CS184a: Computer Architecture (Structures and Organization)

Day15: November 13, 2000

Retiming

Previously

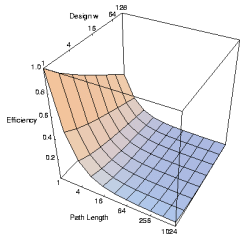
- Reviewed Pipelining
  - basic assignments on
- Saw spatial designs efficient
  - when reuse logic at maximum frequency
- Interconnect dominant delay
  - and dominant area
  - heavy call to reuse to use efficiently

Today

- Systematic transformation for retiming
  - maximize throughput
  - preserve semantics
  - "justify" mandatory registers in design

Motivation

- FPGAs (spatial computing)
  - run efficiently when all resources reused rapidly
    - cycle time minimized

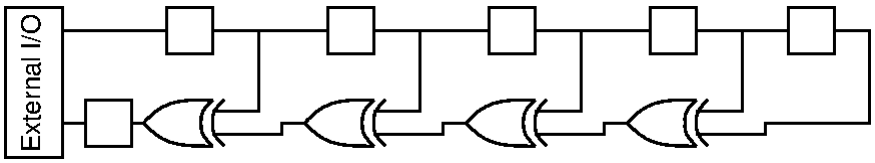


- "Everything in the right place at the right time."

Task

- Move registers to:
  - preserve semantics
  - Minimize path length between registers
  - (make path length 1 for maximum throughput or reuse)
  - Maximize reuse rate
  - …while minimizing number of registers required

Simple Example

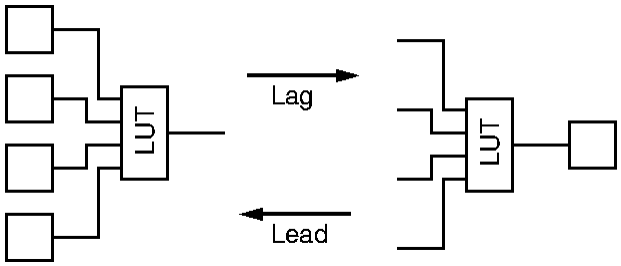


Path Length (L) = 4

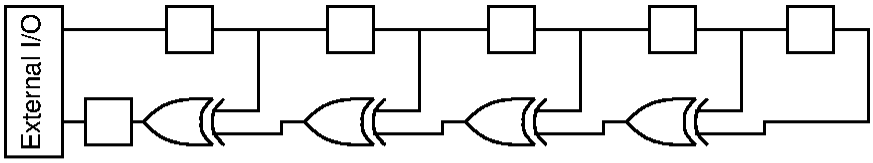
Can we do better?

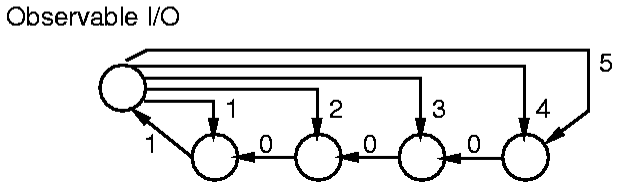
Legal Register Moves

- Retiming Lag/Lead



Canonical Graph Representation



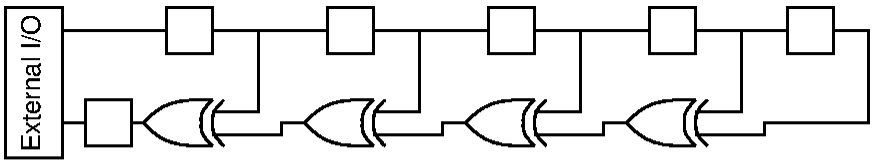


Separate arch for each path

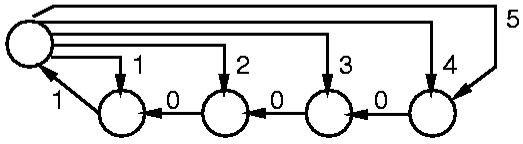
Weight edges by number of registers

(weight nodes by delay through node)

Critical Path Length



Observable I/O

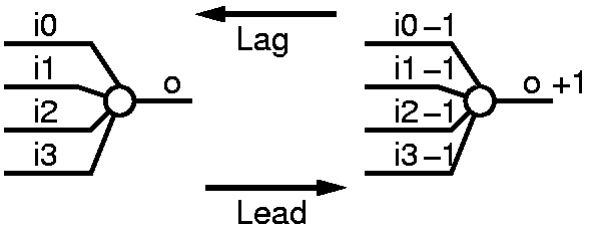


Critical Path: Length of longest path of zero weight nodes

Compute in O(|E|) time by levelizing network:

Topological sort, push path lengths forward until find register.

