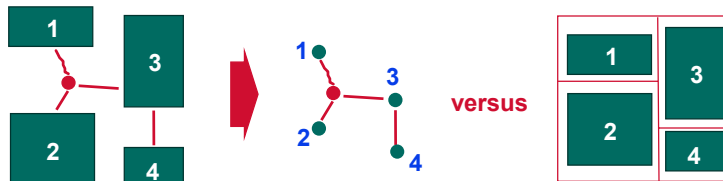


(Lec 16) ASIC Layout: Floorplanning (for Proj3)

▼ What you know...

- ▶ Placement method for objects with “little” shape variation
- ▶ ...eg, placement methods which model objects as points



▼ What you don't know...

- ▶ Methods to do placement for objects with widely varying shapes
- ▶ Methods to deal with objects with different or malleable shape
- ▶ Placement versus floorplanning applications

© R. Rutenbar 2001

CMU 18-760, Fall01 1

Copyright Notice

© Rob A. Rutenbar 2001

All rights reserved.

You may not make copies of this material in any form without my express permission.

© R. Rutenbar 2001

CMU 18-760, Fall01 2

Where Are We?

Physical design--how to wire the placed gates...?

	M	T	W	Th	F	
Aug	27	28	29	30	31	1
Sep	3	4	5	6	7	2
	10	11	12	13	14	3
	17	18	19	20	21	4
	24	25	26	27	28	5
Oct	1	2	3	4	5	6
	8	9	10	11	12	7
	15	16	17	18	19	8
	22	23	24	25	26	9
	29	30	31	1	2	10
Nov	5	6	7	8	9	11
	12	13	14	15	16	12
Thnxgive	19	20	21	22	23	13
	26	27	28	29	30	14
Dec	3	4	5	6	7	15
	10	11	12	13	14	16

Introduction
 Advanced Boolean algebra
 JAVA Review
 Formal verification
 2-Level logic synthesis
 Multi-level logic synthesis
 Technology mapping
 Placement
 Routing
Floorplanning (Project 3)
 Static timing analysis
 Electrical timing analysis
 Geometric data structs & apps

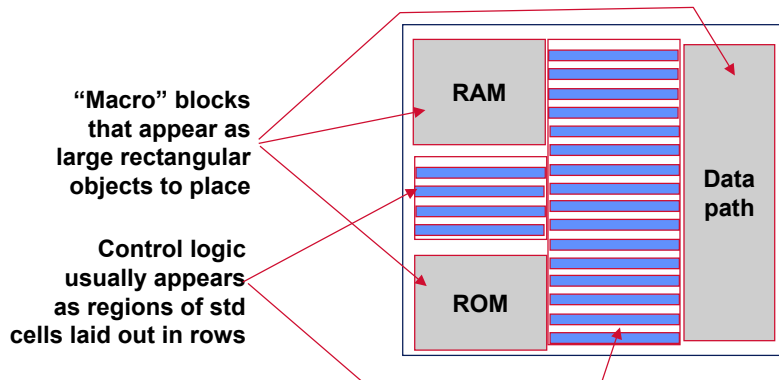
© R. Rutenbar 2001

CMU 18-760, Fall01 3

Dealing with Shapes

Big chips are not just lot of rows of standard cells

- ▶ There can be large blocks (eg, memories, registers, predesigned IP)
- ▶ Also, big chips are done hierarchically, so there can be regions of flat cells and regions with large blocks



© R. Rutenbar 2001

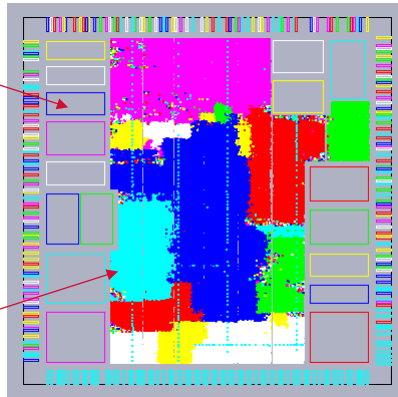
CMU 18-760, Fall01 4

Floorplan Example

▼ Rectangles on this chip are floorplanned regions

Some of these (the grey ones) are pre-designed blocks that we are just placing on this chip...

Some of these (the colored ones) are blocks with standard cells placed inside. In this example, the cells have been allowed to “diffuse” out of these regions to get a better overall placement



Floorplan pic
courtesy L. Pileggi,
Monterey Design

© R. Rutenbar 2001

CMU 18-760, Fall01 5

New Placement Problem: Floorplanning

▼ Dealing with placements of arbitrary rectangular objects

▼ One flavor of this is just like placement, but with large objects

- ▶ Place components with known, fixed (possibly very large) shapes
- ▶ Done in ASICs that mix std cells (random logic) and large functional blocks (memories, pre-design IP)

▼ One very important new twist: object may have flexible shape

- ▶ Place components with variable, maybe unknown shape
 - ▷ May have only an area estimate or lower bound for module
 - ▷ May have a range of allowable shape alternatives
- ▶ Done early in design of very large ASICs and custom ICs, when shape of some modules (eg, a big region of std cells you have not designed yet) are still vague

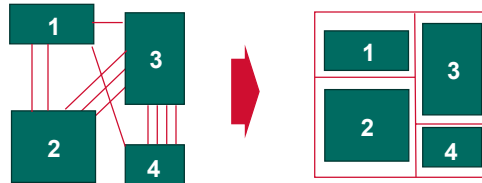
© R. Rutenbar 2001

CMU 18-760, Fall01 6

Basic Floorplanning Placement

Layout scenario

- ▶ You have a set of rectangular placeable objects
- ▶ They have fixed, unvarying size
- ▶ You want to place them to minimize wirelength and overall chip area



Approach

- ▶ We will use simulated annealing to do iterative improvement
- ▶ But we need a much more powerful geometric representation to do all the stuff we really want to do here...

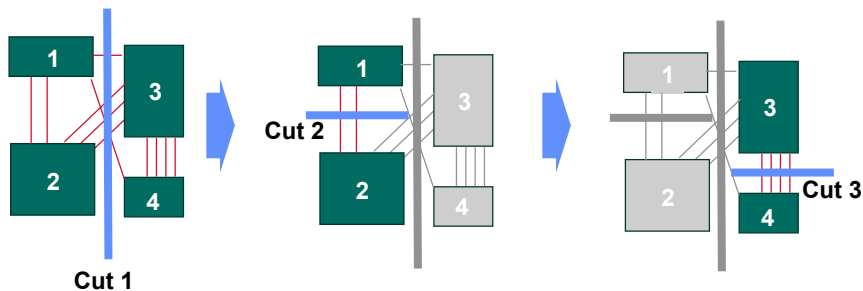
© R. Rutenbar 2001

CMU 18-760, Fall01 7

Remember Recursive Bipartitioning Placement...

It worked for netlists with many small objects...

- ▶ Use your favorite method to partition a netlist to minimize the size of the cut; repeat recursively to get the final relative placement
- ▶ You could actually do that here, too; problem is, you need to know more info about how to really pack the modules you get in each partition



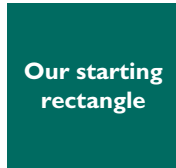
© R. Rutenbar 2001

CMU 18-760, Fall01 8

Related Idea: Slicing Decomposition

▼ New question

- ▶ What layouts can you create by recursively **dissecting a rectangle**?
- ▶ **Dissecting** means “a horizontal or vertical cut all the way across rect”
- ▶ The resulting “rooms” are where you are allowed to put modules



Our starting
rectangle



First cut,
called a “Slice”



2nd cut



3rd cut



4th cut



5th cut, and
we could
keep going...

© R. Rutenbar 2001

CMU 18-760, Fall01 9

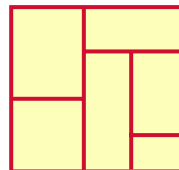
Slicing Decomp. Defines Rooms in a Floorplan

▼ When done slicing, result is a set of floorplan “rooms”

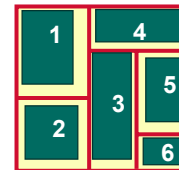
- ▶ The resulting “rooms” are where you are allowed to put modules



Slicing
decomposition



Resulting
rooms in floorplan



Placement assigns
modules to rooms

© R. Rutenbar 2001

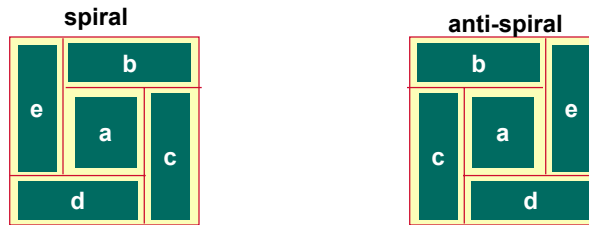
CMU 18-760, Fall01 10

Can You Represent All Floorplans Like This?

▼ Can't you get all possible layouts this way?

► Surprisingly, **NO**

► The canonical example of what you can't get is called a **spiral**; its mirror image is also unrepresentable via slicing, and called an **anti-spiral**



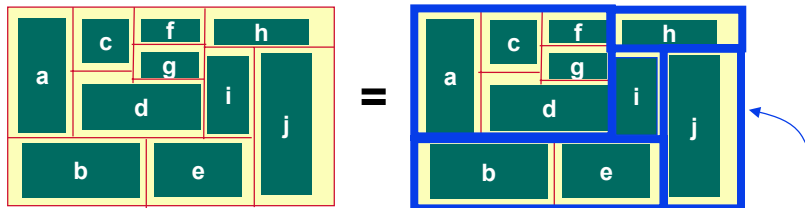
Note – none of these cuts is a “slice” that goes all the way across the entire rectangle. These are “non-slicing” layouts.

© R. Rutenbar 2001

CMU 18-760, Fall01 11

Non-Slicing Layouts

▼ A bigger example of something you cannot get via slicing



Note, it has an anti-spiral in it, though components in the anti-spiral itself are layouts of more than 1 rectangle

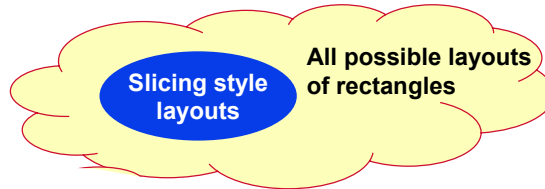
© R. Rutenbar 2001

CMU 18-760, Fall01 12

Nevertheless, Slicing-style Layouts Very Popular

▼ A “slicing style” layout...

- ▶ ..is a subset of all possible floorplans/placements that you can create for a set of rectangles



- ▶ Turns out you don't lose all that much (a few % on area) if you restrict yourself to slicing-style layouts.
- ▶ Turns out you gain a LOT – some very nice data structures and algorithms for placement and floorplanning

© R. Rutenbar 2001

CMU 18-760, Fall01 13

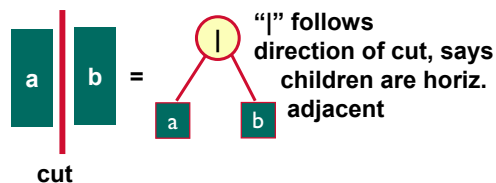
Efficient Data Structure: Slicing Trees

▼ Introduced by Ralph Otten for floorplanning tasks

- ▶ You can represent this slicing decomposition with a single tree data structure: slicing tree

▼ Slicing tree is just a *tree*, and it has 2 kinds of nodes

- ▶ Leaf nodes == placeable modules
- ▶ Internal nodes represent bipartition cuts == slicing cuts
- ▶ 2 kinds of internal nodes: H cuts and V cuts

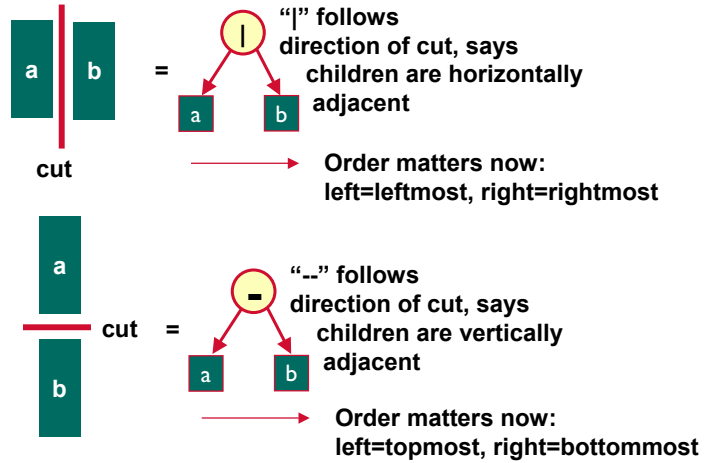


© R. Rutenbar 2001

CMU 18-760, Fall01 14

Slicing Trees

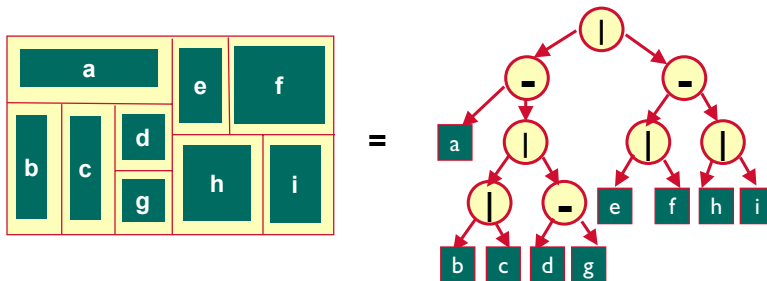
Simplest example of slicing trees, for just 2 objects



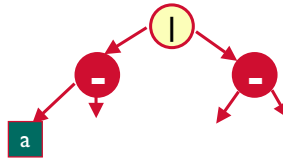
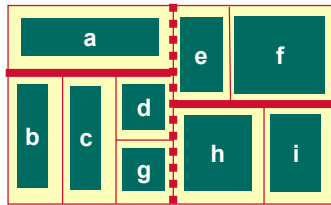
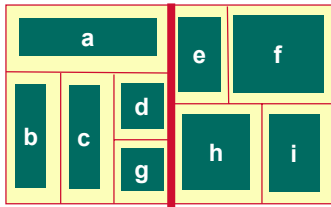
Slicing Trees

Bigger example

Nice feature is you need just 1 structure to capture both H and V info



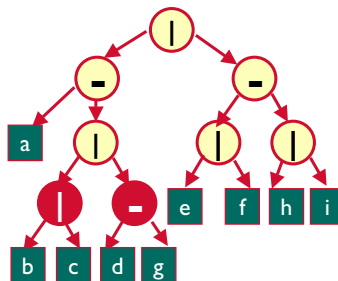
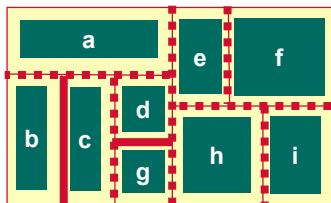
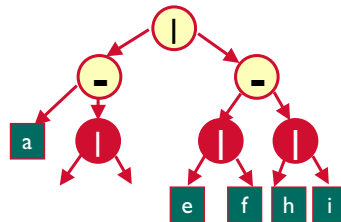
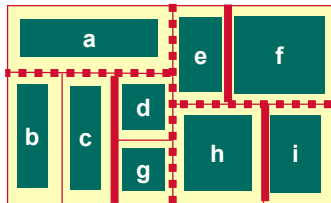
Draw It Incrementally, From Top Down



© R. Rutenbar 2001

CMU 18-760, Fall01 17

Draw It Incrementally, From Top Down



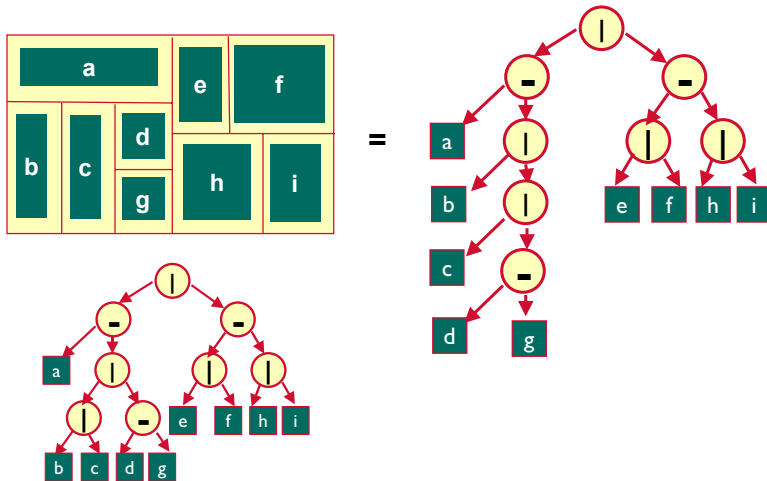
© R. Rutenbar 2001

CMU 18-760, Fall01 18

Slicing Trees

Same example, again

Problem: not unique (yet); here is *another* legal tree for this same layout

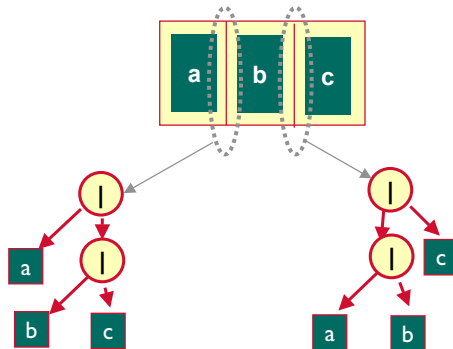


© R. Rutenbar 2001 CMU 18-760, Fall01 19

Slicing Trees: Non-uniqueness

Why does this happen?

Basically, with a binary tree, many ways to represent *parallel* cuts at the same level of hierarchy

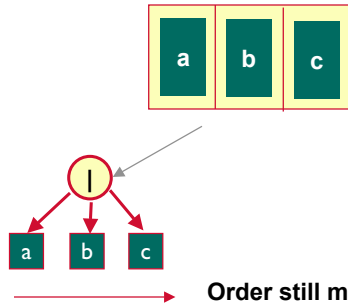


© R. Rutenbar 2001 CMU 18-760, Fall01 20

Slicing Trees: Non-uniqueness

Solution 1:

- ▶ Don't restrict to binary trees
- ▶ Parallel cuts at same level go in tree at the same level
- ▶ (This is sort of a pain – much nicer if we keep it just a binary tree...)



Order still matters:
leftmost (or topmost) child goes first

© R. Rutenbar 2001 CMU 18-760, Fall01 21

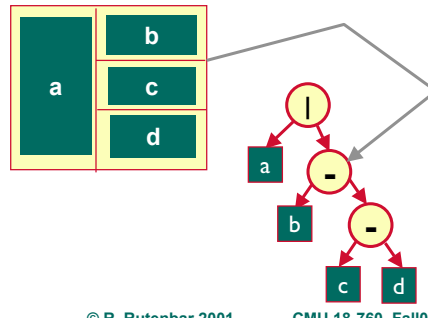
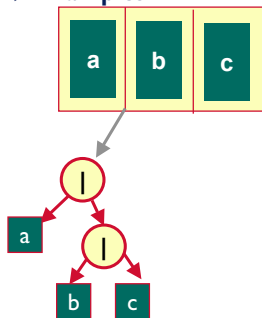
Slicing Trees: Nonuniqueness

Solution 2

- ▶ Keep binary trees, agree on **ORDER** of cuts when building tree



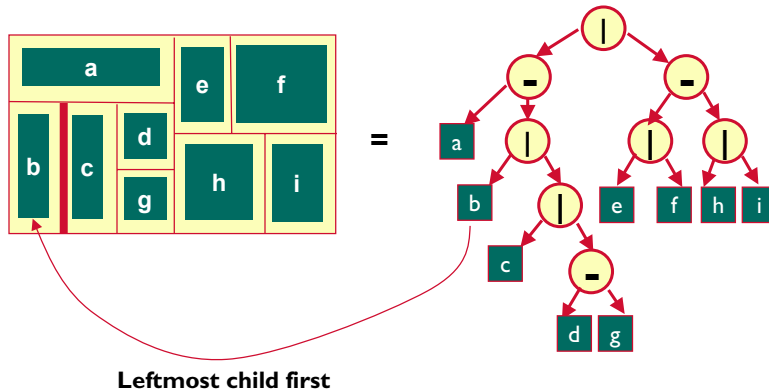
Examples



© R. Rutenbar 2001 CMU 18-760, Fall01 22

Slicing Trees: Canonical Binary Form

▼ So, our tree should look like this in canonical form



© R. Rutenbar 2001

CMU 18-760, Fall01 23

Using Slicing Trees for Placement

▼ Now what?

- ▶ So, how do we do **placement** using these as the core data struct?
- ▶ You can anneal the tree very efficiently

▼ Annealing on a slicing tree: what do we need?

- ▶ State representation: it's just the slicing tree itself
- ▶ Move set: what do we perturb? What changes?
- ▶ Cost function: what do we measure for "goodness"?
- ▶ Cooling schedule: usual, standard stuff will work fine

▼ Most of the "action" is in the move set and cost function...

- ▶ New, very important idea: **topological placement**

© R. Rutenbar 2001

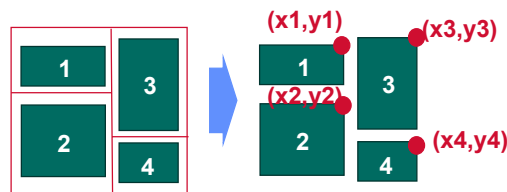
CMU 18-760, Fall01 24

First: Need to Actually Get Module Locations

▼ Note: you don't have them yet

- ▶ Slicing tree only stores a **relative** placement of the objects
- ▶ “Relative” means that we know “relationships” between objects, like “is left of” and “is above”. We don't know real coordinates
- ▶ This is also called a **topological representation**, as distinct from an **absolute representation** of the placement geometry

▼ Need to transform from relative to absolute representation



© R. Rutenbar 2001

CMU 18-760, Fall01 25

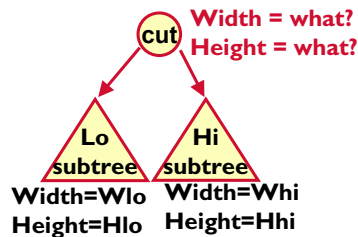
Sizing a Slicing Tree

▼ “Sizing” means “getting real coordinates”

- ▶ It's a recursive algorithm (...are you surprised?)
- ▶ 2 pass algorithm, top-down recursive on tree root

▼ First pass: compute (width, height) of each subtree

- ▶ Easy, once you know (width, height) of your children

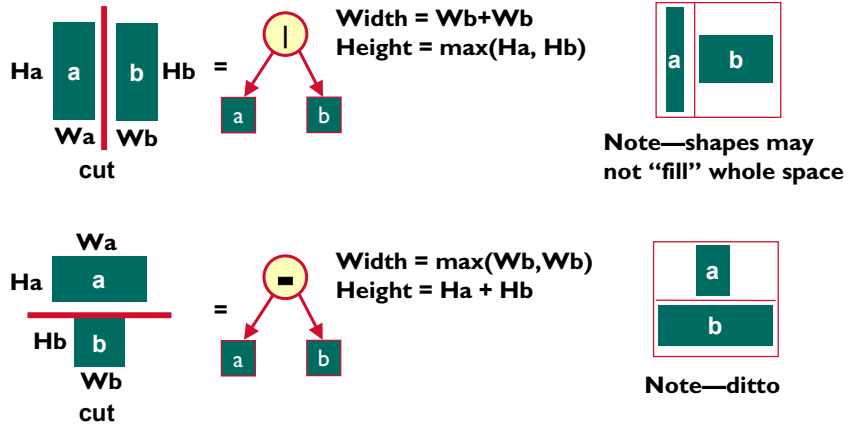


© R. Rutenbar 2001

CMU 18-760, Fall01 26

Sizing Tree Nodes Based on Children...

▼ Easy if your children are leaf modules with fixed sizes



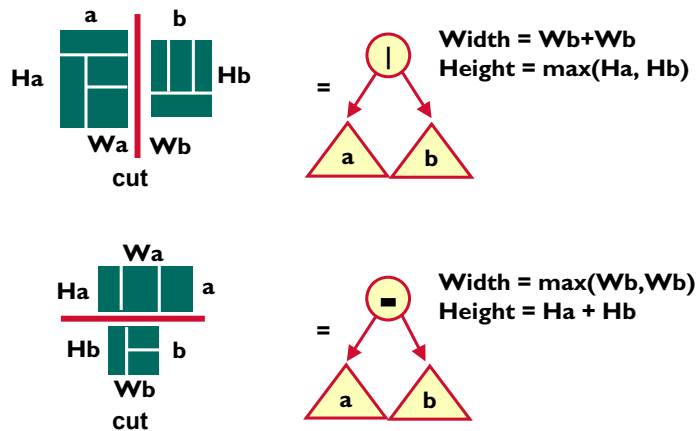
© R. Rutenbar 2001

CMU 18-760, Fall01 27

Sizing Tree Nodes Based on Children...

▼ SAME if children are subtrees with known (width, height)

► So, you compute (width, height) of your children first, recursively



© R. Rutenbar 2001

CMU 18-760, Fall01 28

Sizing a Slicing Tree: First Pass

Algorithm

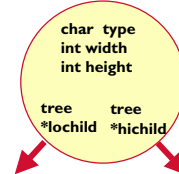
```

Size( slicing tree node T ) {
    // if node T is a leaf node in slicing tree
    if(T.type == leaf node ) {
        T.width = width of leaf module;
        T.height = height of leaf module;
    }

    // size this subtree node T depending on cut type
    else if(T.type == a "|" cut) {
        // compute size, label T "|" cut node
        T.width=lochild(T).width + hichild(T).width;
        T.height=max( lochild(T).height, hichild(T).height);
    }
    else {
        // it's a "-" cut; compute size, label T "-" cut node
        T.width=max( lochild(T).width, hichild(T).width);
        T.height=lochild(T).height + hichild(T).height;
    }

    // that's it
}
    
```

Minimal slicing tree node type



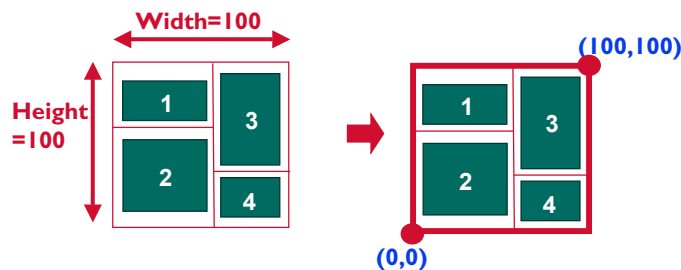
© R. Rutenbar 2001

CMU 18-760, Fall01 29

Sizing a Slicing Tree: Second Pass

Now what?

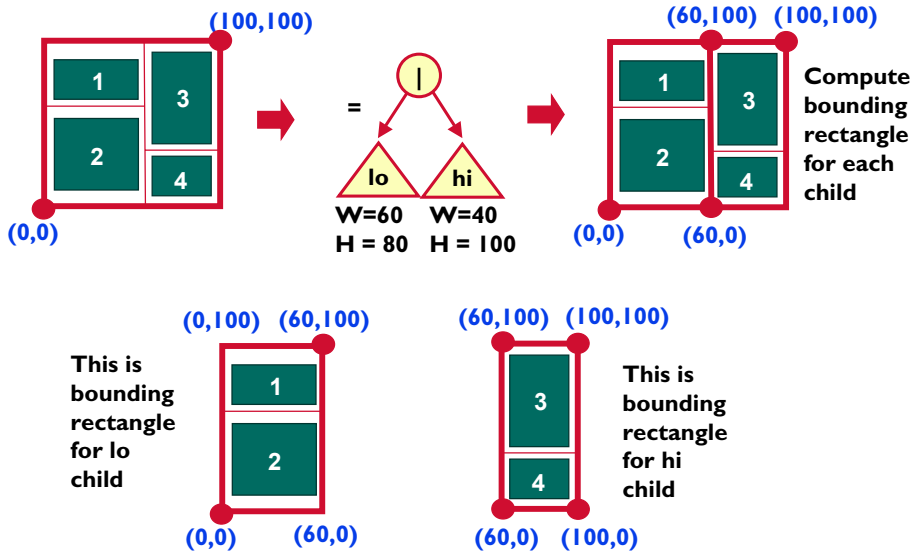
- ▶ You know the (width, height) of each subtree of the slicing tree
- ▶ At the top, this defines a bounding rectangle for the overall layout
- ▶ Pick a coord system for this known rectangle, compute absolute coords
- ▶ Then pass abs coords for the bounding rectangle of each CHILD down



© R. Rutenbar 2001

CMU 18-760, Fall01 30

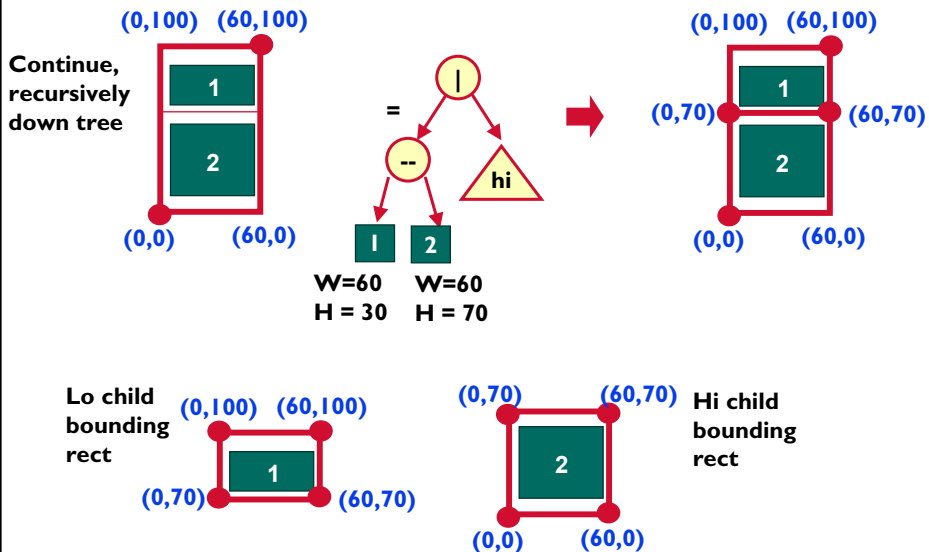
Sizing a Slicing Tree: Second Pass



© R. Rutenbar 2001

CMU 18-760, Fall01 31

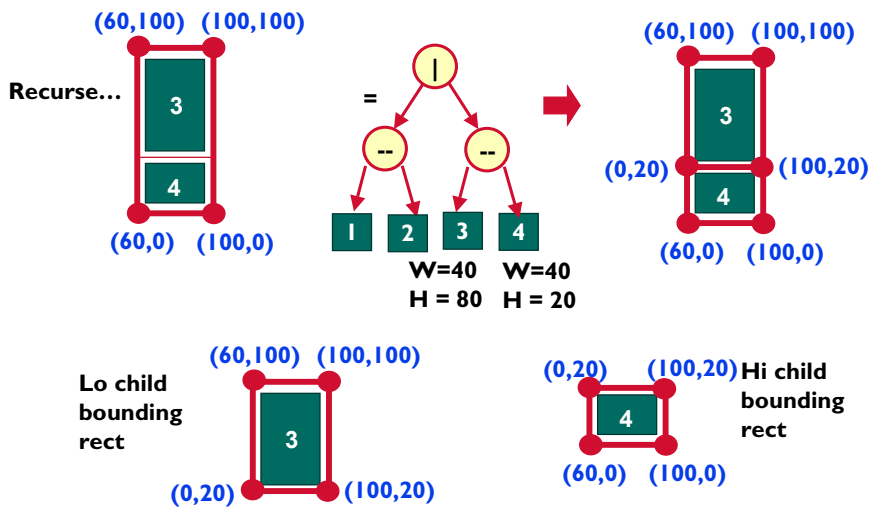
Sizing a Slicing Tree: Second Pass



© R. Rutenbar 2001

CMU 18-760, Fall01 32

Sizing a Slicing Tree: Second Pass



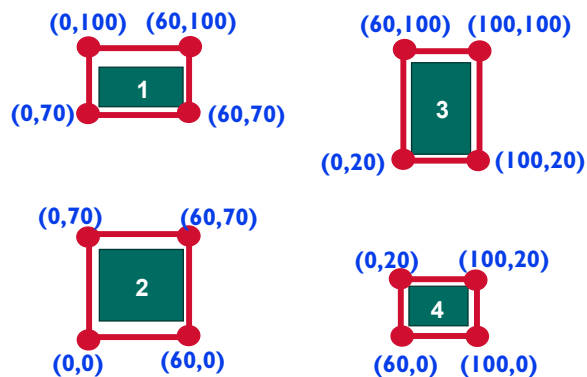
© R. Rutenbar 2001

CMU 18-760, Fall01 33

Sizing a Slicing Tree: Second Pass

▼ So, what do we get as a final result here?

- Absolute coords of the **ROOM** in the floorplan for each placed object
- Note – you now have a new (easy) problem: where to put the module inside the room, since the room can be bigger than the mod. Usually, just center it.



© R. Rutenbar 2001

CMU 18-760, Fall01 34

Sizing a Slicing Tree: Second Pass

▼ Pseudo code is basically like the first pass

- ▶ Now, each tree node stores its floorplan bounding room rectangle

```
SizeRoom( tree node T, bounding rectangle R=[left, right, top, bot] ) {  
    // label the node with the room rectangle  
    T.room = R;  
  
    // if this is a leaf node, we're done  
    if( T.type == leaf node ) {  
        // center the module in room, and return  
    }  
    else if (T.type == "|" cut) {  
        // compute bounding room rects for left, right children  
        leftRect = [R.left, R.left+lochild(T).width, R.top, R.bot];  
        rightRect = [R.left+lochild(T).width, R.right, R.top, R.bot];  
        // push the bounding room rects down the slicing tree  
        SizeRoom(lochild(T), leftRect);  
        SizeRoom(hichild(T), rightRect);  
    }  
    else if (T.type == "." cut) {  
        // compute bounding room rects for top, bottom children  
        topRect = [R.left, R.right, R.top, R.top - lochild(T).height];  
        botRect = [R.left, R.right, R.right, R.top - lochild(T).height, R.bot];  
        // push the bounding room rects down the slicing tree  
        SizeRoom(lochild(T), topRect);  
        SizeRoom(hichild(T), botRect);  
    }  
}
```

© R. Rutenbar 2001

CMU 18-760, Fall01 35

Slicing Trees: Annealing the Placement

▼ Where are we?

- ▶ Given a slicing tree, we can **SIZE** the tree so we can translate the relative topological placement into a real, absolute placement

▼ Next problem: how to we change the floorplan layout?

- ▶ Powerful idea: anneal the slicing tree itself
- ▶ Each annealing move perturbs the topology of the whole tree
- ▶ So, a small move can relocate **ALL** modules in layout, quickly
- ▶ And, the layout is always legal (no overlaps, like in HW4)

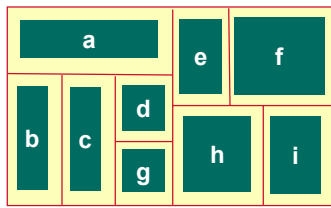
▼ What are the right “moves” for a slicing tree?

- ▶ Turns out we need a few different types

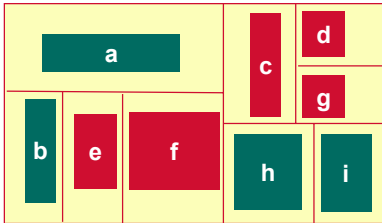
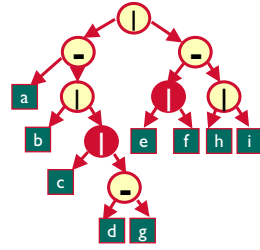
© R. Rutenbar 2001

CMU 18-760, Fall01 36

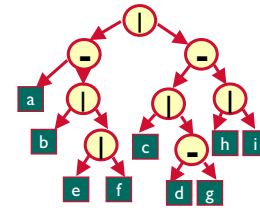
Swapping 2 SubTrees: Bigger Move Example



=

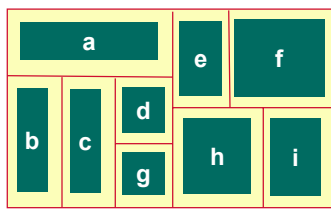


=

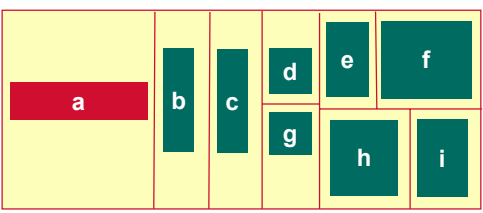
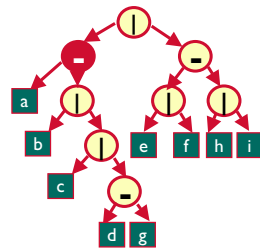


© R. Rutenbar 2001 CMU 18-760, Fall01 39

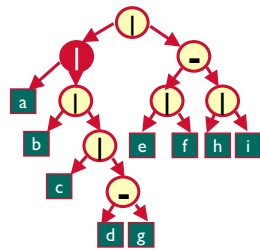
Node Cut Chain Inversion: 1-Node Example



=

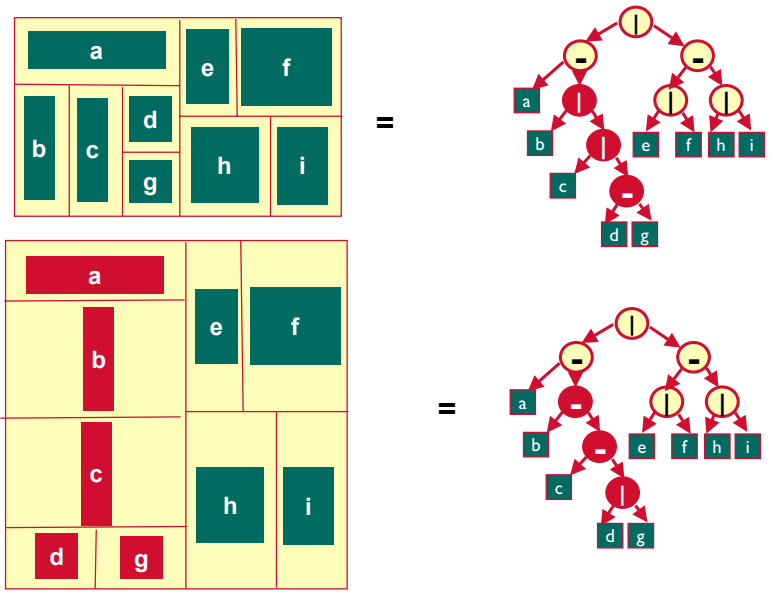


=



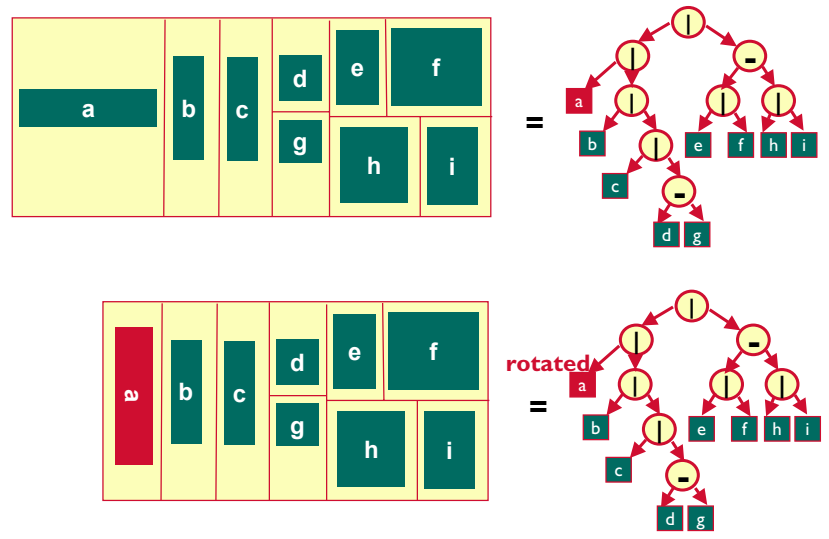
© R. Rutenbar 2001 CMU 18-760, Fall01 40

Node Cut Chain Inversion: 3-Node Example



© R. Rutenbar 2001 CMU 18-760, Fall01 41

Rotating Leaf Node Ex: Rotate Module "a"



© R. Rutenbar 2001 CMU 18-760, Fall01 42

Annealing a Slicing Tree

▼ Mechanically, do this in the annealing inner-loop

- ▶ Pick one of 3 random move types: swap, invert, leaf change
- ▶ Pick random node(s) to do the move on
- ▶ Do the move on the tree
- ▶ So the sizing operation to get an absolute location for each module
- ▶ This tells where all the pins are on the modules, and any pins around the outside of the layout
- ▶ Now, calculate the total wirelength of this perturbed floorplan
- ▶ Evaluate the cost function
- ▶ Decide to accept or reject the move
- ▶ If you reject it, you have to “undo” this change in the tree

▼ Note

- ▶ Not very incremental, on the move evaluation for wirelength
- ▶ That’s just the way it works for slicing trees.

© R. Rutenbar 2001

CMU 18-760, Fall01 43

Annealing Cost Function for Slicing Tree

▼ Can be pretty simple

- ▶ (Area of layout) + weight* (\sum wirelengths)
- ▶ You need to pick the weight to normalize the relative contribution of area (in units of length²) and wirelen (in units of length)

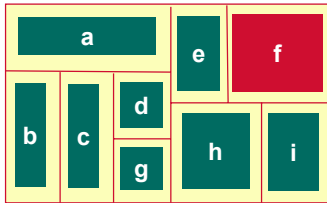
▼ This is enough to do a pretty good job of floorplanning...

© R. Rutenbar 2001

CMU 18-760, Fall01 44

..But Wait—There's More

- What if the individual modules you want to place have flexible or malleable shapes? How do we know which shape to pick?

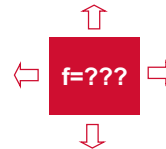


What if module *f* can be in any of these shapes?



Or, what if *f* doesn't have a shape yet, just some general constraints, like

Area ~ 100
Height < 50
Width < 50



© R. Rutenbar 2001

CMU 18-760, Fall01 45

Choosing Optimal Shape: Stockmeyer Algorithm

- Amazingly enough, for slicing trees, there is an *exact* solution

- ▶ Stockmeyer algorithm (originally developed by Stockmeyer and extended by Otten) can do this
- ▶ Solution is optimal with respect to area, obtainable in polynomial time in number of placed modules

Big idea

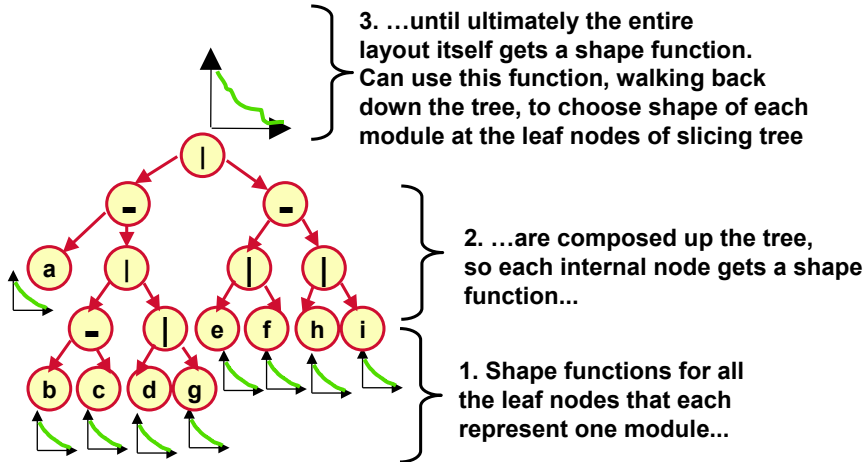
- ▶ Set of allowable shapes for each placed module is captured in a data structure called a **shape function**
- ▶ In a slicing tree, shape functions for the children of a node can be efficiently combined to make the shape function for the parent node
- ▶ Can start with shapes for the leaf node modules in the tree, and can “walk” up the tree to get a shape function for the entire floorplan

© R. Rutenbar 2001

CMU 18-760, Fall01 46

Choosing Optimal Shape: Stockmeyer Algorithm

Informally



© R. Rutenbar 2001 CMU 18-760, Fall01 47

Shape Functions

Definition

- ▶ Start with a shape relation = set of specified (x,y) pairs
- ▶ Shape relation is

$$R_M = \{ (x,y) \mid \text{module } M \text{ would "fit" in an } x \text{ by } y \text{ rectangle} \}$$

Examples

- ▶ Module M not allowed to rotate

$$R_M = \{ (x,y) \mid x \geq w \text{ and } y \geq h \}$$

- ▶ Module M, allowed to rotate

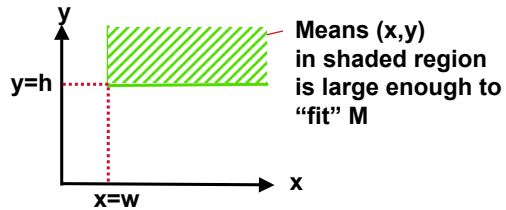
$$R_M = \{ (x,y) \mid x \geq w \text{ and } y \geq h \text{ OR } x \geq h \text{ and } y \geq w \}$$

© R. Rutenbar 2001 CMU 18-760, Fall01 48

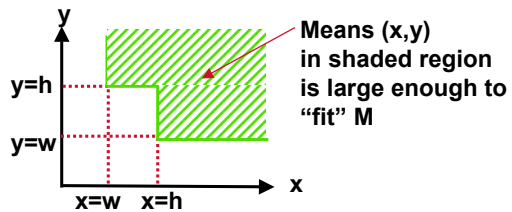
Shape Functions

Shape relations are easiest to see if we graph them

Module, no rotation



Module with rotation



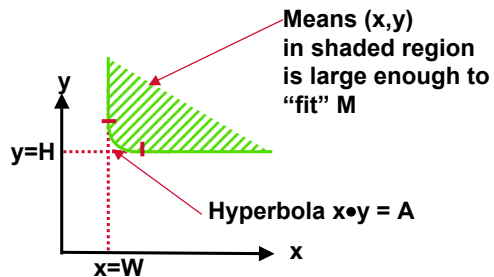
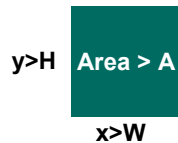
© R. Rutenbar 2001

CMU 18-760, Fall01 49

Shape Functions

Can be as complex as you like/want

Module, has $\text{Area} > A$, $x > W$, $y > H$



© R. Rutenbar 2001

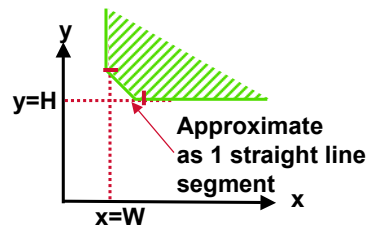
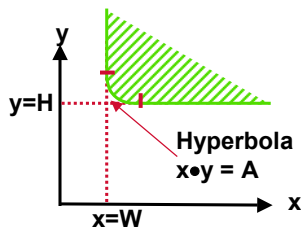
CMU 18-760, Fall01 50

Shape Functions

Big idea

- ▶ The relation R_M is very complex
- ▶ But, you can represent it entirely by just representing its lower boundary, ie, the line that separates the “space too small” (x,y) pairs from the “space big enough to fit” (x,y) pairs
- ▶ In practice, people represent these things using piece-wise linear approximations, for efficiency in space

Example



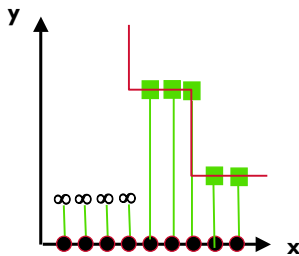
© R. Rutenbar 2001

CMU 18-760, Fall01 51

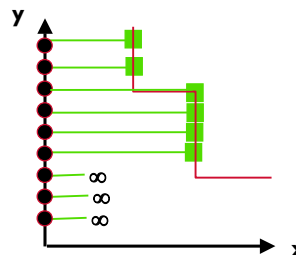
Shape Functions

For us, its OK to be really simple if you want implement it

- ▶ This is not at all efficient, but for a first cut, its much easier to code



An array, one slot for each x value, store the corresponding y value of the shape function



Another array, one slot for each y value, store the corresponding x value of the shape function

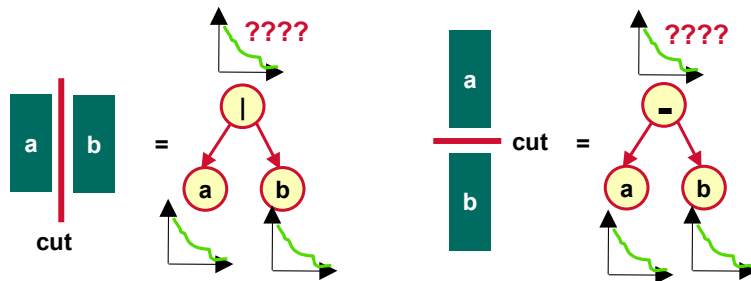
© R. Rutenbar 2001

CMU 18-760, Fall01 52

Composing Shape Functions

Goal

- ▶ Suppose we have shape functions for all the individual modules, the leaf nodes in our slicing tree
- ▶ How do we get a shape function for the entire, overall layout itself?
- ▶ Given structure of a slicing tree, turns into 2 basic composition questions

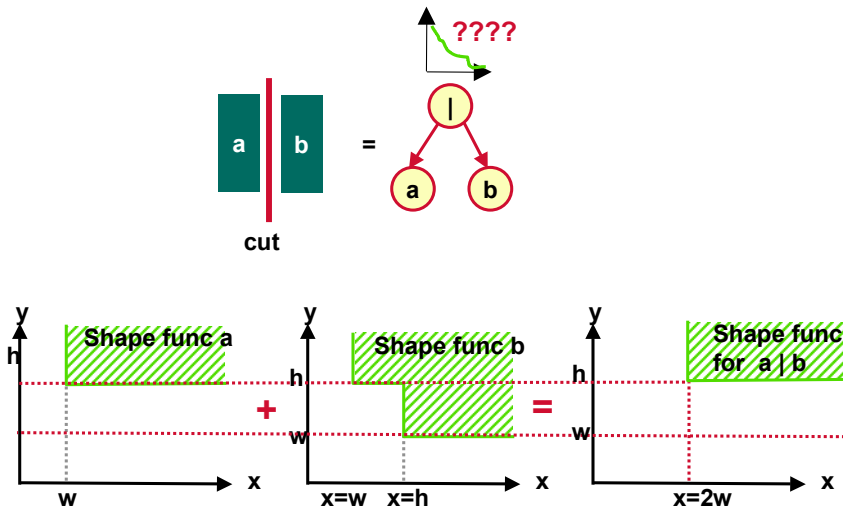


© R. Rutenbar 2001

CMU 18-760, Fall01 53

Composing Shape Functions

It's surprisingly easy: you just add them "graphically"

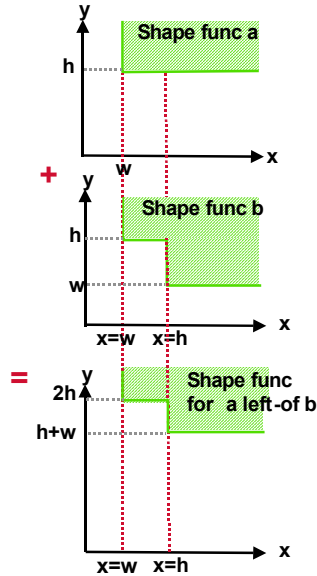
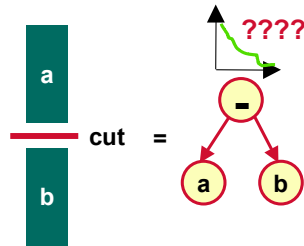


© R. Rutenbar 2001

CMU 18-760, Fall01 54

Composing Shape Functions

▼ Ditto for vertical adjacency



© R. Rutenbar 2001

CMU 18-760, Fall01 55

Using Shape Functions

▼ Methodology

- ▶ Obtain slicing tree (however you like...)
- ▶ Build shape functions for each leaf node (each module)
- ▶ Compose UP the tree, building shape functions for EACH internal node of the slicing tree
- ▶ Stop when you have the shape function for the root; this is the set of allowable shapes for the whole layout

▼ New problem

- ▶ From a shape function for the whole layout, how do we “invert” the functions and pick shapes for the individual modules?

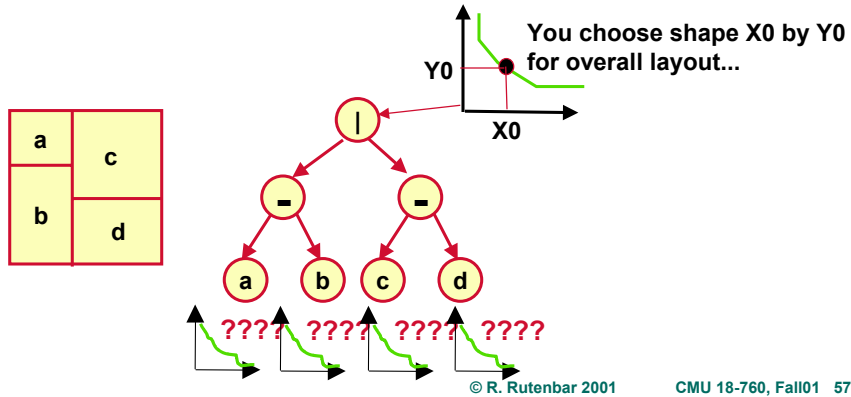
© R. Rutenbar 2001

CMU 18-760, Fall01 56

Shape Function Inversion

Task

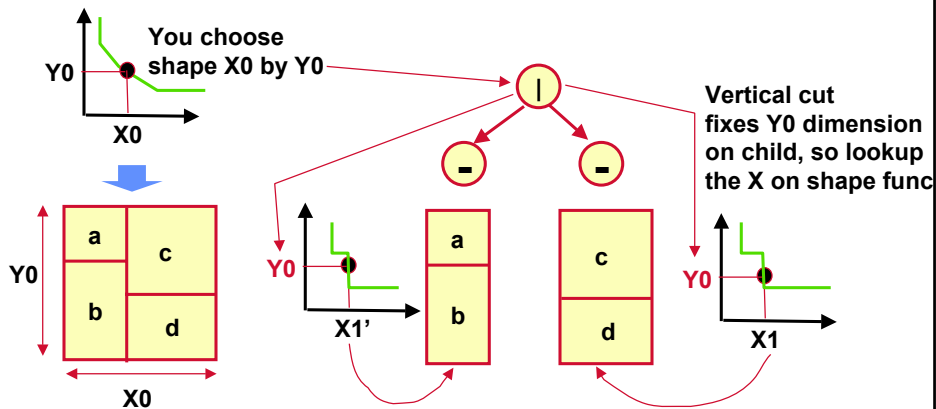
- ▶ Pick a desirable “final shape” for the whole layout, by selecting a point on the root shape function.
- ▶ “Invert” the functions down the tree to get a selected shape for each leaf module



Shape Function Inversion

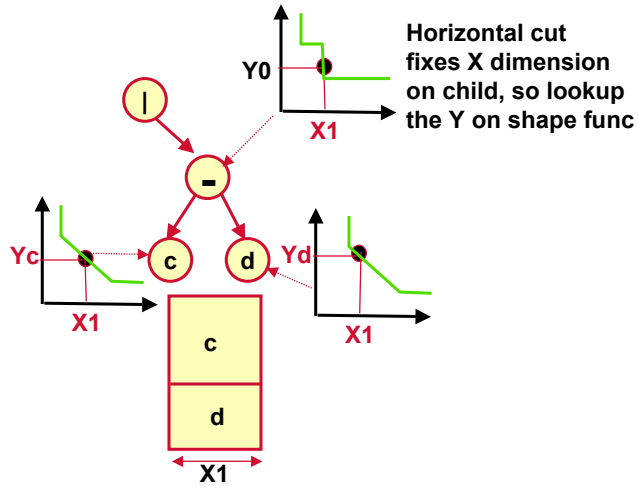
Task

- ▶ Remember, you have a shape function for EACH internal node...
- ▶ “Invert” the functions down the tree to get a selected shape for each leaf module



Shape Function Inversion

Continue this idea



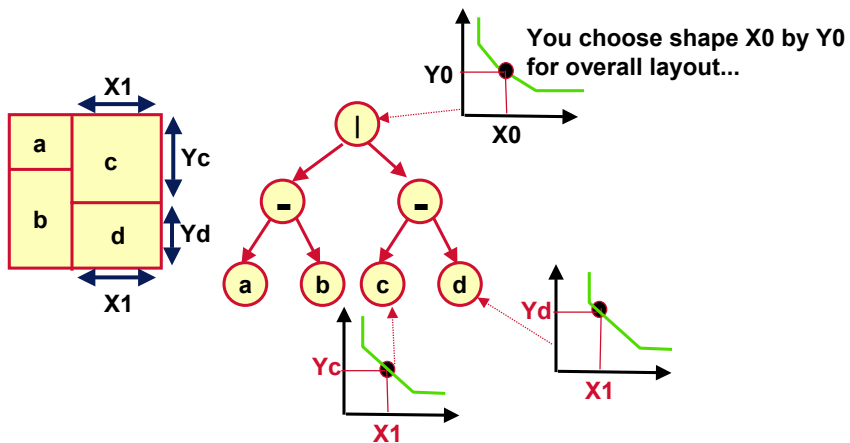
© R. Rutenbar 2001

CMU 18-760, Fall01 59

Shape Function Inversion

Result

- ▶ Inverting down tree selects the right X x Y shape for each leaf module
- ▶ In this example, we only did right side



© R. Rutenbar 2001

CMU 18-760, Fall01 60

Shape Function Inversion

▼ Why does this work?

- ▶ Individual module shape functions specify size of “holes” big enough for the individual modules
- ▶ Composing shape functions adds the sizes of the hole left-right and top-bottom
- ▶ At top, shape function applies to whole layout. An $X \times Y$ you pick is guaranteed to be realizable
- ▶ The shape function at each “|” or “-” cut node specifies that one dimension is fixed, and the other varies
- ▶ The individual shape functions are used to translate the one fixed coord of the “hole” into the other dimension we need
- ▶ ...and down the tree we go

▼ Result

- ▶ Individual leaf modules can be **fixed**, or have a finite **set** of shapes, or have a **continuum** of shapes, and we can represent slicing layout

© R. Rutenbar 2001

CMU 18-760, Fall01 61

Placing with Shape Functions

▼ How do we actually place these modules

- ▶ ie, we know how to represent a slicing tree and solve for right module shape, given we pick a point on root shape function
- ▶ How do we create the slicing tree in the first place?

▼ Just a tweak on annealing-based floorplanning

- ▶ Still iteratively improve (perturb) a slicing tree in the same way
- ▶ And, after each perturbation, we propagate shape functions **up** tree to build root shape function
- ▶ Then, we pick a min area layout, and invert functions **down** tree to get module locations -- this is the “sizing” step
- ▶ Then given all the room locations, we compute the wirelengths
- ▶ Then, we can again evaluate a cost function with Area+Wirelength
- ▶ Turns out you can do this all **really fast** for a few hundred blocks

© R. Rutenbar 2001

CMU 18-760, Fall01 62

Slicing Tree + Shape Functions

▼ Summary

▼ Pro

- ▶ Very convenient subclass of layouts
- ▶ Easy to get placements via annealing ideas
- ▶ Can represent as one tree, can solve fast for optimal shapes for individual objects in the placement, even with malleable shapes

▼ Con

- ▶ Cannot represent every possible layout you could draw
- ▶ Example: spirals and anti-spirals
- ▶ *Turns out there are a lot of other, more recent topological representations that can let us do these layouts. We don't have time to do any of these.*

© R. Rutenbar 2001

CMU 18-760, Fall01 63

(Project3) Slicing-Tree Floorplanning

▼ Input

- ▶ A netlist of malleable rectangular blocks, and nets connecting them...
- ▶ ...and, if you choose to handle these: "timing arcs" for block delay

▼ Output

- ▶ A placed floorplan for the blocks, and info about overall area, netlength,
- ▶ ...and, if you choose to handle this: critical path timing

▼ Strategy

- ▶ Slicing tree annealing placement ideas
- ▶ ...and, if you choose to do this: combine with static timing ideas

▼ Logistics

- ▶ You can work in groups of 2
- ▶ No paper writeup: web-page required
- ▶ Demo to TAs also required

▼ Due: last week of class

© R. Rutenbar 2001

CMU 18-760, Fall01 64

Our Block Model for Floorplans

Simple rectangles, but with variable shape

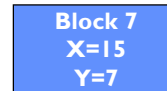
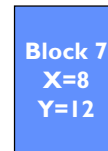
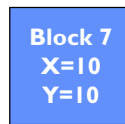
- ▶ Blocks numbered consecutively: 1,2,3, ... ,B
- ▶ Blocks have a finite number (eg, a few) alternative shapes
- ▶ Each shape is a rectangle

Input file:

Block id #shapes x1 y1 x2 y2 ... xn yn

```
...
block 7 3 10 10 8 12 15 7
...
```

Block #7 has
3 rectangular
shapes, shown
at right



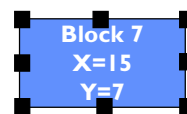
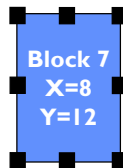
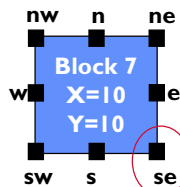
© R. Rutenbar 2001

CMU 18-760, Fall01 65

Our Block Model for Floorplans

Blocks have pin sites at which nets connect

- ▶ Pin sites are an abstraction of the real locations of the pins--a simplification to a small set of fixed "sites"
- ▶ Pins are always at the 8 compass points: n, s, e, w, ne, se, nw, sw
- ▶ We name pins and refer to them in the netlist input file using these 1 char & 2char lower case names



This pin is referred to as the pair <blockID, pintype>
which is "7 se" for this pin, in the input file

© R. Rutenbar 2001

CMU 18-760, Fall01 66

Our Block Model for Floorplans

▼ Blocks can be placed anywhere on chip

- ▶ Blocks have integer width (x) and height (x) for all shapes
- ▶ Floorplan itself is an integer grid
- ▶ Blocks can be rotated in increments of 90 degrees: we name the rotations: 0, 90, 180, 270
- ▶ Blocks **CANNOT** be reflected (about x or y axes)
 - ▷ This just makes life a little simpler....

▼ Specifying a block in a layout: *location & rotation & shape*

- ▶ To specify the location of a placed block, we use the **CENTER** coords of the block (note, they will be ints, or int+1/2, write them out as floats)
- ▶ To specify rotation of a placed block, we use one of {0, 90, 180, 270}, ie, write this out as an int
- ▶ To specify the shape of a placed blocks, we use the order in which shapes were listed in input netlist: 1, 2, 3, ...
 - ▷ A block with 1 fixed shape gets a "1"

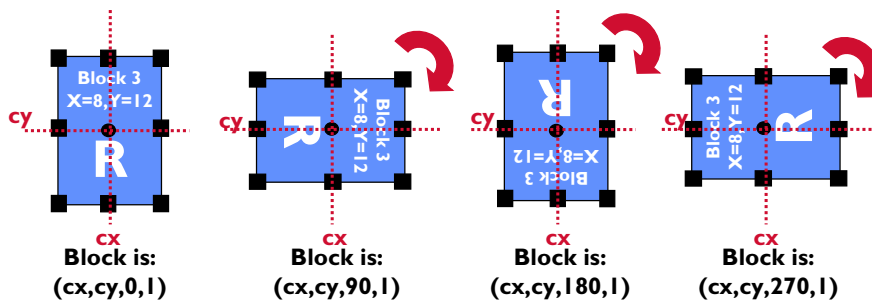
© R. Rutenbar 2001

CMU 18-760, Fall01 67

Our Block Model for Floorplans

▼ Example:

- ▶ Assume this block has just one shape
- ▶ This block placed at constant center, but all in 4 different orientations



© R. Rutenbar 2001

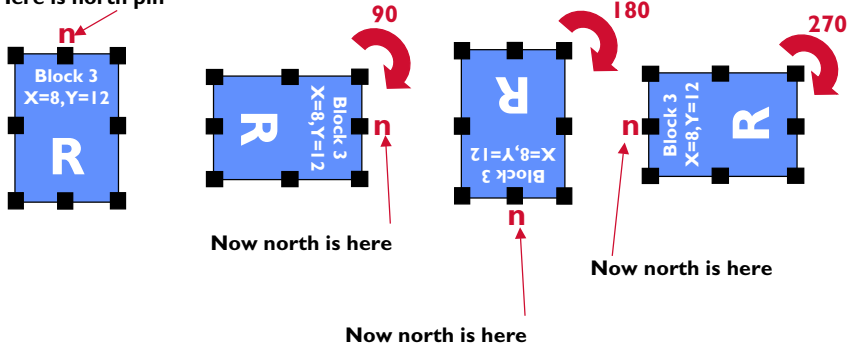
CMU 18-760, Fall01 68

Our Block Model for Floorplans

How does pin naming work for rotations?

- ▶ Pins rotate too: you have to remember to figure out where the pin ends up (pinX, pinY) when block rotates
- ▶ This block placed at constant center, but all in 4 different orientations

Here is north pin



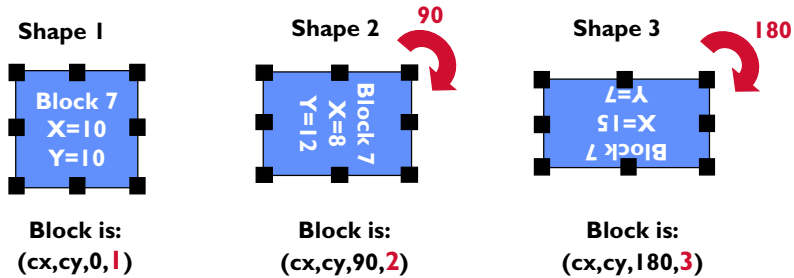
© R. Rutenbar 2001

CMU 18-760, Fall01 69

Our Block Model for Floorplans

What if there are more shapes?

...
 block 7 3 10 10 8 12 15 7 3 different shapes for block 7
 ...



© R. Rutenbar 2001

CMU 18-760, Fall01 70

Our Block Model for Floorplans

Implementation hint: rotations

- ▶ Make a table for each block, for each shape
- ▶ Entries for each of the 4 rotations: 0, 90, 180, 270
- ▶ Save the ΔX and ΔY values you need to add to the (centerX, centerY) location of the block to compute location of pin
- ▶ These $(\Delta X, \Delta Y)$ values are constant, independent of the block location, only depending on the block, the shape of the block.
- ▶ This saves you the grief of computing these every time a block move; you only do it once, at start of the program

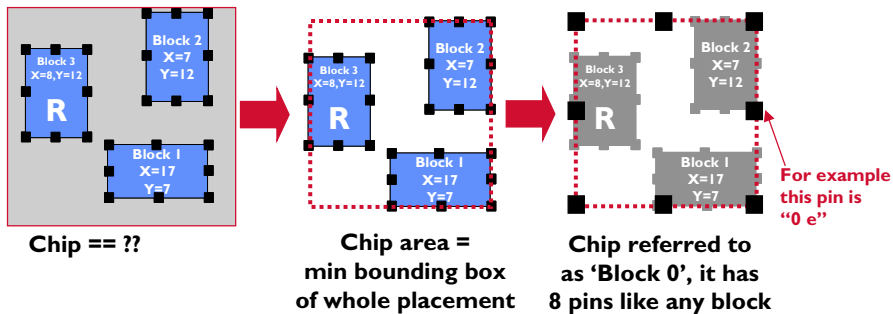
© R. Rutenbar 2001

CMU 18-760, Fall01 71

Our Chip Model for Floorplans

The “chip” itself is treated as a “special” block -- block 0

- ▶ It has flexible shape--we don't know what it is until we are done with the slicing floorplan.
- ▶ It has pins just like an ordinary block: n, s, e, w, ne, se, nw, sw
- ▶ It is *defined* to be the min bounding box of all placed blocks, ie, the shape for the final floorplan at the root of the slicing tree



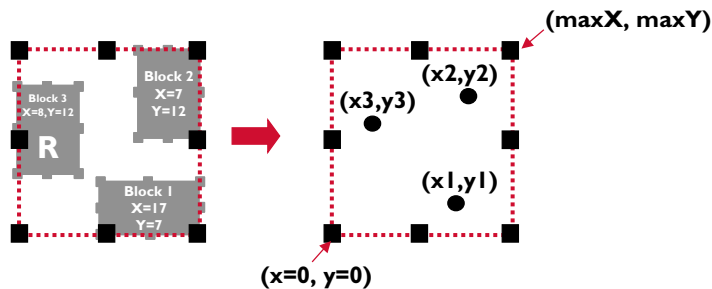
© R. Rutenbar 2001

CMU 18-760, Fall01 72

Our Chip Model for Floorplans

What is the coordinate system?

- ▶ Origin for *chip* is at lower left; all (x,y) coord positive numbers
- ▶ All placed objects specified by their center coords in this frame
- ▶ Center coords will be ints or 1/2 ints, eg (45, 64), (45.5, 52), (57.5, 88)...
- ▶ But you only have to print this out at the end of the placement



© R. Rutenbar 2001

CMU 18-760, Fall01 73

Our Net Model for Floorplans

A net is just a set of 2 or more pins

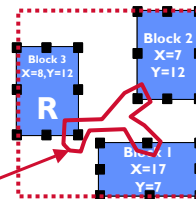
- ▶ Nets numbered consecutively from 1: 1,2,3, ..., N
- ▶ Pins specified as “blockID pinSide”; pins on whole chip are “0 pinSide”
- ▶ First pin listed is the *driver* (eg, gate output), next ones listed are *inputs*
 - ▷ You need to know this direction stuff for timing

Input file:

Net id #pins block pin block pin

```
...
net 6 3 3 se 2 sw 1 n
...
```

Net #6 has
3 pins on the
blocks, shown
at right

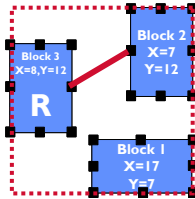


© R. Rutenbar 2001

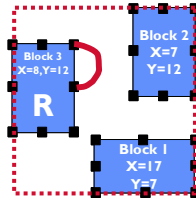
CMU 18-760, Fall01 74

Our Net Model for Floorplans

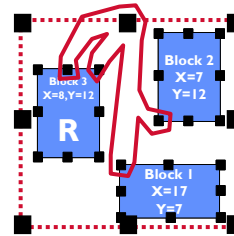
Examples



Simple 2 pt net:
net i 2 3 e 2 w



Another 2 pt net:
Nets can have all their
pins on one (real) block:
net i 2 3 ne 3 e



A 5 pt net:
This one goes to a chip pin
and to 4 other block pins;
chip outline drawn bigger
here for clarity:
net i 5 0 n 3 n 3 ne l nw s w

© R. Rutenbar 2001

CMU 18-760, Fall01 75

Our Net Model for Floorplans

What do we care about for the nets?

- ▶ **Length:** we want a placement of blocks to make them short
- ▶ **Timing:** we will also have a detailing timing model, so we can work directly on the critical path itself

Netlength model

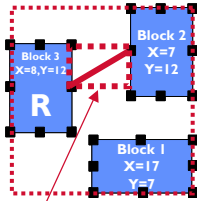
- ▶ **Simple:** $1/2$ perimeter metric for each net
- ▶ **Total netlength =** add them all up = $\sum_{(\text{all nets } i)} (\text{net length } i)$
- ▶ **Pins are modeled as a single dimensionless point:** a pair of ints
- ▶ **Find leftX, rightX, topY, bottomY for all pins on your net #i**
- ▶ $1/2$ perimeter length metric is just: $| \text{rightX} - \text{leftX} | + | \text{topY} - \text{bottomY} |$

© R. Rutenbar 2001

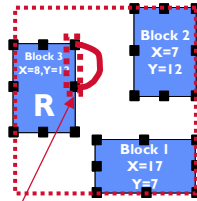
CMU 18-760, Fall01 76

Our Net Model for Floorplans

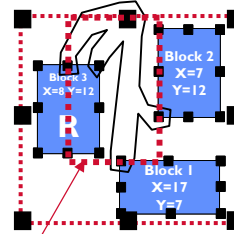
Net length examples



Length for this 2pt net is box $\Delta X + \Delta Y$



Length for this 2pt net is also box $\Delta X + \Delta Y$,
But = $0 + \Delta Y$ in this case



Length for this 5pt net is also box $\Delta X + \Delta Y$.
It's a much bigger box now,
And remember that the chip pin is on the top,
at X center, Y top coord of the layout bounding box

© R. Rutenbar 2001

CMU 18-760, Fall01 77

Floorplan Goals: Simplified

So, what do we want the floorplanner tool to do?

Let's first ignore the timing issues

Goals

- ▶ Place all blocks: determine (Xcenter, Ycenter, rotation, shape) for each
- ▶ Pick good shape for each block from among variants listed in netlist
- ▶ Make chip area small
- ▶ Make total netlength small

How?

- ▶ Represent as a slicing tree, annealing the slicing tree
- ▶ Work out the sizing issues so you can get an absolute placement of each block, and use this to figure out where the pins went, so you can do wirelength

© R. Rutenbar 2001

CMU 18-760, Fall01 78

Annealing Formulation

▼ Suggested cost function

- ▶ W_n empirically chosen weight to balance terms in cost

$$\text{Cost} = \underbrace{[\text{Area}] +}_{\substack{\text{Objective:} \\ \text{Make area of} \\ \text{whole chip} \\ \text{(block \#0)} \\ \text{=small}}} \underbrace{W_n * [\text{Netlength}]}_{\substack{\text{Objective:} \\ \text{Make} \\ \sum \text{netlen's} \\ \text{=small}}}$$

© R. Rutenbar 2001

CMU 18-760, Fall01 79

Basic Floorplanning: Implementation Hints

▼ How do I know what random move to pick?

- ▶ Implement so you can easily pick, up front, fraction F_i of total moves that will go to moves of type- i
- ▶ Suppose we have these moves:
 - ▷ Swap 2 subtrees
 - ▷ Invert cut chain
 - ▷ Reshape or rotate a block
- ▶ We want 3 fractions F_{swap} , F_{invert} , F_{shape} that sum to =1
- ▶ We want to guarantee that if we do N moves at this temp, that:
 - ▷ $\sim N * F_{\text{swap}}$ block swaps get tried
 - ▷ $\sim N * F_{\text{invert}}$ block rotates get tried
 - ▷ $\sim N * F_{\text{shape}}$ block reshapes get tried

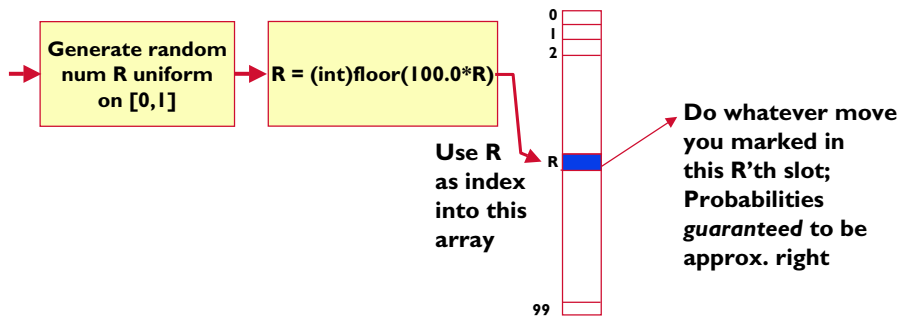
© R. Rutenbar 2001

CMU 18-760, Fall01 80

Basic Floorplanning: Implementation Hints

▼ Easy trick

- ▶ Suppose you want: $F_{\text{swap}}=50\%$ $F_{\text{invert}}=30\%$ $F_{\text{shape}}=20\%$
- ▶ Make an array with 100 entries
- ▶ In the first 50 entries, put a marker that says “do swap”
- ▶ In next 20 entries, put a marker for “do invert”
- ▶ Ditto remaining entries: last 20 = “do shape”



© R. Rutenbar 2001

CMU 18-760, Fall01 81

Optional (Cool, Advanced, Harder) Parts of Proj3

▼ Do shape functions

- ▶ Note – you don't HAVE to do this
- ▶ You can just do simple moves that change shape of your leaf cells
- ▶ But, its more interesting to do it with shape functions, and more work

▼ Handle our timing model

- ▶ Again – it's optional, you don't HAVE to do it

© R. Rutenbar 2001

CMU 18-760, Fall01 82

Floorplanning -> Timing

▼ Project goals

- ▶ First goal is to be able to get a decent floorplan:
 - ▷ Packed, small area, small wirelength, no overlap
 - ▷ (or, not much overlap--hard to make it 0 without more fancy stuff)
- ▶ Next (optional) goal: **good timing**

▼ We also have a timing model

- ▶ Each block has a timing model: *timing arcs*
- ▶ Each net has a timing model: *length-based delay*
- ▶ You get to build, maintain, update timing graph
- ▶ As placement evolves, blocks move, so nets change, so net delay changes, so critical path changes, so timing changes
- ▶ You get to track all this...

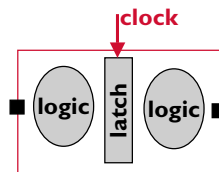
© R. Rutenbar 2001

CMU 18-760, Fall01 83

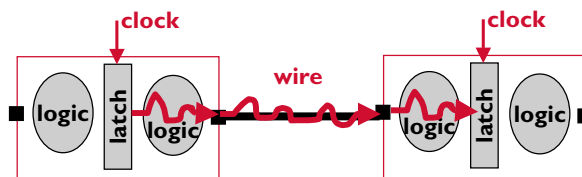
Our Timing Model for Floorplanning

▼ Big assumption: simple, edge-triggered, synchronous clock

- ▶ Every block, internally, looks like this



- ▶ 2 sources of delay: thru **logic** inside a block, thru **wires** that connect blocks



© R. Rutenbar 2001

CMU 18-760, Fall01 84

Our Timing Model for Floorplanning

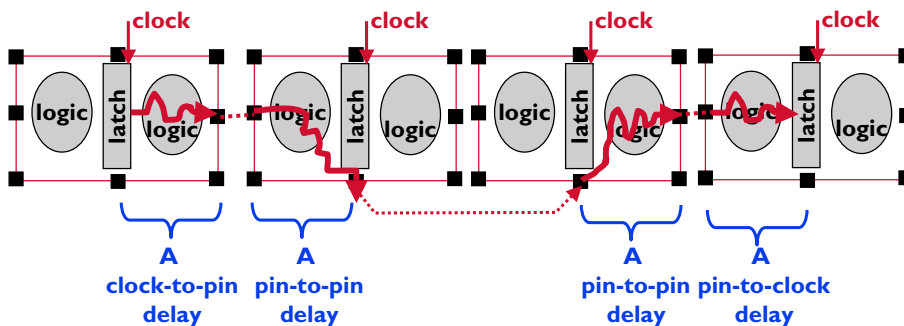
- ▼ 3 components of timing model
- ▼ Delays thru a block
 - ▶ Pin to pin delay
 - ▶ Pin to clock delay
 - ▶ Clock to pin delay
- ▼ Delays thru a net that connects blocks
 - ▶ Length-based delay for a net
- ▼ Delay thru a net that connects to a chip pin
 - ▶ Length-based clock to pin delay (input pin)
 - ▶ Length-based pin to clock delay (output pin)

© R. Rutenbar 2001

CMU 18-760, Fall01 85

Delays Thru a Block

- ▼ How fast can the chip go?
 - ▶ Depends on maximum delay from latch to latch
 - ▶ If we ignore wire delay (for now), where do these delays come from?



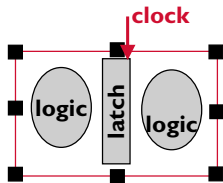
© R. Rutenbar 2001

CMU 18-760, Fall01 86

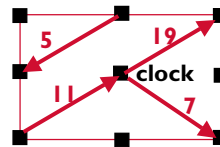
Delays Thru a Block

How do we model these 3 delays

- ▶ Pretend the “latch” is like a pin; call it the “clock” pin
- ▶ We give a delay edge from a pin to a pin (clock counts here)
- ▶ Edge gives direction (which way signal goes) and delay number
- ▶ Standard name for these: **timing arcs**



Timing model



Note:
 “5” is pin-to-pin
 “11” is pin-to-clock
 “19”, “7” are clock-to-pin

Each arc always has one “from” pin, one “to” pin, and a delay number.
 Arcs legal between any pair of pins, including the “clock” pin, inside a block

© R. Rutenbar 2001 CMU 18-760, Fall01 87

Delays Thru a Block

Specifying these in input file

- ▶ We give all arcs with each block
- ▶ We number arcs *globally, consecutively*, across all blocks: 1, 2, ... T
- ▶ Shape doesn’t affect timing arcs in our model: constant per block
- ▶ Format: `arc arcID fromPin toPin delay`

Input file:

```
...
block 7 3 10 10 8 12 15 7
```

```
timing 4
```

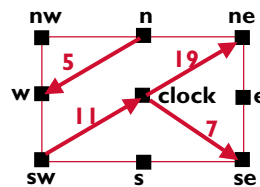
```
arc 21 n w 5
arc 22 sw c 11
arc 23 c se 7
arc 24 c ne 19
```

Block #7 has 4 arcs:
 #21, #22, #23, #24,
 and there is one line
 per arc in input file

```
block 8 .....
```

```
timing ....
```

```
arc ...
```

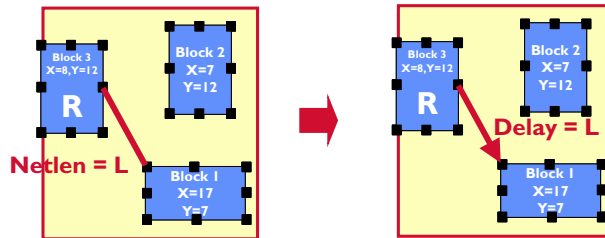


© R. Rutenbar 2001 CMU 18-760, Fall01 88

Delays Thru a Wire

Longer wires have longer delay

- ▶ How do we model this?
- ▶ Crudest possible model: $\text{delay} = 1/2$ perimeter wire length
- ▶ (This is a *lousy* model in reality--but we want to keep it simple here)
- ▶ Note that which pin is driver, which are receives *matters* for timing



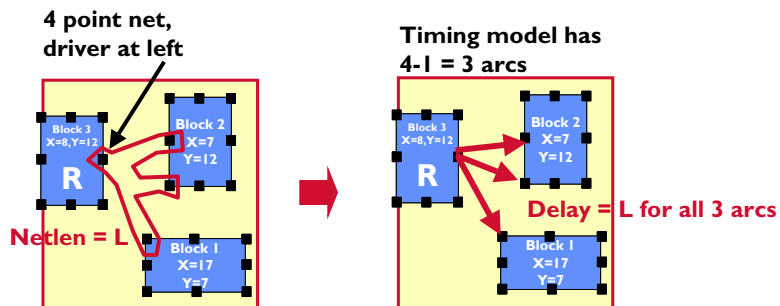
© R. Rutenbar 2001

CMU 18-760, Fall01 89

Delays Thru a Wire

Multipoint nets...?

- ▶ How do we model this? As **multiple** timing arcs from driver to receivers
- ▶ Which pin is driver, which are receives *matters* for timing



© R. Rutenbar 2001

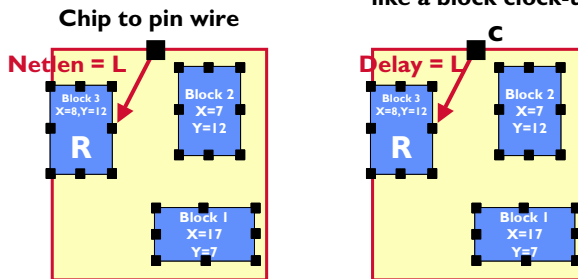
CMU 18-760, Fall01 90

Delay Thru Wires to Chip Pins

▼ New problem: how to model wires to chip IOs?

- ▶ Question is: where is the “clock” for these external signals
- ▶ Turns out there is a standard assumption: external signals use same clk
- ▶ Model it explicitly

Timing model:
Pretend this chip pin
is a “clock” pin, so this is
like a block clock-to-pin delay



© R. Rutenbar 2001

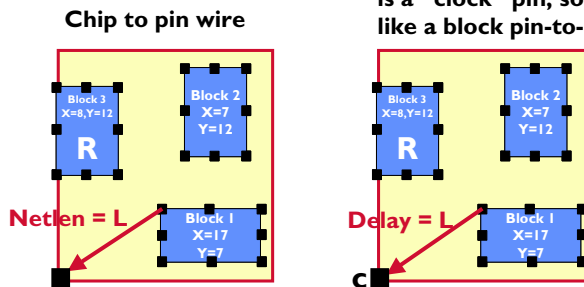
CMU 18-760, Fall01 91

Delay Thru Wires to Chip Pins

▼ Ditto for block-pin to chip

▼ Note: to make life easy, these nets are *always* 2 point nets

Timing model:
Pretend again this chip pin
is a “clock” pin, so this is
like a block pin-to-clock delay



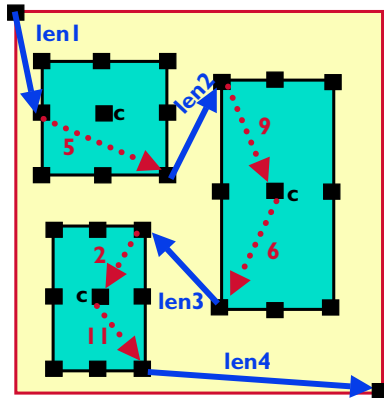
© R. Rutenbar 2001

CMU 18-760, Fall01 92

Handling Critical Paths

▼ Why are we doing this? We want to track *critical path*

- ▶ We can use delays thru a block + delays thru wires to build *timing graph*
- ▶ Consider a simple example with all arcs shown



2 chip pins
 5 internal block timing arcs (dotted)
 4 nets (solid)
 2 are pin-to-pin
 1 is chip-to-pin
 1 is pin-to-chip

© R. Rutenbar 2001

CMU 18-760, Fall01 93

Handling Critical Paths

▼ We want to build the *timing graph* (from *next* lecture...)

- ▶ It's actually mechanical: for this timing model, has a simple structure

src

One distinguished "start" node, called "source"

Nodes for each
block pin connected
to a net

Edges for each net,
and for each
timing arc

A lot of nodes and edges,
but we are guaranteed the overall
graph is a DAG -- no cycles

snk

One distinguished "end" node, called "sink"

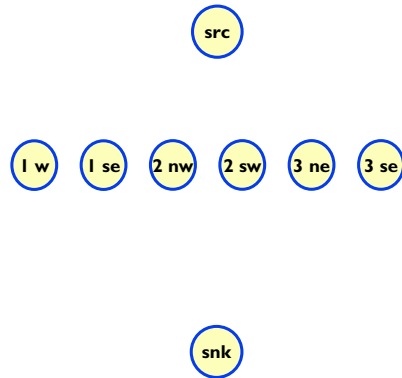
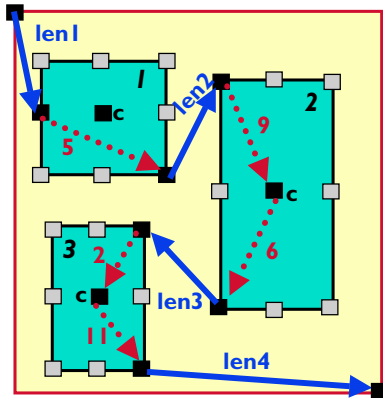
© R. Rutenbar 2001

CMU 18-760, Fall01 94

Handling Critical Paths

Step 1. Build all the nodes in graph

- ▶ One per block pin that is connected to a net
- ▶ (no clocks now)

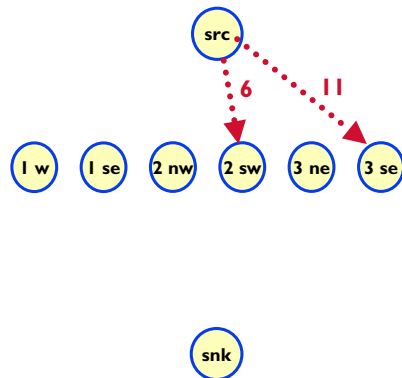
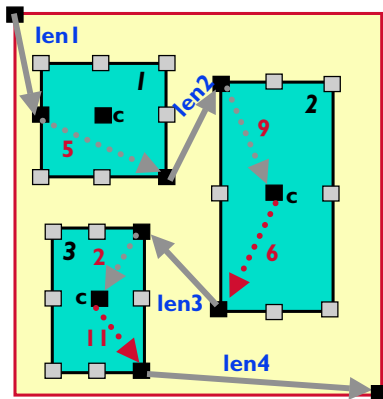


© R. Rutenbar 2001 CMU 18-760, Fall01 95

Handling Critical Paths

Step 2. Clock-to-pin edges

- ▶ For every timing arc FROM a clock node TO a block pin, add an edge in graph FROM source TO correct pin node

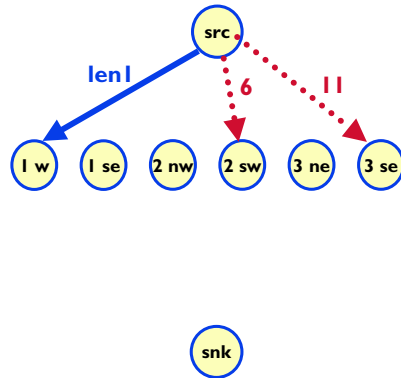
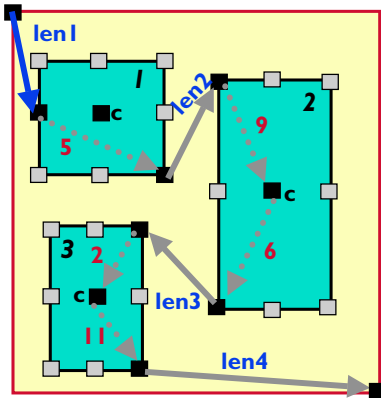


© R. Rutenbar 2001 CMU 18-760, Fall01 96

Handling Critical Paths

Step 3. Chip-to-pin edges

- ▶ For every net FROM a chip pin TO a block pin, add an edge in graph FROM source TO correct pin node

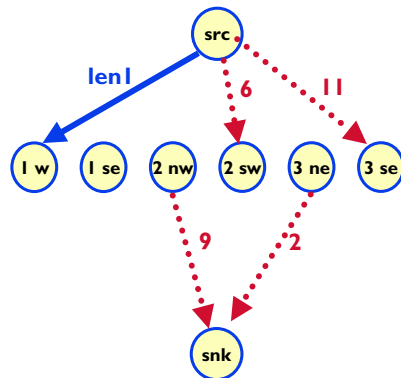
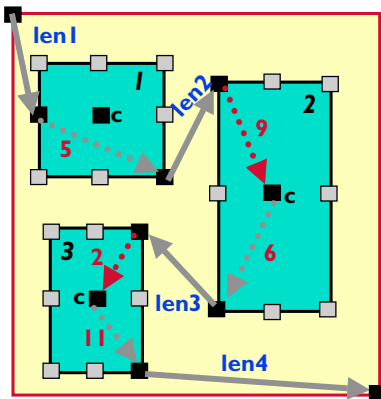


© R. Rutenbar 2001 CMU 18-760, Fall01 97

Handling Critical Paths

Step 4. Pin-to-clock edges

- ▶ For every timing arc FROM a block pin TO a clock, add an edge in graph FROM correct pin node TO sink

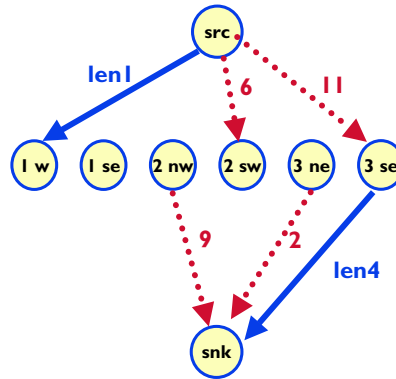
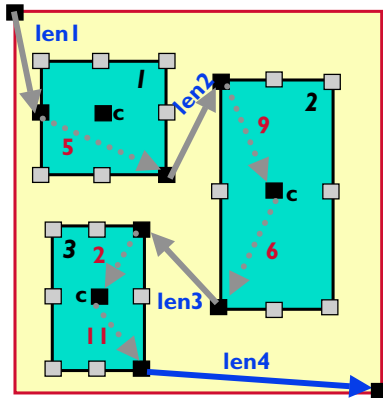


© R. Rutenbar 2001 CMU 18-760, Fall01 98

Handling Critical Paths

Step 5. Pin-to-chip edges

- ▶ For every net FROM a block pin TO a chip pin, add an edge in graph FROM correct pin node TO sink

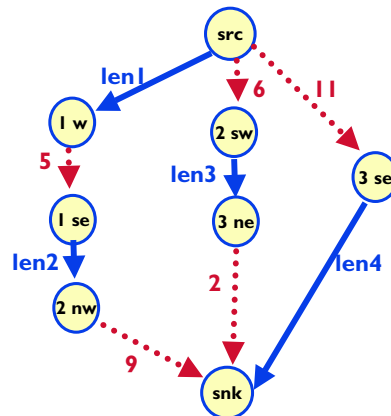
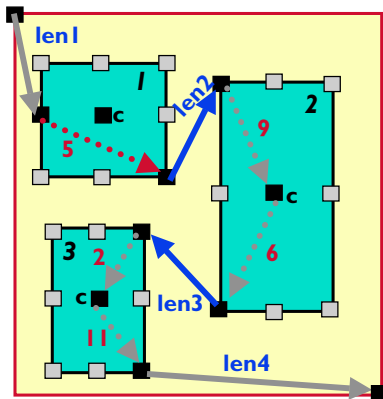


© R. Rutenbar 2001 CMU 18-760, Fall01 99

Handling Critical Paths

Step 6. Pin-to-pin edges

- ▶ For every net and every arc FROM a block pin TO a block pin, add an edge in graph FROM correct pin node TO correct pin node

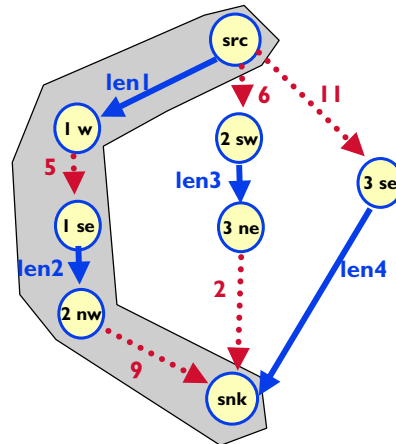
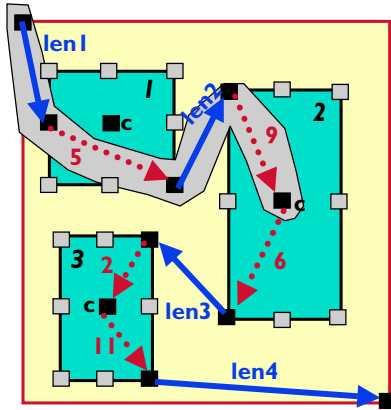


© R. Rutenbar 2001 CMU 18-760, Fall01 100

Handling Critical Paths

▼ Done. This is the required timing graph

- ▶ Longest path from Source to Sink == worst-case delay, latch-to-latch
- ▶ One path highlighted below



© R. Rutenbar 2001

CMU 18-760, Fall01 101

Observations

▼ Graph structure is constant--you only build it once

- ▶ Same nodes, same edges, always

▼ Timing arcs (dotted edges) are constant

- ▶ Placement does nothing to change intra-block timing in our simple model of floorplanning

▼ Placement *changes* the net delays (solid edge nums) in graph

- ▶ Move a block, pins moves, net lengths change, delays change in graph
- ▶ So, the critical path delay can change...
- ▶ ... and even *what* nets are on critical path

▼ If you update the net delays in graph during placement...

- ▶ You can track what the critical path is, and what worst delay is

© R. Rutenbar 2001

CMU 18-760, Fall01 102

Observations

▼ Aside

- ▶ This is why every net, and every timing arc, has its own ID in our netlist
- ▶ Makes it much easier to update edges in timing graph when all edges have a unique name

▼ Engineering decision: How will you couple placement & timing analysis?

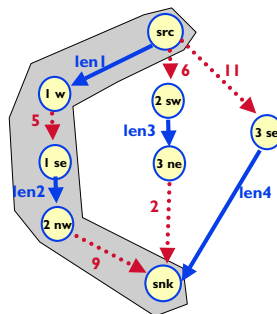
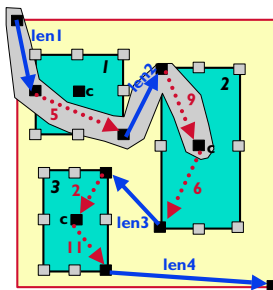
- ▶ Could update timing graph after **EVERY** move. Very accurate. Very slow.
- ▶ Could update timing graph every **K** moves. Just assume the **SAME** nets comprise the critical path in between. To eval timing changes as a result of a placement move, eval change is **JUST** $\Delta\Sigma$ (these net delays)
- ▶ Could update timing graph only every temperature. Do same as above.
- ▶ Could do timing graph **ONCE** only near beginning of placement, **HOPE** its always same critical path, never update it again till all done. (Very, very dumb...)

© R. Rutenbar 2001

CMU 18-760, Fall01 103

Coupling Example

▼ Update every temperature. Assume same crit path in between



Assume this path, thru nets 1, 2, is always the critical path.

▼ So, how do we eval Δ timing on subsequent placement moves?

- ▶ Δ timing == $\Delta(\text{len1} + \text{len2})$!! That's it. Very nice, very simple.

© R. Rutenbar 2001

CMU 18-760, Fall01 104

Implementation Hints

▼ Some messy issues

- ▶ What happens if several paths with same length, **ALL** critical?
 - ▷ You could try to track them **ALL** (your call)
- ▶ You could pick one, only worry about it.
 - ▷ When you update, if your placement changes screwed up other paths, your timing update will automatically always pick **A** worst path.
 - ▷ It will work ~OK if you update often enough
 - ▷ This is the easiest way to do it.

▼ Graph path mechanics

- ▶ Next lecture (for static timing stuff) and, actually, maze routing mechanics (want now **MAX** path thru this graph).

© R. Rutenbar 2001

CMU 18-760, Fall01 105

Coupling Timing into Annealing Placement

▼ How? 3 options

- ▶ Option 1: don't. Just ignore timing issues.
- ▶ Option 2: as an objective to minimize, like area:

$$\text{Cost} = [\text{Area}] + W_n * [\text{Netlen}] + W_t * [\text{max Delay}]$$

- ▶ Option 3: as a constraint. We give you a target **T**, you try to meet it:

$$\text{Cost} = [\text{Area}] + W_n * [\text{Netlen}] + W_t * [\text{TimeMiss}]^2$$

$$\text{TimeMiss} = \begin{cases} \text{If } (\text{maxDelay} > \text{target } T) \\ \text{then } |T - \text{maxDelay}| \\ \text{else } 0 \end{cases}$$

© R. Rutenbar 2001

CMU 18-760, Fall01 106

Overall Input File Format

▼ All ints and short lower-case-only strings at start of a line

```
#blocks #nets timingSpec
block 1 #shapes x1 y1 ... xn yn
timing #arcs
arc 1 fromPin toPin delay
arc 2 fromPin toPin delay
...
arc m fromPin toPin delay
block 2 #shapes x1 y1 ... xn yn
timing #arcs
arc <m+1> fromPin toPin delay
arc <m+2> fromPin toPin delay
...
block B .....
timing ....
arc ...
```

```
net 1 #pins blockID pin ... blockID pin
net 2 #pins blockID pin ... blockID pin
net 3 #pins blockID pin ... blockID pin
...
net N #pins blockID pin ... blockID pin
```

© R. Rutenbar 2001

CMU 18-760, Fall01 107

Timing Spec in Input File

▼ About that first line:

```
#blocks #nets timingSpec
```

▼ timingSpec is an integer

- ▶ timingSpec < 0 => just ignore timing completely
- ▶ timingSpec == 0 => just try to minimize overall worst critical path
- ▶ timingSpec > 0 ==> this is T, the target timing you should try to meet

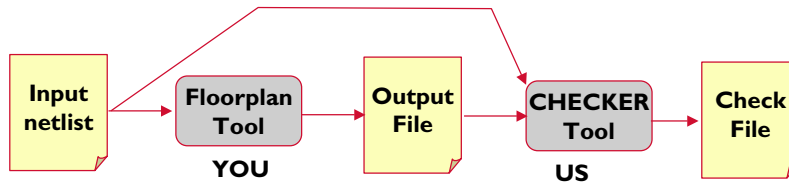
© R. Rutenbar 2001

CMU 18-760, Fall01 108

Output File Format

Philosophy

- ▶ You read the netlist, do timing-driven placement, write a file out
- ▶ File tells us the placement, and your numbers for area, wirelength, overlap, critical path delay, and one critical path
- ▶ We (actually, your earnest, hardworking TAs) provide a CHECKER tool



- ▶ CHECKER tells you if your placement is OK, if your area, wirelength, overlap, critical path delay, critical path are indeed CORRECT
 - ▷ Very useful for your debugging
 - ▷ Major pain in the butt for us to build (go hug a TA...)

© R. Rutenbar 2001 CMU 18-760, Fall01 109

Output File Format

Simple, minimal (nothing not already lying around in placer)

```

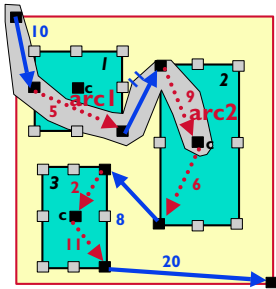
<Σnetlengths number>
<Σpairwise block-block overlap area number>
<overallArea number>
<overallCriticalPathDelay number>
block 1 centerX centerY rotation shape
block 2 centerX centerY rotation shape
...
block B centerX centerY rotation shape
net 1 length
net 2 length
...
net N length
path #edges
}
<edge type> edgeID
<edge type> edgeID
...
<edge type> edgeID
  
```

Overlap?
Ought to be 0
always for your
slicing tree floorplanner.

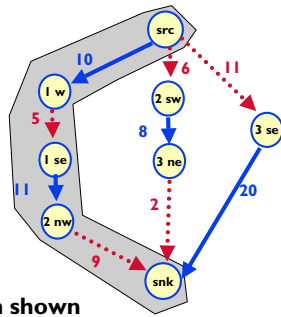
If you did this like in
HW4, then you could get
some "residual"
overlaps in final
floorplan. We check
for this, anyway.

© R. Rutenbar 2001 CMU 18-760, Fall01 110

Output File Format Example



Critical path shown shaded here in this placement



Output file

```
49
0
area
35
block 1 cenX cenY rot shape
block 2 cenX cenY rot shape
block 3 cenX cenY rot shape
net 1 10
net 2 11
net 3 8
net 4 20
path 4
net 1
arc 1
net 2
arc 2
```

© R. Rutenbar 2001

CMU 18-760, Fall01 111

For Credit

Logistics

- ▶ You can work in groups of 2 or alone. Other ideas -- ask RAR

Code

- ▶ You will write a slicing tree floorplanner.
- ▶ Optional – you can handle timing. OK to NOT to do it.
- ▶ Your choice on platform, language
- ▶ BUT, it has to be something WE can get to, so YOU can demo for US

Checking

- ▶ YOU will run the CHECKER, dump its output into your writeup
- ▶ This determines how well your program did (both correctness, and competitive results against others in class)

© R. Rutenbar 2001

CMU 18-760, Fall01 112

For Credit

▼ Writeup

- ▶ **Not paper. Web page.** You submit it to us end of class.
- ▶ **PLEASE** make it portable: we copy the whole directory structure to our 760 web pages. If you put absolute pathnames, links, it messes up
- ▶ **Suggestion**
 - ▷ **Make a directory:** <yourname>760Web, eg, *bubba760Web*
 - ▷ **Inside it, put all your html web pages:** foo*.html
 - ▷ **Inside it, also make 2 directories:** 760Stuff and 760Code
 - ▷ **Inside 760Stuff, put ALL your graphics and pics and sounds and explanatory video clips, etc.** Inside 760Code, put all your code.
 - ▷ **Use only relative link names for internals:** ./760Stuff/foo.gif etc
 - ▷ **If its on the machine in your dorm room, and it will disappear before break--TELL US WHEN.**
 - ▷ **If we don't see a web page, you don't get a grade...**

For Credit

▼ About Writeup--basic pieces

- ▶ **Introduction:** summarize the problem
- ▶ **Formulation:** you had to make some assumptions, since there are lots of degrees of freedom in this project. Explain them. Justify them.
- ▶ **Optimization goals:** tell us what you tried to do well.
- ▶ **Implementation:** describe any interesting data structures, algorithms, optimizations, tricks, etc
- ▶ **Results:** what did you run, how well did you do?
 - ▷ Think neat tables, plots, pics of layouts, graphs of cost vs temp, etc
 - ▷ Explain your results: why did they happen like this
- ▶ **Post mortem:** given you could do it over, what would you do different?
- ▶ **Code:** put it someplace in the web page (preferably in 760Code dir)

For Credit

▼ You have to *demo*, too

- ▶ Last week of class on a couple days--signup sheets
- ▶ We will release some new benchmarks during the demo, and ask you to run one. It will be small; available in a couple of flavors.
- ▶ You should print (or, better, *draw*) something enlightening
- ▶ You run the **CHECKER**, we look over your shoulder and see what it says
- ▶ Goal: *it works, it gives an OK answer.*

Points = [120] (But Weighted Like Proj2 Overall)

▼ Breakdown

- ▶ [30 pts] Web Writeup: Approach & Implementation
- ▶ [30 pts] Web Writeup: Results & Analysis
- ▶ [10 pts] Code: Reasonableness
- ▶ [20 pts] Demo: Works, Quality, Style, Discussion
- ▶ [30 pts] Coolness
 - ▷ You actually got the thing to work (nice floorplans)
 - ▷ Results quality (bigger, better, faster, etc)
 - ▷ Interesting algorithms (more sophisticated annealing, you actually **DID** the timing, interesting coupling of timing to layout, etc)
 - ▷ Interesting implementation (eg, did it in **JAVA**, but its *not* slow...)
 - ▷ Graphics (animated like **RAR**'s placer videos)

Benchmarks

▼ Will appear in /afs/ece/class/ee760/proj3/benchmarks

▼ 3 level of test cases

- ▶ Level 0: no timing at all, just pack the blocks, minimize wirelength, area; blocks have only one shape apiece; you can ignore rotations of the blocks to get a good layout
- ▶ Level 1: level-0, but blocks can have multiple shapes, and you need to do rotations to get a good layout
- ▶ Level 2: level-0 geometry, but now we have timing arcs too
- ▶ Level 3: whole shebang -- placement, shapes, rotations, timing arcs

▼ Size

- ▶ 5-50 blocks, 5-100s of nets, 5-100s of timing arcs
- ▶ 3 different timing optimizations: none, minimize, and hit-target-timing

Graphics

▼ Just mazingly helpful for a layout tool

- ▶ It's very hard to debug a layout algorithm if you cannot SEE it run
- ▶ Also, more points for some animation

▼ Use cmuview2 tcl code

- ▶ (You can use whatever you like here: JAVA, etc, is fine too)
- ▶ Think about drawing floorplan every K moves, or end of each temp
- ▶ Think about drawing the wires, and critical path
- ▶ Think about intelligent use of colors
 - ▷ blocks, nets, rooms on floorplan, pins, nets on critical path, etc.
 - ▷ You will amazed how useful this can be...

Code Complexity

Basic floorplanner

- ▶ Parsing: moderate pain
- ▶ Annealer for floorplanner is pretty straightforward
 - ▷ Use skeleton from TSP problem and HW4 placer problem
 - ▷ New stuff is the slicing tree data structure, moves, sizing, and shape functions if you choose to do them

Timing component

- ▶ Building timing graph: messy book-keeping, but conceptually OK
- ▶ Longest path: not too bad, you have to **THINK** how you will get not just the length, but the nets on this path as well
- ▶ Coupling to annealing placer:
 - ▷ Easiest is probably to update graph every K moves or every Temp
 - ▷ Easiest is probably to just treat maxDelay as an objective to min
- ▶ Graphics: once past brief learning curve, not hard to do something simple like dump blocks/nets as boxes/lines to screen
 - ▷ Just like HW4 placer

© R. Rutenbar 2001

CMU 18-760, Fall01 119

Where Are We?

About a month to do this--more if it drags over into finals.

	M	T	W	Th	F	
Aug	27	28	29	30	31	1
Sep	3	4	5	6	7	2
	10	11	12	13	14	3
	17	18	19	20	21	4
	24	25	26	27	28	5
Oct	1	2	3	4	5	6
	8	9	10	11	12	7
	15	16	17	18	19	8
	22	23	24	25	26	9
	29	30	31	1	2	10
Nov	5	6	7	8	9	11
	12	13	14	15	16	12
Thnxgive	19	20	21	22	23	13
	26	27	28	29	30	14
Dec	3	4	5	6	7	15
	10	11	12	13	14	16

Introduction
 Advanced Boolean algebra
 JAVA Review
 Formal verification
 2-Level logic synthesis
 Multi-level logic synthesis
 Technology mapping
 Placement
 Routing
Floorplanning (Project 3)
 Static timing analysis
 Electrical timing analysis
 Geometric data structs & apps

Want demos, web writeup by 14th

© R. Rutenbar 2001

CMU 18-760, Fall01 120