Departament de Disseny i Programació de Sistemes Electrònics
UNIVERSITAT POLITÈCNICA DE CATALUNYA

Arquitectura de computadors

Pràctica 1

Exemples de VHDL

Antoni Escobet

PRÀCTICA 1 – Exemple de VHDL

1. Objectius

Aprendre (refrescar) a dissenyar circuits lògics digitals mitjançant l'ús de llenguatges de descripció de maquinari com el VHDL.

2. Introducció al VHDL

VHDL (Very high speed Maquinari Description Language) és un llenguatge orientat a la descripció o modelatge de maquinari que hereta molts conceptes dels llenguatges de programació d'alt nivell com el C o el PASCAL. Malgrat aquesta certa similitud amb els llenguatges de programació, el programador en VHDL ha de tenir molt present que el codi desenvolupat serà implementat finalment en algun tipus de dispositiu lògic programable i que les característiques de VHDL busquen facilitar aquesta tasca però no eximeixen el programador de intentar que el seu codi sigui "sintetitzables", és a dir, que aquest pot ser implementat en algun dispositiu final. Per aquest motiu, és una bona idea tenir sempre present l'estructura d'allò que es vol desenvolupar per tal que els diferents components estiguin adequadament interconnectats i sincronitzats entre si per a un bon funcionament del disseny total.

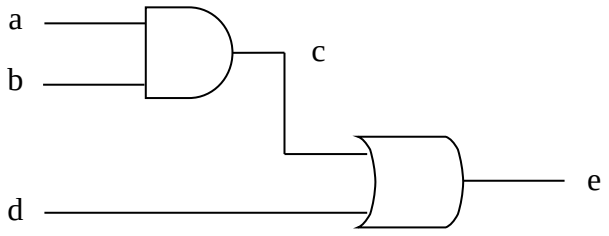
En VHDL, la interfície del dispositiu a desenvolupar amb l'exterior rep el nom d'entitat (***entity***) i la descripció de la seva funcionalitat és el que s'anomena la seva arquitectura (***architecture***). La interfície del dispositiu té com a objectiu definir quins senyals són visibles des de l'exterior, i s'anomenen ports (***ports***). En l'arquitectura es defineixen les accions que es realitzen sobre les dades introduïdes a través dels ports d'entrada per tal d'obtenir les dades que seran transmeses cap als ports de sortida.

VHDL incorpora el concepte de component (***component***) per tal d'utilitzar elements ja definits en descripcions estructurals d'un nou disseny. Així mateix, qualsevol element bàsic pot definir-se pel que s'anomena un procés (***process***). Un procés es pot entendre com un conjunt de sentències que descriuen el comportament d'un determinat element, de manera que el codi que conté aquest procés s'executa de manera seqüencial. No obstant, tots els processos continguts en una descripció VHDL s'executaran de forma paral·lela. Així, una descripció VHDL pot considerar-se com una barreja de processos executant simultàniament de manera paral·lela i és aquí on hi ha la gran diferència amb els llenguatges de programació d'alt nivell.

Els processos executant-se en paral·lel han de poder comunicar-se entre si. L'element bàsic de comunicació entre processos és el senyal (***signal***). Tot procés té una sèrie de senyals sensibles, que significa que cada vegada que un d'aquests senyals es modifiqui el seu valor, el procés s'executarà fins que trobi una seqüència de finalització del procés (***wait***). Per exemple, si volem descriure mitjançant VHDL el circuit lògic de la figura. Un possible codi seria:

```
AND2: process
begin
    c <= a and b;
    wait on a, b;
end process AND2;

OR2: process
begin
    e <= c or d;
    wait on c, d;
end process OR2;
```



El primer procés (AND2) s'executarà sempre que canviï alguna de les dues entrades *a* o *b* a aquest procés, realitzant-se en aquest cas l'operació "I lògica" entre ambdues, i deixant al procés en espera fins que alguna de les dues entrades torni a modificar-se. De la mateixa manera, el segon procés (OR2) s'executarà sempre que variï la sortida de la porta AND anterior o l'entrada *d*, executant-se en aquest cas una operació "O lògica" entre elles.

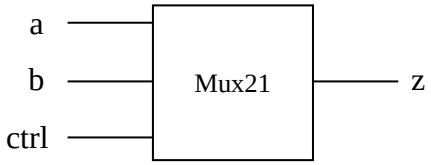
2.1. Unitats bàsiques de disseny

La declaració d'una entitat permet definir la visió externa del dispositiu que aquesta entitat representa, és a dir, la interfície amb el seu entorn. La sintaxi VHDL per a la declaració d'una entitat és la següent:

```
entity identificador is
    [genèrics]
    [ports]
    [declaracions]
[begin sentències]
end [entity] [identificador];
```

