2. Introducción a Verilog

2.01

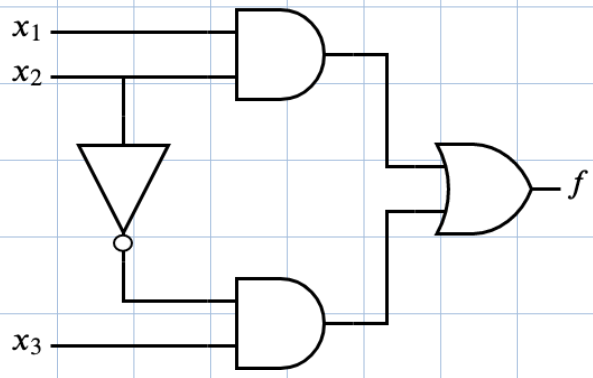
Primeros pasos con Verilog

Verilog

- Fue producido en los 80’s
- De dominio público desde 1990
- Originalmente diseñado para simulación y verificación
  - Luego se le agregaron capacidades de síntesis
- Verilog es complicado
  - Pero es posible hacer muchísimo con lo básico
- Verilog es texto plano y tiene un parecido con el lenguaje C
- Trataremos de aprender Verilog aplicando lo visto en el curso y lo usaremos para describir circuitos digitales
- Hay 2 estilos principales para describir hardware:
  - Estructural
  - De comportamiento

Especificación estructural de circuitos lógicos

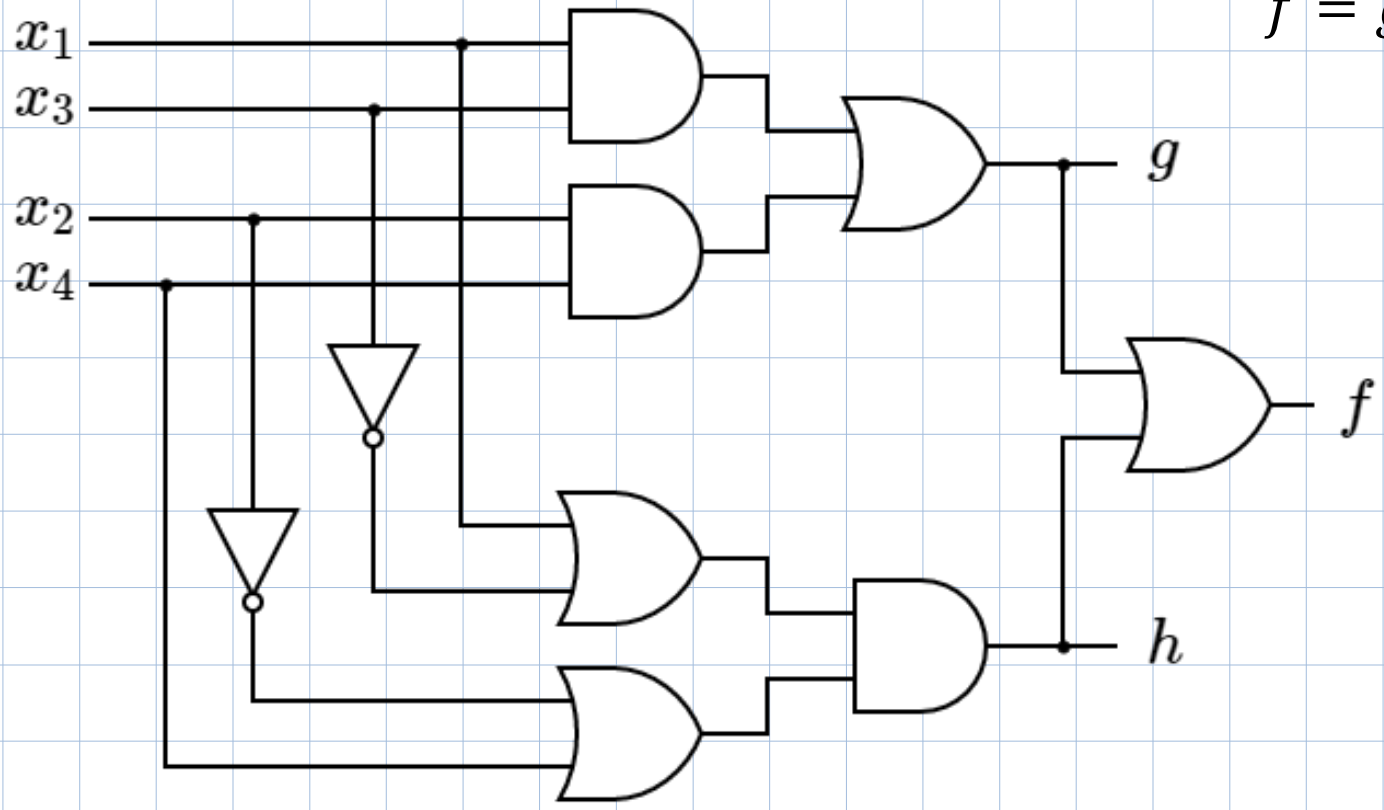
- Especificamos directamente la estructura del circuito – conexiones entre módulos
- Podemos usar las primitivas de Verilog, por ejemplo **and**, **not**, **or**, **nand** y **nor**
- Los circuitos son modulares (descritos en módulos)
- Cada módulo tiene ports de entrada y salida
  - Son declarados al principio
- El módulo finaliza con **endmodule**
- Podemos **instanciar** este mismo módulo muchas veces en un sistema digital jerárquico




```
1  module example1(x1,x2,x3,f);
2    input x1, x2, x3;
3    output f;
4
5    and(g, x1, x2);
6    not(k, x2);
7    and(h, k, x3);
8    or(f, g, h);
9
10 endmodule
```

