Diseño de Circuitos Síncronos

Jhon Jairo Padilla Aguilar

Fuente: Peter Chambers. “The ten commandments of excellent design”.

Justificación

- Los ingenieros de diseño se ven en la necesidad de crear circuitos sincronizados por reloj para aplicaciones con frecuencias desde pocos Hz hasta Ghz.
- Los sistemas síncronos son propensos a ciertos tipos de fallas.
- Ciertas fallas no son fácilmente descubiertas durante el proceso de diseño.
- Tales fallas pueden causar problemas de inestabilidad y baja confiabilidad.

Cómo evitar tales fallas?

- Siguiendo ciertas recomendaciones que se describen en este documento.
- Se estudiarán diferentes fuentes de los problemas más comunes y sus soluciones.
- Se estudiará cómo aplicar estas ideas a sus diseños.

Por qué usar un circuito síncrono?

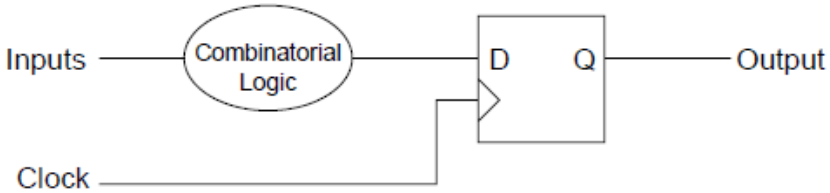
- El diseño síncrono elimina problemas asociados con variaciones de velocidad a través de diferentes caminos lógicos.
- Con el muestreo de señales en intervalos de tiempo bien definidos, tanto los caminos rápidos como los lentos pueden ser manejados en una forma simple.
- Los circuitos síncronos trabajan bien bajo variaciones de procesos, temperatura y voltaje. Esta estabilidad es clave para altos volúmenes de fabricación.

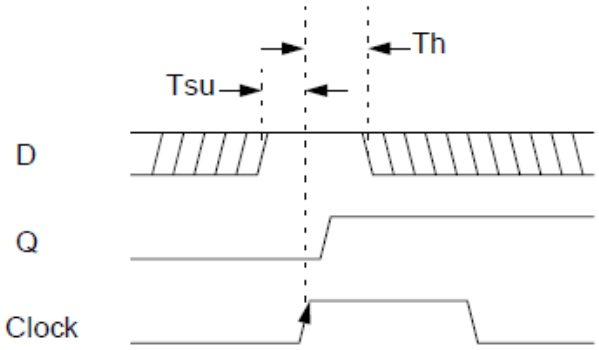
Por qué usar un circuito síncrono?

- Muchos diseños deben ser portables: fáciles de migrar a una nueva y mejorada tecnología.
- El comportamiento determinístico de los diseños síncronos hace que sean mucho más fáciles de trasladar a una nueva tecnología.
- La interconexión de dos bloques de lógica se simplifica definiendo un comportamiento síncrono estandarizado.
- La interconexión asíncrona demanda protocolos elaborados para asegurar la integridad de la información. Los diseños síncronos con características de tiempo conocidas pueden garantizar la recepción correcta de los datos.

Fuentes de Error en un Circuito típico

- Parámetros importantes:
  - Tsu: Setup Time (Tiempo de establecimiento):
    - Tiempo en que la entrada D debe ser estable antes de que se active el reloj
  - Th: Hold Time (Tiempo de retención):
    - Es el tiempo que la entrada D debe permanecer estable después de que se inactive el reloj
- Si estos tiempos son violados, no se puede predecir el estado final del Flip-Flop. (El sistema entero se sale de control!!!)





Fuentes de Error

- Distribución de relojes:
  - En un diseño FPGA de tamaño considerable pueden haber muchas señales de reloj.
  - La distribución de señales de reloj a través de un diseño ha recibido considerable atención con el incremento de la velocidad de la lógica (Hoy: velocidades de relojes de 300Mhz).

Fuentes de Error

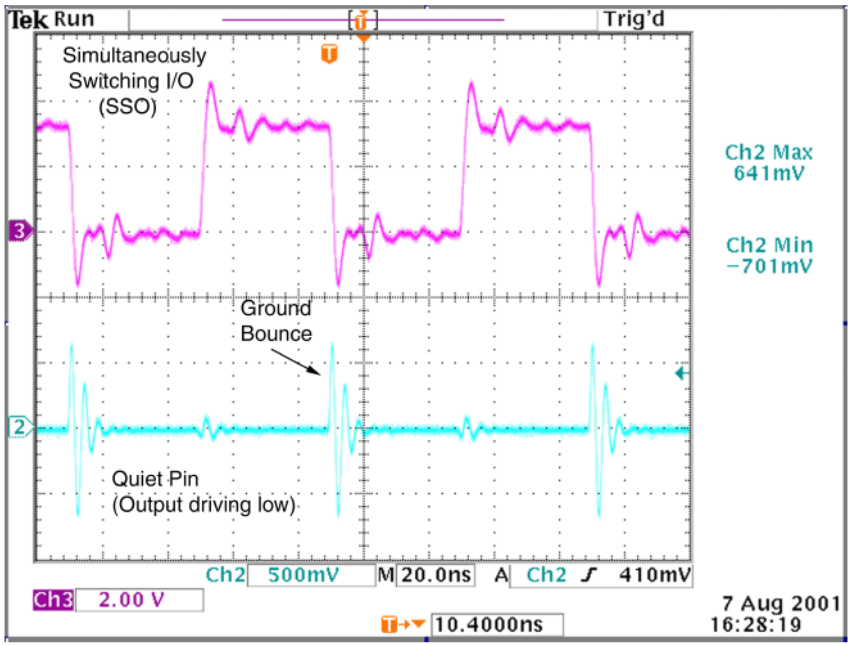
- Distribución de relojes:
  - Se debe minimizar la diferencia de retardo por diferentes caminos de reloj (Skew):
    - Esto se hace para no violar los tiempos de Setup y Hold de ningún Flip-Flop.
    - Algunas soluciones a este problema son: diseño con iguales longitudes, Buffers basados en PLL con cero retardo y lógica adicional para agrandar el retardo en ciertos caminos.

Fuentes de Error

- Fidelidad del Reloj:
  - La forma de onda del reloj debe ser lo más limpia y determinística posible.
  - Algunas técnicas para garantizar un comportamiento del reloj consistente incluyen mejorar la terminación de la línea, minimización del rebote de tierras, uso de generadores de reloj idénticos.

Rebote de Tierra

- El rebote de tierra ocurre cuando la mayoría, sino todos, de los bits de un bus de datos cambian simultáneamente de 1 a 0.
- Durante el rebote de tierra, en la señal de tierra del dispositivo aparece un transitorio (rebote) con respecto a la tierra de la tarjeta impresa.



