CENTRO DE ENSEÑANZA TÉCNICA INDUSTRIAL
Organismo Público Descentralizado Federal
ceti

Ingeniería de Desarrollo de Software

Arquitectura de Computadoras

Material de apoyo

Por José Florentino Chavira Sánchez

Martes 18 de diciembre de 2018

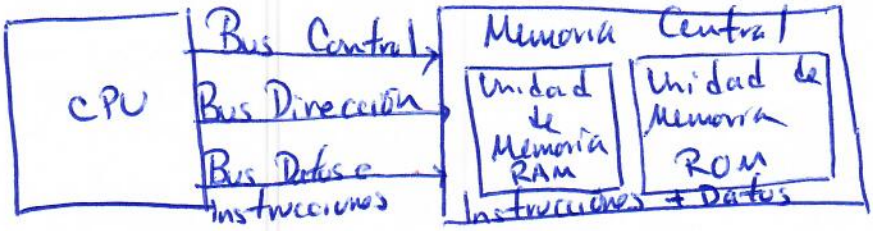
Índice

Historia

- Las técnicas y mecanismos de cálculo nacen para facilitar el trabajo con los números.
- El ábaco fue el primer instrumento de cálculo, se diseñó en el Lejano Oriente 3000 años a.C.
- El quipu o cordel parlante fue un desarrollo americano basado en cordeles y nudos, usado por el imperio Inca entre los siglos XII y XVI d.C.
- Los logaritmos fueron propuestas por John Neper en 1642 y facilitan notablemente las operaciones matemáticas.
- William Oughtred inventó la regla de cálculo en 1630, basada en los logaritmos.
- Blaise Pascal elaboró en 1642 una sumadora mecánica.
- Leibniz presentó un mecanismo que sumaba y multiplicaba en 1673.
- La técnica de tarjetas perforadas fue utilizada por Joseph Jacquard en el siglo XVII para elaborar patrones de tejido.
- Charles Babbage proyectó la máquina diferencial y la máquina analítica entre 1812 y 1832 que son antecedentes de la computadora moderna.
- Las tarjetas perforadas fueron utilizadas por Herman Hollerith para procesar la información del censo estadounidense de 1890.
- Los desarrollos electrónicos que permitieron la computación moderna son el bulbo (1906), el transistor (1947) y el C.I. (1958)
- Los circuitos electrónicos de las computadoras tienen como fundamento el álgebra lógica de George Boole.
- Las más renombradas calculadoras electromecánicas de principios de este siglo han sido: analizador diferencial (Bush, 1931), el Z3 (Zuse, 1936) el Mark I (Aiken, 1944).
- La primera generación de computadoras (1946–1959), se caracterizó por los bulbos y el lenguaje máquina. Algunos equipos representativos son: ENIAC, EDVAC, UNIVAC I

Arquitectura de Von Neumann.

- Tradicionalmente los sistemas con µP se basan en esta arquitectura.
- La Unidad central de proceso (CPU), está conectada a una memoria principal única (casi siempre sólo) RAM) donde se guardan las instrucciones del programa y los datos. A dicha memoria se accede a través de un sistema de buses único (control, direcciones y datos).



- bit
- byte
- ALU
- Bus Control
- Bus Dirección
- Bus Datos
- Bus inst.

El tener un único bus hace que el µP sea más lento en su respuesta, ya que no puede buscar en memoria una nueva instrucción mientras no finalice las transferencias de datos de la instrucción anterior.

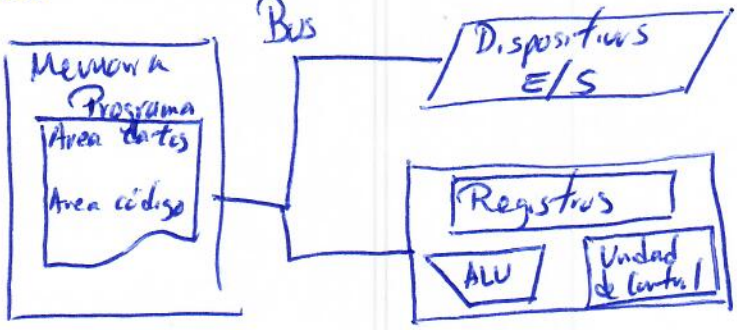
Limitaciones.

- La limitación de la longitud de las instrucciones por el bus de datos, que hace que el µP tenga que realizar varios accesos a memoria para buscar instrucciones complejas.
- La limitación de la velocidad de operación a causa del bus único para datos e instrucciones que no deja acceder simultáneamente a unos y otros, lo cual impide superponer ambos tiempos de acceso.

Partes

La Arq. Von Neuman consta de 4 secc

- ALU
- Control
- Memoria
- Disp-E/S

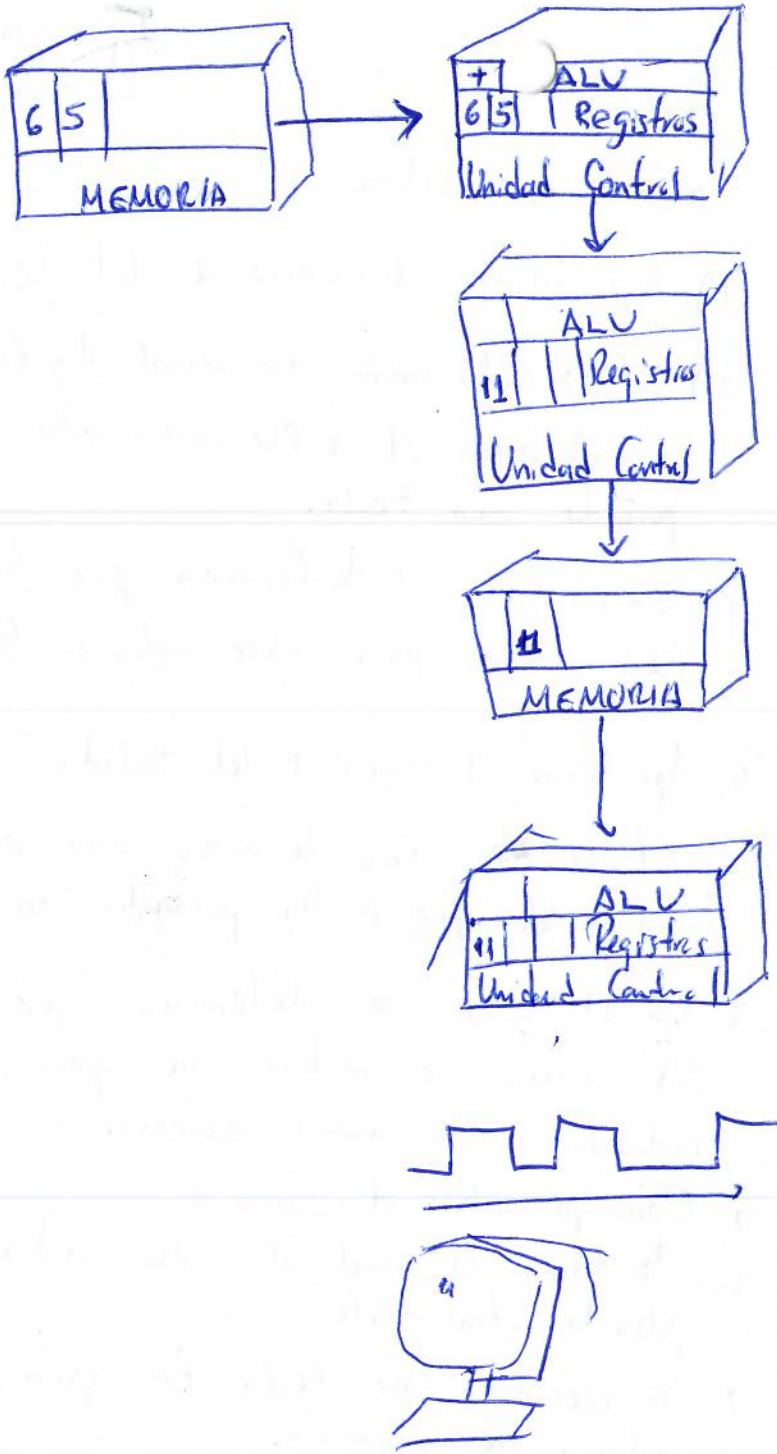


10. Para efectuar la operación se trasladan los operandos desde la memoria hasta los registros internos del CPU

11. Se suman el contenido de los registros y se coloca el resultado en un registro. Y posteriormente se regresa a la memoria (la operación aritmética se realiza por medio de la ALU

12. Como deseamos que nos muestre el resultado, el CPU busca nuevamente en memoria, extrae el resultado y lo coloca en un registro.

13. Del registro se envía un tren de pulsos hacia el monitor, donde estos son desplegados en forma del mismo M.



La **arquitectura Von Neumann** realiza o emula los siguientes pasos secuencialmente:

1) Obtiene la siguiente instrucción desde la memoria en la dirección indicada por el contador de programa y la guarda en el registro de instrucción.

2) Aumenta el contador de programa en la longitud de la instrucción para apuntar a la siguiente.

3) Descodifica la instrucción mediante la unidad de control. Ésta se encarga de coordinar el resto de componentes del ordenador para realizar una función determinada.

4) Se ejecuta la instrucción. Ésta puede cambiar el valor del contador del programa, permitiendo así operaciones repetitivas.

5) Regresa al paso N° 1.

Conclusión:

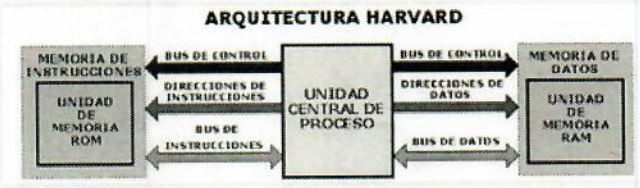
* La mayoría de las computadoras todavía utilizan la arquitectura Von Neumann, propuesta a principios de los años 40 por John Von Neumann.

* La arquitectura Von Neumann describe a la computadora con 4 secciones principales: la unidad lógica y aritmética (ALU), la unidad de control, la memoria, y los dispositivos de entrada y salida (E/S).

* En este sistema, la memoria es una secuencia de celdas de almacenamiento numeradas, donde cada una es un bit, o unidad de información.
La instrucción es la información necesaria para realizar, lo que se desea, con la computadora
Las celdas contienen datos que se necesitan para llevar a cabo las instrucciones, con la computadora.

* El tamaño de cada celda y el número de celdas varía mucho de computadora a computadora, y las tecnologías empleadas para la memoria han cambiado bastante: van desde los relés electromecánicos, tubos llenos de mercurio en los que se formaban los pulsos acústicos, matrices de imanes permanentes, transistores individuales a circuitos integrados con millones de celdas en un solo chip.

Arquitectura Harvard: Este modelo, que utilizan los Microcontroladores PIC, tiene la unidad central de proceso (CPU) conectada a dos memorias (una con las instrucciones y otra con los datos) por medio de dos buses diferentes.



Una de las memorias contiene solamente las instrucciones del programa (Memoria de Programa), y la otra sólo almacena datos (Memoria de Datos).

Ambos buses son totalmente independientes lo que permite que la CPU pueda acceder de forma independiente y simultánea a la memoria de datos y a la de instrucciones. Como los buses son independientes estos pueden tener distintos contenidos en la misma dirección y también distinta longitud.
Tambien la longitud de los datos y las instrucciones puede ser distinta, lo que optimiza el uso de la memoria en general.

Para un procesador de **Set de Instrucciones Reducido**, o **RISC (Reduced Instrucción Set Computer)**, el set de instrucciones y el bus de memoria de programa pueden diseñarse de tal manera que todas las instrucciones tengan una sola posición de memoria de programa de longitud.
Además, al ser los buses independientes, la CPU puede acceder a los datos para completar la ejecución de una instrucción, y al mismo tiempo leer la siguiente instrucción a ejecutar.

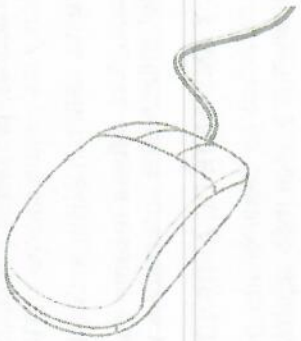
Ventajas de esta arquitectura:

* El tamaño de las instrucciones no esta relacionado con el de los datos, y por lo tanto puede ser optimizado para que cualquier instrucción ocupe una sola posición de memoria de programa, logrando así mayor velocidad y menor longitud de programa.

* El tiempo de acceso a las instrucciones puede superponerse con el de los datos, logrando una mayor velocidad en cada operación.

Tareas
1) • Modelos.
2) • Definiciones y pasos y aplicaciones.
3) • 2 Resumenes en ingles.
4) Historia
5) • IoT
6) • Prog. Nivel
7) • Big Data
8) • Super Computadora

- El bit es la mínima unidad de información, vale 0 o 1.
- El byte es un conjunto de ocho bits asociados, considerado como unidad.
- Los múltiplos del byte son: kilobyte (1024 bytes), megabyte (1024 KBytes), gigabyte (1024 MBytes), terabyte (1024 gigabytes).
- El *word* (palabra) es el grupo más grande de bits que el CPU puede manejar a un mismo tiempo. En general no es unidad de medición de memoria sino de la capacidad del procesamiento del CPU.
- La longitud de *word* (palabra) es el número de bits que hay en un word.
- La memoria primaria (principal o interna) pertenece al equipo central de la computadora. Es por completo electrónica y las transferencias de datos se verifican con gran rapidez.
- La memoria primaria se divide en RAM y ROM.
- La memoria RAM (*Random Access Memory*) permite la lectura y la escritura de datos. Requiere de energía eléctrica para mantener la información y es de acceso directo.
- La memoria ROM (*Read Only Memory*) sólo permite lectura de datos. Los datos que contiene son instrucciones de fábrica que son indispensables para el funcionamiento de la máquina. No requiere de energía eléctrica para mantener la información
- La memoria interna de una computadora se representa como un gran "casillero". Cada uno tiene un número de identificación (dirección) único. En cada casilla se guarda un byte de información.
- La memoria secundaria es externa a la computadora. Se utiliza para almacenar datos por tiempo prolongado. Es de tipo electromecánico, magnético u óptico.
- Algunos dipositivos de memoria secundaria son discos flexibles (disquetes), cintas magnéticas. discos duros, cartuchos, CD-ROM.
- Los datos en memoria secundaria se organizan en unidades llamadas archivos.
- El bus es el canal de comunicación entre el CPU y las demás unidades.
- El bus está físicamente formado por laminillas conductoras de electricidad.
- El bus principal de una computadora se divide en cuatro sub-buses: alimentación, control, direcciones y datos.



RESUMEN

- Una computadora analógica soluciona problemas utilizando señales analógicas (mecánicas, eléctricas o electrónicas).
- Una computadora digital es un dispositivo electrónico que procesa información mediante pulsos eléctricos.
- El *hardware* es el equipo físico de la computadora.
- El *software* son los programas y datos de las computadora.
- Las computadoras se componen de unidad de entrada, unidad de salida, unidad de memoria y unidad central de procesamiento (arquitectura Von Neumann).
- Los aparatos periféricos trabajan junto con la computadora y le permiten comunicación con el exterior.
- La unidad de entrada permite recibir información del exterior.
- Son dispositivos de entrada el teclado, el mouse y el joystick.
- La unidad de salida permite emitir información al exterior.
- Son dispositivos de salida el monitor de video, e impresoras.
- La unidad central de procesamiento (CPU) ejecuta las instrucciones, realiza las operaciones y controla los procesos.
- El CPU se compone de bloque de control, unidad aritmética/lógica, registros, reloj y contador de programa.
- La memoria almacena datos, instrucciones y resultados.
- Las memorias actuales se construyen con componentes electrónicos, mecánicos o magnéticos.
- Las unidades utilizadas para medición de memoria digital son el bit y el byte.

- El sub-bus de alimentación distribuye energía eléctrica.
- El sub-bus de control transmite señales para controlar los circuitos.
- El sub-bus de dirección lleva las direcciones de las celdas de memoria.
- El sub-bus de datos transporta datos entre los componentes.

¿?

1. Menciona la dif
2. Define el *hardw*
3. Describe la arqu
4. ¿Qué son los pe
5. Menciona tres e
6. ¿Para qué sirve
7. Describe los cor
8. ¿Qué es la mem
9. Define los térmi
10. ¿Cuáles son los
11. ¿Cuál es la difer
12. Explica las mem
13. ¿Qué son las dire
14. Menciona tres di
15. ¿Cómo se organiz
16. ¿Qué es un bus?
17. ¿Cuáles son los s

PREGUNTAS

1. Menciona la diferencia entre una computadora digital y una analógica.
2. Define el *hardware* y el *software*.
3. Describe la arquitectura de Von Neumann.
4. ¿Qué son los periféricos?
5. Menciona tres ejemplos de periféricos de entrada y tres de salida.
6. ¿Para qué sirve el CPU?
7. Describe los componentes del CPU.
8. ¿Qué es la memoria?
9. Define los términos bit, byte, word, longitud de word.
10. ¿Cuáles son los múltiplos del byte?
11. ¿Cuál es la diferencia entre memoria primaria y memoria secundaria?
12. Explica las memorias RAM y ROM.
13. ¿Qué son las direcciones de memoria?
14. Menciona tres dispositivos de memoria secundaria.
15. ¿Cómo se organiza la información en la memoria secundaria?
16. ¿Qué es un bus?
17. ¿Cuáles son los sub-buses de una computadora?

Otras arquitecturas

Como mencionamos anteriormente, el esquema teórico que sustenta las computadoras personales es conocido como arquitectura Von Neumann; sin embargo, existen otros modelos que sirven de base para construir otras computadoras.

A continuación mencionamos algunos, pero su explicación sale del alcance de este texto:

1. Arquitectura RISC. Utiliza un mínimo de instrucciones básicas y permite trabajar a gran velocidad. Esta arquitectura es utilizada por las estaciones de trabajo.

2. Arquitectura de procesamiento paralelo. Con este modelo se pueden ejecutar programas en forma simultánea ya que utiliza varios CPU trabajando al mismo tiempo.

3. Arquitectura de procesamiento vectorial. Según este modelo se pueden efectuar operaciones matemáticas con grandes grupos de datos a un mismo tiempo. Esta arquitectura es utilizada por las supercomputadoras.

Unit U 1.4 Week 1

Hardware Description Language

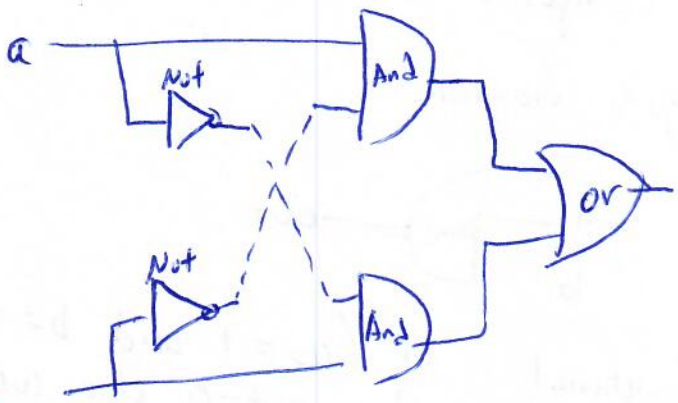
Design: from requirements to interface

General idea:
out = 1 when:
a And Not(b)
OR
b And Not (a)

a	b	out
0	0	0
0	1	1
1	0	1
1	1	0

Requirement

- Name chip
- Name of the inputs of the chips
- name of chip's output.




```
/** Xor gate: out = (a And Not(b)) Or
    (Not(a) And b)) */

CHIP Xor {
    IN a, b;
    OUT out;

    PARTS:
    // Implementation missing
}
```

Interface.

Gat interface.

```
PARTS:
Not (in=a, out=nota);
Not (in=b, out=notb);
And (a=a, b=notb, out=aAndNotb);
And (a=nota, b=b, out=notaAndb);
Or (a=aAndNotb, b=notaAndb, out=out);
```

Implementation

Negative numbers - Sign bit

0000 — 0
0001 — 1
0010 — 2
0011 — 3
0100 — 4
0101 — 5
0110 — 6
0111 — 7
1000 — -0
⋮
1111 — -1

2's Complement (15)

4 2 1
1 0 1
NOT
1 0 1 1

Using 4-bits 2's Complement representation

-2 +
-3
-5

1111 +
1011
10|1|0

14
+13
11

1110
+
1101
11011 = 27

1101 +
1011
10|0|0

0111 +
1011
10010

10

Adder

1) Half Adder – adds two bits
2) Full Adder – adds three bits
3) Adder – Adds two numbers.

Bin

4 2 1
0 1 0 1 +
1
0 1 1 0

8 4 2 1
0 1 0 1 +
1
0 1 1 0
1 0 0 1

1000 -8
1001 -7
1010 -6
1011 -5

Boolean Arithmetic

- Binary/Numb.
- Binary Add.
- Negative Num.

1111 -
0100
1011

1111 -
0100
0011 f
1

1111
0110
1001
1

1010

0111 f
1011
1010

0101 +
1011
10000
1011
0110 +
1011
10001

0110
0100
1010

0101
1011
10000

111
1011
10010

0100
1100
01000
1

0110
10010

Unit 1.7: Project 1 Overview

Elementary logic gates

- Not
- And
- Or
- Xor
- Mux
- DMux

16-bit variants

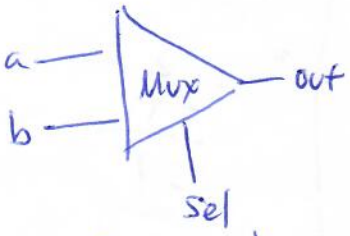
- Not16
- And16
- Or16
- Mux16

Multi-way variants

- Or8Way
- Mux4Way16
- Mux8Way16
- DMux4Way
- DMux8Way

1001
1011
10100

Multiplexor



if (sel==0)
  out=a
else
  out=b

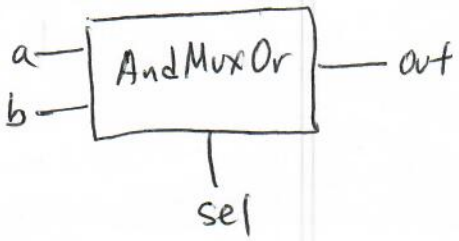
a	b	Sel	Out
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

Sel	out
0	a
1	b

CHIP Mux {
  IN a, b, sel;
  OUT out;
  PARTS:

- A 2-way multiplexor enables selecting, and outputting, one out of two possible inputs.
- Widely used in:
  - Digital design
  - Communication networks

Example: using mux logic to build a programmable gate



if (sel==0)
  out= (a And b)
else
  out = (a Or b)

- HDL is a functional / declarative language
- The order of HDL statements is insignificant.
- Before using a chip part, you must know its interface.

Common HDLs:
- VHDL
- Verilog
- Many more HDLs...

Our HDL
- Similar in spirit to other HDLs
- Minimal and simple
- Provides all you need for this course

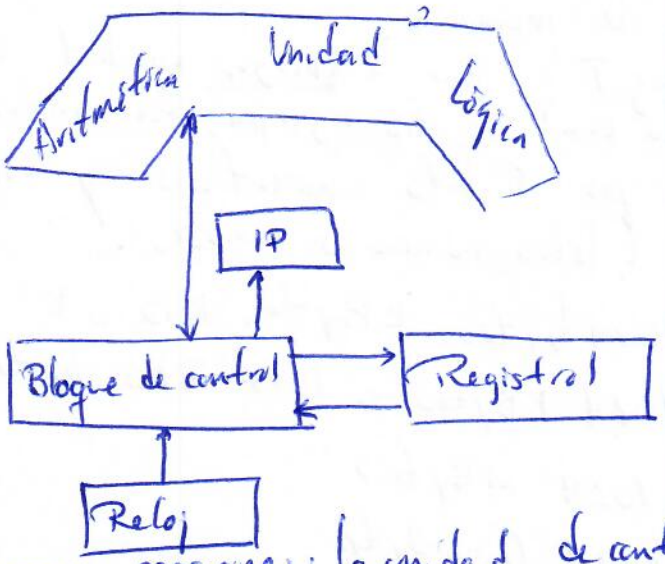
Hardware Simulation

Or.hdl

Simulation process
- Load the HDL file into the Hardware simulator.
- Enter values (0's and 1's) into the chip's input pins (e.g. a and b)
- Evaluate the chip's logic
- Inspect the resulting values of
  - The output pins (e.g. out)
  - The internal pins (e.g. notb, aAndNotb, notaAndb)

Interactive simulation

Unidad Central de procesamiento



El CPU se integra de varias secciones: la unidad de control, ALU, los registros, el reloj y el contador de programa.

1. Unidad de Control. Acepta y ejecuta las instrucciones recibidas desde la memoria.

La unidad de control se haya en un constante ciclo en el cual realiza actividades:

- Lectura de instrucción desde memoria
- Decodificación de la instrucción
- Ejecución de la instrucción
- Ajusta el contador de programa

2. ALU. Operaciones aritméticas: suma, resta, multiplicación o división, y las operaciones de tipo lógico: comparaciones, conjunciones, disyunciones, y negaciones

3. Registros. son una memoria provisional para almacenar los datos básicos que el procesador requiere en sus operaciones. La cantidad de registros varía de acuerdo al tipo de procesador, algunos de los más importantes son el acumulador, el apuntador de pila y las banderas.

Reloj emite pulsos eléctricos a intervalos constantes, lo que permite sincronizar las diferentes tareas del CPU.

5. Contador de Programa: indica la zona de la memoria donde se localiza

Demultiplexor

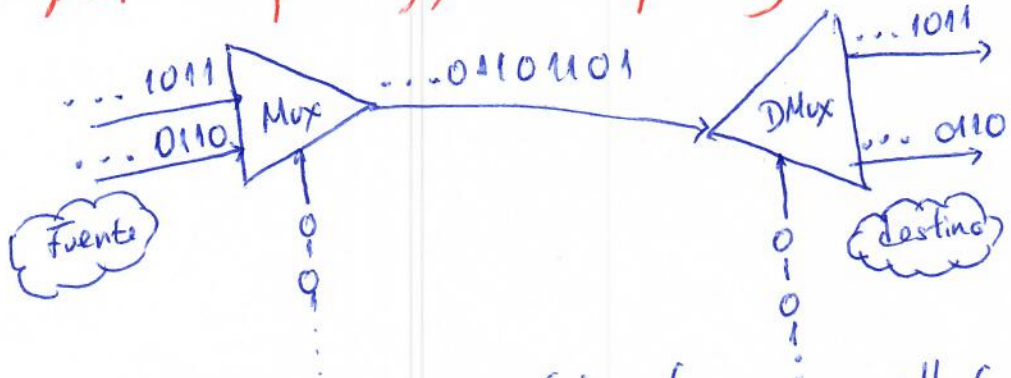

```
if (sel==0)
   {a,b} = {in, 0}
else
   {a,b} = {0, in}
```

in	sel	a	b
0	0	0	0
0	1	0	0
1	0	1	0
1	1	0	1

- Acts like the inverse of a multiplexor
- Distributes the single input value into one of two possible destinations

```
CHIP DMux {
   IN in, sel;
   OUT a, b;
PARTS:
```

Example: Multiplexing / demultiplexing in communications networks



- Each sel bit is connected to an oscillator that produces a repetitive train of alternating 0 and 1 signals
- Enables transmitting multiple messages on a single, shared communications line
- A common use of multiplexing /

NOT (NOT (x) AND) NOT (x OR y)) = x OR y

x	y	
0	0	0
0	1	1
1	0	1
1	1	1

Síntesis de Funciones Booleanas

Truth table to Boolean Expression.

x	y	z	f	
0	0	0	01	→ (NOT(x) AND NOT (Y) AND NOT (z)) OR
0	0	1	0	
0	1	0	1	→ (NOT (x) AND (Y) AND NOT(z)) OR
0	1	1	0	
1	0	0	1	→ ((X) AND NOT (Y) AND NOT (z))
1	0	1	0	
1	1	0	0	
1	1	1	0	

Identidades Booleanas

$(X \text{ AND } Y) = (Y \text{ AND } X)$
$(X \text{ OR } Y) = (Y \text{ OR } X)$ } ley conmutativa

$(X \text{ AND } (Y \text{ AND } Z)) = ((X \text{ AND } Y) \text{ AND } Z)$
$(X \text{ OR } (Y \text{ OR } Z)) = ((X \text{ OR } Y) \text{ OR } Z)$ } Asoc.

