Static Timing Analysis (STA)

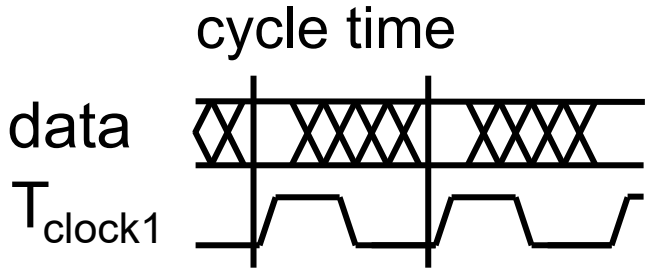
Some contributions from
Lei He
Andrew B. Kahng
Igor Markov
Mohammad Tehranipoor

Logistics

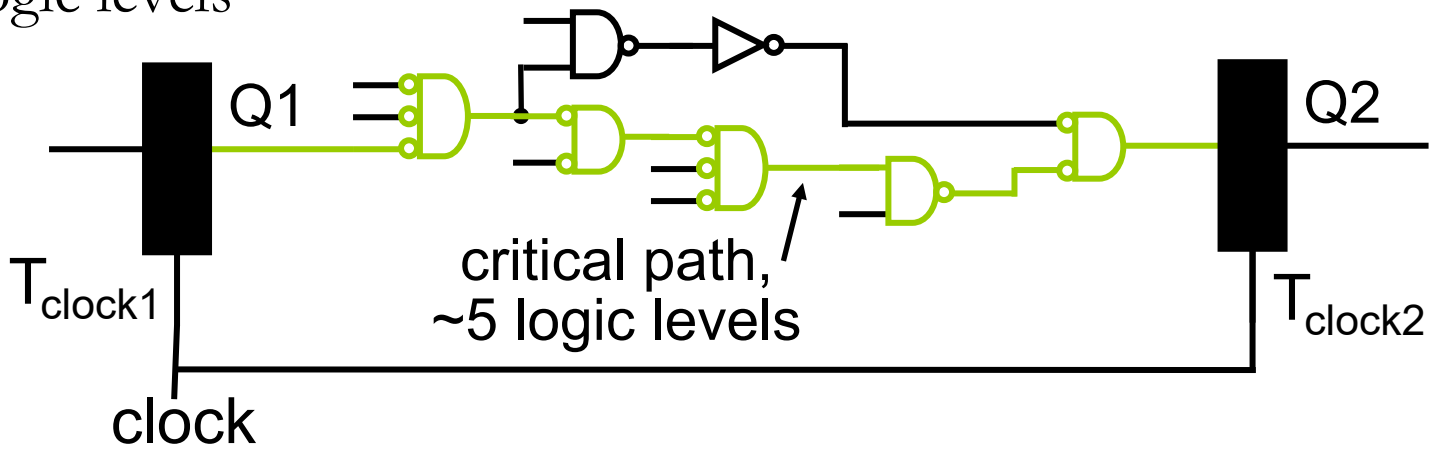
- Lab 2 is assigned.
  - The course will be *very* fast paced.
- We will have labs every week
  - Lab 4 is critical to prepare for the Final Project
- Final Project v0 should be assigned in Week 5/6.
  - Midterm project report due in Week 8
- Quiz 1, Feb 8 in class at beginning of class
  - Second half of class: Cadence Innovus Tutorial
- No Office hours for me on Feb 7.
  - Please email me with any questions not answered on Piazza or if you want to have a zoom call for something.

Let's Revisit Cycle Time and Path Delay

- Cycle time (T) cannot be smaller than longest path delay (T_{max})
- Longest (critical) path delay is a function of:
- Total gate, wire delays
  - logic levels



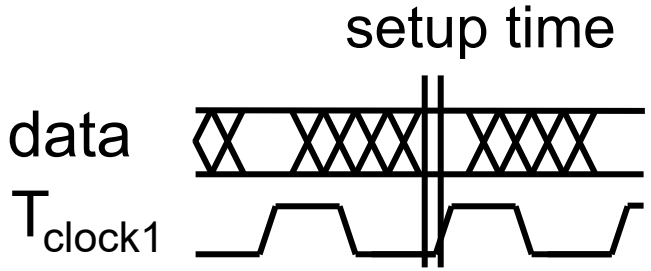
$T_{max} \leq T$



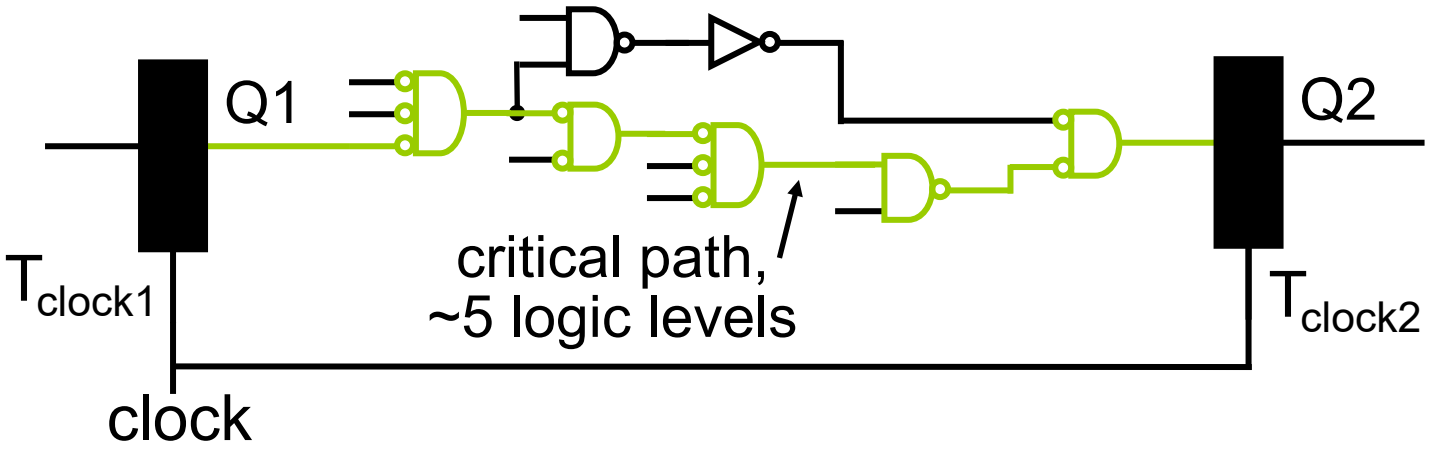
Courtesy K. Keutzer et al. UCB

Cycle Time - Setup Time

- For FFs to correctly capture data at inputs, must be stable for:
- Setup time (T_{setup}) *before* clock arrives



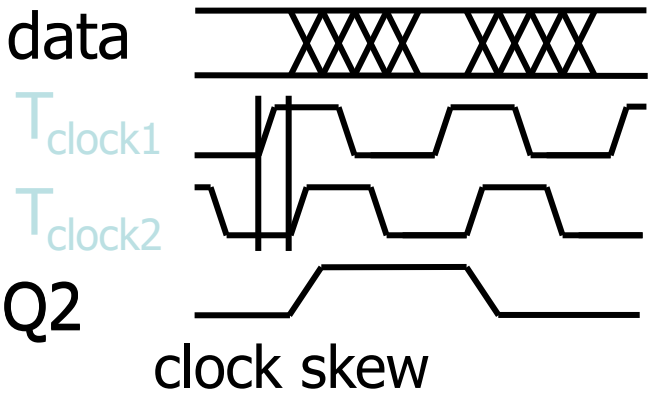
$T_{max} + T_{setup} \leq T$



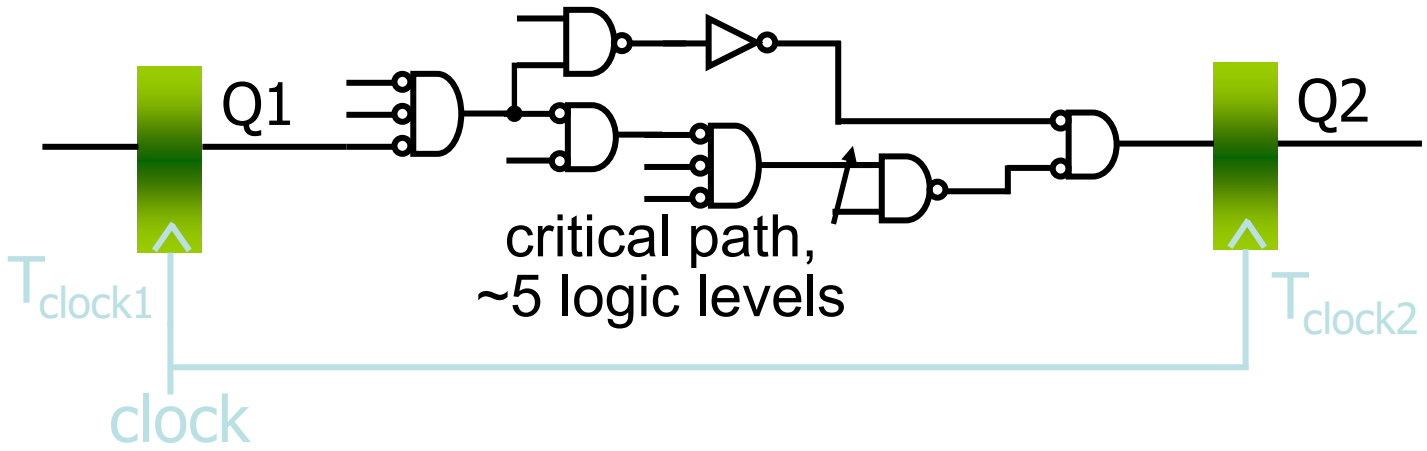
Courtesy K. Keutzer et al. UCB

Cycle Time – Clock Skew

- If clock network has unbalanced delay – clock skew
- Cycle time is also a function of clock skew (T_{skew})



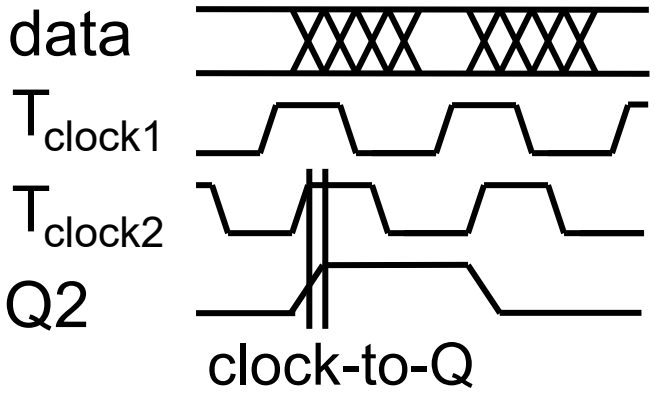
$$T_{max} + T_{setup} + T_{skew} \leq T$$



Courtesy K. Keutzer et al. UCB

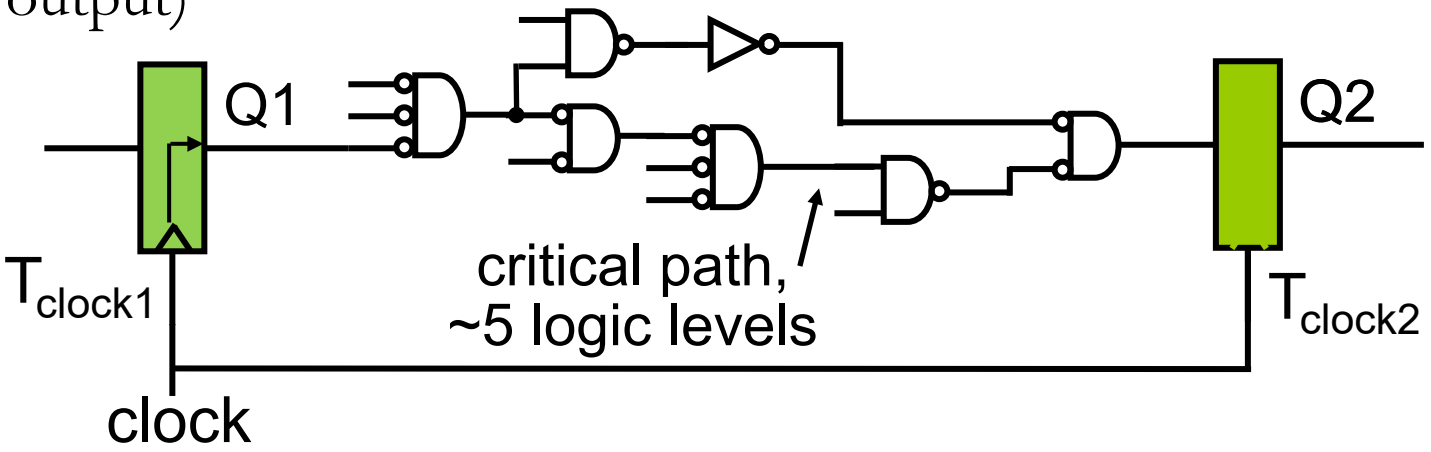
Cycle Time – Flip-Flop Delay (Clock to Q)

- Cycle time is also a function of propagation delay of FF ($T_{clk\text{-}to\text{-}Q}$ or T_{c2q})



- T_{c2q} : time from arrival of clock signal till change at FF output)

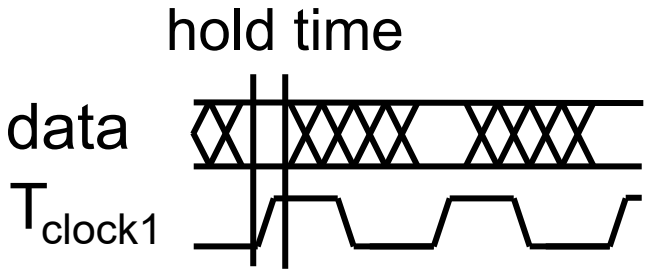
$$T_{max} + T_{setup} + T_{skew} + T_{clk-to-Q} \leq T$$



