# P3 Sessió 4: Controlador de Byte

Un cop verificat el controlador de bit, ja tenim tots els mòduls bàsics i podem procedir a dissenyar el controlador de byte que serà el cervell del nostres mestre $I^2C$. El controlador de byte s'encarregarà de gestionar la resta de mòduls en funció de les ordres rebudes a traves del bus de dades. Per facilitar-ne la verificació, en aquest punt generarem el nostre fitxer top del nostre mestre $I^2C$, on definirem totes les connexions entre mòduls.

## Objectius

- **Disseny i síntesis d'una màquina d'estats** que control l'enviament del byte
- Disseny del top del sistema amb les instàncies dels diferents mòduls del mestre $I^2C$ seguint l'arquitectura definida.
- **Auto-Test i verificació del mestre $I^2C$**, reaprofitant tasques i funcions ja desenvolupades.

## Tasques a realitzar

1. Determineu les entrades i sortides de la màquina d'estats, en base a l'arquitectura del sistema.
2. Identifiqueu i enumereu la seqüencia que ha de seguir per enviar (escriure) un byte i per rebre (llegir) un byte.
   Ajut 1: Primer establiu què s'ha d'escriure als registres i en quin ordre per genera cada tipus de transferència (escriptura o lectura).
   Ajut 2: Dibuixeu el diagrama temporal de les entrades i sortides de la controlador de byte.
   Ajut 3: Enviar l'adreça de 7-bits més el bit de direcció és el mateix que enviar un byte. Es considera una operació d'escriptura.
   Ajut 4: Per generar una senyal de *Start*, repetició de *Start* i *Stop* obligueu que estigui definit si la transmissió és d'escriptura o lectura al registre de comandament.
   Ajut 5: Ha de ser possible que s'activin el bits *Start*, *Write*, i *Stop* del registre de comandament, i es generi la seqüència correcta completa de transmissió d'un sol byte, el mateix per bit de *Read*.
3. Dibuixeu el diagrama d'estats que implementi la seqüencia establerta.
   Feu que la ordre de lectura tingui prioritat sobre l'ordre d'escriptura, bits *Read* i *Write* del registres de comandament.

4. Genereu un fitxer anomenat *i2c_master_top* que serà el top del nostre mestre $I^2C$ on heu d'instanciar tots els mòduls generats en les sessions anteriors incloent el *i2c_master_byte_ctrl*.
5. Codifiqueu la màquina d'estats que anomenarem *i2c_master_byte_ctrl*.
6. Dissenyeu el test que verifiqui la transmissió i recepció de dades el DUT és el *i2c_master_top*.
   - Recordeu afegir la descripció Verilog dels buffers de tres estats.

     ```
     assign scl = sclPadEn ? 1'bz : sclPadOut;
     assign sda = sdaPadEn ? 1'bz : sdaPadOut;
     assign sclPadIn = scl;
     assign sdaPadIn = sda;
     ```

   - Per simular les resistències de pull-up afegiu en el test testbench s'utilitza la funció de verilog: pullup nom_instancia (nom_linea).

     ```
     pullup i_P1(scl);
     pullup i_P2(sda);
     ```

   - Feu el test utilitzant els registres de configuració i control per realitzar transferències de escriptura i lectura. Primer alliberant el bus (generant senyal ***Stop***) i després sense alliberar el bus, és a dir, repetint el senyal ***Start*** entre operacions.
   - Primer comproveu que escriviu i llegiu correctament els registres de configuració i control. *Podeu reutilitzar les tasques dissenyades durant les sessions anteriors.*
   - Dissenyeu una tasca (*waitEnd*) que monitoritzi el bit d'estat *busy* del registre de control, és a dir, ha d'esperar que el bit *busy* passi de 1 a 0.
   - Al campus virtual trobareu un model d'esclau $I^2C$ molt senzill anomenat *i2c_slave_model*. Recordeu adjuntar el model als projectes de ModelSim.
7. Creu un fitxer SDC on heu d'especificar:
   - Un rellotge del sistema de 100 MHz amb la comanda *create_clock*.
   - Els senyals d'entrada i sortida del xip (FPGA/ASIC) tenen un retard associat. Aquest retard és pot especificar en el fitxer de restriccions SDC per tal que les eines de síntesis el tinguin present a l'hora de sintetitzar i de "mapejar" i implementar el nostre disseny. Per fer-ho, s'utilitzen les comandes *set_input_delay* i *set_output_delay*.

