GRADO EN INGENIERÍA INFORMÁTICA

Módulo de Formación Obligatoria

ESTRUCTURA DE COMPUTADORES

Dr. D. Jack Fernando Bravo Torres

viu Universidad Internacional de Valencia

Edita
Universidad Internacional de Valencia

Grado en

Ingeniería Informática

Estructura de computadores

Módulo de Formación Obligatoria

6ECTS

Dr. D. Jack Fernando Bravo Torres

Índice

Leyenda

abc **Glosario**
Términos cuya definición correspondiente está en el apartado "Glosario".

Tema 1.
Introducción a la estructura del computador

Durante los últimos 30 años hemos presenciado una evolución constante y dinámica en el mundo de las **computadoras**. Esto ha producido una profunda transformación social—una tercera revolución: la de la información y comunicación—, que ha cambiado la forma en la cual las personas percibimos e interactuamos con nuestro entorno. La digitalización de la información, la ubicuidad de los sistemas de computación y comunicación junto con el desarrollo de Internet—en especial de las redes sociales en línea— han hecho posible el despliegue de aplicaciones y dispositivos autónomos e inteligentes nunca antes vistos y solo pensados en la ciencia ficción (Patterson & Hennessy, 2017; Stallings, 2013).

Comprender los conceptos relacionados a la organización y los procesos subyacentes al funcionamiento de las computadoras y cómo los **programas** escritos en un **lenguaje de alto nivel** son ejecutados por el hardware, proporcionan a los ingenieros en computación las herramientas necesarias para mejorar la eficiencia de sus programas y evaluar las características que hacen mejor a una computadora con respecto a otra, frente a una aplicación particular. En ese sentido, **el rendimiento de un programa estará dado en función de la combinación de tres elementos (Patterson & Hennessy, 2017): (1) la eficiencia de los algoritmos usados en el programa; (2) los sistemas de software usados para crear y trasladar el programa a instrucciones de máquina, entendibles por el hardware; y (3) la eficiencia del computador al ejecutar esas instrucciones**. Durante el trascurso de esta asignatura,

nos enfocaremos a los dos últimos elementos. El primero, referente a los algoritmos, queda fuera de nuestro objeto de estudio.

En el proceso de descripción de las computadoras, a menudo se hace una diferenciación entre la arquitectura del computador y su estructura. En el primer caso, nos referimos a aquellos atributos del sistema que tienen un impacto directo en la ejecución lógica del programa: repertorio de instrucciones, tipos y formatos de datos, número y ancho de los registros, técnicas de entrada/salida, técnicas de direccionamiento de la memoria. En el segundo caso, hablamos de los elementos operacionales y sus interconexiones, que dan cuenta de las especificaciones de la arquitectura. Por lo tanto, describe un conjunto de elementos y sus interrelaciones que son transparentes al programador: los componentes del computador y su organización (Hamacher, 2012; Stallings, 2013). Esta separación de la estructura y arquitectura del computador ha cumplido un rol importante en el desarrollo de familias de computadoras, las cuales han mantenido la misma arquitectura, pero con diferencias en su organización. Esto permite que las computadoras usando, por ejemplo, un mismo repertorio de instrucciones, tengan diferente coste y rendimiento. Un ejemplo de procesadores con una misma arquitectura y diferente estructura son AMD Opteron e Intel Core i7. Ambos ocupan el mismo registro de instrucciones, pero difieren en la organización.

En lo que sigue de este primer tema de estudio, daremos una visión general de la estructura del computador, su definición, sus elementos constitutivos y la evolución que ha tenido a lo largo de la historia. Esto nos brindará el conocimiento de base necesario para abordar y profundizar en el desarrollo de los temas siguientes.

1.1. El computador

El computador puede ser definido como una máquina electrónica que realiza tareas de procesamiento, almacenamiento y movimiento de datos junto con el control de los procesos requeridos. Para ello, ejecuta una secuencia de instrucciones, conocidas como programas, que le permiten desarrollar la tarea asignada (Tanenbaum, 2016).

Desde la perspectiva de los sistemas, se puede ver al computador como un sistema complejo y jerárquico, conformado por un conjunto de subsistemas interrelacionados, los cuales, a su vez, se organizan en una estructura jerárquica, en forma recurrente, hasta alcanzar la estructura más elemental (Stallings, 2013). Este enfoque nos permite mantener un nivel de abstracción que oculta las complejidades de los diferentes subsistemas (niveles) a los de orden superior. De esta forma, un nivel o capa de la estructura jerarquía del computador, solo requiere conocer una caracterización simplificada abstracta de nivel inmediato inferior.

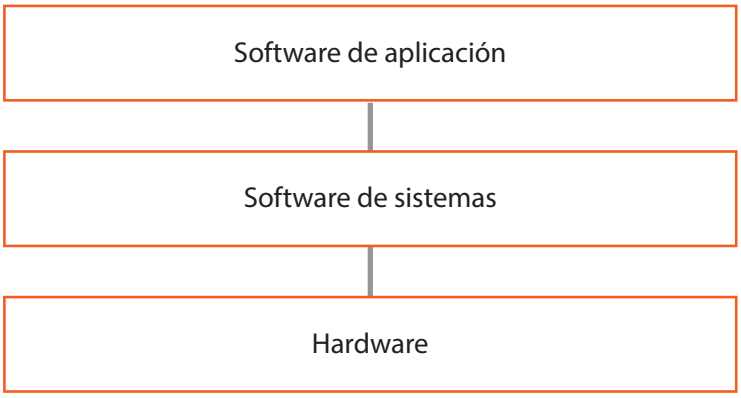


Figura 1. Visión simplificada del hardware y software como capas o niveles jerárquicos. Fuente: Adaptado desde (Patterson & Hennessy, 2017).

En la figura 1 se muestra una visión del hardware y software como capas jerárquicas. Un conjunto de software de sistemas se encuentra localizado entre el software de las aplicaciones y el hardware. La funcionalidad básica de este nivel es proveer diversos servicios que son comúnmente útiles. Dentro de esta capa podemos destacar el **sistema operativo** y el **compilador**. Por una parte, el sistema operativo gestiona y supervisa los recursos del computador para beneficio de los programas que están ejecutándose. Entre sus funciones tenemos (Patterson & Hennessy, 2017): manejo de las operaciones básicas de entrada y salida, asignación de almacenaje y memoria, provisión de una compartición protegida del computador entre las múltiples aplicaciones que acceden en forma simultánea. Ejemplos de sistemas operativos son Linux, iOS y Windows. Por otra parte, el compilador permite trasladar un programa escrito en un lenguaje de alto nivel (C++, Java, Python,…) a instrucciones que pueden ser ejecutadas por el hardware.

Recordemos que el **alfabeto del lenguaje que ejecuta el hardware está compuesto por dos símbolos: unos (1) o ceros (0), los cuales son denominados como bits**. Las órdenes que le damos al computador son conocidas como instrucciones, las cuales están conformadas por un conjunto de bits que el computador entiende y obedece. Por ejemplo, la instrucción:

1001010100101110

le indica al computador que realice la suma de dos números. Este lenguaje es conocido como **lenguaje de máquina**, el cual es la representación binaria de las instrucciones de máquina. Sin embargo, desarrollar programas usando este lenguaje es complejo y limita el desempeño de los programadores. Por ello, se desarrolló una notación simbólica, conocida como **lenguaje ensamblador**, que permitió usar símbolos, más cercanos al pensamiento humano, para posteriormente, a través de un programa denominando **ensamblador**, trasladar esta versión simbólica a la binaria. De esta forma, la instrucción de máquina del ejemplo anterior es representada en lenguaje ensamblador como:

add A, B

No obstante, la mejora en el proceso de abstracción desde el lenguaje de máquina al de ensamblador, todavía exige al programador desarrollar una línea de código por cada instrucción que el computador ejecutará, obligándolo a pensar como este último. Los lenguajes de programación de alto nivel

solventan este inconveniente. Los compiladores trasladan los programas escritos en esos lenguajes a instrucciones. La Figura 2 muestra una representación del proceso de traslado desde un lenguaje de alto nivel a las instrucciones de máquina.

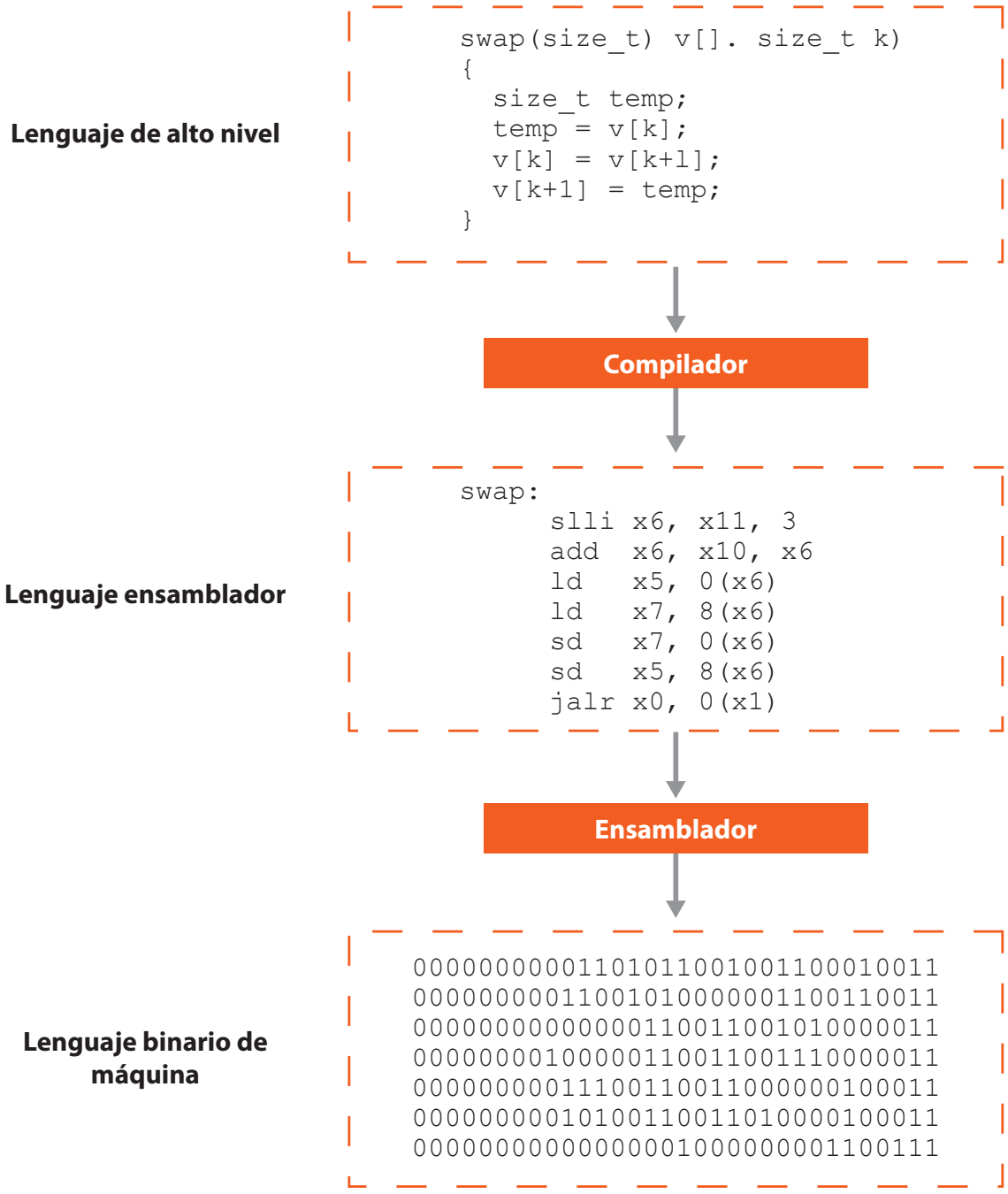


Figura 2. Traslado desde un lenguaje de alto nivel a instrucciones de máquina. Fuente: Adaptado desde (Patterson & Hennessy, 2017).

1.2. Estructura

En esta sección describiremos brevemente los elementos básicos en la estructura de un computador, los cuales permiten ejecutar sus funciones básicas; así como los tipos de estructuras más utilizadas en la industria.

1.2.1. Unidades funcionales

Como hemos mencionado en las secciones precedentes, las cuatro funciones básicas que realiza un computador son:

- Almacenamiento de datos
- Procesamiento de datos
- Transferencia de datos
- Control

Estas funciones son ejecutadas por varias unidades o elementos funcionales. Así, los datos recibidos son almacenados en la **memoria del computador**—como veremos posteriormente, existen varios tipos de memorias, las cuales permiten al computador almacenar los datos conforme a sus necesidades—. Esta información recibida es procesada por la **unidad lógica y aritmética** (ALU, del inglés <<*arithmetic and logic unit*>>), de acuerdo a un conjunto de instrucciones especificadas en un programa, el cual también es almacenado en la memoria. Todas estas acciones, son coordinadas por la **unidad de control** y en conjunto con la ALU son denominadas como **procesador**—también conocido como unidad central de procesamiento (CPU, del inglés <<*central processing unit*>>)—. La transferencia de datos e información entre las unidades del computador (procesador, unidades de entrada/salida (E/S), memoria) es desarrollada a través de una red de interconexión, la cual provee los medios necesarios para el intercambio de información y la coordinación de las acciones requeridas. La **unidad de entrada/salida** (E/S), está conformada por un conjunto de dispositivos periféricos como teclado, mouse, cámaras, micrófonos, LCDs, entre otros, que permiten la transferencia de datos entre el computador y el entorno externo. La Figura 3 muestra un diagrama de las unidades básicas funcionales de una computadora.

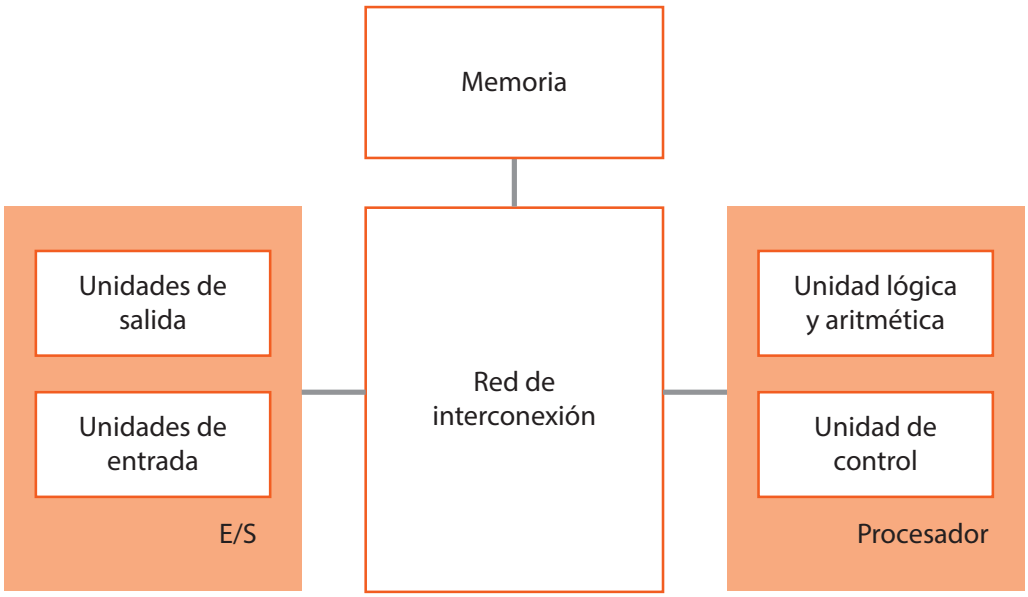


Figura 3. Unidades funcionales básicas de un computador. Fuente: Adaptado desde (Hamacher, 2012).

1.2.2. Tipos de estructuras

Dos son las estructuras[1] que han dominado la organización del computador: **Von Neumann** y **Harvard**. **Su principal diferencia se encuentra en el mapa de memoria. En la estructura Von Neumann los datos e instrucciones comparten un único espacio de memoria; mientras que en la Harvard, hay dos espacios de memoria separados: uno para los datos y otro para las instrucciones**.

- **Estructura Harvard**

  Los elementos constitutivos de esta estructura son una unidad central de procesamiento (CPU), dos espacios de memoria que almacenan por separado las instrucciones y los datos, tanto de entrada como de salida, y los elementos de E/S (ver la Figura 4).

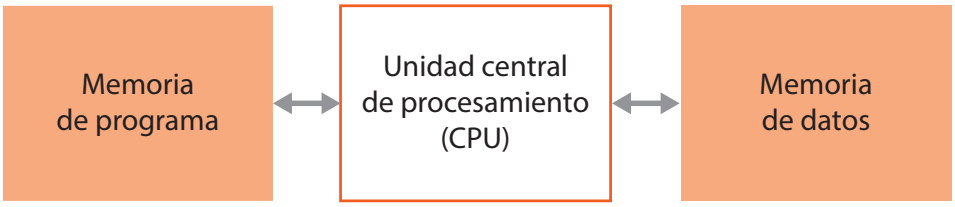


Figura 4. Estructura Hardvard. Fuente: Elaboración propia .

Esta forma de disponer los espacios de memoria, permiten al procesador ganar independencia y simultaneidad en el acceso a los datos e instrucciones. Esto significa que puede ejecutar las instrucciones en un menor tiempo. Además, permite un mapa de direcciones de instrucciones

[1] Aunque en la literatura es común su denominación como arquitecturas; sin embargo, en realidad hacen referencia a la estructura del computador, más que a su arquitectura.

y datos separados; así como, diferentes características para cada memoria: tamaño de las palabras de memoria, tamaño de cada memoria y la posibilidad de usar diferente tecnología.

Esta estructura es ampliamente usada en microcontroladores y en procesadores digitales de señales (DSP, del inglés <<*digital signal processor*>>). Estos tipos de computadores son usados para propósitos específicos.

- **Estructura Von Neumann**

A diferencia de la estructura Harvard, la Von Neumann hace uso de un solo espacio de memoria para el almacenamiento de instrucciones y datos. El proceso de acceso se basa en un solo **bus** que se usa tanto para la transferencia de datos como para la obtención de instrucciones. La Figura 5 muestra un esquema de esta estructura.

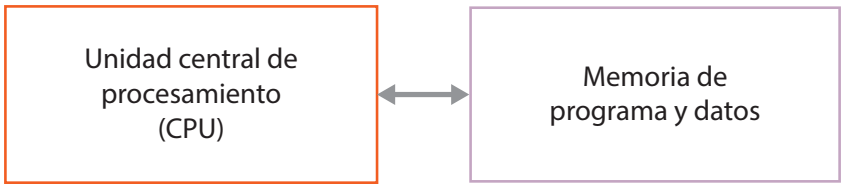


Figura 5. Estructura Von Neumann. Fuente: Elaboración propia .

Una de sus características principales es que la ejecución de las instrucciones se desarrolla de manera secuencial. Esto es, una vez que se lleva a efecto una instrucción se continúa con la siguiente, la cual está almacenada en la memoria principal. No obstante, existen clases de instrucción que permiten romper esta secuencialidad. Lo que se busca con este tipo de estructura es que pueda ser usada en forma general para solventar diferentes problemas, para lo cual únicamente se necesita usar programas que se adapten a sus requerimientos.

1.3. Clases de computadoras

Conforme a sus aplicaciones y capacidades, las computadoras pueden ser clasificadas en: **embebidas**, **personales**, **servidores**, **supercomputadoras** y **computadoras en red** (del inglés <<*grid computers*>>).

1.3.1 Computadoras embebidas

Este tipo de computadoras forman parte de dispositivos de mayor tamaño o diversos sistemas, con el propósito de apoyar, en forma automática, el control o monitoreo de procesos físicos o ambientales. Son diseñados para propósitos específicos más que para tareas generales. Entre las aplicaciones más destacada tenemos la automatización industrial y de los hogares, productos de telecomunicaciones y vehículos. Dado que estos computadores son diseñados para ejecutarse sobre una aplicación o un conjunto de aplicaciones que generalmente están integrados dentro de un hardware específico, pero mostrado como un único sistema, muchos usuarios no llegan a ser conscientes de que ellos están usando computadoras, ni el rol que cumplen en esos dispositivos (Hamacher, 2012; Patterson & Hennessy, 2017).

1.3.2. Computadoras personales

Esta clase de computadoras seguramente son las más conocidas por su extenso uso social. Ellas están dedicadas al desarrollo de tareas como preparación de documentación, diseño soportado por computador, entretenimiento multimedia, comunicación interpersonal y navegación por Internet. Estas computadoras enfatizan la entrega de un buen rendimiento a usuarios individuales.

1.3.3. Servidores

Son computadoras grandes destinadas a ser compartidas por un amplio conjunto de usuarios, quienes acceden usualmente a través de una red. Pueden alojar grandes bases de datos y proveer mayores capacidades de procesamiento para la ejecución de aplicaciones complejas.

1.3.4. Supercomputadoras y computadoras en red

Generalmente, estas computadoras son las que proporcionan el más alto rendimiento. Las supercomputadoras son usadas en el desarrollo de aplicaciones altamente demandantes, como por ejemplo para la ejecución de procesos de simulaciones que requieren grandes volúmenes de cálculo. Presentan un coste elevado.

Por su parte, las computaras en red proveen una alternativa con una mejor relación coste-beneficio. Combinan un gran conjunto de computadoras personales y dispositivos de almacenamiento en una red de alta velocidad distribuida físicamente, los cuales son gestionados como un recurso de computación.

1.4. Evolución histórica

La evolución de las computadoras ha estado ligada al desarrollo tecnológico de la electrónica, específicamente de los dispositivos **semiconductores** y los **circuitos integrados** (CI). Uno de los puntos clave de esta evolución es el vertiginoso incremento de los recursos disponibles en los CI, que conforme a las predicciones desarrolladas por la ley de Moore, duplican sus recursos cada 18 o 24 meses. Es acorde a ese desarrollo tecnológico que se ha propuesto en la literatura una clasificación de la evolución de los computadores, organizada en diferentes generaciones (Hamacher, 2012; Patterson & Hennessy, 2017; Stallings, 2013; Tanenbaum, 2016):

1.4.1. Primera generación

En esta primera generación, el primer computador desarrollado (ENIAC, por sus siglas en inglés <<*Electronic Numerical Integrator And Computer*>>) estaba constituido por 18.000 tubos de vacío 1500 relés. Pesaba alrededor de 30 toneladas y consumía 140 kilovatios de potencia. Se basaba en un sistema decimal y no binario. Uno de sus mayores inconvenientes era su programación manual a través de conmutadores y la conexión y desconexión de cables.

Para dar solución a ese problema, Von Neumann propuso el concepto de programa almacenado, el cual buscaba que la información sea representada de una mejor manera para que pueda ser almacenada

en la memoria junto con los datos. De esta forma, el proceso dejó de ser manual y el computador solo tenía que leer sus instrucciones desde la memoria, pudiéndose hacer o modificar el programa, ubicándolo en una zona de memoria. Esta idea se mantiene hasta nuestros días.

1.4.2. Segunda generación

El surgimiento de los **transistores** dio paso a la segunda generación de computadores. La sustitución de los tubos de vacío por los transistores, los cuales desarrollaban las mismas tareas pero con un tamaño mucho más pequeño, menor consumo de potencia y con un menor coste permitieron el desarrollo de computadores de menor tamaño y más prestaciones. Dentro de esta segunda generación, también se dio paso al desarrollo de unidades lógicas y aritméticas y de unidades de control más complejas, el uso de lenguajes de programación de alto nivel, y la presencia de un software de sistema con el computador (Stallings, 2013).

1.4.3. Tercera generación

Los computadores de segunda generación contenían cientos de miles de transistores lo que hacía cada vez más complejo el hecho de construir máquinas más potentes—esto debido al incremento de consumo de energía y los retardos generados por los tiempos que se suscitaban por la separación entre los elementos. La invención de los circuitos integrados dio paso a una nueva etapa en el desarrollo de los computadores. Estos dispositivos integraban en un solo elemento, conocido como chip, una docena de transistores.

En su estructura más básica, un computador puede verse como una colección de puertas lógicas y celdas de memoria. Interconectando estos elementos básicos podemos construir un computador. Los primeros chips solo permitían tener unas pocas celdas y puertas por lo que se les denominó como de pequeña escala de integración (SSI, del inglés <<*small scale integration*>>).

El surgimiento de esta tecnología permitió la construcción de procesadores más rápidos y menos costosos. De igual forma, las memorias en circuitos integrados empezaron a reemplazar las memorias de núcleo magnético (Hamacher, 2012).

1.4.4. Cuarta generación

El rápido crecimiento del número de transistores al interior de un chip permitió que se llegue a tener millones de transistores en estos circuitos integrados. Esta capacidad de integración de transistores en un solo chip se conoció como integración a muy gran escala (VLSI, del inglés <<*very large scale integration*>>).

El desarrollo de esta tecnología permitió que procesadores completos y gran parte de la memoria principal de pequeños computadores, puedan ser implementados en un solo circuito integrado. Un procesador completo, fabricado en un solo chip es conocido como un microprocesador. Actualmente VLSI permite la integración de múltiples procesadores (núcleos) y memoria caché en un solo chip.

De igual forma, el desarrollo de una clase especial de tecnología VLSI, conocida como **FPGAs** (del inglés <<*Field Programmable Gate Arrays*>>) han permitido la generación e implementación de

procesadores, memoria y circuitos de E/S sobre un solo chip, útiles en el despliegue de sistemas de computación embebida.

1.5. Multiprocesamiento

El incremento de la densidad de la lógica y la velocidad de reloj en los chips ha generado dificultades en la disipación del calor, lo que a su vez produce serios problemas en temas de diseño. Estos límites de potencia han llevado a un cambio drástico en el diseño de los microprocesadores. Desde el 2006, todas las compañías de computadoras personales y servidores se han orientado al uso de microprocesadores con múltiples procesadores por chip. Los beneficios de esta decisión se reflejan no el tiempo de ejecución—el tiempo entre el inicio y la finalización de la tarea—, sino en el <<*throughput*>>—la cantidad total de trabajo desarrollado en un tiempo determinado. Para obtener significativos mejoramientos en el tiempo de respuesta, los programadores, hoy en día, deben reescribir sus programas para tomar ventaja de los múltiples procesadores.

En la terminología usada, el procesador hace referencia a un núcleo (del inglés <<*core*>>). Por ejemplo, un microprocesador de cuatro núcleos (<<*quadcore*>>) es un microprocesador que contiene cuatro procesadores o núcleos en un solo chip.

Tema 2.
Arquitectura del repertorio de instrucciones

El nivel de arquitectura del repertorio de instrucciones (ISA, del inglés <<*Instruction Set Architecture*>>) puede ser visto como una interface abstracta entre el hardware y el software de nivel más bajo, que permite escribir un programa en lenguaje de máquina (ver Figura 6). Esto es, nos muestra una descripción funcional de las localizaciones de almacenamiento: registros y memoria; operaciones: sumar, multiplicar, almacenar, cargar, etc.; y la forma de invocar esas funciones. Así, este nivel incluye la organización del almacenaje, los tipos de datos, el registro de instrucciones, los modos de direccionamiento de las instrucciones/datos, y el manejo de las excepciones visibles del programa.

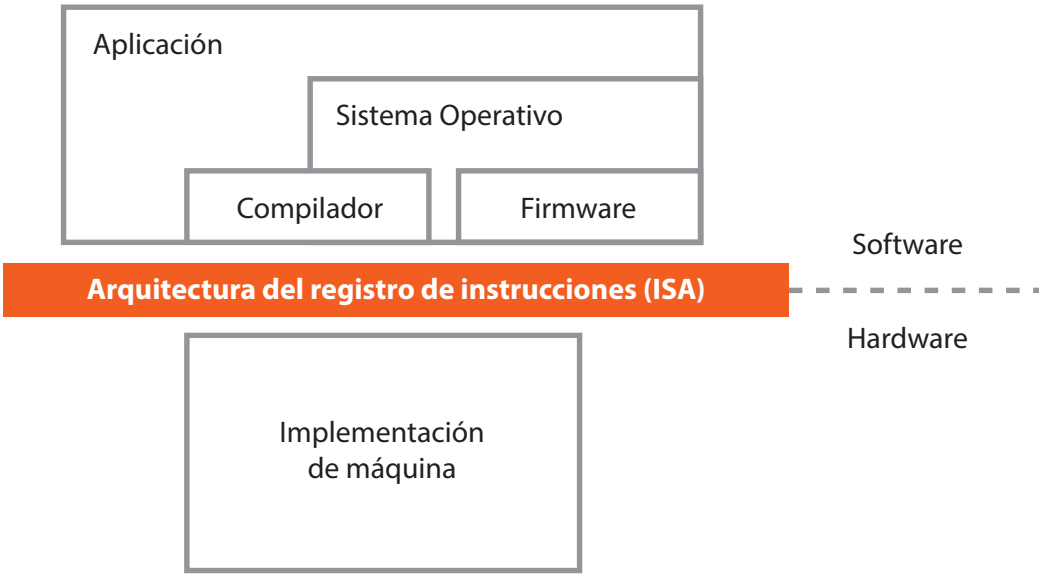


Figura 6. ISA conceptualizada como interfaz entre software y hardware. Fuente: Adaptado desde (Tanenbaum, 2016) y CS-224 Computer Organization Lecture 01 (https://www.youtube.com/watch?v=CDO28Esqmcg)

Si bien es posible desarrollar hardware que ejecute directamente programas escritos en lenguajes de alto nivel, esto no sería eficiente ni práctico. La mayoría de las computadoras requieren ejecutar programas escritos en diferentes lenguajes, por lo que resulta necesario introducir una interfaz que traduzca los programas desarrollados en lenguajes de alto nivel a una forma intermedia común—el nivel ISA— y construir hardware que soporte programas en el nivel ISA. Desde este punto de vista, este nivel es la frontera en la que tanto el diseñador como el programador observan la misma máquina.

La arquitectura del repertorio de instrucciones permite dar las especificaciones funcionales del procesador. Los diseñadores de computadoras deben generar un lenguaje que haga fácil el construir hardware y compiladores que maximicen el rendimiento y minimicen el coste y el consumo de energía (Patterson & Hennessy, 2017). Una de las características básicas de los repertorios de instrucciones—el conjunto de comandos comprendidos por una arquitectura dada— es que sus diferentes versiones o evoluciones deben ser compatibles con sus modelos anteriores, lo que hace que las nuevas máquinas puedan ejecutar programas viejos. Esto permite que, mientras los diseñadores hagan posible la compatibilidad del ISA con los modelos anteriores, tengan mayor libertad para hacer los cambios que consideren en el hardware (Tanenbaum, 2016).

2.1. Clasificación de las arquitecturas del repertorio de instrucciones

Una de las formas más comunes de clasificación se basa en el tipo de almacenamiento interno. Dado que las instrucciones requieren de un conjunto de operandos, según los criterios de su localización y explicitación: implícita o explícita, se puede identificar los siguientes tipos de arquitecturas basadas en (Hennessy, John L and Patterson, 2011): pila, acumulador y registros de propósito general.

En el caso de las arquitecturas basadas en pila, los operandos son implícitos y se encuentran ubicados en un área de memoria denominada pila y en las arquitecturas basadas en acumulador, uno de sus operandos está implícito en el **acumulador**. Mientras que, en las arquitecturas basadas en registros

de propósito general, tienen solo operandos explícitos, ubicados o en registros de propósito general o en la memoria. Dentro de esta última clase de arquitectura se pueden distinguir tres subtipos: registro-memoria, <<*load-store*>> y memoria-memoria.

En las instrucciones del subtipo *registro-memoria*, se puede acceder a la memoria con uno de sus operandos. En el caso de la arquitectura <<*load-store*>>, solo pueden acceder a la memoria instrucciones de carga y almacenamiento. Y en la *memoria-memoria*, prácticamente sin uso en la actualidad, mantiene todos los operandos en memoria. Existe una clase adicional de arquitectura del repertorio de instrucciones denominada de *acumulador extendido* o *registro de propósito especial*, en el cual existen más registros que un solo acumulador, pero tienen restricciones de uso.

2.2. Ubicaciones de memoria y direcciones

Las memorias están constituidas por millones de celdas, cada una de las cuales puede almacenar un bit de información: 0 o 1. Dado que esto representa muy poca información, generalmente se maneja grupos de bits de tamaño fijo. Por ello, la memoria es organizada de tal forma que grupos de *n* bits pueden ser almacenados o recuperados en una sola operación. Cada uno de esos grupos de *n* bits son una *palabra* de información, donde *n* es la *longitud de la palabra*. Las computadoras actuales tienen longitudes de palabra que van en el rango de 16 a 64 bits. En la Figura 7, se puede observar una representación esquemática de la memoria como una colección de *palabras*.

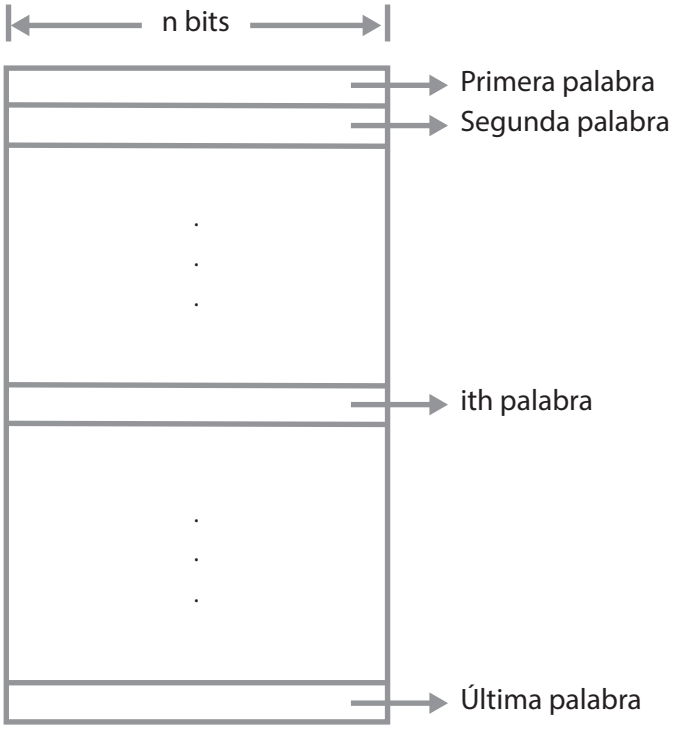


Figura 7. Visión esquemática de la memoria como colección de palabras. Fuente: Adaptado desde (Hamacher, 2012).

