GRADO EN INGENIERÍA INFORMÁTICA

Módulo Común de la Rama de Informática

ARQUITECTURA DE COMPUTADORES

George Enrique Figueras Benítez

viu Universidad Internacional de Valencia

Edita
Universidad Internacional de Valencia

Grado en

Ingeniería Informática

Arquitectura de Computadores

Módulo Común de la Rama de Informática

6 ECTS

George Enrique Figueras Benítez

Leyenda

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

Enlace de interés
Dirección de página web.

Indice

1. FUNDAMENTOS DE DISEÑO Y EVALUACIÓN DE COMPUTADORES 7
1.1. Definiciones básicas de la arquitectura del computador 7
1.2. Fundamentos de diseño 11
1.3. Fundamentos de evaluación del rendimiento 13
1.3.1. Métricas del rendimiento 14
1.3.2. Evaluación del rendimiento mediante programas 15
1.3.3. Tendencias en el consumo energético 17

2. LENGUAJE ENSAMBLADOR 19
2.1. Formato de las declaraciones en ensamblador 20
2.2. Macros 22
2.3. El ensamblado 22
2.4. Enlazado y carga 24
2.5. Lenguaje ensamblador en la arquitectura ARM 25
2.5.1. Instrucciones básicas 25
2.5.2. Instrucciones avanzadas 26
2.5.3. Instrucciones de bifurcación 27
2.5.4. Instrucciones de memoria 29
2.5.5. Codificación de las instrucciones 30

3. JERARQUÍA DE MEMORIA 33
3.1. Tipos de memoria en la jerarquía 35
3.2. Memoria caché 36
3.2.1. Tipos de direccionamiento 38
3.2.2. Políticas de emplazamiento 39
3.2.3. Políticas de reemplazamiento 40
3.2.4. Políticas de actualización 41
3.2.5. Otros parámetros de diseño 41
3.3. Memoria virtual 43
3.3.1. Mecanismos de traducción de direcciones 43
3.3.2. Memoria virtual paginada 44

3.3.3. Memoria virtual segmentada 45
3.3.4. Memoria virtual segmentada-paginada 45

4. SISTEMAS DE ALMACENAMIENTO 47
4.1. Discos magnéticos 47
4.2. Arreglo redundante de discos independientes 50
4.3. Dispositivos ópticos 51
4.4. Dispositivos de estado sólido 53

5. REPERTORIO DE INSTRUCCIONES 55
5.1. Parámetros de diseño 57
5.1.1. Tipos de instrucciones 57
5.1.2. Tipos de datos 59
5.1.3. Registros 59
5.1.4. Modos de direccionamiento 61
5.1.5. Formato de las instrucciones 65

6. SEGMENTACIÓN Y PARALELISMO EN EL DISEÑO DE COMPUTADORES 69
6.1. Principios de segmentación 69
6.2. Principios del diseño de computadores segmentados 71
6.3. Segmentación de instrucciones en arquitectura RISC 73
6.4. Atascos de un cauce 74
6.5. Arquitectura segmentada superescalar 76
6.6. Procesamiento multihilos 79
6.7. Procesadores multinúcleos 81
6.8. Jerarquía de la memoria caché en procesadores multinúcleos 82
6.8.1. Coherencia de la caché 82
6.8.2. Sincronización de la caché 84
6.8.3. Consistencia de la caché 85

GLOSARIO 87
ENLACES DE INTERÉS 89
BIBLIOGRAFÍA 91

1. Fundamentos de diseño y evaluación de computadores

1.1. Definiciones básicas de la arquitectura del computador

El diseño de un computador puede verse como un conjunto de niveles relacionados entre sí que pueden ser optimizados para cumplir con los requerimientos de desempeño deseados. Dentro de los diferentes niveles encontramos: dispositivos semiconductores, lógico, microarquitectura, arquitectura, sistema operativo, lenguajes de alto nivel, algoritmo y finalmente de aplicaciones. Según la aplicación final que tendrá el computador se hace necesario definir las características de los parámetros de diseño. Es importante realizar la distinción entre dos términos que usualmente tienden a mezclarse, ellos son: **la arquitectura y la organización del computador**. El primero de ellos representa las especificaciones sobre el funcionamiento del computador que definen la manera en que deben implementarse los *softwares*; mientras que el segundo representa la implementación del computador a nivel de *hardware*. La arquitectura del computador define esencialmente la funcionalidad o los requerimientos de funcionalidad en el computador, el cual se especifica mediante la **arquitectura del repertorio de instrucciones** (**ISA**, por sus siglas en inglés). Por otra parte, se tiene la **microarquitectura**, la cual es responsable de la interconexión de las diferentes unidades funcionales encargadas de llevar a cabo la ISA, así como de su implementación de la manera más eficiente.

La implementación eficiente de la ISA dependerá en gran medida del tipo de computador que se pretende diseñar, ya que cada uno de estos tendrá diferentes requerimientos funcionales y por ende estarán diseñados para optimizar diversas métricas de su rendimiento. **Alguno de los**

tipos de computadores más comunes disponibles hoy en día son los dispositivos móviles, las computadoras personales, servidores y los computadores embebidos. En el caso de los dispositivos móviles los principales criterios de diseño son: la capacidad de respuesta, la predictividad y sobre todo la eficiencia energética. Es por ello que en el diseño de estos dispositivos se priorizan los elementos que permitan incrementar el desempeño de la eficiencia energética. Por otra parte, cuando se desea diseñar un servidor, los requerimientos prioritarios son la disponibilidad y escalabilidad, debido a dichos requerimientos los diseñadores tienen como objetivo optimizar el rendimiento.

Según la clasificación de Flynn para la arquitectura de los computadores se tienen cuatro grupos diferentes:

- **Única instrucción-única secuencia de datos** (**SISD**, por sus siglas en inglés)
- **Única instrucción-múltiples secuencias de datos** (**SIMD**, por sus siglas en inglés)
- **Múltiples instrucciones-única secuencia de datos** (**MISD**, por sus siglas en inglés)
- **Múltiples instrucciones-múltiples secuencias de datos** (**MIMD**, por sus siglas en inglés)

En la arquitectura SISD cada instrucción opera sobre una sola secuencia de datos, de esta manera un único procesador interpreta una única secuencia de instrucciones. En este grupo es posible emplear el paralelismo a nivel de instrucciones para mejorar el desempeño. Un ejemplo típico de esta clase de arquitectura lo constituyen los computadores basados en el modelo de Von Neumann o los de un único procesador.

En la arquitectura SIMD una instrucción opera sobre múltiples secuencias de datos controlando múltiples unidades de proceso; en esta se puede implementar el paralelismo a nivel de los datos para incrementar el desempeño. Algunos ejemplos de esta clase son la arquitectura vectorial, los arreglos de procesadores y las unidades de procesamiento gráfico, entre otras.

En la arquitectura (MISD) una única secuencia de datos es ejecutada mediante un conjunto de procesadores. Hasta el momento no existe ninguna implementación viable de sistemas basado en esta clase.

Finalmente, en la arquitectura MIMD se tienen múltiples procesadores que ejecutan un conjunto de instrucciones que operan sobre diferentes flujos de datos de manera simultánea. En esta categoría se puede implementar el paralelismo a nivel de procesos para incrementar el desempeño del computador. Los sistemas multiprocesadores constituyen un ejemplo importante dentro de este tipo de arquitectura.

La arquitectura del computador está compuesta por la interrelación de la ISA, la microarquitectura y la implementación de estas; de aquí la importancia que tiene cada uno de estos elementos en los principios que rigen el diseño del computador. A partir de las especificaciones de las aplicaciones en que se emplearán los computadores, se definen los requerimientos funcionales y la ISA apropiada para emplearse en cada caso. Una vez elegida la ISA, dentro de todas las microarquitecturas disponibles para esta ISA se selecciona la más eficiente. Finalmente, para la microarquitectura fijada es necesario realizar la implementación más adecuada en dependencia de los requerimientos planteados; de esta

forma, se cuenta con múltiples opciones de diseños donde se puede estructurar una arquitectura eficiente que posibilite la implementación de un computador eficiente que cumpla con las exigencias de las aplicaciones que serán utilizadas.

Otro tipo de clasificaciones de la ISA está relacionado con el tipo de instrucciones que estas pueden soportar; pudiéndose tener cuatro tipos diferentes:

- De **pila**
- De acumulador
- De registro-memoria
- De registro-registro

La arquitectura de pila esencialmente soporta las operaciones aritméticas, conjuntamente con otras dos operaciones que se realizan con la pila: *push* y *pop*.

La arquitectura de acumulador consta de operandos implícitos para la acumulación, para ello se dispone de un tipo de registro especial denominado acumulador donde puede almacenarse uno de los operandos empleados en una instrucción y luego almacenar el resultado de esta.

En la arquitectura registro-memoria, las operaciones y los resultados de estas pueden emplear operandos almacenados indistintamente en cualquiera de los elementos que dan nombre a dicha arquitectura. Un ejemplo representativo de este tipo de arquitectura lo constituyen los computadores **IBM360** y su sucesor el **Intelx86** (Flynn, 1995).

La arquitectura de registro-registro, conocida también como arquitectura de carga-almacenamiento en registro, **es similar a la arquitectura de registro-memoria con la diferencia de que en esta las operaciones de la Unidad Aritmética Lógica** (**ALU**, por sus siglas en inglés) **no pueden realizarse en la memoria**; dicho de otra forma, tanto los operandos como los resultados de las operaciones deben ser almacenados en registros. En esta última clase las operaciones en la ALU poseen un formato más uniforme. La mayor rapidez de los registros respecto a las memorias permite que mediante estos puedan realizarse las operaciones en la ALU de una forma muy rápida, propiciando la tendencia de implementar un número creciente de registros de propósito general. Otras de las ventajas que ofrecen los registros son: un manejo más sencillo por parte de los compiladores en comparación con la pila, reducen el tráfico de la memoria, requieren la misma cantidad de tiempo para la ejecución del mismo tipo de instrucciones en la ALU, lo que facilita al compilador la segmentación y optimización, entre otras.

Todas estas ventajas han hecho que la ISA de carga-almacenamiento en registro sea la más popular de todas las arquitecturas, dentro de esta categoría se encuentran los **computadores con repertorio de instrucciones reducido** (**RISC**, por sus siglas en inglés) como el **PowerPC**, **SPARC**, **RISC-V**, **ARM** y el **MIPS** (Flynn, 1995). Un factor importante a tener en cuenta a la hora de elegir una de las arquitecturas que involucran las operaciones con registros (registro-memoria y registro-registro) es la cantidad de registros requeridos, lo cual dependerá considerablemente de la calidad del compilador.

Es posible además clasificar las distintas ISA que involucran operaciones con registros (registro-memoria y registro-registro) en dependencia del número y el tipo de operandos involucrados en las instrucciones que se ejecutan en la ALU. Se pueden tener dos o tres operandos y algunos de ellos pueden almacenarse en registros mientras que otros pueden almacenarse en la memoria. Teniendo en cuenta lo antes mencionado se tienen las siguientes clases de ISA:

- Ningún operando en memoria y tres en registro
- Un operando en memoria y uno en registro
- Dos operandos en memoria y ninguno en registro
- Tres operandos en memoria y ninguno en registro

La ISA más popular dentro de la categoría registro-registro es la que posee ningún operando en memoria y tres en registro, dos de estos destinados para los datos fuentes y el tercero para el resultado; esta puede encontrarse en los computadores PowerPC, SPARC, ARM y el MIPS (Flynn, 1995). La categoría de un operando en memoria y uno en registro se implementa en el Intel x86 pudiéndose emplear cualquiera de ellos indistintamente para el dato fuente y el otro para el resultado. Las otras dos clasificaciones (dos operandos en memoria y ninguno en registro, tres operandos en memoria y ninguno en registros) han sido implementadas en el pasado en computadores como el VAX. Cada una de estas categorías tiene sus propias ventajas y desventajas las cuales tienen implicaciones en el tamaño de las instrucciones, la densidad del código, facilidad de segmentación, tiempo de ejecución, número de bits requeridos para la codificación de los registros, entre otros aspectos. Dada la importancia y gran diversidad de configuraciones posibles, la selección de la ISA constituye el punto de inicio a partir de dónde empezar el diseño de un procesador; no obstante, existen otros parámetros de importancia a tener en cuenta.

Existen diferentes características deseables para toda ISA, dentro de las cuales se tienen: que sea completa, concisa, genérica y simple. De todas estas características la más crítica es que sea completa; lo que implica que la arquitectura diseñada contenga todas las instrucciones necesarias para implementar cualquier tipo de programa. Este criterio plantea varios interrogantes a la hora de definir la ISA, como, por ejemplo: el número de instrucciones necesarias, su funcionamiento específico y su complejidad.

La manera de abordar los interrogantes anteriores ha dado lugar al desarrollo de dos paradigmas diferentes para el diseño de la ISA:

- **La arquitectura RISC**
- **La arquitectura de computadores con repertorio de instrucciones complejo** (**CISC**, por sus siglas en inglés)

En la arquitectura RISC se dispone de muy pocas instrucciones, relativamente simples y con una estructura muy regular que permite un fácil manejo por parte del procesador. Los procesadores ARM, de amplia utilidad en los dispositivos móviles, son un ejemplo de este tipo de arquitectura.

Por otra parte, se tiene el **paradigma CISC, donde los procesadores disponen de un repertorio con un mayor número de instrucciones que poseen funcionalidades más complejas**. Un ejemplo de esta arquitectura son los procesadores Intel y AMD.

1.2. Fundamentos de diseño

Uno de los principios fundamentales a tener en cuenta para crear una arquitectura del computador eficiente es emplear las ventajas que ofrece el paralelismo. Este parámetro puede encontrarse en múltiples niveles dentro del diseño del computador, desde el más elemental a nivel de bits hasta el más complejo a nivel de procesos. El paralelismo a nivel de bits puede emplearse en el caso de que por ejemplo se realicen operaciones de cuatro bits de forma paralela como parte de operaciones de 8 bits; como resultado se obtendrá una mejora en el desempeño.

A nivel de instrucciones, **el paralelismo hace uso del hecho de que las instrucciones pueden ser independientes una de las otras en algunos escenarios específicos, e inclusive siendo dependientes se pueden superponer sus operaciones posibilitando que la ejecución de estas se solape**; siendo un ejemplo de la arquitectura SISD. Este solapamiento se logra debido a las diferentes etapas en que se subdivide el procesamiento de una instrucción: búsqueda, decodificación, ejecución y almacenamiento de los resultados. De esta forma, cuando una instrucción se encuentra en una etapa avanzada de su ejecución, la siguiente instrucción puede encontrarse en una etapa posterior posibilitando su solapamiento; a esta técnica se le conoce como **segmentación**. Este concepto permite mejorar el desempeño del procesador específicamente en el caso de que las instrucciones sean de cierta forma dependientes unas de las otras. En el caso de que las instrucciones sean completamente independientes y si el procesador posee múltiples unidades funcionales, es posible realizar cada una de estas instrucciones de manera independiente mediante procesadores superescalares.

Un nivel de paralelismo superior se tiene a nivel de datos. Este se basa en que no siempre las operaciones requieren utilizar datos de un tamaño fijo. Por ejemplo, las aplicaciones gráficas no siempre realizan operaciones con datos de 32 bits, la mayor parte del tiempo estas se ejecutan a nivel de bits. Este tipo de paralelismo es un ejemplo de la arquitectura SIMD pudiéndose realizar las operaciones sobre una cadena de datos.

El último ejemplo de paralelismo se tiene a nivel de proceso, en este se utiliza la posibilidad que ofrecen las aplicaciones en dividirse en múltiples tareas independientes. La ejecución de estas múltiples tareas en un *hardware* con la capacidad del procesamiento simultáneo posibilita el incremento del rendimiento proporcionalmente al número de procesos que se pueden realizar en paralelo. Este tipo de arquitectura es un ejemplo de la MIMD. Para lograr los beneficios que ofrecen cada uno de los métodos de paralelismo mencionados anteriormente es necesario que la arquitectura (ISA, microarquitectura y *hardware*) empleada proporcione los mecanismos necesarios para el correcto funcionamiento de dichos métodos.

Otro principio relevante a tener en cuenta en el diseño del computador es el principio de localidad, tanto en el caso temporal como en el espacial. Este principio se aplica no solo para el diseño de la memoria caché como un componente esencial dentro de la jerarquía de memoria, sino que es aplicable para muchos otros contextos donde se necesite el manejo de datos; es por ello que la

arquitectura del computador debe soportar la memoria caché y una configuración óptima de esta que proporcione el mejor desempeño posible. No por último es menos importante el "**principio del caso más común**"; siendo este el principal responsable de las mejorías más significativas en el desempeño del computador como resultado de un diseño eficiente. Este principio sugiere que se dediquen los mayores esfuerzos a optimizar los componentes de la arquitectura que mayor uso tienen, en vez de dedicarle estos esfuerzos a la optimización de aquellos que rara vez son utilizados.

Cuando se realiza una mejora en la arquitectura del computador, el repertorio de instrucciones, la implementación del *hardware* o el compilador para acelerar la ejecución de los programas es importante tener en cuenta la **ley de Amdahl** que establece un límite en el impacto que tiene una mejora determinada para incrementar el rendimiento del computador, la cual estará relacionada con el uso que se le da a la característica mejorada. Dicho de otra forma, esta ley permite cuantificar el impacto en el desempeño como resultado de la optimización de un componente específico que posibilita validar diferentes alternativas de diseño. Para entender el alcance de esta ley es necesario primero definir el significado de la **mejora total** en la ejecución de los programas; esta se define como la razón entre los tiempos de ejecución de un programa antes ($Tiemp\ Eje_{inicial}$) y después ($Tiemp\ Eje_{final}$) de realizada la modificación en el computador. De forma equivalente **podemos determinar la mejora a partir de la razón entre el rendimiento luego de la modificación y antes de esta**.

$$mejora\ total = \frac{Tiemp\ Eje_{inicial}}{Tiemp\ Eje_{final}}$$

Por lo general una modificación en el computador solo influirá en el rendimiento de una parte del programa ($Tiemp\ Eje_{afectado}$), de esta forma siempre existirá un tiempo de ejecución que se mantendrá inalterado ($Tiemp\ Eje_{Noafectado}$). Es por ello que la mejora total del rendimiento estará limitada por la fracción del tiempo en que se ejecutan las instrucciones favorecidas por la modificación realizada; por lo que resulta más conveniente introducir pequeñas mejoras en los elementos que se usan con mayor frecuencia o hacer más rápido el caso común (Lanchares Dávila, 2000).

Otra forma de expresar la ley de Amdahl es a partir de la *mejoraparcial* resultante de la mejoría introducida y de la fracción del tiempo en que se utiliza dicha mejoría ($Frac\ Tiempo_{afectado}$) como se aprecia en la siguiente expresión,

$$mejora\ total = \frac{1}{(1 - Frac\ Tiempo_{afectado} + \frac{Frac\ Tiempo_{afectado}}{mejoraparcial}}$$

A partir de la *mejoraparcial* y de los intervalos del tiempo de ejecución, tanto para la parte del programa que se ve afectada por la mejoría introducida como la que no, se puede determinar el tiempo ejecución resultante como:

$$Tiemp\ Eje_{final} = Frac\ Tiempo_{Noafectado} + \frac{Tiemp\ Eje_{afectado}}{mejoraparcial}$$

Para ilustrar mejor las implicaciones de la ley de Amdahl analicemos el siguiente ejemplo: supongamos que un computador requiere 100 ns para la ejecución de un programa, en dicho computador se realiza una modificación que implica una mejora parcial del rendimiento por un factor de 2 en una parte del programa que consume el 75% del tiempo de ejecución; como resultado, se tiene que la mejora total en el rendimiento de este computador para el programa dado es igual a 1.6; o sea, que se requiere 1.6 veces menos tiempo para la ejecución de dicho programa.

1.3. Fundamentos de evaluación del rendimiento

Debido a la gran variedad de técnicas que pueden ser utilizadas en las diferentes arquitecturas empleadas para la fabricación de los computadores, se hace necesario valorar el impacto que tiene estas en el rendimiento del computador. De esta forma no solo es posible evaluar el rendimiento de una nueva arquitectura por parte de los diseñadores; sino que, además, es posible que los usuarios puedan evaluar diferentes tipos de computadores con el objetivo de seleccionar el más adecuado para sus necesidades. **Teniendo en cuenta lo antes mencionado, el rendimiento puede ser evaluado desde una perspectiva del usuario final que utilizará el computador o desde el punto de vista del diseñador que le permita comprobar qué alternativa de diseño es la que brinda mejores resultados.** En cualquiera de estas dos perspectivas es necesario establecer métricas que permitan comparar cuantitativamente el rendimiento de diferentes computadores o de opciones de diseño en un mismo computador.

Desde el punto de vista del usuario final es deseable que el computador brinde el mejor desempeño al menor coste posible, lo que ofrece diferentes formas de relacionar estos dos parámetros (precio-rendimiento). Por una parte, se puede determinar el mejor desempeño para un precio dado, o la razón entre el desempeño y el precio, o la variante que garantiza el mejor desempeño para un mismo precio.

Desde el punto de vista del diseñador existen diferentes variantes de repertorios de instrucciones y múltiples maneras de implementarlas a nivel de *hardware*, lo que repercute directamente en el desempeño del computador y su coste asociado. De manera similar a los usuarios finales, los diseñadores también se encuentran ante la disyuntiva de optimizar el desempeño y el coste teniendo diferentes opciones como por ejemplo el repertorio de instrucciones y el *hardware* que brinden el mejor desempeño para un precio fijo, o las opciones más económicas para un desempeño dado. La respuesta a estos interrogantes requiere definir el desempeño de una manera precisa, mediante parámetros cuantitativos que puedan ser medibles, reportables y reproducibles.

Un aspecto importante a la hora de especificar el desempeño de los computadores es que este dependerá de la forma en que se fijen los diferentes criterios posibles a tener en cuenta en la evaluación. Teniendo en cuenta esto, en algunos casos una opción determinada pudiera tener un mejor desempeño respecto a un criterio específico, mientras que para otro criterio pudiera existir otra opción más adecuada. Otro aspecto importante a considerar es el tipo de programa que se utilice a la hora de evaluar el efecto que tienen diferentes tipos de *hardware* o de ciertas opciones de diseños en el rendimiento de un computador. En algunos casos un programa puede tener un mejor comportamiento en un computador respecto a otros, mientras que otros programas pueden ser peores en ese mismo computador.

En el contexto del rendimiento de los computadores dos criterios de vital importancia son **el tiempo de respuesta o latencia** y **el rendimiento**. Este último es un término general que puede reflejar la cantidad de tareas a ejecutarse simultáneamente, el número de tareas realizadas por unidad de tiempo y la tasa de ejecución promedio de tareas. Las anteriores métricas de rendimiento en algunos casos pueden contraponerse, por ejemplo: desde el punto de vista de los usuarios es deseable que el tiempo de respuesta sea pequeño; mientras que, en un servidor web es más crítico ofrecer servicio a una mayor cantidad de usuarios pasando a un segundo plano el tiempo requerido por cada usuario individual. Algunas opciones de diseño pueden mejorar simultáneamente el comportamiento tanto del tiempo de respuesta como del rendimiento y en otras es necesario sacrificar una de ellas para mejorar la otra. Es por ello que es importante tener en cuenta cual es el objetivo principal que se desea, de forma tal que se pueda priorizar una métrica respecto a la otra.

1.3.1. Métricas del rendimiento

Una forma común de definir el rendimiento de un computador de manera individual se obtiene a partir del recíproco del tiempo de respuesta; de esta forma, mientras menor sea el tiempo requerido para ejecutar una tarea específica mejor será el rendimiento del computador. El tiempo de respuesta puede definirse de varias formas; una de ellas resulta de la suma del tiempo de ejecución de la **unidad central de procesamiento** (**CPU**, por sus siglas en inglés), el tiempo requerido por los dispositivos de entrada/salida (**E/S**) y **el tiempo de espera en ambientes multitarea. De manera estricta se puede considerar que el tiempo de ejecución de la CPU está compuesto por el tiempo requerido para realizar las instrucciones del programa y el tiempo que el sistema operativo necesita para ejecutar el programa**. Dada una arquitectura específica, una modificación de esta solo repercute en el tiempo requerido por el CPU (Tiempo de la CPU) para realizar las instrucciones de un programa.

El tiempo de la CPU puede determinarse a partir de la multiplicación de la cantidad de ciclos de reloj requeridos por el período del reloj (inverso de la frecuencia de la CPU). Adicionalmente, la cantidad de ciclos de reloj requeridos es igual al producto del total de instrucciones en el programa y la cantidad promedio de ciclos por instrucción (**CPI**, por sus siglas en inglés). De lo anterior podemos notar que el tiempo de la CPU es igual al producto de la cantidad de instrucciones requeridas para ejecutar el programa, el CPI y el período de reloj.

$$TiempodelCPU = totaldeinstrucciones \times CPI \times períododereloj$$

Es importante tener en cuenta que debido a que las diferentes instrucciones requieren de una cantidad diferente de ciclos para su ejecución, dada las disímiles maneras de su implementación a nivel de hardware, el CPI por definición tiene un valor real. De esta manera se pueden obtener diferentes valores del CPI para distintos programas, ya que en estos se pueden tener diferentes tipos y número de instrucciones. Realizando un análisis de las unidades de medidas utilizadas para el cálculo del tiempo de la CPU se tiene que este se formula de la forma siguiente:

$$segundos = \frac{instrucciones}{programa} \times \frac{ciclosdereloj}{instrucciones} \times \frac{segundos}{ciclosdereloj}$$

La expresión anterior muestra que el rendimiento del computador interrelaciona tres aspectos: el número de instrucciones, el número de ciclos de reloj y el tiempo de duración del periodo de reloj

requerido para ejecutar un programa. Teniendo en cuenta dichos aspectos con el fin de mejorar el rendimiento del computador es necesario reducir la cantidad de ciclos por programas, los ciclos de reloj o los segundos por ciclos de reloj. Es fundamental tener en cuenta al realizar cualquier modificación en la arquitectura del computador, que esta puede influir de manera simultánea en más de uno de los parámetros de los cuales depende el rendimiento; de esta forma el cambio realizado puede propiciar una mejora en uno de los parámetros a expensas del deterioro en uno de los otros.

A partir de los parámetros relacionados en la definición previa, es posible referirse al rendimiento del computador tanto en términos del tiempo de ejecución, como del número ciclos de reloj o del número de instrucciones necesarias para la ejecución de un programa. Estos tres parámetros están estrechamente relacionados entre sí, por ejemplo: la relación entre los ciclos de reloj y las instrucciones puede expresarse en función del CPI o de manera recíproca en términos del número promedio de instrucciones por ciclo de reloj (**IPC**, por sus siglas en inglés). El IPC es muy útil en el caso de referirse al rendimiento de computadores donde se realizan múltiples instrucciones por cada ciclo de reloj. Por otra parte, la relación entre el número de ciclos y el tiempo absoluto se puede expresar mediante el periodo de reloj o su inverso (la frecuencia de reloj); mientras que, una manera conveniente de referirse a la relación entre el número de instrucciones y el tiempo absoluto se expresa en millones de instrucciones por segundo (**MIPS**, por sus siglas en inglés). Este último refleja la razón promedio a la cual son ejecutadas las instrucciones.

Otra métrica de rendimiento utilizada en el pasado se basaba en la cuantificación del número de operaciones realizadas en punto flotante (**MFLOPS**, por sus siglas en inglés). Esta métrica es similar a las MIPS, pero en este caso solo se enfoca en las operaciones de punto flotante. Debido a que los programas no ejecutan únicamente este tipo de operaciones, dicha métrica no ofrece de manera real el rendimiento del computador. Teniendo en cuenta la gran variedad de métricas disponibles para cuantificar el rendimiento (el número de ciclos y de instrucciones requeridas para ejecutar un programa, el número de ciclos por segundo, los valores promedio de ciclos por instrucciones y de instrucciones por segundo) es importante cerciorarse que el indicador elegido pueda reflejar de manera real el rendimiento y no proporcione resultados sesgados del mismo.

1.3.2. Evaluación del rendimiento mediante programas

De todos los términos vistos hasta el momento el más importante a la hora de expresar el rendimiento es el tiempo de ejecución. Una manera efectiva de cuantificar el rendimiento del computador en función de este indicador se logra mediante la ejecución de aplicaciones reales. En el caso de los usuarios que usualmente utilizan programas específicos, la evaluación del tiempo de ejecución de dichas aplicaciones puede constituir un método efectivo de comparar el rendimiento en diferentes computadores. No obstante, es mucho más útil contar con programas de referencia que permitan evaluar de forma más general el rendimiento de los diferentes elementos del computador; a estos programas se le conocen como *benchmark*. Estas herramientas de evaluación pueden estar constituidas por una única aplicación o por un conjunto de aplicaciones que permitan caracterizar apropiadamente la mayor cantidad de características representativas del rendimiento del computador.

