Warsaw University of Technology
Faculty of Electronics and Information Technology

Laboratory of Digital Circuits

Functional blocks & Microprogramming

Introduction

Introduction

One of the biggest problems faced while designing a digital system is how to handle its complexity. The easiest and most effective method is decomposition. The system is divided into smaller, repeatable parts called functional blocks. In silicon compilers, functional blocks are predefined and available in a large variety.

The digital system usually consists of two cooperating parts:

- Operational or Data Path Unit – responsible for data processing, like calculations, comparisons, data storage
- Control Unit – a device used to control the data path unit implementing the algorithm.
- The control unit communicates with the data path unit using control signals, called flags. The control unit generates commands like load data or count and receives status information from the data path unit. The general structure of a digital system is shown in Fig. 1.

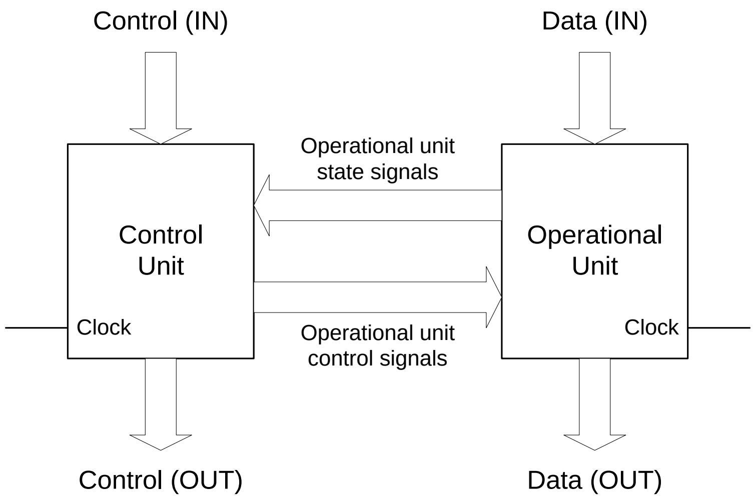
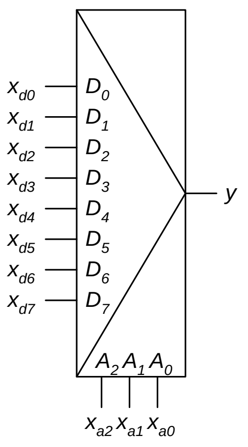


Figure 1

Operational unit

An operational unit is usually composed of functional blocks executing elementary operations on data and driven by control signals. Basic functional blocks include:

- **Multiplexer (MUX)** – a controlled switch also used for Boolean form synthesis. The output of the block is the signal from the data input selected by the address input. N bit multiplexer has n-bit address input and 2^n data inputs. The 3-bit MUX is shown in Fig.2.
- **Demultiplexer** (DMUX) or **Decoder**– a controlled switch also used for multiple output synthesis. DMUX connects single input to the outputs addressed by the address input. N-bit demultiplexer has n-bit address input and 2^n data outputs. In decoders data input is costant and set to 1. The 3-bit DMUX is shown in Fig.3.
- **Comparator** –used to compare two numbers supplied to their A and B inputs. Both input vectors are treated as Natural Binary Code (NBC) numbers. Comparators have three outputs – greater, equal, and smaller. Comparators are cascadable. We can connect n-bit blocks to compare 2n-bit or broader numbers. N-bit comparator is presented in Fig. 4.
-

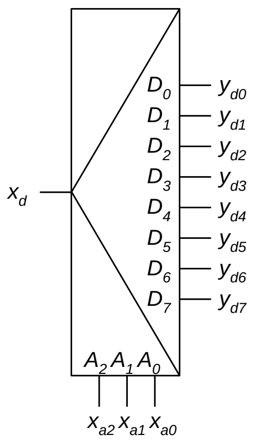


x_{di} - data inputs, $i = \{0, \ldots, 7\}$

x_{ai} - address inputs, $i = \{0, 1, 2\}$

y - data output

Figure 2

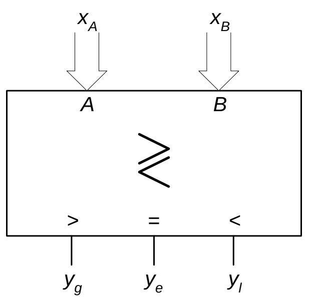


x_d - data input

x_{ai} - address inputs, $i = \{0, 1, 2\}$

y_{di} - data outputs, $i = \{0, \ldots, 7\}$

Figure 3

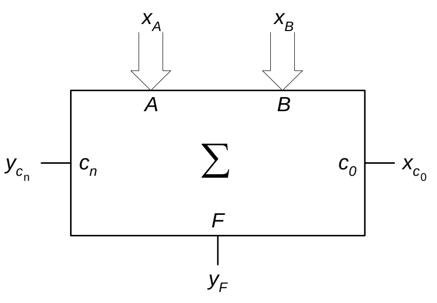


x_A, x_B - input numbers

y_g - greater
y_e - equal
y_l - smaller (less)

•

Figure 4



x_A, x_B - input numbers

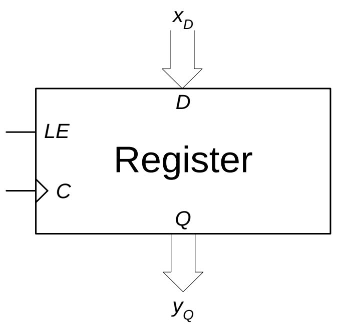
y_F - output number ($x_A + x_B$)

x_{c_0} - carry in

y_{c_n} - carry out

Figure 5

- **Adders**– used to add numbers. Both input vectors are treated as n-bit numbers, usually encoded in Natural Binary Code (NBC}. Adder returns the sum of input A - x_A, input B - x_B, and *carry in* x_{co} numbers. N-bits of the total are outputted in y_F output, while the most significant bit is a *carry out*. Carry input and carry output allow iterative connection of to process 2n-bit or wider numbers. Adder is presented in Fig.5.

