CS184a: Computer Architecture (Structure and Organization)

Day 19: February 23, 2005
Retime 1: Transformations

Previously

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

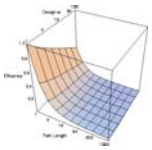
Today

- Systematic transformation for retiming
  - preserve semantics (meaning)

Motivation

Motivation

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



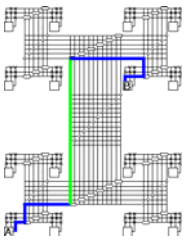
- "Everything in the right place at the right time."

Motivating Questions

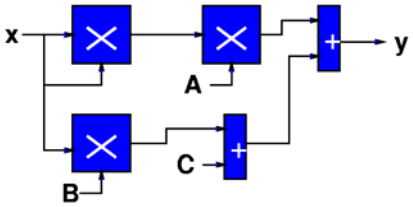
- Can I build a fixed-frequency (fixed clock) programmable architecture?
- Can I always make a design run at maximum clock rate?
- How do we systematically transform any computation to
  - Operate on fixed-frequency array?
  - Coordinate around mandatory registers in design?

Interconnect Retiming

- Long Paths Slow
- Could limit cycle
- Add registers to long distance interconnect
  - At each switch?
  - In the middle of long wires?
- How justify these registers?

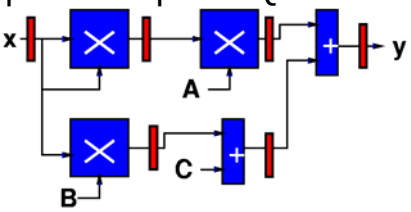


Spatial Quadratic

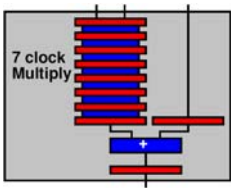


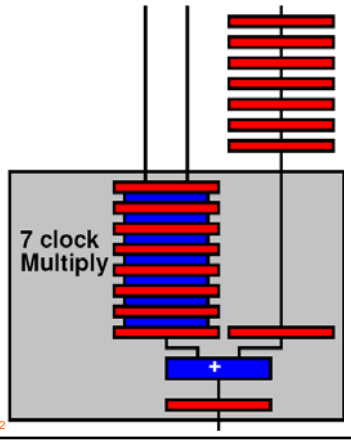
- How do we pipeline a design?

Pipelined Spatial Quadratic



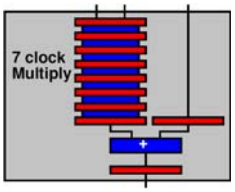
How do you use?





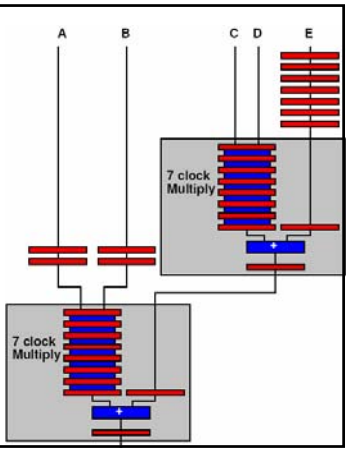
How do you use?

- To compute A*B+C*D+E



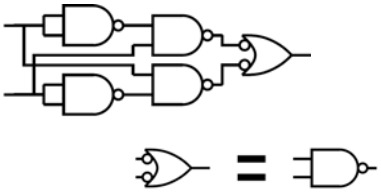
Compute

- A*B+C*D+E

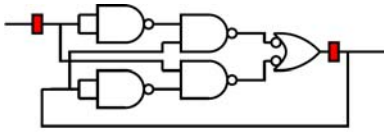


How Compute?

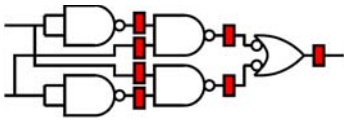
- $Y_i=Y_{i-1}$ xor X_i
- With pipelined nand2 gates?

