

TimingAnalyzer Script Documentation

version 0.954

Dan Fabrizio

February 04, 2011

Contents

Python Scripting	1
Scripting Introduction	1
Using Jython	1
Using the Script File Dialog from the TimingAnalyzer program.	1
Using the Command Line	1
Script Examples	2
Drawing a Timing Diagram	2
Creating a D Flip Flop	2
Dumping Signal Values	3
Generating VHDL Test Vectors	4
TimingAnalyzer API	5
TimingDiagram API	5
DigitalClock API	8
The Constructors	8
Adding Clocks	8
Changing Clock Parameters	8
DigitalSignal API	9
The Constructors	9
Adding DigitalSignals	9
DigitalBus API	10
The Constructors	10
Adding DigitalBus Signals	10
Changing DigitalBus Parameters	10
Signal API	11
Setting Parameters	11
Getting Parameters	11
Edges API	12
Pulses API	12
Delay API	12
Constraint API	12
Logic API	13
Indices and tables	14
Index	15

Python Scripting

Contents:

Scripting Introduction

Python scripts can be used to add new features, draw timing diagrams, generate test vectors, convert simulation waveform diagrams into timing diagrams, and check for timing constraint errors in digital logic systems.

Use any text editor to write the scripts and store them in the scripts directory. Select "Script" from the file menu, a dialog listing all the scripts is displayed. Choose the script and hit the "Execute" button. It's best to start the TimingAnalyzer from the command line when developing and testing scripts because debugging information is output to the command window or shell.

Using Jython

Jython is a Python Interpreter written completely in Java. There are 2 ways to execute TimingAnalyzer Python scripts.

- Select the script file from the script dialog window that is displayed when you select the File Menu -> Script.
- From a dos command line window if you run the Windows OS, or from a shell command line window if you run a Linux or Unix OS.

Using the Script File Dialog from the TimingAnalyzer program.

- Start the TimingAnalyzer
- File Menu -> Script or Ctrl T. This brings up the Script selector dialog window.
- Select the script, dff.py, then click the execute button.

The interpreter is started, then both the os and sys modules are imported, then the script is executed.

Using the Command Line

- Change directory to the install directory of the program.
- Start Jython.
- Start the application
- Execute scripts

The listing below shows that I started Jython in the install directory

```
Directory of C:\Apps\TimingAnalyzer_b951
08/17/2009  07:59 PM    <DIR>      .
08/17/2009  07:59 PM    <DIR>      ..
08/07/2009  11:40 AM    <DIR>      docs
08/07/2009  11:40 AM    <DIR>      examples
08/07/2009  11:40 AM    <DIR>      images
08/07/2009  11:40 AM    <DIR>      jlib
08/07/2009  11:40 AM    <DIR>      libs
08/07/2009  11:40 AM    <DIR>      pics
08/07/2009  11:40 AM    <DIR>      scripts
```

```

08/07/2009 11:40 AM <DIR> settings
07/19/2009 10:53 PM 1,107 start_app.py
08/07/2009 11:40 AM <DIR> themes
08/07/2009 11:34 AM 536,901 TimingAnalyzer.jar
                2 File(s) 538,008 bytes
                12 Dir(s) 41,963,491,328 bytes free

```

```

C:\Apps\TimingAnalyzer_b951>java -jar jlib\jython.jar
Jython ...
[Java HotSpot(TM) Client VM (Sun Microsystems Inc.)] on java1.6.0_20
Type "help", "copyright", "credits" or "license" for more information.
>>>

```

Now execute the start_app.py script. This initializes the Jython interpreter so you can use all the TimingAnalyzer classes and methods.

```

>>>execfile('./start_app.py')
>>>execfile('./scripts/dff.py')

```

Script Examples

Drawing a Timing Diagram

The script draw_diagram.py is shown below

```

1 # start a new timing diagram
2 if taApp.getFileName() != "new.tim":
3     taApp.fileNew("TimingDiagram")
4
5 # get the reference to the timing diagram
6 td = taApp.getTimingDiagram()
7
8 # add the signals
9 dclock = td.addDigitalClock("test-clk", 20.0e6, "H")
10 dsignal = td.addDigitalSignal("mem_read", "L")
11 dbus = td.addDigitalBus("mem_add[15:0]", "CC00", "Hex")
12
13 # add pulses to the signal and bus
14 td.addPulse(dsignal, 50.0e-9, 100e-9, "H")
15 td.addPulse(dbus, 30.0e-9, 120e-9, "CC05")
16
17 # add 2 edges to the signal
18 td.addEdge(dsignal, 200.0e-9, "H")
19 td.addEdge(dsignal, 250.0e-9, "L")
20
21 # add 2 edges to the bus
22 td.addEdge(dbus, 180.0e-9, "CCCC")
23 td.addEdge(dbus, 270.0e-9, "CD00")