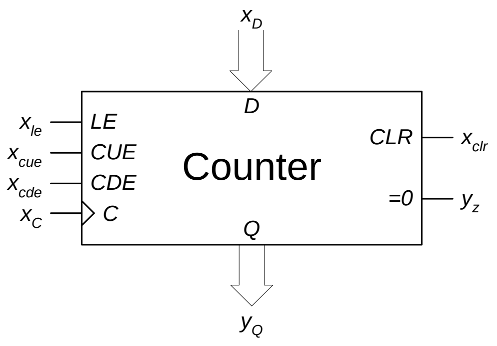


x_D - input vector
y_Q - output vector

LE - load enable
C - clock

Figure 6

- **Registers** – used for data storage. Registers are classified depending on the method used to input and output data – these can be parallel or serial. Hence, there are four possibilities: PIPO (Parallel Input Parallel Output), PISO (Parallel Input Serial Output), SIPO (Serial Input Parallel Output), SISO (Serial Input Serial Output). Registers can work synchronously or asynchronously (both static and dynamic). Register can be cleared, loaded with the input data or shifted. Only one of these operations can be taken at the moment. PIPO register example is presented in Fig. 6.



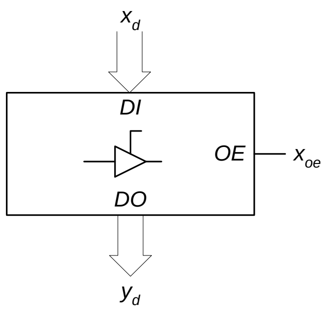
x_D - input number
y_Q - output number

x_{le} - load enable
x_{cue} - count up enable
x_{cde} - count down enable
x_C - clock
x_{clr} - clear

y_z - zero indicator

Figure 7

- **Counter** – a special kind of memory used for counting. The counting is executed by adding (count up) or subtracting (count down) one from the previous counter value (synchronously). A counter can work synchronously or asynchronously (both static and dynamic). It can be cleared, loaded with the input data, incremented, and. Only one of these operations can be taken at the moment. Example of bidirectional (reversible) counter is shown in Fig. 7.



x_d - data input
x_{oe} - output enable
y_d - data output

Figure 8

- **Three-State Gate** – a buffer used for connection with bidirectional buses. It allows the connection of outputs. A three-state gate can transmit the signal or present a high impedance state at the output. It will enable several units to send data over the same signal bus. However, only one unit can send at a time. Other units can only listen and must be disconnected by setting three-state buffer's output to the high impedance state. Fig.8 presents a three-state gate.

Control unit

A Control unit constitutes a finite state machine of Moore or Mealy type. Among many ways of its implementation, three are worth mentioning.

- Minimal automaton – a state machine with a minimal number of states necessary to implement an algorithm.
- Sequencer – a state machine with states coded with one of n code.
- Micro-programmed unit.

Minimal automata and sequencers are individually designed and application-dependent. Each algorithm change requires redesign and structural changes. On the other hand, micro-programmed units are flexible and can implement any algorithm. Algorithm changes do not require changes in hardware structure. The only thing that needs to be changed is the "program" stored in memory. The general structure of the micro-programmed Moore unit is shown in Fig. 9.

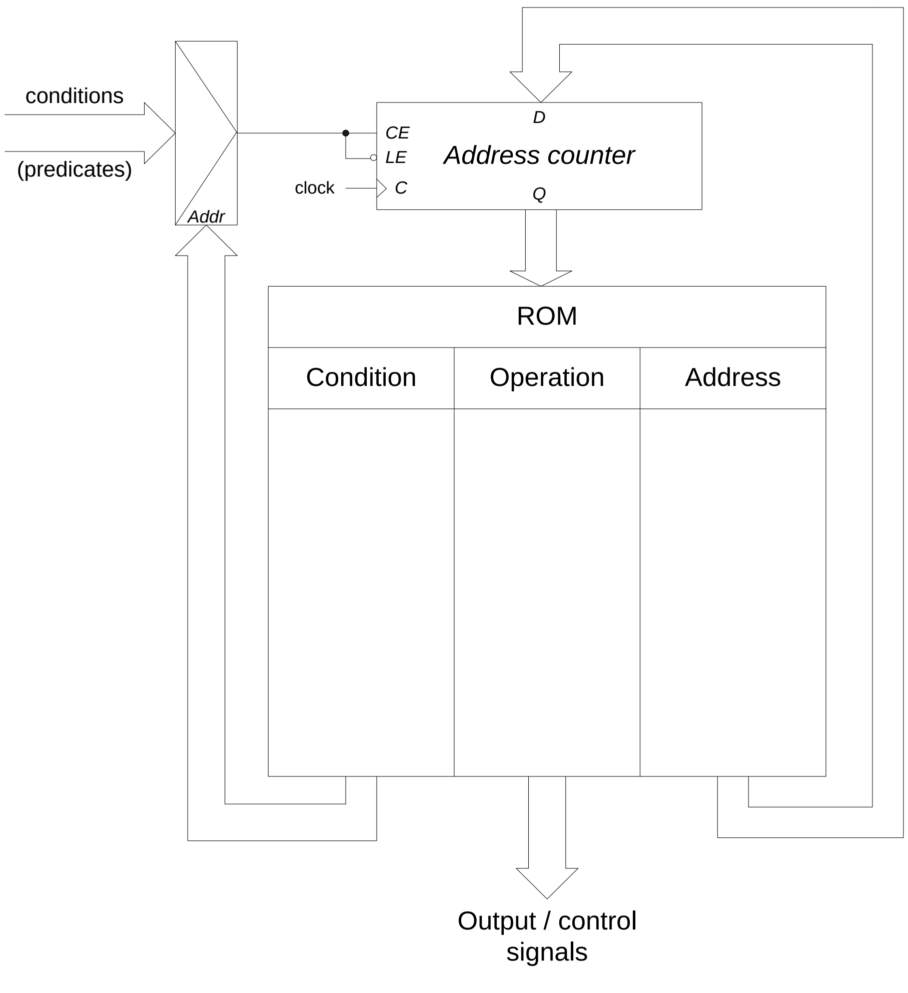


Figure 9

Typical structure of a micro-programmed control unit consists of memory, address counter and condition multiplexer (used to switch between conditions). Rows of the memory represent consecutive states of the state machine. The current state is determined by the value of the address counter, which selects an active row from memory. With each clock pulse, the counter can increment its value by one – which is equivalent to a transition to the next state, or load any value from an external source – which is equivalent to a jump in an algorithm to the the state addressed in the memory. Whether to increment counter's value or load jump address is made based on the value obtained from the multiplexer.