De forma general se puede disponer de *benchmarks* con diferentes grados de complejidad, desde los más simples hasta los más elaborados; pudiendo ser definidos por los usuarios, los fabricantes o

por ambos. Es importante tener en cuenta que, si bien los *benchmarks* de menor complejidad pueden ser optimizados para facilitar la interpretación de los resultados en determinados escenarios, estos pueden reflejar de manera inapropiada el rendimiento. Es por ello que a finales de la década de los 80 se creó la ***Standard Performance Evaluation Corporation*** (**SPEC**) constituida por un conjunto de importantes actores en la industria de la computación con el objetivo de establecer una herramienta que permitiera estandarizar los procedimientos para la definición de los *benchmarks*, establecimiento de las métricas del rendimiento y de sus valores referenciales para la evaluación de computadores.

Este esfuerzo conjunto tuvo como objetivo estandarizar diferentes aplicaciones reales para su utilización como parte de una plataforma de evaluación versátil que evitase la obtención de rendimientos poco fiables como resultado de la optimización de la arquitectura para favorecer aplicaciones específicas. Si bien es posible hacer algunas modificaciones en la arquitectura para brindar una falsa sensación de un mejor rendimiento del computador, el número de aplicaciones disponibles en las distintas versiones del proyecto SPEC hace más difícil este proceso. La variedad de aplicaciones, junto con su programación en un lenguaje de alto nivel, posibilitan la obtención de indicadores de rendimiento fiables tanto del computador como del compilador.

Típicamente en este *benchmark* se tienen un conjunto de programas para la evaluación del rendimiento del computador que se agrupan en dos grupos:

- Uno destinado a las aplicaciones con operaciones enteras (**SPEC int**)
- Otro para las aplicaciones con operaciones de punto flotante (**SPEC fp**)

Por citar algunos ejemplos de la amplia diversidad de aplicaciones disponibles para operaciones enteras, tomando como referencia la versión **SPEC 95**,se tienen por ejemplo: 'go' (programa de inteligencia artificial basado en el juego 'go'), 'm88ksim' (simulador del procesador Motorola 88k), 'X.gcc' (basado en compilador de C en GNU), 'li' (intérptreteLisp), ijpeg (compresor-descompresor jpeg), 'perl' (programa basado en perl para manipular cadenas y números primos), 'vortex' (programa para gestión de bases de datos), 'tomcatv' (programa para la generación de mayas vectorizadas).

Dentro de las diversas áreas del conocimiento relacionadas con los *benchmarks* implementados en las sucesivas generaciones de SPEC se encuentran: física cuántica, astrofísica, física del plasma, gráficos en 3D, ecuaciones diferenciales parciales, fluidos, termodinámica y química cuántica, entre muchas otras más. Toda la familia de *benchmarks* disponibles en las diferentes versiones de SPEC son revisadas periódicamente de forma tal que se mantengan actualizadas con respecto a las aplicaciones, computadores y cantidad de memoria disponible modernos. SPEC CPU 2017 contiene la versión más reciente de los *benchmarks* orientados a las mediciones y comparación del rendimiento de los CPU, el sistema de memoria y el compilador. Esta versión contiene 43 *benchmarks* agrupados en cuatro categorías, dos de ellas para evaluar el rendimiento con operaciones enteras (**SPECspeed 2017 Integer** y **SPECrate 2017 Integer**) y las otras dos para operaciones con punto flotante (**SPECspeed 2017 Floating Point** y **SPECrate 2017 Floating Point**). Los *benchmarks* contenido en las categorías **SPECspeed** permiten cuantificar el rendimiento en función del tiempo de ejecución; mientras que, los contenidos en las categorías **SPECrate** se expresan en función del rendimiento (cantidad de tareas completadas por unidad de tiempo).

Es importante destacar el hecho de que el conjunto de *benchmarks* contenidos en SPEC permite cuantificar el rendimiento en su totalidad del computador, no de los elementos aislados del mismo. Por lo que, en los resultados que se obtienen está reflejado el efecto en conjunto del funcionamiento tanto de la CPU como del sistema de memoria, los dispositivos E/S, los periféricos, el compilador y el sistema operativo.

1.3.3. Tendencias en el consumo energético

La eficiencia energética constituye todo un reto de suma importancia y complejidad en el diseño de todos los tipos de computadores contemporáneos. Existen tres aspectos esenciales relacionados con el consumo energético, a tener en cuenta en el diseño de un computador (Hennessy & Patterson, 2011):

- El primero de ellos es el **consumo máximo de potencia que requiere el procesador para su funcionamiento**, aspecto que en los procesadores actuales puede ser regulado empleando métodos de ajustes dinámicos del voltaje.

- El segundo aspecto constituye una **métrica del desempeño** conocida como **potencia de diseño térmico** (**TDP**, por sus siglas en inglés), el cual refleja el consumo de potencia sostenido y permite determinar los requerimientos para la disipación de calor del procesador.

- El último factor a tener en cuenta no solo en el diseño sino posteriormente para su utilización, es la **energía y la eficiencia energética**. Este último factor crea una interrogante a la hora de elegir cuál métrica es más representativa para analizar el desempeño de un procesador, ¿energía o potencia?

De forma general, la energía constituye una mejor métrica que la potencia debido a que esta refleja el tiempo de ejecución requerido para una tarea específica, que es igual a la potencia promedio del tiempo de ejecución. Lo antes mencionado implica que un procesador energéticamente más eficiente realiza una tarea determinada en un menor tiempo, o de forma equivalente podrá realizar mayor cantidad de tareas en un tiempo fijo. Debido a la creciente importancia de la eficiencia energética en el diseño de los procesadores actuales, el criterio de evaluación primario es la cantidad de tareas por joule de energía o el desempeño por Watt, a diferencia del pasado donde se priorizaba el desempeño por milímetro cuadrado de silicio (Hennessy & Patterson, 2011).

2. Lenguaje ensamblador

El ensamblador es un lenguaje de programación de bajo nivel donde generalmente cada declaración corresponde a una instrucción en lenguaje de máquina, básicamente es la representación simbólica del lenguaje de máquina; es por ello que el ensamblador puede verse como un lenguaje muy cercano al *hardware* del computador. El ensamblador es un ejemplo de los programas conocidos como traductores, siendo esencialmente una representación simbólica del lenguaje de máquina. El lenguaje ensamblador es específico para la ISA implementada, lo que implica que este lenguaje sea exclusivo para cada procesador en particular o para cada familia de procesadores. Como resultado, **el ensamblador lejos de ser un lenguaje único es una familia de lenguajes de bajo nivel**. Esta especificidad imposibilita la ejecución de un programa en ensamblador en procesadores con diferentes ISA.

La gran popularidad del uso de este lenguaje respecto al lenguaje de máquina se debe a que es mucho más sencillo utilizar nombres y direcciones simbólicas en lugar de su correspondiente representación numérica (binaria o hexadecimal). Adicionalmente, el ensamblador tiene otras bondades que contribuyen en algunos casos a su elección respecto a otros lenguajes de alto nivel. Dentro de sus bondades también está la de posibilitar una mayor eficiencia, normalmente los programas realizados en ensamblador suelen ser más pequeños y rápidos que versiones equivalentes realizadas en lenguajes de alto nivel. En muchos casos es imperativa la programación en ensamblador ya que varios dispositivos no ofrecen la posibilidad de implementar los programas en lenguajes de alto nivel, como por ejemplo algunos dispositivos embebidos de amplia utilización hoy en día. Otra de

las bondades que ofrece el ensamblador es que usualmente no está disponible por varios lenguajes de alto nivel, ya que este posibilita el acceso a todas las características e instrucciones disponibles en el computador. Dicho de otra forma, **todo lo que pueda realizarse en lenguaje de máquina también puede realizarse en ensamblador**. A pesar de las bondades que ofrece dicho lenguaje, este requiere de un mayor nivel de dificultad que los lenguajes de alto nivel; adicionalmente resulta más complicado su depuración y mantenimiento.

2.1. Formato de las declaraciones en ensamblador

Los programas en ensamblador están constituidos por una secuencia de declaraciones por cada línea de código. Las declaraciones se escriben empleando mnemotécnicos que representan las instrucciones correspondientes. **Típicamente una declaración en ensamblador está compuesta por dos partes, un código que representa un identificador textual de una instrucción de máquina (codop) y un conjunto de operandos que constituyen los argumentos de la instrucción.** Como se había mencionado previamente, normalmente cada declaración corresponde a una instrucción de máquina. Debido a que el ensamblador será específico para cada ISA implementada en el procesador, se tendrán diferentes familias del lenguaje con sintaxis específicas; de esta forma para la familia x86 se tendrá un tipo de ensamblador diferente al empleado en los procesadores ARM. En la Figura 1 se muestra un formato genérico para una declaración en ensamblador.

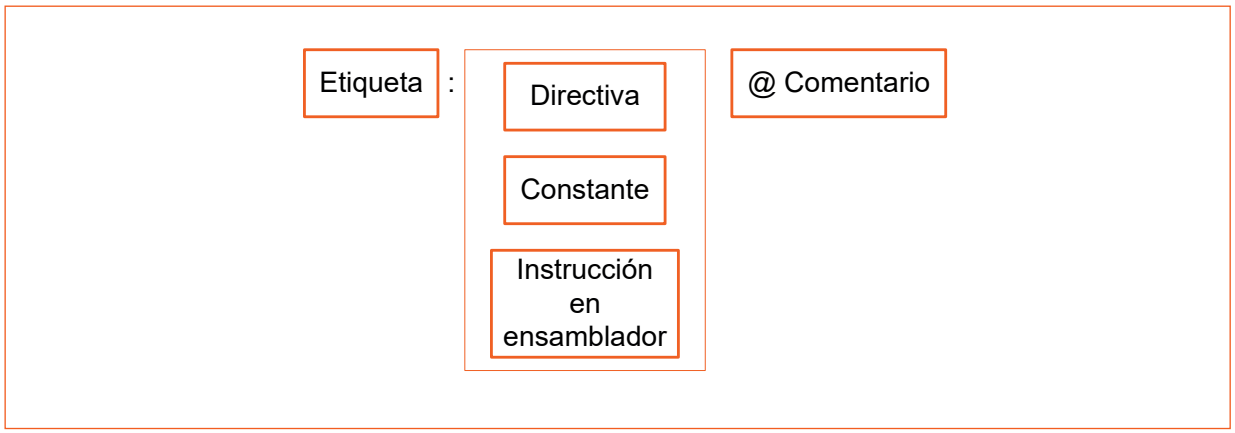

Figura 1. Estructura genérica de una declaración en ensamblador.

Las declaraciones se dividen en cuatro campos:

- Etiqueta
- Codop
- Operando
- Comentarios

Al inicio de las declaraciones se puede incluir opcionalmente una etiqueta que permite identificar la línea del código donde está ubicada la declaración. Las etiquetas pueden ser empleadas en caso de ser necesario realizar un salto hasta la instrucción donde esta se encuentra.

Algunas variantes de ensamblador utilizan dos puntos para identificar las etiquetas, mientras que otras emplean otros requerimientos. La utilización de dos puntos permite una mejor compilación del código. En algunos ensambladores se limita la longitud de los caracteres que pueden utilizarse en las etiquetas, dificultando un poco la posibilidad hacer más entendibles los programas.

En el campo dedicado al codop se realiza propiamente la declaración de la operación; esta puede referirse tanto a una instrucción en ensamblador, a la definición de una constante o a la declaración de una directiva. Es habitual emplear abreviaciones simbólicas en este campo de forma que se facilite el entendimiento de la operación que realiza la instrucción. Las directivas, conocidas también como pseudo-operaciones, se emplean para indicar al ensamblador la ejecución de una operación específica, las mismas pueden utilizarse para declarar una subrutina o el inicio de una nueva sección, estas son comandos que solo son interpretadas por el ensamblador, por lo que no corresponden a instrucciones de máquina. Existen una amplia variedad de directivas que pueden variar de un procesador a otro que no dependerán propiamente de las arquitecturas sino de la elección de los creadores del lenguaje. Dentro de las utilidades que permiten se tienen: definición de constantes, designación espacio en memoria para el almacenamiento de datos, inicialización de espacios de memoria, ubicación de tablas y otros tipos de datos en memoria y referenciado a otros programas (Stallings, 2013).

Los operandos pueden ser de diferentes tipos; por ejemplo, pueden ser valores constantes, registros o direcciones de memoria. A los operandos constantes se les suele llamar inmediatos y pueden referirse tanto a valores numéricos, caracteres o cadenas de caracteres. **En las declaraciones que contienen múltiples operandos, normalmente el primero de ellos representa el destino donde se almacena el resultado de la operación; mientras que, el resto indican los argumentos fuentes**. **A continuación de los operandos opcionalmente se puede incluir un comentario**; pudiéndose además utilizar una línea completa para ello. Los comentarios permiten incluir información adicional para facilitar la comprensión de los programas para posteriores revisiones.

De forma general se pueden tener cuatro tipos de instrucciones:

- De procesamiento de datos (aritméticas o lógicas)
- De transferencia de datos
- De bifurcación
- Especiales

Dentro del grupo de instrucciones para el procesamiento de datos se incluyen aquellas que permiten la comparación numérica y lógica. Las instrucciones de transferencia de datos posibilitan el intercambio de información entre los registros y las direcciones de memoria. Las instrucciones de bifurcación se emplean para implementar los bucles y saltos. Por otra parte, las instrucciones especiales permiten interactuar con los dispositivos periféricos, manejar parámetros específicos y acceder a partes específicas del computador, entre otras funciones.

2.2. Macros

La definición de una macro es similar a una subrutina, donde una sección de un programa puede escribirse y reutilizarse posteriormente en cualquier parte de dicho programa. La principal diferencia entre las macros y las subrutinas es que cuando el ensamblador encuentra una llamada a una macro, la reemplaza por la definición correspondiente de esta. Las macros permiten hacer referencia a un conjunto de instrucciones que posibilitan abordar de manera eficiente la necesidad de utilizar repetidamente una secuencia de instrucciones en un programa. Para la definición de las macros se utilizan diferentes notaciones, teniendo en común un identificador el inicio de la macro, la sección del código que se necesita utilizar repetidamente y un identificador para indicar el fin de la definición de la macro. Una vez definida la macro, cada vez que se requiera utilizar basta con invocarla mediante el nombre asignado. Este proceso se denomina **llamada a una macro**. Cuando se realiza una llamada a una macro, en la sección del programa donde se hace la llamada, se realizan las instrucciones incluidas en el cuerpo de la macro; a este paso se le conoce como **expansión**. Cuando la macro es llamada durante el proceso de ensamblado, en el programa en lenguaje de máquina correspondiente se introducen todas las instrucciones definidas dentro de dicha macro. Es por ello que el programa en lenguaje de máquina resultante cuando se utilizan macros es idéntico al que se tiene cuando no se utilizan.

Las macros también permiten reutilizar secuencias de códigos que no necesariamente deben ser idénticas, para ello se emplean parámetros. En esta aplicación de las macros los valores de los parámetros formales especificados en la definición de las macros pueden reemplazarse por los valores reales correspondientes cuando se invocan dichas macros. Como resultado, cuando se realiza la expansión de la macro, los parámetros formales contenidos dentro del cuerpo de esta son reemplazados por los parámetros reales correspondientes. En este caso, al igual que cuando se emplean macros para repetir secuencias de código idénticas, el programa en lenguaje de máquina correspondiente será idéntico al que se tiene cuando no se utilizan las macros.

Un aspecto a tener en cuenta cuando se emplean macros es la posibilidad de que ocurran duplicaciones en las etiquetas contenidas en su cuerpo en caso de que se realicen múltiples llamadas a dicha macro. Este inconveniente puede evitarse mediante diferentes métodos, algunos de ellos se basan en la asignación de diferentes etiquetas para cada llamada a la macro mientras que otros utilizan una declaración local para cada llamada. Otras características especiales de las macros que permiten ampliar su utilidad son la posibilidad de definir macros dentro del cuerpo de otras macros y de que una macro pueda llamar a otra macro o llamarse a ella misma.

2.3. El ensamblado

En el proceso de ensamblado las instrucciones de un programa en lenguaje ensamblador son traducidas al lenguaje de máquina binario para luego poder ser ejecutadas. En este proceso interviene el ensamblador, que es un *software* que lee el programa fuente escrito en lenguaje ensamblador para generar el programa objeto en lenguaje de máquina en su forma binaria. **En el ensamblado las direcciones simbólicas son reemplazadas por sus direcciones numéricas equivalentes**, los **codops** simbólicos por los códigos de las instrucciones de máquina correspondientes, **reserva espacio en memoria para el almacenamiento de las instrucciones y los**

datos; y, traduce las constantes a una representación entendible por el computador. Debido a la diversidad de lenguajes ensamblador existentes dependiendo de las diferentes **ISA**, para cada una de ellas se dispondrán de diversos programas de ensamblado según sea el caso; no obstante, todos estos funcionan de manera similar.

La manera más intuitiva de realizar el proceso de ensamblado podría ser la traducción secuencial de cada una de las instrucciones en lenguaje ensamblador al correspondiente código en lenguaje de máquina. Desafortunadamente no es posible realizar esta traducción directa debido al inconveniente que se presenta cuando se tienen declaraciones de bifurcación hacia partes del código que aún no han sido ensambladas. Esta limitante se conoce como **problema de referencia hacia adelante**, el cual impide que se traduzca una instrucción que emplea una referencia a otra que no ha sido traducida previamente.

El **problema de referencia hacia adelante** puede solventarse mediante la realización de dos lecturas del programa en ensamblador. Un primer enfoque basado en esta doble lectura emplea la primera para identificar los símbolos utilizados en la definición de las etiquetas para posteriormente almacenarlos en una tabla que contiene la lista de todas estas etiquetas y los valores del contador de dirección (**LC**, por sus siglas en inglés) asociados. Una vez identificados todos los símbolos es posible realizar una segunda lectura para efectuar la traducción de cada declaración sin que ocurra el **problema de referencia hacia adelante**. Todos los traductores que emplean esta filosofía se conocen como **ensambladores de dos pasos**, y constituyen **el tipo más común de ensamblador**. Luego de la conversión inicial se realiza el segundo paso de la traducción, donde se genera el código en lenguaje de máquina y adicionalmente se pueden ofrecer otras informaciones útiles para el proceso de enlazado. Para ello se vuelve a leer el programa desde el inicio realizando la traducción de cada declaración en su correspondiente código binario en lenguaje de máquina.

Un procedimiento alternativo de solventar el **problema de referencia hacia adelante** se basa en una lectura inicial del código en ensamblador para su traducción a una forma intermedia que es almacenada en una tabla de la memoria conocida como **tabla de símbolos**. La tabla de símbolos posee una entrada por cada símbolo existente, estos pueden estar definidos mediante etiquetas o de forma explícita. La organización de esta tabla puede realizarse de diferentes maneras, implementándose en la mayor parte de los casos en forma de memoria asociativa. El método más simple se realiza con un arreglo bidimensional. En esta variante se tienen dos vectores con dimensiones iguales a la cantidad de símbolos existentes, el primero de ellos hace referencia a los símbolos mientras que el otro al valor de estos. La principal ventaja de este método es su sencillez a la hora de programar, lo que se ve afectado por la lentitud de su ejecución debido a las búsquedas que requiere. Una solución a esta limitación se logra mediante la organización de la tabla de símbolos junto con la implementación del algoritmo de búsqueda binaria. Otro enfoque para implementar la tabla de símbolos es mediante la utilización de la técnica conocida como **código *hash***.

En la mayoría de los ensambladores, adicionalmente a la tabla de símbolos, se emplea como mínimo otras dos tablas durante el primer paso: de pseudo-instrucciones y de **codop** (Tanenbaum & Austin, 2013). La tabla de **codop** contiene como mínimo una entrada por cada instrucción que se tenga. Estas entradas pueden contener, entre otras informaciones, el **codop** propiamente, los operandos, el tipo y tamaño de la instrucción.

2.4. Enlazado y carga

Las aplicaciones generalmente están formadas por un conjunto de módulos objeto que han sido compilados o ensamblados. El enlazado de los diferentes procedimientos que generalmente conforman las aplicaciones permite la obtención del código ejecutable que permite correr el programa. Para este propósito, **el enlazador relaciona entre sí todos los procedimientos de forma que el programa se pueda ejecutar como una única unidad ejecutable binaria**. La ejecución de los programas requiere que el enlazador cargue todos los módulos objeto en la memoria virtual; en caso de que no se disponga de memoria virtual o que su capacidad no sea suficiente, los objetos pueden cargarse en la memoria principal del computador. El enlazador es la entidad que permite combinar todos los módulos objeto resultantes del ensamblado de múltiples módulos por separado en un único fichero que contiene un código ejecutable.

Otra de las funciones que permite el enlazador es la de solucionar los problemas de reubicación y referencia externa que ocurre cuando los módulos objeto son representados en espacios de memoria separados. Como resultado, el enlazador combina los módulos objeto asignándoles direcciones de memorias contiguas; adicionalmente, cada instrucción que hace referencia a direcciones de memoria se le adiciona un valor constante igual a la dirección de inicio de su módulo correspondiente.

Los módulos objeto normalmente constan de seis campos: identificación, tabla de puntos de entrada, tabla de referencia externa, instrucciones de máquina y constantes, y diccionario de reubicación del módulo final (Tanenbaum & Austin, 2013). El primer campo posibilita la identificación del módulo además de ofrecer la información adicional requerida por el enlazador. La tabla de puntos de entrada contiene una lista de los símbolos definidos en el módulo que pueden ser referenciados por otros módulos. Por otra parte, el campo dedicado a la tabla de referencia externa contiene la lista de símbolos que se han definido en otros módulos y que se usan dentro del módulo correspondiente. El cuarto campo se utiliza para almacenar el código ensamblado y las constantes definidas. El diccionario de reubicación contiene las constantes que son adicionadas a las instrucciones que utilizan direccionamiento de memoria. El último campo permite indicar el final del módulo y la dirección a partir de la cual empezar la ejecución; adicionalmente, puede utilizarse para incluir mecanismos de detección de errores.

El primer paso que debe realizarse para ejecutar un programa es cargarlo en la memoria principal; en el caso de que el programa posea un único módulo no es necesario realizar el enlazado y se carga directamente en memoria. En general hay tres tipos de enfoques en los que las aplicaciones pueden ser cargadas en la memoria: absoluto, reubicable y basado en tiempo de ejecución dinámico (Abd-El-Barr & El-Rewini, 2005). En la carga absoluta se requiere que los módulos sean cargados siempre en las mismas direcciones de memoria, conllevando ello la dificultad de que los programadores requieran conocer *a priori* la estrategia de asignación de memoria. Otra desventaja de esta técnica radica en que cuando se realizan modificaciones al programa que impliquen eliminar o adicionar líneas de código como resultado se alteran todas las direcciones posteriores a las líneas modificadas, esta última limitante puede solventarse mediante el uso de referencias a memoria de manera simbólica.

Un problema propio de los ambientes de multiprogramación, donde los programas pueden cargarse en la memoria por un periodo de tiempo para luego ser almacenados en disco hasta que vuelvan a cargarse nuevamente, es la dificultad de representar y ubicarlos en las mismas direcciones de memoria.

Dicho problema está relacionado con el tiempo en el que se realiza la última vinculación entre los nombres simbólicos y sus correspondientes direcciones de memoria, conocido como *binding time*. Las técnicas basadas en el mapeo de memorias virtuales en direcciones físicas permiten solventar el inconveniente de reubicación de los programas; ejemplos de este tipo de enfoques son el paginado y el enlazado dinámico. Otro tipo de mecanismo, aunque menos general que los anteriores, se basa en la utilización de registros de relocalización en tiempo de ejecución; alternativamente, se tienen las técnicas que hacen referencia a la memoria relativa al contador de programa, conocidas como **independientes de la posición**.

2.5. Lenguaje ensamblador en la arquitectura ARM

El lenguaje ensamblador ARM es uno de los más populares y de más amplia utilidad actualmente; este se utiliza en los procesadores que se encuentran en la mayoría de los dispositivos de comunicaciones móviles como teléfonos inteligentes y tabletas. El repertorio de instrucciones de los procesadores ARM es muy versátil. Este contiene instrucciones enteras, de puntos flotantes y vectoriales que permiten realizar múltiples adiciones en el mismo ciclo de reloj. La arquitectura ARM permite la posibilidad de utilizar dos tipos de repertorios de instrucciones, uno estándar donde todas las instrucciones ocupan 32 bits y otro ligeramente reducido conocido como *Thumb* en el que la mayoría de las instrucciones ocupan 16 bits. Los procesadores ARM poseen una arquitectura de la memoria de tipo Von Neumann y cuentan con 16 registros (r0, r1,…, r15), siendo algunos de ellos reservado para uso específico como por ejemplo el r15 que es el contador de programa.

2.5.1. Instrucciones básicas

La instrucción más sencilla de la ISA ARM es la mov; esta permite transferir el valor de un registro a otro y adicionalmente puede ser utilizada para transferir un valor inmediato hacia un registro. De esta manera, siempre al primer operando le corresponderá un registro mientras que el segundo puede ser un registro o un valor inmediato. Para indicar que se está haciendo referencia a un valor inmediato se emplea el carácter **#**. Cuando se tiene la siguiente declaración en **ARM**:

mov *r1,#7*

Se le está asignando el valor 7 al registro r1. Existe además en **ARM** una variante de la instrucción **mov** conocida como **mvn**. Esta es similar a la instrucción **not** empleada en muchas arquitecturas **RISC** pudiéndose utilizar en el segundo operando tanto un registro como un valor inmediato al igual que en **mov**. Un ejemplo de esta instrucción se tiene en la siguiente declaración:

mvn *r1,r2*

mvn esencialmente realiza lo mismo que la instrucción **mov** con la diferencia de que en lugar de transferirse el contenido de *r2* a *r1*, cada bit individual de *r1* es reemplazado por el complemento a uno de los bits contenidos en *r2*.

Por otra parte, se tienen las instrucciones aritméticas, como por ejemplo las destinadas a realizar la adición (**add**) y substracción (**sub**). En estas instrucciones se tiene un primer operando que indica el registro destino donde se almacenará el resultado de las operaciones. Los otros dos operandos

contienen los valores de los datos que quieren adicionarse o sustraerse según sea el caso, existiendo la posibilidad de declarar valores inmediatos en el tercer operando. En la siguiente declaración:

sub *r1,r2,r3*

Se sustrae el valor almacenado en *r3* (sustraendo) al valor contenido en *r2* (minuendo) y el resultado se almacena en *r1*. Adicionalmente a la instrucción **sub**, existe otra instrucción que realiza la substracción, pero de forma reversa (**rsb**); en esta, se invierte el orden del minuendo y sustraendo.

Otro ejemplo dentro del tipo de instrucciones aritméticas es la de multiplicación (**mul**), que es muy similar a las **add** y **sub**. A pesar de que la arquitectura **ARM** constituye un ejemplo de un repertorio **RISC**, este dispone de algunas instrucciones algo más elaboradas; algunas de ellas de gran utilidad para el manejo de operaciones propias de algebra lineal. Un ejemplo de este tipo de instrucciones es la **mla** que permite realizar la multiplicación y acarreo. En esta instrucción se utilizan cuatro operandos, el primero de ellos indica el destino, el segundo y tercero se emplean para los operandos de la multiplicación y el último para no de los sumandos. La siguiente declaración:

mla *r1,r2,r3,r4*

Realiza la multiplicación de los valores en los operandos *r2*, *r3* y el resultado se suma con *r4*. Otros tipos de instrucciones específicas para la multiplicación que se tienen en la arquitectura **ARM** son **smull** y **umull**. La primera de estas permite realizar la multiplicación con signo del contenido de dos registros (de 32 bits) y el resultado se almacena en otros dos registros (los 32 bits más significativos en uno de ellos y los 32 bits menos significativos en el otro). La instrucción **umull** realiza la misma operación que **smull** con la diferencia de que en este caso la multiplicación se realiza sin signos.

