



Sistemas Combinacionales y mas Verilog

Diseño de Sistemas con FPGA

Patricia Borensztein

Verilog para síntesis

Circuitos Combinacionales a nivel RT

Tipos de Datos en Verilog

- Dos tipos de datos:
 - Net: representan las conexiones físicas entre componentes hardware
 - wire:
 - Y otros tipos que no usaremos (wand, supply0, ...)
 - Variable: representan almacenamiento abstracto en los módulos “de comportamiento”
 - reg
 - integer
 - Real, time, realtime: solamente en simulación

Representación de los números

- Una constante entera se puede representar en varios formatos. Su forma general es:
`[s i g n][s i z e]' [b a s e][v a l u e]`
 - [base] especifica la base de numeración: (si el número es decimal, se puede omitir la base)
 - b or B: binary
 - o or O: octal
 - h or **H**: hexadecimal
 - d or D: decimal
- [value] es el valor del número en la base especificada.
- [size] es el número de bits del número. Es opcional

Operaciones

Table 3.1 Verilog operators

Type of operation	Operator symbol	Description	Number of operands
Arithmetic	+	addition	2
	-	subtraction	2
	*	multiplication	2
	/	division	2
	%	modulus	2
	**	exponentiation	2
Shift	>>	logical right shift	2
	<<	logical left shift	2
	>>>	arithmetic right shift	2
	<<<	logical left shift	2
Relational	>	greater than	2
	<	less than	2
	>=	greater than or equal to	2
	<=	less than or equal to	2
Equality	==	equality	2
	!=	inequality	2
	===	case equality	2
	!==	case inequality	2
Bitwise	~	bitwise negation	1
	&	bitwise and	2
		bitwise or	2
	^	bitwise xor	2
Reduction	&	reduction and	1
		reduction or	1
	^	reduction xor	1
Logical	!	logical negation	1
	&&	logical and	2
		logical or	2
Concatenation	{ }	concatenation	any
	{ { } }	replication	any
Conditional	? :	conditional	3

Aritméticos

- Operadores '+' y '-' :
 - Infieren circuitos sumadores y restadores
- Operadores '*':
 - En la familia Spartan3 existen bloques multiplicadores, por tanto XST infiere estos circuitos hardware embebidos. Solo hay un número limitado (18 de 18 bits, varían según el modelo)
- Operadores '/', '%', '**' :
 - no sintetizan automáticamente.

Shift

- Lógicos (‘>>’ ‘<<’) y Aritméticos (‘>>>’ ‘<<<’)
 - En los lógicos, entra un ‘0’ por la derecha o por la izquierda
 - En los aritméticos, el MSB cuando es a la derecha, y un ‘0’ a la izquierda
- La sentencia :
assign q= a << b
infiere un circuito : barrel shifter (luego lo haremos)
- La sentencia:
assign q= a<<“cte”
solo rutea

Relacionales, Bitwise (nivel de bit), Reducción

- Operadores ' $<$ ', ' $>$ ', ' \leq ', ' \geq ', ' $=$ ', ' \neq ' :
 - Infieren comparadores
- Operadores Bitwise ' $\&$ ', ' $|$ ', ' \wedge (xor)', ' \sim (not)'
 - Se sintetizan por medio de LUT's
- Operadores de Reducción:

```
wire [3:0] a;  
wire y;
```

```
assign y = | a; // only one operand
```

es lo mismo que:

```
assign y = a[3] | a[2] | a[1] | a[0];
```

Lógicos

- Los operadores lógicos : ' && ', ' !! ', ' ! ', devuelven el valor lógico (un bit) falso o true. Se usan en las expresiones booleanas. (Descripciones de comportamiento)

Table 3.4 Logical and bitwise operation examples

a	b	a&b	a b	a&& b	a b
0	1	0	1	0 (false)	1 (true)
000	000	000	000	0 (false)	0 (false)
000	001	000	001	0 (false)	1 (true)
011	001	001	011	1 (true)	1 (true)

Concatenación

- El operador `{}` se implementa con ruteo

```
wire a1;  
wire [3:0] a4;  
wire [7:0] b8, c8, d8;  
.  
.  
.  
assign b8 = {a4, a4};  
assign c8 = {a1, a1, a4, 2'b00};  
assign d8 = {b8[3:0], c8[3:0]};
```

este es un ejemplo

```
wire [7:0] a;  
wire [7:0] rot, shl, sha;  
  
// rotate a to right 3 bits  
assign rot = {a[2:0], a[8:3]};  
// shift a to right 3 bits and insert 0 (logic shift)  
assign shl = {3'b000, a[8:3]};  
// shift a to right 3 bits and insert MSB  
// (arithmetic shift)  
assign sha = {a[8], a[8], a[8], a[8:3]};
```

Operadores Condicionales

- Infieren multiplexores (luego veremos detalle)

```
assign max = (a>b) ? a : b;
```