Each row in memory is divided into three columns:

- Condition – holds the address of a condition/predicate.
- Operation – contains control signals to be passed to the operational unit and/or output signals for the external environment.
- Address – contains jump address.

Microinstruction realized by the state machine can be described in the following way (see Fig. 10):

Address: *Operation*; **if** *condition* **then** goto *Address + 1*; **else** goto *Address*$_{jump}$; **fi**.

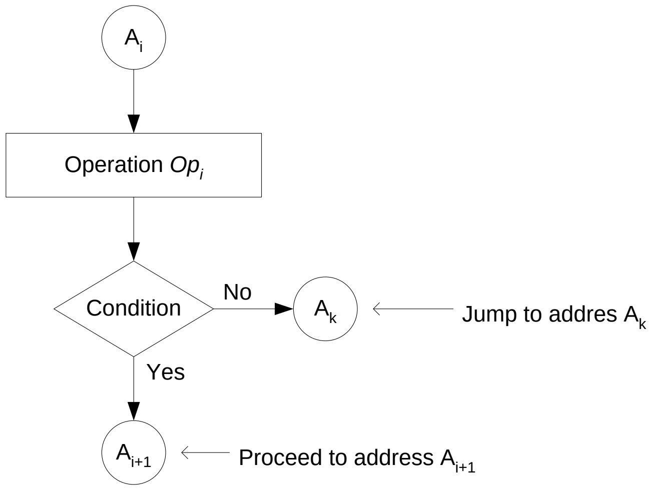


Figure 10

The assumed structure of the state machine imposes several restrictions on methods of creating its flow diagram. Since each row of the memory contains a condition, operation, and jump address, this means that in each state, we choose a condition, send control signals to the operational unit, and possibly make a jump. Consequently, each state has to have a determiner and executioner element of a flow diagram. When a unit is not executing a jump, it proceeds to the next state, represented by the next row of memory – that is, a row with an address next to the address of the current row. This implies that each next state's number must be greater than the current state's number by precisely one (unless we are executing a jump). All of these restrictions must be considered when designing a flow diagram and assigning states to each of its points.

Example of the Sequence Adder

Design a digital system computing:

$$Y = \frac{1}{2} \cdot \sum_{i=1}^{n} x_i$$

Where n and x_i for $i = 1, ..., n$ are 4-bit binary code numbers (n>0). The result Y is an 8-bit binary code whole number. Bits of numbers n and x_i are supplied in parallel. The circuit first receives n, and then consecutive x_i follows. The handshaking is implemented using the following signals:

- Input NR = 1 (*number ready*) – signals the readiness of the following number on input.
- Output R = 1 (*ready for the following number*) requests the following number from the external circuit.
- Output SR = 1 (*system ready*) signals the readiness of the result.

The system is divided into the control and operational subparts. The control subpart is designed as a micro-programmed unit or sequencer. The computation speed is a critical factor.

Design

Before we start designing our circuit, let's try to create a general idea of the unit's operation we are supposed to build. We know that:

- Both n and x_i are 4-bit binary numbers, supplied in parallel using the same input. That means we need to have a 4-bit data input.
- The result Y is an 8-bit number, so we need 8-bit data output.
- Three control signals are used to interact with external circuits – namely *NR* (input), *R* and *SR* (outputs), so we need to incorporate them into our design.

All of the preceding points lead us to the model presented in Fig.11.

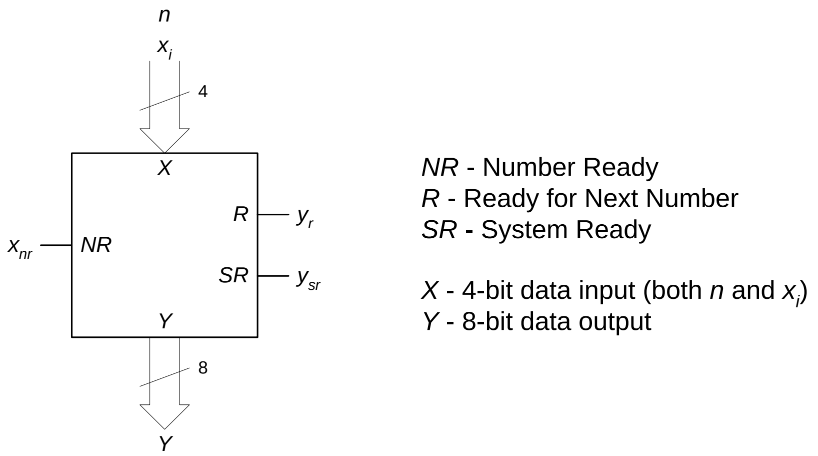


Figure 11

Now, let's try to create a basic flowchart by identifying the operations of our unit. Firstly, we are supposed to read the number of numbers being added (n). Then, we need to read a sequence of numbers, adding them on the way. Finally, we divide the result by 2. Fig.12 presents our flowchart.