L'identificador és el nom que rebrà l'entitat i servirà per poder-lo referenciar. Totes menys la primera i l'última línia de la declaració són opcionals. No obstant, normalment se solen definir els ports de comunicació amb l'exterior. Per exemple, suposem que tenim un multiplexor 2 a 1, com el que es mostra en la figura. Una possible declaració d'una entitat que l'implementi podria ser la següent:

```
entity Mux21 is
    port ( a : in bit;
           b : in bit;
        ctrl : in bit;
           z : out bit);
end;
```



Els senyals d'entrada i de sortida són de tipus bit, el que vol dir que poden prendre els valors lògics '0 'o '1'. Hi ha un tipus de dades, definit a la llibreria *std_logic_1164*, que és el ***std_logic*** que, a més dels valors lògics '0 'i '1', permet que el senyal prengui els valors: no inicialitzat ('U'), desconegut ('X'), alta impedància ('Z'), etc. Per poder utilitzar aquest tipus de dades és necessari incloure a l'inici del codi VHDL, les dues sentències següents:

```
library IEEE;
use IEEE.std_logic_1164.all;
```

L'arquitectura defineix la funcionalitat de l'entitat i la seva sintaxi és:

```
architecture ident of ident_entitat is
    [declaracions]
begin
    [sentencies concurrents]
end [architecture] [identificador];
```

Existeixen diferents estils per definir l'arquitectura d'una entitat. L'estil algorítmic defineix la funcionalitat del dispositiu mitjançant un algoritme executat seqüencialment, de forma similar a com es faria amb un llenguatge d'alt nivell. Un exemple d'estil algorítmic pel multiplexor de dos bits seria el següent:

```
architecture Algoritme of Mux21 is
begin
    process (a,b,ctrl)
    begin
        if (ctrl = '0') then
            z <= a;
        else
            z <= b;
        end if;
    end process;
end Algoritme;
```

Una altra possibilitat és la definició de la funcionalitat mitjançant un conjunt d'equacions executades concurrentment, que determinen un flux que seguiran les dades entre mòduls encarregats d'implementar les operacions. Per al cas del multiplexor anterior, un possible codi mitjançant el flux de dades podria ser:

```
architecture FluxDades of Mux21 is
signal ctrl_n, n1, n2 : bit;
begin
    ctrl_n <= not(ctrl) after 1 ns;
    n1 <= ctrl_n and a after 2 ns;
    n2 <= ctrl and b after 2 ns;
    z <= (n1 or n2) after 2 ns;
end FluxDades;
```

Observeu que s'han definit tres senyals internes per definir la interconnexió dels diferents senyals. A més a més, s'ha inclòs un retard als senyals de sortida de cada operació intermèdia realitzada mitjançant la clàusula ***after***. En tot cas, l'ús d'aquesta clàusula no sol ser convenient ja que pot donar problemes de síntesi, pel que normalment és preferible no incloure-la en la definició de sentències concurrents.

Una darrera possibilitat seria definir l'arquitectura mitjançant un estil estructural integrat per un conjunt de components interconnectats entre si. Un possible exemple estructural pel multiplexor de dos bits seria:

```
architecture Estructural of Mux21 is
    signal ctrl_n,n1,n2 : bit;
    component INV
        port (    y : in bit;
                  z : out bit);
    end component;
    component AND2
        port (    x,y : in bit;
                  z : out bit);
    end component;
    component OR2
        port (    x,y : in bit;
                  z : out bit);
    end component;
    begin
        U0: INV port map (ctrl,ctrl_n);
        U1: AND2 port map (ctrl_n,a,n1);
        U2: AND2 port map (ctrl,b,n2);
        U3: OR2 port map (n1,n2,z);
end Estructural;
```

Quan es té que una mateixa entitat està descrita mitjançant diferents arquitectures, es farà ús de la configuració per definir quina es vol utilitzar. La sintaxi per definir una configuració és la següent:

```
configuration ident of ident_entitat is
    for ident_arquitectura
        { for (ref_component {, …} | others | all) : id_component
            use entity id_entitat[(id_arquitectura); |
            use configuration id_configuracio;]
        end for; }
    end for;
end [configuration] [ident];
```

ident és el nom que rep la configuració i servirà per poder-lo referenciar. En el cas que l'arquitectura sigui jeràrquica, caldrà definir les entitats i arquitectures que s'utilitzaran pels components de més baix nivell o identificar la configuració a utilitzar per a aquest component. Per exemple, la configuració de l'entitat per a l'arquitectura de flux de dades, es definiria mitjançant el següent codi:

```
configuration Mux21Confi of Mux21 is
    for FluxDades
    end for;
end Mux21Confi;
```

Pel cas d'una configuració amb un model jeràrquic de tipus estructural, la definició de la configuració podria ser:

```
configuration Mux21Conf of Mux21 is
   for Estructural
      for U0 : INV use work.entity INV(Algoritmic); end for;
      for all : AND2 use work.entity AND2(Algoritmic); end for;
      for U3 : OR2 use work.entity OR2(Algoritmic); end for;
   end for;
```

