

Introducción a Arquitectura de las Computadoras

Diego Feroldi

Arquitectura del Computador
Departamento de Ciencias de la Computación
FCEIA-UNR

Agosto de 2020



Índice

1. Definición de computadora	1
2. Descripción básica de la arquitectura de una computadora	1
2.1. Procesador	1
2.2. Memoria principal	2
2.3. Buses	3
2.4. Dispositivos de entrada/salida	3
3. Ejecución de un programa	3
4. Jerarquía de memoria	5
5. Representación de programas a nivel de máquina	5
6. Definición de Arquitectura de Computadores	8
7. Breve historia de la computación	8
7.1. Computadoras mecánicas	8
7.2. Computadoras con válvulas de vacío	10
7.3. Computadoras con transistores	11
7.4. Computadoras con circuitos integrados	12
7.5. Computadoras con integración a muy gran escala	13

1. Definición de computadora

“Una computadora digital es una máquina que puede resolver problemas ejecutando las instrucciones que recibe de las personas”^[7].

La anterior es una de las tantas definiciones posibles de qué es una computadora. Esta definición indica que a pesar de las enormes prestaciones de las computadoras actuales, en su nivel más elemental las computadoras resuelven operaciones muy simples. En efecto, es importante destacar que los circuitos integrados de una computadora solo pueden reconocer y ejecutar un conjunto muy limitado de instrucciones sencillas. A lo largo de esta asignatura veremos como se realizan estas operaciones. En primer lugar, veamos un poco cómo están constituidas las computadoras.

2. Descripción básica de la arquitectura de una computadora

Los componentes básicos de una computadora incluyen una Unidad Central de Procesamiento (CPU), almacenamiento primario o memoria de acceso aleatorio (RAM, por su sigla en inglés *Random Access Memory*), almacenamiento secundario, dispositivos de entrada/salida (pantalla, teclado, mouse, etc.) y una interconexión entre los diferentes componentes denominada Bus. Un diagrama muy básico de la arquitectura de una computadora se muestra en la Figura [1].

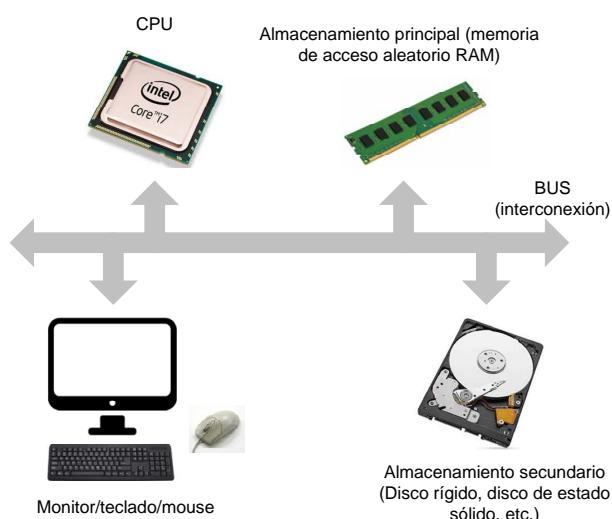


Figura 1: Esquema básico de la arquitectura de una computadora.

Esta arquitectura se conoce típicamente como Arquitectura de von Neumann, o Arquitectura de Princeton, y fue descrita en 1945 por el matemático y físico John von Neumann. Los programas y los datos generalmente se almacenan en un almacenamiento secundario (por ejemplo, un disco rígido o de estado sólido). Cuando se ejecuta un programa, tanto los programas como los datos deben copiarse desde el almacenamiento secundario hacia el almacenamiento primario o memoria principal (RAM). Luego, la CPU ejecuta el programa desde el almacenamiento primario.

2.1. Procesador

La unidad central de procesamiento (CPU), o simplemente el procesador, es el “motor” que interpreta (o ejecuta) las instrucciones almacenadas en la memoria principal. La CPU incluye

varias unidades funcionales, incluida la unidad aritmética lógica (ALU, por su sigla en inglés *Arithmetic Logic Unit*), que es la parte del chip que realmente realiza los cálculos aritméticos y lógicos. Para que la ALU pueda operar, el chip también contiene registros y memoria caché. Un registro de la CPU, o simplemente un registro, es un almacenamiento temporal o una ubicación de trabajo integrada en la propia CPU (separada de la memoria). El intercambio de datos entre la CPU y la memoria es una parte crucial de los cálculos en una arquitectura von Neumann: las instrucciones deben recuperarse de la memoria, los operandos también deben recuperarse de la memoria y algunas instrucciones almacenan resultados también en la memoria. Esto crea un cuello de botella y conduce a una pérdida de tiempo de la CPU cuando espera la respuesta de datos del chip de memoria. Por ello, para evitar una espera constante, el procesador está equipado con sus propias celdas de memoria, llamadas registros. Estos son pocos pero rápidos. Por otra parte, la memoria caché es un pequeño subconjunto del almacenamiento primario o RAM ubicado en el chip de la CPU. Los registros de la CPU y la memoria caché se describirán con mayor detalle en la Sección 4. Cabe señalar que el diseño interno y la configuración de un procesador moderno es bastante complejo. Este apunte proporciona una vista de alto nivel muy simplificada de algunas unidades funcionales clave dentro de una CPU.

En el núcleo de la CPU hay un dispositivo de almacenamiento (o registro) del tamaño de una palabra llamado contador de programa (PC). En cualquier momento, el PC apunta a (tiene la dirección de) alguna instrucción en lenguaje máquina en la memoria principal. Desde el momento en que se aplica la alimentación al sistema, hasta el momento en que se apaga la alimentación, un procesador ejecuta repetidamente la instrucción señalada por el contador del programa y actualiza el contador del programa para que apunte a la siguiente instrucción. El funcionamiento de un procesador se puede interpretar de acuerdo con un modelo de ejecución de instrucciones muy simple, definido por el conjunto de instrucciones de la arquitectura (ISA, por las siglas en inglés *Instruction Set Architecture*). En este modelo, las instrucciones se ejecutan en secuencia estricta, y ejecutar una sola instrucción implica realizar una serie de pasos: el procesador lee la instrucción de la memoria señalada por el contador del programa (PC), interpreta los bits en la instrucción, realiza una operación simple dictada por la instrucción y luego actualiza el PC para que apunte a la siguiente instrucción, que puede o no ser contigua en memoria a la instrucción que se acaba de ejecutar.

Solo hay algunas de estas operaciones simples, y giran en torno a la memoria principal, el archivo de registros y la unidad aritmética/lógica (ALU). La función de la ALU es calcular nuevos datos y valores de dirección. Estos son algunos ejemplos de las operaciones simples que la CPU podría realizar a pedido de una instrucción:

- **Cargar:** copiar información de la memoria principal en un registro, sobrescribiendo los contenidos anteriores del registro.
- **Almacenar:** copiar información de un registro a una ubicación en la memoria principal, sobrescribiendo el contenido anterior de esa ubicación.
- **Operar:** copiar el contenido de dos registros en la ALU, realizar una operación aritmética entre las dos palabras y almacenar el resultado en un registro, sobrescribiendo el contenido anterior de ese registro.
- **Saltar:** Extraer una palabra de la instrucción misma y copiarla en el contador del programa, sobrescribiendo el valor anterior del PC.

2.2. Memoria principal

La memoria principal es un dispositivo de almacenamiento temporal que contiene tanto el programa como los datos que manipula mientras el procesador ejecuta el programa. Físicamente, la memoria principal consiste en una colección de chips de memoria dinámica de acceso aleatorio

(DRAM, por su sigla en inglés *Dynamic Random Access Memory*). A nivel lógico, la memoria está organizada como una matriz lineal de bytes¹, cada uno con su propia dirección única (índice de matriz) que comienza en cero. En general, cada una de las instrucciones de la máquina que constituyen un programa puede consistir en un número variable de bytes.

Los tamaños de los elementos de datos que corresponden a las variables del programa C varían según el tipo. Por ejemplo, en una máquina x86-64 que ejecuta Linux, los datos de tipo `char` requieren 1 byte, los `short` requieren dos bytes, los `int` 4 bytes, los tipo `long` 8 bytes, los `float` 4 bytes y los `double` 8 bytes.

El almacenamiento primario o la memoria principal también se conoce como memoria volátil dado que cuando se corta la energía, la información no se retiene y, por lo tanto, se pierde. Por el contrario, el almacenamiento secundario (por ejemplo, el disco rígido) se denomina memoria no volátil ya que la información se retiene cuando la computadora se apaga.

2.3. Buses

A lo largo de todo el sistema existe una colección de conductores eléctricos llamados buses que transportan bytes de información de un lado a otro entre los componentes. Los buses suelen estar diseñados para transferir fragmentos de información de tamaño fijo conocidos como palabras. El número de bytes en una palabra (el tamaño de la palabra) es un parámetro fundamental del sistema que varía de un sistema a otro. La mayoría de las máquinas actuales tienen tamaños de palabra de 4 bytes (32 bits) u 8 bytes (64 bits). En este curso nos centraremos en arquitecturas de 64 bits.

2.4. Dispositivos de entrada/salida

Los dispositivos de entrada/salida (*I/O devices*) son la conexión del sistema al mundo externo. La computadora del ejemplo de la Figura 1 tiene cuatro dispositivos de E/S: un teclado y un mouse para la entrada del usuario, una pantalla para la salida del usuario y una unidad de disco (o simplemente un disco rígido) para el almacenamiento a largo plazo de datos y programas. Cada dispositivo de E/S está conectado al bus de E/S mediante un controlador o un adaptador. Los controladores son conjuntos de chips en el propio dispositivo o en la placa de circuito impreso principal del sistema (a menudo denominada placa base). Un adaptador es una tarjeta que se conecta a una ranura en la placa base. En ambos casos, el propósito es transferir información de ida y vuelta entre el bus de E/S y un dispositivo de E/S.

3. Ejecución de un programa

Para comprender qué sucede cuando ejecutamos un programa (vamos a ejemplificar con el típico `hello.c`), necesitamos comprender la organización del hardware de una computadora típica. Para ello en la Figura 2 vemos una versión un poco más detallada de la Figura 1. Esta imagen en particular sigue el modelo de la familia de microprocesadores Intel. Simplificando y omitiendo muchos detalles que se verán con más detalle más adelante, podemos describir el proceso de la siguiente manera. Inicialmente, el programa shell está ejecutando sus instrucciones, esperando que escribamos un comando. A medida que escribimos los caracteres `./hello` en el teclado, el programa shell lee cada uno en un registro y luego lo almacena en la memoria. Cuando presionamos la tecla `Enter` en el teclado, el shell sabe que hemos terminado de escribir el comando. Luego, el shell carga el archivo ejecutable `hello` ejecutando una secuencia de

¹El bit (acrónimo de *Binary digit*) es la mínima unidad de información en un sistema binario. Un bit o dígito binario puede representar uno de estos dos valores: 0 o 1. A su vez, un conjunto ordenado de ocho bits se denomina byte. Este tema se verá con mayor detalle en el apunte **Representación Computacional de Datos**.

instrucciones que copia el código y los datos del archivo objeto `hello` del disco a la memoria principal. Los datos incluyen la cadena de caracteres "holá, mundo \n" que finalmente se imprimirá. Usando una técnica conocida como acceso directo a la memoria (DMA, por la sigla en inglés *Direct Memory Access*), los datos viajan directamente desde el disco a la memoria principal, sin pasar por el procesador. Una vez que el código y los datos en el archivo de objeto `hello` se cargan en la memoria, el procesador comienza a ejecutar las instrucciones en lenguaje de máquina en la rutina principal del programa `hello`. Estas instrucciones copian los bytes de la cadena "holá, mundo \n" de la memoria a los registros de la CPU, y desde allí al dispositivo de visualización, donde se muestran en la pantalla.

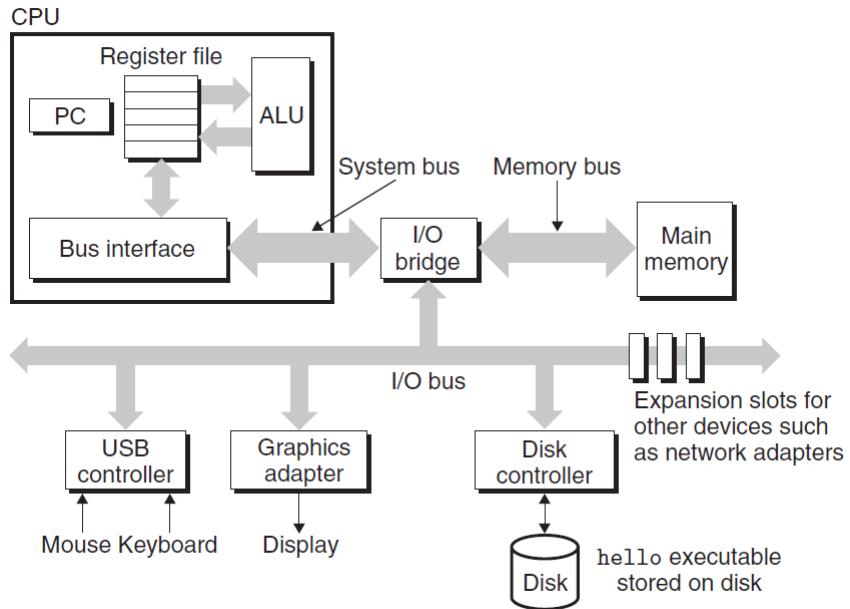


Figura 2: Organización de hardware de una computadora típica. CPU: Unidad central de procesamiento, ALU: Unidad aritmética/lógica, PC: Contador de programas, USB: Bus serie universal (fuente: [2]).

Un punto importante a destacar es que una computadora pasa mucho tiempo moviendo información de un lugar a otro. Las instrucciones de máquina en el programa `hello` se almacenan originalmente en el disco. Cuando se carga el programa, se copian en la memoria principal. A medida que el procesador ejecuta el programa, las instrucciones se copian de la memoria principal al procesador. De manera similar, la cadena de datos "holá, mundo \n", originalmente en el disco, se copia en la memoria principal y luego se copia desde la memoria principal al dispositivo de visualización. Gran parte de estas operaciones de copia constituyen una sobrecarga que ralentiza el "trabajo real" del programa. Por lo tanto, un objetivo principal para los diseñadores de sistemas es hacer que estas operaciones de copia se ejecuten lo más rápido posible.

Debido a las leyes físicas, los dispositivos de almacenamiento más grandes son más lentos que los dispositivos de almacenamiento más pequeños. Además, los dispositivos más rápidos son más caros de construir que sus homólogos más lentos. Por ejemplo, la unidad de disco en un sistema típico puede ser 1.000 veces más grande que la memoria principal, pero el procesador puede tardar 10.000.000 veces más en leer una palabra del disco que de la memoria. De manera similar, los registros de la CPU almacenan solo unos pocos bytes de información, a diferencia de miles de millones de bytes en la memoria principal. Sin embargo, el procesador puede leer datos de los registros casi 100 veces más rápido que de la memoria. Aún más problemático, a medida que la tecnología de semiconductores progresara a lo largo de los años, esta brecha en la memoria del procesador continúa aumentando. Es más fácil y económico hacer que los

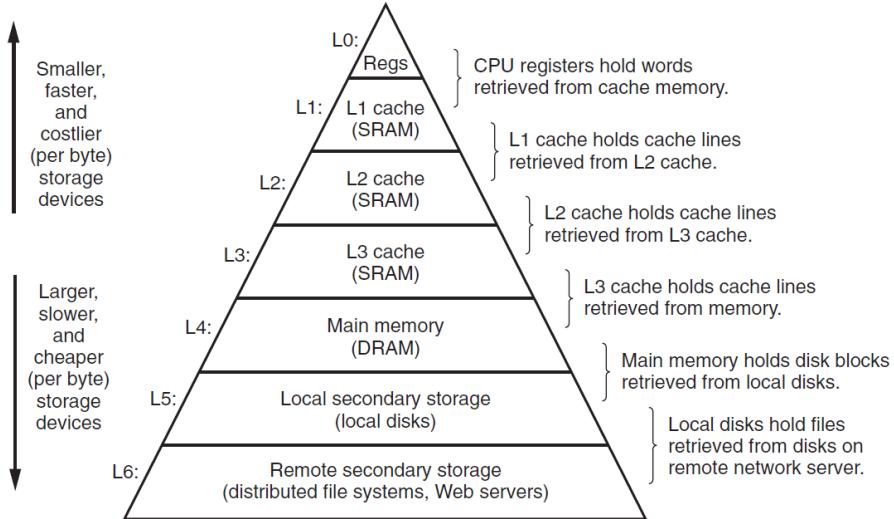


Figura 3: Ejemplo de jerarquía de memoria (fuente: [2]).

procesadores funcionen más rápido que hacer que la memoria principal funcione más rápido.

Para lidiar con la brecha de memoria del procesador, los diseñadores del sistema incluyen dispositivos de almacenamiento más pequeños y más rápidos llamados memorias de caché (o simplemente cachés) que sirven como áreas de almacenamiento temporales para la información que el procesador probablemente necesitará en el futuro cercano. Los cachés se implementan con una tecnología de hardware conocida como memoria estática de acceso aleatorio (SRAM, *Static random-access memory*). Los sistemas más nuevos y potentes incluso tienen tres niveles de caché: L1, L2 y L3. La idea detrás del almacenamiento en caché es que un sistema puede obtener el efecto de una memoria muy grande y muy rápida explotando la localidad, la tendencia de los programas a acceder a datos y códigos en regiones localizadas.

4. Jerarquía de memoria

Los dispositivos de almacenamiento en cada sistema informático están organizados como una jerarquía de memoria similar a la Figura 3. A medida que avanzamos desde la parte superior de la jerarquía hacia la parte inferior, los dispositivos se vuelven más lentos, más grandes y menos costosos por byte. Los registros ocupan el nivel superior en la jerarquía, que se conoce como nivel 0 o L0. En la figura se muestran tres niveles de almacenamiento en caché de L1 a L3, ocupando los niveles de jerarquía de memoria 1 a 3. La memoria principal ocupa el nivel 4, y así sucesivamente. La idea principal de una jerarquía de memoria es que el almacenamiento en un nivel sirve como caché para el almacenamiento en el siguiente nivel inferior. Por lo tanto, los registros son un caché para el caché L1. Los cachés L1 y L2 son cachés para L2 y L3, respectivamente. El caché L3 es un caché para la memoria principal, que a su vez es un caché para el disco. La caché L1 se divide en dos tipos: una de datos y otra de instrucciones. La tendencia actual es colocar los tres niveles de caché en el procesador y que cada nueva serie de procesadores tenga tamaños de caché más grandes.

5. Representación de programas a nivel de máquina

Las computadoras ejecutan código de máquina, secuencias de bytes que codifican las operaciones de bajo nivel que manipulan datos, administran la memoria, leen y escriben datos en dispositivos de almacenamiento y se comunican a través de redes. Un compilador genera código

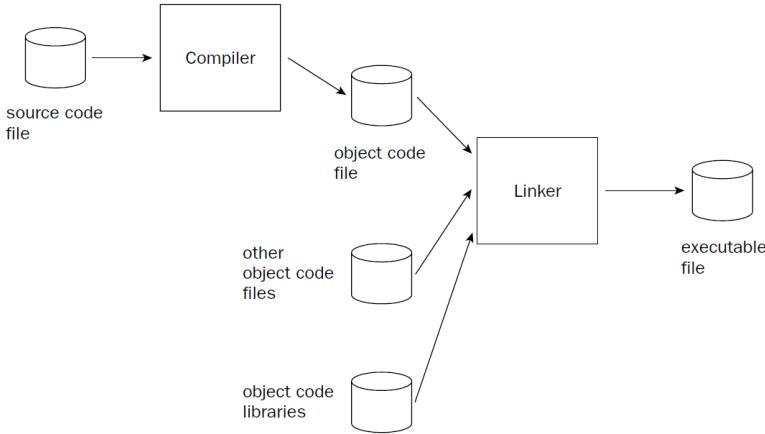


Figura 4: Proceso de compilación (fuente: [\[1\]](#)).

de máquina a través de una serie de etapas, basadas en las reglas del lenguaje de programación, el conjunto de instrucciones de la máquina de destino y las convenciones seguidas por el sistema operativo. Por ejemplo, el compilador *gcc* genera su salida en forma de código Assembler, una representación textual del código de máquina que proporciona las instrucciones individuales en el programa. Luego, *gcc* invoca tanto un ensamblador como un enlazador para generar el código de máquina ejecutable a partir del código Assembler. La Figura 4 ilustra este proceso.

Es interesante analizar el código de máquina y sobre todo su representación legible por humanos como es el código Assembler. Por ejemplo, el código equivalente del simple programa en lenguaje C que imprime por pantalla la !Hola Mundo!:

```
#include<stdio.h>
int main()
{
    printf("¡Hola Mundo!\n");
    return 0;
}
```

es el siguiente:

400506:	48 c7 c7 d8 08 60 00	mov \$0x6008d8, %rdi
40050d:	e8 ce fe ff ff	callq 4003e0 <puts@plt>
400512:	c3	retq

En el código precedente podemos visualizar a la izquierda las direcciones de memoria de las instrucciones, luego en el centro el equivalente en lenguaje de máquina escrito en formato hexadecimal mientras que en la parte derecha vemos el equivalente en lenguaje Assembler. Si bien todavía no sabemos Assembler, es evidente que esta representación es mucho más legible y comprensible para un humano que el código de máquina puro.

El lenguaje ensamblador es específico de la máquina. Por ejemplo, el código escrito para un procesador x86-64 no se ejecutará en un procesador diferente, por ejemplo en un procesador RISC (popular en tabletas y teléfonos inteligentes). El lenguaje ensamblador es un lenguaje de “bajo nivel” y proporciona la interfaz de instrucción básica para el procesador de la computadora. Para un programador, el lenguaje ensamblador es lo más cercano al procesador. Los programas escritos en un lenguaje de alto nivel se traducen al lenguaje ensamblador para que el procesador ejecute el programa. El lenguaje de alto nivel es una abstracción entre el lenguaje y las instrucciones reales del procesador. El lenguaje ensamblador le brinda al programador control directo de los recursos del sistema. Esto implica establecer registros del procesador,

acceder a ubicaciones de memoria e interactuar con otros elementos de hardware. Esto requiere una comprensión significativamente profunda de cómo funcionan exactamente el procesador y la memoria.

Al programar en un lenguaje de alto nivel como C, no necesitamos preocuparnos por la implementación detallada de nuestro programa a nivel de máquina. Por el contrario, al escribir programas en código Assembler (como se hizo en los primeros días de la informática), un programador debe especificar las instrucciones de bajo nivel que el programa utiliza para llevar a cabo un cálculo. La mayoría de las veces, es mucho más productivo y confiable trabajar en el nivel más alto de abstracción proporcionado por un lenguaje de alto nivel. La verificación de tipo proporcionada por un compilador ayuda a detectar muchos errores de programa y asegura que hagamos referencia y manipulemos los datos de manera consistente. Con los compiladores y optimizadores modernos, el código generado suele ser al menos tan eficiente como lo que escribiría a mano un programador experto en lenguaje ensamblador. Lo mejor de todo es que un programa escrito en un lenguaje de alto nivel se puede compilar y ejecutar en varias máquinas diferentes, mientras que el código de ensamblaje es altamente específico de la máquina.

Entonces, ¿por qué deberíamos aprender Assembler? El lenguaje ensamblador tiene varios beneficios:

- **Velocidad.** Los programas en lenguaje ensamblador son generalmente los programas más rápidos.
- **Espacio.** Los programas en lenguaje ensamblador suelen ser los más pequeños.
- **Capacidad.** Se pueden hacer cosas en lenguaje ensamblador que son difíciles o imposibles en HLL².
- **Conocimiento.** El conocimiento del lenguaje ensamblador ayuda a escribir mejores programas, incluso cuando use HLL.

Aunque los compiladores hacen la mayor parte del trabajo en la generación de código Assembler, poder leerlo y comprenderlo es una habilidad importante para los programadores avanzados. Al invocar al compilador con los parámetros de línea de comandos apropiados, el compilador generará un archivo que muestra su salida en forma de código de ensamblaje. Al leer este código, podemos entender las capacidades de optimización del compilador y analizar las inefficiencias subyacentes en el código. Los programadores que buscan maximizar el rendimiento de una sección crítica de código a menudo prueban diferentes variaciones del código fuente, cada vez que compilan y examinan el código de ensamblaje generado para tener una idea de qué tan eficientemente se ejecutará el programa. Además, hay momentos en que la capa de abstracción proporcionada por un lenguaje de alto nivel oculta información sobre el comportamiento en tiempo de ejecución de un programa que debemos comprender. Por ejemplo, cuando se escriben programas concurrentes utilizando un paquete de subprocessos, es importante saber qué región de la memoria se usa para contener las diferentes variables del programa. Esta información es visible a nivel de código Assembler. Como otro ejemplo, muchas de las formas en que se pueden atacar los programas, permitiendo que gusanos y virus infesten un sistema, implican matices de la forma en que los programas almacenan la información en tiempo de ejecución. Muchos ataques implican la explotación de las debilidades en los programas del sistema para sobrescribir la información y, por lo tanto, tomar el control del sistema. Comprender cómo surgen estas vulnerabilidades y cómo protegerse de ellas requiere un conocimiento de la representación de los programas a nivel de máquina. La necesidad de que los programadores aprendan código Assembler ha cambiado con el paso de los años de poder escribir programas directamente Assembler a ser capaz de leer y comprender el código generado por los compiladores. Finalmente, podríamos concluir diciendo que la razón principal para aprender el lenguaje

²HLL, programas de alto nivel por su sigla en inglés *high level languages*.

ensamblador radica más en entender cómo funciona una computadora en lugar de desarrollar programas grandes.

6. Definición de Arquitectura de Computadores

En el pasado, el término arquitectura del computador o arquitectura de computadores a menudo se refería solo al diseño del conjunto de instrucciones. Otros aspectos del diseño de la computadora fueron llamados *implementación*, a menudo insinuando que la implementación no es interesante o menos desafiante. Sin embargo, el trabajo del arquitecto o diseñador es mucho más que el diseño del conjunto de instrucciones, y los obstáculos técnicos en los otros aspectos del proyecto son probablemente más desafiantes que los encontrados en el diseño del conjunto de instrucciones.

El término conjunto de instrucciones de la arquitectura (ISA, por las siglas en inglés *Instruction Set Architecture*) se refiere al conjunto de instrucciones real disponible para programar. El ISA sirve como límite entre el software y el hardware. El resto de la arquitectura del computador se refiere al diseño de la organización y el hardware para cumplir objetivos y requisitos funcionales. La implementación de una computadora tiene dos componentes: *organización* y *hardware*. El término organización incluye los aspectos de alto nivel del diseño de una computadora, como el sistema de memoria, la interconexión de memoria y el diseño del procesador interno o CPU (unidad central de procesamiento, donde se implementa la aritmética, lógica, ramificación y transferencia de datos). Por ejemplo, dos procesadores con las mismas arquitecturas de conjunto de instrucciones pero organizaciones muy diferentes son el AMD Opteron 64 y el Intel Pentium 4. Ambos procesadores implementan el conjunto de instrucciones x86, pero tienen organizaciones muy diferentes de canalización y caché. El hardware se refiere a los detalles de una computadora, incluido el diseño lógico detallado y la tecnología de empaquetado de la computadora. A menudo, una línea de computadoras contiene computadoras con arquitecturas de conjuntos de instrucciones idénticas y organizaciones casi idénticas, pero difieren en la implementación detallada del hardware. Por ejemplo, el Pentium 4 y el Mobile Pentium 4 son casi idénticos, pero ofrecen diferentes velocidades de reloj y diferentes sistemas de memoria, lo que hace que el Mobile Pentium 4 sea más efectivo para computadoras de gama baja.

En esta asignatura nos vamos a focalizar en primer lugar en la arquitectura x86-64. El x86-64 es un diseño de CPU de conjunto de instrucciones complejas (CISC, por su sigla en inglés *Complex Instruction Set Computing*). Esto se refiere a la filosofía de diseño del procesador interno. Los procesadores CISC generalmente incluyen una amplia variedad de instrucciones (a veces superpuestas), diferentes tamaños de instrucciones y una amplia gama de modos de direccionamiento. El término se acuñó retroactivamente en contraste con la computadora de conjunto de instrucciones reducidas (RISC, por su sigla en inglés *Reduced Instruction Set Computer*). Una de las arquitecturas tipo RISC es la denominada RISC-V, la cual estudiaremos al final del curso.

7. Breve historia de la computación

Se han construido cientos de tipos diferentes de computadoras desde sus orígenes. Algunas con gran impacto y otras sección es dar un pantallazo muy breve sobre los orígenes de la computación marcando algunos hitos relevantes.

7.1. Computadoras mecánicas

- Blaise Pascal (1623-1662). Dispositivo construido en 1642 construido totalmente mecánico con engranajes. Solo podía restar y sumar.

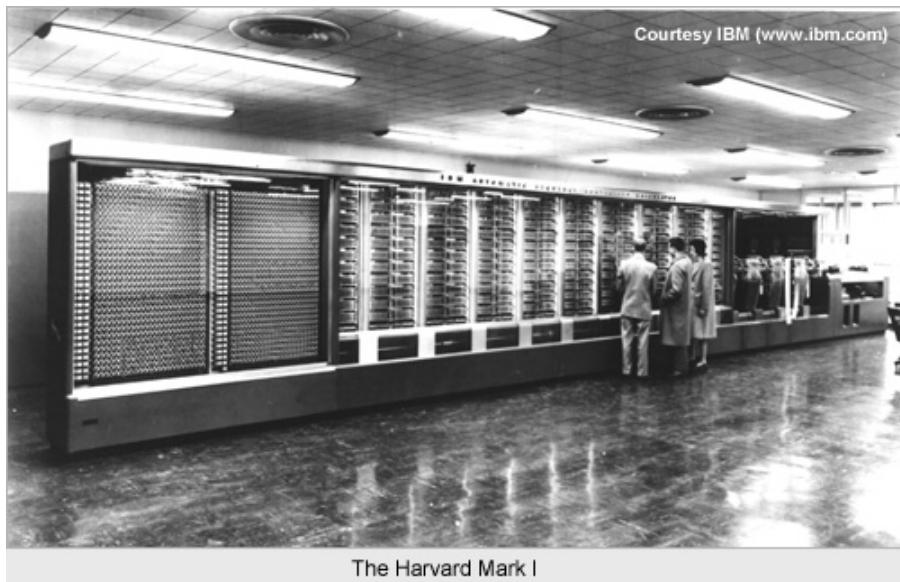


Figura 5: Imagen de la computadora Mark I.

- Gottfried von Leibnitz (1646-1716). Construyó otra máquina totalmente mecánica que además podía multiplicar y dividir.
- Charles Babbage (1792-1871). Construyó una máquina diseñada para ejecutar un solo algoritmo con el objetivo de calcular tablas numéricas útiles para la navegación. Perforaba sus resultados en una placa de cobre.
- Luego desarrolló la máquina analítica para poder ejecutar diferentes algoritmos. Tenía cuatro componentes: el almacén (memoria), el molino (unidad de cómputo), la sección de entrada (lector de tarjetas perforadas) y la sección de salida (salidas perforadas e impresas). Podía sumar, restar, multiplicar y dividir operandos.
- Ada Lovelace (1815-1852). Primera programadora de computadoras del mundo.
- Konrad Zuse (1910-1995). Construyó una serie de máquinas calculadoras automáticas empleando contactores electromagnéticos³.
- John Atanasoff (1903-1995). Diseñó una máquina que utilizaba aritmética binaria y tenía una memoria de condensadores empleando un proceso que denominó “refrescar la memoria”. Nunca funcionó por problemas de implementación aunque el concepto era correcto.
- George Stibitz (1904-1995). Realizó una máquina más primitiva que la de Atanasoff aunque si funcionó.
- Howard Aiken (1900-1973). Construyó con relés la máquina de propósito general que Babbage no había podido construir con ruedas dentadas. La primera máquina de Aiken (Mark I) se completó en Harvard en 1944. Tenía 72 palabras de 23 dígitos decimales cada una y un tiempo de instrucción de 6 segundos. Las entradas y salidas se efectuaban con cintas de papel perforadas (Ver Figura 5). Para cuando se terminó de desarrollar la Mark II las máquinas de relés ya eran obsoletas.

³También denominados relés o relevadores. Un relé es un dispositivo electromagnético que funciona como un interruptor controlado por un circuito eléctrico en el que, por medio de una bobina y un electroimán, se acciona un juego de uno o varios contactos que permiten abrir o cerrar otros circuitos eléctricos independientes.

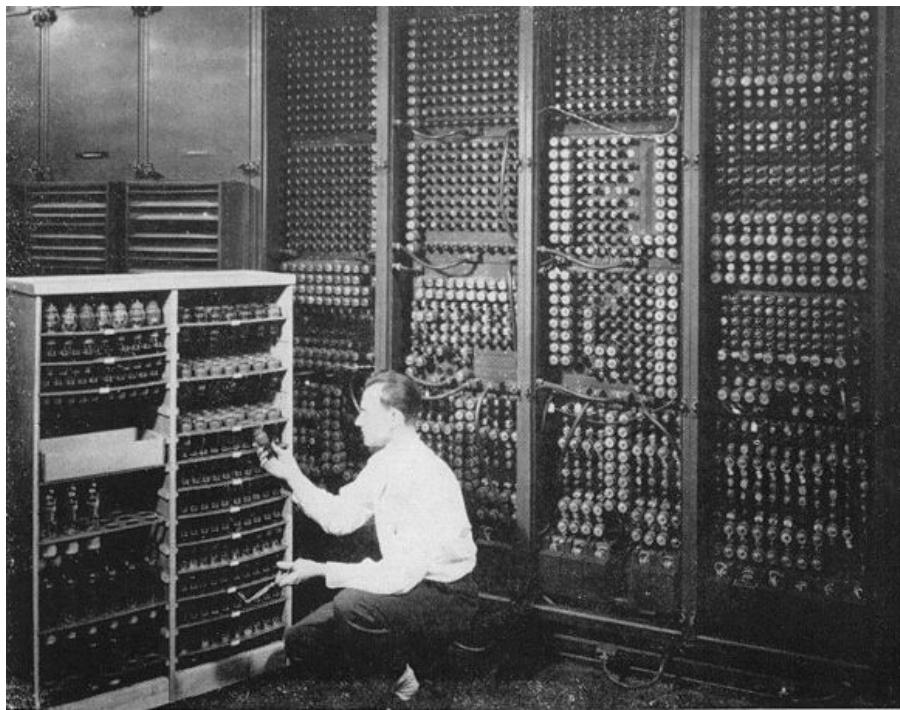


Figura 6: Imagen de la computadora ENIAC.

7.2. Computadoras con válvulas de vacío

La válvula electrónica, válvula de vacío, tubo de vacío o bulbo, es un componente electrónico utilizado para amplificar, conmutar, o modificar una señal eléctrica mediante el control del movimiento de los electrones en un espacio “vacío” a muy baja presión, o en presencia de gases especialmente seleccionados. La válvula originaria fue el componente crítico que posibilitó el desarrollo de la electrónica durante la primera mitad del siglo XX, incluyendo la expansión y comercialización de la radiodifusión, televisión, radar, audio, redes telefónicas, computadoras analógicas y digitales, control industrial, etc. Algunas de estas aplicaciones son anteriores a la válvula, pero experimentaron un crecimiento explosivo gracias a ella [10].

A continuación se enumeran algunos de los avances más notables de la computación utilizando esta tecnología:

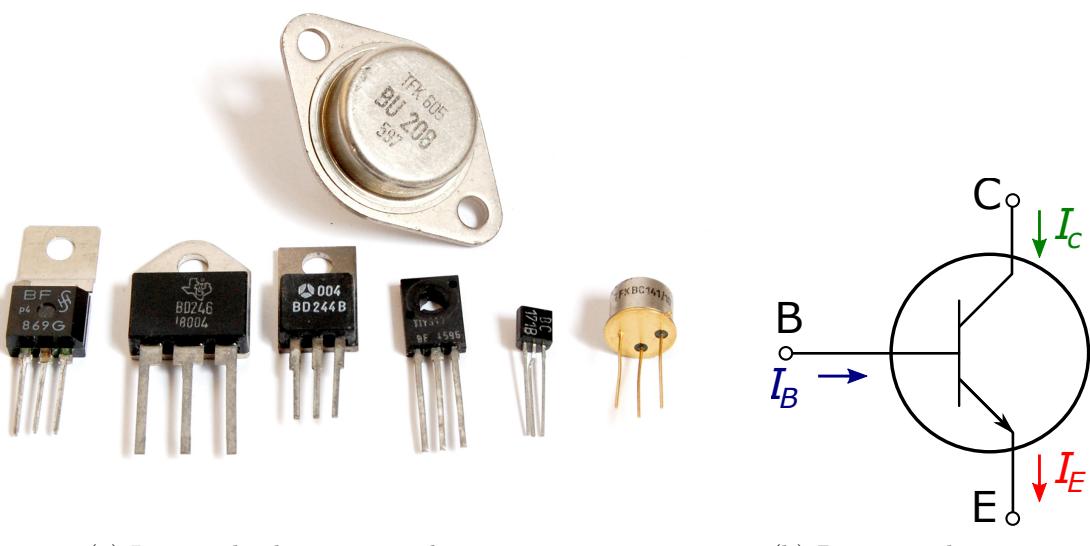
- Alan Turing (1912-1954). Contribuyó a la creación de COLOSSUS, la primera computadora electrónica, desarrollada por el gobierno inglés.
- John Mauchley (1907-1980). Construyó ENIAC en 1946, la primera computadora moderna (Ver Figura 6). Tenía 18000 válvulas y 1500 relés. Pesaba 30 toneladas y consumía 140 kW. En la Figura 7 se puede apreciar una válvula de vacío.
- EDSAC. Máquina sucesora de ENIAC en 1949.
- John von Neumann (1903-1957). Desarrolló en 1952 la máquina IAS donde se emplea el diseño que ahora se conoce como **máquina de Von Neumann**, la cual tiene cinco partes básicas: la memoria, la unidad aritmética lógica, la unidad de control y el equipo de entrada y salida. Este diseño sigue siendo la base de casi todas las computadoras digitales aun hoy día.
- Whirlwind I (1951). Diseñada en el MIT. Tenía palabras de 16 bits y estaba diseñada para el control en tiempo real.



Figura 7: Imagen de una válvula electrónica.

7.3. Computadoras con transistores

En 1948, John Bardeen, Walter Brattain y William Shockley inventan el transistor trabajando en los Laboratorios Bell. El transistor es un dispositivo electrónico semiconductor utilizado para entregar una señal de salida en respuesta a una señal de entrada. Puede cumplir funciones de amplificador, oscilador, interruptor o rectificador. Actualmente se encuentra prácticamente en todos los aparatos electrónicos de uso diario tales como radios, televisores, reproductores de audio y video, relojes de cuarzo, computadoras, lámparas fluorescentes, tomógrafos, teléfonos celulares, aunque casi siempre dentro de los llamados circuitos integrados [1]. Los transistores pueden ser utilizados como interruptores, al igual que las válvulas, pero con mucho menor tamaño, costo y disipación de calor. En la Figura 8(a) se pueden apreciar diferentes tipos de transistores mientras que en la Figura 8(b) se ilustra un esquema de un transistor tipo NPN, dónde de observan las tres partes fundamentales de un transistor: la *base*, el *colector* y el *emisor*.



(a) Imagen de algunos tipos de transistores.

(b) Diagrama de Transistor tipo NPN.

Figura 8: Transistores.

El impacto del transistor ha sido enorme, pues además de iniciar la industria de los semiconductores, ha sido el precursor de otros inventos como son los circuitos integrados, los

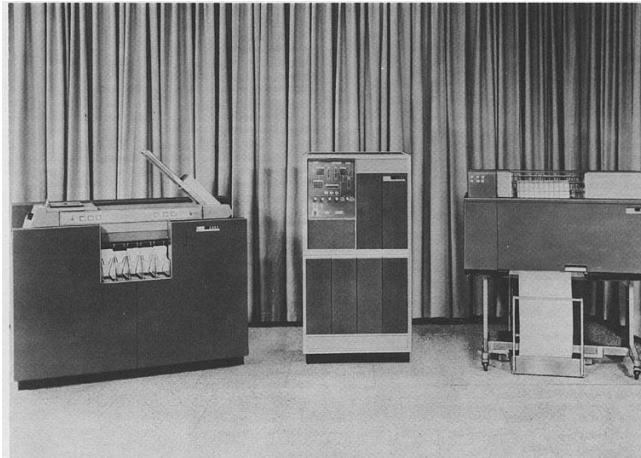


Figura 9: Imagen de la computadora IBM 1401.

dispositivos optoelectrónicos y los microprocesadores. Sin embargo, los cambios más notables han sido en la computación. Según palabras de Albert P. Malvino [6], el transistor no se creó para mejorar la computación sino que él la creó. Antes de 1950, una computadora ocupaba todo un salón y costaba millones de dólares. Luego, a partir del uso del transistor se pudo reducir notablemente el tamaño y el costo. Hoy, una computadora cabe en un bolsillo con un costo muy bajo y enormes prestaciones.

A continuación se enumeran algunos de los avances más notables de la computación utilizando esta tecnología:

- TX-0 (1956). Primera computadora transistorizada, desarrollada en el Lincoln Laboratory del MIT.
- PDP-1 (1960). Primera microcomputadora. 4K de palabras de 18 bits y tiempo de ciclo de $5 \mu s$.
- 1401 (1961). Desarrollada por IBM y orientada a la contabilidad comercial (Ver Figura 9).
- 7094 (1962). Desarrollada por IBM y orientada a la computación científica.
- 6600 (1964). Desarrollada por CDC. Primera supercomputadora científica.

7.4. Computadoras con circuitos integrados

En 1958, Robert Noyce inventó el circuito integrado de silicio, lo cual hizo posible colocar docenas de transistores en un solo chip. Un circuito integrado, también conocido como chip o microchip, es una estructura de pequeñas dimensiones de material semiconductor, normalmente silicio, de algunos milímetros cuadrados de superficie, sobre la que se fabrican circuitos electrónicos generalmente mediante fotolitografía y que está protegida dentro de un encapsulado de plástico o de cerámica. El encapsulado posee conductores metálicos apropiados para hacer conexión entre el circuito integrado y un circuito impreso [8].

