华数杯

所有问题的代码

In [1]:
```
import pylatex
import latexify
import numpy as np
import matplotlib.pyplot as plt

from numba import jit
from numpy import argsort
import pandas as pd
import plotly.graph_objects as go
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = 'all'
InteractiveShell.ast_node_interactivity = 'last'
from scipy.optimize import minimize, rosen, rosen_der

import warnings
warnings.filterwarnings('ignore')

from sko import GA, PSO
from colorama import Fore
from time import time
```

In [2]:
```
data = pd.read_csv('环形振荡器输出频率计算表.csv', encoding='gbk', index_col=0)
data
```

Out[2]:

	反相器个数/个	PMOS宽/m	NMOS宽/m	MOS长/m
0	11	4.000000e-07	2.000000e-07	1.000000e-07
1	11	8.000000e-07	4.000000e-07	2.000000e-07
2	11	1.600000e-06	8.000000e-07	4.000000e-07
3	31	2.000000e-07	4.000000e-07	1.000000e-07
4	31	4.000000e-07	8.000000e-07	2.000000e-07
5	31	8.000000e-07	1.600000e-06	4.000000e-07
6	51	5.000000e-07	5.000000e-07	1.000000e-07
7	51	1.000000e-06	1.000000e-06	2.000000e-07
8	51	1.800000e-06	1.800000e-06	3.000000e-07
9	99	2.000000e-06	1.000000e-06	5.000000e-07

In [ ]:

In [3]:

```
VDD = 1.2  # V
Vtp = 0.398  # V
Vtn = 0.42  # V

Vds = VDD  # V
Vdsatn = Vds - Vtn  # V
Vdsatp = Vds - Vtp  # V

K_n = 111.6634e-6  # A/V^2
K_p = 68.7134e-6  # A/V^2

gate_len_min = 60e-9  # m
gate_wide_min = 120e-9  # m
gate_len_max = 100000e-9  # m
gate_wide_max = 100000e-9  # m

beta_n = lambda W, L, K_n=K_n: W / L * K_n  # A/V  # 223.2
beta_p = lambda W, L, K_p=K_p: W / L * K_p  # A/V  # 274.8

CL_ratio = 3.137  # F/m^2
gate_covered_area = lambda W, L: W * L  # m^2 栅极覆盖的沟道面积
CL = lambda W_n, W_p, L: CL_ratio * gate_covered_area(W_n + W_p, L)  #  F 负载电容(正比于下一级反相器的栅极面积)

sd_area  =lambda W: 2 * 190e-9 * W  # m^2 源漏面积 / 栅极两侧的红色矩形面积
```

```python
mos_area = lambda W, L: gate_covered_area(W, L) + sd_area(W)  # m^2 NMOS 或 PMOS 的面积
single_inverter_area = lambda W_n, W_p, L: mos_area(W_n, L) + mos_area(W_p, L) + 70e-9 * (L + 2 * 190e-9)  # m^2 单个反相器的面积
ring_oscillator = lambda N, W_p, W_n, L: N * single_inverter_area(W_n, W_p, L)  # 环形振荡器的面积

N, W_p, W_n, L = 51, 120e-9, 120e-9, 60e-9  # 测试样例 (极限)
```

问题一：计算延迟时间

t_r 和 t_f

$$t_r = \frac{2(V_{TN} - 0.1V_{DD})C_L}{\beta_p(V_{DD} - |V_{TP}|)^2} + \frac{(0.9V_{DD} - V_{TN})C_L}{k_p\left(\frac{W}{L}\right)_p\left[(V_{DD} - V_{TP})V_{DS} - \frac{1}{2}V_{DD}^2\right]}$$

$$t_f = \frac{2(V_{TP} - 0.1V_{DD})C_L}{\beta_n(V_{DD} - |V_{TN}|)^2} + \frac{(0.9V_{DD} - V_{TP})C_L}{k_n\left(\frac{W}{L}\right)_n\left[(V_{DD} - V_{TN})V_{DS} - \frac{1}{2}V_{DD}^2\right]}$$

In [4]:

```python
InteractiveShell.ast_node_interactivity = 'all'
# InteractiveShell.ast_node_interactivity = 'last'

t_r = lambda W_p, W_n, L: 2 * (Vtn - 0.1 * VDD) * CL(W_n, W_p, L) / beta_p(W_p, L) / (VDD - Vtp) ** 2 + \
                          (0.9 * VDD - Vtn) * CL(W_n, W_p, L) / (beta_p(W_p, L) * ((VDD - Vtp) * Vds - 0.5 * VDD ** 2))
t_f = lambda W_p, W_n, L: 2 * (Vtp - 0.1 * VDD) * CL(W_n, W_p, L) / beta_n(W_n, L) / (VDD - Vtn) ** 2 + \
                          (0.9 * VDD - Vtp) * CL(W_n, W_p, L) / (beta_n(W_n, L) * ((VDD - Vtn) * Vds - 0.5 * VDD ** 2))
T = lambda N, W_p, W_n, L: (t_r(W_p, W_n, L) + t_f(W_p, W_n, L)) * N / 2
# T = lambda W_n, W_p, L, N: N * VDD * CL(W_n, W_p, L) / (VDD - Vt) ** 2 * (1 / beta_n(W_n, L) + 1 / beta_p(W_p, L))
# f = lambda N, W_p, W_n, L: 1 / T(W_n, W_p, L, N)
f1 = lambda data: 1 / T(data[0], data[1], data[2], data[3])

ans1_arg = np.array(data.apply(f1, axis=1)) / 1e6
print(f"问题一的输出频率（计算方法1）(MHz): ")
ans1_arg
print(f"51个反相器最大输出频率：{f1((N, W_p, W_n, L)) / 1e6}MHz")
```

```
问题一的输出频率（计算方法1）(MHz):
51个反相器最大输出频率：19.364629202901142MHz
```

t_{pHL} 和 t_{pLH}

$$t_{pHL} = \frac{3ln2}{4}\frac{V_{DD}C_L}{I_{DSATn}} = \frac{3ln2}{4}\frac{V_{DD}\ C_L}{\left(\frac{W}{L}\right)_n k_n'\ V_{DSATn}\left(V_{DD} - V_{Tn} - \frac{V_{DSATn}}{2}\right)}$$

$$t_{pLH} = \frac{3ln2}{4}\frac{V_{DD}C_L}{I_{DSATp}} = \frac{3ln2}{4}\frac{V_{DD}\ C_L}{\left(\frac{W}{L}\right)_p k_p'\ V_{DSATp}\left(V_{DD} - V_{Tp} - \frac{V_{DSATp}}{2}\right)}$$

In [5]:
```
InteractiveShell.ast_node_interactivity = 'all'
# InteractiveShell.ast_node_interactivity = 'last'

# tpHL = lambda W_p, W_n, L: (3 * np.log(2) / 4) * (VDD * CL(W_n, W_p, L) / (beta_n(W_n, L) * Vdsat * (VDD - Vtn - Vdsat / 2)))
# tpLH = lambda W_p, W_n, L: (3 * np.log(2) / 4) * (VDD * CL(W_n, W_p, L) / (beta_p(W_p, L) * Vdsat * (VDD - Vtp - Vdsat / 2)))
tpHL = lambda W_p, W_n, L: (3 * np.log(2) / 4) * (VDD * CL(W_n, W_p, L) / (W_n / L * K_n * Vdsatn * (VDD - Vtn - Vdsatn / 2)))
tpLH = lambda W_p, W_n, L: (3 * np.log(2) / 4) * (VDD * CL(W_n, W_p, L) / (W_p / L * K_p * Vdsatp * (VDD - Vtp - Vdsatp / 2)))

tpd = lambda W_p, W_n, L: (tpHL(W_n, W_p, L) + tpLH(W_n, W_p, L)) / 2
f2 = lambda data:1 / (2 * data[0]  * tpd(data[1], data[2], data[3]))

ans2_arg = np.array(data.apply(f2, axis=1)) / 1e6
print(f"问题一的输出频率（计算方法2）(MHz): ")
ans2_arg
print(f"51个反相器最大输出频率: {f2((N, W_p, W_n, L)) / 1e6}MHz")
```

```
问题一的输出频率（计算方法2）(MHz):
```

Out[5]:
```
array([25.81995182,  6.45498795,  1.61374699, 10.55319363,  2.63829841,
        0.6595746 ,  6.70725955,  1.67681489,  0.74525106,  0.11475534])
51个反相器最大输出频率: 18.631276519057344MHz
```

In [6]:
```
# TODO 问题一结果
ans_arg = (ans1_arg + ans2_arg) / 2
print("问题一的输出频率（平均）(MHz): ")
ans_arg.round(10)
```

```
问题一的输出频率（平均）(MHz):
```

Out[6]:
```
array([28.22836536,  7.05709134,  1.76427283, 10.07521981,  2.51880495,
        0.62970124,  6.83926303,  1.70981576,  0.75991811,  0.1254594 ])
```

问题二：求解尺寸

In [7]:
```
# TODO DJH
W_p_djh, W_n_djh, L_djh = 120e-9, 120e-9, 83.5e-9  # m  # djh 数据 (验算)
print(f"PMOS宽: {round(W_p_djh * 1e9, 2)}nm\tPMOS长: {round(L_djh * 1e9, 2)}nm\
```

```
      \nNMOS宽: {round(W_n_djh * 1e9, 2)}nm\tNMOS长: {round(L_djh * 1e9, 2)}nm")
print("面积: ", ring_oscillator(51, W_p_djh, W_n_djh, L_djh))
print("频率: %f MHz" % (f2((51, W_p_djh, W_n_djh, L_djh)) / 1e6))
```

```
PMOS宽: 120.0nm PMOS长: 83.5nm
NMOS宽: 120.0nm NMOS长: 83.5nm
面积:  7.327934999999999e-12
频率: 9.619936 MHz
```

In [8]:

```
VDD = 1.2  # V
Vtp = 0.398  # V
Vtn = 0.42  # V

Vds = VDD  # V
Vdsatn = Vds - Vtn  # V
Vdsatp = Vds - Vtp  # V

K_n = 111.6634e-6  # A/V^2
K_p = 68.7134e-6  # A/V^2

gate_len_min = 60e-9  # m
gate_wide_min = 120e-9  # m
gate_len_max = 100000e-9  # m
gate_wide_max = 100000e-9  # m

beta_n = lambda W, L, K_n=K_n: W / L * K_n  # A/V  # 223.2
beta_p = lambda W, L, K_p=K_p: W / L * K_p  # A/V  # 274.8

CL_ratio = 3.137  # F/m^2
gate_covered_area = lambda W, L: W * L  # m^2 栅极覆盖的沟道面积
CL = lambda W_n, W_p, L: CL_ratio * gate_covered_area(W_n + W_p, L)  #  F 负载电容(正比于下一级反相器的栅极面积)

sd_area  =lambda W: 2 * 190e-9 * W  # m^2 源漏面积 / 栅极两侧的红色矩形面积
mos_area = lambda W, L: gate_covered_area(W, L) + sd_area(W)  # m^2 NMOS 或 PMOS 的面积
single_inverter_area = lambda W_n, W_p, L: mos_area(W_n, L) + mos_area(W_p, L) + 70e-9 * (L + 2 * 190e-9)  # m^2 单个反相器的面积
ring_oscillator3 = lambda N, W_p, W_n, L: N * single_inverter_area(W_n, W_p, L)  # 环形振荡器的面积
```

