Pre-Informe 3

Pablo Aravena Núñez
201773044-7

1 Pregunta 1

1.1

Un lenguaje de descripción de hardware (o HDL) es un lenguaje especializado para poder describir el comportamiento y estructura de un circuito electrónico, sirviendo para poder hacer una simulación y análisis de este último. Se diferencia de un lenguaje de programación con que el primero (HDL) contiene explícitamente una noción del tiempo. Puesto que las compuertas lógicas y módulos funcionan independiente unas de otras al mismo tiempo en un circuito real, es lógico que su simulación deba ser así también.

1.2

El tipo *"wire"* representa y es tratado como una conexión entre módulos del circuito (literalmente como un cable), por lo que este no puede realmente contener valores, solo enviar y recibir entre distintos *"drivers"* (módulos, compuertas, etc), y se usa en asignaciones continuas (las que explicaremos en la siguiente pregunta). Tanto *"reg"* como *"logic"* sirven para guardar valores como las variables de otros lenguajes de programación; *"wire"* es el más similar a estas: contiene el valor hasta la siguiente vez que se le asigne uno nuevo y lo guarde; *"logic"* es más una mezcla: puede guardar valores y además funciona como un *"wire"* solo que con 1 *driver*, y se le pueden asignar valores tanto de manera *procedural* (*"blocking"* y *"non-blocking"*) como *continua*. Ambos *"reg"* y *"logic"* pueden contener 4 estados: *"x"*, *"z"*, 0 y 1.

1.3

El operador *"¡="* sirve para asignaciones que no bloquean el flujo de ejecución de código, y es recomendado para circuitos o módulos secuenciales, mientras que el operador *"="* se recomienda para circuitos o módulos combinatorios, pues este bloquea el flujo de ejecución al momento de hacer la asignación . Las asignaciones *"non-blocking"* permiten que estas sean programadas para otro momento (el valor se actualiza en otro momento), mientras que las asignaciones *"blocking"* actualizan el valor de inmediato. Ambos de estos operadores (usados por sí solos) son asignaciones *procedurales* (usadas en variables tipo *reg* o *logic* vistas anteriormente), llamadas así porque guardan valores al estilo de los otros lenguajes de programación (en el sentido de ejecutar un grupo de sentencias en orden). Por otro lado, el modificador *"assign"* se usa para asignaciones *continuas* (usadas en variables tipo *wire* o *logic* también), las cuales funcionan recibiendo y actualizando continuamente un valor recibido o enviado desde otros *drivers* del circuito.

1.4

Para el RHS: *"reg"* es el tipo de dato explicado anteriormente, el cual significa que la variable a asignar (y específicamente una asignación del tipo *"blocking"*) será guardada hasta su futura nueva asignación, sin ser leída o actualizada por otros *drivers* como un cable. Los brackets indican que la variable tendrá 16 bits asignados desde derecha (LSB) a izquierda (MSB), es decir del índice 0 al 15, los cuales podrán ser accedidos como un array en cualquier otro lenguaje de programación (o también accedidos como *"slices"*). Luego, para el LHS: el valor dado indica el string (no confundir con el tipo de dato) binario que se intenta asignar, el cual en este caso está en hexadecimal (dado por la "h") y no especifica ningún tamaño particular de bits (Nota: en realidad este string debiera ser `'hA6B2`). Finalmente, el valor en binario es: `16*10 + 6 _ 16*11 + 2 = 10100110 _ 10110010 = 1010011010110010`.

1.5

Por lo explicado antes, las asignaciones `"<="` pueden ser programadas para tiempos distintos, por lo que se generaría un delay hasta actualizar realmente la variable `arr[first]` (que en este caso sería en el canto de subida). Dentro de `always_ff@(posedge clk)`, las sentencias se ejecutarán solo cuando el *"clock"* esté en subida, en donde `arr[first]` tendrá un valor distinto cada vez que este suba, pues se le asigna el valor opuesto al que tenía anteriormente.

