Pre-Informe 3
Laboratorio de Integración Tecnológica

Raúl Álvarez Cortés
201773010-2

Investigación de Conceptos

a) Un lenguaje de descripción de hardware (HDL) es un tipo de lenguaje de programación especializado que se usa para definir la estructura, diseño y operación en circuitos electrónicos y además de esto, poder tener un análisis automático y la posibilidad de simular estos de forma digital. La diferencia de este tipo de lenguajes en comparación con los lenguajes de programacion de computadores es que estos incluyen de manera explicita la noción de tiempo, a su vez, los HDL son, como en su descripción inicial se dijo, utilizados para describir la estructura y el comportamiento de circuitos electrónicos mientras que los lenguajes de programación son utilizados para escribir un algoritmo o serie de instrucciones las cuales permitiran que el procesador pueda realizar una tarea especifica.

b) Las variables `wire, reg` y `logic` son variables utilizadas para representar valores lógicos

- `wire`: Dato utilizado para asignamiento continuo y que simula un cable, es decir, solo posee un valor cuando esta activo y designado, sin embargo, no puede registrar valores. Se utiliza para conectar diferentes elementos.
- `reg`: Es un elemento utilizado para almacenar estados o mantener su valor hasta que este se asigne nuevamente. Esta nueva asignación no depende de una nueva instrucción de asignación, si no que puede cambiar con el tiempo como un Flip-flop o un Latch por ejemplo.
- `logic` Es un elemento que puede ser utilizado tanto como una asignación continua como para una de bloqueo, es decir puede actuar tanto como un wire o un reg.

c) Existen dos formas de asignación en Verilog, las cuales son la forma continua(`assing`) y la procedural(`=,=`)

- `assign`: se utiliza solo para modelar lógica combinacional, por ende, solo puede ser declarada como un wire. Esta asignación puede ser declarada solamente fuera de cualquier proceso. Estas pueden ser ejecutadas de forma secuencial mediante temporización.

Asignación Procesural: Se les puede asignar un valor de cualquier tipo y estas se encuentran normalmente dentro de un proceso *always* o un *initial*. Estas además pueden tener un delay o temporización designado. Estas dos asignaciones son:

- `=` (*con bloqueo*): La asignación se realiza en ese instante antes de proceder con la siguiente asignación.
- `<=` (*sin bloqueo*): Si bien el termino se evalúa en el instante, no se asigna hasta finalizar dicho instante.

d) `reg [15:0]g = 16'hA6B2`

- `reg` : corresponde a una variable de asignación continua
- `[15:0]` : Su tamaño es de 16 bits
- `g` : El nombre de la variable que será asignada
- `=` : Tendrá un tipo de asignación procedural con bloqueo (tendrá ese valor de forma inmediata antes de la siguiente asignación)
- `16'hA6B2` : Según el formato N'Bvalue corresponde a un numero de base hexadecimal el cual posee un tamaño de 16 bits y su equivalente binario sería `1010 0110 1011 0010`

e) En este caso en específico notamos que si bien dentro del `always_ff` se asigna un nuevo valor (1) para `first` con bloqueo y se asigna dentro de él, no produce un cambio global en el resto de valores asociados a este hasta después de la subida del `clk`, por ende la siguiente asignación en la cual se encuentra `second` el cual esta conectado a este, no cambiara de valor hasta lo dicho anteriormente quedando `second` con el valor de 1 y luego de completar todo dentro del `always_ff` recién cambiar su valor a 0, quedando `arr=0111, first=1` y `second=0` después de la subida del `clk`.

Analisis de Código

Tomando para este caso `r=3'b0` y en el ruteo quedarán con *cursiva* todas aquellas asignaciones procedurales sin bloqueo.

Estado	x	y	z	n	m	s	y[m]	y[z]	x[m]
Prev. FlipFlop	00000000	01110111	000	01110111	001	0	1	1	0
Subida Ckl 1			*000*						
						1			
			000			1			
							1	*0*	
									0
			001						
Final Subida	00000000	011101110	001	01110110	010		1	0	0
2nd Clk=1							1	*0*	
									0
			010						
Final Subida	00000000	01110100	010	01110100	011		1	0	0
Subida Ckl 3							0	*1*	
									0
			011						
Final Subida	00000000	01110100	011	01110100	100		0	1	0
Subida Ckl 4							1	*0*	
									1
			100						
Final Subida	0010000	01110100	100	10000100	101		1	0	1
Subida Ckl 5							1	*0*	
									0
			101						
Final Subida	00010000	01100100	101	01110100	110		1	0	0
Subida Ckl 6							1	*0*	
									0
			110						
Final Subida	0010000	01000100	110	01010100	111		1	0	0
Subida Ckl 7							0	*1*	
									0
			111						
Final Subida	00010000	01000100	111	01010100	000		0	1	0
Subida Ckl 8							0	*1*	
									1
			000						
Final Subida	00010001	11000100	000	11010101	001		0	1	1
Subida Ckl 9							0	*1*	
									1
			001						
Final Subida	00010011	11000101	001	11011000	010		0	1	1
Subida Ckl 10							1	*0*	
									1
			010						
Final Subida	00010111	11000101	010	11011100	011		1	0	1