In [9]:

```
# %%time
InteractiveShell.ast_node_interactivity = 'last'

NEED = 10e6  # 10 MHz
EPS = 5e5  # 0.5MHz
wmin, wmax = 120e-9, 100000e-9
lmin, lmax = 60e-9, 100000e-9

inverter_num = 51
```

```python
ring_oscillator2 = lambda W_p, W_n, L: inverter_num * single_inverter_area(W_n, W_p, L)  # 环形振荡器的面积

f = lambda data: (f1((data[0], data[1], data[2], data[3])) + f2((data[0], data[1], data[2], data[3]))) / 2


def check2(p, n, l):
    print(Fore.RED)
    print("验算：")
    print(f"PMOS宽：{round(p * 1e9, 2)}nm\tPMOS长：{round(l * 1e9, 2)}nm\
        \nNMOS宽：{round(n * 1e9, 2)}nm\tNMOS长：{round(l * 1e9, 2)}nm")
    print("面积：", ring_oscillator2(p, n, l))
    print("频率：%f MHz" % (f((inverter_num, p, n, l)) / 1e6))
    print(Fore.RESET)


class SOLVER2:
    def __init__(self):
        self.S_MIN = 1  # m^2
        self.WP = 0  # m
        self.WN = 0  # m
        self.L = 0  # m
        self.F = 0  # Hz
        self.ga = GA.GA(
            ring_oscillator2,
            n_dim=3,  # p, n, L
            lb=[wmin, wmin, lmin],
            ub=[wmax, wmax, lmax],
            constraint_ueq=(lambda x: abs(f((inverter_num, x[0], x[1], x[2])) - NEED) - EPS, ),
        )
    def save_best(self, s, p, n, l):
        if s < self.S_MIN:
            self.S_MIN = s
            self.WP = p
            self.WN = n
            self.L = l
            self.F = f((inverter_num, p, n, l))
    def print_best(self):
        print(Fore.RED)
        print(f"PMOS宽：{round(self.WP * 1e9, 2)}nm\tPMOS长：{round(self.L * 1e9, 2)}nm\
            \nNMOS宽：{round(self.WN * 1e9, 2)}nm\tNMOS长：{round(self.L * 1e9, 2)}nm")
        print("面积：", self.S_MIN)
        print("频率：%f MHz" % (self.F / 1e6))
        print(Fore.RESET)
    @jit
    def run_ga(self):
        print("遗传算法：")
```

```
        epochs = 10
        printt = epochs // 10

        t0 = time()
        xbests, ybests = [], []
        for epoch in range(epochs):
            xbest, ybest = self.ga.run()
            xbests.append(xbest)
            ybests.append(float(ybest))
            if epoch % printt == 0:
                print("epoch:", epoch)
        print("遗传算法用时：", time() - t0, 's')

        xbests, ybests = np.array(xbests), np.array(ybests)
        idx = np.argsort(ybests)
        min_idx = idx[0]
        xbest, ybest = xbests[min_idx, :], ybests[min_idx]  # p, n, l, 面积

        p, n, l = xbest
        self.save_best(ybest, p, n, l)

        return p, n, l

    @jit
    def run_travel(self, p, n, l, step=100):
        print('变步长搜索：')

        low1 = max(wmin, 0.5 * p)
        high1 = min(wmax, 2 * p)
        low2 = max(wmin, 0.5 * n)
        high2 = min(wmax, 2 * n)
        low3 = max(lmin, 0.5 * l)
        high3 = min(lmax, 2 * l)

        pp = np.linspace(low1, high1, step)
        nn = np.linspace(low2, high2, step)
        ll = np.linspace(low3, high3, step)

        min_area = 1
        ii, jj, kk = None, None, None

        t0 = time()
        epochs = len(pp)
        printt = epochs // 10
        for epoch, i in enumerate(pp):
```

```python
            for j in nn:
                for k in ll:
                    f_tmp = f((inverter_num, i, j, k))
                    if abs(f_tmp - NEED) < EPS:
                        area = ring_oscillator2(i, j, k)
                        if area < min_area:
                            min_area = area
                            ii, jj, kk = i, j, k
            if epoch % printt == 0:
                print("epoch:", epoch, "/", epochs, "time:", time() - t0)
                t0 = time()
        print("变步长搜索用时: ", time() - t0, 's')

        self.save_best(min_area, ii, jj, kk)

        return ii, jj, kk

    def run_plan(self, p, n, l):
        print('规划: ')

        low1 = max(wmin, 0.5 * p)
        high1 = min(wmax, 2 * p)
        low2 = max(wmin, 0.5 * n)
        high2 = min(wmax, 2 * n)
        low3 = max(lmin, 0.5 * l)
        high3 = min(lmax, 2 * l)

        b0 = (low1, high1)
        b1 = (low2, high2)
        b2 = (low3, high3)
        bnds = (b0, b1, b2)

        minimize_fun = lambda x: ring_oscillator2(x[0], x[1], x[2])

        constraint1 = lambda x: EPS - abs(f((inverter_num, x[0], x[1], x[2])) - NEED)
        constraint2 = lambda x: 1

        # >= 0
        con1 = {'type':  'ineq',  'fun': constraint1}
        con2 = {'type':  'ineq',  'fun': constraint2}
        con3 = {'type':  'ineq',  'fun': lambda x: x[0] - low1}
        con6 = {'type':  'ineq',  'fun': lambda x: high1 - x[0]}
        con4 = {'type':  'ineq',  'fun': lambda x: x[1] - low1}
        con7 = {'type':  'ineq',  'fun': lambda x: high2 - x[1]}
        con5 = {'type':  'ineq',  'fun': lambda x: x[2] - low3}
        con8 = {'type':  'ineq',  'fun': lambda x: high3 - x[2]}
```

```python
        cons = ([con3, con6, con4, con7, con5, con8, con1, con2])
        # cons = ([con1, con2])

        t0 = time()
        opt = minimize(
            minimize_fun,
            x0=[p, n, l],
            bounds=bnds,
            constraints=cons,
            method='trust-constr',  #  SLSQP  trust-constr
#             options={'gtol': 1e-9, 'disp': True},
            options={'xtol': 1e-30,  'gtol': 1e-30, 'disp': True}
        )
        xopt = opt.x
        s = minimize_fun(xopt)
        p, n, l = xopt

        print("规划用时: ", time() - t0, 's')

        if wmin < p < wmax and wmin < n < wmax and lmin < l < lmax and s > 0:
            self.save_best(s, p, n, l)

        return p, n, l
```

遗传 + 变步长

In [10]:
```python
solver2 = SOLVER2()
```

In [11]:
```python
ptmp, ntmp, ltmp = solver2.run_ga()
solver2.print_best()
```

```
遗传算法：
epoch: 0
epoch: 1
epoch: 2
epoch: 3
epoch: 4
epoch: 5
epoch: 6
epoch: 7
epoch: 8
epoch: 9
遗传算法用时： 4.977769613265991 s

PMOS宽：705.81nm        PMOS长：60.0nm
NMOS宽：3732.47nm       NMOS长：60.0nm
面积： 1.0116579354838711e-10
频率：10.418814 MHz
```

In [12]:

```
ptmp, ntmp, ltmp = solver2.run_travel(ptmp, ntmp, ltmp)
solver2.print_best()
```

```
变步长搜索：
epoch: 0 / 100 time: 0.9787731170654297
epoch: 10 / 100 time: 1.7014474868774414
epoch: 20 / 100 time: 2.3546242713928223
epoch: 30 / 100 time: 1.8747622966766357
epoch: 40 / 100 time: 2.501204490661621
epoch: 50 / 100 time: 2.766569137573242
epoch: 60 / 100 time: 2.5535428524017334
epoch: 70 / 100 time: 2.596317768096924
epoch: 80 / 100 time: 2.9749605655670166
epoch: 90 / 100 time: 4.448933124542236
变步长搜索用时： 4.145752191543579 s

PMOS宽：352.9nm PMOS长：60.0nm
NMOS宽：1866.24nm       NMOS长：60.0nm
面积： 5.136829677419356e-11
频率：10.418814 MHz
```

In [13]:

```
check2(solver2.WP, solver2.WN, solver2.L)
```

```
验算：
PMOS宽： 352.9nm PMOS长： 60.0nm
NMOS宽： 1866.24nm       NMOS长： 60.0nm
面积：  5.136829677419356e-11
频率： 10.418814 MHz
```

绘制遗传迭代收敛图

In [14]:

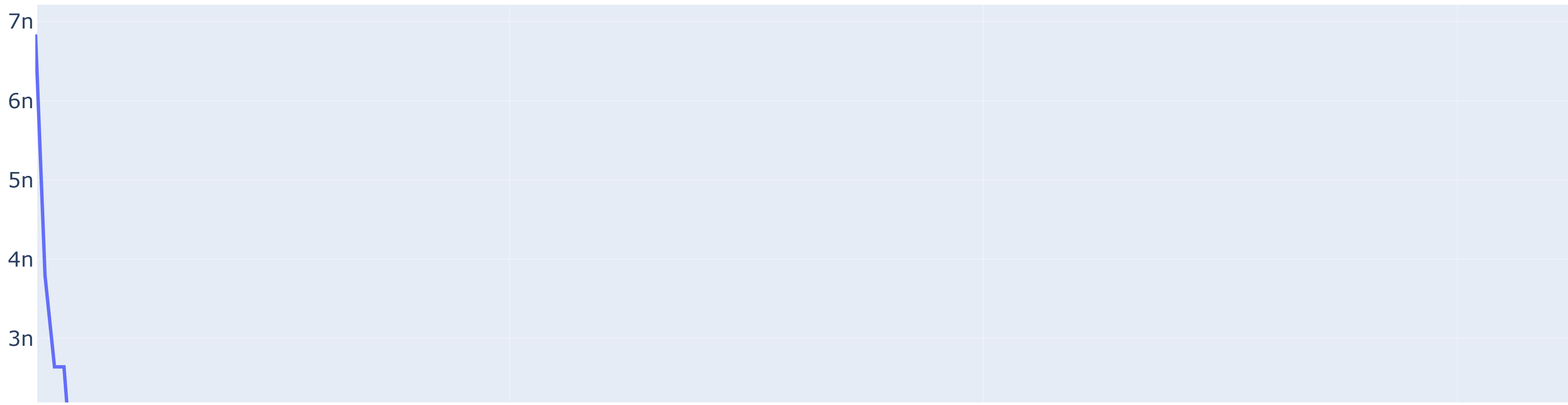
```python
# TODO
import matplotlib.pyplot as plt
import plotly.graph_objects as go

ys = []
low, high = 1, 501
t0 = time()
X = [i for i in range(low, high)]
for x in X:
    ga = GA.GA(
        ring_oscillator2,
        n_dim=3,  # p, n, L
        lb=[wmin, wmin, lmin],
        ub=[wmax, wmax, lmax],
        constraint_ueq=(lambda x: abs(f((inverter_num, x[0], x[1], x[2])) - NEED) - EPS, ),
        max_iter=x,
    )
    if x % (high // 200) == 0:
        _, y = ga.run()
        if not len(ys) or y < ys[-1]:
            ys.append(float(y))
        else:
            ys.append(ys[-1])
    if x % 50 == 0:
        print("epoch:", x, "time:", time() - t0)
        t0 = time()
```

```
epoch: 50 time: 3.3996694087982178
epoch: 100 time: 7.770452976226807
epoch: 150 time: 12.53598690032959
epoch: 200 time: 18.224925756454468
epoch: 250 time: 21.13223433494568
epoch: 300 time: 26.561934232711792
epoch: 350 time: 29.69418430328369
epoch: 400 time: 35.393126487731934
epoch: 450 time: 38.91854166984558
epoch: 500 time: 44.48362970352173
```

