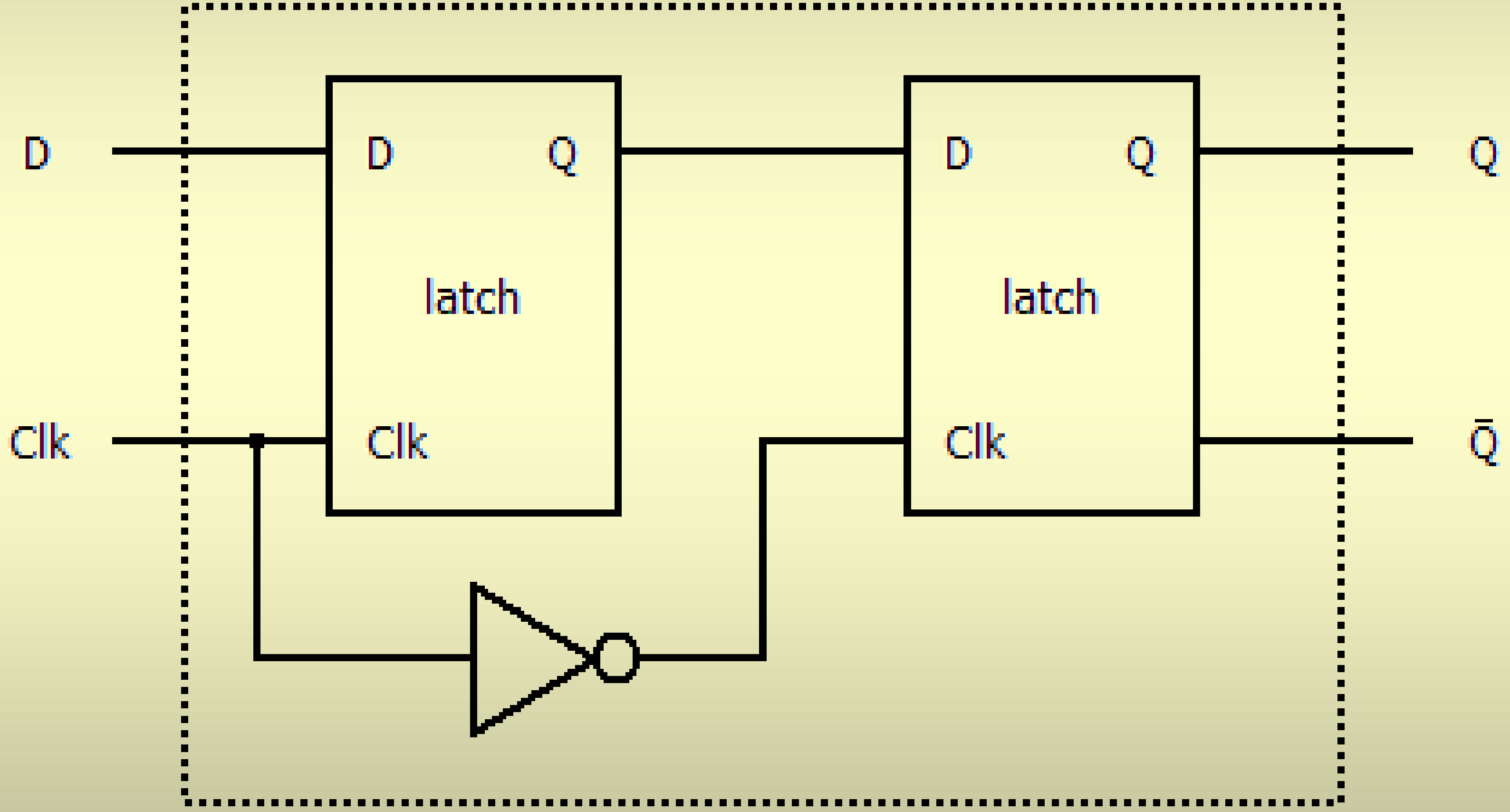
Flip-flop

- inputs are considered on the rising (or falling) edge of the clock signal
- ways of making a flip-flop
  - electronics - derive the clock signal
    - results in an impulse-like signal
  - use latches → master-slave circuits

Master-slave D Flip-flop



Latch vs. Flip-flop

- each category has its utility
- flip-flops - used for controlling digital systems
  - the edge of the clock signal is very short compared with the clock period
    - i.e., it can be considered as a moment
  - during each clock period, the system makes exactly one step of its evolution
- latches - asynchronous systems

