

10. Diseño de sistemas digitales

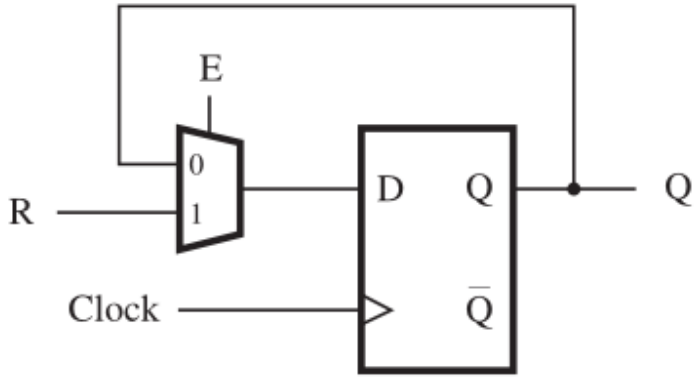
angel@uc.cl

IEE2713 Sistemas Digitales

10.1 Bloques circuitales

- Ya aprendimos cómo diseñar diferentes bloques circuitales tales como flip-flops, contadores, registros, lógica combinacional, etc.
- En este capítulo presentaremos ejemplos de circuitos más complejos que pueden ser construidos con bloques más sencillos
  - Estos circuitos más complejos forman un sistema digital
- Aprenderemos técnicas útiles en diseños de cualquier tamaño
- Un sistema digital tiene dos partes:
  - Camino de la información (datapath): guarda, manipula y transfiere datos entre diferentes partes
  - Control circuit: FSM que controla la operación del camino de la información

10.1.1 Flip-flops y registros con habilitación



¿Por qué no usamos un enable sobre el reloj mediante un AND lógico?

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

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

endmodule
```

FF D de 1 bit

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

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

endmodule
```

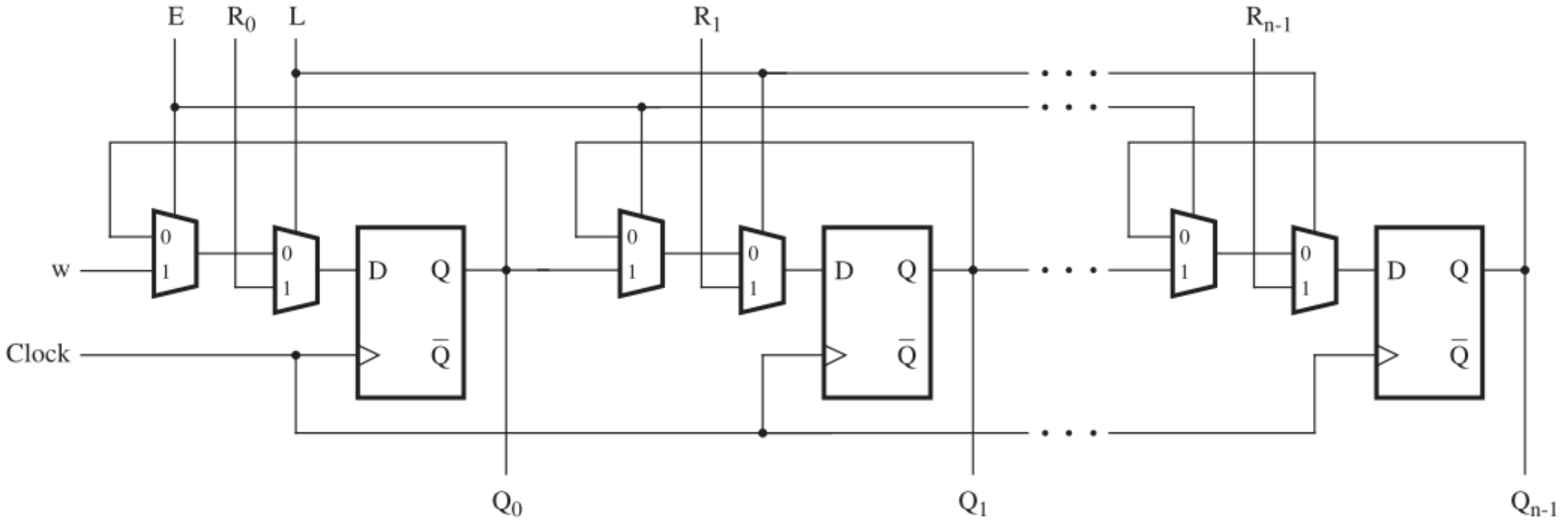
FF D de múltiples bit

10.1.2 Shift registers con habilitación

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

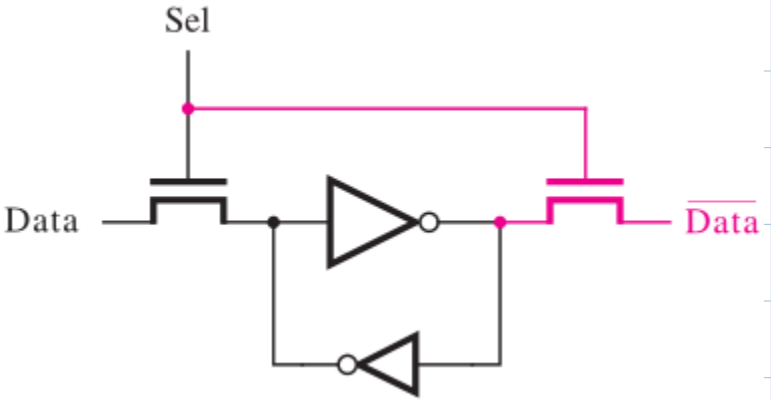
always @(posedge Clock)
begin
    if (L)
        Q <= R;
    else if (E)
        begin
            Q[0] <= w;
            for (k = 1; k < n; k = k+1)
                Q[k] <= Q[k–1];
        end
end

