# Dividers

April 26, 2024

## 1 Notación

Asumiremos que un vector binario $x$ tiene la siguiente forma:

| | | | | | | |
|---|---|---|---|---|---|---|
| $x_{B-1}$ | $x_{B-2}$ | $\cdots$ | $x_3$ | $x_2$ | $x_1$ | $x_0$ |

$$\begin{aligned} q &= n/d \\ n &= q \cdot d + r \end{aligned} \quad (1)$$

donde

- $q$: quotient
- $n$: numerador
- $d$: denominador
- $r$: remainder, debe cumplir la condición $0 \leq r < d$

Siempre la hipótesis será $n > d > 0$ (sin signo US). El caso de signado (2C) queda para el alumno.

## 2 Counter based divider

- Se inicia el registro $Q$ en 0.
- Se inicia el registro $N$ con el numerador.
- Se inicia el registro $D$ en 0.

```
while N ≥ D {
 N = N - D
 Q = Q + 1
}
```

La cantidad de pasos depende de los datos de entrada $n$, $d$. En el peor de los casos, debe contar tantas veces como la cantidad de bits de palabra.

## 3 Binary search divider

- Se inicia el registro $L$ en 0.
- Se inicia el registro $H$ con el numerador.

```
while (H > L+1) {
    mid = floor(L+H+1)/2
    if (mid*D > H) {
        H = mid
    } elsif (mid*D < L) {
        L = mid
    }
}
Q = L
```

**Ejemplo:**

N = 42, D = 8

1. $L = 0, H = 42$
$m = (L + H + 1)/2 = (0 + 42 + 1)/2 = 43/2 = 21$
$m * D = 21 * 8 = 168 \rightarrow L = 0, H = 21$

2. $L = 0, H = 21$
$m = (L + H + 1)/2 = (0 + 21 + 1)/2 = 22/2 = 11$
$m * D = 11 * 8 = 88 \rightarrow L = 0, H = 11$

3. $L = 0, H = 11$
$m = (L + H + 1)/2 = (0 + 11 + 1)/2 = 12/2 = 6$
$m * D = 6 * 8 = 48 \rightarrow L = 0, H = 6$

4. $L = 0, H = 6$
$m = (L + H + 1)/2 = (0 + 6 + 1)/2 = 7/2 = 3$
$m * D = 3 * 8 = 24 \rightarrow L = 3, H = 6$

5. $L = 3, H = 6$
$m = (L + H + 1)/2 = (3 + 6 + 1)/2 = 10/2 = 5$
$m * D = 5 * 8 = 40 \rightarrow L = 5, H = 6$

6. Como $H$ no es mayor que $L + 1$ entonces $Q = 5$.

Desventajas: Usa un multiplicador.

Ventajas: Es más rápido que el anterior. Si $B$ es la cantidad de bits de palabra, en el peor de los casos toma $log_2(B)$ pasos.

# 4 Long division

Long division comprende todos aquellos métodos de división en los cuales el cociente se va hallando dígito por dígito.

**Ejemplo:**
En base $b = 10$, se divide 7547/31.

| | 7 | 5 | 4 | 7 | 31 |
|---|---|---|---|---|---|
| | 7 | 5 | 4 | 7 | 0 |
| - | 0 | 0 | 0 | 0 | |
| | 7 | 5 | 4 | 7 | 02 |
| - | 6 | 2 | 0 | 0 | |
| | 1 | 3 | 4 | 7 | 024 |
| - | 1 | 2 | 4 | 0 | |
| | | 1 | 0 | 7 | 0243 |
| - | | | 9 | 3 | |
| | | | 1 | 4 | |

Sea $R^0 = N$, entonces

$$\begin{array}{rcl} R^1 & = & R^0 - q[3] \times D \times 10^3 \\ R^2 & = & R^1 - q[2] \times D \times 10^2 \\ R^3 & = & R^2 - q[1] \times D \times 10^1 \\ R & = & R^3 - q[0] \times D \times 10^0 \end{array}$$

Luego,

$$R = N - D \times \sum_{i=0}^{B-1} q_i * 10^{B-1-i}$$

En general para un sistema numérico en base $b$ se tendrá

$$R = N - D \times \sum_{i=0}^{B-1} q_i * b^{B-1-i} \tag{2}$$

$$R^{i+1} = R^i - q^{B-1-i} \times D \times b^{B-1-i} \tag{3}$$

Diferentes condiciones se pueden plantear para elegir los coeficientes $q_i$ en función de los restos parciales $R^i$:

- Restoring division: se toma $q^{B-1-i}$ tal que $0 \leq R^{i+1} < D \times b^{B-1-i}$
- Non-restoring division: se toma $q^{B-1-i}$ tal que $|R^{i+1}| < D \times b^{B-1-i}$
- SRT division: $|R^{i+1}| \leq k \times D \times b^{B-1-i}$, $1/2 \leq k \leq 1$

# 5 Restoring division

Sea $B$ la cantidad de bits de palabra. Como la base es $b = 2$, la ecuación de recurrencia se convierte:

$$R^{i+1} = R^i - q^{B-1-i} \times D \times 2^{B-1-i} \tag{4}$$

De la condición de restoring division:

$$0 \leq R^{i+1} = R^i - q^{B-1-i} \times D \times 2^{B-1-i} < D \times b^{B-1-i} \tag{5}$$

Como $q^{B-1-i}$ sólo puede ser 0 o 1 ($b = 2$), entonces

- Si $R^i < D \times b^{B-1-i}$, entonces $q^{B-1-i} = 0$.
- Si $R^i \geq D \times b^{B-1-i}$, entonces $q^{B-1-i} = 1$.

