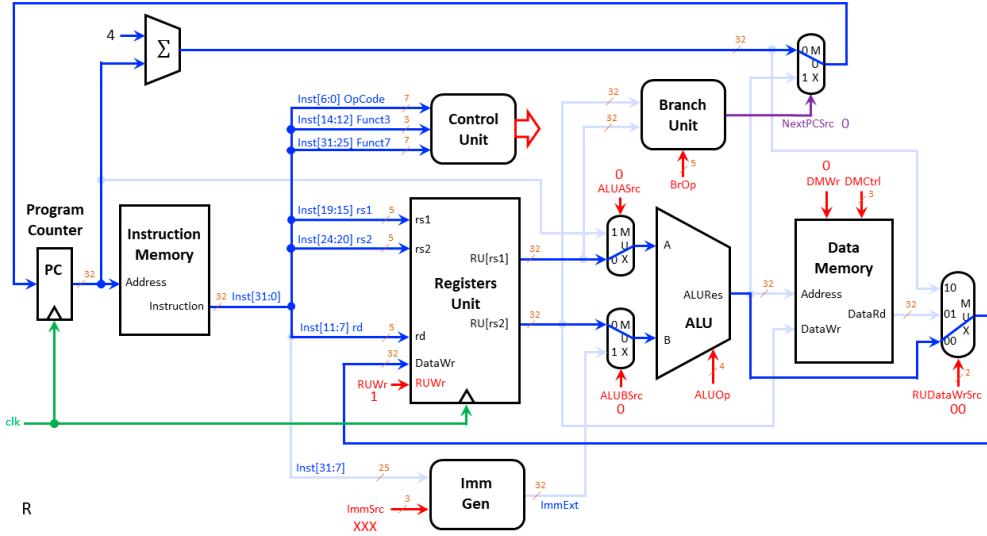


Figura 3.10: Flujo de datos tipo R.

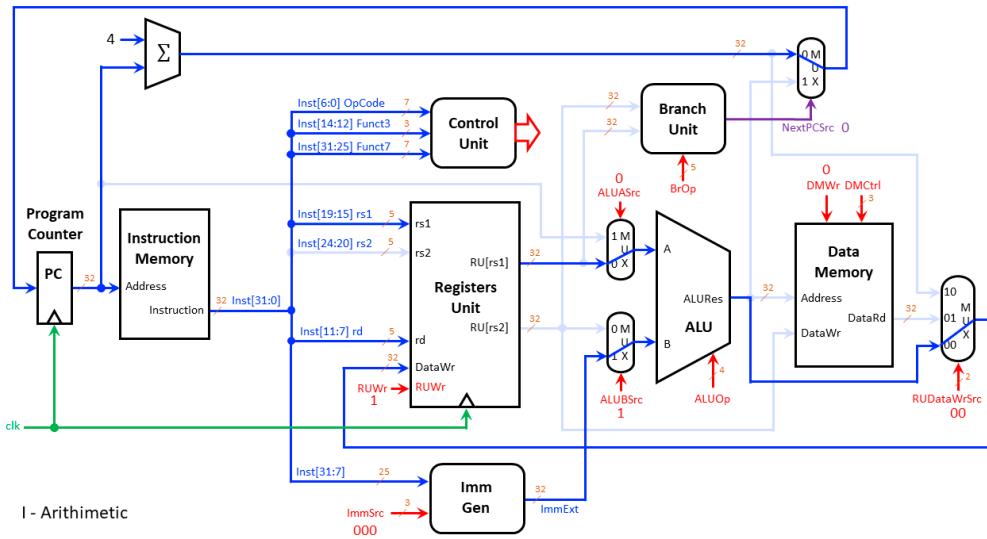


Figura 3.11: Flujo de datos tipo I aritmético lógico.

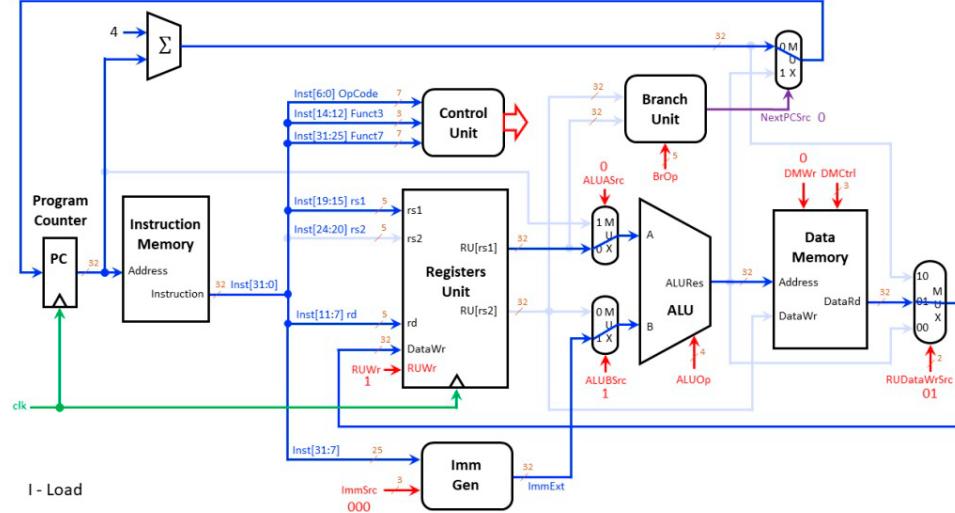


Figura 3.12: Flujo de datos tipo I de carga.

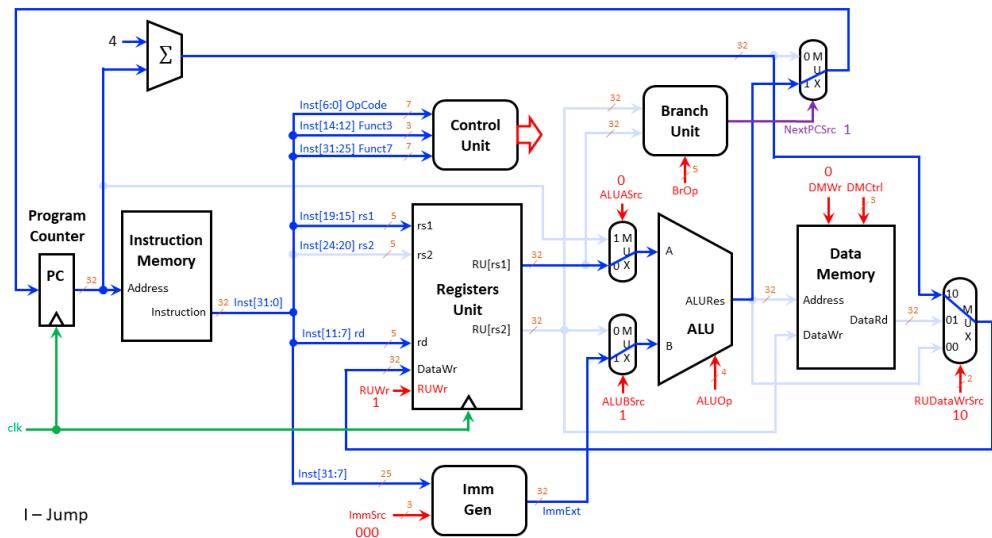


Figura 3.13: Flujo de datos tipo I de salto.

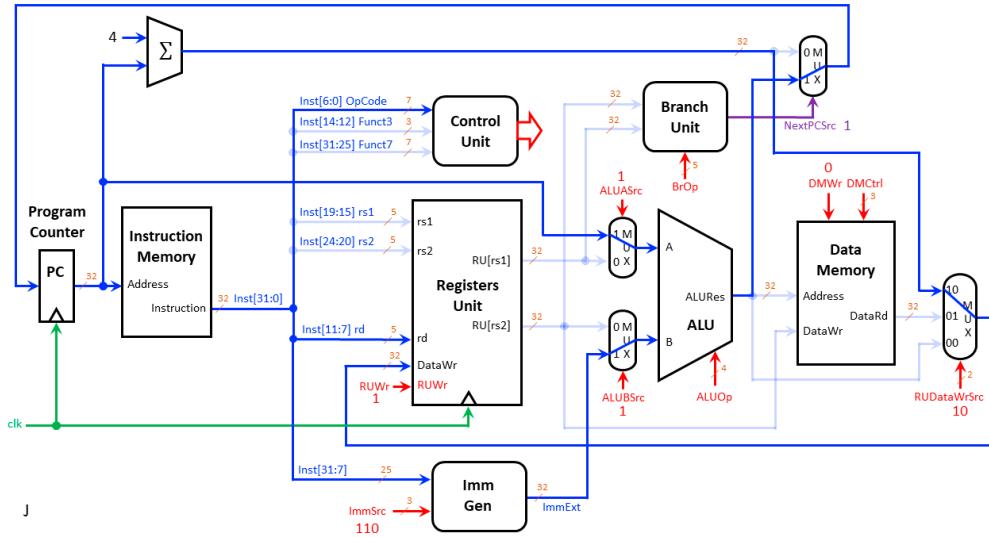


Figura 3.14: Flujo de datos tipo J.

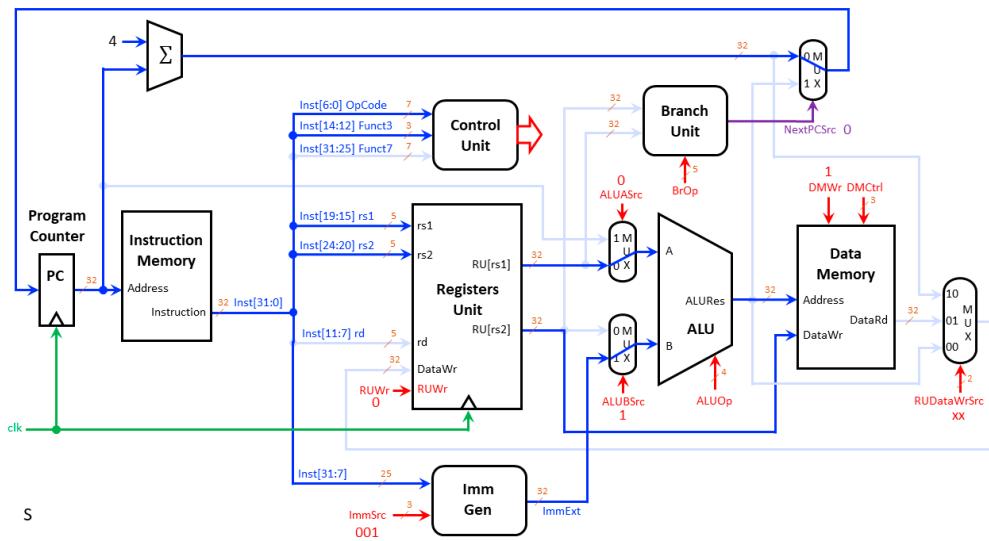


Figura 3.15: Flujo de datos tipo S.

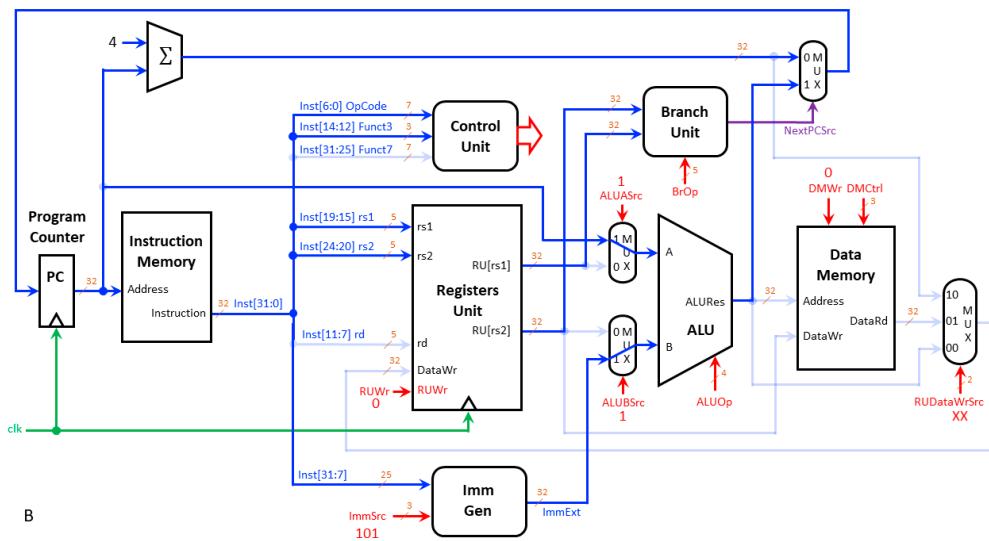


Figura 3.16: Flujo de datos tipo B.