Retiming Lag/Lead



Retiming: Assign a lag to every vertex

weight(e′) = weight(e) + lag(head(e))-lag(tail(e))

Valid Retiming

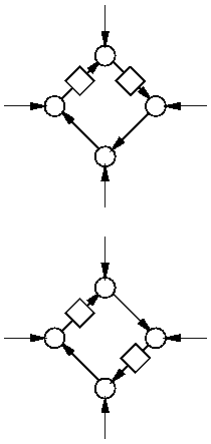
- Retiming is valid as long as:
  - $\forall e$ in graph
    - $weight(e') = weight(e) + lag(head(e)) - lag(tail(e)) \geq 0$
- Assuming original circuit was a valid synchronous circuit, this guarantees:
  - non-negative register weights on all edges
    - no travel backward in time :-)
  - all cycles have strictly positive register counts
  - propagation delay on each vertex is non-negative (assumed 1 for today)

Retiming Task

- Move registers ≡ assign lags to nodes
  - lags define all locally legal moves
- Preserving non-negative edge weights
  - (previous slide)
  - guarantees collection of lags remains consistent globally

Retiming Transformation

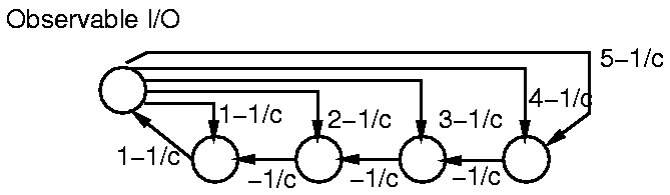
- N.B. -- unchanged by retiming
  - number of registers around a cycle
  - delay along a cycle
- Cycle of length *P* must have
  - at least *P/c* registers on it
  - to be retimeable to cycle *c*



Optimal Retiming

- There is a retiming of
  - graph *G*
  - w/ clock cycle *c*
  - *iff* *G*-1/*c* has no cycles with negative edge weights
- *G*-α ≡ subtract α from each edge weight

G-1/*c*



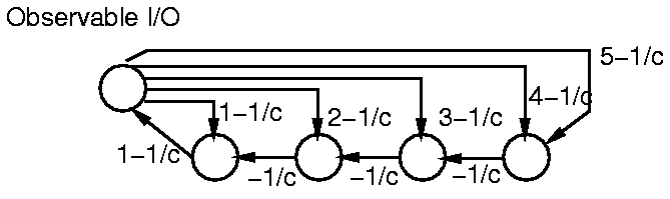
Compute Retiming

- Lag(v) = shortest path to I/O in *G*-1/*c*
- Compute shortest paths in O(|V||E|)
  - Bellman-Ford
  - also use to detect negative weight cycles when *c* too small

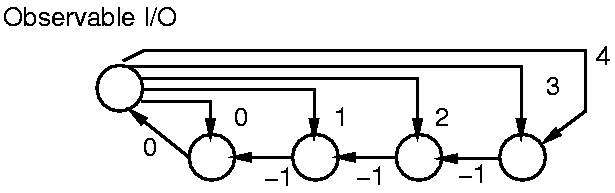
Bellman Ford

- For I←0 to N
  - $u_i \leftarrow \infty$ (except u_i=0 for IO)
- For k←0 to N
  - for $e_{i,j} \in E$
    - $u_i \leftarrow \min(u_i, u_j + w(e_{i,j}))$
- for $e_{i,j} \in E$
    - if $u_i > u_j + w(e_{i,j})$
      - cycles detected