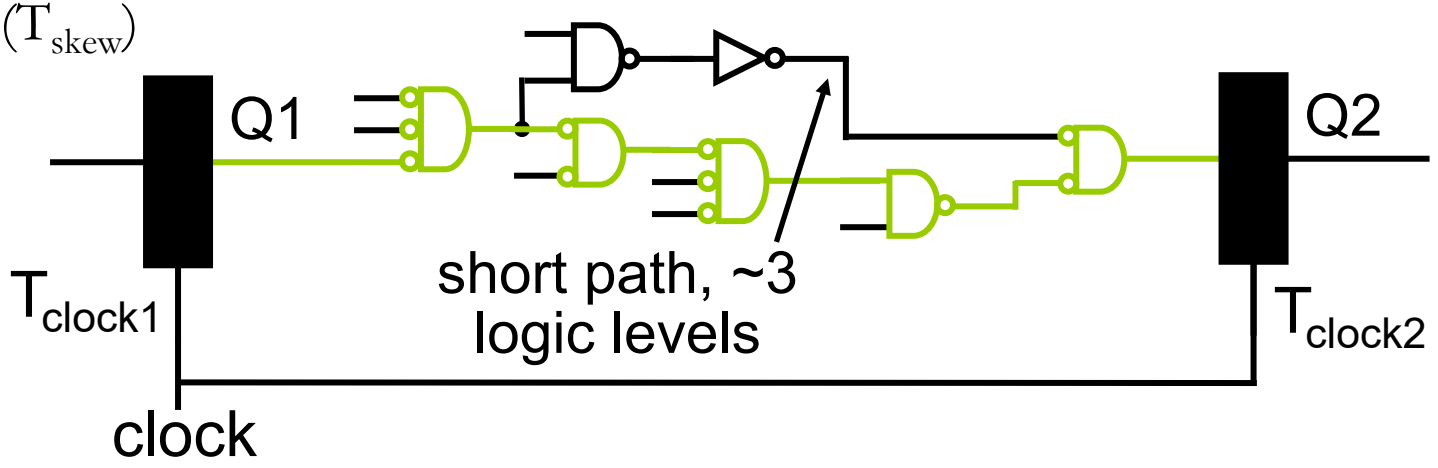
Courtesy K. Keutzer et al. UCB

Min Path Delay - Hold Time

- For FFs to correctly latch data, data must be stable during:
  - Hold time (T_{hold}) *after* clock arrives
  - Determined by delay of shortest path in circuit (T_{min}) and clock skew (T_{skew})

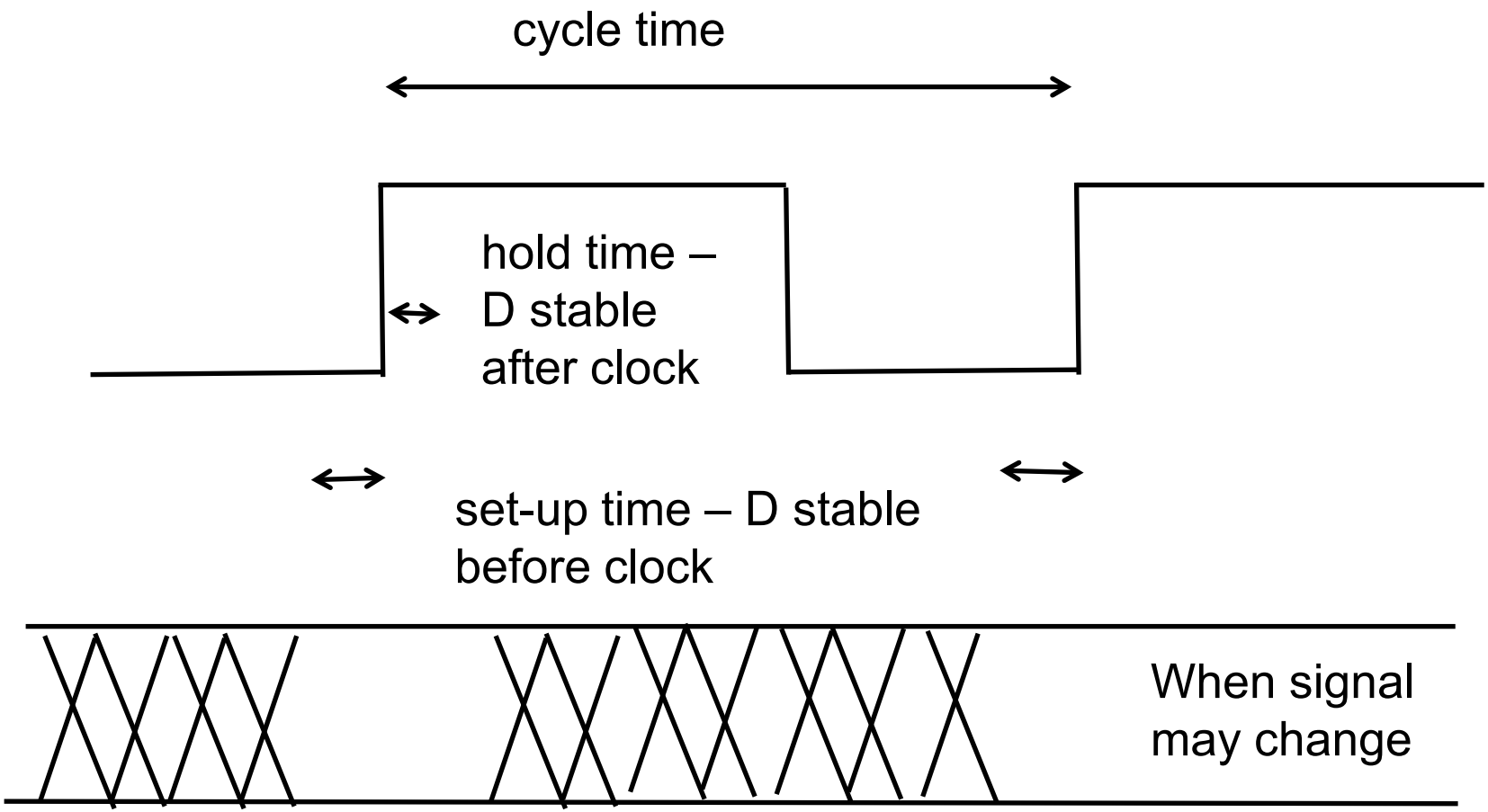


$T_{min} \geq T_{hold} + T_{skew}$



Courtesy K. Keutzer et al. UCB

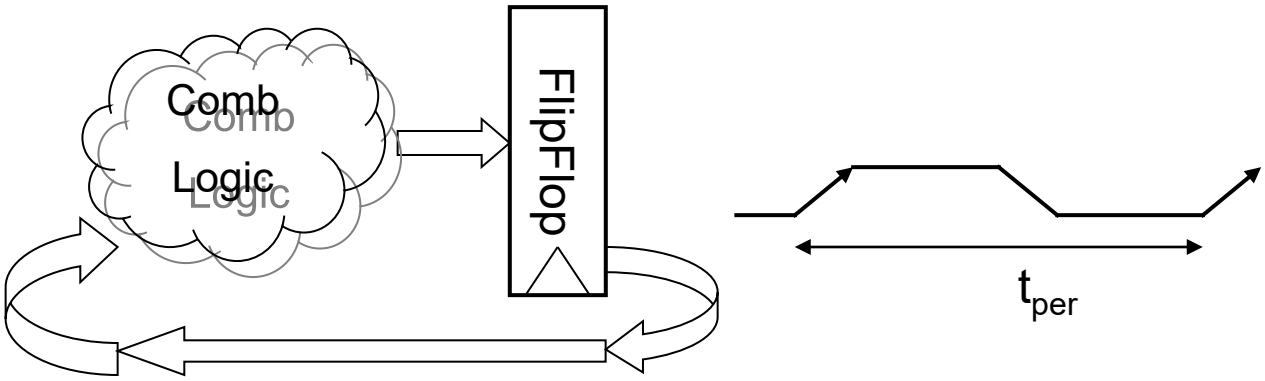
Setup, Hold, Cycle Times



Example of a single phase clock

Courtesy K. Keutzer et al. UCB

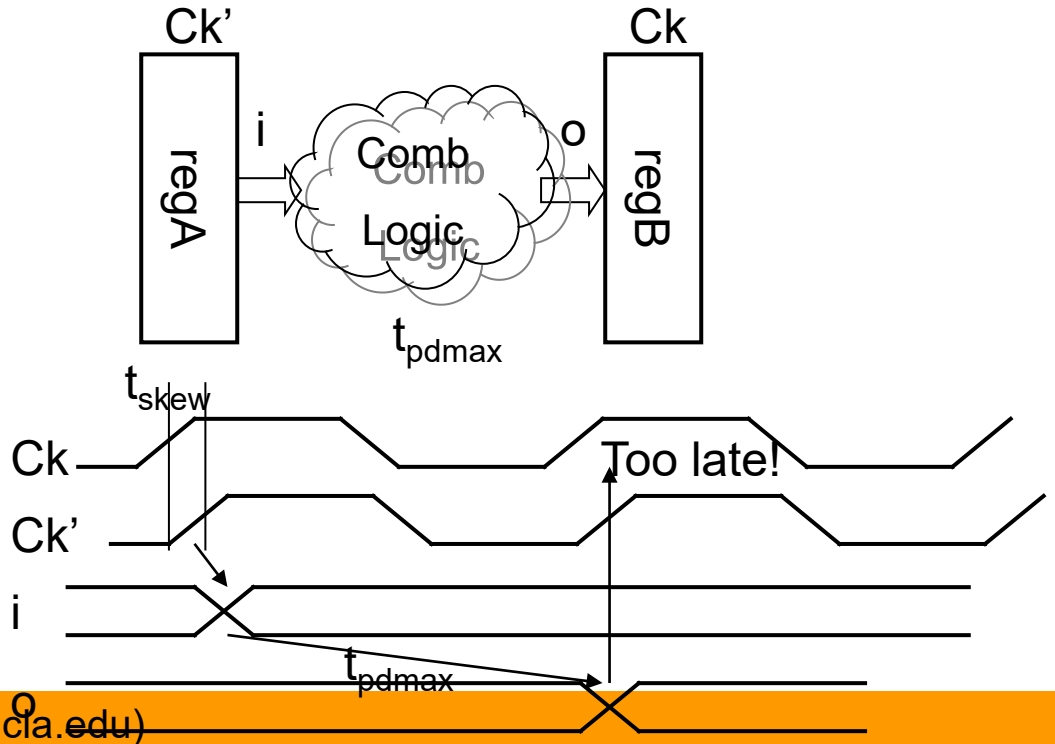
Summary of Constraints (Edge-Triggered FFs)



- $Max(t_{pd}) < t_{period} - t_{setup} - t_{c2q} - t_{skew}$
  - Delay is too long for data to be captured
- $Min(t_{pd}) > t_{hold} - t_{c2q} + t_{skew}$
  - Delay is too short and data can race through, skipping a state

Example of t_{pdmax} Violation

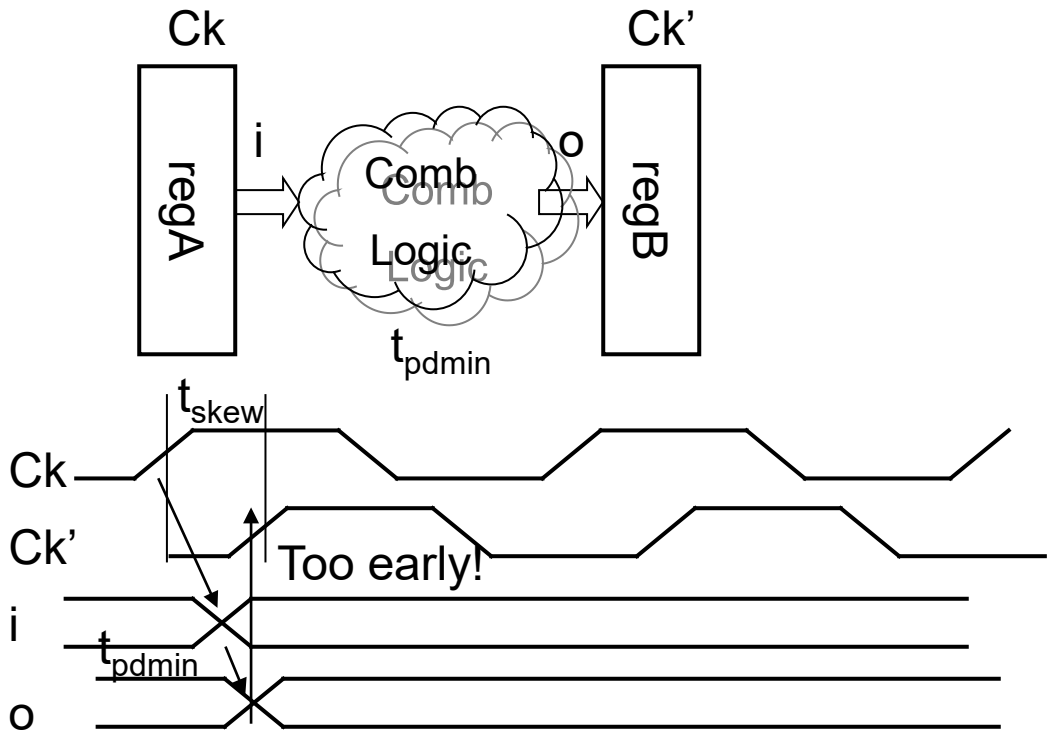
- Suppose there is skew between the registers in a dataflow (regA after regB)
- "i" gets its input values from regA at transition in Ck'
- CL output "o" arrives after Ck transition due to skew
- To correct this problem, can *increase* cycle time



Courtesy K. Yang, UCLA

Example of t_{pdmin} Violation: Race Through

- Suppose clock skew causes regA to be clocked before regB
- "i" passes through the CL with little delay (tpdmin)
- "o" arrives before the rising Ck' causes the data to be latched.
- Cannot be fixed by changing frequency → have rock instead of chip



Courtesy K. Yang, UCLA

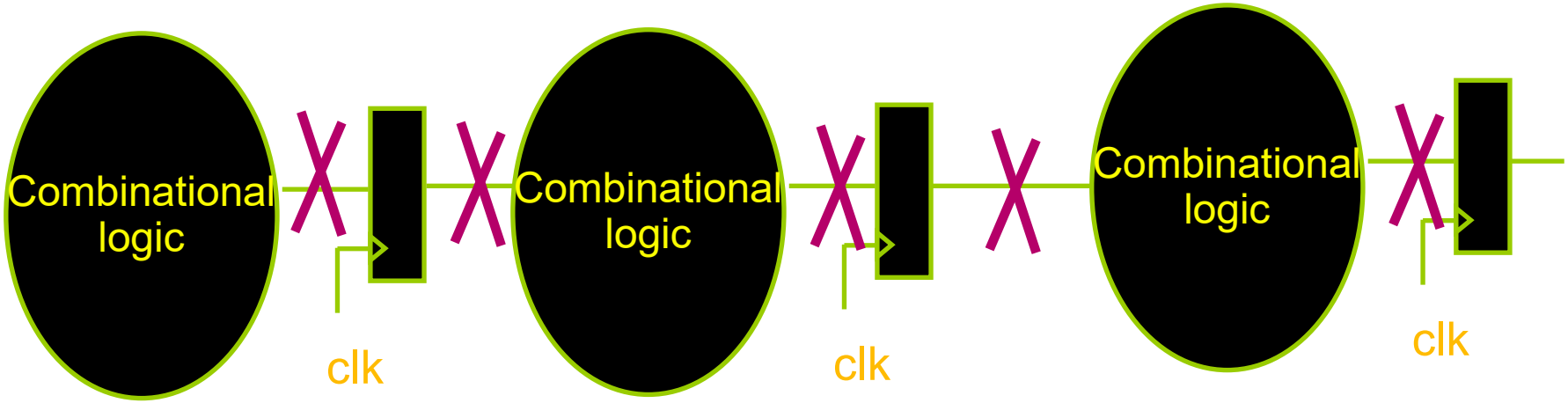
Timing Analysis for Digital Chips

- Need to figure out how fast the chip runs
  - Setup and hold checks
- Need to be able to analyze 1M+ gate design in seconds to minutes
  - Since will need to figure out how fast the chip runs many times during circuit optimizations
- Don't necessarily need to know if the chip is implementing the correct function at the same time
  - That can be verified separately

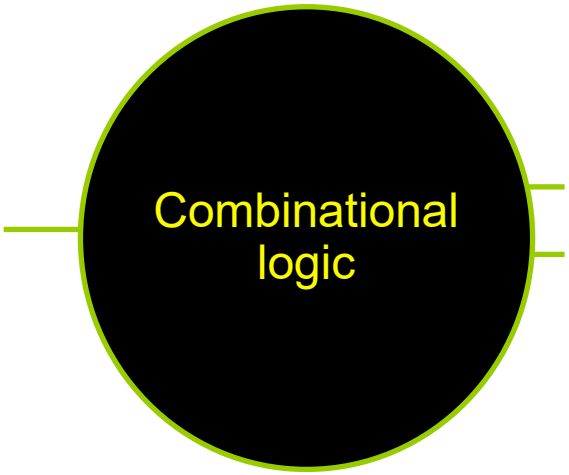
Why "Static"

- Dynamic timing analysis: input vector dependent
  - E.g., SPICE circuit simulation, Verilog simulation with timing
  - Accurate but..
  - Impractical for chips with 100s of inputs
- Static timing analysis: smart way of worst-casing vectors
  - No input vectors required
    - Modern timers take a lot of vector like hints
  - Tends to be pessimistic (though not always)

Approach – Reduce to Combinational



original circuit

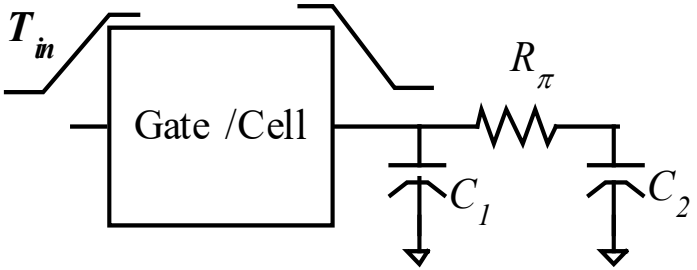


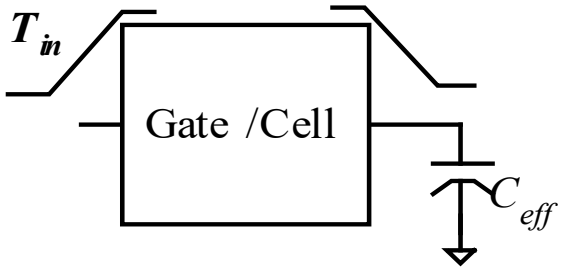
extracted block

Courtesy K. Keutzer et al. UCB

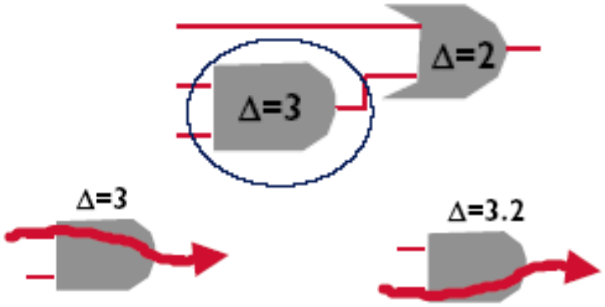
Gate Delay Models

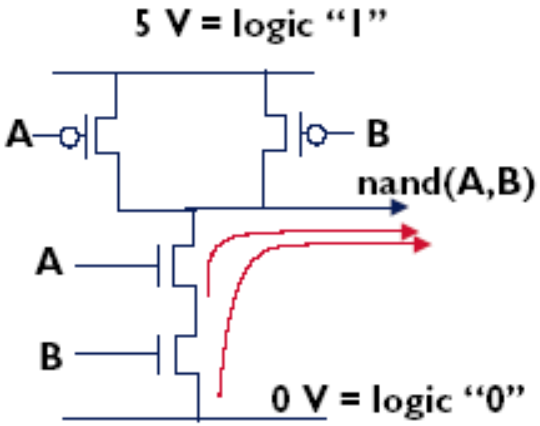
- Delay is a function of fanout/slew
  - Table based
  - Wire load is not just capacitance: wires have resistance! → common way is to use a "Effective Capacitance" model





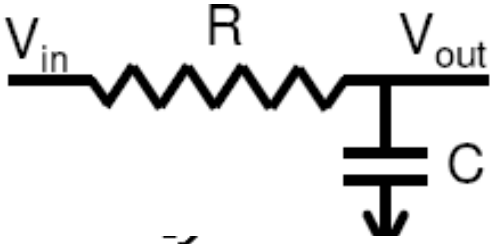
- Input pins are different



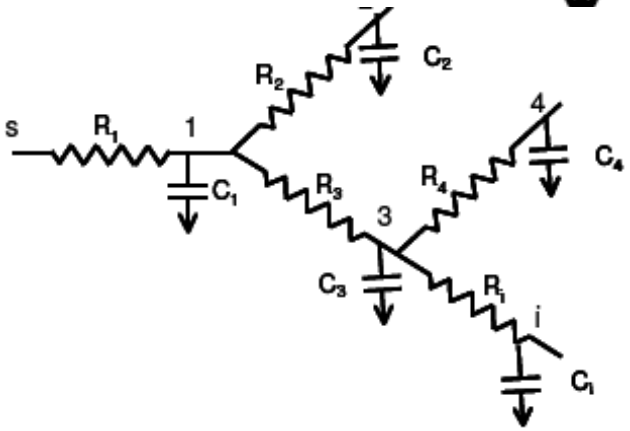


Interconnect Delay Model

- Interconnect delay becomes a dominant portion of total delay
- Lumped RC model



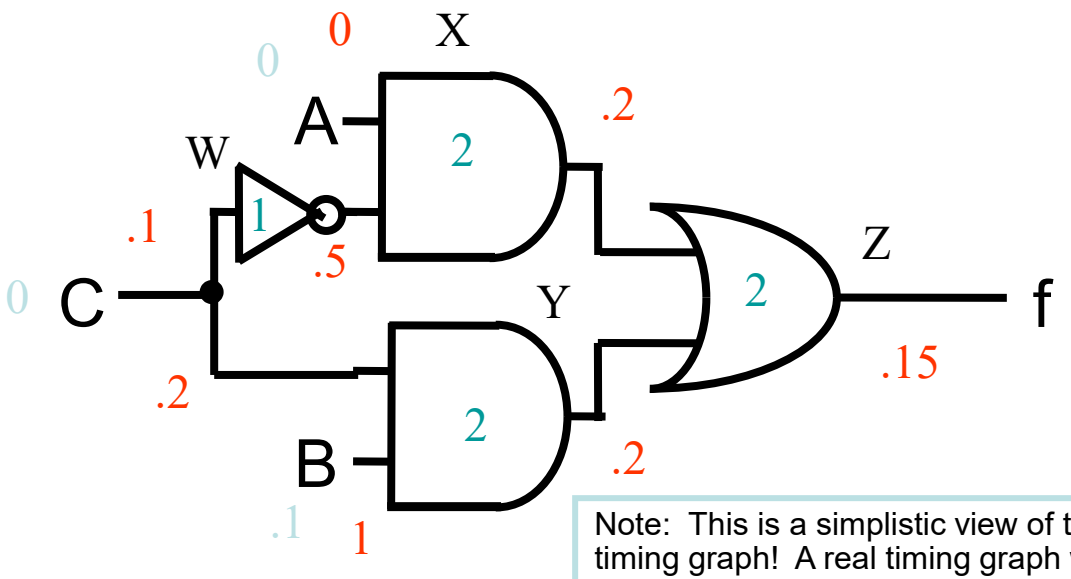
- Distributed RC tree



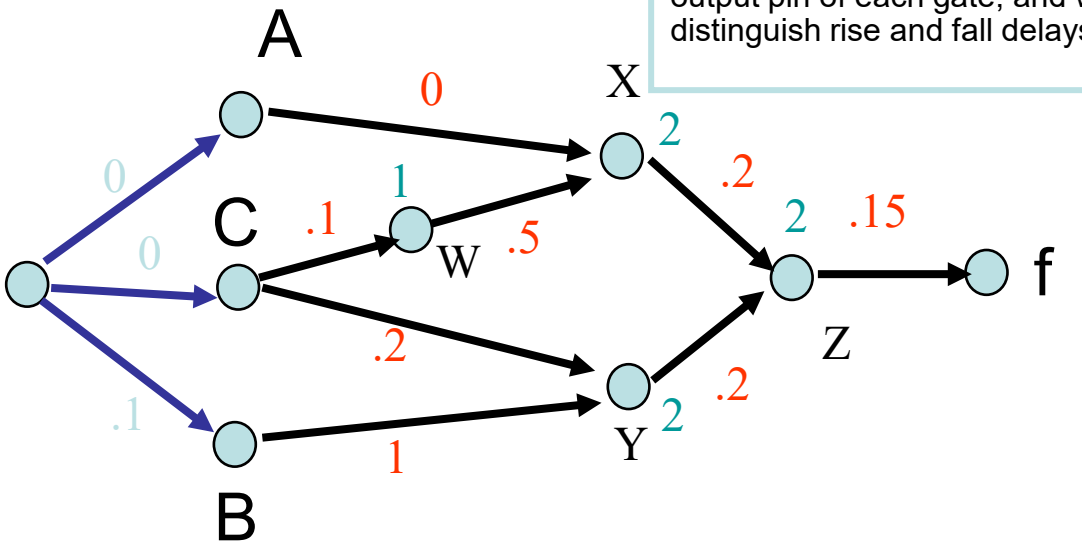
- For our purpose, we assume point-to-point wiring delays are pre-characterized as cell delays
  - Different interconnect have different delays
  - Example: Elmore Delay Model, AWE, etc (EE201C covers this)

Problem Formulation - 1

- Use a labeled *directed* graph
- $G = <V,E>$
- *Vertices* represent gates, primary inputs and primary outputs
- *Edges* represent wires
- *Labels* represent delays



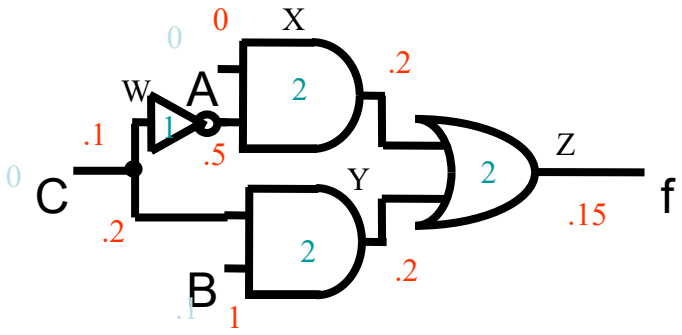
Note: This is a simplistic view of the timing graph! A real timing graph will (at least) have a vertex for each input or output pin of each gate, and will also distinguish rise and fall delays.