$(X \text{ AND } (Y \text{ OR } Z)) = (X \text{ AND } Y) \text{ OR } (X \text{ AND } Z)$
$(X \text{ OR } (Y \text{ AND } Z)) = (X \text{ OR } Y) \text{ AND } (X \text{ OR } Z)$ } Dist.

$\text{NOT } (X \text{ AND } Y) = \text{NOT } (X) \text{ OR } \text{NOT } (Y)$
$\text{NOT } (X \text{ OR } Y) = (\text{NOT } (X) \text{ AND } \text{NOT } (Y))$ } ley De Morgan

---

Algebra Booleana

$\text{NOT}(\text{NOT}(X) \text{ AND } \text{NOT } (X \text{ OR } Y)) =$ ← De Morgan
$\text{NOT } (\text{NOT } (X) \text{ AND } (\text{NOT } (X) \text{ AND } \text{NOT } (Y))) =$ ← Asociativa
$\text{NOT } ((\text{NOT}(X) \text{ AND } \text{NOT } (X)) \text{ AND } \text{NOT } (Y)) =$ ← Idempotencia
$\text{NOT } (\text{NOT } (X) \text{ AND } \text{NOT } (Y)) =$
$\text{NOT } (\text{NOT}(X)) \text{ OR } \text{NOT}(\text{NOT } (Y)) =$ — De Morgan
$X \text{ OR } Y$ — Doble negación

Datasheet
Intel®Galileo Gen 2
Development Board

Intel® Galileo Gen 2 Development Board

The 2nd generation Intel®Galileo board provides a single board controller for the maker community, students, and professional developers. Based on the Intel®Quark™SoC X1000, a 32-bit Intel®Pentium®processor-class system on a chip (SoC), the genuine Intel®processor and native I/O capabilities of the Intel Galileo board (Gen 2) provide a full-featured offering for a wide range of applications. Arduino-Certißed and designed to be hardware-, software-, and pin-compatible with a wide range of Arduino Uno R3 shields, the Intel Galileo Gen 2 board also provides a simpler and more cost-effective development environment compared to the Intel®Atom™processor- and Intel®Core™ processor-based designs.

2nd Generation Product Enhancements

The Intel Galileo board (Gen 2) delivers improved features and functionality in the following areas:

- 12 GPIOs fully native for greater speed and improved drive strength.
- 12-bit PWM for more precise control of servos and smoother response.
- 12 V Power-over-Ethernet capable.
- Power supplies from 7 V to 15 V are supported.
- Serial console UART header is compatible with FTDI USB converters.
- Console UART1 can be redirected to Arduino* headers in sketches, which can eliminate the need for soft-serial.

Arduino Uno R3*-compatible

Getting familiar with the board and developing applications is a snap because the Intel Galileo board (Gen 2) matches the Arduino 1.0 pinout and is also software-compatible with the Arduino Software Development Environment.

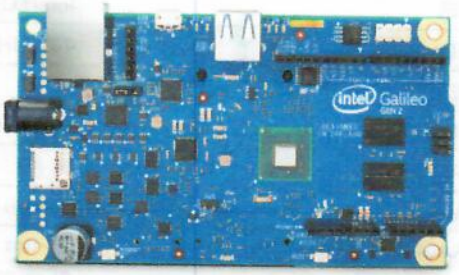
Open Source Hardware

The Intel Galileo board (Gen 2) is an open source hardware design. Schematics, Cadence Allegro board ßles, and bill of materials (BOM) are freely available for download.

Extensive Expandability

In addition to Arduino hardware and software compatibility, the Intel Galileo board (Gen 2) includes the following industry-standard I/O ports and features:

- Full-sized mini-PCI Express* slot
- 10/100 Mbps Ethernet* RJ45 port with PoE support
- Micro-SD slot
- TTL UART 6-pin header
- USB 2.0 Host port
- USB 2.0 Client port



Target Software

Use the Arduino Software Development Environment to create programs for Galileo called "sketches." To run a sketch on the board:

1. Connect power.
2. Connect the board's USB Client port to a computer.
3. Upload the sketch using the IDE interface.

The sketch runs on the Galileo board and communicates with the Linux* kernel in the board ßrmware using the Arduino I/O adapter. For complete details on programming your board, see the Intel®Galileo Getting Started Guide.

La salida del comparador es normalmente un voltaje bajo (0 lógico), pero cambia a un voltaje alto (1 lógico) cuando V_T excede a V_{TR}, lo cual indica que la temperatura del proceso es excesiva. Una disposición similar se utiliza para medir la presión; así, su salida asociada con el comparador pasa de bajo a alto cuando la presión es excesiva.

Ya que deseamos que la alarma se active cuando la temperatura *o* la presión sean demasiado altas, recuerde que las dos salidas del comparador pueden alimentarse a una compuerta OR de dos entradas. Así, la compuerta OR pasa al nivel ALTO (1) para cualquier condición de alarma y de este modo activará dicha alarma. Esta misma idea puede ampliarse con claridad a situaciones con más de dos variables de proceso.

EJEMPLO 3-2

Determine la salida de la compuerta OR en la figura 3-5. Las entradas *A* y *B* varían de acuerdo con los diagramas de tiempos que se muestran en la figura. Por ejemplo, *A* comienza en BAJO en t_0, va hacia ALTO en t_1, regresa a BAJO en t_3 y así sucesivamente.

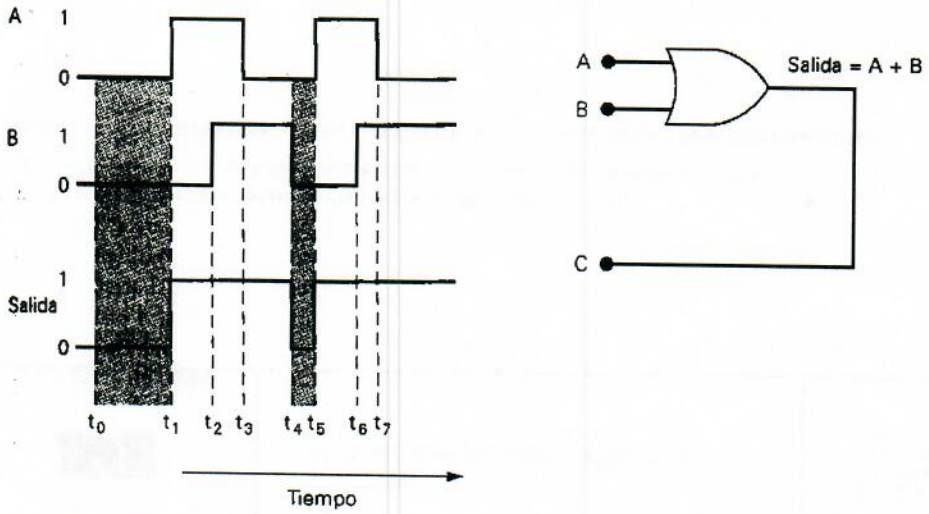
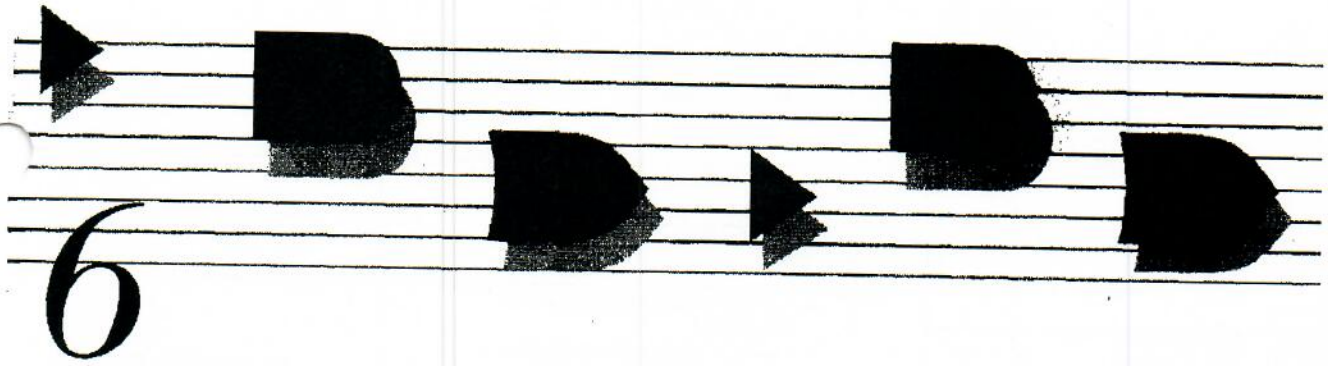


Figura 3-5 Ejemplo 3-2.

Solución

La salida de la compuerta OR se determina al observar que ésta se encontrará en ALTO siempre que *cualquiera* de las entradas se encuentre en este nivel alto. Cuando *A* cambia a ALTO en t_1, la SALIDA pasará a ALTO. Y permanecerá en este nivel hasta t_4, cuando ambas entradas estén en BAJO. Observe que los cambios que ocurren en el nivel de las entradas en t_2 y t_3 no tienen ningún efecto sobre la SALIDA, dado que una de las entradas permanece en el nivel ALTO mientras la otra cambia. Siempre y cuando una de las entradas de la compuerta OR permanezca en ALTO, la salida continuará en ALTO sin importar lo que ocurra en las demás entradas. El mismo razonamiento se emplea para deteminar el diagrama de tiempos para la SALIDA.



6

Aritmética digital: operaciones y circuitos

TEMARIO

Primero observaremos cómo se realizan diversas operaciones aritméticas con números binarios mediante el uso de "lápiz y papel"; luego estudiaremos los circuitos lógicos reales que estas operaciones efectúan en un sistema digital.

6-1 ADICIÓN BINARIA

La adición o suma de dos números binarios se efectúa exactamente en la misma forma que la suma de números decimales. De hecho, la adición binaria es más simple, ya que existen menos casos que deben aprenderse. Primero repasaremos brevemente la adición decimal:

$$\begin{array}{rccc} & & \downarrow & \downarrow \\ & 3 & 7 & 6 \quad \text{LSD} \\ + & 4 & 6 & 1 \\ \hline & 8 & 3 & 7 \end{array}$$

El dígito menos significativo (LSD; *least-significant digit*) se opera primero, produciendo una suma de 7. Luego se suman los dígitos que se encuentran en la segunda posición, para producir una suma de 13, lo que produce un **acarreo** de 1 en la tercera posición. Esto produce un resultado de 8 en la tercera posición.

En la adición binaria se siguen los mismos pasos generales. Sin embargo, sólo pueden ocurrir cuatro casos al sumar dos cifras binarias (bits) en cualquier posición. Éstos son:

$$\begin{aligned} 0 + 0 &= 0 \\ 1 + 0 &= 1 \\ 1 + 1 &= 10 = 0 + \text{acarreo de 1 a la siguiente posición} \\ 1 + 1 + 1 &= 11 = 1 + \text{acarreo de 1 a la siguiente posición} \end{aligned}$$

El último caso ocurre cuando los dos bits que se encuentran en cierta posición son 1 y existe un acarreo desde la posición anterior. A continuación se dan varios ejemplos de la suma de dos números binarios (los equivalentes decimales están entre paréntesis):

$$\begin{array}{rcrcrr} 011 & (3) & 1001 & (9) & 11.011 & (3.375) \\ + \ 110 & (6) & + \ 1111 & (15) & + \ 10.110 & (2.750) \\ \hline 1001 & (9) & 11000 & (24) & 110.001 & (6.125) \end{array}$$

No es necesario considerar la adición de más de dos números binarios al mismo tiempo, ya que en todos los sistemas digitales la circuitería que en realidad efectúa la suma sólo puede manejar dos números a la vez. Cuando van a sumarse más de dos números, se suman los dos primeros y el resultado se agrega al tercer número; y así sucesivamente. Esto no constituye una desventaja grave, ya que las máquinas digitales modernas pueden realizar comúnmente una operación de adición en microsegundos o menos.

La suma o adición es la operación aritmética de mayor importancia en los sistemas digitales. Como veremos más adelante, las operaciones de sustracción, multiplicación y división, que se efectúan en la mayoría de las computadoras y calculadoras digitales modernas, en realidad utilizan únicamente la adición como operación básica.

PREGUNTAS DE REPASO

1. Sume los siguientes pares de números binarios:
   (a) 10110 + 00111 (b) 011.101 + 010.010 (c) 10001111 + 00000001

6-2 REPRESENTACIÓN DE NÚMEROS CON SIGNO

En las computadoras digitales, los números binarios se representan por medio de un conjunto de dispositivos de almacenamiento binario (por lo general flip-flops). Cada dispositivo representa un bit. Por ejemplo, un registro de FF de 6 bits podría almacenar números binarios que van desde 000000 hasta 111111 (de 0 a 63 en decimal). Esto representa la *magnitud* del número. Como la mayoría de las computadoras y calculadoras digitales manejan números negativos y positivos, se necesita algún medio de representación para el *signo* del número (+ o −). Esto se lleva a cabo agregando otro bit al número denominado **bit de signo.** En términos generales, la convención común que se ha adoptado es que un 0 en el bit de signo representa un número positivo y un 1 representa un número negativo. Esto se ilustra en la figura 6-1. El registro *A* contiene los bits 0110100. El cero en el bit del extremo izquierdo (A_6) es el bit de signo que representa +. Los otros seis bits son la magnitud del número 110100_2, que es igual a 52 en decimal. De este modo, el número almacenado en el registro *A* es + 52. En forma análoga, el número almacenado en el registro *B* es −52, ya que el bit de signo es 1, que representa −.

El bit de signo se utiliza para indicar la naturaleza positiva o negativa del número binario almacenado. Los números en la figura 6-1 están formados por un bit de signo y seis bits de magnitud. Estos últimos son el verdadero equivalente binario del valor decimal que representan. Lo anterior recibe el nombre de sistema **signo-magnitud** para la representación de números binarios con signo.

Aunque el sistema signo-magnitud es bastante sencillo, por lo general las computadoras y calculadoras no lo utilizan, porque la implementación del circuito es mucho más compleja que la de otros sistemas. El sistema más empleado para representar números binarios con signo es el **sistema de complemento a 2.** Antes de estudiarlo, primero veremos cómo formar el complemento a 1 y el complemento a 2 de un número binario.

Forma complemento a 1 El complemento a 1 de un número binario se obtiene cambiando cada 0 por 1 y viceversa. En otras palabras, se cambia cada bit del número por su complemento. A continuación se ilustra este proceso.

```
1 0 1 1 0 1   número binario original
↓ ↓ ↓ ↓ ↓ ↓
0 1 0 0 1 0   se complementa cada bit para formar el complemento a 1
```

De este modo, se afirma que el complemento a 1 de 101101 es 010010.

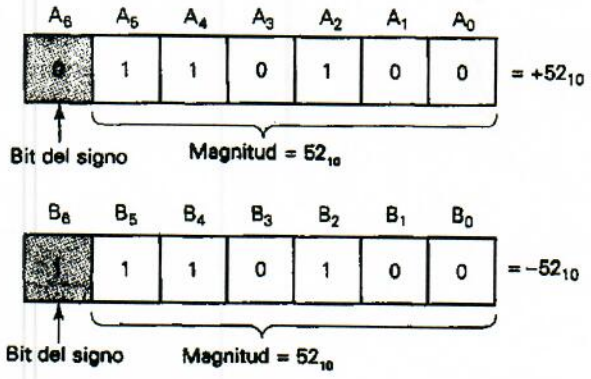


Figura 6-1 Representación de números con signo en forma de signo magnitud.

Forma complemento a 2 El complemento a 2 de un número binario se obtiene tomando el complemento a 1 y sumándole 1 al bit menos significativo. A continuación se ilustra este proceso para el número $101101_2 = 45_{10}$.

1 0 1 1 0 1	equivalente binario de 45
0 1 0 0 1 0	se complementa cada bit para formar el complemento a 1
+ 1	se suma 1 para obtener el complemento a 2
0 1 0 0 1 1	representación en complemento a 2 del número binario original

Entonces decimos que 010011 es la representación del complemento a 2 de 101101.

El siguiente es otro ejemplo de conversión de un número binario a su representación en complemento a 2:

1 0 1 1 0 0	número binario original
0 1 0 0 1 1	complemento a 1
+ 1	se suma 1
0 1 0 1 0 0	representación en complemento a 2 del número binario original

Representación de números con signo mediante el complemento a 2 El sistema complemento a 2 para representar números con signo, trabaja de la siguiente manera:

- Si el número es positivo, la magnitud está representada por su equivalente binario verdadero y se agrega un cero antes del bit más significativo. Esto se muestra en la figura 6-2 para el número $+45_{10}$.
- Si el número es negativo, la magnitud está representada por su equivalente en complemento a 2 y se agrega un 1 antes del bit más significativo. Lo anterior se ilustra en la figura 6-2 para el número -45_{10}.

El sistema complemento a 2 se emplea para representar números con signos porque, como se verá más adelante, permite efectuar la operación de sustracción mediante una adición. Esto es importante ya que significa que la computadora digital puede usar la misma circuitería tanto para sumar como para restar, ahorrando así en hardware.

Figura 6-2 Representación de números con signo en el sistema complemento a 2.

EJEMPLO 6-1

Represente cada uno de los siguientes números decimales con signo como números binarios con signo en el sistema complemento a 2. Utilice un total de 5 bits, incluido el bit de signo:
(a) + 13, **(b)** −9, **(c)** +3 **(d)** −2, **(e)** −8.

Solución

(a) Como el número es positivo, la magnitud (13) se representará en su forma de magnitud verdadera, es decir $13 = 1101_2$. Si se agrega el bit de signo 0 se tiene

+13 = 01101

bit de signo ↑

(b) Puesto que el número es negativo, la magnitud (9) tiene que ser representada por su forma de complemento a 2:

$9_{10} = 1001_2$	
0110	complemento a 1
+ 1	se suma 1 al LSB
0111	complemento a 2

Cuando se agrega el bit de signo 1, el número complemento con signo se convierte en

−9 = 10111

bit de signo ↑

El procedimiento que acabamos de seguir requirió dos etapas. Primero, determinamos el complemento a 2 de la magnitud y luego agregamos el bit de signo. Esto se puede realizar en un solo paso si se incluye el bit de signo en el proceso complemento a 2. Por ejemplo, para determinar la representación de −9, se inicia con la representración de +9, *incluyendo el bit de signo* y se complementa a 2 a fin de obtener la representación de −9.

+ 9 = 01001	
10110	se complementa cada bit
+ 1	se suma 1 al LSB
10111	representación complemento a 2 de −9

El resultado es, desde luego, el mismo que antes.

(c) El valor decimal 3 se puede representar en binario utilizando sólo 2 bits. Sin embargo, el enunciado del problema pide una magnitud de 4 bits precedida por un bit de signo. De este modo, se tiene

$+3_{10} = 00011$

En muchas situaciones el número de bits se fija por la capacidad de los registros que almacenarán los números binarios, de manera que quizá tengan que agregarse ceros a fin de llenar el número solicitado de posiciones de bit.

(d) Comience por escribir +2 usando 5 bits:

+ 2 = 00010	
11101	complemento a 1
+ 1	se suma uno
11110	representación en complemento a 2 de −2

(e) Comience con +8:

+ 8 = 01000	
10111	complemento de cada bit
+ 1	se suma uno
11000	representación en complemento de 2 a −8

Negación La negación es la operación de convertir un número positivo a su equivalente negativo o un número negativo a su equivalente positivo. Cuando los números binarios con signo se presentan en el sistema complemento a 2, la negación se efectúa simplemente al hacer la operación de complemento a 2. Para ilustrarlo, comencemos con +9 cuya representación con signo es 01001. Si se le hace complemento a 2, se obtiene 10111. Está claro que este es un mero negativo, por que el bit de signo es un 1. En realidad, 10111 representa –9, que es el equivalente negativo del número con el que se empezó. Asimismo, se puede comenzar con la representación de –9, que es 10111. Si se emplea el complemento a 2, se obtiene 01001, que reconocemos como +9. Estos pasos se diagraman a continuación:

empezar con → 01001 = +9
complemento a 2 (negar) → 10111 = –9
volver a negar → 01001 = +9

Por tanto, se niega a un número binario con signo al someterlo a complemento a 2. Esta negación cambia el número a su equivalente del signo opuesto. En el ejemplo 6-1 utilizamos la negación en los pasos (d) y (e) para convertir números positivos a sus equivalentes negativos.

EJEMPLO 6-2

Cada uno de los siguientes números es un número binario con signo en el sistema complemento a 2. Determine el valor decimal en cada caso: **(a)** 01100, **(b)** 11010, **(c)** 10001.

Solución

(a) El bit de signos es 0, de modo que el número es *positivo*, los otros 4 bits representan la verdadera magnitud del número. Es decir, $1100_2 = 12_{10}$. De esta manera el número decimal es **+12.**

(b) El bit de signo de 11010 es un 1, de modo que sabemos que el número es *negativo*, pero no podemos conocer su magnitud. Podemos encontrar cuál es esta magnitud, negando (sacando el complemento a 2) el número para convertirlo en su equivalente positivo.

	11010	número negativo original
	00101	complemento a 1
+	1	se suma 1
	00110	(+6)

Como el resultado de la negación es 00110 = +6, el número original debe ser equivalente a –6.

(c) Siga el mismo procedimiento que en (b):

	10001	número negativo original
	01110	complemento a 1
+	1	se suma 1
	01111	(+15)

Así, 10001 = **–15.**

Caso especial de la representación en complemento a 2 Siempre que un número con signo tiene un 1 en el bit de signo y todos los bits de magnitud son ceros, su equivalente decimal es -2^N, donde N es el número de bits que hay en la *magnitud*. Por ejemplo,

$$1000 = -2^3 = -8$$
$$10000 = -2^4 = -16$$
$$100000 = -2^5 = -32$$

y así sucesivamente.

Así, podemos decir que el intervalo completo de valores que se puede representar en el sistema complemento a 2 que tiene N bits de magnitud es

$$-2^N \text{ a } + (2^N - 1)$$

En total, existen 2^{N+1} valores diferentes, *incluido* el 0.

Por ejemplo, la tabla 6-1 incluye una lista de todos los números con signo que pueden representarse con 4 bits utilizando para ello el sistema complemento a 2 (note que la secuencia inicia en $-2^N = -2^3 = -8_{10} = 1000_2$ y continúa hasta $+ (2^N - 1) = + 2^3 - 1 = + 7_{10} = 0111_2$, al sumar 0001 en cada paso como en los contadores ascendentes.

Tabla 6-1

Valor decimal	Representación binaria con signo mediante complemento a 2
$+7 = 2^3 - 1$	0111
+6	0110
+5	0101
+4	0100
+3	0011
+2	0010
+1	0001
0	0000
−1	1111
−2	1110
−3	1101
−4	1100
−5	1011
−6	1010
−7	1001
$-8 = -2^3$	1000

EJEMPLO 6-3

¿Cuál es el intervalo de números decimales *sin signo* que se puede representar con 8 bits?

Solución

Ya que no hay bit de signo, se pueden emplear los ocho bits para representar la magnitud. Por tanto, los valores se encuentran en el intervalo que abarca desde

$$00000000_2 = 0_{10}$$

hasta

$$11111111_2 = 255_{10}$$

Esto es un total de 256 valores.

EJEMPLO 6-4

¿Cuál es el intervalo de números decimales con *signo* que se puede representar con 8 bits?

Solución

El número negativo más grande es

$$10000000_2 = -2^7 - 128_{10}$$

El número positivo más grande es

$$01111111_2 = +2^7 - 1 = +127_{10}$$

Por tanto, el intervalo abarca de −128 a +127; existe un total de 256 valores diferentes, incluido el cero. En forma alterna, dado que hay 7 bits de magnitud ($N = 7$), entonces hay $2^{N+1} = 2^8 = 256$ valores diferentes.

EJEMPLO 6-5

Cierta computadora guarda en su memoria dos números con signo usando el sistema complemento a 2. Mientras ejecuta un programa, la computadora recibe instrucciones de cambiar el signo de cada número; esto es, cambiar + 31 a −31 y −12 a + 12 ¿Cómo hará esto?

$$00011111_2 = +31_{10}$$
$$11110100_2 = -12_{10}$$

Solución

Se puede cambiar el signo de un número realizando la operación complemento a 2 en *todo* el número, incluyendo el bit de signo. La circuitería de la computadora toma de la memoria el número con signo; calcula su complemento a 2 y coloca el resultado de regreso en la memoria.

PREGUNTAS DE REPASO

1. Represente cada uno de los siguientes valores como un número de cinco bits con signo en el sistema complemento a 2:
   (a) +13 (b) −7 (c) −16
2. Cada uno de los siguientes es un número binario con signo representado en el sistema de complemento a 2. Determine su equivalente decimal:
   (a) 10011 (b) 1000000 (c) 0111111
3. ¿Cuál es el intervalo de números decimales con signo que se puede representar con 12 bits (incluido el bit de signo)?
4. ¿Cuántos bits se requieren para representar los números decimales varían de − 50 a + 50?
5. ¿Cuál es el mayor número negativo decimal que se puede representar usando un total de 16 bits?
6. Realice la operación complemento a 2 en los siguientes números:
   (a) 10000 (b) 10000000 (c) 1000
7. Defina la operación de negación.

6-3 ADICIÓN EN EL SISTEMA COMPLEMENTO A 2

Ahora investigaremos cómo se realizan las operaciones de adición y sustracción en máquinas digitales que usan la representación en complemento a 2 para números negativos. En los diversos casos a ser considerados, es importante observar que el bit de signo de cada número se opera en la misma forma como los bits de magnitud.

