Hilos de Procesos de Comunicación Interna

Vargas Meza Aldo Alexandro, 213495653

Para la siguiente actividad, se llevará a cabo una serie de situaciones que involucran los diferentes nuevos comandos y funciones implicados en el tema tratado.

I. Problema 1

1. For the following code determine the order and time of execution for each statement if a `join` or `join_none` or `join_any` is used. Hint: the order and time of execution between the `fork` and `join/join_none/join_any` is the same, only the order and execution time of the statements after the `join` are different.

```
initial begin
  $display("@%0t: start fork...join example", $time);
  fork
    begin
      #20 $display("@%0t: sequential A after #20", $time);
      #20 $display("@%0t: sequential B after #20", $time);
    end
    $display("@%0t: parallel start", $time);
    #50 $display("@%0t: parallel after #50", $time);
    begin
      #30 $display("@%0t: sequential after #30", $time);
      #10 $display("@%0t: sequential after #10", $time);
    end
  join // or join_any or join_none
  $display("@%0t: after join", $time);
  #80 $display("@%0t: finish after #80", $time);
end
```

En este caso, el código va a tener 3 comportamientos diferentes dependiente tan solo en este caso de la parte join en el fork. Esto afecta al comportamiento del sistema, porque **join** tiene un comportamiento que como regla exclusiva, no avanza el programa hasta que todos sus hilos hayan sido ejecutados completamente, el **join_any** seguirá el corrimiento del programa tan pronto como un hijo termine su proceso, y el **join_none** solo ejecuta sus tareas y sigue con el programa.

```
//  ModelSim SE-64 10.1c Jul 28 2012
//
//  Copyright 1991-2012 Mentor Graphics Corporation
//  All Rights Reserved.
//
//  THIS WORK CONTAINS TRADE SECRET AND PROPRIETARY INFORMATION
//  WHICH IS THE PROPERTY OF MENTOR GRAPHICS CORPORATION OR ITS
//  LICENSORS AND IS SUBJECT TO LICENSE TERMS.
//
Loading sv_std.std
Loading work.Ejercicios2(fast)
run -all
0: start fork...join example
0: parallel start
20: sequential A after #20
30: sequential after #30
40: sequential B after #20
40: secuential after #10
50: parallel after #50
50: after join
130: finish after #80
 q -f
```

Imagen 01"fork join"

```
//  ModelSim SE-64 10.1c Jul 28 2012
//
//  Copyright 1991-2012 Mentor Graphics Corporation
//  All Rights Reserved.
//
//  THIS WORK CONTAINS TRADE SECRET AND PROPRIETARY INFORMATION
//  WHICH IS THE PROPERTY OF MENTOR GRAPHICS CORPORATION OR ITS
//  LICENSORS AND IS SUBJECT TO LICENSE TERMS.
//
Loading sv_std.std
Loading work.Ejercicios2(fast)
run -all
0: start fork...join example
0: parallel start
0: after join
20: sequential A after #20
30: sequential after #30
40: sequential B after #20
40: secuential after #10
50: parallel after #50
80: finish after #80
 q -f
```

Imagen 02"fork join_any"

```
//  ModelSim SE-64 10.1c Jul 28 2012
//
//  Copyright 1991-2012 Mentor Graphics Corporation
//  All Rights Reserved.
//
//  THIS WORK CONTAINS TRADE SECRET AND PROPRIETARY INFORMATION
//  WHICH IS THE PROPERTY OF MENTOR GRAPHICS CORPORATION OR ITS
//  LICENSORS AND IS SUBJECT TO LICENSE TERMS.
//
Loading sv_std.std
Loading work.Ejercicios2(fast)
run -all
0: start fork...join example
0: after join
0: parallel start
20: sequential A after #20
30: sequential after #30
40: sequential B after #20
40: secuential after #10
50: parallel after #50
80: finish after #80
 q -f
```

Imagen 03"fork join_none"

La razón por la que el comportamiento cambia, es porque el proceso está diseñado de tal manera que los tiempos se modifiquen con el simple comando join.

