LogicMin: Logic Minimization in Python

Minimize logic functions.

Description

LogicMin is a Python package that minimize boolean functions using the tabular method of minimization (Quine-McCluskey). An object represent a truth table to which rows are added. After all rows are added, call a solve function. The solve function returns the minimized Sum of Products. The sum of products can be printed or analyzed.

For more information, look into references [1][2].

Full-adder

```
# Full-adder
import logicmin
# truth table 3 inputs, 2 outputs
t = logicmin.TT(3,2);
# add rows to the truth table (input, ouutput)
# ci a b  |  s co
t.add("000","00")
t.add("001","10")
t.add("010","10")
t.add("011","01")
t.add("100","10")
t.add("101","01")
t.add("110","01")
t.add("111","11")
# minimize functions and get
```

[1] Edward J. McCluskey. 1986. Logic Design Principles with Emphasis on Testable Semicustom Circuits. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[2] John F. Wakerly. 1989. Digital Design Principles and Practices. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

```
# solution for analysis and print
sols = t.solve()
# print solution mapped to var names (xnames=inputs, ynames=outputs)
# add debug information
print(sols.printN(xnames=['Ci','a','b'],ynames=['s','Co'], info=True))
```

Output:

```
Co <= a.b + Ci.b + Ci.a
s <= Ci'.a'.b + Ci'.a.b' + Ci.a'.b' + Ci.a.b
```

Get expression in VHDL syntax:

```
print(sols.printN(xnames=['Ci','a','b',ynames=['s','Co'], syntax='VHDL'))
```

```
Co <= a and b or Ci and b or Ci and a
s <=  not(Ci) and  not(a) and b or  not(Ci) and a and  not(b) or Ci and  not(a) and  not(b)
```

BCD to 7 segment converter

```
# BCD-8421 to 7 segment
import logicmin
t = logicmin.TT(4,7);
# b3 b2 b1 b0  | a b c d e f g
t.add("0000","1111110")
t.add("0001","0110000")
t.add("0010","1101101")
t.add("0011","1111001")
t.add("0100","0110011")
t.add("0101","1011011")
t.add("0110","0011111")
t.add("0111","1110000")
t.add("1000","1111111")
t.add("1001","1110011")
t.add("1010","-------")
t.add("1011","-------")
t.add("1100","-------")
t.add("1101","-------")
t.add("1110","-------")
t.add("1111","-------")
# Outputs minimized independently
sols = t.solve()
print(sols.printN( xnames=['b3','b2','b1','b0'], ynames=['a','b','c','d','e','f','g']))
```

```
g <= b2'.b1 + b2.b1' + b2.b0' + b3
f <= b1'.b0' + b2.b1' + b2.b0' + b3
e <= b2'.b0' + b1.b0'
```

```
d <= b2.b1'.b0 + b2'.b0' + b2'.b1 + b1.b0'
c <= b1' + b0 + b2
b <= b1'.b0' + b1.b0 + b2'
a <= b2'.b0' + b1.b0 + b2.b0 + b3
```

Finite-state machines

For finite-state machines, use the FSM object.

Binary counter with hold

```
# Finite-state machine
# x=0 => hold
# x=1 => binary up count
# y = 1 in states: e1 and e3
import logicmin
# state labels
states = ['e0','e1','e2','e3']
# 2 bits for state codes
# 1 input variable
# 1 output variable
m = logicmin.FSM(states,2,1,1)
# transition table
m.add('0','e0','e0','0')
m.add('1','e0','e1','0')
m.add('0','e1','e1','1')
m.add('1','e1','e2','1')
m.add('0','e2','e2','0')
m.add('1','e2','e3','0')
m.add('0','e3','e3','1')
m.add('1','e3','e0','1')
# asign code to states
codes = {'e0':0,'e1':1,'e2':2,'e3':3}
m.assignCodes(codes)
# solve with D flip-flops
sols = m.solveD()
# print solution with input and output names
print(sols.printN(xnames=['X','Q1','Q0'], ynames=['D1','D0','Y']))
```

Output:

The advantages of FSM objects are

1. Names for the states
2. Decouple code assignment from table initialization.

Other examples

Look into examples directory.

Install

```
pip install logicmin
```