Para poder almacenar o recuperar información—sean palabras completas o grupos de 8 bits, denominados bytes— desde esas posiciones de memoria, es necesario contar con distintas ***direcciones*** a cada localización. El espacio de direcciones del computador hace referencia al conjunto de direcciones de 0 a 2^k, con un valor de k adecuado, que representa las direcciones sucesivas de

las ubicaciones de memoria disponibles. Por ejemplo, un direccionamiento de 24 bits genera 2^{24} (16.777.216) posiciones de memoria.

Es impráctico asignar direcciones individuales a cada bit en el espacio de memoria, por ello, se utiliza un direccionamiento a sucesivos bytes en la memoria. Las localizaciones de los bytes tienen direcciones 0, 1, 2,…En el caso de una *palabra* de 32 bits, las palabras sucesivas se encuentran ubicadas en las direcciones 0, 4,8,…dado que cada palabra está constituida por 4 bytes.

Por otra parte, la asignación del byte dentro de la palabra tiene dos formas: <<*big-edian*>>, el byte inferior de memoria direccionado es usado para los bytes más significativos de la *palabra*; mientras que con <<*little-edian*>>, el byte inferior de memoria direccionado es usado para los bytes menos significativos de la *palabra*. La Figura 8 muestra cómo los bytes son almacenados en memoria, conforme a las dos formas de asignación.

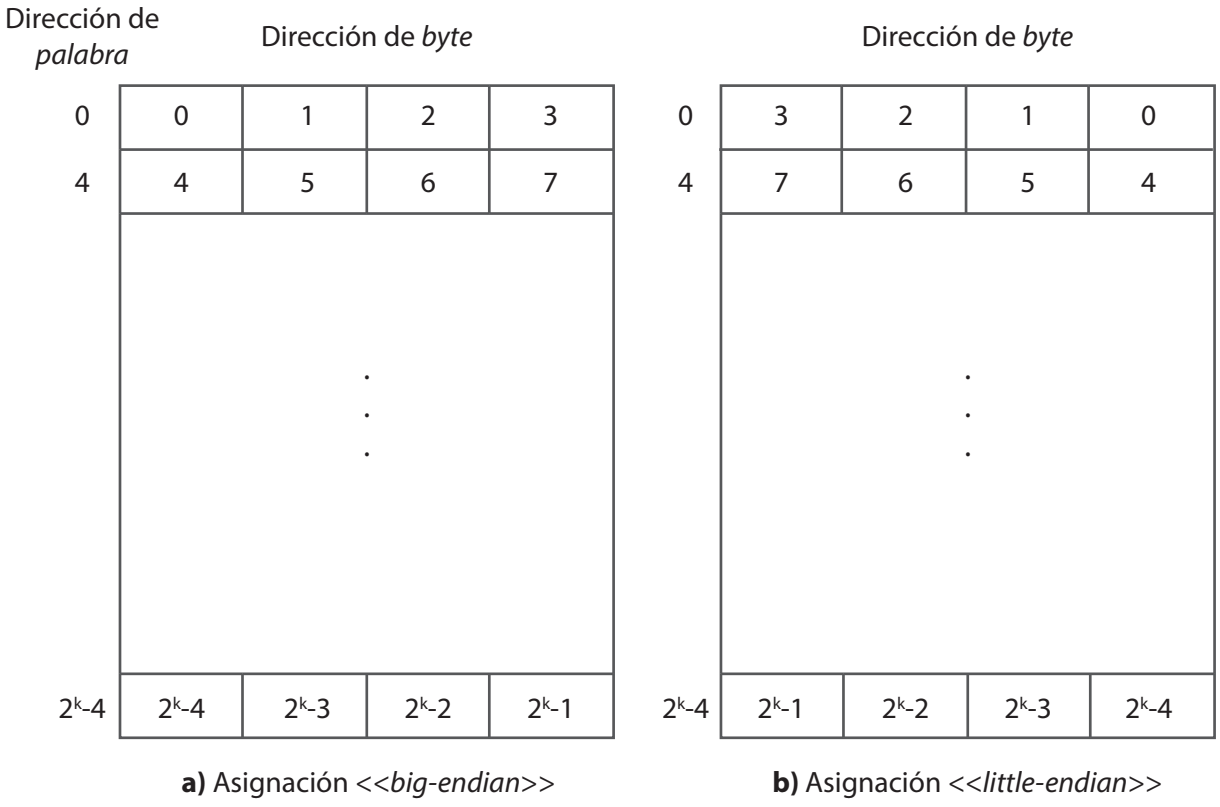


Figura 8. Formas de asignación (a) <<big-edian>> y (b) <<little-edian>>. Fuente: Adaptado desde (Hamacher, 2012).

Al interior de las memorias se almacenan tanto las instrucciones como los operandos. Para la ejecución de las instrucciones por el procesador, es necesario que pueda acceder a la palabra que contiene la instrucción, así como a los operandos. De igual forma, es necesario que los resultados puedan ser almacenados posteriormente, de ser el caso, en la memoria. En ese sentido, dos operaciones son necesarias en la memoria: *Lectura* y *Escritura*. En el primer caso, se transfiere una copia de los contenidos de una posición específica de la memoria al procesador. En el segundo caso, se envía un ítem de información desde el procesador hacia una ubicación específica de la memoria, sobrescribiendo el contenido preexistente. Posteriormente, analizaremos con más detalle los aspectos relacionados a los sistemas de memoria (Tema 4).

2.3. Instrucciones

Las instrucciones—conocidas también como ***instrucciones de máquina*** o *instrucciones del computador*— constituyen las palabras del lenguaje del computador y, por tanto, determinan el funcionamiento del procesador (Stallings, 2013). Esto es, determinan lo que el computador puede hacer. **Para desarrollar las tareas que se le requieren, el computador necesita ejecutar programas, los cuales están constituidos por subtareas (instrucciones) tales como sumar dos números, probar determinadas condiciones, leer un carácter desde memoria, entre otras. Al conjunto de instrucciones que componen el lenguaje del computador se le conoce como repertorio de instrucciones.**

En los computadores modernos básicamente hay dos diferentes aproximaciones para el diseño de los registros de instrucciones. La primera aproximación, denominada Repertorio Reducido de Instrucciones para Computadoras (RISC, del inglés <<*Reduced Instruction Set Computers*>>), restringen cada instrucción a encajar en una sola *palabra*, reduciendo la complejidad y el número de instrucciones que pueden ser incluidas en el registro de instrucciones del computador. Esto se basa en la premisa de que se puede obtener un mayor rendimiento si cada instrucción ocupa una sola palabra en la memoria y si los operandos necesarios para ejecutar instrucciones de tipo lógica o aritmética se encuentran ya en los registros del procesador. Una filosofía diferente es la aplicada en el Repertorio Complejo de Instrucciones para Computadoras (CISC, del inglés <<*Complex Instruction Set Computers*>>), el cual utiliza un repertorio más complejo de instrucciones, las cuales pueden abarcar más de una palabra de memoria, pudiendo especificar operaciones más complejas. Para propósitos de estudio, en este curso, trabajaremos con la arquitectura MIPS, la cual hace uso de instrucciones de estilo RISC.

Algunas de las características principales de los dos estilos de instrucciones son las siguientes (Hamacher, 2012):

RISC:

- Modos simples de direccionamiento.
- Todas las instrucciones encajan en una sola *palabra*.
- Pocas instrucciones en el repertorio de instrucciones, como consecuencia de los modos simples de direccionamiento.
- Operaciones lógicas y aritméticas que pueden ser desarrolladas solo sobre operandos en registros del procesador.
- Arquitectura de carga/almacenamiento que no permite transferencias directas desde una ubicación de memoria a otra. Esta operación debe ser desarrollada a través de un registro del procesador.
- Instrucciones simples que son propicias para la ejecución rápida por la unidad de procesamiento usando técnicas como segmentación.

- Programas que tienden a ser grandes en tamaño, debido al uso de un mayor número de instrucciones simples para ejecutar tareas más complejas.

CISC:

- Modos de direccionamiento más complejos.
- Instrucciones más complejas, donde una instrucción puede expandirse en múltiples *palabras.*
- Muchas instrucciones que implementan tareas complejas.
- Operaciones lógicas y aritméticas que pueden ser desarrolladas sobre operandos en memoria como también sobre operandos en registros del procesador.
- Transferencias desde una posición de memoria a otra usando una solo instrucción *Move.*
- Programas que tienden a ser pequeños en tamaño, pero se requieren instrucciones más complejas para desarrollar tareas complejas.

2.3.1. Representación de las instrucciones

Los elementos constitutivos de una instrucción de máquina (Stallings, 2013) son:

- **Código de operación *(codop)***: establece la operación que desarrolla la instrucción.
- **Referencia a operandos fuente**: las operaciones desarrolladas por las instrucciones pueden necesitar, como entrada, uno o más operandos, los cuales pueden ser explícitos o implícitos.
- **Referencias al operando de destino**: la instrucción puede arrogar un resultado que debe ser almacenado.
- **Referencia a la instrucción siguiente**: indica al procesador dónde se encuentra la siguiente instrucción a ejecutar. Esta información suele ser implícita, dado que como veremos posteriormente, el procesador continúa con la instrucción siguiente a la última ejecutada. Una excepción se da en aquellas instrucciones que especifican una ruptura de secuencia.

Por ejemplo, una instrucción que permita la suma entre los valores de dos variables y su almacenamiento en una posición de memoria determinada podría ser la mostrada en la figura 9:

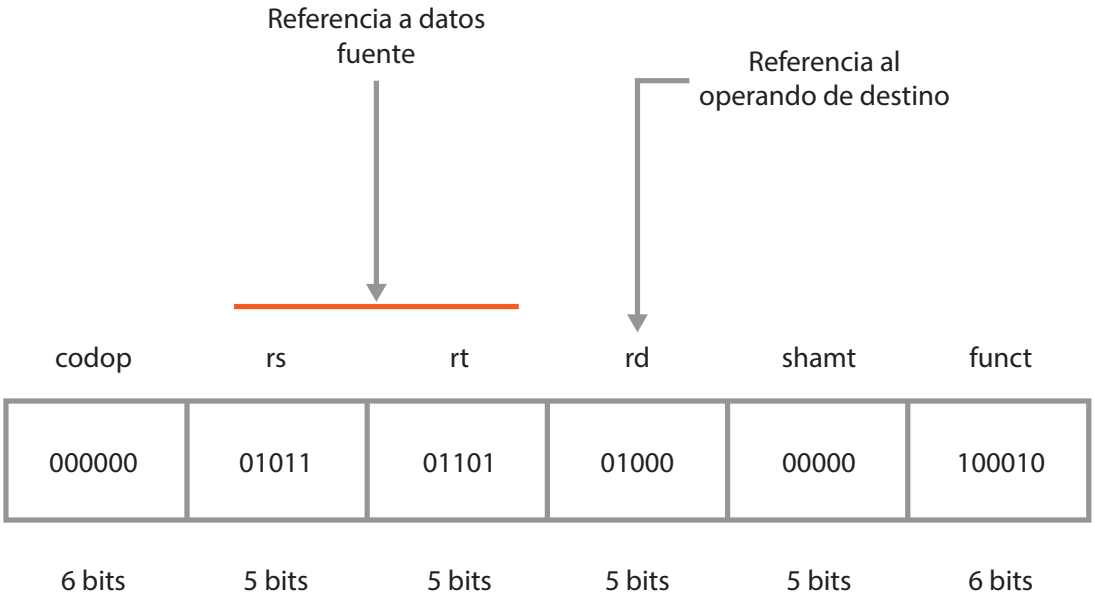


Figura 9. Ejemplo de instrucción de de máquina. Fuente: Adaptado desde (Hamacher, 2012).

Dado que las instrucciones del computador están constituidas por bits, resulta muy tedioso y complicado para el programador desarrollar sus programas en este lenguaje. Por ello, se recurre a una notación más amigable: el lenguaje ensamblador, para representar las instrucciones de máquina.

Así, los *codops* son representados a través de abreviaturas, denominadas nemotécnicos, la cuales nos indican las operaciones que se van a desarrollar. Por ejemplo, en la arquitectura MIPS, que es la que usaremos durante nuestro curso, algunos *codops* son:

add	sumar
sub	restar
lw	cargar datos de memoria
sw	almacenar datos en memoria

Por ejemplo, tenemos la instrucción en lenguaje ensamblador:

add a, b, c

El nemotécnico *add* es el *codop* e indica la operación que se va a ejecutar: una suma. Esta operación es desarrollada sobre *b* y *c*, los operandos fuente; mientras que el resultado, es escrito en *a*, el operando destino.

Algo que hay que tener en cuenta es que instrucciones como la de suma, requieren tres operandos, no más ni menos; es decir, un número fijo. Esto permite mantener el hardware lo más simple posible—un conjunto variable de operandos requiere de un hardware mucho más complejo que para un número fijo—, lo que nos lleva al primer principio del diseño de hardware: **la simplicidad favorece la regularidad** (David, A Patterson and John, 2005; Harris, David and Harris, 2010).

Como analizamos en las secciones previas, la arquitectura RISC restringe el número de instrucciones que pueden ser incluidas en el registro de instrucciones del computador. Esto se fundamenta en un segundo principio de diseño: **más pequeño es más rápido**. El número de instrucciones es mantenido pequeño, tal que el hardware usado para decodificar las instrucciones y sus operaciones, puede ser simple y rápido.

2.3.2. Operandos, registros, memoria y constantes.

Las instrucciones se desarrollan sobre operandos. Estos representan los datos que se utilizaran en la ejecución de las instrucciones. Pueden ser almacenados en diferentes localizaciones: *registros*, *memoria*, o en *constantes* auto-contenidas en la misma instrucción. Este formato de almacenamiento permite optimizar la velocidad de acceso y la capacidad de los datos (Harris, David and Harris, 2010). De esta forma, los operandos almacenados en registros o como constantes, tienen mayor rapidez en su acceso pero están limitados en su capacidad de almacenamiento de datos. Mientras que, el uso de memorias permiten una mayor capacidad, pero con un acceso más lento.

- **Registro**. Los registros pueden ser consideradas como memorias de alta velocidad localizadas en el procesador. Estos registros son limitados en su número. En el caso de MIPS se tienen 32 o 64 registros, dependiendo de la versión que se use (en este curso analizaremos la versión de 32 registros). Generalmente son usados con las instrucciones que desarrollan operaciones lógicas y matemáticas.

- **Memoria**. Otra posibilidad de almacenamiento, que extiende la capacidad de uso de variables en los programas, es el uso de memorias. Si bien, se tiene una mayor capacidad de almacenamiento, sin embargo, el acceso a esos datos es mucho más lento. Por ello, los datos usados con mayor frecuencia son generalmente mantenidos en los registros.

- **Constantes/inmediatos**. Los datos se encuentran auto-contenidos en la propia instrucción.

2.3.3. Tipos de operandos

Los operandos representan las ubicaciones en donde se encuentran los datos que desea usar en la instrucción. Se puede establecer las siguientes categorías:

- **Dirección**. Este es un tipo de dato permite delinear la ubicación de un elemento específico dentro del espacio de memoria. En varios casos se debe desarrollar procesos de cálculo adicionales sobre la referencia a un operando de una instrucción para determinar la dirección de memoria.

- **Número**. Este tipo de datos se usan para expresar los valores numéricos. Debido a que estos valores son almacenados en espacios de memoria, debemos ser conscientes de que hay limitaciones en magnitud y, en el caso de los números de coma flotante, su precisión.

En el lenguaje ensamblador, se puede representar los valores numéricos en formas diferentes: decimal, hexadecimal, binario. No obstante, en el hardware, los números son representados siempre como señales electrónicas de nivel alto o bajo, las cuales son consideradas como números de base 2.

Si consideramos una *palabra* de 32 bits de una arquitectura MIPS, el dato 1001_2 es almacenado de la siguiente forma.

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	1 0 0 1

(32 bits)

Podemos observar que los bits están numerados de izquierda a derecha, donde el bit 0, ubicado más a la derecha, es denominado como el *bit menos significativo*; mientras que el bit 31, ubicado en el extremo izquierdo, es conocido como el *bit más significativo*. En el caso de un registro de 32 bits, como en el ejemplo anterior, tenemos la posibilidad de representar 2^{32} diferentes patrones, esto es desde 0 a 2^{32}-1 (4,294,967,295). Estos números positivos son conocidos como números sin signo.

El hardware es diseñado para desarrollar diferentes operaciones aritméticas sobre esos patrones binarios. Si el resultado no puede ser representado por ese hardware, puesto que excede los valores establecidos, se dice que existe un desbordamiento (del inglés <<*overflow*>>). Cómo se actúa frente a un desbordamiento, depende del lenguaje de programación, el sistema operativo y el programa (David, A Patterson and John, 2005).

Para la representación de números positivos y negativos, generalmente se hace uso de una convención denominada *complemento a 2*: los primeros 0s significan positivo y los primeros 1s significan negativo.

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000 = 0_{10}$

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001 = 1_{10}$

… …

$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111 = 2{,}147{,}483{,}647_{10}$

$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000 = -2{,}147{,}483{,}648_{10}$

$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001 = -2{,}147{,}483{,}647_{10}$

… …

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110 = -2_{10}$

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111 = -1_{10}$

Las computadoras hacen uso de complemento a 2, tanto para representaciones binarias de números con signo y sin signo.

Como podemos observar, el sistema binario es el que nos permite la representación de la información en las computadoras. Recordemos entonces, cómo podemos pasar de una representación binaria a decimal y hexadecimal. Por ejemplo, si tenemos:

$$11001001_2$$

Su representación decimal sería:

$(1x2^7)+ (1x2^6)+ (0x2^5)+ (0x2^4) + (1x2^3)+ (0x2^2)+ (0x2^1)+ (1x2^0)$

$= (1x128) + (1x64) + (0x32) + (0x16) + (1x8) + (0x4) + (0x2) + (1x1)$

$= 128 + 64 + 0 + 0+ 8 + 0 + 0 + 1$

$= 201_{10}$

Y en formato hexadecimal tendríamos :

1100	1001_2
$= (1x2^3) + (1x2^2) + (0x2^1)+ (0x2^0)$	$(1x2^3) + (0x2^2) + (0x2^1) + (1x2^0)$
$= 8 + 4 + 0 + 0 = 12_{10} = C_{16}$	$8 + 0 + 0 + 1 = 9_{10} = 9_{16}$
$= C9_{16}$	

Caracteres. Hace referencia a los datos de tipo texto o secuencia de caracteres. Al igual que en el caso de los operandos tipo número, la representación de los caracteres en el computador es a través de números binarios. Para ello, uno de los sistemas de representación que se usa es el <<*American Standard Code for Information Interchange*>> (ASCII). Así, por ejemplo:

ASCII	Decimal	Hexadecimal	Binario
A	65	41	01000001
9	57	39	00111001

Datos lógicos. Cuando en una unidad direccionable (bytes, palabra, etc.), se considera a cada uno de sus elementos como unidades o datos, donde cada uno de ellos tiene un valor de 1 o 0. En ese caso son considerados como datos lógicos y pueden ser aplicados con el uso de instrucciones lógicas como AND, OR o XOR.

2.3.4. Tipos de instrucciones

Las instrucciones, en general, deben ser capaces de ejecutar cuatro tipos de operaciones: (1) transferencia de datos entre la memoria y los registros del procesador, (2) operaciones lógicas y aritméticas sobre los datos, (iii) transferencia de E/S y (4) operaciones de secuenciación y control del programa. De ello, se desprende que las instrucciones de máquina pueden ser categorizadas en los siguientes tipos:

Procesamiento de datos: son instrucciones que ejecutan operaciones lógicas y aritméticas. Estas operaciones se realizan principalmente con datos en registros del procesador.

En su mayoría, las computadoras modernas proporcionan las operaciones básicas: suma, resta, multiplicación y división; y otras como negar, incrementar o decrementar. En muchas ocasiones, estas

operaciones implican procesos adicionales de transferencia de datos para ubicar los datos en la ALU y para almacenar su salida.

De igual forma, las computadoras ofrecen un conjunto de instrucciones para desarrollar operaciones lógicas básicas como la AND, OR, OR-exclusiva (XOR), NOT. Estas operaciones pueden ser aplicadas bit a bit o a unidades lógicas de n bits. Por ejemplo, si tenemos los registros (R1) y (R2) (note que la notación (X) significa el contenido almacenado en la posición X), el resultado de la operación AND es como sigue:

(R1) = 1111 0000 1110 1010 0000 1111 1111 0011

(R2) = 0000 1111 1111 1111 0000 0000 0011 1111

La operación:

(R1) AND (R2) = 0000 0000 1110 1010 0000 0000 0011 0011

Obsérvese que la operación AND se puede usar para *enmascarar* bits (forzando a bits no deseados a cero).

Por su parte la operación OR es útil para combinar bits provenientes desde dos registro, por ejemplo, en el caso de los registros R1 y R2, tenemos:

(R1) OR (R2) = 1111 1111 1111 1111 000 1111 1111 1111

Un conjunto adicional de operaciones son las de **desplazamiento lógico y aritmético**, y las de **rotación** (ver Figura 10). El en el desplazamiento lógico se desplazan a la derecha o a la izquierda, según corresponda, los bits de la palabra. El bit saliente se pierde; mientras que el bit entrante es un 0. En el caso del desplazamiento aritmético, el dato es un entero con signo (en complemento a dos) y, por lo tanto, no se desplaza el signo. Con el desplazamiento aritmético a la izquierda, se procede en igual forma que un desplazamiento lógico, llevando los bits a la izquierda, excepto el de signo. Mientras que, con el desplazamiento aritmético a la derecha, el bit de signo se replica en la posición a su derecha.

En el caso de la rotación, no se desechan los bits—como en el caso de los desplazamientos—, sino que se desarrolla un desplazamiento cíclico, pasando el bit del extremo saliente a ser el bit de entrada. Se tiene instrucciones para hacer una rotación a la izquierda o a la derecha.

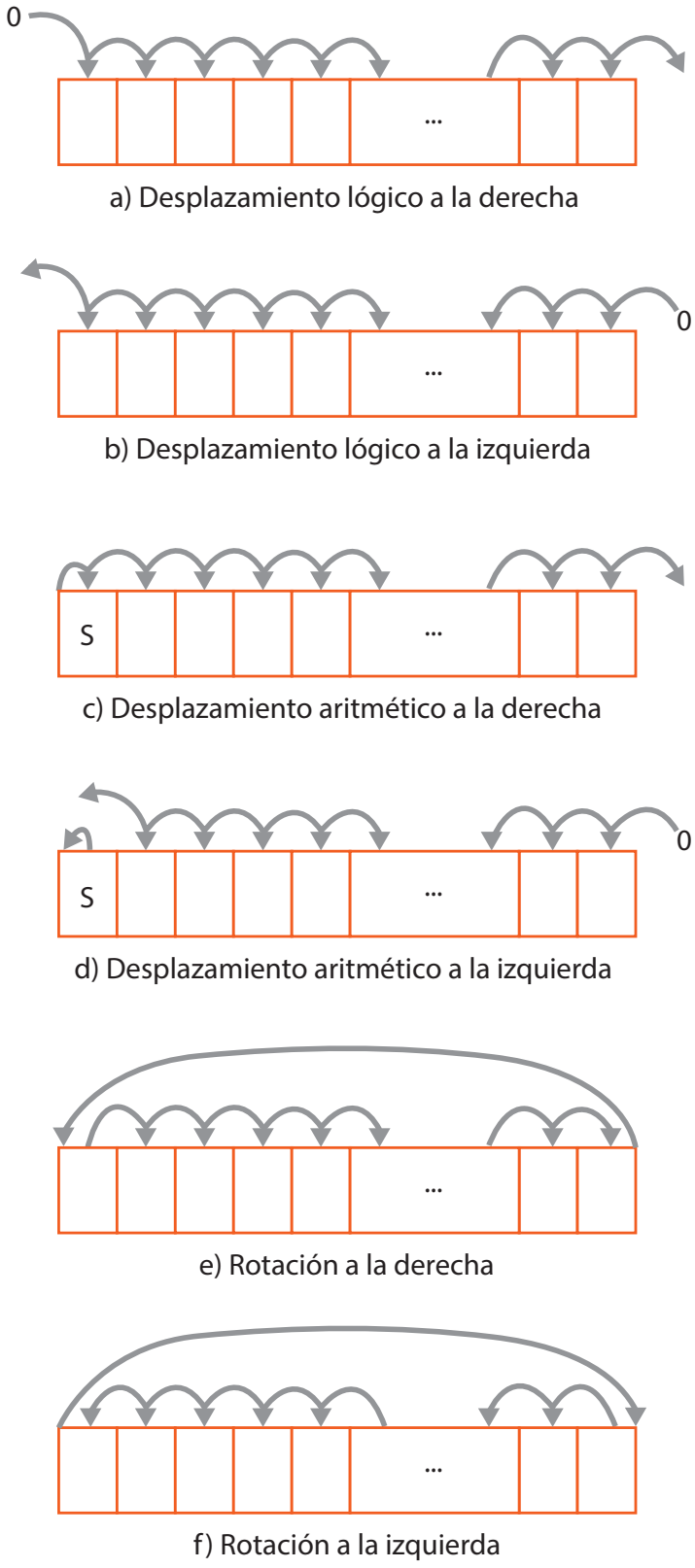


Figura 10. Operaciones de desplazamiento y rotación. Fuente: Adaptado desde (Stallings, 2013) .

Por ejemplo:

Si hacemos, para el dato 10001010 tenemos:

Desplazamiento lógico a la derecha (3 bits):	00010001
Desplazamiento lógico a la izquierda (3 bits):	01010000
Desplazamiento aritmético a la derecha (3 bits):	11110001
Desplazamiento aritmético a la izquierda (3 bits):	11010000
Rotación a la derecha (3 bits):	01010001
Rotación a la izquierda (3 bits):	01010100

Transferencia de datos: aquellas orientadas a desarrollar las operaciones de transferencia de datos entre la memoria y los registros del procesador.

Generalmente, estas instrucciones deben especificar las posiciones de los operandos origen y destino: memoria, registro o la cabecera de la pila; la longitud de los datos a transferir—aunque, en algunos computadores, puede estar implícito en la instrucción, dentro del código de operación— y el modo de direccionamiento para cada operando. Ejemplos de instrucciones de este tipo, considerando el lenguaje ensamblador de MIPS, son: *lw* (del inglés, <<*load word*>>), que permite transferir palabras desde memoria a un registro; *sw* (del inglés, <<*store word*>>), para la transferencia de palabras desde registros a memoria. En el siguiente tema, veremos con mayor detalle algunas de las instrucciones usadas por MIPS para la transferencia de datos.

E/S: son instrucciones para E/S. Nos permiten desarrollar procesos de lectura y escritura en un puerto de E/S. Dependiendo del computador, se puede tener dos modos de direccionamiento: asignado en memoria e independiente.

En el caso de las asignadas a memoria, se hace uso del mismo conjunto de instrucciones que las de transferencia de datos. Se tiene un mapeo común de la memoria y los dispositivos de E/S: el procesador considera a los registros de estado y de datos de los módulos de E/S como posiciones de memoria y utiliza las mismas instrucciones para acceder a ambos (Stallings, 2013).

Por su parte, en el caso del direccionamiento independiente, las instrucciones para acceder a los puertos de E/S son independientes de las utilizadas en la trasferencia de datos. Los puertos de E/S solo son accesibles mediante una orden específica de E/S. Por ejemplo, las instrucciones IN, que permite leer un puerto; o la instrucción OUT, que permite escribir en un puerto.

Control: relacionadas con las operaciones de secuenciación y control del programa. Veamos brevemente cómo se desarrolla la ejecución de un programa, para posteriormente comprender algunas de las instrucciones que nos permitirán controlar el flujo del programa.

El procesador contiene un registro denominado PC (del inglés, <<*program counter*>>), el cual tiene como finalidad mantener la dirección de la próxima instrucción a ser ejecutada. El procesador utiliza

la información del PC para captar y ejecutar las instrucciones, una a la vez, proceso conocido como *secuenciación en línea recta*. Durante la ejecución de la instrucción, el PC es incrementado para apuntar a la siguiente secuencia.

La ejecución de una instrucción tiene dos fases: la captación y ejecución. En la primera fase, la instrucción es captada desde la localización de memoria indicada en el PC. Esta instrucción es almacenada en el registro de instrucción (IR, del inglés <<*instruction register*>>) del procesador. En la segunda fase, la instrucción en el IR es examinada para determinar cuál es la operación que se desea. Luego de ello, el procesador la ejecutará. En esta fase, se puede desarrollar operaciones adicionales como la captación de operandos, ejecución de operaciones lógicas o aritméticas, y el almacenamiento del resultado en la ubicación de destino. Luego de esta fase, se procede con la siguiente instrucción, la cual es apuntada por el PC.

Instrucciones de bifurcación. Este tipo de instrucciones permite almacenar una nueva dirección en el PC. Como consecuencia, el procesador capta y ejecuta la instrucción indicada por la nueva dirección en la PC, llamada *objetivo de bifurcación*, en lugar de aquella que sigue a la instrucción de bifurcación, de acuerdo al orden secuencial de ejecución. En este tipo de instrucciones tenemos dos formas: condicionales e incondicionales.

Las *instrucciones de bifurcación condicional*, causa una bifurcación siempre y cuando se satisface alguna condición establecida. Si esta condición no se cumple, el PC incrementa su valor en forma normal y la próxima instrucción en el orden de dirección secuencial es captada y ejecutada. Ejemplos de este tipo de instrucciones, en el lenguaje ensamblador MIPS, son las instrucciones: *beq* (del inglés <<*branch if equal*>>), la cual produce un salto a una dirección determinada, si el contenido de sus operandos (registros) son iguales. En el caso de la instrucción *bne* (del inglés <<*branch if not equal*>>), se produce el salto, siempre que el contenido de los operandos (registros) de la instrucción no sean iguales.

Por su parte, las *instrucciones de bifurcación incondicional*, obligan al procesador a seguir siempre la bifurcación. Por ejemplo, en el caso de MIPS, tenemos la instrucción *j* (del inglés, <<*jump*>>).

Instrucciones de llamada y retorno de procedimientos o funciones. Las funciones son una herramienta de programación estructurada que permite que los programas sean más fáciles de entender y que el código pueda ser reusado. Permite a los programadores concentrarse en una sola tarea del programa. Los parámetros actúan como interfaces entre el procedimiento y el programa, dado que estos permiten ingresar datos y retornar valores.

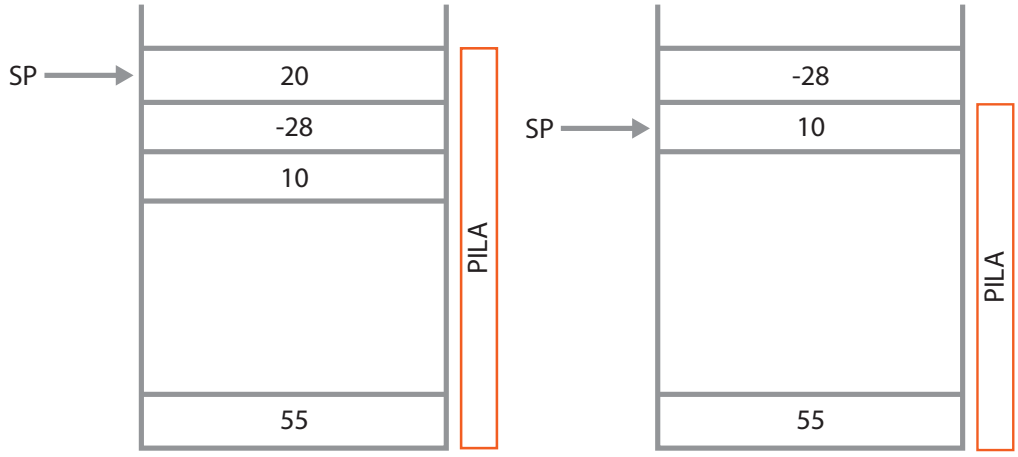
Cuando un programa bifurca hacia un procedimiento, se dice que está llamando a un procedimiento. La instrucción que desarrolla esta operación es denominada *instrucción de llamada*. Después de que el procedimiento fue ejecutado, el programa llamante debe reanudar la ejecución, continuando inmediatamente después de la instrucción que llamó el procedimiento. En este caso se dice que el procedimiento retorna al programa que la llamó; para ello, ejecuta una *instrucción de retorno*.

Antes de continuar con nuestro análisis del proceso de las instrucciones de llamada a procedimientos, veamos brevemente la estructura de un registro especial denominado ***pila***, el cual es de mucha utilidad en estos procesos—obsérvese que este quiebre en nuestro escrito, es similar a lo que hace el

programa con los procedimientos. Saltamos a estudiar la pila y luego retornamos al punto de partida sobre los procedimientos—.

La pila es un espacio de memoria, considerada como una lista de elementos de datos, *palabras*, con la restricción de acceso de que los elementos pueden ser añadidos o removidos en un extremo de la lista solamente (*cima de la pila*). El otro extremo de la pila es el fondo. Esta mecanismo de almacenaje es conocido también como *pila último en entrar primero en salir* (LIFO, del inglés <<*last-in-first- out*>>). Los términos, en inglés, <<*push*>> y <<*pop*>> son usados para referirse a las operaciones de ubicar un nuevo dato en la pila o remover un ítem desde la cima de la pila, respectivamente.

Generalmente, los computadores implementan la pila como una porción de la memoria principal. Un registro del procesador, denominado puntero de la pila (SP, del inglés <<*stack pointer*>>, es usado para apuntar a una estructura de pila particular, denominada *pila del procesador*. En la Figura 11, observamos los efectos de las operaciones *push* y *pop* sobre la pila. Así, por ejemplo, antes de las operaciones mencionadas, la pila contiene valores numéricos con -28 en la cima de la pila y 55 en la base. Como es de suponerse, dado que el SP es usado para mantener un seguimiento de la dirección del elemento que se encuentra en la cima de la pila, en esta ocasión está apuntando al de valor -28. Luego de la operación de *push*, el valor del nuevo elemento introducido se ubica en la a la cima de la pila y el SP se decrementa para pasar a apuntarlo (ver Figura 11.a). Por el contrario, en el caso de la instrucción *pop*, el SP es incrementado para apuntar a la dirección del nuevo elemento en la cima, en nuestro ejemplo el registro con valor 10 (ver Figura 11.b).



a) Después del <<*push*>> a) Después del <<*pop*>>

Figura 11. Efectos de las operaciones push y pop sobre la pila. Fuente: Adaptado desde (Hamacher, 2012) .