La integración de grandes cantidades de pequeños transistores dentro de un pequeño espacio fue un gran avance en la elaboración manual de circuitos utilizando componentes electrónicos discretos. La capacidad de producción masiva de los circuitos integrados, así como la fiabilidad y acercamiento a la construcción de un diagrama a bloques en circuitos, aseguraba la rápida adopción de los circuitos integrados estandarizados en lugar de diseños utilizando transistores discretos.

A continuación se enumeran algunos de los avances más notables de la computación utilizando esta tecnología:

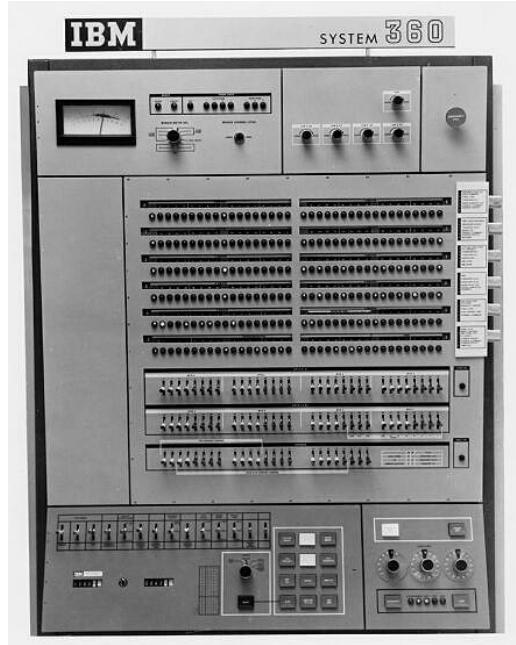


Figura 10: Imagen de la computadora IBM 360.

- System/360 (1964). Desarrollada por IBM como familia de productos tanto para computación científica como comercial. Una importante innovación fue la multiprogramación (Ver Figura 10).
- PDP-8 (1965). Primera minicomputadora con mercado masivo (50000 unidades vendidas).
- PDP-11 (1970). Dominó el mercado de las minicomputadoras en los años setenta.

7.5. Computadoras con integración a muy gran escala

La integración a escala muy grande o VLSI (sigla en inglés de very-large-scale integration) es el proceso de crear un circuito integrado compuesto por cientos de miles de transistores en un único chip. VLSI comenzó a usarse en los años 70, como parte de las tecnologías de semiconductores y comunicación que se estaban desarrollando [9].

- Intel 8080 (1974). Primera computadora de propósito general de 8 bits en un chip. Corría a 2 MHz y se le considera el primer diseño de microprocesador verdaderamente usable. En la Figura 11 se puede apreciar el microprocesador Intel 8080:



Figura 11: Microprocesador Intel 8080.

- Apple II (1977). Primera serie de microcomputadores de producción masiva hecha por la empresa Apple Computer (Ver Figura 12). Arquitectura de 8 bits.
- 8086 (1978). Uno de los primeros chips individuales. Microprocesador de 16 bits. 29 K transistores. El 8088, una variante del 8086 con un bus externo de 8 bits fue utilizado en computadora personal original de IBM.



Figura 12: Computadora Apple II.

- 8087 (1980) Intel introdujo el coprocesador 8087 de punto flotante (45 K transistores) para operar junto a un procesador 8086 o 8088 ejecutando la instrucciones de punto flotante. El 8087 estableció el modelo de punto flotante para la línea x86, a menudo denominado “x87”.
- IBM Personal Computer (1981). Se convirtió en la computadora más vendida de la historia. Venía equipada con el sistema operativo MS-DOS provisto por la compañía Microsoft Corporation:
- 80286 (1982). Agregó más modos de direccionamiento de memoria. Formó la base de la computadora personal IBM PC-AT, la plataforma original para MS Windows. 134 K transistores
- RISC-I (1982). Desarrollada dentro del proyecto RISC en la Universidad de Berkeley bajo la dirección de David A. Patterson. RISC, del inglés Reduced Instruction Set Computer, en español Computador con Conjunto de Instrucciones Reducidas, propone reemplazar arquitecturas complejas (CISC) por otras mucho más sencillas pero más rápidas.
- R2000 (1985). Primer diseño MIPS por John L. Hennessy, el cual mejoró drásticamente el rendimiento mediante el uso de la segmentación.
- i386 (1985). Expidió la arquitectura a 32 bits. 275 K transistores.
- i486 (1989). mejoró el desempeño de la línea x86 e integró la unidad de punto flotante en el chip pero no tuvo cambios significativos en el conjunto de instrucciones. 1.2 M transistores.
- Pentium (1993). Mejoró el desempeño pero solo agregó extensiones menores al conjunto de instrucciones. 3.1 M transistores.
- PentiumPro (1995). Introdujo un diseño de procesador radicalmente nuevo, conocido internamente como la microarquitectura P6. Se agregó una clase de instrucciones de movimiento condicional al conjunto de instrucciones. 5.5 M transistores.

- Pentium II (1997). Continuación of microarquitectura P6. 7 M transistores.
- Pentium III (1999). Introdujo el SSE, una clase of instrucciones para manipular vectores de datos enteros o de punto flotante. 8.2 M transistores pero versiones posteriores llegaron a 24 M transistores.
- Pentium 4 (2000). Extendió SSE a SSE2, agregando nuevos tipos de datos (incluyendo punto flotante de doble precisión), junto con muchas más instrucciones para estos tipos de datos. 42 M transistores
- Pentium 4E (2004). Se agregó *hyperthreading*, un método para ejecutar dos programas simultáneamente en un solo procesador, así como EM64T, la implementación de Intel de una extensión de 64 bits a IA32 desarrollada por Advanced Micro Devices (AMD), a la que nos referimos como x86-64. 125 M transistores.
- Core 2 (2006). Regresó a una microarquitectura similar a P6. Primer microprocesador Intel multinúcleo, donde se implementan múltiples procesadores en un solo chip. No es compatible con *hyperthreading*. 291 M transistores.
- Core i7 (2008). Incorpora tanto *hyperthreading* como *multi-core*, con la versión inicial que admite dos programas en ejecución en cada núcleo y hasta cuatro núcleos en cada chip. 781 M transistores

Referencias

- [1] Richard Blum. *Professional assembly language*. John Wiley & Sons, 2007.
- [2] Randal E Bryant and David Richard O'Hallaron. *Computer systems: a programmer's perspective*, volume 281. Prentice Hall Upper Saddle River, second edition, 2003.
- [3] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.
- [4] Randall Hyde. *The art of assembly language*. No Starch Press, 2003.
- [5] Ed Jorgenson. *X86-64 Assembly Language Programming with Ubuntu*. 2019.
- [6] Albert Paul Malvino and David J. Batesr. *Principios de Electrónica*. Mcgraw-Hill, 2007.
- [7] Andrew S Tanenbaum. *Organización de computadoras: un enfoque estructurado*. Pearson educación, 2000.
- [8] Wikipedia. Circuito integrado — Wikipedia, La enciclopedia libre. https://es.wikipedia.org/w/index.php?title=Circuito_integrado&oldid=122233649, 2019. [Internet; descargado 27-enero-2020].
- [9] Wikipedia. Integración a muy gran escala — Wikipedia, La enciclopedia libre. https://es.wikipedia.org/w/index.php?title=Integraci%C3%B3n_a_muy_gran_escala&oldid=117810308, 2019. [Internet; descargado 27-enero-2020].
- [10] Wikipedia. Válvula termoiónica — Wikipedia, La enciclopedia libre. https://es.wikipedia.org/w/index.php?title=V%C3%A1lvula_termoi%C3%B3nica&oldid=121814534, 2019. [Internet; descargado 27-enero-2020].

- [11] Wikipedia. Transistor — Wikipedia, La enciclopedia libre. <https://es.wikipedia.org/w/index.php?title=Transistor&oldid=122923620>, 2020. [Internet; descargado 27-enero-2020].
- [12] Igor Zhirkov. *Low-Level Programming: C, Assembly, and Program Execution on Intel[®] 64 Architecture*. Apress, 2017.

Representación Computacional de Datos

Diego Feroldi
feroldi@fceia.unr.edu.ar

Arquitectura del Computador
Departamento de Ciencias de la Computación
FCEIA-UNR

Agosto 2019



Índice

1. Introducción	1
2. Organización de los datos	1
2.1. Bits	1
2.2. Nibble	2
2.3. Byte	2
2.4. Palabra	3
3. Sistemas de numeración posicionales	3
4. Sistema binario	4
4.1. Formatos binarios	6
4.2. Conversión entre sistemas	6
4.3. Operaciones elementales en sistema binario	8
4.4. Números con signo	10
4.5. Complementos a la base y a la base menos uno	10
4.5.1. Complemento a uno	13
4.6. Operaciones en complemento a dos	14
4.7. Banderas	16
5. Otras representaciones	18
5.1. Sistema hexadecimal	18
5.1.1. Operaciones en sistema hexadecimal	19
5.2. Representación octal	20
5.3. Representación BCD	21
5.3.1. Suma en formato BCD	22
6. Operaciones para números de precisión arbitraria	22

1. Introducción

La aritmética que utilizan las computadoras difiere de la que estamos acostumbrados a usar por lo cual es importante estudiar primero como se realiza la representación de datos dentro la computadora antes de abordar temas más específicos de programación. La diferencia fundamental radica en que las computadoras solo pueden trabajar con números de precisión finita y, además, esta precisión generalmente es fija. Por otra parte, la mayoría de las computadoras trabajan en sistema binario en lugar del sistema decimal al que estamos habituados a utilizar. Por eso, haremos una revisión de los sistemas de numeración posicionales y, en particular, del sistema binario. Veremos también en este apunte otros dos sistemas de numeración muy útiles en computación: el hexadecimal y el BCD. En primer lugar, veremos como se organizan los datos dentro de la computadora.

2. Organización de los datos

Cuando escribimos números binarios en papel podemos utilizar un número arbitrario de bits. Sin embargo, esto no es posible dentro de la computadora. Las computadoras trabajan con cantidades específicas de bits dependiendo de la máquina en particular (actualmente 64 bits). Veremos a continuación que se suele trabajar con grupos de bits y que estos grupos reciben denominaciones en particular. En primer lugar, como ya se mencionó, un grupo de un solo dígito binario recibe el nombre de *bit*. Por otra parte, un grupo de 4 bits recibe el nombre de *nibble*¹, un grupo de ocho bits se denomina *byte*, un grupo de 16, 32 ó 64 bits se denomina *palabra (word)*, dependiendo de la máquina en cuestión. Análogamente, *Double word* el doble que una palabra y *Quad word* cuatro palabras. Estos tamaños no son arbitrarios. Cuando se diseña una arquitectura de computación, la elección de la longitud de palabra es una cuestión de suma importancia.

2.1. Bits

El bit es la mínima unidad de información en un sistema binario. Dado que solo puede representar dos valores distintos (cero o uno), en principio pareciera que no es de mucha utilidad pero en realidad se puede usar para representar una gran cantidad de cosas: verdadero o falso, encendido o apagado, masculino o femenino, presente o ausente, etc. Todo depende de como

¹Las denominaciones se dan directamente en inglés dado que es la forma usual en que las vamos a encontrar en la bibliografía.

definamos la estructura de datos. Por ejemplo, podemos definir que si el bit es cero representa que algo es falso mientras que si el bit es uno representa que es verdadero. Notar que esto es una convención y que aunque es muy usada tiene notables excepciones: bash y comandos unix.

2.2. Nibble

Un nibble es una colección de cuatro bits. No es una estructura de datos muy interesante excepto en dos casos: números en formato BCD (binary coded decimal) y números en formato hexadecimal. En efecto, con un nibble podemos representar 16 valores distintos. En el caso de números hexadecimales, los valores 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, y F. En formato BCD se usan diez dígitos diferentes (0, 1, 2, 3, 4, 5, 6, 7, 8, 9), por lo tanto también se requiere cuatro bits. Por lo tanto, con un nibble (4 bits) podemos representar un dígito BCD o un dígito hexadecimal.

2.3. Byte

La estructura de datos más utilizada en los microprocesadores 80x86 es el byte. Un byte consiste en ocho bits y es la estructura de datos más pequeña que se puede direccionar en un microprocesador 80x86. En efecto, la memoria principal en 80x86 está direccionada por bytes. Esto significa que el ítem más pequeño al que se puede acceder individualmente mediante un programa 80x86 es un valor de 8 bits. Para acceder a cualquier ítem más pequeño es necesario leer el byte que contiene el dato y “enmascarar”² los bits no deseados.

Los bits en un byte se pueden numerar desde cero hasta siete usando la siguiente convención:

b ₇	b ₆	b ₅	b ₄	b ₃	b ₂	b ₁	b ₀
----------------	----------------	----------------	----------------	----------------	----------------	----------------	----------------

El bit 0 es el bit de menor orden o bit menos significativo mientras que el bit 7 es el bit de mayor orden o bit más significativo.

Notar que un byte contiene exactamente dos nibbles. Los bits del 0...3 comprenden el *low order nibble*, mientras que los bits 4...7 forman el *high order nibble*. Dado que un byte contiene exactamente dos nibbles, un byte requiere dos dígitos hexadecimales para ser representados.

²En informática se denomina máscara al conjunto de datos que junto con una determinada operación permite extraer selectivamente ciertos datos almacenados en otro conjunto.

2.4. Palabra

Una *palabra* es un conjunto de n bits que son manejados como un conjunto por la máquina. Por lo tanto, numeraremos los bits desde el cero hasta el $n - 1$. Análogamente al byte, el bit cero es el menos significativo mientras que el $n - 1$ es el más significativo. Notar que una palabra de 16 bits contiene dos bytes y por tanto 4 nibbles como se observa en la siguiente figura:

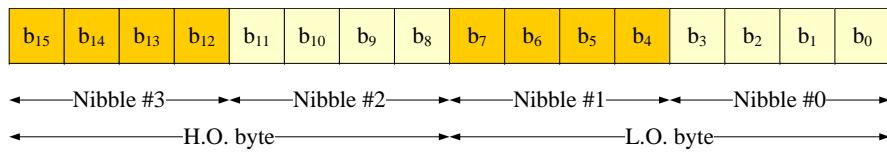


Figura 1: Bytes y nibbles en una palabra de 16 bits.

Con n bits se pueden representar 2^n valores diferentes. Las máquinas más actuales trabajan con palabras de 64 bits. Uno de los principios usos de las palabras consiste en representar valores enteros (*Integers*). Con una palabra de n bits se pueden representar enteros en el rango $[0, 2^n - 1]$ o $[-2^{n-1}, 2^{n-1} - 1]$.

Los valores numéricos sin signo (*Unsigned*) se representan directamente por el valor en binario de los bits. Para representar valores numéricos con signo existen diferentes enfoques. Uno de los más utilizados es el concepto de complemento a dos (ver Sección 4.5).

3. Sistemas de numeración posicionales

Para representar números, lo más habitual es utilizar un sistema posicional de base 10, llamado *sistema decimal*. En este sistema, los números son representados usando diez diferentes caracteres, llamados dígitos decimales: 0, 1, 2, 3, 4, 5, 6, 7, 8 y 9. La magnitud con la que un dígito dado contribuye al valor del número depende de su posición en el número de manera tal que si el dígito a ocupa la posición n a la izquierda del punto decimal (o coma)³, el valor con que contribuye es $a \times 10^{n-1}$, mientras que si ocupa la posición n a la derecha del punto decimal, su contribución es $a \times 10^{-n}$. Por ejemplo, la secuencia de dígitos 123.59 significa

$$123.59 = 1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0 + 5 \times 10^{-1} + 9 \times 10^{-2}.$$

³En este apunte utilizaremos el punto como símbolo para indicar la separación entre la parte entera y la parte fraccional.

En general, la representación decimal

$$(-1)^s(a_n a_{n-1} \dots a_1 a_0.a_{-1} a_{-2} \dots)$$

corresponde al número

$$(-1)^s(a_n 10^n + a_{n-1} 10^{n-1} + \dots + a_1 10^1 + a_0 10^0 + a_{-1} 10^{-1} + a_{-2} 10^{-2} + \dots),$$

donde s depende del signo del número ($s = 0$ si el número es positivo y $s = 1$ si es negativo). De manera análoga, se pueden concebir otros sistemas posicionales con una base distinta de 10.

En principio, cualquier número natural $\beta \geq 2$ puede ser utilizado como base. Entonces, fijada una base, todo número real admite una *representación posicional* en la base β de la forma

$$(-1)^s(a_n \beta^n + a_{n-1} \beta^{n-1} + \dots + a_1 \beta^1 + a_0 \beta^0 + a_{-1} \beta^{-1} + a_{-2} \beta^{-2} + \dots),$$

donde los coeficientes a_i son los dígitos en el sistema con base β , esto es, enteros positivos tales que $0 \leq a_i \leq \beta - 1$. Los coeficientes a_i con $i \geq 0$ se consideran como los dígitos de la parte entera, en tanto que los a_i con $i < 0$, son los dígitos de la parte fraccionaria. Si, como en el caso decimal, utilizamos un punto para separar tales partes, el número es representado en la base β como

$$(-1)^s(a_n a_{n-1} \dots a_1 a_0.a_{-1} a_{-2} \dots)_{\beta},$$

donde hemos utilizado el subíndice β para evitar cualquier ambigüedad con la base escogida. En efecto, a lo largo de este apunte utilizaremos un subíndice en cada número para indicar cual es su base y por lo tanto en qué sistema está representado. Por ejemplo:

- $(1001)_2$, sistema binario
- $(159.23)_{10}$, sistema decimal
- $(0A45)_{16}$, sistema hexadecimal

4. Sistema binario

Una de las grandes ventajas de los sistemas posicionales es que se pueden dar reglas generales simples para las operaciones aritméticas. Además, tales reglas resultan más simples cuanto más pequeña es la base. Esta observación nos lleva a considerar el sistema de base $\beta = 2$, o sistema binario, en donde sólo tenemos los dígitos 0 y 1. Pero existe otra importante razón. Una

computadora, en su nivel más básico, solo puede registrar si fluye o no electricidad por cierta parte de un circuito. Estos dos estados pueden representar entonces dos dígitos, convencionalmente, 1 cuando hay flujo de electricidad y 0 cuando no lo hay. Con una serie de circuitos apropiados una computadora puede entonces contar (y realizar operaciones aritméticas) en el sistema binario. El sistema binario consta, pues, solo de los dígitos 0 y 1, llamados *bits* (del inglés *binary digits*). El 1 y el 0 en notación binaria tienen el mismo significado que en notación decimal:

$$0_2 = 0_{10}, \quad 1_2 = 1_{10}.$$

Se puede lograr una equivalencia entre sistemas de representación. Así, por ejemplo, 1101.01 es la representación binaria del número 13.25 del sistema decimal, esto es, $(1101.01)_2 = (13.25)_{10}$, puesto que,

$$1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} = 13.25$$

Además del sistema binario, otros dos sistemas posicionales resultan de interés en el ámbito computacional, a saber, el sistema con base $\beta = 8$, denominado *sistema octal*, y el sistema con base $\beta = 16$, denominado *sistema hexadecimal*. El sistema octal usa dígitos del 0 al 7, en tanto que el sistema hexadecimal usa los dígitos del 0 al 9 y las letras A, B, C, D, E, F. Siguiendo la misma regla se pueden hallar las siguientes equivalencias:

$$(13.25)_{10} = (1101.01)_2 = (15.2)_8 = (D.4)_{16}$$

El sistema hexadecimal se desarrollará en detalle en la Sección 5.1

La gran mayoría de las computadoras actuales (y efectivamente todas las computadoras personales, o PC) utilizan internamente el sistema binario ($\beta = 2$). Las calculadoras, por su parte, utilizan el sistema decimal ($\beta = 10$). Ahora bien, cualquiera sea la base β escogida, todo dispositivo de cálculo solo puede almacenar un número finito de dígitos para representar un número. En particular, en una computadora solo se puede disponer de un cierto número finito fijo n de posiciones de memoria para la representación de un número. El valor de n se conoce como **longitud de palabra**. Además, aun cuando en el sistema binario un número puede representarse tan sólo con los dígitos 1 y 0, el signo “-” y el punto, la representación interna en la computadora no tiene la posibilidad de disponer de los símbolos signo y punto. De este modo una de tales posiciones debe ser reservada de algún modo para indicar el signo y cierta distinción debe hacerse para representar la parte entera y fraccionaria. Esto puede hacerse de distintas formas. En el apunte *Representación de Números Reales* veremos la representación de punto flotante. En este apunte solamente utilizaremos la representación en punto fijo.

4.1. Formatos binarios

Un número en formato binario se puede representar de diferentes maneras ya que cualquier bit cero a la izquierda del uno más significativo no tiene peso en el número. Por ejemplo, el número 6 se puede representar como $(110)_2$, como $(0110)_2$, como $(00000110)_2$, etc. Sin embargo, es usual utilizar una notación donde los ceros a la izquierda no se escriben. Otra convención usual, derivada de las máquinas 80x86, es trabajar con grupos de cuatro u ocho bits. Además, para mayor claridad, es usual separar los grupos de 4 bits con un espacio en blanco, análogamente a la práctica en decimal de poner un punto cada tres dígitos. Por ejemplo, el número seis lo podemos representar en binario como $(0000\ 0110)_2$.

Al momento de trabajar con números binarios, podemos necesitar referirnos a un bit en particular del número. Para ello le asignaremos un valor a cada posición del bit. De esta manera en números binarios con ocho bits tendremos desde el bit cero hasta el bit siete:

$$b_7 b_6 b_5 b_4 \ b_3 b_2 b_1 b_0$$

El bit más a la derecha en un número binario es el bit con posición cero. Este bit se denomina bit menos significativo (LSB, *Least Significant Bit*). Cada bit a la izquierda va teniendo una posición mayor. El bit más a la derecha se denomina bit más significativo (MSB, *Most Significant Bit*). Análogamente, se procede para los decimales desde b_{-1} :

$$b_7 b_6 b_5 b_4 \ b_3 b_2 b_1 b_0.b_{-1} b_{-2} b_{-3} b_{-4} \ b_{-5} b_{-6} b_{-7} b_{-8}$$

En este caso el bit menos significativo es el b_{-8} .

4.2. Conversión entre sistemas

Binario a decimal

La conversión de binario a decimal es directa empleando la definición de sistema de numeración posicional ya vista:

$$(b_3 b_2 b_1 b_0.b_{-1} b_{-2} b_{-3} b_{-4})_2 = (b_3 \times 2^3 + b_2 \times 2^2 + b_1 \times 2^1 + b_0 \times 2^0 + b_{-1} \times 2^{-1} + b_{-2} \times 2^{-2} + b_{-3} \times 2^{-3} + b_{-4} \times 2^{-4})_{10}$$

Por ejemplo:

$$(0110.1101)_2 = 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-4} = 6.8125$$

Decimal a binario

Para convertir un número de decimal a binario el primer paso es separar la parte *entera* de la parte *fraccionaria*. Para convertir la parte entera, un método de conversión consiste en realizar sucesivas divisiones por dos hasta llegar a cero e ir registrando los restos que resultan ser los bits del número en binario. El primer resto obtenido es b_0 , el segundo es b_1 y así sucesivamente.

Por otro lado, para convertir la parte fraccionaria se realizan multiplicaciones sucesivas por dos de las partes fraccionarias (sin el entero) y se van registrando los dígitos enteros obtenidos. El primer dígito obtenido es b_{-1} y así sucesivamente.

Por ejemplo, para convertir el número $(149.56)_{10}$ a binario primero convertimos la parte entera dividiendo sucesivamente por dos hasta que el cociente entero sea 0 y registrando el valor de los restos, tal cual se muestra en el siguiente procedimiento:

$$\begin{array}{ll} 149/2=74 & \text{Resto}=1 \rightarrow b_0=1 \\ 74/2=37 & \text{Resto}=0 \rightarrow b_1=0 \\ 37/2=18 & \text{Resto}=1 \rightarrow b_2=1 \\ 18/2=9 & \text{Resto}=0 \rightarrow b_3=0 \\ 9/2=4 & \text{Resto}=1 \rightarrow b_4=1 \\ 4/2=2 & \text{Resto}=0 \rightarrow b_5=0 \\ 2/2=1 & \text{Resto}=0 \rightarrow b_6=0 \\ 1/2=0 & \text{Resto}=1 \rightarrow b_7=1 \end{array}$$

Notar que el último resto se considera 1 aunque en realidad es menor que 1. Entonces, $(149)_{10} = (1001\ 0101)_2$.

Luego procedemos con la parte fraccionaria:

$$\begin{array}{ll} 0.56 \times 2 = 1.12 & \rightarrow b_{-1}=1 \\ 0.12 \times 2 = 0.24 & \rightarrow b_{-2}=0 \\ 0.24 \times 2 = 0.48 & \rightarrow b_{-3}=0 \\ 0.48 \times 2 = 0.96 & \rightarrow b_{-4}=0 \\ 0.96 \times 2 = 1.92 & \rightarrow b_{-5}=1 \\ 0.92 \times 2 = 1.84 & \rightarrow b_{-6}=1 \\ 0.84 \times 2 = 1.68 & \rightarrow b_{-7}=1 \\ 0.68 \times 2 = 1.36 & \rightarrow b_{-8}=1 \end{array}$$

De esta manera, $(0.56)_{10} = (1000\ 1111)_2$ empleando 8 bits. Por lo tanto, uniendo ambos resultados obtenemos

$$(149.56)_{10} = (1001\ 0101.1000\ 1111)_2$$

En realidad se podría haber seguido operando, obteniéndose más dígitos binarios. Al haber truncado el número para utilizar solo 8 bits para la parte fraccionaria se ha cometido un error por truncamiento. Se puede ver que en realidad $(1001\ 0101.1000\ 1111)_2 = 149.5586$, con lo cual el error absoluto de representación es $\Delta p = |p - p^*| = 0.0014$ y el error relativo es $\frac{|p - p^*|}{|p|} = 9.4026 \times 10^{-6}$, donde p es el verdadero valor y p^* es la aproximación del mismo, siempre que $p \neq 0$. Si se hubieran empleado más bits para la parte fraccionaria el error hubiera sido menor.

Se puede definir una *cota superior* de error en función del número de dígitos empleados para representar la parte fraccionaria, es decir:

$$\frac{|p - p^*|}{|p|} < \frac{1}{2}\beta^{-t},$$

donde t es la cantidad de dígitos empleados para la parte fraccionaria y β es la base empleada.

4.3. Operaciones elementales en sistema binario

Suma

En binario, la tabla de adición toma la siguiente forma:

$$\begin{array}{rcl} 0 + 0 & = & 0 \\ 0 + 1 & = & 1 \\ 1 + 0 & = & 1 \\ 1 + 1 & = & 10 \end{array}$$

Note que al sumar $1 + 1$ es $(10)_2$, es decir, llevamos 1 a la siguiente posición de la izquierda (acarreo). En el sistema decimal esto es equivalente a sumar, por ejemplo, $8 + 7$ que resulta 15. Por lo tanto, el resultado es un 5 en la posición que estamos sumando y un 1 de acarreo en la siguiente posición a la izquierda.

Ejemplo, sumar 0101_2 y 0011_2 :

$$\begin{array}{r} 1 \ 1 \ 1 \\ 0 \ 1 \ 0 \ 1 \\ + \ 0 \ 0 \ 1 \ 1 \\ \hline 1 \ 0 \ 0 \ 0 \end{array}$$

En sistema decimal sería $5 + 3 = 8$.

Resta

La tabla de resta toma la siguiente forma:

$$\begin{array}{r} 0 - 0 = 0 \\ 1 - 0 = 1 \\ 1 - 1 = 0 \\ 0 - 1 = \mathbf{1} \end{array}$$

La resta $0 - 1$ se resuelve igual que en el sistema decimal, tomando una unidad prestada de la posición siguiente: $0 - 1 = 1$ y *me llevo* 1.

Ejemplo, restar 0010_2 a 1001_2 :

$$\begin{array}{r} & \mathbf{1} & \mathbf{1} \\ & 1 & 0 & 0 & 1 \\ - & 0 & 0 & 1 & 0 \\ \hline & 0 & 1 & 1 & 1 \end{array}$$

En sistema decimal sería $9 - 2 = 7$. Otra forma mucho más práctica de realizar las restas es utilizar el *complemento a dos* o el *complemento a uno*. Este tema se desarrolla en la Sección [4.5](#).

Multiplicación

La tabla de multiplicación binaria toma la siguiente forma:

$$\begin{array}{r} 0 \times 0 = 0 \\ 0 \times 1 = 0 \\ 1 \times 0 = 0 \\ 1 \times 1 = 1 \end{array}$$

La multiplicación de números en binario es muy sencilla dado que cero por cualquier número es cero y el uno es el elemento neutro de la multiplicación.

Ejemplo, multiplicar 1100_2 por 0110_2 :

$$\begin{array}{r} & 1 & 1 & 0 & 0 \\ \times & 0 & 1 & 1 & 0 \\ \hline & 0 & 0 & 0 & 0 \\ & \mathbf{1} & 1 & 1 & 0 & 0 \\ & \mathbf{1} & 1 & 1 & 0 & 0 \\ & 0 & 0 & 0 & 0 & \\ \hline & 1 & 0 & 0 & 1 & 0 & 0 & 0 \end{array}$$

En sistema decimal sería $12 \times 6 = 12$.

4.4. Números con signo

Hasta el momento hemos tratado con números sin signo donde todos los números son positivos. Obviamente, también necesitamos representar números negativos. Lo primero que se puede pensar para representar números negativos es emplear una analogía con el sistema decimal donde el número negativo se escribe como un número positivo precedido del signo “-”. Este enfoque se conoce como formato *magnitud y signo*.

Para representar el número dentro de la computadora se emplea un bit, generalmente el más significativo, para representar el signo mientras que el resto de los bits indican la magnitud (valor absoluto) del número a representar. Por convención, el bit más significativo en uno indica un número negativo.

Por ejemplo, para representar el número $(-28)_{10}$ primero se convierte el número $(28)_{10}$ a binario, esto es $(0001\ 1100)_2$ y luego se indica que es negativo haciendo uno el bit más significativo: $(-28)_{10} = (1001\ 1100)_2$.

Sin embargo, este enfoque tiene dos grandes desventajas. En primer lugar, tiene doble representación del cero ya que tanto $(1000\ 0000)_2$ como $(0000\ 0000)_2$ representan el cero, lo que complica una operación muy usual: la comparación con cero. Por otra parte, las operaciones aritméticas son más complejas ya que, por ejemplo, para realizar una suma primero hay que determinar si los dos números tienen el mismo signo y en caso afirmativo realizar la suma de la parte significativa. En caso contrario, restar el mayor del menor y asignar el signo del mayor (en valor absoluto). Por lo tanto, este enfoque casi no fue usado y veremos a continuación enfoques más prácticos basados en el *complemento del número*.

4.5. Complementos a la base y a la base menos uno

Los complementos son muy usados en los sistemas digitales para representar números negativos y, por lo tanto, al momento de realizar operaciones de resta. Existen dos tipos de complementos. El complemento a la base (β) y el complemento a la base menos uno ($\beta - 1$). Es decir, para los números binarios (donde la base es 2) existen los complementos a 2 y a 1. En base octal serían complemento a 8 y a 7. En el sistema decimal, complemento a 10 y a 9, etc.

Complemento a la base

Sea un número N representado con n dígitos, el complemento a la base β de N se define como

$$C_{\beta}^N = \begin{cases} 0 & \text{si } N = 0, \\ \beta^n - |N| & \text{si } N \neq 0. \end{cases}$$

Esto se cumple para todo N incluso con fracción decimal. El único caso especial a considerar es cuando la parte entera es cero. Esto se interpreta como que $n = 0$.

Ejemplos en base 10:

- Complemento a 10 de $(987)_{10}$. En este caso $N = 987$ y $n = 3$, entonces $C_{10}^N = 10^3 - 987 = 1000 - 987 = (13)_{10}$
- Complemento a 10 de $(0.125)_{10}$. En este caso $N = 0.125$ y $n = 0$, entonces $C_{10}^N = 10^0 - 0.125 = 1 - 0.125 = (0.875)_{10}$
- Complemento a 10 de $(987.125)_{10}$. En este caso $N = 987.125$ y $n = 3$, por lo tanto $C_{10}^N = 10^3 - 987.125 = 1000 - 987.125 = (12.875)_{10}$

Es importante notar que NO es lo mismo calcular el complemento de la parte entera y de la fracción decimal por separado y juntar los resultados.

Complemento a la base menos uno

Por otro lado, tenemos también el complemento a la base menos uno: $\beta - 1$. En este caso, teniendo un número positivo N en base β con n dígitos enteros y m dígitos en la fracción decimal, se define el complemento a $\beta - 1$ de N como

$$C_{\beta-1}^N = \beta^n - \beta^{-m} - N.$$

Por ejemplo, para el complemento a 9 de $(987)_{10}$ tenemos que $N = 987$, $n = 3$ y $m = 0$, por lo tanto:

$$C_9^N = 10^3 - 10^0 - 987 = 1000 - 1 - 987 = (12)_{10}$$

Para el complemento a 9 de $(0.125)_{10}$ tenemos que $N = 0.125$, $n = 0$ y $m = 3$, entonces:

$$C_9^N = 1 - 10^{-3} - 0.125 = 0.999 - 0.125 = (0.874)_{10}$$

Para calcular el complemento a 9 de $(987.125)_{10}$, $N = 987.125$, $n = 3$ y $m = 3$, por lo tanto:

$$C_9^N = 10^3 - 10^{-3} - 987.125 = 1000 - 0.001 - 987.125 = (12.874)_{10}.$$

Observar que en este caso SÍ es lo mismo calcular el complemento de la parte entera y el de la fracción decimal por separado y juntar o sumar los resultados.

Complemento a dos

El complemento a dos es un sistema posicional de representación para números enteros en el que los positivos se representan en binario natural y los negativos se representan restando su magnitud de la cantidad 2^n , siendo n el número de dígitos:

$$C_2^N = \begin{cases} N & \text{si } N \geq 0, \\ 2^n - |N| & \text{si } N < 0. \end{cases}$$

Ejemplos:

- Para el complemento a 2 de $N = (1010\ 1100)_2$ tenemos que $n = 8$, entonces:

$$C_2^N = (1\ 0000\ 0000)_2 - (1010\ 1100)_2 = (0101\ 0100)_2$$

También podemos hacer el cálculo pasando al sistema decimal como paso intermedio:

$$C_2^N = (2^8)_{10} - (172)_{10} = (256)_{10} - (172)_{10} = (84)_{10} = (0101\ 0100)_2$$

- Análogamente, el complemento a 2 de $(1010)_2$ es $(1\ 0000 - 1010)_2 = (0110)_2$

Complemento a dos: método alternativo

El cálculo del complemento a dos puede realizarse de manera muy sencilla mediante un método alternativo. En efecto, se puede ver que para calcular el complemento a 2 de un número binario sólo basta con revisar todos los dígitos desde el menos significativo hacia el más significativo y mientras se consiga un cero dejarlo igual. Al conseguir el primer número 1, dejarlo igual para luego cambiar el resto de ellos hasta llegar al más significativo. Así podemos decir rápidamente que el complemento a 2 de $(1010\ 0000)_2$ es $(0110\ 0000)_2$, que el complemento a 2 de $(111)_2$ es $(001)_2$, etc.

Otra forma muy sencilla de hallar el complemento a 2 de un número binario es invirtiendo todos los dígitos (que como veremos a continuación

es lo que se conoce como complemento a 1) y sumándole uno al resultado obtenido, esto es $C_2^N = C_1^N + 1$. Estos métodos son muy fáciles de realizar mediante puertas lógicas, donde reside su mayor utilidad.

El rango de valores decimales para n bits será:

$$-2^{n-1} \leq Rango \leq 2^{n-1} - 1$$

El total de números positivos (incluyendo el cero) será 2^{n-1} y el de negativos 2^{n-1} . Por ejemplo, con cuatro bits el rango representable es $-8 \leq Rango \leq 7$.

4.5.1. Complemento a uno

De acuerdo a la definición de complemento a la base menos uno, el complemento a uno de un número N con n bits se define como:

$$C_1^N = 2^n - N - 1$$

Ejemplos: Para el complemento a 1 de $N = (1010\ 1100)_2$ sabemos que $n = 8$ y $m = 0$, entonces:

$$C_1^N = (1\ 0000\ 0000 - 1 - 1010\ 1100)_2 = (0101\ 0011)_2.$$

Análogamente, el complemento a 1 de $N = (1010)_2$ es

$$C_1^N = (1\ 0000 - 1 - 1010)_2 = (0101)_2$$

En estos dos últimos ejemplos se puede observar que para conseguir el complemento a 1 de un número binario tan solo basta con invertir todos los dígitos (esto quiere decir cambiar 0 por 1 y viceversa). Este método es muy simple y es que habitualmente se realiza.

El rango de valores decimales para n bits será:

$$-2^{n-1} + 1 \leq Rango \leq 2^{n-1} - 1$$

El total de números positivos (incluyendo el cero) será 2^{n-1} y el de negativos $2^{n-1} - 1$. Por ejemplo, con cuatro bits el rango representable es $-7 \leq Rango \leq 7$.

En el Cuadro 1 se encuentra el complemento a dos y el complemento a uno con enteros de 4 bits. Observar que el complemento a uno tiene dos representaciones posibles del cero, lo cual es una desventaja.

Cuadro 1: Complemento a dos con enteros de 4 bits

Decimal	Complemento a dos	Complemento a uno
7	0111	0111
6	0110	0110
5	0101	0101
4	0100	0100
3	0011	0011
2	0010	0010
1	0001	0001
0	0000	0000
0	0000	1111
-1	1111	1110
-2	1110	1101
-3	1101	1100
-4	1100	1011
-5	1011	1010
-6	1010	1001
-7	1001	1000
-8	1000	

4.6. Operaciones en complemento a dos

En primer lugar vamos a analizar como operar con números en complemento a la base de manera genérica y luego vamos a ver de manera particular el complemento a la base dos dada su gran utilidad.

Sean dos números A y B en base β . Si deseamos obtener la suma $S = A + B$, analizaremos los distintos casos posibles según sea el signo de cada uno de los números.

1. $A > 0$ y $B > 0$. En este caso la suma $S = A + B$ es correcta y el resultado es positivo ya que ambos números lo son.
2. $A < 0$ y $B < 0$. En este caso podemos escribir la suma como $S = (\beta^n - |A|) + (\beta^n - |B|) = \beta^n + \beta^n - (|A| + |B|)$, siendo la suma correcta si se ignora β^n . El resultado es negativo ya que ambos números lo son y queda expresado en el complemento correspondiente.
3. $A < 0$ y $B > 0$ y $|A| < |B|$. En este caso podemos escribir la suma como $S = (\beta^n - |A|) + B = \beta^n + (|B| - |A|)$, siendo la suma correcta si se ignora β^n . El resultado es positivo por la relación propuesta para los valores absolutos.

4. $A > 0$ y $B < 0$ y $|A| < |B|$. En este caso la suma se puede escribir como $S = A + (\beta^n - |B|) = \beta^n - (|B| - |A|)$, siendo la suma correcta pues el resultado es negativo y está expresado en el complemento correspondiente.

La conclusión es que cuando sumamos números positivos y negativos utilizando complemento a la base se obtendrá siempre el resultado correcto. Solo hay que ignorar β^n (acarreo) en los casos 2 y 3. Por lo tanto, la suma implementada con complemento a la base es muy simple, motivo por el cual es muy empleada. A continuación, veremos de forma particular el caso con base $\beta = 2$ mediante ejemplos.

1. $A = 5$, $B = 12$, $S = A + B = 17$. En binario:

$$\begin{array}{r} 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 0 \ 1 \\ + \ 0 \ 0 \ 0 \ 0 \ 1 \ 1 \ 0 \ 0 \\ \hline 0 \ 0 \ 0 \ 1 \ 0 \ 0 \ 0 \ 1 \end{array}$$

y acarreo $c = 0$.

2. $A = -5$, $B = -12$, $S = A + B = -17$. En binario:

$$\begin{array}{r} 1 \ 1 \ 1 \ 1 \ 1 \ 0 \ 1 \ 1 \\ + \ 1 \ 1 \ 1 \ 1 \ 0 \ 1 \ 0 \ 0 \\ \hline 1 \ 1 \ 1 \ 0 \ 1 \ 1 \ 1 \ 1 \end{array}$$

y acarreo $c = 1$.

3. $A = -5$, $B = 12$, $S = A + B = 7$. En binario:

$$\begin{array}{r} 1 \ 1 \ 1 \ 1 \ 1 \ 0 \ 1 \ 1 \\ + \ 0 \ 0 \ 0 \ 0 \ 1 \ 1 \ 0 \ 0 \\ \hline 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 1 \ 1 \end{array}$$

y acarreo $c = 1$.

4. $A = 5$, $B = -12$, $S = A + B = -7$. En binario:

$$\begin{array}{r} 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 0 \ 1 \\ + \ 1 \ 1 \ 1 \ 1 \ 0 \ 1 \ 0 \ 0 \\ \hline 1 \ 1 \ 1 \ 1 \ 1 \ 0 \ 0 \ 1 \end{array}$$

y acarreo $c = 0$.

En todos los casos el resultado es correcto, ignorando el acarreo.

4.7. Banderas

Al momento de operar con valores en representación computacional es importante analizar el estado de las banderas en el registro `rflags`⁴, en particular la *carry flag* (CF) y la *overflow flag* (OF). Es importante tenerlas en cuenta (y no confundirlas!) dado que la ALU (*Arithmetic Logic Unit*)⁵ no tiene en cuenta si el programador está trabajando con matemática con signo (*Signed*) o sin signo (*Unsigned*). La ALU simplemente realiza la operación binaria bit a bit y setea las banderas apropiadamente. Depende del programador chequear el valor de las banderas una vez realizada la operación.