Caso I: Dos números positivos. La adición de dos números positivos es bastante sencilla. Considere la suma de + 9 y + 4:

```
+9  →  0 1001    cosumando
+4  →  0 0100    sumando
       0 1101    suma = +13
       ↑
       └ bits de signo
```

Note que los bits de signo del **cosumando** y el **sumando** son 0 y el bit de signo de la suma es 0, lo que indica que la suma es positiva. Note asimismo que el cosumando y el sumando se forman con el mismo número de bits. Esto *siempre* debe llevarse a cabo en el sistema complemento a 2.

Caso II: Número positivo y número negativo menor. Considere la adición de + 9 y −4. Recuerde que el número −4 estará en su forma complemento a 2. De este modo, + 4 (00100) debe convertirse a −4 (11100)

```
          ┌ bits de signo
          ↓
+9  →  0 1001    cosumando
−4  →  1 1100    sumando
   X   0 0101
   ↑
   └──────────── este acarreo se descarta; el resultado es 00101 (suma = + 5)
```

En este caso, el bit de signo del sumando es 1. Observe que el bit de signo también participa en el proceso de adición. De hecho, se genera un acarreo en la última posición de la suma. *Este acarreo siempre se descarta*, de modo que la suma final es 00101, que es equivalente a + 5.

Caso III: Número positivo y número negativo mayor. Considere la adición de −9 y + 4:

```
−9  →  10111
+4  →  00100
       11011    suma = −5
       ↑
       └── bit de signo negativo
```

Aquí la suma tiene un bit de signo 1, lo que indica un número negativo. Como la suma es negativa, ésta se encuentra en su forma complemento a 2, de manera que los últimos cuatro bits, 1011, representan en realidad el complemento a 2 de la suma. Para determinar la verdadera magnitud de la suma, debemos tomar el complemento a 2 de 11011; el resultado es 00101 = +5. De este modo, 11011 representa el número −5.

Caso IV: Dos números negativos.

```
−9  →  10111
−4  →  11100
   X  10011
   ↑  ↑
   │  └── bit de signo
   └───── este acarreo se descarta; el resultado es 10011 (suma = −13)
```

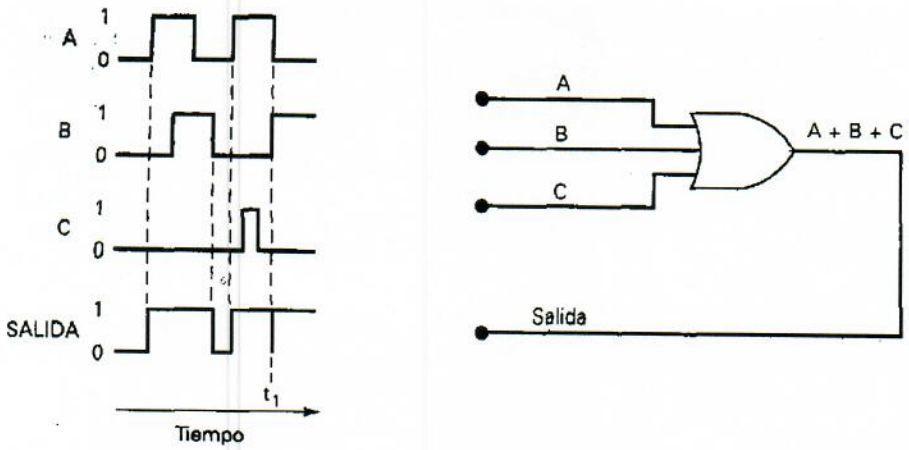


Figura 3-6 Ejemplo 3-3.

EJEMPLO 3-3A

Para el caso que se representa en la figura 3-6, determine la forma de onda en la salida de la compuerta OR.

Solución

Las tres entradas de la compuerta OR, *A*, *B* y *C* varían, como lo muestran sus diagramas de formas de onda. La salida de la compuerta OR se determina entendiendo que será alta cuando *cualquiera* de las tres entradas esté en un nivel alto. De acuerdo con este razonamiento, en la figura se muestra la onda de salida de la compuerta OR. Debe prestarse especial atención a lo que sucede en el tiempo t_1. El diagrama muestra que en este instante la entrada *A* pasa de alto a bajo, en tanto que la entrada *B* pasa de bajo a alto. Ya que estas entradas efectúan transiciones al mismo tiempo y debido a que se llevan cierto tiempo, hay un intervalo corto en el que estas entradas de la compuerta OR se encuentran en el intervalo indefinido entre 0 y 1. Cuando esto sucede, la salida de la compuerta OR es asimismo indefinida, como lo indica la transición falsa (glitch) o espiga en la onda de salida en t_1. La aparición de esta espiga y su magnitud (amplitud y anchura) dependen de la velocidad con que se efectúen las transiciones de entrada.

EJEMPLO 3-3B

¿Que ocurriría con la espiga en la salida en la figura 3-6 si la entrada *C* permanece en el estado ALTO mientras *A* y *B* cambian en t_1?

Solución

Con la entrada *C* en ALTO en t_1, la salida de la compuerta OR permanecerá en ALTO sin importar lo que ocurra en las demás entradas, ya que cualquier entrada que esté en ALTO hará que la salida de la compuerta OR se encuentre en ALTO. Por consiguiente, la espiga no aparecerá en la salida de la compuerta.

(b) Si la forma de onda de salida de una compuerta OR es siempre ALTA, entonces una de sus entradas se conserva permanentemente en ALTO.

3-5. ¿Cuántos conjuntos diferentes de condiciones de entrada producirán una salida ALTA de una compuerta OR de cinco entradas?

SECCIÓN 3-4

3-6. Cambie la compuerta OR en la figura 3-45 por una compuerta AND.
(a) Dibuje la forma de onda de salida.
(b) Dibuje la forma de onda de salida si la entrada *A* se mantiene permanentemente en el nivel más bajo.
(c) Trace la forma de onda de salida si *A* se mantiene permanentemente a + 5V.

3-7. Consulte la figura 3-4. Modifique el circuito de manera que la alarma se active solamente cuando la presión y la temperatura excedan sus límites máximos al mismo tiempo.

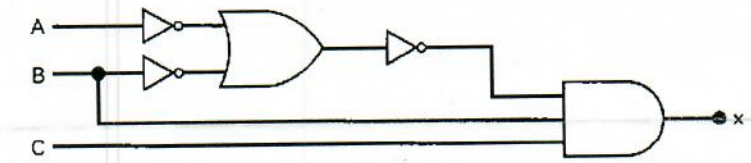
3-8. Cambie la compuerta OR de la figura 3-6 por una compuerta AND y trace la forma de onda de salida.

SECCIONES 3-5 a 3-7

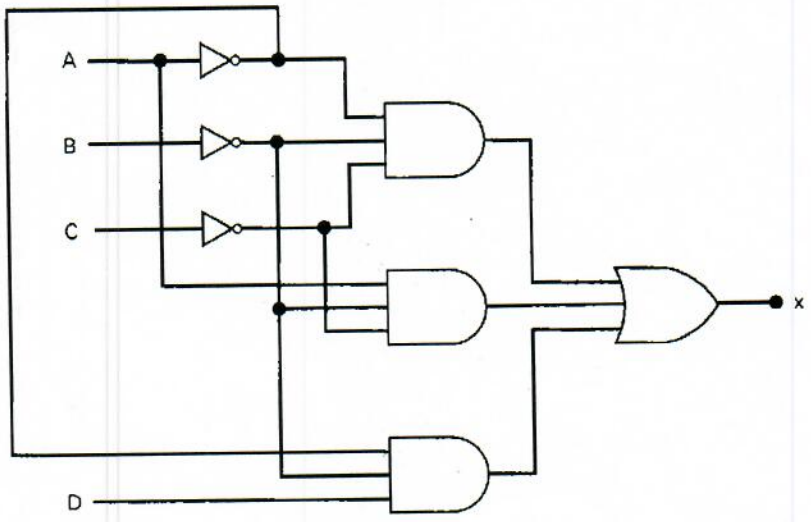
3-9. Agregue un INVERSOR a la salida de la compuerta OR de la figura 3-45. Dibuje la forma de onda en la salida del INVERSOR.

3-10. **(a)** Escriba una expresión booleana para la salida *x* de la figura 3-46(a). Determine el valor de *x* en todas las posibles condiciones de entrada y enlístelas en una tabla de verdad.

Figura 3-46



(a)



(b)

que la salida de una compuerta OR se torna ALTA cuando cualquier entrada es ALTA, la salida de la compuerta NOR pasa a BAJA cuando cualquier entrada es ALTA. Esta misma operación se puede aplicar a las compuertas NOR con más de dos entradas.

EJEMPLO 3-8

Determine la forma de onda en la salida de una compuerta NOR para las ondas de entrada que se muestran en la figura 3-20.

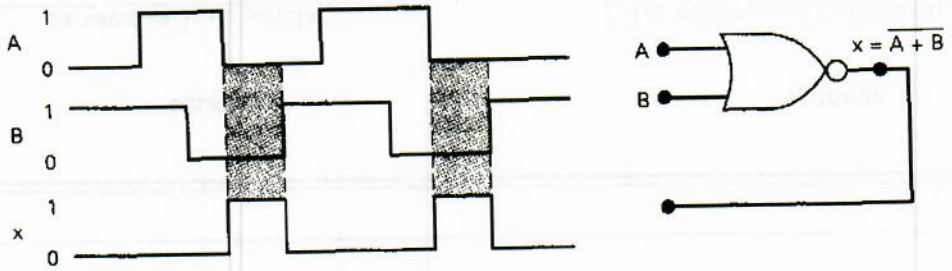


Figura 3-20 Ejemplo 3-8.

Solución

Existen varias maneras de determinar la onda de salida de la compuerta NOR. Una de ellas consiste primero en obtener la onda de salida de OR y luego invertirla (cambiar todos los unos por ceros y viceversa). Otra forma hace uso del hecho de que la salida de una compuerta NOR será ALTA *sólo* cuando todas las entradas sean BAJAS. Así, uno puede examinar las ondas de entrada, hallar aquellos intervalos donde todas sean BAJAS y hacer que la salida de la compuerta NOR sea ALTA en esos intervalos. La salida de la compuerta NOR será BAJA en todos los otros intervalos de tiempo. La onda de salida resultante se muestra en la figura.

EJEMPLO 3-9

Determine la expresión booleana para una compuerta NOR de tres entradas seguida de un INVERSOR.

Solución

Consulte la figura 3-21, donde se muestra el diagrama de circuito. La expresión en la salida de la compuerta NOR es $\overline{(A + B + C)}$, que luego se alimenta a través de un INVERSOR para producir

$$x = \overline{\overline{(A + B + C)}}$$

La presencia de los signos de inversión dobles indica que la cantidad $(A + B + C)$ ha sido invertida en dos ocasiones. Debe estar claro que esto simplemente produce la siguiente expresión $(A + B + C)$ sin ninguna alteración. Es decir,

$$x = \overline{\overline{(A + B + C)}} = A + B + C$$

Siempre que dos barras de inversión estén sobre la misma variable o cantidad, se cancelan una con otra, como en el ejemplo anterior. Sin embargo en casos como $\overline{\overline{A} + \overline{B}}$ las barras de

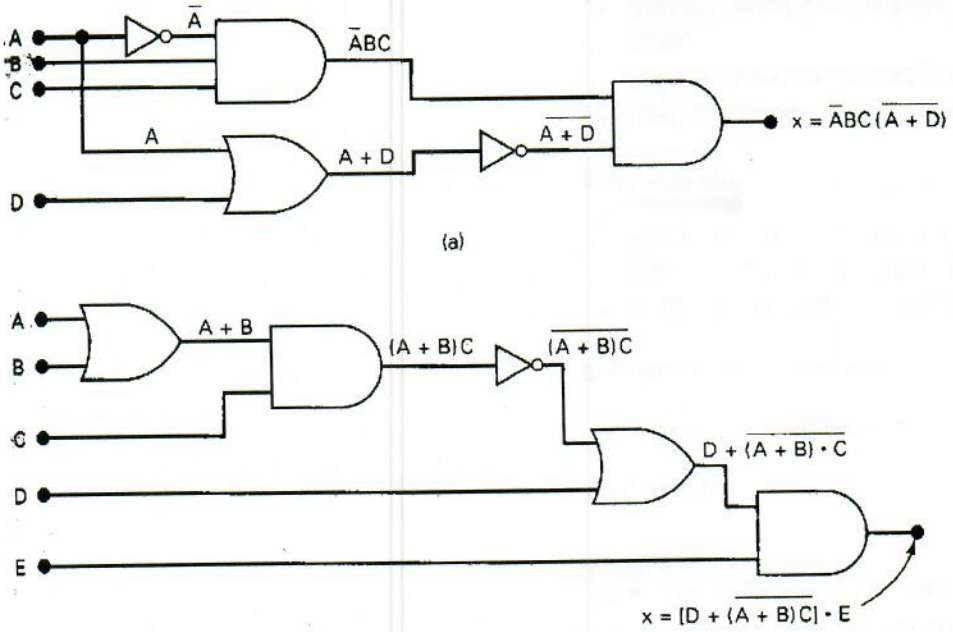


Figura 3-15 Más ejemplos.

PREGUNTA DE REPASO

1. En la figura 3-15(a) cambie cada compuerta AND por una compuerta OR y cambie la compuerta OR por una AND. Luego escriba la expresión para la salida x.

3-7 EVALUACIÓN DE LAS SALIDAS DE LOS CIRCUITOS LÓGICOS

Una vez que se obtiene la expresión booleana para la salida de un circuito, el nivel lógico de la salida se puede determinar para cualquier valor de las entradas del circuito. Por ejemplo, suponga que deseamos conocer el nivel lógico de la salida x para el circuito de la figura 3-15(a) en el caso donde $A = 0$, $B = 1$, $C = 1$ y $D = 1$. Como sucede en el álgebra ordinaria, el valor de x se puede determinar sustituyendo los valores de las variables en la expresión y efectuando las operaciones que se indican de la siguiente manera:

$$\begin{aligned} x &= \bar{A}BC(\overline{A + D}) \\ &= \bar{0} \cdot 1 \cdot 1\,(\overline{0 + 1}) \\ &= 1 \cdot 1 \cdot 1 \cdot (\overline{0 + 1}) \\ &= 1 \cdot 1 \cdot 1 \cdot (\bar{1}) \\ &= 1 \cdot 1 \cdot 1 \cdot 0 \\ &= 0 \end{aligned}$$

Este resultado final vuelve a ser negativo y está en forma complemento a 2 con un bit de signo 1. Al negar este resultado (al sacar su complemento a 2), produce 01101 = + 13.

Caso V: números iguales y opuestos.

$$\begin{array}{rcl} -9 & \rightarrow & 10111 \\ +9 & \rightarrow & 01001 \\ \hline 0 & & \not{1}\ 00000 \end{array}$$

↑——— Se descarta; el resultado es 00000 (suma = +0)

El resultado es obviamente + 0, como se esperaba.

PREGUNTAS DE REPASO

Para las siguientes preguntas, suponga que se emplea el sistema de complemento a 2.

1. *Cierto o falso:* Siempre que la suma de dos números binarios con signo tiene un bit de signo 1, la magnitud de la suma está en forma de complemento a 2.
2. Sume los siguientes pares de números con signo. Exprese la suma como un número binario con signo y como un número decimal:
   (a) 100111 + 111011 (b) 100111 + 011001

6-4 SUSTRACCIÓN EN EL SISTEMA COMPLEMENTO A 2

La operación de sustracción que utiliza el sistema complemento a 2 en realidad comprende la operación de adición y realmente no difiere de los varios casos que se consideraron en la sección 6-3. Cuando se resta un número binario (el **sustraendo**) de otro número binario (el **minuendo**), el procedimiento es el siguiente:

1. **Niegue el sustraendo.** Esto cambiará el sustraendo a su valor equivalente con signo contrario.
2. **Súmelo al minuendo.** El resultado de esta suma va a representar la diferencia entre el sustraendo y el minuendo.

Otra vez, igual que en todas las operaciones aritméticas de complemento a 2, es necesario que ambos números tengan el mismo número de bits en sus representaciones.

Consideremos el caso donde +4 se restará de + 9.

$$\begin{array}{rcl} \text{minuendo } (+9) & \rightarrow & 01001 \\ \text{sustraendo } (+4) & \rightarrow & 00100 \end{array}$$

Se niega el sustraendo para producir 11100, lo que representa −4. Ahora, sume esto al minuendo.

$$\begin{array}{rll} & 01001 & (+9) \\ + & 11100 & (-4) \\ \hline \not{1} & 00101 & (+5) \end{array}$$

↑——— se descarta; así que el resultado es 00101 = +5

Cuando el sustraendo se cambia por su complemento a 2, en realidad se convierte en −4, así que *sumamos* −4 y +9, que es lo mismo que restar + 4 de + 9. Este es el caso II que se muestra en sección 6-3. Por tanto, cualquier operación de sustracción en realidad se convierte en una de adición cuando se emplea el sistema complemento a 2. Esta característica del sistema

complemento a 2 lo ha convertido en el método que más se utiliza, ya que permite que la misma circuitería efectúe la adición y la sustracción.

El lector debe verificar los resultados de utilizar el procedimiento anterior en las siguientes restas: (a) +9 – (–4); (b) –9 – (+ 4); (c) –9 – (–4); (d) + 4 – (–4). Recuerde que cuando el resultado tiene un bit de signo 1, éste es negativo y está en forma complemento a 2.

Desborde aritmético En cada uno de los anteriores ejemplos de adición y sustracción, los números que se sumaron constan de un bit de signo y 4 bits de magnitud. Las respuestas también constan de un bit de signo y 4 bits de magnitud. Cualquier acarreo hacia la sexta posición de bit fue descartada. En todos los casos que se consideraron, la magnitud del resultado fue lo suficientemente pequeña como para caber en 4 bits. Veamos la suma de + 9 y + 8.

$$\begin{array}{rcl} +9 & \rightarrow & \boxed{0}\,|\,1001 \\ +8 & \rightarrow & 0\,|\,1000 \\ \hline & & \boxed{1}\,|\,0001 \end{array}$$

signo incorrecto ↑ ↑ magnitud incorrecta

El resultado tiene un bit de signo negativo, lo que es obviamente incorrecto. La respuesta debe ser +17, pero la magnitud 17 necesita más de 4 bits y, por tanto, *sobrepasa* la posición de bit de signo. Esta condición de desborde siempre produce un resultado incorrecto y se detecta al examinar el bit de signo del resultado y comparándolo con los bits de signo de los números que se suman. En una computadora, se utiliza un circuito especial para detectar cualquier condición de desborde y para señalar que la respuesta es errónea. Encontraremos un circuito de este tipo en uno de los problemas de final de capítulo.

PREGUNTAS DE REPASO

1. Realice la sustracción de los siguientes pares de números con signo utilizando el sistema complemento a 2. Exprese los resultados como números binarios con signo y como valores decimales: (a) 01001-11010, (b) 10010-10011.
2. ¿Cómo puede detectarse el desborde aritmético cuando se suman números con signo?

6-5 MULTIPLICACIÓN DE NÚMEROS BINARIOS

La multiplicación de números binarios se lleva a cabo de la misma forma que la multiplicación de números decimales. En realidad el proceso es más simple, ya que las cifras multiplicadoras son 0 o 1, de modo que siempre se multiplica por 0 o por 1 y no por otros dígitos. El siguiente ejemplo ilustra los números binarios sin signo.

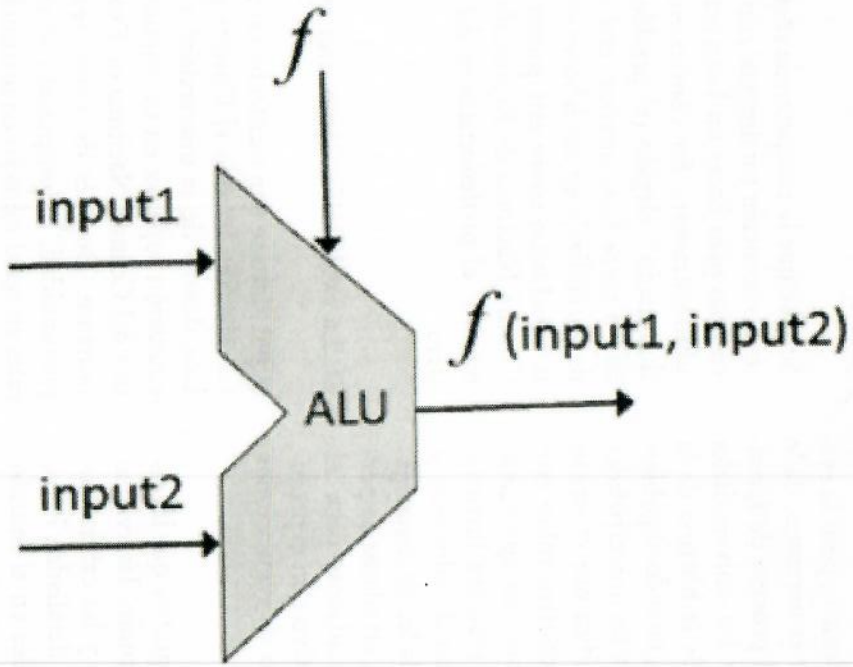
$$\begin{array}{rl} 1001 & \leftarrow \text{multiplicando} = 9_{10} \\ 1011 & \leftarrow \text{multiplicador} = 11_{10} \\ \hline 1001 & \\ 1001 & \text{productos parciales} \\ 0000 & \\ 1001 & \\ \hline 1100011 & \text{producto final} = 99_{10} \end{array}$$

From Nand To Tetris
www.nand2tetris.org

The Arithmetic Logic Unit

The ALU computes a function on the two inputs, and outputs the result

f: one out of a family of pre-defined arithmetic and logical functions

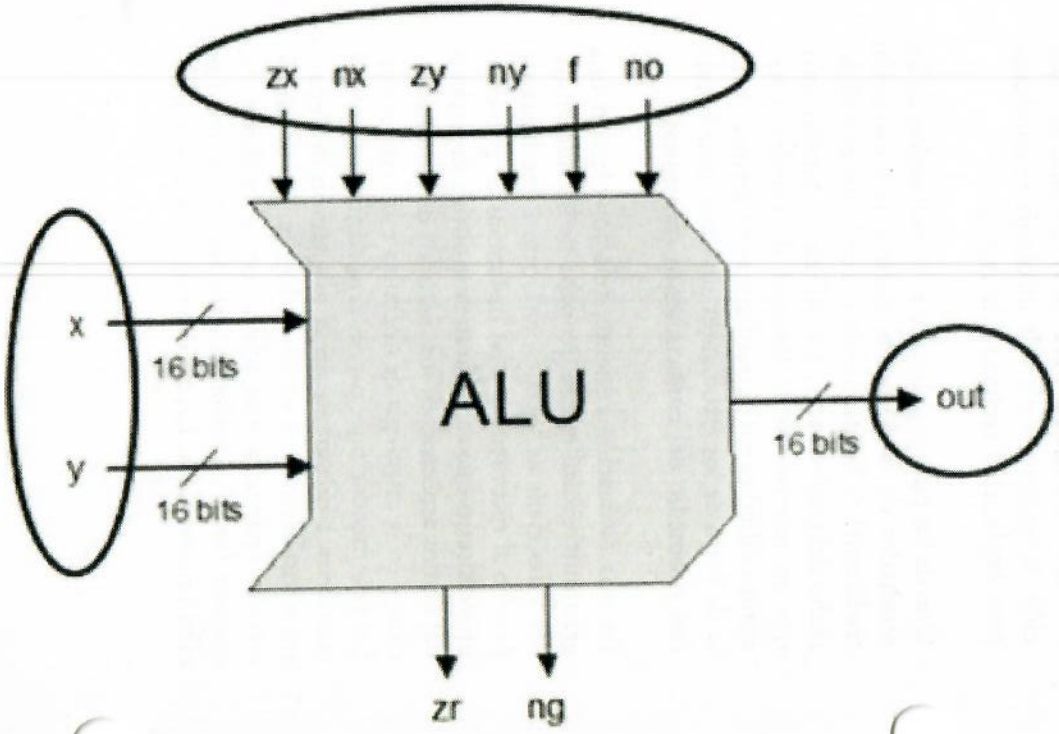


- Arithmetic operations: integer addition, multiplication, division, ...
- logical operations: And, Or, Xor, …

The Hack ALU

www.nand2tetris.org

- Operates on two 16-bit, two's complement values
- Outputs a 16-bit, two's complement value
- Which function to compute is set by six 1-bit inputs
- Computes one out of a family of 18 functions
- Also outputs two 1-bit values (to be discussed later).

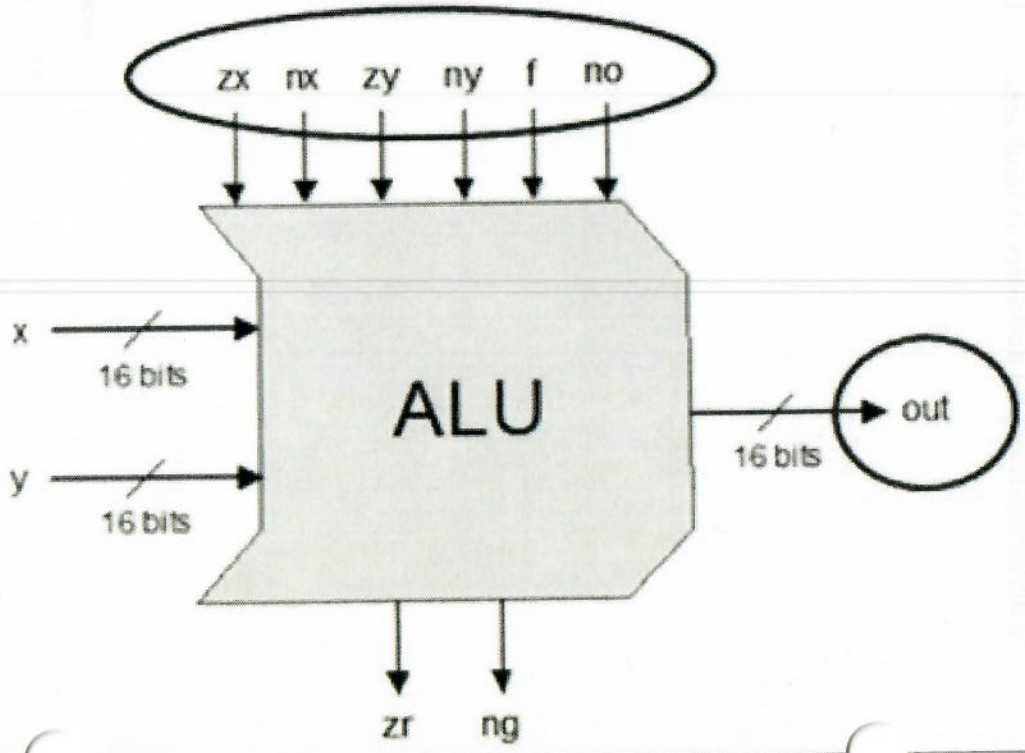


out
0
1
-1
x
y
!x
!y
-x
-y
x+1
y+1
x-1
y-1
x+y
x-y
y-x
x&y
x|y

The Hack ALU

www.nand2tetris.org

To cause the ALU to compute a function, set the control bits to the binary combination listed in the table.