In [15]:

```
trace = go.Scatter(x=[i for i in range(len(ys))], y=ys)
fig = go.Figure(trace)
fig.update_layout(xaxis=dict(title='$迭代次数$'), yaxis=dict(title='$环形振荡器面积$'))
fig.write_image('./遗传迭代收敛图.svg')
fig.show()
```



遗传 + 规划

In [16]:
```
solver2 = SOLVER2()
```

In [17]:
```
ptmp, ntmp, ltmp = solver2.run_ga()
solver2.print_best()
```

```
遗传算法：
epoch: 0
epoch: 1
epoch: 2
epoch: 3
epoch: 4
epoch: 5
epoch: 6
epoch: 7
epoch: 8
epoch: 9
遗传算法用时： 5.56471848487854 s

PMOS宽：998.71nm        PMOS长：60.0nm
NMOS宽：5392.26nm       NMOS长：60.0nm
面积： 1.4498411612903224e-10
频率：10.275405 MHz
```

In [18]:

```
ptmp, ntmp, ltmp = solver2.run_plan(ptmp, ntmp, ltmp)
solver2.print_best()
```

```
规划：
`xtol` termination condition is satisfied.
Number of iterations: 726, function evaluations: 1396, CG iterations: 564, optimality: 6.44e-06, constraint violation: 0.00e+00, execution time:  1.2 s.
规划用时： 1.1769442558288574 s

PMOS宽：924.54nm        PMOS长：60.12nm
NMOS宽：5292.25nm       NMOS长：60.12nm
面积： 1.4111335186810904e-10
频率：9.837604 MHz
```

In [19]:

```
# 问题二结果
check2(solver2.WP, solver2.WN, solver2.L)
```

```
验算：
PMOS宽：924.54nm        PMOS长：60.12nm
NMOS宽：5292.25nm       NMOS长：60.12nm
面积： 1.4111335186810904e-10
频率：9.837604 MHz
```

问题三：求解最小功耗

In [20]:
```
VDD = 1.2  # V
Vtp = 0.398  # V
Vtn = 0.42  # V

Vds = VDD  # V
Vdsatn = Vds - Vtn  # V
Vdsatp = Vds - Vtp  # V

K_n = 111.6634e-6  # A/V^2
K_p = 68.7134e-6  # A/V^2

gate_len_min = 60e-9  # m
gate_wide_min = 120e-9  # m
gate_len_max = 100000e-9  # m
gate_wide_max = 100000e-9  # m

beta_n = lambda W, L, K_n=K_n: W / L * K_n  # A/V  # 223.2
beta_p = lambda W, L, K_p=K_p: W / L * K_p  # A/V  # 274.8

CL_ratio = 3.137  # F/m^2
gate_covered_area = lambda W, L: W * L  # m^2 栅极覆盖的沟道面积
CL = lambda W_n, W_p, L: CL_ratio * gate_covered_area(W_n + W_p, L)  #  F 负载电容(正比于下一级反相器的栅极面积)

sd_area  =lambda W: 2 * 190e-9 * W  # m^2 源漏面积 / 栅极两侧的红色矩形面积
mos_area = lambda W, L: gate_covered_area(W, L) + sd_area(W)  # m^2 NMOS 或 PMOS 的面积
single_inverter_area = lambda W_n, W_p, L: mos_area(W_n, L) + mos_area(W_p, L) + 70e-9 * (L + 2 * 190e-9)  # m^2 单个反相器的面积
ring_oscillator3 = lambda n, W_p, W_n, L: n * single_inverter_area(W_n, W_p, L)  # 环形振荡器的面积
```

In [ ]:

In [21]:
```
Id_max = lambda KW_L, Vgs, Vt: 0.5 * KW_L * (Vgs - Vt) ** 2
Id_n = lambda W_n, L: Id_max(K_n * W_n / L, VDD / 2, Vtn)
Id_p = lambda W_p, L: Id_max(K_p * W_p / L, VDD / 2, Vtp)
IMAX = lambda W_p, W_n, L: Id_n(W_n, L) + Id_p(W_p, L)

freq3 = 5e6
freq4 = 2e3
f_fan = lambda N: 2 * N * freq3
PT = lambda N, W_p, W_n, L: CL(W_n, W_p, L) * f_fan(N) * VDD ** 2
PA = lambda N, W_p, W_n, L: VDD * IMAX(W_p, W_n, L)
single_P = lambda N, W_p, W_n, L: PT(N, W_p, W_n, L) + PA(N, W_p, W_n, L)  # 1 个反相器
total_P = lambda N, W_p, W_n, L: 2 * N * single_P(N, W_p, W_n, L)  # 2 * N 个反相器
```

In [ ]:

In [22]:
```python
# %%time
InteractiveShell.ast_node_interactivity = 'last'

question = 3

if question == 3:
    EPS = 5e5  # 0.5MHz
    NEED = 5e6  # 5 MHz
else:
    EPS = 100  # 100Hz
    NEED = 2e3  # 2KHz

wmin, wmax = 120e-9, 100000e-9
lmin, lmax = 60e-9, 100000e-9

ring_oscillator3 = lambda n, W_p, W_n, L: n * single_inverter_area(W_n, W_p, L)  # 环形振荡器的面积

f = lambda data: (f1((data[0], data[1], data[2], data[3])) + f2((data[0], data[1], data[2], data[3]))) / 2


def check3(geshu, p, n, l):
    print(Fore.RED)
    print("验算：")
    print(f"反相器个数：", geshu)
    print(f"PMOS宽：{round(p * 1e9, 2)}nm\tPMOS长：{round(l * 1e9, 2)}nm\
          \nNMOS宽：{round(n * 1e9, 2)}nm\tNMOS长：{round(l * 1e9, 2)}nm")
    print("面积：", ring_oscillator2(p, n, l))
    print("频率：%f MHz" % (f((geshu, p, n, l)) / 1e6))
    print("功耗：", total_P(geshu, p, n, l))
    print(Fore.RESET)


class SOLVER3:
    def __init__(self):
        self.S_MIN = 1  # m^2
        self.P_MIN = 1  # W
        self.geshu = 0
        self.WP = 0  # m
        self.WN = 0  # m
        self.L = 0  # m
        self.F = 0  # Hz
        self.ga = GA.GA(
            total_P,
            n_dim=4,  # N, p, n, L
            lb=[3, wmin, wmin, lmin],
            ub=[3.01, wmax, wmax, lmax],
```

```python
            constraint_ueq=(lambda a: abs(f((a[0], a[1], a[2], a[3])) - NEED) - EPS, ),  # 5 MHz
            precision=[1, 1e-7, 1e-7, 1e-7],
        )
    def save_best(self, P, geshu, p, n, l):
        if P < self.P_MIN:
            self.P_MIN = P
            self.S_MIN = ring_oscillator3(geshu, p, n, l)
            self.geshu = geshu
            self.WP = p
            self.WN = n
            self.L = l
            self.F = f((geshu, p, n, l))
    def print_best(self):
        print(Fore.RED)
        print(f"反相器个数：", self.geshu)
        print(f"PMOS宽：{round(self.WP * 1e9, 2)}nm\tPMOS长：{round(self.L * 1e9, 2)}nm\
              \nNMOS宽：{round(self.WN * 1e9, 2)}nm\tNMOS长：{round(self.L * 1e9, 2)}nm")
        print("面积：", self.S_MIN)
        print("频率：%f MHz" % (self.F / 1e6))
        print("功耗：", self.P_MIN)
        print(Fore.RESET)
    @jit
    def run_ga(self):
        print("遗传算法：")

        epochs = 10
        printt = epochs // 10

        t0 = time()
        xbests, ybests = [], []
        for epoch in range(epochs):
            xbest, ybest = self.ga.run()
            xbests.append(xbest)
            ybests.append(float(ybest))
            if epoch % printt == 0:
                print("epoch:", epoch)
        print("遗传算法用时：", time() - t0, 's')

        xbests, ybests = np.array(xbests), np.array(ybests)
        idx = np.argsort(ybests)
        min_idx = idx[0]
        xbest, ybest = xbests[min_idx, :], ybests[min_idx]  # p, n, l, 面积

        geshu, p, n, l = xbest
        self.save_best(ybest, geshu, p, n, l)
```

```python
        return geshu, p, n, l
    @jit
    def run_travel(self, g, p, n, l, step=50):
        print('变步长搜索：')

        low0 = max(3, 0.5 * g)
        high0 = min(100, 2 * g)
        low1 = max(wmin, 0.5 * p)
        high1 = min(wmax, 2 * p)
        low2 = max(wmin, 0.5 * n)
        high2 = min(wmax, 2 * n)
        low3 = max(lmin, 0.5 * l)
        high3 = min(lmax, 2 * l)

        gg = np.arange(low0, high0)
        pp = np.linspace(low1, high1, step)
        nn = np.linspace(low2, high2, step)
        ll = np.linspace(low3, high3, step)

        min_power = 1
        hh, ii, jj, kk = None, None, None, None

        t0 = time()
        t00 = time()
        epochs = len(pp)
        printt = epochs // 10
        for epoch, i in enumerate(pp):
            for j in nn:
                for k in ll:
                    for h in gg:
                        f_tmp = f((h, i, j, k))
                        if abs(f_tmp - NEED) < EPS:
                            area = ring_oscillator3(h, i, j, k)
                            power = total_P(h, i, j, k)
                            if power < min_power:
                                min_power = power
                                hh, ii, jj, kk = h, i, j, k
            if epoch % printt == 0:
                print("epoch:", epoch, "/", epochs, "time:", time() - t00)
                t00 = time()
        print("变步长搜索用时：", time() - t0, 's')

        self.save_best(min_power, hh, ii, jj, kk)

        return hh, ii, jj, kk
```

```python
    def run_plan(self, g, p, n, l):
        print('规划: ')

        low0 = max(3, 0.5 * g)
        high0 = min(100, 2 * g)
        low1 = max(wmin, 0.5 * p)
        high1 = min(wmax, 2 * p)
        low2 = max(wmin, 0.5 * n)
        high2 = min(wmax, 2 * n)
        low3 = max(lmin, 0.5 * l)
        high3 = min(lmax, 2 * l)

        b_ = (low0, high0)
        b0 = (low1, high1)
        b1 = (low2, high2)
        b2 = (low3, high3)
        bnds = (b_, b0, b1, b2)

        minimize_fun = lambda x: total_P(x[0], x[1], x[2], x[3])

        constraint1 = lambda x: EPS - abs(f((x[0], x[1], x[2], x[3])) - NEED)
        constraint2 = lambda x: 1

        # >= 0
        con1 = {'type':  'ineq',  'fun': constraint1}
        con2 = {'type':  'ineq',  'fun': constraint2}
        con3 = {'type':  'ineq',  'fun': lambda x: x[0] - low0}
        con6 = {'type':  'ineq',  'fun': lambda x: high0 - x[0]}
        con4 = {'type':  'ineq',  'fun': lambda x: x[1] - low1}
        con7 = {'type':  'ineq',  'fun': lambda x: high1 - x[1]}
        con5 = {'type':  'ineq',  'fun': lambda x: x[2] - low2}
        con8 = {'type':  'ineq',  'fun': lambda x: high2 - x[2]}
        con9 = {'type':  'ineq',  'fun': lambda x: x[3] - low3}
        con10 = {'type':  'ineq',  'fun': lambda x: high3 - x[3]}

        cons = ([con3, con6, con4, con7, con5, con8, con9, con10, con1, con2])
        # cons = ([con1, con2])

        t0 = time()
        opt = minimize(
            minimize_fun,
            x0=[g, p, n, l],
            bounds=bnds,
            constraints=cons,
            method='trust-constr',  #  SLSQP  trust-constr
#             options={'gtol': 1e-9, 'disp': True},
```

```python
            options={'xtol': 1e-30,  'gtol': 1e-30, 'disp': True}
        )
        xopt = opt.x
        power = minimize_fun(xopt)
        g, p, n, l = xopt

        print("规划用时：", time() - t0, 's')

        if 3 <= g <= 100 and wmin < p < wmax and wmin < n < wmax and lmin < l < lmax and power > 0:
            self.save_best(power, g, p, n, l)

        return g, p, n, l
```

