Facultat d'Informàtica de Barcelona
2016-2017 Q2

# Visualitzador de circuits lògics

Belén San Martín Díez
Adrián Sánchez Albanell
Grup 12

## INDEX

# Descripció del projecte

El projecte consisteix en generar un traductor capaç de llegir descripcions de circuits lògics en terme d'equacions i composició de mòduls i generar-ne la seva representació gràfica. Per a la descripció dels circuits inicial utilitzarem un subconjunt de Verilog. Per a generarla gramàtica utilitzarem antlr3 i per al resultat gràfic farem ús del paquet TikZ per LaTeX, que inclou una llibreria per a circuits lògics.

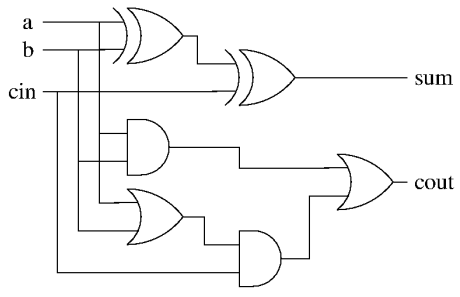
Aquí podem veure un exemple d'entrada per al nostre programa. Es tracta d'un sumador de 4 bits:

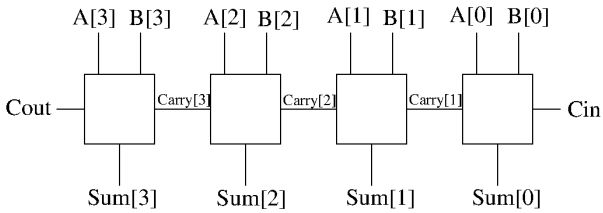
```
module full_adder (a, b, cin, sum, cout);
  input  a, b, cin;
  output sum, cout;
  sum  = a ^ b ^ c;
  cout = a & b | cin & (a | b);
endmodule

module adder4 (A, B, Cin, Sum, Cout);
  input  [3:0] A, B;
  input  Cin;
  output [3:0] Sum;
  output Cout;
  wire   [3:1] Carry;

  full_adder FA0 (A[0], B[0], Cin, Sum[0], Carry[1]);
  full_adder FA1 (A[1], B[1], Carry[1], Sum[1], Carry[2]);
  full_adder FA2 (A[2], B[2], Carry[2], Sum[2], Carry[3]);
  full_adder FA3 (A[3], B[3], Carry[3], Sum[3], Cout);
endmodule
```

La sortida generada pel nostre programa amb l'entrada anterior hauria de ser la següent:



# Subconjunt Verilog

Com ja hem mencionat, en la nostra pràctica només farem ús d'una part del llenguatge Verilog. En les següents pàgines descriurem les funcionalitats que inclourà aquest subconjunt.

## El llenguatge

### Comentaris

Com en el cas de C++, poden ser de línia, marcats pels caràcters "//" i anant des d'aquests fins al final de la línia, o de bloc, començant amb els caràcters "/*" fins que es tanqui el bloc amb els caràcters "*/".

### Números

L'expressió general per constants numèriques en Verilog té la forma:

```
<mida> '<base> <número>
```

- La mida indica el nombre de bits del número. És opcional. Si no s'indica, serà 32 bits per defecte.
- La base indica en quina base s'expressa el número. És opcional i si no s'indica serà decimal.

    - b: base binaria.
    - o: base octal.
    - d: base decimal.
    - h: base hexadecimal.

- El número es la quantitat i estarà expressada en la base corresponent.
  Les xifres permeses són 0-9 i A-F o a-f, segons la base en la que estigui el número, i els caràcters "x" o "X" per indicar valor d'un bit desconegut i "z" o "Z" per indicar alta impedància en bits concrets.
  També és pot intercalar el caràcter "_" al número per a facilitar la lectura.

Per indicar nombres negatius s'afegeix el símbol "-" davant del número.

