Lecture 4. Automating Circuit Simulations (Mulan/Simba Tutorial)

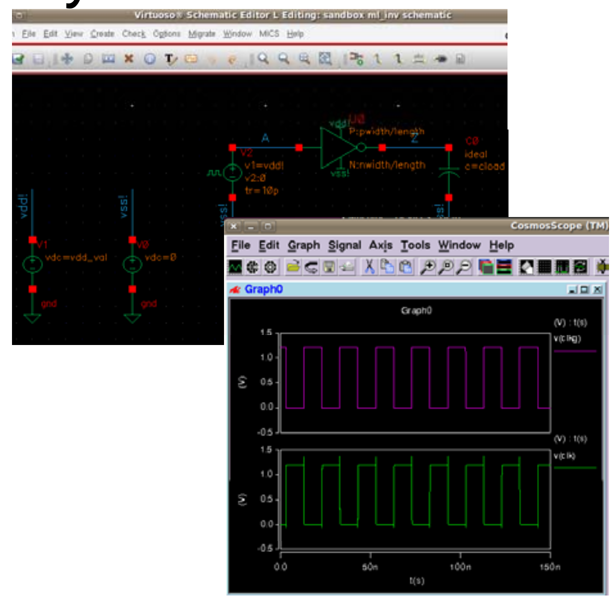
Jaeha Kim
Mixed-Signal IC and System Group (MICS)
Seoul National University
jaeha@ieee.org

Objectives

- To automate tedious jobs in circuit simulation
  - Sweeping parameters (sizes, bias conditions, etc.)
  - Sweeping PVT corners
  - Post-process the results and plot as graphs
- To share methodologies across different technologies
  - Parameterize the technology differences
  - Reuse and ease process migrations
- To share methodologies across multiple circuits of the same purpose/functionality
  - e.g. one gain measuring script for various op-amp topologies

Scripting Language

- Python, Perl, Tcl/Tk, Ruby, SKILL, …
- Typical IC design process involves many different tools
  - e.g. in a schematic-driven design flow:
    - Draw circuits in a schematic editor
    - Extract circuit netlists
    - Run SPICE simulations
    - Plot waveforms
    - Process the results (e.g. Matlab)
    - Revise circuits based on the results



- Scripts can help automate repetitive processes that do not require any human brain!

Iterative Design Process

- Many system designers or programmers dream about "*Correct by Construction*"
  - That is, you get a working system at the first try simply by putting together its building blocks
- However, reality is closer to "*Construct By Correction*"
  - You build a system by continually correcting it
  - It means repetitive simulation/verifications of wide variety of circuit choices – iterative refinements
- Never think that you will do that job only once (and do it by hand) – it's likely that you will do it more than 50x!
  - Hopefully this motivates you to spend extra efforts upfront

The First Look at Python Script

```
#!/usr/bin/env python

def sum(n):
    """ computes a sum of integers from 1 to n """
    if n == 1: return 1
    else: return n + sum(n-1)

def prod(n):
    """ computes a product of integers from 1 to n """
    result = 1
    for i in range(1,n+1):
       result *= i
    return result

print "Hello World!"
print "1+2+...+10=", sum(10)
print "1*2*...*10=", prod(10)
```

Executing a Python Script

```
unix> python myfirstscript.py
Hello World!
1+2+...+10= 55
1*2*...*10= 3628800

Or,

unix> chmod +x myfirstscript.py
unix> ./myfirstscript.py
Hello World!
1+2+...+10= 55
1*2*...*10= 3628800
```

- You can also execute python commands interactively in a shell (e.g. type just ‘python’)

Why Python?