Un element interessant de VHDL són els paquets. Un paquet permet agrupar un conjunt de declaracions perquè puguin ser usades per diversos dispositius sense haver de ser repetides en la declaració de cadascun d'ells. Normalment en un paquet se solen definir constants, tipus i subtipus de dades, subprogrames i components. La definició d'un paquet es divideix en dues unitats de disseny diferenciades: la declaració i el cos (***body***).
La sintaxi VHDL per declarar un paquet és:

```
package identificador
   [declaracions]
end [package] [identificador];
```

i pel cos del paquet:

```
package body identificador is
   [declaracions cos]
end [package body] [identificador];
```

Com podem observar, la sintaxi és molt semblant tant en la declaració del paquet com en la definició del seu cos, l'únic que canvia és la naturalesa de les declaracions en cada un d'ells. Així, normalment solen declarar les constants i funcions en la declaració del paquet, i en el cos els seus valors i l'estructura funcional de les funcions declarades.

Un altre aspecte important en VHDL són les llibreries, que ens permeten emmagatzemar dissenys anteriors per utilitzar-los en nous dissenys. Per això, caldria declarar la llibreria a l'inici del codi VHDL i identificar el paquet o paquets que es volen utilitzar:

```
library Llibreria;
use Llibreria.PaquetExemple.all;
```

Les biblioteques work i std són excepcions en el sentit que sempre són visibles, i no requereixen de la sentència ***library***.

Realització pràctica

- La primera part d'aquesta pràctica consisteix en realitzar i comprovar el funcionament correcte del multiplexor de 2 a 1 explicat anteriorment. Per tal d'acabar d'entendre la programació estructural de components interconnectats, dissenyeu el codi en VHDL de les entitats descrites (NOT, AND i OR), i encara que es tracti de components molt simples, escriviu-los en diferents fitxers i genereu un codi per fer la comprovació del circuit.

2.2. Objectius, tipus de dades i operadors

En VHDL existeixen quatre classes diferents d'objectes: les constants, les variables, els senyals i els fitxers.

Una **constant** és un objecte que manté sempre el seu valor inicial, de manera que no pot ser modificada un cop s'ha creat. La sintaxi per declarar una constant és la següent:

constant identificador {, …} : tipus [:= expressió];

L'identificador dona nom a la constant i serveix per poder-lo esmentar, el tipus indica la naturalesa del valor que conté i l'expressió serveix per inicialitzar la constant.

Les **variables** poden canviar el seu valor un cop han estat declarades mitjançant les sentències d'assignació. Una variable no té cap analogia directa en el maquinari, pel que normalment s'utilitzen en l'estil algorítmic per emmagatzemar valors intermedis d'un procés. La sintaxi per declarar una variable és la següent:

variable identificador {, …} : tipus [:= expressió];

Observeu que excepte la paraula reservada, la sintaxi per declarar una variable és anàloga a la d'una constant. Per modificar el valor d'una variable s'utilitzen sentències d'assignació, que per al cas de les variables prenen la forma següent:

identificador := expressió;

Un **senyal** és un objecte que pot modificar el seu valor dins dels possibles valors del seu tipus, però que té una analogia directa en el maquinari, ja que es pot considerar com una abstracció d'una connexió física o d'un bus. Al contrari que les variables, no es limita a un procés sinó que serveix per interconnectar components d'un circuit i per sincronitzar l'execució i suspensió de processos. La sintaxi en VHDL per declarar un senyal és:

signal identificador {, …} : tipus [:= expressió];

A diferència de les variables, un senyal no es declararà en la part declarativa d'un procés sinó en l'arquitectura del dispositiu.

Els ports d'una entitat són senyals que s'utilitzen per interconnectar el dispositiu amb altres dispositius. La seva declaració és una mica especial, ja que a part de determinar un identificador i un tipus de dades cal indicar l'adreça del senyal respecte a l'entitat. La secció de declaració de ports d'una entitat té la següent sintaxi:

port ({identificador {, …} : direcció tipus [:= expressió];});

En aquest cas, l'expressió opcional s'utilitzarà en el cas de que el port estigui desconnectat.