     ```
     set_input_delay -clock <clockName> <delay value in ns> [get_ports <portName>]
     ```

```
set_output_delay -clock <ClockName> <delay value in ns> [get_ports
 <portName>]
```

La ordre *set_input_delay* especifica el temps d'arribada de dades als ports d'entrada especificats en relació amb el rellotge especificat per l'opció -clock. On el rellotge ha de fer referència al nom del rellotge al disseny. Per tant, determinarà el temps disponible per dur la senyal d'entrada des del port fins el registres intern (Tclk - InputDelay). De la mateixa manera l'ordre *set_output_delay* especifica el temps requerit en que dades han d'estar als ports de sortida especificats en relació amb el rellotge especificat per l'opció *-clock*. (Noteu que les comandes tenen més paràmetres que els esmentats. Es recomana consultar l'ajuda del Quartus referent a les ordres TCL.)

En el nostre cas modelitzarem totes les entrades i sortides amb un retard de 2 ns. Per fer-ho heu d'incloure les següents comandes al fitxer SDC, on heu de substituir el <nomClk> pel nom que heu donat al rellotge de 100 MHz que heu definit.

```
set_input_delay -clock <nomClk>  2.0 [all_inputs]
set_output_delay -clock <nomClk> 2.0 [all_outputs]
```

8. Sintetitzeu i verifiqueu el disseny per FPGA **Cyclone IV E EP4CE22F17C6**. No us oblideu d'incloure el fitxer de SDC. Definiu els pins d'entrada i sortida al GPIO.
9. Comproveu que el disseny està completament definit (no té cap *unconstrained path*) i que compleix els requisits temporals.
10. Simuleu la *netlist* generada amb el mateix fitxer de testbench. Per simular a nivell de porta i veure els retards interns seguiu la guia (Simulacio Netlist Quartus) que trobareu al campus virtual.

# Entrega

Un fitxer ZIP amb el directori de treball:

1. A la carpeta **rtl**: el codi RTL.
2. A la carpeta **tb**: el codi del testbench complet.
3. A la carpeta **sdc**: el fitxer de restriccions temporals (.sdc).
4. Dins la carpeta **doc** hi heu de posar l'informe complimentat que trobareu al campus virtual, que constarà de:
    - Tipus i explicació de la màquina d'estats implementada i diagrama d'estats.
    - Enumerar les tasques del testbench i explicar breument què fan.

- Captures de les simulacions funcionals, amb una explicació breu i ressaltant les zones d'interès.
- Captura del terminal del ModelSim amb els missatges de l'auto verificació.
- Captura del esquema RTL resultant de la síntesi (no oblideu el diagrama d'estats) amb el Quartus i taula on consti freqüència màxima d'operació, i recursos utilitzats de la FPGA Cyclone IV E EP4CE22F17C6.
- Captures de les simulacions post-síntesis, amb una explicació breu i ressaltant les zones d'interès.

# Annex Pràctica 3

## Màquines d'estat

Una màquina d'estats (en anglès, *Finite State Machine*) és un model que utilitzem per controlar el comportament d'un sistema. En el cas d'aquesta assignatura, utilitzem les màquines d'estat per controlar el comportament de circuits digitals.

En la **Figura 9** es presenta el diagrama de blocs d'una màquina d'estats. Qualsevol màquina d'estats consisteix **d'un bloc de lògica combinacional per al càlcul de l'estat futur** (*i.e.* definir les transicions d'estats), **un bloc de lògica seqüencial** per actualitzar l'estat amb l'arribada del flanc de rellotge, i **un últim bloc per al càlcul de les sortides** generades per la màquina d'estats. Aquest últim bloc pot ser combinacional o seqüencial segons les nostres necessitats.