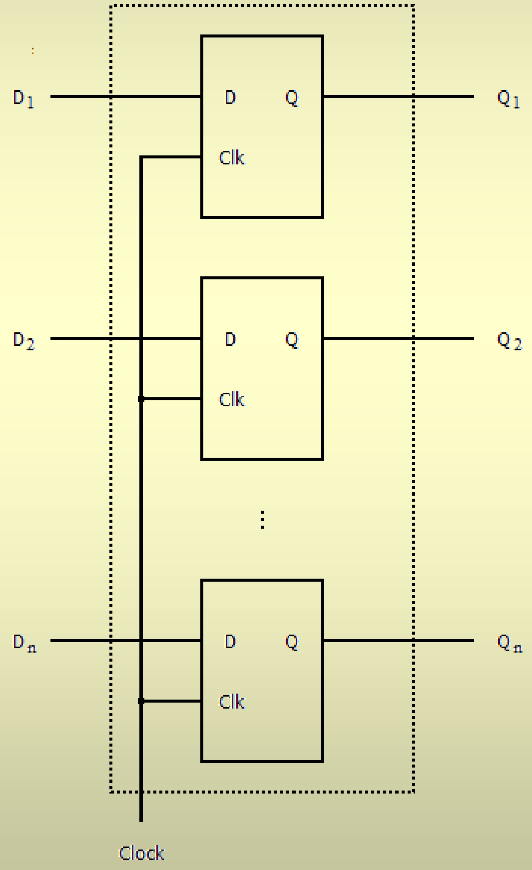
III.2. Complex Sequential Circuits

Registers

- a bistable circuit implements a single bit
  - not very useful in practice
- we can use several bistable circuits together
  - all receive the same command at the same time
  - such a circuit is called register
- types of registers
  - parallel registers
  - shift (serial) registers

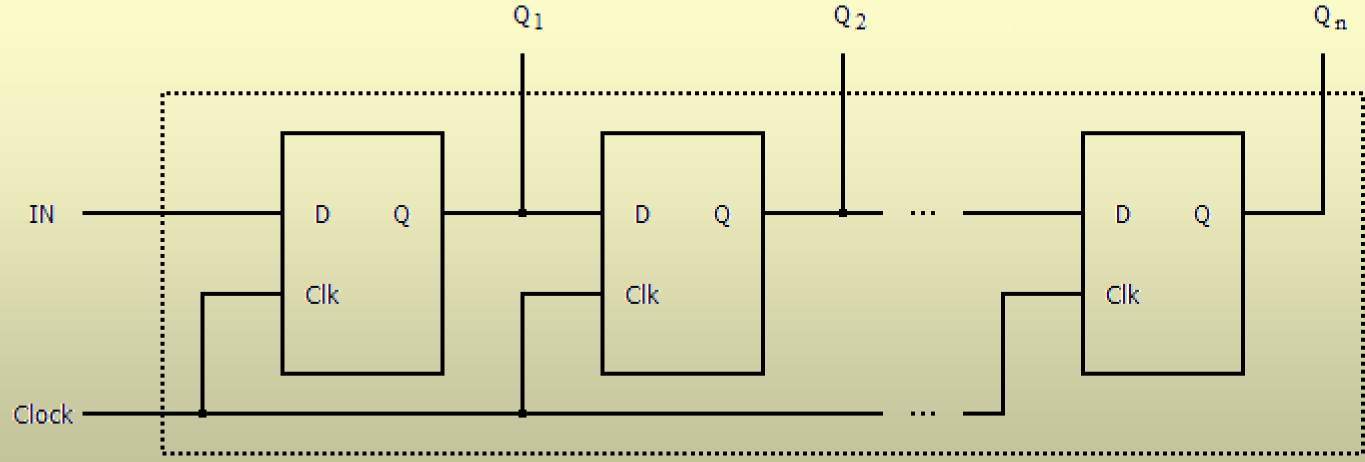
Parallel Register

- implementation with D bistable circuits
  - can be latches or flip-flops, as needed
- the same command (clock)
  - all bits change at the same moments
- natural extension of the bistable circuit