Pueden usarse en cascada o anidados

```
assign eq = (~i1 & ~i0) ? 1'b1 :  
            (~i1 & i0)   ? 1'b0 :  
            (i1 & ~i0)   ? 1'b0 :  
            1'b1;
```

```
assign max = (a>b) ? ((a>c) ? a : c) :  
              ((b>c) ? b : c);
```

Precedencia de operadores

Table 3.2 Operator precedence

Operator	Precedence
! ~ + - (unary)	highest
**	
* / %	
+ - (binary)	
>> << >>> <<<	
< <= > >=	
== != === !==	
&	
~	
&&	
?:	lowest

Ajustes de longitud en las expresiones

- Verilog ajusta las longitudes usando reglas implícitas:
 - Determina el operando que tiene la máxima longitud (de ambos lados de la expresión)
 - Extiende los operandos del lado derecho, opera
 - Asigna el resultado a la señal del lado izquierdo, truncando si es necesario

Ajustes de longitud en las expresiones

- Ejemplos:

```
wire [7:0] a, b;  
  
assign a = 8'b00000000;  
assign b = 0;
```

- La segunda asignación es un entero de 32 bits, truncado a 8 bits

```
wire [7:0] a, b;  
wire [7:0] sum8;  
wire [8:0] sum9;  
  
assign sum8 = a + b;  
assign sum9 = a + b;
```

- La primera asignación es una suma en 8 bits. El carry out se descarta.
- La segunda asignación
Verilog extiende los operandos a 9 bits, y la suma es de 9 bits. (esto en VHDL daría error)

Ajustes de longitud en las expresiones

- Ejemplos:

sum1, a y b son de 8 bits

```
// shift 0 to MSB of sum1  
assign sum1 = (a + b) >> 1;  
// shift carry-out of a+b to MSB of sum2  
assign sum2 = (0 + a + b) >> 1;
```

En la primera asignación, se descarta el carry out de la suma.

En la segunda asignación, Verilog extiende a 32 bits los operandos porque hay un entero en ella, así que no se descarta el carry out que queda en el sum[7] luego de hacer el shift

<i>Consejo: No dejar que Verilog ajuste por sus reglas!!!!</i>
--

Síntesis de z (alta impedancia)

- El valor 'Z' significa alta impedancia y no es un valor que pueda ser asignado como el '0' lógico o '1' lógico sino que solo puede ser sintetizado mediante un buffer de tres estados (tri-state buffer)



oe	y
0	Z
1	a_in

```
assign y = (oe) ? a_in : 1'bz;
```

Síntesis de 'z'

- ¿Para que podemos necesitar un buffer tri-state? Para implementar un port bidireccional.

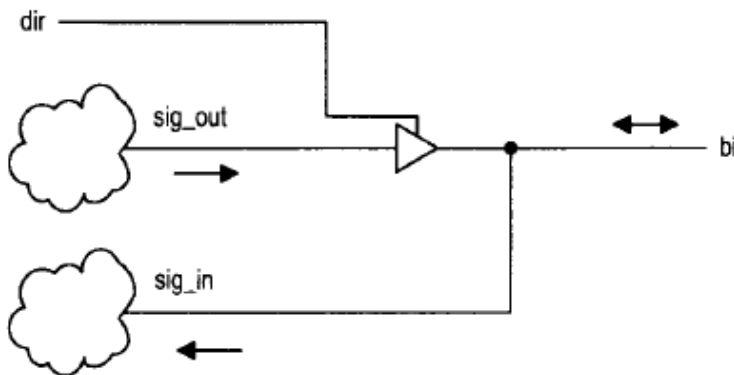


Figure 3.2 Single-buffer bidirectional I/O port.

```
module bi_demo(  
    inout wire bi,  
    . . .  
  
    assign sig_out = output_expression;  
  
    . . .  
    assign some_signal = expression_with_sig_in;  
    . . .  
    assign bi = (dir) ? sig_out : 1'bz;  
    assign sig_in = bi;  
    . . .  
);
```

- En Spartan3, los tri-state buffers solo existen en los IOB's (input output blocks) de un pin físico.

Síntesis de 'x'

- Se utiliza para denotar valores o combinaciones de valores de las entradas que no se darían nunca en la realidad. Ejemplo:

Table 3.5 Truth table with don't-care

input	output
i	y
0 0	0
0 1	1
1 0	1
1 1	x

- Al simular el circuito, la respuesta a una entrada '11' sería 'x'.
- En síntesis, el circuito real, frente a una entrada '11' daría o '0' o '1'.
- Discrepancia entre síntesis y simulación.

Always Block, Initial Block (para circuitos combinacionales)

- Son construcciones de los lenguajes que encapsulan “sentencias de procedimiento”, que son aquellas que se ejecutan secuencialmente.
- La construcción “Initial” block no sintetiza y solo se usa en simulación.
- Estos bloques pueden verse como cajas negras que describen comportamientos usando las sentencias de procedimientos.

