ZEBRA

CONTENTS

Why Zebra?

Zebra is a language dedicated to the simulation of **digital electronics circuit and interfacing them with real breadboard circuits**. Zebra will help us achieve the precise results of digital hardware using conventional coding. With unlimited digital resources, we can construct any circuit we desire without having to worry about finding actual hardware components. Moreover, we can connect them out to real breadboard circuits and run them together seamlessly. Thus, shortage of electronics components for breadboard circuits will never be an issue. Zebra breaks down the boundary between the real and virtual circuits. Whether we are trying to learn digital electronics or trying to bring our favourite circuit to life, Zebra will always have our back.

Features of Zebra:

>Zebra is a procedure oriented compiler-interpreter language

>It allows a host of functionalities with commands like !PLOT, !PRINT, !SCAN etc.

>It has a huge collection of predefined subcircuits to select from and use in our code. This inspires modular programming approach and allows code reusability

>Zebra breaks down circuits in terms of fundamental gates and evaluates each of them at bitwise level much like the physical gates do

>As such Zebra can also connect virtual circuits to breadboard ICs and other components, via serial communication and run them together without us having to worry about the boundary between real and virtual components

Starting out with Zebra:

>Requirements (the Zebra installer installs these):

- # Python 2.7 required
- # Python Launcher
- # packages numpy, matplotlib and pyserial (for serial communication)

>How to install Zebra:

- # double click on the file Zebra_<version number>_<system bit (32 or 64)>.exe
- # select the location where you wish to have zebra installed
- # try not to install in system drive(C drive). If installed in C drive then cmd needs to be started as administrator. So better not to install in C drive.
- # proceed with the installation
- # always install python with Zebra, whether or not python is already present in system
- # open the folder Zebra at the install location

>After opening Zebra, note the following files and folders:

- # **SUBCIRCUITS_INBUILT** : This folder houses the predefined subcircuit files.
- # **SUBCIRCUITS_USER_DEFINED** : All the user defined subcircuits are created inside this folder for temporary usage only. We will come to this point later.
- # **IC74XX** : This folder holds the predefined 74XX IC circuit files.
- # **pkgs :** This folder contains the scripts to run Zebra.
- # **ARDUINO_BOARD_INFO :** Contains different atmega board data.
- # **arduinocomtest0 :** Houses the script that needs to be uploaded to the microcontroller for interfacing the virtual circuits to external circuits.
- # **bin :** This folder contains the zebra executables.
- # **zebra uninstaller :** This uninstalls zebra from the system.
- # **zebra icon :** Zebra icon file.
- # **Zebra.launch.py :** Zebra launcher.
- # The rest are setup files and must not be deleted or moved.

How to start writing circuits in Zebra:

>Open any directory.

>Create a new text file, name it say test_circuit.txt, open it and try out the following piece of code:

```
*This code simulates the working of a three input AND gate
*nodes 0, 1 and 2 are the input nodes
*node 3 is the output node
!MAINCKT
AND_2    0          1          4
AND_2    2          4          3
!FIX_VOLTAGE       0          1
!FIX_VOLTAGE       1          0
!FIX_VOLTAGE       2          1
!END_MAINCKT       OT_ANALYSIS
!PRINT     0          1          2          3
```

>Note the following facts about the above code:

All the lines beginning with the character '*' are ignored by Zebra, and are referred to as comment lines.
Zebra is case sensitive, i.e. capital and small letters are viewed differently by Zebra.
All the lines enclosed by the statements !MAINCKT and !END_MAINCKT constitute the real circuit that will be simulated.
The character '!' before a keyword establishes it as a command.
AND_2 is a fundamental gate used in here. Following are the list of all fundamental gates which can be used inside a circuit description without subcircuit call:

~ ***AND_2*** - two input AND gate
AND_2 <input A> <input B> <output O>

~ ***OR_2*** - two input OR gate
OR_2 <input A> <input B> <output O>

~ ***NOT*** - NOT gate
NOT <input A> <output O>

~ ***NAND_2*** - two input NAND gate
NAND_2 <input A> <input B> <output O>

~ ***NOR_2*** - two input NOR gate
NOR_2 <input A> <input B> <output O>

~ ***XOR_2*** - two input XOR gate
XOR_2 <input A> <input B> <output O>

~ ***XNOR_2*** - two input XNOR gate
XNOR_2 <input A> <input B> <output O>

~ ***BUFFER*** – buffer gate
BUFFER <input> <output>

~ ***TRISTATE_BUFFER*** – tristate buffer gate
TRISTATE_BUFFER <enable> <input> <output>

The command !FIX_VOLTAGE as the name suggests defines the starting voltage of a node. The first argument being the name of the node and the next being the voltage that needs to be defined at that node.
Note the keyword immediately after !END_MAINCKT. It defines the type of analysis that we wish to carry out on our circuit. Zebra allows 4 types of analysis :

- ~ **OT_ANALYSIS** : refers to one time analysis. The circuit is simulated once with the predefined node values. By default all circuit nodes are defined with state 0.
- ~ **RT_ANALYSIS** : refers to run time analysis. Simulates the circuit in real time. Changes can be made to the circuit inputs in real time, which affects the circuit in real time.
- ~ **TT_ANALYSIS** : refers to truth table analysis. Performs RT_ANALYSIS with every possible combination of the input node values, or with provided combination of input node values at specified time values.
- ~ **TIME_ANALYSIS** : refers to time analysis. This analysis is much like the RT_ANALYSIS but, in this case no changes can be made in real time. Moreover, instead of carrying out the analysis in real time, the simulation can be carried out for a fixed time interval.

!PRINT is used to print the voltages at the desired nodes after OT_ANALYSIS takes place. The arguments are the desired node names.

Executing circuit scripts in Zebra:

type cmd in the address bar of the current directory and press Enter

at the cmd prompt, type: *zebra <file_name>.txt, i.e. zebra test_circuit.txt*; press Enter

if Zebra fails to find the packages matplotlib, numpy or pyserial in the system, it will ask the user if he/she wants to install the packages, enter y to proceed with the installation. This installation requires an internet connection. You can also reject the installation. This will not disrupt execution of circuits. Only commands like !PLOT will not be available.

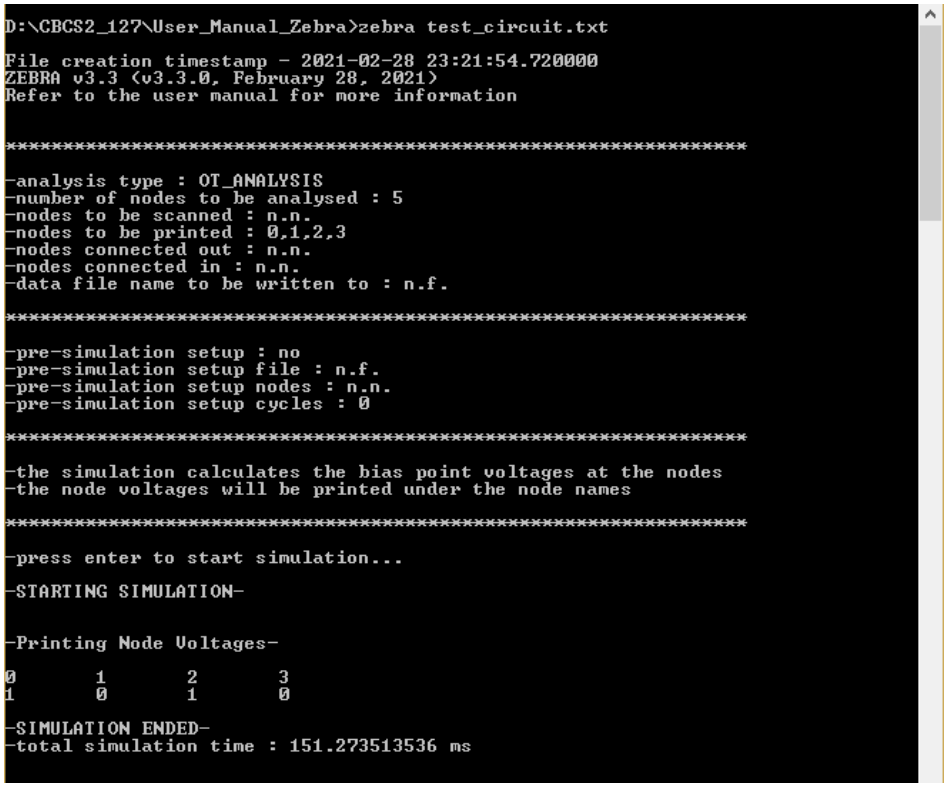

Figure 1

The figure shows the output generated for the script test_circuit.txt. The first section gives the file creation time and the installed Zebra version.
The second section gives the type of analysis carried out on the circuit, and other general information about the script.
The third section provides information about the circuit pre-simulation.
The next section summarizes purpose of the analysis type.

The simulation is started following the previous section and the node voltages are printed for OT_ANALYSIS.
Following this, the total simulation time is displayed.

Inside Zebra

Zebra is a compiler interpreter language. Much like java, Zebra first expands the code into a fundamental form. This form of circuit is in terms of the fundamental gates only. Zebra then compiles the expanded circuit as a whole to perform the simulation. While expanding the circuit, the ***local nodes*** are assigned unique values in the form of ***real nodes*** inside the expanded circuit. This ensures multiple usage of same subcircuits inside the main circuit construct.

There are two main parts in which Zebra simulation takes place :

- Pre-simulation : This is part of simulation takes place before the main simulation. Commands !FIX_VOLTAGE and !MAINCKT_SETUP construct takes place under pre-simulation. Zebra clock is disabled during circuit pre-simulation. The pre-simulation is often used to set the circuit for main simulation. For example, we may want clear counters and registers in the pre-simulation before they can be used in the main simulation.
- Main-simulation : As the name suggests this is the main simulation which is carried out as per the specified analysis.

The different processes in Zebra are carried out in separate threads, ensuring real time simulation for the circuits. Zebra communicates with external circuits via serial communication as a separate thread on its own. Delays of the order of 10 microseconds can be expected during serial communication. More details on serial interfacing are provided in the section **Interfacing Virtual and Breadboard circuits using atmega328P.**

Following this chapter, we will now dive into the formalities of Zebra.

Nodes

Nodes are the points in a circuit through which a circuit element is connected to the circuit. Connections in circuits are considered ideal wires with negligible resistance, so a node may consist of the entire section of wire between elements not just a single point.

For example, a two input AND gate has 2 input nodes and one output node as shown,

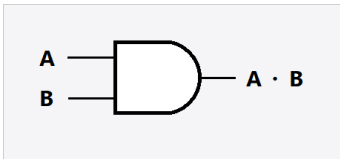

Figure 2

Nodes in Zebra can be defined using the following construct:
ND <node_name_1> <node_name_2> …
The ND construct can be used multiple times as and when required. But nodes must be defined within a circuit description construct, i.e. within the !MAINCKT or the !SUBCKT construct. Node definition is analogous to identifying a particular hole on a breadboard.

A node is defined as well as declared whenever it is used as input or an output to a gate or a subcircuit. For example, for figure 2, we can have,
AND_2 0 1 2
where, 0 and 1 are the input nodes and 2 is the output node. Declaring a node is analogous to connecting an identified breadboard hole to a gate, circuit or an IC input or output.

The node names in Zebra must satisfy the following:
- The node names must be integers.
- The node names must be greater than or equal to 0.
- Any other choice for the node names will result in a Zebra error.

During simulation, Zebra **calculates the voltage at each individual node** by evaluating the inputs to the gate it is output to.

Voltages at all the nodes are **by default set to 0** at the beginning of simulation, unless specified using !FIX_VOLTAGE, or the !MAINCKT_SETUP construct (at pre-simulation)

The node voltages can be set at run time using !SCAN, !CLOCK or by using TT_ANALYSIS input construct.

Nodes which are **not output to any gate** are not evaluated by Zebra. Their values **remain same** during circuit simulation, unless set during simulation using !SCAN or !CLOCK.

