

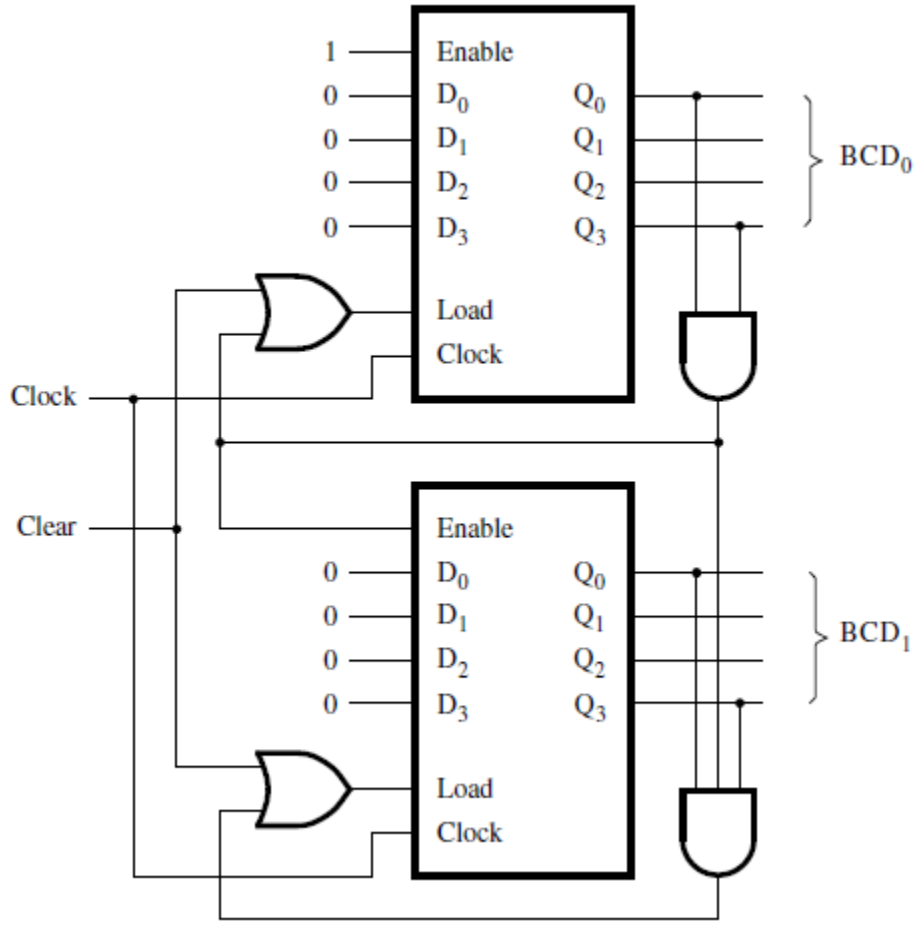
7.11 – 7.11.4

Otros contadores

angel@uc.cl

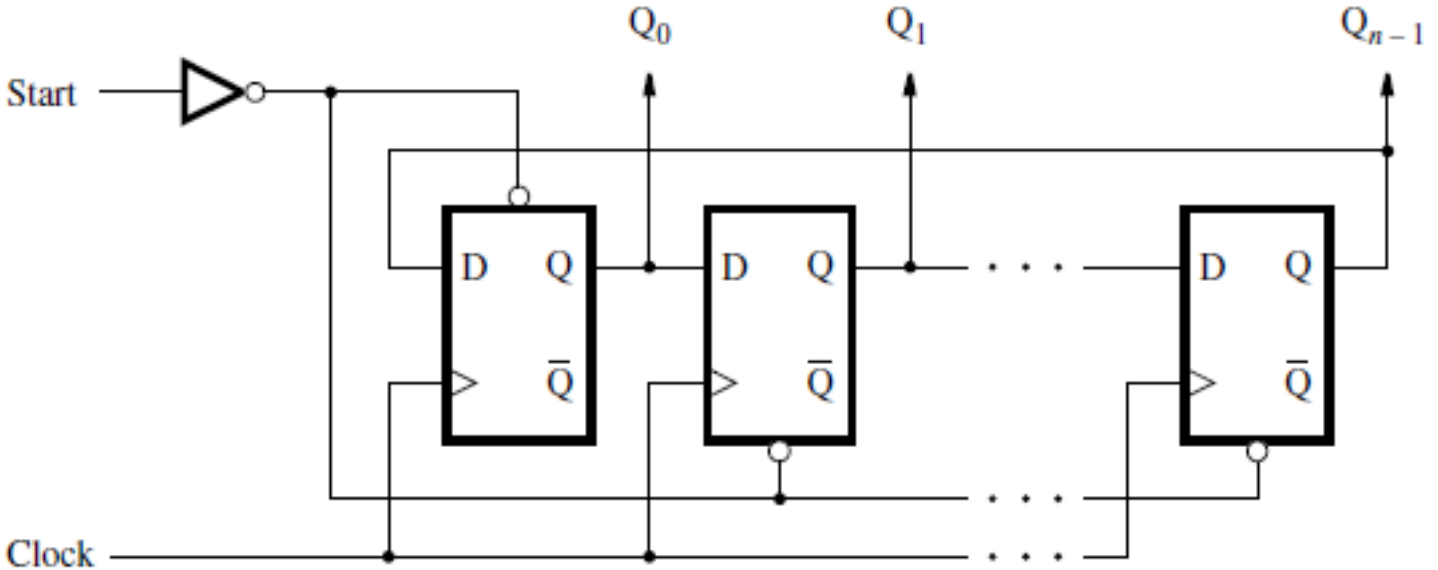
IEE2713 Sistemas Digitales

7.11.1 Contador BCD



- Un Contador de 2 dígitos BCD es, en principio, dos contadores de modulo 10 con algunas consideraciones:
  - Cuando Q_3 y Q_0 son iguales a 1 (contador alcanza el 9), load se vuelve 1 y resetea el contador
  - El enable de BCD_1 solo se activa cuando BCD_0 llega a 9

7.11.2 Contador de anillo



(a) An n-bit ring counter

Con n flip-flops, hacemos una cuenta de n números

- En un contador de anillo, la cuenta es llevada por un que 1 va pasando entre los distintos bits de salida
  - Por ejemplo, la cuenta parte como 1000, luego se avanza a 0100, 0010 y 0001 (*one-hot*)
  - No es muy eficiente en términos de hardware, pero puede ser útil en aplicaciones que requieren contador+decoder

7.11.3 Contador Johnson

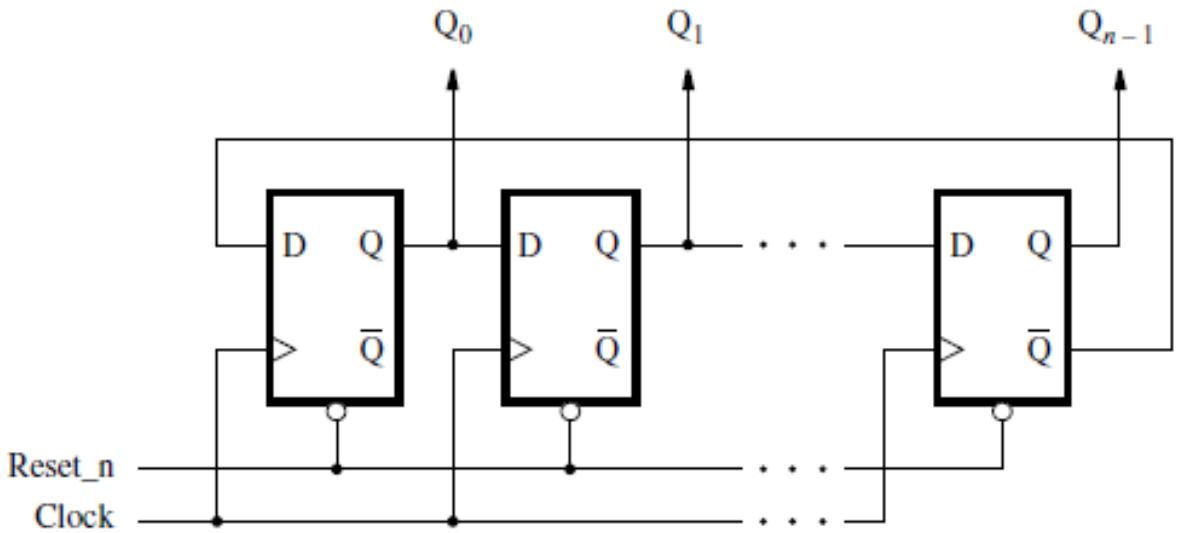


Figure 5.29 Johnson counter.