Ejemplo con código Verilog

$$g = x_1x_3 + x_2x_4$$
$$h = (x_1 + \overline{x_3})(\overline{x_2} + x_4)$$
$$f = g + h$$




```
module example2(x1,x2,x3,x4,f,g,h);
  input x1, x2, x3, x4;
  output f, g, h;

  and(z1, x1, x3);
  and(z2, x2, x4);
  or(g, z1, z2);
  or(z3, x1, ~x3);
  or(z4, ~x2, x4);
  and(h, z3, z4);
  or(f, g, h);

endmodule
```

Especificación de comportamiento de circuitos lógicos

- Describimos el comportamiento que queremos que tenga el circuito
  - El compilador se encarga de definir la estructura
- Usamos & para AND, | para OR y ~ para NOT
- Ejemplo: implementemos $f = x_1x_2 + \overline{x_2}x_3$
- La palabra assign realiza una asignación continua y **concurrente** de la salida
  - Como una conexión interna permanente
- La siguiente lámina muestra el ejemplo de la lámina anterior

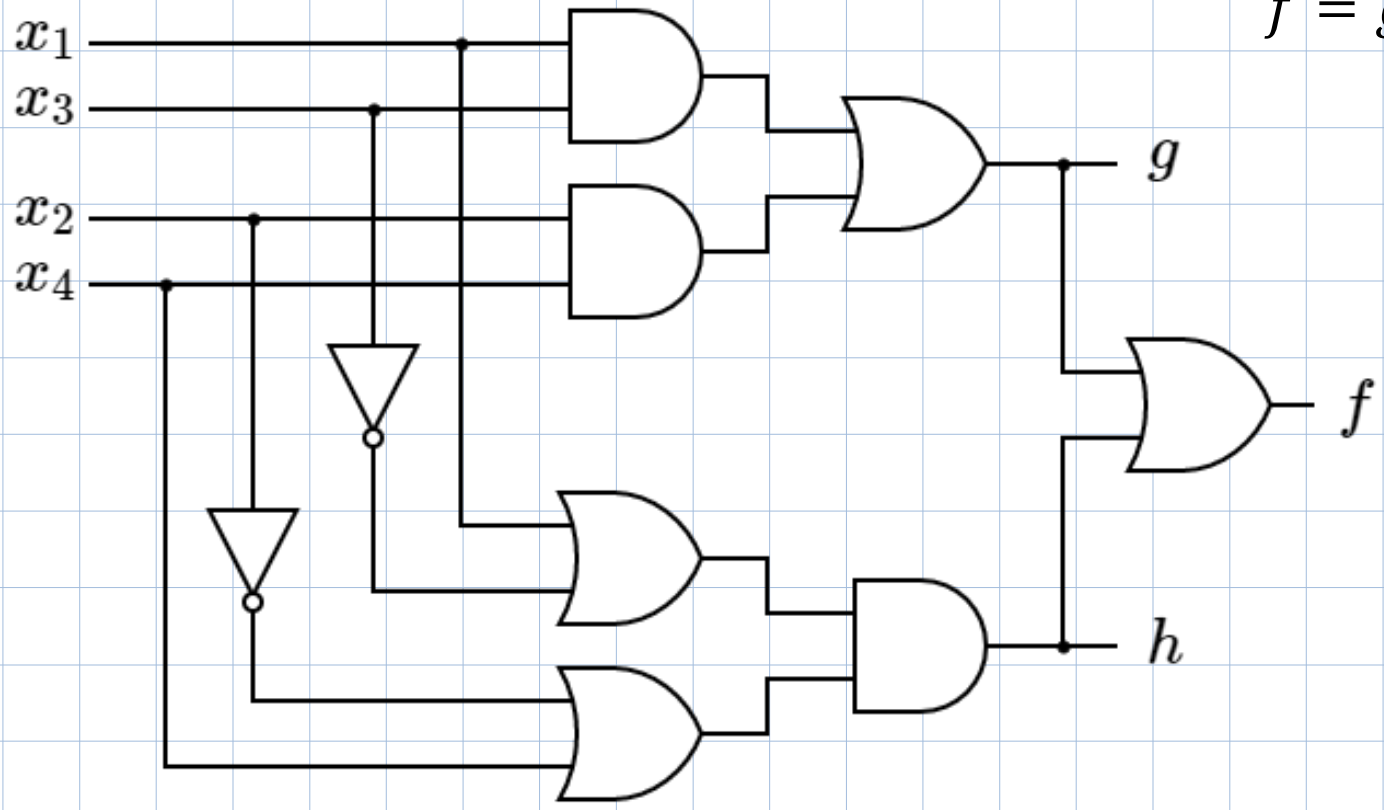
```
module example3(x1,x2,x3,f);
  input x1, x2, x3;
  output f;

  assign f = (x1 & x2)|(~x2 & x3);

endmodule
```