Volvamos ahora a nuestro análisis sobre las instrucciones de llamada y retorno de procedimientos o funciones.

Dado que un procedimiento puede ser llamado desde distintas ubicaciones, es necesario proveer algún mecanismo que permite mantener la correcta localización para el retorno. Para ello, la ubicación apuntada por el PC, cuando la instrucción de llamada al procedimiento en ejecución fue hecha, debe ser mantenida para el proceso de retorno al programa. Este proceso de hacer posible el llamar y retornar de procedimientos es conocido como *método de vinculación de procedimiento*. El principio

básico es guardar la dirección de retorno en un registro dedicado para este propósito, denominado *registro de enlace* y luego de completar la tarea, usar el contenido de este registro para volver a la ubicación original, a través de la instrucción de retorno. Como se puede intuir, las instrucciones de llamada y de retorno son instrucciones especiales de bifurcación que hacen uso del registro de enlace para mantener la dirección que les permitirá volver al programa llamante luego de ejecutar el procedimiento.

En el caso de procedimientos anidados, cuando un procedimiento llama a otro, la dirección del segundo procedimiento también es guardada sobre el registro de enlace, sobrescribiendo su contenido previo, haciendo que se pierda. Por tanto, necesitamos algún mecanismo que nos permita mantener esta dirección para poder retornar adecuadamente de los distintos procedimientos. Recordemos, que el anidamiento se puede realizar en cualquier nivel de profundidad. Por lo tanto, necesitamos que el último procedimiento que finaliza su ejecución retorne al procedimiento que lo llamó y así sucesivamente. Como se observa, las direcciones de retorno se irán necesitando desde la última hasta alcanzar a la primera, en un formato LIFO. Por ello, una buena opción es hacer uso de la pila para ir almacenando las direcciones de retorno de los diferentes procedimientos. Una vez que el procedimiento termina, saca la dirección de retorno de la pila y la coloca en el contador de programa.

2.3.5. Diseño del repertorio de instrucciones.

El repertorio de instrucciones tiene una alta importancia debido a su impacto en el funcionamiento del procesador y su efecto significativo sobre su implementación. Más aún, este es el medio que tiene el programador para controlar el procesador. Por tanto, es necesario considerar las necesidades del programador al momento de diseñar el repertorio de instrucciones. Aquí existe un problema de compromiso que permita facilitar el diseño del procesador y satisfacer las necesidades del programador.

Como hemos visto hasta el momento, las instrucciones se fundamentan en tres aspectos: *codifican* una operación básica que se ejecutan sobre unos *operandos*, ubicados en memoria y accedidos a través de una *dirección* determinada. A partir de esto, un criterio importante en el diseño de un repertorio de instrucciones es la *ortogonalidad*. Esto parte del hecho de que dos aspectos de la arquitectura son ortogonales si son independientes. En el caso de las instrucciones, se podría decir que un repertorio es ortogonal cuando pueden combinarse los aspectos anteriores sin restricciones.

Entre las cuestiones más importantes de diseño a tener en cuenta tenemos (Stallings, 2013):

- El **repertorio de operaciones**: determinar las operaciones que hay que llevar a cabo y su complejidad.
- El **tipo de datos**: determinar los tipos de datos para efectuar las operaciones.
- Los **formatos de instrucciones**: longitud, número de operandos, número de direcciones, tamaño de los campos, etc.
- Los **registros**: Determinar el número de registros del procesador.

- El **direccionamiento**: Establecer los modos de direccionamiento que se pueden usar en los operandos.

2.4. Direccionamiento

Los modos de direccionamiento indican la manera de acceder a un dato por parte de una instrucción. Esto es, nos indican las diferentes formas de especificar las ubicaciones de los operandos de las instrucciones. Entre los modos de direccionamiento más comunes tenemos (Stallings, 2013): inmediato, directo, registro, indirecto con registro, con desplazamiento y pila.

2.4.1. Direccionamiento inmediato

En este caso, el operando se encuentra presente en la propia instrucción (ver Figura 12). La ventaja de este tipo de direccionamiento es que no se necesita procesos adicionales de acceso a memoria, por lo que se hace mucho más rápida la ejecución de la instrucción. Sin embargo, su limitación se da en el tamaño del número, restringido a la longitud del campo de dirección correspondiente, el cual es pequeño comparado a la longitud de *palabra*.

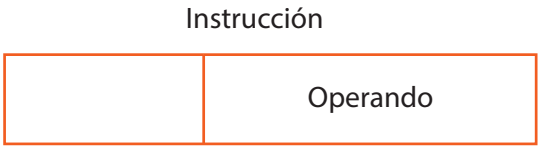


Figura 12. Direccionamiento inmediato. Fuente: Adaptado desde (Stallings, 2013).

2.4.2. Direccionamiento directo

En este tipo de direccionamiento, el campo de direcciones de la instrucción contiene la dirección efectiva del operando, la cual corresponde a la dirección de memoria principal. No se requiere ningún cálculo adicional; pero su limitación se da en su espacio de direcciones restringido. La Figura 13, ilustra este modo de direccionamiento.

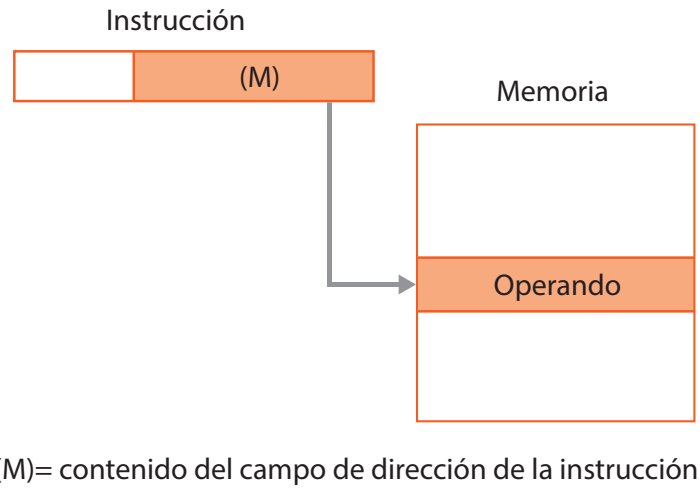


Figura 13. Direccionamiento directo. Fuente: Adaptado desde (Stallings, 2013).

2.4.3. Direccionamiento indirecto

La dirección efectiva del operando está contenida en una posición de memoria principal que es especificada en la instrucción (ver Figura 14). Con esta forma de direccionamiento se solventa en parte la limitación del espacio de direcciones del modo directo, dado que con un ancho de palabra de N, se puede acceder a un espacio de direcciones de 2^N. Su desventaja se da que para obtener el valor del operando se necesita referenciar a dos espacios de memoria: uno para captar la dirección donde está contenido y otra para obtener su valor.

Hay que considerar que el campo de direccionamiento que se puede alcanzar con este modo de direccionamiento es igual a 2^N; no obstante, la longitud del campo de direcciones K, generalmente es diferente a N, por lo que el campo efectivo de direcciones está limitado a 2^K. Por lo general, el tamaño de las direcciones del campo efectivo de direcciones 2^K es superior al que se puede acceder desde el modo de direccionamiento indirecto. Por ello, solo se podrá acceder a un bloque de memoria que habitualmente se reserva como tabla de punteros en las estructuras de datos que utiliza el programa.

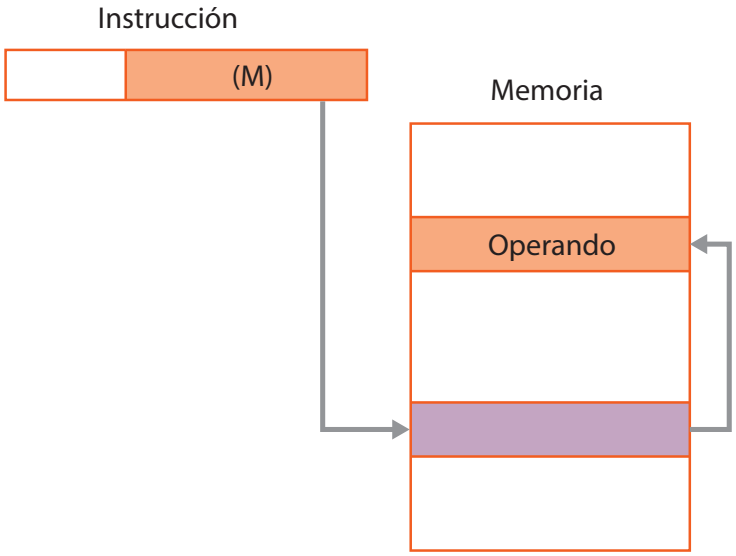


Figura 14. Direccionamiento indirecto. Fuente: Adaptado desde (Stallings, 2013).

2.4.4. Direccionamiento de registros

Se podría ver como un caso de direccionamiento directo, en donde se direcciona a un registro y no a una ubicación de memoria principal (ver Figura 15). Dado que se referencia a un registro del procesador, se tiene la ventaja de que solo es necesario un campo pequeño de direcciones en la instrucción y, además, no se requiere referencias a posiciones de memoria principal, disminuyendo el tiempo de acceso. No obstante, su desventaja se da en da en la restricción de un espacio de direcciones muy limitado.

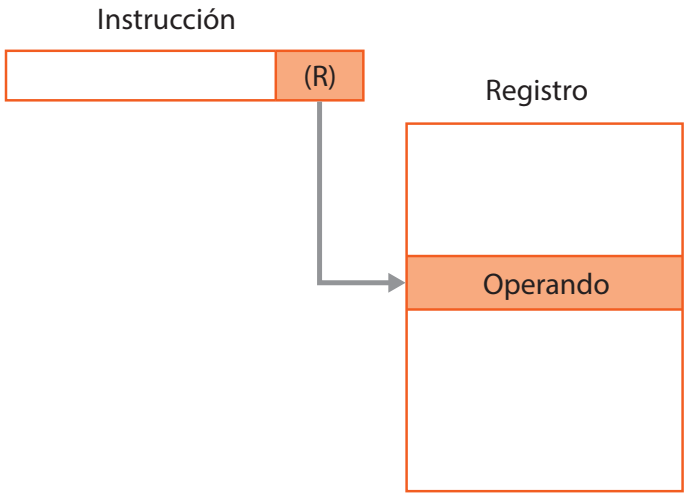


Figura 15. Direccionamiento de registros. Fuente: Adaptado desde (Stallings, 2013).

2.4.5. Direccionamiento indirecto con registros

Es similar al direccionamiento indirecto, con la diferencia de que se hace referencia a un registro más que a una ubicación en la memoria principal (ver Figura 16). Las ventajas y desventajas son prácticamente las mismas que en el caso del direccionamiento indirecto. Sin embargo, como es de suponer, el direccionamiento indirecto a registro emplea una referencia menos a memoria principal que el direccionamiento indirecto.

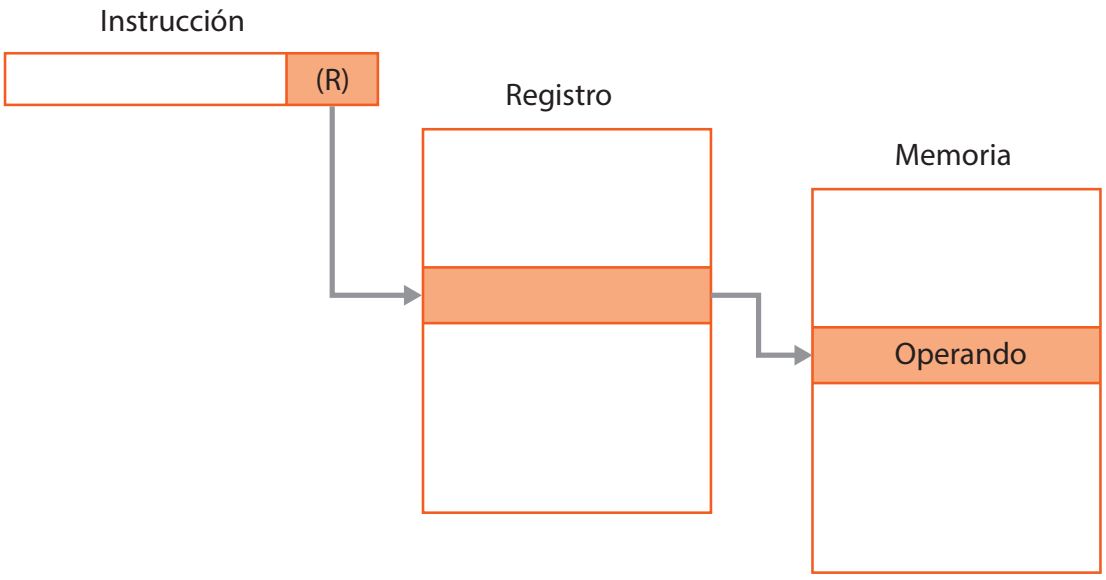
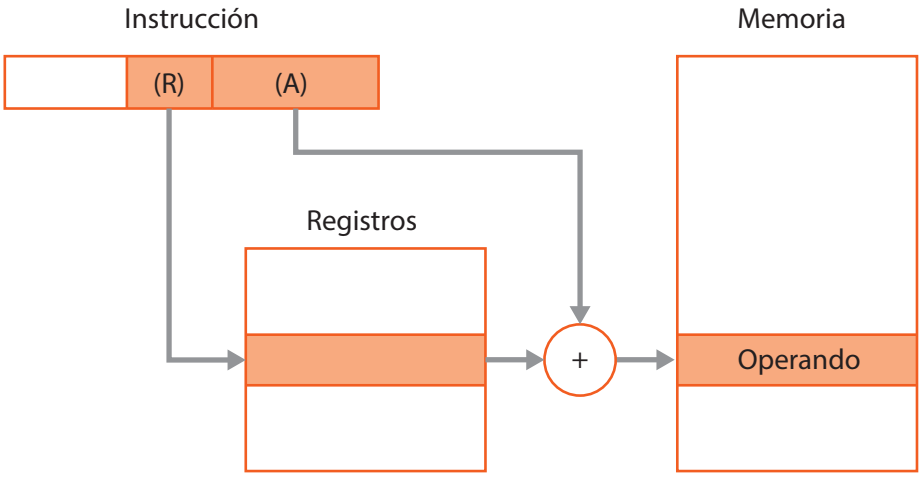


Figura 16. Direccionamiento indirecto con registros. Fuente: Adaptado desde (Stallings, 2013).

2.4.6. Direccionamiento con desplazamientos

En este tipo se combinan las potencialidades del direccionamiento directo e indirecto con registros (ver Figura 17). Para ello, las instrucciones deben expresar dos valores en los campos de direcciones: el primero se utiliza directamente (desplazamiento); mientras que el segundo, se refiere a un registro cuyo contenido se suma al primero para generar la dirección efectiva (EA, del inglés <<*effective address*>>).

dirección efectiva (EA) = contenido del registro + desplazamiento



(R)= contenido del campo de dirección en la instrucción que referencia un registro

(A)= contenido del campo de dirección en la instrucción

Figura 17. Direccionamiento con desplazamientos. Fuente: Adaptado desde (Stallings, 2013).

Entre los principales usos tenemos:

- **Desplazamiento relativo**. Se hace uso del contador de programa (PC) para obtener la dirección efectiva. Para obtener la EA se suma al contenido del PC el valor del desplazamiento.
- **Direccionamiento con registro base**. En este caso, el registro referenciado (registro base) contiene una dirección de memoria, a la cual se le suma el desplazamiento. La referencia al registro base puede ser implícita o explícita.
- **Indexado**. Sigue el mismo proceso que el direccionamiento con registro base, con la diferencia de que el valor del desplazamiento se almacena en un registro denominado *registro índice*; mientras que la dirección de memoria se encuentra explícita en la instrucción.

2.4.7. Direccionamiento de pila

Este es un modo de direccionamiento implícito, que trabaja directamente con la pila. No se requiere hacer una referencia explícita a la pila, sino que direcciona directamente a la cima de la pila, a través de un registro denominado puntero de pila (SP, del inglés <<*stack pointer*>>). Las instrucciones más

habituales, como ya se vio en la Sección 2.3.4, son *push* (poner un elemento en la pila) y *pop* (sacar un elemento de la pila).

Tema 3.
Estructura de un computador en el nivel de lenguaje de máquina y programación en ensamblador

Para profundizar en los conceptos estudiados en el tema anterior, ahora vamos a analizar diversos ejemplos de la implementación de instrucciones en un lenguaje ensamblador de una máquina real, como es MIPS (<<*Microprocessor without Interlocked Pipeline Stages*>>). Esto nos permitirá observar cómo se traduce un programa en lenguaje de alto nivel en las instrucciones de lenguaje ensamblador correspondiente. Los ejemplos utilizados en esta sección están, en su mayoría, tomados o basados en (David, A Patterson and John, 2005) y (Harris, David and Harris, 2010).

MIPS maneja un repertorio de 32 registros, con un ancho de registro (*palabra*) de 32 bits:

Nombre	Número	Uso	Conservado a través de un procedimiento
$zero	0	El valor constante 0	N.A
$at	1	Ensamblador temporal	No
$v0-$v1	2-3	Valores para resultados de funciones y evaluación de expresiones	No

Nombre	Número	Uso	Conservado a través de un procedimiento
$a0-$a3	4-7	Argumentos	No
$t0-$t7	8-15	Temporales	No
$s0-$s7	16-23	Temporarios guardados	Si
$t8-$t9	24-25	Temporarios	No
$k0-$k1	26-27	Reservados para el Kernel del SO	No
$gp	28	Puntero global	Si
$sp	29	Puntero a pila	Si
$fp	30	Puntero a trama	Si
$ra	31	Dirección de retorno	Si

Como podemos observar en la tabla anterior, los nombres de los registros están precedidos por el signo $. MIPS generalmente almacena variables en 18 de los 32 registros: $s0-$s7 y $t0-$t9. Los registros $s sirven para almacenar variables y tienen uso en los llamados a procedimientos; mientras que los registros $t almacenan variables temporales. Otro importante registro es el $zero, el cual mantiene un valor constante de 0, de frecuente uso por los programadores. A medida que avancemos en nuestro estudio iremos ampliando la información de los otros registros.

En cuanto a la memoria principal. MIPS usa direcciones de memoria de 32 bits y *palabras* con un ancho de 32 bits. En cuanto al direccionamiento de la memoria, MIPS trabaja con una memoria direccionable por byte. Esto es, cada palabra de 32 bits, tiene una única dirección de 32 bits (ver Figura 18).

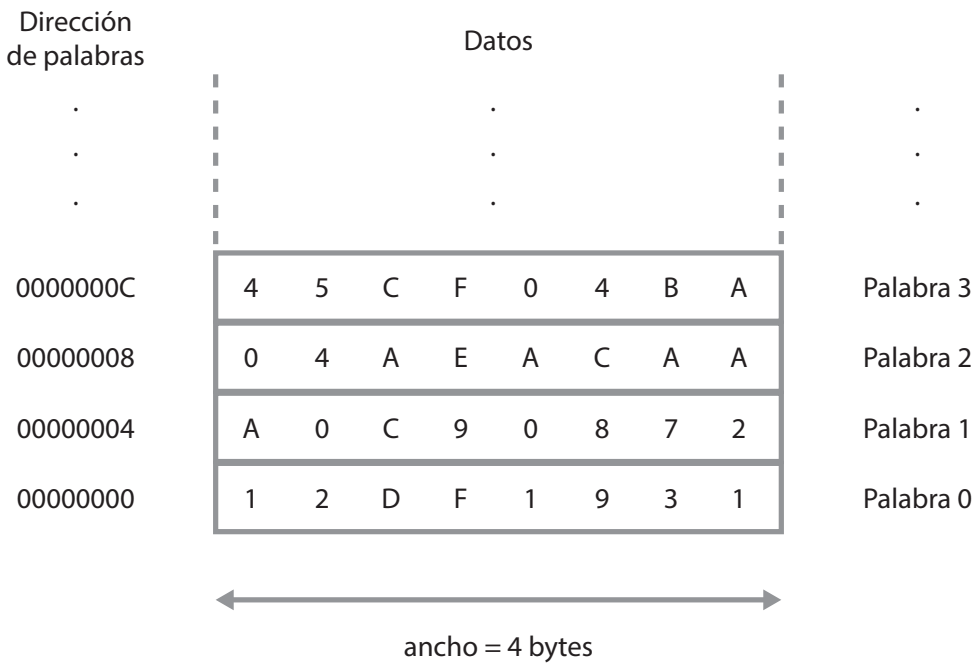


Figura 18. Memoria direccionable por byte. Fuente: Adaptado desde (Harris, David and Harris, 2010) .

En cuanto al lenguaje de máquina, MIPS usa instrucciones de 32 bits, por lo que todas las instrucciones pueden ser almacenadas en una *palabra* de la memoria. Posee tres formatos de instrucciones:

Instrucciones tipo R. Estas instrucciones son las denominadas tipo registro (del inglés <<*register-type*>>). Hacen uso de tres registros como operandos: dos como fuente y uno como destino.

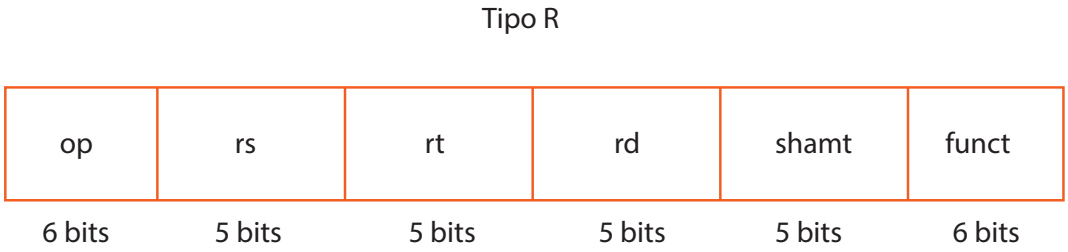


Figura 19. Formato de la instrucción de máquina tipo R. Fuente: Adaptado desde (Harris, David and Harris, 2010) .

En la Figura 19, se puede observar los siguientes campos en las instrucciones tipo R. *op* (el código de operación), en conjunto con *funct* (función) indican la operación que debe realizar la instrucción. Todo *op* de las instrucciones tipo R tienen un valor de 0, la función es determinada por el campo *funct*. Por ejemplo, para el caso de la instrucción *add*, la cual permite desarrollar una suma, el *op* tiene un valor de 00000_2 y *funct* un valor de 100000_2.

Los operandos son codificados en tres campos: *rs, rt* y *rd*. Los dos primeros son los registros fuente; mientras que el tercero es el registro destino. Finalmente, el campo *shamt* es usado en operaciones de desplazamiento. En los demás casos tiene un valor de 0.

Instrucciones tipo I. Son denominadas de tipo inmediato (del inglés <<*immediate-type*>>). Estas instrucciones usan dos operandos tipo registro y uno de tipo inmediato.

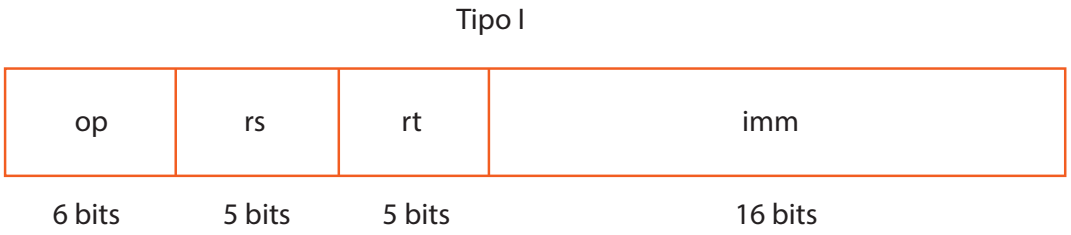


Figura 20. Formato de la instrucción de máquina tipo I. Fuente: Adaptado desde (Harris, David and Harris, 2010).

En la Figura 20, podemos observar los diferentes campos de este tipo de instrucciones:

Tenemos el código de la operación representada únicamente por el campo *op* y los operandos que están en tres campos: *rs, rt* e *imm*. *rs* e *imm* son usados como los operandos fuente; mientras que *rt* puede ser usado, dependiendo de la instrucción, como destino o como otro operando fuente.

Instrucciones tipo J. Son denominadas de tipo bifurcación (del inglés <<*jump-type*>>). Este formato es usado exclusivamente con instrucciones para bifurcaciones. Al igual que los otros dos tipos, se tiene un campo de *op* para representar la codificación de la instrucción; pero los restantes bits de la palabra son usados para el campo *addr*, que representa el operando de dirección.

Tipo J

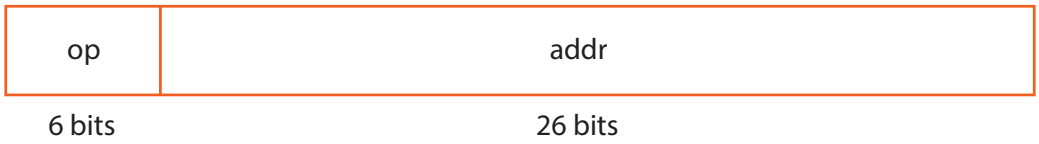


Figura 21. Formato de la instrucción de máquina tipo J. Fuente: Adaptado desde (Harris, David and Harris, 2010).

Como ejemplo, descifremos la siguiente instrucción de máquina traduciéndola al lenguaje ensamblador: 0x02F34020.

En primera instancia, observemos que la instrucción de máquina está expresada en hexadecimal, por lo que tenemos que ubicarla en binario. Recordemos que cada uno de los números en formato hexadecimal los tenemos que establecer sobre 4 bits. Esto es:

0	2	F	3	4	0	2	0
0000	0010	1111	0011	0100	0000	0010	0000

Dado que todos formatos de instrucción tienen un op de 6 bits, este es nuestro punto de partida para determinar de qué instrucción se trata.

El *op* es 000000, por lo que se trata de una instrucción tipo R. El campo *funct* es 100000, lo cual nos indica que es de adición.

op	rs	rt	rd	shamt	funct
000000	10111	10011	01000	00000	100000

Traducido al lenguaje ensamblador, revisando las direcciones de los registros, tendríamos:

add $t0, $s7, $s3

3.1. Programación

Como observamos, la instrucción de suma anterior está escrita en lenguaje ensamblador. En ella, *add* corresponde al nemotécnico que codifica la operación que debe realizar la instrucción, en este caso sumar; el registro destino es $t0; mientras que los dos registros fuente son $s7 y $s3. Esto difiere de la ubicación de los registros en la instrucción de máquina, en donde primero aparecen los registros fuente y luego el registro destino.

En esta sección analizaremos cómo se lleva al lenguaje ensamblador, muchas de las construcciones de software comunes en los lenguajes de alto nivel: operaciones lógicas y aritméticas, declaraciones condicionales, lazos y llamados a procedimientos.

3.1.1. Instrucciones lógicas y aritméticas

Las instrucciones lógicas que implementa MIPS tenemos *and*, *or*, *xor* y *nor*. Estas operaciones son de tipo R y actúan bit a bit sobre dos registros fuente y el resultado es escrito en un registro destino.

Por ejemplo, si tenemos que los registros $s1 y $s2 contiene los valores: 0x125FFAA0 y 0XFFFF0000, respectivamente. Se requiere mostrar los resultados de ejecutar las operaciones *and, or, xor* y *nor* sobre esos registros, los cuales se almacenaran en el registro $s3, $s4, $s5 y $s6, respectivamente

Las instrucciones en lenguaje ensamblador serían:

```
and, $s3, $s1, $s2     # and entre los valores de $s1 y $s2 es ubicada en $s3

or,   $s4, $s1, $s2    # or entre los valores de $s1 y $s2 es ubicada en $s4

xor,  $s5, $s1, $s2    # xor entre los valores de $s1 y $s2 es ubicada en $s5

nor,  $s6, $s1, $s2    # nor entre los valores de $s1 y $s2 es ubicada en $s6
```

Los valores de los registros $s1 y $s2 son los siguientes:

$s1	0001	0010	0101	1111	1111	1010	1010	0000
$s2	1111	1111	1111	1111	0000	0000	0000	0000

El contenido de los registros destino, luego de las operaciones es:

$s3	0001	0010	0101	1111	0000	0000	0000	0000
$s4	1111	1111	1111	1111	1111	1010	1010	0000
$s5	1110	1101	1010	0000	1111	1010	1010	0000
$s6	0000	0000	0000	0000	0000	0101	0101	1111

MIPS no ofrece la operación *not*, pero podemos hacer la misma acción a través de la operación *nor*; así, A *nor* $zero = *not* A.

Para trabajar operaciones lógicas con valores inmediatos tenemos las instrucciones *andi*, *ori*, *xori*. Recordemos que estas instrucciones son de tipo I, por lo que el valor inmediato y el del registro deben contener 16 bits.

Así, las instrucciones:

```
andi, $s2, $s1, 0xFF40     # and entre los valores de $s1 y 0xFF40 es ubicada en $s2

ori,   $s3, $s1, 0xFF40    # or entre los valores de $s1 y 0xFF40 es ubicada en $s3

xori,  $s4, $s1, 0xFF40    # xor entre los valores de $s1 y 0xFF40 es ubicada en $s4
```

Los valores del registro $s1 y del operando inmediato es:

$s1	0000	0000	0000	0000	1111	1010	1010	0000
imm	0000	0000	0000	0000	1111	1111	0100	0000

El contenido de los registros destino, luego de las operaciones es:

$s2	0000	0000	0000	0000	1111	1010	1010	0000
$s3	0000	0000	0000	0000	1111	1111	1110	0000
$s4	0000	0000	0000	0000	0000	0101	1110	0000

En el caso de las operaciones aritméticas, *add, sub* y *addi*, las operaciones son similares a las explicadas en el caso de las operaciones lógicas. No obstante, cabe indicar que una instrucción en lenguaje ensamblador equivale a una instrucción en lenguaje de máquina. Dado que las operaciones aritméticas y lógicas trabajan solo con tres operandos, ¿qué sucede cuando tenemos operaciones que involucran más variables? Por ejemplo, si queremos ejecutar la siguiente operación expresada en un lenguaje de alto nivel como C.

$$x = (y + z)-(w + a)$$

Como no podemos ejecutar esta instrucción en su totalidad, el compilador debe fraccionarla en varias instrucciones, dado que MIPS ejecuta una instrucción a la vez. Por ende, lo primero que se debe hacer es sumar la variables *y* y *z*, guardando su resultado en un registro temporal, digamos $t0. Vamos a suponer que el compilador asoció las variables del programa x, y, z, w, a con los registros $s0, $s1, $s2, $s3 y $s4, respectivamente

add $t0, $s1, $s2 # la variable temporal $t0 contiene el valor de y+z.

Ahora necesitamos obtener el valor de w+a, por lo que usamos un segundo registro temporal para almacenar el resultado, $t1.

add $t1,$s3, $s4 # la variable temporal $t1 contiene el valor de w+a.

Finalmente realizamos la substracción entre $t1 y $t0.

sub, $s0, $t0, $t1 # la variable guardada $s0 contiene el valor de x = (y+z)-(w+a).

En el caso de las instrucciones de desplazamiento, MIPS provee de las instrucciones *sll* (<<*shift left logical*>>), para el desplazamiento lógico a la izquierda; *srl* (<<*shift right logical*>>), para el desplazamiento lógico a la derecha y *sra* (<<*shift right arithmetic*>>), para el desplazamiento aritmético a la derecha.

Así, las instrucciones en código ensamblador:

sll, $t0, $s0, 4 # Desplazamiento lógico a la izquierda del contenido de $s0

srl, $t1, $s0, 4 # Desplazamiento lógico a la derecha del contenido de $s0

sra, $t2, $s0, 4 # Desplazamiento aritmético a la derecha del contenido de $s0

Observemos la primera de estas instrucciones en lenguaje de máquina:

op	*rs*	*rt*	*rd*	*shamt*	*funct*
000000	00000	10001	01000	00100	000000

Los valores del registro $s0 son:

$s0	1110	0000	0000	0000	1111	1010	1010	0000

Además el contenido del campo shamt indica el valor del desplazamiento 4_{10}.

El contenido de los registros destino, luego de las operaciones es:

$t0	0000	0000	0000	1111	1010	1010	0000	0000
$t1	0000	1110	0000	0000	0000	1111	1010	1010
$t2	1111	1110	0000	0000	0000	1111	1010	1010

3.1.2. Bifurcación

Veamos ahora algunos ejemplos de instrucciones para bifurcaciones. MIPS tiene dos instrucciones de bifurcación condicional: *beq* (del inglés <<*branch if equal*>>), bifurcación si es igual; y *bne* (del inglés <<*branch if not equal*>>), bifurcación si no es igual. Estas instrucciones son conocidas como bifurcaciones condicionales.

En lenguaje ensamblador, el formato de estas instrucciones es como sigue:

beq registro_1, registro_2, L_1

Esta instrucción significa que si el contenido del registro_1 es igual al del registro_2, se continúe la ejecución del programa desde la parte etiquetada como L_1. En el caso de *bne*, su formato es el siguiente:

bne registro_1, registro_2, L_1

A diferencia de la anterior, *bne* desarrolla la bifurcación hacia L_1 siempre que el contenido de los dos registros no sea igual.

Veamos un ejemplo de cómo una declaración condicional en un lenguaje de alto nivel como C, es representada en lenguaje de ensamblador en la arquitectura MIPS:

Supongamos que tenemos las variables *a, b, d, e, f* y deseamos ejecutar el siguiente código: *if (a==b) d = e + f; else d= d – a;*

Si consideramos que las variables *a, b, d, e* y *f*, corresponden a los registros desde $s0 a $s4, respectivamente, la correspondiente traducción en lenguaje ensamblador sería:

```
bne $s0, $s1, else

sub $s2, $s2, $s0          # d = d - a

else:

add $s2, $s3, $s4          # d = e + f
```

Observe que para hacer más eficiente el programa, hemos hecho uso de la instrucción *bne*, más que de *beq*.

Para el caso de bifurcaciones incondicionales, MIPS hace uso de tres tipos de instrucciones de salto: saltar, *j* (del inglés <<*jump*>>); saltar y enlazar, *jal* (del inglés <<*jump and link*>>); y registro de salto, *jr* (del inglés <<*jump register*>>).

