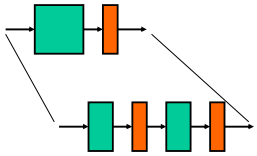
Pipelining

- Adding registers along a path
  - split combinational logic into multiple cycles
  - each cycle smaller than previously
  - increase throughput



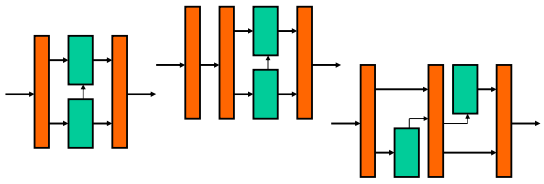
Pipelining

- Delay, d, of slowest combinational stage determines performance
- Throughput = 1/d : rate at which outputs are produced
- Latency = n·d : number of stages * clock period
- Pipelining increases circuit utilization
- Registers slow down data, synchronize data paths

- Wave-pipelining
  - no pipeline registers - waves of data flow through circuit
  - relies on equal-delay circuit paths - no short paths

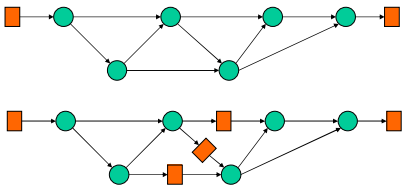
When and How to Pipeline?

- Where is the best place to add registers?
  - splitting combinational logic
  - overhead of registers (propagation delay and setup time requirements)
- What about cycles in data path?
- Example: 16-bit adder, add 8-bits in each of two cycles



Retiming

- Process of optimally distributing registers throughout a circuit
  - minimize the clock period
  - minimize the number of registers

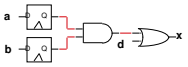


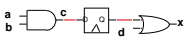
Retiming (cont'd)

- Fast optimal algorithm (Leiserson & Saxe 1983)
- Retiming rules:
  - remove one register from each input and add one to each output
  - remove one register from each output and add one to each input

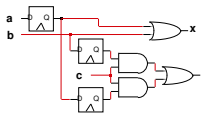
Retiming examples

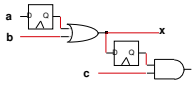
- Shortening critical paths





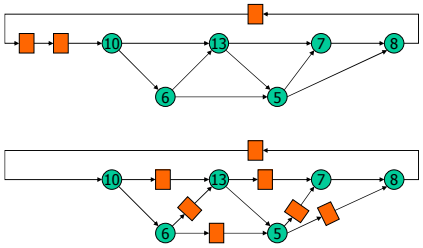
- Create simplification opportunities





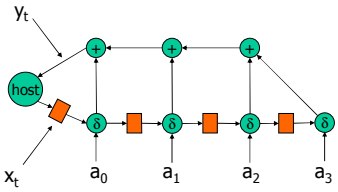
Optimal Pipelining

- Add registers - use retiming to find optimal location



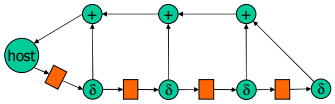
Example - Digital Correlator

- $y_t = \delta(x_t, a_0) + \delta(x_{t-1}, a_1) + \delta(x_{t-2}, a_2) + \delta(x_{t-3}, a_3)$
- $\delta(x_t, a_0) = 0$ if $x \neq a$, 1 otherwise (and passes x along to the right)

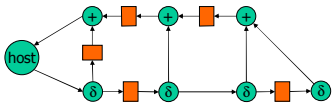


Example - Digital Correlator (cont'd)

- Delays: adder, 7; comparator, 3; host, 0



cycle time = 24



cycle time = 13

Retiming Algorithm

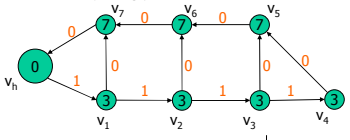
- Representation of circuit as directed graph
  - nodes: combinational logic
  - edges: connections between logic that may or may not include registers
  - weights: propagation delay for nodes, number of registers for edges
  - path delay (D): sum of propagation dealys along path nodes
  - path weight (W): sum of edge weights along path
    - always > 0, no asynchronous feedback
- Problem statement
  - given: cycle time, T, and a circuit graph
  - adjust edge weights (number of registers) so that all path delays < T, unless their path weight ≥ 1, and the outputs to the host are the same (in both function and delay) as in the original graph