Diseño de Módulo

Antes de exponer todo el código, debemos ver como funcionará el bit que detectará cuando el contador arroje un número que pertezca a la serie de Fibonacci por lo que a través de mapas de Karnaugh quedaría así:
Sea `cont[4]`=A,`cont[3]`=B,`cont[2]`=C,`cont[1]`=D,`cont[0]`=E para simplificar la notación, entonces:

`fibo`=$\sum$(0,1,2,3,5,8,13,21)
Quedando nuestro mapa de Karnaugh así:

	$\bar{D}.\bar{E}$	$\bar{D}.E$	$D.E$	$D.\bar{E}$
$\bar{A}.\bar{B}.C$	1	1	1	1
$\bar{A}.\bar{B}.C$	0	1	0	0
$\bar{A}.B.C$	0	1	0	0
$\bar{A}.B.\bar{C}$	1	0	0	0
$A.\bar{B}.\bar{C}$	0	0	0	0
$A.\bar{B}.C$	0	1	0	0
$A.B.C$	0	0	0	0
$A.B.\bar{C}$	0	0	0	0

Figure 1: Mapa de Karnaugh de `fibo`

Donde finalmente quedará que `fibo`= $\bar{A}\bar{B}\bar{C} + \bar{A}\bar{C}\bar{D}\bar{E} + \bar{A}C\bar{D}E + \bar{B}C\bar{D}E$
Finalmente solo basta desarrollar el código de un contador de 5 bits que detecte si este contador es un número de Fibonacci:

```
1   module fibocont(clk,cont,fibo);
2           input logic clk;
3           output logic[4:0] cont;
4           output logic fibo;
5           reg [4:0]ini=0;
6           assign cont=ini;
7           assign fibo= ((~cont[4])&(~cont[3])&(~cont[2]))|
8                        ((~cont[4])&(~cont[2])&(~cont[1])&(~cont[0]))|
9                        ((~cont[4])&(cont[2])&(~cont[1])&(cont[0]))|
10                       ((~cont[3])&(cont[2])&(~cont[1])&(cont[0]));
11          always_ff@(posedge clk) begin
12                  ini <= ini+1'b1;
13          end
14  endmodule
15
```

Figure 2: Código Contador de 5bits + Detector de Fibonacci

```
1   module test;
2           logic clk;
3           fibocont test(clk);
4           initial begin
5                   forever begin
6                   #10 clk=0;
7                   #10 clk=1;
8                   end
9           end
10
11  endmodule
12
```

Figure 3: Test Bench

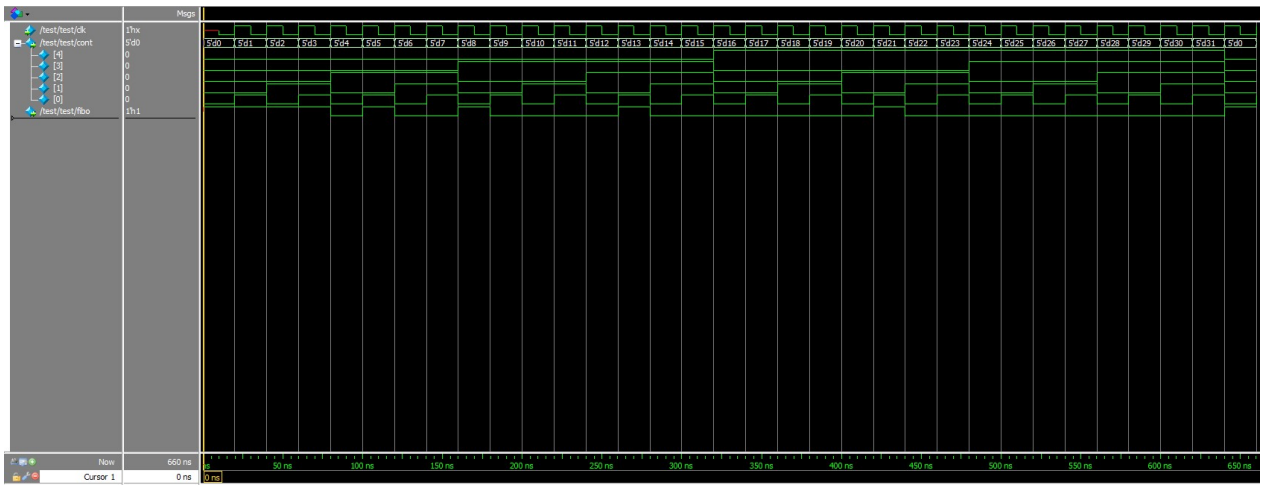

Figure 4: Resultado Simulación