Universidad de Buenos Aires
Facultad de Ingeniería

Especialización en Microelectrónica

Diseño Lógico

Diseño de un Divisor por Búsqueda Binaria

Alumno: Agustín Galdeman
Fecha: 5 de mayo de 2025

Buenos Aires, Argentina

1. Divisor por Búsqueda Binaria

Este módulo implementa un divisor parametrizable en Verilog utilizando una búsqueda binaria sobre el valor absoluto del cociente. Soporta números con signo en complemento a dos, de ancho arbitrario N.

Objetivo

Dado un dividendo D y un divisor d, ambos con signo, se desea obtener:

$$Q = \left\lfloor \frac{D}{d} \right\rfloor, \qquad R = D - Q \cdot d$$

Algoritmo Utilizado

Se utiliza búsqueda binaria sobre el cociente absoluto siguiendo los siguientes pasos:

1. Ingresan $|D|$ y $|d|$.
2. Se inicializan los límites: $L = 0$, $H = |D|$.
3. Mientras $H > L + 1$:
   - $mid = \left\lfloor \frac{L+H+1}{2} \right\rfloor$
   - Si $mid \cdot |d| > |D|$ entonces $H = mid$
   - Si no, $L = mid$
4. El cociente final se obtiene con signo:

$$Q = (\text{signo}(D) \oplus \text{signo}(d))? - L : L$$

Descripción del módulo:

Registros Importantes

```
1    reg signed [N-1:0] abs_dividend;
2    reg signed [N-1:0] abs_divisor;
3    reg signed [N:0] L, H;
4    reg q_sign;
5
6    reg signed [N:0] mid_next;
7    reg signed [2*N-1:0] prod_next;
```

- `abs_dividend`, `abs_divisor`: valores absolutos de los operandos
- `q_sign`: determina el signo final del cociente
- `L`, `H`: límites inferior y superior del cociente absoluto
- `mid_next`: punto medio actual entre L y H
- `prod_next`: resultado de `mid_next` $\times$ `abs_divisor`

Máquina de Estados Finita (FSM)

El módulo está controlado por una FSM con los siguientes estados:

```
    case (state)
                //------------------------------------------------
                S_IDLE: begin
                    busy <= 1'b0;
                    if (start) begin
                        if (divisor == 0) begin
                            quotient <= 0;
                            done     <= 1'b1;
                        end else begin
                            abs_dividend <= (dividend[N-1]) ? -dividend
                               : dividend;
                            abs_divisor  <= (divisor[N-1])  ? -divisor
                               : divisor;
                            q_sign       <= dividend[N-1] ^ divisor[N
                               -1];
                            state        <= S_INIT;
                            busy         <= 1'b1;
                        end
                    end
                end

                //------------------------------------------------
                S_INIT: begin
                    L          <= 0;
                    H          <= {1'b0, abs_dividend};
                    state      <= S_SEARCH;
                end

                //------------------------------------------------
                S_SEARCH: begin
                    mid_next  = (L + H + 1) >>> 1;
                    prod_next = mid_next * abs_divisor;

                    if (prod_next > {{(2*N - N){1'b0}}, abs_dividend})
                        H <= mid_next;
                    else
                        L <= mid_next;

                    if (H <= (L + 1))
                        state <= S_FINISH;
                end

                //------------------------------------------------
                S_FINISH: begin
                    quotient <= q_sign ? -L[N-1:0] : L[N-1:0];
                    busy     <= 1'b0;
                    done     <= 1'b1;
                    state    <= S_IDLE;
                end

                //------------------------------------------------
                default: state <= S_IDLE;
            endcase
```

- **S_IDLE**: espera un pulso `start`. Si el divisor es 0, se asigna cociente = 0 y `done` = 1.
- **S_INIT**: se calculan los valores absolutos y se inicializan los límites.
- **S_SEARCH**: se realiza la búsqueda binaria, evaluando el producto $mid \cdot |d|$ en cada iteración.
- **S_FINISH**: se ajusta el signo final del cociente y se termina la operación.

Ejemplo Paso a Paso

Ejemplo: $D = -42$, $d = 8$.