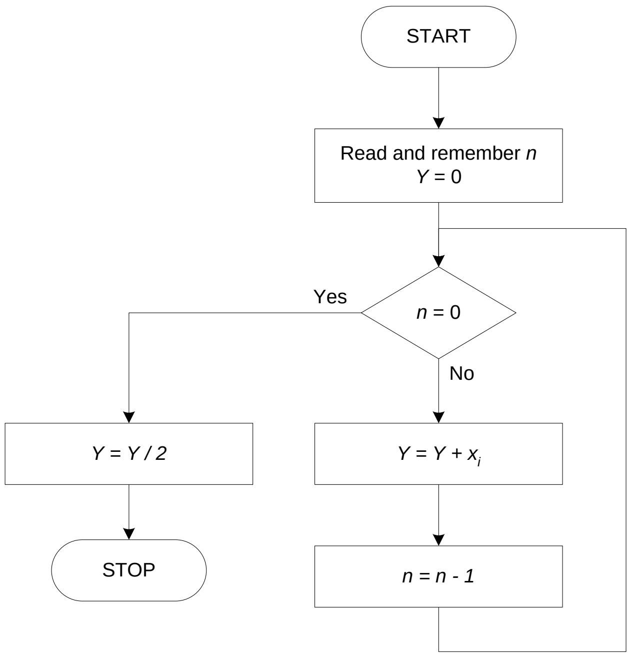


Figure 12

As we can see, we need to perform the following operations:

- Load *n* into memory and clear temporary result Y.
- Test whether *n* is zero.
- With each number read, decrement *n* by one and
- Add x_i to the current value of *Y*.
- Divide *Y* by 2.

Now, let's map the above operations to functional blocks:

- Since *Y* is the total sum of all input numbers and its value depends on the numbers preceding the current number, we need a memory to store its value. Usually, we use registers for data storage. For *Y* is an 8-bit number, we need an 8-bit register. We know that we have to clear *Y* (set it to zero) upon circuit initialization, so we need a clear input. We also know that we will not change *Y* value in every clock cycle, so we need to have a load enable input. It will also be more evident once we take handshaking into account later in this chapter. So, what we get is an *8-bit register with clear and load enable inputs.*
- In the beginning, we receive the number *n*. It is then decremented by one with every incoming number. That implies using a counter designed to increment/decrement its value by one when enabled. In our case, *n* is a 4-bit value, and we will only need to decrement it, so our counter needs to be 4-bit wide and count downwards. Since we will not decrement *n* in every state, we need to have the ability to disable counting – that means our counter should have count enable input. Next, the initial value of *n* must be read from the input, so the counter should also have the ability to load any value. Finally, we also need to test whether *n* is zero, so the counter should have a zero-value indicator. We need a *4-bit synchronous counter with zero detection that counts down with count enable and load enable inputs*. The clear input is not necessary since the first operation of our circuit is load *n*.

- The next thing to do is add. Since Y is an 8-bit value, we need an *8-bit adder*. Carry output can be left floating while carry input is connected to zero. *The addition is not an operation by means of microprogramming.*
- Last but not least, we need a functional block to divide. The implementing division will be described later in this chapter.

Before we present the schematic of the operational unit, we need to consider the data flow. For adding, we connect the adder's output to the register's input. The output of the register contains a temporary total that should be added to the input number when it comes. Therefore the output of the register must be connected to one adder input while input bus must be connected to the second adder input. So, the addition is done by loading the register. For every x_i, the adder returns a total of x_i and the current value of Y, and when enabled, the result is stored inside register memory. To divide, we need a divider, so the register output should be connected to divider input. The output of the divider is the output of the circuit. The operational circuit schematic is shown in Fig.13.

Finally, we need to adjust the width to the required range of numbers. The 4-bit input imposes a 4-bit adder, while an 8-bit output imposes an 8-bit register and adder. Since the adder is an 8-bit unit and the input bus is only 4-bit, we connect the input bus to the lower four bits of adder input. The remaining adder upper four bits must be connected to zero