```

Creating a D Flip Flop

The script dff.py is shown below

```

1 # start a new diagram
2 if taApp.getFileName() != "new.tim":
3     taApp.fileNew("TimingDiagram")
4
5 # get the current timing diagram

```

Dumping Signal Values

```
6 td = taApp.getTimingDiagram()
7
8 # add the CLK, D, and Q signals
9 clk = td.addDigitalClock("CLK",20.0e6,"H")
10 sigD = td.addDigitalSignal("D","L")
11 sigQ = td.addDigitalSignal("Q","L")
12
13 # add a pulse and two edges to the D input
14 sigD.addPulse(75.0e-9,125.0e-9,"H")
15 sigD.addEdge(175.0e-9,"H")
16 sigD.addEdge(225.0e-9,"L")
17
18 # define and add a UserDelay
19 clk2q_min = 2.0e-9
20 clk2q_typ = 4.0e-9
21 clk2q_max = 6.0e-9
22 clk2q_dly = td.addUserDelay("CLK2Q",clk2q_min,clk2q_typ,clk2q_max,"DFF Clock to Q ou
23
24 # get the clock edge list
25 # loop on each edge in the list
26 #     if its the rising edge
27 #         save the edge time
28 #         save the D input state
29 #         if next state doesn't equal last state and
30 #             this new edge time is greater the last edge time
31 #             add the new edge
32 #             add a delay to the edge
33 clk_edge_list = clk.getEdgeList()
34 ls = "L"
35 for clk_edge in clk_edge_list:
36     if clk_edge.getNextState() == "H":
37         et = clk_edge.getPt2()
38         ns = sigD.getStateAtTime(et)
39         if ((ns != ls) and (et + clk2q_min > sigQ.getLastEdgePt2())):
40             sigQ_edge = sigQ.addEdge(et,ns)
41             td.addDelay(clk2q_dly,clk_edge,sigQ_edge)
42         ls = ns
```

Dumping Signal Values

The script `dump_edges.py` is shown below

```
1 import ta_utils
2 from org.dmad.ta import DigitalClock
3
4 outfile = ta_utils.setOutputFileName("dump_edges.txt")
5 td = taApp.getTimingDiagram()
6
7 signal_list = td.getSignalList()
8
9 clock_signal = None
10 for signal in signal_list:
11     if isinstance(signal, DigitalClock):
12         clock_signal = signal
13     else:
14         outfile.write("%s " % (signal.getName()))
15
16 outfile.write("\n")
17
18 for clock_edge in clock_signal.getEdgeList():
```

```

19     if clock_edge.getNextState() == "H":
20         edge_time = clock_edge.getPt2()
21         for sig in signal_list:
22             if sig != clock_signal:
23                 outfile.write("%s " % (sig.getStateAtTime(edge_time)))
24                 outfile.write("\n")
25
26 outfile.close()

```

Generating VHDL Test Vectors

The script `vhdl_test_vectors.py` is shown below

```

1 import ta_utils
2
3 from org.dmad.ta import DigitalSignal
4 from org.dmad.ta import DigitalBus
5 from org.dmad.ta import DigitalClock
6
7 out_file = ta_utils.setOutputFileName("vhdl_test_vectors.txt")
8 td = taApp.getTimingDiagram()
9 ts = td.getTimeScale()
10 ts_text = ta_utils.getTimeScaleText(ts)
11
12 signals = td.getSignalList()
13 for sig in signals:
14     i = 0
15
16     out_file.write("%s <= " % (sig.getName()))
17
18     for edge in sig.getEdgeList():
19         if isinstance(sig, DigitalSignal) or isinstance(sig, DigitalClock):
20             line = "'%s'" % (edge.getNextState())
21             if i != 0:
22                 line = "    %s after %s %s" % \
23                     (line,
24                      ta_utils.round3Places(edge.getPt2() / ts),
25                      ts_text)
26             elif isinstance(sig, DigitalBus):
27                 state_format = {
28                     "Hex": "X\\\"",
29                     "Bin": "\\\""
30                 }
31
32                 sStart = state_format[sig.getStateFormat()]
33
34                 line = "%s%s\\" % (sStart, edge.getNextState())
35                 if i != 0:
36                     line = "    %s after %s %s" % \
37                         (line,
38                          ta_utils.round3Places(edge.getPt2() / ts),
39                          ts_text)
40
41                 if ( i == len(sig.getEdgeList())-1):
42                     out_file.write("%s;\n\n" % (line))
43                 else:
44                     out_file.write("%s,\n" % (line))
45
46                 i += 1
47 out_file.close()