Si el programa trata los bits como números sin signo, debe verificarse si se ha seteado en “1” la bandera de *carry*, indicando que el resultado es erróneo. En este contexto, el valor de la bandera *overflow* es irrelevante. Por el contrario, si el programa trata a los bits como valores con signo en complemento a dos, lo que hay que verificar es la bandera *overflow* en lugar de la bandera *carry*. Veamos con ejemplos el uso de cada una de estas dos banderas.

Carry flag

Las reglas para “encender” la bandera *carry* son dos:

- Si la suma de dos números en binario entero causa un acarreo en el bit más significativo, la bandera *carry* se enciende.

Ejemplo:

$$\begin{array}{r} 1111 \\ +0001 \\ \hline 0000 \\ \text{CF=1} \end{array}$$

El resultado es incorrecto dado que $15 + 1 \neq 0$.

- Si la resta de dos números requiere un acarreo en el bit más significativo, la bandera *carry* se enciende.

Ejemplo:

⁴El registro `rflags` se estudiará en detalle en el apunte correspondiente a *Assembler X86-64*.

⁵La ALU es un circuito digital en el microprocesador que calcula operaciones aritméticas y operaciones lógicas, entre valores de los argumentos.

$$\begin{array}{r}
 0000 \\
 -0001 \\
 \hline
 1111 \\
 \text{CF=1}
 \end{array}$$

El resultado es incorrecto dado que $0 - 1 \neq 15$.

En conclusión, si se está realizando aritmética sin signo, la bandera *carry* encendida significa que el resultado es incorrecto, dado que el verdadero resultado es demasiado grande en valor absoluto para poder ser representado con el número de bits disponible.

Overflow flag

Las reglas para “encender” la bandera *overflow* son dos:

- Si la suma de dos números con el bit más significativo en cero resulta en un número con el bit de signo en uno, la bandera *overflow* se enciende.

$$\begin{array}{r}
 0100 \\
 +0100 \\
 \hline
 1000 \\
 \text{OF=1}
 \end{array}$$

El resultado es incorrecto dado que $4 + 4 \neq -8$.

- Si la suma de dos números con el bit más significativo en uno resulta en un número con el bit de signo en cero, la bandera *overflow* se enciende.

$$\begin{array}{r}
 1000 \\
 +1000 \\
 \hline
 0000 \\
 \text{OF=1}
 \end{array}$$

El resultado es incorrecto dado que $-8 + (-8) \neq 0$.

Si se está realizando aritmética en complemento a dos, la bandera *overflow* encendida significa que el resultado es incorrecto. Es decir, se están sumando dos números positivos y el resultado es negativo o bien se están sumando dos números negativos y el resultado es positivo. En resumen, si se está operando con números *signed*, la bandera *overflow* indica si el resultado es correcto o no.

5. Otras representaciones

5.1. Sistema hexadecimal

Un problema importante del sistema binario es su “verbosidad”, es decir, el considerable número de dígitos necesarios para representar un determinado valor. Por ejemplo, para representar $(158)_{10} = (1001\ 1110)_2$ son necesarios ocho cifras en lugar de las tres necesarias en el sistema decimal. Por lo tanto, una primera conclusión es que la representación en el sistema decimal es más compacta aunque sería mucho más complejo implementar una computadora cuyo microprocesador trabaje directamente con sistema decimal. Por otra parte, aunque se puede realizar la conversión entre decimal y binario y viceversa, no es una tarea trivial (ver Sección 4.2).

El sistema hexadecimal (base 16) resuelve este problema de manera mucho más eficiente. En efecto, el sistema hexadecimal tiene las dos ventajas que necesitamos: la notación es muy compacta y es muy simple convertir entre hexadecimal y binario, y viceversa. Por estos motivos, en el ámbito de la computación se utiliza mucho el sistema de numeración hexadecimal.

Dado que en hexadecimal la base es 16, cada dígito a la izquierda del punto representa el valor de ese dígito multiplicado por una potencia de 16 dependiendo de la posición del dígito. Cada dígito hexadecimal representa un valor dentro de un conjunto de 16 valores distintos entre 0 y 15_{10} . Dado que existen solo 10 dígitos decimales, se necesitan 6 dígitos adicionales para representar el rango entre 10_{10} y 15_{10} . Los símbolos que se utilizan son las letras del abecedario entre la A y la F. Así se llega a la tabla de equivalencia entre sistemas mostrada en el Cuadro 2. De esta manera, el número $(15E)_{16}$ representa a

$$(15E)_{16} = 1 \times 16^2 + 5 \times 16^1 + 14 \times 16^0 = (350)_{10},$$

dado que $(E)_{16} = (14)_{10}$.

Como se puede observar el sistema hexadecimal es muy compacto y fácil de leer. Además la conversión entre hexadecimal y binario es muy fácil. El Cuadro 2 contiene toda la información necesaria. Para convertir de hexadecimal a binario simplemente hay que sustituir cada dígito hexadecimal por sus correspondientes cuatro bits. Por ejemplo, para convertir el número $(0AF3)_{16}$ a binario:

HEXADECIMAL	0	A	F	3
BINARIO	0000	1010	1111	0011

Para convertir un número de binario a hexadecimal es casi tan fácil. El primer paso es llenar el número con ceros para asegurarse que el número de bits es múltiplo de cuatro. Luego, separar el número en grupos de cuatro bits

Cuadro 2: Equivalencia entre diferentes sistemas de numeración.

Hexadecimal	Octal	Decimal	Binario	BDC
0	0	0	0000	0000 0000
1	1	1	0001	0000 0001
2	2	2	0010	0000 0010
3	3	3	0011	0000 0011
4	4	4	0100	0000 0100
5	5	5	0101	0000 0101
6	6	6	0110	0000 0110
7	7	7	0111	0000 0111
8	10	8	1000	0000 1000
9	11	9	1001	0000 1001
A	12	10	1010	0001 0000
B	13	11	1011	0001 0001
C	14	12	1100	0001 0010
D	15	13	1101	0001 0011
E	16	14	1110	0001 0100
F	17	15	1111	0001 0101

y convertir cada grupo utilizando en Cuadro 2. Por ejemplo, para convertir el número binario $(0001\ 0011\ 0010\ 1110)_2$ en hexadecimal:

BINARIO	0001	0011	0010	1110
HEXADECIMAL	1	3	2	E

5.1.1. Operaciones en sistema hexadecimal

A la hora de realizar operaciones en sistema hexadecimal se puede optar por hacer un cambio de sistema como paso intermedio o bien operar directamente en hexadecimal. Veamos la suma y la resta.

Suma

La suma se realiza de manera análoga a lo visto teniendo en cuenta el acarreo correspondiente. Ejemplo:

$$\begin{array}{r}
 & & 1 \\
 & 1 & A & E \\
 + & & 1 & 8 \\
 \hline
 1 & C & 6
 \end{array}$$

Resta

Para realizar la resta, podemos proceder restando dígito a dígito y teniendo en cuenta los acarreos. Ejemplo:

$$\begin{array}{r} & & 1 \\ & 1 & A & 6 \\ - & & 1 & C \\ \hline 1 & 8 & A \end{array}$$

Un manera más simple es calcular el complemento a la base (C_{16}^N) del sustraendo y luego sumarlo al minuendo. Previamente, hay que igualar la cantidad de dígitos. En el ejemplo anterior primero se calcularía el complemento a 16 de $N=01C$. Para ello recordar la relación entre el complemento a la base y el complemento a la base menos uno, dado que es más simple calcular C_{15}^N :

$$\begin{aligned} C_{16}^N &= C_{15}^N + 1 \\ C_{16}^N &= FE3 + 1 = FE4 \end{aligned}$$

Entonces,

$$\begin{array}{r} 1 & A & 6 \\ + & F & E & 4 \\ \hline 1 & 1 & 8 & A \end{array}$$

El resultado es correcto ignorando el primer uno.

5.2. Representación octal

De manera totalmente análoga se puede trabajar con otras bases. Por ejemplo, otro sistema de representación muy utilizado es el **sistema octal**. Este sistema numérico tiene base 8, por lo cual utiliza 8 ocho cifras para la representación. Las cifras utilizadas son los número enteros del 0 al 7. En el Cuadro 2 se puede ver la equivalencia entre sistemas. La conversión entre los mismos se puede realizar de manera análoga a lo ya visto lo visto. Tener en cuenta que para representar los 8 dígitos en sistema octal son necesarios 3 dígitos binarios. Ejemplo:

OCTAL	1	5	1	4
BINARIO	001	101	001	100

Cuadro 3: Representación de dígitos BCD

Decimal	BCD
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

5.3. Representación BCD

El sistema Decimal Codificado en Binario (en inglés *Binary-Coded Decimal* (BCD)) es un estándar para representar números decimales en el sistema binario en donde cada dígito decimal es codificado con una secuencia de 4 bits. Con esta codificación especial de los dígitos decimales en el sistema binario se pueden realizar operaciones aritméticas como suma, resta, multiplicación y división de números en representación decimal sin perder en los cálculos la precisión que normalmente ocurre con las conversiones de decimal a binario “puro” y viceversa. La conversión de los números decimales a BCD y viceversa es muy sencilla, pero los cálculos en BCD demandan más tiempo y tienen más complejidad que los realizados con números binarios puros. En primer lugar veamos la tabla de codificación de los dígitos BCD (Cuadro 3):

Entonces para convertir un número decimal a formato BCD reemplazamos cada dígito por su correspondiente equivalente en binario. Por ejemplo, para convertir el número $(138)_{10}$ en BCD reemplazamos cada uno de sus dígitos por su equivalente en BCD:

DECIMAL	1	3	8
BCD	0001	0011	1000

Entonces, $(138)_{10} = (0001\ 0011\ 1000)_{BCD}$. Notar que el número obtenido es diferente al que obtendríamos en formato binario “puro”, el cual sería $(1000\ 1010)_2$. Es más, notar que en formato BCD son necesarios más bits para representar el mismo número.

5.3.1. Suma en formato BCD

Se pueden sumar dos números BCD utilizando las reglas de la suma binaria vistas anteriormente. Sin embargo, si una suma de 4 bits es mayor que 9 el resultado no es válido. En este caso, se debe sumar 6, $(0110)_2$, al grupo de 4 bits para saltar así los seis estados no válidos. Si se genera un acarreo al sumar 6, éste se suma al grupo de 4 bits siguiente. Ejemplo: $8 + 7 = 15$

$$\begin{array}{r} 1 \ 0 \ 0 \ 0 \\ + \ 0 \ 1 \ 1 \ 1 \\ \hline 1 \ 1 \ 1 \ 1 \end{array}$$

El resultado es correcto en binario pero 1111 no es un número válido en BCD porque es mayor que 9. Entonces sumamos 6 a este resultado:

$$\begin{array}{r} 1 \ 1 \ 1 \ 1 \\ + \ 0 \ 1 \ 1 \ 0 \\ \hline 1 \ 0 \ 1 \ 0 \ 1 \end{array}$$

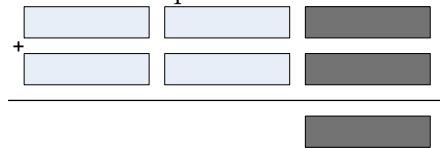
Esto se interpreta como 0001 0101, es decir 15 en BCD. Con este simple ejemplo se puede notar la mayor complejidad de las operaciones en BCD. Sin embargo, el formato BCD es muy común en sistemas electrónicos donde las operaciones se realizan directamente mediante sistemas digitales sin programación.

6. Operaciones para números de precisión arbitraria

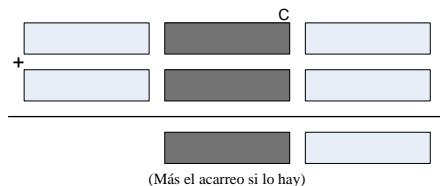
La aritmética con precisión arbitraria no se utiliza como aritmética nativa de procesadores, sino sólo en bibliotecas especializadas. La precisión máxima está limitada por los parámetros de la biblioteca, la memoria disponible y el tiempo de procesador disponible. Veamos cómo se procede. La instrucción `add` del 80x86 suma dos números de 8, 16, o 32 bits. Luego de la ejecución de la instrucción, la *carry flag* queda establecida en uno si se produce un *overflow* en el bit MSB. Esta información se puede usar para realizar operaciones con precisión arbitraria de manera secuencial análogamente a como procedemos cuando sumamos números manualmente.

Por ejemplo, si queremos realizar $125 + 158$ primero sumamos $5 + 8 = 3$ y tenemos un acarreo de uno. Luego hacemos $2 + 5 + 1 = 8$ y así sucesivamente. El 80x86 procede de manera análoga excepto que en lugar de dígitos son palabras. En los pasos mostrados en la Figura 2 se indica cómo se procedería para sumar dos números de 48 bits representados cada uno con tres palabras.

Paso 1: Sumar las palabras de menor orden.

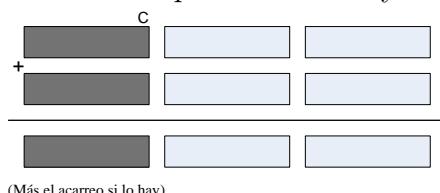


Paso 2: Sumar las palabras intermedias.



(Más el acarreo si lo hay)

Paso 3: Sumar las palabras de mayor orden.



(Más el acarreo si lo hay)

Figura 2: Ejemplo de suma con precisión arbitraria

Referencias

- [1] Andrew S. Tanenbaum , *Organización de computadoras: Un enfoque estructurado*, cuarta edición, Pearson Education, 2000
- [2] Paul A. Carter, *PC Assembly Language*, Disponible en formato electrónico: <http://www.drpaulcarter.com/pcasm/>, 2006.
- [3] M. Morris Mano, *Computer system architecture*, tercera edición, Prentice-Hall, 1993.
- [4] Randall Hyde, *The art of assembly language*, segunda edición, No Starch Pr, 2003.

Manejo de Bits en Lenguaje C

Licenciatura en Ciencias de la Computación-FCEIA-UNR
Arquitectura del Computador
Dr. Diego Feroldi

Agosto de 2019

1. Introducción

Como hemos visto en los apuntes anteriores, la computadora internamente trabaja en formato binario. Por lo tanto, la manipulación de bits es algo muy usual. En efecto, las operaciones de bits (u operaciones bit a bit) son muy comunes en Assembler dado que es un programa de bajo nivel. El lenguaje C no provee un tipo de datos específico para trabajar con bits. Sin embargo, podemos operar a nivel bits trabajando con datos de tipo entero considerando su representación interna.

Por ejemplo, si queremos expresar el valor $(01000001)_2$ podemos declarar una variable A como `char` y luego escribir en C:

- `A=65` (decimal)
- `A=0x41` (hexadecimal)
- `A=0101` (octal)
- `A=0b01000001` (binario)
- `A='A'` (carácter)

Estas expresiones son equivalentes dado que la representación interna es la misma, es decir es la misma secuencia de bits.

Como ya hemos visto, la forma más conveniente es utilizar hexadecimal dado que es una representación compacta y además la conversión binario-hexadecimal es fácil. Recordar que para realizar dicha conversión basta con tomar grupos de cuatro bits y realizar la conversión individual de cada uno de esos grupos.

2. Operadores de bits en C

Veamos primero en la siguiente tabla cuáles son los principales operadores de bits (*bitwise operators*) en lenguaje C:

Operador	Acción
<code>&</code>	Operación AND
<code> </code>	Operación OR
<code>^</code>	Operación XOR
<code>~</code>	Complemento a uno
<code>>></code>	Desplazamiento a la derecha
<code><<</code>	Desplazamiento a la izquierda

Importante: Los operadores de bits pueden ser aplicados a variables de tipo entero ya sea con signo o sin signo (`char`, `short`, `int`, `long`, etc.) y NO pueden ser aplicados a variables tipo flotante (`float`, `double`, etc.).

2.1. Operador AND (&)

La operación AND esta definida por la siguiente tabla de verdad:

A	B	A&B
0	0	0
0	1	0
1	0	0
1	1	1

Supongamos que tenemos dos variables, `a=56` y `b=72`, entonces la operación AND entre estas dos variables resulta:

$$\begin{array}{r}
 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\
 \& 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
 \hline
 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0
 \end{array}$$

Es decir, `a&b=8`.

Notar que el resultado hubiera sido diferente si se hubiera aplicado el operador lógico `&&`. En este caso habría resultado `a&&b=1`. El operador AND lógico (`&&`) devuelve el valor booleano *true* si ambos operandos son *true*. En caso contrario, devuelve *false*. Los operandos se convierten implícitamente al tipo *bool* antes de su evaluación y el resultado de la operación es de tipo *bool*.

Una aplicación útil para el operador `&` consiste en determinar si un determinado bit de cierto número es 1 o 0.

Ejemplo 1. Si se tiene el número `a=72` y se quiere averiguar si el cuarto bit de dicho número es 1 o 0, podemos aplicar el operador `&` realizando la operación `a&b`, donde `b` es un número cuya representación binaria es `00001000`, es decir un número cuyo cuarto bit es 1 y todos los demás son ceros. A este último número se lo denomina “máscara”¹. Entonces:

$$\begin{array}{r}
 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
 \& 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
 \hline
 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0
 \end{array}$$

Por lo tanto, al realizar `a&b` vemos que efectivamente el cuarto bit del número es 1. Ya veremos que este ejemplo se puede mejorar utilizando el operador desplazamiento.

2.2. Operador OR (|)

La operación OR esta definida por la siguiente tabla de verdad:

¹Las “máscaras” son secuencias de bits que combinadas en una operación binaria con cualquier otra secuencia de bits permite modificar esta última u obtener alguna información sobre ella.

A	B	A B
0	0	0
0	1	1
1	0	1
1	1	1

Supongamos que tenemos dos variables, $a=56$ y $b=72$, entonces la operación OR entre estas dos variables resulta:

$$\begin{array}{cccccccc}
 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\
 | & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\
 \hline
 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0
 \end{array}$$

Es decir, $a|b=120$. Notar que el resultado es distinto al obtenido si se utiliza el operador OR lógico ($\mid\mid$).

El operador $|$ se puede utilizar para “encender” determinados bits de un número.

Ejemplo 2. Sea el número $a=28$, si queremos poner en uno el séptimo bit podemos realizar $a=a|b$, donde $b=01000000$:

$$\begin{array}{cccccccc}
 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\
 | & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
 \hline
 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0
 \end{array}$$

Podemos observar que el séptimo bit de a ahora es uno.

2.3. Operador XOR (\wedge)

La operación XOR está definida por la siguiente tabla de verdad:

A	B	$A \wedge B$
0	0	0
0	1	1
1	0	1
1	1	0

Supongamos que tenemos dos variables, $a=56$ y $b=72$, entonces la operación XOR entre estas dos variables resulta:

$$\begin{array}{cccccccc}
 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\
 \wedge & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\
 \hline
 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0
 \end{array}$$

Es decir, $a\wedge b=112$.

El operador \wedge se puede utilizar para invertir determinados bits de un número.

Ejemplo 3. Sea el número $a=28$, si queremos invertir el cuarto bit podemos realizar $a=a\wedge b$, donde $b=00001000$:

$$\begin{array}{cccccccc}
 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\
 \wedge & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
 \hline
 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0
 \end{array}$$

Vemos que el cuarto bit de a pasó de ser uno a ser cero.

2.3.1. Propiedades del operador XOR

A partir de la tabla de la verdad del operador XOR se pueden determinar las siguientes propiedades:

- Comutativa: $A \wedge B = B \wedge A$
- Asociativa: $A \wedge (B \wedge C) = (A \wedge B) \wedge C$
- $A \wedge A = 0$
- $A \wedge 0 = A$

A partir de las propiedades anteriores resulta:

- $(B \wedge A) \wedge A = B \wedge 0 = B$

Esto resulta útil para hacer cifrado de datos. Una cadena de texto puede ser cifrada aplicando el operador de bit XOR sobre cada uno de los caracteres utilizando una clave. Para descifrar la salida, solo hay que volver a aplicar el operador XOR con la misma clave.

Ejemplo 4. La cadena de caracteres “Hola” se puede representar utilizando código ASCII como 01001000 01101111 01101100 01100001. Esta cadena puede ser cifrada con la clave 11110011 de la siguiente manera:

$$\begin{array}{r} 01001000 \ 01101111 \ 01101100 \ 01100001 \\ \wedge \ 11110011 \ 11110011 \ 11110011 \ 11110011 \\ \hline 10111011 \ 10011100 \ 10011111 \ 10010010 \end{array}$$

Si a este resultado se le vuelve a aplicar el operador XOR con la misma clave volvemos a tener la cadena original:

$$\begin{array}{r} 10111011 \ 10011100 \ 10011111 \ 10010010 \\ \wedge \ 11110011 \ 11110011 \ 11110011 \ 11110011 \\ \hline 01001000 \ 01101111 \ 01101100 \ 01100001 \end{array}$$

2.4. Operador complemento a uno (\sim)

El operador binario \sim se conoce como operador complemento a uno o también como operador NOT. Es un operador unario, es decir solo necesita un operando. La tabla de la verdad de este operador es la siguiente:

A	$\sim A$
0	1
1	0

Por lo tanto, si tenemos la variable `a=56`, resulta `~a=-57` dado que la secuencia de bits que representa a la variable `a` es 00111000 y entonces la secuencia que resulta de aplicarle el operador \sim resulta 11000111.

Ejemplo 5. La expresión `x=x&~0xff` pone los últimos 8 bits de `x` en cero. Notar que esto es independiente de la longitud del tipo de dato a diferencia de la expresión `x=x&0xffffffff00` que asume que `x` tiene 32 bits.

2.5. Operadores desplazamiento de bits (\ll , \gg)

Se pueden realizar corrimientos de N bits, con $N \in \mathbb{N}$, es decir $N = 1, 2, \dots$. Los desplazamientos de bits pueden ser en dos direcciones: desplazamiento hacia la derecha o desplazamiento hacia la izquierda. Cada vez que se hace un desplazamiento se completa con ceros en el otro lado (no se trata de una rotación).

Supongamos que tenemos el número $a=56$ ($56_{10} = 00111000_2$) y le realizamos un corrimiento de un bit hacia la derecha: $a \gg 1$. El resultado es $00011100_2 = 28_{10}$. Es decir, se ha dividido al número a por dos. Por lo tanto, desplazando el número n bits hacia la derecha se realiza la división entera del número:

$$a \gg n = a/2^n$$

Análogamente, el desplazamiento a la izquierda es equivalente a multiplicar por potencias de dos:

$$a \ll n = a \times 2^n \quad (1)$$

La ventaja de los desplazamientos es que son menos costosos que hacer las operaciones de multiplicación y división.

Importante: Hay que tener cuidado al realizar desplazamientos hacia la izquierda dado que se obtienen resultados erróneos (con respecto a la ecuación 1) debido al *overflow*.

También hay que tener precauciones al desplazar hacia la derecha:

- En los desplazamientos hacia la derecha de valores sin signo siempre se completan los bits vacíos con ceros.
- En los desplazamientos hacia la derecha de valores con signo se completa de acuerdo al bit de signo (“*arithmetic shift*”) en algunas máquinas y con ceros (“*logical shift*”) en otras.

Ejemplo 6. En el Ejemplo 1 vimos cómo determinar si un bit determinado de un número es 0 o 1. Dicho ejemplo se puede mejorar realizando un desplazamiento hacia la derecha luego de aplicar el operador &:

```
(a&b)>>3
```

El resultado es 1, indicando que el cuarto bit es efectivamente 1.

3. Operador ternario (? :)

El operador ternario² se puede interpretar desde el punto de vista de un *if/else*. El operador ternario funciona con tres expresiones: $E1$, $E2$ y $E3$. Por lo tanto, el código $E1?E2:E3$ resulta equivalente al siguiente código:

```
if (E1)
    E2
else
    E3
```

²Si bien el operador ternario no es un operador bit a bit, lo introducimos en este apunte dado que es útil al trabajar con operadores de bits como se puede ver en el Ejemplo 1.

Ejemplo 1. Determinar si un número es par o impar.

```
#include <stdio.h>

int main()
{
    char a;
    printf ("Ingrese un número \n" );
    scanf ("%d",&a);
    (a&1)?printf("El número es impar \n" ):printf("El número es par \n");
    return 0;
}
```

4. Campos de bits

Cuando el espacio de almacenamiento es escaso, puede ser necesario empaquetar varios objetos en una sola palabra de máquina. El método que se utiliza en C para operar con campos de bits, está basado en las estructuras (*structs*). La forma general de definición de un campo de bits es la siguiente:

```
typedef struct bits
{
    <tipo de dato> nombre1 : <cantidad de bits>;
    <tipo de dato> nombre2 : <cantidad de bits>;
    . . .
    <tipo de dato> nombre3 : <cantidad de bits >;
}campo_bits ;
```

Cada campo de bits puede declararse como `char`, `unsigned char`, `int`, `unsigned int`, etc. y su longitud puede variar entre 1 y el máximo número de bits disponible de acuerdo a la arquitectura del microprocesador en que se esté trabajando.

Ejemplo 1. Uso de campos de bits.

```
#include <stdio.h>

typedef struct s_bits
{
    unsigned int mostrar: 1;
    unsigned int rojo: 8;
    unsigned int azul: 8;
    unsigned int verde: 8;
    unsigned int transparencia: 1;
}campo_bits;

int main()
{
    campo_bits unColor;
    /* Se crea un color verde */
    unColor.rojo = 51;
```

```
unColor.azul = 55;
unColor.verde = 255;
unColor.transparencia = 1;

/* Se verifica si el color es transparente */
if (unColor.transparencia == 1)
{
    printf("El color es transparente\n");
}
return 0;
}
```

Referencias

- [KR17] Brian Kernighan and Dennis M Ritchie. *The C programming language*. Prentice Hall, 2017.
- [War13] Henry S Warren. *Hacker's delight*. Pearson Education, 2013.

Representación Computacional de Números Reales

Diego Feroldi

feroldi@fceia.unr.edu.ar

Arquitectura del Computador

Departamento de Ciencias de la Computación
FCEIA-UNR

Octubre de 2018



Índice

1. Introducción	1
2. Representación de números reales con punto fijo	1
3. Representación de números reales con punto flotante	2
3.1. Notación científica normalizada	2
3.2. Redondeo de un número real	4
3.3. Representación computacional	5
4. Standard IEEE 754 para números en punto flotante	7
4.1. Exponente sesgado	8
4.2. Significante	9
4.3. Conversión de decimal a IEEE 754 simple precisión	9
4.4. Conversión de IEEE 754 simple precisión a decimal	10
4.5. Características principales	11
4.6. Números denormalizados	11
4.7. Ceros	13
4.8. Infinitos	13
4.9. Formato NaN	14
5. Operaciones de números en punto flotante	14
5.1. Suma o resta	14
5.2. Multiplicación	15
5.3. División	16
6. Densidad de los números en punto flotante	16
7. Precauciones al operar con números en punto flotante	17
A. Cálculo del epsilon de máquina	18

1. Introducción

Para representar números reales es necesario utilizar una coma o punto para separar la parte entera de la parte fraccionaria, independientemente del sistema de representación con que se trabaje (decimal, binario, etc.). Existen dos formas de resolver el problema:

- Se considera la coma o punto en cierta posición fija.
- Se almacena la posición que la coma o punto ocupa en el número (posición flotante).

2. Representación de números reales con punto fijo

Este método considera que el punto está siempre en una misma posición. En los sistemas digitales se utilizan registros para almacenar los datos y entonces se adoptan dos posiciones posibles para el punto:

1. En el extremo izquierdo con lo cual el valor almacenado en el registro representa la parte fraccionaria del número.
2. En el extremo derecho del registro con lo cual el valor almacenado representa la parte entera del número.

En ninguno de los dos casos el punto existe realmente, pero su presencia se supone porque el número almacenado en el registro se trata como una fracción o un entero.

Ejemplo:

Trabajando con 8 bits de los cuales hemos fijado y reservado 5 para la parte entera y 3 para la fraccionaria:

$$(11011.011)_2 = 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3} = (27.375)_{10}$$

En este ejemplo se puede observar que existe una limitación en cuanto al rango de los números que se pueden representar:

- Menor número representable: $(00000.001)_2 = 2^{-3} = (0.125)_{10}$
- Mayor número representable: $(11111.111)_2 = 2^5 - 2^{-3} = (31.875)_{10}$

En forma general:

$$2^{-m} \leq \text{Rango} \leq 2^n - 2^{-m}$$

donde n es la cantidad de bits de la parte entera y m es la cantidad de bits de la parte fraccionaria.

Ventajas: La aritmética de punto fijo es relativamente simple.

Desventajas: El rango de representación, es decir el número de cantidades a representar, es muy limitado.

3. Representación de números reales con punto flotante

La representación anterior tiene importantes limitaciones. En efecto, viendo el ejemplo anterior se deduce que números muy grandes y fracciones muy pequeñas no pueden ser representadas. En números decimales esta limitación puede superarse utilizando la notación científica que permite representar números muy grandes o muy pequeños utilizando relativamente pocos dígitos.

3.1. Notación científica normalizada

Para la representación de números reales sobre un amplio rango de valores con menos dígitos se emplea la notación científica. Por ejemplo, el número 9760000000000000 se puede representar como 0.976×10^{15} mientras que el número 0.00000000000000976 se puede representar como 0.976×10^{-15} . En esta notación la coma o punto decimal se mueve dinámicamente a una posición conveniente y se utiliza el exponente en base 10 para registrar la posición del punto decimal.

Sin embargo, observemos que este tipo de notación tiene cierta ambigüedad dado que un mismo número puede ser representado de diferentes maneras. En el ejemplo anterior, el número 9760000000000000 también puede escribirse como 9.76×10^{14} , etc. Por lo tanto es necesario definir una normalización.

Definición

Todo número real no nulo se puede escribir en forma única en la **notación científica normalizada** como:

$$(-1)^s 0.a_1 a_2 a_3 \dots a_t \dots \times 10^e$$

siendo el dígito $a_1 \neq 0$. Por lo tanto, en el ejemplo anterior la forma 0.976×10^{-15} es normalizada.

De forma general, todo número real no nulo puede representarse en forma única respecto a la base β de la siguiente forma:

$$(-1)^s 0.a_1 a_2 a_3 \dots a_t \dots \times \beta^e,$$

donde los “dígitos” a_i son enteros positivos tales que $1 \leq a_1 \leq \beta - 1$, $0 \leq a_i \leq \beta - 1$ para $i = 2, 3, \dots$ y constituyen la parte fraccional o **mantisa** del número, en tanto que e es el **exponente**, el cual indica la posición del punto correspondiente a la base β .

Si m es la fracción decimal correspondiente a $(0.a_1 a_2 a_3 \dots)_\beta$, es decir $m = a_1 \times \beta^{-1} + a_2 \times \beta^{-2} + a_3 \times \beta^{-3} + \dots$, entonces el número representado corresponde al número decimal $(-1)^s \times m \times \beta^e$, siendo $\beta^{-1} \leq m < 1$.

Ejemplo

En este ejemplo, un número N se representa en sistema binario (base $\beta = 2$) con un bit para el signo (s), una mantisa fraccionaria de 8 bits (m) y un exponente con signo de 6 bits (e). El punto o coma fijo de la fracción se encuentra inmediatamente después del bit de signo, pero no está representada realmente en el registro.

Número decimal	Número binario	Signo	Mantisa	Exponente
+3.3750	+11.011	1	00011011	000101

Por lo tanto:

$$N = (-1)^s \times m \times \beta^e = (-1)^0 \times (.00011011)_2 \times 2^{+5}$$

Sin embargo, este número no está normalizado. Un número con punto flotante está normalizado si el dígito más significativo de la mantisa es distinto de cero. Por lo tanto, normalizando tenemos:

$$N = (-1)^0 \times (.11011000)_2 \times 2^{+2}$$

Mantisa (normalizada)	Exponente (normalizado)
11011000	000010

Nota: Al realizar la normalización se debe tener en cuenta la corrección del exponente.

Rango de los números representables

Es importante notar que en todo dispositivo el número de dígitos posibles para representar la mantisa es finito y el exponente también varía en un rango finito:

$$L \leq e \leq U, \quad \text{con } L < 0 \text{ y } U > 0.$$

Además, la mantisa también puede representar un número acotado de valores distintos. Por lo tanto, esto implica que solo un rango finito de números puede ser representado. En el ejemplo anterior tenemos $-31 \leq e \leq 32$ y $2^{-1} \leq m \leq (1 - 2^{-8})$. Por lo tanto:

$$2^{-1}2^{-31} \leq |N| \leq (1 - 2^{-8})2^{32}.$$

Propiedades

Sea el conjunto de números de punto flotante $\mathbb{F}(\beta, t, L, U)$, donde β es la base, t es la cantidad de dígitos significativos de la mantisa, L es valor mínimo del exponente y U es el valor máximo del exponente:

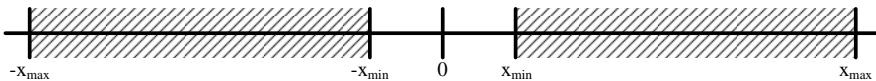
- El número de elementos del conjunto \mathbb{F} es $2(\beta - 1)\beta^{t-1}(U - L + 1) + 1$ dado que existen dos posibles elecciones del signo, $\beta - 1$ elecciones posibles para el dígito principal de la mantisa, β elecciones para cada uno de los restantes dígitos de la mantisa, $U - L + 1$ posibles valores para el exponente y un valor para representar el cero.
- El cero no puede ser representado como punto flotante normalizado y se representa como caso particular.
- Si $x \in \mathbb{F}$, entonces su opuesto $-x \in \mathbb{F}$.

4. El conjunto está acotado tanto superior como inferiormente:

$$x_{\min} = \beta^L \beta^{-1} \leq |x| \leq x_{\max} = \beta^U (1 - \beta^{-t})$$

5. Hay cinco regiones excluidas para los números del conjunto \mathbb{F} :

- Los números negativos menores que $-x_{\max}$ (“overflow” negativo).
- Los números negativos mayores que $-x_{\min}$ (“underflow” negativo).
- El cero.
- Los números positivos menores que x_{\min} (“underflow” positivo).
- Los números positivos mayores que x_{\max} (“overflow” positivo).



6. Los números en punto flotante no están igualmente espaciados sobre la recta real, si no que están más próximos cerca del origen y más espaciados a medida que nos alejamos del origen (ver Sección 6).

7. Una cantidad de gran importancia es el denominado **Epsilon de máquina**:

$$\epsilon_m = \beta^{1-t}, \quad (1)$$

la cual representa la distancia entre el número 1 y el número en punto flotante siguiente más próximo: $1 \oplus \epsilon_m > 1$, donde el operador \oplus indica una suma en el sistema de números en flotante.

8. Para evitar la proliferación de diversos sistemas de punto flotante se desarrolló la norma IEEE 754. Como se verá en la Sección 4, esta norma define los siguientes formatos $(\mathbb{F}(\beta, t, L, U))$:

- $\mathbb{F}(2, 24, -126, 127)$ precisión simple de 32 bits (`float` de C).
- $\mathbb{F}(2, 53, -1022, 1023)$ precisión doble de 64 bits (`double` de C).
- $\mathbb{F}(2, 64, -16382, 16383)$ precisión extendida de 80 bits (`long double` de C).
- $\mathbb{F}(2, 113, -16382, 16383)$ precisión extendida de 128 bits (`_float128` de C).

3.2. Redondeo de un número real

Dado que sólo los valores del conjunto \mathbb{F} pueden ser representados, entonces los números reales deben ser aproximados a un valor perteneciente al conjunto, al cual denotaremos $fl(x)$. La manera usual de proceder consiste en aplicar el redondeo simétrico a t dígitos a la mantisa de representación de punto flotante (de longitud infinita) de x . Esto es, a partir de

$$x = (-1)^s 0.a_1 a_2 \dots a_t a_{t+1} \dots \times \beta^e.$$

Si el exponente está en el rango $L \leq e \leq U$ obtenemos $fl(x)$ como

$$fl(x) = (-1)^s 0.a_1 a_2 \dots \tilde{a}_t \times \beta^e$$

con

$$\tilde{a}_t = \begin{cases} a_t & \text{si } a_{t+1} < \beta/2 \\ a_t + 1 & \text{si } a_{t+1} \geq \beta/2 \end{cases}.$$

El error que resulta se denomina error de redondeo. Todo número real x dentro del rango de los números de punto flotante puede ser representado con un error relativo $\delta_r(x)$:

$$\delta_r(x) = \frac{|x - fl(x)|}{|x|} \leq \frac{1}{2} \beta^{1-t}.$$

Aquí se observa la importancia del Epsilon de máquina, dado que recordando la expresión (1) podemos expresar el error relativo cometido por redondeo de la siguiente manera:

$$\delta_r(x) = \frac{|x - fl(x)|}{|x|} \leq \frac{1}{2} \epsilon_m.$$

Ejemplos:

Si queremos redondear utilizando 5 dígitos, $t = 5$. Por lo tanto, los errores resultan:

$$\begin{aligned} x = (0.4567689010\dots)_10 \times 10^3 &\rightarrow fl(x) = (0.45677)_10 \times 10^3 \rightarrow \delta_r(x) \leq 0.5 \times 10^{-4} \\ x = (0.110101010\dots)_2 \times 2^4 &\rightarrow fl(x) = (0.11011)_2 \times 2^4 \rightarrow \delta_r(x) \leq 0.5 \times 2^{-4} \end{aligned}$$

3.3. Representación computacional

Para poder utilizar los números en punto flotante en sistemas de cómputo, sus valores de exponente y mantisa deben ser almacenados en registros. Las principales características de la representación de números en punto flotante son las siguientes:

- Mantisa normalizada, convención magnitud y signo, asumiendo que la misma es una fracción (el punto está implícito).
- Al exponente se le asigna un tipo de representación sesgada o polarizada. Se le suma un valor fijo o sesgo (o *bias*) para obtener un valor final que es siempre positivo aunque el valor que representa puede ser negativo (este tema se verá en detalle).
- La base del sistema con que se trabaja se asume y no se representa.

Ejemplo:

En este ejemplo se representan números en punto flotante utilizando 24 bits con la siguiente distribución:

S: Signo

S	Exponente con sesgo	Mantisa
1 bit	7 bits	16 bits

7 bits en el exponente implica 128 valores distintos en el rango $[0, 127]$. Esto rango puede asociarse a un rango $[-63, 64]$ utilizando un sesgo de 63 para poder contar con exponentes negativos y positivos. Por lo tanto, dada la siguiente representación no normalizada:

$$\underbrace{0}_{s} \underbrace{1010100}_{exponente} \underbrace{00000000000011011}_{mantisa}$$

Signo: $s = 0$

Exponente: $E = (1010100)_2 = 84 \rightarrow e = E - \text{sesgo} = 84 - 63 = 21$

Mantisa: $00000000000011011 \rightarrow m = 1 \times 2^{-12} + 1 \times 2^{-13} + 1 \times 2^{-15} + 1 \times 2^{-16}$

Por lo tanto:

$$\underbrace{0}_{s} \underbrace{1010100}_{exponente} \underbrace{00000000000011011}_{mantisa} = (-1)^s \times 2^e \times m = 864$$

Para normalizar la representación se debe mover la fracción 11 lugares hacia la izquierda y por lo tanto restar 11 al exponente:

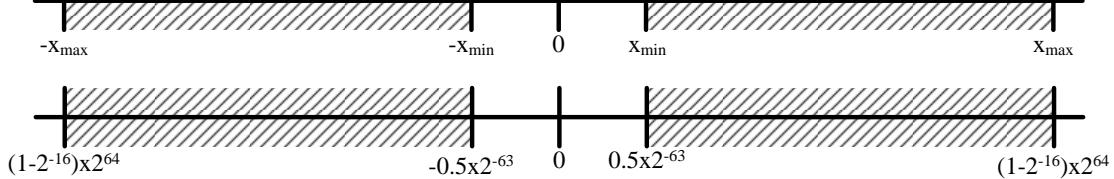
- Signo: $s = 0$
- Exponente: $e = 21 - 11 = 10 \rightarrow$ se almacena $E = e + \text{sesgo} = 10 + 63 = 73 = (01001001)_2$
- Mantisa: $(.1101100000000000)_2 \rightarrow m = 1 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-4} + 1 \times 2^{-5} = 0.84375$

Por lo tanto:

$$\underbrace{0}_{s} \underbrace{1001001}_{exponente} \underbrace{1101100000000000}_{mantisa} = 2^{10}(1 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-4} + 1 \times 2^{-5}) = 864$$

Analizando el ejemplo anterior se pueden realizar las siguientes observaciones:

- Con 24 bits se pueden representar 2^{24} números distintos
- Se pueden representar los siguientes rangos:
 - Número negativos entre $-(1 - 2^{-16}) \times 2^{64}$ y -0.5×2^{-63} .
 - Números positivos entre 0.5×2^{-63} y $(1 - 2^{-16}) \times 2^{64}$



Por lo tanto, cuando una magnitud es demasiado pequeña para ser representada se produce un desbordamiento a cero mientras que cuando una operación resulta en un número con exponente mayor que 64 se produce un desbordamiento (“*overflow*”).

Es importante notar que si bien con 24 bits se pueden representar 2^{24} valores distintos, estos valores están divididos en dos intervalos (uno positivo y uno negativo). Además, al contrario que en punto fijo estos números no están distribuidos en forma equiespaciada. Si se incrementa el número de bits del exponente (a expensas de la cantidad de bits en la mantisa) aumenta el rango pero decrece la precisión dado que la cantidad de números representables depende de la cantidad de bits totales. Por lo tanto, como se ampliará más adelante, existe un compromiso entre rango y precisión. Este tema se desarrollará en detalle en la Sección 6. Notar además que el cero tiene muchas representaciones posibles (mantisa nula con distintos exponentes) o ninguna representación si exigimos normalización, dado que en este caso la mantisa será siempre no nula. Por lo tanto, es necesario contar con una normalización.

