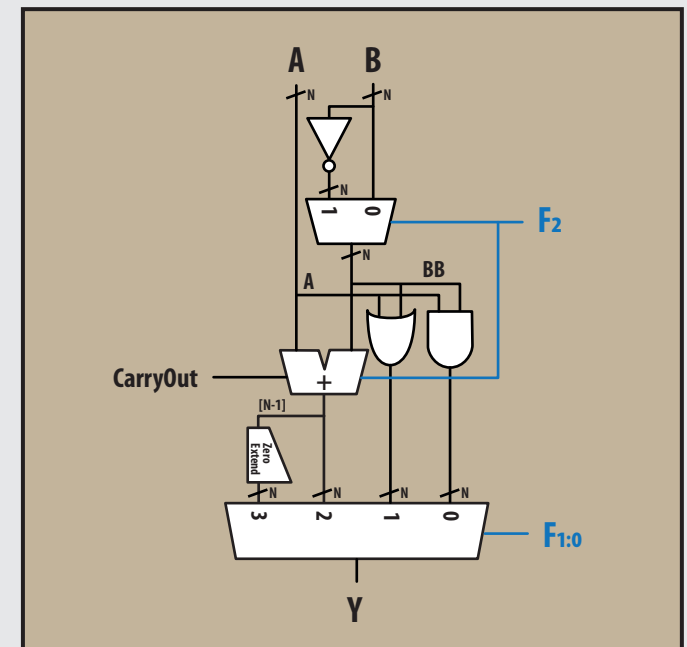


Computer Architecture

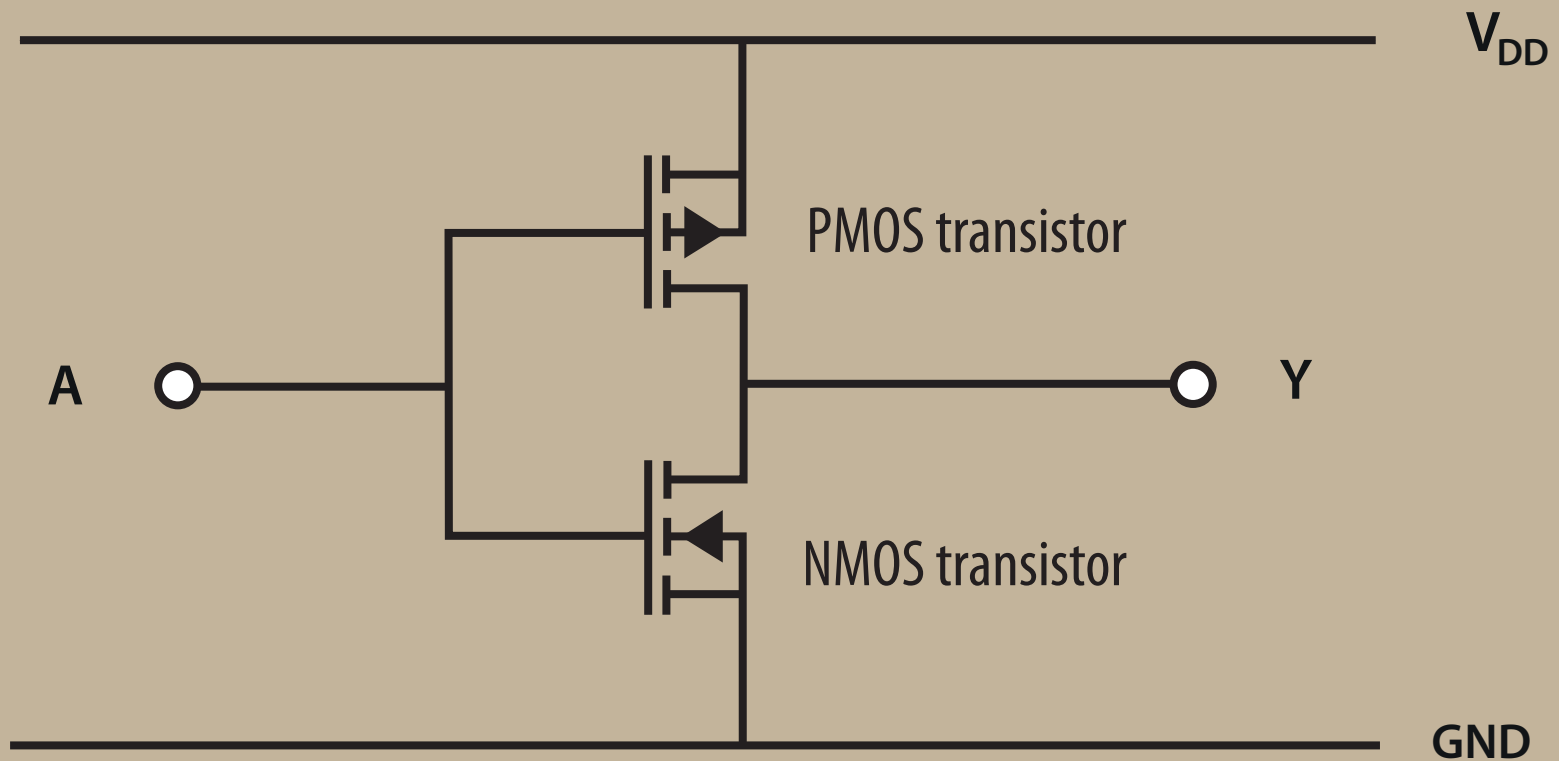
Paul Mellies

Lecture 12 : Digital Building Blocks



Where we started from...

CMOS inverter



What we have just studied...

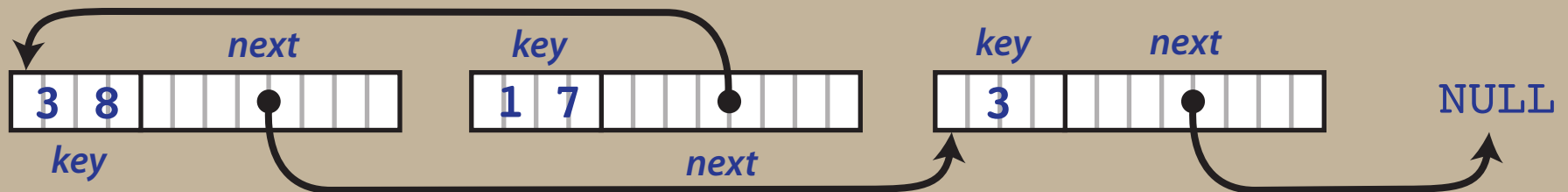
Linked lists and pointers in C

A list will be represented in the type `(struct cell *)` by:

- the `NULL` pointer if the list is empty
- a pointer to its first cell if the list is non empty

For that reason, the field `next` always represents a list: in particular, the field `next` of the last cell represents the empty list.

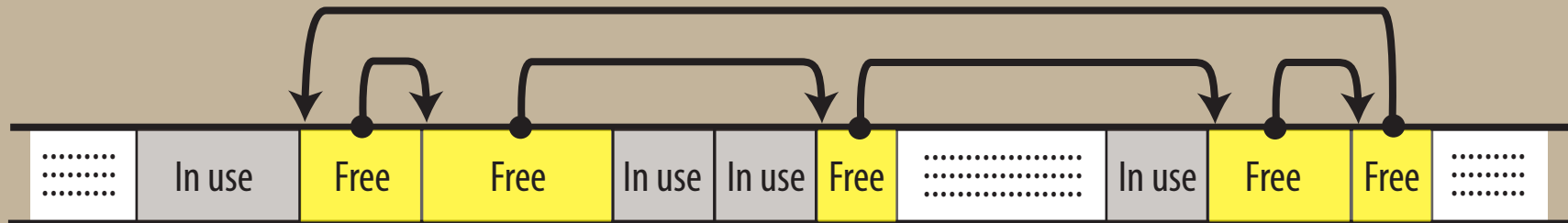
Note that the cells are connected together but they may appear anywhere in memory (a list is not an array !)



What we have just studied...

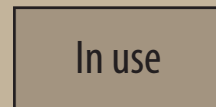
Memory allocation in C

The `malloc` algorithm



Free

Free, owned by `malloc`



In use

In use, owned by `malloc`

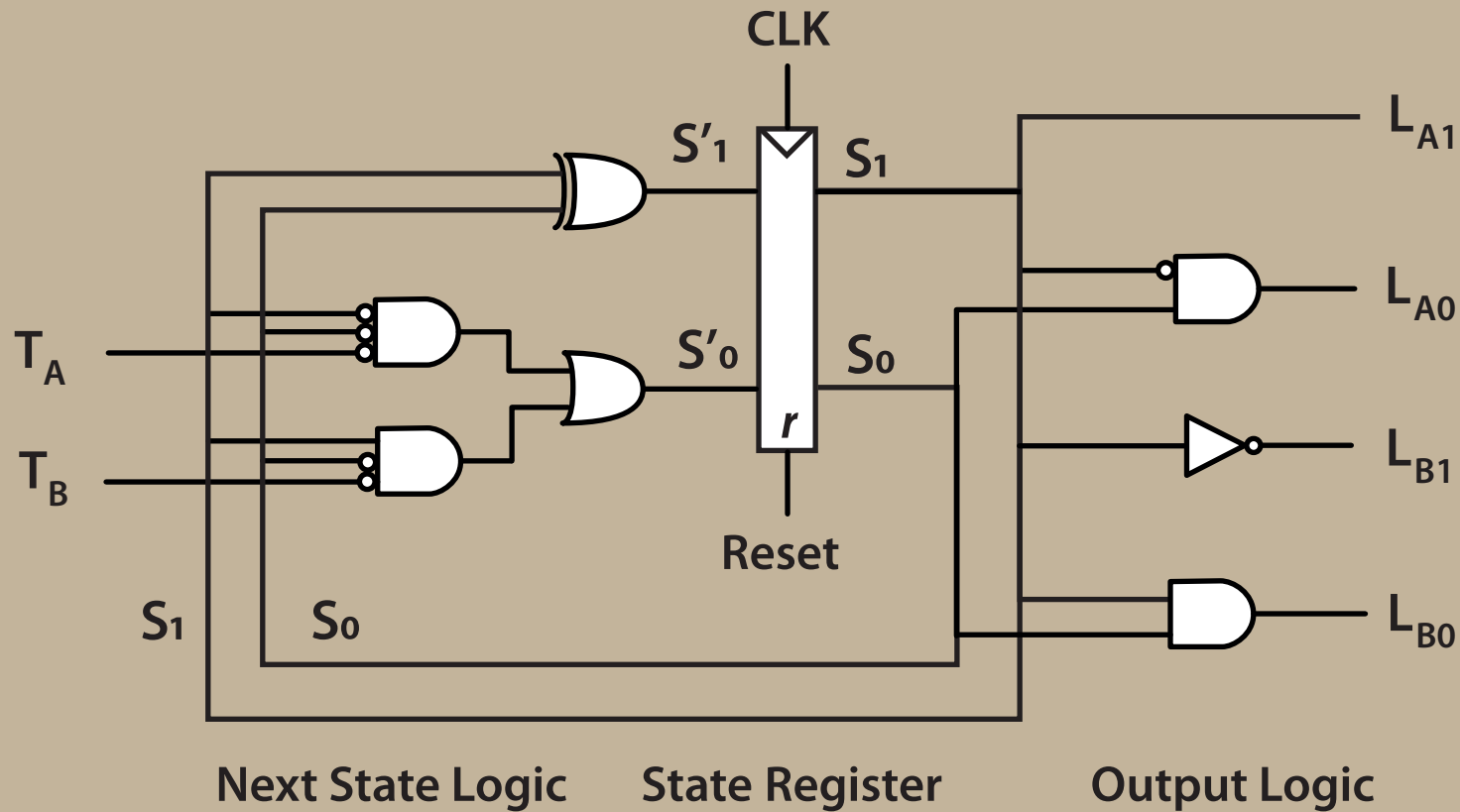


.....

Not owned by `malloc`

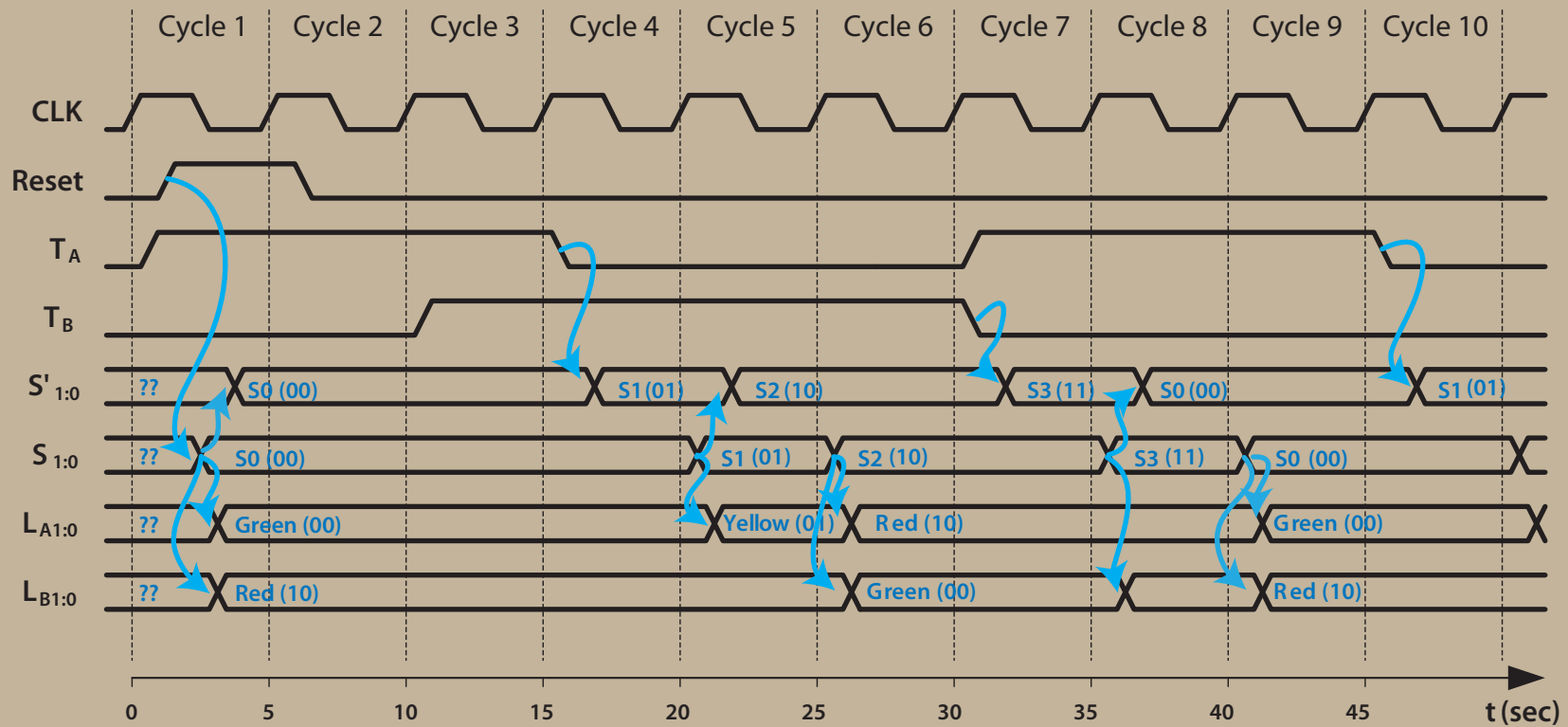
What we have just studied...