Las instrucciones lógicas son bastante similares a las aritméticas, dentro de estas se tienen la: **and**, **eor** (OR exclusivo), **orr** (OR) y **bic** (instrucción de borrado de bits). Las tres primeras instrucciones realizan las operaciones lógicas correspondientes, en cambio la última realiza una operación AND entre los bits del segundo operando y el complemento de los bits del tercer operando. **En todas estas instrucciones el primer operando siempre indicará el destino mientras que los otros dos son la fuente, pudiendo ser el último de estos un registro o un valor inmediato.**

2.5.2. Instrucciones avanzadas

En la arquitectura ARM se consideran diferentes tipos de instrucciones avanzadas y algunas de ellas se utilizan para realizar desplazamientos en los registros. En la mayoría de las variantes de ensamblador disponibles para ARM no se cuenta con instrucciones dedicadas únicamente a los desplazamientos, en lugar de esto el desplazamiento se realiza en la misma definición del operando fuente. Existen cuatro formas en las cuales se pueden llevar a cabo estos desplazamientos:

- Desplazamiento lógico a la izquierda (**lsl**)
- Desplazamiento lógico a la derecha (**lsr**)
- Desplazamiento aritmético a la derecha (**asr**)
- Rotación a la derecha (**ror**)

Todas estas instrucciones pueden operar en el valor contenido en un registro o en un valor inmediato. En **lsl** los bits se desplazan a la izquierda y se adiciona un '0' al bit menos significativo (**LSB**, por sus siglas en inglés). En el caso de **lsr** el desplazamiento se realiza hacia la derecha y el '0' se adiciona al bit más significativo (**MSB** por sus siglas en inglés). En el caso de la instrucción **asr** los contenidos del registro son tratados como dos números enteros con signos complementarios y el bit dedicado al signo se copia en la posición vacante. La instrucción **ror** realiza la rotación un bit a la derecha de todo el contenido del registro, de manera que el bit que sale por el extremo derecho del registro producto a la rotación se coloca en el extremo izquierdo.

Una de las instrucciones que se emplean para realizar comparaciones en **ARM** es **cmp**, en la que se substrae del primer operando (registro) el valor del segundo operando (registro o un valor inmediato). En dependencia del resultado que se tenga, dicha instrucción podrá modificar los bits bandera cero (**Z**), bandera negativa (**N**), acarreo (**C**) y desbordamiento (**F**) del registro del estado actual del programa (**cpsr**). Una manera de utilizar la instrucción **cmp** es:

cmp *r1,r2*

En esta declaración se comparan los valores contenidos en los registros *r1* y *r2*. En caso de que *r1*= *r2* se activa la bandera cero del **cpsr**, mientras que en el caso de que *r1*<*r2* se activará la bandera negativa. La instrucción **cmn** es similar a la **cmp**, solo que en lugar de restar los operandos los suma. Otras de las instrucciones de comparación disponibles en **ARM** que no modifican los bits C y F del **cpsr** son: **tst** y **teq** que se basan en la realización de operaciones AND y OR exclusivo.

Es importante mencionar que las instrucciones de comparación no son las únicas que permiten modificar los valores del **cpsr**. Adicionalmente es posible que las instrucciones **add** y **sub** modifiquen las banderas del **cpsr** para indicar si el resultado de la operación que realizan es igual a cero, es negativo o se ha desbordado, para ello solo basta adicionar una 's' al final de dichas instrucciones. Otras instrucciones como la de adición con acarreo (**adc**), substracción con acarreo (**sbc**) y substracción reversa con acarreo (**rsc**) pueden utilizar el valor almacenado previamente en el bit de acarreo del **cpsr** siendo muy útiles para realizar operaciones de 64 bits empleando registros de 32 bits.

2.5.3. Instrucciones de bifurcación

Las instrucciones de bifurcación son de especial relevancia a la hora de implementar bucles o tomar decisiones en dependencia del valor de determinadas variables. En la arquitectura **ARM** se dispone de algunas instrucciones de bifurcación muy simples como un salto incondicional a una etiqueta que se representa mediante la letra **b** seguida de la etiqueta a donde desea realizarse el salto. Los saltos condicionales en **ARM** poseen un formato estándar, seguidamente de la letra **b** estos emplean un sufijo de dos letras que identifican la condición del salto. Un ejemplo de algunos de los saltos condicionales disponibles en **ARM** son el **beq** y **bne**. El **beq** realiza un salto a la etiqueta especificada si la última instrucción que manipuló la bandera **Z** del **cpsr** arrojó como resultado una igualdad; por otra parte, en **bne** el salto se realiza si el resultado se corresponde a una desigualdad.

En la Tabla 1 se muestran los diferentes tipos de saltos condicionales disponibles en **ARM** y los valores que toman las banderas del **cpsr** en cada caso.

Tabla 1
Diferentes tipos de saltos condicionales disponibles en la arquitectura ARM

Condición	Sufijo	Significado	Estado de las banderas
0	eq	Igual	Z=1
1	ne	Desigual	Z=0
2	cs/hs	Configuración del acumulador/mayor o igual (sin signo)	C=1
3	cc/lo	Limpiado del acumulador/menor que (sin signo)	C=0
4	mi	Negativo/menos	N=1
5	pl	No negativo/mas	N=0
6	vs	Con desbordamiento	F=1
7	vc	Sin desbordamiento	F=0
8	hi	Mayor (sin signo)	(C=1) ∩ (Z=0)
9	ls	Menor o igual (sin signo)	(C=0) U (Z=1)
10	gr	Mayor o igual (con signo)	N=0
11	lt	Menor (con signo)	N=1
12	gt	Mayor (con signo)	(Z=0) ∩ (N=0)
13	le	Menor o igual (con signo)	(Z=1) U (N=1)
14	al	Siempre	
15	-	reservado	

Una instrucción un poco más elaborada es la de **bifurcación y enlazado** (**bl**) que permite invocar a una subrutina mediante la realización de dos operaciones, primero se efectúa un salto incondicional a la etiqueta que apunta a la primera instrucción de la subrutina y se almacena el siguiente valor del Contador del Programa (**PC**, por sus siglas en inglés) en el registro de dirección de retorno que en ARM se conoce como registro de enlace (**lr**). Un aspecto a tener en cuenta cuando se trabaja con subrutinas

en ARM es que no se dispone de retorno; para ello existen distintas variantes, una de ellas se basa en la utilización del **lr**.

Otra de las instrucciones que permite realizar un salto incondicional es la **bx**, esta es una versión de la instrucción **b**, pero con la diferencia de que en vez de realizar un salto a un registro se efectúa a la dirección contenida en un registro. La instrucción **bx** permite realizar el retorno de una subrutina en las variantes de los ensambladores **ARM** que la implementan.

Los sufijos empleados en los saltos condicionales pueden ser utilizados en conjunto con varias de las instrucciones básicas vistas con anterioridad, proporcionando una variante condicional de dichas instrucciones. Este tipo de instrucciones también son conocidas como predicadas y se ejecutan solo en el caso de que se cumpla la condición indicada. Para la definición de estas instrucciones basta con añadirle el sufijo a la instrucción deseada, un ejemplo de estas son **addeq** y **addmi**.

2.5.4. Instrucciones de memoria

En la arquitectura **ARM** existen, además, las **instrucciones básicas para la carga y almacenamiento de datos**, alguna de ellas son las **ldr** y **str**. La instrucción **ldr** permite leer el contenido de las direcciones de memoria indicada por el segundo operando y cargarlas en el primero. De manera similar **str** permite almacenar en una dirección de memoria indicada, por el segundo operando, el contenido del primer operando. Adicionalmente en **ARM** se tiene un elevado número de variantes en las que se pueden emplear las instrucciones para el manejo de memoria. Una de estas variantes emplea **ldr** con un desplazamiento respecto a la posición inicial (*offset*); en las siguientes declaraciones se muestran las formas de implementación de este enfoque.

ldr *r1*, [*r0*, *#7*]

ldr *r1* [*r0*, *r2*]

Esta variante de la instrucción **ldr** esencialmente carga en el registro *r1* el valor contenido en las direcciones de memoria resultante de la suma del valor base *r0* más el *offset* indicado por el valor inmediato *#7* o por *r2*. Es importante destacar que esta variante no constituye en sí un modo basado en índice, más bien es un modo basado en un *offset*. De manera similar se puede utilizar **ldr** en una variante indexada, para ello también se emplea *r0* para almacenar la dirección base y adicionalmente es necesario disponer un índice. Este modo tiene mucha utilidad para acceder a los arreglos, en caso de que los elementos que lo componen tengan un tamaño de 4 bytes será necesario emplear una versión escalada del índice conformada por los múltiplos de 4 incluyendo al 0. En la siguiente declaración se puede apreciar una forma de utilizar este modo:

ldr *r1* [*r0, r2,* **lsl** #2]

En esta declaración la dirección de memoria, al igual que en el modo anterior, se indica entre los corchetes; dicha dirección será el resultado de sumar el valor de *r0* con el resultado de multiplicar los valores en *r2* (0, 1, 2, 3, 4, 5…) por 4, para ello se utiliza el desplazamiento a la izquierda (**lsl** #2).

Una manera avanzada de instrucciones para el manejo de memoria se consigue mediante el modo de direccionamiento con postindexado. Esta variante permite acceder al arreglo y actualizar el

índice en una misma instrucción en la que se emplea un formato diferente en la definición de **ldr**. El funcionamiento de este modo se ilustra mediante la siguiente declaración,

ldr *r0*, [*r1*], *r2*, **lsl**#2

Esta instrucción asigna a *r0* el valor de la dirección de memoria contenida en *r1* y posteriormente incrementa el valor de *r1* en cuatro unidades. De forma similar se dispone de un modo de direccionamiento con pre indexado en el cual la actualización del índice se realiza antes. La dirección de memoria puede indicarse mediante cualquier formato, bien sea de manera indirecta o mediante los modos basados en *offset*, índice o índice escalado. En la siguiente declaración se puede apreciar un ejemplo de este modo de direccionamiento,

ldr *r0*, [*r1*, #4]!

En este caso se accede a la dirección resultante de sumar el contenido de *r1* con 4 y posteriormente se actualiza *r1* incrementándole 4 unidades al valor actual.

Las instrucciones para la manipulación de la memoria también suelen ser muy útiles cuando se realiza una llamada a una subrutina. En estos casos, son empleadas para almacenar el valor de un conjunto de registros antes de la llamada y para restituirlos una vez se retorne. Existen dos esquemas mediante los cuales se puede salvar el contexto de los registros durante la llamada a una subrutina, en una de estas los registros pueden ser directamente utilizados por la función (sin ser preservados), mientras que en el otro los valores de los registros sobrevivirán a la llamada de función.

En **ARM** el almacenamiento y restitución de los registros, como parte de una llamada a una subrutina, se puede realizar de una manera sencilla mediante las instrucciones **stmfd** y **ldmfd**. La primera de estas esencialmente salva en la pila todo el conjunto de registros del contexto actual, mientras que la segunda permite la restitución de dichos registros. El formato de ambas instrucciones es el mismo, primero se especifica el puntero de la pila de manera similar al modo de direccionamiento pre indexado donde se emplea el signo de exclamación y luego se indica los registros que se desean salvar o recuperar según sea el caso.

2.5.5. Codificación de las instrucciones

La mayoría de las instrucciones en ARM poseen un formato genérico constituido por 32 bits (Figura 2).Consideremos un formato genérico para una instrucción de procesamiento de datos o instrucciones efectuadas por la **ALU**.

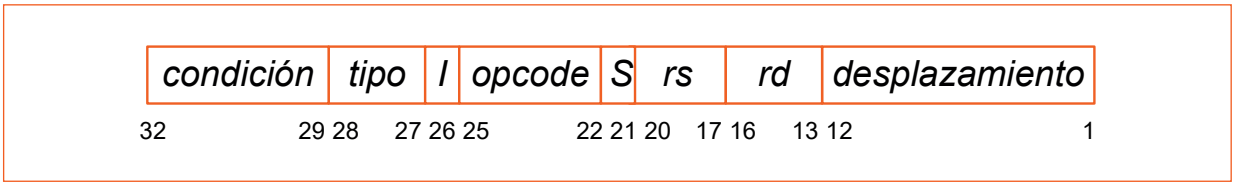

Figura 2. Sistema de memoria caché constituido por tres niveles.

En este formato los primeros 4 bits (29-32) se emplean para especificar la condición de la instrucción deseada dentro de todas las disponibles mostradas en la Tabla 1. Los siguientes dos bits (27-28) se

utilizan para representar el tipo de instrucción, con lo que es posible soportar cuatro tipos diferentes. En este caso como se tiene una instrucción de procesamiento de datos se representa con el valor 00. El bit inmediato (26) se usa para indicar si se utiliza un valor inmediato o un registro para el desplazamiento en el segundo operando fuente. Los bits correspondientes al campo **opcode** (22-25) se destinan para indicar el tipo de operación que se realiza. El bit de sufijo (21) se utiliza para permitir que cualquier instrucción de manejo de datos pueda manipular las banderas del **cpsr**. El campo **rs** de 4 bits (17-20) sirve para indicar el primer registro fuente, mientras que el campo **rd** (13-16) indica el registro destino. Los últimos bits del formato (1-12) se dedican para especificar un operando de desplazamiento de registro o un valor inmediato.

Para la codificación inmediata en **ARM** se disponen de 12 bits, estos bits se dividen en dos partes: 8 bits para la carga útil y los otros 4 para indicar el valor de la rotación, de esta manera se pueden codificar números de 32 bits mediante su correspondiente representación en estos 12 bits utilizando rotaciones. Cuando se emplean registros en el último operando de la instrucción también se tiene la posibilidad de poderlo desplazar.

Otro tipo de instrucciones disponibles en la arquitectura **ARM** son las de lectura y almacenamiento que poseen un formato igual al analizado previamente (Figura 3). En este caso los bits que definen el tipo de instrucción toman el valor 01 y posteriormente se tienen los bits I, P, U, B, W y L. Cuando el bit 'I' toma valor 0 indica que los últimos 12 bits representan un valor inmediato, en caso contrario representará un operando de desplazamiento. El bit 'P' indica el modo de direccionamiento, cuando toma valor 0 indica post indexado y con valor 1 pre indexado. El bit 'U' se utiliza para substraer o adicionar el *offset* al valor base. En el caso del bit 'B', se emplea para transferir una palabra de 4 bytes a la memoria o un solo byte. El bit 'W' se dedica para indicar si se utiliza un direccionamiento regular o los modos de pre y post indexado. Finalmente se tiene el bit 'L' que se destina para indicar si se realiza una operación de lectura o escritura en memoria. En una instrucción de lectura el campo **rd** indica el destino mientras que **rs** se utiliza para el indicar el registro base. En el caso de una instrucción de almacenamiento el campo **rs** tiene la misma utilidad mientras que **rd** indica el registro desde donde proviene el dato. Una distinción importante es que en este tipo de instrucciones el valor inmediato indicado corresponde a un número de 12 bits sin signo.

Figura 3. Sistema de memoria caché constituido por tres niveles.

Por otra parte, se tienen las instrucciones de bifurcación en las cuales se emplea el formato similar de 32 bits como se puede apreciar en la Figura 4. Al igual que en las instrucciones anteriores, en este caso se utilizan los cuatro primeros bits para indicar la condición de la instrucción y luego se tendrán dos bits para indicar el tipo de instrucción. Adicionalmente se incluye un tercer bit para indicar el subtipo de instrucción. Subsecuentemente se tiene el bit de enlace 'L' para indicar que se configura el registro

para la dirección de retorno. Finalmente se dispone de 24 bits para indicar el *offset* de la bifurcación, este se expresa en términos de palabras de memoria (4 bytes en este caso).

Figura 4. Sistema de memoria caché constituido por tres niveles.

3. Jerarquía de memoria

Desde el punto de vista de los usuarios, es deseable disponer de cantidades ilimitadas de memoria para no estar sujeto a estas restricciones a la hora de elaborar cualquier programa. En este sentido, no es posible contar con recursos ilimitados de memoria y al mismo tiempo garantizar altas velocidades de acceso a todos estos recursos de manera simultánea. Esta limitación práctica encuentra sustento en el principio de localidad, el cual postula que un programa acede en un instante de tiempo dado a una cantidad relativamente pequeña del espacio de memoria, este principio implica dos tipos de localidad:

- **Localidad temporal:** si un recurso ha sido referenciado en un instante dado, existe una elevada tendencia de que sea referenciado nuevamente en un tiempo relativamente corto.

- **Localidad espacial:** si un recurso ha sido referenciado, existe una tendencia elevada de referenciar los recursos con direcciones cercanas en un tiempo relativamente corto.

Existe una relación estrecha y contrapuesta entre las principales características que deben tener las memorias; dígase capacidad, tiempo de acceso y costes. Por un lado, se tiene que las memorias más rápidas son mucho más costosas y por ende su tamaño es más reducido; mientras que, las más lentas son mucho más económicas lo que permite un mayor tamaño. Estas características plantean un reto importante a la hora de implementar una tecnología de memoria que proporcione una elevada capacidad con un tiempo de acceso pequeño a un precio razonable. La solución a este problema

se encuentra utilizando una jerarquía de memoria en lugar de emplear una única tecnología. Esta jerarquía se implementa teniendo en cuenta el planteamiento del principio de localidad, a partir de lo cual se establecen varios niveles con distintos tamaños y velocidades; los cuales se ubican en dependencia de la distancia al procesador como se puede apreciar en la Figura 5. De esta forma, **resulta conveniente ubicar las memorias más rápidas (o primarias) cercanas al procesador con el objetivo de garantizar un menor tiempo de acceso**; mientras que, las memorias más lentas (o secundarias) se ubican en el nivel más bajo de la jerarquía ofreciéndole al usuario elevados volúmenes de memoria con tecnologías más económicas. Como resultado se tiene que el desempeño del sistema en su conjunto se aproxime al de la memoria más rápida, con un coste cercano al de la memoria más lenta.

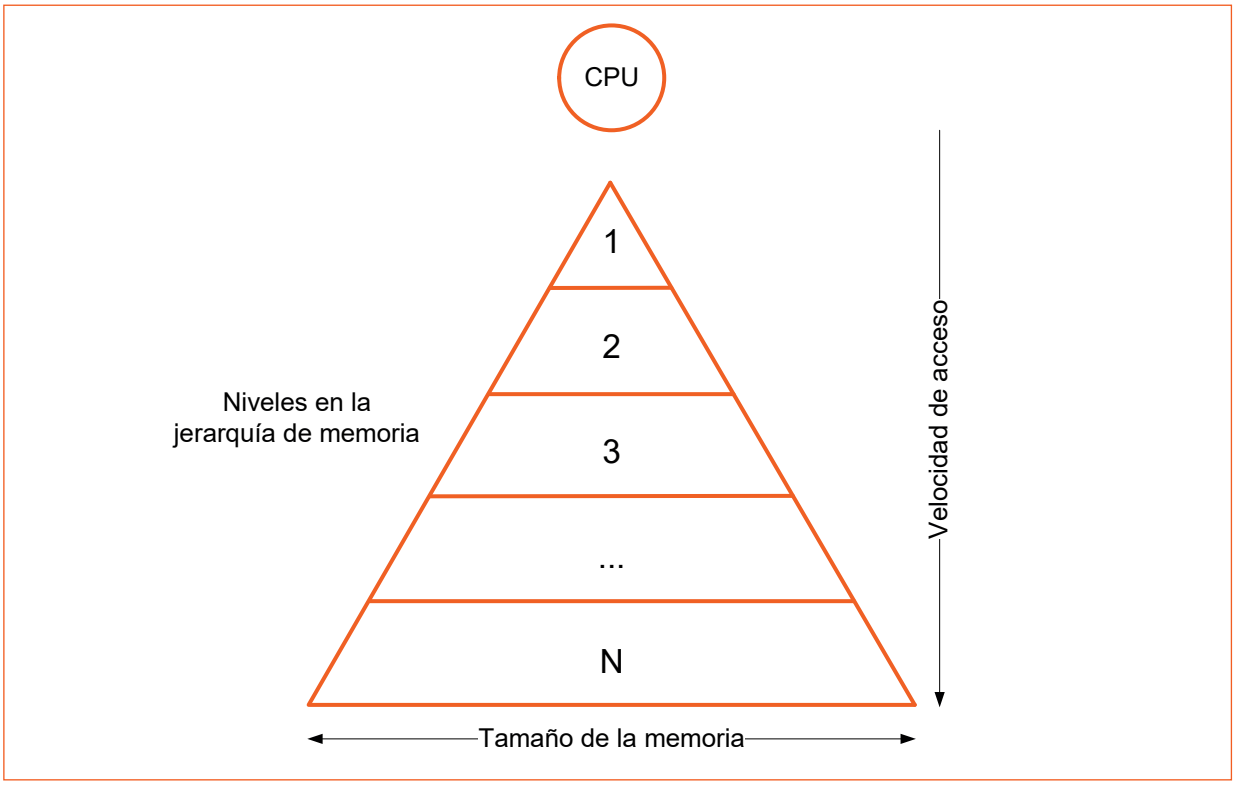

Figura5. Ejemplo de la jerarquía de memoria constituida por varios niveles con tamaños y velocidades de acceso diferentes.

Las memorias primarias se fabrican basadas en la técnica de acceso aleatorio, razón por la cual son conocidas como **memoria de acceso aleatorio** (**RAM**, por sus siglas en inglés), estas garantizan un menor tiempo de acceso debido a que este se realiza de forma independiente de la posición de la palabra. Por otra parte, las memorias secundarias se fabrican basadas en dispositivos de almacenamiento de acceso directo (**DASD**, por sus siglas en inglés) y opcionalmente basados en la técnica de acceso secuencial (***Sequential Access Memory***). La jerarquía de memoria normalmente se estructura de forma tal que las direcciones de memoria en los niveles superiores forman un subconjunto de los espacios en los niveles inferiores. Esta relación conlleva que haya redundancia en parte de la información en los diferentes niveles; sin ello implicar que las ubicaciones de las mismas direcciones de memoria necesariamente deban coincidir para distintos niveles. A pesar de la redundancia que existe en la información, puede darse el caso de que en un nivel la información esté más actualizada que en el otro; dando lugar a lo que se conoce como el **problema de consistencia o coherencia**.

Las memorias primarias se fabrican basadas en la técnica de acceso aleatorio, las cuales garantizan un menor tiempo de acceso debido a que este es independiente de la posición de la palabra. Por otra parte, las memorias secundarias se fabrican basadas en dispositivos de almacenamiento de acceso directo y opcionalmente basadas en la técnica de acceso secuencial. La jerarquía de memoria usualmente se estructura de forma tal que las direcciones de memoria en los niveles superiores forman un subconjunto de los espacios en los niveles inferiores. Esta relación conlleva a que haya redundancia en parte de la información en los diferentes niveles; aunque ello no implica que las ubicaciones de las mismas direcciones de memoria necesariamente deban coincidir para distintos niveles.

Cuando el procesador requiere utilizar un dato para una instrucción este lo busca en el nivel de memoria más cercano y, de encontrarlo, se considerará a este evento como un acierto. De no encontrarlo, entonces procederá a buscar el dato en el nivel inmediato inferior hasta encontrarlo, en este caso ocurre un fallo. **El principio de localidad permite organizar los datos de manera que el porcentaje de acierto sea mayor que el de fallos.** La tasa de aciertos es un parámetro importante a tener en cuenta para modelar el desempeño de la jerarquía de memoria; esta se define como la probabilidad de encontrar la información requerida en un determinado nivel. De una manera inversa se puede definir la tasa de fallos como la probabilidad de no encontrar la información en este nivel.

Otra métrica de desempeño importante es el **tiempo de acierto**, este se define como el tiempo que se requiere para acceder al nivel superior de la jerarquía que incluye el tiempo destinado para determinar la ocurrencia tanto de los aciertos como de los fallos. La suma del tiempo necesario para remplazar un bloque de memoria en el nivel superior y entregar dicho bloque al procesador se conoce como **penalización por fallo**. Dada la importancia del sistema de memoria en el funcionamiento del computador, se requiere de un diseño apropiado que emplee mecanismos muy sofisticados que ofrezcan el mejor desempeño del sistema.

Si bien el desempeño de la jerarquía de memoria puede cuantificarse a partir del tiempo de acceso efectivo a esta en cada referencia, cabe destacar que dicho desempeño depende de un conjunto interrelacionado de factores que influyen directamente en el coste de implementación. Algunos de los parámetros más comunes son: el comportamiento del programa respecto a las referencias de memorias, los tiempos de acceso y tamaños de la memoria en cada nivel, el tamaño del bloque de información, la red de interconexión con el procesador, entre otras (Hwang & Briggs, 1984). Es por ello que el diseño de la jerarquía conduce a un problema de optimización en el que se trata de minimizar el tiempo de acceso a la memoria sujeto a diferentes restricciones asociadas a los parámetros mencionados previamente.

3.1. Tipos de memoria en la jerarquía

De acuerdo con la ubicación en el computador, las memorias se pueden clasificar como internas o externas. Normalmente se tiende a asociar la memoria principal del procesador como el único tipo de memoria interna; si bien este tipo de memoria pertenece a esta categoría también existen otras formas de memorias internas como por ejemplo los registros, la memoria destinada a la unidad de control y la memoria caché. Las memorias más rápidas, de menor capacidad y más costosas se emplean para implementar los registros internos del procesador. Por otra parte, tenemos la memoria principal,

que constituye el sistema de memoria interna del computador más relevante, la cual se utiliza para almacenar los datos que se utilizan de manera inmediata en la ejecución de las instrucciones. La caché es una memoria de gran velocidad que permite extender la memoria principal, constituyendo una interface que facilita el intercambio de información entre los registros y la memoria principal; generalmente, los usuarios no tienen acceso a esta. Todos los tipos de memorias mencionadas con anterioridad tienen en común que son volátiles y se implementan con tecnologías semiconductoras.

Una de las soluciones que es posible implementar para mejorar el desempeño de la memoria principal en cuanto al tiempo de acceso a un bloque es la utilización de la técnica de memoria entrelazada. Esta solución se basa en la organización de los bancos de memoria de forma tal que puedan leerse o escribirse simultáneamente múltiples palabras en un mismo acceso. Para ello se divide la memoria principal en diferentes módulos que pueden trabajar en paralelo lográndose así un entrelazamiento de orden alto o de orden bajo.

Para el almacenamiento de grandes volúmenes de datos usualmente se realiza de forma permanente en dispositivos de almacenamiento externos que forman parte de la categoría de memorias externas. Algunos de los principales tipos de dispositivos de almacenamiento son el disco duro, dispositivos extraíbles como los discos magnéticos, los discos ópticos, las memorias *flash*, entre otros. Estos dispositivos generalmente se emplean para el almacenamiento de los programas y datos en forma de ficheros y registros; aunque, también pueden utilizarse como una extensión de la memoria principal conocida como memoria virtual. Estos tipos de dispositivos tienen en común su elevada capacidad, menores costes, y tiempos de acceso considerablemente mayores en comparación con las memorias internas.

3.2. Memoria caché

La principal limitación en cuanto al tiempo de procesamiento de los computadores se debe a la gran diferencia existente entre las velocidades del procesador y las velocidades de acceso al sistema de memoria. Una solución que permite resolver esta problemática de manera efectiva es la implementación de la memoria caché. Este tipo de memoria se diseña para combinar las características de las memorias más costosas utilizadas en los registros con las memorias más económicas empleadas en los sistemas de almacenamiento externos. De esta forma se logra un compromiso entre los bajos tiempos de acceso y una mayor capacidad de almacenamiento. **La memoria caché constituye una interface entre el CPU y la memoria principal; permitiendo almacenar una porción de la información de esta última.**

Cuando el procesador necesita cargar los datos de la memoria, primero revisa a ver si está almacenada en la caché y de encontrarse allí procede a leerla; en caso contrario, se carga un bloque de la memoria principal, constituido por varias palabras, en la caché para que luego el procesador lea la palabra que contiene los datos requeridos. Este proceso de cargar un bloque entero en la caché se sustenta en el principio de localidad visto con anterioridad, garantizando una reducción considerable del tiempo de acceso. Usualmente la memoria caché se divide en diferentes niveles, cada uno de ellos con capacidades y velocidades diferentes. En la Figura 6 se muestra un sistema de caché basado en tres niveles. Típicamente el primer nivel posee la mayor velocidad y el menor tamaño, decreciendo el primero de estos parámetros y aumentando el segundo de un nivel a otro.

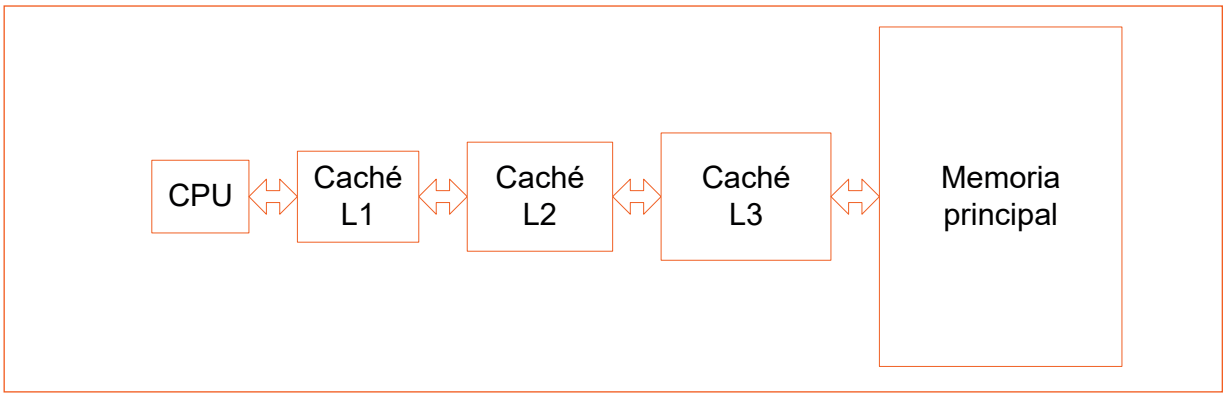

Figura 6. Sistema de memoria caché constituido por tres niveles.