	MATLAB	Perl	Python
Process Control	Poor	Good	Good
Text Manipulation	Poor	Good	Good
Scientific Library	Good	Fair	Good
Plot graphics	Good	Fair	Good
Shell support	Yes	No	Yes
Native OOP	Fair	Fair	Good
Embedding	No	Yes	Yes

References on Python

- Python in General
  - Beginner's guide: http://wiki.python.org/moin/BeginnersGuide
  - Quick references: http://rgruet.free.fr/#QuickRef
- NumPy/SciPy: Scientific Library
  - http://www.scipy.org/Tentative_NumPy_Tutorial
  - http://www.scipy.org/NumPy_for_Matlab_Users
- Matplotlib/Pylab: Plotting Library
  - http://matplotlib.sourceforge.net/
- Embedded Python (empy)
  - http://www.alcyone.com/software/empy/

Python Setups

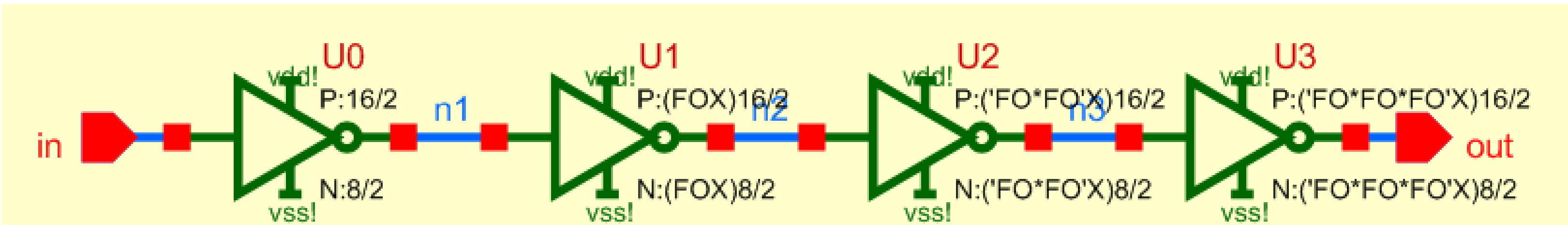
- On Ubuntu, execute the following commands to install Python and its necessary libraries

```
unix> sudo apt-get install python
unix> sudo apt-get install python-dev
unix> sudo apt-get install python-numpy
unix> sudo apt-get install python-scipy
unix> sudo apt-get install python-matplotlib
unix> sudo apt-get install python-empy
unix> sudo apt-get install python-sympy
unix> sudo apt-get install python-sqlite
unix> sudo apt-get install libsqlite3-dev

unix> cd /cad/packages/python/sqlite3-backup
unix> sudo python setup.py install
```

First Example: Measuring Inverter Delays

- Circuit Schematic: inv_delay



HSPICE Simulation Deck

```
* inv_delay.hsp: measure inverter delays

.lib '/cad/tech/ptm045/spice/models.lib' TT
.option scale=22.5n
.option post accurate
.param supply=1.0

.param FO=4
.include 'netlist/sandbox.inv_delay.sp'

.global vdd! vss!
vdd     vdd! gnd    dc=supply
vss     vss! gnd    dc=0
vin     in gnd      pulse( 0 supply 1ns 5ps 5ps 5ns 10ns )

.tran 1p 30n
.measure tran delay_r trig v(n1) val='supply/2' fall=2
+                     targ v(n2) val='supply/2' rise=2
.measure tran delay_f trig v(n1) val='supply/2' rise=2
+                     targ v(n2) val='supply/2' fall=2
.end
```

Python Script using Mulan

- Launch simulation and process the results

```
#!/usr/bin/env python
"""
inv_delay.py
measuring inverter delays.
"""

from sim import mulan

M = mulan("inv_delay.ml")

for row in M.sw_iter():
    runbase = row.hspice('inv_delay.hsp')
    row.readmeas(runbase + '.mt0')
    row.delay_avg = (row.delay_r + row.delay_f)/2.0
    row.list()

M.save()
```

