

(Lec 17) Timing Analysis at the Logic Level

▼ What you know

- ▶ A lot of logic synthesis: going from a spec to a gate-level design
- ▶ How to simulate a design to verify what it does

▼ What you don't know

- ▶ Verifying *timing behavior* of some synthesized object
- ▶ Important example: *Static Timing Analysis*
 - ▷ I give you a gate-level netlist
 - ▷ I give you some “timing models” of the gates and maybe wires too
 - ▷ You tell me:
 - When signals arrive at various points in the network, or ...
 - Longest and shortest delays through gate network, or ...
 - Does the netlist meet some timing requirement?

▼ This is surprisingly complicated in the real world...

© R. Rutenbar 2001

CMU 18-760, Fall01 1

Acknowledgements

- ▼ Early versions of this talk used material from Karem Sakallah (U Michigan) and Tom Szymanski (Bell Labs)
- ▼ Current version of the talk extensively modified/updated by David Hathaway (IBM Essex Junction, VT)
 - ▶ Current version has also benefited from versions of 18-760 taught jointly by John Cohn (IBM) and Dave Hathaway (IBM) at the University of Vermont Dept of EE.
- ▼ *Many thanks to Karem, Tom, John, and especially Dave for all the inputs on this material*

© R. Rutenbar 2001

CMU 18-760, Fall01 2

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 3

Where Are We?

▼ After logic synthesis--how estimate delay of a netlist?

	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
Static timing analysis
Electrical timing analysis
Geometric data structs & apps

© R. Rutenbar 2001

CMU 18-760, Fall01 4

Readings

▼ De Micheli

- ▶ Chapter 8 on multilevel synthesis has a little bit about this.
- ▶ Read 8.6 on 'Algorithms for Delay Evaluation and Optimization'

Analyzing Design Performance

▼ Basic question

- ▶ Does the design meet a given timing requirement, or
- ▶ How fast can I run the design?
- ▶ Assume we know the delays of blocks in the network

▼ Why not just use ordinary gate-level delay simulation ...?

- ▶ Requires too many patterns
 - ▷ Exponential in the number of design inputs
 - ▷ Even worse if we consider sequences needed to initialize latches

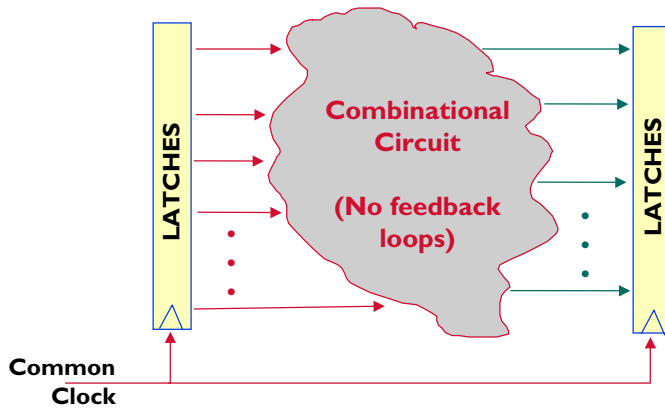
▼ So what do we do instead?

- ▶ Separate function from time
- ▶ Determine when transitions occur without worrying about how

Analyzing Design Performance

▼ Assume design is synchronous

- ▶ All storage is in explicit latch or flip-flop elements
- ▶ All cycles cut by clocked storage elements



© R. Rutenbar 2001

CMU 18-760, Fall01 7

Analyzing Design Performance

▼ Consider an arbitrary signal in a clocked design

- ▶ Takes on a value every cycle, sometimes one, sometimes zero
- ▶ Changes occur at different times in each cycle
 - ▷ Specific time of change depends on pattern causing it
 - ▷ May not change at all in some cycles
 - ▷ May make multiple changes before settling to final value



© R. Rutenbar 2001

CMU 18-760, Fall01 8

Static Timing Analysis

▼ Basic idea of static timing analysis

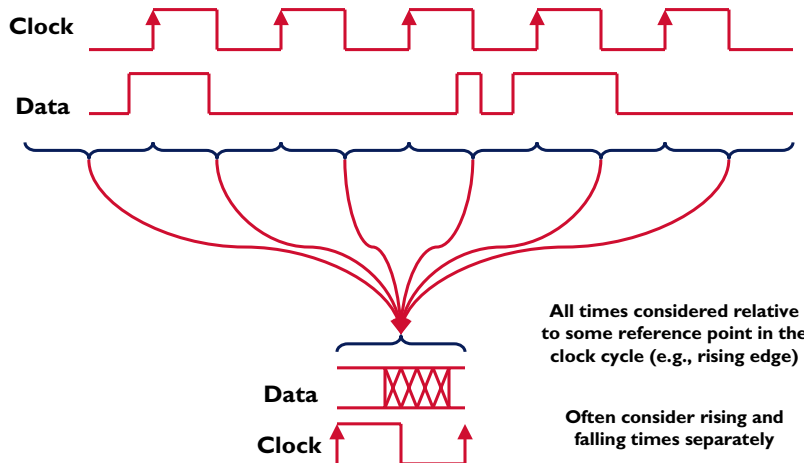
- ▶ Instead of considering an infinitely long simulation sequence
- ▶ Fold all possible transitions back into a single clock cycle
- ▶ Assume that signal becomes *stable* at latest possible time
- ▶ Assume signal becomes *unstable* at the earliest possible time
- ▶ If the design works at these extremes, we can *guarantee* it always will
- ▶ “Static” part just means we aren’t doing simulation (dynamic)

© R. Rutenbar 2001

CMU 18-760, Fall01 9

Static Timing Analysis

▼ Look at our data signal again



© R. Rutenbar 2001

CMU 18-760, Fall01 10

Timing Analysis: Basic Model

▼ So, the basic questions are:

- ▶ Does data always **reach** a stable value at all latch inputs in time for the clock to capture it?
 - ▷ Determine this by looking at **late mode** timing, or **longest path**
- ▶ Does data always **stay** stable at all latch inputs long enough after the clock to get stored?
 - ▷ Determine this by looking at **early mode** timing, or **shortest path**

▼ What do we need to answer this?