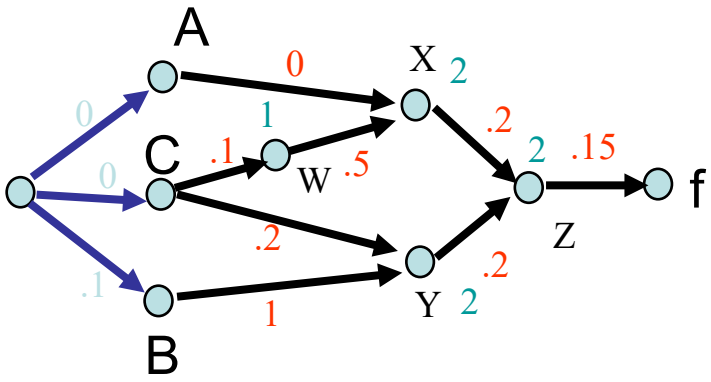


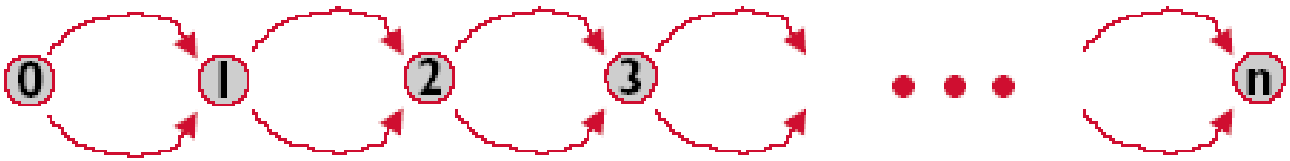
Courtesy K. Keutzer et al. UCB

Problem Formulation - 2

- Use a labeled *directed* graph
- $G = \langle V, E \rangle$
- *Find the circuit maximum delay*
- *Enumerate all paths - choose the longest?*
  - How many paths in this simple graph ?
    - 2^n

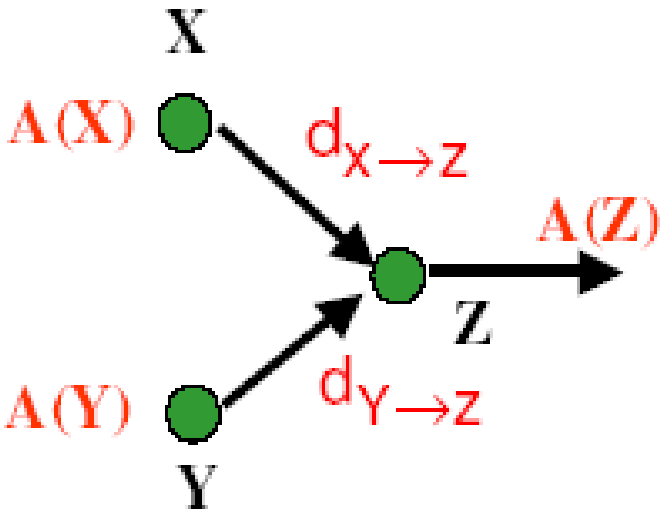






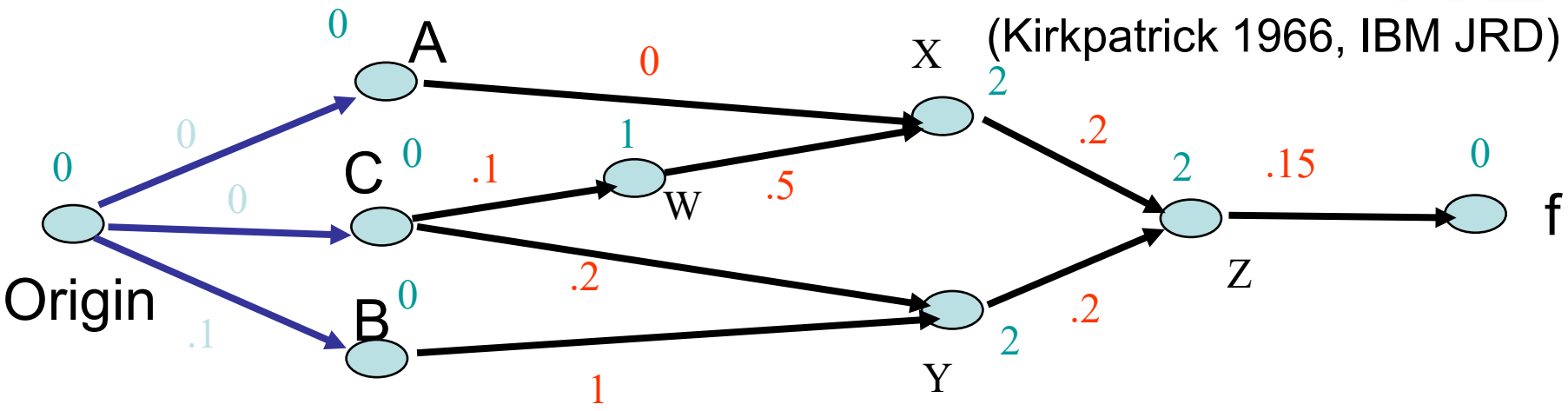
Block Based STA

- Arrival times (AT) at a node
  - Time when signal arrives at the node
  - Latest time signal can become stable
  - Determined by longest path from source



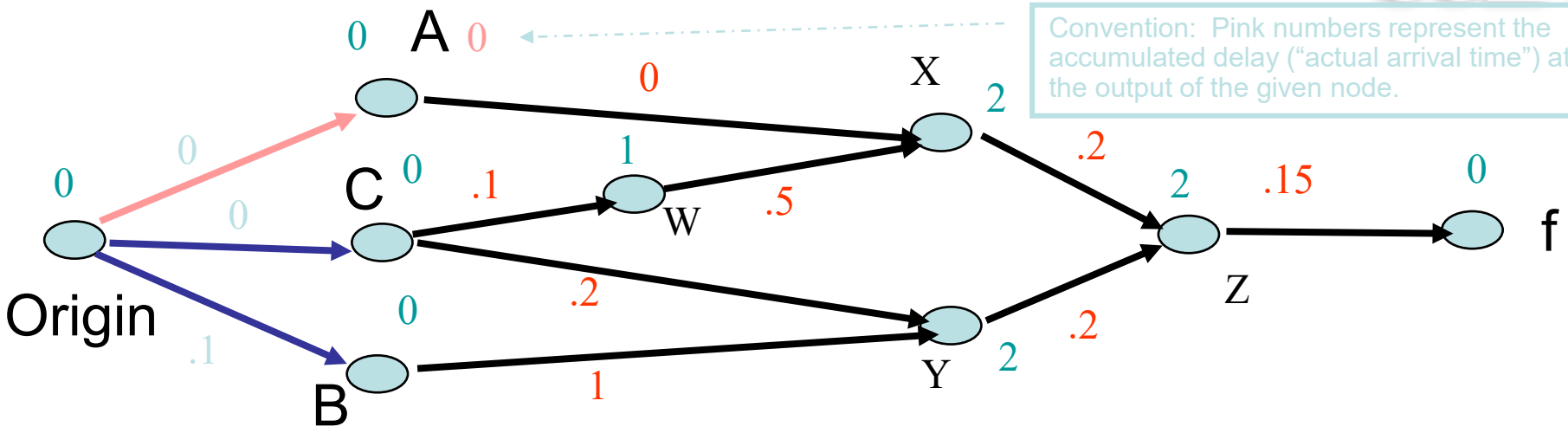
$$A(v) = \max_{u \in FI(v)} (A(u) + d_{u \to v})$$

Problem Formulation - 3



Compute longest path in a graph *G = <V,E,delay,Origin>*
// delay is set of labels, Origin is the super-source of the DAG
Forward-prop(*W*){
for each vertex *v* in *W*
for each edge <*v,w*> from *v*
Final-delay(w) = max(Final-delay(w), delay(v) + delay(w) + delay(<v,w>))
if all incoming edges of *w* have been traversed, add *w* to *W*
}
Longest path(G){
Forward_prop(Origin) }

Algorithm Execution



Compute longest path in a graph *G = <V,E,delay,Origin>*

// delay is set of labels, Origin is the super-source of the DAG

Forward-prop(*W*){

for each vertex *v* in *W*

for each edge <*v,w*> from *v*

Final-delay(w) = max(Final-delay(w), delay(v) + delay(w) + delay(<v,w>))

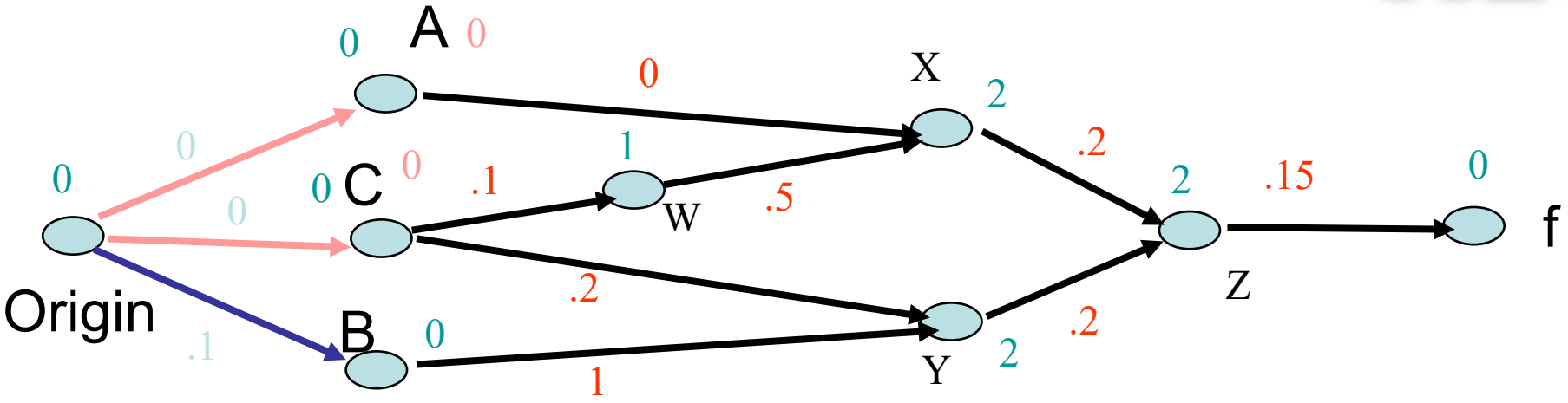
if all incoming edges of *w* have been traversed, add *w* to *W*

}

Longest path(G){

Forward_prop(Origin) }

Algorithm Execution



Compute longest path in a graph *G = <V,E,delay,Origin>*

// delay is set of labels, Origin is the super-source of the DAG

Forward-prop(*W*){

for each vertex *v* in *W*

for each edge <*v,w*> from *v*

Final-delay(w) = max(Final-delay(w), delay(v) + delay(w) + delay(<v,w>))

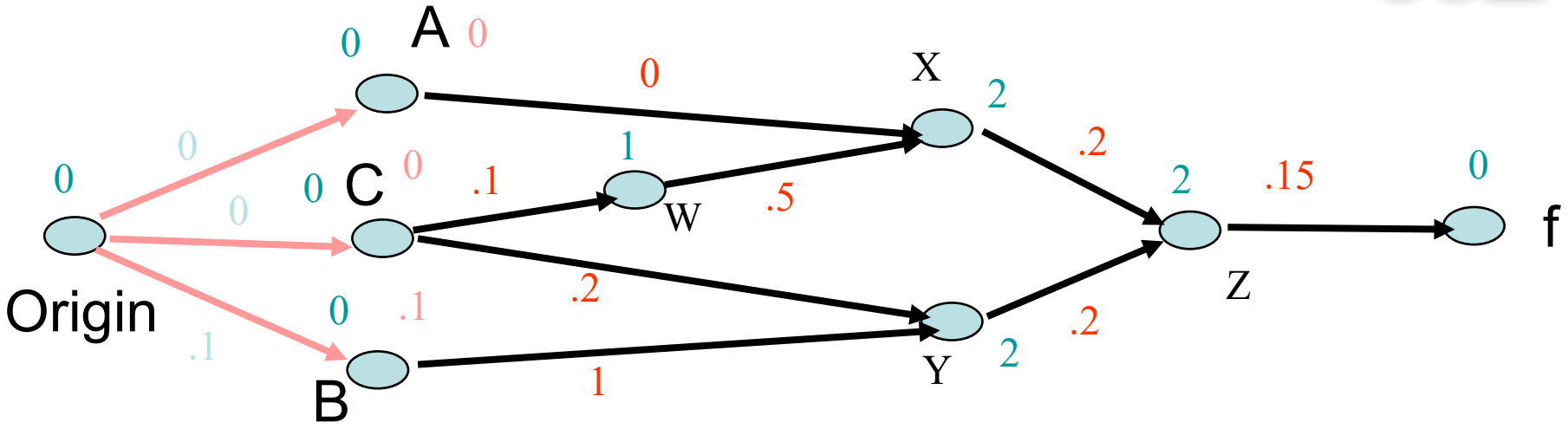
if all incoming edges of *w* have been traversed, add *w* to *W*

}

Longest path(G){

Forward_prop(Origin) }

Algorithm Execution



Compute longest path in a graph *G = <V,E,delay,Origin>*

// delay is set of labels, Origin is the super-source of the DAG

Forward-prop(*W*){

for each vertex *v* in *W*

for each edge <*v,w*> from *v*

Final-delay(w) = max(Final-delay(w), delay(v) + delay(w) + delay(<v,w>))

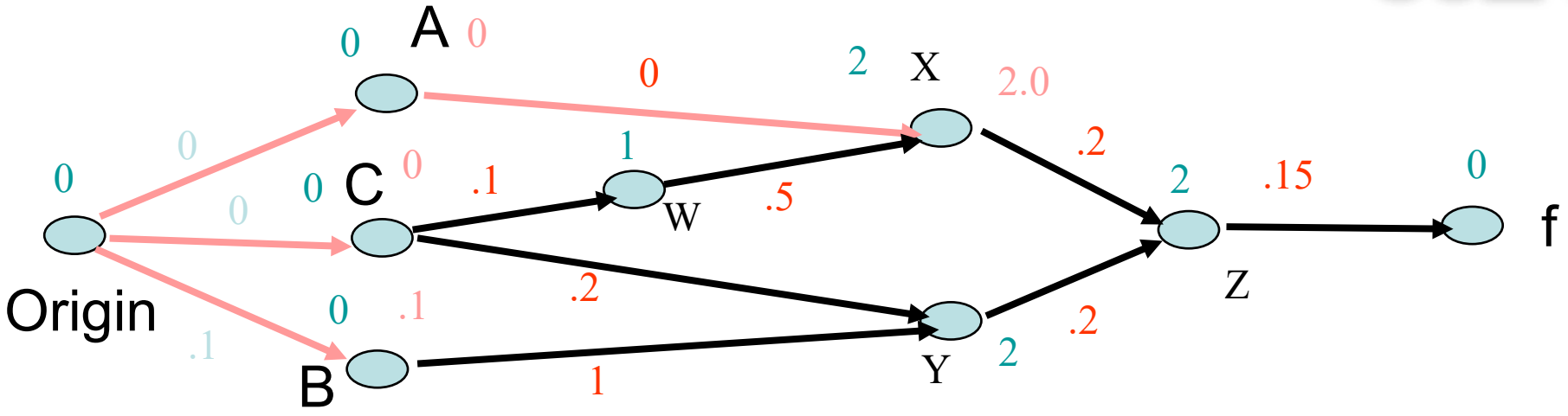
if all incoming edges of *w* have been traversed, add *w* to *W*

}

Longest path(G){

Forward_prop(Origin) }

Algorithm Execution



Compute longest path in a graph *G = <V,E,delay,Origin>*

// delay is set of labels, Origin is the super-source of the DAG

Forward-prop(*W*){

for each vertex *v* in *W*

for each edge <*v,w*> from *v*

Final-delay(w) = max(Final-delay(w), delay(v) + delay(w) + delay(<v,w>))