```

TimingAnalyzer API

The TimingAnalyzer class is the main application class. It controls most all of the GUI related functions in the main window like multiple tabs for timing diagrams, the menus, the toolbar, and the status bar. It also controls switching from the timing diagram view to the image view and other views that might be added in the future. Possibly a "transaction editor" diagram view, or "analysis reports" view.

fileNew (*String typeDiagram*)

Opens a new timing diagram file. This will start a new timing diagram in a new tab with the title new.tim. Set typeDiagram to "TimingDiagram". Maybe someday there will be others, like a transaction diagram.

```
taApp.fileNew("TimingDiagram")
```

fileNew (*String type, String dir, String fileName*)

Opens a new timing diagram file in the specified directory dir and file name.

```
taApp.fileNew("TimingDiagram", "/home/danf/projects/test_fpga", "pci_write.tim")
```

getFileName ()

Returns a String that contains the name of the file in the current tab window

```
fileName = taApp.getFileName()
```

getTimingDiagram ()

Returns the TimingDiagram object for the currently displayed tab window

```
timDiagram = taApp.getTimingDiagram()
```

TimingDiagram API

The TimingDiagram class controls all functions related to drawing the timing diagram. Multiple timing diagrams can be viewed and edited in separate tab windows. A command manager implements Undo and Redo functions for most all the functions. Some of the functions use multiple commands that are executed together as one command from a linked list of commands.

addDigitalClock (*DigitalClock dclk*)

Adds a DigitalClock to the timing diagram. A DigitalClock instance is required as the argument. This function is undoable and redoable and is the same as adding a clock from the GUI

```
td.addDigitalClock(myClock)
```

addDigitalClock (*String name, double frequency, String startState*)

Adds a DigitalClock to the timing diagram. It creates a new instance of a DigitalClock object using the arguments. This function is undoable and redoable.

Returns a DigitalClock object that can be used by other routines. The name, frequency, duty cycle, rise time, fall time, start delay, start state, and height can be changed using updateDigitalClock()

```
dclk = td.addDigitalClock("CLK25", 25e6, "H")
```

addDigitalClock (*String name, double frequency, String startState, int dutyCycle*)

Adds a DigitalClock to the timing diagram. It creates a new instance of a DigitalClock object using the arguments. This function is undoable and redoable.

Returns a Digital Clock object that can be used by other routines. The name, frequency, duty cycle, rise time, fall time, start delay, start state, and height can be changed using updateDigitalClock()

```
dclk = td.addDigitalClock("CLK25", 25e6, "H", 40)
```

addDigitalBus (*DigitalBus myBus*)

Adds a DigitalBus to the timing diagram. The DigitalBus instance is required as the argument. This function is undoable and redoable and is the same as adding a clock from the GUI

```
td.addDigitalBus(myBus)
```

addDigitalBus (*String name, String startState, String stateFormat*)

Adds a DigitalBus to the timing diagram. It creates a new instance of a DigitalBus object using the arguments. This function is undoable and redoable.

Returns a DigitalBus object that can be used by other routines needed to do timing analysis. The parameters can be changed using updateDigitalClock. The stateFormat can be "Hex", "Bin", "Dec", or "Text"

```
dbus = td.addDigitalBus("ADDR[15:0]", "Z", "Hex")
```

addDigitalSignal (*String name, String startState*)

Adds a DigitalSignal to this timing diagram. It creates a new instance of a DigitalSignal object using the arguments. This function is undoable and redoable.