La memoria principal está constituida por un conjunto de 2^n palabras, cada una de estas tiene una dirección única. Para facilitar el mapeo de la memoria se agrupan un conjunto fijo de *K* palabras constituyendo un bloque; de forma tal que se tenga un número de bloques igual a $M=2^n/K$. Por otra parte, la caché está constituida por un número *m* de bloques, conocidos como líneas para distinguirla con respecto a la memoria principal, siendo por lo general $m << M$. Cada línea está constituida por un conjunto de *K* palabras más algunos bits adicionales que cumplen funciones de identificación y control.

Para ilustrar el modo de funcionamiento del sistema de memoria consideremos uno constituido por dos niveles (la caché y la memoria principal); tengamos presente, además, que los datos que contiene el nivel superior son un subconjunto de los datos en el nivel inferior. En la Figura 7 se representa el proceso de lectura de una dirección de memoria para los sistemas de caché contemporáneos. Durante la operación de lectura el procesador genera la dirección de la palabra que se desea leer (**RA**, por sus siglas en inglés). De encontrarse esta palabra en la caché ocurre un acierto y se envía directamente al procesador; de lo contrario, ocurre un fallo y se busca en la memoria principal. Cuando ocurre un fallo, en los computadores actuales, simultáneamente se carga en una de las líneas de la caché el bloque completo que contiene la dirección de memoria solicitada y se envía al procesador la palabra que contiene el dato requerido.

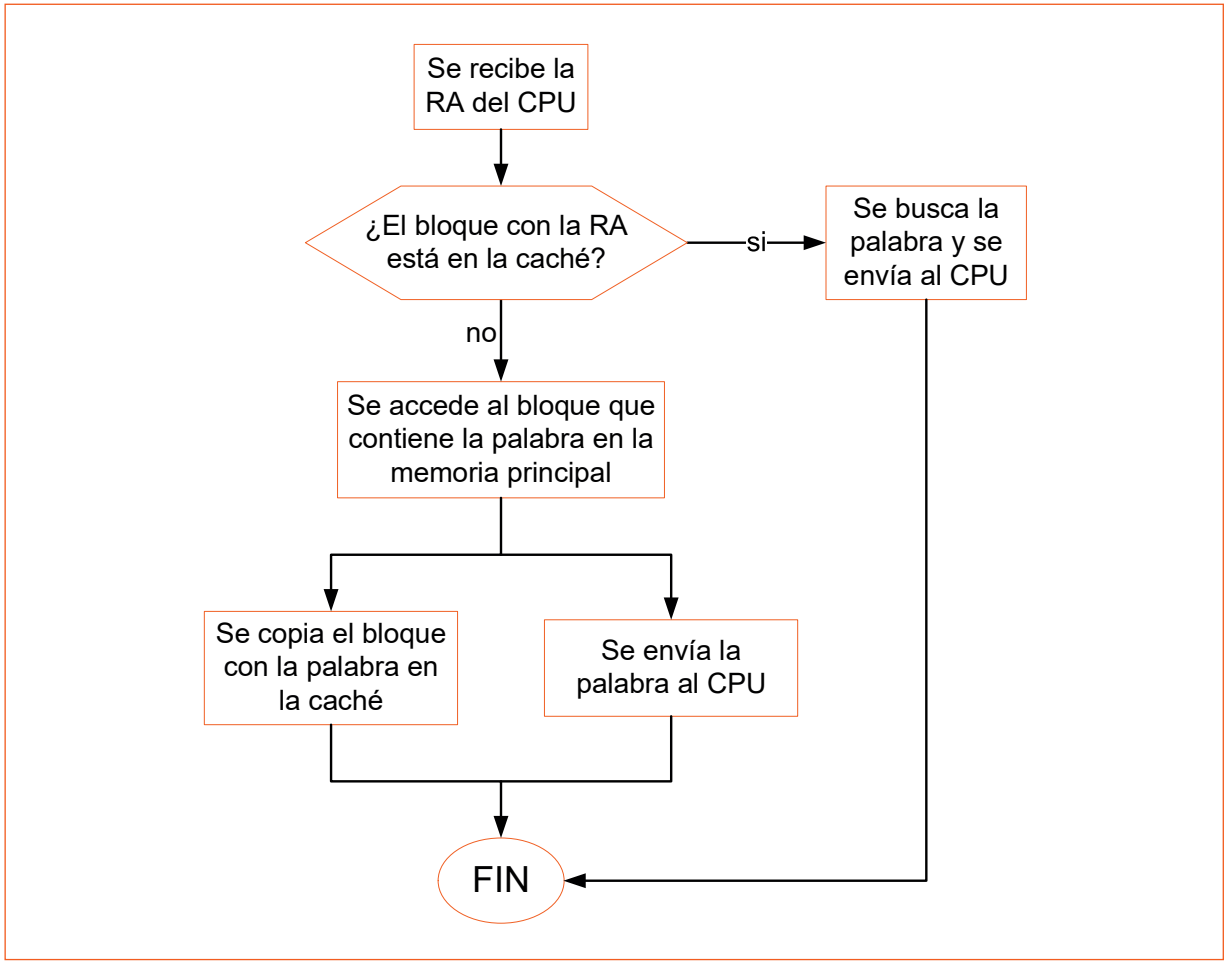

Figura 7. Proceso de lectura de una dirección de memoria para los sistemas de caché contemporáneos.

3.2.1. Tipos de direccionamiento

Cuando en el computador se implementa un módulo de memoria virtual, existirán dos tipos de direccionamientos para la caché en dependencia de su ubicación respecto a la **unidad de administración de memoria** (**MMU**, por sus siglas en inglés) (Hamacher *et al*., 2012). **La función de la MMU es la conversión de direcciones virtuales a direcciones físicas. Cuando la caché se ubica entre el CPU y el MMU se tiene la configuración conocida como caché lógica o virtual**; en este esquema el procesador accede directamente a la caché y el direccionamiento se realiza mediante el **MMU** utilizando direcciones virtuales. **En el caso de que la caché se ubique entre el MMU y la memoria principal se obtiene la configuración de caché física**, en la que el direccionamiento se realiza mediante direcciones físicas.

La principal ventaja de utilizar la caché lógica respecto a la física es que el acceso es más rápido debido a su conexión directa con el **CPU**. Por otra parte, la desventaja de la caché lógica radica en que la mayoría de los sistemas de memoria virtual les asignan a todas las aplicaciones el mismo espacio de memorias virtuales. Es por ello que en esta configuración se necesita vaciar la caché completamente cada vez que se pasa de una aplicación a otra; o de lo contrario, se requiere utilizar algunos bits para la identificación de los distintos espacios de memoria virtual.

3.2.2. Políticas de emplazamiento

Debido a que el tamaño de la memoria caché es mucho menor que el de la memoria principal, se requiere de un mecanismo que permita determinar en qué líneas de la caché se ubicarán los nuevos bloques de información. A este proceso se le conoce como **emplazamiento** y puede llevarse a cabo mediante tres tipos de políticas: directa, asociativa y asociativa por conjuntos.

- **Emplazamiento directo**

  Es la técnica más sencilla y se basa en la asignación fija de la misma línea de la caché para cada bloque. En la Figura 8 se ilustra este tipo de emplazamiento cuando se tienen *M* bloques en la memoria principal y *m* líneas en la caché. Para identificar qué bloque se está copiando en la caché se utiliza una etiqueta que está constituida por algunos bits adicionales a los de datos que se incluyen en las líneas.

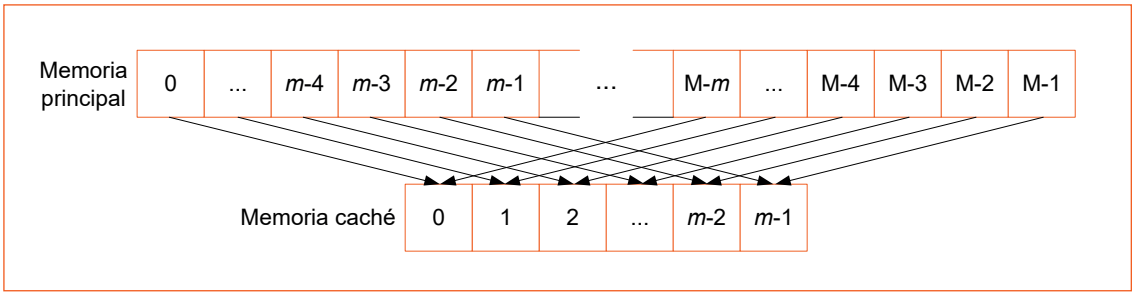

Figura 8. Emplazamiento directo para una caché de *m* líneas y una memoria principal de *M* bloques.

En la Figura 9 se muestra con más detalle la estructura de la dirección de memoria principal que se divide en tres campos. El campo menos significativo se conoce como palabra, este permite la identificación de la **palabra** a la cual se quiere hacer referencia dentro del bloque. El segundo campo (**bloque**) se utiliza para identificar la línea de la caché que ocupa el bloque que contiene a la palabra direccionada; mientras que el campo menos significativo (**etiqueta**) se destina a la etiqueta.

Figura 9. Estructura de la dirección de memoria principal en la política de emplazamiento directa.

La principal ventaja de esta política en comparación con las otras es su facilidad de implementación puesto que requiere de una baja complejidad a nivel de *hardware*. Adicionalmente, garantiza una alta velocidad de operación debido a que en la dirección se especifican todos los campos que permiten ubicar la posición exacta de cada bloque en la caché; y, no requiere de la implementación de algoritmos de reemplazamiento. Por otra parte, la principal desventaja radica en la elevada tasa de fallos que se presenta cuando se hace referencia a diferentes bloques que comparten la misma línea en la caché (Stallings, 2013).

Esta limitación se conoce como el fenómeno de **híper paginación** (*thrashing*, del inglés) y está condicionada por el hecho de que las ubicaciones en la caché son fijas.

- **Emplazamiento asociativo**

  Para solventar la limitación del emplazamiento directo, se utiliza la política de emplazamiento asociativo; en esta, los bloques de la memoria principal pueden ubicarse en cualquier línea de la caché. En este caso la dirección de la memoria principal solo incluye los campos **etiqueta** y **palabra**. La única forma que se tiene para poder identificar el bloque de datos direccionado es realizando una búsqueda en toda la caché. Para agilizar el proceso de búsqueda la caché se construye mediante memorias asociativas, las que permiten realizar la búsqueda en paralelo (Null & Lobur, 2003). A pesar de que esta política permite solventar la limitación de la elevada tasa de fallos presente en el emplazamiento directo, se requiere de una mayor complejidad del *hardware* para la implementación de la memoria asociativa. Otro inconveniente que posee esta técnica es la necesidad de implementar algoritmos de reemplazamiento para determinar qué bloques se desecharán cuando la caché está llena.

- **Emplazamiento asociativo por conjuntos**

  Este esquema permite combinar las potencialidades de las políticas anteriores, para lo cual la caché se divide en subconjuntos de varias líneas. En el asociamiento por conjuntos la dirección de la memoria principal está conformada por tres campos: **etiqueta**, **conjunto** y **palabra**. Los campos **etiqueta** y **palabra** tienen la misma función que en las políticas previas; mientras que, el campo **conjunto** permite identificar el subconjunto de la caché que contiene el bloque direccionado. Como resultado se tiene una técnica cuya complejidad y desempeño se encuentra en el intermedio entre el emplazamiento directo y asociativo. Otra ventaja de la política asociativa por conjuntos es que mediante esta se puede implementar el acceso paralelo a la palabra que se desea leer simultáneamente con la comprobación del bloque cargado en la línea de la caché (Lanchares Dávila, 2000).

3.2.3. Políticas de reemplazamiento

Cuando la caché se encuentra llena y el procesador intenta acceder a una dirección que no se encuentra en esta, es necesario reemplazar uno de los bloques existentes que contiene a la dirección solicitada; a este proceso se le conoce como **reemplazo**. **Cuando se utiliza la política de emplazamiento directa el proceso de reemplazo es único, debido a que los bloques de la memoria principal solo pueden ocupar una única línea de la caché**. Por otra parte, **cuando se emplea emplazamiento asociativo puro o por conjunto, se requiere de la implementación de un algoritmo que garantice un remplazo adecuado**. Uno de los requerimientos más importantes de cualquier algoritmo de remplazo es que sea implementado en *hardware* para garantizar una mayor velocidad.

Existen diversos algoritmos de remplazo, siendo uno de los más comunes el menos recientemente usado (**LRU**, por sus siglas en inglés); en este se reemplaza el bloque que haya estado la mayor cantidad de tiempo sin haber sido referenciado. Otra política común es la **primero entra-primero sale** (**FIFO**, por sus siglas en inglés), en la que el bloque que haya permanecido el mayor tiempo en la caché es reemplazado independientemente de haber sido referenciado recientemente. Otra

solución posible es la conocida como uso menos frecuente (**LFU**, por sus siglas en inglés); en este se remplaza el bloque que haya sido referenciado la menor cantidad de veces. Las técnicas anteriores tienen como desventaja que es necesario que se tenga guardado un historial de los accesos a cada bloque de la caché, lo que requiere disponer de un espacio de memoria para ello; además, hace más lento su funcionamiento. Una política que solventa la anterior dificultad es la basada en una **selección aleatoria**, donde el bloque a remplazarse se selecciona aleatoriamente. Un inconveniente de este esquema es que el remplazo puramente aleatorio es muy difícil de implementar, en su lugar se emplea una selección pseudoaleatoria que pudiera disminuir el comportamiento promedio del método.

3.2.4. Políticas de actualización

Una vez que se ha determinado el bloque de la caché que será remplazado, es necesario establecer lo que se hará con dicho bloque o con los bloques que deben ser modificados. En el caso de que se realice una escritura en alguno de los bloques de la memoria principal, de encontrarse dicho bloque en la caché, es indispensable actualizarlo para mantener la coherencia de los datos almacenados. En el caso de que se modifique un dato en la caché, las políticas de escritura de esta determinan la manera en que se actualizarán dichos datos en la memoria principal. Básicamente existen dos tipos de políticas de actualización:

- **Escritura directa:** permite la actualización tanto de la caché como de la memoria principal en cada proceso de escritura. Su principal ventaja es que siempre garantiza la consistencia de los datos en toda la jerarquía de la memoria, a expensas de siempre requerir, para ello, del acceso a la memoria principal lo que ralentiza considerablemente el funcionamiento del sistema.

- **Post-escritura:** actualiza los bloques en la memoria principal solo cuando estos son seleccionados para ser remplazados en la caché. La principal ventaja respecto a la escritura directa es que, generalmente, es más rápida debido a que no se requiere actualizar constantemente la memoria principal. Por otra parte, esta técnica posibilita que exista inconsistencia entre los datos almacenados en la caché y en la memoria principal. Otra desventaja de esta solución radica en que el acceso de los dispositivos de **E/S** a la memoria principal solo puede realizarse a través de la caché, lo que requiere de una mayor complejidad del *hardware*.

Un problema que se tiene en los computadores donde la memoria principal se comparte con varios procesadores es que, si en la caché de uno de estos se altera un dato existirá una inconsistencia tanto con la memoria principal como con el resto de la caché. Para prevenir este conflicto existen diferentes mecanismos como: la vigilancia del bus con escritura directa, transparencia del *hardware* y memoria no compartida, entre otros. (Stallings, 2013).

3.2.5. Otros parámetros de diseño

Dado que la principal función de la memoria caché es incrementar la velocidad de acceso a la memoria, esta se fabrica utilizando la tecnología **memoria estática de acceso aleatorio** (**SRAM**, por sus siglas en inglés); otra característica que permite incrementar la velocidad es que en ella se almacenan los

datos utilizados más recientemente. No es necesario que la memoria caché sea extremadamente grande para que ella posibilite un adecuado rendimiento. En algunas computadoras personales contemporáneas la caché L1 puede tener un tamaño entre 8K y 32K, mientras que la L2 puede ser de 256K a 512K. Una regla general a tener en cuenta para el dimensionamiento de la caché es hacer su tamaño lo suficientemente pequeño para que el coste promedio de los bits del sistema de memoria sea similar al coste de la memoria principal (Null & Lobur, 2003). Por otra parte, en el tamaño de la caché se tienen en cuenta otros aspectos que influyen en su rendimiento; por ejemplo, a pesar de que se utilice la misma tecnología de circuito integrado y se tenga la misma ubicación dentro de este, la caché de mayor tamaño tiende a ser ligeramente más lenta que las de menor tamaño (Stallings, 2013).

Una solución que permite incrementar la velocidad de acceso a la memoria es implementar la caché completamente o una parte de ella, dentro del mismo circuito integrado del procesador. En algunos computadores actuales se utiliza una caché de dos niveles, el primero de ellos (L1) se implementa dentro del procesador separando una parte destinada para los datos y otra para las instrucciones proporcionando altas velocidades; mientras que, el L2 se implementa fuera de este garantizando un mayor espacio de memoria que el L1.

Debido a que el proceso de escritura en memoria es lento, cuando se utiliza la política de actualización mediante escritura directa se requiere acceder a la memoria principal cada vez que se necesita actualizar el contenido de una dirección. Este acceso continuo a memoria hace que el desempeño del procesador se vea notablemente afectado debido a que este debe esperar a que se realice la escritura del nuevo dato para continuar con la siguiente instrucción. Para solventar esta limitación e incrementar el desempeño del procesador es posible utilizar un *buffer* de escritura donde el procesador pueda almacenar temporalmente el dato que se requiere actualizar en la memoria principal, de manera que no necesite esperar a que se realice la escritura para poder continuar con la siguiente instrucción. El *buffer* de escritura también puede implementarse con el propósito de mejorar el desempeño del procesador cuando se utiliza la política de actualización de post-escritura; en este caso los bloques seleccionados para ser actualizados en la memoria principal se almacenan en el *buffer* de escritura mientras el nuevo bloque es traído hacia la caché, posibilitando que el procesador no requiera esperar hasta que se realice la operación de escritura para utilizar los nuevos datos.

Una manera de disminuir el tiempo de espera cuando se necesita cargar en la caché el contenido de una dirección de la memoria principal es anticiparse a este requerimiento copiando en la caché bloques que todavía no han sido demandados por el procesador. Esta técnica se conoce como pre-búsqueda y puede realizarse tanto por *software* como por *hardware*. A pesar de que esta técnica implica una mayor complejidad y en algunos casos puede posibilitar la carga de bloques en la caché que no son utilizados de manera global, permite un mejor desempeño del procesador. (Lanchares Dávila, 2000).

Normalmente cuando ocurre un fallo al acceder a la caché, el procesador debe esperar que se atienda dicho fallo y se traiga de la memoria principal el bloque requerido imposibilitando que este pueda acceder a otra dirección de la caché. Un método empleado para reducir este tiempo de espera se logra mediante la utilización de **cachés no bloqueantes**. Este tipo de cachés permite al procesador, cuando ocurre un fallo de lectura, acceder a la caché de instrucciones mientras se carga la nueva información en la caché de datos. **Las cachés no bloqueantes pueden diseñarse para continuar operando bajo la ocurrencia de múltiples fallos**, esta particularidad es de gran importancia en los

computadores segmentados donde se realizan simultáneamente múltiples instrucciones. De esta manera se garantiza que la ocurrencia de un fallo en una instrucción no impida la ejecución del resto de las instrucciones.

3.3. Memoria virtual

Cuando el tamaño y la cantidad de los programas que se ejecutan en un computador imposibilitan que puedan ser cargados completamente en la memoria principal, se hace necesario guardar la parte inactiva de estos en algún dispositivo de almacenamiento secundario (normalmente discos duros). Cuando el procesador requiere acceder a un segmento del programa que no se encuentra en la memoria principal, estos son cargados desde el dispositivo externo; dando lugar a la técnica conocida como **memoria virtual**. Esta técnica puede verse como una extensión de la jerarquía de la memoria, administrada por el sistema operativo, que permite aumentar el tamaño limitado de la memoria principal de manera similar a como lo hace la caché con los registros. Es por ello que el diseño de la memoria virtual se basa en los mismos principios empleados para el diseño de la memoria caché, diferenciándose principalmente en los detalles de implementación.

Las direcciones utilizadas en un sistema de memoria virtual se conocen como direcciones virtuales o lógicas, por lo que se hace necesario realizar una traducción de estas a direcciones físicas mediante la **MMU**. Cuando la sección del programa que se está ejecutando reside en la memoria principal, la **MMU** convierte directamente la dirección virtual en la dirección de la memoria principal correspondiente posibilitando así, el acceso a la memoria de manera usual. En el caso contrario, la **MMU** indica al sistema operativo la necesidad de buscar el bloque que contiene dicha dirección en el dispositivo de almacenamiento secundario. Teniendo en cuenta la forma en que se asocian los bloques de palabras existen diferentes tipos de memoria virtual; estos pueden clasificarse en: paginados, segmentados y paginados-segmentados.

3.3.1. Mecanismos de traducción de direcciones

Los mecanismos de traducción de direcciones virtuales a físicas se clasifican en directos, asociativos y basados en un ***translation lookaside buffer*** (**TLB**) (Lanchares Dávila, 2000). Estas técnicas tienen en común que todas emplean una estructura conocida como **tabla de páginas**; dicha estructura contiene la información sobre la ubicación de las páginas de los programas que se están ejecutando, asociando cada dirección virtual con su correspondiente dirección física.

En el mecanismo directo la dirección virtual se divide en dos campos, uno para el número de la página virtual y el otro para el *offset*. El número de página virtual permite direccionar de manera directa la posición correspondiente de la tabla de páginas; cuando este campo hace referencia a una página válida, entonces el contenido de dicha página indicará la dirección física buscada. En el caso en que la tabla no contenga un número de página física válido, se produce un fallo de página, ante lo cual se requiere buscar la página requerida en el dispositivo de almacenamiento secundario. Este mecanismo posee como ventaja la simplicidad del direccionamiento directo, a expensas de requerir un tamaño considerable para la tabla de páginas. Este gran tamaño implica que, generalmente, la tabla sea implementada en la memoria principal; haciendo que la velocidad de los programas disminuya considerablemente por la necesidad de realizar dos accesos a memoria por cada lectura de instrucción.

Una manera de solucionar la desventaja del mecanismo anterior es empleando la **traducción asociativa**; en esta al igual que en la traducción directa la dirección virtual está constituida por los mismos campos. La diferencia de esta técnica radica en que cada elemento de la tabla está constituido por dos campos: el número de página virtual y el número de página física; lo que posibilita reducir considerablemente el tamaño de dicha tabla. A pesar de la ventaja que ofrece este método, posee el inconveniente de que se requiere implementar una búsqueda para hacer coincidir el número de página virtual de la dirección con todos los almacenados en la tabla, razón por la cual se necesita de una complejidad adicional del *hardware*.

Las ventajas de la simplicidad del mecanismo de traducción directa y la eficiencia de la traducción asociativa se pueden combinar dando lugar a la traducción asociativa por conjuntos. En esta técnica la dirección virtual se compone de tres campos: **la cabecera, el índice y el desplazamiento**. El campo índice permite identificar el conjunto donde se realizará la búsqueda de la dirección física (mecanismo directo), a partir del cual se verifica la coincidencia de todos los elementos de dicho conjunto con el valor de la cabecera (mecanismo asociativo). Este mecanismo proporciona tiempos de acceso reducidos con una complejidad moderada solamente en el caso de que la tabla sea pequeña, constituyendo su principal desventaja (Lanchares Dávila, 2000).

Todos los mecanismos de traducción de dirección anteriores tienen en común la necesidad de un acceso adicional a la memoria principal para acceder a la tabla de páginas influyendo negativamente en la velocidad. Para evitar este acceso adicional a la memoria principal, en la mayoría de los computadores modernos, se copia una parte de la tabla de direcciones en el **TLB**; este posee un tamaño muy pequeño, y contiene las entradas de la tabla de páginas referenciadas más recientemente (haciendo uso del principio de localidad temporal); para lo cual, se basa en la técnica de memoria asociativa. A pesar de su reducido tamaño, en el orden de 16 a 64 entradas según, posibilita elevadas tasas de acierto con tiempos similares a los que se tienen en el acceso a la memoria principal (Abd-El-Barr& El-Rewini, 2005).

3.3.2. Memoria virtual paginada

El método de memoria virtual paginada constituye la manera más común de implementar la memoria virtual; en esta la memoria principal se divide en bloques de palabras contiguos con un tamaño fijo. Estos bloques se conocen como marcos de páginas y constituyen la unidad básica de información. De manera similar las instrucciones de los programas se agrupan en conjuntos denominados páginas, cuyo tamaño es igual al de los marcos de páginas. En esta técnica las páginas no necesariamente deben estar almacenadas de manera contigua. Cuando en la memoria principal no se encuentra la página que contiene una dirección de memoria requerida por el procesador se produce un fallo de página.

La elección del tamaño de las páginas posee una importancia vital en el desempeño de la memoria virtual. Debe asegurarse que el tamaño de la página no sea muy pequeño para reducir la frecuencia con que se accede a la memoria secundaria la que requiere de elevados tiempos. Por otra parte, se debe garantizar que el tamaño de la página no sea excesivamente grande para evitar transferir un elevado volumen de información que ocuparía un valioso espacio en la memora principal y que luego no sea utilizada.

En dependencia del mecanismo de traducción de direcciones implementado, la dirección virtual estará conformada por diferentes campos; pudiéndose implementar cualquiera de las técnicas abordadas previamente. En la memoria principal se almacenan las tablas de páginas para cada uno de los procesos que se estén ejecutando; de esta forma la memoria asignada a cada proceso se divide en porciones de tamaño fijo igual al de las páginas. Si bien el uso de páginas y marcos de tamaño fijo facilita al sistema operativo la gestión de la memoria virtual, estos posibilitan la ocurrencia del fenómeno conocido como **fragmentación interna**, el cual se debe a la existencia de porciones de la página que no siempre son utilizadas.

3.3.3. Memoria virtual segmentada

En la memoria virtual segmentada, a diferencia de la variante paginada, el espacio de la memoria no se divide en porciones de igual tamaño; en lugar de esto, el espacio de memoria se divide en segmentos lógicos de longitud variable. Los segmentos están constituidos por un conjunto de elementos de datos contiguos, lógicamente relacionados. **Estas unidades de longitud variable permiten gestionar de manera eficiente espacios de memoria virtual muy grandes.**

Los segmentos contienen la dirección de memoria base y el desplazamiento respecto a esta posición que abarca todo su tamaño. Cada programa tiene asociada una tabla de segmentos donde se indica el inicio y el tamaño de cada uno de estos. En esta técnica la traducción de la memoria virtual a física se puede realizar mediante el método directo, asociativo o basado en **TLB**.

Debido a que la técnica de segmentación se basa en bloques lógicos de longitud variable, se hace más simple la protección y la compartición en comparación con la memoria paginada. Si bien la memoria virtual segmentada permite solventar el problema de la fragmentación interna, esta técnica se ve afectada por la fragmentación externa; este fenómeno ocurre debido a la gran cantidad de pequeños espacios de memorias disjuntos que aparecen cuando los segmentos son liberados. Estos espacios disjuntos en conjunto pueden tener la capacidad de un nuevo segmento que se requiera cargar; pero, debido a que no son contiguos imposibilita su uso. La fragmentación externa puede combatirse mediante la utilización de un mecanismo similar al recolector de basura que permite compactar la memoria virtual; no obstante, este proceso consume un tiempo considerable.

3.3.4. Memoria virtual segmentada-paginada

La combinación de los dos métodos anteriores da lugar a la memoria virtual segmentada-paginada, la cual ofrece las ventajas de ambos. Esta técnica puede implementarse de dos maneras diferentes: una basada en la segmentación lineal y la otra en un espacio de nombres segmentados (Hwang & Briggs, 1984). En ambos enfoques la memoria virtual se divide en segmentos de longitud variable, y a su vez estos segmentos son divididos en páginas de longitud fija. De esta forma cada programa posee una tabla de segmento, donde cada una de sus entradas posee una tabla de página. Las direcciones virtuales están constituidas por tres campos: el número del segmento, el número de la tabla y el desplazamiento dentro de la página. A pesar de que la memoria virtual segmentada-paginada ofrece la combinación de las ventajas del uso de cada una de las técnicas que la componen por separado, requiere de una complejidad del *hardware* mucho mayor.