Getting Started with Mulan

- First, import a Python class named ‘mulan’

```
from sim import mulan
```

- Then, make an instance either attached to a file or residing in the memory

```
M = mulan(‘my_results.ml’)
```

or

```
M = mulan()
```

Accessing Mulan Data Structure

- Contents of a mulan instance is accessed “row-by-row”
  - Each row corresponds to a sweep case
  - M.sw_iter() returns one row object at a time
- With a row object, you can run simulations, read measurement results, and compute new data
  - More on using row later

```
for row in M.sw_iter():
    runbase = row.hspice('inv_delay.hsp')
    row.readmeas(runbase + '.mt0')
    row.delay_avg = (row.delay_r + row.delay_f)/2.0
    row.list()
```

File I/O Methods

- You can load and save the contents of a mulan instance from/to a file by using the following commands:
  - Note: Mulan uses SQLITE3 as a backend database and the file will store the data based on its internal format
- File I/O methods are:
  - **M.load([filename])** : reads data from the file
  - **M.save([filename])** : writes data to the file
  - **M.revert()** : reverts data to the previously saved state
  - **M.close()** : closes the sqlite DB connection of the mulan instance – unsaved data will be lost!

Example 1: Demo

```
unix> inv_delay.py
 delay_r     delay_f      temper   alter_shar   delay_avg
 64.9500p    67.1600p     25.0000  1.00000      66.0550p
```

Accessing Mulan/Simba Documentations

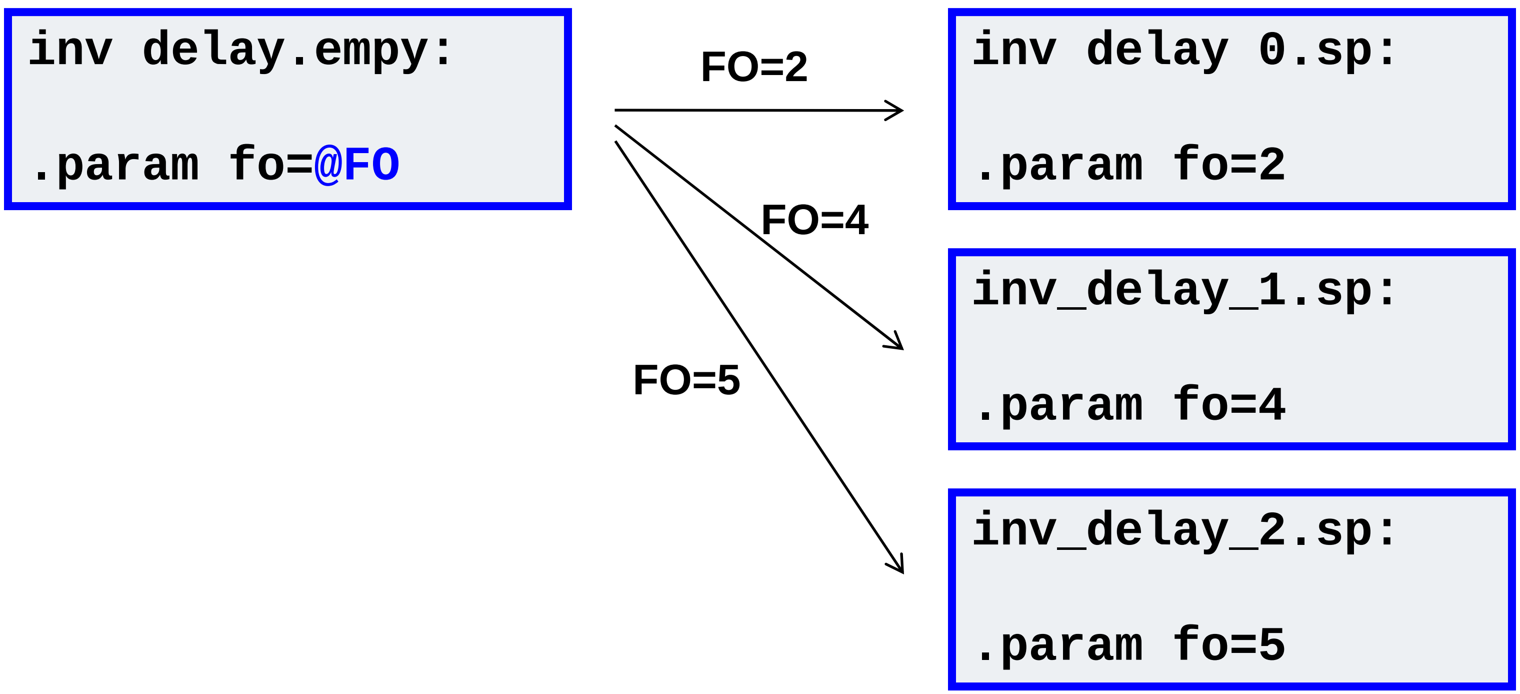
- Strings embedded within the source codes become the documentations (similar to Matlab)
- To access these documentations:
  - unix> pydoc sim.mulan
  - unix> pydoc sim.mulan.<function_name>
- Or, equivalently, from the Python shell:
  - python>> help(sim.mulan)
  - python>> help(sim.mulan.<function_name>)
- Use these methods to find more complete, detailed information about how to use Mulan/Simba