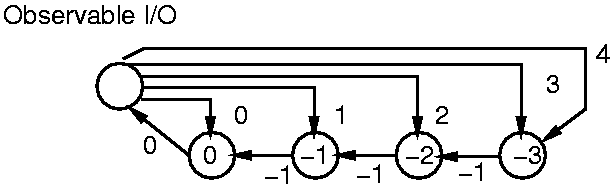
Apply to Example



Apply: Find Lags

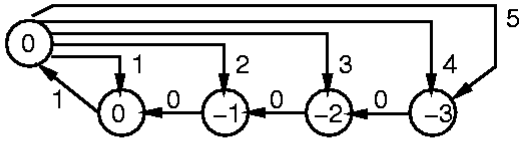


Apply: Lags

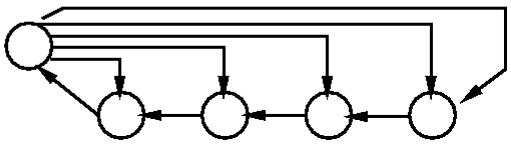


Apply: Move Registers

Observable I/O



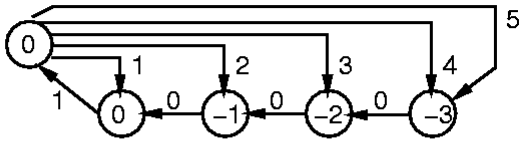
Observable I/O



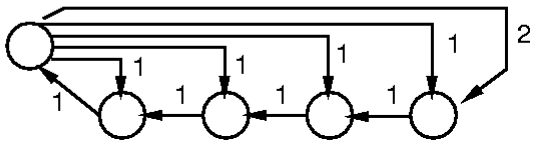
weight(e′) = weight(e) + lag(head(e))-lag(tail(e))