4. Sistemas de almacenamiento

4.1. Discos magnéticos

Este tipo de dispositivo ha sido el precursor del sistema de almacenamiento externo de la mayoría de los computadores. Está constituido por uno o varios substratos circulares no magnéticos que son recubiertos por una película fina de un material magnético, usualmente en ambas caras como puede apreciarse en la Figura 10. Estos discos se hacen girar alrededor de las **cabezas** de lectura/escritura, permitiendo así almacenar los datos o acceder a estos. Estas **cabezas** están constituidas por bobinas conductoras que pueden ser fijas o móviles. A través de estas bobinas circula una corriente eléctrica con diferente polaridad para cada uno de los dígitos binarios. En dependencia de la rigidez del substrato se pueden tener discos magnéticos rígidos o flexibles; estos últimos fueron un medio de almacenamiento extraíble muy popular en las primeras computadoras personales que ya son obsoletos.

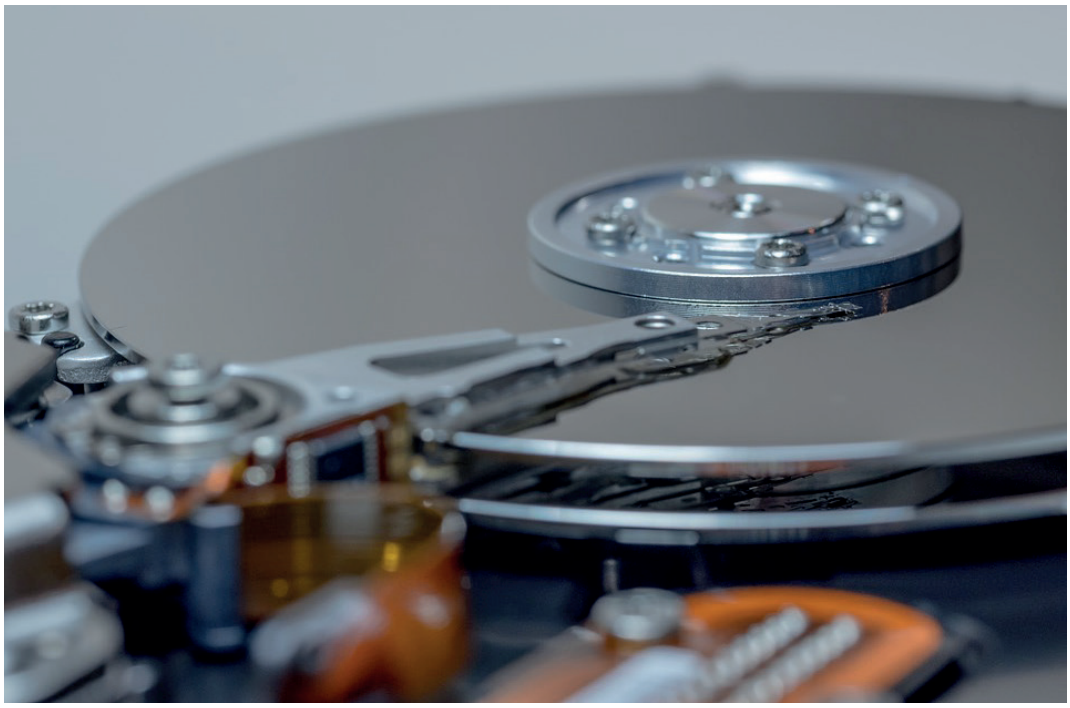

Figura 10. Disco duro magnético. Recuperado de https://pixabay.com/es/photos/unidad-de-disco-duro-hdd-hardware-611514/

Para identificar la posición de inicio y el fin de cada bit en los discos más antiguos se requería una señal de sincronía dedicada, para lo cual se reservaba una porción del disco donde se almacenaba dicha señal. En los discos más recientes se utiliza una manera más eficiente de representar los bits sin necesidad de requerir una señal de sincronía adicional; para ello, se emplean códigos auto-sincronizables como por ejemplo el Manchester o codificación de fase. La principal desventaja de este código es que posee una baja densidad de almacenamiento. Esta desventaja se ha solventado a partir del desarrollo de otros tipos de códigos más compactos que requieren de una mayor complejidad para su funcionamiento.

La escritura se basa en el hecho de que el campo magnético resultante de la circulación de pulsos de corriente a través de la cabeza permite obtener diferentes patrones magnéticos que son grabados en la superficie del disco. Por otra parte, la lectura se realiza a partir de la corriente eléctrica que se induce en la cabeza producto de la variación del campo magnético en su cercanía. En los discos duros magnéticos más recientes hay una **cabeza** de lectura constituida por un sensor magneto resistivo que produce diferentes variaciones de voltaje en dependencia los patrones magnéticos presentes (Stallings, 2013).

Los discos están divididos en diferentes circunferencias concéntricas conocidas como **pistas** con un ancho igual al de la cabeza. Las **pistas** adyacentes están separadas entre sí por un espacio que permite disminuir los errores que pueden producirse debido a una mala alineación de la cabeza. A su vez cada una de las **pistas** está dividida en **sectores** que pueden tener una longitud fija o variable; siendo la primera de estas variantes la más común (Stallings, 2013).

Debido a la diferencia de las velocidades tangenciales en las diversas **pistas**, una manera de garantizar la misma tasa de transferencia se logra asignando longitudes desiguales a los **sectores** pertenecientes

a **pistas** diferentes manteniendo constante la velocidad angular. Este método tiene como ventaja que permite direccionar los bloques de datos directamente por **sectores** y **pistas** requiriendo una menor complejidad en el *hardware*. Por otra parte, su principal desventaja radica en la pobre densidad de bits por unidad de longitud en las **pistas** exteriores en comparación con las interiores. Esta limitación se solventa en los discos que emplean la técnica de múltiples zonas, la cual divide cada una de las **pistas** en **sectores** de igual tamaño.

Es indispensable que las **cabezas** de lectura/escritura se encuentren lo más próximas posible a la superficie magnetizada del disco para asegurar una mejor interacción del campo magnético y de esta forma garantizar un adecuado funcionamiento y altas densidades de bits. Teniendo en cuenta la posición de las **cabezas** existen tres variantes diferentes de discos magnéticos.

- En la primera de estas la **cabeza** se encuentra a una distancia fija de la superficie de los discos sin llegar a entrar en contacto con esta.
- En la segunda variante existe el contacto físico entre las **cabezas** y la superficie de los discos.
- La última de las categorías se conoce como discos Winchester y al igual que en la primera no existe contacto físico entre la **cabeza** y la superficie del disco. En este caso el espacio existente se reduce considerablemente gracias a que todo el mecanismo se encuentra sellado herméticamente garantizando que no existan impurezas. Esta menor distancia entre la **cabeza** y la superficie permite una interacción magnética más eficiente entre estos, incrementándose la densidad de bits por unidad de longitud pudiendo de esta forma aumentar la capacidad de los dispositivos.

Para evaluar el desempeño de los discos magnéticos se deben tener en cuenta los diferentes parámetros implicados en los procesos implicados en las operaciones de lectura y escritura. La primera operación que debe realizar el disco bien sea para acceder a un dato existente o guardar uno nuevo, es localizar la posición de la **pista** requerida para ubicar la **cabeza** en esta. A la demora requerida durante este proceso se le conoce como **tiempo de búsqueda**; en los discos actuales los valores típicos pueden oscilar entre 50 y 100 ms (Stallings, 2013).

Una vez que la **cabeza** está ubicada en la **pista** debe localizarse el **sector** deseado, al lapso de tiempo requerido en este proceso se le conoce como retardo rotacional y está relacionado con la velocidad de rotación del disco teniendo un valor promedio de 2 ms (para media rotación) para discos de 15000 rpm (Stallings, 2013). La suma del tiempo de búsqueda y el retardo rotacional conforman el tiempo de acceso.

Una vez localizada la **pista** y el **sector** deseados, se realiza la operación de escritura o lectura según sea el caso, siendo necesario para la realización de estas operaciones de un lapso de tiempo conocido como **tiempo de transferencia de los datos**, este parámetro depende tanto de la velocidad de rotación del disco como de la densidad de datos por unidad de longitud, pero en cualquier caso es mucho menor que el tiempo de búsqueda y el retardo rotacional. El comportamiento anterior se debe a las demoras involucradas en las operaciones mecánicas requeridas para la ubicación de los sectores requeridos. Estas demoras son las responsables de los elevados tiempos de acceso de estos dispositivos lo cual constituye su principal limitación.

4.2. Arreglo redundante de discos independientes

Una manera de disminuir el tiempo de acceso en los discos magnéticos se logra mediante la utilización de varios de estos dispositivos funcionando en paralelo. Esta técnica se conoce como **Arreglo Redundante de Discos Independientes** (**RAID** por sus siglas en inglés). La posibilidad de utilizar simultáneamente varios discos permite que la información se distribuya de múltiples formas, dando lugar a diferentes configuraciones de esta técnica. Inicialmente se definieron seis esquemas diferentes para la operación de esta técnica conocidos como niveles RAID 0 al 5. El término de niveles no implica que exista una estructura jerárquica, solo se refiere a diferentes formas de implementar esta técnica. Todas estas variantes de RAID poseen en común que el sistema operativo reconoce al arreglo de discos (unidades físicas) como una única unidad lógica, los datos se distribuyen entre todas las unidades físicas y se almacena información redundante para facilitar la recuperación de los datos en caso de ocurrir un fallo.

- En el **nivel 0 de RAID la capacidad de todos los discos es combinada de forma tal que el sistema operativo reconozca una unidad de almacenamiento equivalente a la suma de las capacidades individuales**. En esta variante no existe redundancia de la información, de forma que si llegara a fallar algún disco es imposible recuperar la información almacenada en este. En RAID 0 la capacidad total del arreglo se divide en un número entero de tiras de datos. En un conjunto de **n** unidades físicas, la información se va distribuyendo sucesivamente por capas de **n** tiras a través de las **n** unidades físicas. El desempeño de este nivel aumenta a medida que se incrementan las peticiones de múltiples tiras contiguas debido a la posibilidad de acceder a estas de forma paralela. Por otra parte, el desempeño se ve afectado en los sistemas operativos que encuestan secuencialmente cada sector para el manejo de datos ya que en estos casos no es posible explotar el paralelismo.

- El nivel **RAID1 emplea un número par de unidades físicas agrupadas en dos conjuntos de manera que uno sea un espejo del otro, permitiendo así duplicar completamente la información**. Cuando se realiza una operación de escritura se ejecuta en ambos conjuntos por lo que el desempeño de esta operación no es mejor que en el caso de que se utilice un único disco. Por otra parte, el desempeño de la operación de lectura se puede realizar simultáneamente desde ambos conjuntos; disminuyendo a la mitad, en el peor de los casos, el tiempo de lectura en comparación con el de un disco único. Esta variante posee una gran tolerancia a fallos a expensas de su elevado coste ya que requiere duplicar el doble del espacio físico que se necesita soportar.

- En el nivel **RAID2 el tamaño de las tiras de datos usualmente es mucho menor que en los niveles anteriores, pudiendo llegar a ser del tamaño de una palabra o de un *byte***. Este menor tamaño de las tiras posibilita el acceso paralelo de manera sincronizada a todos los discos del arreglo donde se encuentran distribuidos los bits correspondientes a la palabra codificada (bits de datos y de paridad) para la corrección de errores. Normalmente se emplea el **código Hamming**, este permite detectar patrones de dos errores por bloque y corregir patrones de un error por bloque. A pesar de que en esta variante se requiere una menor cantidad de unidades físicas que en **RAID1**, sigue siendo una opción cara debido al número de unidades necesarias para almacenar los bits de paridad. Esta variante es útil para escenarios donde ocurren una gran cantidad de errores en los discos.

- El nivel **RAID3 posee la misma dimensión para las tiras de datos que el RAID2; la diferencia respecto a este último radica en que solo se emplea un bit de paridad para la detección de errores**. En el caso de que falle una de las unidades físicas, este bit de paridad además de detectar la ocurrencia del fallo permite corregirlo una vez reemplazada la unidad defectuosa. Este método posibilita elevadas tasas de transferencia de datos, aunque su desempeño se ve limitado debido a que solo puede ejecutar una operación de lectura o escritura a la vez.

- En el nivel **RAID4 al igual que en RAID3 se emplea un dispositivo físico para almacenar la información de paridad, con la diferencia de que en este caso en lugar de almacenarse un bit de paridad se almacena una tira de paridad correspondiente a cada una de las tiras de datos**. Cada uno de los dispositivos físicos constituye una unidad de acceso independiente, por lo que es posible realizar de manera paralela tantas operaciones de lectura/escritura como dispositivos para el almacenamiento de datos haya en el arreglo. Esta particularidad proporciona una gran utilidad a esta variante en escenarios donde se requiere un gran número de peticiones de lectura/escritura y bajas velocidades de acceso. De manera similar a **RAID3** la información de paridad permite recuperar la información contenida en cualquiera de los discos de datos en caso de que alguno de estos falle. El principal inconveniente de **RAID4** radica en su bajo desempeño en pequeñas actualizaciones de datos, para lo cual no solo es necesario actualizar los datos en el disco correspondiente, sino que es imprescindible actualizar la tira de paridad.

- La limitación presente en la variante anterior se solventa en **RAID5** y **RAID6**. **En la primera de estas versiones se calcula una única tira de paridad que es distribuida uniformemente de manera cíclica a través de todas las unidades que conforman el arreglo. En el caso de RAID6 se realizan dos cálculos de paridad de manera diferente y son almacenados en unidades físicas independientes.** Al igual que en **RAID5** las tiras de paridad son distribuidas de manera cíclica entre todos los discos que conforman el arreglo. La duplicación de la paridad presente en **RAID6** le proporciona una protección adicional contra la ocurrencia de fallos, ya que puede recobrar los datos en el caso extremo de que fallen simultáneamente dos unidades físicas.

4.3. Dispositivos ópticos

El desarrollo y posterior auge de los sistemas de audio digital basados en discos compactos posibilitó la aparición de una gran variedad de dispositivos de almacenamiento ópticos. Esta tecnología ha facilitado la implementación masiva de dispositivos de almacenamiento extraíbles con una alta fiabilidad a un coste mucho menor que el de los discos magnéticos. Diversos fabricantes han propuesto una considerable variedad de formatos de discos ópticos; a continuación, se exponen las particularidades de algunos de las variantes más populares de estos medios de almacenamiento.

- **Discos compactos**

  Inicialmente los discos compactos (**CD** por sus siglas en inglés) se desarrollaron para el almacenamiento de audio digital; posteriormente fue utilizado como dispositivo de almacenamiento de datos, dentro de los cuales se tienen el **CD** de solo lectura (**CD-ROM**, por

sus siglas en inglés) y el **CD**-grabable (**CD-R**, por sus siglas en inglés). Todas las variantes de discos se fabrican a partir de una resina poli carbonatada transparente y rígida a la que se le añade una película fina de un material altamente reflectivo donde es grabada la información digital en pequeños agujeros realizados mediante un láser de alta intensidad, el cual altera de manera diferente la reflectividad de la superficie. Los dígitos binarios se identifican mediante una combinación de zonas con una alta dispersión denominados **hoyos** y otras de alta reflectividad conocidos como **valles**. La superficie del material reflectivo se protege de la acción de la suciedad y otros agentes externos mediante una capa de acrílico. La lectura de la información se realiza mediante un fotodetector que capta la luz reflejada por el disco al incidir sobre este un haz proveniente de un láser de menor intensidad. Los cambios en la intensidad de la luz reflejada debido a las diferentes formas en que se dispersa esta cuando el haz incide en los **hoyos** o los **valles** permiten diferenciar los dígitos binarios.

A diferencia de los métodos de distribución de la información en los discos magnéticos, donde se tienen **pistas** concéntricas divididas en **sectores** que pueden tener longitudes fijas o variables en dependencia de su posición relativa al centro, en la mayoría de los discos compactos la información se organiza a lo largo de una espiral que recorre toda la superficie del mismo de manera que se mantenga constante la velocidad lineal durante la lectura y/o escritura (Null & Lobur, 2003). En los **CD** los datos son agrupados en **sectores** de una longitud de 2352 bytes; cada **sector** posee diferentes campos dentro de los cuales se tienen: la sincronía, el encabezado y la carga útil. Los **sectores** pueden tener tres tipos de modos diferentes (0, 1 y 2), para cada uno de los cuales se tienen ligeras modificaciones en el formato, principalmente en el campo destinado a la carga útil. Los modos 0 y 2 se emplean para la grabación de audio digital, por lo que no poseen mecanismos de corrección de errores. El modo 1 se emplea para el almacenamiento de datos; en este se destina una parte del campo de la carga útil para guardar los bits de redundancia de los algoritmos para la detección y corrección de errores (Null & Lobur, 2003).

La principal desventaja que presentan los **CD** es que solo permiten una sola operación de escritura en ellos, razón por la cual se desarrolló una variante de estos que permiten múltiples escrituras conocido como **CD** reescribible (**CD-RW**, por sus siglas en inglés). La superficie reflectiva de los CD-RW se construye a partir de una aleación especial que posee dos estados estables diferentes (amorfo y cristalino) cada uno con diferente reflectividad. Estos estados pueden conseguirse en dependencia de la forma en que se calienta y se enfría el material (Hamacher *et al.*, 2012). De esta manera se puede modificar de manera controlada la reflectividad de la superficie para representar los **hoyos** y **valles** en múltiples procesos de escritura.

- **Discos digitales versátiles**

Los discos digitales versátiles (**DVD**, por sus siglas en inglés) constituyeron un salto cualitativo en los dispositivos de almacenamiento óptico debido a los grandes volúmenes de información que estos ofrecen en comparación con los **CD**. Los **DVD** permitieron el desarrollo del video digital ofreciendo una calidad en la imagen sin precedentes y la posibilidad de almacenar información adicional como por ejemplo subtítulos y traducciones. Los **DVD** comparten

las mismas características generales de diseño y dimensiones que los **CD**, estos también se fabrican a partir de una resina poli carbonatada transparente y rígida a la que se le añade una película fina de un material altamente reflectivo que contiene los **hoyos** y **valles**. Los cambios en la intensidad de la luz reflejada en la superficie con diferentes reflectividades permiten diferenciar los dígitos binarios.

La mayor capacidad de almacenamiento del **DVD** se debe a varias mejoras respecto a los **CD** como, por ejemplo: menores punto focal y espaciamiento entre las pistas contiguas debido a la menor longitud de onda del láser, posibilidad de tener dos capas para la grabación una semi-reflectante y otra completamente reflectante y posibilidad de grabación por ambas caras. De esta forma se pueden tener **DVD** de una o dos capas y de una o dos caras, llegando a tener, en el mejor de los casos, una capacidad de más de 17 GB (Stallings, 2013).

Al igual que sus antecesores, los **DVD** pueden encontrarse en versiones de solo lectura (**DVD-ROM**, por sus siglas en inglés), grabables (**DVD-Recordable**, por sus siglas en inglés)y re-escribible (**DVD-RW**, por sus siglas en inglés). El continuo progreso en la tecnología de láseres ha posibilitado el desarrollo de nuevas versiones de DVDs incrementando notablemente la capacidad; ejemplo de ello lo constituye el estándar conocido como *Blue-Ray*. Esta variante se basa en un láser azul cuya menor longitud de onda, comparado con los rojos utilizados en los **DVD** convencionales, posibilita una menor dimensión para los **hoyos, valles y pistas** conllevando a un incremento notable en la capacidad en el orden de 25 GB para el caso de un disco de una sola cara. Otro aspecto que ha ido revolucionando paulatinamente en la tecnología de almacenamiento óptica es la tasa de datos; no obstante, aún no es comparable con las alcanzadas por los dispositivos magnéticos (Tanenbaum & Austin, 2013).

4.4. Dispositivos de estado sólido

Los dispositivos de almacenamiento secundarios de estado sólido están construidos a partir de componentes electrónicos semiconductores no volátiles; estos pueden emplearse tanto para unidades de almacenamiento externo como interno. Uno de los dispositivos de almacenamiento externo de mayor uso son las **memorias *flash***; estas se fabrican mediante celdas de memorias basadas en transistores de compuerta flotante que permiten almacenar de manera no volátil los voltajes que representan los niveles lógicos. En dependencia de la lógica digital en las cuales se basan existen dos tipos de memorias *flash*: NOR y NAND (Stallings, 2013).

En las memorias NOR la menor unidad de almacenamiento es el bit y su organización lógica se basa en esta operación digital, este tipo de memoria posee una alta velocidad de acceso lo que le proporciona una gran utilidad para la ejecución de sistemas operativos de equipos móviles. Por otra parte, en las memorias NAND la información se agrupa en conjuntos de 16 o 32 bits con una organización lógica basada en esta operación digital; estas memorias se utilizan ampliamente en dispositivos de almacenamiento externo.

Otro tipo de dispositivo de almacenamiento de creciente relevancia en los últimos tiempos que emplea esta tecnología son los **discos de estado sólido** (**SSD**, por sus siglas en inglés). Al igual que las memorias *flash*, los **SSD** se fabrican mediante celdas de memorias no volátiles basadas en

transistores de compuerta flotante. La principal ventaja de estos dispositivos en comparación con los discos magnéticos es su menor tiempo de acceso debido a que no se requiere de ningún mecanismo móvil. La ausencia de estos mecanismos en los **SSD** le proporciona una mayor confiabilidad en equipos portátiles donde los movimientos pueden afectar el desempeño de los discos magnéticos. Si bien el uso de los **SSD** ha ido incrementándose, todavía no reemplazan completamente a los discos magnéticos; principalmente por los elevados costes de fabricación que aún siguen siendo mucho mayores que el de su rival. Otra limitación de los **SSD** es que presentan un número finito de ciclos de escritura que generalmente se encuentran en el orden de los 100,000 (Tanenbaum & Austin, 2013).

5. Repertorio de instrucciones

El conjunto de las diferentes instrucciones máquina o instrucciones del computador constituyen el **repertorio de instrucciones del procesador**. La definición de dicho repertorio constituye unas de las etapas más críticas en el diseño de un computador, ya que estas **definen las operaciones que se realizan y permiten describir el modelo de la programación que se puede ejecutar en este**. Cualquier repertorio de instrucciones debe garantizar la ejecución de las operaciones requeridas en el computador de una manera eficiente; ello implica la necesidad de maximizar el desempeño y la eficiencia con el mínimo coste posible de la implementación de las instrucciones a nivel de *hardware*. Otras características deseables del repertorio de instrucciones son la facilidad para su implementación, el bajo consumo energético y que sea completo. La definición del repertorio de instrucciones requiere de la definición de los siguientes parámetros:

- Tipo de instrucciones que se pretenden realizar.
- Tipo de representación de datos.
- Modos de direccionamiento.
- Formatos de las instrucciones.

Una instrucción debe contener todos los elementos que incorporan la información necesaria para poderse ejecutar; estos elementos son: el código de operación, la referencia a los operandos fuente

y origen, referencia al operando destino y la referencia a la siguiente instrucción (Stallings, 2013). Los operandos de origen pueden estar localizados tanto en la memoria principal o virtual, como en los registros del procesador o en un dispositivo de **E/S**. Las instrucciones deben cumplir con un conjunto de propiedades esenciales como, por ejemplo:

- Deben ser sencillas y realizar una única función.
- Deben tener un número finito de operando.
- Cada operando debe tener una representación determinada.
- Debe sistematizarse su codificación de forma que se garantice su decodificación.
- Deben ser independientes y autocontenidas.

En el lenguaje de máquina todas las operaciones se realizan mediante un conjunto de instrucciones elementales básicas que sean capaces de facilitar la realización de cualquier instrucción en un lenguaje de programación de alto nivel. De manera general, las instrucciones pueden agruparse en cuatro grupos (Stallings, 2013):

- **Procesamiento de datos:** incluye las instrucciones para operaciones aritméticas que permiten el manejo de los datos numéricos, y las instrucciones lógicas para el manejo de datos booleanos.
- **Almacenamiento de datos:** incluye las instrucciones de memoria que permiten la transferencia de los datos entre la memoria y los registros.
- **Transferencia de datos:** incluye las instrucciones de **E/S** permitiendo la transferencia de programas y datos entre los dispositivos de **E/S** y la memoria.
- **De control:** incluye las instrucciones de bifurcación basadas o no en condiciones y las instrucciones de control que permiten comprobar la validez de un dato o el estado de un cálculo.

Las instrucciones que se realizan en el computador poseen información, tanto de las operaciones requeridas como de las direcciones de los operandos necesarios, para ejecutar dichas operaciones. Las instrucciones pueden clasificarse de acuerdo al número de operandos que implican en (Abd-El-Barr & El-Rewini, 2005):

- **Instrucciones de tres direcciones:** son aquellas que realizan una operación en la cual se combina el contenido de dos operandos y su resultado es almacenado en un tercero. Cabe destacar que todos los operandos implicados son de un mismo tipo, ya sean direcciones de memorias o registros.
- **Instrucciones de dos direcciones:** son aquellas que realizan una operación en la cual se combina el contenido de dos operandos y su resultado es almacenado en uno de ellos. Al igual que en la clasificación anterior, todos los operandos implicados son de un mismo tipo, ya sean direcciones de memorias o registros.

- **Instrucciones de una dirección:** son aquellas que realizan una operación en la cual se combina el contenido de un operando con el del registro acumulador y su resultado se almacena en este último.

- **Instrucciones de una dirección y media:** al igual que las instrucciones de dos operandos, realizan una operación en la cual se combina el contenido de dos operandos y su resultado es almacenado en uno de ellos, con la diferencia de que en esta los operandos son diferentes (registros y direcciones de memoria); por lo que es necesario utilizar dos tipos de direccionamientos diferentes.

- **Instrucciones con cero dirección:** son aquellas que como su nombre lo indica no requiere ningún operando. Estas instrucciones utilizan las operaciones con la pila, para lo cual emplean un registro especial conocido como puntero de la pila.

La selección del número de direcciones por instrucción es un aspecto importante y a menudo complicado en el diseño del repertorio, donde se contraponen la complejidad y versatilidad del computador. Un menor número de direcciones trae como beneficio la utilización de instrucciones más sencillas que requieren de procesadores menos complejos, a expensas de incrementar el tamaño y la complejidad de los programas y por consiguiente elevar su tiempo de ejecución. Las instrucciones con múltiples direcciones permiten el uso de varios registros de uso general, los cuales facilitan la realización de operaciones mucho más rápidas que en el caso de utilizarse memorias. Como resultado, en la mayoría de los procesadores se estila usar una combinación de diferentes formatos de instrucciones permitiendo así combinar las ventajas que ofrece cada uno de estos.

5.1. Parámetros de diseño

El diseño del repertorio de instrucciones trae aparejado la confluencia de un conjunto considerable de aspectos complejos a tener en cuenta que a menudo suelen contraponerse unos con otros. En este proceso es importante que el diseñador considere las necesidades de los programadores usuarios del computador con el objetivo de incorporar las instrucciones requeridas. Dentro de los aspectos de diseño más importantes a tener en cuenta se pueden mencionar:

- Tipos de instrucciones
- Tipos de datos
- Formato de las instrucciones
- Registros
- Modos de direccionamiento

5.1.1. Tipos de instrucciones

Existe una amplia variedad de operaciones que pueden implementarse en un computador. A pesar de esta gran diversidad en la mayoría de los computadores se encuentran un grupo de instrucciones que se clasifican en los siguientes grupos:

- **Transferencia de datos:** constituye el tipo de instrucción más básico y permiten mover datos provenientes de un origen hacia un destino determinado. Tanto la fuente como el destino pueden ser un registro, una dirección de memoria o la cabecera de la pila. Las instrucciones de transferencia de datos deben especificar: la posición de los operandos origen y destino, la longitud de los datos a transferir y el modo de direccionamiento que se utilizará. Las diferentes maneras de especificar estos parámetros posibilitan la definición de una amplia variedad de instrucciones para la transferencia de datos. Por ejemplo, la posición de los operandos debe especificarse en el campo correspondiente al **codop** o en los correspondientes a los operandos; pueden definirse diferentes tamaños de datos (ej. 8, 16, 32, 64 bits), existen una amplia variedad de modos de direccionamiento.

- **Aritméticas:** en la mayoría de los computadores se implementan instrucciones aritméticas básicas como la suma, resta, multiplicación y división; siendo la primera de estas la operación más básica de todas. Estas operaciones se implementan siempre para números enteros de coma flotante. Otras de las instrucciones que suelen implementarse son las encargadas de las operaciones de punto flotante, empaquetado decimal, valor absoluto, negación, incremento y decremento, entre otras.

- **Lógicas:** son instrucciones de gran importancia que se implementan en la mayoría de los computadores; siendo las más comunes AND, OR, NOT, XOR y EQUAL. Estas instrucciones operan a nivel de bit de forma independiente dentro de una palabra o de otra unidad direccionable. Las instrucciones de comparación son otro tipo de instrucciones que pueden realizarse mediante una resta o una operación XOR. Adicionalmente a las operaciones lógicas a nivel de bit mencionadas con anterioridad, existen otras que permiten el desplazamiento y rotación; donde se tienen los desplazamientos lógicos, aritméticos y cíclicos o de rotación.