Sweeping Parameters

- You will often find a need to repeat the same simulation for a variety of changes
  - But, SWEEP and ALTER in SPICE are limited
- Possible scenarios are:
  - Sweep parameters such as PVT conditions or sizes (basic)
  - Sweep 8-bit binary codes as an input to a DAC
  - Measure inverter delays across multiple IC technologies
  - Compare gain and bandwidth of various op-amp topologies
- To do this, we need a simulation deck that can take a different form depending on parameter values

Using Embedded Python

- You can embed python variables within the simulation deck to parameterize its appearance



Using Embedded Python (2)

- You can play more advanced tricks with empy
  - For more information, see http://www.alcyone.com/pyos/empy

```
@[for i in range(5)]
R@i n@i n@(i+1) r=@(2**i)
@[end for]
```

→

```
R0 n0 n1 r=1
R1 n1 n2 r=2
R2 n2 n3 r=4
R3 n3 n4 r=8
R4 n4 n5 r=16
```

```
@[for i in range(4)]
@[if i % 2]
X@i n@i n@(i+1) buf_odd
@[else]
X@i n@i n@(i+1) buf_even
@[end if]
@[end for]
```

→

```
X0 n0 n1 buf_even
X1 n1 n2 buf_odd
X2 n2 n3 buf_even
X3 n3 n4 buf_odd
```

Example 2: Sweeping Parameters

```
* inv_delay.empy: measure inverter delays

.lib '/cad/tech/ptm045/spice/models.lib' @proc
.option scale=22.5n
.option post accurate
.param supply=@vdd

.param FO=@fanout
.include 'netlist/sandbox.inv_delay.sp'

.global vdd! vss!
vdd     vdd! gnd    dc=supply
vss     vss! gnd    dc=0
vin     in gnd      pulse( 0 supply 1ns 5ps 5ps 5ns 10ns )

.tran 1p 30n
.measure tran delay_r trig v(n1) val='supply/2' fall=2
+                     targ v(n2) val='supply/2' rise=2
.measure tran delay_f trig v(n1) val='supply/2' rise=2
+                     targ v(n2) val='supply/2' fall=2
.end
```

Example 2: Sweeping Parameters

```
#!/usr/bin/env python
"""
inv_delay.py
measuring inverter delays.
"""

from sim import mulan

M = mulan("inv_delay.ml")
M.new_sw(proc=['TT','FF','SS'], vdd=[1.0,1.1,0.9],
         fanout=[1,2,4,8])

for row in M.sw_iter():
    runbase = row.hspice('inv_delay.empy')
    row.readmeas(runbase + '.mt0')
    row.delay_avg = (row.delay_r + row.delay_f)/2.0

M.save()
```

Defining Sweeps (1)