State machine circuit for traffic light controller



What we have just studied...

Timing Diagram for Traffic Light Controller



What we are aiming at...

Compilation from C to MIPS

Recursive function calls

A function which does not call other functions is called a *leaf* function.

A function which calls itself is called a *recursive* function.

A typical illustration is provided by the `factorial` function below.

High-Level Code

```
int factorial(int n)
{
    if (n <= 1)
        return 1;

    else
        return (n * factorial(n - 1));
}
```

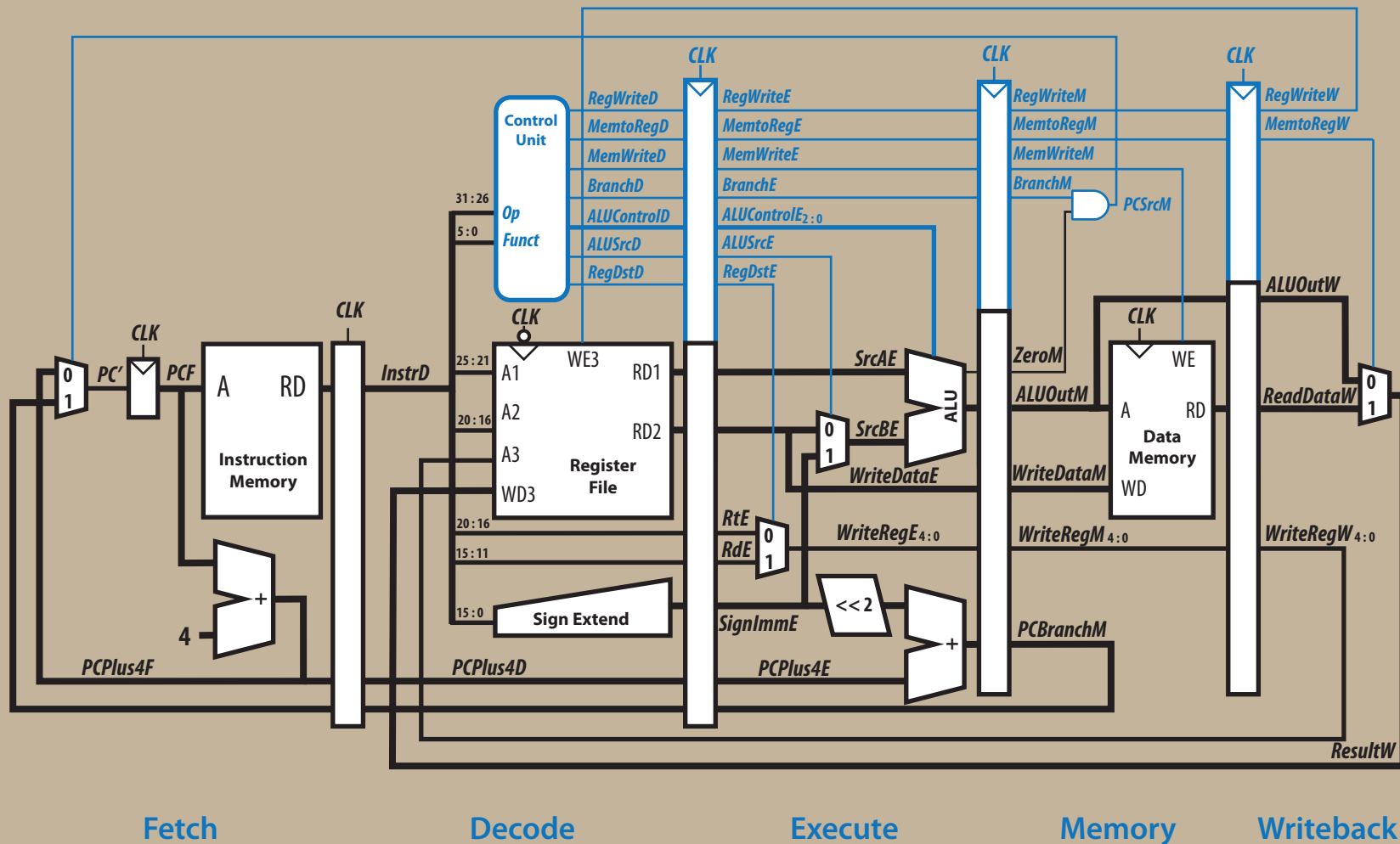
MIPS Assembly Code

```
0x90 factorial: addi $sp, $sp, -8 # make room on stack
0x94           sw  $a0, 4($sp) # store $a0
0x98           sw  $ra, 0($sp) # store $ra
0x9C           addi $t0, $0, 2 # $t0 = 2
0xA0           slt  $t0, $a0, $t0 # n <= 1 ?
0xA4           beq  $t0, $0, else # no: goto else
0xA8           addi $v0, $0, 1 # yes: return 1
0xAC           addi $sp, $sp, 8 # restore $sp
0xB0           jr   $ra # return
0xB4           else: addi $a0, $a0, -1 # n = n - 1
0xB8           jal  factorial # recursive call
0xBC           lw   $ra, 0($sp) # restore $ra
0xC0           lw   $a0, 4($sp) # restore $a0
0xC4           addi $sp, $sp, 8 # restore $sp
0xC8           mul  $v0, $a0, $v0 # n * factorial(n-1)
0xCC           jr   $ra # return
```

Example 6.27 in H & H

What we are aiming at...