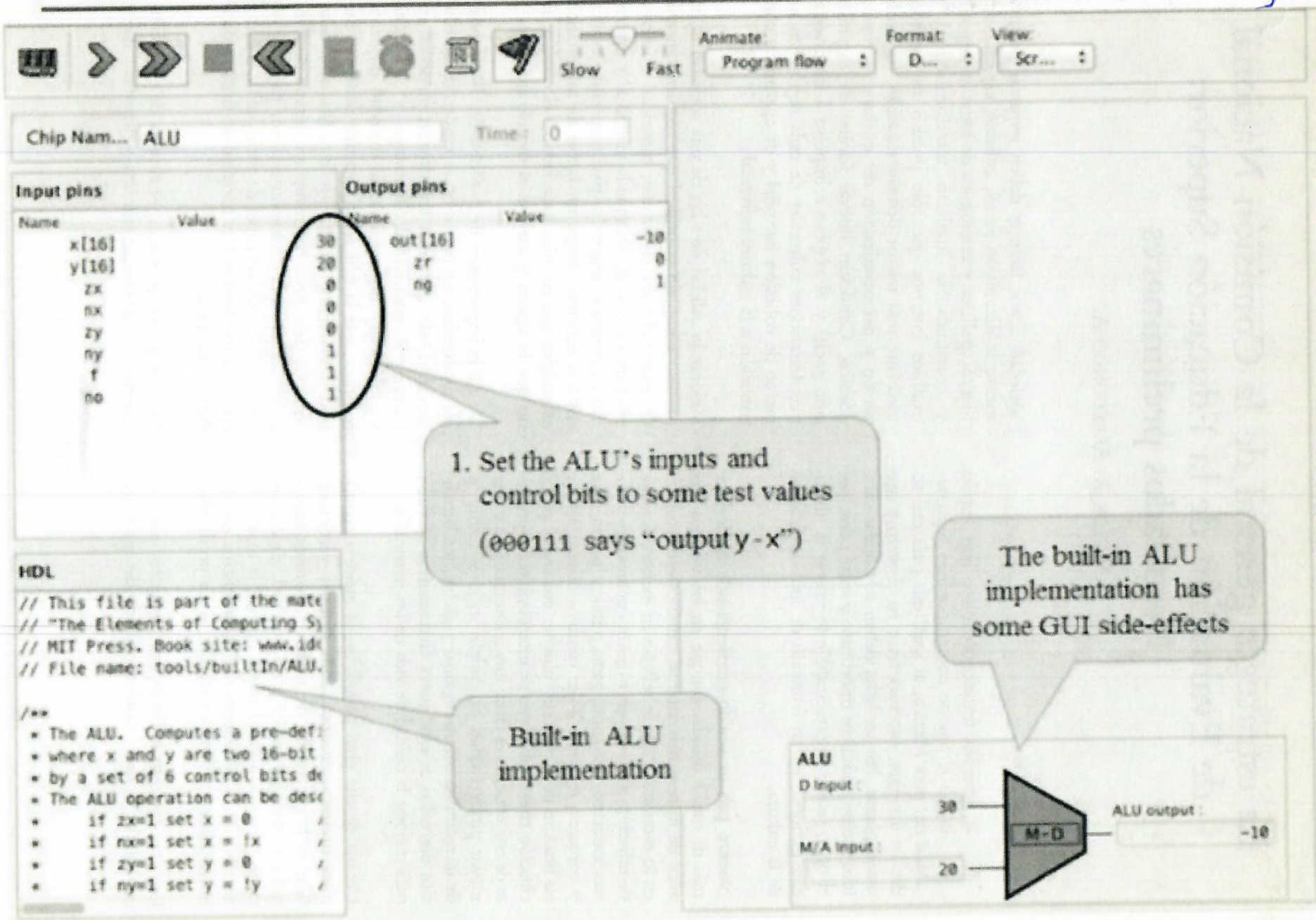


control bits

zx	nx	zy	ny	f	no	out
1	0	1	0	1	0	0
1	1	1	1	1	1	1
1	1	1	0	1	0	-1
0	0	1	1	0	0	x
1	1	0	0	0	0	y
0	0	1	1	0	1	!x
1	1	0	0	0	1	!y
0	0	1	1	1	1	-x
1	1	0	0	1	1	-y
0	1	1	1	1	1	x+1
1	1	0	1	1	1	y+1
0	0	1	1	1	0	x-1
1	1	0	0	1	0	y-1
0	0	0	0	1	0	x+y
0	1	0	0	1	1	x-y
0	0	0	1	1	1	y-x
0	0	0	0	0	0	x&y
0	1	0	1	0	1	x|y

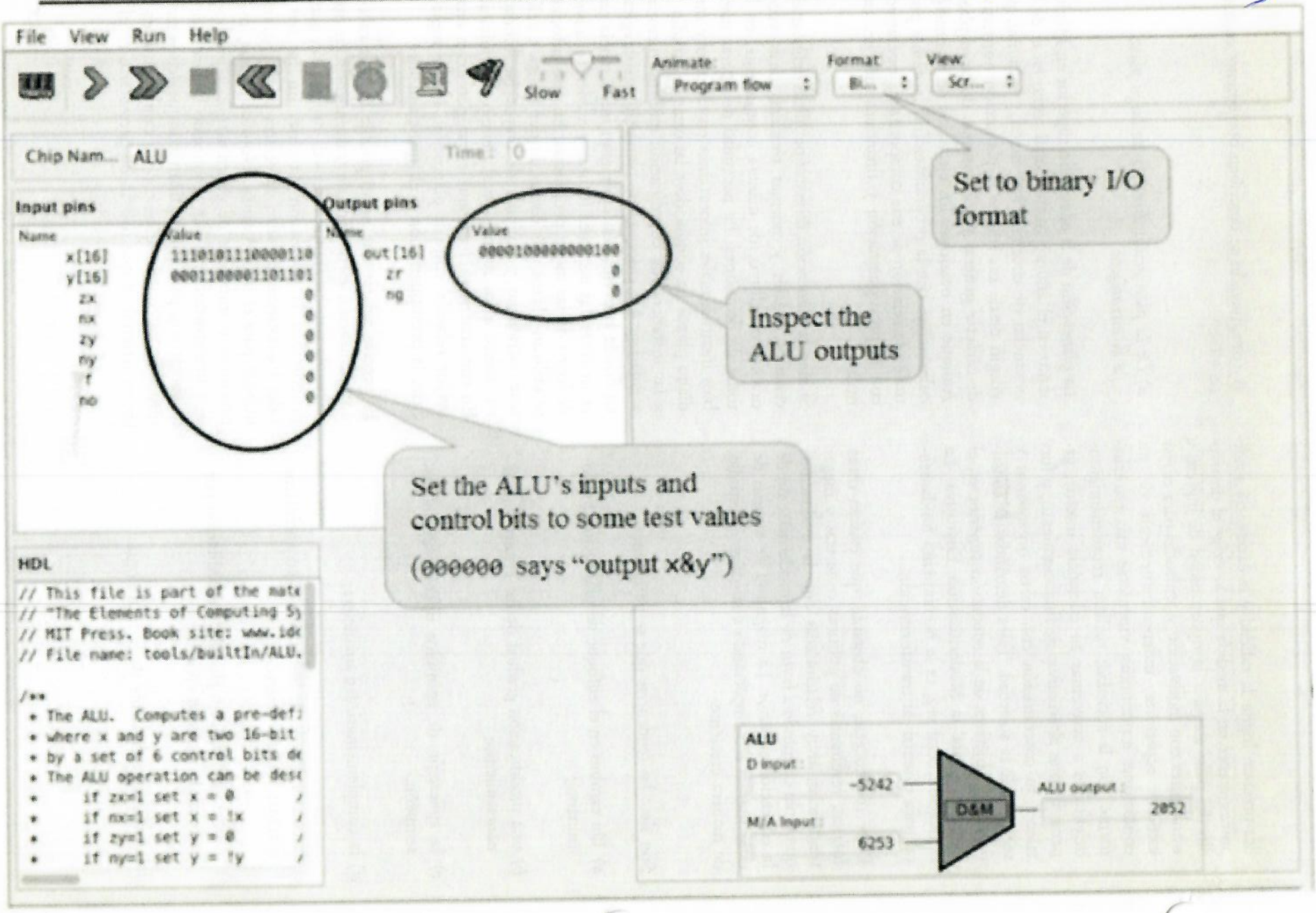
The Hack ALU in action: compute y-x

www.nand2tetris.org



The Hack ALU in action: compute x & y

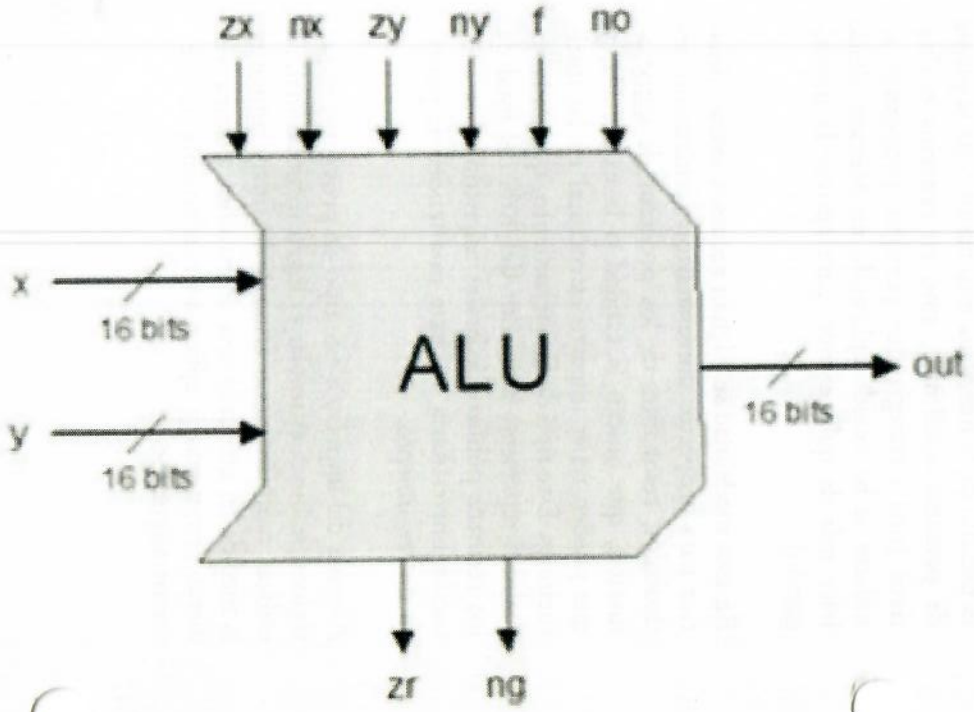
www.nandtetris.org



The Hack ALU operation

www.nand2tetris.org

pre-setting the x input		pre-setting the y input		selecting between computing + or &	post-setting the output	Resulting ALU output
zx	nx	zy	ny	f	no	out
if zx then x=0	if nx then x=!x	if zy then y=0	if ny then y=!y	if f then out=x+y else out=x&y	if no then out=!out	out(x,y)=



```
  1 1
  1101
  1001
 -----
 10110
```

7 − 5 = 2

−6 = 1100

10 − 4 = 6

X: 1001

Y: 0101

1001

Six Control bits

zx	nx
if zx then x=0	if nx then x=!x

If zx = 1, set the x input to 0.

If nx = 1, set the x input to not x (this is "Bitwise negation").

These two things happen one after the other

For example, if zx = 1 and nx = 1, first of all, we 0 the input and then we negate it.

zy	ny
if zy then y=0	if ny then y=!x

The same thing exactly happens with a y input using the zy and ny directives if you will

f

If f then out = x+y else out = x&y

If f = 1, x + y

If f = 0, we compute x and y.

no

If no then out = !out

- no b.t = 1, we negate the resulting output
- no b.t = 0, we leave it as is.

The Hack ALU operation

www.nand2tetris.org

pre-setting the x input		pre-setting the y input		selecting between computing + or &	post-setting the output	Resulting ALU output
zx	nx	zy	ny	f	no	out
if zx then x=0	if nx then x=!x	if zy then y=0	if ny then y=!y	if f then out=x+y else out=x&y	if no then out=!out	out(x,y)=
1	0	1	0	1	0	0
1	1	1	1	1	1	1
1	1	1	0	1	0	-1
0	0	1	1	0	0	x
1	1	0	0	0	0	y
0	0	1	1	0	1	!x
1	1	0	0	0	1	!y
0	0	1	1	1	1	-x
1	1	0	0	1	1	-y
0	1	1	1	1	1	x+1
1	1	0	1	1	1	y+1
0	0	1	1	1	0	x-1
1	1	0	0	1	0	y-1
0	0	0	0	1	0	x+y
0	1	0	0	1	1	x-y
0	0	0	1	1	1	y-x
0	0	0	0	0	0	x&y
0	1	0	1	0	1	x|y

ALU operation example: compute !x

www.nand2tetris.org

pre-setting the x input		pre-setting the y input		selecting between computing + or &	post-setting the output	Resulting ALU output
zx	nx	zy	ny	f	no	out
if zx then x=0	if nx then x=!x	if zy then y=0	if ny then y=!y	if f then out=x+y else out=x&y	if no then out=!out	out(x,y)=
1	0	1	0	1	0	0
1	1	1	1	1	1	1
1	1	1	0	1	0	-1
0	0	1	1	0	0	x
1	1	0	0	0	0	y
0	0	1	1	0	1	!x
1	1	0				!y
0	0	1				-x
1	1	0				-y
0	1	1				x+1
1	1	0				y+1
0	0	1				x-1
1	1	0				y-1
0	0	0				x+y
0	1	0				x-y
0	0	0				y-x
0	0	0				x&y
0	1	0				x|y

Example: compute !x

x: 1 1 0 0
y: 1 0 1 1

Following pre-setting:

x: 1 1 0 0
y: 1 1 1 1

Computation and post-setting:

x&y: 1 1 0 0
!(x&y): 0 0 1 1 (!x)

ALU operation example: compute y-x

www.nand2tetris.org

pre-setting the x input		pre-setting the y input		selecting between computing + or &	post-setting the output	Resulting ALU output
zx	nx	zy	ny	f	no	out
if zx then x=0	if nx then x=!x	if zy then y=0	if ny then y=!y	if f then out=x+y else out=x&y	if no then out=!out	out(x,y)=
1	0	1	0	1	0	0
1	1	1	1	1	1	1
				1	0	-1
				0	0	x
				0	0	y
				0	1	!x
				0	1	!y
				1	1	-x
				1	1	-y
				1	1	x+1
				1	1	y+1
				1	0	x-1
				1	0	y-1
				1	0	x+y
				1	1	x-y
0	0	0	1	1	1	y-x
0	0	0	0	0	0	x&y
0	1	0	1	0	1	x|y

Example: compute y-x

x: 0 0 1 0 (2)
y: 0 1 1 1 (7)

Following pre-setting:

x: 0 0 1 0
y: 1 0 0 0

Computation and post-setting:

x+y: 1 0 1 0
!(x+y): 0 1 0 1 (5)

There are 6 control bits, meaning that the ALU can compute 2^6 (=64) possible functions. The interface acknowledges only 18 of these functions, but in this question we ask about an unspecified combination, to see that you understand the internal logic.

What would be the resulting function of zx=1, nx=0, zy=1, ny=1, f=1, no=1?

- (●) 0

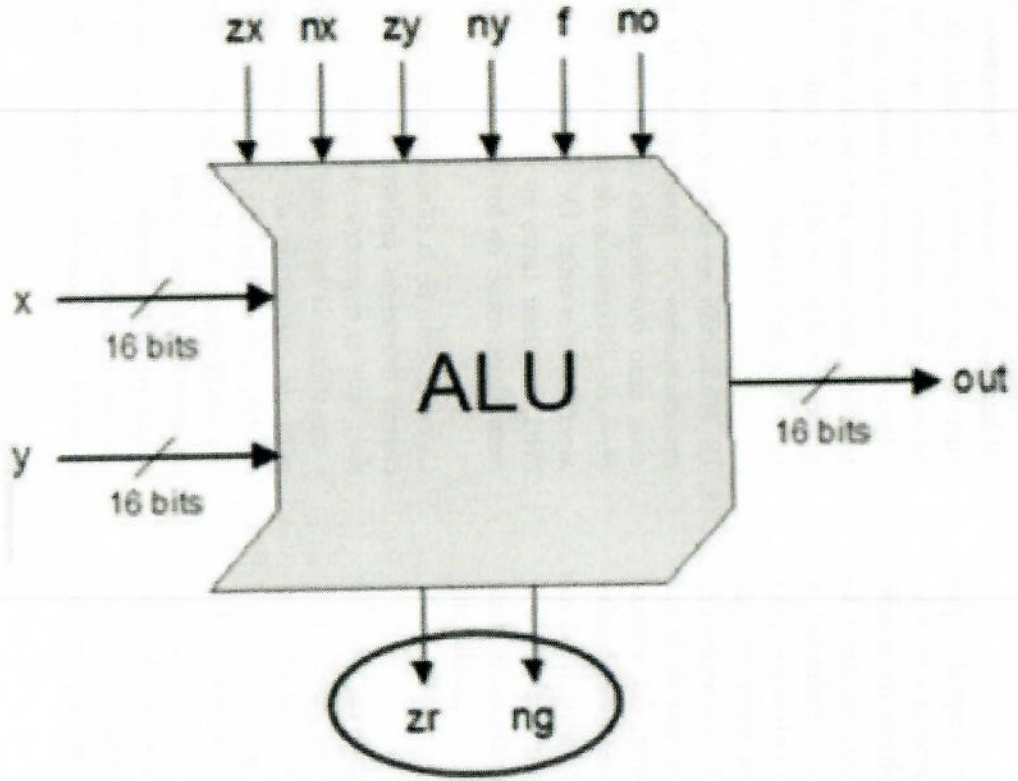
Correct

- ( ) 1
- ( ) -1
- ( ) x+y

Input: zx=1, nx=0, zy=1, ny=1, f=1, no=1
Let's examine what the ALU does.
1. zx=1, so x=0000000000000000
2. nx=0, so x stays the same.
3. zy=1, so y=0000000000000000
4. ny=1, so y=1111111111111111 (bitwise negation)
5. f=1, so out=1111111111111111 (addition)
6. no=1, so out=0000000000000000 (bitwise negation)

zx	nx	zy	ny	f	no
if zx then x=0	if nx then x=!x	if zy then y=0	if ny then y=!y	if f then out=x+y else out=x&y	if no then out=!out

The Hack ALU output control bits

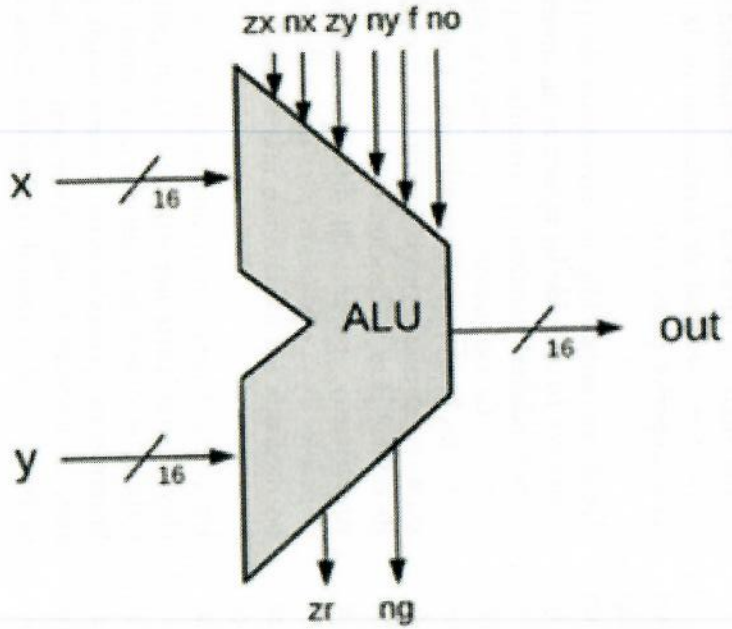



```
if out == 0 then zr = 1, else zr = 0

if out < 0  then ng = 1, else ng = 0
```

These two control bits will come into play when we build the complete computer's architecture.

ALU



ALU.hdl

```
/** The ALU. */
// Manipulates the x and y inputs as follows:
// if (zx  == 1) sets x = 0        // 16-bit true constant
// if (nx  == 1) sets x = !x       // bitwise Not
// if (zy  == 1) sets y = 0        // 16-bit true constant
// if (ny  == 1) sets y = !y       // bitwise Not
// if (f   == 1) sets out = x + y  // int. 2's-complement addition
// if (f   == 0) sets out = x & y  // bitwise And
// if (no  == 1) sets out = !out   // bitwise Not
// if (out == 0) sets zr = 1       // 1-bit true constant
// if (out < 0)  sets ng = 1       // 1-bit true constant
...
```

```
// This file is part of www.nand2tetris.org
// and the book "The Elements of Computing Systems"
// by Nisan and Schocken, MIT Press.
// File name: projects/02/ALU.hdl
/**
* The ALU (Arithmetic Logic Unit).
* Computes one of the following functions:
* x+y, x-y, y-x, 0, 1, -1, x, y, -x, -y, !x, !y,
* x+1, y+1, x-1, y-1, x&y, x|y on two 16-bit inputs,
* according to 6 input bits denoted zx,nx,zy,ny,f,no.
* In addition, the ALU computes two 1-bit outputs:
* if the ALU output == 0, zr is set to 1; otherwise zr is set to 0;
* if the ALU output < 0, ng is set to 1; otherwise ng is set to 0.
*/
// Implementation: the ALU logic manipulates the x and y inputs
// and operates on the resulting values, as follows:
// if (zx == 1) set x = 0 // 16-bit constant
// if (nx == 1) set x = !x // bitwise not
// if (zy == 1) set y = 0 // 16-bit constant
// if (ny == 1) set y = !y // bitwise not
// if (f == 1) set out = x + y // integer 2's complement addition
// if (f == 0) set out = x & y // bitwise and
// if (no == 1) set out = !out // bitwise not
// if (out == 0) set zr = 1
// if (out < 0) set ng = 1
CHIP ALU {
IN
x[16], y[16], // 16-bit inputs
```

zx, // zero the x input?
nx, // negate the x input?
zy, // zero the y input?
ny, // negate the y input?
f, // compute out = x + y (if 1) or x & y (if 0)
no; // negate the out output?

OUT

out[16], // 16-bit output
zr, // 1 if (out == 0), 0 otherwise
ng; // 1 if (out < 0), 0 otherwise

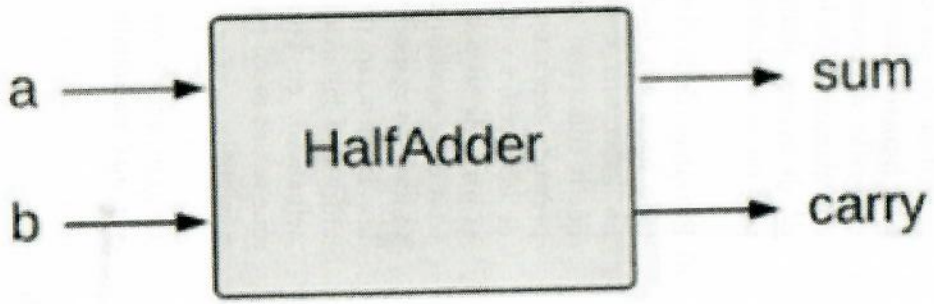
PARTS:

Goal: Build the following chips:

- HalfAdder
- FullAdder
- Add16
- Inc16
- ALU

A family of *combinational* chips, from simple adders to an Arithmetic Logic Unit.

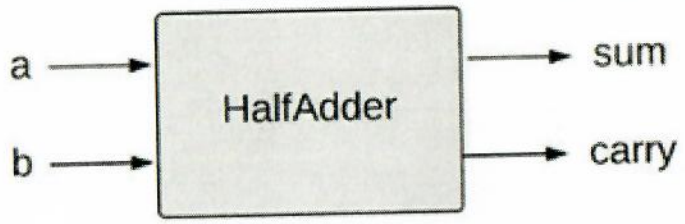
Half Adder



a	b	sum	carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

What two chips correspond to the sum and carry columns?

(If you can't see all the answers you should scroll down)

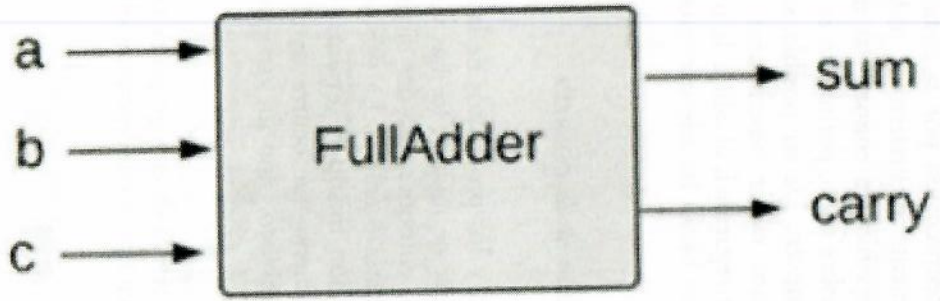


a	b	sum	carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

- sum: or, carry: and
- sum: xor, carry: or
- sum: and, carry: nand
- ◉ sum: xor, carry: and

Correct

Full Adder



a	b	c	sum	carry
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

Implementation tips

Can be built from two half-adders.

Week 3 / Unit 3.4

Counters

Noam Nisan and Shimon Schocken

Where counters come to play

- The computer must keep track of which instruction should be fetched and executed next
- This control mechanism can be realized by a Program Counter
- The PC contains the address of the instruction that will be fetched and executed next
- Three possible control settings:

Reset: fetch the first instruction — `PC = 0`

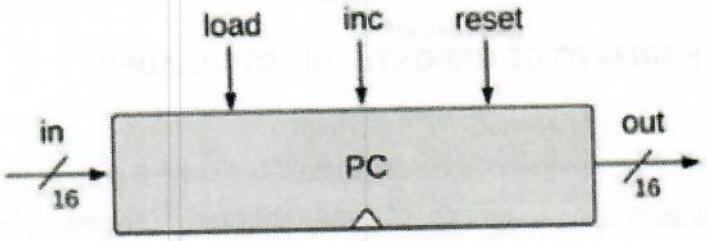
Next: fetch the next instruction — `PC++`

Goto: fetch instruction *n* — `PC = n`

Counter:

A chip that realizes this abstraction.

Counter abstraction



PC.hdl

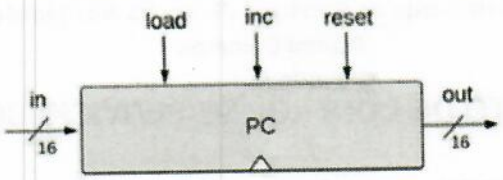
```
/**
 * A 16-bit counter with load, inc, and reset control bits.
 *
 * if (reset[t]==1) out[t+1] = 0                    // resetting: counter = 0
```

If the reset bit equals one,
if the reset bit is asserted,

well in that case,
I want the counter to emit zero.