**Algoritmo:**

```
R = 0
for i=0,...,B-1 {
    {R,N} = {R,N} << 1
    if (R<D) {
        N[0] = 0
        R = R   <--- Restoring step
    } else {
        N[0] = 1
        R = R - D
    }
}

Q = N
```

**Ejemplo:**

$N = 14$ (1110), $D = 3$ (0011), $B = 4$ bits
Se inicializa $R = 0000$, entonces $\{R, N\} = \{0000, 1110\}$

$i = 0$.
$\{R, N\} = \{R, N\} << 1 = \{0001, 1100\}$
$R < D$, entonces $R = R = 0001$ (restoring), $N[0] = 0$.
$\{R, N\} = \{0001, 1100\}$

$i = 1$.
$\{R, N\} = \{R, N\} << 1 = \{0011, 1000\}$
$R = D$, entonces $R = R - D = 0000$, $N[0] = 1$.
$\{R, N\} = \{0000, 1001\}$

$i = 2$.
$\{R, N\} = \{R, N\} << 1 = \{0001, 0010\}$
$R < D$, entonces $R = R = 0001$ (restoring), $N[0] = 0$.
$\{R, N\} = \{0001, 0010\}$

$i = 3$.
$\{R, N\} = \{R, N\} << 1 = \{0010, 0100\}$
$R < D$, entonces $R = R = 0010$ (restoring), $N[0] = 0$.
$\{R, N\} = \{0010, 0100\}$

Luego, $Q = N = 0100 = 4$, $R = 0010 = 2$.

## 5.1 Implementación Ejemplo

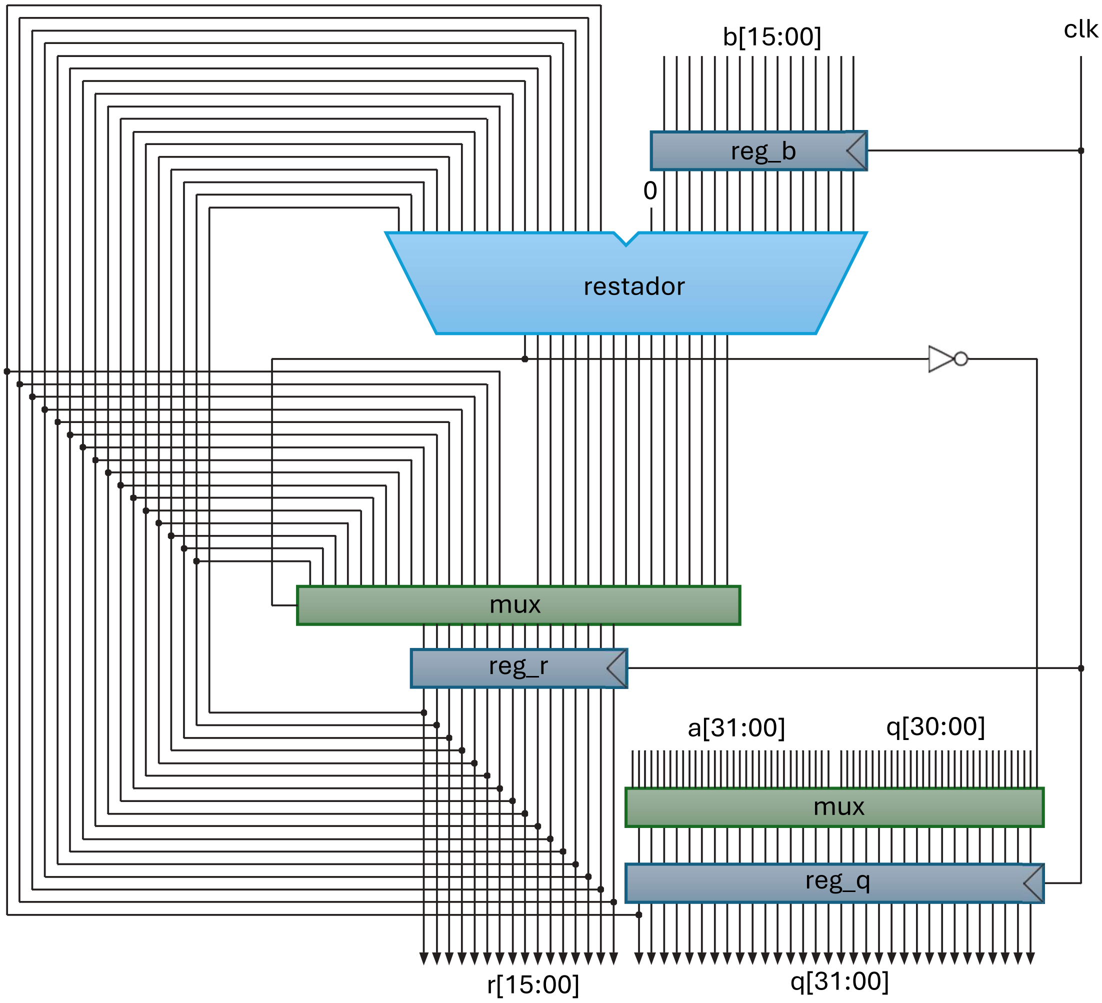


Figure 1: Esquemático Restoring Divider

- $a = b \times q + r$ y $r < b$
- Hay 3 registros:
  - reg_b: Divisor $b$
  - reg_r: Resto $r$
  - reg_q: Cociente $q$. Inicialmente almacena el dividendo $a$
- Se utiliza un restador para restar $b$ del *resto parcial*. El MSB de la salida del restador se utiliza para determinar si el resultado de la resta es negativo o no.
- El Multiplexor sobre reg_q se usa para cargar inicialmente $a$ y para realizar el shift left de reg_q (a y b). Este multiplexor implementa el restoring.
  - Si el resultado de la resta es negativo, selecciona el resto parcial original.
  - De lo contrario, selecciona el resultado de la resta.
- En cada iteración se obtiene un bit de $q$ del bit de signo del resultado del restador y se escribe en el LSB de reg_q.