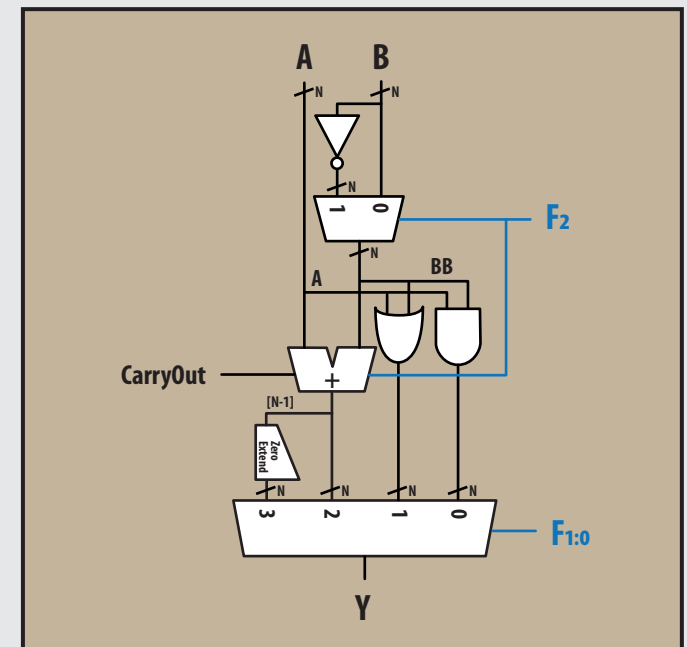
Pipelined MIPS microprocessor



Computer Architecture

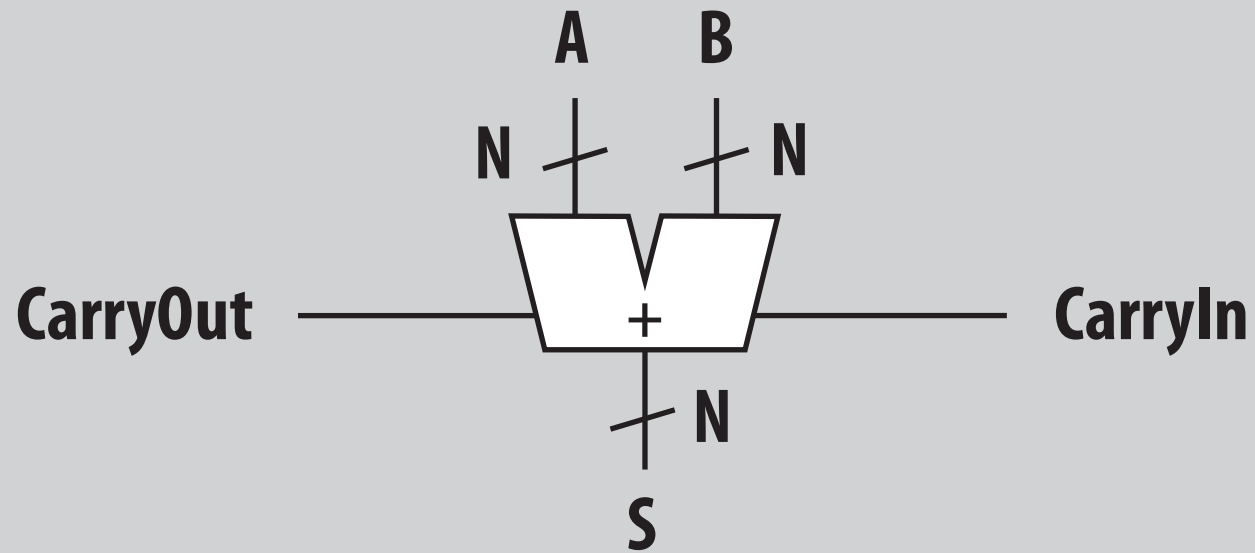
Paul Mellies

Lecture 14 : Digital Building Blocks



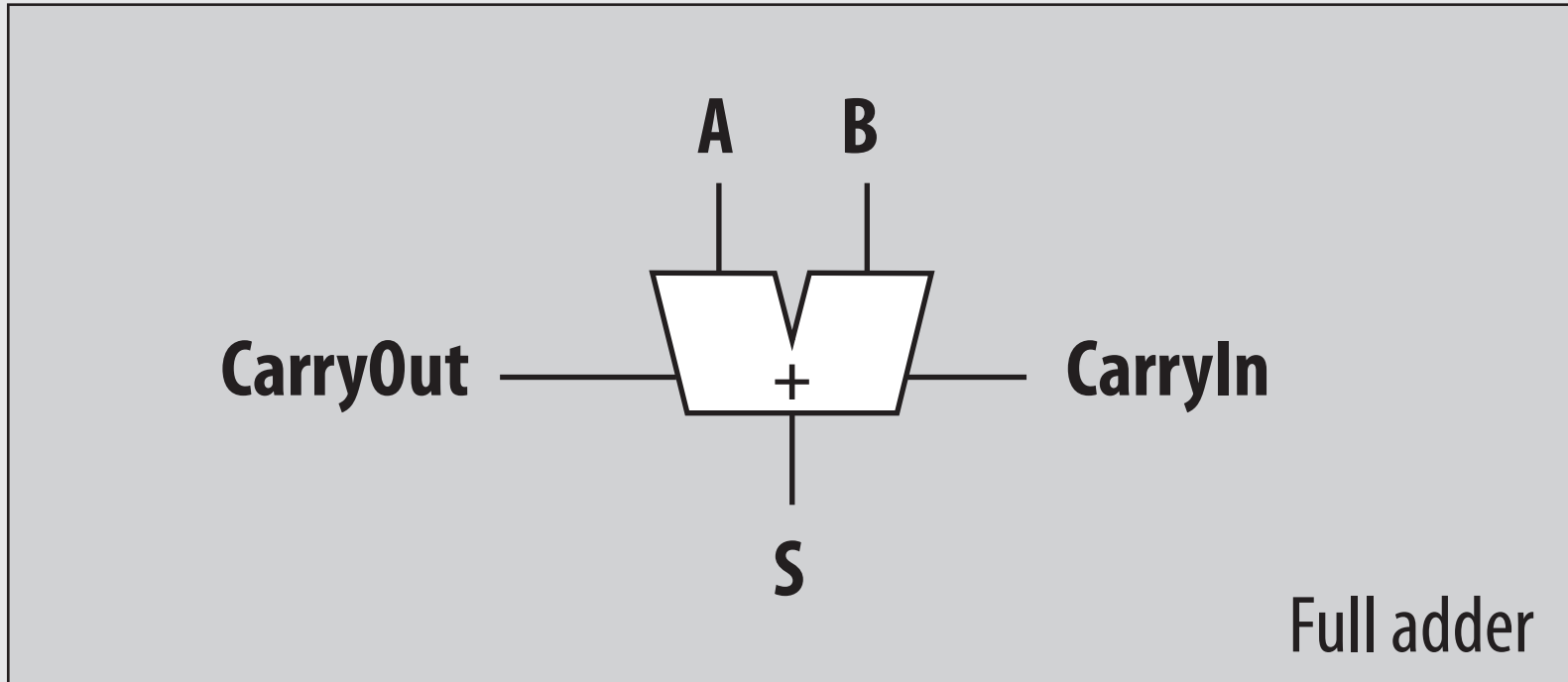
Arithmetic Circuits

Carry Propagate Adder



Carry Propagate Adder

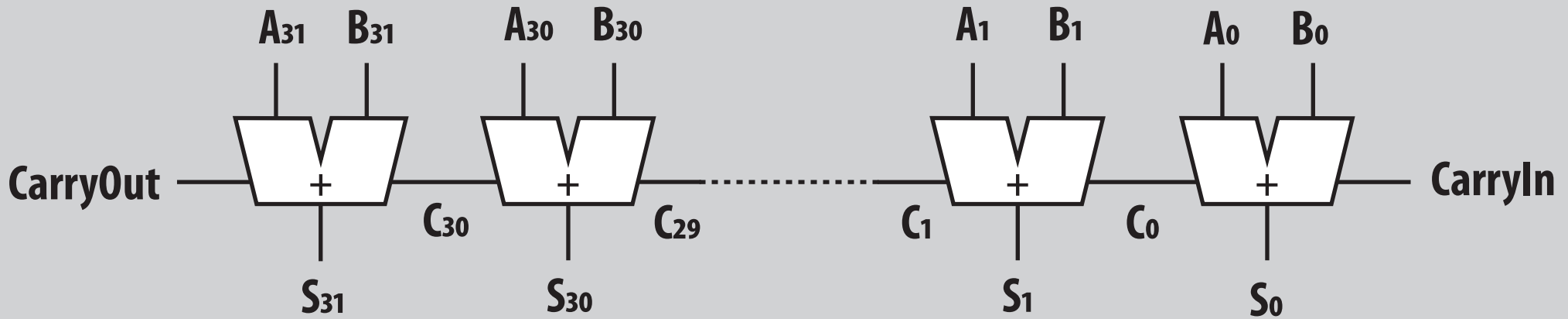
Full Adder



$$S = A \oplus B \oplus \text{CarryIn}$$

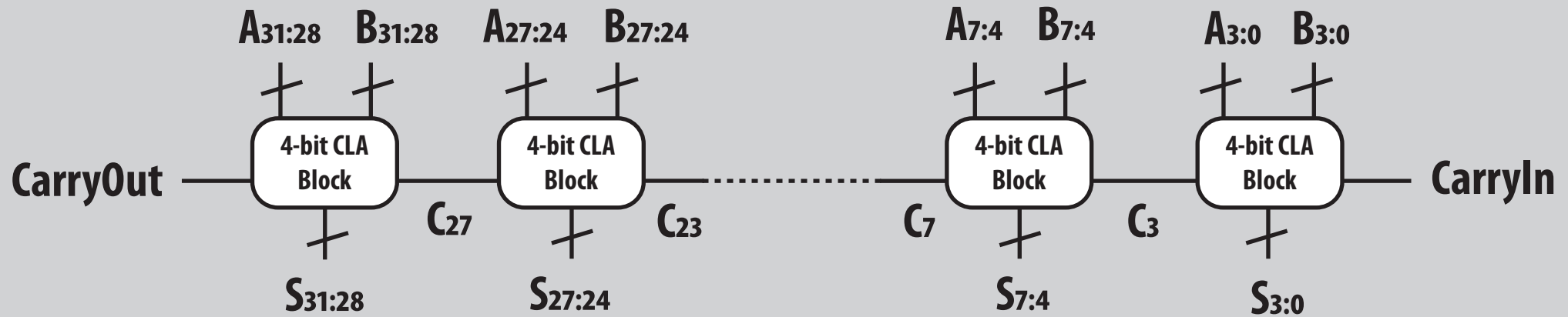
$$\text{CarryOut} = A.B + A.\text{CarryIn} + B.\text{CarryIn}$$

Ripple Carry Adder



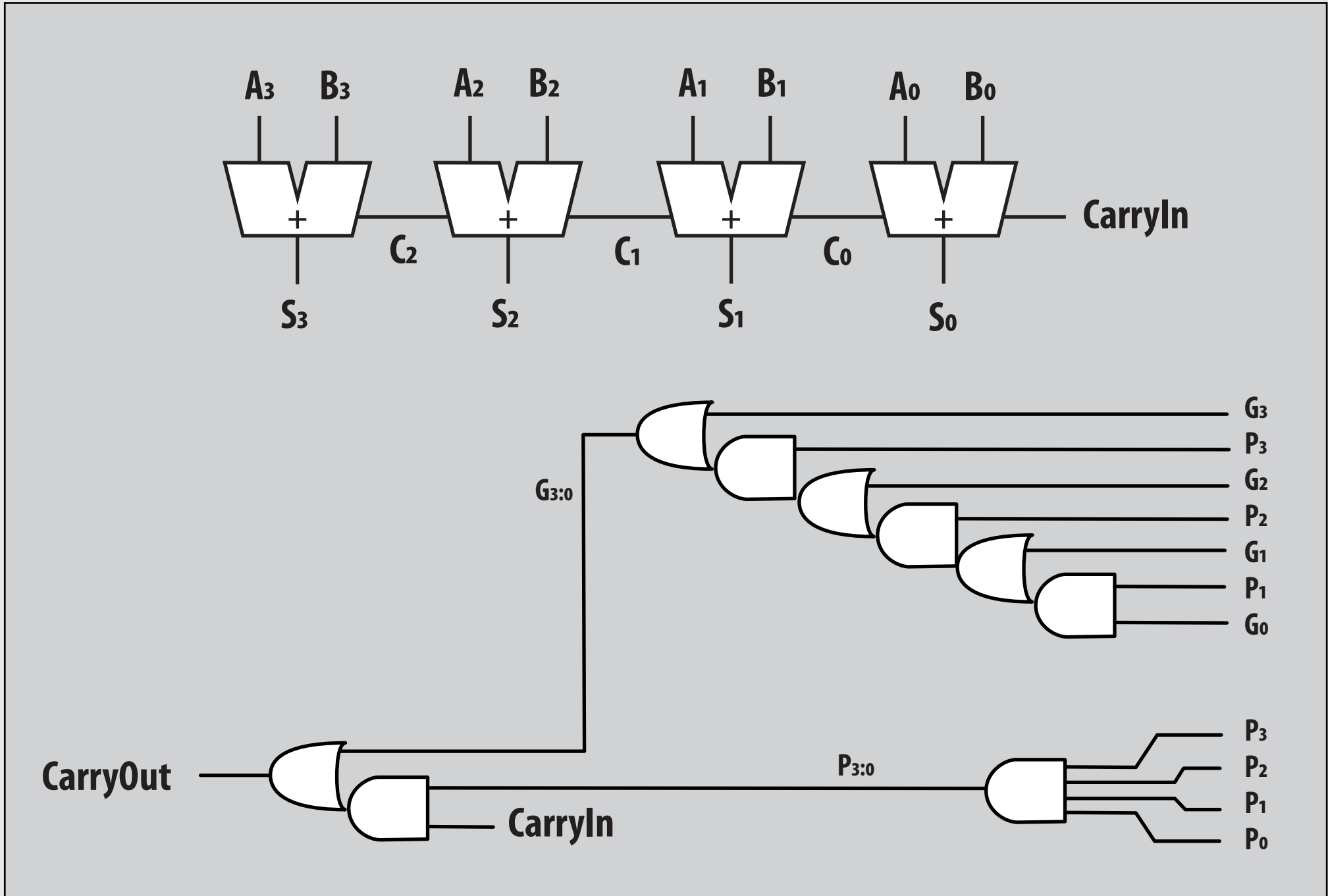
32-bit ripple carry adder

Carry Lookahead Adder

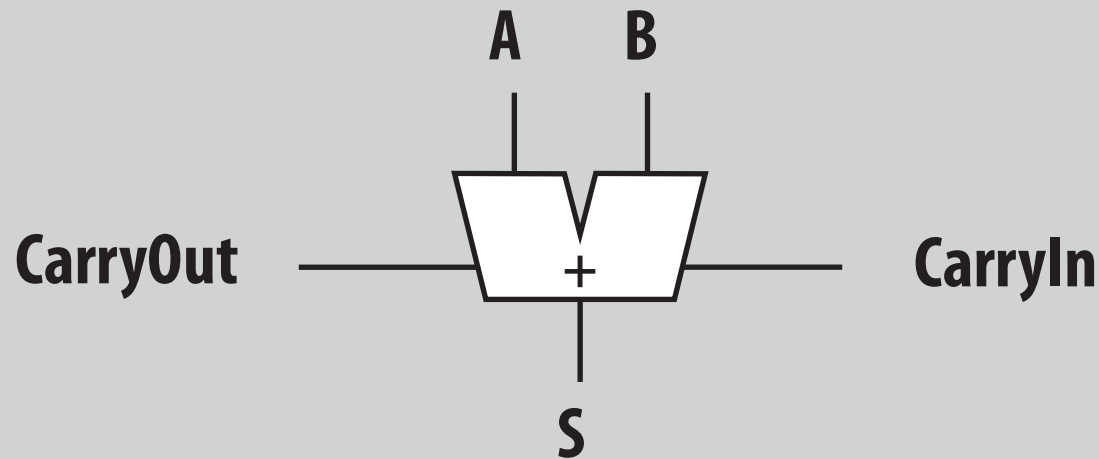


32-bit carry lookahead adder

4-bit CLA Block



Generate (G) and Propagate (P) signals



$$\mathbf{G = A \cdot B}$$

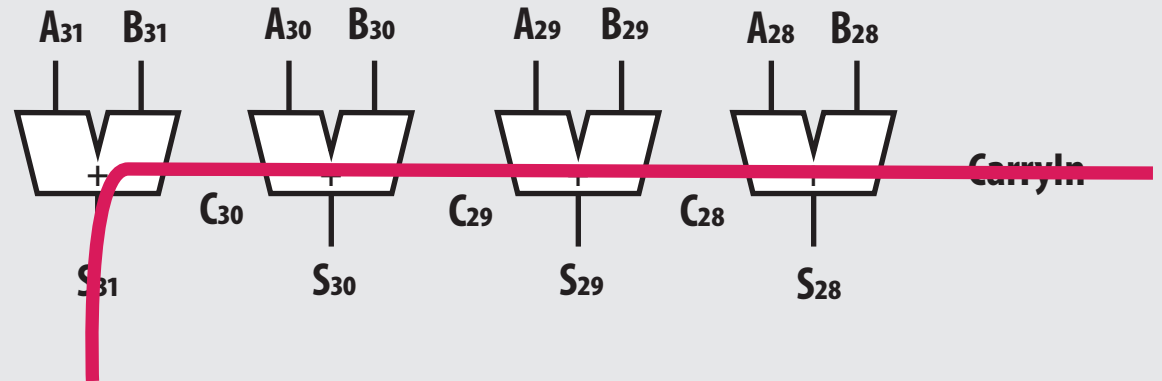
$$\mathbf{P = A + B}$$

G = 1 when the adder *generates* a CarryOut whatever the value of CarryIn

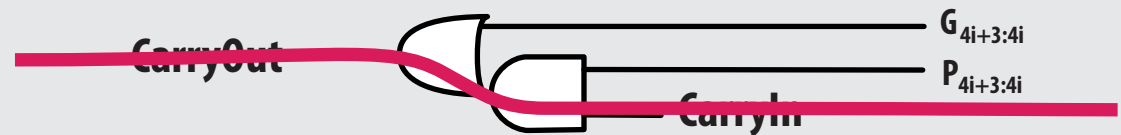
P = 1 when the adder *propagates* the CarryIn to the CarryOut

Critical path of the Carry Lookahead Adder

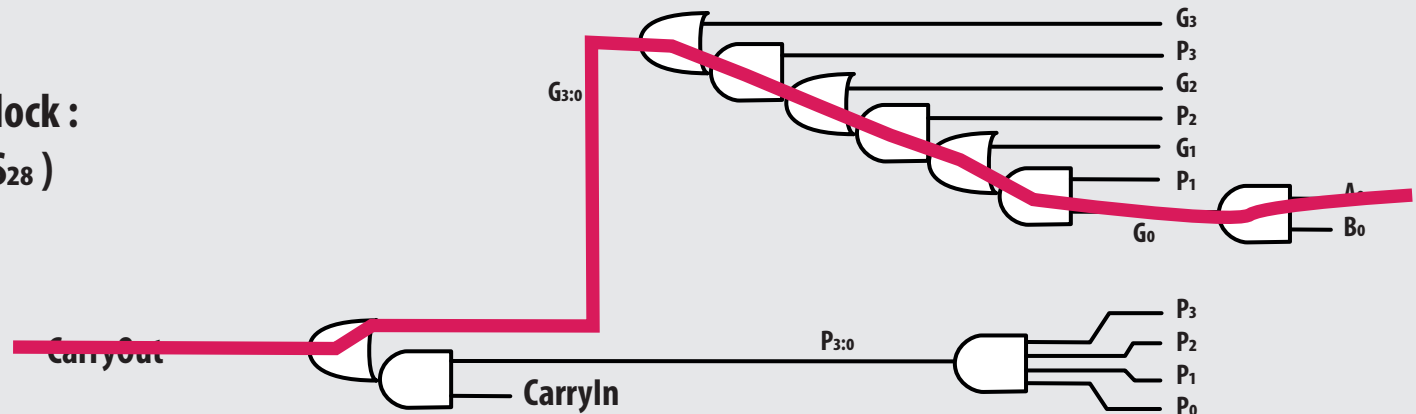
On the lefthand side 4-bit CLA block :
(most significant bits from S_{31} to S_{28})