**Exemples:**

187, 8'h0a, -4'b10, 16'o73, 'hA6

## Identificadors

Un identificador està format per una lletra o "_" seguit de lletres, números i els caràcters "$" o "_". Distingeix entre majúscules i minúscules.

## Tipus de dades estructurals

Hi ha de dos tipus:

- ***reg***: representen registres.
- ***wire***: representen una connexió del circuit (un cable, per exemple). Permeten representar la connexió d'elements entre ells o connexions d'entrada i/o sortida.

Per defecte seran variables d'un bit, però se'n poden declarar de més. La sintaxis per definir variables es:

- Per variables d'un bit:

```
<tipus> <nom>
```

- Per variables de diversos bits:

```
<tipus> <[msb : lsb]> <nom>
```

On *msb* i *lsb* són nombres naturals i *msb* és el bit més significatiu i *lsb* el menys significatiu de la variable.
És poden definir diverses variables en una sola línia.

**Exemple:**

```
wire [5:3] C; // C és un cable de 6 bits, on C[3] és el bit menys significatiu.
reg a, b; // a i b registres.
```

## parameter

Una variable tipus parameter representa una constant numèrica o expressió booleana. Després de definir una variable de tipus parameter, aquesta pot ser utilitzada en qualsevol expressió menys dins un *assign* (es veurà més endavant).
La sintaxis per definir-ne una és:

```
parameter <nom> = <expressió>
```

**Exemple:**

```
parameter ADD = 1'b1;
parameter TRUE = 1, FALSE = 0;
parameter [1:0] S0 = 3, S1 = 1, S2 = 2;
// S0, S1 i S2 seran constants de 2 bits amb el valor indicat.
```

## Estructura general

De les possibilitats que ens ofereix Verilog, nosaltres ens centrarem únicament en les enfocades a la definició estructural de circuits.

La descripció dels circuits estarà formada per conjunts de mòduls. Cada mòdul pot tenir un conjunt d'entrades i sortides amb les que es pot connectar a altres mòduls.

A Verilog no existeixen variables globals. Fora els mòduls només hi pot haver directives de compilador, que afecten el fitxer a partir que apareixen.

### Estructura dels mòduls

```
module <nom del mòdul> ( <senyals> );
      <Declaració de senyals>
      <Descripció del diseny>
endmodule
```

### Senyals

Conjunt de senyals. Una senyal pot ser o bé un identificador, un bit o conjunt de bits d'un vector de bits declarat al mòdul o una concatenació dels anteriors. Les senyals s'hauran de declarar dins el mòdul en algun moment.

### Declaració de senyals

Les senyals definides com a paràmetres han de ser declarades. Hi ha tres maneres de declarar-les:

- **Inputs**: les senyals declarades com a inputs no cal que siguin definides i són sempre del tipus *wire*.
- **Output**: poden ser del tipus *reg* o *wire*. S'haurà de definir de quin tipus són després de les declaracions. Si no se'ls assigna tipus, seran considerades del tipus *wire* per defecte.
- **Inout**: són variables d'entrada i sortida i són de tipus *wire*.

La declaració de senyals es pot fer en qualsevol moment dins el mòdul. Totes les senyals s'han de definir com a input, output o inout i han d'estar declarades abans de poder ser utilitzades en qualsevol expressió o definir de quin tipus són si fes falta (nomes en el cas dels outputs).

**Exemple:**

```
input [7:0] in0, in1; // dues senyals input de 8 bits.
input sel;
output [3:0] out1, out2;
wire [3:0] out1; // realment no fa falta, ja que és wire per defecte.
reg [3:0] out2;
```

### Nodes interns

A part de les senyals, es poden declarar altres objectes de tipus *wire* o *reg* internament dins un mòdul.

