Resumen Final de Orga 1

The FurfiOS Corporation

Diciembre 2020

Índice general

1. Introduccion 7
1.1. ¿Qué es una computadora? 7
1.2. Arquitectura vs Organización 7
1.3. Retrocompatibilidad 7
1.4. Principio de Equivalencia HW - SW 7
1.5. Estructura vs Función 8
1.5.1. Visión funcional 8
1.5.2. Visión estructural 9

2. Sistemas de Representación 11
2.1. Representación de la información 11
2.2. Sistemas de numeración 11
2.3. Cambio de base 11
2.4. Representación de números enteros 12
2.4.1. Magnitud signada 13
2.4.2. Sistema de Complemento 13
2.4.3. Complemento a 1 13
2.4.4. Complemento a 2 14
2.4.5. Multiplicación y División 14
2.4.6. Notación exceso-N 15
2.5. Representación de números reales 15
2.5.1. Punto fijo 15
2.5.2. Aproximación de reales 16
2.5.3. Punto flotante 16
2.6. IEEE - 754 18
2.6.1. Aritmética de Punto Flotante 20
2.7. Representación de Caracteres 21
2.7.1. ASCII 21
2.7.2. Unicode 22
2.7.3. UTF-8 22

3. Lógica Digital: Parte 1 25
3.1. Introducción 25
3.2. Algebra Booleana 26
3.2.1. Introducción 26
3.2.2. Identidades Lógicas 27
3.2.3. Fórmulas Equivalentes 27
3.3. Circuitos booleanos 28
3.4. Compuertas lógicas 29
3.4.1. Compuertas NAND y NOR 30
3.5. Buffer de tres estados 33
3.6. Circuitos Combinatorios 34

3.6.1. Sumador . . . 34
3.6.2. Decodificador . . . 38
3.6.3. Multiplexor . . . 38
3.6.4. Demultiplexor . . . 40
3.6.5. Memoria ROM . . . 40
3.6.6. Comparador . . . 42
3.7. ALU . . . 42
3.7.1. ALU de 1-bit . . . 43
3.7.2. ALU de 8-bits . . . 44

4. Lógica Digital: Parte 2 **45**
4.1. Introducción: Circuitos Secuenciales . . . 45
4.2. Clock . . . 45
4.3. Flip-Flop . . . 46
4.3.1. Flip-Flop: S - R . . . 47
4.3.2. S-R Flip-Flop Sincronico . . . 49
4.3.3. Flip-Flop D . . . 49
4.3.4. Flip-Flop: J-K . . . 50
4.4. Registros . . . 51
4.4.1. Parallel Registers . . . 51
4.4.2. Shift Register . . . 52
4.5. Organización de Memorias . . . 53
4.6. Contadores . . . 57
4.6.1. Contadores asincrónicos . . . 57
4.6.2. Contadores sincrónicos . . . 58

5. Instruction Set Architecture (ISA) - Parte 1 **61**
5.1. Introducción . . . 61
5.2. Ciclo de instrucción . . . 63
5.3. Retrocompatibilidad . . . 63
5.4. Propiedades de una ISA . . . 64
5.4.1. RISC vs CISC . . . 64
5.4.2. Longitud de las instrucciones . . . 64
5.4.3. Forma de representar la información . . . 64
5.4.4. Big endian vs Little endian . . . 64
5.5. Tipos de Arquitecturas . . . 65
5.5.1. Ejemplo de Arquitectura de Acumulador: MARIE . . . 65
5.5.2. Arquitectura de Registros de Propósito General: Máquina ORGA1 . . . 67
5.5.3. Arquitectura de Stack: . . . 69
5.6. Subrutinas . . . 70

6. Instruction Set Architecture (ISA) - Parte 2 **73**
6.1. Introducción . . . 73
6.2. Ortogonalidad . . . 75
6.3. Código de operación variable . . . 75

7. Unidad de control y Microprogramacíon **77**
7.1. Introducción . . . 77
7.2. Problemas a resolver . . . 79
7.3. Diseño de una computadora: pasos necesarios . . . 79
7.4. Diseñando una Unidad de Control para la arquitectura MIPS . . . 80
7.4.1. Paso 1: Analizar el conjunto de instrucciones para determinar los requerimientos del camino de datos. . . . 82
7.4.2. Paso 2: Selección de componentes electrónicos . . . 84

7.4.3. Paso 3: Construcción del camino de datos según los requerimientos con las componentes seleccionadas . . . 86
7.4.4. Paso 4: Analizar la implementación de cada instrucción para determinar las señales de control necesarias . . . 90
7.4.5. Paso 5: Construir la unidad de control que implemente el comportamiento necesario 91
7.5. Implementaciones de la Unidad de Control: . . . 94
7.5.1. ROM . . . 94
7.5.2. PLA . . . 95
7.6. Microprogramación . . . 96
7.7. Ventajas y Desventajas . . . 98

8. Entrada / Salida 99
8.1. Introducción . . . 99
8.2. Métodos de acceso a E/S . . . 101
8.2.1. Puertos dedicados . . . 101
8.2.2. Mapear a Memoria . . . 101
8.2.3. Puertos vs Mapeo a Memoria . . . 102
8.3. Métodos de control de E/S . . . 103
8.3.1. Programmed Input / Output: Polling . . . 103
8.3.2. Interrupciones . . . 104
8.3.3. DMA . . . 107

9. Entrada / Salida: Conversión de señales 111
9.1. Introducción . . . 111
9.2. Información analógica . . . 111
9.3. Amplificador operacional . . . 112
9.4. Conversiones Analógico-Digital y Digital-Analógico . . . 113
9.4.1. Conversión Digital a Analógica . . . 113
9.4.2. Conversión de Analógica a Digital . . . 114

10. Memoria y Cache 117
10.1. Introducción . . . 117
10.2. Tipos de Memorias . . . 118
10.2.1. ROM . . . 118
10.2.2. RAM . . . 118
10.3. Estructura de bus clásica . . . 121
10.3.1. Estructura de bus con cache . . . 123
10.4. Organización de la Cache . . . 124
10.5. Esquemas de Mapeo . . . 126
10.5.1. Mapeo Directo o de Correspondencia Directa . . . 126
10.5.2. Mapeo Completamente Asociativo . . . 126
10.5.3. Mapeo Asociativo por Conjuntos de N vías . . . 127
10.6. Políticas de reemplazo de contenido . . . 127
10.7. Políticas de Escritura . . . 128
10.8. Cache Multinivel . . . 129

11. Buses 131
11.1. Introducción . . . 131
11.2. Buses . . . 132
11.3. Diseño de un bus . . . 134
11.3.1. Ancho del bus . . . 134
11.3.2. Tipo de línea . . . 134
11.3.3. Temporización . . . 135
11.3.4. Arbitraje . . . 137

Este resumen fue hecho en base a las clases teóricas de Charly del Segundo Cuatrimestre 2020, complementado con la bibliografía de la materia: Null [4], Tanenbaum (Structured Computer Organization) [2] (Modern Operating Systems) [1], Stallings [5], Kaufmann [3]. Si quieren modificar el documento, tienen el overleaf acá `https://www.overleaf.com/7718548219vxdwcqcgtpvb`.

Capítulo 1

Introduccion

1.1. ¿Qué es una computadora?

Una computadora es una máquina digital (opera en base a valores discretos) electrónica programable para el tratamiento de la información, capaz de recibirla, operar sobre ella, y suministrar los resultados de tales operaciones.

1.2. Arquitectura vs Organización

La arquitectura de computadora se refiere a aquellos atributos de un sistema visibles para un programador o, dicho de otra manera, aquellos atributos que tienen un impacto directo en la ejecución lógica de un programa. Los atributos arquitectónicos incluyen el conjunto de instrucciones, el número de bits que se usan para representar varios tipos datos (por ejemplo int, char, etc), mecanismos de E / S, y modos de direccionamiento de la memoria.

La organización se refiere a cómo se implementan esos atributos, es decir cuales son las unidades operacionales y sus interconexiones que realizan las especificaciones arquitectónicas. Los atributos organizacionales incluyen aquellos detalles de hardware transparentes para el programador, como señales de control, interfaces entre la computadora y los periféricos, y la tecnología de memoria utilizada (SRAM, DRAM, etc).

1.3. Retrocompatibilidad

Muchos fabricantes de computadoras ofrecen una familia de modelos de computadora, todos con la misma arquitectura pero con diferencias en la organización. Este es el caso de toda la familia x86 de Intel que comparte la misma arquitectura básica, permitiendo la retrocompatibilidad del código de los programas antiguos.

1.4. Principio de Equivalencia HW - SW

Cualquier cómputo que pueda ser realizado por una intervención de una pieza de software, puede ser realizado por un artefacto de hardware, y viceversa.

1.5. Estructura vs Función

La computadora es un sistema complejo. Las computadoras contemporáneas contienen millones de componentes electrónicos, entonces ¿Cómo se pueden describir claramente? La clave es reconocer la naturaleza jerárquica de los sistemas más complejos.

Un sistema jerárquico es un conjunto de subsistemas interrelacionados, cada uno de estos de estructura jerárquica hasta llegar al nivel elemental más bajo del subsistema. Solo se necesita tratar con un nivel particular del sistema a la vez, trabajando sobre la caracterización abstracta de las capas inmediatamente anterior y posterior.

En cada nivel, nos preocupamos por la estructura y la función:

- **Estructura**: la forma en que las componentes están interrelacionadas para llevar a cabo una tarea.
- **Función**: la operación de cada componente individual como parte de la estructura (procesamiento, almacenamiento, transferencia de datos, control).

1.5.1. Visión funcional

Cuando se reciben o entregan datos de un dispositivo que está directamente conectado a la computadora, el proceso se conoce como entrada-salida (E/S), y el dispositivo se conoce como periférico. Cuando se mueven los datos en distancias más largas, hacia o desde un dispositivo remoto, el proceso se conoce como data-communications.

Las funciones básicas de una computadora son:

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

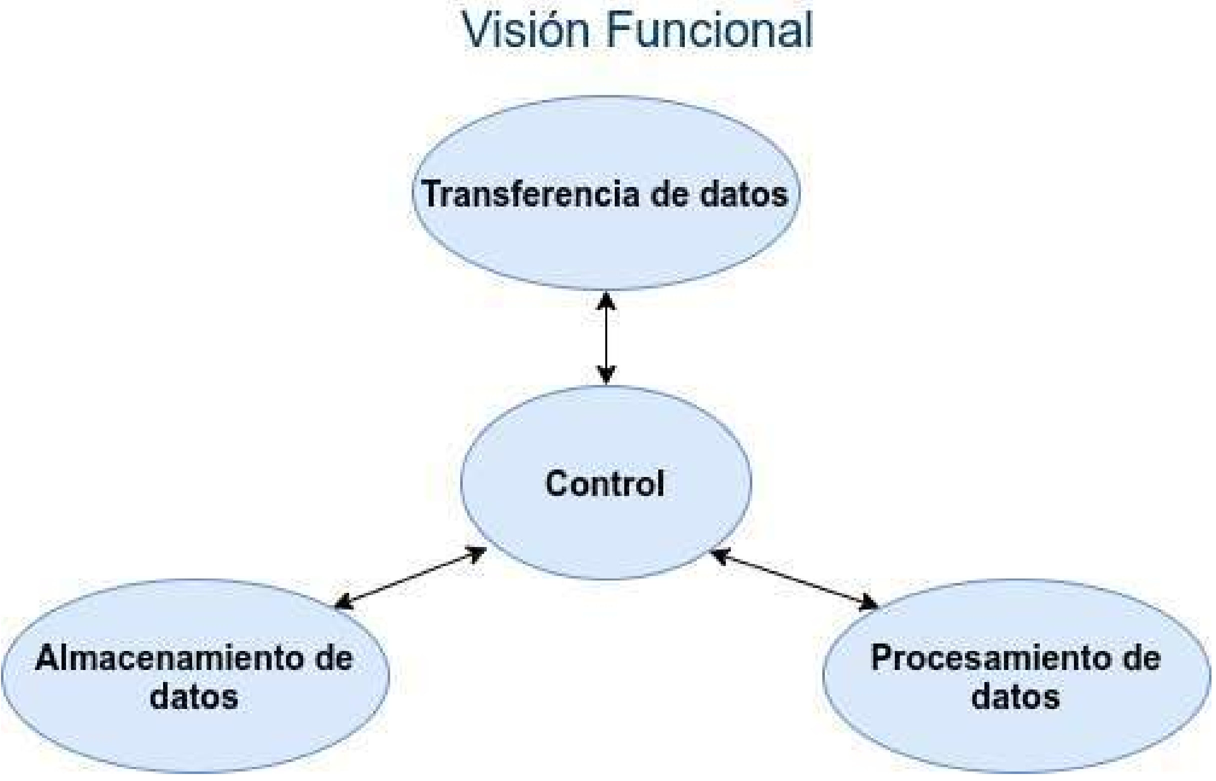


Figura 1.1

1.5.2. Visión estructural

Hay cuatro componentes estructurales principales:

- **Unidad central de procesamiento** (CPU): controla el funcionamiento de la computadora y realiza sus funciones de procesamiento de datos; a menudo simplemente se conoce como procesador. Sus principales componentes estructurales son los siguientes:
  - **Unidad de control**: controla el funcionamiento de la CPU.
  - **Unidad aritmética y lógica (ALU)**: realiza las funciones de procesamiento de datos de la computadora.
  - **Registros**: proporciona almacenamiento interno a la CPU.
  - **Interconexión de la CPU**: algún mecanismo que permite la comunicación. entre la unidad de control, ALU y registros.
- **Memoria principal**: almacena datos.
- **E/S**: mueve los datos entre la computadora y su entorno externo.
- **Interconexión del sistema**: algún mecanismo que permite la comunicación entre CPU, memoria principal y E / S. La conexión suele ser por medio de un bus del sistema, que consiste en una serie de cables a los que se unen todos los demás componentes.

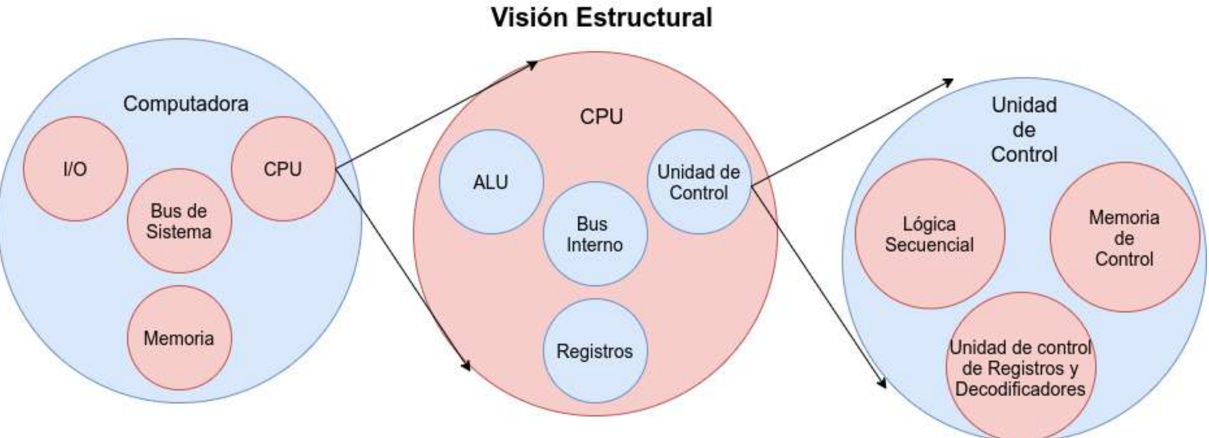


Figura 1.2

Capítulo 2

Sistemas de Representación

2.1. Representación de la información

Lo primero que tenemos que comprender es que, si vamos a apelar al modelo de cómputo Von Neumann - Turing, para poder construir una máquina que toma algún concepto de programa y de entrada, para luego arrojar un resultado, necesitamos poder ofrecer una manera de representar esta información.

2.2. Sistemas de numeración

Un factor decisivo en la organización de una computadora es la manera en la que está representada la información (números, caracteres, etc). Los números son formas de expresar magnitudes preexistentes, y las mismas se pueden representar usando un sistema de numeración.

Un sistema de numeración es un conjunto de símbolos y reglas combinables que permiten representar información. En particular, nos vamos a centrar en la representación de los números enteros y fraccionarios. Dentro de los sistemas de numeración posibles un conjunto importante, destacado, es el constituido por los sistemas posicionales.

En los sistemas posicionales, tenemos un conjunto de símbolos que podemos utilizar $(0, \ldots, base-1)$, donde la base es el cardinal del conjunto de símbolos. Luego, la representación de la información va a estar dada por una tira de símbolos, de manera que cada posición en la tira está ligada con una potencia de la base.

Ejemplos de sistemas posicionales son el sistema decimal (base 10), el sistema hexadecimal(base 16), y el sistema binario(base 2). Los dígitos utilizados para representar a las magnitudes en estos sistemas están comprendidos entre el 0 y $b-1$. Por ejemplo, en el sistema decimal, los dígitos van del 0 al $(10-1 = 9)$. En la modernidad, el sistema usado para representar la información es el binario.

2.3. Cambio de base

Como racionalizamos la magnitudes en base 10, nos interesa poder pasar de cualquier base a base 10, y viceversa. Esto nos va a permitir vincular resultados obtenidos bajo un sistema de representación, con cualquier otro sistema. Existen maneras de cambiar la representación de una misma magnitud cambiando la base, como por ejemplo el método de restas sucesivas o el método de restos de cocientes:

- **Restos de cocientes**: Tenemos que ir dividiendo/multiplicando por la base a la que queremos cambiar. El procedimiento termina cuando el resultado de la división es menor (estricto) que la base. Los dígitos que forman el número en la nueva base se forma empezando por el último resultado

seguido de todos los restos.

- **Restas sucesivas**: Restamos por un múltiplo (menor estricto que la base) de la mayor potencia de la base (a la que queremos cambiar) que sea menor (o igual) a n, de manera iterativa.

Restas sucesivas:

104
-81 = 3^4*1
23
-0 = 3^3*0
23
-18 = 3^2*2
5
-3 = 3^1*1
2
-2 = 3^0*2
0

$(104)_{10} = (10212)_3$

Restos de cocientes:

104 ⌊ 3
2 34 ⌊ 3
1 11 ⌊ 3
2 3 ⌊ 3
0 1

$(104)_{10} = (10212)_3$

Figura 2.1

El método de restos de cocientes es más eficiente ya que no tiene que calcular potencias, siendo estas operaciones costosas.

Para realizar la operación inversa, es decir pasar de cualquier base c a decimal, se evalúa el siguiente polinomio:

De representación a magnitud

$$(a_n \ldots a_2 a_1 a_0 . a_{-1} a_{-2} \ldots a_{-m})_{10} = a_n * 10^n + \ldots + a_2 * 10^2 + a_1 * 10 + a_0 + a_{-1} * 10^{-1} + a_{-2} * 10^{-2} + \ldots + a_{-m} * 10^{-m}$$

$$\parallel$$

$$(b_p \ldots b_2 b_1 b_0 . b_{-1} b_{-2} \ldots b_{-q})_c = b_p * c^p + \ldots + b_2 * c^2 + b_1 * c + b_0 + b_{-1} * c^{-1} + b_{-2} * c^{-2} + \ldots + b_{-q} * c^{-q}$$

Figura 2.2

donde c es la base, y b_i es el coeficiente en la i-esima posición de la tira. El separador que aparece entre los dígitos b_0 y b_{-1} se denomina separador fraccionario

2.4. Representación de números enteros

La forma más sencilla de representar una magnitud con signo es agregar un símbolo distinguido (que suele ser el símbolo "-"). Pero este "-" no forma parte de nuestro alfabeto, y por lo tanto no podemos representarlo en un sistema de base 10 (necesitaríamos un símbolo extra).

Luego, para representar magnitudes con signo, por convención, se define que un 1 en el bit más significativo representa un número negativo. El resto de los bits se usan para representar el número en sí. Cómo se representa dicho número depende del método utilizado.

Además, nos interesa tener una representación de precisión fija para poder implementar una ALU que nos permita operar sobre tiras de bits de tamaño fijo.

2.4.1. Magnitud signada

Se utiliza el dígito más significativo para representar el signo (0 si es positivo, cualquier otro si es negativo). Incluye el concepto de acarreo y, como se cuenta con una cantidad fija de dígitos, puede provocar *overflow*.

El rango de representación es: $[-b^{n-1}+1, b^{n-1}-1]$, donde n es la cantidad de dígitos disponibles. Veamos un ejemplo de uso de la Magnitud signada con 8 dígitos:

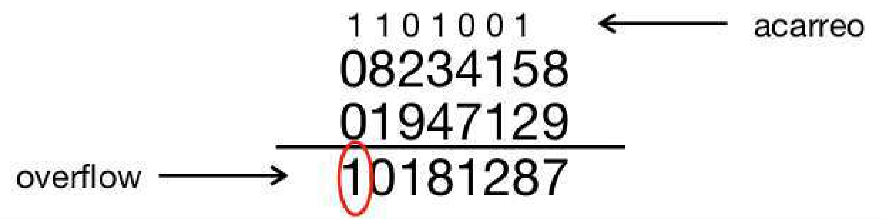


Figura 2.3

Como podemos ver, en este caso tuvimos un overflow. Un error de overflow ocurre cuando se intenta almacenar un número que supera el rango de valores representables. En este caso estamos sumando dos números positivos, obtuvimos como resultado un número negativo. Esto se puede detectar mirando un registros especial (EFLAGS en Intel) que nos indica si hubo overflow en la última operación.

Para extender la representación de un número de n bits a más bits se agregan ceros detrás del bit de signo, manteniendo el signo. Ejemplo:

$\bullet 101 = -1$ $\quad \bullet\ 001 = 1$ $\quad$ (3 digitos)

$\bullet 1001 = -1$ $\quad \bullet\ 0001 = 1$ $\quad$ (4 digitos)

Notemos que 0 = 100 = 000, por lo que tenemos dos representaciones del 0, lo cual nos puede traer algunos inconvenientes.

2.4.2. Sistema de Complemento

Se utilizan los números más grandes para representar a los negativos. Esto es ventajoso respecto al método de magnitud signada ya que no es necesario procesar los bits de signo por separado, y además conservamos la capacidad de verificar el signo de un número mirando solo su bit más significativo.

Además, podemos representar $b-1$ veces más información (donde b es la base), y son muy simples de implementar en hardware, ya que solo se necesita de dos circuitos: uno que sabe sumar, y otro que sabe complementar.

2.4.3. Complemento a 1

Podemos calcular el complemento a 1 de un número al invertir los bits que lo componen. El complemento a 1 de n se obtiene restándole al máximo representable n. La principal desventaja del Complemento a 1 es que hay dos representaciones del 0: el 000...000 y el 11...111, lo cual trae ciertos inconvenientes. Por este y otros motivos, hace mucho que se dejó usar, en favor de la representación Complemento a 2 de números binarios.

Para extender la representación a más bits, se extiende con el bit más significativo:

$$\bullet 1 = 001 \qquad \bullet -1 = 7 - 1 = 110 \qquad (\text{ 3 digitos})$$
$$\bullet 1 = 0001 \qquad \bullet -1 = 15 - 1 = 1110 \qquad (\text{ 4 digitos})$$

Por otro lado, notemos que para obtener el complemento a 1 consiste simplemente en invertir (swapear) los bits, lo cual es muy fácil de implementar en HW, por lo que obtenemos una resta muy eficiente.

Notemos que $0 = 000 = 111$, por lo que tenemos, nuevamente, dos representaciones para el cero.

2.4.4. Complemento a 2

El complemento a 2 se computa como el Ca1, pero luego se le suma 1 al resultado. Con esto evitamos tener dos representaciones del cero, siendo la única el 0...0. Podemos calcular el complemento a 2 de n restándole n al menor número mayor al máximo representable. Además, para extender la representación a más bits, se extiende con el bit más significativo:

$$\bullet - 1 = 8 - 1 = 111 \qquad \bullet 1 = 001 \qquad (\text{ 3 dígitos})$$
$$\bullet - 1 = 16 - 1 = 1111 \qquad \bullet 1 = 0001 \qquad (\text{ 4 dígitos})$$

En este caso, obtener el complemento a 2 consiste en invertir (swapear) los bits y luego sumarle 1 a ese resultado, lo que sigue siendo sencillo de implementar a nivel de HW.