La instrucción *j*, salta directamente a ejecutar la instrucción en la etiqueta especificada. La instrucción *jal*, actúa en forma similar que *j*, pero es usada en los procedimientos para guardar la dirección de retorno. En el caso de *jr*, el salto se da a la dirección especificada en el contenido de un registro. En el siguiente ejemplo, se muestra el accionar de la instrucción *j*:

```
addi $s0, $zero, 10        #  $s0 = 10

addi $s1, $zero, 4         # $s1 = 4

j  L_1

addi $s0, $s0, 1           # no ejecutada

sub $s1, $s1, $s0          # no ejecutada

L_1:

add $s1 $s1, $s0           # $s1 = 4 + 10 = 14
```

3.1.3. Lazos

Algunas instrucciones en lenguaje de alto nivel, permiten implementar lazos, por ejemplo las instrucciones *while* o *for*. Veamos cómo se traducirían estas instrucciones al lenguaje ensamblador de MIPS.

Para el caso de *while*:

Código de alto nivel	Código ensamblador en MIPS

int x = 0;
int i = 1;
 	
while (x != 10)
{
x = x + i;
i = i + 1;
}
	# $s0=x, $s1=i
	addi $s0, $zero, 0 # x = 0
	addi $s1, $zero, 1 # i =1
	addi $t0, $zero, 10 # $t0 = 10
	while:
	beq $s0, $t0, L_1 # si x==10, salir del bloque *while*
	add $s0, $s0, $s1 # x = x + i
	addi $s1, $s1, 1 # i = i + 1
	L_1:

Hagamos el mismo proceso anterior, pero usando la instrucción *for*:

Código de alto nivel	Código ensamblador en MIPS
int x = 0; *for (x=0; x != 10; x++)* *{* *i = i + 1;* *}*	# $s0=x, $s1=i addi $s0, $zero, 0 # x = 0 addi $s1, $zero, 1 # i =1 addi $t0, $zero, 10 # $t0 = 10 for: beq $s0, $t0, L_1 # si i==10, salir del bloque for addi $s0, $s0, 1 # x = x + i addi $s1, $s1, 1 # i = i + 1 j for L_1:

En muchas ocasiones es útil analizar si una variable es menor que otra. Para esos casos MIPS ofrece una instrucción que compara dos registros, fijando un tercer registro a 1 si el primero es menor que el segundo; caso contrario lo establece en 0. Se denomina *slt*, del inglés <<*set on less than*>>. Por ejemplo:

```
slt    $t0,  $s0,  $s1      #  $t0 = 1   si   $s0 < $s1
```

3.1.4. Vectores

Los vectores (del inglés <<*arrays*>>) son organizadas como direcciones secuenciales de datos en memoria. Cada elemento del vector es identificado a través de un número, denominado *índice*. El número de elementos del vector es conocido como ancho del vector. En esta sección veremos cómo trabajar con ellos en el lenguaje ensamblador.

Trabajemos con el siguiente código de alto nivel y su traducción a ensamblador (Harris, David and Harris, 2010):

Nº	Código de alto nivel	Código en ensamblador MIPS
1	*int* array [5];	# $s0 dirección base del vector
2		*lui* $s0, 0x1000 # $s0 = 0x10000000
3		*ori* $s0, $s0, 0x7000 # $s0 = 0x10007000
4		
5	array [0] = array [0] * 8;	*lw* $t1, 0($s0) # $t1 = array[0]
6		*sll* $t1, 3 # $t1 = $t1 << 3 = $t1 * 8
7		*sw* $t1, 0($s0) # array[0] = $t1
8		
9	array [1] = array [1] * 8;	*lw* $t1, 4($s0) # $t1 = array[1]
10		*sll* $t1, 3 # $t1 = $t1 << 3 = $t1 * 8
11		*sw* $t1, 4($s0) # array[1] = $t1

Con fines de claridad en la explicación del código anterior, hemos numerado cada una de sus líneas. En la línea 1, del código de alto nivel, podemos observar la declaración de una vector, con cinco elementos. Para poder comprender cómo se traduce esto en un lenguaje ensamblador es necesario analizar cómo son almacenados los vectores en la memoria principal. La Figura 22 muestra el vector <<*array*>> de cinco elementos, localizado en cinco posiciones consecutivas desde la dirección 0x10007000, conocida como *dirección base*, la cual muestra la dirección del primer elemento del vector, array[0].

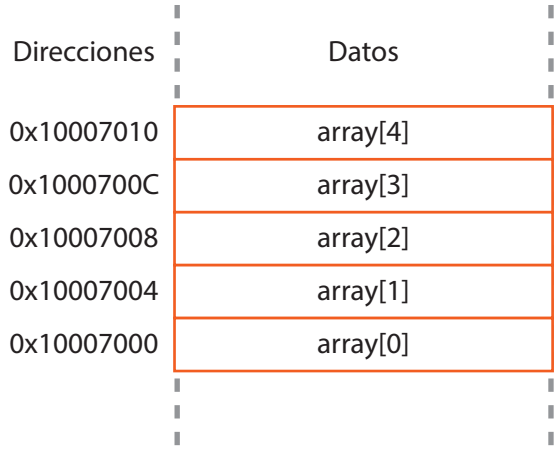


Figura 22. Vector de cinco elementos con dirección base 0x10007000. Fuente: Adaptado desde (Harris, David and Harris, 2010).

En el código de ejemplo, las líneas 2 y 3 cargan la dirección base en el registro $s0. Para ello, utilizan dos instrucciones: *lui* (del inglés <<*load upper immediate*>>), que permite almacenar un dato de 16 bits de la parte superior de una *palabra*; mientras que con la instrucción *ori* (*or* con un dato inmediato), podemos usarla para completar la dirección con los 16 bits menos significativos. Así, con la primera instrucción cargamos en $s0 el valor de 0x1000000; mientras que con la segunda instrucción hacemos una *or* entre $s0=10000000 y 0x00007000, de esta forma se combinan los dos valores y obtenemos la dirección base almacenada en $s0=10007000.

Siguiendo con nuestro código ejemplo, en la línea 5, se observa que el contenido del elemento 0 del vector *array* es multiplicado por 8 y almacenado en el mismo elemento. Para hacer esto, es necesario copiar el valor del elemento 0 en un registro temporal y sobre este desarrollar la operación necesaria y su resultado cargarlo de nuevo al elemento 0 del vector.

Por ello, en la línea 5, usamos la instrucción *lw* (del inglés <<*load word*>>) para copiar el contenido del array[0] en el registro temporal $t1. Esto es:

lw $t1, 0($s0)

La instrucción dice que se cargue en el registro temporal 1, el contenido de la posición de memoria direccionada por el contenido de $s0 + constante = 0. Dado que la dirección base del *array* está almacenado en $s0, y el desplazamiento es de 0 bytes, entonces 0($s0) está apuntando al array[0]. Si observamos la línea de código 9, 4($s0) está apuntando 4 bytes después de la dirección base. Como las palabras son de 32 bits, esta vez está apuntando al array[1].

Las líneas de código 6 y 10, nos muestran cómo desarrollar la multiplicación por 8 a través de un desplazamiento lógico a la izquierda. Cada desplazamiento de un bit a la izquierda implica una multiplicación por dos, por ello un desplazamiento de 3 bits permite la multiplicación por 8.

Finalmente, las líneas 7 y 11 nos muestran el uso de la instrucción *sw* (del inglés <<*store word*>>) para almacenar la *palabra* contenida en el registro a la posición de memoria del array[0] y el array[1], respectivamente.

3.1.5. Llamadas a procedimientos

MIPS hace uso de los siguientes registros para llamar a un procedimiento (David, A Patterson and John, 2005):

- $a0 - $a3: cuatro registros argumento en el cual pasar los argumentos.
- $v0 - $v1: dos registros valor, en el cual pasar los valores de retorno.
- $ra: un registro de retorno de dirección para retornar al punto de origen.

MIPS hace uso de la instrucción *jal* (<<*jump and link*>>) para llamar a un procedimiento y de la instrucción *jr* (<<*jump register*>>) para retornar. *jal* permite saltar a la posición de memoria donde se encuentran las instrucciones del procedimiento llamado, pero simultáneamente almacena la información de la dirección de la siguiente instrucción al de la llamada al procedimiento, en el registro $ra. Para ello, se hace uso del contenido del registro PC (<<*program counter*>>) —el cual tiene almacenado la dirección de la instrucción que está en ejecución— de forma tal que $ra almacena PC + 4. De esta forma, cuando se retorna del procedimiento llamado mediante el uso de la instrucción *jr*, se hace uso del registro $ra para retornar a la posición original desde donde se invocó al procedimiento. Por ejemplo, el siguiente código nos muestra la implementación de un procedimiento para sumar cuatro argumentos: en lenguaje de alto nivel y su traducción al lenguaje de máquina.

Nº	Código de alto nivel	Código en ensamblador MIPS
1	*int* main();	# $s0 = x
2	{	main:
3	*int* x	…
4	…	*addi* $a0, $zero, 3 # argumento 0 = 3
5		*addi* $a1, $zero, 7 # argumento 1 = 7
6	x = sum (3, 7, 4, 2);	*addi* $a2, $zero, 4 # argumento 2 = 4
7	….	*addi* $a3, $zero, 2 # argumento 3 = 2
8	}	*jal* sum # llamada a procedimiento
9		*add* $s0, $v0, $zero # y= valor retornado
10	*int* sum (int a, int b, int c, int d)	…
11	{	
	int y;	# $s0 = y
	y = a + b + c + d;	sum:
	return y;	*add* $t0, $a0, $a1 # $t0 = a + b
	}	*add* $t1, $a2, $a3 # $t1 = c + d
		add $s0, $t0, $t1 # y = a + b + c + d
		addi $v0, $s0, $zero # valor de retorno en $v0
		jr $ra # retorno al llamante

MIPS divide los registros en dos categorías preservados y no preservados. Por una parte, los registros temporarios $t0-t9 son no preservados; mientras que los registros salvados $s0-$s7 están dentro de la categoría de preservados. Un procedimiento debe guardar y restaurar algún registro preservado que desee utilizar; pero puede cambiar libremente los registros no preservados.

En el código anterior, hace falta completar el lenguaje ensamblador de manera que el registro $s0 sea guardado y restaurado por el procedimiento llamado, sum.

Nº	Código de alto nivel	Código en ensamblador MIPS
1	*int* sum (int a, int b, int c, int d)	# $s0 = y
2	{	
3	*int* y;	
4	y = a + b + c + d;	sum:
5	*return* y;	*addi* $sp, $sp, -4 # obtener espacio en la pila para almacenar un registro
6	}	*add* $t0, $a0, $a1 # $t0 = a + b
7		*add* $t1, $a2, $a3 # $t1 = c + d
8		*add* $s0, $t0, $t1 # y = a + b + c + d
9		*addi* $v0, $s0, $zero # valor de retorno en $v0
10		*lw* $s0, 0($sp) # restauramos el valor de $s0.
11		*addi* $sp, $sp, 4 # desasignamos el espacio de pila
12		*jr* $ra # retorno al llamante
13		

Como se puede observar, se debe almacenar los registros antes de ser usados en los procedimientos. La idea detrás de esto es que no se debería modificar algún registro más allá del que contiene el valor de retorno, para no tener efectos secundarios no deseados. Para ello, uno de los usos de la pila es permitir almacenar los registros que son usados en los procedimientos y su posterior restauración. El proceso es como sigue:

1. Generar un espacio en la pila para almacenar los valores de uno o más registros.
2. Almacenar los valores de los registros sobre la pila.
3. Ejecutar el procedimiento usando los registros.
4. Restaurar los valores originales de los registros desde la pila.
5. Desasignar el espacio sobre la pila.

Dado que se requiere el uso de la pila para almacenar información adicional, revisaremos brevemente la gestión de la pila que hace MIPS.

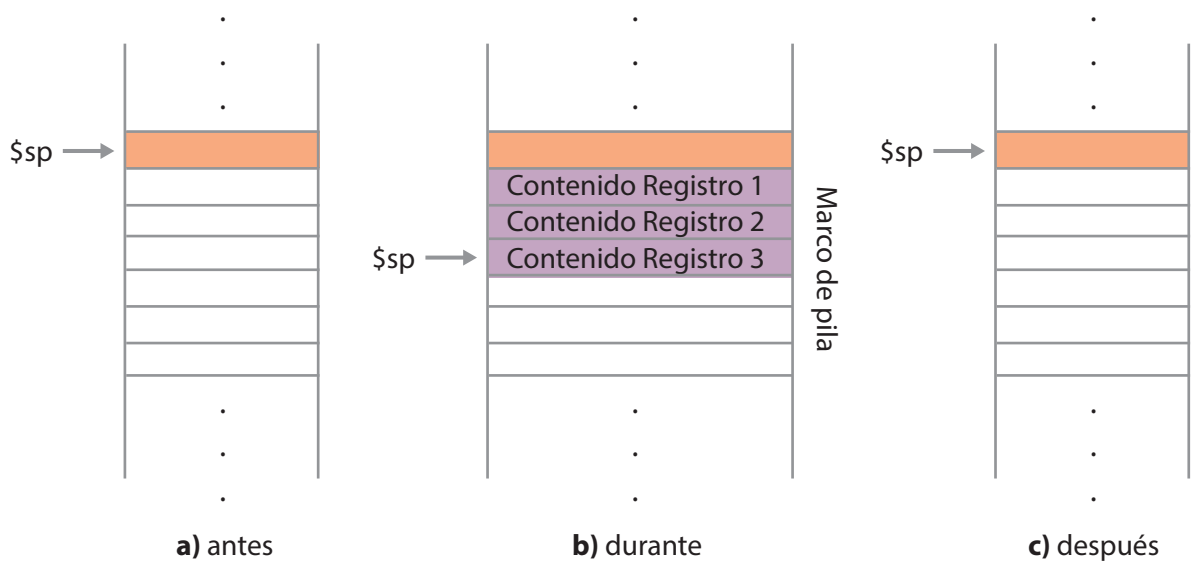


Figura 23. Los valores del puntero de pila y de la pila antes, durante y después del llamado a un procedimiento. Fuente: Elaboración propia.

En la Figura 23 podemos observar el estado del $sp antes, durante y después del llamado a un procedimiento. El llamado genera un espacio de memoria para almacenar los registros preservados que necesitará durante su ejecución. Esto lo realiza por decrementar el puntero de la pila en función del número de registros que necesite. Por ejemplo, si requiere almacenar 3 registros como en el caso de la Figura 23, el decremento será de 12 bytes (recordemos que MIPS usa una pila de 32 bits, lo que equivale a 4 bytes). Con ello, como se ve en la Figura 23.b, el $sp cambia su posición inicial para apuntar a la dirección de memoria en donde se almacenará al registro 3. El espacio de memoria, asignado para sí mismo por el procedimiento llamado se denomina *marco de pila*. Luego de la ejecución del procedimiento llamado, se desasigna el espacio de pila ocupado, por lo que el $sp vuelve a su estado inicial incrementando su valor (ver Figura 23.c).

3.1.6. Procedimientos anidados y recursivos

Con anidados, nos referimos a la capacidad de los procedimientos de llamar a otras funciones e incluso de llamar a <<*copias*>> de sí mismos (recursividad) durante su ejecución. En este proceso es necesario ser cuidadoso con el manejo de los registros, de manera que no se produzca pérdida de la información almacenada en ellos cuando se invoca a otras funciones o a copias de sí mismo. Para ello, la función llamada almacena en la pila, copias de los registros que deben ser preservados. El $sp se ajusta para tener en cuenta la cantidad de registros colocados en la pila. En el retorno de la llamada, los registros son restaurados desde la memoria y el puntero de la pila es reajustado.

Para analizar el comportamiento de la pila y los diferentes registros en procedimientos anidados, veamos lo que sucede con una función que calcula el factorial de un número (David, A Patterson and John, 2005; Hamacher, 2012):

Nº	Código de alto nivel	Código en ensamblador MIPS
1	*int* factorial (int n)	0x90 fact : *addi* $sp, $sp, 8 # obtenemos espacio en la pila
2	{	0x94 *sw* $a0, 4($sp) # almacenamos $a0
3	if (n<=1)	0x98 *sw* $ra, 0($sp) # almacenamos $ra
4	return 1;	0x9C *addi* $t0, $zero, 2 # $t0=2
5		0xA0 *slt* $t0, $a0, $t0 # $a<= 1
6		0xA4 *beq* $t0, $zero, else # no: ir a else
7		0xA8 *addi* $v0, $zero, 1 # si: retornar 1
8		0xAC *addi* $sp, $sp, 8 # restaurar $p
9		0xB0 *jr* $ra # retornar
10		0xB4 else: *addi* $a0, $a0, -1 # n= n - 1
11	else	0xB8 *jal* fact # llamada recursiva
12	return (n*factorial (n-1);	0xBC *lw* $ra, 0($sp) # restauramos $ra
13	}	0xC0 *lw* $a0, 4($sp) # restauramos $a0
		0xC4 *addi* $sp, $sp, 8 # restauramos $sp
		0xC8 *mul* $v0, $a0, $v0 # n* factorial (n – 1)
		0xCC *jr* $ra # retornamos

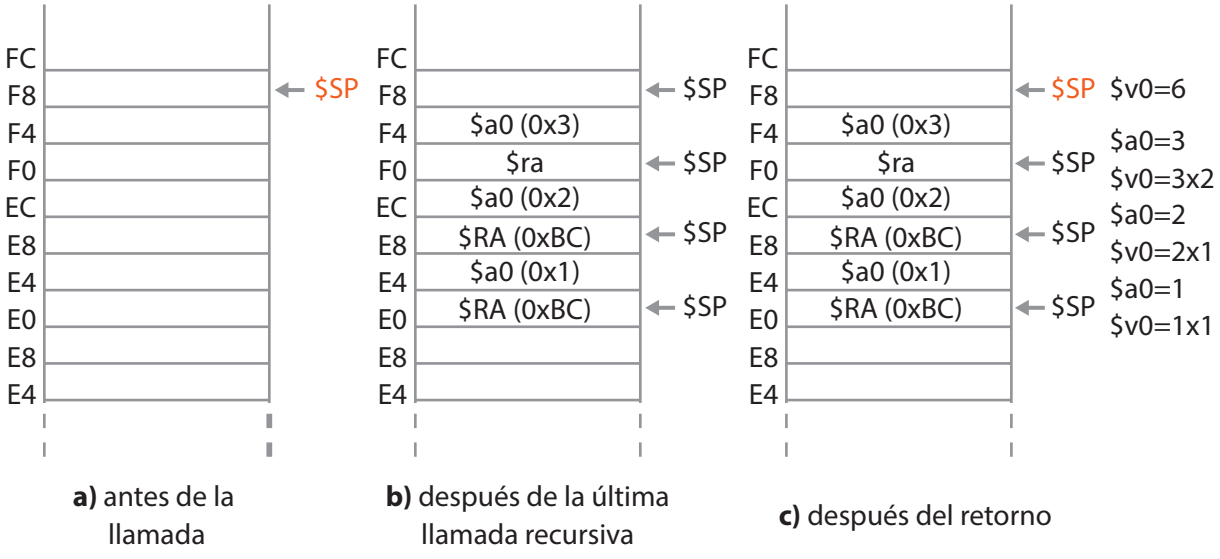


Figura 24. Los valores del puntero de pila y de la pila antes, durante y después del llamado al procedimiento factorial n=3. Fuente: Adaptado desde (Harris, David and Harris, 2010) .

Analicemos cómo es la asignación que se hace en pila para, por ejemplo, el factorial de 3. En la Figura 24, se muestra la asignación de los registros y los desplazamientos del puntero de pila cuando con el factorial de 3. Asumimos que $sp inicialmente apunta a 0xFC, la función crea un marco de trama de dos palabras para almacenar a $a0 y $ra, en la primera llamada, el procedimiento guarda $a0 (conteniendo n=3) en 0xF8 y $ra en 0xF4. Luego de ello, factorial cambia $a0 a n=2 y llama recursivamente a factorial (2). Obsérvese en el código ensamblador anterior que la dirección siguiente a este proceso es 0XBC; por ello se carga $ra con ese valor. En la segunda invocación de factorial, se almacena el registro $a0 (con n=2) en la dirección 0xF0 y $ra en 0xEC. En el siguiente llamado recursivo, factorial (1), se guarda $a0 en 0xE8, con el valor de n=1; y $ra en 0xBC. Esta tercera invocación retorna el valor $v0=1 y desasigna el marco de trama anterior retornando a la segunda invocación. En esta segunda invocación se restaura n=2 y el valor de $ra a 0x8C, se desasigna el marco de trama y se retorna $v0=2x1 a la primera invocación. Finalmente, en esta invocación se restaura n=3 y $ra a la dirección de retorno de la función llamante. Se retorna el valor de $v0=3x2=6. El $sp es restaurado a su posición original, 0xFC.

3.1.7. Argumentos adicionales y variables locales

Los lenguajes de alto nivel, como es el caso de C, tienen dos clases de almacenaje: automático y estático. Las automáticas son las variables locales a un procedimiento y por ello se descartan cuando se sale de este. Por su parte, las estáticas, conocidas también como variables globales, se mantienen a través del programa, dentro y fuera de los procedimientos. Para acceder a estas variables estáticas, MIPS reserva un registro denominado *puntero global*: $gp.

Cuando en los procedimientos se utiliza un número superior a cuatro argumentos y variables locales, la pila es usada para almacenar esas variables temporales. Los cuatro primeros argumentos son ubicados sobre los registros de argumentos y los restantes son ubicados en la pila, sobre el $sp. La función llamante debe expandir su espacio de pila para almacenar esas variables. En el caso de las variables locales, estas son almacenadas en los registros guardados $s0-$s7, si existe un número superior, estos son almacenados en la pila. Lo mismo ocurres con vectores locales.

En adición, es necesario espacio de memoria para las variables estáticas y estructuras dinámicas de datos. La Figura 25, muestra la convención de MIPS para el manejo de memoria. La parte inferior de la memoria está reservada para cuestiones del sistema operativo y no puede ser usado directamente por el programa. Luego se tiene un espacio en donde ubica el programa en lenguaje de máquina, este espacio es llamado tradicionalmente como el *segmento de texto*. Sobre este segmento, tenemos el espacio para los datos estáticos. A continuación tenemos el espacio reservado para los datos dinámicos, los cuales crecen en forma incremental (conocido como <<*heap*>>). Como se puede observar, la pila inicia en la cabecera de la memoria y tiene un avance decreciente.

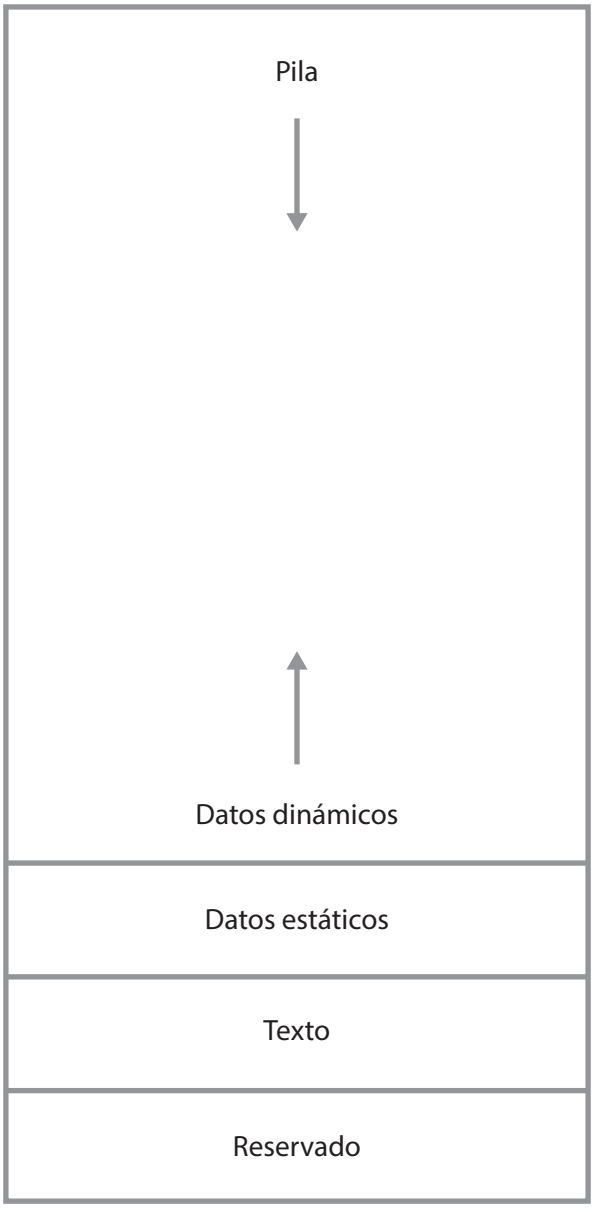


Figura 25. Asignación de memoria de MIPS para programs y datos. Fuente: Adaptado desde (David, A Patterson and John, 2005).

3.2. Compilar, ensamblar y cargar

La figura 26 muestra el proceso requerido para traducir un programa desde un lenguaje de alto nivel al lenguaje de máquina y su ejecución. Los pasos son los siguientes:

1. El código de alto nivel es compilado en un código ensamblador.

2. El código ensamblador, por su parte, es ensamblado, en un archivo objeto. Un proceso intermedio antes de crear el ejecutable. Se tiene dos pasos: en primera instancia, el ensamblador asigna las direcciones de las instrucciones y encuentra todos los *símbolos*, tales como etiquetas y los nombres de las variables globales. Estos nombres de los símbolos son contenidos en una tabla *de símbolos*. Luego de este paso, el ensamblador produce el código en lenguaje de

máquina. Las direcciones de las variables globales y las etiquetas son tomadas desde la *tabla de símbolos*. El código de lenguaje de máquina y la *tabla de símbolos* son almacenadas en el *archivo objeto*.

3. El enlazador (del inglés <<*linker*>>), combina el código de máquina con el código objeto desde librerías y otros archivos para producir el programa ejecutable.

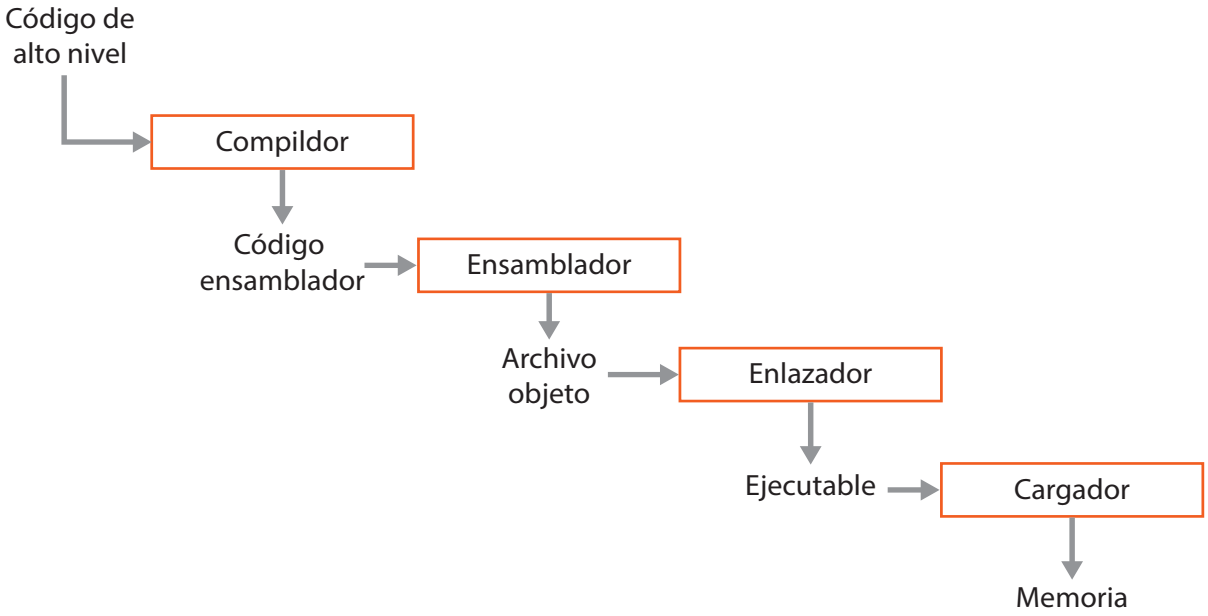


Figura 26. Pasos para traducir y empezar un programa. Fuente: Adaptado desde (Harris, David and Harris, 2010).

Tema 4.
Sistema de memoria

Los programas y datos con los que opera el procesador son mantenidos en la memoria. Esto hace que la velocidad con que se puedan acceder a ellos, sea de mucha importancia en su rendimiento. **Precisamente, uno de los principales cuellos de botella en la ejecución de las instrucciones se da en la velocidad de acceso a ellas, dado que actualmente la velocidad de procesamiento es superior a la de acceso**.

Una de las medidas de la velocidad de las unidades de memoria, conocida como el *tiempo de acceso a memoria*, es el tiempo transcurrido desde el inicio de una operación de transferencia de una palabra de datos hasta la finalización de esa operación. Otra medida, de mucha utilidad, es el *tiempo del ciclo de memoria*, el cual es el *retardo de tiempo mínimo* requerido entre dos operaciones de memoria consecutivas.

La memoria está organizada como una matriz bidimensional de celdas de memoria (ver Figura 27), donde se lee o escribe una de las filas de la matriz. Esta fila es especificada a través de una dirección. Con direcciones de longitud N, se tendrá 2^N filas y con datos de M bits, tendremos M columnas. Cada una de las filas, es una *palabra*. En este modelo, el número de filas se conoce como *profundidad*; mientras que el *ancho* está dado por el número de columnas. El tamaño de la memoria está dada por *profundidad* x *ancho*. Por ejemplo, 1024x32 es 32Kb. Lo que implica 1024 palabras de 32 bits.

Cada una de las celdas está conectadas a una *línea de palabra* (<<*wordline*>>) y una *línea de bit* (<<*bitline*>>). Para cada dirección, la memoria asegura que una sola línea de palabra activa las celdas en esa fila. Las *líneas de palabra* y *de bit*, son utilizadas para los procesos de lectura y escritura.

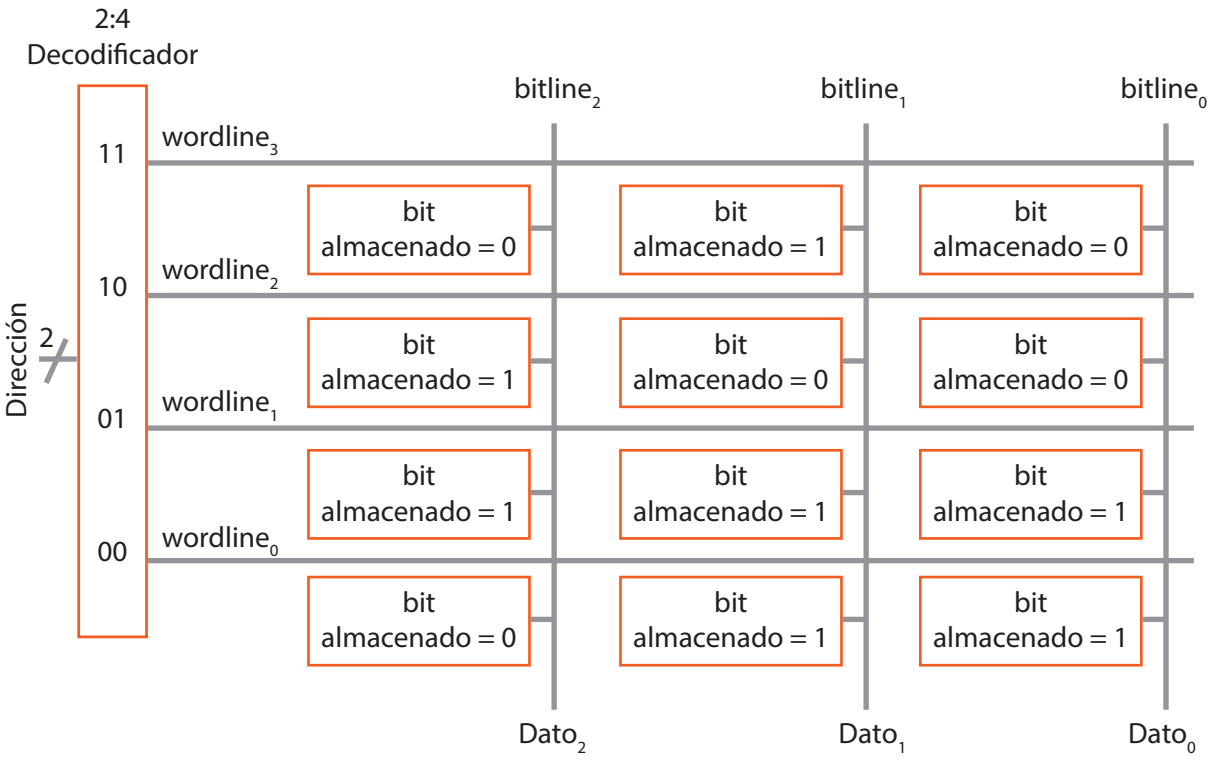


Figura 27. Matriz de memoria 4x3. Fuente: Adaptado desde (Harris, David and Harris, 2010).

A más de ello, las memorias tienen uno o más puertos. Cada puerto da acceso de lectura o escritura a una dirección de memoria. Las memorias de múltiple-puerto pueden acceder en forma simultánea a varias direcciones.

4.1. Tipos de memorias

Las memorias, en general, son clasificadas conforme almacenan los bits en las celdas. Desde una visión amplia tenemos la *memoria de acceso aleatorio* (RAM, del inglés <<*random access memory*>>), la cual es volátil y, por tanto, pierde sus datos cuando no está energizada; y la memoria de sólo lectura (ROM, del inglés <<*read only memory*>>), no volátil, por lo que mantiene su información, aún sin una fuente de poder.

El nombre de RAM y ROM pueden llevar a confusión, dado que en un principio, las memorias ROM solo eran de lectura y no podían ser escritas y además no podían ser accedidas en forma aleatoria; sin embargo, en la actualidad, este tipo de memoria pueden ser leídas y escritas, y además accedidas en forma aleatoria. Por lo tanto, la principal diferencia entre estos dos tipos de memoria se da en que la RAM es volátil, mientras que la ROM es no volátil.