### 5.2 Código HDL

```verilog
module div_restoring (
    input  wire [31:0] a,      // dividend
    input  wire [15:0] b,      // divisor
    input  wire        start,  // start
    input  wire        clk,    // clk
    input  wire        rst_n,  // reset
    output wire [31:0] q,      // quotient
    output wire [15:0] r,      // remainder
    output reg         busy,   // busy
    output reg         ready,  // ready
    output reg    [4:0] count  // counter
);

// Internal
reg  [31:0] reg_q;
reg  [15:0] reg_r;
reg  [15:0] reg_b;
wire [16:0] sub_out;
wire [15:0] mux_out;

// Restador
assign sub_out = {reg_r, reg_q[31]} - {1'b0, reg_b};

// restoring
assign mux_out = sub_out[16] ? {reg_r[14:0], reg_q[31]}
                             : sub_out[15:0];  // or not
assign q = reg_q;
assign r = reg_r;

always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        busy  <= 0;
        ready <= 0;
    end else begin
        if (start) begin
            reg_q <= a;  // load a
            reg_b <= b;  // load b
            reg_r <= 0;
            busy  <= 1;
            ready <= 0;
            count <= 0;
        end else if (busy) begin
            reg_q <= {reg_q[30:0], ~sub_out[16]};  // << 1
            reg_r <= mux_out;
            // counter++
            count <= count + 5'd1;
            // finished
            if (count == 5'h1f) begin
                busy  <= 0;
                ready <= 1;  // q,r ready
            end
        end
    end
end
endmodule
```

Listing 1: Restoring Divisor

"Desenrrollando" el circuito sequencial, se logra una implementación combinacional.

## 6 Non-restoring division

Sea $B$ la cantidad de bits de palabra. Como la base es $b = 2$, la ecuación de recurrencia se convierte:

$$R^{i+1} = R^i - q^{B-1-i} \times D \times 2^{B-1-i} \tag{6}$$

Condición de non-restoring division: $q^{B-1-i}$ tal que $|R^{i+1}| < D \times 2^{B-1-i}$
donde $q^i \in \{-1, 1\}$.

Entonces,

$$-D \times 2^{B-1-i} < R^{i+1} < D \times 2^{B-1-i}$$

Es decir,

$$\begin{array}{rcccl} -D \times 2^{B-1-i} & < & R^i - q^{B-1-i} \times D \times 2^{B-1-i} & < & D \times 2^{B-1-i} \\ (q^{B-1-i} - 1) \times D \times 2^{B-1-i} & < & R^i & < & (q^{B-1-i} + 1) \times D \times 2^{B-1-i} \end{array}$$

- Si $0 < R^i < D \times 2^{B-i}$, entonces $q^{B-1-i} = 1$.
- Si $-D \times 2^{B-i} < R^i < 0$, entonces $q^{B-1-i} = -1$.

**Ejemplo:**
En base $b = 10$, calculamos $273/3$.

| | | | | |
|---|---|---|---|---|
| | 2 | 7 | 3 | 3 |
| | 2 | 7 | 3 | 1 |
| - | 3 | 0 | 0 | |
| | | -2 | 7 | $1\bar{1}$ |
| - | | -3 | 0 | |
| | | | 3 | $1\bar{1}1$ |
| - | | | 3 | |
| | | | 0 | |

El resultado es $1\bar{1}1$. Como $\bar{1}1 = 9$, entonces el resultado es 91.

**Ejemplo:**
Mismo ejemplo anterior, en base $b = 2$, calculamos $273/3 = 100010001/11$.

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 11 |
| | | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 |
| - | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| | -1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | $1\bar{1}$ |
| - | | -1 | -1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| | | | -1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | $1\bar{1}\bar{1}$ |
| - | | | -1 | -1 | 0 | 0 | 0 | 0 | 0 | 0 | |
| | | | | 1 | 0 | 1 | 0 | 0 | 0 | 1 | $1\bar{1}\bar{1}1$ |
| - | | | | 1 | 0 | 1 | 0 | 0 | 0 | 0 | |
| | | | | | -1 | 1 | 0 | 0 | 0 | 1 | $1\bar{1}\bar{1}1\bar{1}$ |
| - | | | | | -1 | -1 | 0 | 0 | 0 | 0 | |
| | | | | | 1 | 0 | 0 | 0 | 0 | 1 | $1\bar{1}\bar{1}1\bar{1}1$ |
| - | | | | | | 1 | 1 | 0 | 0 | 0 | |
| | | | | | | | 1 | 0 | 0 | 1 | $1\bar{1}\bar{1}1\bar{1}11$ |
| - | | | | | | | 1 | 1 | 0 | 0 | |
| | | | | | | | | -1 | 0 | 1 | $1\bar{1}\bar{1}1\bar{1}11\bar{1}$ |
| - | | | | | | | | -1 | -1 | 0 | |
| | | | | | | | | | 1 | 1 | $1\bar{1}\bar{1}1\bar{1}11\bar{1}1$ |
| - | | | | | | | | | 1 | 1 | |
| | | | | | | | | | | 0 | |

El resultado es $1\bar{1}\bar{1}1\bar{1}11\bar{1}1 = 256 - 128 - 64 + 32 - 16 + 8 + 4 - 2 + 1 = 91$.

Observar que el cociente queda expresado en SD. Si esto no es deseado, se puede reemplazar: $1\bar{1}\ldots\bar{1} = 0\ldots1$. En el ejemplo anterior, $1\bar{1}\bar{1}1\bar{1}11\bar{1}1 = 001011011 = 91_{10}$.