遗传 + 变步长

In [23]:
```python
solver3 = SOLVER3()
```

In [24]:
```python
gtmp, ptmp, ntmp, ltmp = solver3.run_ga()
solver3.print_best()
```

```
遗传算法：
epoch: 0
epoch: 1
epoch: 2
epoch: 3
epoch: 4
epoch: 5
epoch: 6
epoch: 7
epoch: 8
epoch: 9
遗传算法用时：  5.766547679901123 s

反相器个数：  3.0
PMOS宽：608.17nm        PMOS长：353.08nm
NMOS宽：120.0nm NMOS长：353.08nm
面积：  1.7553699176993665e-12
频率：5.196174 MHz
功耗：  0.0002308652503192018
```

In [25]:
```python
# %%time
gtmp, ptmp, ntmp, ltmp = solver3.run_travel(gtmp, ptmp, ntmp, ltmp)
solver3.print_best()
```

```
变步长搜索：
epoch: 0 / 50 time: 1.4284493923187256
epoch: 5 / 50 time: 0.8896236419677734
epoch: 10 / 50 time: 1.063575267791748
epoch: 15 / 50 time: 1.065330982208252
epoch: 20 / 50 time: 1.0292222499847412
epoch: 25 / 50 time: 0.994377851486206
epoch: 30 / 50 time: 1.2509689331054688
epoch: 35 / 50 time: 0.9951081275939941
epoch: 40 / 50 time: 1.069199800491333
epoch: 45 / 50 time: 1.1190121173858643
变步长搜索用时： 11.780662536621094 s

反相器个数： 3.0
PMOS宽：304.09nm        PMOS长：425.14nm
NMOS宽：120.0nm NMOS长：425.14nm
面积： 1.1934195575496305e-12
频率：5.239943 MHz
功耗： 0.00015749510838499896
```

遗传 + 规划

In [26]:
```
solver3 = SOLVER3()
```

In [27]:
```
gtmp, ptmp, ntmp, ltmp = solver3.run_ga()
solver3.print_best()
```

```
遗传算法：
epoch: 0
epoch: 1
epoch: 2
epoch: 3
epoch: 4
epoch: 5
epoch: 6
epoch: 7
epoch: 8
epoch: 9
遗传算法用时：  9.476129531860352 s

反相器个数：  3.0
PMOS宽：120.0nm PMOS长：450.77nm
NMOS宽：217.63nm        NMOS长：450.77nm
面积：  1.0159540503894303e-12
频率：5.274254 MHz
功耗：  0.00013272758188433945
```

In [28]:

```
# %%time 问题三结果
gtmp, ptmp, ntmp, ltmp = solver3.run_plan(gtmp, ptmp, ntmp, ltmp)
solver3.print_best()
```

```
规划：
`xtol` termination condition is satisfied.
Number of iterations: 716, function evaluations: 1980, CG iterations: 681, optimality: 7.54e+01, constraint violation: 7.53e-09, execution time:  2.3 s.
规划用时：  2.331120729446411 s

反相器个数：  3.0
PMOS宽：120.0nm PMOS长：450.77nm
NMOS宽：217.63nm        NMOS长：450.77nm
面积：  1.0159540503894303e-12
频率：5.274254 MHz
功耗：  0.00013272758188433945
```

问题四：求解振荡器个数

先把单位换成 μm, 最后再把结果换成 m

In [29]:

```
import itertools
import gurobipy as gp
```

```python
from gurobipy import GRB
from pprint import pprint

channel_width = 80e-6  # m
wafer_length, wafer_width = 4e-3, 3e-3  # m
chip1_length, chip1_width = 1.76e-3, 1.46e-3  # m
chip2_length, chip2_width = 1.046e-3, 1.146e-3  # m
chip3_length, chip3_width = 1.096e-3, 0.846e-3  # m
chip4_length, chip4_width = 1.05e-3, 2.16e-3  # m
chip5_length, chip5_width = 1.77e-3, 0.8e-3  # m
chip6_length, chip6_width = 1.5e-3, 0.5e-3  # m

channel_width = 80  # um
wafer_length, wafer_width = 4000, 3000  # um
chip1_length, chip1_width = 1760, 1460  # um
chip2_length, chip2_width = 1046, 1146  # um
chip3_length, chip3_width = 1096, 846  # um
chip4_length, chip4_width = 1050, 2160  # um
chip5_length, chip5_width = 1770, 800  # um
chip6_length, chip6_width = 1500, 500  # um
```

In [ ]:

In [30]:

```python
chips, lengths, widths = gp.multidict({
    '芯片1': [chip1_length, chip1_width],
    '芯片2': [chip2_length, chip2_width],
    '芯片3': [chip3_length, chip3_width],
    '芯片4': [chip4_length, chip4_width],
    '芯片5': [chip5_length, chip5_width],
    '芯片6': [chip6_length, chip6_width],
})
```

In [ ]:

In [31]:

```python
# rotation list
rotation_binary = []
for i in range(64):
    b = list(("".join(list(bin(i))[2:])).rjust(6, '0'))
    binary = []
    for s in b:
        binary.append(int(s))
    rotation_binary.append(binary)
# rotation_binary

# permutation list
```

```python
permutations = list(itertools.permutations([i for i in range(6)]))
# permutations

print(len(rotation_binary), len(permutations))
```

```
64 720
```

In [ ]:

In [32]:

```python
def is_overlap(loc1, loc2):
    """
    :param loc1: 一个矩形的对角线坐标 (四元组)
    :param loc2: 另一个矩形的对角线坐标 (四元组)
    :return: boolean 两个矩形是否重叠
    """
    x1, y1, x2, y2 = loc1
    x3, y3, x4, y4 = loc2
    L1, W1 = x2 - x1, y2 - y1
    L2, W2 = x4 - x3, y4 - y3
    Cx1, Cy1 = (x1 + x2) / 2, (y1 + y2) / 2
    Cx2, Cy2 = (x3 + x4) / 2, (y3 + y4) / 2
    Dx, Dy = abs(Cx1 - Cx2), abs(Cy1 - Cy2)
    return Dx < (L1 + L2) / 2 and Dy < (W1 + W2) / 2

def judge_put(to_put_chip_real_loc, chip_swell_loc):
    """
    :param to_put_chip_real_loc: 即将放置的芯片真实坐标
    :param chip_swell_loc: 已经放置的芯片的膨胀坐标
    :return: boolean 是否可以放即将放置的芯片
    """
    x1, y1, x2, y2 = to_put_chip_real_loc
    for key in chip_swell_loc.keys():
        if is_overlap((x1, y1, x2, y2), chip_swell_loc[key]):
            return False
    if 0 <= x1 <= wafer_length and  0 <= y1 <= wafer_width and 0 <= x2 <= wafer_length and  0 <= y2 <= wafer_width:
        return True
    else:
        return False
def real2swell(x1, y1, x2, y2):
    return (max(x1 - channel_width, 0),
            max(y1 - channel_width, 0),
            min(x2 + channel_width, wafer_length),
            min(y2 + channel_width, wafer_width))
def judge_no_above_level(above_level_chip):
    """
    :param above_level_chip:
```

```python
    :return: 判断是否全部都在基准线之下
    """
    return list(above_level_chip.values()) == [False for _ in range(len(above_level_chip))]
def update_above_level_chip(level, above_level_chip, chip_swell_loc):
    min_y = wafer_width
    for key in above_level_chip.keys():
        if above_level_chip[key]:
            min_y = min_y if min_y < chip_swell_loc[key][3] else chip_swell_loc[key][3]
    level = min_y  # 更新基准线
    for key in chip_swell_loc.keys():  # 更新基准线之上的芯片
        if chip_swell_loc[key][3] <= min_y:
            above_level_chip[key] = False
    return level, above_level_chip
def put_chip_on_levelleft(chip, level, chip_real_loc, chip_swell_loc, dx, dy):
    """
    :param :
    在基准线的最左边放置一个芯片
    """
    x1, y1, x2, y2 = 0, level, 0, level
    putx1, puty1, putx2, puty2 = x1, level, x1 + dx, level + dy  # 基准线之上的芯片右边放置一个芯片
    if judge_put((putx1, puty1, putx2, puty2), chip_swell_loc):  # 如果可以放置
        chip_real_loc[chip] = (putx1, puty1, putx2, puty2)
        chip_swell_loc[chip] = real2swell(putx1, puty1, putx2, puty2)
        return [True, chip_real_loc, chip_swell_loc]
    else:
        return [False]
def put_chip(chip, length, width, chip_real_loc, chip_swell_loc):
    """
    :param chip: (str) 准备放置的芯片
    :para length: 准备放置芯片的长
    :para width: 准备放置芯片的宽
    :param chip_real_loc: 已经放置的芯片的真实坐标
    :param chip_swell_loc: 已经放置的芯片的膨胀坐标
    :return: [boolean, chip_real_loc, chip_swell_loc]
    """
    level = 0
    dx, dy = length, width

    # 首先在基准线最左点尝试是否可以放置（判断是否可以放置）
    res = put_chip_on_levelleft(chip, level, chip_real_loc, chip_swell_loc, dx, dy)
    if res[0]:
        return res
    above_level_chip = dict()  # 位置在基准线之上的芯片
    for key in chip_swell_loc.keys():
        above_level_chip[key] = True
    while not judge_no_above_level(above_level_chip):  # 如果还有在基准线之上的芯片
```

```python
#           print(above_level_chip)
        for key in above_level_chip.keys():  # 遍历在基准线之上的芯片
            if above_level_chip[key]:  # 找到的基准线之上的芯片
                x1, y1, x2, y2 = chip_swell_loc[key]
                putx1, puty1, putx2, puty2 = x2, level, x2 + dx, level + dy
                if judge_put((putx1, puty1, putx2, puty2), chip_swell_loc):
                    chip_real_loc[chip] = (putx1, puty1, putx2, puty2)
                    chip_swell_loc[chip] = real2swell(putx1, puty1, putx2, puty2)
                    return [True, chip_real_loc, chip_swell_loc]
        # 如果不能放置
        level, above_level_chip = update_above_level_chip(level,
                                                          above_level_chip,
                                                          chip_swell_loc)  # 更新基准线和基准线之上的芯片
        if level >= wafer_width:
            return [False]

        res = put_chip_on_levelleft(chip, level, chip_real_loc, chip_swell_loc, dx, dy)
        if res[0]:
            return res

    # 最后，此时已经没有在基准线之上的芯片了，基准线也是最大值，于是直接在基准线之上的最左点处放置一个芯片（判断是否可以放置）
    res = put_chip_on_levelleft(chip, level, chip_real_loc, chip_swell_loc, dx, dy)
    if res[0]:
        return res
    return [False]

def put_chips(permutation, chips, lengths, widths):
    chip_real_loc, chip_swell_loc = dict(), dict()
    for order in permutation:  # 按照顺序放置芯片
        chip = chips[order]
        length = lengths[chip]
        width = widths[chip]
        res = put_chip(chip, length, width, chip_real_loc, chip_swell_loc)
        if res[0]:  # 如果芯片可以放置
            _, chip_real_loc, chip_swell_loc = res
        else:  # 如果芯片不可以放置
            return [False]
    return [True, chip_real_loc, chip_swell_loc]
```