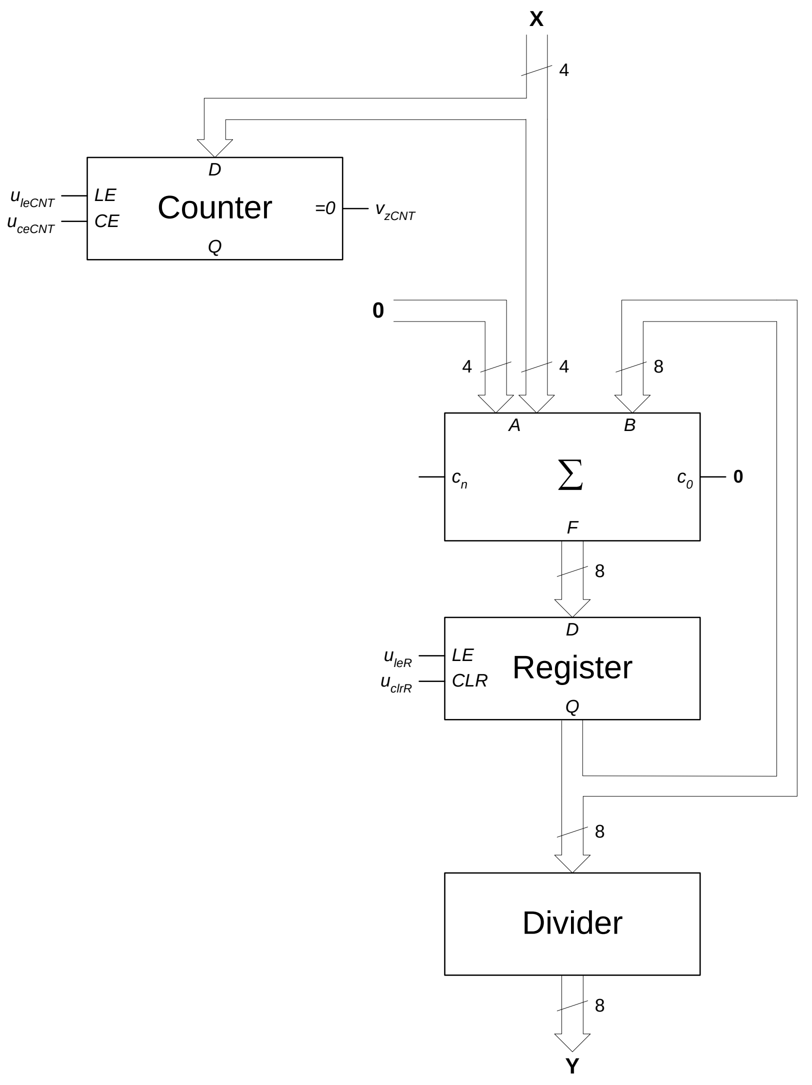


Figure 13

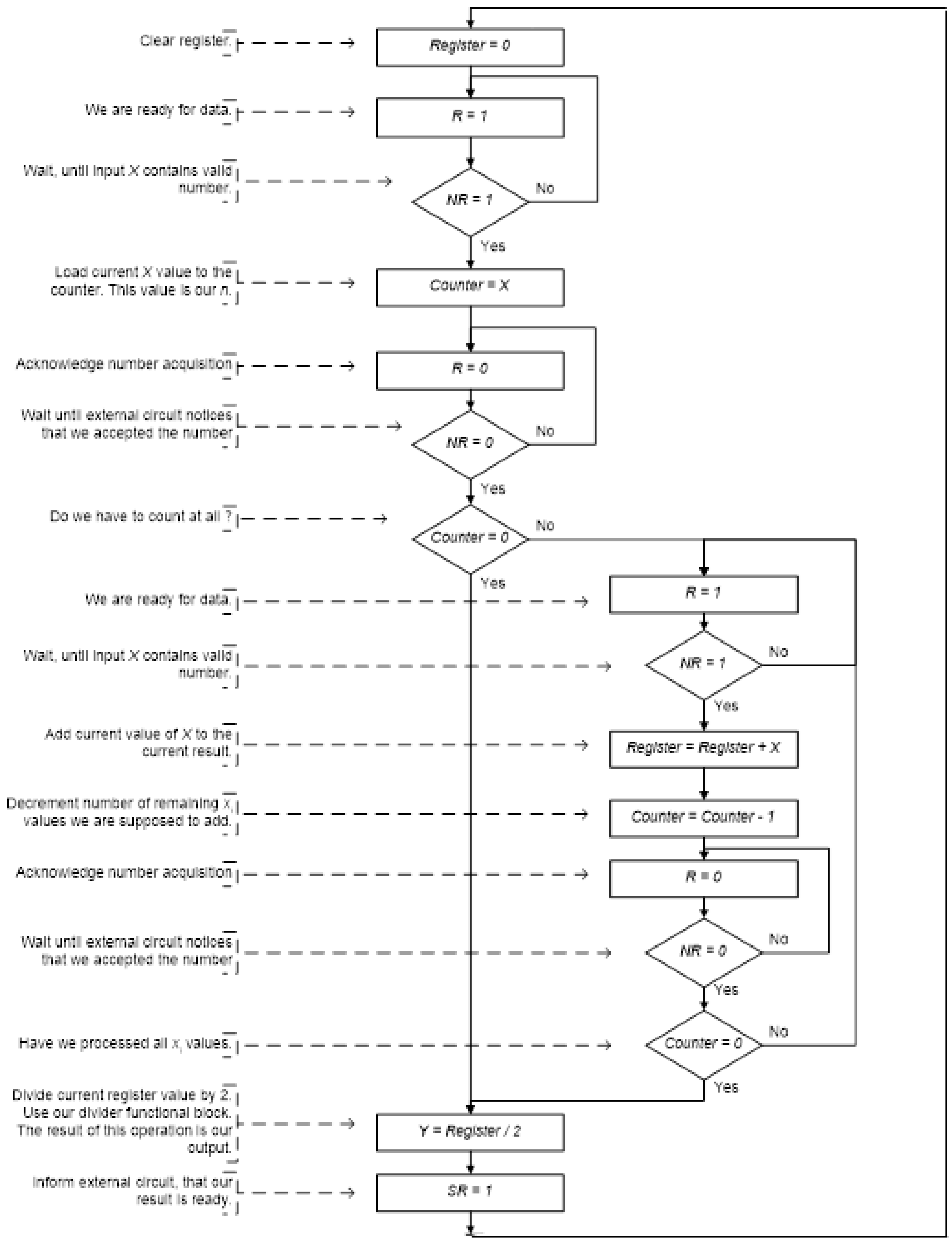

Clear register.
Register = 0
We are ready for data.
R = 1
Wait, until input X contains valid number.
NR = 1
No
Yes
Load current X value to the counter. This value is our n.
Counter = X
Acknowledge number acquisition
R = 0
Wait until external circuit notices that we accepted the number
NR = 0
No
Yes
Do we have to count at all ?
Counter = 0
No
Yes
We are ready for data.
R = 1
Wait, until input X contains valid number.
NR = 1
No
Yes
Add current value of X to the current result.
Register = Register + X
Decrement number of remaining x_i values we are supposed to add.
Counter = Counter - 1
Acknowledge number acquisition
R = 0
Wait until external circuit notices that we accepted the number
NR = 0
No
Yes
Have we processed all x_i values.
Counter = 0
No
Yes
Divide current register value by 2. Use our divider functional block. The result of this operation is our output.
Y = Register / 2
Inform external circuit, that our result is ready.
SR = 1

Figure 14

Now, knowing how our operational circuit looks, we can create a flowchart representing our control unit's operation. We start with clearing the total register, and then we inform the external circuit that we are ready to accept the number of numbers *n*. After the external circuit signals that *X* contains valid data, we load this value into the counter, and then we acknowledge number acquisition to the external circuit. Next, we check whether *n* is zero, and if not, we proceed with the addition. The addition is performed in the following order: wait for valid data on input *X*, add *X* to the current value of the register, decrement counter value by one, and acknowledge number acquisition. After the counter value reaches zero, we divide the register value by 2 and signal that the result is ready. This concludes operation of our circuit. Fig.14 presents the resulting flowchart.

When we refer to the number theory will find that division by two is equivalent to the shift of the number to the left. Therefore divider can be implemented with a shift register. Furthermore, we can use our result register in that place. However, it would require an additional state, slowing the system's operation. Instead, we will implement it by connecting seven upper bits of the register to seven lower bits of the output. The most significant bit of the output will be hardwired to zero. This way, a division is eliminated from the above flowchart. The resulting operational unit is shown in Fig.15.