**Algoritmo:**
El registro $R$ tiene $B + 1$ bits.

```
R = 0
for i=0,...,B-1 {
    {R,N} = {R,N} << 1
    if (R<D) {
        R = R + D
    } else {
        R = R - D
    }

    if (R<0) {
        N[0] = 0
    } else {
        N[1] = 1
    }
}

Q = N

if (R<0) {
    R = R + D
} else {
    R = R - D
}
```

**Ejemplo:**

$N = 14$ (1110), $D = 3$ (0011), $B = 4$ bits
Se inicializa $R = 00000$ ($B + 1$ bits), entonces $\{R, N\} = \{00000, 1110\}$

$i = 0$.
$\{R, N\} = \{R, N\} << 1 = \{00001, 1100\}$
$R \geq 0$, entonces $R = R - D = 11110$,
Como $R < 0$, entonces $N[0] = 0$.
$\{R, N\} = \{11110, 1100\}$

$i = 1$.
$\{R, N\} = \{R, N\} << 1 = \{11101, 1000\}$
$R < 0$, entonces $R = R + D = 00000$,
Como $R \geq 0$, entonces $N[0] = 1$.
$\{R, N\} = \{00000, 1001\}$

$i = 2$.
$\{R, N\} = \{R, N\} << 1 = \{00001, 0010\}$
$R \geq 0$, entonces $R = R - D = 11110$,
Como $R < 0$, entonces $N[0] = 0$.
$\{R, N\} = \{11110, 0010\}$

$i = 3$.
$\{R, N\} = \{R, N\} << 1 = \{11100, 0100\}$
$R < 0$, entonces $R = R + D = 11111$,
Como $R < 0$, entonces $N[0] = 0$.
$\{R, N\} = \{11111, 0100\}$

Finalmente, como $R < 0$, $R = R + D = 00010$.
Luego, $Q = N = 0100 = 4$, $R = 0010 = 2$.

## 6.1 Implementación Ejemplo

- No hay un multiplexor sobre el registro *reg_r* (utilizado en el caso anterior para el restoring)
- En lugar de un restador, en este algoritmo requiere un componente capaz de sumar y restar (add/sub)
- Dado que el contenido del registro *reg_r* puede ser negativo en la última iteración, se necesita un sumador y un multiplexor para ajustar el *resto* final. Si $r$ no es negativa es el valor final. En caso contrario, se debe restaurar el resto sumandole b a r, tal como se hizo en el algoritmo anterior.
  - Si no se utiliza la salida de resto, puede eliminarse toda esta parte del circuito.

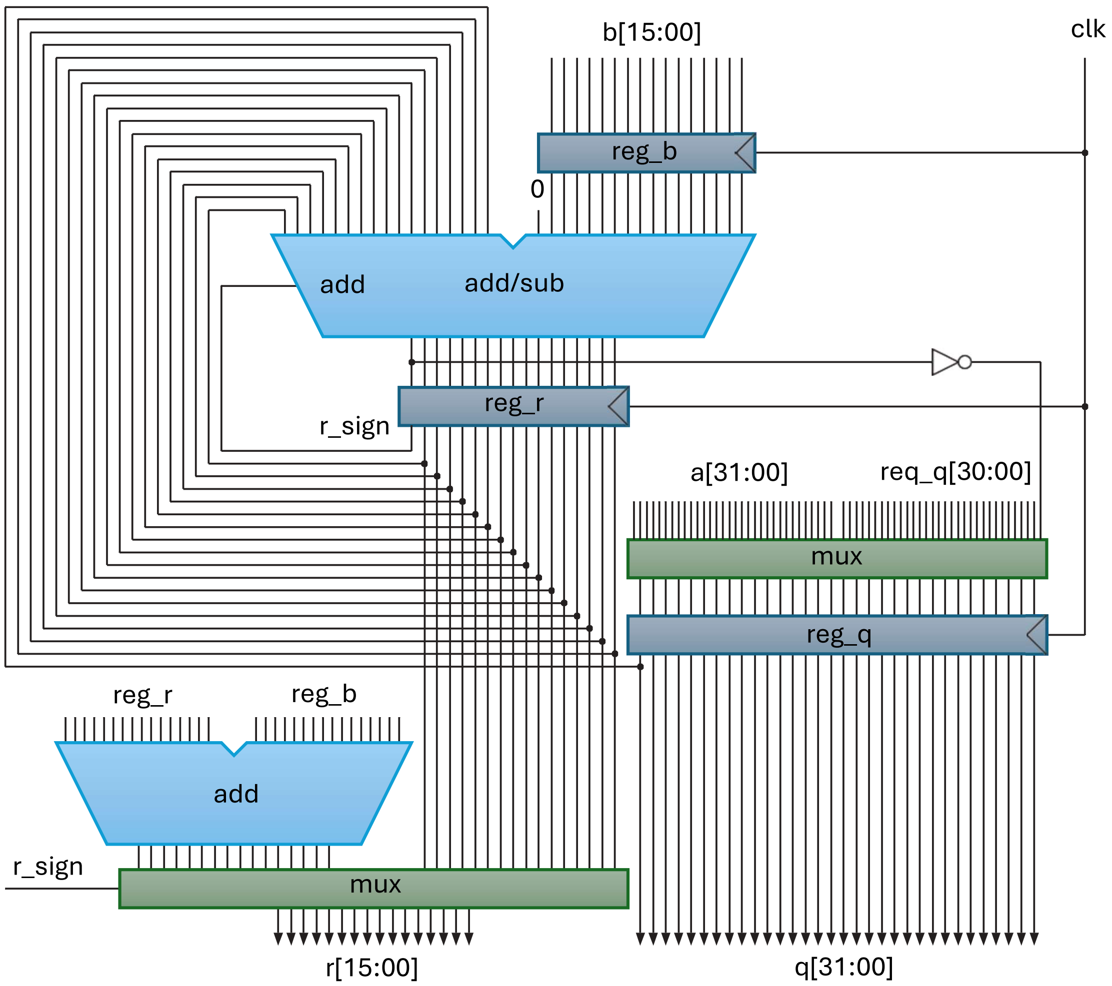


Figure 2: Non-Restoring Divider

## 6.2 Código HDL

```verilog
module div_nonrestoring (
  input  wire [31:0] a,      // dividend
  input  wire [15:0] b,      // divisor
  input  wire        start, // start
  input  wire        clk,    // clk
  input  wire        rst_n, // reset
  output wire [31:0] q,      // quotient
  output wire [15:0] r,      // remainder
  output reg         busy,  // busy
  output reg         ready, // ready
  output reg  [4: 0] count  // count
);
    // Internal
    reg  [31:0] reg_q;
    reg  [15:0] reg_r;
    reg  [15:0] reg_b;
    wire [16:0] sub_add;

    assign sub_add = reg_r[15]?
                       {reg_r,reg_q[31]} + {1'b0,reg_b} : // + b
                       {reg_r,reg_q[31]} - {1'b0,reg_b};  // - b
    assign q = reg_q;
    assign r = reg_r[15]? reg_r + reg_b : reg_r; // adjust r

    always @ (posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            busy  <= 0;
            ready <= 0;
        end else begin
            if (start) begin
                reg_q <= a; // load a
                reg_b <= b; // load b
                reg_r <= 0;
                busy  <= 1;
                ready <= 0;
                count <= 0;
            end else if (busy) begin
                reg_q <= {reg_q[30:0], ~sub_add[16]}; // << 1
                reg_r <= sub_add[15:0];
                count <= count + 5'b1; // count++
                if (count == 5'h1f) begin
                    busy  <= 0;
                    ready <= 1; // q,r ready
                end
            end
        end
    end
endmodule
```