Apply: Retimed

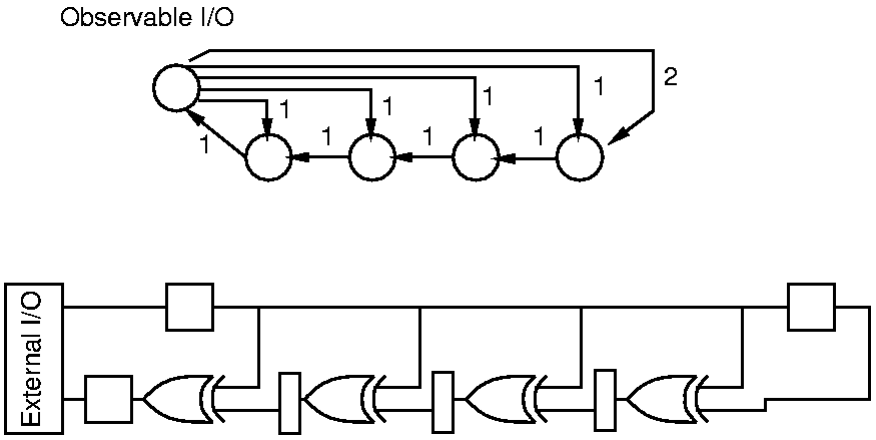
Observable I/O



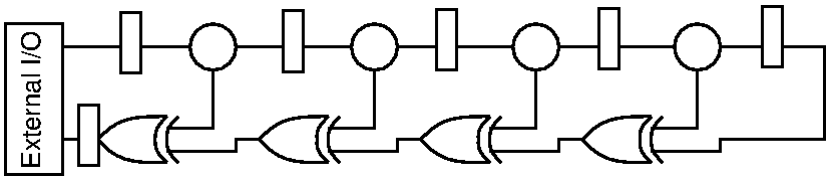
Observable I/O



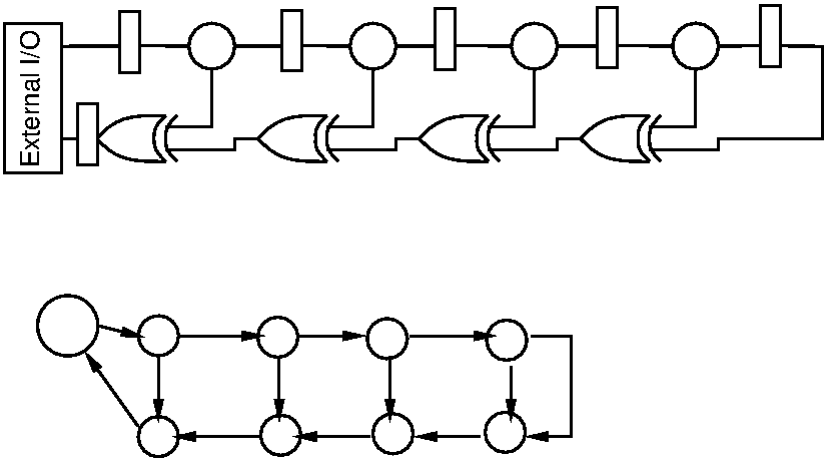
Apply: Retimed Design



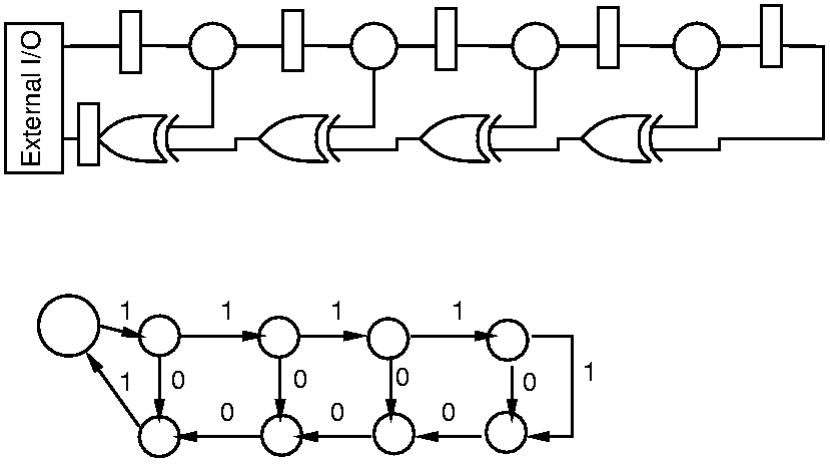
