

Encriptador a base de N.º aleatorios en PicoBlaze 8 bits

Compuesto por 2 periféricos y 1 instrucción:

Encriptador (Maquina de estados de 9 estados):

El encriptador funciona mediante una operación lógica XOR entre una clave de 1 Byte cada una, proporcionada por el usuario, y un dato de 1 Byte que se desea cifrar. El resultado de esta operación es el dato encriptado, que se transmite para evitar el reconocimiento y el robo por parte de un intruso o ciberdelincuente. Para descifrar el dato encriptado, el usuario debe ingresar la misma clave junto con el dato encriptado, lo que permite recuperar el dato original mediante una operación XOR inversa.

Veamos la manera en la que hemos implementado este periférico.

Definición del periférico en el toplevel:

Primero se ha definido el periférico en el toplevel con todos los puertos de entrada y salida.

```
component encriptador

port( data_in: in std_logic_vector(7 downto 0);
      port_id: in std_logic_vector(7 downto 0);
      write_strobe: in std_logic;
      clk: in std_logic;
      reset: in std_logic;
      data_out: out std_logic_vector(7 downto 0));

end component;
```

Entradas del periférico:

Nuestro periférico tiene 5 registros, cada uno de ellos escucha en un puerto diferentes:

R0=>port_id = x"41"

R1=>port_id = x"42"

R2=>port_id = x"43"

R3=>port_id = x"44"

R7=>port_id = x"47"

Todos los registros están definidos de un modo similar al siguiente, (hemos introducido la señal asíncrona rst, para poder resetear todos los registros una vez que termine el proceso de encriptar o desencriptar)

```

    ---PROCESO R0  x41 C1
    process(clk, reset, rst) begin
        if(reset='1' or rst='1') then
            R0 <=(others =>'0');
        elsif rising_edge(clk) then
            if (port_id = x"41" and write_strobe = '1') then
                R0 <= data_in;
            end if;
        end if;
    end process;
    |

```

Salidas del periférico:

La salida del periférico se hace por el puerto 50.

data_out=> conectada a una señal de salida (data_out_encryptador) en el puerto x"50".

El valor de la salida de nuestro periférico dependerá de la operación que ha solicitado el usuario (encriptar o desencriptar).

```

    process(clk, reset) begin
        if(reset='1') then
            data_out <=(others =>'0');
        elsif rising_edge(clk) then
            if(encripter_ok='1') then
                data_out<=data_encryptad;

            elsif(desencripter_ok='1') then
                data_out<= data_desencriptad;

            elsif(enable_error='1')then
                data_out <=x"3F";
            end if;
        end if;
    end process;

```

Se ha definido en el toplevel una señal de salida que es la que conectará la salida del periférico con la entrada del multiplexor:

```

-- señal de salida del periférico--
    signal data_out_encryptador: std_logic_vector(7 downto 0);

```

Procedemos a "conectar" la salida de nuestro periférico "data_out" con la salida que va conectada al multiplexor.

```
encriptador1:encriptador
port map(
    data_in => outport,
    port_id => portid,
    write_strobe => writestrobe,
    clk => clk,
    reset => reset,
    data_out => data_out_encriptador);
```

Finalmente, definimos la conexion al multiplexor del siguiente modo:

```
-- Multiplexor import
inport <= RAM_out when (readstrobe = '1' and portid<x"40") else
    rxbuff_out when (readstrobe = '1' and portid=x"FF") else
    data_out_encriptador when (readstrobe='1' and portid=x"50") else
```

Internamente nuestro periférico consiste en una máquina de 8 estados: inicio, clave, control, encripter, desencrypter, final, enc_plus, denc_plus.