- M.new_sw(<list of variable/value-list pairs>)
  - Defines a new set of sweep variables whose values take all possible combinations of each variable values
  - Note: existing sweep variable and data will be lost!

```
unix> python
>>> from sim import mulan
>>> M = mulan()
>>> M.new_sw(tech=['TT','SS'], vdd=1.0)
>>> M.list()
   tech        vdd
        TT     1.00000
        SS     1.00000
```

Defining Sweeps (2)

- M.multiply_sw(<list of variable/value-list pairs>)
  - has the same syntax with new_sw() except that the new sweep set *“multiplies”* with the existing set
  - All the variables must be new to the mulan instance

```
>>> M.multiply_sw(fanout=[1,2,3])
>>> M.list()
   tech        vdd        fanout
        TT     1.00000          1
        SS     1.00000          1
        TT     1.00000          2
        SS     1.00000          2
        TT     1.00000          3
        SS     1.00000          3
```

Defining Sweeps (3)

- M.add_sw(<list of variable/value-list pairs>)
  - has the same syntax with new_sw() except that the new sweep set *"adds"* to the existing set
  - All the variables must already exist in the mulan instance

```
>>> M.add_sw(tech='FF', vdd=0.9, fanout=5)
>>> M.list()
   tech          vdd          fanout
        TT      1.00000              1
        SS      1.00000              1
        TT      1.00000              2
        SS      1.00000              2
        TT      1.00000              3
        SS      1.00000              3
        FF     900.000m              5
```

Attaching Sweep Tags

- Sweep tags are basically names for the sweep cases
  - Handy when you want to select a case using a short name
  - e.g. M['TTTT'] instead of M[proc='TT', vdd=1.0, temp=25]
  - Also used as labels when plotting graphs
  - Without sweep tags, each sweep case is addressed by an integer (0, 1, 2, …)
- M.attach_swtag(<map_func>)
  - map_func is a function that maps a set of sweep parameter values to a tag string
  - e.g. map_func(proc='TT', vdd=1.0, temp=25) returns 'TTTT'

Attaching Sweep Tags (2)

```
M.new_sw(tech=[‘TT’,’FF’,’SS’], vdd=[1.0,1.1,0.9],
         temp=[25,-40,110])

def map_func(tech, vdd, temp):
   map_vdd = {1.0:’T’, 1.1:’F’, 0.9:’S’}
   map_temp = {25:’T’, -40:’F’, 110:’S’}
   return tech + map_vdd[vdd] + map_tech[temp]

M.attach_swtag(map_func)

row_TTTT = M[‘TTTT’]
row_FFFF = M[‘FFFF’]
row_SSSS = M[‘SSSS’]
```

Data Listing Methods

- For debugging or display purposes, you can use the following commands to view the contents of a mulan instance
- M.list_vars(vartype=['sw','ag','ls'], out=sys.stdout)
  - Lists the variable names and their types to a file stream
- M.list_data(<variable_list>, out=sys.stdout)
  - Lists the data contents of a mulan instance
  - A shorthand is M.list()
- M.plot(<variable_list>, filename='plot_png')
  - Plots graph – yet to be implemented

More Data Access Methods

- M.get_vars(var_type=[‘sw’, ‘ag’, ‘ls’]
  - Returns a list of variables of specified types
  - Variable types can be sweep (‘sw’), aggregate (‘ag’), or list (‘ls’) – aggregate variables are scalars and list are vectors
- M.get_data(<variable names>, var_type=[‘sw’,’ag’,’ls’])
  - Returns the data array of the named variables or types
- M.get_varlen()
  - Returns the number of variables
- M.get_swlen() or M.get_datalen()
  - Returns the number of sweep cases

Sweep Iterator: M.sw_iter()

- In the previous example, we had 12 sweep cases
- We want to run simulation for each of the cases and store/process the results
- M.sw_iter() function returns one row object at a time which corresponds to a single sweep case

```
for row in M.sw_iter():
    runbase = row.hspice('inv_delay.hsp')
    row.readmeas(runbase + '.mt0')
    row.delay_avg = (row.delay_r + row.delay_f)/2.0
    row.list()
```