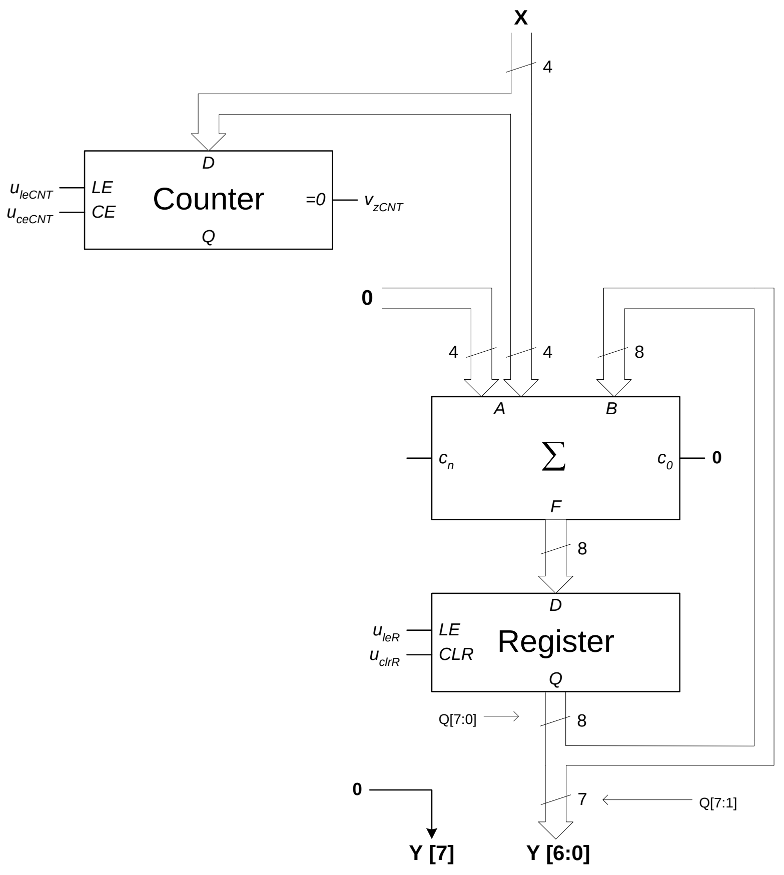
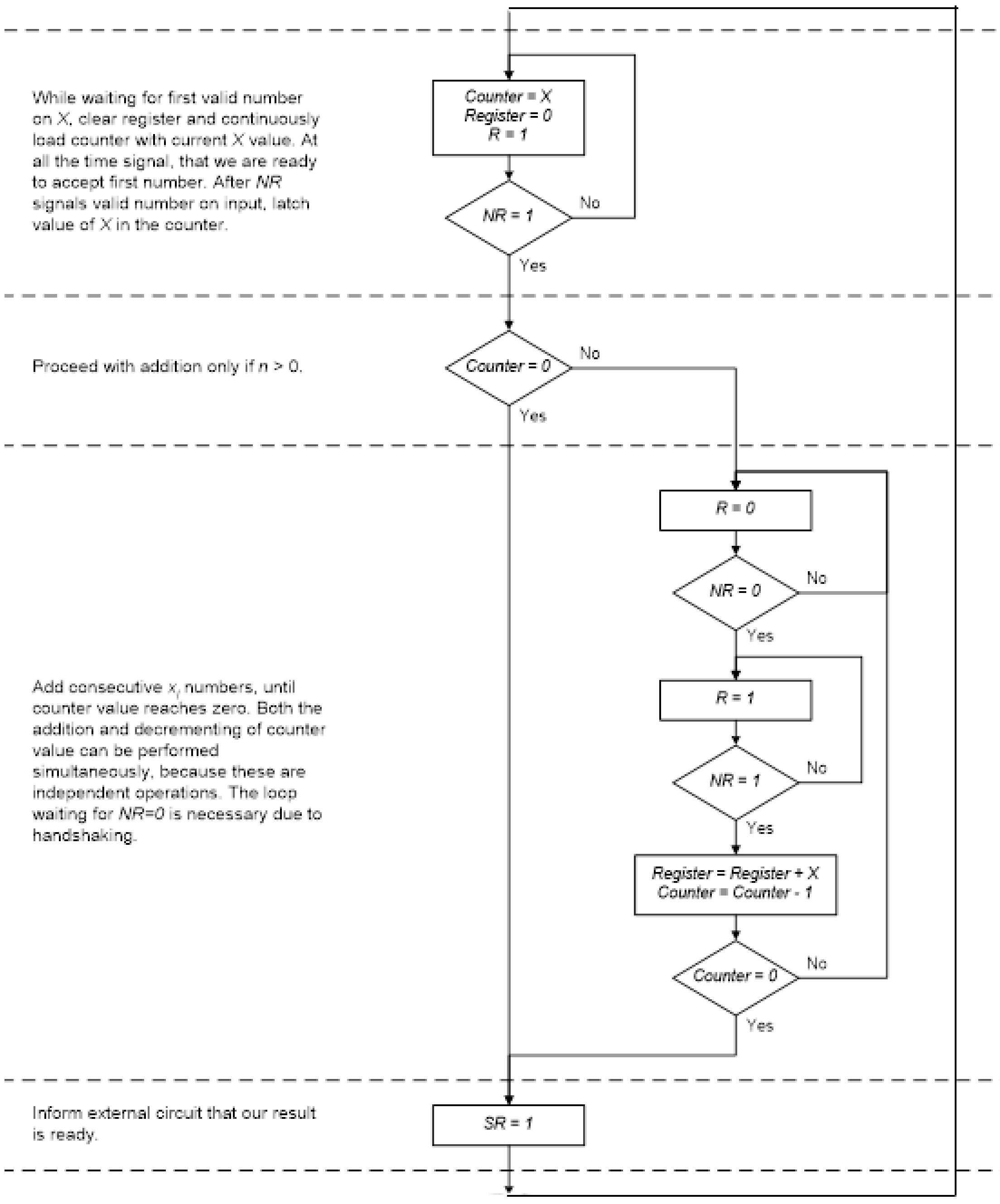


Figure 15

Next, we will continue with the optimization of the control unit. Thanks to the optimizations made to the operational subpart; the division has already been removed from the flowchart. Looking closely at the initial circuit operation, we can see that we can clear the register while waiting for valid data on the input. Furthermore, instead of loading the counter after we receive valid data, we can load it continuously and stop loading as soon as we receive valid data. This way, the counter will constantly monitor *X* and latch its value when *NR (Number Ready)* signal goes high.