- Un contador de Johnson es similar a un contador de anillo donde la salida Q_b es realimentada
  - Con n flip-flops, es posible hacer secuencias de 2n números

Secuencia 0000 – 1000 – 1100 – 1110 – 1111 – 0111 – 0011 – 0001 – 0000

7.12 – 7.13.3

Circuitos con memoria en CAD

angel@uc.cl

IEE2713 Sistemas Digitales

7.12.1 Elementos con memoria en esquemáticos

- Las memorias en los circuitos son representadas como latches o flip-flops
- Estos son ampliamente usados y suelen ser bloques preconfigurados

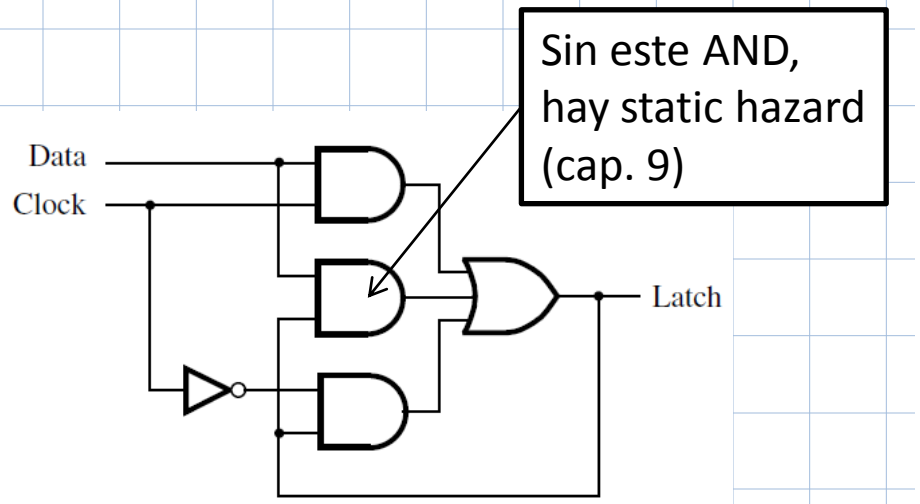
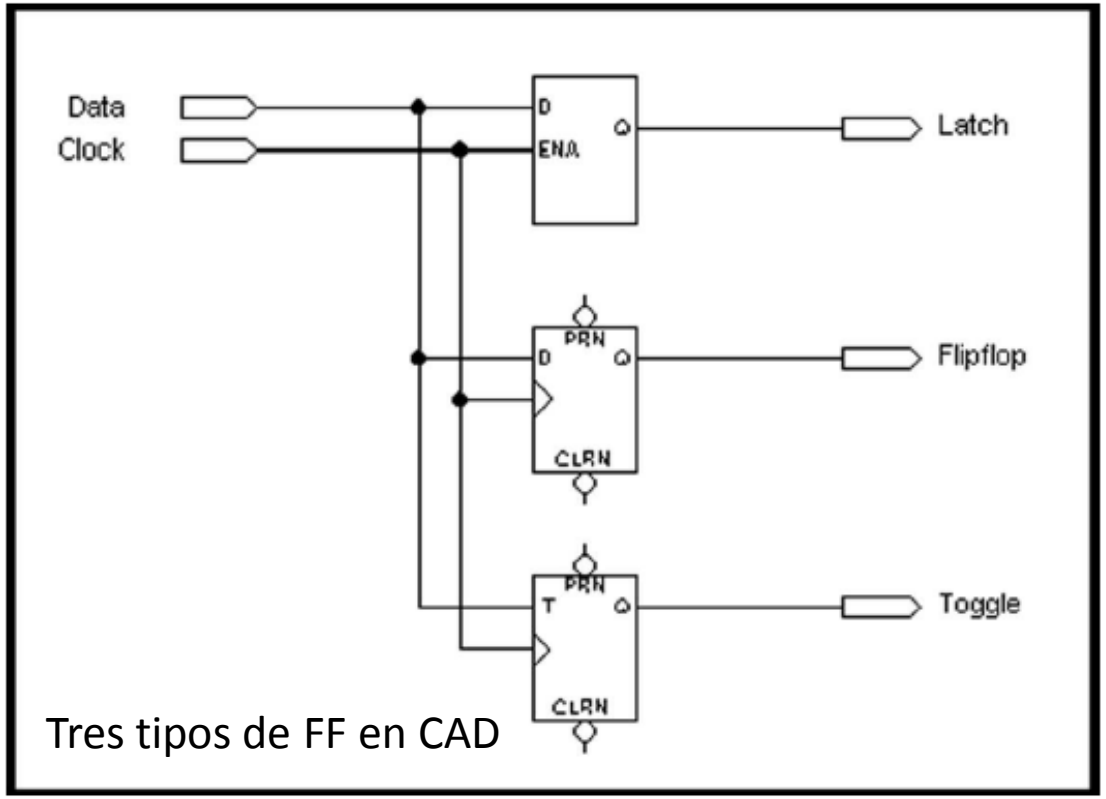


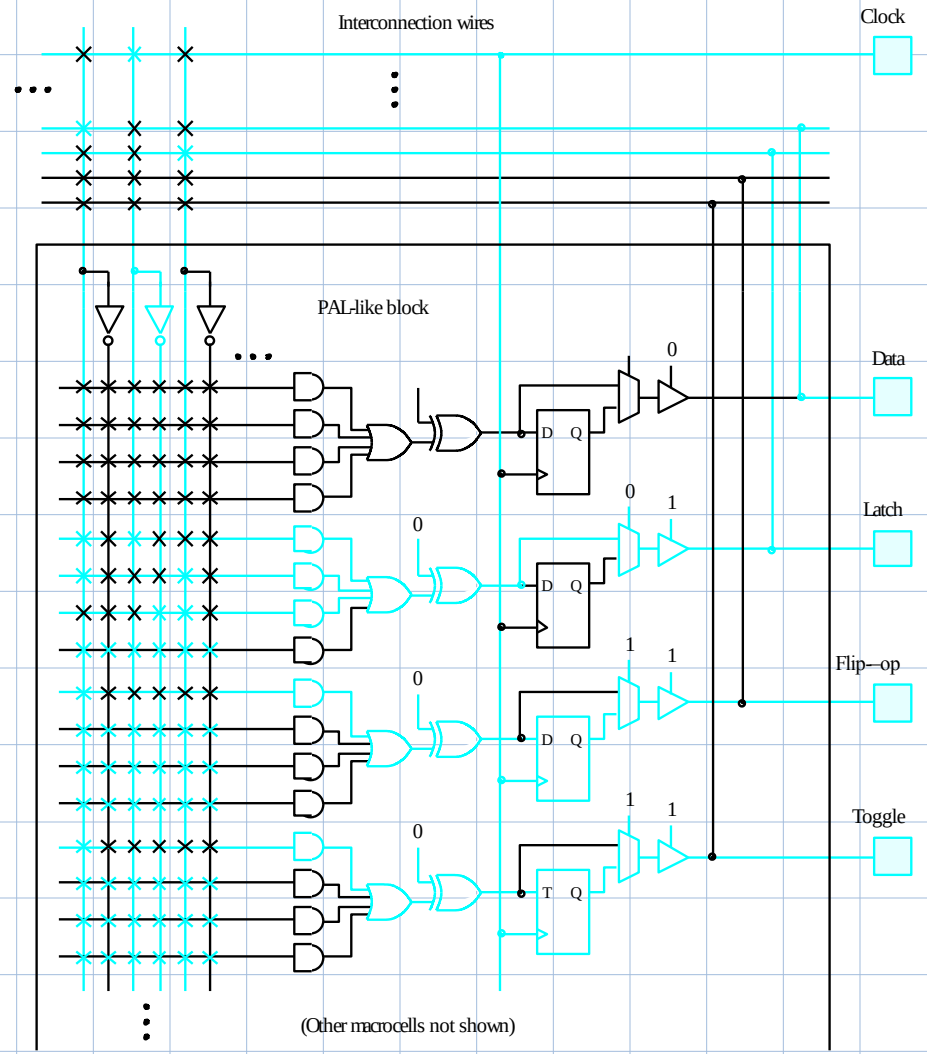
Figure 5.31 Gated D latch generated by CAD tools.

Implementación equivalente a la de NAND o NOR



Tres tipos de FF en CAD

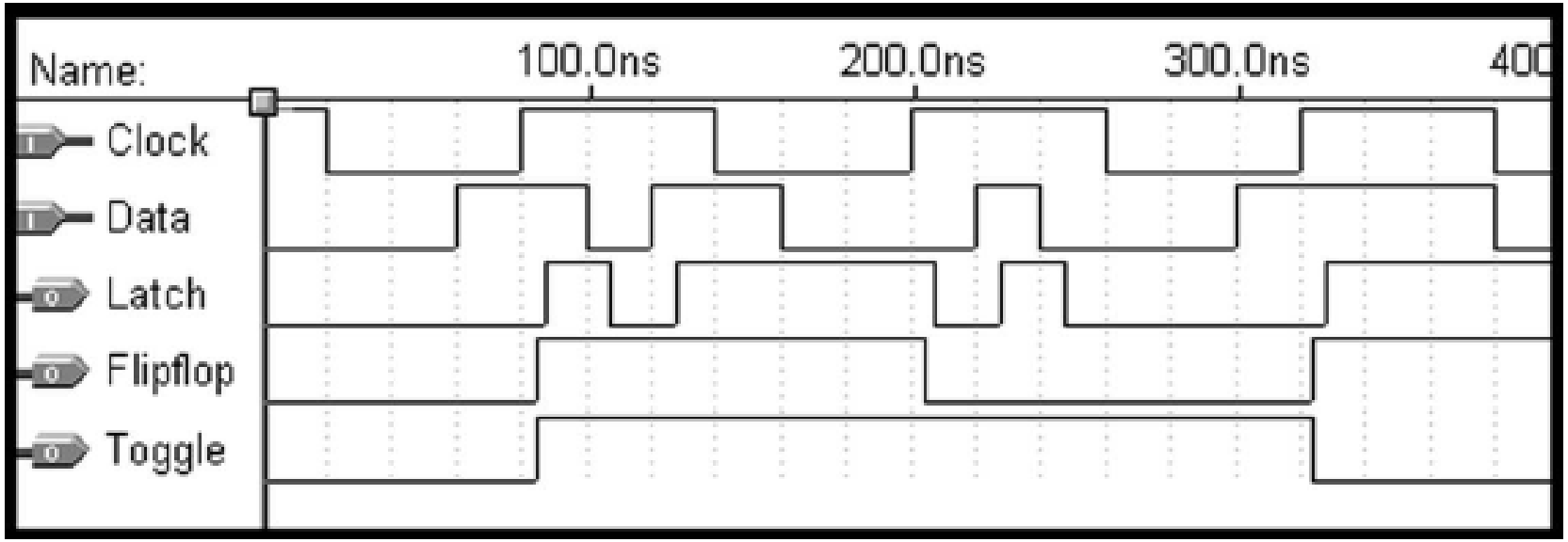
7.12.1 Elementos con memoria en esquemáticos



Aquí se observa la implementación del esquemático de tres FFs de la página anterior, en un CPLD

7.12.1 Elementos con memoria en esquemáticos

- En la figura se muestra la salida simulada
  - La salida *Latch* sigue a *Data* con un pequeño delay de propagación
- La salida *Flipflop* copia el valor de *Data* solo cuando *clock* tiene flanco de subida
- La salida *Toggle* alterna su estado cuando Clock va en subida y *Data=1*



7.12.2 Verilog y elementos con memoria

Una forma simple de especificar el comportamiento de un latch es mediante instrucciones *if-else:*

```
module D_latch (D, Clk, Q);
    input D, Clk;
    output reg Q;

    always @(D, Clk)
        if (Clk)
            Q = D;

endmodule
```

Figure 5.34 Code for a gated D latch.