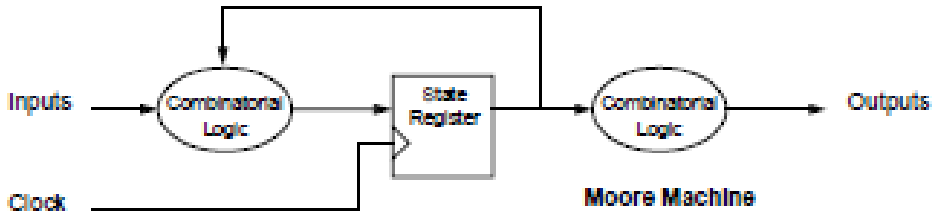
Diseño con FSMs

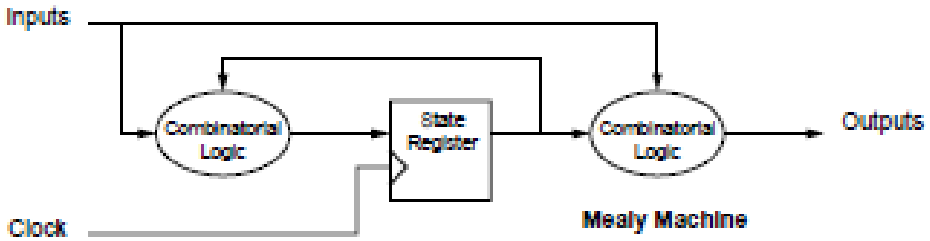
- Una de las herramientas más poderosas de circuitos digitales es la construcción de los circuitos mediante máquinas de Estados Finitos (FSM- Finite State Machines).
- Una FSM combina lógica combinacional y secuencial.
- Una FSM permite la toma de decisiones con base en ciertas señales de entrada y el Estado actual de la máquina.

FSMs

- Una FSM tiene un funcionamiento síncrono y toma todas las decisiones en el instante de llegada del reloj.
- Hay dos tipos clásicos de FSMs:
  - Máquina de Mealy
  - Máquina de Moore

Tipos de FSMs



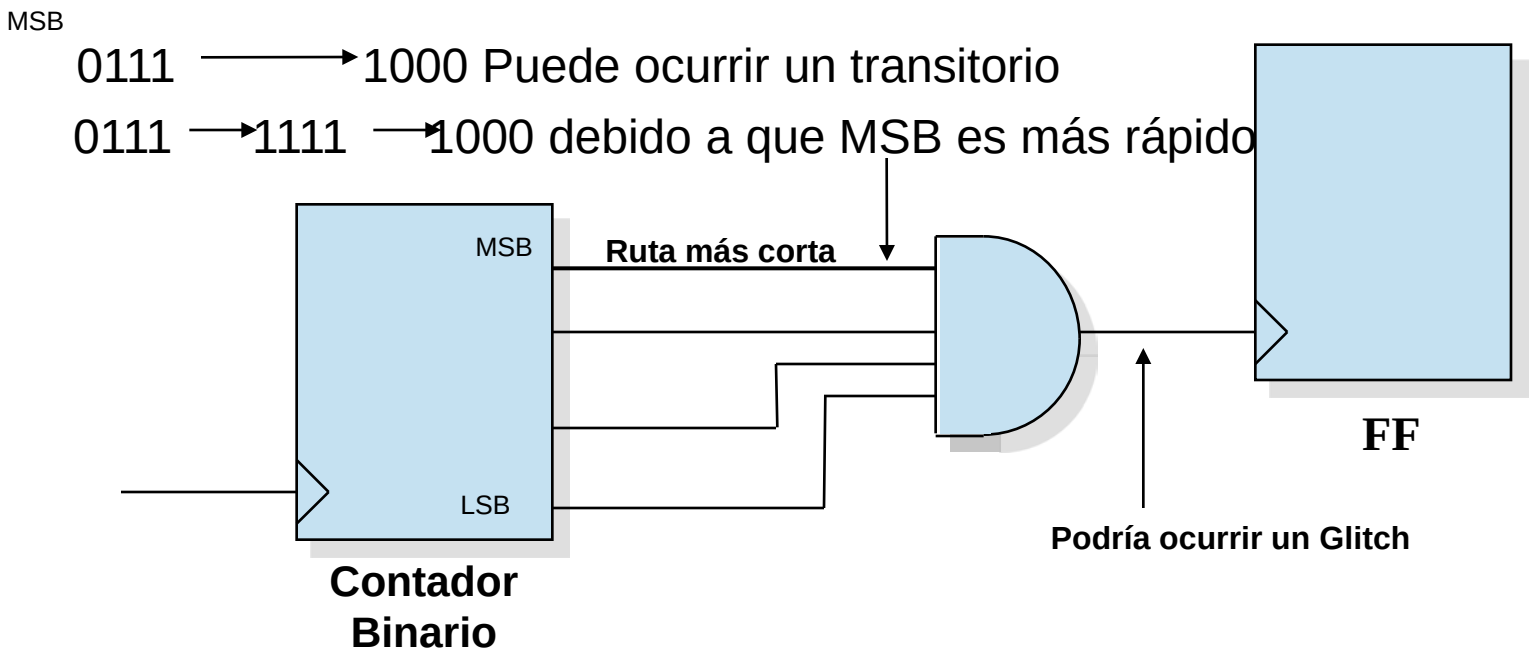


- Máquina de Moore:
  - Es la más simple de las dos.
  - Las salidas sólo dependen del estado
- Máquina de Mealy:
  - Las salidas dependen del estado más las entradas.
  - Es más flexible
  - Es más difícil de comprender su operación

Qué está mal con las FSMs de Mealy y Moore?

- Al tener lógica combinacional en las salidas que decodifican el estado (Moore) y también las salidas (Mealy), puede haber ciertos riesgos que hacen caer a los incautos:
  - Los dispositivos secuenciales que funcionan con relojes pueden cambiar su estado cuando ocurren glitches (fallos en el sistema en forma de pulsos transitorios).

Glitch

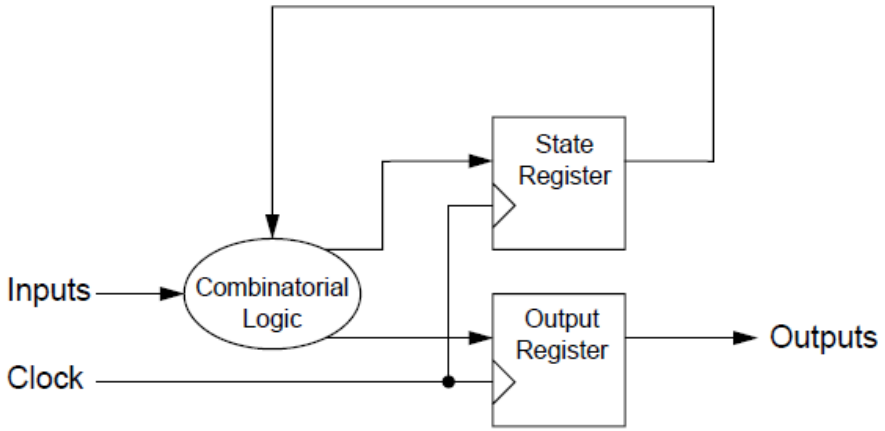

MSB
0111 → 1000 Puede ocurrir un transitorio
0111 → 1111 → 1000 debido a que MSB es más rápido
MSB
Ruta más corta
LSB
Contador Binario
FF
Podría ocurrir un Glitch

Es el código Gray una solución?

- La ventaja del código Gray es que sólo cambia un bit a la vez en el conteo.
- Sin embargo, diseñar máquinas con esta restricción sólo es práctico en máquinas muy simples.