Another part of the flowchart worth examining is the one responsible for addition. It is not difficult to figure out that the addition of *X* to the current value of the register is independent of decrementing counter's value. So they can be performed simultaneously.

Still, another part worth redesigning is the one responsible for handshaking. While closely looking at loops responsible for signaling number acquisition, we notice that one of them is redundant. Suppose we place the "number acquisition" loop at the beginning of the part performing addition (instead of its current position after loading the register). In that case, we can eliminate the "number acquisition" loop executed after loading the counter. We will still have correct handshaking.



While waiting for first valid number on X, clear register and continuously load counter with current X value. At all the time signal, that we are ready to accept first number. After NR signals valid number on input, latch value of X in the counter.
Counter = X
Register = 0
R = 1
NR = 1
No
Yes
Proceed with addition only if n > 0.
Counter = 0
No
Yes
R = 0
NR = 0
No
Yes
R = 1
NR = 1
No
Yes
Add consecutive x_i numbers, until counter value reaches zero. Both the addition and decrementing of counter value can be performed simultaneously, because these are independent operations. The loop waiting for NR=0 is necessary due to handshaking.
Register = Register + X
Counter = Counter - 1
Counter = 0
No
Yes
Inform external circuit that our result is ready.
SR = 1

Figure 16

The flowchart resulting from the above optimizations is shown in Fig.16.

The programming flowchart can now be transformed into microprogramming, We replace instructions with microoperation - the corresponding control signals and test results. The resulting flowchart is shown in Fig.17.

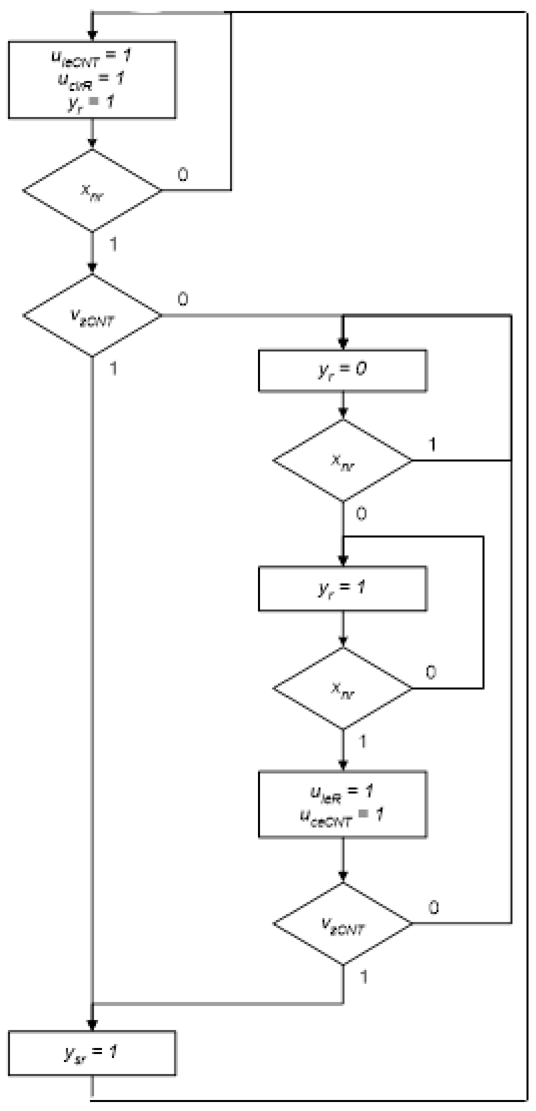


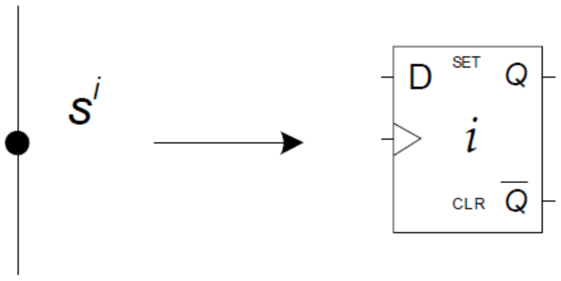
Figure 17

Finally, the microprogramming flowchart can now be transformed into the sequencer,

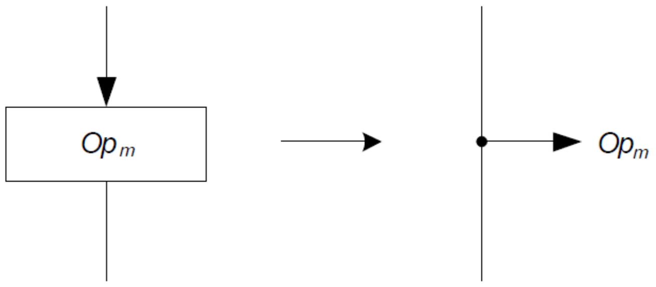
However, we must be aware that the above flowchart will not assure proper work of the device because handshaking is not correctly declared. The message ‘result is ready,’ and the result itself is available only for one clock cycle. The probability that an external unit will inspect our device at that time is infinitely small. Therefore handshaking must be modified by introducing some signal responding to the signal R.