### Declaració d'altres mòduls

Dins un mòdul podem fer us d'altres mòduls declarats anteriorment. Per indicar-ho és declarara una variable del tipus del mòdul que es vulgui declarar, se li donarà un nom per identificar-lo i es passaran com a paràmetres les variables necessàries en l'ordre i del tipus corresponent.
La sintaxi és la següent:

```
<tipus mòdul> <nom declaració mòdul> (<senyals>);
```

**Exemple de mòdul:**

```
module full_adder (a, b, cin, sum, cout);
  input  a, b, cin;
  output sum, cout;
  sum  = a ^ b ^ c;
  cout = a & b | cin & (a | b);
endmodule
```

**Exemple de declaracions del mòdul anterior dins d'un altre mòdul:**

```
module adder4 (A, B, Cin, Sum, Cout);
  input  [3:0] A, B;
  input  Cin;
  output [3:0] Sum;
  output Cout;
  wire   [3:1] Carry;

  full_adder FA0 (A[0], B[0], Cin,   Sum[0], Carry[1]);
  full_adder FA1 (A[1], B[1], Carry[1], Sum[1], Carry[2]);
  full_adder FA2 (A[2], B[2], Carry[2], Sum[2], Carry[3]);
  full_adder FA3 (A[3], B[3], Carry[3], Sum[3], Cout);
endmodule
```

## Assignació continua

Serveix per modelar lògica combinacional. Permet portar un valor a un *wire*. La seva sintaxi és:

```
assign <variable> = <expressió de tipus wire, reg o constant numèrica>
```

La variable ha de ser del tipus *wire*, un conjunt de bits d'un vector *wire*, o una concatenació dels anteriors, i estat declarada anteriorment.
L'expressió ha de ser un operador o una expressió que utilitzi funcions i/o variables (prèviament declarades).
Si els bits retornats per l'expressió són majors que els de la variable, els bits de més valor sobrants són descartats.
Altrament, si els bits de la variable són més que els retornats en l'expressió, els restants són posats a 0.

També es pot fer una assignació al moment de declarar el wire:

```
wire <nom wire> = <expressió de tipus wire, reg o constant numèrica>
```

**Exemple:**

```
wire cout;
wire [7:0] sum;
reg [3:0] a, b;
assign {cout, sum} = a + b;
wire c = |(a & b);
assign [5:2] sum = 4'b11 - b;
```

## Assignació

També es pot assignar expressions a variables directament de la forma:

```
<nom variable> <rang> = <expressió de tipus wire, reg o constant numèrica>
```

El rang és opcional i pot ser entre certs valors o un accés a vector.

**Exemple:**

```
wire [5:0] a;
a[4:2] = x ^ b ^ c;
wire [4:2] m;
m[3] = 1'b1;
```

# Operadors

## Operadors aritmètics unaris

- Operador “+”
  **Exemple**: a = +8’h12;
- Operador “-”
  **Exemple**: b = -23’b111;

## Operadors aritmètics binaris

- Operador “+”
  **Exemple**: a = b + c;
- Operador “-”
  **Exemple**: a = b - c;
- Operador ”*”
  **Exemple**: a = b * c;
- Operador ”/”: divisió entera.
  **Exemple**: a = b / c;
- Operador ”%”
  **Exemple**: a = b % c;

Si algun bit es indefinit (x), el resultat es x.
Els nombres negatius en complement a 2.

## Operadors relacionals

- Operador “<”
  **Exemple**: a < b;
- Operador “>”
  **Exemple**: a > b;
- Operador “<=”
  **Exemple**: a <= b;
- Operador “>=”
  **Exemple**: a >= b;

Si algun bit es indefinit (x), el resultat es x.

## Operadors d'igualtat

- Operador "=="
  **Exemple**: a == b;
- Operador "!="
  **Exemple**: a != b;

La comparació es fa bit a bit. Els bits indefinits (x) s'ignoren.

## Operadors lògics

- Operador "!" negació lògica.
  **Exemple**: !a
- Operador "&&" AND lògica.
  **Exemple**: a && b
- Operador "||" OR lògica.
  **Exemple**: a || b

Si algun operant conte algun bit indefinit (x) o d'alta impedància, el resultat es x.
Els operadors lògics retornen 1 en cas que s'evalui l'expressió com a certa, 0 en cas contrari (menys en el cas mencionat anteriorment).