Returns a DigitalSignalObject that can be used by other routines. The parameters can be changed using updateDigitalSignal(). The start state of signal can be "H" or "L" or "Z"

```
dsig = td.addDigitalSignal("GNT_N", "H")
```

addDigitalSignal (*DigitalSignal dsig*)

Adds a DigitalSignal to the timing diagram. A DigitalSignal instance is required as the argument. This function is undoable and redoable. The parameters can be changed using updateDigitalSignal()

```
td.addDigitalSignal(mySignal)
```

addEdge (*Signal sig, double edgeTime, String newState*)

Adds an Edge to a Signal. The edge is added to the signal at the specified time. The edge object contains the time and new state of the edge. Adding edges should only be used when the new edge will be last edge in the signal. Use addPulse() to change the state of a signal otherwise. This function is undoable and redoable.

Returns the Edge object

```
thisEdge = td.addEdge(myBus, 340.0e-9, "FF22")
```

addPulse (*Signal sig, double time1, double time2, String newState*)

Adds a Pulse to a Signal. The pulse occurs from time1 to time2. The newState argument defines the state of the pulse. This actually adds 2 edges to the signal. This function is undoable and redoable

```
td.addPulse(mySignal, 50.0e-9, 100.0e-9, "H")
```

addPulseWidthLabel (*Edge e1, Edge e2, String label, String labelPos*)

Adds a PulseWidthLabel between two edges. The label is can be displayed on the left, right, or center of the edges.

```
td.addPulseWidthLabel(edge1, edge2, "t_min", "Center")
```

addStateBar (*Edge ed, String label, String lineType, int offsetX, int offsetY*)

Adds a StateBar to an Edge. The label is displayed to the right of the line on top of the waveform. The label can be moved by specifying an offset in pixels in the X and Y directions. The line type string can be "Dashed" or "Solid".

This function is undoable and redoable.

```
td.addStateBar(clk_edge, "", "Dashed", 0, 0)
```

addTextAboveSignal (*String sigName, String text, double time, int offset*)

Adds a text label to the timing diagram above a signal. The following example adds the label "Write Transaction" 20 pixels above the top of the write_cmd signal at 50 ns

```
td.addTextAboveSignal("write_cmd", "Write Transaction", 50.0e-9, 20)
```

addTextBelowSignal (*String sigName, String text, double time, int offset*)

Adds a text label to the timing diagram below a signal. The following example adds the label "Write Transaction" 20 pixels below the bottom of the write_cmd signal at 50 ns

```
td.addTextBelowSignal("write_cmd", "Write Transaction", 50.0e-9, 20)
```

addTimeWarp (*double startTime, double endTime*)

Adds a TimeWarp that begins at the startTime and ends at the endTime. startTime should be less than endTime.

```
td.addTimeWarp(50.0e-9, 800.0e-9)
```

addUserDelay (*String delayName, Edge ed1, Edge ed2, double min, double typ, double max*)

Add a User Defined Delay. The Delay is added to the second edge. Minimum, Typical and Maximum delay times are required as arguments and are used when a timing analysis is executed. The time of the second edge time is the first edge time plus the delay. This function is undoable and redoable and is the same as adding a Delay from the GUI. All other Delay parameters are set to the default values.

Returns the Delay object

```
tpDelay = td.addUserDelay("tpd", edge1, edge2, 4.0e-9, 8.0e-9, 12.0e-9)
```

addUserConstraint (*String myConstraint, Edge e1, Edge e2, double min, double typ, double max*)

Add a User Defined Constraint. The Constraint is added to the second edge. Minimum, Typical and Maximum constraint times are required as arguments and are used when a timing analysis is executed. This function is undoable and redoable and is the same as adding a Constraint from the GUI. All other Constraint parameters are set to the default values.

Returns the new Constraint

```
myConstraint = td.addUserConstraint("tsetup", edge1, edge2, 12.0e-9, 12.0e-9, 12.0e-9)
```

getTimeScale ()

Returns the time scale of the timing diagram. Valid values are 1.e-3 1.0e-6 1.0.e-9 1.0e-12

```
tScale = td.getTimeScale()
```

getSignalList ()

Returns an ArrayList which contains all the signals in the current timing diagram

```
sigList = td.getSignalList()
```

setEndTime (*double ts*)

Sets the end time of the timing diagram.

```
td.setEndTime(300.0e-9)
```

setStartTime (*double st*)

Sets the start time of the timing diagram.

```
td.setStartTime(20.0e-9)
```

setTimePerDivision (*double tpd*)

Sets the time for each large division. tpd should not include the timeScale

```
td.setTimePerDivision(20.0)
```

setTimeScale (*double ts*)