4. Standard IEEE 754 para números en punto flotante

El Instituto de Ingenieros Eléctricos y Electrónicos (IEEE) creó un comité para formular una norma en 1985 (Standard IEEE 754) que define los siguientes formatos estándar para números en punto flotante:

- Precisión simple con 32 bits y sesgo 127 (**float** de C):

1	8	23
S	Exponente con sesgo	Fracción
$b_{31} b_{30}$	$b_{23} b_{22}$	b_0

- Precisión doble con 64 bits y sesgo 1023 (**double** de C):

1	11	52
S	Exponente con sesgo	Fracción
$b_{63} b_{62}$	$b_{52} b_{51}$	b_0

- Precisión doble extendida con 80 bits y sesgo 16383 (**long double** de C):

1	15	1	63
S	Exponente con sesgo	Parte entera	Fracción
$b_{79} b_{78}$		$b_{64} \uparrow b_{62}$ b_{63}	b_0

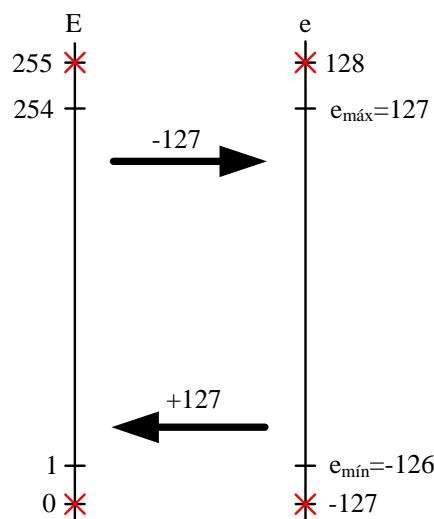
Notar que a diferencia los formatos simple y doble precisión, en este formato no se utiliza un bit implícito. Por el contrario, el bit 63 contiene la parte entera del significante y los bits 0-62 contienen la parte fraccional. El bit 63 será uno en todos los números normalizados.

- Precisión cuádruple con 128 bits y sesgo 16383 (**_float128** de C):

1	15		112	
S	Exponente con sesgo		Fracción	
b_{127}	$b_{126} \dots b_1 b_0$		$b_{112} \dots b_{111}$	

4.1. Exponente sesgado

Los valores mínimos (0) y máximo del exponente (255, 2047 y 32767, según el formato) no se utilizan para número normalizados sino que tienen usos especiales como se verá más adelante. Además, se utilizada una representación sesgada para el exponente. Es decir, al exponente (e) se le suma un sesgo o “bias” para poder representar exponentes negativos. De esta manera, el exponente codificado (E) está desplazado con respecto al verdadero exponente. La siguiente figura esquematiza el concepto de representación de números con exponente sesgado según norma IEEE 754 simple precisión (Sesgo=-127):



Por lo tanto, los valores mínimos y máximos de exponente realmente efectivos resultan:

- Precisión simple (sesgo 127):
 - $e_{min} = 1 - 127 = -126$
 - $e_{max} = 254 - 127 = 127$
- Precisión doble (sesgo 1023):
 - $e_{min} = 1 - 1023 = -1022$
 - $e_{max} = 2046 - 1023 = 1023$
- Precisión extendida y cuádruple (sesgo 16383):
 - $e_{min} = 1 - 16383 = -16382$
 - $e_{max} = 32766 - 16383 = 16383$

4.2. Significante

Como se mencionó en la sección anterior, una fracción normalizada binaria comienza siempre con un punto binario seguido por un bit igual a 1 y luego el resto de los bits de la representación. Entonces, ese bit igual a 1 no necesita ser almacenado ya que puede asumirse su presencia. Por lo tanto el estándar IEEE 754 define esta fracción de una manera distinta a la usual y para evitar confusiones la llama **significante**. Este significante consta de un 1 implícito seguido del punto y luego $t = p - 1$ bits, donde t es la cantidad de bits en el significante incluyendo el “1” implícito (Ver Figura 1). Si todos los bits son cero el significante vale 1.0, mientras que si todos son uno el valor del mismo es ligeramente inferior a 2. Notar que esta forma de normalización tampoco permite la representación del cero.

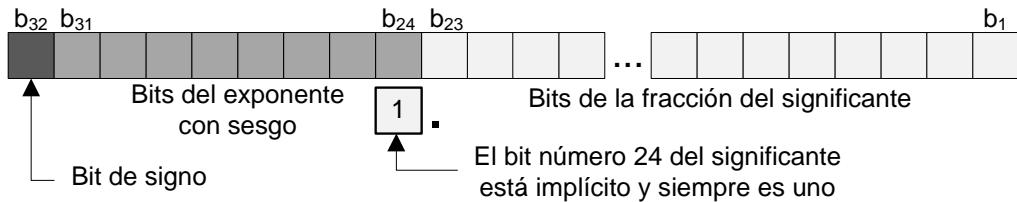


Figura 1: Formato IEEE 754 para números en punto flotante en precisión simple.

Por lo tanto, el valor de un número en formato IEEE 754 puede ser computado como

$$(-1)^s \times 2^e \times \text{significante}$$

donde:

$$s = \begin{cases} 0 & \text{si el número es positivo} \\ 1 & \text{si el número es negativo} \end{cases}$$

- El exponente se interpreta como el valor resultante de restar el sesgo correspondiente según el formato (127 en simple precisión, 1023 en precisión doble) al valor cargado en el registro: $e = E - \text{sesgo}$.
- El significante tiene un bit no visible a la izquierda con valor 1, luego un punto tampoco visible, y luego una sucesión de bits cuyos pesos son potencias negativas decrecientes de dos:

$$\text{significante} = 1.f$$

donde f es cualquier secuencia de bits que representa una fracción, de manera que $0 \leq f < 1$. Por lo tanto

$$1 \leq \text{significante} < 2.$$

En la Sección 4.6 veremos una excepción a esta regla.

4.3. Conversión de decimal a IEEE 754 simple precisión

Ejemplo 1. Se quiere convertir el número decimal 2.625 al formato IEEE 754 simple precisión. La conversión se puede realizar a través de los siguientes pasos:

1. Convertir la parte entera a binario: $(2)_{10} = (10)_2$

2. Convertir la parte fraccional a binario:

$$\begin{aligned} 0.625 \times 2 &= 1.25 \longrightarrow b_{-1} = 1 \\ 0.25 \times 2 &= 0.5 \longrightarrow b_{-2} = 0 \\ 0.5 \times 2 &= 1.0 \longrightarrow b_{-3} = 1 \end{aligned}$$

Por lo tanto:

$$(0.625)_{10} = (0.101)_2$$

3. Ya tenemos el número convertido a binario: $(2.625)_{10} = (10.101)_2$. Sin embargo, este número no está normalizado según la norma IEEE 754.

4. Para normalizar, primero agregar el exponente: $(10.101)_2 = (10.101)_2 \times 2^0$

5. Luego, normalizar según lo visto previamente: $(10.101)_2 \times 2^0 = (1.0101)_2 \times 2^1$

6. Por lo tanto, el significante resulta

$$(1.010100000000000000000000)_2$$

donde el primer 1 y el punto están implícitos, es decir no ocupan bits en el registro.

7. Corregir el exponente sumando el sesgo correspondiente (sesgo=127):

$$E = e + sesgo = 1 + 127 = 128$$

8. Convertir el exponente a binario:

$$E = (128)_{10} = (10000000)_2$$

9. El número es positivo. Por lo tanto el bit de signo es $s = (0)_2$

10. Finalmente, el número 2.625 convertido a formato IEEE 754 simple precisión resulta:

$$\underbrace{0}_{signo} \underbrace{10000000}_{exponente} \underbrace{01010000000000000000000000000000}_{significante\ con\ "1." \ implícito}$$

4.4. Conversión de IEEE 754 simple precisión a decimal

Ejemplo 2. Se quiere convertir el siguiente número expresado en IEEE 754 simple precisión a decimal:

$$N = (0100\ 0001\ 0101\ 1010\ 0000\ 0000\ 0000\ 0000)_{IEEE\ 754\ simple\ prec.}$$

La conversión se puede realizar a través de los siguientes pasos:

- Separar el número en sus diferentes partes:

Signo	Exponente	Significante (con 1. implícito)
1 bit	8 bits	23 bits
0	10000010	1011010000000000000000000

- El bit de signo ($s = 0$) es cero por lo tanto el número es positivo.

- Convertir el exponente a decimal:

$$E = (10000010)_2 = (130)_{10}$$

- Restar al exponente el sesgo correspondiente (sesgo=127):

$$e = E - \text{sesgo} = 130 - 127 = 3$$

- Convertir el significante a decimal:

$$(1.f)_2 = (1.10110100000000000000000)_2 = 1 + 1 \times 2^{-1} + 1 \times 2^{-3} + 1 \times 2^{-4} + 1 \times 2^{-6} = (1.703125)_{10}$$

- Finalmente, el número convertido a decimal resulta:

$$N = (-1)^s \times (1.f)_2 \times 2^e = 1.703125 \times 2^3 = (13.625)_{10}$$

4.5. Características principales

La Tabla 1 se muestra un resumen de las características de los números en punto flotante de la norma IEEE 754 en precisión simple, doble y cuádruple. La norma IEEE 754 va más allá de la simple definición de formatos, detallando cuestiones prácticas y procedimientos para que la aritmética en punto flotante produzca resultados uniformes y predecibles independientemente de la plataforma utilizada. Con este fin, el estándar IEEE 754 agrega a los números normalizados cuatro tipos numéricos que se describen en la Tabla 2.

4.6. Números denormalizados

El menor número normalizado en simple precisión es 1.0×2^{-126} . Antes de la definición del estándar si se presentaban problemas de “underflow” debía redondearse a cero. Los números denormalizados del estándar tienen una mejor aproximación al problema. Estos números tienen un exponente con todos los bit en cero (no permitido para los números normalizados) y una fracción de 23, 52, 64 o 112 bits, según el formato, dónde al menos un bit tiene que ser no nulo. Por otra parte, el uno implícito del significante es cero para los números denormalizados. El exponente para los números denormalizados es de -126 o -1022 , según corresponda.

Los números denormalizados permiten representar números más pequeños. Por ejemplo, trabajando en simple precisión el rango que cubren los números denormalizados (que se suma al rango de los números normalizados) es $[2^{-23} \times 2^{-126}, (1 - 2^{-23}) \times 2^{-126}]$. Notar que en la representación de los números denormalizados el exponente es $e = e_{\min}$ y no $e = e_{\min} - 1$.

Tabla 1: Características de los números en punto flotante (Norma IEEE 754).

	Precisión Simple	Precisión Doble	Precisión Cuádruple
Bits de signo	1	1	1
Bits de exponente	8	11	15
Bits de fracción	23	52	112
Bits totales	32	64	128
Sesgo del exponente	+127	+1023	+16383
Rango del exponente	[-126, 127]	[-1022, 1023]	[-16382, 16383]
Valor más chico (normalizado)	2^{-126}	2^{-1022}	2^{-16382}
Valor más grande (normalizado)	$\approx 2^{128}$	$\approx 2^{1024}$	$\approx 2^{16382}$
Rango decimal	$[\approx 10^{-38}, \approx 10^{38}]$	$[\approx 10^{-308}, \approx 10^{308}]$	$[\approx 10^{-4932}, \approx 10^{4932}]$
Valor más chico (desnormalizado)	$2^{-126} \cdot 2^{-23} \cong 10^{-45}$	$2^{-1022} \cdot 2^{-52} \cong 10^{-324}$	$2^{-16382} \cdot 2^{-112} \cong 10^{-4966}$

Tabla 2: Tipos numéricos del standard IEEE 754

Signo	Exponente e	Fracción f	Representa	Denominación
\pm	$e = e_{min} - 1$ codificado como $E = (00\dots0)_2$	$f = 0$	± 0	Ceros
\pm	$e = e_{min} - 1$ codificado como $E = (00\dots0)_2$	$f \neq 0$	$\pm 0.f \times 2^{e_{min}}$	Núm. denormalizados
\pm	$e_{min} \leq e \leq e_{max}$	Cualquier patrón	$\pm 1.f \times 2^e$	Núm. normalizados
\pm	$e = e_{max} + 1$ codificado como $E = (11\dots1)_2$	$f = 0$	$\pm\infty$	Infinitos
\pm	$e = e_{max} + 1$ codificado como $E = (11\dots1)_2$	$f \neq 0$	NaN	<i>Not a Number</i>

Es decir, el exponente del número desnormalizado es $e = -126$ para simple precisión a pesar de que en el registro realmente se almacena el número $E = (0000\ 0000)_2$ que correspondería al exponente $e = -127$. De lo contrario quedaría un “hueco” entre 2^{-127} y 2^{-126} . En la Figura 2 se observa el desbordamiento a cero gradual mediante números de punto flotante denormalizados con simple precisión.

4.7. Ceros

El estándar IEEE 754 provee una representación para el cero. La representación consiste en todos cero en el exponente y todos ceros en la fracción. El signo puede ser cero o uno, dando lugar a ± 0 . Esta representación está por fuera de la normalización previamente descrita.

4.8. Infinitos

El estándar IEEE 754 provee una representación para $\pm\infty$. Esta representación consiste en todos unos en el exponente ($e = e_{max} + 1$) y todos ceros en la fracción. El signo puede valer cero o uno, dando lugar a $\pm\infty$. Estos números pueden ser utilizados como operador y cumplen las siguientes reglas matemáticas:

1. $N + (+\infty) = +\infty$
2. $N - (+\infty) = -\infty$
3. $N + (-\infty) = -\infty$

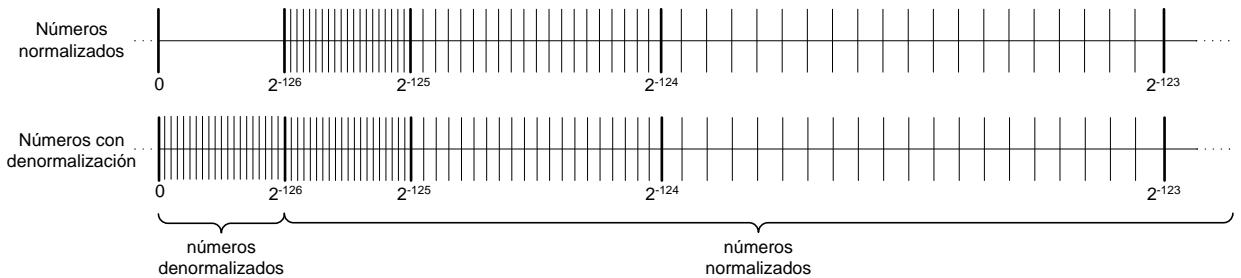


Figura 2: Números con formato de punto flotante denormalizados.

4. $N - (-\infty) = +\infty$
5. $(+\infty) + (+\infty) = +\infty$
6. $(-\infty) + (-\infty) = -\infty$
7. $(-\infty) - (+\infty) = -\infty$
8. $(+\infty) - (-\infty) = +\infty$

4.9. Formato NaN

El formato NaN (*Not a Number*) es una entidad simbólica utilizada en punto flotante. Sirve para representar valores de variables no inicializadas y tratamiento de tipo aritmético que no están contempladas en el estándar. La representación está compuesta por todos unos en el exponente y cualquier secuencia de bits en la fracción a excepción de la secuencia compuesta por todos ceros ($f \neq 0$). Estos números también pueden ser utilizados como operador y cumplen las siguientes reglas matemáticas:

1. $0/0 = \text{NaN}$
2. $\pm\infty / \pm\infty = \text{NaN}$
3. $0 * \pm\infty = \text{NaN}$
4. $\infty - \infty = \text{NaN}$

5. Operaciones de números en punto flotante

Para sumar o restar números en punto flotante hay que igualar los dos exponentes desplazando la mantisa. Luego, se pueden sumar o restar las mantisas. Por otra parte, para multiplicar, se multiplican las mantisas y se suman los exponentes. Análogamente, para dividir, se dividen las mantisas y se restan los exponentes. Concretamente, la secuencia de acciones que debe realizarse para cada operación es la siguiente:

5.1. Suma o resta

1. Chequear si alguno de los operandos en cero, NaN o $\pm\text{INF}$.
2. Igualar los exponente (si es necesario), desplazando el significante.
3. Completar con ceros (si es necesario).
4. Sumar o restar los significantes.
5. Normalizar el resultado (si es necesario).

Ejemplo Restar los siguientes números en IEEE 754 simple precisión:

$$S = 123 - 4.75$$

Primero convertimos ambos números de acuerdo a la norma IEEE 754 simple precisión:

$$\begin{array}{rcl} 123 & = & 0|10000101|110110000000000000000000 \\ 4.75 & = & 0|10000001|0011000000000000000000 \end{array}$$

Ahora tenemos que alinear los significantes de manera tal de igualar los exponentes. Se procede modificando el menor de los números para perder menos precisión. En este ejemplo modificamos el significante del número 4.75 hasta igualar el exponente del número 123, el cual es 133. Para ello debemos correr el punto 4 veces hacia la izquierda:

$$4.75 = 0|10000101|000100110000000000000000$$

Notar que este número no está normalizado dado que el significante no tiene un uno implícito sino que tiene un cero implícito. Sin embargo, ahora podemos restar los significantes:

$$\begin{array}{r} 1.1110110000000000000000 \\ - 0.0001001100000000000000 \\ \hline 1.1101100100000000000000 \end{array}$$

Este significante ya está normalizado. Por lo tanto, podemos juntar con el exponente y el resultado está expresado en IEEE 754:

$$S = 0|10000101|1101100100000000000000 = 118.25$$

5.2. Multiplicación

1. Chequear si alguno de los operandos es cero, NaN o $\pm\text{INF}$.
2. Sumar los exponentes.
3. Multiplicar los significantes.
4. Normalizar el producto (si es necesario).

Ejemplo Multiplicar los siguientes números en IEEE 754 simple precisión:

$$P = 56 \times 12$$

Primero convertir los números a IEEE 754 simple precisión:

$$\begin{array}{rcl} 56 & = & 0|10000100|110000000000000000000000 \\ 12 & = & 0|10000010|100000000000000000000000 \end{array}$$

Multiplicar los significantes:

$$\begin{array}{r}
 & 1. & 1 & 1 & 0 & 0 & 0 & \dots & 0 \\
 \times & 1. & 1 & 0 & 0 & 0 & 0 & \dots & 0 \\
 \hline
 & 0 & 0 & 0 & 0 & 0 & 0 & \dots & 0 \\
 & \vdots & & & & & & \vdots & \\
 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & \dots & 0 & 0 \\
 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & \dots & 0 & 0 \\
 \hline
 1 & 0. & 1 & 0 & 1 & 0 & 0 & 0 & 0 & \dots & 0 & 0
 \end{array}$$

El significante resultante no está normalizado. Normalizando, resulta:

$$1.0101000000000000000000000000000$$

Luego, sumar los exponentes y sumar 1 debido al corrimiento del punto realizado al normalizar:

$$e = e_1 + e_2 + 1 = 9$$

Por lo tanto, el exponente almacenado resulta:

$$E = e + \text{sesgo} = 136$$

Finalmente, el resultado es el siguiente:

$$P = 0|10001000|010100000000000000000000000000 = 672$$

5.3. División

1. Chequear si alguno de los operandos es cero, NaN o $\pm\text{INF}$.
2. Restar los exponentes.
3. Dividir los significantes.
4. Normalizar (si es necesario).

6. Densidad de los números en punto flotante

Al contrario que con los números en punto fijo, en punto flotante la distribución en la recta numérica no es uniforme y a medida que nos alejamos del origen se van separando. Cómo se verá a continuación, existe un compromiso entre rango y precisión.

Supongamos que P (el número de bits del significando) sea 24. En el intervalo $[1, 2)$ (con exponente $e = 0$) es posible representar 2^{23} números equiespaciados y separados con una distancia $1/2^{23}$. De modo análogo, en cualquier intervalo $[2^e, 2^{e+1})$ habrá 2^{23} números equiespaciados pero la densidad de este caso será $2^e/2^{23}$.

Por ejemplo, entre $2^{20} = 1048576$ y $2^{21} = 2097152$ hay $2^{23} = 8388608$ números pero el espaciado es de solo $1/8$. De este modo se deriva una regla práctica que cuando es necesario comparar dos números en punto flotante relativamente grandes es preferible comparar la diferencia relativa entre esos dos números en lugar de las magnitudes absolutas de los mismos dado que la precisión es menor cuanto más grandes sean los números. En la Figura 3 se puede apreciar cómo aumenta la separación entre dos números consecutivos en función del exponente e en el rango $e = [20, 30]$.

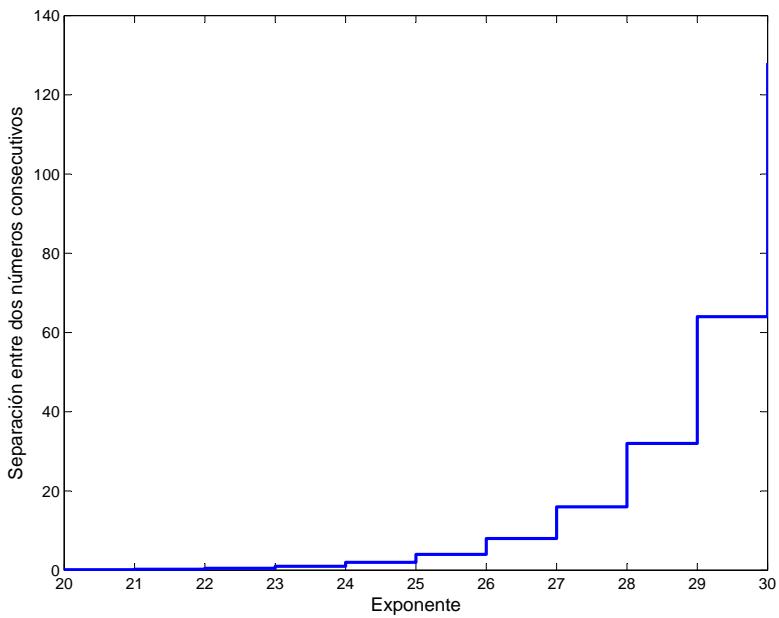


Figura 3: Evolución de la separación entre dos números consecutivos en función del exponente e en punto flotante.

7. Precauciones al operar con números en punto flotante

La alineación o ajuste se consigue desplazando a la derecha el número más pequeño (incrementando su exponente) o desplazando a la izquierda el más grande (decrementando su exponente). Dado que cualquiera de estas operaciones puede ocasionar que se pierdan dígitos, conviene desplazar el número más pequeño ya que los dígitos que se pierden tienen una importancia relativa menor.

Por ejemplo, supongamos que queremos sumar $S = 123 + 2.000001$ en IEEE 754 simple precisión. Convirtiendo ambos números resulta:

$$\begin{array}{rcl} 123 & = & 0|10000101|11101100000000000000000000000000 \quad (e = 6) \\ 2.000001 & = & 0|10000000|0000000000000000000000000000000100 \quad (e = 1) \end{array}$$

Desplazando el significante del número menor hasta igualar ambos exponentes (cinco veces hacia la derecha), el significante del número 2.000001 resulta 0.000010 \dots 0. Por lo tanto, la suma de ambos significantes resulta:

$$\begin{array}{r} 1. \quad 1 \quad 1 \quad 1 \quad 0 \quad 1 \quad 1 \quad 0 \quad 0 \quad \dots \quad 0 \\ + \quad 0. \quad 0 \quad 0 \quad 0 \quad 0 \quad 1 \quad 0 \quad 0 \quad 0 \quad \dots \quad 0 \\ \hline 1. \quad 1 \quad 1 \quad 1 \quad 1 \quad 0 \quad 1 \quad 0 \quad 0 \quad \dots \quad 0 \end{array}$$

Juntando con el exponente resulta:

$$S = 0|10000101|11101100000000000000000000000000 = 125$$

El resultado obtenido es incorrecto y el error cometido es 1×10^{-6} .

Como resultado de lo anterior se pueden establecer los siguientes enunciados:

1. Se tiene más precisión si se opera con número de magnitud similares.
2. El orden de evaluación puede afectar la precisión
3. Cuando se restan dos números del mismo signo o se suman dos números con signo opuesto, la precisión del resultado puede ser menor que la precisión disponible en el formato.
4. Cuando se realiza una cadena de cálculos tratar de realizar primero los productos y cocientes: $x \otimes (y \oplus z) \neq x \otimes y \oplus x \otimes z$
5. Cuando se multiplica y divide un conjunto de números, tratar de juntarlos de manera que se multipliquen números grandes y pequeños y por otro lado se dividan números de magnitud similares.
6. Cuando se comparan dos números en punto flotante, siempre comparar con respecto a un valor de tolerancia pequeño. Por ejemplo, en lugar de comparar $x=y$, realizar la evaluación `IF ABS((x-y)/y)) <= tol`. De todas maneras, además hay que tener precaución con los casos límites: $x = y = 0$, $y = 0$.

Apéndices

A. Cálculo del epsilon de máquina

Con el siguiente código en C se puede determinar el epsilon de máquina:

```
#include <stdio.h>
int main() {
double x = 1.0;
int n = 0;
while (1.0 + (x * 0.5) > 1.0) {
++n;
x *= 0.5;
}
printf("Epsilon de la máquina en forma binaria = 2^(-%d)\n", n);
printf("Epsilon de la máquina en forma decimal = %G\n", x);
return 0;
}
```

Referencias

- [1] Mano, M.M., *Computer system architecture*, Prentice-Hall, 1993.
- [2] Hyde, R., *The art of assembly language*, No Starch Pr, 2003.

- [3] Goldberg, D, *What every computer scientist should know about floating-point arithmetic*, ACM Computing Surveys (CSUR), **(23)**, 5-48, 1991.
- [4] IEEE Computer Society, *IEEE Standard for Floating-Point Arithmetic*, 2008.
- [5] Carter, P., *PC Assembly Language*, 2006.

Ensamblador y Arquitectura x86-64

Federico Bergero
Diego Feroldi

Arquitectura del Computador *
Departamento de Ciencias de la Computación
FCEIA-UNR



* Actualizado Agosto 2019 (D. Feroldi, feroldi@fceia.unr.edu.ar)

Índice

1. La arquitectura x86-64	1
1.1. Registros	1
1.2. Registros Especiales	2
1.2.1. Registro rflags	3
1.3. Lenguaje de máquina	4
1.4. Lenguaje Ensamblador de x86-64	4
2. Instrucciones	6
2.1. Tipos de instrucciones	6
2.2. Instrucciones para copia de datos	10
2.3. Comparaciones, Saltos y Estructuras de Control	11
2.3.1. Saltos	11
2.3.2. Estructuras de Control	12
2.3.3. Iteraciones	13
2.4. Manejo de Arreglos y Cadenas	15
2.4.1. Copia y manipulación de datos	15
2.4.2. Búsquedas y Comparaciones	17
2.4.3. Iteraciones de instrucciones de cadena	18
2.5. Instrucciones de Conversión	19
3. Acceso a datos en memoria	20
3.1. Directivas al Ensamblador	20
3.2. Etiquetas	21
3.3. Definir una etiqueta global	21
3.4. Endianness	22
3.5. Definición de variables	22
3.6. Acceso a datos: modos de direccionamiento	23
3.7. Comentario sobre acceso a memoria	26
3.8. Instrucción “ <i>load effective address</i> ”	27
3.9. Gestión de la pila	28
4. Aritmética de Punto Flotante	32
4.1. Copias y conversiones	33
4.2. Operaciones de punto flotante	33
4.3. Instrucciones <i>packed</i> - SIMD	34
5. Funciones y Convención de Llamada	36
5.1. Convención de llamada	38
6. Compilando código ensamblador con GNU as	40

Nota: Estas notas de clase reseñan las principales características de la arquitectura x86-64 y de su ensamblador. No es para nada una referencia completa del lenguaje ensamblador ni de la arquitectura sino que debe ser utilizado como material complementario con lo visto en las clases teóricas.

1. La arquitectura x86-64

x86-64 es una ampliación de la arquitectura x86. La arquitectura x86 fue lanzada por Intel con el procesador Intel 8086 en el año 1978 como una arquitectura de 16 bits. Esta arquitectura de Intel evolucionó a una arquitectura de 32 bits cuando apareció el procesador Intel 80386 en el año 1985, denominada inicialmente i386 o x86-32 y finalmente IA-32. Desde 1999 hasta el 2003, AMD amplió esta arquitectura de 32 bits de Intel a una de 64 bits y la llamó x86-64 en los primeros documentos y posteriormente AMD64. Intel pronto adoptó las extensiones de la arquitectura de AMD bajo el nombre de IA-32e o EM64T, y finalmente la denominó Intel 64.

La arquitectura x86-64 (AMD64 o Intel 64) de 64 bits da un soporte mucho mayor al espacio de direcciones virtuales y físicas. Proporciona registros de propósito general de 64 bits y buses de datos y direcciones también de 64 bits por lo cual las direcciones de memoria (punteros) son también valores de 64 bits. Aunque posee registros de 64 bits también permite operaciones con valores de 256, 128, 32, 16, y 8 bits.

1.1. Registros

La arquitectura x86-64 posee 16 registros de propósito general (cada uno de 64 bit): **rax**, **rbx**, **rcx**, **rdx**, **rsi**, **rdi**, **rbp**, **rsp** y **r8-r15**. Los 8 primeros registros se denominan de manera parecida a los 8 registros de propósito general de 32 bits disponibles en la arquitectura IA-32 (**eax**, **ebx**, **ecx**, **edx**, **esi**, **edi**, **ebp** y **esp**). Además, dependiendo de la versión cuenta con registros adicionales para control, punto flotante, etc. Dentro de los registros hay algunos de uso especial como el **rsp** y el **rip** que son utilizados para manipular la pila (como veremos en la Sección 3.9) y apuntar a la próxima instrucción, respectivamente.

Adicionalmente, la arquitectura proporciona 16 registros SSE, cada uno de 128 bits de ancho (**xmm1-xmm15**). Intel AVX (*Advanced Vector Extensions*) proporciona además 16 registros AVX de 256 bits de ancho (**ymm0-ymm15**). Los 128 bits inferiores de **ymm0-ymm15** tienen un alias a los respectivos registros SSE de 128 bits (**xmm0-xmm15**).

La mayoría de los registros de 64 bits están divididos en subregistros de 32, 16 y 8 bits. Así, el registro **rax** de 64 bits contiene en sus 32 bits más bajos al subregistro **eax** (e por *extended*), en sus 16 bits más bajos al subregistro **ax** y a su vez **ax** se divide en dos registros de 8 bits, llamados **ah** (por *high*) y **al** (por *low*), respectivamente. Por razones históricas, esta última división en dos registros de 8 bits sólo se realiza para los registros **rax**, **rbx**, **rcx** y **rdx**. Para el resto de los registros sólo existe la parte baja de 8 bits.

Los registros introducidos en la versión de 64 bits (**r8-r16**) se dividen en **r8d** (por doble word, 32 bits), **r8w** (de word, 16 bits) y **r8b** (por byte, 8 bits).

En la Fig. 1 vemos (casi) todos los registros del x86-64 con sus subregistros y su uso durante una llamada a función.

	63	31	15	8 7	0	
%rax		%eax	%ax	%ah	%al	Return value
%rbx		%ebx	%ax	%bh	%bl	Callee saved
%rcx		%ecx	%cx	%ch	%cl	4th argument
%rdx		%edx	%dx	%dh	%dl	3rd argument
%rsi		%esi	%si		%sil	2nd argument
%rdi		%edi	%di		%dil	1st argument
%rbp		%ebp	%bp		%bp1	Callee saved
%rsp		%esp	%sp		%spl	Stack pointer
%r8		%r8d	%r8w		%r8b	5th argument
%r9		%r9d	%r9w		%r9b	6th argument
%r10		%r10d	%r10w		%r10b	Callee saved
%r11		%r11d	%r11w		%r11b	Used for linking
%r12		%r12d	%r12w		%r12b	Unused for C
%r13		%r13d	%r13w		%r13b	Callee saved
%r14		%r14d	%r14w		%r14b	Callee saved
%r15		%r15d	%r15w		%r15b	Callee saved

Figura 1: Registros de propósito general del x86-64 y sus subdivisiones [5].

1.2. Registros Especiales

Existen varios registros más que no son de uso general y no pueden ser modificados o leídos por las instrucciones habituales:

rip (Instruction Pointer o Contador de programa): Apunta o guarda la dirección de memoria de la próxima instrucción a ejecutar.

ss (Stack segment): Indica cuál es el segmento utilizado para la pila ¹.

cs (Code Segment): Indica cuál es el segmento de código. En este segmento debe alojarse el código ejecutable del programa. En general este segmento es marcado como sólo lectura.

ds (Data Segment): Indica cuál es el segmento de datos. Allí se alojan los datos del programa (como variables globales).

¹El comienzo y longitud de los segmentos son guardados en una tabla. Este registro es sólo un índice en esa tabla

es, fs, gs: Estos registros tienen un uso especial en algunas instrucciones (las de cadena) y también pueden ser utilizados para referir a uno o más segmentos extras.

1.2.1. Registro rflags

El procesador incluye un registro especial llamado registro **rflags** o de status, en el cual se refleja el estado del procesador, información acerca de la última operación realizada y ciertos bits de control permiten cambiar el comportamiento del procesador.

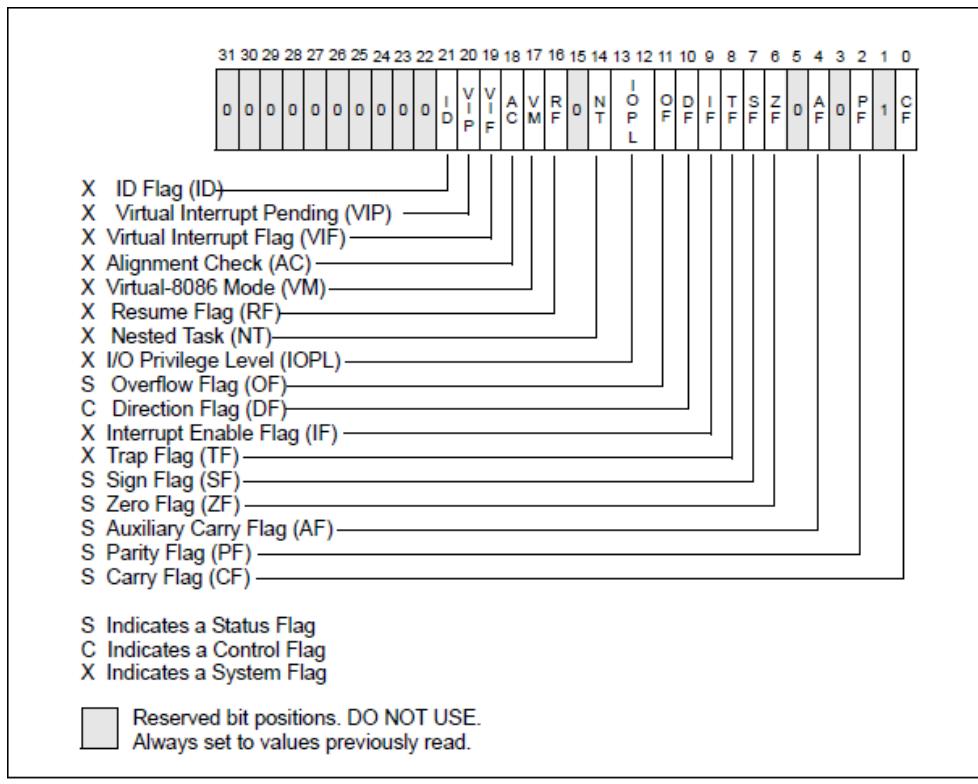


Figura 2: Registro EFLAGS

En la Fig. 2 vemos el registro EFLAGS (la versión de 32 bits del **rflags**). Vemos que hay varios bits de estado (todos los marcados con “S”). Describimos brevemente algunos de ellos:

CF Carry: en 1 si la última operación realizó acarreo.

ZF Zero: en 1 si la última operación produjo un resultado igual a cero.

OF Overflow: en 1 si la última operación desbordó (el resultado no es representable en el operando destino).

SF Sign: en 1 si la última operación arrojó un resultado negativo.

DF Direction: indica la dirección para instrucciones de manejo de cadenas (que veremos más adelante).

El registro **rflags** no es de propósito general por lo cual no puede ser accedido ni modificado por instrucciones regulares (**add**, **mov**, etc).

1.3. Lenguaje de máquina

Los procesadores son dispositivos de hardware encargados de ejecutar el programa alojado en memoria. En la actualidad un programador escribe un programa en algún lenguaje de programación de alto nivel, por ejemplo, C, Java, Haskell, etc. La CPU no ejecuta el programa descripto en este lenguaje sino que este debe ser traducido (o compilado) a *lenguaje de máquina*.

El lenguaje de máquina es una representación muy críptica para los humanos, por ejemplo el código de máquina x86-64 (escrito en hexadecimal) de una función que suma dos enteros es como sigue:

```
48 89 f8  
48 01 f0  
c3
```

Para facilitar la tarea de los programadores de computadores en los años 50 se introdujo el lenguaje ensamblador, el cual tiene una representación más legible para las personas. Por ejemplo, el mismo código de la función para sumar dos enteros se escribiría en ensamblador x86-64 como:

```
movq %rdi, %rax  
addq %rsi, %rax  
ret
```

En este fragmento de código se ve una sintaxis de operación seguida de argumentos donde las operaciones, llamadas instrucciones, tienen un nombre representativo (`mov` por mover aunque en realidad copia un valor, `add` por adicionar, etc). Veremos de aquí en adelante qué significa cada instrucción de ensamblador y sus formas de uso.

1.4. Lenguaje Ensamblador de x86-64

En este apunte utilizaremos la sintaxis de AT&T de lenguaje ensamblador ya que es la utilizada por el Ensamblador de GNU: “as”.

Detallamos aquí las principales características de esta sintaxis:

- Los comentarios de línea comienzan con `#` (a partir de `#` comienza un comentario hasta el fin de línea).
- El nombre de los registros comienza con `%`. Por ejemplo, el registro `rax` se escribe como `%rax`.
- Las constantes se prefijan con `$`. Así, la constante `5` se escribe como `$5`. Un caso particular que veremos luego son las etiquetas.
- Las direcciones de memoria se escriben sin ninguna decoración. Por lo tanto, la expresión `3000` refiere a la dirección de memoria `3000` y **no a la constante `3000`** (que se escribiría `$3000` por lo antes dicho).

- Las instrucciones que manipulan datos (tanto registros como memoria) se sufijan con el tamaño del dato. Por ejemplo, agregar el sufijo **q** a la instrucción **mov** resultando **movq**.

Los sufijos posibles son los siguientes:

Sufijo	Denominación	Tamaño en bytes
b	<i>Byte</i>	1
w	<i>Word</i>	2
l	<i>Double word (o long)</i>	4
q	<i>Quad</i>	8
t	<i>Ten</i>	10
s	<i>Single precision float</i>	4
d	<i>Double precision float</i>	8

En el ensamblador de GNU (as) este sufijo es opcional cuando el tamaño de los operandos puede ser deducido, aunque es conveniente escribirlo siempre para detectar posibles errores.

- Las instrucciones se escriben como:

operando origen, destino

Notar que el destino es el argumento de la derecha por lo cual la instrucción **movq %rax, %rbx** representa **rax → rbx** (copiar el valor de **rax** a **rbx**, es decir que luego de ejecutar la instrucción **rbx=rax** y el valor del registro **rax** sigue siendo el mismo).

- Para de-referenciar un valor se utilizan los paréntesis, por ejemplo **(%rax)** refiere a lo **apuntado** por **rax**. Ejemplo: **movq (%rax), %rbx** copiará en el registro **rbx** lo apuntado por el registro **rax** y no el contenido del mismo.

Esta notación permite también las formas:

- **K(%reg)** refiere al valor apuntado por **reg** más un corrimiento de **K** bytes, donde **K** es entero. El valor de **K** puede ser negativo, por lo cual se puede conseguir un corrimiento ascendente o descendente. Notar que aquí la constante **K** no lleva **\$**. Ejemplo: **8(%rbp), -16(%rbp)**.
- **K(%reg1, %reg2, S)** donde **K** y **S** son constantes enteras y **S={1, 2, 4, 8}** refiere al valor **reg1 + (reg2 * S + K)**.

Ejemplos: **(%rax,%rax,2)**, **-4(%rbp, %rdx, 4)**, **8(%rax,4)**. En el último caso vemos que **reg1** es opcional. Este tipo de direccionamiento sirve para acceder a arreglos.

Por ejemplo, si tenemos un arreglo de enteros de 32 bits (4 bytes) apuntado por **rax** y queremos acceder el elemento 6 podemos hacer:

```
movq $6, %rcx
movl (%rax,%rcx, 4), %edx # edx <- *(rax+4*6)
```

Estos modos de direccionamiento los veremos con mayor detalle en la Sección 3.6.

2. Instrucciones

Como vimos previamente, las instrucciones de ensamblador en la arquitectura x86-64 están compuestas por una operación (por ejemplo, suma, resta, comparación, etc.) acompañada de operandos (por ejemplo, valores a sumar). En algunos casos las instrucciones no toman operandos o sus operandos son implícitos. Por ejemplo, la instrucción `ret` no toma operandos e `inc` incrementa en uno el valor de su operando, este uno está implícito.