endmodule
```



10.1.3 Memoria RAM estática (SRAM)

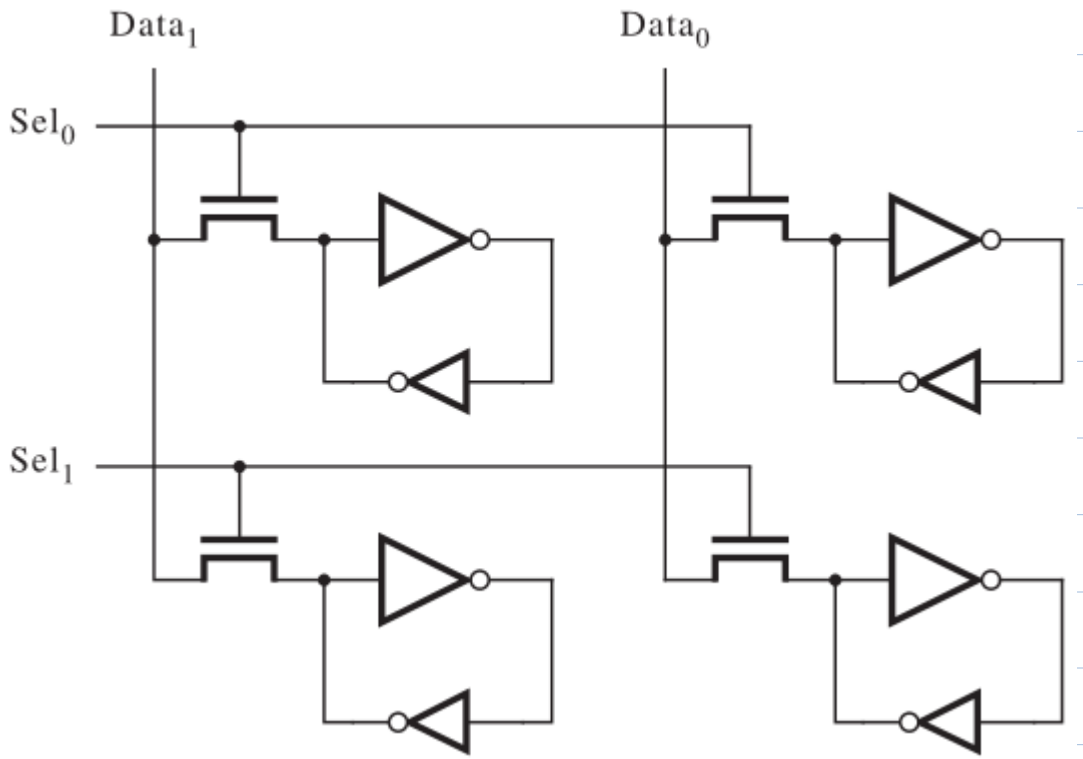
- No es muy práctico guardar mucha información en flip-flops
  - Ocupa mucho espacio y requiere muchos circuitos adicionales
- Mejor usamos un bloque de SRAM
  - Arreglo bidimensional de celdas de memoria volátil
  - Cada celda puede guardar un bit
  - Razón de aspecto m x n, m = número de palabras, n = bits por palabra
  - Accesamos (leemos o escribimos) con Sel = 1
  - Para leer, el bus Data debe estar en alta impedancia



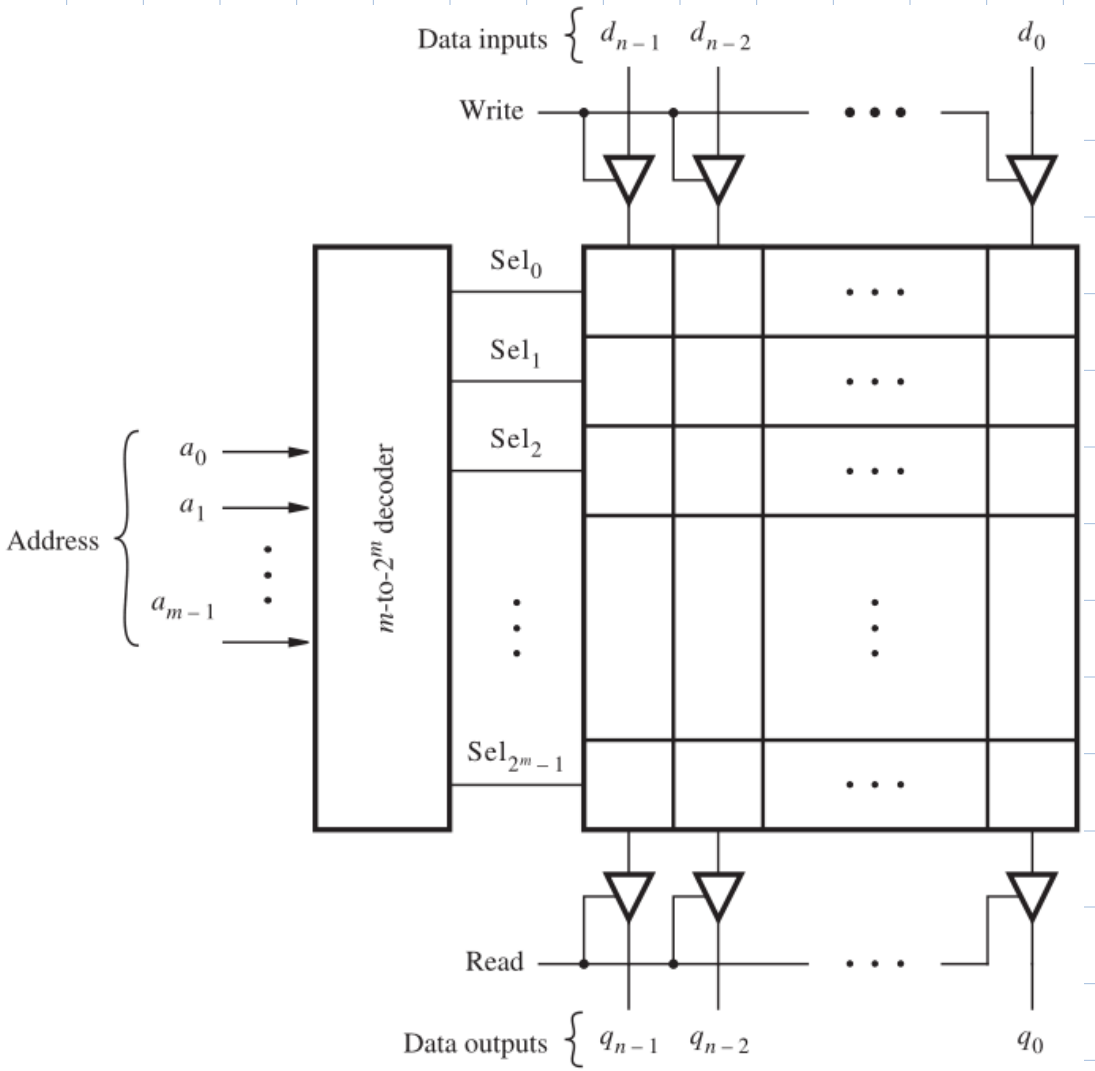
Notar que un inversor es más débil. Esto ayuda a que Data le gane. El transistor rosado podría no ser implementado.

10.1.3 Memoria RAM estática (SRAM)

- Un bloque de SRAM tiene un arreglo de celdas
  - Ejemplo: un arreglo de 2 x 2
- Cada palabra tiene una dirección de memoria (address)
- Para elegir la palabra que queremos leer o escribir usamos el selector
- Data es un bus al que van conectadas todas las celdas de memoria



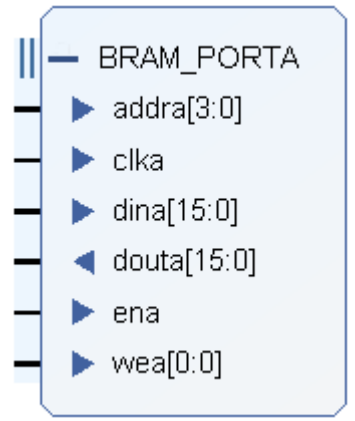
10.1.3 Memoria RAM estática (SRAM)



- Para habilitar el acceso a memoria usamos un decodificador one-hot que activa el selector correspondiente a la dirección
- Las entradas Read y Write activan los buffers tri-state
  - No pueden estar activas al mismo tiempo
- Hay libros completos dedicados a diseño de memorias digitales

10.1.4 SRAM en PLDs

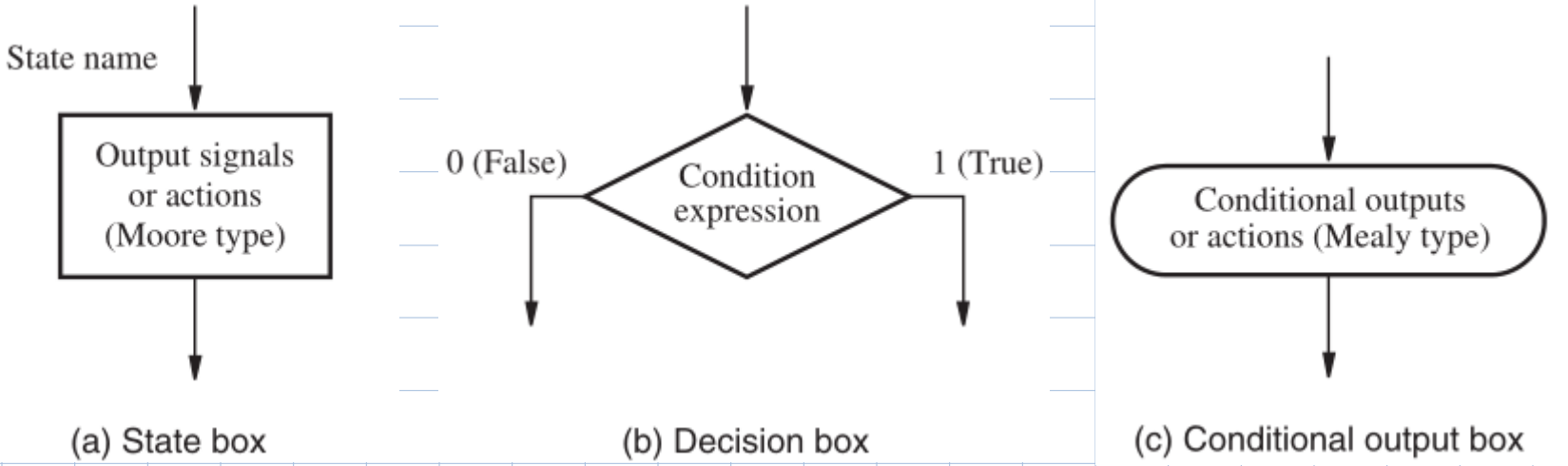
- Algunos PLDs contienen bloques de SRAM, e.g., 2k
- Software facilita el diseño
  - Ejemplo: en Vivado hay un “block memory generator” en IP Catalog para la creación automática de bloques de memoria
  - Aguanta memoria de uno o dos puertos
    - Permite leer o escribir, o leer y escribir al mismo tiempo
  - Razón de aspecto configurable



10.2 Ejemplos de diseño

Algorithmic State Machine (ASM – cap 8.10)

- Diagrama de flujo que permite describir el comportamiento de una FSM
- Bloques incluyen estado, decisión y salidas condicionales
- A diferencia de un diagrama de flujo, una ASM implica temporalidad



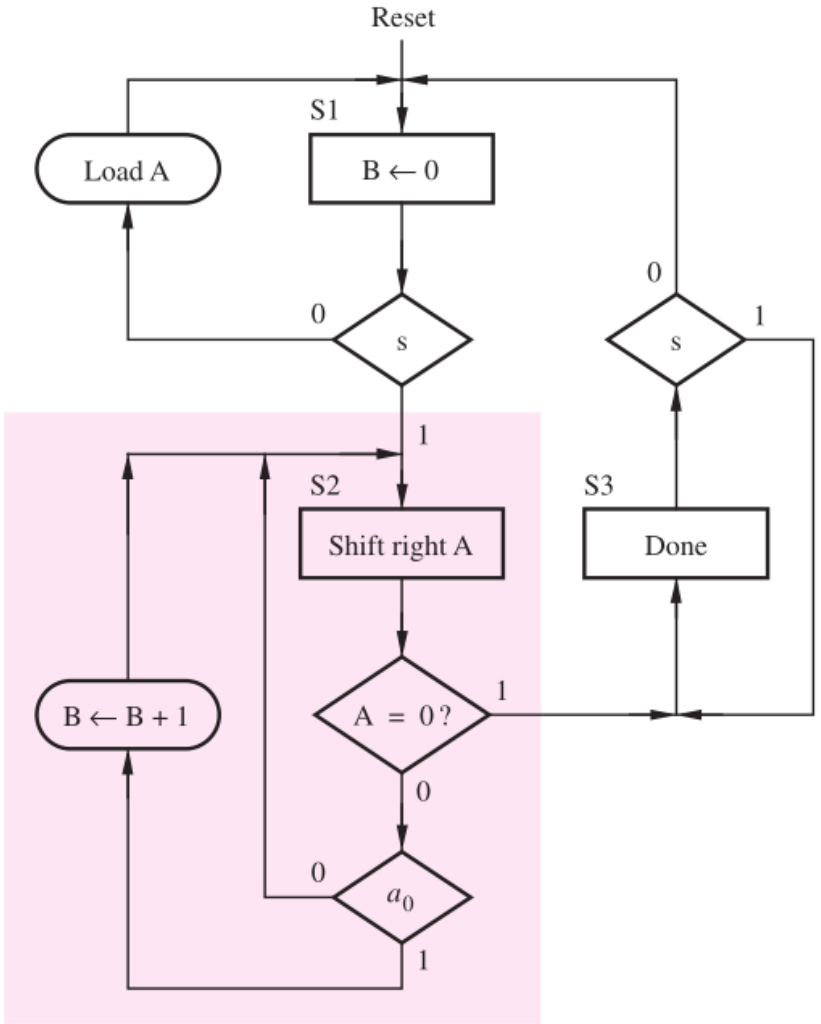
10.2.1 Circuito para contar bits

- Supongamos que queremos contar bits que valen 1 en un registro A de muchos bits
- Podemos comenzar con pseudocódigo para el algoritmo:

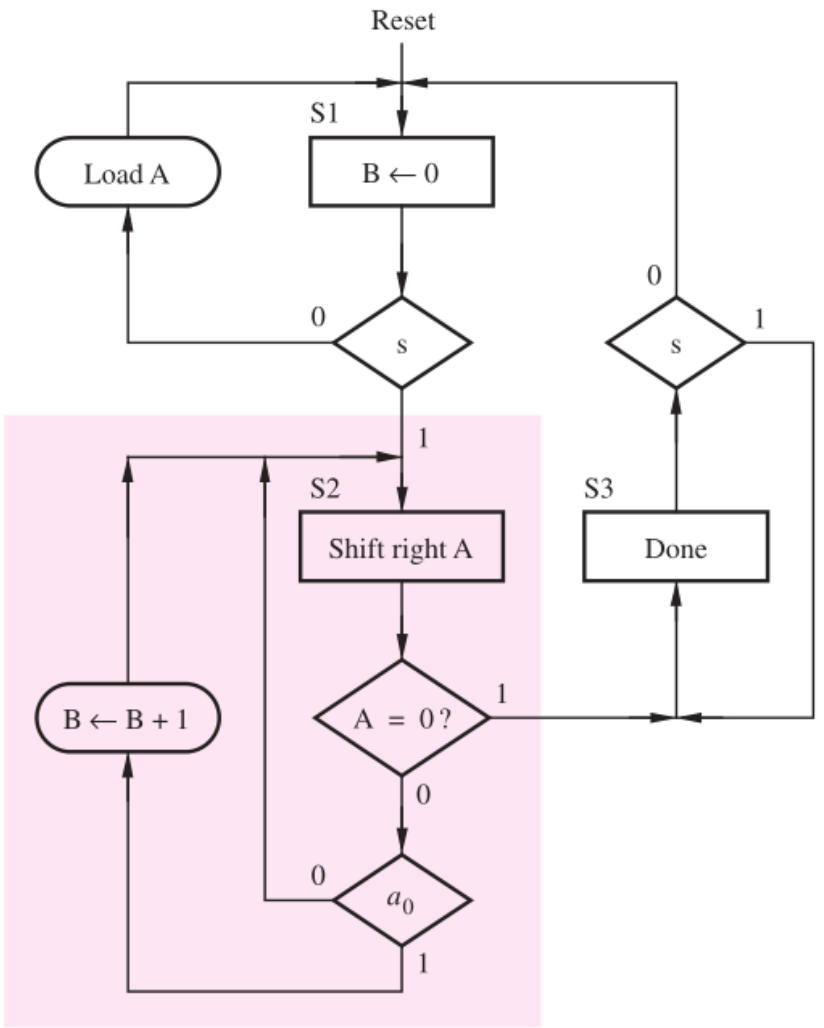
```
B = 0 ;
while A ≠ 0 do
      if a0 = 1 then
            B = B + 1 ;
      end if ;
      Right-shift A ;
end while ;
```

- La respuesta queda almacenada en registro B
- Veamos cómo queda el diagrama de flujo:

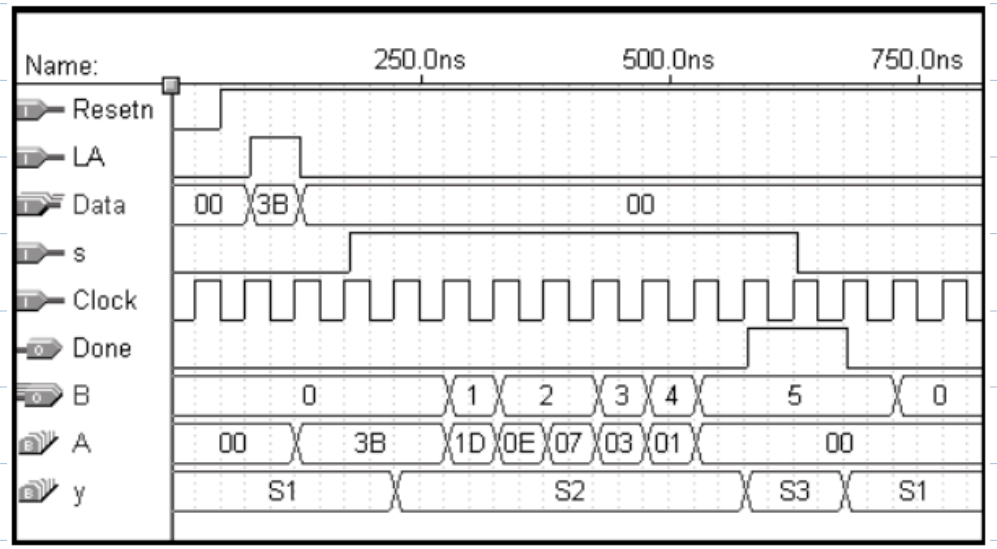
10.2.1 Circuito para contar bits



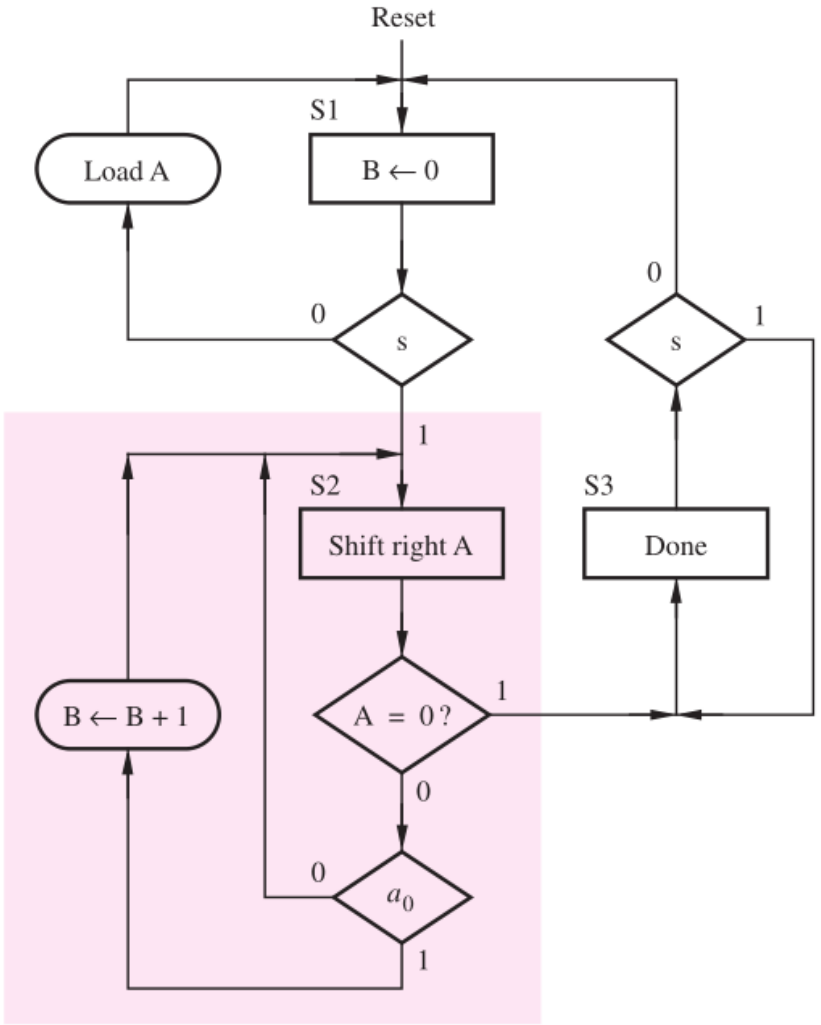
10.2.2 Información temporal implicada por diagrama de la ASM



- La ASM lleva temporalidad
- Los cambios ocurren en forma síncrona, con un reloj
- Todos los flip-flops, en todas las partes del diagrama de la ASM, cambian al mismo tiempo
- Veremos cómo esto afecta la forma de diseñar

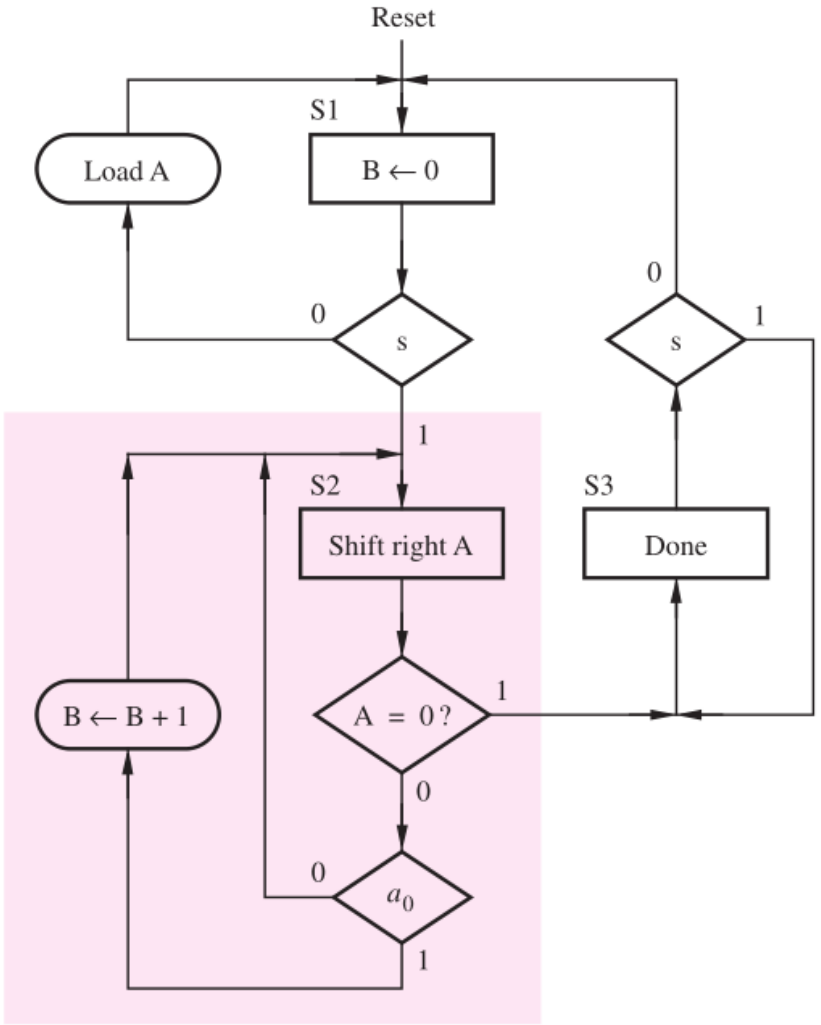


10.2.2 Información temporal implicada por diagrama de la ASM

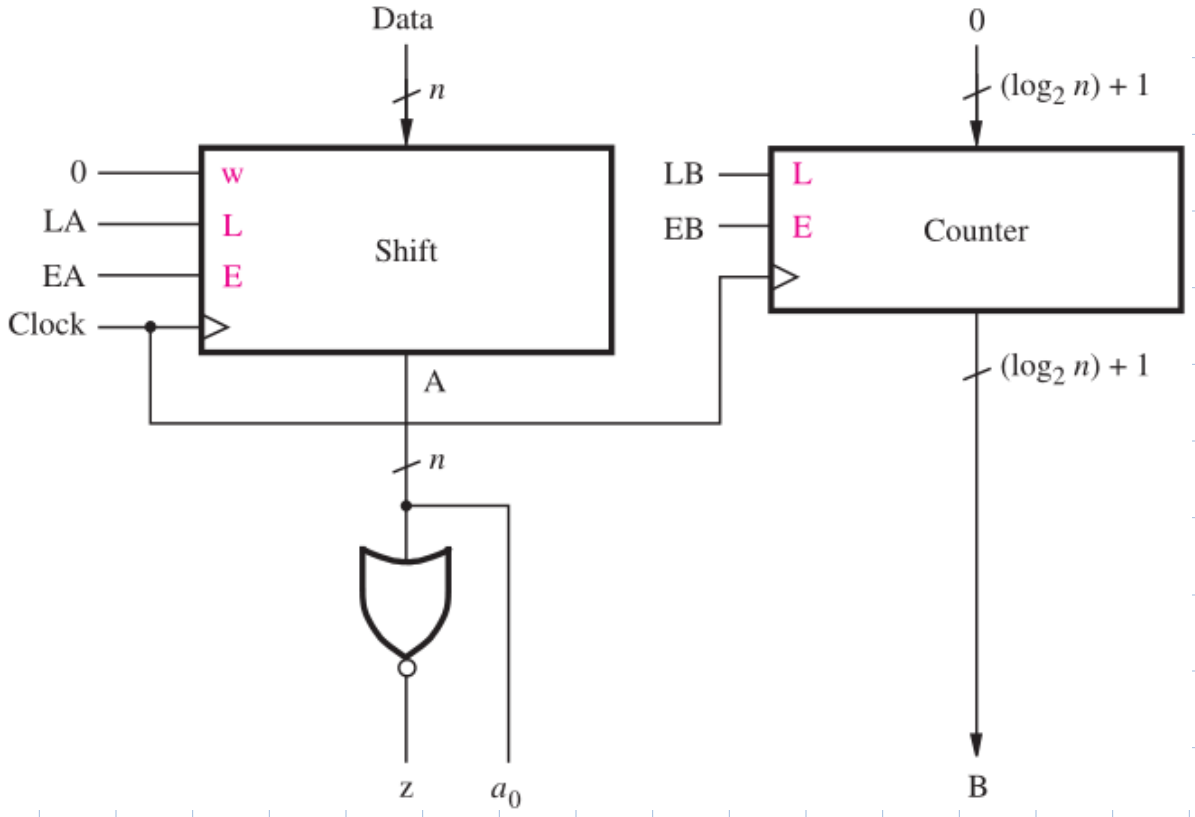


- La ASM requiere:
  - Un Shift Register para A
    - Capacidad de carga paralela
    - Una entrada de enable, ya que sólo opera en S2
  - Un contador para B
    - Carga paralela
    - Inicialización a 0 en estado S1
      - No es un reset
      - El reset cumple la función de partida y recuperación
- A partir de estas consideraciones podemos bosquejar el datapath

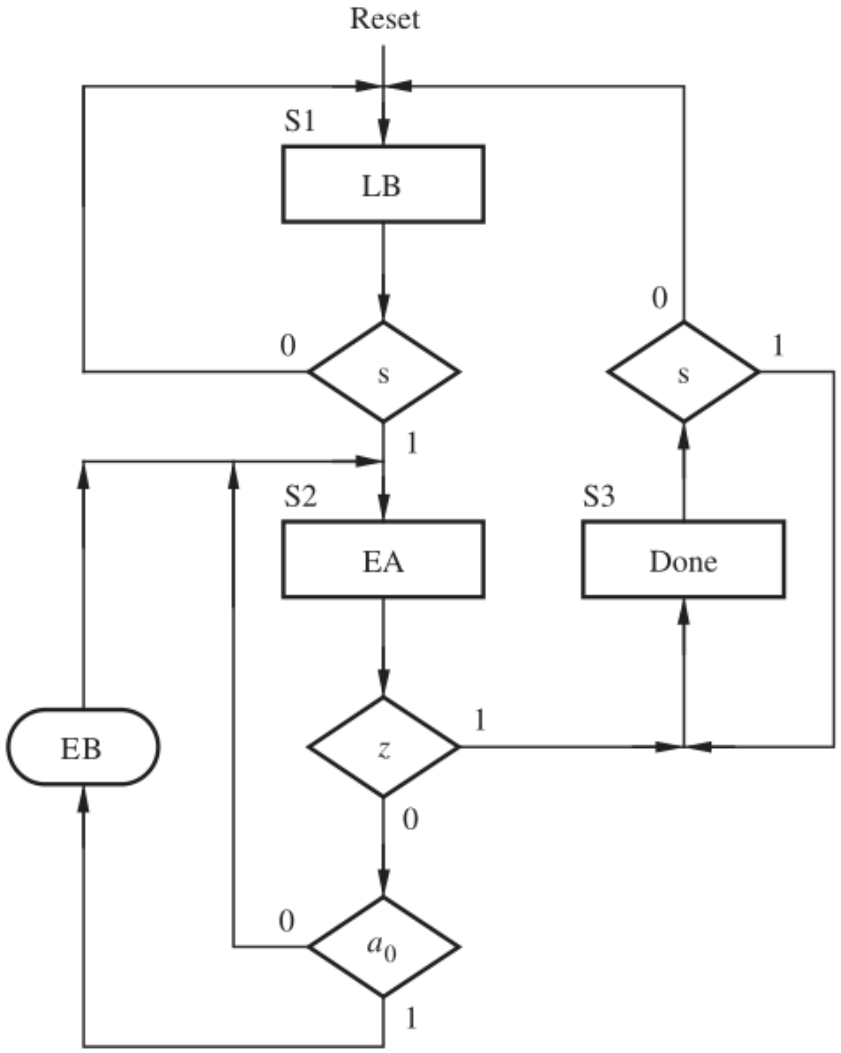
10.2.2 Información temporal implicada por diagrama de la ASM



Datapath para la ASM



10.2.2 Información temporal implicada por diagrama de la ASM



ASM del circuito de control

- Entradas:
  - s implica que estamos en S1 mientras s=0
  - a0 =1 habilita la cuenta de B
  - z=1 cuando A = 0
- LB carga en paralelo B
- EA y EB habilitan los shift registers
- Veamos el código Verilog…

10.2.2 Información temporal implicada por diagrama de la ASM