Como estamos trabajando con una representación finita, vamos a tener *overflow*, y necesitamos una regla simple para detectarlo:

- Si el `carry in` al bit de signo es igual al `carry out` del bit de signo $\implies$ no hubo overflow.
- Si el `carry in` al bit de signo es diferente del `carry out` del bit de signo $\implies$ hubo overflow.

2.4.5. Multiplicación y División

La multiplicación binaria se realiza como sucesión de sumas desplazadas tal como en base 10, pero resulta más fácil de implementar:

- Si el bit es 1 se suma el multiplicando, y se desplaza
- Si es 0 solo se desplaza.

Ejemplo: $9 * 13 = 117 = 01110101$

Multiplicando	Multiplicando	Producto parcial
1001	1101	
10010	1101	00001001
100100	1101	00001001
1001000	1101	00101101
		01110101

Figura 2.4

La división se realiza a partir de realizar sustracciones sucesivas del divisor al dividendo. En el caso de la división entera el resultado se expresa como el par (cociente, resto). En el caso de la división fraccionaria depende de la representación, y lo veremos más adelante cuando veamos la representación de los números reales.

2.4.6. Notación exceso-N

La notación exceso-N es similar al complemento a 2, salvo que se asume que el número 000...00 representa la magnitud menos significativa, mientras que el 11...111 la más significativa. En el caso de que N sea $2^{(k-1)}$ donde k es la cantidad de bits, se obtiene una representación análoga a la de complemento a 2, en términos de rango representable.

Para encontrar la representación de un número m en notación exceso-N simplemente se hace $N+m$. Esto permite representar rangos asimétricos pues N determina la representación del 0, además de que la representación del 0 es única. Veamos unos ejemplos en notación exceso-7:

$$\bullet(-101)_2 = (00111)_2 - (101)_2 = (00010)_2$$

$$\bullet(-111)_2 = (00111)_2 - (111)_2 = (00000)_2$$

2.5. Representación de números reales

Los números reales no pueden ser representados totalmente, ni siquiera en un intervalo acotado, ya que existen reales con infinitos decimales y los registros de un procesador tienen un tamaño finito de bits, por lo que no pueden representar la infinidad de números reales. Es decir, no sólo tenemos un problema con la magnitud, sino que también tenemos un problema de precisión.

Aún así, existen formas de representar los números reales: la representación de punto fijo (pudiendo ser magnitud signada o complemento a 2) y la representación de punto flotante.

2.5.1. Punto fijo

Se expresa mediante el siguiente polinomio:

$$(a_n...a_2a_1a_0.a_{-1}a_{-2}...a_{-m})_2 = a_n * 2^n + ... + a_2 * 2^2 + a_1 * 2 + a_0 + a_{-1} * 2^{-1} + a_{-2} * 2^{-2} + ... + a_{-m} * 2^{-m}$$

Esta representación convierte al conjunto de los reales en uno discreto, donde la distancia entre dos números es, como mínimo, de 2^{-m}. La parte entera del número se representa usando alguna de las formas de representación vistas para números enteros.

Existe un algoritmo para convertir la parte fraccionaria a otra base. El mismo consiste de multiplicaciones sucesivas de parte fraccionaria (SOLO la parte fraccionaria) por la base hasta llegar a un número entero o reconocer una periodicidad. Luego, se toman los valores enteros desde el primero al último, y estos corresponden a la representación.

Restas sucesivas:

0.4304
-0.4000 = $5^{-1}*2$

0.0304
-0.0000 = $5^{-2}*0$

0.0304
-0.0240 = $5^{-3}*3$

0.0064
-0.0064 = $5^{-4}*4$

0.0000

$(0.4304)_{10} = (0.2034)_5$

Restos de cocientes:

0.4304
* 5

2.1520
* 5

0.7600
* 5

3.8000
* 5

4.0000

$(0.4304)_{10} = (0.2034)_5$

Figura 2.5

2.5.2. Aproximación de reales

Cuando trabajamos con números reales, debemos representar un número con n dígitos en un sistema con m dígitos siendo $m < n$. Para esto existen dos soluciones:

- **Truncamiento**: Descartamos los dígitos fraccionarios de orden mayor a m. El error en peor caso es de 1 bit.

- **Redondeo**: Descartamos los dígitos fraccionarios de orden mayor a m y se suma 1 al bit menos significativo en caso de que el siguiente sea 1. El error en peor caso es de 0.5 bit.

2.5.3. Punto flotante

La representación de punto flotante se basa fuertemente en la idea de notación científica. En esta, los números se expresan en dos partes: la parte fraccionaria o mantisa y la parte exponencial que indica la potencia de 10 necesaria para incrementar la mantisa hacia el número esperado. El número n se escribe como $m * 10^e$, donde m y e son números enteros con signo:

$$\bullet - 725{,}832 = -7{,}25832 \times 102 = -725{,}832 \times 100$$

$$\bullet 3{,}14 = 0{,}314 \times 101 = 3{,}14 \times 100$$

$$\bullet 0{,}000001 = 0{,}1 \times 10 - 5 = 1{,}0 \times 10 - 6$$

Para unificar la notación se suele recurrir a la normalización exigiendo que $0{,}1 \leq m < 1$. Entonces, el número $(n)_b$ se escribe como $m * b^e$, donde m y e son números enteros con signo. Luego se los puede representar con un par (m, e) tal que m se encuentra normalizado.

Nota: ==

Acá pasa algo medio raro. Como que no queda claro qué es la mantisa. En el Tanenmbaum dice: Para evitar confusiones con una fracción convencional, la combinación del 1 implícito, el punto binario implícito y los 23 o 52 bits explícitos se denomina un **significando** y en lugar de una fracción o mantisa. Todos los números normalizados tienen un significado, s, en el rango $1 \leq s < 2$.

En el Stallings dice: La porción final de la palabra (23 bits en este caso) es el significando (haciendo referencia a la figura B-4 2.8). Con una nota al pie que dice: El término mantisa, que a veces se usa en lugar de significando, se considera obsoleto. Mantisa también significa “la parte fraccionaria de un logaritmo”, por lo que es mejor evitarlo en este contexto.

Así que se utilizará en ambos sentidos (solo la parte fraccionaria, o el significando completo), de manera indiferenciada.

==

En la representación de punto flotante, en general, se reservan un bit para el signo, varios para la mantisa, y otros tantos para el exponente. Tanto la mantisa como el exponente se representan usando alguno de los métodos mencionados anteriormente.

Veamos cómo obtener la representación de $(n)_{10}$ en punto flotante normalizado en base 2. Para ello tenemos que:

1. Encontrar e tal que: $2^{e-1} \leq |n| < 2^e$
2. Hallar la mantisa: por definición es $|m| = |n| / 2^e$
3. Representar m y e en base 2
4. Construir la representación: usualmente es signo, p bits para el exponente (en notación exceso) y q bits para la mantisa (en notación complemento).

Los números de punto flotante pueden ser usados para modelar a los números reales, aproximando a cada real al número representable más cercano, pero no de forma completa. Para visualizar esto, podemos dividir al los reales en siete regiones:

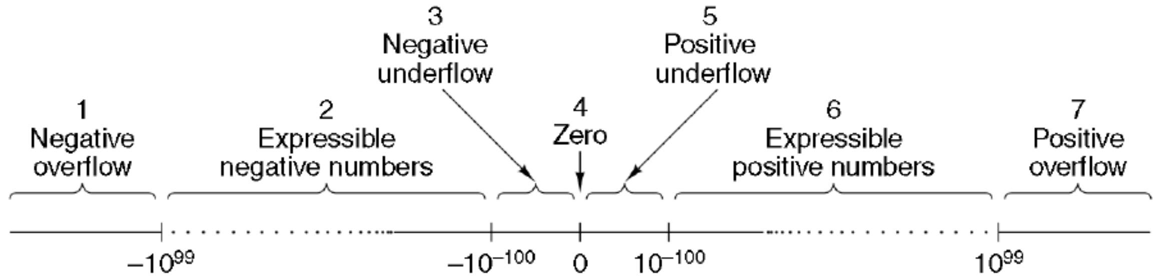


Figure B-1. The real number line can be divided into seven regions.

Figura 2.6

Notamos que no podemos representar con números flotantes ningún real en las regiones 1, 2, 5 o 7. Cuando una operación aritmética resulta en un número que cae en la región 1 o en la 7, estamos frente un **overflow error**, y por lo tanto la respuesta será incorrecta. La razón de esto ocurra es que estamos tratando de representar un conjunto infinito con uno finito, y por lo tanto este tipo de errores son inevitables.

De forma similar, un resultado que cae en la región 3 o en la 5 tampoco puede ser expresado bajo este sistema de representación. Esta situación se llama **underflow error**. Sin embargo, este tipo de errores son menos importantes, ya que normalmente podemos aproximar el resultado a 0 de forma satisfactoria.

Otra diferencia entre los números reales y los de punto flotantes es su *densidad*. El espaciamiento entre dos números expresables adyacentes no es constante a lo largo de la región 2 ni en la 6. La separación entre $+0{,}998 \times 10^{99}$ y $+0{,}999 \times 10^{99}$ es mucho mayor a la separación entre $+0{,}998 \times 10^{0}$ y $+0{,}999 \times 10^{0}$.

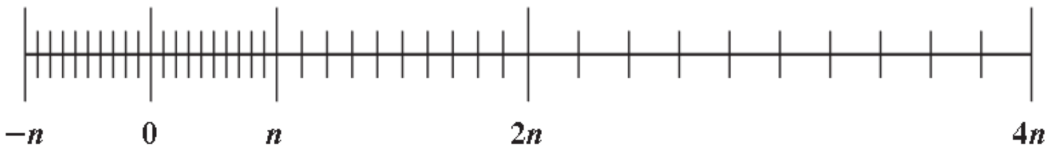


Figura 2.7

Sin embargo, el error relativo al momento de aproximar valores reales es aproximadamente el mismo para números grandes que para números chicos.

2.6. IEEE - 754

En 1985, el IEEE (Institute of Electrical and Electronic Engineers) publicó un estándar para números flotantes de simple y doble precisión:

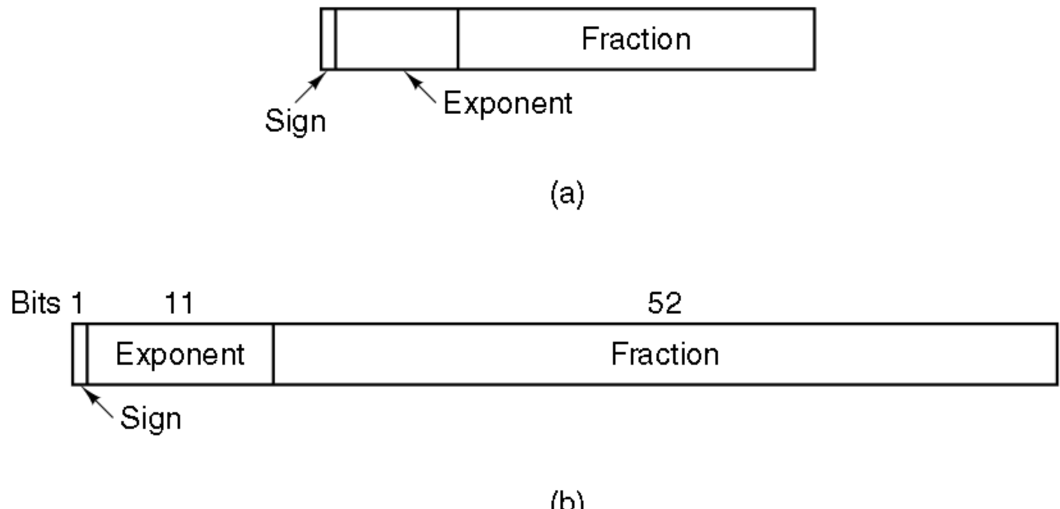


Figura 2.8

El IEEE 754 de simple precisión usa 32 bits:

- 8 bits para el exponente expresado en exceso a 127 (el exponente máximo que se representa es 127 y el mínimo -126. El -127 y el 128 se usan para valores especiales como ceros, infinitos y nans).
- 23 bits para la mantisa expresada en notación sin signo y normalizada a 1 en la mayoría de los casos.
- Un bit para el signo.

De forma que un real, en general, se representa así: signo $1.m_{22}...m_0 * 2^{e_7...e_0}$

El IEEE 754 de precisión doble usa 64 bits:

- 11 bits para el exponente representado en exceso 1023.
- 52 bits para la mantisa.
- Un bit para el signo.

Por otro lado, el standard IEEE 754 especifica números que no satisfacen los criterios de normalización explicados anteriormente:

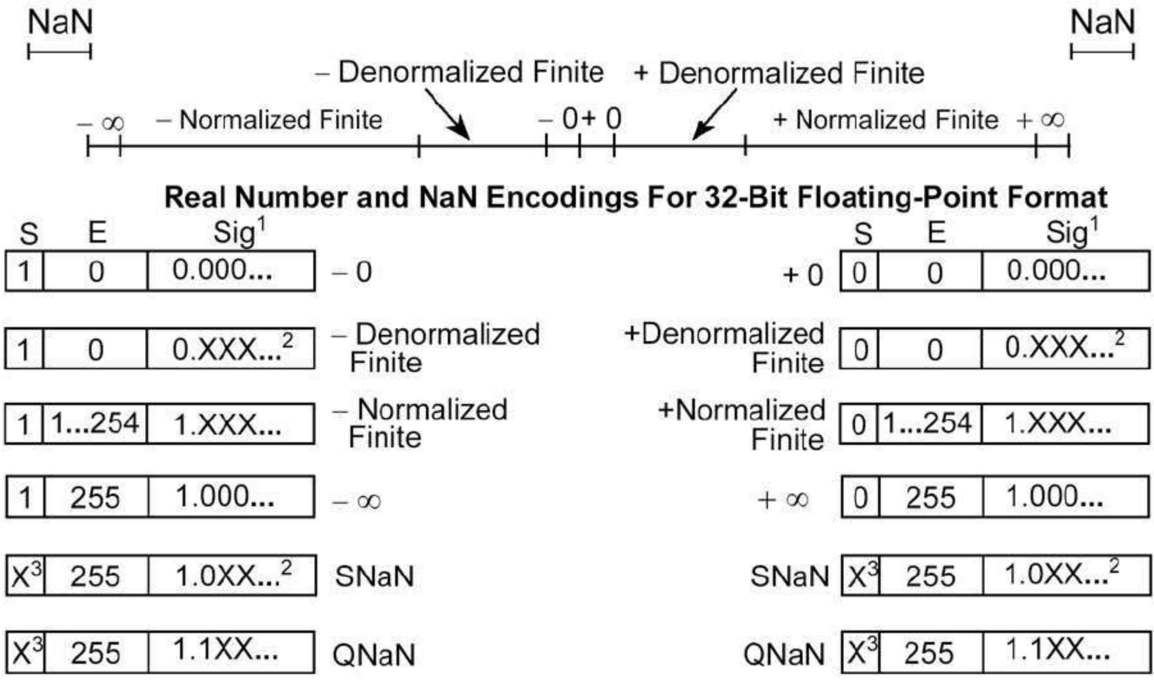


Figura 2.9

- **números denormalizados:**resultan de números que se aproximan demasiado a 0 y -126 no es suficiente para normalizar la mantisa.

- **Ceros con signo**: se interpreta como 0 pero el signo proporciona información sobre el resultado que hubiera dado la operación si la precisión hubiera sido suficiente

- **infinitos con signo:**números cuya magnitud es más grandes que cualquier número representable, también poseen signo para identificara el rango del resultado.

- **NaN (Not a number):** resultan de una operación inválida y hay de dos tipos, Signaling (SNaN), que lanzan una excepción, y Quiet (QNan), que son propagadas.

  Los números denormalizados (o normalizados a 0: s $0.m_{22}...m_0 * 2^{e_7 \cdots e_0}$) se usan para expandir la representatividad en el rango de números cercanos a 0. Tienen exponente 0 (-127 en exceso a 127), mantisa distinta a 0. La idea es que m_i pueden valer 0, y por lo tanto estamos aumentando nuestro rango representable para números cercanos a cero (de 2^{-128} a 2^{-127+m}, con $m = 23$ o 52), a costa de reducir el tamaño de la mantisa.

2.6.1. Aritmética de Punto Flotante

Las operaciones básicas de aritmética de punto flotante son la suma (resta) y la multiplicación y división.

Table 10.6 Floating-Point Numbers and Arithmetic Operations

Floating-Point Numbers	Arithmetic Operations
$X = X_S \times B^{X_E}$ $Y = Y_S \times B^{Y_E}$	$\left.\begin{array}{l} X + Y = (X_S \times B^{X_E - Y_E} + Y_S) \times B^{Y_E} \\ X - Y = (X_S \times B^{X_E - Y_E} - Y_S) \times B^{Y_E} \end{array}\right\} X_E \leq Y_E$ $X \times Y = (X_S \times Y_S) \times B^{X_E + Y_E}$ $\frac{X}{Y} = \left(\frac{X_S}{Y_S}\right) \times B^{X_E - Y_E}$

Figura 2.10

Adición y Substracción

En la aritmética de punto flotante, la adición y la substracción son más complejas que la multiplicación y la división. Esto se debe a que es necesario *alinear* a los significandos para que $X_E = Y_E$. Hay cinco etapas básicas del algoritmo para la suma y la resta:

1. **Obtener el complemento del substrayendo**: como la suma y la resta son idénticas, salvo por el cambio del signo del substrayendo, lo primero que hacemos es obtener el complemento del substrayendo en caso de que se trate de una resta.

2. **Revisar si tenemos ceros**: Si alguno de los operandos es 0, el otro es reportado como el resultado.

3. **Alinear los significandos**: La siguiente etapa consiste en manipular los números de manera que ambos exponentes sean iguales. Este alineamiento se consigue a través de shiftear a derecha (incrementado el exponente) al número más chico. Esta operación puede resultar en pérdida de dígitos, pero como estamos shifteando al más chico, podemos suponer que los dígitos que se pierden son poco significativos.

4. **Sumar o restar los significandos**: Se suman los significandos, tomando en cuenta el signo. Si ocurre overflow sobre los significandos, incrementamos en 1 el exponente, y shifteamos a derecha el resto de la mantisa en una posición. Si ocurre overflow sobre el exponente, esto sería reportado como un error.

5. **Normalizar el resultado**: La última etapa normaliza el resultado. Esta consiste en shiftear a izquierda los dígitos del significando hasta que el dígito más significativo sea distinto de cero. Cada shifteo decrementa el exponente.

Notemos que si la diferencia entre los exponentes es mayor a la cantidad de bits en la mantisa, entonces el resultado de la suma es equivalente a sumar cero al mayor.

Multiplicación y División

Primero consideraremos el caso de la multiplicación.

1. Lo primero que hacemos es revisar si alguno de los operandos son cero, y si alguno de ellos es cero, devolvemos cero como resultado.

2. El siguiente paso es sumar los exponentes, pero como estos están en notación exceso, tenemos que restar el exceso N de la suma.

3. El siguiente paso es multiplicar a los significandos, teniendo en cuenta el signo de los mismos. Esta se realiza como si se trataran de números enteros.

4. Después de que el producto sea haya calculado, el resultado es normalizado y redondeado.

Para el caso de la división,

1. Lo primero que hacemos es fijarnos si hay algún cero. Si el divisor es cero, se reporta un error (o infinito), y si el dividendo es cero, se devuelve 0.

2. Luego, se restan los exponentes, y como estamos trabajando con exceso N, tenemos que sumarle N al resultado.

3. El siguiente paso es dividir a los significandos.

4. Esto es seguido de la usual normalización y redondeo.

2.7. Representación de Caracteres

2.7.1. ASCII

El ASCII (American Standard Code for Information Interchange) utiliza 7 bits para representar los caracteres, aunque inicialmente empleaba un bit adicional de paridad para detectar los errores en la transmisión. Todos los sistemas informáticos actuales utilizan el código ASCII o una extensión compatible para representar textos.

El ASCII define códigos para 33 caracteres no imprimibles, de los cuales la mayoría son caracteres de control obsoletos que tienen efecto sobre cómo se procesa el texto, otros 95 caracteres imprimibles, de los cuales 10 son dígitos, 52 son letras (mayúsculas y minúsculas), 32 son caracteres especiales, y por último el carácter de espacio.

Tabla ASCII

Char	Dec	Oct	Hex	Char	Dec	Oct	Hex	Char	Dec	Oct	Hex
(sp)	32	0040	0x20	@	64	0100	0x40	`	96	0140	0x60
!	33	0041	0x21	A	65	0101	0x41	a	97	0141	0x61
"	34	0042	0x22	B	66	0102	0x42	b	98	0142	0x62
#	35	0043	0x23	C	67	0103	0x43	c	99	0143	0x63
$	36	0044	0x24	D	68	0104	0x44	d	100	0144	0x64
%	37	0045	0x25	E	69	0105	0x45	e	101	0145	0x65
&	38	0046	0x26	F	70	0106	0x46	f	102	0146	0x66
'	39	0047	0x27	G	71	0107	0x47	g	103	0147	0x67
(	40	0050	0x28	H	72	0110	0x48	h	104	0150	0x68
)	41	0051	0x29	I	73	0111	0x49	i	105	0151	0x69
*	42	0052	0x2a	J	74	0112	0x4a	j	106	0152	0x6a
+	43	0053	0x2b	K	75	0113	0x4b	k	107	0153	0x6b
,	44	0054	0x2c	L	76	0114	0x4c	l	108	0154	0x6c
-	45	0055	0x2d	M	77	0115	0x4d	m	109	0155	0x6d
.	46	0056	0x2e	N	78	0116	0x4e	n	110	0156	0x6e
/	47	0057	0x2f	O	79	0117	0x4f	o	111	0157	0x6f
0	48	0060	0x30	P	80	0120	0x50	p	112	0160	0x70
1	49	0061	0x31	Q	81	0121	0x51	q	113	0161	0x71
2	50	0062	0x32	R	82	0122	0x52	r	114	0162	0x72
3	51	0063	0x33	S	83	0123	0x53	s	115	0163	0x73
4	52	0064	0x34	T	84	0124	0x54	t	116	0164	0x74
5	53	0065	0x35	U	85	0125	0x55	u	117	0165	0x75
6	54	0066	0x36	V	86	0126	0x56	v	118	0166	0x76
7	55	0067	0x37	W	87	0127	0x57	w	119	0167	0x77
8	56	0070	0x38	X	88	0130	0x58	x	120	0170	0x78
9	57	0071	0x39	Y	89	0131	0x59	y	121	0171	0x79
:	58	0072	0x3a	Z	90	0132	0x5a	z	122	0172	0x7a
;	59	0073	0x3b	[	91	0133	0x5b	{	123	0173	0x7b
<	60	0074	0x3c	\	92	0134	0x5c	|	124	0174	0x7c
=	61	0075	0x3d	]	93	0135	0x5d	}	125	0175	0x7d
>	62	0076	0x3e	^	94	0136	0x5e	~	126	0176	0x7e
?	63	0077	0x3f	_	95	0137	0x5f				

Figura 2.11: Tabla ASCII

2.7.2. Unicode

Dado que ASCII fue armado para representar caracteres del alfabeto latino, se necesitaba desarrollar otro código compatible que permitiera representar los caracteres de otros alfabetos, es decir, universalizar la codificación de texto a cualquier idioma. Así se desarrolló Unicode.

Unicode representa caracteres con 16 bits, razón por la cual tiene la capacidad de representar la mayoría de los caracteres de cualquier lenguaje, y también posee un mecanismo de extensión por si los anteriores no son suficientes.