El **fitxer** és un objecte que permet comunicar un disseny VHDL amb un entorn extern, de manera que un model pugui escriure i llegir dades. Un ús bastant comú dels fitxers és el d'emmagatzemar els estímuls de simulació que es volen aplicar al model en un fitxer d'entrada i salvar els resultats de simulació en un fitxer de sortida per al seu posterior estudi. La sintaxi per declarar un fitxer és la següent:

file identificador {, …} : tipus_fitxer [**is** direcció "nom";]

El tipus de dades és un concepte fonamental en VHDL, ja que cada objecte ha de ser d'un tipus concret que determinarà el conjunt de valors que pot assumir i les operacions que es podran realitzar amb aquest objecte. La declaració d'un nou tipus de dades pren la següent forma:

```
type identificador is definició_tipus;
```

La part de definició de tipus serveix per indicar el conjunt de valors del tipus. Per exemple:

```
type DiaMes is range 1 to 31;
type PuntsCardinals is (nord,sud,est,oest);
```

Un cop definit un tipus de dades, es poden declarar objectes d'aquest tipus. Els tipus existents de dades escalars es poden classificar en enters i reals, físics i enumerats. La sintaxi per declarar als primers d'ells és:

```
type identificador is range literal to | downto literal;
```

Depenent de si s'escriuen literals sencers o en punt flotant, el tipus de dades declarat serà de tipus sencer o real. Les paraules reservades ***to*** i ***downto*** s'utilitzen per indicar un rang creixent o decreixent. Els tipus físics serveixen per representar mesures del món real i la seva sintaxi de declaració és la següent:

```
type identificador is range literal to | downto literal
   units
      identificador;
      {identificador = literal_físic;}
   end units [identificador];
```

Un exemple per definir un tipus físic de temps seria el següent:

```
type temps is range 0 to 1E20
   units
      fs;
      ps = 1000 fs;
      ns = 1000 ps;
      us = 1000 ns;
      ms = 1000 us;
      sec = 1000 ms;
      min = 60 sec;
      hr = 60 min;
   end units;
```

Finalment, el tipus enumerat defineix un conjunt específic de possible valors mitjançant una llista on es defineixen tots i cadascun dels valors possibles. La sintaxi per declarar un tipus enumerat és la següent:

```
type identificador is ( identificador | caràcter {, …});
```

Alguns exemples de tipus enumerats podrien ser:

```
type comandes is (esquerra,dreta,amunt,avall,tret);
type tecles is ('a','d','w','x','');
type mescla is ('e',esquerra,'d',dreta);
type logic is (false,true);
type bit is ('0','1');
```

Donat un tipus de dades, es pot definir un subtipus del mateix mitjançant la clàusula:

```
subtype identificador is id_tipo [range literal to | downto literal];
```

Per exemple, el tipus *natural* o el tipus *positiu* són subtipus del tipus integer:

```
subtype natural is integer range 0 to integer'high;
subtype positiu is integer range 1 to integer'high;
```

També existeixen els anomenats tipus de dades compostos constituïts pels vectors i els registres. En els primers d'ells, tenim un conjunt d'objectes del mateix tipus ordenats mitjançant un o més índexs que indiquen la posició de cada objecte dins del vector. Per declarar un vector s'haurà de crear un tipus que determini la base del tipus dels objectes que formaran el vector i el rang dels índexs que sempre serà d'un tipus discret, és a dir, sencer o enumerat. La sintaxi per declarar un vector és la següent:

```
type identificador is array (rang {, …}) of tipus_objectes;
```

L'identificador dóna nom al vector. Els rangs es poden descriure explícitament en la declaració o bé es pot donar directament un nom de tipus o subtipus que ja inclogui una restricció de rang i, finalment, el tipus indicarà el conjunt de valors possibles que poden prendre els objectes del vector. Uns exemples poden ser:

```
type Byte is array (0 to 7) of bit;
subtype Decimal is character range '0' to '9';
type Byte2 is array (Decimal range '0' to '7') of bit;
type PuntsCardinals is (nord,sud,est,oest);
type Estat is array (PuntsCardinals range nord to est) of integer;
```

A continuació es podrien declarar objectes dels tipus anteriors:

```
variable operador1 : Byte;
variable opeardor2, operador3 : Byte2;
variable EstatActual : Estat;
```

Amb els vectors, es poden fer sentències d'assignació com:

```
operador1 := "10010101";
operador1(3) := '1';
operador1(3 to 6) := "1001";
operador2('5') := '1';
operador3 := operador2;
EstadoActual(norte) := 35;
```

també és possible crear vectors de més d'una dimensió. Per exemple:

```
type Memoria is array (0 to 7, 0 to 63) of bit;
variable RamA, RamB : Memoria;
```

Possibles sentències d'assignació podrien ser:

```
RamA := RamB;
RamA(4,7) := '1';
```

Quan es vol declarar un tipus de vector no restringit la sintaxi és:

```
type identificador is array (tipus_índex range <> {, …}) of tipus_objecte;
```

Un exemple seria la definició d'una cadena com una successió de caràcters:

```
type string is array (positiu range <>) of character;
```

L'altra classe d'objecte compost és el **registre** que, a diferència dels vectors, està format per unitats atòmiques de diferent tipus, que reben el nom de camps. La sintaxi per declarar un registre és:

```
type identificador is record
    identificador {, …} : tipus;
    {…}
end record [identificador];
```

Un exemple, pot ser:

```
type Fecha is record
Dia : integer range 1 to 31;
Mes : integer range 1 to 12;
Anyo : integer range 0 to 2100;
end record;
```

Finalment, tractarem el tema dels operadors que poden utilitzar-se per generar expressions en VHDL. Els operadors bàsics definits en VHDL es mostren a la següent taula.