Listing 2: Non-Restoring Divisor

"Desenrrollando" el circuito sequencial, se logra una implementación combinacional.

# 7 Radix $2^k$ non-restoring divider

Queda para el alumno.

# 8 SRT Divider

Nombrado SRT debido a sus autores que lo dedujeron de forma independiente y casi simultáneamente: D. W. Sweeney (IBM, February 1957), James E. Robertson (University of Illinois, September 1958), K. D. Tocher (Imperial College London, January 1958).

SRT es similar a non-restoring division, pero usa una Lookup Table para determinar cada dígito del cociente.

Queda para el alumno.

# 9 Goldschmidt Division

## 9.1 Introducción

- Dados $a$ y $b$ en formato punto fijo $0.1x\cdots x$, siendo:

$$\tfrac{1}{2} \leq a \text{ y } b < 1$$

El algoritmo de Goldschmidt utiliza multiplicaciones iterativas para obtener $q = \frac{a}{b}$

- Consideramos la fracción:

$$\frac{a \times r_0 \times r_1 \times r_2 \times \cdots \times r_{n-1}}{b \times r_0 \times r_1 \times r_2 \times \cdots \times r_{n-1}} \tag{7}$$

- Si el denominador converge a 1, entonces el numerador converge a $\frac{a}{b} = q$
- El factor $r_i$ puede ser calculado de la siguiente manera:
  - Se define $\delta = 1 - b$
  - Luego $0 < \delta \leq \frac{1}{2}$ y $b = 1 - \delta$
  - Definimos $x_0 = a$, $y_0 = b = 1 - \delta$
  - Calculamos:

$$\begin{array}{rclclcl}
r_0 & = & 2 - y_0 & = & 1 + \delta & & \\
x_1 & = & x_0 \times r_0 & & & & \\
y_1 & = & y_0 \times r_0 & = & (1-\delta) \times (1+d) & = & 1 - \delta^2 \\
r_1 & = & 2 - y_1 & = & 1 + \delta^2 & & \\
x_2 & = & x_1 \times r_1 & & & & \\
y_2 & = & y_1 \times r_1 & = & (1-\delta^2) \times (1+d^2) & = & 1 - \delta^4 \\
 & \cdots & & \cdots & & \cdots & \\
r_{i-1} & = & 2 - y_{i-1} & = & 1 + \delta^{2^{i-1}} & & \\
x_i & = & x_{i-1} \times r_{i-1} & & & & \\
y_i & = & y_{i-1} \times r_{i-1} & = & (1-\delta^{2^{i-1}}) \times (1+\delta^{2^{i-1}}) & = & 1 - \delta^{2^i} \\
 & \cdots & & \cdots & & \cdots & \\
r_{n-1} & = & 2 - y_{n-1} & = & 1 + \delta^{2^{n-1}} & & \\
x_n & = & x_{n-1} \times r_{n-1} & & & & \\
y_n & = & y_{n-1} \times r_{n-1} & = & (1-\delta^{2^{n-1}}) \times (1+\delta^{2^{n-1}}) & = & 1 - \delta^{2^n}
\end{array} \tag{8}$$

    Hasta que $y_n$ converge a 1, luego $x_n$ converge a $q$
  - La velocidad de convergencia de $y_n \longrightarrow 1$ depende de $\delta$. El caso más lento es cuando $\delta = \frac{1}{2}$
  - Por qué $y_n \longleftarrow 1$? Debido a que $y_n = 1 - \delta^{2^n}$ siendo $0 < \delta \leq \frac{1}{2}$

Falta demostrar la convergencia del algoritmo. Leer bibliografía.

Este algoritmo puede ser implementado en Python de la siguiente forma:

```python
def goldschmidt_division(a, b, iterations=5):
    # Condiciones de contorno
    if not (0.5 <= b < 1):
        raise ValueError("Variable b debe cumplir: 0.5 <= b < 1 en Goldschmidt")

    # Inicializamos
    delta = 1 - b
    x = a
    y = b

    # Calcula r0
    r = 1 + delta

    # Refinamos iterativamente
    for i in range(iterations):
        x *= r
        y *= r
        r = 2 - y
        print(f"Iteracion {i + 1}:")
        print(f"  x = {x}")
        print(f"  y = {y}")
        print(f"  r = {r}\n")

        # Terminamos si y converge a 1
        if y > 0.999 and y < 1.001:
            break

    return x

# Ejemplo
result = goldschmidt_division(0.85, 0.5)
print(f"Final result: {result}")
```

Listing 3: Python Goldschmidt

```
Iteracion 1:
  x = 1.275
  y = 0.75
  r = 1.25

Iteracion 2:
  x = 1.59375
  y = 0.9375
  r = 1.0625

Iteracion 3:
  x = 1.693359375
  y = 0.99609375
  r = 1.00390625

Iteracion 4:
  x = 1.6999740600585938
  y = 0.9999847412109375
  r = 1.0000152587890625

Final result: 1.6999740600585938
```