2 Pregunta 2

El ruteo de 10 subidas para un `r = 3'b000` de *clock* es el siguiente:

```
    # subida 1
# ----------------------
# outside flip flop:
#    x: 00000000
#    y: 01110111
#    z: 000
#    m: 001
#    n: 01110111
#    s: 0
# inside flip flop:
#    inside s statement:
#       z: 000
#       s: 0
#    outside s block:
#       z: 000
#       y[z]: 1
#       y[m]: 1
#       x[m]: 0
# ----------------------
# subida 2
# ----------------------
# outside flip flop:
#    x: 00000000
#    y: 01110110
#    z: 001
#    m: 010
#    n: 01110110
#    s: 1
# inside flip flop:
#    outside s block:
#       z: 001
#       y[z]: 1
#       y[m]: 1
#       x[m]: 0
# ----------------------
# subida 3
# ----------------------
# outside flip flop:
#    x: 00000000
#    y: 01110100
#    z: 010
#    m: 011
```

```
#    n: 01110100
#    s: 1
# inside flip flop:
#    outside s block:
#       z: 010
#       y[z]: 1
#       y[m]: 0
#       x[m]: 0
# ----------------------
# subida 4
# ----------------------
# outside flip flop:
#    x: 00000000
#    y: 01110100
#    z: 011
#    m: 100
#    n: 01110100
#    s: 1
# inside flip flop:
#    outside s block:
#       z: 011
#       y[z]: 0
#       y[m]: 1
#       x[m]: 0
# ----------------------
# subida 5
# ----------------------
# outside flip flop:
#    x: 00010000
#    y: 01110100
#    z: 100
#    m: 101
#    n: 10000100
#    s: 1
# inside flip flop:
#    outside s block:
#       z: 100
#       y[z]: 1
#       y[m]: 1
#       x[m]: 0
# ----------------------
# subida 6
# ----------------------
# outside flip flop:
#    x: 00010000
#    y: 01100100
#    z: 101
#    m: 110
#    n: 01110100
#    s: 1
# inside flip flop:
#    outside s block:
#       z: 101
#       y[z]: 1
#       y[m]: 1
```

```
#       x[m]: 0
# ----------------------
# subida 7
# ----------------------
# outside flip flop:
#   x: 00010000
#   y: 01000100
#   z: 110
#   m: 111
#   n: 01010100
#   s: 1
# inside flip flop:
#   outside s block:
#     z: 110
#     y[z]: 1
#     y[m]: 0
#     x[m]: 0
# ----------------------
# subida 8
# ----------------------
# outside flip flop:
#   x: 00010000
#   y: 01000100
#   z: 111
#   m: 000
#   n: 01010100
#   s: 1
# inside flip flop:
#   outside s block:
#     z: 111
#     y[z]: 0
#     y[m]: 0
#     x[m]: 0
# ----------------------
# subida 9
# ----------------------
# outside flip flop:
#   x: 00010001
#   y: 11000100
#   z: 000
#   m: 001
#   n: 11010101
#   s: 1
# inside flip flop:
#   outside s block:
#     z: 000
#     y[z]: 0
#     y[m]: 0
#     x[m]: 0
# ----------------------
# subida 10
# ----------------------
# outside flip flop:
#   x: 00010011
#   y: 11000101
```

```
#    z: 001
#    m: 010
#    n: 11011000
#    s: 1
# inside flip flop:
#    outside s block:
#       z: 001
#       y[z]: 0
#       y[m]: 1
#       x[m]: 0
# ———————————————-
```

3 Pregunta 3

Para el módulo de contador e indicador de número de fibonacci usamos el siguiente código:

```
Ln#
 1  module count(input logic clk, rst,
 2                  output logic [4:0] ct,
 3                  output logic fib);
 4
 5          reg [63:0] sequen = 64'b11;
 6          reg [31:0] str1 = 32'b1;
 7          reg [31:0] str2 = 32'b0;
 8          reg [31:0] current_val = 32'b1;
 9
10
11          always_ff @(posedge clk or posedge rst) begin
12                  if (rst) ct = 5'b00000;
13                  else ct = ct + 1;
14
15                  fib = sequen[ct];
16
17          end
18
19          always_ff @(negedge clk) begin
20                  sequen[current_val] = 1;
21                  str2 = str1;
22                  str1 = current_val;
23                  current_val = str1 + str2;
24          end
25
26  endmodule
```

Figure 1: Screenshot del módulo.

Luego, para correr esto usamos lo siguiente:

```
Ln#
 1  `timescale 1ns/1ns;
 2
 3  module run();
 4          logic clk = 0;
 5          reg rst;
 6
 7          wire [4:0]ct;
 8
 9          count start(clk, rst, ct);
10
11          initial begin
12                  rst = 1;
13                  #4;
14                  rst = 0;
15          end
16
17          always begin
18                  clk <= 0; #4;
19                  clk <= 1; #4;
20          end
21  endmodule
```

Figure 2: Screenshot del código principal.

Para luego obtener las siguientes ondas en la simulación:

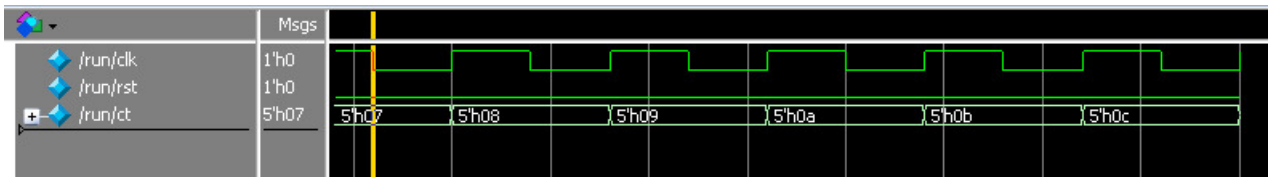

Figure 3: Screenshot de la simulación.