```
module bitcount (Clock, Resetn, LA, s, Data, B, Done);
    input Clock, Resetn, LA, s;
    input [7:0] Data;
    output [3:0] B;
    output Done;
    wire [7:0] A;
    wire z;
    reg [1:0] Y, y;
    reg [3:0] B;
    reg Done, EA, EB, LB;

// control circuit

    parameter S1 = 2'b00, S2 = 2'b01, S3 = 2'b10;

    always @(s or y or z)
    begin: State_table
        case (y)
            S1: if (!s) Y = S1;
                else Y = S2;
            S2: if (z == 0) Y = S2;
                else Y = S3;
            S3: if (s) Y = S3;
                else Y = S1;
            default: Y = 2'bxx;
        endcase
    end
```

```
always @(posedge Clock or negedge Resetn)
begin: State_flipflops
    if (Resetn == 0)
        y <= S1;
    else
        y <= Y;
end
```

```
    always @(y or A[0])
    begin: FSM_outputs
        // defaults
        EA = 0; LB = 0; EB = 0; Done = 0;
        case (y)
            S1: LB = 1;
            S2: begin
                    EA = 1;
                    if (A[0]) EB = 1;
                    else EB = 0;
                end
            S3: Done = 1;
        endcase
    end

// datapath circuit

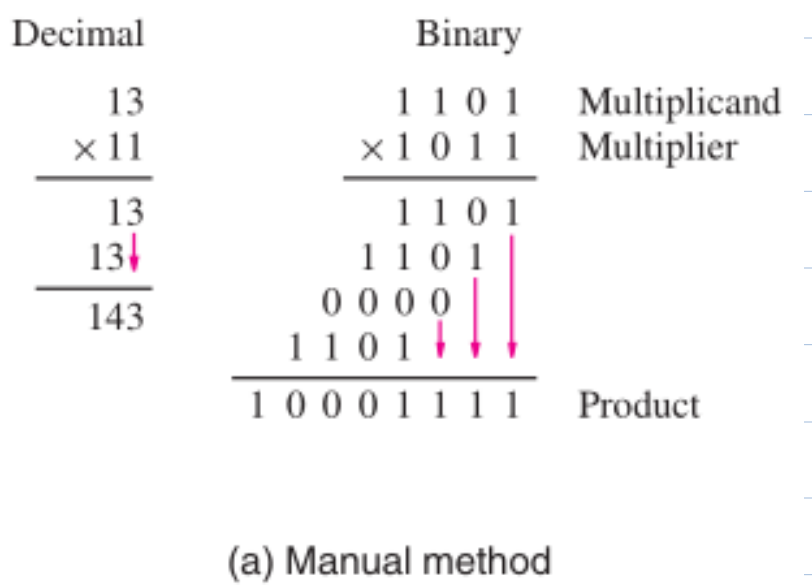
    // counter B
    always @(negedge Resetn or posedge Clock)
        if (!Resetn)
            B <= 0;
        else if (LB)
            B <= 0;
        else if (EB)
            B <= B + 1;

    shiftrne ShiftA (Data, LA, EA, 0, Clock, A);
    assign z = ~| A;