Nodes are of two types:
- **local nodes** : these are the nodes local to a circuit (whether be subcircuit or maincircuit)
- **real nodes** : these are the real nodes whose values are evaluated during the circuit simulation

Whenever a subcircuit is invoked inside a maincircuit, the local nodes are **named uniquely** which now **become the real nodes**. Only the **relative position of the local nodes remain preserved** which is all we need. As such every time a subcircuit is expanded inside the maincircuit, new node numbers are allotted to the subcircuit description inside the maincircuit, keeping the relative configuration of the local nodes same.

As such the **local node names have nothing to do with the real node names**. Only the order of the functionality of the nodes matter. Thus when invoking subcircuits inside the maincircuit, the **order of the functionality of the nodes** needs to be kept in mind.

Before passing nodes as arguments into commands like !PRINT, !SCAN, FIX_VOLTAGE etc., the **nodes must be defined with respect to some gate** in the circuit. Not doing so will result in error.

!Commands

Zebra supports a number of commands. Each of these commands perform a specific task. Commands in Zebra are identified by their preceding '!' sign. Zebra identifies any word with an '!' in front as a command. The table below lists all the Zebra commands:

COMMANDS	**FUNCTIONALTY**
!MAINCKT	Marks the beginning of main circuit description. *Syntax : !MAINCKT*
!END_MAINCKT	Marks the end of main circuit description. *Syntax : !END_MAINCKT <analysis type>*
!SUBCKT	Declares and defines user-defined subcircuits. *Syntax : !SUBCKT <subcircuit name> <node 1> <node 2> . . .*
!END_SUBCKT	Marks the end of user-defined subcircuit description. *Syntax : !END_SUBCKT*
!PRINT	Prints the node voltages at the cmd terminal screen. *Syntax : !PRINT <time(keyword)>(optional) <node 1> <node 2> …*
!PRINTF	Prints the node voltages to a file. *Syntax : !PRINTF <time(keyword)>(optional) <node 1> <node 2> …*
!SCAN	Scans the node voltages from the cmd terminal. *Syntax : !SCAN <node 1> <node 2> <node 3> . . .*
!PLOT	Plots the node voltage data against simulation time. *Syntax : !PLOT <node 1> <node 2> <node 3> . . .*
!FIX_VOLTAGE	Defines the voltage data at a particular node at pre-simulation. *Syntax : !FIX_VOLTAGE <node name> <voltage value>*
!CLOCK	Defines clock input to a particular node. *Syntax : !CLOCK <node name> <offset time> <on time> <off time> <starting clock value>*
!MAINCKT_SETUP	Sets up node values for the pre-simulations. *Syntax : !MAINCKT_SETUP <node 1> <node 2> <node 3> . . .*
!END_MAINCKT_SETUP	Marks end of main circuit setup construct. *Syntax : !END_MAINCKT_SETUP <input file name>(optional)*
!WRITE_TO_FILE	Opens a writable file in which the analysis data can be written. *Syntax : !WRITE_TO_FILE <file name with extension>*
!SET_SCAN_SAMPLING_TIME	Sets the rate at which serial communication between Zebra and external circuits takes place. *Syntax : !SET_SCAN_SAMPLING_TIME <time in milliseconds>*
!CONNECT_OUT_ARDUINO	Defines the interfacing between the internal and external nodes and their functionality: Input or Output nodes. *Syntax : !CONNECT_OUT_ARDUINO <board name> <port name>*
!END_CONNECT_OUT_ARDUINO	Marks the end of !CONNECT_OUT_ARDUINO construct. *Syntax : !END_CONNECT_OUT_ARDUINO*

*the last three commands are to be described in the section **Interfacing Virtual and Breadboard circuits using atmega328P.**

A detailed description for each such command is given below:

!MAINCKT – marks the beginning of main circuit description.

!END_MAINCKT – marks the end of main circuit description.
- Anything enclosed between !MAINCKT and !END_MAINCKT is treated as a part of the main circuit construct.
- The command is followed by the analysis type to be carried out on the circuit.
- Syntax : !END_MAINCKT <analysis type>
- We can have a single maincircuit in one script.
- Declaring multiple !MAINCKT constructs would simply merge them together.

```
1  *This is a half adder circuit
2  !MAINCKT
3  XOR_2  0  1  2
4  AND_2  0  1  3
5  !FIX_VOLTAGE  0  1
6  !FIX_VOLTAGE  1  1
7  !END_MAINCKT OT_ANALYSIS
```

Figure 3

!SUBCKT – declares and defines user-defined subcircuits.
- Marks the beginning of subcircuit body.
- Takes the first argument as the subcircuit name.
- The next arguments are the nodes using which the subcircuit connects to the maincircuit.
- Syntax : !SUBCKT <subcircuit name> <node 1> <node 2> . . .

!END_SUBCKT – marks the end of user-defined subcircuit description.
- Anything enclosed between !SUBCKT and !END_SUBCKT is treated as a part of the subcircuit construct.
- We can have multiple user-defined subcircuits inside a single script.

```
1  !SUBCKT  XOR_3  0  1  2  3
2  XOR_2  0  1  4
3  XOR_2  4  2  3
4  !END_SUBCKT
```

Figure 4

!PRINT – prints the node voltage or simulation real time in the cmd terminal screen during or after simulation.
- Takes the node names as the arguments whose voltages are to be printed.
- Can also take the word ***time*** as an argument. This prints the real time during simulation.
- Syntax : !PRINT <node 1> time(optional) <node 3> . . .
- !PRINT can be used anywhere inside the code as long as its arguments are defined before.
- !PRINT works with OT_ANALYSIS, RT_ANALYSIS, TT_ANALYSIS and TIME_ANALYSIS to print the data in cmd terminal.

```
1  *This is a half adder circuit
2  !MAINCKT
3  XOR_2  0  1  2
4  AND_2  0  1  3
5  !END_MAINCKT OT_ANALYSIS
6  !SCAN  0  1
7  !PRINT  0  1  2  3
```

Figure 5

!PRINTF – prints the node voltages to an opened file after simulation.

- Takes the node names as the arguments whose voltages are to be printed.
- Can also take the word ***time*** as an argument. This prints the real time during simulation.
- Syntax : !PRINT <node 1> time(optional) <node 3> . . .
- !PRINTF can be used anywhere inside the code as long as its arguments are defined before.
- !PRINTF works with OT_ANALYSIS, RT_ANALYSIS, TT_ANALYSIS and TIME_ANALYSIS to print the data in cmd terminal.

```
1  *This is a half adder circuit
2  !MAINCKT
3  XOR_2  0  1  2
4  AND_2  0  1  3
5  !END_MAINCKT OT_ANALYSIS
6  !WRITE_TO_FILE data.txt
7  !SCAN  0  1
8  !PRINTF  0  1  2  3
```

!SCAN – scans node voltage from the cmd terminal screen during or before simulation.
- Takes the node names as the arguments whose voltages are to be scanned.
- Syntax : !SCAN <node 1> <node 2> <node 3> . . .
- !SCAN can be used anywhere inside the code as long as its arguments are defined before.
- !SCAN works with OT_ANALYSIS and RT_ANALYSIS to scan the data from cmd terminal.

```
D:\CBCS2_127\User_Manual_Zebra>zebra test_circuit.txt

File creation timestamp - 2021-03-01 16:17:05.265000
ZEBRA v3.3 (v3.3.0, February 28, 2021)
Refer to the user manual for more information


**********************************************************************

-analysis type : OT_ANALYSIS
-number of nodes to be analysed : 4
-nodes to be scanned : 0,1
-nodes to be printed : 0,1,2,3
-nodes connected out : n.n.
-nodes connected in : n.n.
-data file name to be written to : n.f.

**********************************************************************

-pre-simulation setup : no
-pre-simulation setup file : n.f.
-pre-simulation setup nodes : n.n.
-pre-simulation setup cycles : 0

**********************************************************************

-the simulation calculates the bias point voltages at the nodes
-the node voltages will be printed under the node names

**********************************************************************

-press enter to start simulation...

-STARTING SIMULATION-


-Scanning Node Voltages-

0 : 0
1 : 1

-Printing Node Voltages-

0       1       2       3
0       1       1       0

-SIMULATION ENDED-
-total simulation time : 154.320224741 ms
```

Figure 6

!PLOT – plots the node voltage data generated using RT_ANALYSIS, TIME_ANALYSIS or TT_ANALYSIS.
- Takes the node names as the arguments whose voltages are to be plotted.
- Syntax : !PLOT <node 1> <node 2> <node 3> . . .
- !PLOT can be used anywhere inside the code as long as its arguments are defined before.
- When simulation is carried out for a finite amount of time, !PLOT plots the node voltages over a finite time interval, however when simulation is carried out in real time, !PLOT animates the current voltage states of the nodes in real time.

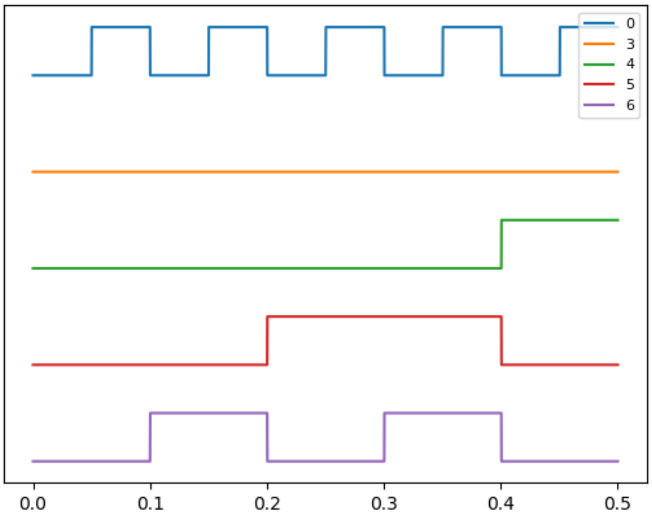

Figure 7

!FIX_VOLATGE – defines the voltage data at a particular node before at pre-simulation.

- Takes the first argument as the node name whose voltage is to be set and the next argument as the voltage value to be set.
- Syntax : !FIX_VOLTAGE <node name> <voltage value>
- !FIX_VOLTAGE can be used only inside a circuit description (maincircuit as well as subcircuit description) provided its argument is defined before.

!CLOCK – defines clock input to a particular node.

- Takes the node name as first argument, the successive arguments being offset time(in ms), on time(in ms), off time(in ms) and the initial clock state(0/1).

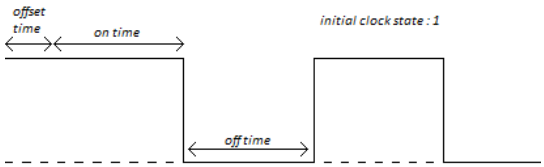

Figure 8

- Syntax : !CLOCK <node name> <offset time> <on time> <off time> <initial clock state>
- The clock input is synchronised to CPU time.
- Works only with TIME_ANALYSIS and RT_ANALYSIS, and is disabled at pre-simulation.
- !CLOCK can be used only inside maincircuit description provided that its node argument is defined before.
- Only one clock input can be defined in code. Defining more than one clock input will result in error.

```
!MAINCKT

SUBCKT   UP_COUNTER_PR_4bit   0  1  2  3  4  5  6
!FIX_VOLTAGE   1  1
!CLOCK 0  0  50    50    0

!END_MAINCKT TIME_ANALYSIS    500
!PRINT  0  3  4  5  6
!TIME_ANALYSIS_SETUP    2  2
```

Figure 9

!MAINCKT_SETUP – to carry out a pre-simulation by setting different voltages at different nodes prior to the main simulation. The node voltages can also be set at specified times if provided. Pre-simulation is often used to set the circuit to certain configuration, before starting the main simulation. For example, a ring counter needs to be set before it can start counting. This can be done by setting the counter at pre-simulation.

