Digital Circuits Theory

Classes - Semester 2

Didactic materials

by

Urszula Stańczyk
Asynchronous static flip-flops

- Obtain all describing elements (logic symbol, characteristic function, state diagram, excitation table, characteristic table, transition map, timing chart) for the following flip-flops:
  - $\sim sr$
  - $s \sim r$
  - $\sim sr1$ ($\sim sr$ with dominant set)
  - $\sim sr0$ ($\sim sr$ with dominant reset)
  - $s \sim r1$ ($s \sim r$ with dominant set)
  - $s \sim r0$ ($s \sim r$ with dominant reset)

- For the circuit consisting of a flip-flop or built around such element, complete timing charts for provided changes of input conditions
  - For the flip-flop complete the chart

- For the flip-flop complete the chart

- For the flip-flop complete the chart

- For the flip-flop complete the chart
– For the circuit complete the chart

– For the circuit complete the chart

– For the circuit complete the chart
Flip-flop excitation functions

Find input excitation functions for various flip-flops of $sr$ type for implementation of state variables defined as follows.

\[
D = \left[ \sum (6,9,12,13,14,15)_{dzb} \right]_{dzb} \quad Z = \left[ \sum (0,8,9,12,13,15)_{dzb} \right]_{dzb}
\]

Input excitation functions can be found by various methods. One of them is by using excitation tables of a flip flop. For $sr$ it looks as follows.

\[
\begin{array}{c|cc}
q & s & r \\
0 & 0 & 0 \quad - \\
0 & 1 & 1 \quad 0 \\
1 & 0 & 0 \quad 1 \\
1 & 1 & - \quad 0 \\
\end{array}
\]

The excitation table shows the logic values that should be fed to the inputs of the flip-flop for it to behave in the required manner. In the similar way the state tables show the dependence of the next state on the present state and input states. When each sequence of a present and next state is replaced with the flip-flop input combination that corresponds to such sequence, we obtain tables defining input excitation functions for flip-flops. Firstly from canonical numerical forms defining state variables there are constructed Karnaugh map representations.

Next for all cells in the map there is checked the sequence of present and next state, the sequence is looked up in the excitation table for the flip-flop, and and content of the cell replaced with the corresponding logical values for flip-flop inputs.

The obtained maps define input excitation functions for the two flip-flops required for implementation, thus the only action that still needs to be done is obtaining minimal SoP or PoS form of logic expressions. As the working mode of a flip-flop prevents it from suffering any problems resulting from static hazards occurring on inputs, no anti-hazard groups are required. Typically there are fewer active than inactive states on inputs (states that cause the flip-flops to change its state), thus
we choose the form dictated by these active states, in the case of \( sr \) flip-flop it is SoP.

\[
\begin{align*}
\text{s}_D &= b & \text{r}_D &= \overline{z} \overline{a} \\
\text{s}_Z &= \overline{z} & \text{r}_Z &= \overline{d} \overline{z} + b \overline{a}
\end{align*}
\]

\[
Z = \left\{ \sum (6,9,12,13,14,15,)_{dzba} \right\} \quad D = \left\{ \prod (4,6,14)_{dzba} \right\}
\]

\[
Z = \left\{ \prod (6,9,12,13,14,15,)_{dzba} \right\} \quad D = \left\{ \sum (0,8,9,12,13,15)_{dzba} \right\}
\]

\[
D = \left\{ \prod (6,9,12,13,14,15,)_{dzba} \right\} \quad Z = \left\{ \sum (0,8,9,12,13,15)_{dzba} \right\}
\]
Switching Sequence Table

Using Switching Sequence Table method design an asynchronous circuit with the program given by a timing chart. Present the solution in the form of the logical diagram built with either NAND or NOR gates.

In the first step we need to construct a Switching Sequence Table. The rows of the table correspond to all variables, while the columns list all states of the circuit. The initial state, denoted as "0", there are listed states of variables at the beginning of the working cycle. Then in the following columns all changes in the circuit are marked with symbols of "+" and "−", indication switching either on or off. It is important to remember that in each state only one variable can change its state.

For the example timing chart in the initial state all variables are in their off state. The first change is \( x_1 \) becoming "1", which is indicated in the state 1. This rising edge causes the output to become "1" too, which is shown in the second state. Then \( x_1 \) is switched off in the third state. The next change refers to \( x_2 \) that is switched on in the fourth state, and it results in turning off of the output, indicated in the fifth state. The last change is \( x_2 \) becoming "0", which completes the working cycle and means the return to the initial state, which is why the last column in the table is denoted as "0".