In [ ]:

In [33]:

```python
def place_chips(permutation, rotation):
    # 确定宽长
    chips, lengths, widths = gp.multidict({
        '芯片1': [chip1_length * (1 - rotation[0]) + chip1_width * rotation[0],
                 chip1_width * (1 - rotation[0]) + chip1_length * rotation[0]],
```

```
        '芯片2': [chip2_length * (1 - rotation[1]) + chip2_width * rotation[1],
                  chip2_width * (1 - rotation[1]) + chip2_length * rotation[1]],
        '芯片3': [chip3_length * (1 - rotation[2]) + chip3_width * rotation[2],
                  chip3_width * (1 - rotation[2]) + chip3_length * rotation[2]],
        '芯片4': [chip4_length * (1 - rotation[3]) + chip4_width * rotation[3],
                  chip4_width * (1 - rotation[3]) + chip4_length * rotation[3]],
        '芯片5': [chip5_length * (1 - rotation[4]) + chip5_width * rotation[4],
                  chip5_width * (1 - rotation[4]) + chip5_length * rotation[4]],
        '芯片6': [chip6_length * (1 - rotation[5]) + chip6_width * rotation[5],
                  chip6_width * (1 - rotation[5]) + chip6_length * rotation[5]],
    })
    # 检查
#     print(lengths)
#     print(widths)
#     print()
    # 按照顺序放置芯片
    res = put_chips(permutation, chips, lengths, widths)
    return res
```

In [ ]:

In [34]:

```
# %%time
chip_real_loc, chip_swell_loc = [], []

from time import time
t0 = time()
for epoch, permutation in enumerate(permutations):
    for rotation in rotation_binary:
        res = place_chips(permutation, rotation)
        if res[0]:
            _, real_loc, swell_loc = res
            chip_real_loc.append(real_loc)
            chip_swell_loc.append(swell_loc)
    if epoch % (len(permutations) // 10) == 0:
        print("epoch:", epoch, "time:", time() - t0)
print(len(chip_real_loc), len(chip_swell_loc))
```

```
epoch: 0 time: 0.016350507736206055
epoch: 72 time: 1.034231424331665
epoch: 144 time: 2.066129446029663
epoch: 216 time: 2.9790122509002686
epoch: 288 time: 4.2622199058532715
epoch: 360 time: 5.304689884185791
epoch: 432 time: 6.52071213722229
epoch: 504 time: 7.691202640533447
epoch: 576 time: 8.830366611480713
epoch: 648 time: 9.976160764694214
2425 2425
```

In [35]:

```
pre = 50

for idx, i in enumerate(chip_real_loc[:pre]):
    print(idx)
    print(i)
```

0
{'芯片1': (0, 0, 1760, 1460), '芯片2': (1840, 0, 2886, 1146), '芯片3': (2966, 0, 3812, 1096), '芯片5': (2966, 1176, 3766, 2946), '芯片4':
(0, 1540, 2160, 2590), '芯片6': (2240, 1226, 2740, 2726)}
1
{'芯片1': (0, 0, 1460, 1760), '芯片2': (1540, 0, 2586, 1146), '芯片3': (2666, 0, 3762, 846), '芯片6': (1540, 1226, 3040, 1726), '芯片5': (3
120, 926, 3920, 2696), '芯片4': (0, 1840, 2160, 2890)}
2
{'芯片1': (0, 0, 1460, 1760), '芯片2': (1540, 0, 2586, 1146), '芯片3': (2666, 0, 3512, 1096), '芯片6': (1540, 1226, 3040, 1726), '芯片5':
(3120, 1176, 3920, 2946), '芯片4': (0, 1840, 2160, 2890)}
3
{'芯片1': (0, 0, 1460, 1760), '芯片2': (1540, 0, 2686, 1046), '芯片3': (2766, 0, 3862, 846), '芯片6': (1540, 1126, 3040, 1626), '芯片5': (3
120, 926, 3920, 2696), '芯片4': (0, 1840, 2160, 2890)}
4
{'芯片1': (0, 0, 1460, 1760), '芯片2': (1540, 0, 2686, 1046), '芯片3': (2766, 0, 3612, 1096), '芯片6': (1540, 1176, 3040, 1676), '芯片5':
(3120, 1176, 3920, 2946), '芯片4': (0, 1840, 2160, 2890)}
5
{'芯片1': (0, 0, 1460, 1760), '芯片2': (1540, 0, 2586, 1146), '芯片4': (2666, 0, 3716, 2160), '芯片3': (0, 1840, 1096, 2686), '芯片5': (154
0, 1226, 2340, 2996), '芯片6': (2420, 2240, 3920, 2740)}
6
{'芯片1': (0, 0, 1460, 1760), '芯片2': (1540, 0, 2686, 1046), '芯片4': (2766, 0, 3816, 2160), '芯片3': (1540, 1126, 2636, 1972), '芯片5':
(0, 2052, 1770, 2852), '芯片6': (1850, 2240, 3350, 2740)}
7
{'芯片1': (0, 0, 1460, 1760), '芯片2': (1540, 0, 2586, 1146), '芯片4': (2666, 0, 3716, 2160), '芯片5': (1540, 1226, 2340, 2996), '芯片3':
(0, 1840, 1096, 2686), '芯片6': (2420, 2240, 3920, 2740)}
8
{'芯片1': (0, 0, 1460, 1760), '芯片2': (1540, 0, 2586, 1146), '芯片4': (2666, 0, 3716, 2160), '芯片5': (1540, 1226, 2340, 2996), '芯片3':
(0, 1840, 846, 2936), '芯片6': (2420, 2240, 3920, 2740)}
9
{'芯片1': (0, 0, 1460, 1760), '芯片2': (1540, 0, 2686, 1046), '芯片4': (2766, 0, 3816, 2160), '芯片5': (1540, 1126, 2340, 2896), '芯片3':
(0, 1840, 1096, 2686), '芯片6': (2420, 2240, 3920, 2740)}
10
{'芯片1': (0, 0, 1460, 1760), '芯片2': (1540, 0, 2686, 1046), '芯片4': (2766, 0, 3816, 2160), '芯片5': (1540, 1126, 2340, 2896), '芯片3':
(0, 1840, 846, 2936), '芯片6': (2420, 2240, 3920, 2740)}
11
{'芯片1': (0, 0, 1460, 1760), '芯片2': (1540, 0, 2586, 1146), '芯片4': (2666, 0, 3716, 2160), '芯片5': (1540, 1226, 2340, 2996), '芯片6':
(2420, 2240, 3920, 2740), '芯片3': (0, 1840, 1096, 2686)}
12
{'芯片1': (0, 0, 1460, 1760), '芯片2': (1540, 0, 2586, 1146), '芯片4': (2666, 0, 3716, 2160), '芯片5': (1540, 1226, 2340, 2996), '芯片6':
(2420, 2240, 3920, 2740), '芯片3': (0, 1840, 846, 2936)}
13
{'芯片1': (0, 0, 1460, 1760), '芯片2': (1540, 0, 2686, 1046), '芯片4': (2766, 0, 3816, 2160), '芯片5': (1540, 1126, 2340, 2896), '芯片6':
(2420, 2240, 3920, 2740), '芯片3': (0, 1840, 1096, 2686)}
14
{'芯片1': (0, 0, 1460, 1760), '芯片2': (1540, 0, 2686, 1046), '芯片4': (2766, 0, 3816, 2160), '芯片5': (1540, 1126, 2340, 2896), '芯片6':
(2420, 2240, 3920, 2740), '芯片3': (0, 1840, 846, 2936)}
15

```
{'芯片1': (0, 0, 1760, 1460), '芯片2': (1840, 0, 2886, 1146), '芯片5': (2966, 0, 3766, 1770), '芯片4': (0, 1540, 2160, 2590), '芯片6': (224
0, 1226, 2740, 2726), '芯片3': (2820, 1850, 3916, 2696)}
16
{'芯片1': (0, 0, 1760, 1460), '芯片2': (1840, 0, 2886, 1146), '芯片5': (2966, 0, 3766, 1770), '芯片4': (0, 1540, 2160, 2590), '芯片6': (224
0, 1226, 2740, 2726), '芯片3': (2820, 1850, 3666, 2946)}
17
{'芯片1': (0, 0, 1760, 1460), '芯片2': (1840, 0, 2986, 1046), '芯片5': (3066, 0, 3866, 1770), '芯片4': (0, 1540, 2160, 2590), '芯片6': (224
0, 1126, 2740, 2626), '芯片3': (2820, 1850, 3916, 2696)}
18
{'芯片1': (0, 0, 1760, 1460), '芯片2': (1840, 0, 2986, 1046), '芯片5': (3066, 0, 3866, 1770), '芯片4': (0, 1540, 2160, 2590), '芯片6': (224
0, 1126, 2740, 2626), '芯片3': (2820, 1850, 3666, 2946)}
19
{'芯片1': (0, 0, 1760, 1460), '芯片2': (1840, 0, 2886, 1146), '芯片5': (2966, 0, 3766, 1770), '芯片6': (0, 1540, 1500, 2040), '芯片4': (158
0, 1850, 3740, 2900), '芯片3': (0, 2120, 1096, 2966)}
20
{'芯片1': (0, 0, 1760, 1460), '芯片2': (1840, 0, 2986, 1046), '芯片5': (3066, 0, 3866, 1770), '芯片6': (0, 1540, 1500, 2040), '芯片4': (158
0, 1850, 3740, 2900), '芯片3': (0, 2120, 1096, 2966)}
21
{'芯片1': (0, 0, 1460, 1760), '芯片2': (1540, 0, 2586, 1146), '芯片6': (1540, 1226, 3040, 1726), '芯片3': (2666, 0, 3762, 846), '芯片5': (3
120, 926, 3920, 2696), '芯片4': (0, 1840, 2160, 2890)}
22
{'芯片1': (0, 0, 1460, 1760), '芯片2': (1540, 0, 2586, 1146), '芯片6': (1540, 1226, 3040, 1726), '芯片3': (2666, 0, 3512, 1096), '芯片5':
(3120, 1176, 3920, 2946), '芯片4': (0, 1840, 2160, 2890)}
23
{'芯片1': (0, 0, 1460, 1760), '芯片2': (1540, 0, 2686, 1046), '芯片6': (1540, 1126, 3040, 1626), '芯片3': (2766, 0, 3862, 846), '芯片5': (3
120, 926, 3920, 2696), '芯片4': (0, 1840, 2160, 2890)}
24
{'芯片1': (0, 0, 1460, 1760), '芯片2': (1540, 0, 2686, 1046), '芯片6': (1540, 1126, 3040, 1626), '芯片3': (3120, 0, 3966, 1096), '芯片5':
(3120, 1176, 3920, 2946), '芯片4': (0, 1840, 2160, 2890)}
25
{'芯片1': (0, 0, 1460, 1760), '芯片2': (1540, 0, 2586, 1146), '芯片6': (1540, 1226, 3040, 1726), '芯片5': (3120, 0, 3920, 1770), '芯片4':
(0, 1840, 2160, 2890), '芯片3': (2240, 1850, 3336, 2696)}
26
{'芯片1': (0, 0, 1460, 1760), '芯片2': (1540, 0, 2586, 1146), '芯片6': (1540, 1226, 3040, 1726), '芯片5': (3120, 0, 3920, 1770), '芯片4':
(0, 1840, 2160, 2890), '芯片3': (2240, 1850, 3086, 2946)}
27
{'芯片1': (0, 0, 1460, 1760), '芯片2': (1540, 0, 2686, 1046), '芯片6': (1540, 1126, 3040, 1626), '芯片5': (3120, 0, 3920, 1770), '芯片4':
(0, 1840, 2160, 2890), '芯片3': (2240, 1850, 3336, 2696)}
28
{'芯片1': (0, 0, 1460, 1760), '芯片2': (1540, 0, 2686, 1046), '芯片6': (1540, 1126, 3040, 1626), '芯片5': (3120, 0, 3920, 1770), '芯片4':
(0, 1840, 2160, 2890), '芯片3': (2240, 1850, 3086, 2946)}
29
{'芯片1': (0, 0, 1460, 1760), '芯片3': (1540, 0, 2386, 1096), '芯片2': (2466, 0, 3612, 1046), '芯片5': (2466, 1126, 3266, 2896), '芯片4':
(0, 1840, 2160, 2890), '芯片6': (3346, 1126, 3846, 2626)}
30
{'芯片1': (0, 0, 1460, 1760), '芯片3': (1540, 0, 2386, 1096), '芯片2': (2466, 0, 3612, 1046), '芯片5': (2466, 1126, 3266, 2896), '芯片6':
```