- The results of the pre-simulations are printed in the cmd terminal window or in an opened file, as arguments of !PRINT or !PRINTF respectively.
- Can be used with any analysis type.
- It takes the node numbers and time data(optional) as the arguments. If the time data is not provided, then the voltages are to be set at time intervals of 30 milliseconds. Whereas if time data is provided, the voltages will be set at the specified times only.
- Syntax : !MAINCKT_SETUP time(optional) <node 1> <node 2> <node 3> . . .
- If the time data is to be provided we have to write **time** as the first argument to MAINCKT_SETUP.
- To provide the time data(optional) and the voltage data, each of the following lines should begin with a '$' sign followed by the time value(optional) at which the list of voltages are to be set and then the voltage values corresponding to each of the nodes passed as arguments to !MAINCKT_SETUP. The time data must be provided as the first value, only.

!END_MAINCKT_SETUP – marks the end of maincircuit setup construct.

- Anything enclosed between !MAINCKT_SETUP and !END_MAINCKT_SETUP is treated as data to pre - simulations.
- E.g.(without time data)

```
!MAINCKT_SETUP    0    2    5
$    1    1    0
$    1    0    1
!END_MAINCKT_SETUP
```

E.g.(with time data)

```
!MAINCKT_SETUP    time    0    2    5
$    50    1    1    0
$    150    1    0    1
!END_MAINCKT_SETUP
```

- In case we need to read the setup voltages for the nodes from a file, the file name needs to be mentioned after the command !END_MAINCKT_SETUP. These files need to be created and saved beforehand in the same directory as the script. Zebra reads the data for the nodes first from the list defined under the !MAINCKT_SETUP construct and then from the file. Data in the file is written in the same way as under the !MAINCKT_SETUP construct, but without the '$' sign. No spaces should be left in between the data rows.
- E.g.

```
!MAINCKT_SETUP    0    2    5
$    1    1    0
$    1    0    1
!END_MAINCKT_SETUP    setup_data.txt
```

!WRITE_TO_FILE - to open a writable file in which the analysis data can be written using !PRINTF.

- It takes the file name along with its extension as its only argument and creates the file under the same directory as the script.
- Syntax : !WRITE_TO_FILE <file name with extension>
- !WRITE_TO_FILE can be used anywhere in the code.
- The file is created even when there is no !PRINTF command used with it. In that case the file does not contain the voltages at any specific node or time rather it contains the general simulation information.

Analysis Types

Analysis type determines the way in which the circuit will be simulated. Analysis type should be chosen in a way to best match the requirements of the circuit. For example, a sequential circuit needs to be simulated in real time, whereas a combinational circuit has no such need. Once a combinational circuit reaches a stable state, no further simulation is required. Thus the choice of analysis type should be made carefully. Each analysis type has its own features. The analysis type needs to be mentioned at the end of maincircuit description as an argument to the command !END_MAINCKT. Zebra provides four types of analysis, namely OT_ANALYSIS, RT_ANALYSIS, TT_ANALYSIS and TIME_ANALYSIS.

~ ***OT_ANALYSIS*** :

- refers to one time analysis
- the circuit is simulated with the predefined node values till the bias point is reached.
- the initial node voltages can be set by using !FIX_VOLTAGE, !SCAN and !MAINCKT_SETUP construct.
- Does not take any argument
- Does not !PLOT and !CLOCK
- is generally useful with combinational circuits
- syntax : !END_MAINCKT OT_ANALYSIS

~ ***RT_ANALYSIS***

- refers to run time analysis
- simulates the circuit in real time in a separate window
- the compiler keeps track of CPU time
- the node voltages can be set during simulation using !SCAN in the shell window
- initial node voltage values can be set either using !FIX_VOLTAGE or the !MAINCKT_SETUP construct
- has uses for both combinational as well as sequential circuits
- to exit simulation press enter without entering any value
- cannot use !PRINT and !PLOT together; If used, only printing takes place
- does not take any argument
- syntax : !END_MAINCKT RT_ANALYSIS

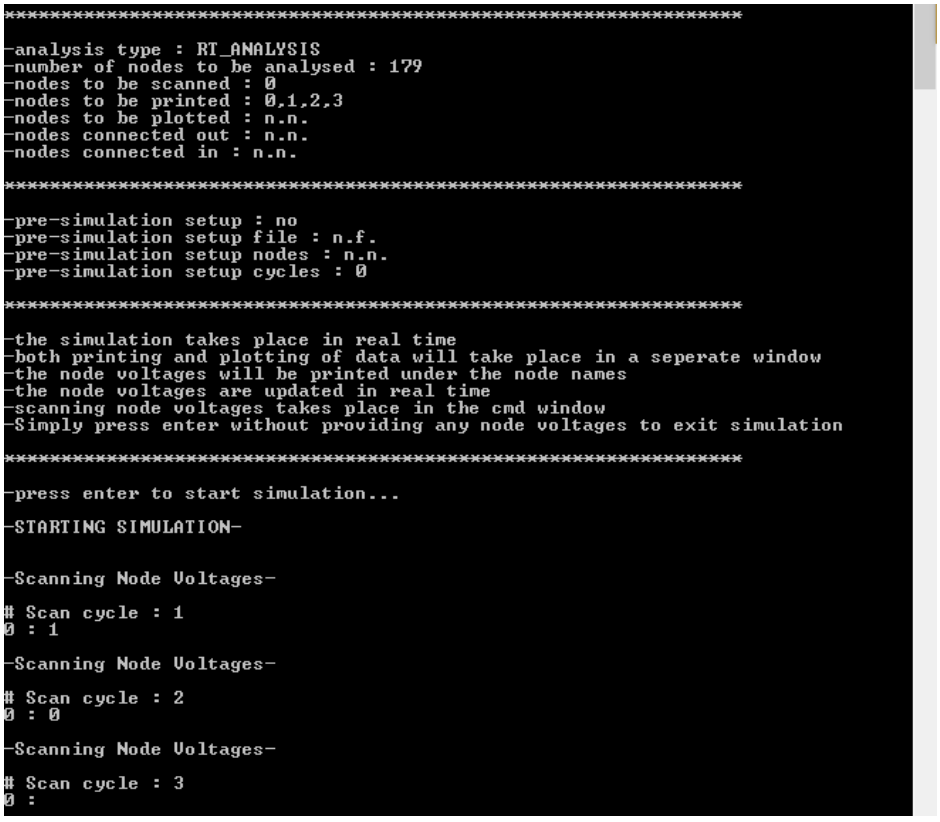

Figure 10

The above figure shows how scan cycles take place during real time simulation in the cmd terminal window. The inputs affect the circuit in real time.

```
######################## THIS IS REAL TIME PRINTING ########################

                    t(s)    0       1       2       3
                    9.93    1       0       0       1
```

Figure 11

The real time printing of node voltages take place in a separate window as shown above. The same applies for real time plotting as shown below.

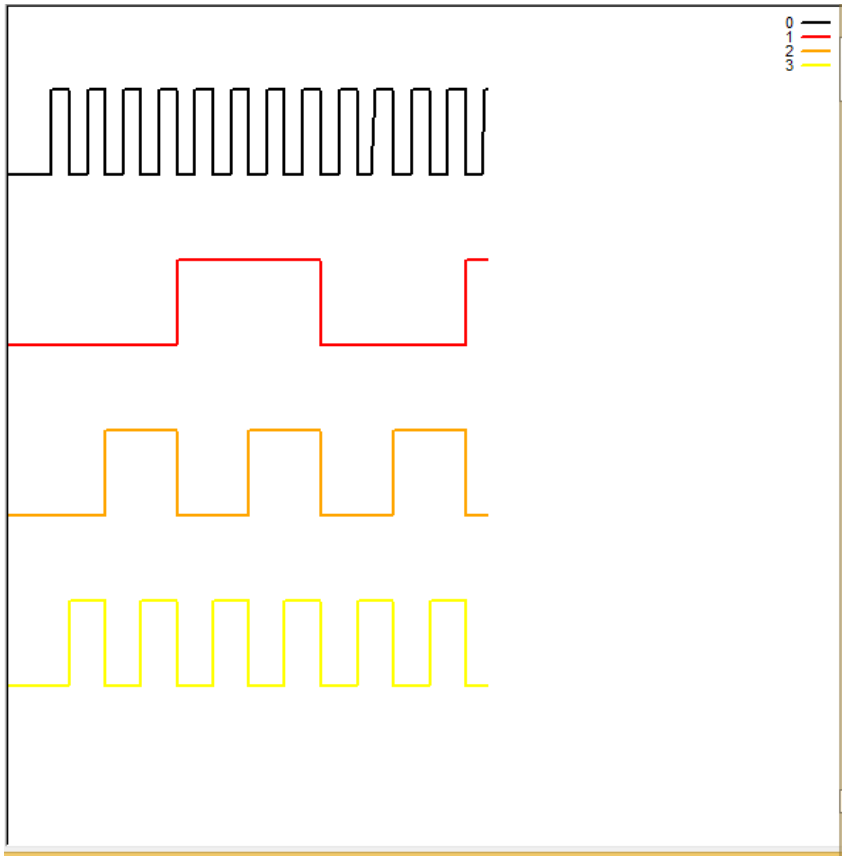

Figure 12

~ ***TT_ANALYSIS*** :

- refers to truth table analysis
- performs OT_ANALYSIS with every possible combination of the input node values if input file is not provided
- the input node names need to be provided as arguments to TT_ANALYSIS
- we can also provide an input file as an argument to TT_ANALYSIS in which case TT_ANALYSIS is carried out only on the specified combination of node voltages as provided in the file. Also we can set the specified voltage combinations to the input nodes at specified time values by providing the time data
- !PLOT can be used with TT_ANALYSIS only when the time data is provided in the input file
- does not support !SCAN and !CLOCK
- generally useful for combinational circuits
- syntax :
  !END_MAINCKT TT_ANALYSIS <input node 1>, <input node 2>, . . . <file_name>.txt : time (optional)
- the input file as well as the **:time** to the argument is optional
- if the input file is provided without **:time** then only the voltage values for the input nodes are to be provided in the same way as for !MAINCKT_SETUP input file

- if **:time** is used then the time values are to written as the first value in each row of the data value just like for !MAINCKT_SETUP input files

```
1   !MAINCKT
2   SUBCKT   XOR_3 0 1 2 3
3   !END_MAINCKT TT_ANALYSIS   0,1,2
4   !PRINT  0 1 2 3
5
```

Figure 13

- The figures above and below show the script and output for an example of TT_ANALYSIS.