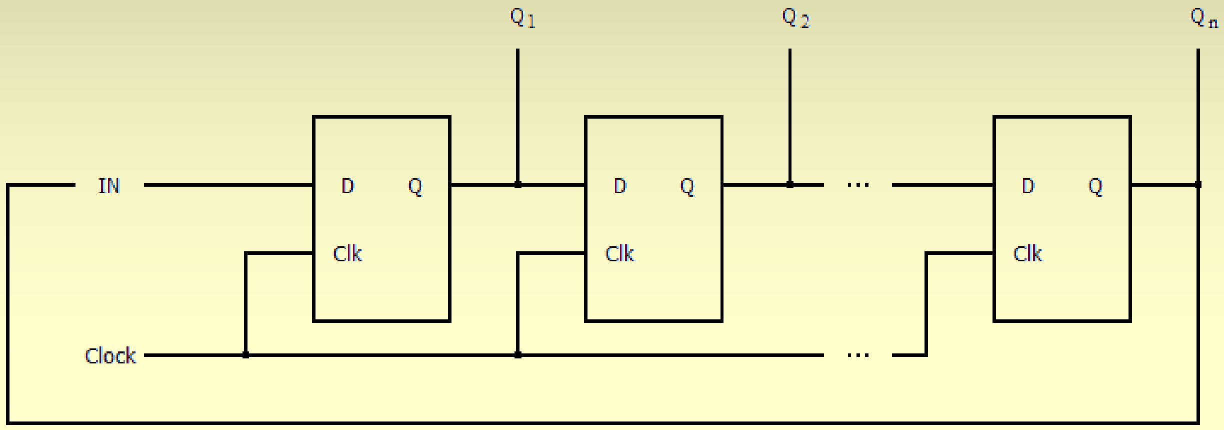


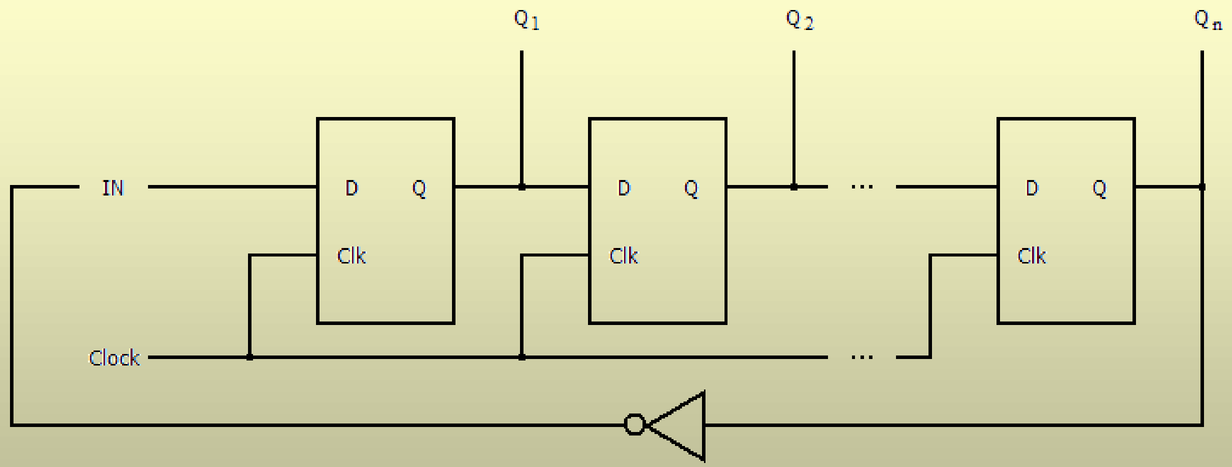
Classic Shift Register

- memorizes the last *n* values applied on the input
- can be implemented only with flip-flops
  - homework: why?



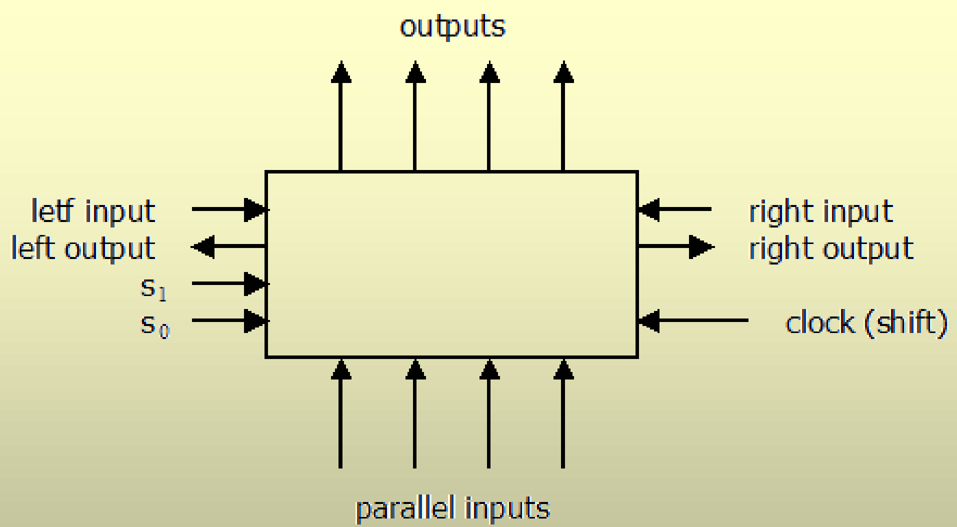
Other Shift Registers





Universal Shift Register

- serial or parallel inputs and outputs
- right or left shift operations
- one can use any of the features above, as needed



s_0	s_1	function
0	0	unchanged
0	1	shift right
1	0	shift left
1	1	parallel load

Designing a Sequential Circuit (1)

- finite state machine (automaton)