Una FSM mucho mejor…

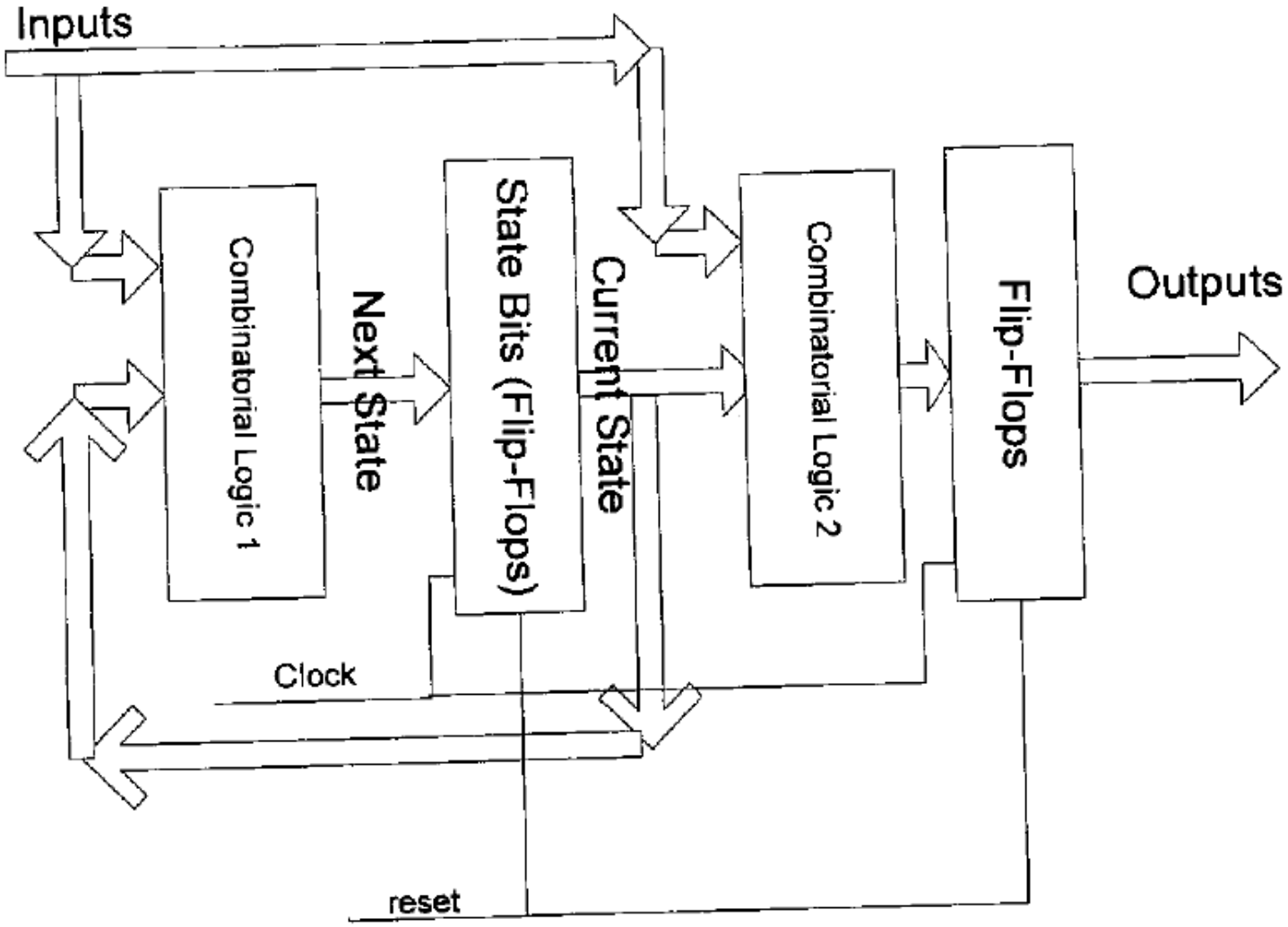
- Se agrega un Registro en las salidas (Output Register).
- La señal de reloj debe ser limpia.
- De esta manera, las salidas se actualizan con cada flanco de reloj y se garantiza que estarán libres de Glitches.
- La calidad de las salidas es independiente del número de estados o el número de salidas.



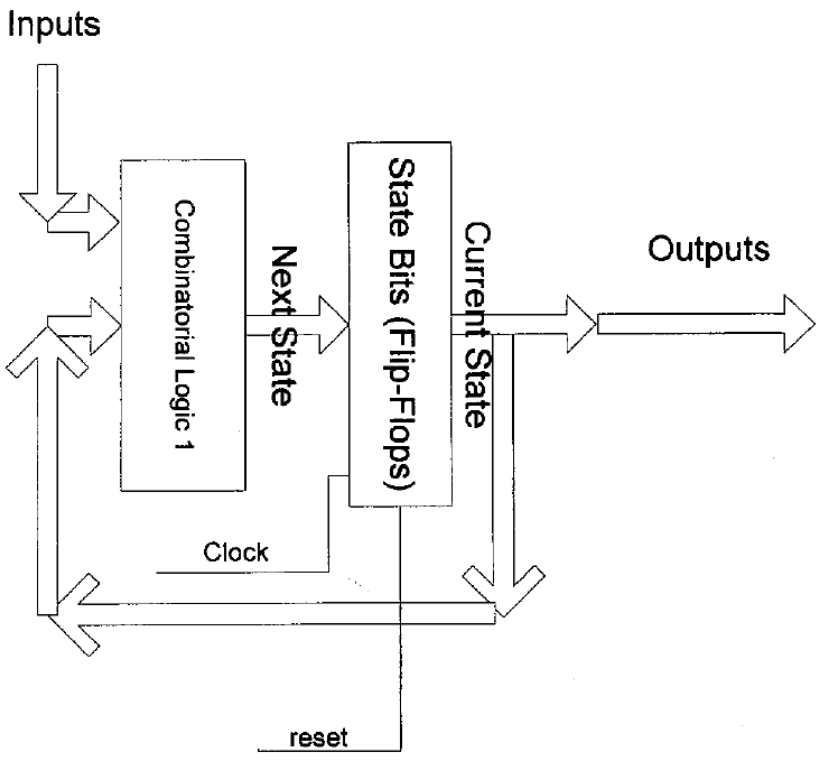
Desventajas de esta solución:

- Hay un retardo de un período de reloj que se adiciona a las salidas. Esto además confunde a los diseñadores que intentan entender la operación de la FSM.
- No hay dependencia directa de las salidas en las entradas.
- Por cada salida habrá un registro. Esto podría causar una gran cantidad de registros necesarios para la implementación.

La solución de los registros en las salidas se puede ver así….



Y los bloques se pueden mover para quedar así:



- Esto se logra:
  - Adicionando los bits de las salidas a los bits de estado y
  - Uniendo la lógica combinacional 2 a la lógica combinacional 1.
  - Y creando una nueva FSM que no tenga la lógica combinatoria 2 o los FF de salida.

Un método más eficiente…

- Se puede diseñar la lógica combinatoria 2 de manera que se elimina la necesidad de poner un registro para cada salida. Una salida puede ser:
  1. Manejada directamente por un bit de estado
  2. Manejada por la negación de un bit de estado
  3. Manejada por una lógica combinatoria que dependa de sólo 1 bit de estado, pero puede depender de otras entradas o constantes
  4. Un multiplexor cuyas entradas son entradas o constantes. Las entradas del mux son seleccionadas por los bits de estado o sus bits negados.