Operador	Descripció	Tipus d'perants	Resultat
**	potència	sencer *op* sencer real *op* sencer	Sencer real
abs	valor absolut	numèric	igual operant
not	negació	bit, booleà, vector bits	igual operant
*	multiplicació	sencer *op* sencer real *op* real físic *op* sencer físic *op* real sencer *op* físic real *op* físic	sencer real físic físic físic físic
/	divisió	sencer *op* sencer real *op* real físic *op* sencer físic *op* real físic *op* físic	sencer real físic físic sencer
mod	mòdul	sencer *op* sencer	sencer
rem	restant	sencer *op* sencer	sencer
+	més (bit a bit)	numèric	igual operant
-	menys (bit a bit)	numèric	igual operant
+	suma	numèric *op* numèric	igual operants
-	resta	numèric *op* numèric	igual operants
&	Concatenació	vector *op* vector vector *op* element element *op* vector element *op* element	vector vector vector vector
=	igual que	no fitxer *op* no fitxer	booleà
/=	diferent que	no fitxer *op* no fitxer	booleà
<	menor que	no fitxer *op* no fitxer	booleà
>	més gran que	no fitxer *op* no fitxer	booleà
<=	menor o igual que	no fitxer *op* no fitxer	booleà
>=	més gran o igual que	no fitxer *op* no fitxer	booleà
and	I lògic	bit, booleà, vector bits *op* bit, booleà, vector bits	igual operants
or	O lògic	bit, booleà, vector bits *op* bit, booleà, vector bits	igual operants
nand	I lògic negat	bit, booleà, vector bits *op* bit, booleà, vector bits	igual operants
nor	O lògic negat	bit, booleà, vector bits *op* bit, booleà, vector bits	igual operants
xor	O exclusiu	bit, booleà, vector bits *op* bit, booleà, vector bits	igual operants
xnor	O exclusiu negat	bit, booleà, vector bits *op* bit, booleà, vector bits	igual operants

2.3. Sentències seqüencials

Les sentències seqüencials són aquelles que ens permeten modelar la funcionalitat d'un component. Les podem classificar en sentències d'assignació (variable o senyal), sentències condicionals (*if*, *case*), sentències iteratives (*loop*, *exit*, *next*), altres sentències (*wait*, *assert*, *null*) i crides a subprogrames.

La sentència ***wait*** s'utilitza per suspendre l'execució d'un procés. La seva sintaxi és:

```
[etiqueta:] wait [on senyal {, ...}]
                 [until expressió_booleana]
                 [for expressió_temps]
```

L'assignació a *senyal* com sentència seqüencial presenta la següent sintaxi:

```
[etiqueta:] nom_senyal <= valor [after expressió_temps];
```

La forma d'assignació a *senyal* dependrà bàsicament del model de retard triat. VHDL permet escollir entre dos tipus de retard: el transport (***transport***) i el inercial (***inertial***). El model de transport propaga qualsevol canvi que es produeixi, mentre que el inercial filtra aquells canvis de durada inferior a un mínim. El model de transport és el que reflecteix una línia de transmissió ideal, mentre que el model inercial és el que regeix el comportament d'una porta lògica.

Per defecte, VHDL suposa que les assignacions a *senyal* segueixen el model inercial. Per utilitzar el model de transport s'ha d'especificar en l'assignació:

```
[etiqueta:] nom_senyal <= [transport] valor [after expressió_temps];
```

L'assignació a una variable substitueix el valor actual de la variable amb el valor especificat per una expressió. La seva sintaxi és la següent:

```
[etiqueta:] nom_variable := expressió;
```

La sentència ***if*** s'utilitza per escollir en funció d'alguna condició quines sentències s'han d'executar. La seva sintaxi és la següent:

```
if condició then sentencies_seqüencials
   {elsif condició then sentencies_seqüencials}
   [else sentencies_seqüencials]
end if;
```

Les condicions han de ser de tipus booleà, de manera que tornin verdader (true) o fals (false) per tal de veure si s'executen les sentències seqüencials indicades.

La sentència ***case*** s'utilitza per escollir quin grup de sentències s'han d'executar entre un conjunt de possibilitats, basant-se en el rang de valors d'una determinada expressió de selecció. La seva sintaxi és la següent:

```
[etiqueta:] case expressió is
   when valor => sentencies_seqüencials;
   {when valor => sentencies_seqüencials;}
end case;
```

Es pot utilitzar la paraula clau ***others*** a valor per especificar tots els altres rangs no declarats específicament. En aquest cas, cal especificar aquesta opció l'última, després de tots els altres casos.

