Pre-Informe 3

Franco Parra
201773051-K

Sección 3.1: Investigación de conceptos

1. ¿Qué es un lenguaje de descripción de hardware? Un lenguaje de descripción de hardware o HDL (por sus siglas en inglés) es un lenguaje de programación especializado en la descripción de la estructura y comportamiento de circuitos electrónicos, y aunque más frecuente, circuitos digitales. Su intención no es simular ser un programa trivial de computadora (montado a través de algún sistema operativo quizás), sino más bien, una herramienta para el diseño de hardware permitiendo entregar los circuitos más óptimos en la aplicación. Es algo (no en todo aspecto) a lo que sería un lenguaje de enmarcado con alguna que otra utilidad más.

2. Realmente las diferencias son sutiles, pero aquí se enumeran:

   - Wire: Una variable de tipo wire o cable es aquella que no es capaz de mantener un valor fijo, es decir, sólo se puede usar como un medio para leer. Generalmente son utilizadas para conectar distintos módulos.
   - Una variable de tipo reg o registro (abreviado) no es propiamente tal un hardware que almacena valores (es decir, que al definirla no puede ser interpretada como uno), pero a nivel lógico sí lo permite. Como tienen la capacidad de retener valores son usados generalmente en bloques always.
   - Logic: Es una variable que convive como un wire y reg, aunque en realidad extiende el tipo rand. Fue implementada en Systemverilog para evitar las confusiones entre las declaraciones de tipo y pueden ser usadas en casi todo ámbito del lenguaje ya que este último es una extensión de verilog, lenguaje que por supuesto ya contenía las variables del tipo Wire y Reg.

3. "Assign" es el tipo de asignación predilecta si lo que se busca es generar lógica combinacional en el diseño puesto que se encarga de definir y redefinir tantas veces a la variable como las expresiones involucradas cambien, es decir,

   ```
   assign Y = A & B
   ```

   Y cambiará apenas A o B cambien.

   Por otro lado, la asignación "=" o *blocking assignment* puede ser utilizada en cualquier instancia del programa (hasta en declaraciones *always*); tal como su nombre lo indica, estas sentencias bloquean cualquier ejecución del programa hasta finalizar su evaluación, y luego, permiten a las sentencias posteriores funcionar.

   Finalmente, la asignación "¡=" o *non-blocking assignment* puede ser utilizada (al igual que en *blocking assignment*) en cualquier lugar, aunque preferentemente se utilizan en bloques *always.* La gracia de este método es que un conjunto de expresiones puede ser evaluada simultáneamente, y por ende, no aseguran un orden de asignación (esto claramente no sucede en el caso anterior).

   Para mayor claridad,

   ```
   always @ (posedge CLK)
   begin
   ```

```
        A0 <= 1'b1;
        A1 <= A0;
        A2 <= A1;
    end
```

Se ejecutará la asignación hacia A0 en el primer canto de subida del clock, luego se esperará hasta el otro canto de subida para asignar A1 y así.

En cambio, en estas líneas de código

```
    always @ (posedge CLK)
    begin
        A0 = 1'b1;
        A1 = A0;
        A2 = A1;
    end
```

Se asignan A0, A1 y A2, uno detrás de otro apenas el reloj haya subido.

4. En primer lugar, definimos un vector de registros de 16 bits llamado "g" el cual se asigna usando *blocking assignment*, para el cual se llena con una cadena hexadecimal de bits con valores A6B2, lo cual compáctamente viene a resumir la escritura "1010 0110 1011 0010".

5. Un efecto de delay propiamente tal no se aprecia (al menos para mi criterio sería visible si tuviésemos más asignaciones *non-blocking* en el `always_ff`), sin embargo, lo que sí afecta es el hecho de que la variable "second" cambia apenas "first" lo haga (dado el *assign*). Dada esta particularidad, en el `always_ff` apenas first es uno, second cambia inmediatamente a $\overline{\text{first}}$, es decir, cero; por eso mismo vemos

```
    arr[1] <= ~arr[0];
```

Sección 3.2: Análisis de código

A continuación se adjunta la tabla indicando el número de canto de subida como #CLK.

R	#CLK	N	X	Y	Z	M	S
000	0						
			00000000				
				01110111			
					000		
		01110111					
						001	
							0
	1				000		
						001	
							1
			00000000	01110110	001		
		01110110					
						010	
	2		00000000	01110100	010		
		01110100					
						011	
	3		00000000	01110100	011		
		01110100					
						100	
	4		00010000	01110100	100		
		10000100					
						101	
	5		0001000	01100100	101		
		01110100					
						110	
	6		00010000	01000100	110		
		01010100					
						111	
	7		00010000	01000100	111		
		01010100					
						000	
	8		00010001	11000100	000		
		11010101					
						001	
	9		00010011	11000101	001		
		11011000					
						010	
	10		00010111	11000101	010		
		11011100					
						011	

Sección 3.3: Diseño de módulo

Por simplicidad para la interpretación de curvas, se adjuntan únicamente tres variables:

- clk: El reloj.
- counter: El contador.
- meets: Un booleano que indica si counter pertenece o no a la serie fibonacci.