Explicación de cada caso:

- Casos (1) y (2) evidentemente son libres de glitch
- Caso (3):
  - Los glitches no se generan por el bit de estado, pero se podrían generar si las entradas cambian y la salida resultante es la misma.
  - Esta opción puede usarse principalmente si las entradas son constantes o casi-constantes (si permanecen constantes durante la operación de la FSM).
- Caso (4):
  - La salida no cambiará mientras los bits de estado que seleccionan la salida no cambien.
  - Es permitido usar tantos bits de estado como sean necesarios para seleccionar las diferentes entradas

Pasos para diseñar FSMs

- Defina la misión de la FSM
- Defina las entradas y salidas de la FSM y su propósito
- Dibuje el Diagrama de Estados en lenguaje natural (Español, Inglés)
- Dibuje el Diagrama de Estados definiendo eventos, transiciones, estados de salidas
- Codificación en HDL:
  - Defina los estados y su codificación (puede asignar nombres especiales a algunos bits de estado si lo desea)
  - Defina los registros que almacenan los estados
  - Defina el cambio de estados mediante un case específico para esto
  - Defina las salidas que no son generadas por bits de estado mediante lógica combinacional (assign), **poniendo atención en que no causen Glitches**!

Ventajas del Método de Diseño

- Libre de Glitches!
- Metodología Universal de Diseño para las FSMs
- Always únicos, verificados para las transiciones de estado: Usted sabe cómo se sintetiza (no hay sorpresas asíncronas debidas a always con estructuras complejas). Utilice always “full case” para minimizar lógica innecesaria.
- Código muy legible y limpio
- El paso de los Diagramas de Estados a HDL es directo
- Se puede extraer el diagrama de estados directamente del código HDL
- La adición o eliminación de estados es muy fácil y segura
- Si se usan bits de estado para las salidas:
  - De los códigos de los estados, usted puede saber exactamente como se comportan las salidas y cambiarlas es fácil.
  - Se elimina el retardo de un período de reloj para almacenarlas (si se usara un registro aparte)
  - Se reduce circuitería (no se requiere lógica combinacional para salidas, ni registros de salida)

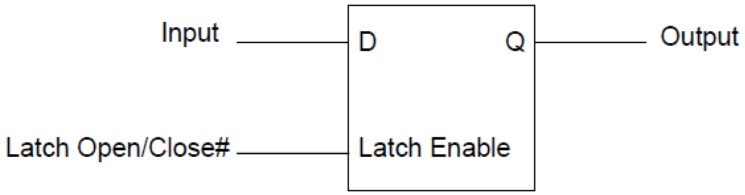
RECOMENDACIONES PARA DISEÑO DE FSM

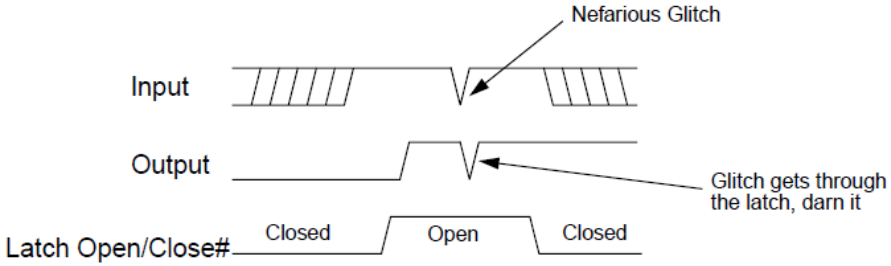
Estados Muertos

- Las FSMs podrían no usar todos los estados posibles (menos de 2^N).
- A los estados no usados se les llama “Estados muertos”
- Si la FSM llega a uno de estos estados por error, no se sabría cómo reaccione (podría bloquearse o empezar a hacer cosas locas)
- **Solución:** Los estados que no se utilicen se contemplan dentro de la opción “Default” del Case, la cual puede enviar a la FSM a un estado Inactivo o simplemente al estado inicial.

No Usar Latches!

- Es tentador usar latches porque utilizan menos circuitería que los Registros (alrededor del 60% de la circuitería de un Registro D).
- El problema es que en los Latch, los glitch en la entrada de datos o en la entrada de habilitación se propagan hacia la salida
- **Conclusión:** Se deben usar registros para brindar robustez al circuito ante glitches.





Alimente las entradas y Resets de su FSM

- La activación del Reset envía los Registros a un estado inicial.
- Si el Reset se desactiva cerca a un flanco de reloj, algunos FFs reaccionarán más rápido y otros más despacio, creando estados no deseados.
- Podría llegarse a un estado erróneo.
- Solución:
  - El reset debe ser sincronizado con la señal de reloj (se requieren circuitos sincronizadores)
- Recomendación:
  - **Todas las entradas a su FSM deben estar sincronizadas con el reloj! (No violar tiempos de Establecimiento y Retención!)**

Más recomendaciones

- Evitar tener que hacer muchas sincronizaciones entre los diferentes módulos
- **Todas las FSM deben operar con el mismo reloj! (un reloj Global)**
- En Xilinx, todos los FFs inician su operación en Cero, por tanto, **es recomendable que el código para el estado inicial tenga todos sus bits en cero** (los que no deben estar así se pueden negar)
- Si usted duda si una salida puede causar un Glitch, asegúrese que no lo haga poniendo un registro que la almacene