II. PROBLEMA 2

2. For the following code what would the output be with and without a `wait fork` inserted in the indicated location?

```
initial begin

  fork
    transmit(1);
    transmit(2);
  join_none

  fork: receive_fork
    receive(1);
    receive(2);
  join_none

  // What is the output with/without a wait fork here?

  #15ns disable receive_fork;
  $display("%0t: Done", $time);
end

task transmit(int index);
  #10ns;
  $display("%0t: Transmit is done for index = %0d",
            $time, index);
endtask

task receive(int index);
  #(index * 10ns);
  $display("%0t: Receive is done for index = %0d",
           $time, index);
endtask
```

En este problema, nos preguntan cual será la salida del sistema con la implementación del **wait fork** que básicamente de asegurarse de que todos los procesos "hijo" hayan terminado su ejecución.

Simulando el código sin el comando requerido y con el comando ya codificado nos da los siguientes resultados.

```
//  ModelSim SE-64 10.1c Jul 28 2012
//
//  Copyright 1991-2012 Mentor Graphics Corporation
//  All Rights Reserved.
//
//  THIS WORK CONTAINS TRADE SECRET AND PROPRIETARY INFORMATION
//  WHICH IS THE PROPERTY OF MENTOR GRAPHICS CORPORATION OR ITS
//  LICENSORS AND IS SUBJECT TO LICENSE TERMS.
//
Loading sv_std.std
Loading work.ejercicio2(fast)
run -all
10: Receive is done for index = 2
10: Transmit is done for index = 2
10: Transmit is done for index = 2
15: Done
 q -f
```

Imagen 04"Sin wait fork"

```
//  ModelSim SE-64 10.1c Jul 28 2012
//
//  Copyright 1991-2012 Mentor Graphics Corporation
//  All Rights Reserved.
//
//  THIS WORK CONTAINS TRADE SECRET AND PROPRIETARY INFORMATION
//  WHICH IS THE PROPERTY OF MENTOR GRAPHICS CORPORATION OR ITS
//  LICENSORS AND IS SUBJECT TO LICENSE TERMS.
//
Loading sv_std.std
Loading work.ejercicio2(fast)
run -all
10: Receive is done for index = 2
10: Transmit is done for index = 2
10: Transmit is done for index = 2
20: Receive is done for index = 2
35: Done
 q -f
```

Imagen 05"Con wait fork"

Es fácil ver en el tiempo de ejecución, como el código que utiliza el comando fork wait espera el término de la ejecución, para seguir con el programa, lo cual lo atrasa casi 20ns del primer programa.

III. PROBLEMA 3

3. What would be displayed with the following code? Assume that the events and task `trigger` is declared inside a program declared as automatic.

```
event e1, e2;
task trigger(event local_event, input time wait_time);
  #wait_time;
  ->local_event;
endtask

initial begin
  fork
    trigger(e1, 10ns);
    begin
      wait(e1.triggered());
      $display("%0t: e1 triggered", $time);
    end
  join
end

initial begin
  fork
    trigger(e2, 20ns);
    begin
      wait(e2.triggered());
      $display("%0t: e2 triggered", $time);
    end
  join
end
```

El problema nos pide que tomemos ambos eventos, asi como la tarea trigger, declarados dentro de un programa automatico.

```
//  ModelSim SE-64 10.1c Jul 28 2012
//
//  Copyright 1991-2012 Mentor Graphics Corporation
//  All Rights Reserved.
//
//  THIS WORK CONTAINS TRADE SECRET AND PROPRIETARY INFORMATION
//  WHICH IS THE PROPERTY OF MENTOR GRAPHICS CORPORATION OR ITS
//  LICENSORS AND IS SUBJECT TO LICENSE TERMS.
//
Loading sv_std.std
Loading work.ejercicio3(fast)
run -all
10: E2 triggered
 q -f
```

Imagen 06"Sin automatic"

El programa a pesar de tener dos ciclos initial, solo ejecuta el segundo. Al momento de integrarle una función automática:

```
//  ModelSim SE-64 10.1c Jul 28 2012
//
//  Copyright 1991-2012 Mentor Graphics Corporation
//  All Rights Reserved.
//
//  THIS WORK CONTAINS TRADE SECRET AND PROPRIETARY INFORMATION
//  WHICH IS THE PROPERTY OF MENTOR GRAPHICS CORPORATION OR ITS
//  LICENSORS AND IS SUBJECT TO LICENSE TERMS.
//
Loading sv_std.std
Loading work.ejercicio3(fast)
run -all
10: E1 triggered
20: E2 triggered
 q -f
```

Imagen 07"Con automatic"

Obtenemos ambas ejecuciones en tiempo y forma correcto.