- **Conversión:** son instrucciones que permiten el cambio del formato de los datos u operan sobre dicho formato. Un ejemplo de este tipo de instrucción es la que permite la conversión de un código decimal a binario.

- **Entrada/Salida:** permite la transferencia de información entre los registros de los periféricos en caso de que dichos periféricos no se encuentren mapeados directamente en memoria. Existe una gran diversidad de este tipo de instrucciones como las **E/S** programadas aisladas, **E/S** programadas asignadas en memoria, acceso directo a memoria y el uso de un procesador de **E/S**. En la mayoría de los computadores se dispone de un amplio repertorio de instrucciones para la **E/S** asignada a memoria, siendo más reducido las orientadas a **E/S** aislada. La gran variedad de instrucciones en el primero de estos modos tiene como inconveniente la mayor utilización de las direcciones de memoria que siempre son un recurso limitado.

- **Control del sistema:** son instrucciones reservadas para tareas específicas que generalmente son realizadas por el sistema operativo. Ejemplo de algunas operaciones de control son: lecturas y modificaciones de registros de control y claves de protección de memoria, acceso a bloques de control de procesos, entre otros.

- **Control de flujo:** permiten modificar la secuencia de ejecución normal de los programas en lenguaje de máquina que se ejecutan secuencialmente. Este tipo de instrucción modifica el

registro contador de programa haciendo que este salte hasta una dirección de memoria determinada permitiendo realizar programas de manera más eficiente. Dentro de este tipo de instrucción se encuentran las de bifurcación que permiten tomar decisiones no secuenciales como por ejemplo las instrucciones de **salto condicional e incondicional**. Otro tipo de instrucciones de control de flujo son las de **salto implícito**, donde se especifica de manera implícita la dirección de memoria a donde saltará el contador de programa. Un tipo de instrucciones perteneciente a este grupo con una amplia utilidad son las orientadas a la llamada de procedimientos. Esta categoría emplea dos instrucciones básicas para su funcionamiento, la de llamada y la de retorno.

5.1.2. Tipos de datos

Uno de los elementos relevantes a tener en cuenta en el diseño del repertorio de instrucciones es el tipo de datos que se manejará en el lenguaje de máquina; dentro de los más importantes se encuentran:

- **Direcciones:** son un tipo de datos que pueden considerarse números enteros sin signo que se utilizan para ciertas operaciones requeridas en algunos tipos de direccionamiento permitiendo determinar la referencia a un operando.

- **Números:** este tipo de datos es empleado en todos los lenguajes de máquina, siendo los más usuales: los enteros de punto fijo, reales con coma flotante y los decimales. Es importante recalcar que los números que pueden manejarse en un computador siempre tendrán una representación finita.

- **Caracteres:** representan una forma de datos común, son representados en formato binario mediante una cadena de bits para facilitar un mejor manejo de estos por los sistemas de cómputo y comunicación. El método de representación de más amplia utilidad es el **Código Estándar Estadounidense para el Intercambio de Información** (**ASCII**, por sus siglas en inglés). En este cada carácter se representa por un código binario de 7 bits, pudiendo representarse un total de 127 caracteres diferentes.

- **Datos lógicos:** son aquellos datos que se utilizan en operaciones lógicas, para cuya identificación resulta más eficiente emplear un único bit de manera independiente, el cual es considerado como un tipo de dato.

5.1.3. Registros

Los registros constituyen el nivel más alto de la jerarquía de memoria, por encima de la memoria principal y la caché y se emplean para obtener y almacenar, de una manera extremadamente rápida, los resultados de las operaciones y los cálculos de la CPU. La variedad en cuanto al número, tipo y tamaño de los registros permite definir diferentes conjuntos de registros en el computador. Algunos de los registros pueden ser visibles a nivel del repertorio de instrucciones; o, dicho de otra manera, son accesibles por los usuarios. Estos registros se emplean para el control de la ejecución del programa, almacenar temporalmente resultados y realizar tareas de propósito general. Una ventaja importante que ofrece este tipo de registro es que permite minimizar las referencias a memoria

principal. Por otra parte, existen registros que no son visibles por el usuario que, generalmente, son destinados a controlar el funcionamiento del procesador y la ejecución de los programas.

Los registros visibles por el usuario pueden clasificarse en dos categorías:

- **De propósito específico:** en esta categoría se encuentran registros con funciones específicas como por ejemplo los punteros a segmentos, registros de índice, el contador de programa, el puntero de la pila, entre otros.

- **De propósito general:** se emplean para almacenar variables locales y resultados intermedios de los cálculos, su principal función es la de proporcionar un acceso rápido a los datos de uso intensivo. Este tipo de registro puede contener tanto direcciones como datos, pudiendo existir casos donde se especifique el tipo de dato se maneja por cada registro. En algunos computadores estos son simétricos e intercambiables, dichas características les permiten realizar las mismas funciones (ortogonalización del repertorio de instrucciones). En la arquitectura RISC generalmente poseen como mínimo 32 registros de propósito general. Actualmente la tendencia es de aumentar el número de registros.

Adicionalmente a los registros de uso específico accesibles por los usuarios, siempre existe una variedad de registros de uso específico que solo es accesibles por el sistema operativo para tareas de control de memoria, caché y dispositivos de E/S entre otras.

El tamaño de los registros es un factor importante a tener en cuenta a la hora de realizar el diseño del computador. La longitud de los registros de datos debe ser suficientemente grande como para poder almacenar todos los tipos de datos disponibles. En algunos computadores es posible asociar registros contiguos para de esta forma manipular datos con el doble de tamaño. Por otra parte, los registros de direccionamiento deben tener una longitud tal que se pueda direccionar todo el rango de memoria; estos pueden diseñarse para su uso en un modo de direccionamiento específico o para cualquier modo.

La cantidad de registros de uso general de direcciones y de datos es un aspecto de diseño importante. A mayor número de registro, se requiere una mayor cantidad de bits en los operandos para su direccionamiento. Un menor número de registros aumenta la referencia a memoria; mientras que, una mayor cantidad no necesariamente la disminuye (Stallings, 2013).

La decisión de los tipos de registros a implementar también influye directamente en el diseño del repertorio de instrucciones. La utilización de registros de uso específico usualmente permite la identificación del tipo de registro que se utiliza en la instrucción en el **codop**. Con ello solo sería necesario indicar el registro específico dentro del conjunto de su tipo. Por otra parte, una mayor especificidad de los registros limita la flexibilidad de su uso.

Otro tipo de registros de gran importancia en todos los computadores son los de control y estado; estos se utilizan para controlar las operaciones del procesador y la ejecución de programas por parte del sistema operativo. Si bien la mayoría de ellos generalmente no son visibles por el usuario, algunos

pueden ser accesibles. Dentro de los registros de estado y control que pueden ser implementados en un computador se tienen:

- **Contador de programa** (**PC**, por sus siglas en inglés): contiene la dirección de la instrucción que se ejecuta en cada ciclo.
- **Registro de instrucción** (**IR**, por sus siglas en inglés): contiene la dirección de la instrucción ejecutada recientemente.
- **Registro de dato de memoria o intermedio de memoria** (**MDR** o **MBR**, por sus siglas en inglés): es un tipo de registro que solamente puede ser manipulado exclusivamente por el CPU; permite almacenar de manera intermedia la palabra de dato que se quiere leer o escribir en una dirección de memoria. No todos los procesadores disponen de él.
- **Registro de acceso de memoria** (**MAR**, por sus siglas en inglés): al igual que el MDR es para uso exclusivo de la CPU, y permite almacenar la dirección de memoria donde se desea leer o escribir una palabra de dato. No todos los procesadores disponen de él.

Un tipo de registro de estado implementado en muchos de los computadores, que puede ser parcialmente visible por los usuarios, es el registro de bandera o palabra de estado del programa (**PSW**, por sus siglas en inglés) que se utiliza para indicar el resultado de una operación mediante la activación, por *hardware*, de los bits de estados o banderas. Este tipo de registro se emplea en las instrucciones de salto condicional.

Un criterio que a menudo es tenido en cuenta para el diseño y organización de los registros de control y estado es el sistema operativo que será instalado en el computador. De esta forma es posible adaptar la organización de los registros a las exigencias del sistema operativo. De igual manera, es importante tener en cuenta a la hora del diseño la distribución de la información de control entre los registros y la memoria para poder especificar la cantidad de información de control que se requiere almacenar en estos.

5.1.4. Modos de direccionamiento

Básicamente existen dos tipos de operaciones que se pueden realizar con las memorias:

- La operación de lectura
- La operación de escritura

Para poder leer o escribir una palabra desde su ubicación en la memoria se requiere del direccionamiento de dicha memoria. Existen diversos modos de direccionamiento teniendo en cuenta la manera en que pueden ser direccionados los operandos. Esta variedad de modos posibilita diversas ventajas como, por ejemplo: ahorran espacio en memoria, simplifican las estructuras de datos complejas, proporcionan mayor flexibilidad en la programación en lenguaje de máquina y facilitan el diseño de los compiladores. A continuación, se presentan algunos de los modos de direccionamientos más comunes.

- **Direccionamiento inmediato**

  Es el más sencillo. En este se incluye el operando dentro de la instrucción de manera que no se requiere información adicional sobre su ubicación en la memoria. Un ejemplo de este tipo de direccionamiento se tiene cuando se desea asignar directamente un valor numérico a un registro de la forma **ASIGNAR** Valor al Registro. Este tipo de direccionamiento es poco recomendable debido a que, de requerirse una modificación del valor, sería necesario cambiar cada instrucción donde se utiliza dicho operando.

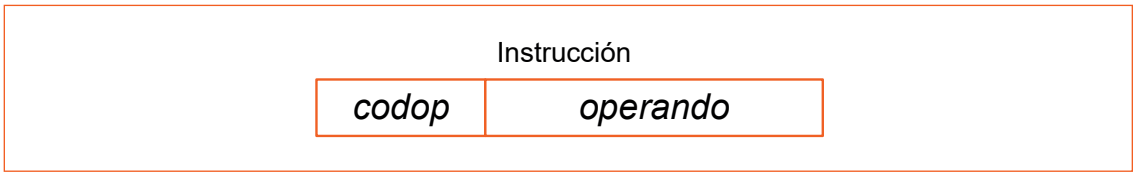

Figura 11. Modo de direccionamiento inmediato.

- **Direccionamiento directo o absoluto**

  En este se proporciona de manera explícita la dirección de memoria donde se encuentran los operandos implicados en la instrucción. Un ejemplo de utilización de este modo ocurre cuando se desea cargar el valor de una dirección en memoria a un registro de la manera **ASIGNAR** Valor en memoria al Registro.

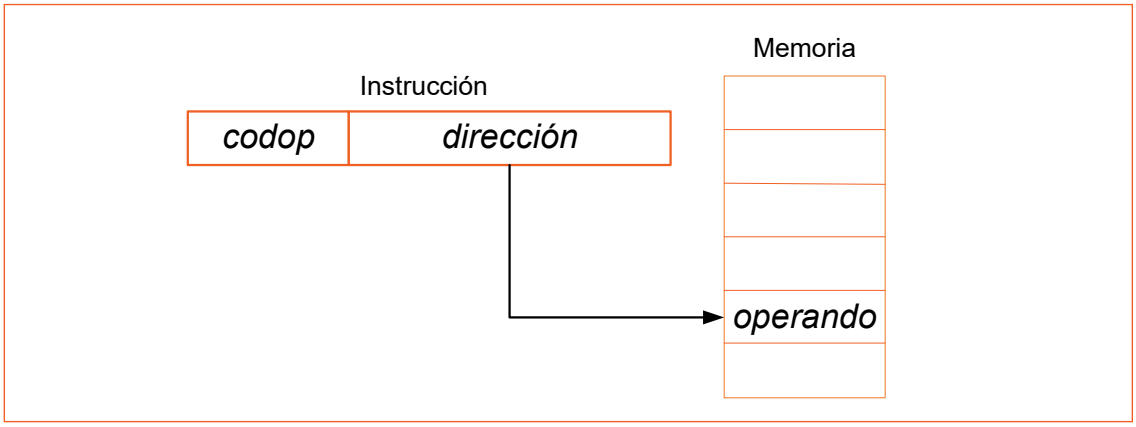

Figura 12. Modo de direccionamiento directo o absoluto.

- **Direccionamiento indirecto**

Permite mayor flexibilidad que el modo directo debido a que en lugar de proporcionar de manera explícita las direcciones de los operandos (ya sea una dirección de memoria o un registro), se asigna el lugar donde se encuentran guardadas dichas direcciones. En dependencia del tipo de operando que se utilice, se tienen los direccionamientos indirectos por memoria o por registro.

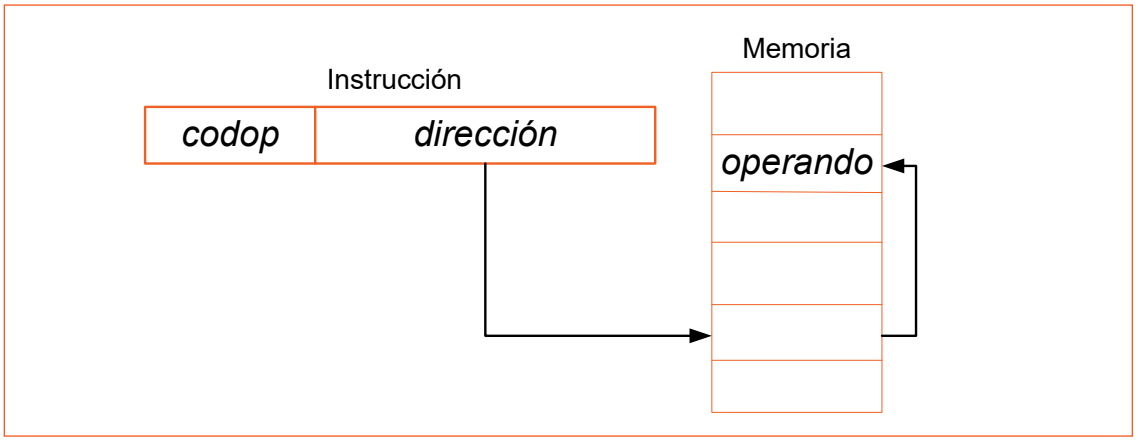

Figura 13. Modo de direccionamiento indirecto por memoria.

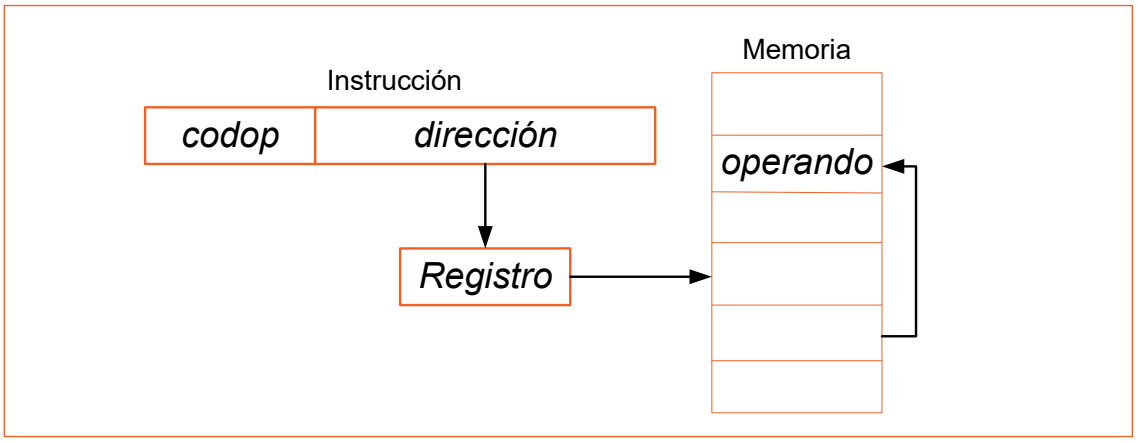

Figura 14. Modo de direccionamiento indirecto por registro.

- **Direccionamiento indexado**

  Es un poco más elaborado que el modo indirecto debido al requerimiento de una operación aritmética para su funcionamiento. Un ejemplo de este modo ocurre cuando se desea asignar a un registro el valor del contenido de una dirección de memoria indexada que resulta de la suma de una constante más el contenido de un registro.

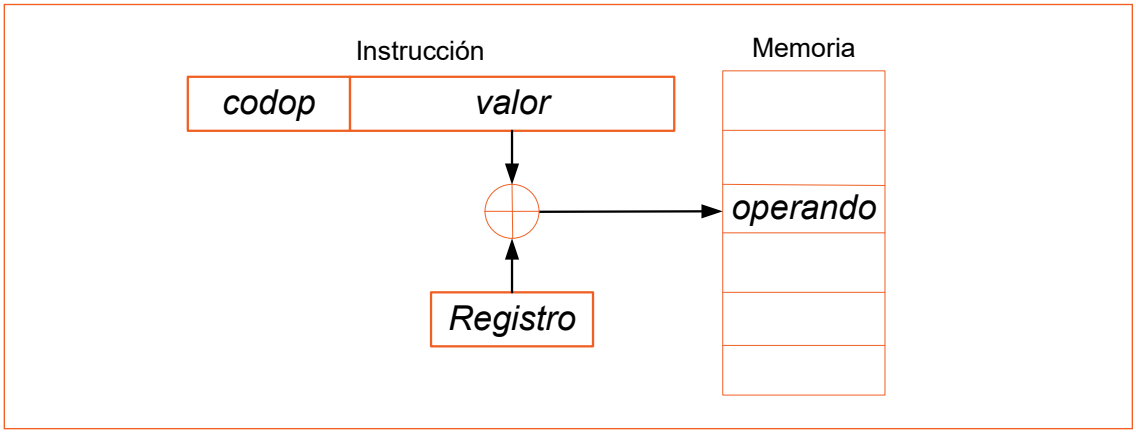

Figura 15. Modo de direccionamiento indexado.

- **Direccionamiento relativo**

  Funciona de manera similar al modo indexado, con la diferencia de que en este caso como registro de se utiliza el contador de programa.

- **Direccionamiento de autoincremento**

  Funciona de manera similar al modo de direccionamiento indirecto por registro. La diferencia con este modo radica en que el contenido del registro de indexado es incrementado al finalizarse la instrucción. Ejemplo: **ASIGNAR** el contenido de la dirección a donde apunta el RegistroIndx al Registroi; al finalizar la instrucción el valor de RegistroIndx se incrementa en 1.

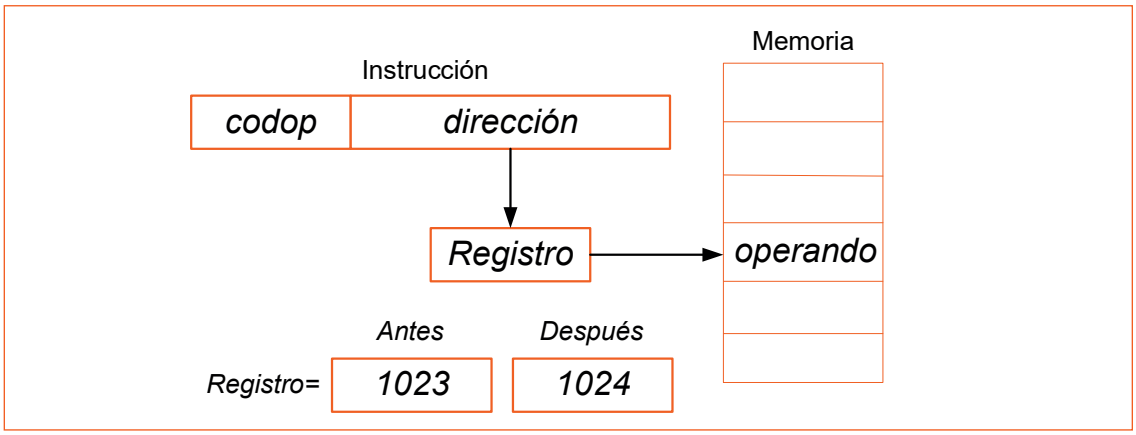

Figura16. Modo de direccionamiento de autoincremento.

- **Direccionamiento de auto decremento**

  Funciona de manera similar al modo anterior, con la diferencia de que el contenido del registro de indexado se decrementa antes de realizarse la instrucción.

- **Direccionamiento de pila**

  Teniendo en cuenta que la pila es una cola del tipo FIFO este direccionamiento permite asignar y extraer elementos de la pila. Previo a las asignaciones el computador debe incrementar el registro puntero de la pila (*stack pointer*); mientras que, luego de extraerse un elemento de la pila dicho registro debe ser decrementado.

5.1.5. Formato de las instrucciones

Las instrucciones están constituidas por un conjunto de campos que representan los elementos que las constituyen. Dichas instrucciones se representan por una cadena de bits, como se puede apreciar en el ejemplo mostrado en la Figura 17 donde se muestra una instrucción de dos operandos. En cualquier caso, los operandos siempre deben ser referenciados mediante alguno de los modos de direccionamientos disponibles. Para un mejor manejo por parte de los usuarios, en lugar de utilizarse la representación binaria de las instrucciones, es común utilizar una representación simbólica mediante abreviaturas que permiten identificar el codop y los operandos relacionados.

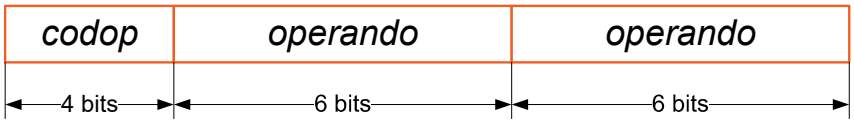

Figura 17. Formato de una instrucción máquina de 16 bits que emplea dos operandos.

El formato de las instrucciones permite definir aspectos como su tamaño y la cantidad de bits que se destinan para describir cada uno de los campos que la constituyen. A la hora del diseño de dicho formato, se deben considerar diversos factores que implican un compromiso entre la versatilidad que ofrece el repertorio de instrucciones y la complejidad requerida en el computador para su implementación. Los diferentes formatos pueden diferenciarse uno de otros según el número de bits; el tipo, cantidad y ubicación de los operandos; así como los tipos de instrucciones y de datos. Las múltiples maneras de definir el formato de las instrucciones proporcionan una amplia variedad de arquitecturas.

La arquitectura del repertorio de instrucciones puede clasificarse según diversos factores como: la cantidad de espacio de memoria que requieren los programas, las complejidades de la decodificación y de las tareas necesarias para ejecutar una instrucción; así como, el tamaño y cantidad de instrucciones. La característica más elemental a tener en cuenta a la hora de definir el formato de las instrucciones es su tamaño. Los formatos de mayor tamaño permiten definir un mayor número de instrucciones, operandos, modos de direccionamiento y rango de direcciones; facilitando de esa forma el desarrollo de programas más sofisticados que requieren una menor cantidad de códigos para realizar las mismas tareas. No obstante, el incremento de la longitud de las instrucciones no debe

aumentarse indiscriminadamente debido a que la utilidad que ofrecen no necesariamente crece proporcionalmente con su tamaño.

Otro aspecto a tener en cuenta a la hora de especificar el tamaño de las instrucciones es, que este sea igual al tamaño de la transferencia de memoria o un múltiplo entero de esta. Dicha particularidad garantiza que se tenga un número entero de instrucciones durante un ciclo de decodificación. Una implicación adicional del tamaño de las instrucciones es su tiempo de ejecución. El crecimiento de la velocidad de transferencia de las memorias no ha tenido el mismo ritmo de crecimiento que la velocidad de los procesadores. Esta limitación no solo es aplicable para la memoria principal, también se presenta en la memoria caché. De aquí se deriva que la velocidad de los procesadores se incrementa a medida que disminuye la longitud de las instrucciones. Esta disminución es indispensable en los procesadores modernos que pueden realizar múltiples instrucciones en un mismo ciclo de reloj.

Cabe destacar que, si bien las instrucciones de menor tamaño tardan menor tiempo en ser decodificadas, no necesariamente estas se ejecutarán más rápido en la misma proporción que decrece su longitud. Por último, es recomendable que la longitud de la instrucción sea igual a un múltiplo entero de la longitud de un carácter; debido a que ellos son una forma muy común de datos. De esta forma, se garantiza un uso más eficiente de todos los bits que conforman la instrucción.

Otro criterio de diseño a tener en cuenta en el formato de las instrucciones es el de garantizar una adecuada asignación de bits para los distintos campos que la constituyen. A la hora de definir la cantidad de bits asignados al direccionamiento deben tenerse en cuenta diversos aspectos, en Stallings (2013) se enuncian algunos de los aspectos más relevantes a tenerse en cuenta:

- **Modos de direccionamiento:** a pesar de que es posible que los codops permiten hacer referencia a un determinado modo de direccionamiento, en otros casos se requiere uno o varios bits para identificar el modo.

- **Número de operandos:** si bien es posible que solo uno de los operandos requiera un indicador de modo, debe considerarse que en ocasiones se requiere un indicador de modo para cada operando.

- **Número de registros:** en la mayoría de los computadores es necesario especificar más de un registro para realizar las operaciones, en los computadores actuales suelen tenerse no menos de 32.

- **Conjunto de registros:** en la actualidad los computadores tienden a tener bancos de registros de usos específicos, posibilitando que se requiera una menor cantidad de bits para su referencia ya que los codops permiten especificar el conjunto de registros que se usa.

- **Rango de direccionamiento:** debido al gran número de direcciones de memorias disponibles, la cantidad de bits requeridos para su direccionamiento constituye un factor crítico; usualmente este número es considerablemente grande.

- **Granularidad de las direcciones:** se refiere a la posibilidad de direccionar palabras o bytes en la referencia a direcciones de memorias. La primera de estas reduce la cantidad de bits para el direccionamiento; mientras que, la segunda facilita la manipulación de caracteres.

Para instrucciones del mismo tamaño existe un compromiso contrapuesto entre el número de codop y el rango de direccionamiento. Es evidente que si se asigna un mayor número de bits a los codop se dispondrá de una mayor diversidad de operaciones, pero el precio que se pagaría sería la peor resolución de la memoria. Una solución que ofrece al diseñador la facilidad de modificar acorde a sus necesidades el número de codops para un tamaño fijo de la instrucción, es la utilización de codops de longitud variable o codops expandidos. En esta técnica el número de bits que se utilizan para los codops no es fijo, posibilitando incrementar la cantidad de instrucciones disponibles.

En algunos computadores todas las instrucciones tienen el mismo tamaño, mientras que en otros pueden existir instrucciones de diferentes longitudes. La utilización del mismo tamaño en todas las instrucciones facilita su decodificación; y, al mismo tiempo implica un desperdicio de espacio ya que independientemente de su complejidad todas deben tener una longitud constante. El diseño de instrucciones de longitud variable ofrece una mayor flexibilidad en cuanto al número de codops y configuraciones de los bits de direccionamiento. Una importante estrategia a tener en cuenta en esta técnica es la de garantizar que la instrucción de mayor longitud sea un múltiplo entero de bytes o de palabras, facilitando de esta forma su decodificación.

6. Segmentación y paralelismo en el diseño de computadores

La segmentación constituye una de las principales estrategias que permite un incremento sustancial del número de instrucciones ejecutadas en cada ciclo. Su principio de funcionamiento está basado en la utilización del paralelismo a nivel de instrucción, lo que permite subdividir el procesamiento de las instrucciones en varias subetapas:

- Búsqueda de la instrucción
- Decodificación
- Búsqueda de operandos
- Ejecución
- Escritura de resultados

Como consecuencia, la ejecución de una instrucción se dividirá en pequeñas secciones, estas múltiples secciones independientes pueden ser procesadas por separado en diferentes etapas permitiendo de esa forma incrementar el desempeño proporcionalmente al número de segmentos que se tengan.

6.1. Principios de segmentación

En el caso ideal, cuando un cómputo se divide en n segmentos, el incremento del desempeño obtenido será n veces mayor. Pudiera pensarse entonces que sería conveniente aumentar la

división de la instrucción en un número elevado de segmentos para mejorar proporcionalmente el rendimiento del computador. Desafortunadamente este comportamiento solo es válido en un caso ideal, dado que en la práctica existen varios factores que impactan en la elección del número de segmentos, haciendo impráctico un aumento de esta partición por encima de un valor dado.

La principal restricción que limita el número de etapas segmentadas es la restricción de temporización. Dicha restricción está relacionada con los tiempos máximos (T_{max}) y mínimos (T_{min}) para la demora de propagación; así como, el tiempo requerido para el almacenamiento de la información (T_{lat}) en los circuitos digitales que conforman cada una de las etapas segmentadas. Aún en el caso de que el diseño de la lógica combinacional en cada una de las etapas se realice apropiadamente, de manera que la longitud de los caminos de conducción críticos sea igual posibilitando que $T_{max}=T_{min}$, el número máximo de etapas para la segmentación está limitado por T_{lat} y por el margen admisible para la desviación de la señal de reloj. Teniendo en cuenta las restricciones anteriores, dada una lógica combinatoria, se dispondrá de un límite máximo para el número **n** de etapas en que se podrá segmentar la ejecución de las instrucciones. No obstante, es importante destacar que este valor máximo no siempre puede ser el óptimo, dependiendo de la manera en que se determinan los requisitos de optimalidad.