Código VHDL de la máquina de estados del Encriptador:

```
process(estado_actual,enable_enc,enable_denc,R3,R7)
begin
    rst<='0';enable_enc<='0'; enable_denc<='0'; encripter_ok<='0'; desencrypter_ok<='0'; enable_error<='0';

    case estado_actual is

    when inicio =>
        if(R7 =x"42") then
            estado_siguiente<=clave;
        else
            estado_siguiente<=inicio;
        end if;

    when clave=>
        --CONMUTO A CONTROL SI EN R3 HAY UN #
        if(R3=X"23") then
            estado_siguiente<=control;
        else
            estado_siguiente<=clave;
        end if;

    when control=>
        if(R3=x"2b") then -- R3=+
            enable_enc<='1';
            estado_siguiente<=encripter;

        elsif(R3=x"2d") then -- R3=-
            enable_denc<='1';
            estado_siguiente<=desencrypter;
        elsif(R3=x"31") then -- R3=1
            enable_encp<='1';
            estado_siguiente<=enc_plus;
        elsif(R3=x"32") then -- R3=2
            enable_dencp<='1';
            estado_siguiente<=denc_plus;
        else
            enable_error<='1';
            estado_siguiente <= control;
        end if;
```

```

when encripter=>
    enable_enc<='1';
    encripter_ok<='1';
    estado_siguiete<=final;

when desencripter=>
    enable_denc<='1';
    desencripter_ok<='1';
    estado_siguiete<=final;

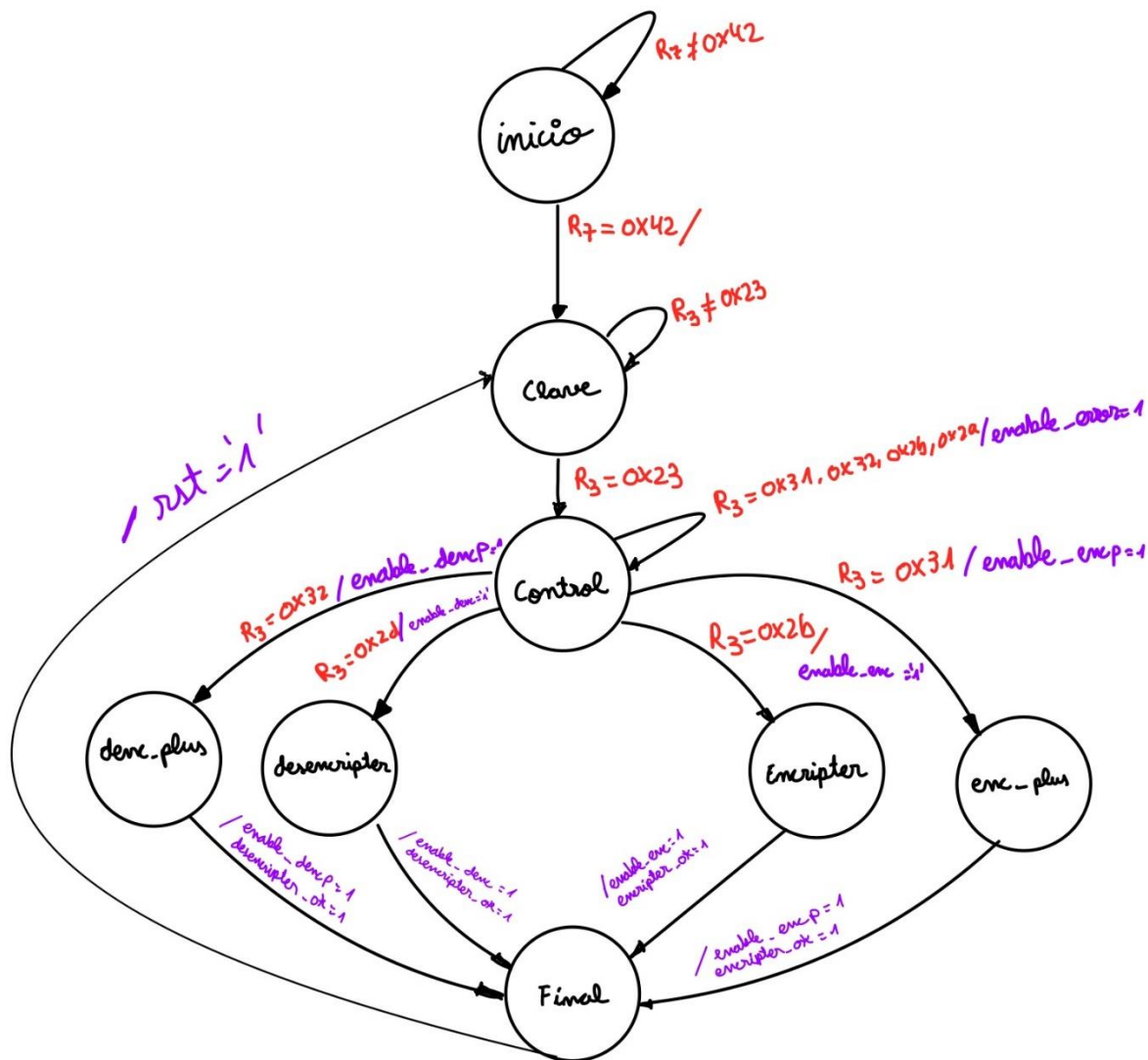
when enc_plus =>
    enable_encp<='1';
    desencripter_ok<='1';
    estado_siguiete<=final;

when denc_plus =>
    enable_encp<='1';
    desencripter_ok<='1';
    estado_siguiete<=final;

when final=>
    rst <= '1';
    estado_siguiete <=clave;

when others => estado_siguiete <= clave;