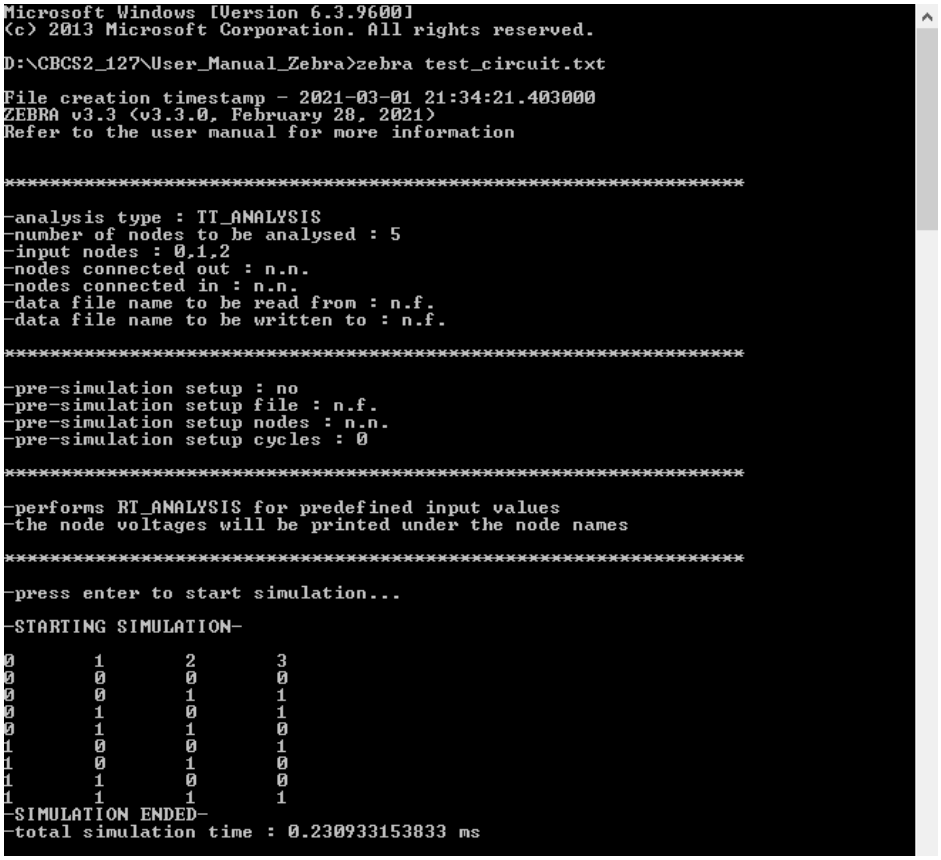

Figure 14

~ ***TIME_ANALYSIS*** :

- this analysis lets us carry out the simulation over a fixed time interval, or in real time but no changes can be made to the circuit in real time.
- !SCAN cannot be used in this analysis type
- allows two variations :
  a) real time analysis – simulation takes place in real time, results are displayed while the simulation is still running. !PRINT prints the node voltages in real time. !PLOT animates the node voltages with time. To exit simulation press Enter.
  Syntax : !END_MAINCKT TIME_ANALYSIS

  b) finite time analysis – simulation takes place for a finite amount of time. The results of simulation are displayed after the simulation ends. !PRINT and !PLOT displays at the end of simulation. The simulation time needs to be mentioned in milliseconds after analysis type.
  Syntax : !END_MAINCKT TIME_ANALYSIS 1000

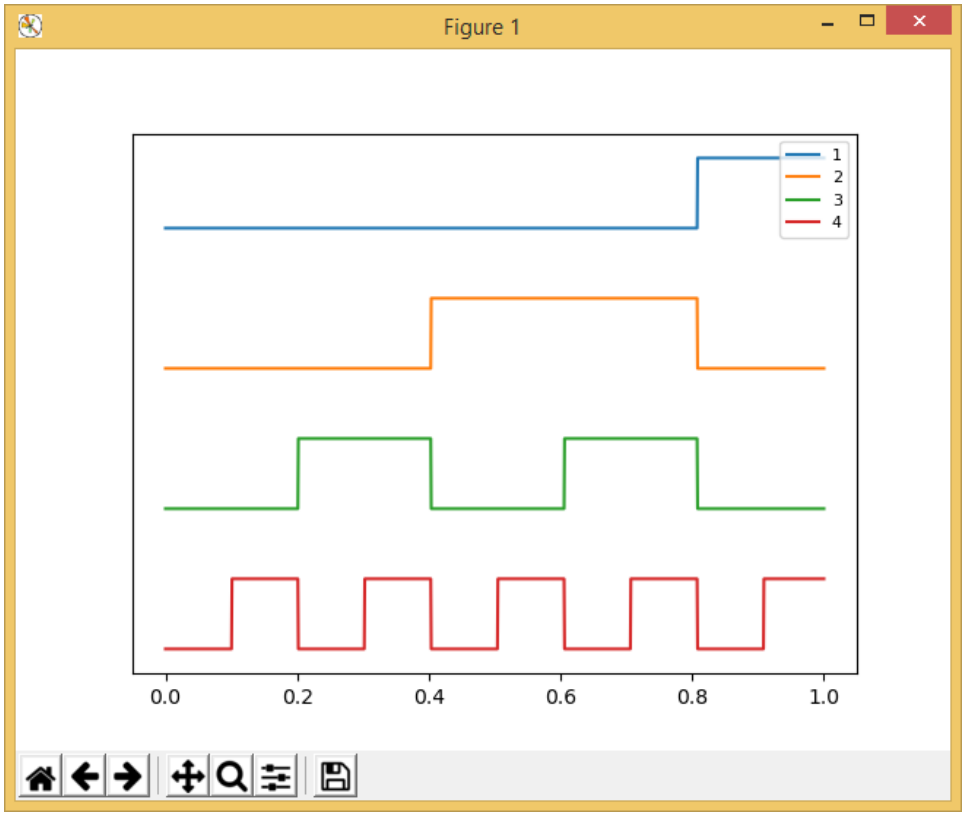

Figure 15

```
1  !MAINCKT
2  SUBCKT   UP_COUNTER_4bit    0  1  2  3  4
3  !CLOCK 0  0  50    50    0
4  !END_MAINCKT    TIME_ANALYSIS    1000
5  !PLOT   1  2  3  4
```

Figure 16

Although there is no restriction on when we should one analysis and not the other, some analysis type may prove to be more advantageous than the others depending on the specific circuit.

Subcircuits

Subcircuits are predefined circuits that can be used to make code modular. These can be used in place of full-fledged circuits. Once created, subcircuits can be expanded again and again by invoking them as and when required. All subcircuits need to be defined, before they can be used anywhere in the maincircuit.

Advantages of subcircuits:

- provides circuit abstraction
- allows reusability of circuits
- helps make circuit modular

Of two types:

- predefined - subcircuits already defined by the language developers
- user-defined - subcircuits developed by the user as per his needs

User defined subcircuit definition:

- a user defined subcircuit needs to be defined in order to use it in the maincircuit or in another user-defined subcircuit
- can be defined in any part of the code
- cannot be defined inside another circuit description

- Syntax:
  !SUBCKT <subcircuit_name> <node_no_1> <node_no_2> . . . . . .
  . . .
  . . .
  <subcircuit_body>
  . . .
  . . .
  !END_SUBCKT

- Subcircuit naming conventions:
  a) can contain digits, but not just digits
  b) can use special characters
  c) name should convey a meaning to the functionality of the subcircuit

- Node numbers conventions:
  a) must be of type integer
  b) negative numbers are not allowed
  c) floating point numbers are not allowed
  d) 0 is allowed as a node number

- Subcircuit body conventions:
  a) can use the fundamental gates
  b) can call other subcircuits
  c) allows the command !FIX_VOLTAGE

Expanding subcircuits:

- connects the corresponding subcircuit body with the allotted nodes at the location where the subcircuit is invoked
- this serves as a pre-processing instruction - the subcircuits expand at the invoked location before the circuit simulation takes place
- can be invoked multiple times inside the same circuit description

- Syntax:
  SUBCKT<subcircuit_name> <node_no_1> <node_no_2> . . . . . .

- subcircuit needs to be defined in order to be able to use it in the maincircuit description

- the node numbers with respect to subcircuit expansion are of two types:
  a) local node numbers - the node numbers provided at the time of subcircuit definition
  b) real node numbers - these are the node numbers provided at the time of subcircuit invocation

- the local node numbers need not match the real node numbers
- the total number of nodes provides at the invoking site should match the total number
- provided at the subcircuit definition
- the order of the functionality of the nodes at subcircuit calling should match the one at subcircuit definition

```
1   !SUBCKT  AND_4 0 1 2 3 4
2   AND_2 0 1 5
3   AND_2 5 2 6
4   AND_2 6 3 4
5   !END_SUBCKT
6
7   !MAINCKT
8   SUBCKT   AND_4 10   11   12   13   14
9   !FIX_VOLTAGE  10   1
10  !FIX_VOLTAGE  11   1
11  !FIX_VOLTAGE  12   1
12  !FIX_VOLTAGE  13   1
13  !END_MAINCKT   TT_ANALYSIS   10,11,12,13
14  !PRINT 10   11   12   13
```

Figure 17

Predefined subcircuits:

- Zebra provides a range of inbuilt subcircuits and ICs for use as and when required
- The following table provides the list of some of the inbuilt subcircuits provided by Zebra. Refer to the next chapter for more details.

SUBCIRCUIT NAME	SUBCIRCUIT DESCRIPTION
4bit_ADDER	4 bit parallel adder
4bit_SUBTRACTOR_2COM	4 bit subtractor using 2's complement logic
AND_3	3 input and gate
COMPARATOR_4bit	Compares two 4 bit numbers for equality, less than and greater than
D_FLIPFLOP	Edge triggered master-slave D flip-flop without preset and reset inputs
D_FLIPFLOP_PR	Edge triggered master-slave D flip-flop with preset and reset inputs
DECADE_COUNTER_PR	Asynchronous decade up counter with enable and reset inputs
DECODER_1_2	1:2 decoder circuit
DECODER_2_4	2:4 decoder circuit
DECODER_3_8	3:8 decoder circuit
DOWN_COUNTER_2bit	2 bit asynchronous down counter without enable and reset inputs
DOWN_COUNTER_3bit	3 bit asynchronous down counter without enable and reset inputs
DOWN_COUNTER_4bit	4 bit asynchronous down counter without enable and reset inputs
DOWN_COUNTER_PR_2bit	2 bit asynchronous down counter with enable and reset inputs
DOWN_COUNTER_PR_3bit	3 bit asynchronous down counter with enable and reset inputs
DOWN_COUNTER_PR_4bit	4 bit asynchronous down counter with enable and reset inputs
FULL_ADDER	Full adder circuit
FULL_SUBTRACTOR	Full subtractor circuit
HALF_ADDER	Half adder circuit
JOHNSON_COUNTER	4 bit Johnson counter circuit
MS_JK_FLIPFLOP	Master-slave JK flip-flop circuit without preset and reset inputs
MS_JK_FLIPFLOP_PR	Master-slave JK flip-flop circuit with preset and reset inputs

MUX_2	2x1 multiplexer circuit
MUX_4	4x1 multiplexer circuit
MUX_8	8x1 multiplexer circuit
NAND_3	3 input nand gate
NAND_4	4 input nand gate
NOR_3	3 input nor gate
OR_3	3 input or gate
REGISTER_PIPO_4bit	4 bit parallel in parallel out shift register with clear input
REGISTER_PISO_4bit	4 bit parallel in shift out shift register with clear input
REGISTER_SIPO_4bit	4 bit serial in parallel out shift register with clear input
REGISTER_SISO_4bit	4 bit serial in serial out shift register with clear input
REGISTER_UNIVERSAL_4bit	4 bit universal shift register with clear input
RING_COUNTER	4 bit Ring counter circuit
SR_FLIPFLOP	Level triggered SR flip-flop circuit
SR_LATCH	SR latch circuit with NAND gates
T_FLIPFLOP	Master-slave T flip-flop without preset and reset inputs
T_FLIPFLOP_PR	Master-slave T flip-flop with preset and reset inputs
UP_COUNTER_2bit	2 bit asynchronous down counter without enable and reset inputs
UP_COUNTER_3bit	3 bit asynchronous down counter without enable and reset inputs
UP_COUNTER_4bit	4 bit asynchronous down counter without enable and reset inputs
UP_COUNTER_PR_2bit	2 bit asynchronous down counter with enable and reset inputs
UP_COUNTER_PR_3bit	3 bit asynchronous down counter with enable and reset inputs
UP_COUNTER_PR_4bit	4 bit asynchronous down counter with enable and reset inputs
XNOR_3	3 input xnor gate
XOR_3	3 input xor gate

- The user may define subcircuits with names same as that of the inbuilt subcircuits. In that case the user defined subcircuit gets a higher priority, i.e. the user defined subcircuit gets expanded whenever invoked with that name.
- In case the user wants to include a user defined subcircuit as an inbuilt subcircuit, all he needs to do is after the simulation ends, open the folder SUBCIRCUITS_USER_DEFINED inside the installation directory for Zebra, copy the temporary subcircuit file created during simulation, copy it and paste it inside the SUBCIRCUITS_INBUILT folder.
- Apart from these predefined subcircuits, Zebra also provides us with a range of 74XX IC subcircuits. These ICs exactly behave like physical ICs. Each of them needs to be hooked up to power and ground before they can be used.
- The Vcc and Gnd pins to these ICs need to be fixed to voltages 1 and 0 respectively.
- IC subcircuits can be invoked in the same way as normal subcircuits. Each IC subcircuit name is preceded by 'IC' followed by the IC number. E.g. IC7400, IC74138, IC7489, etc.
- While invoking the IC subcircuits, the order of the nodes succeeding the IC name have the same order as the actual IC pins.
- For example, consider the pin diagram of IC 7400 (quad 2 input NAND gate):

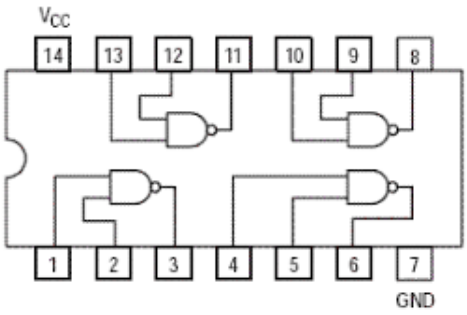

Figure 18

Pin 1 and Pin 2 are the inputs and Pin 3 is the output to a NAND gate. Now to use this IC, we can write :

```
SUBCKT      IC7400      10    13    12    14    5     6     7     8     19
10    21    22    23    25
```

Thus the node 10 in the circuit description gets connected to Pin 1 (first input) in the IC, node 13 gets connected to Pin 2 (second input) in the IC and node 12 gets connected to Pin 3 (output). Thus the index (starting from 1 for node 10 to 14 for node 25) of the nodes passed as arguments to the IC name should match the real pin numbers of the IC.