### 9.2 Implementación Ejemplo

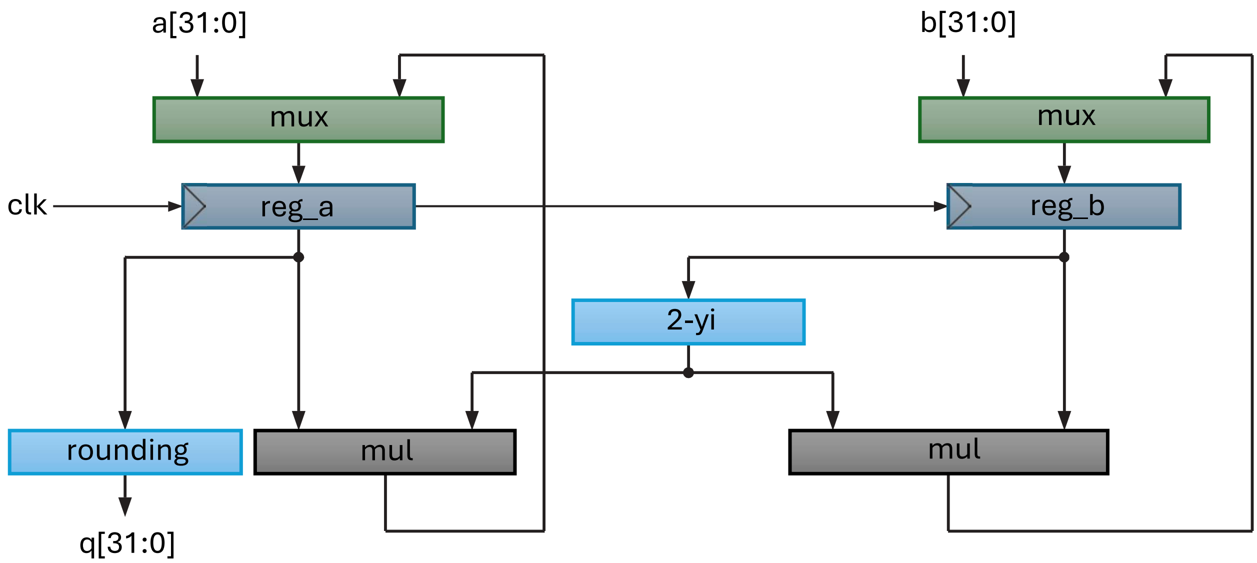


Figure 3: Goldschmidt Division

## 9.3 Código HDL

```
module div_goldschmidt (
  input  wire [31:0] a,     // dividend: .1xxx...x
  input  wire [31:0] b,     // divisor: .1xxx...x
  input  wire        start, // start
  input  wire        clk,   // clock
  input  wire        rst_n, // reset
  output wire [31:0] q,     // quotient: x.xxx...x
  output reg         busy,  // busy
  output reg         ready, // ready
  output reg  [2: 0] count, // counter
  output wire [31:0] yn    // .11111...1
);
  reg [63:0] reg_a; // x.xxxx...x
  reg [63:0] reg_b; // 0.xxxx...x

  wire [63:0] two_minus_yi = ~reg_b + 1'b1;     // 1.xxxx...x (2 - yi)
  wire [127:0] xi = reg_a * two_minus_yi;       // 0x.xxx...x
  wire [127:0] yi = reg_b * two_minus_yi;       // 0x.xxx...x

  assign two_minus_yi = ~reg_b + 1'b1;          // 1.xxxx...x (2 - yi)
  assign xi           = reg_a * two_minus_yi;   // 0x.xxx...x
  assign yi           = reg_b * two_minus_yi;   // 0x.xxx...x

  assign q  = reg_a[63:32] + |reg_a[31:29];     // rounding up
  assign yn = reg_b[62:31];

  always @ (posedge clk or negedge rst_n)
  begin
    if (!rst_n) begin
      busy  <= 0;
      ready <= 0;
      reg_a <= 0;
      reg_b <= 0;
      count <= 0;
    end else  begin
      if (start) begin
        reg_a <= {1'b0,a,31'b0}; // 0.1x...x0...0
        reg_b <= {1'b0,b,31'b0}; // 0.1x...x0...0
        busy  <= 1;
        ready <= 0;
        count <= 0;
      end else begin
        reg_a <= xi[126:63]; // x.xxx...x
        reg_b <= yi[126:63]; // 0.xxx...x
        count <= count + 3'b1; // count++
        if (count == 3'h4) begin // finish
          busy <= 0;
          ready <= 1; // q is ready
        end
      end
    end
  end
endmodule
```

Listing 4: Verilog HDL Goldschmidt Divisor

# 10 Newton-Raphson Division

## 10.1 Introducción

Para la implementación de este divisor vamos a utilizar el método de Newton-Raphson