1. determine the states of the circuit
2. determine the state transitions
   - how the next state and the outputs depend on the inputs and the current state
3. state encoding
   - using the necessary number of bits
4. write the truth table for the state transitions

Designing a Sequential Circuit (2)

5. minimization
6. implementation
   - the state - memorized by flip-flops
   - combinational part - from the minimization
     - the inputs of the combinational part (current state) are collected from the outputs of the flip-flops and the input variables
     - the outputs of the combinational part (next state) are applied at the inputs of the flip-flops

Binary Counter

- at each moment keeps an *n*-bit number
- at each clock "tick" - incrementation
  - could also be decrementation
  - after the maximum value, 0 comes next
  - no inputs, only state variables
    - which keep the current value of the number
  - outputs are identical to the state variables

Example: n=4

current state				next state				current state				next state			
q_3	q_2	q_1	q_0	d_3	d_2	d_1	d_0	q_3	q_2	q_1	q_0	d_3	d_2	d_1	d_0
0	0	0	0	0	0	0	1	1	0	0	0	1	0	0	1
0	0	0	1	0	0	1	0	1	0	0	1	1	0	1	0
0	0	1	0	0	0	1	1	1	0	1	0	1	0	1	1
0	0	1	1	0	1	0	0	1	0	1	1	1	1	0	0
0	1	0	0	0	1	0	1	1	1	0	0	1	1	0	1
0	1	0	1	0	1	1	0	1	1	0	1	1	1	1	0
0	1	1	0	0	1	1	1	1	1	1	0	1	1	1	1
0	1	1	1	1	0	0	0	1	1	1	1	0	0	0	0

Example: *n*=4

- by minimization we get the equations below

$$d_0 = \overline{q_0} = q_0 \oplus 1$$

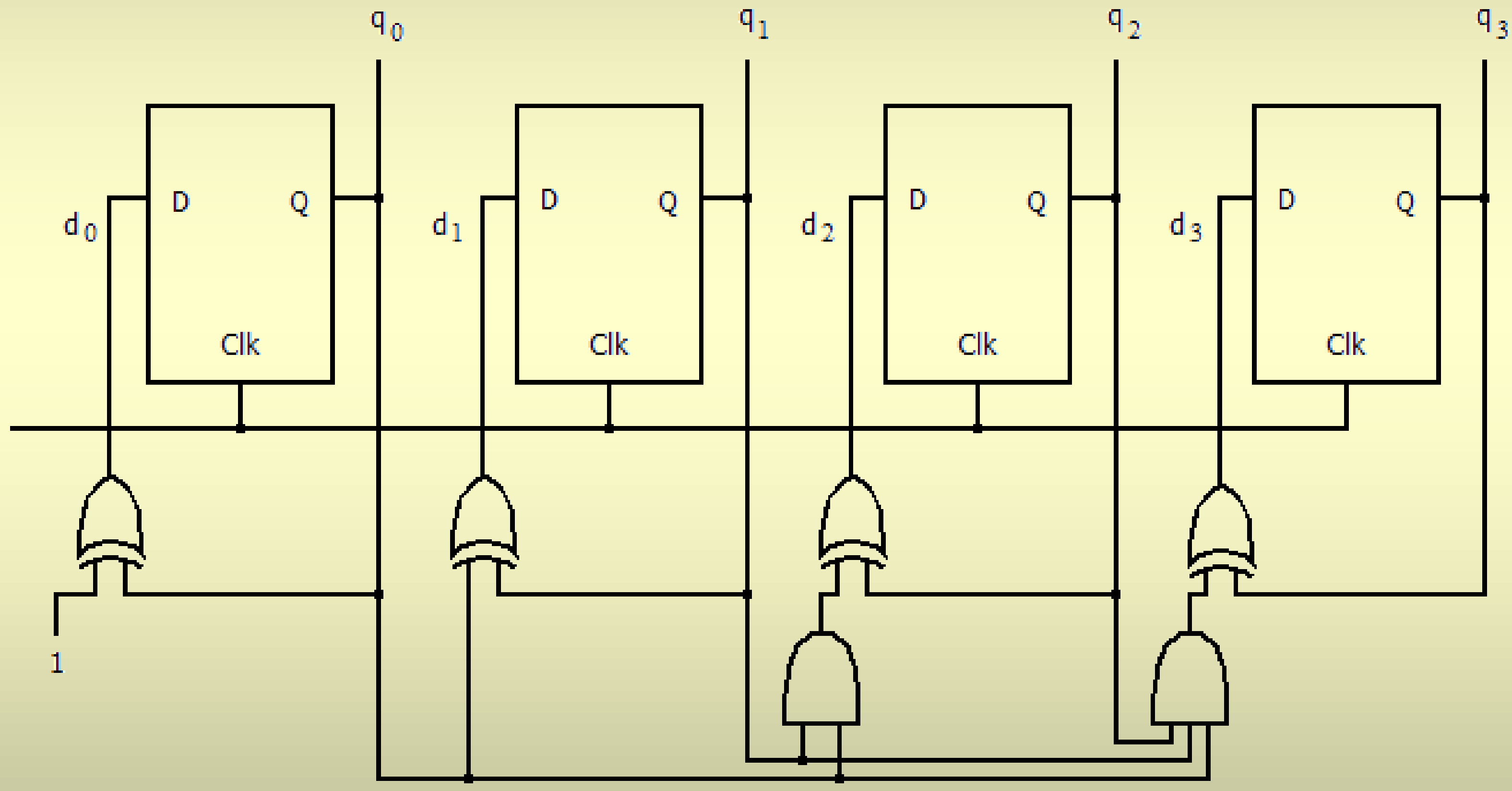
$$d_1 = \overline{q_1} \cdot q_0 + q_1 \cdot \overline{q_0} = q_1 \oplus q_0$$