- Following is the list of some of the ICs Zebra provides as inbuilt subcircuits :

IC NAME	IC DESCRIPTION
7400	Quad 2 input NAND gate IC
7402	Quad 2 input NOR gate IC
7404	Hex inverter IC
7408	Quad 2 input AND gate IC
7410	3 input NAND gate IC
7411	3 input AND gate IC
7420	Dual 4 input NAND gate IC
7421	Dual 4 input AND gate IC
7432	Quad 2 input OR gate IC
7473	Dual JK flip-flop with clear
7474	Dual D flip-flop with positive edge trigger
7476	Dual JK flip-flop with preset and clear
7483	4bit parallel full adder
7485	4 bit magnitude comparator
7486	Quad 2 input XOR gate
7489	64 bit RAM
7490	Decade counter
7493	4 bit asynchronous counter
74138	3 to 8 line decoder
74139	Dual 2 to 4 line decoder
74147	10 to 4 line priority encoder
74148	8 to 3 line priority encoder
74151	8 to 1 multiplexer
74157	Quad 2 to 1 multiplexer
74164	8 bit SIPO shift register
74166	8bit PISO shift register
74194	4 bit universal shift register
74266	Quad two input XNOR gate
74279	Quad SR latch
74299	8 bit universal shift register
74595	8 bit SIPO shift register with 3 state output

- Refer to the datasheet of the ICs for more information.
- It is obviously wiser to use individual gates instead of ICs, to avoid extra hassle of the large number of nodes that need to be mentioned for each IC.

Predefined Subcircuits

Let us go through a detailed description of the inbuilt subcircuits provided by Zebra.

Combinational Subcircuits

 # AND_3
~3 input AND gate
~function description:
AND_3 <input_1> <input_2> <input_3> <output>

 # OR_3
~3 input OR gate
~function description:
OR_3 <input_1> <input_2> <input_3> <output>

 # NAND_3
~3 input NAND gate
~function description:
NAND_3 <input_1> <input_2> <input_3> <output>

 # NAND_4
~4 input NAND gate
~function description:
NAND_4 <input_1> <input_2> <input_3> <input_4> <output>

 # NOR_3
~3 input NOR gate
~function description:
NOR_3 <input_1> <input_2> <input_3> <output>

 # XOR_3
~3 input XOR gate
~function description:
XOR_3 <input_1> <input_2> <input_3> <output>

 # XNOR_3
~3 input XNOR gate
~function description:
XNOR_3 <input_1> <input_2> <input_3> <output>

 # HALF_ADDER
~adds two bits A and B to produce sum and carry
~function description:
HALF_ADDER <A> <B> <sum> <carry>

 # FULL_ADDER
~adds three bits A, B and carry in to produce sum and carry
~function description:
FULL_ADDER <A> <B> <C_{in}> <sum> <carry>

 # FULL_SUBTRACTOR
~subtracts bit B from A using borrow in to produce difference and borrow out
~function description:
FULL_SUBTRACTOR <A> <B> <B_{in}> <difference> <B_{out}>

 # 4bit_ADDER
~4 bit parallel adder that adds two four bit numbers $A_3A_2A_1A_0$ and $B_3B_2B_1B_0$ with a carry in to produce 4 bit sum $S_3S_2S_1S_0$ and a carry out
~function description:
4bit_ADDER <C_{in}> <A_3> <A_2> <A_1> <A_0> <B_3> <B_2> <B_1> <B_0> <C_{out}>

 # 4bit_SUBTRACTOR_2COM
~4 bit parallel subtractor that subtracts one four bit number $B_3B_2B_1B_0$ from another 4 bit number $A_3A_2A_1A_0$ to produce a 4 bit result $D_3D_2D_1D_0$ and a sign of the result
~function description:
4bit_SUBTRACTOR <A_3> <A_2> <A_1> <A_0> <B_3> <B_2> <B_1> <B_0> <sign>

 # MUX_2
~2 x 1 multiplexer that selects one out of two input lines I_0 and I_1 using select line S_0 into a single output line O
~function description:
MUX_2 <S_0> <I_0> <I_1> <O>

 # MUX_4
~4 x 1 multiplexer that selects one out of four input lines I_0, I_1, I_2 and I_3 using select lines S_1 and S_0 into a single output line O
~function description:
MUX_4 <S_1> <S_0> < I_0> <I_1> <I_2> <I_3> <O>

 # MUX_8
~8 x 1 multiplexer that selects one out of four input lines I_0, I_1, I_2, I_3, I_4, I_5, I_6 and I_7 using select lines S_2, S_1 and S_0 into a single output line O
~function description:
MUX_8 <S_2> <S_1> <S_0> <I_0> <I_1> <I_2> <I_3> <I_4> <I_5> <I_6> <I_7> <O>

 # COMPARATOR_4bit
~4 bit comparator that compares two 4 bit numbers $A_3A_2A_1A_0$ and $B_3B_2B_1B_0$ by subtracting second number from first using 2,s complement method and gives output: A>B , A<B and A=B
~function description:
COMPARATOR_4bit <A_3> <A_2> <A_1> <A_0> <B_3> <B_2> <B_1> <B_0> <A>B>
<A<B> <A=B>

 # DECODER_1_2
~A 1x2 decoder that has one active high input B_0 and decodes it into two outputs D_0 and D_1
~function description:
DECODER_1_2 <B_0> <D_0> <D_1>

 # DECODER_2_4
~A 2x4 decoder that has two active high input B_1 B_0 and decodes it into four outputs D_0 , D_1 , D_2 and D_3
~function description:
DECODER_2_4 <B_1> <B_0> <D_0> <D_1> <D_2> <D_3>

 # DECODER_3_8
~A 3x8 decoder that has three active high inputs B_2 B1 B_0 and decodes it into four outputs D_0 , D_1 , D_2 , D_3 , D_4 , D_5 , D_6 and D_7
~function description:
DECODER_3_8 <B_2> <B_1> <B_0> <D_0> <D_1> <D_2> <D_3> <D_4> <D_5> <D_6> <D_7>

Sequential Subcircuits

Flip-flops

 # SR_LATCH
~SR latch made from NAND gates with Preset (PR') and Clear (CLR'). PR' and CLR' are active Low.
~function description:
SR_LATCH <S'> <R'> <Q> <Q'> <PR'> <CLR'>

 # SR_FLIPFLOP
~SR flip flop made from NAND gates with Preset (PR') and Clear (CLR') inputs. PR' and CLR' are active Low. The flip flop is High Level Triggered.
~function description:
SR_FLIPFLOP <Clock> <S> <R> <Q> <Q'> <PR'> <CLR'>

 # D_FLIPFLOP
~Master-slave D flip flop made from NAND gates. The flip flop is Negative Edge Triggered.
~function description:
D_FLIPFLOP <Clock> <D> <Q> <Q'>

 # D_FLIPFLOP_PR
~Master-slave D flip flop made from NAND gates with Preset (PR') and Clear (CLR') inputs. PR' and CLR' are active Low. The flip flop is Negative Edge Triggered.
~function description:
D_FLIPFLOP_PR <Clock> <D> <Q> <Q'> <PR'> <CLR'>

 # JK_FLIPFLOP
~JK flip flop made from NAND gates with Preset (PR') and Clear (CLR') inputs. PR' and CLR' are active Low. The flip flop is High Level Triggered.
~function description:
JK_FLIPFLOP <Clock> <J> <K> <Q> <Q'> <PR'> <CLR'>

 # MS_JK_FLIPFLOP
~Master-slave JK flip flop with Negative Edge Triggered clock input.
~function description:
JK_FLIPFLOP <Clock> <J> <K> <Q> <Q'>

 # MS_JK_FLIPFLOP_PR
~Master-slave JK flip flop with Preset (PR') and Clear (CLR') inputs. PR' and CLR' are active Low. The flip flop is Negative Edge Triggered.
~function description:
MS_JK_FLIPFLOP_PR <Clock> <J> <K> <Q> <Q'> <PR'> <CLR'>

 # T_FLIPFLOP
~Master-slave T flip flop with Negative Edge Triggered clock input.
~function description:
T_FLIPFLOP <Clock> <T> <Q> <Q'>

 # T_FLIPFLOP_PR
~Master-slave T flip flop with Preset (PR') and Clear (CLR') inputs. PR' and CLR' are active Low. The flip flop is Negative Edge Triggered.
~function description:
T_FLIPFLOP_PR <Clock> <T> <Q> <Q'> <PR'> <CLR'>

Counters

 # UP_COUNTER_2bit
~ It is a 2 bit up-counter with outputs Q_1 and Q_0.
~function description:
UP_COUNTER_2bit <Clock> <Q_1> <Q_0>

 # UP_COUNTER_3bit
~ It is a 3 bit up-counter with outputs Q_2, Q_1 and Q_0.
~function description:
UP_COUNTER_3bit <Clock> <Q_2> <Q_1> <Q_0>

 # UP_COUNTER_4bit
~ It is a 4 bit up-counter with outputs Q_3, Q2, Q_1 and Q_0.
~function description:
UP_COUNTER_4bit <Clock> <Q_3> <Q_2> <Q_1> <Q_0>

 # UP_COUNTER_PR_2bit
~ It is a 2 bit up-counter with outputs Q_1 and Q_0. It has an active high Enable input E that enables the counter and a clear input CLR'.
~function description:
UP_COUNTER_PR_2bit <Clock> <E> <Q_1> <Q_0> <CLR'>

 # UP_COUNTER_PR_3bit

~ It is a 3 bit up-counter with outputs Q_2, Q_1 and Q_0. It has an active high Enable input E that enables the counter and a clear input CLR'.

~function description:

UP_COUNTER_3bit <Clock> <E> <Q_2> <Q_1> <Q_0> <CLR'>

 # UP_COUNTER_PR_4bit

~ It is a 4 bit up-counter with outputs Q_3, Q2, Q_1 and Q_0. It has an active high Enable input E that enables the counter and a clear input CLR'.

~function description:

UP_COUNTER_4bit <Clock> <E> <Q_3> <Q_2> <Q_1> <Q_0> <CLR'>

 # DOWN_COUNTER_2bit

~ It is a 2 bit down -counter with outputs Q_1 and Q_0.

