Simulació de sistemes digitals mitjançant llenguatges descriptors de maquinari

Juan Antonio Martínez Carrascal

PID_00170587

Universitat Oberta de Catalunya
www.uoc.edu

Índex

Introducció

Si recapitulem el que hem vist fins ara, veurem que hem evolucionat des de les plaques de silici fins al disseny de sistemes digitals de complexitat creixent.

Acadèmicament, podríem pensar que amb aquests conceptes i una dosi d'experiència, seria abordable la creació de circuits digitals arbitraris. Des d'un punt de vista pràctic, però, aquesta realització no és viable. Menys encara si pensem que com més va es demanen circuits més fiables, i amb temps de desenvolupament més curts.

És en aquest escenari en què es planteja la utilitat dels llenguatges descriptors de maquinari. Seria desitjable disposar d'un conjunt d'eines que ens permetés descriure el que fa el nostre circuit, aplicar-li un conjunt d'entrades, veure'n les sortides i comprovar que tot és correcte. En definitiva, simular el funcionament del circuit abans d'enviar-lo al complicat –i car– procés de fabricació. Estem parlant de simular per programari el nostre disseny.

Doncs bé, això és precisament el que veurem tot seguit. Com podem descriure i modelar el nostre disseny per tal de poder-lo simular? Com en podem veure els resultats de la simulació? Funciona el nostre circuit com esperem? Totes aquestes preguntes les podrem resoldre amb l'ajuda d'un llenguatge descriptor de maquinari.

En el nostre cas, farem servir el llenguatge VHDL (acrònim de *VHSIC hardware description language*). Tot i que n'hi ha d'altres, aquest té el suport de la indústria, és d'ús generalitzat i com veurem, serà possible disposar d'eines reals que facin totes les funcions descrites.

VHSIC

VHSIC és l'acrònim de *very high speed integrated circuits*.

Amb això tindrem el nostre disseny simulat. Ens mancarà un pas més, que serà la implementació real del disseny. VHDL també ens donarà eines per a fer-ho. De moment, però, ens centrarem en la simulació. Els mòduls següents tancaran el cercle, i ens portaran fins a la implementació real dels circuits que volem.

Objectius

En acabar aquest mòdul haurem de ser capaços del següent:

1. Entendre el concepte de llenguatge descriptor de maquinari.
2. Conèixer la filosofia del llenguatge VHDL.
3. Conèixer la sintaxi bàsica del llenguatge VHDL.
4. Crear bancs de proves per a simular els circuits.
5. Dissenyar circuits digitals i simular-los mitjançant VHDL.
6. Profunditzar en el disseny digital de sistemes complexos mitjançant màquines d'estats.
7. Dissenyar sistemes seqüencials síncrons.

1. Disseny digital: de les portes lògiques a la descripció del maquinari

Avui dia podem dir sense exagerar que la nostra vida seria impensable sense l'electrònica. Pràcticament qualsevol de les nostres activitats quotidianes té el suport d'un dispositiu electrònic –habitualment, digital. Una característica inherent a aquest tipus de dispositius és la seva evolució constant, i el fet que els demanem cada vegada més funcionalitats i que aquestes es facin de forma més ràpida.

És precisament aquesta evolució constant i ràpida –juntament amb la complexitat inherent als dissenys– un dels motors que fa que les eines de disseny digital tinguin una importància fonamental. Igual que avui dia qualsevol projecte arquitectònic té el suport d'una eina CAD[1] al darrere, no hi ha circuit electrònic de certa complexitat que s'implementi sense un disseny mitjançant un llenguatge descriptor de maquinari, o més genèricament, d'una eina de descripció de disseny electrònic (EDA[2]).

(1)*CAD* és la sigla de *computer aided design*.

(2)*EDA* és la sigla d'*electronic design automation*.

Un llenguatge descriptor de maquinari (HDL) descriu mitjançant codi el funcionament d'un circuit.

I és aquesta la filosofia amb la qual, als anys vuitanta, es desenvolupen els llenguatges descriptors de maquinari. Inicialment, amb la filosofia de descriure i simular el comportament dels circuits. Amb posterioritat, però, els diferents fabricants s'adonen que es pot aprofitar la potència del llenguatge per a efectuar implementacions pràctiques dels dissenys.

Malgrat que nosaltres centrarem el nostre estudi en el llenguatge VHDL, cal dir que n'hi ha d'altres. Nosaltres ens centrarem en el VHDL per tres motius fonamentals:

- Disposa del suport de l'IEEE i dels principals fabricants.
- És, juntament amb Verilog, el llenguatge més utilitzat.
- Disposa d'eines que permeten fer tot el cicle de disseny.

Web recomanat

A http://en.wikipedia.org/wiki/Hardware_description _language podreu trobar un recull de llenguatges descriptors de maquinari.

Abans d'entrar en la modelització de circuits amb VHDL, és important entendre tot el flux de disseny d'un circuit. La figura 1 mostra les etapes que constitueixen el procés.

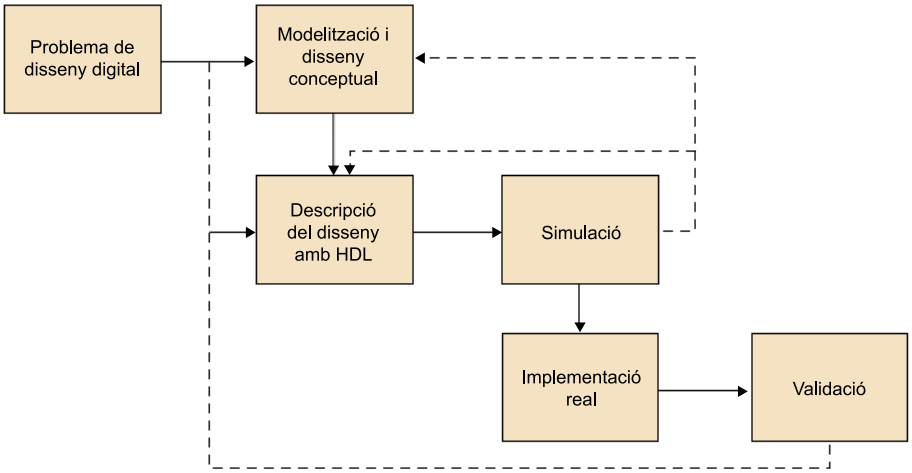

Figura 1. Procés de disseny amb un llenguatge descriptor de maquinari

Fixem-nos que el procés consta de sis subprocessos, que podem dividir en tres grans blocs:

1) Un primer bloc de disseny conceptual. Aquest disseny conceptual ens permet arribar a una descripció de comportament d'un circuit que resol una determinada problemàtica. Partint d'unes especificacions arribem a un diagrama de blocs (o fins i tot a una implementació amb portes) d'un determinat circuit. De fet, és el que hem fet en el mòdul anterior de l'assignatura.

2) Un segon bloc en el qual el disseny que hem fet es modelitza mitjançant VHDL. Aquesta modelització ens permet fer una simulació. En cas que la simulació no proporcioni els resultats desitjats, caldrà revisar bé el disseny o bé la codificació amb el llenguatge descriptor.

3) El tercer bloc, si el resultat de la simulació és l'esperat, consisteix en la implementació real del circuit. No oblidem que, fins al moment, el nostre circuit es troba dins un simulador. L'etapa final porta a la realització del prototipus i, si tot és correcte, a una possible fabricació massiva. Malgrat que hem fet la simulació, cal fer sempre una simulació del funcionament del dispositiu real. En cas que aparegui algun tipus d'error no detectat en el disseny i la simulació, caldrà tornar a repetir el procés.

Comencem, doncs, a descriure i simular el funcionament dels circuits que ja sabem dissenyar mitjançant VHDL.

2. Descripció de sistemes digitals amb VHDL

2.1. Què és VHDL?

Tot i que podem dir que la idea de descriure mitjançant un llenguatge el funcionament de l'electrònica és tan antic com l'electrònica en si, les bases de VHDL són del final de la dècada de 1980. En concret, l'IEEE el va definir com l'estàndard 1076 l'any 1987.

VHDL és un llenguatge que permet descriure l'estructura i el funcionament d'un circuit digital, de tal manera que amb aquesta descripció es pugui simular i finalment ser implementat en un dispositiu de lògica programable.

Òbviament, l'estàndard va anar evolucionant. És per aquest motiu que en algunes referències podreu trobar VHDL 87 –i en general VHDL i un any–, que fa referència a la versió concreta de VHDL. Malgrat l'evolució, les bases del llenguatge continuen essent les mateixes. En el fons l'hem d'imaginar tal com imaginem altres llenguatges de programació d'altres entorns, que han anat evolucionant però mantenen el gruix de la seva sintaxi.

VHDL va ser un llenguatge pensat originalment per a descriure i simular el circuit. Va ser la seva evolució la que va permetre que addicionalment, i amb el suport dels fabricants, pogués donar lloc també a la implementació real de circuits.

Com a llenguatge, disposa d'un entorn de programació o, millor dit, de descripció. De la mateixa manera que el llenguatge C presenta múltiples compiladors, el llenguatge VHDL també presentarà múltiples entorns. En principi tot el que pertoca a la descripció i simulació del circuit serà comú, i ens valdrà per a qualsevol dels entorns. En la part d'implementació estarem ja lligats a un fabricant determinat. Tot i això, tot el que estudiem serà perfectament vàlid per a qualsevol tecnologia i fabricant.

Els exemples que trobareu tot seguit estan fets amb l'eina Modelsim XE III de Mentor Graphics. Tot i això, si feu servir qualsevol altre compilador de VHDL han de funcionar sense problemes. El fet d'utilitzar aquesta eina és simplement pel fet que hi ha una versió gratuïta i la possibilitat d'implementació directa dels dissenys en FPGA, com veurem al proper mòdul.

Vegeu també

Al mòdul "Implementació de sistemes digitals sobre dispositius de tipus FPGA" es tractarà en profunditat el tema de la implementació directa dels dissenys en FPGA.

Dit això, comencem a veure com funciona el llenguatge VHDL. I ho farem de la manera més simple possible. Farem una porta AND en VHDL, cosa que d'alguna manera seria com el "Hello world" que s'utilitza sovint per a introduir qualsevol llenguatge de programació.

2.2. "Hello world": el nostre primer programa VHDL

Començarem el nostre aprenentatge de VHDL amb un exemple molt senzill, que ens permetrà entendre la filosofia del llenguatge. Comencem per descriure mitjançant codi una porta AND. El codi VHDL seria:

```
library ieee;
use ieee.std_logic_1164.all;

-- definició de l'entitat
entity porta_and is
    port(
        in0, in1: in bit;
        out0: out bit
    );
end entity porta_and;

-- definició de l'arquitectura
architecture arquit_and of porta_and is
begin
    out0 <= in0 and in1;
end architecture arquit_and;
```

Si llegiu el codi anterior, veureu que és força intuïtiu. Fixem-nos que tenim:

- Una descripció de les biblioteques que s'utilitzen: aquestes biblioteques ens permeten fer ús dels tipus, operadors, funcions... que podem fer servir en VHDL.