Es soportado a alto nivel por sistemas operativos, formatos de distribución de documentos como XML y lenguajes de programación como Java, C#, Python, etc.

2.7.3. UTF-8

El 8-bit Unicode Transformation Format es una norma de transmisión de longitud variable para caracteres de Unicode. El mismo utiliza grupos de bytes para representar el estándar Unicode de distintos alfabetos, y utiliza de 1 a 4 bytes por carácter pero su transferencia se hace sobre 8 bits, siendo así compatible con los servicios de correo.

Ventajas

- La secuencia de bytes que representa a un carácter nunca es prefijo de ningún otro caracter, lo que las hace fáciles de manipular
- El primer byte codifica a longitud de la secuencia correspondiente al caracter
- Está diseñado para no generar conflicto con sistemas legacy, y así evitar incompatibilidades (no utiliza bytes de control ASCII o ISO-8859-1)

Desventajas

- Es de longitud variable, haciendo más complicada su implementación.
- Necesita una capa abstracción más entre el software del sistema y la arquitectura de la computadora.
- La necesidad de analizadores de UTF-8 podrían introducir nuevos errores.

Capítulo 3

Lógica Digital: Parte 1

3.1. Introducción

Una computadora puede ser pensada por capas de abstracción. Cada una de estas capas proporciona una vista de la computadora, de manera tal que se permite hacer un uso determinado de las herramientas que ofrece la capa inferior. Es decir, cada nivel brinda a la capa superior una **interfaz** sobre la máquina que está debajo.

Nivel 6	Usuario	Programa ejecutables
Nivel 5	Lenguaje de alto nivel	C++, Java, Python, etc.
Nivel 4	Lenguaje ensamblador	Assembly code
Nivel 3	Software del sistema	Sistema operativo, bibliotecas, etc.
Nivel 2	Lenguaje de máquina	Instruction Set Architecture (ISA)
Nivel 1	Unidad de control	Microcódigo / hardware
Nivel 0	Lógica digital	Circuitos, compuertas, memorias

Figura 3.1

A nivel 0 tenemos la construcción de la computadora, desde el punto de su organización, su lógica digital. Es decir, estamos en presencia de componentes electrónicos, corriente viajando entre compuertas lógicas. Por encima de eso, tenemos una capa (Nivel 1), que cuenta con la **unidad de control**, la cual provee a las capas superiores un lenguaje rudimentario, más estructurado, y más sencillo de utilizar, que permite manipular estas señales.

Por encima de este, tenemos el lenguaje de máquina (Nivel 2). Este nivel ofrece al resto de la máquina una **tira de bits** que puede ser leída, sistemáticamente y de **forma cíclica**. Esta tira de bits le da indicaciones para la realización de determinadas acciones, que nos terminan ofreciendo un lenguaje de programación nativo del procesador, llamado **ISA** (Instruction Set Architecture).

Por encima de la ISA, están todas las cosas que se pueden programar y ejecutar en ese procesador. En particular, lo que primero se ejecuta por encima del procesador es un sistema operativo, que permite **administrar** los recursos de la computadora. Por encima de este, tenemos ensambladores, compiladores, lenguajes de alto nivel, y finalmente aplicaciones de usuarios.

Solo nos interesa:

- **Nivel 0**: el estudio de la lógica digital sobre la que se construye una computadora.
- **Nivel 1**: cómo se administra la lógica digital para poder ejecutar secuencialmente programas.
- **Nivel 2**: definir un lenguaje de programación para una computadora.

El nivel de la lógica digital es el nivel de la organización, y ofrece a la capa de microprogramación (o de la unidad de control) una interfaz para la manipulación de **señales**. En esta capa, la manipulación de señales es **explícita**, es decir sabemos en cada lugar dónde hay un 0 o un 1, donde hay corriente o no fluyendo de una compuerta a otra.

Los elementos de una computadora están construidos a partir de componentes electrónicas, las cuales van a depender de cómo va a estar representada la información (no es lo mismo operar con números en Ca2 que con Magnitud Signada).

Recordemos que estamos trabajando bajo el modelo Von Neumann - Turing, y por lo tanto, los programas y datos (sobre los que operan los programas) se encuentran almacenados en la **memoria**.

Entonces, la idea de que los bits son porciones de información, como mínima unidad de información, va a determinar no solo cómo se almacenan de los datos, sino que además van a determinar los mecanismos de control de nuestra máquina, permitiendo que nuestro programa pueda enviar señales de un lugar a otro. Esas señales son, por ejemplo, la información de la memoria que se trae hacia el **procesador**.

3.2. Algebra Booleana

3.2.1. Introducción

El cálculo de una computadora digital está basado en el almacenamiento y el procesamiento de información binaria, ya que es más sencillo (y barato) identificar entre únicamente dos estados, y por lo tanto hay una menor generación de errores.

Los circuitos operan con valores lógicos, estos son:

- Verdadero $\iff$ 1 $\iff$ $2 - 5V(+-10\,\%)$
- Falso $\iff$ 0 $\iff$ 0 - 0.8 V

A continuación veremos cómo es posible implementar dichos circuitos mediante la lógica digital, específicamente circuitos combinatorios y secuenciales. Veremos un repaso del álgebra Booleana (que es la fundación matemática para la lógica digital), luego compuertas, y finalmente circuitos combinatorios y secuenciales.

George Boole desarrolló métodos para expresar procesos lógicos usando símbolos algebraicos, creando una rama de las matemáticas conocida como lógica simbólica. Esta hace uso de variables y operaciones lógicas. Por lo tanto, una variable puede valer o bien "1" (`TRUE`) o "0" (`FALSE`).

Las operaciones lógicas básicas son `AND`, `OR` y `NOT` (el `NOT` tiene la mayor precedencia, luego el `AND`, y luego el `OR`). Otras operaciones lógicas importantes son `XOR` (o-exclusivo), `NAND` (`AND` negado), `NOR` (`OR` negado).

Un operador booleano es descripto usando tablas de verdad:

(a) Boolean Operators of Two Input Variables

P	Q	NOT P ($\overline{P}$)	P AND Q ($P \cdot Q$)	P OR Q ($P + Q$)	P NAND Q ($\overline{P \cdot Q}$)	P NOR Q ($\overline{P + Q}$)	P XOR Q ($P \oplus Q$)
0	0	1	0	0	1	1	0
0	1	1	0	1	1	0	1
1	0	0	0	1	1	0	1
1	1	0	1	1	0	0	0

Figura 3.2

Las mismas pueden ser generalizadas para más de dos operandos (excepto **NOT**).

(b) Boolean Operators Extended to More than Two Inputs (A, B, . . .)

Operation	Expression	Output = 1 if
AND	$A \cdot B \cdot \ldots$	All of the set {A, B, ...} are 1.
OR	$A + B + \ldots$	Any of the set {A, B, ...} are 1.
NAND	$\overline{A \cdot B \cdot \ldots}$	Any of the set {A, B, ...} are 0.
NOR	$\overline{A + B + \ldots}$	All of the set {A, B, ...} are 0.
XOR	$A \oplus B \oplus \ldots$	The set {A, B, ...} contains an odd number of ones.

Figura 3.3

3.2.2. Identidades Lógicas

La siguiente tabla resume las identidades clave del álgebra Booleana. Hay dos clases de identidades: las reglas básicas (postulados), que son definidos sin pruebas, y otras identidades que pueden ser derivados de dichos postulados.

Table 11.2 Basic Identities of Boolean Algebra

Basic Postulates		
$A \cdot B = B \cdot A$	$A + B = B + A$	Commutative Laws
$A \cdot (B + C) = (A \cdot B) + (A \cdot C)$	$A + (B \cdot C) = (A + B) \cdot (A + C)$	Distributive Laws
$1 \cdot A = A$	$0 + A = A$	Identity Elements
$A \cdot \overline{A} = 0$	$A + \overline{A} = 1$	Inverse Elements
Other Identities		
$0 \cdot A = 0$	$1 + A = 1$	
$A \cdot A = A$	$A + A = A$	
$A \cdot (B \cdot C) = (A \cdot B) \cdot C$	$A + (B + C) = (A + B) + C$	Associative Laws
$\overline{A \cdot B} = \overline{A} + \overline{B}$	$\overline{A + B} = \overline{A} \cdot \overline{B}$	DeMorgan's Theorem

Figura 3.4

3.2.3. Fórmulas Equivalentes

Varias fórmulas pueden tener la misma tabla de verdad, es decir que son lógicamente equivalentes. Existen dos formas canónicas (es decir que, para dos fórmulas equivalentes cualesquiera, se obtiene la misma forma canónica) para traducir desde una tabla de verdad a una función lógica:

- **Suma de Productos** (SOP): Se toman las filas que evalúan a 1, y luego se describe la fórmula a través de los valores de entrada de dichas filas.
- **Producto de Sumas** (POS)

En este punto parecería que una función Booleana podría ser implementada usando SOP o POS, dependiendo si la tabla contiene más 1s (SOP) o más 0s (POS). Sin embargo, hay otras cosas a tener en cuenta:

- Normalmente es posible obtener una expresión Booleana más simple que no esté en una forma canónica.
- A veces es preferible implementar las funciones usando un único tipo de compuerta (`NAND` o `NOR`), a pesar de no ser la implementación con la menor cantidad de compuertas.

Hay tres métodos para poder simplificar las expresiones:

- **Simplificaciones Algebraicas**: Aplicar identidades para reducir la expresión.
- **Mapas de Karnaugh**: menos de 4 variables.
- **Tablas de Quine-McCluskey**: más de 4 variables, útil para automatizar con un programa.

3.3. Circuitos booleanos

Lo que necesitamos para construir circuitos electrónicos son compuertas lógicas. Estas nos van a servir para combinar y producir nuevos valores. Hasta ahora estas compuertas tenían un comportamiento algebraico, y ahora le vamos a dar un comportamiento eléctrico: vamos a construir circuitos físicos que evalúan a las funciones booleanas.

En un circuito digital solo hay presentes dos valores lógicos. En general, una señal alrededor de los 0 V representa un valor (0, Falso, apagado), y una señal alrededor de los 5 V representa el otro valor (1, Verdadero, prendido). Voltajes por fuera de esos rangos no están permitidos. La componente elemental que vamos a utilizar para construir circuitos electrónicos son los **transistores**.

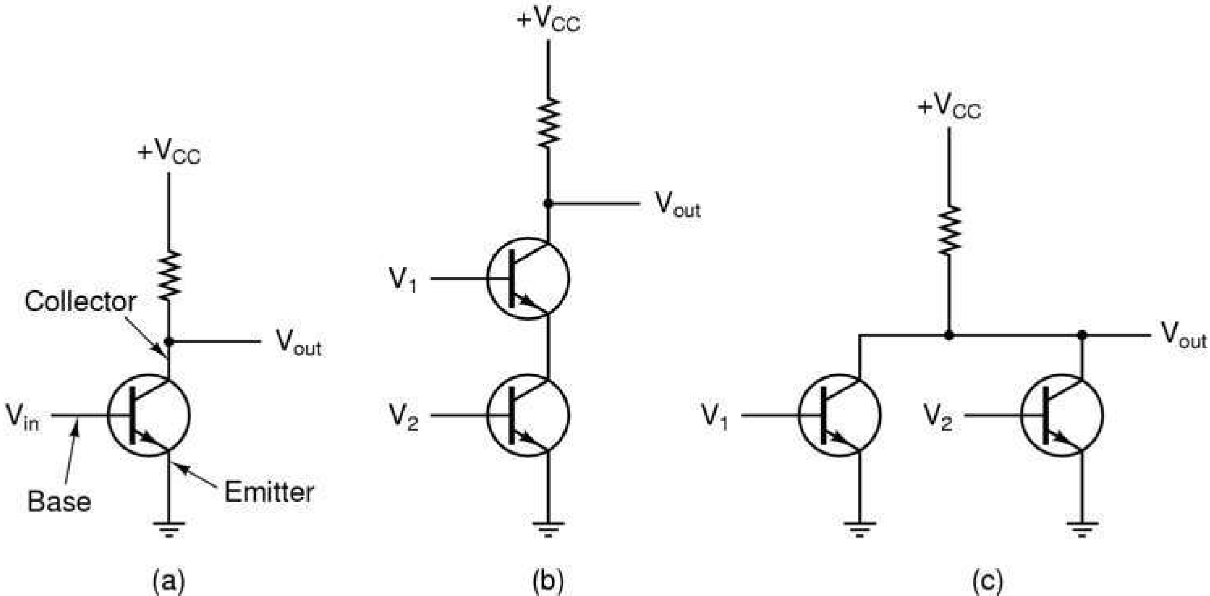


Figura 3.5: Esquema de una compuerta `NOT` (a), una `NAND` (b) y una `NOR` (c).

Podemos ver transistores bipolares (los círculos) embebidos en distintos circuitos. Un transistor es

un dispositivo que tiene tres puntas o conexiones: el colector, la base, y el emisor. A estas conectamos las diferentes entradas y salidas.

Pensemos en que el transistor es una llave de luz. La diferencia es que no necesitamos que una persona la prenda o apague, sino que necesitamos que alguien conecte el voltaje de alimentación del circuito en V_{in}.

Entonces, el transistor, dependiendo del voltaje en V_{in}, permite abrir o cerrar el circuito. Cuando se abre, la "luz" se apaga, porque no hay corriente fluyendo entre el emisor y el colector (la luz eléctrica fluye del emisor al colector). Si en el camino encuentra una lámpara, esta se prende.

Cuando V_{in} está por debajo de un cierto valor crítico (digamos 5V), es decir que V_{in} vale alrededor de 0, el transistor se apaga, y actúa como una resistencia infinita. Esto genera que en Vout veamos un resultado muy parecido a $+V_{cc}$ (digamos 5V). En cambio, cuando en V_{in} se alanza un valor mayor al valor crítico ($V_{in} = 5V$), el transistor se prende, y actúa como si fuese un cable (resistencia cercana a 0), causando que V_{out} sea tirado a tierra (por convención 0V). Entonces, el transistor termina **invirtiendo** la señal de V_{in}.

3.4. Compuertas lógicas

Las compuertas son el **bloque fundamental** de los circuitos en lógica digital. Una **compuerta lógica** es un dispositivo electrónico que produce un resultado en base a un conjunto de valores de entrada, de forma casi instantánea, retrasado únicamente por el tiempo que tarda la propagación de las señales a través de la compuerta (conocido como gate delay). Se corresponden exactamente con las funciones booleanas que vimos, e implementan las aritmética y lógica de la CPU.

Estas compuertas se implementan haciendo uso de transistores que permiten modificar una señal de salida de un circuito dependiendo de una señal de entrada dado que actúan como una llave controlada por señal. En resumen, la componente física básica de una computadora es el transistor, el elemento lógico básico es la compuerta lógica.

Las compuertas básicas utilizadas en lógica digital son `AND`, `OR`, `NOT`, `NAND`, `NOR` y `XOR`. Cada compuerta es definida a partir de un símbolo gráfico, notación algebraica, y una tabla de verdad.

Name	Graphical Symbol	Algebraic Function	Truth Table
AND	A, B → F	$F = A \bullet B$ or $F = AB$	A B | F 0 0 | 0 0 1 | 0 1 0 | 0 1 1 | 1
OR	A, B → F	$F = A + B$	A B | F 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 1
NOT	A → F	$F = \overline{A}$ or $F = A'$	A | F 0 | 1 1 | 0
NAND	A, B → F	$F = \overline{AB}$	A B | F 0 0 | 1 0 1 | 1 1 0 | 1 1 1 | 0
NOR	A, B → F	$F = \overline{A + B}$	A B | F 0 0 | 1 0 1 | 0 1 0 | 0 1 1 | 0
XOR	A, B → F	$F = A \oplus B$	A B | F 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 0

Figure 11.1 Basic Logic Gates

Figura 3.6: Esquema de una compuerta `NOT` (a), una `NAND` (b) y una `NOR` (c).

3.4.1. Compuertas NAND y NOR

Usualmente no se usan todos las compuertas en una implementación. El diseño y fabricación son más sencillos si únicamente se usan uno o dos tipos de compuertas. Por este motivo, es importante identificar conjuntos funcionalmente completos de compuertas lógicas, por ejemplo:

- `AND, NOT, OR`
- `AND, NOT`
- `OR, NOT`
- `NAND`
- `NOR`

Está claro que `AND, OR` y `NOT` constituyen un conjunto funcionalmente completo, porque representan las tres operaciones del álgebra Booleana. Podemos ver en la siguiente imagen cómo las funciones `AND`, `OR` y `NOT` pueden ser implementadas únicamente con compuertas `NAND`, o con compuertas `NOR`.

Además, tanto las compuertas NAND como las compuertas NOR resultan sumamente baratas de fabricar dado que requieren únicamente de 2 transistores. Es por este motivo que los circuitos digitales suelen ser implementado únicamente con compuertas NAND o NOR.

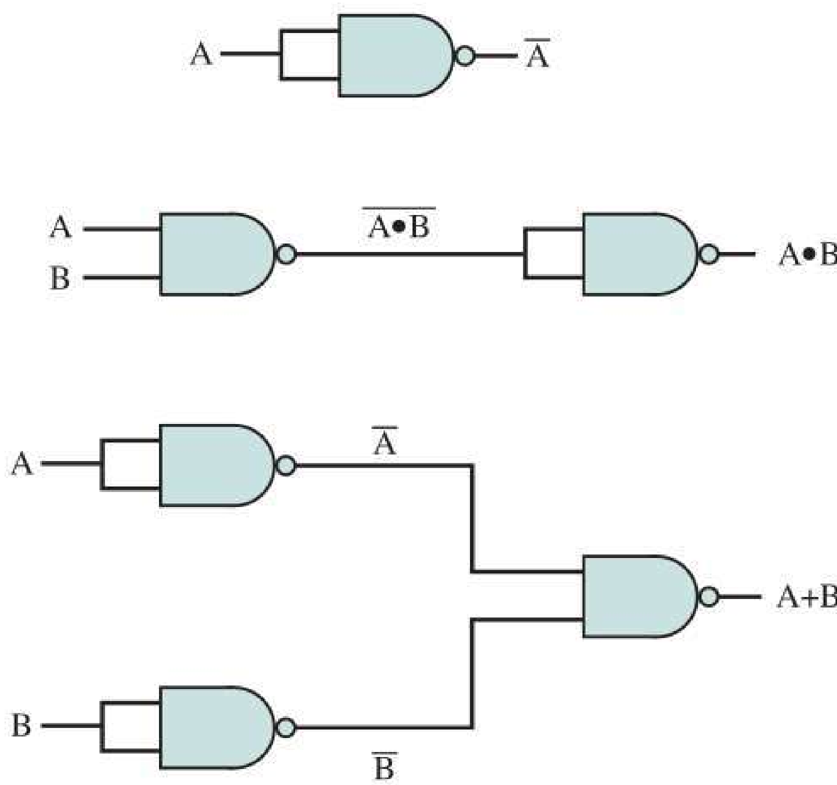


Figura 3.7: Compuertas NAND como conjunto funcionalmente completo

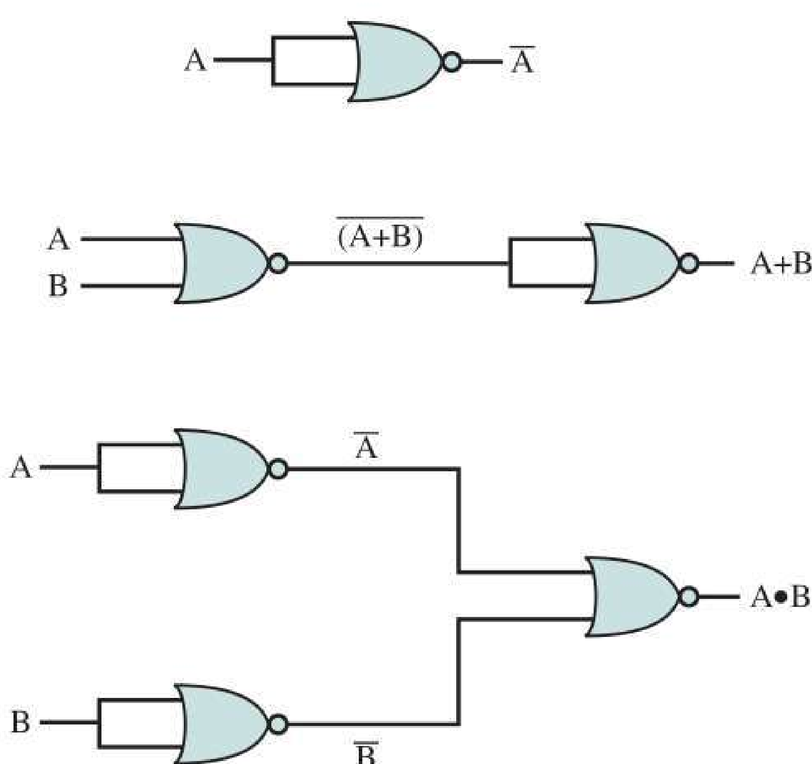
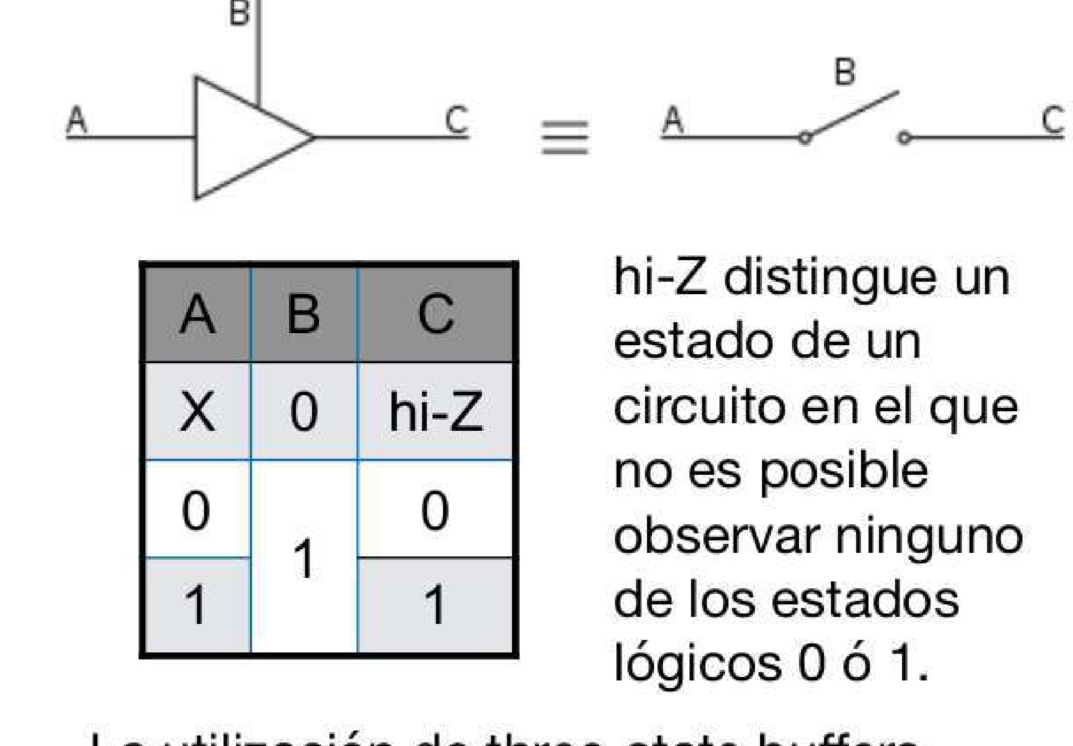


Figura 3.8: Compuertas NOR como conjunto funcionalmente completo

3.5. Buffer de tres estados

La utilización de buffers de tres estados permite que varios dispositivos compartan una misma salida si se tiene la precaución de que solo uno de estos tenga habilitada la salida. Cuando el valor de la señal de control es 1, el buffer cierra el circuito (actuando como cable), y cuando es 0, lo deja abierto (como si alguien hubiese removido esa parte del circuito).



