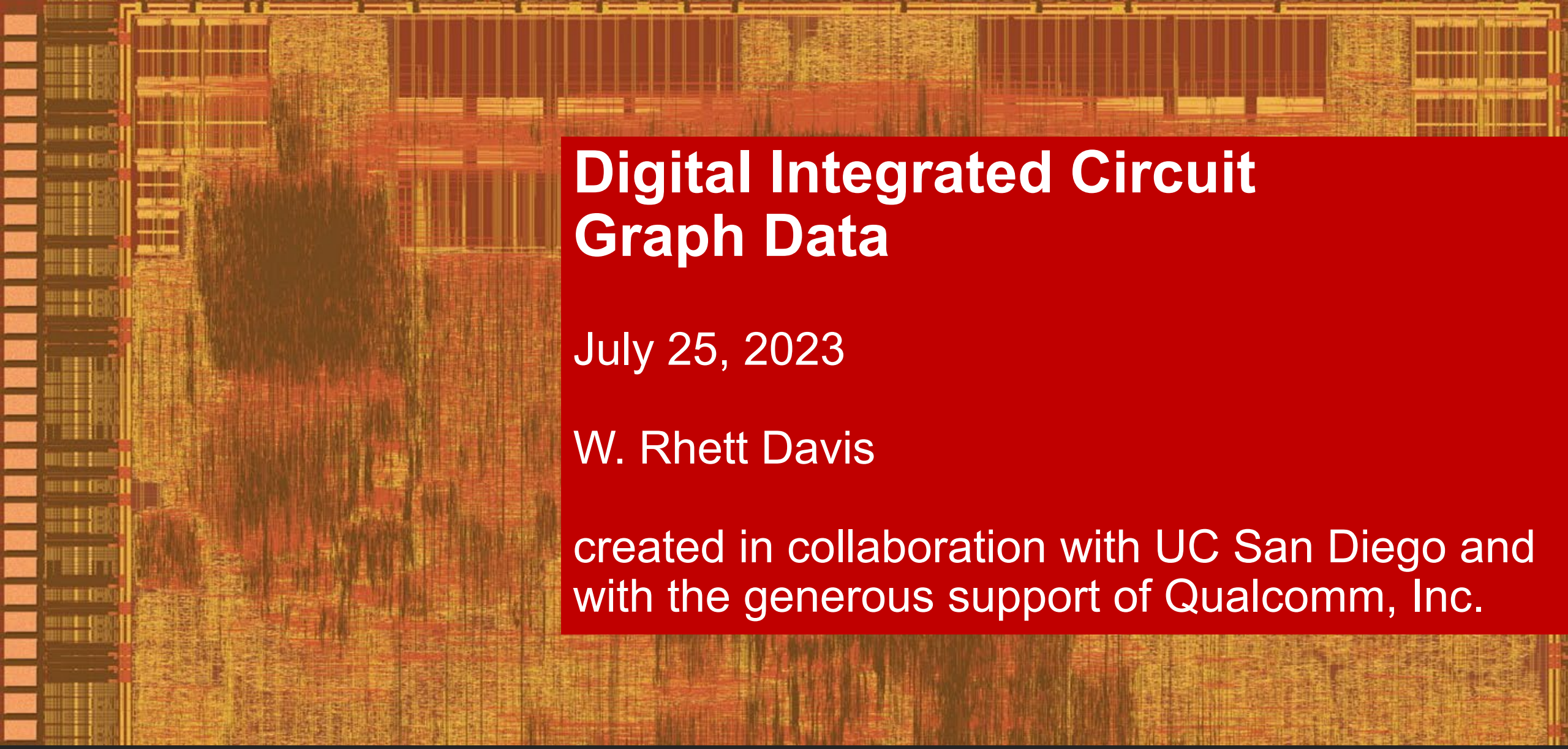


Digital Integrated Circuit Graph Data

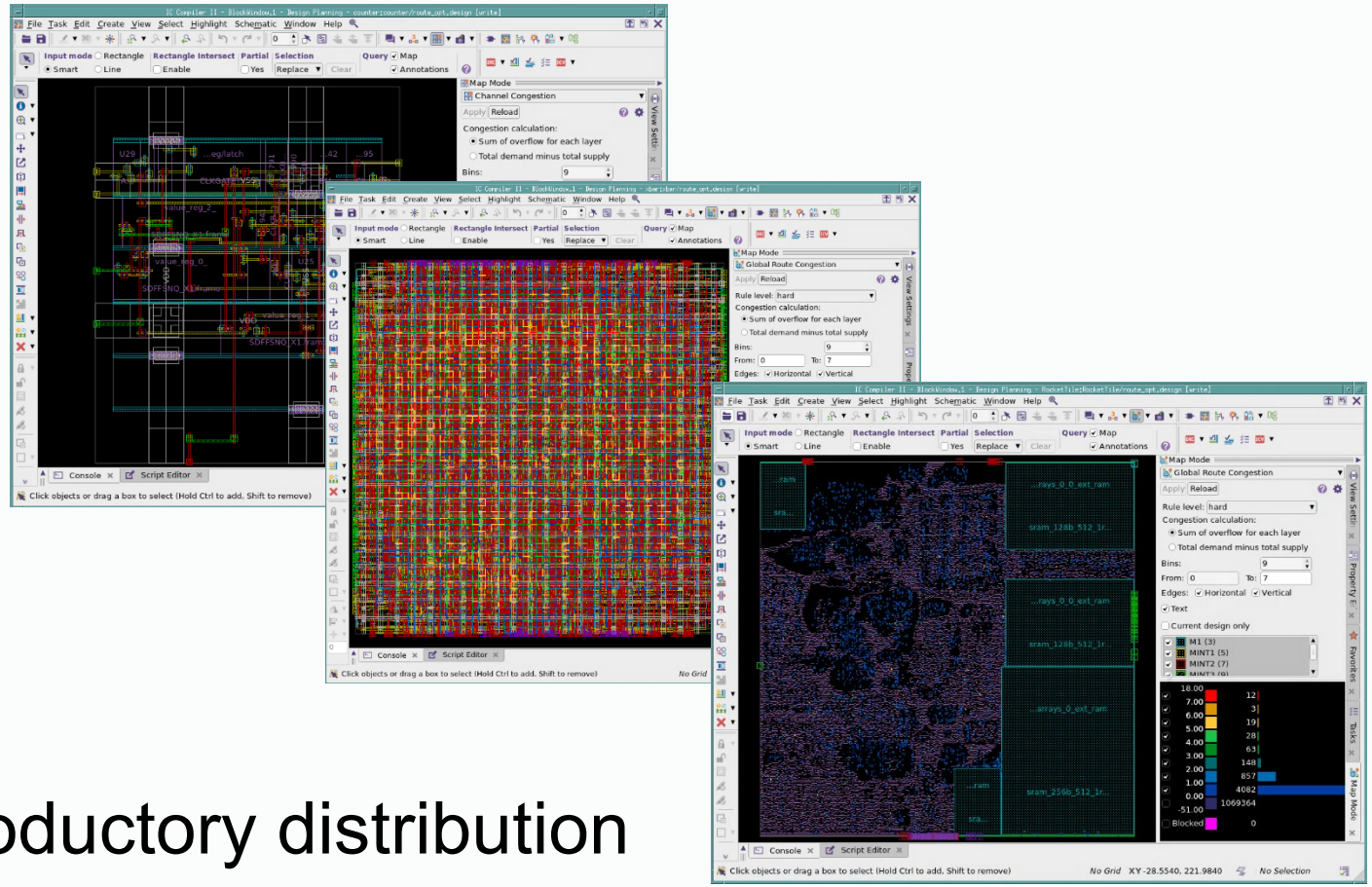
July 25, 2023

W. Rhett Davis

created in collaboration with UC San Diego and with the generous support of Qualcomm, Inc.

Example Designs

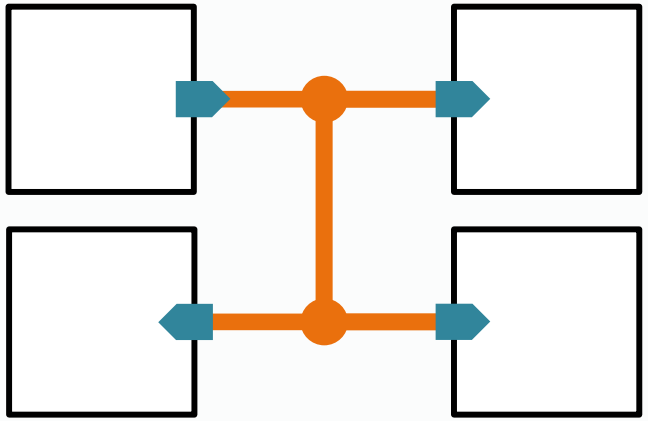
- counter
  - Simple starting example
  - 23 cells & 29 nets
- xbar
  - Highly congested routing
  - 3952 cells & 4484 nets
- RocketTile
  - Contains 5 memory macros
  - 183,560 cells & 84,494 nets
- Find these designs in the introductory distribution



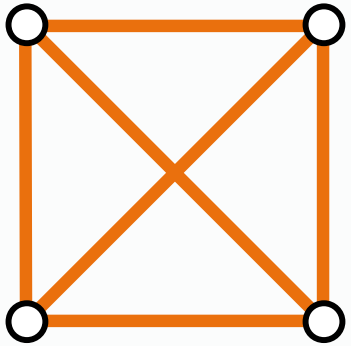
Hypergraph Representation

- How to represent hyperedges (*i.e.* nets) in a graph data-structure?
- Clique method is simple and popular, but high-fanout nets can dominate graph
- A bipartite graph with instance and net nodes is more efficient for these designs
- Best represented with a modified incidence matrix – sparse matrix with instance rows and net columns, value represents connected terminal index (or 0 if no connection)

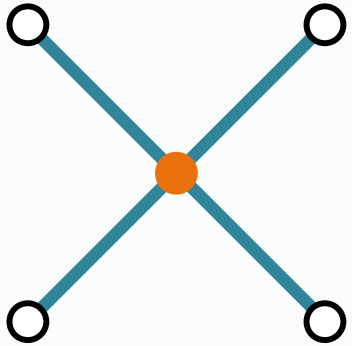
Hyperedge



Undirected Clique



Bipartite Graph



Distibution Contents

- Data organized in directories as [design]/[variant]
  - **[design].json.gz** – Feature data for each design, including instance and net data
  - **[design]_connectivity.npz** – Incidence matrix data as NumPy arrays
  - **[design]_congestion.npz** – congestion as NumPy arrays (if available)

- Top level directory contains additional information
  - **cells.json.gz** – Cell library data
  - **celllist** – Simple list of all library cell names
  - **settings.csv** – List of feature settings for each design variant
  - **README** – Additional details, most importantly the DBUtoUU divisor and details about each dataset

Design Feature Data (Nets)

- Because multi-dimensional sparse arrays are not yet widely available, we cannot easily encode feature data in the incidence matrix
- Therefore, feature data for instances and nets are provided in [design].json.gz
- Contains a JSON object with the ‘nets’ member, which is an array of net objects
- Each net object has the following members:
  - name (str) – name of the net
  - id (int) – index of the net in the array

```
>>> with gzip.open('counter.json.gz','rb') as f:
...     design = json.loads(f.read().decode('utf-8'))
>>> design['nets'][3]
{'name': 'zero', 'id': 3}
```

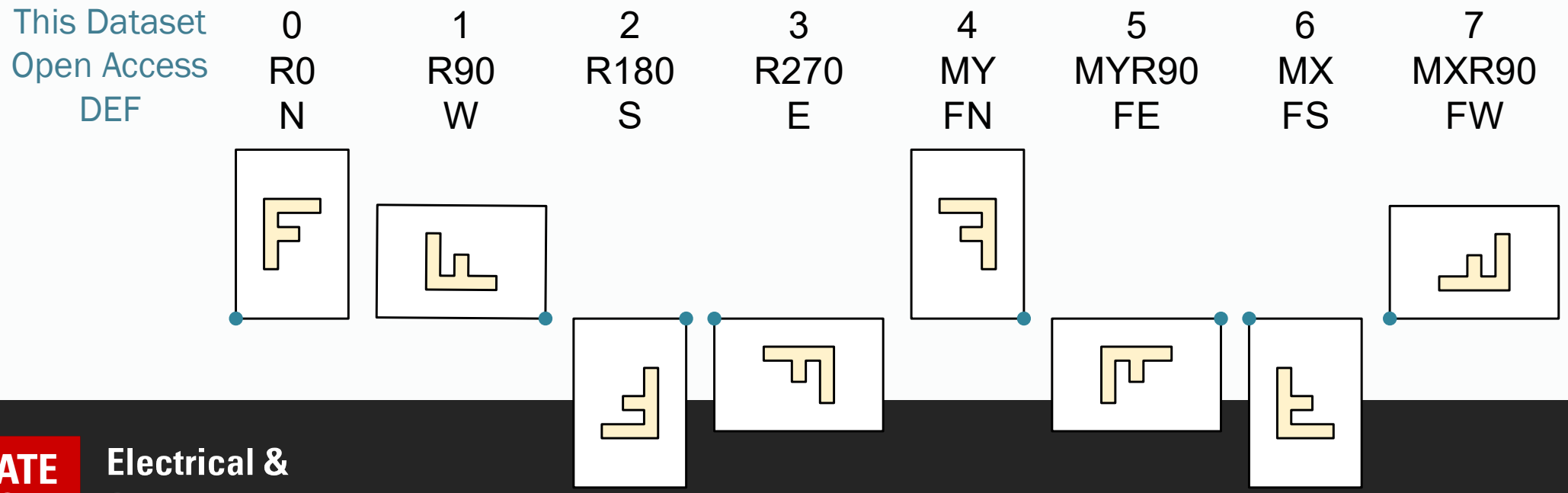
Design Feature Data (Instances)

- The JSON object in [design].json.gz also contains the ‘instances’ member, which is an array of instance objects
- Each instance object has the following members:
  - name (str) – name of the instance
  - id (int) – index of the instance in the array
  - cell (int) – Master library cell ID (array index)
  - xloc, yloc (int) – location of the instance in database units (DBU)
    - divide this by the DBUtoUU divisor (*e.g.* 1000 or 2000, from README) to get user-units (UU), *i.e.* microns
  - orient (int) – orientation of the instance

```
>>> design['instances'][6]
{'name': 'U22', 'id': 6, 'xloc': 6784, 'yloc': 7680, 'cell': 7, 'orient': 6}
```

Orientations

- Orientation of each instance within a design is given by an integer as shown below (with OpenAccess and DEF orientations for comparison)
- Blue dot indicates the location (xloc & yloc) for each orientation
- Note that DEF always restricts the location to lower-left corner (unlike the Open Access location, which is used in our data-set)



Cell Library Data

- Cell library data provided in cells.json.gz
- Contains an array of JSON objects with the following members
  - name (str) – name of the cell / module
  - id (int) – index of the cell in the array
  - width (int) – width of the cell in database units (divide by DBUtoUU to get microns)
  - height (int) – height of the cell in database units (divide by DBUtoUU to get microns)
  - terms – an array of terminal objects

```
>>> import json, gzip
>>> with gzip.open('cells.json.gz','rb') as f:
...   cells = json.loads(f.read().decode('utf-8'))
>>> cells[7]
{'name': 'AOI21_X1', 'id': 7, 'width': 768, 'height': 1536, 'terms': [{'name': 'A1', ...
```

Terminal Objects

- Terminal objects contain the following members
  - name (str) – name of the terminal
  - id (int) – integer index of terminal as it appears in the incidence matrix (same as array index plus 1)
  - dir (int) – direction of the terminal
    - 0 – input
    - 1 – output
    - 2 – inout
  - xloc, yloc (int) – location of the terminal in database units (DBU)
    - divide by DBUtoUU to get user-units (UU), *i.e.* microns

```
>>> cells[7]['terms'][2]
{'name': 'B', 'id': 3, 'dir': 0, 'xloc': 1920, 'yloc': 1565}
```

Incidence Matrix Data

- The file [design]_connectivity.npz contains NumPy arrays that can be used to define SpiPy sparse matrices in coordinate (COO) format
- COO matrices can be quickly constructed and easily converted into other matrices
- Each non-zero entry in the matrix gives the terminal index
- Remember to subtract 1 to get the index for each cell's terminal array
- Note that the matrix will contain duplicate indices if a net connects multiple terminals on one instance

```
>>> import numpy as np
>>> from scipy.sparse import coo_matrix
>>> conn=np.load('counter_connectivity.npz')
>>> coo = coo_matrix((conn['data'], (conn['row'], conn['col'])), shape=conn['shape'])
>>> coo.getrow(6).getcol(3).toarray()
array([[3]])
>>> cells[design['instances'][6]['cell']]['terms'][3-1]
{'name': 'B', 'id': 3, 'dir': 1}
```

Example Incidence Matrix

Nets

Terminal Indices

```
>>> coo.toarray()
array([[1, 0, 0, 0, 0, 0, 3, 0, 0, 0, 0, 0, 2, 4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 3, 0, 2, 0, 0, 0, 0, 6, 0, 5, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 2, 3, 0, 6, 0, 0, 0, 5, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 6, 0, 0, 5, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 3, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0],
       [0, 0, 0, 3, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 3, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 4, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 3, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 3, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0],
       [0, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 1, 0, 4, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3, 0, 0, 0, 0, 2, 0, 0, 0],
       [0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
       [0, 3, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 4, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 2, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 1],
       [0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0],
       [0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 2, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2]])
```

Instances

- Incidence matrix for counter design shown

NumPy Congestion Data

```
import numpy as np
data=np.load('counter_congestion.npz')
lyr=list(data['layerList']).index('M1')
ybl=data['yBoundaryList']
xbl=data['xBoundaryList']
i,j=getGRCIndex(xloc,yloc,xbl,ybl)
data['demand'][lyr][i][j]
data['capacity'][lyr][i][j]
```

- read NumPy file
- get the index for layer M1
- get the array of y-coordinates for GRC left-edge boundaries
- get the array of x-coordinates for GRC bottom-edge boundaries
- get the array indices for the GRC containing location (xloc,yloc)
- get the demand for layer M1 at location (xloc,yloc)
- get the capacity for layer M1 at location (xloc,yloc)

- Routers report congestion information for each Global Route Cell (GRC) using the following values
  - Demand – number of routing tracks needed in the GRC
  - Capacity – number of possible routing tracks in the GRC (this number does not vary with demand)
- GRCs are organized in a rectangular grid, but not all GRCs have the same dimensions.
  - We use the x|yBoundaryList arrays to keep track of the GRC boundaries. This helps convert locations to array indices.
  - All x,y dimensions are in database units (i.e. divide by DBUtoUU from README to get user units, *i.e.* microns)

getGRCIndex Function

```
def getGRCIndex(x,y,xbl,ybl):
    j=0
    for b in xbl[1:]:
        if x<b:
            break
        j+=1
    i=0
    for b in ybl[1:]:
        if y<b:
            break
        i+=1
    return i,j
```

- Routers report congestion information for each Global Route Cell (GRC).
- The getGRCIndex() function used decode GRC indices from x,y coordinates (shown on the previous slide) can be defined as shown on this slide.
- Note that the **demand** and **capacity** arrays are organized with x-coordinates increasing in rows (*i.e.* last array index) and y-coordinates increasing in columns (*i.e.* next-to-last array index). Therefore, in the reference data['demand'][lyr][i][j], i references the y-location, while j references the x-location.

Faster getGRCIndex Function

```
def buildBST(array,start=0,finish=-1):
    if finish<0:
        finish = len(array)
    mid = (start + finish) // 2
    if mid-start==1:
        ltl=start
    else:
        ltl=buildBST(array,start,mid)
    if finish-mid==1:
        gtl=mid
    else:
        gtl=buildBST(array,mid,finish)
    return((array[mid],ltl,gtl))

xbst=buildBST(data['xBoundaryList'])
ybst=buildBST(data['yBoundaryList'])
```

```
def getGRCIndex(x, y, xbst, ybst):
    while (type(xbst)==tuple):
        if x < xbst[0]:
            xbst=xbst[1]
        else:
            xbst=xbst[2]
    while (type(ybst)==tuple):
        if y < ybst[0]:
            ybst=ybst[1]
        else:
            ybst=ybst[2]
    return ybst, xbst
```

- If finding the GRC index limits performance, then building a binary search tree can help
- This getGRCIndex function runs about 170X faster