~function description:

DOWN _COUNTER_2bit <Clock> <Q_1> <Q_0>

 # DOWN_COUNTER_3bit

~ It is a 3 bit down -counter with outputs Q_2, Q_1 and Q_0.

~function description:

DOWN _COUNTER_3bit <Clock> <Q_2> <Q_1> <Q_0>

 # DOWN_COUNTER_4bit

~ It is a 4 bit down -counter with outputs Q_3, Q2, Q_1 and Q_0.

~function description:

DOWN _COUNTER_4bit <Clock> <Q_3> <Q_2> <Q_1> <Q_0>

 # DOWN_COUNTER_PR_2bit

~ It is a 2 bit down -counter with outputs Q_1 and Q_0. It has an active high Enable input E that enables the counter and a clear input CLR'.

~function description:

DOWN _COUNTER_PR_2bit <Clock> <E> <Q_1> <Q_0> <CLR'>

 # DOWN_COUNTER_PR_3bit

~ It is a 3 bit down -counter with outputs Q_2, Q_1 and Q_0. It has an active high Enable input E that enables the counter and a clear input CLR'.

~function description:

DOWN _COUNTER_PR_3bit <Clock> <E> <Q_2> <Q_1> <Q_0> <CLR'>

 # DOWN_COUNTER_PR_4bit

~ It is a 4 bit down -counter with outputs Q_3, Q2, Q_1 and Q_0. It has an active high Enable input E that enables the counter and a clear input CLR'.

~function description:

DOWN _COUNTER_PR_4bit <Clock> <E> <Q_3> <Q_2> <Q_1> <Q_0> <CLR'>

 # JOHNSON_COUNTER

~ It is a 4 bit Johnson counter with outputs Q_3, Q2, Q_1 and Q_0 a clear input CLR' which is active low.

~function description:

JOHNSON _COUNTER <Clock> <Q_3> <Q_2> <Q_1> <Q_0> <CLR'>

 # RING_COUNTER

~ It is a 4 bit Ring counter with outputs Q_3, Q2, Q_1 and Q_0 a clear input CLR' which sets the flip flop Q_3 and clears the rest.

~function description:

RING _COUNTER <Clock> <Q_3> <Q_2> <Q_1> <Q_0> <PRESET_CLR>

 # DECADE_COUNTER_PR

~ It is a 4 bit decade counter that counts from 0000 to 1001 with outputs Q_3, Q2, Q_1 and Q_0. It has Preset (PR') and Clear (CLR') inputs which are active Low.

~function description:

DECADE _COUNTER <Clock> <Q_3> <Q_2> <Q_1> <Q_0> <PR'> <CLR'>

Registers

 # REGISTER_PIPO_4bit

~ 4 bit parallel in parallel out shift register with active low clear input CLR'. The parallel inputs are D_3, D_2, D_1 and D_0 and the parallel outputs are O_3, O_2, O_1 and O_0.

~ function description:

REGISTER_PIPO_4bit <Clock> <CLR'> <D_3> <D_2> <D_1> <D_0> <O_3> <O_2> <O_1> <O_0>

 # REGISTER_PISO_4bit

~ 4 bit parallel in serial out shift register with active low clear input CLR' and a LOAD/SHIFT input. The register loads parallel bits when the SHIFT/LOAD input is 1 and shifts right when the SHIFT/LOAD input is 0. The parallel inputs are D_3, D_2, D_1 and D_0 and the serial output is O.

~ function description:

REGISTER_PISO_4bit <Clock> <CLR'> <SHIFT/LOAD> <D_3> <D_2> <D_1> <D_0> <O>

 # REGISTER_SIPO_4bit

~ 4 bit serial in parallel out shift register with active low clear input CLR'. The serial input is S_{in} and the parallel outputs are O_3, O_2, O_1 and O_0.

~ function description:

REGISTER_SIPO_4bit <Clock> <CLR'> <S_{in}> <O_3> <O_2> <O_1> <O_0>

 # REGISTER_SISO_4bit

~ 4 bit serial in serial out shift register with active low clear input CLR'. The serial input is S_{in} and the serial output is S_{out}.

~ function description:

REGISTER_SISO_4bit <Clock> <S_{in}> <S_{out}> <CLR'>

 # REGISTER_UNIVERSAL_4bit

~ 4 bit universal shift register with active low clear input CLR' and select lines S_0 and S_1 for mode select. SR is the serial data input for right shift and SL is the serial data input for left shift. The following table gives a description of the register operation depending on the values of select lines.

S_1	S_0	Register mode
0	0	Data hold
0	1	Shift right
1	0	Shift left
1	1	Parallel load

The parallel inputs are D_3, D_2, D_1 and D_0 and the parallel outputs are O_3, O_2, O_1 and O_0.

~ function description:

REGISTER_UNIVERSAL_4bit <Clock> <CLR'> <S_1> <S_0> <SR> <SL> <D_3> <D_2> <D_1> <D_0> <O_3> <O_2> <O_1> <O_0>

 # RAM_3_4

~ 32 bit RAM with 3 address lines A_2, A_1, A_0 and 4 data lines with a read/write enable input WRE and an active low clear input CLR'. The parallel data inputs are D_3, D_2, D_1, D_0 and the parallel data outputs are O_3, O_2, O_1, O_0. We can write data when WRE goes from high to low. All the output lines are low when the ram is in write mode. The ram produces the data at a particular address, when WRE is high. Following is the function table for the ram subcircuit.

Clear'	WRE	A_2	A_1	A_0	O_3	O_2	O_1	O_0
0	X	x	x	x	0	0	0	0

1	1	a_2^*	a_1^*	a_0^*	o_3^*	o_2^*	o_1^*	o_0^*
1	0	x	x	x	0	0	0	0
1	↓	a_2	a_1	a_0	D_3	D_2	D_1	D_0

*$o_3o_2o_1o_0$ is the data stored at address $a_2a_1a_0$.

~ function description:

RAM_3_4 <WRE> <CLR'> <A_2> <A_1> <A_0> <D_3> <D_2> <D_1> <D_0>
<O_3> <O_2> <O_1> <O_0>

Writing Combinational Circuits

Now, that we have discussed the necessary tools to use the various features of Zebra, let us now try to implement some real circuits. This chapter deals with the implementation of combinational logic using Zebra. Refer to the next chapter for the sequential part.

- F(A,B,C) = ABC' + AB'C + A'BC

  Suppose, we wish to implement this circuit using basic gates only.
  Then, we will have the following circuit diagram:

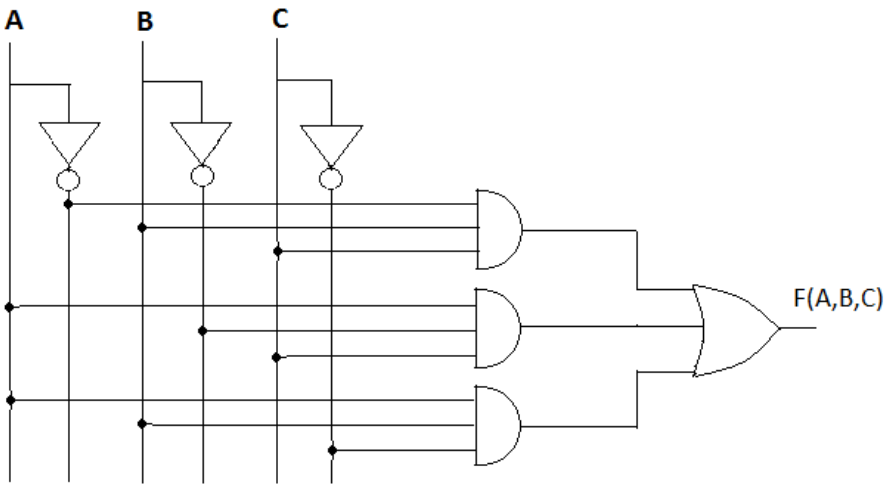

Figure 19

We write the following script to implement the above circuit:

```
!MAINCKT

*node description
*0 – A
*1 – B
*2 – C
*9 – F(A,B,C)

NOT     0       3
NOT     1       4
NOT     2       5

SUBCKT          AND_3           0       1       5       6
SUBCKT          AND_3           0       4       2       7
SUBCKT          AND_3           3       1       2       8

SUBCKT          OR_3            6       7       8       9

*fixing initial voltages to the input lines, A – 0, B – 1 and C – 1
!FIX_VOLTAGE 0          0
!FIX_VOLTAGE 1          1
!FIX_VOLTAGE 2          1
```

```
*carrying out OT_ANALYSIS on the above circuit
!END_MAINCKT          OT_ANALYSIS

*printing the values of the input and output nodes as a result of OT_ANALYSIS
!PRINT  0       1       2       9
```

- $F(S_1,S_0, A,B,C,D) = S_1'S_0'A + S_1'S_0B + S_1S_0'C + S_1S_0D$

  This is the boolean expression for a 4x1 MUX, we can implement the logic as shown in the circuit diagram below:

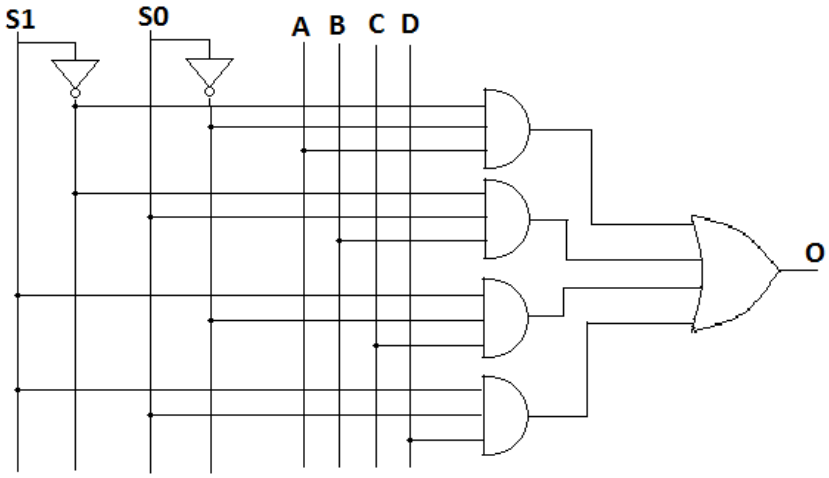

Figure 20

We write the following script to implement the above circuit:

```
*user defined subcircuit for a 4 input OR gate
!SUBCKT         OR_4    0       1       2       3       4
SUBCKT          OR_3    0       1       2       5
OR_2            5       3       4
!END_SUBCKT

!MAINCKT

*node description
*0 – S1
*1 – S0
*2 – A
*3 – B
*4 – C
*5 – D
*6 – MUX output

NOT     0       7
NOT     1       8

SUBCKT          AND_3           7       8       2       9
SUBCKT          AND_3           7       1       3       10
SUBCKT          AND_3           0       8       4       11
SUBCKT          AND_3           0       1       5       12
```

```
*invoking user defined subcircuit OR_4
SUBCKT          OR_4   9        10       11       12       6

*performing truth table analysis on the circuit
*providing nodes 0,1,2,3,4,5 as inputs
*printing the voltages at nodes 0,1,2,3,4,5,6 to a file
*the circuit will be simulated for every combination of the voltages at the input nodes
*saving the analysis data into a .txt file named MUX_4_tt_data.txt

!END_MAINCKT          TT_ANALYSIS    0,1,2,3,4,5
!PRINTF          0        1        2        3        4        5        6
!WRITE_TO_FILE          MUX_4_tt_data.txt
```

- 4 bit parallel adder circuit

This circuit requires 4 full adder circuits to be cascaded in parallel, as shown in the diagram below. This circuit inputs two 4 bit numbers $A_3A_2A_1A_0$ and $B_3B_2B_1B_0$ and a C_{in}; and then produces a five bit sum $C_{out}S_3S_2S_1S_0$.