```



Generador N.º aleatorios:

Creado a base de un LFSR, es capaz de generar secuencias de bits pseudoaleatorias. En este caso, genera una secuencia de datos de 16 bits. Esta secuencia se separa en 2 datos de 8 bits y usarlos como entrada (clave) para el periférico encriptador.

El periférico está en continuo funcionamiento y tan solo cuando el usuario introduce una "r" por teclado, emplea los dos datos de 8 bits (generador pseudoaleatorio) como clave para el periférico encriptador. Esta funcionalidad proporciona al usuario una opción para evitar introducir manualmente una clave a través de teclado, permitiendo utilizar una clave de manera aleatoria.

Veamos la manera en la que hemos implementado este periférico.

Definición del periférico en el toplevel:

Primero se ha definido el periférico en el toplevel con todos los puertos de entrada y salida.

(init_z1,init_z2,init_z3 son unos valores iniciales necesarios para generar los números aleatorios)

```
component generadorAleatorio

  generic (
    init_z1      : std_logic_vector(15 downto 0) := x"EF70";
    init_z2      : std_logic_vector(15 downto 0) := x"BB56";
    init_z3      : std_logic_vector(15 downto 0) := x"A780"
  );

  port(
    clk : in std_logic;
    rst : in std_logic;
    data_out_generador_1 : out std_logic_vector(7 downto 0);
    data_out_generador_2 : out std_logic_vector(7 downto 0);
    port_id: in std_logic_vector(7 downto 0);
    write_strobe: in std_logic;
    data_in: in std_logic_vector(7 downto 0));

end component;
```

Entradas del periférico:

Nuestro periférico generador solamente tiene un registro, que escucha por el 47:

R0=>port_id = x"47"

```
---PROCESO R0  x47
process(clk, reset,rst) begin
  if(reset ='1' or rst ='1') then
    R0 <=(others =>'0');
  elsif rising_edge(clk) then
    if (port_id = x"47" and write_strobe = '1') then
      R0 <= data_in;
    end if;
  end if;
end process;
```

Salidas del periférico:

Para la salida de este periférico, hemos implementado dos salidas (cada una de 8 bits)

data_out_generador_1 : conectada a una señal de salida (data_out_aleatorio_1) en el puerto x"61".

data_out_generador_2 : conectada a una señal de salida (data_out_aleatorio_2) en el puerto x"62".

Los valores de nuestras dos salidas son dos valores totalmente pseudoaleatorios.

```
process(clk,rst) begin
    if(rst='1' ) then
        data_out_generador_1<= (others =>'0');
        data_out_generador_2<= (others =>'0');
    elsif (rising_edge(clk) and enable='1') then

        data_out_generador_1 <= data_1;

        |
        data_out_generador_2 <= data_2;

    end if;
end process;
```

Se ha definido en el toplevel una señal de salida que es la que conectará la salida del periférico con la entrada del multiplexor:

```
--señales salida del periférico
signal data_out_aleatorio_1: std_logic_vector(7 downto 0);
signal data_out_aleatorio_2: std_logic_vector(7 downto 0);
```

Procedemos a "conectar" las salidas de nuestro periférico "data_out_generador_1" y "data_out_generador_2" con las salidas que van conectadas al multiplexor.

```
generadorAleatorio1:generadorAleatorio
port map(
    data_in => outport,
    port_id => portid,
    write_strobe => writestrobe,
    clk => clk,
    rst => reset,
    data_out_generador_1 => data_out_aleatorio_1,
    data_out_generador_2 => data_out_aleatorio_2);
```

Finalmente, definimos la conexión con el multiplexor del siguiente modo:

```
import <= RAM_out when (readstrobe = '1' and portid=x"40") else
    rxbuff_out when (readstrobe = '1' and portid=x"FF") else
    data_out_encryptador when(readstrobe='1' and portid=x"50") else
    data_out_aleatorio_1 when(readstrobe='1' and portid=x"61") else
    data_out_aleatorio_2 when(readstrobe='1' and portid=x"62") else
    x"00";
```

Internamente nuestro periférico generadorAleatrio consiste en una máquina de 2 estados: inicio, fin.