Launching Simulations with Simba

- row.hspice(empy_deck, config, prefix, params, options)
  - Launches HSPICE simulation after instantiating an empy deck with the parameter values of the sweep case
  - Input arguments:
    - empy_deck : name of the simulation deck with embedded python (called 'empy deck')
    - config : configuration object that defines global parameters (more on this later)
    - prefix : prefix of the instantiated deck name (called 'run deck')
    - params : additional parameter definitions
    - options : simba options
- row.spectre() has the same syntax except it uses spectre

Empy Deck vs. Run Deck

- Empy deck refers to a template deck that contains empy
  - e.g. inv_delay.empy in our second example
- Simba instantiates a run deck by evaluating the empy commands with the sweep parameter values
  - Default location of the run deck for HSPICE is: <basename of empy deck>/<prefix>_<sw_tag>.sp
  - e.g. inv_delay/inv_delay_0.sp
- row.hspice() and row.spectre() functions return the basename (that is, without the extension) of the run deck
  - You can use it to locate the simulation result files (e.g. '.mt0')

Reading Simulation Results

- For example, HSPICE gives various result files depending on the performed analysis
  - '.tr#' for transient analysis
  - '.sw#' for DC sweep analysis
  - '.ac#' for AC analysis
  - '.mt#', '.ms#', '.ma#' for measurement results
- row.readmeas(<filename>, format) reads the file into the mulan data structure
  - Most of the time, it figures out the file format based on its file extension
  - You can force it by giving the 'format' argument value

Accessing and Processing Row Data

- row.<variable name> returns the variable value
- row.<variable name> = <value> assigns the value
- The value can be either a scalar (aggregate type) or a vector (list type)
- Example of aggregate types:

```
runbase = row.hspice('inv_delay.hsp')
row.readmeas(runbase + '.mt0')
row.delay_avg = (row.delay_r + row.delay_f)/2.0
row.max_delay = min(row.delay_r, row.delay_f)
```

Accessing and Processing Row Data (2)

- Example of list types:

```
>>> runbase = row.hspice('inv_delay.hsp')
>>> row.readmeas(runbase + '.tr0')
>>> row.get_vars()
[u'TIME', u'_0', u'in', u'n1', u'n2', u'n3', u'out',
u'vdd_bang_', u'vss_bang_', u'I_colon_vdd',
u'I_colon_vss', u'I_colon_vin']
>>> row.n2
array([1.53376050e-08, 1.53375854e-08, 1.53375677e-08,
       1.53375641e-08, 1.53376458e-08, 1.53376671e-08,
       1.53376156e-08, 1.53376050e-08, 1.53375552e-08,
       ...])
```

- The lists are returned as numpy arrays (similar to the matrices/vectors in Matlab) and can be processed with rich set of methods defined in numpy/scipy (see docs)

Accessing and Processing Row Data (3)

- Some list-type variables are ‘waveforms’ with non-uniformly sampled independent variables
  - For example, waveforms from SPICE transient analysis are non-uniformly sampled time-domain ones
- Use row[‘var_x’:’var_y’] to access the waveform object
  - Some methods such as mean() take account for the fact that the waveform is non-uniformly sampled

```
>>> runbase = row.hspice('inv_delay.hsp')
>>> row.readmeas(runbase + '.tr0')
>>> row['TIME':'n1'].mean()
0.89601049719688464
>>> row.n1.mean()
0.92673369188291466
```

Waveform Methods

- Refer to embedded documentation for full details
  - **wf.find_when**(value, count, td)
  - **wf.find_cross**(value, direction, count, td)
  - **wf.meas_trig**(value, rise/fall/cross, td)
  - **wf.meas_between**(value1, rise1/fall1/cross1, td1, other, value2, rise2, fall2, cross2, td2)
  - **wf.derive_diff**() : returns the derivative waveform
  - **wf.derive_integ**() : returns the integrated waveform
  - **wf.eval_at**(t0) : returns the value at t0
  - **wf.eval_diff**(t0) : returns the slope value at t0