Retiming Algorithm Approach

- Compute path weights and delays between each pair of nodes
  - W and D matrices
- Choose a cycle time T
- Determine if it is possible to assign new weights so that all paths with delays greater than T have a weight that is 1 or greater (use linear programming)
- Choose a smaller cycle time and repeat until the smallest T is found

Computing W and D

- W matrix: number of registers on path from $u \rightarrow v$
- D matrix: total delay along path from $u \rightarrow v$



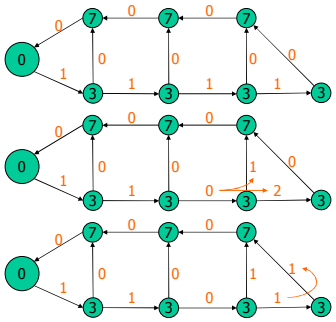
W	h	1	2	3	4	5	6	7
h	0	1	2	3	4	3	2	1
1	0	0	1	2	3	2	1	0
2	0	1	0	1	2	1	0	0
3	0	1	2	0	1	0	0	0
4	0	1	2	3	0	0	0	0
5	0	1	2	3	4	0	0	0
6	0	1	2	3	4	3	0	0
7	0	1	2	3	4	3	2	0

D	h	1	2	3	4	5	6	7
h	0	3	6	9	12	16	13	10
1	10	3	6	9	12	16	13	10
2	17	20	3	6	9	13	10	17
3	24	27	30	3	6	10	17	24
4	24	27	30	33	3	10	17	24
5	21	24	27	30	33	7	14	21
6	14	17	20	23	26	30	7	14
7	7	10	13	16	19	23	20	7

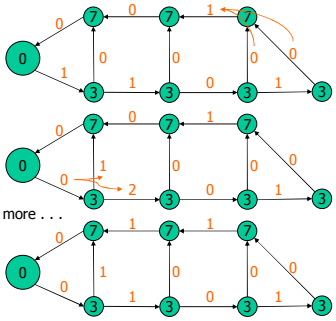
Computing W and D

- W[u,v] = number of registers on the minimum weight path from u → v
  - Any retiming changes the weight of all paths by the same constant
  - i.e. Retiming cannot change which is the minimum weight path
- D[u,v] = maximum delay over all paths with W[u,v] registers
  - Retiming does not affect D[u,v]
- These matrices contain all the required register and delay information
  - If retiming removes all registers from the path u → v, then D[u,v] is the largest delay path that results

Retiming: One Step at a Time



Retiming: One Step at a Time (cont'd)

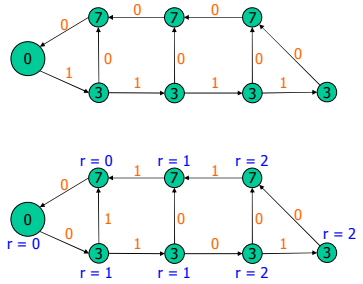


and after a few more . . .

Retiming: Problem Formulation

- r(v): number of registers pushed through a node in the forward direction
  - $w_{new}(u, v) = w_{old}(u, v) + r(u) - r(v)$
- Problem statement
  - $r(v_h) = 0$ *(host is not retimed)*
  - $w_{new}(u, v) = w_{old}(u, v) + r(u) - r(v) \geq 0$, for all u, v
    - $r(u) - r(v) \geq - w_{old}(u, v)$ *(no negative registers!)*
  - For all D[u,v] > Tclk,
    $w_{new}(u, v) = w_{old}(u, v) + r(u) - r(v) \geq 1$
    - $r(u) - r(v) \geq - w_{old}(u, v) + 1$ *(every long path has at least 1 reg)*
- Difference constraints like this can be solved by generating a graph that represents the constraints and using a shortest path algorithm like Bellman-Ford to find a set of r(v) values that meets all the constraints
- The value of r(v) returned by the algorithm can be used to generate the new positions of the registers in the retimed circuit

Retimed Correlator



Extensions to Retiming

- Host interface
  - add latency
  - multiple hosts
- Area considerations
  - limit number of registers
  - optimize logic across register boundaries
    - peripheral retiming
    - incremental retiming
    - pre-computation
- Generality
  - different propagation delays for different signals
  - widths of interconnections

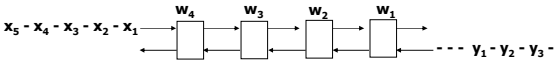
Systolic Arrays

- Set of identical processing elements
  - specialized or programmable
- Efficient nearest-neighbor interconnections (in 1-D, 2-D, other)
- SIMD-like
- Multiple data flows, converging to engage in computation