Sets the time scale for the timing diagram. Valid values are 1.0e-3, 1.0e-6, 1.0e-9. and 1.0e-12

```
td.setTimeScale(1.0e-9)
```

zoomIn (*double startTime, double endTime*)

This zooms in to a window in time in the timing diagram.

```
td.zoomIn(300.0e-9, 800.0e-9)
```

zoomIn ()

This divides the time per division by 2.0.

```
td.zoomIn()
```

zoomOut ()

This multiplies the time per division by 2.0.

```
td.zoomOut()
```

DigitalClock API

The constructors are used to specify the parameters of the clock. Synchronous events are usually connected to edges in the DigitalClock. Increasing the clock frequency is a common operation used to check if the logic design still meets all the constraints.

All signals below the clock are considered to synchronous to this clock when adding pulses to other signals. Jitter is not functional at this time and should be set to 0.0. It will be used to model the accuracy of a clock in ppms.

The Constructors

DigitalClock (*TimingAnalyzer taApp, String name, double freq, int dutyCycle, double riseTime, double fallTime, double jitter, double startDelay, String startState*)

The signal height and space above are set to the default values.

DigitalClock (*TimingAnalyzer taApp, String name, double freq, String startState*)

The rise time, fall time, start delay, duty cycle, signal height, and space above are set to the default values.

Adding Clocks

addDigitalClock (*DigitalClock dclk*)

Adds a DigitalClock to this timing diagram. The DigitalClock instance is required as the argument.

```
timDiagram.addDigitalClock(myClock)
```

addDigitalClock (*String name, double frequency, String startState*)

Adds a DigitalClock to this timing diagram. It creates a new instance of a DigitalClock object. Returns a Digital Clock object.

```
dclk = timDiagram.addDigitalClock("CLK25", 25e6, "H")
```

addDigitalClock (*String name, double frequency, String startState, int dutyCycle*)

Adds a DigitalClock to this timing diagram. It creates a new instance of a DigitalClock object. Returns a Digital Clock object.

```
dclk = timDiagram.addDigitalClock("CLK25", 25e6, "H", 40)
```

Changing Clock Parameters

setFrequency (*double frequency*)

Set the Clock Frequency. This can be used to test faster clocks in systems to see if the design meets the defined timing constraints.

```
dclk.setFrequency(50e6)
```

getFrequency ()

Returns a double that equals the Clock Frequency.

```
currentFrequency = dclk.getFrequency()
```

setDutyCycle (*int dutycycle*)

Set the Clock Duty Cycle.

```
dclk.setDutyCycle(40)
```

getDutyCycle ()

Returns an integer that is the clock duty cycle.

```
dutyCycle = dclk.getDutyCycle()
```

setJitter (*double jitter*)

Set the Clock Jitter. This can be used to model the clock accuracy in ppms. Currently, this is not functional and should be set to 0.0

getJitter ()

Returns a double that is the Clock Jitter.

```
clkJitter = dclk.getJitter()
```

setStartDelay (*double startDelay*)

Set the Clock Start Delay. This can be used to shift a clock in time, so you can make a clock that is out of phase with another clock. A feature coming will link the clocks together.

```
dclk.setStartDelay(25.0e-9)
```

getStartDelay ()

Returns a double that is the Clock Start Delay.

```
startDelay = dclk.getStartDelay()
```

DigitalSignal API

The Constructors

DigitalSignal (*TimingAnalyzer taApp, String name, double riseTime, double fallTime, String startState*)

The signal height and space above are set to the default values.

DigitalSignal (*TimingAnalyzer taApp, String name, String startState*)

The signal height, space above, rise time, and fall time are set to the default values.

Adding DigitalSignals

addDigitalSignal (*String name, String startState*)

Adds a DigitalSignal to this timing diagram. It creates a new instance of a DigitalSignal object. Returns a DigitalSignal object that can be used by other routines needed to do timing analysis. The start state of signal can be "H" or "L" or "Z".

```
dsig = timDiagram.addDigitalSignal("GNT_N", "H")
```

addDigitalSignal (*DigitalSignal dsig*)

Adds a DigitalSignal to this timing diagram. A DigitalSignal instance is required as the argument.

```
timDiagram.addDigitalSignal(mySignal)
```

DigitalBus API

The Constructors

Groups of signals are represented by a DigitalBus. For example, a processor address bus or data bus. The DigitalBus name must be in the format bus_name[MS:LS]. Where MS is the most significant signal position and LS in the least significant signal position.