Ejemplo con código Verilog

$$g = x_1x_3 + x_2x_4$$
$$h = (x_1 + \overline{x_3})(\overline{x_2} + x_4)$$
$$f = g + h$$




```
module example4(x1,x2,x3,x4,f,g,h);
  input x1, x2, x3, x4;
  output f, g, h;

  assign g = (x1 & x3)|(x2 & x4);
  assign h = (x1 | ~x3)&(~x2 | x4);
  assign f = g | h;

endmodule
```

Cómo NO ESCRIBIR código Verilog

- Con los procedimientos, nos vemos tentados a escribir código Verilog como si fuera un programa
  - Verilog no es para programar
  - Debemos evitar variables internas que ocuparán más recursos de la FPGA
  - Es difícil predecir cómo será compilado un código escrito estilo programa
- Se sugiere seguir el estilo de los ejemplos del curso
- Regla de oro: si al diseñador no le es fácil determinar que circuito es descrito por el código, entonces es improbable que la síntesis vaya a realizar el circuito que el diseñador está tratando de describir

Jerarquía en Verilog

- En Verilog usamos jerarquía para describir circuitos
- Por ejemplo, **definimos** un modulo
  - El módulo tiene entradas y salidas
  - El módulo realiza una función
- Podemos **instanciar** ese mismo módulo en diferentes partes de nuestro hardware
  - Si cambiamos la definición del módulo, ésta cambia en todas sus instancias
- Esto permite hacer código más portable

¿Qué aprendimos hoy?

- Introducción al lenguaje Verilog
- Especificación estructural
- Especificación de comportamiento
- Jerarquía en Verilog
- Cómo no escribir código Verilog
  - Muchos ejemplos!

2.02

Tipos de datos, conexiones, registros, asignaciones y procedimientos

Tipos de datos en Verilog

- Existen cuatro tipos de datos:
  - `0`: cero lógico
  - `1`: uno lógico
  - `x`: dato desconocido
  - `z`: alta impedancia o “don’t care” (más adelante veremos esto)
- Podemos usarlos de diversas formas - ejemplos:

  `4'b0011` corresponde al número 3 escrito con palabra de 4 bits

  `3'bxz1` corresponde a 3 bits (desconocido, alta impedancia, 1)

  `12'b0010_1101_1100` el guion bajo facilita lectura (separador)

  `8'hA` es el dígito hexadecimal A, que corresponde a 1010 en binario

  `4'd15` es el número decimal 15, es decir, 1111 en binario

  `3'o6` es un octal de 3 bits, de valor 110

Conexiones o wires en Verilog

- Verilog describe hardware
  - Utiliza jerarquía para instanciar módulos
  - Los módulos se comunican mediante conexiones
- Las conexiones en Verilog se denominan **wires**
  - Los wires conectan módulos mediante puertos de entrada y salida
    - Si un wire no va conectado a una salida, su valor es indefinido
  - Cada wire lleva un nombre y una señal
- Los wires no tienen memoria
  - Podemos usarlos en comando assign (bloques estructurales)

```
module example4(x1,x2,x3,x4,f,g,h);
  input x1, x2, x3, x4;
  output f, g, h;

  assign g = (x1 & x3)|(x2 & x4);
  assign h = (x1 | ~x3)&(~x2 | x4);
  assign f = g | h;

endmodule
```

Registros en Verilog

- Con frecuencia necesitamos guardar datos en Verilog
  - Más adelante aprenderemos que los circuitos con memoria corresponden a una categoría especial y muy importante de circuitos digitales
- Para almacenar datos en Verilog usamos registros (**reg**)
  - La sintaxis de registros es muy similar a la de wires
- Los registros son actualizados en bloques **always** (bloques de comportamiento o procedimientos)
  - No podemos usar un comando assign para actualizar registros
  - Tampoco podemos usar el comando assign dentro de un bloque always

Señales vectorizadas

- En Verilog un vector es un arreglo unidimensional de elementos
  - Tanto variables wire como reg pueden ser vectores
- Los vectores se definen indicando el tipo de variable y el número de bits:

<type> [<MSB_index>:<LSB_index>] vector_name

- Este módulo es de un sumador (más adelante) que suma dos vectores de 32 bits (X e Y) con una señal de un bit (carryin) y el resultado lo guarda en un registro de 32 bits (S)

```
1  module addern(carryin,X,Y,S);
2    parameter n = 32;
3    input carryin;
4    input [n-1:0] X, Y;
5    output [n-1:0] S;
6    reg [n-1:0] S;
7
8    always @(X or Y or carryin)
9      S = X + Y + carryin;
10
11 endmodule
```

Asignaciones continuas en Verilog

- Permiten asignar una expresión a un wire
- Corresponden a asignaciones continuas
  - Ocurren constantemente, todo el tiempo
- Ejemplo:

```
module example4(x1,x2,x3,x4,f,g,h);
  input x1, x2, x3, x4;
  output f, g, h;

  assign g = (x1 & x3)|(x2 & x4);
  assign h = (x1 | ~x3)&(~x2 | x4);
  assign f = g | h;

endmodule
```

Procedimientos en Verilog

- Corresponden a bloques que describen lo que ocurre bajo ciertas condiciones

```
always @ ( eventos ) begin
... elementos ...
end
```

- **Los bloques always actúan sobre registros, no sobre wires**
- Los procedimientos realizan asignaciones
  - Bloqueantes (=): ocurren en secuencia
  - No bloqueantes (<=): ocurren en paralelo
  - No mezclarlas en un mismo bloque always

Este bloque always se ejecuta cada vez que cambia el valor de a, b, c o d

```
1  module example1_always(a,b,c,d,out)
2
3    input a, b, c, d;
4    output out;
5
6    always @(a or b or c or d)
7      out <= ~((a & b) | (c ^ d));
8
9  endmodule
```

Ejemplos – declaraciones de wire y reg

- Al definir las entradas y salidas de un módulo, estas por defecto son variables tipo *wire*

```
1  module wire_reg_example(Out)
2
3    output Out;            // Variable tipo wire por defecto
4    // con "output reg Out;" la salida sería un registro
5
6    wire A, B, C;          // Cables de 1 bit
7    wire [7:0] Byte;       // Un bus de 8 bits
8
9    reg Q1, Q2;            // Registros de 1 bit
10   reg [7:0] registro;    // Un registro de 8 bits
11
12   assign Out = Byte[6]   // Asignamos el bit 6 de Byte a Out
13
14 endmodule
```

Constantes en Verilog

- Las usamos como apoyo en la descripción del hardware
  - Útil para definir módulos que procesan datos de tamaño arbitrario

```
1  // Módulo multiplexor con ancho variable
2  module mux4 #(parameter WIDTH=1) (a,b,c,d,sel,z)
3
4    input [WIDTH-1:0] , b, c, d;  // Entradas de WIDTH bits
5    input [1:0] sel;              // Selector
6    output [WIDTH-1:0] z;         // Salida de WIDTH bits
7
8    wire [WIDTH-1:0] t0, t1;      // Cables intermedios
9
10   assign t0 = sel[1] ? c : a;
11   assign t1 = sel[1] ? d : b;
12   assign z = sel[0] ? t0 : t1;
13
14 endmodule
```

- Luego podemos instanciar el módulo especificando el parámetro:

```
1  module mux4_test(x1,x2,x3,x4,sel,out);
2
3    input [31:0] x1, x2, x3, x4;
4    input [1:0] sel;
5    output [31:0] salida;
6
7    // Instancación de mux4
8    mux4 #(32) alu_mux(.a(x1), .b(x2), .c(x3), .d(x4), .sel(sel), .z(salida))
9
10 endmodule
```

¿Qué aprendimos hoy?

- Tipos de datos
- Conexiones en Verilog
  - Variables tipo Wires
- Variables tipo registros
- Asignaciones
  - Bloqueantes y no bloqueantes
- Constantes en Verilog

Operadores en Verilog

Operadores en Verilog

Tipo	Símbolo	Operación	N° Operandos
Bitwise	~	Complemento de 1	1
	&	bitwise AND	2
	|	bitwise OR	2
	^	bitwise XOR	2
	~^ or ^~	bitwise XNOR	2
Lógico	!	NOT	1
	&&	AND	2
	||	OR	2
De reducción	&	Reducción AND	1
	~&	Reducción NAND	1
	|	Reducción OR	1
	~|	Reducción NOR	1
	^	Reducción XOR	1
	~^ or ^~	Reducción XNOR	1

- Hay distintos tipos:
  - Operadores bit a bit (bitwise)
  - Lógicos
  - De reducción
  - Aritméticos
  - Relacionales o de comparación
  - De igualdad
  - De concatenación
  - De replicación
  - Condicional (siguiente video)

Operadores en Verilog

Tipo	Símbolo	Operación	N° Operandos
Aritmético	+	Adición	2
	–	Resta	2
	–	Complemento de 2	1
	*	Multiplicación	2
	/	División	2
Relacional	>	Mayor que	2
	<	Menor que	2
	>=	Mayor o igual que	2
	<=	Menor o igual que	2
De igualdad	==	Igualdad lógica	2
	!=	Desigualdad lógica	2
De desplazamiento	>>	Desplazamiento a la derecha	2
	<<	Desplazamiento a la izquierda	2
Concatenación	{, }	Concatenación	Cualquier número
Replicación	{{}}	Replicación	Cualquier número
Condicional	?:	Condicional	3

- Hay distintos tipos:
  - Operadores bit a bit (bitwise)
  - Lógicos
  - De reducción
  - Aritméticos
  - Relacionales o de comparación
  - De igualdad
  - De concatenación
  - De replicación
  - Condicional (siguiente video)

Operadores bit a bit

- Tablas de verdad para operadores bit a bit

&	0	1	x
0	0	0	0
1	0	1	x
x	0	x	x

|	0	1	x
0	0	1	x
1	1	1	1
x	x	1	x

^	0	1	x
0	0	1	x
1	1	0	x
x	x	x	x

~^	0	1	x
0	1	0	x
1	0	1	x
x	x	x	x

Operadores lógicos

- El operador ! tiene el mismo efecto que ~ en escalares
- Pero si el argumento es un vector $\mathrm{A} = a_1 a_2 a_3$, entonces

$$!\,\mathrm{A} = \overline{a_2 + a_1 + a_0}$$

- El operador && produce un AND lógico entre dos vectores

$$\mathrm{A\&\&B} = (a_2 + a_1 + a_0) \cdot (b_2 + b_1 + b_0)$$

- El operador || produce un OR lógico entre dos vectores

$$\mathrm{A||B} = (a_2 + a_1 + a_0) + (b_2 + b_1 + b_0)$$

Operadores de reducción

- Reducen un vector a un bit de salida
- Por ejemplo:

$$\&A = a_2 \cdot a_1 \cdot a_0$$

- Mientras que:

$$\hat{}A = a_2 \oplus a_1 \oplus a_0$$

- Y sirve para calcular paridad

Operadores aritméticos

- Producen operaciones aritméticas:
  - $A + B$ es la suma de A y B
  - $A - B$ es la diferencia entre A y B
  - $-A$ es el complemento de 2 de A (más adelante...)
- La operación / no siempre es soportada por sistemas de CAD
  - Requiere alguna biblioteca
- **Eviten multiplicar y dividir**, a menos que sepan exactamente lo que están haciendo
  - En lo posible, usen multiplicación o división por potencias de 2

Operadores relacionales y de igualdad

```
1  module compare(A,B,AeqB,AgtB,AltB);
2    input [3:0] A, B;
3    output reg AeqB, AgtB, AltB;
4
5    always @(A, B)
6      begin
7        AeqB = 0;
8        AgtB = 0;
9        AltB = 0;
10       if (A == B)
11         AeqB = 1;
12       else if (A > B)
13         AgtB = 1;
14       else
15         AltB = 1;
16     end
17
18 endmodule
```

- Típicamente usados en expresiones condicionales y en loops for
- Misma sintaxis que en C
- El resultado del operador igualdad (==) o desigualdad (!=) es ambiguo (x) si algún operando contiene x o z

Operadores de desplazamiento

- Llenan con ceros el resto de los bits
  - Si $\mathrm{A} = a_2 a_1 a_0$, $\mathrm{B} = \mathrm{A} \ll 1$ produce $\mathrm{B} = a_1 a_0 0$ (multiplica por 2)
  - Si $\mathrm{A} = a_2 a_1 a_0$, $\mathrm{B} = \mathrm{A} \gg 2$ produce $\mathrm{B} = 00 a_2$ (divide por 4)
- Si uno quiere hacer desplazamiento circular o rotación, debe implementarlo usando concatenación
- Si uno quiere hacer desplazamiento a la derecha con extensión de signo, también hay que usar concatenación

Operadores de concatenación y replicación

- Concatenación:
  - $\mathrm{D} = \{\mathrm{A}, \mathrm{B}\}$ produce $\mathrm{D} = a_2 a_1 a_0 b_2 b_1 b_0$
  - $\mathrm{E} = \{3'b111, \mathrm{A}, 2'b00\}$ produce $\mathrm{E} = 111 a_2 a_1 a_0 00$
- El operador replicación puede ser usado junto al de concatenación
  - $\{3\{\mathrm{A}\}\}$ es equivalente a $\{\mathrm{A}, \mathrm{A}, \mathrm{A}\}$
  - $\{3\{2'\mathrm{b10}\}\}$ produce $8'\mathrm{b10101010}$

Precedencia de operadores

Tipo	Símbolo	Precedencia
Complemento	! ~ –	Mayor
Aritmético	* / + –	
De desplazamiento	<< >>	
Relacional	< <= > >=	
Igualdad	== !=	
De reducción	& ~& ^ ~^ | ~|	
Lógico	&& ||	
Condicional	?:	Menor

¿Qué aprendimos hoy?

- Operadores en Verilog
  - Bit a bit
  - Lógicos
  - De reducción
  - Aritméticos
  - Relacionales
  - De igualdad
  - De desplazamiento
  - De replicación y concatenación
- Precedencia de operadores

Operadores condicionales en Verilog

El operador condicional (?)

- Es un operador que toma decisiones según una condición
- El uso de paréntesis es muy útil para mejorar la lectura
- Se puede usar en asignación continua y en declaraciones de procedimiento (bloques always)
- Podemos implementar multiplexores con este operador

```
1  module mux2to1(x0,x1,s,f);
2    input x0, x1, s;
3    output f;
4
5    assign f = s ? x1 : x0;
6
7  endmodule
```

```
1  module mux2to1(x0,x1,s,f);
2    input x0, x1, s;
3    output reg f;
4
5    always @(x0, x1, s)
6      f = s ? x1 : x0;
7
8  endmodule
```

El operador condicional (?)