Código VHDL de la máquina de estados:

```
type mis_estados is (inicio, fin);
signal actual,siguiente: mis_estados;

begin

    z1_next <= z1(9 downto 1) & (z1(15 downto 9) xor z1(8 downto 2));
    z2_next <= z2(15 downto 7) & (z2(12 downto 6) xor z2(7 downto 1));
    z3_next <= z3(8 downto 0) & (z3(6 downto 0) xor z3(14 downto 8));

    data_1<=data(7 downto 0);
    data_2<= data(15 downto 8);

    process(clk,rst) begin
        if(rst='1') then
            actual<= inicio;
        elsif rising_edge(clk) then
            actual<= siguiente;
        end if;
    end process;

    process(R0,actual) begin

        reset <='0';
        enable <='0';

        case actual is
```



```

process(clk,rst) begin
    if(rst ='1') then

        z1 <= init_z1; --inicialmente le damos los valores por defecto
        z2 <= init_z2;
        z3 <= init_z3;
    elsif rising_edge(clk) then
        data <= z1_next xor z2_next xor z3_next;
        z1 <= z1_next ; -- actualizamos
        z2 <= z2_next ;
        z3 <= z3_next ;
    end if;
end process;

```

```

process(clk,rst) begin
    if(rst='1' ) then
        data_out_generador_1<= (others =>'0');
        data_out_generador_2<= (others =>'0');
    elsif (rising_edge(clk) and enable='1') then

        data_out_generador_1 <= data_1;

        data_out_generador_2 <= data_2;

    end if;
end process;

```

```

when inicio => if(R0 =x"72") then -- si recibimos una r

        siguiente <= fin;

    else

        siguiente <= inicio;
    end if;

when fin => enable <= '1';
            reset <= '1';
            siguiente <= inicio;

```

Instrucción invertir paridad:

Se ha implementado una instrucción que altera la paridad de la clave generada. En otras palabras, si el primer dato de la clave generada por el usuario es par, esta instrucción la convierte en impar, y viceversa. Este cambio en la paridad asegura que la clave utilizada

para la encriptación no sea idéntica a la introducida inicialmente por el usuario a través del teclado (internamente).

Este proceso tiene como objetivo aumentar la seguridad del sistema, ya que la clave utilizada para la encriptación difiere de la clave original ingresada por el usuario. Sin embargo, es importante destacar que este cambio en la paridad no afecta la capacidad de desencriptar la información.

Proceso que se ha seguido para implementar la función:

Añadir la nueva línea de código para asignar un código a nuestra nueva instrucción buscando los códigos disponibles:

```
/* flip */ /* added new instruction */  
char *flip_id = "11111";  
char *tomy_id = "11101";
```

Se añade a la lista de instrucciones:

```
161 "ENABLE", /* 24 */  
162 "DISABLE", /* 25 */  
163 "CONSTANT", /* 26 */  
164 "NAMEREG", /* 27 */  
165 "ADDRESS", /* 28 */  
166 "FLIP", /* 29 */  
167 "TOMY" }; /* 30 */ /* added new instruction */
```

Aumentamos el número de instrucciones reconocibles a 31:

```
92  
93 /* increase instruction_count for added new instruction */  
94 #define instruction_count 31 /* total instruction set */  
95
```

Añadimos instrucción en el parser de chequeo de sintaxis:

```
case 30: /* TOMY */ /* added new instruction, same syntax with shift/rotate */  
if(op[i].op2 != NULL){  
    printf("ERROR - Too many Operands for %s on line %d\n", op[i].instruction, i+1);  
    fprintf(ofp, "ERROR - Too many Operands for %s on line %d\n", op[i].instruction, i+1);  
    error++;  
} else if(op[i].op1 == NULL){  
    printf("ERROR - Missing operand for %s on line %d\n", op[i].instruction, i+1);  
    fprintf(ofp, "ERROR - Missing operand for %s on line %d\n", op[i].instruction, i+1);  
    error++;  
}  
break;
```