Plotting Waveforms

- Use wf.plot() method – for full information, see documentations for pylab.plot() function

```
import pylab

pylab.figure()
for row in M.sw_iter():
    runbase = row.hspice('inv_delay.hsp')
    row.readmeas(runbase + '.tr0')
    row['TIME':'n1'].plot()
```

About Result Caching

- To minimize redundant work, Mulan/Simba tries to reuse the previous results as much as possible
  - Running simulations
  - Reading simulation results
- It is based on timestamps of the dependent files – if nothing has changed, then reuse
- Caution: the decision process whether to reuse is not very smart and can be over-greedy; to force re-run:
  - Delete the previous results (simulation directory and .ml file)
  - Or, use M.sw_iter(caching=False)

Example: Empy Simulation Deck

```
* inv_delay.hsp
.lib '/cad/tech/magna018/lib/hspice/HL18G-S3.5S.lib' @proc
.option scale=0.09u
.option post accurate
.param supply=@vdd

.include 'netlist/sandbox.inv_delay.sp'
.param FO=4

.global vdd! vss!
Vdd     vdd! gnd    dc=supply
Vss     vss! gnd    dc=0
Vin     in gnd      pulse( 0 supply 1ns 5ps 5ps 5ns 10ns )

.tran 1p 30n sweep FO 1 8 1
.measure tran delay_r trig v(n1) val='supply/2' fall=2
+                     targ v(n2) val='supply/2' rise=2
.measure tran delay_f trig v(n1) val='supply/2' rise=2
+                     targ v(n2) val='supply/2' fall=2
.end
```

Example: Python Script (1)

```
#!/usr/bin/env python

from sim import mulan
import pylab

M = mulan("inv_delay.ml")

# defining sweeps and sweep tags
M.new_sw(proc=['tt_tn','ff_tn','ss_tn'],
         vdd=[1.8, 1.9, 1.7])

def map_func(proc, vdd):
    map_vdd = {1.8:'T', 1.9:'H', 1.7:'L'}
    return proc[:2].upper() + map_vdd[vdd]

M.attach_swtag(map_func)
```

Example: Python Script (2)

