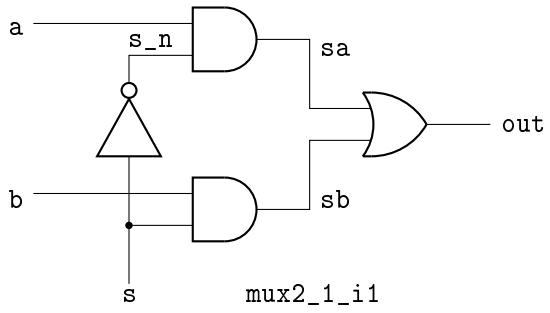
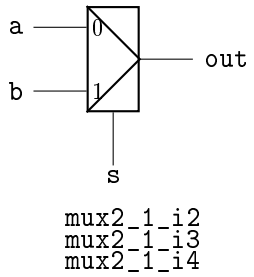
Práctica 1

El objetivo de la práctica 1 es servir de introducción al Verilog con un ejemplo de diseño puramente combinacional. El ejemplo propuesto es el de construir una pequeña ALU capaz de realizar sumas y restas en complemento a dos de 4 bits y generar los bits de flags asociados, así como varias operaciones lógicas. Seguiremos un proceso incremental, diseñando los módulos componentes más simples que combinados constituirán la ALU. Escribiremos cada módulo en un fichero diferente, cuyo nombre será el mismo de como hayamos denominado al módulo añadiendo la extensión "`.v`"

Objetivo 1

El profesor suministrará varios ficheros ejemplo con diferentes implementaciones de un multiplexor de dos entradas de un bit. Después de examinarlos, se comprobará su funcionamiento con ayuda de un fichero *testbench* específico, simulando su ejecución.





Objetivo 2

Basándonos en los ejemplos anteriores, implementar un multiplexor de 4 entradas de un bit con el siguiente prototipo (la salida `out` podrá ser también de tipo `wire` si se cree conveniente)

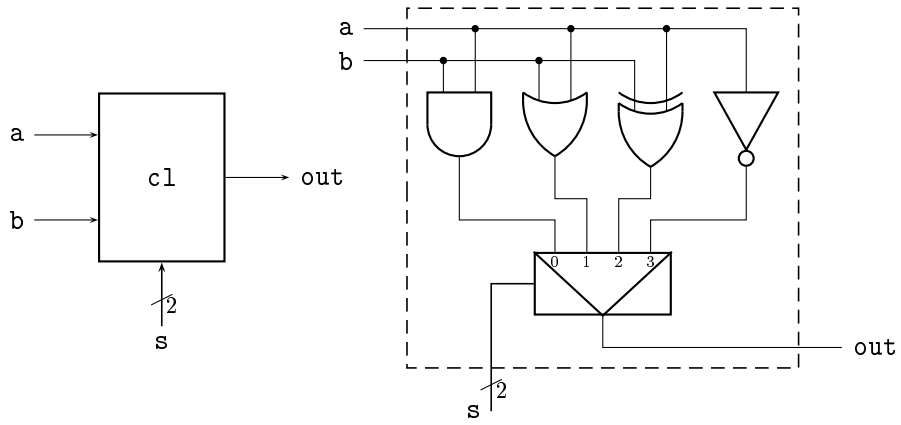
```
mux4_1(output reg out, input wire a, b, c, d, input wire [1:0] s);
```

Objetivo 3

a) Implementar una "celda lógica" con el prototipo siguiente

```
cl(output reg out, input wire a, b, input wire [1:0] s);
```

Dicha celda lógica calculará sobre los bits `a` y `b` las operaciones lógicas `and`, `or`, `xor` e inversión del bit a cuando el vector de dos bits `s` vale 00, 01, 10 y 11 respectivamente.

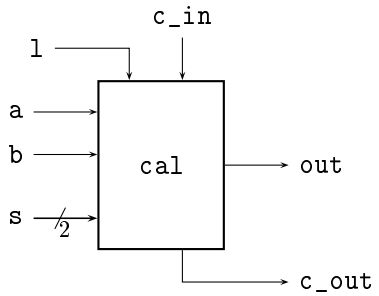


b) Implementar un "Full-Adder" de la manera más simple posible (operador concatenación '$\{\cdots\}$'), con el prototipo

```
fa(output wire cout, sum, input wire a, b, cin);
```

c) Unir las dos implementaciones en una "celda arimético-lógica" mediante un multiplexor que conecte sus salidas controlado por una entrada '`l`') (de "Lógica") de forma que cuando esta señal esté a 1 se tenga la salida de la celda lógica y a 0 se obtenga la salida del "full-adder". El prototipo sería el siguiente:

```
cal(output wire out, c_out, input wire a, b, l, c_in, input wire [1:0] s);
```



Objetivo 4

El objetivo de esta fase es completar la ALU. Las operaciones lógicas que puede realizar son las que ya se han implementado en la celda lógica. La operaciones aritméticas que debemos desarrollar (dados dos operandos `A` y `B`, vectores de cuatro bits) son: la suma `A+B`, la resta `A-B`, el `complemento a 2 de A` y el `complemento a 2 de B`.