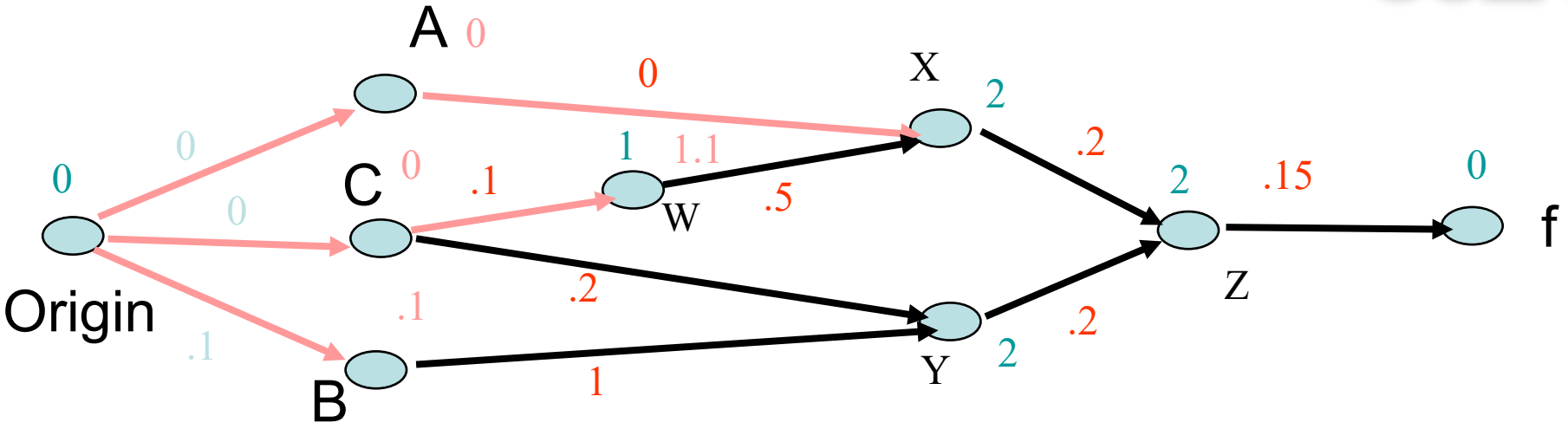
if all incoming edges of *w* have been traversed, add *w* to *W*

}

Longest path(G){

Forward_prop(Origin) }

Algorithm Execution



Compute longest path in a graph *G = <V,E,delay,Origin>*

// delay is set of labels, Origin is the super-source of the DAG

Forward-prop(*W*){

for each vertex *v* in *W*

for each edge <*v,w*> from *v*

Final-delay(w) = max(Final-delay(w), delay(v) + delay(w) + delay(<v,w>))

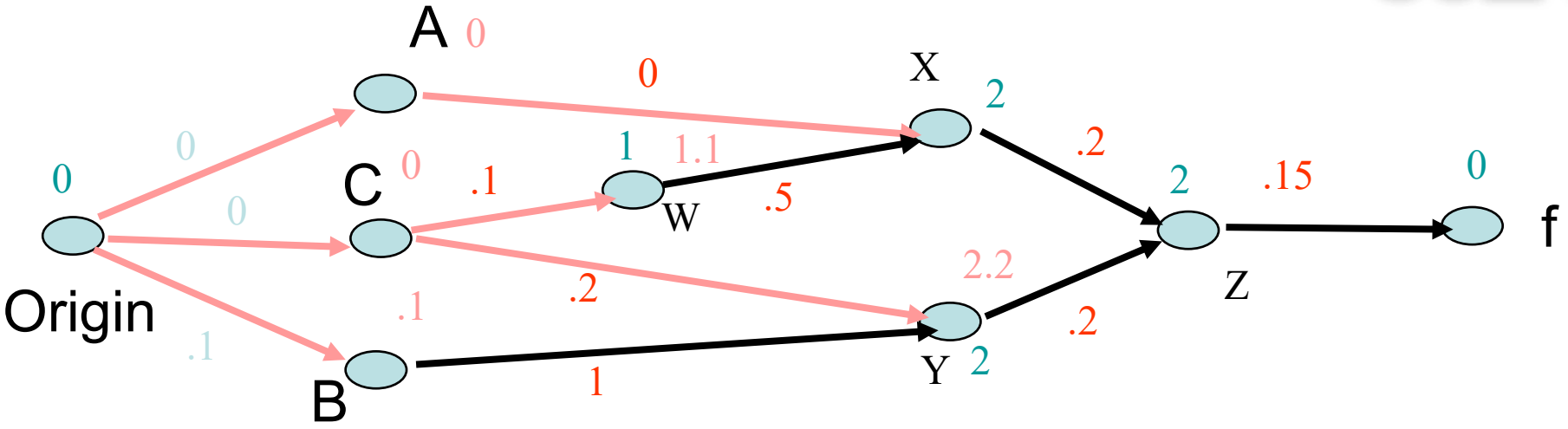
if all incoming edges of *w* have been traversed, add *w* to *W*

}

Longest path(G){

Forward_prop(Origin) }

Algorithm Execution



Compute longest path in a graph *G = <V,E,delay,Origin>*

// delay is set of labels, Origin is the super-source of the DAG

Forward-prop(*W*){

for each vertex *v* in *W*

for each edge <*v,w*> from *v*

Final-delay(w) = max(Final-delay(w), delay(v) + delay(w) + delay(<v,w>))

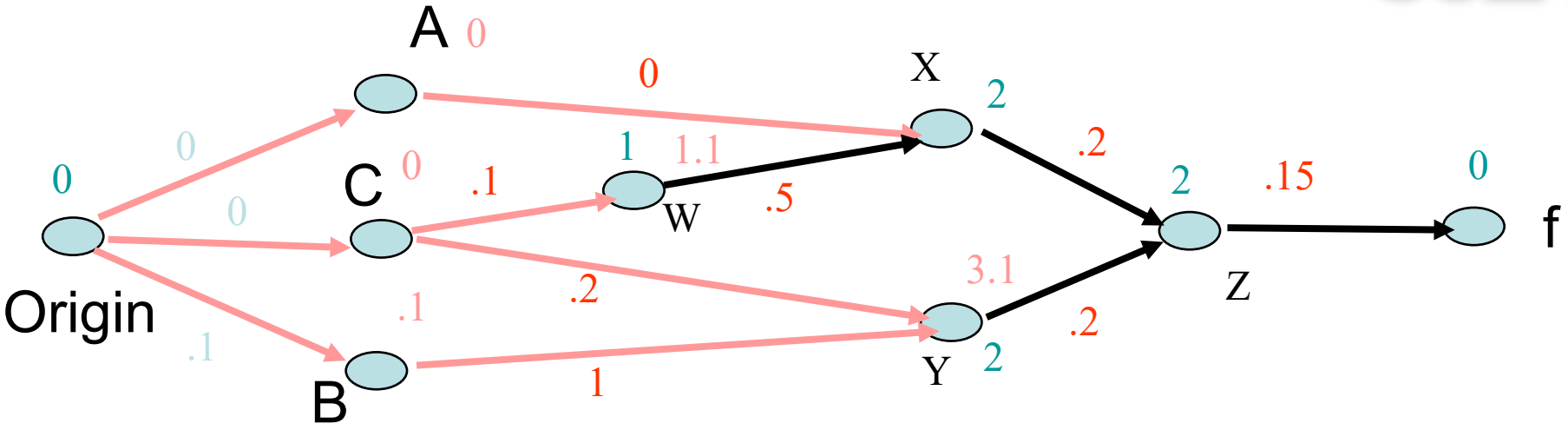
if all incoming edges of *w* have been traversed, add *w* to *W*

}

Longest path(G){

Forward_prop(Origin) }

Algorithm Execution



Compute longest path in a graph *G = <V,E,delay,Origin>*

// delay is set of labels, Origin is the super-source of the DAG

Forward-prop(*W*){

for each vertex *v* in *W*

for each edge <*v,w*> from *v*

Final-delay(w) = max(Final-delay(w), delay(v) + delay(w) + delay(<v,w>))

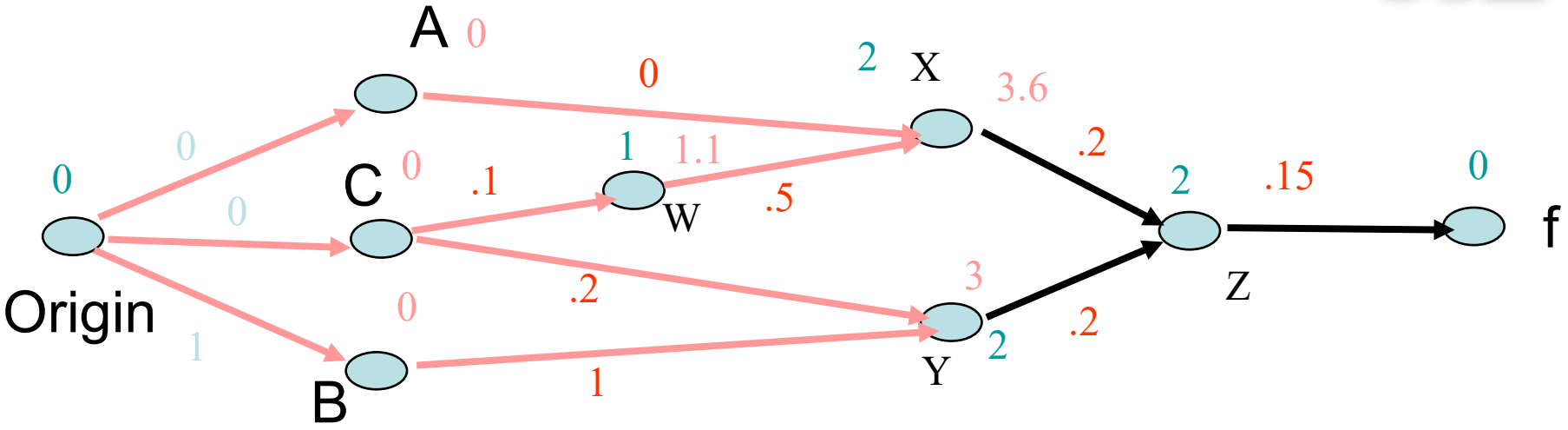
if all incoming edges of *w* have been traversed, add *w* to *W*

}

Longest path(G){

Forward_prop(Origin) }

Algorithm Execution



Compute longest path in a graph *G = <V,E,delay,Origin>*

// delay is set of labels, Origin is the super-source of the DAG

Forward-prop(*W*){

for each vertex *v* in *W*

for each edge <*v,w*> from *v*

Final-delay(w) = max(Final-delay(w), delay(v) + delay(w) + delay(<v,w>))

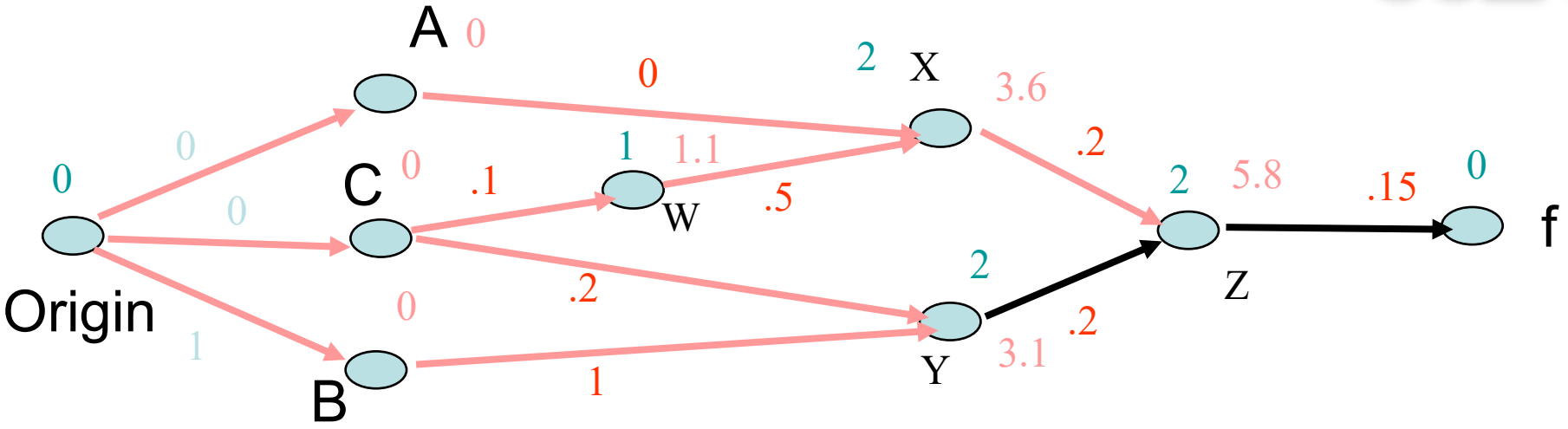
if all incoming edges of *w* have been traversed, add *w* to *W*

}

Longest path(G){

Forward_prop(Origin) }

Algorithm Execution



Compute longest path in a graph *G = <V,E,delay,Origin>*

// delay is set of labels, Origin is the super-source of the DAG

Forward-prop(*W*){

for each vertex *v* in *W*

for each edge <*v,w*> from *v*

Final-delay(w) = max(Final-delay(w), delay(v) + delay(w) + delay(<v,w>))