Revise Example (fanout delay)



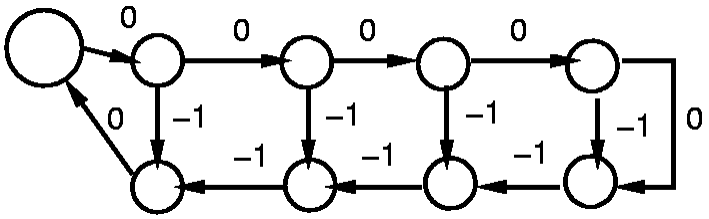
Revised: Graph



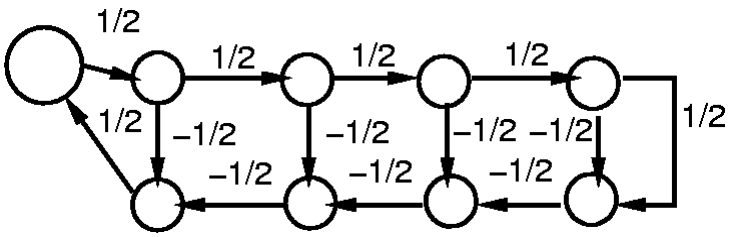
Revised: Graph



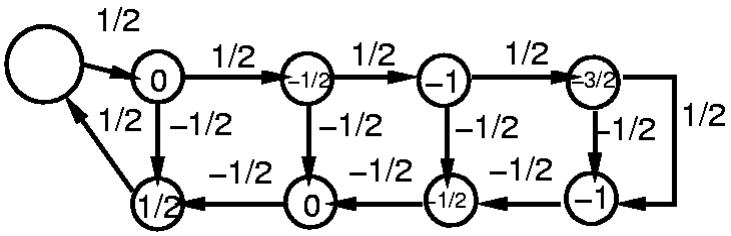
Revised: C=1?



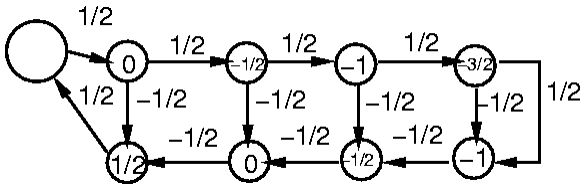
Revised: C=2?