Analogy: data flowing through the system in a rhythmic fashion – from main memory through a series of processing elements and back to main memory

Example - Convolution

- $y_j = x_j w_1 + x_{j+1} w_2 + \ldots + x_{j+n-1} w_n$

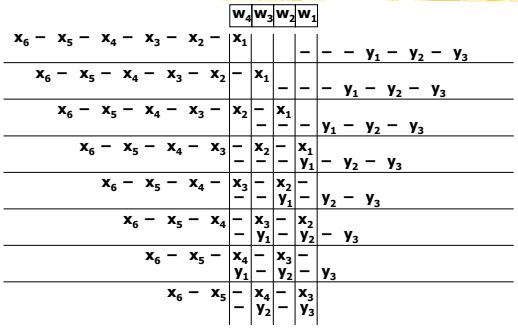


$y_1 = x_1 w_1 + x_2 w_2 + x_3 w_3 + x_4 w_4$

$y_2 = x_2 w_1 + x_3 w_2 + x_4 w_3 + x_5 w_4$

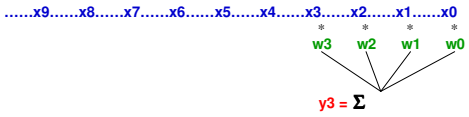
$y_3 = x_3 w_1 + x_4 w_2 + x_5 w_3 + x_6 w_4$

Example - Convolution

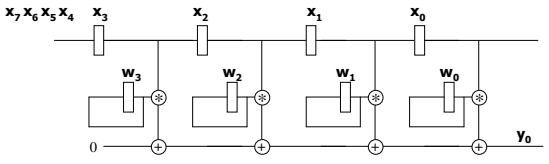


Convolution - Another Look

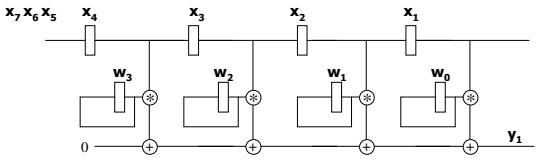
- Repeated vector product



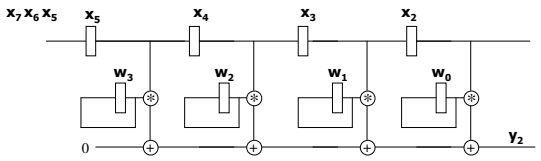
Convolution Example

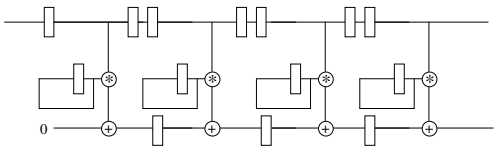


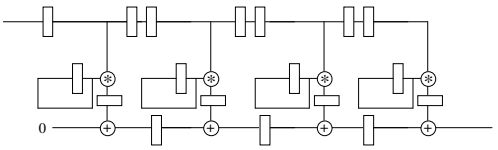
Convolution Example

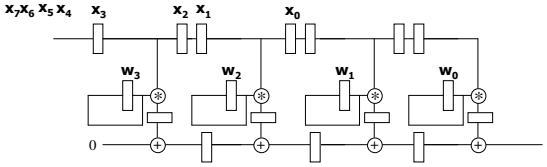


Convolution Example



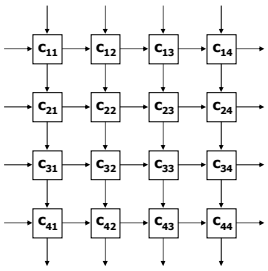




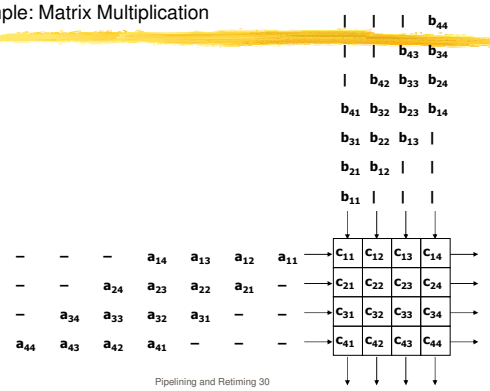


Example: Matrix Multiplication

- $C = A \times B$ $\quad c_{ij} = \Sigma_{k=1}^{n} a_{ik} b_{kj}$



Example: Matrix Multiplication



Systolic Algorithms

- 2D Convolution
  - Image processing
- FFT
- String matching
  - Dynamic programming
  - DNA comparison
- Matrix computations
  - LU decomposition
  - QR factorization