- La definició de l'**entitat** (*entity*), que diu explícitament quins ports té el nostre circuit i de quin tipus són. En el nostre cas, són senzillament dos bits d'entrada i un de sortida.

- La definició de l'**arquitectura** (*architecture*), que indica com està constituït internament el circuit que relaciona entrades i sortides.

Una descripció VHDL d'un circuit constarà d'un conjunt d'entitats, per a cadascuna de les quals caldrà definir una o més arquitectures que la implementen.

És important remarcar que una determinada entitat pot disposar de més d'una arquitectura. És a dir, una determinada entitat es podrà fer de més d'una manera. Finalment, VHDL ens ha de permetre veure les diferències entre fer un circuit amb una arquitectura o una altra. A tall d'exemple: un determinat cir-

cuit combinacional el podem fer amb una determinada funció sense simplificar, o simplificant-la. Per a una mateixa entitat tindrem dues possibles arquitectures, i probablement una serà més òptima que l'altra.

Per descomptat, aquest és el nostre primer exemple, i no hem vist ni tots els tipus de dades, ni tots els operadors que podem fer servir ni les sentències de control de flux. Les anirem veient en els exemples successius. Abans, però comentem alguns aspectes genèrics importants:

a) VHDL no distingeix entre majúscules i minúscules. Per tant, *Entitat_a* i *entitat_a* seran equivalents. Tot i això, i per evitar errors o interpretacions incorrectes, mirarem d'escriure sempre els diferents elements i variables de la mateixa manera.

b) Les normes bàsiques que són vàlides per a altres llenguatges de programació són vàlides també per a VHDL. Així, hem d'intentar sempre de fer codi documentat, clar, senzill, modular i, per descomptat, eficient. En relació amb la documentació, qualsevol línia que comenci amb dos guionets (--) és considerada un comentari.

c) Els identificadors de senyals i variables poden contenir lletres, nombres i _. Sempre han de començar per una lletra i no es permeten dos símbols _ seguits.

d) És un bon costum, quan es treballa amb projectes grans, que els fitxers disposin d'una capçalera explicativa del seu funcionament, on s'indiqui una descripció breu del contingut del fitxer.

Sobre els identificadors VHDL

VHDL permet utilitzar també identificadors estesos amb qualsevol caràcter fent servir seqüències d'escapament (essencialment \ ). Tot i això, ho intentarem evitar per a ser més clars.

Hem vist el codi d'un programa en VHDL. Tot i que molt senzill, ens ha de permetre simular el nostre primer circuit.

2.3. Simular la porta AND

Ben segur, llegint la descripció anterior, molts de vosaltres deveu pensar que VHDL és més aviat un llenguatge purament teòric. Com veurem tot seguit, això no és cert, i podrem veure com evolucionen els senyals dins el nostre circuit.

Malgrat que les diferents eines de treball amb VHDL tenen una filosofia semblant, els exemples a continuació faran servir el simulador Modelsim XE. El que farem essencialment és crear un projecte nou en què inclourem el codi que ja hem creat.

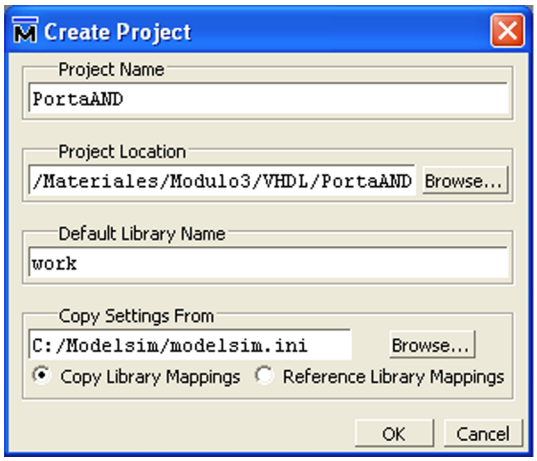

Figura 2. Creació d'un nou projecte de simulació

Tot seguit només haurem d'afegir els fitxers VHDL existents o crear-los de nou. A la figura anterior, i tot i que per ara no és crític, fixem-nos en l'apartat "Default Library Name", que ens permetrà fer referències a objectes, com veurem més endavant. Un cop desats els fitxers els podem compilar. Si tot és correcte, obtindrem un missatge que ens ho indicarà així.

De moment, però, tenim una descripció del circuit, però haurem de crear el banc de proves. Aquest banc de proves ens ha de permetre determinar si el funcionament del circuit és l'esperat.

Aquest cas és molt senzill, i podríem aplicar directament uns determinats valors a les entrades i veure quin és el valor de les sortides. Tot i això, optarem per un enfocament més metodològic, que farem servir tant per als casos senzills com per a altres més complicats. Essencialment, el que farem és separar l'entitat que volem provar de la generació dels vectors de prova.

Analitzem-lo gràficament. El que tenim és una caixa negra, amb dues entrades, in_0 i in_1 i una sortida out_0. El que farem ara és aplicar-li uns senyals d'entrada a aquesta caixa negra i mirar quin és el valor de la sortida. Això ho farem generant un segon fitxer VHDL per al nostre entorn de test.

```
-- fitxer banc_proves.vhd
-- proves per la porta and

library ieee;
use ieee.std_logic_1164.all;

entity banc_proves is
end banc_proves;

architecture arq_banc_proves of banc_proves is

    signal a,b,c: bit;

begin
    -- primer de tot instanciem el circuit
    circuit_test: entity work.porta_and(arquit_and)
        port map(in0=>a,in1 => b, out0 => c);

        process
        begin
            a <= '0';
            b <= '0';
            wait for 200 ns;
            a <='1';
            b <='0';
            wait for 200 ns;
            a <='1';
            b <='1';
            wait for 200 ns;
            a <='0';
            b <='1';

            assert false
                report "Fi de simulacio"

                severity failure;
        end process;
end arq_banc_proves;
```

Abans d'entendre més en detall el codi, expliquem què és el que fem conceptualment. Estem definint una nova entitat sense entrades ni sortides, que és el banc de proves. Aquesta entitat té una arquitectura, que essencialment el que fa és referenciar una caixa negra amb dues entrades i una sortida. A aquesta caixa negra li aplica els senyals *a* i *b* a les entrades i recull la sortida en el senyal *b*.

Nota

Malgrat que en aquest exemple pugui semblar trivial, és important separar el codi del circuit del seu banc de proves. Aporta més claredat, i fa més senzill trobar errors en el disseny.

Gràficament el que fem és:

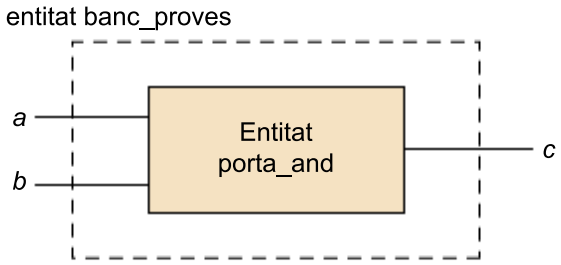

Figura 3. Diagrama de blocs del model VHDL

Fixem-nos que el que fem és crear una supraentitat que aïlla l'entitat que estem analitzant i ens permet centrar-nos únicament en els senyals de proves. Abans de veure el resultat de la simulació, analitzem alguns aspectes importants del codi anterior:

- L'arquitectura del banc de proves la fem a partir d'un altre component: en concret, de l'entitat *porta_and* (podria contenir-ne més d'una), que internament té l'arquitectura *arq_porta_and*. Fixem-nos que s'instancia com *work.porta_and(arq_porta_and)*.

- Els senyals que generem els associem a la disposició de pius de l'entitat *porta_and*. En concret, això ho fem amb el codi *map(in0=>a,in1 => b, out0 => c)*;. A partir d'aquest punt, la constitució interna de la porta ens és irrellevant.

- És fonamental la directiva *process*. Tot i que no ho hem dit fins ara, el codi VHDL és d'execució paral·lela. La directiva *process* ens permet la seqüenciació d'operacions.

El nostre objectiu és simular el disseny. Des de l'entorn mateix de Modelsim disposem de l'opció "Simulate". Hi triem l'opció "Simulate all" i indiquem que volem actuar sobre el nostre banc de proves:

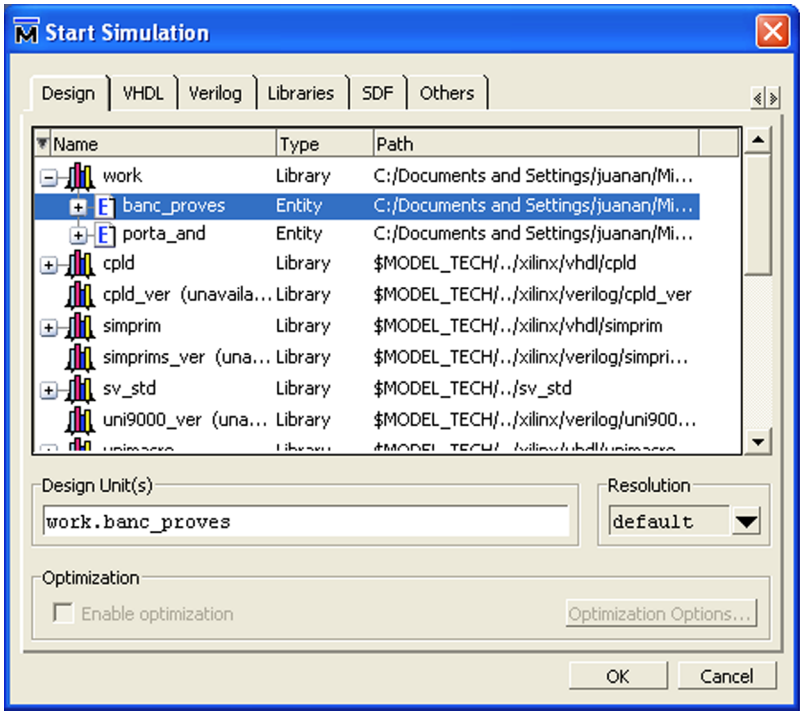

Figura 4. Inici de la simulació

Si ara executem la simulació ("Simulation-Run-All"), obtindrem un gràfic amb les variables que tenim disponibles per visualitzar. En el nostre cas, tenim:

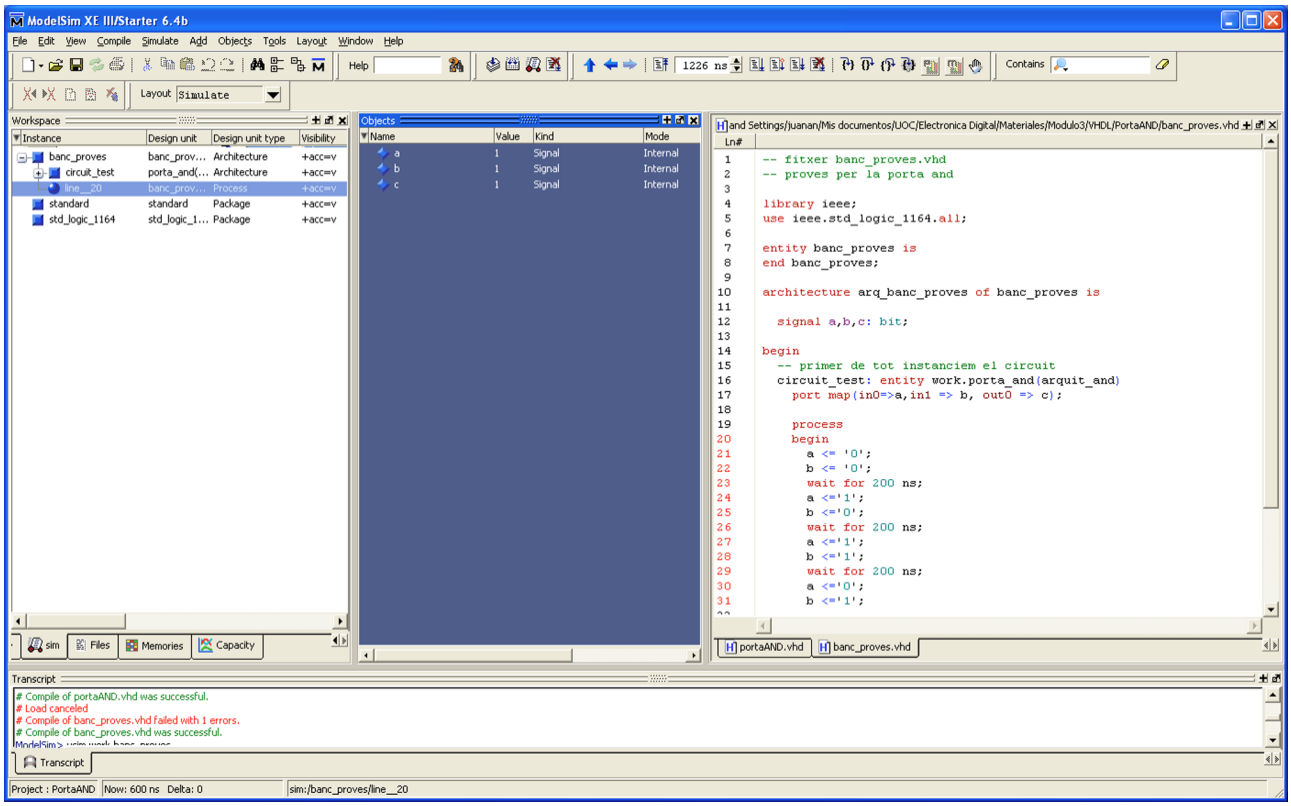

Figura 5. Selecció de variables que s'han de visualitzar

Ara només hem d'afegir les variables per graficar. En el nostre cas, són *a*, *b* i *c*. Des del panell d'objectes, i amb el botó dret, podem afegir-les amb l'opció "Add to wave". Tenim així el resultat de la nostra simulació, que és:

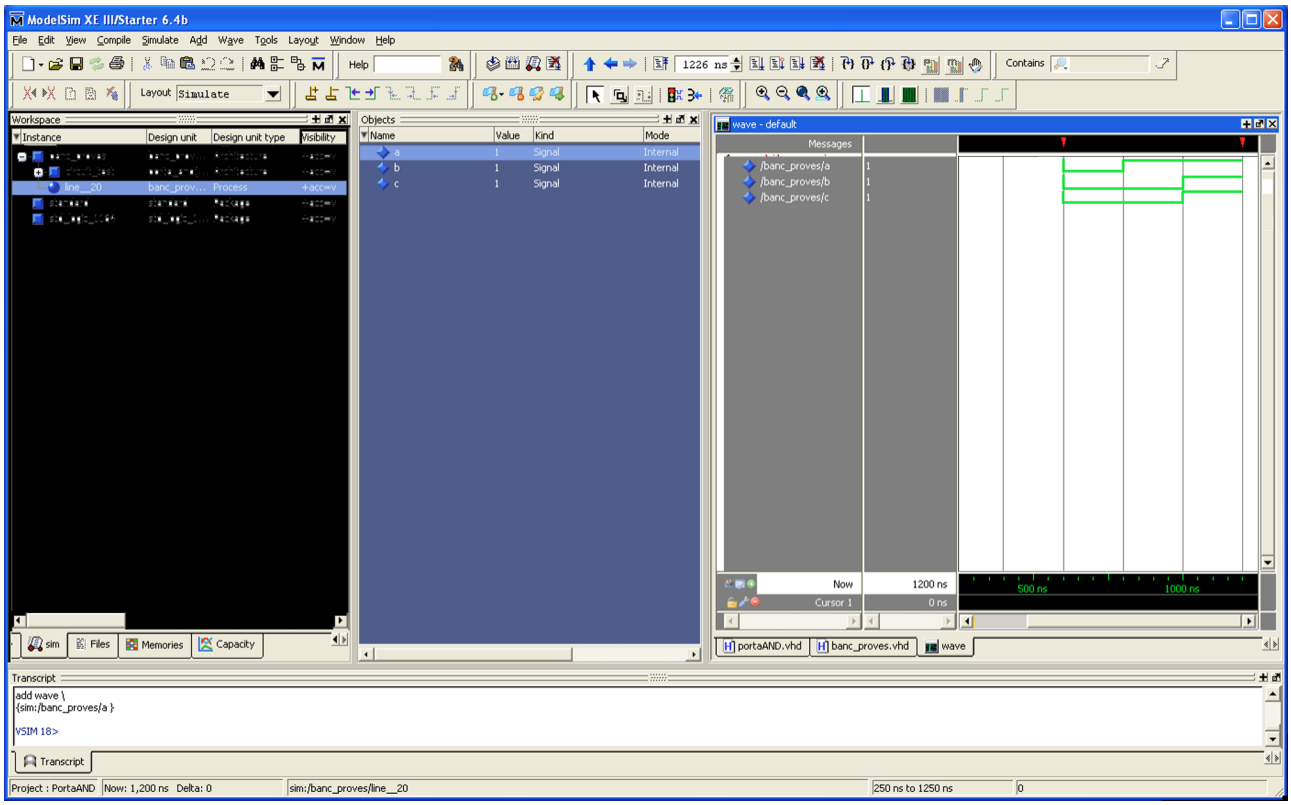

Figura 6. Visualització dels resultats

Com veiem, el resultat és l'esperat. El nostre exemple senzill es comporta realment com una porta AND. Només quan les dues entrades estan a 1 lògic la sortida també ho està. I el que és més important: ja tenim un mètode per a començar a descriure circuits més complexos. Tot seguit ho veurem.

3. Simulació de circuits combinacionals genèrics amb VHDL

Prenent com a base el que hem vist en l'apartat anterior abordarem el disseny de circuits combinacionals. De fet, i per veure un component ja analitzat, prendrem com a model el multiplexor.

Com en el cas anterior, generarem un fitxer per a descriure el circuit, i en farem servir un altre com a banc de proves. Abans, però, incidirem sobre alguns aspectes més que ens seran d'utilitat a mesura que es compliquin els dissenys: els tipus de dades que podem fer servir i les operacions que es permeten. Comencem per veure quins tipus de dades ens permet utilitzar VHDL.

3.1. Tipus de dades habituals i operacions permeses

Els tipus de dades determinen quins valors es poden assignar i quines operacions estan permeses sobre una determinada variable.

VHDL permet la definició de tipus personalitzats. Tot i això, hi ha un conjunt de tipus estàndards que farem servir habitualment:

- *bit*: que presentarà dos possibles valors, 0 o 1.
- *std_logic*: és un valor que utilitzarem sovint, ja que a banda de permetre els valors 0 i 1, permet també els valors 'Z' (alta impedància), U (no inicialitzat) i X (valor desconegut).
- *boolean*: que presenta valors true o false. Com cal suposar, s'utilitzarà per a fer tasques de control de flux en un procés.
- *integer*/*real*: que permeten emmagatzemar valors numèrics, enters o de coma flotant.
- *time*: per a definir unitats de temps. Es pot utilitzar a l'hora de fer bancs de proves.
- *caracter*: per a emmagatzemar valors alfanumèrics.

Addicionalment, podem definir **matrius** de senyals del mateix tipus. Per defecte, VHDL incorpora les matrius de bits (*bit_vector*), *string* (matriu de caràcters) i *std_logic_vector*. La definició es farà de la forma següent:

```
signal bus_exemple : bit_vector (7 downto 0);
```

Amb això definiríem un bus de 8 bits, que numeraríem del 7 al zero.

Ara per ara, aquests tipus seran suficients per a les nostres necessitats.

Implementació física dels tipus

Com veurem, no tots els tipus són implementables directament. Alguns seran d'utilitat únicament per a la nostra simulació.

Molta cura amb les matrius

Podem assignar valors –o matrius– a una altra matriu sempre que siguin del mateix tipus. Cal tenir en compte que en la definició de matrius s'estableix un ordre. Així, no és el mateix definir:

```
signal bus_exemple : bit_vector (7 downto 0);
```

que

```
signal bus_exemple_2 : bit_vector (0 to 7);
```

Penseu –i simuleu– què passaria amb una assignació:

```
bus_exemple_2 <= bus_exemple;
```

Coneguts els tipus amb els quals treballarem, analitzarem ara quines operacions estan permeses:

- *not:* negació. És l'operació més prioritària. Està definida per variables o senyals de tipus *bit*, *bit_vector* o *boolean*.
- *and*, *or*, *nand*, *nor*, *xor*, *xnor*: definides pel mateix tipus que l'operació *not*.
- Assignació: totes les matrius permeten l'assignació amb l'operador <=. Es poden assignar parts d'una matriu mitjançant la definició de rangs –per exemple *a(0)*, en què *a* representa una matriu.
- Comparació: tots els tipus estàndard permeten la comparació amb els operadors clàssics: més petit (<), més petit o igual (<=), igual (=), més gran (>), més gran o igual (>=) o diferent (/=).
- Per al cas de les matrius, quan s'efectuï una comparació, aquesta es du a terme d'esquerra a dreta. Així, una matriu 110 és més gran que 10111 (fixem-nos que no depèn de la longitud de la matriu).
- Enters, reals i temporals permeten operacions aritmètiques (+, -, *, /, ** –potència–, *mod*, *abs*, *rem* –resta de la divisió–).

