18-760 Fall97 VLSI CAD Project 3: Timing-Driven Placement

Out: Wed Nov 5, 1997. **Due: Wed, Dec 3, 1997**

1.0 Background

You now know enough to actually develop a gate-level placement program using annealing-style iterative improvement. But, to make it interesting, we will add two new twists:

- Your placer will use a somewhat more abstract partitioning-style model.
- Your placer must not only place each gate/pad, and minimize the wirelength, it must also *optimize* a select set of critical paths derived from a static timing analysis.

Your assignment is to implement a program that can take a netlist provided by us, a description of the geometry of the chip and the timing paths to optimize, and decide which gate goes where to minimize the overall wirelength and optimize the timing.

2.0 Modeling the Problem

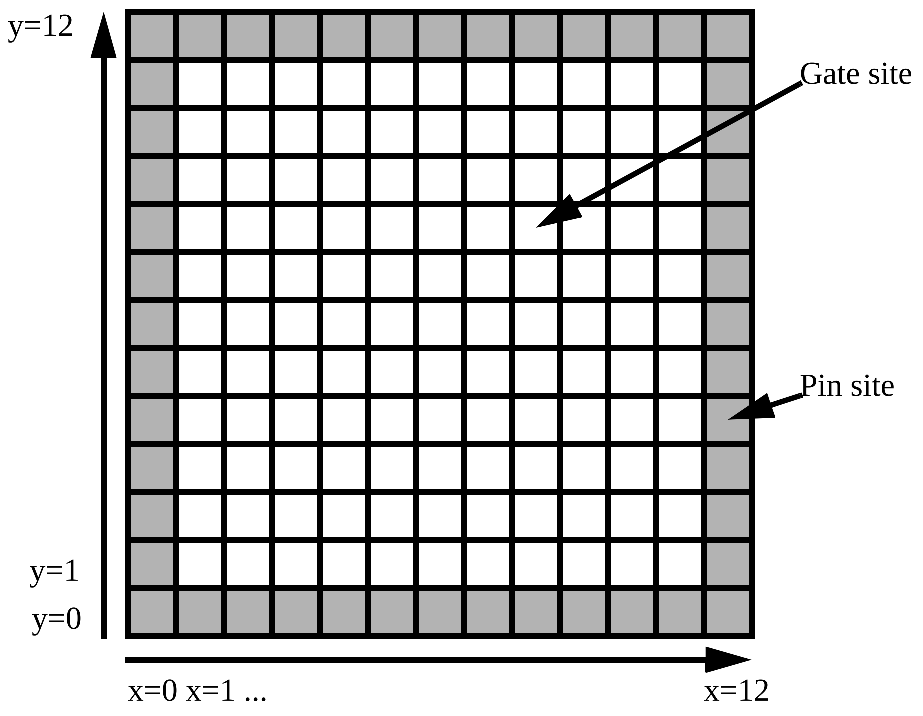
We describe here the geometric model of the *chip* itself, and the appropriate ways to model the *wires* and the *delay*.

2.1 The Chip

For simplicity, we shall model all geometric objects in this problem as being rectangles whose dimensions are multiples of some unspecified "fundamental unit". Everything looks like some sort of simple checkerboard, where the individual squares have certain properties.

In our model, the overall chip is a rectangle with *pin sites* at its periphery, as shown in the figure on the next page.

Individual gates drop onto this grid at arbitrary locations (see next subsection). The chip itself has pins, but they are restricted to be around the outside of the rectangle. Our model is using a fairly *coarse* grid--it is a sort of detailed partitioning model and not a complete, exact placement of each gate. We allow each gate site to hold some maximum number of physical gates. This number is the *capacity* of the site. We shall also allow each pin site to hold more than one physical pin. The gate capacity (how many gates you can put in a gate site) and pin capacity (how many pins you can put in a pin site) are read in from the input file.



In the above example, the chip is 13 fu X13 fu (fu=fundamental unit), with (x=0,y=0) modeled as being the cell at the lower left. There are on this chip 121 gate sites and 48 pin sites. If we allowed 10 physical pins per pin site, and 100 physical gates per gate site, this chip could have 10X48=480 actual pins and 121X100 = 12,100 gates if it was fully populated.