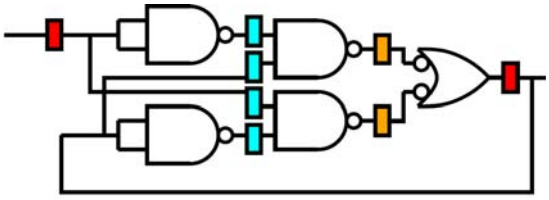


want



have



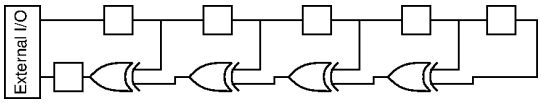


Retiming Algorithm

Task

- Move registers to:
  - Preserve semantics
  - Minimize path length between registers
  - *i.e.* Make path length 1 for maximum throughput or reuse
  - …while minimizing number of registers required

Simple Example

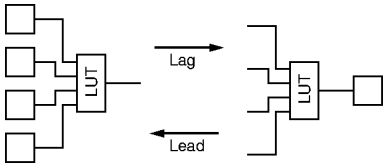


Path Length (L) = 4

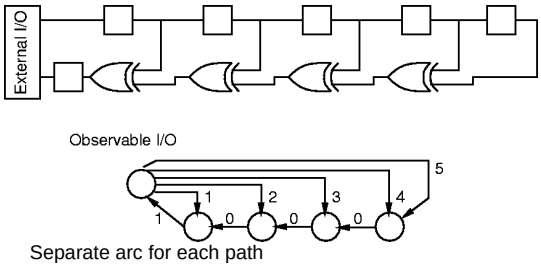
Can we do better?

Legal Register Moves

- Retiming Lag/Lead

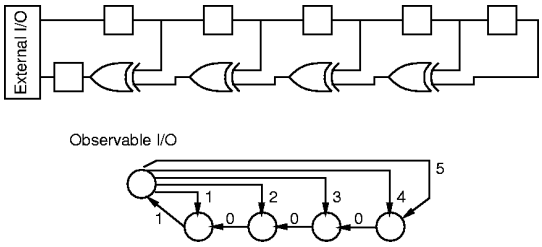


Canonical Graph Representation



Separate arc for each path
Weight edges by number of registers
(weight nodes by delay through node)

Critical Path Length

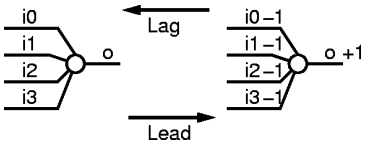


Critical Path: Length of longest path of zero weight nodes

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

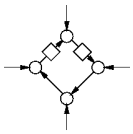
- Retiming is valid as long as:
  - ∀e in graph
    - weight(e') = weight(e) + lag(head(e))-lag(tail(e)) ≥ 0
- Assuming original circuit was a valid synchronous circuit, this guarantees:
  - non-negative register weights on all edges
    - no travel backward in time :-)
  - all cycles have strictly positive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

Retiming Task

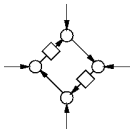
- Move registers ≡ assign lags to nodes
  - lags define all locally legal moves
- Preserving non-negative edge weights
  - (previous slide)
  - guarantees collection of lags remains consistent globally

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*



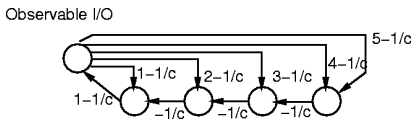
Optimal Retiming

- 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

1/c Intuition

- Want to place a register every c delay units
- Each register adds one
- Each delay subtracts 1/c
- As long as remains more positives than negatives around all cycles
  - can move registers to accommodate
  - Captures the regs=P/c constraints

G-1/*c*



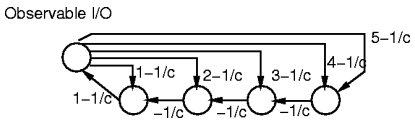
Compute Retiming

- 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

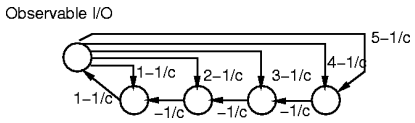
Bellman Ford

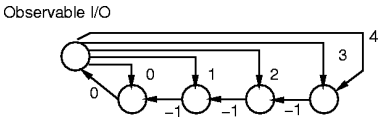
- For I←0 to N
  - $u_i \leftarrow \infty$ (except u_i=0 for IO)
- For k←0 to N
  - for $e_{i,j} \in E$
    - $u_i \leftarrow \min(u_i, u_j + w(e_{i,j}))$
- For $e_{i,j} \in E$ *//still update→negative cycle*
  - if $u_i > u_j + w(e_{i,j})$
    - cycles detected