3.2. Simular el multiplexor

Ja coneixem el funcionament del multiplexor del segon mòdul de l'assignatura. Descrivim-ne tot seguit el comportament en VHDL. Segons hem comentat, seguirem una metodologia en què separem l'arquitectura del banc de proves. Per a l'arquitectura tenim:

```
-- Simulació del  multiplexor
library ieee;
use ieee.std_logic_1164.all;


entity multiplexor is
  port(
  entrada: in std_logic_vector(1 downto 0);
    selector: in std_logic;
    sortida: out std_logic
  );
end multiplexor;


architecture arq_1 of multiplexor is
begin
  sortida <= (entrada(0) and (not(selector))) or ((entrada(1) and
              selector));
end arq_1;
```

I com a banc de proves, podem fer servir aquest:

```
-- Banc de proves per al disseny del multiplexor
library ieee;
use ieee.std_logic_1164.all;

entity test_arq is
end test_arq;

architecture arq_test_arq of test_arq is

  signal a: std_logic_vector(1 downto 0);
  signal b,c: std_logic;

begin
  -- primer de tot instanciem el circuit
  circuit_test: entity work.multiplexor(arq_1)
    port map(entrada=>a,selector => b, sortida => c);

    process
    begin
      a <= "00";
      b <= '0';
      wait for 200 ns;
      a <="01";
      b <='0';
      wait for 200 ns;
      a <="10";
      b <='0';
      wait for 200 ns;
      a <="10";
      b <='1';
      wait for 200 ns;
      a <="11";
      b <='1';
      wait for 2000 ns;

      assert false
        report "Fi de simulacio"

        severity failure;
    end process;
end arq_test_arq;
```

Si mirem de fer la simulació obtenim el resultat següent:

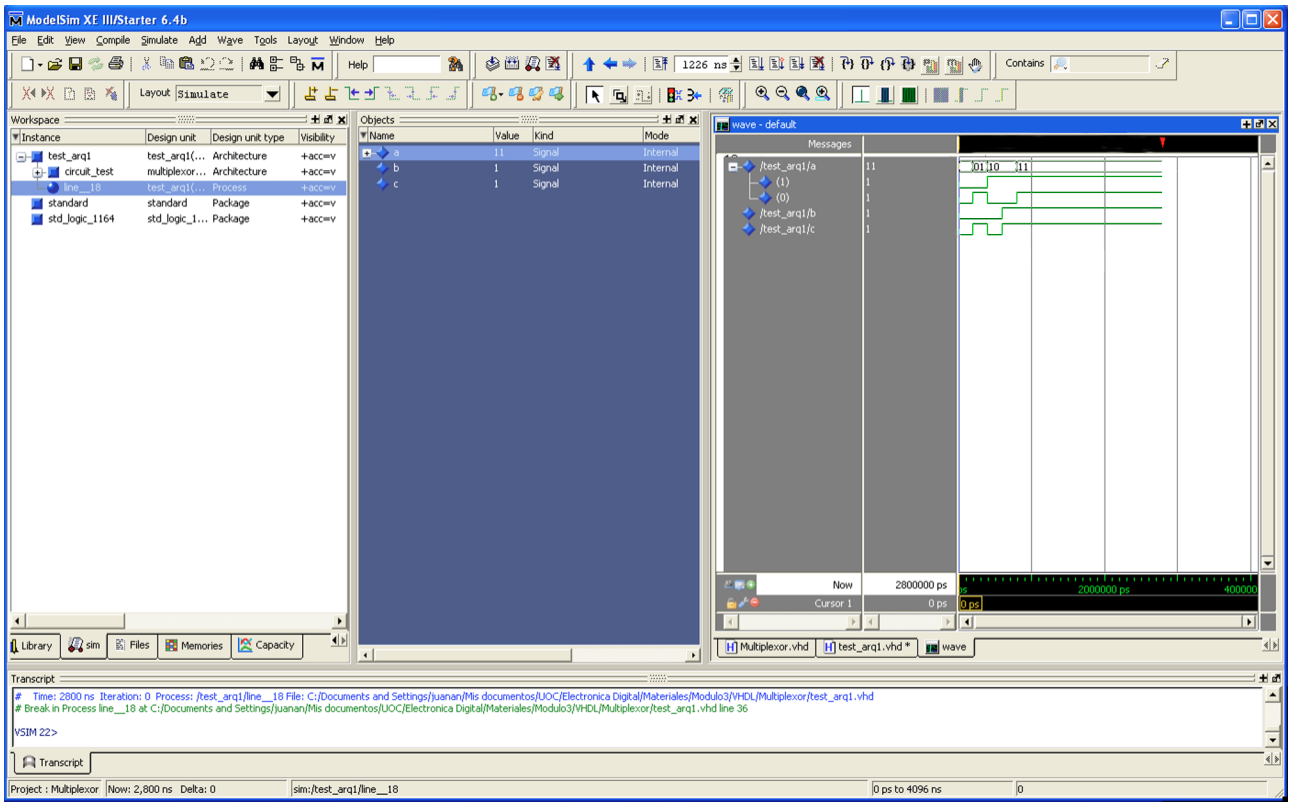

Figura 7. Resultat de la simulació del multiplexor

El resultat de la simulació és l'esperat. Comentem, però, alguns aspectes del codi:

- Com sempre, fixem-nos en la separació entre entitat, arquitectura i banc de proves.

- S'ha utilitzat un format de matriu per a les entrades al multiplexor. Fixeu-vos en la notació *std_logic_vector(1 downto 0)* per al senyal *entrada*, i en la referenciació *entrada(0)*.

- Per a les entrades d'un únic bit s'ha utilitzat el tipus *std_logic*. Podríem haver utilitzat també el tipus *bit*, ja que en aquest cas no és rellevant.

- Hem utilitzat els operadors lògics (*and*, *not*...) sobre les variables *std_logic*, bé directament o bé com a part de les matrius. És important, i aclaridor, marcar l'ordre de les operacions amb parèntesis.

Per al cas concret del multiplexor, VHDL ens dóna una possibilitat afegida, que és l'assignació condicional. Aquesta assignació permet assignar una variable en funció d'una expressió. En concret, la sintaxi seria:

```
senyal <= expressio_1 when (expressio_booleana) else
          expressio_2 when (segona_expressio_booleana)
...
          expressio_n when (n-ésima_expressio_booleana;
```

Activitat

Dissenyeu una arquitectura alternativa per al multiplexor fent servir l'assignació condicional. Quins canvis caldrien per a utilitzar aquesta arquitectura en el banc de proves?

Solució

Per a dissenyar una arquitectura alternativa, només ens cal afegir-la al fitxer on ja disposem de l'anterior:

```
architecture arq_2 of multiplexor is
begin
 sortida <= entrada(0) when selector='0' else entrada(1);
end arq_2;
```

I si ara volem que el nostre banc de proves utilitzi aquesta nova arquitectura, només caldrà canviar en el fitxer del banc de proves la referència a l'arquitectura, que serà:

```
circuit_test: entity work.multiplexor(arq_2)
```

Fixeu-vos que amb un enfocament modular podem descriure i comprovar circuits combinacionals arbitràriament complexos. Addicionalment, la possibilitat de definir diferents arquitectures ens serà molt útil a l'hora de provar diferents dissenys per a una mateixa funció. Estem, doncs, en condicions d'avançar envers definicions intrínsecament més complexes: els circuits seqüencials.

4. Modelatge i simulació de circuits seqüencials

Fins ara ens hem limitat a circuits combinacionals, és a dir, aquells en els quals les sortides en un instant donat són funció únicament de les entrades del circuit en aquell moment. Tot i això, habitualment ens trobarem amb sistemes seqüencials: aquells en els quals la sortida depèn de l'entrada i de l'evolució que ha tingut el sistema.

Per tal d'analitzar-los, començarem per fer algunes consideracions sobre aquests sistemes. A continuació modelitzarem mitjançant VHDL un sistema seqüencial senzill. Amb això estarem en condicions d'abordar el disseny de sistemes seqüencials complexos que, com veurem, tindran una representació funcional fonamentada en una màquina d'estats.

4.1. Consideracions generals sobre els sistemes seqüencials

Un sistema seqüencial serà aquell en el qual la sortida en un instant determinat depèn de les entrades actuals i passades del sistema. És a dir, té en compte l'evolució històrica del sistema.

Aquesta evolució històrica del sistema s'emmagatzema en el que anomenem *variables d'estat* o, simplificant, *estat*. Un estat serà un indicador del punt en el qual es troba el nostre sistema. A manera d'exemple, podem tenir un sistema que, quan es posi en funcionament, vagi a un estat que anomenarem *inicialització*, en el qual s'efectuen un conjunt de tasques i, hipotèticament, es queda a l'espera que succeeixi alguna cosa.

Per posar un exemple més concret, un ordinador és un sistema seqüencial: fixem-nos que quan arrenca, va a un estat inicial –en què inicialitza tot un conjunt de variables– i a partir d'aquell moment va actuant en funció de com evolucionen les entrades –per exemple, el teclat o el ratolí. En aquest cas, evoluciona al ritme que marca un rellotge.

De forma gràfica, podem representar un sistema seqüencial de la manera següent:

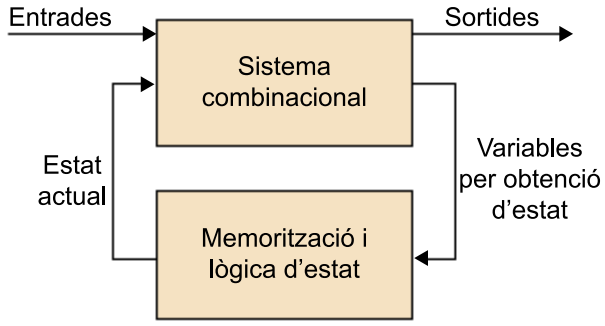

Figura 8. Representació del sistema seqüencial

Una de les entrades al sistema ens permetrà classificar els sistemes seqüencials en dos tipus:

1) Sistemes seqüencials asíncrons: en aquests es fa una anàlisi contínua de les entrades, i la sortida canvia en el moment que ho provoquen les entrades.

2) Sistemes seqüencials síncrons: en aquests l'anàlisi de les variables, i per tant els possibles canvis de sortida, es fan en el moment en què ho indica un senyal específic, el rellotge.

Els sistemes asíncrons, tenen l'avantatge de ser més ràpids de resposta que els síncrons (caldrà esperar un cicle de rellotge per a obtenir la sortida). Addicionalment, consideraran qualsevol canvi que es produeixi en les entrades. En el cas dels sistemes síncrons, una variació molt ràpida d'alguna entrada que es produeixi en un temps més petit que la freqüència del rellotge pot no ser considerada.

Els síncrons, per la seva banda, tenen l'avantatge de més simplicitat en el disseny. D'altra banda, amb els rellotges actuals –velocitat de MHz o GHz–, l'anàlisi que es fa de les entrades és pràcticament contínua.

Amb aquestes consideracions, ens decantarem per l'ús i disseny de sistemes seqüencials síncrons sempre que sigui possible. Cal remarcar, però, que hi pot haver alguns casos en què una problemàtica concreta pugui requerir dissenys asíncrons.

Igual que hem parlat del rellotge (*clock*) com a senyal especial, els sistemes seqüencials acostumen a tenir com a mínim un altre senyal especial. Es tracta del **senyal de *reset***, que porta el sistema a condicions inicials. Malgrat que dissenyem sistemes síncrons, aquest senyal acostuma a tenir una consideració asíncrona, i provoca que el sistema torni a condicions inicials.