endmodule
```

10.2.3 Multiplicador shift-and-add – algoritmo

- El multiplicador paralelo que vimos en 5.36 requiere muchas compuertas y es costoso en área
- Otra opción es hacer un algoritmo similar al que hacemos cuando multiplicamos a mano



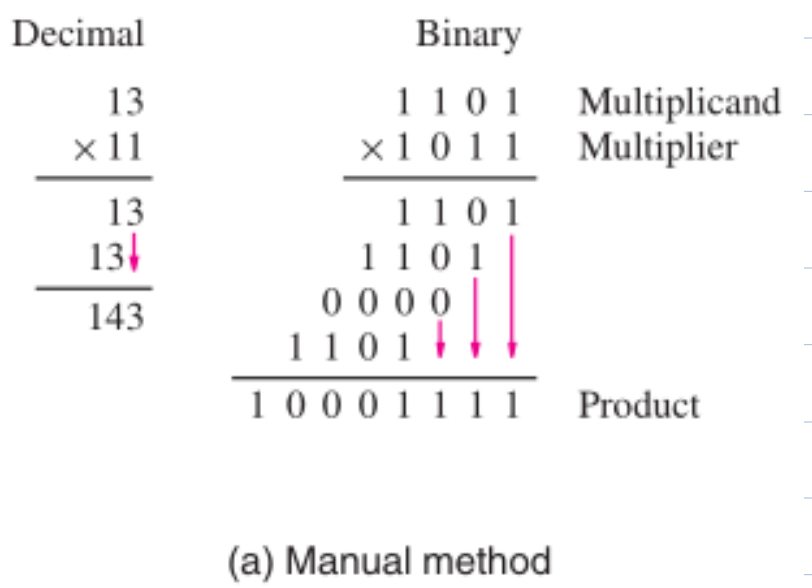
(a) Manual method

$P = 0$;
for $i = 0$ to $n - 1$ do
    if $b_i = 1$ then
        $P = P + A$;
    end if ;
    Left-shift A ;
end for ;

(b) Pseudo-code

10.2.3 Multiplicador shift-and-add – pseudocódigo

- El multiplicador paralelo que vimos en 5.36 requiere muchas compuertas y es costoso en área
- Otra opción es hacer un algoritmo similar al que hacemos cuando multiplicamos a mano



(a) Manual method

$P = 0$;
for $i = 0$ to $n - 1$ do
    if $b_i = 1$ then
        $P = P + A$;
    end if ;
    Left-shift A ;
end for ;

(b) Pseudo-code

10.2.3 Multiplicador shift-and-add – ASM

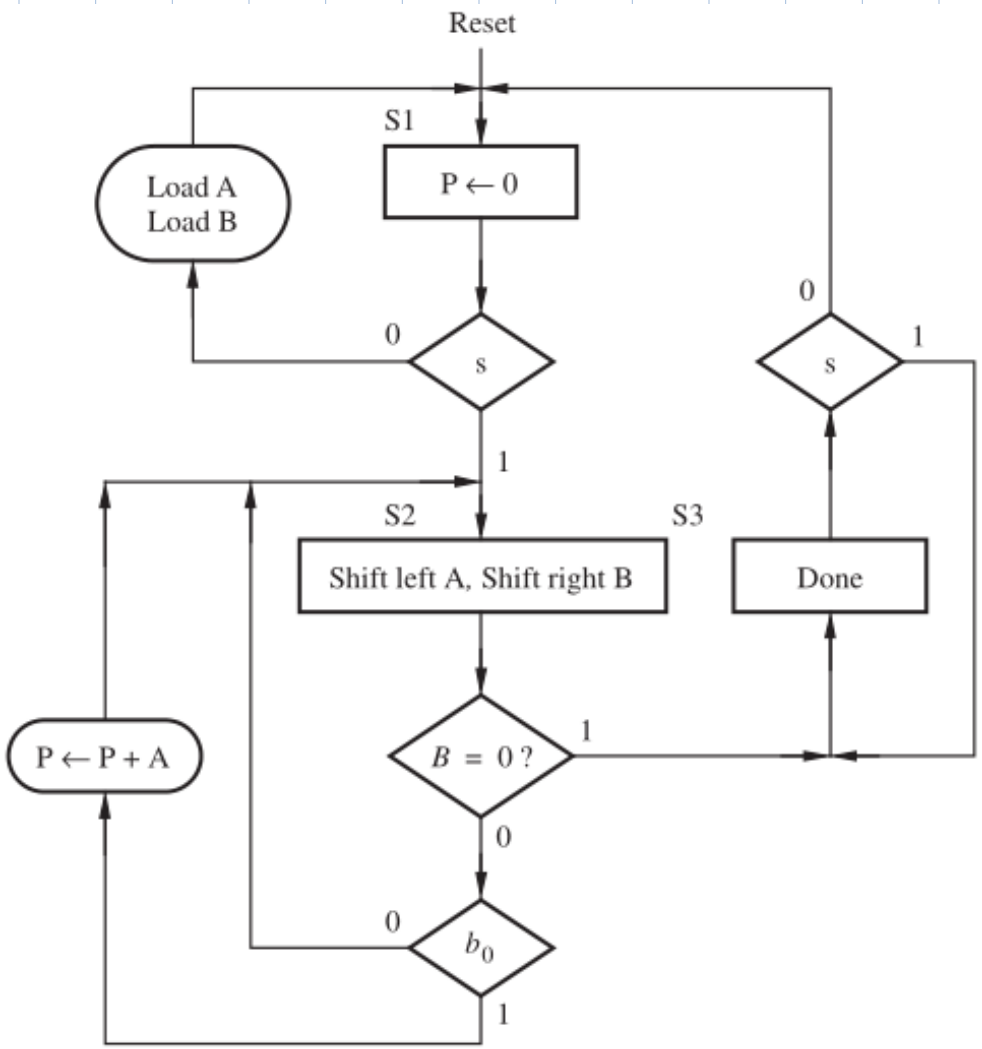
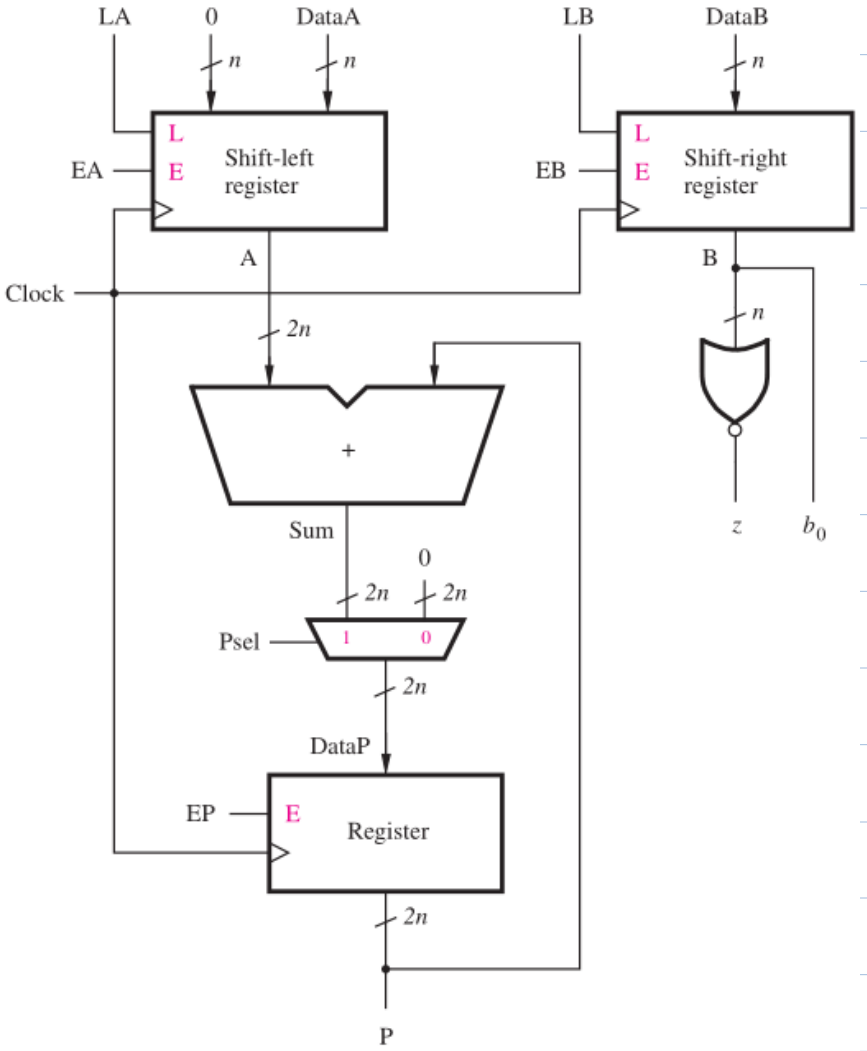


Diagrama ASM para el multiplicador

- Estado S1 carga datos
- s inicia el proceso y pasa a S2
- En S2 vemos LSB de B:
  - Si es 1, agregamos A a P
  - Si es 0
- Termina cuando B = 0
- P entrega el producto

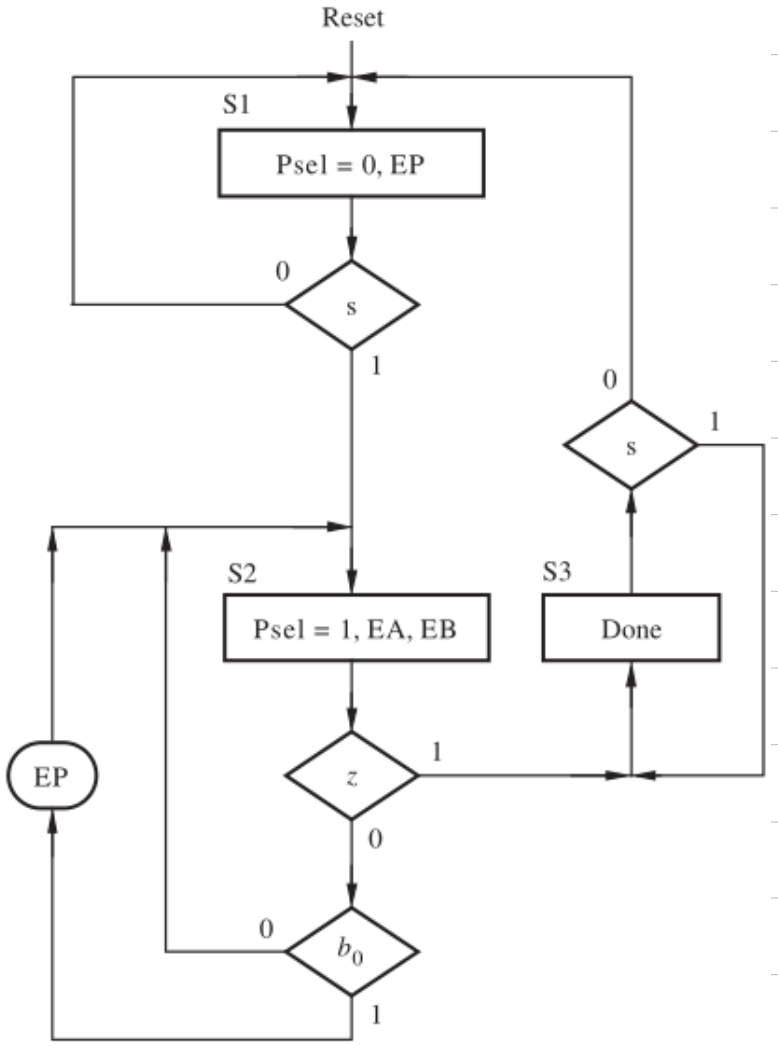
10.2.3 Multiplicador shift-and-add – datapath



Datapath del circuito multiplicador

- P tiene habilitación porque la asignación de suma está en bloque condicional (lámina anterior)
- P es cargado con 0 (al terminar la multiplicación) o con P+A
- Un NOR comprueba si B = 0

10.2.3 Multiplicador shift-and-add – circuito de control



Circuito de control del multiplicador

10.2.3 Multiplicador shift-and-add – Verilog

```
module multiply (Clock, Resetn, LA, LB, s, DataA, DataB, P, Done);
    parameter n = 8;
    input Clock, Resetn, LA, LB, s;
    input [n−1:0] DataA, DataB;
    output [n+n−1:0] P;
    output Done;
    wire z;
    reg [n+n−1:0] A, DataP;
    wire [n+n−1:0] Sum;
    reg [1:0] y, Y;
    reg [n−1:0] B;
    reg Done, EA, EB, EP, Psel;
    integer k;

// control circuit

    parameter S1 = 2'b00, S2 = 2'b01, S3 = 2'b10;

    always @(s or y or z)
    begin: State_table
        case (y)
            S1: if (s == 0) Y = S1;
                else Y = S2;
            S2: if (z == 0) Y = S2;
                else Y = S3;
            S3: if (s == 1) Y = S3;
                else Y = S1;
            default: Y = 2'bxx;
        endcase
    end

    always @(posedge Clock or negedge Resetn)
    begin: State_flipflops
        if (Resetn == 0)
            y <= S1;
        else
            y <= Y;
        end

    always @(s or y or B[0])
    begin: FSM_outputs
        // defaults
        EA = 0; EB = 0; EP = 0; Done = 0; Psel = 0;
        case (y)
            S1: EP = 1;
            S2: begin
                    EA = 1; EB = 1; Psel = 1;
                    if (B[0]) EP = 1;
                    else EP = 0;
                end
            S3: Done = 1;
        endcase
    end

//datapath circuit

    shiftrne ShiftB (DataB, LB, EB, 0, Clock, B);
        defparam ShiftB.n = 8;
    shiftlne ShiftA ({{n{1'b0}}, DataA}, LA, EA, 0, Clock, A);
        defparam ShiftA.n = 16;

    assign z = (B == 0);
    assign Sum = A + P;

    // define the 2n 2-to-1 multiplexers
    always @(Psel or Sum)
        for (k = 0; k < n+n; k = k+1)
            DataP[k] = Psel ? Sum[k] : 0;

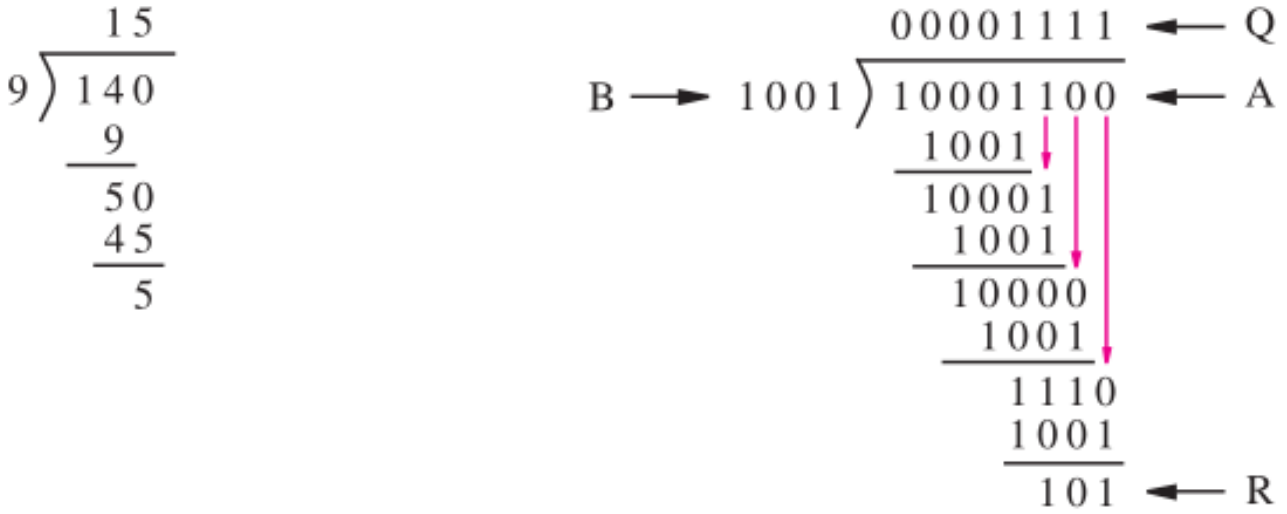
    regne RegP (DataP, Clock, Resetn, EP, P);
        defparam RegP.n = 16;

endmodule
```

Verilog del multiplicador

10.2.4 Divisor – algoritmo

- Intentaremos implementar división según el algoritmo de la figura, empleando números binarios



(a) An example using decimal numbers (b) Using binary numbers

- En el caso binario es un poco más sencillo ya que cada dígito del cuociente Q puede ser únicamente 0 o 1

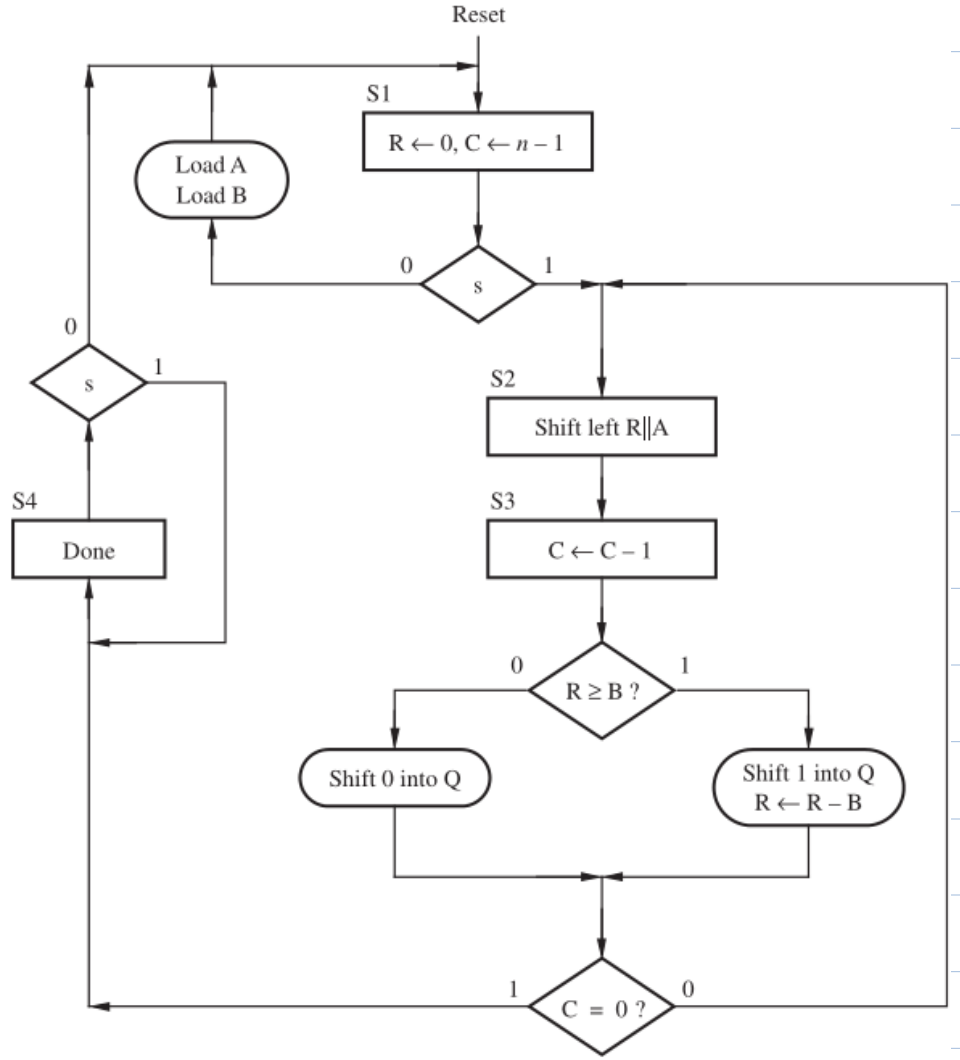
10.2.4 Divisor – pseudocódigo

- Queremos A/B = Q; también generaremos R
- Podemos pasar bits de A, uno a uno, mediante desplazamiento de bits a la izquierda a un registro R
- Después de cada shift comparamos R con B
  - Si R ≥ B ponemos 1 en el cuociente y restamos B de R
  - Si no, ponemos 0 en el cuociente
- El pseudocódigo muestra la operación
  - La notación R||A es un shift register de 2n bits concatenando R en los MSBs y A en los LSBs

$R = 0$;
for $i = 0$ to $n - 1$ do
    Left-shift $R \| A$;
    if R ≥ B then
        $q_i = 1$;
        $R = R - B$;
    else
        $q_i = 0$;
    end if ;
end for ;

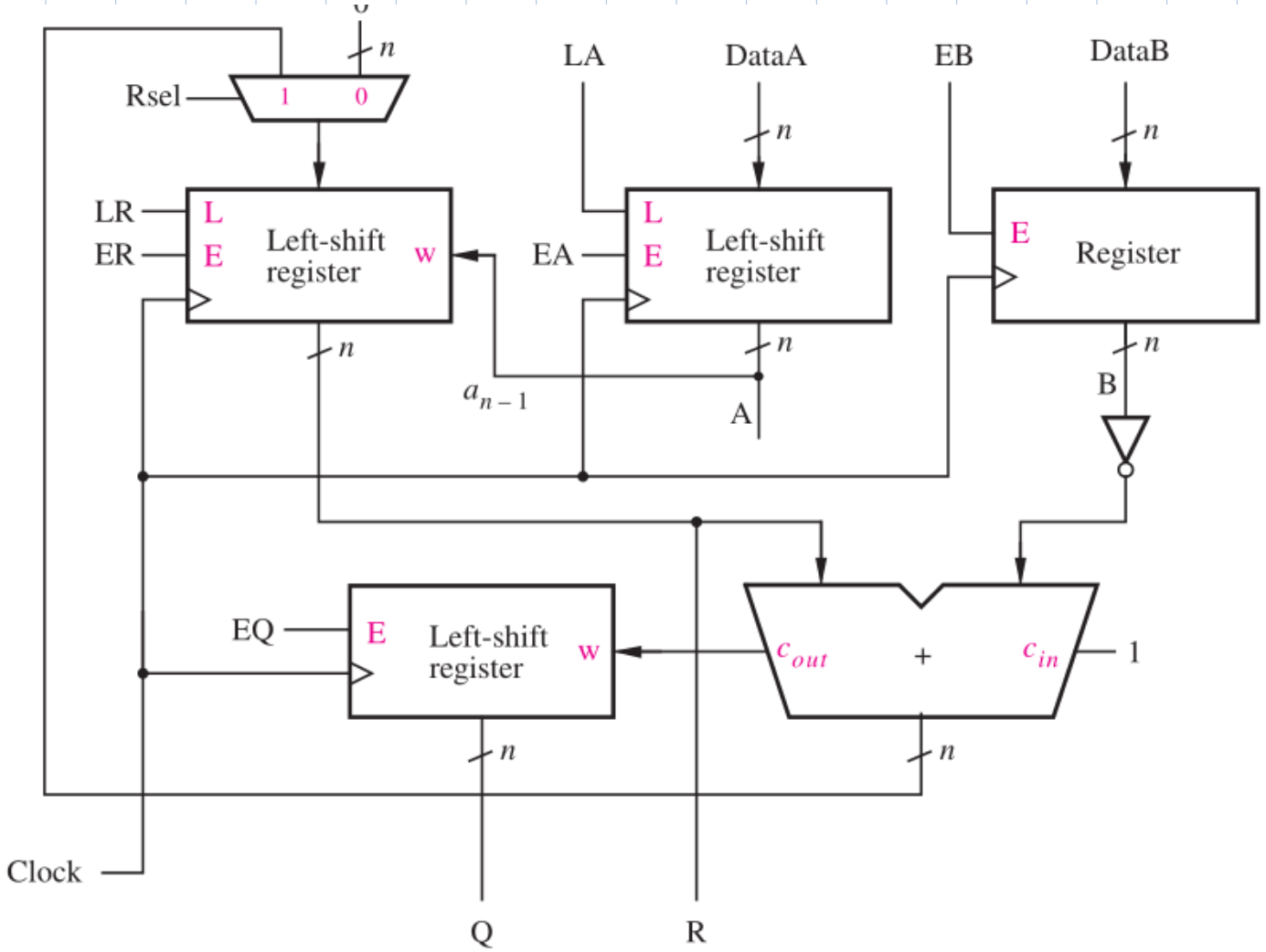
(c) Pseudo-code

10.2.4 Divisor – ASM



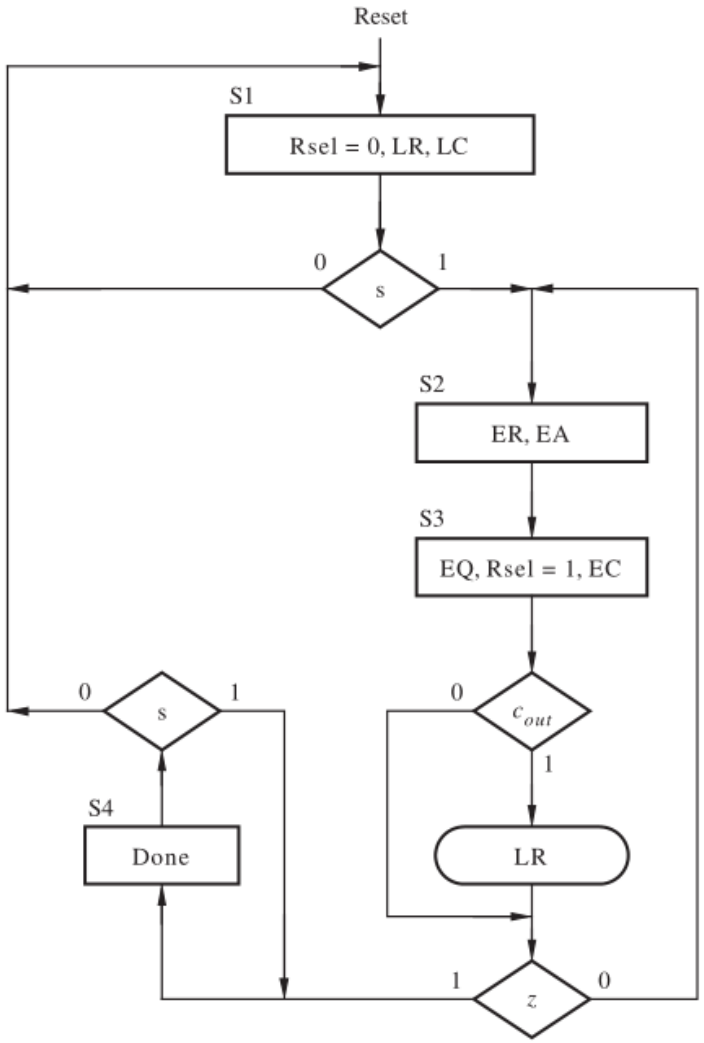
- Aquí vemos el diagrama de flujo de la ASM para el divisor
- El contador C es inicializado a n-1
- El divisor finaliza la operación cuando C llega a 0

10.2.4 Divisor – datapath



- Shift registers de n bits para A, B y R
- Resta para producir R-B
  - Usamos módulo sumador en que carry in = 1 y B es complementado
- El carry-out es 1 si R ≥ B
  - Entonces el carry out es directamente el bit del cuociente Q

10.2.4 Divisor – diagrama ASM del circuito de control



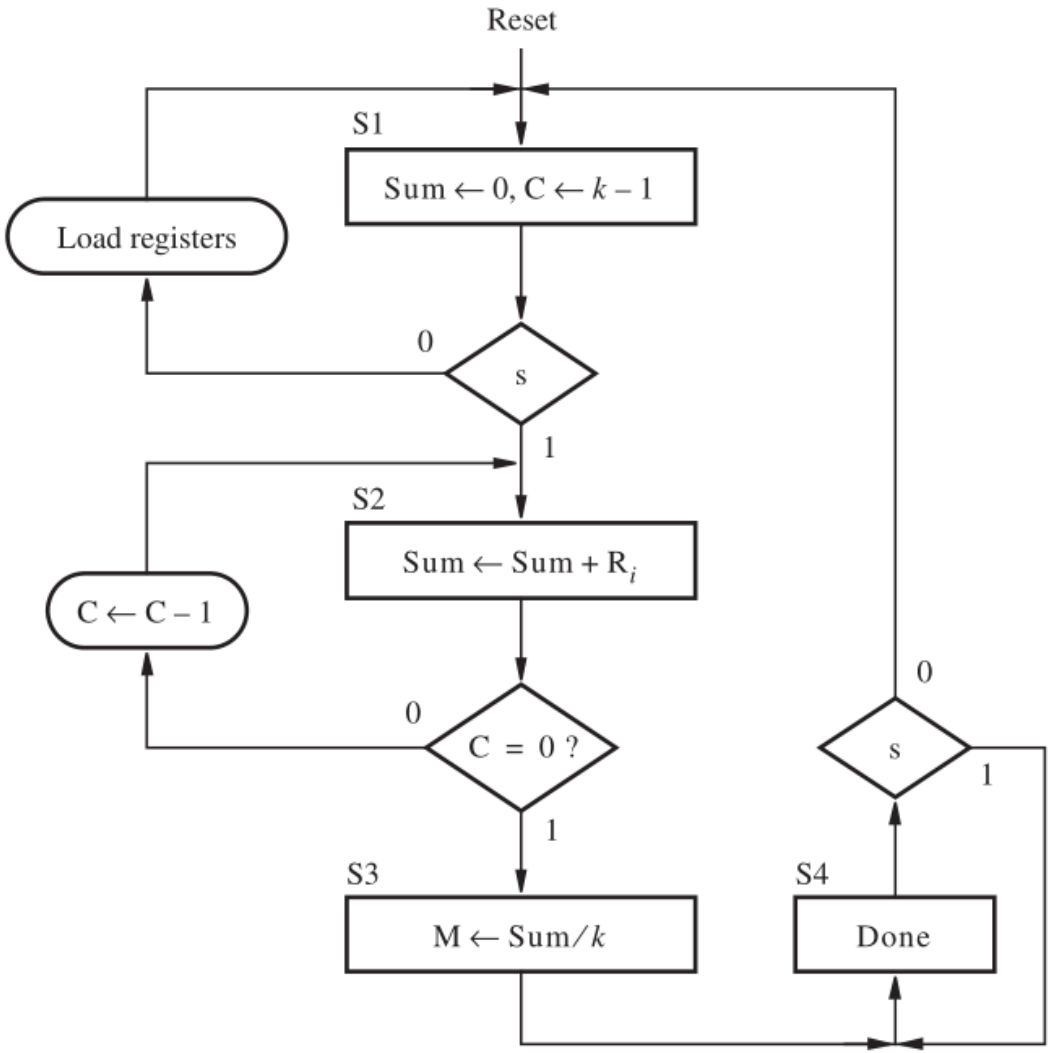
- Similar al de dos láminas atrás, pero dejando únicamente el circuito de control
- Se deja como tarea sin nota la descripción en Verilog del datapath y del circuito de control
- Hay 2n ciclos saltando entre estados S2 y S3
- Es posible combinar estados S2 y S3 para reducir el número de ciclos de 2n a n
  - El libro muestra cómo hacerlo

10.2.5 Media aritmética – pseudocódigo

- Diseñemos un circuito que calcula la media aritmética M de los números binarios guardados en un conjunto de registros $R_0 \ldots R_{k-1}$
- Primero sumamos todos los números en una variable Sum
- Luego de calcular la suma, dividimos por k
  - Asumimos división entera, aunque igual producimos residuo R
- La siguiente lámina muestra el diagrama de flujo

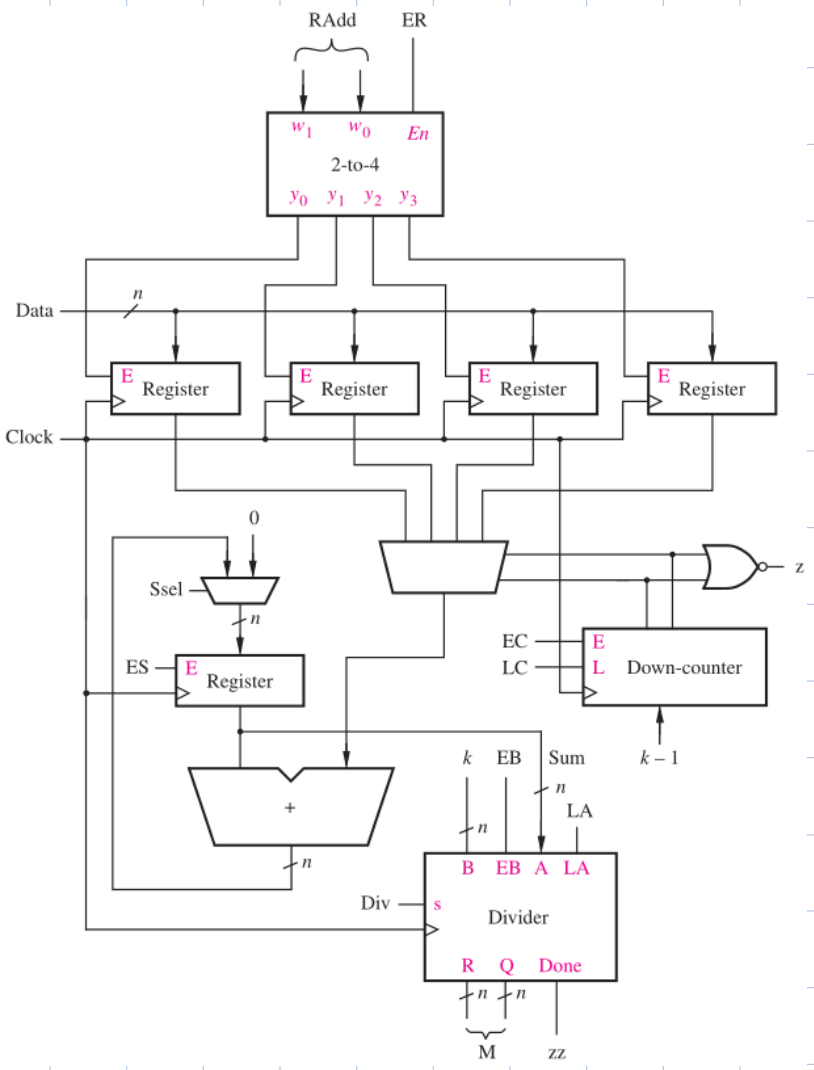
$$
\begin{aligned}
&Sum = 0\ ;\\
&\text{for } i = k-1 \text{ down to } 0 \text{ do}\\
&\qquad Sum = Sum + R_i\\
&\text{end for}\ ;\\
&M = Sum \div k\ ;
\end{aligned}
$$

10.2.5 Media aritmética – diagrama de flujo de la ASM



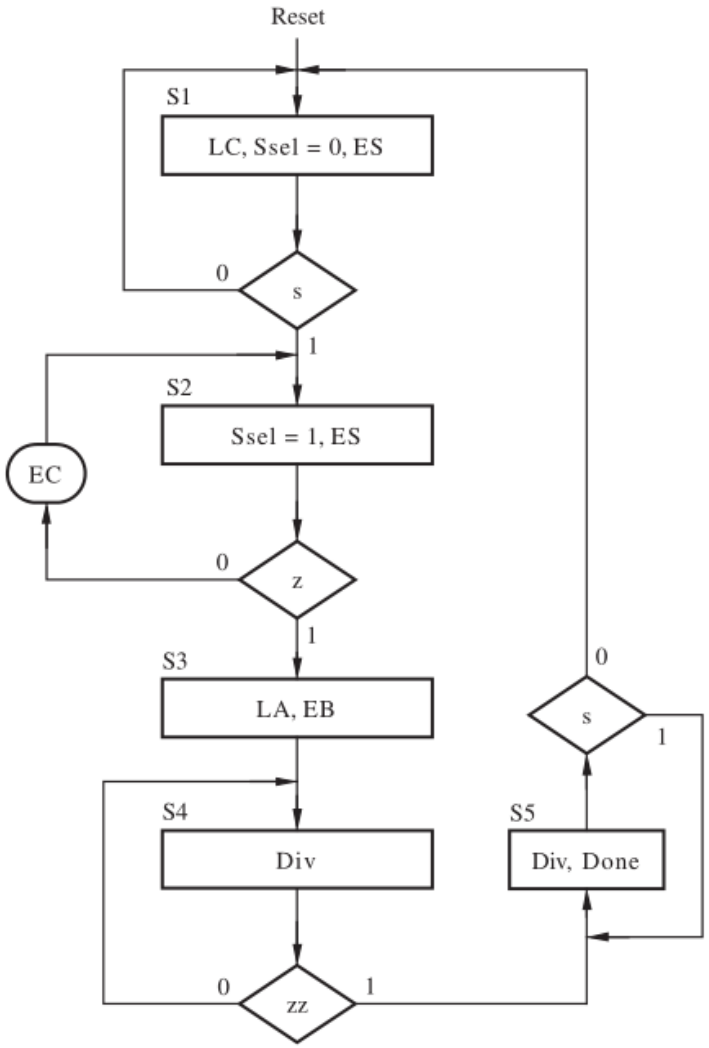
- Antes de iniciar, s = 0 y cargamos registros externamente