2.2 The Gates

For our problem, gates are the "atomic" placeable objects. We start with a large netlist specifying gates and nets and pins. Your job is decide the following:

1. **Which gates go where?** You have to decide in which gate site on the chip to put the gate, and you should not go over the capacity limit for any gate site when your placement in done. And, you can only put gates in gate sites, not in pin sites.
2. **Which pins go where?** Basically the same problem. The input netlist will tell you which objects are pins. You have to determine which pin site to put each pin in, and you should not violate the capacity constraint. And, you cannot put pins in gate sites.Also, pins come with constrained "edges" of the chip they must be on, ie, the input will tell you if the pin must stay on the top edge, bottom edge, left edge or right edge of this grid.

In this problem, gates are simply unit-sized objects that can be placed in internal chip gate sites. Each gate can be connected to several nets. Pins are likewise unit-sized objects that can be placed in pin sites. Most pins connect to only 1 gate but in general a pin can connect to several gates.

2.3 Modeling the Wire Length

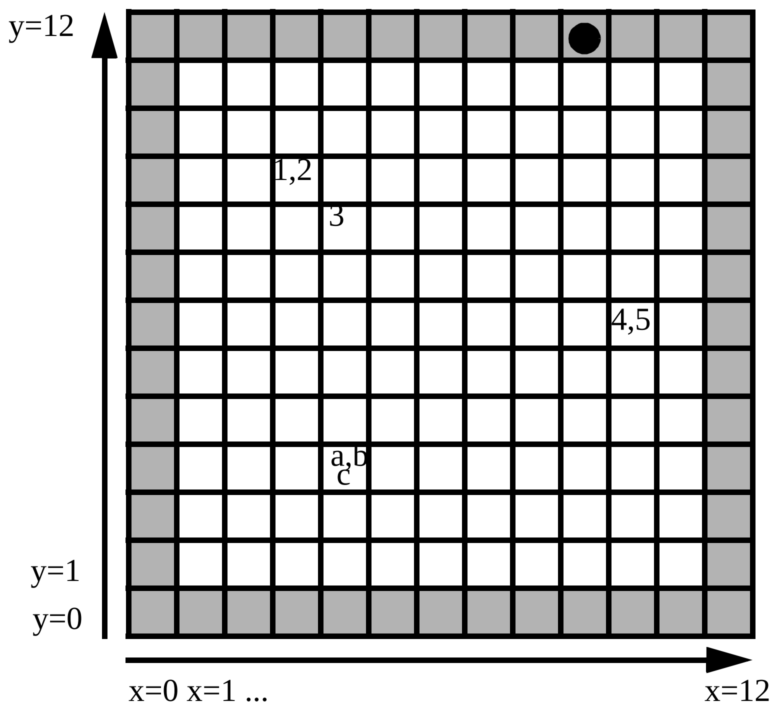
Wirelength is a little more interesting to estimate in this model, since there are really two different kinds of wires:

1. **Intra-site wires:** that connect gates located inside one single gate site on a chip.
2. **Inter-site wires:** that connect gates located in different gate or pin sites on one chip.

And notice that one single net can have parts in both of these categories.

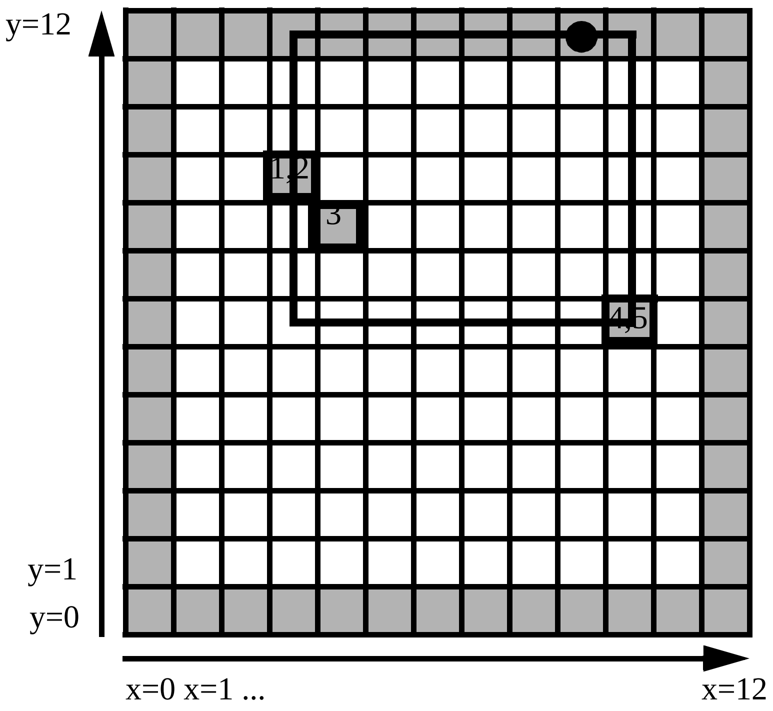
Let's look at an example to see how this works out. Suppose we have two nets in a simple problem:

- **Net N** is connected to 5 gates, numbered 1,2,3,4,5. Net N is also required to be connected to the chip pin marked by the circle in the top row.
- **Net M** is connected to 3 gates, numbered a,b,c. It is not connected to any chip pin:



Consider net N first. Assume that capacity for gates in any gate site on this chip is **G** gates per site, and that G=64 here.

- **Net N intra-site parts:** notice that gates 1,2 are in a single gate site. Each net that must be wired among multiple gates inside a single gate site contributes a constant $\mathbf{G^{1/2}}$ to the length of that wire. (Even if there were *many* gates in this single site connected to net N, the overall contribution is still modeled as the constant length $\mathbf{G^{1/2}}$.) The idea here is that since this site can hold G gates, and we treat each gate as a unit sized object, we can pretend the gate site it itself a small square grid that has $\mathbf{G^{1/2}}$ X $\mathbf{G^{1/2}}$ slots that can each hold 1 gate. In this case, each site can hold 64 gates, so we treat this as a little 8x8 grid, and each intra-site wire contributes length $\mathbf{G^{1/2}}$= 8 units to the overall wire-length. Since gates 4,5 are also inside a single gate site here, they also contribute another $\mathbf{G^{1/2}}$= 8 to the wirelength. Gate 3 is alone in its site, so we don't add any extra intra-site wiring for it. So, intra-site wires contribute "8 + 8 = 16" to the length of net N.
- **Net N inter-site parts:** we now ignore the fact that some sites have more than one gate in them, and just look at the half-perimeter of the bounding box of the sites where the net has gates, and any pins if the net needs them. Here, there are 3 separate gate sites and 1 pin site that define the bounding box. Let's redraw just the relevant portions



Overall, the contribution of the inter-site wiring is just the half-perimeter of this box, which is 6 sites high ($\Delta y = 12 - 6 = 6$) plus 7 sites wide ($\Delta x = 10 - 3 = 7$). Since we model each individual gate site as being a square of dimension $\mathbf{G^{1/2}}$ X $\mathbf{G^{1/2}}$ = 8 x 8, our total estimated length is:

(Normalized bounding box half perimeter) X (gate site size) = (6 + 7)*8 = 104

We just add up all these components of the "length" of net N--the intra-site components and the inter-site components, and we get an estimate of 8 + 8 + 104 = 120 for net N.

Consider net M next:

- **Net M intra-site parts:** net M has 3 gates in a single site, so we get a contribution of $\mathbf{G^{1/2}}$ = 8 to the total length. Again, note that in our model we don't care how many gates there are in the site or on net M in this site, it's still a constant 8 contribution here.
- **Net M inter-site parts:** none here, net M is entirely in one gate site.

So, net M has a total length of 8 in our model. Total netlength for nets N, M together is 128 in this model.

2.4 Modeling Gate and Pin Delay

Another important part of this project is that your placer is supposed to be "timing driven" which means it tries to optimize both the wirability and delay. The last section gave us the model of wire length for our placer. Now we turn to the timing models.

In this project we use the simplest possible unit delay model:

- **Pin delay:** the delay through any pin is 1 time unit.
- **Gate delay:** the delay through any gate is 1 time unit.

