Capítulo 2

Práctica 2. Diseño de una máquina de estados finitos

En este práctica vamos a estudiar técnicas para el diseño de sistemas secuenciales, más en particular, el diseño de una máquina de estados finitos (FSM, por su acrónimo en inglés).

Estas técnicas abordarán el diseño, la codificación VHDL y la implementación con la herramienta de síntesis.

2.1. Especificaciones

Diseñar el sistema de control de una muñeca interactiva mediante una máquina de estados finitos tipo Moore.

Spec 1. El sistema funcionará con una señal de reloj, `clk`, a 25 MHz. El flanco activo de la señal de reloj es el flanco de subida.

Spec 2. El sistema tiene una señal de reset asíncrono activa a nivel bajo, `rst_n`.

Spec 3. El sistema tiene una señal de entrada de 1 bit, `r`, que indica si hay ruido en el entorno de la muñeca. $r = 1$ indica que hay ruido y $r = 0$ indica que no hay ruido.

Spec 4. El sistema tiene una señal de entrada de 1 bit, `c`, que indica si a la muñeca se le ha dado el chupete. $c = 1$ indica que tiene el chupete y $c = 0$ indica que no lo tiene.

Spec 5. El sistema tiene una señal de salida de 1 bit, `g`, que indica si la muñeca está en silencio. $g = 0$ indica que la muñeca está en silencio y $g = 1$ que está en modo no-silencio. La muñeca está en modo no-silencio si está llorando o hablando. Para indicar en cuál de esas dos situaciones se encuentra se define la siguiente señal de salida.

Spec 6. La señal de salida de 1 bit `l` indica si la muñeca está hablando o llorando. Esta señal sólo es válida cuando la señal `g` está activa. Es decir, cuando $g = 1$, $l = 1$ indica que la muñeca está llorando y cuando $g = 1$, $l = 0$ que está hablando.

Spec 7. La entidad de la muñeca viene definida por el siguiente código VHDL:

```
entity muneca is
  port (clk   : in  std_logic;
        rst_n : in  std_logic;
        r     : in  std_logic;
        c     : in  std_logic;
        g     : out std_logic;
        l     : out std_logic);
end muneca;
```

Spec 8. Tras el reset del sistema, la muñeca estará tranquila y no habla ni llora.

Spec 9. Cuando la muñeca está tranquila ni habla ni llora.

Spec 10. Cuando la muñeca está dormida ni habla ni llora.

Spec 11. Cuando la muñeca está asustada llora.

Spec 12. Cuando la muñeca está hablando sólo habla.

Spec 13. Si está tranquila y oye un ruido sin tener el chupete pasará a hablar.

Spec 14. Si está tranquila y se le pone el chupete sin que haya ruido pasará a estar dormida.

Spec 15. Si está hablando y se le pone el chupete pasará a estar dormida.

Spec 16. Si está dormida y hay ruido pasará a estar asustada.

Spec 17. Si está asustada y se le pone el chupete sin que haya ruido pasará a estar dormida.

Spec 18. Si está asustada y no hay ni ruido ni tiene el chupete pasará a estar tranquila.

Spec 19. Para cualquier otra combinación de las entradas la muñeca permanecerá en el estado en el que se encuentre.

2.2. Diseño

Las Especificaciones anteriores detallan el comportamiento de una FSM. El número de estados depende el tipo de máquina –Mealy o Moore– que se escoja para realizar el diseño. En esta práctica escogeremos una implementación tipo Moore.

Si la describimos como máquina de Moore, la muñeca posee 4 estados: tranquila, habla, dormida y asustada. Las transiciones entre ellos están gobernadas por las señales de entrada `g` y `l`. Por otra parte al ser una máquina de Moore la salida solo depende del estado actual.

La implementación en VHDL de este sistema puede hacerse de varias maneras. Paso a detallar cada una de ellas.