- Al iniciar s = 1, pasamos a S2 mientras la cuenta C sea distinta de 0
- Cuando C = 0 pasamos a S3 y calculamos la división
  - Sabemos que ésta toma múltiples ciclos de reloj, luego de lo cual activa Done

10.2.5 Media aritmética – Datapath



- Registro con enable guarda la suma Sum
  - n bits sin overflow por ahora
- En estado S1 el registro Sum es 0
- En estado S2 Sum recibe la salida de un sumador
- El selector del mux es controlado por un contador C
- Aquí usamos k = 4
- El divisor es el diseñado en 10.2.4
  - Pero con k = 4 podríamos hacer dos shift right y listo
- Otra alternativa de datapath se muestra en el libro, en que usamos tri-state buffers para la entrada
- Para k grande mejor usamos un bloque de SRAM

10.2.5 Media aritmética – Circuito de control



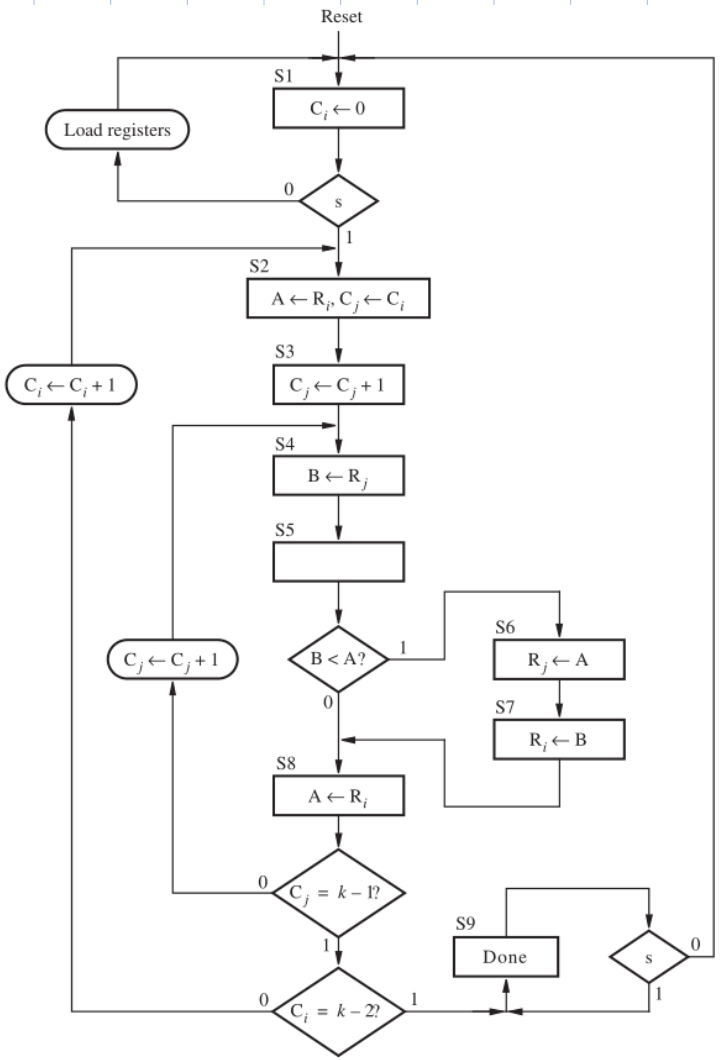
- En S1, cargamos los datos en los registros
- En s = 1 pasamos a S2 en que
  - Habilitamos la suma
  - Reducimos la cuenta de C
- Cuando C = 0 (z = 1) vamos a S3 en que cargamos suma y k en el circuito divisor
- La FSM entra en S4 y divide
- Al terminar la división, zz = 1 y la FSM va a S5
- Es posible combinar estados S3 y S4

10.2.6 Operación ordenar – algoritmo y pseudocódigo

- Hagamos un circuito que ordena una lista de números en registros $R_0 \ldots R_{k-1}$ en orden ascendente
- El pseudocódigo muestra el algoritmo:
  - Encuentra el menor número de una sublista
    - Para ello compara el número con todos los de la sublista; si encuentra uno menor, los intercambia
  - Mueve ese número al primer ítem de esa sublista
  - Analiza el resto de los números en una sublista más corta

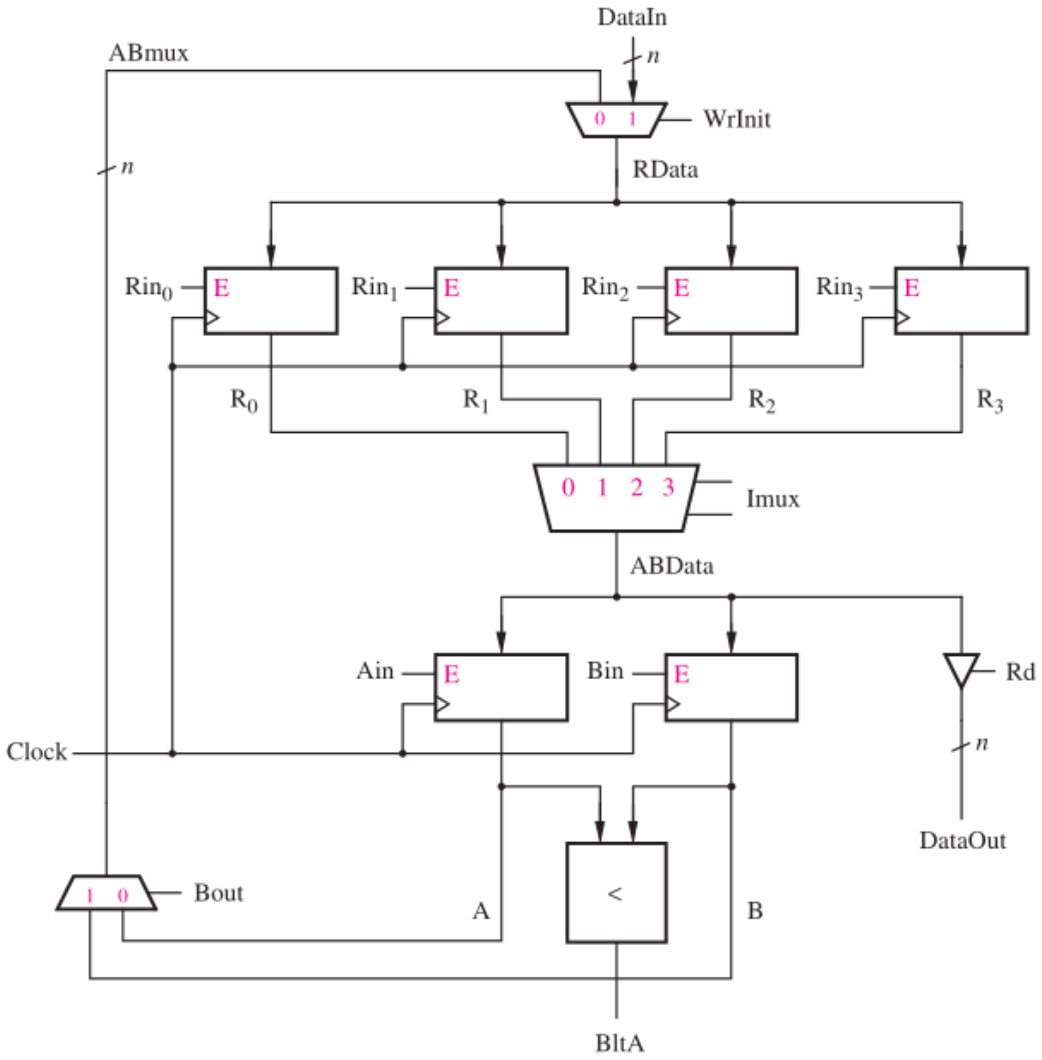
```
for i = 0 to k − 2 do
        A = R_i ;
        for j = i + 1 to k − 1 do
                B = R_j ;
                if B < A then
                        R_i = B ;
                        R_j = A ;
                        A = R_i ;
                end if ;
        end for ;
end for ;
```

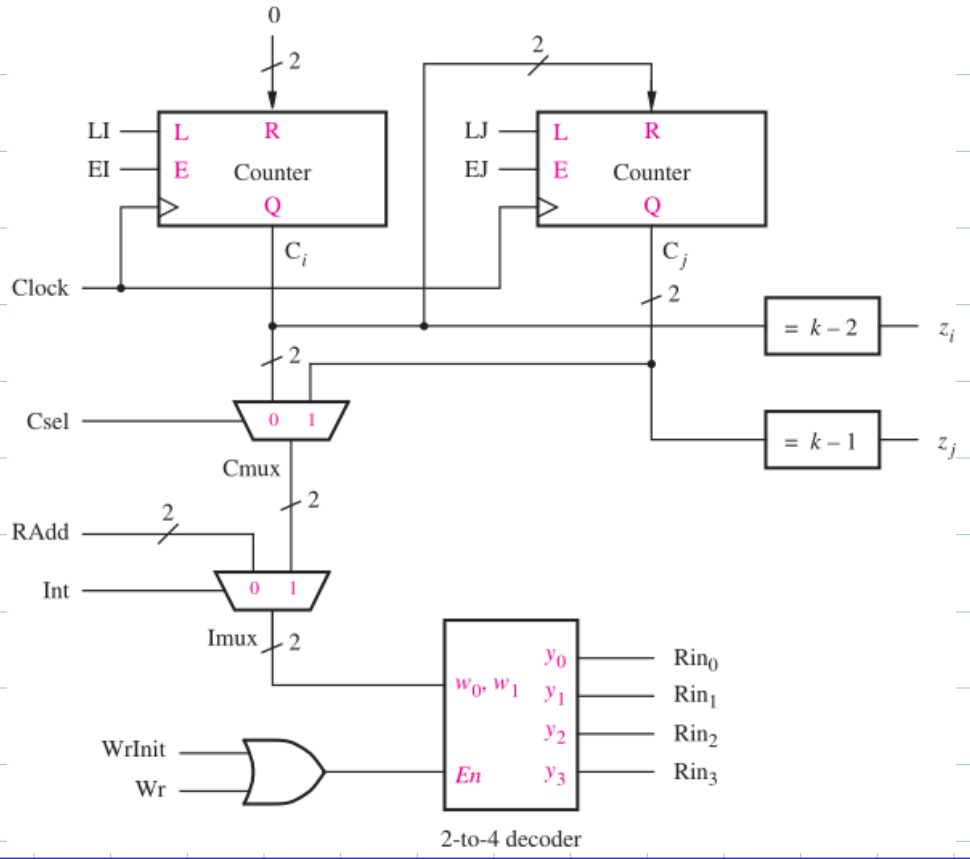
10.2.6 Operación ordenar – ASM



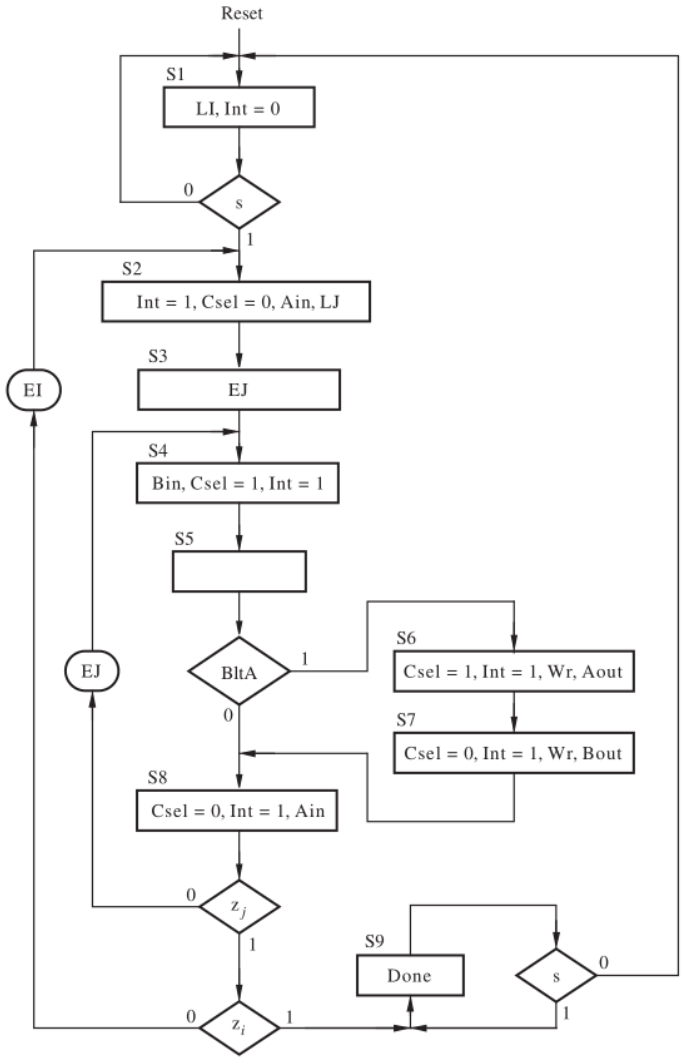
- Aquí vemos (apenas) el diagrama de flujo de la ASM
- En S1 (s=0) cargamos datos en registros
  - También el contador Ci (loop externo) es inicializado
- En S2 cargamos Ri en A
  - Además inicializamos Cj (loop interno) al valor de I
- En S3 inicializamos j al valor i+1
- En S4 cargamos Rj en B
- En S5 comparamos A y B; si B < A vamos a S6
- En S6 y S7 intercambiamos Ri y Rj
- En S8 cargamos A desde Ri
- Si Cj no llega a k-1, volvemos a S4
- Si Cj = k-1 y Ci no llega a k-2, volvemos a S2

10.2.6 Operación ordenar – Datapath





10.2.6 Operación ordenar – ASM para el circuito de control



10.2.6 Operación ordenar – Verilog