Sintaxis del always block

- Lista de sensibilidad:
 - debe contener todas las entradas al que responde el bloque

```
always @([sensitivity_list])  
begin [optional name]  
    [optional local variable declaration];  
  
    [procedural statement];  
    [procedural statement];  
    . . .  
end
```

Always block

- Asignación Procedural:
 - Hay de dos tipos: las bloqueantes y las no bloqueantes.
 - Bloqueantes: las usaremos en los circuitos combinacionales. El funcionamiento es el siguiente: se evalúa la expresión a ser asignada a la variable, y se asigna inmediatamente, antes de la ejecución de la siguiente sentencia
 - No bloqueantes: la expresión se evalúa pero se asigna al final del always block. Lo usaremos en los secuenciales.

```
[variable_name] = [expression];    // blocking assignment  
[variable_name] <= [expression];   // nonblocking assignment
```

Always Block

- Tipos de Datos : Variable
 - En una asignación procedural, las salidas solo pueden ser asignadas a variables cuyos tipos pueden ser: *reg*, *integer*, *real*

Listing 3.1 Always block implementation of a 1-bit comparator

```
module eq1_always
(
    input wire i0, i1,
    output reg eq // eq declared as reg
5 );

    // p0 and p1 declared as reg
    reg p0, p1;

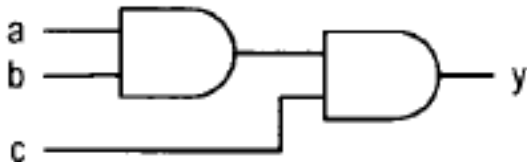
10 always @(i0, i1) // i0 and i1 must be in sensitivity list
begin
    // the order of statements is important
    p0 = ~i0 & ~i1;
    p1 = i0 & i1;
15 eq = p0 | p1;
end

endmodule
```

Continuos vs Procedural Assignments

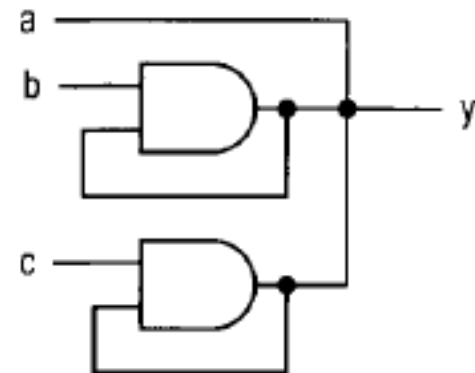
```
module and_block_assign
(
  input wire a, b, c,
  output reg y
5 );

  always @*
  begin
    y = a;
10    y = y & b;
    y = y & c;
  end
endmodule
```



```
module and_cont_assign
(
  input wire a, b, c,
  output wire y
3 );

  assign y = a;
  assign y = y & b;
  assign y = y & c;
10
endmodule
```



Sentencia “If”

```
if [boolean_expr]
  begin
    [procedural statement];
    [procedural statement];
    . . .
  end
else
  begin
    [procedural statement];
    [procedural statement];
    . . .
  end
```

```
if [boolean_expr_1]
  . . .
else if [boolean_expr_2]
  . . .
else if [boolean_expr_3]
  . . .
else
  . . .
```

- Es una sentencia procedural que SOLO puede ser utilizada dentro de un always block.
- Al sintetizar, veremos, genera una estructura de ruteo.

Sentencia IF:

Codificador con prioridad

Listing 3.4 Priority encoder using an if statement

```
module prio_encoder_if
(
    input wire [4:1] r,
    output reg [2:0] y
5 );

always @*
    if (r[4]==1'b1)           // can be written as (r[4])
        y = 3'b100;
    else if (r[3]==1'b1)      // can be written as (r[3])
10        y = 3'b011;
    else if (r[2]==1'b1)      // can be written as (r[2])
        y = 3'b010;
    else if (r[1]==1'b1)      // can be written as (r[1])
15        y = 3'b001;
    else
        y = 3'b000;

endmodule
```

input r	output pcode
1---	100
01--	011
001-	010
0001	001
0000	000

Sentencia IF: Decodificador

Listing 3.5 Binary decoder using an if statement

```
module decoder_2_4_if
(
    input wire [1:0] a,
    input wire en,
5    output reg [3:0] y
);

always @*
    if (en==1'b0)           // can be written as (~en)
10        y = 4'b0000;
    else if (a==2'b00)
        y = 4'b0001;
    else if (a==2'b01)
        y = 4'b0010;
15    else if (a==2'b10)
        y = 4'b0100;
    else
        y = 4'b1000;

20 endmodule
```

Truth table of a 2-to-4 decoder with enable

en	input		output
	a(1)	a(0)	y
0	—	—	0000
1	0	0	0001
1	0	1	0010
1	1	0	0100
1	1	1	1000

Sentencia “Case”

```
case [case_expr]
  [item]:
    begin
      [procedural statement];
      [procedural statement];
      . . .
    end
  [item]:
    begin
      [procedural statement];
      [procedural statement];
      . . .
    end
  [item]:
    begin
      [procedural statement];
      [procedural statement];
      . . .
    end
  . . .
default:
  begin
    [procedural statement];
    [procedural statement];
    . . .
  end
endcase
```