So, in the next cycle,
the counter will emit the number zero.

Counter abstraction



PC.hdl

```
/**
 * A 16-bit counter with load, inc, and reset control bits.
 *
 * if (reset[t]==1) out[t+1] = 0                    // resetting: counter = 0
 * else if (load[t]==1) out[t+1] = in[t]            // setting counter = value
```

If the load bit is asserted,

well in that case I want to set the counter to a particular value.

So if I want the counter to go to number 17,

well, I put the number 17 or in binary.

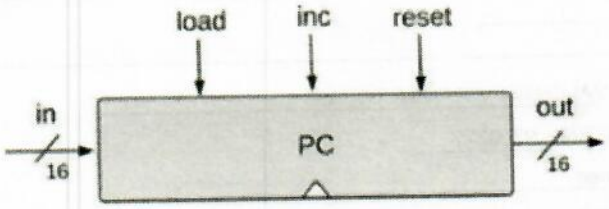
I put the 16-bit value that represents 17 in the input, and I assert the load bit.

This will bypass the regular increment of the counter and

it will set the counter to the number 17 in the next cycle.

And if inc equals one, if the inc bit is asserted, well, in that case,

Counter abstraction



PC.hdl

```
/**
 * A 16-bit counter with load, inc, and reset control bits.
 *
 * if (reset[t]==1) out[t+1] = 0                    // resetting: counter = 0
 * else if (load[t]==1) out[t+1] = in[t]            // setting counter = value
 *      else if (inc[t]==1) out[t+1] = out[t] + 1   // incrementing: counter++
```

And if inc equals one, if the inc bit is asserted, well, in that case,

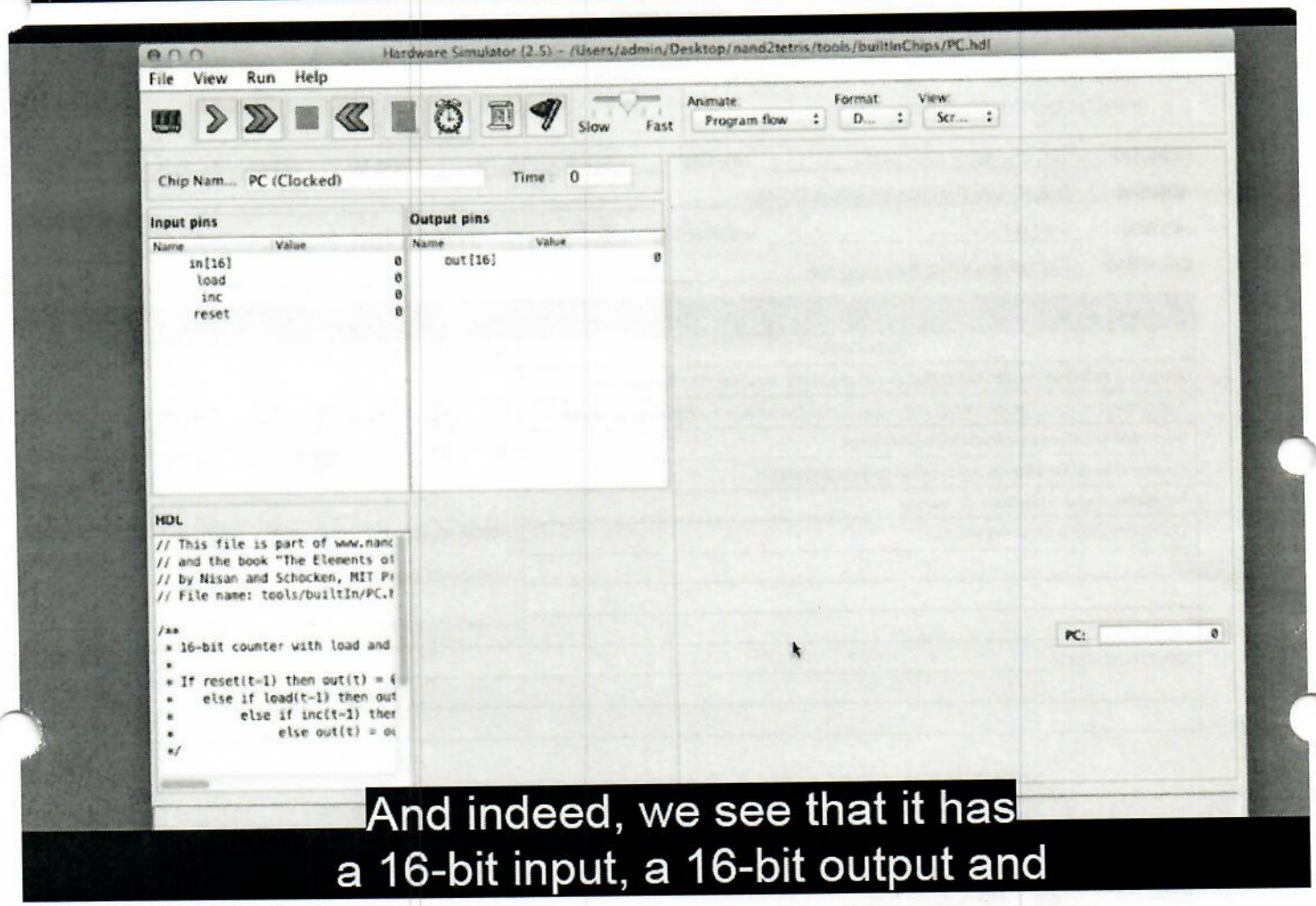
the output of the counter will be the current state of the counter, plus one.

This is a default operation of the counter.

And finally, if none of these control bits is asserted,

the counter does nothing, it emits its current state.

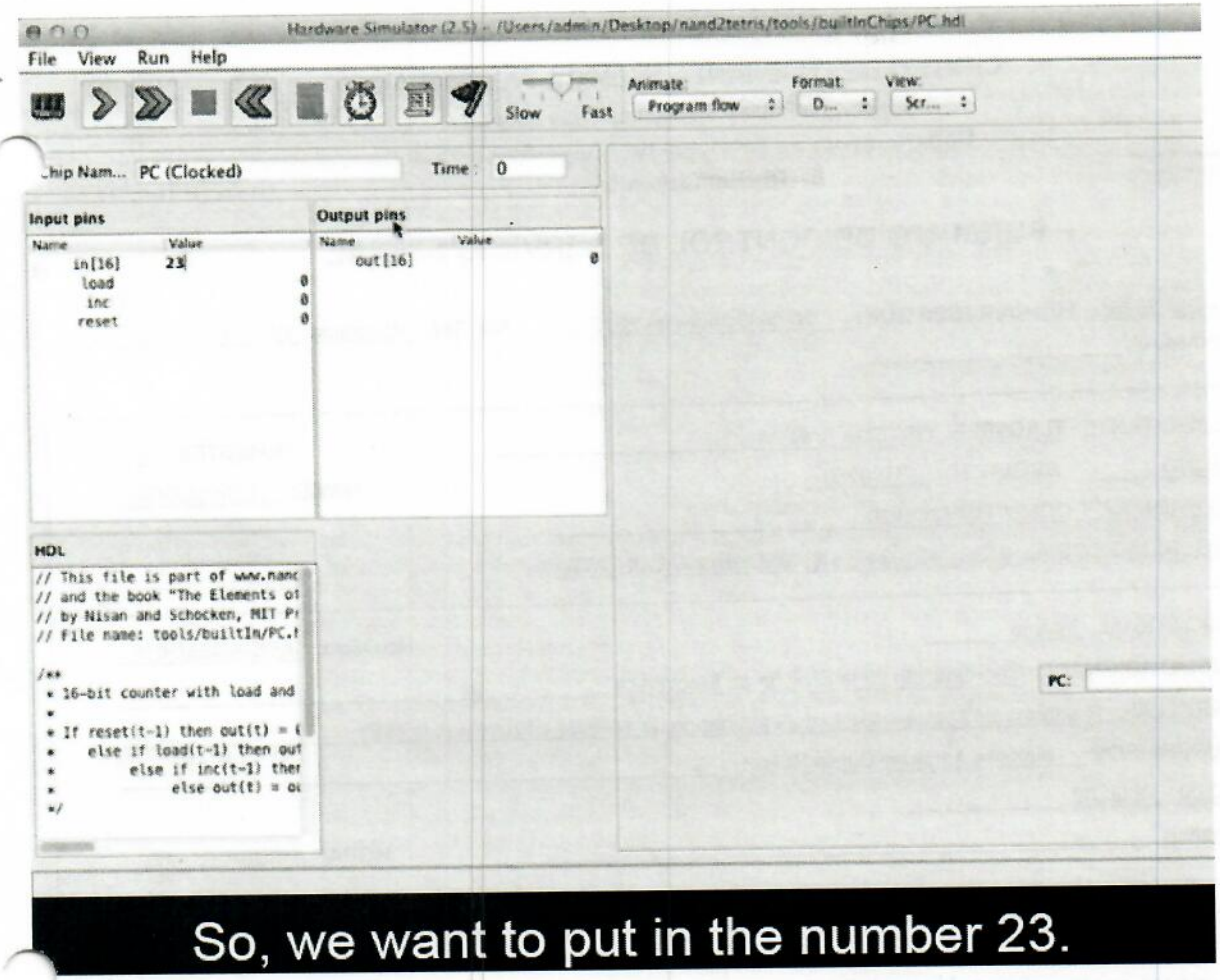
So, this is the desired functionality of the counter and



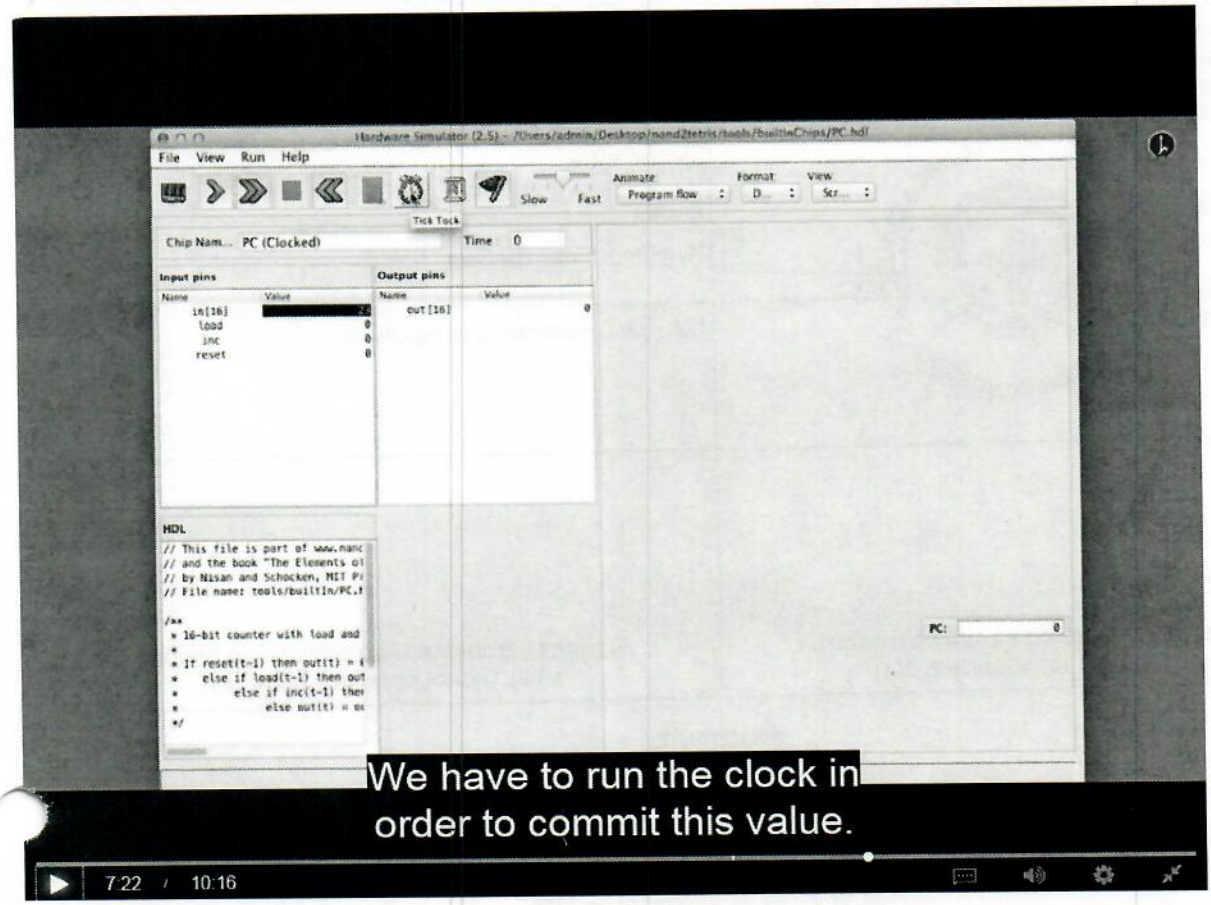
And indeed, we see that it has a 16-bit input, a 16-bit output and

So, let us load some value into the program counter.

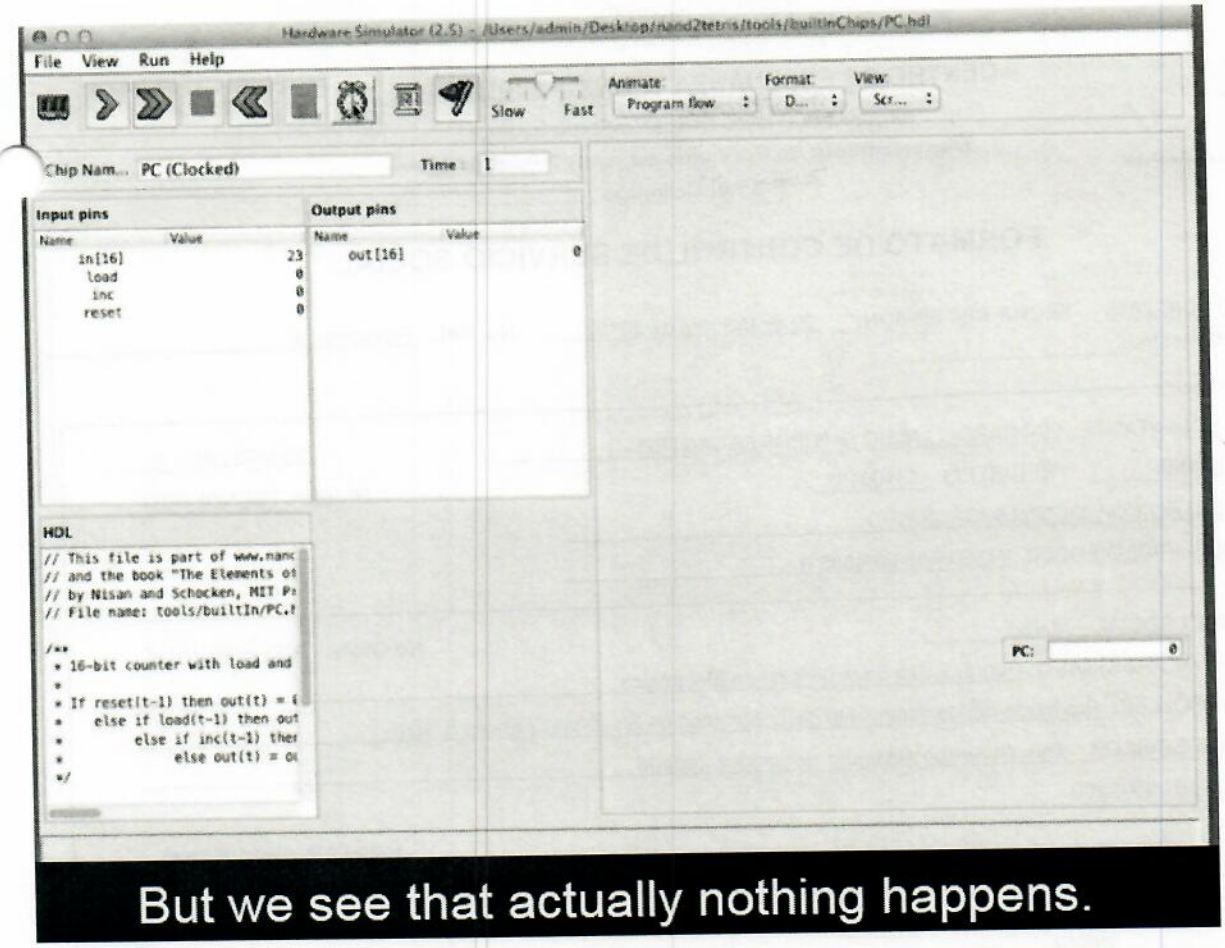
Conta dores 5/14



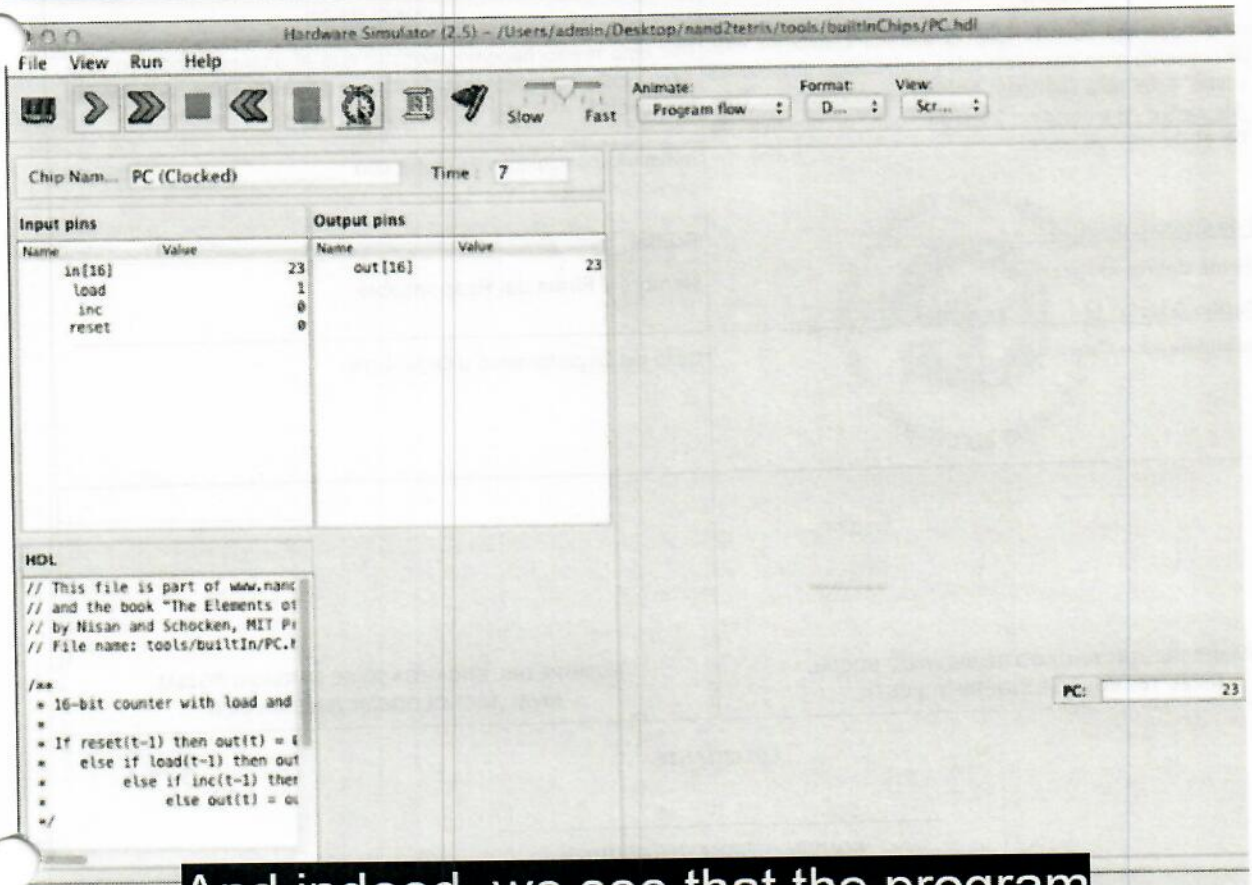
So, we want to put in the number 23.



Contadores 6/14

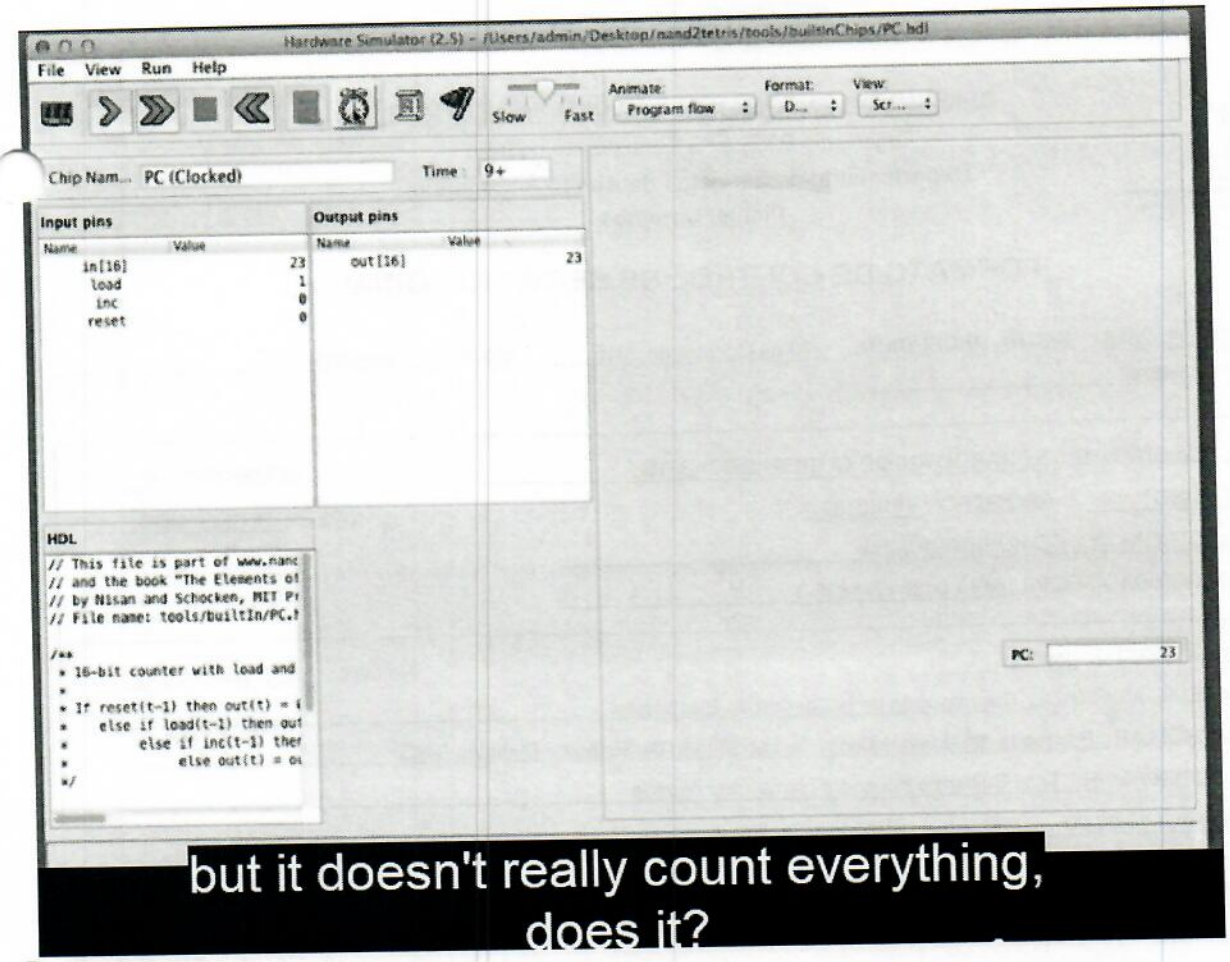


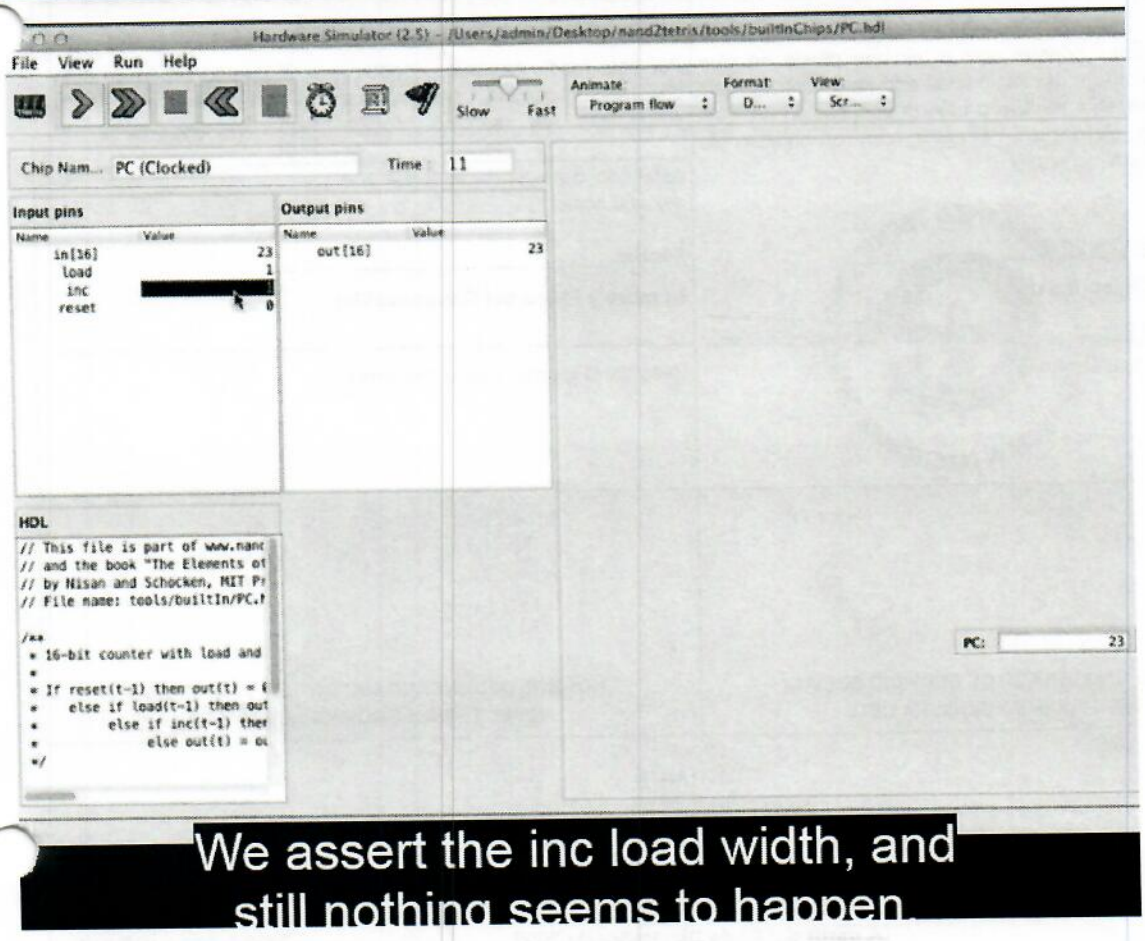
But we see that actually nothing happens.