```
module sort (Clock, Resetn, s, WrInit, Rd, DataIn, RAdd, DataOut, Done);
    parameter n = 4;
    input Clock, Resetn, s, WrInit, Rd;
    input [n−1:0] DataIn;
    input [1:0] RAdd;
    output [n−1:0] DataOut;
    output Done;
    wire [1:0] Ci, Cj, CMux, IMux;
    wire [n−1:0] R0, R1, R2, R3, A, B, RData, ABMux;
    wire BltA, zi, zj;
    reg Int, Csel, Wr, Ain, Bin, Aout, Bout;
    reg LI, LJ, EI, EJ, Done, Rin0, Rin1, Rin2, Rin3;
    reg [3:0] y, Y;
    reg [n−1:0] ABData;

// control circuit
    parameter S1 = 4'b0000, S2 = 4'b0001, S3 = 4'b0010, S4 = 4'b0011;
    parameter S5 = 4'b0100, S6 = 4'b0101, S7 = 4'b0110, S8 = 4'b0111, S9 = 4'b1000;

    always @(s or BltA or zj or zi)
    begin: State_table
        case (y)
            S1: if (s == 0) Y = S1;
                else Y = S2;
            S2: Y = S3;
            S3: Y = S4;
            S4: Y = S5;
            S5: if (BltA) Y = S6;
                else Y = S8;
            S6: Y = S7;
            S7: Y = S8;
            S8: if (!zj) Y = S4;
                else if (!zi) Y = S2;
                else Y = S9;
            S9: if (s) Y = S9;
                else Y = S1;
            default: Y = 4'bx;
        endcase
    end
```

```
always @(posedge Clock or negedge Resetn)
begin: State_flipflops
    if (Resetn == 0)
        y <= S1;
    else
        y <= Y;
end

always @(y or zj or zi)
begin: FSM_outputs
    // defaults
    Int = 1; Done = 0; LI = 0; LJ = 0; EI = 0; EJ = 0; Csel = 0;
    Wr = 0; Ain = 0; Bin = 0; Aout = 0; Bout = 0;
    case (y)
        S1: begin LI = 1; Int = 0; end
        S2: begin Ain = 1; LJ = 1; end
        S3: EJ = 1;
        S4: begin Bin = 1; Csel = 1; end
        S5:; // no ouputs asserted in this state
        S6: begin Csel = 1; Wr = 1; Aout = 1; end
        S7: begin Wr = 1; Bout = 1; end
        S8: begin
                Ain = 1;
                if (!zj) EJ = 1;
                else
                begin
                    EJ = 0;
                    if (!zi) EI = 1;
                    else EI = 0;
                end
            end
        S9: Done = 1;
    endcase
end
```

Continúa…

10.2.6 Operación ordenar – Verilog

```
//datapath circuit

regne  Reg0 (RData, Clock, Resetn, Rin0, R0);
   defparam Reg0.n = n;
regne  Reg1 (RData, Clock, Resetn, Rin1, R1);
   defparam Reg1.n = n;
regne  Reg2 (RData, Clock, Resetn, Rin2, R2);
   defparam Reg2.n = n;
regne  Reg3 (RData, Clock, Resetn, Rin3, R3);
   defparam Reg3.n = n;

regne  RegA (ABData, Clock, Resetn, Ain, A);
   defparam RegA.n = n;
regne  RegB (ABData, Clock, Resetn, Bin, B);
   defparam RegB.n = n;

assign  BltA = (B < A) ? 1 : 0;
assign  ABMux = (Bout == 0) ? A : B;
assign  RData = (WrInit == 0) ? ABMux : DataIn;

upcount  OuterLoop (0, Resetn, Clock, EI, LI, Ci);
upcount  InnerLoop (Ci, Resetn, Clock, EJ, LJ, Cj);

assign  CMux = (Csel == 0) ? Ci : Cj;
assign  IMux = (Int == 1) ? CMux : RAdd;
```

```
always @(WrInit or Wr or IMux)
begin
   case (IMux)
      0:  ABData = R0;
      1:  ABData = R1;
      2:  ABData = R2;
      3:  ABData = R3;
   endcase

   if (WrInit || Wr)
      case (IMux)
         0:  {Rin3, Rin2, Rin1, Rin0} = 4'b0001;
         1:  {Rin3, Rin2, Rin1, Rin0} = 4'b0010;
         2:  {Rin3, Rin2, Rin1, Rin0} = 4'b0100;
         3:  {Rin3, Rin2, Rin1, Rin0} = 4'b1000;
      endcase
   else {Rin3, Rin2, Rin1, Rin0} = 4'b0000;
end

assign  zi = (Ci == 2);
assign  zj = (Cj == 3);
assign  DataOut = (Rd == 0) ? 'bz : ABData;