2.1. Tipos de instrucciones

El juego de instrucciones de los procesadores x86-64 es muy amplio. Podemos organizar las instrucciones según los siguientes tipos:

1. Instrucciones de transferencia de datos:

- **mov destino, fuente**: instrucción genérica para mover un dato desde un origen a un destino. Ejemplo: `movq %rax, %rbx # rbx=rax` (Ver en detalle en la Sección 2.2).
- **push fuente**: instrucción que mueve el operando de la instrucción al tope de la pila. Ejemplo: `pushq %rax` (Ver en detalle en la Sección 3.9).
- **pop destino**: mueve el dato que se encuentra en el tope de la pila al operando destino. Ejemplo: `popq %rax` (Ver en detalle en la Sección 3.9).
- **xchg destino, fuente**: intercambia contenidos de los operandos. Ejemplo: `xchg %rax, %rbx`

2. Instrucciones aritméticas y de comparación. La familia de procesadores x86 ofrece múltiples instrucciones para realizar operaciones numéricas, entre ellas:

- **add fuente, destino**: suma aritmética de los dos operandos.
Ejemplo: `addq %rax, %rbx # rbx+=rax`
- **adc fuente, destino**: suma aritmética de los dos operandos considerando el bit de transporte. Ejemplo:

```
movb $0, %dl
movb $0xFF, %al
addb $0xFF, %al # al=0xFE, CF=1
adcb $0, %dl # dl=1
```

- **sub fuente, destino**: resta aritmética de los dos operandos.
Ejemplo: `subq %rax, %rbx # rbx-=rax`
- **sbb fuente, destino**: resta aritmética de los dos operandos considerando el bit de transporte. Ejemplo:

```
movl $1, %edx
movl $0, %eax,
subl $1, %eax # CF=1.
sbbl $0, %edx # edx=0
```

- **inc destino**: incrementa el operando en una unidad.
Ejemplo: `incq %rax # rax++`

- **dec destino:** decrementa el operando en una unidad.
Ejemplo: `decq %rax # rax--`
- **imul fuente:** multiplicación entera con signo. La instrucción `imul` tiene tres formatos: con un operando, con dos operandos y con tres operandos. La forma con un operando utiliza los registros `rax` y `rdx` de forma implícita: multiplica el valor del operando con `rax` y el resultado queda en `rdx:rax`. La forma con dos operandos multiplica sus dos operandos y almacena el resultado en el segundo operando. El operando resultado (es decir, el segundo) debe ser un registro. La forma con tres operandos multiplica sus operandos segundo y tercero y almacena el resultado en su último operando. Nuevamente, el operando resultante debe ser un registro. Además, el primer operando se limita a ser un valor constante.
- **mul fuente:** multiplicación entera sin signo. Esta instrucción a diferencia de la anterior solo admite el formato con un operando.

Ejemplos:

```
.text
.global main
main:
    movq $0xfffffffffffffff, %rax
    movq $4, %rbx
    mulq %rbx      # rax=0xfffffffffffffff y rdx=3
    movq $9, %rax
    movq $3, %rbx
    imulq %rbx, %rax      # rax=27
    imulq $4, %rax      # rax=108
    imulq $2, %rax, %rbx      # rax=216
    movq $5, %rax      # rax=5
    imulq $3, %rax, %rbx      # rax=15
    ret
```

- **idiv fuente:** división entera con signo. La instrucción `idiv` divide el contenido del entero de 128 bits `rdx:rax` (construido interpretando a `rdx` como los ocho bytes más significativos y a `rax` como los ocho bytes menos significativos) por el valor del operando especificado. El resultado del cociente de la división se almacena en `rax`, mientras que el resto se coloca en `rdx`. Ejemplo:

```
movq $0xfffffffffffffff, %rax
movq $0xff, %rdx
movq $1024, %rbx
divq %rbx      # rax=0x3fffffffffffff y rdx=0x3ff
```

- **div fuente:** división entera sin signo.
- **neg destino:** negación aritmética en complemento a 2.

Ejemplo:

```
movb $0xff, %al
negb %al      # rax=1
```

- **cmp destino, fuente:** comparación de los dos operandos; hace una resta sin guardar el resultado.

Ejemplo: `cmp %rax, %rbx # Si rax==rbx entonces ZF=1, si no ZF=0`

3. Instrucciones lógicas y de desplazamiento

- a) Operaciones lógicas (Ver en detalle en el Apunte *Manejo de Bits en Lenguaje C*):

- **and fuente, destino:** operación “y” lógica bit a bit.

Ejemplo: `andq %rax, %rbx # rbx=rax&rbx`

- **or fuente, destino:** operación “o” lógica bit a bit.

Ejemplo: `orq %rax, %rbx # rbx=rax|rbx`

- **xor fuente, destino:** operación “o exclusiva” lógica bit a bit.

Ejemplo: `xorq %rax, %rax # rax=rax^rax=0`

- **not destino:** negación lógica bit a bit.

Ejemplo:

```
movb $0xff, %al
notb %al      # al=0
```

- **test fuente, destino:** comparación lógica de los dos operandos; hace una “y” lógica sin guardar el resultado.

Ejemplo: `test %cl, %cl # ZF=1 si cl=0 y SF=1 si cl<0`

- b) Operaciones de desplazamiento (Ver en detalle en el Apunte *Manejo de Bits en Lenguaje C*):

- **sal cantidad, destino / shl cantidad, destino:** desplazamiento aritmético/lógico a la izquierda (estas dos instrucciones producen el mismo resultados). Ejemplo:

```
movb $0xaa, %al
salb $1, %al    # al=0x54
```

- **sar cantidad, destino:** desplazamiento aritmético a la derecha. Ejemplo:

```
movb $0xaa, %al
sarb $1, %al    # al=0xd5
```

- **shr cantidad, destino:** desplazamiento lógico a la derecha. Ejemplo:

```
movb $0xaa, %al
shrb $1, %al    # al=0x55
```

- **rol cantidad, destino:** rotación lógica a la izquierda. Ejemplo:

```
movb $0xaa, %al
rolb $1, %al    # al=0x55
```

- **ror cantidad, destino:** rotación lógica a la derecha. Ejemplo:

```
movb $0xaa, %al
rorb $1, %al    # al=0x55
```

- **rcl cantidad, destino:** rotación lógica a la izquierda considerando el bit de transporte. Ejemplo:

```
movb $0xaa, %al
stc # CF=1
rclb $1, %al    # al=0x55
```

- **rcr cantidad, destino:** rotación lógica a la derecha considerando el bit de transporte. Ejemplo:

```
movb $0xaa, %al
stc # CF=1
rcrb $1, %al    # al=0xd5
```

4. Instrucciones de ruptura de secuencia (Ver en detalle en la Sección 2.3)

a) Salto incondicional:

- **jmp etiqueta:** salta de manera incondicional a la etiqueta. Ejemplo:
`jmp etiqueta .`

b) Saltos que consultan un bit de resultado concreto:

- **je etiqueta / jz etiqueta:** salta a la etiqueta si igual, si el bit de cero está activo.
- **jne etiqueta / jnz etiqueta:** salta a la etiqueta si diferente, si el bit de cero no está activo.
- **jc etiqueta etiquet:** salta a la etiqueta si el bit de transporte está activo.
- **jnc etiqueta:** salta a la etiqueta si el bit de acarreo no está activo.
- **jo etiqueta:** salta a la etiqueta si el bit de desbordamiento está activo.
- **jno etiqueta:** salta a la etiqueta si el bit de desbordamiento no está activo.
- **js etiqueta:** salta a la etiqueta si el bit de signo está activo.
- **jns etiqueta:** salta a la etiqueta si el bit de signo no está activo.

c) Saltos condicionales sin considerar el signo:

- **jb etiqueta / jnae etiqueta:** salta a la etiqueta si es más pequeño.
- **jbe etiqueta / jna etiqueta:** salta a la etiqueta si es más pequeño o igual.
- **ja etiqueta / jnbe etiqueta:** salta a la etiqueta si es mayor.
- **jae etiqueta / jnb etiqueta:** salta a la etiqueta si es mayor o igual.

d) Saltos condicionales considerando el signo:

- **jl etiqueta / jnge etiqueta:** salta si es más pequeño.
- **jle etiqueta / jng etiqueta:** salta si es más pequeño o igual.
- **jg etiqueta / jnle etiqueta:** salta si es mayor.
- **jge etiqueta / jnl etiqueta:** salta si es mayor o igual.

e) Otras instrucciones de ruptura de secuencia:

- **loop etiqueta:** decrementa el registro **rcx** y salta si **rcx** es diferente de cero (Ver en detalle en la Sección 2.3.3).
- **call etiqueta:** llamada a (Ver en detalle en la Sección 5).
- **ret:** retorno de subrutina (Ver en detalle en la Sección 5).

5. Instrucciones para el registro **rflags**

Existen instrucciones especiales para trabajar con el registro **rflags**. Entre ellas distinguimos varias clases:

Apagar un bit: `clc` (*clear carry*), `cld` (*clear direction*).

Prender un bit: `stc` (*set carry flag*), `std` (*set direction flag*), `sti` (*set interruption flag*).

Sumar añadiendo el carry: `adc` toma dos operandos, los suma junto con el bit de carry y lo guarda en el destino.

Acceder al registro: `lahf` y `sahf` copian ciertos bits del registro `ah` hacia el `flags` y viceversa, `popfq` guarda en la pila el registro `flags` y `pushfq` trae de la pila el registro `flags`.

El uso del registro `rflags` se verá más claro en breve cuando expliquemos cómo se usa el registro para hacer saltos condicionales.

6. Instrucciones de entrada/salida

- **in destino, fuente:** lectura del puerto de E/S especificado en el operando fuente y se guarda en el operando destino.
- **out destino, fuente:** escritura del valor especificado por el operando fuente en el puerto de E/S especificado en el operando destino.

2.2. Instrucciones para copia de datos

Una operación muy común es la de copiar valores de un lugar a otro. Un programa debe intercambiar valores con la memoria, registros, etc. La arquitectura x86-64 ofrece varias instrucciones para hacer copias de datos siendo la más importante la instrucción `mov`. Esta instrucción toma la forma *movS origen, destino* donde “S” es el sufijo ². Los operandos pueden ser registros, memoria o constantes inmediatas. Por ejemplo, para escribir un valor 3 en el registro `rax` podemos hacer:

```
movq $3, %rax
```

Algunos ejemplos de uso de `mov` (con el equivalente en C en lo posible):

Ejemplo

```
movb $65, %al          # al = 'A'  
movq %rax, %rcx        # rcx=rax  
movw (%rax), %dx       # Copia en dx dos bytes comenzando en la  
                        # dirección rax.  
movw dx, (%rax)         # copia dx en la dirección rax  
movl 16(%rbp), %ecx     # Copia en ecx cuatro bytes (debido al  
                        # sufijo l) comenzando en la dirección rbp+16.
```

²Recordar que la instrucción `mov` no mueve sino que copia valores

2.3. Comparaciones, Saltos y Estructuras de Control

2.3.1. Saltos

Cualquier código estructurado requiere que la ejecución no siempre siga con la siguiente instrucción escrita, sino que ciertas veces el procesador debe continuar la ejecución en otra porción de código (por ejemplo, al llamar a una función o en distintas ramas de una estructura *if*). Para ello, todas las arquitecturas incluyen funciones de salto. Veremos la más simple primero.

La instrucción `jmp` toma como único operando una dirección a la cual “saltar”. El efecto que tiene este salto es que la próxima instrucción a ejecutar no será la siguiente al `jmp` sino la indicada en su operando. La dirección del salto en general se da usando etiquetas (ver Sección 3.2). Veamos un ejemplo:

```
movq $0, %rax
jmp cont
movq $1, %rax
cont:
    movq $2, %rax
```

En el fragmento de código anterior la instrucción `movq $1, %rax` **nunca** es ejecutada ya que la instrucción `jmp` hace que el procesador salte a la instrucción en la dirección `cont`. Notar aquí que aunque `cont` es una constante (la dirección de memoria donde está la instrucción `movq 2, %rax`) ésta no va prefijada por `$`.

La instrucción `jmp` permite hacer saltos y es el equivalente a un `goto` de un lenguaje de alto nivel. Pero ¿cómo podemos implementar estructuras de control como bucles y condicionales con ella? Respuesta: no se puede. Para ello debemos introducir los saltos condicionales.

Los saltos condicionales tienen la misma función que la instrucción `jmp` salvo que se realizan **sólo si** se da una condición, por ejemplo, el resultado de la última operación fue cero. Como vimos en la Sección 1.2.1, el procesador mantiene en el registro `rflags` el estado de la última operación realizada. Luego, los saltos condicionales de x86-64 hacen uso de este registro y realizan el salto si cierto bit de ese registro está en 1. De hecho, por cada bit de estado del registro `rflags` hay dos saltos condicionales, por ejemplo `jz` realiza el salto si el bit ZF está en uno y `jnz` lo realiza si el bit ZF **no** está en uno.

Tanto los saltos condicionales como los incondicionales no llevan sufijo ya que su operando es siempre una dirección de memoria (dentro del segmento de código).

Junto con los saltos condicionales la arquitectura x86-64 incluye una esta instrucción para comparar dos valores, la instrucción `cmp`. Como ya se mencionó, esta instrucción realiza una diferencia (resta) entre sus dos operandos, descartando el resultado pero **prendiendo los bits del registro rflags** acorde al resultado obtenido.

Siguiendo la lógica de la instrucción `sub`,

```
cmpq %rax, %rbx
```

realiza la resta de `rbx-rax`, se prende el bit SF (que indica negatividad) si `rax` es mayor que `rbx` pero a diferencia de `sub`, **no modifica el valor del registro destino rbx**. Notar que si ambos valores son iguales la resta tendrá un resultado nulo, prendiendo el

bit ZF. Como la relación que guardan dos valores (cuál es menor y cuál es mayor) depende de si dichos números se asumen con signo o sin signo existen dos versiones de los saltos condicionales por desigualdad: `j1` y `jg` (por lower y greater) para datos con signo y `ja` y `jb` (por above y below) para datos sin signo.

2.3.2. Estructuras de Control

Tratemos ahora de traducir el siguiente fragmento de función C en ensamblador:

```
long a=0;
if (a==100) {
    a++;
}
// seguir
```

Teniendo en cuenta lo que vimos sobre saltos y comparaciones, una posible traducción sería:

```
.global main
main:
    movq $0, %rax
        cmpq $100, %rax # comparamos el valor de rax con la constante 100
        jz igual_a_cien # si el resultado dio cero (rax-100) es porque
                        # son iguales
                        # en este caso debo incrementar a
        jmp seguir
igual_a_cien:
    incq %rax
    jmp seguir
seguir:
```

Veamos en el fragmento anterior varias cosas:

- El orden de los argumentos en la instrucción `cmp` es importante ya que la resta no es conmutativa. Notar también que esta instrucción necesita un sufijo de tamaño.
- Inmediatamente después de hacer la comparación realizamos el salto condicional. De tener más instrucciones en el medio, éstas podrían modificar el estado del registro `rflags`.
- Por la naturaleza del `if`, debemos definir dos etiquetas, una para saltar cuando la condición es verdadera (`igual_a_cien`) y otra para continuar la ejecución tanto si la condición fue verdadera o no (`seguir`). Notar que si la condición resulta falsa el programa saltará al bloque `igual_a_cien`.

Vemos ahora cómo traduciríamos el siguiente fragmento:

```
long a;
if (a==100) {
    a++;
} else {
    a--;
}
// seguir
```

En este caso el if tiene un else. Una posible traducción sería:

```
movq $0, %rax
cmpq $100, %rax # comparamos el valor de rax con la constante 100
jz igual_a_cien # si el resultado dio cero (rax-100) es porque son
iguales
# en este caso debo incrementar rax
decq %rax
jmp seguir

igual_a_cien:
incq %rax
jmp seguir

seguir:
...
...
```

Vemos en el fragmento anterior varias cosas:

- En este caso si el salto condicional no se realiza (porque la condición resultó falsa) se ejecutará el decremento.
- Como ambas ramas del if deben unificarse, luego de hacer el decreemento saltamos a `seguir` “salteando” la rama verdadera del if.
- Notar que como la etiqueta `seguir` está a continuación del bloque `igual_a_cien` el salto puede ser obviado.

2.3.3. Iteraciones

Otra estructura común en los lenguajes de alto nivel son las iteraciones, bucles o lazos. Con lo visto hasta ahora podemos ya traducir la mayoría de las estructuras iterativas. Supongamos que queremos traducir la siguiente estructura:

```
long int i;
while (i!=0) {
    cuerpo_del_while();
    i--;
}
```

Como antes, asumiremos que en ensamblador `i` es una etiqueta que aloja lugar para un entero de ocho bytes. Esto puede traducirse como:

```
while_1:
    cmpq $0, i # Evaluar la condicion
    je fin_1    # Si resulto falsa, el lazo termino

cuerpo_del_while_1: # Aca ira el cuerpo del while
...
```

```

...
decq i
jmp while_1
fin_1:
...
...

```

El código corresponde a la estructura de control que puede verse en la Fig. 3

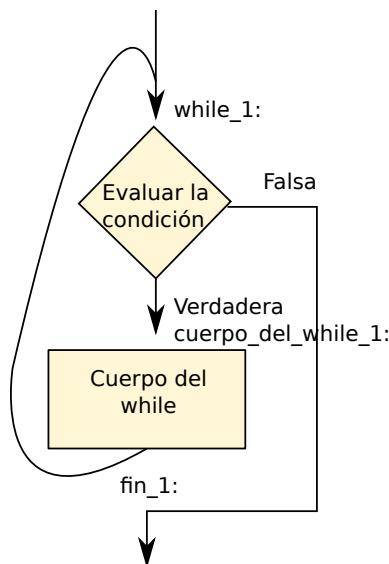


Figura 3: Estructura de *While*.

Las estructuras del tipo **for** son también muy comunes en lenguajes de alto nivel. Una forma particular de **for** es repetir un bloque de código una cantidad de veces dadas. Por ejemplo es muy común algo como:

```

int i;
for (i=100;i>0;i--) {
    cuerpo_del_for();
}

```

De hecho esto es tan común que la arquitectura incluye la instrucción **loop** para implementar estas estructuras. Esto puede traducirse como:

```

movq $100, %rcx # rcx se utiliza como iterador. Inicialmente 100
cuerpo_del_for_1:
...
...
...
loop cuerpo_del_for_1

```

La instrucción **loop** tiene dos efectos:

- Decrementa en uno el registro **rcx**. Aquí vemos que **rcx** tiene un uso especial.
- Luego, salta a la etiqueta **sólo si** el resultado de decrementar **rcx** dio distinto de cero. Si el resultado dio cero, el flujo del programa sigue en la siguiente instrucción al **loop**.

2.4. Manejo de Arreglos y Cadenas

Un arreglo es una estructura de datos que almacena una colección de elementos del mismo tipo (por lo tanto del mismo tamaño) y le asigna un índice entero a cada uno. Existen distintas variantes de arreglos (largo fijo/variable, uni/multi-dimensional) pero en este apunte nos centraremos en arreglos a la “C”, esto es, un arreglo **a** será la dirección del primer elemento (el de índice 0). Como cada elemento del arreglo tiene tamaño fijo al que llamaremos **s**, podemos calcular la dirección del elemento **i** del arreglo **a** como **a+i*s**.

Como esta estructura de datos es muy utilizada la arquitectura x86-64 incluye varias instrucciones (llamadas de cadena) para realizar copias, comparaciones, búsquedas, etc.

Esta familia de instrucciones hace uso especial de dos registros: **rsi** (source index) y **rdi** (destination index)³. Cuando el procesador ejecuta una instrucción de cadena, éste incrementa/decrementa automáticamente esos registros⁴ para apuntar al próximo elemento del arreglo. La cantidad incrementada/decrementada depende del tamaño del dato en cuestión. El bit DF (direction flag) del registro **rflags** le indica al procesador si debe incrementar o decrementar los registros de índice (se puede apagar con **cld** para que se incrementen o prender con **std** para que se decrementen).

2.4.1. Copia y manipulación de datos

El procesador ofrece tres instrucciones para la copia y manipulación de datos almacenados en arreglos:

lods (de *load string*) Copia en el registro **rax** (o en su sub-registro correspondiente) el valor apuntado por **rsi** e incrementa/decrementa **rsi** en la cantidad de bytes indicada por el sufijo de tipo. Así la instrucción **lodsw** (asumiendo DF=0) es equivalente a:

```
movw (%rsi),%ax
addq $2,%rsi
```

Notar que aquí se utiliza el subregistro **ax** para compatibilizar con el sufijo **w** de word y que por lo tanto el incremento es dos bytes.

stos (de *store string*) Almacena el valor del registro **rax** (o su sub-registro correspondiente) en la dirección apuntada por **rdi** y luego incrementa/decrementa el valor de **rdi** en la cantidad de bytes indicada por el sufijo de tipo. Así la instrucción **lodsl** (asumiendo DF=1) equivale a:

³Aunque su nombre sugieren que son índices, estos registros son apuntadores.

⁴Algunas instrucciones solo incrementan/decrementan uno de estos registros.

```

    movl %eax, (%rdi)
    subq $4, %rdi

```

movs (de *move string*) realiza las acciones de **lod\$** y **stos** aunque sin utilizar el registro **rax**, esto es, copia el valor apuntado por **rsi** en la posición de memoria apuntada por **rdi** e incrementa/decrementa **ambos** en la cantidad de bytes indicada por el sufijo de tipo. Así la instrucción **movsb** (asumiendo DF=0) es equivalente a

```

    movb (%rsi),%regtemp
    movb %regtemp, (%rdi)
    addq $1, %rsi
    addq $1, %rdi

```

siendo **regtemp** un registro temporario del procesador (en realidad no existe ese registro).

Notar que las instrucciones para manejo de arreglos y cadenas trabajan con operandos implícitos, es decir, los operandos no se declaran explícitamente sino que ya viene prefijado con que operandos se trabaja.

Un caso típico de uso de estas instrucciones de cadena es para traducir el siguiente fragmento C:

```

int f(char *a, char *b) {
    int i;
    for (i=0;i<100;i++)
        a[i]=b[i];
}

```

que puede ser implementado en ensamblador como

```

.global f
f:
    # por convención de llamada tenemos en rdi el puntero a "a"
    # y en rsi el puntero a b
    movq $100, %rcx      # debemos iterar 100 veces
    cld                  # iremos incrementando rsi y rdi (DF=0)
sigue:
    movsb
    loop sigue
    ret

```

Al repetir 100 veces la instrucción **movsb** copiamos los 100 bytes de **b** hacia **a**. El mismo efecto se podría haber obtenido copiando 50 veces un word (con **movsw**), 25 veces un long (con **movsl**) o 12 veces un quad (con **movsq**) y un long **extra**.

Supongamos que ahora debemos modificar el arreglo como sigue:

```

int f(int *a) {
    int i;
    for (i=0;i<100;i++)
        a[i]++;
}

```

Esto puede ser escrito utilizando instrucciones de cadena como sigue:

```

.global f
f:
# suponemos que rdi tiene "a"
movq %rdi, %rsi # el origen y el destino son el mismo arreglo
movq $100, %rcx # iteramos 100 veces
cld      # iremos incrementando rsi y rdi (DF=0)
l:
lodsl    # cargamos en eax el elemento del arreglo (apuntado por rsi)
incl %eax # lo incrementamos
stosl    # lo guardamos en el arreglo (apuntado por rdi)
loop l   # pasamos al siguiente elemento
ret

```

Vemos que en este caso el uso del registro `eax` es útil para obtener el valor original del elemento (con `lodsl`), modificar el registro (con `incl`) y luego guardarlo de nuevo (con `stosl`). Notar también que en este caso el arreglo destino y origen son el mismo, por ello copiamos `rdi` en `rsi` al iniciar la función.

2.4.2. Búsquedas y Comparaciones

Una operación común es buscar un elemento dentro de un arreglo o comparar dos arreglos. La arquitectura ofrece para esto dos instrucciones:

scas (de scan string) compara lo apuntado por `rdi` con el valor del registro `rax` (o del sub-registro según corresponda) e incrementa/decrementa `rdi` en la cantidad de bytes dada por el sufijo de tipo.

cmps (de compare string) compara el valor apuntado por `rsi` con el valor apuntado por `rdi` e incrementa/decrementa ambos registros en la cantidad de bytes dada por el sufijo de tipo.

Al igual que la instrucción `cmp` estas comparaciones prenden los bits correspondiente en el registro `rflags`.

Veamos un caso de uso de estas instrucciones. Supongamos que queremos implementar en ensamblador la siguiente función C que busca un elemento en un arreglo.

```

int find(int *a, int k) {
    int i;
    for (i=0;i<100;i++)
        if (a[i]==k) return 1;
    return 0;
}

```

Esta función puede ser implementada en ensamblador como sigue:

```
.global find
find:
    cld      # iremos incrementando rdi (DF=0)
    movq $100, %rcx # iteramos 100 veces
    movl %esi, %eax # buscamos el 2do argumento
sigue:
    scasl    # comparamos el elemento actual con eax
    je found  # si lo encontramos terminamos
    loop sigue # si no seguimos
    movq $0, %rax # no lo encontramos, retornar 0
    jmp fin
found:
    movq $1, %rax # lo encontramos, retornar 1
fin:
    ret
```

2.4.3. Iteraciones de instrucciones de cadena

Como vimos en los ejemplos anteriores, es lógico que una instrucción de cadena se repita muchas veces. Por ejemplo, una por cada elemento del arreglo o cadena. Para facilitar la escritura de estas estructuras iterativas la arquitectura ofrece la familia de prefijos `rep` que pueden ser antepuestos a cualquier instrucción de cadena. Al igual que la instrucción `loop` el prefijo repite la instrucción la cantidad de veces indicada por `rcx`. Así, el ejemplo de copia de un arreglo a otro de la Sección 2.4.1 puede ser reescrito en ensamblador como:

```
.global f
f:
    # por convención de llamada tenemos en rdi el puntero de a
    # y en rsi el puntero a b
    movq $100, %rcx # debemos iterar 100 veces
    cld  # iremos incrementando rsi y rdi (DF=0)
    rep movsb # repite movsb rcx veces
    ret
```

Al igual que existen los saltos condicionales, existen los prefijos de repetición condicionales. Así, los prefijos `repe` y `repne` repiten la instrucción mientras el bit Z esté prendido/apagado a lo sumo `rcx` veces. El ejemplo de la búsqueda de un entero de la Sección 2.4.2 puede ser reescrito utilizando prefijos de repetición condicional como:

```
.global find
find:
    cld      # iremos incrementando rdi (DF=0)
    movq $100, %rcx # iteramos 100 veces
    movl %esi, %eax # buscamos el 2do argumento
    repne scasl # repetimos mientras sea distinto o a lo sumo rcx veces
    je found  # si lo encontramos terminamos
```

```

movq $0, %rax # no lo encontramos, retornar 0
jmp fin
found:
    movq $1, %rax # lo encontramos, retornar 1
fin:
    ret

```

Notemos que el prefijo **repne** repite la instrucción mientras la comparación resulte distinta y a lo sumo **rcx** veces, pero ¿cómo saber por cuál de las dos causas finalizó la repetición?

Cuando la condición del prefijo resulta falsa los registros **rsi**,**rdi** son incrementados o decrementados según corresponda y el registro **rcx** es decrementado pero los bits del registro **rflags** quedan intactos dejando allí el valor de la última comparación. Por lo tanto, podemos realizar un salto condicional para ver si la última comparación dio igual o distinto.

2.5. Instrucciones de Conversión

Las instrucciones de conversión de datos realizan varias transformaciones de datos, como duplicación de tamaño de operando mediante extensión de signo, conversión de formato little-endian a big-endian, extracción de máscaras de signos, búsqueda en una tabla y soporte para operaciones con números decimales.

La arquitectura x86-64 ofrece instrucciones para convertir entre enteros de distintos tamaño:

cbw Extiende (con signo) **al** a **ax**.

cwde Extiende (con signo) **ax** a **eax**.

cwd Extiende (con signo) **ax** a **dx:ax** (los 16 bits altos a **dx** y los 16 bits mas bajo a **ax**).

cltq Extiende (con signo) **eax** a **rax**.

cqto Extiende (con signo) **rax** a **rdx:rax** (los 64 bits altos a **rdx** y los 64 bits bajos a **rax**).

movs Copia un valor del origen al destino extendiendo con el bit de signo. Esta instrucción se utiliza para extender datos con signo y tiene dos sufijos, el primero es el tamaño del dato origen y el segundo es el tamaño del dato destino. Algunos ejemplos:

```

movsbl %al, %eax # convierte un byte a un long
movswl %ax, %eax # convierte un word a un long
movswq %ax, %rax # convierte un word a un quad

```

movz Copia un valor del origen al destino extendiendo con *cero*. Esta instrucción se utiliza para extender datos sin signo y tiene dos sufijos, el primero es el tamaño del dato origen y el segundo es el tamaño del dato destino. Algunos ejemplos:

```
movzbl %al, %eax # convierte un byte a un long  
movzwl %ax, %eax # convierte un word a un long  
movzwq %ax, %rax # convierte un word a un quad
```

3. Acceso a datos en memoria

Para acceder a datos de memoria en ensamblador, como sucede en los lenguajes de alto nivel, lo haremos por medio de variables que deberemos definir previamente para reservar el espacio necesario para almacenar la información. Veamos primero algunos conceptos importantes.

3.1. Directivas al Ensamblador

Al igual que el compilador de C permite al programador indicar ciertas operaciones de preprocesamiento (`#include`, `#define`, `#pragma`, etc.), el compilador de ensamblador **as** permite al programador indicar ciertas operaciones y valores inicializados. Las directivas al compilador ensamblador comienzan siempre con “.”. Dentro de ellas destacamos las siguientes:

Describir el segmento Con éstas el programador indica a **qué** segmento debe agregarse el siguiente bloque. Las más comunes son **.data** (el siguiente bloque debe ir al segmento de datos) y **.text** (indicando que lo que sigue es código ejecutable).

Inicializar valores Esta clase emite valores constantes indicados por el programador directamente en el bloque, es decir no se hace traducción. Dentro de esta clase tenemos:

ascii, asciz Permiten inicializar una lista de cadenas con y sin carácter nulo al final de cada una. Ejemplos: `.asciz "Hola mundo"`, `.ascii "a" "b" "c"` (este ejemplo es una lista de strings).

byte Inicializa una lista de bytes. Ejemplo: `.byte 'a' 'b'`, `.byte 97`, `.byte 0x61`.

double, float Inicializa una lista de valores de punto flotante de doble y simple precisión, respectivamente. Ejemplo: `.double 3.1415 2.16`, `.float 5.3`.

short, long, quad Emite una lista de valores enteros de 2, 4 y 8 bytes. Ejemplo: `.short 20`, `.long 50`, `.quad 0`.

space Emite un bloque de tamaño fijo inicializado en cero o en un valor pasado como argumento. Ejemplo: `.space 128`, `.space 5000`, `0`. Esta directiva es útil para obtener un bloque de memoria de tamaño dado (ya sea inicializado o no).

Es importante notar que todas estas directivas toman como argumento una **lista** de valores a inicializar. Un error muy común es no indicar ningún elemento en esa lista, por ejemplo:

```
.data  
.long
```

lo cual **NO** reserva espacio. La versión correcta sería `.long 0`.

3.2. Etiquetas

Las etiquetas son una parte fundamental del lenguaje ensamblador. Una etiqueta hace referencia a un elemento dentro del programa ensamblador. Su función es facilitar al programador la tarea de hacer referencia a diferentes elementos del programa. Las etiquetas sirven para definir constantes, variables o posiciones del código y las utilizamos como operandos en las instrucciones o directivas del programa.

Por ejemplo, cuando uno define en C una variable (`long i;`) está indicándole al compilador que reserve espacio de memoria para un entero y que este espacio lo nombraremos mediante el identificador `i`. Tanto en C como en ensamblador nombrar un espacio de memoria es útil para el programador pero esta información no es usada por la computadora sino que una etiqueta se convierte en una **dirección de memoria**.

Ejemplo

En ensamblador con sintaxis AT&T una etiqueta es un nombre seguido de “:”.

```
.data
i: .long 0
f: .double 3.14

.text
.global main
main:
    movq $0, %rax
    retq
```

Aquí vemos que la etiqueta `i` (dentro del segmento de datos) define la posición de memoria donde el ensamblador alojará un entero inicializado en 0 (4 bytes). Luego en `f` un valor de punto flotante inicializado en 3.14 (8 bytes).

Finalmente, vemos que dentro del segmento de código se define una etiqueta **global** llamada `main`. Este será el punto de inicio de todo programa.

3.3. Definir una etiqueta global

La directiva `.global` indica que la etiqueta nombrada a continuación es de alcance global.

Ejemplos

```
.global main
.global sum
```

De no especificar esta directiva la etiqueta desaparece luego del proceso de compilación. Las etiquetas globales deben ser utilizadas, por ejemplo, con las etiquetas que definen funciones que serán llamadas fuera del archivo ensamblador. Por ejemplo, cuando se enlaza un programa C con uno escrito en ensamblador, las funciones incluidas en ensamblador deben ser definidas como globales (siendo `main` el caso más común). Esto se verá en detalle en la Sección 6.

3.4. Endianness

El término inglés *endianness* designa el formato en el que se almacenan en memoria los datos de más de un byte. El problema es similar a los idiomas en los que se escribe de derecha a izquierda, como el árabe, o el hebreo, frente a los que se escriben de izquierda a derecha, pero trasladado de la escritura al almacenamiento en memoria.

Supongamos que tenemos que almacenar el entero 168496141 en la dirección de memoria `a`. Este valor se representa mediante los cuatro bytes 0x0A 0x0B 0x0C 0x0D (escribiendo más a la izquierda el byte más representativo).

Una opción es guardar el byte **más** significativo (0x0A) en la dirección `a`, el segundo (0x0B) en la dirección `a+1`, y así sucesivamente. Esto se conoce como convención *Big-Endian* como puede verse en la Fig. 4(a).

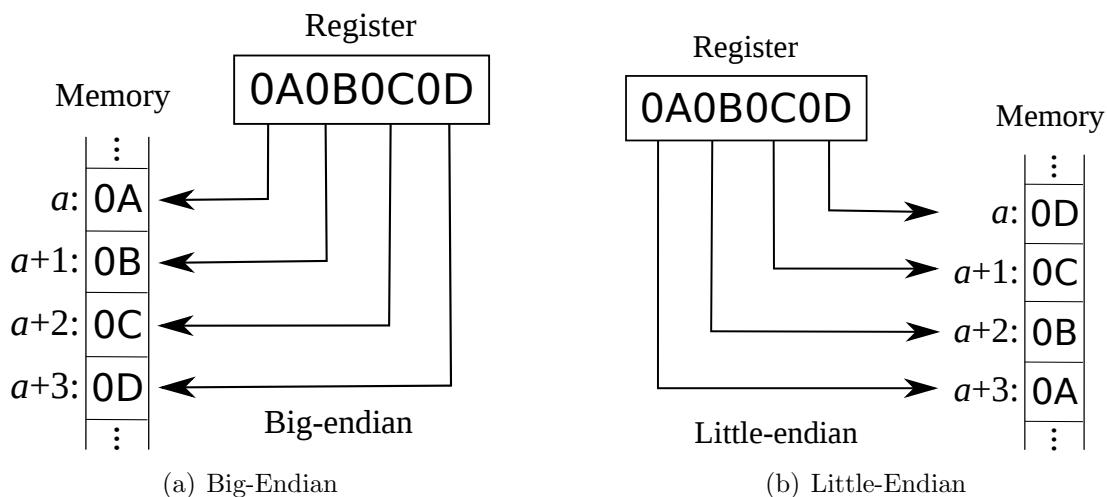


Figura 4: Convenciones de *Endianness* (Fuente Wikipedia).

Otra opción es almacenar en la dirección `a` el byte **menos** significativo (0x0D), el siguiente (0x0C) en la dirección `a+1` y así. Esta última convención se denomina *Little-Endian* y es la utilizada por las arquitecturas x86 y por la tanto también por x86-64 . La Fig. 4(b) muestra la convención *Little-Endian*.

3.5. Definición de variables

La declaración de variables en un programa en ensamblador se puede incluir en la sección `.data`. Las variables de esta sección se definen utilizando las directivas vistas en la Sección 3.1. Por ejemplo, `var: .long 0x12345678` es una variable con el nombre `var` de tamaño 4 bytes inicializada con el valor 0x12345678 que comienza en la dirección de memoria cuya etiqueta es `var`. Es importante destacar que en ensamblador hay que estar

muy alerta cuando accedemos a las variables que hemos definido previamente. Las variables se guardan en memoria consecutivamente a medida que las declaramos y no existe nada que delimita las unas de las otras. Veamos a continuación un ejemplo ilustrativo.

Ejemplo

```
.data
var1: .byte 0
var2: .byte 0x61
var3: .word 0x0200
var4: .long 0x0001E26C
```

Las variables se encontrarán en memoria tal como muestra la siguiente tabla (suponiendo que la variable var1 está en la dirección 0x600880):

	Dirección (en bytes)	Valor
var1:	0x600880	0x00
var2:	0x600881	0x61
var3:	0x600882	0x00
	0x600883	0x02
var4:	0x600884	0x6C
	0x600885	0xE2
	0x600886	0x01
	0x600887	0x00

Si ejecutamos la instrucción `mowq var1, %rax`, el procesador tomará como primer byte el valor de var1, pero también los 7 bytes que están a continuación, por lo tanto, como los datos se tratan en formato little-endian, en el registro `rax` quedará cargado el valor 0x0001E26C02006100. Si este acceso no es el deseado, el compilador no reportará ningún error, ni tampoco se producirá un error durante la ejecución; solo podremos detectar que lo estamos haciendo mal probando el programa y depurando. Por una parte, el acceso a los datos es muy flexible, pero, por otra parte, si no controlamos muy bien el acceso a las variables, esta flexibilidad puede causar serios problemas.

3.6. Acceso a datos: modos de direccionamiento

A continuación, veremos los diferentes modos de direccionamiento que podemos utilizar en un programa ensamblador para acceder a datos:

1. **Inmediato.** En este caso, el operando hace referencia a un dato que se encuentra en la propia instrucción. El valor especificado debe poder ser expresado con 32 bits como máximo, que será el resultado de evaluar una expresión aritmética formada por valores numéricos y operadores aritméticos. También se puede sumar una dirección de memoria representada mediante una etiqueta (nombre de una variable), con la excepción de la instrucción `mov` cuando el segundo operando es un registro de 64 bits, para el que podremos especificar un valor que se podrá expresar con 64 bits.

Ejemplos

```
movq $0x1122334455667788, %rax
```

*Carga en el registro **rax** el valor 0x1122334455667788.*

```
movb $100, var
```

*Carga el valor 100 en la dirección de memoria **var**.*

```
$var, %rbx
```

*Carga el valor de la dirección de memoria de la variable **var** en el registro **rbx**.*

```
movq $var+16*2, %rbx
```

*Carga en el registro **rbx** el valor de la dirección de memoria de la variable **var** más 32.*

2. **Directo a registro.** En este caso, el operando hace referencia a un dato que se encuentra almacenado en un registro (no hay acceso a memoria). En este modo de direccionamiento podemos especificar cualquier registro de propósito general.

Ejemplo

```
movq %rax, %rbx # rbx=rax
```

3. **Directo a memoria.** En este caso, el operando hace referencia a un dato que se encuentra almacenado en una posición de memoria. El operando habrá de especificar el nombre de una variable de memoria.

Ejemplos

```
movq var, %rax
```

*Carga en el registro **rax** 8 bytes a partir de la dirección de memoria **var**.*

```
movq 0x600880, %rax
```

*Carga en el registro **rax** 8 bytes a partir de la dirección 0x600880.*

```
movq %rax, var
```

*Carga el contenido del registro **rax** en la dirección de memoria **var**.*

4. **Indirecto a registro.** En este caso, el operando hace referencia a un dato que se encuentra almacenado en una posición de memoria. El operando habrá de especificar un registro entre paréntesis; el registro contendrá la dirección de memoria a la cual queremos acceder.

Ejemplo

```
movq (%rax), %rbx
```

El primer operando utiliza la dirección que tenemos en rax para acceder a memoria. Se mueven 8 bytes a partir de la dirección especificada por rax y se guardan en rbx.

5. **Indexado.** En este caso, el operando hace referencia a un dato que se encuentra almacenado en una posición de memoria. El operando especifica una dirección de memoria como dirección base (que puede ser expresada mediante un número o el nombre de una variable que tengamos definida) sumada a un registro que actúa como índice respecto a esta dirección de memoria entre paréntesis.

Ejemplos

```
movq 2(%rax), %rbx
```

Carga en el registro rbx 8 bytes a partir de la dirección de memoria rax+2.

```
movq var(%rax), %rbx
```

Carga en el registro rbx 8 bytes a partir de la dirección de memoria rax+var.

6. **Relativo.** En este caso, el operando hace referencia a un dato que se encuentra almacenado en una posición de memoria. El operando especifica una dirección de memoria de la siguiente manera:

$$[\text{base} + \text{índice} \times \text{escala} + \text{desplazamiento}]$$

donde la base y el índice pueden ser cualquier registro de propósito general, la escala puede ser 1, 2, 4 u 8 y el desplazamiento ha de ser un número representable con 32 bits que será el resultado de evaluar una expresión aritmética formada por valores numéricos y operadores aritméticos:

$$\text{desplazamiento}(\text{registro base}, \text{registro índice}, \text{escala})$$

También podemos sumar una dirección de memoria representada mediante una etiqueta (nombre de una variable). Podemos especificar solo los elementos que nos sean necesarios.

Ejemplos

```
movq 3(%rbx, %rcx, 4), %rax
```

*Carga en el registro **rax** 8 bytes a partir de la dirección **rbx+rcx*4+3**.*

```
movq (%rax, %rax, 2), %rax
```

*Carga en el registro **rax** 8 bytes a partir de la dirección **rax+rax*2**.*

```
movq 4(%rbp, %rdx, 4), %rax
```

*Carga en el registro **rax** 8 bytes a partir de la dirección **rbp+rdx*4-4**.*

```
movq 8(%rax, 4), %rax
```

*Carga en el registro **rax** 8 bytes a partir de la dirección **rax*4+8**. En este caso vemos que el registro base es opcional.*

3.7. Comentario sobre acceso a memoria

Como hemos visto, podemos acceder a un dato en memoria utilizando la etiqueta que define la dirección de memoria donde dicho dato comienza. Ahora supongamos que queremos incrementar el valor de una variable definida por la etiqueta **i**, esto podemos hacerlo simplemente escribiendo:

```
incq i
```

Es importante notar que aunque la etiqueta **i** es una constante (la dirección de memoria donde se aloja ese entero) no lleva el signo **\$**.

Si ahora quisiéramos sumar **i** con el registro **rax** podemos escribir:

```
addq i, %rax
```

Sin embargo, notar que **addq \$i, %rax** produce un efecto muy diferente. En este caso sumará una constante (la dirección de **i**) y no el valor alojado en **i**.

Muchas veces es útil conocer la dirección de memoria donde está alojado un valor. Esto en C se conoce como obtener un puntero al dato. Así, si tenemos una variable **long int i;** podemos obtener un puntero a dicha variable utilizando el operador de referencia, escribiendo **&i**. Como antes mencionamos, en ensamblador una etiqueta es una dirección de memoria constante. Por ello si quisiéramos obtener el valor de esa dirección podríamos escribir:

```
movq $i, %rax
```

Luego **rax** guardará la dirección de memoria del entero antes definido.