La sentència ***loop*** s'utilitza per executar un grup de sentències seqüencials de manera repetida. El nombre de repeticions es pot controlar en la mateixa sentència amb diferents opcions. La seva sintaxi és:

```
[etiqueta:] [while condició_booleana | for control_repetició] loop
    sentencies_seqüencials;
end loop [etiqueta];
```

Podem utilitzar la sentència ***loop*** sense cap tipus de control sobre el nombre de repeticions del bucle, de manera que es provoqui l'execució infinita del grup de sentències seqüencials especificades.

La sentència ***exit*** està relacionada amb la sentència ***loop***, i ofereix una manera d'acabar l'execució del bucle. La seva sintaxi és:

```
[etiqueta:] exit [etiqueta_loop] [when condició_booleana];
```

La sentència ***next*** s'utilitza per aturar l'execució d'una sentència loop i passar a la següent iteració. La seva sintaxi és:

```
[etiqueta:] next [etiqueta_loop] [when condició_booleana];
```

La sentència ***assert*** permet reportar missatges en funció de si una determinada condició es compleix o no. També permet interrompre la simulació en funció d'aquesta condició. La seva sintaxi és:

```
[etiqueta:] assert expressió_booleana [report cadena_caràcters]
[expressió_exigència];
```

El valor de l'expressió d'exigència ha de ser: *note*, *warning*, *error* o *failure*. En cas de no especificar el nivell d'exigència, per defecte és *error*. Normalment el simulador permet a l'usuari determinar per quin nivell d'exigència s'ha d'interrompre la simulació.

Els subprogrames (procediments o funcions) que tinguem definits en alguna part del codi poden ser cridats per a la seva execució en qualsevol part d'un codi seqüencial. La sintaxi de crida a un procediment (***procedure***) és:

```
[etiqueta:] nom_procediment [{paràmetres, ...}];
```

La sintaxis de la crida a una funció (***function***) es:

```
nom_funció [{paràmetres, ...}];
```

La diferència és que una crida a una funció forma part d'una expressió d'assignació o és l'assignació en si mateixa, mentre que la crida a un procediment és una sentència seqüencial per si sola.

La sentència ***return*** s'utilitza per acaba l'execució d'un subprograma. La seva sintaxi general és:

```
[etiqueta:] return [expresión];
```

En un procediment no es retorna cap expressió, mentre que en una funció es retorna el valor a assignar.

La sentència ***null*** s'usa per indicar que no s'ha de realitzar cap acció. La seva sintaxi general és:

```
[etiqueta:] null;
```

És útil, per exemple, en sentències ***case*** per indicar que en determinats casos (opcions) no es faci res.

2.4. Sentències concurrents

Les sentències concurrents són aquelles que s'executen en paral·lel, de manera que no estan incloses en cap procés, sinó que apareixen en l'arquitectura del model.

Un procés és una sentència concurrent que defineix un comportament a través de sentències seqüencials. Qualsevol sentència concurrent o seqüencial en VHDL té el seu procés equivalent. La sintaxi general d'un procés és:

```
[etiqueta:] process [(nom_senyal {, ...})] [is]
declaracions;
begin
   sentencies_seqüencials;
end process [etiqueta];
```

Les assignacions a senyal poden donar-se en el món concurrent. La seva sintaxi és molt semblant a l'assignació seqüencial:

```
[etiqueta:] nom_senyal <= [transport] forma_onda;
```

L'assignació concurrent condicional és una forma compacta d'expressar les assignacions a senyal usant la sentència ***if*** seqüencial. La seva sintaxi és:

```
[etiqueta:] nom_senyal <= [transport]
   {forma_onda when expressió_booleana else}
   forma_onda [when expressió_booleana];
```

L'assignació concurrent amb selecció és una forma compacta d'expressar les assignacions a senyal usant la sentència ***case*** seqüencial. La seva sintaxi és:

```
[etiqueta:] with expressió select
   nom_senyal <= [transport]
      {forma_onda when valor,}
      forma_onda when valor;
```

La sentència ***assert*** pot donar-se també en el món concurrent. La seva sintaxi és:

```
[etiqueta:] assert expressió_booleana [report expressió]
[expressió_exigència];
```

La crida concurrent a un subprograma pren la següent forma per un procediment:

```
[etiqueta:] nom_procediment [{paràmetres , ...}];
```

Per una funció, se tindrà:

```
nom_funció [{paràmetres, ...}];
```

2.5. Sentències estructurals

VHDL proporciona una llista de declaracions dedicades a la descripció estructural de maquinari. Són també sentències concurrents ja que s'executen en paral·lel amb qualsevol altra sentència concurrent de la descripció i apareixen en l'arquitectura d'un model fora de qualsevol procés.