2.3. Descripción VHDL

2.3.1. Definición de los estados y señales

La implementación de cualquier FSM siempre requiere la definición de al menos dos señales: el estado actual (`current_state`) que almacena el estado en el que se encuentra la FSM en el ciclo de reloj actual y el estado siguiente (`next_state`) que contiene el estado al que va a pasar la FSM en el siguiente ciclo de reloj.

La muñeca tiene cuatro estados: tranquila, habla, dormida y asustada. La definición de los estados puede hacerse de dos formas. Una primera es trabajar con una descripción de alto nivel de los estados y dejar que sea la herramienta de síntesis quien encuentre la codificación óptima de los estados. Otra forma es indicar de forma explícita que codificación se desea para cada uno de los estados de la FSM. Cada una de las opciones tiene sus ventajas e inconvenientes:

- Si permitimos que la herramienta realice la codificación ésta buscará la codificación que bien minimice los requisitos de área o reduzca la longitud del camino crítico. Pero a cambio, no podremos controlar la codificación que realice de forma que cuando realicemos una comparación entre la simulación del código y la simulación de la implementación aparecerán discrepancias puesto que los valores de las señales de estado serán distintos.

  En este caso, la manera más elegante de codificar en VHDL esta aproximación es definir un tipo de datos nuevo, p.e. el tipo `t_state`, y definir las señales `current_state` y `next_state` de tipo `t_state`.

```
type t_state is (tranquila_st, habla_st, dormida_st, asustada_st);
signal current_state, next_state : t_state;
...
next_state <= asustada_st;
```

La enumeración de los valores que definen el tipo `t_state` llevan todos el sufijo `_st` para indicar, de acuerdo con la guía de estilo de la asignatura, que es el nombre de un estado.

- Si realizamos nosotros la codificación entonces ocurrirá el caso contrario. La implementación tendrá exactamente el mismo comportamiento que el código RTL desarrollado pero a cambio no será una implementación óptima salvo que nosotros hayamos optimizado la codificación de estados.

  En este caso, la manera más elegante es definir constantes para cada uno de los estados y asignárselas a las señales de estado según sea necesario en cada caso. Así:

```
constant c_tranquila_st : std_logic_vector(1 downto 0) := "00";
constant c_dormida_st   : std_logic_vector(1 downto 0) := "11";
...
signal current_state, next_state : std_logic_vector(1 downto 0);
...
next_state <= c_tranquila_st;
...
```

Las constantes llevan el prefijo `c_` para indicar que es una constante y el sufijo `_st` para indicar, de acuerdo con la guía de estilo de la asignatura, que es el nombre de un estado.

2.3.2. Definición de señales *don't care*

Los valores de la salida `l` no están definidos para todos los estados de la FSM: solo está definida en aquellos estados en los que la salida `g` toma el valor 1. Podemos tratar esta situación con dos aproximaciones distintas:

1. Asignar a la salida l el valor *don't care* en aquellos casos en los que no esté especificada. Si la señal l ha sido definida de tipo **std_logic**, la asignación es:

   ```
   l <= '-';
   ```

   En este caso, permitimos que la herramienta de síntesis asigne el valor conveniente para cumplir con las restricciones de optimización que se han impuesto. El inconveniente es que si comparamos la simulación pre-layout con la simulación post-layout aparecerán discrepancias debido a que en una y otra la señal l toma valores distintos.

2. Asignar a la salida l el valor binario que consideremos: bien 0 bien 1. En este caso, no dejamos libertad a la herramienta de síntesis para encontrar el valor óptimo de l pero, a cambio, tendremos simulaciones pre-layout y post-layout completamente idénticas, con lo que se simplifica la tarea de verificar la corrección funcional del diseño en la simulación post-layout. Bastará con comparar que las formas de onda de ambas simulaciones son idénticas.