Dentro de las memorias RAM tenemos dos clasificaciones, las memorias dinámicas (DRAM, del inglés <<*dynamic RAM*>>) y las estáticas (SRAM, del inglés <<*static RAM*>>). En el primer caso, la memoria DRAM almacena un bit como presencia o ausencia de carga en un capacitor. Los valores de datos en este tipo de memoria deben ser constantemente refrescados para no perder su información, dado las características de los capacitores. Por su parte, la memoria SRAM es estática dado que los bits almacenados no necesitan ser refrescados. El bit de datos es almacenado en inversores con acoplamiento cruzado, lo que le permite restaurar sus valores en caso de que sean degradados por cuestiones de ruido. En cuanto al área que requieren en su implementación y el nivel de latencia, la SRAM necesita mayor número de transistores (seis) en comparación que las DRAM (uno) o los flip-flops (aproximadamente veinte). No obstante, la SRAM tiene mucha menor latencia que las DRAM, debido a que está última depende de la carga del condensador y además presenta un menor <<*throughput*>> dado su necesidad de constante refresco de los datos.

Con respecto a la memoria ROM, esta almacena un bit como presencia o ausencia de un transistor. El contenido de la memoria es especificada durante su manufacturación. Una ROM *programable* (PROM, del inglés <<*programmable ROM*>>) ubica transistores en cada celda, pero provee una forma de conectar o desconectar estos a tierra. Uno de las formas es a través de fusibles, donde un fusible permite determinar si conectamos o desconectamos un transistor de tierra y por tanto la celda almacena un 0 o un 1. Este tipo de ROM es también denominada como ROM programable una sola vez.

Otro tipo de ROM son las reprogramables. Entre ellas tenemos las EPROMs (del inglés <<*Erasable PROMs*>>), conformado por transistores de puerta flotante. Cuando se aplica un valor de voltaje alto apropiado se programa la celda a cero; mientras que exponiendo la memoria a la luz ultravioleta por un periodo de tiempo determinado, hace que la memoria se borre y vuelva a su estado de todas sus celdas en uno. Las EEPROMs (del inglés <<*Electrically erasable PROMs*>>) y las memorias Flash incluyen circuitería sobre el chip para borrar y programar, lo que hace innecesario el uso de luz ultravioleta. En el caso de las EEPROMs, los bits de celdas son borrables en forma individual; mientras que las memorias Flash, borran grandes bloques de bits, siendo más baratas porque se necesita menos circuitería de borrado.

4.2. Jerarquía de memoria

La memoria ideal debería cumplir con la característica de tener una gran capacidad de almacenamiento, rápida en su acceso y no costosa. Sin embargo, como hemos visto en las diferentes tecnologías empleadas hasta el momento en la construcción de las memorias, esto no es posible de cumplir y por lo tanto debe existe un compromiso entre estos tres factores. Por tanto, diferentes clases de memorias son usadas en el computador, en función de los requerimientos. De esta forma, la memoria del computador puede ser visto en forma jerárquica.

Dos principios son útiles al momento de establecer la jerarquía de memoria: *localidad temporal* y espacial. En el primer caso se establece que si se hace referencia a una localización de datos, entonces hay una mayor tendencia a que sea referenciada próximamente. Con respecto a la localidad espacial, nos dice que si se hace referencia a una localización de datos, las localidades cercanas tenderán a ser referenciadas pronto.

La jerarquía de memoria consiste en organizar la estructura de la memoria en múltiples niveles, con diferentes velocidades y anchos. Lo que se busca es proveer una mayor capacidad de memoria, con la tecnología más barata, a la vez brindar una mayor velocidad que ofrece la memoria más rápida.

En cuanto a los datos, tenemos un nivel de jerarquización similar, aquellos más cercanos al procesador, generalmente el nivel más cercano al procesador es un subconjunto de algún nivel más lejano, donde todos los datos se almacenan en el nivel más bajo. Si nos basamos en este primer modelo de gestión de los datos, podemos observar la jerarquía de memoria desde dos niveles amplios: un nivel superior, cercano al procesador, más pequeño y rápido que el nivel inferior, dado que usa tecnologías que son más costosas.

La unidad mínima de información que puede estar presente en la jerarquía de nivel dos es llamada como *bloque* o *línea*. Si el dato requerido por el procesador aparece en el nivel superior, este es llamado un *acierto* (<<del inglés *hit*>>). Si por el contrario, no es encontrado en el nivel superior es llamado un desacierto (<<del inglés *miss*>>). En ese caso, el dato requerido es obtenido desde la memoria de nivel inferior. La *tasa de aciertos* es la fracción de accesos de memoria encontrados en el nivel superior; mientras que, la *tasa de desaciertos* es la fracción de accesos de memoria no encontrados en el nivel superior. Por su parte, el *tiempo de acierto* es el tiempo de acceso al nivel superior de la jerarquía de memoria, lo que incluye el tiempo necesario para determinar si el acceso es un acierto o no. La *penalización por desacierto* es el tiempo para reemplazar un bloque en el nivel superior por el correspondiente bloque desde el nivel inferior, más el tiempo para entregar este bloque al procesador.

$$\textit{Tasa de aciertos} = \frac{\textit{(Número de aciertos)}}{\textit{(Número total de accesos a la memoria)}}$$

$$\textit{Tasa de desaciertos=1-Tasa de aciertos}$$

En la Figura 28 se muestra que la jerarquía de memoria usa tecnologías de memoria más rápidas y más pequeñas, cercanas al procesador; mientras que el nivel siguiente contiene capacidades superiores pero mucho más lento.

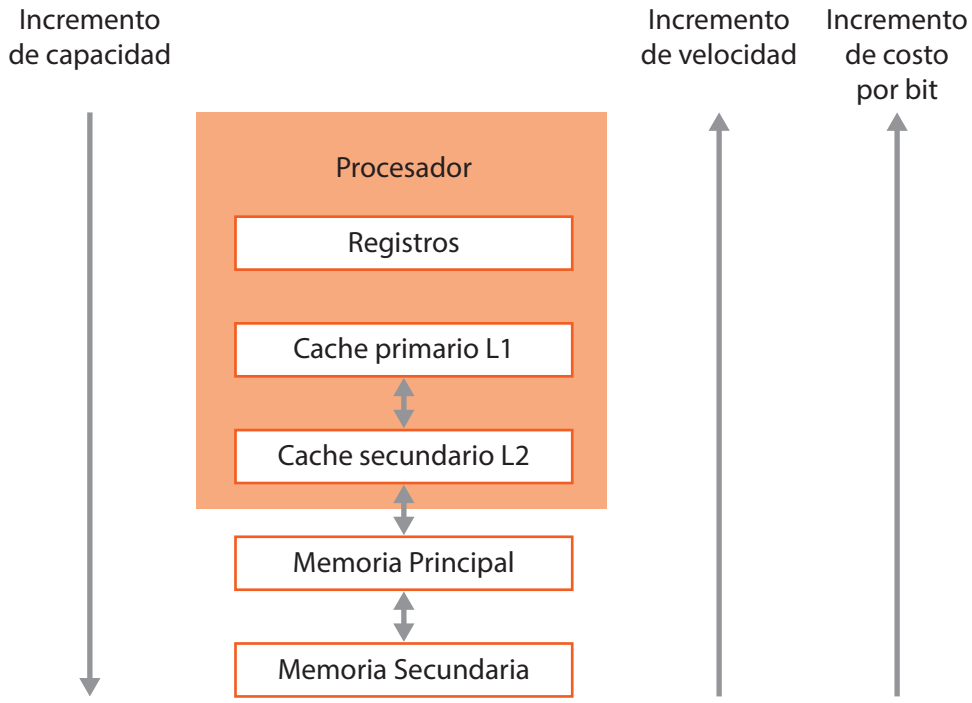


Figura 28. Estructura de una jerarquía de memoria. Fuente: Adaptado desde (Hamacher, 2012).

4.3. Memoria caché

La **memoria caché** es un espacio de memoria pequeño y rápido, intermedio entre la memoria principal y el procesador. Lo que se busca es agilitar el proceso de acceso a la información (datos o instrucciones) que requiere el procesador desde la memoria principal. Para ello, se aprovecha el principio de localidad temporal y espacial. Los programas muestran una estructura que hace uso de instrucciones y datos que son usados en forma repetida durante algún tiempo (lazos, llamadas a funciones, llamadas anidadas,…). Esta realidad puede ser aprovechada para mejorar la velocidad de acceso a esas instrucciones. De esta forma, la localidad temporal sugiere que cualquier información que sea necesitada por primera vez, debe ser llevada a chache dado que es probable que sea requerida pronto. Pero además, la localidad espacial, establece que, a más de esa instrucción, es útil llevar varios ítems que estén localizados en la cercanía. A este grupo de ítems contiguos se le conoce como un *bloque de caché* o una *línea de caché*.

De esta forma, la caché está constituida en *bloques* o *líneas de caché*. Cada una de estas *líneas* contiene un conjunto *k* de *palabras*, más una *etiqueta* de unos cuantos bits, que le permite identificar a cada *línea* qué bloque particular de la memoria almacena. Esta *etiqueta*, generalmente, tienen correspondencia con un conjunto de los bits más significativos de la dirección de memoria principal.

Dado que el espacio de memoria de almacenamiento de la caché es menor que el de la memoria principal, es necesario establecer una correspondencia entre los bloques de ambas memorias; para ello se especifica una *función de correspondencia*. De igual forma, cuando se requiere insertar un

nuevo bloque, es necesario establecer qué *línea* debe ser removida para crear el espacio necesario. Para ello, se tiene un *algoritmo de reemplazo de la caché*.

La presencia de la caché debe ser transparente al procesador. Este hace sus requerimientos de lectura o escritura con base en las direcciones de memoria. El circuito controlador de la caché es el responsable de analizar si el requerimiento se encuentra en esta memoria. Si es así, la operación de lectura o escritura es desarrollada en la localización de la caché apropiada. En este caso, se dice que un *acierto* de lectura o escritura ha ocurrido. Para un *acierto* de lectura, la memoria principal no está envuelta. Sin embargo, para un acierto de escritura, existen dos alternativas. En la primera, conocido como *protocolo de escritura directa*, tanto la posición de memoria principal como la de la caché son actualizados. En el segundo formato, solo la memoria caché es actualizada y se marca el bloque que contiene el ítem modificado a través de una bandera especial, conocido como *bit modificado*. La memoria principal es actualizada posteriormente. Solo cuando se va a eliminar el bloque modificado para dar paso a un nuevo bloque, se procede con la actualización de la memoria principal. Esta técnica es conocida como *protocolo de reescritura* o *protocolo de copia de respaldo*. Esta última es la técnica más usada para tomar ventaja de la velocidad con la que los bloques de datos pueden ser transferidos a los chips de memoria (David, A Patterson and John, 2005; Hamacher, 2012).

Los *desaciertos* de lectura implican que el bloque de *palabras* conteniendo la *palabra* requerida sea copiado desde la memoria principal en la caché. Luego de ello, la palabra requerida es remitida al procesador. Una alternativa es que la palabra solicitada sea enviada al procesador tan pronto como es leída desde la memoria principal (carga directa o reinicio temprano), lo que reduce el tiempo de espera del procesador; sin embargo, se requiere una circuitería más cara. En el caso de un *desacierto* de escritura, el uso del *protocolo de escritura directo* hace que la información sea escrita directamente en la memoria principal. En el caso del *protocolo de reescritura*, el bloque conteniendo la *palabra* direccionada es traído a la caché y entonces la palabra direccionada es sobrescrita con la nueva información.

4.3.1. Función de correspondencia

Existen varios métodos para determinar cómo los bloques de la memoria principal son ubicados en el caché. Para analizar algunos de ellos, haremos uso del ejemplo mostrado en la Figura 29.

Se tiene una memoria principal que se gestiona a través de una dirección de 16 bits; por lo que tiene 64K *palabras*, con 4K bloques de 16 *palabras* cada uno. Por su parte, el caché está constituido por 128 bloques de 16 *palabras* cada uno, con un total de 2048 (2K) *palabras*.

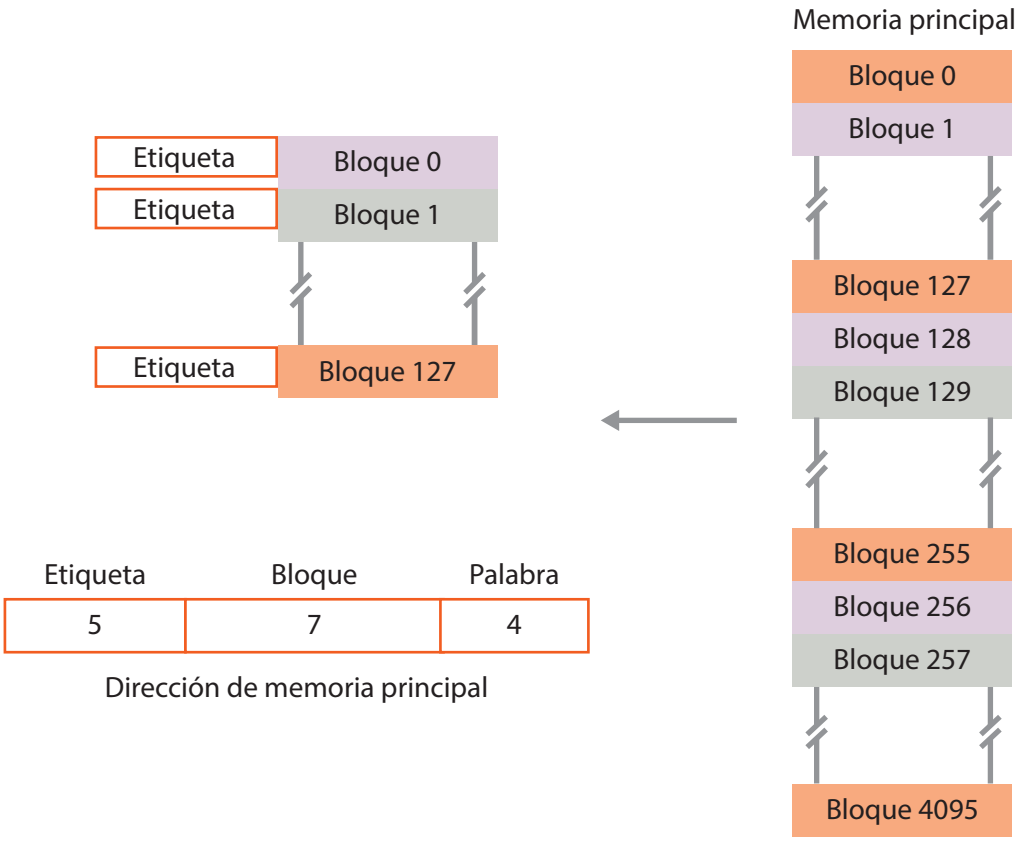


Figura 29. Correspondencia directa de caché. Fuente: Adaptado desde (Hamacher, 2012).

- **Correspondencia directa**. Para nuestro caso de estudio, un bloque j desde la memoria principal, se mapea sobre el bloque j módulo 128 de la caché. Así, los bloques de la memoria principal 0, 128, 256,… son mapeados al bloque 0 de la memoria caché. Los bloques 1, 129, 257,… son mapeados al bloque 1, y así sucesivamente. Como se puede observar, existe la posibilidad de que dos bloques de memorias sean mapeados a la misma posición de la caché. En ese caso, la contención se resuelve por permitir al nuevo bloque sobrescribir el bloque residente actual. De esta forma la correspondencia se expresa como:

$$i=j \text{ módulo } m$$

Donde:

i= número de línea de caché

j= número de bloque de memoria principal

m= número de líneas en la caché

Como se muestra en la Figura 29, la dirección de memoria es dividida en tres campos: *etiqueta*, *bloque* y *palabra*. Los cuatro bits menos significativos seleccionan una de las 16 *palabras* del bloque. Los siguientes 7 bit, *bloque*, determinan la posición del caché en la cual debe ser almacenado. Finalmente los últimos 5 bits más significativos, permiten etiquetar

las posiciones de memoria, identificando 32 bloques de la memoria principal que han sido mapeados dentro de la memoria de caché. Su rol es permitir identificar si una dirección de memoria solicitada se encuentra dentro de los bloques de memoria de la caché, comparando los 5 bits más significativos de la dirección con la etiqueta de los bloques de caché. En este tipo de asignación, un bloque de la memoria principal solo puede estar en un único bloque de la memoria caché. Lo cual a su vez es su mayor desventaja al convertirla en poco flexible.

- Correspondencia asociativa. A diferencia de la función de correspondencia directa, en el mapeo asociativo permite que un bloque de memoria principal pueda ser ubicado en cualquier línea de memoria de la caché. La dirección de memoria principal es dividida en dos bloques: etiqueta y palabra.

Etiqueta (s bits)	Palabra (w bits)

Número de bloques en memoria principal: 2^s

Tamaño de bloque: 2^w

Número de líneas en caché: no determinado

La etiqueta identifica un bloque de la memoria principal. Los bits de la etiqueta de la dirección recibida desde el procesador es comparada con los bits de las etiquetas de cada uno de los bloques de la caché para ver si el bloque deseado está presente. De esta forma, cuando un nuevo bloque de memoria es traído hacia el caché, se procede a reemplazar algún bloque existente, solo si la caché está llena. En ese caso se requiere de un algoritmo de reemplazo. Hay mayor complejidad en este modelo de mapeo por la necesidad de búsqueda de todas las etiquetas. A más de ello, se tiene un mayor retardo. Para evitar esto, se desarrolla búsquedas en paralelo, conocido como *búsqueda asociativa*. Precisamente, la complejidad de este tipo de búsquedas es su mayor limitación.

- **Correspondencia asociativa por conjuntos**. En este proceso de correspondencia se usa una combinación de las formas directa y por asociación. Las líneas de caché son agrupadas en conjuntos, de forma tal que la correspondencia permite a un bloque de la memoria principal residir en algún bloque de un conjunto específico. De esta forma, se mejora la limitación de flexibilidad del método directo, permitiendo un pequeño conjunto de bloques para su selección; y, además, se disminuye la complejidad de la búsqueda asociativa del segundo método, disminuyendo el espacio de búsqueda asociado.

  La caché se divide en *v* conjuntos, cada uno con *k* líneas. De esta forma tenemos:

$$m=v \times k$$

$$i=j \text{ módulo } v$$

Donde:

i = número de conjunto de caché

j = número de bloque de memoria principal

m = número de líneas de la caché

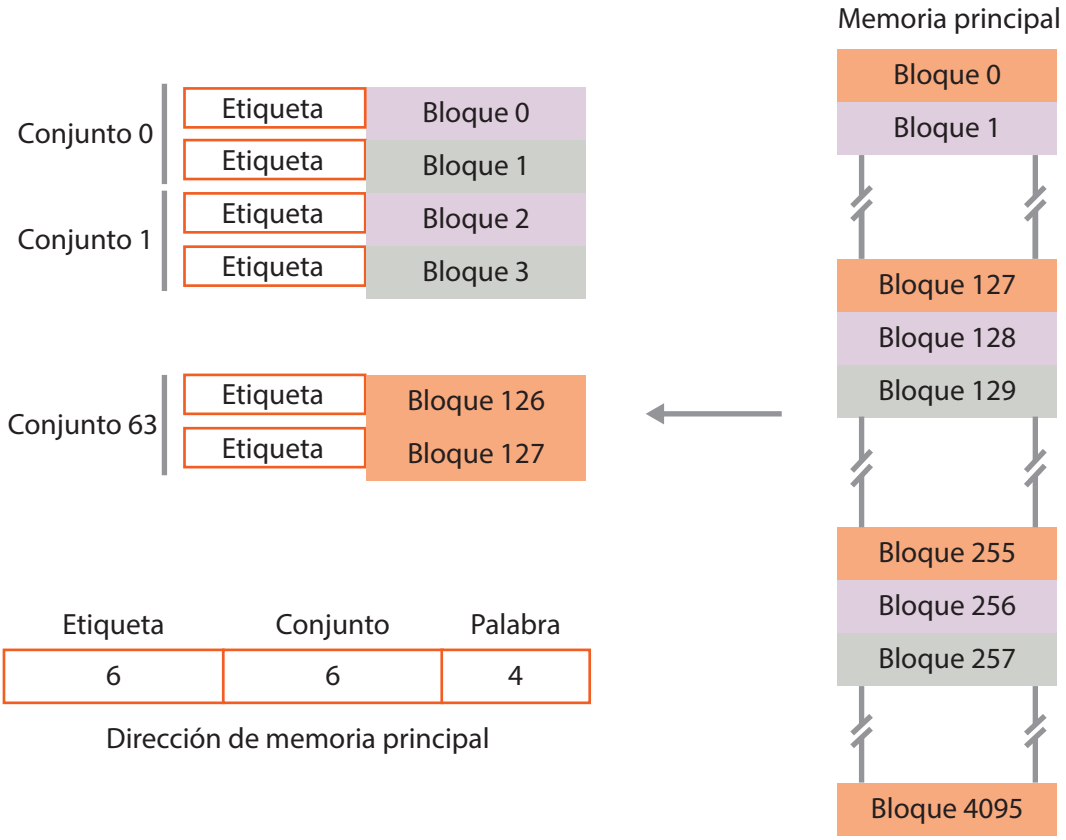


Figura 30. Correspondencia asociada por conjuntos. Fuente: Adaptado desde (Hamacher, 2012).

En la Figura 30 se puede observar el mismo caso ejemplo que usamos en nuestra explicación de la correspondencia directa, con una memoria principal de 4096 bloques y la caché de 128 líneas. En este caso, la dirección de memoria es dividida en tres partes: *etiqueta*, *conjunto* y *palabra*. Obsérvese que en el caso de los bloques de memoria 0, 64, 128,…, 4032 se corresponden con el conjunto 0, pudiendo ocupar una de las dos líneas de ese conjunto. El campo de la etiqueta de la dirección, debe entonces ser comparado asociativamente con las etiquetas de los bloques de los conjuntos para comprobar si el bloque deseado está presente.

Cuando se enciende por primera vez el computador, el caché contiene datos que no son válidos. Para ello, se hace uso de un bit de control, denominado bit de validación (del inglés <<*valid bit*>>), el cual debe ser provisto a cada bloque para indicar si los datos en aquel bloque son válidos o no. Este bit es puesto a 1 cuando contiene información válida y 0 en caso contrario. De esta forma, permite asegurar que el procesador no obtendrá datos obsoletos desde la caché.

4.3.2. Algoritmo de reemplazo

Cuando la memoria caché está llena, es necesario establecer procesos para sustituir una de las líneas por el bloque entrante. En el caso de correspondencia directa, solo hay una línea para cada bloque, por lo que el reemplazo es directo. Para las otras técnicas de correspondencia se requiere el uso de algoritmos.

El objetivo es mantener bloques en la caché que probablemente sean referenciados en un futuro cercano. Una de las estrategias usadas en la denominada <<*utilizado menos recientemente*>> (LRU, del inglés <<*least recently used*>>): se sustituye el bloque que se ha mantenido por más tiempo en la caché sin haber sido referenciado. Para ello, el controlador del caché debe hacer un seguimiento a las referencias de todos los bloques a medida que avanzan los cálculos. Se requiere un contador que permita registrar el nivel de uso de los bloques. Por ejemplo, en el caso de correspondencia asociativa, cuando se da un acierto, el bloque correspondiente se fija a 0, mientras que aquellos que tienen un valor inferior a este, son incrementados; mientras que los restantes son mantenidos. En caso de un desacierto, y si la caché no está llena, el contador del nuevo bloque es establecido a 0; mientras que el resto es incrementado en uno. En caso de que la memoria caché esté llena, el bloque reemplazado es aquel que tenga el valor más alto.

Otra posibilidad es el denominado <<*primero en entrar, primero en salir*>> (FIFO, del inglés <<*first in, first out*>>): se sustituye aquel bloque del conjunto que ha estado más tiempo en el caché. Otra posibilidad es la del <<*utilizado menos frecuentemente*>> (LFU, del inglés <<*least frequently used*>>): se sustituye aquel bloque del conjunto que ha experimentado menos referencias.

4.3.3. Número de cachés

Actualmente, es normal el uso de múltiples caché. Generalmente, hoy en día, en los procesadores de alto rendimiento, se usa dos niveles de caché: un nivel L1 para instrucciones y datos y un nivel de caché L2 de mayor tamaño, con lo cual se asegura una mayor tasa de aciertos. El nivel L1 es mucho más rápido que el nivel L2. La inclusión de un segundo nivel, además, reduce el impacto de la velocidad de la memoria principal sobre el rendimiento del computador. Recientemente, dado la creciente disponibilidad de superficie para caché en el propio chip, los microprocesadores han incorporado un tercer nivel L3 en el chip. Al parecer, este tercer nivel supone una mejora en las prestaciones del computador (Stallings, 2013).

4.4. Memoria virtual

Generalmente, el espacio de direcciones del procesador es mucho mayor que lo que ofrece la memoria principal. Por ejemplo, un procesador de 32 bits puede direccionar hasta 4G bytes. No obstante, el ancho típico de la memoria principal de un computador con un procesador de 32 bits es del orden de 1G a 4G bytes. De esta forma, para aquellos programas que no encajan completamente en la memoria principal, la parte que no está siendo usada actualmente es ubicada en la memoria secundaria. Cuando se las requiera en el proceso de ejecución se las traslada a la memoria principal, a través de un proceso desarrollado en forma automática por el sistema operativo. El esquema empleado es conocido como *memoria virtual*. Todo este proceso es transparente para las aplicaciones de programación.

Las direcciones de memoria emitidas por el procesador son denominadas como *direcciones lógicas o virtuales*. Esas direcciones son trasladadas a direcciones físicas a través de una combinación de acciones hardware y software. La Figura 31 muestra una organización de la memoria virtual. En ella se presenta la unidad de gestión de memoria (MMU, del inglés <<*Memory Management Unit*>>), la cual mantiene un seguimiento de cuales partes del espacio de direcciones virtuales están en la memoria física. Cuando el dato o instrucción deseada se encuentre en la memoria física, el MMU traslada la dirección virtual en su correspondiente física. En caso de no ser así, el MMU causa que el sistema operativo transfiera el dato desde el disco a la memoria. La transferencia se desarrolla a través del esquema de acceso directo a memoria DMA (la cual será analizada en el tema 5).

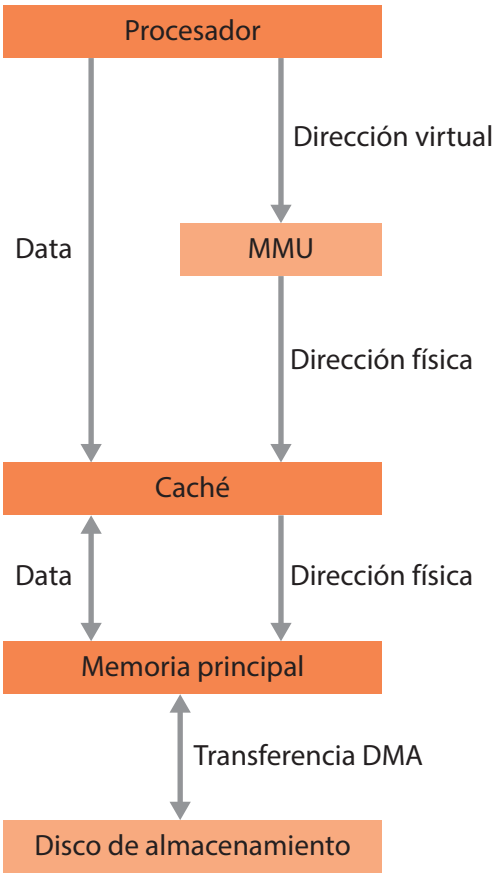


Figura 31. Organización de la memoria virtual. Fuente: Adaptado desde (Hamacher, 2012).

4.4.1. Traducción de direcciones

En la traducción de direcciones virtuales a direcciones físicas, se asume que todos los programas y datos están compuestos por unidades de longitud fija, llamadas *páginas*. Cada una de estas páginas consiste de un bloque de palabras que ocupan ubicaciones contiguas en la memoria principal, con un ancho que está entre 2K y 16K bytes. Esta es la unidad básica de traslado de información desde el disco magnético a la memoria principal.

Cada una de las direcciones virtuales generadas por el procesador para la recuperación de instrucciones o el desarrollo de operaciones de carga/almacenamiento de datos, están constituidas

por un *número de página virtual* (los bits de orden superior) más un *desplazamiento* (bits de orden inferior), que especifican la ubicación de un byte (o palabra) particular dentro de una *página*.

Dirección virtual desde el procesador

Número de página virtual	Desplazamiento

La información sobre la ubicación de la memoria principal de cada *página* se encuentra en una *tabla de páginas*. Esta información incluye la dirección de la memoria principal donde se encuentra la *página* que es almacenada y su estado actual. Se denomina como *marco de página* al área de la memoria principal que puede contener una página. La dirección de inicio de la tabla de página es mantenida en un *registro base de tabla de páginas*. Esta dirección de inicio más el número de página virtual, proporcionado por la dirección virtual, nos permite conocer la dirección de la entrada de la tabla de página que contiene la dirección de inicio de la *página* (marco de página), si es que esta página actualmente reside en la memoria principal. Esta dirección junto con el desplazamiento conforma la dirección física en la memoria principal. En adición, cada una de las entradas de la tabla de páginas contiene bits de control que indican el estado de la página mientras permanece en la memoria principal.

La tabla de páginas debería estar ubicada dentro de la MMU; no obstante, debido a que su longitud puede ser muy grande y dado que el MMU se implementa como parte del chip del procesador, no es posible incorporar toda la tabla de páginas como parte del MMU. Por ello, una parte se mantiene dentro de MMU, con las entradas de las páginas más recientemente accedidas. Estas entradas son almacenadas en una tabla conocida como TBL, del inglés <<*Translation Lookaside Buffer*>>. Su funcionamiento es similar a la caché pero para la tabla de páginas en la memoria principal. Este mantiene copia de las entradas correspondientes presentes en la *tabla de páginas*.

4.5. Almacenamiento secundario

Los requisitos de grandes capacidades de almacenamiento en la mayoría de los sistemas de computación son cubiertos por discos magnéticos u ópticos, conocidos como dispositivos de almacenamiento secundario. Estos almacenamientos son de tipo no volátil, por lo que los datos que se desean mantener por un tiempo indefinido pueden ser mantenidos en este tipo de dispositivos.

4.5.1. Disco duro magnético

Este tipo de almacenamiento se fundamenta en el uso de superficies magnéticas y cabezales de lectura y escritura. Un solo dispositivo está constituido por un conjunto de platos —montados sobre un eje común—, cada uno con sus propias cabezas de lectura y escritura.

El almacenamiento se fundamenta en cintas magnéticas delgadas, depositadas por ambos lados de los platos. Estas superficies son divididas en pistas concéntricas, y cada una de ellas en sectores. Los datos son almacenados en forma serial en cada pista. Cada uno de los sectores puede contener 512 bytes o más. Cada uno de estos sectores contiene una *cabecera de sector* que contiene información de identificación usada para encontrar el sector sobre la pista seleccionada. Adicional a los datos, se tienen unos bits que constituyen un código corrector de errores (ECC, del inglés <<*error correcting*

code>>) que permite detectar y corregir errores que pueden ocurrir durante la lectura o escritura de los datos. Entre sectores y entre pistas, existen bandas vacías, que permiten a los circuitos de control minimizar los errores por des-alineamiento de los cabezales y distinguir entre dos sectores consecutivos, respectivamente. Los datos son accedidos por especificar el número de superficie, el número de pista y el número de sector.

El proceso de formatear escribe marcas que dividen el disco en pistas y sectores. Durante este proceso el controlador del disco puede determinar algunas o todas las pistas con defectos. Se mantiene un registro de estas y se las excluye de su uso. La información de formateo incluye: cabeceras de sectores, bits ECC y los espacios intersector.

El *tiempo de acceso*, comprendido como el retardo de tiempo entre que el disco recibe una dirección y el inicio real de la transferencia de datos, está compuesto por dos partes: el *tiempo de búsqueda*, que constituye el tiempo necesario para ubicar el cabezal de lectura/escritura en la pista correspondiente (entre 5 y 8 mseg.); y el *retardo rotacional*, llamado también tiempo de latencia, el cual es el tiempo requerido para alcanzar el sector direccionado después de que el cabezal se encuentra ubicado en la pista correcta.

Una de las formas para disminuir el tiempo de acceso es usar múltiples discos operando en paralelo. Este sistema de almacenamiento fue denominado como RAID (del inglés, <<Redundant Array of Independent Disck>>). Este sistema permite mejorar el rendimiento y la fiabilidad del sistema de almacenamiento.

Para ello, las operaciones de E/S pueden ser llevadas en paralelo si los datos requeridos en cada operación se encuentran distribuidos en los diferentes discos. Además, se puede ejecutar en paralelo una única petición de E/S si el bloque de datos al que se va a acceder está distribuido a lo largo de varios discos.

Varias configuraciones de RAID han sido diseñadas, con cada nivel en la jerarquía proveyendo diferentes características adicionales. Así, los datos son distribuidos en los discos de acuerdo a diferentes configuraciones y usando diferente niveles de redundancia y capacidad de recuperación frente a fallos. Se tiene 7 niveles de RAID, desde le nivel RAID 0 hasta el RAID 6, en los cuales se proporcionan diferentes sistemas de control de paridad y detección y control de errores.

Por ejemplo, en RAID 0, un simple archivo grande es almacenado en varias unidades de disco separadas, por dividirlo en archivos de menor tamaño. Cuando se hace una operación de lectura a ese archivo, todos los discos acceden a su porción de dato en paralelo. Dado que se trabajan en forma independiente, se mantiene un sistema de buffer que permite re-ensamblar toda la información extraída desde los discos antes de ser transferida. Uno de los problemas se da en que el retardo rotacional no se reduce, dado que los discos operan en forma independiente.

4.5.2. Discos ópticos

Este tipo de tecnología se basa en el uso de la luz láser en el proceso de lectura y escritura de datos en un soporte extraíble. Dado que este tipo de luz se puede enfocar a puntos muy pequeños, se hace uso

un rayo de láser dirigido a un disco giratorio con pequeñas hendiduras las cuales reflejan la luz hacia un fotodetector, el cual detecta los patrones binarios.

Se hace uso del hecho de que la combinación de dos haces de luz coherentes en fase produce una luz más brillante; mientras que desfasados 180 grados se cancelan entre sí. Esto es aprovechado para poder almacenar datos a través de la impresión de hoyos (pozos), que permiten desfasar la luz al pasar de un nivel a otro.

En la actualidad se tienen varias tecnologías de uso de láser que permiten una mayor capacidad de almacenamiento, que van desde los centenares de Mbytes, en el caso de los CDs, Gbytes con los DVD y decenas de Gbytes con los Blu-ray.