Systolic Architectures

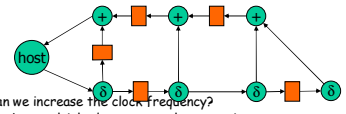
- Highly parallel
  - "fine-grained" parallelism
  - deep pipelining
- Local communication
  - wires are short - no global communication (except CLK)
    - linear array → no clock skew
  - increasingly important as wire delays increase (relative to gate delays)
- Linear arrays
  - most systolic algorithms can be done with a linear array
  - include memory in each cell in the array
  - linear array a better match to I/O limitations
- Contrast to superscalar and vector architectures

Systolic Computers

- Custom chips - early 1980's
- Warp (CMU) - 1987
  - linear array of 10 or more processing cells
  - optimized inter-cell communication for low-latency
  - pipelined cells and communication
  - conditional execution
  - compiler partitions problem into cells and generates microcode
- i-Warp (Intel) - 1990
  - successor to Warp
  - two-dimensional array
  - time-multiplexing of physical busses between cells
  - 32x32 array has 20Gflops peak performance
  - not a commercial success
- Currently confined to ASIC implementations

Digital Correlator Revisited

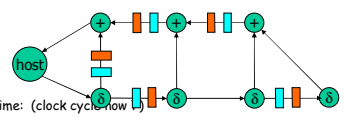
- Optimally retimed circuit (clock cycle 13)



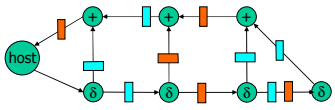
- How can we increase the clock frequency?
  - Work on multiple data sets at the same time

C-slow'ing a Circuit

- Replace every register with C registers

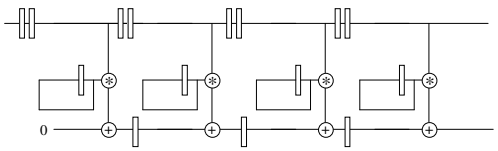


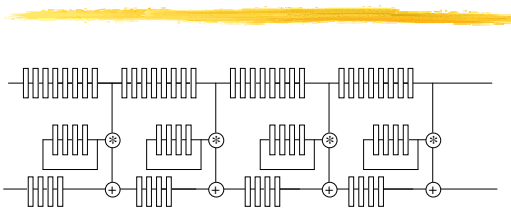
- Now retime: (clock cycle now 1)



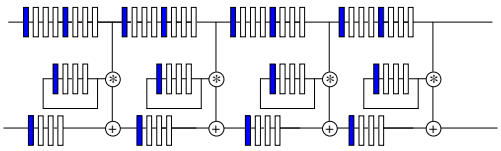
C-slowing/Retiming for Resource Sharing

- Correlator circuit

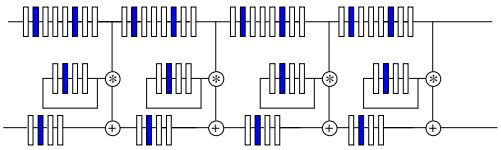




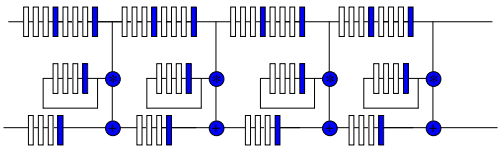
C-slowed by 4



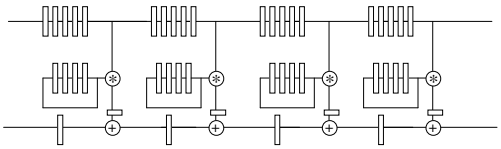
Insert Data every 4 cycles (one data set)

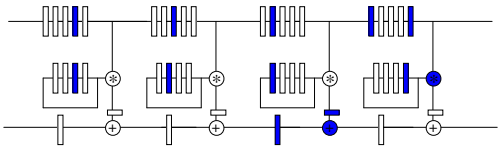


Computation Active only every 4 Cycles



Retime and remove extra Pipelining





Computation spread over time

- Only need one multiplier and one adder
- We can use this method to schedule for any number of resources