3.2. Procesador segmentado

La segmentación es una técnica de implementación de procesadores que desarrolla el paralelismo a nivel instrucción, en el cual los módulos del procesador se dividen en conjuntos ordenados llamados segmentos, estos permiten realizar operaciones con los datos de cada conjunto de módulos para así poder almacenarlos y tener un flujo hasta de 5 instrucciones por ciclo de procesador como se ve en la figura (ver Figura: 3.17), permitiendo así una ganancia de tiempo de procesamiento. El procesador debido a su arquitectura puede tener hasta cinco segmentos, donde a cada uno le tomara un tiempo de ejecución determinado por el tipo de instrucción, ya que realizar una instrucción de guardado toma 3 ciclos de reloj en cuanto una instrucción de operaciones aritméticas o lógicas, que solo tomarían un tiempo de un ciclo de reloj.

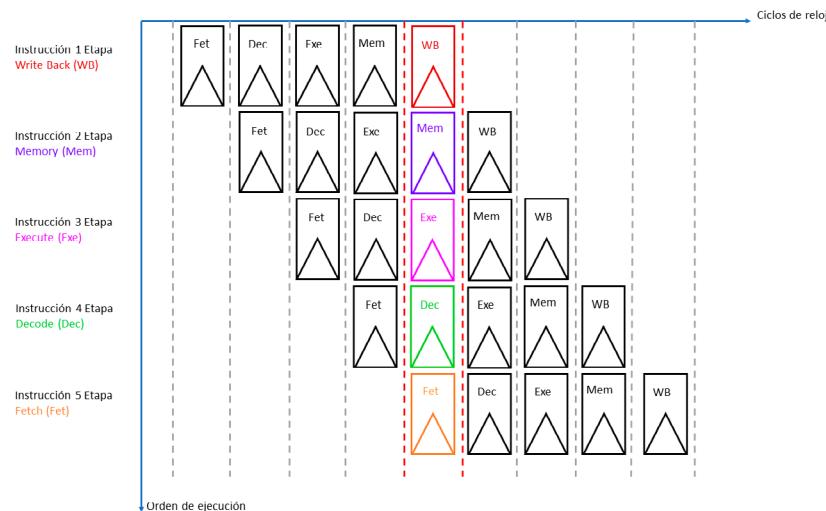


Figura 3.17: Proceso de segmentación.

La principal contribución que brinda la segmentación es permitir la ejecución continua de hasta cinco instrucciones en un mismo momento de tiempo, por ejemplo, un programa compuesto por n instrucciones, puede ser evaluado por la cantidad de tiempo que le toma ejecutar cada instrucción, por lo general estas instrucciones se procesan de forma totalmente

secuencial, el tiempo necesario para procesar el conjunto de instrucciones total será la suma de los tiempos necesarios para la terminación de cada una de ellas, como se muestra en la figura (ver Figura: 3.18). Este esquema representa la instrucción j dentro del programa i . Debe tenerse claro que para comenzar la compilación de un programa será menester esperar un tiempo t que permita a las instrucciones más demoradas lograr su procesamiento final, por ejemplo, si se tratara de una instrucción de carga, el procesador debería parar un tiempo específico mientras la carga se realiza de manera efectiva sobre algún registro, este caso toma 3 ciclos de reloj, por lo cual si alguna de las siguientes instrucciones tiene algún tipo de dependencia, este ya sería resuelto. En conclusión, existirán casos en los cuales los problemas de dependencia de registros deban ser evaluados de manera más rigurosa, evitando la superposición de estas instrucciones dependientes, más adelante se declarará un conjunto de tipos de dependencia y algunas estrategias para evitarlas.

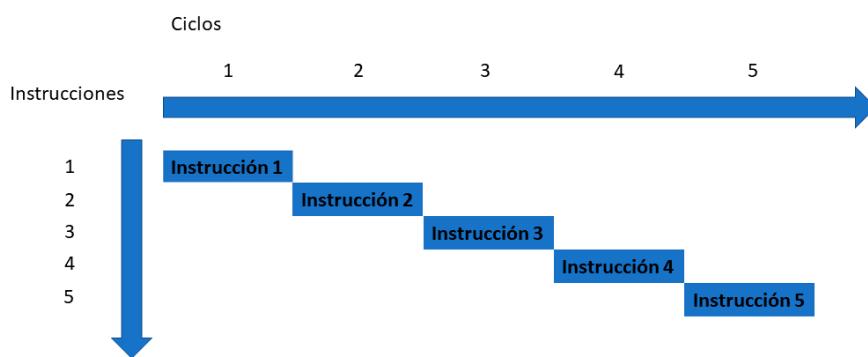


Figura 3.18: Conjunto de instrucciones procesadas de forma totalmente secuencial.

Un procesador segmentado se basa en ejecutar en un mismo instante de tiempo un conjunto de instrucciones, permitiendo así acoplar el modelo de paralelismo vía instrucciones como se muestra en la figura. (ver Figura: 3.19), en esta gráfica se observa que si bien no se profundiza en el tema de las dependencias, cada instrucción toma un tiempo de uno respectivamente, al cabo de cinco tiempos se habrán procesado ya 15 instrucciones, en cuanto al procesador secuencial apenas cinco.

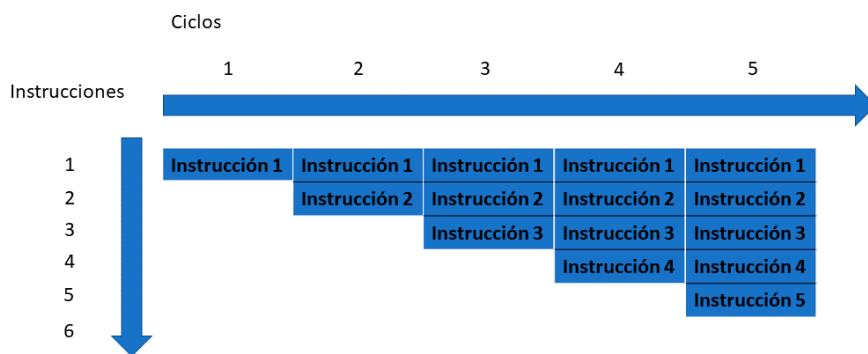


Figura 3.19: Conjunto de instrucciones procesadas de forma segmentada.

3.2.1. Procesador segmentado con solución a los problemas de segmentación por software

El procesador segmentado con solución a los problemas de segmentación por software (ver Figura: 3.20) es el primer procesador bajo la arquitectura de tuberías o pipeline, en la cual se conecta por canales directos los diferentes segmentos o etapas del procesador. En este procesador se tienen 5 etapas.

- En la primera etapa se realiza la búsqueda de la instrucción.
- En la segunda etapa se realiza la decodificación de la instrucción.
- En la tercera etapa se realiza la ejecución de la instrucción.
- En la cuarta etapa se realiza la lectura o escritura sobre la memoria de datos.
- En la quinta etapa se almacena el resultado en la unidad de registros.

Este procesador tiene la particularidad de no tener identificador de errores. Estos errores son aquellos en los cuales las instrucciones tienen registros dependientes de tipo 1 o tipo 2. Al ser la primera versión del procesador con arquitectura de segmentación. La solución que se propone para resolver los problemas es ingresando instrucciones comodín `nop`, con la intención de simular la lectura de una instrucción vacía y consumir el tiempo necesario mientras se realiza el proceso en el segmento que tiene la instrucción con dependencia, `nop` significa *NotOperation*.

Nota: las instrucciones `nop` en lenguaje ensamblador son equivalentes a una operación usualmente suma, donde el registro destino debe ser 0, de la siguiente manera: `nop = add x0, x0, x0`.

A continuación se presenta el diseño estructural del procesador segmentado, con solución por software. Los módulos son similares al procesador monociclo, pero se tienen los segmentos que se muestran en pequeños módulos, que tienen entrada de reloj, adicionalmente se han llamado con su nombre de salida y con guion bajo seguido del segmento o etapa a la que pertenezcan, por ejemplo para referirse a la dirección de la primera instrucción en etapa de *fetch* se referencia de la siguiente manera: *PC_fe*.

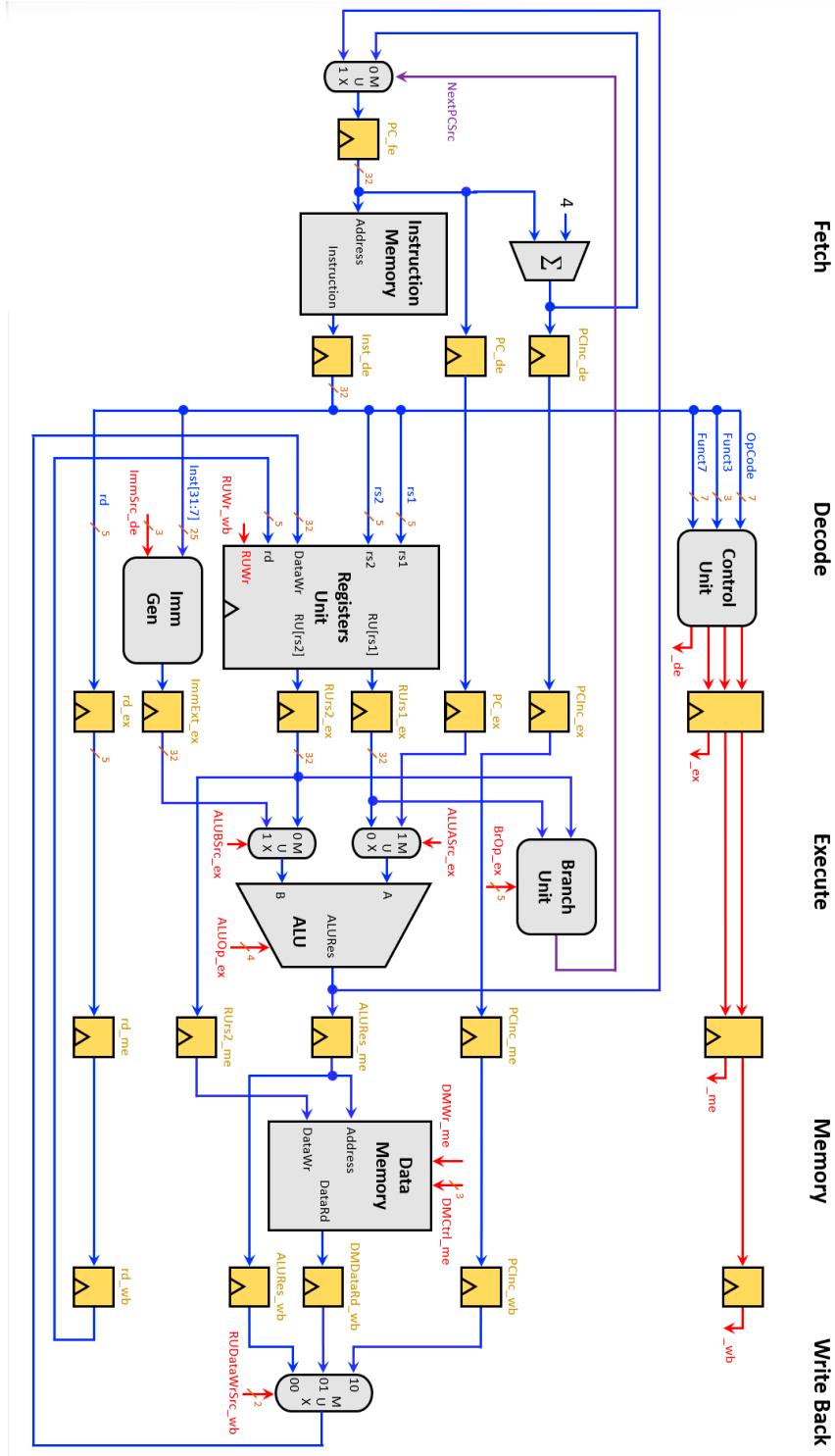


Figura 3.20: Procesador segmentado con solución a los problemas de la segmentación por software.