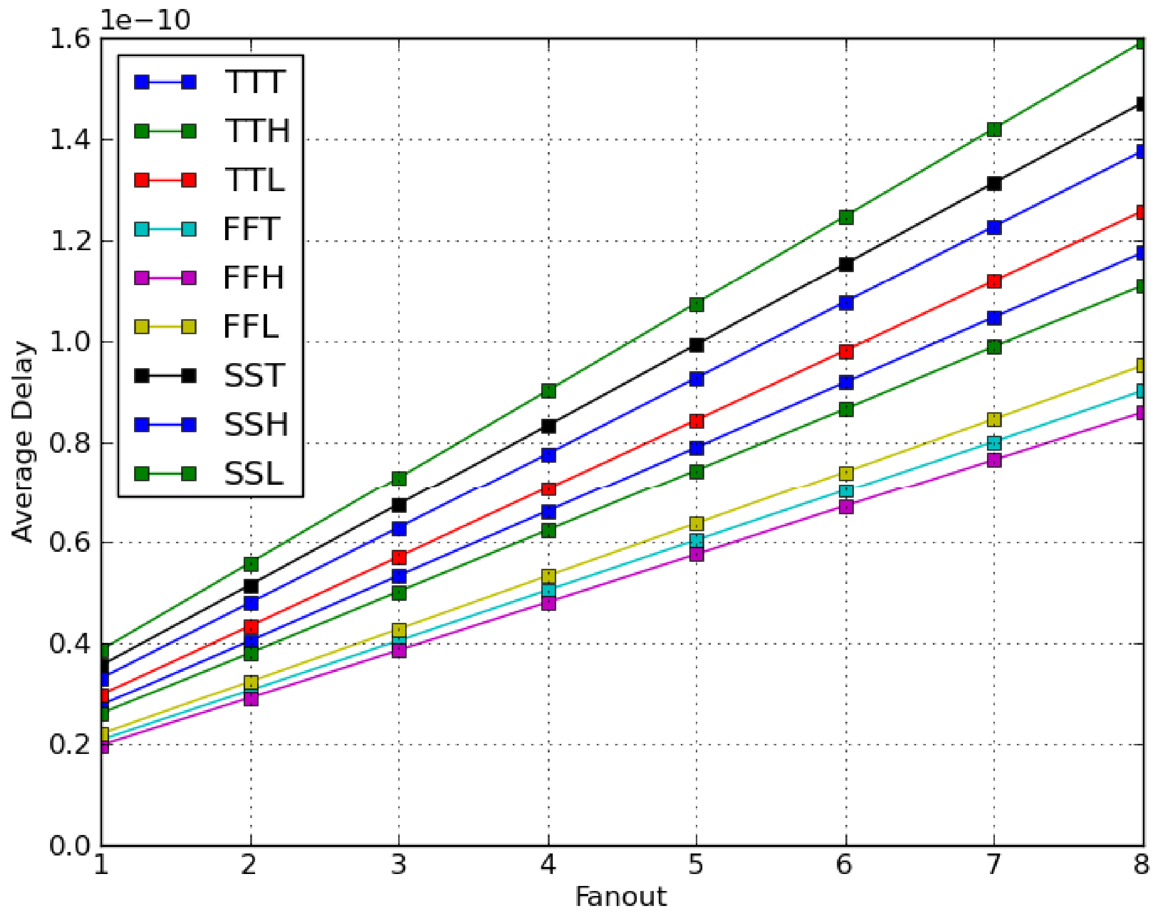
```
(... continued ...)
# run simulations and plot results
pylab.figure()

for row in M.sw_iter():
    runbase = row.hspice('inv_delay.empy')
    row.readmeas(runbase + '.mt0')
    row.delay_avg = (row.delay_r + row.delay_f)/2.0
    row['fo':'delay_avg'].plot('s-')

M.save()

pylab.grid(True)
pylab.xlabel('Fanout')
pylab.ylabel('Average Delay')
pylab.legend(loc='upper left')
pylab.savefig('inv_delay.png')
pylab.show()
```

Plot Result


1e−10
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
Average Delay
TTT
TTH
TTL
FFT
FFH
FFL
SST
SSH
SSL
1
2
3
4
5
6
7
8
Fanout

Example 3: Process Independence

- We want to use the mulan scripts and empy decks for as many IC technologies as possible *without change*
  - That way, it is much easier to maintain the script
  - Imagine you have 10 separate scripts for measuring inverter delays for 10 technologies and you just found a bug in them!
- One way to reuse a single script/deck for multiple technologies is to separate the process information
  - e.g. name of the SPICE model
  - e.g. nominal Vdd
  - e.g. lambda, etc.

Using Configuration File

- A configuration file defines global parameters for a given process technology
  - These parameters can be used within the empy deck
- Configuration file example:

```
vdd_nom = 1.0
lambda = “32.5n”
def_model = “.lib ‘/cad/tech/ptm065/models.lib‘”
```

- Within the empy deck:

```
.option scale=@lambda
.param supply=@(vdd_nom*ratio)
@def_model @proc
```

Using Configuration File (2)

- To load the configuration from a file, use “config” class

```
from sim import mulan, config
cfg = config('cfg_ptm065.py')

M = mulan()
for row in M.sw_iter():
   row.hspice('mydeck.empy', config=cfg)
   ...
```

- Caching will consider the changes in the configuration file – that is, if your config file gets updated, all the simulations will be re-run
- You may load multiple files into a single config instance