Tema 5.
Sistema de E/S

Una de las características de las computadoras es su capacidad de comunicación con otros dispositivos. Esto permite, por ejemplo, a una persona ingresar información a través de un teclado o recibir información visual, a través del monitor. Otros dispositivos, por su parte, pueden obtener información analógica, digitalizarla y remitirla a un computador para su procesamiento. En definitiva, los computadores deben tener la capacidad de intercambiar información con diversos tipos de dispositivos, en muy variados ambientes.

En este tema, analizaremos el acceso a los dispositivos de entrada/salida (E/S) del computador, desde la óptica del programador.

5.1. Acceso a los dispositivos de E/S

La comunicación al interior del computador se desarrolla mediante una red de interconexión. Esta red está constituida por circuitos que permiten la comunicación entre el procesador, la memoria y los dispositivos de E/S. La Figura 32 muestra, en forma esquemática, este concepto.

Figura 32. Esquema del sistema de computación junto con su red de interconexión. Fuente: Adaptado desde (Hamacher, 2012).

Al igual que el sistema de memoria, analizado en el tema anterior, los dispositivos de E/S pueden ser accedidos a través de direcciones. Para ello, las ubicaciones son usualmente implementadas mediante *flip-flops*, organizados en forma de registros, referidos como *registros de E/S*. Dado que el espacio de direcciones es compartido por la memoria y los dispositivos de E/S, este arreglo es conocido como *E/S asignada en memoria*, del inglés <<*memory-mapped I/O*>>. Con esta correspondencia de los dispositivos de E/S en la memoria, las instrucciones utilizadas para acceder a memoria, pueden ser también usadas para transferir datos desde y hacia los dispositivos de E/S. La ventaja de este tipo de acceso es su sencillez, dado que no se requiere implementar circuitería especial para los dispositivos de E/S. Son utilizados en procesadores RISC y en sistemas empotrados.

Un modelo de acceso diferente es el conocido como E/S aislado (del inglés <<*isolated I/O*>>). Se plantea el uso de direcciones diferentes para la memoria y para los dispositivos de E/S, cada uno con sus correspondientes instrucciones. Por ello, el procesador, en caso de tener un solo bus de direcciones, necesita una señal de control especial que le permite identificar si la dirección corresponde a un E/S o a memoria; otra posibilidad es usar un bus exclusivo para el direccionamiento de los dispositivos de E/S. Los procesadores x86 de Intel, hacen uso de este tipo de acceso.

Dado que nuestro enfoque, durante este curso, está en los procesadores RSIC, nos centraremos en los sistemas de acceso basado en E/S asignado por memoria.

En este proceso de acceso, la *interfaz del dispositivo* provee los medios requeridos para la transferencia de datos y el intercambio de información de estado y control necesarios para gobernar la operación del dispositivo. Esta interfaz incluye registros que son accesibles desde el procesador. Estos sirven para diferentes propósitos: buffer, mantener información del estado actual del dispositivo, almacenar información que controla su comportamiento,… Su acceso es a través de instrucciones de programa, como si fueran ubicaciones de memoria.

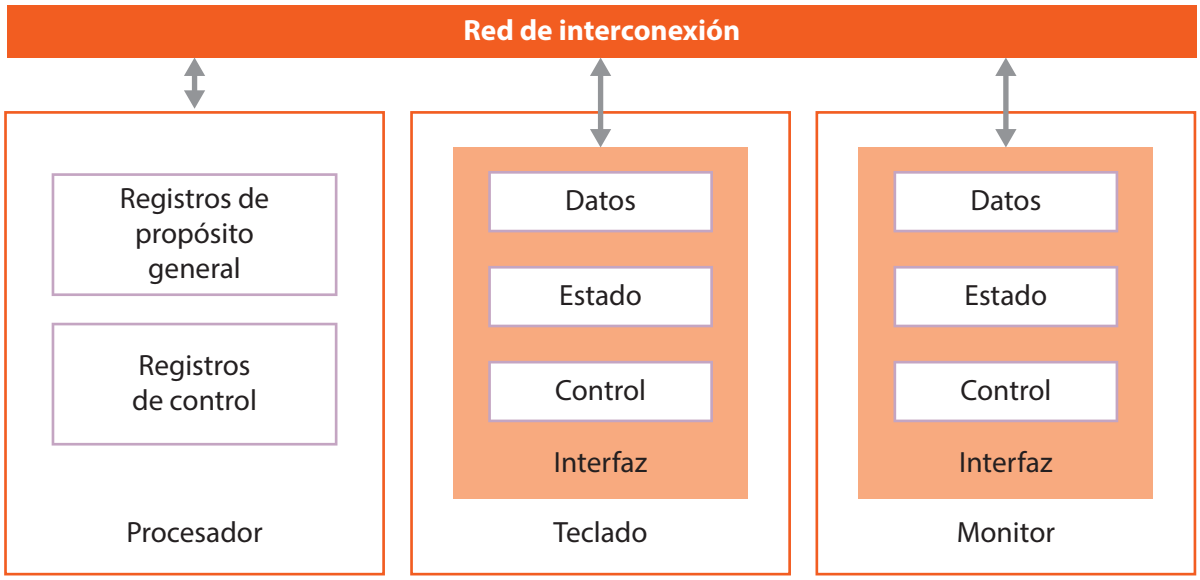


Figura 33. Conexión para el procesador, teclado y monitor. Fuente: Adaptado desde (Hamacher, 2012).

5.1.1. E/S controlado por programa

Este método permite, a través de la ejecución de un conjunto de instrucciones establecidas en un programa, desarrollar tareas en los que intervenga dispositivos de entrada salida. Por ejemplo: captar la información ingresada a través de un teclado, guardarla en memoria y mostrarla a través de un monitor.

Para ello, es necesario, adicionalmente, considerar que cada una de esas acciones deben ser desarrolladas en un tiempo adecuado. Dado que cada una de las tareas y dispositivos tienen distintas velocidades de ejecución —por ejemplo, el ingreso de información desde el teclado está limitado por la capacidad humana; sin embargo, la interfaz con el computador y, más aún, el procesador, puede desarrollar la transmisión y procesamiento de los datos a una velocidad muy superior—, es necesario establecer mecanismos para sincronizar la transferencia de datos entre los distintos elementos del computador.

Una solución es el uso de un protocolo de señalización. Se puede implementar un sistema con notificación. Por ejemplo, en el caso de que se necesite mostrar información textual en el monitor, el procesador envía el primer carácter y espera una señal desde el monitor que le indique que el siguiente carácter puede ser enviado. La misma forma de actuar puede ser usada para el caso de la recepción de información desde un teclado. Así, el procesador espera por una señal que le indique que una tecla ha sido pulsada y que la información se encuentra en un registro de E/S asociado con el teclado. Esto es, se activa un bit (bandera) en el registro de estado asociado al teclado que indica la presencia de datos. El procesador accede a este registro y evalúa el estado de este registro, y en especial de esta bandera, para saber si existen datos disponibles. Cuando el procesador actúa de esta forma, se dice que el procesador *sondea* (<<*polls*>>) *el dispositivo de E/S*. En este caso, el procesador debe ejecutar instrucciones de máquina que verifiquen las banderas de los registros de estado y transfieran los datos entre el procesador y los dispositivos de E/S.

Si los registros en las interfaces de E/S son accedidas como si ellas fueran localizaciones de memoria, cada uno debe ser asignado una dirección que será reconocida por el circuito de la interfaz.

En este formato de acceso de E/S controlado por el programa, el procesador debe gastar tiempo en la espera en el proceso de sincronización de los dispositivos, de manera que la información sea accesible o que el dispositivo se encuentre en condiciones de enviar o recibir información o de actuar. Para evitar este tipo de esperas, se utiliza el concepto de *interrupciones*.

5.2. Interrupciones

En el acceso controlado por programa, el procesador entra en lazos de espera, durante los cuales sondea los registros de estado del dispositivo E/S. Sin embargo, este tiempo podría ser utilizado por el procesador para desarrollar alguna tarea útil para el computador. Para ello, sería conveniente que el dispositivo de E/S alerte al procesador cuando esté listo; lo cual puede ser hecho a través de una señal de hardware, denominada *requerimiento de interrupción*, enviada al procesador.

La subrutina ejecutada en respuesta a una interrupción es denominada como rutina de servicio a una interrupción. La interrupción se ejecuta de forma muy similar a las llamadas a procesos. Por ejemplo, si el procesador se encuentra ejecutando una instrucción *i,* el procesador primero ejecuta esa instrucción y luego carga el contador de programa (PC) con la dirección de la primera instrucción de la *rutina de servicio a la interrupción*. Luego de la ejecución de esta rutina, se retorna a la instrucción *i+1* del programa que se estaba ejecutando antes de la interrupción (ver Figura 34). De esta forma, antes de pasar al servicio de la interrupción, el contenido del PC debe ser almacenado, de manera que cuando se produzca el retorno pueda ser reestablecido su valor. Esta dirección de retorno debe ser almacenada o en un registro de propósito general o en la pila del procesador.

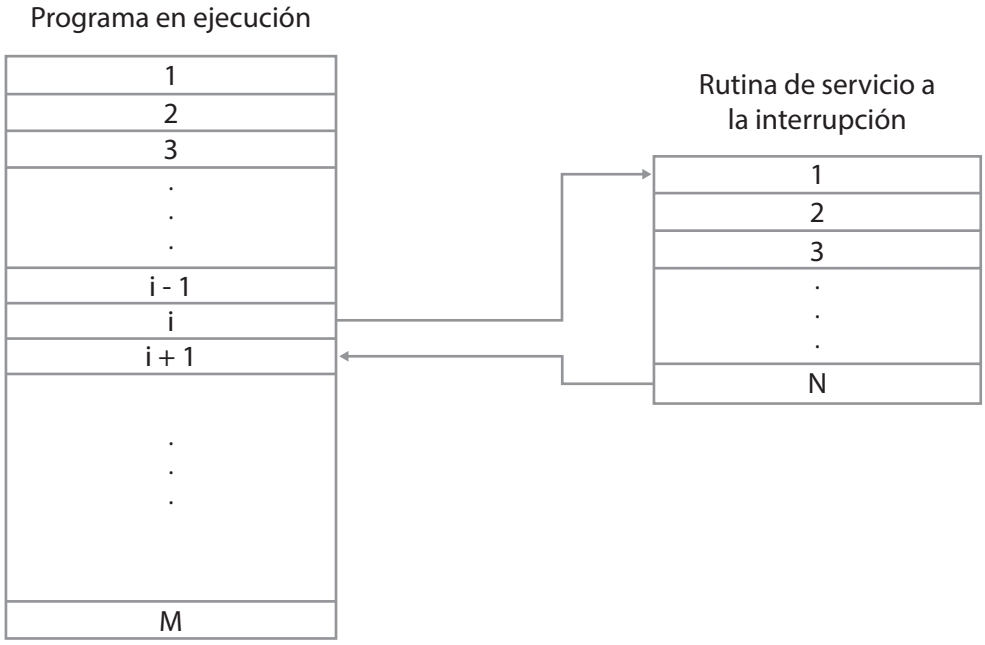


Figura 34. Transferencia de control a través de las interrupciones. Fuente: Adaptado desde (Hamacher, 2012).

Es necesario que el procesador informe al dispositivo causante de la interrupción, que su requerimiento está siendo atendido, de manera que pueda remover su señal de requerimiento de interrupción. Esta señal de control especial, conocida como *reconocimiento de interrupción* (del inglés, <<*interrupt acknowledge*>>) puede ser enviada desde el procesador al dispositivo de E/S, a través de la red de interconexión. Una alternativa a esta señal especial de reconocimiento, es que una instrucción de acceso a los registros de datos o de estado de la interfaz del dispositivo, implícitamente le informa que su solicitud de interrupción ha sido reconocida.

Dado que la información de estado y el contenido de los registros del procesador pueden ser alterados durante la ejecución de la rutina de servicio a la interrupción, es necesario almacenar esta información para ser restaurada antes del retorno desde la interrupción. Este proceso puede ser desarrollado en forma automático por el procesador o mediante instrucciones. Los procesadores modernos solo resguardan la información mínima necesaria para mantener la integridad de la ejecución del programa—generalmente se guarda el contenido del PC y el registro de estado del procesador—. La información adicional que se requiera almacenar debe ser hecha en forma explícita a través de instrucciones en la rutina de la interrupción. Esto es debido a que el proceso de almacenar información de los registros, incrementa el tiempo requerido entre que el requerimiento de la interrupción fue hecha y su ejecución (*latencia de la interrupción*). En algunas aplicaciones, una latencia de interrupción muy grande no es aceptable.

Otra aproximación es usar un conjunto de *registros sombra* (del inglés <<*shadow register*>>), los cuales son duplicados de los registros del procesador. De esta forma, no se requiere restituir los registros originales, dado que se trabaja sobre los duplicados.

En un sentido general, las interrupciones permiten la transferencia de control desde un programa a otro para ser iniciado por un evento externo al computador (Hamacher, 2012).

Dada la capacidad de las interrupciones de suspender la ejecución de un programa para dar paso a otro, deben ser gestionadas adecuadamente. Por ello, las computadoras tienen la posibilidad de habilitar o deshabilitar las interrupciones como lo deseen. El procesador puede aceptar o ignorar los requerimientos de interrupción; mientras que los dispositivos E/S pueden ser autorizados para generar señales de interrupción o, por el contrario, pueden ser prevenidos de hacer eso. Para ello, una forma de realizar este control es a través de bits de control en un registro especial, el cual puede ser accedido a través de programación.

Dentro del procesador, un registro especial denominado *registro de estado* (PS, del inglés <<*processor status*>>) mantiene información acerca del estado actual de operación. Dentro de este registro, un bit especial (IE), puede ser usado que el procesador acepte y sirva las interrupciones (IE=1) o las ignore (IE=0). De igual forma, las interfaces de los dispositivos de E/S tienen registros de control que contienen información que controlan el modo de operación del dispositivo; uno de los bits de ese registro puede ser usado para la gestión de las interrupciones.

Cuando un dispositivo activa la señal de requerimiento de interrupción, esta permanece en ese estado hasta que aprenda que el procesador ha aceptado esa solicitud. Por lo tanto, es importante que esta señal sea desactivada y no lleve a continuas interrupciones, causando que el sistema entre en un lazo infinito del cual no pueda recuperarse. Una alternativa es que el procesador automáticamente

desactive esta señal de requerimiento de interrupción antes de la ejecución de la subrutina de servicio a la interrupción. El procesador guarda el contenido del PC y del registro de estado (PS). Posterior a esto, deshabilita el bit IE para evitar futuras interrupciones. Posterior al servicio de interrupción los valores del PC y del PS son reestablecidos.

5.2.1. Gestionando múltiples dispositivos

Dado que los dispositivos son independientes, no existe un orden definitivo en el cual generarán las interrupciones. Como se vio en el apartado anterior, el requerimiento de interrupción de un dispositivo se plasma en la activación de un bit de interrupción (IRQ) en su registro de control. Una de las formas más sencillas para que el procesador identifique cual o cuales dispositivos están solicitando una interrupción, es que en la rutina de servicio a la interrupción se proceda a sondear el IRQ de todos los dispositivos; el primer dispositivo encontrado con el IRQ activo es el que será servido. La principal desventaja de este esquema de sondeo es el tiempo que se gasta en revisar todos los bits IRQ de los dispositivos, dado que muchos de ellos no estarán solicitando ese servicio.

Para evitar esto, una aproximación es el uso de *interrupciones vectorizadas* que permite que sea el mismo dispositivo quien se identifique con el procesador al momento de solicitar la interrupción, a través de su propia señal de interrupción o del envío de un código a través de la red de interconexión. Un espacio de memoria es destinado para mantener las direcciones de las rutinas de servicio de las interrupciones de los dispositivos. Esas direcciones son generalmente denominadas como *vectores de interrupciones* y constituyen la *tabla de vector de interrupción*. Por ejemplo, 128 bytes pueden almacenar 32 vectores de interrupción. Cuando una interrupción es requerida, la información provista desde el dispositivo requirente es usada como un puntero en la tabla de vector de interrupciones, de modo que la dirección en el vector de interrupción correspondiente es cargada inmediatamente en el contador de programa.

Al igual que en el caso de los llamados a subrutinas, con las interrupciones también existe la posibilidad de que durante una interrupción se produzca otra solicitud de interrupción desde otro dispositivo (siempre que el bit IE esté habilitado), en cuyo caso estamos hablando de interrupciones anidadas. Para ello, es necesario que se tenga un sistema de prioridades, de forma que una interrupción de mayor prioridad debe ser atendida cuando se está ejecutando la subrutina de atención a un pedido de una interrupción de menor nivel; pero no al contrario. Para poder desarrollar este esquema, se puede asignar mediante programa un nivel de prioridad al procesador, el cual corresponde al programa que está siendo ejecutado. El procesador aceptará solicitudes de interrupciones de dispositivos con un nivel superior al que está siendo ejecutado en ese momento. Automáticamente o a través de instrucciones especiales, el procesador elevará su prioridad a la de la nueva interrupción siendo servida. Este nivel de interrupción puede ser codificado en unos pocos bits del registro de estado (PS). No obstante, es necesario que antes de dar servicio a la nueva interrupción se proceda a guardar en la pila los contenidos del contador de programa y del registro de estado para su posterior restablecimiento.

Adicionalmente, es necesario mantener mecanismos en los circuitos de las interfaces los dispositivos de E/S que permitan controlar si este es permitido o no interrumpir al procesador. Esto se logra a través de un *bit de habilitación de interrupción*. Dado que la complejidad de los dispositivos puede ser variable, una aproximación para gestionar los distintos modos de funcionamiento es a través

de registros de control ubicados en sus interfaces. Estos registros son accesibles como ubicaciones direccionables, justo como con los datos y los registros de estado. Un bit de este registro sirve como bit de habilitación de la interrupción (IE).

5.2.2. Excepciones

Las excepciones hacen referencia a eventos que causan una interrupción. Un ejemplo de excepción son los eventos de interrupción producidos por los dispositivos de E/S. Otra forma de excepción se da cuando se detecta algún error. Por ejemplo, en las computadoras se incluyen un código detector de error en la memoria principal para detectar errores en los datos almacenados. Cuando un error es detectado por el hardware de control, se informa al procesador para iniciar una interrupción. De igual forma, se puede generar una interrupción cuando se detecta una condición inusual de una instrucción o una operación no permitida, como una división por cero. En esos casos, usualmente la instrucción, a diferencia de las interrupciones que no son causadas por errores, causante del error no es finalizada y el procesador inicia inmediatamente el proceso de excepción.

Tema 6.
Buses

Como hemos observado en los temas anteriores, las computadoras deben tener la capacidad de transferir datos hacia y desde los dispositivos de E/S, el procesador y la memoria. Para ello, es necesario una red de interconexión, la cual es llamada como *bus*. Durante el desarrollo de este tema, nos enfocaremos en el hardware necesario para hacer posible ese proceso de transferencia de datos.

6.1. Estructura de bus

La Figura 35 muestra un diagrama conceptual de la estructura de un bus. Este consiste de tres conjuntos de líneas usadas para llevar direcciones, datos y señales de control. Las interfaces de los dispositivos son conectadas a esas líneas. A cada uno de los registros de las interfaces de estos dispositivos se les asigna un conjunto único de direcciones. Cuando el procesador ubica una dirección en las líneas de direcciones, los decodificadores de direcciones de todos los dispositivos analizan este valor y el dispositivo correspondiente responde al comando enviado a través de las líneas de control. Estos comandos buscan la lectura o escritura de datos, los cuales son enviados a través de las líneas de datos.

Figura 35. Transferencia de control a través de las interrupciones. Fuente: Adaptado desde (Hamacher, 2012).

6.2. Operación del bus

La operación del bus está guiada por un conjunto de reglas a las cuales se les conoce como el *protocolo del bus*. Este protocolo establece cuándo un dispositivo puede ubicar información en el bus, cuándo los datos en el bus pueden ser cargados en sus registros,… Estas reglas son implementadas a través de comandos sobre las líneas de control, que indican qué y cuándo las acciones son desarrolladas.

Una de las líneas de control usualmente es lectura y escritura *($R/\overline{W}$)*. Como se puede observar, la nomenclatura indica que la lectura está activa cuando existe un valor de 1 en esta línea; mientras que la escritura cuando el valor corresponde a 0. Además, las líneas de control especifican información de temporización, de manera que el procesador y los dispositivos conozcan cuándo ubicar o recibir datos desde las líneas de datos. Estos esquemas de funcionamiento pueden ser clasificados en *síncronos* y *asíncronos*. En las operaciones de transferencias de datos entre dispositivos, uno de ellos —generalmente el procesador— funge como *maestro*; mientras que el otro es referido como *esclavo*.

En el caso de un *bus síncrono*, todos los dispositivos obtienen la información de temporización desde una línea de control denominada *reloj del bus* (*bus clock*). Como se puede observar en la Figura 36, está compuesta por dos fases: un nivel alto seguido por un nivel bajo; en su conjunto constituyen el *ciclo de reloj*. Se considera a la primera mitad de este ciclo, como el *pulso de reloj*. Una convención útil para indicar que algunas líneas están en alto y bajo, dependiendo de sus valores particulares, es mostrar como que estuvieran llevando ambos valores en la gráfica. Una línea en la mitad entre el valor alto y bajo indican periodos en los cuales la señal no es legible y debe ser ignorado por todos los dispositivos.

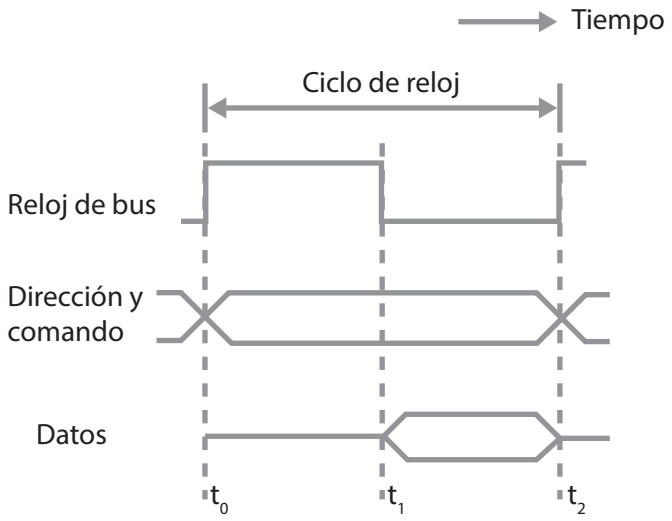


Figura 36. Temporización de una transferencia entrante en un bus síncrono. Fuente: Adaptado desde (Hamacher, 2012).

Es necesario considerar que la información viaja en el bus a una velocidad determinada por sus características físicas y eléctricas. Por ello, la longitud del pulso de reloj debe ser superior al máximo valor del retardo de propagación del bus. Además, este tiempo debe permitir que los dispositivos puedan decodificar las direcciones y las señales de control; de modo que el dispositivo requerido pueda en el tiempo t1 responder ubicando los datos correspondientes en las líneas de datos. Sobre el final, en t2, el maestro carga los datos disponibles en las líneas de datos en uno de sus registros.

Dada la presencia de retardos de propagación de las señales por las condiciones físicas y eléctricas de las líneas de bus, la visión de las señales que comparten los dispositivos es diferente. El dispositivo esclavo recibirá la señal emitida por el maestro un tiempo después; lo mismo ocurre con los datos enviados por el esclavo al maestro.

El esquema anterior es el más simple; no obstante, tiene algunas limitaciones. Debido a que la transferencia de datos debe ser completada dentro de un ciclo de reloj, este periodo debe ser escogido de forma tal que se ajuste al retardo más grande del bus y la interfaz del dispositivo más lento. Además, el procesador no tiene forma de conocer si el dispositivo direccionado realmente ha respondido. Para superar estas limitaciones los buses incorporan señales de control que representan las respuestas de los dispositivos. Estas señales informan al maestro que el esclavo ha reconocido la dirección y que se encuentra listo para la operación de transferencia de datos; además, permiten ajustar el periodo de transferencia de datos para armonizar las velocidades de respuesta de los diferentes dispositivos. Esto se logra permitiendo que una transferencia completa de datos pueda durar varios ciclos de reloj.

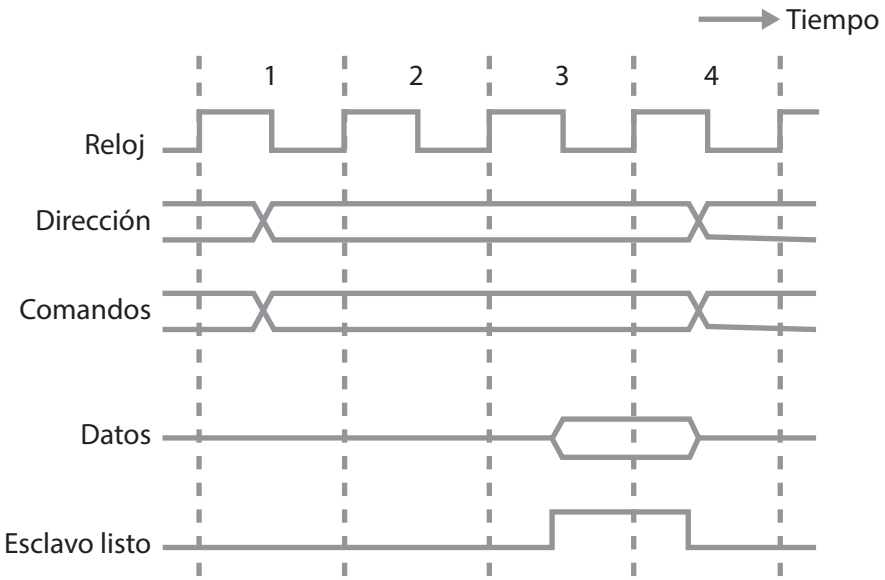


Figura 37. Una transferencia entrante usando múltiples ciclos de reloj. Fuente: Adaptado desde (Hamacher, 2012).

En la Figura 37 se puede observar un ejemplo de un proceso de lectura de datos usando múltiples ciclos de reloj. En primera instancia, el máster ubica la dirección y los comandos correspondientes. No obstante, debido a los retardos, recién en el tercer ciclo de reloj se ubica los datos en el bus correspondiente y se activa la señal de esclavo activo. Esta señal de control permanece activa hasta mediados del ciclo 4, donde se desactiva la señal y el bus de datos pasa a alta impedancia.

Por su parte, el *bus asíncrono*, funciona en base a un protocolo que se basa en el intercambio de comandos y respuestas entre el maestro y el esclavo. El proceso es como sigue. El maestro ubica la dirección y el comando sobre el bus. Luego de ello, procede a avisar a los dispositivos esclavos a través de la señal de control maestro listo. Con ello, los dispositivos decodifican la dirección recibida y verifican si es para alguno de ellos y la operación a desarrollar. El destinatario realiza la operación requerida y avisa al maestro de esta acción mediante la señal esclavo listo. Cuando el maestro recibe esta señal, remueve su señal del bus. La Figura 38 muestra este proceso para la lectura de datos.

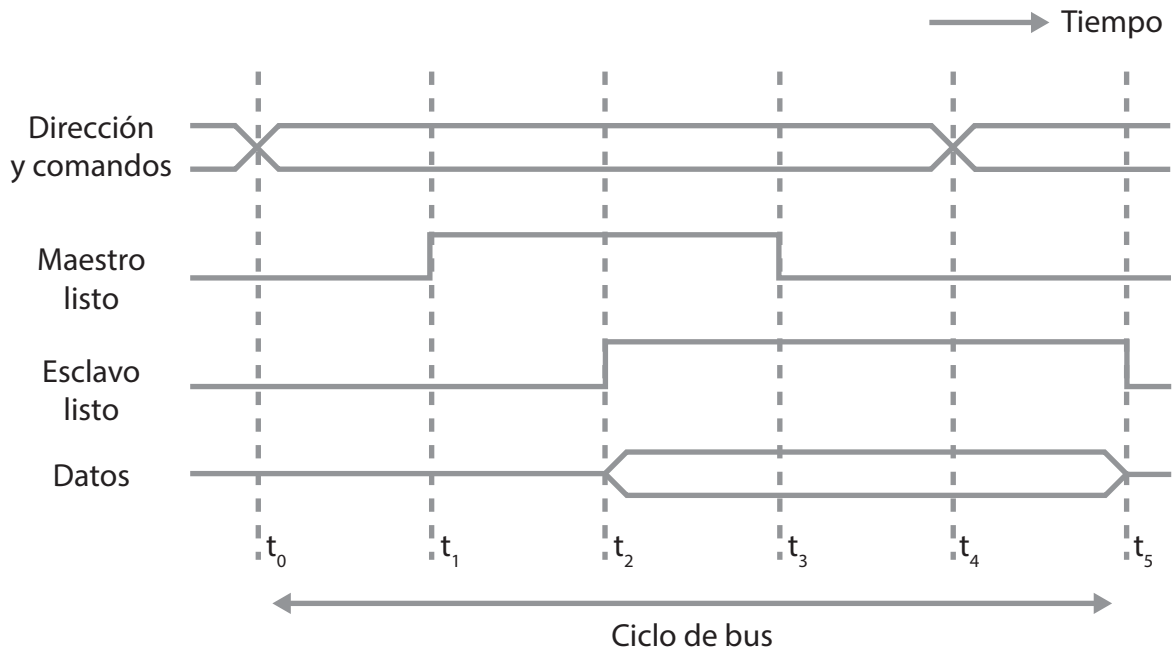


Figura 38. Intercambio de control entre el maestro y el esclavo en la transferencia de datos durante una operación de entrada. Fuente: Adaptado desde (Hamacher, 2012).

6.3. Arbitraje

En determinadas ocasiones puede suceder que dos o más dispositivos contiendan por el uso de un recurso. En ese caso, es necesario determinar cuál de ellos accede primero. Una forma de coordinar esto, es a través de un circuito de arbitraje. El árbitro determina cuál de los requerimientos para uso del recurso es el que cuenta con mayor prioridad. Por ejemplo, si tenemos varios dispositivos de E/S que desean acceder a memoria a través de un único bus, el árbitro analiza la prioridad de estos dispositivos para dar paso a su solicitud. En caso de que no existan diferencia en sus prioridades, se suele hacer uso del método round-robin en la asignación.

6.4. Circuitos de interfaz

El proceso de conexión de los dispositivos de E/S se desarrolla a través de unos circuitos de interfaz, los cuales, por uno de sus lados, se enlaza a las líneas de datos, direcciones y control y, por el otro lado, se tiene las conexiones necesarias para la transferencia de datos entre la interfaz y el dispositivo de E/S. Este último, es conocido como puerto, el cual puede ser serial o paralelo. La comunicación con el procesador es la misma para ambos formatos, la conversión desde serial a paralelo y viceversa se desarrolla en el circuito de interfaz. En el caso de comunicación serial, la transmisión y recepción de la información se desarrolla bit a bit; mientras que en la comunicación en paralelo, se desarrolla el envío y recepción de múltiples bits desde y hacia el dispositivo.

Las principales funciones de una interfaz de E/S son las siguientes (Hamacher, 2012):

- Proveer almacenamiento temporario para los datos.

- Incluir un registro de estado, accesible desde el procesador, con información sobre el estado del dispositivo.
- Incluir un registro de control que mantiene la información que gobierna el comportamiento de la interfaz.
- Contener la circuitería de decodificación de las direcciones para determinar cuándo está siendo direccionada por el procesador.
- Generar las señales de tiempo requeridas.
- Realizar las conversiones necesarias para la transferencia de datos entre el procesador y el dispositivo de E/S (serial a paralelo o paralelo a serial).

6.5. Estándares de interconexión

Con el objeto de que los dispositivos de E/S puedan usar interfaces que sean independientes de algún procesador en particular, varios estándares han sido desarrollados. En esta sección analizaremos algunos de ellos.

6.5.1. USB

El Bus Universal en Serie (USB, del inglés <<Universal Serial Bus>>) es un estándar de interconexión ampliamente usado en una variedad de dispositivos. Su éxito se debe principalmente a su simplicidad, bajo costo y su facilidad de operación a través del modo <<*plug-and-play*>>. En su versión inicial se tenía dos velocidades: 1.5 Mbits/seg. y 12 Mbits/seg. Posteriormente, se introdujeron versiones de velocidades superiores como el USB 2 que soporta 480 Mbits/seg., y el USB 3 con tasas hasta 5 Gigabits/seg.

El estándar USB define tanto el hardware como el software que se comunica con este. La característica <<*plug-and-play*>> significa que cuando un nuevo dispositivo es conectado, el sistema detecta automáticamente su existencia. Este estándar usa conexiones punto a punto y un formato de transmisión serial.

Cuando varios dispositivos son conectados, estos son dispuestos en una estructura de árbol, en la que en cada nodo existe un dispositivo llamado *hub*, el cuál actúa como punto de transferencia intermedio entre el procesador y los dispositivos de E/S. En la raíz del árbol, el *hub raíz* conecta todo el árbol al computador. El USB funciona estrictamente por un método de encuesta. Un dispositivo puede enviar un mensaje solo en respuesta a un mensaje de encuesta desde el procesador. De esta forma, dos dispositivos no pueden enviar mensajes al mismo tiempo, lo cual permite que los hub sean simples y de bajo costo. A cada uno de los dispositivos sobre el USB se le asigna una dirección de 7 bits, las cuales son locales al dispositivo, no relacionadas con el espacio de memoria del procesador. La comunicación entre el procesador y los dispositivos en el USB se da a través del *hub raíz*.

Una de las principales características de la USB es su capacidad de soportar transferencias isócronas. Para ello, se transmite un conjunto variable de datos mediante una secuencia de paquetes de tamaño fijo, a intervalos regulares.

Las conexiones eléctricas se basan en cuatro cables, de los cuales dos son de energía, +5V, y tierra, y dos para datos. Dos formatos de transmisión son usados: en el primero, transmisión de un solo extremo (del inglés <<*single-ended transmission*>>), en la cual un voltaje alto relativo a tierra se transmite sobre uno de los cables de datos para representar un 0 y sobre el otro para representar un 1. La principal dificultad con este tipo de transmisión es el ruido que se introduce y que por lo tanto limita su velocidad de transmisión. Para superar este problema, se utiliza el método de señalización diferencial (del inglés <<*differential signaling*>>), en este método, el cable de tierra no es usado, de manera que el receptor es sensitivo a la diferencia de voltaje entre los dos cables de datos, por lo que algún ruido introducido en los cables de datos es cancelado.