$$d_2 = \overline{q_2} \cdot q_1 \cdot q_0 + q_2 \cdot \overline{q_1} + q_2 \cdot \overline{q_0} = q_2 \oplus \left(q_1 \cdot q_0\right)$$

$$d_3 = \overline{q_3} \cdot q_2 \cdot q_1 \cdot q_0 + q_3 \cdot \overline{q_2} + q_3 \cdot \overline{q_1} + q_3 \cdot \overline{q_0} =$$
$$= q_3 \oplus \left(q_2 \cdot q_1 \cdot q_0\right)$$

  – state implementation - D flip-flops

Implementation



Microprogramming (1)

- alternative implementation technique
  - the state is still memorized by flip-flops
  - combinational part - implemented by a ROM circuit
  - the inputs of the Boole functions are applied to the address inputs of the ROM
  - the outputs of the Boole functions are collected from the data outputs of the ROM

Microprogramming (2)

- implementation of the combinational part
  - start from the truth table
  - to each location - write the desired output values
- advantage - flexibility
  - any change of the automaton requires only the rewriting of the contents of the ROM
- drawback - low speed
  - ROM circuits are slower than logic gates

The Same Example

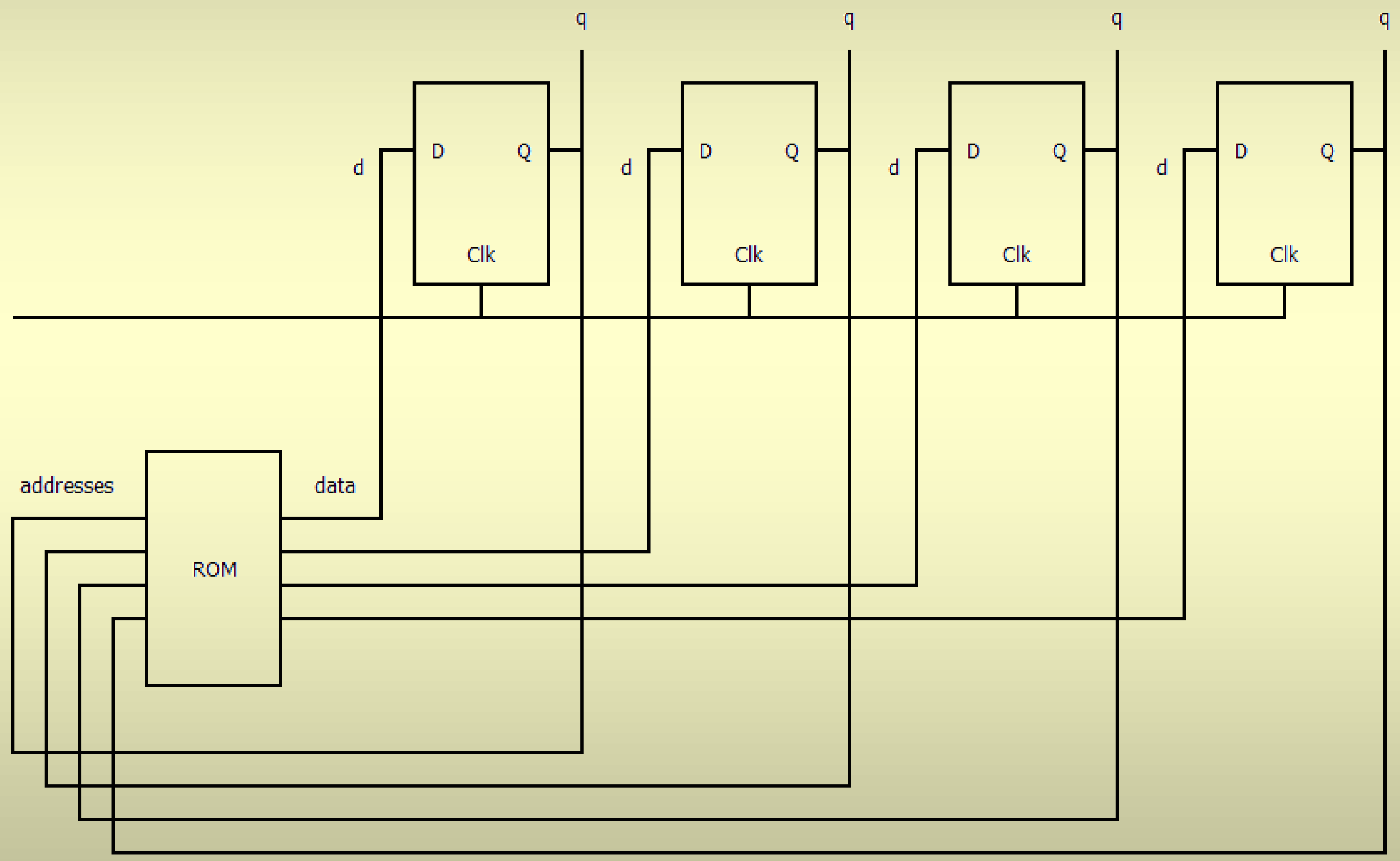
- there are 16 (= 2^4) states
  - – encoded with 4 state bits
- so the ROM circuit will have
  - – 2^4 addresses $\rightarrow$ 4 address bits
    - 16 locations
  - – 4 data bits $\rightarrow$ locations are 4 bits wide
  - – in this example there are no inputs and outputs of the system - only state bits