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x_1 )</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( x_2 )</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( Z )</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NCS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

After all changes are recorded, we assign weights to variables, and basing on those weights and recorded on and off conditions, numerical state of the circuit (NCS) is calculated as the sum of products: state of a variable by its weight. As the condition for the output to change its state always occur in the state shown as preceding the state that included the change, when we mark the on-conditions for the variable we need to shift it to the left by one state.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>( Z^0 ) ( x_1 )</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( Z^1 ) ( x_2 )</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( Z^2 ) ( Z )</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NCS</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>4</td>
<td>6</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

The second, and very important step within SST method, is obtaining so-called solvable table. The table is solvable when it does not include contradictory states. The contradictory states are such states that have the same value of NCS and occur for both on- and off-conditions of some state variable. Once the table is constructed it always needs to be checked with respect to solvability. If the table is not solvable,
it needs to be reconstructed in such way that is becomes solvable, which is executed by addition of auxiliary state variables. Only when the table is solvable we can move to the next design stage.

In this case the table is solvable as no NCS is repeated. When NCS are without any repetitions there can be also no contradictions and we can simply read the definition of the function. Typically the functions describing state variables are not completely specified, thus we need to read on-conditions for PoS form, and off-conditions for SoP form. The order of variables listed (as always for canonical numerical form) at the end need to correspond to weights previously assigned to variables.

\[
Z = \sum (1,4,5)_{2x_2x_1} \prod (0,2,6)_{2x_2x_1}
\]

Basing on the obtained definition (in case of more state variables, there are always as many definitions as many variables of this type) it is possible to go to the implementation stage. For the intended implementation with gates, minimal logic expressions for all state variables are required. It can be found by means of Karnaugh maps. Thus we construct K-map for the required number of variables, and fill it in basing on the definition by the canonical form.

<table>
<thead>
<tr>
<th></th>
<th>$x_2x_1$</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>0</td>
</tr>
</tbody>
</table>

Which form of the function, SoP or PoS, should be chosen, can be dictated by type of elements to be used for implementation, by some optimisation indicators, or simply required. In this case SoP form looks as follows.

\[
Z = x_1 + z \overline{x_2}
\]

SoP most often leads to implementation with NAND gates, but firstly we need some transformations. Double complementation is required, then application of DeMorgan’s law. Then the expression is ready for implementation.

\[
Z = \overline{x_1} \overline{z} \overline{x_2} = \overline{x_1} \overline{x_2}
\]
Circuit cycle
Huffman’s method for asynchronous circuits

Using Huffman’s method design an asynchronous circuit with the program given by a timing chart. Execute both stages of state reduction and obtain minimal number of rows in the flow table. Present the solution in the form of the logical diagram built with various logical elements, gates or flip-flops.

The primitive flow table has as many columns as there are combinations of input values, plus one more, dedicated to indicating output state. The number of rows corresponds to the number of stable states, that is states when the circuit waits for inputs to change, as opposed to unstable states that show transitions between stable states and occur as a consequence of the change of inputs.

Each stable state is placed in a separate row, and all respective transitions between stable states shown by unstable states. In the flow map stable states are represented by encircled symbol (most often number), while unstable by just the symbol denoting the stable state. The last row of the table need to include the transition to the initial state, as the working cycle needs to be closed.

<table>
<thead>
<tr>
<th>Present state</th>
<th>$x_1x_2$</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(0)</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td></td>
<td>(1)</td>
<td>1</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>(2)</td>
<td>0</td>
<td></td>
<td>1</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>(3)</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

Once the primitive flow table is obtained, it is then analysed with respect to the possible reduction of states. The first stage of reduction involves reduction of equivalent states, for all types of equivalence. In this case this step is empty, as no equivalent states are present.

In the second phase of reduction compatible states are considered. For that end it is best to construct a merger diagram. The number of nodes required corresponds to the number of states (rows) in the map. They are connected by either continuous or dashed lines, depending on the mergeability to either both Moore and Meally type of machine, or only for Meally automata structure. In the example row 0 is compatible with 3, and row 1 with 2, both for both types of automata, as the outputs in mergeable states are the same.
As a result of merging states a reduced flow table is obtained, containing only two rows.

<table>
<thead>
<tr>
<th>Present state</th>
<th>$x_1, x_2$</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,3</td>
<td>0 0 1 1 0 1</td>
<td>0</td>
</tr>
<tr>
<td>1,2</td>
<td>2 3 1 1 1 1</td>
<td>1</td>
</tr>
</tbody>
</table>

For encoding of a flow map with just two rows a single state variable is sufficient, and there can be no problems resulting from dynamics. Possible encoding is

0,1: $q = 0$

1,2: $q = 1$

Then the binary flow map is as follows.

<table>
<thead>
<tr>
<th>$x_1, x_2$</th>
<th>q</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0 0 1 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1 0 1 1 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The functions describing the circuit need to include some form of expression for the internal state variable $q$ and the output signal $z$.