6.5.2. FireWire

Este tipo de bus hace uso de enlaces seriales diferenciales punto a punto. A diferencia del bus USB, los dispositivos en FireWire son conectados en una estructura a manera de cadena Daisy. Un primer dispositivo es conectado al procesador, el segundo al primero, el tercero al segundo y así sucesivamente. Es apropiado para conexiones de dispositivos de audio y video, pudiendo operar en modo de transferencia isócrono para lo cual está altamente optimizado.

Además, FireWire soporta el modo de operación llamado <<peer-to-peer>>, lo que permite que los datos puedan ser transferidos en forma directa entre dispositivos de E/S. Varias versiones de este estándar han sido definidas, las cuales pueden operar en velocidades entre 400 Megabits/seg., a 3.6 Gigabits/seg.

6.5.3. Bus PCI

El bus PCI (del inglés <<Peripheral Component Interconnect)>> está ubicado en la tarjeta madre del computador. Este es usado para conectar las interfaces de E/S para una amplia variedad de dispositivos; de manera que aparece a ojos del procesador como si estuviera conectado directamente a su bus. A sus registros de interfaz se les asigna direcciones en el espacio de direcciones del procesador.

El bus PCI es conectado al procesador mediante un controlador denominado *puente*. Este dispositivo tiene un puerto especial que conecta la memoria la memoria principal del computador (ver Figura 39). El puente traduce y retransmite comandos y respuestas desde un bus a otro y transfiere datos entre ellos. Por ejemplo, cuando el procesador envía un requerimiento de lectura a un dispositivo E/S, el puente retransmite el comando y direcciona el bus PCI; mientras que, cuando recibe la respuesta del dispositivo de E/S, este reenvía la respuesta al procesador a través del bus del procesador (Hamacher, 2012).

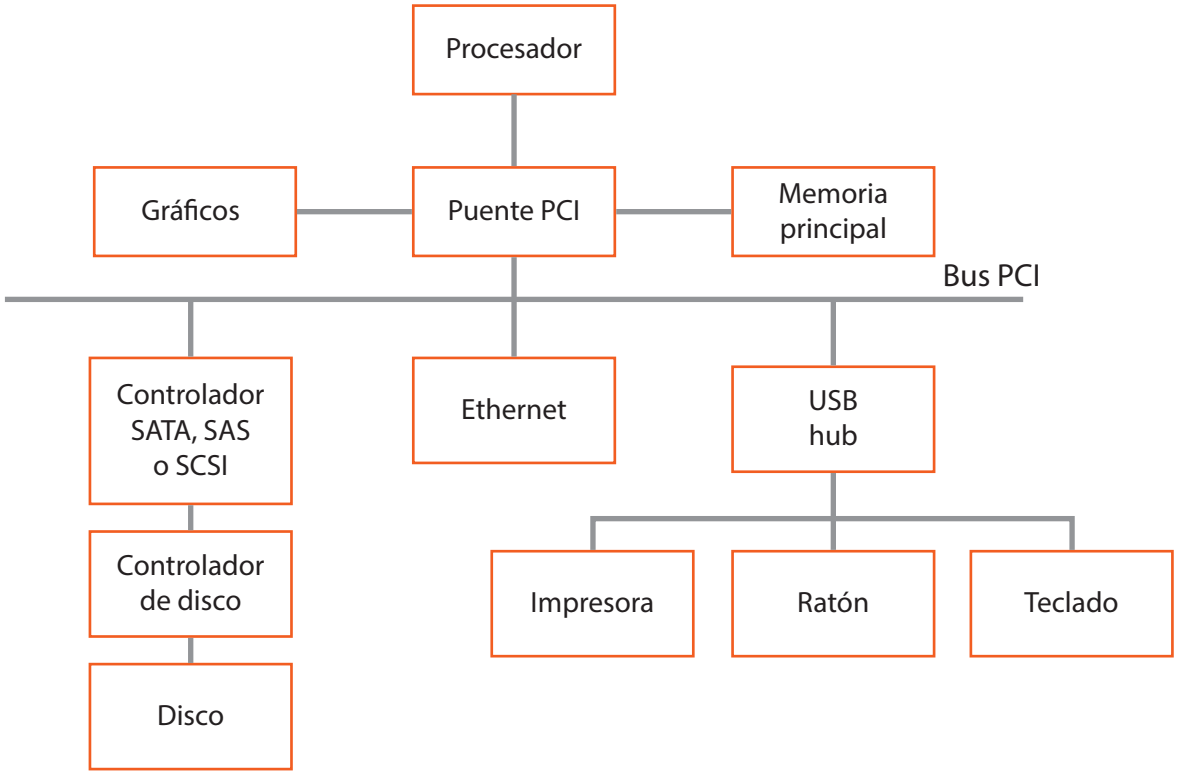


Figura 39. Uso del bus PCI en un sistema de computación. Fuente: Adaptado desde (Hamacher, 2012).

Como se puede observar en la Figura 39, los dispositivos E/S son conectados al bus PCI, posiblemente a través de puertos que usan diversos estándares. Tres posibles espacios de memoria son disponibles para este tipo de bus: memoria, E/S y configuración. El diseñador puede escoger E/S asignada en memoria aun cuando exista un espacio de direcciones separado para E/S. Esta elección permite una mayor compatibilidad. Por su parte, el espacio de configuración, permite el despliegue del modo <<*plug-and-play*>>.

El bus PCI está diseñado para soportar transferencias de múltiples palabras. Además, este bus usa las mismas líneas para transmitir direcciones y datos. Para ello, se mantiene la dirección hasta que es decodificada por el esclavo y luego de esto es liberado para la transmisión de los datos. En el caso de múltiples palabras, el esclavo puede almacenar la dirección en un registro interno e incrementar este para acceder a direcciones sucesivas.

Los dispositivos conectados al bus PCI no son asignados direcciones permanentes dentro del registro de su hardware de interfaz; por el contrario, su asignación se desarrolla mediante software. Se puede conectar hasta 21 conectores para las tarjetas de interfaz de los dispositivos de E/S. Cada conector tiene un pin llamado Selección del Dispositivo de Inicialización (IDEL#), los cuales son conectados a una de las 21 líneas de datos y direcciones. La selección de un dispositivo se hace a través de la activación de la línea correspondiente; mientras que las otras permanecen en 0. Si un dispositivo responde, este es asignado una dirección, la cual es escrita en su registro designado para su propósito.

6.5.4. Bus SCSI

SCSI es el acrónimo para <<*Small Computer System Interface*>>. Este estándar puede ser usado con una amplia variedad de dispositivos; sin embargo, es apropiado para usar con controladores de disco. La especificación original permitía una longitud de 25 metros y una tasa de transferencia de 5 Megabytes/seg. Posteriormente, los estándares SCSI-2 y SCSI-3 han incrementado sus tasas de transmisión. Los datos son transferidos en 8 bits o 16 bits en paralelo, usando velocidades de reloj de hasta 80MHz. En cuanto al esquema de señalización eléctrica, se puede usar transmisión de un solo extremo o señalización diferencial.

6.5.5.SATA

La versión inicial del bus usado por IBM, llamado AT, se convirtió en un estándar de la industria, denominado ISA (del inglés, <<Industry Estándar Architecture>>). Una versión mejorada, la cual incluyó el software necesario para el soporte de controladores de discos, fue denominada como ATA (del inglés <<AT Attachment>>). La versión serial de esta arquitectura es conocida como SATA (del inglés << Serial Advanced Technology Attachment >>), ampliamente usada como interfaz para discos. El esquema del conector tiene 7 pins, conectando dos pares gemelos y tres de tierra. La señalización eléctrica es diferencial, con frecuencias de reloj que van desde 1.5 a 6 Gigabits/seg. Recientes versiones proveen la capacidad de transmisión isócrona, lo que facilita el soporte de dispositivos de video y audio.

Tema 7.
Organización del procesador

Como hemos visto en los capítulos anteriores, el procesador busca las instrucciones una a una desde posiciones continuas de memoria —hasta encontrar una instrucción de salto o bifurcación— las deposita en *el registro de instrucciones* (IR) y las ejecuta. La primera instancia, esto es la búsqueda de la instrucción y su ubicación en el registro IR, es conocido como la *fase de búsqueda de la instrucción*; mientras que, la ejecución de la operación especificada en esa instrucción es conocido como *fase de ejecución de la instrucción.*

Las acciones necesarias para ejecutar las operaciones especificadas por una instrucción son ejecutadas a través de los siguientes componentes de hardware: *la interfaz memoria-procesador*, la cual está encargada de transferir los datos desde y hacia la memoria durante las operaciones de lectura y escritura; *el generador de las direcciones de instrucción*, actualiza el contenidos del PC (contador de programa) luego de que cada instrucción es buscada; *el archivo de registro*, es una unidad de memoria que almacena las locaciones que son organizadas desde los registros de propósito general del procesador; la unidad lógica y aritmética (ALU) es la que realiza los requerimientos de computación.

Tal como se vio en los capítulos anteriores, muchas de las operaciones ejecutadas por el procesador son desarrolladas con datos almacenados en registros. Para ello, esos datos son procesados por circuitos combinacionales, tales como sumadores, y los resultados son ubicados en un registro. No obstante,

la complejidad de ejecutar todas las acciones del procesador a través de un solo bloque de circuitería combinacional puede ser disminuida si es subdividida en varios pasos más simples, ejecutados por sub-circuitos del circuito original. Estos pueden ser ubicados en un formato en cascada dentro de una estructura multi-estado. De esta forma, si se tiene n estados, la operación será ejecutada en n ciclos de reloj.

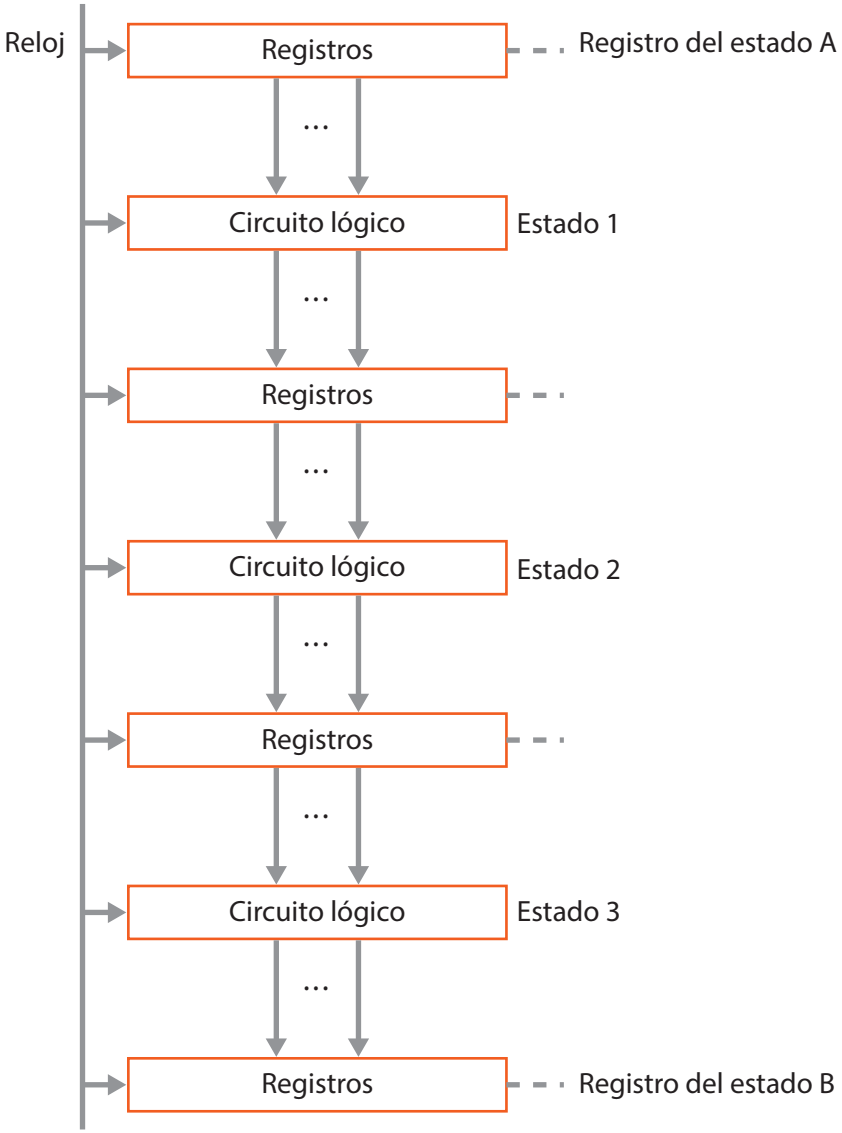


Figura 40. Una estructura de hardware con múltiples estados. Fuente: Adaptado desde (Hamacher, 2012).

La secuencia de acciones para la búsqueda y ejecución de una instrucción puede ser establecida en cinco pasos (Hamacher, 2012):

1. Buscar una instrucción e incrementar el contador de programa.
2. Decodificar la instrucción y leer registros desde el archivo de registro.
3. Realizar una operación en el ALU.

4. Leer o escribir la memoria de datos si la instrucción envuelve una operación de memoria.

5. Escribir los resultados dentro del registro de destino, si es necesario.

7.1. Componentes de hardware

Estos cinco pasos para la ejecución de instrucciones en procesador RISC nos permite organizarlo en cinco estados, de forma tal que cada uno de los estados desarrolla las acciones de uno de los pasos. A continuación analizaremos los componentes de hardware y su estructuración dentro de estos cinco estados.

7.1.1. Archivo de registro

Corresponde a un bloque de memoria pequeño y de acceso rápido, sobre el cual se organizan los registros de propósito general. La circuitería de acceso es desarrollada de forma tal que permite habilitar la lectura de dos registros en forma simultánea, entregando su contenido en dos salidas diferenciadas: A y B. Se provee de dos entradas de direcciones, uno para cada uno de los registros, las cuales están conectadas a los campos correspondientes en el registro de instrucciones (IR), que especifican los registros fuente. Adicionalmente, este registro posee una entrada de datos C, y la entrada de dirección que selecciona el registro correspondiente.

En las unidades de memoria, se denomina como puertos a sus entradas o salidas. Las memorias con dos puertos de salida son conocidas como de doble puerto.

7.1.2. Unidad lógica y aritmética

La ALU es la encargada de realizar las operaciones lógicas y aritméticas, tales como suma, resta, AND, OR y XOR. Como se puede ver en la Figura 41, cuando se ejecuta una operación lógica o aritmética, los valores de los operandos son obtenidos a través de una lectura del registro de archivo. Un multiplexor (MuxB) permite seleccionar, dependiendo de la instrucción, un valor inmediato o desde un registro. La salida del ALU es conectada al registro de archivo de manera que pueda ser almacenado en el registro de destino indicado a través de la dirección C.

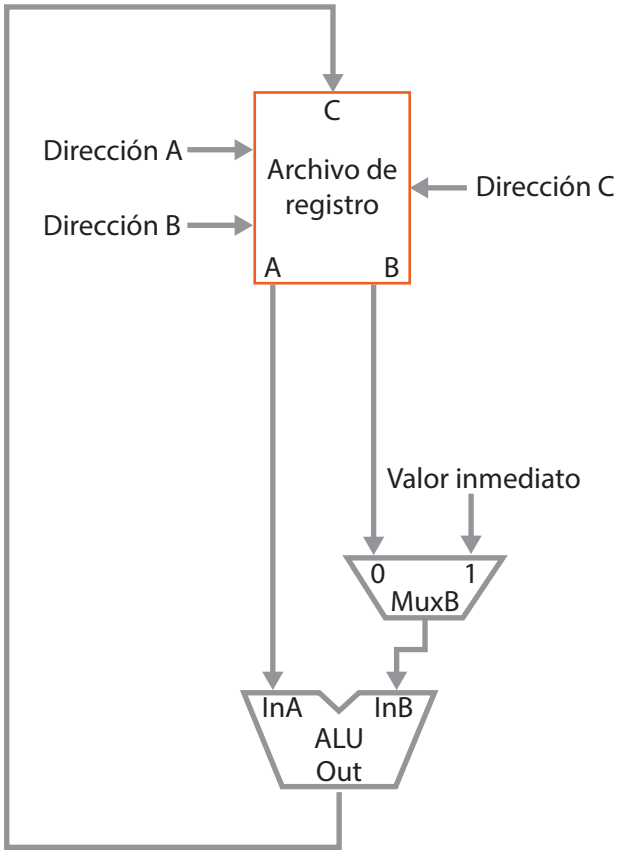


Figura 41. Visión conceptual del hardware necesario para la computación. Fuente: Adaptado desde (Hamacher, 2012).

7.1.3. El camino de datos

El procesamiento de las instrucciones consisten de dos fases: búsqueda de la instrucción y ejecución de la instrucción. En la primera, a más de buscar la instrucción, también es responsable por decodificarla y por generar las señales de control que causen las apropiadas acciones que tomen lugar durante la ejecución. La sección de ejecución lee los datos de los operandos especificados en las instrucciones, realiza la computación requerida y almacena los resultados.

Una organización del hardware en múltiples estados puede ser definido en los siguientes: búsqueda de la instrucción, registros fuente, ALU, acceso a memoria, registro de destino. El hardware correspondiente a los estados 2 al 5, mostrados en la Figura 42, son denominados como camino de datos (del inglés <<*datapath*>>).

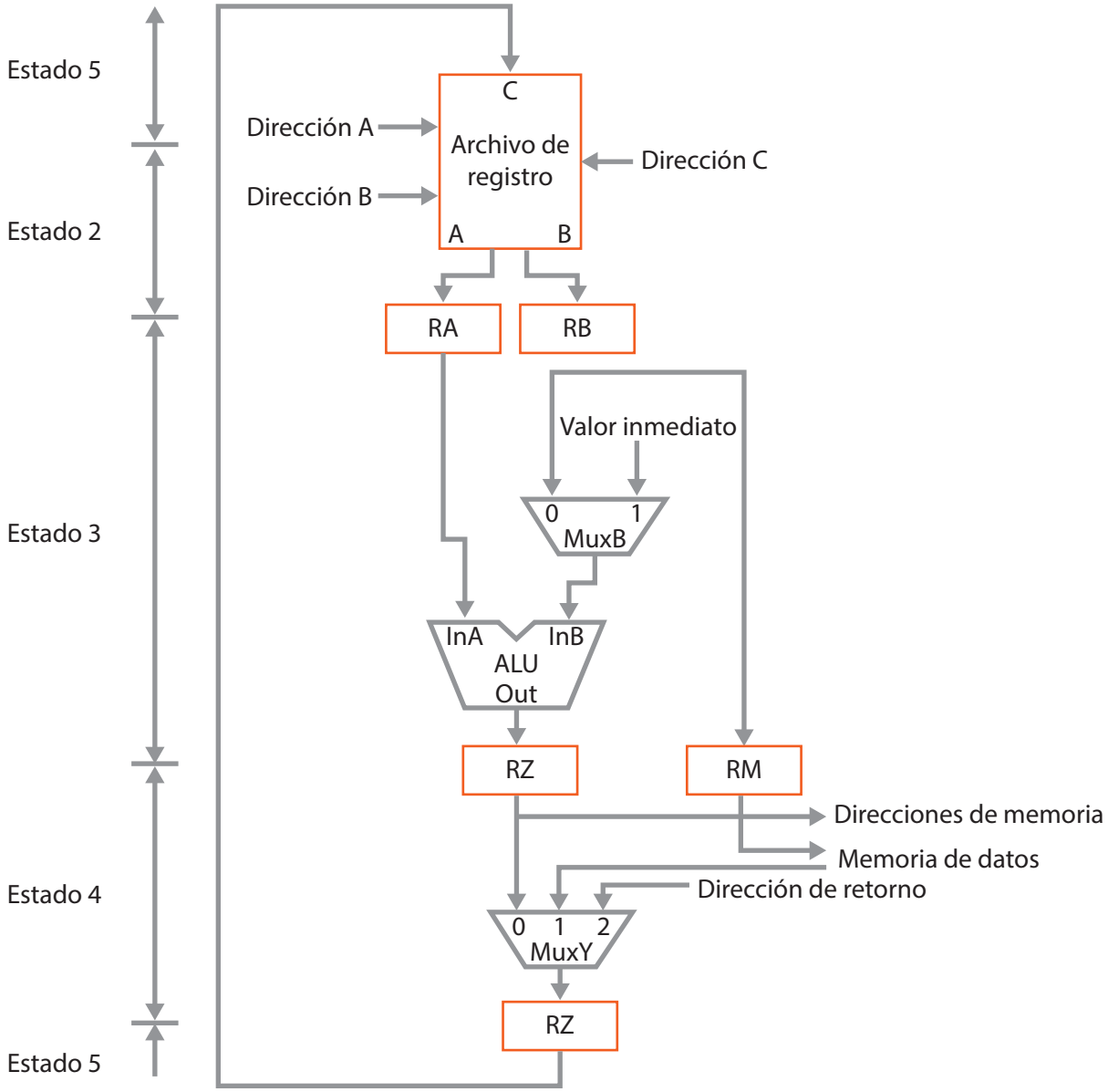


Figura 42. Camino de datos en un procesador. Fuente: Adaptado desde (Hamacher, 2012).

Los registros inter estados mantienen los resultados producidos en un estado, tal que pueden ser empleados como entrada para el próximo estado. En la Figura 42 se muestran cinco de estos registros: RA, RB, RZ, RM y RY. Los dos primeros están relacionados con las salidas del archivo de registro, en donde se almacenan los datos fuente que se usaran en las operaciones al interior de la ALU. En el caso de aquellas instrucciones que no requieren el uso de unidad lógica y aritmética, que es el estado 3, se encuentra el registro RM. Por ejemplo, para las instrucciones de almacenaje, el dato leído desde el archivo de registro, estado 2, es mantenido en el registro RB. Dado que el acceso a memoria se da en el estado 4, el registro RM se usa para recoger esta información y, de esta forma, mantener el correcto flujo de datos. El ALU constituye el estado 2 y su resultado es almacenado en el registro RZ. El multiplexor MuxY (Figura 42) permite seleccionar, dependiendo de la instrucción, tres alternativas: el resultado desde la ALU; información desde la memoria de datos; o la dirección de retorno, en el caso

de llamadas a subrutinas. Esta información es almacenada en el registro RY y es enviado en el estado 5 al registro destino.

7.1.4. La sección de búsqueda de la instrucción

Una visión conceptual del hardware empleado en la fase de búsqueda de las instrucciones es presentada en la Figura 43. En él se observa que el multiplexor MuxMA permite el direccionamiento de memoria desde dos vías: el PC y el registro RZ desde el camino de datos cuando se accede operandos de instrucciones. La opción seleccionada es enviada a la interfaz memoria-procesador. El PC se encuentra dentro del bloque del generador de direcciones de instrucción, el cual permite su actualización después de que cada instrucción es buscada.

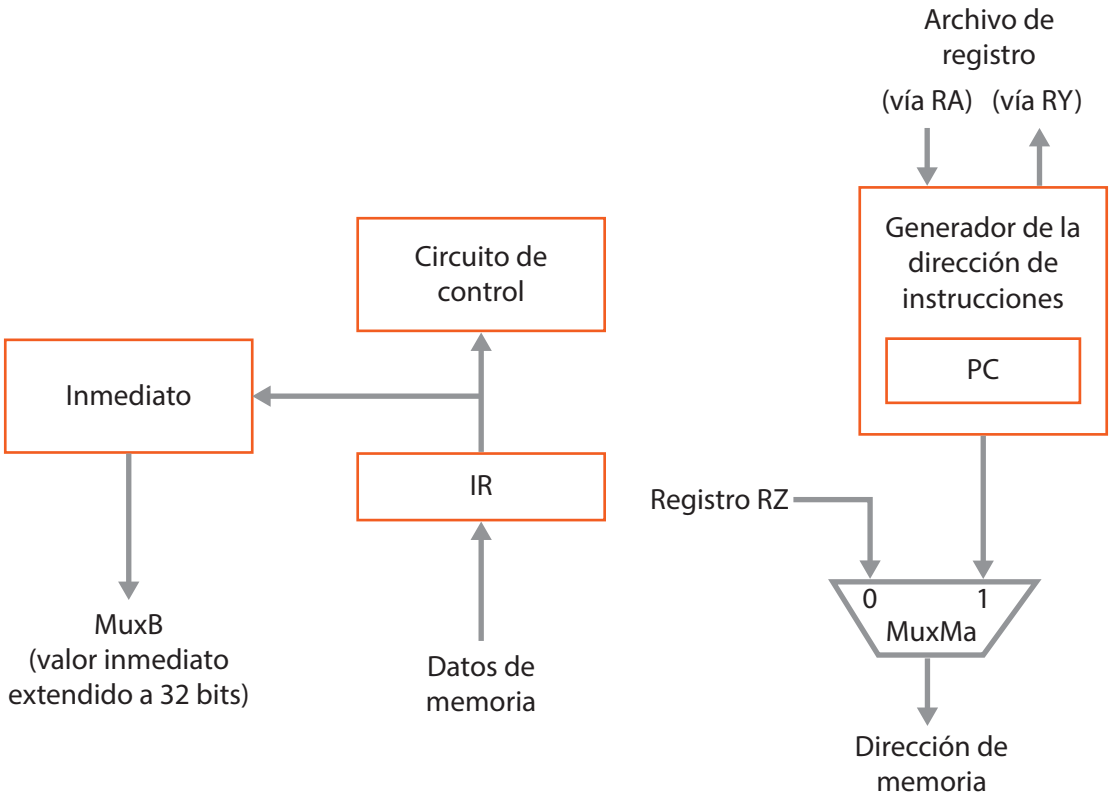


Figura 43. Sección de búsqueda de instrucción. Fuente: Adaptado desde (Hamacher, 2012).

La instrucción leída desde memoria es almacenada en el registro de instrucciones (IR) hasta que es ejecutada y la próxima instrucción es buscada. Como se puede ver en el diagrama de la Figura 43, la instrucción es examinada por el circuito de control para generar las señales necesarias para controlar el hardware del procesador. A más de ello, también puede ser usado por el registro inmediato para su uso con algunas instrucciones, donde permite computar directamente la dirección efectiva de un operando.

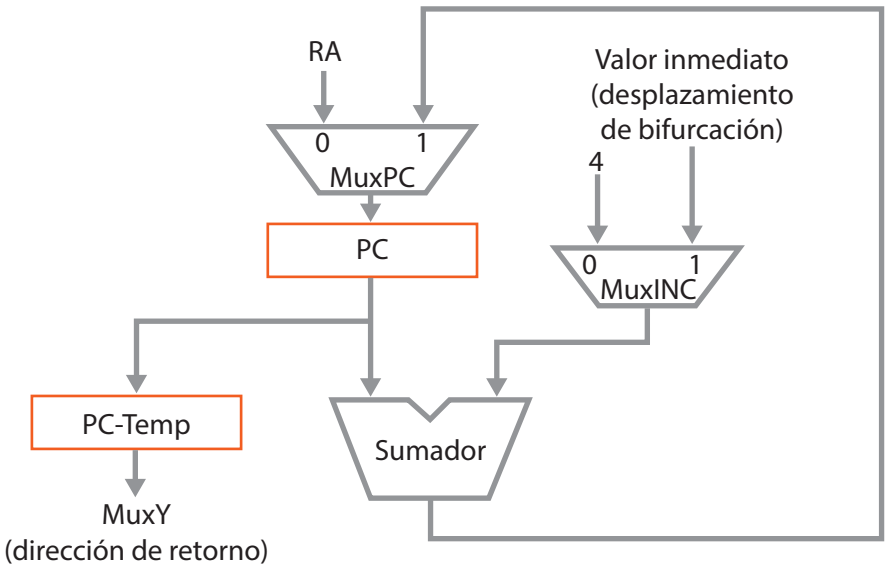


Figura 44. Generador de direcciones de instrucciones. Fuente: Adaptado desde (Hamacher, 2012).

La Figura 44 ilustra la estructura del generador de direcciones de las instrucciones. El sumador es usado para el incremento del PC por 4 y para computar el valor a ser cargado en el PC cuando se ejecuta instrucciones de bifurcación o de llamados a subrutinas. El multiplexor MuxPC tiene como entradas la salida del sumador y el dato proveniente del registro RA. Este último es necesario cuando se ejecutan instrucciones de enlace a subrutinas. El registro PC-Temp es necesario para mantener temporalmente la dirección de la PC durante los procesos de guardar la dirección de retorno de las subrutinas o interrupciones.

7.1.5. Señales de control

Las operaciones del hardware de los componentes del procesador están gobernadas por *señales de control*. Estas señales determinan las entradas de los multiplexores que deben ser seleccionadas, qué operaciones deben ser desarrolladas por la ALU, entre otras. Como ya se analizó en la sección correspondiente al camino de datos, tenemos un conjunto de registros inter estados que nos permiten mantener la información necesaria de un estado al otro; por lo tanto, estos registros siempre están habilitados. No obstante, es necesario recalcar que los otros registros: PC, IR, el archivo de registros no deben cambiar en cada ciclo de reloj, sino cuando son llamados en un paso de procesamiento particular. Por ello, estos registros deben estar habilitados en esos tiempos. La Figura 45 nos muestra las diferentes señales de control para el camino de datos.

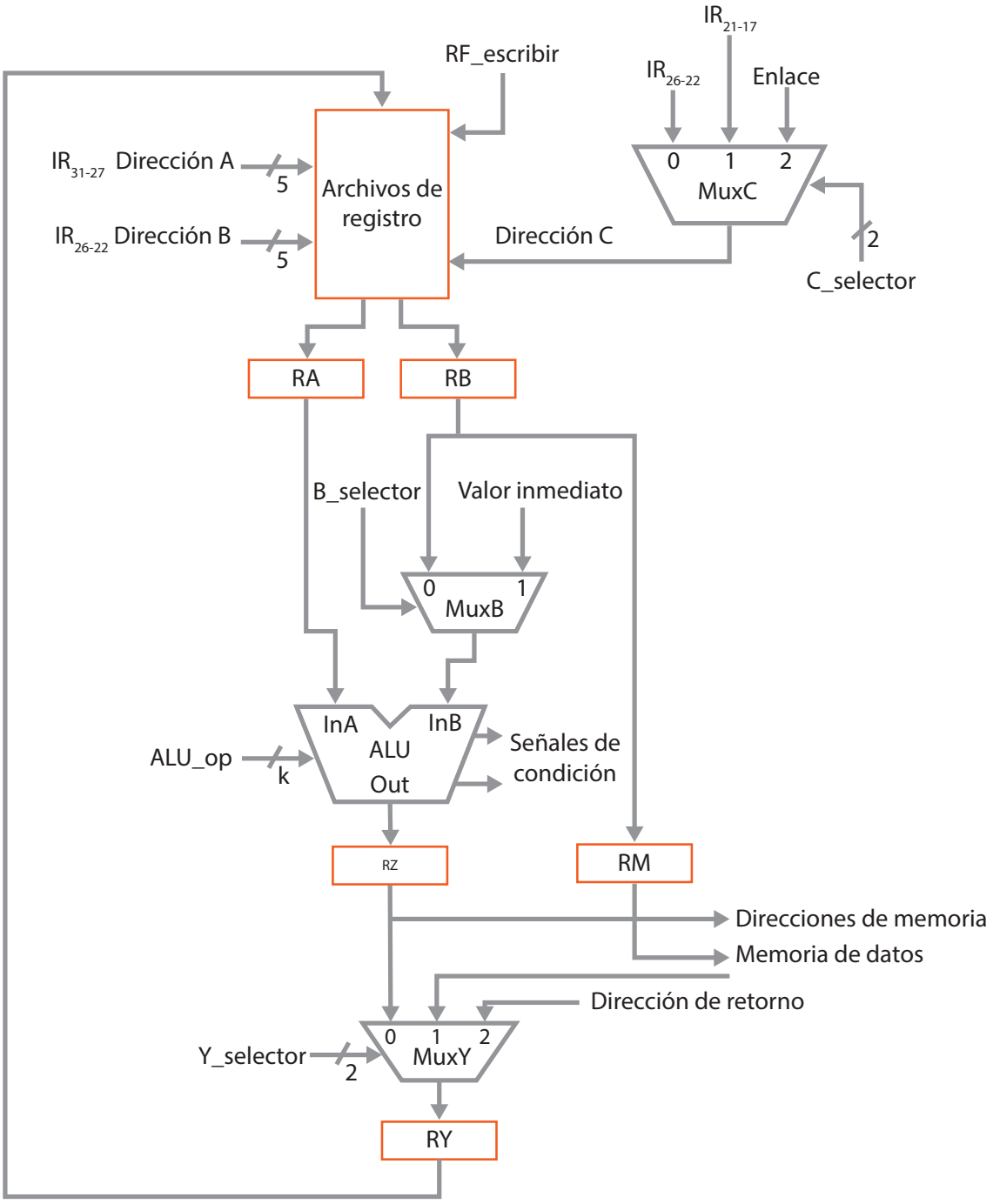


Figura 45. Señales de control para el camino de datos. Fuente: Adaptado desde (Hamacher, 2012).

Los multiplexores están controlados por señales que seleccionan qué entrada debe aparecer en su salida. Por su parte, la operación desarrollada por la ALU está gestionada por un código de control de *k* bits, con lo cual se puede especificar hasta 2^k operaciones distintas. Al igual que estas señales, la interfaz memoria-procesador y el generador de instrucciones presentan señales similares que les permite, por ejemplo, en el caso del primero, habilitar operaciones de lectura o escritura en memoria; habilitar el registro de instrucciones para cargar una nueva instrucción, la cual debe ser desarrollada solo después de que la interfaz memoria-procesador active una señal de control específica MFC que

se activa solo cuando la operación de lectura o escritura ha sido completada. En el caso del generador de direcciones de instrucciones (ver Figura 44), existen algunas señales que permiten, por una parte la selección de las entradas del multiplexor MuxINC —para seleccionar el valor a ser añadido al PC— y del MuxPC —para determinar si se usa la dirección actualizada o el contenido del registro RA—; mientras que otras permiten habilitar el registro PC.

Hay dos aproximaciones para la generación de las señales de control desde el procesador: control cableado y control microprogramado. Analicemos en primera instancia el sistema de control cableado.