On the six internal 4-bit CLA blocks :
 $i = 6, 5, 4, 3, 2, 1$



On the righthand side 4-bit CLA block :
(less significant bits from S_{31} to S_{28})



Propagation Delay

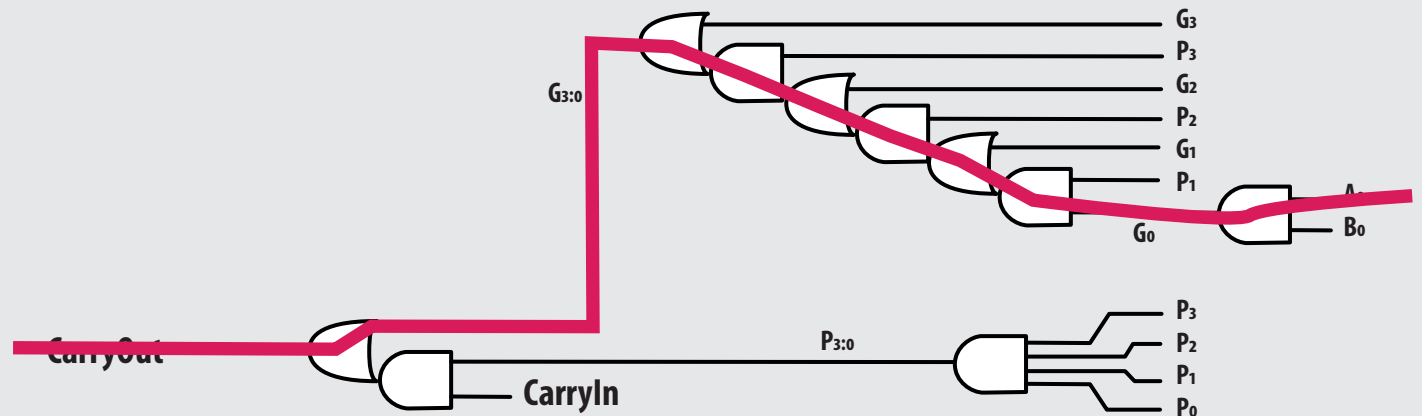
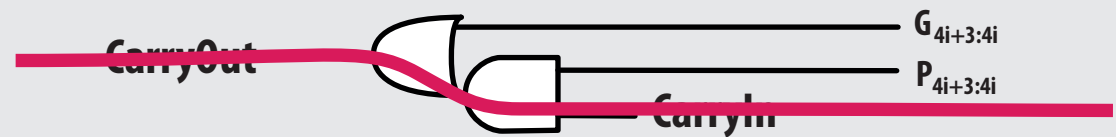
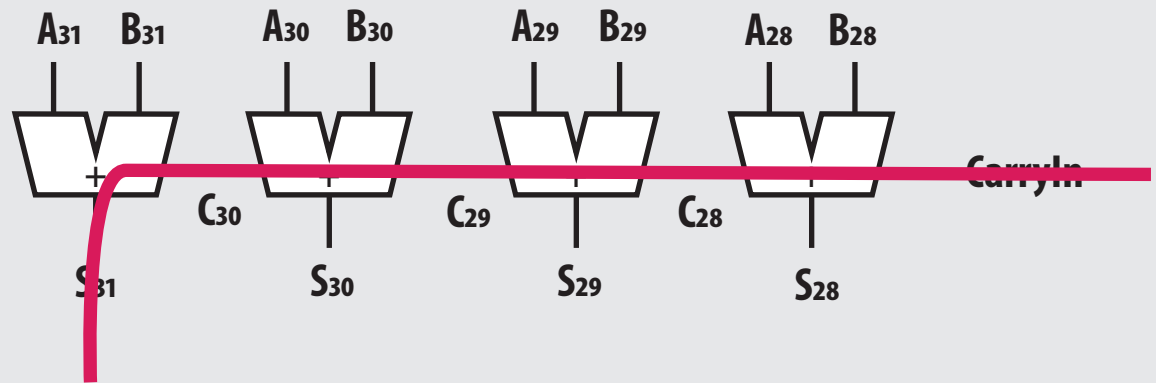
$$4 t_{\text{FULL_ADDER}}$$

+

$$6 \times (t_{\text{AND}} + t_{\text{OR}})$$

+

$$4 t_{\text{AND}} + 4 t_{\text{OR}}$$



$$t_{\text{CLA}} = 10 t_{\text{AND}} + 10 t_{\text{OR}} + 4 t_{\text{FULL_ADDER}}$$

Exercise

(H&H Example 5.1)

Compare the delays of a 32-bit ripple-carry adder and a 32-bit carry-lookahead adder with 4-bit blocks.

Assume that each two-input gate delay is 100 ps and that a full adder delay is 300 ps.

Solution

(H&H Example 5.1)

Compare the delays of a 32-bit ripple-carry adder and a 32-bit carry-lookahead adder with 4-bit blocks.

Assume that each two-input gate delay is 100 ps and that a full adder delay is 300 ps.

32-bit ripple-carry adder:

$$32 \times 300 \text{ ps} = 9\,600 \text{ ps} = 9.6 \text{ ns}$$

32-bit carry-lookahead adder:

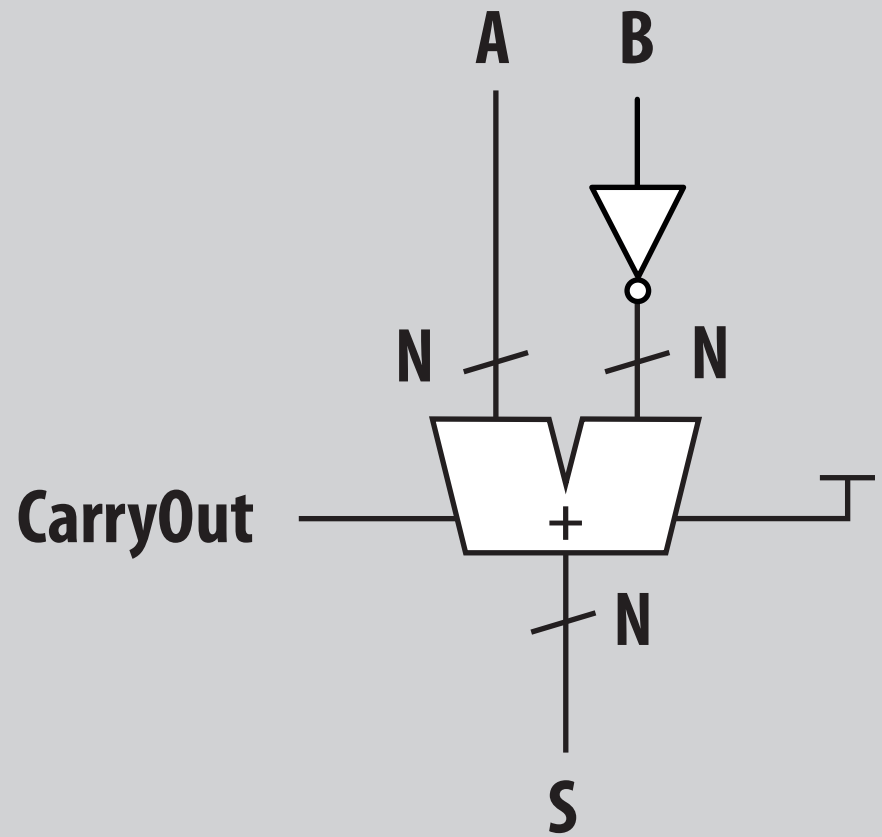
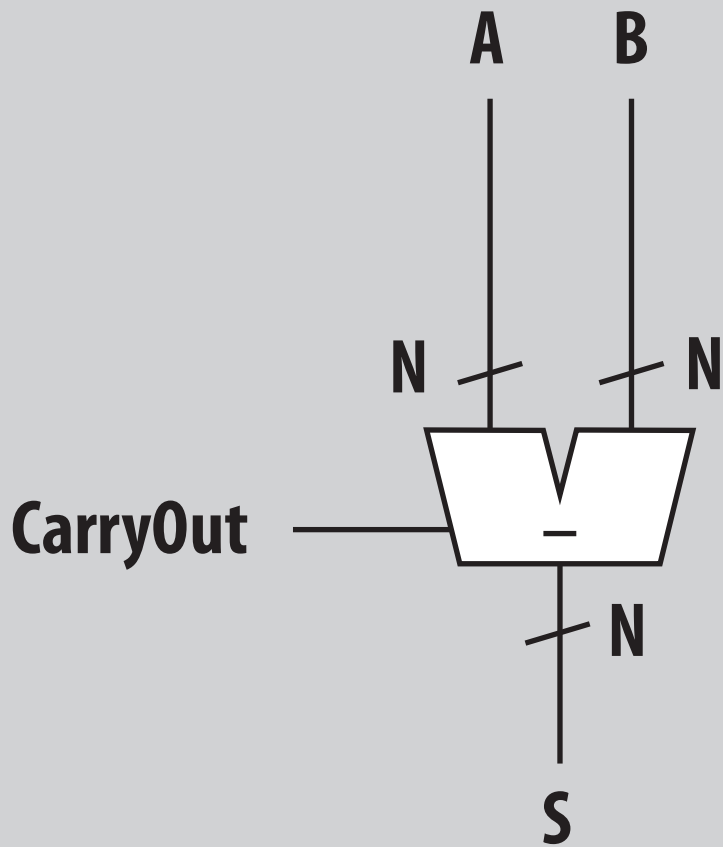
$$10 \times 100 \text{ ps} + 10 \times 100 \text{ ps} + 4 \times 300 \text{ ps} = 3\,200 \text{ ps} = 3.2 \text{ ns}$$

Carry Lookahead Adder (summary)

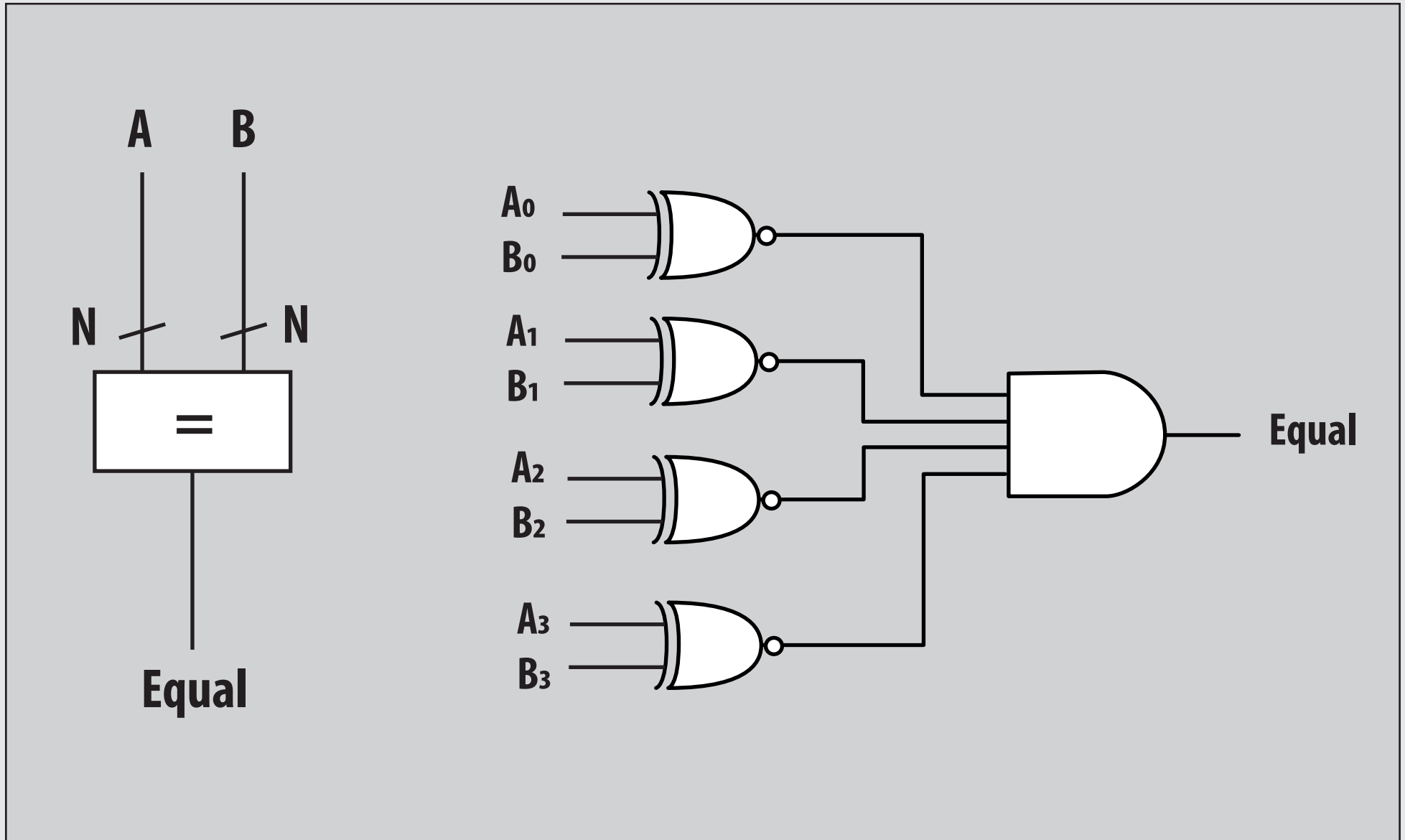
The 32-bit Carry Lookahead Adder is faster than the 32-bit Ripple Carry Adder because :

1. In each 4-bit block, the generate and propagate signals do not depend on the input signal **CarryIn**.
2. for that reason, the two signals can be computed *independently* and *in parallel* by each 4-bit block.
3. once a 4-bit block « knows » the values of the two signals **P** and **G**, it can generate and/or transmit carries much faster.
4. for that reason, each 4-bit block « receives » the signal **CarryIn** faster than in the Ripple Carry Adder circuit.