(3346, 1126, 3846, 2626), '芯片4': (0, 1840, 2160, 2890)}
31
{'芯片1': (0, 0, 1460, 1760), '芯片3': (1540, 0, 2386, 1096), '芯片2': (2466, 0, 3612, 1046), '芯片6': (2466, 1126, 2966, 2626), '芯片4': (0, 1840, 2160, 2890), '芯片5': (3046, 1126, 3846, 2896)}
32
{'芯片1': (0, 0, 1460, 1760), '芯片3': (1540, 0, 2636, 846), '芯片2': (2716, 0, 3762, 1146), '芯片6': (1540, 1226, 3040, 1726), '芯片5': (3120, 1226, 3920, 2996), '芯片4': (0, 1840, 2160, 2890)}
33
{'芯片1': (0, 0, 1460, 1760), '芯片3': (1540, 0, 2386, 1096), '芯片2': (2466, 0, 3512, 1146), '芯片6': (1540, 1226, 3040, 1726), '芯片5': (3120, 1226, 3920, 2996), '芯片4': (0, 1840, 2160, 2890)}
34
{'芯片1': (0, 0, 1460, 1760), '芯片3': (1540, 0, 2636, 846), '芯片2': (2716, 0, 3862, 1046), '芯片6': (1540, 1126, 3040, 1626), '芯片5': (3120, 1126, 3920, 2896), '芯片4': (0, 1840, 2160, 2890)}
35
{'芯片1': (0, 0, 1460, 1760), '芯片3': (1540, 0, 2386, 1096), '芯片2': (2466, 0, 3612, 1046), '芯片6': (2466, 1126, 2966, 2626), '芯片5': (3046, 1126, 3846, 2896), '芯片4': (0, 1840, 2160, 2890)}
36
{'芯片1': (0, 0, 1460, 1760), '芯片3': (1540, 0, 2636, 846), '芯片4': (2716, 0, 3766, 2160), '芯片2': (1540, 926, 2586, 2072), '芯片5': (0, 2152, 1770, 2952), '芯片6': (1850, 2240, 3350, 2740)}
37
{'芯片1': (0, 0, 1460, 1760), '芯片3': (1540, 0, 2386, 1096), '芯片4': (2466, 0, 3516, 2160), '芯片2': (0, 1840, 1046, 2986), '芯片5': (1540, 1176, 2340, 2946), '芯片6': (2420, 2240, 3920, 2740)}
38
{'芯片1': (0, 0, 1460, 1760), '芯片3': (1540, 0, 2636, 846), '芯片4': (2716, 0, 3766, 2160), '芯片2': (0, 1840, 1146, 2886), '芯片5': (1540, 926, 2340, 2696), '芯片6': (2420, 2240, 3920, 2740)}
39
{'芯片1': (0, 0, 1460, 1760), '芯片3': (1540, 0, 2386, 1096), '芯片4': (2466, 0, 3516, 2160), '芯片2': (0, 1840, 1146, 2886), '芯片5': (1540, 1176, 2340, 2946), '芯片6': (2420, 2240, 3920, 2740)}
40
{'芯片1': (0, 0, 1460, 1760), '芯片3': (1540, 0, 2636, 846), '芯片4': (2716, 0, 3766, 2160), '芯片5': (1540, 926, 2340, 2696), '芯片2': (0, 1840, 1046, 2986), '芯片6': (2420, 2240, 3920, 2740)}
41
{'芯片1': (0, 0, 1460, 1760), '芯片3': (1540, 0, 2386, 1096), '芯片4': (2466, 0, 3516, 2160), '芯片5': (1540, 1176, 2340, 2946), '芯片2': (0, 1840, 1046, 2986), '芯片6': (2420, 2240, 3920, 2740)}
42
{'芯片1': (0, 0, 1460, 1760), '芯片3': (1540, 0, 2636, 846), '芯片4': (2716, 0, 3766, 2160), '芯片5': (1540, 926, 2340, 2696), '芯片2': (0, 1840, 1146, 2886), '芯片6': (2420, 2240, 3920, 2740)}
43
{'芯片1': (0, 0, 1460, 1760), '芯片3': (1540, 0, 2386, 1096), '芯片4': (2466, 0, 3516, 2160), '芯片5': (1540, 1176, 2340, 2946), '芯片2': (0, 1840, 1146, 2886), '芯片6': (2420, 2240, 3920, 2740)}
44
{'芯片1': (0, 0, 1460, 1760), '芯片3': (1540, 0, 2636, 846), '芯片4': (2716, 0, 3766, 2160), '芯片5': (1540, 926, 2340, 2696), '芯片6': (2420, 2240, 3920, 2740), '芯片2': (0, 1840, 1046, 2986)}
45
{'芯片1': (0, 0, 1460, 1760), '芯片3': (1540, 0, 2386, 1096), '芯片4': (2466, 0, 3516, 2160), '芯片5': (1540, 1176, 2340, 2946), '芯片6': (2420, 2240, 3920, 2740), '芯片2': (0, 1840, 1046, 2986)}

```
46
{'芯片1': (0, 0, 1460, 1760), '芯片3': (1540, 0, 2636, 846), '芯片4': (2716, 0, 3766, 2160), '芯片5': (1540, 926, 2340, 2696), '芯片6': (2420, 2240, 3920, 2740), '芯片2': (0, 1840, 1146, 2886)}
47
{'芯片1': (0, 0, 1460, 1760), '芯片3': (1540, 0, 2386, 1096), '芯片4': (2466, 0, 3516, 2160), '芯片5': (1540, 1176, 2340, 2946), '芯片6': (2420, 2240, 3920, 2740), '芯片2': (0, 1840, 1146, 2886)}
48
{'芯片1': (0, 0, 1460, 1760), '芯片3': (1540, 0, 2386, 1096), '芯片5': (2466, 0, 3266, 1770), '芯片2': (0, 1840, 1046, 2986), '芯片4': (1126, 1850, 3286, 2900), '芯片6': (3346, 0, 3846, 1500)}
49
{'芯片1': (0, 0, 1460, 1760), '芯片3': (1540, 0, 2386, 1096), '芯片5': (2466, 0, 3266, 1770), '芯片2': (0, 1840, 1146, 2886), '芯片4': (1226, 1850, 3386, 2900), '芯片6': (3346, 0, 3846, 1500)}
```

In [36]:

```
for idx, i in enumerate(chip_swell_loc[:pre]):
    print(idx)
    print(i)
```