- En Verilog no es necesario incluir todos los posibles valores, y mas de uno puede hacer match...
- Si mas de una expresión hace match, entonces se ejecuta la primera de ellos.
- Si estan explicitadas todas las posibles combinaciones → Full Case
- Si son mutuamente excluyentes → Parallel Case
- Si no son mutuamente excluyentes → Non Parallel Case

Ejemplo Case. Decoder

Listing 3.6 Binary decoder using a case statement

```
module decoder_2_4_case
(
    input wire [1:0] a,
    input wire en,
5    output reg [3:0] y
);

    always @*
        case({en,a})
10        3'b000, 3'b001, 3'b010, 3'b011: y = 4'b0000;
           3'b100: y = 4'b0001;
           3'b101: y = 4'b0010;
           3'b110: y = 4'b0100;
           3'b111: y = 4'b1000; // default can also be used
15        endcase

endmodule
```

Todas las expresiones del case están cubiertas: FULL CASE.
Como además, son mutuamente excluyentes: PARALLEL CASE

Variantes: Casez y Casex

- En el Casez, 'z' y '?' son tratados como "no importa".
- En el casex, tanto 'z' como 'x' como '?' son tratados como no importa.
- Mejor usar solo '?'

Listing 3.8 Priority encoder using a casez statement

```
module prio_encoder_casez
(
    input wire [4:1] r,
    output reg [2:0] y
5   );

    always @*
        casez(r)
            4'b1??? : y = 3'b100;
            4'b01?? : y = 3'b011;
            4'b001? : y = 3'b010;
            4'b0001 : y = 3'b001;
            4'b0000 : y = 3'b000; // default can also be used
            endcase
15
endmodule
```

- Este Case es Full Case y Parallel

Non parallel Case and Non Full Case

- En el ejemplo, faltan explicitar las combinaciones : 001,010,011 . Por lo tanto la salida **y**, en esos casos, mantendrá el valor anterior.
- La combinación 111 y 1?? No son excluyentes, pero si se da 111, y valdrá '1' pues es la primera del case.

```
reg [2:0] s
. . .
casez (s)
    3'b111: y = 1'b1;
    3'b1??: y = 1'b0;
    3'b000: y = 1'b1;
endcase
```

- Algo muy importante:

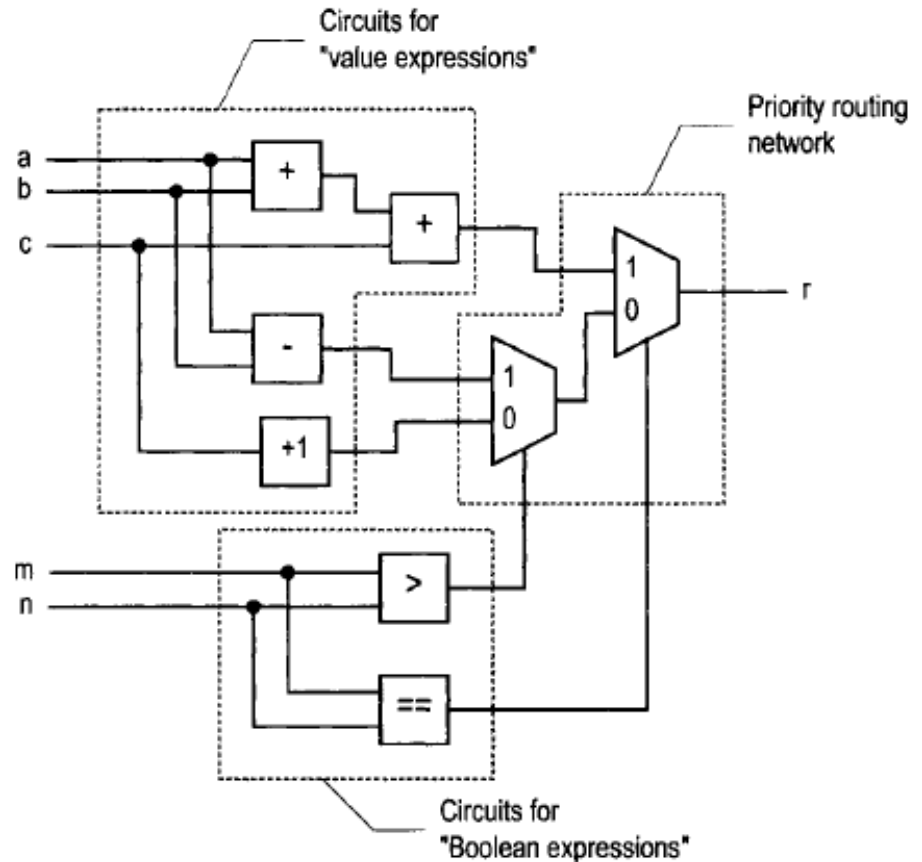
EN LOS CIRCUITOS COMBINACIONALES, EL CASE DEBE SER FULL CASE pues todas las combinaciones de las entradas deben tener su salida. Usar default para las combinaciones no especificadas.