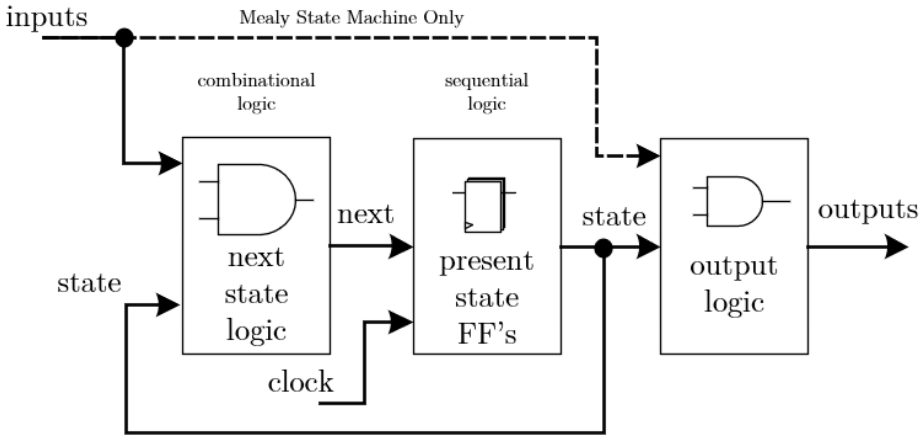


**Figura 9.** Diagrama de blocs d'una màquina d'estats (imatge extreta de l'article "*State Machine Coding Styles for Synthesis*", SNUG 1998)

## Disseny d'una màquina d'estats pel control d'un semàfor

A continuació detallarem els diferents passos per a codificar, en llenguatge Verilog, una màquina d'estats per a controlar un semàfor. Cada bloc descrit en la **Figura 10** el definirem emprant un bloc *always*. Així doncs, tota màquina d'estats codificada en **Verilog** utilitzarà **3 blocs *always***.

Els requisits del sistema son:

- Quan pitgem un botó de la FPGA es genera la senyal "Boto" que estarà a 1 durant un cicle de rellotge i al següent cicle tornarà a 0. Canviarem d'estat quan pitgem el botó (senyal "Boto" igual a 1)
- L'ordre d'il·luminació de LEDs ha de ser verd, groc i per últim vermell, podent repetir aquesta seqüència els cops que es vulgui.
- Rellotge d'1MHz

## Diagrama d'estats

La definició del diagrama d'estats és un pas primordial ja que ens permet fixar les diferents variables de la màquina d'estats. És en aquest pas on fem una reflexió sobre les diferents entrades de la màquina d'estats, definim el nombre d'estats, definim les sortides i fins i tot definim si dissenyarem una màquina d'estats de Moore (*i.e.* les sortides només depenen de l'estat actual) o de Mealy (*i.e.* les sortides depenen de l'estat actual i de les senyals d'entrada).

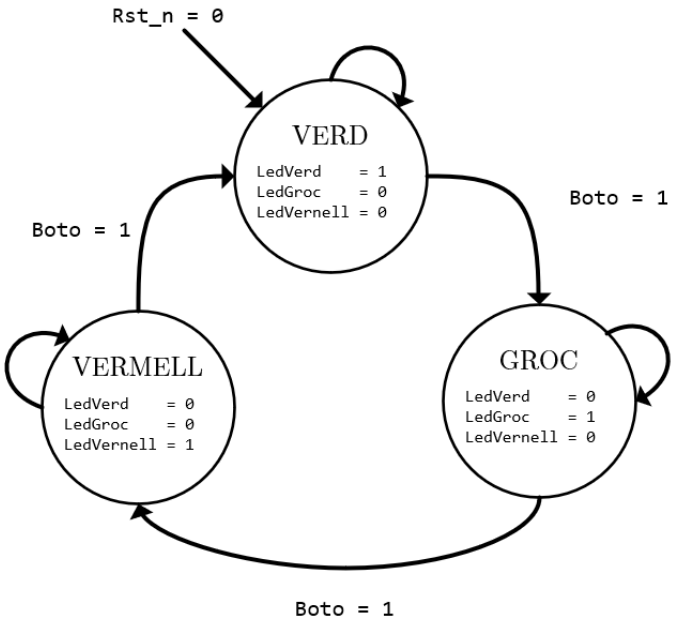


**Figura 10** Diagrama d'estats.

En el cas del semàfor, podem resoldre el problema definint **3 estats**, 1 per cada color de LED del semàfor. A més, en aquest exemple només hi ha **una senyal d'entrada**, "Boto", que ens permet passar d'un estat a un altre. Com sabem, el semàfor té 3 colors diferents,

així que definim **3 sortides** independents per a que il·luminin el LED que toca. La senyal d'entrada només ens indica el moment quan fer la transició d'un color a un altre, així doncs, podem definir la màquina d'estats com una **màquina d'estats de Moore**. Tot el que acabem de definir ho podem simplificar com el diagrama d'estats presentat a la F**igura 10**, on cada rodona defineix un estat, on podem veure que a cada estat s'hi defineixen les sortides i podem observar quines transicions entre estats podem realitzar i quina senyal les provoca.

Un cop tenim el diagrama d'estats podem passar a la codificació de la màquina d'estats.