$Q = x_1 + q \overline{x_2}$

$Z = q$

The circuit can be implemented with either gates or flip-flops.
COMMENT: This timing chart illustrates the work of $sr$ flip-flop. If we treat it as any other asynchronous circuit and proceed through the steps of the synthesis, at the end we should obtain the characteristic functions of the flip-flop, which helps to control correctness of the design process. Unless all steps of reduction process are successfully implemented, the end result will differ from the one that is expected.

COMMENT: This timing chart illustrates the work of $\sim s \sim r$ flip-flop. If we treat it as any other asynchronous circuit and proceed through the steps of the synthesis, at the end we should obtain the characteristic functions of the flip-flop, which helps to control correctness of the design process.
Analysis with respect to races

Analyse the circuit defined by the flow map given with respect to races. Show a way to solve all problems (provide a binary flow map without any races).

\[
\begin{array}{ccccc}
    & x_1 & x_2 & q_1 & q_0 \\
    00 & 10 & 11 & 11 & 00 \\
    01 & 01 & 01 & 01 & 00 \\
    10 & 10 & 10 & 10 & 00 \\
    \end{array}
\]

In the first stage of analysis it is best to clearly mark stable states, as it helps with observation of transitions. Also additional symbols denoting rows are used.

\[
\begin{array}{ccccc}
    & x_1 & x_2 & a & b & c & d & q_1 & q_0 \\
    00 & 10 & 11 & 11 & 00 & 00 \\
    01 & 01 & 01 & 01 & 00 & 00 \\
    11 & 01 & 11 & 11 & 00 & 00 \\
    10 & 10 & 10 & 10 & 00 & 00 \\
    \end{array}
\]

In the first column, for \(x_1 = x_2 = 0\), there are two transitions, from row \(a\) to \(d\), and from \(c\) to \(b\), but both require a change of single state variable thus they are logically adjacent and cannot cause any problem. In the second column, for \(x_1 = 0\) and \(x_2 = 1\), there is a single transition (from \(a\) to \(c\) row) that requires the change of both state variables, which is not logically adjacent. As it happens, regardless of which state variable wins the race, the circuit ends up in the incorrect target state, which means that there is a critical race problem. The third column, for \(x_1 = x_2 = 1\), there are two transitions, one logically adjacent (from \(b\) to \(c\) row) and the other not (from \(a\) row to \(c\) row). The circuit can reach the undesired stable state as the outcome of the race which makes it critical. And finally in the last fourth column, for \(x_1 = 1\) and \(x_2 = 0\), there are three transitions, one of which is not logically adjacent (from \(c\) row to \(a\)). However, no matter which state variable wins the race, the circuit still will end up in the expected state, which means that the race is non-critical.

The transition diagram obtained from this analysis is constructed as follows. For the first column for the current encoding there is no dynamic problem detected, but only because rows \(a\) and \(d\), and \(b\) and \(c\), are logically adjacent. If they were not, then there would be problem of critical race. Which is why in the transition diagram these pairs of rows need to be connected with solid continuous lines.

\[
\begin{array}{c}
    a \bullet \\
    d \bullet \\
\end{array} \quad \begin{array}{c}
    b \bullet \\
    c \bullet \\
\end{array}
\]

For the second column without logical adjacency between rows \(a\) and \(c\) there will be critical race, so these two rows also need to be connected with continuous line.
For the third column, for the current encoding there is a problem of critical race, yet, due to two transitions lead to the same state, which can be used to solve the problem of race by directing transitions. Which means that even though rows $a$ and $b$ need to be connected with $c$, the line used for connection is dashed.

And finally for the last column, since the problem is at most of non-critical race, rows $b$, $c$ and $d$ should be connected with dashed line with the row $a$.

When all these lines are taken into account, and added together, with continuous lines taking precedence over the dashed ones, as the result the transition diagram for the whole circuit looks as follows.

As the continuous lines show these rows that need to be encoded by logically adjacent binary combinations in order to prevent critical races, the number of such lines connecting a row with others is critical with respect to the number of state variables that are required. In this case for all rows the maximal number of such lines is two, which means that preventing races can be executed with two state variables, however, logical adjacencies need to be observed which means a change of the previous encoding is necessary, as it does not support logical adjacency of rows $a$ and $c$. There are several possibilities of encoding, one of them is $a$: 00, $b$:11, $c$: 01, $d$:10. In order to introduce this new encoding without any mistakes, it is best to assign symbols to the stable states and from binary obtain the symbolic flow map, which is then encoded in the regular manner.
Present $x_1 x_2$
state
\begin{tabular}{|c|c|c|c|c|}
\hline
a & 0 & 2 & 5 & (7) \\
\hline
b & (1) & (3) & 5 & 7 \\
\hline
c & 1 & (2) & (5) & 7 \\
\hline
d & (0) & (4) & (6) & 7 \\
\hline
\end{tabular}

Next state

To simplify encoding even further we can change the order of rows in this symbolic flow map.