The Contents of the ROM

address	value
0	0 0 0 1
1	0 0 1 0
2	0 0 1 1
3	0 1 0 0
4	0 1 0 1
5	0 1 1 0
6	0 1 1 1
7	1 0 0 0

address	value
8	1 0 0 1
9	1 0 1 0
10	1 0 1 1
11	1 1 0 0
12	1 1 0 1
13	1 1 1 0
14	1 1 1 1
15	0 0 0 0

Implementation



IV. Internal Representations

- elementary internal representations
  - they are part of the computer's architecture
  - so they are implemented in hardware
  - directly accessible to the programmers
- more complex data structures
  - based on elementary representations
  - defined and accessible to the programmers by software

Elementary Representations

- numerical data
  - integer/rational numbers
  - only certain subsets of these sets
- alpha-numerical data
  - characters etc.
- instructions
  - the only system-specific representations
  - thus non-standardized and non-portable

Studying the Representations

- numerical representations

  $\text{repr}(n_1)\ \textit{op}\ \text{repr}(n_2) = \text{repr}(n_1\ \textit{op}\ n_2)$???

- example - if we add two integer variables, will the result fit into its destination?
- representation errors
  - approximations
  - overflows

Sending the Information

- between various physical media
  - between computers/systems
  - between the components of a computer/system
- transmission errors may occur
  - due to perturbations/incorrect working
  - digital signal - some bits are inverted
  - we wish to detect to occurrence of such errors
  - and even fix them, where possible (correction)

Ways of Detection/Correction

- use additional *redundant* bits
- parity - 1 additional bit
  - allows detecting the occurrence of a (1 bit) error
  - odd/even parity: odd/even number of bits 1
- Hamming code
  - 4 information bits, 3 additional bits
  - detection/correction of multiple errors simultaneously

Example: Odd Parity

- transmitter
  - has to send value $(110)_2$
  - 2 bits of value 1 (even) $\rightarrow$ the additional bit is 1
  - sends $(1101)_2$
- receiver
  - receives the bit string
  - if the number of bits of value 1 is even - error
  - else - eliminate the parity bit and get $(110)_2$

IV.1. Alpha-numerical Codes

Alpha-numerical Codes

- the computer cannot represent characters directly
  - or any non-numerical information: images etc.
- each character is associated a unique number
  - the character is encoded
  - encoding can be at hardware level (elementary representation) or at software level

Standards

- ASCII
  - each character - 7 bits plus one parity bit
- EBCDIC
  - former competitor of ASCII
- ISO 8859-1
  - extends the ASCII code
- Unicode, UCS
  - non-latin characters

ASCII Code

- small letters are assigned consecutive codes
  - in the order given by the English alphabet
  - 'a' - 97; 'b' - 98; ...; 'z' - 122
- similarly - capitals (65, 66, ..., 90)
- similarly - characters that display decimal digits
  - attention: character '0' has code 48 (not 0)
- lexicographic comparison - binary comparison circuit

IV.2. Internal Number Representation

Positional Representation

- also a representation
  - 397 is not a number, but a number representation
- invented by Indians/Arabs
- implicit factor attached to each position in the representation
- essential in computer architecture
  - allows efficient computing algorithms