El siguiente ejemplo es interesante para ver la diferencia entre usar una etiqueta y el valor allí guardado.

Ejemplo

```

.data
str: .asciz "hola mundo"

.text
.global main
main:
movq str , %rax # Instrucción 1
movq $str, %rax # Instrucción 2
retq

```

¿Qué diferencia hay entre la instrucción 1 y la 2? Aunque casi similares, las dos instrucciones son muy distintas entre sí. Ambas son un movimiento con destino a `rax`, pero veamos qué mueven...

Al ejecutar la primera, `rax` toma el valor de 7959387902288097128. ¿Qué ha ocurrido aquí? La instrucción le indica al procesador que debe copiar 8 bytes (ya que es un quad) desde la región de memoria indicada por la etiqueta `str` a `rax`. Como en esa región de memoria se aloja la cadena de caracteres "hola mundo" los primeros 8 bytes son hola mun y de allí el valor tan extraño. El valor 7959387902288097128 se puede descomponer en hexadecimal en los siguientes bytes 0x6e 0x75 0x6d 0x20 0x61 0x6c 0x6f 0x68, donde cada uno corresponde en decimal a 110 117 109 32 97 108 111 104 y al convertirlo en caracteres ASCII son “num aloh” (notar que la frase aparece al revés por ser x86-64 little endian).

Al ejecutar la segunda lo que ocurrirá es que en `rax` se guardará la dirección de memoria donde está guardada la cadena de caracteres. Este valor dependerá del proceso de compilación. Notemos que en este caso ningún carácter de esa cadena será copiado a `rax`. De hecho esa instrucción no accede a la memoria.

3.8. Instrucción “load effective address”

La arquitectura x86-64 ofrece una instrucción similar al operador de referencia de C. Esta instrucción es `lea` (por “load effective address”). Así la instrucción `mov $str, %rax` es equivalente a

```
leaq str, %rax
```

Las instrucciones `lea` y `mov` (desde memoria) están relacionadas: `mov` carga el contenido de una dirección de memoria mientras que `lea` carga la dirección en sí.

La instrucción `lea` a menudo se usa como un “truco” para hacer ciertos cálculos, aunque ese no sea su propósito principal. Usando sintaxis AT&T, los modos de direccionamiento útiles con `lea` son los siguientes:

```

lea desplazamiento(%base), %dest
lea (,%índice, multiplicador), %dest
lea desplazamiento(, %índice, multiplicador), %dest
lea (%base, %índice, multiplicador), %dest
lea desplazamiento(%base, %índice, multiplicador), %dest

```

lo cual corresponde a

```
%dest = desplazamiento + %base
%dest = %índice * multiplicador
%dest = desplazamiento + %índice * multiplicador
%dest = %base + %índice * multiplicador
%dest = desplazamiento + %base + %índice * multiplicador
```

donde el desplazamiento es una constante entera, el multiplicador es 2, 4 u 8, y `%dest`, `%índice` y `%base` son registros.

Se puede usar esto para multiplicar un registro por 2, 3, 4, 5, 8, o 9, y sumar una constante en solo paso:

```
lea constante(, %src, 2), %dst
lea constante(%src, %src, 2), %dst
lea constante(, %src, 4), %dst
lea constante(%src, %src, 4), %dst
lea constante(, %src, 8), %dst
lea constante(%src, %src, 8), %dst
```

donde `%src` y `%dst` pueden ser el mismo registro.

3.9. Gestión de la pila

Una pila es una estructura de datos que permite almacenar información. Su funcionamiento puede analizarse pensando en una pila de platos sobre una mesa. Uno puede agregar platos y la pila irá creciendo. Luego, si uno quiere sacar un plato quitará el plato del “tope”, achicando la pila de platos. Como se ve, cuando uno saca un elemento de la pila, sacará el último elemento insertado (si lo hubiera). Por ello la estructura de datos pila se conoce como *LIFO* (*Last-In First-Out*), dado que el último en entrar es el primero en salir.

La arquitectura x86-64 permite al programador utilizar una porción de la memoria como pila. Esto se conoce como el segmento de pila (que **no** es el mismo segmento que el segmento de datos ni de código).

La pila puede usarse para varias cosas:

- Espacio de almacenamiento temporal. Las variables automáticas de C por ejemplo se almacenan en la pila.
- Implementar el llamado a función (y en especial las recursivas). Notar que el orden de llamada y finalización de las funciones ocurre como una pila. Así, si tenemos que la función `f` llama a `g` y `g` llama a `h`, la primera función que finalizará es `h`, luego `g` y finalmente `f`.
- Preservar el valor de registros durante un llamado a función. Como veremos en la Sección 5, algunos registros son modificados cuando uno realiza un llamado a función. El programador puede guardar el valor de ese registro en la pila y restaurarlo luego de la llamada.

Aunque la arquitectura permite utilizar la pila con cualquier fin, es muy común que cada **función** utilice una porción de la pila para guardar sus variables locales, argumentos, dirección de retorno, etc. A esta sub-porción de pila se la conoce como **marco de**

activación de la función. En la Fig. 5 vemos un posible estado de la pila con diferentes marcos de activación (sólo **uno** está activo en un momento dado, el de la función que se está ejecutando).

En la Fig. 5 se ve que el último elemento insertado en la pila está ubicado en direcciones **más bajas de memoria**, es decir, en la implementación de x86-64 la pila crece hacia direcciones más bajas. Esto es así por cuestiones históricas y para permitir que tanto el segmento de datos como el de pila crezcan de forma de optimizar el espacio libre (el de datos crece desde abajo hacia arriba y el de pila desde arriba hacia abajo).

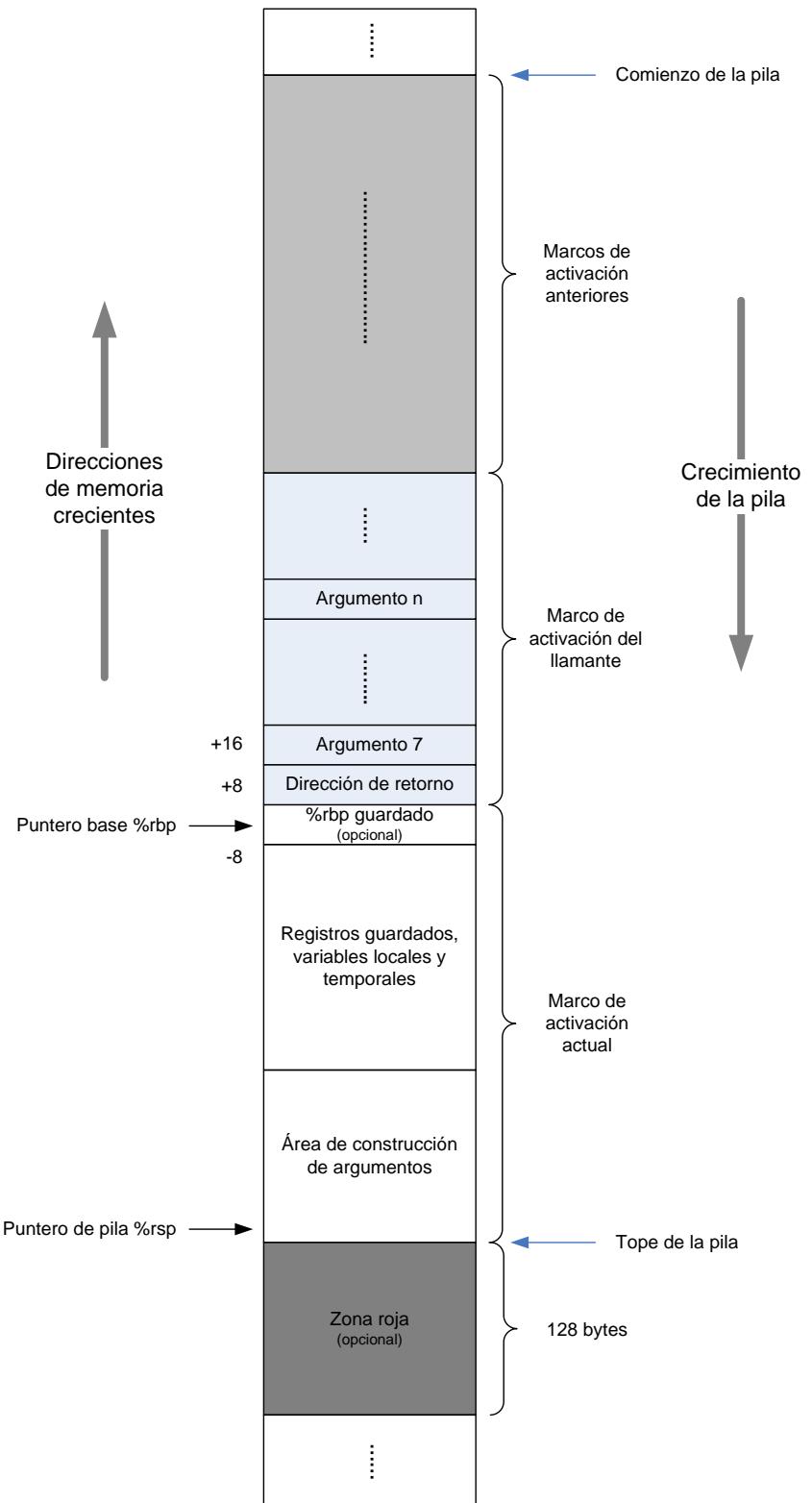


Figura 5: Diagrama de la estructura de pila x86-64.

La arquitectura posee dos registros especiales para manipular la pila:

rsp (*stack pointer*) Es un registro de 64 bits que apunta (guarda la dirección de memoria) al último elemento apilado dentro del segmento de pila (**tope**).

rbp (*base pointer*) Es un registro de 64 bits que apunta al **inicio** de la sub-pila o marco de activación.

Aunque ambos registros tienen este uso particular pueden ser manipulados por las instrucciones habituales (`add`, `mov`, etc). La arquitectura x86-64 ofrece también dos instrucciones especiales para apilar/desapilar elementos:

pushq Primero decrementa el registro `rsp` en 8 (recordemos que la pila crece hacia direcciones más bajas) y luego almacena en esa dirección el valor que toma como argumento. Así, la instrucción `pushq $0x12345678` es equivalente a

```
subq $8, %rsp  
movq $0x12345678, (%rsp)
```

El comportamiento de la instrucción `pushq` puede verse en la Figura 6.

popq Primero copia el valor apuntado por el registro `rsp` en el operando que toma como argumento, luego incrementa el registro `rsp` en 8 (la pila decrece hacia direcciones más altas). Así, la instrucción `popq %rax` es equivalente a

```
movq (%rsp), %rax  
addq $8, %rsp
```

El comportamiento de la instrucción `popq` puede verse en la Fig. 7. Notar que el valor `0x12345678` continua almacenado en la dirección `0xff8`.

En la descripción de las instrucciones `push` y `pop` solo podemos utilizar el sufijo `q` (entero de 8 bytes) ya que por cuestiones de alineación los datos insertados en la pila deben ser de 8 bytes.

En la Fig. 5 se puede observar un área denominada **zona roja**. La arquitectura x86-64 especifica que los programas pueden usar los 128 bytes más allá del puntero de la pila actual (es decir, en direcciones más bajas que el puntero). Así, el área de 128 bytes más allá de la ubicación señalada por `rsp` se considera reservada y no debe modificarse mediante señales o manejadores de interrupciones. Por lo tanto, las funciones pueden usar esta área para datos temporales que no son necesarios en las llamadas a funciones. En particular, las funciones de “hoja” (*leaf functions*) pueden usar esta área directamente, en lugar de ajustar el puntero de la pila en el prólogo y el epílogo. La zona roja es una optimización. El código puede suponer que los 128 bytes por debajo de `rsp` no se modifican por señales o controladores de interrupción, y por lo tanto puede usarlo para datos temporales, sin mover explícitamente el puntero de pila. Sin embargo, hay que tener en cuenta que la zona roja será utilizada por llamadas a función, por lo que generalmente es más útil en funciones que no llaman a otras funciones.

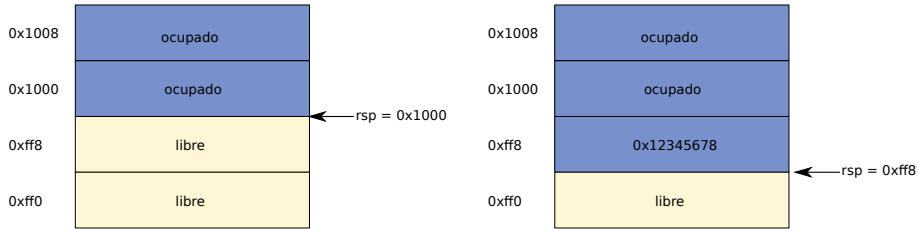


Figura 6: Diagrama de la memoria antes y después de ejecutar la instrucción `pushq`.

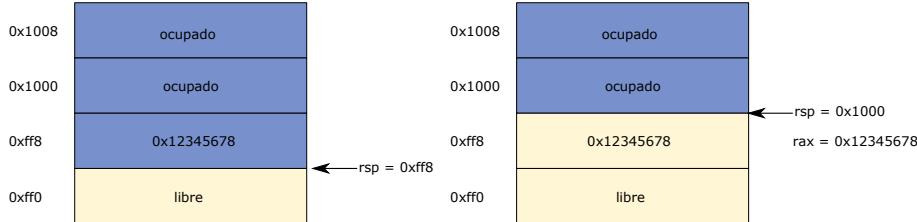


Figura 7: Diagrama de la memoria antes y después de ejecutar la instrucción `popq`.

4. Aritmética de Punto Flotante

La arquitectura x86-64 soporta aritmética de datos de punto flotante utilizando el estándar IEEE 754 tanto para simple como doble precisión. Las operaciones de punto flotante se realizan a través de una extensión de la arquitectura que podemos considerar separada conceptualmente de la ALU (llamada SSE -Streaming SIMD Extension)

Por lo tanto se utilizan otros registros e instrucciones. Para esto hay 16 registros de 128 bits (16 bytes): `xmm0` a `xmm15`. Cada registro puede contener un elemento (i.e.: un flotante de simple o doble precisión) en cuyo caso el valor se considera “escalar” (scalar) y se usa sólo una parte del registro, o puede contener múltiples elementos del mismo tamaño (formato “empaquetado” -packed-). Por ejemplo, en `xmm0` entran 4 flotantes de simple precisión o también 16 enteros de 1 byte (chars). El formato empaquetado permite que algunas instrucciones realicen la misma operación sobre varios datos a la vez (SIMD: Single Instruction Multiple Data).

Las instrucciones siguen algunas reglas:

- Las letras **s** (por “scalar”) y **p** (“packed”) indican qué formato se utiliza.
- Las letras **s** (por “single”), **d** (“double”) e **i** (integer) indican el tipo de datos involucrado. Además **q** indica que un entero es tamaño quadword (i.e.: 8 bytes).

Por ejemplo `cvtssi2sdq`, que permite convertir un entero almacenado en un quadword a un doble en formato escalar, se interpreta así:

- **cvt**: convert (convertir)
- **si**: scalar integer (un entero con signo)
- **2**: two (“two” suena como “to” - a -)
- **sd**: scalar double (un flotante escalar de doble precisión)
- **q**: quadword (el entero mencionado es un quadword)

Veremos primero las instrucciones de copias y conversiones, luego las operaciones aritméticas escalares y luego las operaciones sobre datos empaquetados (SIMD).

4.1. Copias y conversiones

Al igual que con los registros de propósito general, existen para los registros de punto flotante las instrucciones `movss` y `movsd` que copian un dato de precisión simple (*float*) y doble precisión (*double*) respectivamente de un registro `xmm` a otro o desde/hacia la memoria.

A su vez existen múltiples instrucciones para convertir entre enteros y datos de punto flotante. En la Fig. 8 se recopilan las instrucciones de conversión.

Instruction	Source	Destination	Description
<code>movss</code>	M_{32}/X	X	Move single precision
<code>movss</code>	X	M_{32}	Move single precision
<code>movsd</code>	M_{64}/X	X	Move double precision
<code>movsd</code>	X	M_{64}	Move double precision
<code>cvtss2sd</code>	M_{32}/X	X	Convert single to double precision
<code>cvtsd2ss</code>	M_{64}/X	X	Convert double to single precision
<code>cvtssi2ss</code>	M_{32}/R_{32}	X	Convert integer to single precision
<code>cvtssi2sd</code>	M_{32}/R_{32}	X	Convert integer to double precision
<code>cvtssi2ssq</code>	M_{64}/R_{64}	X	Convert quadword integer to single precision
<code>cvtssi2sdq</code>	M_{64}/R_{64}	X	Convert quadword integer to double precision
<code>cvttss2si</code>	X/M_{32}	R_{32}	Convert with truncation single precision to integer
<code>cvttsd2si</code>	X/M_{64}	R_{32}	Convert with truncation double precision to integer
<code>cvttss2siq</code>	X/M_{32}	R_{64}	Convert with truncation single precision to quadword integer
<code>cvttsd2siq</code>	X/M_{64}	R_{64}	Convert with truncation double precision to quadword integer

X : XMM register (e.g., `%xmm3`)

R_{32} : 32-bit general-purpose register (e.g., `%eax`)

R_{64} : 64-bit general-purpose register (e.g., `%rax`)

M_{32} : 32-bit memory range

M_{64} : 64-bit memory range

Figura 8: Instrucciones de copia y conversiones [5].

Veamos por ejemplo como inicializar una variable de tipo double (en el registro `xmm0`) con el valor 1.0:

```
movq $1, %rax          # Copiar un 1 entero a rax
cvtsi2sdq %rax, %xmm0 # Convierte el 1 de rax al double 1.0 en xmm0
```

4.2. Operaciones de punto flotante

Las operaciones entre valores de punto flotante siempre involucran dos operandos, el operando fuente puede ser tanto un registro `xmm` como un valor almacenado en memoria. El destino debe ser un registro `xmm`. La Fig. 9 resume las operaciones más utilizadas para simple y doble precisión.

Veamos, con lo que tenemos cómo traducir la siguiente función C:

```
double convert(double t) {
    return t*1.8 + 32;
}
```

Veremos en la Sección 5 que la convención de llamada indica que los argumentos de punto flotante se pasan por los registros `xmm` y el valor de retorno se deja en el registro `xmm0`. Sabiendo esto podemos escribir:

Single	Double	Effect	Description
addss	addsd	$D \leftarrow D + S$	Floating-point add
subss	subsd	$D \leftarrow D - S$	Floating-point subtract
mulss	mulsd	$D \leftarrow D \times S$	Floating-point multiply
divss	divsd	$D \leftarrow D / S$	Floating-point divide
maxss	maxsd	$D \leftarrow \max(D, S)$	Floating-point maximum
minss	minsd	$D \leftarrow \min(D, S)$	Floating-point minimum
sqrtss	sqrtsd	$D \leftarrow \sqrt{S}$	Floating-point square root

Figura 9: Instrucciones de copia y conversiones [4].

```
.global convert
convert: # en xmm0 viene t
    # Carga el valor 1.8 en xmm1
    # La constante es la representacion según IEEE 754 de 1.8
    movq $4610785298501913805, -8(%rsp)
    movsd -8(%rsp), %xmm1
    # Carga el valor 32.0 convirtiendo el valor entero 32 de rax a xmm2
    movq $32, %rax
    cvtsi2sdq %rax, %xmm2
    # xmm0=xmm0*xmm1 => xmm0=t*1.8
    mulsd %xmm1, %xmm0
    # xmmo=xmm0+xmm2 => xmmo=t*1.8+32
    addsd %xmm2, %xmm0
    # como el valor de retorno se escribe en xmm0 hemos terminado
    ret
```

Al igual que con los valores enteros la arquitectura ofrece comparaciones de valores de punto flotante. Las instrucciones de comparación comparan dos valores (haciendo una resta virtual) y prenden las banderas correspondientes en el registro `rflags`. La comparación se comporta como una comparación de datos unsigned (i.e.: conviene utilizar `jae` para saltar por mayor o igual). Además, si los valores son incomparables (alguno es NaN) se prende la bandera PF (Parity Flag). Las instrucciones de comparación en punto flotante son:

`ucomiss` para comparación de precisión simple.

`ucomisd` para comparación de precisión doble.

4.3. Instrucciones *packed* - SIMD

El software para procesamiento de señales multimedia (audio, imágenes, video, etc) muchas veces requiere repetir la misma operación en una gran cantidad de datos, por ejemplo para cada píxel realizar una determinada operación. Por ello, las arquitecturas actuales incluyen lo que se conoce como instrucciones *Single Instruction Multiple Data* (SIMD). Estas instrucciones aplican la misma operación a muchos datos a la vez.

Recordemos que los registros `xmm` son de 128 bits por lo cual éstos pueden alojar 4 valores de precisión simple o 2 de precisión doble o también 16 bytes, 8 words, 4 enteros

de 32 bits o 2 de 64 bits. En la Fig. 10 se ven los distintos valores que puede contener un registro **xmm**.

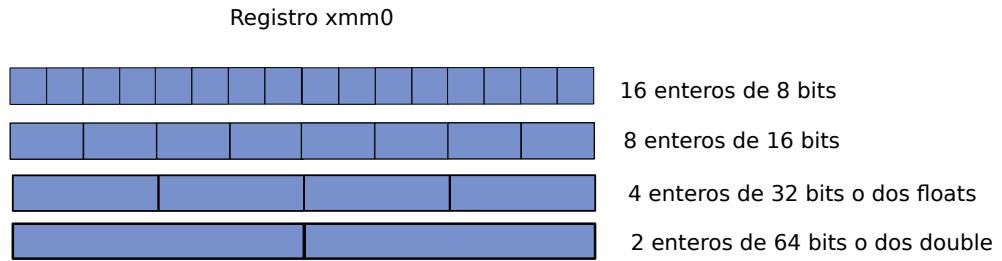


Figura 10: El contenido de un registro **xmm** (128 bits).

Veamos un ejemplo. Dada la siguiente función C:

```
void sum(float a[4], float b[4]) {
    int i;
    for (i=0;i<4;i++)
        a[i]=a[i]+b[i];
}
```

El código equivalente en lenguaje ensamblador es:

```
.global sum
sum:
# notar que en este caso los argumentos
# son punteros y vienen en rdi y rsi respectivamente
# copia los 4 floats de "a" a xmm0
movaps (%rdi), %xmm0
# copia los 4 floats de "b" a xmm1
movaps (%rsi), %xmm1
# suma los 4 floats a la vez
addps %xmm0, %xmm1
# guarda el resultado en "a"
movaps %xmm1, (%rdi)
ret
```

Aquí la instrucción interesante es la **addps** que suma los 4 valores de precisión simple a la vez. La Fig. 11 grafica esta instrucción.

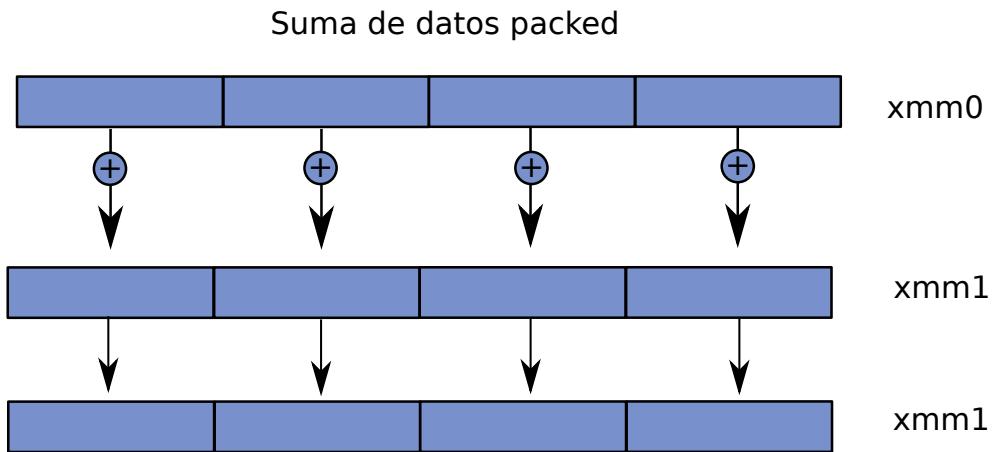


Figura 11: Instrucción addps

Hay varios tipos de instrucciones SSE:

- Instrucciones SSE de Transferencia de datos.
- Instrucciones SSE de Conversión.
- Instrucciones SSE Aritméticas.
- Instrucciones SSE lógicas.

La Tabla 1 muestra algunas instrucciones. Sin embargo, las extensiones SSE contienen muchas más instrucciones. En este apunte sólo se pretende dar una introducción. Un listado completo se puede consultar en [8] o [9]. Por otra parte, en el 2011 se introdujo una nueva tecnología de instrucciones SIMD llamadas AVX, pero estas no serán vistas en este apunte.

5. Funciones y Convención de Llamada

Otra parte fundamental del código estructurado son los procedimientos y funciones. Una función dentro de un programa puede pensarse como una función matemática que se aplica a ciertos valores del dominio y arroja un valor en el conjunto de llegada.

Así vemos por ejemplo que la función C:

```
long int sum(long int a, long int b);
```

tomará dos enteros largos y devolverá otro entero largo.

Desde el punto de vista del procesador una llamada a función es muy similar a un salto ya que el flujo del programa debe ser modificado (para ejecutar el código de la función llamada). La diferencia radica en que, como el código es secuencial, luego de una llamada a función el flujo del programa debe continuar la ejecución **con el código que sigue** a la llamada. Veamos esto en C:

```
...
i++;
printf("%d\n", i);
i--;
...
```

Mnemotécnico	Descripción
movaps	Mueve cuatro flotantes simple precisión alineados entre registros XMM o memoria
movss	Mueve flotantes de simple precisión entre registros XMM o memoria
addps	Suma flotantes simple precisión empaquetados
addss	Suma flotantes simple precisión escalares
divps	Divide flotantes simple precisión empaquetados
divss	Divide flotantes simple precisión escalares
mulps	Multiplica flotantes simple precisión empaquetados
mulss	Multiplica flotantes simple precisión escalares
subps	Resta flotantes simple precisión empaquetados
subss	Resta flotantes simple precisión escalares
cmpss	Compara flotantes simple precisión empaquetados
cmpss	Compara flotantes simple precisión escalares
comiss	Compara flotantes simple precisión escalares y setea banderas en el registro EFLAGS
andnps	Realiza la operación AND NOT bit a bit de flotantes simple precisión empaquetados
andps	Realiza la operación AND bit a bit de flotantes simple precisión empaquetados
orps	Realiza la operación OR bit a bit de flotantes simple precisión empaquetados
xorps	Realiza la operación XOR bit a bit de flotantes simple precisión empaquetados

Tabla 1: Algunas instrucciones SSE.

Aquí vemos tres instrucciones. La segunda es una llamada a la función `printf` con dos argumentos, una cadena de caracteres "%d\n" y el valor de `i`. Luego de finalizada la impresión por parte de `printf` el código debe seguir con el decremento de `i`. Pero ¿cómo sabe `printf` que debe continuar con esa instrucción (siendo que `printf` podría ser llamada de múltiples lugares distintos)? La respuesta es que no lo sabe, sino que **el código que invoca** a esta función debe indicarle adonde continuar la ejecución luego de finalizar la llamada. Esta **dirección** donde debe continuar se conoce como dirección de retorno.

Para realizar llamadas a función, la arquitectura x86-64 provee dos instrucciones:

call Realiza la invocación a la función indicada como operando (la etiqueta que la define) guardando en la pila la dirección de retorno (la dirección de la próxima instrucción al `call`). Así la instrucción `call f` sería equivalente a

```
pushq $direccion_de_retorno
jmp f
direccion_de_retorno:
```

donde la constante `direccion_de_retorno` indica la dirección de la próxima instrucción a la llamada.

ret Retorna de una función sacando el valor de retorno que se encuentra en el tope de la pila (puesto allí por el `call`) y salta a ese lugar. Así la instrucción `ret` equivale a

```
popq %rdi  
jmp *%rdi
```

aunque **ret no modifica** ningún registro (más que el **%rip**). Aquí el asterisco es necesario por la sintaxis.

Cuando las funciones son “llamadas” dentro de un programa se reconocen **dos** actores en cuanto a responsabilidades:

El llamante (*caller*) es la parte de código que invoca a la función en cuestión. El *caller* quiere computar el valor de la función para ciertos valores de argumentos y luego seguir computando con el resultado obtenido.

El llamado (*callee*) es la parte de código que **implementa** la función. Éste debe computar el resultado (valor de retorno) de la función a partir de los argumentos recibidos por el llamante.

5.1. Convención de llamada

Se conoce como convención de llamada al acuerdo previo que tienen estos dos actores (llamante y llamado) sobre cómo invocar funciones, obtener sus resultados y sobre el estado de la máquina previa y posteriormente a la llamada. En lo específico, una convención de llamada describe a nivel de ensamblador:

- Dónde deben ir los argumentos al invocar a una función.
- Dónde quedará el resultado obtenido.
- Qué registros mantendrán su valor luego de la llamada.

La convención de llamada para x86-64 es la siguiente⁵:

- Los seis primeros argumentos a la función son pasados por registro en el siguiente orden: **%rdi, %rsi, %rdx, %rcx, %r8, %r9** (cuando son valores enteros o direcciones de memoria).
- Si son valores de punto flotantes pueden utilizarse hasta 8 de los registros **xmm**: **%xmm0, %xmm1, %xmm2, %xmm3, %xmm4, %xmm5, %xmm6** y **%xmm7**.
- Parámetros grandes mayores a 64 bits, por ejemplo estructuras pasadas por valor, se pasan utilizando la pila.
- Cuando la función toma como argumento una mezcla de valores enteros y flotantes **rdi** será el primer valor entero, **xmm0** el primer valor flotante, y así sucesivamente. Así, en la función **void f(int, double, int, double)** los argumentos irán en **rdi, xmm0, rsi, xmm1**.
- Si hubiera más argumentos éstos son pasados por pila a la función.
- El resultado de la función (si lo hubiera) es devuelto en el registro **%rax** si es entero y sencillo o en el registro **xmm0** si es flotante.

⁵En la Fig. 1 se puede observar el rol de los registros en la llamada a función.

- El llamado **se compromete** a preservar el valor de los registros `%rbx`, `%rbp`, `%rsp`, y `%r12` a `%r15`. Esto no quiere decir que no los pueda usar sino que al retornar deben tener el mismo valor que al comenzar la función. La función podría guardarlos temporalmente en memoria o pila y restaurarlos antes de retornar. Estos registros se conocen como *callee saved* ya que es responsabilidad del llamado preservarlos.

Los otros registros (incluso los de los argumentos) pueden ser modificados libremente por la función sin necesidad de restaurar sus valores. Si el llamante desea preservar sus valores es responsabilidad de él, por lo cual estos registros se conocen como *caller saved*.

- El bit DF de `rflags` está inicialmente apagado (esto incrementará los punteros en instrucciones de manejo de cadena) y debe ser apagado al finalizar la función (y antes de llamar a otra función).
- Como `%rbp` y `%rsp` son preservados durante una llamada a función, el estado de la pila de llamante se mantiene.

Respecto a este último punto (la preservación de la pila), es muy común que cada función demarque el comienzo de **su porción** de pila utilizando el `%rbp`. Como este registro es *calle saved* debe ser preservado por el llamado. Por esta razón es muy común ver porciones llamadas prólogo y epílogo en una función como:

```
#prologo
pushq %rbp      # Guardar el valor del rbp del llamante
movq %rsp, %rbp # La pila para esta función comienza en el tope (vacía)
...
...
...
#epilogo
movq %rbp, %rsp # El tope anterior de la pila es el inicio de la pila actual
popq %rbp       # Retaurar el rbp del llamante
```

Veamos cómo llamaríamos a la función `sum` antes vista con los valores 40 y 45:

```
...
movq $40, %rdi    # el valor del primer argumento es 40 y va en registro rdi
movq $45, %rsi    # el valor del segundo argumento es 45 y va en registro rsi
call sum          # guarda la dirección de retorno en pila y salta a sum
movq %rax, i       # aquí %rax contiene el resultado (85)
...
```

Veamos ahora una posible implementación de `sum`:

```
.global sum        # la etiqueta sum debe ser global
sum:
    # Prologo
    pushq %rbp
    movq %rsp, %rbp

    movq %rdi, %rax # copio el valor del primer arg en %rax
    addq %rsi, %rax # y le sumo el segundo argumento
    # aquí el resultado YA esta en rax
```

```

# Epilogo
movq %rbp, %rsp
popq %rbp

ret           # Tomara el valor de retorno de la pila y saltara a el

```

6. Compilando código ensamblador con GNU as

Un programador puede escribir todo su programa en ensamblador. El único requerimiento es que el código defina una etiqueta global dentro del segmento de código llamada `main`. Una vez escrito el código, el programa puede ser compilado utilizando `gcc`:

```
#gcc sum.s
```

Luego podemos ejecutar mediante:

```
./a.out
```

También podemos usar la opción `-o`:

```
#gcc -o sum sum.s
```

y luego ejecutar mediante:

```
./sum
```

Sin embargo, escribir todo el programa en ensamblador no es la mejor opción. Es mejor escribir solo la parte que necesariamente debe ser escrita en ensamblador (con fines de optimizar, acceder al hardware, etc). Por ello, podemos mezclar código C con ensamblador siempre y cuando el ensamblador respete la convención de llamada vista en la Sección 5. Vemos un ejemplo.

Ejemplo

```

// Este archivo es main.c
#include<stdio.h>
double suma(double a, double b);
int main(){
printf("La suma es: %f\n",suma(12,3.14));
return 0;
}

```

donde la implementación de suma en ensamblador sería

```
// este archivo es suma.s
.global suma
sumas:
    # por convención de llamada
    # el primer argumento viene en xmm0
    # y el segundo en xmm1
    addsd %xmm1, %xmm0
    # el valor de retorno en xmm0
    ret
```

Luego podemos compilar todo junto:

```
#gcc main.c suma.s
```

Luego podemos ejecutar mediante:

```
./a.out
```

y obtendremos el resultado:

```
15.140000
```

El enlazador se encargará de que la llamada a `suma` se corresponda con su implementación en ensamblador.

Referencias

- [1] Andrew S. Tanembaum, *Organización de computadoras: Un enfoque estructurado*, cuarta edición, Pearson Education, 2000
- [2] Paul A. Carter, *PC Assembly Language*, Disponible en formato electrónico: <http://www.drpaulcarter.com/pcasm/>, 2006.
- [3] M. Morris Mano, *Computer system architecture*, tercera edición, Prentice-Hall, 1993.
- [4] Randall Hyde, *The art of assembly language*, segunda edición, No Starch Pr, 2003.
- [5] Randal E. Bryant - David R. O'Hallaron, *x86-64 Machine-Level Programming*, 2005.
- [6] Bryant, Randal E, David Richard, O'Hallaron y David Richard, O'Hallaron, *Computer systems: a programmer's perspective*, Prentice Hall, 2003.
- [7] *AMD64 Architecture Programmer's Manual Volume 1: Application Programming*, AMD64 Technology, 2015.
- [8] *AMD64 Architecture Programmer's Manual Volume 4: 128-Bit and 256-Bit Media Instructions*, AMD64 Technology, 2015.

- [9] *x86 Assembly Language Reference Manual*, Oracle, 2012.
- [10] Miquel Albert Orenga y Gerard Enrique Manonellas, *Programación en ensamblador (x86-64)*, Universitat Oberta de Catalunya (UOC), 2011.
- [11] M. Matz, J. Hubicka, A. Jaeger, M. Mitchell, *System V Application Binary Interface: AMD64 Architecture Processor Supplement*, Draft Version 0.99.7, 2014.

RISC-V

Mariano Street <mstreet@fceia.unr.edu.ar>

Arquitectura del Computador
Licenciatura en ciencias de la computación
Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Universidad Nacional de Rosario



Versión 1.0
19 de noviembre de 2019

Índice general

1. Introducción	3
1.1. ¿Qué es una arquitectura abierta?	3
1.2. ¿Qué es una arquitectura RISC?	3
1.3. ¿Por qué desarrollar una nueva arquitectura?	4
1.4. Breve historia de RISC-V	4
1.5. ¿Por qué enseñar RISC-V?	5
1.6. ¿Quién usa RISC-V en la industria?	5
1.7. ¿Cómo probar RISC-V?	6
1.7.1. Software	6
1.7.2. Hardware	7
2. Arquitectura	8
2.1. Objetivos de RISC-V	8
2.2. Módulos	8
2.2.1. Distintas bases	9
2.2.2. RV32E: para embebidos	10
2.2.3. RV128I: planificación a futuro	10
2.3. Registros	11
2.4. Endianness	12
2.5. Lenguaje de ensamblador	13
2.5.1. Sintaxis general	13
2.5.2. Directivas del ensamblador	14
2.5.3. Instrucciones	16
2.5.4. Etiquetas	16
2.6. Instrucciones básicas	17
2.6.1. Operaciones aritméticas	17
2.6.2. Operaciones lógicas	18
2.6.3. Operaciones de corrimiento	18
2.6.4. Carga/Descarga	19
2.6.5. Saltos	20
2.6.6. Otras	21

2.7. Pila	21
2.8. Convención de llamada	22
2.9. Codificación	23
2.9.1. Tipo R: registro/registro	23
2.9.2. Tipo I: inmediato	23
2.9.3. Tipo U: inmediato superior	23
2.9.4. Tipo S: almacenamiento	23
2.9.5. Tipo B: ramificación	23
2.9.6. Tipo J: salto	24
3. Extensión M: multiplicación y división	25
3.1. ¿Por qué en una extensión?	25
3.2. Instrucciones	26
3.3. División por cero y desborde	27
4. Extensiones F y D: punto flotante	28
4.1. Registros	28
4.2. Redondeo	29
4.3. Instrucciones de F	29
4.3.1. Operaciones	29
4.3.2. Conversión	31
4.3.3. Comparación y clasificación	31
4.3.4. Carga y descarga	32
4.3.5. Control y estado	32
4.4. Instrucciones de D	33
4.4.1. Operaciones	33
4.4.2. Conversión	34
4.4.3. Comparación y clasificación	35
4.4.4. Carga y descarga	35
4.5. Clasificación de números	36
5. Referencia de instrucciones	37
5.1. Instrucciones de I	37
5.2. Instrucciones de M	38
5.3. Instrucciones de F	39
5.4. Instrucciones de D	40
5.5. Pseudoinstrucciones	41
6. Enlaces y libros	43
6.1. Enlaces	43
6.2. Libros	44

Capítulo 1

Introducción

RISC-V es una arquitectura de computadoras abierta y moderna de tipo RISC. Surgió en la academia y hoy en día es desarrollada por la Fundación RISC-V, una asociación de cientos de organizaciones interesadas en esta arquitectura.

1.1. ¿Qué es una arquitectura abierta?

Una arquitectura abierta es un diseño que puede ser usado por cualquier persona para hacer su propio hardware o software. No está atada a una empresa que gobierne su desarrollo y decida a quién permitir usarla y bajo qué condiciones. No hay que pagar regalías, derechos de uso. No se está bajo la amenaza de que el fabricante pueda iniciar un juicio por patentes ante la difusión de un producto basado en la arquitectura.

Esto es algo poco común en la industria de las computadoras e incluso en la academia. Han surgido algunos proyectos o diseños que en un principio eran privativos y luego se liberaron o estandarizaron, pero ninguno había tomado tanto empuje hasta hace unos años. Si se contrasta con la popularidad y el desarrollo que alcanzó en contrapartida el software libre, todavía hay mucho por recorrer en el terreno del hardware libre. RISC-V puede considerarse un paso importante en este sentido.

1.2. ¿Qué es una arquitectura RISC?

RISC son las siglas de *Reduced Instruction Set Computer*. Se trata de un término acuñado en la década de 1980 en torno a desarrollos determinantes que se dieron en EEUU en cuanto al diseño de procesadores.

Las ideas de RISC giran en torno a dos objetivos que van de la mano y se complementan:

1. Sacar el máximo provecho a un “pipeline” de instrucciones.
2. Simplificar el diseño de procesadores.

1.3. ¿Por qué desarrollar una nueva arquitectura?

De acuerdo a lo relatado por los autores de RISC-V, de acuerdo a su experiencia en investigación y enseñanza usando arquitecturas comerciales, identificaron las siguientes desventajas:

- Las arquitecturas comerciales son privativas.
- Las arquitecturas comerciales solo son populares en ciertos dominios de mercado.
- Las arquitecturas comerciales vienen y van.
- Las arquitecturas comerciales populares son complejas.
- Las arquitecturas comerciales por sí solas no son suficientes para levantar aplicaciones.
- Las arquitecturas comerciales populares no se diseñaron para ser extensibles.
- Una arquitectura comercial modificada es una arquitectura nueva.

1.4. Breve historia de RISC-V

Empezó en 2010 como un proyecto académico en UCB, la Universidad de California en Berkeley (EEUU). David Patterson, famoso científico de la computación, quien estuvo en la década de 1980 a la cabeza del proyecto RISC original y quien publicó importantes libros sobre arquitectura de computadoras, fue quien lideró esta nueva iniciativa. Se trata de un sucesor espiritual de aquel RISC, y se lo identifica como la quinta generación, por eso el nombre RISC-V.

En 2011 se publica la versión 1.0 de la especificación de ISA de usuario de RISC-V, también conocida como volumen 1 de la especificación. El ISA privilegiado quedaría para un futuro volumen 2.

En 2014 se publica la versión 2.0 del volumen 1. Esta es la versión final congelada.

En 2015 se arma la Fundación RISC-V con más de 100 miembros. También se organiza la primera conferencia del proyecto, en EEUU.

En 2016, SiFive, empresa estadounidense, lanza el primer chip comercial de RISC-V: el microcontrolador Freedom E310, con un procesador a 320 MHz.

En 2017, SiFive lanza el primer chip RISC-V capaz de arrancar Linux y FreeBSD: U54-MC Coreplex.