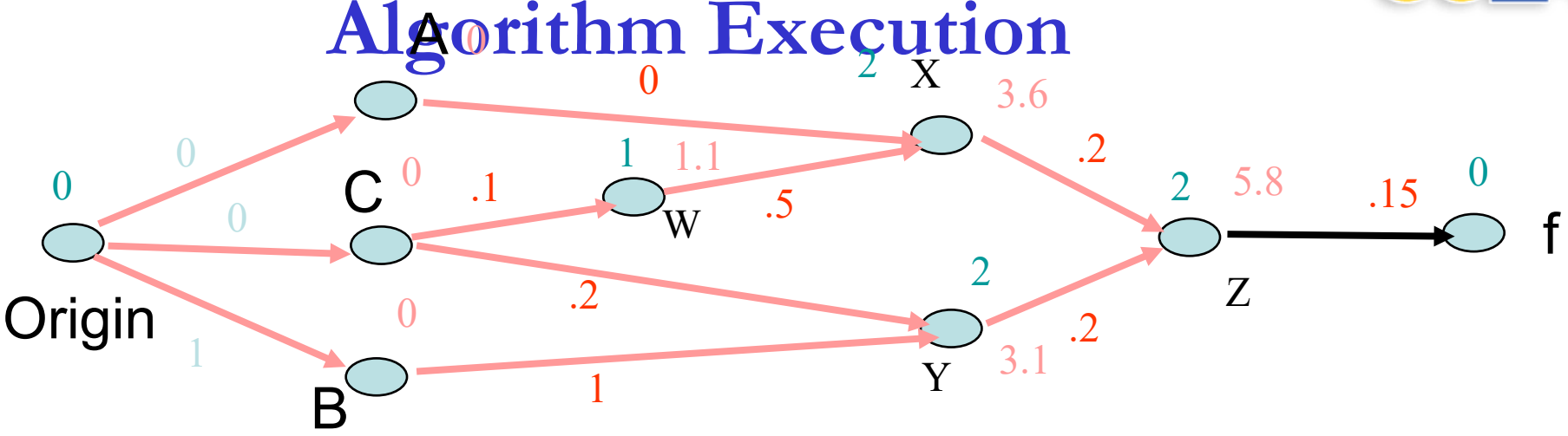
if all incoming edges of *w* have been traversed, add *w* to *W*

}

Longest path(G){

Forward_prop(Origin) }

Algorithm Execution



Compute longest path in a graph *G = <V,E,delay,Origin>*

// delay is set of labels, Origin is the super-source of the DAG

Forward-prop(*W*){

for each vertex *v* in *W*

for each edge <*v,w*> from *v*

Final-delay(w) = max(Final-delay(w), delay(v) + delay(w) + delay(<v,w>))

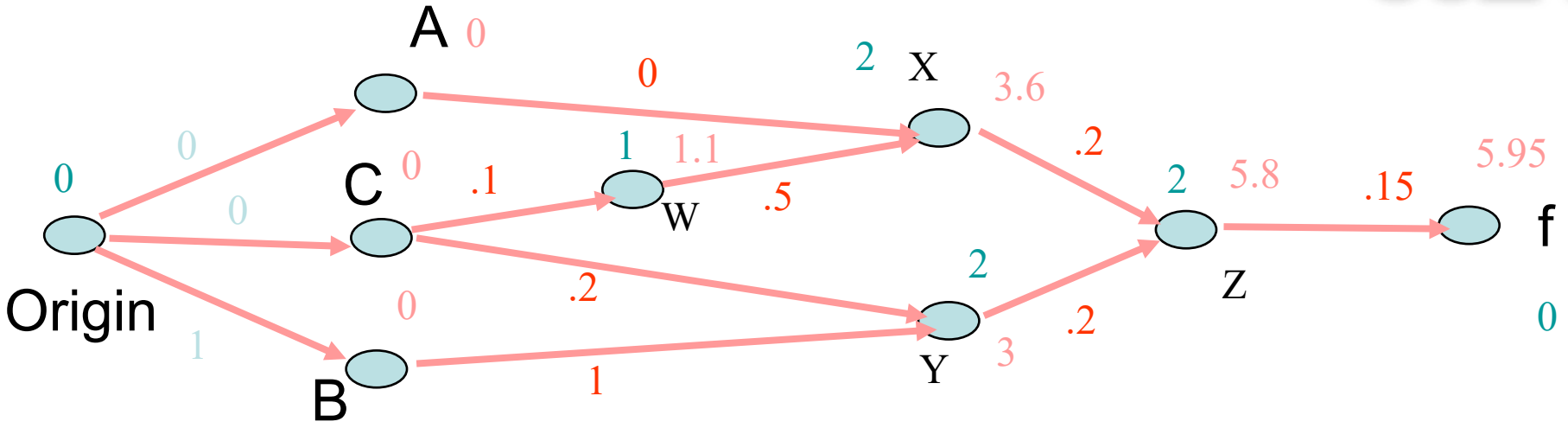
if all incoming edges of *w* have been traversed, add *w* to *W*

}

Longest path(G){

Forward_prop(Origin) }

Algorithm Execution



Compute longest path in a graph *G = <V,E,delay,Origin>*

// delay is set of labels, Origin is the super-source of the DAG

Forward-prop(*W*){

for each vertex *v* in *W*

for each edge <*v,w*> from *v*

Final-delay(w) = max(Final-delay(w), delay(v) + delay(w) + delay(<v,w>))

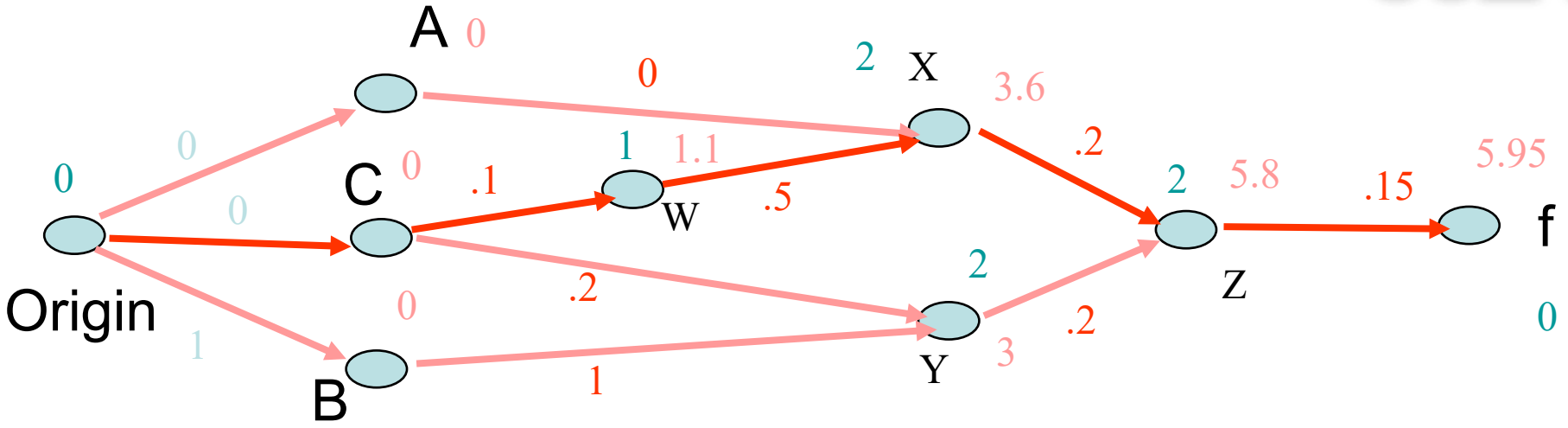
if all incoming edges of *w* have been traversed, add *w* to *W*

}

Longest path(G){

Forward_prop(Origin) }

Critical Path (sub-graph)



Compute longest path in a graph *G = <V,E,delay,Origin>*

// delay is set of labels, Origin is the super-source of the DAG

Forward-prop(*W*){

for each vertex *v* in *W*

for each edge <*v,w*> from *v*

Final-delay(w) = max(Final-delay(w), delay(v) + delay(w) + delay(<v,w>))

if all incoming edges of *w* have been traversed, add *w* to *W*

}

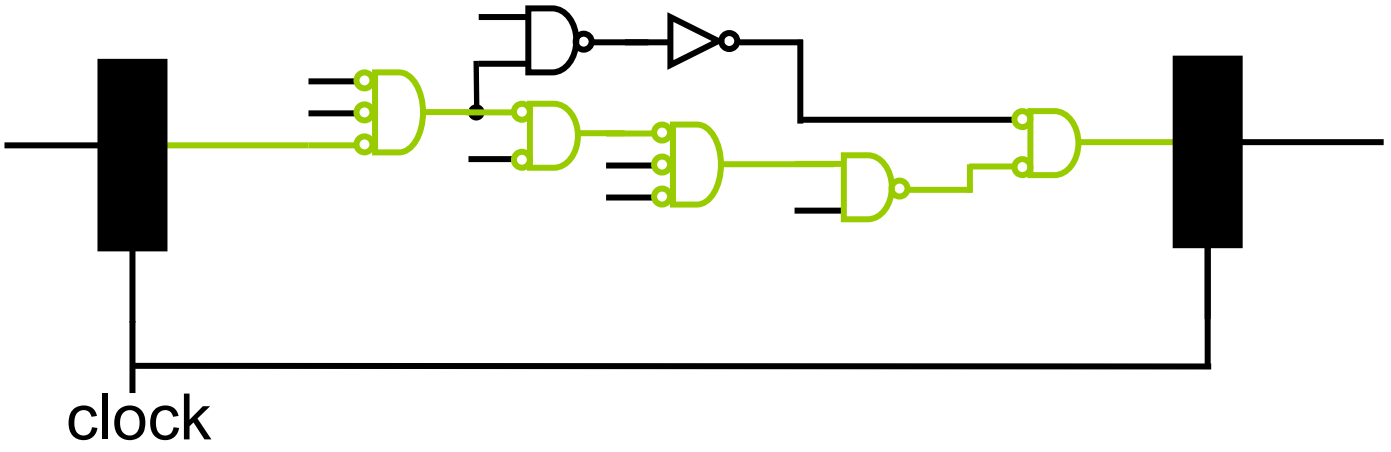
Longest path(G){

Forward_prop(Origin) }

5 min break

Timing for Optimization: Extra Requirements

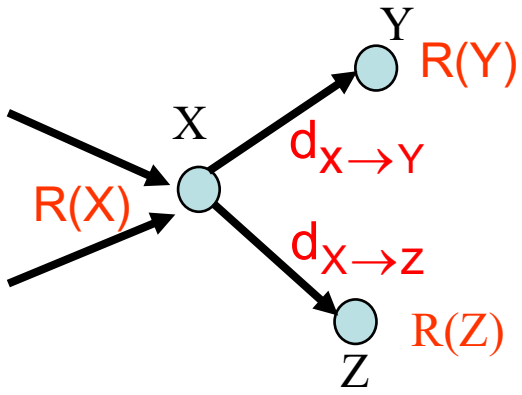
- Longest-path algorithm computes arrival times at each node
- If we have constraints, need to propagate slack to each *node*
  - A measure of how much timing margin exists at each node
  - Can optimize a particular branch
- Can trade slack for power, area, robustness



Courtesy K. Keutzer et al. UCB

Required Arrival Time

- Required arrival time R(v) is the time before which a signal must arrive to avoid a timing violation



Required time is user defined at output:

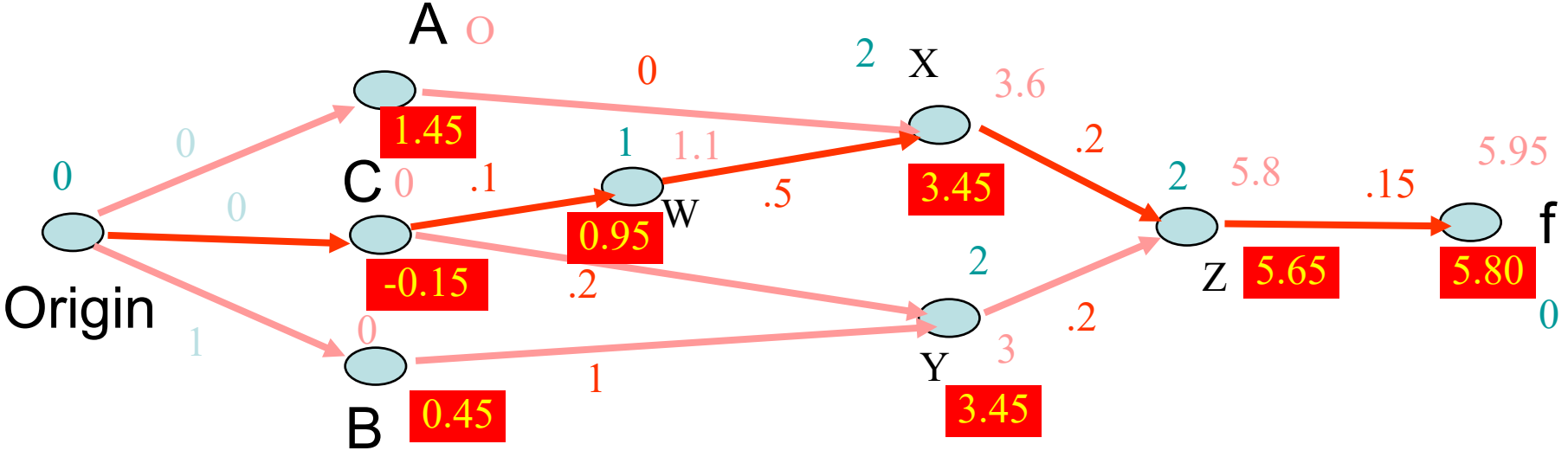
$$R(v) = T - T_{setup}$$

- Then recursively

$$R(\upsilon) = \min_{u \in FO(\upsilon)} (R(u) - d_{\upsilon \to u})$$

where $FO(\upsilon) = \{Y, Z\}$ and $\upsilon = \{X\}$

Required Time Propagation: Example



- Assume required time at output R(f) = 5.80
- Propagate required times backwards

Timing Slack

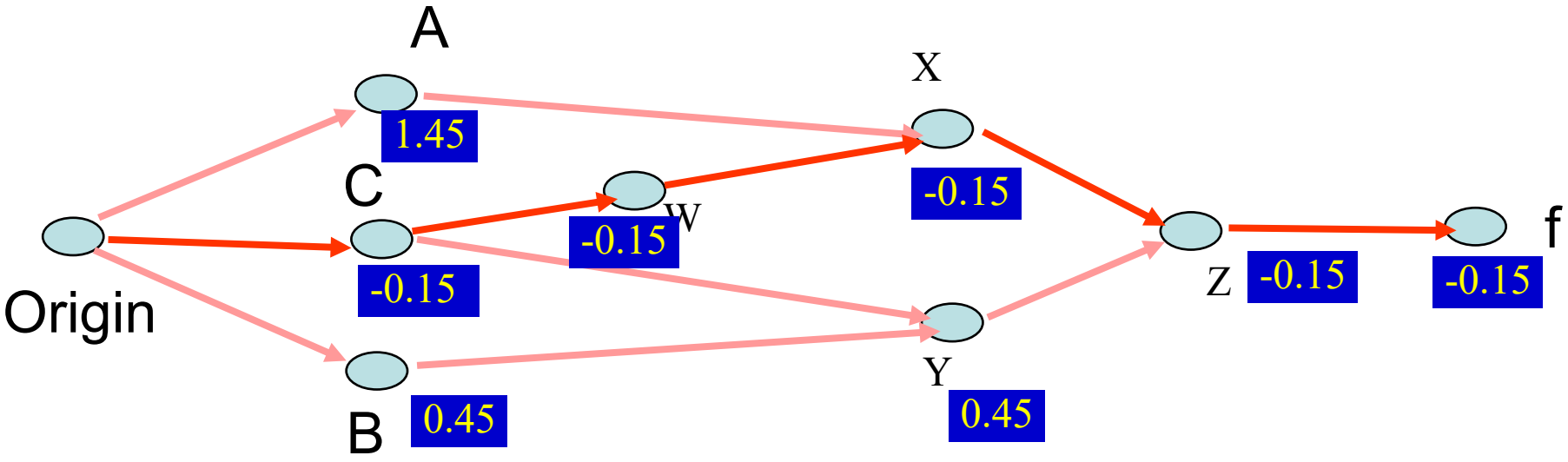
- From arrival and required time can compute slack. For each node v:

$$S(\upsilon) = R(\upsilon) - A(\upsilon)$$

- Slack reflects criticality of a node
- Positive slack
  - Node is not on critical path. Timing constraints met.
- Zero slack
  - Node is on critical path. Timing constraints are barely met.
- Negative slack
  - There is a timing violation
- Slack distribution is key for timing optimization!