2.5 Modeling Wire Delay

We adopt a simple first-order model to measure the delay caused by the physical layout of a net. Since we are not actually routing any wires, all we have to go on here is the gate locations that define each net. Our model for the delay of an individual net is based on two parameters we can get for each net:

- **Estimated wirelength L**: the previous section gave our model of how to compute this.
- **Fanout F:** this is just [(total number of gates + pins on the net) - 1]. In other words, a net with 5 objects on it is assumed to be one "driving" object and 4 "driven loads", so the fanout is 4.

We use a simple model here. Given the parameters L, F for a net, the delay is:

$$\text{delay} = \mathbf{D}(L, F) = (K1 \bullet L^2) + (K2 \bullet L \bullet F)$$

where K1 and K2 are technology parameters. This is a very simple worst-case sort of a model that deals with the fact that the wiring itself causes delay (the L parts) and the more you load that wiring (the F parts) the more the delay goes up.

2.6 Modeling Path Delay

Knowing the delay through pins, gates and individual wires is important, but it's not our real problem. We want to know and to optimize how fast the chip runs, which means we really care about *path delays*, where a path starts at a pin, goes through some number of gates and wires, and ends on another pin. So, a path for us looks like this:

InputPin -> wire1 -> gate1 -> wire2 -> gate2 -> ... -> gateN-1 ->wireN ->OutputPin

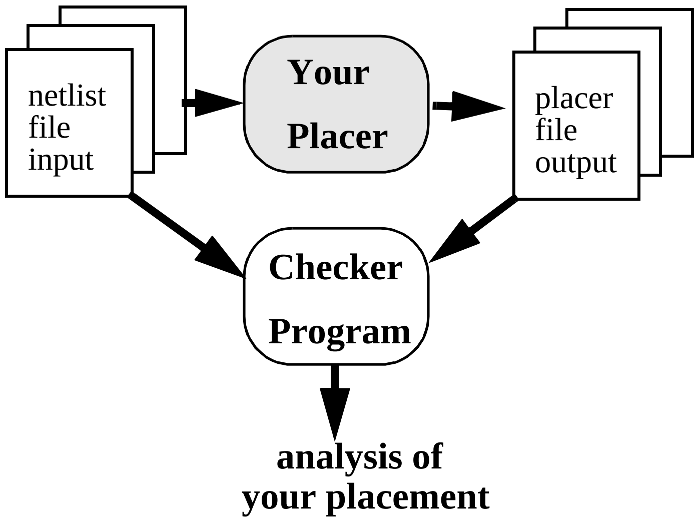
Recall that each pin has a delay of 1, and each gate has a delay of 1. The variability in delay comes from the delay through each wire, which depends on the wirelength, which depends on the placement. To compute the delay of a path, you just add up all the delays on its pins, gates and wires.

3.0 Project Function, Objectives and Constraints

Obviously, this layout problem is to be solved with certain functional goals, objectives to optimize, and constraints not to violate (if you can help it). Here we describe these.

3.1 Project Function

Here is the overall structure of the project:



We supply some netlist input files that tell the geometry of the chip, all the relevant delay and capacity parameters, and the specific paths we want you to optimize for. Your placer reads this, and generates a placement and some output information in an output file. A checker program we will supply reads these two files and lets you know how well you really did: did you violate any constraints, is your wirelength estimate correct, did you really meet timing on those paths, how about the actual overall worst case timing? etc.

3.2 Constraints

Let's begin with the constraints not to be violated.

1. Every gate in the input netlist must be placed on some gate site on the chip. You can't put them anyplace else (like pin sites), or simply lose them!

2. You cannot put more gates in a gate site than the capacity of that gate site. If a site has a capacity of 10 gates, putting 11 gates there is a violation of this constraint.
3. Every pin in the input netlist must be placed on some pin site on the chip. You can’t put them anyplace else (like gate sites), or simply lose them!
4. You cannot put more pins in a pin site that the capacity of that pin site.
5. You will be given an overall delay target for your chip, which means each of your paths that we specify should have a delay that is less than this target--if you can.

3.3 Objectives

Objectives are the things you want to optimize. You can decide yourself on what your exact priorities are, but we will be looking at the following:

- Make the total wirelength short.
- Try to make your software run in a reasonable amount of time.
- If you can actually meet the delay specs, you may try to optimize to see how fast you can actually get the paths to be.

4.0 Implementation Issues

Here we describe what the input and output files look like, what sorts of benchmark problems you have to run, and what sort of software help we will provide.