En 2018 SiFive lanza la placa de desarrollo HiFive Unleashed, que contiene el chip anterior. También se revelan planes de Western Digital y NVIDIA para adoptar RISC-V. La empresa india Shakti lanza un procesador a 400 MHz capaz de arrancar Linux.

En 2019, la Fundación Linux lanza la Alianza CHIPS, un grupo industrial para fomentar el desarrollo de hardware abierto. Este año también se publica el volumen 2 de la especificación de RISC-V, el ISA privilegiado.

A lo largo de los últimos años, también se han ido adaptando los principales componentes de los sistemas GNU/Linux: el núcleo Linux, el compilador GCC, las herramientas Binutils, la biblioteca Glibc y el gestor de arranque GRUB ya son compatibles con la arquitectura.

1.5. ¿Por qué enseñar RISC-V?

Dos motivos: uno, que tiene una simplicidad tal que lo vuelve idóneo para aprender los conceptos de arquitecturas de computadoras; el otro, que es abierto y promueve el espíritu científico que se busca cultivar en esta carrera.

1.6. ¿Quién usa RISC-V en la industria?

En 2015 se fundó la Fundación RISC-V. De ella forman parte, de acuerdo a su sitio web, 275 organizaciones. Algunos nombres conocidos son: Google, NVIDIA, NXP, Qualcomm, Samsung, Western Digital, Allwinner Technology, Boston University, ETH Zurich, Hewlett Packard Enterprise, Hitachi, Huawei, IBM, Inria, MediaTek, Nokia, Princeton University, Raspberry Pi, Seagate, Siemens, Sony, STMicroelectronics y TSMC.

Se pueden destacar como casos concretos de adopción de RISC-V los siguientes:

Western Digital Lleva desde 2017 transicionando a RISC-V sus productos de procesamiento de datos de alto rendimiento. Desde 2018 tienen desarrollada su propia CPU RISC-V de 32 bits para dispositivos embebidos, SweRV Core, la cual fue publicada como código abierto.

NVIDIA Anunció en 2016 que desarrollaría un controlador RISC-V de 32 bits como sucesor del controlador Falcon que usan en sus GPU desde 2006. En 2019, se informó que también migrarían algunas de sus líneas de productos SoC a RISC-V.

Rambus Lanzó en 2018 núcleos de procesamiento seguro con RISC-V.

Otros Otros fabricantes vienen desarrollando desde aceleradores de aprendizaje profundo hasta controladores SSD, así como placas de desarrollo capaces de arrancar Linux.

Hay varias empresas que fabrican tanto CPU RISC-V como sistemas en chip (SoC) enteros que contienen varios módulos de los cuales uno es una CPU RISC-V. La más famosa es SiFive. También están lowRISC y otras...

<https://riscv.org/risc-v-cores/>

Hay muchas empresas que patrocinan RISC-V.

1.7. ¿Cómo probar RISC-V?

1.7.1. Software

Compilador cruzado

Para compilar, se puede usar GCC. Dado que se lo va a ejecutar en una máquina con otra arquitectura (como puede ser x86_64), es necesario lo que se llama un *compilador cruzado*: una versión de GCC compilada para generar código para una arquitectura distinta que aquella sobre la que corra.

Ubuntu ya trae tal versión en sus repositorios, basta con instalar el paquete `gcc-riscv64-linux-gnu`. Una vez instalado, se lo usa con el comando `riscv64-linux-gnu-gcc`. Los argumentos básicos son los mismos; en algunos casos, resulta conveniente agregar `-static` para generar un binario estático, sin enlaces a bibliotecas dinámicas que pueden no estar instaladas para la arquitectura destino. Por ejemplo:

```
$ riscv64-linux-gnu-gcc -static -o prog prog.c
```

Emulador/Simulador

Para ejecutar programas compilados para RISC-V, hay diversos emuladores y simuladores. QEMU es un emulador muy popular en GNU/Linux para multitud de arquitecturas y que desde hace unos años permite emular RISC-V. Una vez que se tiene un binario del programa a ejecutar, se puede correr `qemu-riscv64` de forma sencilla:

```
$ qemu-riscv64 prog
```

En Ubuntu, a partir de su versión 16.04 LTS Xenial Xerus, está disponible el paquete `qemu-system-misc`. Pero esto solo incluye el emulador de sistema.

Alternativamente, se pueden usar simuladores como rv8 y Spike.

1.7.2. Hardware

El ecosistema de dispositivos RISC-V está en sus primeras etapas. Se espera que surjan muchos productos en los próximos años. Por lo pronto, para desarrollos de propósito general, son de destacar las placas que ofrece SiFive:

HiFive1 Lanzada en 2016, primer producto comercial con un procesador RISC-V, de 32 bits. Compatible con Arduino.

HiFive1 Rev B Lanzada en 2019, versión mejorada de la anterior. Con más puertos, Wi-Fi y Bluetooth, entre otros cambios.

HiFive Unleashed Lanzada en 2018, primera placa de desarrollo que corre Linux. Incluye un SoC de 64 bits desarrollado por SiFive, 8 GiB de RAM, Gigabit Ethernet y ranura para tarjeta microSD. A 2019, su precio es muy prohibitivo.

Capítulo 2

Arquitectura

En esta sección se abordan los principales detalles técnicos de la arquitectura.

2.1. Objetivos de RISC-V

- Apertura.
- Adaptabilidad a muy diversos casos de uso.

2.2. Módulos

RISC-V tiene un diseño modular. Hay un conjunto de instrucciones y registros base, y extensiones que agregan conjuntos de instrucciones adicionales. Cada una de las posibles bases, así como las extensiones, se llaman módulos.

Para referirse a cierto conjunto de módulos, se usa una nomenclatura estandarizada. Esta tiene el formato $RV\langle bits \rangle\langle base \rangle[\langle exts \rangle]$, donde $\langle bits \rangle$ es el tamaño de palabra de la arquitectura (32, 64 o 128 bits), $\langle base \rangle$ es una letra que indica el perfil básico (I o E) y $\langle exts \rangle$, que es opcional, es una secuencia de letras que indica las extensiones. Cada extensión lleva una letra mayúscula, opcionalmente seguida de letras minúsculas. G es un caso especial, es una abreviatura de la secuencia de extensiones más común: MAFDZicsrZifencei. Ejemplos de cadenas completas: RV32I, RV64IGB, RV32ECZtso.

A continuación se listan los módulos oficiales.

Base	Descripción	Estado
RVWMO	–	ratificado
RV32I	base, enteros de 32 bits	ratificado
RV32E	base, enteros de 32 bits, para embebidos	borrador
RV64I	base, enteros de 64 bits	ratificado
RV128I	base, enteros de 128 bits	borrador
Extensión	Descripción	Estado
Zicsr	–	ratificado
Zifencei	–	ratificado
M	extensión para multiplicación y división entera	ratificado
A	extensión para instrucciones atómica	congelado
F	extensión para punto flotante de precisión simple	ratificado
D	extensión para punto flotante de precisión doble	ratificado
G	abreviatura para las seis extensiones anteriores	–
Q	extensión para punto flotante de precisión cuádruple	ratificado
L	extensión para punto flotante decimal	borrador
C	extensión para instrucciones comprimidas	ratificado
B	extensión para manipulación de bits	borrador
J	extensión para lenguajes traducidos dinámicamente	borrador
T	extensión para memoria transaccional	borrador
P	extensión para instrucciones de SIMD empaquetado	borrador
V	extensión para operaciones vectoriales	borrador
N	extensión para interrupciones de nivel de usuario	borrador
Ztso	–	congelado
Zam	–	borrador
Counters	–	borrador

¿Cuál estudiaremos nosotros? RV64IMFD.

2.2.1. Distintas bases

Como muestra la tabla anterior, hay 4 bases disponibles en RISC-V: RV32I, RV32E, RV64I y RV128I. ¿Qué diferencias hay entre ellas?

La primera diferencia es el tamaño de los registros enteros y las direcciones de memoria: 32 bits en el caso de RV32I y RV32E, 64 bits en RV64I y la poco común cifra de 128 bits en RV128I.

Las bases con el módulo I cuentan con 32 registros enteros, que se presentarán en la sección siguiente. Lo distintivo de E, por su parte, es que solo incluye los primeros 16 registros.

2.2.2. RV32E: para embebidos

La base RV32E difiere de RV32I en exactamente una cosa: la cantidad de registros se reduce de 32 a 16. Todas las instrucciones son las mismas.

¿Cuál es la importancia de esta base? Los registros resultan ser una buena parte del circuito de una CPU. Muchos sistemas embebidos son lo suficientemente pequeños como para que no se justifique el costo de tener 32 registros en el procesador. Al reducir la cantidad a la mitad, se logra una reducción importante en el tamaño del chip y todo lo que eso acarrea: consumo de energía, generación de calor y precio.

2.2.3. RV128I: planificación a futuro

El manual ofrece la siguiente cita:

Hay solo un error que se puede cometer en diseño de computadoras del cual es difícil de recuperarse – no tener suficientes bits de direcciones para direccionar y gestionar memoria.

– Bell and Strecker, ISCA-3, 1976.

Y luego presenta la motivación para la confección de esta base de la siguiente manera:

La razón principal para extender el ancho de registros enteros es soportar espacios de direcciones más grandes. No está claro cuándo se requerirá un espacio de direcciones plano de más de 64 bits. Al momento de escribir esto, la supercomputadora más rápida del mundo de acuerdo a lo medido por el benchmark Top500 tenía más de 1 PB de DRAM, y requeriría más de 50 bits de espacio de direcciones si toda la DRAM residiera en un único espacio de direcciones. Algunas memorias a escala *warehouse* y tecnologías de interconexión rápida podrían impulsar una demanda de espacios de memoria incluso más grandes. La investigación de sistemas de exaescala está apuntando a sistemas de 100 PB de memoria, los cuales ocupan 57 bits de espacio de direcciones. A las tasas de crecimiento históricas, es posible que se requieran espacios de direcciones de más de 64 bits antes de 2030.

La historia sugiere que en cuanto quede claro que se necesitarán más de 64 bits de espacio de direcciones, los arquitectos repetirán intensivos debates sobre alternativas a extender el espacio de direcciones, incluyendo segmentación, espacios de direcciones de 96 bits y paliativos por software, hasta que, finalmente, se adopten

espacios de direcciones de 128 bits como la solución más simple y mejor.

2.3. Registros

RISC-V, en sus módulos base I (RV32I, RV64I y RV128I), cuenta con 32 registros enteros de propósito general. Todos ellos tienen el mismo tamaño, que puede ser 32, 64 o 128 bits dependiendo de la variante de la arquitectura que se use.

De forma resumida, los registros son los siguientes:

zero: siempre cero.

ra: dirección de retorno.

sp: puntero de pila.

gp: puntero global.

tp: puntero de hilo.

t0–t6: temporales.

s0–s11: preservados (puntero de marco en el primero).

a0–a7: argumentos (valor de retorno en los dos primeros).

Estos nombres indican el uso que se da a cada registro, usos que están determinados en su mayor parte por la convención de llamada, con la excepción de **zero**, que está diseñado por hardware para que siempre contenga cero. Adicionalmente, los registros también van numerados de 0 a 31 y se los puede nombrar usando el número asociado más el prefijo **x**. A nivel técnico es lo mismo usar una nomenclatura o la otra; la primera tiene una evidente ventaja en claridad, mientras que la segunda es independiente de la convención de llamada; nosotros usaremos siempre la primera: los nombres funcionales.

La siguiente tabla indica todos los detalles sobre cada registro:

Nombre	Nombre funcional	Función	Tipo de uso
x0	zero	Siempre cero	—
x1	ra	Dirección de retorno	temporal
x2	sp	Puntero de pila	preservado
x3	gp	Puntero global	—
x4	tp	Puntero de hilo	—
x5	t0	Temporal / Dirección de retorno alternativa	temporal
x6–x7	t1–t2	Temporal	llamante
x8	s0/fp	Preservado / Puntero de marco	preservado
x9	s1	Preservado	preservado
x10–x11	a0–a1	Argumento de función / Valor de retorno	temporal
x12–x17	a2–a7	Argumento de función	temporal
x18–x27	s2–s11	Preservado	preservado
x28–x31	t3–t6	Temporal	temporal

En cuanto al tipo de uso, en la tabla se emplean los nombres temporal y preservado. Con los primeros se refiere a los registros preservados por la función llamante (“caller saved”) y los segundos, por la llamada (“callee saved”). Véase la sección *Convención de llamada* para más información.

¿Por qué no son contiguas las asignaciones de registros temporales y preservados? Ciertos detalles de RISC-V llevan a cuestionarse por qué no se optó por un ordenamiento más simple. Más aun siendo que no tiene retrocompatibilidad que mantener, como pasa en otras arquitecturas. La respuesta está en la adaptabilidad de RISC-V: como se indicó en una sección anterior, hay una variante E de la arquitectura que, en contraste con la I, incluye solamente los primeros 16 registros. Al reducir la cantidad de registros, es deseable en lo posible mantener las distinciones de finalidad (temporales, preservados, etc.). Entonces hay que distribuir los 32 registros de forma que en la primera mitad haya de todos los tipos en alguna proporción, y que la segunda mitad amplíe esas proporciones.

2.4. Endianness

RISC-V es siempre little-endian. Es decir, los bytes menos significativos se codifican en memoria en las direcciones más bajas. Por ejemplo, el entero 0x12345678, de 4 bytes, se codifica así:

...	0x78	0x56	0x34	0x12	...
-----	------	------	------	------	-----

2.5. Lenguaje de ensamblador

Como toda arquitectura, RISC-V tiene su propio lenguaje de ensamblador. Un programa llamado ensamblador lee archivos en este lenguaje y genera código máquina. Los archivos llevan la extensión `.s` o `.S`. La segunda, con mayúscula, indica que se debe pasar el preprocesador de C antes de ensamblar: esto permite incluir directivas como `#include` y `#define`.

2.5.1. Sintaxis general

La sintaxis de ensamblador para RISC-V se caracteriza por ser limpia, en el sentido de que prescinde de adornos allá donde sea posible. Que el conjunto de instrucciones sea simple y ortogonal facilita llevar a cabo tal diseño.

Véanse algunos ejemplos. Esta es la implementación de una función que suma dos números enteros y devuelve el resultado. Se incluyen comentarios que describen cada instrucción:

```
.text
.global sumar
sumar:
    add a0, a0, a1      # a0 := a0 + a1
    ret                  # Retorno de la función
.global main
main:
    addi sp, sp, -16    # Se reserva espacio en la pila
    sd ra, (sp)         # Se guarda dirección de retorno
    li a0, 4             # Primer argumento para 'sumar'
    li a1, 5             # Segundo argumento para 'sumar'
    call sumar          # Se llama a 'sumar'
    ld ra, (sp)          # Se restaura dirección de retorno
    addi sp, sp, 16       # Se libera espacio en la pila
    ret
```

En la variante que sigue, la función toma un arreglo de enteros de 4 bytes como argumento y retorna la suma de los primeros dos elementos:

```
.rodata
valores: .word 3
          .word 4
```

```

.text
.global sumar
sumar:
    lw  t0, (a0)
    lw  t1, 4(a0)
    add a0, t0, t1
    jr  ra
    .global main
main:
    addi sp, sp, -16
    sd  ra, (sp)
    la  a0, valores # Argumento para 'sumar'
    call sumar
    ld  ra, (sp)
    addi sp, sp, 16
    jr  ra

```

Características:

- Los operandos de destino van a la izquierda, los de fuente a la derecha.
- Las instrucciones de operación usan operandos separados para destino y fuente. Lo más común es el formato de 3 operandos: destino, fuente, fuente.
- Las instrucciones no llevan adornos, como pueden ser sufijos de tamaño. Toda la información requerida está indicada en el nombre de la instrucción misma.
- Los registros no llevan adornos, es decir que no se usa un prefijo para distinguirlos, basta con su nombre.
- Los literales de enteros no llevan adornos, es decir que no se usa un prefijo para distinguirlos, basta con indicar el número.
- Para direccionar memoria a partir de registros, hay una única sintaxis: `offset(reg)` donde `offset` es un entero opcional con signo y `reg` es un registro. Se suma `offset` al valor contenido en `reg`.

2.5.2. Directivas del ensamblador

Son las líneas que empiezan con punto. No se trata de instrucciones, sino indicaciones que se le dan al programa ensamblador para que genere el binario

ejecutable de cierta forma.

Directivas para emisión de datos y control de alineación:

.byte N...

Emitir un byte.

.half N...

Emitir media palabra (2 bytes), con alineación natural.

.word N...

Emitir una palabra (4 bytes), con alineación natural.

.dword N...

Emitir doble palabra (8 bytes), con alineación natural.

.2byte N...

Emitir valor de 2 bytes sin alinear.

.4byte N...

Emitir valor de 4 bytes sin alinear.

.8byte N...

Emitir valor de 8 bytes sin alinear.

.string C

Emitir cadena de caracteres, terminada en carácter nulo.

.asciz C

Alias para **.string**.

.zero N

Emitir una cantidad de ceros.

.align N

Alinear a potencia de 2.

Directivas para control de símbolos y secciones:

.globl S

Emitir símbolo global.

.equ S, V

Definir constante. Ejemplo:

```
.equ MAGIC_NUMBER, 0x40302040
```

```

.text
    Emitir sección de código y hacerla la actual.

.data
    Emitir sección de datos inicializados y hacerla la actual.

.rodata
    Emitir sección de datos inicializados de solo lectura y hacerla la actual.

.bss
    Emitir sección de datos no inicializados y hacerla la actual.

.section X
    Emitir la sección indicada y hacerla la actual.

Directivas varias:

.option 0...
    Opciones específicas de RISC-V.

.macro A...
    Empezar definición de macro.

.endm
    Terminar definición de macro.

```

2.5.3. Instrucciones

Las instrucciones son la primera palabra de una línea, descartando la posible etiqueta, y no empiezan con punto. El conjunto de instrucciones disponible variará dependiendo de la variante de RISC-V que se utilice. Como mínimo, sin embargo, las instrucciones básicas, definidas por el módulo I o E, estarán disponibles.

La sección *Instrucciones básicas* de este capítulo describe cada instrucción del módulo I. Capítulos siguientes describen las de los módulos M, F y D.

2.5.4. Etiquetas

Una etiqueta es un nombre legible que define una posición en alguna sección del programa; posteriormente será traducido a una dirección de memoria. Se define una etiqueta indicándola inmediatamente antes de la instrucción o directiva cuya posición va a indicar, con su nombre y el carácter dos puntos como sufijo. Puede ir en la misma línea o en alguna anterior.

2.6. Instrucciones básicas

Solo veremos las instrucciones de nivel de usuario, no las privilegiadas. Hay un estándar separado para cada una: el volumen I y el volumen II del manual de referencia, respectivamente.

Antes que nada, es conveniente tener en cuenta los distintos tipos de operando que puede tomar una instrucción. Son los siguientes:

Registro El nombre de un registro. Indica que se va a operar con el valor contenido en este.

Inmediato Un entero indicado como constante en la instrucción misma; puede ser un literal, por ejemplo el número -12, o puede ser una etiqueta. Se opera directamente con ese valor literal en el primer caso, o con la dirección de memoria que se vaya a asociar a una etiqueta, en el segundo.

Dirección de memoria Un operando con la sintaxis de direccionamiento de memoria (`offset(reg)`). Se obtiene el valor del registro `reg`, se le suma `offset` y el resultado se usa como operando.

Algo fundamental de las arquitecturas RISC es que cada función toma ciertos tipos específicos de operando. No es posible pasar una dirección de memoria donde se espera un registro: para cada caso hay que usar una instrucción distinta.

Las mayoría de instrucciones aritméticas y lógicas llevan el formato de 3 registros: uno de salida, dos de entrada. Estos se notarán con los nombres `rd`, `rs1` y `rs2`. Aquellas que para la segunda fuente tomen en cambio un inmediato, lo notan con `imm`.

2.6.1. Operaciones aritméticas

`add rd, rs1, rs2 (add)`

Sumar registros.

`addi rd, rs1, imm (add immediate)`

Sumar un registro y un inmediato (de 12 bits).

`sub rd, rs1, rs2 (subtract)`

Restar registros.

`lui rd, imm (load upper immediate)`

Cargar inmediato (de 20 bits) en la parte superior de un registro.

auipc rd, imm (*add upper immediate to PC*)

Sumar inmediato (de 20 bits) a la superior del contador del programa (PC) y guardar resultado en registro.

En RV64I:

addw rd, rs1, rs2 (*add as word*)

Como add pero opera sobre palabras (4 bytes).

addiw rd, rs1, imm (*add immediate as word*)

Como addi pero opera sobre palabras (4 bytes).

subw rd, rs1, imm (*subtract as word*)

Como sub pero opera sobre palabras (4 bytes).

2.6.2. Operaciones lógicas

xor rd, rs1, rs2

Calcular xor (disyunción exclusiva) sobre registros.

xori rd, rs1, imm

Calcular xor (disyunción exclusiva) sobre un registro y un inmediato.

or rd, rs1, rs2

Calcular or (disyunción inclusiva) sobre registros.

ori rd, rs1, imm

disyunción inclusiva de inmediato.

and rd, rs1, rs2

Calcular and (conjunción) sobre registros.

andi rd, rs1, imm

Calcular and (conjunción) sobre un registro y un inmediato (12 bits).

2.6.3. Operaciones de corrimiento

sll rd, rs1, rs2 (*shift left logical*)

Calcular corrimiento lógico a izquierda sobre registros.

slli rd, rs1, imm (*shift left logical immediate*)

Calcular corrimiento lógico a izquierda sobre un registro y un inmediato.

srl rd, rs1, rs2 (*shift right logical*)

Calcular corrimiento lógico a derecha.

srlti rd, rs1, imm (*shift right logical immediate*)

Calcular corrimiento lógico a derecha sobre un registro y un inmediato.

sra rd, rs1, rs2 (*shift right arithmetic*)

Calcular corrimiento aritmético a derecha sobre registros.

srai rd, rs1, imm (*shift right arithmetic immediate*)

Calcular corrimiento aritmético a derecha sobre un registro y un inmediato.

En RV64I:

sllw rd, rs1, rs2 (*shift left logical as word*)

Como **sll** pero opera sobre palabras (4 bytes).

slliw rd, rs1, imm (*shift left logical immediate as word*)

Como **slli** pero opera sobre palabras (4 bytes).

srlw rd, rs1, rs2 (*shift right logical as word*)

Como **srl** pero opera sobre palabras (4 bytes).

srlwi rd, rs1, imm (*shift right logical immediate as word*)

Como **srlti** pero opera sobre palabras (4 bytes).

sraw rd, rs1, rs2 (*shift right arithmetic as word*)

Como **sra** pero opera sobre palabras (4 bytes).

sraiw rd, rs1, imm (*shift right arithmetic as word*)

Como **srai** pero opera sobre palabras (4 bytes).

2.6.4. Carga/Descarga

lb rd, mem (*load byte*)

Copiar byte de memoria a registro.

lbu rd, mem (*load byte unsigned*)

Copiar byte de memoria a registro, extender con ceros.

lh rd, mem (*load halfword*)

Copiar media palabra (2 bytes) de memoria a registro, extender con signo.

lhu rd, mem (*load halfword unsigned*)

Copiar media palabra (2 bytes) de memoria a registro, extender con ceros.

lw rd, mem (*load word*)

Copiar palabra (4 bytes) de memoria a registro, extender con signo si es necesario.

sb rs1, mem (*store byte*)

Copiar byte de registro a memoria.

sh rs1, mem (*store halfword*)

Copiar media palabra (2 bytes) de registro a memoria.

sw rs1, mem (*store word*)

Copiar palabra (4 bytes) de registro a memoria.

En RV64I:

lwu rd, mem (*load word unsigned*)

Copiar palabra (4 bytes) de memoria a registro, extender con ceros.

ld rd, mem (*load doubleword*)

Copiar doble palabra (8 bytes) de memoria a registro.

sd rs1, mem (*load doubleword*)

Copiar doble palabra (8 bytes) de memoria a registro.

2.6.5. Saltos

jal rd, imm (*jump and link*)

Saltar a dirección inmediata y guardar dirección de retorno en registro.

jalr rd, rs1, imm (*jump and link register*)

Saltar a dirección de memoria con registro y offset y guardar dirección de retorno en registro.

beq rs1, rs2, imm (*branch if equal*)

Saltar si es igual.

bne rs1, rs2, imm (*branch if not equal*)

Saltar si es distinto.

blt rs1, rs2, imm (*branch if less than*)

Saltar si es menor, con signo.

bltu rs1, rs2, imm (*branch if less than, unsigned*)
Saltar si es menor, sin signo.

bge rs1, rs2, imm (*branch if greater than or equal*)
Saltar si es mayor o igual, con signo.

bgeu rs1, rs2, imm (*branch if greater than or equal, unsigned*)
Saltar si es mayor o igual, con signo.

2.6.6. Otras

slt rd, rs1, rs2 (*set if less than*)
Comparar dos registros por menor, con signo, y guardar resultado (0 o 1) en registro.

slti rd, rs1, imm (*set if less than immediate*)
Comparar un registro y un inmediato por menor, con signo, y guardar resultado (0 o 1) en registro.

sltu rd, rs1, rs2 (*set if less than, unsigned*)
Comparar dos registros por menor, sin signo, y guardar resultado (0 o 1) en registro.

sltiu rd, rs1, imm (*set if less than immediate, unsigned*)
Comparar un registro y un inmediato por menor, sin signo, y guardar resultado (0 o 1) en registro.

fence ...
Barrera para sincronizar hilos. Ordena accesos a memoria y dispositivos. *No la usaremos.*

ecall (*environment call*)
Llamada al entorno de ejecución (llamada a sistema). *No la usaremos.*

ebreak (*environment break*)
Interrupción al entorno de ejecución (punto de ruptura del depurador).
No la usaremos.

2.7. Pila

Como en todas las arquitecturas de CPU modernas, en RISC-V cada proceso tiene una pila: un segmento dinámico de la memoria principal donde se van guardando los marcos de activación de cada llamada a función.

La pila crece hacia abajo: los nuevos elementos se van agregando en direcciones de memoria más bajas que las anteriores. El registro **sp** se llama puntero de pila (“stack pointer”) y es el que determina el tope de la pila: contiene la dirección del último elemento agregado.

Al ser de tipo RISC, esta arquitectura no ofrece instrucciones específicas para manipular la pila. En cambio, se usan la suma y resta para cambiar el tope, alterando el registro **sp**, y las instrucciones generales de carga y descarga para leer y escribir en la memoria.

Opcionalmente, uno puede usar, además de **sp**, un puntero de marco (“frame pointer”): un registro que apunte al principio del marco de activación actual. La convención es usar para esto el registro **s0**, al cual también se lo llama **fp**.

2.8. Convención de llamada

- Los primeros 8 argumentos enteros van en registros: **a0..a7**.
- Los primeros 8 argumentos flotantes van en registros: **fa0..fa7**.
- Los argumentos adicionales van en la pila.
- Las estructuras se desmembran y sus partes se colocan en registros o pila de acuerdo a sus tipos y cantidades, a grandes rasgos como si fueran varios argumentos.
- El valor de retorno entero va en el registro **a0** y, de ser necesario más tamaño, en el par **a0, a1**.
- El valor de retorno flotante va en el registro **fa0** y, de ser necesario más tamaño, en el par **fa0, fa1**.
- Registros temporales enteros: **ra, a0..a7, t0..t6**.
- Registros temporales flotantes: **fa0..fa7, ft0..ft11**.
- Registros preservados enteros: **sp, s0..s11**.
- Registros preservados flotantes: **fs0..fs11**.
- El puntero de pila siempre va alineado a 16 bytes.

2.9. Codificación

Las instrucciones que uno escribe textualmente en lenguaje de ensamblador deben luego traducirse a código máquina, ejecutable por una CPU. ¿Cómo es la codificación que adquieren en código máquina?

Por defecto, todas las instrucciones en RISC-V se codifican en 32 bits. Dentro de estos 32 bits, se distribuyen los distintos datos que debe indicar cada instrucción. De acuerdo a cuáles sean estos datos, las instrucciones adoptan unos formatos u otros; hay 6 formatos distintos para las instrucciones base, cada una emplea uno de estos.

La extensión C agrega codificaciones alternativas de 16 bits, lo cual reduce el tamaño de los programas y es importante en sistemas embebidos donde el espacio escasea. Por otro lado, la arquitectura incorpora previsiones para el posible agregado de instrucciones de más de 32 bits en un futuro.

A continuación se presenta cada formato base de 32 bits.

2.9.1. Tipo R: registro/registro

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

2.9.2. Tipo I: inmediato

31 – 20	19 – 15	14 – 12	11 – 7	6 – 0
imm[11:0]	rs1	funct3	rd	opcode

2.9.3. Tipo U: inmediato superior

31 – 12	11 – 7	6 – 0
imm[31:12]	rd	opcode

2.9.4. Tipo S: almacenamiento

31 – 25	24 – 20	19 – 15	14 – 12	11 – 7	6 – 0
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode

2.9.5. Tipo B: ramificación

31	30 – 25	24 – 20	19 – 15	14 – 12	11 – 8	7	6 – 0
[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	[11]	opcode

2.9.6. Tipo J: salto

31 [20]	30 – 21 imm[10:1]	20 [11]	19 – 12 imm[19:12]	11 – 7 rd	6 – 0 opcode
------------	----------------------	------------	-----------------------	--------------	-----------------

Capítulo 3

Extensión M: multiplicación y división

La extensión M agrega instrucciones de multiplicación y división de enteros. Emplea los mismos registros enteros de I.

3.1. ¿Por qué en una extensión?

¿Por qué las operaciones de multiplicación y división vienen en una extensión? ¿No son lo suficientemente básicas como para merecer ser incluidas en el conjunto básico, I, de la misma forma que la suma y la resta?

Resulta que a nivel de hardware, implementar estas operaciones no es tan simple como implementar suma y resta. En procesadores embebidos, ciertos diseños limitados podrían directamente prescindir de las mismas: se trata de una decisión de diseño del procesador que reduce directamente la cantidad de transistores, el consumo de energía y la generación de calor. En cuanto al software, hay casos en que se sabe de antemano que no necesitarán multiplicar y dividir; pero si no es así, siempre se puede relegar estas operaciones a implementaciones por software (es decir, escribir funciones con los algoritmos de multiplicación y división en base a la suma y la resta), que serán más lentas que instrucciones nativas pero funcionarán.

En resumen, son caras y no siempre hacen falta.

Así lo justifica el manual de la arquitectura:

Separamos la multiplicación y la división enteras de lo que es la base para simplificar implementaciones de baja gama, o aplicaciones donde las operaciones de multiplicación y división enteras o bien sean infrecuentes o bien se manejen mejor en aceleradores añadidos.

3.2. Instrucciones

mul rd, rs1, rs2 (*multiply*)

Multiplicar.

mulh rd, rs1, rs2 (*multiply upper half*)

Multiplicar mitad superior.

mulhsu rd, rs1, rs2 (*multiply upper half, signed × unsigned*)

Multiplicar mitad superior, con signo × sin signo.

mulhu rd, rs1, rs2 (*multiply upper half, unsigned*)

Multiplicar mitad superior, sin signo.

div rd, rs1, rs2 (*divide*)

Dividir y obtener cociente.

divu rd, rs1, rs2 (*divide unsigned*)

Dividir sin signo y obtener cociente.

rem rd, rs1, rs2 (*remainder*)

Dividir y obtener resto.

remu rd, rs1, rs2 (*remainder unsigned*)

Dividir sin signo y obtener resto.

En RV64M:

mulw rd, rs1, rs2 (*multiply as word*)

Como **mul** pero opera sobre palabras (4 bytes).

divw rd, rs1, rs2 (*divide as word*)

Como **mul** pero opera sobre palabras (4 bytes).

divuw rd, rs1, rs2 (*divide unsigned as word*)

Como **mul** pero opera sobre palabras (4 bytes).

remw rd, rs1, rs2 (*remainder as word*)

Como **mul** pero opera sobre palabras (4 bytes).

remuw rd, rs1, rs2 (*remainder unsigned as word*)

Como **mul** pero opera sobre palabras (4 bytes).

3.3. División por cero y desborde

En algunas arquitecturas, la división por cero genera una excepción, se interrumpe el flujo de ejecución para pasar a ejecutar código dedicado al tratamiento de esta situación anormal. Lo mismo puede ocurrir con el desborde en la división con signo, cuando el mínimo negativo se multiplica por -1.

RISC-V, en cambio, no tiene instrucciones aritméticas que generen excepciones. La división no es la excepción. En lugar de esto, hay resultados predefinidos para cuando ocurre alguna de las circunstancias mencionadas.

En el caso de la división por cero, el cociente resulta en la secuencia de todos bits en 1 ($2^L - 1$ sin signo, -1 con signo), y el resto por su parte resulta en el valor del divisor.

En el caso del desborde en la división con signo, el cociente resulta en -2^{L-1} y el resto en 0.

Capítulo 4

Extensiones F y D: punto flotante

Las extensiones F, D y Q de RISC-V incorporan manejo nativo de números en punto flotante. Agregan un conjunto propio de registros y diversas instrucciones. Son compatibles con el estándar IEEE 754.

Las tres extensiones se diferencian en el tamaño de flotante que permiten manejar: F (“float”) habilita flotantes de precisión simple (32 bits), D (“double”) los habilita precisión doble (64 bits) y Q (“quad”), de precisión cuádruple (128 bits).

En este material se presenta solo el contenido de las extensiones F y D. No tendremos en cuenta Q debido a que no está tan extendida, el GCC que usamos no la emplea por defecto y no aporta algo nuevo a nivel pedagógico: sus agregados son análogos a los de F y D.

4.1. Registros

Se agregan 33 registros: 32 de punto flotante (`f0 .. f31`) y uno de estado y control (`fcsr`).

Los 32 registros flotantes solo pueden almacenar números en punto flotante, nunca enteros. Hay instrucciones para trasladar valores de un conjunto de registros al otro, ya sea convirtiendo el número en el proceso o dejando la secuencia de bits tal cual.

El registro `fcsr` no se accede de forma directa sino con instrucciones dedicadas. Consta de dos campos: uno indica el modo de redondeo y el otro, banderas de excepción acumuladas. Se puede leer y escribir tanto el registro entero como cada campo por separado.

El campo de redondeo contiene 3 bits y permite establecer el redondeo

dinámico. Véase la sección *Redondeo* para más información.

El campo de banderas contiene 5 bits, que indican las siguientes banderas:

Bit	Bandera	Significado
0	NX	Inexacto
1	UF	<i>Underflow</i>
2	OF	Desborde (<i>Overflow</i>)
3	DZ	División por cero
4	NV	Operación inválida

4.2. Redondeo

Se permiten distintos modos de redondeo. Hay dos maneras de seleccionarlos. Una es el redondeo estático, determinado por un campo contenido en cada instrucción que indica el modo que esta debe usar. Otra es redondeo dinámico, que lo determina el valor del campo correspondiente de `fcsr`.

En ambos casos, el modo se indica con 3 bits, que combinados permiten las siguientes opciones:

Secuencia	Nombre	Significado
000	RNE	Redondear al más cercano, desempata el par
001	RTZ	Redondear a cero
010	RDN	Redondear para abajo (hacia $-\infty$)
011	RUP	Redondear para arriba (hacia $+\infty$)
100	RMM	Redondear al más cercano, desempata el de mayor magnitud
101	—	<i>Reservado para uso futuro</i>
110	—	<i>Reservado para uso futuro</i>
111	DYN	En instrucción: selección dinámica; en <code>fcsr</code> : <i>inválido</i>

4.3. Instrucciones de F

4.3.1. Operaciones

`fadd.s rd, rs1, rs2`

Sumar dos flotantes simples.

`fsub.s rd, rs1, rs2`

Restar dos flotantes simples.

`fmul.s rd, rs1, rs2`

Multiplicar dos flotantes simples.

fdiv.s rd, rs1, rs2
Dividir dos flotantes simples.

fsqrt.s rd, rs1
Calcular la raíz cuadrada de un flotante simple.

fmin.s rd, rs1, rs2
Calcular el mínimo entre dos flotantes simples.

fmax.s rd, rs1, rs2
Calcular el máximo entre dos flotantes simples.

fmadd.s rd, rs1, rs2, rs3
Multiplicar dos flotantes simples y sumarles otro.
 $rd := rs1 \times rs2 + rs3$

fmsub.s rd, rs1, rs2, rs3
Multiplicar dos flotantes simples y restarles otro.
 $rd := rs1 \times rs2 - rs3$

fnmsub.s rd, rs1, rs2, rs3
Multiplicar dos flotantes simples, negar producto y sumarles otro.
 $rd := -(rs1 \times rs2) + rs3$

fnmadd.s rd, rs1, rs2, rs3
Multiplicar dos flotantes simples, negar producto y restarles otro.
 $rd := -(rs1 \times rs2) - rs3$

fsgnj.s rd, rs1, rs2
Inyectar signo en precisión simple. Tomar de $rs1$ todos los bits salvo el de signo, y de $rs2$ tomar el de signo.

fsgnjn.s rd, rs1, rs2
Inyectar signo negado en precisión simple. Tomar de $rs1$ todos los bits salvo el de signo, y de $rs2$ tomar el de signo negado.

fsgnjx.s rd, rs1, rs2
Inyectar signo con xor en precisión simple. Tomar de $rs1$ todos los bits salvo el de signo, y para el signo calcular el xor de los signos de $rs1$ y $rs2$.

4.3.2. Conversión

fcvt.w.s rd, rs1

Convertir flotante simple a palabra entera (4 bytes) con signo. *rd* debe ser un registro entero y *rs1* un registro flotante.

fcvt.wu.s rd, rs1

Convertir flotante simple a palabra entera (4 bytes) sin signo. *rd* debe ser un registro entero y *rs1* un registro flotante.

fcvt.s.w rd, rs1

Convertir palabra entera (4 bytes) con signo a flotante simple. *rd* debe ser un registro flotante y *rs1* un registro entero.

fcvt.s.wu rd, rs1

Convertir palabra entera (4 bytes) sin signo a flotante simple. *rd* debe ser un registro flotante y *rs1* un registro entero.

En RV64F:

fcvt.l.s rd, rs1

Convertir flotante simple a doble palabra entera (8 bytes) con signo. *rd* debe ser un registro entero y *rs1* un registro flotante.

fcvt.lu.s rd, rs1

Convertir flotante simple a doble palabra entera (8 bytes) sin signo. *rd* debe ser un registro entero y *rs1* un registro flotante.

fcvt.s.l rd, rs1

Convertir doble palabra entera (8 bytes) con signo a flotante simple. *rd* debe ser un registro flotante y *rs1* un registro entero.

fcvt.s.lu rd, rs1

Convertir doble palabra entera (8 bytes) sin signo a flotante simple. *rd* debe ser un registro flotante y *rs1* un registro entero.

4.3.3. Comparación y clasificación

feq.s rd, rs1, rs2

Comparar flotantes simples por igualdad y guardar el resultado en un registro. Si cualquier operando fuente es *NaN*, el resultado es 0.

flt.s rd, rs1, rs2

Comparar flotantes simples por menor y guardar el resultado en un registro. Si cualquier operando fuente es *Nan*, el resultado es 0. Comparar por menor.

fle.s rd, rs1, rs2

Comparar flotantes simples por menor o igual y guardar el resultado en un registro. Si cualquier operando fuente es *Nan*, el resultado es 0.

fclass.s rd, rs1

Clasificar flotante simple según la clase de valor. Asigna un entero que indica si es normalizado, desnormalizado, cero, infinito o *Nan* (véase la sección *Clasificación de números* para más información). *rd* debe ser un registro entero y *rs1* un registro flotante.

4.3.4. Carga y descarga

flw rd, mem

Copiar flotante simple de memoria a registro. *rd* debe ser un registro flotante.

fsw rs1, mem

Copiar flotante simple de registro a memoria. *rs1* debe ser un registro flotante.

fmv.x.w rd, rs1

Copiar secuencia de bits representando un flotante simple en registro flotante a registro entero. Se copia tal cual, sin conversión. *rd* debe ser un registro entero y *rs1* un registro flotante.

fmv.w.x rd, rs1

Copiar secuencia de bits representando un flotante simple en registro entero a registro flotante. Se copia tal cual, sin conversión. *rd* debe ser un registro flotante y *rs1* un registro entero.

4.3.5. Control y estado

frcsr rd

Copiar **fcsr** en registro entero.

fscsr rs1

Copiar registro entero en **fcsr**.

frrm rd

Copiar campo RM (modo de redondeo) de **fcsr** en registro entero.

fsrm rs1

Copiar registro entero en campo RM (modo de redondeo) de **fcsr**.

frflags rd

Copiar campo de banderas acumuladas de **fcsr** en registro entero.

fsflags rs1

Copiar registro entero en campo de banderas acumuladas de **fcsr**.

4.4. Instrucciones de D

4.4.1. Operaciones

fadd.d rd, rs1, rs2

Sumar dos flotantes dobles.

fsub.d rd, rs1, rs2

Restar dos flotantes dobles.

fmul.d rd, rs1, rs2

Multiplicar dos flotantes dobles.

fdiv.d rd, rs1, rs2

Dividir dos flotantes dobles y obtener cociente.

fsqrt.d rd, rs1

Calcular la raíz cuadrada de un flotante doble.

fmin.d rd, rs1, rs2

Obtener el mínimo entre dos flotantes dobles.

fmax.d rd, rs1, rs2

Obtener el máximo entre dos flotantes dobles.

fmadd.d rd, rs1, rs2, rs3

Multiplicar dos flotantes dobles y sumarles otro.

$$rd := rs1 \times rs2 + rs3$$

fmsub.d rd, rs1, rs2, rs3

Multiplicar dos flotantes dobles y restarles otro.

$$rd := rs1 \times rs2 - rs3$$

fnmsub.d rd, rs1, rs2, rs3

Multiplicar dos flotantes dobles, negar producto y sumarles otro.

$$rd := -(rs1 \times rs2) + rs3$$

fnmadd.d rd, rs1, rs2, rs3

Multiplicar dos flotantes dobles, negar producto y restarles otro.

$$rd := -(rs1 \times rs2) - rs3$$

fsgnj.d rd, rs1, rs2

Inyectar signo en precisión doble. Tomar de *rs1* todos los bits salvo el de signo, y de *rs2* tomar el de signo.

fsgnjn.d rd, rs1, rs2

Inyectar signo negado en precisión doble. Tomar de *rs1* todos los bits salvo el de signo, y de *rs2* tomar el de signo negado.

fsgnjx.d rd, rs1, rs2

Inyectar signo con *xor* en precisión doble. Tomar de *rs1* todos los bits salvo el de signo, y para el signo calcular el *xor* de los signos de *rs1* y *rs2*.