Present $x_1 x_2$
state
\begin{tabular}{|c|c|c|c|c|}
\hline
a & 0 & 2 & 5 & (7) \\
\hline
c & 1 & (2) & (5) & 7 \\
\hline
b & (1) & (3) & 5 & 7 \\
\hline
d & (0) & (4) & (6) & 7 \\
\hline
\end{tabular}

Next state

\[
\begin{array}{|c|c|c|c|c|}
\hline
q_1 q_0 & 00 & 01 & 11 & 10 \\
\hline
a & 00 & 10 & 01 & 01 & \text{[00]} \\
\hline
c & 01 & 11 & 01 & 01 & \text{[00]} \\
\hline
d & 11 & 11 & 01 & 00 \\
\hline
d & 10 & 10 & 01 & \text{[00]} \\
\hline
\end{array}
\]

\[Q_1 Q_0\]

Then

\[
\begin{array}{|c|c|c|c|c|}
\hline
q_1 q_0 & 00 & 01 & 11 & 10 \\
\hline
a & 00 & 10 & 01 & 01 & \text{[00]} \\
\hline
c & 01 & 11 & 01 & 01 & \text{[00]} \\
\hline
d & 11 & 11 & 01 & 01 \\
\hline
d & 10 & 10 & 01 & \text{[00]} \\
\hline
\end{array}
\]

\[Q_1 Q_0\]

The change of encoding does not affect the non-critical race in the last column. For the solution that is completely race-free, we need to direct the transition which is not logically adjacent through other unstable states that lead to the required target stable state.

\[
\begin{array}{|c|c|c|c|c|}
\hline
q_1 q_0 & 00 & 01 & 11 & 10 \\
\hline
a & 00 & 10 & 01 & 01 & \text{[00]} \\
\hline
c & 01 & 11 & 01 & 01 & \text{[00]} \\
\hline
d & 11 & 11 & 01 & 01 \\
\hline
d & 10 & 10 & 01 & \text{[00]} \\
\hline
\end{array}
\]

\[Q_1 Q_0\]

\[
\begin{array}{|c|c|c|c|c|}
\hline
y_1 y_2 & 00 & 01 & 11 & 10 \\
\hline
y_1 y_2 & 00 & 11 & 01 & \text{[00]} \\
\hline
y_1 y_2 & 01 & 11 & 01 & \text{[00]} \\
\hline
y_1 y_2 & 11 & 11 & 10 & \text{[00]} \\
\hline
y_1 y_2 & 10 & 10 & 10 & \text{[00]} \\
\hline
\end{array}
\]

\[Y_1 Y_2\]
\[
\begin{array}{c|cccc}
    x_1, x_2 & 00 & 01 & 11 & 10 \\
    \hline
    y_1, y_2 & 00 & 01 & 00 & 01 \\
              & 01 & 01 & 00 & 01 \\
              & 11 & 01 & 00 & 11 \\
              & 10 & 01 & 10 & 10 \\
    Y_1Y_2 & & & & \\
\end{array}
\]

\[
\begin{array}{c|cccc}
    x_1, x_2 & 00 & 01 & 11 & 10 \\
    \hline
    y_1, y_2 & 00 & 11 & 00 & 10 \\
              & 01 & 11 & 00 & 11 \\
              & 11 & 11 & 00 & 11 \\
              & 10 & 10 & 00 & 10 \\
    Y_1Y_2 & & & & \\
\end{array}
\]

\[
\begin{array}{c|cccc}
    x_1, x_0 & 00 & 01 & 11 & 10 \\
    \hline
    q_1, q_0 & 00 & 11 & - & 11 \\
              & 01 & 10 & 11 & 01 \\
              & 11 & 10 & 11 & 11 \\
              & 10 & 10 & 11 & 01 \\
    Q_1Q_0 & & & & \\
\end{array}
\]
Synchronous flip-flops

For the circuit consisting of a flip-flop or built around such element, complete timing charts for provided changes of input conditions

- For the flip-flop complete the chart

- For the flip-flop complete the chart

- For the flip-flop complete the chart

- For the circuit complete the chart

- For the circuit complete the chart
State diagrams

- Draw a state diagram for the synchronous sequential circuit detecting that in the sequence of bits sent to the serial input $x$, on any three subsequent bits the middle one is different from its two neighbours and setting in such case the output $Z$ for the time of one clock period.

  The incoming sequence of bits is not divided into words of any size, and for the condition to be satisfied we need to check any three neighbouring bits, regardless of their positioning in the sequence. The satisfied condition should be indicated with the changed output state for one clock period, thus Moore type of automata needs to be chosen. These considerations lead to the state diagram as follows.