Dado que una instrucción es ejecutada en una secuencia de pasos, donde cada paso requiere de un ciclo de reloj, es necesario tener un contador de pasos para mantener un seguimiento al progreso de la ejecución. Dependiendo de la instrucción que se ejecute, diversas acciones deben ser desarrolladas, por ello, la configuración de las señales de control dependen de (Hamacher, 2012):

- Contenido del contador de pasos
- Contenidos del registro de instrucciones
- Resultados de una computación o una operación de comparación
- Señales de entrada externas, tal como requerimientos de interrupción

La Figura 46 muestra la circuitería necesaria para generar las señales de control. El decodificador de instrucciones interpreta los códigos recibidos desde el IR y establece a 1 salidas INSi correspondientes. Durante cada ciclo de reloj, una de las salidas del contador de pasos es establecida a 1, conforme al paso que está en curso. El generador de señales de control es un circuito combinacional que produce las señales de control necesarias conforme con los parámetros de ingreso que recibe.

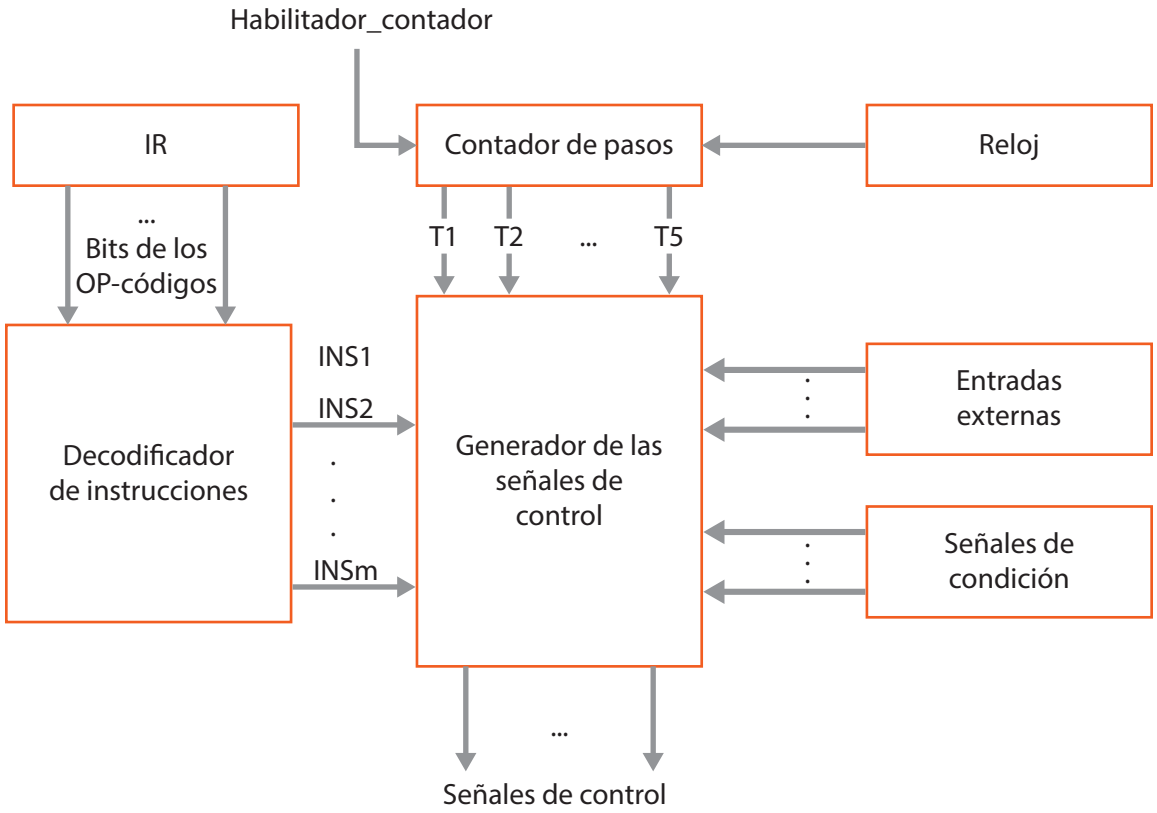


Figura 46. Generación de las señales de control. Fuente: Adaptado desde (Hamacher, 2012).

Otra forma de implementación de las señales de control, es la basada en programación (software). En ella, la configuración de las señales den control en cada paso son determinadas por un programa, denominado microprograma, almacenado en una memoria especial (memoria de microprograma o almacén de control) dentro del chip del procesador.

Suponiendo que se tienen n palabras de control y que una señal de control está representada por un bit en una palabra de control de n bits o microinstrucción. Una palabra de control está almacenada en la memoria de microprograma para cada paso en la ejecución de la secuencia de una instrucción. Una micro-rutina constituye una secuencia de microinstrucciones para una determinada instrucción de máquina.

El control microprogramado es simple de implementar y provee considerable flexibilidad; sin embargo es más lento que el control cableado. Este último se ha convertido en la elección preferida.

7.2. Segmentación

La segmentación (del inglés <<*pipeline*>>) es una forma de organizar la actividad actual de un sistema de computación, a nivel de instrucción, de forma que se permita la ejecución de más que una operación a la vez.

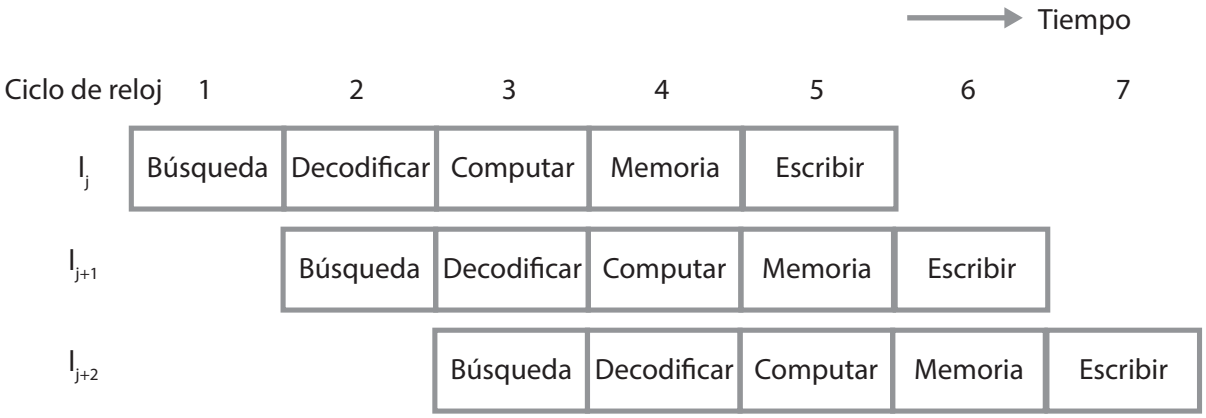


Figura 47. Caso ideal de la segmentación de la ejecución. Fuente: Adaptado desde (Hamacher, 2012).

La Figura 47 muestra el desarrollo de un caso ideal de la segmentación. Los cinco estados correspondientes a: búsqueda de la instrucción, decodificación, computación, memoria y escritura son presentados en una línea de tiempo, con un ciclo de reloj por estado. Como en una línea de ensamblaje, las instrucciones arriban a cada estado una a la vez; esto es, no es necesario esperar que una instrucción finalice los cinco estados antes de proceder con la siguiente instrucción. En este caso, lo que se hace es que una vez que la instrucción j pasa al segundo estado, se procede con la siguiente instrucción j+1, que arriba al estado 1, y así con cada una de las instrucciones y estados. En algún determinado tiempo, cada estado de la segmentación está procesando una instrucción diferente. Información como dirección de registro, dato inmediato y las operaciones a ser realizadas deben ser llevadas a lo largo de toda la segmentación, como una instrucción procede de un estado al próximo. Así, esta información es mantenida en los buffers inter estado, los cuales incluyen: RA, RB, RM RY, RZ, IR y el PC-Temp.

No obstante, en muchas ocasiones no es posible que una instrucción ingrese al proceso de segmentación en cada ciclo de reloj. Por ejemplo, en el caso de que el resultado del cálculo ejecutado en la instrucción j sea el dato fuente para la instrucción j+1. En este caso, la instrucción j+1 debe ser retrasada hasta que se den las condiciones necesarias para que continúe con el proceso segmentado; y esto con el resto de instrucciones. Las condiciones que causan este detenimiento o estancamiento del proceso de segmentación son llamadas como <<*riesgos*>>. El escenario anterior corresponde a un riesgo de datos; pero también se pueden dar a causa de retardos en memoria, instrucciones de bifurcación y limitaciones de recursos.

7.2.1. Dependencias de los datos

Como en el caso analizado en la sección precedente, existen operaciones en las que se requiere tener el resultado de una operación previa para poder continuar con la siguiente instrucción. Por ejemplo, en las instrucciones

add R1, R2, #20

add R4, R1, #10

Para ejecutar la segunda, se requiere que la primera instrucción haya sido completada, dado que su resultado R1 es un registro fuente para la segunda operación de adición. Esto se conoce como una *dependencia de datos* entre instrucciones. La Figura 48 muestra un mayor detalle del retardo producido.

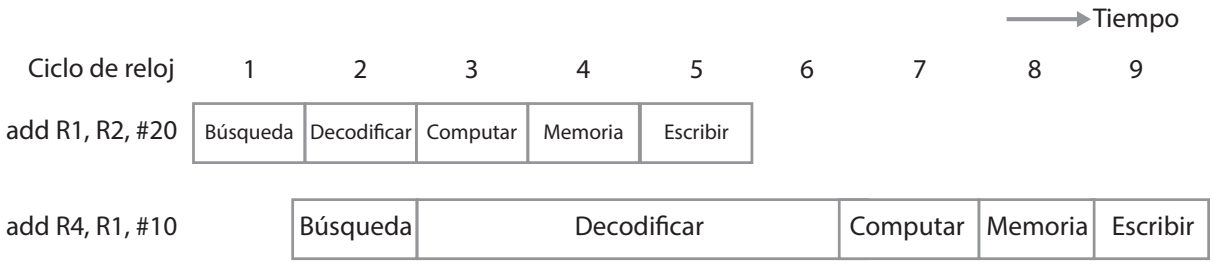


Figura 48. Retrazo en el proceso segmentado debido a una dependencia de datos. Fuente: Adaptado desde (Hamacher, 2012).

En el caso que se ilustra, en primera instancia, el circuito de control debe reconocer la dependencia de datos cuando decodifica la segunda instrucción de adición, en el ciclo 3, por comparar el identificador de su registro fuente, mantenido en el buffer inter estado correspondiente a la salida del circuito de búsqueda de instrucciones, con el registro destino de la primera instrucción de suma que está almacenada en el buffer inter estado correspondiente de la decodificación. Por lo tanto, la segunda instrucción de adición debe ser mantenida durante los ciclos del 3 al 5 hasta que se completa el proceso de la instrucción antecedente y en el ciclo 6 contar con el valor adecuado en el registro R1. Para ello, señales de control pueden ser configuradas en los buffer inter estado correspondientes para generar una instrucción implícita de no operación (NOP) que no modifique la memoria o el archivo de registro. Cada NOP crea un ciclo de reloj de tiempo de inactividad, denominado *burbuja*.

Este problema puede ser aliviado a través del *avance de operandos*. La Figura 49 muestra la idea central de este proceso. Como se analizó en el ejemplo anterior, la segunda operación de suma debía ser retardada por tres ciclos hasta que el valor de R1 esté disponible. No obstante, este valor ya está listo para ser usado al finalizar el ciclo 3, cuando la ALU completa su cómputo y carga este valor en el registro RZ (Figura 42). Más que retardar las segunda instrucción de suma, el hardware puede enviar el valor del registro RZ a donde se necesita en el ciclo 4, esto es la entrada del ALU.

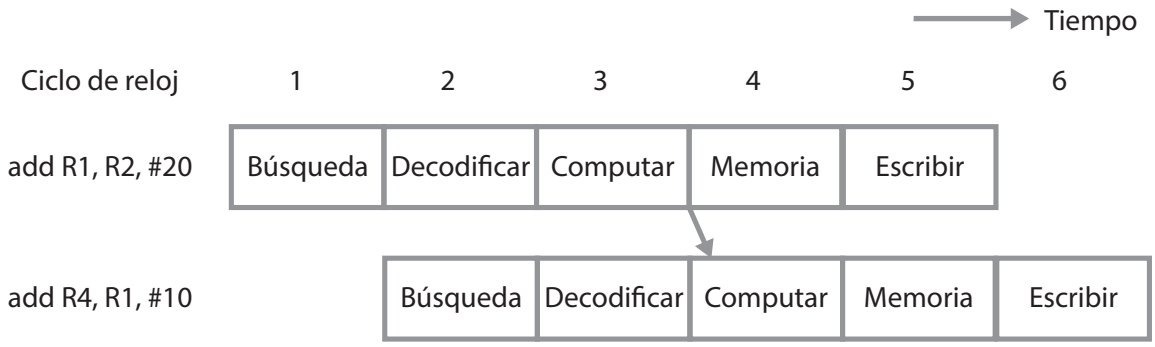


Figura 49. Uso del avance de operandos. Fuente: Adaptado desde (Hamacher, 2012).

Para ello, se puede desarrollar varias modificaciones al hardware mostrado en la Figura 42, de forma tal que los valores necesarios estén disponibles en las entradas del ALU para el cómputo necesario. Pero, también se puede llevar a cabo esta tarea a través de software. Una alternativa es permitir que esta

tarea sea desarrollada a través del compilador. Cuando el compilador detecta una dependencia de datos entre dos instrucciones sucesivas, puede insertar tres instrucciones NOP explícitas entre ellos. Esto permite que se produzca el retardo necesario para que la instrucción j+1 lea el nuevo valor desde el archivo de registros después de que éste es escrito. Una de las limitaciones de este procedimiento basado en software es el incremento del ancho del código y, por lo tanto, el tiempo de ejecución podría no ser reducido como en el caso del *avance de operandos*.

7.2.2. Retardos de memoria

El retardo que se puede producir en el acceso a memoria es otra de las formas de retardos en la segmentación. Por ejemplo, cuando la información o instrucción que se busca no se encuentra en la memoria caché, esto obliga pueda tomar más de un ciclo de reloj (un acceso a memoria puede tomar diez o más ciclos de reloj). Esto produce que todas las demás instrucciones sean retardadas.

Por ejemplo, con las instrucciones (Hamacher, 2012):

```
Load      R2, (R3)

Subtract  R9, R2, #30
```

Un proceso de avance de operando no puede ser desarrollado, dado que la lectura del dato desde memoria (el caché, en este caso) no estará disponible hasta que sea cargado en el registro RY (ver Figura 42) en el inicio del ciclo de reloj 5 (escribir) por lo que la instrucción debe estar retardada por un ciclo de reloj.

El compilador puede eliminar un ciclo de retardo para este tipo de dependencia de datos, reordenando las instrucciones de manera que se inserte una instrucción útil entre la instrucción de carga y la instrucción que depende del dato leído desde la memoria. Si una instrucción útil no puede ser encontrada por el compilador, el hardware introduce un ciclo de retardo automáticamente. Si el hardware del procesador no tiene mecanismos para hacer frente a las dependencias, entonces el compilador debe insertar una instrucción NOP explícita.

7.2.3. Retardos en bifurcaciones

Las bifurcaciones pueden tener un efecto en los procesos de segmentación dado que rompen la secuencia de las instrucciones. Por ejemplo, en el caso de las bifurcaciones no condicionales, la Figura 50 muestra la penalización de salto que puede suceder al ejecutar el proceso de segmentación con este tipo de instrucciones.

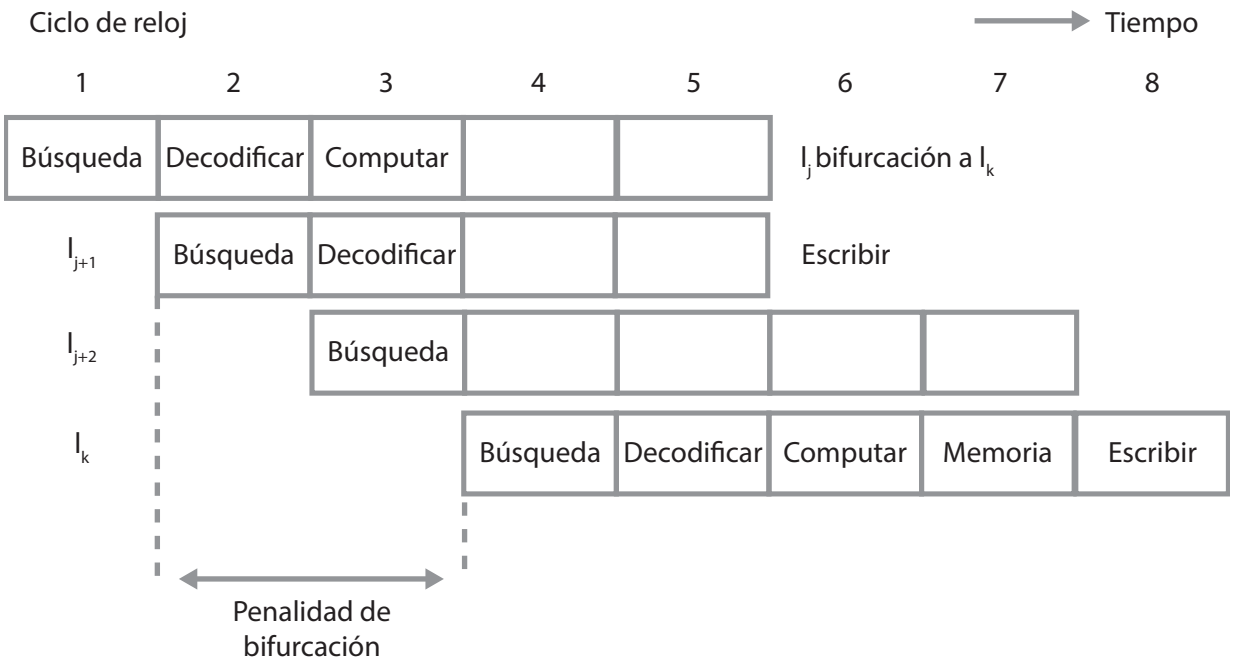


Figura 50. Penalidad de bifurcación cuando la dirección objetivo es determinada en el estado de cómputo de la segmentación. Fuente: Adaptado desde (Hamacher, 2012).

Como se puede observar, la instrucción Ij desarrolla un salto incondicional hacia la instrucción Ik. La búsqueda de la instrucción se hace en el ciclo 1, la decodificación en el ciclo 2, pero la determinación de la dirección objetivo, recién se computa en el ciclo 3. Así que la instrucción Ik es buscada en el ciclo 4. Las instrucciones Ij+1 e Ij+2 son buscadas en los ciclos 2 y 3 respectivamente, antes de que la instrucción de salto sea decodificada y la dirección de destino conocida. Por lo que estas deben ser descartadas. Como resultado los dos ciclos de retardo constituyen la *penalidad de bifurcación*.

Para disminuir este retardo, una aproximación es que la determinación de la dirección objetivo de la bifurcación no se desarrolle durante el estado de cómputo, sino que se anticipe y se ejecute durante el estado de decodificación. De esta forma, la búsqueda de la instrucción Ik puede ser desarrollada un ciclo antes, con lo cual solo la instrucción Ij+1 sería buscada en forma errónea y se reduce la *penalidad de bifurcación* a un solo ciclo de reloj.

En el caso de las bifurcaciones condicionales, para el proceso de segmentación, la condicion de bifurcación debe ser evaluada tan pronto como sea posible para limitar la *penalidad de bifurcación*. Para ello, el comparador que prueba la condición de bifurcación puede ser movido hacia el estado de decodificación, de forma que permita que la decisión de la bifurcación sea hecha al mismo tiempo que la dirección objetivo es determinada. De esta forma, se asegura que solo un ciclo de reloj es la penalidad de bifurcación para este tipo de instrucciones.

Otra técnica para mitigar los efectos de la bifurcación es la denominada *bifurcación retrasada*. La ubicación que sigue a la instrucción de bifurcación es conocida como *ranura de retardo de bifurcación*. Como se analizó previamente, si la condición de bifurcación en verdadera, existirá una penalización de un ciclo de reloj antes de que la instrucción Ik sea buscada; mientras que si la condición es falsa, la instrucción Ij+1 (aquella localizada luego de la instrucción de bifurcación Ij) será ejecutada y no

existirá penalización. Se puede aprovechar esta situación, de forma tal que la instrucción en la ranura de retardo sea siempre ejecutada. Para ello, se debe arreglar la disposición de las instrucciones de manera que se ubique en este espacio una instrucción útil que necesite ser ejecutada aun cuando la bifurcación se produzca. Si no se puede ubicar una instrucción útil en la ranura de retardo, una NOP debe ser ubicada en vez de ella.

La siguiente técnica se basa en la predicción de bifurcaciones. Para reducir aún más la penalidad de bifurcación, el procesador necesita anticipar que una instrucción que está siendo buscada es una instrucción de bifurcación y *predecir* su resultado para determinar cuál instrucción debería ser buscada durante el ciclo 2. Dos perspectivas son propuestas en la literatura: estática y dinámica.

En la *predicción estática* una de las formas más sencillas de realizar este proceso es asumir que la bifurcación no será tomada y que por tanto se buscará la siguiente instrucción en el orden secuencial. Si la predicción es correcta, la instrucción buscada será completada y no habrá penalización; sin embargo, si no es así, la penalización será completa. Para mejorar el rendimiento de este tipo de predicciones, es necesario considerar las características de los diferentes tipos de bifurcaciones. Por ejemplo, en el caso de los lazos, es más probable que la bifurcación sea tomada la mayor parte de las veces. De igual forma, bifurcaciones hacia adelante, hay una mayor certeza de que la opción más adecuada es de que no será tomada la instrucción siguiente. El procesador puede determinar una predicción estática de tomar o no, por verificar el signo del desplazamiento de bifurcación. Una alternativa es que la codificación de máquina de la instrucción de bifurcación incluya un bit que indique si debería ser predicha para ser tomada o no.

En el caso de la *predicción de bifurcación dinámica*, el hardware del procesador evalúa la probabilidad de que una bifurcación sea tomada por mantener un seguimiento de las decisiones de bifurcación cada vez que una instrucción de bifurcación es ejecutada. Una de las formas más simples es que el procesador asuma que la próxima vez que una instrucción es ejecutada, la decisión de bifurcación será la misma que la de última vez. En este caso, lo podemos ver como una máquina de dos estados, donde cada vez que la predicción es acertada, se mantiene la permanencia en ese estado y cuando falla pasa al otro estado. No obstante esta forma, aunque simple, puede llevar a disminución en el rendimiento; por ejemplo, en el caso de la última instrucción de un lazo, que hará que pase al otro estado y cuando se produzca una nueva instrucción de lazo se producirá un error. Una forma de mejorar el rendimiento de la predicción es mantener una mayor información acerca de la ejecución histórica de la instrucción.

7.2.4. Operación superescalar

Otra forma de incrementar el rendimiento es proporcionar al procesador de múltiples unidades de ejecución, cada una con la capacidad de segmentación y de esta forma incrementar la capacidad de procesador de gestionar varias instrucciones en paralelo. Con ello, varias instrucciones empiezan su ejecución en el mismo ciclo de reloj pero en diferentes unidades de ejecución. Estos son conocidos como procesadores *superescalar*.

Para ello se tiene unidades de búsqueda mucho más complejas que permiten buscar dos o más instrucciones por ciclo antes de que sean necesitadas y las ubican en una cola de instrucciones. Una

unidad diferente, conocida como *unidad de despacho*, toma dos o más instrucciones desde el frente de la cola de instrucciones, las decodifica, y envía a la unidad de ejecución apropiada. Al final de la segmentación, otra unidad es encargada de escribir el resultado dentro del archivo de registro. La Figura 51, muestra un procesador superescalar con dos unidades de ejecución.

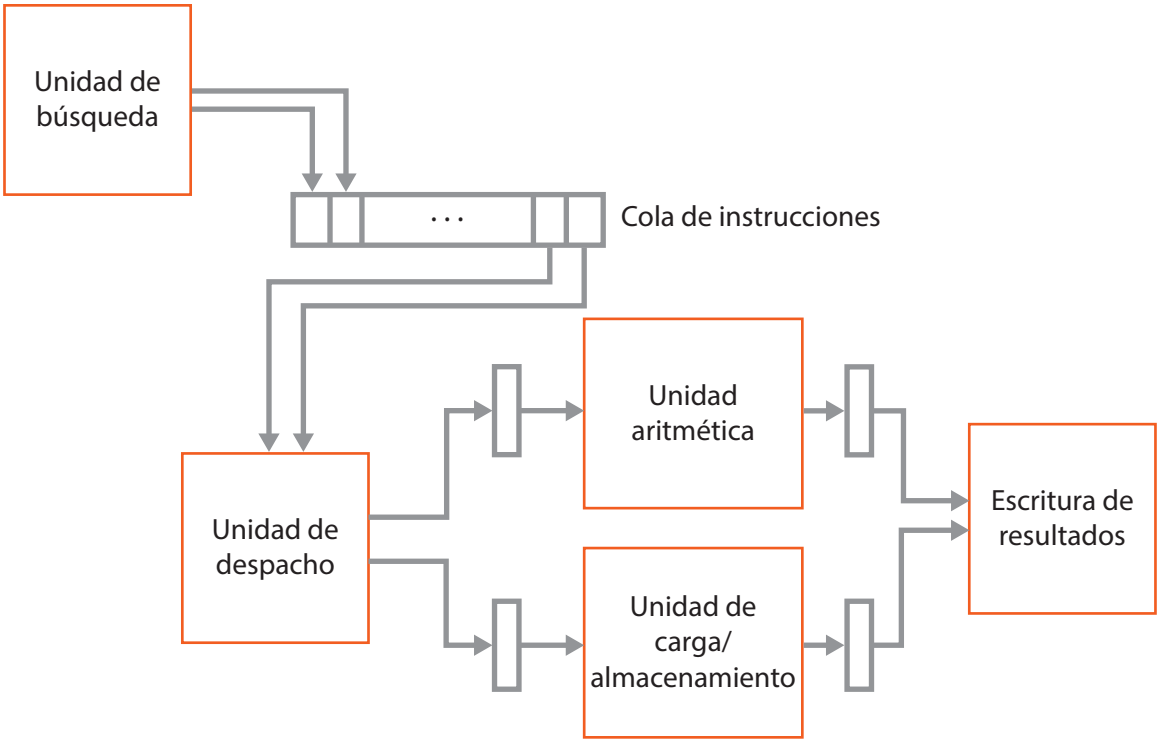


Figura 51. Un procesador superescalar con dos unidades de ejecución. Fuente: Adaptado desde (Hamacher, 2012).

La presencia de instrucciones de bifurcación y la dependencia de datos imponen restricciones de ordenamiento secuencial. Además, el retardo de memoria puede producir, ocasionalmente, el retraso de la búsqueda y despacho de las instrucciones. Un procesador superescalar debe asegurar que las instrucciones sean ejecutadas en una secuencia adecuada. Para hacer frente a esta problemática, se requiere del uso de hardware adicional. Una combinación de *predicción* junto con una técnica denominada *ejecución especulativa* es empleada. Esta técnica se basa en buscar las instrucciones subsecuentes basadas en una predicción no confirmada, despachadas y posiblemente ejecutadas. Estas tienen un estatus de como especulativas por lo que estas instrucciones y sus resultados pueden ser descartados en caso de que la predicción no sea correcta. Se requiere de hardware que permita mantener la información acerca de las instrucciones ejecutadas especulativamente y asegurar que los registros y la memoria no son modificados hasta que la predicción sea confirmada.

Otro problema puede surgir con la ejecución de instrucciones fuera de orden. Por ejemplo, cuando se considera la posibilidad de que una instrucción genere una excepción. Pero la estructura de instrucciones en paralelo puede haber modificado la causa de la excepción y ocultado el problema, con lo que se llega a una situación en el que el procesador tiene *excepciones imprecisas*. Para evitar esta situación, es necesario garantizar el estado de consistencia cuando ocurren excepciones. Los resultados de la ejecución de una instrucción debe ser escrita en su localización de destino es el

estricto orden del programa. Si una excepción ocurre durante la ejecución de una instrucción, todas las instrucciones subsecuentes y sus resultados almacenados son descartados.

Glosario

Acumulador

Es un registro en el que son almacenados temporalmente los resultados aritméticos y lógicos intermedios que serán tratados por la ALU.

Arquitectura del computador

Una interfaz abstracta entre el hardware y el software de nivel más bajo que abarca toda la información necesaria para escribir un programa de lenguaje de máquina que se ejecutará correctamente, incluyendo instrucciones, registros, acceso a memoria, E / S, etc.

Bus

Una ruta de comunicación compartida que consta de más de una línea, ya que las rutas son compartidas, solo un dispositivo puede transmitir a la vez.

Circuitos integrados

También llamado chip. Un dispositivo que combina de docenas a millones de transistores.

Compilador

Un programa que traduce las declaraciones de lenguaje de alto nivel en declaraciones de lenguaje ensamblador.

Computador embebido

Una computadora dentro de otro dispositivo utilizado para ejecutar una aplicación predeterminada o una colección de software.

Computador personal

Una computadora diseñada para ser utilizada por un individuo, generalmente incorporando una pantalla gráfica, un teclado y un mouse.

Computadora en red

Combinan un gran conjunto de computadoras personales y dispositivos de almacenamiento en una red de alta velocidad distribuida físicamente, los cuales son gestionados como un recurso de computación

Computadoras

Máquina electrónica que realiza tareas de procesamiento, almacenamiento y movimiento de datos junto con el control de los procesos requeridos.

Conjunto de instrucciones

El vocabulario de los comandos entendidos por una arquitectura determinada.

Dirección

Un valor utilizado para delinear la ubicación de un elemento de datos específico dentro de una matriz de memoria.

Ensamblador

Un programa que traduce una versión simbólica de instrucciones en la versión binaria

Estructura del computador

Hace referencia a los elementos operacionales y sus interconexiones, que dan cuenta de las especificaciones de la arquitectura.

Estructura Harvard

Una estructura de programa almacenado en la cual hay un solo procesador que opera secuencialmente en datos que están almacenados en la misma memoria física y en el mismo formato que las instrucciones

Estructura Von Neumann

Una estructura que hace uso de un solo espacio de memoria para el almacenamiento de instrucciones y datos.

FPGA

Es un dispositivo programable que contiene bloques de lógica cuya interconexión y funcionalidad puede ser configurada en el momento mediante un lenguaje de descripción especializado

Lenguaje de máquina

Una representación binaria de instrucciones de la máquina.

Lenguaje ensamblador

Una representación simbólica de las instrucciones de la máquina.

Lenguajes de alto nivel

Un lenguaje portátil como C, Java que está compuesto por palabras y notación algebraica que puede ser traducida por un compilador al lenguaje ensamblador.

Memoria del computador

El área de almacenamiento en la que se guardan los programas cuando se están ejecutando y que contienen los datos necesarios para los programas en ejecución.

Pila

Una estructura de datos para el derrame de registros organizada como cola de último en entrar primero en salir.

Procesador

La parte activa de la computadora, que contiene la ruta de datos y el control y que agrega números, números de prueba, señales de dispositivos de E / S para activar, y así sucesivamente.

Programas

Secuencia de instrucciones escritas para realizar una tarea específica en una computadora.

Semiconductor

Una sustancia que no conduce bien la electricidad.

Servidor

Una computadora utilizada para ejecutar programas más grandes para múltiples usuarios, a menudo simultáneamente, y por lo general solo se accede a través de una red.

Sistema operativo

Programa supervisor que administra los recursos de una computadora para el beneficio de los programas que se ejecutan en esa computadora.

Supercomputadora

Una clase de computadoras con el mayor rendimiento y costo; están configurados como servidores y generalmente cuestan decenas o cientos de millones de dólares.

Transistor

Un interruptor de encendido / apagado controlado por una señal eléctrica.

Unidad de control

El componente del procesador que controla los datos, la memoria y los dispositivos de E / S de acuerdo con las instrucciones del programa.

Unidad de entrada/salida

Elemento que proporciona un método de comunicación eficaz entre el sistema central y el periférico.

Unidad lógica y aritmética

Circuito digital que calcula operaciones aritméticas: suma, resta, multiplicación, etc., y operaciones lógicas: and, or, not, entre valores de los argumentos.

Enlaces de interés

CS-224 Computer Organization Lecture 01

Grabaciones de las clases del curso <<Organización de Computadoras>> dictado por el profesor William Sawyer de la Bilkent University. Son alrededor de 40 lecturas en las cuales se explican en forma muy amplia y clara los fundamentos de la organización de computadoras.

https://www.youtube.com/watch?v=CDO28Esqmcg&index=1&list=PLhwVAYxlh5dvB1MkZrcRZy6x_a2yORNAu

Computer System Archictecture

Curso abierto sobre arquitectura de sistemas de computadoras, dictado por los profesores Joel Emer, Krste Asanovic y Arvind, del Instituto Tecnológico de Massachusetts. Presenta un amplio material para lectura y comprensión de la arquitectura de computadoras

https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-823-computer-system-architecture-fall-2005/

Universidad Carlos III de Madrid - Open Course Ware

Curso abierto de organización de computadores dictado por profesores de la Universidad Carlos III de Madrid. Presenta un conjunto amplio de materiales y ejercicios. Está en español.

http://ocw.uc3m.es/ingenieria-informatica/organizacion-de-computadores

SPIM Simulator

Página donde se encuentra información sobre el simulador SPIM.

http://spimsimulator.sourceforge.net/

Bibliografía

Referencias bibliográficas

David, A Patterson and John, L. H. (2005). *Computer organization and design: the hardware/software interface*. San mateo, CA: M organ Kaufmann Publishers.

Hamacher, C. (2012). *Computer organization and embedded systems*. McGraw-Hill Education.

Harris, David and Harris, S. (2010). *Digital design and computer architecture*. Morgan Kaufmann.

Hennessy, John L and Patterson, D. A. (2011). *Computer architecture: a quantitative approach*. Elsevier.

Patterson, D., & Hennessy, J. (2017). *Computer Organization and Design RISC-V Edition*. Morgan Kaufmann.

Stallings, W. (2013). *Computer Organization and Architecture: Designing for Performance (9th Edition)*. Pearson Education.

Tanenbaum, A. S. (2016). *Structured computer organization*. Pearson Education India.

Bibliografía recomendada

Tarnoff, D., & Edition, R. F. (2005). Computer Organization and Design Fundamentals.

Agradecimientos

Autor
Dr. D. Jack Fernando Bravo Torres

Departamento de Recursos para el Aprendizaje
D.ª Carmina Gabarda López
D.ª Cristina Ruiz Jiménez
D.ª Sara Segovia Martínez