4.2. VHDL i disseny de sistemes seqüencials

Al mòdul "Introducció al disseny de sistemes digitals" ja vam veure una introducció a alguns sistemes seqüencials. De fet, ens va servir per a analitzar que el disseny es feia més complex, i que potser ens calien altres eines com el VHDL.

La pregunta que ens fem ara és: disposa el llenguatge VHDL d'algun tipus d'ajuda a l'hora de dissenyar sistemes seqüencials? La resposta, com no podia ser d'una altra manera, és sí. En concret, tindrem ajuda per als punts següents:

a) Detecció de pujada o baixada d'un senyal: òbviament, si volem treballar amb un sistema síncron que funcioni al ritme del rellotge, n'haurem de poder detectar els flancs. De fet, els sistemes síncrons poden operar mitjançant flanc de pujada –les variables s'analitzen quan el rellotge passa de 0 a 1– o de baixada –quan el rellotge passa de 1 a 0.

VHDL permet detectar el flanc amb la paraula reservada *event*. Així, si tenim un senyal anomenat *rellotge*, per a detectar-ne el flanc es farà de la manera següent:

```
if rellotge'event
```

I en concret, podem detectar el flanc de pujada de la manera:

```
if (rellotge'event and rellotge='1')
```

que serà el mode habitual de funcionament dels sistemes síncrons: per flanc de pujada del rellotge.

Nota

El *package standard logic* de VHDL proporciona les funcions *rising_edge (senyal)* i *falling_edge (senyal)*, que també fan la funció de detecció de flancs.

b) Processament de canvis: si indiquem a la directiva *process* un conjunt de senyals, les instruccions que segueixen no s'executaran fins que no hi hagi un canvi d'algun d'aquests senyals. Així, per exemple, en la modelització VHDL de sistemes seqüencials serà habitual una estructura del tipus:

```
process(reset, rellotge)
begin
-- conjunt d'instruccions
end
```

En aquest cas, les directives entre *begin* i *end* s'analitzen quan es produeix un canvi en *reset* o *rellotge*. El cas més habitual és fer que si el senyal de *reset* ha passat de 0 a 1, portarem el sistema a condicions inicials, i en cas contrari analitzarem si el rellotge ha tingut un flanc de pujada. Això en VHDL quedaria com:

Nota

Estem parlant d'un exemple: podríem tenir altres sistemes en els quals el *reset* operi per zero lògic, o bé el rellotge actuï per flanc de baixada. Res no ho impedeix, i tampoc el VHDL.

```
process(reset, rellotge)
begin
    if (reset='1') then
    -- inicialització del sistema
    elseif rellotge='1' and rellotge'event then
    -- lògica del sistema
    end if;
end process;
```

c) Ús de variables i senyals de memorització: una de les coses que caracteritza els sistemes seqüencials és la necessitat de monitorar l'estat. Amb VHDL podem fer ús de variables i senyals i assignar-los valors. A tall d'exemple, si tenim un sistema amb 4 estats, podríem definir a l'hora de definir l'arquitectura d'un sistema el senyal següent:

```
signal estat: integer range 0 to 3;
```

Això ens permetria treballar amb una variable –en aquest cas *integer*– per a memoritzar l'estat. Si volem fer més entenedor el codi, podem definir un nou tipus:

```
type possibles_estats is(EstatInicial,EstatMitja,EstatFinal)
variable estat: possibles_estats;
```

I ara podríem assignar a la variable qualsevol dels valors permesos.

4.3. Descripció de sistemes seqüencials mitjançant màquines d'estats

Suposem un sistema de control d'una porta automàtica (per exemple una porta d'un aparcament). Aquesta porta es trobarà habitualment tancada, i es pot obrir mitjançant una ordre d'activació (que en el nostre exemple pot provenir d'un control remot). En el cas de rebre aquesta ordre, s'ha d'activar un senyal per a obrir la porta.

La porta s'ha de mantenir oberta mentre no rebem una altra ordre. Un sensor ens avisarà que la porta està totalment oberta. A partir d'aquell moment, la porta es pot tancar mitjançant una nova ordre, que en aquest cas indicarà el tancament. És a dir:

- Disposarem de dos sensors que ens identifiquen, respectivament, que la porta està completament oberta o tancada.
- Disposarem d'una entrada de comandament que indica que la porta s'ha d'obrir en cas que estigui tancada o s'ha de tancar en cas que estigui oberta.

El nostre objectiu és generar dos senyals, que controlaran el tancament i l'obertura de la porta, respectivament. Aquests senyals operen sobre un motor, i en el cas que la porta estigui completament oberta o completament tancada, han d'estar desactivats. Gràficament podem veure el sistema com la caixa negra següent:

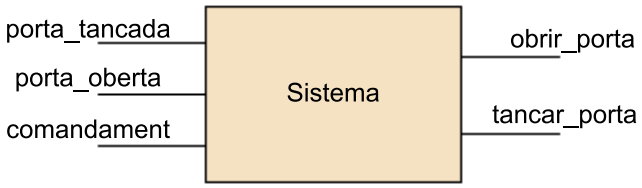

Figura 9. Esquema conceptual del sistema

L'exemple anterior constitueix un exemple de sistema seqüencial (que, a més, suposarem que el dissenyem de manera síncrona i, per tant, farà ús intern d'un rellotge). De fet, el seu comportament el podríem definir gràficament segons es mostra a la figura següent:

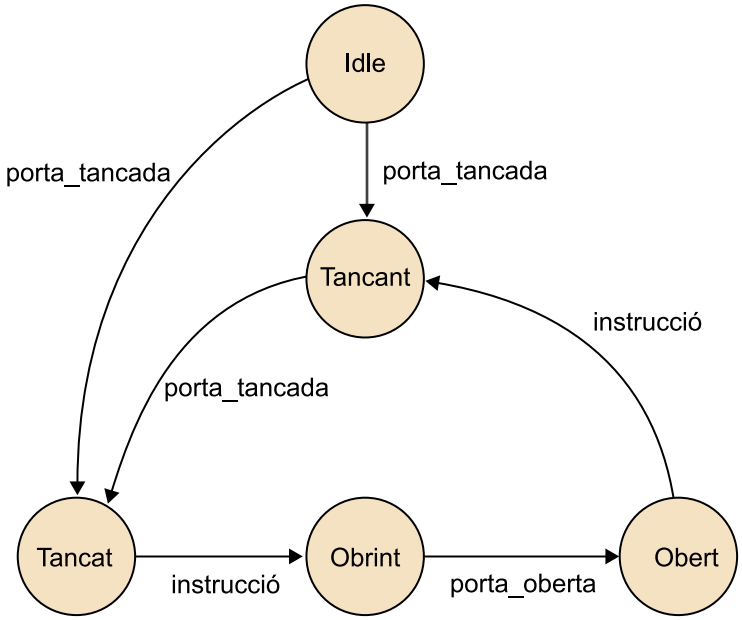

Figura 10. Diagrama d'estats

La figura anterior constitueix un diagrama d'estats del que anomenarem *màquina d'estats finits* (MSF[3]). Aprofitarem aquest exemple concret per a aprendre a dissenyar i modelitzar els sistemes mitjançant VHDL.

(3)Abreujarem *màquina d'estats finits* com a MSF. Sovint la trobarem a la bibliografia com a FSM, de l'anglès *finite state machine*.

Una màquina d'estats finits ens permetrà modelitzar un sistema que transiciona entre un nombre finit de situacions. El diagrama mostrarà els diferents estats i les transicions que provoquen un canvi d'un a un altre.

En el disseny d'aquesta màquina d'estats finits hi haurà dues parts clarament diferenciades:

- La lògica d'estat: que determina, en funció de l'estat actual i les entrades, quin ha de ser l'estat següent.
- La lògica de sortida: que determina a partir de l'estat, i en certs casos de les entrades, quina ha de ser la sortida.

Per evitar errors de disseny, separarem sempre la lògica d'estat de la lògica de sortida.

Si ens fixem en les parts de la MSF, veiem que el valor de la sortida depèn de l'estat i, en certs casos, també directament de les entrades. Això dóna lloc a les dues representacions habituals de les MSF:

- Representació de Mealy: la sortida depèn de l'estat i de les entrades.
- Representació de Moore: la sortida depèn únicament del valor de l'estat.

Gràficament, les dues representacions es mostren a la figura següent:

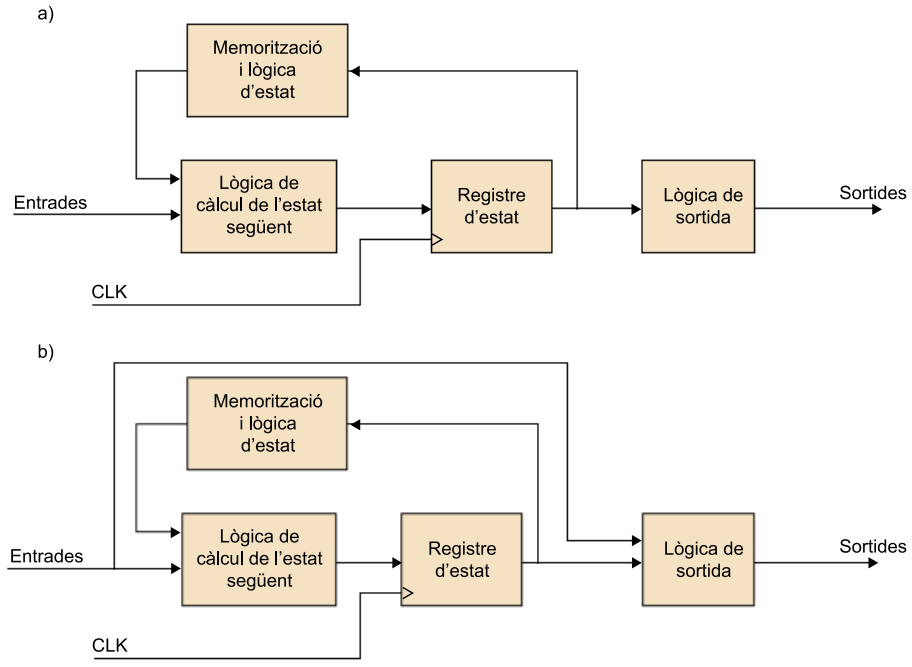

Figura 11. Esquemes de l'autòmat de Mealy (a) i de Moore (b)

No és el nostre objectiu entrar en les diferències conceptuals entre tots dos models. De fet, i sense entrar en detall, podem veure que la representació de Mealy és més genèrica que la de Moore.

Tornant al nostre problema, la nostra MSF defineix què fa el sistema, però no com ho fa. Anem a veure-ho tot seguit. Fixem-nos que en les representacions de les MSF hem definit un registre d'estat que permet memoritzar l'estat.

Abans d'entrar en la codificació, tinguem present la representació de la figura de Mealy –o de Moore– en què definirem els senyals següents abans i després del registre d'estat:

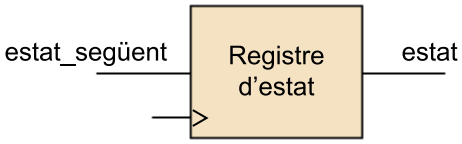

Figura 12. Esquema del registre d'estat

Abans de continuar tinguem en ment:

- La MSF del sistema que volem dissenyar (figura 10).
- L'esquema de Mealy/Moore.
- El nom dels senyals que farem servir per a memoritzar l'estat.
- I el fet que sempre separarem la lògica d'estat de la lògica de sortida.