La declaració d'un component pot aparèixer en una arquitectura o en un paquet. Si apareix en l'arquitectura, llavors es poden fer còpies del component en aquesta arquitectura, si apareix en un paquet, es poden fer

còpies del component en totes aquelles arquitectures que facin servir aquest paquet. La sintaxi de la declaració d'un component és:

```
component nom_component [is]
    [generic (llista_genèrics);]
    [port (llista_ports);]
end component [nom_component];
```

Un cop declarat un component, es poden fer referències a si mateix dins de l'arquitectura. La sintaxi de referència a un component és:

```
[etiqueta_referència:] nom_component
    [generic map (llista_associació);]
    [port map (llista_associació)];
```

També és possible referenciar un component sense necessitat d'haver-lo declarat abans. La sintaxi de referència és en aquest cas:

```
[etiqueta_referència:] entity nom_entitat[(nom_arquitectura)]
    [generic map (llista_genèrics);]
    [port map (llista_ports)];
```

Una forma habitual de descriure estructures al maquinari és la realització de múltiples còpies d'elements iguals (o semblants). Aquestes descripcions estructurals es podrien realitzar amb la còpia o referència a component que s'ha descrit anteriorment, però VHDL ofereix una forma més còmoda i compacta per realitzar descripcions que es basen en la repetició de la mateixa estructura: la sentència ***generate***. La seva sintaxi és:

```
[etiqueta_generate:] {[for especificació_for | if condició]} generate
    {sentencies_concurrents}
end generate;
```

Una altra possible sintaxi per la sentència ***generate*** que permet la inclusió d'una part declarativa és la següent:

```
[etiqueta_generar:] {[for especificació_for | if condició]} generate
    [{part_declarativa}
    begin]
        {sentencies_concurrents}
end generate;
```

A la part declarativa pot aparèixer qualsevol element que pugui aparèixer a la part declarativa d'una arquitectura (constants, tipus, subtipus, subprogrames i senyals).

Com s'ha vist abans, la configuració d'un disseny permet escollir quina de les possibles arquitectures d'una entitat serà utilitzada. La configuració d'un disseny pot aparèixer en una arquitectura, llavors es diu especificació de configuració, o pot aparèixer com una unitat de disseny independent, llavors es diu declaració de configuració. Si dins la mateixa arquitectura, en la qual s'usa una referència a altres components, es vol indicar quina arquitectura es vol utilitzar per a cada un dels components referenciats, es pot utilitzar l'especificació de configuració. La sintaxi general per a l'especificació de configuració és la següent:

```
for (ref_component {, ...} | others | all) : id_component
    use entity id_entitat[(id_arquitectura); |
    use configuration id_configuació;]
```

Quan la configuració s'utilitza com un disseny independent, la sintaxi d'aquesta declaració de configuració és:

```
configuration identificador of identificador_entitat is
    for identificador_arquitectura
        {for (ref_component {, ...} | others | all): id_component
            use entity id_entitat[(id_arquitectura); |
            use configuration id_configuració;]
        end for; }
    end for;
end [configuration] [identificador];
```

Tal com hem vist anteriorment, VHDL ofereix la possibilitat de generalitzar un model afegint uns paràmetres anomenats genèrics (***generics***) en la definició de l'entitat del model. Si volem desenvolupar un model genèric, hem d'incloure a l'entitat del mateix la clàusula ***generic***, i incloure els paràmetres genèrics del model. Per donar valors concrets a un mòdul amb genèrics quan és usat com component s'ha d'utilitzar la clàusula ***generic map***.

La sentència concurrent ***block*** és una forma de reunir o agrupar sentències concurrents, a més de permetre compartir declaracions d'objectes que seran visibles només per les sentències englobades en la sentència ***block***. La sintaxi d'aquesta sentència és la següent:

```
etiqueta: block [expressió_guarda] [is]
    [generic (llista_genèrics);
    [generic map (llista_associació_genèrics);]]
    [port (llista_ports);
    [port map (llista_associació_ports);]]
    {part declarativa}
begin
    {sentencies_concurrents};
end block [etiqueta];
```

Els subprogrames s'usen per escriure algorismes reutilitzables. Els subprogrames consten de dues parts: la definició del subprograma i la definició del cos del subprograma. Per al cas de les funcions, la sintaxi de definició és:

```
function nom_funció [(llista_paràmetres)] return tipus_retorn;
```

La sintaxis de definició del cos d'una funció es:

```
[pure | impure] function nom_funció [(llista_paràmetres)] return tipus_retorn is
    {part_declarativa}
begin
        {sentencies_seqüencials};
end [function] [nom_funció];
```

Una funció es considera pura (***pure***) si donat un conjunt de valors dels seus paràmetres d'entrada sempre retorna el mateix resultat, mentre que una funció impura (***impure***) pot trencar estar regla.