0
{'芯片1': (0, 0, 1840, 1540), '芯片2': (1760, 0, 2966, 1226), '芯片3': (2886, 0, 3892, 1176), '芯片5': (2886, 1096, 3846, 3000), '芯片4': (0, 1460, 2240, 2670), '芯片6': (2160, 1146, 2820, 2806)}
1
{'芯片1': (0, 0, 1540, 1840), '芯片2': (1460, 0, 2666, 1226), '芯片3': (2586, 0, 3842, 926), '芯片6': (1460, 1146, 3120, 1806), '芯片5': (3040, 846, 4000, 2776), '芯片4': (0, 1760, 2240, 2970)}
2
{'芯片1': (0, 0, 1540, 1840), '芯片2': (1460, 0, 2666, 1226), '芯片3': (2586, 0, 3592, 1176), '芯片6': (1460, 1146, 3120, 1806), '芯片5': (3040, 1096, 4000, 3000), '芯片4': (0, 1760, 2240, 2970)}
3
{'芯片1': (0, 0, 1540, 1840), '芯片2': (1460, 0, 2766, 1126), '芯片3': (2686, 0, 3942, 926), '芯片6': (1460, 1046, 3120, 1706), '芯片5': (3040, 846, 4000, 2776), '芯片4': (0, 1760, 2240, 2970)}
4
{'芯片1': (0, 0, 1540, 1840), '芯片2': (1460, 0, 2766, 1126), '芯片3': (2686, 0, 3692, 1176), '芯片6': (1460, 1096, 3120, 1756), '芯片5': (3040, 1096, 4000, 3000), '芯片4': (0, 1760, 2240, 2970)}
5
{'芯片1': (0, 0, 1540, 1840), '芯片2': (1460, 0, 2666, 1226), '芯片4': (2586, 0, 3796, 2240), '芯片3': (0, 1760, 1176, 2766), '芯片5': (1460, 1146, 2420, 3000), '芯片6': (2340, 2160, 4000, 2820)}
6
{'芯片1': (0, 0, 1540, 1840), '芯片2': (1460, 0, 2766, 1126), '芯片4': (2686, 0, 3896, 2240), '芯片3': (1460, 1046, 2716, 2052), '芯片5': (0, 1972, 1850, 2932), '芯片6': (1770, 2160, 3430, 2820)}
7
{'芯片1': (0, 0, 1540, 1840), '芯片2': (1460, 0, 2666, 1226), '芯片4': (2586, 0, 3796, 2240), '芯片5': (1460, 1146, 2420, 3000), '芯片3': (0, 1760, 1176, 2766), '芯片6': (2340, 2160, 4000, 2820)}
8
{'芯片1': (0, 0, 1540, 1840), '芯片2': (1460, 0, 2666, 1226), '芯片4': (2586, 0, 3796, 2240), '芯片5': (1460, 1146, 2420, 3000), '芯片3': (0, 1760, 926, 3000), '芯片6': (2340, 2160, 4000, 2820)}
9
{'芯片1': (0, 0, 1540, 1840), '芯片2': (1460, 0, 2766, 1126), '芯片4': (2686, 0, 3896, 2240), '芯片5': (1460, 1046, 2420, 2976), '芯片3': (0, 1760, 1176, 2766), '芯片6': (2340, 2160, 4000, 2820)}
10
{'芯片1': (0, 0, 1540, 1840), '芯片2': (1460, 0, 2766, 1126), '芯片4': (2686, 0, 3896, 2240), '芯片5': (1460, 1046, 2420, 2976), '芯片3': (0, 1760, 926, 3000), '芯片6': (2340, 2160, 4000, 2820)}
11
{'芯片1': (0, 0, 1540, 1840), '芯片2': (1460, 0, 2666, 1226), '芯片4': (2586, 0, 3796, 2240), '芯片5': (1460, 1146, 2420, 3000), '芯片6': (2340, 2160, 4000, 2820), '芯片3': (0, 1760, 1176, 2766)}
12
{'芯片1': (0, 0, 1540, 1840), '芯片2': (1460, 0, 2666, 1226), '芯片4': (2586, 0, 3796, 2240), '芯片5': (1460, 1146, 2420, 3000), '芯片6': (2340, 2160, 4000, 2820), '芯片3': (0, 1760, 926, 3000)}
13
{'芯片1': (0, 0, 1540, 1840), '芯片2': (1460, 0, 2766, 1126), '芯片4': (2686, 0, 3896, 2240), '芯片5': (1460, 1046, 2420, 2976), '芯片6': (2340, 2160, 4000, 2820), '芯片3': (0, 1760, 1176, 2766)}
14
{'芯片1': (0, 0, 1540, 1840), '芯片2': (1460, 0, 2766, 1126), '芯片4': (2686, 0, 3896, 2240), '芯片5': (1460, 1046, 2420, 2976), '芯片6': (2340, 2160, 4000, 2820), '芯片3': (0, 1760, 926, 3000)}
15

{'芯片1': (0, 0, 1840, 1540), '芯片2': (1760, 0, 2966, 1226), '芯片5': (2886, 0, 3846, 1850), '芯片4': (0, 1460, 2240, 2670), '芯片6': (2160, 1146, 2820, 2806), '芯片3': (2740, 1770, 3996, 2776)}
16
{'芯片1': (0, 0, 1840, 1540), '芯片2': (1760, 0, 2966, 1226), '芯片5': (2886, 0, 3846, 1850), '芯片4': (0, 1460, 2240, 2670), '芯片6': (2160, 1146, 2820, 2806), '芯片3': (2740, 1770, 3746, 3000)}
17
{'芯片1': (0, 0, 1840, 1540), '芯片2': (1760, 0, 3066, 1126), '芯片5': (2986, 0, 3946, 1850), '芯片4': (0, 1460, 2240, 2670), '芯片6': (2160, 1046, 2820, 2706), '芯片3': (2740, 1770, 3996, 2776)}
18
{'芯片1': (0, 0, 1840, 1540), '芯片2': (1760, 0, 3066, 1126), '芯片5': (2986, 0, 3946, 1850), '芯片4': (0, 1460, 2240, 2670), '芯片6': (2160, 1046, 2820, 2706), '芯片3': (2740, 1770, 3746, 3000)}
19
{'芯片1': (0, 0, 1840, 1540), '芯片2': (1760, 0, 2966, 1226), '芯片5': (2886, 0, 3846, 1850), '芯片6': (0, 1460, 1580, 2120), '芯片4': (1500, 1770, 3820, 2980), '芯片3': (0, 2040, 1176, 3000)}
20
{'芯片1': (0, 0, 1840, 1540), '芯片2': (1760, 0, 3066, 1126), '芯片5': (2986, 0, 3946, 1850), '芯片6': (0, 1460, 1580, 2120), '芯片4': (1500, 1770, 3820, 2980), '芯片3': (0, 2040, 1176, 3000)}
21
{'芯片1': (0, 0, 1540, 1840), '芯片2': (1460, 0, 2666, 1226), '芯片6': (1460, 1146, 3120, 1806), '芯片3': (2586, 0, 3842, 926), '芯片5': (3040, 846, 4000, 2776), '芯片4': (0, 1760, 2240, 2970)}
22
{'芯片1': (0, 0, 1540, 1840), '芯片2': (1460, 0, 2666, 1226), '芯片6': (1460, 1146, 3120, 1806), '芯片3': (2586, 0, 3592, 1176), '芯片5': (3040, 1096, 4000, 3000), '芯片4': (0, 1760, 2240, 2970)}
23
{'芯片1': (0, 0, 1540, 1840), '芯片2': (1460, 0, 2766, 1126), '芯片6': (1460, 1046, 3120, 1706), '芯片3': (2686, 0, 3942, 926), '芯片5': (3040, 846, 4000, 2776), '芯片4': (0, 1760, 2240, 2970)}
24
{'芯片1': (0, 0, 1540, 1840), '芯片2': (1460, 0, 2766, 1126), '芯片6': (1460, 1046, 3120, 1706), '芯片3': (3040, 0, 4000, 1176), '芯片5': (3040, 1096, 4000, 3000), '芯片4': (0, 1760, 2240, 2970)}
25
{'芯片1': (0, 0, 1540, 1840), '芯片2': (1460, 0, 2666, 1226), '芯片6': (1460, 1146, 3120, 1806), '芯片5': (3040, 0, 4000, 1850), '芯片4': (0, 1760, 2240, 2970), '芯片3': (2160, 1770, 3416, 2776)}
26
{'芯片1': (0, 0, 1540, 1840), '芯片2': (1460, 0, 2666, 1226), '芯片6': (1460, 1146, 3120, 1806), '芯片5': (3040, 0, 4000, 1850), '芯片4': (0, 1760, 2240, 2970), '芯片3': (2160, 1770, 3166, 3000)}
27
{'芯片1': (0, 0, 1540, 1840), '芯片2': (1460, 0, 2766, 1126), '芯片6': (1460, 1046, 3120, 1706), '芯片5': (3040, 0, 4000, 1850), '芯片4': (0, 1760, 2240, 2970), '芯片3': (2160, 1770, 3416, 2776)}
28
{'芯片1': (0, 0, 1540, 1840), '芯片2': (1460, 0, 2766, 1126), '芯片6': (1460, 1046, 3120, 1706), '芯片5': (3040, 0, 4000, 1850), '芯片4': (0, 1760, 2240, 2970), '芯片3': (2160, 1770, 3166, 3000)}
29
{'芯片1': (0, 0, 1540, 1840), '芯片3': (1460, 0, 2466, 1176), '芯片2': (2386, 0, 3692, 1126), '芯片5': (2386, 1046, 3346, 2976), '芯片4': (0, 1760, 2240, 2970), '芯片6': (3266, 1046, 3926, 2706)}
30
{'芯片1': (0, 0, 1540, 1840), '芯片3': (1460, 0, 2466, 1176), '芯片2': (2386, 0, 3692, 1126), '芯片5': (2386, 1046, 3346, 2976), '芯片6':

```
(3266, 1046, 3926, 2706), '芯片4': (0, 1760, 2240, 2970)}
31
{'芯片1': (0, 0, 1540, 1840), '芯片3': (1460, 0, 2466, 1176), '芯片2': (2386, 0, 3692, 1126), '芯片6': (2386, 1046, 3046, 2706), '芯片4': (0, 1760, 2240, 2970), '芯片5': (2966, 1046, 3926, 2976)}
32
{'芯片1': (0, 0, 1540, 1840), '芯片3': (1460, 0, 2716, 926), '芯片2': (2636, 0, 3842, 1226), '芯片6': (1460, 1146, 3120, 1806), '芯片5': (3040, 1146, 4000, 3000), '芯片4': (0, 1760, 2240, 2970)}
33
{'芯片1': (0, 0, 1540, 1840), '芯片3': (1460, 0, 2466, 1176), '芯片2': (2386, 0, 3592, 1226), '芯片6': (1460, 1146, 3120, 1806), '芯片5': (3040, 1146, 4000, 3000), '芯片4': (0, 1760, 2240, 2970)}
34
{'芯片1': (0, 0, 1540, 1840), '芯片3': (1460, 0, 2716, 926), '芯片2': (2636, 0, 3942, 1126), '芯片6': (1460, 1046, 3120, 1706), '芯片5': (3040, 1046, 4000, 2976), '芯片4': (0, 1760, 2240, 2970)}
35
{'芯片1': (0, 0, 1540, 1840), '芯片3': (1460, 0, 2466, 1176), '芯片2': (2386, 0, 3692, 1126), '芯片6': (2386, 1046, 3046, 2706), '芯片5': (2966, 1046, 3926, 2976), '芯片4': (0, 1760, 2240, 2970)}
36
{'芯片1': (0, 0, 1540, 1840), '芯片3': (1460, 0, 2716, 926), '芯片4': (2636, 0, 3846, 2240), '芯片2': (1460, 846, 2666, 2152), '芯片5': (0, 2072, 1850, 3000), '芯片6': (1770, 2160, 3430, 2820)}
37
{'芯片1': (0, 0, 1540, 1840), '芯片3': (1460, 0, 2466, 1176), '芯片4': (2386, 0, 3596, 2240), '芯片2': (0, 1760, 1126, 3000), '芯片5': (1460, 1096, 2420, 3000), '芯片6': (2340, 2160, 4000, 2820)}
38
{'芯片1': (0, 0, 1540, 1840), '芯片3': (1460, 0, 2716, 926), '芯片4': (2636, 0, 3846, 2240), '芯片2': (0, 1760, 1226, 2966), '芯片5': (1460, 846, 2420, 2776), '芯片6': (2340, 2160, 4000, 2820)}
39
{'芯片1': (0, 0, 1540, 1840), '芯片3': (1460, 0, 2466, 1176), '芯片4': (2386, 0, 3596, 2240), '芯片2': (0, 1760, 1226, 2966), '芯片5': (1460, 1096, 2420, 3000), '芯片6': (2340, 2160, 4000, 2820)}
40
{'芯片1': (0, 0, 1540, 1840), '芯片3': (1460, 0, 2716, 926), '芯片4': (2636, 0, 3846, 2240), '芯片5': (1460, 846, 2420, 2776), '芯片2': (0, 1760, 1126, 3000), '芯片6': (2340, 2160, 4000, 2820)}
41
{'芯片1': (0, 0, 1540, 1840), '芯片3': (1460, 0, 2466, 1176), '芯片4': (2386, 0, 3596, 2240), '芯片5': (1460, 1096, 2420, 3000), '芯片2': (0, 1760, 1126, 3000), '芯片6': (2340, 2160, 4000, 2820)}
42
{'芯片1': (0, 0, 1540, 1840), '芯片3': (1460, 0, 2716, 926), '芯片4': (2636, 0, 3846, 2240), '芯片5': (1460, 846, 2420, 2776), '芯片2': (0, 1760, 1226, 2966), '芯片6': (2340, 2160, 4000, 2820)}
43
{'芯片1': (0, 0, 1540, 1840), '芯片3': (1460, 0, 2466, 1176), '芯片4': (2386, 0, 3596, 2240), '芯片5': (1460, 1096, 2420, 3000), '芯片2': (0, 1760, 1226, 2966), '芯片6': (2340, 2160, 4000, 2820)}
44
{'芯片1': (0, 0, 1540, 1840), '芯片3': (1460, 0, 2716, 926), '芯片4': (2636, 0, 3846, 2240), '芯片5': (1460, 846, 2420, 2776), '芯片6': (2340, 2160, 4000, 2820), '芯片2': (0, 1760, 1126, 3000)}
45
{'芯片1': (0, 0, 1540, 1840), '芯片3': (1460, 0, 2466, 1176), '芯片4': (2386, 0, 3596, 2240), '芯片5': (1460, 1096, 2420, 3000), '芯片6': (2340, 2160, 4000, 2820), '芯片2': (0, 1760, 1126, 3000)}
```

```
46
{'芯片1': (0, 0, 1540, 1840), '芯片3': (1460, 0, 2716, 926), '芯片4': (2636, 0, 3846, 2240), '芯片5': (1460, 846, 2420, 2776), '芯片6': (2340, 2160, 4000, 2820), '芯片2': (0, 1760, 1226, 2966)}
47
{'芯片1': (0, 0, 1540, 1840), '芯片3': (1460, 0, 2466, 1176), '芯片4': (2386, 0, 3596, 2240), '芯片5': (1460, 1096, 2420, 3000), '芯片6': (2340, 2160, 4000, 2820), '芯片2': (0, 1760, 1226, 2966)}
48
{'芯片1': (0, 0, 1540, 1840), '芯片3': (1460, 0, 2466, 1176), '芯片5': (2386, 0, 3346, 1850), '芯片2': (0, 1760, 1126, 3000), '芯片4': (1046, 1770, 3366, 2980), '芯片6': (3266, 0, 3926, 1580)}
49
{'芯片1': (0, 0, 1540, 1840), '芯片3': (1460, 0, 2466, 1176), '芯片5': (2386, 0, 3346, 1850), '芯片2': (0, 1760, 1226, 2966), '芯片4': (1146, 1770, 3466, 2980), '芯片6': (3266, 0, 3926, 1580)}
```