3.2.2. Segmentos del procesador

En esta subsección se estudian las etapas o segmentos que componen la arquitectura del procesador segmentado a cinco etapas, estas permiten mejorar el rendimiento en comparación con el procesador monociclo. Finalmente, se presentará una visión detallada de cada etapa, su funcionamiento y su conexión con las demás.

Etapa de búsqueda de instrucción (Fetch)

El proceso de la etapa Fetch (ver Figura: 3.21), inicia recibiendo la señal PC (Program Counter) en la cual se encuentra la dirección de la instrucción que se desea recuperar de la memoria de instrucciones, el cual se localiza y se recupera la instrucción completa de 32 bits para su debido procesamiento en los siguientes módulos.

Cuando se recupera la señal PC, esta también entra a un sumador constante donde aumentan la dirección en 4 unidades ($PC + 4$) Se lleva de nuevo al multiplexor inicial. Cuando una instrucción previa de transferencia de control de salto requiera su modificación; el multiplexor elige en este caso la dirección específica del salto y no la siguiente instrucción ($PC + 4$). En esta etapa solo se cuenta con el registro propio del PC.

Etapa de decodificación y lectura de operandos (Decode)

En este proceso (Instruction decode) (ver Figura: 3.22), se decodifica la instrucción recuperada del Instruction Memory. En el momento de la decodificación, la unidad de control recibe 3 parámetros (OpCode, Funct3, Funct7) con los cuales se encarga de gestionar todas las señales de control requeridas para ejecutar la instrucción en este proceso. Estas señales se encuentran en la parte superior de la etapa, con el indicador *de*.

Dentro del proceso de decode está el módulo Immediate Generator, el cual recibe la señal de control `ImmSrc_de` y parte de la instrucción (`Inst[31:7]`). Se utiliza la señal de control para extender el signo del in-

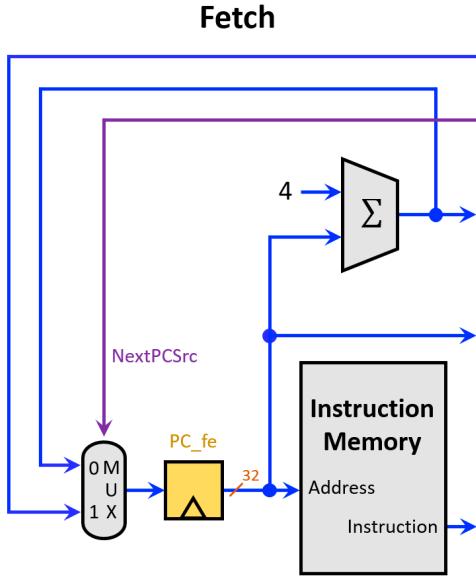


Figura 3.21: Etapa de búsqueda de instrucción.

mediato recibido en el fragmento de instrucción ($\text{Inst}[31:7]$), retornando el resultado en un registro de 32 bits que será utilizado en el siguiente proceso.

Además, se tiene la unidad de registros. Este módulo recibe dos identificadores ($rs1$ y $rs2$) para ser leídos de los registros. Se retorna el contenido de estos identificadores por medio de las salidas $RU[rs1]$ y $RU[rs2]$ respectivamente. Estas salidas serán utilizadas en el siguiente proceso.

Los registros propios de este proceso son: $PC + 4$ (identificador de la siguiente instrucción), PC (identificador de la instrucción actual), Inst (instrucción completa).

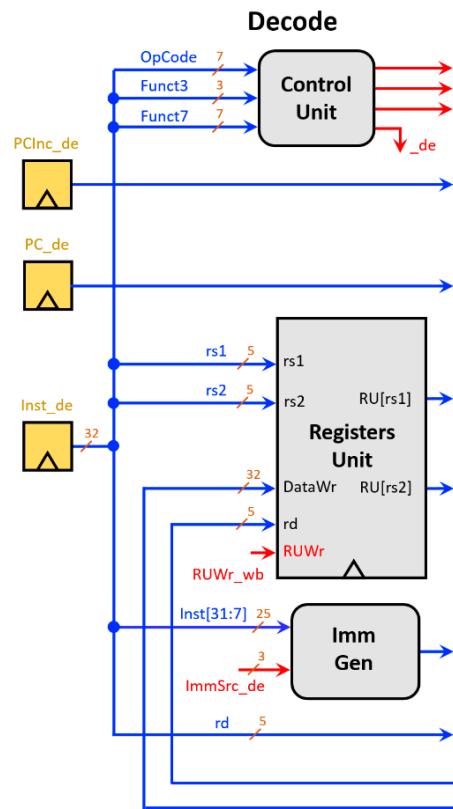


Figura 3.22: Etapa de decodificación y lectura de operandos.

Etapa de ejecución (Execute)

En esta Etapa del procesador, las señales de control con sufijo *ex* generadas en la etapa de decodificación son las siguientes: BrOp, ALUASrc, ALUBSrc y ALUOp. Después de que estas señales son asignadas, cada una ingresa a su respectivo módulo (ver Figura: 3.23).

El *branch Unit* hace parte de la Etapa de *Excute*. A este módulo ingresan los registros de RU[rs1] y RU[rs2], los cuales son comparados mediante una operación lógica especificada con la señal de control BrOp, dando como resultado un valor binario que determina si se debe ejecutar la siguiente instrucción o se hace un desplazamiento. Este resultado es enviado a la Etapa Fetch.

En este proceso se encuentra el módulo ALU al cual están conectados dos multiplexores encargados de seleccionar la entrada para los registros A y B. El multiplexor con la señal de control ALUASrc se encarga de seleccionar entre las señales PC o RU[rs1] para ser ingresada a la entrada A. El multiplexor con la señal de control ALUBSrc selecciona entre los registros RU[rs2] e ImmExt ingresando el resultado a la entrada B. Después de obtener estas entradas se define la operación a realizar mediante la señal de control ALUOp entregando el resultado de la misma para la siguiente etapa.

Los registros propios de este proceso son: PCinc (identificador de la siguiente instrucción), PC (identificador de la instrucción actual), RU[rs1] (contenido del registro identificado en rs1), RU[rs2] (contenido del registro identificado en rs2), ImmExt (Inmediato extendido), rd (identificador registro destino).

Etapa de acceso a memoria (Memory)

En esta Etapa (ver Figura: 3.24) se realizan operaciones en el módulo *Data Memory* solo en el caso en que el formato de instrucción sea de carga o almacenamiento (*load o store*). Se utilizan dos señales de control (DMWr, DMCtrl). Este módulo está leyendo en todo momento la dirección que llega desde la entrada Address en el formato especificado por la señal DMCtrl

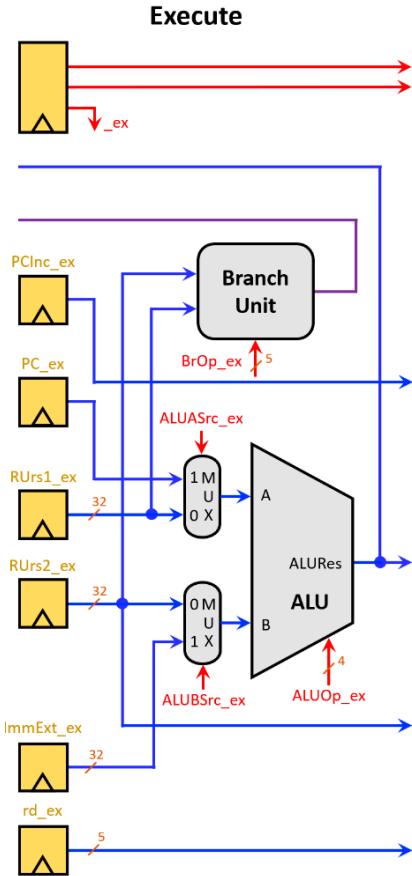


Figura 3.23: Etapa de ejecución

(Byte, HalfWord o Double). El resultado de la lectura es entregado a la salida DataRd para ser procesado en la siguiente etapa. En caso de que la señal DMWr esté activa, la dirección ingresada por Address es accedida para almacenar la información que trae el registro DataWr desde la etapa anterior.

Con este proceso se encuentran los siguientes registros propios: PCinc (identificador de la siguiente instrucción), ALURes (Resultado de la salida de la ALU en la etapa anterior), RU[rs2] (contenido del registro identificado en rs2 desde la etapa anterior), rd (identificador registro destino). Tanto el registro ALURes como el registro rd se entregan a la siguiente etapa para ser procesados.

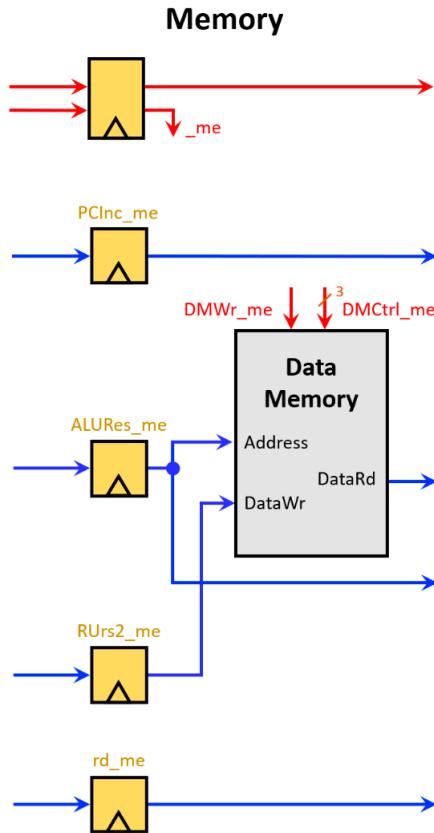


Figura 3.24: Etapa de acceso a memoria.

Etapa de escritura de resultado (Write Back)

En el momento de ejecución de la etapa de escritura de resultado (ver Figura: 3.25), se tienen dos señales de control (`RUDataWrSrc`, `RUWr`) donde se utiliza `RUDataWrSrc` dentro del multiplexor 3 a 1 para seleccionar cuál registro propio de este proceso (`PCInc`, `DMDataRd`, `ALURes`) se almacenará en el módulo Register Unit dependiendo del tipo de instrucción que se esté ejecutando. Después de hacer el proceso de selección, se verifica la señal `RUWr` para decidir si se guarda en la matriz de registros el valor saliente del multiplexor en el indicador del registro `rd` de la etapa. De esta manera cumpliendo todos los procesos en las 5 etapas.

Los registros propios de este proceso son: `PCInc` (identificador de la siguiente instrucción), `DMDataRd` (resultado de la lectura de la Data Memory), `ALURes` (Resultado de la salida de la ALU en la etapa anterior), `rd` (identificador registro destino).

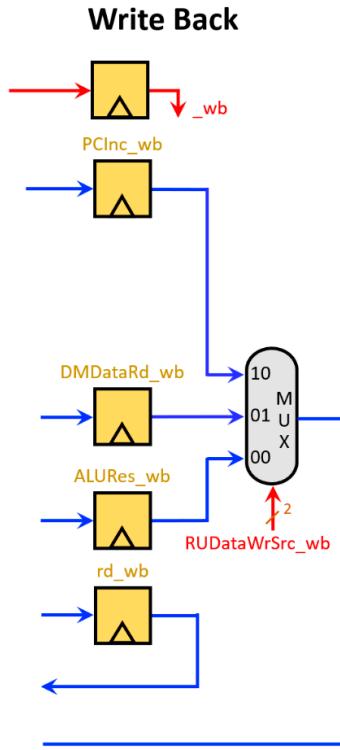


Figura 3.25: Etapa de escritura de resultado.

3.2.3. Procesador segmentado con solución a los problemas de segmentación por hardware

El procesador segmentado con solución a los problemas de segmentación por hardware (ver Figura: 3.26), resuelve los problemas de dependencia de registros que se generan a partir de la segmentación. Esta solución tiene como base añadir un conjunto de módulos, señales y conexiones que detectan cuando suceden estos problemas de dependencia, además de anticipar los registros dependientes a través del flujo del procesador.

Dentro de la arquitectura de este procesador se tiene una característica especial para su funcionamiento y es que el módulo de registros debe hacer escritura en flancos de bajada, para que la instrucción que está en la última etapa sobrescriba y eso le tomaría solo medio ciclo de reloj. Lo cual genera que en la etapa de decodificación tenga el valor correcto del registro dependiente.