Flip flop en Verilog

La palabra **posedge** indica al bloque always que se debe ejecutar en cada flanco de subida

```
module  flipflop (D, Clock, Q);
    input  D, Clock;
    output  reg  Q;

    always @(posedge Clock)
        Q = D;

endmodule
```

Figure 5.35 Code for a D flip-flop.

Como la salida Q es un registro, su valor se mantiene al menos hasta el próximo flanco

7.12.3 Asignaciones bloqueantes y no bloqueantes

- En Verilog, al signo = se le llama asignación **bloqueante**
  - Las asignaciones bloqueantes ocurren en orden
  - Si a una variable se le da un valor con una asignación bloqueante, este nuevo valor es usado para todas las declaraciones siguientes
- Por ejemplo, tenemos un código de Verilog dondedos flip-flops son conectados en cascada
- Como se emplearon asignaciones bloqueantes, Verilog asume que la variable D también entra a Q2 y que D = Q1 = Q2, conectado los flip-flops en paralelo al hacer la síntesis
  - Cuando en verdad queríamos conectarlos en cascada

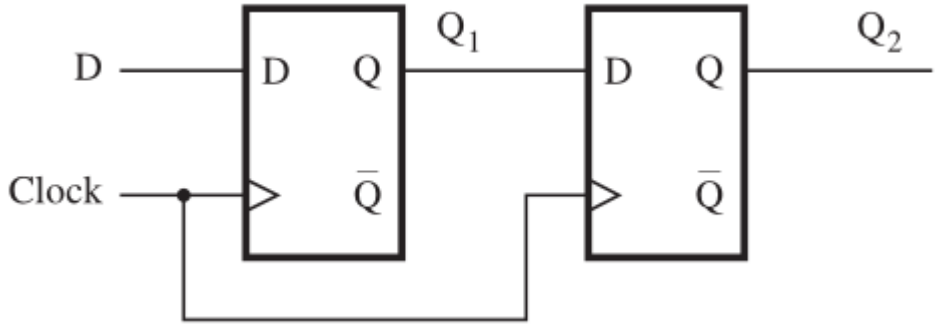
```
module example5_3 (D, Clock, Q1, Q2);
  input D, Clock;
  output reg Q1, Q2;

  always @(posedge Clock)
  begin
    Q1 = D;
    Q2 = Q1;
  end

endmodule
```

7.12.3 Asignaciones bloqueantes y no bloqueantes

- Verilog emplea la asignación <= como una asignación **no bloqueante**
  - En un bloque always, todas las asignaciones no bloqueantes son evaluadas usando los valores que había **al iniciarse el bloque**
- Mediante estas asignaciones podemos disponer de los flip flops en cascada como se buscaba en el ejemplo anterior




```
module example7_4 (D, Clock, Q1, Q2);
   input D, Clock;
   output Q1, Q2;
   reg Q1, Q2;

   always @(posedge Clock)
   begin
      Q1 <= D;
      Q2 <= Q1;
   end

endmodule
```

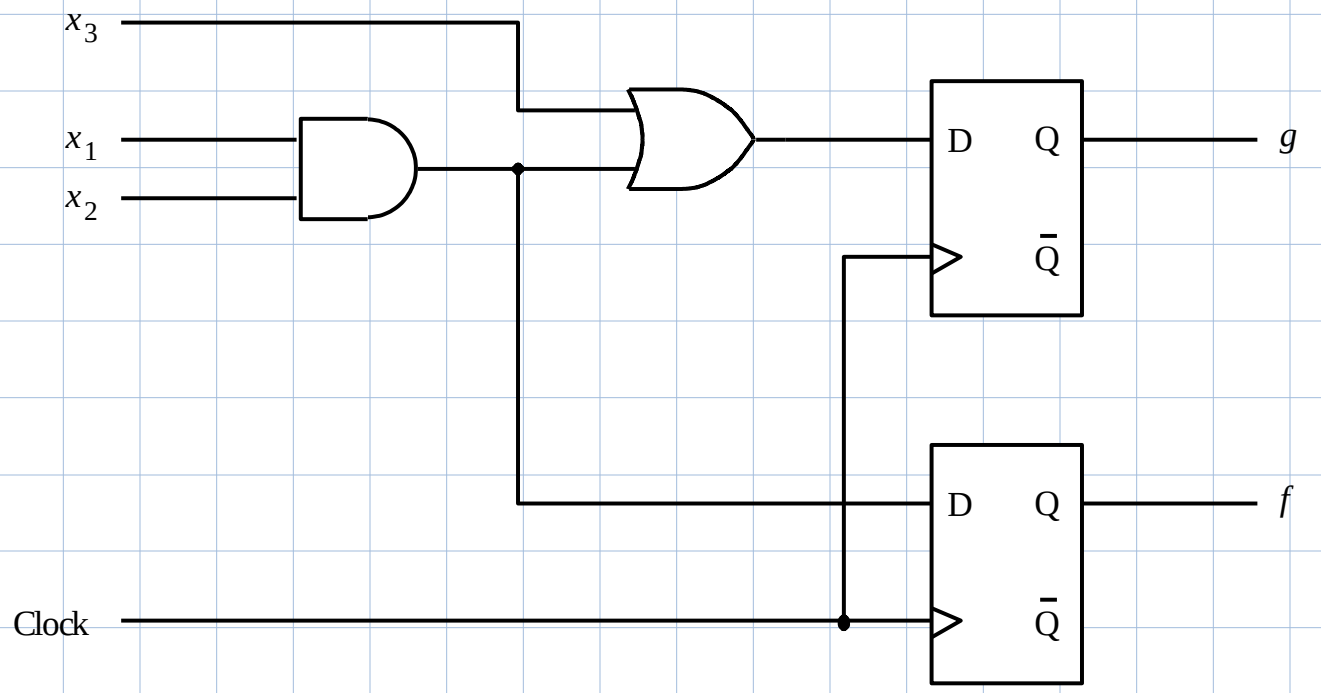
7.12.3 Asignaciones bloqueantes y no bloqueantes

Veamos ahora otro ejemplo, primero con asignaciones bloqueantes

```
module example5_5 (x1, x2, x3, Clock, f, g);
    input x1, x2, x3, Clock;
    output reg f, g;

    always @(posedge Clock)
    begin
        f = x1 & x2;
        g = f | x3;
    end

endmodule
```



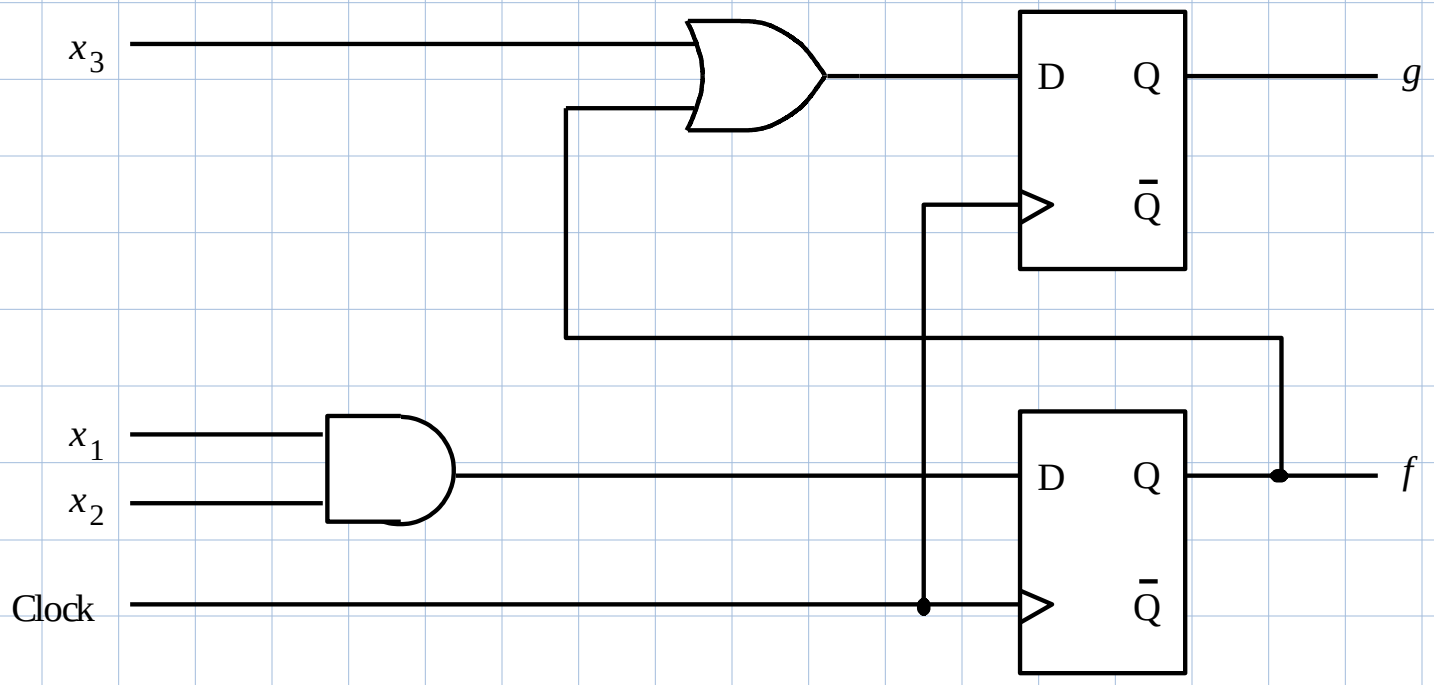
7.12.3 Asignaciones bloqueantes y no bloqueantes

Mismo ejemplo, ahora con asignaciones no bloqueantes

```
module example5_6 (x1, x2, x3, Clock, f, g);
    input x1, x2, x3, Clock;
    output reg f, g;

    always @(posedge Clock)
    begin
        f <= x1 & x2;
        g <= f | x3;
    end

endmodule
```



Conclusión: **NO USAR ASIGNACIONES BLOQUEANTES en circuitos secuenciales**

7.12.4 Asignaciones no bloqueantes para circuitos combinacionales

Pregunta obvia: ¿podemos utilizar asignaciones **no bloqueantes** para circuitos combinacionales?

```
always @(A)
begin
    f = A[1] & A[0];
    f = f | (A[2] & A[1]);
end
```

Bloqueante

```
f <= A[1] & A[0];
f <= f | (A[2] & A[1]);
```

No bloqueante

Se debe tener en cuenta dos puntos:
1. Los resultados de las asignaciones no bloqueantes son solo visibles después de evaluar el bloque always
2. Cuando hay múltiples asignaciones de una misma variable en un bloque always, se mantiene la última

Se puede, pero **MEJOR NO LO HAGAN** para evitar crear bloques secuenciales de manera accidental

7.12.5 Flip-flops con clear

Definiendo una lista de sensibilidad adecuada podemos generar un flip flop con reset asíncrono o síncrono

```
module regn (D, Clock, Resetn, Q);
    parameter n = 16;
    input [n-1:0] D;
    input Clock, Resetn;
    output reg [n-1:0] Q;

    always @(negedge Resetn or posedge Clock)
        if (!Resetn)
            Q <= 0;
        else
            Q <= D;

endmodule
```

Reset asíncrono

No es posible tener una lista de sensibilidad que combine flanco y nivel, por eso ambas son flanco

Lista es evaluada en orden

```
module flipflop (D, Clock, Resetn, Q);
    input D, Clock, Resetn;
    output reg Q;

    always @(posedge Clock)
        if (!Resetn)
            Q <= 0;
        else
            Q <= D;

endmodule
```

Reset síncrono

7.13 Registros y contadores en Verilog

Registro de n bits: Como los circuitos lógicos suelen requerir registros de tamaño n, es posible construirlos en forma vectorizada

```
module regn (D, Clock, Resetn, Q);
    parameter n = 16;
    input [n-1:0] D;
    input Clock, Resetn;
    output reg [n-1:0] Q;

    always @(negedge Resetn, posedge Clock)
        if (!Resetn)
            Q <= 0;
        else
            Q <= D;

endmodule
```

7.13 Registros y contadores en Verilog

Registro de desplazamiento de 4 bits con carga paralela: Una alternativa para esto es implementar cuatro flip flops con multiplexores, y luego instanciarlos en un bloque principal:

```
module muxdff (D0, D1, Sel, Clock, Q);
    input D0, D1, Sel, Clock;
    output reg Q;

    always @(posedge Clock)
        if (!Sel)
            Q <= D0;
        else
            Q <= D1;

endmodule
```

```
module shift4 (R, L, w, Clock, Q);
    input [3:0] R;
    input L, w, Clock;
    output [3:0] Q;
    wire [3:0] Q;

    muxdff Stage3 (w, R[3], L, Clock, Q[3]);
    muxdff Stage2 (Q[3], R[2], L, Clock, Q[2]);
    muxdff Stage1 (Q[2], R[1], L, Clock, Q[1]);
    muxdff Stage0 (Q[1], R[0], L, Clock, Q[0]);

endmodule
```

7.13 Registros y contadores en Verilog

```
module shift4 (R, L, w, Clock, Q);
    input [3:0] R;
    input L, w, Clock;
    output reg [3:0] Q;

    always @(posedge Clock)
        if (L)
            Q <= R;
        else
        begin
            Q[0] <= Q[1];
            Q[1] <= Q[2];
            Q[2] <= Q[3];
            Q[3] <= w;
        end

endmodule
```

- Otra forma de implementar este registro de desplazamiento es definiendo todos los flip flops en un solo bloque always
- Al igual que el bloque anterior, el bloque tiene funcionalidad de carga paralela cuando L = 1

7.13 Registros y contadores en Verilog

```
module shiftn (R, L, w, Clock, Q);
    parameter n = 16;
    input [n-1:0] R;
    input L, w, Clock;
    output reg [n-1:0] Q;
    integer k;

    always @(posedge Clock)
        if (L)
            Q <= R;
        else
        begin
            for (k = 0; k < n-1; k = k+1)
                Q[k] <= Q[k+1];
                Q[n-1] <= w;
        end

endmodule
```

Registro de desplazamiento de n bits:

- Similar a los registros de desplazamiento ya vistos, solo que en este caso R y Q están parametrizados para n
- Se utiliza un loop for para cambiar los registros de valor

7.13 Registros y contadores en Verilog

- **Contador de 4 bits ascendente:** Incorpora funcionalidades de enable (E) y reset asíncrono (Resetn)
- A la derecha se muestra el mismo contador con funcionalidad de carga paralela (L) que carga una entrada R

```
module upcount (Resetn, Clock, E, Q);
    input Resetn, Clock, E;
    output reg [3:0] Q;

    always @(negedge Resetn, posedge Clock)
        if (!Resetn)
            Q <= 0;
        else if (E)
            Q <= Q + 1;

endmodule
```

```
module upcount (R, Resetn, Clock, E, L, Q);
    input [3:0] R;
    input Resetn, Clock, E, L;
    output reg [3:0] Q;

    always @(negedge Resetn, posedge Clock)
        if (!Resetn)
            Q <= 0;
        else if (L)
            Q <= R;
        else if (E)
            Q <= Q + 1;

endmodule
```

7.13 Registros y contadores en Verilog

- **Contador descendente de n bits y carga paralela:** Incorpora una parametrización de la cantidad de bits
  - Si L = 1, se carga la entrada R al contador
  - si L = 0, el contador empieza a bajar

```
module downcount (R, Clock, E, L, Q);
    parameter n = 8;
    input [n-1:0] R;
    input Clock, L, E;
    output reg [n-1:0] Q;

    always @(posedge Clock)
        if (L)
            Q <= R;
        else if (E)
            Q <= Q - 1;

endmodule
```

7.13 Registros y contadores en Verilog

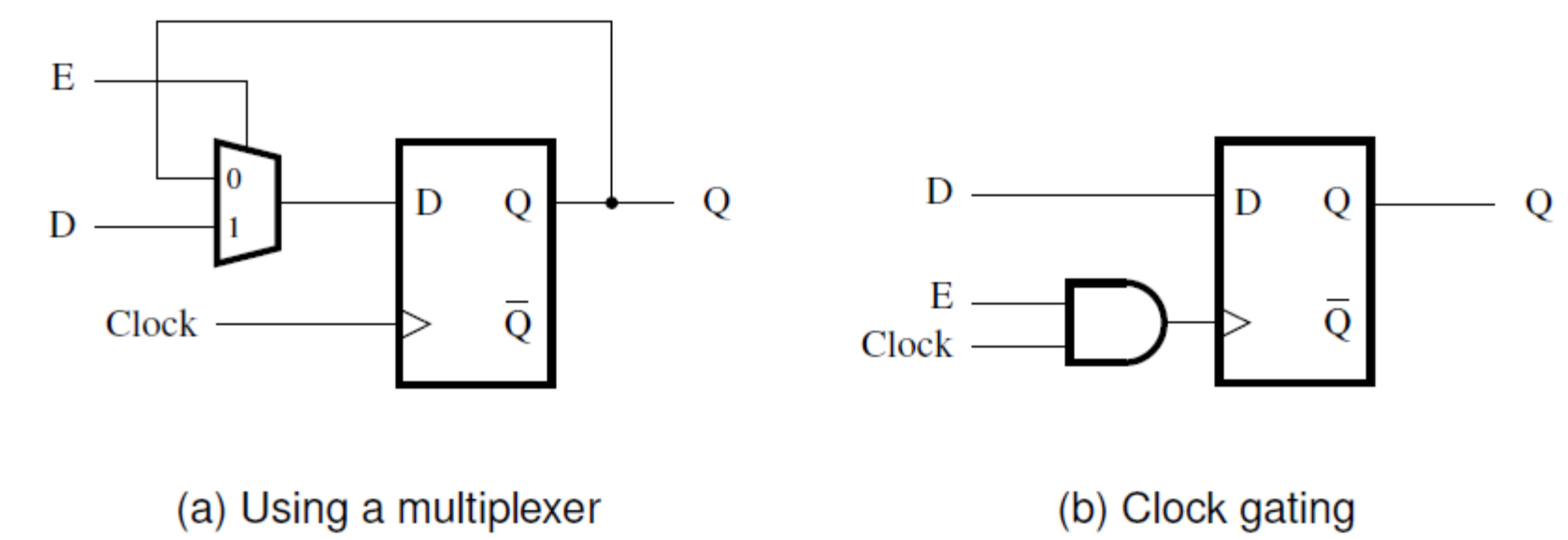
Contador ascendente y descendente: Incorpora una entrada *updown* que le indica al contador en qué dirección contar

```
module updowncount (R, Clock, L, E, up_down, Q);
    parameter n = 8;
    input [n-1:0] R;
    input Clock, L, E, up_down;
    output reg [n-1:0] Q;
    always @(posedge Clock)
        if (L)
            Q <= R;
        else if (E)
            Q <= Q + (up_down ? 1 : -1);

endmodule
```