Ejemplo: Lectura de una memoria

- Misión:
  - Leer posiciones consecutivas de una memoria de manera indefinida

Análisis de las señales de entrada/salida

M29DW323DT, M29DW323DB

Figure 11. Read Mode AC Waveforms

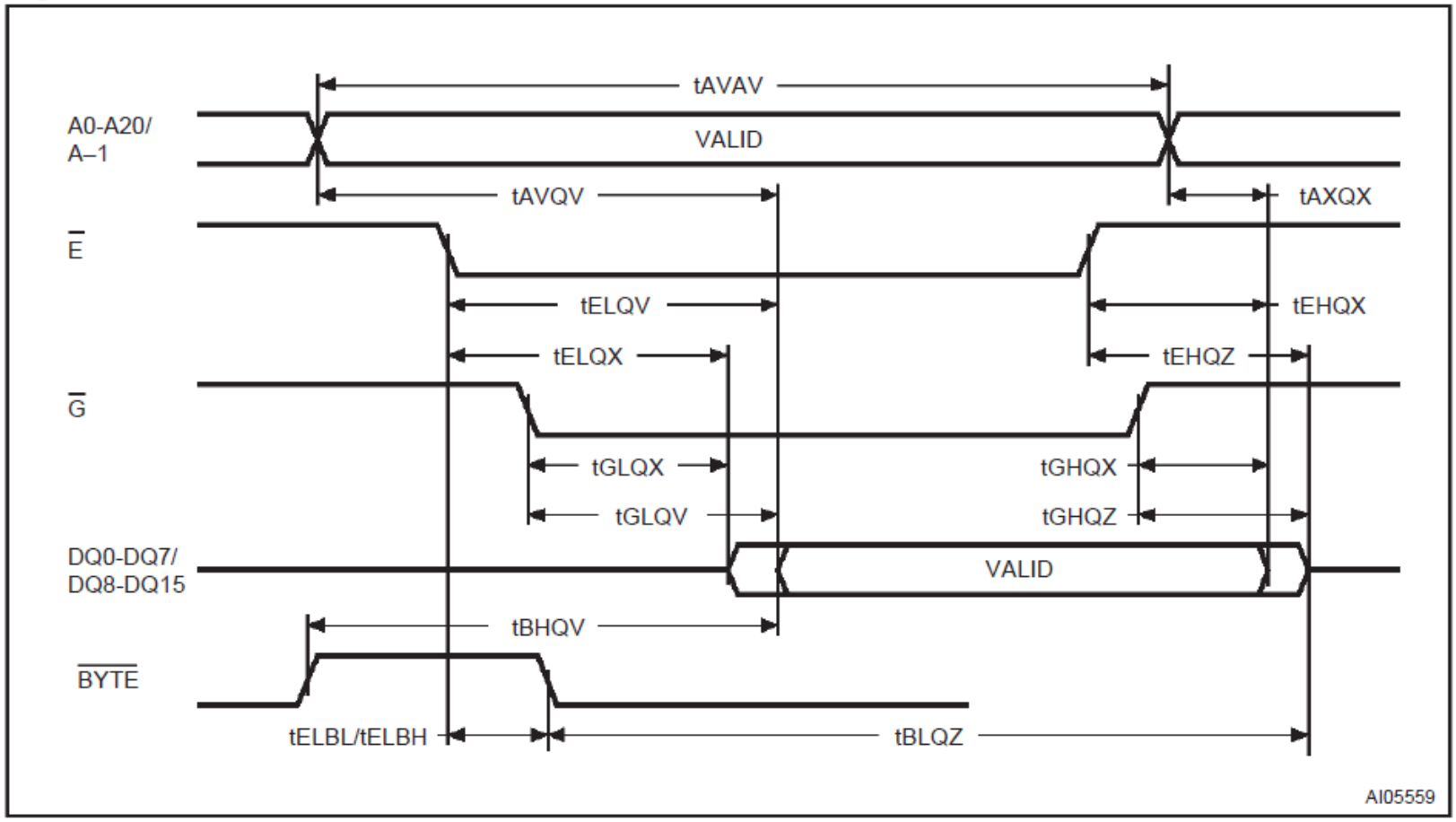
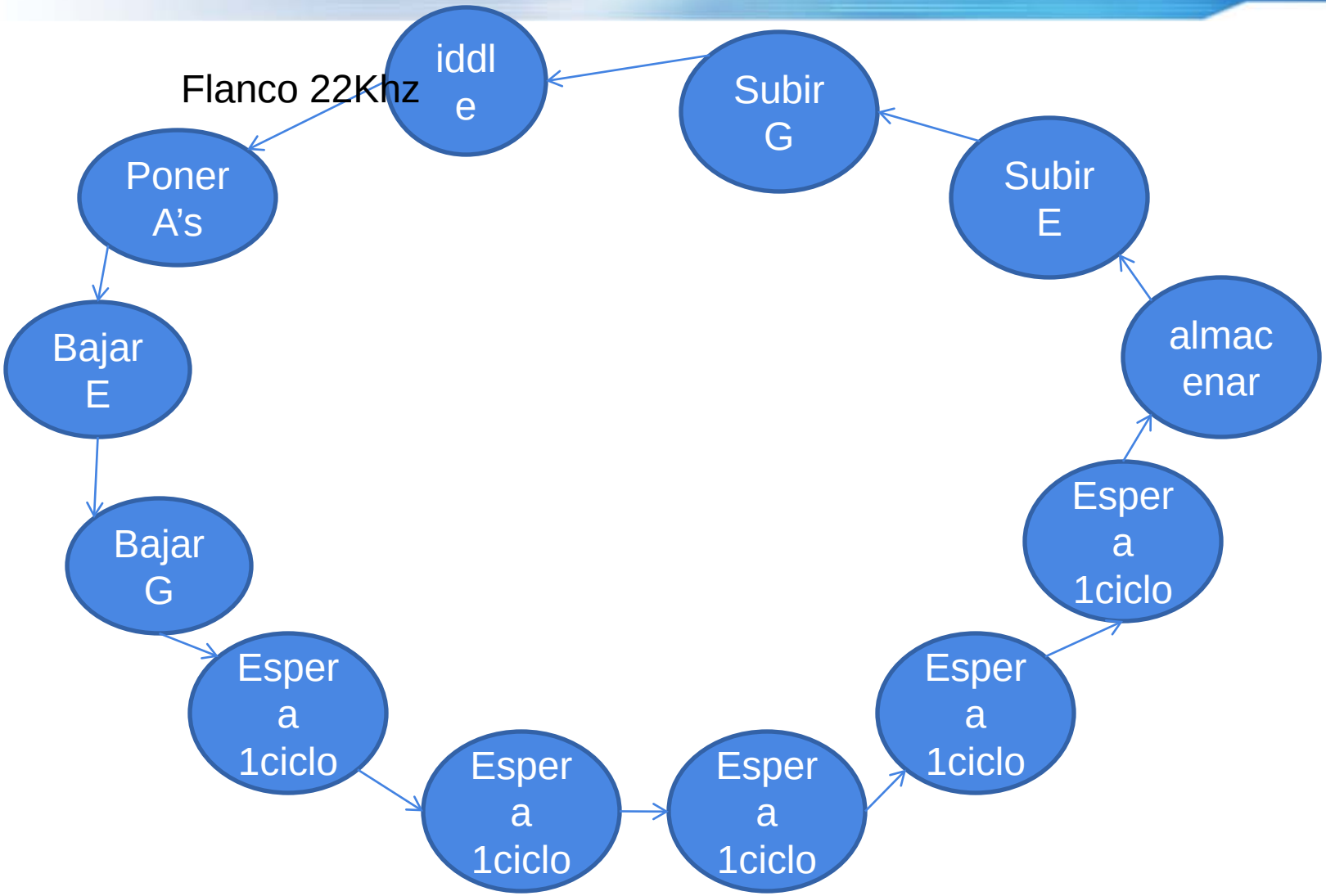


Diagrama de Estados para lectura Memoria Flash



Definir módulo FSM, variables y códificación estados

```
module fsmlecturamemoria (state, e, g, almacenar, clk_50Mhz, start, rst, direccion);

input start, clk_50Mhz, rst;
output [4:0] state;

parameter [21:1] DIR_INICIAL=21'h100000;

output reg [21:1] direccion=DIR_INICIAL;
output reg e,g,almacenar;

reg [4:0] state=1;
reg [4:0] next_state;
reg [21:1] direccion_next;

parameter [4:0] OCIOSO  = 1;
parameter [4:0] PONERDIR = 2;
parameter [4:0] BAJAR_E  = 3;
parameter [4:0] BAJAR_G  = 4;
parameter [4:0] ESPERA_CICLO_1= 5;
parameter [4:0] ESPERA_CICLO_2= 6;
parameter [4:0] ESPERA_CICLO_3= 7;
parameter [4:0] ESPERA_CICLO_4= 8;
parameter [4:0] ESPERA_CICLO_5= 9;
parameter [4:0] ALMACENAR  = 10;
parameter [4:0] SUBIR_E  = 11;
parameter [4:0] SUBIR_G  = 12;
```

Definir registros para los estados

```
always @(posedge clk_50Mhz or posedge rst) // sequential
begin
  if (rst) state <= OCIOSO;
  else state <= next_state;
  if (rst) state <= OCIOSO;
  else direccion <= direccion_next;
end
```