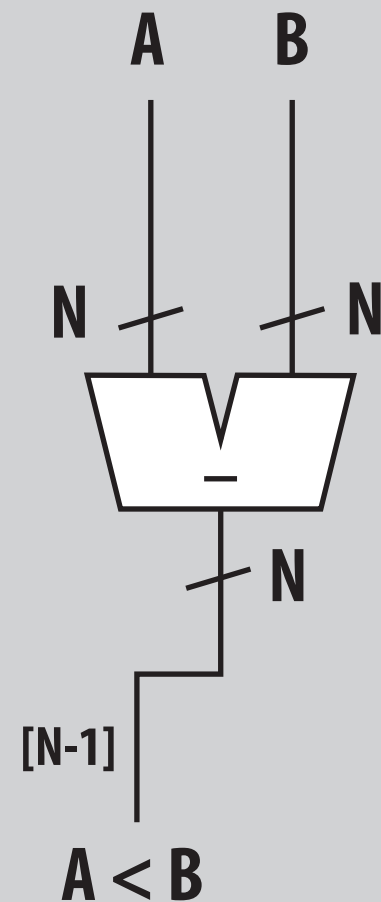
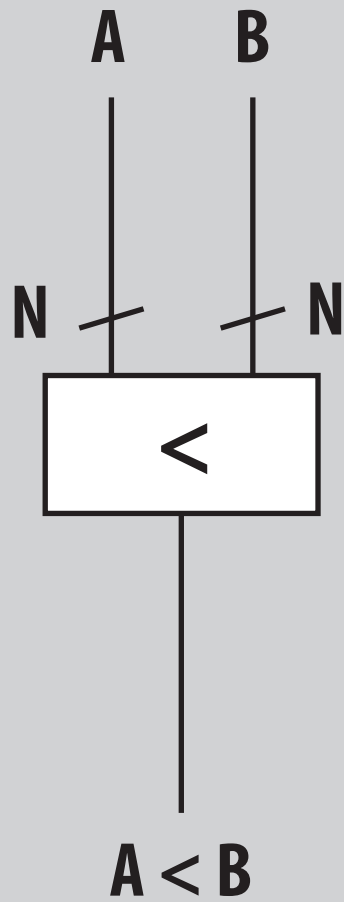
Subtraction



Equality Operator

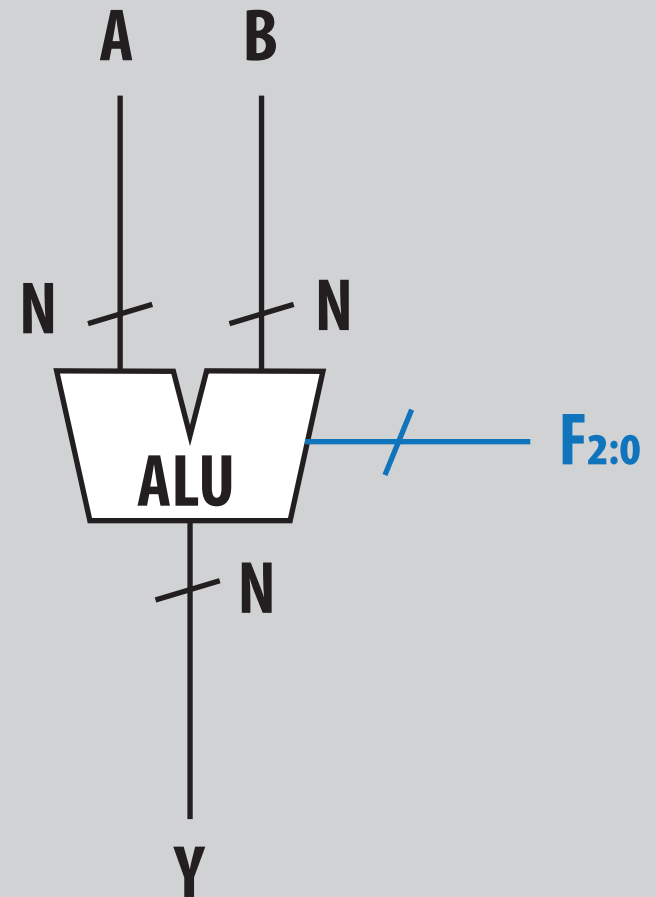


Magnitude Operator



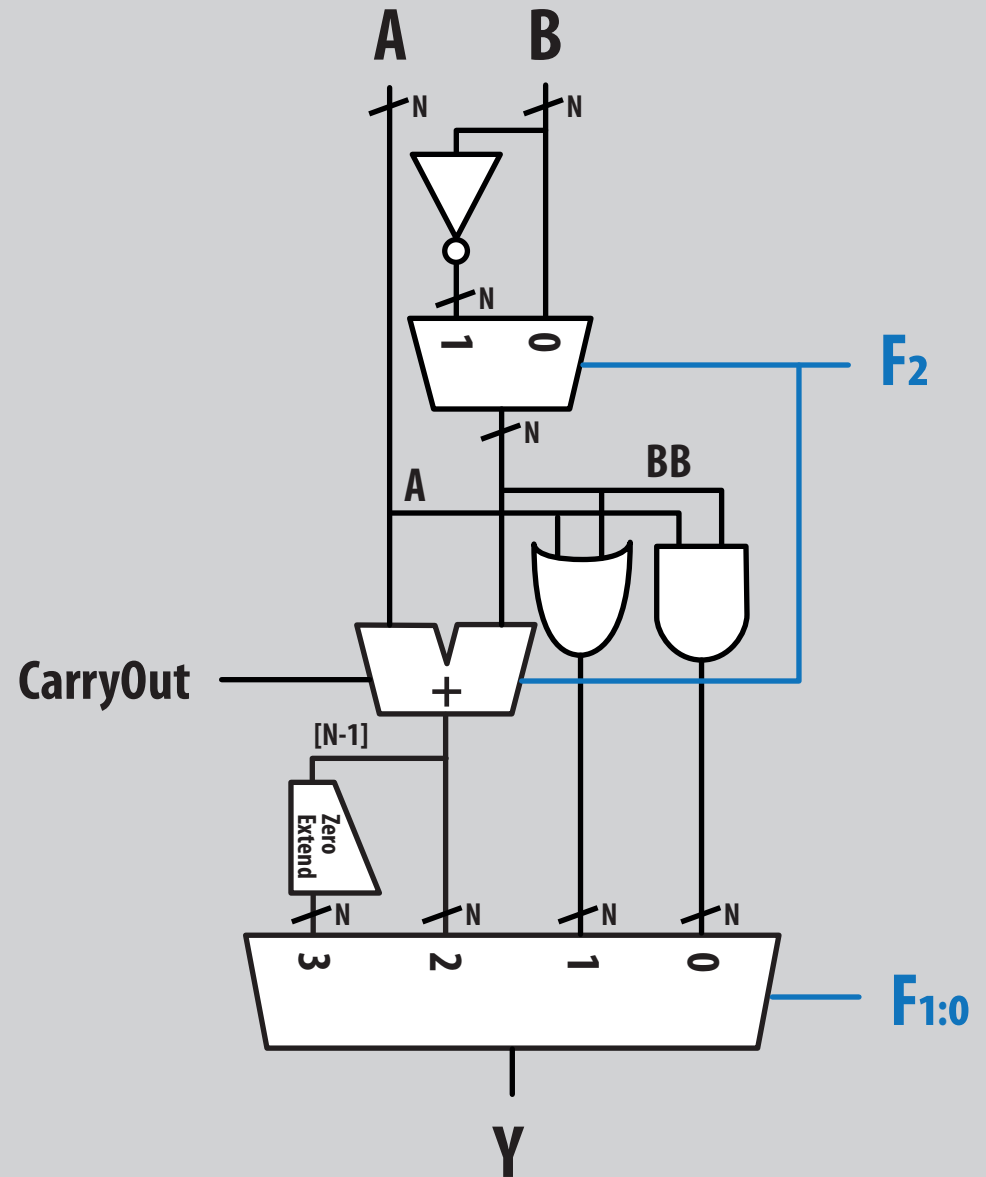
Arithmetic Logical Unit (ALU)

F_{2:0}	Function
000	A AND B
001	A OR B
010	A + B
011	not used
100	A AND \bar{B}
101	A OR \bar{B}
110	A - B
111	SLT

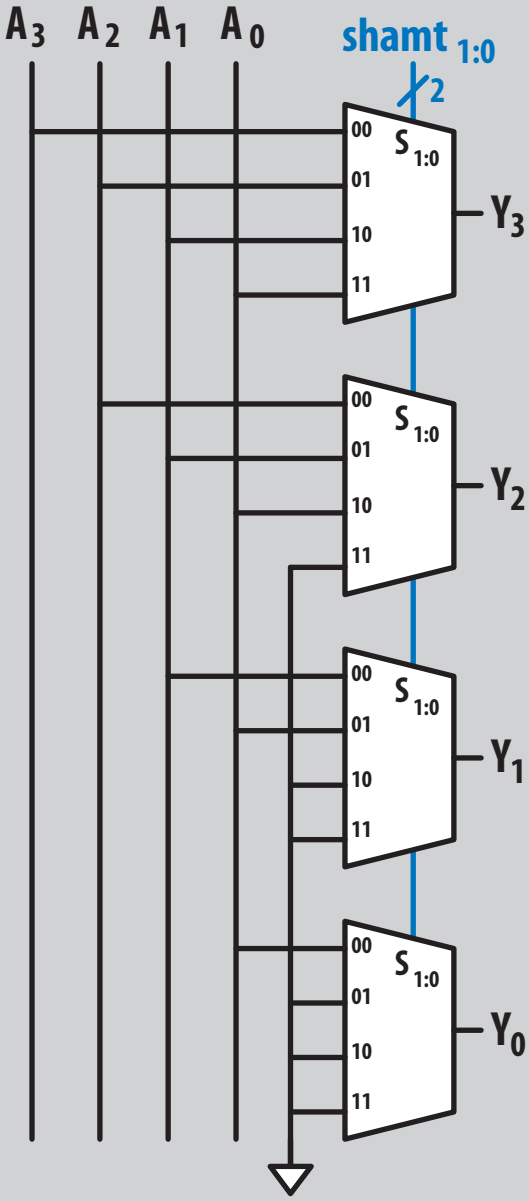
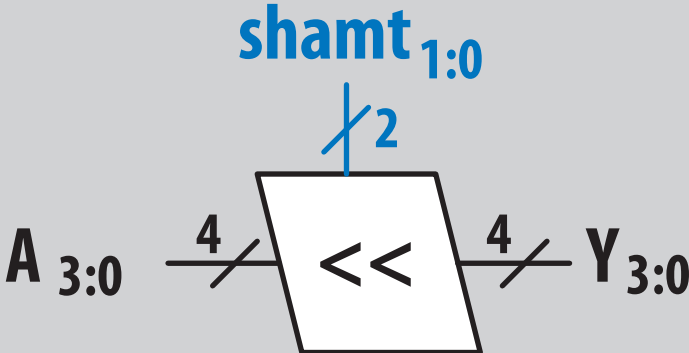


Arithmetic Logical Unit (ALU)

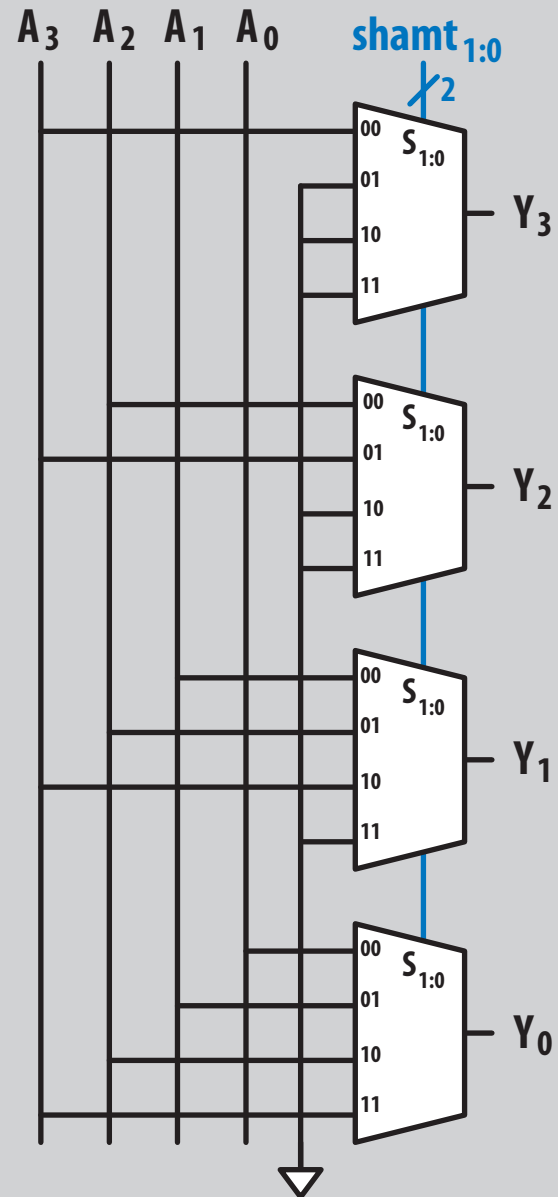
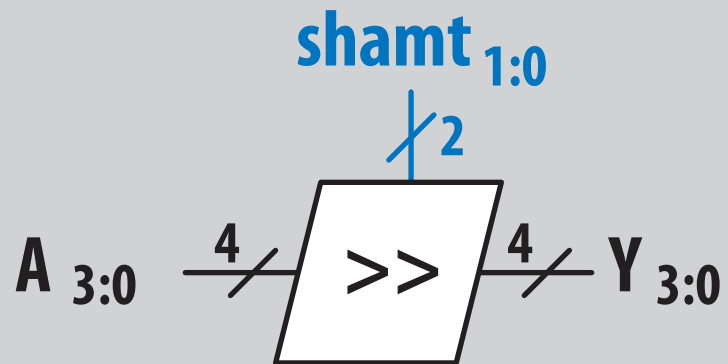
$F_{2:0}$	Function
000	A AND B
001	A OR B
010	A + B
011	not used
100	A AND \bar{B}
101	A OR \bar{B}
110	A - B
111	SLT