## Operadors bit-wise (bit a bit)

- Operador "~" negació.
  **Exemple**: ~a
- Operador "&" AND.
  **Exemple**: a & b
- Operador "|" OR.
  **Exemple**: a | b
- Operador "^" OR exclusiva o XOR.
  **Exemple**: a ^ b

Es poden combinar operadors per obtenir-ne d'altres (com NAND amb "~&" o XNOR amb "~^", etc).

Per als bits indefinits (x):

~x = x
0&x = 0
1&x = x
1|x = 1
0|x = x
0^x = 1^x = x

## Operadors de reducció

- Operador “&” AND.
  **Exemple**: & a
- Operador “|” OR.
  **Exemple**: |a
- Operador “^” OR exclusiva o XOR.
  **Exemple**: ^ a
- Operador “~&” NAND.
  **Exemple**: ~& a
- Operador “~|” NOR.
  **Exemple**: ~| a
- Operador “~^” o “^~” OR exclusiva negada o XNOR.
  **Exemple**: ~^ a

El resultat es aplicar l’operació entre tots els bits de la variable. En el cas de les negacions, primer s’aplica l’operador que no és la negació i després es fa la negació del resultat.

## Operadors de desplaçament

- Operador “<<”
  **Exemple**: a = b << 2;
- Operador “>>”.
  **Exemple**: a = b >> 2;

Els bits desplaçats s’omplen amb 0.

## Operador condicional

- Operador “? :”
  **Exemple**: a = (b == c) ? a + b : a - c;

## Operador concatenació

- Operador “{ }” permet concatenar els operands per formar un vector. Els operands se separen per comes. Els operands poden ser qualsevol expressió menys constants sense mida definida.
  **Exemple**:

```
{a, b} // El resultat tindrà mida bits d'a més bits de b
{ a[4:0], b[2:0] } // El resultat tindra mida 8
{ 3{a} } // Equivalent a { a, a, a }
{b, 2{ c, d } } // Equivalent a { b, c, d, c, d }
```

## Precedència d'operadors

Taula amb la precedència dels operadors (de més a menys)

| **Operador** | **Descripció** |
|---|---|
| [ ] | Selecció de bit (o conjunt de bits). |
| ( ) | Parentesis |
| + - | Operadors unaris |
| ! ~ | Negació lògica i bit-wise |
| & \| ~& ~\| ^ ~^ ^~ | Operadors de reducció |
| { } | Concatenació |
| * / % | Operadors aritmètics |
| + - | Operadors aritmètics |
| << >> | Desplaçaments |
| > >= < <= | Operadors relacionals |
| == != | Igualtats lògiques |
| & | AND bit-wise |
| ^ ^~ ~^ | XOR i XNOR bit-wise |
| \| | OR bit-wise |
| && | AND lògica |
| \|\| | OR lògica |
| ? : | Condicional |

## Operands

En qualsevol expressió es poden fer servir els següents operands:

- Números
- Dades *wire* o *reg*
    - Bit-select (selecció d'un bit a un vector)
    - Part-select (selecció de conjunt de bits d'un vector)

# Estructures de control

## Estructura begin-end

Permet agrupar un conjunt de declaracions en una sola. És necessari per poder fer més d'una declaració dins altres estructures de control (com un if o dins d'una funció, per exemple).
S'escriu de la següent forma:

```
begin
        <declaracions>
end
```

Verilog permet crear i usar estructures begin-end amb nom, nosaltres no les permetem fer servir en el nostre llenguatge.

## if-else

Executa una instrucció o bloc d'instruccions depenent del resultat d'una o varies expressions. L'estructura que segueix és:

```
if (<expr>)
        <declaració o bloc begin-end>
else if (<expr>)
        <declaració o bloc begin-end>
<més else if>
...
else
        <declaració o bloc begin-end>
```

Es poden concatenar tants *else if* com es vulgui i tant els *else if* com el *else* final són opcionals.