  ![State diagram](attachment:state_diagram.png)

  State $S_0$ serves only the purpose of dealing with the first bit of the sequence, later on there is no transition to it. States $S_1$ and $S_2$ are used for cases when there is a group of bits in the sequence that have the same logic value. The loops ensure that the circuit waits in these states for some change on the input. If the logic values of zero and one are encountered interchangeably ($..101010101...$) in the input sequence the output is set as long as this pattern continues.

- Draw a state diagram for the synchronous sequential circuit detecting the pattern "010" anywhere in the sequence of bits sent to the serial input $x$, and setting in such case the output $W$ for the time not longer than one clock period. The sequences cannot overlap - the last bit in the sequence cannot commence the next sequence.

  The sequence is not divided into words, thus the state diagram is allowed to include loops. The sequence to be detected begins with zero and ends with zero. The detection should be indicated on the output for the time of no more than one clock period, which means that the structures for both Moore and Mealy automata are acceptable. The one given below corresponds to Moore automata.
- Draw a state diagram for the synchronous sequential circuit detecting the pattern "101" anywhere in the sequence of bits sent to the serial input $x$, and setting in such case the output $W$ for the time not longer than one clock period. The sequences can overlap - the last bit in the sequence can commence the next sequence.

- Draw a state diagram for the synchronous sequential circuit detecting the pattern "101" anywhere in the sequence of bits sent to the serial input $x$, and setting in such case the output $W$ for the time not longer than one clock period. The sequences cannot overlap - the last bit in the sequence cannot commence the next sequence.

- Draw a state diagram for the synchronous sequential circuit detecting that the 3-bit word sent to the serial input $x$ corresponds to the pattern "101". If the pattern is detected, the output of the circuit should become equal logic 0 for the time equal to one period of the system clock.

- Draw a state diagram for the synchronous sequential circuit checking numbers of logic ones present in any 3-bit word sent to the serial input $x$. If a number of ones is an odd number, then the output $Y$ should become equal 1 for the period of time corresponding exactly to one clock period.

- Draw a state diagram for the synchronous sequential circuit which works as a parallel comparator of two $N$-bit binary numbers, which are fed into two serial inputs $a$ and $b$ from the most significant positions. The circuit should have three outputs corresponding to three possible verdicts of numbers being equal, $E = 1$, of case when $A$ is greater than $B$, $G = 1$, or for $A$ being less than $B$, $L = 1$.

- Draw a state diagram for the serial subtractor of two $N$-bit binary numbers, fed from the least significant positions into two serial inputs $a$ and $b$. 

![State Diagram](image-url)
Counters

- Design a synchronous reverse mod 3 counter with JK flip-flops.

- Design a parallel counter with the working cycle corresponding to 4-bit Gray code. Use T flip-flops.

The counter generating on its inputs 4-bit Gray code requires four flip-flops for the implementation. It is worth noticing that since Gray code is a complete code, as it uses all combinations that can be represented on the given number of bits, the counter with such working cycle cannot have any illegal states.

In the design present state/next state table will be used. The columns on the left present all states that are expected in the counting sequence. To speed up and simplify the design it is best to put them in such order as they occur in the cycle, then for each row the next state is shown by the next row, and for the last row of the table the next one is the first row, as the cycle needs to be closed, as always for sequential circuits. The columns on the right show which combinations of values should be fed to flip-flops to ensure that they behave in the expected manner which means obtaining input excitation functions by referring to excitation tables of the flip-flop. For the T flip-flop the rule is simple - if we want a flip-flop to maintain its current state, we need 0, if the change of state is required, we need 1. Which results in the table as follows.

<table>
<thead>
<tr>
<th>$Q_3^t$</th>
<th>$Q_2^t$</th>
<th>$Q_1^t$</th>
<th>$Q_0^t$</th>
<th>$T_3^t$</th>
<th>$T_2^t$</th>
<th>$T_1^t$</th>
<th>$T_0^t$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Sometimes in the present state/next state tables we can observe some simple relationships between the input and outputs that make application of minimisation procedures unnecessary, but most often basing on the table Karnaugh maps for outputs are created.
As there are significantly fewer 1s than 0s, the forms chosen for descriptions are SoP. However, it is clear from these maps that actually no minimisation for these forms are possible, as all 1s have no logically adjacent neighbours. The expressions defining functions of inputs for flip-flops are as follows.

\[
T_3' = \overline{Q_3} Q_2 Q_1' Q_0' + Q_3' \overline{Q_2} Q_1 Q_0
\]

\[
T_2' = \overline{Q_3} Q_2 Q_1' Q_0' + Q_3' Q_2^t Q_1 Q_0
\]

\[
T_1' = \overline{Q_3} Q_2 Q_1 Q_0' + Q_3' Q_4 Q_1' Q_0 + \overline{Q_3} Q_2 Q_1 Q_0' + Q_3' Q_2 Q_1 Q_0
\]