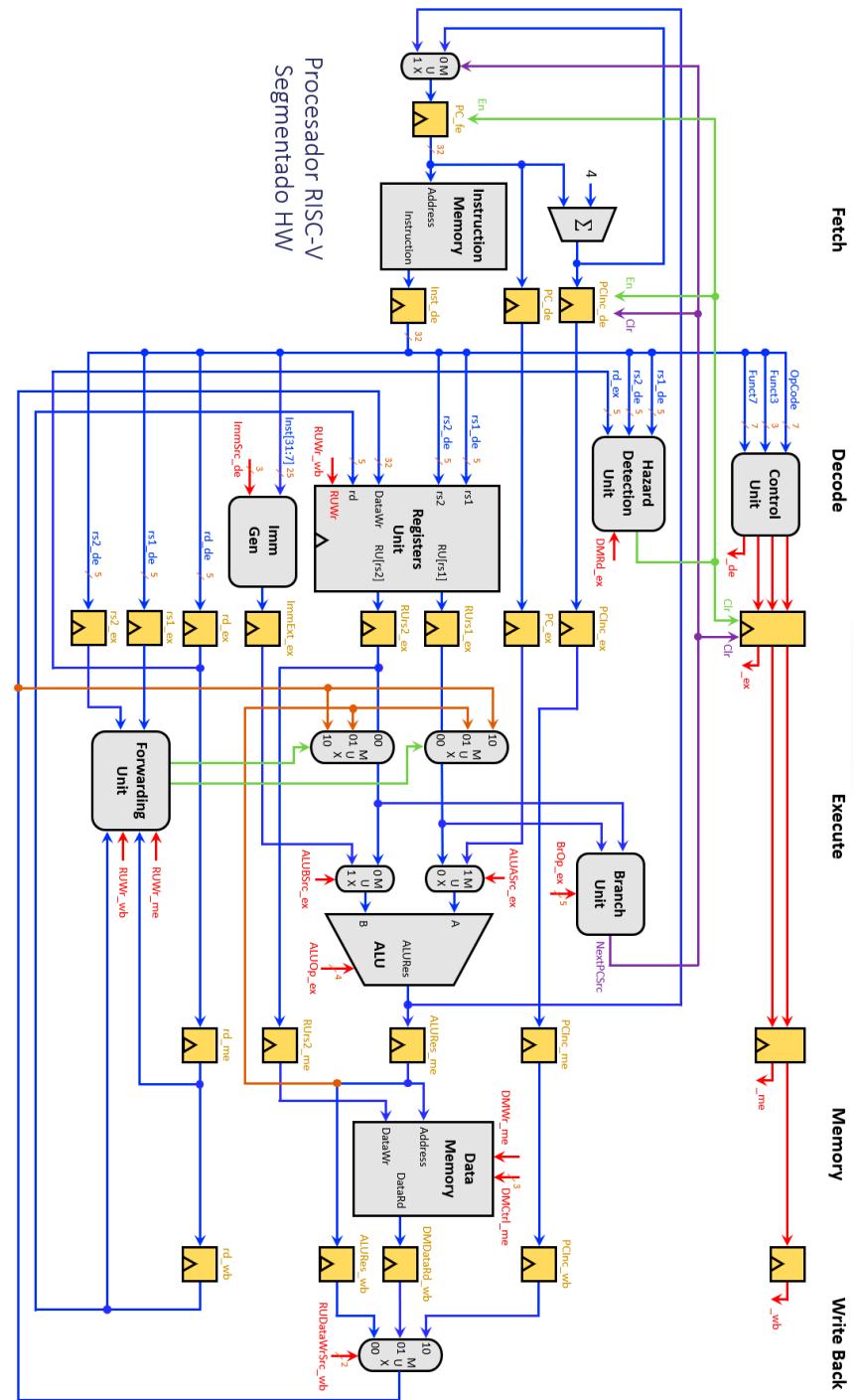


Figura 3.26: Procesador segmentado con solución a los problemas de segmentación por hardware.

Análisis de los problemas de la segmentación asociados a la dependencia de datos

Los problemas de dependencia de datos se generan cuando existen instrucciones contiguas (cero o una instrucción de diferencia) las cuales comparten los mismos registros fuente para realizar operaciones. Este problema se ubica en las etapas de ejecución, memoria y Write-back.

El tipo de dependencia está asociado a cuantas instrucciones hay de por medio. Si son instrucciones contiguas se conoce esto como dependencia de registro tipo 1, de lo contrario el otro caso es que haya una instrucción de por medio, a lo cual se lo conoce como dependencia tipo 2.

Los siguientes son los 2 tipos de dependencia tipo 1:

- Dependencia de registro fuente uno (ver Figura:3.27), de la segunda instrucción

```
add x8, x9, x21
sub x18, x8, x22
```

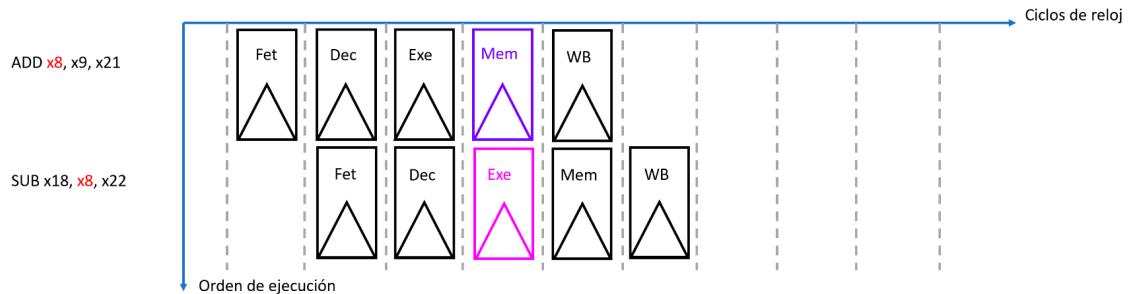


Figura 3.27: Dependencia de registro fuente uno.

- Dependencia de registro fuente dos (ver Figura:3.28), de la segunda instrucción

```
add x8, x9, x21
sub x18, x12, x8
```

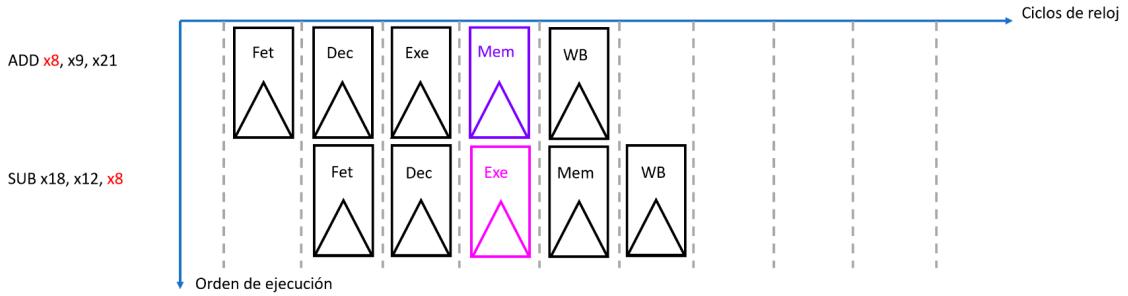


Figura 3.28: Dependencia de registro fuente dos

Los siguientes son los 2 tipos de dependencia tipo 2.

- Dependencia de registro fuente uno, de la primera instrucción y una instrucción de por medio (ver figura: 3.29).

```
add x8, x9, x21
lw x4, x2, 8
sub x18, x8, x22
```

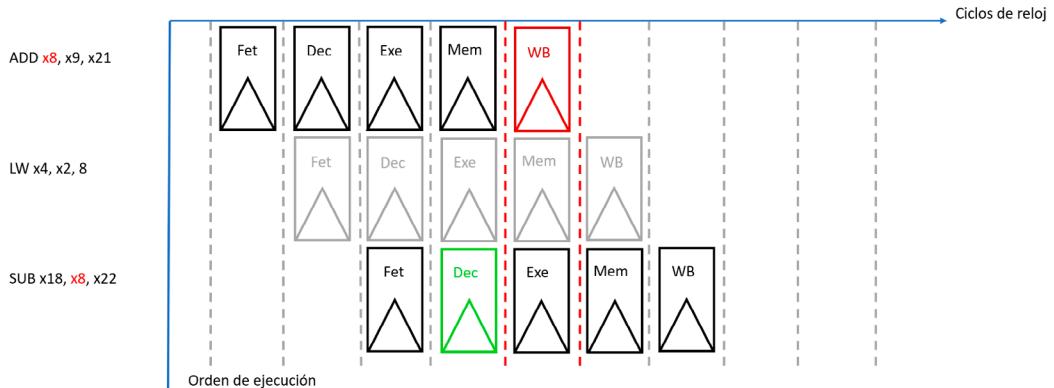


Figura 3.29: Dependencia de registro fuente uno.

- Dependencia de registro fuente dos, de la primera instrucción y una instrucción de por medio (ver Figura: 3.30).

```

add x8, x9, x21
lw x4, x2, 8
add x18, x12, x8

```

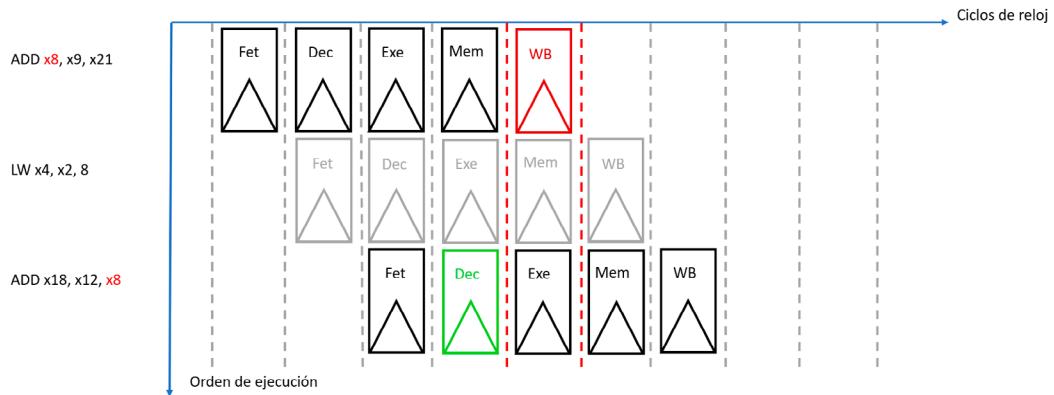


Figura 3.30: Dependencia de registro fuente dos.

Ejemplo de dependencia: Como ejemplo se asume dos instrucciones aritméticas Instrucción 1 (Inst1) e Instrucción 2 (Inst2). La Inst1 está escribiendo sobre un registro x8 y a su vez x8 es un registro que hace parte de la Inst2 que está realizando lectura. En este punto se genera un problema en la segmentación, ya que en el momento donde la Inst1 está en la etapa de execute, la cual está haciendo el cálculo de lo que hay en el registro destino x8, mientras que la Inst2 ya estando en el decode, decodifico el contenido de x8. Algorítmicamente, la Inst2 está leyendo un contenido x8 que no es correcto. Algorítmicamente, se esperaría que se tuviese el resultado de la Inst1, lo cual genera una dependencia de datos con una instrucción contigua.

Unidad de anticipación

La unidad de anticipación compara los registros de la instrucción en etapa de ejecución con los registros de la etapa de memoria, para evaluar

si tienen dependencia tipo 1, también compara los registros de la etapa de ejecución con los registros de la etapa de write-back, para evaluar si tienen dependencia tipo 2.

Este módulo también debe verificar dependencias dobles (ver Figura: 3.31), donde el registro destino se encuentra dentro de las dos instrucciones siguientes.

```
add x8, x9, x21
sll x8, x8, x2
sub x18, x8, x22
```

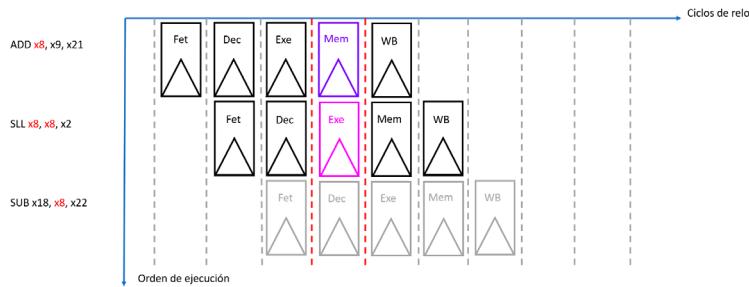


Figura 3.31: Dependencia doble.