Courtesy K. Keutzer et al. UCB

Timing Slack Computation: Example



- Compute slack at each node

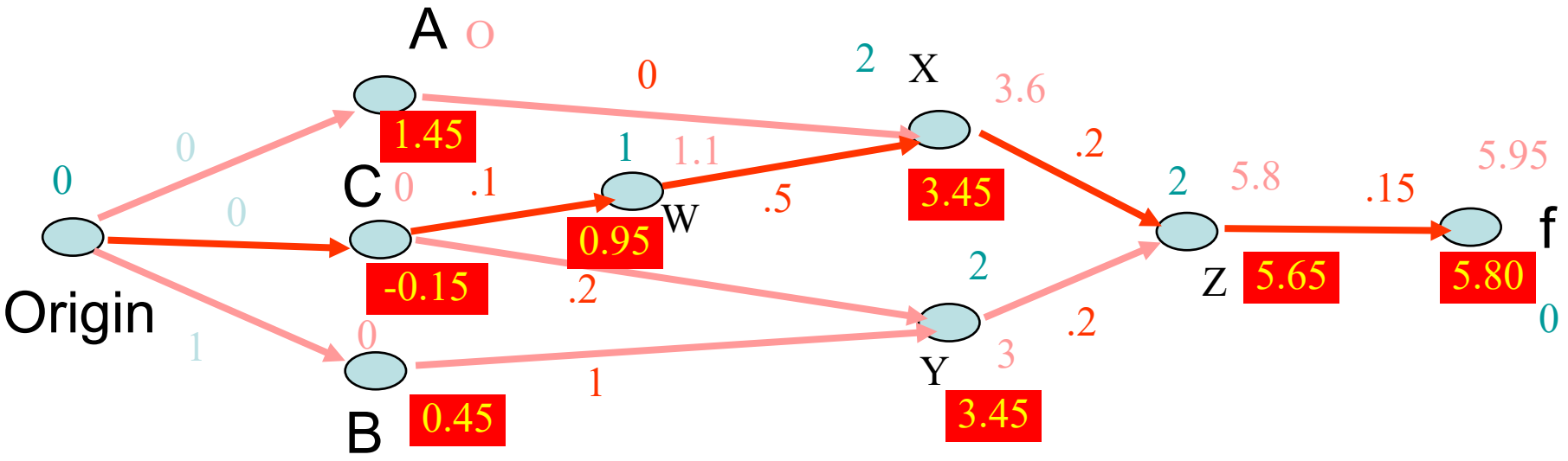
$$S(\upsilon) = R(\upsilon) - A(\upsilon)$$

Enhancements to STA

- Incremental timing analysis
  - What happens if you change size of one gate ?
- Conservatism
  - Multiple inputs switching
  - Interference – crosstalk → coupling induce delay
  - Nanometer-scale process effects – variation (→ probabilistic timing analysis)
  - False paths
- Timing correction driven by STA
  - Resize cells
  - Buffer nets
  - Copy (clone) cells

Courtesy K. Keutzer et al. UCB

Incremental Timing

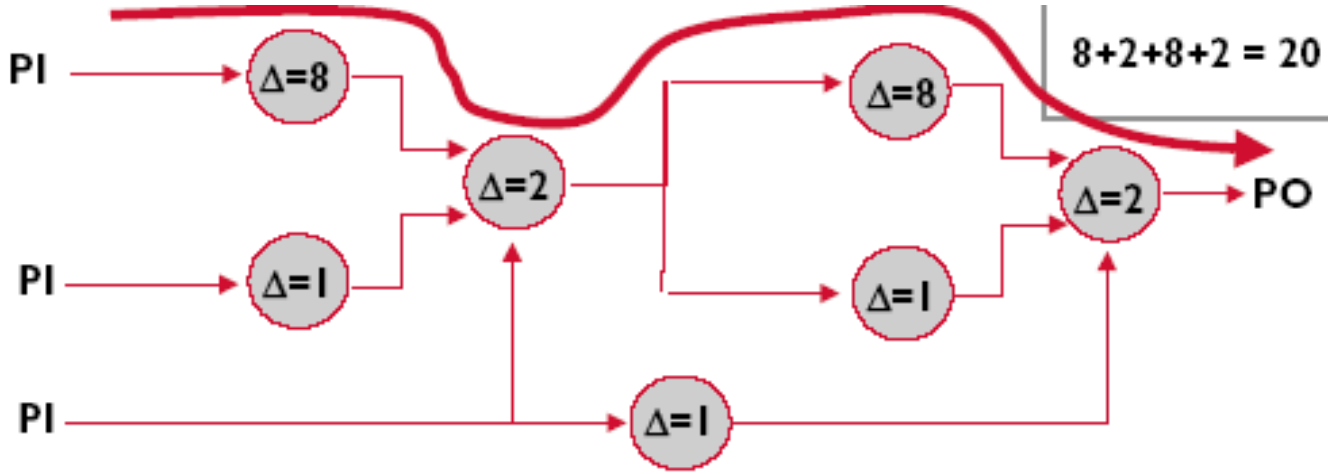


- What needs to be recomputed if gate X is sized up ?
  - Delay?
  - AT ?
  - RT ?
- On a large circuit the difference can be several orders of magnitude…

Courtesy K. Keutzer et al. UCB

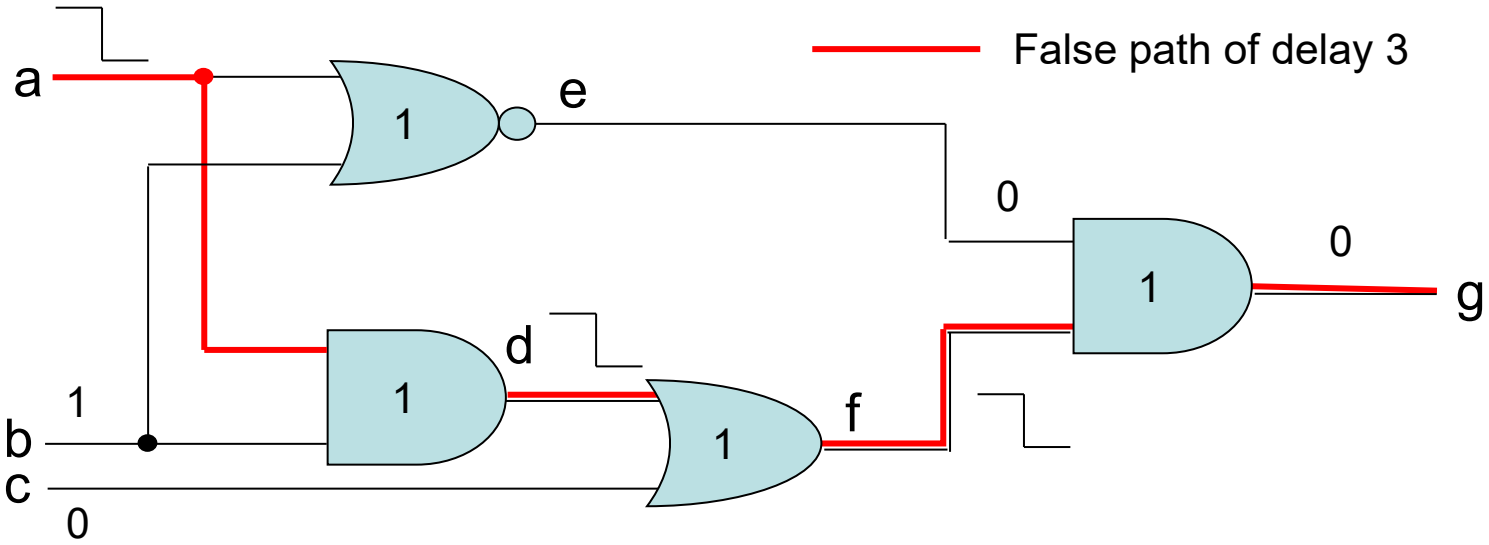
False Paths in STA

- Logical vs. Topological → logic functionality does not matter
- A major (critical) assumption hidden behind STA!
  - All paths are sensitizable:
    - There always exists a set of inputs that will cause the logic propagating along any chosen path



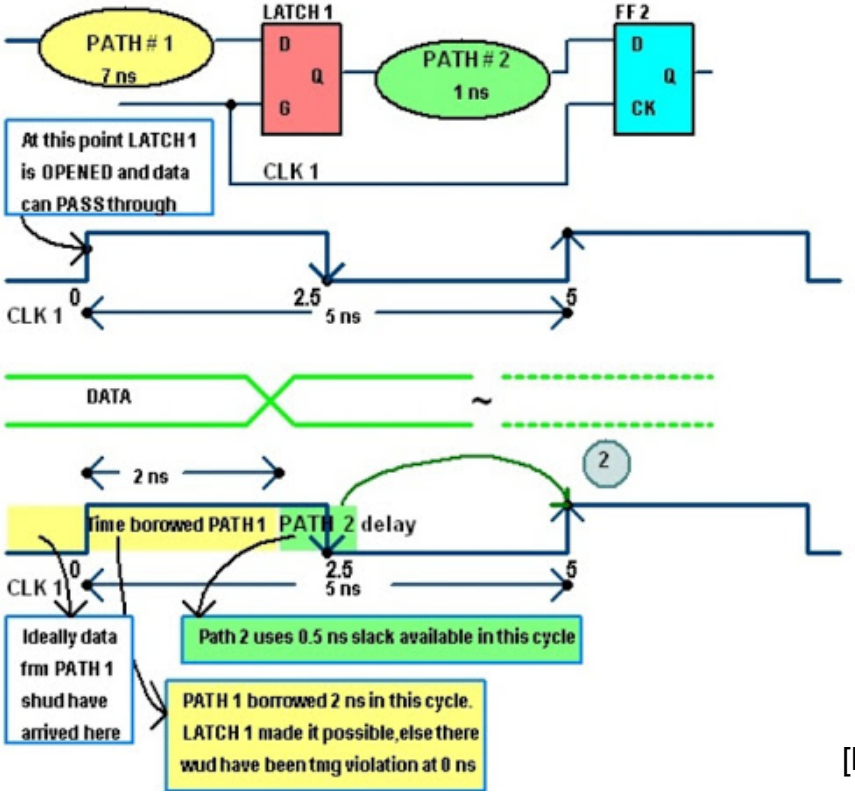
- But this is NOT true in general

False Path Example



- Our previous algorithm will identify the red path as the longest path!
  - Is this path sensitizable?
- Identifying false paths tough: As hard as Boolean Satisfiability (SAT) problem (NP-hard)
  - Several effective heuristics exist
  - Designer guidance is also common (set_false_path)

Time borrowing/cycle stealing

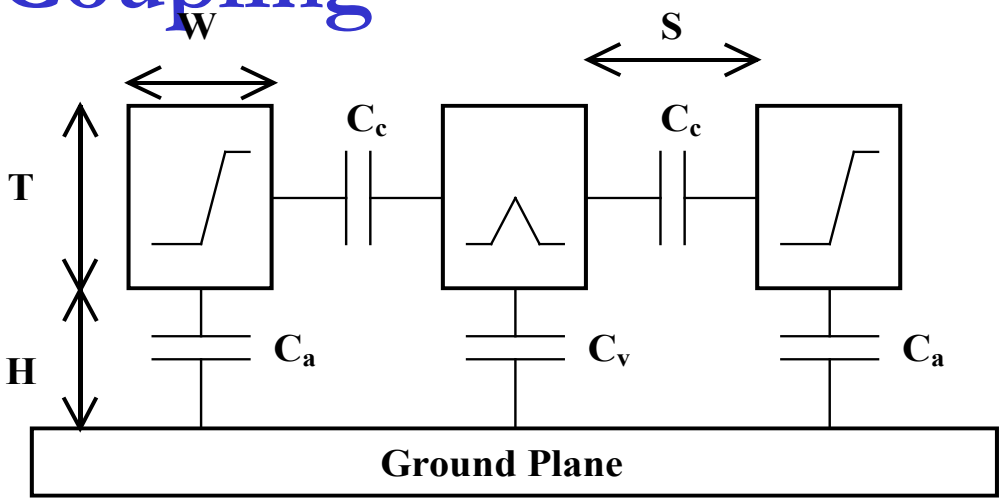


[http://ohotspot.blogspot.com/]

- If these two were FFs, we will have a setup violation. Latches can "borrow" time from previous stage → need to keep track of time borrowing over many stages in STA