DigitalBus (*TimingAnalyzer taApp, String name, double riseTime, double fallTime, String startState, String stateFormat*)

The signal height and space above are all set to the default values.

DigitalBus (*TimingAnalyzer taApp, String name, String startState, String stateFormat*)

The signal height, space above, rise time, and fall time are all set to the default values.

Adding DigitalBus Signals

addDigitalBus (*DigitalBus myBus*)

Adds a DigitalBus to this timing diagram. The DigitalBus instance is required as the argument.

```
timDiagram.addDigitalBus(myBus)
```

addDigitalBus (*String name, String startState, String stateFormat*)

Adds a DigitalBus to this timing diagram. It creates a new instance of a DigitalBus object. Returns a DigitalBus object.

```
dbus = timDiagram.addDigitalBus("ADDR[15:0]", "Z", "Hex")
```

Changing DigitalBus Parameters

getBusName (*String name*)

This returns a String that is the name part of name[MS:LS]. The example below returns "ADDR".

```
busName = dbus.getBusName("ADDR[15:0]")
```

getBusNameX (*String name*)

Returns an integer that is the DigitalBus MS signal index. The example below returns 15.

```
msIndex = dbus.getBusNameX("ADDR[15:0]")
```

getBusNameY (*String name*)

Returns an integer that is the DigitalBus LS signal index. The example below returns 0.

```
lsIndex = dbus.getBusNameY("ADDR[15:0]")
```

getStateFormat ()

Returns a String that is the DigitalBus State Format. It could be "Hex", "Bin", "Dec", or "Text".

```
stateFormat = dbus.getStateFormat()
```

setStateFormat (*String stateFormat*)

Set the DigitalBus State Format. This sets the current state format to a new value and changes all the bus values to the new state format.

```
dbus.setStateFormat("Hex")
```

getNumBits ()

Returns an integer that is the number of signals in the DigitalBus.

```
numSignals = dbus.getNumBits()
```

Signal API

The Signal is the base class for the DigitalClock, DigitalSignal, and DigitalBus. Any one of these classes can use the methods described below.

Setting Parameters

setName (String sigName)

Sets the name of the signal

```
mySig.setName("R/W")
```

setHeight (int sigHeight)

Sets the height, in pixels, of the signal.

```
myBus.setHeight(30)
```

setFallTime (double fallTime)

Sets the fall time of every edge in the signal.

```
myClock.setFallTime(4.0e-9)
```

setRiseTime (double riseTime)

Sets the rise time of every edge in the signal.

```
myClock.setRiseTime(4.0e-9)
```

setStartState (String startState)

Sets the start state of the signal.

```
myClock.setStartState("L")
```

setSpaceAbove (int spaceAbove)

Sets the blank space, in pixels, above the signal.

```
pciClock.setSpaceAbove(50)
```

Getting Parameters

getName ()

Returns a String that is the name of the signal.

```
busName = pciAddrBus.getName()
```

getHeight ()

Returns an int that is the height of the signal in pixels.

```
pciAddrBusHeight = pciAddrBus.getHeight()
```

getFallTime ()

Returns a double that is the fall time of the edges in the signal.

```
fallTime = readSig.getFallTime()
```

getRiseTime ()

Returns a double that is the rise time of the edges in the signal.

```
riseTime = writeSig.getRiseTime()
```

getStartState ()

Returns a String that is the start state of the signal.

```
startState = mySig.getStartState()
```

getSpaceAbove ()

Returns an int that is the space above the signal in pixels.

```
spaceAbove = sig.getSpaceAbove()
```

Edges API

addEdge (Signal sig, double edgeTime, String newState)

Adds an Edge to a Signal. The edge is added to the signal at the specified time. The edge object contains the time and new state of the edge. Adding edges should only be used when the new edge will be last edge in the signal. You can use addPulse() to change the state of a signal at any time.

Returns the Edge object.

```
thisEdge = timDiagram.addEdge(pci_data, 340.0e-9, "FF22")
```

addEdge (Signal sig, double minTime, double maxTime, String newState)

Adds an Edge to a Signal with showing min and max times. The Edge is added to the list of edges for the signal and then sorted by edge times.