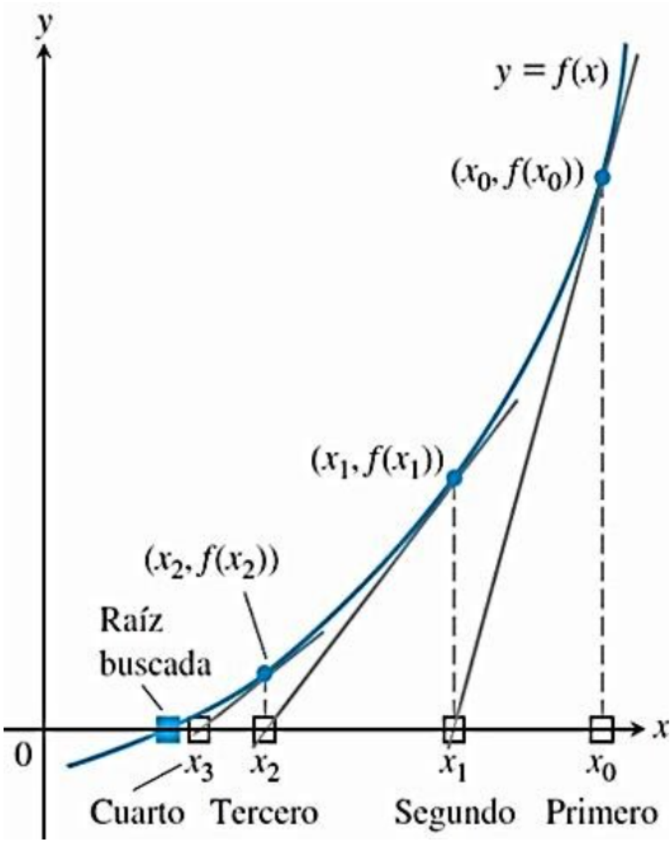


Figure 4: Método de Newton Raphson

- Este divisor también utiliza multiplicadores para obtener el cociente.
- La idea del algoritmo es que para calcular $\frac{a}{b}$, si podemos calcular $\frac{1}{b}$ sin realizar una división, podemos obtener $\frac{a}{b} = a \times (\frac{1}{b})$.
- Dada una $f(x)$, cómo podemos obtener un $x_n$ tal que $f(x_n) \approx 0$?
  Suponemos un valor $x_0$ y usando la eq. tangencial de $f(x)$ en $x_o$
  Tenemos:
  $y - f(x_0) = f'(x_0)(x - x_0)$. Haciendo $y = 0$ podemos obtener $x_1$
  Se demuestra que $x_1$ está más cerca de $x_n$ que $x_0$
- Generalmente, para $y - f(x_i) = f'(x_i)(x - x_i)$ y haciendo $y = 0$, Podemos obtener una nueva $x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$ y repetir hasta obtener la precisión deseada.
- El objetivo es encontrar $1/b$. Para hacer eso, vamos a encontrar la raíz de la función $f(x)$ definida por:

  $$f(x) = \frac{1}{x} - b$$

  Para encontrar dónde esta función es igual a cero, establecemos $f(x) = 0$, lo que nos da $\frac{1}{x} = b$ o $x = \frac{1}{b}$, que es el recíproco que estamos buscando.

  Luego, necesitamos calcular la derivada de $f(x)$ con respecto a $x$, que es $f'(x)$:

  $$f'(x) = -\frac{1}{x^2}$$

Sustituyendo la función $f(x)$ y su derivada $f'(x)$ en la fórmula de Newton-Raphson, obtenemos:

$$x_{n+1} = x_n - \frac{\left(\frac{1}{x_n} - b\right)}{-\frac{1}{x_n^2}} = x_n - (-x_n^2)\left(\frac{1}{x_n} - b\right)$$

Simplificando, eliminamos el signo negativo y los términos $x_n$ se cancelan en el numerador y el denominador:

$$x_{n+1} = x_n + x_n\left(b - \frac{1}{x_n}\right) = x_n\left(1 + b - \frac{1}{x_n}\right) = x_n(2 - x_n b)$$

Esto nos da la fórmula de actualización para cada iteración para aproximar $1/b$:

$$x_{n+1} = x_n(2 - x_n b)$$

Podemos calcular $a/b$ con los siguientes pasos:

1. Aplicar un desplazamiento de bits a $b$ para escalarlo de modo que $0.5 \leq b < 1$. Es decir, $b = 0.1xx \ldots xx$.
2. El mismo desplazamiento de bits debe aplicarse a $a$ para que el cociente no cambie.
3. Usar algunos MSBs de $b$ para obtener $x_0$ de la tabla ROM (memoria solo de lectura), o dejar $x_0 = 1.5$. (Observar que si $0.5 \leq b < 1$, entonces $1 < 1/b < 2$ y un primer aproximador podría ser $x_0 = 1.5$).
4. Repetir el cálculo de $x_{i+1} = x_i(2 - x_i b)$, hasta que tenga suficiente precisión.
5. Calcular $a \times x_n$, para obtener una aproximación del cociente $q$.

## 10.2 Convergencia Cuadrática

Supongamos que $x_i$ tiene $P_i$ bits precisos. Esto significa que el error relativo $e_r$ de $x_i$ respecto a $1/b$ (el valor al que $x_i$ converge) es menor a $2^{-P_i}$:

$$e_r = \left|\frac{x_i - 1/b}{1/b}\right| \leq 2^{-P_i} \tag{9}$$

Es decir,

$$|x_i| \leq \frac{1}{b}2^{-P_i} + \frac{1}{b} \tag{10}$$

Debido a que $x_{i+1} = x_i(2 - x_i b)$, resulta

$$\begin{aligned} |x_{i+1}| &\leq \left(\tfrac{1}{b}2^{-P_i} + \tfrac{1}{b}\right)\left[2 - \left(\tfrac{1}{b}2^{-P_i} + \tfrac{1}{b}\right)b\right] \\ &\leq \tfrac{1}{b}\left(2^{-P_i} + 1\right)\left(1 - 2^{-P_i}\right) \\ &\leq \tfrac{1}{b}\left(1 - 2^{-2P_i}\right) \\ &\leq \tfrac{1}{b} - \tfrac{1}{b}2^{-2P_i} \end{aligned} \tag{11}$$

Luego,

$$\left|\frac{x_{i+1} - 1/b}{1/b}\right| \leq 2^{-2P_i} = 2^{-P_{i+1}} \tag{12}$$

Es decir, $P_{i+1} = 2P_i$, y por lo tanto, el número de bits precisos se duplicará después de una iteración.

De esta forma, si se inicia el algoritmo con una precisión inicial de $P_0$ bits y se requieren $P_{N-1}$ bits finales, la cantidad de iteraciones necesaria será:

$$N = \left\lceil \log_2\left(\frac{P_{N-1}}{P_0}\right)\right\rceil \tag{13}$$

Para el análisis formal de la convergencia del algoritmo leer bibliografía.