Apply to Example

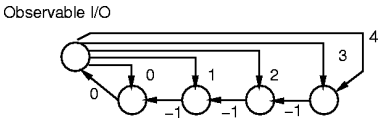


Try c=1



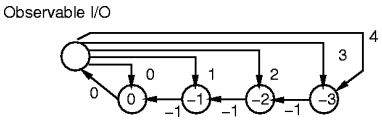


Apply: Find Lags

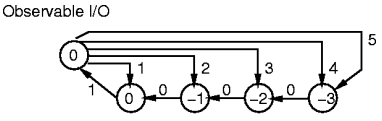


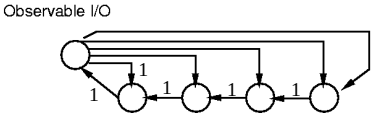
Negative weight cycles?
Shortest paths?

Apply: Lags



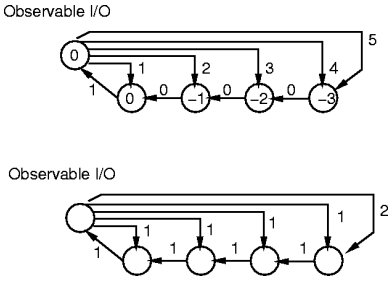
Apply: Move Registers



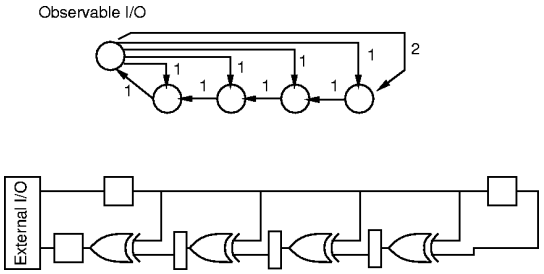


weight(e′) = weight(e) + lag(head(e))-lag(tail(e))

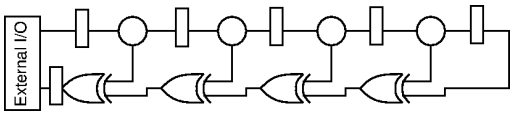
Apply: Retimed



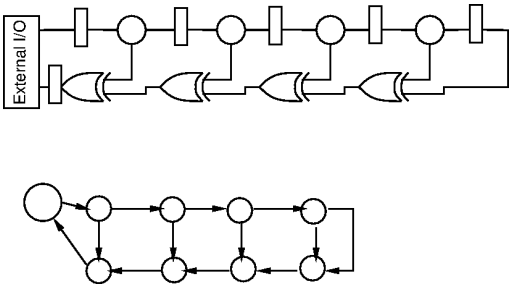
Apply: Retimed Design



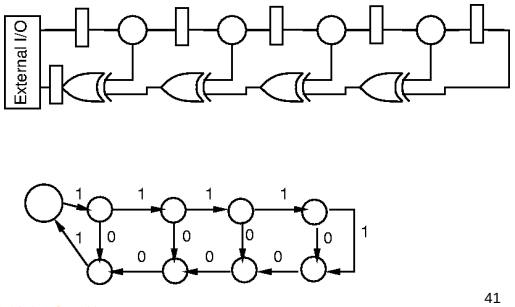
Revise Example (fanout delay)



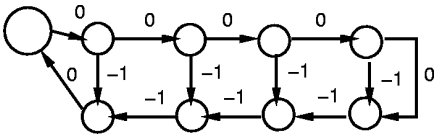
Revised: Graph



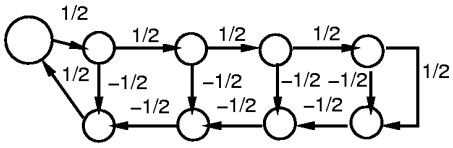
Revised: Graph



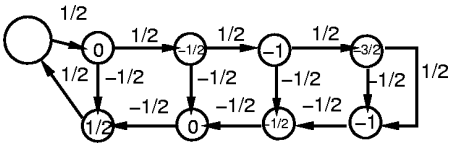
Revised: C=1?