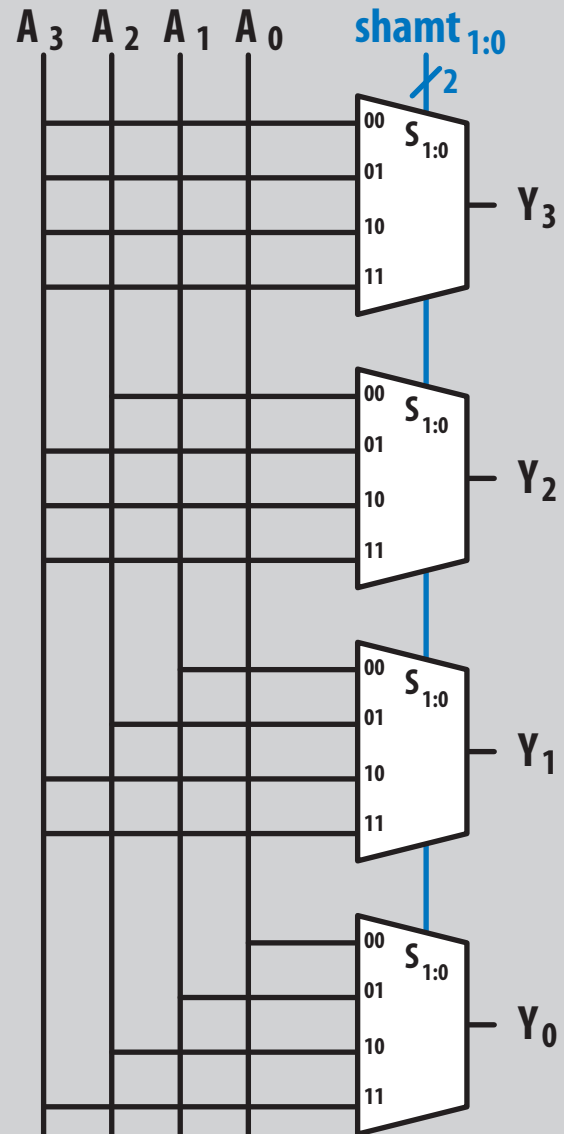
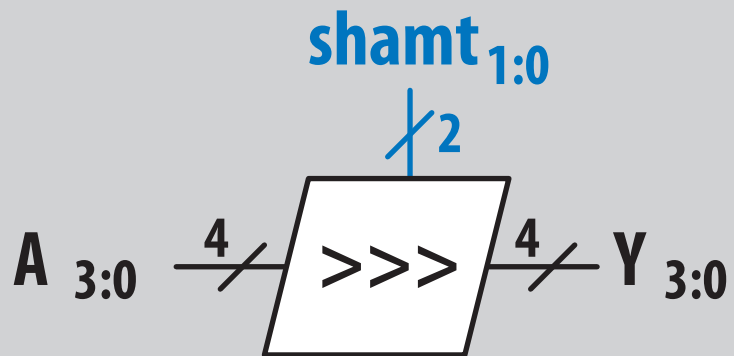
Shift Left



Shift Right



Arithmetic Shift Right



Multiplication

				A_3	A_2	A_1	A_0	
			\times	B_3	B_2	B_1	B_0	
				A_3B_0	A_2B_0	A_1B_0	A_0B_0	
			A_3B_1	A_2B_1	A_1B_1	A_0B_1		
		A_3B_2	A_2B_2	A_1B_2	A_0B_2			
$+$	A_3B_3	A_2B_3	A_1B_3	A_0B_3				
	P_7	P_6	P_5	P_4	P_3	P_2	P_1	P_0

The multiplication algorithm performed sequentially

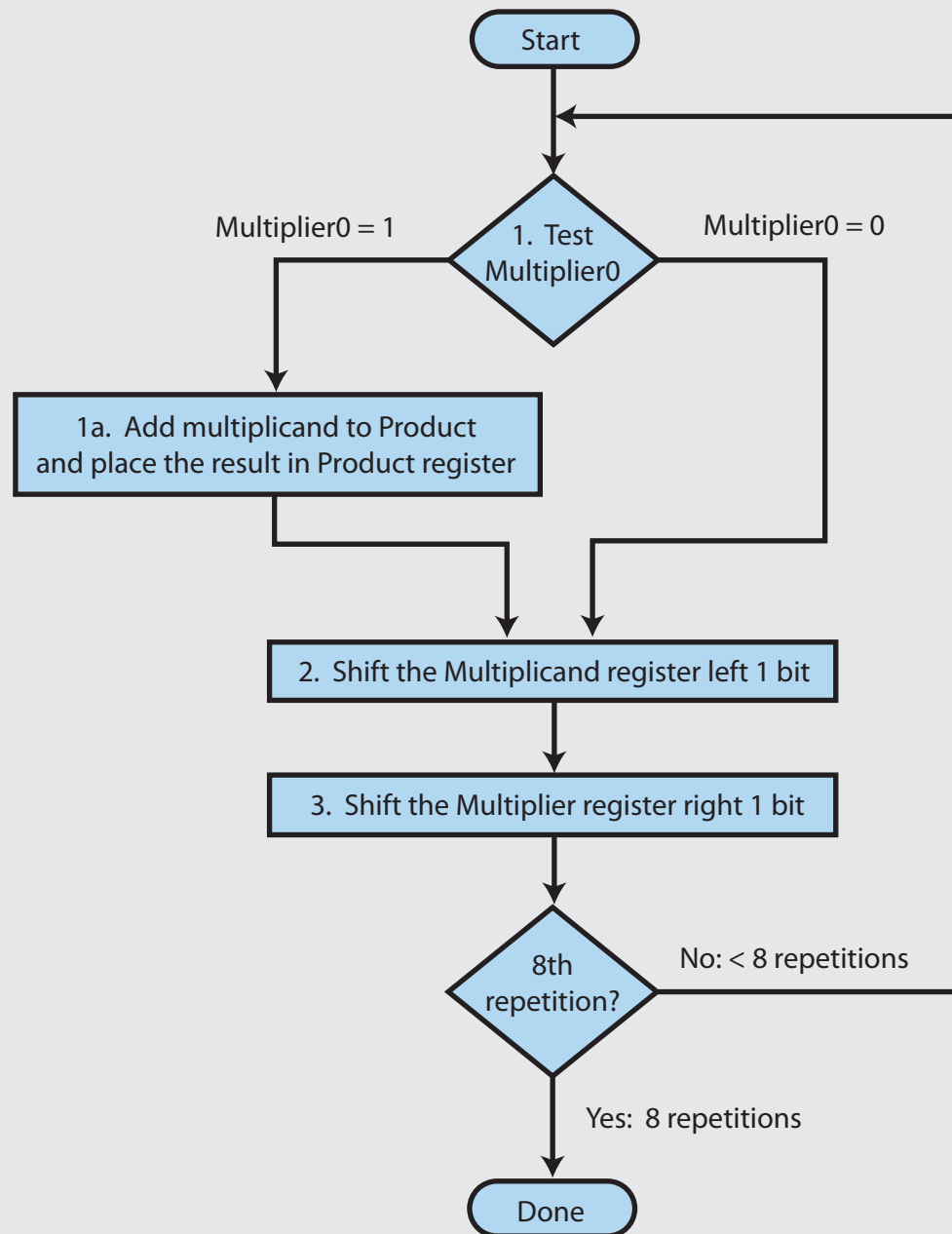
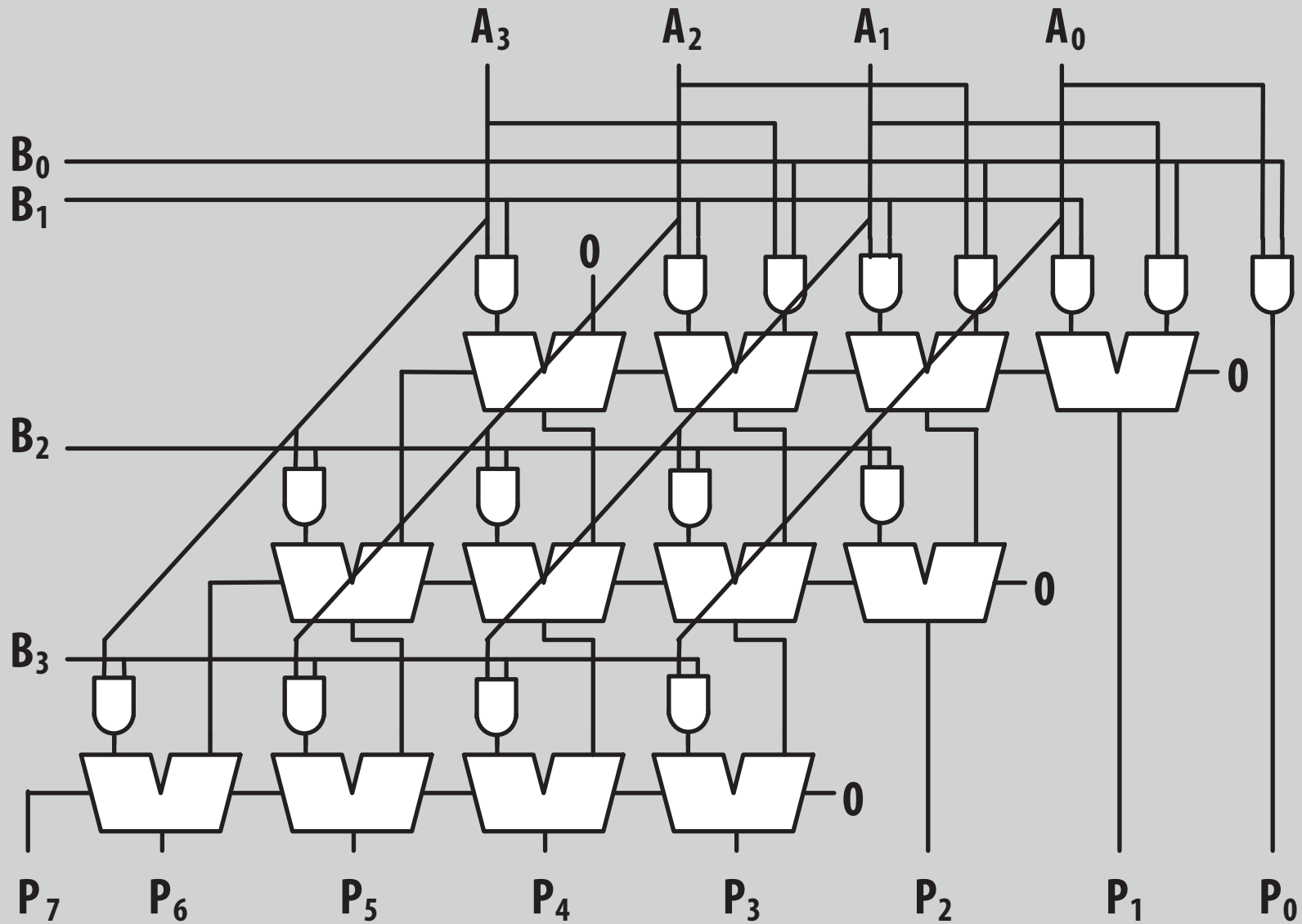


Figure 3.4 in
Patterson and Hennessy

Multiplication



Division Algorithm $A \div B$ for N-bit unsigned numbers

```
R' = 0
for i = N-1 to 0
  R = { R' << 1, Ai } // shift R' to the left
                        // with Ai as least significant bit

  D = R - B
  if D < 0 then Qi = 0, R' = R // R < B
  else          Qi = 1, R' = D // R >= B
R = R'
```

The partial remainder R is initialized to 0

The most significant bit of the dividend A becomes the least significant bit of R

The divisor B is repeatedly subtracted from this partial remainder to determine whether it fits :

- if the difference $R - B$ is negative, then the quotient bit Q_i is 0 and the difference is discarded.
- if the difference $R - B$ is positive, then the quotient bit Q_i is 1 and the partial remainder is updated to be the difference $R - B$.