### 10.3 Implementación Ejemplo

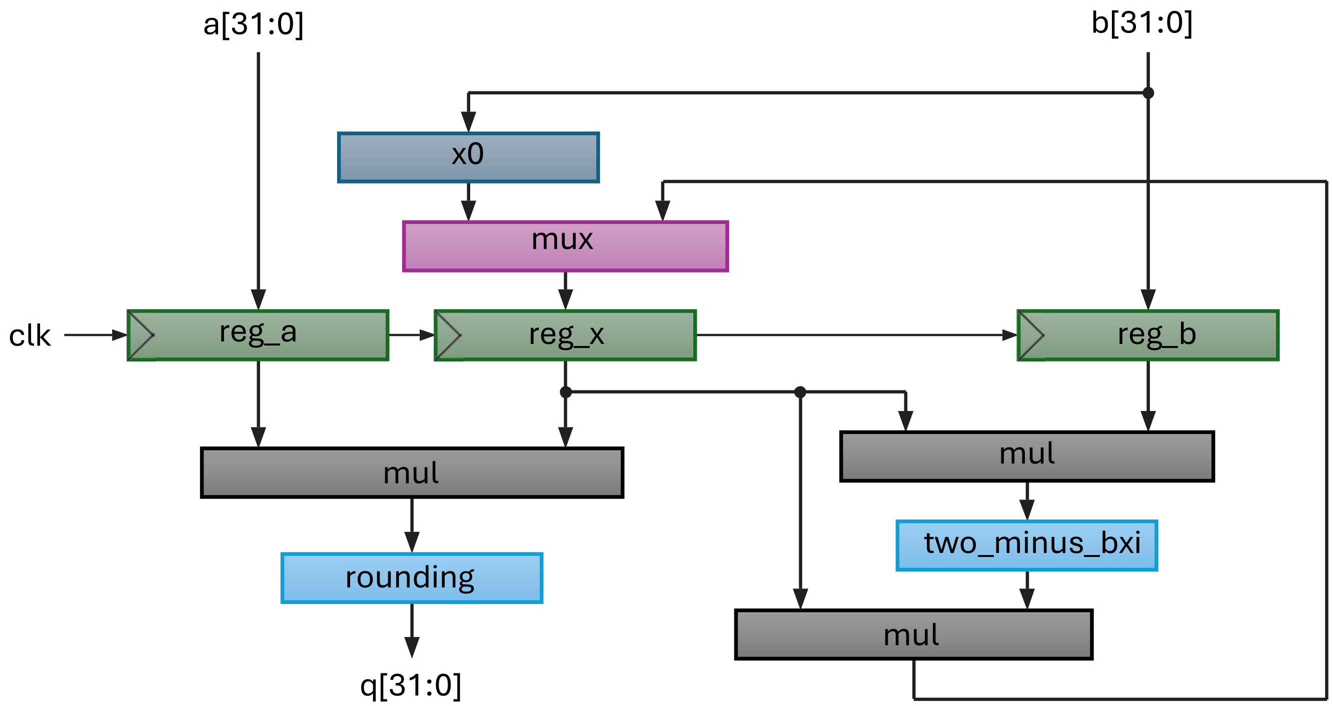


Figure 5: Newton-Raphson Divider

## 10.4 Código HDL

```
module newton (
  input  wire [31:0] a,                  // dividendo: .1xxx...x
  input  wire [31:0] b,                  // divisor:   .1xxx...x
  input  wire        start,              // start
  input  wire        clk,                // clock
  input  wire        rst_n,              // reset
  output wire [31:0] q,                  // cociente: x.xxx...x
  output reg         busy,               // busy
  output reg         ready,              // ready
  output reg  [1:0]  count               // counter
);

  // Internal
  reg [33:0] reg_x;                      // xx.xxxxx...xx
  reg [31:0] reg_a;                      // .1xxxx...xx
  reg [31:0] reg_b;                      // .1xxxx...xx

  // x_{i+1} = x_i * (2 - x_i * b)
  wire [65:0] axi;                       // xx.xxxxx...x
  wire [65:0] bxi;                       // xx.xxxxx...x
  wire [33:0] b34;                       // x.xxxxx...x
  wire [67:0] x68;                       // xxx.xxxxx...x
  wire [7:0]  x0;

  // x_{i+1} = x_i * (2 - x_i * b)
  assign axi = reg_x * reg_a;            // xx.xxxxx...x
  assign bxi = reg_x * reg_b;            // xx.xxxxx...x
  assign b34 = ~bxi[64:31] + 1'b1;       // x.xxxxx...x
  assign x68 = reg_x * b34;              // xxx.xxxxx...x
  assign x0  = rom(b[30:27]);
  assign q   = axi[64:33] + |axi[32:30]; // rounding up

  always @ (posedge clk or negedge rst_n) begin
    if (!rst_n) begin
      busy  <= 0;
      ready <= 0;
      count <= 0;
      reg_a <= a;
      reg_b <= 0;
      reg_x <= 0;
    end else begin
      if (start) begin
        reg_a <= a;                      // .1xxxx...x
        reg_b <= b;                      // .1xxxx...x
        reg_x <= {2'b1,x0,24'b0};        // 01.xxxx0...0
        busy  <= 1;
        ready <= 0;
        count <= 0;
      end else begin
        reg_x <= x68[66:33];             // xx.xxxxx...x
        count <= count + 2'b1;           // count++
        if (count == 2'h2) begin
          // 3 iteraciones
          busy  <= 0;
          ready <= 1;                    // q is ready
        end
      end
    end
  end
  function [7:0] rom; // Tabla ROM
    input [3:0] b;
    case (b)
      4'h0: rom = 8'hff;  4'h4: rom = 8'h93;  4'h8: rom = 8'h4d;  4'hc: rom = 8'h1c;
      4'h1: rom = 8'hdf;  4'h5: rom = 8'h7f;  4'h9: rom = 8'h3f;  4'hd: rom = 8'h12;
      4'h2: rom = 8'hc3;  4'h6: rom = 8'h6d;  4'ha: rom = 8'h33;  4'he: rom = 8'h08;
      4'h3: rom = 8'haa;  4'h7: rom = 8'h5c;  4'hb: rom = 8'h27;  4'hf: rom = 8'h00;
    endcase
  endfunction
endmodule
```

Listing 5: Divisor Newton-Raphson

## References

[1] Lu Mi et al., Arithmetic and Logic in Computer systems. John Wiley & Sons, 2010

[2] Yamin Li et al., Computer Principles and design in Verilog HDL. John Wiley & Sons, 2015