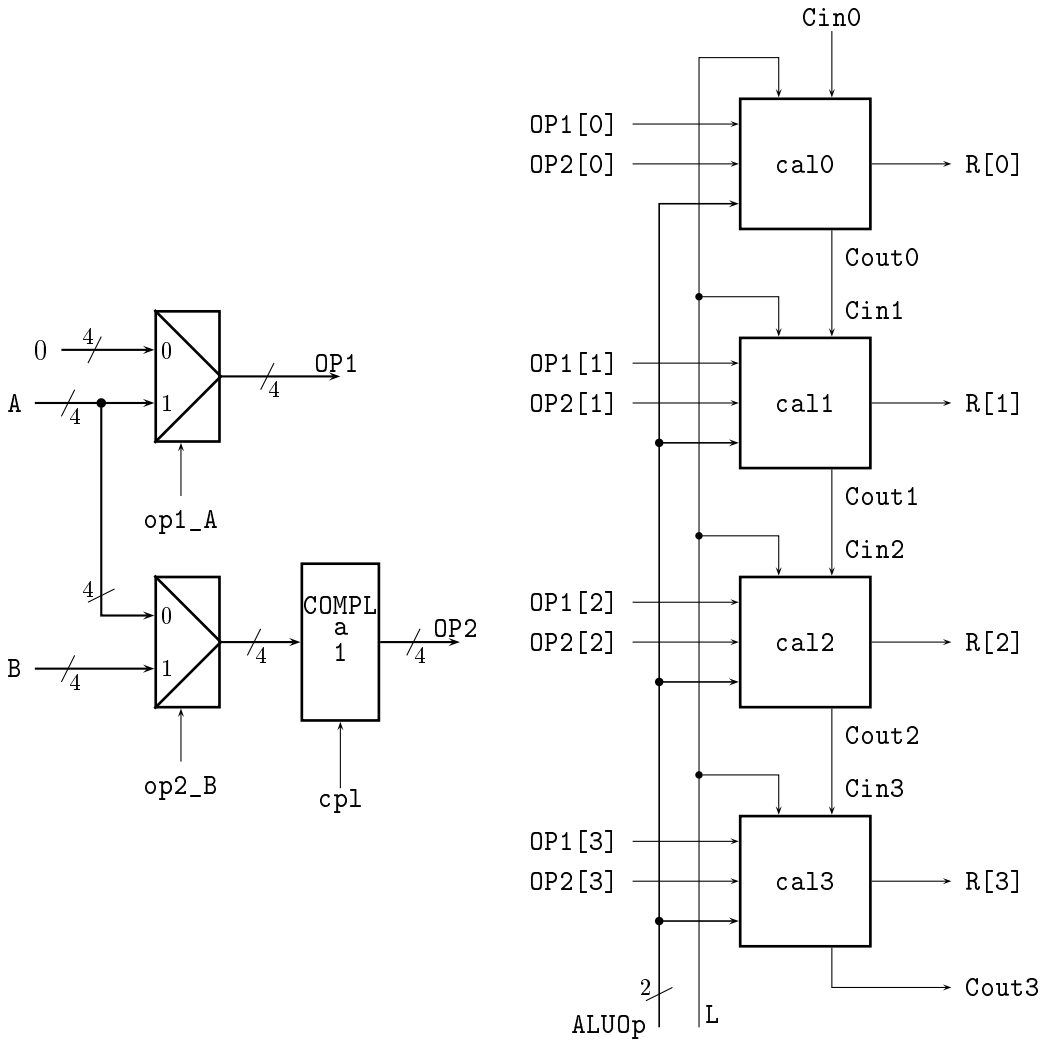
La operación de suma la elaboramos simplemente instanciando cuatro elementos `cal` y conectando los acarreos de salida de un elemento al acarreo de entrada del siguiente, como aparece en la parte derecha de la figura de más abajo. Se han denotado las entradas a ese sumador como `OP1` y `OP2`, ambos vectores de cuatro bits.