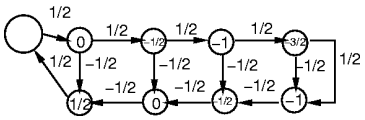
Revised: C=2?



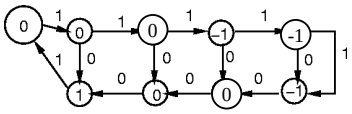
Revised: Lag



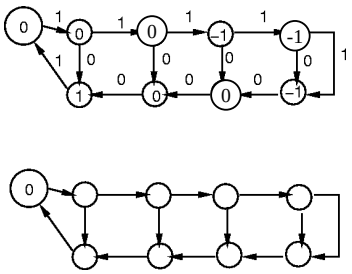
Revised: Lag



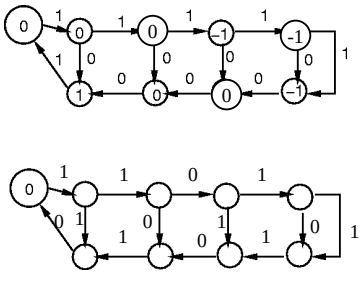
Take ceiling to convert to integer lags:



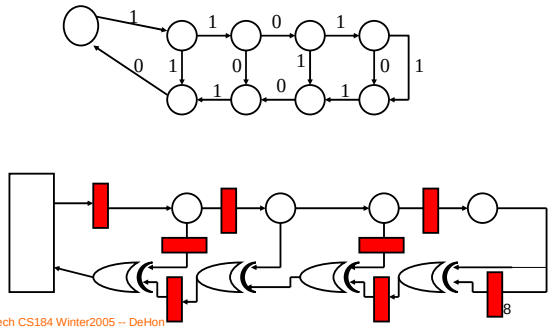
Revised: Apply Lag



Revised: Apply Lag



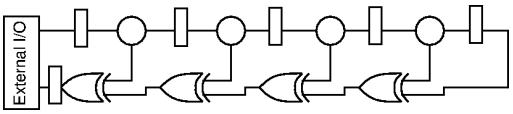
Revised: Retimed



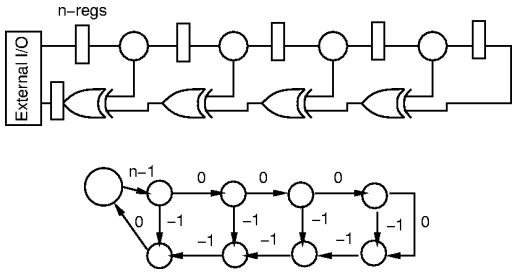
Pipelining

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

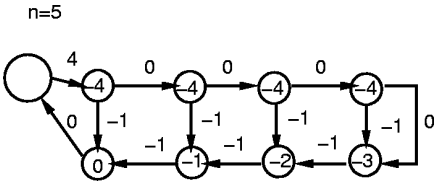
C>1 ==> Pipeline



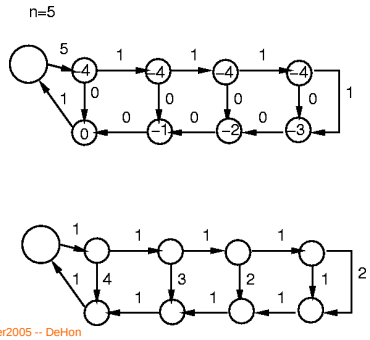
Add Registers



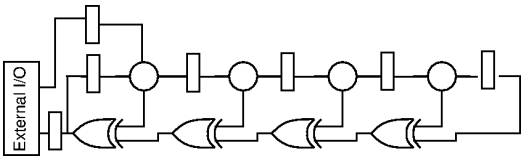
Pipeline Retiming: Lag



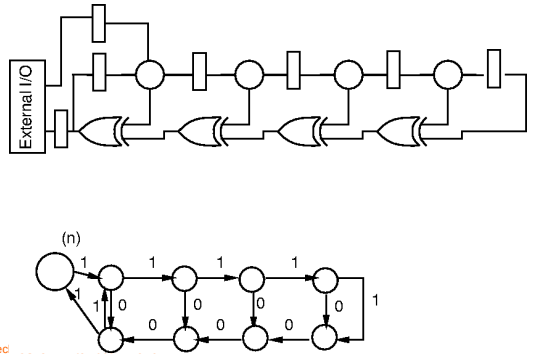
Pipelined Retimed



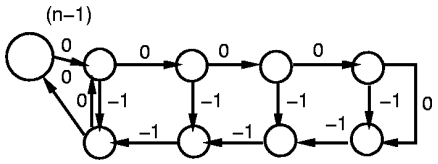
Real Cycle



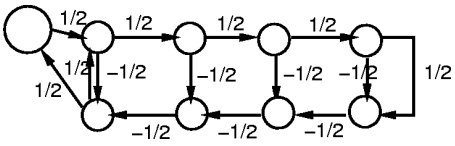
Real Cycle



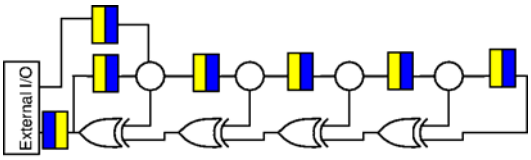
Cycle C=1?



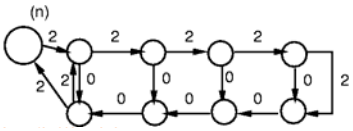
Cycle C=2?



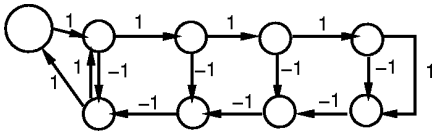
Cycle: C-slow