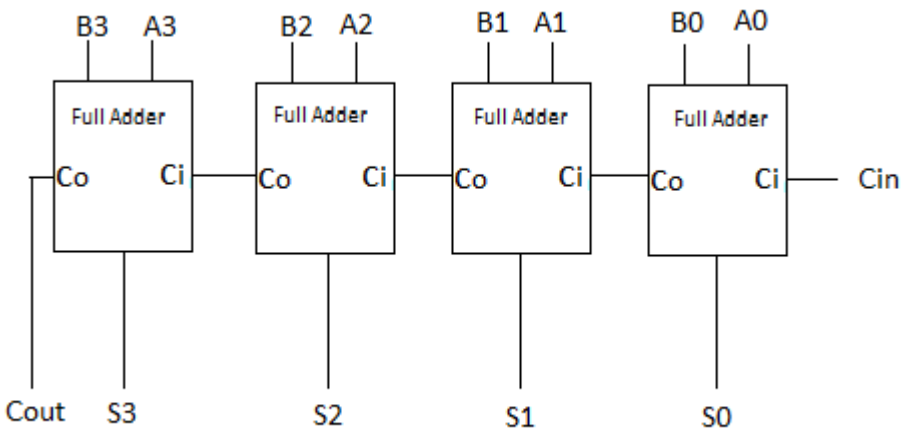

Figure 21

The above circuit can be implemented using the following script:

```
!MAINCKT

*node description
*0 – Cin
*1 – A3
*2 – A2
*3 – A1
*4 – A0
*5 – B3
*6 – B2
*7 – B1
*8 – B0
*9 – S3
*10 – S2
```

*11 – S1
*12 – S0
*13 – Cout

*invoking the FULL_ADDER subcircuits for constructing the 4 bit parallel adder
SUBCKTFULL_ADDER 4 8 0 12 14
SUBCKTFULL_ADDER 3 7 14 11 15
SUBCKTFULL_ADDER 2 6 15 10 16
SUBCKTFULL_ADDER 1 5 16 9 13

*The Cin node is set low since we do not want a second adder in parallel with this adder
!FIX_VOLTAGE 0 0

*performing run-time analysis to accept user inputs during simulation
!END_MAINCKT RT_ANALYSIS

*setting the nodes defining the two 4 bit numbers for input
!SCAN 1 2 3 4 5 6 7 8

*printing the result of addition from MSB to LSB
!PRINT 13 9 10 11 12

- 4 bit comparator using 2's complement subtractor concept:

The comparator circuit inputs two 4 bit numbers and then, complements the second of the two numbers using NOT gate and then feeds both the numbers as inputs to a 4 bit parallel adder, with the Cin input set to 1(for 2's complement). The circuit then analyses the summation. If all the bits of summation are 0, then the output bit for A=B is set high, while the others are set low. Next the sign bit is analysed, if the sign bit is 1 than output bit A>B is set high, and if 0 the output bit A<B is set high while the rest are set to 0. The circuit can be implemented using the shown circuit diagram.

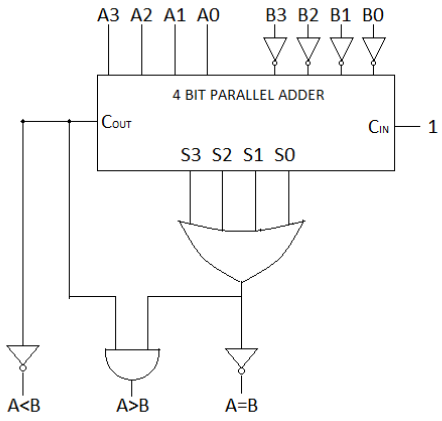

Figure 22

This is the script for simulating the above circuit:

!MAINCKT

*node description
*1 – A3
*2 – A2

```
*3 – A1
*4 – A0
*5 – B3
*6 – B2
*7 – B1
*8 – B0
*9 – A<B output
*10 – A>B output
*11 – A=B output

*complementing the A bits
NOT     5       12
NOT     6       13
NOT     7       14
NOT     8       15

SUBCKT 4bit_ADDER    0       1       2       3       4       12      13      14      15      16      17
18      19      20

*setting the Cin bit to 1 for 2's complement
!FIX_VOLTAGE  0       1

*checking the summation for equality of the two 4 bit numbers
OR_2    16      17      21
OR_2    18      19      22
OR_2    21      22      23

NOT     23      11
AND_2 20        23      10
NOT     20      9

*running one-time analysis for the above circuit
!END_MAINCKT            OT_ANALYSIS

*printing the voltages at the output nodes as a result of the simulation
!PRINT  9       10      11

*setting up the voltage values for the input nodes at the beginning of first simulation
!MAINCKT_SETUP       1       2       3       4       5       6       7       8
$       1       1       0       0       0       1       0       0
!END_MAINCKT_SETUP
```

Writing Sequential Circuits

We have already gone through a few examples regarding how to implement combinational circuits using Zebra. Now let us try to implement sequential circuits using Zebra. But before we do that there are a few things that need to be kept in mind while designing sequential circuits.

- The latch subcircuit has been specially designed to overcome the problem of sequential blockade encountered during simulation of sequential circuits. So whenever designing circuits that make use of a latch circuit, always invoke the subcircuit SR_LATCH.
- Zebra simulates the circuit components one by one, hence in theory it is not possible to simulate multiple gates together that happens in case of real circuits.
- !CLOCK command works only with TIME_ANALYSIS.
- The desired result may take different number of simulation cycles depending on the order of circuit components. Hence it is best to synchronise sequential components of the circuit with an edge triggered mechanism.
- Try to keep the clock time period not too low.

Following are some of the examples involving sequential circuit components :

- Analysing a level triggered SR – flip-flop :

  A level triggered SR – flip-flop introduces a simple enable feature to the SR – latch circuit. When the enable is high the latch becomes transparent to the input. And when the enable is low the latch goes into memory state.

SR flip-flop function table

Clock	S	R	Q_n	Q'_n
0	x	x	Q_{n-1}	Q'_{n-1}
1	0	0	Q_{n-1}	Q'_{n-1}
1	0	1	0	1
1	1	0	1	0
1	1	1	1	1

This circuit is implemented as shown below:

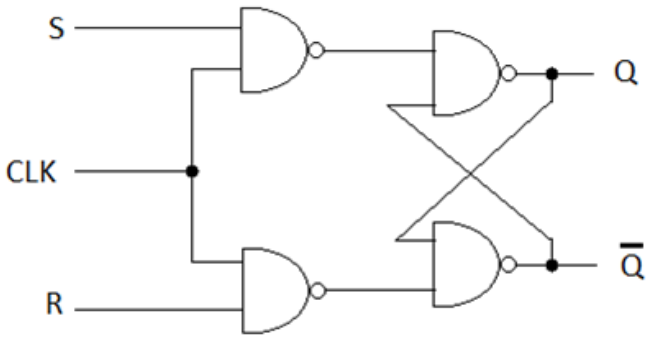

Figure 23

This is the script for simulating the above circuit:

!MAINCKT

*node description
*2 – clock or enable input

```
*0 – S
*1 – R
*5 – Q
*6 – Q'

NAND_2      0       2       3
NAND_2      1       2       4
SUBCKT      SR_LATCH    3       4       5       6       7       8

*these are the preset and clear inputs to the SR latch that are being held high
!FIX_VOLTAGE 7      1
!FIX_VOLTAGE 8      1

*using run time analysis to accept input from user at runtime and display the output voltages
!END_MAINCKT        RT_ANALYSIS

!SCAN  2        0       1
!PRINT  5       6
```

- Setting a T – flip-flop into toggle mode, and using time analysis to show the toggling action

  A T flip-flop is made by making connecting the inputs to a master slave JK flip-flop together. So we first construct a master slave JK flip flop, the set both its inputs high, to set the flip-flop into toggle mode.

T flip-flop function table

Clock	T	Q_n	Q'_n
0	x	Q_{n-1}	Q'_{n-1}
↓	0	Q_{n-1}	Q'_{n-1}
↓	1	Q'_{n-1}	Q_{n-1}

This circuit is implemented as shown below:

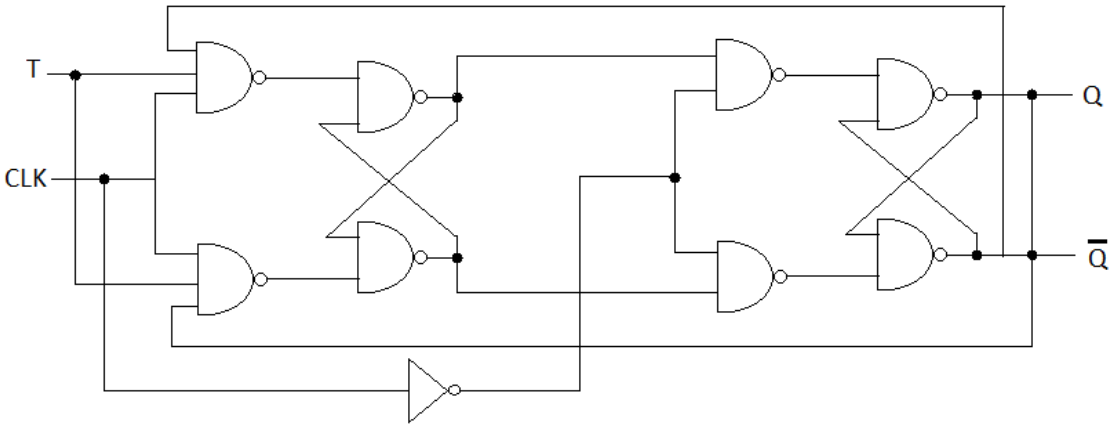

Figure 24

This is the script for simulating the above circuit:

```
!MAINCKT
```

```
*node description
*0 – clock input
*2 – T
*11 – Q
*12 – Q'

NOT     0       1

SUBCKT          NAND_3          0       2       12      3
SUBCKT          NAND_3          0       2       11      4
SUBCKT          SR_LATCH        3       4       5       6       7       8

NAND_2          1       5       9
NAND_2          1       6       10

SUBCKT          SR_LATCH        9       10      11      12      7       8

*disabling the preset and clear inputs and setting T as 1 for toggle mode
!FIX_VOLTAGE 7          1
!FIX_VOLTAGE 8          1
!FIX_VOLTAGE 2          1

*enabling clock input at node 0
!CLOCK          0       0       500     500     0

*using infinite time analysis to print the voltages at node 11 and 12
!END_MAINCKT            TIME_ANALYSIS

!PRINT  11      12
```

- Design a 3 bit asynchronous up counter with a reset input, and use time analysis to plot the counter output for 10 clock cycles

  Three T flip-flops need to be connected successively with the output of one acting as the clock for the next. The T input needs to be set high. And the preset line should be kept disabled for each flip-flop. The counter circuit is implemented as shown in the following diagram.