4.4.2. Conversión

fcvt.w.d rd, rs1

Convertir flotante doble a palabra entera (4 bytes) con signo. *rd* debe ser un registro entero y *rs1* un registro flotante.

fcvt.wu.d rd, rs1

Convertir flotante doble a palabra entera (4 bytes) sin signo. *rd* debe ser un registro entero y *rs1* un registro flotante.

fcvt.d.w rd, rs1

Convertir palabra entera (4 bytes) con signo a flotante doble. *rd* debe ser un registro flotante y *rs1* un registro entero.

fcvt.d.wu rd, rs1

Convertir palabra entera (4 bytes) sin signo a flotante doble. *rd* debe ser un registro flotante y *rs1* un registro entero.

fcvt.s.d rd, rs1

Convertir flotante doble a flotante simple.

fcvt.d.s rd, rs1

Convertir flotante simple a flotante doble.

En RV64D:

fcvt.l.d rd, rs1

Convertir flotante doble a doble palabra entera (8 bytes) con signo. *rd* debe ser un registro entero y *rs1* un registro flotante.

fcvt.lu.d rd, rs1

Convertir flotante doble a doble palabra entera (8 bytes) sin signo. *rd* debe ser un registro entero y *rs1* un registro flotante.

fcvt.d.l rd, rs1

Convertir doble palabra entera (8 bytes) con signo a flotante doble. *rd* debe ser un registro flotante y *rs1* un registro entero.

fcvt.d.lu rd, rs1

Convertir doble palabra entera (8 bytes) sin signo a flotante doble. *rd* debe ser un registro flotante y *rs1* un registro entero.

4.4.3. Comparación y clasificación

fseq.d rd, rs1, rs2

Comparar flotantes dobles por igualdad y guardar el resultado en un registro. Si cualquier operando fuente es *Nan*, el resultado es 0.

fslt.d rd, rs1, rs2

Comparar flotantes dobles por menor y guardar el resultado en un registro. Si cualquier operando fuente es *Nan*, el resultado es 0.

fle.d rd, rs1, rs2

Comparar flotantes dobles por menor o igual y guardar el resultado en un registro. Si cualquier operando fuente es *Nan*, el resultado es 0.

fclass.d rd, rs1

Clasificar flotante doble según la clase de valor. Asigna un entero que indica si es normalizado, desnormalizado, cero, infinito o *Nan* (véase la sección *Clasificación de números* para más información). *rd* debe ser un registro entero y *rs1* un registro flotante.

4.4.4. Carga y descarga

fld rd, mem

Copiar flotante doble de memoria a registro. *rd* debe ser un registro flotante.

fsd rs1, mem

Copiar flotante doble de registro a memoria. *rs1* debe ser un registro flotante.

En RV64D:

fmv.x.d rd, rs1

Copiar secuencia de bits representando un flotante doble en registro flotante a registro entero. Se copia tal cual, sin conversión. *rd* debe ser un registro entero y *rs1* un registro flotante.

fmv.d.x rd, rs1

Copiar secuencia de bits representando un flotante doble en registro entero a registro flotante. Se copia tal cual, sin conversión. *rd* debe ser un registro flotante y *rs1* un registro entero.

4.5. Clasificación de números

Las instrucciones **fclass.s** y **fclass.d** clasifican valores flotantes de acuerdo a la clase de valor flotante que representan. Producen como resultado un valor entero del cual cada bit indica que se trata de una clase de valor flotante distinta. La siguiente tabla muestra las asociaciones:

Bit	Significado
0	$-\infty$
1	Número negativo normalizado
2	Número negativo desnormalizado
3	-0
4	$+0$
5	Número positivo desnormalizado
6	Número positivo normalizado
7	$+\infty$
8	<i>NaN</i> señalizador
9	<i>NaN</i> silencioso

Capítulo 5

Referencia de instrucciones

5.1. Instrucciones de I

La siguiente tabla lista todas las instrucciones del módulo I. Son 40 para RV32 y 12 más para RV64.

Instrucción	Descripción
add R, R, R	Sumar
addi R, R, I12	Sumar inmediato
sub R, R, R	Restar
lui R, I20	Cargar inmediato superior
auipc R, I20	Sumar inmediato a PC superior
xor R, R, R	Calcular disyunción exclusiva (<i>xor</i>) de bits
xori R, R, I12	Calcular disyunción exclusiva (<i>xor</i>) de bits con inmediato
or R, R, R	Calcular disyunción inclusiva (<i>or</i>) de bits
ori R, R, I12	Calcular disyunción inclusiva (<i>or</i>) de bits con inmediato
and R, R, R	Calcular conjunción (<i>and</i>) de bits
andi R, R, I12	Calcular conjunción (<i>and</i>) de bits con inmediato
sll R, R, R	Correr bits a izquierda
slli R, R, I12	Correr bits a izquierda con inmediato
srl R, R, R	Correr bits a derecha de forma lógica
srlti R, R, I12	Correr bits a derecha de forma lógica con inmediato
sra R, R, R	Correr bits a derecha de forma aritmética
srai R, R, I12	Correr bits a derecha de forma aritmética con inmediato
lb R, R, M	Cargar byte, con signo
lbu R, R, M	Cargar byte, sin signo
lh R, R, M	Cargar media palabra, con signo
lhu R, R, M	Cargar media palabra, sin signo
lw R, R, M	Cargar palabra

sb R, R, M	Almacenar byte
sh R, R, M	Almacenar media palabra
sw R, R, M	Almacenar palabra
jal R, M20	Saltar y enlazar
jalr R, R, M	Saltar a registro y enlazar
beq R, R, M	Saltar si es igual
bne R, R, M	Saltar si es distinto
blt R, R, M	Saltar si es menor, con signo
bltu R, R, M	Saltar si es menor, sin signo
bge R, R, M	Saltar si es mayor o igual, con signo
bgeu R, R, M	Saltar si es mayor o igual, sin signo
slt R, R, R	Comparar por menor, con signo
slti R, R, I12	Comparar por menor que inmediato, con signo
sltu R, R, R	Comparar por menor, sin signo
sltiu R, R, I12	Comparar por menor que inmediato, sin signo
fence ...	Ordenar accesos a memoria y dispositivos
ecall	Pedir servicio al entorno de ejecución (llamada a sistema)
ebreak	Retornar control a un entorno de depuración
addw R, R, R	Sumar
addiw R, R, I	Sumar inmediato
subw R, R, I	Restar
sllw R, R, R	Correr bits a izquierda
slliw R, R, I	Correr bits a izquierda con inmediato
srlw R, R, R	Correr bits a derecha de forma lógica
srliw R, R, I	Correr bits a derecha de forma lógica con inmediato
sraw R, R, R	Correr bits a derecha de forma aritmética
sraiw R, R, I	Correr bits a derecha de forma aritmética con inmediato
lwu R, M	Cargar palabra, sin signo
ld R, M	Cargar doble palabra
sd R, M	Almacenar doble palabra

5.2. Instrucciones de M

La extensión M incorpora 8 instrucciones en RV32 y 5 más en RV64.

Instrucción	Descripción
mul R, R, R	Multiplicar y obtener parte baja
mulh R, R, R	Multiplicar y obtener parte alta, con signo × con signo
mulhsu R, R, R	Multiplicar y obtener parte alta, sin signo × sin signo
mulhu R, R, R	Multiplicar y obtener parte alta, con signo × sin signo

<code>div R, R, R</code>	Dividir y obtener cociente, con signo
<code>divu R, R, R</code>	Dividir y obtener cociente, sin signo
<code>rem R, R, R</code>	Dividir y obtener resto, con signo
<code>remu R, R, R</code>	Dividir y obtener resto, sin signo
<code>mulw R, R, R</code>	Como <code>mul</code> pero sobre palabras
<code>divw R, R, R</code>	Como <code>div</code> pero sobre palabras
<code>divuw R, R, R</code>	Como <code>divu</code> pero sobre palabras
<code>remw R, R, R</code>	Como <code>rem</code> pero sobre palabras
<code>remuw R, R, R</code>	Como <code>remu</code> pero sobre palabras

5.3. Instrucciones de F

El módulo F incorpora 36 instrucciones.

Instrucción	Descripción
<code>fadd.s F, F, F</code>	Sumar
<code>fsub.s F, F, F</code>	Restar
<code>fmul.s F, F, F</code>	Multiplicar
<code>fdiv.s F, F, F</code>	Dividir
<code>fsqrt.s F, F</code>	Raíz cuadrada
<code>fmin.s F, F, F</code>	Mínimo
<code>fmax.s F, F, F</code>	Máximo
<code>fmadd.s F, F, F</code>	Multiplicar y sumar
<code>fmsub.s F, F, F</code>	Multiplicar y restar
<code>fnmsub.s F, F, F</code>	Multiplicar, negar y sumar
<code>fnmadd.s F, F, F</code>	Multiplicar, negar y restar
<code>fsgnj.s F, F, F</code>	Inyectar signo
<code>fsgnjn.s F, F, F</code>	Inyectar signo negado
<code>fsgnjx.s F, F, F</code>	Inyectar signo con <i>xor</i>
<code>fcvt.w.s R, F</code>	Convertir flotante a entero con signo
<code>fcvt.l.s R, F</code>	Convertir flotante a entero con signo
<code>fcvt.s.w F, R</code>	Convertir entero con signo a flotante
<code>fcvt.s.l F, R</code>	Convertir entero con signo a flotante
<code>fcvt.wu.s R, F</code>	Convertir flotante a entero sin signo
<code>fcvt.lu.s R, F</code>	Convertir flotante a entero sin signo
<code>fcvt.s.wu F, R</code>	Convertir entero sin signo a flotante
<code>fcvt.s.lu F, R</code>	Convertir entero sin signo a flotante
<code>feq.s R, F, F</code>	Comparar por igualdad
<code>flt.s R, F, F</code>	Comparar por menor
<code>fle.s R, F, F</code>	Comparar por menor o igual

fclass.s R, F	Clasificar valor flotante
flw F, M	Cargar flotante
fsw F, M	Almacenar flotante
fmv.x.w R, F	Copiar secuencia de bits
fmv.w.x F, R	Copiar secuencia de bits
frcsr R	Leer fcsr
fscsr R	Escribir fcsr
frrm R	Leer campo de modo de redondeo de fcsr
fsrm R	Escribir campo de modo de redondeo de fcsr
frflags R	Leer campo de banderas acumuladas de fcsr
fsflags R	Escribir campo de banderas acumuladas de fcsr

5.4. Instrucciones de D

El módulo D incorpora 32 instrucciones.

Instrucción	Descripción
fadd.d F, F, F	Sumar
fsub.d F, F, F	Restar
fmul.d F, F, F	Multiplicar
fdiv.d F, F, F	Dividir
fsqrt.d F, F	Raíz cuadrada
fmin.d F, F, F	Mínimo
fmax.d F, F, F	Máximo
fmadd.d F, F, F	Multiplicar y sumar
fmsub.d F, F, F	Multiplicar y restar
fnmsub.d F, F, F	Multiplicar, negar y sumar
fnmadd.d F, F, F	Multiplicar, negar y restar
fsgnj.d F, F, F	Inyectar signo
fsgnjn.d F, F, F	Inyectar signo negado
fsgnjx.d F, F, F	Inyectar signo con <i>xor</i>
fcvt.w.d R, F	Convertir flotante a entero con signo
fcvt.l.d R, F	Convertir flotante a entero con signo
fcvt.d.w F, R	Convertir entero con signo a flotante
fcvt.d.l F, R	Convertir entero con signo a flotante
fcvt.wu.d R, F	Convertir flotante a entero sin signo
fcvt.lu.d R, F	Convertir flotante a entero sin signo
fcvt.d.wu F, R	Convertir entero sin signo a flotante
fcvt.d.lu F, R	Convertir entero sin signo a flotante
fcvt.s.d F, F	Convertir flotante doble a simple

<code>fcvt.d.s F, F</code>	Convertir flotante simple a doble
<code>feq.d R, F, F</code>	Comparar por igualdad
<code>flt.d R, F, F</code>	Comparar por menor
<code>fle.d R, F, F</code>	Comparar por menor o igual
<code>fclass.d R, F</code>	Clasificar valor flotante
<code>fld F, M</code>	Cargar flotante
<code>fsd F, M</code>	Almacenar flotante
<code>fmv.x.d R, F</code>	Copiar secuencia de bits
<code>fmv.d.x F, R</code>	Copiar secuencia de bits

5.5. Pseudoinstrucciones

La siguiente tabla muestra algunas pseudoinstrucciones comunes.

Pseudoinstrucción	Descripción
<code>la R, M32</code>	Cargar dirección
<code>lla R, M32</code>	Cargar dirección local
<code>lb R, M32</code>	Cargar byte global
<code>lh R, M32</code>	Cargar media palabra global
<code>lw R, M32</code>	Cargar palabra global
<code>sb R, M32, R</code>	Guardar byte global
<code>sh R, M32, R</code>	Guardar media palabra global
<code>sw R, M32, R</code>	Guardar palabra global
<code>nop</code>	No hacer nada
<code>li R, I32</code>	Cargar inmediato
<code>mv R, R</code>	Copiar registro
<code>not R, R</code>	Complemento a uno
<code>neg R, R</code>	Complemento a dos
<code>seqz R, R</code>	Comparar por igualdad a cero
<code>snez R, R</code>	Comparar por distinto de cero
<code>sltz R, R</code>	Comparar por menor que cero
<code>sgtz R, R</code>	Comparar por mayor que cero
<code>beqz R, M</code>	Saltar si es igual a cero
<code>bnez R, M</code>	Saltar si es distinto de cero
<code>blez R, M</code>	Saltar si es menor o igual que cero
<code>bgez R, M</code>	Saltar si es mayor o igual que cero
<code>bltz R, M</code>	Saltar si es menor que cero
<code>bgtz R, M</code>	Saltar si es mayor que cero
<code>bgt R, R, M</code>	Saltar si es mayor, con signo
<code>bgtu R, R, M</code>	Saltar si es mayor, sin signo

ble R, R, M	Saltar si es menor o igual, con signo
bleu R, R, M	Saltar si es menor o igual, sin signo
j M	Saltar
jal M	Saltar y enlazar
jr R	Saltar a registro
jalr R	Saltar a registro y enlazar
ret	Retornar de subrutina
call M	Llamar a subrutina lejana
tail M	Llamar de cola a subrutina lejana

En RV64I:

ld R, M32	Cargar
sd R, M32, R	Almacenar
sext.w R, R	Extender signo en palabra

En RV32F:

fmv.s R, R	Copiar flotante a flotante
fabs.s R, R	Valor absoluto
fneg.s R, R	Negación

En RV32D:

fmv.d R, R	Copiar flotante a flotante
fabs.d R, R	Valor absoluto
fneg.d R, R	Negación

Capítulo 6

Enlaces y libros

6.1. Enlaces

- Sitio web oficial de RISC-V:
<https://riscv.org/>
- Manual de RISC-V, volumen 1 (conjunto de instrucciones no privilegiado):
<https://content.riscv.org/wp-content/uploads/2019/06/riscv-spec.pdf>
- Panorama del ecosistema de software de RISC-V:
<https://riscv.org/software-status/>
- Arquitectura RISC-V en Devopedia:
<https://devopedia.org/risc-v-architecture>
- Convención de llamada:
<https://riscv.org/wp-content/uploads/2015/01/riscv-calling.pdf>
- rv8 – RISC-V instruction set reference:
<https://rv8.io/isa>
- Libros de RISC-V:
<https://riscv.org/risc-v-books/>
- Materiales educativos de RISC-V:
<https://riscv.org/educational-materials/>

- *Instruction sets should be free: the case for RISC-V:*
<https://people.eecs.berkeley.edu/~krste/papers/EECS-2014-146.pdf>

6.2. Libros

- David Patterson, John Hennessy. *Computer organization and design: the hardware/software interface.* RISC-V 5th edition. ISBN: 978-0128122754.
- David Patterson, Andrew Waterman. *The RISC-V reader: an open architecture atlas.* ISBN: 978-0999249116.
<http://www.riscvbook.com/>
- John Hennessy, David Patterson. *Computer architecture: a quantitative approach.* 6th edition. ISBN: 978-0128119051.

Introducción a OpenCL

Leonardo Scandolo

leonardo@fceia.unr.edu.ar

Arquitectura del Computador 2014

Departamento de Ciencias de la Computación

FCEIA-UNR



1. Una historia de GPUs (modernos)

Un GPU (Graphics Processing Unit) moderno es un *coprocesador* que se encuentra en la mayoría de las computadoras de uso personal. Los primeros ejemplos de este tipo de coprocesadores se remontan a computadoras de los 70's y 80's, y podían tener diferentes usos (dibujar sprites, hostear la memoria de video, etc.) Sin embargo, a partir de los 90's, los GPU's empezaron a confluir en la misma función: acelerar el proceso de un tipo de algoritmos gráficos llamados algoritmos de *rasterización*. Los algoritmos de rasterización toman como entrada formas geométricas (triángulos por lo general), y los modifican a través de multiplicaciones con matrices que representan proyecciones y roto-traslaciones para finalmente mostrarlos en pantalla de manera que parezcan ser parte de una escena en 3 dimensiones.

El proceso de cada forma geométrica se daba por separado, siguiendo una arquitectura de *pipeline*. Las etapas de dicho pipeline eran originalmente fijas y algunas eran parametrizables, de manera de poder elegir por ejemplo colores, matrices de proyecciones, tamaños, etc. Dado que cada forma geométrica era procesada independientemente de las demás, se generó una arquitectura paralela con muchos procesadores dedicados a las mismas tareas, y que podían procesar muchas formas al mismo tiempo.

Con el tiempo, se le fueron agregando más parámetros a las etapas fijas, pero fue evidente que este tipo de arquitectura era muy limitada. Por lo tanto, algunas etapas (más notoriamente la etapa de *vertex shading* y *pixel o fragment shading*) se convirtieron en etapas programables. Esto quiere decir que se definió un lenguaje (lenguaje de shaders) con el cual se puede definir (dentro de ciertas restricciones) el comportamiento de las etapas programables. De esta manera, fue necesario que los procesadores de los GPUs fueran de propósito cada vez más general. A mediados de la década del 2000, los procesadores gráficos empezaron a ser construidos con cientos de procesadores de propósito general.

La figura ?? muestra un pipeline sencillo.

2. Introducción

Cuando los procesadores gráficos empezaron a poder ser programables, algunos investigadores notaron que algunos algoritmos de origen no gráfico podían implementarse en los shaders de un GPU, y la imagen final generada contendría la solución al problema que se intentaba resolver. Dado que los GPUs poseían cientos de procesadores (a una frecuencia mucho menor que una CPU), los algoritmos implementados de esta manera eran muchas veces más rápidos que la implementación en CPU. Las compañías productoras de GPU (notablemente NVIDIA y ATI) se dieron cuenta del mercado para usar GPUs como procesadores generales para resolver problemas no gráficos. A partir de esto,

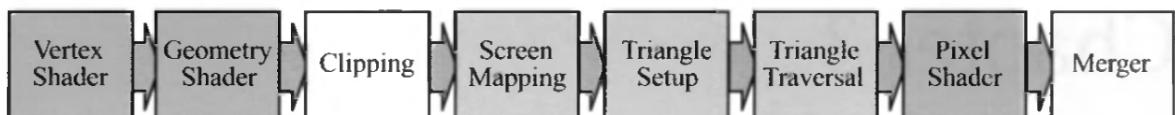


Figura 1: Pipeline simple para un GPU

NVIDIA crea CUDA, que es un lenguaje que permite interactuar con la memoria de sus GPUs, y escribir programas que serán ejecutados dentro del mismo. Para estandarizar este tipo de cálculos, el mismo ente que lleva adelante el estándar libre OpenGL, crea OpenCL (Open Computing Language), para proveer un framework estándar que puedan implementar todos los fabricantes de GPUs y CPUs, y que permite definir algoritmos paralelos para correr en los procesadores que provean soporte al estándar.

3. Arquitectura de un dispositivo OpenCL

3.1. Dispositivos OpenCL

Para estandarizar los diferentes tipos de procesadores que pueden implementar el estándar OpenCL, se definen *dispositivos OpenCL*, que es una máquina abstracta cuyas características son parametrizadas dependiendo del dispositivo real subyacente. Estos dispositivos son programables a través de una librería estándar definida por OpenCL. Esta librería está definida en forma de headers y funcionalidades, y cada fabricante provee su propia implementación. De esta manera es posible leer y escribir la memoria de un dispositivo, y enviar cálculos para que se ejecuten en el mismo. Dichos cálculos están definidos en un lenguaje especial llamado *OpenCL C*, que es muy parecido al lenguaje C.

Cada *dispositivo* expone una cantidad de *unidades de cómputo*, o *compute units*. Estas *unidades de cómputo* son grupos de uno o más procesadores. Los procesadores que forman parte de una *unidad de cómputo* son llamados *elementos de procesamiento* o *processing elements*. Todos los *elementos de procesamiento* dentro de una misma *unidad de cómputo* comparten recursos, como memoria y cache. Al momento de ejecutar instrucciones se comportan como una linea larga SIMD¹. Esto significa que todos los *elementos de procesamiento* de una misma *unidad de cómputo* ejecutan las mismas instrucciones, aunque posiblemente leyendo y escribiendo a lugares diferentes de la memoria. En el caso de expresiones condicionales (**IF-THEN-ELSE**) simplemente se desactivan los *elementos de procesamiento* correspondientes durante la ejecución de la rama condicional que no deben ejecutar.

La figura ?? muestra un esquema de la arquitectura descripta en el párrafo anterior.

Tanto los cómputos que se quieren realizar en un dispositivo OpenCL, como las lecturas o escrituras a memoria del dispositivo se deben encolar en una *cola de comandos* del dispositivo. Normalmente no hay un orden estricto en el orden de ejecución de los comandos de una cola de comandos. Sin embargo, hay varias formas de asegurar un orden. Una forma es llamando a una función llamada *clFinish* que espera hasta que todos los comandos de la cola hayan sido terminados. Otra forma es encolar una barrera en la cola, lo cual asegura que todos los comandos encolados desde ese punto serán ejecutados sólo luego que los comandos existentes en la cola sean ejecutados. OpenCL también define *eventos* que kernels pueden lanzar para que se ejecuten otros kernels que esperan esos eventos.

¹Single Instruction Multiple Data

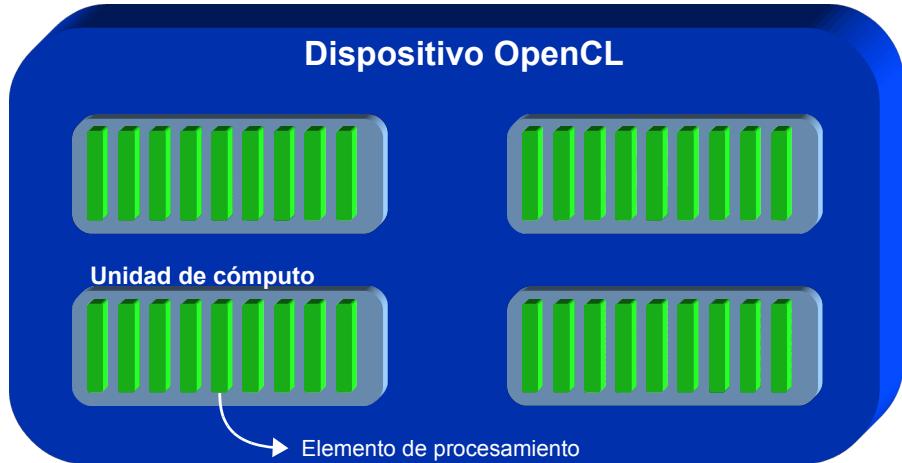


Figura 2: Diagrama de la arquitectura básica expuesta por OpenCL

3.2. Jerarquía de memoria

OpenCL también asume una jerarquía de memoria que está ligada a la forma en que se organizan los procesadores de un dispositivo. Cada *elemento de procesamiento* tiene acceso a su propia memoria privada, que consiste generalmente de registros. A su vez, los *elementos de procesamiento* de una misma *unidad de cómputo* tienen acceso a una memoria compartida que todos los elementos pueden leer y escribir, pero que no es accesible para los elementos de otras *unidades de cómputo*. Finalmente existe una memoria global del *dispositivo* que es accesible a todos los *elementos de procesamiento*.

Como es de esperar cada jerarquía de memoria tiene tamaños y tiempos de acceso diferentes, por lo cual, en la mayoría de los casos, el tiempo de acceso a registros de un *elemento de procesamiento* es menor al de la memoria compartida de una *unidad de cómputo*, que es a la vez menor al de la memoria global de un *dispositivo*. El caso contrario se da con el tamaño de cada jerarquía de memoria.

Cada *dispositivo* posee la capacidad de transferir datos desde y hacia la memoria principal de la computadora donde se encuentra. Los tiempos de transferencia en este caso son por lo general altos.

La figura ?? ejemplifica la jerarquía de memoria que expone OpenCL.

3.3. Kernels de OpenCL

Los algoritmos ejecutados en OpenCL son llamados *kernels*. Éstos son escritos en un lenguaje de programación propio de OpenCL, muy parecido al lenguaje C. Dichos algoritmos serán ejecutados por los *elementos de procesamiento* de un *dispositivo*. Al momento de ejecutar un *kernel* en un *dispositivo* es posible definir cuantas instancias de ese cómputo se ejecutarán. De esta manera un mismo cómputo puede ser ejecutado una gran cantidad de veces con una sola llamada a la biblioteca OpenCL mediante la capacidad de cómputos paralelos de los dispositivos que implementan el estándar OpenCL

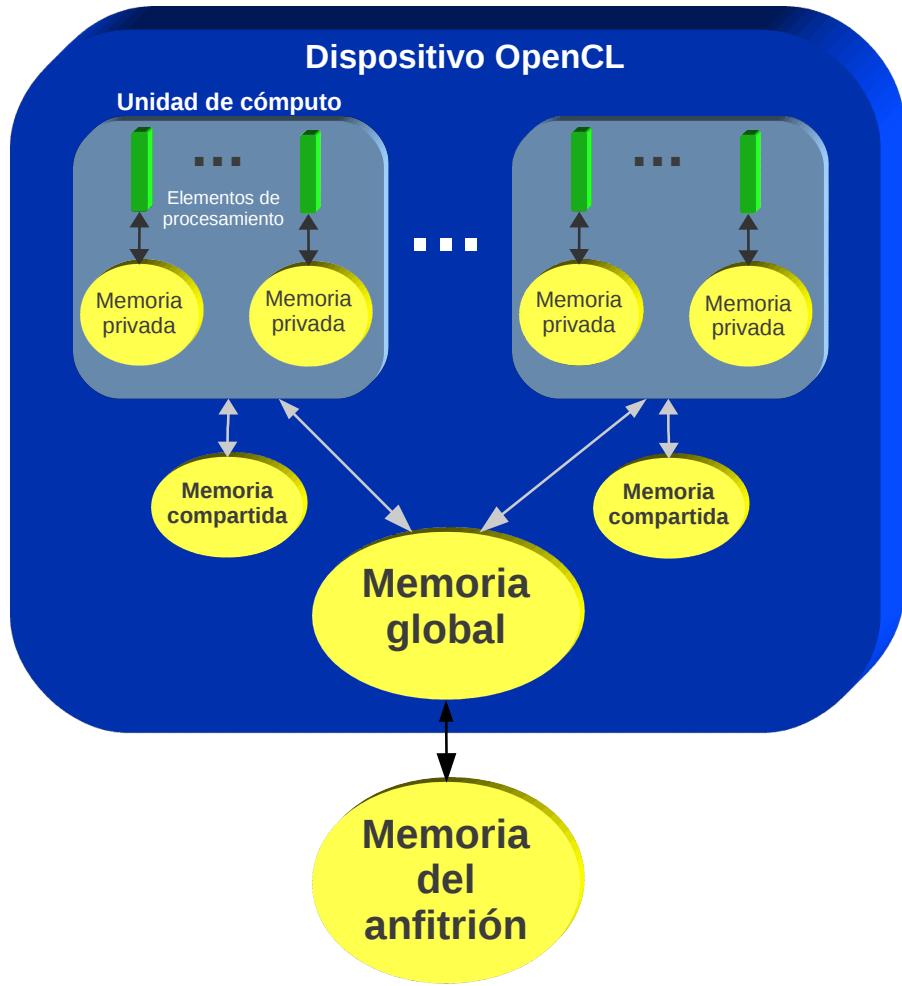


Figura 3: Diagrama de la jerarquía de memoria expuesta por OpenCL

Cada instancia a ejecutar de un *kernel* es llamada un *work item*. Cuando se define la cantidad de work items a ejecutar, también es necesario definir cómo serán agrupadas esas instancias. Los work item se agrupan porque cada instancia será ejecutada por un *elemento de procesamiento*, y cada grupo será ejecutado por una *unidad de cómputo*. Cada grupo de work items es llamado un *work group*. Los work items de un mismo work group tienen acceso a memoria compartida dado a que se ejecutan en la misma *unidad de cómputo*. Los work items de diferentes work groups sólo pueden comunicarse a través de la memoria global.

Para poder ejecutar código sobre diferentes datos, cada work item tiene acceso a un identificador global que lo diferencia de todos los otros work items y a un identificador local que lo identifica dentro de su work group.

La figura ?? ejemplifica la manera en que un *kernel* puede modificar datos distintos mientras ejecuta las mismas instrucciones. El *kernel* contiene sólo una instrucción que establece el valor de una posición de un arreglo. Tanto la posición como el valor son función del identificador de cada work item que se ejecute.

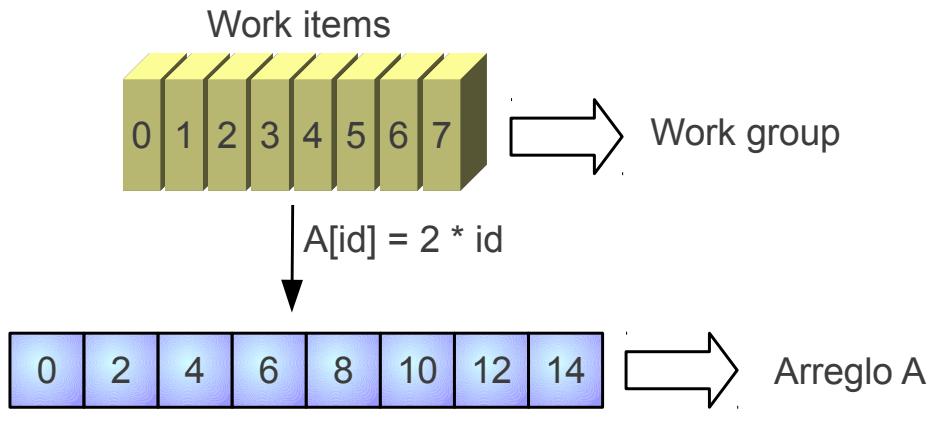
La cantidad de work items en un mismo work group está limitada por la cantidad de *elementos de procesamiento* por *unidad de cómputo* en el dispositivo OpenCL . Sin

```

__kernel void fill_array(global int* A) {
    size_t id = get_global_id(0);
    A[id] = 2 * id;
}

```

(a) Código de un *kernel* de OpenCL que modifica los valores de un arreglo global.



(b) Esquema del código ejecutado por el *kernel*.

Figura 4: Ejemplo de un cómputo para modificar un arreglo usando un *kernel* de OpenCL

embargo, la cantidad total de work items no está acotada, dado que en caso de haber más work groups que *unidades de cómputo*, simplemente se ejecutarán primero tantos como *unidades* haya disponible, y luego que se completen sus cálculos, se continuará con los siguientes, hasta culminar la ejecución de todos los work groups. Se desprende de esta lógica de ejecución que existe una expectativa de concurrencia para la ejecución de los work items de un mismo work group, pero no para work items de diferentes work groups.

Una característica importante de la definición de instancias de un *kernel* a ejecutar es que pueden definirse grupos multidimensionales, es decir que el identificador de cada work item y cada work group puede tener generalmente hasta 3 dimensiones. Esto facilita la división de trabajo en una gran cantidad de algoritmos paralelos.

3.4. Limitaciones de OpenCL

Las propiedades del modelo de cómputo de OpenCL afectan el diseño de la solución que se propone. Por lo tanto a continuación se discutirán algunas limitaciones y factores que afectan la eficiencia de los programas que utilizan OpenCL.

Un factor importante que afecta la eficiencia de OpenCL es que la ejecución de un *kernel* es ineficiente cuando diferentes work items de un mismo work group ejecuten ramas distintas de una expresión condicional. Esto es debido al modelo de cómputo expuesto basado en SIMD. Cuando un *kernel* no sufre este problema, se dice que los datos que está procesando son *coherentes*. Por lo tanto es siempre mejor agrupar los datos a procesar de manera que sean lo más coherentes posibles. A pesar de que no es una limitación que

no permita hacer un cálculo, el modelo de OpenCL fue creado con aplicaciones de alto desempeño como objetivo, por lo cual si no es posible obtener un buen rendimiento es preferible hacer cómputos en unidades de procesos tradicionales, como CPUs.

El lenguaje de definición de *kernels* en OpenCL es un lenguaje muy parecido al lenguaje C. Sin embargo presenta dos limitaciones muy importantes: la falta de recursión y la falta de memoria dinámica.

La falta de recursión implica que cualquier algoritmo usado en OpenCL debe ser completamente iterativo. Esta es sólo una limitación sobre la facilidad de escribir código, dado que cualquier cómputo recursivo (que termine) puede escribirse de forma iterativa.

La segunda limitación es más problemática, dado que implica que debemos establecer la cantidad de memoria que un *kernel* va a utilizar antes de comenzar su ejecución.

Otra limitación importante del modelo es la separación entre memoria del dispositivo que implementa OpenCL y la memoria convencional del sistema. La memoria del dispositivo puede ser accedida rápidamente durante la ejecución de un *kernel* en OpenCL. Sin embargo escribir o leer la memoria del dispositivo desde fuera del mismo puede ser lento, y por lo tanto se debe minimizar en lo posible estas operaciones.

Un último inconveniente resultante del uso de arquitecturas masivamente paralelas es que dado que están diseñadas para hacer cálculos utilizando muchos procesadores, la subutilización de los mismos decrementa la velocidad en la cual se ejecuta un algoritmo. En particular, dado que cada work group es asignado a una *unidad de cómputo* diferente, es siempre más eficiente crear work groups que utilicen todos los recursos de la *unidad de cómputo* donde se ejecutarán.

Resumiendo, las limitaciones más importantes del modelo OpenCL son:

- Necesidad de tener coherencia de datos.
- Falta de recursión.
- Falta de memoria dinámica.
- Lentitud de traspaso de datos entre sistema y dispositivo OpenCL.
- Necesidad de utilizar eficientemente los *elementos de procesamiento* disponibles.

4. Ejemplo de un programa en OpenCL

4.1. Inicialización de un dispositivo OpenCL

```
#include <stdlib.h>
#include <stdio.h>
#include <CL/opencl.h>

int main(int argc, char** argv)
{
    cl_int errNum;

///////////////////////////////
// Crear un contexto de OpenCL usando la primer plataforma disponible
/////////////////////////////
    cl_context context = 0;
    cl_platform_id firstPlatformId;
```

```

    cl_uint numPlatforms;
    errNum = clGetPlatformIDs(1, &firstPlatformId, &numPlatforms);
    if (errNum != CL_SUCCESS || numPlatforms <= 0)
    {
        puts("No se encontraron plataformas OpenCL.\n");
        return 1;
    }
    cl_context_properties contextProperties[] =
        {CL_CONTEXT_PLATFORM,
         (cl_context_properties)firstPlatformId,
         0
        };
    context = clCreateContextFromType(contextProperties,
                                     CL_DEVICE_TYPE_GPU,
                                     NULL, NULL, &errNum);
    if (errNum != CL_SUCCESS)
    {
        puts("No se pudo crear un contexto en GPU\n");
        return 1;
    }

///////////////////////////////
// Crear una cola de comandos en el primer dispositivo disponible,
// en el contexto creado
/////////////////////////////
    cl_command_queue commandQueue = 0;
    cl_device_id *devices;
    cl_device_id device = 0;
    size_t deviceBufferSize;
    errNum = clGetContextInfo(context, CL_CONTEXT_DEVICES, 0, NULL,
                             &deviceBufferSize);
    if (errNum != CL_SUCCESS)
    {
        puts("No se pudo obtener deviceBufferSize\n");
        return 1;
    }
    devices = (cl_device_id*)malloc(deviceBufferSize);
    errNum = clGetContextInfo(context, CL_CONTEXT_DEVICES,
                             deviceBufferSize, devices, NULL);
    if (errNum != CL_SUCCESS)
    {
        puts("No se pudieron obtener los identificadores de los dispositivos\n");
        return 1;
    }
    device = devices[0];
    commandQueue = clCreateCommandQueue(context,
                                       device, 0, NULL);
    if (commandQueue == NULL)
    {
        puts("No se pudo crear la cola de comandos\n");
        return 1;
    }
    free(devices);

...

```

4.2. Un kernel de OpenCL sencillo

```

kernel matrix_multiplication(const int Mdim,
                            const int Ndim,
                            const int Pdim,
                            global float* A,
                            global float* B,
                            global float* C)
{
    int k;

```

```

int i = get_global_id(0);
int j = get_global_id(1);
float tmp;

if( (i < Ndim) && (j < Mdim)) {
    tmp = 0.0;

    for(k=0;k<Pdim;k++)
        tmp += A[i*Ndim+k] * B[k*Pdim+j];
    C[i*Ndim+j] = tmp;
}
}

};


```

4.3. Crear un kernel OpenCL y encollarlo

```

// Asumiendo que tenemos
// #define ARRAY_SIZE 32
// y que "extern char* program_source" es la declaración del string
// que contiene el kernel

...
///////////
// Crear un programa OpenCL desde el código fuente en program_source
///////////
    cl_program program = 0;
    program = clCreateProgramWithSource(context, 1,
                                         (const char**)&program_source,
                                         NULL,&errNum);

    if (errNum != CL_SUCCESS)
    {
        puts("No se pudo crear el programa.\n");
        return 1;
    }
    errNum = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
    if (errNum != CL_SUCCESS)
    {
        puts("No se pudo construir el programa.\n");
        return 1;
    }

///////////
// Crear el kernel de OpenCL
/////////
    cl_kernel kernel = 0;
    kernel = clCreateKernel(program, "hello_kernel", NULL);
    if (kernel == NULL)
    {
        puts("No se pudo crear el kernel\n");
        return 1;
    }
///////////
// Crear los objetos de memoria que van a ser usados como
// argumentos del kernel. Primero se crean los arreglos de memoria que
// van a ser usados para guardar los argumentos del kernel.
/////////
    cl_mem memObjects[3] = { 0, 0, 0 };
    float result[ARRAY_SIZE];
    float a[ARRAY_SIZE];
    float b[ARRAY_SIZE];
    for (int i = 0; i < ARRAY_SIZE; i++)
    {
        a[i] = i;

```

```

        b[i] = i * 2;
    }
    memObjects[0] = clCreateBuffer(context, CL_MEM_READ_ONLY |
                                    CL_MEM_COPY_HOST_PTR,
                                    sizeof(float) * ARRAY_SIZE, a,
                                    NULL);
    memObjects[1] = clCreateBuffer(context, CL_MEM_READ_ONLY |
                                    CL_MEM_COPY_HOST_PTR,
                                    sizeof(float) * ARRAY_SIZE, b,
                                    NULL);
    memObjects[2] = clCreateBuffer(context, CL_MEM_READ_WRITE,
                                    sizeof(float) * ARRAY_SIZE,
                                    NULL, NULL);
///////////////////////
// Establecer los argumentos del kernel (result, a y b)
///////////////////////
    errNum = clSetKernelArg(kernel, 0,
                           sizeof(cl_mem), &memObjects[0]);
    errNum |= clSetKernelArg(kernel, 1, sizeof(cl_mem),
                           &memObjects[1]);
    errNum |= clSetKernelArg(kernel, 2, sizeof(cl_mem),
                           &memObjects[2]);
    if (errNum != CL_SUCCESS)
    {
        puts("No se pudieron establecer los argumentos del kernel.\n");
        return 1;
    }
///////////////////////
// Encolar el kernel para su ejecución
///////////////////
    size_t globalWorkSize[1] = { ARRAY_SIZE };
    size_t localWorkSize[1] = { 1 };
    errNum = clEnqueueNDRangeKernel(commandQueue, kernel, 1, NULL,
                                    globalWorkSize, localWorkSize,
                                    0, NULL, NULL);
    if (errNum != CL_SUCCESS)
    {
        puts("No se pudo encolar el kernel para ejecutar.\n");
        return 1;
    }
///////////////////
// Copiar el resultado a un buffer en el anfitrión
/////////////////
    errNum = clEnqueueReadBuffer(commandQueue, memObjects[2],
                                CL_TRUE, 0, ARRAY_SIZE * sizeof(float),
                                result, 0, NULL, NULL);

    if (errNum != CL_SUCCESS)
    {
        puts("Error al leer el buffer de resultado.\n");
        return 1;
    }
///////////////////
// Imprimir el resultado
/////////////////
    for (int i = 0; i < ARRAY_SIZE; i++)
    {
        printf("%f ", result[i]);
        /* cout << result[i] << " "; */
    }
    puts("\nPrograma ejecutado exitosamente.\n");

...

```

5. Bibliografía

- A. Munshi et al, *OpenCL Programming Guide*, 2012.
- *OpenCL 1.1 specification*, Khronos Group, disponible en www.khronos.org/registry/cl/specs/opencl-1.1.pdf.
- *OpenCL C specification*, Khronos Group, disponible en www.khronos.org/registry/cl/specs/opencl-2.0-openclc.pdf.