## Definim les variables d'estat

Al definir qualsevol màquina d'estats estem definint dues variables d'estat: **l'estat actual** i **l'estat futur**. A ambdues variables se'ls hi assignaran valors dins de blocs *always*, així doncs es definiran com variables tipus registre.

En aquest cas, "state" i "next_state" son variables de 2 bits ja que s'utilitza una codificació binària. Per a aquesta assignatura és suficient utilitzar la codificació binària però n'hi ha d'altres com la codificació *gray*, *one-hot*, etc.

A banda de definir les variables d'estats també es defineixen variables internes. Aquestes es defineixen amb "parameter" i serveixen per assignar un nom que es pugui reconèixer a cada estat. Això ho fem per facilitar la lectura del codi i el seu test, sobretot quan dissenyem màquines d'estats amb un nombre elevat d'estats.

En l'exemple tindríem el següent:

```
// Definició variables d'estat
reg [1:0] state, next_state;    // Tenim 3 estats, codificació binaria
                                // = > necessitem 2 bits
// Definim variables internes
parameter [1:0]     VERD    = 2'b00,
                    GROC    = 2'b01,
                    VERMELL = 2'b10;
```

## Lògica de transició

La lògica de transició és el bloc combinacional que ens calcula el valor de l'estat futur segons el valor de l'estat actual i el valor de certes senyals d'entrada. És un bloc combinacional, per tant senyals com el rellotge o el *reset* del sistema no han d'aparèixer a la llista de sensibilitat; sí que hi ha d'aparèixer l'estat i aquelles senyals d'entrada que afecten a les transicions. En l'exemple del semàfor, el codi és el següent:

```
// Definim bloc comb. càlcul estat futur
always @(state, Boto) begin
  case(state)
     VERD :     if(Boto)    next_state = GROC;      //definim transicions estat VERD
                else        next_state = VERD;
     GROC :     if(Boto)    next_state = VERMELL;   //definim transicions estat GROC
                else        next_state = GROC;
     VERMELL:   if(Boto)    next_state = VERD;      //definim transicions estat VERMELL
                else        next_state = VERMELL;
     default:   next_state = VERD;                 //definim default,no hem definit totes
possibilitats state
    endcase
end
```

Amb la codificació binària emprada podríem definir fins a 4 estats. En aquest exemple en tenim 3, per aquest motiu definim l'estat *default*.

## Lògica seqüencial

La lògica seqüencial en la màquina d'estats serveix per actualitzar el valor de l'estat. Així doncs, en l'exemple del semàfor tenim la següent codificació:

```
// Definim lògica seqüencial (estat futur => estat actual)
always @(posedge Clk or negedge Rst_n)
    if (!Rst_n)     state <= VERD;                    //en cas de reset anem al VERD
    else            state <= next_state;              //actualitzem l'estat a cada cicle
rellotge
```

En aquest cas, "state" actualitza el seu valor quan arriba un flanc positiu de rellotge. A més, també definim l'estat inicial de la màquina d'estats en cas de *reset*.

## Càlcul de les sortides

Aquest últim bloc s'utilitza per assignar el valor desitjat a cadascuna de les sortides de la màquina d'estats, i és el bloc on es pot comprovar el tipus de màquina d'estats dissenyada (Moore o Mealy). Aquest bloc pot ser seqüencial o combinacional segons les necessitats del sistema. Pel cas del semàfor, definim un bloc combinacional com el següent:

```
// Definim lògica output
always @(state)                                 //no depèn de Clk = lògica combinacional
    case(state)
        VERD: begin                             //definim sortides en cada estat
                LedVerd     = 1'b1;
                LedGroc     = 1'b0;
                LedVermell  = 1'b0;             //Outputs només canviem amb estat => Moore
            end
        GROC: begin
                LedVerd     = 1'b0;
                LedGroc     = 1'b1;
                LedVermell  = 1'b0;
            end
        VERMELL: begin
                LedVerd     = 1'b0;
                LedGroc     = 1'b0;
                LedVermell  = 1'b1;
            end
        default: begin                          //en default tot a 0
                LedVerd     = 1'b0;
                LedGroc     = 1'b0;
                LedVermell  = 1'b0;
            end
    endcase
```

Totes les sortides depenen exclusivament del valor de la variable "state", així doncs s'ha dissenyat una màquina d'estats de Moore. El codi complert el trobareu a l'arxiu **fsm_semaphore.v** adjunt. També hi trobareu un arxiu de test per a comprovar el seu funcionament.