Base (Radix)

- any natural number d>1
- the set of digits for base d: {0,1,…,d-1}
- computers work with base d=2
  - technically: 2 digits - easiest to implement
  - theoretically: base 2 "matches" Boole logic
    - symbols and operations
    - operations can be implemented by Boole functions

Limits

- in practice, the number of digits is finite
- example - unsigned integers
  - 1 byte wide: $0 \div 2^{8}-1$ (= 255)
  - 2 bytes wide: $0 \div 2^{16}-1$ (= 65535)
  - 4 bytes wide: $0 \div 2^{32}-1$ (= 4294967295)
- any number that falls outside the limits cannot be represented correctly

Positional Writing

- consider base $d \in N^{*}-\{1\}$
- and the representation given by the string

  $a_{n-1}a_{n-2}\ldots a_1a_0a_{-1}\ldots a_{-m}$
- the corresponding number is

$$\sum_{i=-m}^{n-1}\left(a_i \times d^i\right)_{(10)}$$

- d^i is the implicit factor for position i
  – including negative powers

Converting from Base d to Base 10

- according to the previous formula
- the decimal point stays in the same position
- example

  $5E4.D_{(16)} = 5\times16^{2} + 14\times16^{1} + 4\times16^{0} + 13\times16^{-1} = 20480 + 3584 + 64 + 0.8125 = 24128.8125_{(10)}$

Converting from Base 10 to Base d

Example: $87.35_{(10)} = 1010111.01(0110)_{(2)}$

integer part

87 / 2 = 43 remainder 1
43 / 2 = 21 remainder 1
21 / 2 = 10 remainder 1
10 / 2 = 5 remainder 0
5 / 2 = 2 remainder 1
2 / 2 = 1 remainder 0
1 / 2 = 0 remainder 1

$87_{(10)} = 1010111_{(2)}$

(digits are considered bottom-up)

fractional part

$0.35 \times 2 = 0.7 + 0$
$0.7 \times 2 = 0.4 + 1$
$0.4 \times 2 = 0.8 + 0$
$0.8 \times 2 = 0.6 + 1$
$0.6 \times 2 = 0.2 + 1$
$0.2 \times 2 = 0.4 + 0$
$0.4 \times 2 = 0.8 + 0$

(period)

$0.35_{(10)} = 0.01(0110)_{(2)}$

Conversions between Bases

- one base is a power of the other base
  - $d_1 = d_2^k \Rightarrow$ to each digit in base d_1 correspond exactly k digits in base d_2
- both bases are powers of the same number *n*
  - conversion can be made through base *n*

$$703.102_{(8)} = 111\ 000\ 011.001\ 000\ 010_{(2)} =$$

$$= 0001\ 1100\ 0011.0010\ 0001\ 0000_{(2)} =$$

$$= 1C3.21_{(16)}$$

Approximation and Overflow

- a representation has n digits for the integer part and m digits for the fractional part
  - n and m are finite
- if the number requires more than n digits for the integer part, overflow occurs
- if the number requires more than m digits for the fractional part, approximation occurs
  - at most 2^{-m}

IV.3. BCD and Excess Representations

BCD Representation

- numbers are represented as strings of digits in base 10
  - each digit is represented on 4 bits
- utility
  - business applications (financial etc.)
  - base 10 displays (temperature etc.)
- arithmetical operations - hard to perform
  - addition - cannot simply use a binary adder

BCD Addition (1)

$$
\begin{array}{rcll}
5 & = & 0101 & + \\
3 & = & \underline{0011} & \\
8_{(10)} & = & 1000 & = 8_{BCD}
\end{array}
\qquad
\begin{array}{rcll}
5 & = & 0101 & + \\
8 & = & \underline{1000} & \\
13_{(10)} & = & 1101 & \\
 & & \multicolumn{2}{l}{\neq 13_{BCD} =} \\
 & & \multicolumn{2}{l}{= 0001\ 0011}
\end{array}
$$