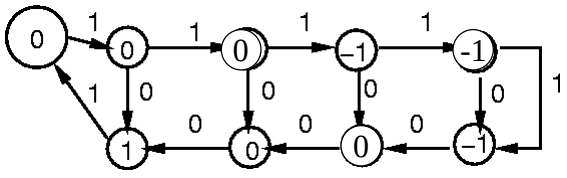
Revised: Lag



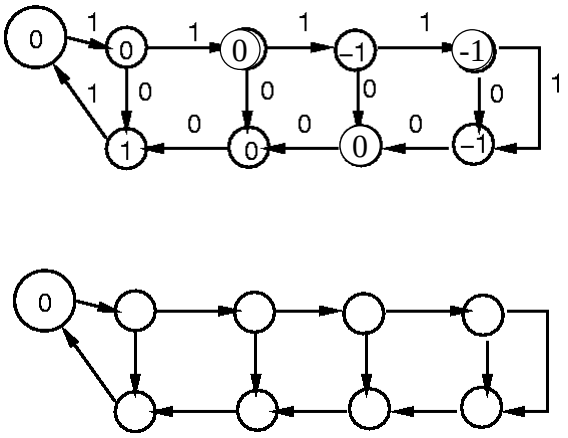
Revised: Lag



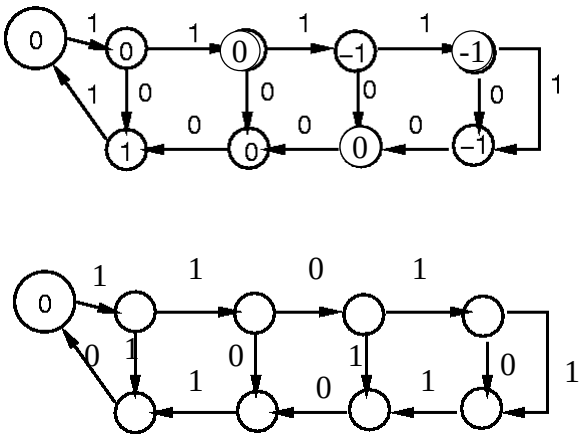
Take ceiling to convert to integer lags:



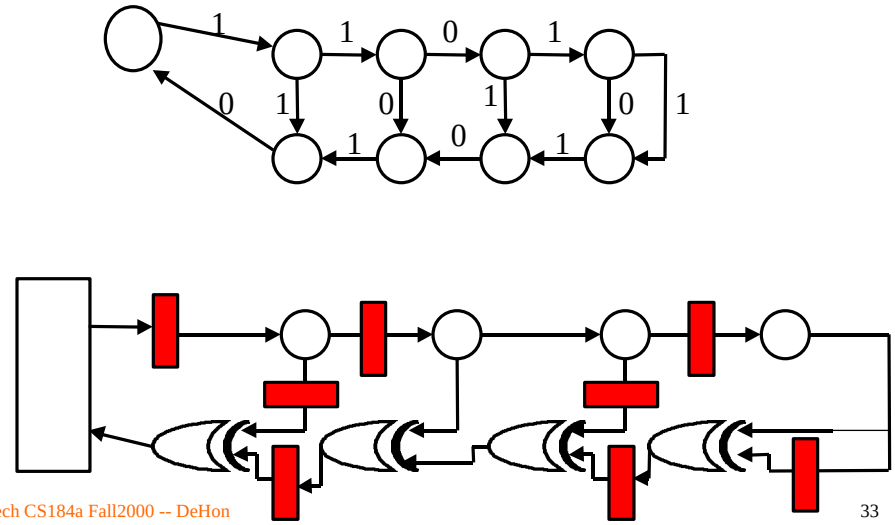
Revised: Apply Lag



Revised: Apply Lag



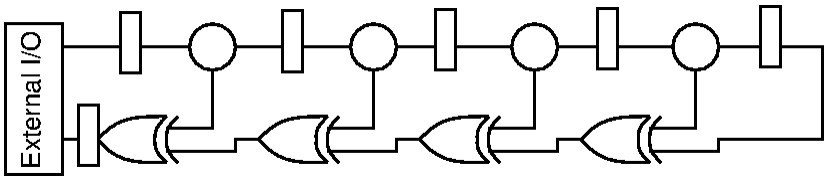
Revised: Retimed



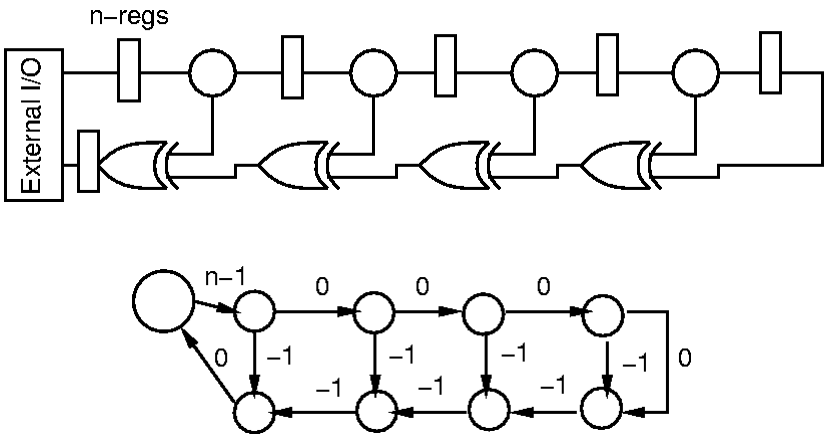
Pipelining

- Can use this retiming to pipeline
- Assume have enough (infinite supply) of registers at edge of circuit
- Retime them into circuit