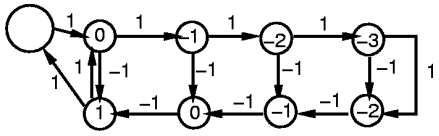
Cycle=c ⇒ C-slow network has Cycle=1



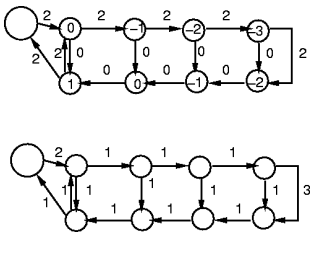
2-slow Cycle ⇒ C=1



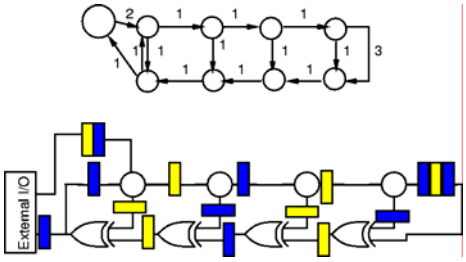
2-Slow Lags



2-Slow Retime



Retimed 2-Slow Cycle

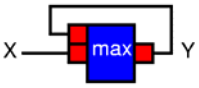


C-Slow applicable?

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

Max Example

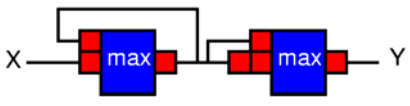
2–Slow design:



X2 X2 X1 X1 X0 X0 → Y2 ? Y1 ? Y0 ?

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

Max Example

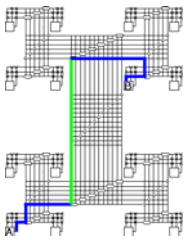


Computes two interleaved streams: even max, odd max

Computes final max of even and odd pairs

HSRA Retiming

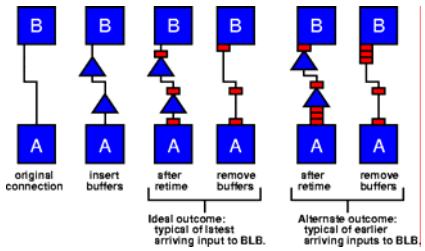
- HSRA
  - adds mandatory pipelining to interconnect
- One additional twist
  - long, pipelined interconnect
    - ⇒ need more than one register on paths



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 transformations 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
- Can accommodate mandatory delays