- Podemos anidar operadores condicionales
  - Pero no hay que abusar de esto (difícil de leer)

```
1 module mux4to1(x0,x1,x2,x3,S,f);
2   input x0, x1, x2, x3;
3   input [1:0] S;
4   output f;
5
6   assign f = S[1] ? (S[0] ? x3 : x2) : (S[0] ? x1 : x0);
7
8 endmodule
```

- El operador condicional no es la única forma de describir un multiplexor

Declaración if-else

- Es otra declaración que permite tomar decisiones
- Caso más general que el operador condicional
- Si la expresión evaluada es verdadera, entonces se ejecuta la primera declaración (if)
- Si es falsa, se ejecuta la segunda declaración (else)
- Deben ir **siempre en un bloque always** porque es un procedimiento que se ejecuta

```
1  module mux2to1(x0,x1,s,f);
2    input x0, x1, s;
3    output reg f;
4
5    always @(x0, x1, s)
6      if (s == 0)
7        f = x0;
8      else
9        f = x1;
10
11 endmodule
```

Declaración if-else

```
1  module mux4to1(x0,x1,x2,x3,S,f);
2    input x0, x1, x2, x3;
3    input [1:0] S;
4    output reg f;
5
6    always @(*)
7      if (S ==2'b00)
8        f = x0;
9      else if (S == 2'b01)
10       f = x1;
11     else if (S == 2'b10)
12       f = x2;
13     else
14       f = x3;
15
16 endmodule
```

- También es posible anidar declaraciones if-else
- Siempre es bueno terminar la declaración con un else

Declaración if-else

```
1  module mux4to1(W,S,f);
2    input [0:3] W;
3    input [1:0] S;
4    output reg f;
5
6    always @(W, S)
7      if (S == 0)
8        f = W[0];
9      else if (S == 1)
10       f = W[1];
11     else if (S == 2)
12       f = W[2];
13     else
14       f = W[3];
15
16 endmodule
```

- Aquí escribimos el multiplexor en una forma más compacta usando vectores
- Es más fácil de leer el código de esta forma que con escalares

Declaración if-else

```
1  module mux16to1(W,S,f);
2    input [0:15] W;
3    input [3:0] S;
4    output f;
5    wire [0:3] M;
6
7    mux4to1 Mux1(W[0:3], S[1:0], M[0]);
8    mux4to1 Mux2(W[4:7], S[1:0], M[1]);
9    mux4to1 Mux3(W[8:11], S[1:0], M[2]);
10   mux4to1 Mux4(W[12:15], S[1:0], M[3]);
11   mux4to1 Mux5(W[0:3], S[3:2], f);
12
13 endmodule
```

- Por supuesto que también podemos trabajar con jerarquía
- En este caso instanciamos 4 multiplexores 4:1 para crear un multiplexor 16:1

¿Qué aprendimos hoy?

- Verilog para circuitos combinacionales
  - Operador condicional (?)
  - Declaraciones if-else
- Muchos ejemplos!

2.05

Case, casex, casez y loop for

Declaración case

- Permite hacer lo mismo que un condicional anidado
- **Muy útil** para describir tablas de verdad
- Debe ir **siempre en un bloque always** porque es un procedimiento
- Es buena práctica terminar con default

```
 6  module mux4to1(W,S,f);
 7
 8  input [0:3] W;
 9  input [1:0] S;
10  output reg f;
11
12  always @(W, S)
13      case(S)
14          0: f = W[0];
15          1: f = W[1];
16          2: f = W[2];
17          3: f = W[3];
18          default: f = 0;
19      endcase
20
21  endmodule
```

Ejemplo – Decodificador con case

- Este ejemplo es un decodificador de binario a one-hot con enable (más adelante)
- Aquí concatenamos en el argumento del case
- Al expresar la entrada y la salida en binario, la tabla de verdad queda a la vista

```
1  module dec2to4(W,En,Y);
2    input [1:0] W;
3    input En;
4    output reg [0:3] Y;
5
6    always @(W, En)
7      case({En,W})
8        3'b100: Y = 4'b1000;
9        3'b101: Y = 4'b0100;
10       3'b110: Y = 4'b0010;
11       3'b111: Y = 4'b0001;
12       default: Y = 4'b0000;
13     endcase
14
15 endmodule
```

Ejemplo – Decodificador con case

- Otra posible implementación es utilizando un if para el enable
- Otra opción sería hacer un AND entre las salidas y el enable

```
1  module dec2to4(W,En,Y);
2    input [1:0] W;
3    input En;
4    output reg [0:3] Y;
5
6    always @(W, En)
7      begin
8        if (En == 0)
9          Y = 4'b0000;
10       else
11         case({En,W})
12           3'b100: Y = 4'b1000;
13           3'b101: Y = 4'b0100;
14           3'b110: Y = 4'b0010;
15           3'b111: Y = 4'b0001;
16           default: Y = 4'b0000;
17         endcase
18     end
19
20 endmodule
```

Ejemplo – Decodificador con case

- Por supuesto que siempre es posible usar jerarquía para implementar bloques más complicados
- En este caso, decodificador de 4 a 16

```
1  module dec4to16(W,En,Y);
2    input [3:0] W;
3    input En;
4    output reg [0:15] Y;
5    wire [0:3] M;
6
7    dec2to4 Dec1(W[3:2], En, M[0:3]);
8    dec2to4 Dec2(W[1:0], M[0], Y[0:3]);
9    dec2to4 Dec3(W[1:0], M[1], Y[4:7]);
10   dec2to4 Dec4(W[1:0], M[2], Y[8:11]);
11   dec2to4 Dec5(W[1:0], M[3], Y[12:15]);
12
13 endmodule
```