Returns the Edge object.

```
thisEdge = timDiagram.addEdge(pci_data, 340.0e-9, 342e-9, "FF22")
```

Pulses API

addPulse (Signal sig, double time1, double time2, String newState)

Adds a Pulse to the Signal. The pulse is added to the signal from time1 to time2. The newState argument defines the state of the pulse. This actually adds 2 edges to the signal.

```
timDiagram.addPulse(read, 50.0e-9, 100.0e-9, "H")
```

Delay API

addUserDelay (String delayName, Edge ed1, Edge ed2, double min, double typ, double max, String note)

Add a User Defined Delay to the diagram database. The user delay stores the minimum, typical and maximum delay times and a text string label.

Returns the UserDelay object.

```
clk2q_udly = td.addUserDelay("CLK2Q", clk2q_min, clk2q_typ, clk2q_max, "DFF Clock to Q out")
```

addDelay (UserDelay usrDly, Edge srcEdge, Edge destEdge)

Adds a Delay to the timing diagram. Requires a UserDelay and 2 Edges as arguments.

Returns the Delay object.

```
clk2q_dly = td.addDelay(clk2q_udly, clk_edge, sigQ_edge)
```

Constraint API

addUserConstraint (String name, double min, double typ, double max, String note)

Add a UserConstraint. Minimum, Typical and Maximum constraint times are required as arguments and are used when a timing analysis is executed.
Returns the new UserConstraint.

```
setup_ucnstrnt = td.addUserConstraint("tsetup", 12.0e-9, 12.0e-9, 12.0e-9, "min pulsewidth")
```

addConstraint (*UserConstraint usrCnstrnt, Edge srcEdge, Edge destEdge*)

Adds a Constraint to the timing diagram. Requires a UserConstraint and 2 Edges as arguments.

Returns the Constraint object.

```
setup_cnstrnt = td.addConstraint(setup_ucnstrnt, clk_edge, sigQ_edge)
```

Logic API

addBuffer (*String dinName, String tphlName, String tplhName*)

- *dinName* is the name of the buffer input signal. It can be a DigitalSignal, DigitalBus, or DigitalClock.
- *tphl* (Optional) is the name of a User Delay which specifies the propagation delay when switching from "H" to "L". If not specified, use "";
- *tplh* (Optional) is the name of a User Delay which specifies the propagation delay when switching from "L" to "H". If not specified, use "";

Example 1: A Buffer with no delays.

```
td.addBuffer("CLK", "", "")
```

Example 2: A Buffer with tphl and tplh delays.

```
td.addUserDelay("tphl", tphlMin, tphlTyp, tphlMax, "tphl prop delay part xyz");
td.addUserDelay("tplh", tplhMin, tplhTyp, tplhMax, "tplh prop delay part xyz");

td.addBuffer("CLK", "tphl", "tplh");
```

addInverter (*String dinName, String tphlName, String tplhName*)

- *dinName* is the name of the Inverter input signal. It can be a DigitalSignal, DigitalBus, or DigitalClock.
- *tphl* (Optional) is the name of a User Delay which specifies the propagation delay when switching from "H" to "L". If not specified, use "";
- *tplh* (Optional) is the name of a User Delay which specifies the propagation delay when switching from "L" to "H". If not specified, use "";

Example 1: An Inverter with no delays.

```
td.addInverter("CLK", "", "")
```

Example 2: An Inverter with tphl and tplh delays.

```
td.addUserDelay("tphl", tphlMin, tphlTyp, tphlMax, "tphl prop delay part xyz");
td.addUserDelay("tplh", tplhMin, tplhTyp, tplhMax, "tplh prop delay part xyz");

td.addInverter("CLK", "tphl", "tplh");
```

addDFF (*String dinName, String clkName, boolean posEdge, String clk2qName, String enName, String rstName, String preName*)

- *dinName* is the name of the D input signal. It can be a DigitalSignal or a DigitalBus.
- *clkName* is the name of the CLK input signal.
- *posEdge* is true for rising edge triggered and false for falling edge triggered.
- *clk2qName* (Optional) is the name of a User Delay which specifies the clock to Q output delay. If not specified, use "".

- `enName` (Optional) is the name of the ENABLE input signal. If not specified, use "".
- `rstName` (Optional) is the name of the synchronous RESET input signal. If not specified, use "".
- `preName` (Optional) is the name of the synchronous PRESET input signal. If not specified, use "".