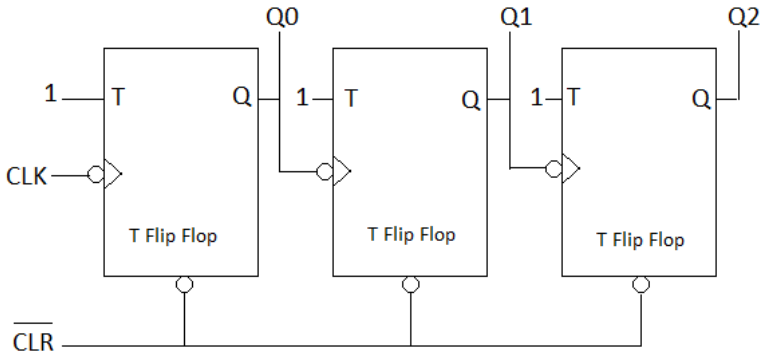

Figure 25

This is the script for simulating the above circuit:

```
!MAINCKT

*node description
*0 – clock input
*6 – T inputs
*5 – Qa
*4 – Qb
*3 – Qc
*10 – Clear input


SUBCKT          T_FLIPFLOP_PR          0       6       5       9       10      2
SUBCKT          T_FLIPFLOP_PR          5       6       4       8       10      2
SUBCKT          T_FLIPFLOP_PR          4       6       3       7       10      2

*fixing the T inputs to logic 1
!FIX_VOLTAGE  6         1

*disabling the preset inputs on the flip-flops
!FIX_VOLTAGE  10        1

*enabling clock input at node 0
!CLOCK          0       0       50      50      0

*carrying out time analysis for 1000 ms to accommodate 10 clock cycles
!END_MAINCKT            TIME_ANALYSIS           1000

*plotting the counter outputs for the simulation time
!PLOT   3       4       5
```

- Design a 4 bit parallel in serial out shift register, such that a 0 at the shift/load input shifts bits to the right, whereas a 1 at the shift/load input loads parallel bits. Use MAINCKT_SETUP construct to load the parallel bits into the register, and then use RT_ANALYSIS to shift the bits out of the register one by one. The circuit is implemented as shown by the following diagram:

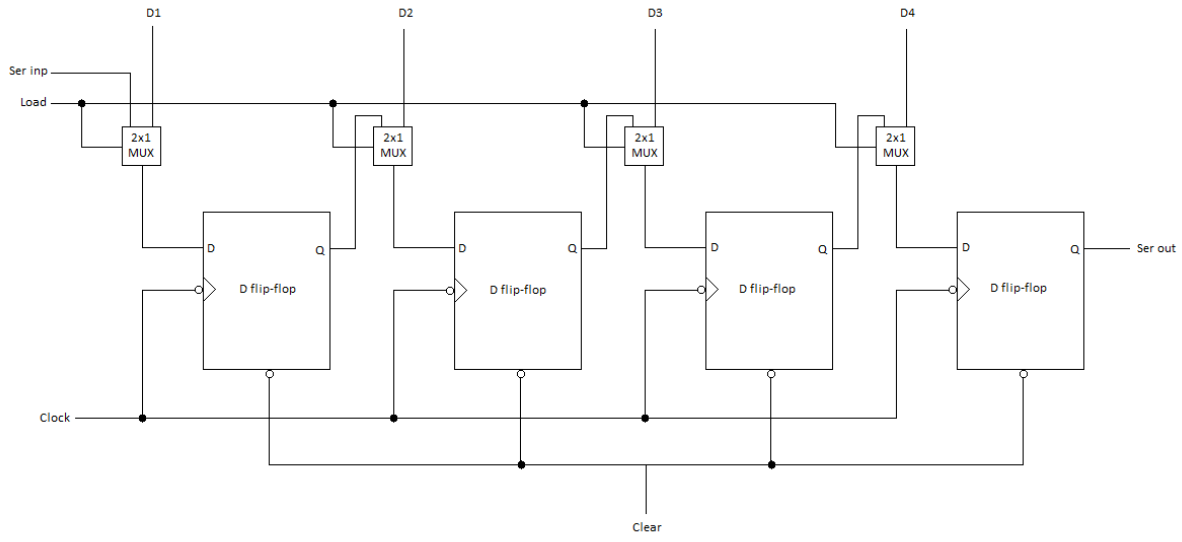

Figure 26

This is the script for simulating the above circuit:

```
!MAINCKT
*node description
*0 – clock input
*1 – clear input
*3 – load/shift select input
*4 – Serial input
*5 – Da
*6 – Db
*7 – Dc
*8 – Dd
*12 – Serial out

SUBCKT          MUX_2           3       4       5       17
SUBCKT          D_FLIPFLOP_PR           0       17      9       13      2       1
SUBCKT          MUX_2           3       9       6       18
SUBCKT          D_FLIPFLOP_PR           0       18      10      14      2       1
SUBCKT          MUX_2           3       10      7       19
SUBCKT          D_FLIPFLOP_PR           0       19      11      15      2       1
SUBCKT          MUX_2           3       11      8       20
SUBCKT          D_FLIPFLOP_PR0          20      12      16      2       1

*setting serial input to 0
!FIX_VOLTAGE  4         0

*setting the load/shift select input by default to shift mode
!FIX_VOLTAGE  3         0

!END_MAINCKT            RT_ANALYSIS

*using MAINCKT_SETUP construct to load the parallel bits into the register
!MAINCKT_SETUP          0       1       3       5       6       7       8
$       0       0       1       0       0       0       0
$       1       1       1       1       0       1       0
$       0       1       1       1       0       1       0
$       0       1       0       0       0       0       0
!END_MAINCKT_SETUP

*scanning the clock node voltage during simulation
!SCAN  0

*printing the serial output data
!PRINT   12
```

- Design a 4 bit multiplier using sequential logic.

  In order to multiply two 4 bit numbers A3A2A1A0 and B3B2B1B0, first we parallely load the first number in a 4 bit PISO register (-ve edge triggered), and the second number into the LSBs of an 8 bit universal shift register (-ve edge triggered), and the MSBs are loaded as 0s. Now the serial out bit from the PISO register is ANDed with every bit from the parallel out of the 8 bit universal shift register and fed into first 8 inputs (first 8bit number) of an 8 bit parallel adder. The output from the adder (neglecting the Cout, since the multiplication of 1111 and 1111 is 11100001 which is 8 bit) is made input to an 8 bit PIPO register (+ve edge triggered). The output to this register is the stored in another 8 bit PIPO register (-ve edge triggered). The output of this register is fed as the other 8 bit number of

the 8 bit parallel adder. Thus the leftmost bit of the number A3A2A1A0, is now available as the serial output of the PISO register. If it is 0 the output of the universal register being ANDed with 0 is fed as 0s to the first input to the parallel adder. Whereas if the serial out bit is 1, the ANDed result is the second number as stored in the universal register. At a positive edge this result gets stored into the first PIPO register connected to the outputs of the parallel adder. Upon the negative edge trigger of the clock, the 4 bit PISO register shifts the next bit of the number A3A2A1A0 right. Also the universal register shifts left inserting a 0 as the rightmost bit into the register. Also the second PIPO register at the output of the first PIPO register now stores the result of the previous addition, providing the new input to the parallel adder. This process continues until all the bits are shifted out of the 4 bit PISO register. The diagram shows a schematic for the multiplier circuit.

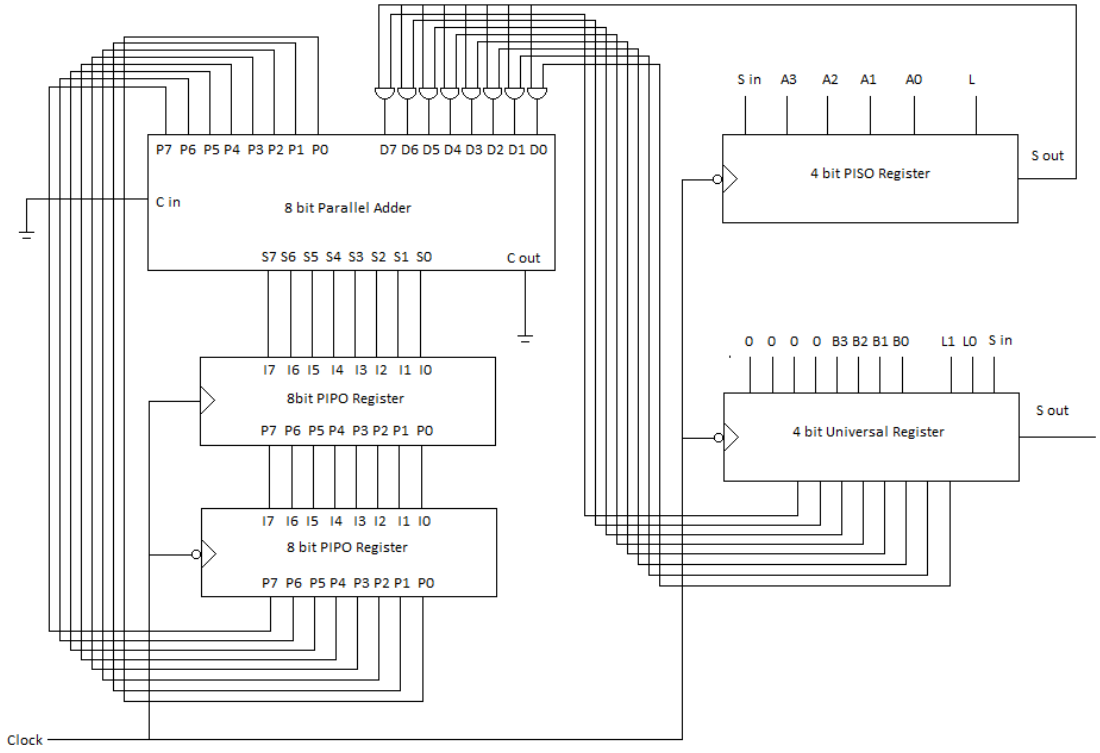

Figure 27

This is the script for simulating the above circuit:

```
*PIPO output register subcircuit definition
!SUBCKT         8bit_REG_OUT 0       1       2       3       4       5       6       7       8       9
10      11      12      13      14      15      16      17

SUBCKT          REGISTER_PIPO_4bit   0       1       2       3       4       5       10      11      12
13
SUBCKT          REGISTER_PIPO_4bit   0       1       6       7       8       9       14      15      16
17

!END_SUBCKT


*8 bit universal input register subcircuit definition
!SUBCKT         8bit_REG_1   0       1       2       3       4       5       6       7       8       9
14      15      16      17      10      11      12      13      18      19      20      21
```

```
SUBCKT          REGISTER_UNIVERSAL_4bit     0       1       2       3       4       18      6       7
8       9       10      11      12      13
SUBCKT          REGISTER_UNIVERSAL_4bit     0       1       2       3       13      5       14      15
16      17      18      19      20      21

!END_SUBCKT

*8 bit parallel adder subcircuit definition
!SUBCKT         8bit_ADDER      0       1       2       3       4       5       6       7       8       9
10      11      12      13      14      15      16      17      18      19      20      21      22      23
24      25

SUBCKT 4bit_ADDER       0       5       6       7       8       13      14      15      16      21      22
23      24      26
SUBCKT 4bit_ADDER       26      1       2       3       4       9       10      11      12      17      18
19      20      25

!END_SUBCKT

!MAINCKT

NOT     0       1

*enabling clock input at node 0
!CLOCK 0        0       50      50      0

SUBCKT          REGISTER_PISO_4bit      0       20      27      23      24      25      26      28

SUBCKT          8bit_REG_1      0       20      10      11      21      22      2       3       4       5
6       7       8       9       12      13      14      15      16      17      18      19

AND_2  28       12      29
AND_2  28       13      30
AND_2  28       14      31
AND_2  28       15      32
AND_2  28       16      33
AND_2  28       17      34
AND_2  28       18      35
AND_2  28       19      36

SUBCKT          8bit_ADDER      37      29      30      31      32      33      34      35      36      38
39      40      41      42      43      44      45      47      48      49      50      51      52      53
54      55

SUBCKT          8bit_REG_OUT 1      20      47      48      49      50      51      52      53      54
60      61      62      63      64      65      66      67

SUBCKT          8bit_REG_OUT 0      20      60      61      62      63      64      65      66      67
38      39      40      41      42      43      44      45


!FIX_VOLTAGE  37      0
!FIX_VOLTAGE  55      0

*carrying out the time analysis on the circuit after data is loaded into the registers
```

```
!END_MAINCKT            TIME_ANALYSIS

*printing the 8 bit output for time analysis
!PRINT  0       38      39      40      41      42      43      44      45

*maincircuit setup construct for inserting data into the input registers before the simulation takes place
!MAINCKT_SETUP      0       20      10      11      27      6       7       8       9       23      24
25      26

$       0       0       0       0       0       0       0       0       0       0       0       0       0
$       1       1       1       1       1       1       1       1       1       1       1       1       1
$       0       1       1       0       0       1       1       1       1       1       1       1       1

!END_MAINCKT_SETUP
```