Pipelining: Some definitions

- **Latency**: Time to perform a computation
  - Data in to data out

- **Throughput**: Input or output data rate
  - Typically the clock rate

- **Combinational delays drive performance**
  - Define $d$ = delay through slowest combinational stage
    - $n$ = number of stages from input to output
  - Latency: $n \cdot d$ (in sec)
  - Throughput: $1/d$ (in Hz)
Pipelining: What and why

- **What?**
  - Subdivide combinational logic
  - Add registers between logic blocks

- **Why?**
  - Increase clock speed
    - Reduce logic delays
    - But…takes a few cycles to fill the pipe
  - Trade latency for throughput
    - Latency worse
    - Throughput better
  - Increase circuit utilization
    - Simultaneous computations
Pipelining: When and where

- **When?**
  - Throughput more important than latency
    - Signal processing
  - Logic delays >> flip-flop setup/hold times
  - No acyclic logic

- **Where?**
  - At natural breaks in the combinational logic
  - Adding registers makes sense
Pipelining example

(a) Nonpipelined version

(b) Pipelined version

Datapath for the computation of $\log(|a + b|)$
Retiming

- **Retiming**: Rearrange storage elements
  - To optimize performance
    - Minimize critical path
    - Optimize logic across register boundaries
    - Reduce register count
  - Without altering functionality

- **Pipelining adds registers**
  - To increase the clock speed

- **Retiming moves registers around**
  - Reschedules computations
Retiming in a nutshell

- Change position of FFs
  - To optimize an FSM after assignment/optimization
    - For speed
    - To suit implementation target

- Retiming modifies state assignment
  - Moving registers alters state codes
  - Preserves FSM functionality
Retiming rules

- Fast optimal algorithm
  - See Leiserson & Saxe, 1983

- 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

- Reduce register count

- Create simplification opportunities
Retiming examples (con’t)

- Move logic to suit implementation target

Original Design

Retimed Design
Optimal pipelining

✦ Add registers
  ➢ Use retiming to find optimal location
Example: Digital correlator

\[ y_t = \delta(x_t, a_0) + \delta(x_{t-1}, a_1) + \delta(x_{t-2}, a_2) + \delta(x_{t-3}, a_3) \]

- \( \delta(x, a) = 1 \) if \( x = a \); 0 otherwise
- \( \delta \) passes \( x \) to the right (unchanged)
Example: Digital correlator (cont’d)

- Delays: Comparator = 3; adder = 7

![Diagram showing an example digital correlator with two configurations]

- cycle time = 24
- cycle time = 13

Pipelining and retiming
Retiming: Step-by-step
Retiming: step-by-step (cont’d)

and after a few more . . .
Formal algorithm for retiming

- Represent circuit as a directed graph
  - Vertices $v$: Logic gates
  - Edges $e$: Connections between logic (0 or more registers)
  - Delay $d$: Delay of vertex $v$
  - Weight $w$: Number of registers on edge $e$

- Problem statement
  - Given cycle time $t$ and the circuit graph
  - Adjust weights $w$ (number of registers) so that all path delays $d < t$
    - Preserving logic functionality

- Approach
  - Generate matrices for $w$ and $d$
  - Iterate to minimize $t$ (use linear programming)
For you to think about...

- **Registers** slow the data path
  - To synchronize delays

- **Hard**: Use latches instead of registers
  - Permits faster circuits
  - Fast data slows down; slow data passes through transparent latch

- **Harder**: Self-timed datapath
  - Handshaking decides when data passes

- **Hardest**: Wave-pipelining
  - Delete the registers
    - Waves of data flow through circuit
    - Requires equal-delay circuit paths