4.1 Input File Format

We keep this simple. The input file is just lines of numbers describing the geometry of all the components, and the netlist. Here is a summary of the format:

```
Xchip Ychip GatesPerGateSite PinsPerPinSite
NumOfGates NumOfNets
1 NumOfNetsOnThisGate NetID NetID ... NetID
2 NumOfNetsOnThisGate NetID NetID ... NetID
...
lastgate NumOfNetsOnThisGate NetID...NetID
NumOfPinsConnectedToNets
1 NetID PinLocation
2 NetID PinLocation
...
NumOfTimingPaths CycleTimeTarget
1 ObjectsOnPath InPinID NetId GateId NetId GateID...NetID OutPinID
2 ObjectsOnPath InPinID NetId GateId NetId GateID...NetID OutPinID
...
```

- The first line tells how wide and tall the chip itself is (in fundamental units), and the capacities of each gate site and pin site.

- The next line tells how many gates (numbered as consecutive integers starting at 1) and nets (ditto) in this problem.
- The next **NumOfgates** lines each describe one gate and the nets it connects to. The first number is the gate ID (integer starting at 1), then how many nets this gate connects to, then the net ID (integer) of each of these **NumOfNetsOnThisGate** nets. (Note: there is no separate list of the nets and the gates on each one; you will need to build this as you read in this gate-oriented netlist.)
- The next line tells how many pins there are that must be placed along the edges of the chip on pin sites.
- The next **NumOfPinsConnectedToNets** lines each list a pin (consecutive integers numbered from 1) and the NetId number (integer) of the net this pin connects to, and the constraint on which edge you can put the pin on. A **PinLocation** is one of the single characters "t" for top, "b" for bottom, "l" for left, or "r" for right edge.
- The next line tells how many paths we want you to watch while placing, and what your target delay--chip cycle time--is.
- The next **NumOfTimingPaths** lines each specify a path. Each path gets a number, starting from 1. The next field tells how many objects on the path: pins + gates + nets. Then the subsequent **ObjectsOnPath** numbers specify the actual elements of the path. The first element is a pin ID. Then there pairs of NetID number and GateID number, then the final NetID and final PinID.

You should expect to read this information in one line at a time, and build up the data structures you will need to be able to find out quickly things like:

- What nets are on this gate?
- What gates are on this net?
- Where is this gate now (what site?)
- Is this net connected to a pin? If so, which pin site?
- What gates are at this gate site? Are there more the gate capacity here?
- *etc etc etc.*

4.2 Output File Format: The CHECKER Program

How do you know if your code is working? Obviously you will print stuff out, and you can also do some live graphics so you can watch it run.

But to help, we will also provide a CHECKER program that will read the input file, and an output file format your tool must generate. CHECKER will then report back a lot of useful statistics, like:

- Total wirelength
- Any gross violations, like gates not accounted for, nets not accounted for, gates placed on places other then gate sites, pins placed on places other than pin sites, etc.

- Any capacity violations: more gates than a site can hold, or more pins than a site can hold.
- What the real delay is on all your paths, versus what you think it is.
- What the real worst case path is in your entire placement.
- A partitioning score that we will use later to try to see how different people's programs attacked the problem. This score will tentatively be a 3-tuple as follows:

  (**TotalWirelength, TotalCapacityViolationGateAndPin, TotalTimingViolation**)

  The first number will measure the wirelength as we have already specified it. The second number tells how much in excess of capacity you were on any pin or gate sites. The final number will sum up how much you were over the target cycle time on any of the paths we specified. A good rule of thumb here is that this tuple should come out like this:

  (**small, 0, 0**)

  We will try to come up with some neat ways of plotting this tuple for your final results to show the trade-offs people made.