La determinación del número de etapas óptimas para la segmentación debe considerar tanto el coste como el desempeño. Para ello se considera el coste *G* y la latencia *T* del diseño de una arquitectura no segmentada y se procede con la planificación de dicho diseño de manera segmentada para lo cual se requiere un número determinado de *latches* cuyos costes y latencia asociados se determinan a partir de *L* y *S*. De esta forma podemos obtener que el coste para la implementación de una arquitectura segmentada constituida por *k* etapas es igual al coste de la arquitectura no segmentada más el coste de los *k latches* requeridos en cada una de las etapas($G+kL$).

Desde el punto de vista de la latencia tenemos que la arquitectura no segmentada posee un valor *T* para este parámetro. En el caso segmentado tendremos una disminución significativa de la latencia igual a $T/k+S$. Dicha disminución se debe a que cada una de las *k* etapas requerirá una latencia *k* veces menor que la arquitectura sin segmentar y a la latencia asociada con las demoras de propagación en los *latches* (*S*). Como resultado se tiene, que cuando se divide una arquitectura no segmentada en k etapas se requiere de un aumento del coste igual a $G+kL$ y al mismo tiempo se obtiene una reducción de la latencia de $T/k+S$.

Definiendo el desempeño como el inverso de la latencia, es posible determinar la cantidad de etapas requeridas en un diseño que sea óptimo simultáneamente en términos del coste y del desempeño. Como resultado tendremos que el coste por desempeño será:

$$GT/k + TL + GS + SLk$$

Para determinar el número óptimo de etapas que minimiza el coste por desempeño; o sea, encontrar el valor de *k* que garantice el mayor desempeño al menor coste. Hallando la primera derivada e igualando a cero tendremos que el valor óptimo es:

$$k_{opt} = \sqrt{(GT/SL)}$$

De aquí podemos percatarnos que el número óptimo de etapas en una arquitectura segmentada, para un circuito combinacional dado, dependerá del coste *G* y la latencia *T* de su equivalente sin segmentar y del coste *L* y la latencia *S* de los *latches* requeridos.

6.2. Principios del diseño de computadores segmentados

En el caso ideal, la utilización de la segmentación para dividir un cómputo en *k* etapas proporcionará un aumento del rendimiento de *k* veces. En la práctica este incremento del rendimiento en un factor *k* no es posible lograr debido a diversas razones que se expondrán a continuación. En un escenario ideal la segmentación asume que los cómputos realizados en las diferentes etapas requieren iguales lapsos de tiempo para la ejecución (cómputos no uniformes), aspecto que no siempre es posible lograr en la realidad. Como resultado de esta duración no uniforme en las diferentes etapas, se tendrá una pérdida del rendimiento debido a que el ciclo de la segmentación estará determinado por la etapa que tenga la mayor duración. Cuando se tiene un diseño de una arquitectura segmentada donde las etapas tienen diferentes tiempos de ejecución se incurre en ciertas pérdidas de tiempo debido a los cómputos no uniformes; esta pérdida es conocida como **fragmentación interna**, constituyendo una de las razones por las cuales el rendimiento no puede incrementarse proporcionalmente al número de segmentos que se tenga.

Otra razón por la cual no se puede alcanzar el valor teórico en el incremento del rendimiento (igual al número *k* de segmentos) es debido a que no siempre se realiza el mismo tipo de instrucciones (cómputos idénticos). En el caso ideal se consideran todas las instrucciones idénticas, pero esta suposición no es válida en la realidad. Para ilustrar el efecto de esta limitante consideremos la ejecución de tres tipos de instrucciones diferentes: una instrucción para la suma, una de almacenamiento y una de bifurcación o salto. Estas instrucciones normalmente tienen la particularidad de que cada una de ellas requiere diferentes números de etapas segmentadas para su ejecución, pudiéndose tener el caso de que alguna de ellas utilice todas las etapas mientras que las otras solo requieran algunas de todas las etapas disponibles. Como resultado, determinadas etapas pueden ser subutilizadas en la ejecución de ciertas instrucciones incurriendo en un uso ineficiente de la técnica de segmentación, a este fenómeno se le conoce como **fragmentación externa**.

La última razón por la cual no se puede obtener un incremento en el rendimiento igual al número de etapas segmentadas se debe a que los cómputos en las instrucciones no siempre son independientes una de otras (cómputos dependientes). En caso de que las instrucciones fuesen completamente independientes la ejecución de cada etapa se realizaría de manera ininterrumpida alcanzándose el incremento deseado del rendimiento. Este caso no se cumple en la realidad, ya que las instrucciones que se ejecutan unas a continuación de las otras dependerán de los resultados de las instrucciones anteriores. Como consecuencia de esta dependencia, en algunas ocasiones ciertas etapas no podrán ejecutarse simultáneamente; ocasionando que no se alcance la mejoría del rendimiento igual al número de etapas (caso ideal).

La selección del ISA a implementar tiene un impacto significativo en el desempeño global que se puede obtener mediante la utilización de la segmentación. Cuando se emplea un número grande de modos de direccionamiento, se tendrá una penalización significativa en el desempeño del computador cuando en este se utiliza la segmentación. La razón de esta penalidad se debe a la

diferencia de latencia existente entre el acceso a memoria en comparación con el acceso a los registros. Esta diferencia es típica al emplear el modo de direccionamiento de memoria en las operaciones de la ALU, en el caso que uno de los operandos se encuentra en la memoria y el otro en un registro. Como resultado se tendrá un incremento significativo del tiempo que se requiere para leer ambos operandos y por ende la ejecución de este tipo de instrucciones será mucho mayor en comparación con aquellas cuyos operandos se encuentran en registros.

Debido a que el acceso a memoria es una operación indispensable en todo computador, no es posible pensar en eliminar su uso como una posible solución a la penalización del rendimiento en las arquitecturas segmentadas. Una manera efectiva de minimizar el efecto adverso del acceso a memoria se obtiene limitando su uso únicamente para instrucciones de lectura y escritura que permitan manipular los datos almacenados en esta. De esa forma se garantiza que todas las operaciones en la ALU se realicen con datos almacenados en los registros, minimizando el impacto de los cómputos no uniformes en las diferentes etapas. De igual modo la utilización de memorias caché, como parte de la jerarquía de la memoria, tendría un impacto positivo en la reducción de los tiempos de acceso y por ende en el rendimiento del computador. Por lo antes mencionado, **a la hora de diseñar una ISA en la cual se implementaría la técnica de segmentación es necesario limitar el número de modos de acceso a memoria y considerar múltiples niveles de memoria caché para garantizar un mejor desempeño**.

Otro parámetro relacionado con el impacto de las especificaciones de la ISA en el desempeño de la segmentación es la **uniformidad de los cómputos**. Para garantizar una mayor uniformidad es deseable que la ISA esté conformada por instrucciones que no sean excesivamente complejas; o dicho de otra forma, es necesario disminuir la diversidad entre los diferentes tipos de instrucciones. Como resultado de esta mayor homogeneidad se requerirá de tiempos de ejecución similares para todas ellas; y por consiguiente se tendrán cómputos idénticos.

La arquitectura RISC combina conjuntamente el uso de instrucciones simples, la ejecución de las operaciones de la ALU cuyos operandos se encuentran en registros, y el uso únicamente de instrucciones de lectura y escritura para el acceso a memoria. En esta arquitectura el hecho de que todas las operaciones de la ALU involucren a los registros, posibilita que todas estas operaciones requieran de tiempos de ejecución similares incrementando la uniformidad en los cómputos y por consiguiente el desempeño alcanzado mediante la segmentación.

Finalmente analicemos el efecto que tiene la ejecución de cómputos independientes en el desempeño de una arquitectura segmentada. Cuando se requiere de modos de direccionamiento de memoria para acceder a los operandos requeridos en las instrucciones, la realización del chequeo de dependencia entre dichos operandos se torna más compleja; inclusive pudiéndose ejecutar dicho chequeo no es posible realizar la siguiente instrucción. De esta forma es necesario esperar a que se completen las instrucciones actuales para poder ejecutar las siguientes, lo que degrada significativamente el desempeño de la arquitectura segmentada. Como resultado, se hace necesario explotar la independencia de los cómputos minimizando el uso del direccionamiento de memoria en la ISA para incrementar el desempeño alcanzado mediante la segmentación.

Una alternativa para solventar la limitación anterior se logra mediante el uso del direccionamiento de registros. Cuando se utiliza este modo de direccionamiento para los operandos de las operaciones en

la ALU, se hace más fácil el chequeo de dependencia entre las salidas de las instrucciones previas y las entradas a las subsiguientes debido a que solo se requiere chequear el valor de los operandos en los registros. Esta particularidad es otra de las razones por las cuales todas las operaciones en la ALU que se realizan en la arquitectura RISC utilizan los modos de direccionamiento de registros.

6.3. Segmentación de instrucciones en arquitectura RISC

Para una mejor comprensión del funcionamiento de la segmentación analicemos el comportamiento de esta técnica para una arquitectura segmentada simple constituida por cinco etapas. La ejecución de las instrucciones involucra diferentes etapas, siendo la primera de estas la **búsqueda** (**F**, *Fetch*). En esta etapa se hace uso de un mecanismo de caché dividida para el almacenamiento de las instrucciones y los datos por separado. Dicho mecanismo previene la ocurrencia de interferencias en el acceso a las instrucciones y a los datos; de esta forma las instrucciones son almacenadas en la caché de instrucciones. Siempre que se requiera ejecutar alguna instrucción el **contador de programa** (**PC**, por sus siglas en inglés) especifica la dirección base donde está almacenada para proceder con la búsqueda. Luego de encontrada la instrucción, la secuencia binara que la representa se almacena en el **registro de instrucciones** (**IR**, por sus siglas en inglés).

Posteriormente a la etapa de búsqueda se realiza la **decodificación de las instrucciones** (**D**, *Decode*), la que permite determinar cuál es la operación particular que ejecutarán las instrucciones. Para ello el contenido del IR se asigna al decodificador de instrucciones y este emplea diferentes mecanismos para realizar la decodificación de la instrucción. Típicamente las instrucciones están constituidas por diversos campos, siendo el primero de ellos el codop que especifica la operación que realiza, mientras que el resto de los campos especifican los operandos. En esta etapa, adicionalmente se efectúa la decodificación de los operandos, conociéndose así el modo de direccionamiento requerido. En caso de que los operandos se encuentren almacenados en registros, se determina directamente su dirección; mientras que, si algunos de los operandos se encuentran en la memoria principal se requiere ejecutar una operación adicional denominada etapa de ejecución. Cuando se tiene una instrucción de bifurcación o salto en la etapa de decodificación también se efectuará un chequeo para la condición del salto. Como resultado, en el ciclo de decodificación las instrucciones de bifurcación son ejecutadas completamente, calculándose en esta la dirección destino del salto.

Una vez determinadas las direcciones de los registros donde se encuentran almacenados los valores de los operandos y leído el contenido de dichos registros, se realiza la etapa de **ejecución** (**E**, *Excecute*) donde se lleva a cabo la operación correspondiente. En la arquitectura RISC analizada se realizaría la operación en la ALU con los valores correspondientes contenidos en los registros implicados. En las instrucciones que utilizan direccionamiento de memoria, como por ejemplo la lectura y escritura, la etapa de decodificación especifica la operación en memoria que se ejecutará, siendo necesario determinar la dirección efectiva mediante la etapa de ejecución. En esta etapa, tanto para las instrucciones de lectura y escritura, la dirección efectiva se obtiene como resultado de la suma de la dirección base y el *offset*. Posteriormente se ejecuta la etapa de **acceso a memoria** (**MA**, *Memory Access*) donde se encuentra el valor de los operandos implicados en la operación. En el caso de una instrucción de lectura se lee el contenido de la ubicación especificada por la dirección efectiva obtenida; por otra parte, cuando se tiene una instrucción de escritura se modifica el contenido de dicha dirección de memoria.

Cuando se emplea direccionamiento de registro, la operación de escritura de los resultados en el registro destino, tanto para las operaciones en la ALU como la de lectura, solo se realiza en la etapa conocida como **almacenamiento** (**WB**, *Write-Back*). Se había mencionado previamente que las operaciones en la ALU se ejecutaban durante la etapa de ejecución y posteriormente los resultados son almacenados en la etapa de almacenamiento.

Tengamos en cuenta que, para la arquitectura RISC segmentada analizada, constituida por cinco etapas, las instrucciones de bifurcación requerirán de dos ciclos, las instrucciones de almacenamiento cuatro ciclos y el resto cinco ciclos de reloj. Habiendo especificado los anteriores detalles, analicemos a continuación el efecto del uso de una arquitectura segmentada en el desempeño del computador. En la Tabla 2 se muestra el proceso de ejecución simultánea de 5 instrucciones mediante la arquitectura propuesta.

Tabla 2
Tiempos de ejecución de 5 instrucciones en una arquitectura segmentada.

Número de instrucción	Ciclos de reloj								
	1	2	3	4	5	6	7	8	9
i	F	D	E	MA	WB				
i+1		F	D	E	MA	WB			
i+2			F	D	E	MA	WB		
i+3				F	D	E	MA	WB	
i+4					F	D	E	MA	WB

Cuando se ejecuta la instrucción *i*, durante el primer ciclo de reloj se realiza la etapa de **búsqueda** (**F**). Posteriormente en el segundo ciclo de reloj, para esta misma instrucción, se efectuará la etapa de **decodificación** (**D**) liberándose la etapa **inicial F**. Debido a que cada una de las etapas que componen esta arquitectura realiza cómputos independientes unas respecto a las otras, es posible realizar simultáneamente diferentes etapas de múltiples instrucciones. De esta forma en el caso ideal (todas las instrucciones son independientes, cada etapa requiere de idénticos tiempos de ejecución, las instrucciones siempre pasan por todas las etapas y todas las etapas pueden manejarse en paralelo) después de los primeros cinco ciclos de reloj se completaría la primera instrucción y a partir de ahí en cada ciclo posterior se completaría una nueva instrucción. Como resultado se obtendría un desempeño de una instrucción por ciclo de reloj, cinco veces mayor comparado con una arquitectura equivalente no segmentada.

6.4. Atascos de un cauce

Si en alguna etapa se tiene una situación que impida que se realice la ejecución de la siguiente instrucción en el ciclo correspondiente se produce un atasco, situación que limita el incremento del desempeño que se puede obtener mediante la segmentación. Estas situaciones adversas pueden

deberse tanto a la organización de la segmentación como a otros factores de dependencia, y se clasifican en tres categorías:

- La primera corresponde a los **atascos estructurales originados por indisponibilidad o conflictos de los recursos**. Este tipo de limitación se tiene cuando dos o más instrucciones necesitan utilizar un mismo recurso de *hardware* en el mismo ciclo.

- La segunda categoría se relaciona con los **atascos por la dependencia de los datos requeridos para la ejecución de las instrucciones en los distintos segmentos**. Esta condición se tiene cuando los operandos fuente o destino de una instrucción no están disponibles en el momento en que se necesitan en una etapa determinada. La dependencia de datos pude clasificarse según el orden en que se realizan las operaciones con estos: lectura después de escritura, escritura después de escritura y escritura después de lectura.

- Finalmente se tienen los **atascos asociados a la dependencia de control existente en la ejecución de instrucciones que dependen de la forma en que se ejecuten otras**. Las principales responsables de esta categoría de atasco son las instrucciones de bifurcación o salto. Este tipo de atasco implica una penalización mayor que la causada por la dependencia de datos; el comportamiento anterior se puede comprender mejor recordando que en la arquitectura de cinco etapas mostrada previamente, la dirección destino del salto se obtiene al final de la etapa de decodificación lo que impide la ejecución de la siguiente etapa antes de este paso.

Una técnica simple para manejar los atascos asociados a la dependencia en las instrucciones de control es el **vaciado de todos los segmentos**. En esta variante el procesador maneja las instrucciones de bifurcación como cualquier otra, garantizando que, si la dirección destino computada durante una bifurcación es diferente a la siguiente instrucción, como resultado se vacían todos los segmentos; en caso contrario se continua con la ejecución. Esta variante posee la ventaja de que no requiere una complejidad adicional en el *hardware* con el inconveniente asociado de que crea una penalidad considerable en el desempeño. Por lo antes mencionado se hace necesario implementar técnicas más elaboradas que permitan manejar esta limitación de manera efectiva; dentro de estas se tienen: flujos múltiples, pre-búsqueda del destino del salto, *buffer* de bucles, predicción de saltos, salto retardado, entre otros (Stallings, 2013).

En la técnica de flujos múltiples se hace uso de la redundancia para duplicar las partes iniciales del cauce permitiendo que se realice la ejecución de la instrucción por ambos caminos. Esta variante posee los inconvenientes de que se introducen retardos producidos por la competencia en el uso de los recursos. Otro inconveniente que se tiene es la posibilidad de que se produzcan instrucciones de saltos múltiples antes de que se resuelva la decisión del salto original.

El siguiente enfoque se basa en la pre-búsqueda de las instrucciones destino del salto; de esta forma cuando se produce el salto ya se conocen las instrucciones que deben ejecutarse. Otra variante hace uso de *buffers* para los bucles donde se almacenan las instrucciones buscadas más recientes; de manera que en caso de producirse un salto se pueda verificar si el destino se encuentra almacenado en estos *buffers*.

Otra técnica para combatir los atascos asociados a la dependencia en las instrucciones de control se basa en la **predicción de los saltos**. Algunas de las variantes más usadas son: predecir que nunca se salta, predecir que siempre se salta, predecir según el código de operación, conmutador saltar/no saltar, tabla de historial de saltos, entre otras (Stallings, 2013). Los tres primeros enfoques no dependen del historial de ejecuciones realizados hasta la ocurrencia del salto (estáticas), mientras que las dos últimas si dependen de este historial (dinámicas). Finalmente se tiene el enfoque mediante salto retardado, en el que posibilita una mejora del funcionamiento del cauce mediante el reordenamiento automático de las instrucciones del programa. Para ello en los ciclos de reloj previos a la ejecución del salto se ejecutan otras instrucciones.

6.5. Arquitectura segmentada superescalar

Hasta el momento se ha visto que una arquitectura segmentada en *k* etapas, en el mejor de los casos, la mejora en el desempeño será *k* mejor en comparación con un diseño equivalente no segmentado. En esta arquitectura segmentada en cualquier instante de tiempo solo una instrucción puede ejecutarse en una etapa específica; ello implica que el número de instrucciones que se pueden ejecutar simultáneamente será igual al número de etapas disponibles (en el caso ideal). Otra particularidad de la segmentación escalar es que las instrucciones se ejecutan a través de todas las etapas, independientemente del tipo de instrucción que se tenga (unificación de todas las instrucciones). Como resultado en algunos escenarios se incurre en ciertas penalizaciones que implican el desperdicio de algunas etapas y por ende se disminuye el rendimiento. Una limitación adicional que se tiene en este enfoque es que las instrucciones se procesan secuencialmente por todas las etapas, de modo que cuando ocurre un atasco la ejecución de las siguientes instrucciones deben esperar porque se resuelva dicho atasco. **Para solventar las limitaciones mencionadas se hace necesario considerar una organización diferente comparada con el principio escalar visto hasta el momento; a esta alternativa de diseño se le conoce como segmentación superescalar.**

La limitación en la segmentación escalar de poder ejecutar solo una instrucción por etapa por ciclo se puede solventar mediante el uso de una arquitectura paralela en la cual cada una de las etapas posee una réplica. Como resultado se pueden ejecutar simultáneamente más de una instrucción por ciclo. La otra limitación de la segmentación escalar (unificación de todas las instrucciones) puede superarse diversificando el diseño de la arquitectura, pudiendo emplearse el enfoque paralelo donde cada rama conste con un número diferente de etapas en dependencia de los diferentes tipos de instrucciones. Resultado que este enfoque diversificado de la segmentación permite minimizar los atascos innecesarios asociados con la dependencia entre las instrucciones y por consiguiente se tendrá una mejoría en el desempeño.

Para solventar la última limitación se emplea el uso de segmentación dinámica; en esta variante en lugar de utilizar un único *buffer* entre dos etapas adyacentes que permita almacenar una sola instrucción se consideran un *buffer* con múltiples entradas. La posibilidad de considerar las diferentes entradas de este tipo de *buffer* como un componente independiente permite mover las instrucciones de una manera más eficiente pudiendo incrementar el desempeño. Este incremento puede lograrse haciendo uso de la independencia a nivel de instrucciones (paralelismo a nivel de instrucciones) mediante la ejecución no ordenada, en la cual las múltiples entradas del *buffer* son reordenadas pudiéndose seleccionar dinámicamente las instrucciones independientes.

Con el objetivo de aprovechar efectivamente el paralelismo a nivel de instrucciones es necesario dimensionar el número de entradas del *buffer* por encima del número de ramas paralelas en la segmentación superescalar. Este diseño prevé los atascos en los escenarios en que no se tienen un número de instrucciones independientes igual a la cantidad de ramas paralelas, proporcionándole al *buffer* una capacidad de procesamiento suficiente. Como resultado de la combinación de las técnicas antes mencionadas en el diseño de un procesador (procesamiento simultáneo de varias instrucciones, segmentación diversificada y *buffers* multi-entrada) se tiene una arquitectura segmentada superescalar, la cual posibilita un incremento significativo en el desempeño del computador. Un procesador superescalar facilita el establecimiento de múltiples cauces de instrucciones independientes; dichos cauces están constituidos por diferentes etapas diversificadas, permitiendo la ejecución de varias instrucciones de manera simultánea.

La dependencia entre las instrucciones limita la ejecución de estas de una manera no ordenada; dicha dependencia puede existir tanto en los datos como en los nombres. En los datos se tiene una verdadera dependencia debido al flujo de datos existente, mientras que en el caso de los nombres existe una falsa dependencia dado que no se tiene ningún manejo de datos. Existen varios enfoques en la microarquitectura que posibilitan la implementación de la planificación dinámica, dentro de los que se tiene el **renombramiento de registro**; esta técnica es un mecanismo basado en *hardware* que permite la eliminación de las falsas dependencias. Los mecanismos de planificación dinámica poseen diversas ventajas respecto a los estáticos tales como: permiten que los códigos compilados en una microarquitectura puedan ejecutarse de manera eficiente en otra microarquitectura diferente, pueden manejar las dependencias no conocidas durante la compilación, el procesador puede tolerar demoras impredecibles, entre otras.

En la arquitectura de carga y almacenamiento una instrucción típicamente está constituida por el codop y los operandos fuente y destino. En el caso de las instrucciones de la ALU usualmente se emplean registros para los operandos. Cuando en este tipo de operaciones la unidad funcional no se encuentra disponible se produce un atasco estructural; si en lugar de ello son los operandos destino los que no están disponibles entonces se tendrá un atasco de datos. Por otra parte, si son los operandos fuentes los que se encuentran indisponibles (situación que se produce por el reúso de los registros) ocurrirá una falsa dependencia, condición que puede solventarse mediante el empleo del **renombramiento de registro**. Esta técnica consta de tres fases fundamentales:

- Asignación de destino
- Actualización de registros
- Carga de los operandos

En la fase de asignación de destino se identifica una entrada disponible en el **Fichero de Renombramiento de Registro** (**RRF**, por sus siglas en inglés) y se mapea el registro seleccionado en el RRF dentro del Fichero de Registro de Arquitectura (**ARF**, por sus siglas en inglés); luego de esto se concluye la etapa y se emite la instrucción a la unidad funcional para su ejecución.

En la segunda fase del **renombramiento de registro** se actualizan los resultados en la posición especificada dentro del **RRF**; en el caso de que la ejecución de la instrucción se realice de manera ordenada, el valor computado es actualizado en el **ARF**.

En la última etapa se realiza la carga de los valores en los operandos fuentes; estos datos pueden leerse directamente en el **ARF** en caso de que estén disponibles allí, o de lo contrario es necesario acceder al **RRF** para ejecutar la carga.

Otra solución empleada en la planificación dinámica que constituye un elemento esencial del diseño de los procesadores superescalares modernos, es el **algoritmo de Tomasulo**. En dicho algoritmo se propone un nuevo diseño para la unidad de punto flotante basado en los conceptos de estación de reservación, reenvío de operandos y la utilización de un bus común conectado a varios ficheros de registros mediante el cual los operandos puedan ser enviados a las instrucciones que los requieran permitiendo incrementar el desempeño en conjunto.

En los procesadores superescalares la ejecución de las instrucciones de manera no ordenada se efectúa mediante el núcleo de ejecución dinámica; este núcleo constituye una versión mejorada del enfoque de Tomasulo en el cual se realizan las tareas de emisión, ejecución y finalización de las instrucciones. En la tarea inicial se realiza la asignación para el **renombramiento del registro** en las entradas de la estación de reservación; así como, la actualización del *buffer* de reordenamiento y el avance de la instrucción desde el *buffer* de emisión a la estación de reservación. Posteriormente las instrucciones son procesadas en las unidades funcionales y luego los resultados son enviados de vuelta a la estación de reservación y al fichero de **renombramiento de registros** a través del bus común para su actualización.

El diseño del planificador dinámico de instrucciones se puede realizar mediante dos mecanismos diferentes: con captura de datos y sin captura de datos. En el primer diseño entre el fichero de registro y la unidad funcional se encuentra la estación de reservación; en esta última se realiza la captura de los datos empleando una ventana. Cada vez que la unidad funcional termina una ejecución, los valores computados se envían tanto a la estación de reservación como al fichero de registro. En el caso de los valores enviados al fichero de registro, se almacenan directamente en la entrada correspondiente tanto para el RRF como para el ARF. En este diseño el tamaño de la estación de reservación es muy grande debido a la necesidad de almacenamiento del contenido de los registros, así como el tamaño de la ruta que interconecta la unidad funcional y dicha estación.

En el enfoque sin captura de datos se realiza un reordenamiento del fichero de registro y la unidad funcional, en este caso el fichero de registro se encuentra entre la estación de reservación y la unidad funcional. La ventana de planificación de instrucciones sin captura de datos presente en la estación de reservación localiza todas las instrucciones que pueden ser enviadas o emitidas a las unidades funcionales. Este mecanismo presenta la limitación de que la lectura del fichero de registros se realiza antes de que se envíen o emitan las instrucciones a la unidad funcional. Esta peculiaridad implica la necesidad de realizar un diseño multipuerto para dicho fichero, incrementando de este modo el tiempo de ejecución.

6.6. Procesamiento multihilos

La técnica de segmentación de las instrucciones y la arquitectura superescalar permiten incrementar notablemente el desempeño del computador en comparación con el enfoque no segmentado. Otros avances en la microarquitectura del procesador, como la ejecución de las instrucciones de manera no ordenada, han permitido obtener mejoras adicionales en su desempeño. A medida que se aumenta el nivel de segmentación y el número de cauces paralelos, se incrementa considerablemente la cantidad de instrucciones independientes que pueden ejecutarse simultáneamente en varias unidades funcionales por cada ciclo de reloj. Una manera intuitiva de seguir incrementando el desempeño del computador se podría alcanzar haciendo un uso más intensivo de estas técnicas. No obstante, en la práctica este incremento no se puede lograr sin requerir una complejidad significativa del sistema; razón por la cual existe un límite para la implementación del paralelismo a nivel de instrucciones. Es por ello que el desempeño de los procesadores superescalares cada vez más complejos no es significativamente mayor en comparación con diseños más sencillos. Este comportamiento asintótico del rendimiento se debe a diversos factores como por ejemplo el hecho de que los programas pueden tener un paralelismo a nivel de instrucciones poco considerable. Otra limitación que influye en el rendimiento de los procesadores superescalares es que estos solo pueden realizar una sola tarea; por consiguiente, si esa única tarea no posee suficiente paralelismo a nivel de instrucciones no se tendrá un incremento significativo del rendimiento.

Las restricciones inherentes a los procesadores superescalares hicieron necesaria la búsqueda de otros mecanismos que permitiesen continuar incrementando el desempeño de los computadores. Una manera de solventar esta limitante se basa en el uso de las unidades funcionales ociosas que se tienen como resultado de la falta del paralelismo a nivel de instrucciones presente en los programas. Para ello se hace necesaria la posibilidad de que diferentes procesos puedan tener acceso a estas unidades ociosas. Como resultado, **la ejecución de una instrucción se puede solapar con múltiples procesos, haciendo un uso más eficiente de las unidades funcionales y por consiguiente posibilitando un incremento del desempeño del sistema**. A este método se le conoce como procesamiento multihilos, el cual dio origen al establecimiento de un nuevo concepto en la arquitectura que permite explotar el paralelismo a nivel de procesos.