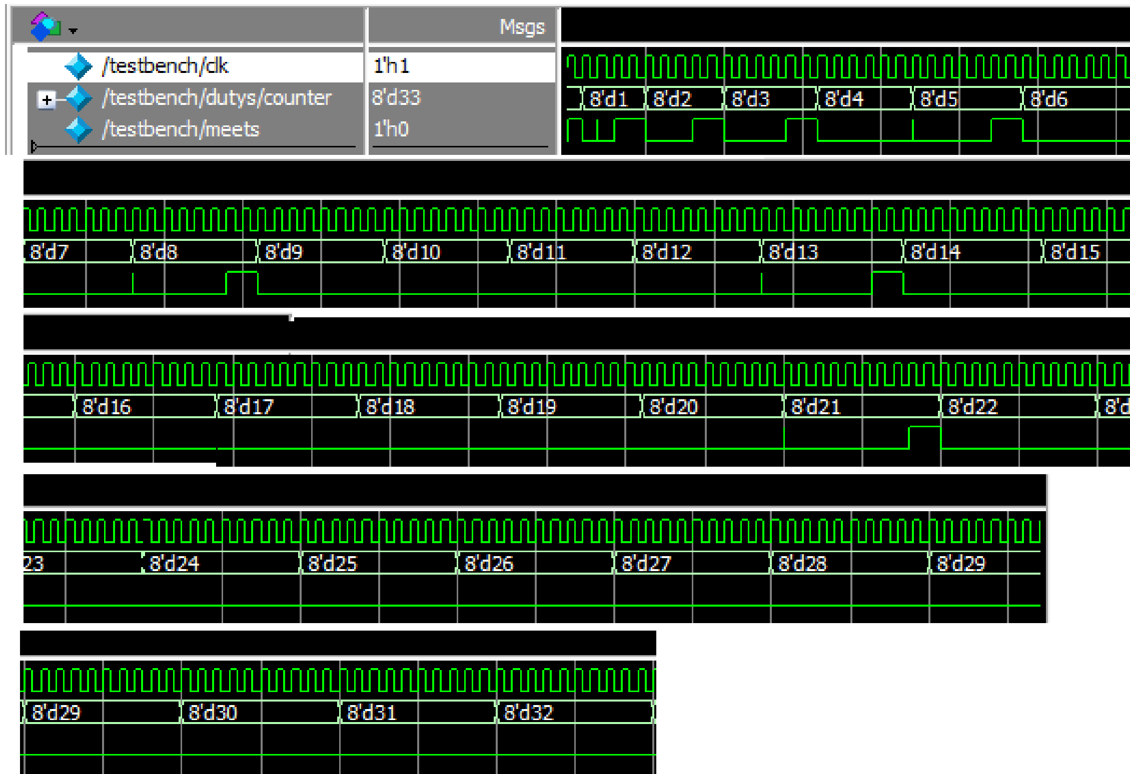

Figure 1: Curvas recortadas

¿Cómo funciona el código? Básicamente consiste en tres módulos principales, el primero de ellos llamado dut, que permite identificar si un número es fibonacci o no mediante una variable una variable de entrada numérica y dos salidas booleanas, la primera de ellas meets, que indica si realmente pertenece a la serie o no, y a su vez, overflow la cual retorna verdadero si la serie ha excedido el número a verificar.

Luego, un módulo duty que básicamente consta de un contador que cambia de valores y los evalua en dut, verificando si el número actual es fibonacci o no.

Finalmente, el código desarrollado es el siguiente:

- testbench:

```
‘timescale 1ns/1ns

module testbench;
 logic clk;
 logic overflow;
 logic meets;
 logic reset;

 duty dutys(.clk(clk), .overflow(overflow), .meets(meets));
```

```
 initial
 begin
 overflow <= 1'b0;
 meets <= 1'b0;
 reset <= 1'b0;
 end

 always
 begin
 clk <= 1; #5;
 clk <= 0; #5;
 end

endmodule
```

- dut

```
module dut(
    input logic clk,
    input logic reset,
    input logic [7:0] number,
    output logic overflow,
    output logic meets);
 logic [7:0] current = 8'b00000001, previous = 8'b0, future = 8'b0;

 always_ff @ (posedge clk, posedge reset)
 begin
 if (reset)
 begin
 current <= 8'd1;
 previous <= 8'd0;
 future <= 8'd0;
 overflow <= 1'd0;
 meets <= 1'd0;
 end

 if (number == previous || number == current)
 meets = 1'b1;

 future   = previous + current;
 previous = current;
 current  = future;

 if (number == future)
 meets = 1'b1;
 if (number < future)
 overflow = 1'b1;
 end
endmodule
```

- duty:

```
module duty(input logic clk, output logic overflow, output logic meets);
```

```
 logic [7:0] counter = 8'd0;
 logic reset = 1'd0;

 dut duts(
     .clk(clk),
     .reset(reset),
     .number(counter),
     .overflow(overflow),
     .meets(meets));

 always_ff @ (posedge clk)
 begin
 if(reset) begin
 reset <= 0;
 end
 if(overflow) begin
 counter <= counter+1;
 reset <= 1;
 end
 end
endmodule
```