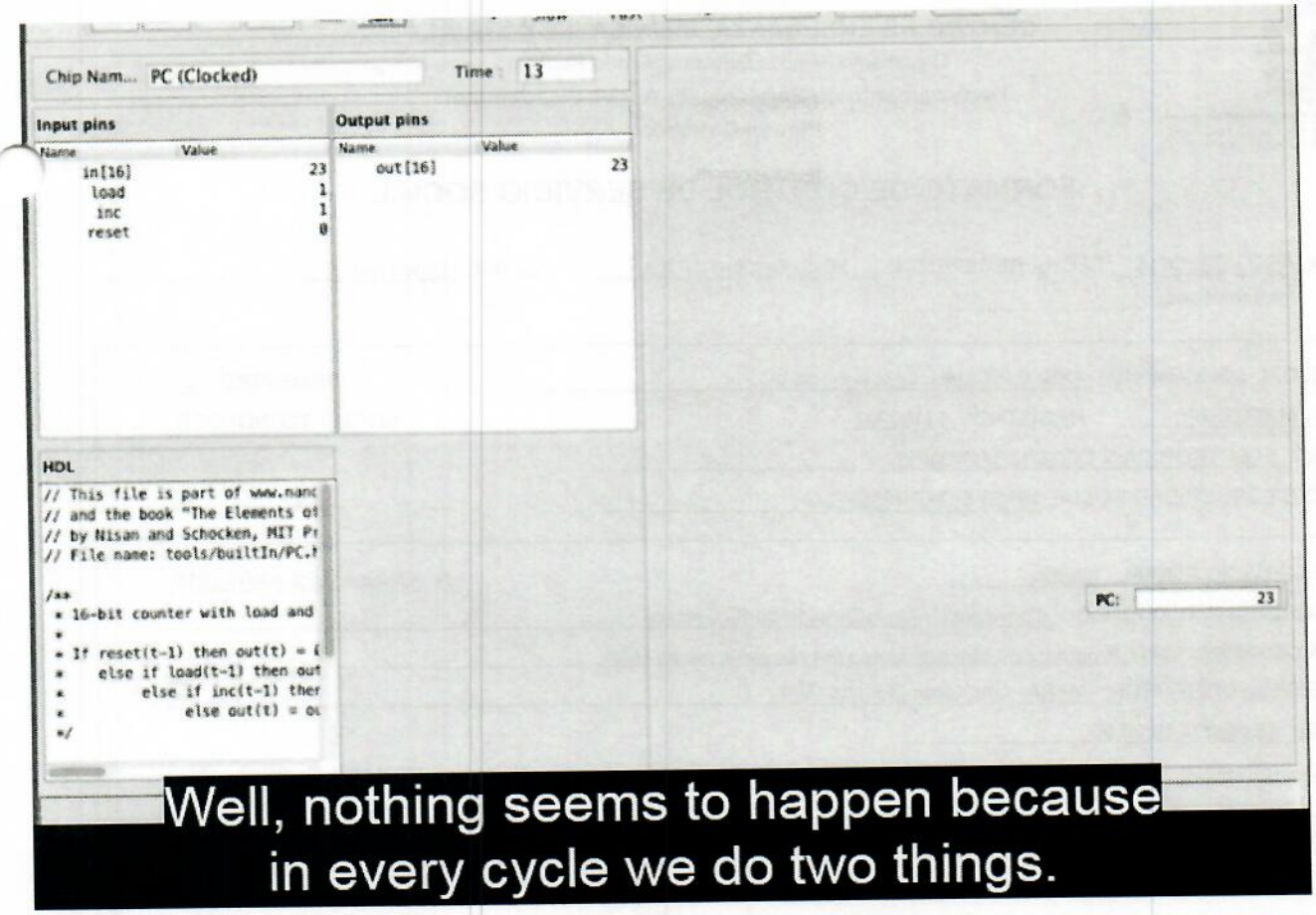
And indeed, we see that the program counter now contains 23,

Contadores 7/14

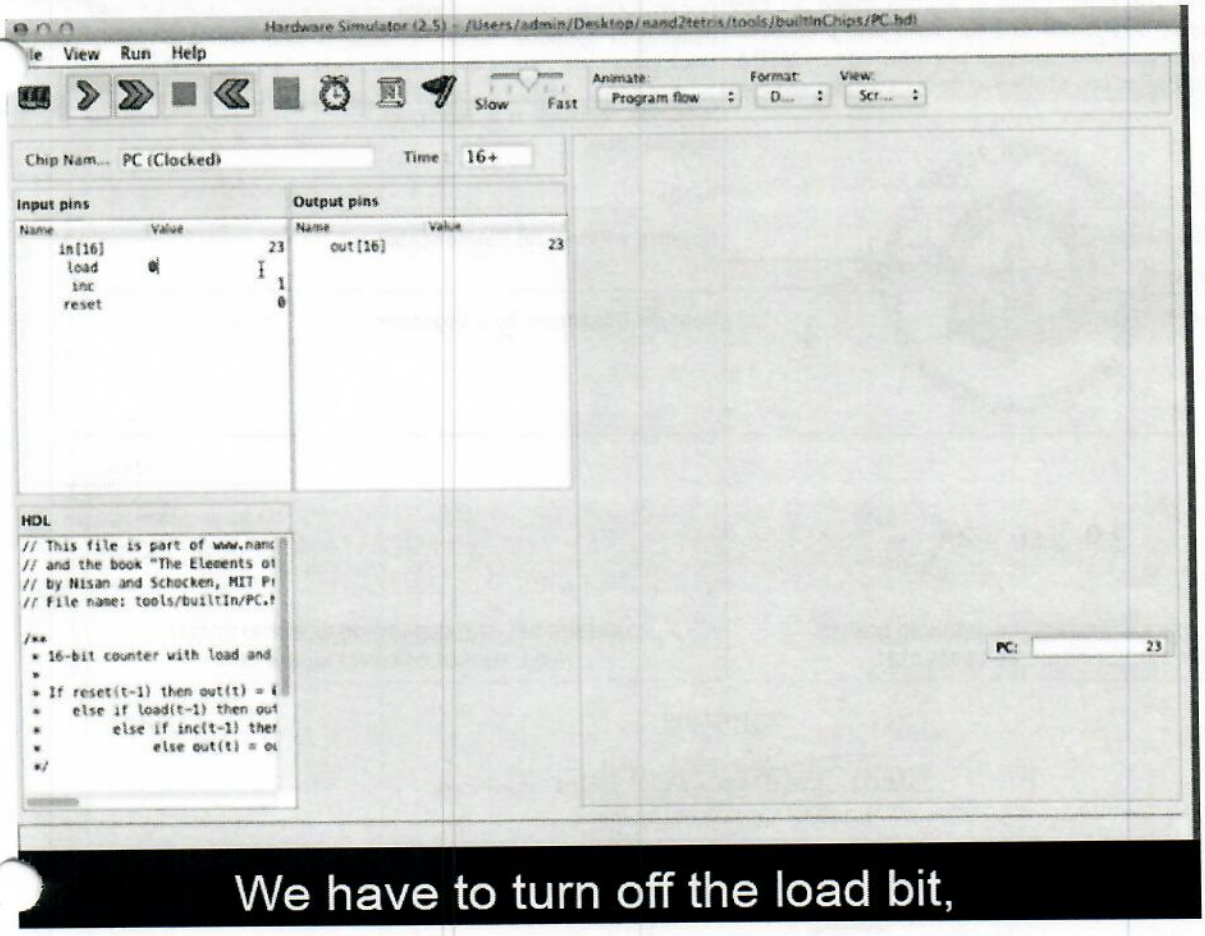




Contadores 8/14



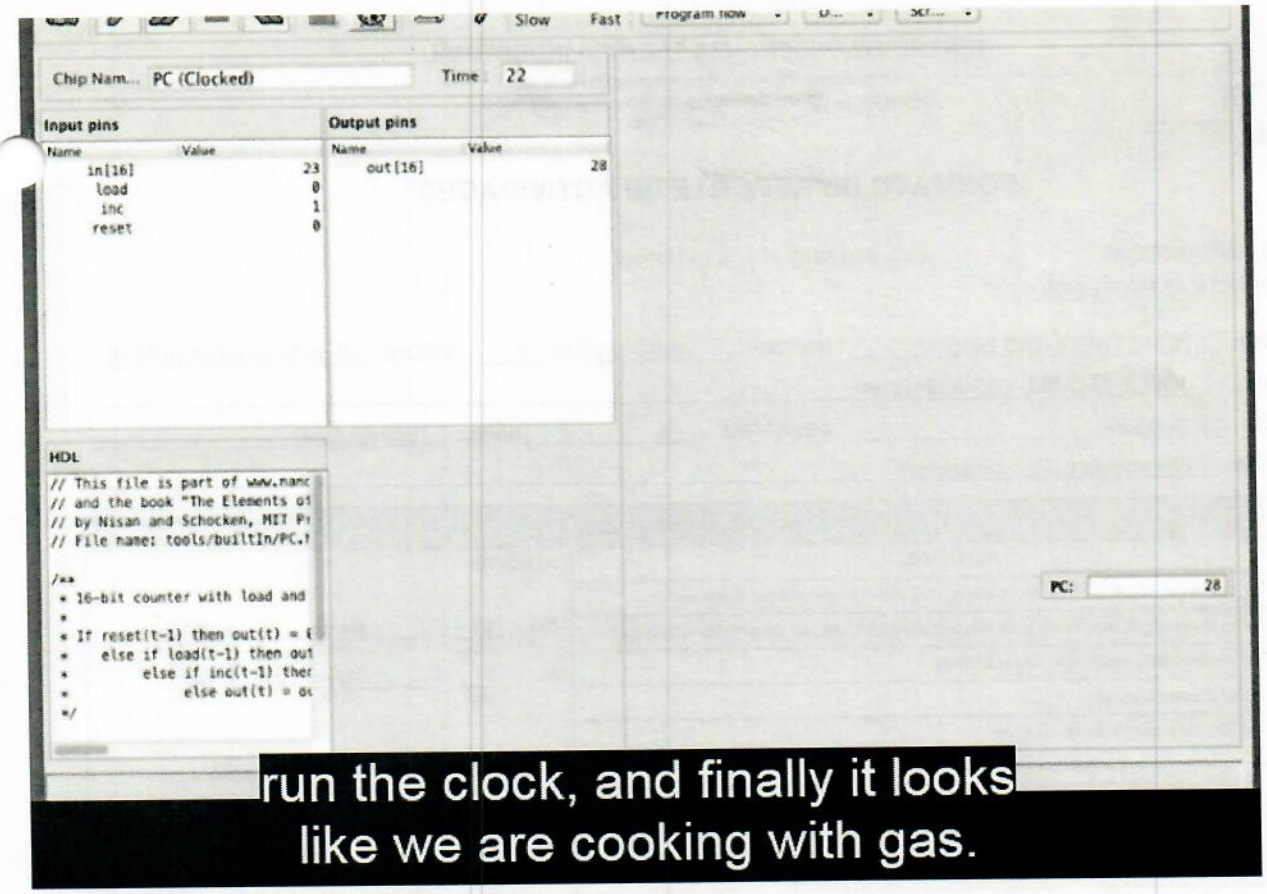
Well, nothing seems to happen because in every cycle we do two things.



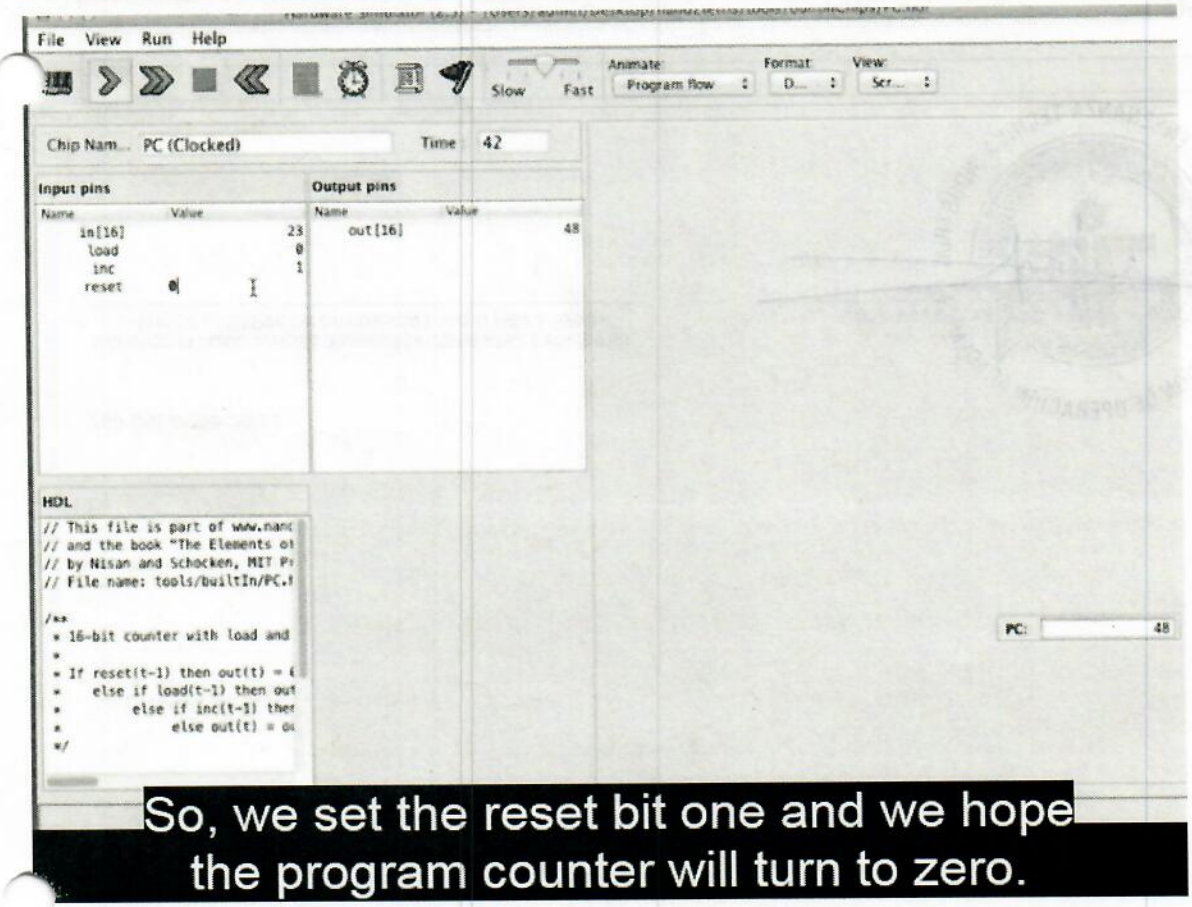
We have to turn off the load bit,

Contadores

9/14

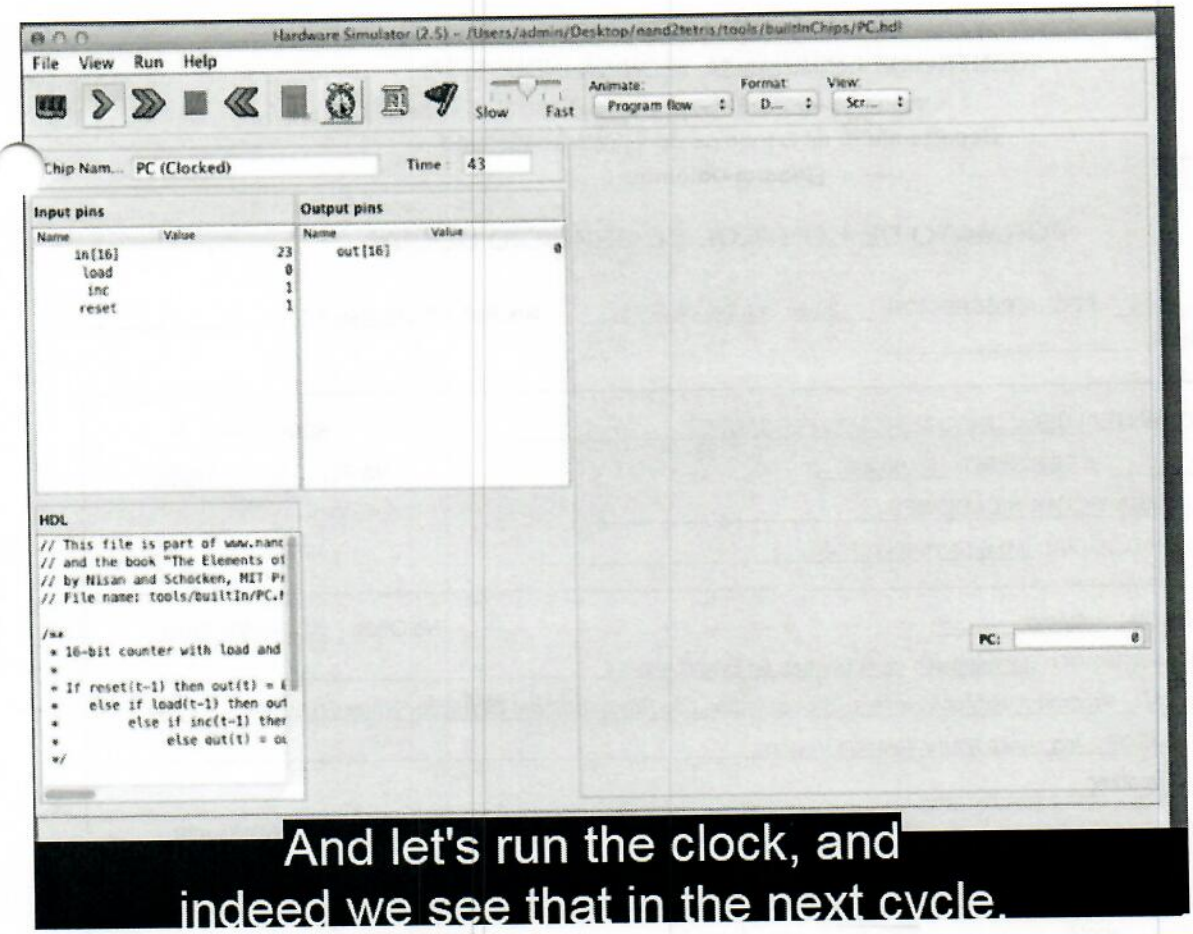


run the clock, and finally it looks like we are cooking with gas.

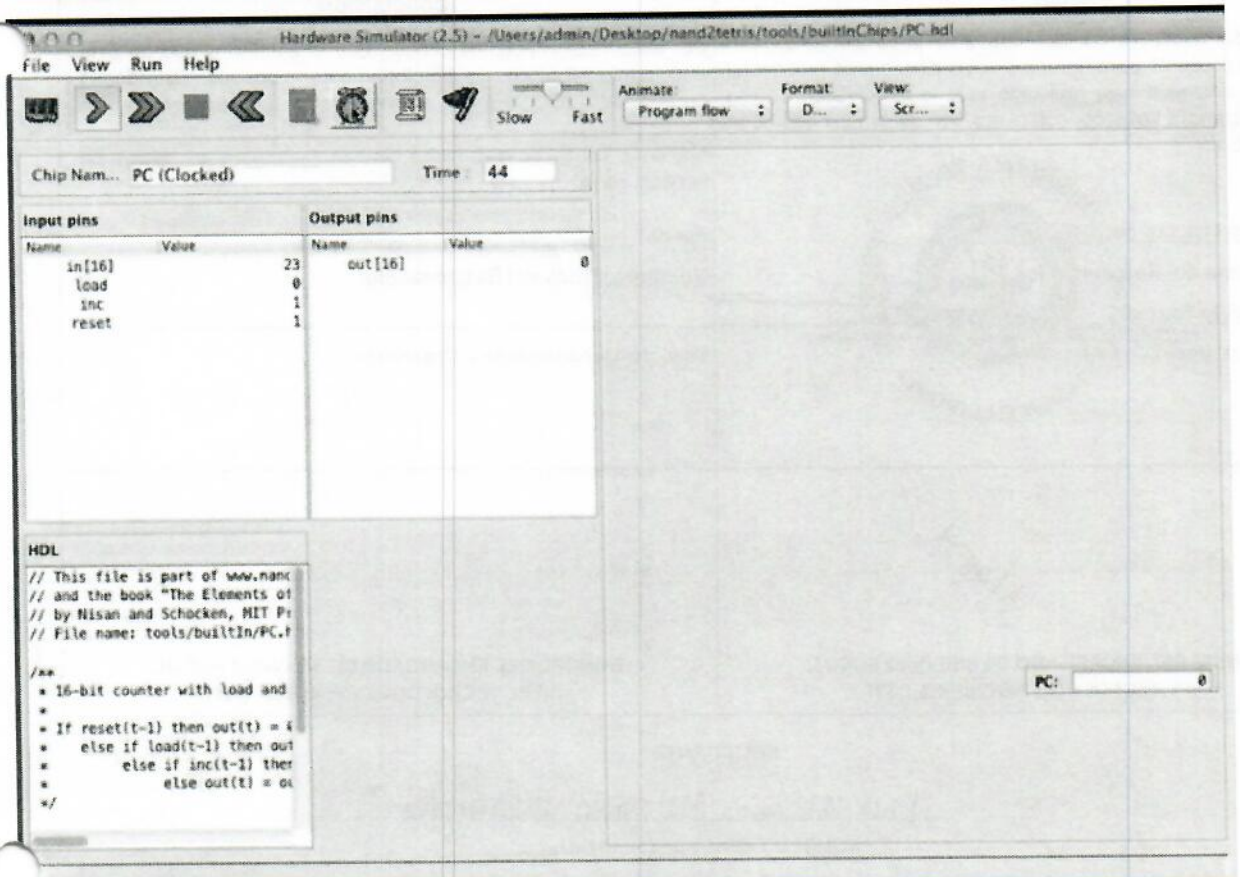


So, we set the reset bit one and we hope the program counter will turn to zero.

Contadores

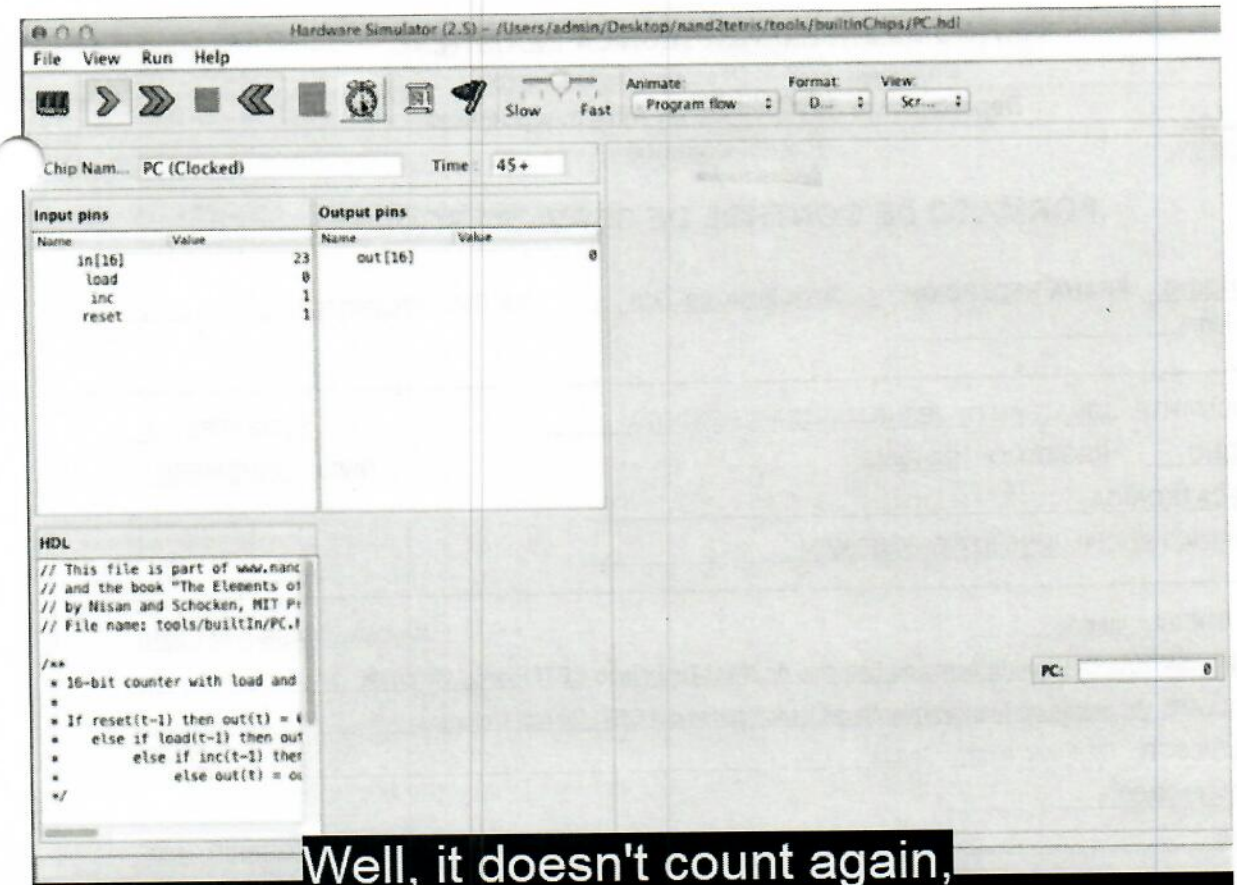


And let's run the clock, and indeed we see that in the next cycle.