For the least significant output bit the form of the map immediately suggest much simplified implementation with XNOR type of operator and gate (or XOR and complementation).

\[
T_0' = Q_3' \oplus Q_2' \oplus Q_1' \oplus Q_0'
\]

With this form in mind we can transform the previously obtained expressions.

\[
T_3' = Q_3' \overline{Q_0} (Q_3' Q_2 + Q_3 Q_2') = Q_3' \overline{Q_0} (Q_3' \oplus Q_2') = Q_3' \overline{Q_0} Q_3' \oplus Q_2'
\]

\[
T_2' = Q_3' \overline{Q_0} (Q_3' Q_2 + Q_3 Q_2') = Q_3' \overline{Q_0} (Q_3' \oplus Q_2')
\]

\[
T_1' = Q_0' (Q_3' Q_2 Q_1' + Q_3 Q_2 Q_1') + Q_3' Q_2 Q_1' + Q_3 Q_2 Q_1' = Q_0' (Q_3' \oplus Q_2' \oplus Q_1')
\]

- Design a parallel adding and subtracting counter mod 3. The working cycle starts with the lowest value, which is then increased by one in each clock period till reaching the maximum value, after which it begins decreasing by one with each period. The working cycle end upon generating the lowest value. Use D flip-flops for implementation.

- Design a serial counter mod 8. Use JK flip-flops for construction of the circuit. Draw timing charts for flip-flops triggered by the positive and negative edge of the system clock signal. Draw conclusions with respect to the resulting working cycle.

- Design a parallel programmable counter. For the value of the programming signal \( p = 0 \) the counter should be adding mod 5 in Watts’ code, for the programming signal \( p = 1 \) the counter is subtracting mod 7 in Excess-3 code. Compare results of implementation with JK flip-flops, with those for T and D flip-flops.
Illegal state recovery

- Design a synchronous illegal state recovery for the reverse mod 3 counter with JK flip-flops

- Design an asynchronous illegal state recovery for the reverse mod 3 counter with JK flip-flops

- Design a synchronous illegal state recovery for the parallel counter with the working cycle corresponding to Excess3 code with T flip-flops

- Design an asynchronous illegal state recovery for the parallel counter with the working cycle corresponding to Excess3 code with JK flip-flops

- Design a synchronous illegal state recovery for the parallel counter with the working cycle corresponding to BCD Gray+3 code with JK flip-flops

As BCD Gray+3 code uses 10 binary combinations out of 16 that can be represented on 4 bits, it means that there are 6 illegal states. In synchronous illegal state recovery for all illegal states there are fed such combinations on the synchronised information inputs for flip-flops which will cause the elements to respectively change or maintain their states, in a way that leads the circuit to the legal state, typically the first in the counting cycle.

In the present state/next state table the data for the information inputs for legal states is prepared as explained before. The legal states are usually separated in the table from illegal states. For all illegal states such combinations are chosen that lead to the chosen legal state.

<table>
<thead>
<tr>
<th>$Q_3^t$</th>
<th>$Q_2^t$</th>
<th>$Q_1^t$</th>
<th>$Q_0^t$</th>
<th>$J_{3}^t K_3^t$</th>
<th>$J_{2}^t K_2^t$</th>
<th>$J_{1}^t K_1^t$</th>
<th>$J_{0}^t K_0^t$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

The corresponding Karnaugh maps for all inputs are as follows.
After minimisation from these maps for all synchronised information inputs of flip-flops logical expressions for SoP forms are as follows.

\[ J'_3 = Q'_3 \overline{Q'}_2 \overline{Q'}_0 \]

\[ K'_3 = \overline{Q'}_2 \]

\[ J'_2 = \overline{Q'}_3 Q'_1 \overline{Q'}_0 \]

\[ K'_2 = Q'_3 Q'_1 \overline{Q'}_0 \]

\[ J'_1 = \overline{Q'}_3 + Q'_3 Q'_0 \]

\[ K'_1 = \overline{Q'}_3 Q'_2 Q'_0 \]

\[ J'_0 = Q'_3 Q'_2 \overline{Q'}_1 + \overline{Q'}_3 Q'_2 Q'_1 \]

\[ K'_0 = \overline{Q'}_2 + \overline{Q'}_3 \overline{Q'}_1 + Q'_3 Q'_1 \]

- Design an asynchronous illegal state recovery for the parallel counter with the working cycle corresponding to BCD Gray+3 code with JK flip-flops.