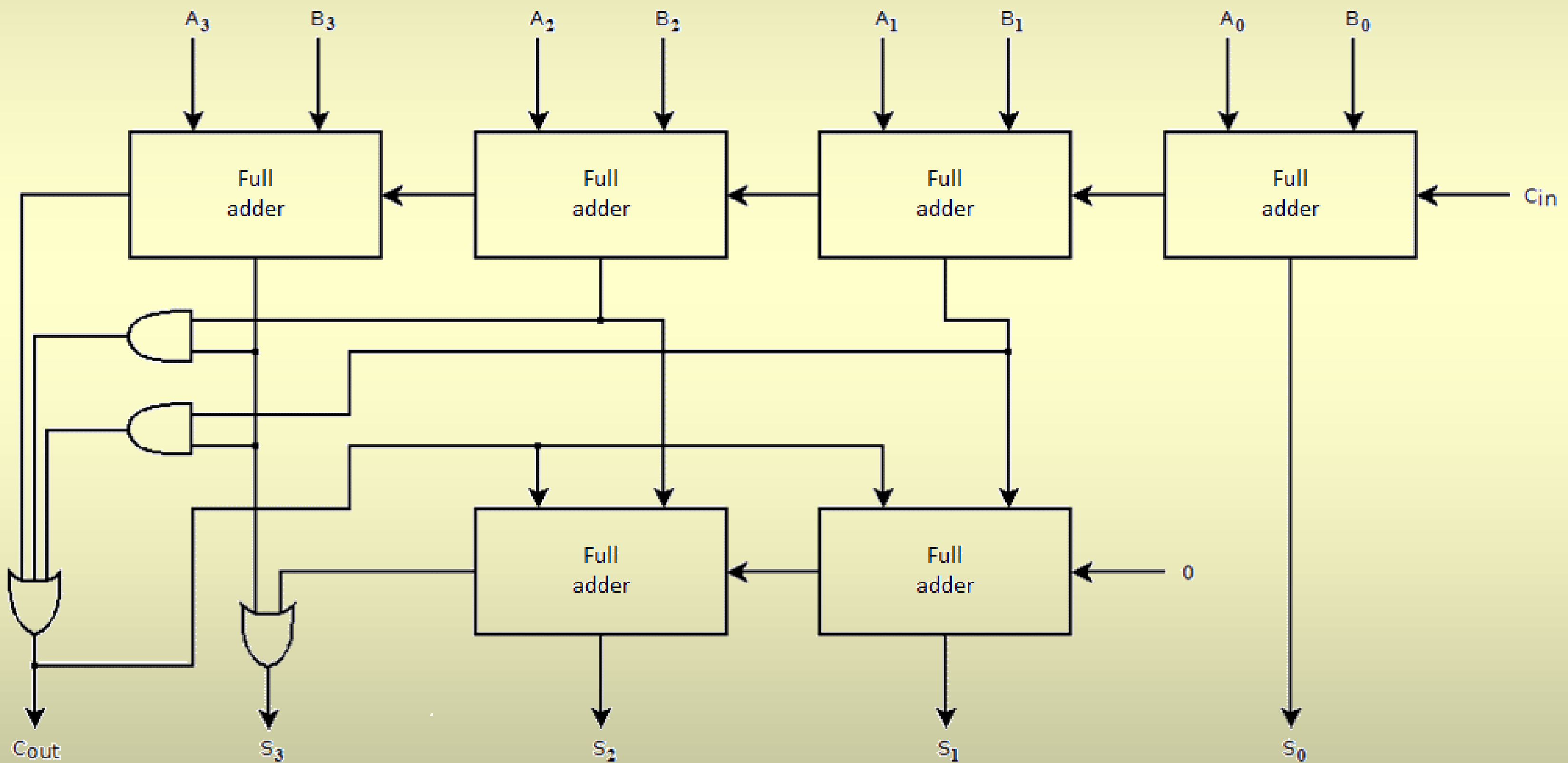
problems occur when the sum of the BCD digits exceeds 9

BCD Addition (2)

- solution
  - add 6 (0110) when the sum exceeds 9
- homework: why?

$$
\begin{array}{rcrl}
5 & = & 0101 & + \\
8 & = & \underline{1000} & \\
 & & 1101 & + \\
6 & = & \underline{0110} & \\
 & & 1\ 0011 & = 13_{BCD}
\end{array}
\qquad
\begin{array}{rcrl}
9 & = & 1001 & + \\
7 & = & \underline{0111} & \\
16_{(10)} & = & 1\ 0000 & \neq 16_{BCD} \\
6 & = & \underline{0110} & \\
 & & 1\ 0110 & = 16_{BCD}
\end{array}
$$

BCD Adder



A_3
B_3
A_2
B_2
A_1
B_1
A_0
B_0
Full adder
Full adder
Full adder
Full adder
Cin
Full adder
Full adder
0
Cout
S_3
S_2
S_1
S_0

Excess Representation

- based on positional writing
  - non-negative numbers
  - on n bits, the interval of numbers that can be represented is $0 \div 2^n-1$
- the Excess-k representation
  - for each bit string, subtract k from its value given by positional writing
  - the interval that can be represented: $-k \div 2^n-k-1$

Example: Excess-5

Binary	Decimal	Excess-5	Binary	Decimal	Excess-5
0000	0	-5	1000	8	3
0001	1	-4	1001	9	4
0010	2	-3	1010	10	5
0011	3	-2	1011	11	6
0100	4	-1	1100	12	7
0101	5	0	1101	13	8
0110	6	1	1110	14	9
0111	7	2	1111	15	10