Estructuras de Ruteo

- Las sentencias condicionales se sintetizan con routing networks (redes de ruteo)
- Hay dos tipos:
 - Priority Routing Network (if then else)
 - Multiplexing Network (parallel case)

Priority Network

```
if (m==n)
    r = a + b + c;
else if (m > n)
    r = a - b;
else
    r = c + 1;
```



(b) Diagram of an if statement

Figure 3.4 Implementation of an if statement.

- Todas las expresiones booleanas y los valores de las expresiones se evalúan concurrentemente .
- Cuantas mas cláusulas else hay, mas aumenta el tiempo de propagación a través de los multiplexores.
- El ruteo se produce a través de una cascada de multiplexores 2 a 1 encadenados por prioridad.

Priority Network

- El operador condicional '?' también infiere una red de multiplexores 2:1.
- El case no paralelo, también.

```
case (expr)
  item1: statement1;
  item2: statement2;
  item3: statement3;
  default: statement4;
endcase
```

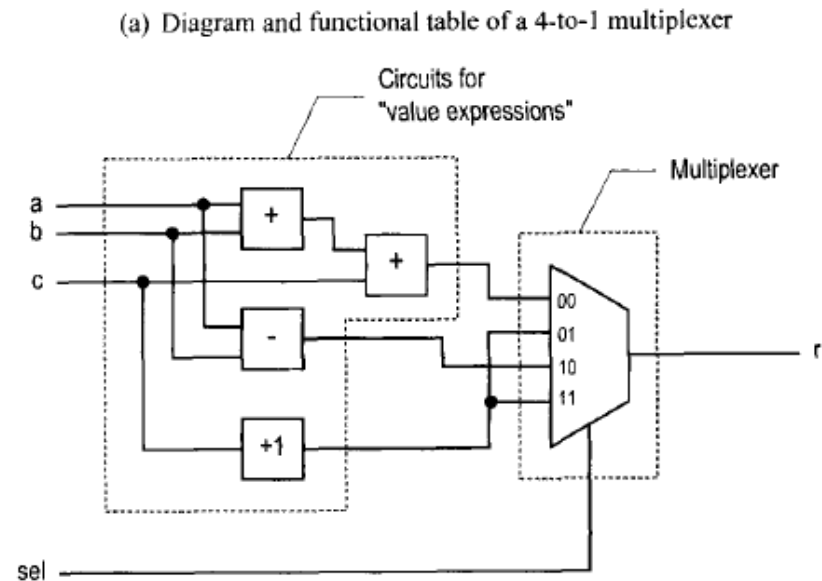


```
if [expr==item1]
  statement1;
else if [expr==item2]
  statement2;
else if [expr==item3]
  statement3;
else
  statement4;
```

Multiplexing Network

- Los case paralelos infieren estos ruteos:

```
wire [1:0] sel;  
.  
.  
.  
case (sel)  
  2'b00: r = a + b + c;  
  2'b10: r = a - b;  
  default: r = c + 1; // 2'b01, 2'b11  
endcase
```



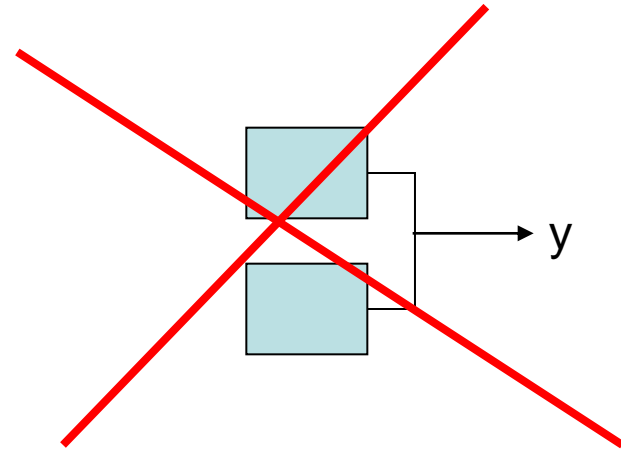
- Las priority networks son adecuadas cuando se la da prioridad a algunas condiciones como en el caso del codificador con prioridad.

Always Block: errores comunes

- Mismas variables que se usan en múltiples always blocks

```
reg y;  
reg a, b, clear;  
...  
always @*  
    if (clear) y = 1'b0;  
  
always @*  
    y = a & b;
```

El código es correcto sintácticamente, pero no sintetiza porque no existe un circuito que se comporte de esa manera.