IV. Problema 4

4. Create a task called `wait10` that for 10 tries will wait for 10ns and then check for 1 semaphore key to be available. When the key is available, quit the loop and print out the time.

El siguiente problema nos pide diseñar una tarea que durante 10 intentos, espere 10ns y luego intente obtener una llave de una función semaphore.
La función semaphore se utiliza para bloquear o desbloquear un recurso al que mas de un proces puede tener acceso, como un tipo bandera..

```
task wait10();
    int f;
    for (int i = 0; i < 10; i++) begin
        #10ns
        if(i==9) begin
            f=sem.try_get();
            if(f==1) begin
                $display("%0t: Key takken",$time);
            end
        end
    end
endtask : wait10
endmodule : ejercicio4
```

Imagen 08"Diseño task wait10"

El diseño utiliza una bandera llamada f, la cual después de ciclar por 10 veces, intenta obtener una "llave", y si lo consigue imprime un comentario junto con el tiempo de ejecución del sistema.

V. Problema 5

5. What would be displayed with the following code that calls the task from Exercise 4?

```
initial begin
  fork
    begin
      sem = new(1);
      sem.get(1);
      #45ns;
      sem.put(2);
    end
    wait10();
  join
end
```

Este problema utiliza la tarea creada en el problema anterior, la cual es mandada a llamar después de un proceso fork, que le da "**llaves**" para poder ser obtenidas por la función get.
El resultado de la simulación del código es la siguiente:

```
//  ModelSim SE-64 10.1c Jul 28 2012
//
//  Copyright 1991-2012 Mentor Graphics Corporation
//  All Rights Reserved.
//
//  THIS WORK CONTAINS TRADE SECRET AND PROPRIETARY INFORMATION
//  WHICH IS THE PROPERTY OF MENTOR GRAPHICS CORPORATION OR ITS
//  LICENSORS AND IS SUBJECT TO LICENSE TERMS.
//
Loading sv_std.std
Loading work.ejercicio4(fast)
run -all
100: Key takken
 q -f
```

Imagen 09"Tiempo ejecución tomar "llave""

Como era de esperarse, el tiempo que le tomó al diseño llegar a obtener una "llave" fue de la multiplicación de los 10 intentos por los 10ns que retardaba cada uno.

VI. Problema 6

6. What would be displayed with the following code?

```
program automatic test;
  mailbox #(int) mbx;
  int value;
  initial begin
    mbx = new(1);
    $display("mbx.num()=%0d", mbx.num());
    $display("mbx.try_get= %0d", mbx.try_get(value));
    mbx.put(2);
    $display("mbx.try_put= %0d", mbx.try_put(value));
    $display("mbx.num()=%0d", mbx.num());
    mbx.peek(value);
    $display("value=%0d", value);
  end
endprogram
```

En este código se puede apreciar el comportamiento del **Mailbox** que es una mecanismo utilizado entre procesos para intercambiar mensajes. Los datos pueden ser enviados directamente entre procesos y pueden ser los 4 tipos de datos en systemverilog.
En la impresión obtenemos:

```
//  ModelSim SE-64 10.1c Jul 28 2012
//
//  Copyright 1991-2012 Mentor Graphics Corporation
//  All Rights Reserved.
//
//  THIS WORK CONTAINS TRADE SECRET AND PROPRIETARY INFORMATION
//  WHICH IS THE PROPERTY OF MENTOR GRAPHICS CORPORATION OR ITS
//  LICENSORS AND IS SUBJECT TO LICENSE TERMS.
//
Loading sv_std.std
Loading work.ejercicio6(fast)
run -all
mbx.num()=0
mbx.try_get= 0
mbx.try_put= 0
mbx.num()= 1
Value= 2
```

Imagen 10"Mailbox"

En la cual es apreciable la creación de uno, y sus respectivos valores y funciones.

VII. Bibliografía

- http://www.asic-world.com/systemverilog/process3.html
- http://www.asic-world.com/systemverilog/sema_mail_events2.html
- http://www.asic-world.com/systemverilog/index.html
- SystemVerilog.for.Verification.A.Guide.to.Learning.the.Testbench.Language.Features.pdf