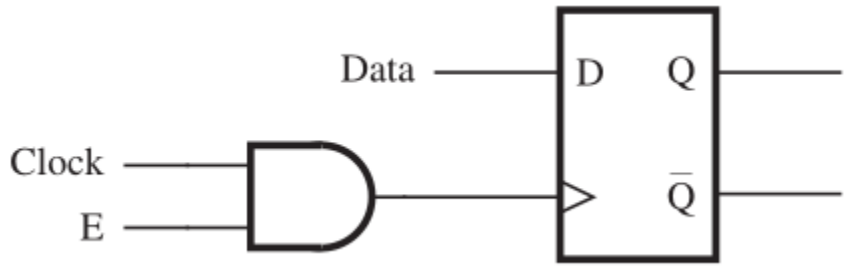
endmodule
```

10.3 Sincronización de reloj

- Hemos aprendido que los circuitos y sistemas digitales utilizan múltiples flip-flops en máquinas de estado que operan concurrentemente
- También aprendimos que en un circuito bien diseñado, todos los FFs debieran estar gatillados por el mismo reloj
- En esta sección discutiremos sobre asuntos de temporización y sincronización de circuitos secuenciales más complejos

10.3.1 Variación de reloj (clock skew)

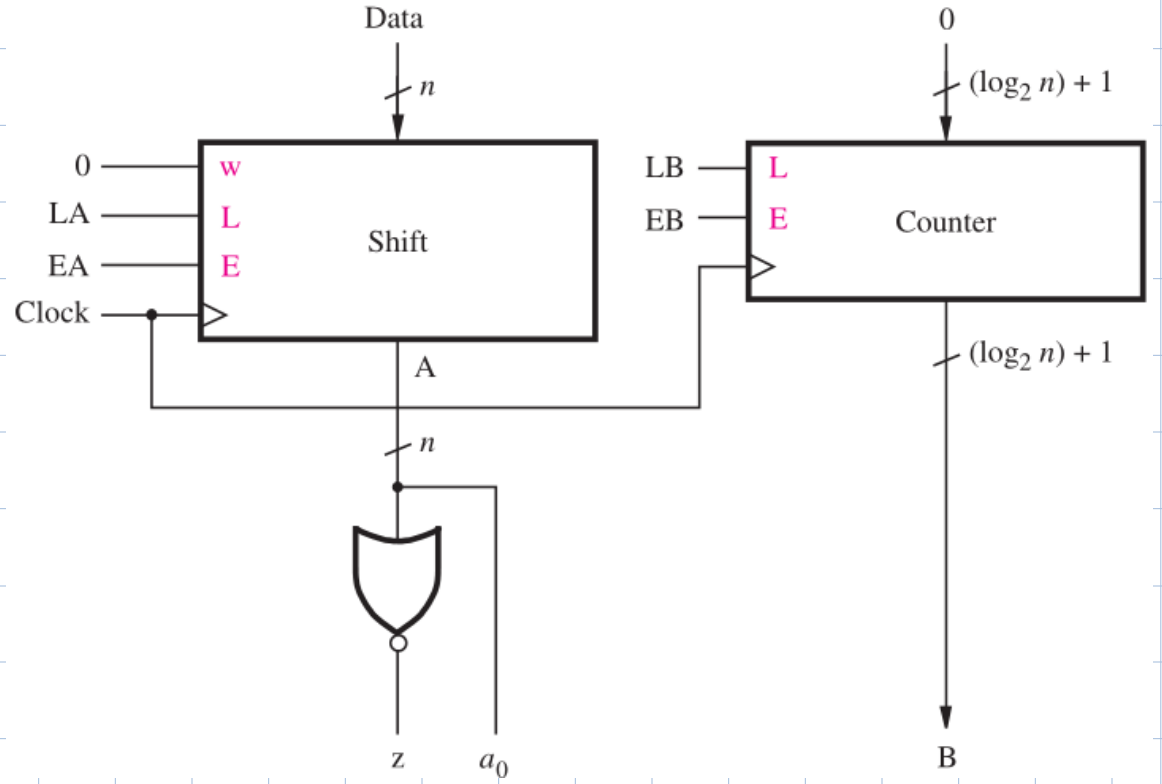
- Cuando queremos que un FF sea sensible a algunos flancos de reloj (ej, que el resultado de un contador cambie un estado) podemos usar un FF con enable
- Otra alternativa es la de la figura
  - Más simple, pero propenso a problemas
  - Los FFs sin enable verán cambios un poco antes que los FFs con enable
  - Esta condición se denomina clock skew o variación de reloj



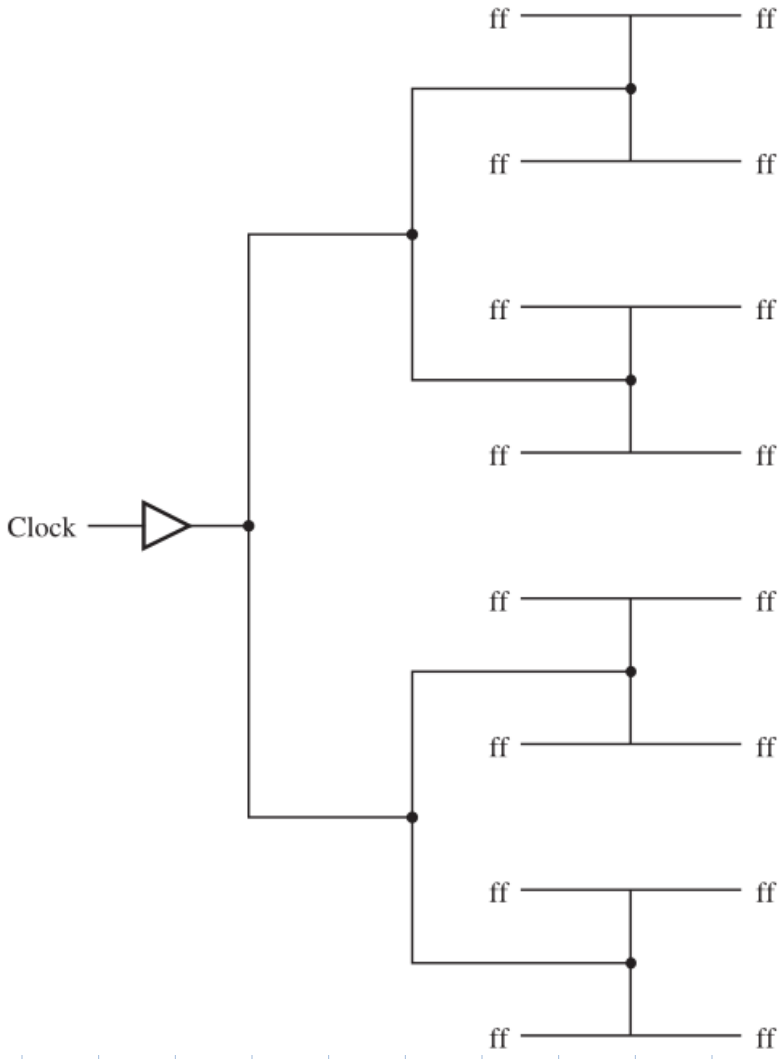
- Existen diversas causas del clock skew
- Estudiemos sus consecuencias mediante un ejemplo

10.3.1 Variación de reloj (clock skew)

- El LSB del SR, a0, controla el incremento de un contador
- Supongamos que el reloj llega antes al SR y después al contador
  - Entonces, la señal EB podría llegar al contador des-sincronizada del reloj, resultando en una cuenta perdida
- Es esencial minimizar el clock skew en circuitos secuenciales
- La lámina siguiente muestra una distribución de reloj típica para este propósito



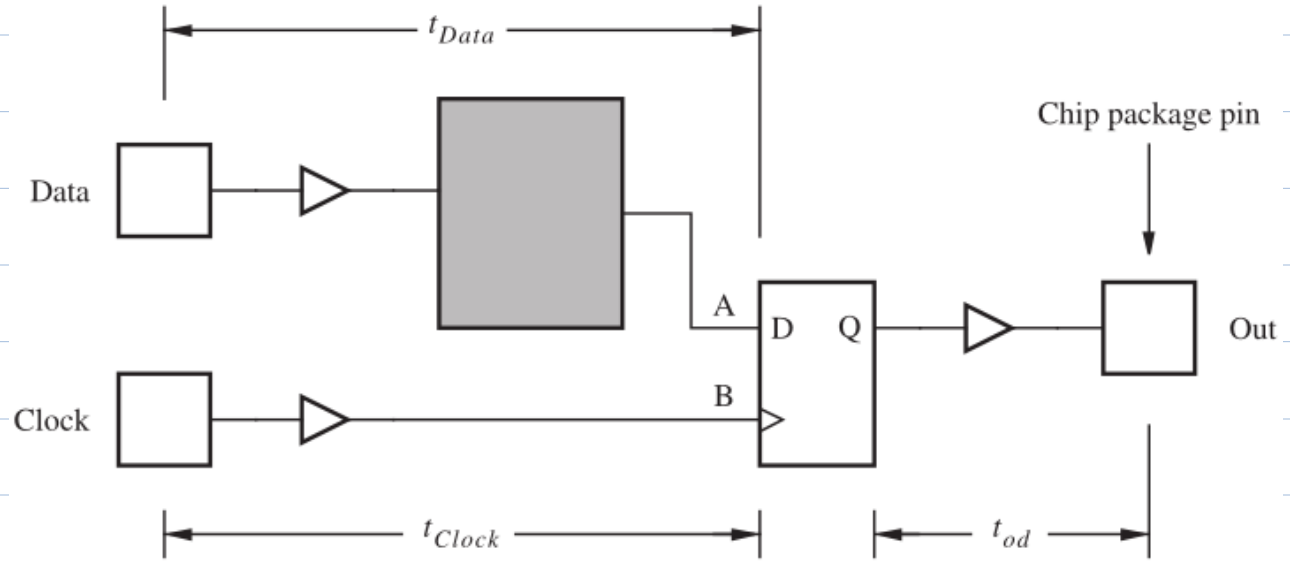
10.3.1 Variación de reloj (clock skew)



- Esta distribución de reloj se denomina Árbol H (H tree)
- En PLDs, el reloj global se refiere a la red de distribución de reloj en el chip
- Los PLDs pueden tener más de una red de distribución de reloj global
  - Todos los FFs deben ir al mismo reloj
  - Evitar en lo posible pasar el reloj por compuertas lógicas
- Los PLDs también cuentan con una red de distribución de reset asíncrono

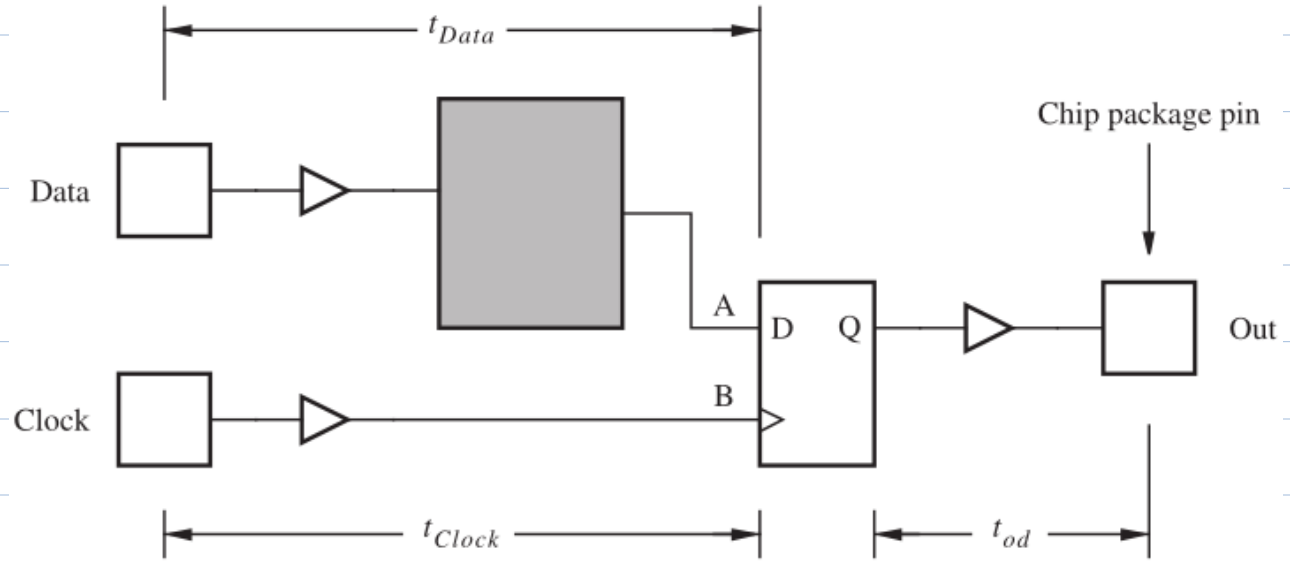
10.3.2 Parámetros de tiempo de los flip-flops

- En 7.3.1 aprendimos tiempos de setup t_{su} y de hold t_{hold}, que deben ser respetados
  - La señal de entrada debe estar estable t_{su} antes del flanco
  - La señal de entrada debe mantenerse estable t_{hold} después del flanco
- Veamos un ejemplo
  - Tiempos de propagación etiquetados
- La salida del chip va conectada a la entrada de otro chip
  - Importante mantener sincro-nización entre D y clock en el siguiente chip



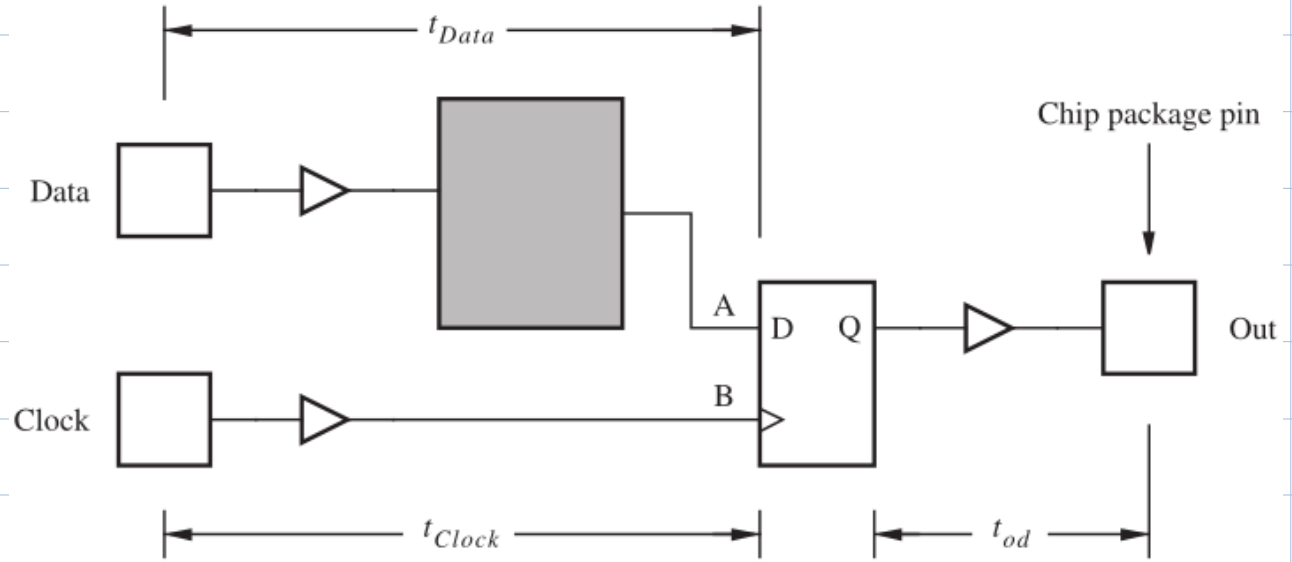
10.3.2 Parámetros de tiempo de los flip-flops

- Veamos cómo estudiar la sincronización entre chips
- t_{co} determina el tiempo entre el flanco de reloj en la entrada de un chip, y el cambio en la salida del chip, y tiene 3 partes
  - Reloj desde pin hasta FF t_{Clock}
  - Retardo del FF t_{rd}
  - Propagación de salida del FF hasta pin del chip t_{od}
- Un CPLD comercial tiene
  - t_{Clock} = 1.5 ns
  - t_{rd} = 1 ns
  - t_{od} = 2 ns
  - Entonces t_{co} = 4.5 ns



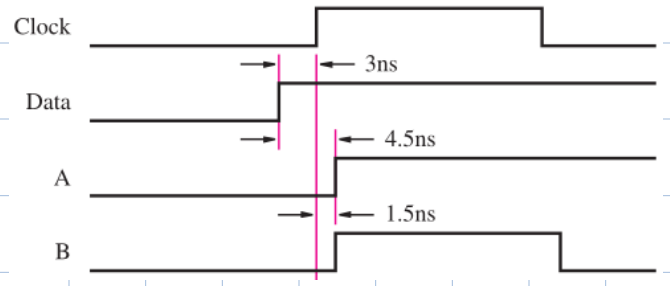
10.3.2 Parámetros de tiempo de los flip-flops

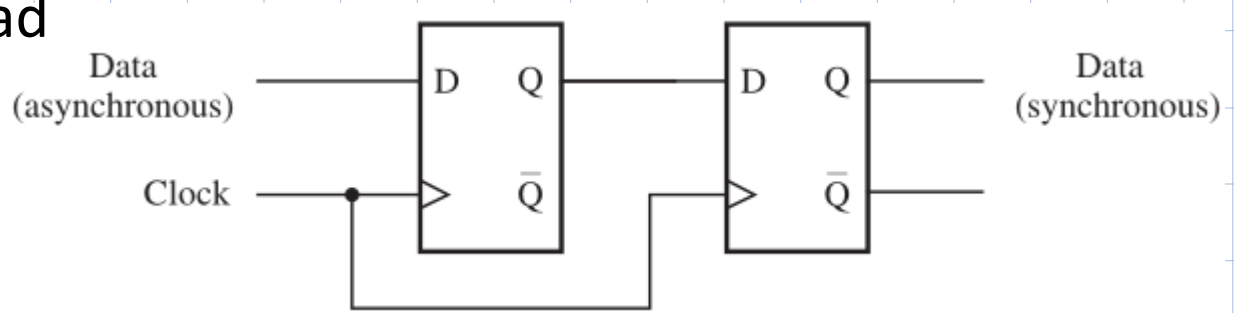
- Si los chips están muy distantes, es necesario tomar esto en cuenta
  - Suele no ser un problema en tarjetas bien diseñadas
- Cuando una señal alcanza el chip, hay que considerar t_{Data} y t_{Clock}
- Ej: t_{Data} = 4.5 ns, t_{Clock} = 1.5 ns
- Además t_{su} = 3 ns
- Supongamos que las entradas al chip respetan t_{su}
- Pero el retardo en t_{Data} dentro del chip produce una violación del tiempo de setup del FF!!!
- Lo mismo pasa con t_{Hold}, que suele verse reducido por interco-nexiones entre chips



10.3.3 Entradas asíncronas en flip-flops

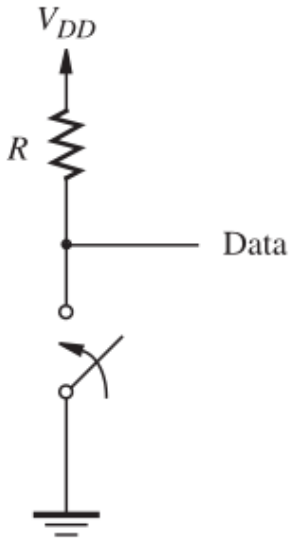
- Salidas de FFs van a lógica combinacional y luego a la entrada de otros FFs
  - Entonces las entradas a los FFs cambian poco después de un flanco activo
- Es posible que algunas entradas no vayan sincronizadas con el reloj
  - Esto puede violar tiempos de setup y hold
  - Esto puede resultar en estados metaestables – salida no es 0 ni 1 por un tiempo
- Una solución es usar SRs de 2 bits
  - Sincroniza y resuelve metaestabilidad

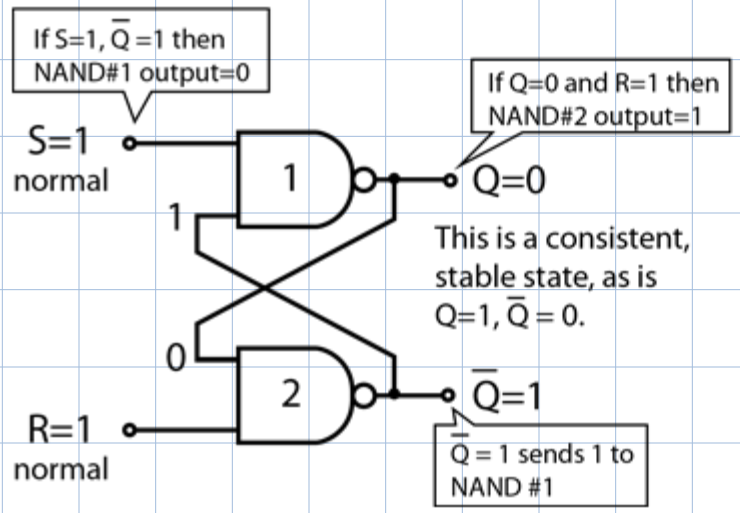




10.3.4 Debouncer

- En la T6 aprendimos que los switches mecánicos tienen rebotes que pueden provocar múltiples flancos durante unos 10 ms
- Una opción es hacer un debouncer que veta la entrada durante un tiempo determinado
- Una mejor alternativa es el circuito de la derecha con un latch SR
  - Durante los rebotes, el latch mantiene la salida correcta

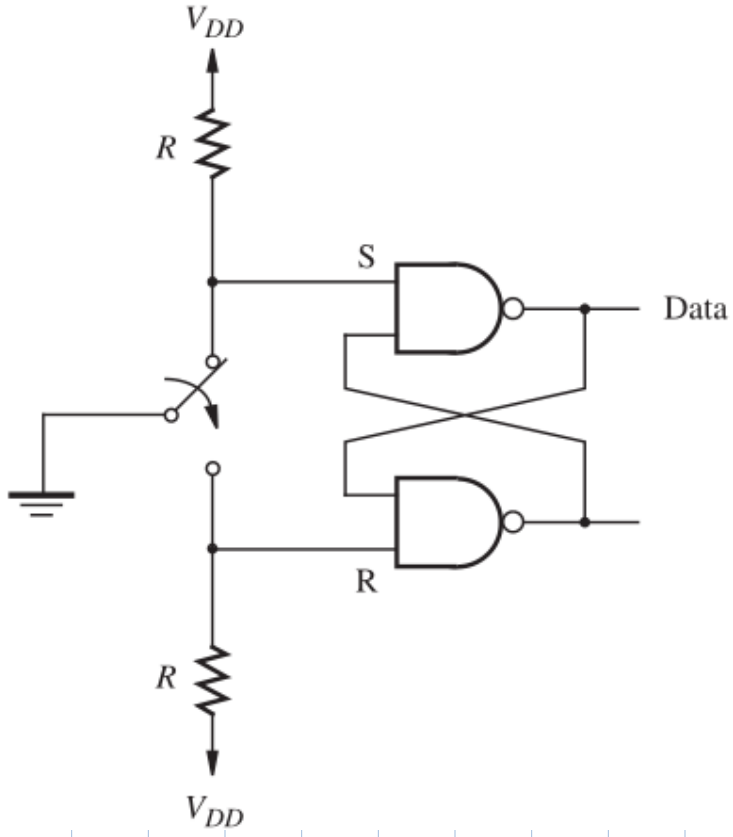




Truth Table

Set	Reset	Output
1	1	No change*
0	1	Q=1
1	0	Q=0
0	0	Invalid state

* can be used for data storage



Recordemos latch SR https://tinyurl.com/zfsltf9

10.4 Conclusiones

- En este capítulo vimos varios ejemplos de ASMs y bloques como sumadores, registros, shift registers y contadores
- Aprendimos a dibujar diagramas de ASMs para diseñar sistemas digitales y hemos visto cómo describirlos mediante Verilog
- También discutimos aspectos prácticos como clock skew, sincronización, debouncer y metaestabilidad