Example 1: A positive edge triggered DFF with enable and `clk2q` delay.

```
td.addUserDelay("clk2q",clk2qmin,clk2qTyp,clk2qMax, "DFF clock to Q delay part xyz");
td.addDFF("FIFO_RD","CLK",true,"clk2q","FIFO_RD_EN","","");
```

Example 2: A positive edge triggered DFF with enable and `clk2q` delay and synchronous reset.

```
td.addUserDelay("clk2q",clk2qmin,clk2qTyp,clk2qMax, "DFF clock to Q delay part xyz");
td.addDFF("FIFO_RD","CLK",true,"clk2q","FIFO_RD_EN","FIFO_RST","");
```

addCounter (*String cntrName, String clkName, boolean posEdge, String clk2qName, boolean upCounter, String enName, String ldName, String pinName, String rstName, String preName*)

- `cntrName` is the name of the counter output signal. A new DigitalBus will be added to the diagram with this name.
- `clkName` is the name of the CLK input signal.
- `posEdge` is true for rising edge triggered and false for falling edge triggered.
- `clk2qName` (Optional) is the name of a User Delay which specifies the clock to Q output delay. If not specified, use "".
- `upCounter` is true for an up counter and false for a down counter.
- `enName` (Optional) is the name of the ENABLE input signal. If not specified, use "".
- `ldName` (Optional) is the name of the counter LOAD signal input. When active, the value of the parallel input (`pinName`) is loaded into the counter on the next clock edge.
- `pinName` (Optional) is the name of the parallel input signal. This should be used with `ldName`. If not specified, use "".
- `rstName` (Optional) is the name of the synchronous RESET input signal. If not specified, use "".
- `preName` (Optional) is the name of the synchronous PRESET input signal. If not specified, use "".

Example 1: A 8 bit up counter, positive edge triggered, with `clk2q` delay, count enable and synchronous reset.

```
td.addUserDelay("clk2q",clk2qmin,clk2qTyp,clk2qMax, "Counter clock to Q delay part xyz");
td.addCounter("FIFO_WR_CNTR[7:0]","CLK",true,"clk2q",true,"FIFO_RD_CNTR_EN","","","RS");
```

Indices and tables

- *genindex*

Index

A

addBuffer() (built-in function)
addConstraint() (built-in function)
addCounter() (built-in function)
addDelay() (built-in function)
addDFF() (built-in function)
addDigitalBus() (built-in function) [1] [2] [3]
addDigitalClock() (built-in function) [1] [2] [3] [4] [5]
addDigitalSignal() (built-in function) [1] [2] [3]
addEdge() (built-in function) [1] [2]
addInverter() (built-in function)
addPulse() (built-in function) [1]
addPulseWidthLabel() (built-in function)
addStateBar() (built-in function)
addTextAboveSignal() (built-in function)
addTextBelowSignal() (built-in function)
addTimeWarp() (built-in function)
addUserConstraint() (built-in function) [1]
addUserDelay() (built-in function) [1]

D

DigitalBus() (built-in function) [1]
DigitalClock() (built-in function) [1]
DigitalSignal() (built-in function) [1]

F

fileNew() (built-in function) [1]

G

getBusName() (built-in function)
getBusNameX() (built-in function)
getBusNameY() (built-in function)
getDutyCycle() (built-in function)
getFallTime() (built-in function)
getFileName() (built-in function)
getFrequency() (built-in function)
getHeight() (built-in function)
getJitter() (built-in function)

getName() (built-in function)
getNumBits() (built-in function)
getRiseTime() (built-in function)
getSignalList() (built-in function)
getSpaceAbove() (built-in function)
getStartDelay() (built-in function)
getStartState() (built-in function)
getStateFormat() (built-in function)
getTimeScale() (built-in function)
getTimingDiagram() (built-in function)

S

setDutyCycle() (built-in function)
setEndTime() (built-in function)
setFallTime() (built-in function)
setFrequency() (built-in function)
setHeight() (built-in function)
setJitter() (built-in function)
setName() (built-in function)
setRiseTime() (built-in function)
setSpaceAbove() (built-in function)
setStartDelay() (built-in function)
setStartState() (built-in function)
setStartTime() (built-in function)
setStateFormat() (built-in function)
setTimePerDivision() (built-in function)
setTimeScale() (built-in function)

Z

zoomIn() (built-in function) [1]
zoomOut() (built-in function)