Definir lógica combinacional para siguiente estado

```
always @* // combinational logic for state
begin
  case(state)
    OCIOSO:   if (start) next_state = PONERDIR;
              else next_state = OCIOSO;

    PONERDIR:  next_state = BAJAR_E;
    BAJAR_E:   next_state = BAJAR_G;
       BAJAR_G:   next_state = ESPERA_CICLO_1;
    ESPERA_CICLO_1: next_state = ESPERA_CICLO_2;
    ESPERA_CICLO_2: next_state = ESPERA_CICLO_3;
    ESPERA_CICLO_3: next_state = ESPERA_CICLO_4;
    ESPERA_CICLO_4: next_state = ESPERA_CICLO_5;
    ESPERA_CICLO_5: next_state = ALMACENAR;
    ALMACENAR:   next_state = SUBIR_E;
       SUBIR_E:   next_state = SUBIR_G;
       SUBIR_G:   next_state = OCIOSO;
    default: next_state = OCIOSO;
  endcase
end
```

Definir la lógica combinacional por c/u de las salidas (agregar registro a la salida)

```
always @* // CKTO COMBINACIONAL SALIDA DIRECCION
begin
  case(state)

    PONERDIR: if (direccion == 21'h1FFFFF) direccion_next= DIR_INICIAL;
                                                    else direccion_next = direccion + 1;
    default: direccion_next = direccion;
  endcase
end

always @ (posedge clk_50Mhz) // CKTO COMBINACIONAL SALIDA E; esta definición agrega un registro a la salida que
        toma el valor e
begin
  case(state)

    BAJAR_E : e <= 0; //asignacion
         BAJAR_G:  e <=0;
    ESPERA_CICLO_1:         e <= 0;
    ESPERA_CICLO_2:         e <= 0;
    ESPERA_CICLO_3:         e <= 0;
    ESPERA_CICLO_4:         e <= 0;
    ESPERA_CICLO_5:         e <= 0;
    ALMACENAR:   e <= 0;
    default: e <= 1;
  endcase
end
```

Lógica de las salidas (continuación)

```
always @ (posedge clk_50Mhz) // CKTO COMBINACIONAL SALIDA g; esta definición agrega un registro a la salida que toma el valor
        g
begin
  case(state)

    BAJAR_G:  g <=0;
    ESPERA_CICLO_1:             g <= 0;
    ESPERA_CICLO_2:             g <= 0;
    ESPERA_CICLO_3:             g <= 0;
    ESPERA_CICLO_4:             g <= 0;
    ESPERA_CICLO_5:             g <= 0;
    ALMACENAR:   g <= 0;
         SUBIR_E: g <=0 ;
    default: g <= 1;
  endcase
end

always @ (posedge clk_50Mhz) // CKTO COMBINACIONAL SALIDA almacenar; esta definición agrega un registro a la salida que toma
        el valor g
begin
  case(state)

    ALMACENAR:   almacenar <= 1;
    default: almacenar <= 0;
  endcase
end
endmodule
```

Diseño Jerárquico (Diseño Modular)

- Estrategia de diseño: Divide y vencerás
- Un sistema grande se sub-divide en sistemas más pequeños
- Cada sub-sistema es una FSM
- Es necesario interconectar diferentes FSMs que intercambian información entre sí.
- Se requiere un protocolo de comunicación para enviar/recibir los parámetros y señales entre FSMs.

Ventajas del diseño Modular:

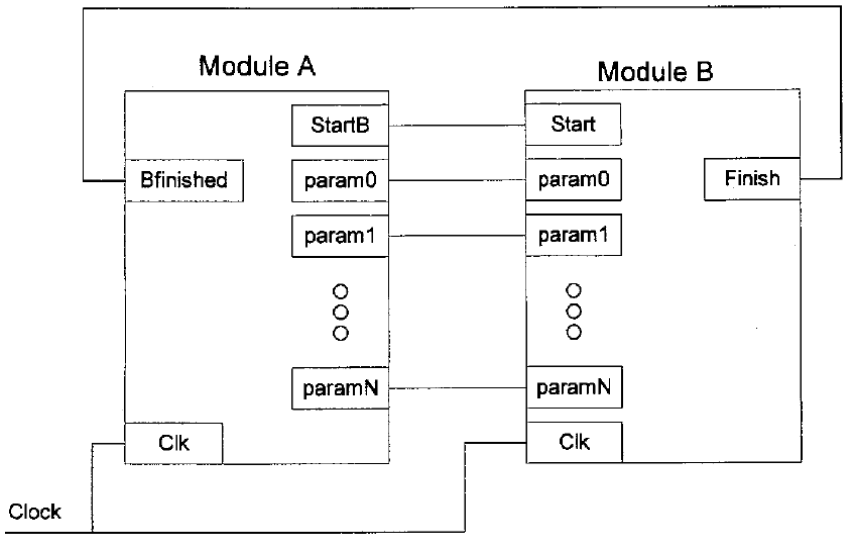
- Separa el código (el hardware) en unidades tratables
- Permite el trabajo en equipo: Cada módulo puede ser desarrollado por una persona diferente
- Permite el re-uso de los módulos

Interconexión entre FSMs

- El protocolo de comunicación entre FSMs debe estandarizarse:
  - Permite mantener el mismo mecanismo de comunicación para diferentes módulos
  - No se está re-inventando la rueda cada vez que se requiera una comunicación entre FSMs
  - Facilita la modularización del diseño
  - Fácil integración de módulos hechos por diferentes diseñadores
  - Un mismo módulo puede ser re-utilizado por diferentes módulos, permitiendo códigos más compactos

Sugerencia: El protocolo Start-Finish

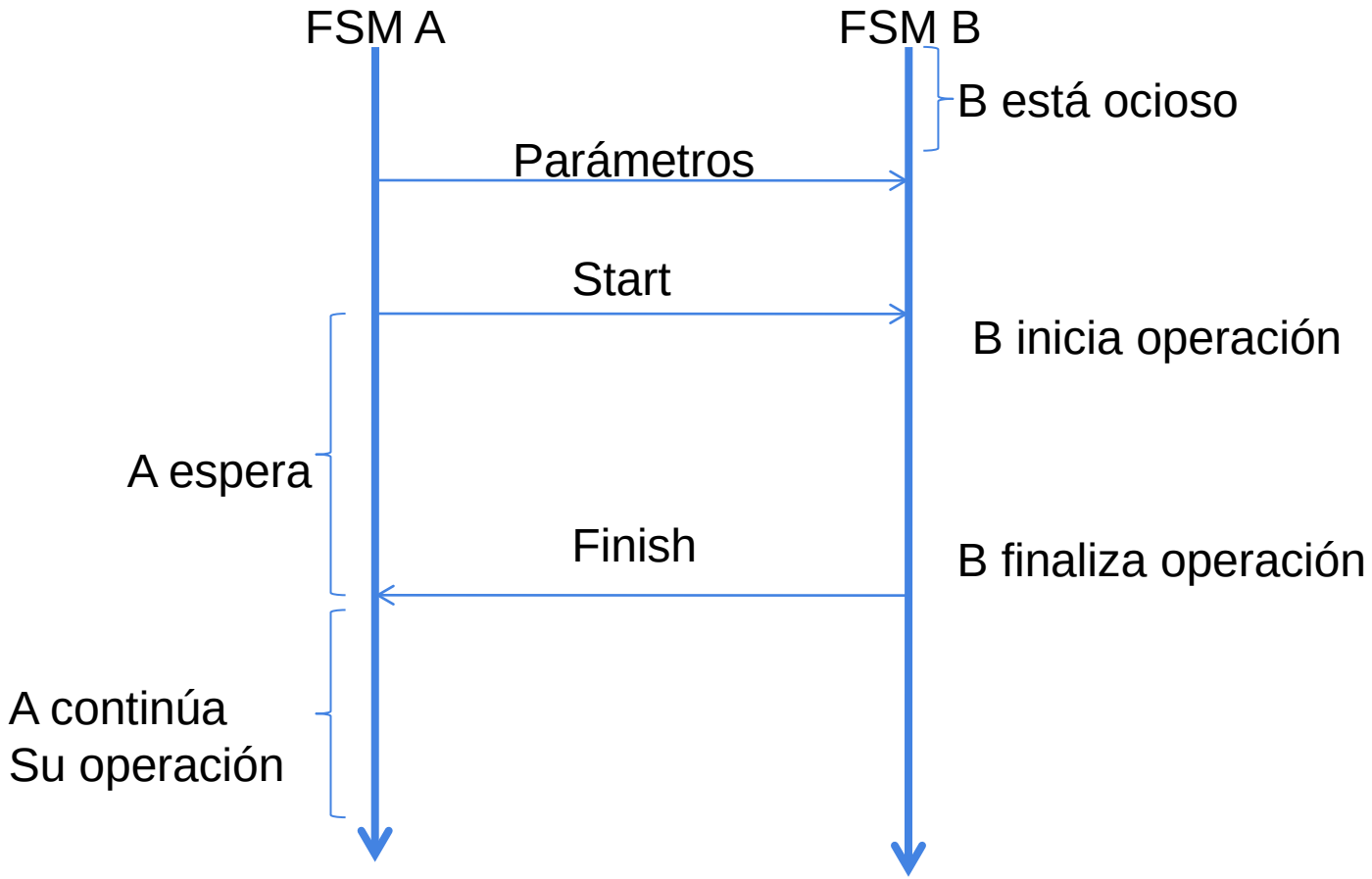
- Suponga dos módulos: A y B
- Suponga que A llama a B (entrega parámetros a B)
- A entrega los parámetros param1 hasta paramN
- La señal de reloj es la misma para las dos FSMs
- La conexión entre módulos debe hacerse así:



Operación del Protocolo Start-finish

- El módulo A llama al módulo B
- El módulo A entrega los parámetros de B a través de las entradas de B.
- El módulo A “activa” la señal Start que entra al módulo B
- B se encuentra por defecto en su estado “Ocioso”
- Cuando B recibe el pulso Start, inicia su operación con los valores que tiene en las entradas
- Cuando B finaliza su operación, envía un pulso por su salida Finish

Operación del protocolo Start-Finish



Características particulares del protocolo

- La señal de reloj es la misma para las dos FSMs (de lo contrario, se requeriría sincronizar los relojes mediante circuitos de sincronización)
- Los pulsos Start y Finish tienen un ancho de 2 períodos de reloj. Esto da una seguridad extra en los requerimientos de tiempo (setup y hold) en los bits de estado de B.
- Los parámetros de entrada a B deben permanecer estables desde que se activa Start hasta que se activa Finish. Esto evita tener que usar registros para almacenar los parámetros de entrada a B. Esto no adiciona complejidad a A porque igual debe quedarse esperando.

Características especiales del protocolo

- El estado Ocioso de B permite que se dedique a reconocer la señal Start y pasar al estado de operación de B.
- Esto permite dar tiempo a que A coloque los parámetros y que se estabilicen.
- El módulo A no puede llamar al módulo B nuevamente antes de 3 períodos de reloj de haberse activado Finish. Esto asegura que el módulo B haya regresado a su estado Ocioso nuevamente y esté estable en él (y pueda reconocer el nuevo Start)

Algunas conclusiones

- Este protocolo permite hacer “llamados” a otros procesos de forma análoga a cuando se llaman subrutinas en software.
- Pero esto implica secuencias de procesos!!!!

Ejemplo: Módulo B

```
module cownter( //Inputs
                SM_CLK,
                RESET,
                START,
                AMOUNT,
                //Outputs
                FINISH);

input           START;
input[23:0]     AMOUNT;
input           RESET;
input           SM_CLK;

output          FINISH;

wire            SM_CLK;
wire            RESET;
wire            START;
wire[23:0]      AMOUNT;

wire            FINISH;

//Internal wiers/Registers
reg[5:0]        state;
reg[23:0]       counter;

parameter idle      =       6'b000000;
parameter enter     =       6'b001001;
parameter move      =       6'b000110;
parameter pulse     =       6'b010010;
parameter end_      =       6'b000011;
parameter finish1   =       6'b100100;
parameter finish2   =       6'b100101;

wire    sample  =       state[3];
wire    cnt_clk =       state[4];
assign  FINISH  =       state[5];
```

```
always @(posedge SM_CLK     or negedge RESET)
                if(~RESET)
                        begin
                                state   <=      idle;
                        end
                else
                        begin
                                case (state)
                                        idle    : begin
                                                  if(START)
                                                    begin
                                                      state <= enter;
                                                    end
                                                  else
                                                    state <=    idle;
                                                  end
                                        enter   :       state   <=      pulse;
                                        pulse   :       state   <=      end_;
                                        end_    :       begin
                                                          if(counter == 24'h0)
                                                            state <=    finish1;
                                                          else
                                                            state <=    pulse;
                                                        end
                                        finish1 :       state   <=      finish2;
                                        finish2 :       state   <=      idle;
                                        default;
                                endcase
                        end


always  @(posedge cnt_clk or posedge sample)
        if(sample)
                counter <=      AMOUNT;
        else
                counter <=      counter - 24'h1;


endmodule
```