The difference in design with respect to the solution presented above lies only in the illegal state recovery mechanism, which is supposed to be asynchronous. Therefore, for all illegal states the combinations for synchronised inputs of flip-flops are are replaced with don't care conditions, and to the preset state/next state table there is added the third group of columns, dedicated to asynchronous inputs to the flip-flops. When in legal states, they should remain inactive, which for typical implementation means equal to "1", if they are active with low level. They become active only for illegal states, and as before we choose such combinations that from illegal states direct to the chosen legal state of the counter, usually the first in the counting cycle.
| $Q_4' Q_2 Q_1' Q_0'$ | $J_3' K_3'$ | $J_2' K_2'$ | $J_1' K_1'$ | $J_0' K_0'$ | $\sim s_3 \sim r_3 \sim s_2 \sim r_2 \sim s_1 \sim r_1 \sim s_0 \sim r_0$ |
|----------------------|----------------------|----------------------|----------------------|----------------------|----------------------|----------------------|----------------------|
| 0 0 1 0              | 0 - 1 - 0 0 -       | 1 1 1 1             | 1 1 1 1             | 1 1 1 1             |
| 0 1 1 0              | 0 - 0 - 0 1 -     | 1 1 1 1             | 1 1 1 1             | 1 1 1 1             |
| 0 1 0 1              | 0 - 0 0 - 1 -   | 1 1 1 1             | 1 1 1 1             | 1 1 1 1             |
| 0 1 0 0              | 1 - 0 0 0 -   | 1 1 1 1             | 1 1 1 1             | 1 1 1 1             |
| 1 1 0 0              | - 0 - 0 1 -   | 1 1 1 1             | 1 1 1 1             | 1 1 1 1             |
| 1 1 0 1              | - 0 1 - 0 -   | 1 1 1 1             | 1 1 1 1             | 1 1 1 1             |
| 1 1 1 1              | - 0 - 0 - 1 - | 1 1 1 1             | 1 1 1 1             | 1 1 1 1             |
| 1 1 1 0              | - 0 - 1 0 0 - | 1 1 1 1             | 1 1 1 1             | 1 1 1 1             |
| 1 0 1 0              | - 1 0 - 0 0 - | 1 1 1 1             | 1 1 1 1             | 1 1 1 1             |
| 1 0 1 1              | -- -- -- --   | 1 0 1 1             | 1 0 1 1             | 1 0 1 1             |
| 1 0 0 1              | -- -- -- --   | 1 0 1 1             | 1 0 1 1             | 1 0 1 1             |
| 1 0 0 0              | -- -- -- --   | 1 0 1 1             | 1 0 1 1             | 1 0 1 1             |
| 0 0 0 0              | -- -- -- --   | 1 0 1 1             | 1 0 1 1             | 1 0 1 1             |
| 0 0 0 1              | -- -- -- --   | 1 0 1 1             | 1 0 1 1             | 1 0 1 1             |
| 0 0 1 1              | -- -- -- --   | 1 0 1 1             | 1 0 1 1             | 1 0 1 1             |

It may seem that in some cases instead of inactive states for asynchronous inputs we could use the form with don’t care conditions instead, yet it is not the correct approach. Don’t care conditions allow for activity on inputs depending on their inclusion in either groups of 1s or 0s. They would not cause an illegal state of the counter, but they would make it to change not in synchronous, but asynchronous manner, which is contrary to the main working mode. Asynchronous reactions are allowed for one purpose only: to change illegal counter state into legal without the necessary wait for the system clock. For correct counter states illegal state recovery mechanism should not be activated.

Basing on the present state/next state table it is a simple matter to obtain defining expressions for all inputs, synchronised, and asynchronised.
Frequency dividers

- Design a synchronous frequency divider with the filling of the signal equal to $3/5$. For construction of the circuit use T flip-flops.

- Design a programmable synchronous frequency divider, with the output signal defined by the table below, where $p$ and $q$ are programming signals, and $f$ the frequency of the input system clock. Ensure filling of the output signal $W$ equal $1/2$. Use JK flip-flops.

<table>
<thead>
<tr>
<th>$p$</th>
<th>$q$</th>
<th>$W = g(f)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>$f/2$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>$f/4$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>$f/6$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>$f/8$</td>
</tr>
</tbody>
</table>

- Design a programmable asynchronous frequency divider, with the output signal defined by the table below, where $p$ and $q$ are programming signals, and $f$ the frequency of the input system clock. Ensure filling of the output signal $W$ equal $1/2$. Use T, D, and JK flip-flops, all three types in one design. Draw a realistic timing chart (taking into account propagation time of all elements) illustrating the work of the circuit.