Always Block

- Se debería hacer de esta manera:

```
always @*  
    if (clear)  
        y = 1'b0;  
    else  
        y = a & b;
```

Always Block: errores comunes

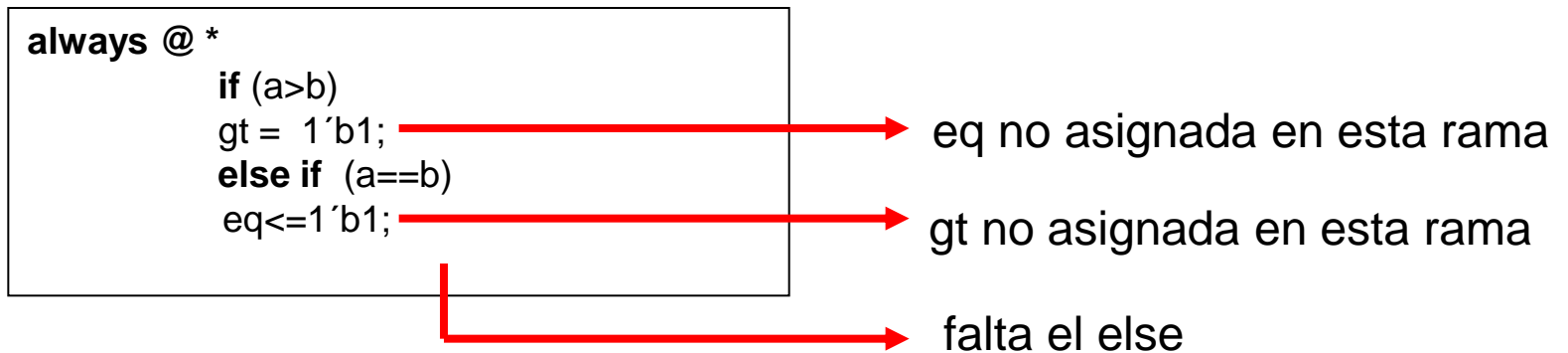
- Lista de sensibilidad incompleta

```
always @(a)
    y = a & b;
```

- Quiere decir que cuando cambia a, se activa el bloque... pero cuando cambia 'b' el bloque no se activa, es decir que 'y' guarda su valor anterior → no existe un circuito con esta conducta. (*ojo! Existe, pero implica memoria, ya lo veremos*)
- El sintetizador puede inferir la puerta AND igualmente. Pero habría discrepancia entre simulación y síntesis.
- Solución: usar siempre la construcción **always @***

Memoria no intencionada

- Verilog establece que una señal no asignada dentro de un always block, mantendrá su valor. Para esto, durante el proceso de síntesis, se infiere un latch o un estado interno. Para evitar esto, se debe:
 - Incluir todas las señales en la lista de sensibilidad
 - Incluir la rama else
 - Asignar valores a las señales en cada rama del condicional



Solución a la memoria no intencionada

```
always @*  
    if (a > b)  
        gt = 1'b1;  
    else if (a == b)  
        eq = 1'b1;
```



```
always @*  
    if (a > b)  
        begin  
            gt = 1'b1;  
            eq = 1'b0;  
        end  
    else if (a == b)  
        begin  
            gt = 1'b0;  
            eq = 1'b1;  
        end  
    else // i.e., a < b  
        begin  
            gt = 1'b0;  
            eq = 1'b0;  
        end  
end
```



```
always @*  
begin  
    gt = 1'b0; // default value for gt  
    eq = 1'b0; // default value for eq  
    if (a > b)  
        gt = 1'b1;  
    else if (a == b)  
        eq = 1'b1;  
end
```


Constantes

- Sumador de 4 bits
- Sumador de N bits

```
module adder_carry_hard_lit
(
  input wire [3:0] a, b,
  output wire [3:0] sum,
5  output wire cout // carry-out
);

// signal declaration
wire [4:0] sum_ext;
10

//body
assign sum_ext = {1'b0, a} + {1'b0, b};
assign sum = sum_ext[3:0];
assign cout= sum_ext[4];
15

endmodule
```

```
module adder_carry_local_par
(
  input wire [3:0] a, b,
  output wire [3:0] sum,
5  output wire cout // carry-out
);

// constant declaration
localparam N = 4,
10          N1 = N-1;

// signal declaration
wire [N:0] sum_ext;

//body
15 assign sum_ext = {1'b0, a} + {1'b0, b};
assign sum = sum_ext[N1:0];
assign cout= sum_ext[N];

20 endmodule
```

Parameter

- La construcción parameter se utiliza para pasar información a una entidad o componente.

```
module adder_carry_para
  #(parameter N=4)
  (
    input wire [N-1:0] a, b,
5    output wire [N-1:0] sum,
    output wire cout // carry-out
  );

  // constant declaration
10  localparam N1 = N-1;

  // signal declaration
  wire [N:0] sum_ext;

15  // body
  assign sum_ext = {1'b0, a} + {1'b0, b};
  assign sum = sum_ext[N1:0];
  assign cout = sum_ext[N];

20 endmodule
```

Instanciación (Parameter)

```
module adder_insta
(
    input wire [3:0] a4, b4,
    output wire [3:0] sum4,
5    output wire c4,
    input wire [7:0] a8, b8,
    output wire [7:0] sum8,
    output wire c8
);

10    // instantiate 8-bit adder
    adder_carry_para #(.N(8)) unit1
        (.a(a8), .b(b8), .sum(sum8), .cout(c8));

15    // instantiate 4-bit adder
    adder_carry_para unit2
        (.a(a4), .b(b4), .sum(sum4), .cout(c4));

endmodule
```