En este caso de dependencia, se evalúan primero los registros de la instrucción en la etapa de memoria y después se procede a evaluar los registros en la etapa de Write-back, en consecuencia este caso sería de tipo 1.

Solución de problema de dependencia de datos: La solución analítica para el problema de dependencia es adicionar un módulo de anticipación, el cual se encarga de comparar los registros asociados a las instrucciones y verificar que tengan dependencia. De haber dependencia, esta unidad debe anticipar el resultado de la unidad aritmético-lógica, para garantizar que el registro de entrada de la instrucción dependiente contenga su información actualizada.

Pasos de la unidad de anticipación (Forwarding Unit)

- Paso 1. Recibe los registros fuentes de la etapa de ejecución, registro fuente 1 y registro fuente 2,(rs1_ex y rs2_ex).
- Paso 2. Recibe el registro destino de la instrucción en etapa de memoria y compara si es igual a alguno de los registros fuente de la anterior instrucción en etapa de ejecución. De cumplirse la condición, envía un valor 2'b01' a los multiplexores 5 y 6 (MUX5 y MUX6).
- Paso 3. Recibe el registro destino de la instrucción en etapa de write back y compara si es igual a alguno de los registros fuente de la etapa de ejecución. De cumplirse la condición, envía una señal de 2'b10' a los multiplexores 5 y 6 (MUX5 y MUX6).

Las señales de escritura en la unidad de registros (RUWr_me y RUWr_wb), en los segmentos de memoria y write-back, indican a la unidad de anticipación si la instrucción a evaluar está en la etapa de memoria o de write-back.

Problemas de la unidad de anticipación: El problema principal surge cuando la unidad de anticipación no cumple su función, esto sucede cuando la instrucción que está en etapa de memoria es de tipo Load, además la instrucción que está en la etapa de ejecución es de tipo aritmético o lógico y tienen dependencia tipo 1. En consecuencia, el valor anticipado es la dirección de la instrucción, cuando en realidad lo que se está esperando es que al momento de hacer la etapa de ejecución el valor del registro dependiente ya tenga cargado el dato, lo cual se realiza apenas en un ciclo adicional.

Procesador segmentado con unidad de detección de riesgos

El procesador segmentado con unidad de detección de riesgos, evalúa si hay instrucciones de lectura de memoria en las cuales sus registros destino tengan dependencia de tipo 1, eso con el propósito de congelar las etapas

de fetch y decode, para que dentro de los siguientes segmentos el único proceso que se realizará será de carga de memoria, que se efectuará en el siguiente ciclo de reloj.

Por ejemplo, se tiene el siguiente par de instrucciones que presentan el problema de dependencia de carga.

```
lw x20, 12(x9)  
add x8, x20, x21
```

Se presenta una instrucción de lectura de memoria en un registro destino **x20**, en la etapa de ejecución, adicionalmente el registro fuente 1 de la instrucción que está en etapa de decodificación, depende del registro destino que va a ser cargado de la memoria.

La solución a este problema es detener las etapas de fetch y decode e insertar una instrucción NOP, de tal manera que la instrucción que está en etapa de ejecución pase a la etapa de memoria, en la cual puede acceder a la memoria y realizar la carga.

Unidad de detección de riesgos

La unidad de detección de riesgos está ubicada entre la etapa de decodificación y etapa de ejecución, sus entradas son los registros fuente 1 y 2 de la etapa de decodificación; adicionalmente el registro destino de la etapa de ejecución, también cuenta con una señal de control que permite conocer si la instrucción que está en etapa de ejecución va a realizar lectura sobre la memoria de datos. Las salidas de esta unidad son una señal que detiene la lectura de instrucciones en la etapa de fetch, una señal que detiene la lectura de instrucciones en la etapa de decodificación y una señal que limpia o borra el contenido de la etapa de ejecución; esta última señal permite que en el siguiente ciclo entre una instrucción comodín NOP.

Los pasos que cumple este módulo son los siguientes:

- Paso 1. Identificar si la instrucción que está en etapa de ejecución va a ser lectura de memoria y adicionalmente va a escribir en un

registro que sea igual al registro fuente 1 o registro fuente 2 de la instrucción que está en etapa de decode, esto representa un problema de dependencia.

- Paso 2. Una vez detectada la dependencia de datos con una lectura de memoria, se activa una señal que deshabilita la lectura de las etapas de decode y de la etapa de fetch, en otras palabras congela la etapa de fetch y decode, adicionalmente hace un clear síncrono en la etapa de ejecución, este clear síncrono básicamente lleva a que en el próximo ciclo se inserte una instrucción que no hace nada y así permitir que la instrucción que está en etapa de ejecución pase a etapa de memoria.

Ejemplo de dependencia de datos con instrucción Load y Add:

```
lw x20, 12(x9)
add x8, x20, x21
```

Siguiendo el ejemplo de dependencia de datos con instrucción de carga y una suma, la instrucción de carga que está en etapa de ejecución pasa a etapa de memoria y en la etapa de ejecución se inserta una instrucción comodín, de tal manera que queda una instrucción comodín entre la instrucción de carga (*Load*) y la instrucción tipo-R (*Add*); en ese momento se desactiva esa señal porque no hay dependencia, se habilita la escritura de la etapa de fetch y decode. En el próximo ciclo la instrucción Load pasa a etapa de write-back, adicionalmente la instrucción comodín pasa a etapa de memoria y en la instrucción Add pasa a etapa de ejecución. Cuando eso sucede, la unidad de anticipación va a hacer el salto correcto, ya que va a preguntar por una dependencia de datos con la instrucción en etapa de memoria, pero en esta etapa de memoria está la instrucción comodín que permite consumir un segmento sin realizar nada, entonces no hay dependencia de datos, luego se pregunta por la dependencia de datos con la instrucción de write-back y en efecto hay dependencia de datos entre la etapa de ejecución y la etapa de write-back, por lo tanto, la unidad de anticipación hace el cálculo del salto que es efectivamente el valor que leyó de la memoria. De esta manera

funcionan de manera armónica la unidad de detección de riesgos y la unidad de anticipación.

Detección de saltos efectivos: Finalmente, para los saltos efectivos lo que se hace es permitirle a la branch unit que cuando un salto sea efectivo borre la instrucción que está en etapa de fetch para que no pase a decode y que el clear síncrono de decode no pase a execute, y eso se realiza con las dos señales que son clear síncrono. Eso le permite a la branch unit borrar las dos instrucciones que se colaron de forma incorrecta porque el salto fue efectivo; si el salto no fue efectivo, pues se deja que las instrucciones que están en la cola sigan porque eso es lo que se espera.

El procesador segmentado es una tecnología muy interesante, ya que permite acelerar el rendimiento del procesamiento de datos con solo la reorganización de las unidades, por lo que este no representa un coste mayor para la construcción. Es muy práctico por el hecho de poder ejecutar varias instrucciones a tiempo, pero esto es tanto una ventaja como una desventaja, ya que la complejidad para entender su funcionamiento es alta, pero aparte de esto, el procesador es mucho más rápido, ya que se usan los recursos en forma simultánea para mayor capacidad de procesamiento haciendo uso de los cinco segmentos que permite hasta cinco procesos a la vez dando así un procesador con grandes capacidades de procesamiento.

3.3. Ejercicios Capítulo 3

1. ¿Cuántos ciclos de reloj toma procesar una instrucción tipo R en el procesador monociclo?
2. ¿Qué tipo de instrucciones necesitan que la señal de control **RUDataWrSrc** tenga el valor de 00?
3. ¿Cuántos ciclos de reloj toma procesar una instrucción tipo R en el procesador segmentado?

4. ¿Qué nombre lleva la siguiente dependencia en lenguaje ensamblador RISC-V?

**add x7, x8, x9
subi x24, x7, 10**

5. ¿Qué tipo de instrucciones necesitan que la señal de control ALUASrc tenga el valor de 0?
6. ¿Cuántos ciclos de reloj toma procesar una instrucción tipo S en el procesador segmentado?
7. ¿Cuántos ciclos de reloj toma procesar una instrucción tipo S en el procesador monociclo?
8. ¿Qué nombre lleva la siguiente dependencia en lenguaje ensamblador RISC-V?

**slli x8, x21, 8
sh x20, 2(x9)
sub x10, x8, x22**

9. ¿Cuál es la diferencia que hay en el número de ciclos entre ejecutar una instrucción de tipo B en el procesador monociclo y el procesador segmentado?
10. ¿Qué nombre lleva la siguiente dependencia en lenguaje ensamblador RISC-V?

**xor x20, x9, x8
lw x21, 2(x22)
sub x14, x13, x20**

11. ¿Qué tipo de instrucciones necesitan que la señal de control DMWr tenga el valor de 1?

12. ¿Qué tipo de instrucciones necesitan que la señal de control RUWr tenga el valor de 1?

4

CAPÍTULO CUATRO

Sistemas de Entrada y Salida

4.1. Espacio de direccionamiento

Los dispositivos de entrada y salida permiten interactuar al usuario con el procesador, existen diferentes dispositivos, en esta sección se presentan 4 periféricos los cuales serán accedidos a partir de la memoria de datos, la cual se segmenta en 4 direcciones base.

- Controlador de video: Permite la conexión y visualización de 4800 caracteres en una pantalla de resolución 800x600 píxeles
- Controlador teclado: Realiza el almacenamiento de símbolos o caracteres
- Controlador de Switch: Realiza el almacenamiento de 10 diferentes valores asociados a un conjunto de 10 switches
- Controlador de LED: Permite la visualización de un registro de 10 bits que representa 10 leds

Cada periférico está mapeado en la memoria de datos, como una dirección constante llamada Base Address, por lo general estas direcciones de mapeado de periféricos está en las direcciones bajas, las cuales son aquellas que están antes de la dirección `0x0...4000`.

Desde el punto de vista de software, se debe acceder al base address de los periféricos con instrucciones de tipo load o store, permitiendo el acceso a la información de mapeo, a continuación se presenta un esquema básico de direcciones base, las cuales serán definidas para un conjunto de 10 LED's, 10 switches, un teclado y una pantalla VGA.

En principio para mapear el módulo VGA se realiza el cálculo preliminar de la cantidad de caracteres que alcanzan en una VGA de 80x60, de esta manera se garantiza que la interacción del procesador con esta memoria se haga a partir de una memoria de caracteres más no con cada pixel. Este cálculo da un valor de 4600 caracteres que pueden ser mapeados en la dirección base de la memoria VGA. La cantidad de bits necesarios para codificar el valor de caracteres es de 15 bits.

El espacio de memoria necesario para interactuar con 10 leds o 10 switches es de 10 bits, por lo cual con un registro es suficiente para almacenar la información de los leds o switches.

El espacio de memoria para un teclado es de 8 bits, ya que con esta cantidad se puede representar el código ASCII, de 256 diferentes elementos.

Estos periféricos están mapeados dentro de la memoria de datos real, ahora para acceder por ejemplo a la información de leds, basta con acceder a la base address de leds y hacer una instrucción de carga.

Como existen 5 posibles memorias de mapeo, es necesario un conjunto de bits para representar las direcciones de mapeo, las direcciones están en la tabla 4.1 que serán los bits 15, 16, 17 de la dirección.

Dispositivo	selector	Dirección base
Controlador de video	000	0x00000000
Controlador de teclado	001	0x00008000
Controlador de switch	010	0x00001000
Controlador LED	011	0x00018000
Memoria de datos	100	0x00020000

Tabla 4.1: Selección de Base Address.

4.1.1. Controlador de video

El VGA (ver Figura: 4.1), es un tipo estándar de puerto para dispositivos de video como son los monitores, proyectores y televisores desarrollados por IBM e introducido en 1987. VGA proporciona una visualización en color con una resolución de 640 x 480 con una frecuencia de actualización de 60 Hz y 16 colores mostrados a la vez.

Protocolo VGA

El protocolo VGA cuenta con una señal de reloj (CLK), señal de sincronización horizontal (HS 1 bit), señal de sincronización vertical (VS 1 bit) y salida RGB (VGA RGB 24 bits). El proceso se divide en dos partes: sincronización horizontal y vertical. La sincronización horizontal se considera como una línea en pantalla en la cual la señal HS debe ir a 0V cuando se está en el intervalo de sincronización Sync (ver Figura a: 4.2) y la señal se conduce a 1V mientras se encuentra entre los intervalos de Back Porch (ver Figura b: 4.2), visualización (Display (ver Figura c: 4.2)) y Front Porch (ver Figura d: 4.2). Los pulsos de la señal HS son cíclicos, debido a que cuando finaliza el intervalo Front Porch (ver Figura d: 4.2), continúa el intervalo de sincronización (ver Figura: 4.2), así también el pulso de HS regresando a 0V. En el intermedio de un ciclo de la señal HS se encuentra un intervalo de visualización (ver Figura: 4.2), en el cual la salida RGB debe representar el color en bits que se quiere visualizar en esa línea. Para los intervalos dentro de un ciclo de la señal HS los tiempos se tienen en unidades de microsegundos (ver Tabla: 4.3), utilizando la señal CLK para medir el tiempo de estos intervalos. Cuando se habla de la sincronización vertical (ver Tabla: 4.4), se puede definir de la misma manera que la sincronización horizontal, exceptuando que la medida para determinar los tiempos de los diferentes intervalos (Sync, Back Porch, Display interval, Front Porch) no dependen directamente de una medida de tiempo, sino del número líneas (ciclo de la señal HS tabla: 4.4). Tanto para los ciclos de HS y VS la salida RGB debe ir a 0V mientras no se encuentren en tiempos de visualización.

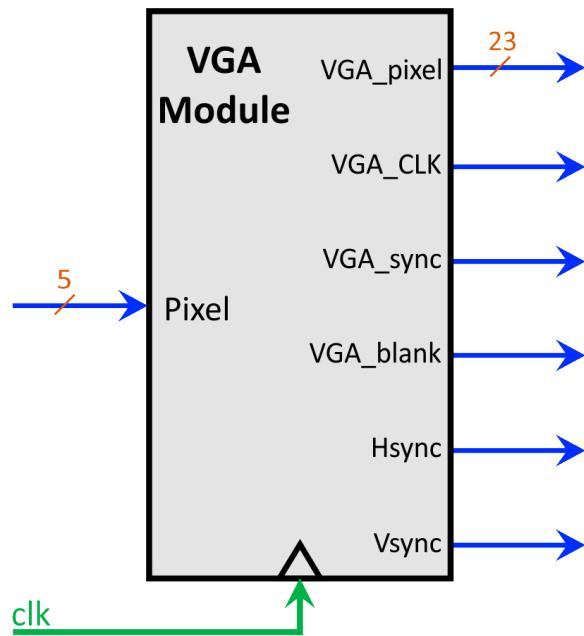


Figura 4.1: Diagrama del controlador de video.

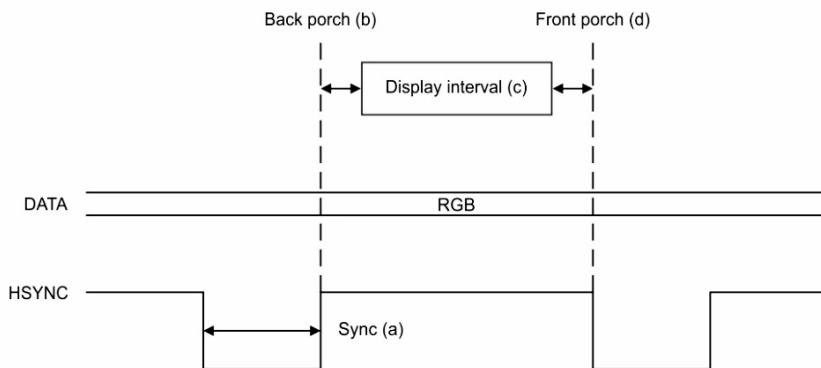


Figura 4.2: Tiempos de Porch

VGA mode		Horizontal Timing Spec				
Configuration	Resolution (HxV)	a(μs)	b(μs)	c(μs)	d(μs)	Pixel clock (MHz)
VGA (60Hz)	640x480	3.8	1.9	25.4	0.6	25

Figura 4.3: Horizontal Timing Specification.

VGA mode		Vertical Timing Spec				
Configuration	Resolution (HxV)	a(lines)	b(lines)	c(lines)	d(lines)	Pixel clock (MHz)
VGA (60Hz)	640x480	2	33	480	10	25

Figura 4.4: Vertical Timing Specification.

4.1.2. Controlador de teclado

El conector PS/2 o puerto de teclado (ver Figura: 4.5), toma su nombre de la serie de computadoras IBM Personal System/2 que es creada por IBM en 1987, y empleada para conectar teclados y ratones.

La interfaz tiene dos líneas de señales principales, Datos y Reloj. Normalmente, la transmisión es del dispositivo al host. Para transmitir un byte, el dispositivo simplemente emite una trama de datos en serie (ver Figura: 4.6), (incluidos 8 bits de datos y un bit de paridad) en la línea de datos en serie, ya que alterna la línea de reloj una vez por cada bit.

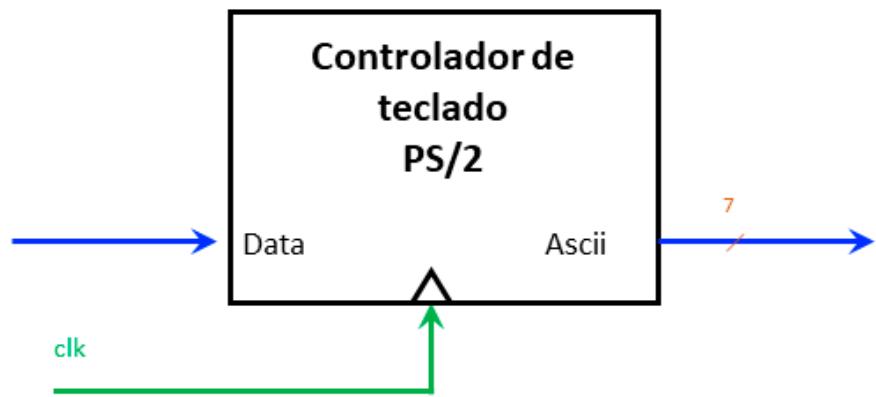


Figura 4.5: Diagrama controlador teclado

4.2. Ejercicios Capítulo 4

1. ¿Desde dónde se puede considerar que una dirección de memoria de datos es una dirección baja?

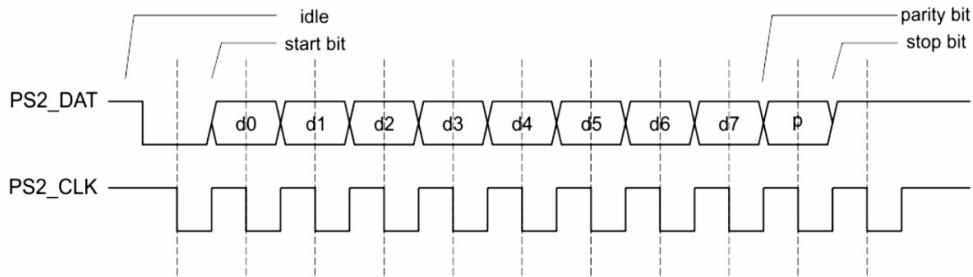


Figura 4.6: Diagrama de tiempos PS2.

2. ¿A través de qué segmento de memoria interactúa el procesador con la dirección base de la memoria VGA?
3. ¿De qué unidad depende el cambio de intervalos en la sincronización horizontal dentro del protocolo VGA?
4. ¿De qué unidad depende el cambio de intervalos en la sincronización vertical dentro del protocolo VGA?
5. Haga una tabla de Tiempos de Porch contra Salida(Hsync y Vsync) en el que indique qué valor en voltaje tienen dichas salidas en determinado intervalo.
6. ¿Qué valor de periodo tiene el Pixel clock dentro del módulo VGA?
7. ¿Cuántos periodos de reloj le toma al módulo VGA pasar por la etapa de Sync?

Apéndice A

Estándar internacional para unidades

Los estándares son la base para la representación de unidades. En el mundo existe una organización internacional de normalización, compuesta por un conjunto de expertos en múltiples temas, esta organización desarrolla y publica estándares internacionales (ISO) [33]. Este trabajo por lo general se realiza en conjunto con la comisión internacional electrotécnica (IEC) [32], el objetivo común es promover la cooperación internacional con base en estandarizaciones en el campo de la eléctrica y la electrónica.

Algunos estándares comúnmente usados por la comunidad científica son:

- Estándar de gestión de calidad (ISO 9000)
- Estándar de gestión del medio ambiente (ISO 14000)
- Estándar de salud y seguridad (ISO 45001)
- Estándar para magnitudes físicas y unidades de medida (ISO/IEC 80000)

A.1. Estándar ISO/IEC 80000

El estándar internacional para magnitudes y unidades de medida ISO/IEC 80000, algunas de las normas que contiene son referentes a las medidas generales, medidas de mecánica, electromagnetismo, acústica, química y física molecular, ciencia de la información y la tecnología. La siguiente tabla condensa algunas de las más usadas (ver tabla: A.1).

ISO	Nombre
ISO 80000-1	General
ISO 80000-2	Signos y símbolos matemáticos para uso en ciencias naturales y tecnología
ISO 80000-3	Espacio y tiempo
ISO 80000-4	Mecánica
ISO 80000-5	Termodinámica
IEC 80000-6	Electromagnetismo
ISO 80000-7	Luz
ISO 80000-8	Acústica
ISO 80000-9	Química y física molecular
ISO 80000-10	Física atómica y nuclear
ISO 80000-11	Números característicos
ISO 80000-12	Física del estado sólido
IEC 80000-13	Ciencia de la información y tecnología
IEC 80000-14	Telebiométrica relativa a la fisiología humana

Tabla A.1: Estándares ISO/IEC 80000.

A.1.1. Estándar para ciencia de la información y tecnología ISO/IEC 80000-13

El valor base de referencia para generar el estándar internacional para ciencia de la información y tecnología (ISO/IEC 80000-13) es el **byte** y su representación es la letra mayúscula *B*. El byte es un conjunto de 8 bits, bajo el estándar internacional se hace referencia a la cantidad de bytes en potencias de 10. El elemento inicial que representa la potencia de 2^0 y es un byte u ocho bits, el siguiente elemento es 2 a la potencia de 10, que es conocido como **kibibyte**, o kilo binary bit, que representa el valor de 1024, de manera general la ecuación es de 2^{n+10} donde *n* es potencia de 10 e inicia en 0.

A continuación se muestra una tabla con los prefijos, el símbolo del prefijo y el símbolo del múltiplo del byte (ver tabla: A.2).

Prefijo	Símbolo del prefijo	Símbolo del múltiplo del byte
Valor de referencia		B
kibi	Ki	KiB
mebi	Mi	MiB
gibi	Gi	GiB
tebi	Ti	TiB
pebi	Pi	PiB
exbi	Ei	EiB
zebi	Zi	ZiB
yobi	Yi	YiB

Tabla A.2: Nomenclatura en base a los prefijos, símbolos del prefijo y símbolos del múltiplo del byte.

Ahora la forma del nombre resultante está conformado por el prefijo más la palabra byte, de acuerdo a la siguiente tabla (ver tabla: A.3).

Prefijo + byte	Factor y valor en el ISO/IEC 80000-13
byte	$2^0 = 1$
kibibyte	$2^{10} = 1024$
mebibyte	$2^{20} = 1\,048\,576$
gibibyte	$2^{30} = 1\,073\,741\,824$
tebibyte	$2^{40} = 1\,099\,511\,627\,776$
pebibyte	$2^{50} = 1\,125\,899\,906\,842\,624$
exbibyte	$2^{60} = 1\,152\,921\,504\,606\,846\,976$
zebibyte	$2^{70} = 1\,180\,591\,620\,717\,411\,303\,424$
yobibyte	$2^{80} = 1\,208\,925\,819\,614\,629\,174\,706\,176$

Tabla A.3: Nomenclatura de los bytes según el estándar ISO/IEC 80000-13.

Apéndice B

Glosario

- **ASCII:** American Standard Code for Information Interchange, código estándar americano para el intercambio de información; el código alfanumérico más utilizado.
- **Álgebra booleana:** Las matemáticas de los circuitos de la lógica.
- **ALU:**(Arithmetic Logic Unit) Unidad aritmético lógica; el elemento clave del procesador, que realiza operaciones aritméticas y lógicas.
- **AND:** Operación lógica básica en la que se obtiene una salida verdadera sólo si todas las condiciones de entrada son verdaderas.
- **ANSI:** American National Standards Institute, Instituto Nacional Americano de Normalización.
- **Asíncrono:** Que no tiene ninguna relación temporal fija. Que no ocurre simultáneamente.
- **Binario:** Que tiene dos valores o estados; describe un sistema de numeración en base dos y utiliza el 1 y el 0 como dígitos.
- **Bit:** Dígito binario que puede ser 1 ó 0.

- **Bit de paridad:** Bit añadido a cada grupo de bits de información para hacer que el número de unos sea par o impar en dicho grupo de bits.
- **Bit de signo:** El bit más a la izquierda de un número binario que designa si el número es positivo (0) o negativo (1).
- **Bit más significativo (MSB, Most Significant Bit):** El bit más a la izquierda de un número entero o código binario.
- **Bit menos significativo (LSB, Least Significant Bit):** Generalmente, el bit de más a la derecha de un número entero o código binario.
- **Borrado (clear):** Entrada síncrona utilizada para resetear una etapa del procesador segmentado (pone la etapa de ejecución). Cuando se pone un registro o contador en el estado en que contiene solamente ceros.
- **Byte:** Grupo de ocho bits.
- **Cadena:** Una secuencia contigua de bytes o palabras.
- **Capacidad de palabra:** El número de palabras que una memoria puede almacenar.
- **Capacidad:** Número total de unidades de datos (bits, nibbles, bytes, palabras) que puede almacenar una memoria.
- **Carácter:** Símbolo, letra o número.
- **Carga (load):** Recibir datos de una memoria de datos y almacenarla en un registro.
- **Codec:** Un codificador y decodificador combinado.
- **Codificador:** Circuito digital (dispositivo) que convierte la información a un formato codificado.

- **Código:** Un conjunto de bits ordenados según un patrón único y utilizados para representar información tal como números, letras y otros símbolos. En System Verilog, instrucciones de programa.
- **Código máquina:** Conjunto de instrucciones RISC-V en código binario que el procesador es capaz de comprender.
- **Comparación:** Circuito digital que compara las magnitudes de dos cantidades y produce una salida booleana.
- **Compilador:** Programa de aplicación disponible en paquetes de desarrollo que controla el flujo de diseño y traduce el código fuente en código objeto en un formato que puede ser lógicamente probado o descargado en un dispositivo de destino.
- **Complemento:** El inverso u opuesto de un número. En el álgebra booleana es la función inversa, que se expresa mediante una barra por encima de la variable. El complemento de 1s 0 y viceversa.
- **CPU:** Central processing Unit, unidad central de proceso. Uno de los componentes principales de todas las computadoras que controla el funcionamiento interno y procesa los datos. El módulo de un DSP que procesa las instrucciones del programa.
- **Diagrama de módulos:** Representación gráfica de un conjunto de módulos conectados.
- **Diagrama de tiempos:** Gráfico de formas de onda digitales que muestra la relación temporal existente entre todas las señales y cómo varía cada una respecto a las restantes.
- **Datos:** Información en formato numérico, alfabético o cualquier otro.
- **Decimal:** Describe un sistema de numeración en base diez.

- **Decodificación:** Una etapa de las operaciones pipeline del procesador segmentado en la que las instrucciones se asignan a unidades funcionales y se decodifican.
- **Detección de errores:** Proceso de detectar fallas en un sistema digital.
- **Diagrama de datos:** Descripción gráfica de una secuencia de módulos transmitiendo valores.
- **Digital:** Relacionado con los dígitos o con cantidades discretas; que posee un conjunto de valores discretos, en oposición a tener valores continuos.
- **Dígito:** Símbolo utilizado para expresar una cantidad.
- **Dirección:** Posición de una determinada celda de almacenamiento o grupo de celdas en memoria. Posición de memoria diferenciada, que contiene un byte.
- **Dirección base:** La dirección inicial de un segmento de memoria.
- **Disco duro:** Dispositivo de almacenamiento magnético; normalmente, una pila de dos o más discos rígidos encerrados en un compartimento sellado.
- **DMA:** Direct Memory Access, acceso directo a memoria. Método para conectar de forma directa un dispositivo periférico con la memoria, sin utilizar la CPU como mecanismo de control.
- **Ejecución:** Proceso de la CPU durante el cual se lleva a cabo una instrucción.
- **Ensamblador:** Programa que convierte las instrucciones en ensamblador en código máquina.
- **Entero:** Un número entero.

- **Entrada:** La señal o línea que entra en un circuito. Señal que controla el funcionamiento de un circuito.
- **Entrada/Salida (E/S):** Un terminal de un dispositivo que puede ser utilizado como entrada o como salida.
- **Escritura:** El proceso de almacenamiento de datos en memoria.
- **Etapa:** Fase, segmento o pipeline del proceso de segmentación.
- **FPGA:** Field Programmable Gate Array, matriz de puertas programable sobre el terreno: un dispositivo lógico programable que utiliza una carga LUT como elemento lógico básico y que emplea, generalmente, tecnología de proceso basada en antifusible o basada en SRAM.
- **Frecuencia (f):** El número de impulsos por segundo que hay en una onda periódica. La unidad de frecuencia es el hertzio.
- **Habilitar:** Activar o poner en modo operacional. Una entrada de un circuito lógico que activa su funcionamiento.
- **Hardware:** La circuitería y componentes físicos de un sistema de computadora (en oposición a las instrucciones que denominamos software).
- **HDL:** Hardware Description Language, lenguaje de descripción hardware. Es utilizado para describir un diseño lógico utilizando software.
- **Hexadecimal:** Describe un sistema de numeración en base 16.
- **IEEE:** Institute of Electrical and Electronic Engineers.
- **Implementación:** Proceso software en el que las estructuras software descritas por la lista de interconexiones se implementan en la estructura del dispositivo de destino.

- **Instrucción:** Unidad de información que le dice a la CPU lo que tiene que hacer.
- **Interrupción:** Señal o instrucción que hace que el proceso actual sea temporalmente detenido mientras se ejecuta una rutina de servicio.
- **Interrupción software:** Una instrucción que invoca una rutina de servicio de interrupción.
- **IP:** Puntero de instrucción. Un registro especial de la CPU que almacena el desplazamiento correspondiente a la siguiente instrucción que se va a ejecutar.
- **Lectura:** El proceso de recuperar datos de una memoria.
- **LED:** Light-Emitting Diode, diodo emisor de luz.
- **Lenguaje de alto nivel:** Un tipo de lenguaje de computadora muy próximo al lenguaje humano que se encuentra en un nivel por encima del lenguaje ensamblador.
- **Lenguaje ensamblador:** Un lenguaje de programación que utiliza palabras similares a las del idioma inglés y que tiene una correspondencia biunívoca con el lenguaje máquina.
- **Lenguaje máquina:** Instrucciones de computadora escritas en código binario que una computadora es capaz de comprender. El nivel más bajo de lenguaje de programación.
- **Longitud de palabra:** El número de bits en una palabra.
- **Matriz de memoria:** Matriz formada por las celdas de memoria colocadas formando filas y columnas.
- **Matriz de registros:** Un conjunto de posiciones de almacenamiento temporal dentro del microprocesador, para almacenar datos y direcciones a las que tiene que acceder rápidamente el programa.

- **MFLOPS:** Million Floating-Point Operations Per Second, millones de operaciones en coma flotante por segundo.
- **Procesador:** Circuito integrado digital que puede ser programado mediante una serie de instrucciones para realizar funciones específicas sobre los datos.
- **Mnemónico:** Instrucción similar al idioma inglés que es convertida por un programa ensamblador en código máquina para que lo use un procesador.
- **Multiplexor (MUX):** Circuito (dispositivo digital) que conmuta los datos digitales de distintas líneas de entrada a una única línea de salida según una secuencia temporal especificada.
- **NOP:** Instrucción en lenguaje ensamblador donde RD=0.
- **Not:** Operación lógica básica que realiza inversiones.
- **Palabra:** Unidad completa de datos binarios
- **Paralelo:** Son datos que se producen simultáneamente a través de varias líneas.
- **Paridad:** Número par o impar de unos en un grupo en un código.
- **Paridad impar:** Condición de poseer un número impar de 1s en cada grupo de bits.
- **Paridad par:** Condición por la que se tiene un número par de 1s en cada grupo de bits.
- **Periférico:** Dispositivo o instrumento que proporciona servicios de comunicación con alguna computadora, o proporciona servicios o funciones auxiliares a una computadora.
- **Período (T):** Tiempo requerido por las señales periódicas para repetirse.

- **Programa:** Listado de instrucciones informáticas organizadas para lograr un resultado específico. Software.
- **Programa fuente:** Un programa escrito en lenguaje ensamblador o en un lenguaje de alto nivel.
- **RAM:** Random-Access Memory, memoria de acceso aleatorio. Memorias semiconductoras volátiles de lectura/escritura.
- **Registro:** Circuito digital capaz de almacenar y desplazar información binaria; típicamente utilizado como dispositivo de almacenamiento temporal.
- **ROM:** Read-Only Memory, memoria semiconductor no volátil de acceso aleatorio de solo lectura.
- **Salida:** Señal o línea que sale de un circuito.
- **Señal:** Conjunto de bits generados por la unidad de control.
- **Síncrono:** Que tiene una relación temporal fija. Que ocurre de forma simultánea.
- **Software:** Programa informático. Programas que dicen a la computadora qué es lo que tiene que hacer para poder realizar un determinado número de tareas.
- **Variable:** Símbolo utilizado para representar una magnitud lógica que puede tener tanto un valor 1 como 0; generalmente se designa mediante una letra cursiva.
- **Volátil:** La característica de un dispositivo lógico programable que describe que se pierden los datos almacenados cuando se elimina la alimentación.

Bibliografía

- [1] John Adam Presper y col. «See Buchholz; A survey of digital computer memory systems». En: *In Proceedings of the I. R. E* 41.10 (1919), págs. 1902-1971.
- [2] Luigia Carlucci Aiello. «The multifaceted impact of Ada Lovelace in the digital age». En: *Artificial Intelligence* 235 (2016), págs. 58-62.
- [3] Gabriel Andrade. «Internet Encyclopedia of Philosophy». En: *Enl@ce: Revista Venezolana de Información, Tecnología y Conocimiento* 8.1 (2011), págs. 106-118.
- [4] Herbert Apaza. «Yupana, Material Manipulativo para la Educación Matemática. Justicia social y el cambio educativo en niños de las comunidades quechua alto andino del Perú». En: *Universidad Complutense de Madrid* (jul. de 2017), pág. 352. URL: <https://docplayer.es/77540368-Tesis-doctoral-tesis-doctoral.html>.
- [5] Gerard Candón Arenas. «Implementación de un procesador RISC-V en una FPGA». Tesis doct. 2020. URL: <https://upcommons.upc.edu/bitstream/handle/2117/329221/151993.pdf?isAllowed=y&sequence=1>.
- [6] Marcia Ascher y Robert Ascher. «Quipu». En: *Encyclopaedia of the History of Science, Technology, and Medicine in Non-Western Cultures*. Springer, Dordrecht, mar. de 2008, págs. 1863-1865. DOI: [10.1007/978-1-4020-4425-0_9192](https://doi.org/10.1007/978-1-4020-4425-0_9192). URL: https://link.springer.com/10.1007/978-1-4020-4425-0_9192.

springer.com/referenceworkentry/10.1007/978-1-4020-4425-0_9192.

- [7] Atanasoff-Berry computer | Encyclopedia of Computer Science. URL: <https://dl.acm.org/doi/abs/10.5555/1074100.1074145> (visitado 03-10-2022).
- [8] Charles Babbage. «Chapter viii-of the analytical engine». En: *Passages from The Life of a Philosopher. Longman, Roberts, and Green, London* (1864), págs. 112-141.
- [9] Kerstin Beer y col. «Training deep quantum neural networks». En: *Nature Communications* 11.1 (2020). ISSN: 20411723. DOI: 10.1038/s41467-020-14454-2. URL: <https://doi.org/10.1038/s41467-020-14454-2>.
- [10] Simon Blackburn. *The Oxford dictionary of philosophy*. OUP Oxford, 2005.
- [11] Capítulo 4. *Historia de la Informática. Introducción a la Informática (GAP)*. Profesor Rafael Menéndez-Barzanallana Asensio. Universidad de Murcia (España). URL: <https://www.um.es/docencia/barzana/II/Ii04.html>.
- [12] Frank Carter. *The turing bombe*. Bletchley Park Trust, 2008.
- [13] George C Chase. «History of mechanical computing machinery». En: *IEEE Annals of the History of Computing* 2.03 (1980), págs. 198-226.
- [14] *Computing History*. URL: <https://jva.cs.iastate.edu/history.php>.
- [15] Allen WM Coombs. «The making of Colossus». En: *Annals of the History of Computing* 5.3 (1983), págs. 253-259.
- [16] B. Jack Copeland. «Colossus: Its origins and originators». En: *IEEE Annals of the History of Computing* 26.4 (oct. de 2004), págs. 38-45. ISSN: 10586180. DOI: 10.1109/MAHC.2004.26.

- [17] Brian Jack Copeland. *Alan Turing: El pionero de la era de la información* - Brian Jack Copeland - Google Libros. 2013. URL: https://books.google.com.co/books?hl=es&lr=&id=CSpZAgAAQBAJ&oi=fnd&pg=PA13&dq=alan+turing&ots=9RtjHrz3X3&sig=3QAaUHsxZpPcAgmuhIkPvqmR-Lo#v=onepage&q=alan%20turing&f=false%20https://books.google.com.pa/books?id=CSpZAgAAQBAJ&pg=PA253&dq=Prueba+de+Turing&hl=es&sa=X&redir_esc=y#v=onepage&q=Prueba%20de%20Turing&f=false (visitado 03-10-2022).
- [18] J. P. Eckert. «A Survey of Digital Computer Memory Systems». En: *Proceedings of the IRE* 41.10 (1953), págs. 1393-1406. ISSN: 00968390. DOI: [10.1109/JRPROC.1953.274316](https://doi.org/10.1109/JRPROC.1953.274316).
- [19] Jr John Presper Eckert y John W Mauchly. *Electronic numerical integrator and computer*. US Patent 3,120,606. 1964.
- [20] Peter Eckstein. «J. Presper Eckert». En: *IEEE Annals of the History of Computing* 18.1 (1996), págs. 25-44. ISSN: 10586180. DOI: [10.1109/85.476559](https://doi.org/10.1109/85.476559).
- [21] Linda Eikmeier Endersby. «ENIAC: The Triumphs and Tragedies of the World's First Computer (review)». En: *Technology and Culture* 41.4 (2000), págs. 825-826. DOI: [10.1353/tech.2000.0151](https://doi.org/10.1353/tech.2000.0151). URL: <https://books.google.com/books/about/ENIAC.html?hl=es&id=GxMIAQAAQAAJ>.
- [22] Daniel T Fitzpatrick y col. «A RISCy approach to VLSI». En: *ACM SIGARCH Computer Architecture News* 10.1 (1982), págs. 28-32.
- [23] W Barkley Fritz. «The women of ENIAC». En: *IEEE Annals of the History of Computing* 18.3 (1996), págs. 13-28.
- [24] Yi Gao, Shilang Tang y Zhangli Ding. *Comparison between CISC and RISC*. Inf. téc. 2000.

- [25] Angelo Garofalo y col. «Pulp-NN: Accelerating quantized neural networks on parallel ultra-low-power RISC-V processors». En: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 378.2164 (2020). ISSN: 1364503X. DOI: [10.1098/rsta.2019.0155](https://doi.org/10.1098/rsta.2019.0155). arXiv: [1908.11263](https://arxiv.org/abs/1908.11263).
- [26] Adrian Gaudebert. «Machine de Turing». En: (2009), págs. 1-5.
- [27] William Burns Glynn. *Decodificación de quipus*. Banco Central de Reserva del Perú, 2002.
- [28] *Gottfried Leibniz (1646 - 1716) - Biography - MacTutor History of Mathematics*. URL: <https://mathshistory.st-andrews.ac.uk/Biographies/Leibniz/> (visitado 03-10-2022).
- [29] John Gustafson. «Reconstruction of the Atanasoff-Berry computer Unum Arithmetic View project Performance Measurement View project». En: (2000). URL: <https://www.researchgate.net/publication/262402944>.
- [30] John L. Hennessy y David A. Patterson. *Computer Organization: A Quantitative Approach*. 6th ed. London, England: Morgan Kaufmann, 2017.
- [31] John L. Hennessy y col. «MIPS: A microprocessor architecture». En: *ACM SIGMICRO Newsletter* 13.4 (1982), págs. 17-22.
- [32] *Homepage | IEC*. URL: <https://iec.ch/homepage>.
- [33] *ISO - International Organization for Standardization*. URL: <https://www.iso.org/home.html>.
- [34] Ed Leibowitz. «The colossus». En: *Architect* 102.6 (ene. de 2013), págs. 172-179. ISSN: 19357001. DOI: [10.1016/b978-0-12-491650-0.50013-7](https://doi.org/10.1016/b978-0-12-491650-0.50013-7).

- [35] Shu Tien Li. «Origin and Development of the Chinese Abacus». En: *Journal of the ACM (JACM)* 6.1 (ene. de 1959), págs. 102-110. ISSN: 1557735X. DOI: 10.1145/320954.320962. URL: <https://dl.acm.org/doi/10.1145/320954.320962>.
- [36] Barbara Liskov, John Guttag y col. *Abstraction and specification in program development*. Vol. 180. MIT press Cambridge, 1986.
- [37] Marcia Sahaya Louis y col. «Towards Deep Learning using TensorFlow Lite on RISC-V». En: (2019). DOI: 10.1145/1122445.1122456. URL: <https://doi.org/10.1145/1122445.1122456>.
- [38] Luigi Federico Menabrea y Ada King Countess of Lovelace. *Sketch of the Analytical Engine Invented by Charles Babbage, Esq.* Richard y John E. Taylor, 1843.
- [39] J. B. Millendorf y M. Kalisman. «The history of computing.» En: *Clinics in plastic surgery* 13.3 (1986), págs. 351-354. ISSN: 00941298. DOI: 10.1016/s0094-1298(20)31561-3.
- [40] Gordon E Moore y col. *Cramming more components onto integrated circuits*. 1965.
- [41] Gordon E Moore. «Excerpts from a conversation with Gordon Moore: Moore Law, 2005». En: () .
- [42] David Nofre. *Turing's Revolution: The Impact of His Ideas about Computability*. 2017.
- [43] Gerard O'Regan. «ABC Computer». En: *The Innovation in Computing Companion*. Springer, Cham, 2018, págs. 7-9. DOI: 10.1007/978-3-030-02619-6_2. URL: https://link.springer.com/chapter/10.1007/978-3-030-02619-6_2.
- [44] Rafael López del Paso. «El origen de las calculadoras actuales: la Pascalina». En: *eXtoikos* 17 (2015), págs. 59-59.

- [45] David A Patterson y John L Hennessy. *Computer organization and design ARM edition: the hardware software interface*. Morgan kaufmann, 2016.
- [46] David A. Patterson y John L. Hennessy. *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. 2nd ed. London, England: Morgan Kaufmann, 2020.
- [47] J.L. Patterson, D.A. Hennessy. *Computer Organization and Design RISC-V Edition: The Hardware Software Interface (ISSN) (English Edition)*. 2017, pág. 1049. ISBN: 978-0-12-812275-4.
- [48] Carlos Radicati di Primeglio. «El sistema contable de los Incas: yupana y quipu». En: *Lima, Perú: Editorial Universo* (1979).
- [49] B. Randell. «Colossus: Godfather of the Computer». En: *The Origins of Digital Computers*. Springer, Berlin, Heidelberg, 1982, págs. 349-354. DOI: [10.1007/978-3-642-61812-3_27](https://doi.org/10.1007/978-3-642-61812-3_27). URL: https://link.springer.com/chapter/10.1007/978-3-642-61812-3_27.
- [50] RISC-V Foundation. *RISC-V International*. 2021. URL: <https://riscv.org/> (visitado 03-10-2022).
- [51] Raúl Rojas. «Konrad Zuse's legacy: the architecture of the Z1 and Z3». En: *IEEE Annals of the History of Computing* 19.2 (1997), págs. 5-16.
- [52] Raul Rojas. «Konrad Zuse's legacy: The architecture of the Z1 and Z3». En: *IEEE Annals of the History of Computing* 19.2 (1997), págs. 5-16. ISSN: 10586180. DOI: [10.1109/85.586067](https://doi.org/10.1109/85.586067). URL: http://ed-thelen.org/comp-hist/Zuse_Z1_and_Z3.pdf.
- [53] Raúl Rojas y Ulf Hashagen. *The first computers: History and architectures*. MIT press, 2002.
- [54] Raull Rojas. «Die Architektur der Rechenmaschinen Z1 und Z3 von Konrad Zuse». En: *Informatik-Spektrum* 19.6 (1996), págs. 303-315. ISSN: 0170-6012. DOI: [10.1007/s002870050041](https://doi.org/10.1007/s002870050041). URL: <https://elibRARY.ru/item.asp?id=1032133>.

- [55] *Story, Story! on JSTOR.* URL: <https://www.jstor.org/stable/45363261> (visitado 03-10-2022).
- [56] Andrew S Tanenbaum. «Implications of structured programming for machine architecture». En: *Communications of the ACM* 21.3 (1978), págs. 237-246.
- [57] Alan M Turing. «“On Computable Numbers, with an Application to the Entscheidungsproblem” Proc. London Math. Soc., 2 (42)(1936), 230-265; “A correction” ibid, 43, 544-546.» En: () .
- [58] Alan M. Turing. «Computing machinery and intelligence». En: *Parsing the Turing Test: Philosophical and Methodological Issues in the Quest for the Thinking Computer* (2009), págs. 23-65. DOI: [10.1007/978-1-4020-6710-5_3](https://doi.org/10.1007/978-1-4020-6710-5_3) / COVER. URL: https://link.springer.com/chapter/10.1007/978-1-4020-6710-5_3.
- [59] Andrew Waterman. *RISC-V Foundation*. 2015. URL: <http://www.riscv.org> (visitado 03-05-2021).
- [60] Andrew Waterman y Krste Asanovic. *The RISC-V Instruction Set Manual Volume I: Unprivileged ISA*. London, England: RISC-V Foundation, 2019.
- [61] Horst Zuse. «The life and work of Konrad Zuse». En: *webpage*, <http://www.epemag.com/zuse/part4a.htm> 3 (1966).
- [62] Konrad Zuse. *Der Computer—Mein Lebenswerk*. 1986. ISBN: 978-3-662-06515-0. DOI: [10.1007/978-3-662-06516-7](https://doi.org/10.1007/978-3-662-06516-7). URL: https://books.google.com/books/about/Der_Computer_Mein_Lebenswerk.html?hl=es&id=Gp0BBwAAQBAJ.
- [63] Konrad Zuse. *Rechnender Raum*. Vieweg+Teubner Verlag, 1969. DOI: [10.1007/978-3-663-02723-2](https://doi.org/10.1007/978-3-663-02723-2).

La Editorial de la Universidad
Tecnológica de Pereira tiene como
política la divulgación del saber
científico, técnico y humanístico para
fomentar la cultura escrita a través
de libros y revistas científicas
especializadas.

Las colecciones de este proyecto son:
Trabajos de Investigación, Ensayos,
Textos Académicos y Tesis Laureadas.

Este libro pertenece a la Colección
Textos Académicos.

La formación en ingeniería de computadoras cumple una función fundamental en la educación en la ciencia de la computación. Esta capacitación proporciona a los estudiantes los conocimientos que necesitan para diseñar, configurar y maximizar las capacidades de los sistemas computacionales. Históricamente, esta área ha sido enseñada con arquitecturas de conjuntos de instrucciones de tipo reducido (RISC) dada la dificultad que conlleva el entendimiento de las arquitecturas de tipo complejo (CISC), tal como su nombre lo indica.

En las últimas tres décadas, esta área ha pasado por varias arquitecturas RISC, tales como DLX, MIPS, SPARC y ARM. Sin embargo, ninguna ha permitido una integración vertical totalmente abierta, de libre uso y sin restricciones por licenciamiento. Esta arquitectura RISC-V pretende brindar un estándar moderno que fácilmente pueda cumplir las necesidades pedagógicas en el aula de clase. De igual forma, RISC-V logra brindar la robustez necesaria para cumplir con los requerimientos para su implementación en procesadores para teléfonos inteligentes, tabletas y sistemas embebidos, en donde actualmente domina la arquitectura ARM; y en computadoras portátiles y de escritorio, servidores y supercomputadoras, en donde domina la arquitectura X86-64. Podría proyectarse que en los próximos años el impacto de RISC-V en la arquitectura de computadoras será equivalente al rol que ha jugado Linux en los sistemas operativos.