Flip flops y registros con función enable

- A veces es necesario mantener el estado de un flip flop aun cuando estamos en flanco de subida de un clock
- Para esto podemos implementar un enable en dichos flip flops ocupando o un multiplexor o una compuerta AND



(a) Using a multiplexer

(b) Clock gating

Flip flops y registros con función enable

- A la izquierda, el código Verilog de un flip flop con función Enable
- Esto puede ser extendido también para disponer de un registro con función enable de tamaño n

```
module rege (D, Clock, Resetn, E, Q);
    input D, Clock, Resetn, E;
    output reg Q;

    always @(posedge Clock, negedge Resetn)
        if (Resetn == 0)
            Q <= 0;
        else if (E)
            Q <= D;

endmodule
```

```
module regne (R, Clock, Resetn, E, Q);
    parameter n = 8;
    input [n-1:0] R;
    input Clock, Resetn, E;
    output reg [n-1:0] Q;

    always @(posedge Clock,  negedge Resetn)
        if (Resetn == 0)
            Q <= 0;
        else if (E)
            Q <= R;

endmodule
```

Registros de desplazamiento de bits con función enable

```
module shiftrne (R, L, E, w, Clock, Q);
    parameter n = 4;
    input [n-1:0] R;
    input L, E, w, Clock;
    output reg [n-1:0] Q;
    integer k;

    always @(posedge Clock)
    begin
        if (L)
          Q <= R;
        else if (E)
        begin
        Q[n-1] <= w;
        for (k = n-2; k >= 0; k = k-1)
          Q[k] <= Q[k+1];
        end
    end
endmodule
```



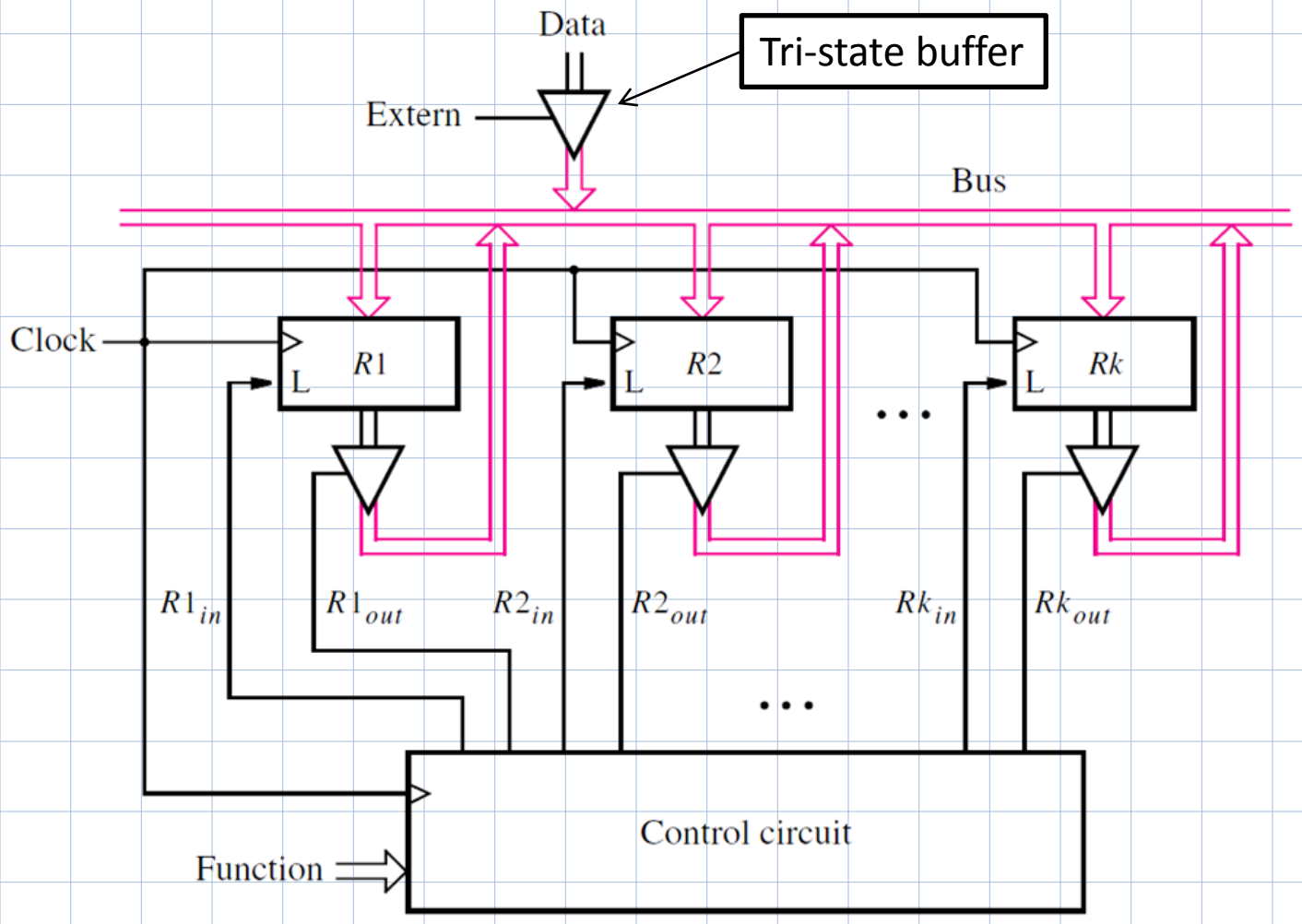
7.14 – 7.15

Ejemplos de diseño

angel@uc.cl

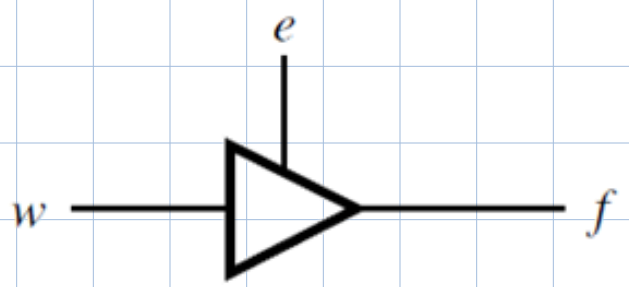
IEE2713 Sistemas Digitales

7.14.1 Estructura de bus

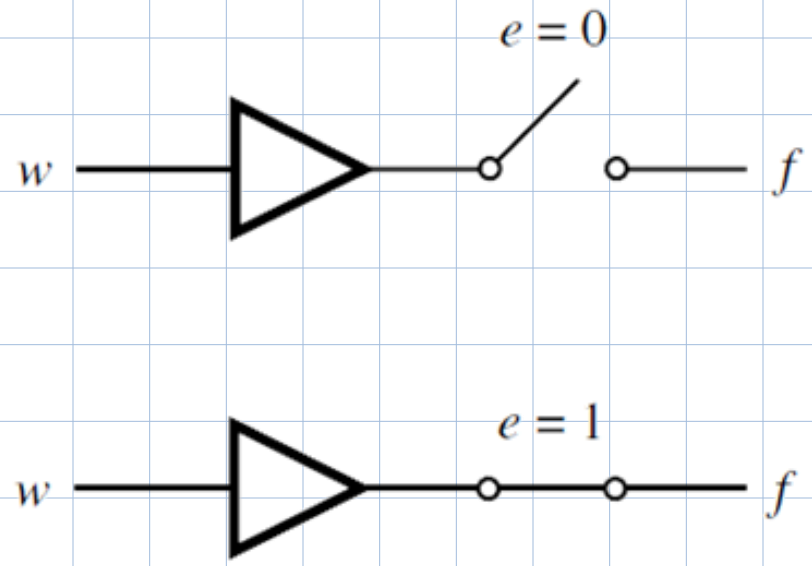


- Los sistemas digitales generalmente emplean un conjunto de registros para almacenar datos de n bits
- Todos los registros van conectados al mismo conjunto de n cables
  - Este conjunto de n cables se denomina **bus**
- El bus permite transferir datos entre registros o hacia otros circuitos
- Sólo un bloque puede poner datos en el bus a la vez

Tri-state buffer - comportamiento



(a) Symbol



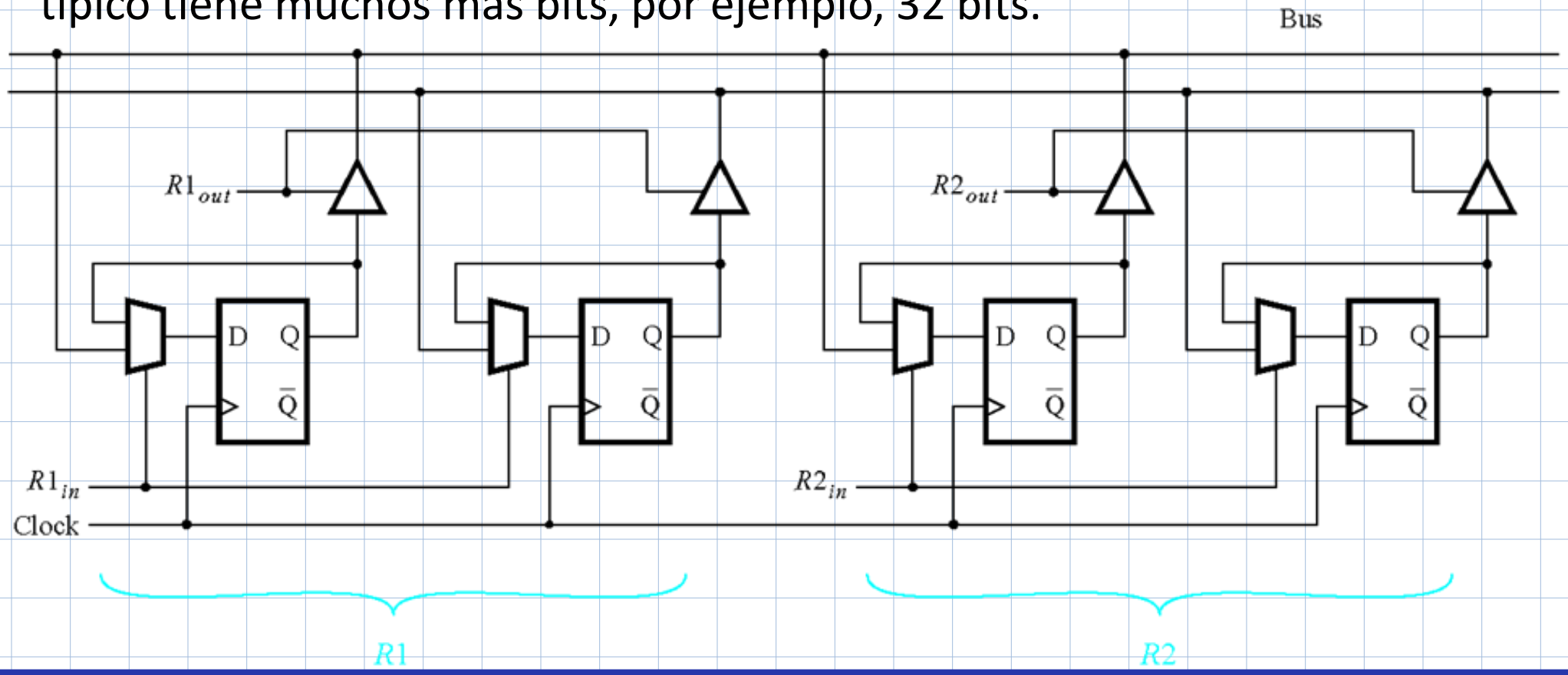
(b) Equivalent circuit

e	w	f
0	0	Z
0	1	Z
1	0	0
1	1	1

(c) Truth table

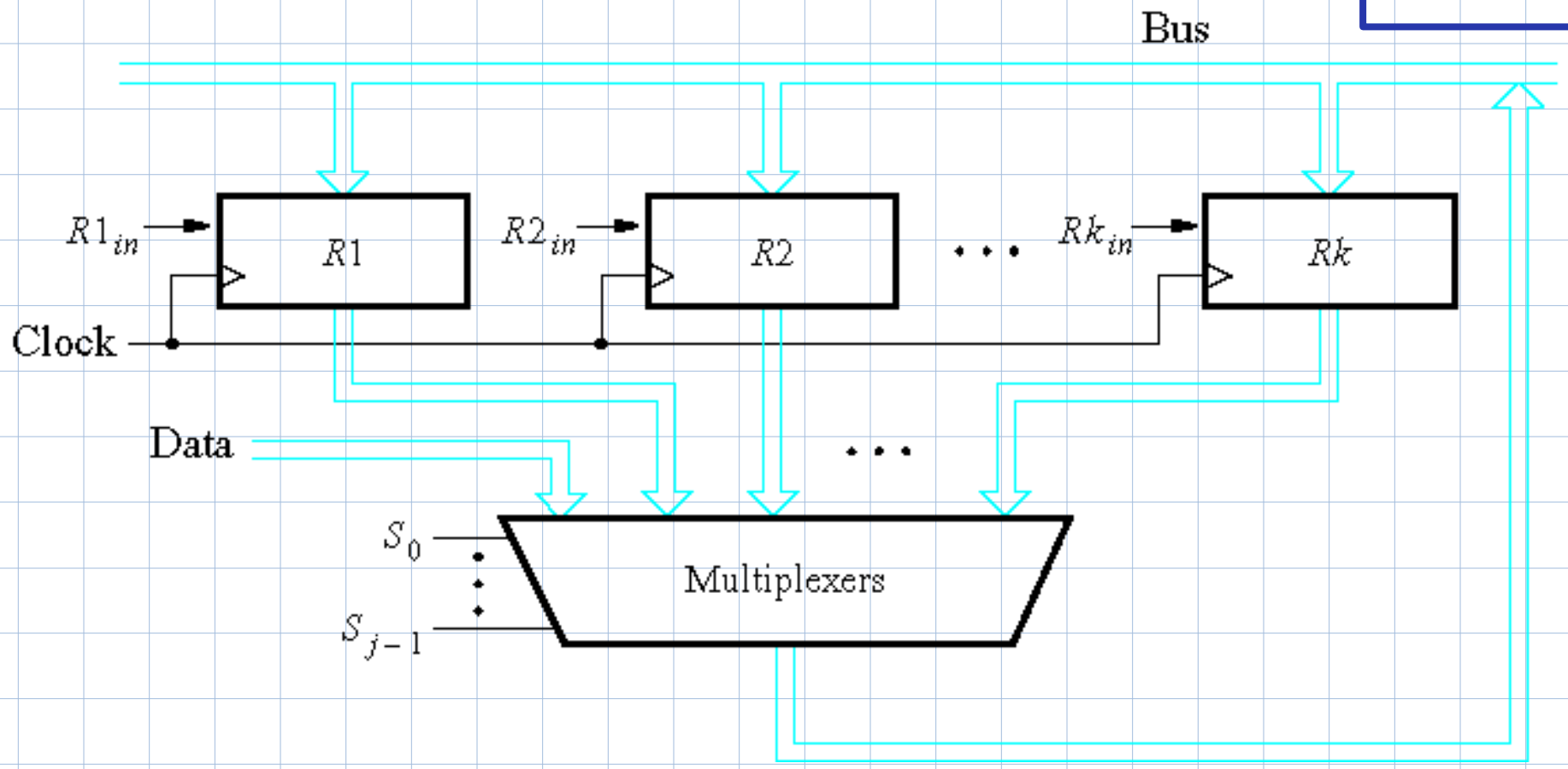
Una posible implementación del bus

Este bus tiene sólo **2 bits** (MSB y LSB) y **2 registros R_1 y R_2** conectados al bus, cada uno con su MSB y su LSB. Un bus típico tiene muchos más bits, por ejemplo, 32 bits.

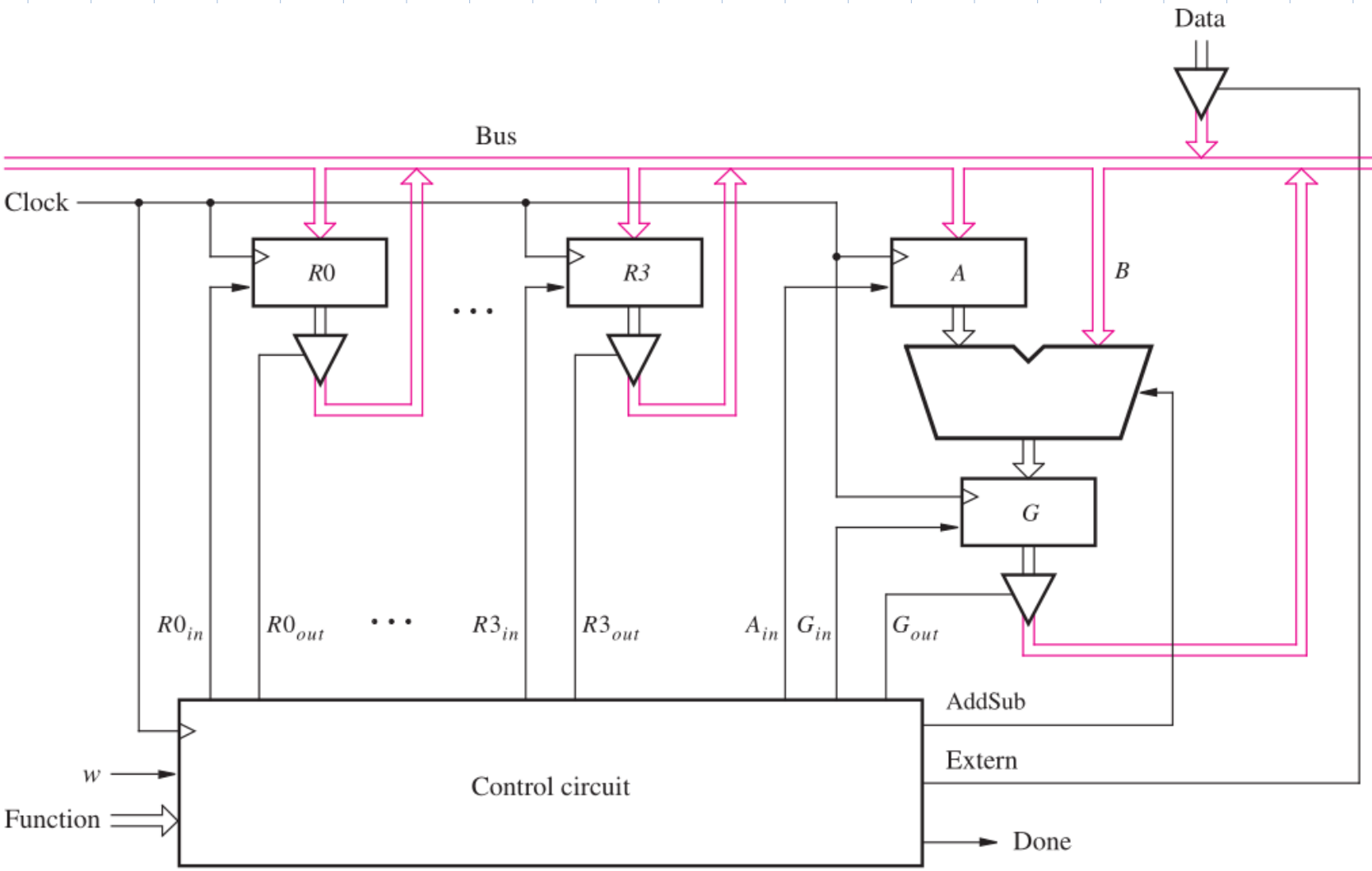


Una implementación alternativa empleando MUX en lugar de tri-state

El MUX controla cuál salida va al bus y evita choques



7.14.2 Procesador simple



- Operaciones sobre cuatro registros R
- Recibe datos desde afuera (*Extern*)
- Carga registros, suma y resta
- Registro G guarda resultado