Práctica:

Combinacionales

- Realizar un sumador de números de 4 bits representados en signo y módulo. Se usará la placa spartan3. (X3S200 FT256)
- Los números se asignarán a la entrada a través de los switches: sw0..3 y sw4..7
- Los botones bt0 y btn1 seleccionarán lo que se visualiza en el LED:
 - 00: operando a
 - 01: operando b
 - 10,11: suma
- Usar los módulos ya prediseñados:
 - hex_to_sseg: que genera los patterns adecuados para visualizar el dígito en el LED de 7 segmentos
 - disp_mux: que multiplexa en dos LEDs uno que muestra el módulo y el otro el signo.
- Lo que hay que hacer es el módulo sumador y armar todo el circuito completo.

Sumador Signo y Magnitud

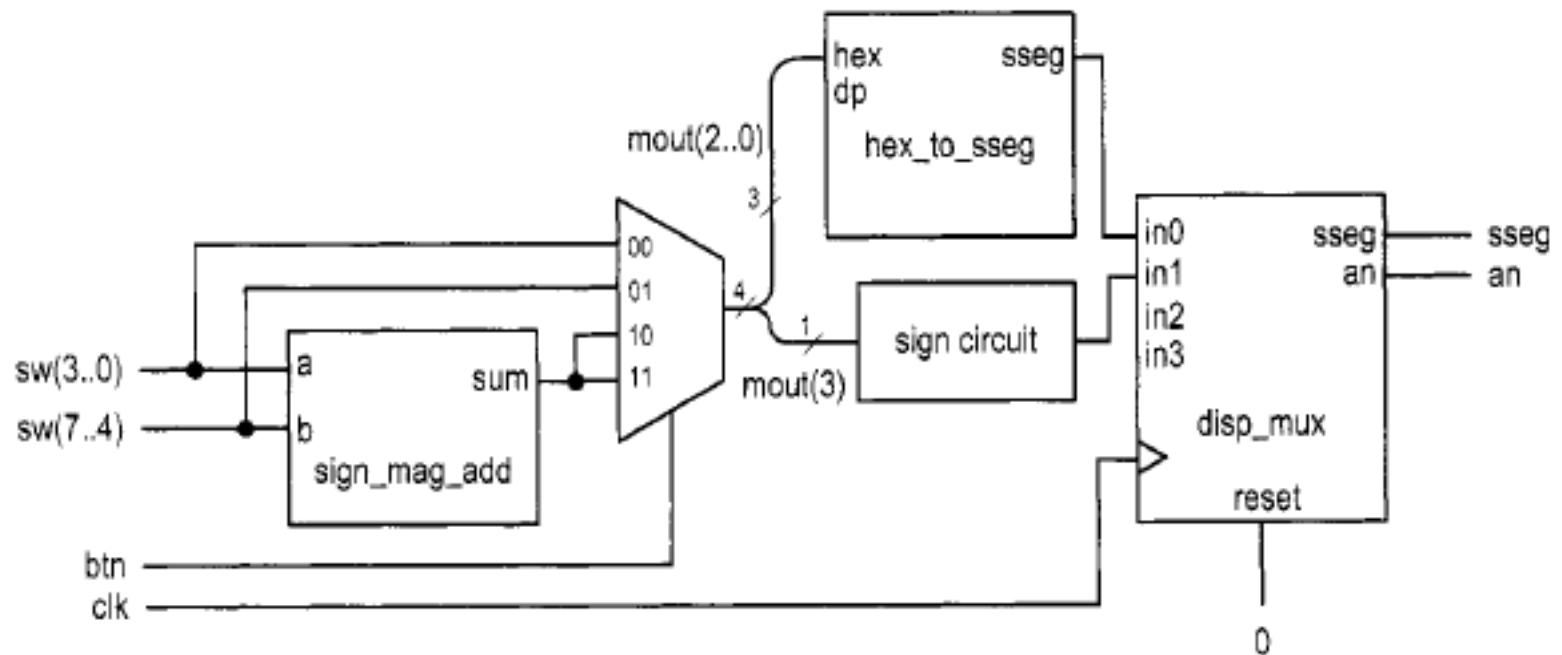
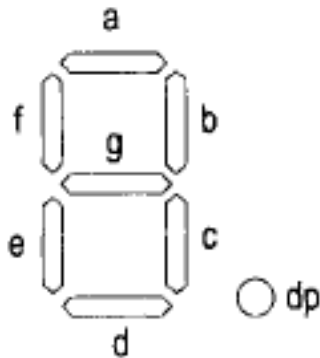


Figure 3.7 Sign-magnitude adder testing circuit.

Módulo Hex to Sseg



Se habilita con 0.