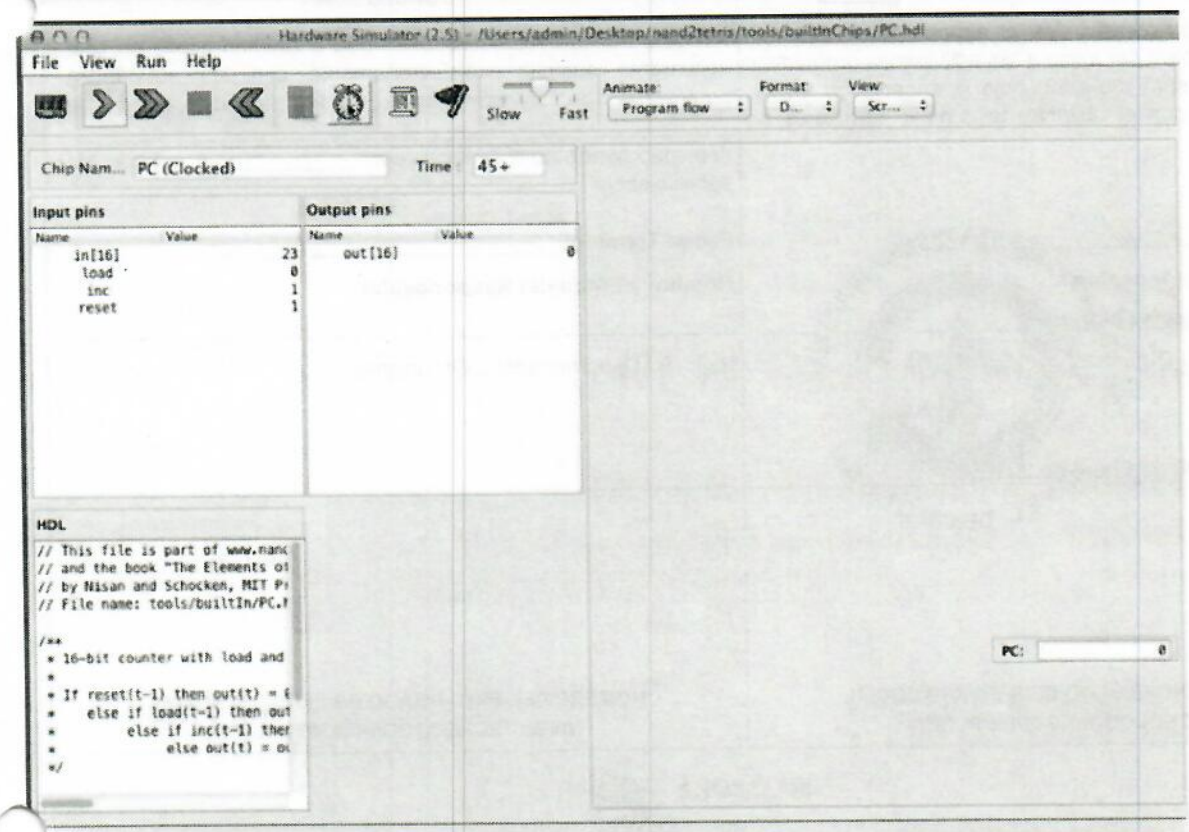


the program counter is zero.

Contadores 11/14



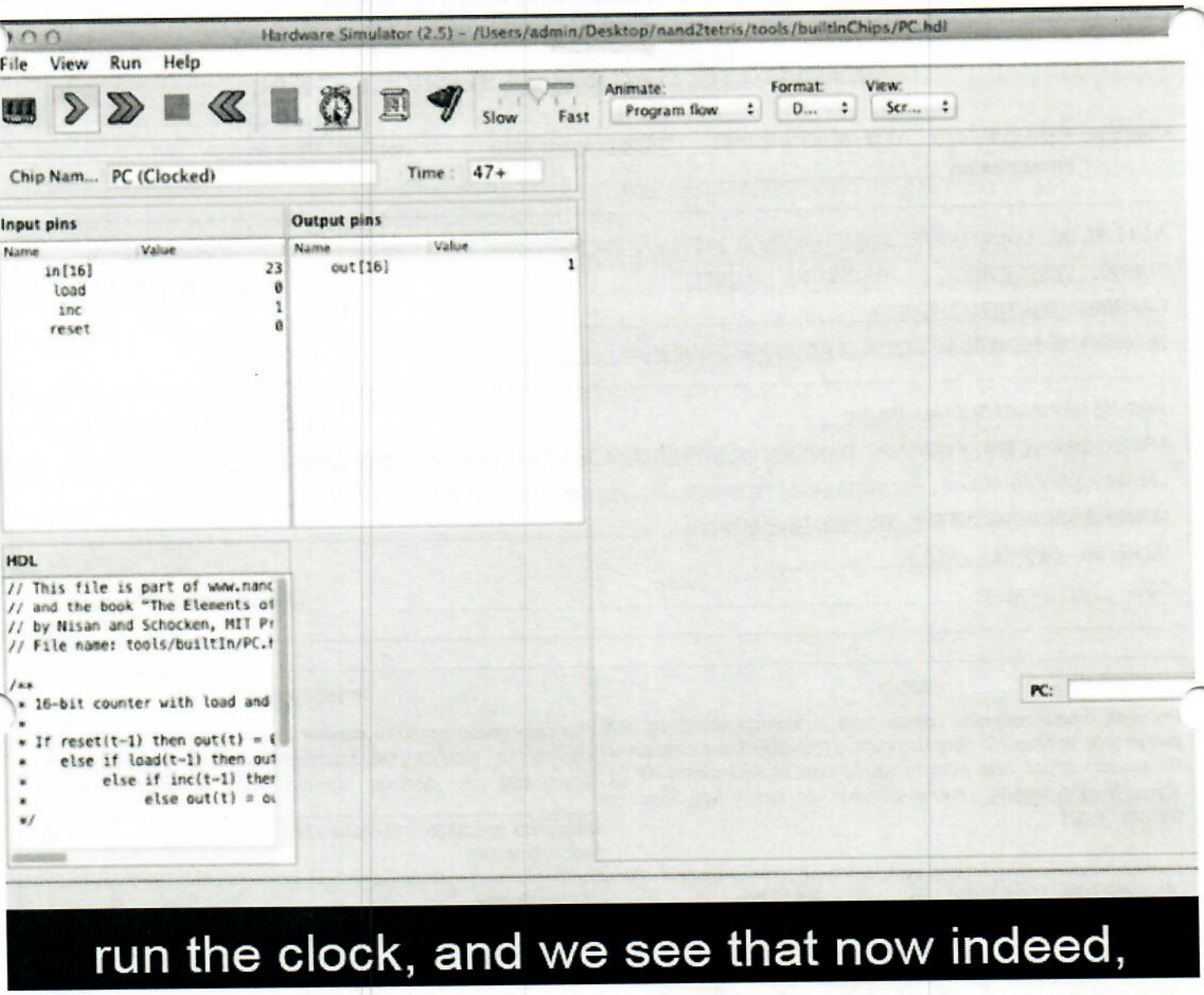
Well, it doesn't count again, because reset is still one.



So we have to turn off the reset bit and

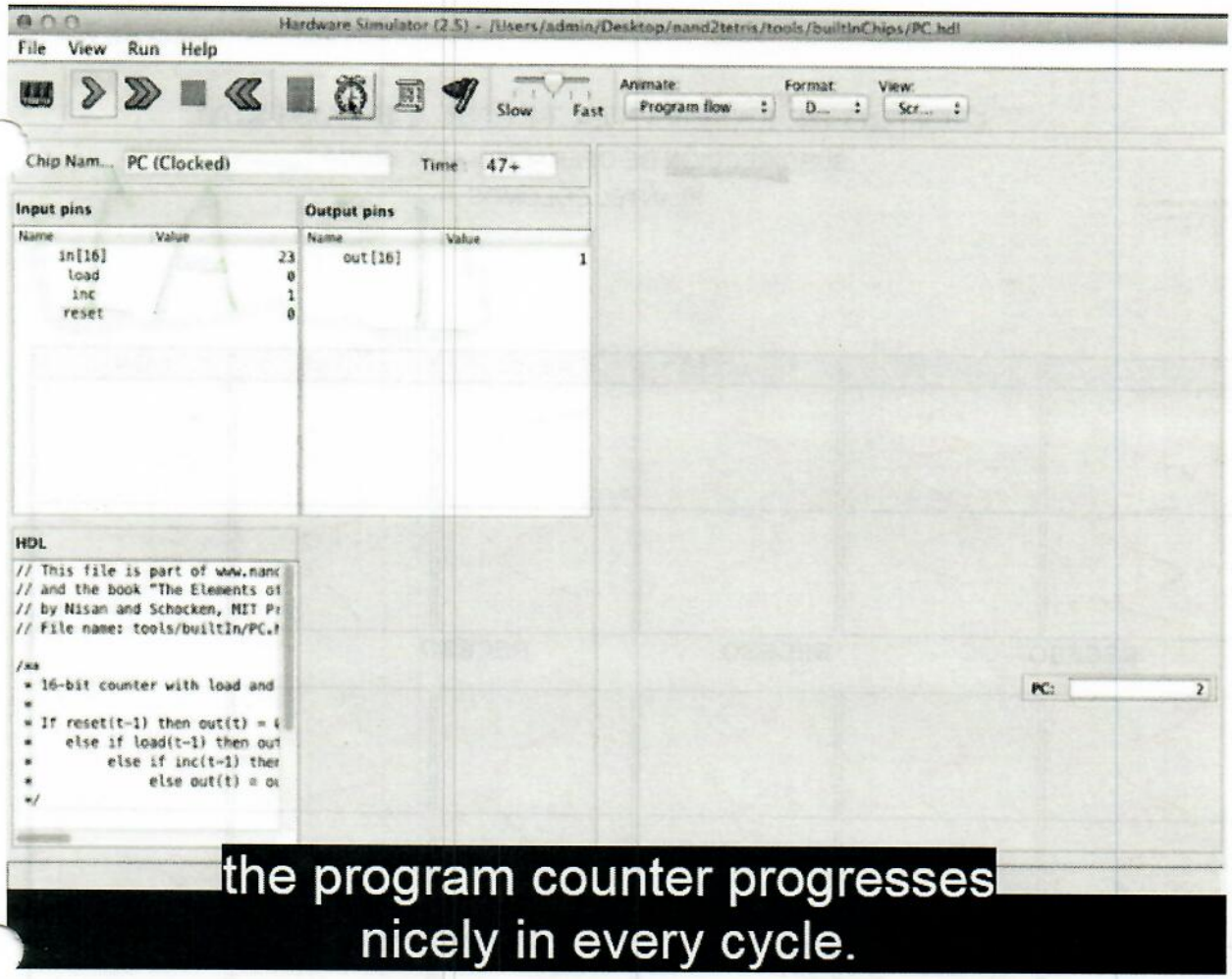
Contadores 12/14

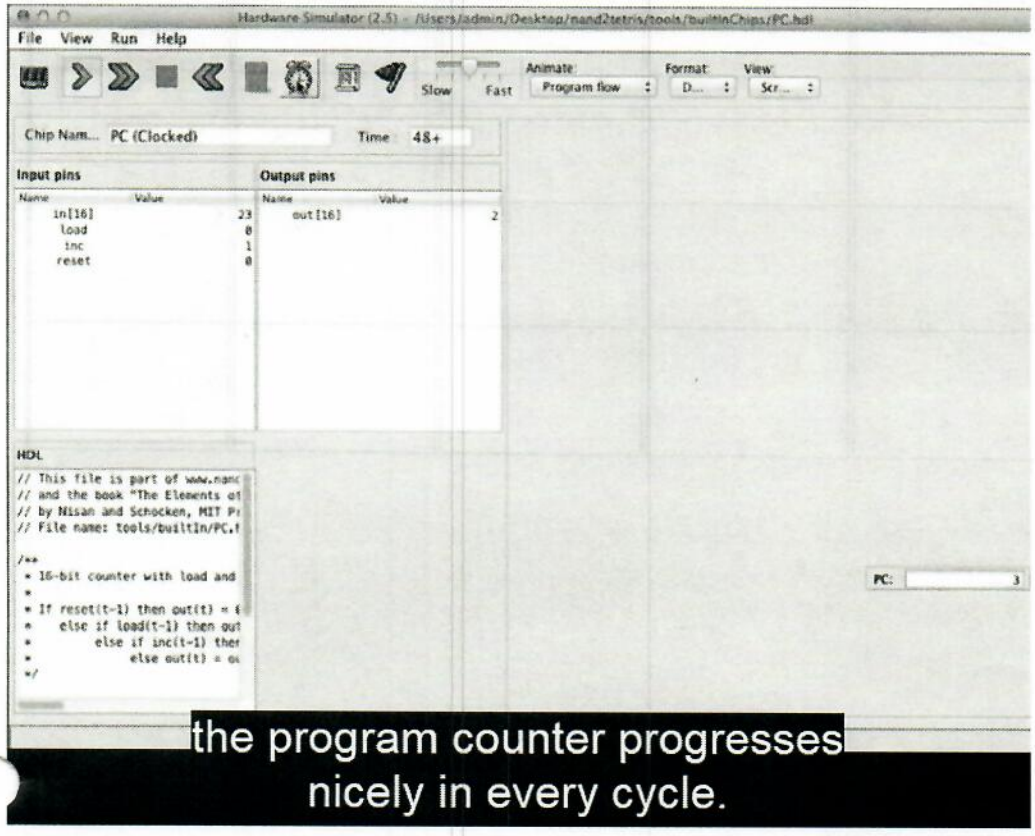
EMPEZÓ A CONTAR DESDE EL VALOR QUE TENÍA, AQUÍ SÓLO CAPTURE HASTA EL 28



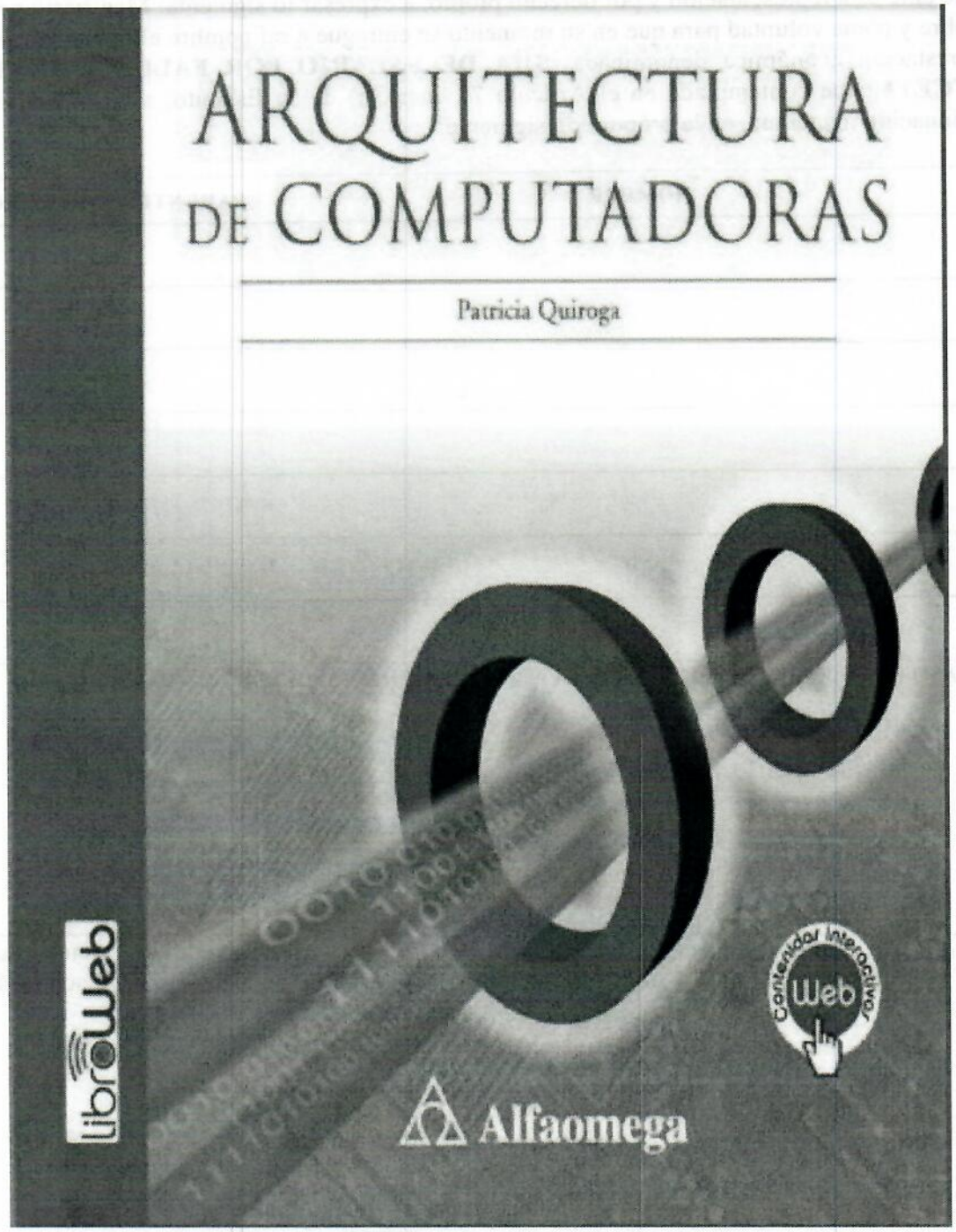
run the clock, and we see that now indeed,

Contadores 13/14


Hardware Simulator (2.5) - /Users/admin/Desktop/nand2tetris/tools/builtInChips/PC.hdl
File View Run Help
Slow Fast
Animate: Program flow
Format:
View:
Chip Nam... PC (Clocked)
Time 47+
Input pins
Name Value
in[16] 23
load 0
inc 1
reset 0
Output pins
Name Value
out[16] 1
HDL
// This file is part of www.nanc
// and the book "The Elements of
// by Nisan and Schocken, MIT P
// File name: tools/builtIn/PC.h
/**
* 16-bit counter with load and
*
* If reset(t-1) then out(t) = 0
* else if load(t-1) then out
* else if inc(t-1) then
* else out(t) = o
*/
PC: 2
the program counter progresses
nicely in every cycle.


Hardware Simulator (2.5) - /Users/admin/Desktop/nand2tetris/tools/builtInChips/PC.hdl
File View Run Help
Slow Fast
Animate: Program flow
Format:
View:
Chip Nam... PC (Clocked)
Time 48+
Input pins
Name Value
in[16] 23
load 0
inc 1
reset 0
Output pins
Name Value
out[16] 2
HDL
// This file is part of www.nanc
// and the book "The Elements of
// by Nisan and Schocken, MIT P
// File name: tools/builtIn/PC.h
/**
* 16-bit counter with load and
*
* If reset(t-1) then out(t) = 0
* else if load(t-1) then out
* else if inc(t-1) then
* else out(t) = o
*/
PC: 3
the program counter progresses
nicely in every cycle.

Contadores


ARQUITECTURA
DE COMPUTADORAS
Patricia Quiroga
libroWeb
Contenidos interactivos
Web
Alfaomega

Quiroga, Irma Patricia.
Arquitectura de computadoras. - 1a ed. - Buenos Aires : Alfaomega Grupo Editor Argentino, 2010.
372 pp. ; 24 x 21 cm.

ISBN 978-987-1609-06-2

1. Informática. 2. Arquitectura de Computadoras. I. Título
CDD 621.39

Edición: Damián Fernández
Corrección: Paula Smulevich y Silvia Mellino
Diseño de interiores: Juan Sosa
Diagramación de interiores: Diego Linares
Corrección de armado: Silvia Mellino
Revisión técnica: José Luis Hamkalo
Diseño de tapa: Diego Linares
Dibujos: Tomas L'Estrange

Internet: http://www.alfaomega.com.mx

ISBN 978-987-1609-06-2

Queda hecho el depósito que prevé la ley 11.723

Empresas del grupo:

Argentina: Alfaomega Grupo Editor Argentino, S.A.
Paraguay 1307 P.B. "11", Buenos Aires, Argentina, C.P. 1057
Tel.: (54-11) 4811-7183 / 8352
E-mail: ventas@alfaomegaeditor.com.ar

México: Alfaomega Grupo Editor, S.A. de C.V.
Pitágoras 1139, Col. Del Valle, México, D.F., México, C.P. 03100
Tel.: (52-55) 5089-7740 – Fax: (52-55) 5575-2420 / 2490. Sin costo: 01-800-020-4396
E-mail: atencionalcliente@alfaomega.com.mx

Colombia: Alfaomega Colombiana S.A.
Carrera 15 No. 64 A 29, Bogotá, Colombia
PBX (57-1) 2100122 - Fax: (57-1) 6068648
E-mail: scliente@alfaomega.com.co

Chile: Alfaomega Grupo Editor, S.A.
General del Canto 370-Providencia, Santiago, Chile
Tel.: (56-2) 235-4248 – Fax: (56-2) 235-5786
E-mail: agechile@alfaomega.cl

3.5 Códigos de representación numérica no decimal

Estos códigos permiten la operación aritmética de datos en sistema binario o en otras bases no decimales. Los operandos ingresan primero en código alfanumérico para luego convertirse a alguno de estos convenios. Si bien la conversión es más compleja que cuando se opera en sistema decimal, los datos que intervendrán en cálculos reducen su tamaño en grado considerable, con lo que se operan con mayor rapidez. A modo de ejemplo, se puede recordar cuántos dígitos ocupa el número $15_{(10)}$ en un convenio decimal, en contraposición al binario:

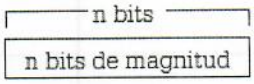
$$0001\ 0101_{(BCD)} \qquad 1111_{(2)}$$

3.5.1 Coma o punto fijo sin signo (enteros positivos)

Es el formato más simple para representar números enteros positivos, expresados como una secuencia de dígitos binarios:

$$X_{n-1};\ X_{n-2};\ \ldots\ X_2;\ X_1;\ X_0.$$

Como este formato sólo permite números sin signo, para la representación de un número se utiliza la totalidad de bits del formato.

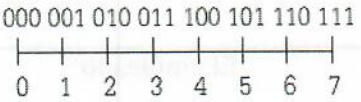


El rango de variabilidad para un número x en este formato es:

$$0 \le x \le 2^n - 1$$

donde "n" es la cantidad de dígitos que componen el número; también se la conoce como tipo de dato "ordinal sin signo".

Por ejemplo, para un formato de n = 3 bits, el rango de variabilidad estará comprendido entre 0 y 7. Expresado de otra manera, permite representar los números comprendidos en ese rango, como se observa en la recta de representación siguiente:



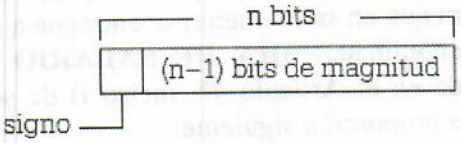
Por lo tanto, si se suman dos números cuyo resultado supere el rango de variabilidad estipulado, este resultado será erróneo, ya que perderá el dígito más significativo. El ejemplo siguiente muestra esta condición para n = 3:

	Comprobación
101	5
+ 111	+ 7
$1\ 100_{(2)}$	$12_{(10)}$

Las unidades de cálculo en una CPU actualizan "señalizadores", llamados flags o banderas, que indican, entre otras, esta circunstancia de desborde u overflow que, para este ejemplo en particular, implica la pérdida del bit de orden superior.

3.5.2 Coma o punto fijo con signo (enteros)

Es igual al formato anterior, pero reserva el bit de extrema izquierda para el signo, de manera que si ese bit es igual a 0 indicará que el número es positivo; asimismo, si, por el contrario, es igual a 1 indicará que el número es negativo. A este tipo de representación también se la llama "magnitud con signo".



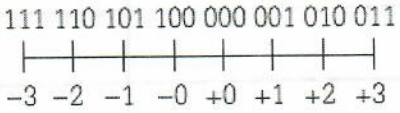
0: positivo

1: negativo

El rango de variabilidad para los binarios puros con signo es:

$$-(2^{n-1} - 1) \leq x \leq +(2^{n-1} - 1)$$

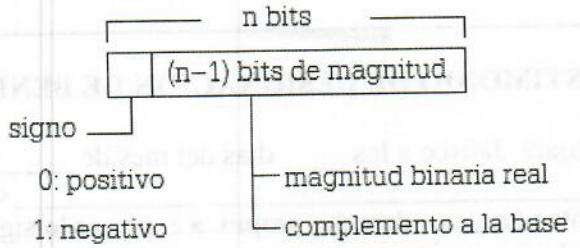
Para el ejemplo de $n = 3$ bits, el rango de variabilidad estará comprendido entre -3 y $+3$, según la recta de representación siguiente:



Nótese la presencia de un cero positivo y otro negativo. Este último surge del cambio de signo, 0 por 1, para esa misma magnitud.

3.5.3 Coma o punto fijo con signo con negativos complementados a "2" (enteros)

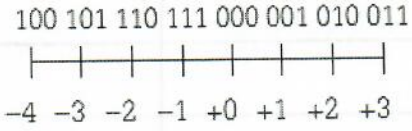
Es igual al formato anterior, reservando el bit de extrema izquierda para el signo, de manera que si ese bit es igual a 0, indicará que el número es positivo; por el contrario, si ese bit es igual a 1, indicará que el número es negativo. La diferencia radica en que los $n - 1$ bits restantes se representan con su magnitud binaria real para el caso de los positivos y en el complemento a la base de la magnitud para el caso de los negativos.



El rango de variabilidad para los binarios con signo en complemento a la base es:

$$-(2^{n-1}) \leq x \leq +(2^{n-1}-1)$$

Para $n = 3$ bits, el rango de variabilidad estará comprendido entre -4 y $+3$, como se muestra en la recta de representación siguiente:



La representación de los números negativos surge de aplicar la definición para hallar el complemento a la base de un número; por lo tanto, el complemento del número $+2$ con un formato de 3 bits es:

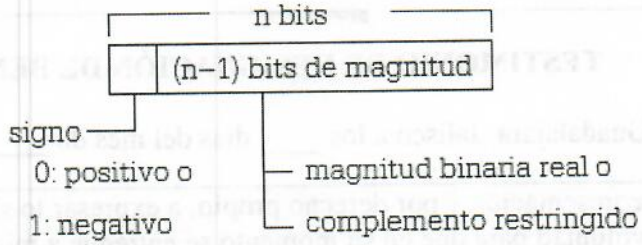
$$\begin{array}{r} 1000 \\ -\ 010 \\ \hline \boxed{110} \end{array}$$

De forma similar, el complemento del número $+3$ es:

$$\begin{array}{r} 1000 \\ +\ 011 \\ \hline \boxed{101} \end{array}$$

3.5.4 Coma o punto fijo con signo con negativos complementados a "1" (enteros)

Es igual al formato anterior, reservando el bit de extrema izquierda para el signo, de manera que si ese bit es igual a 0, indicará que el número es positivo; por el contrario, si ese bit es igual a 1, indicará que el número es negativo. No obstante, los $n-1$ bits restantes se representan con su magnitud binaria real para el caso de los positivos y en complemento restringido de la magnitud para el caso de los negativos.

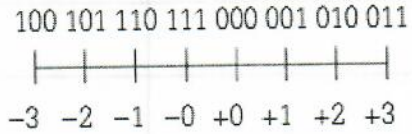


El rango de variabilidad para los binarios puros con signo en complemento restringido o complemento a 1, es:

$$-(2^{n-1}-1) \leq x \leq +(2^{n-1}-1)$$

En esta codificación también hay dos "ceros", uno positivo y otro negativo.