Añadir instrucción en el parser de decodificación de código máquina:

```
    case 30: /* TOMY */ /* added new instruction */ //
insert_instruction(tomy_id, op[i].address); // inserta
if((reg_n = find_namereg(op[i].op1)) != -1) // Solo ti
    insert_sXX(reg_n, op[i].address); // inserta
else if((reg_n = register_number(op[i].op1)) != -1)
    insert_sXX(reg_n, op[i].address); // inserta
else {
    printf("ERROR - Invalid operand %s on line %d\n", op[i].op1, i+1);
    fprintf(ofp, "ERROR - Invalid operand %s on line %d\n", op[i].op1, i+1);
    error++;
}
break;
```

Ahora vamos a modificar el código VHDL de PicoBlaze:

Añadir código de operación:

```
88 -- added new instruction
89 -- flip
90 constant flip_id : std_logic_vector(4 downto 0) := "11111";
91 --tomy
92 constant tomy_id : std_logic_vector(4 downto 0) := "11101";
93 |
```

2. Declaración el componente, instanciar el componente y definir el comportamiento de la instrucción:

Declaramos el componente:

```
170 --Definition of tomy process
171 --
172 component tomy
173     Port (operand : in std_logic_vector(7 downto 0);
174           Y : out std_logic_vector(7 downto 0);
175           clk : in std_logic);
176 end component;
177
```

Instanciación del componente:

```
477
478 -- added new instruction
479 flip_group: flip
480 port map (operand => sX_register,
481           Y => flip_result,
482           clk => clk);
483
484 tomy_group: tomy
485 port map (operand => sX_register,
486           Y => tomy_result,
487           clk => clk);
```

Definimos nuestra instrucción, añadiendo un nuevo archivo VHDL.

```
process(clk)
begin
    if (rising_edge(clk) ) then
        if(operand(0) = '1') then
            Y <= operand +1;
        else
            Y <= operand -1;
        end if;
    end if;
end process;
```

3. Declaramos nuevas señales de control para que, cuando el procesador reconozca el código de nuestra instrucción active una señal de control:

```
-- added new instruction
signal i_flip : std_logic;
signal i_tomy : std_logic;
```

4. Después habilitamos las señales de flag del banco de registros:

```
--
entity register_and_flag_enable is
    Port (i_logical: in std_logic;
          i_arithmetic: in std_logic;
          i_shift_rotate: in std_logic;
          i_flip: in std_logic; -- added new instruction
          i_tomy: in std_logic;
          i_returni: in std_logic;
          i_input: in std_logic;
          active_interrupt : in std_logic;
          T_state : in std_logic;
          register_enable : out std_logic;
          flag_enable : out std_logic;
          clk : in std_logic);
end register_and_flag_enable;
```

5. Añadimos la siguiente línea donde se instancia:

```
reg_and_flag_enables: register_and_flag_enable
Port map (i_logical => i_logical,
          i_arithmetic => i_arithmetic,
          i_shift_rotate => i_shift_rotate,
          i_flip => i_flip, -- added new instruction
          i_tomy=> i_tomy,
          i_returni => i_returni,
          i_input => i_input,
          active_interrupt => active_interrupt,
          T_state => T_state,
          register_enable => register_write_enable,
```

6. Añadimos la siguiente línea donde se instancia:

```
entity register_and_flag_enable is
  Port (i_logical: in std_logic;
        i_arithmetic: in std_logic;
        i_shift_rotate: in std_logic;
        i_flip: in std_logic;           -- added new instruction
        i_tomy: in std_logic;
        i_returni: in std_logic;
        i_input: in std_logic;
        active_interrupt : in std_logic;
        T_state : in std_logic;
        register_enable : out std_logic;
        flag_enable : out std_logic;
        clk : in std_logic);
end register_and_flag_enable;
```

Finalmente, añadimos a la ALU la salida de nuestra instrucción:

```
-- get ALU result
--
ALU_loop: for i in 0 to 7 generate
begin
  ALU_result(i) <= (shift_and_rotate_result(i) and i_shift_rotate)
    or (in_port(i) and i_input)
    or (arithmetic_result(i) and i_arithmetic)
    or (flip_result(i) and i_flip)       -- added new instruction
    or (tomy_result(i) and i_tomy)
    or (logical_result(i) and i_logical);
end generate ALU_loop;
```

ASM:

Partiendo de la idea de que el ASM es quien permite la conexión con el PicoBlaze, hemos definido en él una serie de cadenas de caracteres que hacen la interacción con el dispositivo más personalizada.

También se han hecho una serie de verificaciones para el correcto funcionamiento de los periféricos.