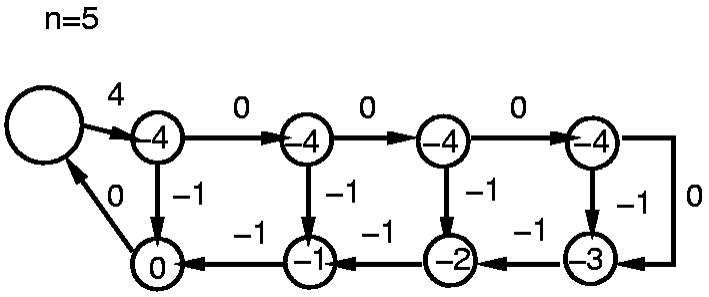
C>1 ==> Pipeline



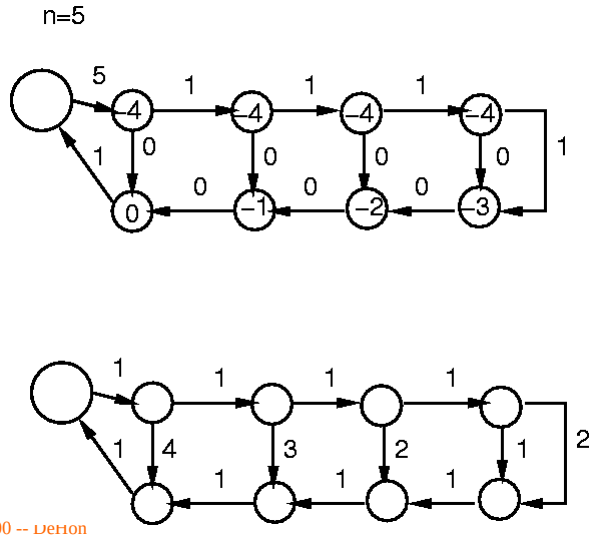
Add Registers



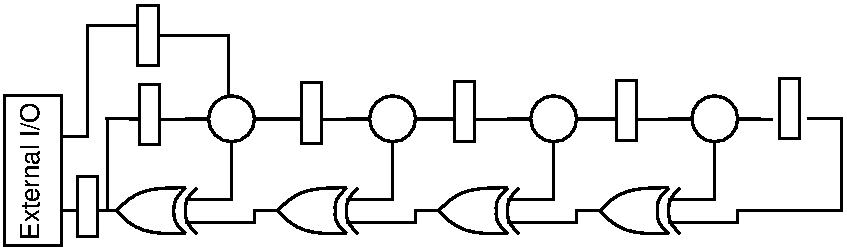
Pipeline Retiming: Lag



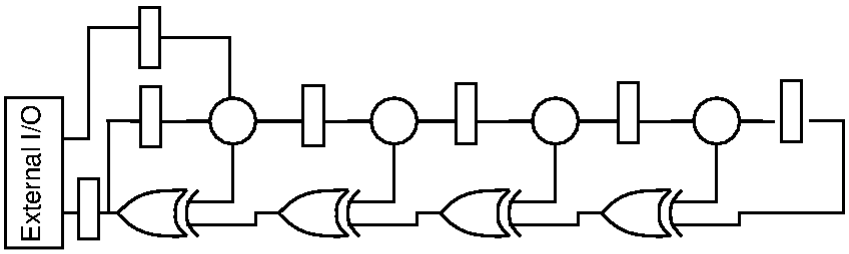
Pipelined Retimed

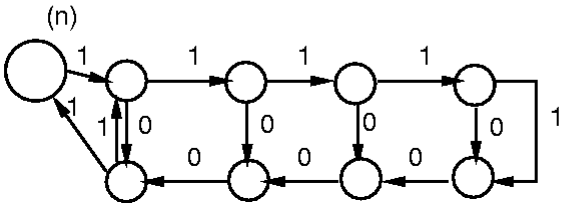


Real Cycle

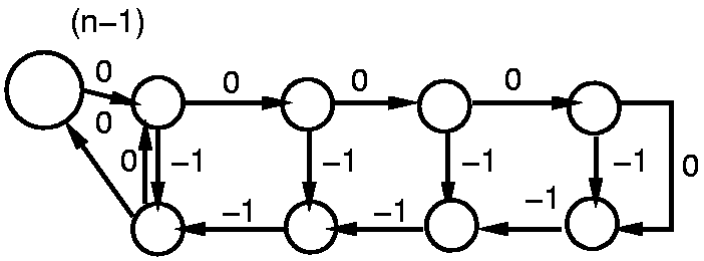


Real Cycle

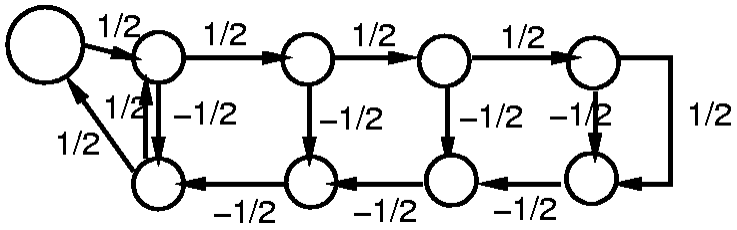




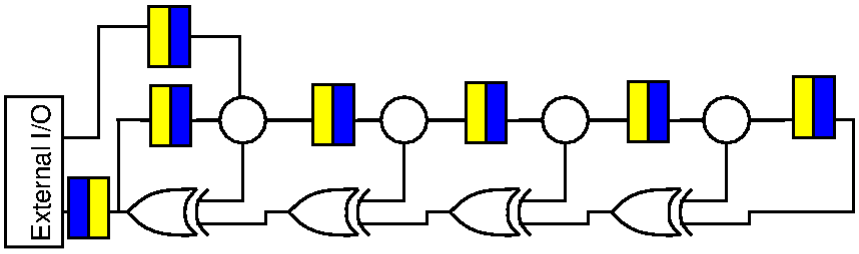
Cycle C=1?



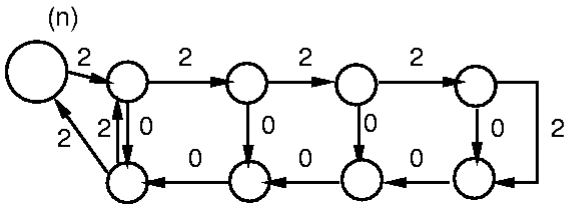
Cycle C=2?



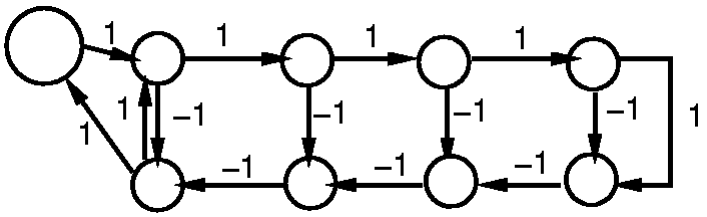
Cycle: C-slow



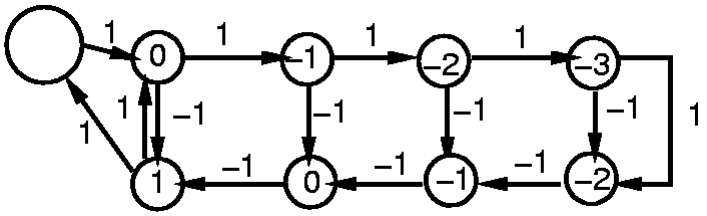
Cycle=c ⇒ C-slow network has Cycle=1



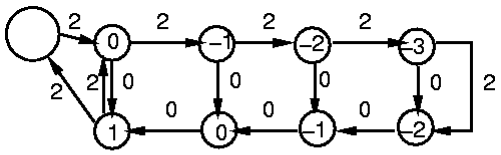
2-slow Cycle ⇒ C=1

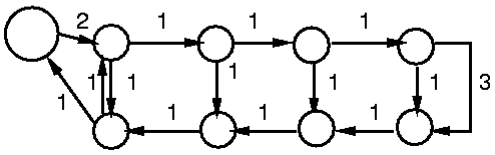


2-Slow Lags

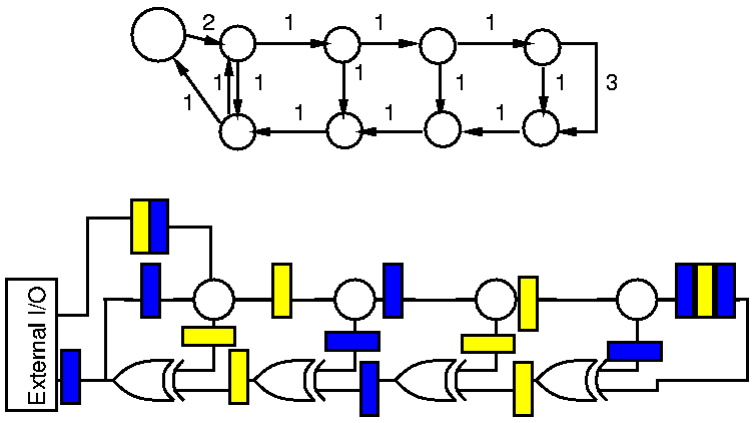


2-Slow Retime





Retimed 2-Slow Cycle

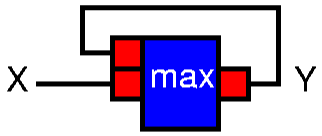


C-Slow applicable?

- Available parallelism
  - solve C identical, independent problems
    - e.g. process packets (blocks) separately
    - e.g. independent regions in images
- Commutative operators
  - e.g. max example

Max Example

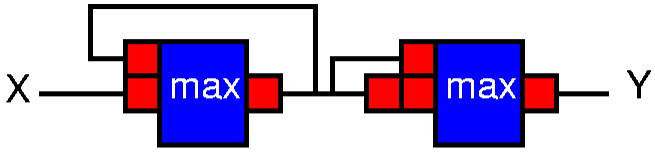
2–Slow design:



X2 X2 X1 X1 X0 X0 → Y2 ? Y1 ? Y0 ?

B2 A2 B1 A1 B0 A0 → YA2 YB1 YA1 YB0 YA0 ?

Max Example



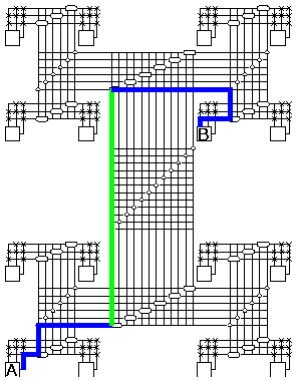
Computes two interleaved streams: even max, odd max

Computes final max of even and odd pairs

Monday Lecture
Stopped Here

HSRA Retiming

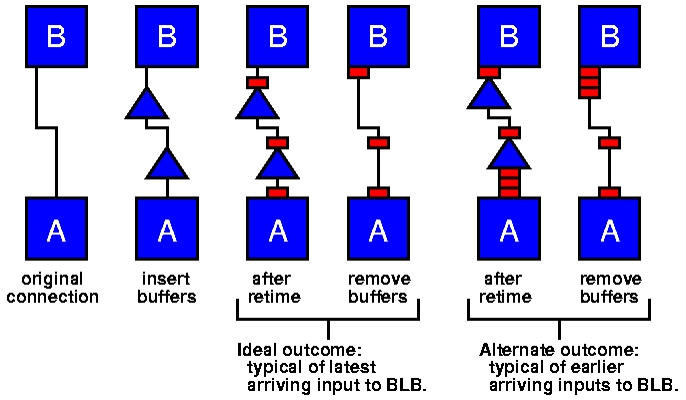
- HSRA
  - adds mandatory pipelining to interconnect
- One additional twist
  - long, pipelined interconnect
    - ⇒ need more than one register on paths



Accommodating HSRA Interconnect Delays

- Add buffers to LUT→LUT path to match interconnect register requirements
- Retime to C=1 as before
- Buffer chains force enough registers to cover interconnect delays

Accommodating HSRA Interconnect Delays



Big Ideas
[MSB Ideas]

- Retiming important to
  - minimize cycles
  - efficiently utilize spatial architectures
- Optimally solvable in O(|V||E|) time
- Tells us
  - pipelining required
  - C-slow
  - where to move registers