Luego, para $n = 3$ la recta representativa del rango de variabilidad será:



3.5.5 Reales en coma o punto flotante (números muy grandes y números reales)

Los números analizados hasta ahora se pueden clasificar como enteros sin signo y enteros signados, en general limitados por un formato de 8, 16, 32 o 64 bits; sin embargo, a veces es necesario operar datos mayores (por ejemplo, para programas que resuelvan cálculos científicos). La primera solución para tratar números muy grandes es ampliar el formato a tres, cuatro u ocho octetos y obtener así valores decimales que oscilan entre alrededor de los 8 millones y los 9 trillones (considerando enteros signados). De todas maneras, en los cálculos no siempre intervienen valores enteros y es entonces donde aparece el inconveniente de cómo representar el punto para manejar cantidades fraccionarias. Por un lado, el punto debería expresarse con un dígito "0", o bien, con un dígito "1", dentro de la cadena de bits que representa al número binario; por otro lado, el punto no asume una posición fija sino variable dentro de la cadena, o sea que si el punto se representa con un bit "1" no se podrá diferenciar en una secuencia " 1111" el binario 1111 del binario 111,1 o bien del binario 1,111. La solución para este problema es eliminar el punto de la secuencia e indicar de alguna manera el lugar que ocupa dentro de ella. De esta forma, los números se representan considerando dos partes. La primera indica los bits que representan al número y se llama **mantisa**, la otra indica la posición del punto dentro del número y se llama **exponente**. Esta manera de tratar números fraccionarios tiene cierta semejanza con lo que se conoce en matemática como notación científica o exponencial. Veamos cómo se puede representar un número decimal:

La mantisa representa todos los bits del número sin coma o punto decimal.

$$3{,}5_{(10)} = \begin{cases} 35 \cdot 10^{-1} \\ 0{,}35 \cdot 10^{+1} \end{cases}$$ o bien, es igual a

El exponente representa la posición de la coma o punto decimal en la mantisa.

En el primer caso, 35 es un número entero que, multiplicado por la base 10 y elevado a la −1, expresa el mismo número inicial sin representar la coma dentro de él. Así, "35" es la mantisa y "−1", el exponente, que se puede leer como "el punto está **un** lugar a la **izquierda** del dígito menos significativo de la mantisa".

En el segundo caso, 0,35 es un número fraccionario puro (se indica entonces que es una fracción **normalizada**, dado que se supone que el punto está a la izquierda del dígito más significativo y distinto de cero de la mantisa). Luego, 0,35 multiplicado por la base 10 y elevado a la +1 expresa el mismo número inicial, sin representar la coma dentro de él. En este caso, "+1" indica que el punto está **un** lugar a la **derecha** del dígito más significativo de la mantisa.

Cuando se trata de números binarios con punto fraccionario se utilizan, entonces, dos entidades numéricas, una para la mantisa (por lo general una fracción normalizada y con signo) y otra para el exponente. La base del sistema es siempre conocida, por lo tanto, no aparece representada. Este nuevo formato de números se conoce como binarios de punto (coma) flotante, ya que el punto varía su posición ("flota") según el contenido binario de la entidad exponente.

La fórmula $\pm M \cdot B^{\pm P}$ representa el criterio utilizado para todo sistema posicional, donde "M" es la mantisa, que podrá tratarse como **fracción pura** o como **entero**, según se **suponga** la coma a **izquierda** o a **derecha** de "M". "B" es la base del sistema; "P" es el exponente que indica la posición del punto en "M" y los signos "+" y "−" que la anteceden indican su desplazamiento a derecha o izquierda respecto del **origen**. En la memoria sólo se almacenarán la mantisa "M" y la potencia "P" en binario. La base "B" y el **origen** del punto siempre son determinados con anterioridad por el convenio y conocidos por los programas que utilizan esta representación.

3.5 Códigos de representación numérica no decimal

La representación de datos en punto flotante incrementa el rango de números más allá de los límites físicos de los registros. Por ejemplo, se sabe que en un registro de 16 bits el positivo mayor definido como entero signado que se puede representar es el +32767. Si se considera el registro dividido en partes iguales, 8 bits para la mantisa entera y 8 bits para el exponente, el entero mayor se calcula según la fórmula siguiente:

$$+(2^7-1)\cdot 2^{+(2^7-1)} = +127 \cdot 2^{+127}$$

que implica que al binario representativo del entero 127 se le agregan 127 ceros a derecha; de modo que enteros muy grandes encuentran su representación en esta modalidad.

Si ahora se representa el valor $3{,}5_{(10)}$ en binario 11,1 resulta:

$+3,5_{(10)} = +11,1_{(2)}$ expresado en notación científica es igual a

$+111,0_{(2)} \cdot 10_{(2)}^{-1_{(2)}} = +7_{(10)} \cdot 2_{(10)}^{-1_{(10)}} = +7/2 = +3,5_{(10)}$ mantisa entera

$+0,111_{(2)} \cdot 10_{(2)}^{+10_{(2)}} = +0,875_{(10)} \cdot 2_{(10)}^{+2_{(10)}} = +3,5_{(10)}$ mantisa fraccionaria

Dado que los registros que operen estos datos tendrán una longitud fija, se asume un formato de "m" bits para la mantisa y "p" bits para el exponente, entonces se puede definir el rango de representación de números reales, que es siempre finito. O sea que determinados reales son "representables" en el formato y otros quedan "afuera" de él.

Cuando el resultado de una operación supera los límites mayores definidos por el rango, entonces es incorrecto. El error se conoce como overflow **de resultado** y actualiza un bit (flag de overflow) en un registro asociado a la ALU, conocido como registro de estado o status register.

Cuando el resultado de una operación cae en el hueco definido por los límites inferiores del rango (del que se excluye el cero), esto es

$$0 > x > 0,1 \quad o \quad 0 < x < 0,1$$

es incorrecto. El error se conoce como underflow de resultado.

3.5.5.1 Convenios de representación en punto flotante con exponente en exceso

Para eliminar el signo del exponente se utilizan convenios que agregan al exponente un exceso igual a 2^{p-1}. Esta entidad nueva se denomina característica y es igual a:

$$\text{CARACTERÍSTICA} = \underbrace{\boxed{\pm P}}_{\text{EXPONENTE}} + \underbrace{\boxed{2^{p-1}}}_{\text{EXCESO}}$$

Siguiendo con el ejemplo de $+3,5_{(10)}$, pero en este caso en un formato de m = 8 (cantidad de bits de la mantisa, incluido el signo) y p = 8 (cantidad de bits de la potencia), el exceso será:

$2^{p-1} = 2^7 = 128\ (10000000_2)$ donde "p" es la cantidad de bits de la característica.

En este convenio la mantisa se representa en la forma de signo y magnitud, o sea, **no se utiliza el complemento para los negativos**. Ahora bien, la forma en que la computadora "ve" la mantisa puede ser entera o fraccionaria y sin normalizar o normalizada. Analicemos las diferentes formas que puede adoptar la mantisa y su manera de representación, siempre para el mismo ejemplo.

Overflow: es un error que se produce cuando el resultado se encuentra fuera de los límites superiores del rango.

Para una mantisa **entera** será:

Formato alojado en una locación de memoria

$3{,}5_{(10)} = 11{,}1_{(2)} = 111{,} \cdot 10^{-1}$

Característica = 128 − 1 = 127

exc. exp.

CARÁCTER.	S	MANTISA
01111111	0	0000111.

+ (positivo)

Si la mantisa fuese **fraccionaria** la forma de representarla sería:

$3{,}5_{(10)} = 11{,}1_{(2)} = {,}0000111 \cdot 10^{+110}$

Característica = 128 + 6 = 134

CARÁCTER.	S	MANTISA
10000110	0	.0000111

En general, la manera de representar la mantisa es normalizada; por lo tanto, si se necesita expresar el número anterior con mantisa **fraccionaria y normalizada**, éste será:

$3{,}5_{(10)} = 11{,}1_{(2)} = {,}111 \cdot 10^{+10}$

Característica = 128 + 2 = 130

CARÁCTER.	S	MANTISA
10000010	0	.1110000

En la tabla 3-5 se muestran las correspondencias entre el exponente y la característica.

Tabla 3-5. Correspondencias entre el exponente y la característica.

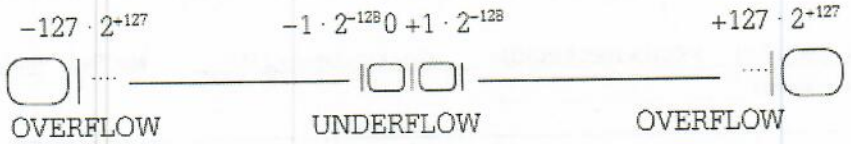
Si el exponente es	la característica será
-128	0
-1	127
0	128
127	255

Analicemos la tabla 3-6, en la que se observan los valores máximos y mínimos que se pueden representar con m = 8 y p = 8, con una **mantisa entera sin normalizar** (excedida en 128).

Tabla 3-6. Valores máximos y mínimos representados con m=8 y p=8

	Mantisa	Carácter	Mant.	Poten.	Resulta de
- el mayor nro. positivo es:	0 1111111	11111111	+127	+127	255 − 128 = +127
- el menor nro. positivo es:	0 0000001	00000000	+1	-128	0 − 128 = −128
- el menor nro. negativo es:	1 0000001	00000000	-1	-128	0 − 128 = −128
- el mayor nro. negativo es:	1 1111111	11111111	-127	+127	255 − 128 = +127

Los valores máximos y mínimos de la tabla se ven reflejados en la recta de representación siguiente:



Genéricamente:

- El número mayor positivo queda definido como $+(2^{m-1}-1)\cdot 2^{2^{(p-1)}-1}$
- El número mayor negativo queda definido como $-(2^{m-1}-1)\cdot 2^{2^{(p-1)}-1}$
- El número menor positivo queda definido como $+2^{-2^{(p-1)}}$
- El número menor negativo queda definido como $-2^{-2^{(p-1)}}$

3.5 Códigos de representación numérica no decimal

En resumen, los valores máximos y mínimos que se pueden representar con una mantisa entera sin normalizar son los que se visualizan en la tabla 3-7.

Tabla 3-7. Límites de overflow y underflow.

$\pm(2^{m-1}-1)\cdot 2^{2^{(p-1)}-1}$	$\pm 1\cdot 2^{+2^{(p-1)}}$

En los rangos de variabilidad la recta no es siempre continua, porque a medida que el exponente "flota" fuera del límite del registro se "agregan ceros" a derecha o a izquierda de la mantisa y, por lo tanto, se expresan valores enteros o fracciones puras muy pequeñas, por lo que quedan sin representación los valores intermedios.

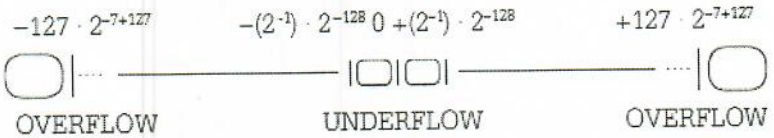
Con una **mantisa normalizada fraccionaria** donde m = 8 y p = 8 (excedida en 128), los valores máximos y mínimos que se pueden llegar a representar son los se muestran en la tabla 3-8. Recuerde que 127 es el valor correspondiente a una mantisa entera, entonces, para calcular el valor 0,1111111 se lo debe multiplicar por 2^{-7}, lo que provoca el desplazamiento del punto de extrema derecha a extrema izquierda:

$$+127\cdot 2^{-7} = \frac{+127}{+128} = 0.99218..$$

Tabla 3-8. Valores máximos y mínimos.

	Mantisa	Carácter	Mant.	Poten.
- el mayor nro. positivo es:	0 1111111	11111111	$127 \cdot 2^{-7}$	+127
- el menor nro. positivo es:	0 1000000	00000000	+.1	−128
- el menor nro. negativo es:	1 1000000	00000000	−.1	−128
- el mayor nro. negativo es:	1 1111111	11111111	$-127 \cdot 2^{-7}$	+127

Los valores máximos y mínimos de la tabla, para mantisa fraccionaria normalizada, se reflejan en la recta de representación siguiente:



Por último, los valores máximos y mínimos que se pueden representar con una mantisa fraccionaria normalizada son los que se visualizan en la tabla 3-9.

Tabla 3-9. Límites de overflow y underflow.

$\pm(2^{m-1}-1)\cdot 2^{-(m-1)}\cdot 2^{2^{(p-1)}-1}$	$\pm(2^{-1})\cdot 2^{-(2^{(p-1)})}$

Los sistemas de representación de datos en punto flotante analizados hasta ahora nos permiten introducirnos en el estudio de otros, cuya comprensión hubiese resultado confusa sin este primer acercamiento.

3.5.5.2. Punto flotante "exceso 127"

En esta convención el número se representa en formato de4 octetos:

1 bit	8 bits	m = 23 bits mantisa
S	±p + 127	

1, (posición supuesta del bit implícito)

El cálculo de la característica o el exponente responde al criterio empleado en la convención anterior:

$$\pm P + (2^{8-1} - 1) = \pm P + (2^7 - 1) = \pm P + 127_{(10)}$$

El primer bit corresponde al signo de la mantisa (0 positiva, 1 negativa) y en los 31 bits restantes se representa la característica y su magnitud binaria real. El lugar "supuesto" dado que no se representa para su almacenamiento, es la posición entre la característica y los bits de mantisa. Como la mantisa es una fracción pura normalizada, el primer bit luego de la coma debe ser 1. Por esta razón este "1" tampoco se almacena y "se supone" antes de la coma. Por lo tanto, la mantisa está constituida por los bits a la derecha de ella hasta llegar a 23.

$$\underline{1 b_{22} b_{21} b_{20} \ldots\ldots\ldots\ldots b_3 b_2 b_1 b_0} \quad \boxed{\text{23 bits de mantisa}}$$

Si ahora representamos $+3{,}5_{(10)} = +11{,}1_{(2)} = +1{,}11_{(2)} \cdot 10_{(2)}^{+1_{(2)}}$ calculamos la característica como (+1) + 01111111 = 10000000

El signo es positivo, por lo tanto, el formato queda:

1 bit	8 bits	23 bits		
0	10000000	11000000	00000000	0000000

El seguimiento de otros ejemplos facilitará su entendimiento:

$$+4{,}0_{(10)} = +100_{(2)} = 1{,}0_{(2)} \cdot 10_{(2)}^{+10_{(2)}}$$

0 10000000 00000000 00000000 0000000

El exponente $+10_{(2)}$ significa "desplazar la coma a la derecha dos lugares".

$$+16{,}0_{(10)} = +10000_{(2)} = 1{,}0_{(2)} \cdot 10_{(2)}^{+100_{(2)}}$$

0 10000011 00000000 00000000 0000000

El exponente +100 significa "desplazar la coma a la derecha cuatro lugares".

$$-1{,}0_{(10)} = -1_{(2)} = -1{,}0_{(2)} \cdot 10_{(2)}^{0_{(2)}}$$

1 01111111 00000000 00000000 0000000

El exponente 0 significa "no desplazar la coma".

$$-0{,}5_{(10)} = -0{,}1_{(2)} - 0{,}1_{(2)} \cdot 10_{(2)}^{-1_{(2)}}$$

1 01111110 00000000 00000000 0000000

El exponente –1 significa "desplazar la coma a la izquierda un lugar".

$$+0,125_{(10)} = +0,001_{(2)} = 1,0_{(2)} \cdot 10_{(2)}^{-11_{(2)}}$$

El exponente – 11 significa "desplazar la coma a la izquierda tres lugares".

3.5 Códigos de representación numérica no decimal

Estos dos últimos ejemplos representan casos de números menores que cero, por lo tanto, la coma sufre un "desplazamiento a izquierda", que se representa en convenio con el signo del exponente negativo y la característica menor que 127.

De la observación de los últimos cinco ejemplos se pueden sacar las conclusiones siguientes:

- Los números negativos sólo se diferencian de los positivos por el valor del bit de signo asociado a la mantisa.
- Toda potencia de 2 se representa con mantisa $1,00..._{(2)}$ y toma su valor según el desplazamiento del punto.
- Con una misma mantisa y distintas características se representan diferentes valores decimales.
- El punto puede "flotar" aun fuera del límite físico que impone el formato.

Pasos para la representación de un número en este formato

1. Convertir el número decimal a binario.
2. Normalizar, eliminar el bit denominado "implícito" y calcular el valor de "P" para determinar el desplazamiento del punto.
3. Calcular y representar la característica.
4. Acomodar el signo y la mantisa, considerando sólo los dígitos fraccionarios hasta completar el formato.

Ahora considérese el ejemplo siguiente:

$$-0,2_{(10)} = -0,\overline{0011}_{(2)} = -0,1100110011..._{(2)} \cdot 10_{(2)}^{11_{(2)}} = -1,\overline{1001} \cdot 10_{(2)}^{11_{(2)}}$$

Esta conversión no es exacta, o sea que la fracción binaria hallada es periódica y, por lo tanto, se aproxima a la fracción decimal original.

El error varía en relación con la cantidad de períodos que se consideren. Cuando se normaliza, los ceros a derecha del punto desaparecen, se indica su existencia con el valor del exponente $-11_{(2)}$ (tres lugares a izquierda del origen de la coma). El número representado queda así:

1 01111100 10011001 10011001 1001100 truncado

110011001...

En la mantisa se repite el período tantas veces como el formato lo permite hasta que se trunca.

Con este último ejemplo se pretende que quede claro que si la cantidad de bits de la mantisa aumenta, mejor será la precisión del número.

A continuación, se presenta un ejemplo que servirá de ejercitación. Representemos en punto flotante convenio exceso -127 los números $-5001,25_{(10)}$ y $+10,1_{(10)}$.

$$-5001,25_{(10)} = -1001110001001,01 = -1,00111000100101_{(2)} \cdot 10_{(2)}^{+1100}$$

1 11110011 00111000100101000000000

–	115	bit implícito

Precisión simple y precisión doble

En esta convención la cantidad de bits que representan el número admite dos variantes:

- Formato de precisión simple: cuatro octetos.
- Formato de precisión doble: ocho octetos.

Los ejemplos aportados pertenecen al formato de precisión simple. La determinación de los límites sobre la recta real nos permite observar la mejora en la precisión del formato mayor. Queda a criterio del programador cuándo utilizar una u otra, al momento de definir en sus programas variables reales; se debe tener en cuenta que una mejora en la precisión implica el almacenamiento de mayor cantidad de bits, y esto tendrá desventajas que deberán evaluarse.

3.6 Representaciones redundantes

3.6.1 Códigos de detección y/o corrección de errores. Introducción

El cambio de un bit en el almacenamiento o la manipulación de datos origina resultados erróneos. Para detectar e incluso corregir estas alteraciones se usan códigos que agregan "bits redundantes". Por ejemplo, los bits de paridad se agregan al dato en el momento de su envío y son corroborados en el momento de su recepción por algún componente de la computadora, y lo mismo sucede al revés. De todas formas, la cantidad de errores que puedan producirse son mensurables probabilísticamente. El resultado de "medir" la cantidad de "ruido" que puede afectar la información se representa con un coeficiente conocido como **tasa de error**. De acuerdo con el valor de la tasa de error, se selecciona un código, entre los distintos desarrollados, para descubrir e incluso corregir los errores. Estos códigos reciben el nombre de códigos de paridad.

3.6.2 Paridad vertical simple o a nivel carácter

Al final de cada carácter se incluye un bit, de manera que la suma de "unos" del carácter completo sea par (paridad par) o impar (paridad impar). Este tipo de codificación se denomina paridad vertical. Suele acompañar la transmisión de octetos en los periféricos y la memoria.

Los ejemplos muestran el bit de paridad que se agrega al octeto, en el caso de usar paridad par o impar:

00110010 | 1 se agrega un uno que permita obtener paridad par

01001011 | 0 se agrega un cero para lograr paridad par

Este código de detección de errores sólo se utiliza si la tasa de error en la cadena de bits que se han de transmitir no es superior a uno; esto es, si hay dos bits erróneos no permite detectarlos.

3.6.3 Paridad horizontal a nivel de bloque

Por cada bloque de caracteres se crea un byte con bits de paridad. El bit 0 será el bit de paridad de los bits 0 del bloque; el bit 1, el de paridad de los bits 1del bloque, y así sucesivamente.

Bit de paridad: es un bit redundante agregado a una cadena de bits en la que se pretende detectar un posible error.

3.6.4 Paridad entrelazada

Utilizando en conjunto el código de paridad vertical con el horizontal se construye el código de paridad entrelazada, que además de detectar errores permite corregirlos.

Supongamos una transmisión de cuatro caracteres en código ASCII de 7 bits, con paridad par. El primer grupo indica cómo deberían llegar los caracteres a su lugar de destino, de no haberse producido errores en la transmisión. El segundo grupo indica con un recuadro el bit que tuvo un error en la transmisión.

	PH		PH
	0 1 0 1 0		0 1 0 1 0
	0 1 1 1 1		0 [0] 1 1 1 ———
	1 1 0 1 1		1 1 0 1 1
	1 0 1 1 1		1 0 1 1 1
	1 1 1 0 1		1 1 1 0 1
	1 0 1 1 1		1 0 1 1 1
	0 1 1 0 0		0 1 1 0 0
PV —	0 1 1 1 1	PV —	0 1 1 1 1
	sin error		con error

La forma de corregir el error es invertir el bit señalado como erróneo. Si hay más de un error en la misma fila o columna, quizá se pueda detectar, pero no se podrá determinar con exactitud cuál es el error. En la práctica este código no se considera óptimo, debido a la cantidad de bits de paridad que se deben agregar, por lo que suele utilizarse el código de Hamming.

3.6.5 Código de Hamming

Este código permite detectar y corregir los errores producidos en una transmisión con sólo agregar p bits de paridad, de forma que se cumpla la relación siguiente:

$$2^p \geq i + p + 1$$

donde i es la cantidad de dígitos binarios que se han de transmitir y p es la cantidad de bits de paridad.

Supongamos que se desean transmitir 4 dígitos binarios. Según el método de Hamming a éstos deberá agregarse la cantidad de bits de paridad necesarios para satisfacer la relación enunciada antes:

$i = 4$	para que la relación $2^p \geq i + p + 1$ se cumpla debe ser	
$p = ?$	$p = 3$, luego	$2^3 \geq 4 + 3 + 1$

De esta relación se deduce que la transmisión será de 4 bits de información más 3 de paridad par. Se debe tener en cuenta que con 2 bits de paridad se puede corroborar la transmisión de 1 dígito de información, con 3de paridad se corroboran hasta 8 bits de información, con 4 de paridad se corroboran hasta 11 de información, y así sucesivamente.

En la cadena de 7 bits del ejemplo: $b_1, b_2, \ldots, b_7$, son bits de paridad los que ocupan las posiciones equivalentes a las potencias de dos y el resto son bits de datos.

La tabla siguiente es la guía para determinar la distribución y el cálculo de los bits de paridad y su corroboración luego de la transmisión de hasta 11 dígitos binarios.

	$p_1 = b_1$	$p_2 = b_2$	$p = b_4$	$p_4 = b_8$
$p_1 = b_1$	*			
$p_2 = b_2$		*		
$i_1 = b_3$	*	*		
$p_3 = b_4$			*	
$i_2 = b_5$		*		*
$i_3 = b_6$			*	*
$i_4 = b_7$	*	*	*	
$p_4 = b_8$				*
$i_5 = b_9$		*		*
$i_6 = b_{10}$		*		*
$i_7 = b_{11}$	*	*		*
$i_8 = b_{12}$			*	*
$i_9 = b_{13}$	*		*	*
$i_{10} = b_{14}$		*	*	*
$i_{11} = b_{15}$	*	*	*	*

Por lo tanto, para corroborar la transmisión de cuatro bits de información, deberán verificarse las igualdades siguientes:

$$p_1 = b_1 + b_3 + b_5 + b_7$$
$$p_2 = b_2 + b_3 + b_6 + b_7$$
$$p_3 = b_4 + b_5 + b_6 + b_7$$

Si llegara a producirse algún error en la transmisión, (por ejemplo, se transmite el número binario 1101 y se recibe el 1100) el método de Hamming indicará el número del bit donde se produjo el error, mediante el proceso de verificación que realiza en el punto receptor. Veamos:

Se transmiten: $i_4\ i_3\ i_2\ p_3\ i_1\ p_2\ p_1$

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

1 0 1 0 1 0 1

Se reciben: 0 0 1 0 1 0 1

Verificación: $p_3\ p_2\ p_1$

1 1 1 = el bit $7_{(10)}$ es el erróneo.

Una vez detectado el error la corrección es sencilla, basta con invertir el bit erróneo.

3.8 Ejercicios propuestos

1) La siguiente es una sentencia en lenguaje de alto nivel, C = A – B.

Indicar un código que la represente en el programa fuente y establecer cuántos octetos ocuparía en memoria.

2) Para un formato de 32 bits, indicar los rangos de variabilidad en decimal, para números representados en ASCII sin signo y en binario signado en complemento a 2, e indicar cómo se representa el número $+25_{(10)}$ en los dos sistemas.

3) Dado el número $7910_{(10)}$, expresarlo en:

a) Decimal codificado en binario.

b) BCD exceso -3.

c) BCD exceso -2421.

d) Binario natural.

4) Si D2 es la representación hexadecimal de un número, exprese cuál es ese número en sistema decimal, si se considera un entero:

a) No signado.

b) Signado.

6) Representar $(-18)_{(10)}$ y $(-64)_{(10)}$ con un formato de 16 bits, en punto fijo en complemento restringido y en punto fijo en complemento a la base.

7) Determinar el rango de variabilidad para un formato de punto fijo con negativos complementados a dos en:

a) 6 bits.

b) 32 bits.

c) 64 bits.

8) Dados los siguientes números, expresarlos en punto flotante, convención exceso 127.

a) $+32_{(10)}$

b) $+3.25_{(10)}$

c) $(-0.4)_{(10)}$

d) $(-10.1)_{(10)}$

e) $+0.5_{(10)}$

9) La siguiente representación hexadecimal determina una cadena binaria: 945A.

5) Expresar en decimal los límites de representación para un formato de 16 bits, si éste contiene:

a) Un entero no signado.

b) Un entero signado.

c) Suponiendo que la cadena es la representación de un número en punto flotante en exceso con mantisa entera, donde m = 8 y p = 8, indicar:

- ¿Cuál es el exceso utilizado?
- ¿Cuál es el número decimal que representa en la forma $\pm M \cdot B^{e-7}$

10) Para un formato que permita una mantisa fraccionaria con m = 8 y p = 8:

a) Determine los límites de representación, a partir de los cuales se produce *overflow* y *underflow* de resultado.

b) ¿Cuál es el exceso que permite emplear este formato?

c) ¿Qué permite la utilización de una mantisa con mayor cantidad de bits?

d) ¿Cuál sería la consecuencia si "p" tuviera mayor cantidad de bits?

a) Indicar el valor decimal que representa si la cadena se considera binario natural.

b) Indicar qué valor decimal representa si la cadena se considera un binario signado en sus tres formas de representación.

11) Determine los límites, a partir de los cuales se produciría *overflow* y *underflow* de resultado, en precisión simple y en precisión doble, para el formato conocido como "exceso-127".

12) Suponga que recibe el número $87_{(10)}$ en el *buffer* de la impresora para ser listado en representación ASCII y que la transmisión utiliza como método de control el código de Hamming:

a) Desarrolle las funciones para determinar el cálculo de los bits de paridad.

b) Suponga que se producen errores en la transmisión en el bit número:

- 10 para el dígito $8_{(10)}$
- 13 para el dígito $7_{(10)}$

c) Exprese cómo se desarrolla la detección y la corrección de ambos errores.