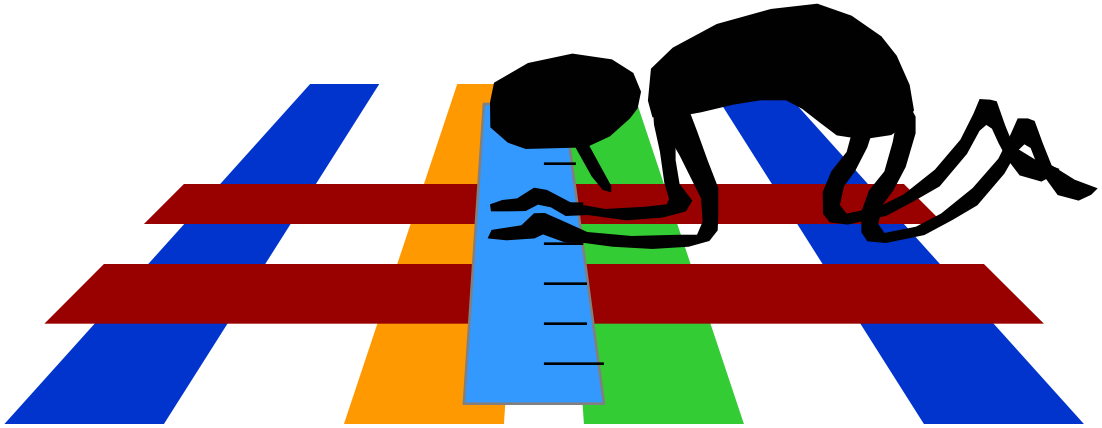
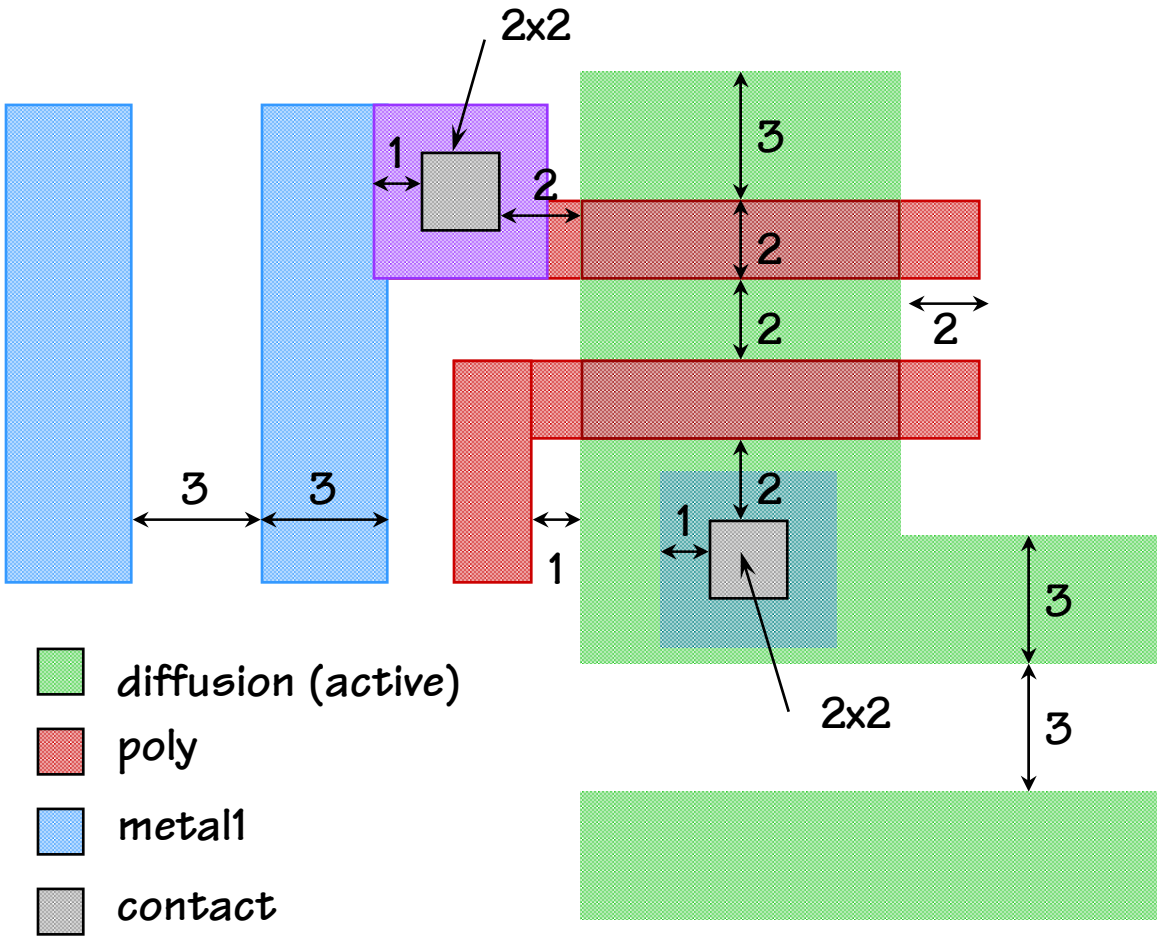
CMOS Layout

Measure twice, fab once



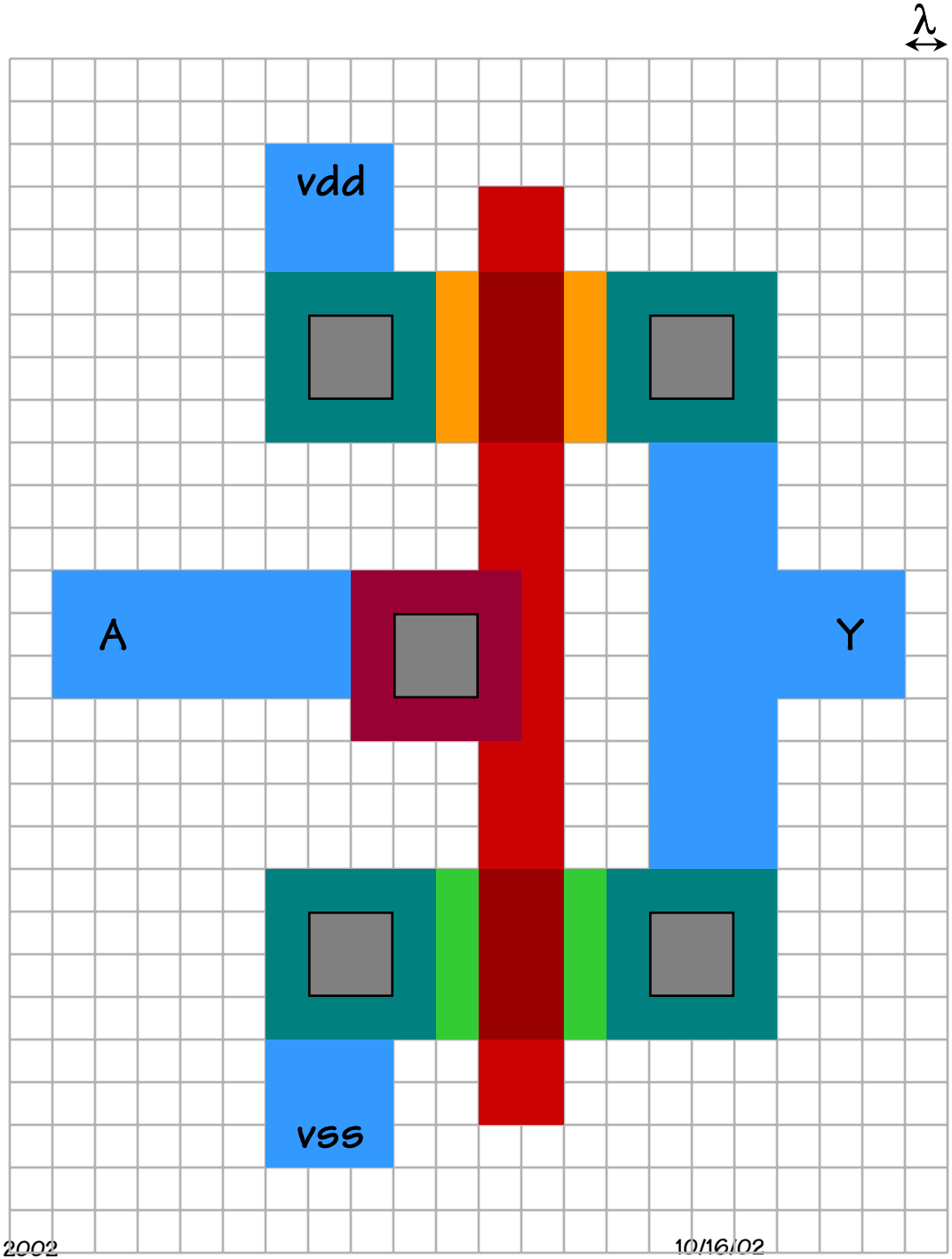
Lambda-based design rules

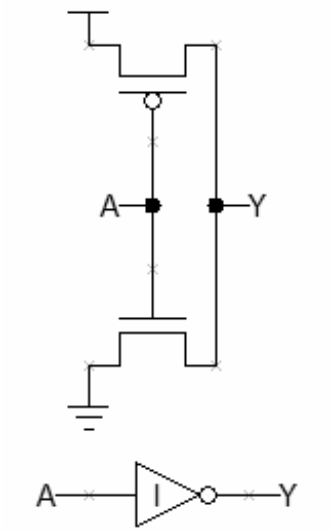
One lambda = one half of the "minimum" mask dimension, typically the length of a transistor channel. Usually all edges must be "on grid", e.g., in the MOSIS scalable rules, all edges must be on a lambda grid.



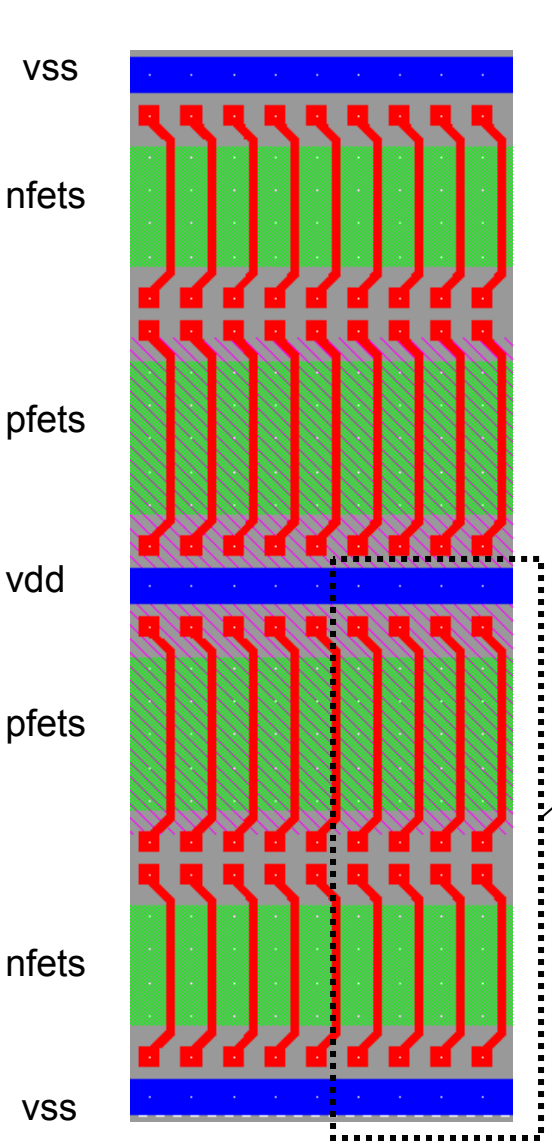
More info at: http://www.mosis.org/Technical/Designrules/scmos/scmos-main.html

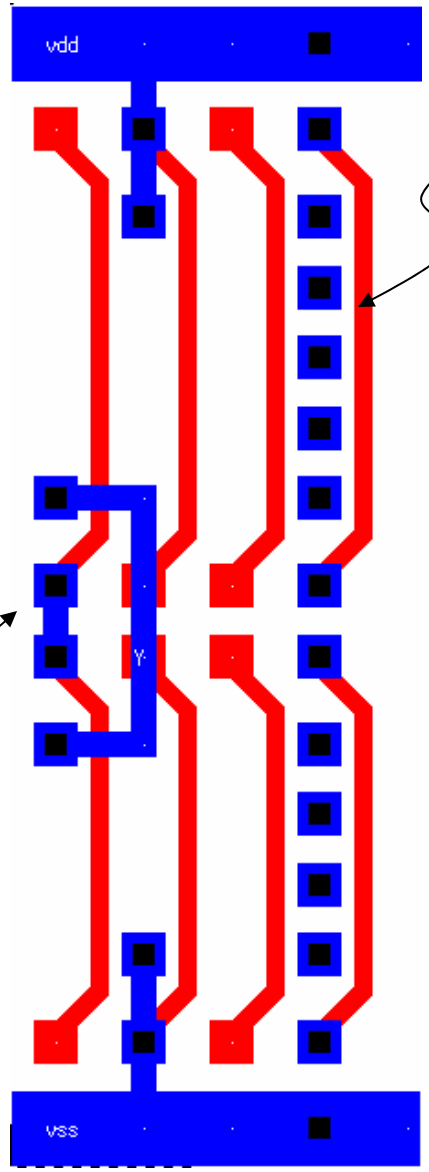
Sample "Lambda" Layout



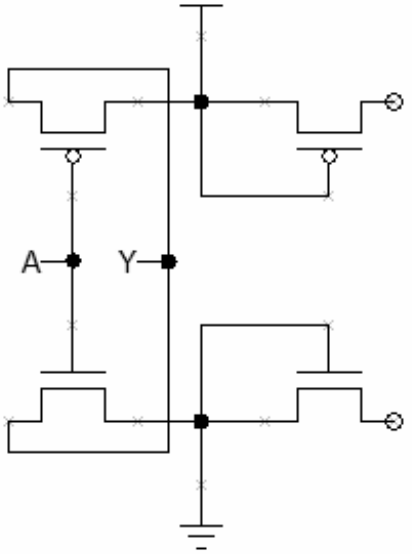


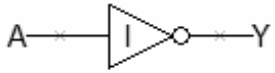
Sample Sea-of-Gates Layout





Column showing all possible contact locations

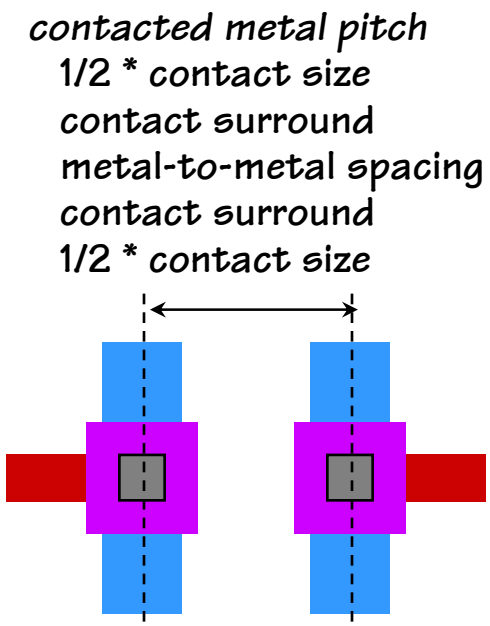




Lamba vs. Micron rules

Lambda-based design rules are based on the assumption that one can scale a design to the appropriate size before manufacture. The assumuption is that all manufacturing dimensions scale equally, an assumption that "works" only over some modest span of time. For example: if a design is completed with a poly width of 2λ and a metal width of 3λ then minimum width metal wires will always be 50% wider than minimum width poly wires.

Consider the following data from *Weste*, Table 3.2:



lambda rule	lambda = 0.5u	micron rule
1λ	0.5μ	0.375μ
1λ	0.5μ	0.5μ
3λ	1.5μ	1.0μ
1λ	0.5μ	0.5μ
1λ	0.5μ	0.375μ
8λ	4μ	2.75μ

Scaled design is legal but much larger than it needs to be!

Retargetable Layouts

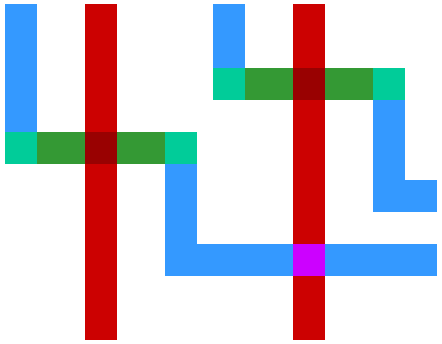
So, should one use lambda rules, or not?

- probably okay for retargeting between "similar" processes, e.g., when later process is a simple "shrink" of the earlier process. This often happens between generations as a mid-life kicker for a process. Some 0.35u processes are shrinks of an earlier 0.5u process. Can be useful for "fabless" semiconductor companies.

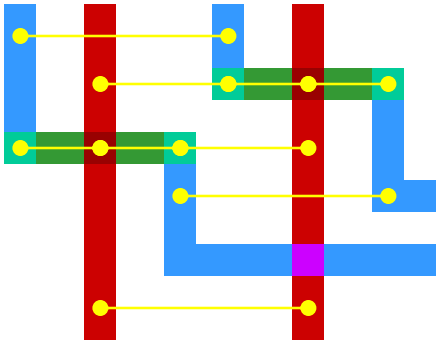
- most industrial designs use micron rules to get the extra space efficiency. Cost of retargeting by hand is acceptable for a successful product, but usually it's time for a redesign anyway.

- invent some way of entering a design symbolically but use a more sophisticated technique for producing the masks for a particular process. Insight: relative sizes may change but topological relationship between components does not. So, instead of shrinking a design, *compact* it! LED offers compaction for leaf cells...

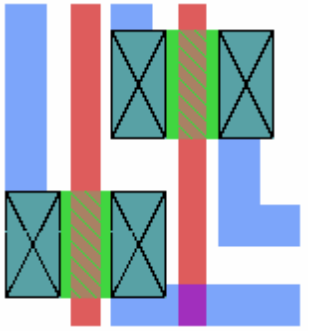
Sticks and Compaction



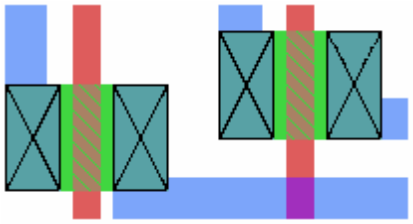
Stick diagram



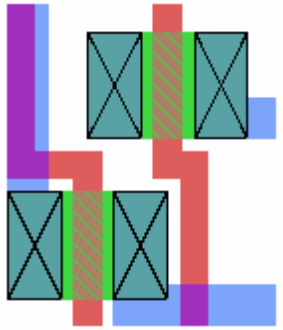
Horizontal constraints for compaction in X



Compact X then Y

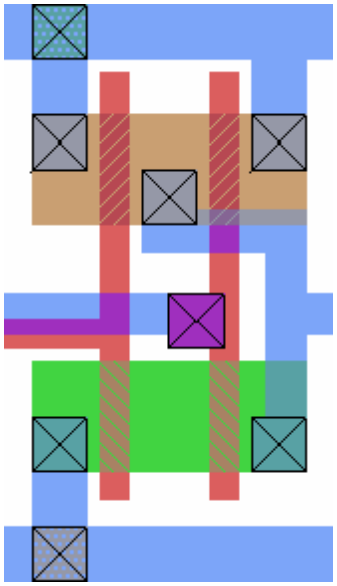


Compact Y then X



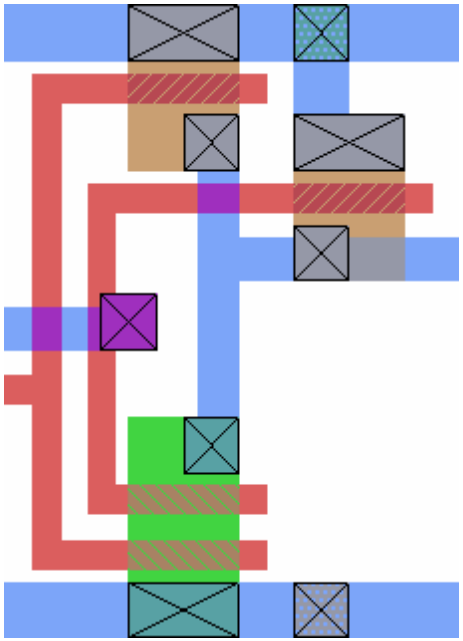
Compact X with jog insertion, then Y

Choosing a "style"



Vertical Gates

Good for circuits where fets sizes are similar and each gate has limited fanout. Best choice for multiple input static gates and for datapaths.



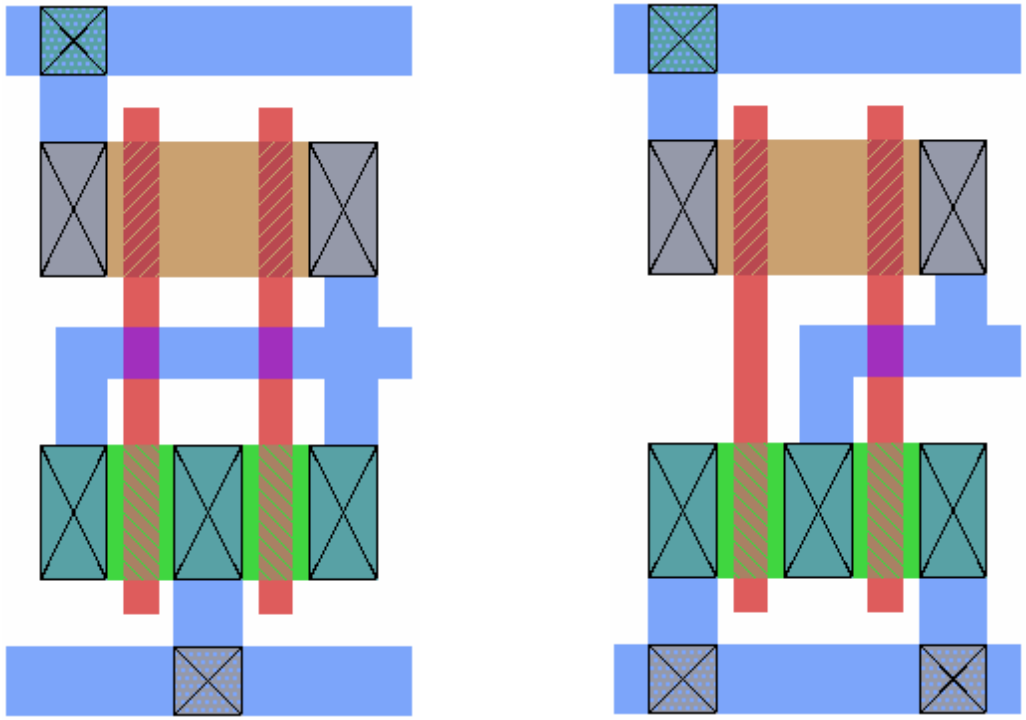
Horizontal Gates

Good for circuits where long and short fets are needed or where nodes must control many fets. Often used in multiple-output complex gates (e.g, sum/carry circuits).

What about routing signals between gates? Note that both layouts block metal/poly routing inside the cell. Choices: metal2 routing over the cell or routing above/below the cell.

- avoid long (> 50 squares) poly runs
- don't "capture" white space in a cell
- don't obsess over the layout, instead make a second pass, optimizing where it counts

Optimizing connections

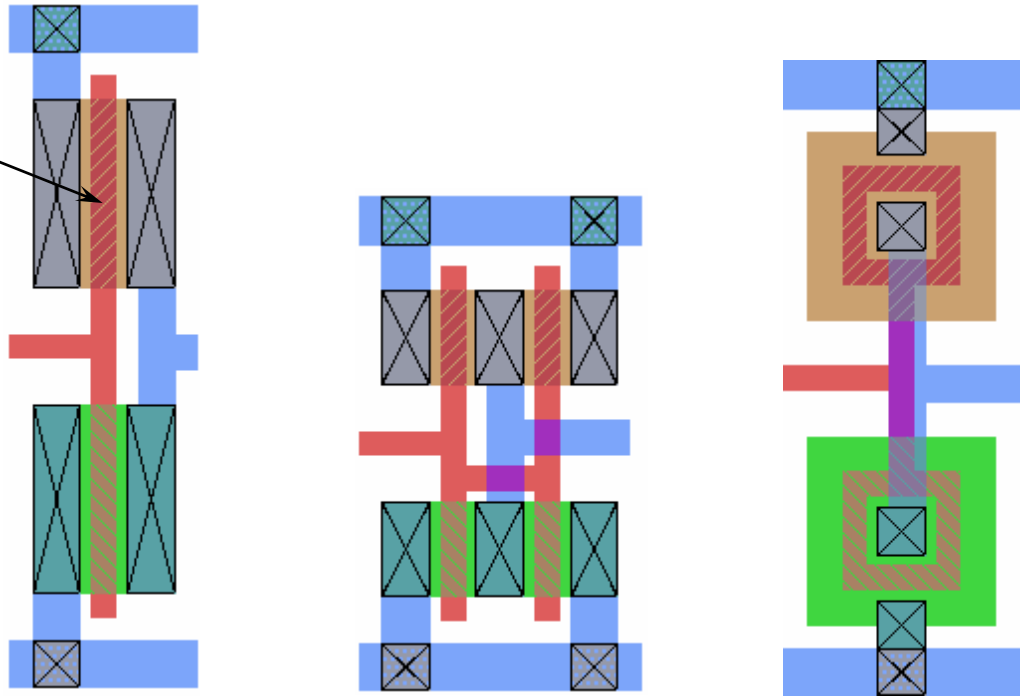


Which is the better gate layout?

- considering node capacitances?
- considering "composibility" with neighboring gates?

Big vs. Parallel

can't make gates too long because of poly resistance! Eventually really large transistors have to broken into smaller transistors in wired in parallel.

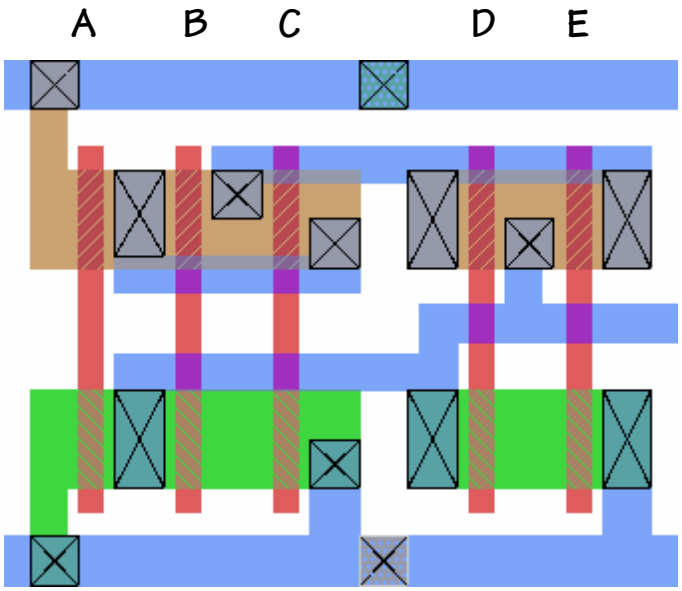


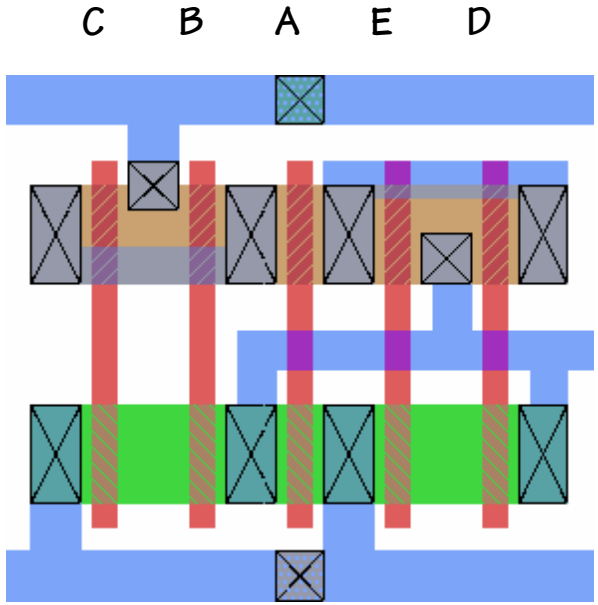
area = 928 area = 1008 area = 1080

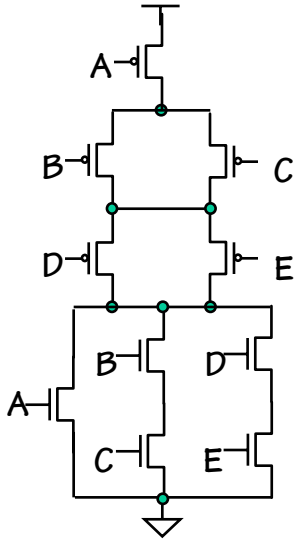
Which is the better gate layout?

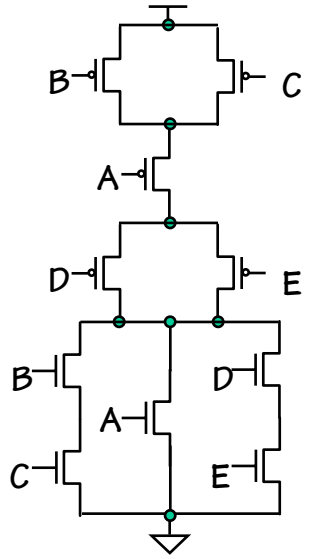
- considering node capacitances?
- considering "composibility" with neighboring gates?

Eliminating Gaps

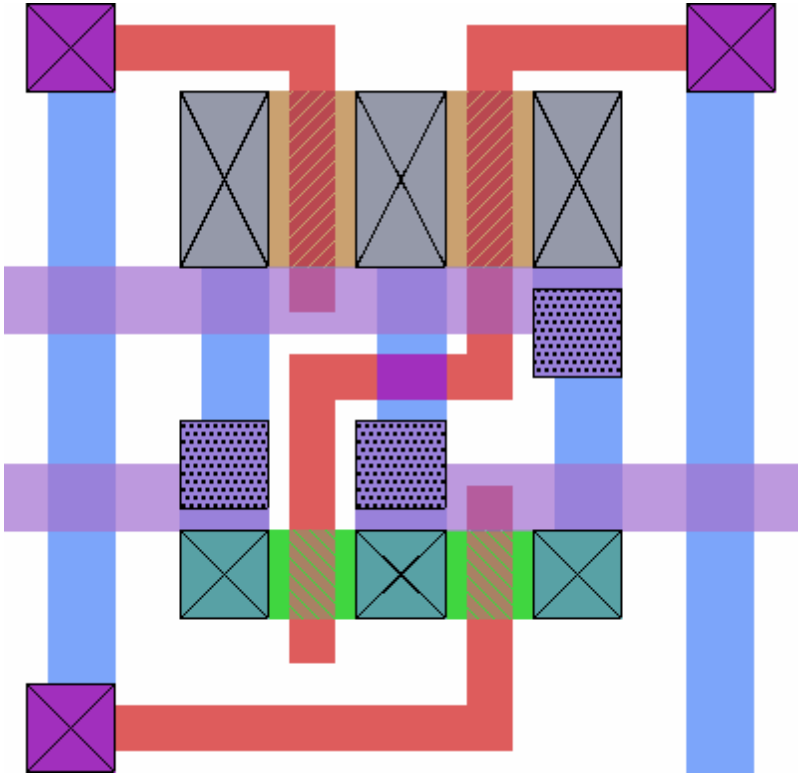








Replicating Cells

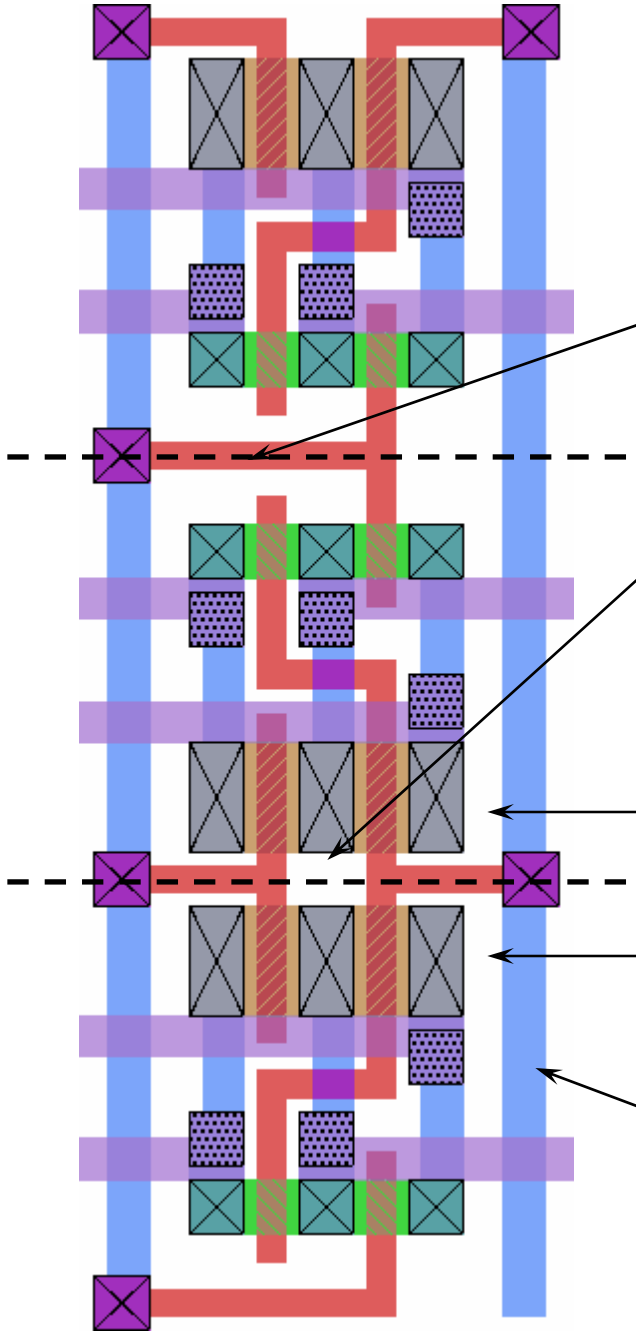


What does this cell do?

What if we want to replicate this cell vertically, i.e., make a stack of the cells, to process many bits in parallel?

- what nodes are shared among the cells?
- what nodes aren't shared?
- how should we arrange the cells vertically?

Vertical Replication



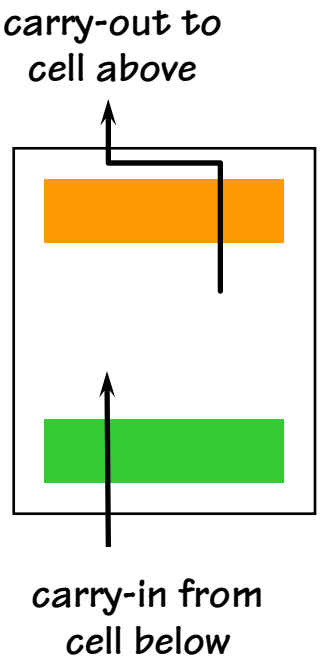
Place shared geometry symmetrically about shared boundary.

Place items that aren't to be shared 1/2 minspacing rule from shared boundary.

Reflect cell about X axis so that Pfets are next to each other: this avoids large ndiff/pdiff spacing.

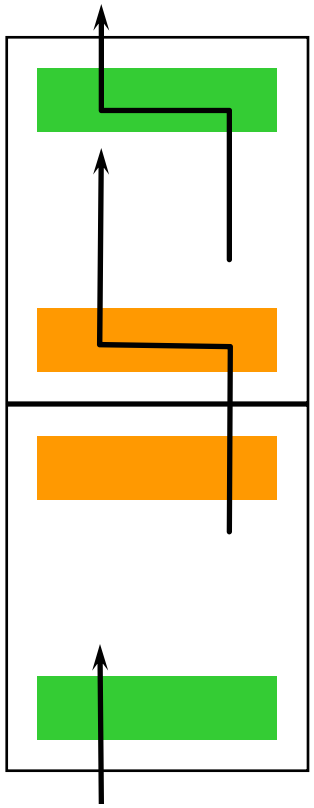
Run shared control signals vertically -- they'll wire themselves up automatically?

Vertical Intercell Routing



S'pose we have a signal that will run vertically from one cell to the next, e.g., the carry-out from one cell becomes the carry-in for the cell above.

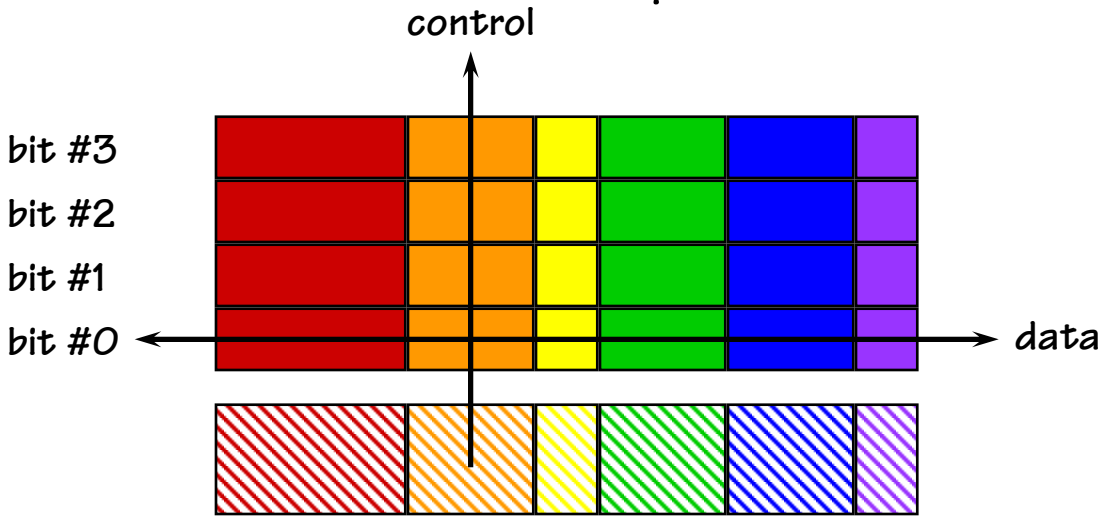
Looks okay until we reflect the cell when we do the vertical replication!



Solution: we have to do the routing for vertical intercell signals for a pair of cells, then replicate the pair (complete with routing) vertically.

Building a Datapath

It's often the case that we want to operate on many bits in parallel. A sensible way to arrange the layout of this sort of logic is as a datapath where data signals run horizontally between functional units and control signals run vertically to all the bits of a particular functional unit:



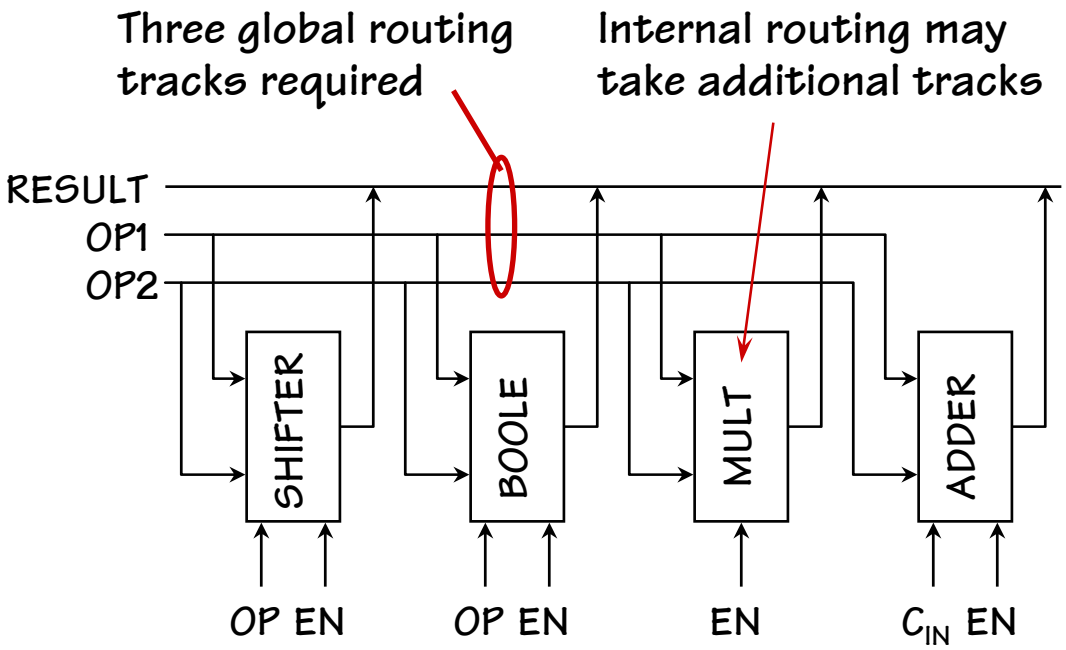
Logic that generates the control signals can be placed at the bottom of the datapath. If control logic is complicated or irregular, it might be placed in a separate standard cell block and only the control signal buffers placed placed just below the datapath. Although it's tempting to run control signals in poly (so they can control fets) this is unwise for tall datapaths because of poly resistance (e.g., 32 bits x 20u/bit = 640u = ~1000 squares = ~20k ohms!)

Datapath Bit Pitch

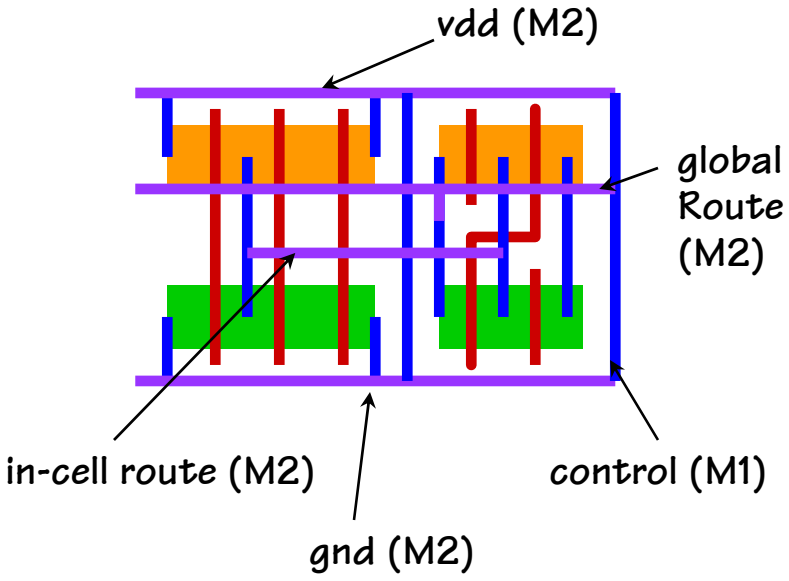
How tall should we make each bit of the datapath? That depends on

- the width of the nfets and pfets
- how much in-cell routing there is
- how much over-the-cell global routing there is

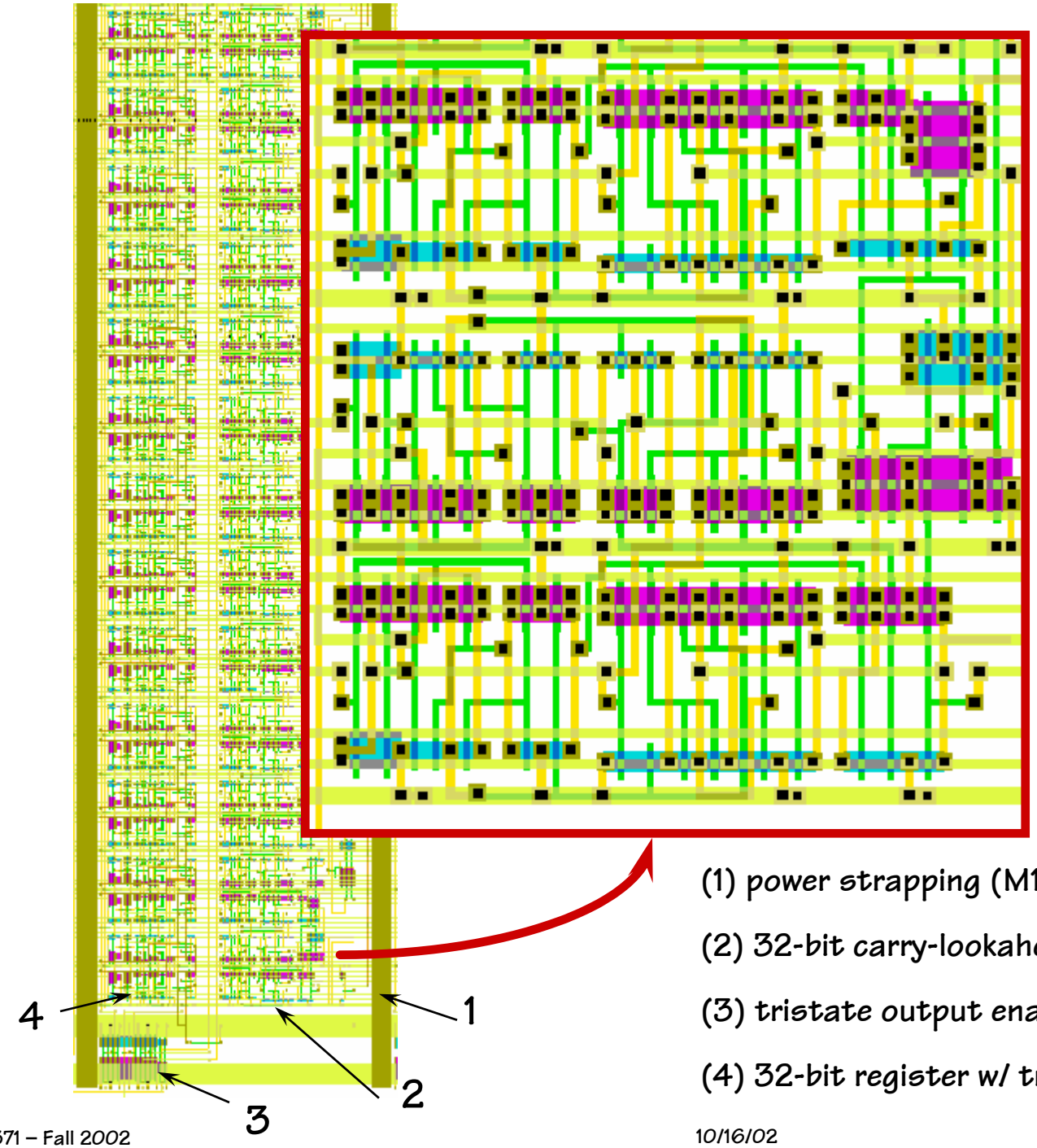
Global routes can be determined from datapath schematic:



Cell routing plan:



Adder Datapath



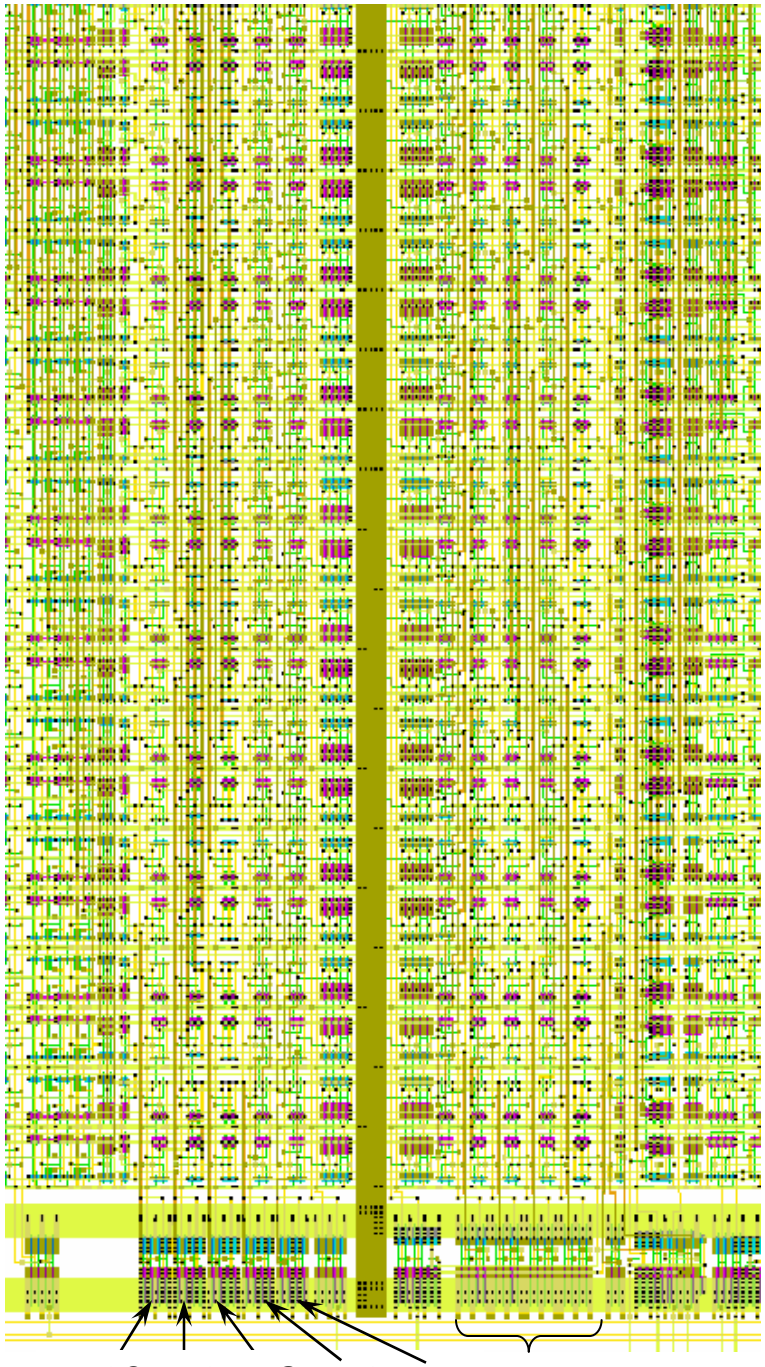
(1) power strapping (M1=GND, M3=VDD)

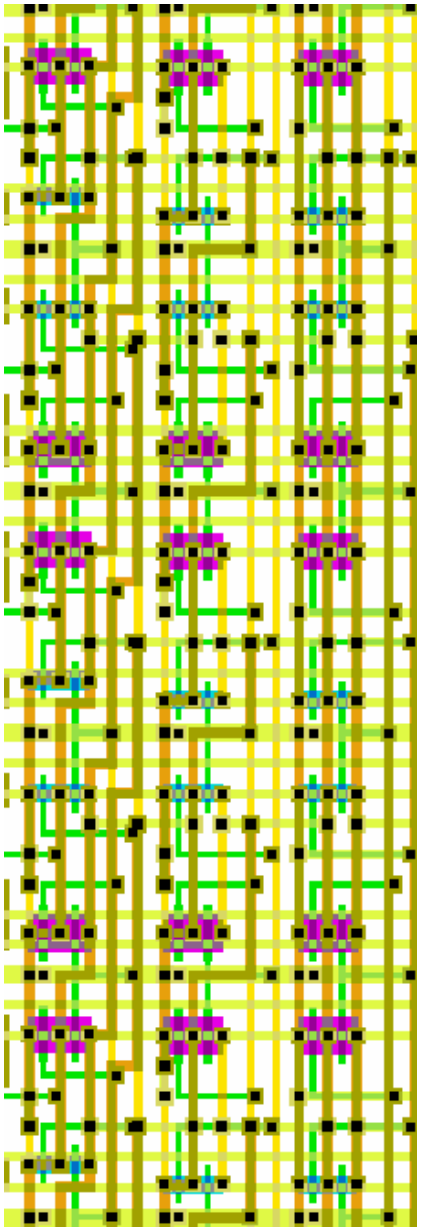
(2) 32-bit carry-lookahead adder

(3) tristate output enable control logic

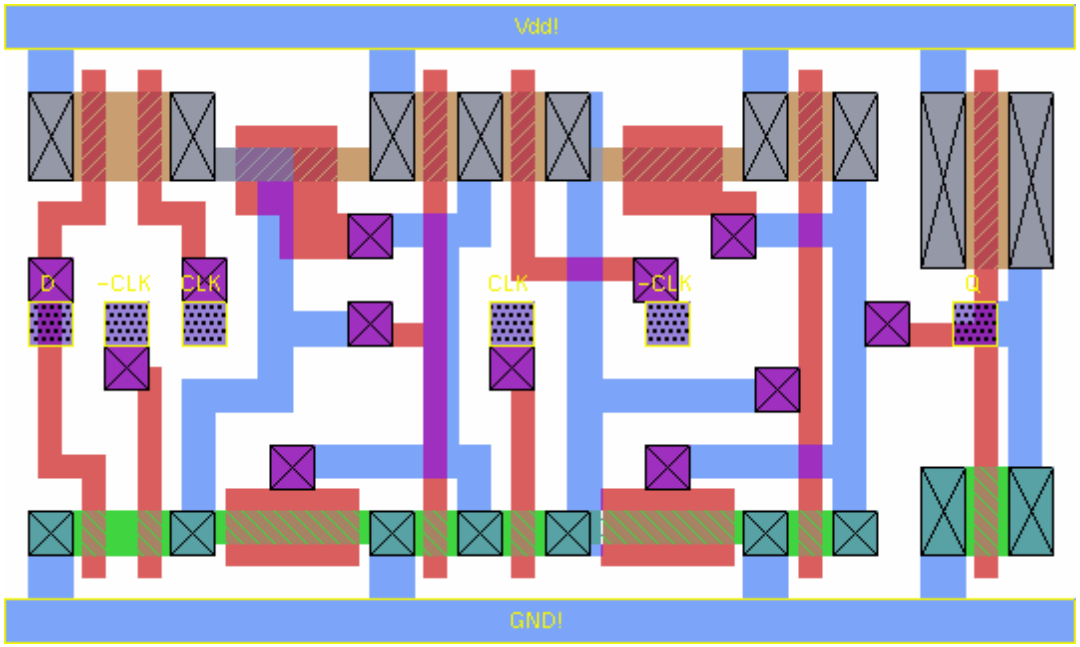
(4) 32-bit register w/ tristate driver

Shifter Datapath



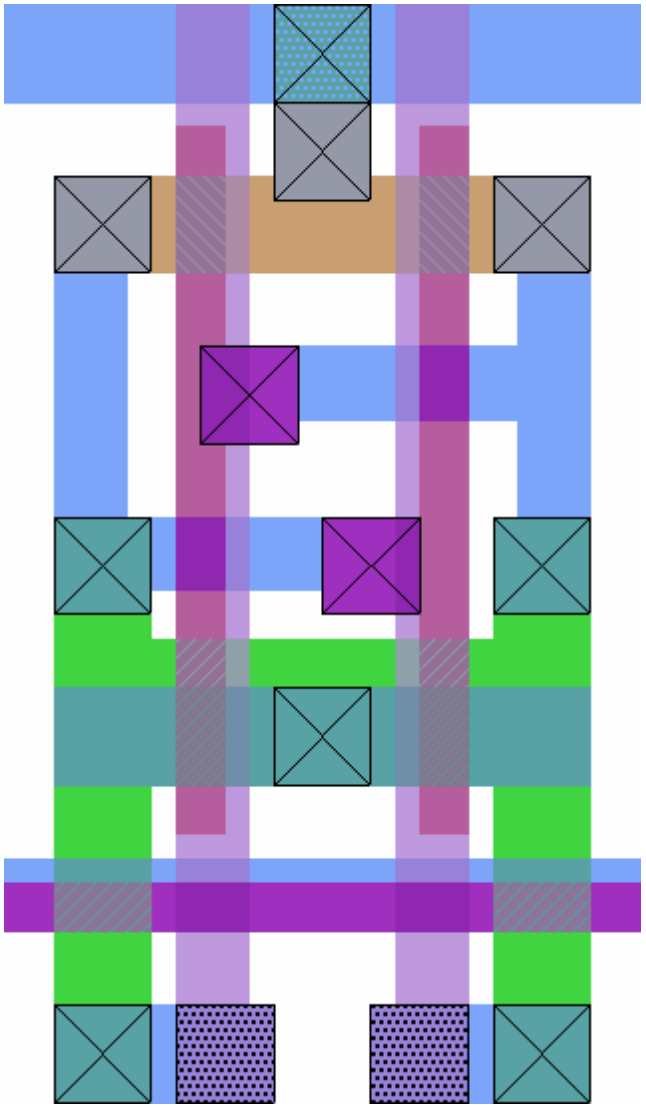


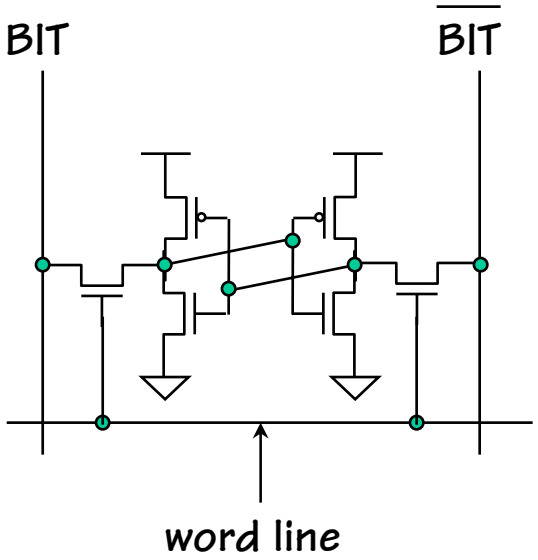
Design for Reuse



- what's this cell do?
- what are the "fat" fets?
- Cell was designed for placement "under" a metal2/metal3 routing grid. How was the layout affected by this design requirement?

Think Globally





- How are neighboring cells placed?
- Isn't the word line a long poly wire?
- Where's the p-substrate contact?

Checking Layouts

Design Rule Checker (DRC). This is a program that checks each piece of the layout against the process design rules. This is a slow process:

- canonicalize layout into a set of leading and trailing non-overlapping mask edges. Some boolean mask operations may be needed.
- determine electrical connectivity and label each edge with the node it belongs to.
- test each edge end point against neighboring edges to check for spacing (leading edges) and width (trailing edges) violations.

Layout vs. Schematic (LVS). First a netlist is extracted from the layout. Use the electrical info generated by the DRC and then recognize transistors are juxtapositions of channel with diffusion. Then see if extracted netlist is isomorphic to the schematic netlist. This is done by a coloring algorithm:

- initialize all nodes to the same color
- compute a new color for each node as some hashing function involving the colors of connected (ie, thru a fet) nodes.
- nodes that have a unique color are isomorphic to similarly colored node in other network
- worry about parallel fets, ambiguous nodes

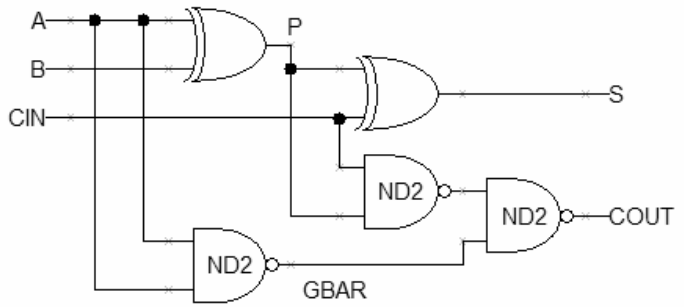
Example Sea-of-Gates Layout

Let's build the layout for an 8-bit ripple carry adders using the Ocean tools. First we have to set up a directory for our project:

```
athena% setup 6.371
Attaching 6.371 ...
Running commands in /mit/6.371/.attachrc ...

6.371% mk6371 rippleadder
------- creating fishbone project rippleadder -------
------- importing primitives -------
------- importing lib6371 -------
------- copying default config files -------

...done. Enjoy your new project "rippleadder" !!!

6.371% cd rippleadder
6.371%
```

Enter SLS netlists

```
#include "sls_prototypes/lib6371.ext"

network fa(terminal a,b,cin,s,cout,vss,vdd)
{
{xgbar}   nand2(a,b,gbar,vss,vdd);
{xp}      xor2(a,b,p,vss,vdd);
{xs}      xor2(p,cin,s,vss,vdd);
{xc1}     nand2(p,cin,c1,vss,vdd);
{xcout}   nand2(c1,gbar,cout,vss,vdd);
}
```




```
network adder8(terminal a[0..7],b[0..7],cin,s[0..7],cout,vss,vdd)
{
{xfa0} fa(a[0],b[0],cin,s[0],c0,vss,vdd);
{xfa1} fa(a[1],b[1],c0,s[1],c1,vss,vdd);
{xfa2} fa(a[2],b[2],c1,s[2],c2,vss,vdd);
{xfa3} fa(a[3],b[3],c2,s[3],c3,vss,vdd);
{xfa4} fa(a[4],b[4],c3,s[4],c4,vss,vdd);
{xfa5} fa(a[5],b[5],c4,s[5],c5,vss,vdd);
{xfa6} fa(a[6],b[6],c5,s[6],c6,vss,vdd);
{xfa7} fa(a[7],b[7],c6,s[7],cout,vss,vdd);
}
```

See /mit/6.371/examples/adder8.sls

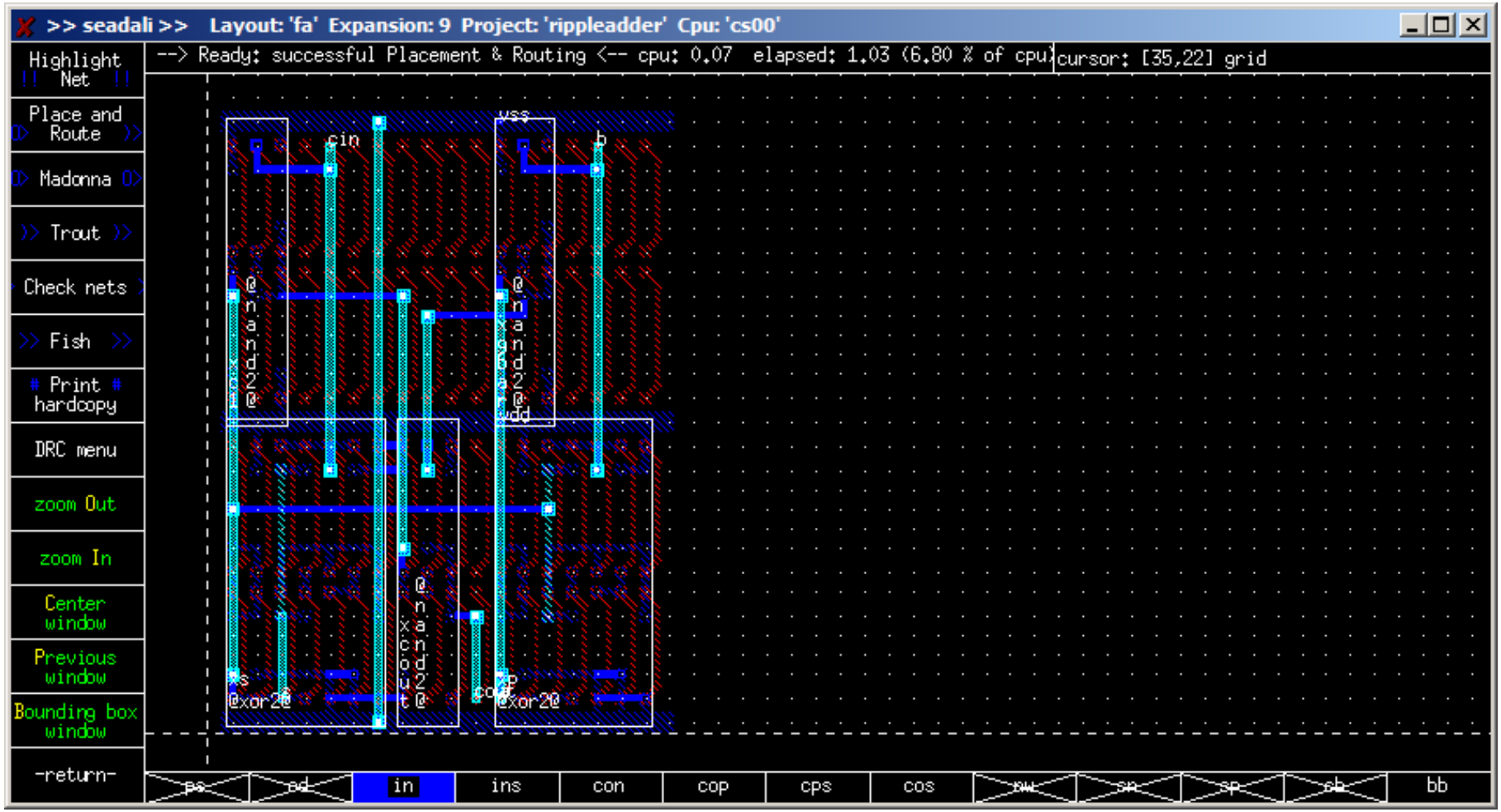
Fire up SEADALI…

```
6.371% csls adder8.sls
File cells.sls:
Parsing network: fa
Parsing network: adder8
6.371% seadali&
```

← Ignore complaint about "Cannot allocate 3 bitplanes"

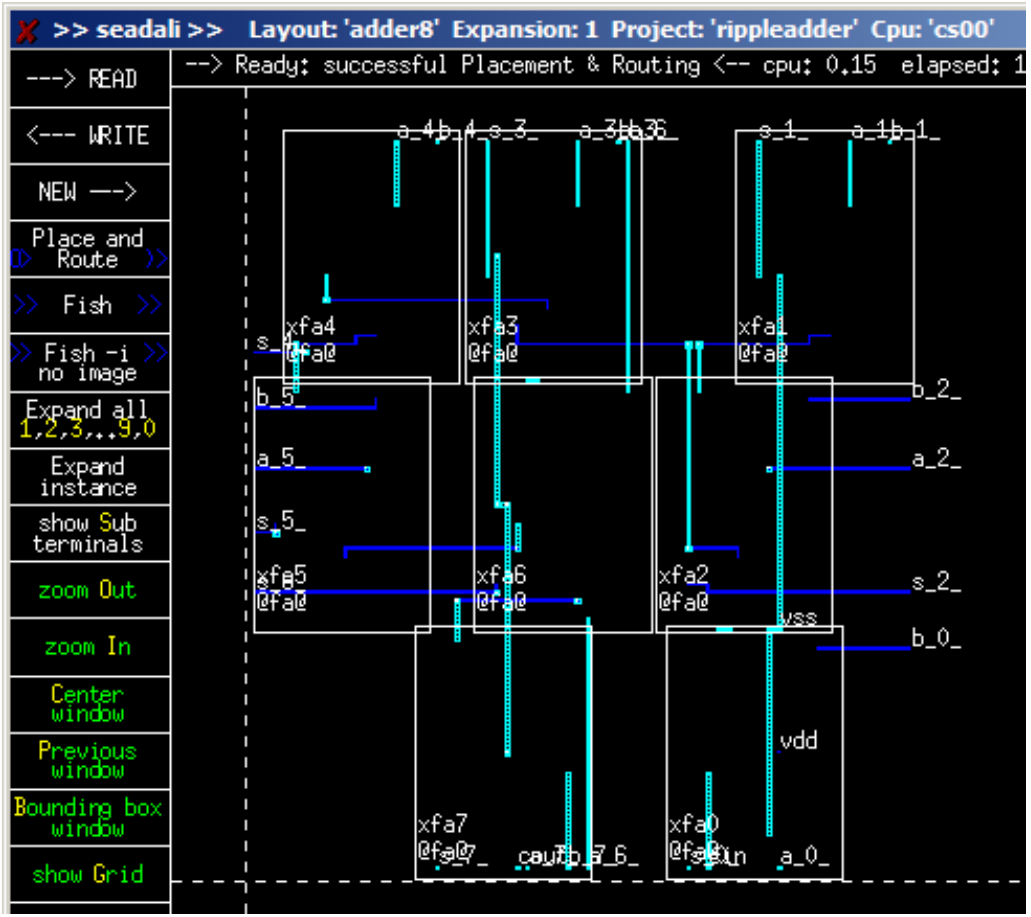
Try automated tools…

[1] click "automatic tools"
[2] click "Place and Route"
[3] Enter circuit: type "fa"
[4] click "** DO IT! **"



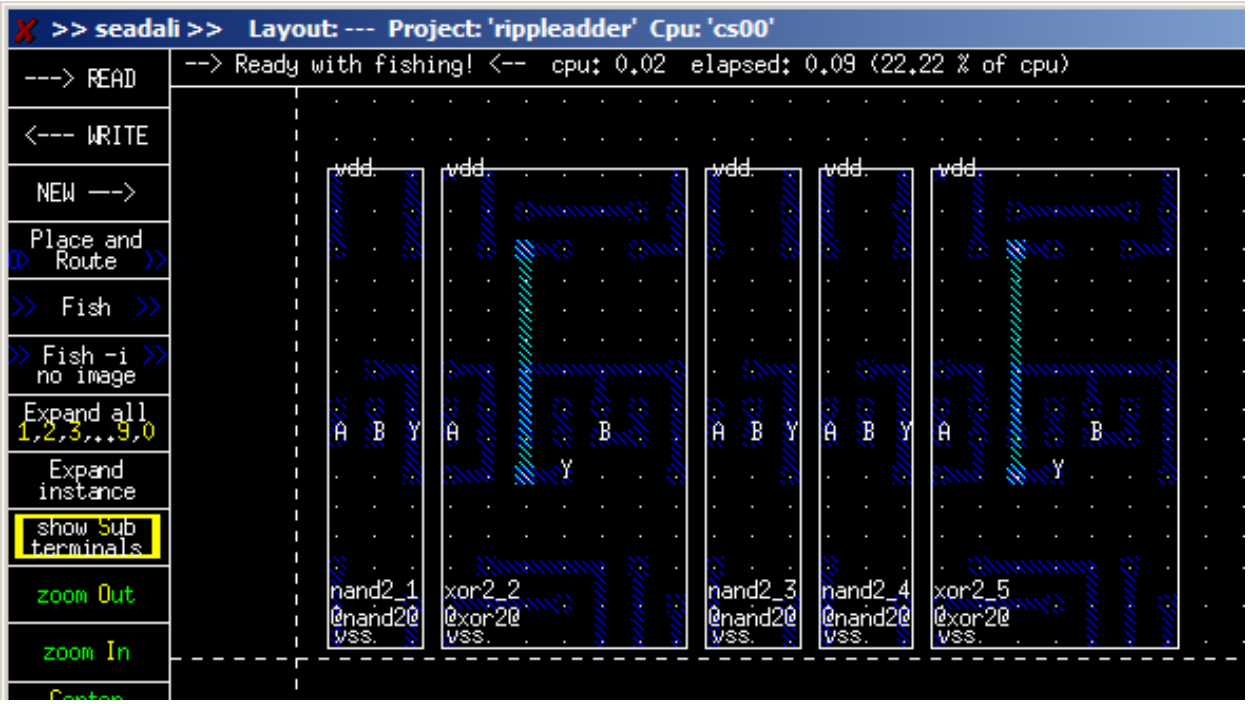
Save layout in database and layout ADDER8

[1] click "-return-"
[2] click "database"
[3] click "<--- WRITE"
[4] click "fa"
[5] click "Place and Route"
[6] type "adder8"
[7] click "** DO IT! **"
[8] click "<--- WRITE"
[9] click "adder8"



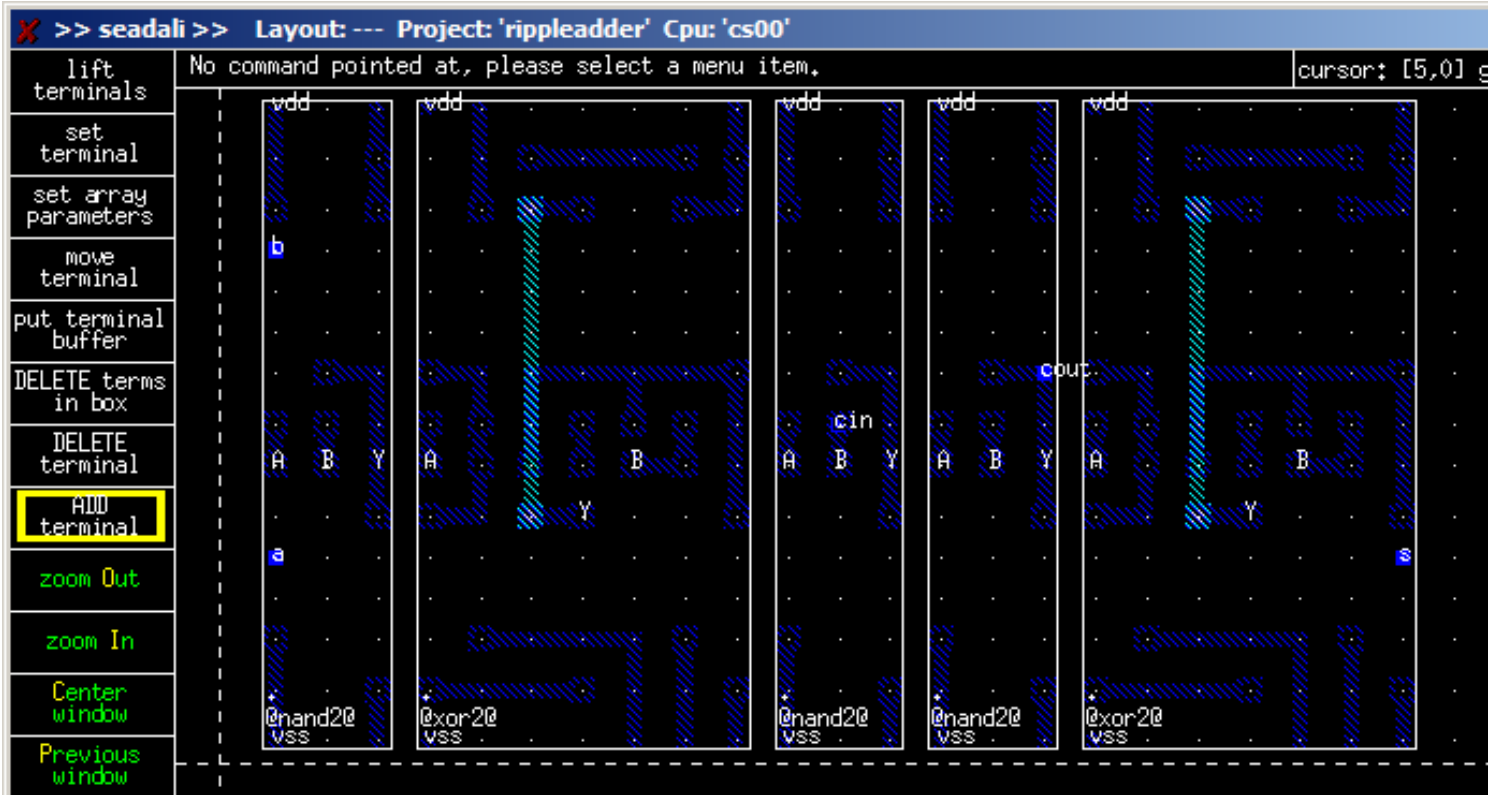
Now try manual layout

[1] click "NEW --->"
[2] click "-return-"
[3] click "instances"
[4] click "ADD imported instance"
[5] click "nand2", click in main screen to place
[6] type "o" several times to zoom out, "c" to recenter screen
[7] click "next", "xor2", click to place next to NAND2
[8] continue, adding 2 more NAND2 and last XOR2
[9] when done, click "-return-",
[10] type "S" to show sub terminals
[11] type "2" show interior routing



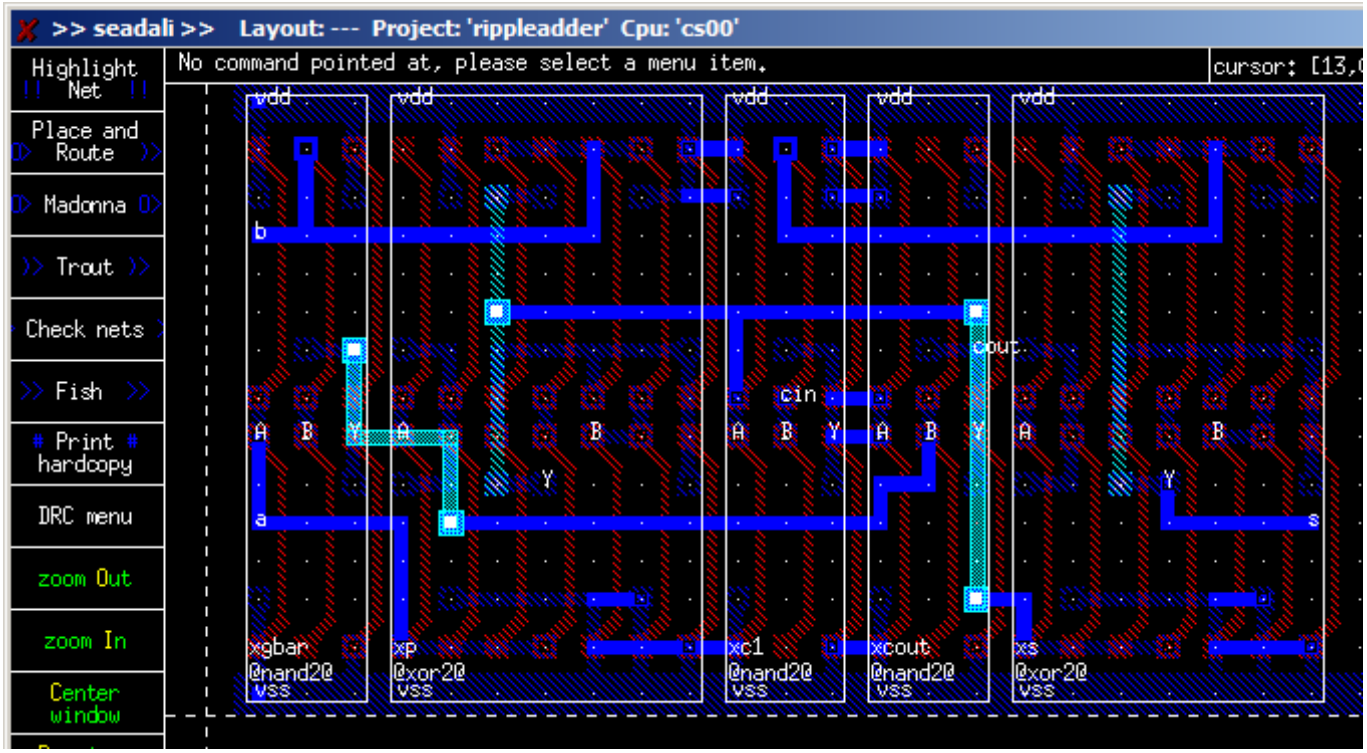
Prepare for autoroute

[1] click "-return-"
[2] click "terminals"
[3] click "ADD terminal"
[4] use cursor to select grid point for terminal "a"
[5] terminal name: type "a" (capitalization counts!)
[6] position terminal "b" on left edge
[7] position terminal "s" on right edge
[8] position terminal "cout" on Y output of rightmost NAND2
[9] position terminal "cin" on B input of middle NAND2



Autoroute

[1] click "-return-"
[2] click "automatic tools"
[3] click ">> Trout >>"
[4] enter circuit name: type "fa",return
[5] click "border terminals" to deselect
[6] click "** DO IT! **"
[7] dismiss various pop up screens
[8] return to database screen, save layout for FA

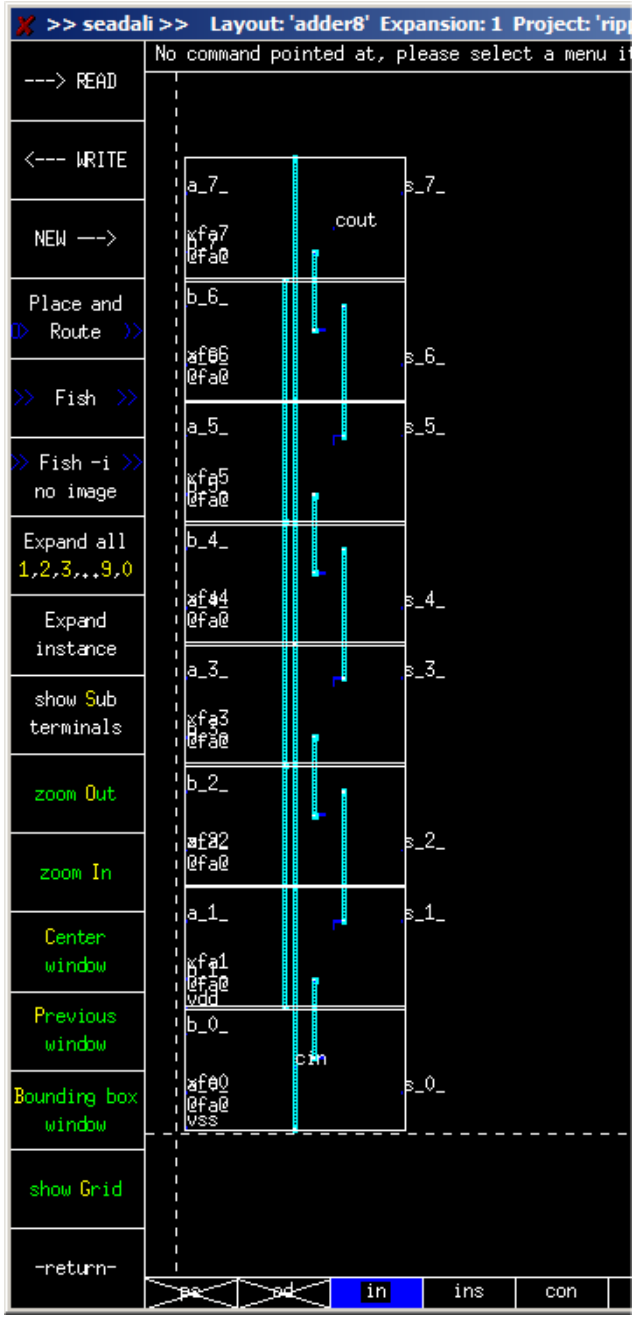


Place and Route ADDER8

[1] start a "NEW --->" layout
[2] use instance tools to
    make a stack of FAs
[3] use "set instance name"
    to name them xfa0..xfa7
[4] use ">> Trout >>" on automated
    tools screen to complete the
    route for ADDER8
[5] save layout in database

Morals from this story:

- you have to help tools find the structure
- control randomness of tools
  - place instances based on global plan
  - place terminals based on global plan
  - automated routing okay
  - automated placement for random logic



Prepare Simulation Stimulus

See /mit/6.371/examples/adder8.cmd
and /mit/6.371/examples/Adder8.cmd

```
plot a[0..7], b[0..7], cin, s[0..7], cout
option level = 3
option simperiod = 4
option sigunit = 50.000000e-09
option outacc = 10p
/*
*%
tstep 0.2n
trise 0.5n
tfall 0.5n
*%
*%
.options cptime=500
*%
*/
set vdd = h*~
set vss = l*~
```

```
set a[0] = l*1 h*1 h*1 h*1
set a[1] = l*1 h*1 h*1 h*1
set a[2] = l*1 h*1 h*1 h*1
set a[3] = l*1 h*1 h*1 h*1
set a[4] = l*1 h*1 h*1 h*1
set a[5] = l*1 h*1 h*1 h*1
set a[6] = l*1 h*1 h*1 h*1
set a[7] = l*1 h*1 h*1 h*1
set b[0] = l*1 l*1 l*1 l*1
set b[1] = l*1 l*1 l*1 l*1
set b[2] = l*1 l*1 l*1 l*1
set b[3] = l*1 l*1 l*1 l*1
set b[4] = l*1 l*1 l*1 l*1
set b[5] = l*1 l*1 l*1 l*1
set b[6] = l*1 l*1 l*1 l*1
set b[7] = l*1 l*1 l*1 l*1
set cin =  l*1 l*1 h*1 l*1
```

Extract and Run Simulation

```
6.371% cp /mit/6.371/examples/*8.cmd .
6.371% space -c fa; ghoti Fa
6.371% space -c adder8; ghoti Adder8
6.371% simeye&
```

[1] select Simulate->Prepare
[2] Circuit: Adder8,
[3] Stimuli: Adder8.cmd
[4] Type: sls-timing
[5] click "Run"