Declaraciones casex y casez

- Son variantes del case
- Permiten usar x's y z's como comodines
  - De esta forma no tenemos que poner una tabla tan larga
  - Casez trata todas las z's en las alternativas (ítems) en la expresión como don't cares
  - Casex trata todas las x's y z's como don't cares
- Es posible que haya más de un ítem case que iguale la expresión
  - En este caso, el primer ítem es el que vale
  - En ese caso, es mejor usar if-else para tener explícita la prioridad
- Usar con precaución o mejor no usarlas

```
1  module priority(W,Y,z);
2    input [3:0] W;
3    output reg [1:0] Y;
4    output reg z;
5
6    always @(W)
7      begin
8        z = 1;
9        casex(W)
10         4'b1xxx: Y = 3;
11         4'b01xx: Y = 2;
12         4'b001x: Y = 1;
13         4'b0001: Y = 0;
14         default: begin
15           z = 0;
16           Y = 2'bx;
17         end
18       endcase
19     end
20
21 endmodule
```

Loop for

- Estructura que permite expresar cierta regularidad
- Debe **ir siempre en un bloque always** porque es un procedimiento
- La variable de control debe ser tipo entero
- A diferencia de los lenguajes de programación en que se especifican cambios en cada iteración del loop, en Verilog **el loop for especifica diferentes subcircuitos en cada iteración**

```
1  module dec2to4(W,En,Y);
2    input [1:0] W;
3    input En;
4    output reg [0:3] Y;
5    integer k;
6
7    always @(W, En)
8      for(k = 0; k <= 3; k = k + 1)
9        if((W == k) && (En == 1))
10         Y[k] = 1;
11       else
12         Y[k] = 0;
13
14 endmodule
```

Loop for en un codificador de prioridad

- En Verilog, si una señal es especificada varias veces durante una ejecución de bloque always, retiene el valor de la última asignación
- Este ejemplo muestra justamente este comportamiento
- Quiero ser honesto, yo (Ángel Abusleme) **no recomiendo** el bloque for en Verilog
  - Al menos no es para principiantes
  - No es como el loop for en programación
  - Puede resultar en circuitos subóptimos
  - Piensen que cada iteración usa un circuito y una lógica nueva

```
1  module priority(W,Y,z);
2    input [3:0] W;
3    output reg [1:0] Y;
4    output reg z;
5    integer k;
6
7    always @(W)
8      begin
9        Y = 2'bx
10       z = 0;
11       for(k = 0; k < 4; k = k + 1)
12         if(W[k])
13           begin
14             Y = k;
15             z = 1;
16           end
17     end
18
19 endmodule
```

¿Qué aprendimos hoy?

- Declaración case
- Declaraciones casex y casez
- Loop for
- Muchos ejemplos!

Testbench en Verilog

Testbench en Verilog

- Un testbench es un módulo de Verilog que verifica el funcionamiento de otro módulo mediante una simulación
  - El proceso de testbench se salta el proceso de síntesis del módulo a verificar
- El módulo bajo prueba se le denomina usualmente como UUT
  - UUT: Unit Under Test
- Veamos un ejemplo para entender mejor como armar un testbench

```
3   module logic_op(A,B,C,D,out1,out2,out3);
4
5     input A, B, C, D;
6     output out1, out2, out3;
7
8     assign out1 = !A;
9     assign out2 = (B & C) | D;
10    assign out3 = A ^ B ^ (!D);
11
12  endmodule
```