Los **procesadores multihilos** poseen diversos flujos de control de ejecución (hilos) que les permite la ejecución de programas que han sido divididos en múltiples hilos, bien sea por el compilador o por los programadores. Cada hilo posee un contexto que puede ser identificado tanto por el contador de programa como por los registros de la arquitectura. Estos contextos permiten almacenar toda la información propia de la ejecución de cada hilo; de esta forma se puede intercambiar el orden de ejecución de los hilos permitiendo restaurar el contexto existente antes de cambio para continuar la ejecución del hilo suspendido.

Para poder procesar la ejecución de múltiples hilos el *hardware* debe contar con ciertos recursos que no se compartan entre los diferentes hilos; dicho de otra forma, es necesario replicar cada uno de estos recursos por cada hilo independiente que soporte el procesador. **Estos recursos de uso exclusivo para cada hilo son: contador de programa, conjunto de registros de la arquitectura y lógica para el renombramiento de registros.** El resto de los recursos del procesador (*buffer* de ordenamiento; *buffers* de carga y almacenamiento; colas de búsqueda, decodificación, envío y emisión; los diferentes

niveles de la memoria caché; unidad de ejecución; entre otros) son compartidos de manera eficiente por cada uno de los hilos en los que es dividido el programa. Los recursos compartidos pueden dividirse de manera aleatoria o dinámica por cada uno de los hilos que soporta el procesador.

Los procesadores **multihilos** solo pueden ejecutar simultáneamente tantos hilos como su arquitectura soporte; dependiendo del diseño del procesador es posible que para un instante de tiempo dado solamente haya un solo hilo activo o todos se encuentren activos simultáneamente. Tanto la política de planificación de los hilos como la forma en que se utilice la segmentación determinarán el tipo de diseño en los procesadores **multihilos**. Considerando las características anteriores se tienen tres tipos de diseños de procesadores **multihilos**:

- **De granos gruesos** (**CGMT**, por sus siglas en inglés)
- **De granos finos** (**FGMT**, por sus siglas en inglés)
- **Multihilos simultáneos** (**SMT**, por sus siglas en inglés)

En el primer diseño (CGMT) las instrucciones de un mismo hilo siempre se ejecutan de manera continua hasta que se produzca un evento de larga latencia, como por ejemplo cuando se produce un fallo en la caché. Para evitar el desperdicio de tiempo que se tiene en esta situación se procede con la ejecución de otro hilo. De esta forma la política de planificación de los hilos posibilita el cambio en la ejecución de los hilos únicamente cuando ocurre un evento de gran latencia. Otra particularidad de la variante CGMT es que las distintas etapas de la segmentación no se comparten entre diferentes hilos; ello implica que siempre que se realice una conmutación entre hilos es necesario borrar todas las instrucciones relacionadas con el hilo que se está ejecutando en ese momento. Como resultado se incurre en un desperdicio innecesario de etapas en la segmentación, ya que se descartarán todas antes de proceder a la realización del nuevo hilo. Para minimizar el impacto negativo de este desperdicio la conmutación siempre se realiza en caso de que ocurran latencias grandes. La modalidad CGMT es de fácil diseño e implementación y permite incrementar el desempeño en términos del rendimiento siendo apropiada para el procesamiento ordenado.

En la segunda técnica (FGMT) el cambio de ejecución de los hilos se realiza en cada ciclo de manera equitativa, de esta manera los diferentes hilos se van rotando y pudiéndose tolerar de cierta manera las latencias presentes en las diferentes instrucciones. **Debido al intercambio continuo en la ejecución de los hilos en cada ciclo de reloj, se hace necesario considerar la asignación dinámica de la segmentación entre los múltiples hilos.** Como resultado en una etapa se puede estar ejecutando una instrucción de un hilo, mientras que en la siguiente es posible que se esté ejecutando una de un hilo diferente; de esta forma se evita el desperdicio en que se incurre en CGMT. El enfoque FGMT posee la ventaja de requerir un diseño conceptualmente simple que proporciona altos desempeños siempre que se tenga un gran número de hilos; por otra parte, en el caso de que se tenga un único hilo el desempeño es muy deficiente, constituyendo la principal limitación de este método.

La última variante (SMT) considera simultáneamente el procesamiento de múltiples hilos, mediante la técnica FGMT y el mecanismo de procesamiento superescalar. A diferencia de CGMT y FGMT, donde no se pueden emitir instrucciones de diferentes hilos al procesador, en SMT se hace uso de la arquitectura superescalar permitiendo de esta manera la ejecución no ordenada de las instrucciones.

Como resultado este enfoque permite explotar el paralelismo a nivel de procesos en toda su extensión. Debido a que en SMT se considera el concepto multihilos de granos finos, en esta se emplea la política de planificación equitativa, pudiéndose cambiar la ejecución de los hilos en cada ciclo de reloj y compartiéndose las diferentes etapas de la segmentación; lo que se conoce como **mecanismo de partición dinámica**.

En la técnica SMT se pueden tolerar mayores latencias tanto de las diferentes etapas de la segmentación como de las de caché. Para poder obtener un desempeño adecuado mediante este enfoque es necesario disponer de más de 2 hilos ejecutándose simultáneamente. La compartición de los recursos entre los diferentes hilos puede realizarte tanto de manera estática como de manera dinámica; en este último modo se puede compartir diferentes proporciones de recursos en dependencia de los requerimientos de las aplicaciones que los utilizan. **La elección del modo en que se comparten los recursos (dinámico o estático) dependerá de la importancia que tenga la equidad o el desempeño.**

La ejecución de múltiples hilos simultáneamente y la utilización de diferentes recursos compartidos posibilitan la ocurrencia de conflictos que degradarían el desempeño del sistema. Algunos de estos conflictos pueden generarse en los accesos simultáneos a los TLBs y a la caché. Es por ello que en el diseño SMT se hace imprescindible implementar mecanismos eficientes para compartir los recursos.

6.7. Procesadores multinúcleos

Los crecientes requerimientos por parte de los usuarios para el procesamiento intensivo en aplicaciones como, por ejemplo, el procesamiento de imágenes, audio y datos han requerido computadores cada vez con mayores desempeños que satisfagan dichas demandas. Las soluciones analizadas hasta el momento se basan en un único núcleo para la ejecución de las instrucciones, siendo necesario incrementar la frecuencia del reloj para poder alcanzar mejores desempeños. Este incremento no puede realizarse sin requerir para ello un aumento considerable en el consumo de energía y por consiguiente mayores requerimientos para la disipación del calor. Esta limitación imposibilitó continuar obteniendo mejorías significativas en el desempeño de los procesadores de un solo núcleo basándose únicamente en las técnicas mencionadas hasta el momento, obligando a los diseñadores a plantearse un nuevo paradigma fundamentado en el uso de múltiples núcleos.

La implementación de múltiples núcleos dentro de un mismo procesador permitió reducir la frecuencia del reloj y por consiguiente el consumo energético por cada núcleo en comparación con un procesador equivalente de un solo núcleo. Aunque en conjunto ambas arquitecturas poseen un consumo energético similar, los sistemas con múltiples núcleos permiten un incremento substancial del desempeño dada la posibilidad de paralelizar de manera simultánea la ejecución de los procesos en los diferentes núcleos. Es por ello que en el diseño de procesadores multinúcleos no es necesaria la implementación de complejos mecanismos propios de la arquitectura superescalar con el fin de aumentar el desempeño. Como resultado, los diseños resultantes reducen considerablemente el área ocupada por este tipo de procesadores. Esta idea es la primicia fundamental que motivó el desarrollo de una arquitectura constituida por un conjunto de núcleos más sencillos y lentos que permitiesen

paralelizar la ejecución de las aplicaciones, en lugar de desarrollar costosos y complejos procesadores de un solo núcleo.

En este nuevo paradigma los diferentes núcleos se integran dentro de un único chip, de manera que la comunicación entre cada uno de estos no requiere de elevados tiempos en comparación con los sistemas multiprocesadores, en los cuales cada procesador está físicamente separado del otro y conectado a través de enlaces de comunicaciones lentos. Como resultado en los sistemas multinúcleos se tiene una mejora substancial del desempeño. Un criterio a tener en cuenta en el diseño de esta tecnología es las características y el número de núcleos disponibles; pudiéndose considerar una misma configuración para todos o una colección de diferentes configuraciones para cada uno de los núcleos. Una vez más es necesario hacer uso de la ley de Amdahl para responder a esta interrogante. Debido a que los programas tendrán una parte del código que no se podrán ejecutar de manera paralela, la *mejora* alcanzada en el desempeño al aumentarse el número de núcleos (n) estará limitada por la fracción del código que puede ejecutarse paralelamente (α).

$$mejora = \frac{1}{(1-\alpha) + \frac{\alpha}{n}}$$

En el caso de que se tenga una aplicación que posea un 95 % del código que se realiza de forma paralela, cuando $n \to \infty$ el valor máximo que se alcanza en la *mejora* es de 20. Esta limitante impone una restricción práctica en cuanto al número de núcleos implementados y a la uniformidad en su diseño. Dependiendo de la naturaleza de las aplicaciones no resulta ventajoso utilizar una implementación homogénea de los sistemas multinúcleos. En el caso que el procesador se emplee con mayor regularidad en aplicaciones con un alto nivel de paralelismo, resulta más eficiente el diseño homogéneo de todos los núcleos. Por otra parte, en el caso del uso en aplicaciones con un pobre nivel de paralelismo resulta más eficiente un diseño heterogéneo constituido por un núcleo más complejo y potente para ejecutar la parte del código predominantemente secuencial y un conjunto de núcleos más simples que permitan explotar el paralelismo en el resto del código.

6.8. Jerarquía de la memoria caché en procesadores multinúcleos

Un aspecto relevante a tener en cuenta en el diseño de procesadores multinúcleos es la implementación de la jerarquía de la memoria caché. Para ello se suelen establecer distintas jerarquías distribuidas entre los diferentes núcleos, pudiéndose compartir alguno de los niveles. Es común tener diseños que consideren tres niveles para la memoria caché, y en algunos casos se tienen hasta cuatro niveles. Típicamente cada núcleo posee una parte de la carché para uso privado, llegando a implementarse hasta los dos primeros niveles de la caché; mientras que el resto de los niveles son de uso común. **En la arquitectura multinúcleos se deben tener en cuenta tres aspectos de vital importancia; el primero es la coherencia de la caché, el segundo la sincronización y el tercero la consistencia de la memoria.**

6.8.1. Coherencia de la caché

El problema de la coherencia de la memoria caché es originado por la posibilidad que presentan los diferentes núcleos de usar simultáneamente la parte compartida de la caché. Como resultado los

diferentes núcleos pueden ejecutar operaciones de lectura y escritura simultánea en las mismas direcciones de memoria, de esta forma en la caché de uso privado en los diferentes núcleos se tendrán réplicas de los mismos datos. El problema de coherencia se presenta cuando en la caché privada de uno de los núcleos se actualizan los datos sin que el resto de los núcleos puedan tener acceso a estos, originando de esta forma la realización de cálculos incorrectos.

Para evitar la ocurrencia de esta limitante es necesario implementar técnicas conocidas como protocolos de coherencia de caché, los cuales permiten actualizar adecuadamente los valores de los datos. La coherencia de la caché se obtiene si, y solo si, el sistema de memoria satisface las propiedades de que los núcleos individuales preserven el orden de ejecución del programa, las escrituras son visibles por todos los núcleos y que estas últimas se serialicen a través de los núcleos. Para garantizar la coherencia en las cachés se requiere tanto de la migración como de la réplica de los datos compartidos. Cuando los datos son replicados se tienen múltiples copias de la misma información en la caché privada de cada núcleo; incluso cuando una réplica de los datos es modificada en una caché específica, el resto de las copias serán actualizadas o invalidadas automáticamente. La migración permite que los bloques de datos contenidos en la caché privada de un núcleo puedan moverse hacia la caché de otro núcleo donde sean requeridos.

Existen dos tipos de protocolos para asegurar la coherencia de los datos, el primero se basa en la invalidación de los datos mientras que el segundo se basa en la actualización de la escritura. En la primera técnica cuando se actualizan los datos en una de las réplicas se envía una señal de invalidación de modo que las cachés que poseen las réplicas de copias inhabiliten dicho dato; dicho de otra forma, al núcleo que realizará la operación de escritura se le asignará un permiso exclusivo para modificar el contenido de la caché. Como resultado el ancho del bus requerido para enviar la señal de invalidación es simple ya que solo se requiere enviar la dirección del bloque que se desea inhabilitar. La principal desventaja de este enfoque consiste en que cuando alguno de los núcleos requiere acceder a los datos invalidados en la caché ocurre un fallo.

En el segundo enfoque cuando se realiza una escritura en cualquiera de las cachés privadas de los núcleos, el resto de las réplicas siempre son actualizadas con el nuevo dato. De esta forma en todas las cachés privadas de cada núcleo siempre se tendrán los datos actualizados resultantes de la ejecución de una operación de escritura en un núcleo particular. Una de las principales desventajas de este método es que cuando en el futuro ninguno de los núcleos requiere los datos actualizados, se incurre en operaciones de escrituras innecesarias lo que se traduce en un desperdicio de energía. Otra desventaja del método de actualización es que requiere un mayor ancho de banda en el bus para la actualización de los datos en las réplicas de las cachés. Como resultado se tendrá un compromiso entre el desempeño de ambos protocolos; por una parte la variante basada en la invalidación de datos minimiza el desperdicio de energía a expensas de una mayor tasa de fallos en la caché, mientras que el basado en la actualización de la escritura permite una mejor actualización de todas las cachés a expensas de un mayor consumo energético.

En los sistemas con un número razonablemente reducido de núcleos (4 u 8) es recomendable considerar un bus común que interconecte cada uno de estos núcleos. Independientemente del tipo de protocolo de coherencia de caché que se implemente en esta arquitectura, en los sistemas basados en buses es viable emplear la **técnica de fisgoneo** (*snooping* del inglés). En este enfoque siempre que se realiza una operación de lectura o escritura se envía la solicitud correspondiente

a través del bus. De esta manera los controladores de las cachés en los diferentes núcleos, que se encuentran sondeando dicho bus, pueden identificar la solicitud en cuestión y posteriormente de existir una coincidencia con los datos almacenados proceder con la acción correspondiente (ya sea invalidar las réplicas de los datos o actualizarlas).

Por otra parte, en los sistemas con un mayor número de núcleos (16, 32, 64 o más) la técnica basada en bus deja de ser adecuada, en lugar de esta es más recomendable interconectar los diferentes núcleos mediante una red. Esta variante se conoce como protocolo de coherencia de caché basado en directorio. A pesar de que el mecanismo de este protocolo sea similar al basado en bus, la forma en que se implementa es ligeramente diferente. En esta variante la caché compartida por todos los núcleos almacena la información en un directorio que permite identificar cuál de las cachés privadas posee la copia del bloque requerido en una instrucción determinada y de esta forma cuando se realice una operación de lectura o escritura se conoce a cuál de ellas enviar la señal de invalidación.

El protocolo basado en directorio puede implementarse utilizando tanto un diseño centralizado como uno distribuido. En diseño centralizado se tiene un único directorio y una única caché compartida monolítica donde confluyen todas las solicitudes en operaciones de lectura y escritura. Es por ello que este directorio debe contar con múltiples puertos, lo que requiere un mayor espacio y gastos adicionales de energía. Estas limitaciones pueden solventarse mediante los sistemas distribuidos donde la caché compartida se distribuye en diferentes secciones entre los múltiples núcleos. Estas secciones de la caché distribuida poseen los directorios donde se almacenan los bloques de información compartida correspondientes. De esta forma se garantiza que cada directorio distribuido requiera de un único puerto, permitiendo solucionar las limitaciones de la modalidad centralizada.

Dependiendo del tipo de protocolo de coherencia que se diseñe se tendrán diferentes estados asociados con cada bloque de la memoria caché privada. Los estados permiten indicar cuando un bloque es inválido, se está compartiendo, se ha modificado, es exclusivo, etc. Esta información de estado se incluye dentro de los metadatos de la caché, los que están conformados por diferentes banderas y bits de validación que permiten identificar la situación de cada bloque.

6.8.2. Sincronización de la caché

Los mecanismos de sincronización posibilitan la consistencia entre todas las estructuras de datos que pueden ser compartidas entre los diferentes núcleos. De esta forma se asegura que en cualquier etapa de ejecución de los programas solo un núcleo tenga acceso a la sección crítica. Existen diferentes formas de garantizar la sincronización como, por ejemplo: la exclusión mutua, la basada en eventos punto a punto y la basada en eventos globales.

En la sincronización por exclusión mutua cuando dos procesos se encuentran compitiendo por una sección crítica, solamente uno será autorizado para acceder a dicha sección mientras que el otro aguardará porque se libere la sección. Para ello se emplean mecanismos de bloqueo y desbloqueo implementados a nivel de *hardware* y de *software*. Los mecanismos basados en *hardware* son más simples, pero poseen la desventaja de que no son escalables debido al número limitado de líneas de bloqueo disponibles en el sistema. Esta limitante puede solventarse mediante la implementación de mecanismos basados en *software* donde se utilizan direcciones de memorias o registros donde se almacenan las variables de bloqueo.

Para evitar que en los mecanismos basados en *software* varios procesos accedan simultáneamente a la sección crítica, es necesario que las operaciones de lectura, comparación y escritura de las variables de bloqueo se realicen de manera atómica; esto requiere que estas tres operaciones se realicen mediante la ejecución de una única instrucción (**instrucción atómica**). Para poder efectuar este tipo de instrucción es necesario que la ISA y el *hardware* brinden el soporte adecuado para el funcionamiento del mecanismo de sincronización. Dicho de otra manera, no solo es necesario que la ISA posea instrucciones de lectura, modificación y escritura; también se requieren primitivas de hardware que se ejecuten de manera atómica. Un tipo de instrucción dentro de esta categoría es ***test-and-set***; mediante ella, simultáneamente se lee el contenido de una dirección de memoria hacia un registro específico del procesador y se almacena un valor diferente de cero en dicha dirección de memoria. Como resultado, cuando una dirección de memoria posee un valor distinto de cero se indica que esta ha sido bloqueada por un proceso, previniendo de ese modo el acceso de otros procesos hasta que no sea liberada.

En dependencia de la ISA implementada se tendrán otras instrucciones atómicas como, por ejemplo: intercambio, operación de búsqueda e instrucción, comparación e intercambio, etc. En el primer tipo de instrucción se intercambian simultáneamente los valores almacenados en una dirección de memoria por los de un registro. En el segundo tipo se pueden realizar de manera conjunta con la operación de búsqueda diferentes tipos de instrucciones como por ejemplo incremento, decremento, adición, etc. De esta forma al mismo tiempo que se busca la dirección de memoria se modifica su contenido y se almacena de vuelta en la misma dirección. El último tipo de instrucción atómica mencionado permite simultáneamente leer el contenido de una dirección de memoria y compararlo con el de un registro y en caso de cumplirse una condición preestablecida intercambia el valor de la memoria con el de otro registro.

6.8.3. Consistencia de la caché

En las arquitecturas multinúcleos donde pueden ejecutarse diversos hilos de una misma aplicación o de diferentes aplicaciones, es necesario garantizar que se comparta de manera apropiada la memoria caché. Para ello no solo es necesario asegurar una adecuada sincronía, además se requiere la consistencia de la memoria. La necesidad de la consistencia de la memoria parte del hecho de que tanto el *hardware* como el compilador pueden intercambiar el orden de ejecución de las instrucciones de un programa para maximizar el desempeño (**ejecución no organizada**). Como consecuencia es posible que se obtengan algunos resultados inesperados, dando lugar a inconsistencias de la memoria que comprometen la correctitud del programa ejecutado. Para garantizar la consistencia requerida, se hace imperativa la implementación de modelos que establezcan ciertas especificaciones que brinden información a los programadores sobre los posibles resultados esperados, y al mismo tiempo establezcan un límite a los diseñadores del *hardware* y del compilador a la hora de implementar las técnicas para el reordenamiento de las instrucciones.

Un modelo para garantizar la consistencia de la memoria en los sistemas multinúcleos es el secuencial.

> Se dice que un sistema es secuencialmente consistente si el resultado de la ejecución de cualquier operación es el mismo que se tiene si la operación de los diferentes núcleos se ejecutase de un modo secuencial determinado y si las operaciones individuales de cada núcleo aparecen en esta secuencia según el orden especificado por el programa. (Lamport, 1979, p.690).

Para que la ejecución sea secuencialmente consistente se requiere que todos los núcleos inserten sus instrucciones de lectura y escritura en un orden específico acorde con el orden del programa; de esta manera se podrá construir un orden global para la memoria.

Algunos aspectos adicionales a ser considerados en el modelo de consistencia secuencial son: la realización de manera atómica de la escritura y la finalización del almacenamiento. Este último requerimiento establece que después de emitir una instrucción de este tipo en una aplicación específica, el proceso que la emite esperará a que finalice su ejecución antes de proceder a la emisión de cualquier otra operación en el orden del programa. Como resultado de los requerimientos necesarios para la consistencia secuencial y de la posibilidad de no emplearse de manera eficiente las técnicas de optimización del desempeño del *hardware* (como por ejemplo cuando se empelan *buffers* de escritura), el rendimiento en conjunto de un sistema que implementa este enfoque puede verse degradado.

La implementación de estas técnicas de optimización requiere considerar modelos de consistencia relajados, de esta manera los programadores pueden contar con una mayor flexibilidad que la ofrecida por la consistencia secuencial. En el caso de que se empleen *buffers* de escritura, los diferentes núcleos pueden pasar a la operación de lectura subsiguiente a una operación de escritura previa. De esta forma se relaja el orden específico existente en la consistencia secuencial donde luego de finalizada la escritura se procede a ejecutar la lectura. Una técnica que emplea estos *buffers* de escritura para mejorar el desempeño mediante la relajación de la condición anterior es la **consistencia de memoria basada en pedido total de almacenamiento** (**TSO**, por sus siglas en inglés). Es por ello que el TSO es considerado como un modelo de consistencia relajado. En el mismo todos los núcleos insertan sus peticiones de lectura y almacenamiento en una solicitud global, respetando el orden del programa. Una forma de incrementar aún más el desempeño se logra mediante la utilización de *buffers* de reordenamiento, posibilitando de esta forma la ejecución de las instrucciones de manera no organizada, dando lugar a los modelos de consistencia relajados.

Glosario

Arquitectura del Repertorio de Instrucciones

Es un modelo abstracto del computador que define su funcionalidad, conocido además como arquitectura del computador. Existen diversas maneras para llevar a cabo su implementación. Constituye una interface entre el *software* y el *hardware*, por lo que cada tipo se *software* será compatible con el *hardware* que soporte las instrucciones correspondientes.

Código Estándar Estadounidense para el Intercambio de Información

Código que permite representar caracteres alfanuméricos mediante 7 bits. Además, permite representar caracteres de control.

Computadores con Repertorio de Instrucciones Complejo

Tipo de arquitectura del computador que ha permitido reducir el tamaño y complejidad de los programas en lenguaje de máquina permitiendo una mayor eficiencia de estos. Para ello emplean una mayor complejidad de la arquitectura incluyendo una mayor cantidad de tipos de datos, modos de direccionamiento y repertorio de instrucciones mucho más amplio y variado.

Computadores con Repertorio de Instrucciones Reducido

Tipo de arquitectura del computador de complejidad reducida donde solo se implementan las instrucciones más usadas, los modos de direccionamiento más sencillos y los tipos de datos básicos. Es común en esta arquitectura contar con un elevado número de registros para el almacenamiento de datos temporales.

Memoria de Acceso Aleatorio

Tipo de memoria en la que cada dirección posee un mecanismo de direccionamiento físico único. En estas, el tiempo de acceso a cualquier dirección es constante e independiente de la secuencia de acceso previa. De esta manera es posible accederse a cualquier dirección de manera aleatoria.

Memoria Estática de Acceso Aleatorio

Tipo de memoria RAM que emplea los mismos elementos lógicos que los procesadores, en estos los valores binarios se almacenan mediante compuertas lógicas tradicionales y se mantienen todo el tiempo que se mantengan energizada la memoria.

Pila

Organización especial de la memoria, constituida por un conjunto de posiciones en forma de cola del tipo FIFO. Básicamente existen tres operaciones con la pila: introducir (*push*), extraer (*pop*) y operaciones con una o dos posiciones cabeceras de la pila.

Primero Entra - Primero Sale

Estructura de datos caracterizada por ser una secuencia de elementos en la que la operación de inserción *push* se realiza por un extremo y la operación de extracción *pop* por el otro; dicho de otro modo, el primer elemento en entrar será también el primero en salir.

Sequential Access Memory

Tipo de memoria organizada en unidades de datos, en la que el acceso se realiza mediante una secuencia lineal específica. En estas se utilizan mecanismos de lectura y escritura compartidos, los cuales deben posicionarse desde la posición en que se encuentren hasta la dirección a la cual se quiere acceder. Como resultado los tiempos de acceso son extremadamente variables.

Standard Performance Evaluation Corporation

Consorcio sin fines de lucro que tiene como objetivos: crear un *benchmark* estándar para medir el rendimiento de computadoras, así como controlar y publicar los resultados de estos *tests*.

Translation Lookaside Buffer

Tipo de caché especial utilizada para almacenar las entradas de la tabla de páginas usadas más recientemente.

Unidad Aritmética Lógica

Unidad del computador encargada de realizar las operaciones elementales: aritméticas (suma, resta, etc.) y lógicas (AND, OR, XOR, etc.). Los datos utilizados por esta unidad provienen de la memoria principal y pueden almacenarse tanto en dicha memoria como en los registros. En la mayoría de los computadores esta unidad se implementa en base a sumadores y restadores.

Unidad Central de Procesamiento

Parte del computador que controla las operaciones del computador y realiza las funciones de procesamiento de datos. Forman parte de esta la unidad aritmética lógica, la unidad de control y los registros.

Unidad de Administración de Memoria

Permite la traducción de las direcciones virtuales a direcciones físicas en la memoria principal para realizar las lecturas y lecturas.

Enlaces de interés

Sitio de acompañamiento del libro *Computer Architecture: A Quantitative Approach*, 5th Edition.

Contiene material suplementario y de acompañamiento al libro de texto. Dentro de estos materiales se encuentran diversos apéndices que abarcan diversos temas como: arquitecturas específicas, sistemas embebidos, procesadores de aplicaciones específicas, entre otras.

http://booksite.mkp.com/9780123838728/

Sitio de acompañamiento del libro *Essentials of Computer Organization and Architecture*, Third Edition.

Contiene programas, transparencias y material relevante del libro de texto.

http://computerscience.jbpub.com/ecoa/3e/

Sitio de acompañamiento del libro *Structure Computer Organization* Sixth Edition.

Contiene recursos para estudiantes e instructores.

https://media.pearsoncmg.com/bc/abp/cs-resources/products/product.html#product,isbn=0132916525

Sitio web del libro *Computer Organization and Architecture*.

Incorpora recursos para estudiantes e instructores que incluyen simulaciones interactivas, manuales de solución, proyectos, bancos de pruebas para el mejor entendimiento de las características más relevantes en el diseño de computadores. Dispone además de asignaciones con problemas específicos para los diferentes temas abordados en el libro de texto.

http://williamstallings.com/ComputerOrganization/

Sitio web del Standard Performance Evaluation Corporation.

Contiene diferentes *benchmarks* y herramientas para la evaluación del desempeño y la eficiencia energética de computadores.

https://www.spec.org/

Bibliografía

Referencias bibliográficas

Abd-El-Barr, M. & El-Rewini, H. (2005). *Fundamentals of Computer Organization and Architecture*. New Jersey, United States: John Wiley & Sons.

Flynn, M. J., (1995). *Computer architecture: pipelined and parallel processor design.* Sudbury, United States: Jones and Barlett Publishers.

Hamacher, C.; Vranesic, Z.; Zaky, S. & Manjikian, N. (2012). *Computer organization and embedded systems 6Ed*. Ney York, United States: McGraw-Hill.

Hennessy, J. L. & Patterson, D. A., (2011). *Computer architecture: a quantitative approach 5Ed*. San Francisco, United States: Elsevier.

Hwang, K. & Briggs, F. A., (1984). *Computer Architecture and Parallel Processing.* Ney York, United States: McGraw-Hill.

Lamport, L. (1979). How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs, *IEEE Transactions on Computers*, (9), 690-691.

Lanchares Dávila, J. (2000). *Apuntes de Estructura de Computadores.* Recuperado de http://www.dacya.ucm.es/lanchares/documentos/2.9.5%20Apuntes%20de%20Estructura%20de%20Computadores.pdf

Null, L. & Lobur, J. (2003). *The Essentials of Computer Organization and Architecture.* Sudbury, United States: Jones and Barlett Publishers.

Stallings, W., (2013). *Computer Organization and Architecture: Designing for Performance 9Ed*. New Jersey, United States: Pearson.

Tanenbaum, A. S. & Austin, T., (2013). *Structured Computer Organization 6Ed.* New Jersey, United States: Pearson.

Agradecimientos

Autor
George Enrique Figueras Benítez