Pedir_Clave=> cadena de caracteres que se imprime por pantalla para pedirle una clave al usuario.

Pedir_dato=> cadena de caracteres que se imprime por pantalla para pedirle un dato al usuario.

SALIDA=> cadena de caracteres que se imprime por pantalla para darle la salida al usuario.

Retorno=> cadena de retorno de carro para la legibilidad de la interacción.

```

start:
        CALL      recibe
        OUTPUT    rxreg,47
        LOAD      txreg,rxreg
        CALL      transmite      ; para imprimir por pantalla el primer dato que envia el usuario

        LOAD      s0,rxreg      ;guardo lo que he recibido en el s0, porque rxreg se machaca con la resta
        SUB       rxreg,2a      ;si recibimos un * por el 47
        JUMP      NZ, clave     ; si hemos recibido algo distinto de * saltamos a la rutina clave:

```

(Verificamos si se ha introducido un “*” ,una “r” , o algo distinto;según estas entradas, enviaremos la información a periférico u otro).

```

CALL      recibe
LOAD      s6,rxreg
TOMY      rxreg
OUTPUT    rxreg,41
LOAD      txreg,s6
CALL      transmite

```

```

CALL      recibe
LOAD      s6,rxreg
TOMY      rxreg
OUTPUT    rxreg,42
LOAD      txreg,s6
CALL      transmite

```

```

CALL      retorno

```

(Recibimos un 2 datos por teclado, le aplicamos TOMY, lo transmitimos por pantalla y enviamos al puerto 41 y 42 Como hemos visto aquí llegamos solo cuando hemos recibido *)

DATO:

```
CALL    retorno

CALL    Pedir_introduce
CALL    Pedir_dato

CALL    recibe                                ;Dato
OUTPUT  rxreg,43
LOAD    txreg,rxreg
CALL    transmite
```

(Pedimos el dato para encriptar)

```
CALL    recibe
OUTPUT  rxreg,44
LOAD    txreg,rxreg
CALL    transmite
```

```
CALL    recibe
OUTPUT  rxreg,44
LOAD    txreg,rxreg
CALL    transmite
```

```
CALL    retorno
CALL    SALIDA
```

```
INPUT    txreg,50

CALL    transmite

CALL    retorno

JUMP    start
```

(Recibimos en primer lugar un #, luego tenemos que recibir un +,-,1 o 2 y transmitimos lo que esta en el puerto 50 que es el valor que nos proporciona el periférico encriptador)

```

clave:                                     ;
SUB          s0,72                        ;
JUMP        NZ,clave
CALL        retorno
CALL        Pedir_introduce
CALL        Pedir_Clave

INPUT        txreg,61
LOAD        s6,txreg
TOMY        txreg
OUTPUT       txreg,41

LOAD        txreg,s6
CALL        transmite

INPUT        txreg,62
LOAD        s6,txreg
TOMY        txreg
OUTPUT       txreg,42

LOAD        txreg,s6
CALL        transmite

CALL        DATO

```

(Un bucle que va comprobando si lo que esta en el s0 es una "r" (0x72), si es asi cogemos dos datos del periférico generador aleatorio, les aplicamos TOMY y se autocompleta los valores de la clave, además de enviarle estos valores al periférico encriptador por el puerto 41 y 42)

Funcionamiento en la practica:

Hay dos modos de funcionamiento como podemos ver en el siguiente esquema:

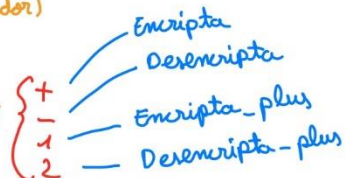
Hay dos tipos de funcionamiento: (*) o (r)

(*) Entrar al periférico (Encriptador)

Introduce clave: (xx)

Introduce dato: (x) #

Tu dato es: (x)



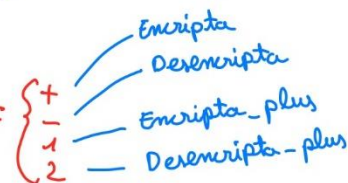
- Salida del periférico
- Entrada por teclado del usuario
- Cadena de texto ASCII

(r) Entrar al periférico (Nº aleatorio)

Introduce clave: (xx)

Introduce dato: (x) #

Tu dato es: (x)



A continuación, imprime por pantalla la cadena de caracteres "Introduce dato:", el usuario ingresa el dato a encriptar (1 carácter ascii), seguido de una # y dependiendo del tipo de operación pulsaría por teclado +,-,1 o 2.

Funcionamiento clave aleatoria

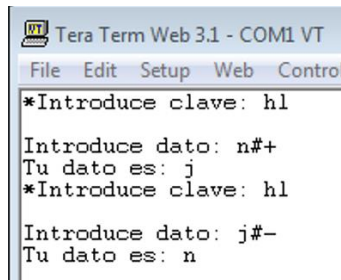
Para seleccionar este funcionamiento basta con teclear un (r).

Seguido de "Introduce clave:" se autocompleta los dos caracteres de la clave imprimiéndose así por pantalla. A continuación, imprime por pantalla la cadena de caracteres "Introduce dato:", el usuario ingresa el dato a encriptar (1 carácter ascii), seguido de una # y dependiendo del tipo de operación pulsaría por teclado +,-,1 o 2.

Todo esto es mas sencillo si ponemos ejemplos y lo visualizamos.

Para comprobar el correcto funcionamiento hemos realizado las mismas comprobaciones sin hacer uso de la instrucción y haciendo uso de ella. Para la misma clave y dato nos debería de dar resultados distintos.

Ejemplo 1. Sin instrucción funcionamiento manual:



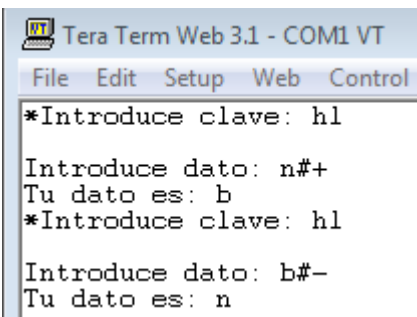
```
Tera Term Web 3.1 - COM1 VT
File Edit Setup Web Control
*Introduce clave: h1
Introduce dato: n#+
Tu dato es: j
*Introduce clave: h1
Introduce dato: j#-
Tu dato es: n
```

$h = 68 ; l = 6C \Rightarrow \text{clave} = 68 \text{ XOR } 6C = 4$

$n = 6E ; (+) \text{ Encriptamos} \Rightarrow n \text{ XOR clave} = 6A (j)$

Proceso inverso para desencriptar.

Ejemplo 1. Instrucción funcionamiento manual:



```
Tera Term Web 3.1 - COM1 VT
File Edit Setup Web Control
*Introduce clave: h1
Introduce dato: n#+
Tu dato es: b
*Introduce clave: h1
Introduce dato: b#-
Tu dato es: n
```

$h = 68 \text{ PAR } (-1) ; l = 6C \text{ PAR } (-1) \Rightarrow \text{clave} = 67 \text{ XOR } 6B = C$

$n = 6E ; (+) \text{ Encriptamos} \Rightarrow n \text{ XOR clave} = 62 (b)$

Proceso inverso para desencriptar

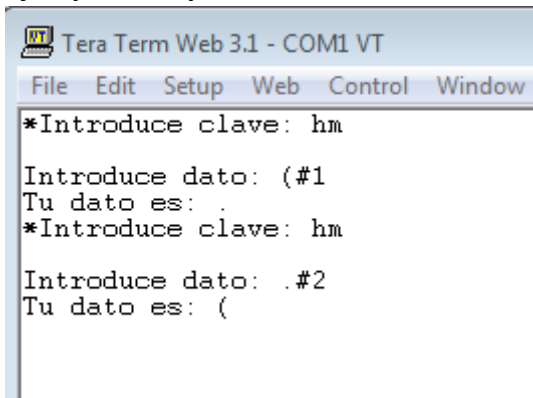
Como podemos salen resultados distintos, es decir, la instrucción esta funcionando correctamente. Sin embargo, no tenemos habilitada la opción de no usar la instrucción, esto era una simple comprobación.