<table>
<thead>
<tr>
<th>$p$</th>
<th>$q$</th>
<th>$W = g(f)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>$f/2$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>$f/4$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>$f/6$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>$f/8$</td>
</tr>
</tbody>
</table>

- Design a programmable synchronous frequency divider by 4, which for the value of the programming signal $p$ equal zero guarantees the smallest possible filling of the output signal, while for $p = 1$ the filling is the largest possible. Employ JK flip-flops in construction of the circuit.
Microrprogrammable circuits

Design as an microrprogrammable circuit in all structures the automaton working accordingly to the state diagram given. Provide the list of microinstructions and the corresponding logical diagram. Consider all solutions fro any structure, without optimisation and with the smallest memory without adding any auxiliary logical elements.

The state diagram includes four states. For Mealy machine it would be possible to reduce state S2 with state S3, but states S0 and S1 cannot be reduced. Which means that even with reduction still two state variables would be necessary. For Moore machine no reduction of states is possible, and also two state variables are required for encoding. This version is therefor chosen for implementation.

We can notice that for each state the values expected on the outputs are different, which means that we can use them for encoding and it will simplify the structure and reduce the size of the memory needed. That is why we assign:

S0: $Q_1 = Q_0 = 1$
S1: \( Q_1^t = 1 \) \( Q_0^t = 0 \)
S2: \( Q_1^t = Q_0^t = 0 \)
S3: \( Q_1^t = 0 \) \( Q_0^t = 1 \)

And

\( W_1 = Q_1^t \) \( W_2 = Q_0^t \)

With this assignment for the universal structure of the microprogrammable circuit, the size of the memory element needed equals the number of state variables plus the number of inputs, which gets 4 address inputs, by the number of state variables to be stored in each memory cell, which is 2.

The memory content corresponding to this structure and assigned encoding is as follows.

<table>
<thead>
<tr>
<th>( Q_1^t )</th>
<th>( Q_0^t )</th>
<th>( x_1 )</th>
<th>( x_2 )</th>
<th>( Q_1^{t+1} )</th>
<th>( Q_0^{t+1} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>1 1</td>
<td>1 1</td>
<td></td>
</tr>
<tr>
<td>0 1</td>
<td>0 1</td>
<td>1 0</td>
<td>1 0</td>
<td>1 0</td>
<td></td>
</tr>
<tr>
<td>1 0</td>
<td>1 1</td>
<td>0 1</td>
<td>0 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1</td>
<td>0 0</td>
<td>1 1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

On the other hand, we can notice that in transitions between states always one out of two input variables is checked, which means that we could use to the advantage the structure with conditional multiplexer. The number of bits in a cell would remain unchanged, while the number of address inputs would be decreased by 1.
Test

1. Using Switching Sequence Table method design an asynchronous circuit with the program given by a timing chart. Present the solution in the form of the logical diagram built with either NAND or NOR gates:

   (a) 

   (b)

2. Using Huffman’s method design an asynchronous circuit with the programme given by a timing chart, obtaining the minimal number of rows in the flow table. Present the solution in the form of the input excitation functions for either sr or sr flip-flops and a logical diagram:

   (a) 

   (b)

3. Analyse the circuit defined by the flow map given with respect to races. Show a way to solve all problems (provide a binary flow map without any races).

   \[
   \begin{array}{cccc}
   x_1 x_2 & 00 & 01 & 11 & 10 \\
   y_1 y_2 & 00 & 01 & 00 & 00 & 01 \\
   01 & 01 & 00 & 10 & 01 & \\
   11 & 01 & 00 & 11 & 10 & \\
   10 & 01 & 10 & 10 & 10 & \\
   \end{array}
   \]
4. Draw a graph for a synchronous circuit which detects that

(a) a binary number constructed out of 2-bit words entered on the serial input $x$ (where the first bit is considered MSB and the second LSB), is an odd number. Detection should cause setting the output $W = 0$ for exactly one period of a clock signal.

(b) in 3-bit words entered on a serial input $x$ the number of zeros is greater than zero, and it is an even number. Detection should cause setting the output $W = 1$ for the time not longer than one clock period.

5. Design a synchronous circuit working as

(a) parallel programmable counter, which for a programming signal $p = 0$ counts in the code $3 \rightarrow 0 \rightarrow 1$, for $p = 1$ in the code $2 \rightarrow 3 \rightarrow 1$. Ensure the synchronous illegal state recovery. Present the solution in the form of the input excitation functions and a logical diagram for T flip-flops.

(b) a programmable frequency divider, which for a programming signal $p = 0$ gives a signal divided by 6 with the greatest possible filling of the signal, and for $p = 1$ divides by 3 with the smallest possible filling. Ensure the asynchronous illegal state recovery. Present the solution in a form of the input excitation functions and a logical diagram for JK flip-flops.

6. Design a microprogrammable circuit working accordingly to the graph given. Write the contents of the memory and draw a logical diagram. Find the solution with the smallest memory without adding any auxiliary logical elements.
Bibliography