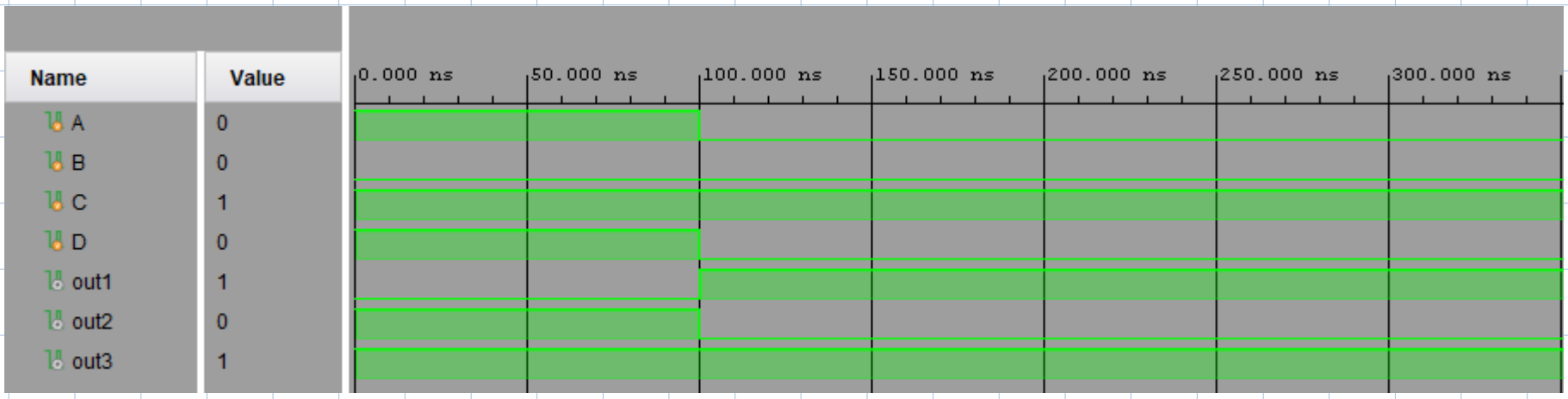
Generación de estímulos

- El testbench no posee entradas ni salidas
- Las entradas del módulo a evaluar las definimos como variables reg y sus salidas como variables wire
- Instanciamos el módulo con el nombre uut y conectamos las señales de entrada y salida del módulo
- Como diseñadores, definimos estímulos con los cuales evaluar el funcionamiento de la uut
  - Asignamos valores a las variables registro para que ingresen al módulo y observemos las salidas
- Las instrucciones dentro de la declaración initial se ejecutan al inicio de la simulación
  - `#100` representa un retardo de 100 ns

```
1  `timescale 1ns / 1ps
2
3  module logic_op_TB;
4
5    reg A, B, C, D;
6    wire out1, out2, out3;
7
8    // Instanciamos UUT = Unit Under Test
9    logic_op uut(A,B,C,D,out1,out2,out3);
10
11   // Generación de estímulos
12   initial begin
13     A = 1'b1;
14     B = 1'b0;
15     C = 1'b1;
16     D = 1'b1;
17     #100;
18     A = 1'b0;
19     B = 1'b0;
20     C = 1'b1;
21     D = 1'b0;
22   end
23
24 endmodule
```

Resultados del testbench

- Al ejecutar el testbench usualmente generamos un diagrama de tiempo para observar las salidas de la uut



- Con un módulo tan simple es fácil concluir del testbench que nuestro módulo funciona sin errores

Testbench en Verilog

- Al trabajar con módulos más grandes y más complejos, diseñar un testbench se vuelve más difícil
- Como buen diseñador, uno debería realizar un testbench de cada módulo que diseña para verificar que esté correcto
- Diseñar un buen testbench puede demorar tanto como diseñar el módulo a evaluar

```
1  `timescale 1ns / 1ps
2
3  module addern_v1(carryin,X,Y,S,carryout);
4
5  parameter n = 32;
6  input carryin;
7  input [n-1:0] X, Y;          // Entradas de tamaño arbitrario
8  output [n-1:0] S;            // Salida de tamaño arbitrario
9  output carryout;
10
11 reg [n-1:0] S;               // Las variables cuyos valores se definen
12 reg carryout;                // dentro de un bloque always deben ser
13 reg [n:0] C;                 // variables tipo reg
14 integer k;                   // Número entero que maneja el loop del for
15
16 always @(X or Y or carryin)
17 begin
18     C[0] = carryin;
19     for (k = 0; k < n; k = k + 1)    // Un for debe ir siempre dentro de
20         begin                        // un bloque always
21             S[k] = X[k] ^ Y[k] ^ C[k];
22             C[k+1] = (X[k] & Y[k])|(X[k] & C[k])|(Y[k] & C[k]);
23         end
24     carryout = C[n];
25 end
26
27 endmodule
```

Testbench en Verilog

```
1  `timescale 1ns / 1ps
2
3  module addern_v1_TB;
4
5  parameter n = 32;
6  reg carryin;
7  reg [n-1:0] X, Y;          // Entradas de tamaño arbitrario
8  wire [n-1:0] S;            // Salida de tamaño arbitrario
9  wire carryout;
10
11 // UUT
12 addern_v1 uut(carryin,X,Y,S,carryout);
13
14 initial begin
15     carryin = 1'b0;
16     X = 32'b111010;
17     Y = 32'b10011011;
18     #150;
19     carryin = 1'b0;
20     X = 32'b110011;
21     Y = 32'b10010000;
22     #150;
23     carryin = 1'b1;
24     X = 32'b10111011;
25     Y = 32'b11010011;
26     #150;
27     carryin = 1'b1;
28     X = 32'b1100110;
29     Y = 32'b10011001;
30     #150;
31 end
32
33 endmodule
```

- Con módulos más complejos, la generación de estímulos debe realizarse con más cuidado para verificar el funcionamiento de la uut en su totalidad

Testbench en Verilog

- Ya sabemos cómo hacer un testbench para evaluar nuestros futuros diseños
  - ¿En dónde realizamos un testbench?
- Típicamente las herramientas CAD cuentan con el software apropiado para ejecutar un testbench, por ejemplo:
  - **EDA Playground** cuenta con una página web para diseñar con Verilog y probar nuestros diseños con testbench (online)
  - **Vivado Design Suite** de Xilinx (más adelante) es un software para diseñar con Verilog y programar FPGAs. Permite realizar testbench de nuestros módulos

¿Qué aprendimos hoy?

- Utilidad de los testbench en Verilog
- Diseñar un testbench