The structure of a sequencer resembles the flowchart – hence process of building the sequencer is quite straightforward. It is only necessary to remember following rules of transformation:

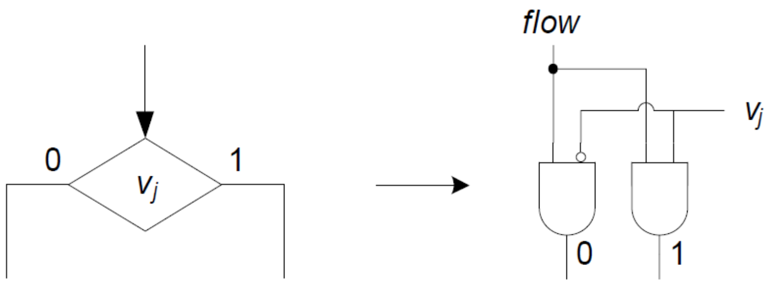
- Each state is assigned a single flip-flop When the circuit enters mentioned state, the flip-flop contains one. When any other state is selected, the flip-flop contains zero.



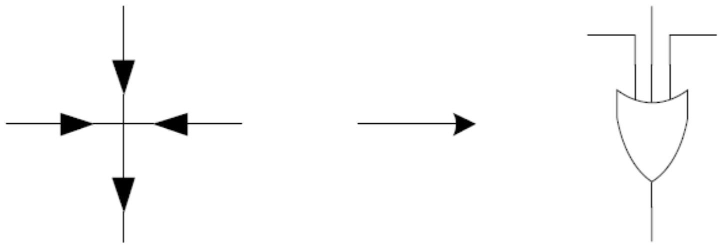
- Each operation is transformed into an output since the operation is executed only when the circuit, operating according to the algorithm described by the flowchart, enters a place containing this operation. In case of the control circuit this means assigning one to the certain signal line, and since this signal is to be delivered to the operational part, an output is required.



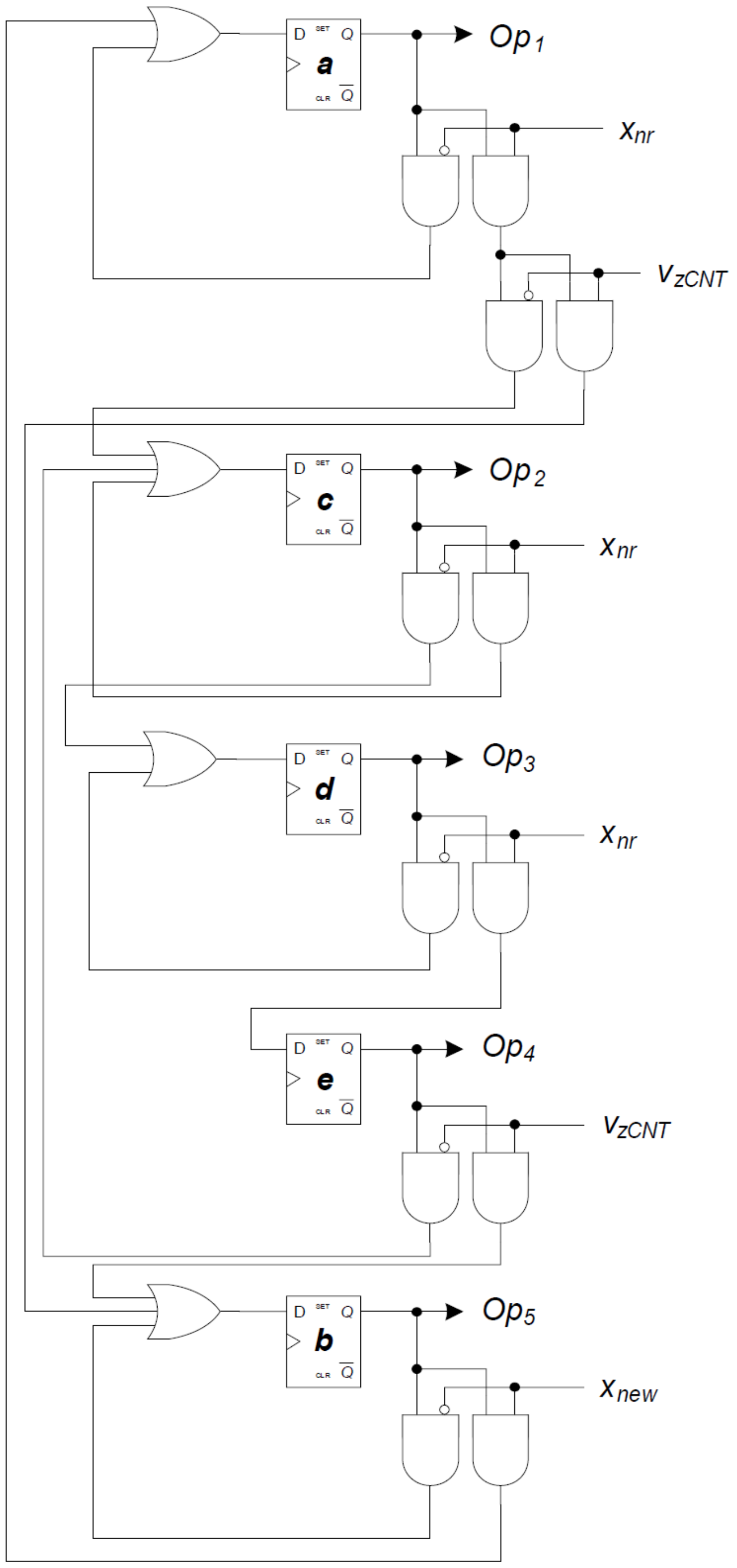
- Each condition is assigned a decision making circuit. It contains two inputs – one of them is the control flow signal, other one is the condition itself. If the control flow reaches the condition (i.e. the first input is high), then the "one" is transferred to one of the outputs, depending on the state of the condition signal. If the control flow has not reached the condition, then both of the outputs are zero.



- Finally, each junction is assigned an OR-gate – it simply passes the signal through.



The sequencer schematic is shown in Fig.18



D SET Q a CLR Q̄
Op1
xnr
vzCNT
D SET Q c CLR Q̄
Op2
xnr
D SET Q d CLR Q̄
Op3
xnr
D SET Q e CLR Q̄
Op4
vzCNT
D SET Q b CLR Q̄
Op5
xnew

Figure 18