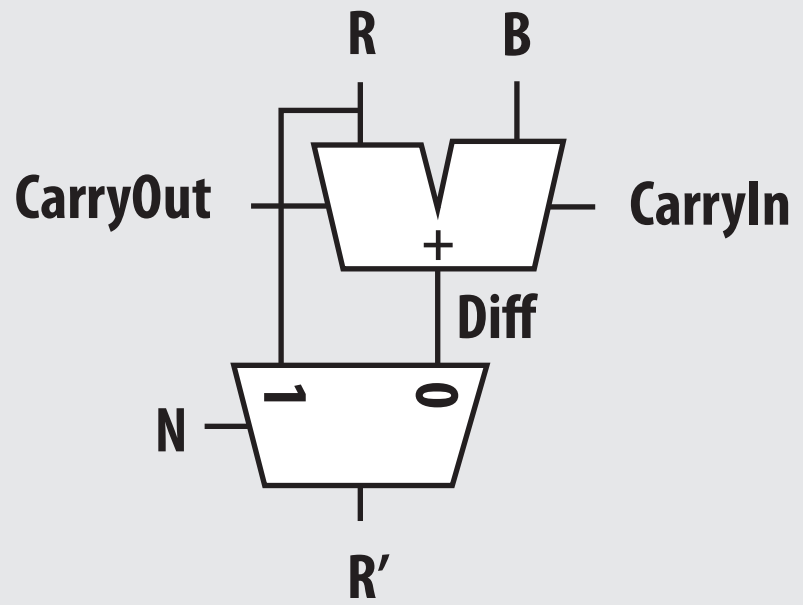
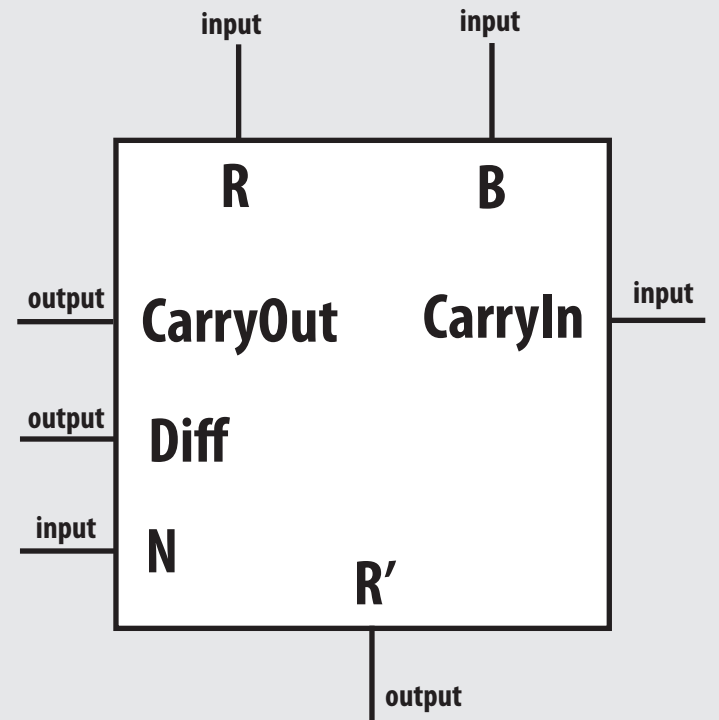
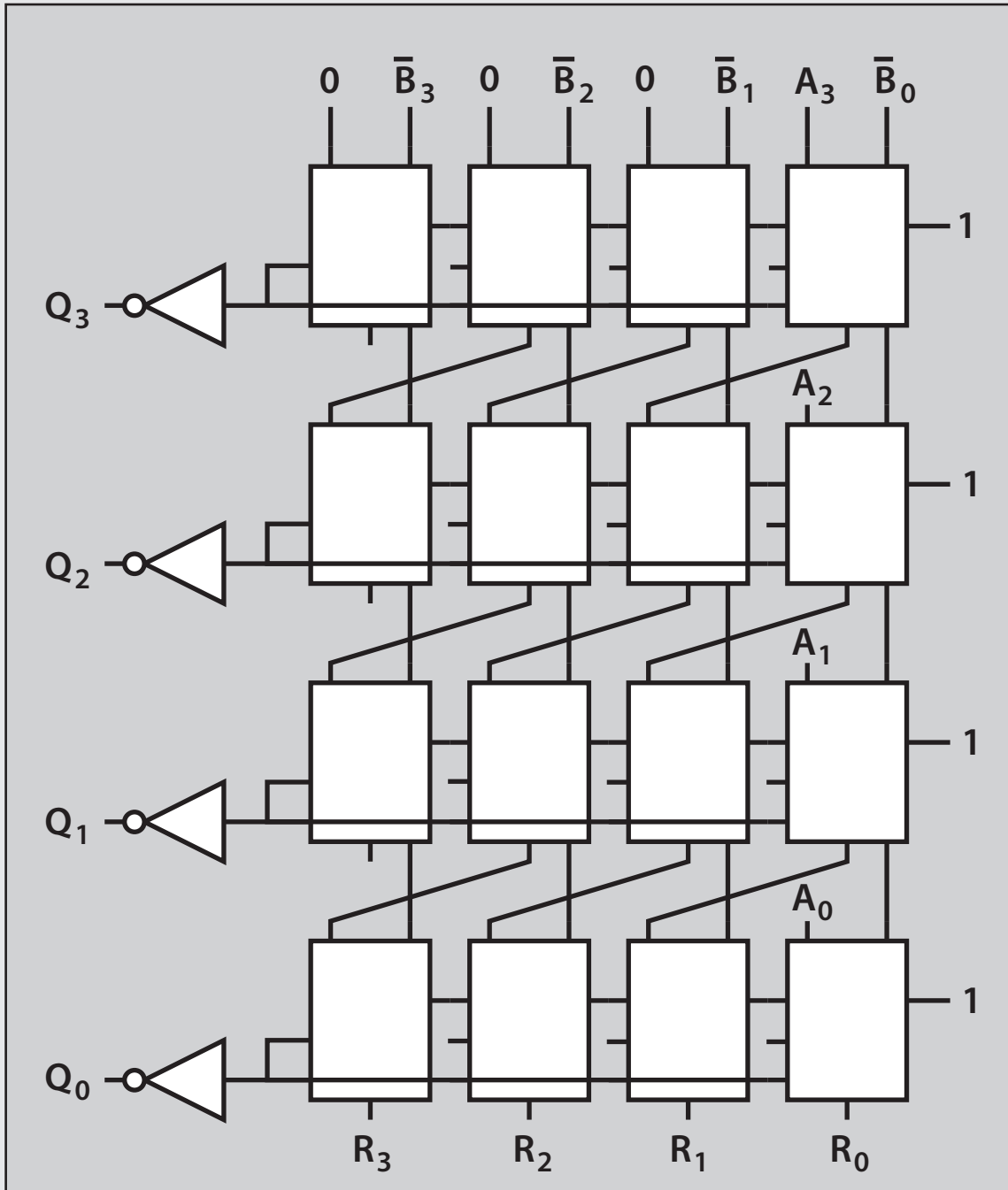
Exercise :

Use the algorithm to compute $A \div B$ for :

1. $A = 83$ and $B = 5$
2. $A = 97$ and $B = 23$

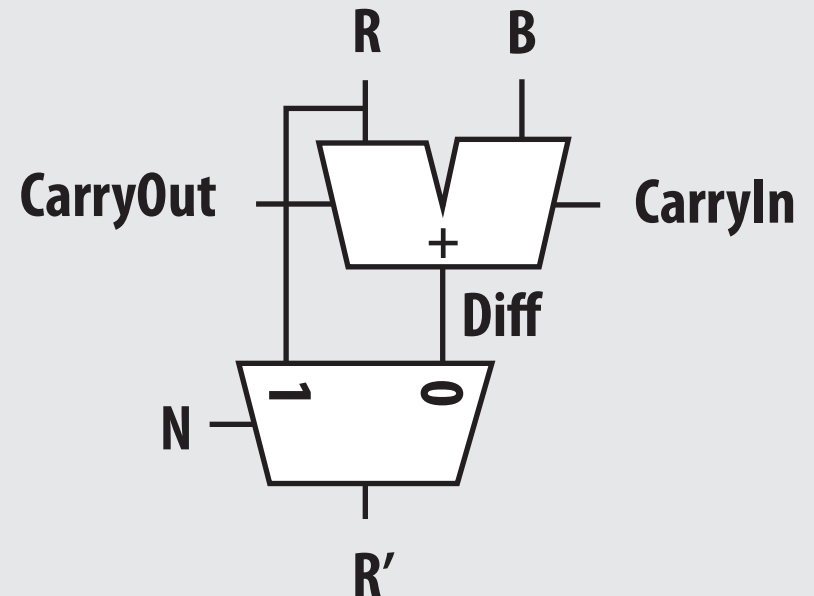
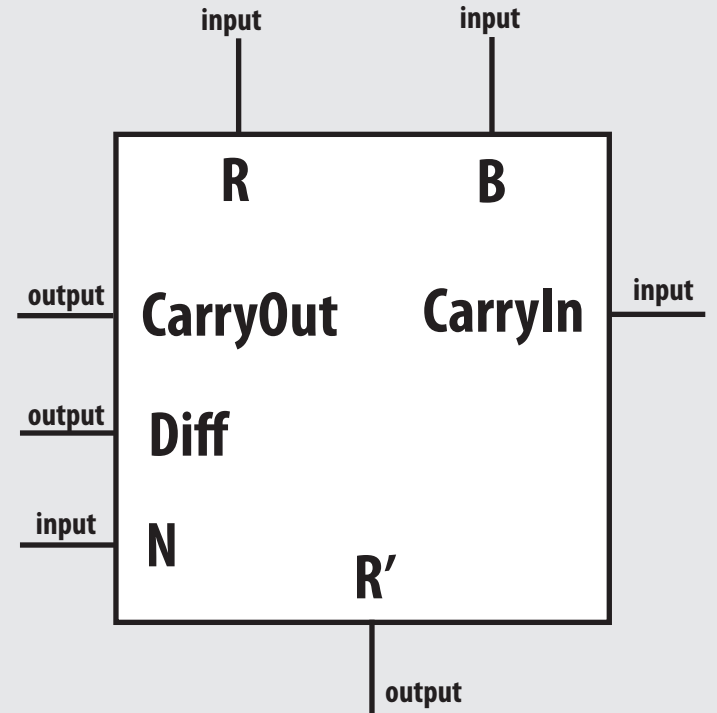
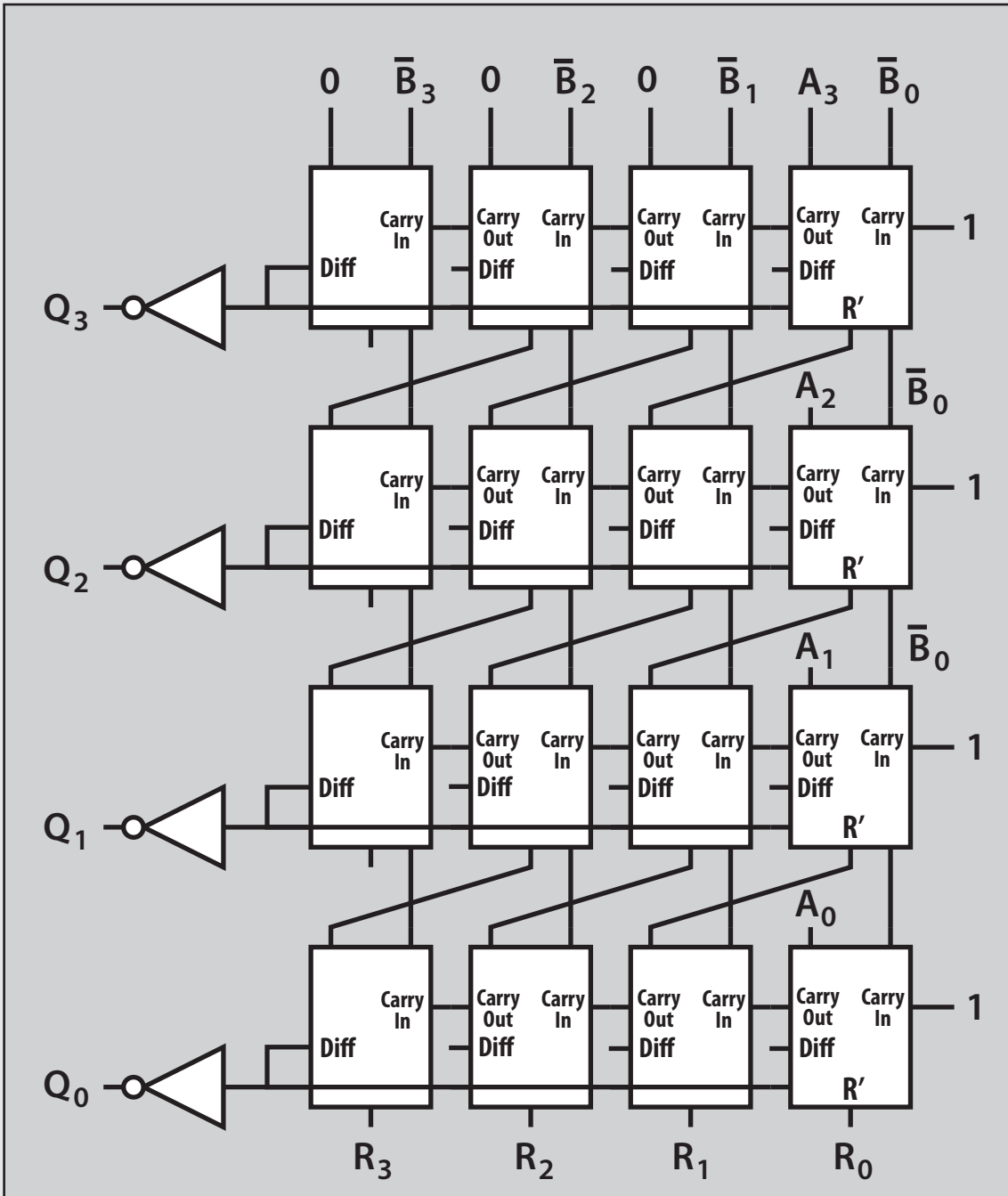
```
R' = 0
for i = N-1 to 0
  R = { R' << 1, Ai } // shift R' to the left
                       // with Ai as least significant bit
  D = R - B
  if D < 0 then Qi = 0, R' = R // R < B
  else          Qi = 1, R' = D // R ≥ B
R = R'
```

Array Divider



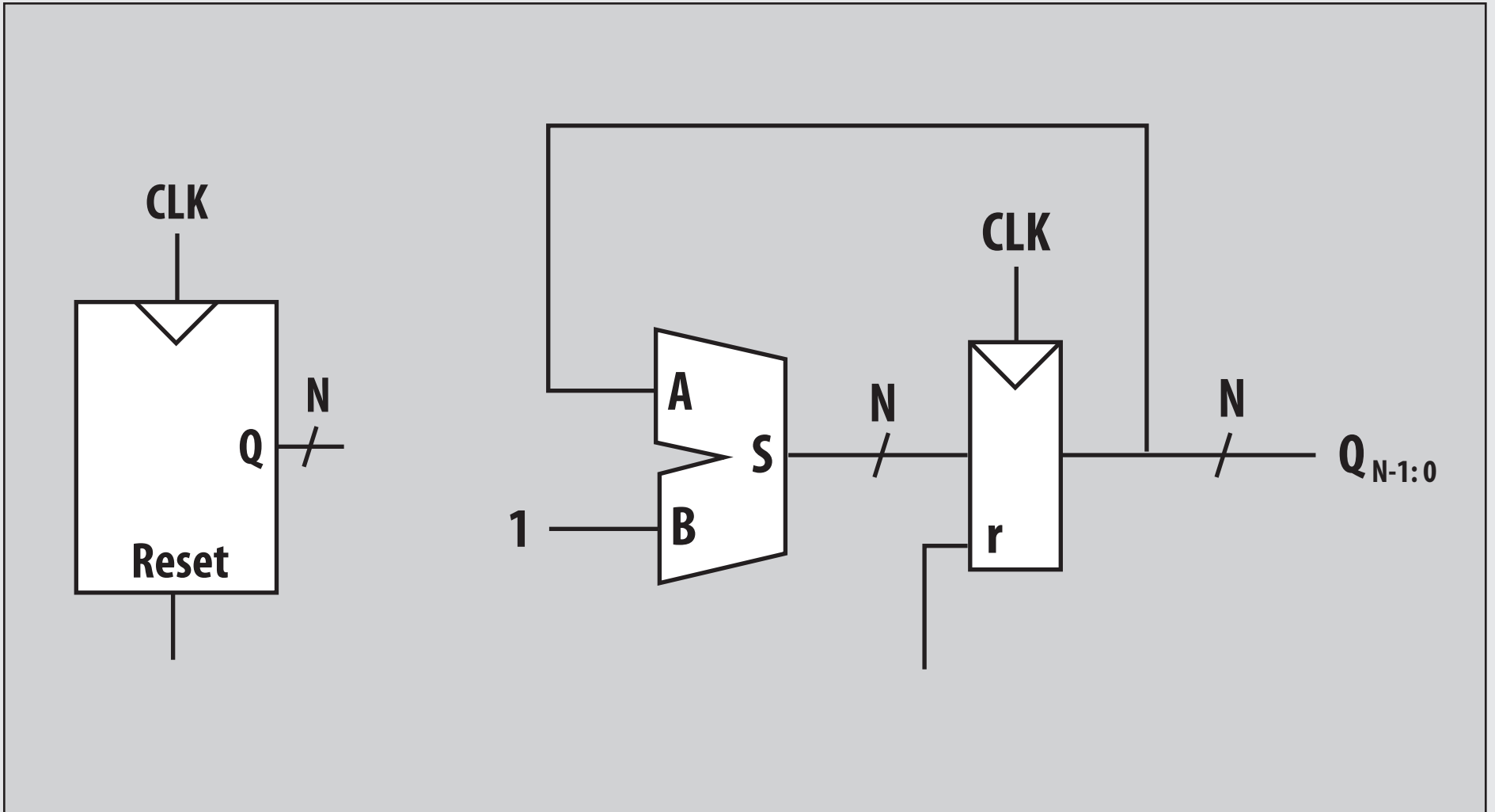
Exercise :

Explain how the circuit works !

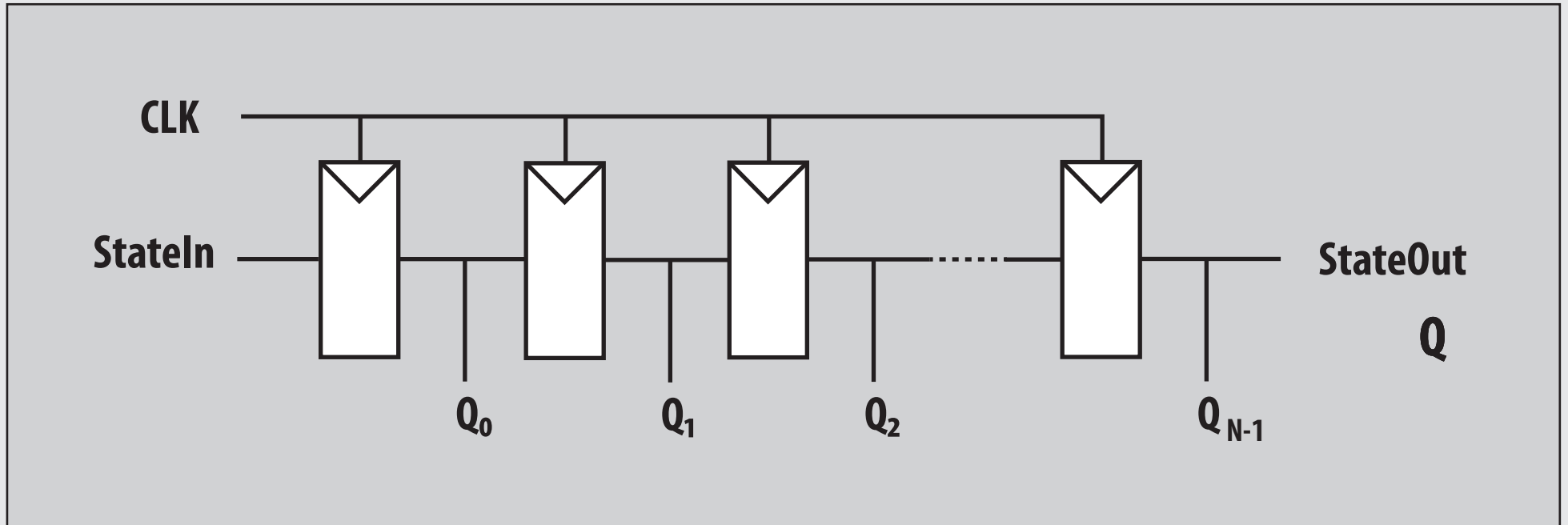


Sequential Building Blocks

Counters

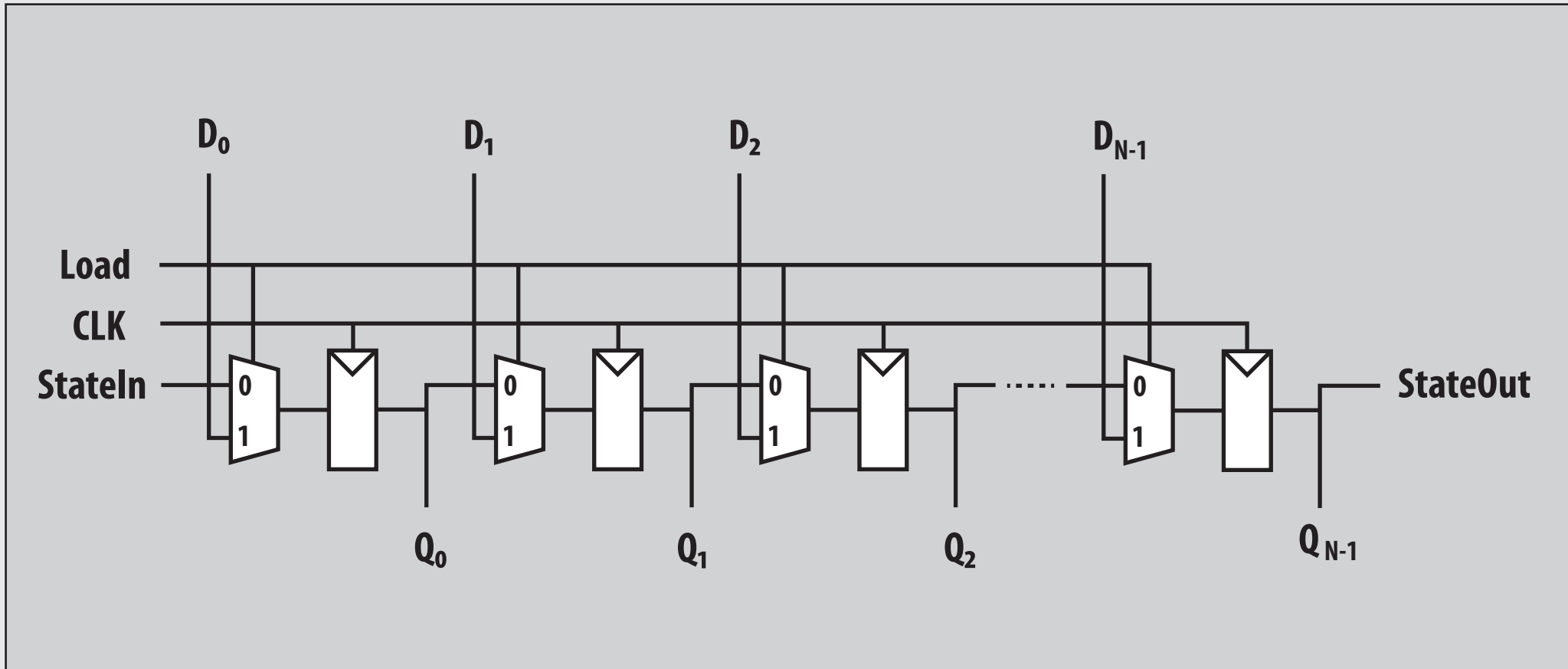


Shift Register



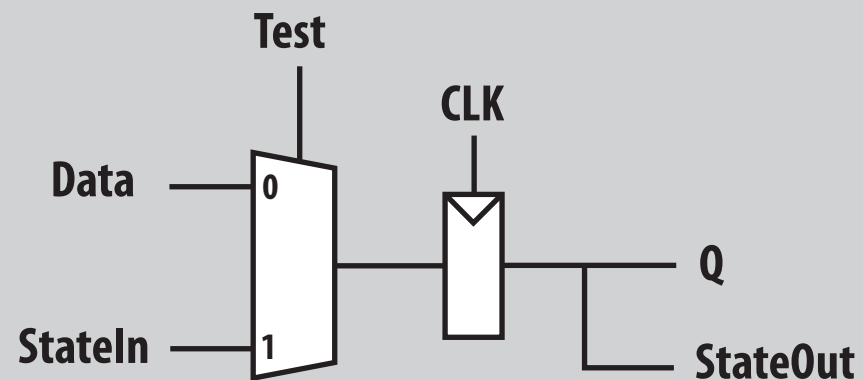
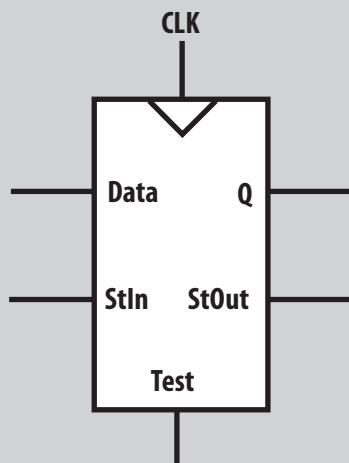
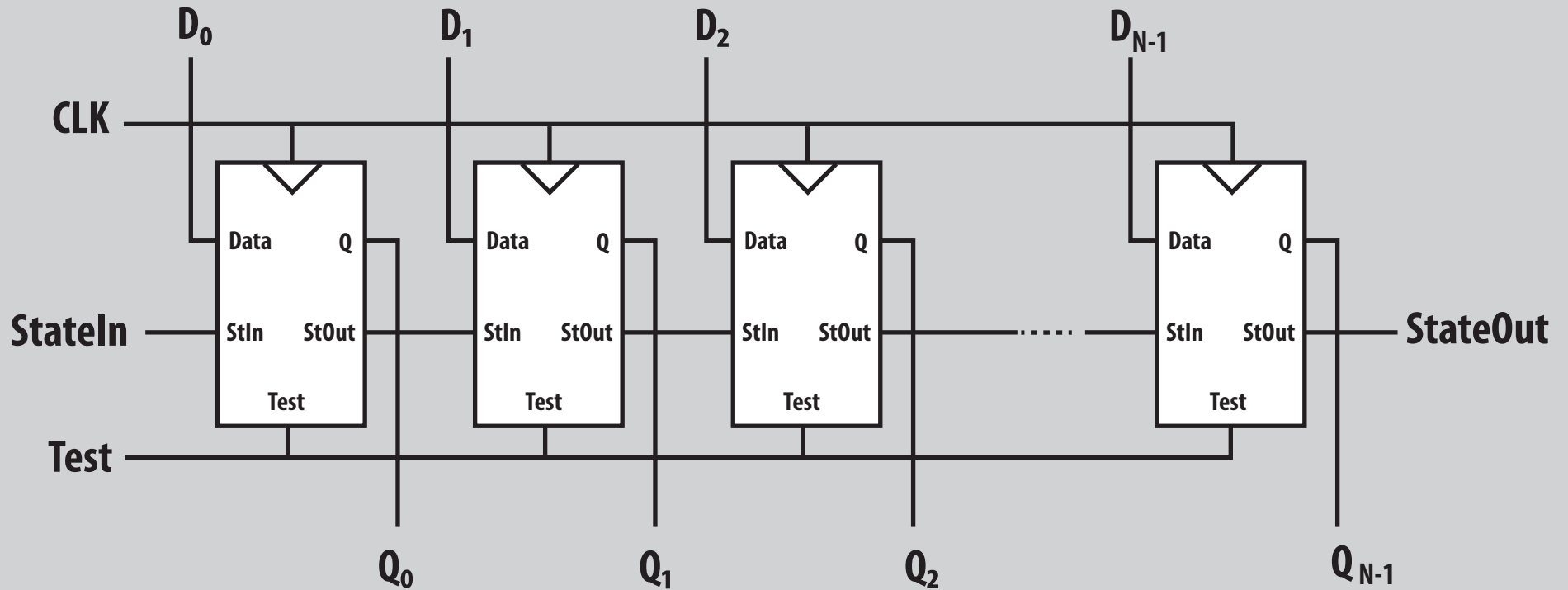
After N cycles, the past N values of the input **StateIn** are available in parallel at the output **Q**.

Shift Register with Parallel Load



Here, the data can be loaded in parallel using the input D

Scannable flip-flop



Scan Chains