Capacitive Coupling

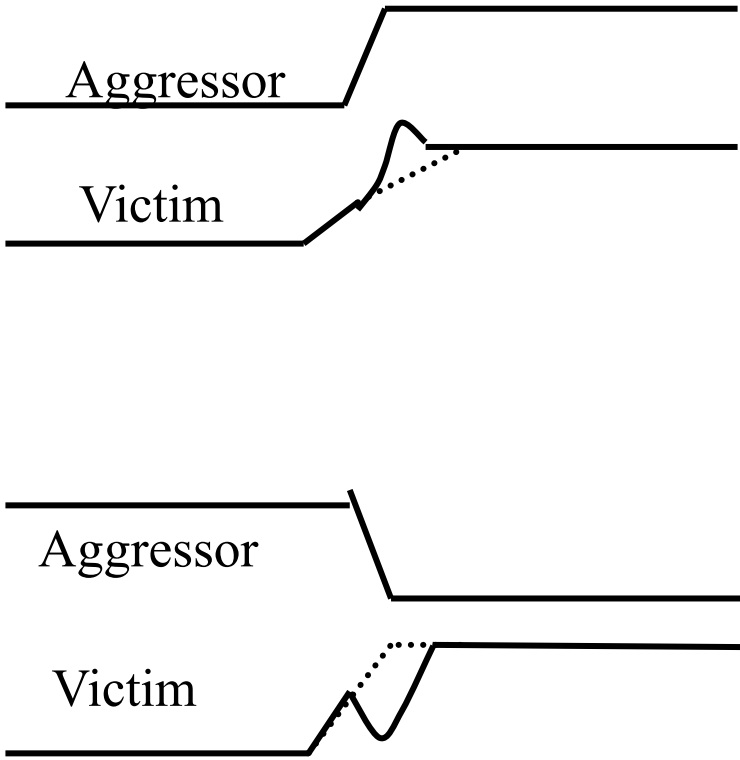
- Cross-section view



- On-chip wires have significant capacitance to adjacent wires
  - On the same layer
  - On adjacent layers
  - Wire under consideration: Victim; other wires with coupling to victim: aggressor
- Charge injected across Cc results in temporary (in static logic) glitch in voltage from the supply rail at the victim

Crosstalk: Timing Impact

- A switching victim is sped up by a coupled aggressor that is switching in the same direction
  - Potential "hold time" violation
  - Fixes include adding delay elements to your path
- A switching victim is hindered (slowed down) by a coupled aggressor that is switching in the opposite direction
  - Potential setup time violation
  - Fixes include spacing the wires, using stronger drivers



Capacitive Coupling

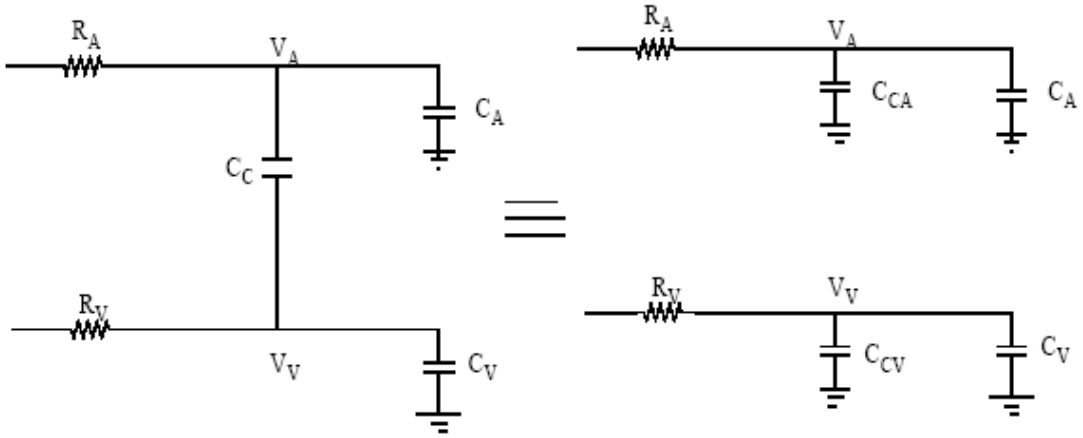
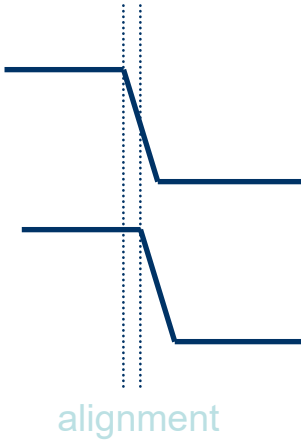


Figure 3: Equivalent circuit for two coupled lines

- $Q = C_c (\Delta V_v - \Delta V_A)$ = charge delivered to coupling capacitor
- A switches but V does not: $\Delta V = V_{DD}$, A sees cap to ground and to B
  - "Miller Coupling Factor" (MCF) = 1
- A and V switch in same direction: no voltage change: $\Delta V = 0$, C_{adj} is effectively absent
  - MCF = 0
- A and V switch in opposite directions: voltage change $\Delta V = 2V_{DD}$, twice as much charge is required, C_{adj} is effectively doubled
  - MCF = 2

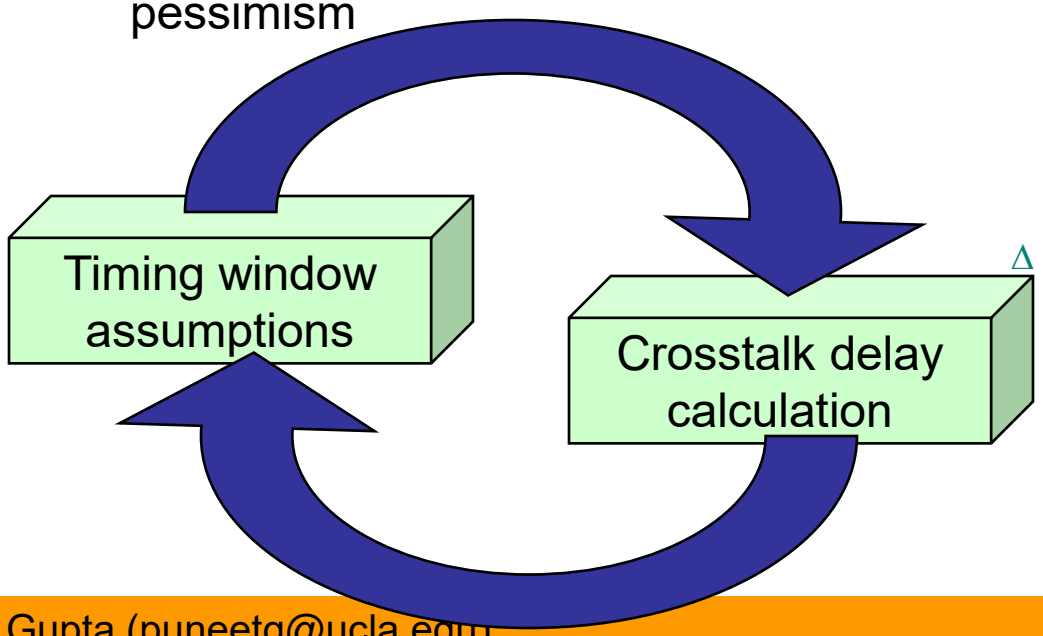
Crosstalk Delay Calculation: Levels of Accuracy

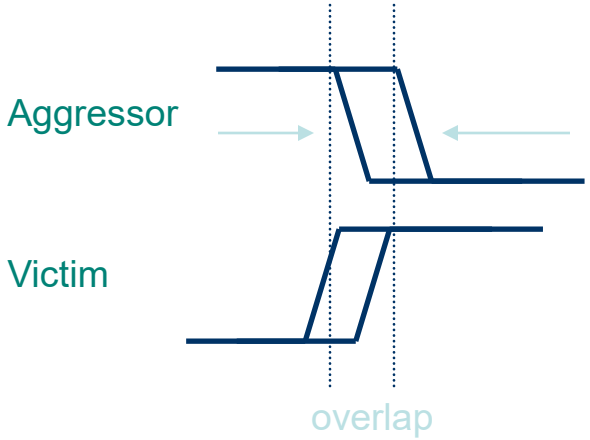
- Discard coupling capacitances or ground them (MCF=0)
- De-coupling by replacing coupling caps by conservative caps (MCF = 2 for setup; MCF=0 for hold)
  - True worst-case can be MCF = 3 for setup and MCF= -1 for hold
    - Why ??
- De-coupling by Miller factors on a per net basis
  - MCF depends on aggressor alignment, slews
- More sophisticated (but very slow) methods
  - E.g., Simulating multi-input multi-output networks

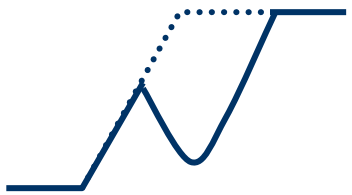


Calculation Flow

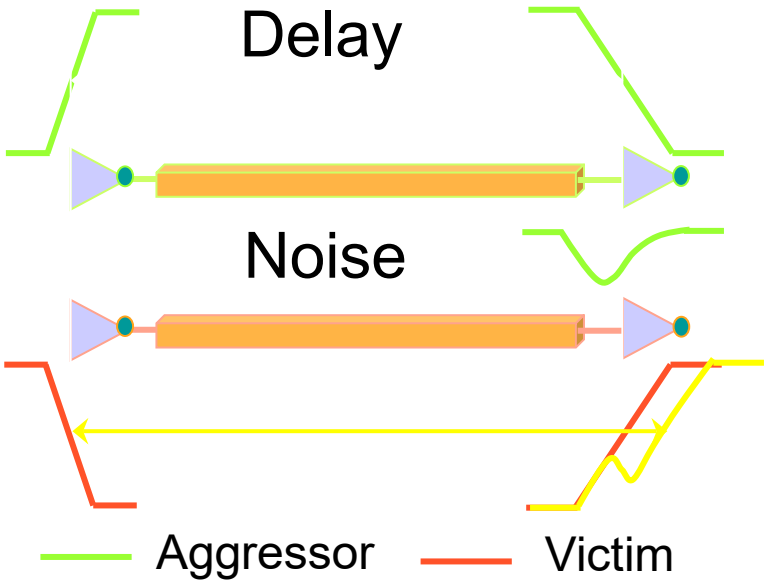
- Timing window overlaps enable crosstalk delay variation
- Chicken-egg situation: delay vs. crosstalk
- Iteration
  - Starting with the assumption that all timing windows are overlapped (pessimistic about the unknowns)
  - Refine calculation by reducing pessimism

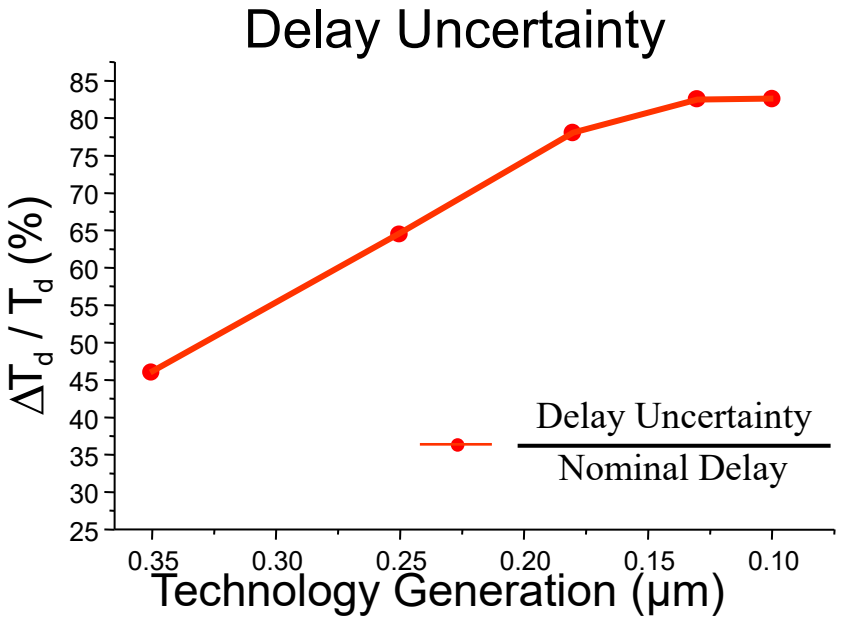






Scaling of Delay Uncertainty

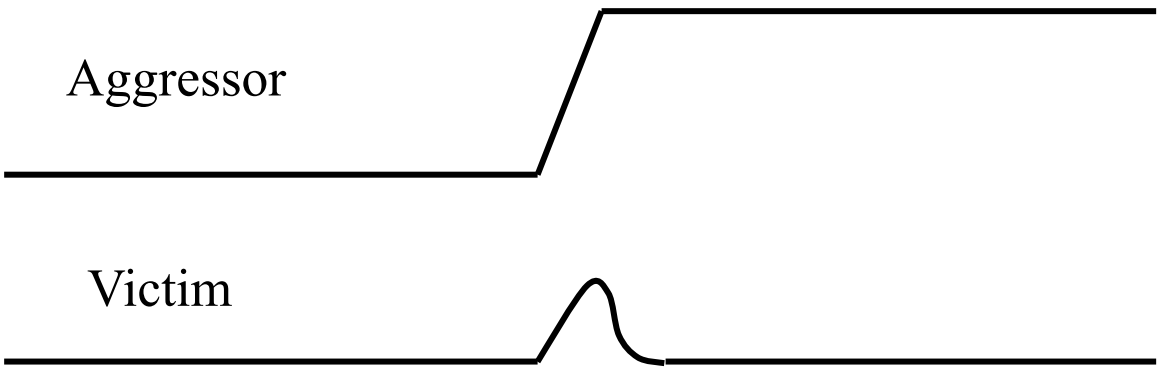




- Accurate crosstalk delay estimation is vector dependent and (almost) impossible →pessimism and simplifications
- Relatively greater coupling noise due to line dimension scaling
- Tighter timing budgets to achieve fast circuit speed ("all paths critical")
- Guardbanding of timing analysis by MCF's (-1 to +3, etc.)

Crosstalk From Capacitive Coupling

- Glitches caused by capacitive coupling between wires
  - An "aggressor" wire switches
  - A "victim" wire is charged or discharged by the coupling capacitance
- An otherwise quiet victim may look like it has temporarily switched
- This is bad if:
  - The victim is a clock or asynchronous reset
  - The victim is a signal whose value is being latched at that moment



Slide courtesy of Paul Rodman, ReShape