Los segmentos están ordenados:
abcdefg

```

module hex_to_sseg
(
    input wire [3:0] hex,
    input wire dp,
    output reg [7:0] sseg // output active low
);

always @*
begin
    case(hex)
        4'h0: sseg[6:0] = 7'b00000001;
        4'h1: sseg[6:0] = 7'b10011111;
        4'h2: sseg[6:0] = 7'b0010010;
        4'h3: sseg[6:0] = 7'b0000110;
        4'h4: sseg[6:0] = 7'b1001100;
        4'h5: sseg[6:0] = 7'b0100100;
        4'h6: sseg[6:0] = 7'b0100000;
        4'h7: sseg[6:0] = 7'b0001111;
        4'h8: sseg[6:0] = 7'b0000000;
        4'h9: sseg[6:0] = 7'b0000100;
        4'ha: sseg[6:0] = 7'b0001000;
        4'hb: sseg[6:0] = 7'b1100000;
        4'hc: sseg[6:0] = 7'b0110001;
        4'hd: sseg[6:0] = 7'b1000010;
        4'he: sseg[6:0] = 7'b0110000;
        default: sseg[6:0] = 7'b0111000; // 4'hf
    endcase
    sseg[7] = dp;

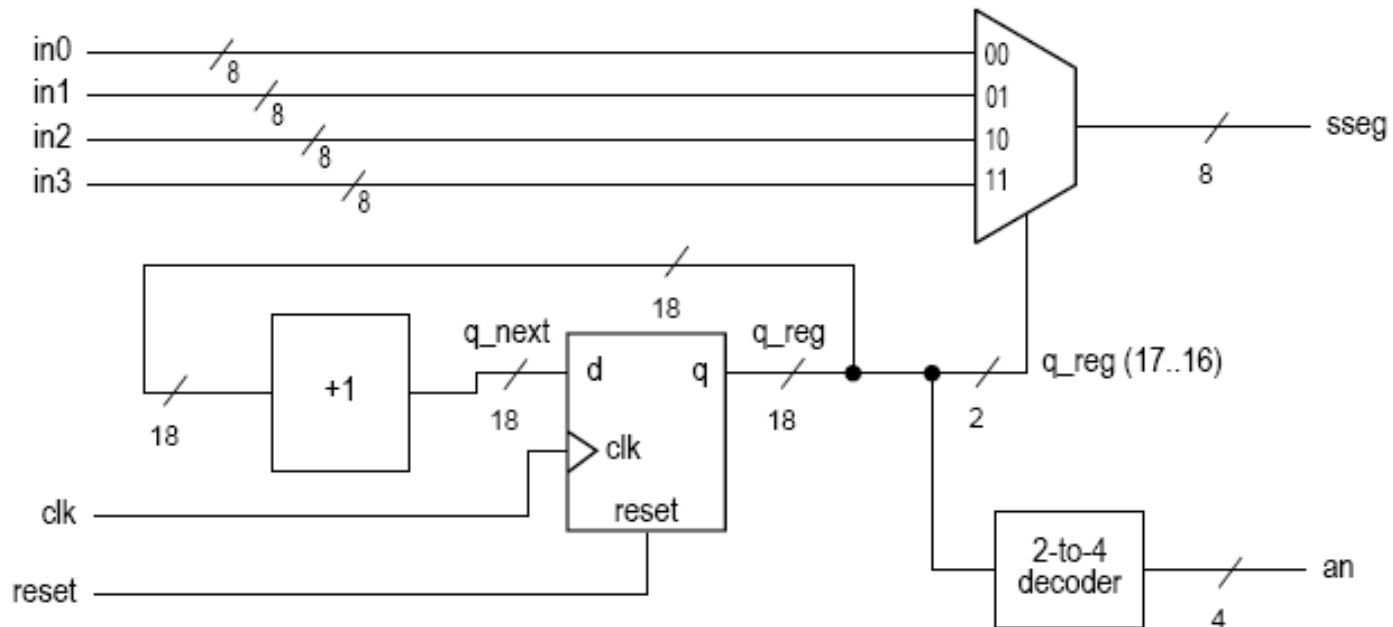
```

Módulo disp_mux

- Para reducir el número de patitas de E/S, los cuatro displays de 7 segmentos comparten las 8 señales para iluminar los segmentos.
- Para poder iluminar los LEDs se necesita un circuito que multiplexe las señales en el tiempo, y cuya velocidad de refresco sea suficientemente alta como para que el ojo humano no perciba la multiplexación.
- El módulo disp_mux está basado en un contador módulo 2^{18} . Los dos bits mas altos del contador se usan para habilitar cada uno de los LEDs. (es decir, 00 habilitan LED0, 01 habilitan LED1, etc)
- Como el reloj de la FPGA funciona a 50 MHz, la frecuencia de refresco de cada LED es de $50/2^{16}$ Mhz aprox. 800 Hz.

Módulo disp_mux

- Es un circuito secuencial, que genera una habilitación (señal an) para cada uno de los LEDS cada 800 Hz.



(b) Block diagram