Ejemplo: Módulo A

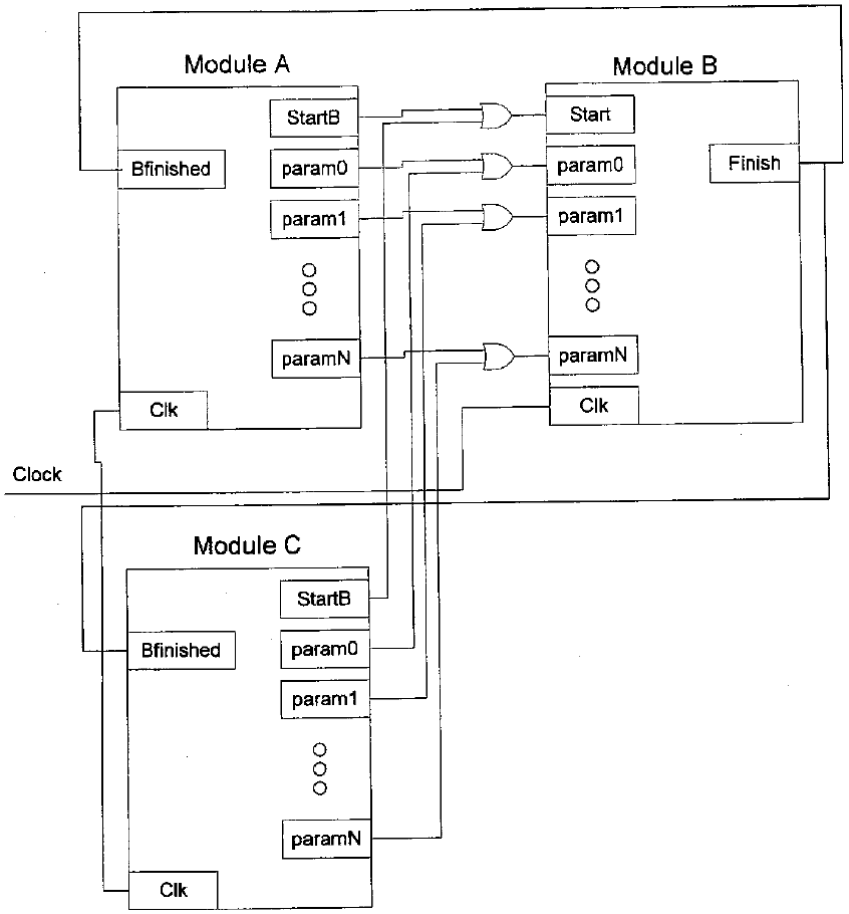
```
module sweeper(enabled,
               upper_scan_limit,
               lower_scan_limit,
               scan_step,
               scan_DC_out,
               lock_metric_in,
               lock_metric_clock,
               t_allow_lock_count,
               t_check_status_count,
               is_locked,
               current_lock_metric,
               sm_clk,
               Gone_Into_Lock_Threshold,
               Gone_Out_of_Lock_Threshold,
               hold_position,
               reset
            );

parameter scan_width = 32;
parameter estimator_width = 8;

input enabled;
input[scan_width-1:0] upper_scan_limit,  //a positive number !
                      lower_scan_limit,  //a negative number !
                      scan_step;
output[scan_width-1:0] scan_DC_out;
input[estimator_width-1:0] lock_metric_in;
input[23:0] t_allow_lock_count,t_check_status_count;
input lock_metric_clock,sm_clk,reset;
output is_locked;
output[estimator_width-1:0] current_lock_metric;
input [estimator_width-1:0] Gone_Into_Lock_Threshold,Gone_Out_of_Lock_Threshold;
input hold_position;

wire [23:0] current_count;
wire [estimator_width-1:0] actual_lock_metric_in;
reg[scan_width-1:0] scan_counter;

reg[11:0] state;
                                          //109876543210
parameter idle                      =     12'b000000000000;
parameter general_reset             =     12'b001100100001;
parameter start_wait_for_lock1      =     12'b001010000010;
parameter start_wait_for_lock2      =     12'b001010000011;
parameter wait_for_counter_finish1  =     12'b011000000100;
parameter allow_lock_update_disable1=     12'b000000000101;
parameter check_lock_estimator1     =     12'b000000000110;
parameter step_scan_counter         =     12'b000001000111;
parameter begin_next_status_check1  =     12'b100010001000;
parameter begin_next_status_check2  =     12'b100010001001;
parameter wait_for_next_status_check=     12'b110000001010;
parameter allow_lock_update_disable2=     12'b100000001011;
parameter check_lock_estimator_2    =     12'b100000001100;
parameter hold_unlocked             =     12'b000000001101;
parameter hold_locked               =     12'b100000001110;


wire reset_scan_counter = state[5];
wire inc_scan_counter = state[6];
wire timing_counter_start = state[7];
wire timing_counter_reset = reset && (~state[8]);
wire count_select = state[9];
wire allow_lock_metric_update = state[10];
assign is_locked = state[11];
```

```
assign current_count = count_select ?  t_allow_lock_count : t_check_status_count;

clock_interface lock_metric_interface(.indata(lock_metric_in),
                                      .outdata(actual_lock_metric_in),
                                      .inclk(lock_metric_clock),
                                      .outclk(sm_clk),
                                      .in_CE(allow_lock_metric_update)
                                     );

defparam lock_metric_interface.width = 8;

assign current_lock_metric = actual_lock_metric_in;

cownter timing_counter( .SM_CLK(sm_clk),
                        .RESET(timing_counter_reset),
                        .START(timing_counter_start),


                .AMOUNT(current_count),
                .FINISH(timing_counter_finish));

signed_bigger_than comp1(.A(actual_lock_metric_in),.B(Gone_Into_Lock_Threshold),.result(gone_into_lock));
signed_bigger_than comp2(.A(Gone_Out_of_Lock_Threshold),.B(actual_lock_metric_in),.result(gone_out_of_lock));

defparam comp1.width = estimator_width;
defparam comp2.width = estimator_width;

assign scan_DC_out = scan_counter;

always @(posedge inc_scan_counter or posedge reset_scan_counter)
begin
        if (reset_scan_counter)
        begin
                scan_counter <= lower_scan_limit;
        end else
        begin
        if ((!scan_counter[scan_width-1]) && (scan_counter > upper_scan_limit))
        //the '>' comparison only has meaning
        //if the current scan counter count is positive
                begin
                                scan_counter <= lower_scan_limit;
                end else
                begin
                                scan_counter <= scan_counter + scan_step;
                end
        end

end
```

Ejemplo: Módulo A

```
always @(posedge sm_clk or negedge reset)
begin
        if (!reset)
        begin
                state <= idle;

        end else
        begin

                case (state) /*synthesis full_case*/
                idle : if (enabled)
                                state <= general_reset;
                        else
                                state <= idle;

                general_reset : state <= start_wait_for_lock1;
                start_wait_for_lock1 : begin
                                state <= start_wait_for_lock2;
                               end
                start_wait_for_lock2 : begin
                                state <= wait_for_counter_finish1;
                               end

                wait_for_counter_finish1 :  if (!enabled) state <= idle; else
                                begin
                                    if (timing_counter_finish)
                                    begin
                                        state <= allow_lock_update_disable1;
                                    end else
                                        state <= wait_for_counter_finish1;
                                end

                allow_lock_update_disable1 : state <=  check_lock_estimator1;

                check_lock_estimator1 : begin
                                    if (hold_position)
                                    begin
                                        state <= hold_unlocked;
                                    end else
                                    begin
                                        if (gone_into_lock)
                                            state <= begin_next_status_check1;
                                        else
                                        state <= step_scan_counter;
                                    end
                                end

                hold_unlocked : if (!hold_position)
                                state <= start_wait_for_lock1;
                            else
                                state <= hold_unlocked;

                step_scan_counter : state <= start_wait_for_lock1;

                begin_next_status_check1 : state <= begin_next_status_check2;

                begin_next_status_check2 : state <= wait_for_next_status_check;

                wait_for_next_status_check :if (!enabled) state <= idle; else
                                    begin
                                        if (timing_counter_finish)
                                                state <= allow_lock_update_disable2;
                                        else
                                          state <= wait_for_next_status_check;
                                    end
                allow_lock_update_disable2 :  state <= check_lock_estimator_2;

                check_lock_estimator_2 : begin
                                    if (hold_position)
                                    begin
                                        state <= hold_locked;
                                    end else
                                    begin
                                            if (gone_out_of_lock)
                                            begin
                                                    state <= step_scan_counter;
                                            end else
                                            begin
                                                    state <= begin_next_status_check1;
                                             end
                                    end
                            end

                hold_locked : if (!hold_position)
                                state <= begin_next_status_check1;
                            else
                                state <= hold_locked;

                endcase
                end

        end

endmodule
```

Algunas conclusiones



- El sub-módulo llamado puede quedar al mismo nivel del módulo que lo llama. Así, no hay anidamiento y B puede ser llamado por otros módulos. Pero esto implica ciertas restricciones!!! (compuertas OR en las entradas!!!)

Restricciones para el caso anterior

- Los módulos A y C nunca deberán llamar a B al mismo tiempo!!!
- Impedir esto es responsabilidad del diseñador
- Cuando las salidas de A y C no están activas, deben estar en cero
- Esto hace que no afecten la salida de la OR