Per al cas dels procediments, la sintaxi de definició és:

```
procedure nom_procediment [(llista_paràmetres)];
```

La sintaxis per la definició del cos del procediment es:

```
procedure nom_procediment [(llista_paràmetres)] is
    {part_declarativa}
begin
    {sentencies_seqüencials};
end [procedure] [nom_procediment];
```

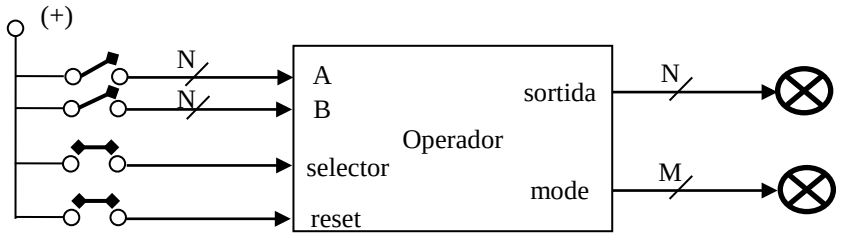
Els paràmetres formals d'un procediment poden ser de tres tipus: ***in***, ***out*** i ***inout***, i per defecte es consideren de tipus ***in***. A les funcions, normalment els paràmetres seran tots de tipus ***in***.

Finalment, cal destacar que VHDL permet la sobrecàrrega de subprogrames, és a dir, es pot definir una mateixa funció (amb el mateix nom) que variï la seva funcionalitat segons quins siguin els tipus dels seus paràmetres d'entrada. Així, és possible definir una mateixa funció per a diferents tipus d'arguments. Un exemple seria la sobrecàrrega d'un operador (que s'ha d'especificar entre cometes dobles), per exemple l'operador suma o l'operador "i lògic", per uns tipus definits per l'usuari:

```
function "+" (a: MeuTipus; b: MeuTipus) return MeuTipus;
function "and" (a: MeuTipus; b: MeuTipus) return MeuTipus;
```

3. Realització practica

En aquesta primera pràctica sobre disseny en VHDL, anem a familiaritzar-nos amb el llenguatge VHDL. Per això, anem a dissenyar un sistema que ens permeti realitzar diferents operacions sobre unes entrades controlades a través d'interruptors. Mitjançant un polsador es podrà seleccionar el tipus d'operació a realitzar (per polsos). Les sortides actuaran com a dispositius de visualització (sobre uns LED's) dels resultats obtinguts.



En primer lloc, dissenyeu un paquet que inclogui les funcions de càlcul de l'operació i selecció de la mateixa. A continuació feu el codi del sistema, per N = 8 utilitzant les funcions del paquet anterior.

En aquest exemple es proposa que el sistema disposi de 6 operacions (necessita 3 bits per codificar-lo, M = 3). Les operacions a realitzar son:

Operació	mode	Sortida
Sumar	000	A + B
Restar	001	A – B
Multiplicar	010	A * B
XOR	011	A xor B
AND	100	A anb B
OR	101	A or B
-	111	0

El mode de funcionament del sistema és el següent:

1) Quan és prem el senyal de reset, l'estat o el mode d'operació s'estableix a la suma (mode = 000) i genera l'operació seleccionada.
2) Si es prem el selector es passa al següent mode d'operació. Per exemple, si està a la suma (000) i es prem el polsador del selector, l'operació pesarà a ser la resta (001) i així successivament fins a l'ultima operació, que saltarà a la primera.
3) Sempre que es modifiqui alguna de les entrades (A o B), s'ha de calcular el resultat.

Una proposta del paquet pot ser:

```
package paquet is
        constant N : natural := 8;
        constant M : natural := 3;
        type operacio is (suma, resta, multiplicacio, op_xor, op_and, op_or);
        function op_seg (operacio_actual : operacio) return operacio;
        function calcular(a, b : signed(N-1 downto 0); operacio_actual : operacio) return signed;
        function tipus(operacio_actual : operacio) return std_logic_vector;
end package paquet;
```

on

- *op_seg* és una funció que té com entrada l'operació actual i retorna la següent operació. Només s'ha d'executar quan es modifica l'entrada *selecció*.
- *calcular* és una funció que retorna el resultat de l'operació actual amb els operants de les entrades A i B. Aquesta funció s'ha d'executar quan es modifiquen les entrades *A*, *B* o la de *selecció*.
- *tipus* retorna el codi de l'operació actual. S'executa quan es modifica l'entrada de *selecció*

L'ultima part de la pràctica serveix per verificar el funcionament correcte del disseny realitzat. Heu de dissenyar un joc de proves en VHDL per comprovar el seu funcionament sobre el simulador: ModelSim d'Altera. Podeu consultar l'annex

Presentació de la practica: Aquesta practica està pensada per fer-la en una sola sessió de laboratori. Però, degut a la gran quantitat de text que hi ha, podreu presentar-la (demostrar el seu funcionament) a la propera sessió de pràctiques (el dia 10 de març).

S'ha de fer una memòria amb els codis comentats dels tres programes de VHDL i unes pantalles amb els resultats de la simulació.