So, you have to write out an output file that the CHECKER code will read as input, along with the actual netlist input file, to figure this stuff out. Here is the format:

```
GateID1 ChipXSite ChipYSite
GateID2 ChipXSite ChipYSite
...
GateIDlast ChipXSite ChipYSite
NetID1 Length Delay
NetID2 Length Delay
...
NetIDlast Length Delay
PinID1 ChipXSite ChipYSite
PinID2 ChipXSite ChipYSite
...
PinIDlast ChipXSite ChipYSite
PathID1 Delay
PathID2 Delay
...
PathIDlast Delay
```

- The first **NumOfGates** lines each list a gate ID (consecutive integers from 1), and the relative location (X,Y) on the chip. Note that (ChipXSite, ChipYSite) is measured from the lower-left corner of the chip, numbered as (0,0).
- The next **NumOfNets** lines likewise list what your program thinks is the Length and Delay of each net. We will use some appropriate fudge factor here (like 0.1%) to get around machine arithmetic variations.
- The next **NumOPinsConnectedToNets** lines list for each pin (consecutive integers starting at 1) the location of the pin site (X,Y) used for that pin.

- The next **NumOfTimingPaths** lines list for each constrained path (consecutive integers from 1) what you think the delay actually is for that path in your placement. Again, some appropriate fudge factor will be used.

As you can see, the output file is pretty simple and basically just dumps the final placement and what data you have for each gate, each pin, each net, and each path after your placement completed.

4.3 Benchmarks

We will provide a suite of benchmarks for you to run your tool on. These will come in three flavors:

- **Toy Problems:** small enough so that you can see what's happening what you run your tool and manually track down all the bugs in your first cut at the solution. These problems will have 10-100 gates.
- **For-Credit Problems:** we will provide 1-2 "real" problems that you must run your partitioner on, and run the CHECKER program on, and turn in the results. These will be smallish industrial semi-custom netlists with 1000-4000 gates.
- **We'll-Be-Very-Impressed-Extra-Credit Problems:** Same thing, only bigger. We'll dig up some problems in the 5000-15000 gate range. We'll be delighted to see what happens when you run your tool on these.

5.0 Algorithmic Formulation

The idea here is to use what you know about simulated annealing to try to build this timing-driven placer. Recalling again the different parts of an annealer (state, moves, cost function, cooling) here are some ideas.

5.1 State

This means how you store the evolving state of the problem. Obviously, something like a 2D array can store the sites for pins and gates, and you need some data structures to know what went where. You will also needs some data structures for the gates, pins, nets and paths that are appropriately interlinked you can get between them as necessary. For example: what nets are on this path? what gates are on this net? what nets are on this gate? what gates are in this site? etc.

5.2 Moves

There are really only a few obvious moves here:

- **Translation:** pick a random pin or a gate site, grab a random pin or gate there, and move it to a random new site. Notice that you may violate the capacity on the target site with a translation.

- **Swap:** pick a pair of random gates or pins (or even whole gate sites or pin sites themselves) and exchange them. Note that if your capacity limits were OK before a swap they are OK after the swap.

You will probably want to be able to do both kinds of moves, perhaps in some fixed proportion, or with more of one kind at the start of annealing and more of another kind at the end.

5.3 Cost Function

Here is where things get more interesting. Obviously you have to have NetLength as part of your overall cost function. But--how do you deal with the capacity stuff? One typical strategy is to use what are called *penalty functions*. A penalty function is part of the overall cost function, but its only nonzero when you violate some constraints. SO, you allow violations of the constraints to happen, but you penalize them and hope the annealer can minimize the violations. You try to formulate these so that when constraint violations are small, the penalties are small, but when the violations are large, the penalty is large. And when there is no violation at all, you get no penalty.

For example, to penalize excess gates at a sites, you might add this to the cost function:

$$\sum_{\text{gate sites}} (\text{number of gates in excess of capacity at this site})^2 \qquad \text{(EQ 1)}$$

The idea is that each site that has too many gates in it contributes a number to the penalty. The square makes excess gates “hurt more” in the overall penalty. And, sites that are not in violation contribute nothing to the penalty; indeed if all sites are below capacity this above penalty is just zero.

Note you can do the same thing for the pin sites, and even for the timing paths themselves (i.e., sum up the square of excess delay in each path that is too slow).

So, an overall cost function might look abstractly like:

$$\text{cost} = w1\left(\sum_{\text{nets}} \text{netlength}\right) + w2\left(\sum_{\text{gate sites}} \text{square of excess gates}\right)$$
$$+\, w3\left(\sum_{\text{pin sites}} \text{square of excess pins}\right) + w4\left(\sum_{\text{timing paths}} \text{square of excess delay}\right) \qquad \text{(EQ 2)}$$

where the first term is an objective (you try to make it small) and the remaining terms are constraint penalties (you try to make each zero--if you can). Of course, you have to figure out decent weights on each term here as well, usually by trial and error.