A	B	C
X	0	hi-Z
0	1	0
1	1	1

Figura 3.9: Tabla de un buffer de tres estados

Esto se utiliza cuando, por ejemplo, se utilizan las mismas líneas (cables) para valores de entrada, y para valores de salida. Un ejemplo de esto es cuando trabajamos con memorias.

Cuando escribimos, necesitamos líneas que nos permitan ingresar los valores que queremos escribir, pero no nos interesa obtener el valor actual almacenado. En cambio, cuando estamos leyendo, no necesitamos las líneas para ingresar datos a almacenar, pero sí necesitamos líneas para obtener el valor almacenado.

Entonces, para economizar, podemos utilizar las mismas líneas, como entrada (escritura), y como salida (lectura). Si no utilizamos un buffer de tres estados (ya sea **inverting** o **noninverting**), el chip de la memoria podría intentar devolver valores, es decir, forzar a cada línea a tener un valor específico, incluso en escrituras, pudiendo interferir con los valores de entrada. Por este motivo, utilizamos este tipo de llave electrónica que hace o deshace conexiones en fracciones de nanosegundos.

Además, estos circuitos **amplifican** señales, por lo que, cuando se requiere alimentar muchas entradas, a veces son utilizados sin utilizar su propiedad de "llave".

3.6. Circuitos Combinatorios

Toda la lógica de cómputo de una computadora está caracterizada por un **circuito combinatorio**. Un circuito combinatorio es un conjunto de compuertas interconectadas cuya salida, está completamente y únicamente determinada por la entrada actual. Al igual que en las compuertas, la salida aparece casi inmediatamente después de ingresar la entrada, únicamente con la tardanza producto de las gate delay. En general, un circuito combinatorio consiste en n entradas que genera m salidas.

3.6.1. Sumador

Vamos a ver cómo podemos implementar la función de suma. La suma aritmética difiere de la suma binaria (`OR`) en que su resultado incluye un *carry*. Sin embargo, la suma puede ser resuelta en términos booleanos, cómo podemos ver en la siguiente tabla:

Table 11.9 Binary Addition Truth Tables

(a) Single-Bit Addition

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

(b) Addition with Carry Input

C_{in}	A	B	Sum	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

0	0	1	1
+0	+1	+0	+1
0	1	1	10

Figura 3.10: Tabla de Verdad de un Sumador (simple y completo) de un Bit

Semi-Sumador (Half-Adder:

Empezamos con el caso sin `CarryIn`. Podemos ver que la `Suma` termina siendo un `XOR` entre las entradas A y B, mientras que la salida `CarryOut` es equivalente a un `AND` entre A y B, y por lo tanto lo podemos implementar esta tabla de verdad de la siguiente manera:

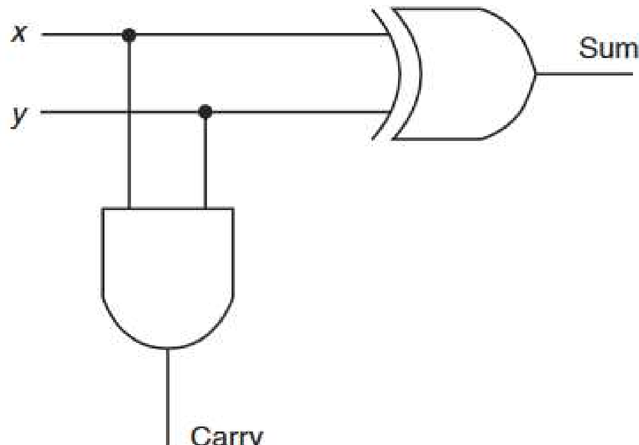


Figura 3.11: Circuito de un Semi-Sumador de un bit

Sumador-Completo (Full-Adder)

El Semi-Sumador es un circuito sencillo que nos permite sumar dos bits, sin embargo no nos alcanza para poder sumar n-bits. La idea es poder conectar el CarryOut a una entrada adicional, para que este acarreo se tome en cuenta en la suma del siguiente dígito. En base a esto, nos construimos un circuito con 3 entradas (x, y, CarryIn) y dos salidas (Suma, CarryOut) llamado *Full-Adder*.

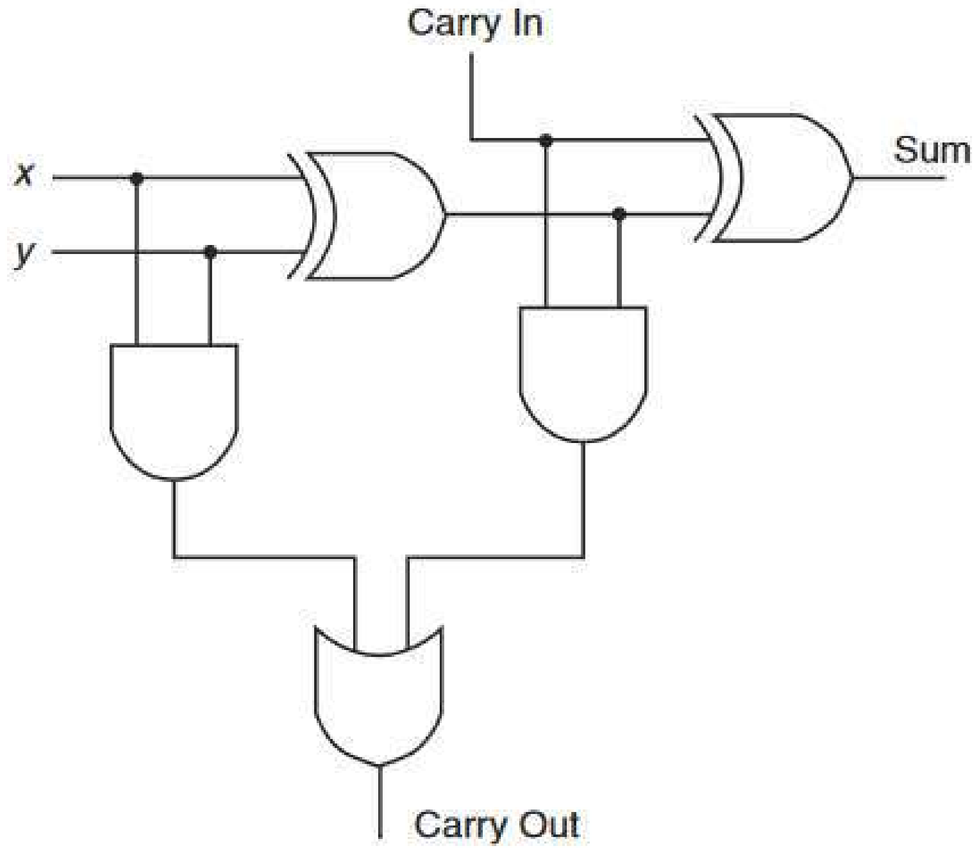


Figura 3.12: Circuito de un Sumador Completo de un Bit

Notemos que este circuito está compuesto por dos *Half-Adders* y una compuerta OR.

Sumador de n-bits

Ahora queremos poder realizar sumas de n-bits. Esto se puede lograr poniendo un conjunto de sumadores de 1-bit tal que el `carry` de cada uno es provisto como entrada para el siguiente. Un sumador de 4-bits se puede ver en la siguiente imagen:

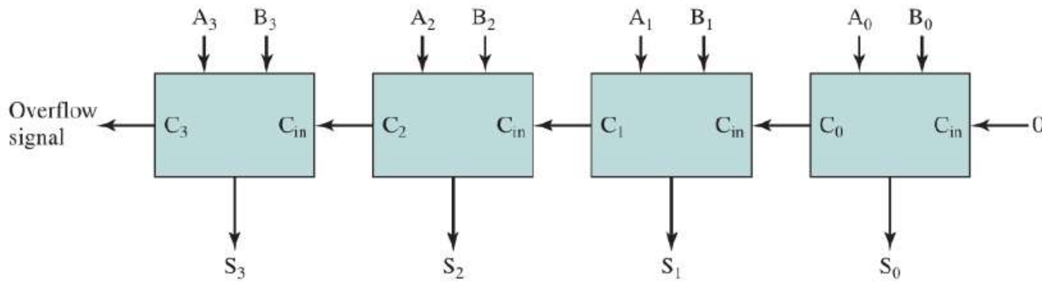


Figura 3.13: Circuito de un Sumador Completo de 4-Bits

Este tipo de sumador se conoce cómo **Ripple Carry Adder** (sumador en cascada). Notemos que la salida de cada Sumador Completo depende del *carry* del anterior, y por lo tanto hay un retardo acumu-

lativo desde el bit menos significativo hasta el más significativo, ya que cada *Full Adder* acumula un gate delay, por lo que tratar de hacer sumadores en cascada demasiado grandes puede resultar en tardanzas demasiado altas. Es por este motivo que existen modificaciones del mismo que atacan este problema.

Look-ahead Adder

Esto se podría mejorar si pudiésemos conseguir que cada sumador de 1-bit funcione de forma independiente. Para esto necesitaríamos conocer el carry de antemano (carry lookahead).

Nos gustaría obtener una expresión que especifique el carry input para cada etapa del sumador de n-bits sin tener que hacer referencia a resultados anteriores. Por ejemplo:

$$C0 = A_0.B_0$$
$$C1 = A_1.B_1 + A_1.A_0.B_0 + B_1.A_0.B_0$$
$$C2 = ...$$

Este proceso puede hacerse indefinidamente. Sin embargo, para números grandes, este acercamiento se vuelve excesivamente complejo, ya que esta expresión crece de forma exponencial (2^{n-1}). Por lo tanto, utilizar carry lookahead solo tiene sentido para 4-8 bits a la vez. Podemos ver en la siguiente figura como resultaría un sumador de 32 bits:

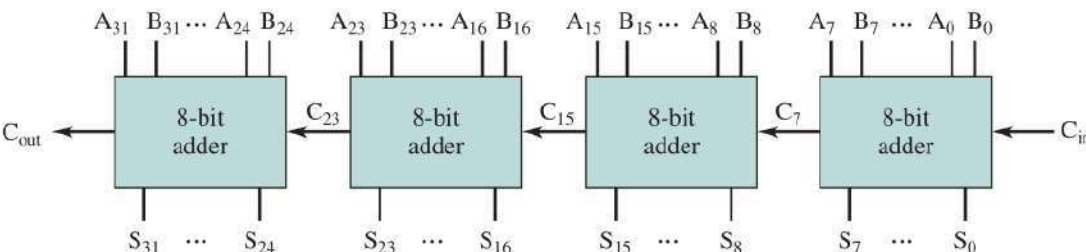


Figura 3.14

Carry Select Adder

La idea de este sumador es dividir el sumador en n, de manera tal de generar 2^n resultados distintos, uno por cada carry que se haya salteado.

Supongamos un caso sencillo de este tipo de sumador. Tenemos un sumador normal de 32 bits, y lo separamos en 2, la parte alta de 16-bits y la parte baja de 16-bits. Cuando la suma empieza, la parte alta todavía no puede trabajar, porque no conoce de antemano el `carry in`.

Ahora, modificaremos el circuito de manera de que en vez de tener una única parte alta, tenga dos sumadores de 16-bits para la parte alta. Nuestro circuito consiste en 3 sumadores de 16-bits, el primero para la parte baja, el segundo para la parte alta (asumiendo cin = 0), y el tercero para la parte alta (asumiendo cin = 1).

Luego de hacer las sumas, solo uno de los dos sumadores para la parte alta tendrá el resultado correcto, y como para esta altura ya tenemos el resultado de la parte baja, podemos seleccionar el mismo con un multiplexor.

Este truco reduce el tiempo de la suma por la mitad, y puede ser replicado de forma recursiva para los sumadores de 16-bits, dividirlos en sumadores de 8-bits, etc.

3.6.2. Decodificador

Un Decodificador es un circuito combinatorio que tiene n líneas de entrada y 2^n líneas de salida (una por cada combinación posible de los valores de entrada), de las cuales solo una devuelve 1 (y el resto 0), dependiendo de los valores de entrada.

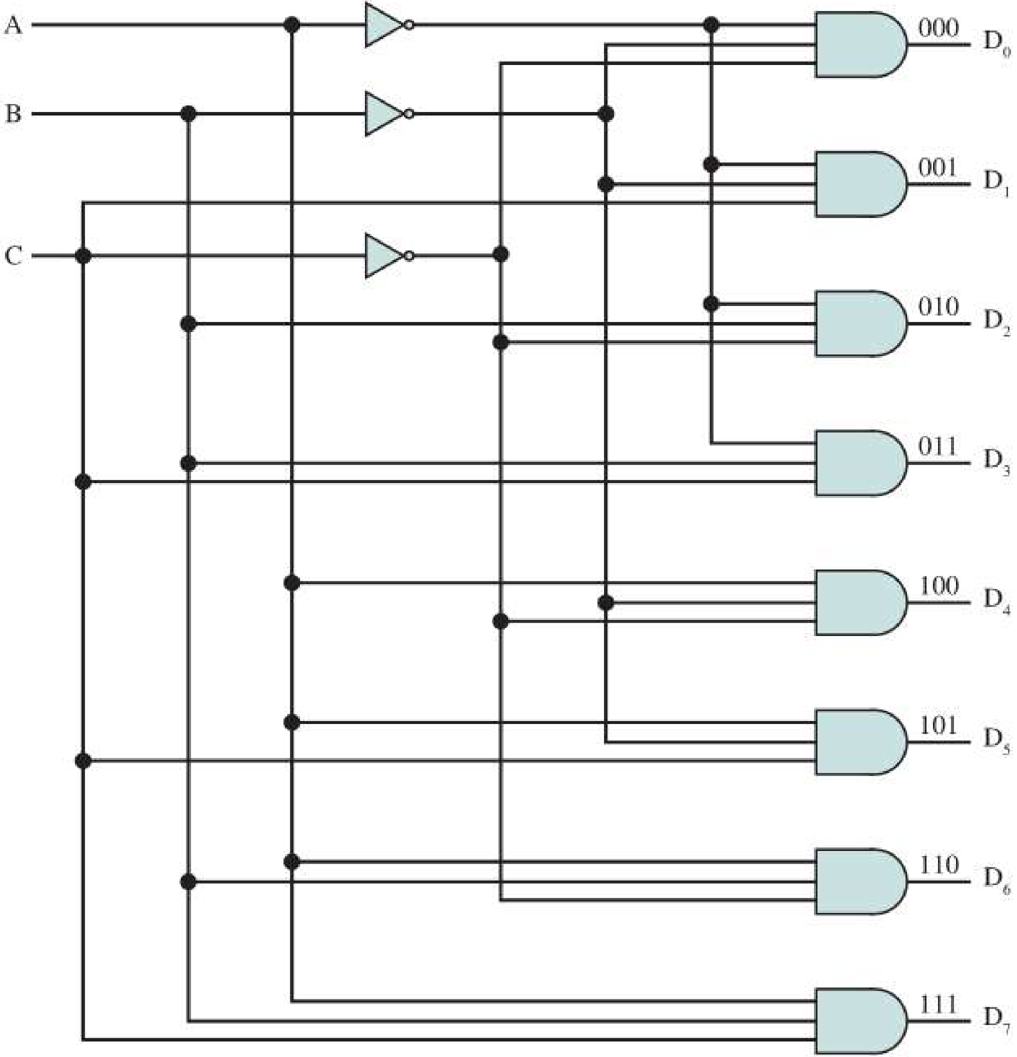


Figure 11.15 Decoder with 3 Inputs and $2^3 = 8$ Outputs

Figura 3.15: Circuito de un decodificador de 3 entradas y $2^3 = 8$ salidas

Un ejemplo de uso es la decodificación de direcciones. En el modelo de Von Neumann - Turing, las distintas posiciones de la memoria van a ser accesibles a partir de una combinación **única** de señales. que llamaremos **señales de direcciones**, que terminan siendo tiras de bits. Para elegir el circuito particular en el que se tiene la información buscada, vamos a tener que *decodificar* estas direcciones.

3.6.3. Multiplexor

Un Multiplexor conecta, típicamente, 2^k entradas (D0, D1, ..., D(2^{k-1})) a una única salida (F), mediante el uso de k señales de control ($S_1, .., S_k$). Una de estas entradas termina siendo seleccionada para ser devuelta como salida. Los Multiplexores son usados en circuitos digitales para el control de señales y navegación de información. Estos son fácilmente escalables.

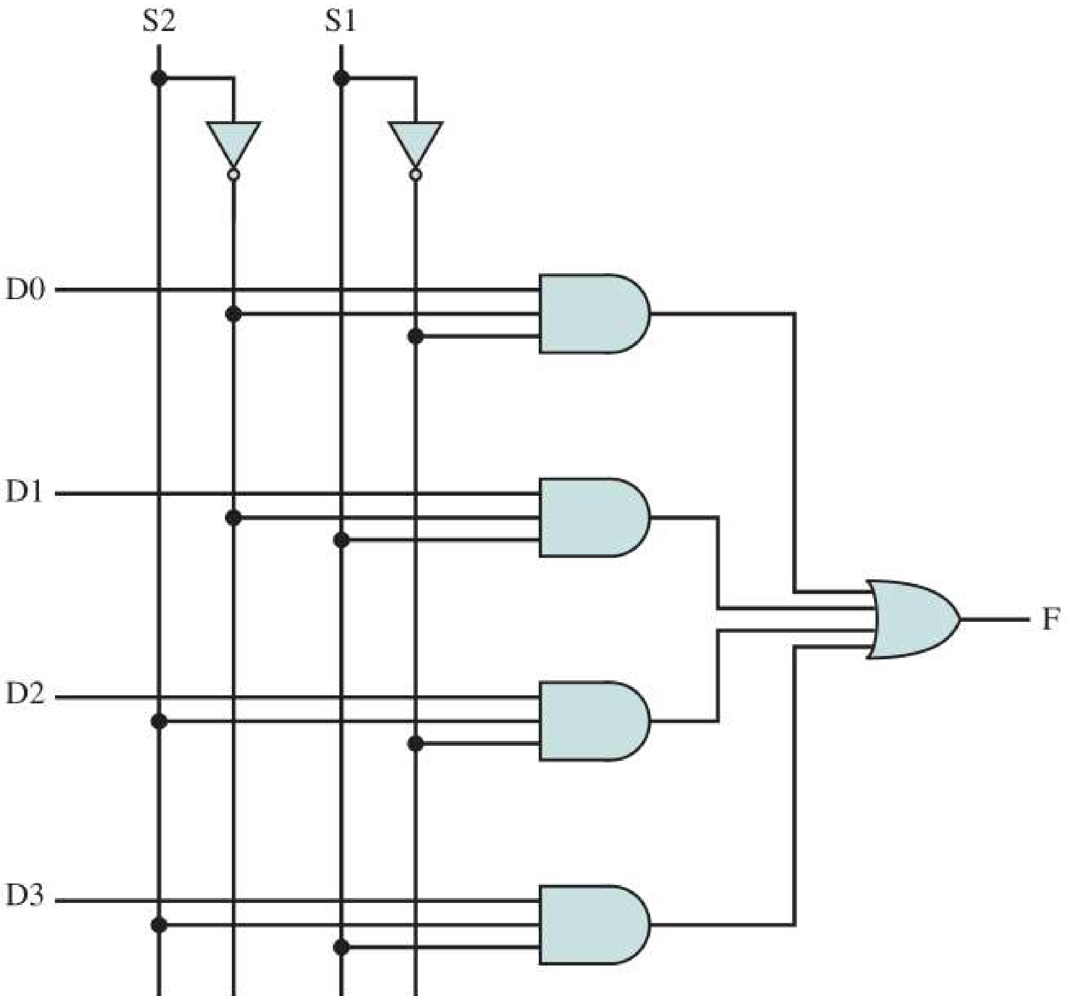


Figure 11.13 Multiplexer Implementation

Figura 3.16: Circuito de un multiplexor de 2 señales de control, y $2^2 = 4$ entradas

3.6.4. Demultiplexor

Un Demultiplexor sirve para, dada una entrada, y n señales de control, devolver la entrada por una de las 2^n salidas. Este se implementa usando un decodificador de n entradas y 2^n salidas, agregándole una entrada que va a ser la que salga por una de las 2^n salidas.

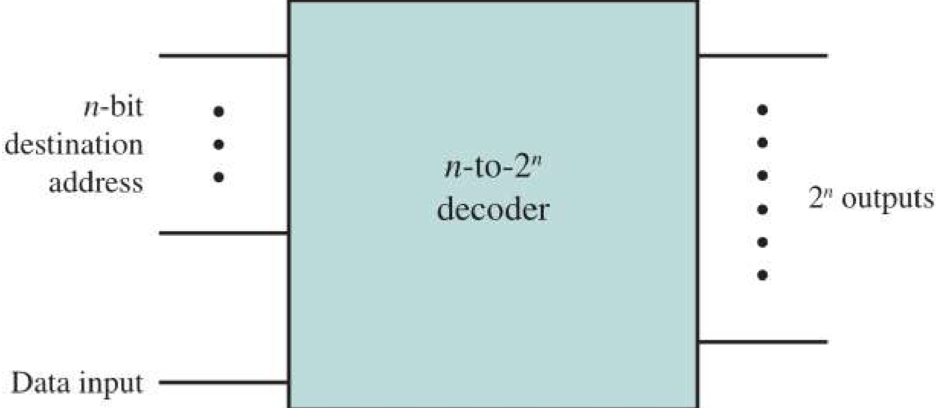


Figura 3.17: Circuito de un demultiplexor

3.6.5. Memoria ROM

Los circuitos combinatorios suelen llamarse "circuitos sin memoria" ya que sus salidas dependen únicamente de las entradas actuales, y no hay retención de entradas anteriores. Sin embargo, las memorias ROM (Read-Only Memory) son implementadas con un circuito combinatorio.

Recordemos que una ROM es una unidad de memoria que realiza únicamente lecturas. Esto implica que la información guardada en la ROM es permanente, y fue guardada durante la fabricación de la misma. Luego, dada una entrada (dirección) a la ROM, siempre devuelve el mismo resultado, por lo que es un circuito combinatorio.

Una ROM puede ser implementada con un Decodificador y un conjunto de compuertas OR:

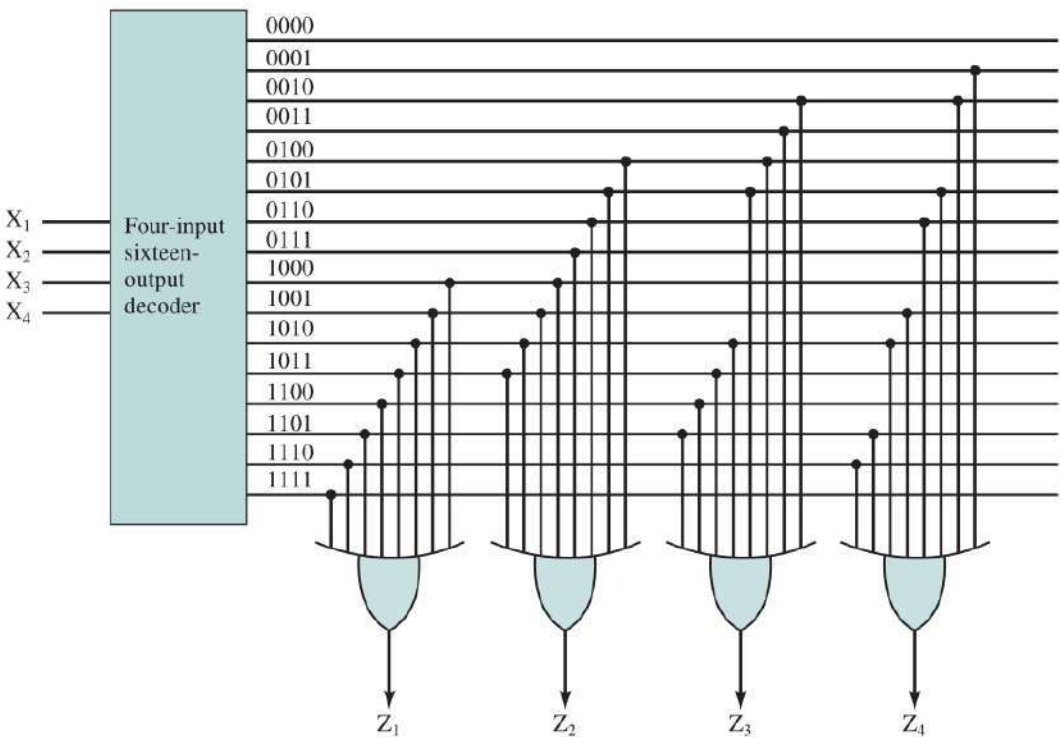


Figura 3.18: Circuito de una memoria ROM

3.6.6. Comparador

Otro circuito útil es el comparador, el cual compara dos entradas. El comparador simple toma dos entradas, A, B de tamaño 4-bits, y devuelve “1” sí son iguales y ‘0” sí son distintos.

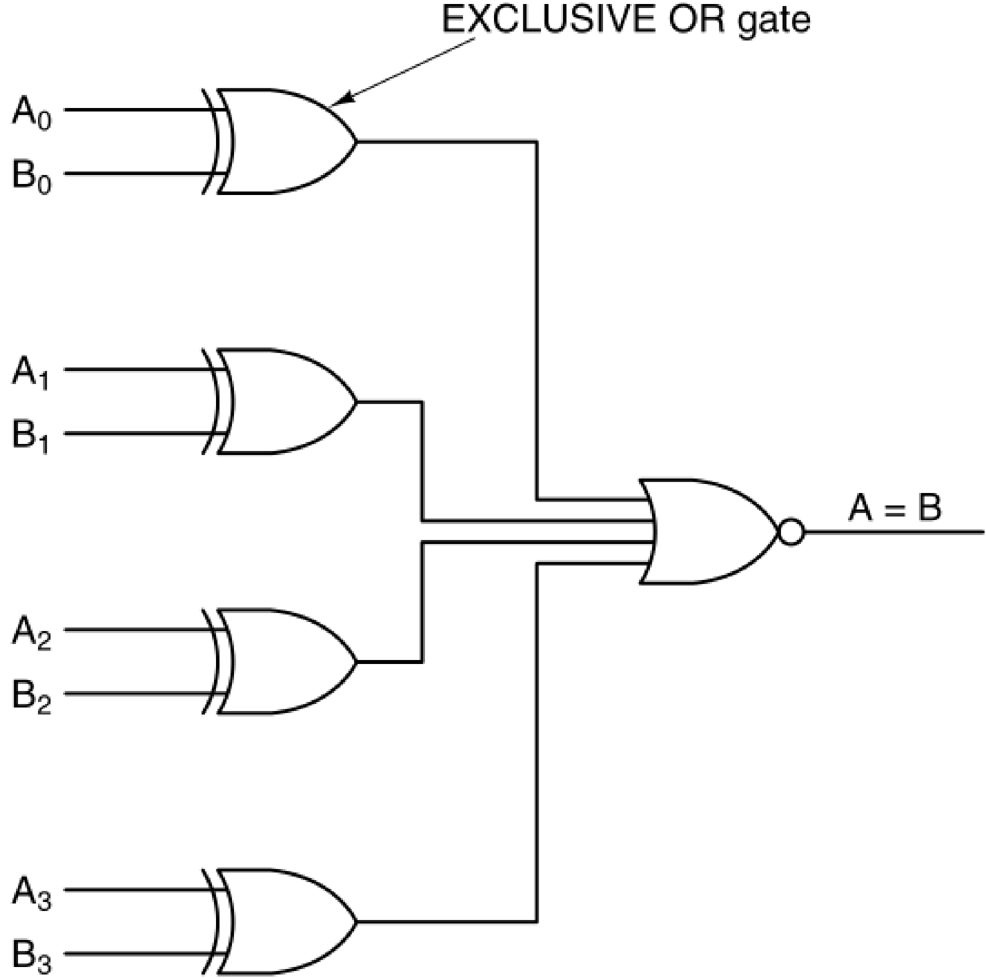


Figura 3.19: Circuito de un comparador de 4-bits

3.7. ALU

La mayoría de las computadoras tienen un único circuito para realizar AND, OR, y sumas de dos palabras. Típicamente, este tipo de circuitos para palabras de n-bits es construido a partir de n circuitos idénticos en cascada. Este tipo de circuito es conocido como ALU.

La idea es que este circuito permita computar cualquiera de estas cuatro funciones: A AND B, A OR B, NOT B, A + B, dependiendo de las líneas de control F0 , F1 .

3.7.1. ALU de 1-bit

Ahora estamos en condiciones de implementar una ALU de 1 bit. Queremos que esta ALU tenga:

- 3 entradas: A, B, Carry In
- 4 operaciones: AND, OR, NOT, Suma(A, B, Carry)
- 2 Salidas: Resultado, Carry Out

Además, para poder elegir entre las 4 operaciones, necesitamos dos señales extra de control: F1 y F2. Para poder, efectivamente, devolver el resultado correcto, usamos un multiplexor (usando como señales de control a F1 y F0).

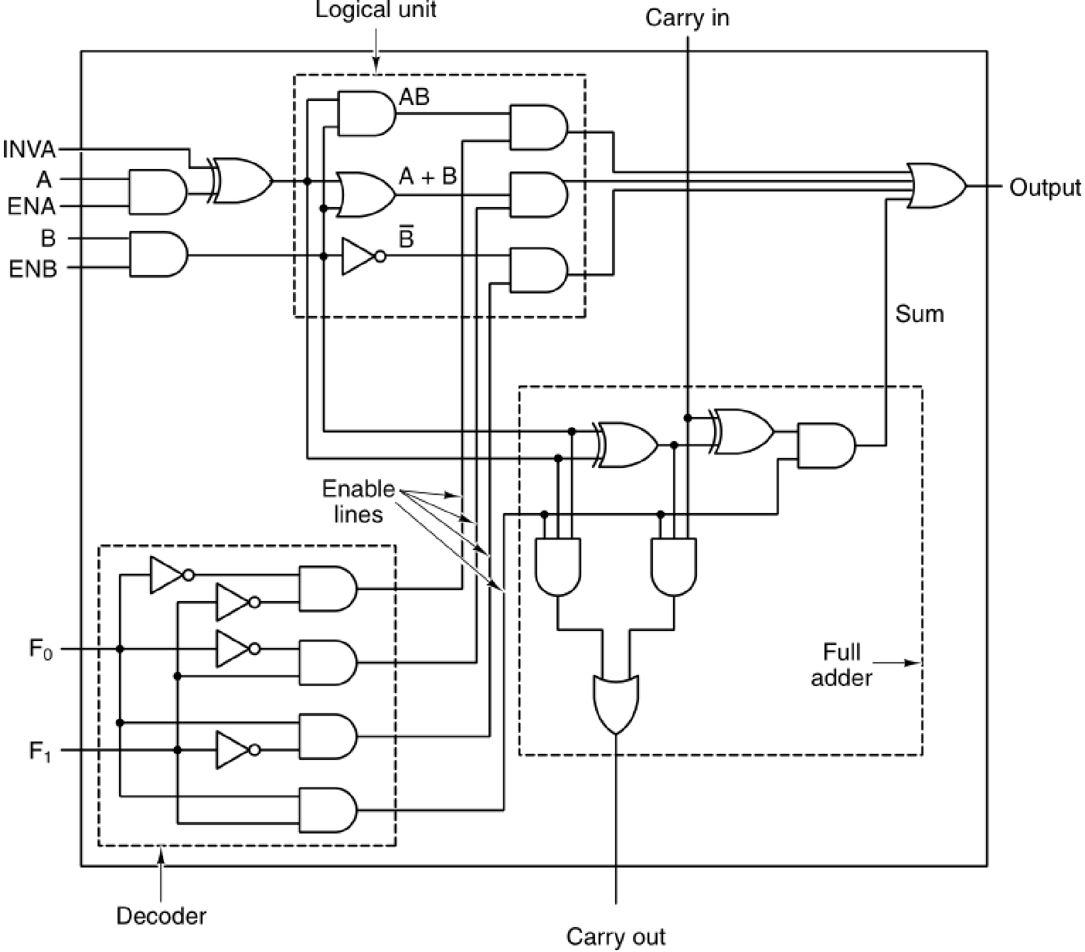


Figure 3-18. A 1-bit ALU.

Figura 3.20: Circuito de una ALU de 1-bit

Líneas de control extras:

- INVA: Sirve para negar A. (en condiciones normales vale 0)
- ENA: Sirve para forzar un 0 en A. (en condiciones normales vale 1)
- ENB: Sirve para forzar un 0 en B. (en condiciones normales vale 1)

3.7.2. ALU de 8-bits

Este diseño permite hacer ALU de n-bits en cascada. En la imagen podemos ver un ejemplo de una ALU de 8-bits. La señal de INC permite hacer cuentas cómo $A + 1$, $A + B + 1$.

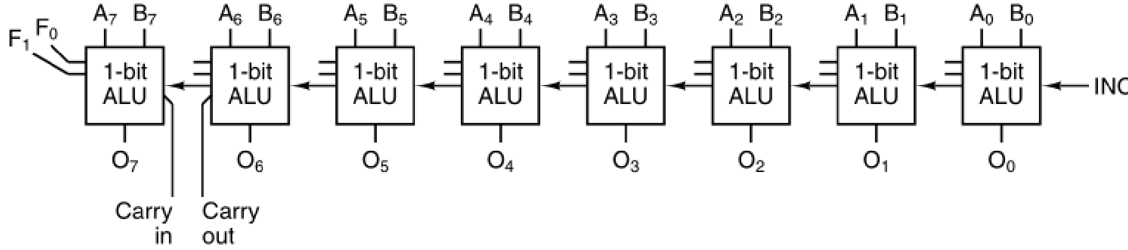


Figure 3-19. Eight 1-bit ALU slices connected to make an 8-bit ALU. The enables and invert signals are not shown for simplicity.

Figura 3.21: Circuito de una ALU de 8-bits

Capítulo 4

Lógica Digital: Parte 2

4.1. Introducción: Circuitos Secuenciales

Recordemos que para poder computar, bajo el modelo de cómputo Von Neumann - Turing, debemos poder cumplir con las siguientes normas:

- Los programas y datos se **almacenan** en una memoria.
- La operación de la maquina depende del **estado** de la maquina.
- El contenido de la memoria es **accedido** a partir de su posición.
- La ejecución es **secuencial** (salvo que se indique lo contrario).

Los circuitos combinatorios implementan las funciones esenciales de una computadora digital. Sin embargo, incluso en el caso especial de las ROM, este tipo de circuitos no permiten implementar **memorias** con un mecanismo para **almacenar** y **acceder** a valores, esenciales para poder computar.

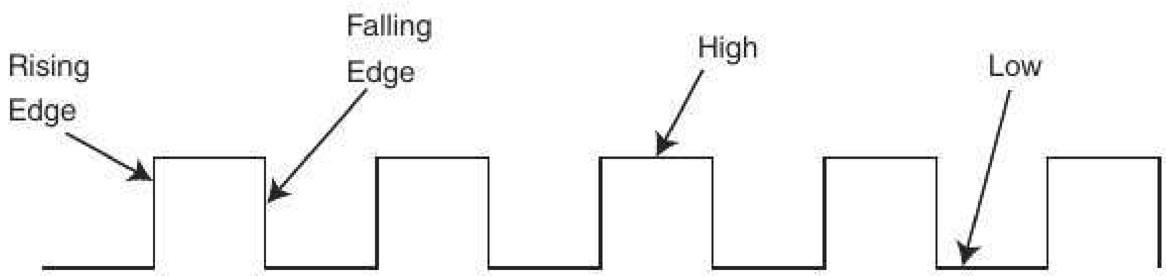
Hasta ahora, construimos circuitos que evalúan funciones booleanas, pero sus resultados no son perdurables en el tiempo. En el momento que cambie la señal va a cambiar la evaluación de la función. Es decir, no hay una noción de **estado**.

Entonces, para poder resolver estos problemas se utilizan circuitos más complejos: los circuitos secuenciales. El valor de salida de un **circuito secuencial** depende, no sólo de los valores actuales de entrada, sino también de los anteriores (es decir, el estado actual del circuito). En esta sección, examinaremos algunos sencillos, pero útiles, circuitos secuenciales.

4.2. Clock

En general, necesitamos una forma de ordenar los diferentes eventos que producen cambios de estados. Para esto usamos relojes: un reloj (clock) es un circuito capaz de producir señales eléctricas oscilantes con una frecuencia uniforme, es decir pulsos. Cada intervalo de tiempo entre dos pulsos consecutivos se conoce como ciclo de tiempo del reloj (clock cycle time).

La frecuencia de los relojes (pulsos por segundo) suele ser de entre 100 MHz y 4 GHz, que se corresponde con ciclos de reloj de 10 nanosegundos a 250 pico-segundos. Para mayor precisión, la frecuencia se suele controlar por un oscilador de cristal (comúnmente un cristal de cuarzo).



Las componentes vienen diseñados para detectar un tipo de señal de clock que permiten determinar cómo se mueve de un estado (t) al siguiente (t+1), y así poder sincronizar el circuito. Cuál usar es algo que se elige durante el diseño. En general, hay dos maneras de medir cambios de fase de las señales:

- **Cambio de flanco**: se detecta cuando hay un cambio en la señal, puede ser flanco ascendente (rising edge) o descendente (falling edge).
- **Nivel**: se verifica que la señal alcance cierto nivel, puede ser alto (high, es decir, un 1) o bajo (low, es decir, un 0).

4.3. Flip-Flop

Para retener sus valores, los circuitos secuenciales recurren a la **realimentación** (feedback).La realimentación se produce cuando una salida se conecta a una entrada. El circuito secuencial más sencillo es el flip-flop. Hay una gran cantidad de flip-flops, los cuales comparten las siguientes propiedades:

- Un flip-flop es un dispositivo **bi-estable**. Existe en uno de dos estados, y en la ausencia de entradas, se mantiene en dicho estado. Por lo que un flip-flop nos permite implementar una memoria de 1-bit.
- Un flip-flop tiene dos salidas, una es siempre el complemento de la otra. Estas se suelen llamar Q y $\overline{Q}$.

Un flip-flop es *edge-triggered*, y por lo tanto el estado de transición ocurre, no cuando la señal que emite el clock vale 1 (que es el caso de los *latch*), es decir no es *level triggered*, sino que durante la transición de 0 a 1 (*rising edge*) o de 1 a 0 (*falling edge*).

Luego, la longitud del pulso del clock es irrelevante, siempre y cuando la transición ocurra rápido. Muchos autores usan “flip-flop” cuando en realidad se refieren a *latch* (es el caso de esta materia). A modo de curiosidad, una manera sencilla de construir un generador de pulsos es la siguiente:

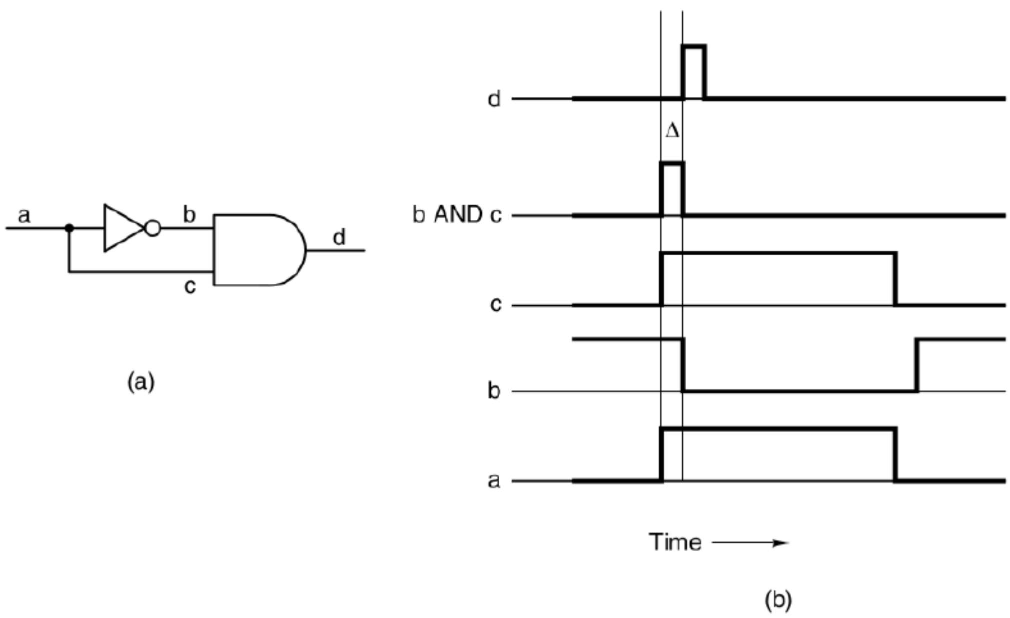


Figure 3-24. (a) A pulse generator. (b) Timing at four points in the circuit.

4.3.1. Flip-Flop: S - R

El S-R flip-flop es un circuito que tiene dos entradas, S (Set, coloca Q = 1) y R (Reset, coloca Q = 0), y dos salidas Q y $\overline{Q}$, donde la salida Q va a ser nuestro valor "memoria".

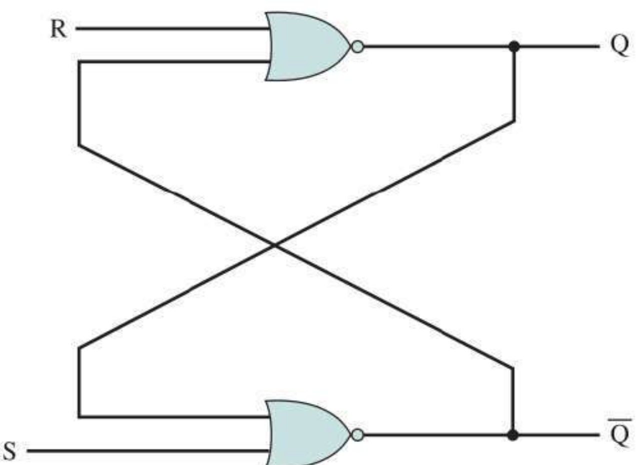


Figure 11.22 The S–R Latch Implemented with NOR Gates

Notar que la combinación S = 1, R = 1 NO está permitida, ya que al evaluarlo siempre obtenemos Q = 0, $\overline{Q}$ = 0, lo cual es un estado **inestable** e **inconsistente**. Este circuito puede ser definido con una tabla similar a la tabla de verdad, llamada *tabla característica*, que muestra el siguiente estado de un circuito secuencial, en función del estado actual y de sus entradas.

(b) Simplified Characteristic Table		
S	**R**	**Q_{n+1}**
0	0	Q_n
0	1	0
1	0	1
1	1	—

Para entender el funcionamiento del mismo, veamos el siguiente diagrama temporal que nos muestra un flip-flop S-R con entradas S = 1, R = 0, y su estado es Q_n = 0, $\overline{Q_n}$ = 1:

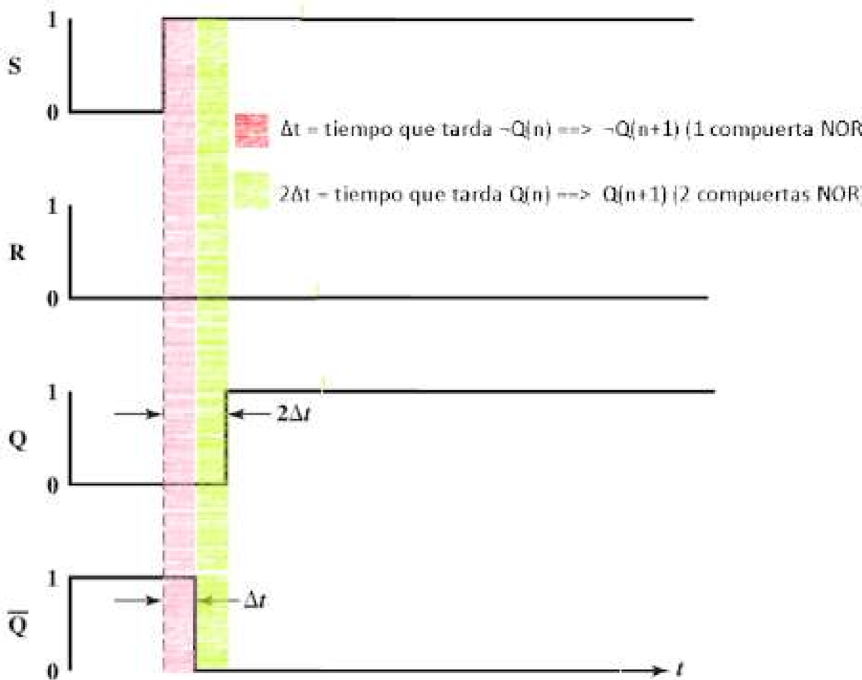


Figure 11.23 NOR S–R Latch Timing Diagram

Notemos que los valores de Q cambian en función del tiempo de la siguiente manera:

Durante el primer Δt estamos en un estado **inestable**:

$$Q_{n+\frac{1}{2}} \leftarrow \neg Q_n \,\textbf{NOR}\, R = 1 \,\textbf{NOR}\, 0 = 0$$

$$\neg Q_{n+1} \leftarrow Q_n \,\textbf{NOR}\, S = 0 \,\textbf{NOR}\, 1 = 0$$

Luego de 2Δt volvemos a un estado **estable**:

$$Q_{n+1} \leftarrow \neg Q_{n+1}\,\textbf{NOR}\,R = 0\,\textbf{NOR}\,0 = 1$$
$$\neg Q_{n+1} \leftarrow Q_{n+\frac{1}{2}}\,\textbf{NOR}\,S = 0\,\textbf{NOR}\,1 = 0$$

Esto quiere decir que nuestro flip-flop se vuelve inestable durante un tiempo Δt, y vuelve a serlo después de un tiempo 2Δt.

4.3.2. S-R Flip-Flop Sincronico

Cuando se diseña un circuito digital, se debe considerar el comportamiento físico de los circuitos electrónicos. Es decir, existen retardos de propagación, que pueden hacer que el comportamiento del sistema difiera del esperado inicialmente, como vimos en el caso anterior (por ejemplo, si cambiamos las señales de entrada mientras el estado sea inestable).

Entonces, es conveniente poder asegurar el momento en el que el flip-flop, efectivamente, cambie de estado. Es decir, queremos evitar que las señales de salida cambien tan rápido como las señales de entrada, y con esto se busca **sincronizar** el circuito.

Si no sincronizamos la memoria, no tendríamos certeza sobre cuando un valor que se colocó en la entrada del circuito está, efectivamente, almacenado, y cuando no. Por ejemplo, si se coloca un valor en la entrada, y lo cambio instantáneamente, ¿se llegaron a propagar las señales?. Si no se esperó a que se propaguen estas señales, la siguiente entrada a evaluar no estaría tomando en cuenta el hecho de que el circuito se encuentra en un estado intermedio inestable.

Para esto debemos realizar una pequeña modificación, agregando un Clock que funcione como un enable, y así obtener un *Clocked S-R Flip-Flop*.

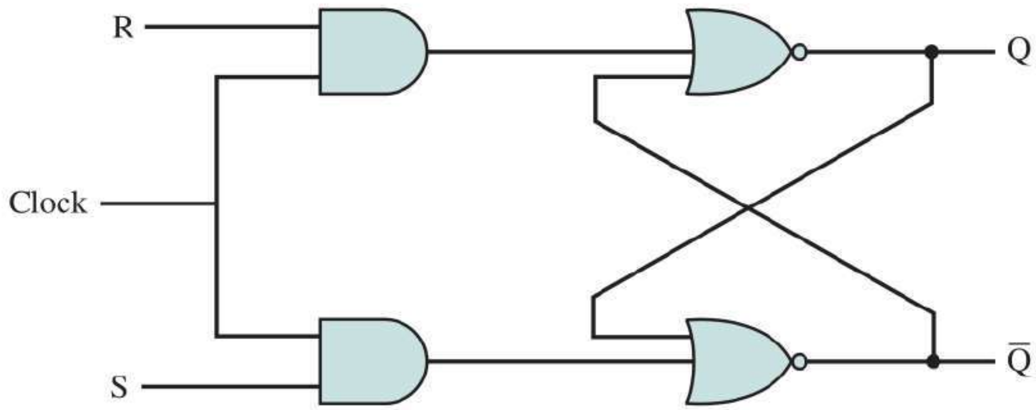


Figure 11.24 Clocked S–R Flip-Flop

Este circuito tiene una entrada adicional, el *clock*, que oscila entre 1 y 0. Cuando el *clock* vale 0, ambas compuertas **AND** devuelven 0, independientemente de los valores S y R, y por lo tanto no hay cambio de estado. Cuando el *clock* vale 1, el efecto del **AND** desaparece, y el flip-flop vuelve a ser dependiente de los valores de S y R.

4.3.3. Flip-Flop D

Un problema con el S-R flip-flop es que la entrada R = 1, S = 1 debe ser evitada, y hay una buena razón para evitarlo. El circuito se vuelve no determinístico cuando R y S vuelve a valer 0. El único estado consistente para S = R = 1 es Q = $\overline{Q}$ = 0, pero tan pronto como las entradas vuelvan a 0, el flip-flop

debe saltar a uno de sus dos estados estables. Si cualquiera de las dos entradas se vuelve a 0 antes que la otra, aquella que haya permanecido por más tiempo en 1 gana, porque una vez la entrada es 1, fuerza el estado.

Una manera de solucionar esto, es tener una única entrada "D". Esta modificación es conocida como D flip-flop (data flip-flop). El cual, su función es guardar en Q el último valor pasado por D.

D	Q_{n+1}
0	0
1	1

Tabla Característica Flip-Flop D

Podemos ver la modificación en la siguiente imágen:

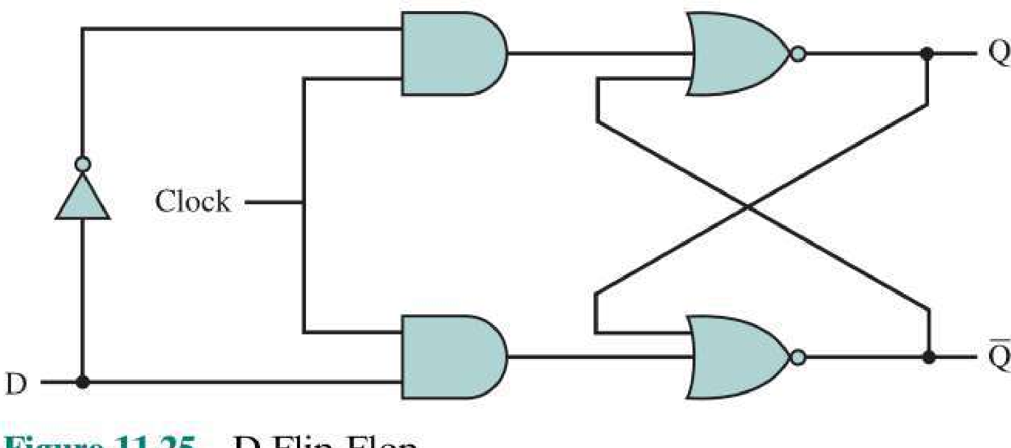


Figure 11.25 D Flip-Flop

Como podemos ver, cuando el Clock esté **bajo**, el Flip-Flop D estará en **modo lectura**, y cuando el Clock esté **alto**, en **modo escritura** (nota: el clock puede estar en un modo **oscilante**, y también se puede poner en un modo que emita una señal plana, lo cual permite fijar el modo de lectura/escritura de este flip-flop).

4.3.4. Flip-Flop: J-K

Otro flip-flop útil es el J-K flip-flop. Este, al igual que el S-R, tiene dos entradas, sin embargo todas las posibles combinaciones son entradas válidas. A continuación veamos una implementación del mismo:

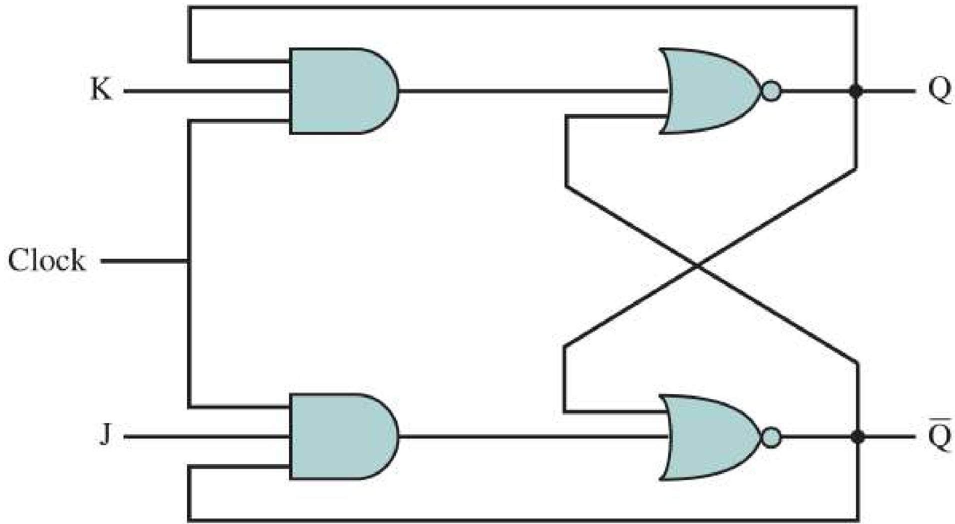


Figure 11.26 J–K Flip-Flop

Y a continuación su tabla característica:

J	K	Q_{n+1}
0	0	Q_n
0	1	0
1	0	1
1	1	$\overline{Q_n}$

Notemos que es idéntico al S-R, salvo por el caso J = 1, K = 1, que invierte el valor de Q.

4.4. Registros

Un elemento esencial de la CPU que hace uso de los flip-flops es el **registro**. Un registro es un circuito digital usado dentro de la CPU para guardar uno o más bits de datos. Dos tipos básicos de registros comúnmente usados son los *parallel registers*, y los *shift registers*.

4.4.1. Parallel Registers

Un *parallel register* consiste en un conjunto de memorias de 1-bit que puede ser leídas o escritas simultáneamente. Es usado para guardar datos.

A continuación veamos una una implementación de un registro de 8-bits usando flip-flops D. La señal de control `load` controla la escritura del registro, funcionando como una línea de enable.

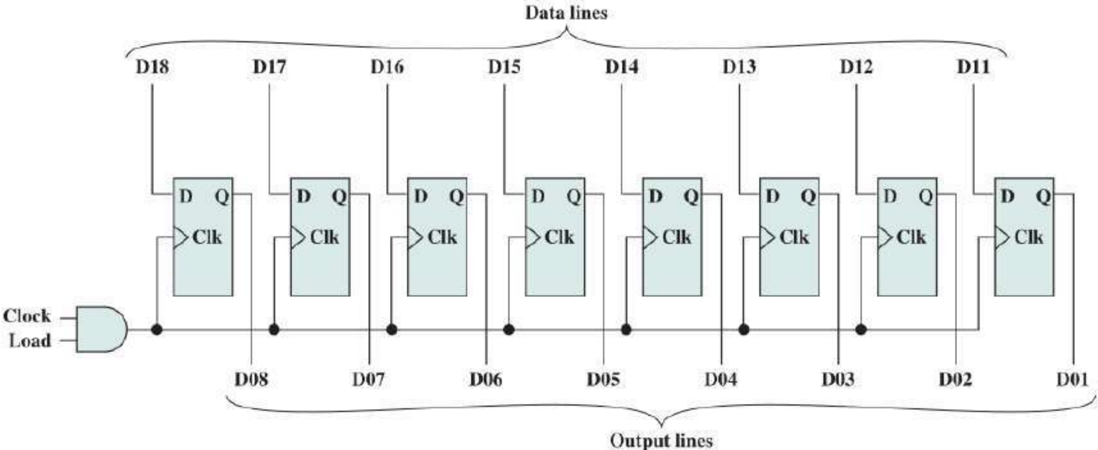


Figure 11.28 8-Bit Parallel Register

Una vez hemos diseñado un registro de 8-bits, podemos usarlo como bloque de construcción para crear registros más grandes. Por ejemplo, un registro de 32-bits podría ser creado a partir de combinar dos registros de 16-bits, atando la señal del clock.

4.4.2. Shift Register

Un *shift register* recibe y/o transfiere información en serie. A continuación veamos una implementación de un *shift register* de 5-bits, construido a base de clocked D flip-flops. Los datos son introducidos únicamente a través del flip-flop de más a la izquierda. Con cada pulso de reloj, los datos son *shifteados* una posición a la derecha, y el dato del flip-flop derecho es transferido como salida.

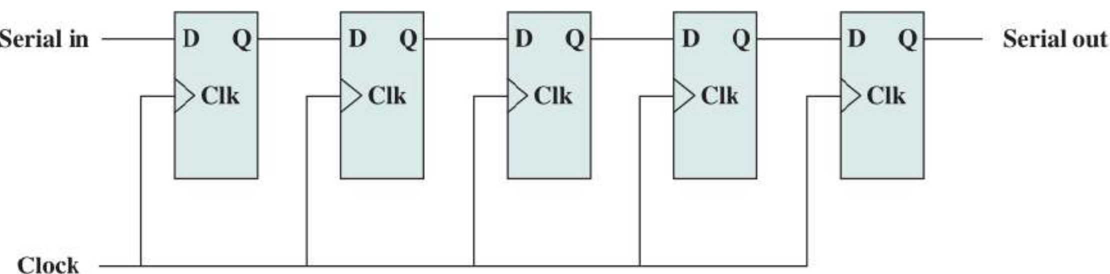
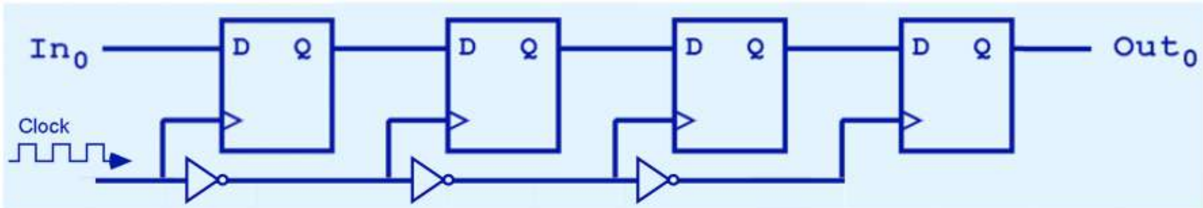


Figure 11.29 5-Bit Shift Register

Los *shift registers* pueden ser utilizados como interfaz para dispositivos de E/S seriales. Además, pueden ser utilizados dentro de la ALU para realizar shifts lógicos y funciones de rotación (para esto último también es necesario que estén equipados con circuitos paralelos de lectura/escritura).

Si estuviésemos usando *latchs* en vez de flip-flops, entonces el tiempo de propagación podría ser menor que el tiempo en el que la señal del clock esté alta, permitiendo que un valor se propague varias posiciones en un solo pulso del clock, lo cual no es el resultado esperado. Una forma de resolver este problema es **intercalar** las señales del clock:



(*) Nota: Este circuito es el que dieron en la teórica. No permite intercalar 1s y 0s (por ejemplo, no podemos escribir 1010).

4.5. Organización de Memorias

A pesar de que podemos usar el esquema anterior para memorias de 8-bits, para poder construir memorias mucho más grandes necesitamos de una organización diferente, una que permita *direccionar palabras* de forma individual.

Empecemos con una memoria de 4 palabras de 3 bits, que permita que cada operación lea o escriba palabras completas de 3-bits:

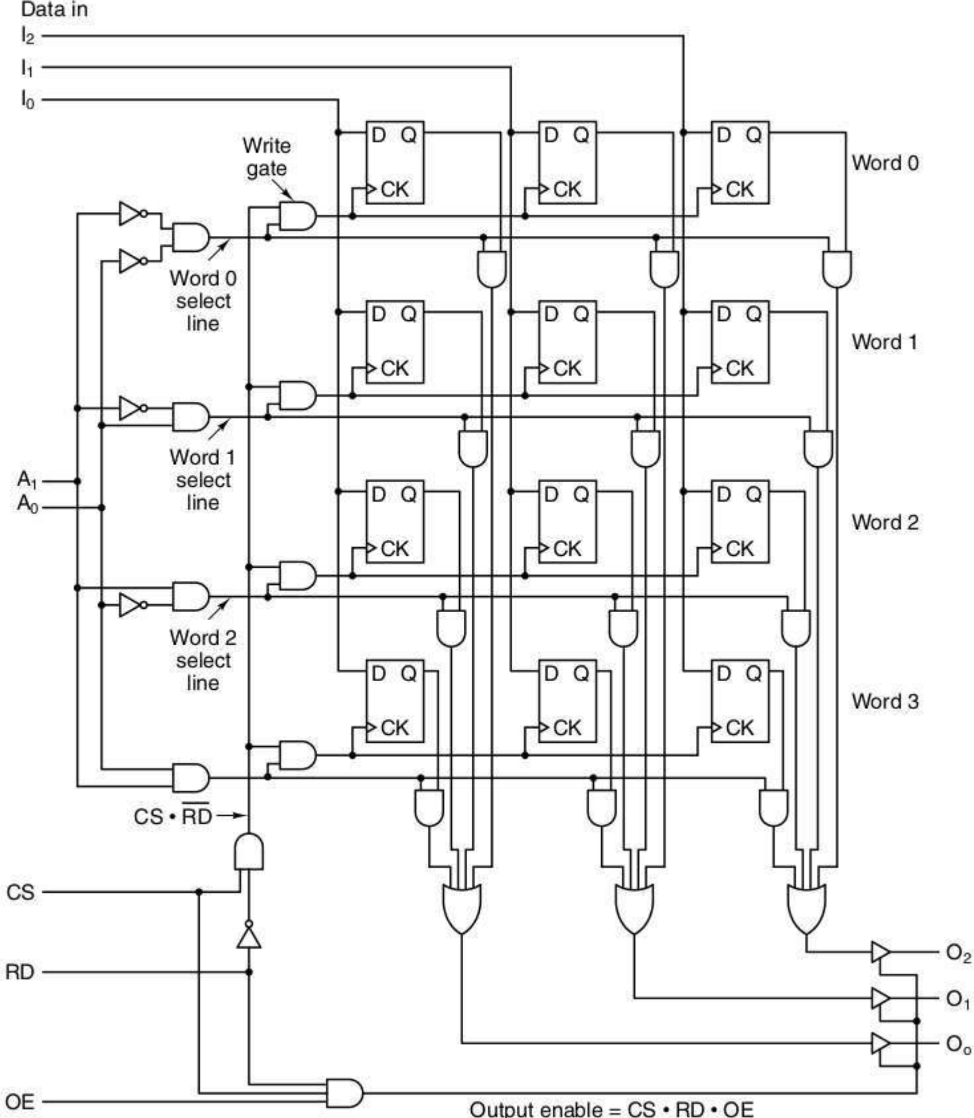


Figure 3-28. Logic diagram for a 4 × 3 memory. Each row is one of the four 3-bit words. A read or write operation always reads or writes a complete word.

Entradas:

- I_0, I_1, I_2 son entradas de datos para la escritura (no se usan en lecturas)
- A_0, A_1 son la dirección de la palabra a acceder
- CS (*Chip Select*), RD (sirve para elegir entre lectura (RD = 1) y escritura (RD = 0)), OE (*Output Enable*)

Salidas:

- O_0, O_1, O_2 son los valores de salida de una lectura (no se usan en escrituras)

Lo bueno de este esquema es que es fácilmente extensible a memorias de mayores tamaños. Si queremos tener una memoria de 4 palabras $\times$ 8 bits, simplemente agregamos 5 columnas, y si queremos una de 8 palabras $\times$ 3 bits, agregamos 4 filas.

Con este tipo de estructura, el número de palabras en la memoria debería ser una potencia de 2,

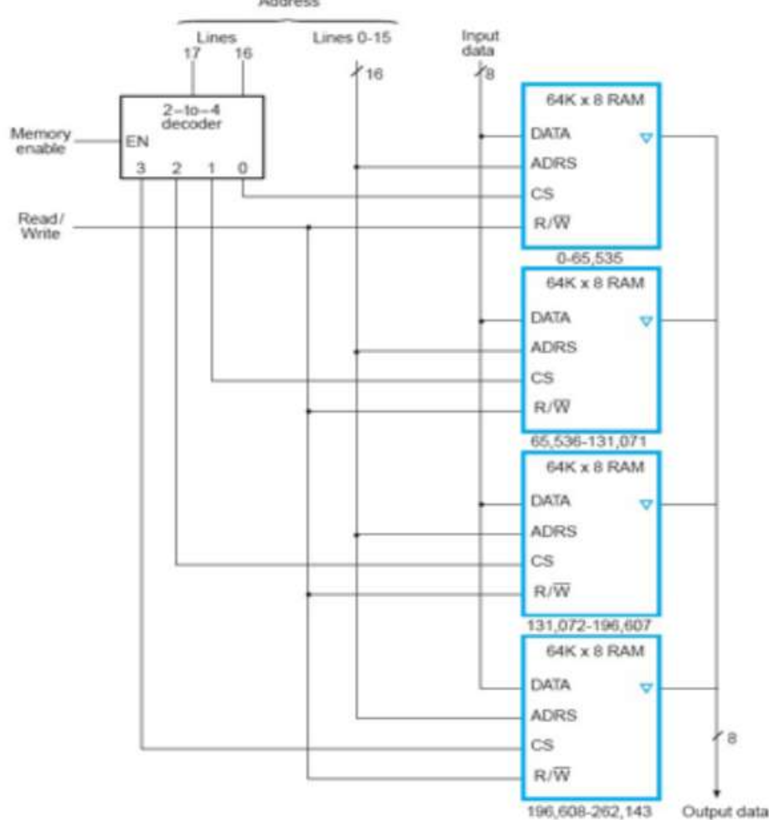
para una máxima eficiencia, pero el número de bits en una palabra puede ser cualquiera.

Una vez tenemos un bloque de memoria, podemos usarlos para construir bancos de memoria más grandes, ya sea con respecto a la cantidad de palabras, o a la longitud de las mismas:

Aumentando la cantidad de palabras:

Una vez tenemos construido un bloque de memoria, podemos utilizarlo para construirnos una memoria con una mayor cantidad de palabras de la siguiente manera:

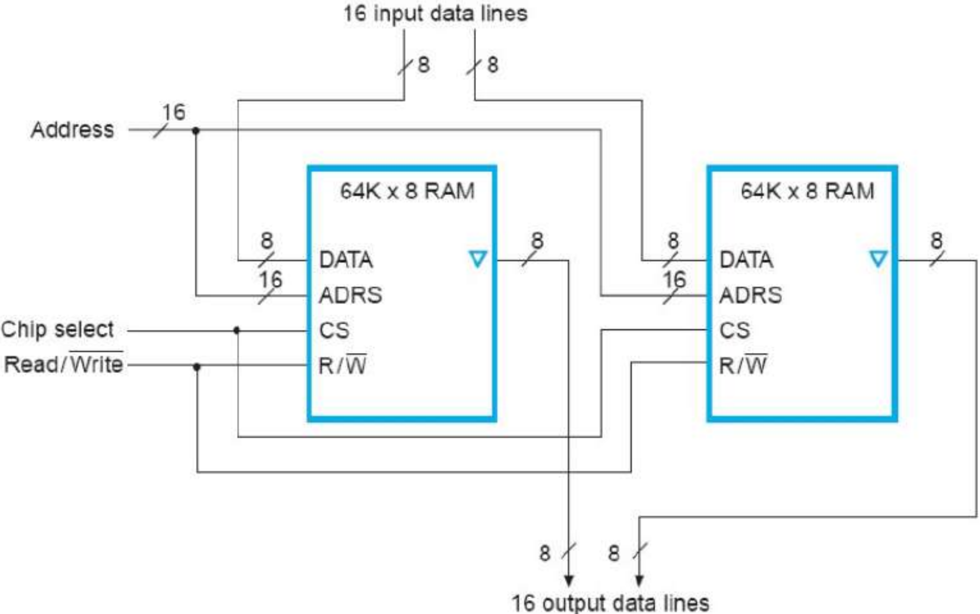
Memoria 256Kx8 usando memorias de 64Kx8



Aumentando el tamaño de palabra:

También podemos aumentar el tamaño de la palabra a partir de bloques de memoria de palabras más chicas:

Memoria 64Kx16 usando memorias de 64Kx8



Notemos que tenemos que mandarle las mismas señales de control a ambas memorias (la mitad de la palabra se encuentra en un chip, y la otra mitad se encuentra en el otro).

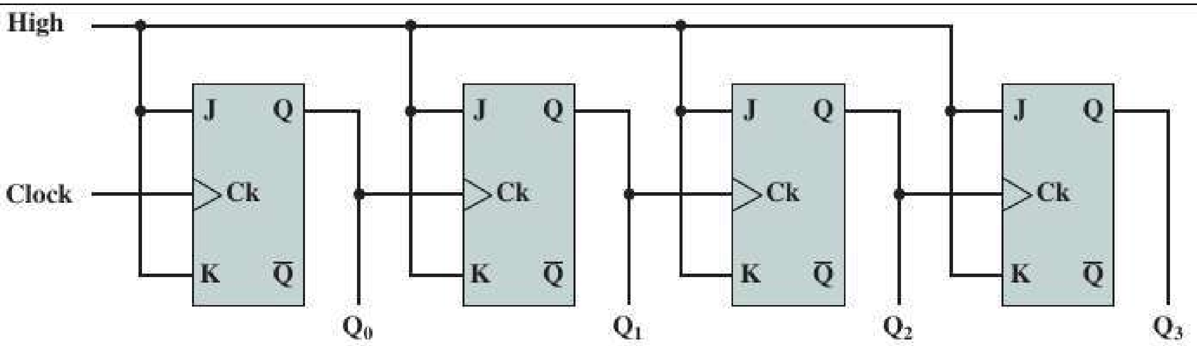
4.6. Contadores

Otro tipo de circuitos secuenciales de suma utilidad son los **contadores**. Estos son registros cuyo contenido se incrementa fácilmente por 1 módulo la capacidad del registro (es decir que una vez que se le suma uno al máximo valor del registro, su valor pasa a ser nuevamente 0). Un registro hecho con n *flip-flops* puede contar hasta $2^n - 1$. Un ejemplo de un *counter register* en la CPU es el program counter.

Hay dos tipos de contadores que son los **sincrónicos** y los **asincrónicos**. En los **asincrónicos** cada flip-flop se modifica a partir de la modificación del flip-flop anterior, haciéndolo un circuito lento. En cambio, en los **sincrónicos** todos los flip-flops cambian de estado al mismo tiempo. Como estos últimos son mucho más rápidos, son lo que se terminan usando en las CPUs. Sin embargo, es útil empezar la discusión viendo los **asincrónicos**.

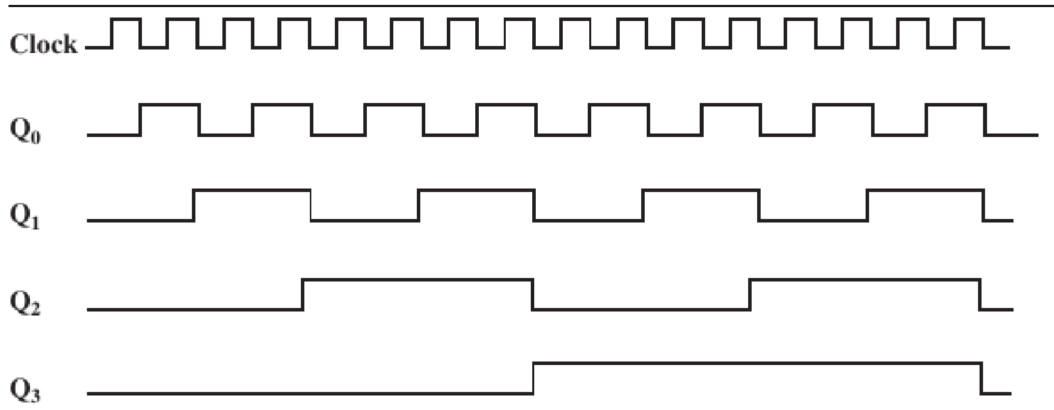
4.6.1. Contadores asincrónicos

Como mencionamos anteriormente, en los contadores asincrónicos el cambio de estado de un flip-flop se da a consecuencia del cambio en anterior. A continuación veamos una implementación de un contador de 4-bits usando J-K flip-flops.