- ▶ First thing we need are “**delay models**” of the logic network
- ▶ Surprising variety of options here
- ▶ Depends on accuracy you need vs. computation you can afford

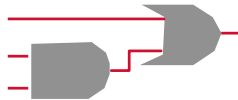
© R. Rutenbar 2001

CMU 18-760, Fall01 11

Delay Models

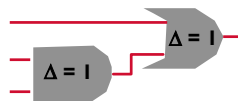
▼ Example gate network

- ▶ 3 primary inputs (PIs) and 1 primary output (PO)



▼ Simplest model: unit delay

- ▶ The delay through a gate -- ANY gate -- is equal to 1 time unit. Period.



Longest path is...

2

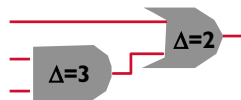
© R. Rutenbar 2001

CMU 18-760, Fall01 12

Delay Models

▼ Better model: Arbitrary but fixed delay per gate

- ▶ Each gate is allowed to have its own fixed delay
- ▶ This delay is constant -- doesn't depend on circuit netlist



Longest path is...

5

▼ Why isn't this enough?

- ▶ Unfortunately, real circuits are made from gates made out of transistors, and a lot of other circuit effects are present...

© R. Rutenbar 2001

CMU 18-760, Fall01 13

Delay Models

▼ The gate "loading" matters for delay

- ▶ Gates with more fanout are slower than gates with less fanout
- ▶ Look at the the AND gate on left and right



Gate output has to electrically drive all the fanout gates. More fanout means more load ==> slower.

- ▶ In real circuit, the loading presented by the connecting wires is actually the dominant contribution to the delay.
- ▶ Gate's delay model will usually depend on load of driven wires & gates
- ▶ Delay through wires can be longer than delays through gates!

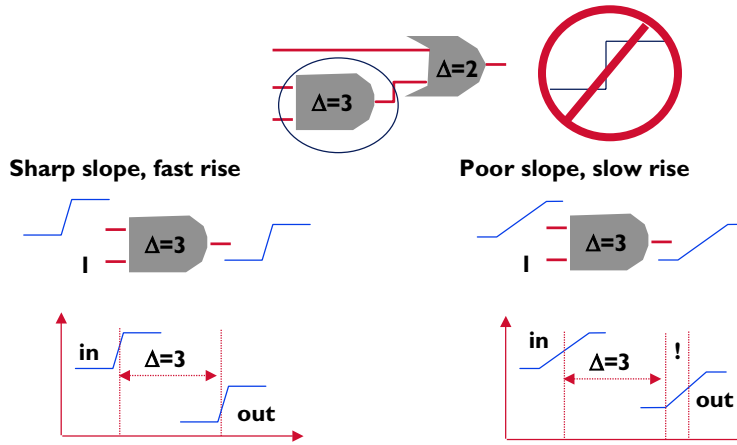
© R. Rutenbar 2001

CMU 18-760, Fall01 14

Delay Models

▼ The waveforms of the signals actually matter for delay

- ▶ Rising signal versus falling signal matters. Delays may be asymmetric
- ▶ Slope of the waveform seriously affects delay (RC circuit stuff)

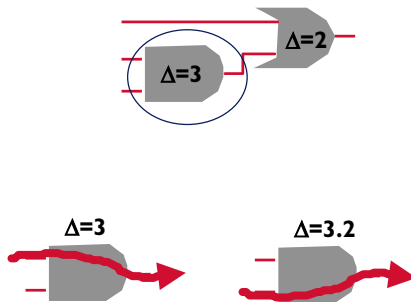


© R. Rutenbar 2001 CMU 18-760, Fall01 15

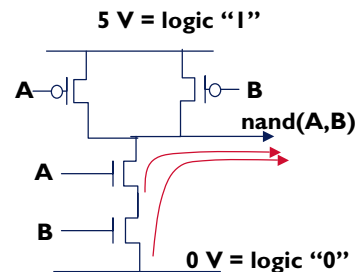
Delay Models

▼ Not all pins are created equal

- ▶ Delay is not really “through” a gate
- ▶ Delay is from each individual pin to gate output(s); all can be *different*



Why? Different transistor-level circuit paths input to output
Simple ex: NAND

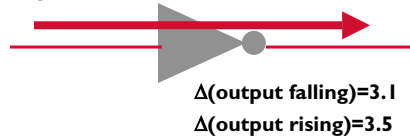


© R. Rutenbar 2001 CMU 18-760, Fall01 16

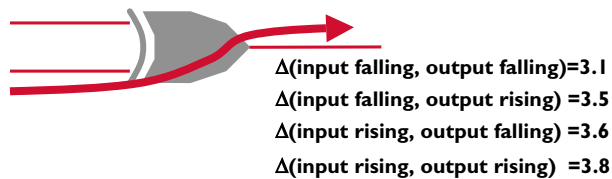
Delay Models

Not all transitions are created equal

- ▶ Separate transistors are used to drive a gate output to high/low values
- ▶ Transistors may be different sizes, P & N devices have different mobilities, and topology of pull-up and pull-down paths differ
- ▶ ... So delay can be different



- ▶ More complicated for non-monotonic functions



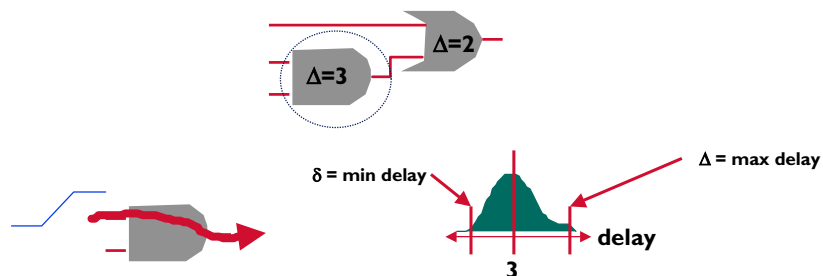
© R. Rutenbar 2001

CMU 18-760, Fall01 17

Delay Models

Delays may not even be scalars; may be a *distribution*

- ▶ Simplest is [min, max] which tries to quantify reasonable extremes on the manufacturing process
- ▶ In most elaborate case, it's a real probability distribution that gives you a real probability of the signal arriving with a given delay...
- ▶ ...and this distribution can still be a function of ALL these factors: waveform slope, output loading, different delay per pin, etc.
- ▶ Messy! Complicated!



© R. Rutenbar 2001

CMU 18-760, Fall01 18

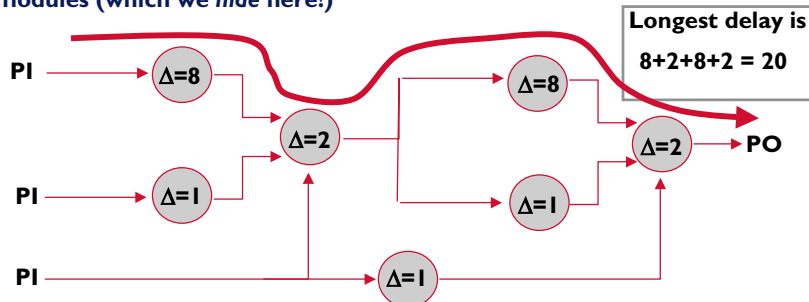
Timing Analysis: Topological vs. Logical

Another problem: Do we worry about gate "function"?

- ▶ Logical timing analysis: **YES**, we care what the gates actually do
- ▶ Topological timing analysis: **NO**, we don't care what gates do

What's the difference? Try an example...

- ▶ *Topological analysis* means we only worry about the delay through the paths through the graph shown below, not the logical function of the modules (which we *hide* here!)

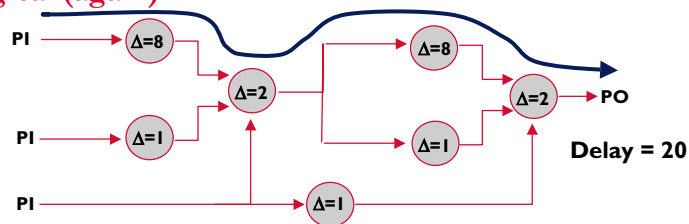


© R. Rutenbar 2001

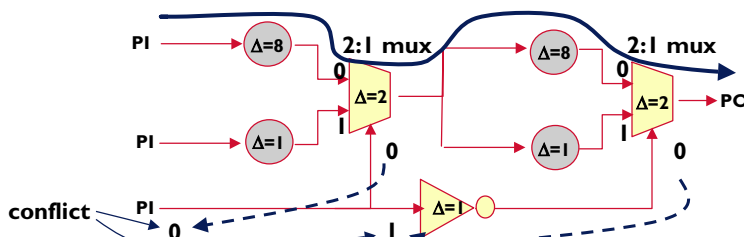
CMU 18-760, Fall01 19

Topological vs. Logical Timing Analysis

Topological (again)



Logical--we tell what gates are



© R. Rutenbar 2001

CMU 18-760, Fall01 20

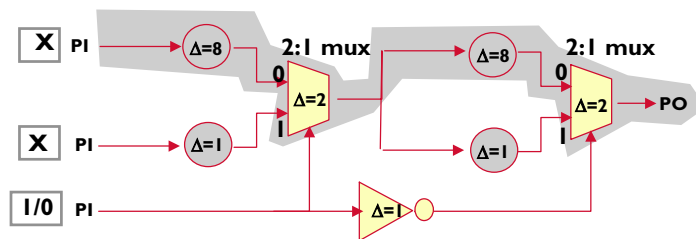
False Paths and Path Sensitization

▼ Oops. We got a false path

- ▶ It is not possible to apply a set of inputs that will cause a logic signal to propagate down this supposed “longest” path from PI to PO
- ▶ This path we found by topological analysis is called a **FALSE PATH**
- ▶ We got this because we didn’t care what the gates did

▼ Sensitization

- ▶ A path is said to be *sensitized* when it allows a logic signal to propagate along it. In this example, there is no way to sensitize this path



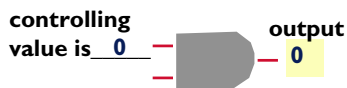
© R. Rutenbar 2001

CMU 18-760, Fall01 21

Sensitization

▼ Definitions

- ▶ **Controlling value for a gate** is a single input value to a gate that uniquely forces the output to a known constant, independent of the other inputs to the gate.



- ▶ A gate is **sensitized** so a logic signal can propagate through it from one particular input to the output if the other inputs have stable **noncontrolling** values



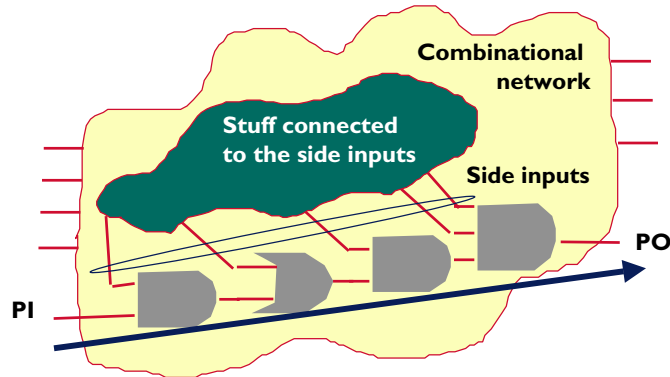
© R. Rutenbar 2001

CMU 18-760, Fall01 22

Sensitization

Definitions

- ▶ A **path** is a set of connected gates and wires that starts with some PI and ends with some PO. Path is defined by 1 input and 1 output per gate
- ▶ **Side inputs** on a path are the “other” inputs to these gates on the path.



© R. Rutenbar 2001

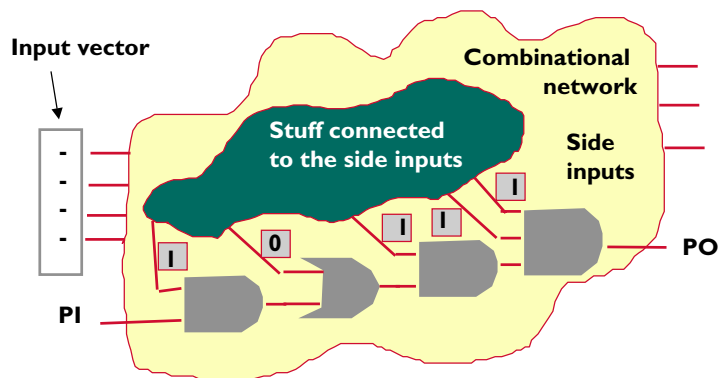
CMU 18-760, Fall01 23

Static Sensitization

Static sensitization

- ▶ A path is **statically sensitizable** when...

There is an input vector which generates *stable* noncontrolling values to all side inputs on the path

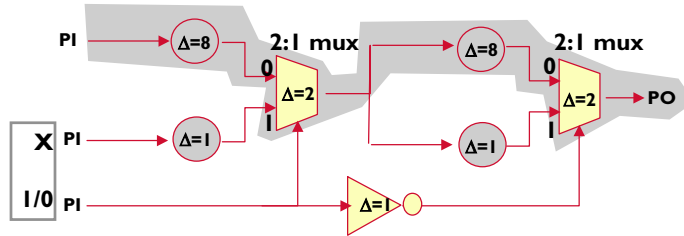


© R. Rutenbar 2001

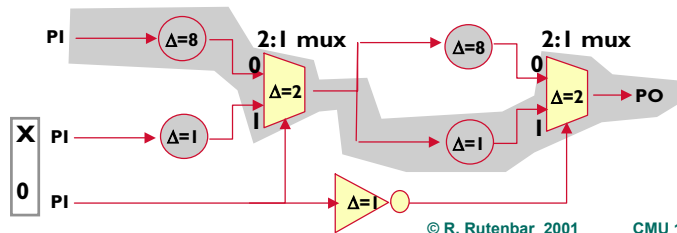
CMU 18-760, Fall01 24

Static Sensitization

NOT statically sensitizable



Statically sensitizable



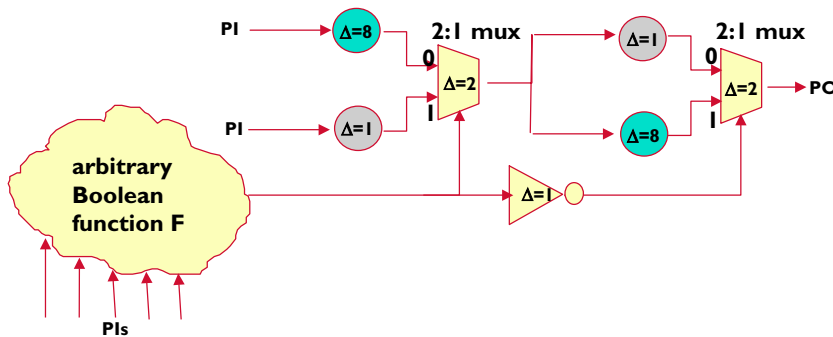
© R. Rutenbar 2001

CMU 18-760, Fall01 25

Sensitization

How hard is it really to do this?

- ▶ In general, very hard, though there are many good heuristics
- ▶ As hard as Boolean satisfiability (find a pattern of inputs to make an arbitrary Boolean function == 1), which is NP hard
- ▶ New example below: delay = 20 if $F=1$ else delay = 6 if $F=0$.



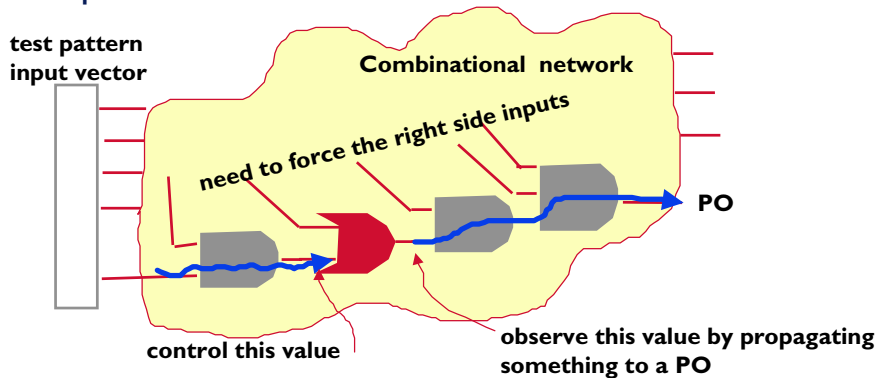
© R. Rutenbar 2001

CMU 18-760, Fall01 26

Aside: Related to Testing for Gate-Level Circuits

▼ What's testing about?

- ▶ Find inputs to a gate network that force a particular value on a particular input of a particular gate...
- ▶ ...and that also allow the output of that gate to propagate to some output.



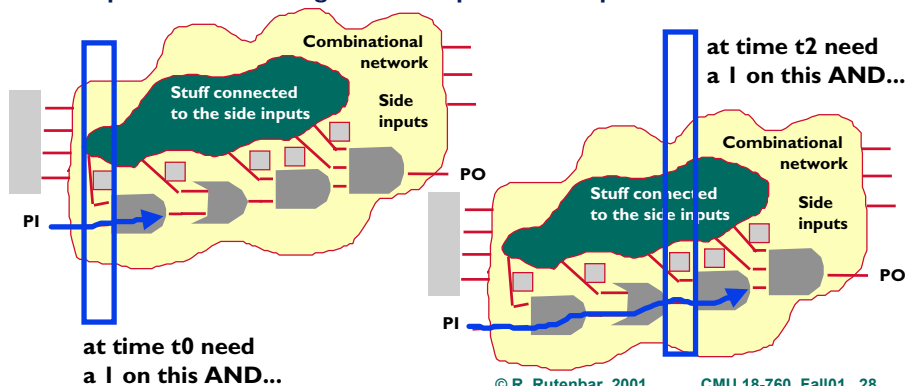
© R. Rutenbar 2001

CMU 18-760, Fall01 27

Beyond Static Sensitization...?

▼ *Dynamic sensitization*

- ▶ Try to find vectors to apply at different times so that the right noncontrolling value appears at each side input *when the propagating signal gets to that particular gate*
- ▶ Messy, hard to do.
- ▶ People are still working on various practical simplifications of this.



© R. Rutenbar 2001

CMU 18-760, Fall01 28

So, What Are We Doing Here?

▼ Simple fixed delay gate model

- ▶ No slopes, etc. Any loading effects are “bundled” back into the gate delay number itself.



▼ Topological path analysis

- ▶ We don't worry about what the gates do
- ▶ We only look at paths through the connected gates
- ▶ Aside: means we assume all paths statically sensitizable.
- ▶ We know we will get false paths -- too bad.
- ▶ This is usually a pessimistic timing model -- delay numbers too big since we find false paths first that are usually overly long

© R. Rutenbar 2001

CMU 18-760, Fall01 29

Topological Path Analysis

▼ Generally what people mean by static timing analysis

▼ PRO

Fast (pattern independent)
Bounds true worst path delay

▼ CON

Can be pessimistic (includes false paths)

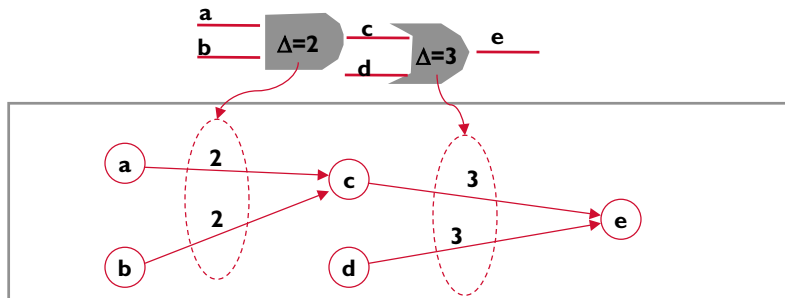
© R. Rutenbar 2001

CMU 18-760, Fall01 30

Representation: Delay Graph

How do we model gate network? *Delay Graph*

- ▶ Gates = edges, 1 edge per input pin.
- ▶ Numbers on edges = delay through gates
- ▶ Wires (signals) = vertices. 1 per gate output
 - ▷ Also 1 for each PI, PO
 - ▷ Leave latches out for now
- ▶ Predecessor: $\text{pred}(n)$ = any node p where there is an edge from $p \rightarrow n$
- ▶ Successor: $\text{succ}(n)$ = any node s where there is an edge from $n \rightarrow s$
- ▶ Note: this ends up as a directed, acyclic graph, a DAG

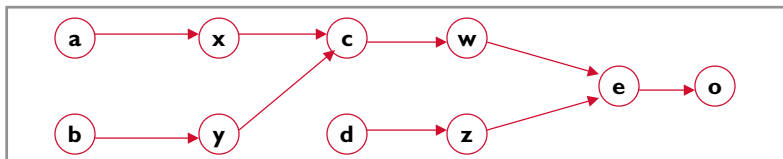


© R. Rutenbar 2001 CMU 18-760, Fall01 31

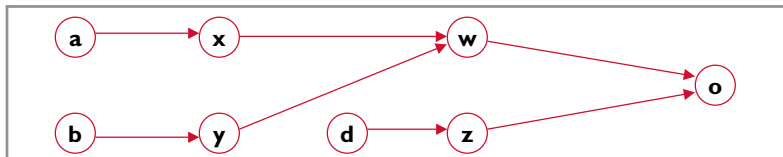
Representation: Delay Graph

What about interconnect delay?

- ▶ Can use delay graph with node for each *pin* instead of each *net*



- ▶ Gate and net delays interact - can have delay edge from input to input



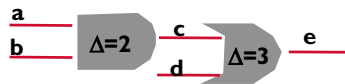
- ▶ We'll stick with one node per net for simplicity

© R. Rutenbar 2001 CMU 18-760, Fall01 32

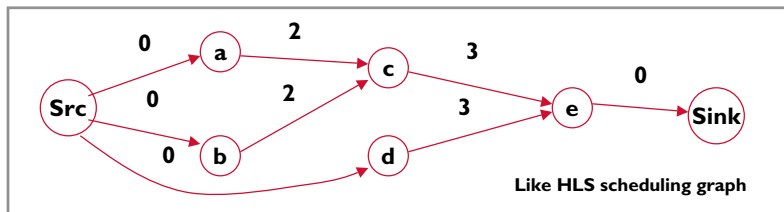
Delay Graph

Source / Sink nodes (pure combinational logic)

- ▶ Often add 1 “source” node that has a 0-weight edge to each PI
- ▶ ..and 1 “sink” node with 0-weight edge from each PO
- ▶ Now network has 1 clear “entry” node, and 1 clear “exit” node
- ▶ Even timers that don’t explicitly add these nodes do something similar
 - ▷ Loop through all PIs (POs) \Leftrightarrow loop through fanout (fanin) of source (sink) node



Non-zero values on Src/Sink edges can be used to represent different timing constraints on different PIs and POs



© R. Rutenbar 2001

CMU 18-760, Fall01 33

Operations on Delay Graph

So how do we use this graph to do timing analysis

- ▶ Simple approach: path enumeration = list all paths, in some order

Easy to do this in a naive way

```

search (path P, delay d) {
    n = last node in P;
    if ( there are no successor nodes to n )
        Output path P, delay d; /* All paths end at sink */
    else {
        foreach (node s in succ(n) ) {
            search ( P+s, d+delay(n,s) );
        }
    }
}
search (source);
    
```

Add one more node to the end of the path and recurse

OK, it works. What’s wrong with this?

© R. Rutenbar 2001

CMU 18-760, Fall01 34

Path Enumeration

▼ Problem is number of paths

- ▶ Can be exponential in length of paths
- ▶ Our “search” algorithm doesn’t visit paths in any useful order



How many paths from node 0 to node n in here?

2^n

- ▶ Some timing analyzers do this anyway
 - ▷ May use pruning methods to control exponential behavior

Operations on Delay Graph

▼ Instead we’ll use what’s been called *block-oriented analysis*

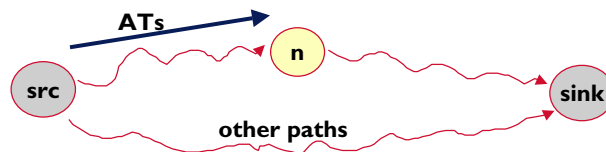
- ▶ Don’t look for paths to the sink (primary outputs)
- ▶ Instead find for *each* node the worst delay to the node along *any* path

▼ Need to define some terms ...

Values on Nodes in Delay Graph

Arrival Times at a node (ATs)

- ▶ $AT_E(n)$ = Earliest signal can become unstable at node n
 - ▷ Determined by shortest path from source
- ▶ $AT_L(n)$ = Latest time signal can become stable at node n
 - ▷ Determined by longest path from source
- ▶ Sometimes called “delays to node”



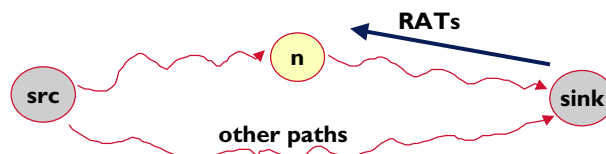
© R. Rutenbar 2001

CMU 18-760, Fall01 37

Values on Nodes in Delay Graph

Required Arrival Times at a node (RATs)

- ▶ $RAT_E(n)$ = Earliest that signal is allowed to become unstable at node n
 - ▷ Determined by shortest path to sink
- ▶ $RAT_L(n)$ = Latest time signal is allowed to become stable at node n
 - ▷ Determined by longest path to sink
- ▶ Related to what is sometimes called “delay from node”



© R. Rutenbar 2001

CMU 18-760, Fall01 38

Values on Nodes in Delay Graph

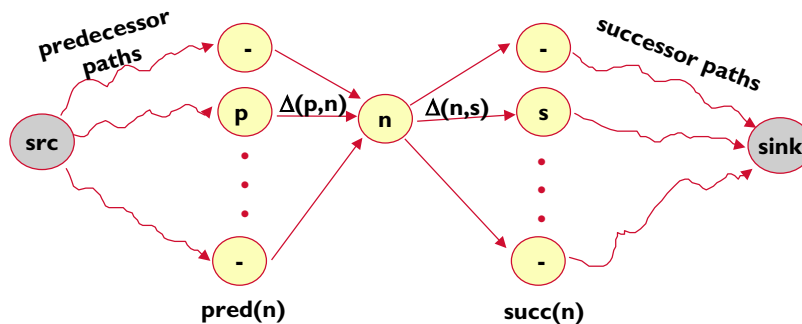
▼ Slacks at a node

- ▶ $\text{Slack}_E(n) = \text{AT}_E(n) - \text{RAT}_E(n)$
 - ▷ Amount of margin in time signal goes unstable
 - ▷ Determined by shortest path through node
 - ▷ Amount by which a signal can be sped up at a node and not decrease the length of the shortest path through the network
- ▶ $\text{Slack}_L(n) = \text{RAT}_L(n) - \text{AT}_L(n)$
 - ▷ Amount of margin in time signal becomes stable
 - ▷ Determined by longest path through node
 - ▷ Amount by which a signal can be delayed at a node and not increase the length of the longest path through the network
 - ▷ Can increase delay at a node (to minimize power, circuit area) with positive late mode slack and not degrade overall performance
- ▶ Defined so negative slack always indicates a timing problem
- ▶ Measures “sensitivity” of network to this node’s delay

© R. Rutenbar 2001

CMU 18-760, Fall01 39

How To Compute...?



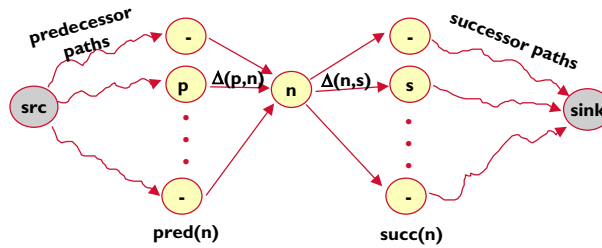
▼ Recursively.

- ▶ In terms of (assumed) known values of the desired quantities for either the successor or predecessor nodes, as shown above.
- ▶ Let’s try it...

© R. Rutenbar 2001

CMU 18-760, Fall01 40

Arrival Times for a Node n



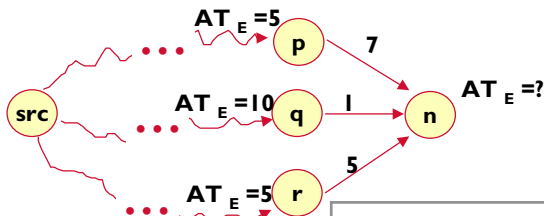
$$AT_E(n) = \text{min delay to } n = \begin{cases} 0 & \text{if } n == \text{src} \\ \text{Min}_{p = \text{pred}(n)} \{ AT_E(p) + \delta(p,n) \} & \end{cases}$$

$$AT_L(n) = \text{max delay to } n = \begin{cases} 0 & \text{if } n == \text{src} \\ \text{Max}_{p = \text{pred}(n)} \{ AT_L(p) + \Delta(p,n) \} & \end{cases}$$

© R. Rutenbar 2001

CMU 18-760, Fall01 41

Aside: Quick Concrete Example



$$AT_E(n) = \text{Min}_{x \in \{p, q, r\}} \{ AT_E(x) + \delta(x,n) \}$$

$$= \text{Min}(5+7, 10+1, 5+5)$$

$$= 10$$

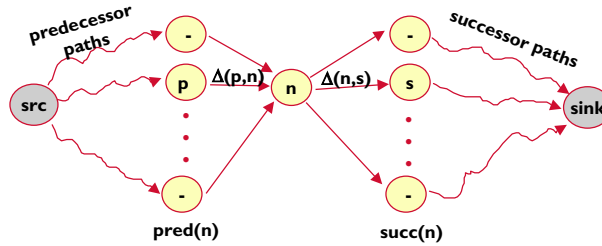
Big idea

- ▶ If a particular path to node n has min (max) delay from source...
- ▶ ...then if we take node n off the end of the path, the shorter partial path (to node r, here) is the min (max) delay path from source to node r
- ▶ This is why the recursion idea works

© R. Rutenbar 2001

CMU 18-760, Fall01 42

Required Arrival Times for a Node n



$$RAT_E(n) = \begin{cases} 0 & \text{if } n == \text{sink} \\ \text{Max}_{s = \text{succ}(n)} \{RAT_E(s) - \delta(n,s)\} \end{cases}$$

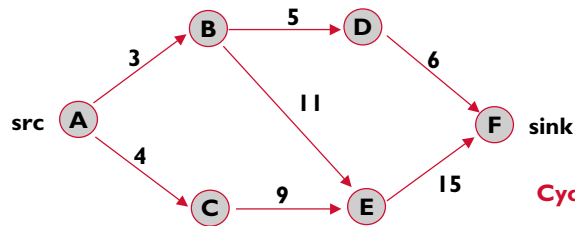
$$RAT_L(n) = \begin{cases} \text{Cycle time if } n == \text{sink} \\ \text{Min}_{s = \text{succ}(n)} \{RAT_L(s) - \Delta(n,s)\} \end{cases}$$

Note reversal of min and max for early and late modes; this is because we're subtracting delays instead of adding them

© R. Rutenbar 2001

CMU 18-760, Fall01 43

Example



Cycle time = 30

$$AT_E(E) = 4+9 = 13$$

$$AT_L(E) = 3+11 = 14$$

$$RAT_E(B) = 0-6-5 = -11$$

$$RAT_L(B) = 30-11-15 = 4$$

$$\text{Slack}_E(B) = 3-(-11) = 14$$

$$\text{Slack}_L(B) = 4-3 = 1$$

For simplicity, assume delays on edges are both min and max values

© R. Rutenbar 2001

CMU 18-760, Fall01 44

Computational Strategy

▼ OK, we can *define* them, but can we *compute* them?

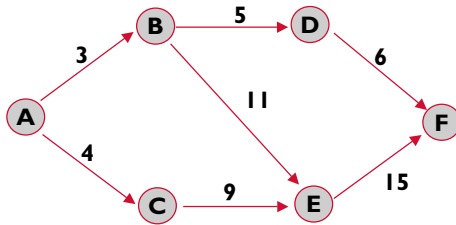
▶ Actually, all pretty easy.

▼ Essential idea: *topological sorting of a DAG*

▶ Sorting of the vertices in the DAG into a total linear ordering...

▶ ...i.e., a single ordered list of vertices in the DAG

▶ Essential property of sort: if there is an edge from $p \rightarrow s$ in the DAG, then p comes *before* s in the sorted order. True for ALL edges



Legal Topological Sort Orders

A,B,D,C,E,F
A,B,C,D,E,F
A,B,C,E,D,F
A,C,B,D,E,F
A,C,B,E,D,F

© R. Rutenbar 2001

CMU 18-760, Fall01 45

Topological Sorting

▼ Pretty easy application of depth-first-search (DFS)

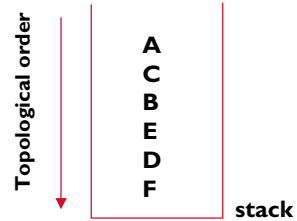
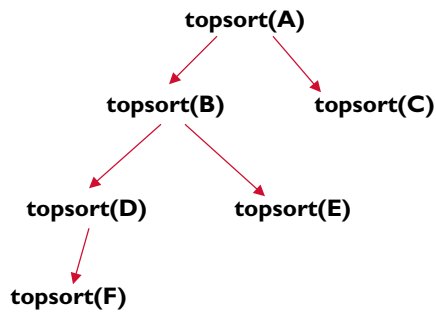
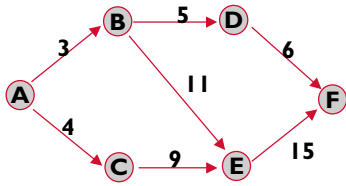
```
topsort( node n ) {  
  for each s in succ(n) {  
    if s has not been visited {  
      topsort( s );  
      push n on stack ;  
      mark n as visited;  
    }  
  }  
}  
  
topsort(SRC);
```

© R. Rutenbar 2001

CMU 18-760, Fall01 46

Topological Sorting

▼ Apply to our example

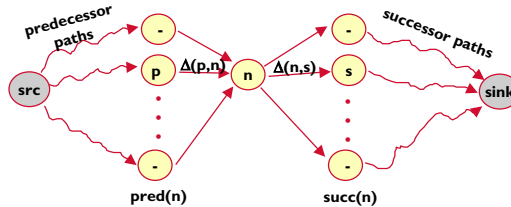


© R. Rutenbar 2001

CMU 18-760, Fall01 47

Computing ATs

▼ Assume we now have the topological sort order



```

get_ATs() {
  AT_E(src) = 0; AT_L(src) = 0;
  for each n in topsort order {
    AT_E(n) = ∞; AT_L(n) = -∞;
    for each p in pred(n) {
      AT_E(n) = min( AT_E(n), AT_E(p) + δ(p,n) );
      AT_L(n) = max( AT_L(n), AT_L(p) + Δ(p,n) );
    }
  }
}

```

Alternatively, we can omit the topological sort and compute AT_E and AT_L for node n on return from recursion (when values for all $pred(n)$ have been computed) during DFS backward from n .

This is called *demand-driven computation*.

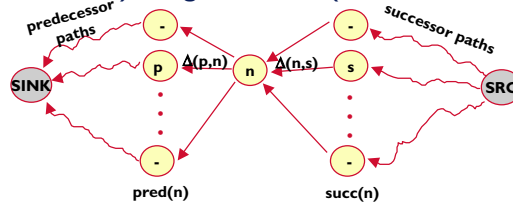
© R. Rutenbar 2001

CMU 18-760, Fall01 48

Computing RATs

Again, assume we have topological sort order

- RATs same as the ATs would be if you reversed all arrows and start from sink (now=source) and go to source (which is now the sink)!



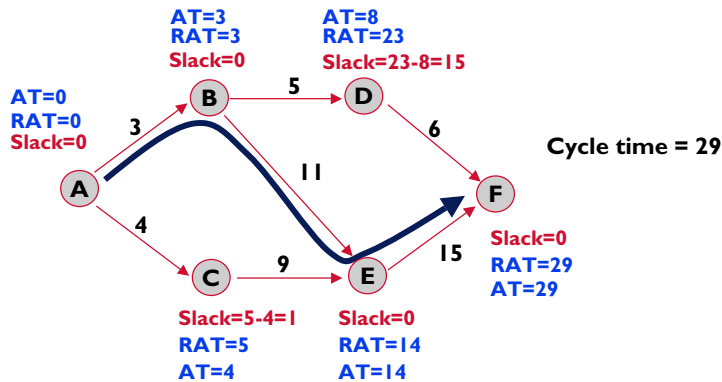
```

get_RATs() {
  RAT_E(sink) = 0; RAT_L(sink) = cycle_time;
  for each n in reverse topsort order {
    RAT_E(n) = -∞; RAT_L(n) = ∞;
    for each s in succ(n) {
      RAT_E(n) = max( RAT_E(n), RAT_E(s) - δ(n,s) );
      RAT_L(n) = min( RAT_L(n), RAT_L(s) - Δ(n,s) );
    }
  }
}
    
```

Slack

Interesting slack property

- All nodes on a critical (longest) path have same slack
- Consider a late mode analysis:

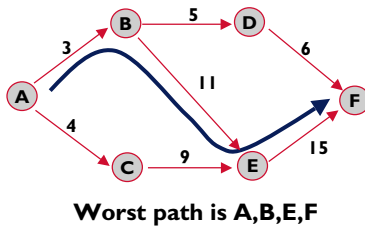


- Allow us to report worst paths, even though we didn't trace them all

Path Reporting

Find N worst paths

- ▶ Keep priority queue (heap) of unfinished partial paths
 - ▷ Sort so path with worst slack endpoint is always on top
 - ▷ Initially contains only the source node
- ▶ Algorithm:
 - ▷ Pull partial path off the heap (will be start of next most critical path)
 - ▷ Until path is finished:
 - Add worst slack successor to current path
 - Add other successors to path and put them on the queue
 - ▷ Repeat until N paths have been reported



First trace path A,B,E,F
 Partial paths: A,B,D, slack = 15
 A,C, slack = 1
 So visit A,C next, expand to
 A,C,E,F
 Finally visit A,B,D, expand to
 A,B,D,F

© R. Rutenbar 2001 CMU 18-760, Fall01 51

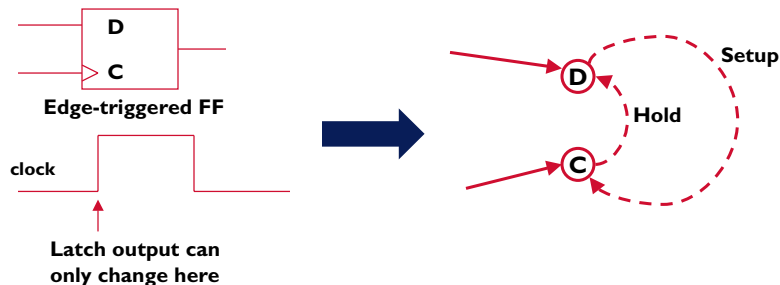
Beyond Combinational Logic

So far we've assumed only combinational logic

- ▶ All path requirements are same
- ▶ No feedback paths or backward interaction in delay graph

Consider a network containing flip-flops

- ▶ We treated it as a PO of our combinational logic
- ▶ OK if all clocks are ideal and arrive at the same time ... but they don't
- ▶ So we add test edges to the delay graph



© R. Rutenbar 2001 CMU 18-760, Fall01 52

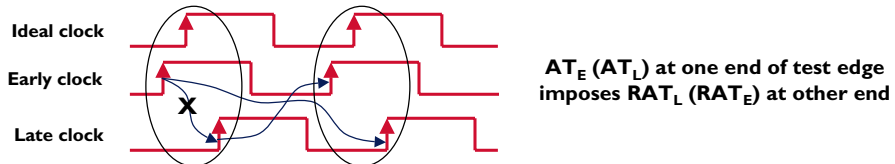
Beyond Combinational Logic

How are tests used?

- ▶ **Hold** test says *late* clock must precede *early* data by some amount
- ▶ **Setup** test says *late* data must precede *early* clock by some amount

Complication - adjusts

- ▶ Remember that many cycles of activity were “folded” into one cycle
- ▶ So data arriving at latch is really for next cycle
- ▶ Need to add/subtract clock cycles so we’re comparing the right times
 - ▷ Need to know which cycle data *should* be latched in
 - Generally assume data is captured by first possible edge of the *ideal* clock following the one that launched it
 - Exceptions must be asserted by user, e.g., multi-cycle paths



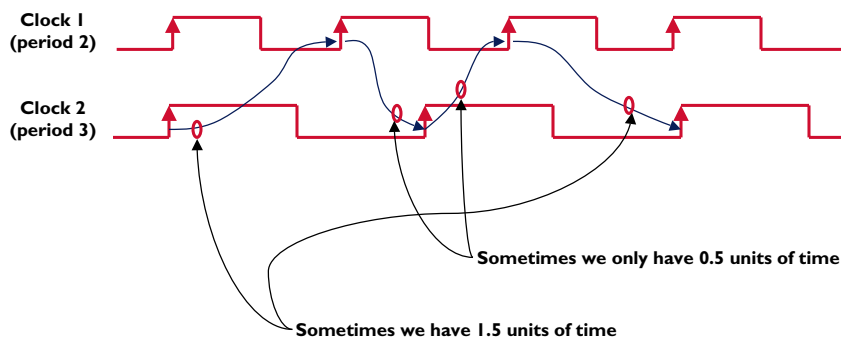
© R. Rutenbar 2001

CMU 18-760, Fall01 53

Beyond Combinational Logic

Gets even more complicated with multiple clock frequencies

- ▶ Use greatest common divisor (GCD) of clock periods to determine smallest possible separation between launch & capture edges
- ▶ Example:



© R. Rutenbar 2001

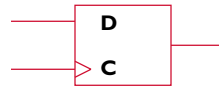
CMU 18-760, Fall01 54

Slack Stealing

▼ So far we've assumed edge-triggered flip-flops

- ▶ Time that data changes at latch output is determined *only* by clock

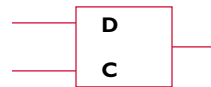
▼ Consider transparent latches



Edge-triggered FF



Latch output can
only change here



Transparent latch



Latch output can
change anywhere in here

Data AT on input can
affect AT on output!

© R. Rutenbar 2001

CMU 18-760, Fall01 55

Slack Stealing

▼ But this means the arrival at the end of one path affects the arrival at the beginning of another path

- ▶ Violates acyclic assumption

▼ How can we handle this?

- ▶ Break all cycles
- ▶ Assume a launch time at each latch
 - ▷ Start with clock leading edge
 - ▷ Add a test to require the capture time to meet this assumption
- ▶ Perform a static timing analysis
- ▶ Adjust your assumptions to equalize slack at latch inputs & outputs
 - ▷ Move the launch time with the clock active window
- ▶ Repeat until convergence or you run out of time

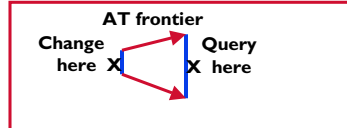
© R. Rutenbar 2001

CMU 18-760, Fall01 56

Incremental Timing Analysis

▼ How do I update timing after making changes?

- ▶ Incremental timing allows efficient update of only changed information after changes to design
 - ▷ Compute *level numbers* when computing original ATs, RATs
 - ▷ All changes can be viewed as change to delay edges
 - Add an edge
 - Delete an edge
 - Change the delay on an edge
 - ▷ Keep track of *frontiers* of timing changes
 - Keep sorted by level number
 - ▷ When a value is requested on a node at level x
 - Recompute, by level, all frontier values \leq than level
 - If value changes, add its fanout to the frontier



Effects of propagated slew changes on delay make RAT case more complicated

© R. Rutenbar 2001

CMU 18-760, Fall01 57

Timing Analysis Summary

▼ Gate-level delay models

- ▶ Can be very complex if you deal with all the effects
- ▶ Load, slope, pin, etc., all really matter
- ▶ Simplification is just a fixed delay per gate (or per input pin, same thing)

▼ Logical \neq Topological path analysis

- ▶ Logical = we worry about false paths, what the gates really do. This is still pretty hard, and a lot of computational work.
- ▶ Topological = we don't worry about logic function of nodes in our delay graph. This is conservative, can overestimate longest delay.

▼ Topological analysis = Depth first search

- ▶ Make delay graph
- ▶ Can compute ATs, RATs, and Slacks for each node

© R. Rutenbar 2001

CMU 18-760, Fall01 58