**Exemple:**

```
if (reset == 1'b1)
        cout1 = a & b;
else if (reset == 1'b0 && enable == 1'b1)
        begin
                cout1 = a & b & c;
                cout2 = b - c;
        end
else cout1 = 1'b0;
```

### case

Semblant al if-else. Té un conjunt de casos associats a una expressió, i executa el conjunt d'instruccions que segueix el cas que ocorri. La sintaxi és la següent:

```
case (<expr>)
        <expr> :
                <bloc begin-end>
        <expr>, <expr> :
                <bloc begin-end>
        <més casos>
        …
        default :
                <bloc begin-end>
endcase
```

Un cas pot tenir més d'un resultat associat. Si qualsevol dels resultats associats a un cas es cert s'executa el bloc d'instruccions associat.
El cas "default" és opcional i, en cas de ser-hi, s'executara si cap dels casos anteriors ha passat.

### Bucle for

Executa una instrucció (o bloc d'instuccions, si usem un *begin-end*) mentre una variable que es va incrementant (o decrementant) compleixi una condició.
Hi ha quatre maneres de declarar un bucle for:

```
for (index = low_range; index < high_range; index = index + step)
for (index = high_range; index < low_range; index = index - step)
for (index = low_range; index <= high_range; index = index + step)
for (index = high_range; index <= low_range; index = index - step)
```

**Exemples**:

```
for( i = 0; i < 8; i = i + 1 )
        example[i] = a[i] & b[7-i];

for( i = 0; i < 8; i = i + 1 )
        for( j = 0; j < 8; j = j + 2 )
                example[i + j] = a[i] & b[7-j] & example[i + j];
```

## Funcions

Una funció en Verilog ens permet declarar estructures per poder utilitzar-les repetides vegades dins un mòdul. Han de ser declarades dins del mòdul i només és podran fer servir en el mòdul on han estat declarades.
Segueixen la següent estructura:

```
function [rang] <nom funció>;
        <inputs>
        <nodes interns>
        <declaració o bloc begin-end>
endfunction
```

El rang és declara de la mateixa forma que un vector, és opcional (en cas de no possar-se serà un bit), i ens indicarà la mida de l'output de la funció.
Si hi ha més d'una expressió, s'hauran de possar dins un bloc begin-end.
Per assignar valors a l'output es farà servir el nom de la funció (com ho faries amb una variable output qualsevol).

**Exemple:**

```
function [7:0] adder;
        input [7:0] a, b;
        reg c;
        begin
                c = 0;
                for (i = 0; i <= 7; i = i + 1) begin
                        adder[i] = a[i] ^ b[i] ^ c;
                        c = a[i] & b[i] | a[i] & c | b[i] & c;
                end
        end
endfunction
```

## Crides a funcions

Per cridar a la funció es fara d'aquesta manera:

```
<nom funció>( <paràmetres> )
```

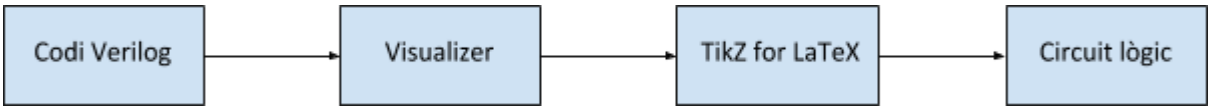
On els paràmetres són els inputs definits a la funció (en l'ordre que s'han definit).
És pot cridar la funció a qualsevol lloc on pugui anar una expressió.

**Exemple:**

```
wire a;
assign a = &(c ^ adder(x, y) & d);
```

# Arquitectura

El nostre projecte serà un compilador, ja que el que farem és traduïr el codi de Verilog a LaTeX (concretament farem servir el paquet de TikZ) de manera que aquest pugui generar-ne el circuit lògic corresponent.




Per a generar la gramàtica i poder interpretar-la farem servir antlr3.

Com a software addicional farem servir la llibreria de circuits lògics del ja esmentat TikZ,que ens servirà per a generar l'output.