La salida Q_0 representa el bit menos significativo y **High** (siempre vale 1) es la señal de entrada a partir de la cual se incrementa el contador. Cabe aclarar que estos son flip-flops J-K *falling-edge triggered* (el cambio de estado se habilita en el flanco descendiente de la señal de clock). Usar flip-flops que responden a la transición en un pulso de clock, en vez de el pulso en sí, provee una mejor sincronización (lo cual es necesario para circuitos más complejos).

Como **High** es la entrada para todos los flip-flops J-K, cada vez que ocurra un pulso, el valor de Q_i pasa a ser $\neg Q_i$ (dado que si J = K = 1 en un J-K flip-flop, se invierte el valor del flip-flop). A continuación veamos un diagrama temporal que ilustre el comportamiento del circuito:



4.6.2. Contadores sincrónicos

Estos contadores, a diferencia de los asincrónicos, modifican el estado de todos los flip-flops al mismo tiempo. En esta subsección vamos a presentar un diseño para un contador sincrónico de 3-bits. En el proceso vamos a mostrar algunos conceptos básicos a la hora de diseñar circuitos sincrónicos.

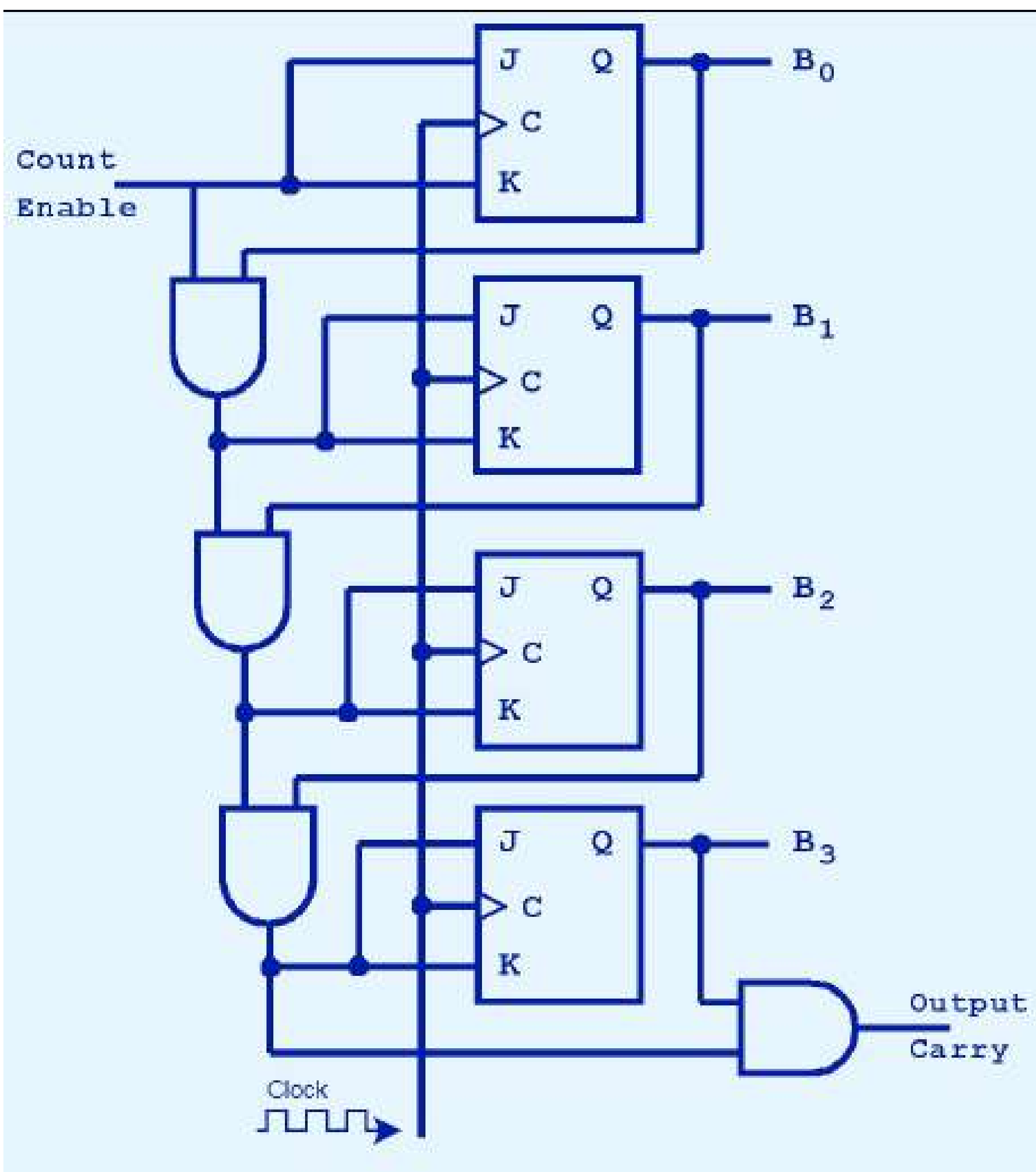
Para un contador de 3-bits, necesitaremos 3 flip-flops. Vamos a usar J-K flip-flops. Llamaremos a la salida (Q) de los mismos A, B, C.El primer paso es construir una tabla de verdad de las entradas y salidas de los flip-flops, para poder así diseñar el circuito.

(a) Truth table

C	B	A	Jc	Kc	Jb	Kb	Ja	Ka
0	0	0	0	d	0	d	1	d
0	0	1	0	d	1	d	d	1
0	1	0	0	d	d	0	1	d
0	1	1	1	d	d	1	d	1
1	0	0	d	0	0	d	1	d
1	0	1	d	0	1	d	d	1
1	1	0	d	0	d	0	1	d
1	1	1	d	1	d	1	d	1

Las primeras tres columnas muestran las posibles combinaciones de las salidas A,B,C. Están listadas en el orden que aparecerán a medida que el contador sea incrementado. Cada fila lista los valores actuales de A,B,C y las entradas que necesitarán para obtener el resultado buscado.

Ahora veamos una posible implementación de este circuito:



A diferencia de los contadores asincrónicos, vemos que en el circuito de arriba todas las entradas de clock están conectadas al clock en paralelo, por lo que la modificación de los flip-flops se da de manera simultánea. En un ciclo de clock, el cambio de estado para B_0 se da cuando *count enable* $= 1$ (con lo que se invierte su valor).

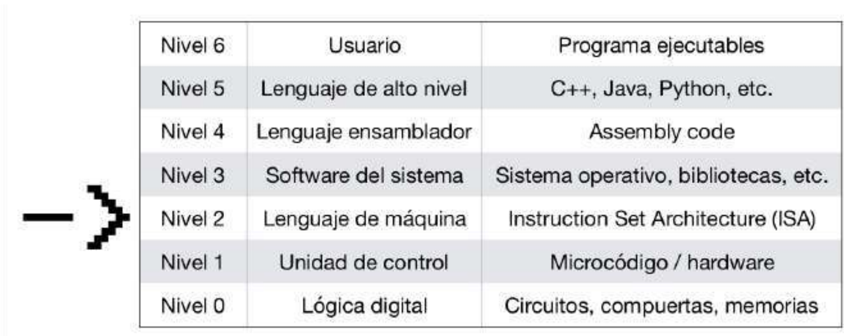
Para el resto de los flip-flops, como tienen su entrada J y K conectadas a la salida de una compuerta "and" que tiene por entradas la salida Q del flip-flop anterior y la salida de la compuerta "and" anterior (en el caso de B_1, de *enable count*), si el estado del flip-flop anterior era 1 y se invirtió, entonces se invierte también el valor de flip-flop siguiente. La última compuerta "and" del circuito devuelve 1 cuando el contador vuelve a 0.

Capítulo 5

Instruction Set Architecture (ISA) - Parte 1

5.1. Introducción

La ISA es la percepción que tiene un programador (de bajo nivel) de una computadora. En este contexto una instrucción es una tira de bits, no hay un lenguaje de programación que se compila, que se linkea, y luego se ejecuta a través del sistema operativo. Estamos montados directamente sobre el procesador. Tenemos las componentes electrónicas, y vamos a tener almacenados en la memoria tiras de bits, para que cuando estas componentes lean estas tiras de bits, las interpreten como instrucciones, que estas van a ser capaces de ejecutar.



Nivel		
Nivel 6	Usuario	Programa ejecutables
Nivel 5	Lenguaje de alto nivel	C++, Java, Python, etc.
Nivel 4	Lenguaje ensamblador	Assembly code
Nivel 3	Software del sistema	Sistema operativo, bibliotecas, etc.
Nivel 2	Lenguaje de máquina	Instruction Set Architecture (ISA)
Nivel 1	Unidad de control	Microcódigo / hardware
Nivel 0	Lógica digital	Circuitos, compuertas, memorias

En la jerarquía de las capas de abstracción de la computadora, la ISA se encuentra a nivel 2. Ese nivel 2 me separa la organización de la computadora, y muestra hacia afuera, hacia los programadores, la arquitectura de la computadora: el conjunto de instrucciones y recursos a través de los cuales el programador puede programar la maquina. Dejamos de ver las componentes, y empezamos a ver la vista de quien las programa, sin tener acceso a ellas. Vamos a definir esta linea divisoria, vamos a ver hacia abajo las componentes, y hacia arriba vamos a mostrar lo que el programador vea.

Recordemos que, dentro del modelo de computo Von Neumann - Turing:

- Los programas y los datos se almacenan en la misma memoria sobre la que se puede leer y escribir
- La operación de la máquina depende del estado de la memoria
- El contenido de la memoria es accedido a partir de su posición
- La ejecución es secuencial (a menos que se indique lo contrario)

Queremos tener una máquina que ejecute secuencialmente instrucciones, y que esa ejecución pro-

duzca, a partir de datos que se encuentran en la memoria, un resultado, que también se guarda en la memoria.

Partimos de que existen datos y programas que se almacenan en una memoria, y la maquina va a interpretar la cadena de bits que recupera de la memoria como una instrucción, ejecutarla, donde esa ejecución implica traer datos de la memoria, operarlos de alguna manera a través de la ALU, y volver a colocarlos en la memoria.

En la arquitectura clásica de Von Neumann tiene 3 componentes centrales:

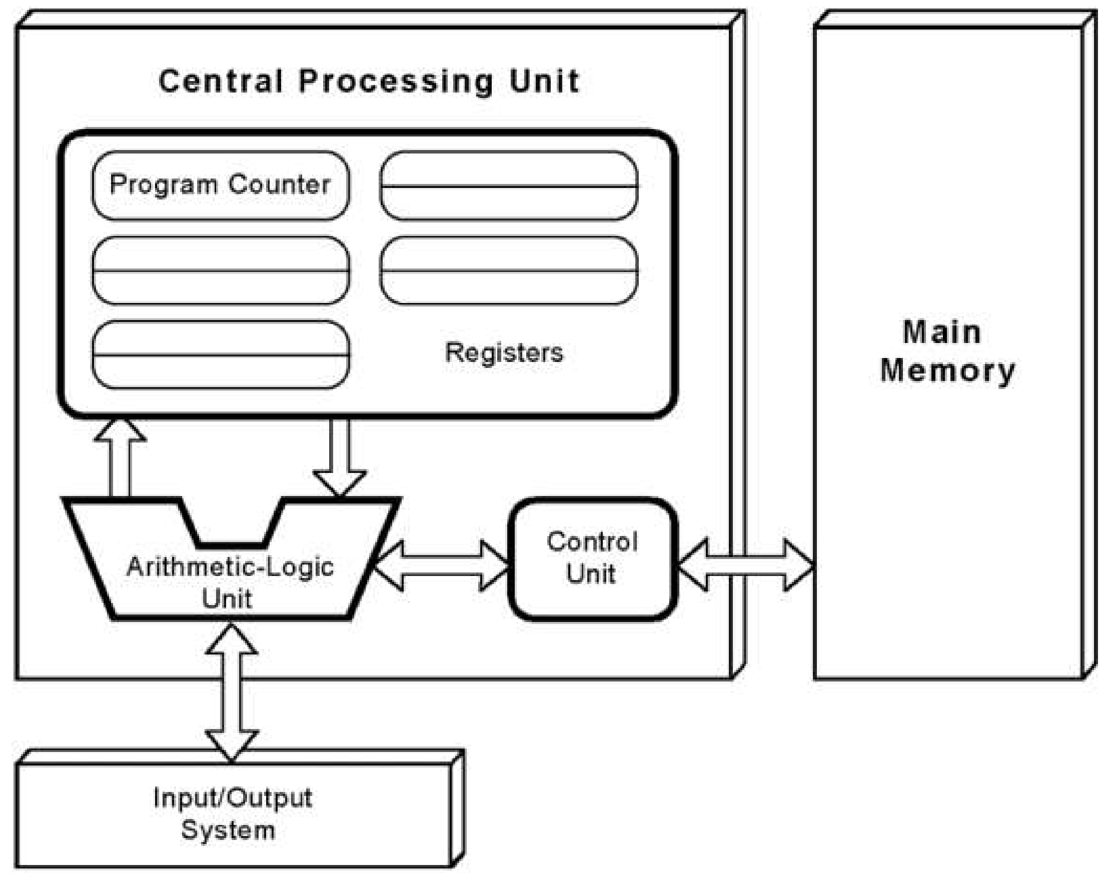
- CPU
- Memoria
- Sistema de E/S: me permite mover datos entre el mundo exterior y la máquina

Ejecuta de manera secuencial los programas, es decir lee una instrucción tras otra. Los datos y los programas están almacenados en binario, y tiene un sistema de interconexión de componentes, que se asume que existe, que cuando veamos buses vamos a ver como se manipula explícitamente.

Este Sistema de interconexión de componentes se encarga de:

- Conecta la Unidad de Control con la Memoria mediante un camino único
- La unicidad del camino fuerza la alternación entre ciclos de lectura / escritura y ejecución
- Esta alternación se llama cuello de botella de von Newmann (von Newmann bottleneck)

Los vamos a mantener como un elemento que vive en el nivel 0, desde el punto de vista de la ISA es totalmente transparente: la utilización de esas componentes electrónicas no van a ser visibles para alguien que utilice la ISA.



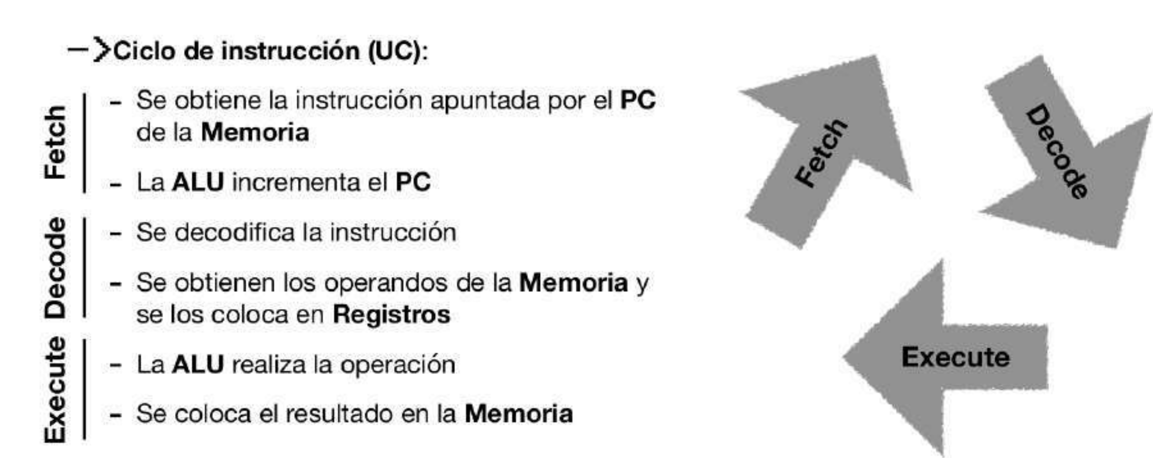
Lo que se ve ahí es la fisonomía que tiene la maquina. Tengo una CPU, con una ALU que tiene acceso a pequeñas porciones de memoria que existen dentro del procesador llamadas registros, tengo una unidad de control que es quien va a ejecutar cíclicamente los programas, tengo un mecanismo de E/S y tengo una memoria donde esta guardada toda la información (programas y datos).

5.2. Ciclo de instrucción

La ejecución de un programa es una ejecución cíclica. Hay un montón de cosas que hay que tener en cuenta cuando pensamos una computadora desde este punto de vista, ya no son solo componentes electrónicas, sino que se empieza a complejizar.

Para poder implementar esta lógica de ejecución cíclica de instrucciones, a bajo nivel, tengo que tomar decisiones que provocaran ciertas regularidades en las acciones. Esta idea de que traigo una instrucción de la memoria al procesador para poder ejecutarla, implica qué tengo que poder determinar cuantos bits tiene la instrucción, que van a tener una codificación especifica que me va a servir para saber que tengo que hacer con la información que acabo de traer. Esta ejecución cíclica se la conoce como ciclo de instrucción.

El ciclo de instrucción tiene, dependiendo del procesador, diferente naturaleza, pero en un esquema básico lo que tenemos es la idea de que tenemos una etapa de **Fetch** (busco la instrucción en la memoria, y actualizo el PC para que apunte a la siguiente instrucción a ejecutar). Una vez me llega la instrucción, la tengo que decodificar, y eventualmente, buscar un operando adicional en la memoria, en la etapa de **Decode**. Por ultimo, una etapa de **Execute**, en donde los datos ya están disponibles para la ALU, por lo que se lleva a cabo la instrucción, y se coloca los resultados donde corresponda (un registro, memoria).



Notemos que, al finalizar el ciclo, estamos en situación para poder realizar este ciclo nuevamente. Una vez que entendimos cómo funciona este ciclo, nos falta poder codificar en las instrucciones que quiero exportar para ser utilizadas en mi procesador.

La ISA es la frontera entre el SW y el HW: por debajo de la ISA tengo componentes electrónicas, por encima tengo un lenguaje de programación para manipular esas componentes electrónicas, pero sin tener que visualizar como están implementadas.

5.3. Retrocompatibilidad

Por ejemplo, si uno agarra la arquitectura de Intel x86, y tomamos un programa escrito en base a la ISA del procesador Intel 8088 (el primero para una computadora personal), ese programa puede ser ejecutado en un procesador de la familia x86 actual (i9 Core x86). Esto se debe a que la ISA actual está

construida para soportar legacy code, por lo que si tomo código construido para una ISA antigua, lo puedo ejecutar en un procesador actual.

La razón de esto es que la arquitectura de Intel es compatible hacia atrás, a pesar de que la organización de estos procesadores modernos es completamente diferente. Es decir, garantizan que, a partir de la ISA que funciona como frontera que nos separa del HW, el procesador sigue aceptando las instrucciones de procesadores anteriores, permitiendo ejecutar código antiguo sobre una organización moderna.

5.4. Propiedades de una ISA

5.4.1. RISC vs CISC

Una de las decisiones a tomar al momento de diseñar una ISA es el set de instrucciones. Este puede ser muy reducido, propio de procesadores RISC (Reduced Instruction Set Computer), de manera que estas instrucciones sencillas se ejecuten rápido, pero programar a bajo nivel con un set de instrucciones tan reducido puede ser una tarea más engorrosa.

Un set de instrucciones complejo, propio de procesadores CISC (Complex Instruction Set Computer), provee una gran cantidad de instrucciones, dando una mayor flexibilidad al momento de programar a bajo nivel, pero tener instrucciones complejas hace que la lógica para ejecutarlas sea más complicada (tengo más instrucciones que decodificar).

También existen otros esquemas más bien teóricos, como lo pueden ser los set de instrucciones MISC (Minimal Instruction Set Computer), que solo ofrecen un mínimo de operaciones necesarias, y los OISC (One Instruction Set Computer), que solo ofrece una única instrucción.

5.4.2. Longitud de las instrucciones

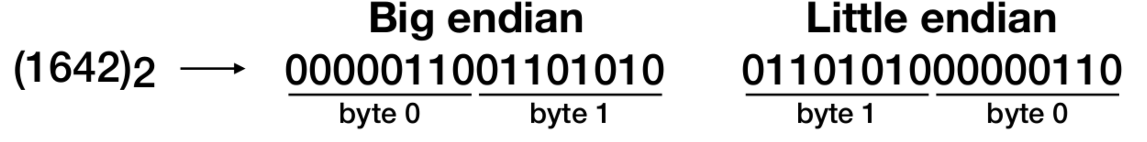
Puede ocurrir que, cuando tengo una gran cantidad de instrucciones, no sea capaz de codificar todas las instrucciones en una sola palabra. Cuando tengo muchas instrucciones, voy a tener que codificar muchas cosas muy diversas, y por lo tanto no me van a alcanzar una sola palabra de, por ejemplo, 16-bits. Entonces, puedo llegar a necesitar que las instrucciones tengan longitud variable: determinadas instrucciones tienen 16-bits, otras 32-bits (y por lo tanto requiere de traer dos palabras), o incluso 3 palabras. Es decir, podemos diseñar una ISA con instrucciones con longitud variable o con longitud fija.

5.4.3. Forma de representar la información

No es lo mismo representar enteros con Ca2 que en notación con signo, etc. Una vez que decido de qué manera represento la información, esto va a determinar de qué manera se va a llevar a cabo las operaciones aritméticas: al programador le digo que codifique los enteros en Ca2, porque espero recibir determinadas tiras de bits que soy capaz de operar a bajo nivel.

5.4.4. Big endian vs Little endian

La convención“endian” refiere a la forma en la que se organizan los datos que requieren más de un bytes para ser almacenados. Si los bytes menos significativos se alojan en las posiciones de memoria menores lo llamamos **Little Endian**. Si se alojan en las posiciones de memoria mayores lo llamamos **Big Endian**. Notemos que no se invierte la representación bit a bit, solo el orden de los bytes que conforman la representación.



Esto surge a partir de que, originalmente, había procesadores cuyo tamaño de palabra era de 8 bits, y por lo tanto solo podía comunicar de a 1 byte de la memoria al procesador, pero los datos eran de 16 bits. Entonces tenían que traer bloques de 8 bits hasta completar los 16. Entonces, cuando traía los primeros 8 bits, ¿estos correspondían a los menos significativos o a los más?. Siempre que tengamos datos que están representados con mayor cantidad de bits que el tamaño de la palabra del procesador, vamos a tener este problema.

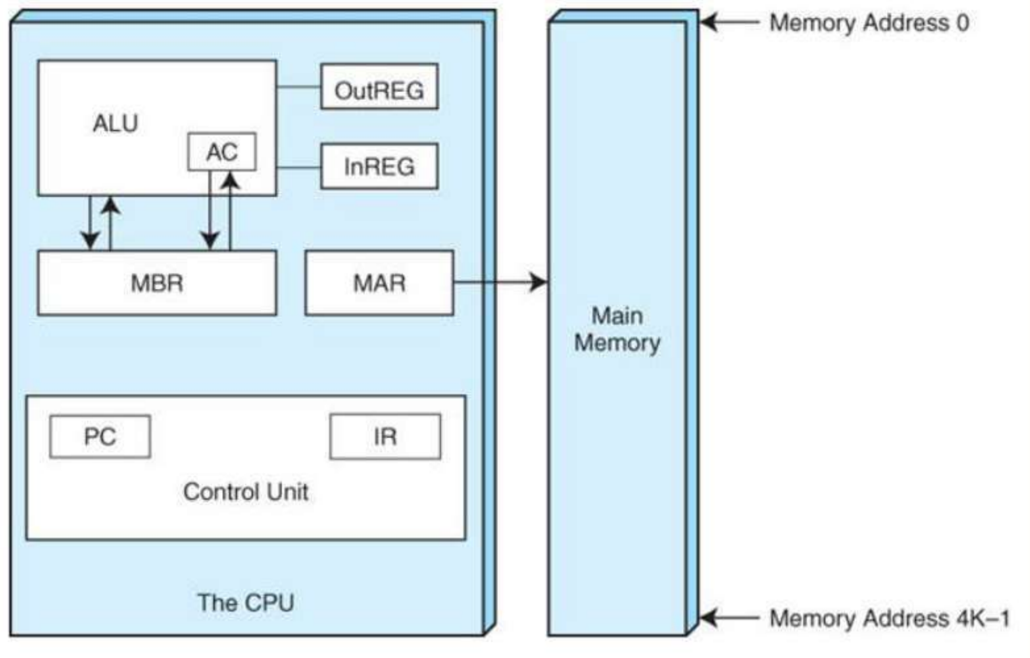
5.5. Tipos de Arquitecturas

- **Arquitectura de Stack**: La máquina de Stack tiene una memoria estandar, que puede ser accedida de forma arbitraria, sin embargo, internamente, el procesador no tiene registros, sino que en su lugar tiene una memoria, pequeña, organizada como un stack (solo se puede acceder a las cosas de arriba). Esta es una arquitectura más bien teórica, pero, al ser una arquitectura de naturaleza diferente, es interesante ver como funcionaría.
- **Arquitectura de Acumulador**: La máquina de acumulador no tiene una pila, sino que tiene registros. En particular, tiene un registro distinguido, el **registro acumulador** (AC), y uno alternativo para multiplicar (MQ). La idea general es que todas las instrucciones de la ISA utilizan como argumento implícito a este registro acumulador, de manera que funciona como registro parcial en el que se acumulan los resultados parciales.
- **Arquitectura con Registros de Propósito General**: Es una máquina que no tiene un stack, no tiene un registro (o dos) distinguidos, sino que cuenta con un **banco de registros** de los que puede disponer y utilizar libremente para guardar información, realizar operaciones aritméticas, etc.

5.5.1. Ejemplo de Arquitectura de Acumulador: MARIE

La arquitectura MARIE (Machine Architecturethat is Really Intuitive and Easy) es una arquitectura de acumulador que cuenta con las siguientes características:

- Representación: binaria en complemento a 2
- Memoria: 4K (direcciones de 12 bits), accedida por palabras
- Palabra: 16 bits
- Instrucciones: tamaño fijo 16 bits (4 para el código de operación y 12 para las direcciones) Fijo el tamaño de la instrucción al tamaño de la palabra, lo cual es conveniente ya que el tamaño de la palabra me determina la cantidad de información que puedo mover entre la memoria y el procesador. Me interesa que las instrucciones midan un múltiplo de palabra (en caso de tamaño variable)
- ALU: 16 bits. Me interesa que los datos tengan el mismo tamaño que la palabra. Ya que sino se complejiza la organización de la computadora (hay que realizar varios ciclos de fetch antes de poder ejecutar).
- Registros: 7 para control y movimiento de datos.



- AC: que tiene un registro interno que usamos como acumulador
- MBR: Memory Buffer Register, es a donde va a parar la información que va a parar la memoria, o es a donde tengo que colocar la informacion para que esta sea almacenada en memoria.
- MAR: Memory Address Register, es donde guardo la dirección de memoria que quiero acceder.
- Tengo una unidad de control con 2 registros internos:
- PC: Program Counter, contiene un puntero a la siguiente instrucción a ejecutar
- IR: Instruction Register, es un lugar que recibe la instrucción que debo ejecutar.
- Out/InREG: dos registros que me sirven para direccionar la memoria, traer y llevar información.

Notemos que todas las operaciones, que no sean de control del flujo del programa, se utiliza de manera implícita el registro acumulador. El conjunto de instrucciones de la arquitectura MARIE es el siguiente:

OpCode	Instrucción	Efecto
0000	JnS X	Almacena PC en X y Salta a X+1
0001	Load X	AC = [X]
0010	Store X	[X] = AC
0011	Add X	AC = AC + [X]
0100	Subt X	AC = AC - [X]
0101	Input	AC = Entrada de Periférico
0110	Output	Enviar a un periférico contenido AC
0111	Halt	Detiene la Ejecución
1000	SkipCond Cond	Salta una instrucción si se cumple la
1001	Jump Dir	PC = Dir
1010	Clear	AC = 0
1011	Addi X	AC = AC + [[X]]
1100	Jumpi X	PC = [X]

5.5.2. Arquitectura de Registros de Propósito General: Máquina ORGA1

- Representación: binaria en complemento a 2.
- Memoria: 64K, accedida por palabras (16 bits de direcciones)
- Palabra: 16 bits
- Instrucciones: tamaño variable 16/32/48 bits dependiendo de si es de 0, 1 ó 2 operandos (4 para el código de operación y 12 para operando 1, 16 para operando 2 y 16 para operando 3)
- ALU: 16 bits
- Registros: 8 para uso general (R0 — R7), 3 para control (PC-program counter, SP-stack pointer, Flags)

No solo tenemos instrucciones de tamaño variable, sino que además tenemos opcodes de tamaño variable, dependiendo del tipo de instrucción. Veamos nuestro set de instrucciones:

***Tipo 1*: Instrucciones de dos operandos**

4 bits	*6 bits*	*6 bits*	*16 bits*	*16 bits*
cod. op.	destino	fuente	constante destino (opcional)	constante fuente (opcional)

operación	cod. op.	efecto
MOV d, f	0001	$d \leftarrow f$
ADD d, f	0010	$d \leftarrow d + f$ (suma binaria)
SUB d, f	0011	$d \leftarrow d - f$ (resta binaria)
AND d, f	0100	$d \leftarrow d$ and f
OR d, f	0101	$d \leftarrow d$ or f
CMP d, f	0110	Modifica los *flags* según el resultado de $d - f$ (resta binaria)
ADDC d, f	1101	$d \leftarrow d + f +$ *carry* (suma binaria)

Formato de operandos destino y fuente.

Modo	Codificación	Resultado
Inmediato	000000	c16
Directo	001000	[c16]
Indirecto	011000	[[c16]]
Registro	100rrr	Rrrr
Indirecto registro	110rrr	[Rrrr]
Indexado	111rrr	[Rrrr + c16]

c16 es una constante de *16 bits*.

Rrrr es el registro indicado por los últimos tres *bits* del código de operando.

Las instrucciones que tienen como destino un operando de tipo *inmediato* son consideradas como inválidas por el procesador, excepto el CMP.

Cuando tengo que caracterizar cuál es la fuente, y cuál es el destino, tenemos que seguir la tabla de formato para estos operandos. Podemos ver que tenemos distintos modos de traer la información con la cual vamos a operar. Analizaremos en detalle los **modos de direccionamiento** en el capitulo 6. Por ahora veamos qué tipos de instrucciones tenemos, y su formato.

***Tipo 2*: Instrucciones de un operando**

Tipo 2a: Instrucciones de un operando destino.

4 bits	*6 bits*	*6 bits*	*16 bits*
cod. op.	destino	000000	constante destino (opcional)

operación	cod. op.	efecto
NEG d	1000	$d \leftarrow 0 - d$ (resta binaria)
NOT d	1001	$d \leftarrow$ not d (bit a bit)

El formato del operando *destino* responde a la tabla de formatos de operando mostrada más arriba.

Tipo 2b: Instrucciones de un operando fuente.

4 bits	*6 bits*	*6 bits*	*16 bits*
cod. op.	000000	fuente	constante fuente (opcional)

operación	cod. op.	efecto
JMP f	1010	PC $\leftarrow f$
CALL f	1011	[SP] $\leftarrow$ PC, SP $\leftarrow$ SP $-$ 1, PC $\leftarrow f$

***Tipo 3*: Instrucciones sin operandos**

4 bits	*6 bits*	*6 bits*
cod. op.	000000	000000

operación	cod. op.	efecto
RET	1100	PC $\leftarrow$ [SP+1], SP $\leftarrow$ SP + 1

***Tipo 4*: Saltos condicionales**

Las instrucciones en este formato son de la forma *Jxx* (salto relativo condicional). Si al evaluar la condición de salto en los *flags* el resultado es *1*, el efecto es incrementar el PC con el valor de los *8 bits* de desplazamiento, representado en *complemento a 2* de *8 bits*. En caso contrario, la instrucción no produce efectos.

8 bits	*8 bits*
cod. op.	desplazamiento

Codop	Operación	Descripción	Condición de Salto
1111 0001	JE	Igual / Cero	Z
1111 1001	JNE	Distinto	not Z
1111 0010	JLE	Menor o igual	Z or (N xor V)
1111 1010	JG	Mayor	not (Z or (N xor V))
1111 0011	JL	Menor	N xor V
1111 1011	JGE	Mayor o igual	not (N xor V)
1111 0100	JLEU	Menor o igual sin signo	C or Z
1111 1100	JGU	Mayor sin signo	not (C or Z)
1111 0101	JCS	Carry / Menor sin signo	C
1111 0110	JNEG	Negativo	N
1111 0111	JVS	Overflow	V

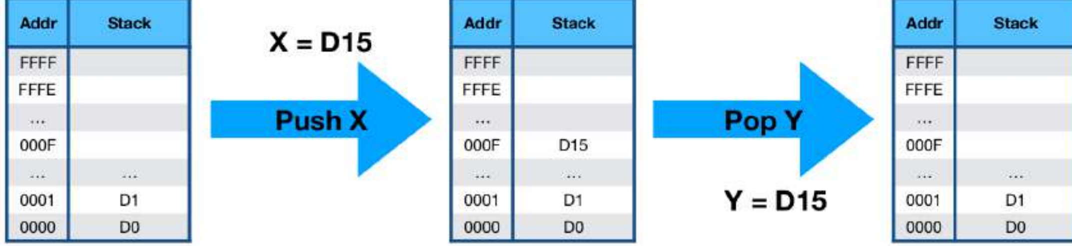
5.5.3. Arquitectura de Stack:

Jan Lukasiewicz: Matemático y filósofo polaco, 1878 — 1956. Entre sus grandes contribuciones se cuenta la axiomatización compacta de la lógica proposicional, la lógica trivaluada y la notación polaca inversa.

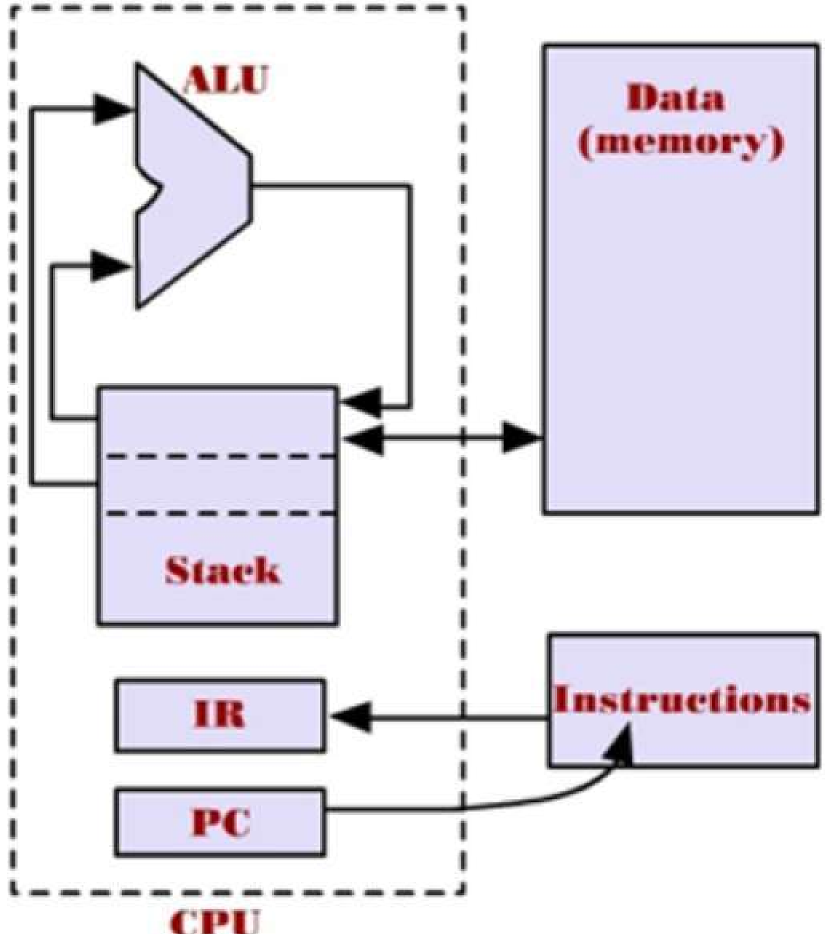
La notación polaca inversa u orden posfijo indica la operación a realizarse detrás de los operandos: $X + Y \rightarrow XY+$

No requiere paréntesis pues no es ambigua.

Un stack o pila posee dos operaciones: Push que permite colocar un dato en la primera posición libre y Pop que permite retirar el último dato que se encuentra en la pila.



En esta arquitectura, en vez de tener un banco de registros, contamos con una pila en la cual vamos apilando los distintos operandos, que traemos de la memoria:



Un set de instrucciones para este tipo de arquitectura podría ser el siguiente:

- Push / Pop: requieren una dirección de memoria como operando.
- Add / Mult, LE / GE/ Eq: realizan la operación con los dos elementos en el tope del stack.
- JMPT / JMPF: (jump true/false) requieren una dirección de memoria como operando.

5.6. Subrutinas

Posibilitan la reutilización de código a partir de modularizar fragmentos que son invocados, ejecutados para luego retornar a la ejecución del programa que lo invocó.

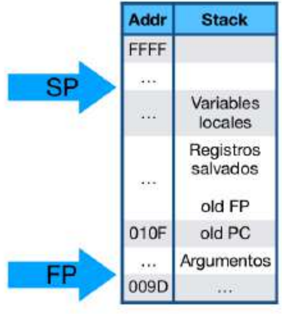
Los sistemas operativos implementan un stack o pila que posibilita la invocación de subrutinas preservando la información del programa que la invocó de forma que la ejecución pueda retornar a la instrucción siguiente a dicha invocación.

La instrucción `call Sub` implícitamente realiza `push PC`, preservando el punto de retorno a la ejecución del programa invocador y $PC = Sub$ indicando que se debe continuar la ejecución dirección de comienzo de la subrutina ‘`Sub`.

La instrucción `ret` implícitamente realiza `pop PC` restaurando el punto a partir del cual se debe continuar la ejecución de programa invocado.

La subrutina es responsable de preservar los valores de todo los registros que utilice en el stack y de restaurarlos antes de retornar al programa invocador.

Cuando las subrutinas reciben parámetros de los programas invocadores estos son pasados utilizando el stack del sistema. Esto introduce el concepto de bloque de activación de un código en ejecución:



–>FP y SP son registros utilizados para apuntar a los límites del bloque de activación de la rutina que se está ejecutando en este momento

–>Esto obliga a la rutina a salvar, entre los registros el FP anterior para retornar restaurando el bloque de activación del programa invocador

Capítulo 6

Instruction Set Architecture (ISA) - Parte 2

6.1. Introducción

Ya vimos cómo se estructura una arquitectura, proveyendo instrucciones que se pueden utilizar para construir programas, con recursos de bajo nivel provistos por el procesador.

Una parte central en la discusión al momento de diseñar una ISA son los **modos de direccionamiento** de la memoria. Si recordamos la definición del modelo de cómputo de Von Neumann - Turing:

- Los programas y los datos se almacenan en la misma memoria sobre la que se puede leer y escribir
- La operación de la máquina depende del estado de la memoria
- El contenido de la memoria es accedido a partir de su posición
- La ejecución es secuencial (a menos que se indique lo contrario)

Notamos que, prácticamente, todo lo que hagamos va a implicar acceder a la memoria, ya que tanto nuestros datos como los programas están almacenados en la misma. Entonces, tanto el ciclo de instrucción, para traer nuestro programa y poder ejecutarlo, como las propias instrucciones del programa, que necesitar ir a buscar datos a la memoria y traerlos para realizar el cómputo correspondiente, implica acceder a la memoria.

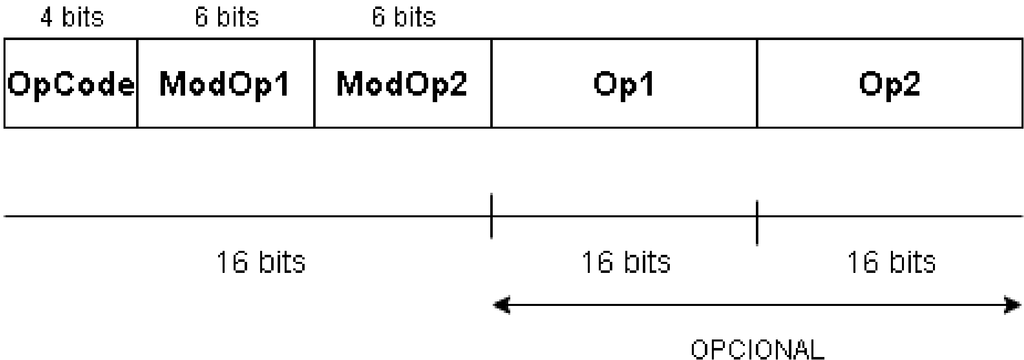
Si nos concentramos únicamente en qué pasa con las instrucciones del lenguaje de programación, prácticamente todas las instrucciones tienen operandos que, mayoritariamente, se encuentran en la memoria. En general, los programas hacen cómputo sobre **variables** las cuales son nada más un pequeño espacio de memoria reservado para almacenar un valor.

Luego, un programa, que manipule una variable, termina yendo a buscar un dato a la memoria, y ese ir a buscar a la memoria para obtener los datos hace que el modo de direccionamiento cobre tanta relevancia. Los modos de direccionamiento son las maneras en las que las instrucciones pueden operar sobre los contenidos de la memoria.

Nos vamos a concentrar en la ISA de Registros de Propósito General, ya que es la más rica en cantidad de instrucciones, en términos de cosas que se pueden hacer (puedo jugar con más operandos), y por lo tanto nos da una mejor pauta sobre qué significa direccionar la memoria y de qué tan complejo puede volverse.

Lo que habíamos visto cuando vimos este set de instrucciones es que teníamos instrucciones de tamaño variable, y podían ser de 16, 32 o 48 bits, dependiendo de la naturaleza de la instrucción y la cantidad de operandos que necesitaba para ejecutar.

Estas instrucciones estaban tabuladas con un código de operación de 4 bits. Después teníamos dos bloques de 6 bits, que nos decían cuales eran las naturalezas de cada uno de los dos operandos que venían a continuación. Luego, teníamos 16 bits, obligatorios, que incluyen código de operación, más dos modalidades de operandos, y, dependiendo de esa modalidad, teníamos 16 bits (para un operando) o 32 bits (para dos operandos) que tenían que traerse de memoria.



Estos operandos podían ser valores constantes (harcodeados en la instrucción), registros, o referencias a la memoria (direccionamiento directo, indirecto, indexado). La enorme mayoría de los direccionamientos van a ser a memoria, ya que programamos usando variables, que hacen referencia a un lugar de la memoria que contiene un dato.

Este set de instrucciones nos provee 6 modos de direccionamiento diferentes:

Formato de operandos destino y fuente.

Modo	Codificación	Resultado
Inmediato	000000	c16
Directo	001000	[c16]
Indirecto	011000	[[c16]]
Registro	100rrr	Rrrr
Indirecto registro	110rrr	[Rrrr]
Indexado	111rrr	[Rrrr + c16]

- **Modos de acceso a memoria**: requieren de un operando en la instrucción, resultando en instrucciones más largas.
  - **Inmediato**: vamos a decirle a una instrucción que el operando es un inmediato cuando esos 16-bits que tengo que traer son un número (constante). Por ejemplo, `ADD X`, **4**.
  - **Directo o Absoluto**: lo que tengo en el operando de la instrucción es una dirección de memoria

que apunta al dato que nos interesa, típicamente una variable. Como estamos trayendo una dirección de memoria, y no el dato en sí,, cuando queremos hacer una operación con ese dato, primero tenemos que ir a buscarlo y traerlo para que la ALU lo pueda utilizar.

- **Indirecto**: la analogía del modo indirecto es la utilización de punteros que hacen referencia a una variable. El puntero termina siendo una dirección de memoria en la que no se almacena el dato, sino que se tiene la posición de memoria en donde está el dato.

- **Modos de acceso a registros**: no requieren un operando en la instrucción, resultando en instrucciones más cortas.

  - **Registro**: se utiliza cuando tengo el dato en el registro número `rrr`, por lo que no se requiere de acceder a la memoria.

  - **Indirecto Registro**: es análogo al acceso directo a memoria, solo que se utiliza cuando tengo la dirección de memoria del dato en el registro número `rrr`, en vez de tener que pasar como operando a la dirección de memoria del dato.

  - **Desplazamiento o Indexado**: es parecido al modo Indirecto Registro, pero en vez de tener la dirección de memoria en el registro `rrr`, este se encuentra desplazado por un inmediato (constante). Este modo de direccionamiento, en general, lo usamos (en Orga 2) para acceder a distintos valores de un struct (de C, no de C++). La idea es que en el registro `rrr` tenemos la dirección de memoria que apunta al inicio del struct, y luego nos desplazamos sobre el mismo (con una constante) para poder acceder a los distintos campos del struct.

Notemos que en esta arquitectura se pueden utilizar cualquier combinación de estos modos de direccionamiento, siempre y cuando tenga sentido en el contexto de la instrucción (no se puede utilizar como destino a un inmediato). Esto no se cumple en otras arquitecturas.

La unidad de control, dependiendo de los primeros 16-bits sabe exactamente cómo ejecutar la instrucción, no tiene el dato, pero sabe cómo planificar la ejecución de la instrucción, ya que sabe qué modos de direccionamientos se están utilizando, y por lo tanto puede realizar las indirecciones necesarias para obtener los valores desde la memoria y para guardar el resultado.

6.2. Ortogonalidad

Hay un elemento que es central a la manera en la que se llevan a cabo los accesos a memoria llamado **ortogonalidad** de la ISA. La manera en la que se construye una ISA flexible es a partir de diseñar los modos de direccionamiento que son útiles para programar, y por otro lado un set de instrucciones que son las cosas que se pueden llegar a querer hacer con los datos.

Entonces, la combinación instrucción con el modo direccionamiento de cada uno de los operandos nos da una **instrucción efectiva**. Luego, la ortogonalidad de una ISA depende de si puedo usar cualquier modo de direccionamiento con cualquier instrucción, brindando una gran flexibilidad al momento de programar. Si se tienen n instrucciones, con m modos de direccionamiento, con k operandos, tenemos $n * m^k$ instrucciones efectivas diferentes.

Puede que muchas de estas instrucciones no sean realmente utilizadas como para justificar este aumento en el costo en la complejidad de la unidad de control, y por lo tanto, al momento de diseñar una ISA, hay que mantener un cierto balance.

6.3. Código de operación variable

Una de las cosas que tenemos que tener en cuenta al momento de diseñar la ISA, es que si nos limitamos a tener 4-bits de código de operación, y siempre son solo 4-bits, entonces como máximo vamos

a tener 16 operaciones diferentes, lo cual muchas veces no va a alcanzar.

Para poder expandir la cantidad de posibles operaciones, se puede utilizar códigos de operación de tamaño variable. Un caso en el que podemos aplicar esta idea es cuando tenemos instrucciones de longitud fija y operaciones con distinta cantidad de operandos:

```
0000 R1   R2   R3   }
                    } 15 3-address codes
1110 R1   R2   R3   }

1111 0000  R1   R2  }
                    } 14 2-address codes
1111 1101  R1   R2  }

1111 1110 0000  R1  }
                    } 31 1-address codes
1111 1111 1110  R1  }

1111 1111 1111 0000 }
                    } 16 0-address codes
1111 1111 1111 1111 }
```