Ejemplo 2. Funcionamiento manual Encripta_plus :

En este caso se realiza una operación con la clave distinta, siendo la clave los 4 bits mas significativos del primer dato concatenados con los 4 bits menos significativos del 2 dato introducido por el usuario. Como podemos observar introduciendo misma clave y en el primer caso el dato a encriptar y en el segundo el dato encriptado para sacar así el dato original.

```
*Introduce clave: a)
Introduce dato: g#1
Tu dato es: e
*Introduce clave: a)
Introduce dato: e#2
Tu dato es: g
```

Ejemplo 3. Repetido funcionamiento manual Encripta_plus:

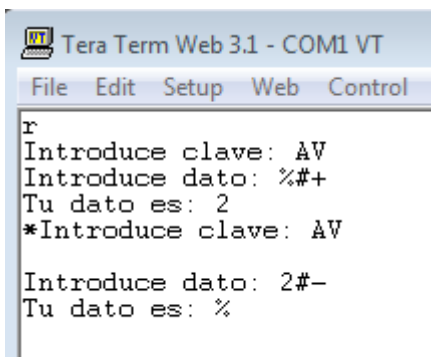


```
Tera Term Web 3.1 - COM1 VT
File Edit Setup Web Control Window
*Introduce clave: hm
Introduce dato: (#1
Tu dato es: .
*Introduce clave: hm
Introduce dato: .#2
Tu dato es: (
```

Como podemos observar en este ejemplo, nos genera un dato encriptado ".", que usamos luego para desencriptar.

Ejemplo 4. Funcionamiento clave aleatoria:

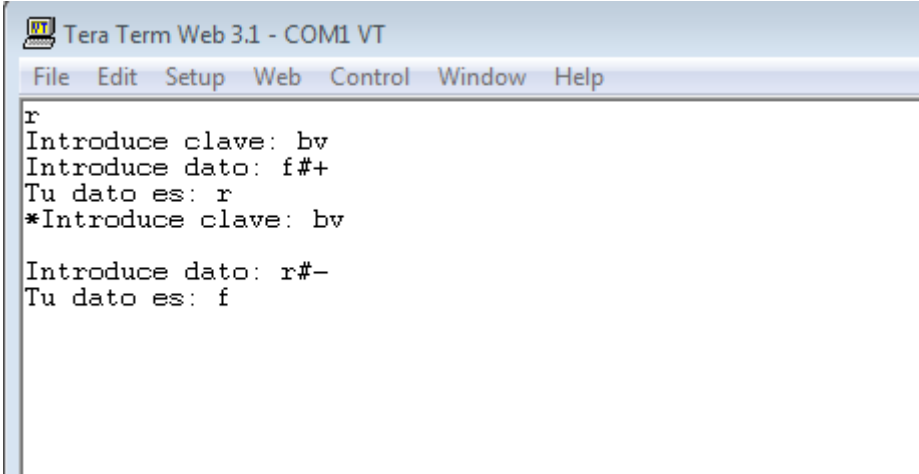
Tras pulsar "r" entramos en este modo de funcionamiento y se autocompleta la clave. Esto conviene verlo mejor en la práctica.



```
Tera Term Web 3.1 - COM1 VT
File Edit Setup Web Control
r
Introduce clave: AV
Introduce dato: %#+
Tu dato es: 2
*Introduce clave: AV
Introduce dato: 2#-
Tu dato es: %
```

Tras tener el dato encriptado, lo desencriptamos de manera manual.

Ejemplo 5. Repetido funcionamiento clave aleatoria:



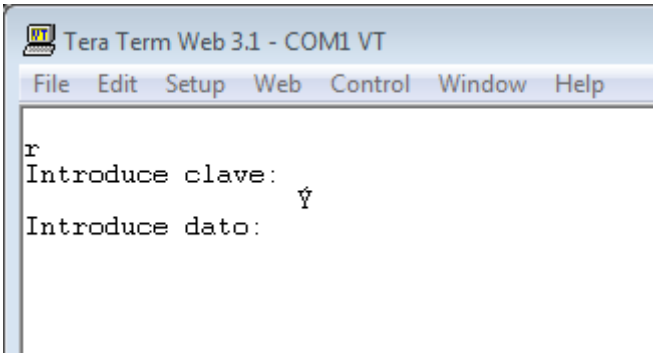
```
Tera Term Web 3.1 - COM1 VT
File Edit Setup Web Control Window Help

r
Introduce clave: bv
Introduce dato: f#+
Tu dato es: r
*Introduce clave: bv

Introduce dato: r#-
Tu dato es: f
```

Ejemplo 6. Caracteres especiales.

Con este ejemplo queremos tratar el caso en el que cuando se genera una clave aleatoria esta genere un carácter especial de la tabla ASCII, como puede ser el retorno de carro. Como podemos observar, se *autocompleta* el campo de la clave con un retorno de carro y el carácter “Y”.



```
Tera Term Web 3.1 - COM1 VT
File Edit Setup Web Control Window Help

r
Introduce clave:
Introduce dato: Y
```