La operación de complemento a dos la podemos lograr a partir de los elementos que tenemos mediante el procedimiento de realizar el complemento a 1 y sumar 1. A su vez, la suma de una unidad la podemos lograr aprovechando la entrada de acarreo que nos ha quedado `Cin0`. Si ponemos dicha entrada a 1, lograríamos sumar una unidad en el elemento `cal` de la posición menos significativa.

Respecto al complemento a 1, es interesante que podamos complementar o no a voluntad, en función de una señal de control `cpl`, de forma que podamos dejar pasar un dato sin modificar o hacer su complemento a uno. Si elaboramos un elemento que realice esto, podría tener el prototipo siguiente:

```
compl1(output wire [3:0] Sal, input wire [3:0] Ent, input wire cpl);
```

Si `cpl = 1`, `Sal = Cpl1(Ent)` y si no, `Sal = Ent`.



Una vez que tengamos estos elementos, podemos elaborar las operaciones de la resta `A-B`, el `complemento a 2 de B` y el `complemento a 2 de A` simplemente presentando a las entradas `OP1` y `OP2` del sumador los valores adecuados:

`A - B` =	`A + (-B) = A + Cpl2(B) = A + Cpl1(B) + 1`
	Para lograr esto, `OP1 = A`, `OP2 = Cpl1(B)`, `Cin0 = 1`
`Cpl2(A)` =	`0 + Cpl2(A) = 0 + Cpl1(A) + 1`
	Para lograr esto, `OP1 = 0`, `OP2 = Cpl1(A)`, `Cin0 = 1`
`Cpl2(B)` =	`0 + Cpl2(B) = 0 + Cpl1(B) + 1`
	Para lograr esto, `OP1 = 0`, `OP2 = Cpl1(B)`, `Cin0 = 1`
`A + B`	se obtiene simplemente con
	`OP1 = A`, `OP2 = B`, `Cin0 = 0`

Por tanto, vemos que `OP1` será o bien `A`, o bien 0. También vemos que `OP2` será `A`, `Cpl1(A)`, `B` ó `Cpl1(B)`. Todo esto se logra con la lógica a base de multiplexores que aparece en la parte izquierda de la figura anterior.

Para elaborar la ALU seguiremos el diseño de la figura, completando los módulos que creamos necesarios, de forma que el módulo final tenga el prototipo

```
alu(output wire [3:0] R, output wire zero, carry, sign,
input wire [3:0] A, B, input wire [1:0] ALUOp, input wire L);
```

y el funcionamiento debe ser el descrito por la siguiente tabla

L	**ALUOp**	**R**	**zero**	**carry**	**sign**
0	00	`A + B`	1 si `(R = 0)`	1 si `(A+B)` tiene acarreo	Bit más significativo de R
0	01	`A - B`	1 si `(R = 0)`	1 si `(A-B)` tiene acarreo	Bit más significativo de R
0	10	`Cpl2(A)`	1 si `(R = 0)`	1 si `Cpl2(A)` tiene acarreo	Bit más significativo de R
0	11	`Cpl2(B)`	1 si `(R = 0)`	1 si `Cpl2(B)` tiene acarreo	Bit más significativo de `R`
1	00	`A and B`	1 si `(R = 0)`	No importa	No importa
1	01	`A or B`	1 si `(R = 0)`	No importa	No importa
1	10	`A xor B`	1 si `(R = 0)`	No importa	No importa
1	11	`Cpl1(A)`	1 si `(R = 0)`	No importa	No importa

Vemos que la entrada `L` indica si la operación es del grupo lógico o aritmético (`L = 1`: op. lógica, `L=0`: op. Aritmética) y que los dos bits de la entrada `ALUOp` especifican la operación dentro de cada uno de estos grupos.

Para terminar el diseño y lograr el comportamiento anterior, tenemos que diseñar las funciones lógicas de las señales que controlan los multiplexores, el complementador a uno y el acarreo de entrada (`op1_A`, `op2_B`, `cpl` y `Cin0`) a partir de las entradas `L`, `ALUOp[1]` y `ALUOp[0]`. Puede ayudar a crearlas el escribir las tablas de verdad de cada una. Por último, también es necesario diseñar las funciones lógicas de los *flags* de `cero`, `acarreo` y `signo` (las dos últimas son muy sencillas, el cero requiere una pequeña expresión lógica de los bits de `R`).

Una vez acabado el diseño, comprobar su buen funcionamiento con el *testbench* que proporcionará el profesor.