Also, it is important to remember that you have to be able to calculate the cost incrementally, after each move. In a big design with 10K gates, you cannot afford to go in and revisit every wire, every site and update their contribution to the cost. You must be able to calculate the cost based *only on what just changed* as a result of this move.

5.4 Cooling

Look in the code for the toy annealer for some ideas here. Basically you need to deduce an appropriately hot initial temperature, moves per temperature, cooling rate, and frozen criterion to get good results in what you personally think of as a reasonable amount of time. Again, you will need some empirical tuning here. And, think about range limiting for efficiency here.

6.0 What To Do For Credit

Here are the constraints:

- **Groups:** you can work in groups of up to 2 people. Pick your partner carefully. Group dynamics is your business.
- **Code**: you can write in JAVA (which will be slooooow...) or C or C++ on your favorite UNIX box. If you want to use something else, make sure you can read and write our mandatory netlist formats, and get at the data, and get your output to our UNIX CHECKER program. We are not going to provide any porting or translation service here for other than Solaris and AIX UNIX.
- **Graphics**: optional but a good idea. It's your business to decide if and what, but your code should could show something *illuminating* as it runs. We suggest a "rat's nest" diagram that shows all the gate and pin sites, and then puts a dot for each gate and a line for each net. You can be fancy if you like: plot all the sites in artistic detail; show the occupancy of each gate site as a color that suggests sparseness or fullness; draw the critical paths in some brighter color, etc. Debugging a placer that uses annealing is vastly easier if you can *see your placements*, even if you have to just print them out on paper as it runs.
- **Demo:** you have to sign up for a live demo to the Prof+TA at the end of the assignment. You come to us, you run your program on a new, small benchmark, and we watch your code run and ask you probing questions about what the heck it's doing. If you don't have graphics you at least have to print out some useful ASCII stuff as it runs.
- **Benchmarks:** you run your placer on at least the minimum set of benchmarks (we'll specify what this is) and also on any of the "big" netlists if you feel ambitious, and you tell us what the CHECKER program said about your solution, and you also tell us how much CPU time it took and on what machine. In UNIX, if your program is called FOOPART, you do this with this command:

  ```
  time FOOPART > FILE
  ```

  which will put a line of timing info into FILE when this thing is finished.

- **Write-up:** this is a big deal. You also submit a write-up (and the code) describing how you designed this, and how it works, and analyzing it.

The requirement of the write-up will be scored using the sheet on the following page.

18-760 Fall97 VLSI CAD
Proj. 3: Timing-Driven Placer [100 pts]

NAMES:

Problem Formulation [10 pts]: How did you decide to attack the placement problem? What where your objectives, what trade-offs or assumptions or simplifications did you choose, and why?

Algorithms & Data Structures [15 pts]: In detail, how did you solve this? Algorithm pseudo-code, data structure diagrams, analysis of complexity.

Live Demo [15 pts]: Did it work? Did it show something useful? Did the demonstrators answer questions in a lucid and coherent fashion?

Benchmark Results & Analysis [15 pts]: What did you run, what did CHECKER say about the result, is this good or bad, why are you getting these answers, did you make the right design trade-offs early in the design, etc.? **Analyze** what your tool can do. Include CHECKER outputs, plots of relevant details of the program doing the placement.

Write-up Style [20 pts]: Professional, neat, word-processed, coherent, grammatically clean, etc. Think of this as a document you are using to try to coax $1,000,000 out of a venture capitalist to fund a start-up CAD company to commercialize your breakthrough placement technology.

Code Quality [5 pts]: Yeah, we want to see it. Commented, indented, structured, etc.

Ambition/Style [20 pts]: This is a subjective judgement on how *well* you achieved your stated goals. You can shoot for a really solid, simple, elegant solution that does fabulously well for the smaller benchmarks only. You can shoot for something more complex that can run the difficult big benchmarks, but maybe not get great answers. One component of this score is this “style” part--does the tool do something well? The other part is your comparative rank against all the other tools in class, using the output of the CHECKER program. Best-in-benchmark-category gets you a few more points here.