En ningún caso debemos dejar sin asignar esta señal pues supondrá que la herramienta de síntesis infiera la presencia de elementos de memoria (latches) cuando sólo queremos describir lógica secuencial.

2.3.3. Organización de la arquitectura

La descripción de la arquitectura asociada a la entidad muneca deberá contener tres procesos:

1. Proceso `p_next_state`: encargado de describir la lógica combinacional que calcula el siguiente estado a partir de las entradas y el estado actual. Luego en la lista de sensibilidad del proceso debe estar definida tanto la señal del estado actual como todas las entradas al módulo. Al no ser lógica secuencial *no hay que incluir* ni la señal de reloj, `clk`, ni la señal de reset, `rst_n`.

   La forma más sencilla de describir la funcionalidad de la lógica es mediante la construcción **case**. Para cada uno de los estados actuales se detalla el siguiente estado en función de los valores que tienen las señales de entrada. Así:

```
p_next_state : process(current_state, r, c)
begin
  case current_state is
    <estado1> =>
       if <condiciones sobre r y c> then
            ...
```

2. Proceso `p_outputs`: encargado de describir la lógica combinacional que calcula las salidas de la FSM a partir del estado actual[1]

   De nuevo, la forma más sencilla de describir la funcionalidad de la lógica es mediante la construcción **case**. Para cada valor del estado actual se detalla los valores de las señales de salida.

```
p_outputs : process(current_state)
begin
  case current_state is
    <estado1> =>
       g <= <valores>;
       l <= <valores>;
    <estado2> =>
       ...
```

   Se pueden unir la descripción de los procesos `p_outputs` y `p_next_state` en un único proceso. Lo que *nunca* debe hacerse es unir la descripción de un proceso combinacional con uno secuencial. En otras palabras, no debemos unir ninguno de estos procesos con el proceso `p_reg` que se describe a continuación.

3. Proceso `p_reg`: encargado de describir la funcionalidad del registro de estado que almacena el estado actual de la máquina. Es un proceso secuencial y por lo tanto en su lista de sensibilidad las únicas señales que deben estar presentes son `clk` y `rst_n`. Debe modelar un registro PIPO disparado por flanco de subida y reset asíncrono activo a nivel bajo.

```
p_reg : process(clk, rst_n)
begin
  if rst_n = '0' then
   ...
  elsif rising_edge(clk) then
   ...
```

2.4. Síntesis

2.4.1. Codificación de los estados

En el caso de que se haya decidido dejar que la herramienta realice la codificación de los estados todavía es posible guiar cómo realiza dicha codificación. Podemos seleccionar la técnica de codificación que deseamos escogiendo una de las siguientes opciones de la propiedad FSM Encoding Algorithm del proceso de síntesis.

- Auto: selecciona la técnica de optimización de forma automática durante el proceso de síntesis.
- One-Hot: asegura que cada FF del registro de estado está dedicado a un único estado. Este tipo de codificación es especialmente útil cuando se dispone de un gran número de FF libres.
- Compacta: minimiza el número de variables de estado y por consiguiente el número de FF. Esta técnica es especialmente útil cuando se está intentando minimizar el área.
- Secuencial: identifica los caminos críticos y aplica códigos módulo 2 de manera sucesiva a los estados en ese camino. De esta manera las ecuaciones del siguiente estado son minimizadas.
- Gray: garantiza que sólo una variable de estado cambia entre dos estados consecutivos. Esta técnica minimiza los **hazards** y **glitches** del circuito. Se puede obtener muy buenos resultados cuando se implementó utilizando registros T o tipo JK.
- User: la herramienta de síntesis utiliza la codificación definida por el diseñador en el archivo fuente.
- Speed: codificación orientada a optimizar la velocidad del circuito. El número de bits en el registro de estado depende cada máquina pero en general es mayor que el número de estados de la FSM.
- None: la herramienta deshabilita la extracción automática de estados.

[1] Al ser una máquina de Moore la salida sólo depende del estado actual. Si fuese una máquina de Mealy las salidas dependerían tanto del estado actual como de las entradas.

2.4.2. Tipos de biestables

El tipo de biestables que es generado durante la síntesis depende de cómo hayan sido descritos en el VHDL. En esta práctica la especificación del diseño requiere que la máquina tenga un reset asíncrono activo a nivel bajo. Por esta razón el registro de estado debe describirse usando la plantilla VHDL que se presentó en la Sección 2.3.

Xilinx dispone de la siguiente lista de FF para escoger cual va a utilizar en cada caso.

Tipo	Descripción
FD*	D flip-flop
FDC	D flip-flop with async. clear
FD*CE	D flip-flop with clock enable, async. clear
FDCP*	D flip-flop with async. preset, async. clear
FDCPE	D flip-flop with clock enable, async. preset and clear
FDP	D flip-flop with async. preset
FDPE	D flip-flop with clock enable, async. preset
FDR	D flip-flop with sync. reset
FD*RE	D flip-flop with clock enable, sync. reset
FDRS	D flip-flop with sync. reset, sync. set
FDRSE	D flip-flop with clock enable, sync. reset and set
FDS	D flip-flop with sync. set
FDSE	D flip-flop with clock enable, sync. set
FDSR	D flip-flop with sync. set and reset
FDSRE	D flip-flop with clock enable, sync. set and reset
FJKC	J-K flip-flop with async. clear
FJKCE	J-K flip-flop with clock enable, async. clear
FJKCP	J-K flip-flop with async. clear and preset
FJKCPE	J-K flip-flop with clock enable, async. clear and preset
FJKP	J-K flip-flop with async. preset
FJKPE	J-K flip-flop with clock enable, async. preset
FJKRSE	J-K flip-flop with clock enable, sync. reset and set
FJKSRE	J-K flip-flop with clock enable, sync. set and reset
FTC	Toggle flip-flop with async. clear
FTCE	Toggle flip-flop with clock enable, async. clear
FTCP*	Toggle flip-flop with async. clear and preset
FTCPE	Toggle flip-flop with clock enable, async. clear and preset
FTCPLE	Loadable toggle flip-flop w/ clock enable, async. clear and preset
FTP	Toggle flip-flop with async. preset
FTPE	Toggle flip-flop with clock enable, async. preset
FTPLE	Loadable toggle flip-flop with clock enable, async. preset
FTRSE	Toggle flip-flop with clock enable, sync. reset and set
FTRSLE	Loadable toggle flip-flop with clock enable, sync. reset and set
FTSRE	Toggle flip-flop with clock enable, sync. set and reset
FTSRLE	Loadable toggle flip-flop with clock enable, sync. set and reset

Tabla 2.1: Tipos de flip-flops en Xilinx.

2.4.3. Camino crítico

NA

2.4.4. Recursos hardware

NA

2.5. Cuestiones y resultados experimentales

La documentación a presentar en la memoria de la práctica es:

1. Diseño de la FSM: diagrama de transición de estados y tabla de salidas.
2. Codificación de estados. Si decidís que la codificación sea realizada de forma automática por la herramienta entonces debéis buscar esta información en el report de síntesis. Si decidís imponer vosotros la codificación entonces basta con que indiquéis los valores que habéis escogido y por qué.
3. Explicar el número y tipo de flip-flops obtenidos durante la síntesis. Indicar para cada flip-flop obtenido con qué elemento de vuestro diseño está relacionado.
4. Determinar frecuencia de reloj máxima.
5. Determinar el número y tipo de slices empleados en la implementación de este diseño.

Para que el código de la práctica sea considerado correcto se deben cumplir los siguientes criterios:

1. La simulación del sistema debe ser correcta.
2. El código debe reflejar los detalles del diseño presentado en la memoria.
3. El código debe seguir todas las reglas incluidas en el documento “Reglas de estilo”.