In [37]:

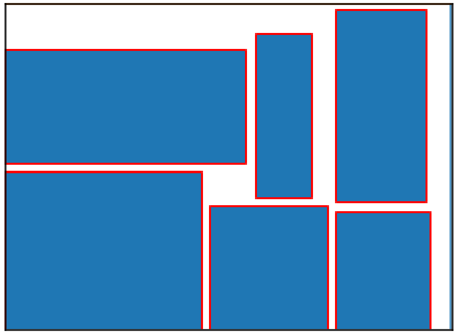
```python
def plot_chips(chip_real_loc, idx=0, save=False):
    fig = plt.figure(figsize=(4, 3))
    plt.xlim(0, wafer_length)
    plt.ylim(0, wafer_width)
    plt.xticks([])
    plt.yticks([])
    plt.plot([wafer_length, wafer_length], [0, wafer_width * 2])
    plt.plot([0, wafer_length * 2], [wafer_width, wafer_width])
    for key in chip_real_loc.keys():
        x1, y1, x2, y2 = chip_real_loc[key]
        x, y, w, h = x1, y1, x2 - x1, y2 - y1
        plt.gca().add_patch(plt.Rectangle((x,y),w,h, edgecolor='r'))
    if save:
        plt.savefig(f'./img/{idx}.png')
    plt.show()
```

In [ ]:

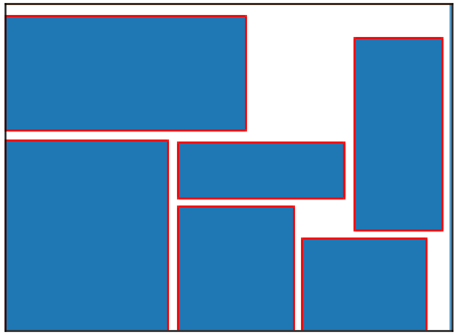
In [38]:

```python
# 当时随便取了前几个摆放方式算的
for idx, i in enumerate(chip_real_loc[:5]):
    print(idx)
    plot_chips(i, idx, True)
```

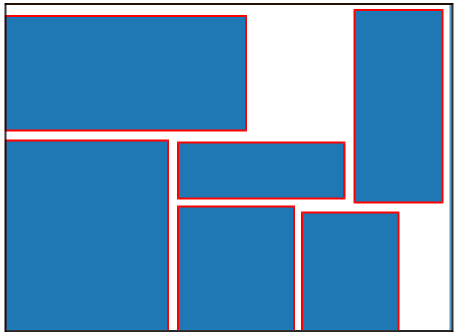
```
0
```



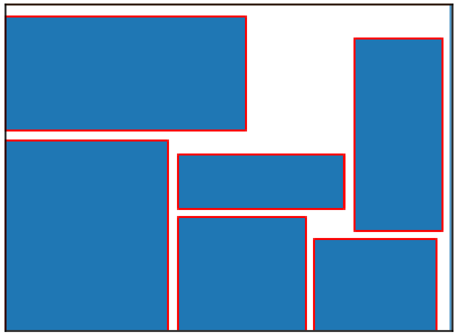
1



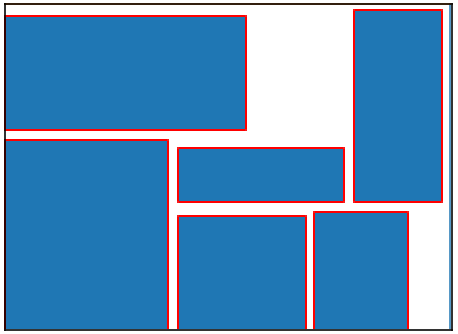
2



3

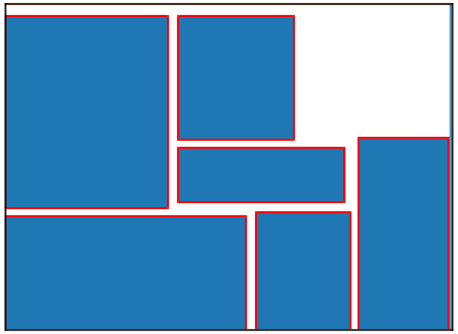


4

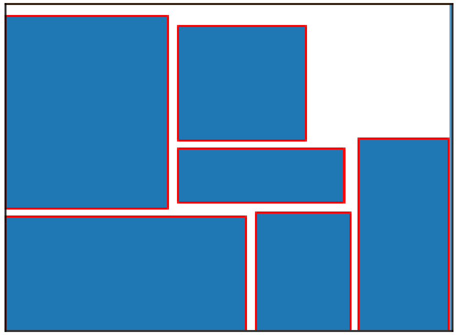


In [39]:
```
# 后面发现有几个更大的......
for idx, i in enumerate(chip_real_loc[1447:1449]):
    print(idx)
    plot_chips(i, idx, True)
```

0



1



计算面积

μm 换成 m！！！

这里本来思路很多的，但是前面卡壳了，题目给的数据有问题，但是举办方在第二天才改数据，我们后面就没时间做了，就人工算了几个看上去面积较大的

思路：

1. 选择芯片摆放方法：用 opencv 计算面积，越大越好
2. 放置环形振荡器：多目标规划......

In [40]:
```
import cv2 as cv

size_L, size_W = 1374, 800  # um
size_L, size_W = 1374e-6, 800e-6  # m

def func(N, P):
    return 1e3 * N * P + 1 / N
```

反相器个数： 3.0 PMOS宽：120.0nm PMOS长：23664.05nm
NMOS宽：120.0nm NMOS长：23664.05nm 面积： 2.2360964488008484e-11 频率：0.002076 MHz 功耗： 0.004618073247648539

反相器个数： 3.0 PMOS宽：510.54nm PMOS长：16960.9nm
NMOS宽：2951.4nm NMOS长：16960.9nm 面积： 1.837408128464667e-10 频率：0.002087 MHz 功耗： 0.04774641199924569

In [41]:
```
cal1 = func(63327, 0.004618073247648539 * 63327)
cal2 = func(62975, 0.004618073247648539 * 62975)

cal3 = func(6237, 0.04774641199924569 * 6237)
cal4 = func(5940, 0.04774641199924569 * 5940)
```

In [42]:
```
cal1, cal2, cal3, cal4
```

Out[42]:
```
(18519900379.82098, 18314588675.482754, 1857343495.9144454, 1684665302.4167535)
```

In [43]:
```
# 问题四结果
InteractiveShell.ast_node_interactivity = 'all'

0.004618073247648539 * 63327
0.004618073247648539 * 62975
0.04774641199924569 * 6237
0.04774641199924569 * 5940
```

Out[43]:
```
283.6136872755194
```

In [ ]: