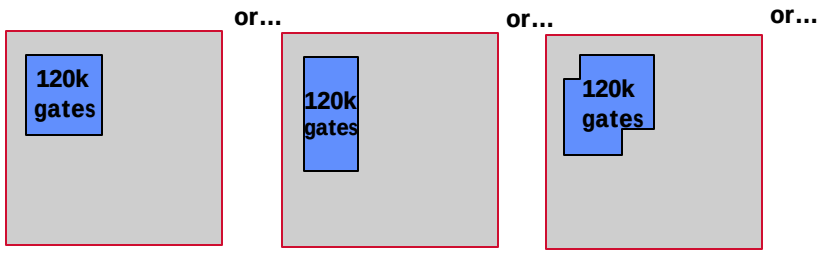
(*Project3*) Timing-Driven Floorplanning

- Input
  - A netlist of malleable rectangular blocks, nets connecting them, and "timing arcs" for block delay
- Output
  - A placed floorplan for the blocks, and information about overall area, netlength, critical path timing
- Strategy
  - Combine annealing placement ideas 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

Copyright Notice

About Floorplanning

- Floorplanning is placement of "complex" blocks
  - In general, arbitrary shaped blocks, and flexible blocks
  - Flexible = you don't know the final shape of the blocks
  - Example: a block that will be ~ 120,000 gates can get laid out in many different shapes when you actually place/route std cell gates in rows



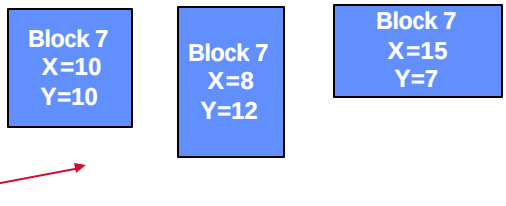
Our Block Model for Floorplans

- 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

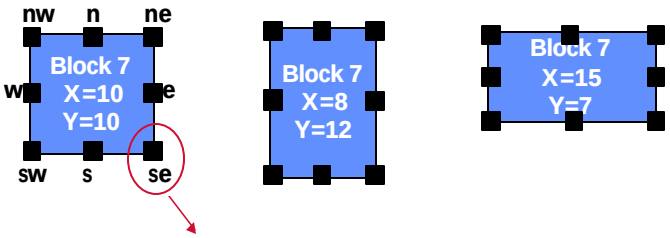
…
block 7 3 10 10 8 12 15 7
…

Block #7 has 3 rectangular shapes, shown at right



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 1char & 2char lower case names



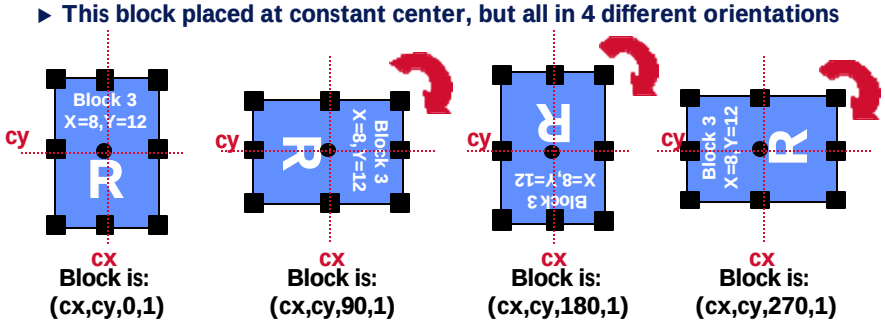
This pin is referred to as the pair <blockID, pintype> which is "7 se" for this pin, in the input file

Our Block Model for Floorplans

- Blocks can be placed anywhere on chip
  - Blocks have integer width (x) and height (x) for all shapes
  - Chip itself is an integer grid: blocks can be placed anywhere on 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 of a placed block, we use the order in which shapes were listed in input netlist: 1, 2, 3, … A block with 1 fixed shape gets a "1"

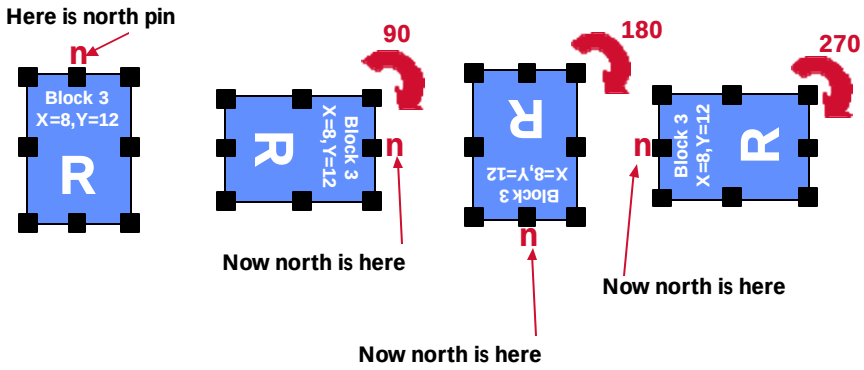
Our Block Model for Floorplans

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



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

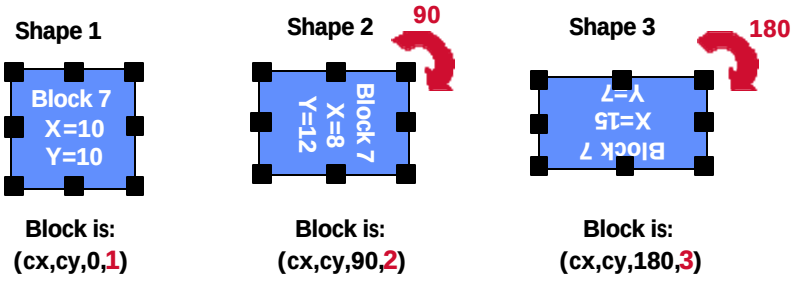


Our Block Model for Floorplans

- What if there are more shapes?

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

3 different shapes for block 7



Block is: (cx,cy,0,1)

Block is: (cx,cy,90,2)

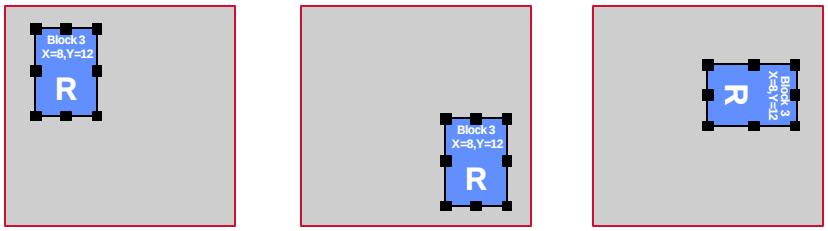
Block is: (cx,cy,180,3)

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 DX and DY values you need to add to the (centerX,centerY) location of the block to compute location of pin
  - These (DX, DY) 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

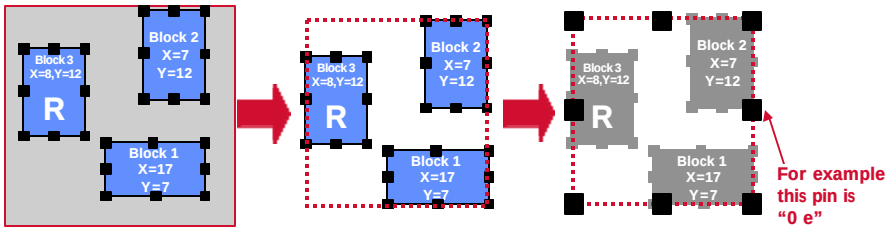
Our Chip Model for Floorplans

- Example:
  - Chip itself is an integer grid, blocks can go anywhere
  - Question we deal with later: so, how big is the chip? We don't know yet, since we don't have the floorplan…



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 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



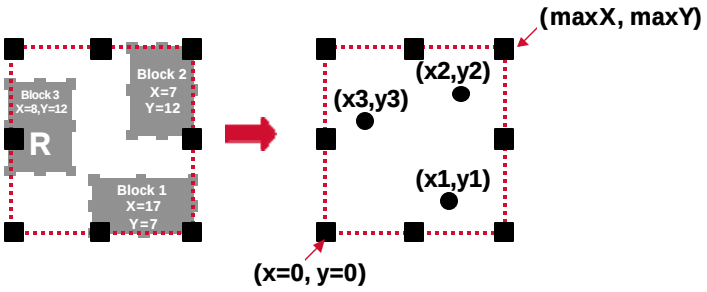
Chip == ??

Chip area = min bounding box of whole placement

Chip referred to as 'Block 0', it has 8 pins like any block

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; while its evolving, you will probably want to use a different coord system; more later on this



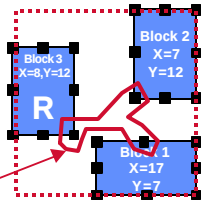
Our Net Model for Floorplans

- A net is just a set of 2 or more pins
  -
  - 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

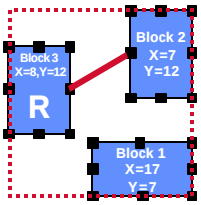
…
net 6 3 3 se 2 sw 1 n
…

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

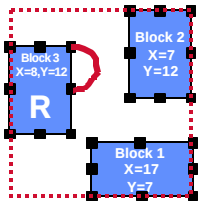


Our Net Model for Floorplans

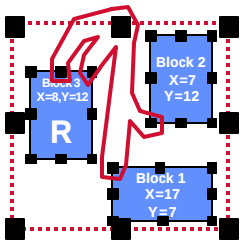
- Examples



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 1 nw s sw

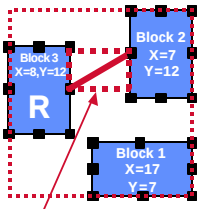
Our Net Model for Floorplans

- 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

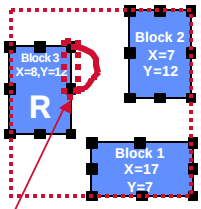
- Netlength model
  - Simple: 1/2 perimeter metric for each net
  - Total netlength = add them all up = $\mathbf{S}_{\text{(all nets i)}}$ (net length i)
  - Pins are modeled as a single dimensionless point: a pair of ints
  - Find leftX, rightX, topY, bottomY for all pins on your net #i
  - 1/2 perimeter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

Our Net Model for Floorplans

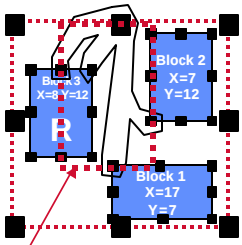
- Net length examples



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 + ΔY in this case



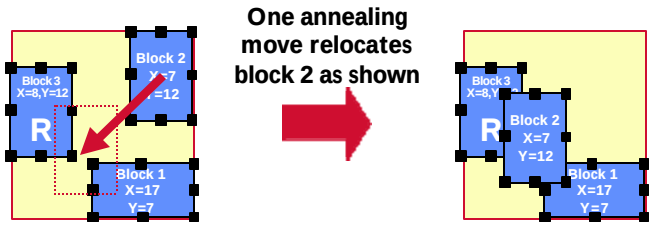
Length for this 5pt net is also box $\Delta X + \Delta Y$. It's a much bigger box now, And remember that the chip pin is on the top, at X center, Y top coord of the layout bounding box

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 placement legal (ie, blocks do not overlap)
  - Make chip area small
  - Make total netlength small
- How?
  - There is nice, fairly simple annealing formulation for this

Floorplanning by Annealing

- One cute idea
  - Treat this like "ordinary" placement we did in lecture, where annealing relocates the blocks, and minimizes wirelength
  - But, big problem: these are not like checkers on a checker-board, there are no grid-slots to ensure these complex shapes pack right.
  - What happens if the blocks overlap?



- Big idea: let them overlap

Annealing Formulation

- Allow overlaps, but use cost function to discourage them
- In any cost function C = $\mathbf{S}_i$ Ci, 2 distinct kinds of terms Ci
  - Objectives:
    - You don't know the right final answer, but you know you want to make this term small
    - Example: area of layout
  - Constraints
    - You DO know the right final answer, so you construct the term Ci to penalize the wrong answer
    - This is called: doing optimization with *penalty* functions
    - Example: overlap among blocks. You know bad==positive num. You know overlap is never negative. You know good==0 overlap. So, you construct Ci = overlap in such a way as to heavily favor the no-overlap solution you want
    - It's a *delicate* business to do this right…

Annealing Formulation

- Suggested cost function
  - Wa, Wn, Wo empirically chosen weights to balance terms in cost

$$\text{Cost} = Wa * [\text{Area}] + Wn * [\text{Netlength}] + Wo * \left[\sum_{i \neq j} \text{overlap}(\text{blocks } i, j)^2 \right]$$

Objective: *Make area of whole chip (block #0) =small*

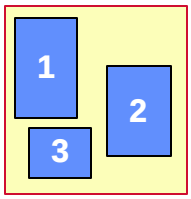
Objective: *Make* $\sum$ *netlen's =small*

Constraint as penalty function: *Compute and sum up all pairwise squared-overlaps between blocks. This "squaring&summing" is the penalty "function". The "best" value we can get for this individual term = 0.*

  - Why it works: makes area and netlength small, makes overlap -> 0

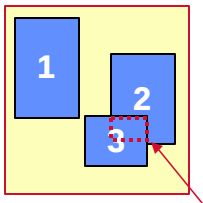
Overlap Penalty Function

- Simple idea: Calculate each pairwise overlap, square it, sum



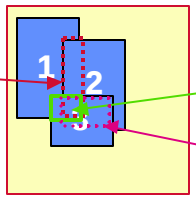
No overlaps

Penalty==0



1 overlap:
Overlap(1,2)=0
Overlap(1,3)=0
Overlap(2,3)^2=box ^{2}

Add them up

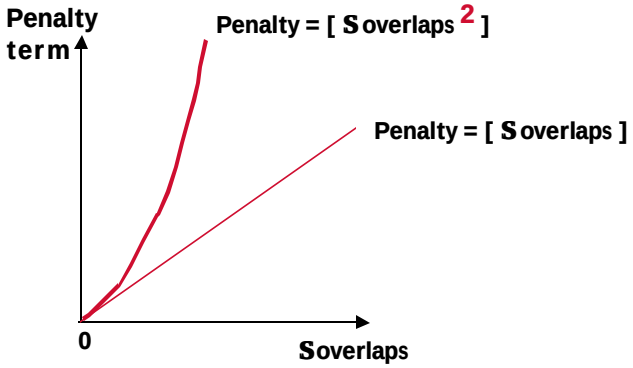


3 overlaps
Overlap(1,2)^2 =box^2
Overlap(1,3)^2 =box^2
Overlap(2,3)^2 =box^2

Add them up

Overlap Penalty Function

- Why "square it"?
  - Annealing folklore: if you have a cost term and you really want to drive it to 0, squaring it makes it "hurt more" in the cost function (esp small overlap terms)
  - So, it "sticks out" in the cost, and the annealer works to minimize it

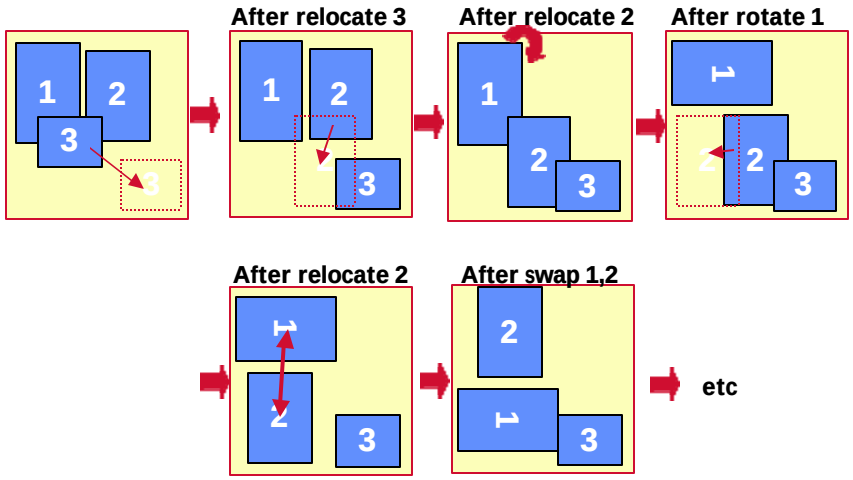


Basic Annealing Formulation

- Remember: all annealers have 4 parts
- State: what is the representation of floorplan?
  - Just (*centerX, centerY, rotation, shape*) for all B blocks
  - Overlaps are allowed
- Cost: what do want to minimize to measure goodness?
  - Cost = Wa *[Area] + Wn*[Netlength] + Wo*[$S_{i \neq j}$ overlap(blocks i, j)^2]
- Moves: how do we perturb the floorplan?
  - Pick random block, relocate it to new (centerX, centerY)
  - Pick random block, reshape, from shape #i to shape #k; same centers
  - Pick random block, rotate it, from rotation R to rotation R'
  - Pick 2 random blocks, swap their center locations
  - ...others are possible, but this is a minimal OK set
- Cooling schedule: how we control hill climbing?
  - Vanilla stuff from lecture notes and sample code is fine

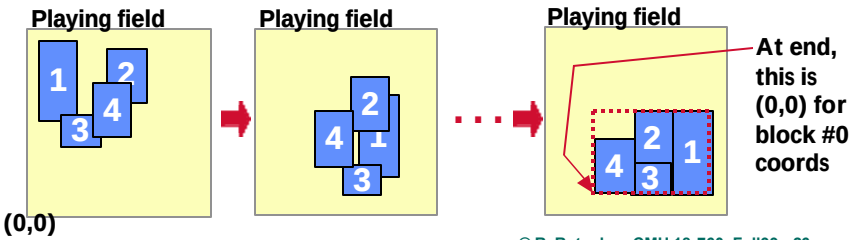
Basic Annealing Formulation

- Example moves



Basic Floorplanning: Implementation Hints

- About the coordinate system
  - This formulation works best if you give the floorplan a lot of "space" to evolve in, a space much bigger than the final layout
  - Make a box that has area ~2X bigger than S (block areas)
  - Conventionally called the *playing field;* lower-left corner is (0,0)
  - Use (centerX, centerY, rotate, shape) coords on playing field for blocks
  - ...ie, blocks will fly around, find good topology, then squish down
  - Where the final layout ends up is random, but overall packing should be good; at end, print out "normalized" Block #0 coords for all blocks



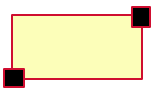
Basic Floorplanning: Implementation Hints

- Cost function
  - It's going to hurt (CPU time) if you re-evaluate every net length after every move, and every overlap after move
  - This IS the easiest way to get started to get functionaly, but slowest code will result
- Tricks
  - Only re-calculate the length of the nets that are attached to blocks that *move*. Store with each net it's "current" length, so you can subtract out the "old" S netlens, and add in "new" S netlens quickly for a move
  - It's OK to do compare each block against other blocks for overlap (don't double count though); slow but easy to code
  - Ask RAR for tricks on how to do the block-block overlap faster…

Basic Floorplanning: Implementation Hints

- How do you compute overlap of 2 boxes?
  - Fast hacks: see any good computer graphics book (Foley & Van Damm), look up "outcodes"; these guys know tricks to do it FAST
  - Vanilla way:
    - a, b, c are rectangle objects, with data items:

      llx (lower left x), lly (lower left y),
      urx (upper right x), ury (upper right y)



    - Overlap(a, b, c) does a = intersect(b, c), computes area of a

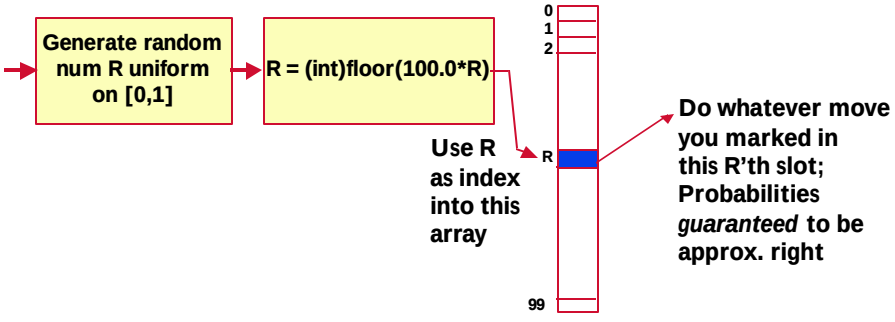
```
Overlap(b,c)        {
   rectangle a;
   //  try to build a = overlap rectangle itself
   a.llx = Max( b.llx, c.llx );
   a.urx = Min( b.urx, c.urx );
   a.lly = Max( b.lly, c.lly );
   a.ury = Min( b.ury, c.ury );

   if(  (a.llx > a.urx) || ( a.lly > a.ury) ) {
      // they don't really overlap
     return (0);
   }
   else return (  (a.urx - a.llx) * (a.ury - a.lly) )
}
```

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:
    - *Relocate* block
    - *Swap* 2 blocks
    - *Rotate* block
    - *Reshape* block
  - We want 4 fractions F_{reloc}, F_{swap}, F_{rot}, F_{shape} that sum to =1
  - We want to guarantee that if we do N moves at this temp, that:
    - ~ N* F_{reloc} block relocations get tried
    - ~ N* F_{swap} block swaps get tried
    - ~ N* F_{rot} block rotates get tried
    - ~ N* F_{shape} block reshapes get tried

Basic Floorplanning: Implementation Hints

- Easy trick
  - Suppose you want: F_{reloc} =50% F_{swap} =20% F_{rot} =30% F_{shape} =20%
  - Make an array with 100 entries
  - In the first 50 entries, put a marker that says "do relocate"
  - In next 20 entries, put a marker for "do swap"
  - Ditto remaining entries: next 30 = "do rotate", last 20 = "do s

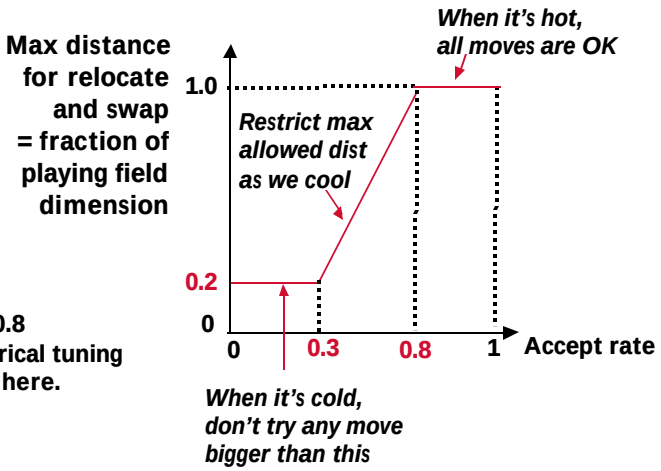


Basic Floorplanning: Implementation Hints

- Think about range limiting: it helps speed/quality a lot
  - Try not to propose moves that have a high probability of being rejected, because they perturb layout too much
  - Easiest one to do: *shrink* the max DISTANCE you are willing to try to do a relocate or a swap as the temp T gets colder
  - You need some normalization hints or this is hard--have to tweak for each problem
- Normalization
  - It is mechanically easier to make this distance a function of *accept rate* (= #accepted moves / #tried moves at this temperature)
    - Reason: accept rate ALWAYS starts ~1, goes to ~0
    - Temperature will vary widely with different problems
  - It is mechanically easier to make this distance itself a *fraction* of the size of the max distance across the playing field

Basic Floorplanning: Implementation Hints

- Range limiting
  - Example: with normalization, you can set this once and forget about it



Note: 0.2, 0.3, 0.8 are empirical tuning numbers here.

Basic Floorplanning: Implementation Hints

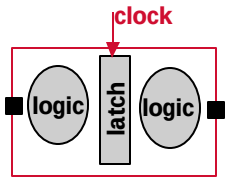
- Range limiting: gotchas
  - If you don't do it: your annealer is WAY slow, since you need to do a zillion moves to get a decent answer
  - If you do it, but you don't range limit "hard" enough: annealer still slow, still do a lot of dumb moves
  - If you do it, but range limit too fast, too "tight": annealer is fast but answers are always lousy, since you are precluding moves you really wanted to do
  - Need to do some empirical tuning on the shape of the range limiting function on previous slide

Floorplanning -> Timing

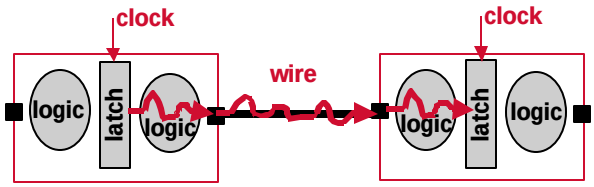
- Project goals
  - First goal is to be able to get a decent floorplan:
    - Packed, small area, small wirelength, no overlap
    - (or, not much overlap--hard to make it 0 without more fancy stuff)
  - Next goal: good timing
- We also have a timing model
  - Each block has a timing model: *timing arcs*
  - 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…

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

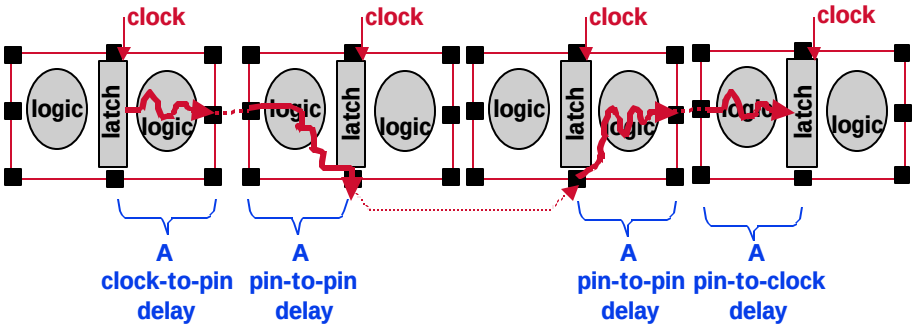


Our Timing Model for Floorplanning

- 4 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)

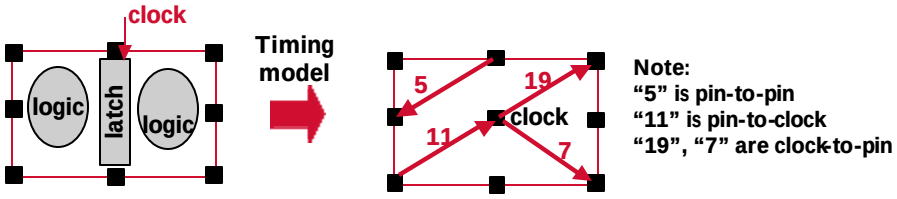
Delays Thru a Block

- 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?



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



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

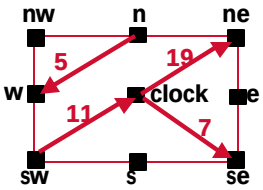
Delays Thru a Block

- 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:

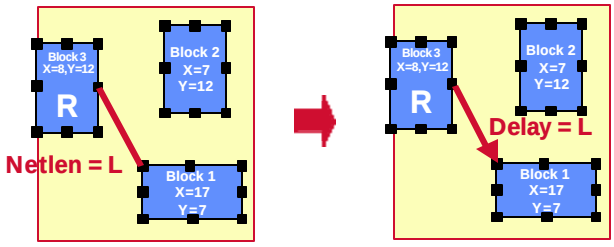
```
…
block 7  3   10 10   8 12   15 7
timing  4
arc  21   n w 5
arc  22   sw c 11
arc  23   c se 7
arc  24   c ne 19
block 8 …..
timing ….
arc …
```

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



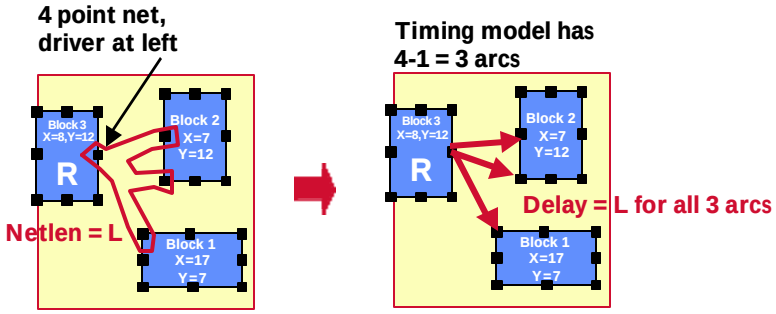
Delays Thru a Wire

- Longer wires have longer delay
  - How do we model this?
  - Crudest possible model: 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



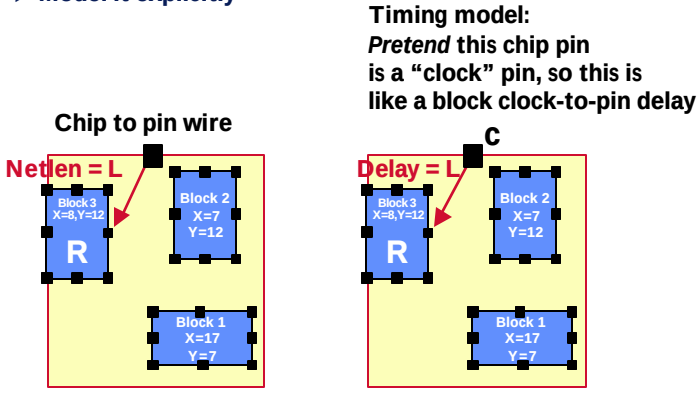
Delays Thru a Wire

- 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



Delay Thru Wires to Chip Pins

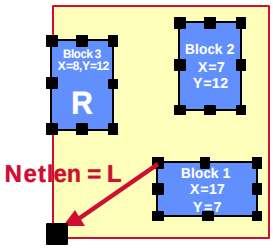
- New problem: how to model wires to chip IOs?
  - Question is: where is the "clock" for these external signals
  - Turns out there is a standard assumption: external signals use same clk
  - Model it explicitly



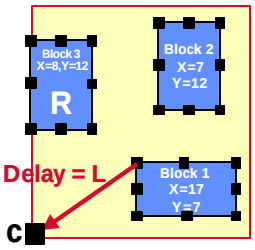
Delay Thru Wires to Chip Pins

- Ditto for block-pin to chip
- Note: to make life easy, these nets are always 2 point nets

Chip to pin wire

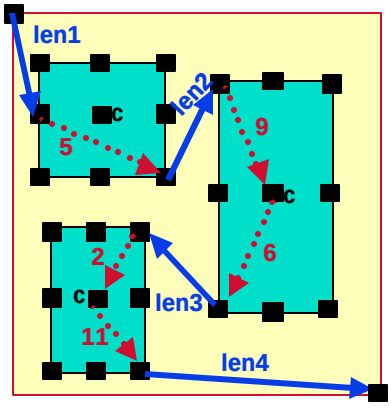


Timing model:
Pretend again this chip pin
is a "clock" pin, so this is
like a block pin-to-clock delay



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

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

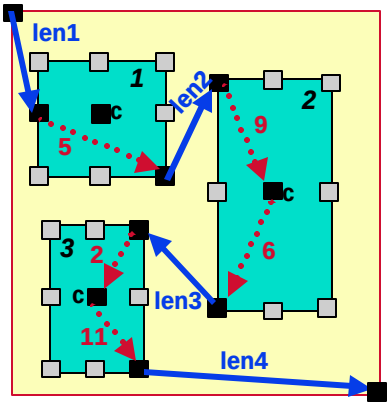
Edges for each net, and for each timing arc

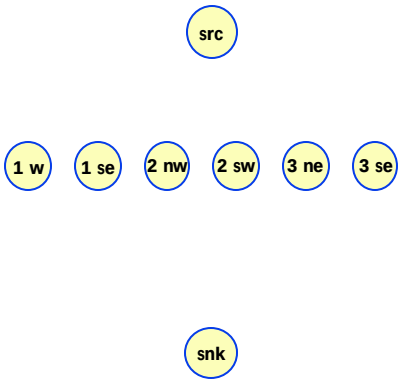
A lot of nodes and edges, but we are guaranteed the overall graph is a DAG -- no cycles

snk — One distinguished "end" node, called "sink"

Handling Critical Paths

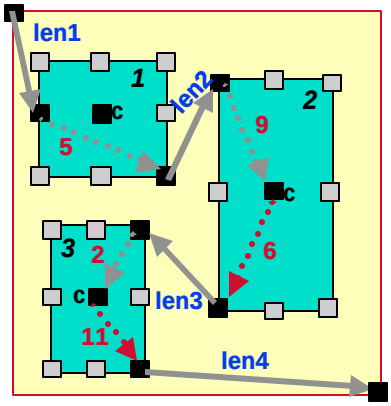
- Step 1. Build all the nodes in graph
  - One per block pin that is connected to a net
  - (no clocks now)

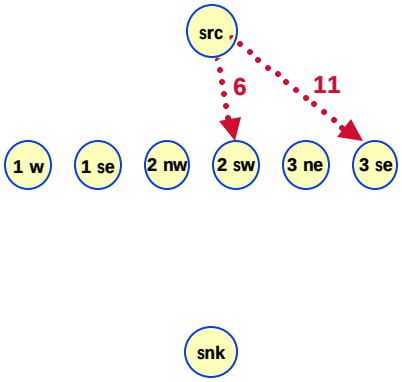




Handling Critical Paths

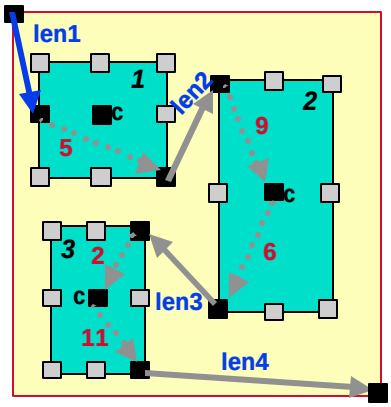
- Step 2. Clock-to-pin edges
  - For every timing arc FROM a clock node TO a block pin, add an edge in graph FROM source TO correct pin node

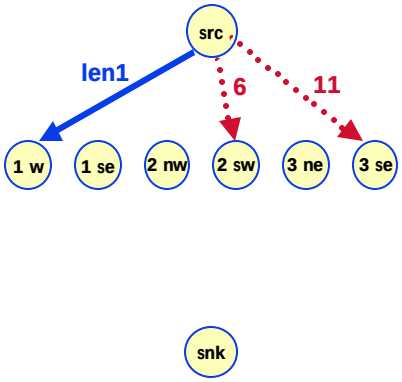




Handling Critical Paths

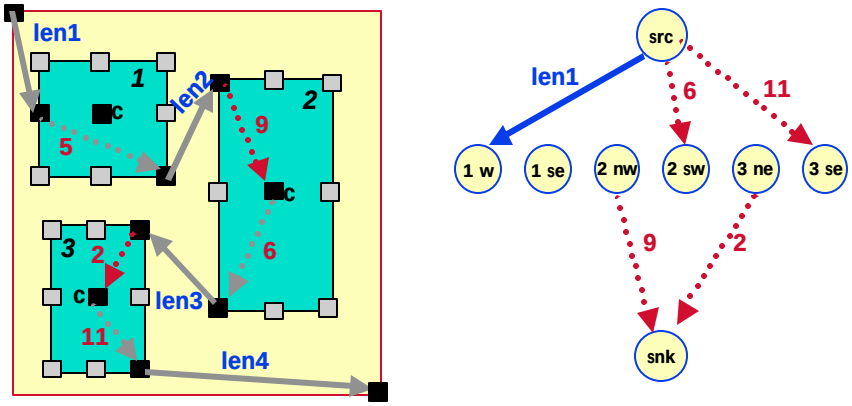
- Step 3. Chip-to-pin edges
  - For every net FROM a chip pin TO a block pin, add an edge in graph FROM source TO correct pin node





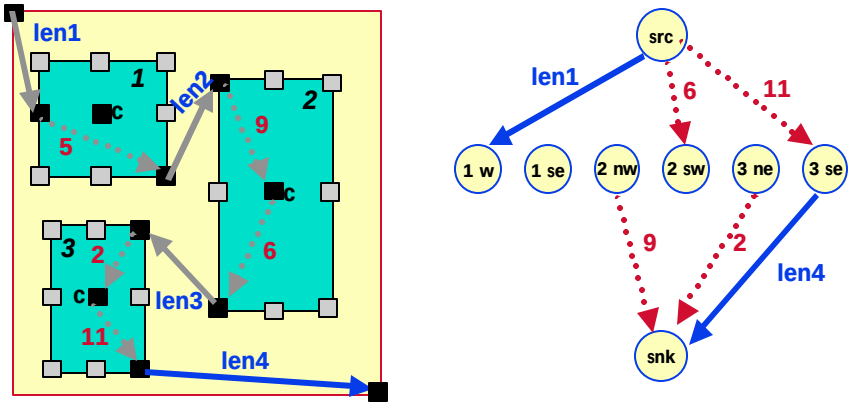
Handling Critical Paths

- 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



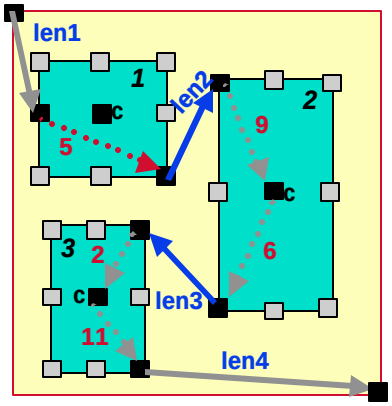
Handling Critical Paths

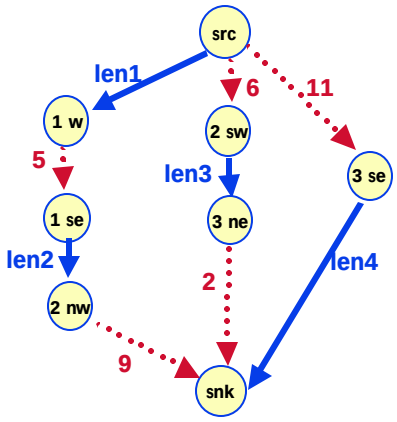
- Step 5. Pin-to-chip edges
  - For every net FROM a block pin TO a chip pin, add an edge in graph FROM correct pin node TO sink



Handling Critical Paths

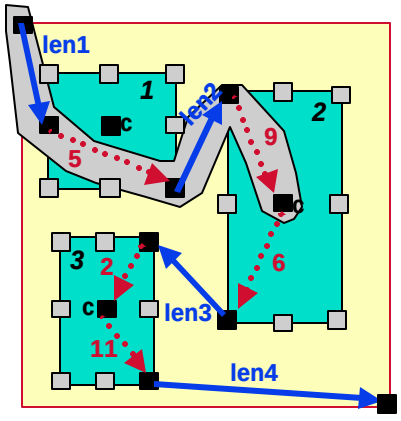
- Step 6. Pin-to-pin edges
  - For every net and every arc FROM a block pin TO a block pin, add an edge in graph FROM correct pin node TO correct pin node

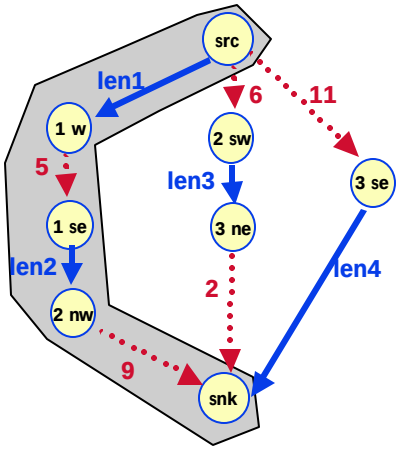




Handling Critical Paths

- Done. This is the required timing graph
  - Longest path form Source to Sink == worst-case delay, latch-to-latch
  - One path highlighted below





Observations

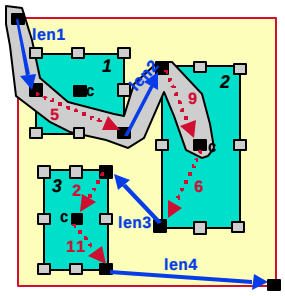
- Graph structure is constant--you only build it once
  - Same nodes, same edges, always
- Timing arcs (dotted edges) are constant
  - Placement does nothing to change intra-block timing in our simple model of floorplanning
- Placement changes the net delays (solid edge nums) in graph
  - Move a block, 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

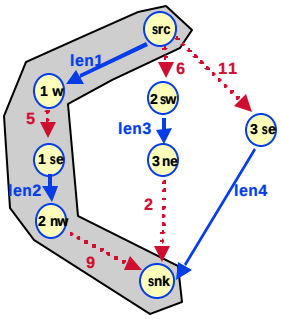
Observations

- 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 **DS** (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…)

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 Dtiming on subsequent placement moves?
  - Dtiming == D(len1 + len2) !! That's it. Very nice, very simple.

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).

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:

$$\text{Cost} = Wa*[\text{Area}] + Wn*[\text{Netlen}] + Wo*\left[\sum_{i \neq j} \text{overlap}^2\right] + Wt*[\text{max Delay}]$$

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

$$\text{Cost} = Wa*[\text{Area}] + Wn*[\text{Netlen}] + Wo*\left[\sum_{i \neq j} \text{overlap}^2\right] + Wt*[\text{TimeMiss}]^2$$

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

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
```

Timing Spec in Input File

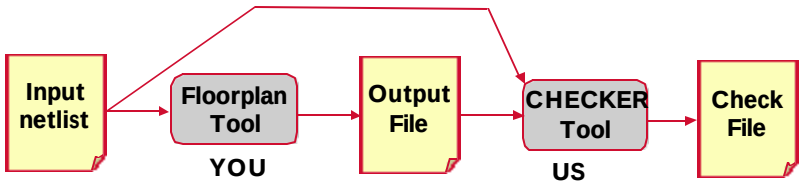
- About that first line:

  #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

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…)

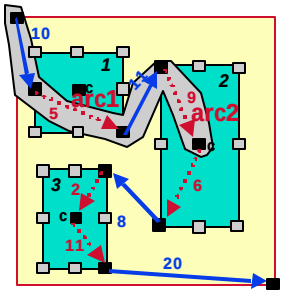
Output File Format

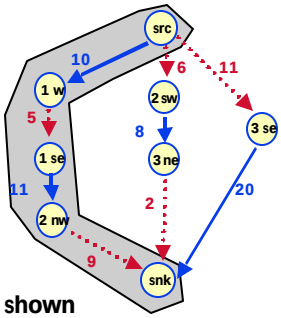
- Simple, minimal (nothing not already lying around in placer)

```
<Snetlengths number>
<Spairwise 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
```

<edge type> is either net or arc

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
```

For Credit

- **Logistics**
  - You can work in groups of 2 or alone. Other ideas -- ask RAR
- **Code**
  - Your will write a timing-driven placer.
  - 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)

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…
  - Check out last Spring's offerings on current 760 page; style counts

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 Big 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 whole thing to work (place, timing, etc)
    - Results quality (bigger, better, faster, etc)
    - Interesting algorithms (more sophisticated annealing, 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

- Are a pain to do, but amazingly helpful
  - It's very hard to debug a layout algorithm if you cannot SEE it run
  - Also, more points for some animation
- We can help
  - We will put some graphics code on class acct
  - C or C++ plus tcl/tk stuff: fires up a window, can draw boxes, lines, circuits, text, in colors. Pretty simple to use.
  - (You can use whatever *you* like here: JAVA, etc, is fine too)
  - Think about drawing placement every K moves, or end of each temp
  - Think about drawing the wires, and critical path
  - Think about intelligent use of colors (blocks with overlap vs no, nets on critical path vs no, etc. You will amazed how useful this can be...)

Code Complexity

- Not as bad as it sounds
  - Parsing: moderate pain
  - Annealer for floorplanner is pretty straightforward
    - And, we already hand out *source code* for a *complete* point-placer which you can use as starting point here
  - 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
    - We will also give you *source code* for some options here

Where Are We?

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

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

A partner is a *very* good idea.

Be *clear* about your goals; when in doubt, ask RAR.

Expect more updates on benchmarks, etc, later

Look at the chapter from *Cohn,Garrod,Rutenbar,Carley* In the hardcopy handout with this lec about how to do an annealing floorplanner like this one: LOTS of relevant details