- $|D| = 42$, $|d| = 8$
- Inicialización: $L = 0$, $H = 42$
- Iteraciones:
  - $mid = 21$, $168 > 42 \Rightarrow H = 21$
  - $mid = 11$, $88 > 42 \Rightarrow H = 11$
  - $mid = 6$, $48 > 42 \Rightarrow H = 6$
  - $mid = 3$, $24 \leq 42 \Rightarrow L = 3$
  - $mid = 5$, $40 \leq 42 \Rightarrow L = 5$
  - $mid = 6$, $48 > 42 \Rightarrow H = 6$
- Resultado: $L = 5$, $H = 6$, $\Rightarrow Q = -5$

2. Testbench y Validación

Simulación con Icarus Verilog

Para compilar y simular el módulo divisor junto con su testbench, se utilizó la herramienta **Icarus Verilog**.

El flujo de trabajo fue el siguiente:

1. Se guardaron los archivos del divisor y del testbench con los siguientes nombres:
   - `divider.v` — módulo divisor
   - `divider_tb.v` — testbench de prueba
2. Se utilizó el siguiente comando para compilar ambos archivos:

```
iverilog -g2001 -o sim.out divider.v divider_tb.v
```

Este comando realiza lo siguiente:

- `-g2001`: habilita compatibilidad con la sintaxis de Verilog-2001
- `-o sim.out`: genera el ejecutable de simulación con ese nombre
- Los dos archivos fuente se pasan como entrada al compilador

3. Una vez generado el binario, se ejecutó la simulación con:

   ```
   vvp sim.out
   ```

   Este comando lanza la simulación y muestra los resultados por consola, línea por línea, con mensajes `PASS` o `FAIL` según la validación de cada caso de prueba.

Requisitos

Para reproducir este entorno de simulación se necesita tener instalado:

- `iverilog`
- `vvp` (parte de Icarus Verilog)

Opcionalmente se puede utilizar `GTKWave` para visualizar señales si se agregan dumps de waveform, aunque este proyecto fue validado exclusivamente con salida por consola.

Funcionamiento del testbench

El testbench implementado tiene como objetivo verificar el correcto funcionamiento del módulo divisor a través de una serie de pruebas dirigidas:

- Se define un parámetro de ancho de palabra (`N = 16`) y se configura un reloj de 100 MHz.
- Se instancia el módulo divisor (`div_binsearch`) y se conectan sus señales.
- Se utiliza una tarea (`do_divide`) que automatiza el proceso de aplicar un par de valores (`dividend`, `divisor`), iniciar la operación con un pulso en `start`, y esperar hasta que la señal `done` se active.
- Una vez completada la división, se compara el valor entregado por el módulo en `quotient` con el resultado de la operación nativa `a / b` de Verilog.
- Si el divisor es cero, el módulo retorna por convención un cociente igual a cero, y se marca como caso manejado correctamente.
- Los resultados se muestran en consola usando mensajes `PASS` o `FAIL`, según corresponda.

Límites de visibilidad

Este testbench no accede a señales internas del módulo, como los registros de búsqueda binaria `L` y `H`, ni calcula el residuo de la división. Se limita a verificar únicamente el valor final del cociente (`quotient`).

Cobertura

Los casos de prueba seleccionados incluyen:

- División entre números positivos y negativos.
- Casos exactos y con residuo.
- Divisiones por potencias de dos.
- División por cero.
- Casos extremos como -2^{15} y $2^{15} - 1$.

Cuadro 1: Resultados obtenidos con el testbench.

Dividendo	**Divisor**	**Esperado**	**Obtenido**	**Resultado**
42	8	5	5	PASS
-42	8	-5	-5	PASS
42	-8	-5	-5	PASS
-42	-8	5	5	PASS
100	3	33	33	PASS
-100	3	-33	-33	PASS
100	-3	-33	-33	PASS
-100	-3	33	33	PASS
257	16	16	16	PASS
-257	16	-16	-16	PASS
257	-16	-16	-16	PASS
-257	-16	16	16	PASS
7	3	2	2	PASS
0	5	0	0	PASS
10	0	0	0	PASS (div/0)
32767	123	266	266	PASS
-32768	-321	102	102	PASS

Observaciones Finales

- El número de ciclos varía con el tamaño del cociente: $O(\log_2 |Q|)$
- El módulo evita el uso de la división '/' directa
- Se puede extender fácilmente para calcular el residuo de manera nativa.