Operaciones del procesador

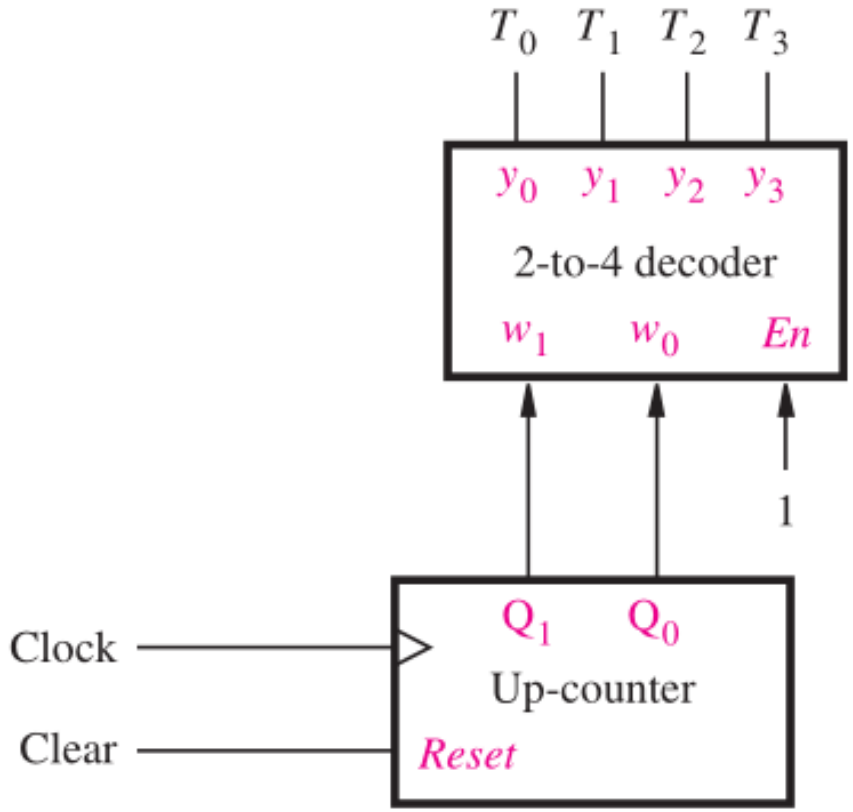
- *AddSub* elige operación suma o resta
- Registro G almacena salida
- En la tabla, [Rx] es “Contenidos de Rx”
- Operaciones *Load* y *Move* requieren un solo ciclo de reloj (un solo paso)
- Operaciones *Add* y *Sub* requieren 3 ciclos de reloj:
  - A ← [Rx]
  - B (Bus) ← [Ry]
    - operación combinacional es realizada
  - Rx ← [G]

Table 7.2 Operations performed in the processor.

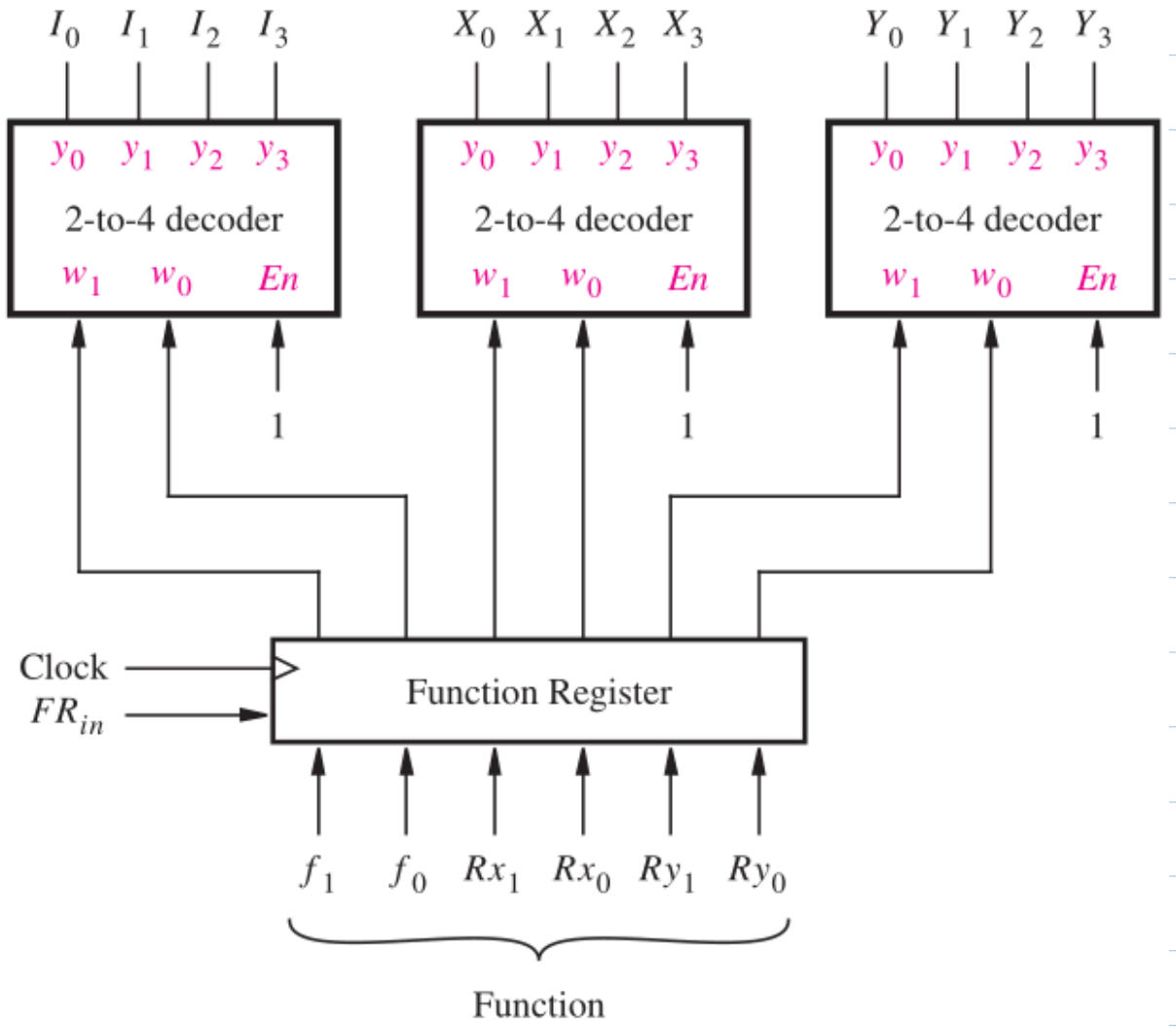
Operation	Function performed
Load *Rx*, *Data*	$Rx \leftarrow Data$
Move *Rx*, *Ry*	$Rx \leftarrow [Ry]$
Add *Rx*, *Ry*	$Rx \leftarrow [Rx] + [Ry]$
Sub *Rx*, *Ry*	$Rx \leftarrow [Rx] - [Ry]$

Circuito de control del procesador

- La entrada *Function* indica la operación
- La operación es iniciada al hacer *w* = 1, luego al terminar el procesador arroja *Done*
- Dado que las operaciones *Add* y *Sub* requieren tres ciclos de reloj, es posible emplear un contador de 2 bits para controlar la operación (ver esquemático)
- T_0 es activada cuando no hay operación en curso
- Varias señales de control son generadas en los pasos T_1-T_3, circuito a continuación



Registro de función y decodificadores



- Cada operación está especificada por 6 bits (*Function*)
  - f_1, f_0 especifican la operación (*Load*, *Move*, *Add*, *Sub*)
  - Los otros términos identifican los operandos
  - Un circuito combinacional toma las salidas I, X e Y, y generan las señales *Extern*, *Done*, A_{in}, G_{in}, G_{out}, *AddSub*, *Clear*, FR_{in}, R_{in} y R_{out}