Comencem, doncs, per dissenyar la lògica d'estat. Si tenim clars els punts que acabem d'anomenar, i el que hem vist en aquest mòdul, el codi següent hauria de ser senzill d'entendre. En primer lloc, modelitzem la màquina d'estats:

```
process (clk, reset)
begin
    if (reset='1') then
        estat <= EstatInicial;
    elsif (clk'event and clk='1') then
        case estat is
            when EstatInicial =>
                if (porta_tancada='0') then estat <= Tancant;
                else estat <= Tancat;
                end if;

            when Tancant =>
                if (porta_tancada='1') then estat <= Tancat;
                end if;
            when Tancat =>
                if (comandament='1') then estat <= Obrint;
                end if;
            when Obrint =>
                if (porta_oberta='1') then estat <= Obert;
                end if;
            when Obert  =>
                if (comandament='1') then estat <= Tancant;
                end if;
        end case;
    end if;
end process;
```

Com a segona part, podem determinar per a cada estat les seves sortides associades:

```
process (estat)
begin
case estat is
    when EstatInicial => obrir_porta<='0';
            tancar_porta<='0';

    when Tancant    =>    tancar_porta<='1';
    when Tancat     =>    tancar_porta<='0';
    when Obrint     =>    obrir_porta<='1';
    when Obert      =>    obrir_porta<='0';

end case;
end process;
```

5. Problemes resolts

Per a complementar el coneixement de VHDL que hem vist en aquest mòdul simularem un sistema combinacional i un sistema seqüencial que ja coneixem, i en comprovarem el funcionament. En concret, resoldrem dos dels problemes plantejats al mòdul "Introducció al disseny de sistemes digitals".

Vegeu també

Podeu trobar aquests dos problemes a l'apartat 5 del mòdul "Introducció al disseny de sistemes digitals".

5.1. Problema 1

Enunciat

Dissenyeu i simuleu utilitzant VHDL un comparador d'un bit, de manera que es pugui utilitzar per a implementar comparadors d'un nombre de n bits. Amb aquest mòdul, construïu un comparador de 4 bits.

Solució

Aprofitarem el disseny d'aquest sistema combinacional per a treballar l'estructura modular i fer crides a mòduls ja dissenyats.

Recordeu que al mòdul "Introducció al disseny de sistemes digitals" hem resolt el problema en dos passos:

a) Creació d'un comparador universal d'un bit, mòdul de cinc entrades i tres sortides. Les entrades corresponen als valors de comparació dels mòduls previs ($a > b$, $a < b$, $a = b$) i als dos bits que volem comparar. D'altra banda, en la sortida tindrem la indicació del resultat de la comparació.

b) Creació del comparador de n bits (en el nostre cas, $n = 4$) partint del comparador d'un bit.

Comencem per la realització del primer punt. En aquest cas, farem servir directament les funcions lògiques que vam trobar al problema del mòdul "Introducció al disseny de sistemes digitals". Podem, per tant, implementar el comparador amb el codi VHDL següent:

```
-- definició d'entitat
entity comparador_un_bit is
    port(
        a_gt_b,a_lt_b,a_eq_b,a,b: in bit;
        a_gt_b_out,a_lt_b_out,a_eq_b_out: out bit
    );
end entity comparador_un_bit;

-- definició arquitectura
architecture arquit_comparador_u of comparador_un_bit is
begin

    a_gt_b_out <= a_gt_b OR (a_eq_b AND (a AND (NOT b)));
    a_lt_b_out <= a_lt_b OR (a_eq_b AND (NOT a AND b));
    a_eq_b_out <= a_eq_b AND (( (not a) and (not b)) OR (a AND b));

end architecture arquit_comparador_u;
```

Podem veure la definició de l'entitat i de la seva arquitectura. A l'hora d'implementar l'arquitectura hem fet servir directament les funcions lògiques que ja havíem trobat.

Manca ara implementar el comparador de 4 bits prenent com a base l'estructura ja construïda. En concret, el que hem de fer és encadenar els comparadors. Recordem que l'estructura que cal implementar és la de la figura 35 del mòdul "Introducció al disseny de sistemes digitals", que recordem tot seguit:

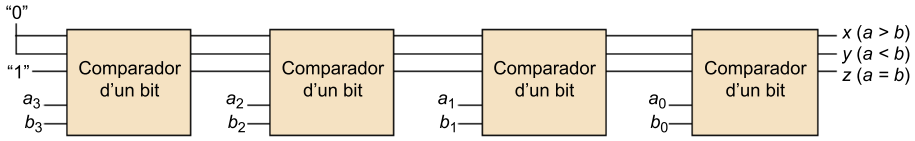

Figura 13. Construcció del comparador de 4 bits a partir de comparadors d'un bit

Anem a veure tot seguit com VHDL ens permet fer aquest disseny modular. Abans de res, i per fer més entenedor el codi, donarem noms als senyals interns que interconnecten els diferents mòduls comparadors d'un bit dins el comparador de 4 bits.

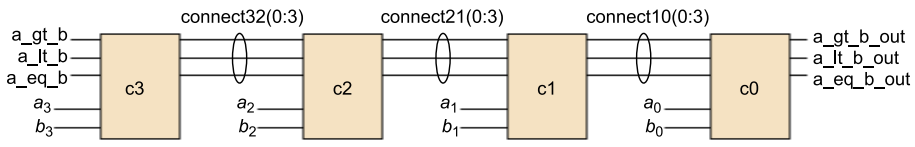

Figura 14. Comparador de 4 bits amb marcatge de les connexions internes

En primer lloc definirem, doncs, l'entitat:

```
entity comparador_multi_bit is
    port(
        a_gt_b,a_lt_b,a_eq_b: in bit;
        a,b: in bit_vector(3 downto 0);
        a_gt_b_out,a_lt_b_out,a_eq_b_out: out bit
    );
end entity comparador_multi_bit;
```

Aquest codi és semblant al que ja havíem vist. L'únic que estem definint és un comparador de 4 bits, amb 3 ports d'entrada que, com hem vist, serveixen per a encadenar comparadors (*a_gt_b*,*a_lt_b*,*a_eq_b*), els dos valors que volem comparar, que ara ja són de 4 bits (*a* i *b* definits com a vectors) i, finalment, els senyals de sortida, que ens indicaran quin dels valors és més gran.

Ens caldrà ara definir com està constituït aquest comparador internament, és a dir, la seva arquitectura. A diferència dels exemples vistos fins ara, no descriurem a escala de porta el que fem, sinó que aprofitarem un mòdul que ja hem definit (el del comparador d'un bit). Exposem el codi i tot seguit l'expliquem:

```
architecture modular of comparador_multi_bit is

  signal connect32 ,connect21 ,connect10: bit_vector(2 downto 0); -- senyals internes

  component comparador_un_bit  -- utilitzarem aquest bloc com a component
      port (a_gt_b,a_lt_b,a_eq_b,a,b: in bit;
      a_gt_b_out, a_lt_b_out,a_eq_b_out: out bit);
  end component;

begin

  -- definim 4 comparadors encadenats
  c3: comparador_un_bit port map (a_gt_b,a_lt_b,a_eq_b,a(3),b(3),connect32(2),connect32(1),
  connect32(0));
  c2: comparador_un_bit port map (connect32(2),connect32(1),connect32(0),a(2),b(2),connect21(2),
  connect21(1),connect21(0));
  c1: comparador_un_bit port map (connect21(2),connect21(1),connect21(0),a(1),b(1),connect10(2),
  connect10(1),connect10(0));
  c0: comparador_un_bit port map (connect10(2),connect10(1),connect10(0),a(0),b(0),a_gt_b_out,
  a_lt_b_out,a_eq_b_out);


end modular;
```

Expliquem tot seguit el codi anterior, ja que ens serà molt útil a l'hora de fer dissenys modulars. Com veiem, hi ha parts rellevants:

- Definició de senyals interns al mòdul.
- Definició dels components.
- Ús dels components del disseny.

Fixem-nos que el que fem és utilitzar un conjunt de peces i interconnectar-les. Mirem de nou la figura 14. Què hi trobem? En primer lloc, veiem un conjunt de blocs interconnectats per un conjunt de senyals. Són aquests senyals els que definim inicialment dins l'arquitectura.

```
signal connect32 ,connect21 ,connect10: bit_vector(2 downto 0);
 -- senyals interns
```

Aquests senyals s'utilitzen per a interconnectar un conjunt de "caixes negres". Com a regla mnemotècnica hem anomenat els senyals *connectxy*, en què *x* és el mòdul del qual provenen i *y* el mòdul de destinació. És a dir, el senyal *connect32* connecta la "caixa negra" 3 amb la "caixa negra" 2.

En el nostre cas, aquestes caixes negres són els comparadors d'un bit. Definir-los és tan simple com invocar la peça de codi que ja hem programat anteriorment:

```
component comparador_un_bit -- utilitzarem aquest bloc com a
                               component
    port (a_gt_b,a_lt_b,a_eq_b,a,b: in bit;
    a_gt_b_out, a_lt_b_out,a_eq_b_out: out bit);
end component;
```

Faltarà finalment reflectir com es connecten. Agafem a tall d'exemple el mòdul *c1*, que rep com a entrades els bits del senyal *connect* que interconnecta el bloc 2 i el 1 (*connect21*), juntament amb els bits *a(1)* i *b(1)*. Obtindrem com a sortida els senyals que connecten el mòdul *c1* i el *c0* (que hem anomenat *connect10*):

```
c1: comparador_un_bit port map
(connect21(2),connect21(1),connect21(0),a(1),b(1),connect10(2),
 connect10(1),connect10(0));
```

En aquest punt, i per tenir la visió global, podem veure com ha quedat en conjunt el nostre comparador, incloent-hi tant el comparador d'un bit com el generat modularment:

```
library ieee;
use ieee.std_logic_1164.all;


-- definició d'entitat
entity comparador_un_bit is
    port(
        a_gt_b,a_lt_b,a_eq_b,a,b: in bit;
        a_gt_b_out,a_lt_b_out,a_eq_b_out: out bit
    );
end entity comparador_un_bit;


-- definició d'arquitectura
architecture arquit_comparador_u of comparador_un_bit is
```

```
begin

    a_gt_b_out <= a_gt_b OR (a_eq_b AND (a AND (NOT b)));
    a_lt_b_out <= a_lt_b OR (a_eq_b AND (NOT a AND b));
    a_eq_b_out <= a_eq_b AND (( (not a) and (not b)) OR (a AND b));

end architecture arquit_comparador_u;


entity comparador_multi_bit is
    port(
        a_gt_b,a_lt_b,a_eq_b: in bit;
        a,b: in bit_vector(3 downto 0);
        a_gt_b_out,a_lt_b_out,a_eq_b_out: out bit
    );
end entity comparador_multi_bit;

architecture modular of comparador_multi_bit is

  signal connect32 ,connect21 ,connect10: bit_vector(2 downto 0); -- senyals interns

  component comparador_un_bit  -- utilitzarem aquest bloc com a component
      port (a_gt_b,a_lt_b,a_eq_b,a,b: in bit;
      a_gt_b_out, a_lt_b_out,a_eq_b_out: out bit);
  end component;

begin

  -- definim 4 comparadors encadenats
  c3: comparador_un_bit port map (a_gt_b,a_lt_b,a_eq_b,a(3),b(3),connect32(2),connect32(1),
  connect32(0));
  c2: comparador_un_bit port map (connect32(2),connect32(1),connect32(0),a(2),b(2),connect21(2),
  connect21(1),connect21(0));
  c1: comparador_un_bit port map (connect21(2),connect21(1),connect21(0),a(1),b(1),connect10(2),
  connect10(1),connect10(0));
  c0: comparador_un_bit port map (connect10(2),connect10(1),connect10(0),a(0),b(0),a_gt_b_out,
  a_lt_b_out,a_eq_b_out);


end modular;
```

Com a punt final, podem fer la simulació del nostre circuit. En aquest cas, hem utilitzat un banc de proves com el que es planteja tot seguit:

```
-- fitxer banc_proves.vhd
-- proves pel comparador multibit
```

```
library ieee;
use ieee.std_logic_1164.all;

entity banc_proves is
end banc_proves;



architecture arq_proves of banc_proves is

  signal major, menor, igual: bit;
  signal a,b: bit_vector(3 downto 0);
  signal major_out,menor_out, igual_out: bit;

begin
  -- primer de tot instanciem el circuit
  circuit_test: entity work.comparador_multi_bit(modular)
  port map(a_gt_b=>major,a_lt_b=>menor, a_eq_b=>igual,a => a,b =>b, a_gt_b_out => major_out,
  a_lt_b_out => menor_out,a_eq_b_out => igual_out);

    process
    begin
      major <= '0';
      menor<='0';
      igual<='1';

      a <="0100";
      b <="0100";
      wait for 200 ns;
      a <="0100";
      b <="0011";
      wait for 200 ns;
      a <="0011";
      b <="1000";
      wait for 200 ns;
      a <="0111";
      b <="0111";

      assert false
        report "Fi de simulacio"
        severity failure;
    end process;
end arq_proves;
```

Aquest codi ens mostra com el comparador opera com esperem:

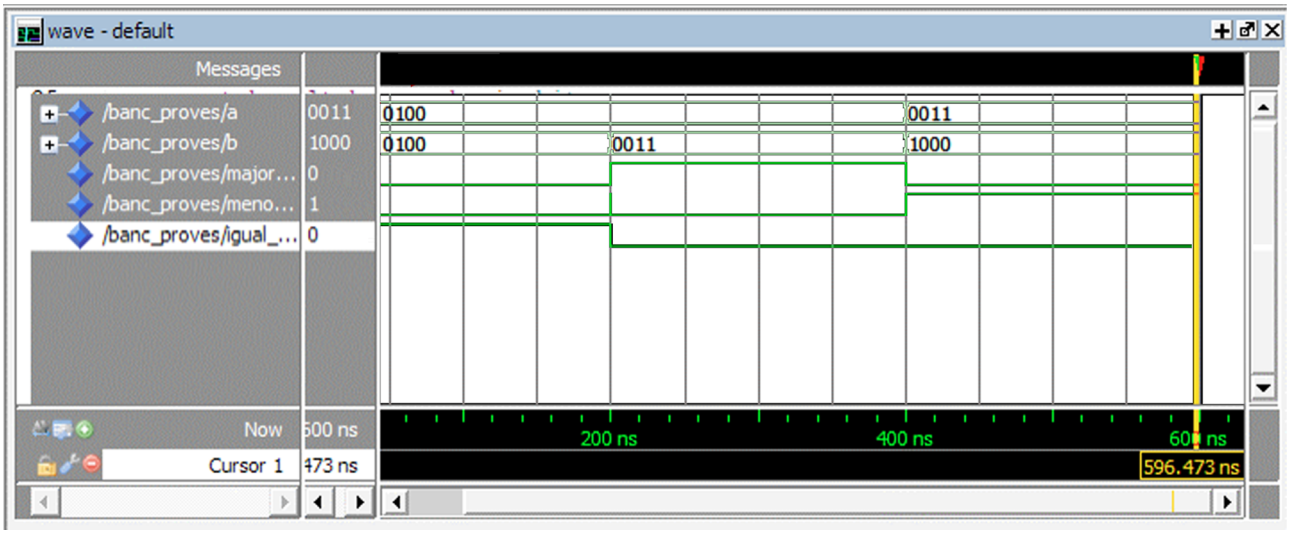

Figura 15. Resultat de la simulació

Amb la qual cosa el problema queda resolt.

5.2. Problema 2

Enunciat

Volem dissenyar un sistema de seguretat que en cas d'alarma enviï un codi a una central. El sistema en qüestió disposa de tres entrades –dues entrades de sensors (I_1, I_2), un polsador (I_3) i un senyal per a inicialitzar el sistema (RST)– i dues sortides, que anomenarem ALARM i LINE.

La sortida ALARM és un senyal indicador d'estat d'alarma. En estat de repòs –quan el sistema s'inicialitza– pren el valor 0. S'activa quan es produeixen 8 flancs de pujada del sensor I_1, seguits de 4 flancs –bé sigui de pujada o baixada– del sensor I_2.

En cas que la sortida ALARM prengui el valor 1, es considera un senyal d'alarma. En aquest cas, LINE, que connecta amb la central d'alarmes, transmet en sèrie a una velocitat de 100 Kbps un codi de manera cíclica (101010...).

L'activació del polsador (I_3, actiu per 1) durant 10 µs desactiva la condició d'alarma i torna el sistema al seu estat de repòs.

Simuleu en VHDL el sistema plantejat i comproveu que funcioni correctament. Considereu que disposeu d'un únic rellotge d'1 MHz.

Nota

Per tal d'incloure el resultat de la simulació en una única gràfica de manera que els resultats siguin visibles, hem considerat que I_3 està actiu únicament 10 µs per a provocar un *reset* (en lloc dels 4 s que havíem proposat al problema del mòdul "Introducció al disseny de sistemes digitals"). Com veurem, no afecta en absolut el disseny, i és únicament per millorar la visualització gràfica del resultat.

Solució

Començarem el problema amb el disseny modular al qual vam arribar en el seu moment (figura 40 del mòdul "Introducció al disseny de sistemes digitals"):

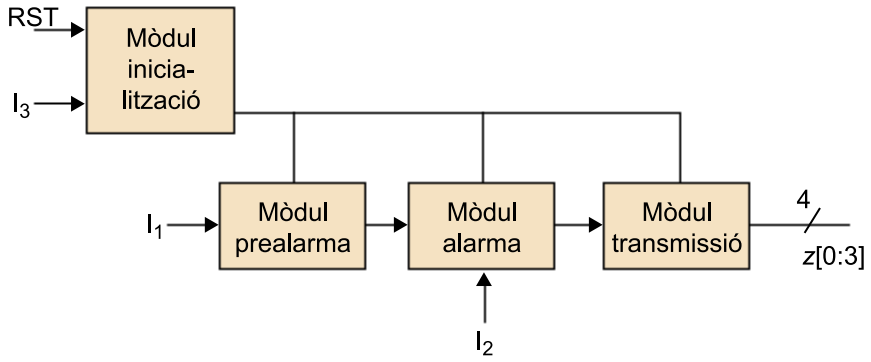

Figura 16. Disseny modular del problema 2

Tal com hem fet en el problema anterior, farem una aproximació modular. Tot i això, simularem aquest cas fonamentant-nos en el comportament dels diferents mòduls en lloc d'una translació directa dels components electrònics als quals vam arribar en el seu moment. És a dir, mirarem de definir de nou els mòduls, no des del punt de vista de components sinó de comportament.

D'entrada, definim el nostre circuit, exposant únicament les entrades i sortides que presenta. Això en VHDL serà:

```
library ieee;
use ieee.std_logic_1164.all;

-- definició d'entitat

entity AlarmaModul3 is
       port(

           i1,i2,i3: in std_logic;
           clk: in std_logic;
           reset: in std_logic;
           alarm,line: out std_logic
       );
end AlarmaModul3;
```

Definim així la nostra entitat, que, com veiem, té cinc entrades (les línies i_1, i_2 i i_3, a banda del rellotge i el senyal de *reset* global) i dues sortides (l'indicador d'alarma i la línia de transmissió). Ens cal ara una definició modular de la nostra entitat:

```
architecture modular of AlarmaModul3 is


  signal pciglobal: std_logic; -- senyals interns
  signal connex_prea: std_logic;
```

```
    signal connex_alarm: std_logic;

    component modul_init
        port (reset,i3,clk: in std_logic;
        pcisist: out std_logic);
    end component;


    component modul_prealarma
        port (pcisist,i1: in std_logic;
        prea: out std_logic);
    end component;


    component modul_alarma
        port (pcisist,prea,i2: in std_logic;
        alarma: out std_logic);
    end component;


    component modul_transmit
        port (pcisist,alarma2,clk: in std_logic;
        z: out std_logic);
    end component;

  begin

    -- definim 4 mòduls
    modul_1: modul_init port map (reset,i3,clk,pciglobal);
    modul_2: modul_prealarma port map (pciglobal,i1,connex_prea);
    modul_3: modul_alarma port map (pciglobal,connex_prea,i2,connex_alarm);
    modul_4: modul_transmit port map (pciglobal,connex_alarm,clk,line);

    --
    -- falta per generar la sortida alarm, que la traiem directament del modul_3
    alarm <=connex_alarm;


  end modular;
```

Fixem-nos com el que fem és definir un total de 4 components, que responen als blocs de la figura 16. Remarquem únicament que el senyal d'alarma s'utilitza tant com a sortida com per a ser utilitzat com a entrada en el quart mòdul. Per aquest motiu hem generat un senyal intern, que permetrà aquesta doble connexió.

Estem, doncs, en condició d'abordar cadascun dels mòduls del sistema:

a) Mòdul d'inicialització

Permet inicialitzar el sistema, bé sigui per activació del senyal de *reset*, o a causa de la presència del senyal i_3 actiu durant un nombre de cicles de rellotge (en el nostre cas, 10 cicles). Podem implementar aquest comportament en VHDL amb l'estructura següent:

```
library ieee;
use ieee.std_logic_1164.all;

entity modul_init is
    port(
   reset, i3,clk: in std_logic;
  pcisist: out std_logic
);
end modul_init;



architecture interna of modul_init is

    begin
    process(clk)

    variable num_ticks: integer;
    variable resetejant: integer;
    variable num_cicles_reset: integer;
    constant max_ticks:integer :=10;

    begin
      if (clk'event and clk='1') then
        if (i3='1') then
          num_ticks:=num_ticks+1;
        else
          num_ticks:=0;
        end if;
    end if;

    if (num_ticks=max_ticks or reset='1') then
      pcisist <='1';
    else
      pcisist <='0';
  end if;


  end process;

end interna;
```

La variable *max_ticks* és l'encarregada de detectar quants cicles de rellotge fa que està actiu el senyal i_3. Si arribem al nombre màxim, el senyal permet l'activació del *reset* del sistema, cosa que tornarà tant la línia com el senyal d'alarma a l'estat de repòs.

b) Mòdul de prealarma

És encarregat simplement de detectar els flancs de pujada del senyal i_1. La seva implementació en VHDL serà:

```
library ieee;
use ieee.std_logic_1164.all;

entity modul_prealarma is
  port(
  pcisist, i1: in std_logic;
  prea : out std_logic
);
end entity modul_prealarma;


architecture interna of modul_prealarma is

  begin
    process(pcisist,i1)


    variable num_pujades: integer;
    constant max_pujades:integer :=4;

    begin

      if (pcisist='1') then
        prea<='0';
        num_pujades :=0;
      elsif (i1'event and i1='1') then
           if (num_pujades < max_pujades) then
             num_pujades:=num_pujades+1;
           end if;
           if (num_pujades = max_pujades) then
               prea<='1';
          end if;
      end if;

end process;
end interna;
```

Fixem-nos com el que fa el sistema és, d'una banda, atendre al possible *reset* global que indica el primer dels mòduls ja analitzats, i d'altra, comptar les pujades del senyal i_1.

c) Mòdul d'alarma

Si un cop s'han produït els flancs de pujada del senyal i_1 se'n produeixen 4 del senyal i_2 (en aquest cas de pujada o de baixada) caldrà activar una alarma. Fixem-nos també que només s'activa en el cas que estem en situació de prealarma (és a dir, ja s'han produït les fluctuacions de i_1).

```
library ieee;
use ieee.std_logic_1164.all;

entity modul_alarma is
  port(
    pcisist, prea, i2: in std_logic;
    alarma: out std_logic
);
end entity modul_alarma;


architecture interna of modul_alarma is

begin
    process(pcisist,i2)

    variable num_pujades: integer;
    constant max_pujades:integer :=4;

    begin

      if (pcisist='1') then
        alarma<='0';
        num_pujades :=0;
      elsif (i2'event and prea='1') then
           if (num_pujades < max_pujades) then
             num_pujades:=num_pujades+1;
           end if;
           if (num_pujades = max_pujades) then
              alarma<='1';
          end if;
      end if;

end process;
end interna;
```

d) Mòdul de transmissió

Efectua una transmissió a una freqüència d'1/10, la del rellotge del sistema, en cas que hi hagi alarma. Això ho podem implementar amb el codi següent:

```
entity modul_transmit is
    port(
      pcisist,alarma2,clk: in std_logic;
      z : out std_logic
);
end entity modul_transmit;



architecture interna of modul_transmit is
   signal sortida: std_logic;
begin
    process(clk)

    variable num_cicles: integer;
    variable enviant : integer;
    variable valor_a_linia: integer;
    constant clk_divider:integer :=5;

    begin

      if (pcisist='1') then
        num_cicles:=0;
        valor_a_linia :=0;
      end if;


        if (clk'event and clk='1') then
          if (num_cicles=clk_divider) then
            num_cicles:=0;
            if (valor_a_linia =0) then
              valor_a_linia:=1;
            else
              valor_a_linia:=0;
            end if;
          else
            num_cicles:=num_cicles+1;
          end if;
        end if;

        if (valor_a_linia =1) then
          z <= alarma2;
        else
          z <='0';
        end if;

end process;
end interna;
```

Ens queda finalment fer la simulació. En el nostre cas, hem utilitzat aquest banc de proves per a validar el comportament del circuit:

```
-- fitxer banc_proves.vhd
-- proves pel comparador multibit

library ieee;
use ieee.std_logic_1164.all;

entity banc_proves is
end banc_proves;


architecture arq_proves of banc_proves is

  signal i1,i2,i3: std_logic;
  signal clk: std_logic;
  signal reset: std_logic;
  signal alarm,line: std_logic;

begin
  -- primer de tot instanciem el circuit
  circuit_test: entity work.AlarmaModul3(modular)
    port map(i1,i2,i3,clk,reset, alarm, line);

    process
    begin
      i1 <= '0';
      i2<='0';
      i3 <='0';
      reset <='1';

      wait for 50 us;
      reset <='0';
      i1 <='1';
      i2 <='0';
      wait for 20 us;
      i1 <='0';
      wait for 50 us;
      i1 <='1';
      wait for 20 us;
      i1 <='0';
      wait for 50 us;
      i1 <='1';
      wait for 20 us;
      i1 <='0';
      wait for 50 us;
      i1 <='1';
      wait for 20 us;
      i1 <='0';
      wait for 50 us;
      i1 <='1';
      wait for 20 us;
      i1 <='0';
      wait for 50 us;
      i1 <='1';
      wait for 20 us;
      i1 <='0';
      wait for 50 us;
      i1 <='1';
      wait for 20 us;
      i1 <='0';
      wait for 50 us;
      i1 <='1';

      wait for 20 us;
      i2 <='0';
      wait for 50 us;
      i2 <='1';
      wait for 20 us;
      i2 <='0';
      wait for 50 us;
      i2 <='1';
      wait for 20 us;
      i2 <='0';
```

```
        wait for 50 us;
        i2 <='1';
        wait for 20 us;
        i2 <='0';
        wait for 50 us;
        i2 <='1';

        wait for 100 us;
        i3 <='1';
        wait for 20 us;
        i3 <= '0';

        assert false
          report "Fi de simulacio"
          severity failure;
      end process;
end arq_proves;
```

Amb això podem veure el resultat gràfic de la simulació, que és el de la figura següent, que, com veiem, respon als requisits del problema.

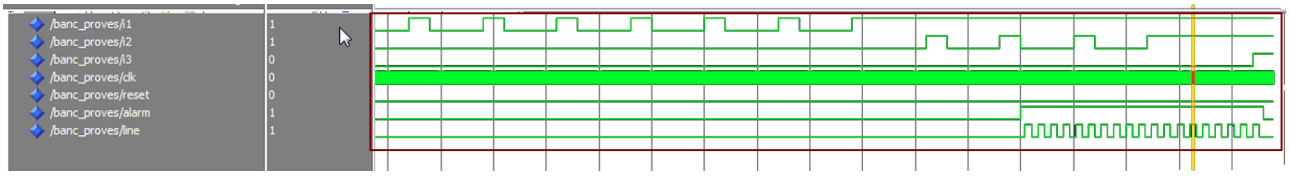

Figura 17. Resultat de la simulació

6. Del disseny VHDL a la síntesi: passos següents

En aquest mòdul hem avançat més en el nostre coneixement dels circuits digitals. Hem après a fer la simulació dels nostres dissenys, cosa que ens permet validar-los i, en cas de detectar errades, corregir-les abans d'enviar el nostre disseny a fabricació.

Si girem la vista enrere en l'assignatura, com més va ens allunyem més del disseny de circuits de transistors i capes de silici. Malgrat que finalment seran peces imprescindibles, ens anem abstraient com més va més d'aquest "baix nivell" de l'electrònica.

De tota manera, ben segur que molts de vosaltres us preguntareu si no ens estem allunyant massa del món tangible. Hem definit el comportament de circuits, però finalment no l'hem muntat. Com passem aquest disseny VHDL a un circuit real i tangible que funcioni? Doncs bé: VHDL ens donarà eines per a sintetitzar aquests dissenys, tal com veurem en el mòdul "Implementació de sistemes digitals sobre dispositius de tipus FPGA".

Vegeu també

En el mòdul "Implementació de sistemes digitals sobre dispositius de tipus FPGA" trobareu les eines del VHDL per a sintetitzar els dissenys de circuits.

Abans de passar de mòdul, cal que fem una reflexió sobre el que ens permet el llenguatge VHDL: fixem-nos que tenim una via per a simular sense fabricar. No cal dir que simular sempre és més econòmic, en temps i diners, que el muntatge de maquinari. No menyspreem, doncs, aquesta potència, i donem per ben emprat el temps que emprem a simular els nostres circuits.

Què hem perdut pel camí? En reflexionarem també al mòdul "Implementació de sistemes digitals sobre dispositius de tipus FPGA", però com més ens allunyem del "baix nivell" –entenent com a tal el silici, els transistors o les portes lògiques– més ens allunyarem de dissenys "òptims", entenent com a tal aquells que fan servir el nombre mínim de transistors –o de portes. De tota manera, l'electrònica ha guanyat tal capacitat d'integració, que fa que possiblement això no sigui un problema, o com a mínim no tan greu com podem pensar. Si fem un paral·lelisme amb el món informàtic, els llenguatges d'alt nivell no són tan òptims com l'assemblador, però la seva flexibilitat i la potència del maquinari fan que siguin avui dia utilitzats majoritàriament.

Finalment, el món de l'enginyeria és un compromís. Tenim més capacitat de disseny i simulació, una correcció d'errades més fàcil i més rapidesa en el disseny, cosa que finalment vol dir que el temps des que el nostre disseny és a la taula fins que el prototip és al carrer és més baix. Deixem pel camí un disseny òptim, que l'electrònica mateixa s'encarrega de compensar donant-nos milions i milions de transistors en un sol xip. Si a més fem que les connexi-

ons entre aquests transistors siguin programables, i que les puguem configurar amb, per exemple, VHDL, farem el pas definitiu en el nostre disseny electrònic de circuits.

Resum

En aquest mòdul hem avançat un pas més en els nostres dissenys digitals. Aquest avanç és paral·lel al que ha fet l'electrònica digital. Avui dia, sintetitzar grans dissenys a partir de portes és complicat, i és necessari disposar d'eines que ens permetin un nivell d'abstracció més elevat.

Per aquest motiu, hem profunditzat en l'anàlisi del llenguatge VHDL com a llenguatge descriptor del maquinari. Hem vist la seva estructura i com ens ha permès simular des de simples portes fins a circuits combinacionals i seqüencials arbitraris.

D'ara endavant, sempre que plantegem un disseny electrònic de certa entitat haurem de tenir present aquest llenguatge per a fer-ne el modelatge. Finalment, és un dels llenguatges més suportats pels fabricants, cosa que ens permetrà no quedar-nos únicament amb la idea d'un codi simulador, cosa que limitaria la seva utilitat, sinó aprofundir i utilitzar-lo per a implementar els nostres dissenys, cosa que seria impossible sense el suport dels fabricants a aquest estàndard.

Bibliografia

Chu, P. P. (2008). *FPGA Prototyping by VHDL examples: Xilinx Spartan-3 Version*. Wiley Interscience.

García Zubía, J. (2002). *Manual de VHDL: síntesis lógica para PLD's*. Deusto: Universidad de Deusto.

Zwolinski, M. (2000). *Digital System Design with VHDL*. Prentice Hall.