CprE 281:
Digital Logic

Instructor: Alexander Stoytchev

http://www.ece.iastate.edu/~alexs/classes/

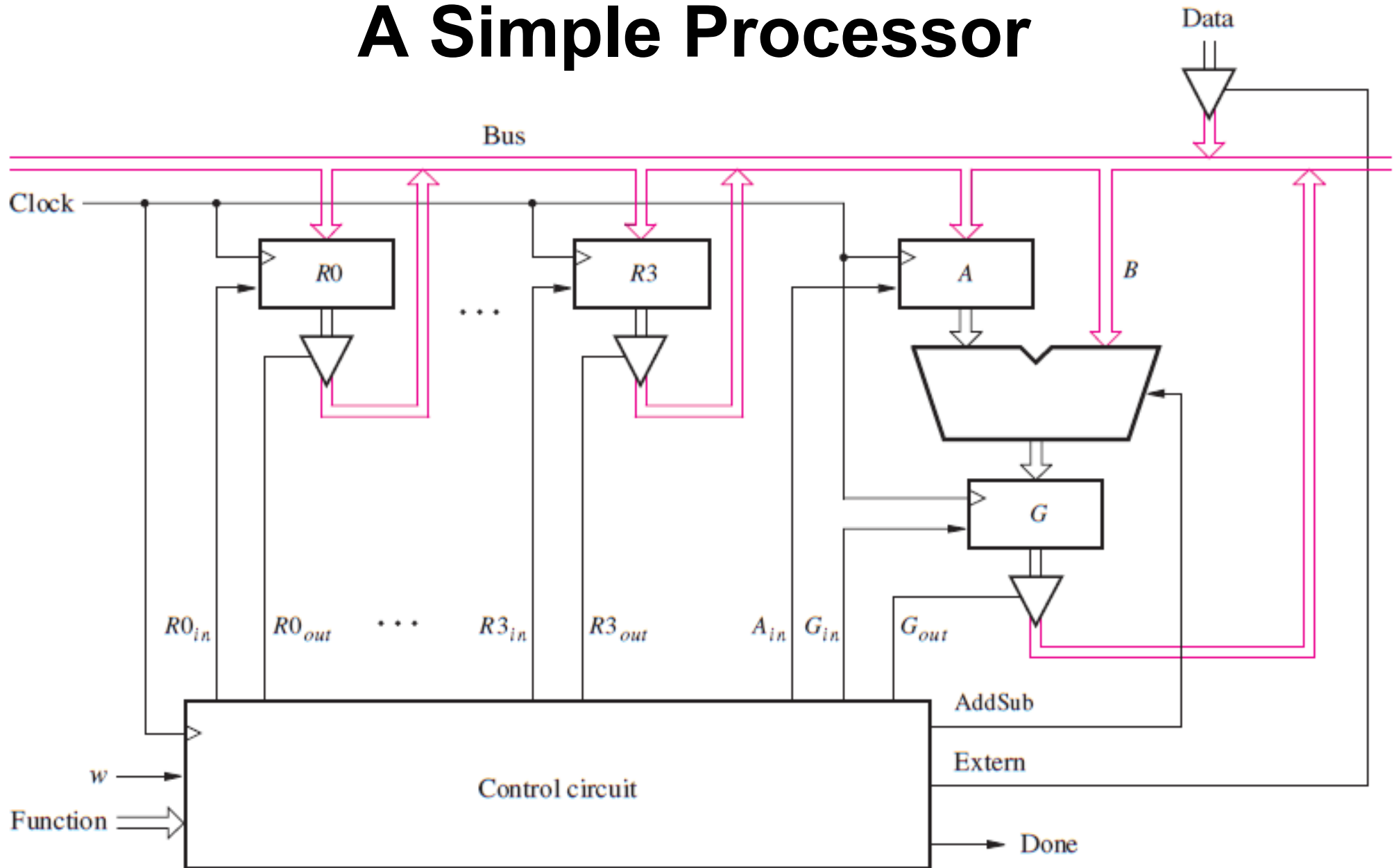# Simple Processor

Digital System

- **Datapath circuit**
  - **To store data**
  - **To manipulate data**
  - **To transfer data from one part of the system to another**
  - **Comprise building blocks such as registers, shift registers, counters, multipliers, decoders, encoders, adders, etc.**
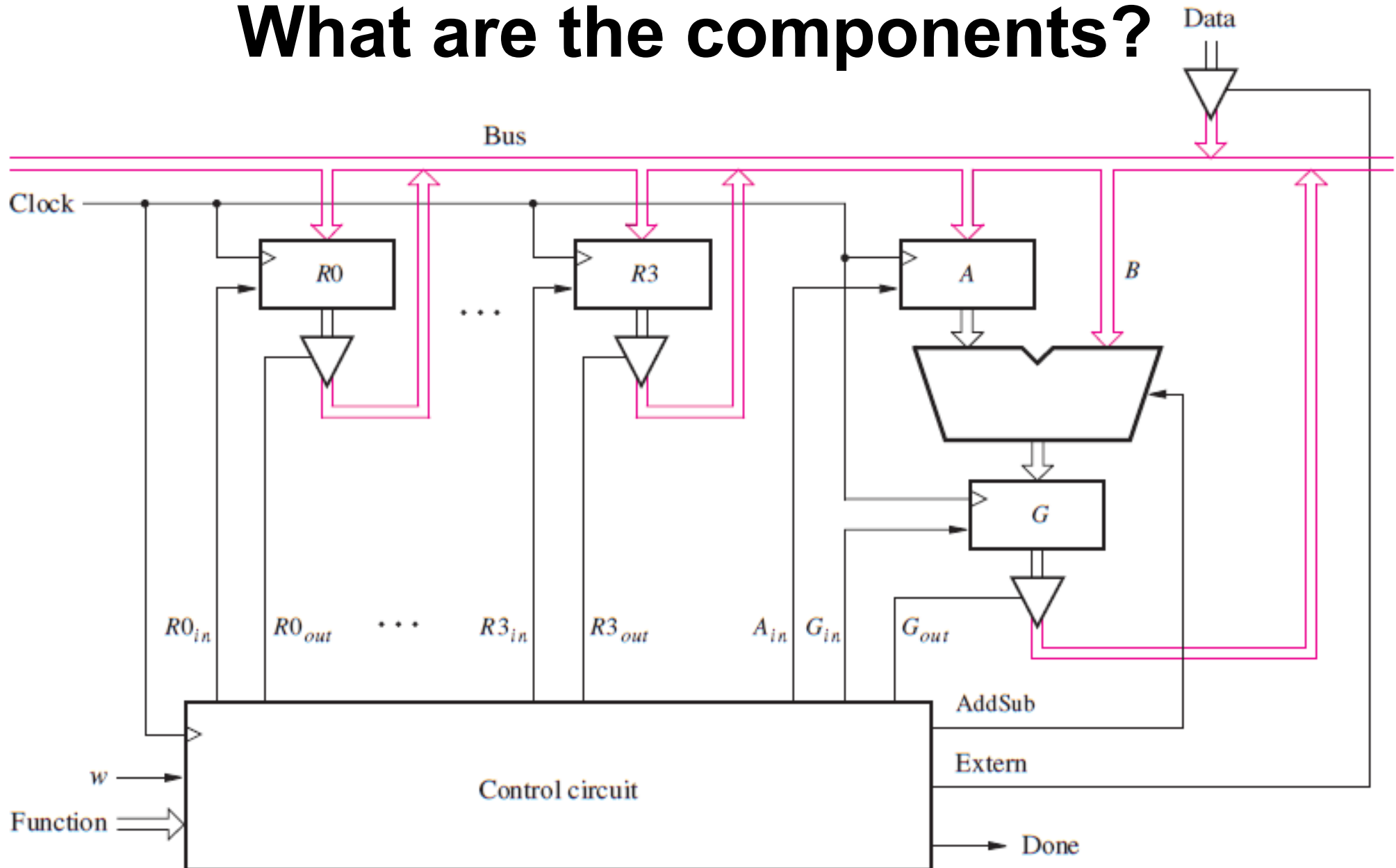
- **Control circuit**
  - **Controls the operation of the datapath circuit**
  - **Designed as a FSM**

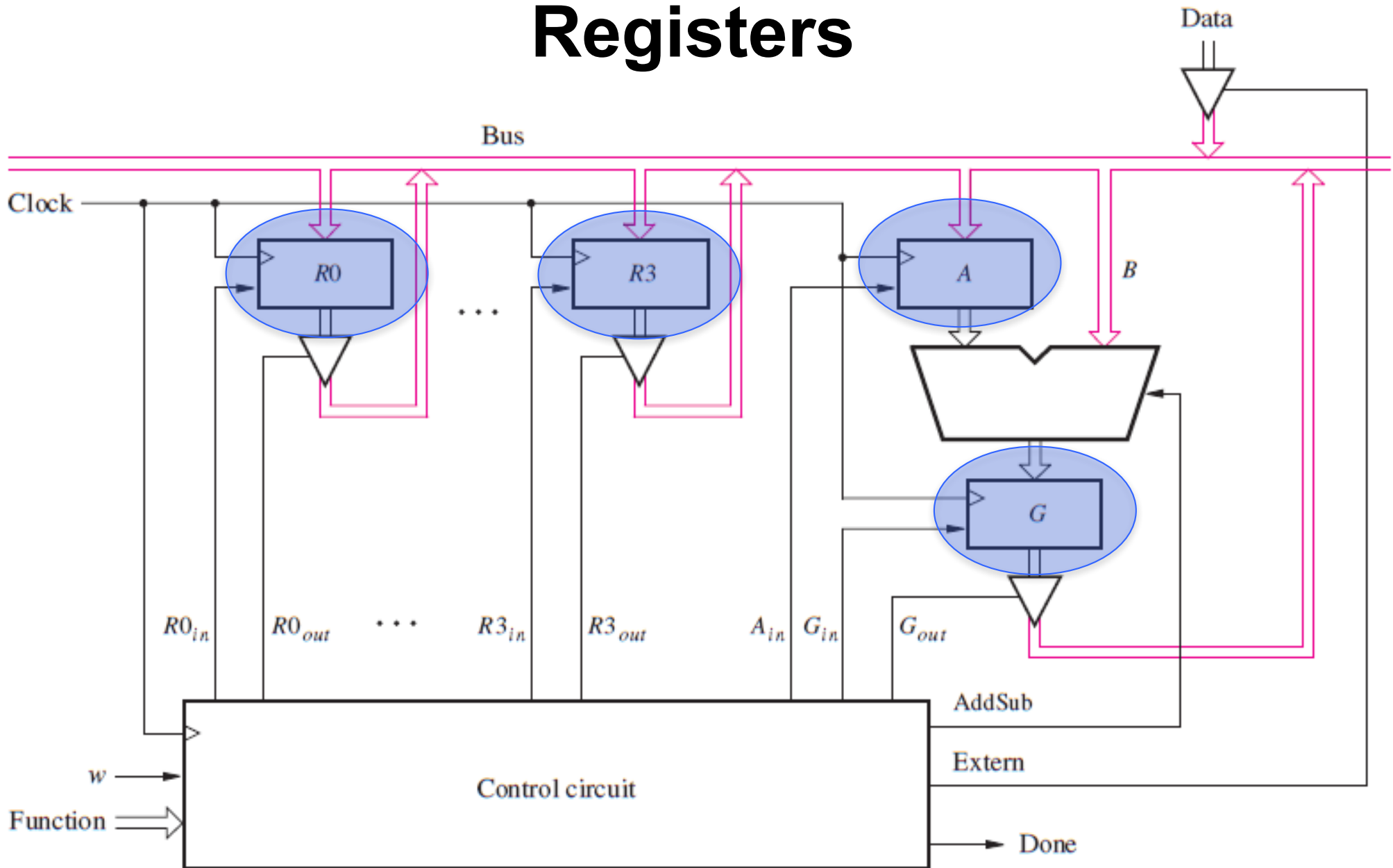# A Simple Processor

[Figure 7.9 from the textbook]

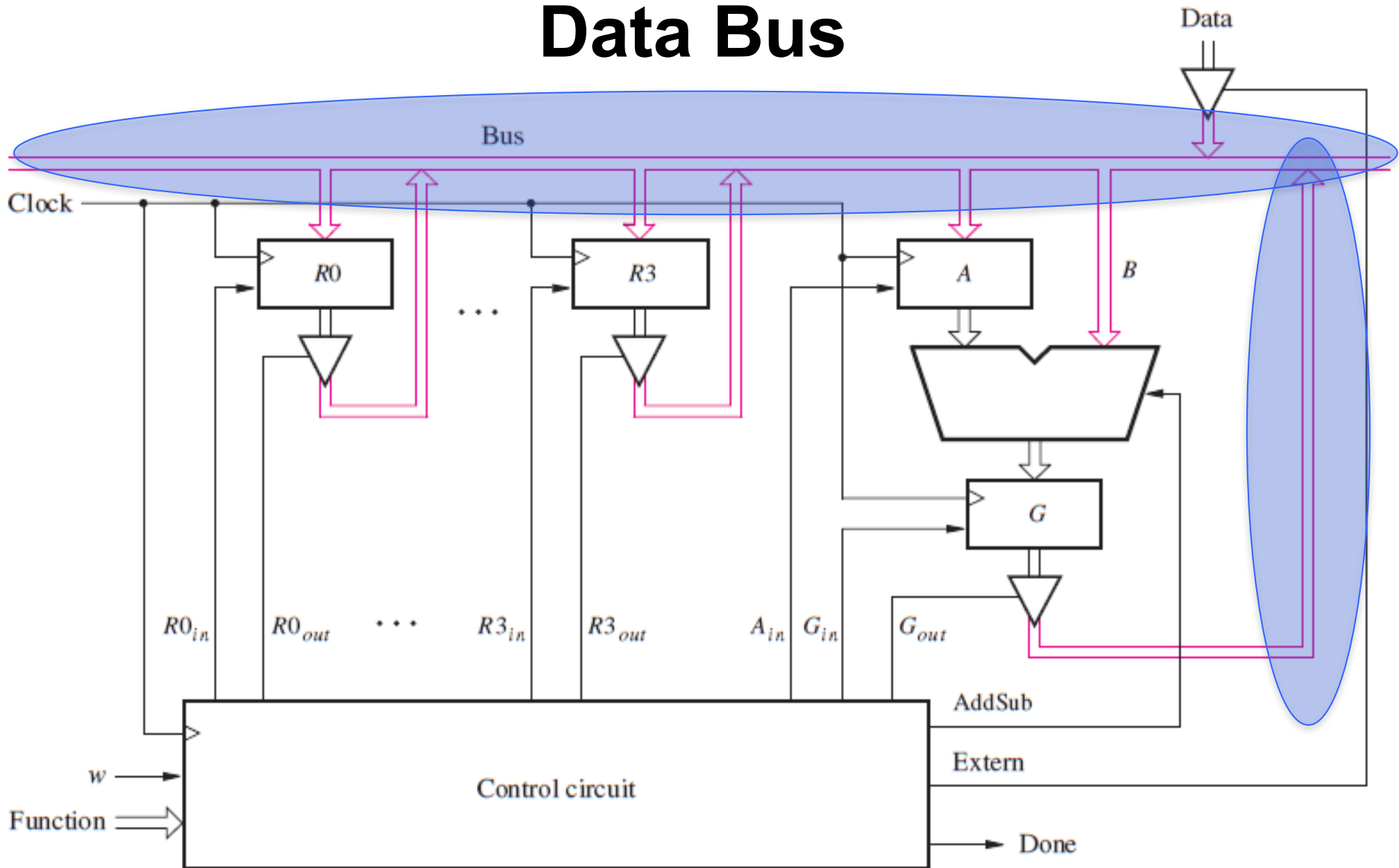# What are the components?

[Figure 7.9 from the textbook]

Registers

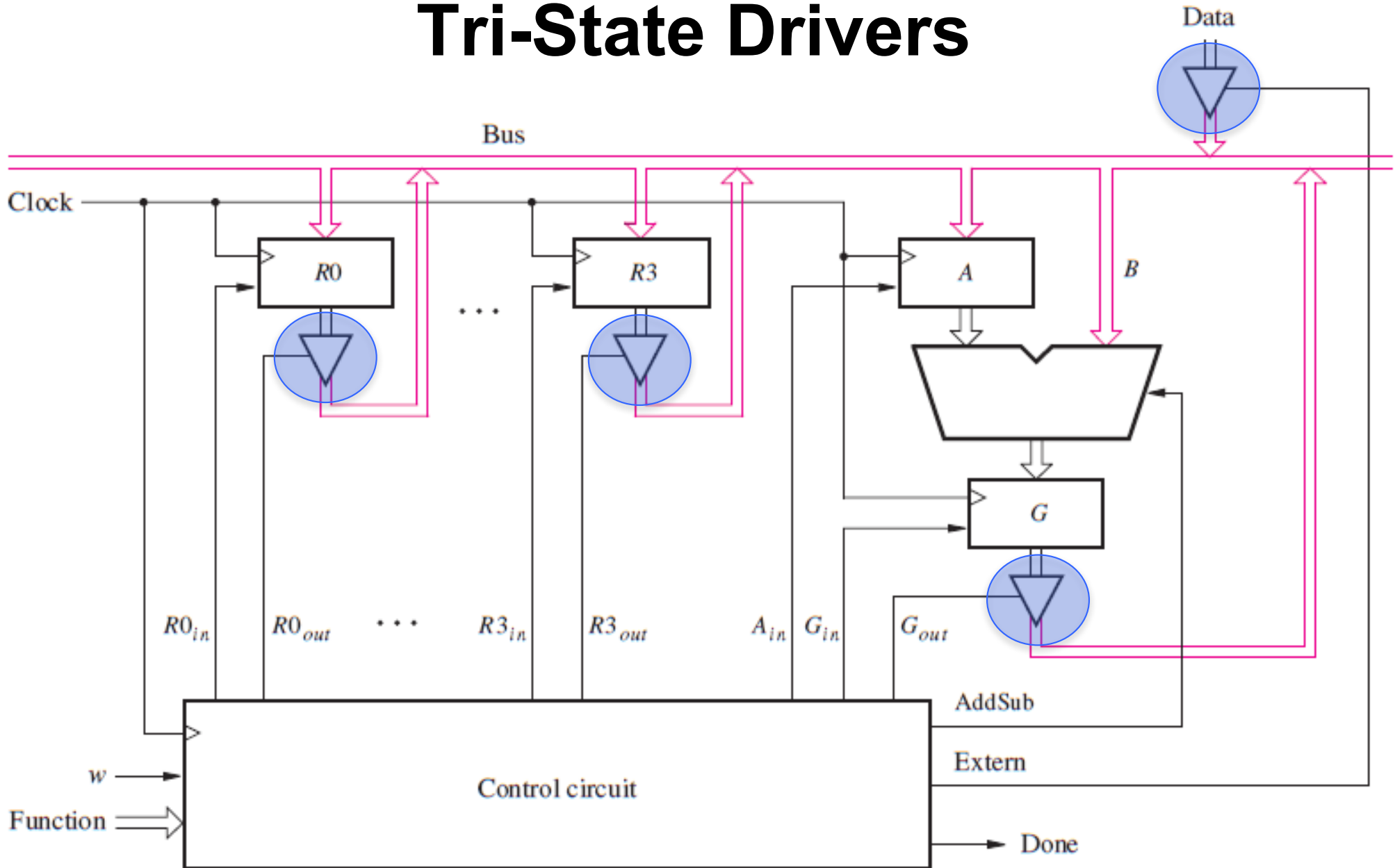[Figure 7.9 from the textbook]

Data Bus
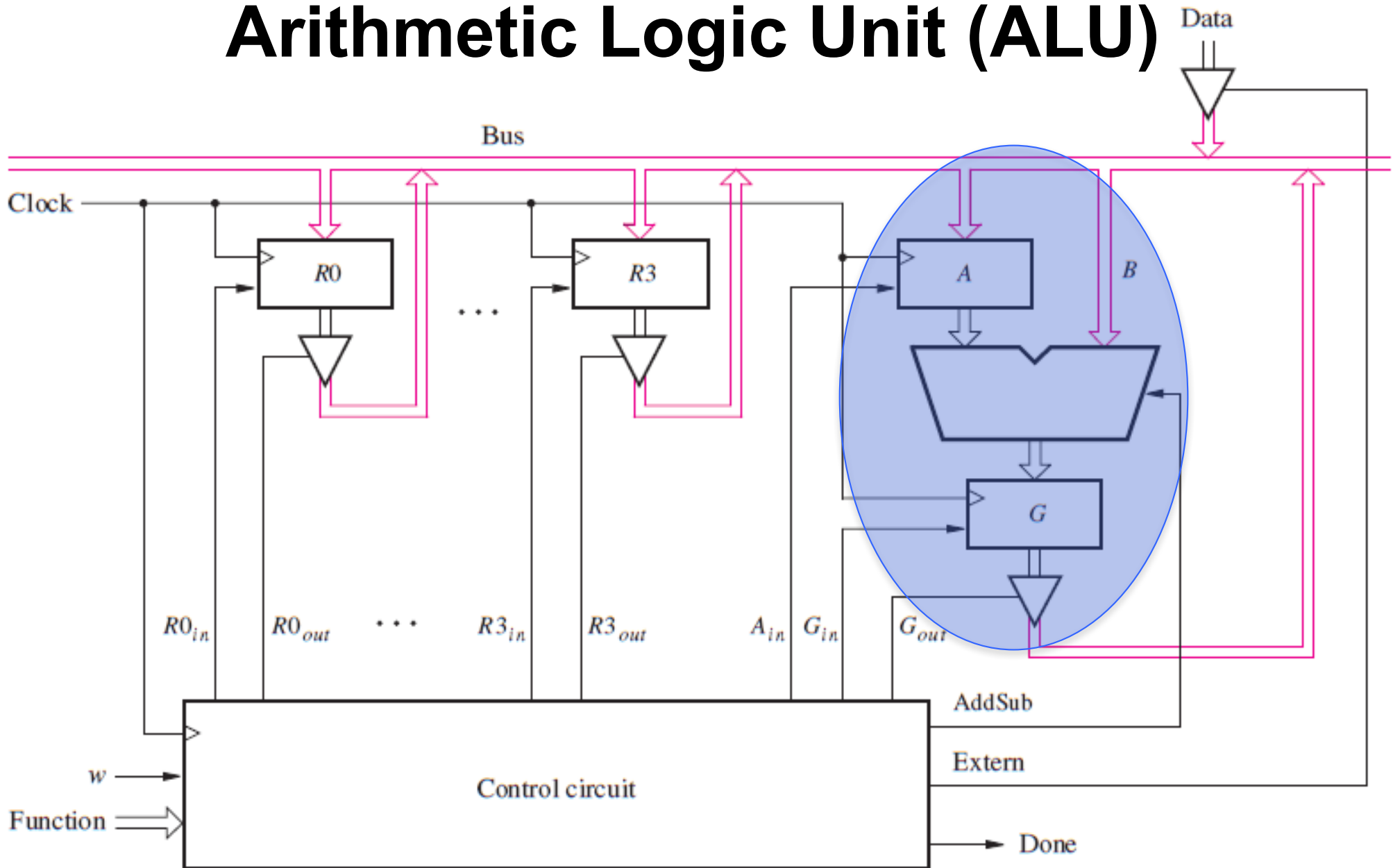
[Figure 7.9 from the textbook]

Tri-State Drivers

[Figure 7.9 from the textbook]

Arithmetic Logic Unit (ALU)

[Figure 7.9 from the textbook]

Control Circuit
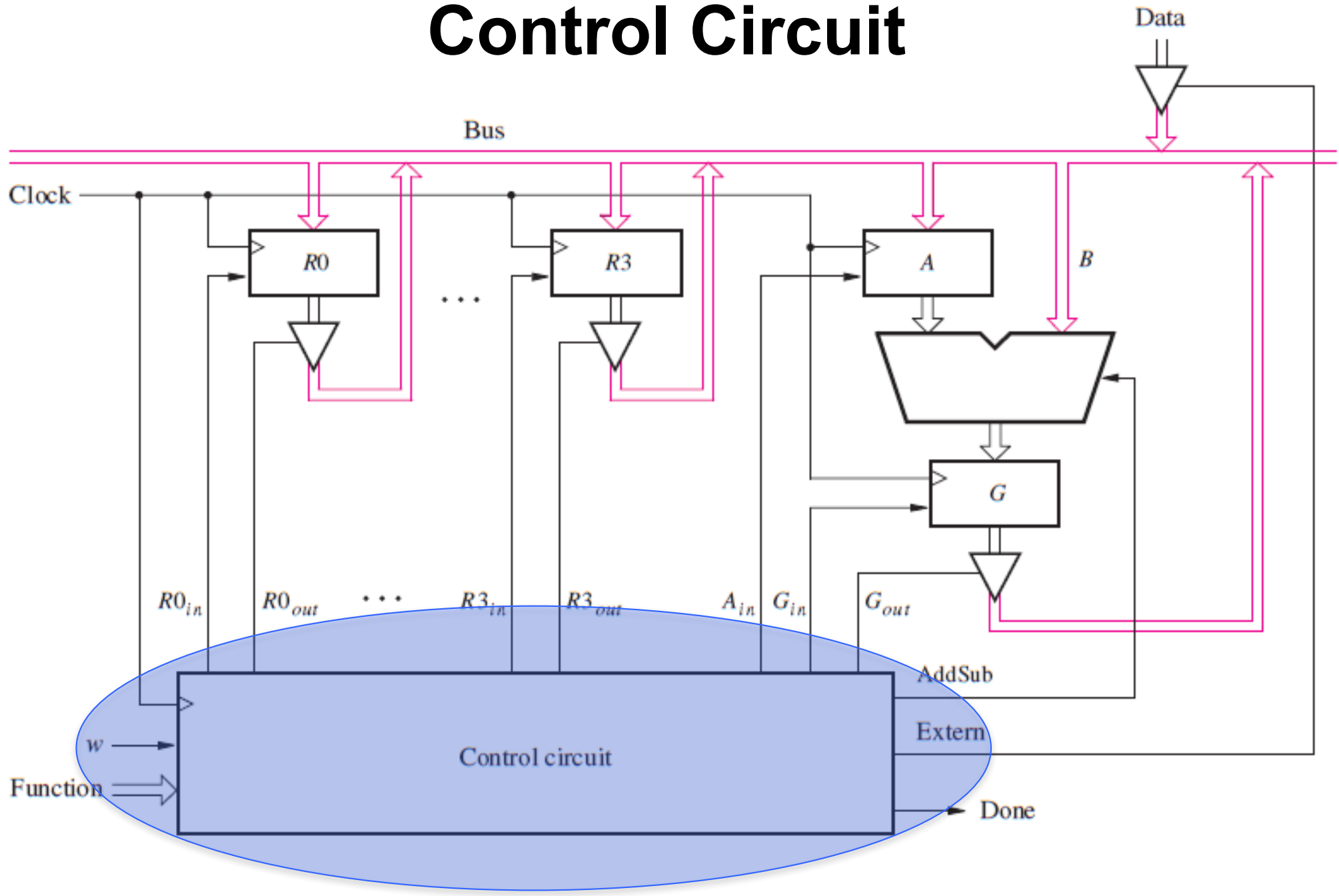
[Figure 7.9 from the textbook]

A Closer Look at the Registers

Register R0

[Figure 7.9 from the textbook]
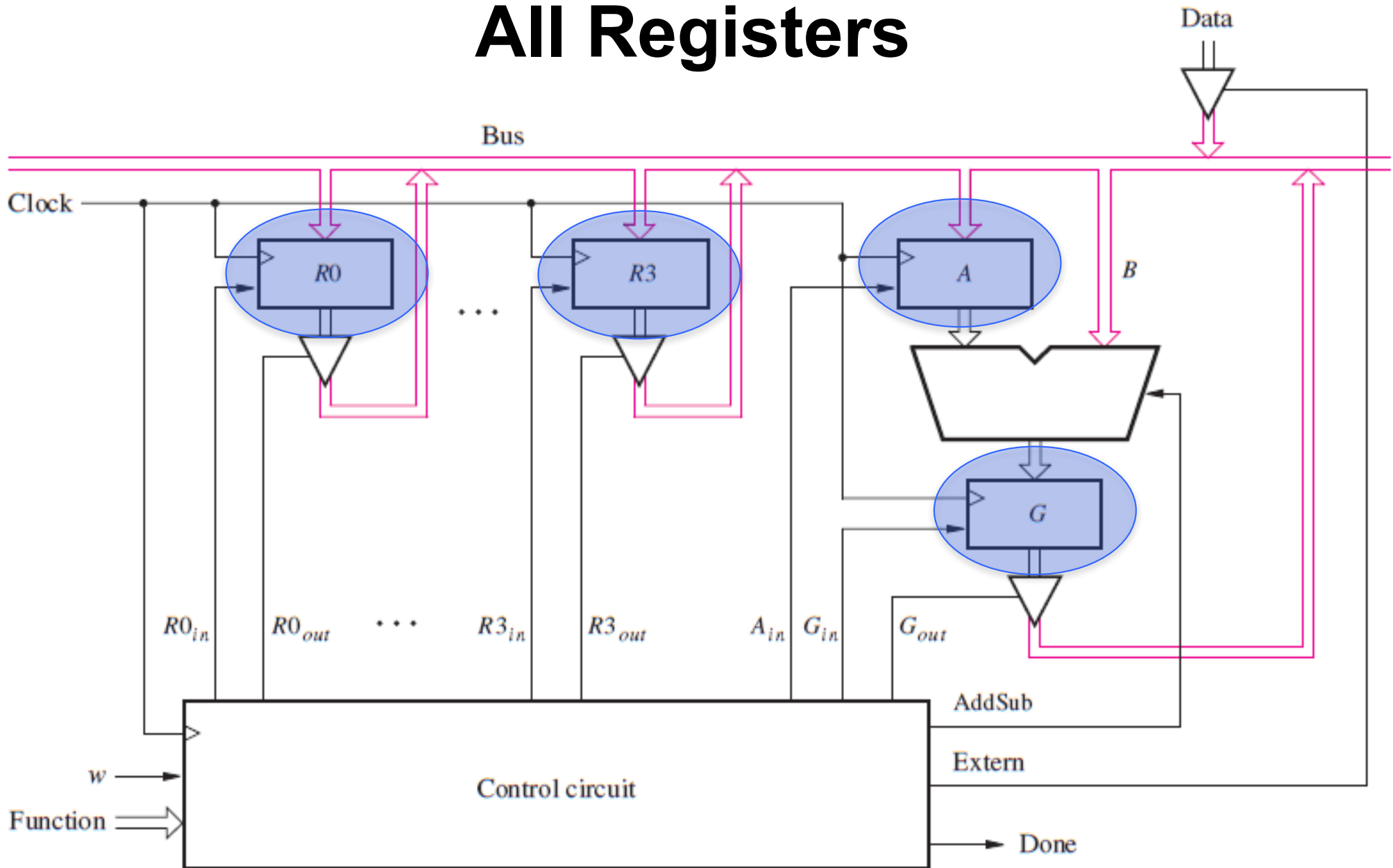
All Registers

[Figure 7.9 from the textbook]

Registers R0, R1, R2 and R3
are accessible to the programmer
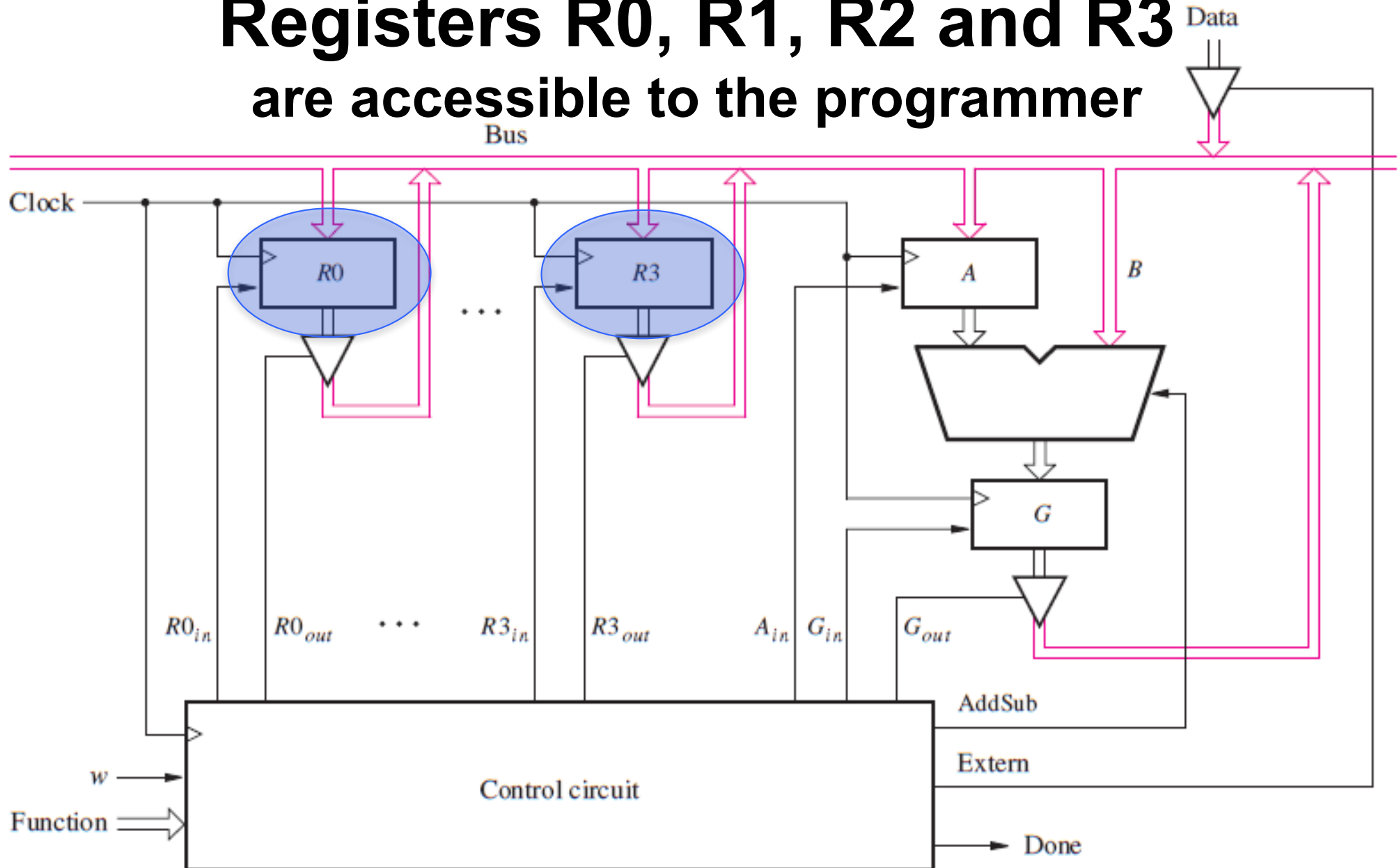
[Figure 7.9 from the textbook]

Registers A and G
are __NOT__ accessible to the programmer
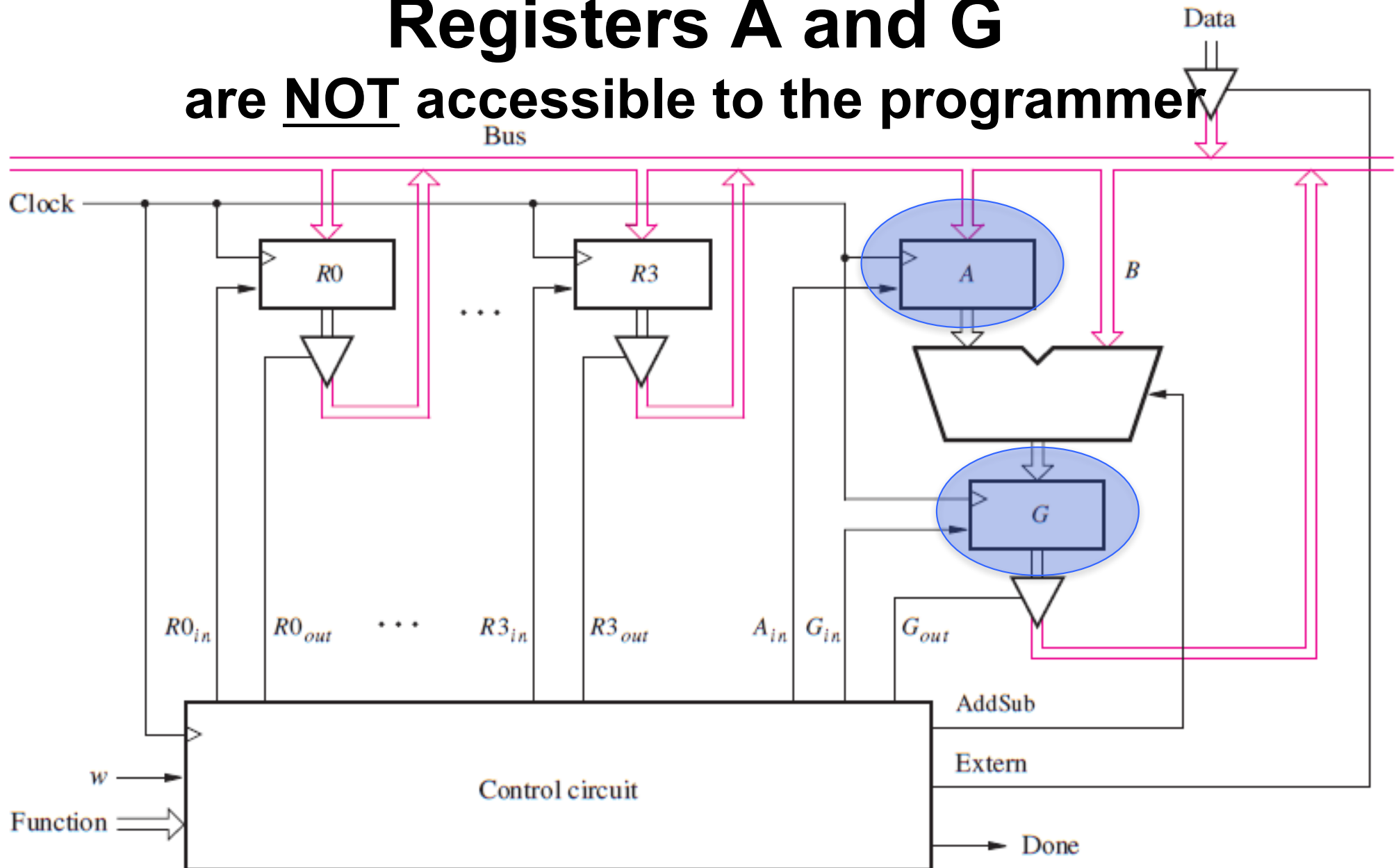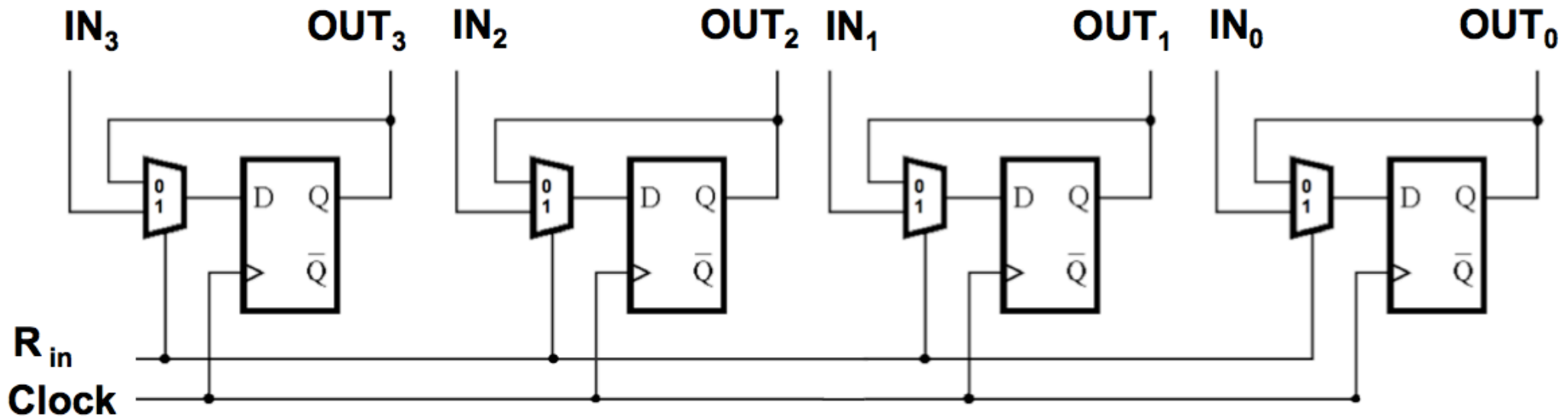
[Figure 7.9 from the textbook]

4-Bit Register

Loading Data into the Register
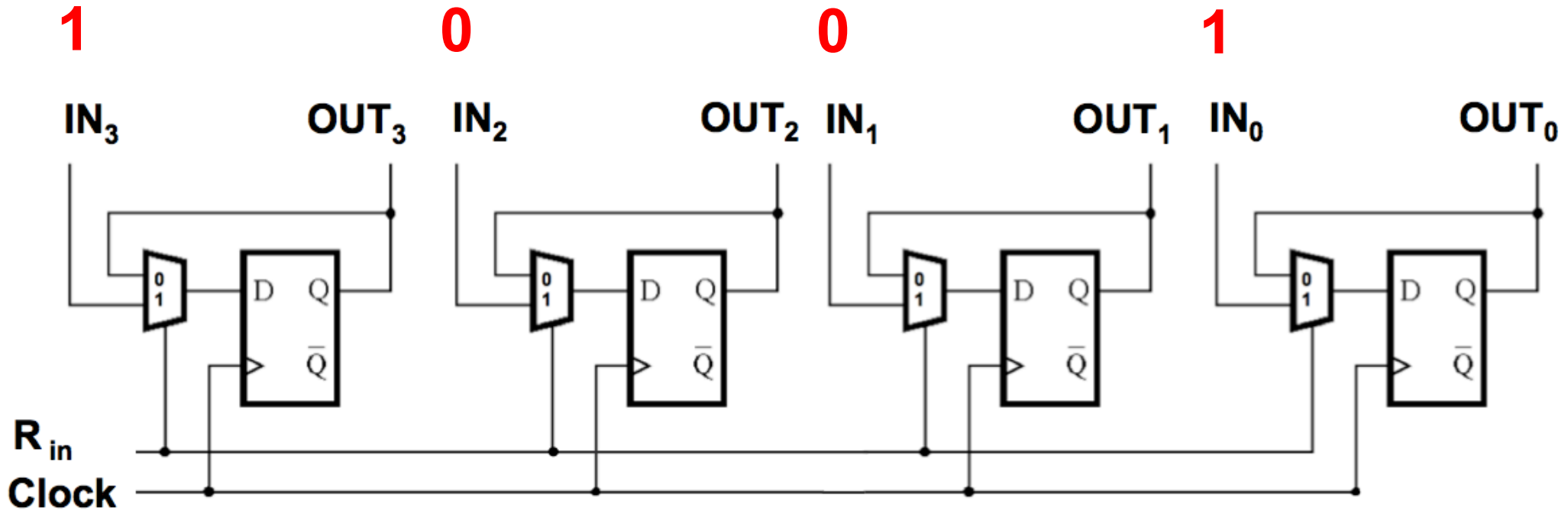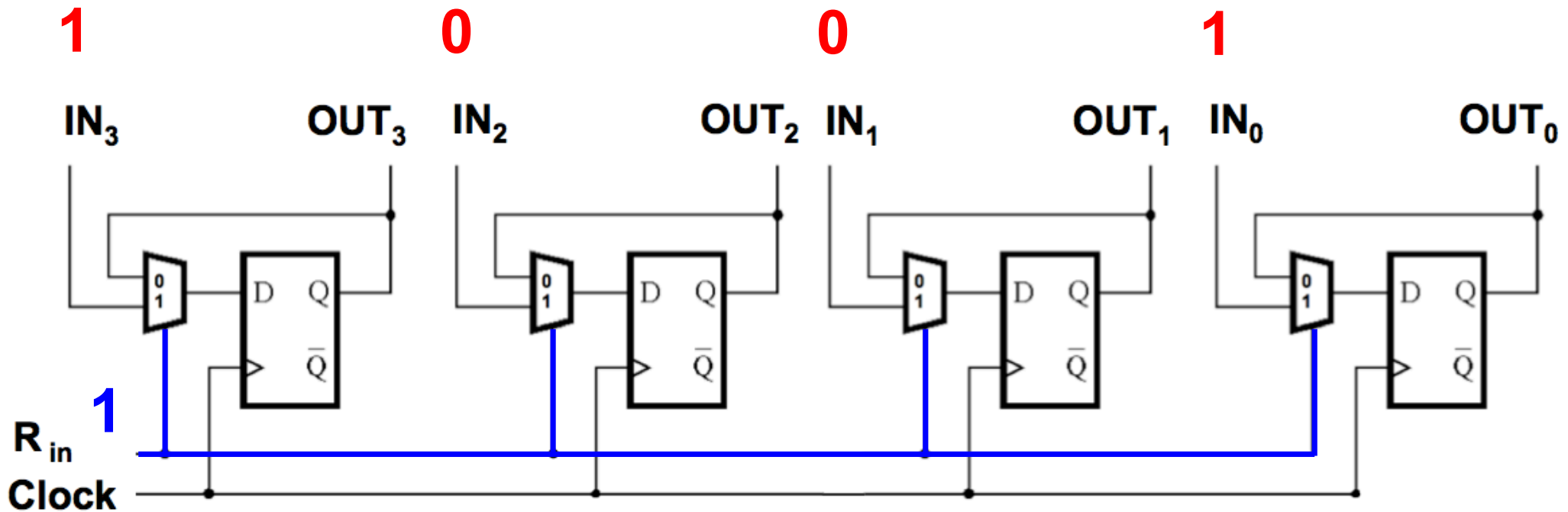
Loading Data into the Register

Loading Data into the Register
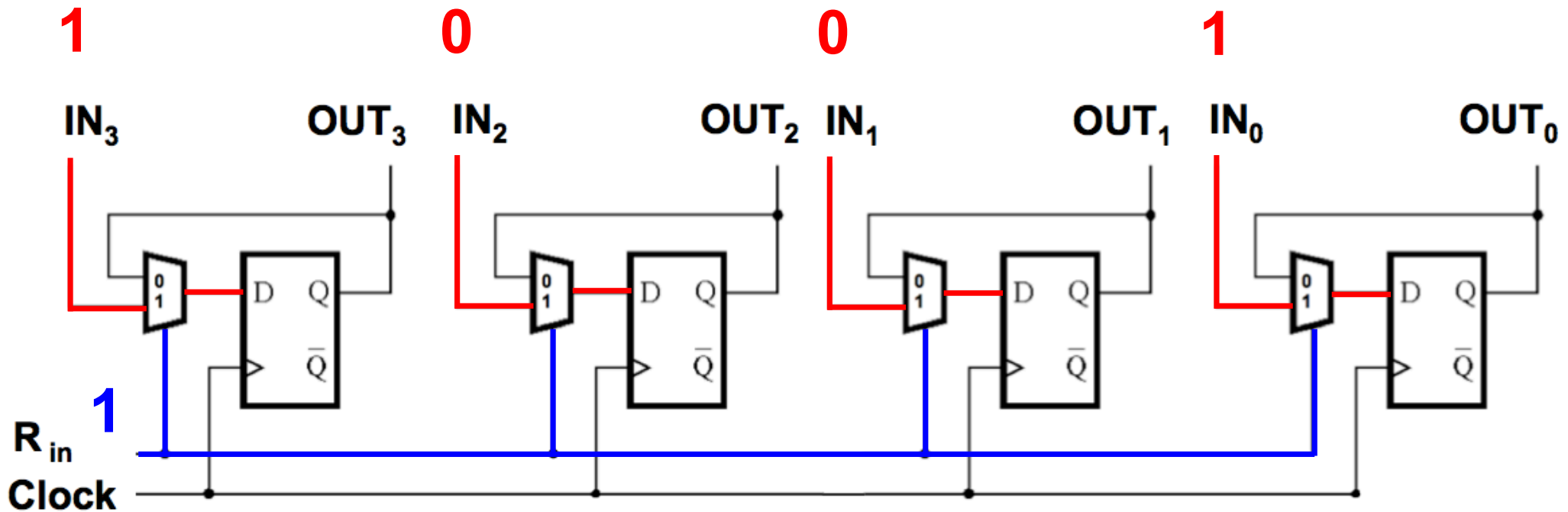
Loading Data into the Register
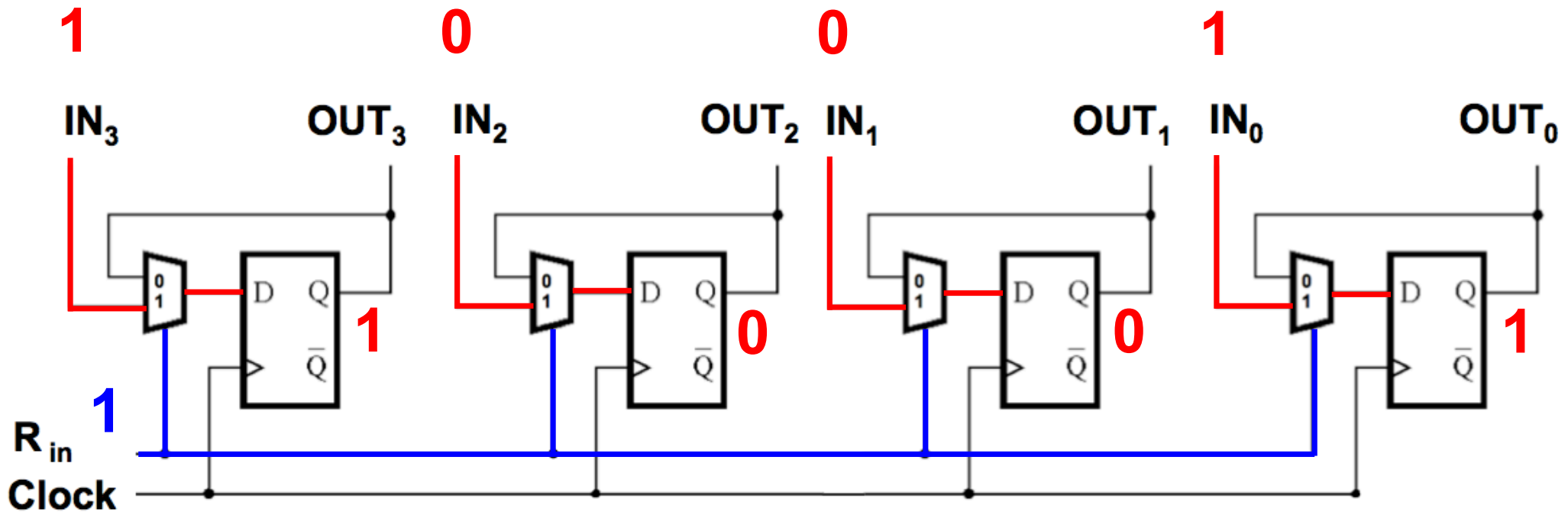
Keeping Data into the Register
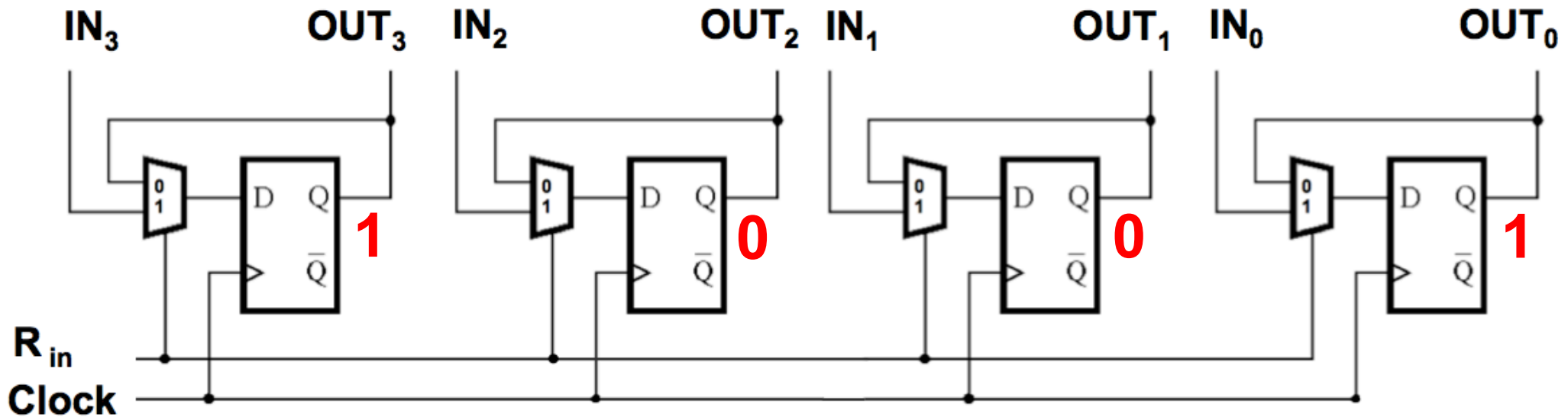
Keeping Data into the Register
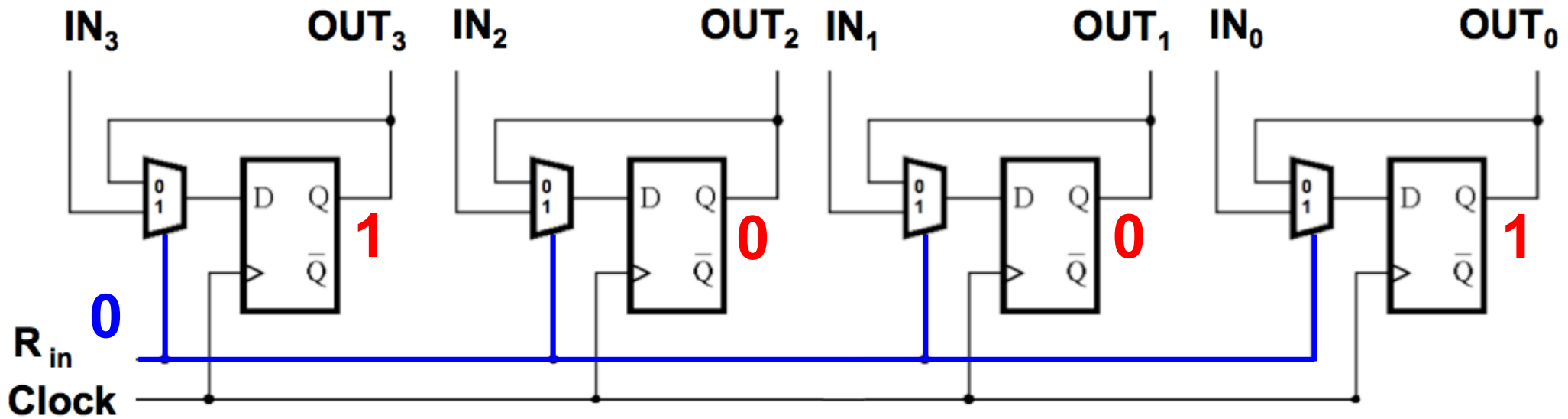
Keeping Data into the Register
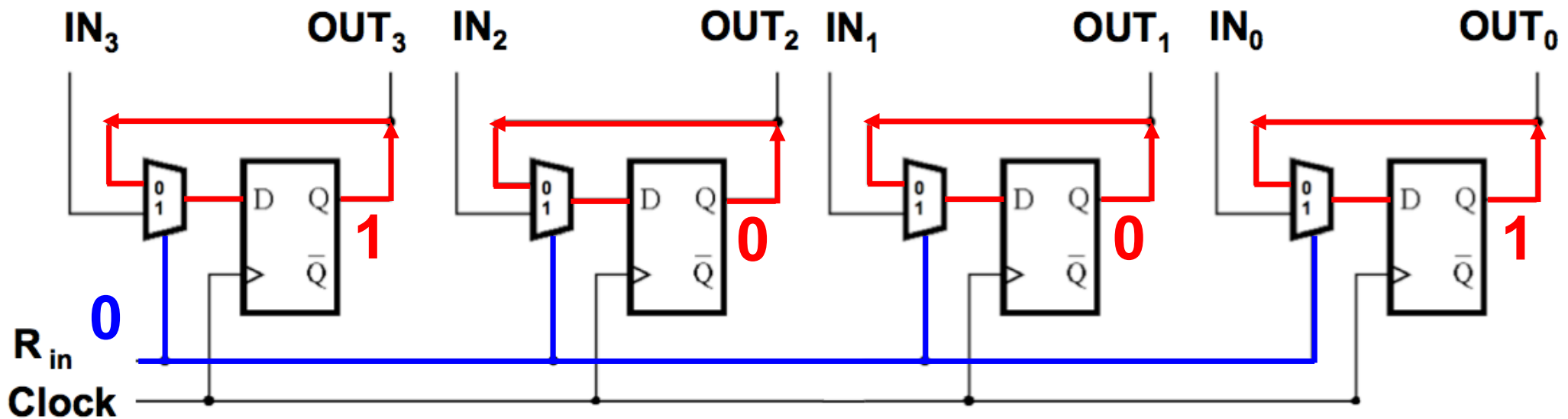
A Closer Look at the Data Bus

Data Bus

[Figure 7.9 from the textbook]

Bus Structure

- **We need a way to transfer data from any register (device) to any other register (device)**

- **A bus is simply a set of n wires to transfer n-bit data**

n

Bus
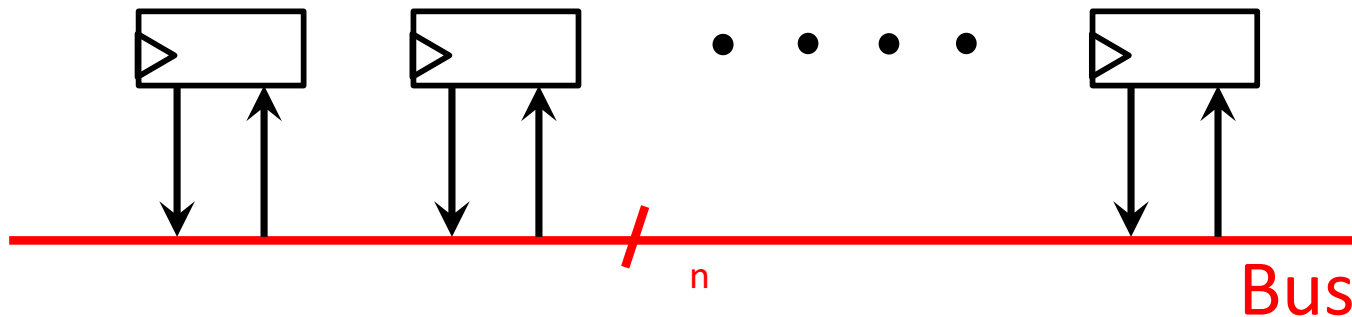
- **What if two registers write to the bus at the same time?**

One way to implement a data bus is to use multiplexers

[Figure 7.4 from the textbook]
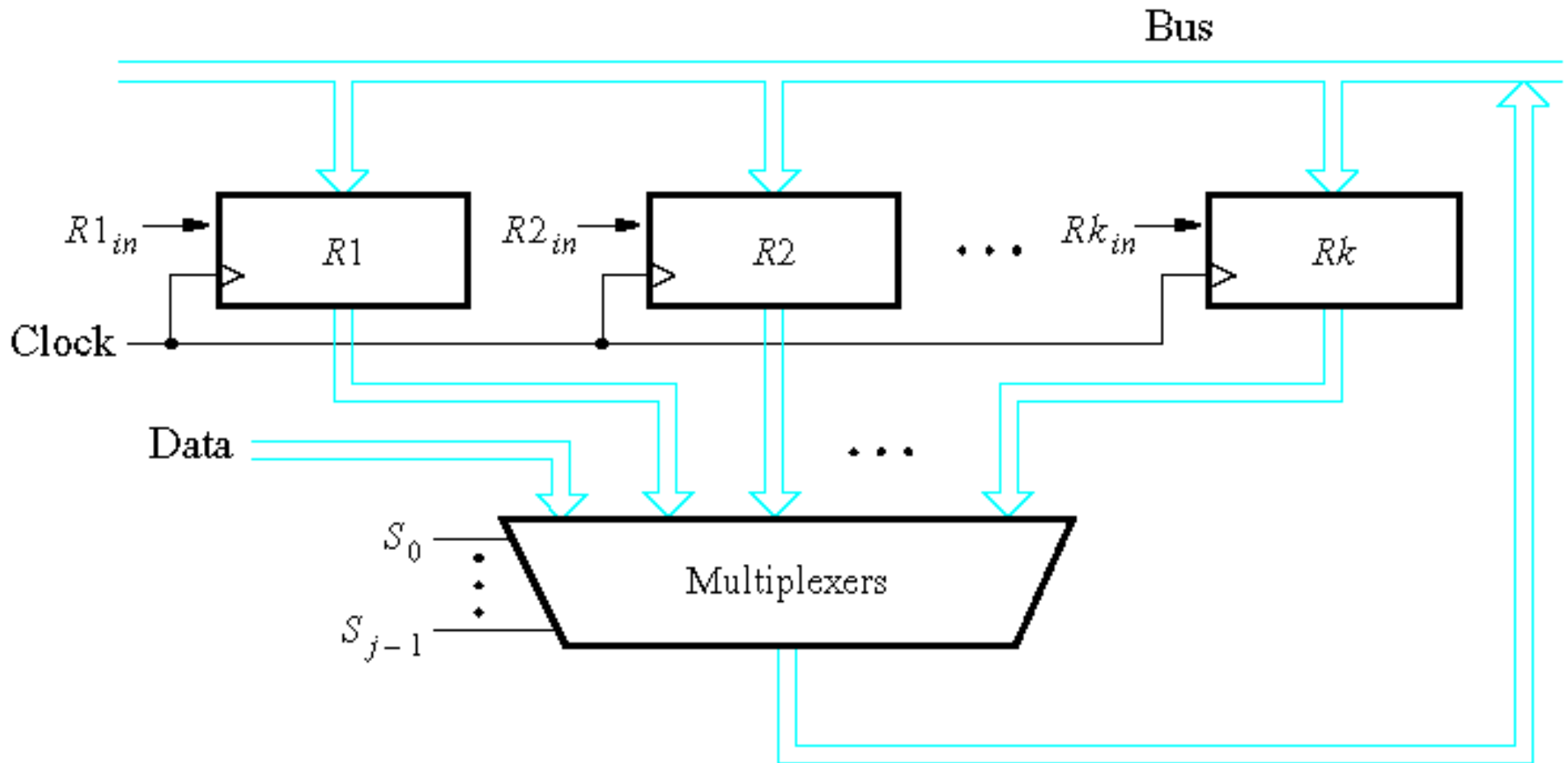
One way to implement a data bus is to use multiplexers
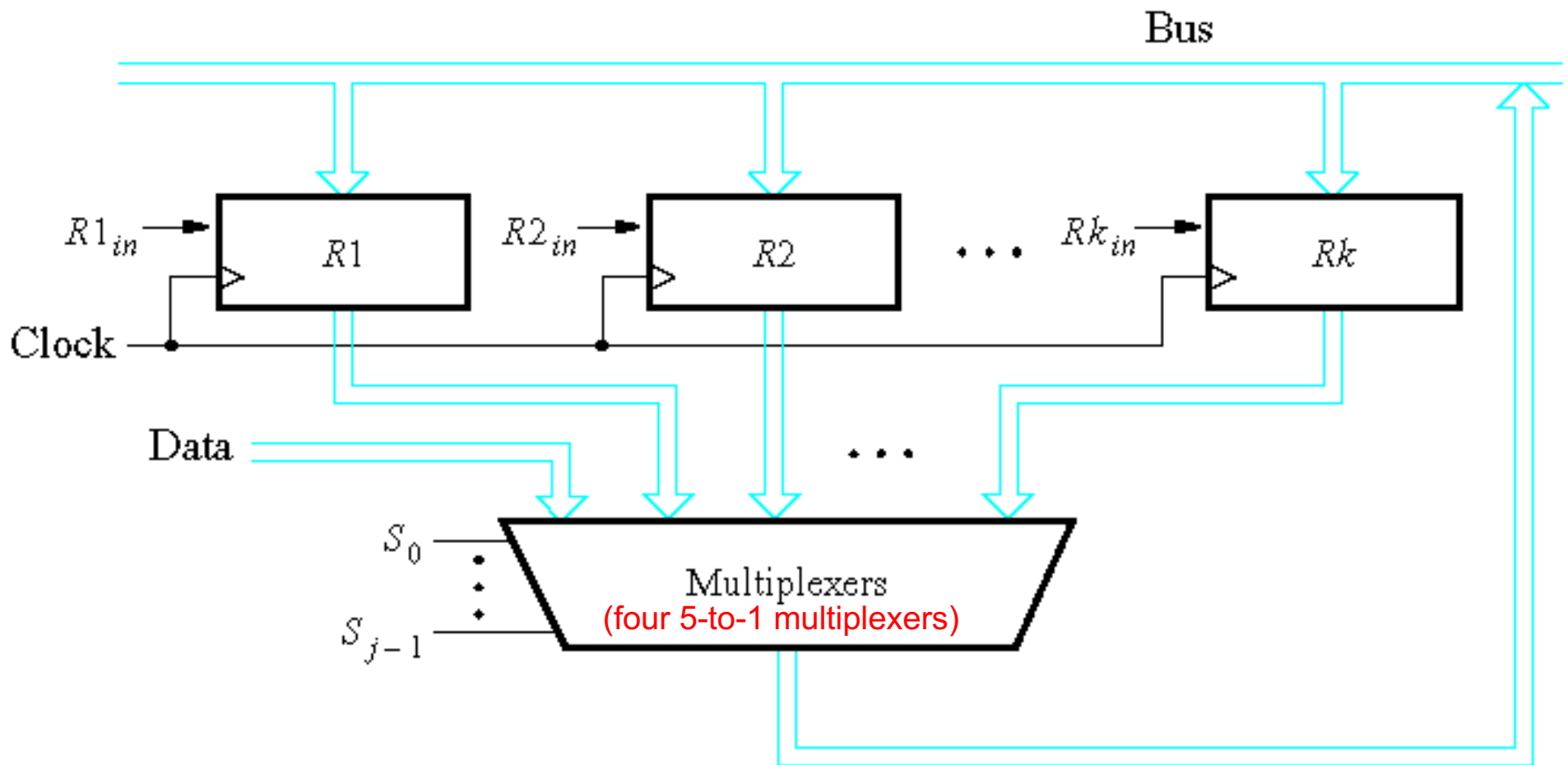
This requires one multiplexer per bit.
Assuming there are four 4-bit registers, we need four 5-to-1 multiplexers.

[Figure 7.4 from the textbook]

One way to implement a data bus is to use multiplexers

Bus

$R1_{in}$ R1 $R2_{in}$ R2 $\cdots$ Rk_{in} Rk

Clock

Data

S_0

S_{j-1}

Multiplexers
(four 5-to-1 multiplexers)

The reason we need
5-to-1 is because
the external data
counts as an "extra register".

[Figure 7.4 from the textbook]
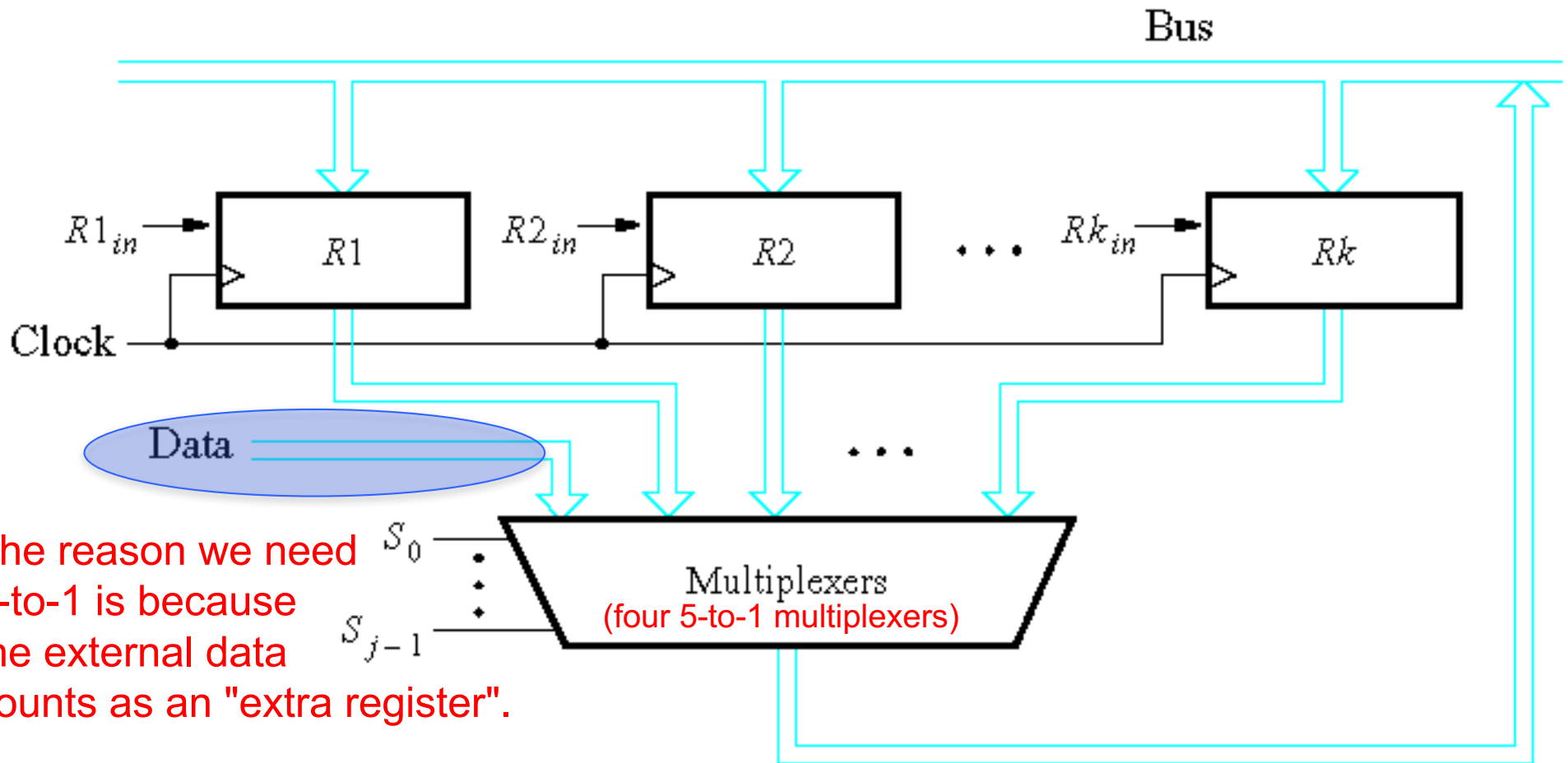
A Closer Look at the Tri-State Driver

Tri-State Driver
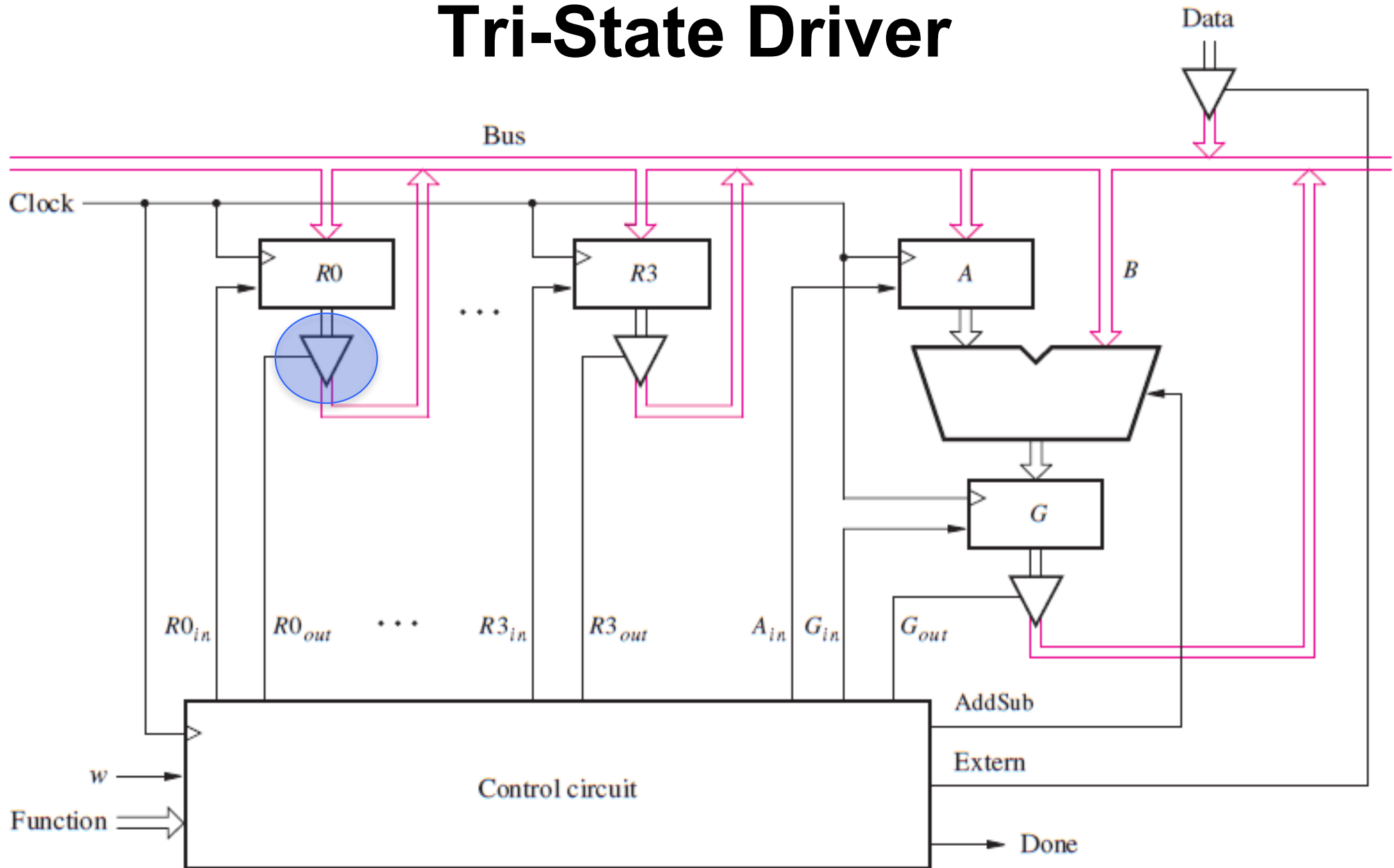
[Figure 7.9 from the textbook]

All Tri-State Drivers
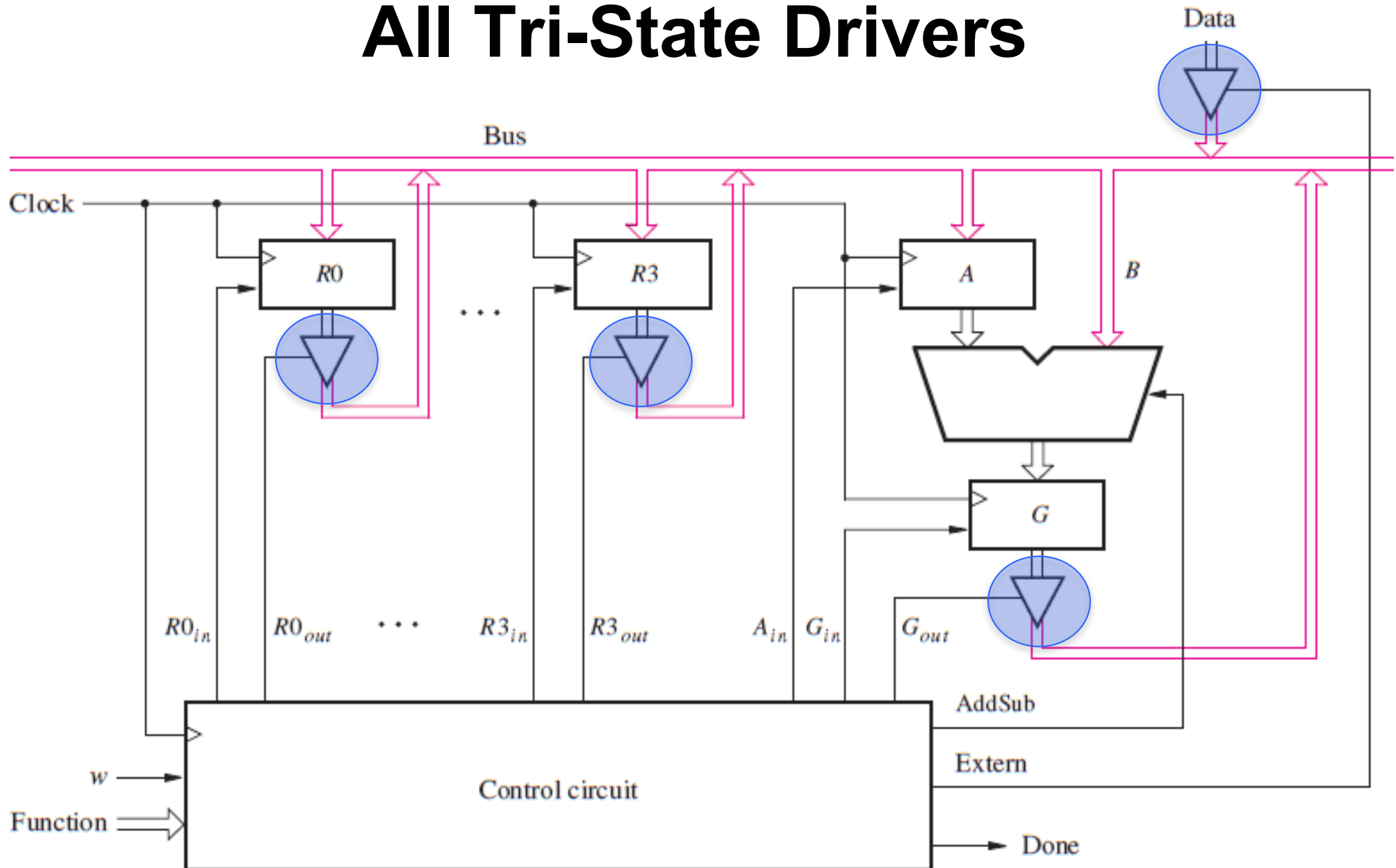
[Figure 7.9 from the textbook]

Tri-state driver
(see Appendix B for more details)

Z: High impedance state

$e = 0$

w —▷— f

$e = 1$

w —▷— f

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

e

w —▷— f
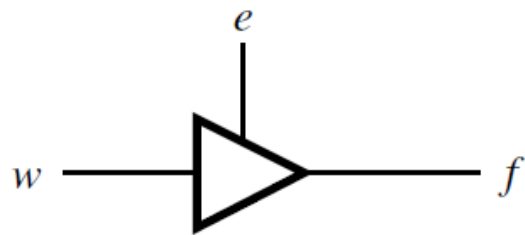
(a) Symbol
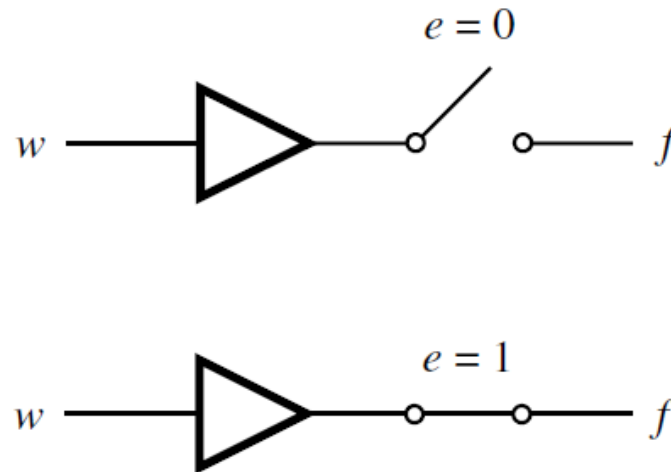
(b) Equivalent circuit

(c) Truth table

[Figure 7.1 from the textbook]

Tri-state driver
(see Appendix B for more details)

- **Alternative way to implement a data bus**

- **Allows several devices to be connected to a single wire (this is not possible with regular logic gates because their outputs are always active; an OR gate is needed)**

- **Note that at any time, at most one of e0, e1, e2, and e3 can be set to 1**
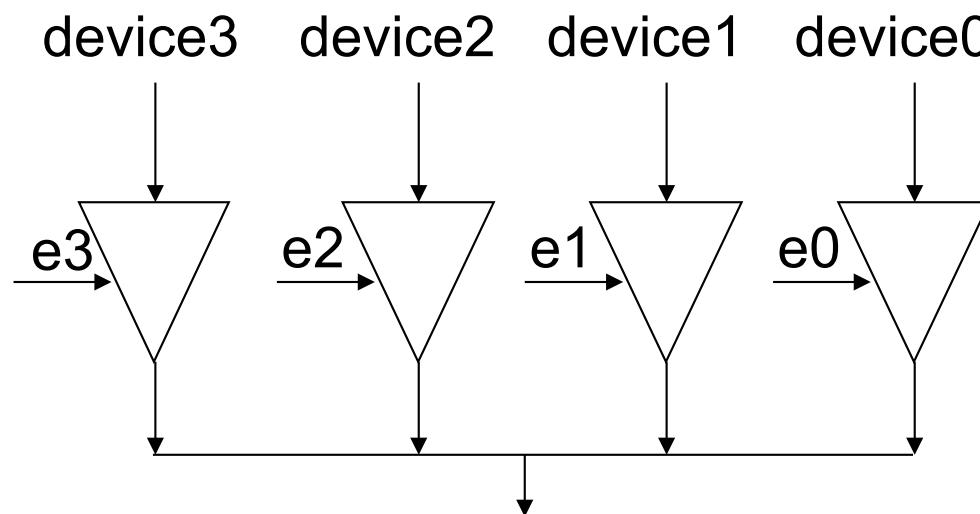
An n-bit Tri-State Driver
can be constructed using n 1-bit tri-state buffers

Input

e

Output

e   e e e
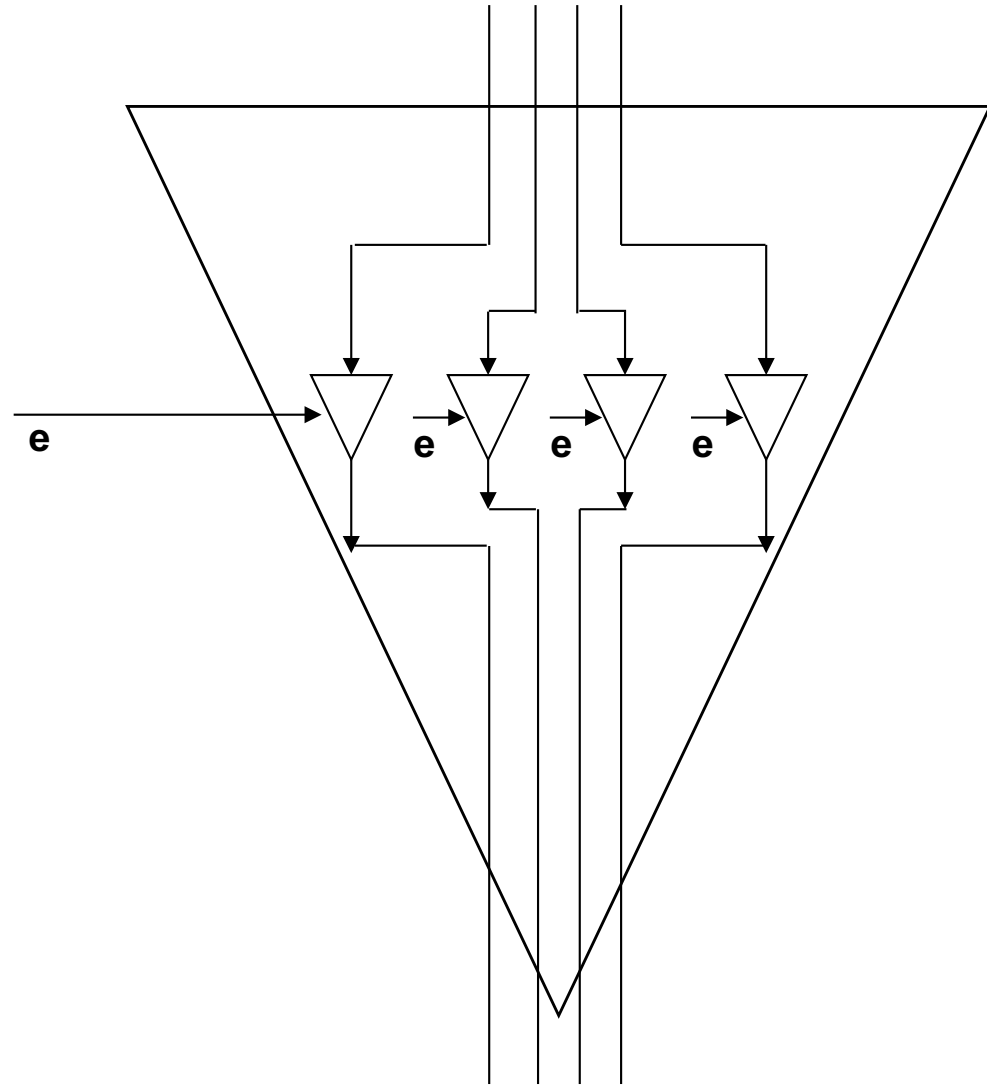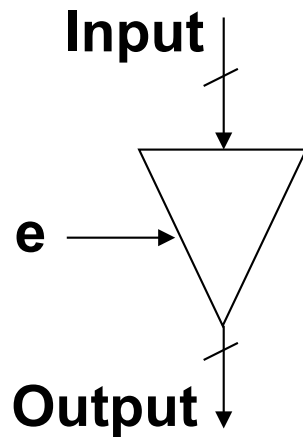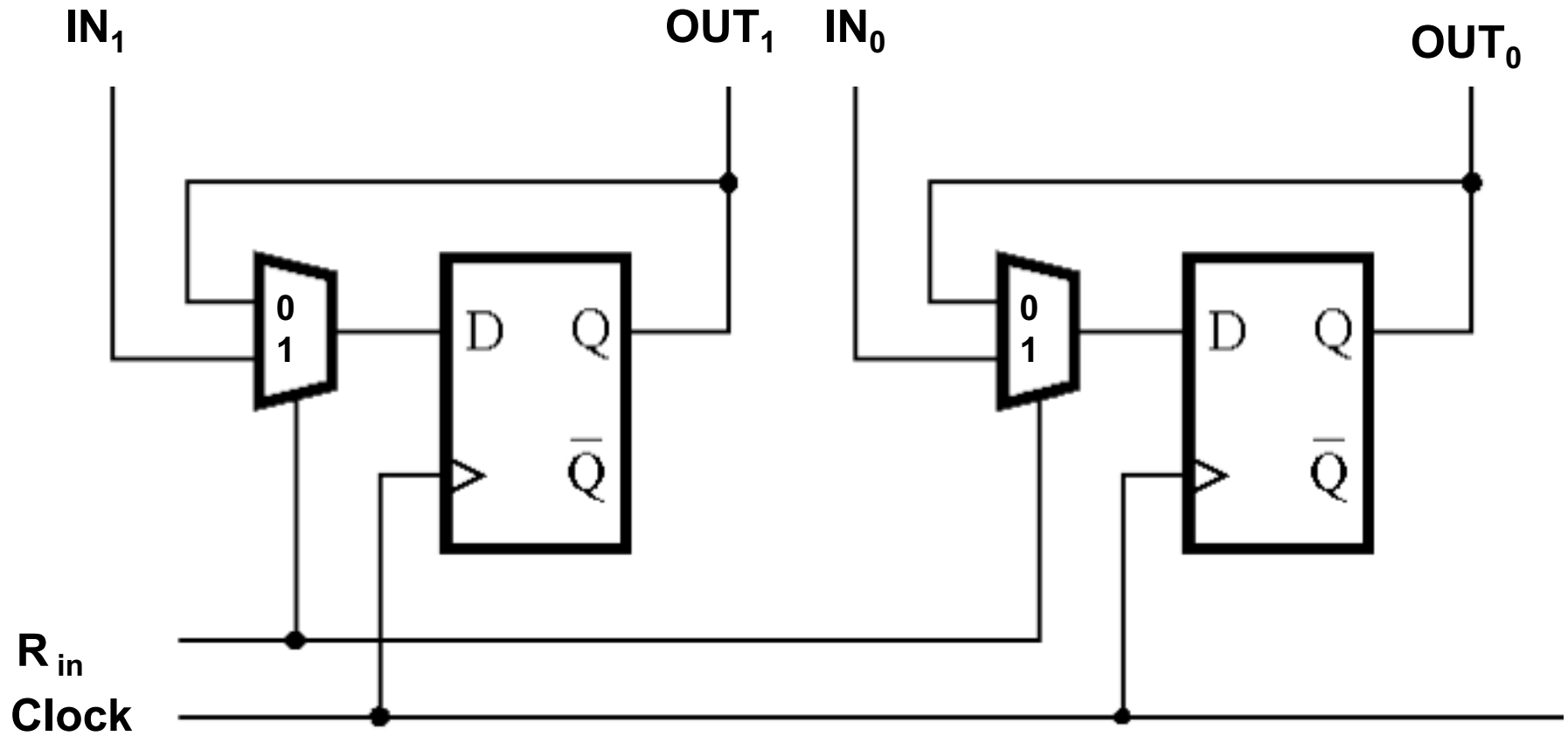
2-Bit Register

How to connect two 2-bit registers to a bus (using tri-state drivers)
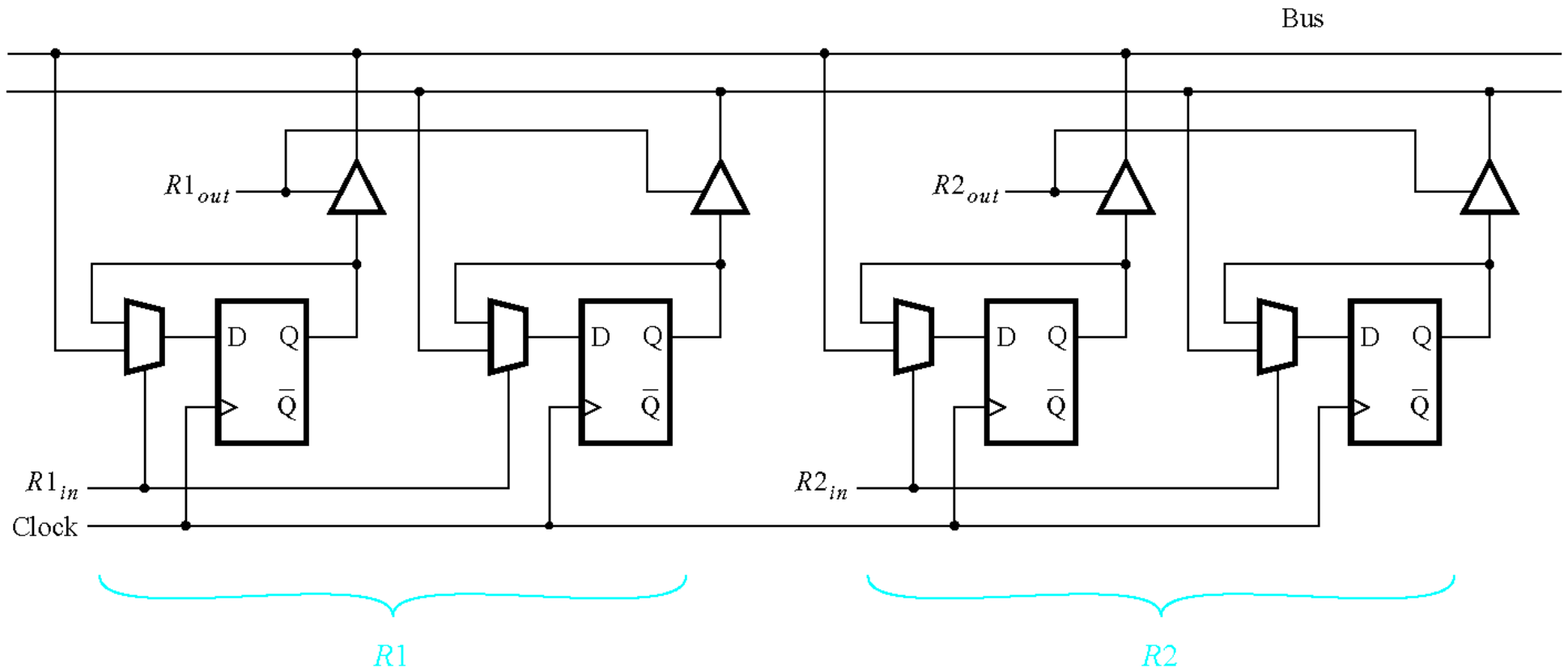
This shows only two 2-bit registers, but this design scales to more and larger registers.

[Figure 7.3 from the textbook]

Moving the Contents of R1 to R2
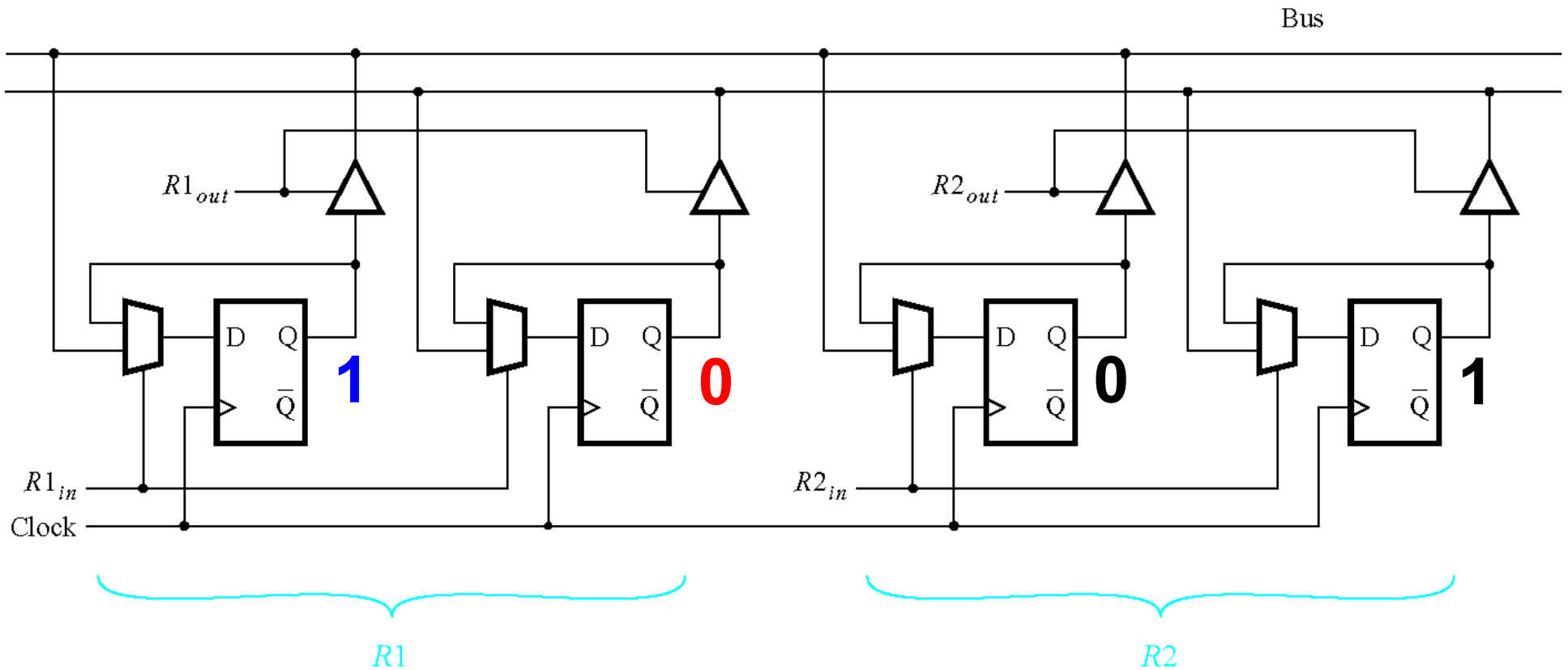
Register 1 stores the number $2_{10} = 10_2$ **Register 2 stores the number $1_{10} = 01_2$**

Moving the Contents of R1 to R2

Initially all control inputs are set to 0 (no reading or writing allowed).

Moving the Contents of R1 to R2
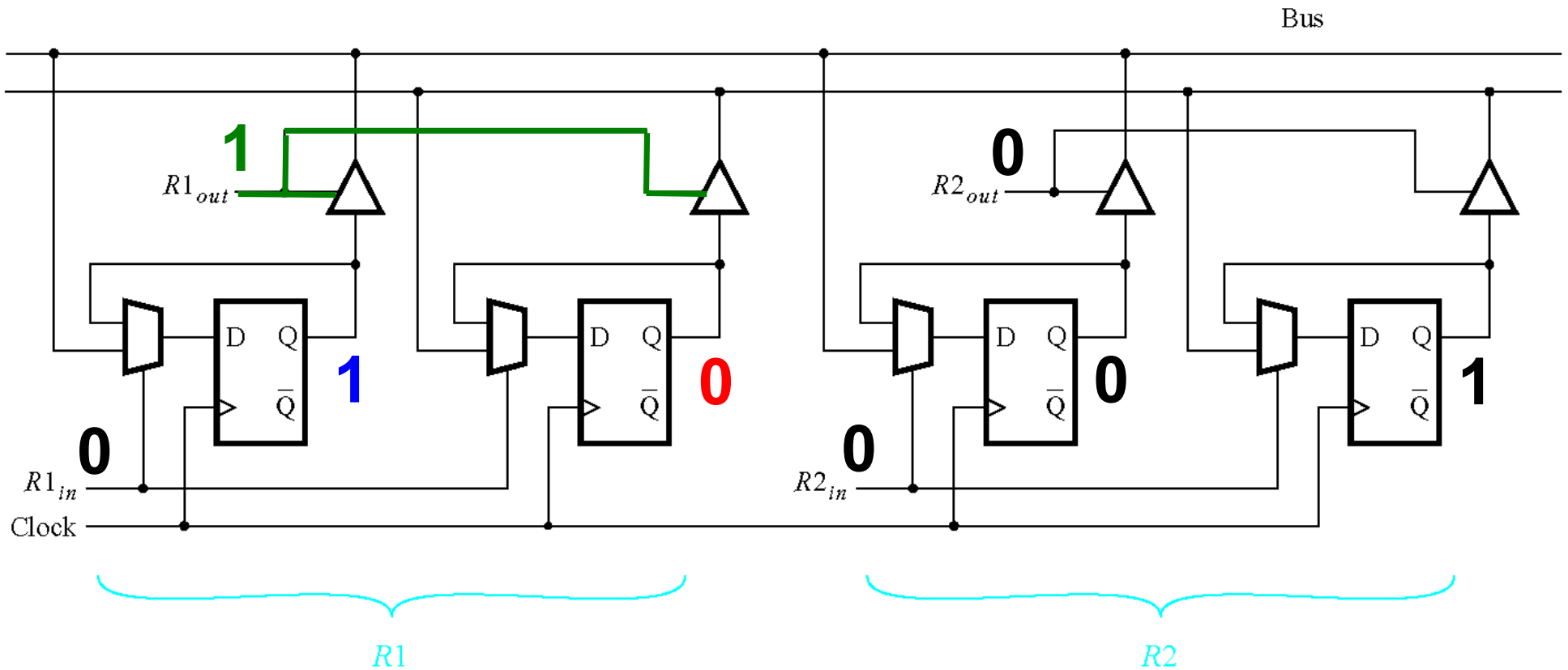
R1_{out} is set to 1 (this enables reading from register 1).

Moving the Contents of R1 to R2
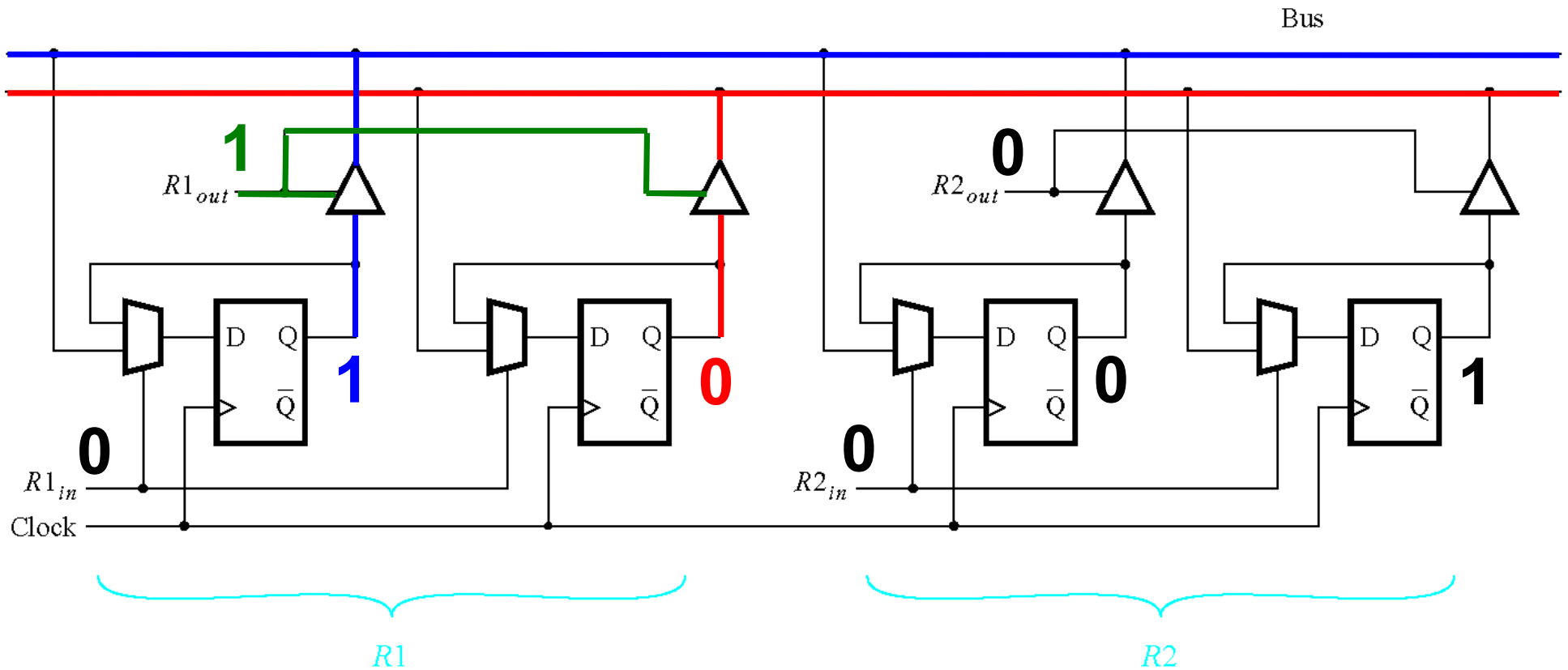
The bits of R1 are now on the data bus (2-bit data bus in this case).

Moving the Contents of R1 to R2
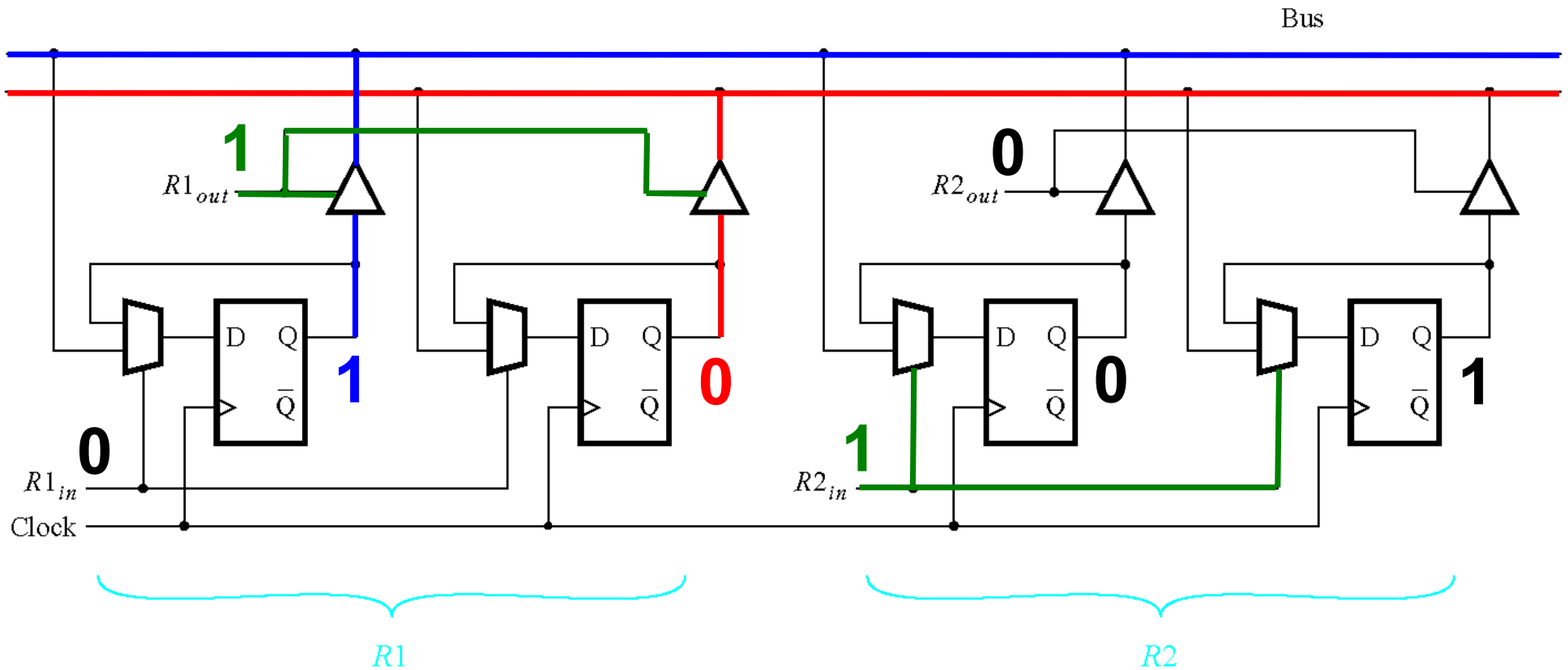
R2$_{in}$ is set to 1 (this enables writing to register 2).

Moving the Contents of R1 to R2
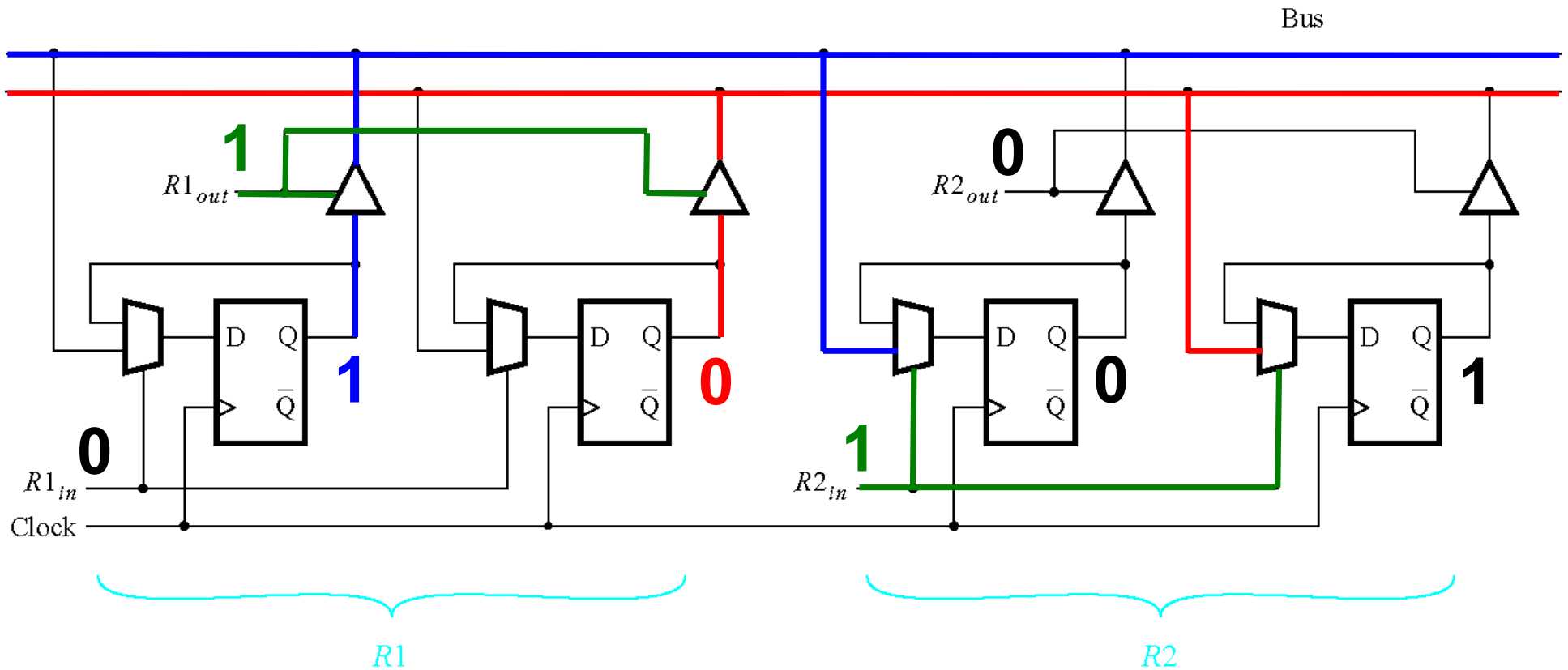
The bits of R1 are still on the bus and they propagate to the multiplexers...

Moving the Contents of R1 to R2
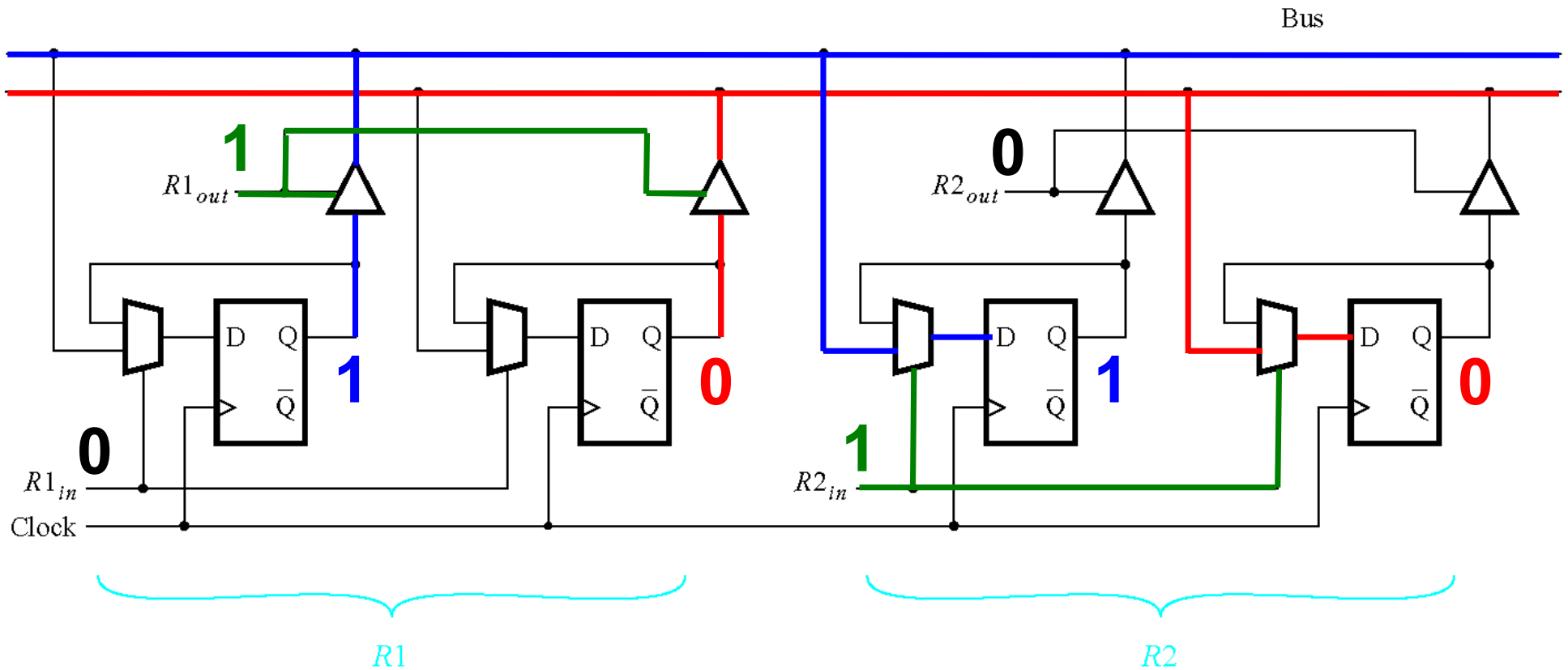
... and on the next positive clock edge to the outputs of the flip-flops of R2.

Moving the Contents of R1 to R2
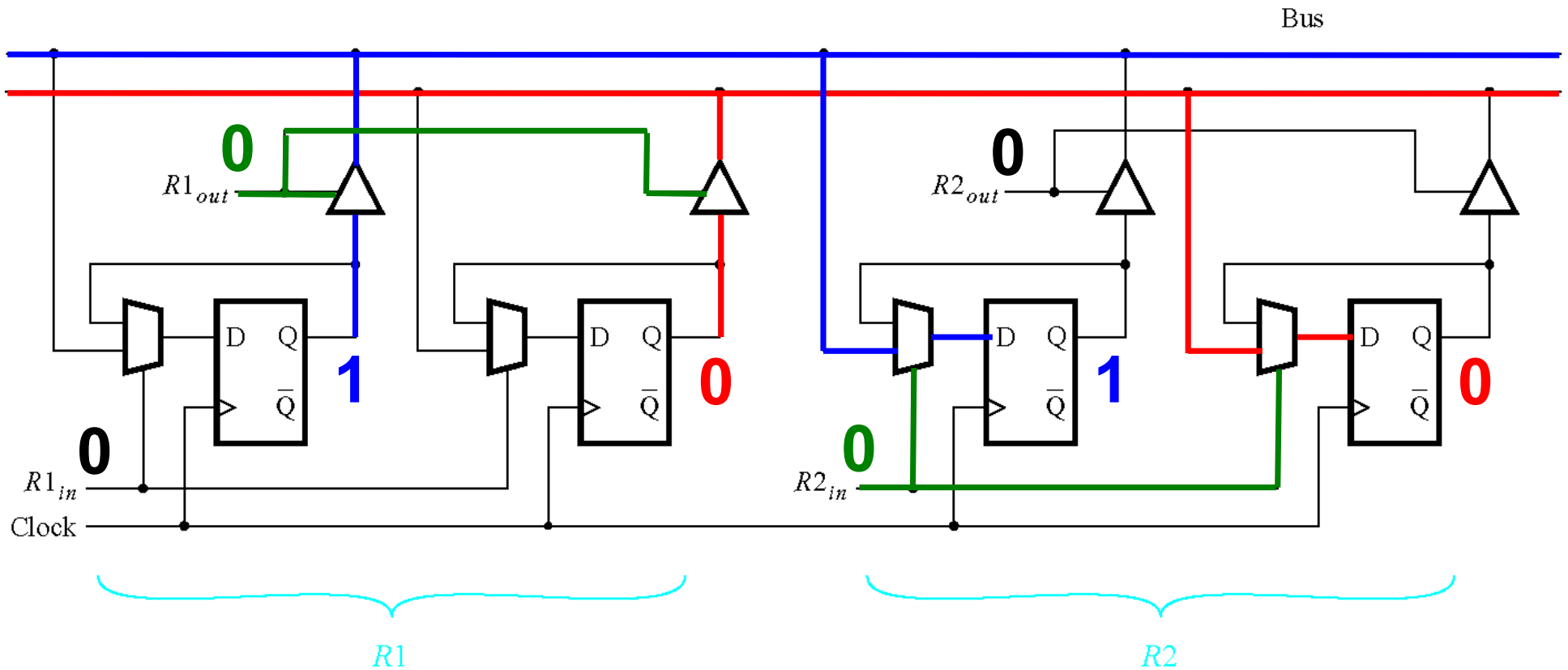
After the copy is complete $R1_{out}$ and $R2_{in}$ are set to 0.

Moving the Contents of R1 to R2
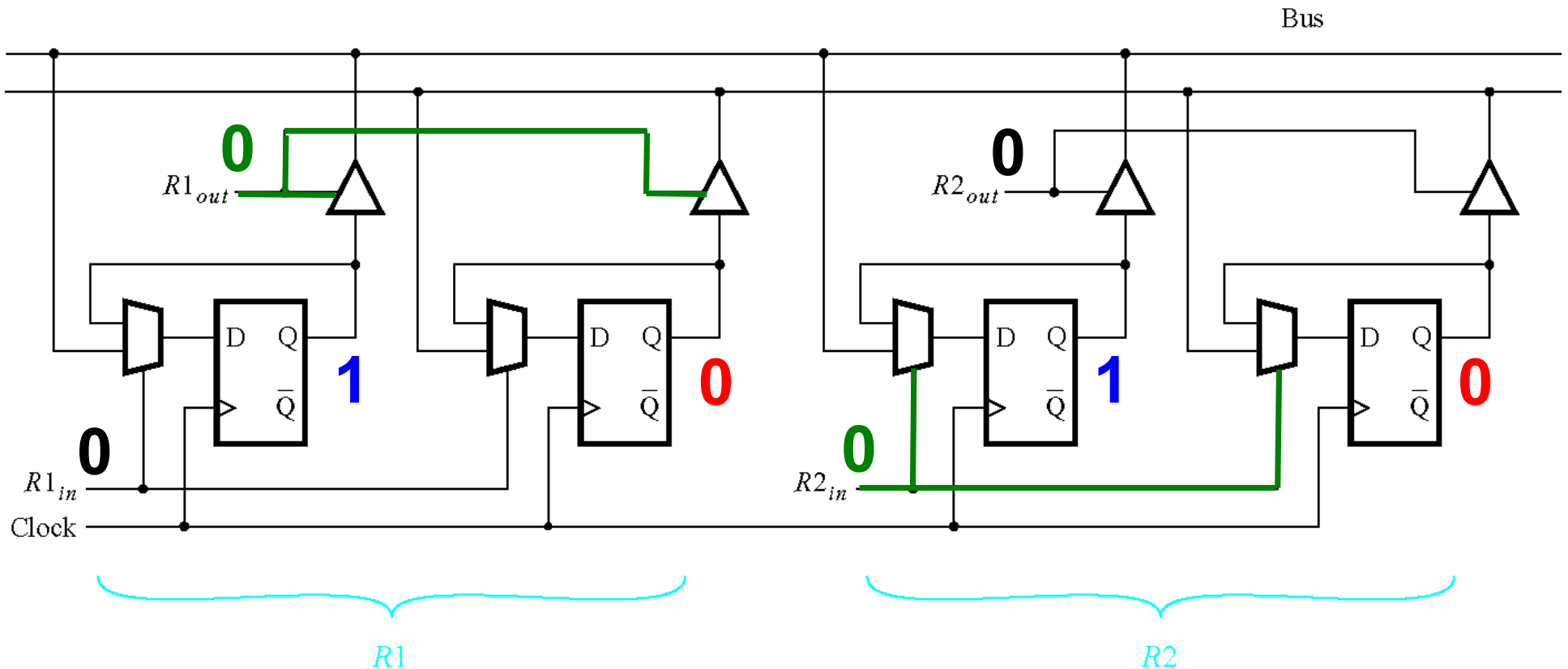
All control inputs are now disabled (no reading or writing is allowed).

Moving the Contents of R1 to R2

Register 2 now holds the same value as register 1.

Another Example

Loading Data From The Bus Into R2
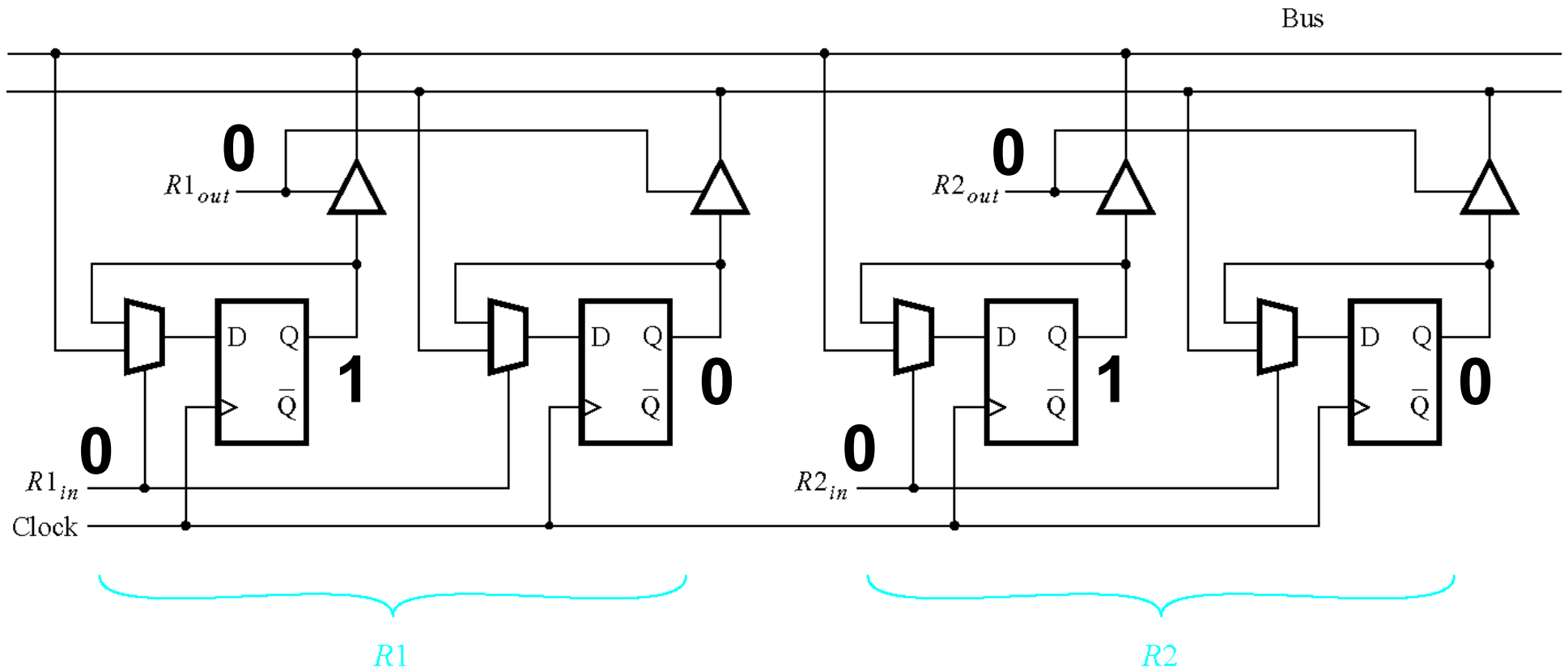
Initially all control inputs are set to 0 (no reading or writing allowed).

Loading Data From The Bus Into R2
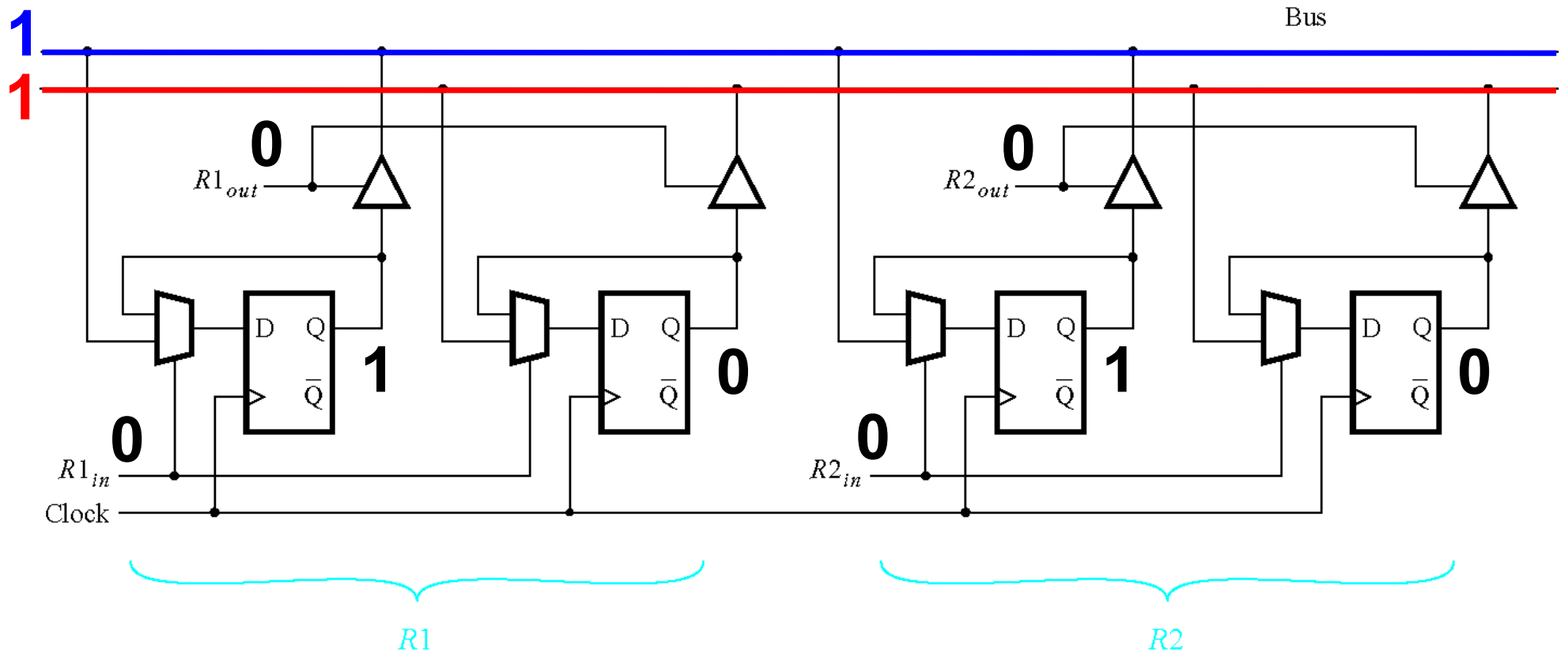
The number $3_{10} = 11_2$ is placed on the 2-bit data bus.

Loading Data From The Bus Into R2
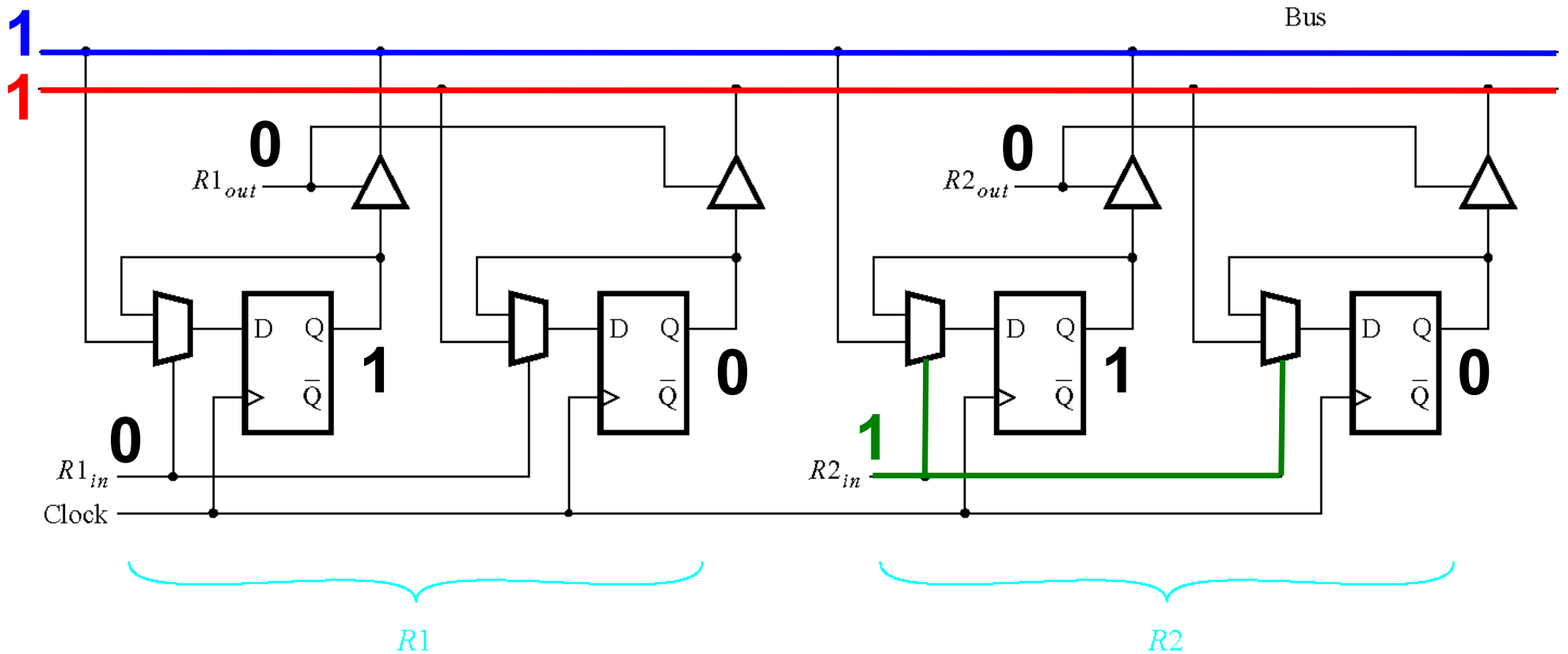
R2$_{in}$ is set to 1 (this enables writing to register 2).

Loading Data From The Bus Into R2
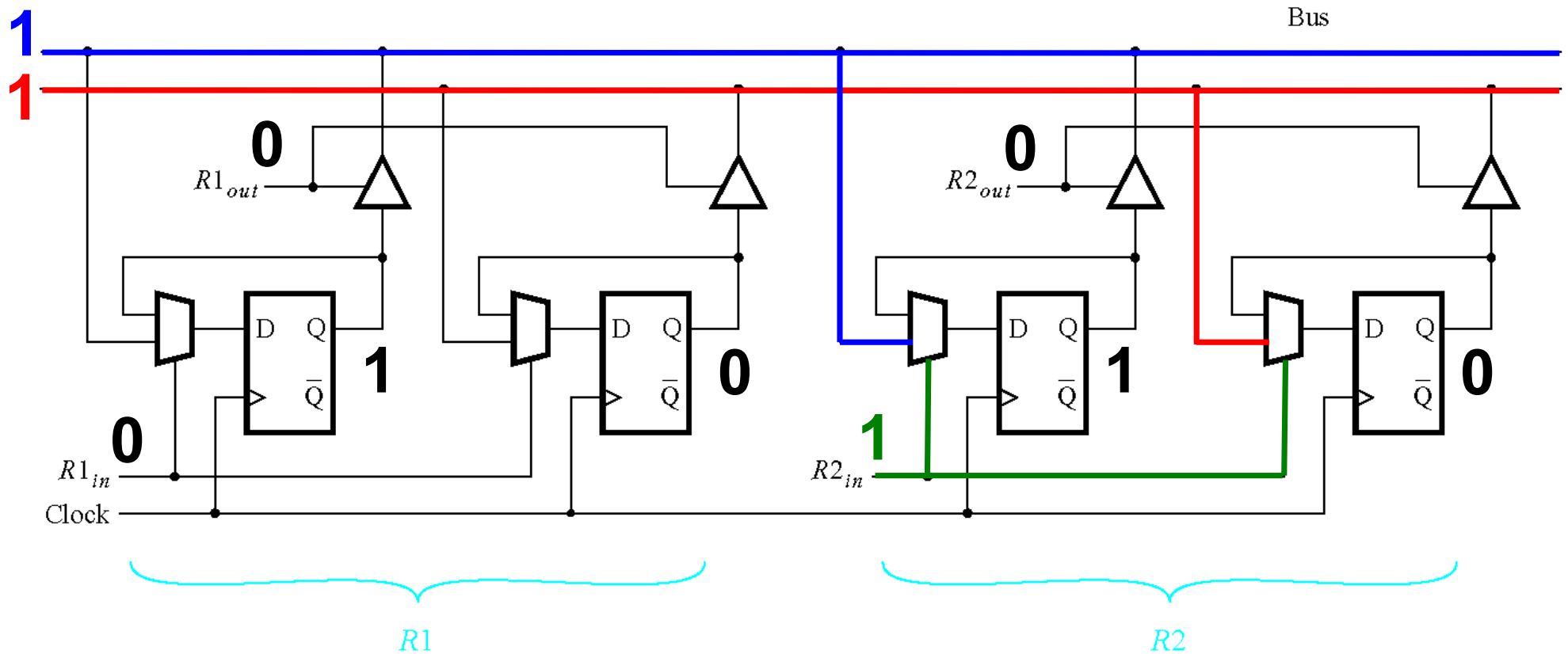
The bits of the data propagate the the multiplexers...

Loading Data From The Bus Into R2
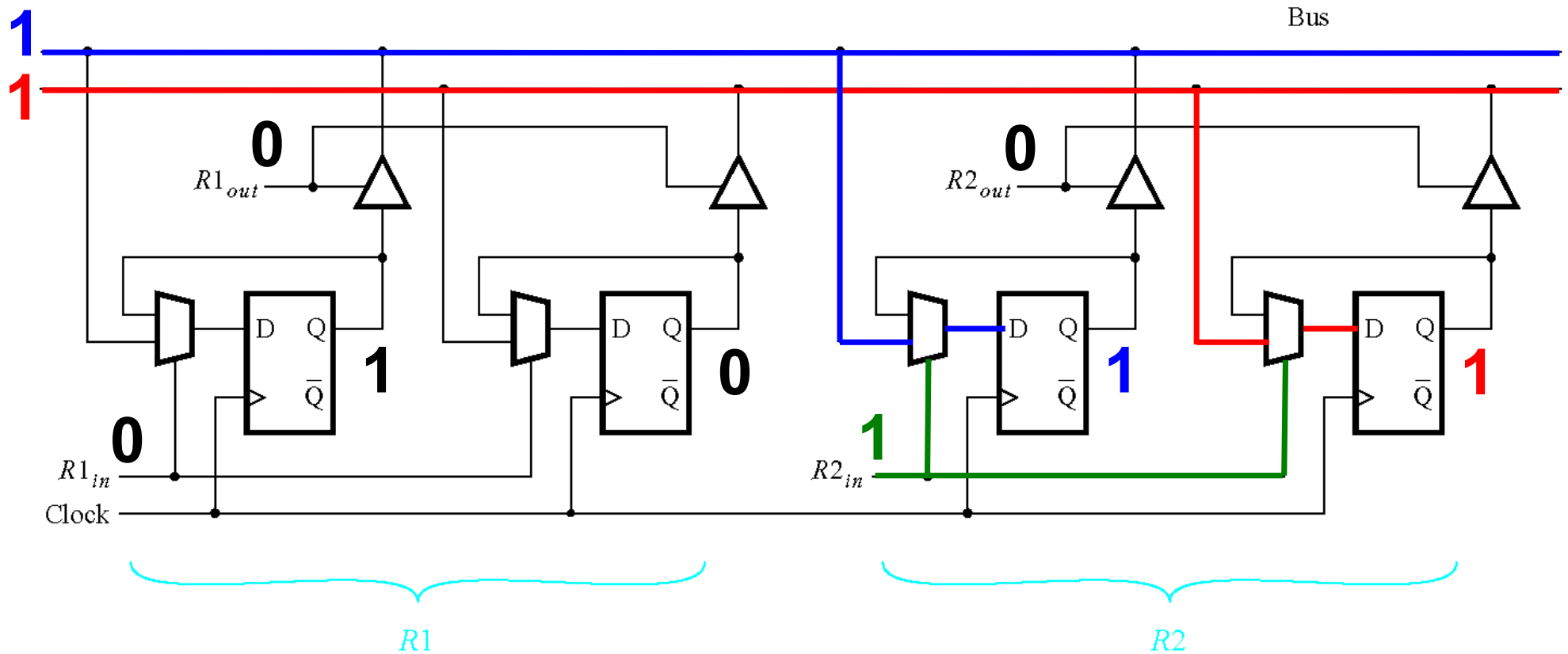
... and on the next positive clock edge to the outputs of the flip-flops of R2.

Loading Data From The Bus Into R2
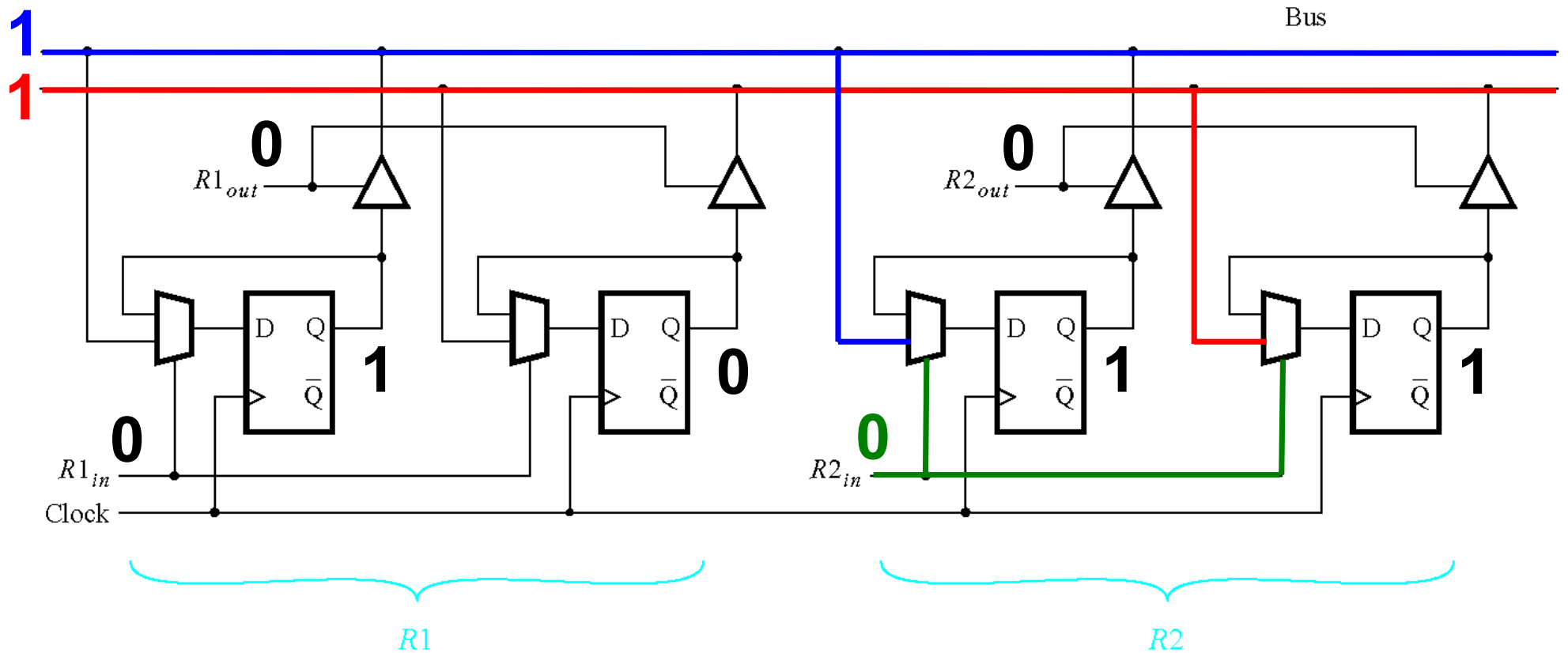
After the loading is complete $R2_{in}$ is set to 0.

Loading Data From The Bus Into R2
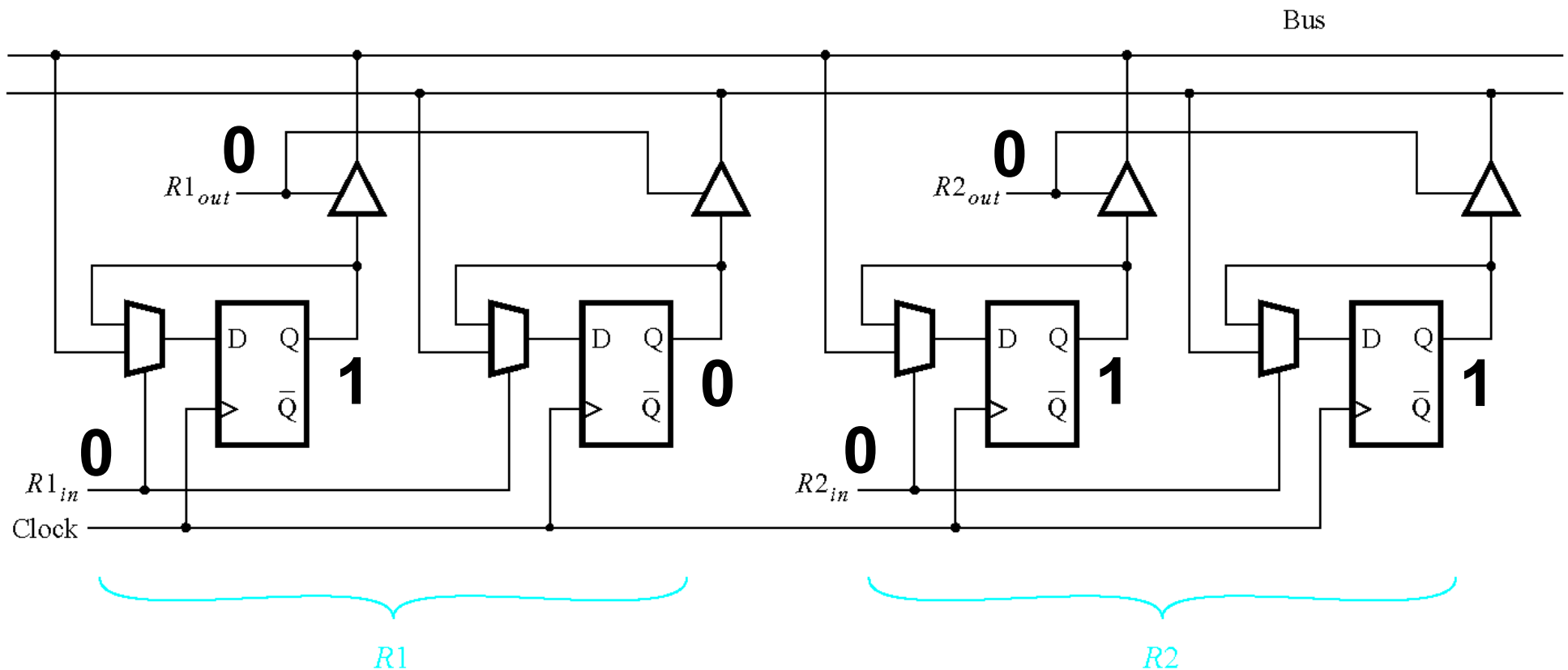
Register 2 now stores the number $3_{10}=11_2$.

A Closer Look at the Arithmetic Logic Unit (ALU)

Arithmetic Logic Unit (ALU)

[Figure 7.9 from the textbook]

Two Registers
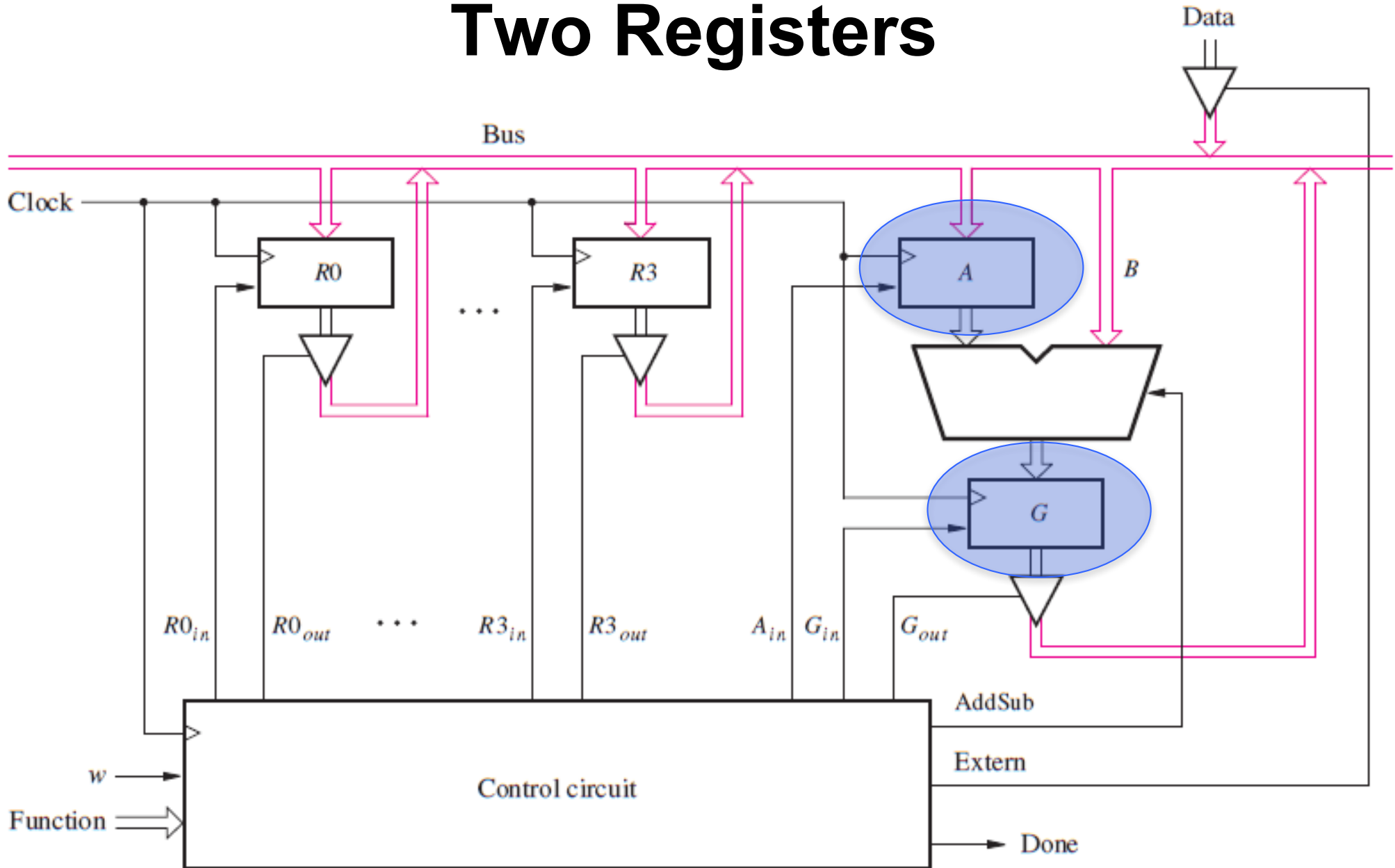
[Figure 7.9 from the textbook]

4-Bit Register
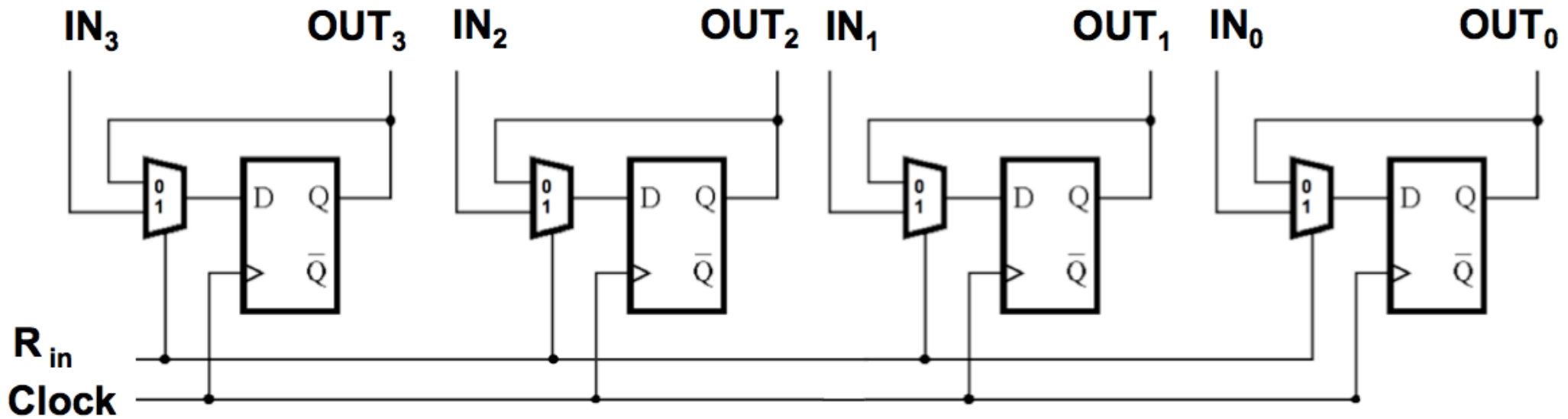
Adder/Subtractor
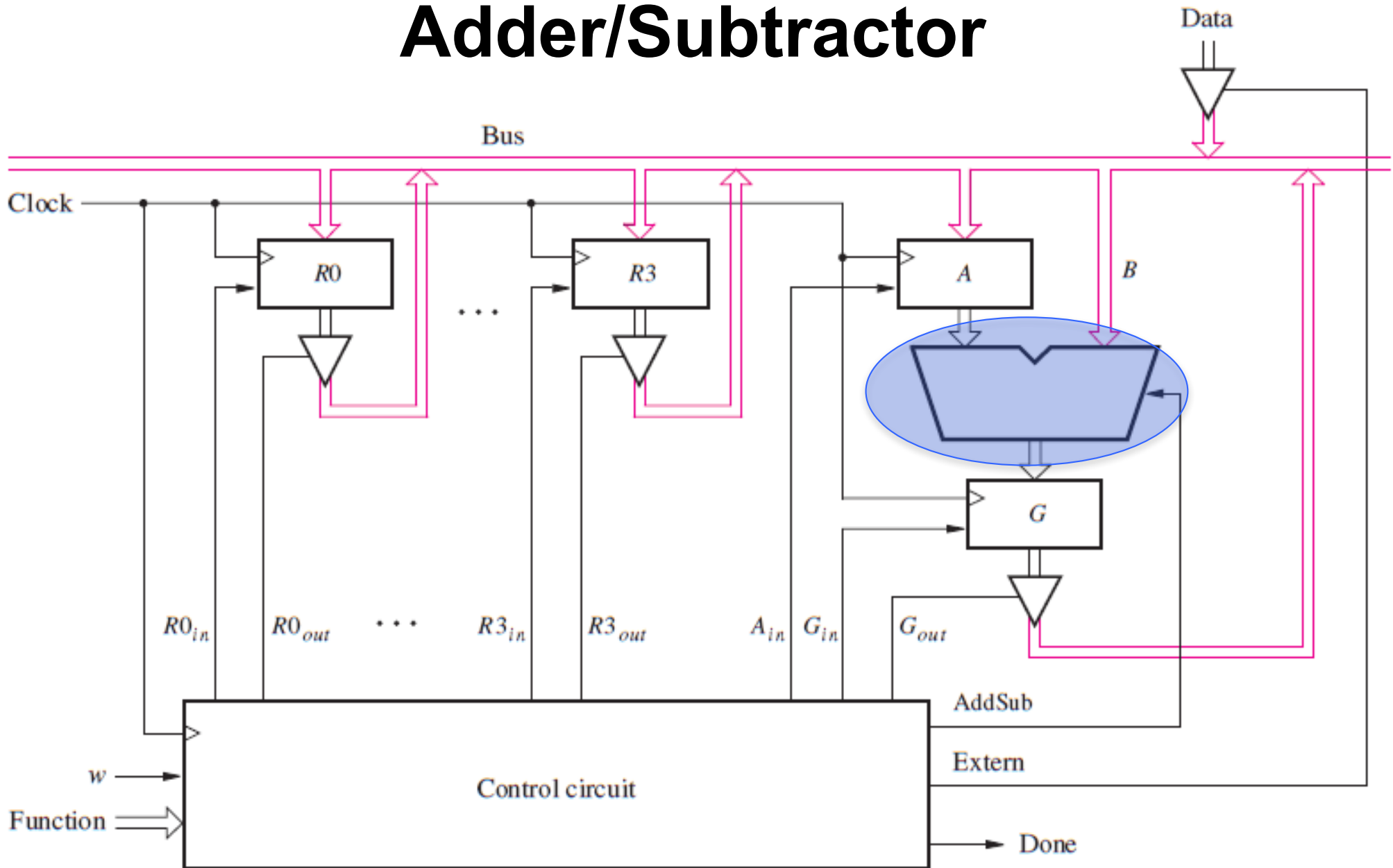
[Figure 7.9 from the textbook]

Adder/Subtractor unit
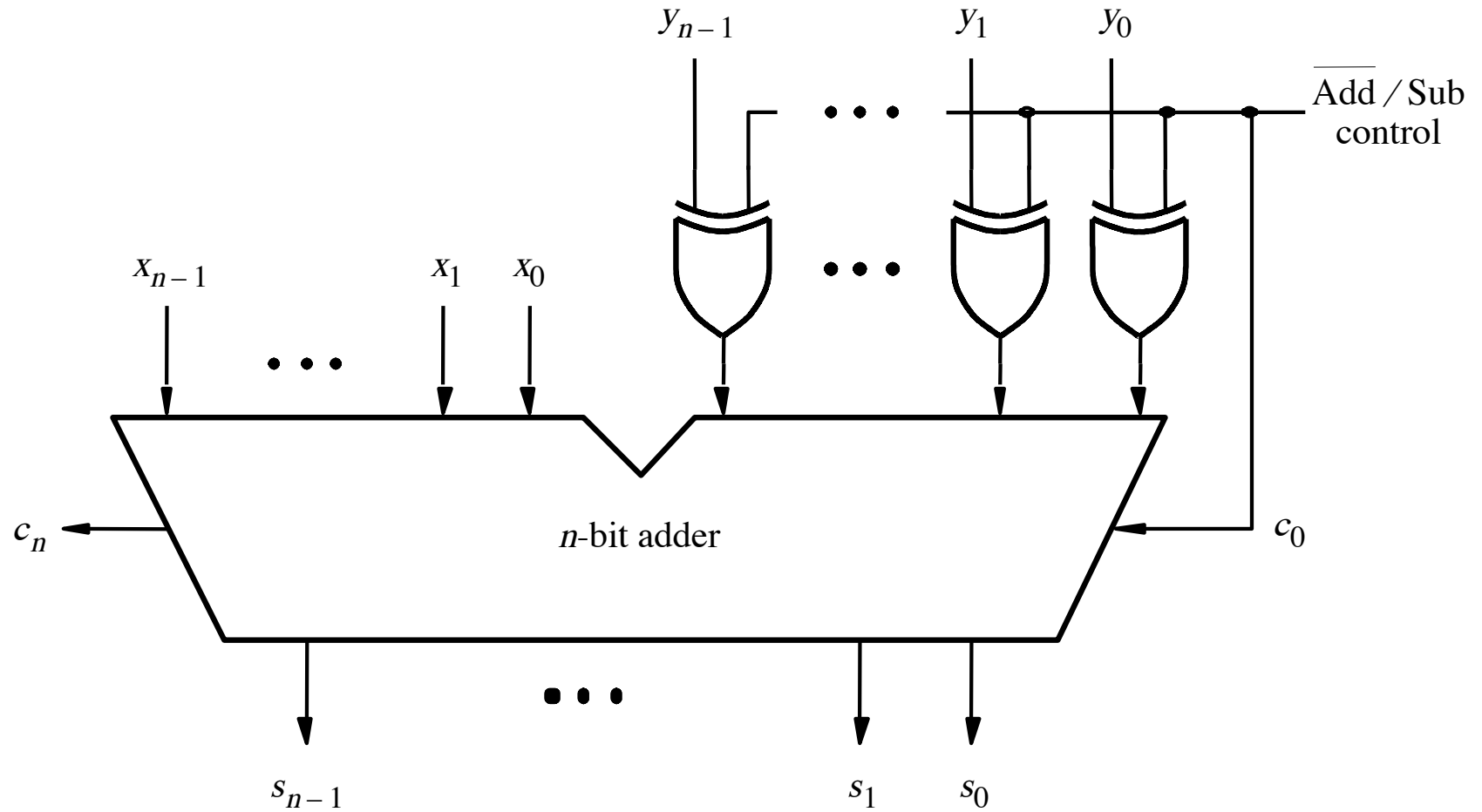
[Figure 3.12 from the textbook]

The first two stages of a carry-lookahead adder
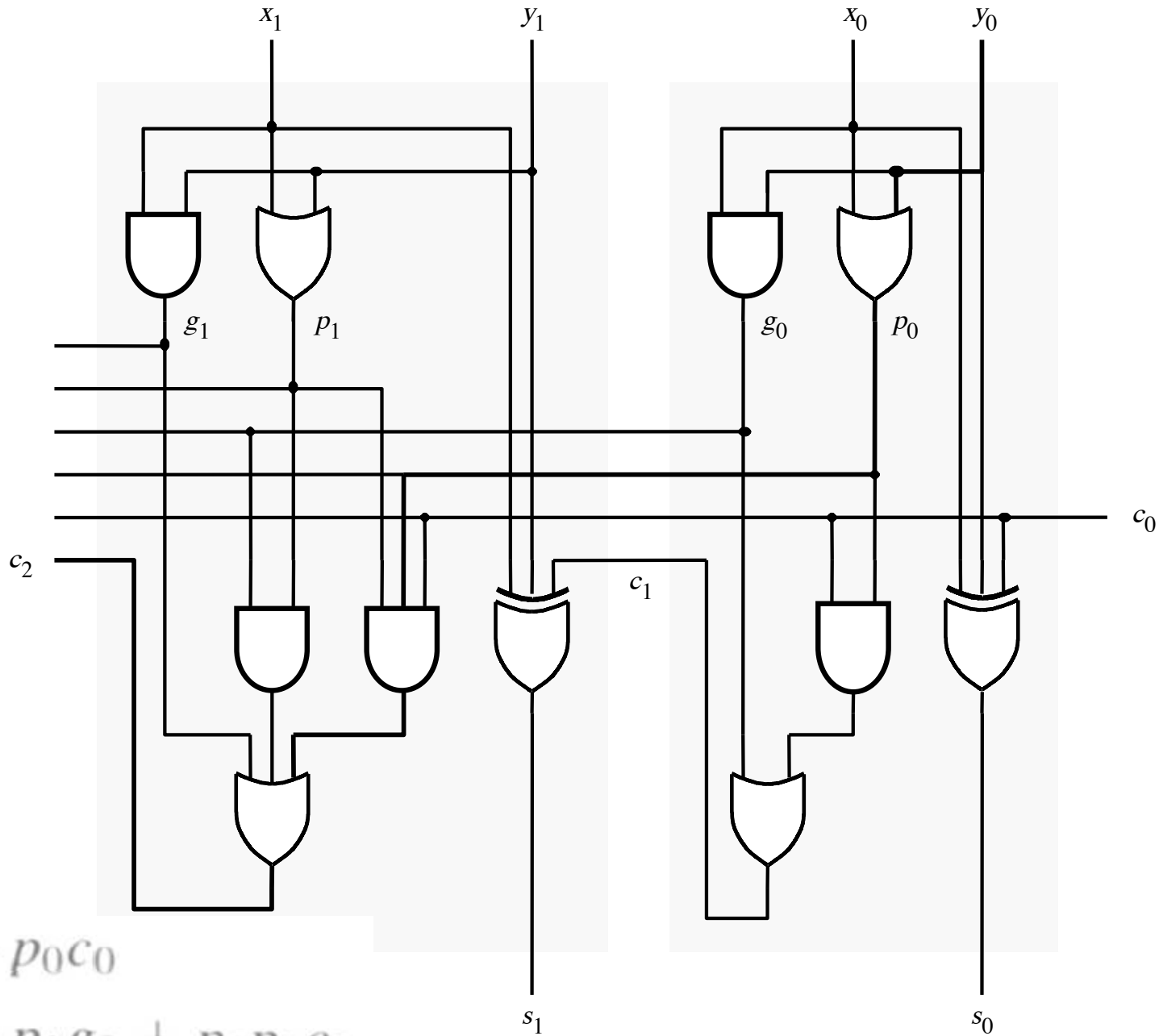
$$c_1 = g_0 + p_0 c_0$$

$$c_2 = g_1 + p_1 g_0 + p_1 p_0 c_0$$

[Figure 3.15 from the textbook]

A hierarchical carry-lookahead adder
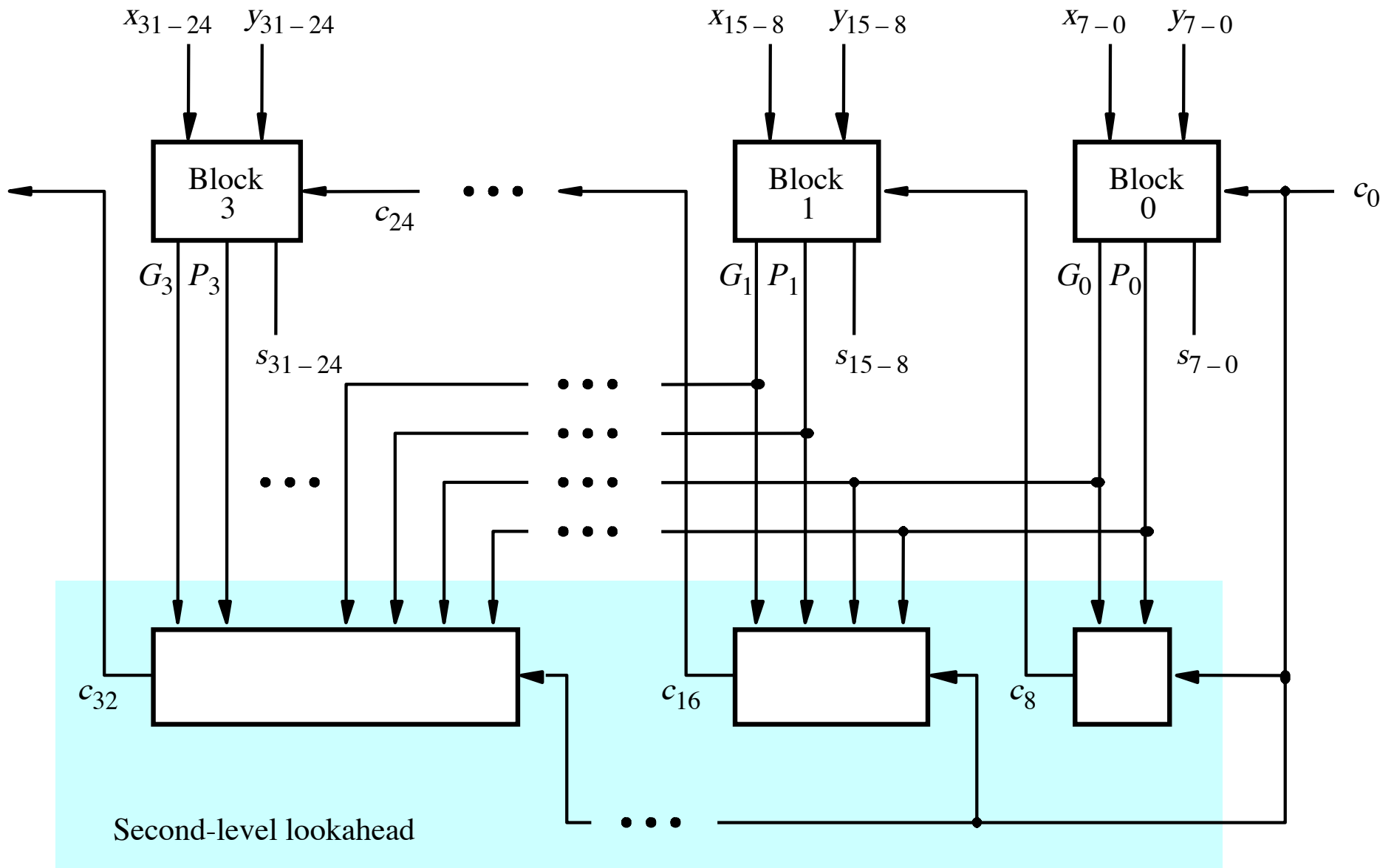
[Figure 3.17 from the textbook]

Adder/subtractor unit

- Subtraction can be performed by simply adding the 2's complement of the second number, regardless of the signs of the two numbers.

- Thus, the same adder circuit can be used to perform both addition and subtraction !!!

Adder/subtractor unit
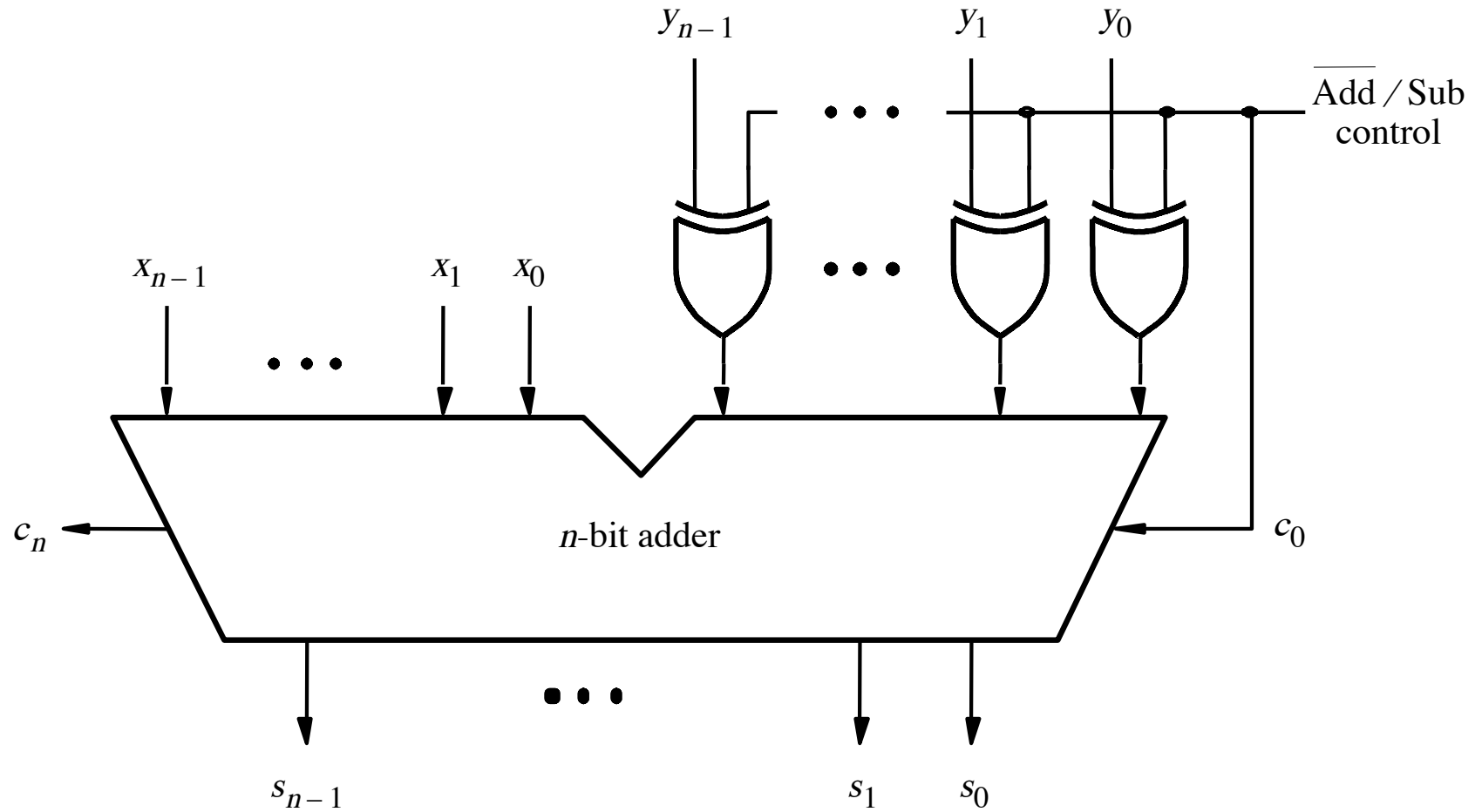
[Figure 3.12 from the textbook]

XOR Tricks

control	y	out
0	0	0
0	1	1
1	0	1
1	1	0

control

y

out
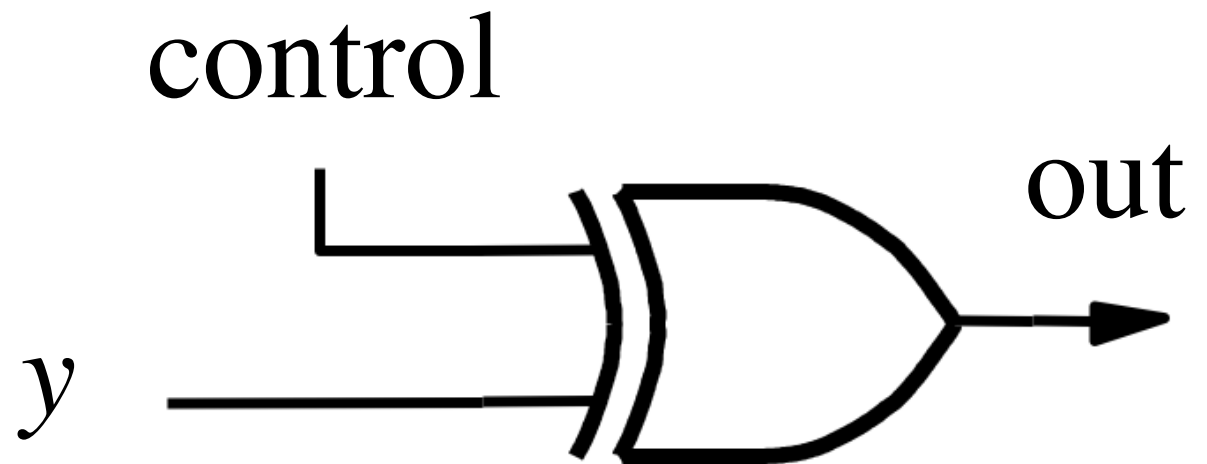
XOR as a repeater

control	y	out
0	0	0
0	1	1
1	0	1
1	1	0
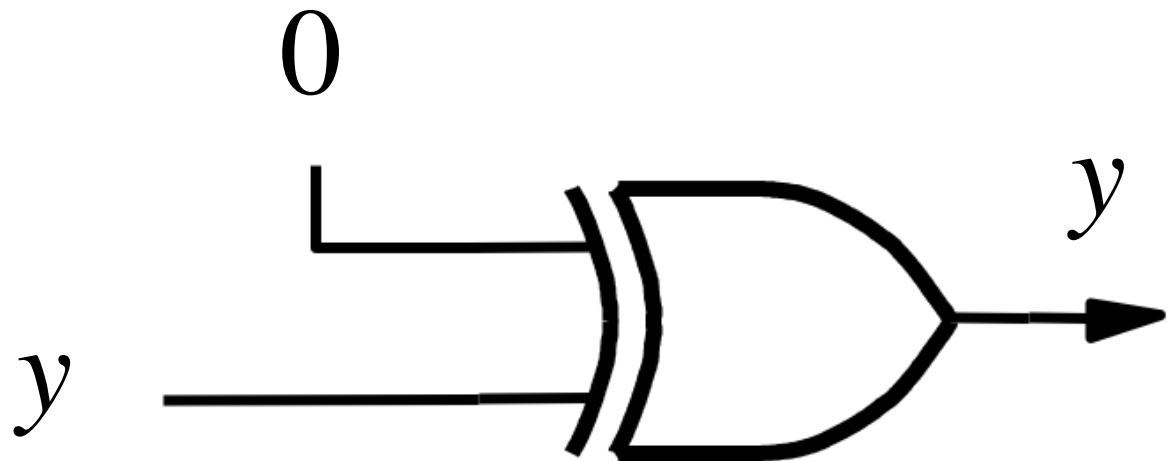
XOR as an inverter

control	y	out
0	0	0
0	1	1
1	0	1
1	1	0
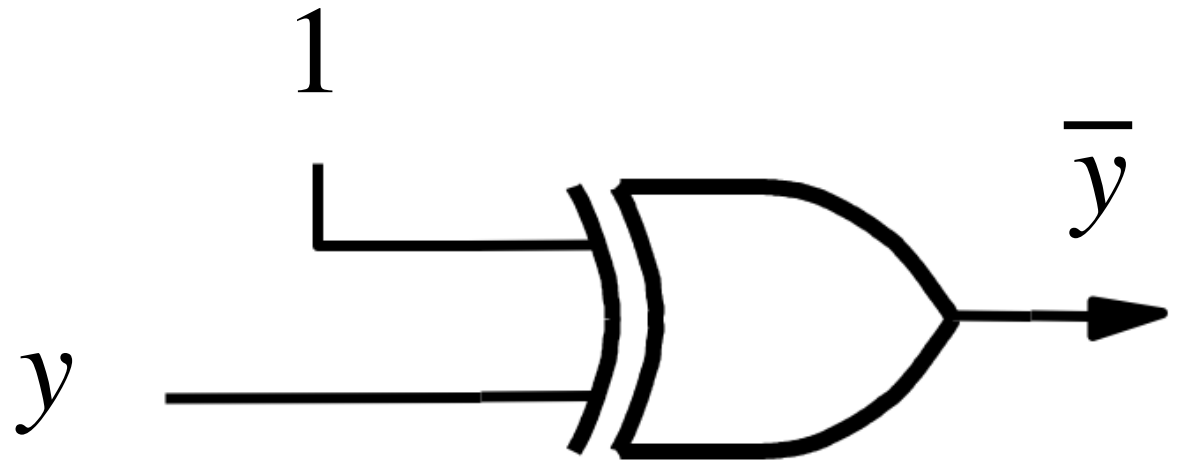
Addition: when control = 0
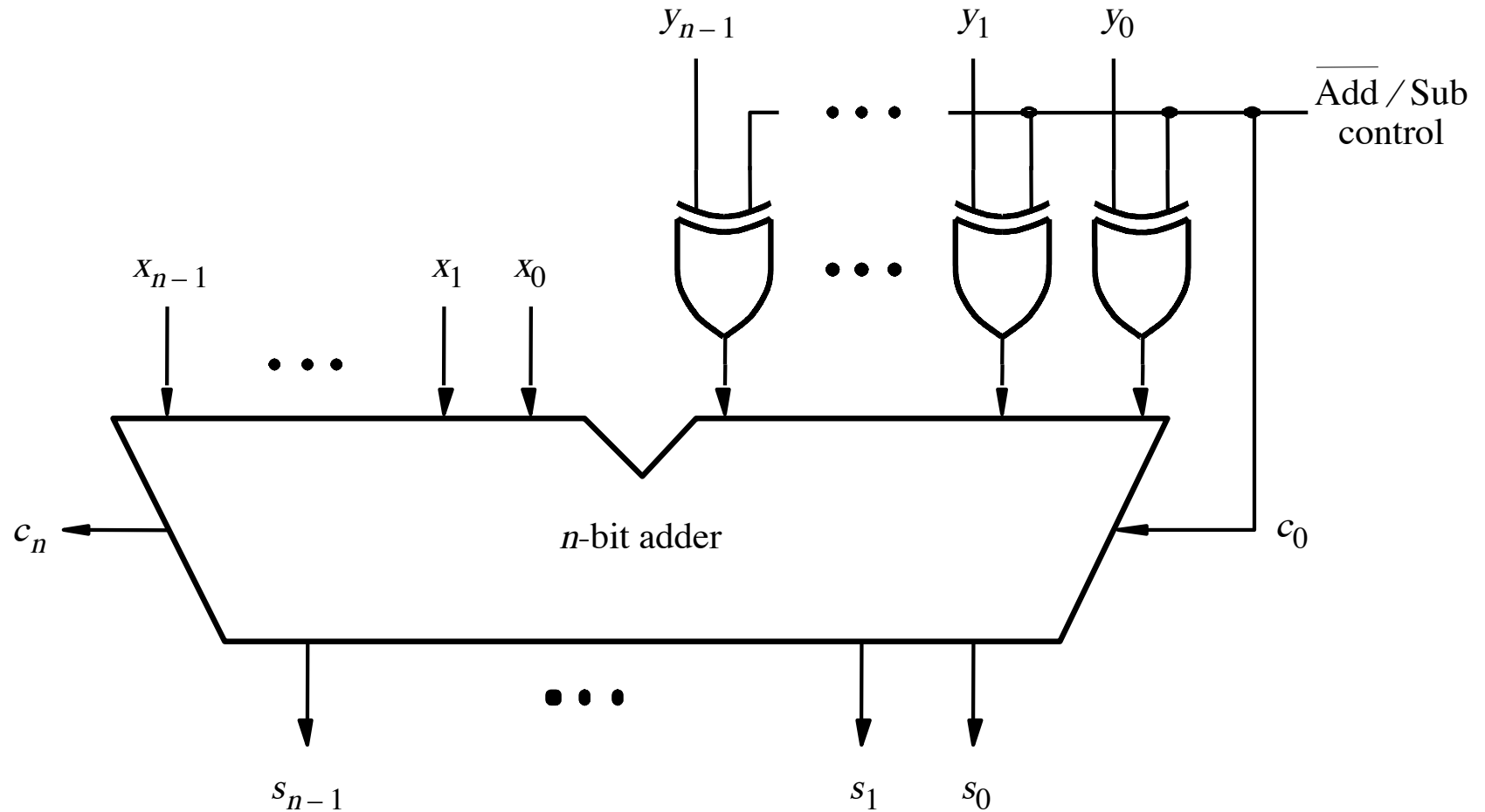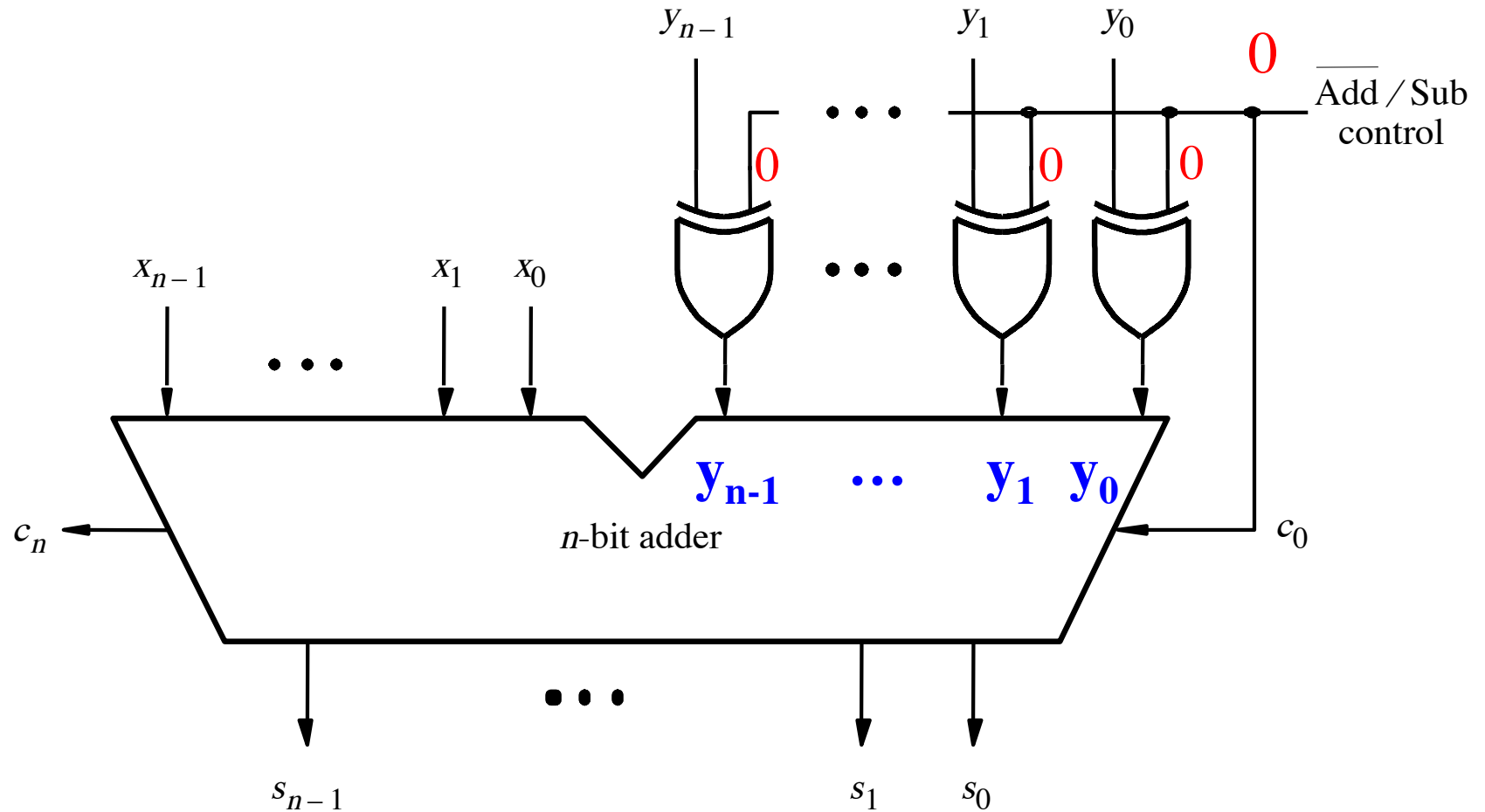
[Figure 3.12 from the textbook]

Addition: when control = 0

[Figure 3.12 from the textbook]

Addition: when control = 0

[Figure 3.12 from the textbook]

Subtraction: when control = 1
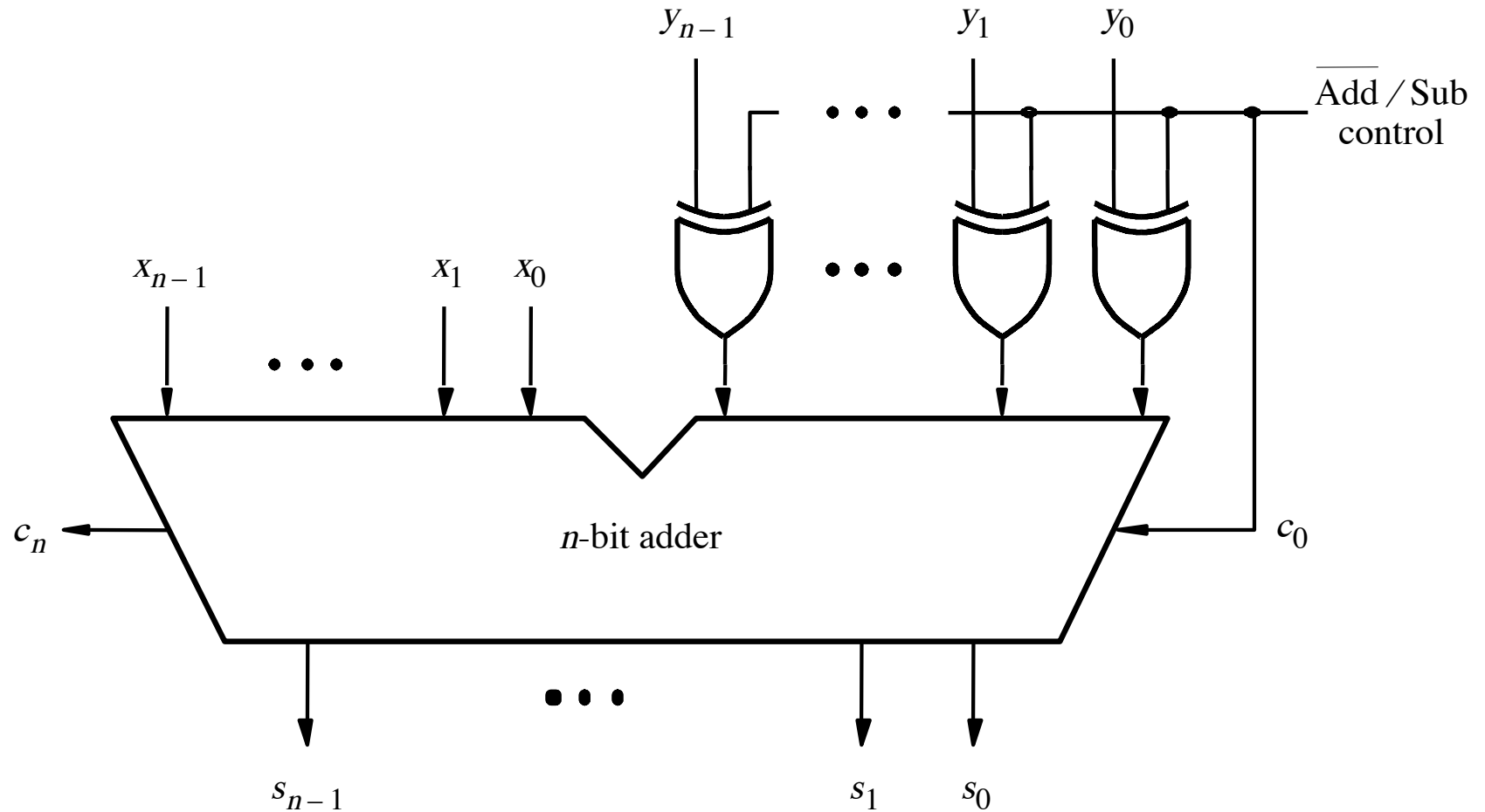
[Figure 3.12 from the textbook]

Subtraction: when control = 1
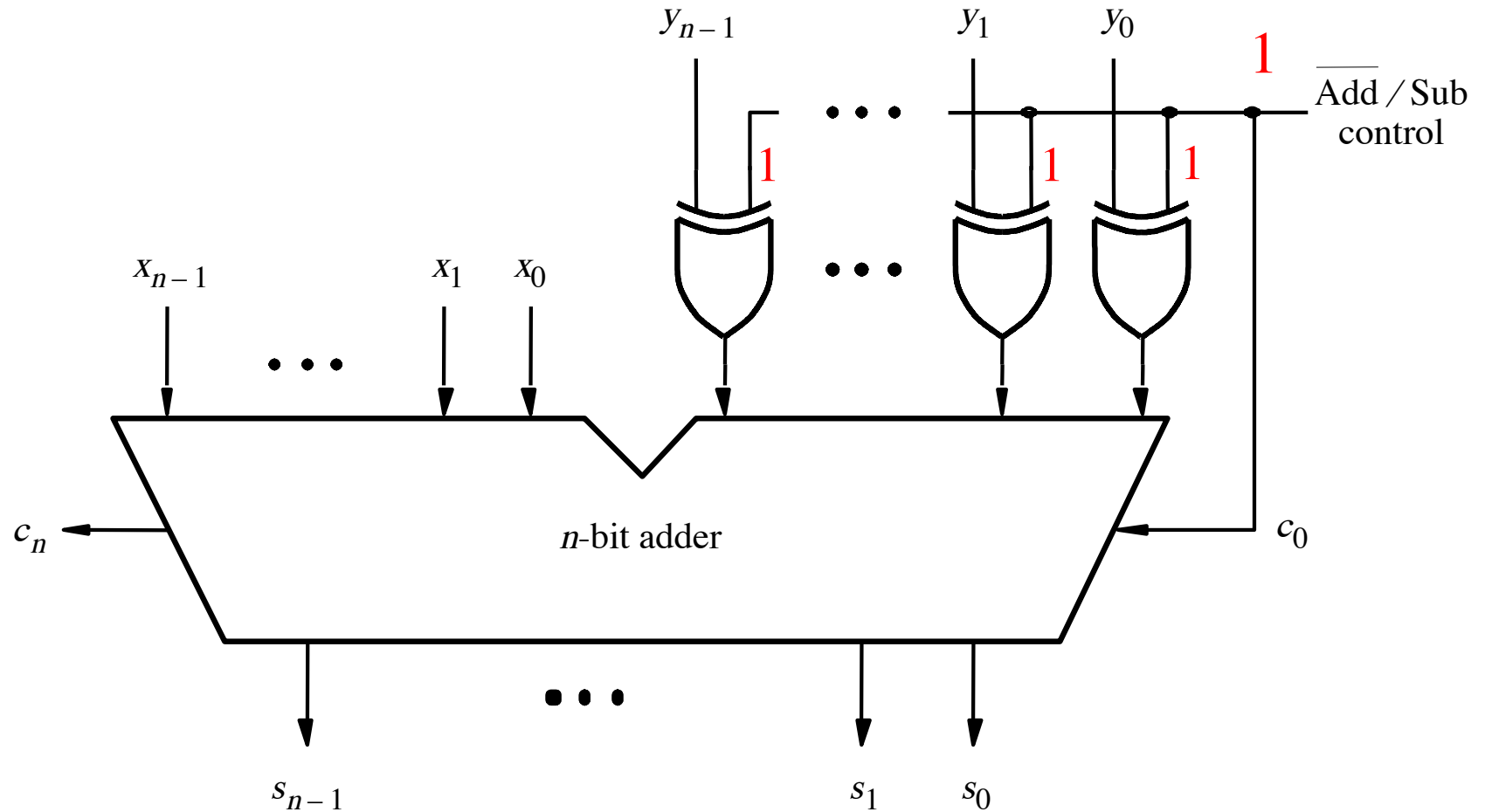
[Figure 3.12 from the textbook]

Subtraction: when control = 1
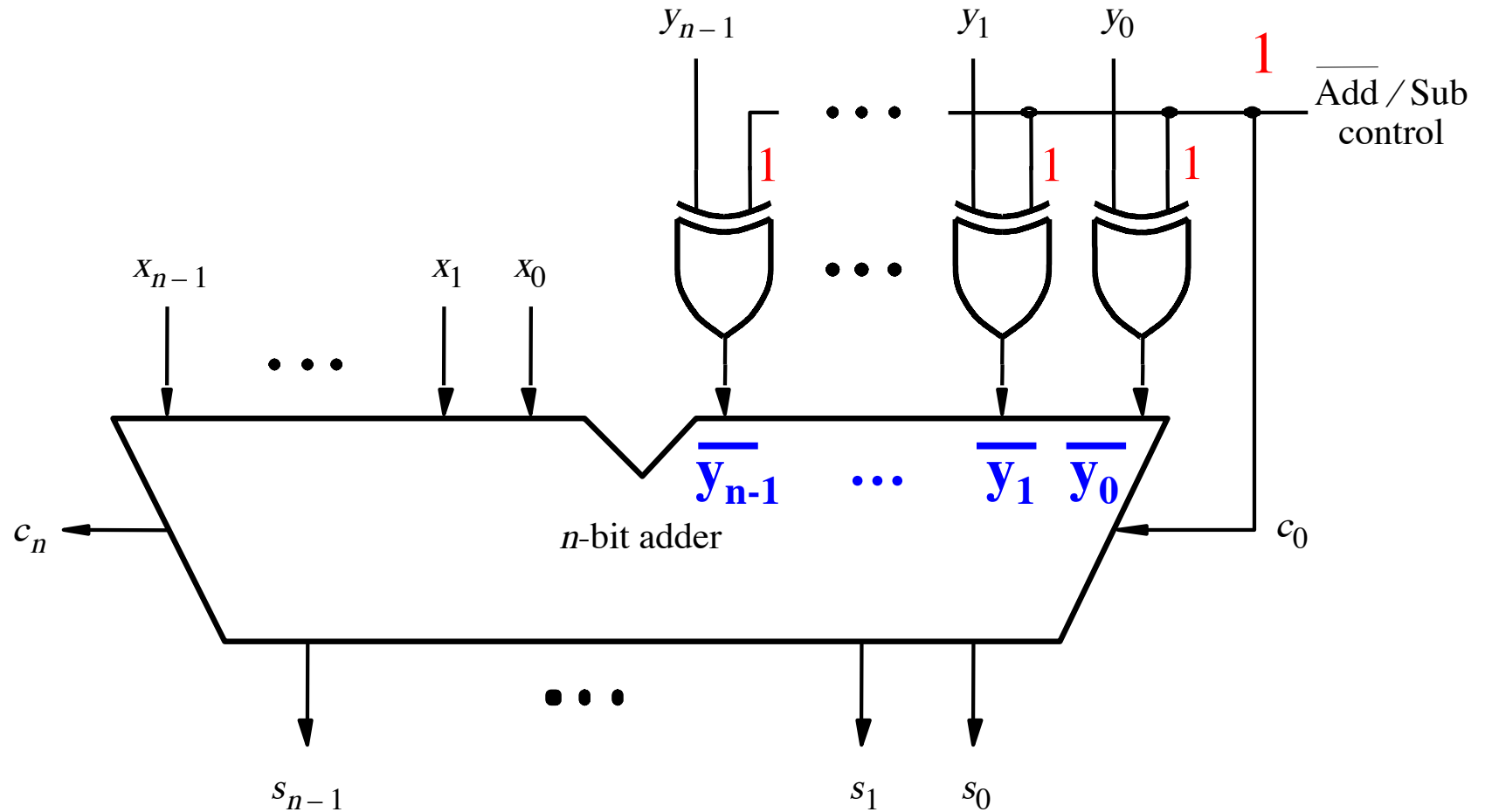
[Figure 3.12 from the textbook]

Subtraction: when control = 1
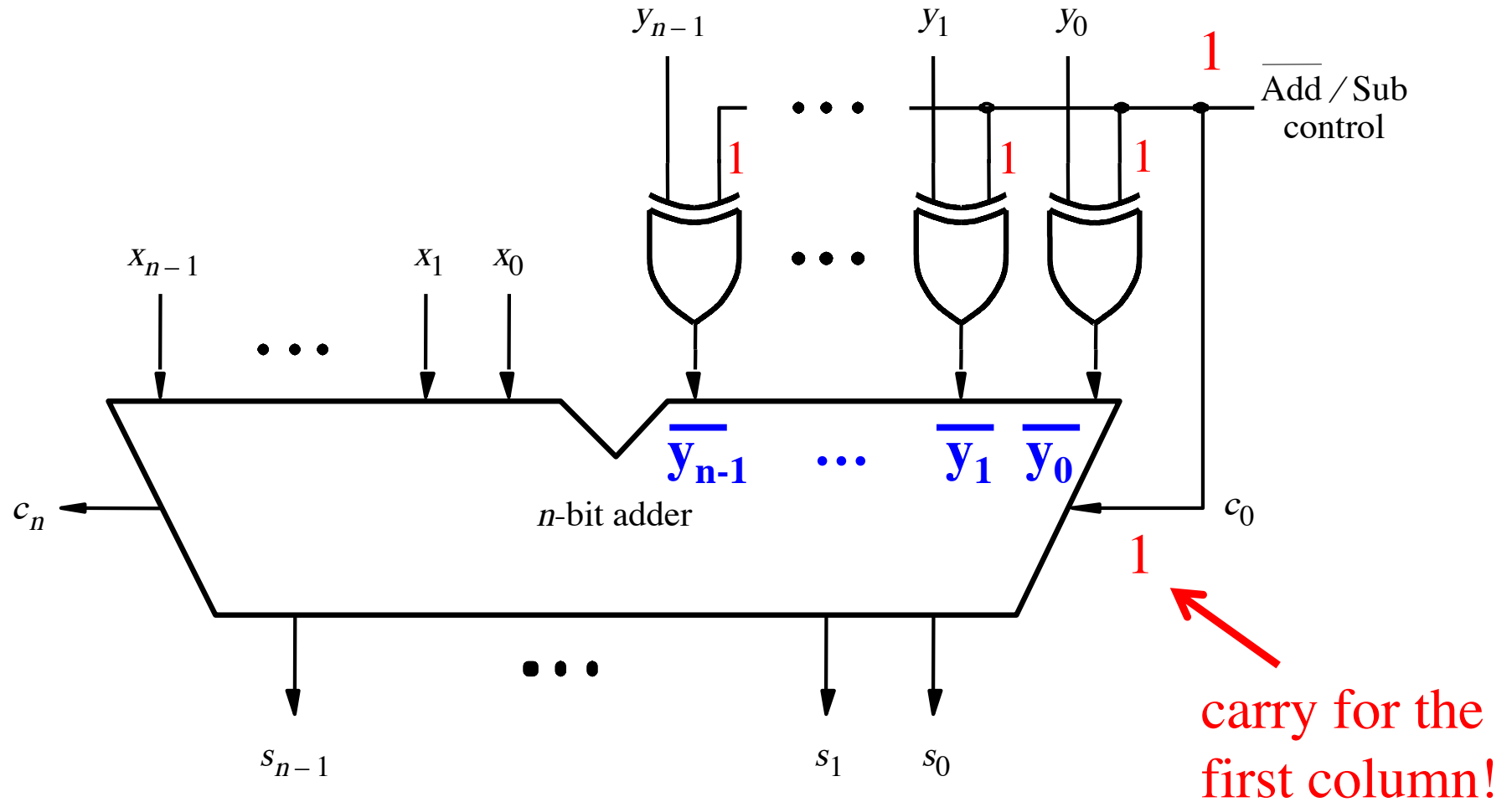
[Figure 3.12 from the textbook]

A Closer Look at the Control Circuit

Control Circuit
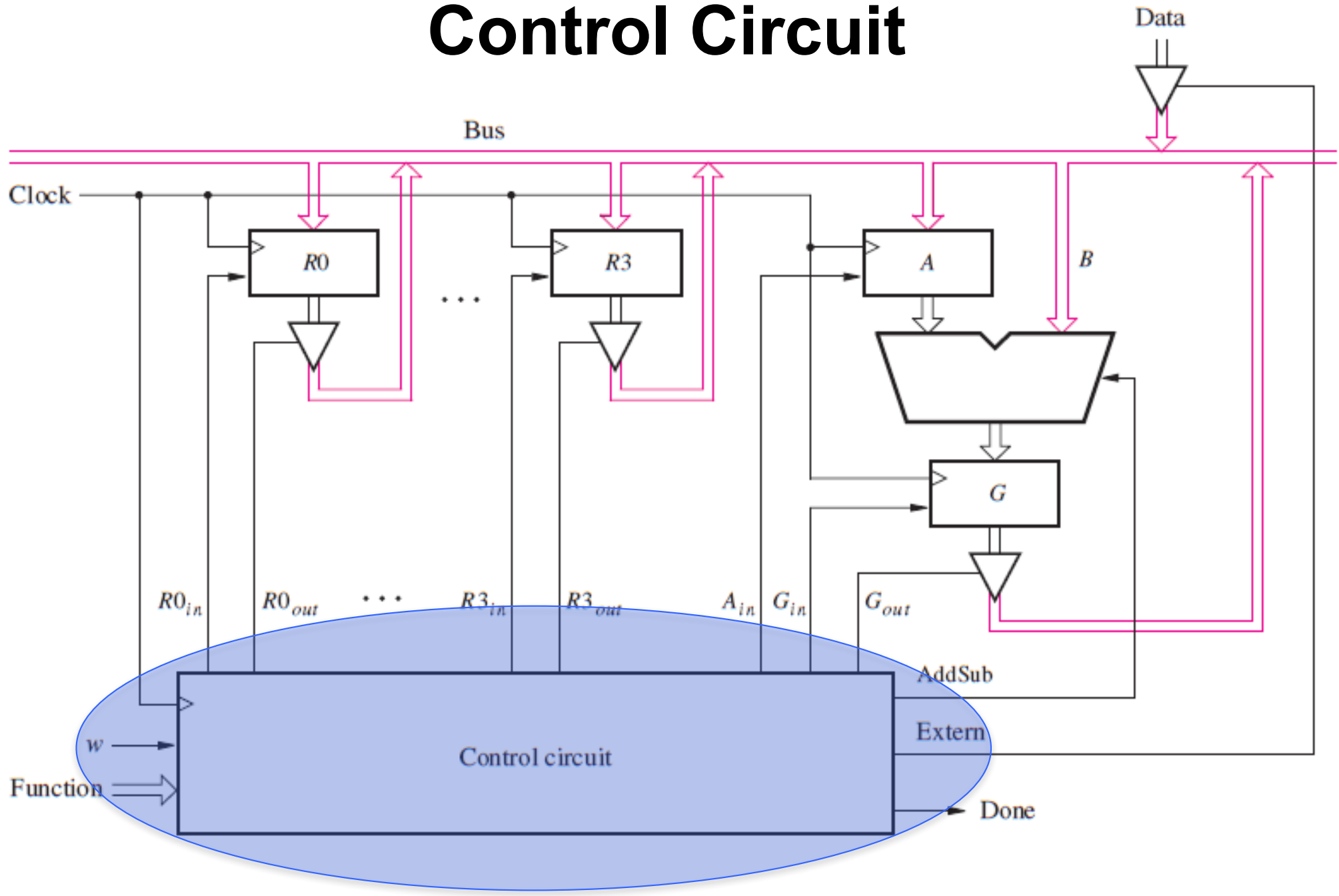
[Figure 7.9 from the textbook]

Control Signals
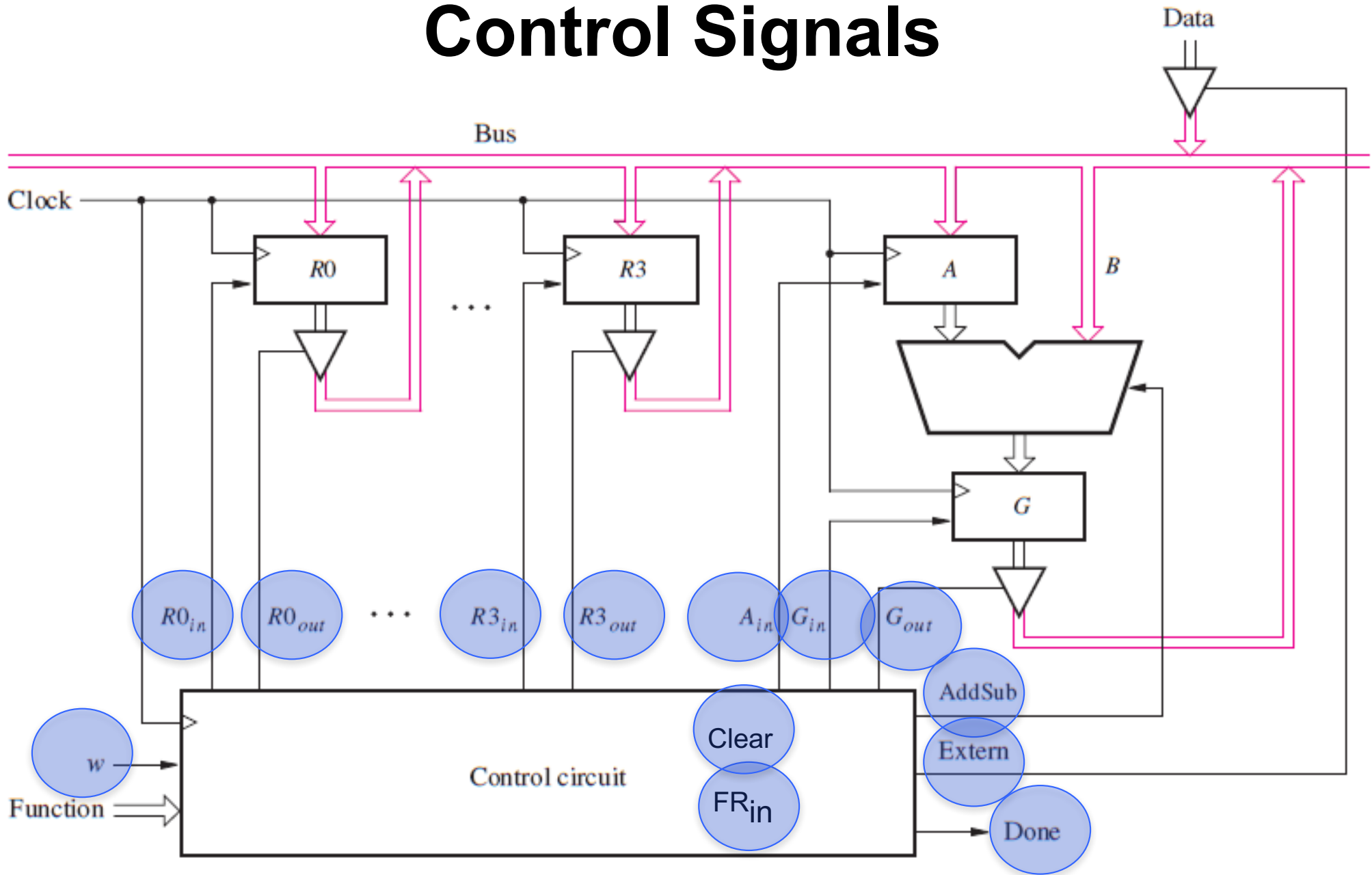
[Figure 7.9 from the textbook]

Design a FSM with input w and outputs

- $R0_{in}$
- $R0_{out}$

- $R1_{in}$
- $R1_{out}$

- $R2_{in}$
- $R2_{out}$

- $R3_{in}$
- $R3_{out}$

- A_{in}

- Gin
- G_{out}

- Clear

- FR_{in}

- AddSub

- Extern

- Done

Design a FSM with input w and outputs

- $R0_{in}$
- $R0_{out}$

- $R1_{in}$
- $R1_{out}$

- $R2_{in}$
- $R2_{out}$

- $R3_{in}$
- $R3_{out}$

- A_{in}

- Gin
- G_{out}

- Clear

- FR_{in}

- AddSub

- Extern

- Done

- T_0
- T_1
- T_2
- T_3

- I_0
- I_1
- I_2
- I_3

- X_0
- X_1
- X_2
- X_3

- Y_0
- Y_1
- Y_2
- Y_3

These are helper outputs that are one-hot encoded. They are used to simplify the expressions for the other outputs.

The function register and decoders
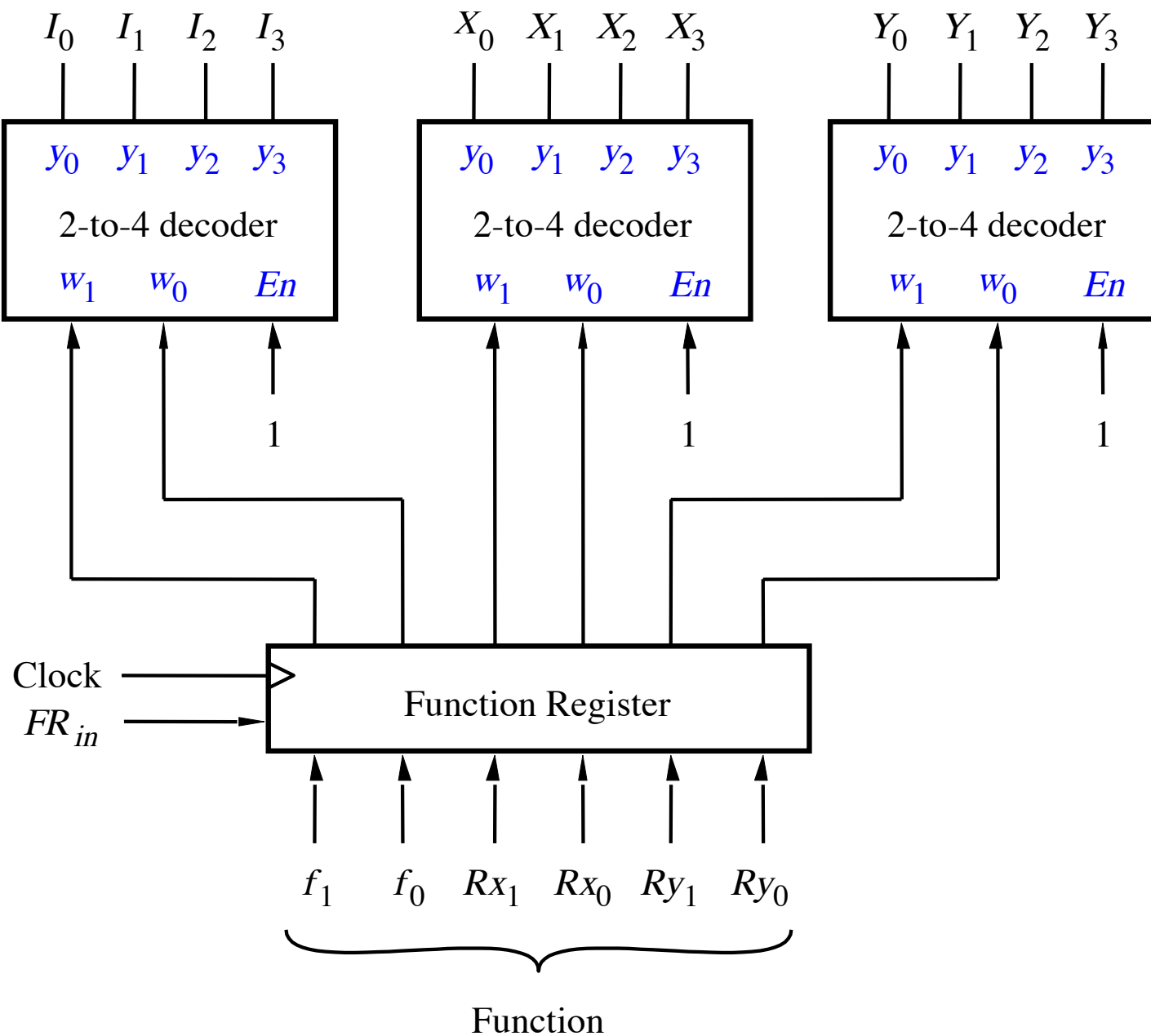
[Figure 7.11 from the textbook]

The function register and decoders

[Figure 7.11 from the textbook]

Operations performed by this processor

Operation	Function Performed
Load Rx, Data	Rx ← Data
Move Rx, Ry	Rx ← [Ry]
Add Rx, Ry	Rx ← [Rx] + [Ry]
Sub Rx, Ry	Rx ← [Rx] - [Ry]

[Table 7.1 from the textbook]

Operations performed by this processor

Operation	Function Performed
Load Rx, Data	Rx ← Data
Move Rx, Ry	Rx ← [Ry]
Add Rx, Ry	Rx ← [Rx] + [Ry]
Sub Rx, Ry	Rx ← [Rx] - [Ry]

Where Rx and Ry can be one of four possible options: R0, R1, R2, and R3

[Table 7.1 from the textbook]

Operations performed by this processor

f_1	f_0	Function
0	0	Load
0	1	Move
1	0	Add
1	1	Sub

Rx_1	Rx_0	Register
0	0	R0
0	1	R1
1	0	R2
1	1	R3

Ry_1	Ry_0	Register
0	0	R0
0	1	R1
1	0	R2
1	1	R3

Operations performed by this processor

f_1	f_0	Function
0	0	Load
0	1	Move
1	0	Add
1	1	Sub

Rx_1	Rx_0	Register
0	0	R0
0	1	R1
1	0	R2
1	1	R3

Ry_1	Ry_0	Register
0	0	R0
0	1	R1
1	0	R2
1	1	R3

The function register and decoders
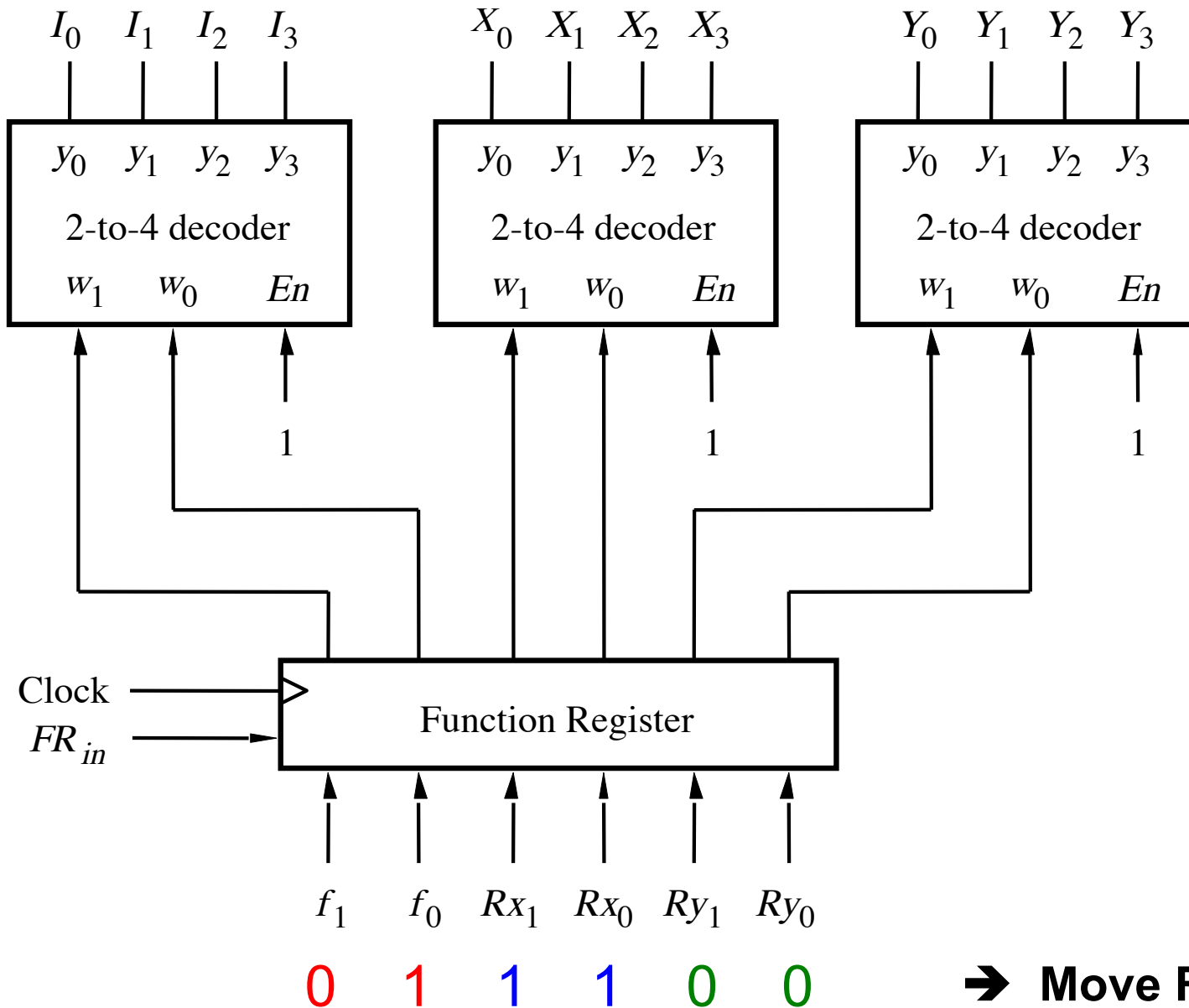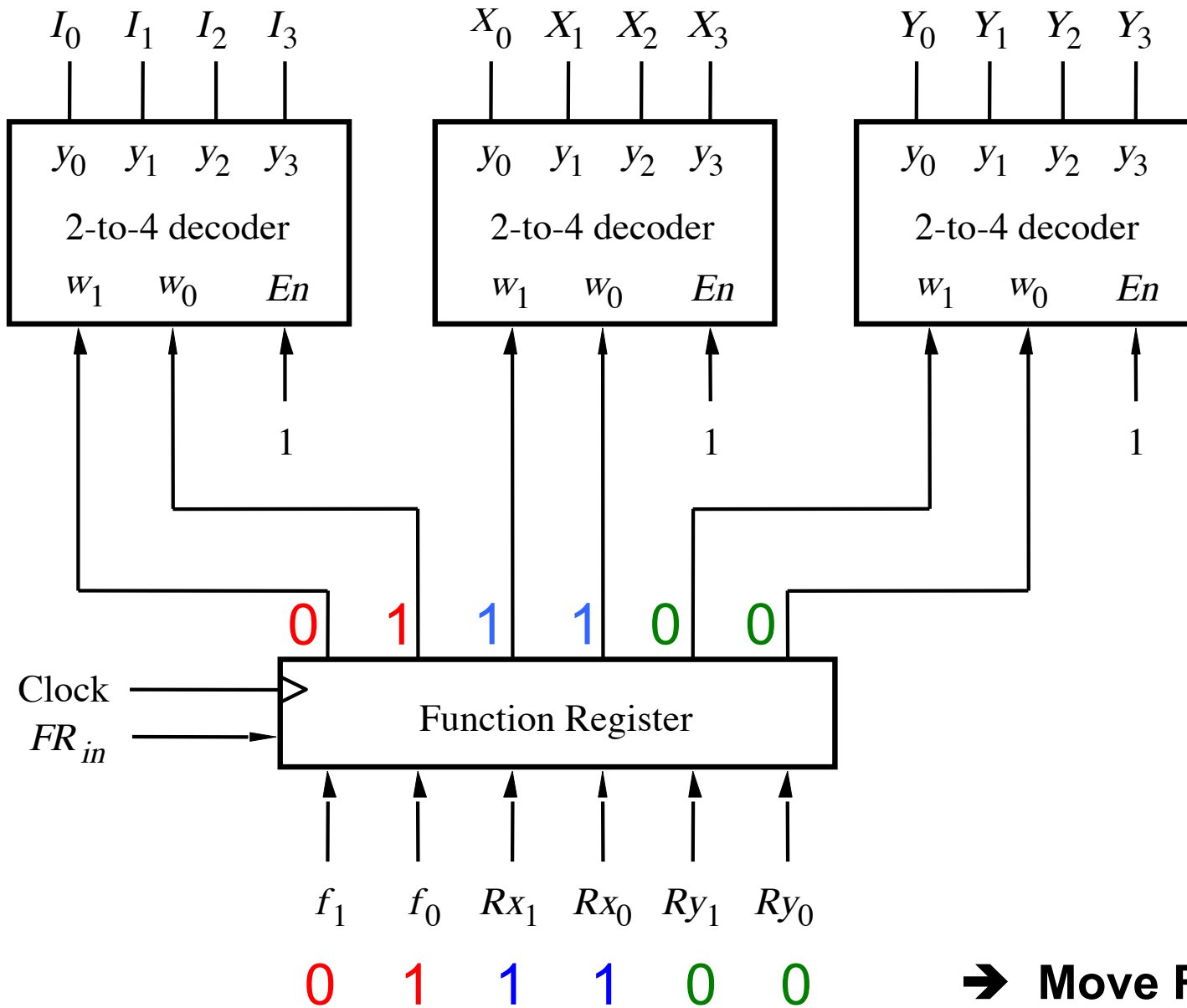
The function register and decoders

The function register and decoders
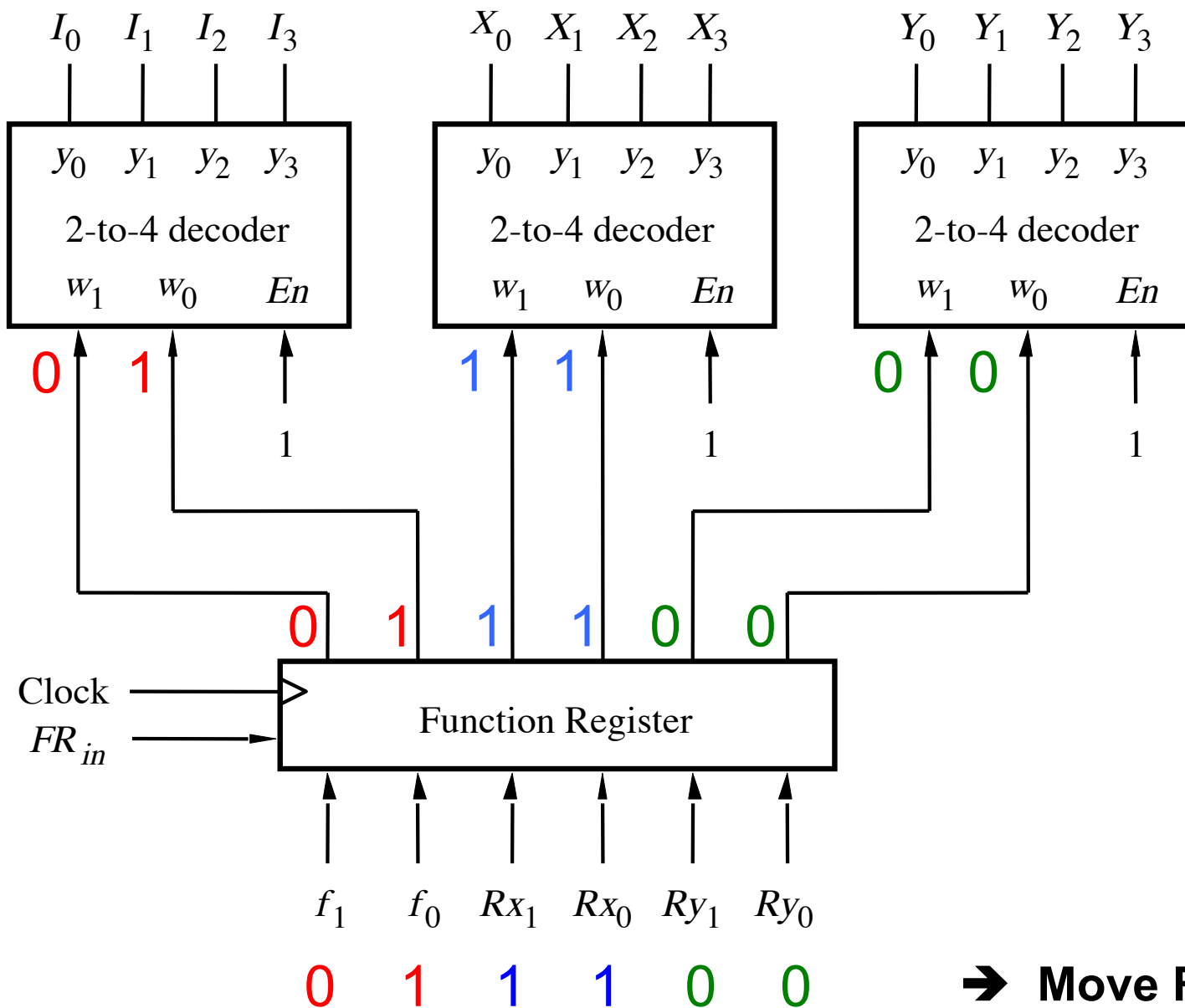
The function register and decoders
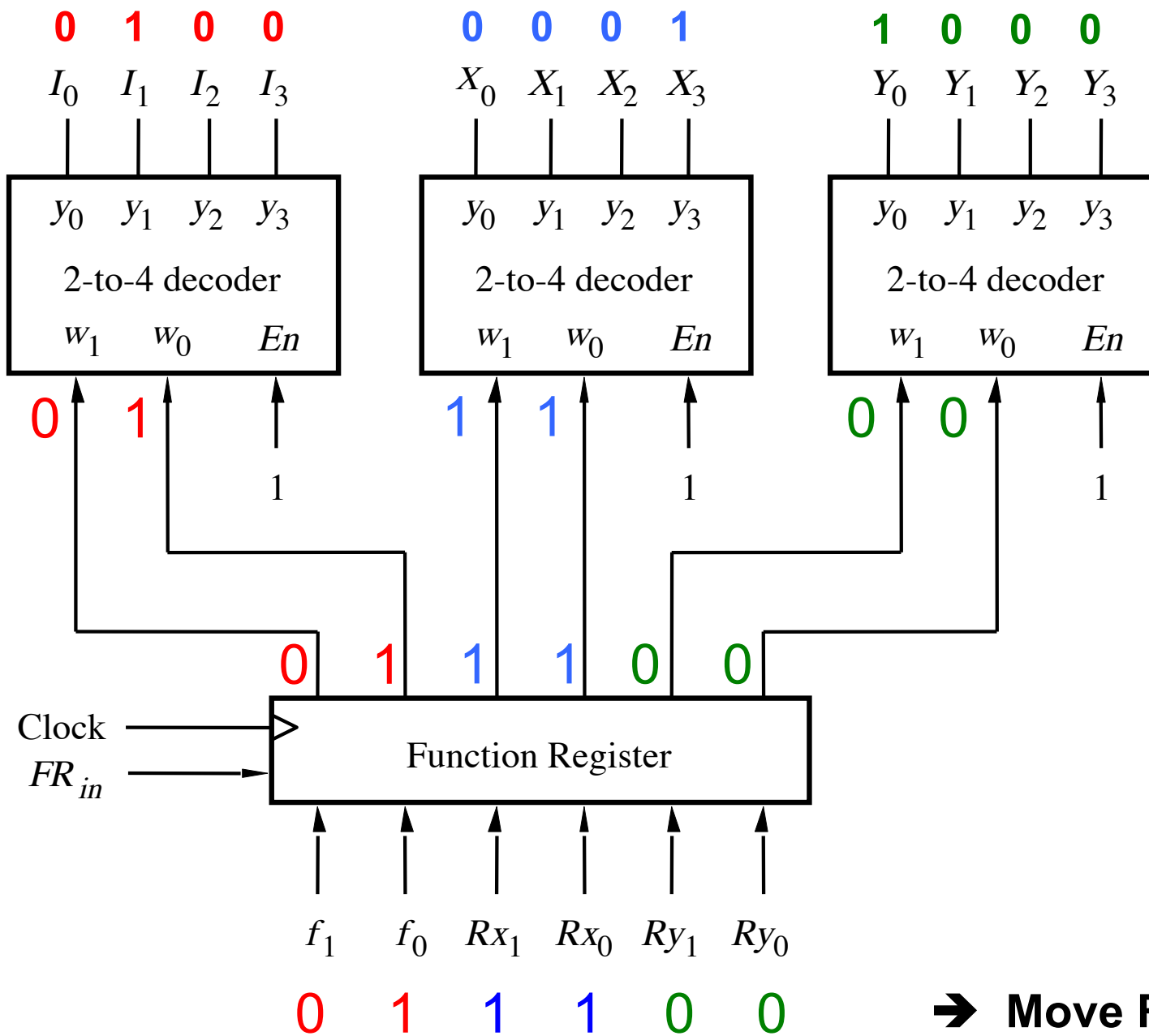
The function register and decoders

Operations performed by this processor
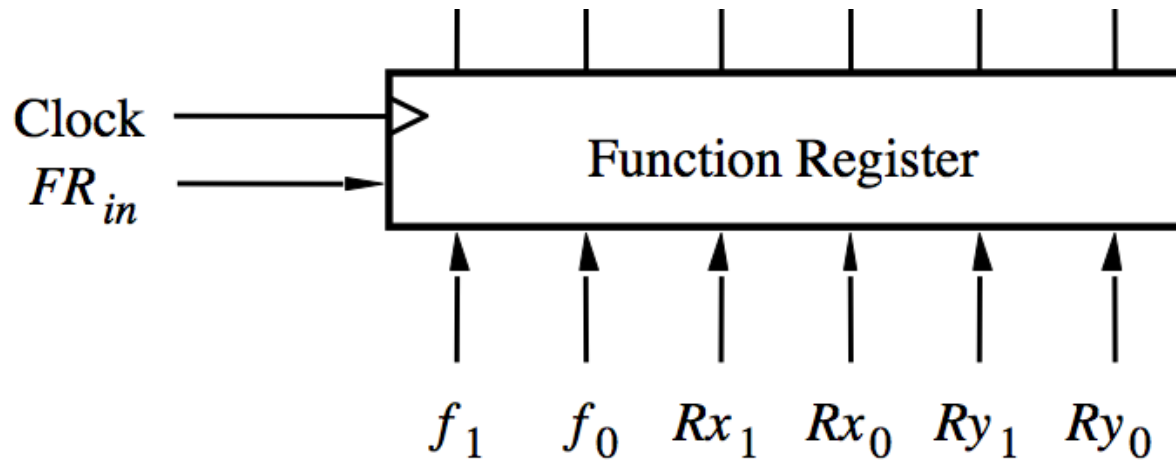
f_1	f_0	Function
0	0	Load
0	1	Move
1	0	Add
1	1	Sub

Rx_1	Rx_0	Register
0	0	R0
0	1	R1
1	0	R2
1	1	R3

Ry_1	Ry_0	Register
0	0	R0
0	1	R1
1	0	R2
1	1	R3
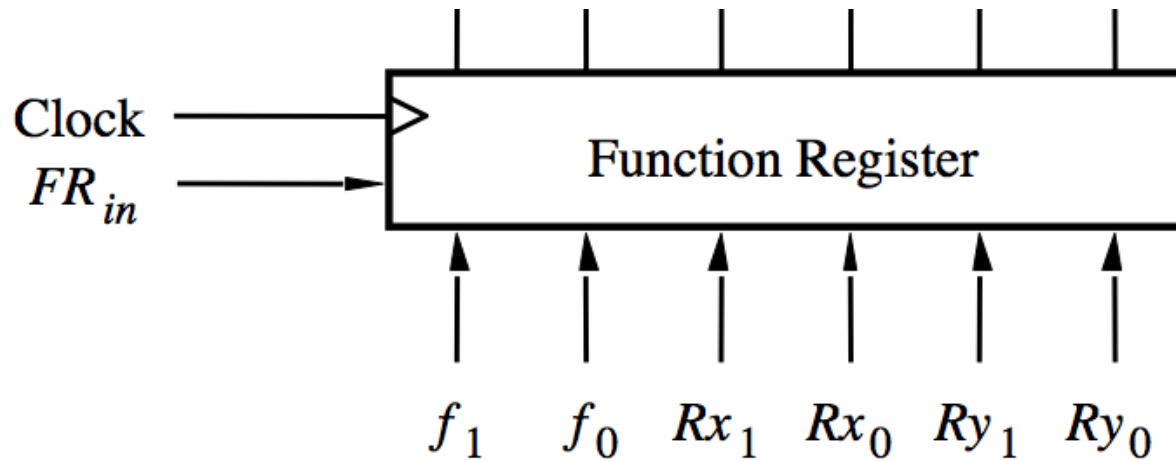
Operations performed by this processor

f_1	f_0	Function
0	0	Load
0	1	Move
1	0	Add
1	1	Sub

Rx_1	Rx_0	Register
0	0	R0
0	1	R1
1	0	R2
1	1	R3

Ry_1	Ry_0	Register
0	0	R0
0	1	R1
1	0	R2
1	1	R3

Operations performed by this processor

f_1	f_0	Function
0	0	Load
0	1	Move
1	0	Add
1	1	Sub

Rx_1	Rx_0	Register
0	0	R0
0	1	R1
1	0	R2
1	1	R3

Ry_1	Ry_0	Register
0	0	R0
0	1	R1
1	0	R2
1	1	R3

Similar Encoding is Used by Modern Chips

MIPS32 Add Immediate Instruction

001000	00001	00010	00000000101011110
OP Code	Addr 1	Addr 2	Immediate value
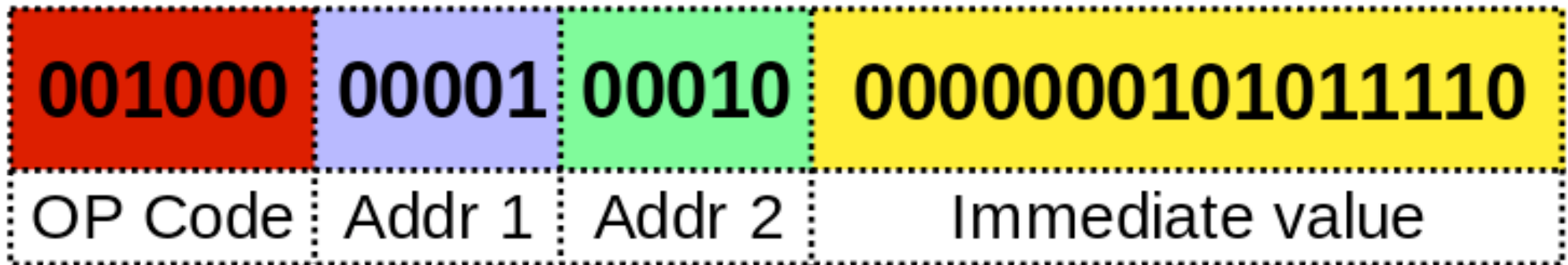
Equivalent mnemonic: addi $r1 , $r2 , 350

Sample Assembly Language Program
For This Processor

```
Move   R3, R0
Add    R1, R3
Sub    R0, R2
Load   R2, Data
```

Machine Language vs Assembly Language

Machine Language	Assembly Language	Meaning / Interpretation
011100	Move R3, R0	R3 ← [R0]
100111	Add R1, R3	R1 ← [R1] + [R3]
110010	Sub R0, R2	R0 ← [R0] − [R2]
001000	Load R2, Data	R2 ← Data

Machine Language vs Assembly Language

Machine Language	Assembly Language	Meaning / Interpretation
011100	Move R3, R0	R3 ← [R0]
100111	Add R1, R3	R1 ← [R1] + [R3]
110010	Sub R0, R2	R0 ← [R0] – [R2]
001000	Load R2, Data	R2 ← Data

Machine Language vs Assembly Language

Machine Language	Assembly Language	Meaning / Interpretation
011100	Move R3, R0	R3 ← [R0]
100111	Add R1, R3	R1 ← [R1] + [R3]
110010	Sub R0, R2	R0 ← [R0] − [R2]
001000	Load R2, Data	R2 ← Data

For short, each line can be expressed as a hexadecimal number

Machine Language vs Assembly Language

Machine Language	Assembly Language	Meaning / Interpretation
1C	Move R3, R0	R3 ← [R0]
27	Add R1, R3	R1 ← [R1] + [R3]
32	Sub R0, R2	R0 ← [R0] − [R2]
08	Load R2, Data	R2 ← Data

Intel 8086

```asm
; _memcpy(dst, src, len)
; Copy a block of memory from one location to another.
;
; Entry stack parameters
;        [BP+6] = len, Number of bytes to copy
;        [BP+4] = src, Address of source data block
;        [BP+2] = dst, Address of target data block
;
; Return registers
;        AX = Zero

0000:1000                            org      1000h        ; Start at 0000:1000h

0000:1000              _memcpy       proc
0000:1000 55                         push     bp           ; Set up the call frame
0000:1001 89 E5                      mov      bp,sp
0000:1003 06                         push     es           ; Save ES
0000:1004 8B 4E 06                   mov      cx,[bp+6]    ; Set CX = len
0000:1007 E3 11                      jcxz     done         ; If len=0, return
0000:1009 8B 76 04                   mov      si,[bp+4]    ; Set SI = src
0000:100C 8B 7E 02                   mov      di,[bp+2]    ; Set DI = dst
0000:100F 1E                         push     ds           ; Set ES = DS
0000:1010 07                         pop      es

0000:1011 8A 04      loop           mov      al,[si]      ; Load AL from [src]
0000:1013 88 05                      mov      [di],al      ; Store AL to [dst]
0000:1015 46                         inc      si           ; Increment src
0000:1016 47                         inc      di           ; Increment dst
0000:1017 49                         dec      cx           ; Decrement len
0000:1018 75 F7                      jnz      loop         ; Repeat the loop

0000:101A 07         done           pop      es           ; Restore ES
0000:101B 5D                         pop      bp           ; Restore previous call frame
0000:101C 29 C0                      sub      ax,ax        ; Set AX = 0
0000:101E C3                         ret                   ; Return
0000:101F                            end proc
```

[http://en.wikipedia.org/wiki/Intel_8086]

Intel 8086

Memory Address

```
; _memcpy(dst, src, len)
; Copy a block of memory from one location to another.
;
; Entry stack parameters
;       [BP+6] = len, Number of bytes to copy
;       [BP+4] = src, Address of source data block
;       [BP+2] = dst, Address of target data block
;
; Return registers
;       AX = Zero

0000:1000                              org     1000h        ; Start at 0000:1000h

0000:1000             _memcpy          proc
0000:1000 55                           push    bp           ; Set up the call frame
0000:1001 89 E5                        mov     bp,sp
0000:1003 06                           push    es           ; Save ES
0000:1004 8B 4E 06                     mov     cx,[bp+6]    ; Set CX = len
0000:1007 E3 11                        jcxz    done         ; If len=0, return
0000:1009 8B 76 04                     mov     si,[bp+4]    ; Set SI = src
0000:100C 8B 7E 02                     mov     di,[bp+2]    ; Set DI = dst
0000:100F 1E                           push    ds           ; Set ES = DS
0000:1010 07                           pop     es

0000:1011 8A 04       loop             mov     al,[si]      ; Load AL from [src]
0000:1013 88 05                        mov     [di],al      ; Store AL to [dst]
0000:1015 46                           inc     si           ; Increment src
0000:1016 47                           inc     di           ; Increment dst
0000:1017 49                           dec     cx           ; Decrement len
0000:1018 75 F7                        jnz     loop         ; Repeat the loop

0000:101A 07          done             pop     es           ; Restore ES
0000:101B 5D                           pop     bp           ; Restore previous call frame
0000:101C 29 C0                        sub     ax,ax        ; Set AX = 0
0000:101E C3                           ret                  ; Return
0000:101F                              end proc
```

[http://en.wikipedia.org/wiki/Intel_8086]

Intel 8086

```asm
; _memcpy(dst, src, len)
; Copy a block of memory from one location to another.
;
; Entry stack parameters
;       [BP+6] = len, Number of bytes to copy
;       [BP+4] = src, Address of source data block
;       [BP+2] = dst, Address of target data block
;
; Return registers
;       AX = Zero
```

Machine Language

```asm
0000:1000                              org     1000h       ; Start at 0000:1000h

0000:1000              _memcpy         proc
0000:1000  55                          push    bp          ; Set up the call frame
0000:1001  89 E5                       mov     bp,sp
0000:1003  06                          push    es          ; Save ES
0000:1004  8B 4E 06                    mov     cx,[bp+6]   ; Set CX = len
0000:1007  E3 11                       jcxz    done        ; If len=0, return
0000:1009  8B 76 04                    mov     si,[bp+4]   ; Set SI = src
0000:100C  8B 7E 02                    mov     di,[bp+2]   ; Set DI = dst
0000:100F  1E                          push    ds          ; Set ES = DS
0000:1010  07                          pop     es

0000:1011  8A 04     loop              mov     al,[si]     ; Load AL from [src]
0000:1013  88 05                       mov     [di],al     ; Store AL to [dst]
0000:1015  46                          inc     si          ; Increment src
0000:1016  47                          inc     di          ; Increment dst
0000:1017  49                          dec     cx          ; Decrement len
0000:1018  75 F7                       jnz     loop        ; Repeat the loop

0000:101A  07        done              pop     es          ; Restore ES
0000:101B  5D                          pop     bp          ; Restore previous call frame
0000:101C  29 C0                       sub     ax,ax       ; Set AX = 0
0000:101E  C3                          ret                 ; Return
0000:101F                              end proc
```

[http://en.wikipedia.org/wiki/Intel_8086]

Intel 8086

```
; _memcpy(dst, src, len)
; Copy a block of memory from one location to another.
;
; Entry stack parameters
;        [BP+6] = len, Number of bytes to copy
;        [BP+4] = src, Address of source data block
;        [BP+2] = dst, Address of target data block
;
; Return registers
;        AX = Zero
```

Assembly Language

```
0000:1000                      org      1000h       ; Start at 0000:1000h

0000:1000          _memcpy      proc
0000:1000  55                   push     bp          ; Set up the call frame
0000:1001  89 E5                mov      bp,sp
0000:1003  06                   push     es          ; Save ES
0000:1004  8B 4E 06             mov      cx,[bp+6]   ; Set CX = len
0000:1007  E3 11                jcxz     done        ; If len=0, return
0000:1009  8B 76 04             mov      si,[bp+4]   ; Set SI = src
0000:100C  8B 7E 02             mov      di,[bp+2]   ; Set DI = dst
0000:100F  1E                   push     ds          ; Set ES = DS
0000:1010  07                   pop      es

0000:1011  8A 04    loop        mov      al,[si]     ; Load AL from [src]
0000:1013  88 05                mov      [di],al     ; Store AL to [dst]
0000:1015  46                   inc      si          ; Increment src
0000:1016  47                   inc      di          ; Increment dst
0000:1017  49                   dec      cx          ; Decrement len
0000:1018  75 F7                jnz      loop        ; Repeat the loop

0000:101A  07       done        pop      es          ; Restore ES
0000:101B  5D                   pop      bp          ; Restore previous call frame
0000:101C  29 C0                sub      ax,ax       ; Set AX = 0
0000:101E  C3                   ret                  ; Return
0000:101F                       end proc
```

[http://en.wikipedia.org/wiki/Intel_8086]

Intel 8086

```asm
; _memcpy(dst, src, len)
; Copy a block of memory from one location to another.
;
; Entry stack parameters
;       [BP+6] = len, Number of bytes to copy
;       [BP+4] = src, Address of source data block
;       [BP+2] = dst, Address of target data block
;
; Return registers
;       AX = Zero
```

Comments

```asm
0000:1000                           org     1000h       ; Start at 0000:1000h

0000:1000           _memcpy     proc
0000:1000 55                        push    bp          ; Set up the call frame
0000:1001 89 E5                     mov     bp,sp
0000:1003 06                        push    es          ; Save ES
0000:1004 8B 4E 06                  mov     cx,[bp+6]   ; Set CX = len
0000:1007 E3 11                     jcxz    done        ; If len=0, return
0000:1009 8B 76 04                  mov     si,[bp+4]   ; Set SI = src
0000:100C 8B 7E 02                  mov     di,[bp+2]   ; Set DI = dst
0000:100F 1E                        push    ds          ; Set ES = DS
0000:1010 07                        pop     es

0000:1011 8A 04     loop            mov     al,[si]     ; Load AL from [src]
0000:1013 88 05                     mov     [di],al     ; Store AL to [dst]
0000:1015 46                        inc     si          ; Increment src
0000:1016 47                        inc     di          ; Increment dst
0000:1017 49                        dec     cx          ; Decrement len
0000:1018 75 F7                     jnz     loop        ; Repeat the loop

0000:101A 07        done            pop     es          ; Restore ES
0000:101B 5D                        pop     bp          ; Restore previous call frame
0000:101C 29 C0                     sub     ax,ax       ; Set AX = 0
0000:101E C3                        ret                 ; Return
0000:101F                           end proc
```

[http://en.wikipedia.org/wiki/Intel_8086]

Another Part of The Control Circuit

A part of the control circuit for the processor

T_0 T_1 T_2 T_3

y_0 y_1 y_2 y_3

2-to-4 decoder

w_1 w_0 En

1

Clock

Clear

Q_1 Q_0

Up-counter

$Reset$

[Figure 7.10 from the textbook]

What are the components?

T_0 T_1 T_2 T_3

y_0 y_1 y_2 y_3

2-to-4 decoder

w_1 w_0 *En*

1

Clock

Q_1 Q_0

Up-counter

Clear

Reset
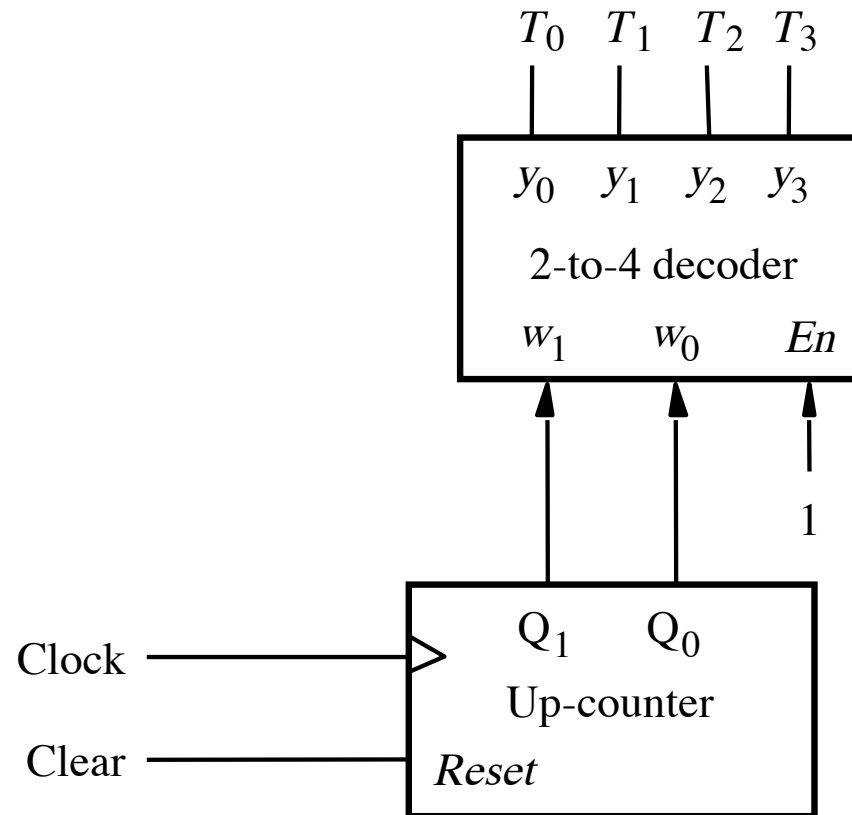
[Figure 7.10 from the textbook]
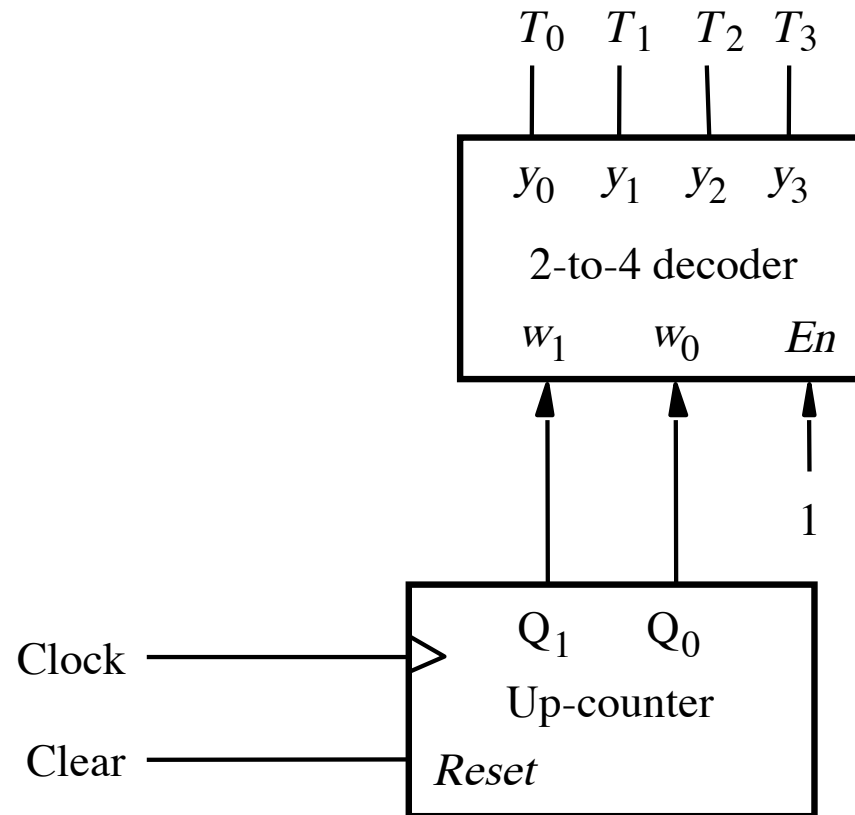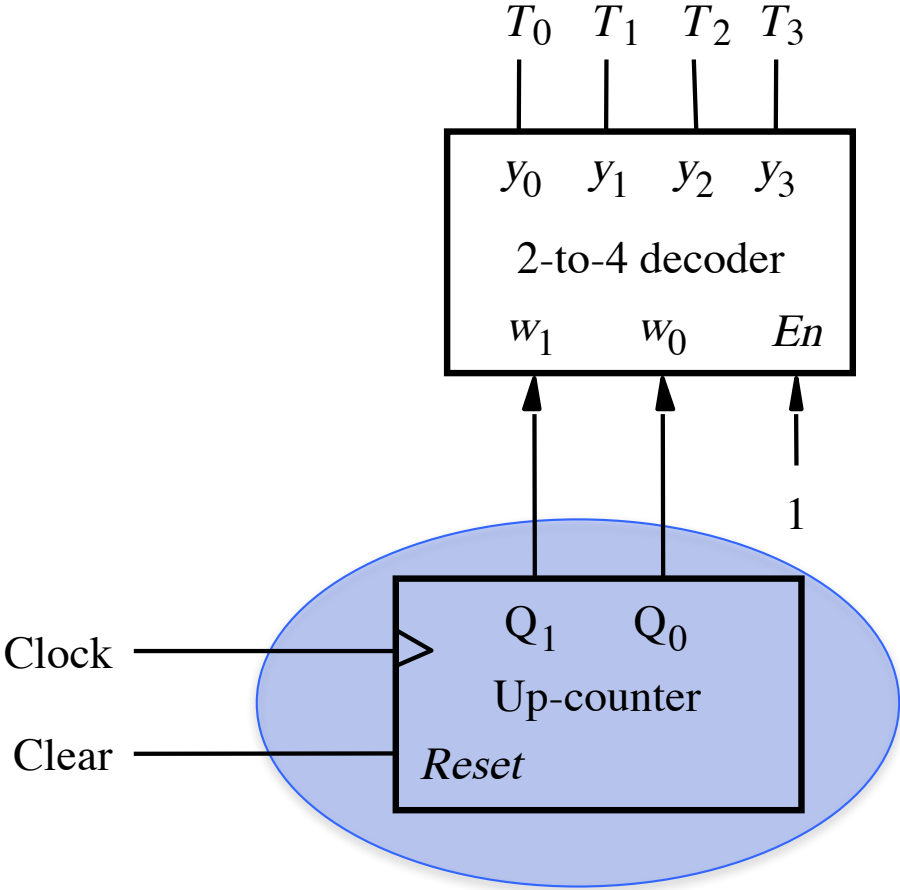
2-Bit Up-Counter

[Figure 7.10 from the textbook]

2-bit Synchronous Up-Counter
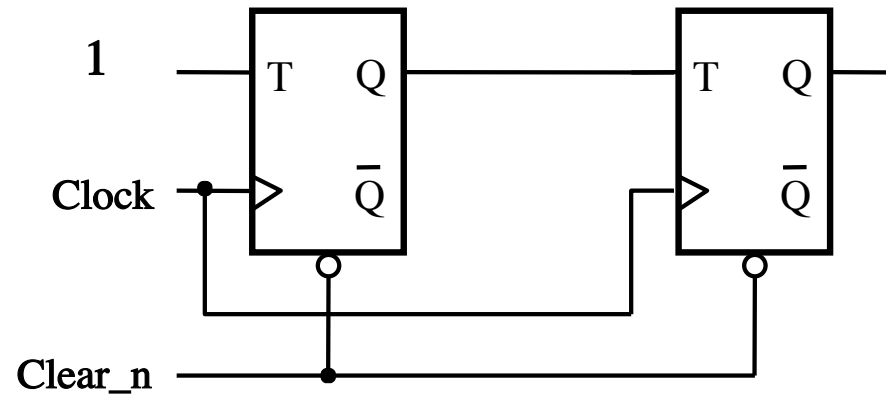
2-bit Synchronous Up-Counter with Enable
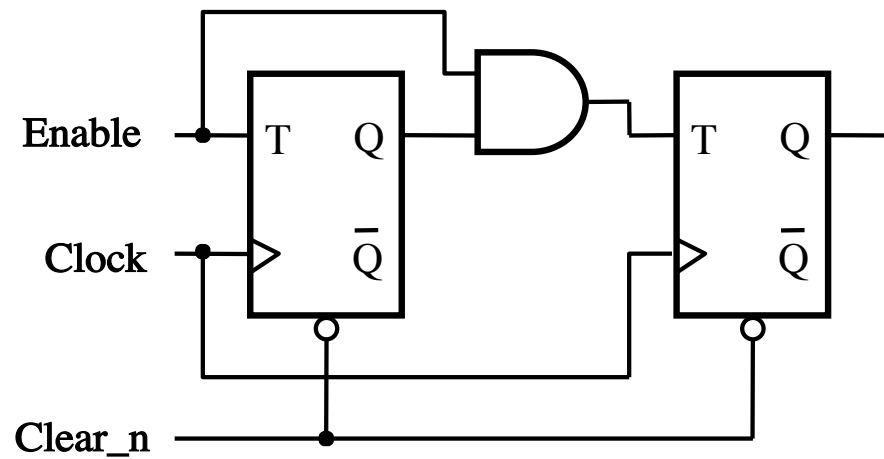
2-to-4 Decoder with Enable Input

[Figure 7.10 from the textbook]
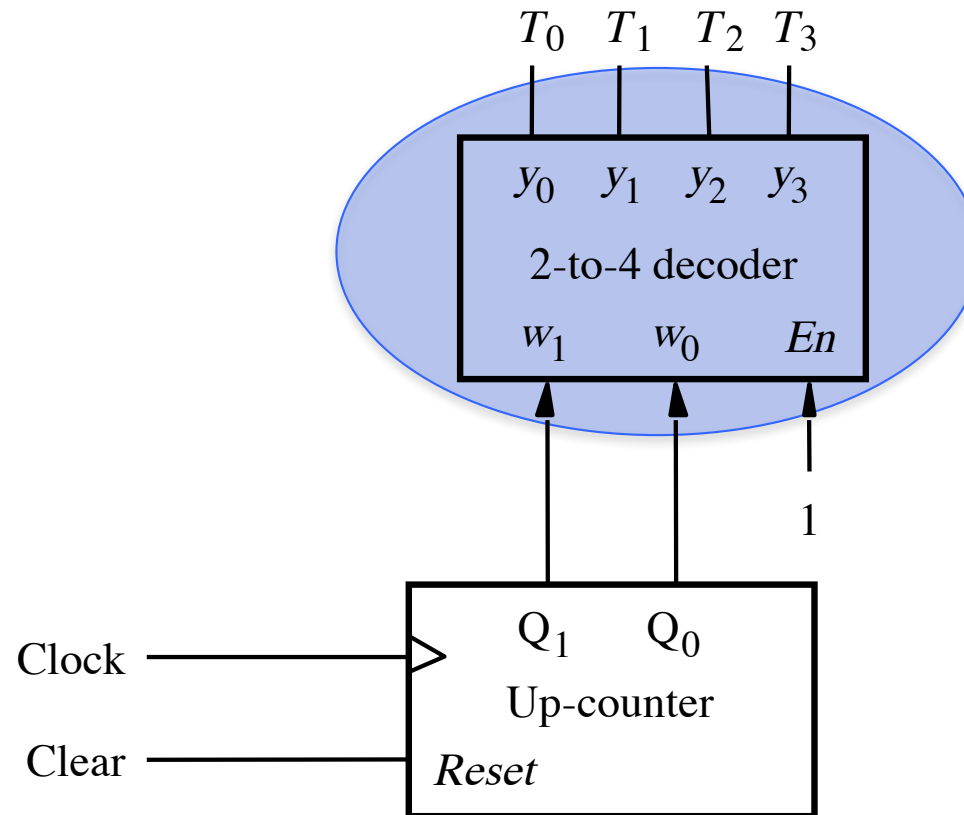
2-to-4 Decoder with an Enable Input

[Figure 4.13c from the textbook]

2-to-4 Decoder with an Enable Input

w_0

w_1

y_0

y_1

y_2

y_3

1
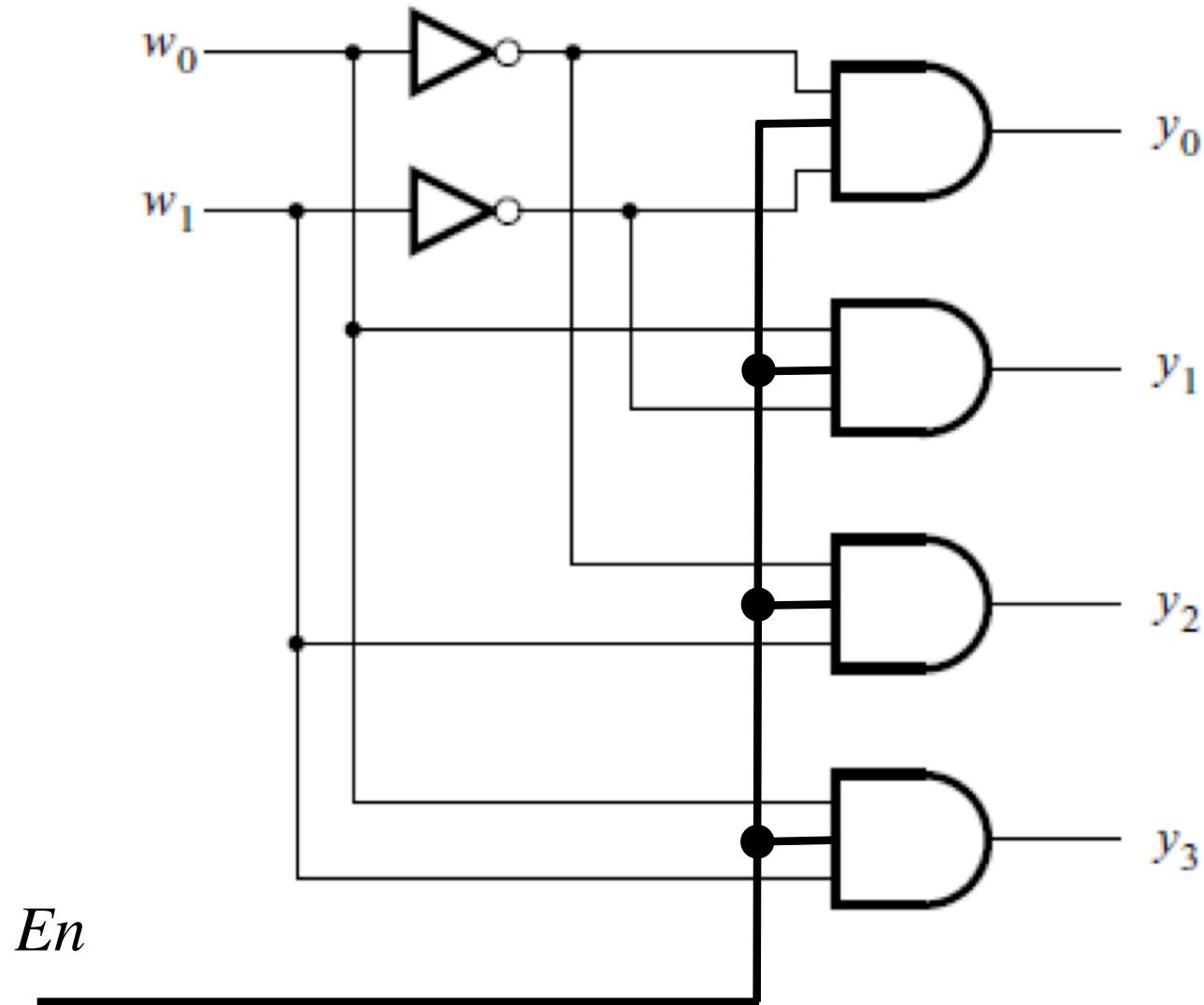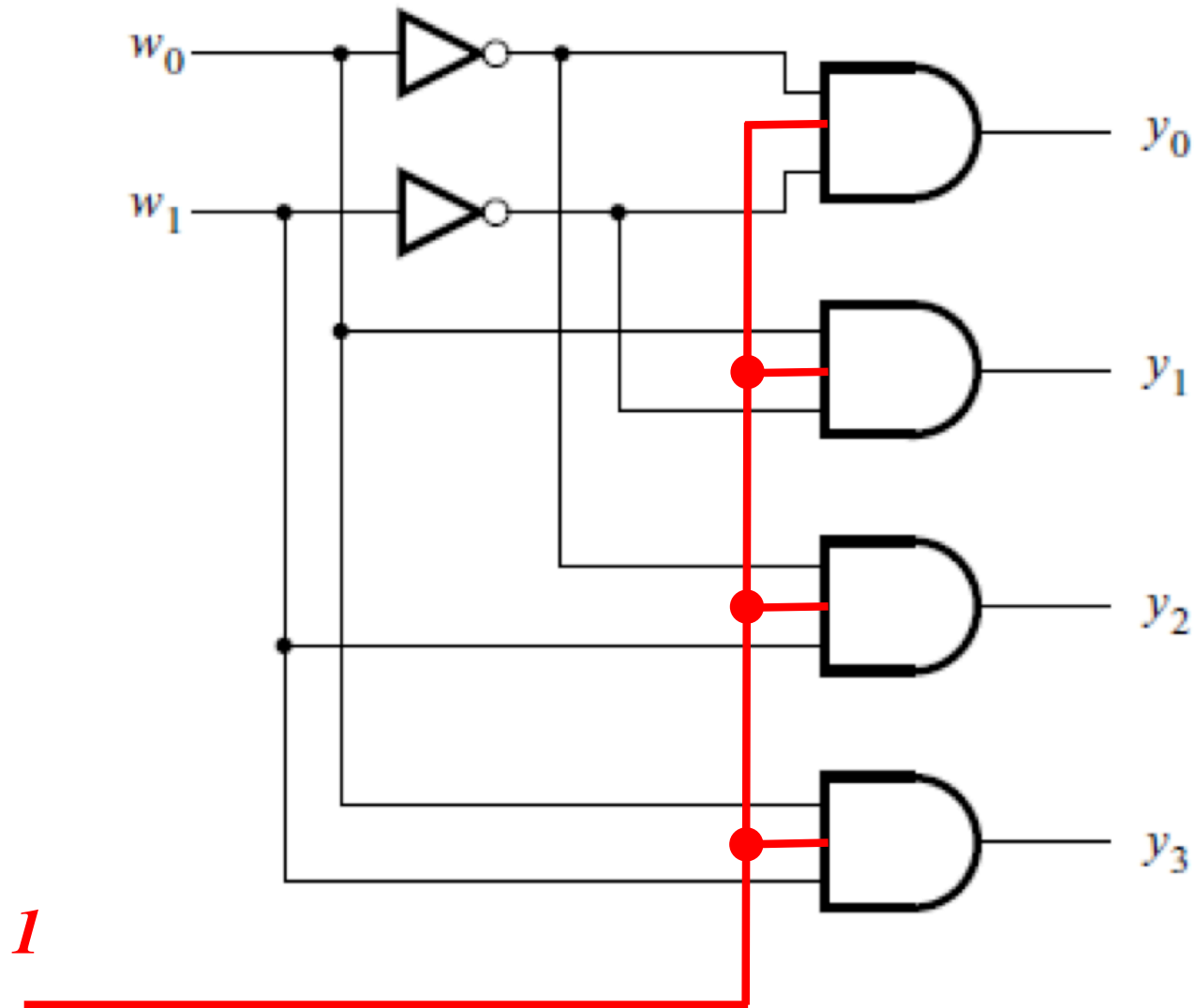
(always enabled in this example)
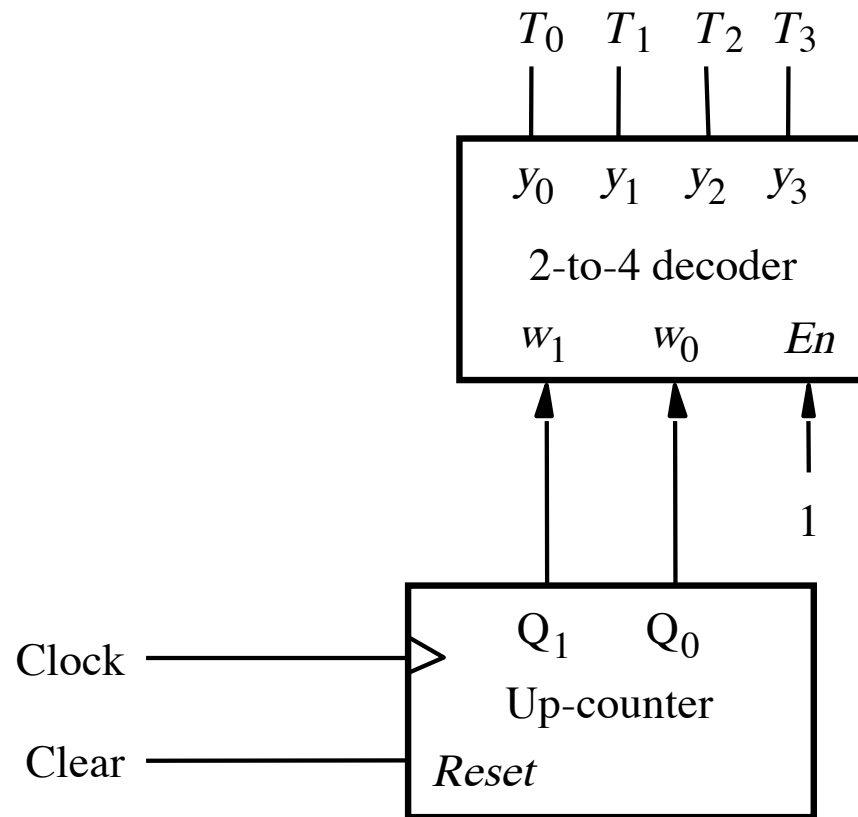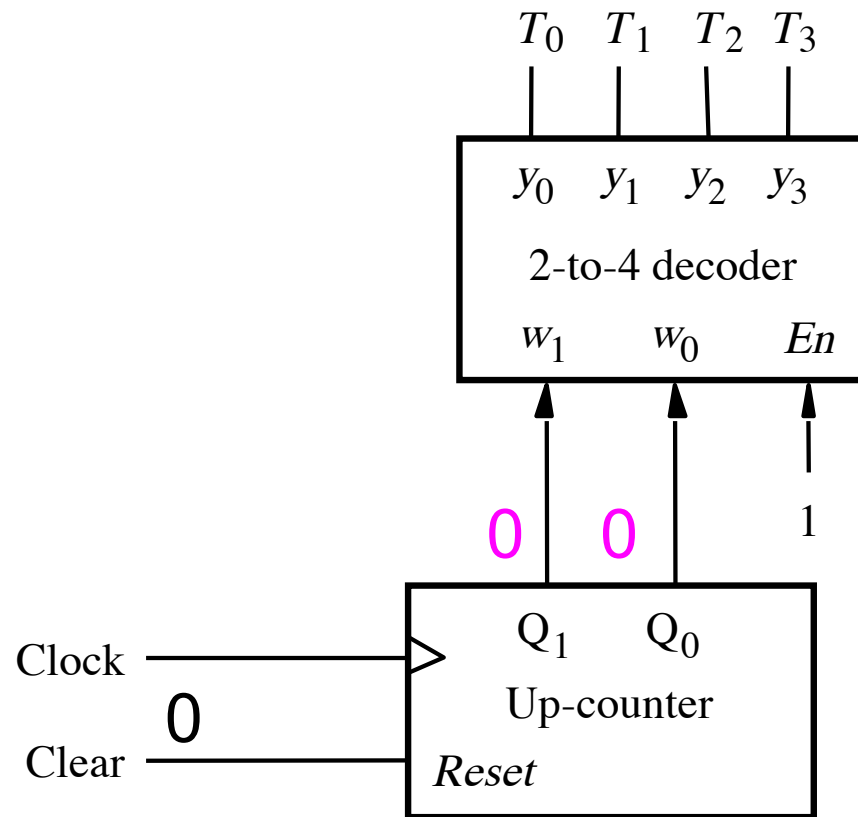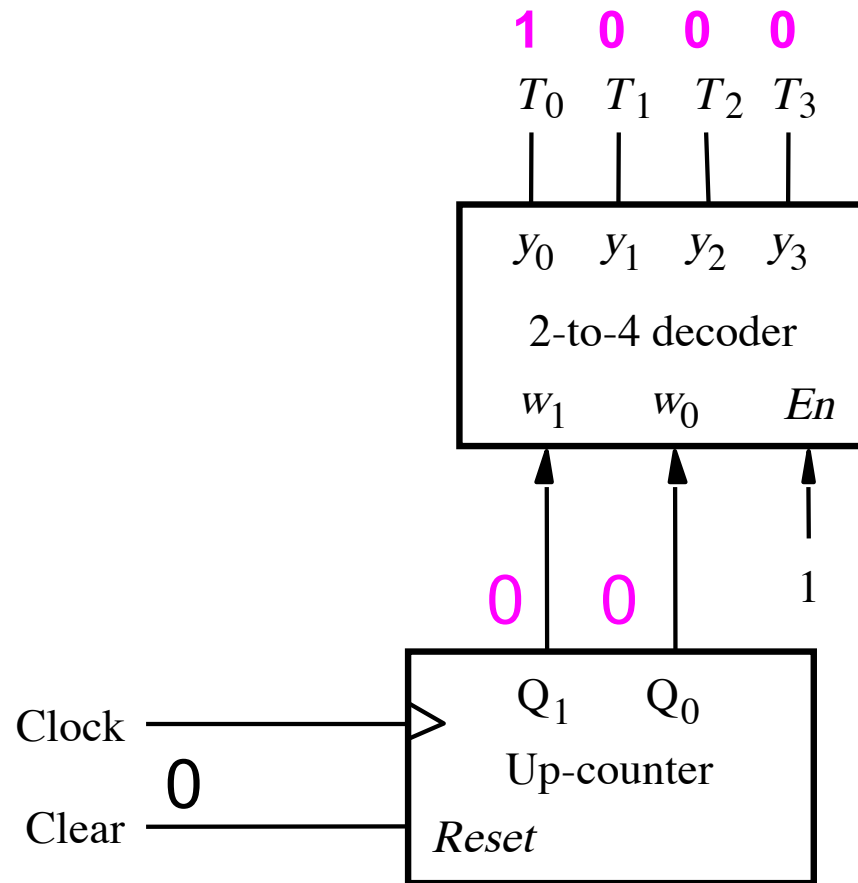
[Figure 4.13c from the textbook]

So How Does This Work?

So How Does This Work?

So How Does This Work?

So How Does This Work?
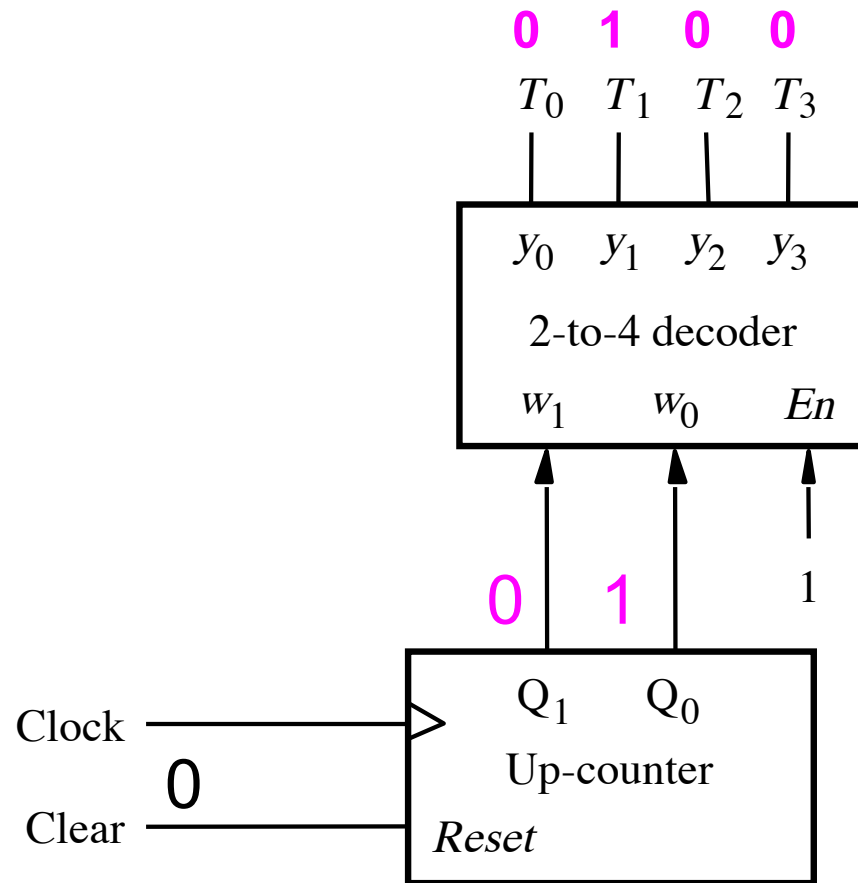
So How Does This Work?
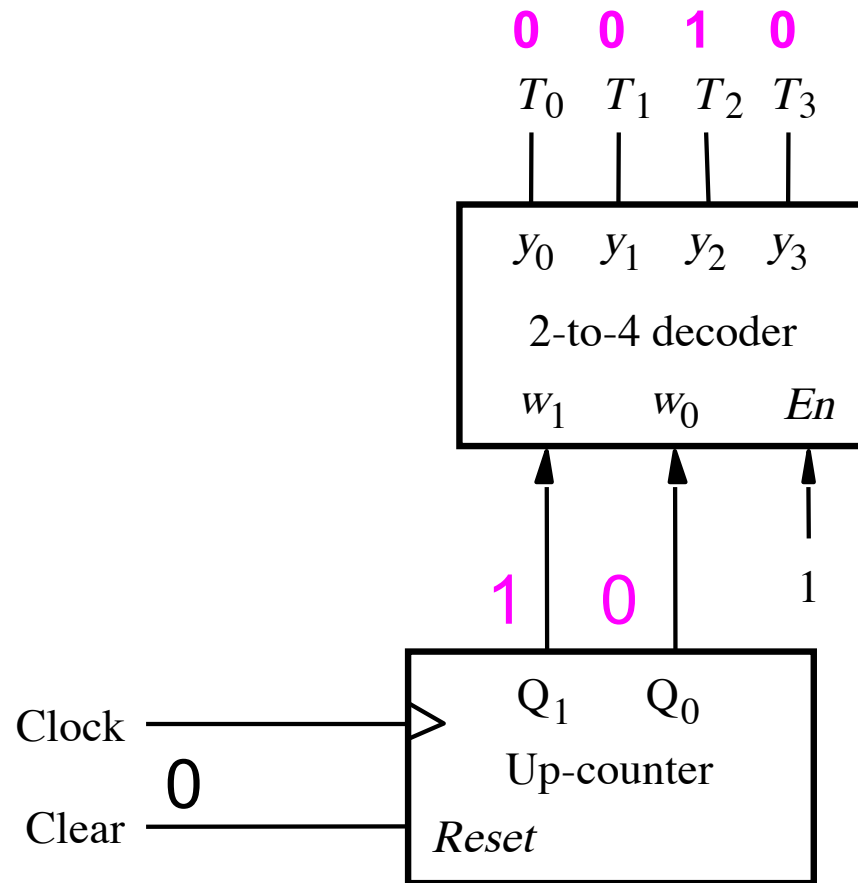
So How Does This Work?
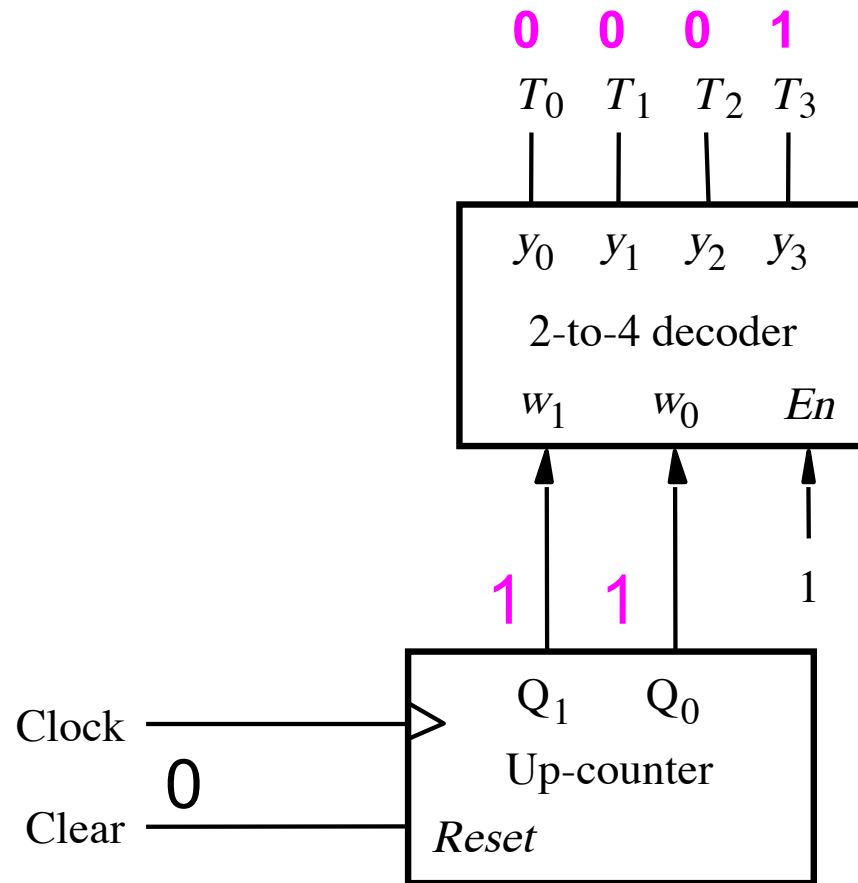
So How Does This Work?
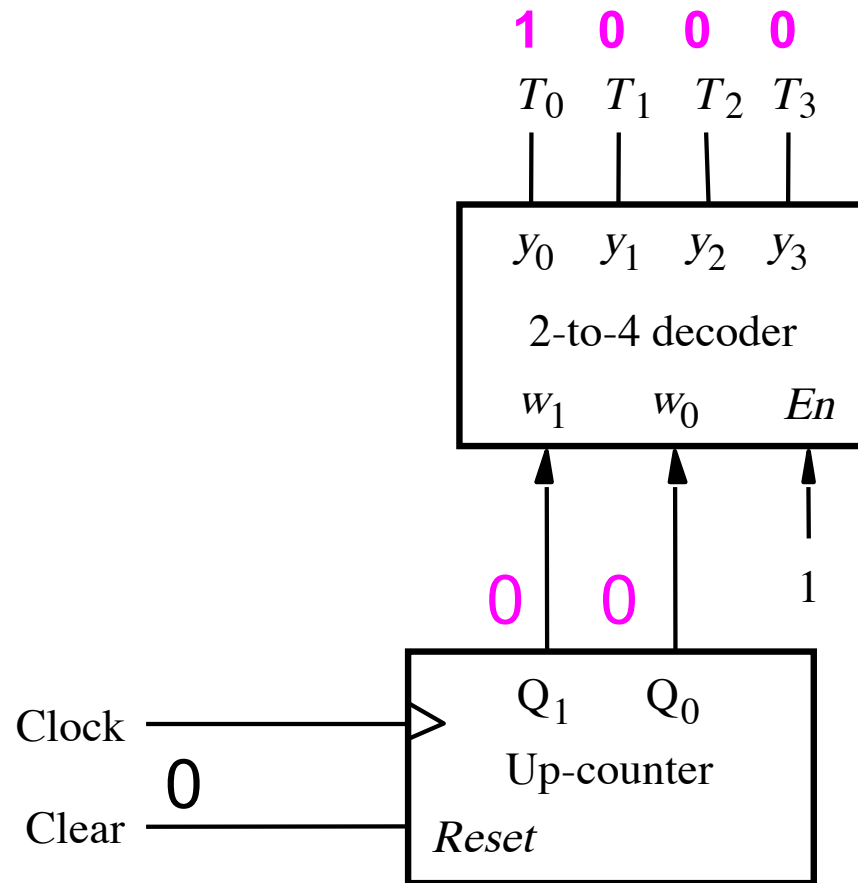
So How Does This Work?
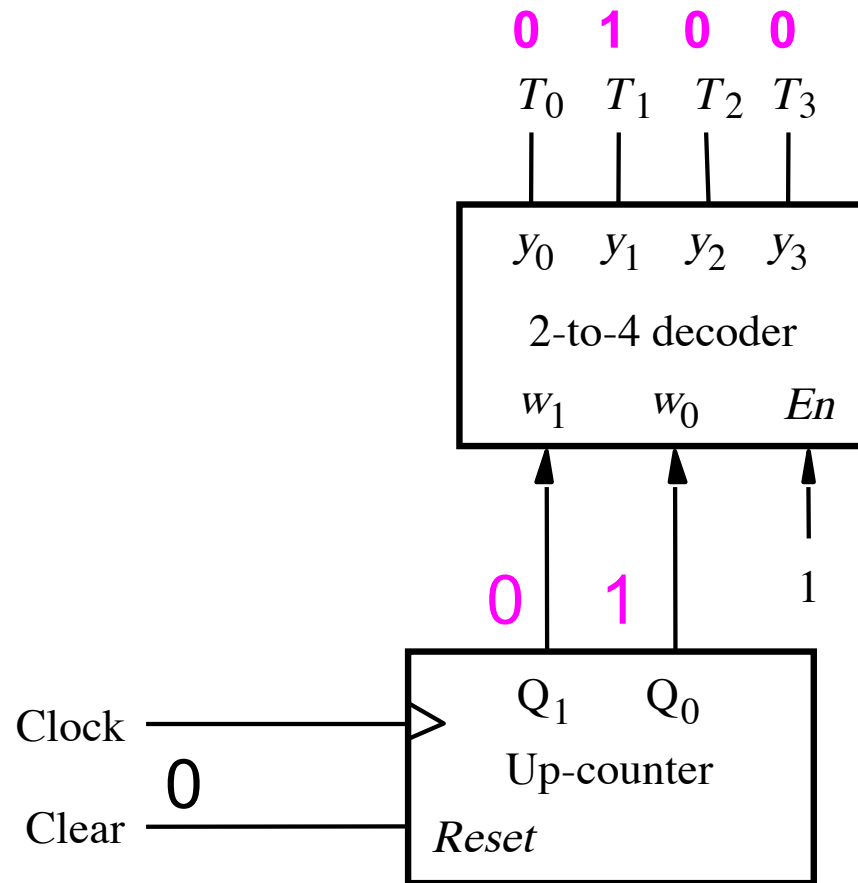
So How Does This Work?
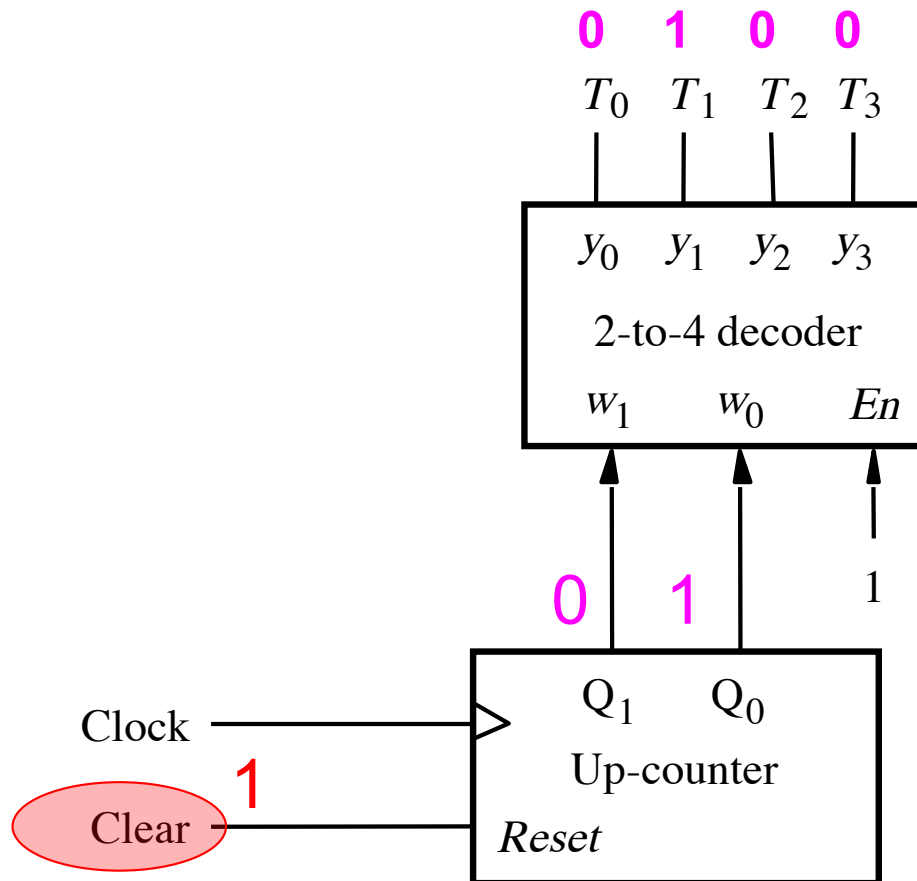
So How Does This Work?
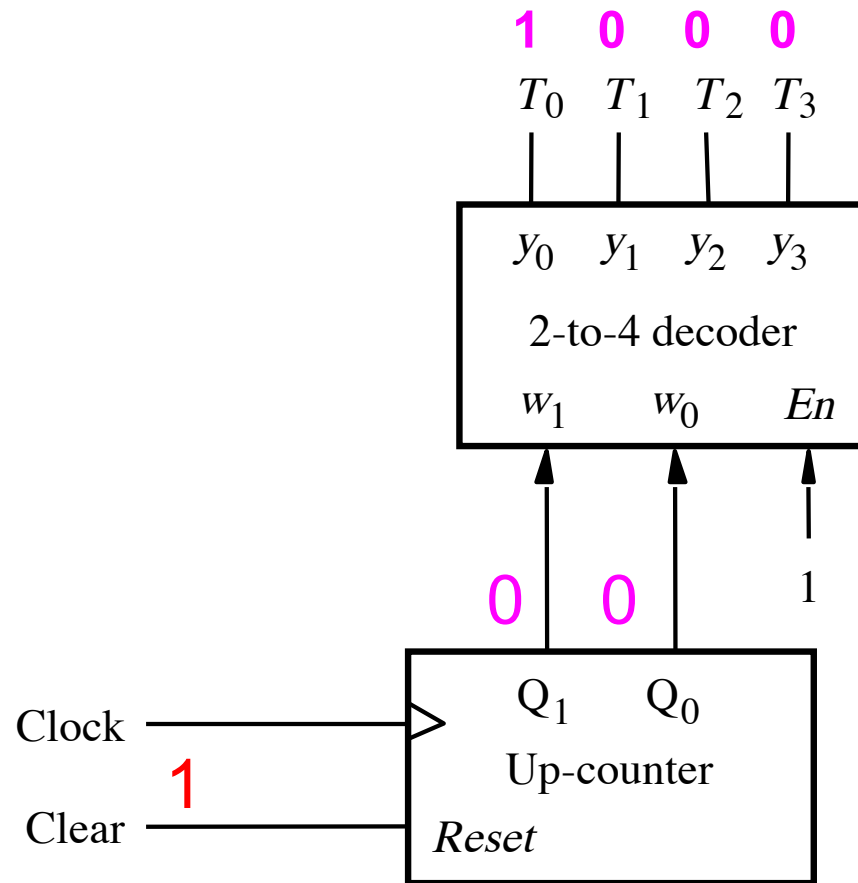
Meaning/Explanation

- This is like a FSM that cycles through its four states one after another.

- But it also can be reset to go to state 0 at any time.

- The implementation uses a counter followed by a decoder. The outputs of the decoder are one-hot-encoded.

- This is like choosing a state assignment for an FSM in which there is one Flip-Flop per state, i.e., one-hot encoding (see Section 6.2.1 in the textbook)

Deriving the Control Signals

Design a FSM with input w and outputs

- $R0_{in}$
- $R0_{out}$

- $R1_{in}$
- $R1_{out}$

- $R2_{in}$
- $R2_{out}$

- $R3_{in}$
- $R3_{out}$

- A_{in}

- Gin
- G_{out}

- Clear

- FR_{in}

- AddSub

- Extern

- Done

- T_0
- T_1
- T_2
- T_3

- I_0
- I_1
- I_2
- I_3

- X_0
- X_1
- X_2
- X_3

- Y_0
- Y_1
- Y_2
- Y_3

These are helper outputs that are one-hot encoded. They are used to simplify the expressions for the other outputs.

Control Signals
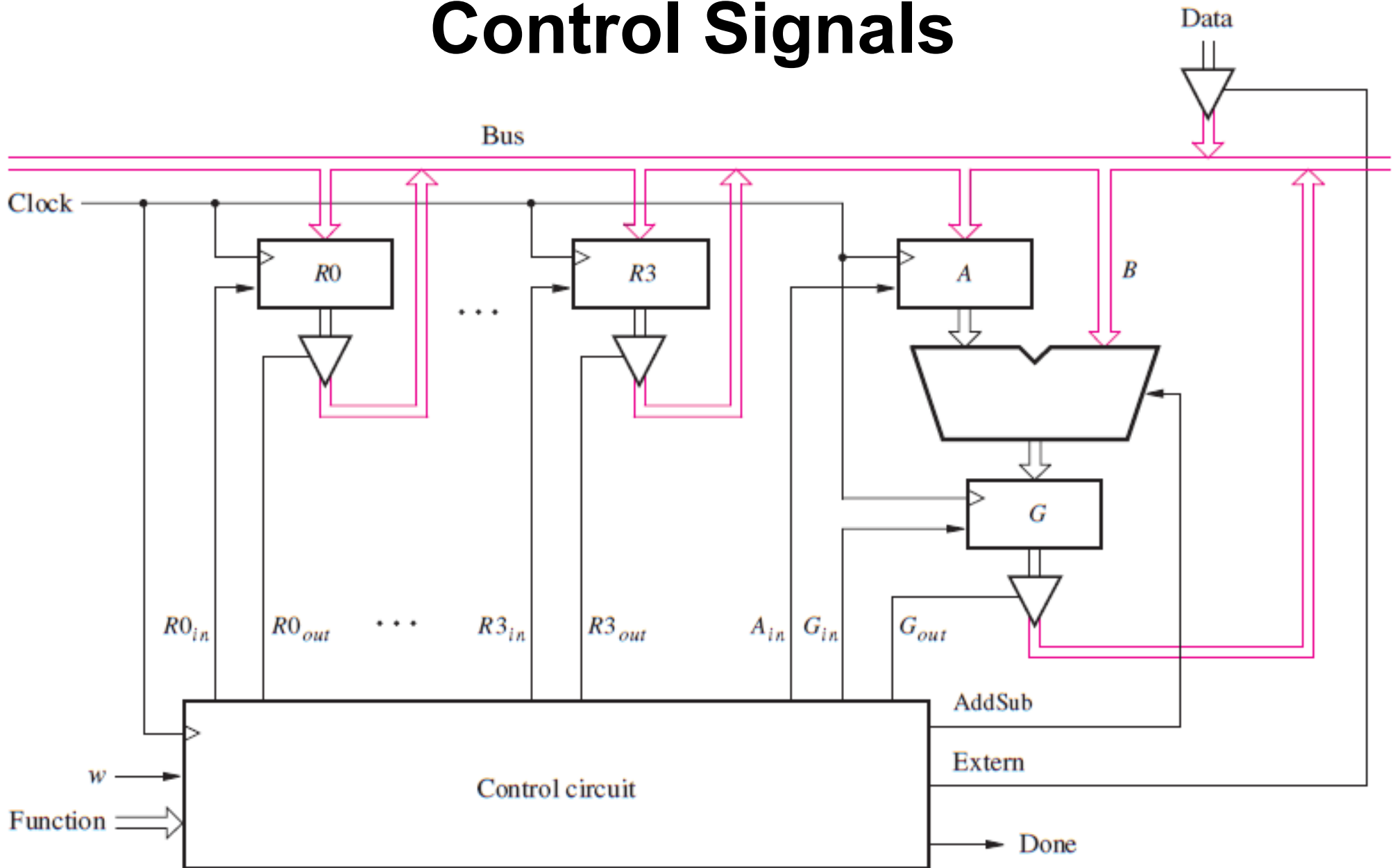
[Figure 7.9 from the textbook]
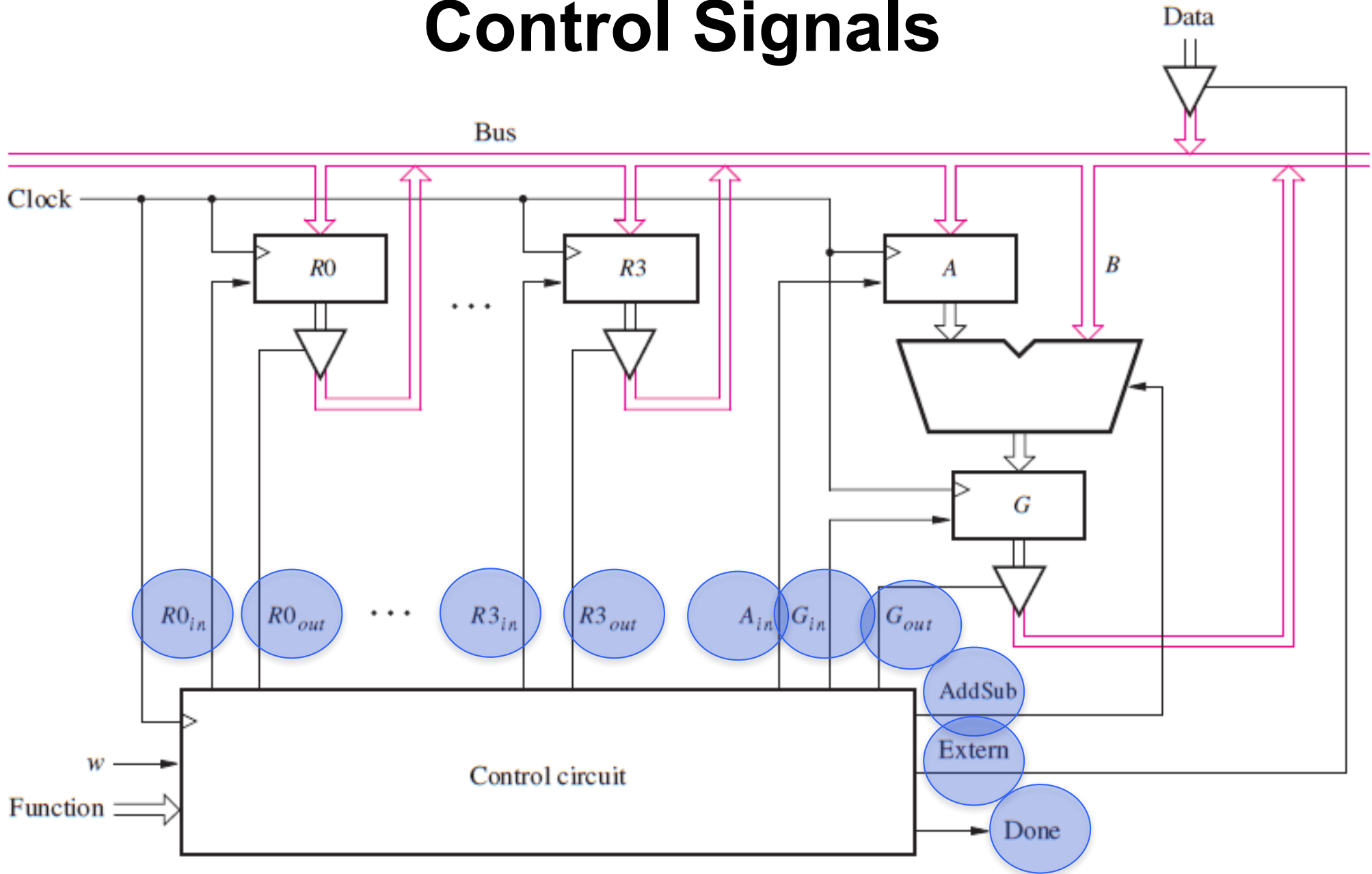
Control Signals

[Figure 7.9 from the textbook]

Another Control Signal

[Figure 7.11 from the textbook]

Yet Another Control Signal

T_0 T_1 T_2 T_3

y_0 y_1 y_2 y_3

2-to-4 decoder

w_1 w_0 En

1

Q_1 Q_0

Clock

Up-counter

Clear

Reset

[Figure 7.10 from the textbook]
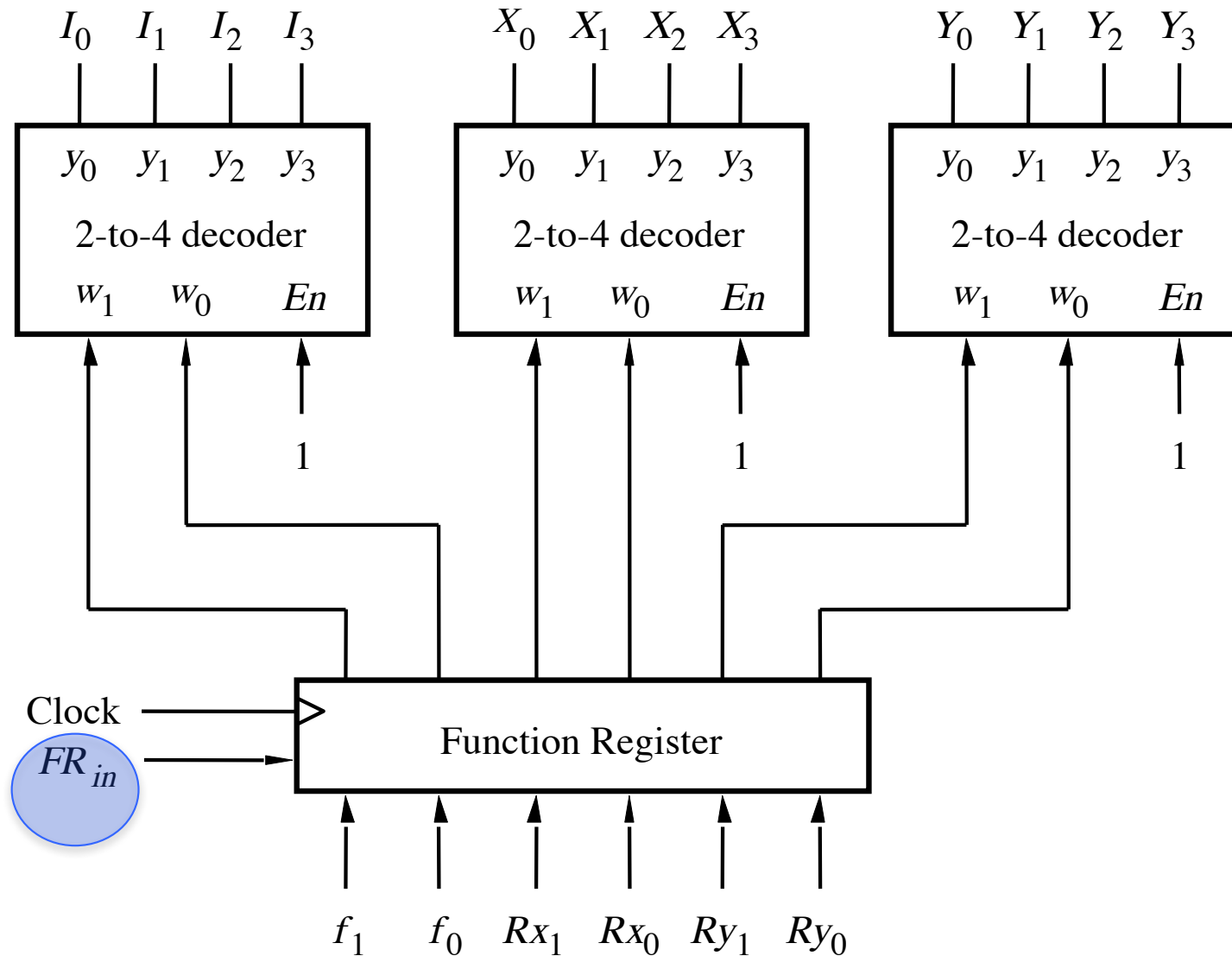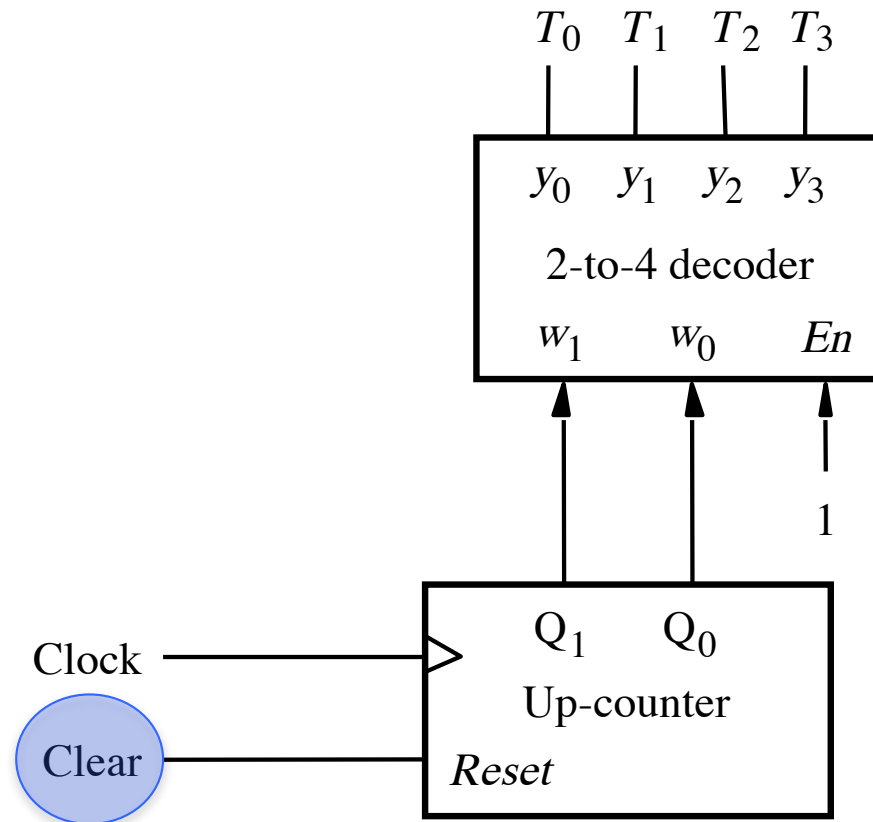
Expressing the 'FR$_{in}$' signal
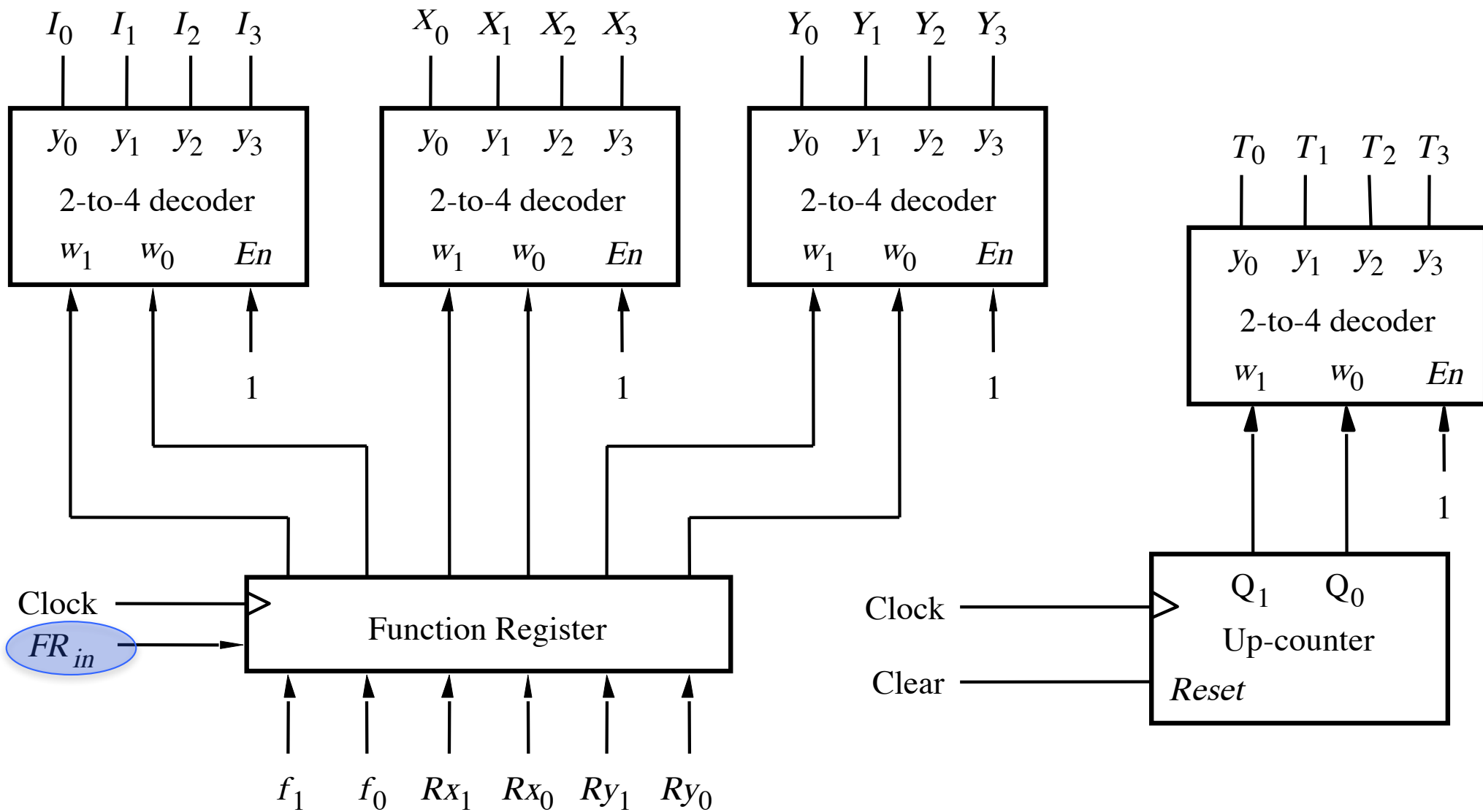
$$FR_{in} = w\, T_0$$

Load a new operation into the function register

Expressing the 'Clear' signal
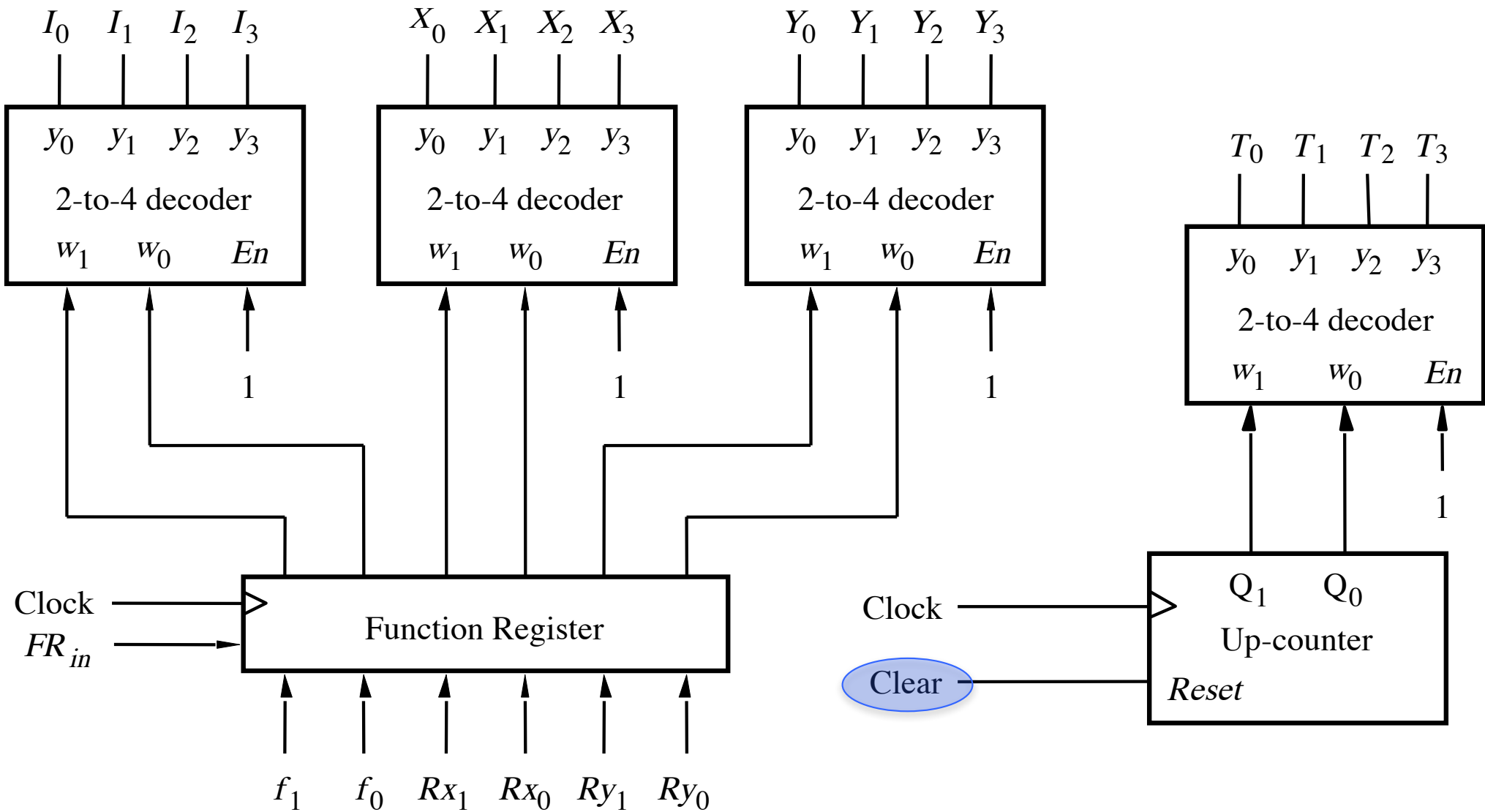
$$\text{Clear} = \overline{w}\, T_0 + \text{Done}$$

Reset the counter when Done or when w=0 and no operation is being executed (i.e., T_0=1).

Control signals asserted in each time step

	T_1	T_2	T_3
(Load): I_0	Extern $R_{in} = X$ Done		
(Move): I_1	$R_{in} = X$ $R_{out} = Y$ Done		
(Add): I_2	$R_{out} = X$ A_{in}	$R_{out} = Y$ G_{in} AddSub = 0	G_{out} $R_{in} = X$ Done
(Sub): I_3	$R_{out} = X$ A_{in}	$R_{out} = Y$ G_{in} AddSub = 1	G_{out} $R_{in} = X$ Done

[Table 7.2 from the textbook]

Control signals asserted in each time step

Time →

	T_1	T_2	T_3
(Load): I_0	Extern $R_{in} = X$ Done		
(Move): I_1	$R_{in} = X$ $R_{out} = Y$ Done		
(Add): I_2	$R_{out} = X$ A_{in}	$R_{out} = Y$ G_{in} AddSub = 0	G_{out} $R_{in} = X$ Done
(Sub): I_3	$R_{out} = X$ A_{in}	$R_{out} = Y$ G_{in} AddSub = 1	G_{out} $R_{in} = X$ Done

These come from the outputs of the 2-to-4 decoder in Figure 7.10. They are also one-hot encoded.

These are the outputs of the first 2-to-4 decoder that is connected to the two most significant bits of the function register. They are one-hot encoded so only one of them is active at any given time (see Fig 7.11).

[Table 7.2 from the textbook]

The I_0, I_1, I_2, I_3 and T_0, T_1, T_2, T_3 Signals

I_0 I_1 I_2 I_3

y_0	y_1	y_2	y_3
2-to-4 decoder			
w_1	w_0		En

1

X_0 X_1 X_2 X_3

y_0	y_1	y_2	y_3
2-to-4 decoder			
w_1	w_0		En

1

Y_0 Y_1 Y_2 Y_3

y_0	y_1	y_2	y_3
2-to-4 decoder			
w_1	w_0		En

1

T_0 T_1 T_2 T_3

y_0	y_1	y_2	y_3
2-to-4 decoder			
w_1	w_0		En

1

Clock

FR_{in}

Function Register

f_1 f_0 Rx_1 Rx_0 Ry_1 Ry_0

Clock

Clear

Q_1 Q_0

Up-counter

$Reset$
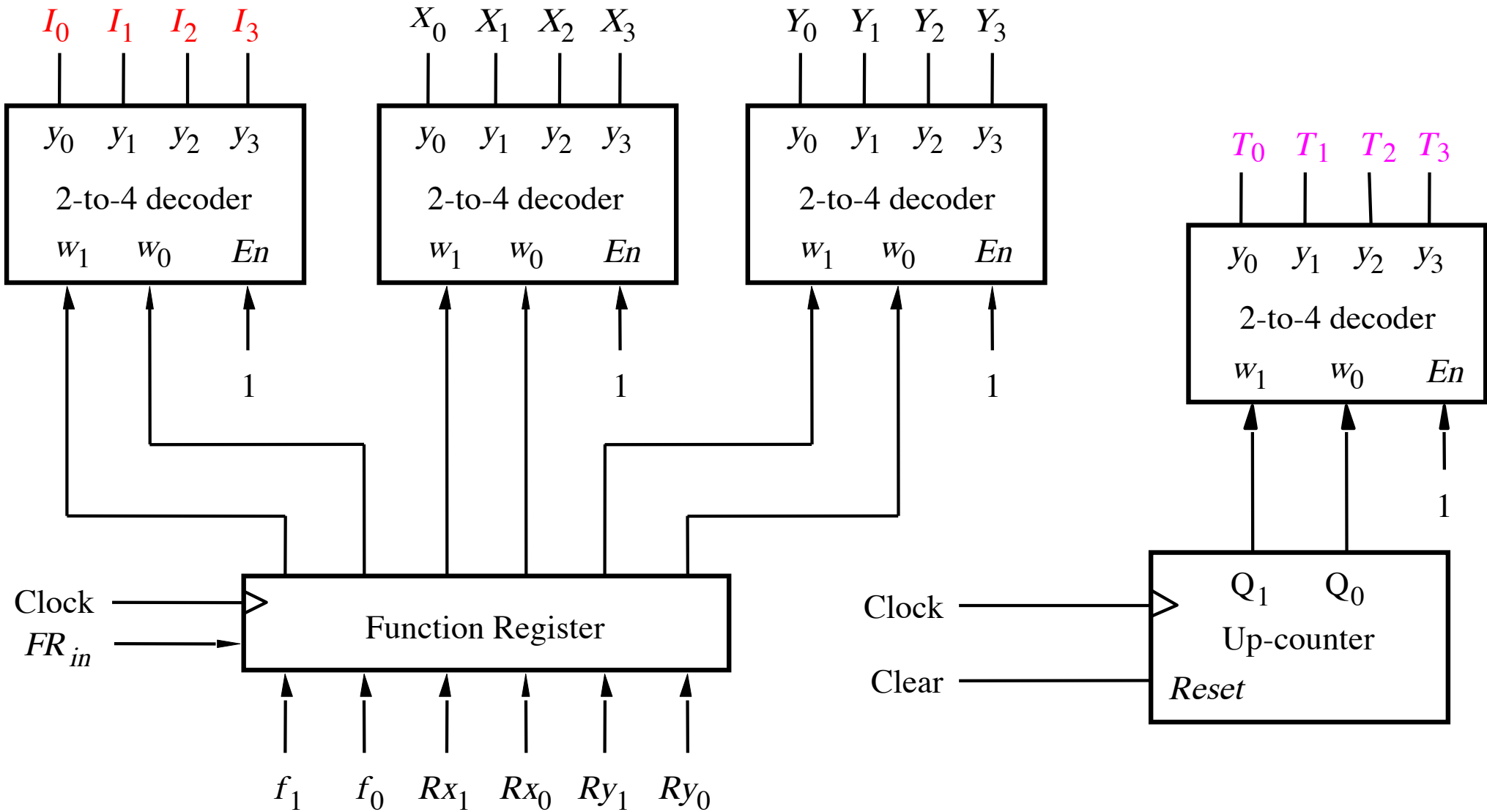
[Figure 7.11 from the textbook]

[Figure 7.10 from the textbook]

Different Operations Take Different Amount of Time

	T_1	T_2	T_3	
(Load): I_0	Extern $R_{in} = X$ Done			1 clock cycle
(Move): I_1	$R_{in} = X$ $R_{out} = Y$ Done			1 clock cycle
(Add): I_2	$R_{out} = X$ A_{in}	$R_{out} = Y$ G_{in} AddSub = 0	G_{out} $R_{in} = X$ Done	3 clock cycles
(Sub): I_3	$R_{out} = X$ A_{in}	$R_{out} = Y$ G_{in} AddSub = 1	G_{out} $R_{in} = X$ Done	3 clock cycles

[Table 7.2 from the textbook]

Operations performed by this processor

Operation	Function Performed
Load Rx, Data	Rx ← Data
Move Rx, Ry	Rx ← [Ry]
Add Rx, Ry	Rx ← [Rx] + [Ry]
Sub Rx, Ry	Rx ← [Rx] - [Ry]

Where Rx and Ry can be one of four possible options: R0, R1, R2, and R3

[Table 7.1 from the textbook]

Simple Processor

Data

Bus

Clock

R0   ...   R3   A   B

G

A_{in} G_{in} G_{out}

AddSub

Extern

Done

	T_1	T_2	T_3
(Load): I_0	Extern R_{in} = X Done		
(Move): I_1	R_{in} = X R_{out} = Y Done		
(Add): I_2	R_{out} = X A_{in}	R_{out} = Y G_{in} AddSub = 0	G_{out} R_{in} = X Done
(Sub): I_3	R_{out} = X A_{in}	R_{out} = Y G_{in} AddSub = 1	G_{out} R_{in} = X Done
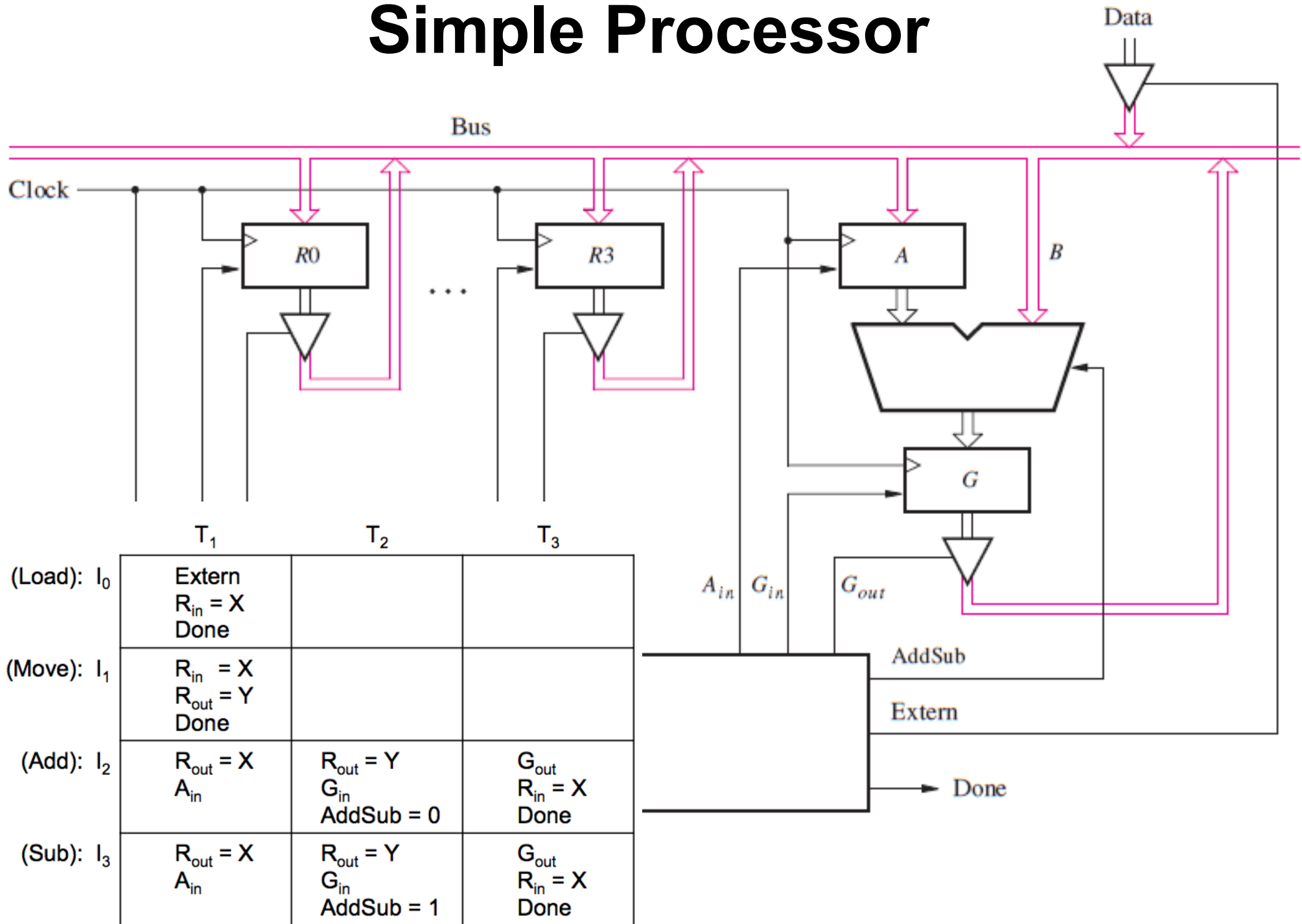
[Figure 7.9 from the textbook]

Expressing the 'Extern' signal

	T_1	T_2	T_3
(Load): I_0	Extern $R_{in} = X$ Done		
(Move): I_1	$R_{in} = X$ $R_{out} = Y$ Done		
(Add): I_2	$R_{out} = X$ A_{in}	$R_{out} = Y$ G_{in} AddSub = 0	G_{out} $R_{in} = X$ Done
(Sub): I_3	$R_{out} = X$ A_{in}	$R_{out} = Y$ G_{in} AddSub = 1	G_{out} $R_{in} = X$ Done

$$\text{Extern} = I_0 \, T_1$$

Expressing the 'Done' signal

	T_1	T_2	T_3
(Load): I_0	Extern $R_{in} = X$ Done		
(Move): I_1	$R_{in} = X$ $R_{out} = Y$ Done		
(Add): I_2	$R_{out} = X$ A_{in}	$R_{out} = Y$ G_{in} AddSub = 0	G_{out} $R_{in} = X$ Done
(Sub): I_3	$R_{out} = X$ A_{in}	$R_{out} = Y$ G_{in} AddSub = 1	G_{out} $R_{in} = X$ Done

$$\text{Done} = (I_0 + I_1)T_1 + (I_2 + I_3)T_3$$

Expressing the 'A_{in}' signal

	T_1	T_2	T_3
(Load): I_0	Extern $R_{in} = X$ Done		
(Move): I_1	$R_{in} = X$ $R_{out} = Y$ Done		
(Add): I_2	$R_{out} = X$ A_{in}	$R_{out} = Y$ G_{in} AddSub = 0	G_{out} $R_{in} = X$ Done
(Sub): I_3	$R_{out} = X$ A_{in}	$R_{out} = Y$ G_{in} AddSub = 1	G_{out} $R_{in} = X$ Done

$$A_{in} = (I_2 + I_3)T_1$$

Expressing the 'G_{in}' signal

	T_1	T_2	T_3
(Load): I_0	Extern $R_{in} = X$ Done		
(Move): I_1	$R_{in} = X$ $R_{out} = Y$ Done		
(Add): I_2	$R_{out} = X$ A_{in}	$R_{out} = Y$ G_{in} AddSub = 0	G_{out} $R_{in} = X$ Done
(Sub): I_3	$R_{out} = X$ A_{in}	$R_{out} = Y$ G_{in} AddSub = 1	G_{out} $R_{in} = X$ Done

$$G_{in} = (I_2 + I_3)T_2$$

Expressing the 'G$_{out}$' signal

	T$_1$	T$_2$	T$_3$
(Load): I$_0$	Extern R$_{in}$ = X Done		
(Move): I$_1$	R$_{in}$ = X R$_{out}$ = Y Done		
(Add): I$_2$	R$_{out}$ = X A$_{in}$	R$_{out}$ = Y G$_{in}$ AddSub = 0	G$_{out}$ R$_{in}$ = X Done
(Sub): I$_3$	R$_{out}$ = X A$_{in}$	R$_{out}$ = Y G$_{in}$ AddSub = 1	G$_{out}$ R$_{in}$ = X Done

$$G_{out} = (I_2 + I_3)T_3$$

Expressing the 'AddSub' signal

	T_1	T_2	T_3
(Load): I_0	Extern $R_{in} = X$ Done		
(Move): I_1	$R_{in} = X$ $R_{out} = Y$ Done		
(Add): I_2	$R_{out} = X$ A_{in}	$R_{out} = Y$ G_{in} AddSub = 0	G_{out} $R_{in} = X$ Done
(Sub): I_3	$R_{out} = X$ A_{in}	$R_{out} = Y$ G_{in} AddSub = 1	G_{out} $R_{in} = X$ Done

$$AddSub = I_3$$

Expressing the 'R0$_{in}$' signal

	T_1	T_2	T_3
(Load): I_0	Extern R_{in} = X Done		
(Move): I_1	R_{in} = X R_{out} = Y Done		
(Add): I_2	R_{out} = X A_{in}	R_{out} = Y G_{in} AddSub = 0	G_{out} R_{in} = X Done
(Sub): I_3	R_{out} = X A_{in}	R_{out} = Y G_{in} AddSub = 1	G_{out} R_{in} = X Done

$$R0_{in} = (I_0 + I_1)T_1 X_0 + (I_2 + I_3)T_3 X_0$$

Expressing the 'R1$_{in}$' signal

	T_1	T_2	T_3
(Load): I_0	Extern R_{in} = X Done		
(Move): I_1	R_{in} = X R_{out} = Y Done		
(Add): I_2	R_{out} = X A_{in}	R_{out} = Y G_{in} AddSub = 0	G_{out} R_{in} = X Done
(Sub): I_3	R_{out} = X A_{in}	R_{out} = Y G_{in} AddSub = 1	G_{out} R_{in} = X Done

$$R1_{in} = (I_0 + I_1)T_1\,X_1 \; + \; (I_2 + I_3)T_3\,X_1$$

Expressing the 'R2$_{in}$' signal

	T_1	T_2	T_3
(Load): I_0	Extern R_{in} = X Done		
(Move): I_1	R_{in} = X R_{out} = Y Done		
(Add): I_2	R_{out} = X A_{in}	R_{out} = Y G_{in} AddSub = 0	G_{out} R_{in} = X Done
(Sub): I_3	R_{out} = X A_{in}	R_{out} = Y G_{in} AddSub = 1	G_{out} R_{in} = X Done

$$R2_{in} = (I_0 + I_1)T_1 X_2 + (I_2 + I_3)T_3 X_2$$

Expressing the 'R3$_{in}$' signal

	T_1	T_2	T_3
(Load): I_0	Extern R_{in} = X Done		
(Move): I_1	R_{in} = X R_{out} = Y Done		
(Add): I_2	R_{out} = X A_{in}	R_{out} = Y G_{in} AddSub = 0	G_{out} R_{in} = X Done
(Sub): I_3	R_{out} = X A_{in}	R_{out} = Y G_{in} AddSub = 1	G_{out} R_{in} = X Done

$$R3_{in} = (I_0 + I_1)T_1 X_3 + (I_2 + I_3)T_3 X_3$$

Expressing the 'R0$_{out}$' signal

	T_1	T_2	T_3
(Load): I_0	Extern $R_{in} = X$ Done		
(Move): I_1	$R_{in} = X$ $R_{out} = Y$ Done		
(Add): I_2	$R_{out} = X$ A_{in}	$R_{out} = Y$ G_{in} AddSub = 0	G_{out} $R_{in} = X$ Done
(Sub): I_3	$R_{out} = X$ A_{in}	$R_{out} = Y$ G_{in} AddSub = 1	G_{out} $R_{in} = X$ Done

$$R0_{out} = I_1 T_1 Y_0 \ + \ (I_2 + I_3)(T_1 X_0 + T_2 Y_0)$$

Expressing the 'R1_out' signal

	T_1	T_2	T_3
(Load): I_0	Extern $R_{in} = X$ Done		
(Move): I_1	$R_{in} = X$ $R_{out} = Y$ Done		
(Add): I_2	$R_{out} = X$ A_{in}	$R_{out} = Y$ G_{in} AddSub = 0	G_{out} $R_{in} = X$ Done
(Sub): I_3	$R_{out} = X$ A_{in}	$R_{out} = Y$ G_{in} AddSub = 1	G_{out} $R_{in} = X$ Done

$$R1_{out} = I_1 T_1 Y_1 + (I_2 + I_3)(T_1 X_1 + T_2 Y_1)$$

Expressing the 'R2$_{out}$' signal

	T_1	T_2	T_3
(Load): I_0	Extern $R_{in} = X$ Done		
(Move): I_1	$R_{in} = X$ $R_{out} = Y$ Done		
(Add): I_2	$R_{out} = X$ A_{in}	$R_{out} = Y$ G_{in} AddSub = 0	G_{out} $R_{in} = X$ Done
(Sub): I_3	$R_{out} = X$ A_{in}	$R_{out} = Y$ G_{in} AddSub = 1	G_{out} $R_{in} = X$ Done

$$R2_{out} = I_1 T_1 Y_2 + (I_2 + I_3)(T_1 X_2 + T_2 Y_2)$$

Expressing the 'R3$_{out}$' signal

	T_1	T_2	T_3
(Load): I_0	Extern $R_{in} = X$ Done		
(Move): I_1	$R_{in} = X$ (R_{out}) = Y Done		
(Add): I_2	(R_{out}) = X A_{in}	(R_{out}) = Y G_{in} AddSub = 0	G_{out} $R_{in} = X$ Done
(Sub): I_3	(R_{out}) = X A_{in}	(R_{out}) = Y G_{in} AddSub = 1	G_{out} $R_{in} = X$ Done

$$R3_{out} = I_1 T_1 Y_3 + (I_2 + I_3)(T_1 X_3 + T_2 Y_3)$$

Derivation of the Control Inputs

- For more insights into these derivations
  see pages 434 and 435 in the textbook

Some Additional Topics

The ALU for the Simple Processor

For this ALU,
A+B if AddSub=1
A-B if AddSub=0

Data

Bus

Clock

R0

R3

A

B

. . .

G

Clock

R0in R0out . . . R3in R3out   Ain Gin   Gout

AddSub

w

Function

Control circuit

Extern

Done
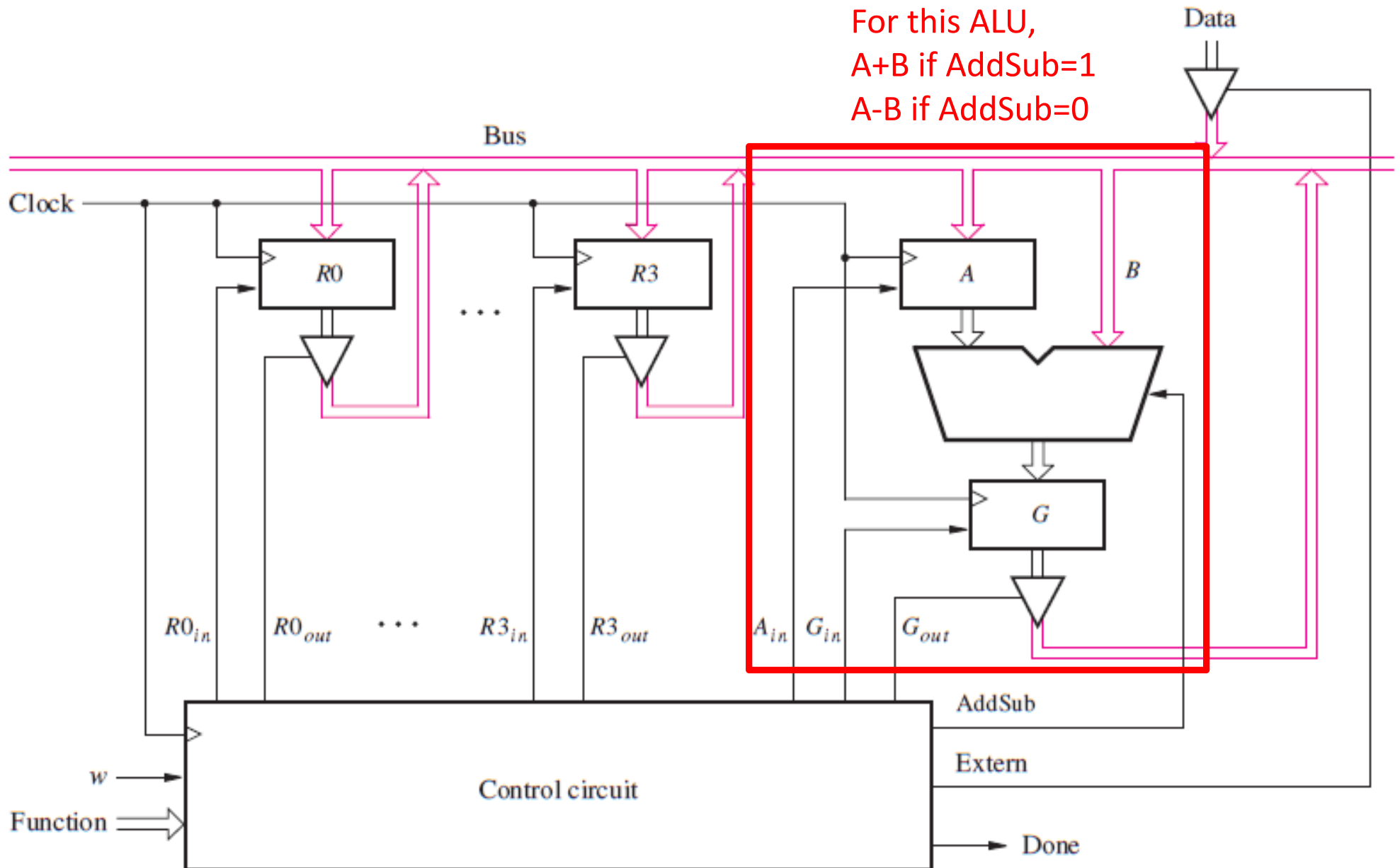
[Figure 7.9 from the textbook]

Another Arithmetic Logic Unit (ALU)

- Arithmetic Logic Unit (ALU) computes arithmetic or logic functions
- Example: A four-function ALU has two selection bits S1 S0

  (also called OpCode) to specify the function
  - 00 (ADD), 01 (SUB), 10 (AND), 11 (OR)
- Then the following set up will work

S1	S0	Function
0	0	ADD
0	1	SUB
1	0	AND
1	1	OR
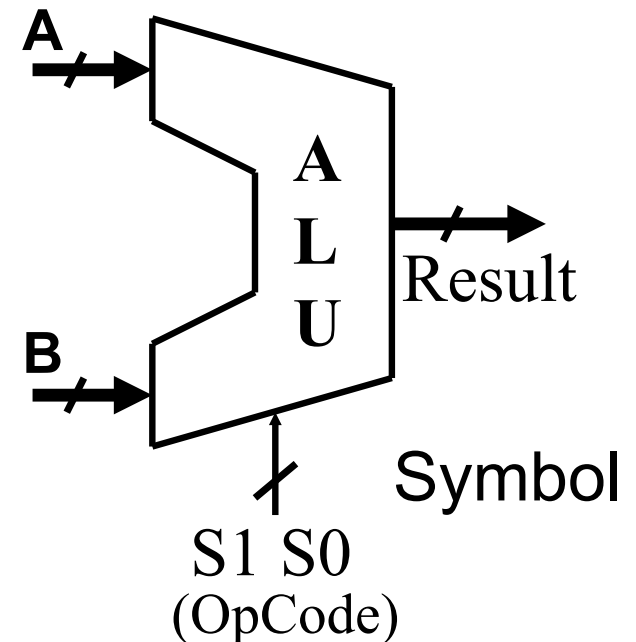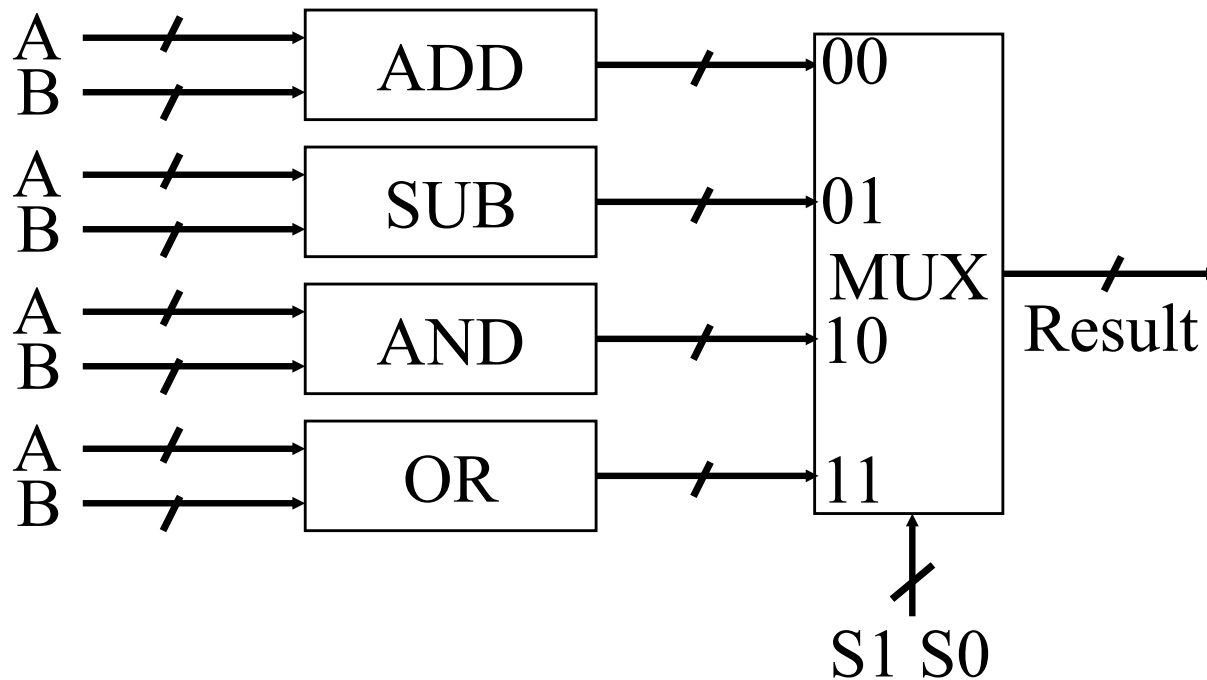
An Alternative Design of Four-Function ALU

- The previous design is not very efficient as it uses an adder and a subtractor circuit
- We can design an add/subtract unit as discussed earlier
- Then we can design a logical unit (AND and OR) separately
- Then select appropriate output as result
- What are the control signals, Add/Sub, Select0, and Select1?
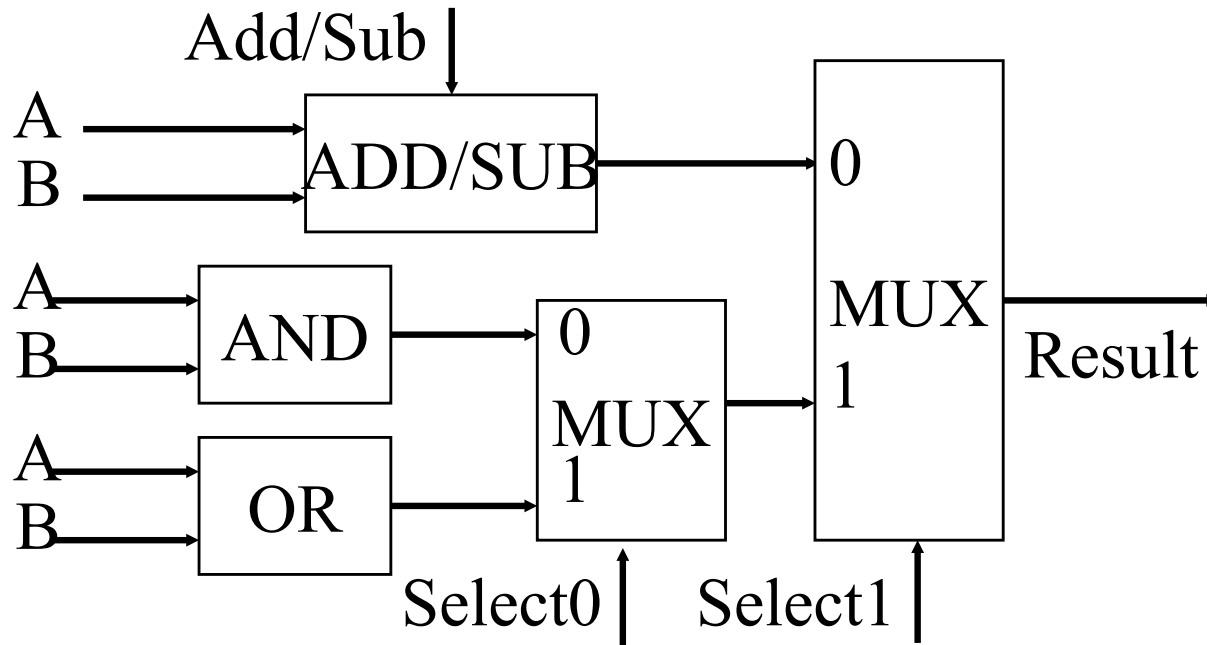
S1	S0	Function
0	0	ADD
0	1	SUB
1	0	AND
1	1	OR

Examples of Some Famous Microprocessors

Intel's 4004 Chip

Reset — 9 | 8 — Sync-Out
Test — 10 | 7 — Clk Phase 2
Memory Control Output { CM-ROM — 11 | 6 — Clk Phase 1
V_{DD} — 12 | 5 — V_{SS}

i4004

Memory Control Outputs {
CM-RAM3 — 13 | 4 — D3
CM-RAM2 — 14 | 3 — D2
CM-RAM1 — 15 | 2 — D1
CM-RAM0 — 16 | 1 — D0
} Data Bus I/O

[http://en.wikipedia.org/wiki/Intel_4004]
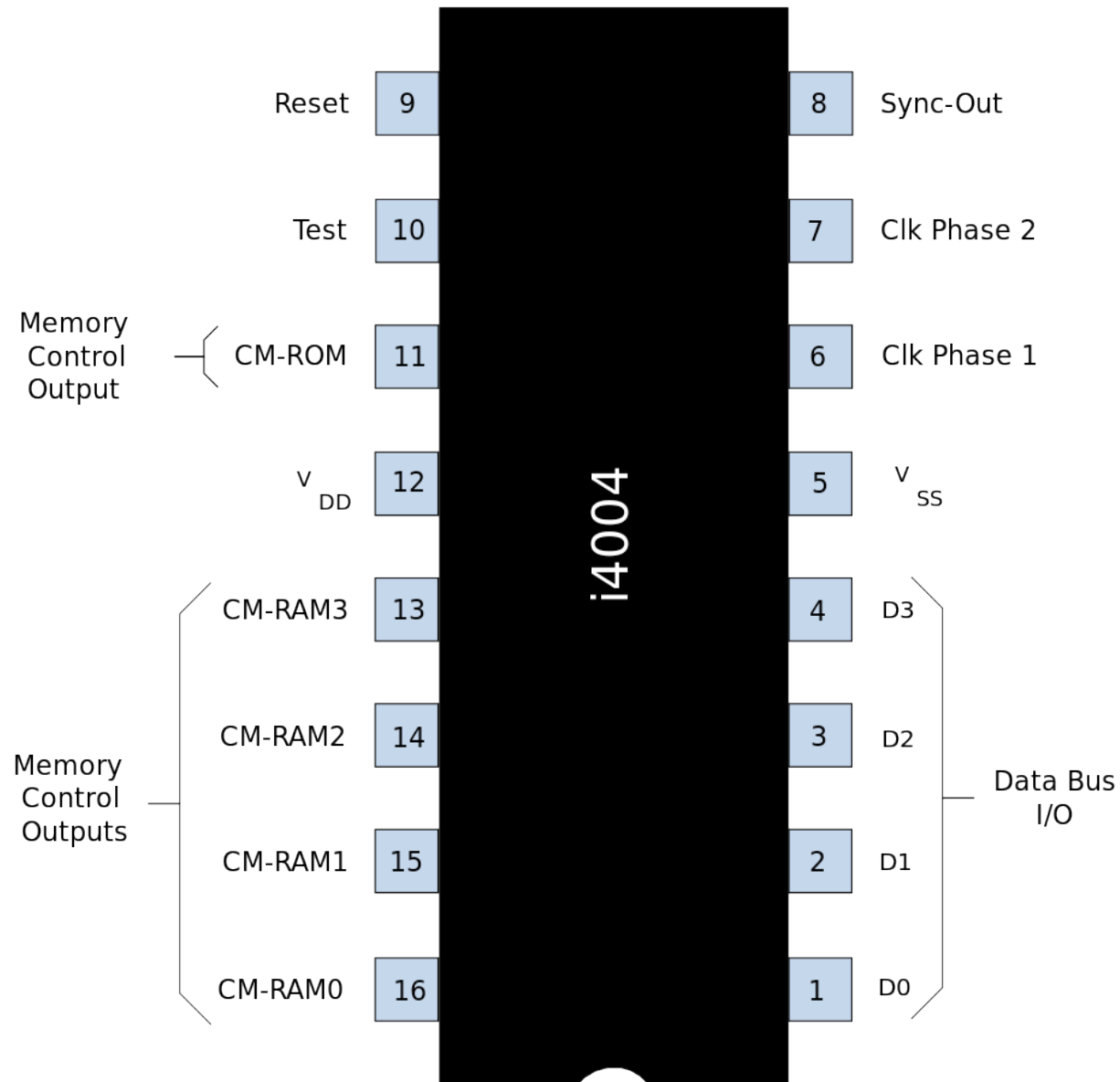
Technical specifications

- Maximum clock speed was 740 kHz

- Instruction cycle time: 10.8 µs
  (8 clock cycles / instruction cycle)

- Instruction execution time 1 or 2 instruction cycles (10.8 or 21.6 µs), 46300 to 92600 instructions per second

- Built using 2,300 transistors

Technical specifications

- **Separate program and data storage.**

- **The 4004, with its need to keep pin count down, used a single multiplexed 4-bit bus for transferring:**
  - **12-bit addresses**
  - **8-bit instructions**
  - **4-bit data words**

- **Instruction set contained 46 instructions (of which 41 were 8 bits wide and 5 were 16 bits wide)**

- **Register set contained 16 registers of 4 bits each**

- **Internal subroutine stack, 3 levels deep.**

[http://en.wikipedia.org/wiki/Intel_4004]

Intel 4004 Architecture

D0-D3 bidirectional Data Bus

Data Bus Buffer

4 Bit internal Data Bus

Accumulator

Temp. Register

Flag Flip Flops

ALU

Decimal Adjust

Instruction Register

Instruction Decoder and Machine Cycle Encoding

Stack Multiplexer

Stack Pointer

Program Counter

Level No. 1

Level No. 2

Level No. 3

Address Stack

Register Multiplexer

Index Register Select

0	1
2	3
4	5
6	7
8	9
10	11
12	13
14	15

Scratch Pad

Timing and Control

ROM Control   RAM Control   Test   Sync   Clocks   Reset

CM ROM   CM RAM 0-3   Test   Sync   Ph1 Ph2

[http://en.wikipedia.org/wiki/Intel_4004]

Intel 4004 registers

$^1_1$ $^1_0$ $^0_9$ $^0_8$ $^0_7$ $^0_6$ $^0_5$ $^0_4$ $^0_3$ $^0_2$ $^0_1$ $^0_0$ (bit position)

Main registers

			A	Accumulator
	R0	R1		
	R2	R3		
	R4	R5		
	R6	R7		
	R8	R9		
	R10	R11		
	R12	R13		
	R14	R15		

Program counter

PC	Program Counter

Push-down address call stack

PC1	Call level 1
PC2	Call level 2
PC3	Call level 3

Status register

	C P Z S	Flags

[http://en.wikipedia.org/wiki/Intel_4004]

Intel's 8086 Chip
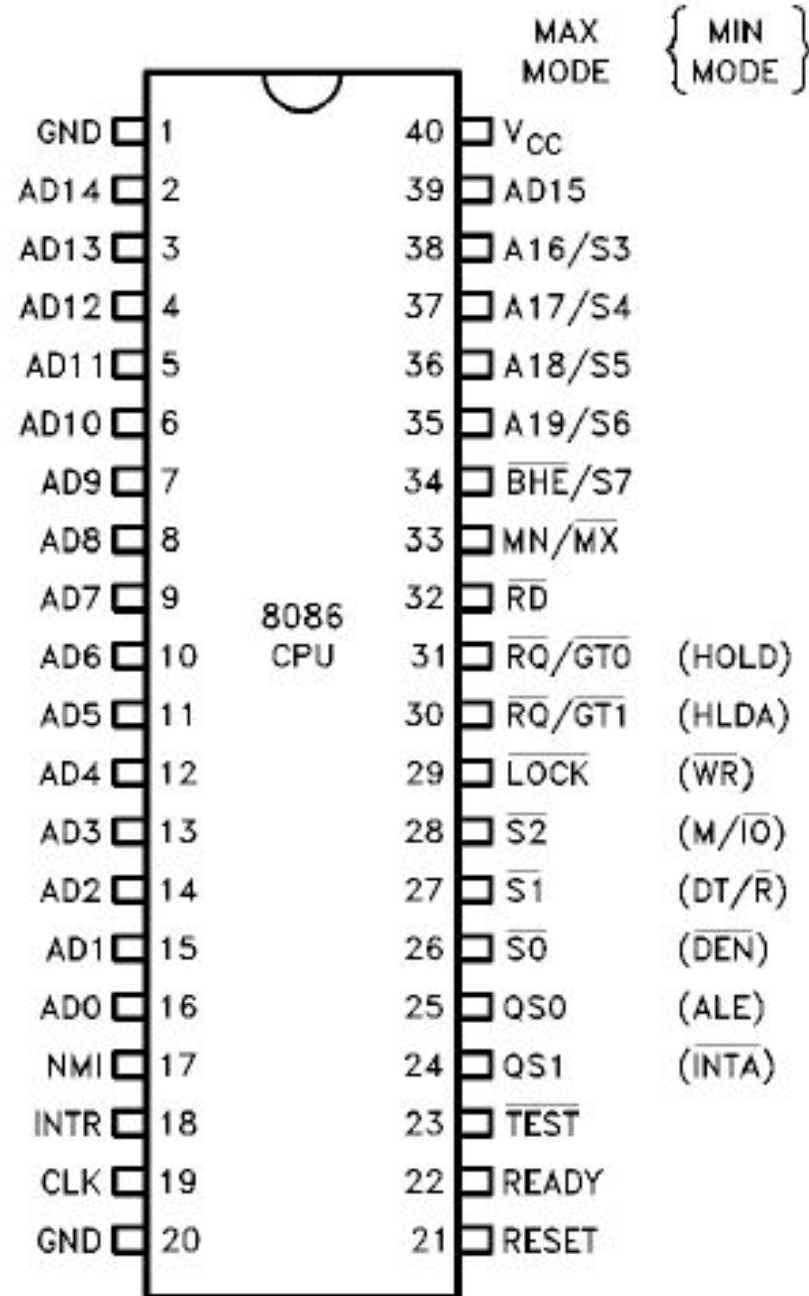
Intel 8086 registers

1_9	1_8	1_7	1_6	1_5	1_4	1_3	1_2	1_1	1_0	0_9	0_8	0_7	0_6	0_5	0_4	0_3	0_2	0_1	0_0	*(bit position)*

Main registers

	AH	AL	**AX** (primary accumulator)
	BH	BL	**BX** (base, accumulator)
	CH	CL	**CX** (counter, accumulator)
	DH	DL	**DX** (accumulator, other functions)

Index registers

0 0 0 0	SI	Source Index
0 0 0 0	DI	Destination Index
0 0 0 0	BP	Base Pointer
0 0 0 0	SP	Stack Pointer

Program counter

0 0 0 0	IP	Instruction Pointer

Segment registers

CS	0 0 0 0	Code Segment
DS	0 0 0 0	Data Segment
ES	0 0 0 0	ExtraSegment
SS	0 0 0 0	Stack Segment

Status register

-	-	-	-	O	D	I	T	S	Z	-	A	-	P	-	C	Flags

[http://en.wikipedia.org/wiki/Intel_8086]
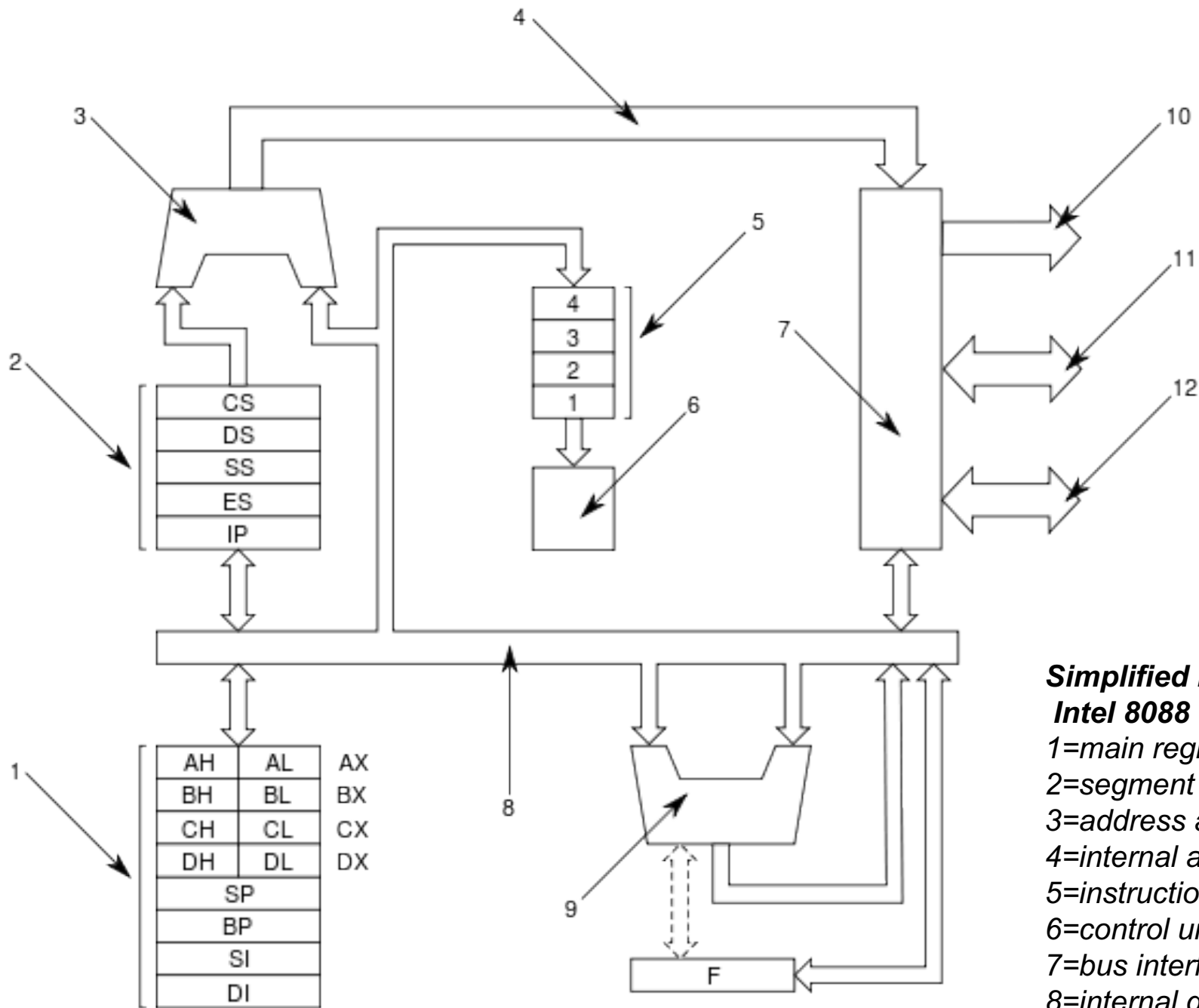
Simplified block diagram of Intel 8088 (a variant of 8086);
1=main registers;
2=segment registers and IP;
3=address adder;
4=internal address bus;
5=instruction queue;
6=control unit (very simplified!);
7=bus interface;
8=internal databus;
9=ALU;
10/11/12=external address/ data/control bus.

[http://en.wikipedia.org/wiki/Intel_8086]

Questions?

THE END