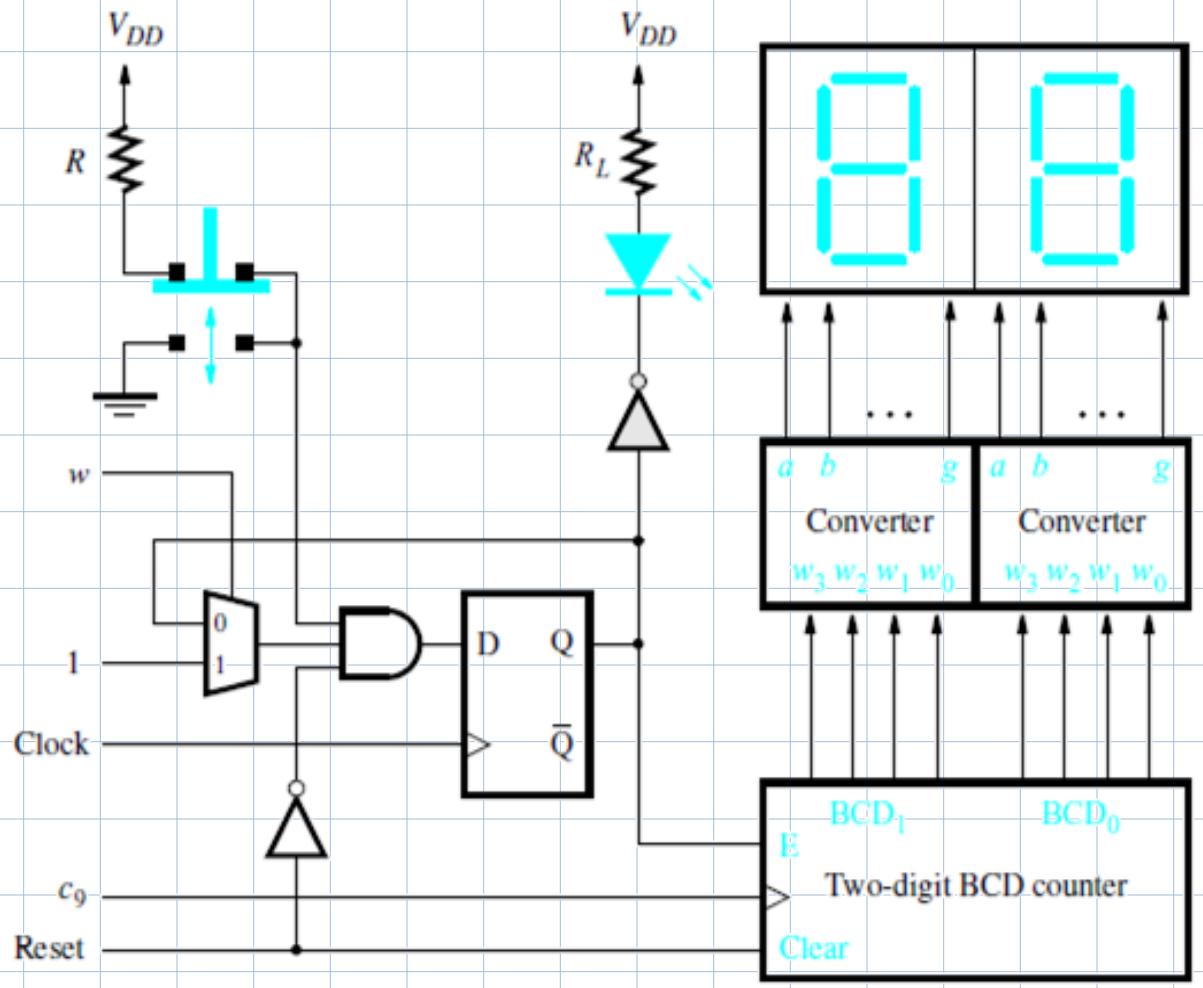
Algunas de las señales que faltan…

- *Clear* es activada para que el contador esté en 00 cuando *w* = 0 y ninguna operación esté en ejecución
- También debe ser activada al terminar una operación, para resetear el contador
- Entonces $Clear = \overline{w}\, T_0 + Done$
- FR_{in} es utilizada para cargar la función en el registro *Function* cuando *w* cambia a 1

$$FR_{in} = wT_0$$

- Y así sucesivamente con el resto de las señales
  - Revisar libro para más detalles, incluyendo Verilog

7.14.3 Temporizador de reacción



(c) Push-button switch, LED, and 7-segment displays

- Circuito que mide el tiempo de reacción de una persona, desde que observa el encendido de un LED hasta que presiona un botón
- C_9 es un reloj lento generado por un divisor de reloj
- El FF comanda el encendido del LED y la habilitación del contador
- Más detalles en el libro

Código Verilog del ejemplo

```
module reaction (Clock, Reset, c9, w, Pushn, LEDn, Digit1, Digit0);
    input Clock, Reset, c9, w, Pushn;
    output wire LEDn;
    output wire [1:7] Digit1, Digit0;
    reg LED;
    wire [3:0] BCD1, BCD0;

    always @(posedge Clock)
    begin
        if (!Pushn | | Reset)
            LED <= 0;
        else if (w)
            LED <= 1;
    end

    assign LEDn = ~LED;
    BCDcount counter (c9, Reset, LED, BCD1, BCD0);
    seg7 seg1 (BCD1, Digit1);
    seg7 seg0 (BCD0, Digit0);

endmodule
```

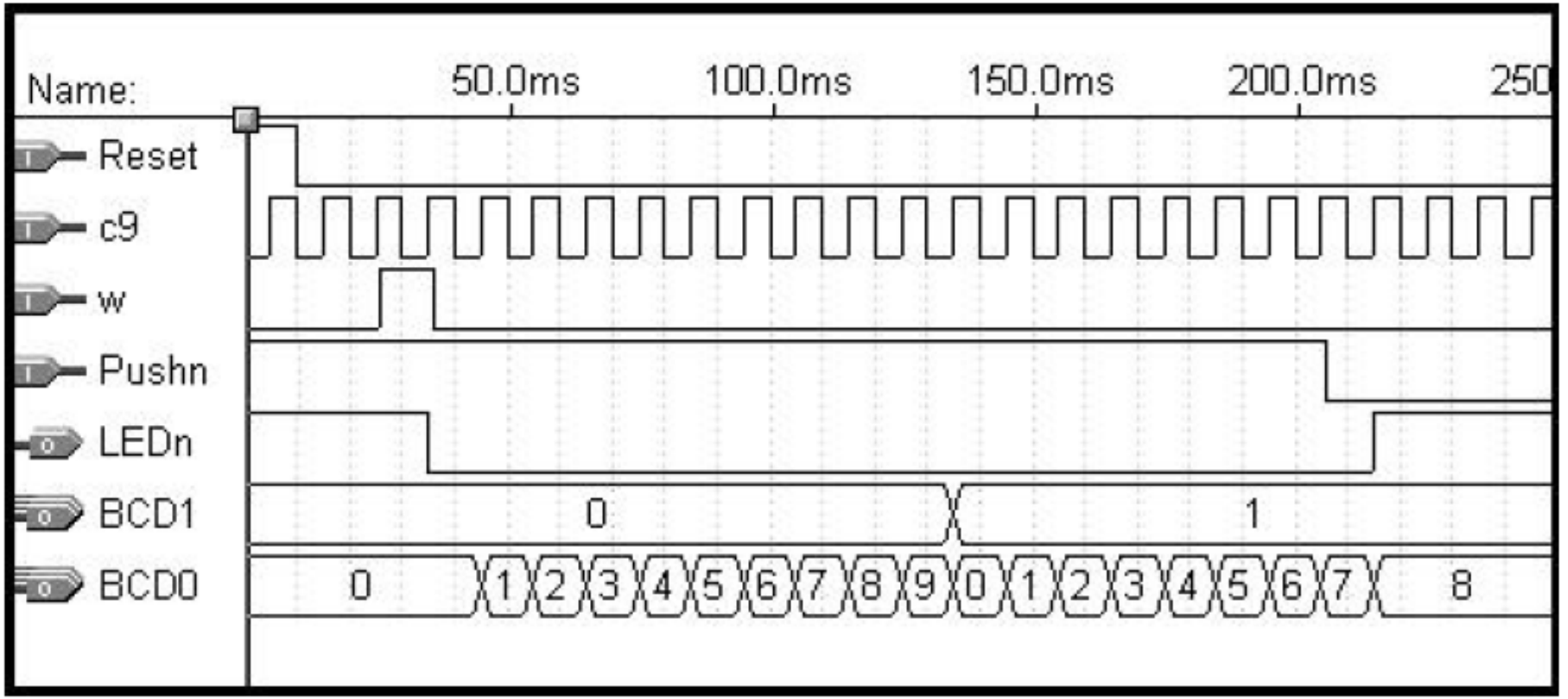
```
module seg7 (bcd, leds);
    input [3:0] bcd;
    output reg [1:7] leds;

    always @(bcd)
        case (bcd)    //abcdefg
            0: leds = 7'b1111110;
            1: leds = 7'b0110000;
            2: leds = 7'b1101101;
            3: leds = 7'b1111001;
            4: leds = 7'b0110011;
            5: leds = 7'b1011011;
            6: leds = 7'b1011111;
            7: leds = 7'b1110000;
            8: leds = 7'b1111111;
            9: leds = 7'b1111011;
            default: leds = 7'bx;
        endcase

endmodule
```

```
module BCDcount (Clock, Clear, E, BCD1, BCD0);
    input Clock, Clear, E;
    output reg [3:0] BCD1, BCD0;
    always @(posedge Clock)
    begin
        if (Clear)
        begin
            BCD1 <= 0;
            BCD0 <= 0;
        end
        else if (E)
            if (BCD0 == 4'b1001)
            begin
                    BCD0 <= 0;
                    if (BCD1 == 4'b1001)
                        BCD1 <= 0;
                    else
                        BCD1 <= BCD1 + 1;
            end
            else
                BCD0 <= BCD0 + 1;
    end
endmodule
```

Diagrama de tiempos del ejemplo



7.14.4 Código a nivel de transferencia de registros (RTL)

- Ya sabemos bastante Verilog
  - Case, if-else, for, etc.
- Podemos escribir código de comportamiento a alto nivel
  - Podemos olvidarnos de compuertas lógicas y los detalles de implementación
  - No es la idea
    - No contribuye al entendimiento
    - No ayuda a diseñar mejor
- Mejor diseño estilo RTL (register-transfer level) es jerárquico
  - Modela circuitos síncronos en términos del flujo de señales de datos entre registros, y las operaciones lógicas involucradas
    - Bloques pequeños sencillos
    - Interconexión arbitrariamente compleja
    - https://en.wikipedia.org/wiki/Register-transfer_level