Micro-architectures and transformations

Lecture 6

Vladimir Stojanović

6.973 Communication System Design – Spring 2006

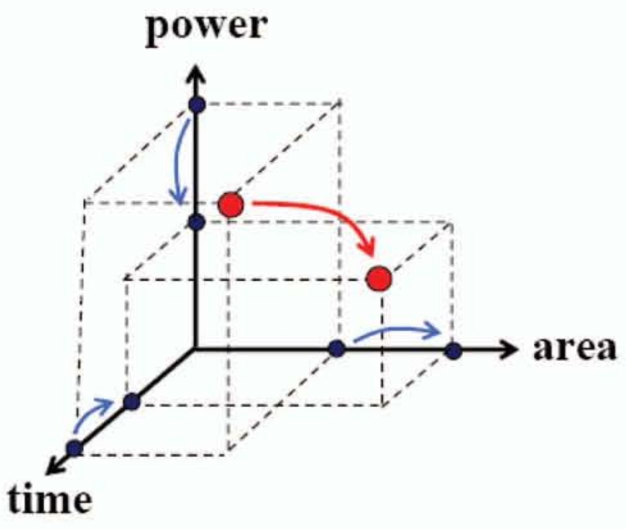
Massachusetts Institute of Technology

Behavioral transformations

- There are a large number of implementations of the same functionality
- These implementations present a different point in the area-time-power design space
- Behavioral transformations allow exploring the design space a high-level

Optimization metrics:

1. **Area** of the design
2. **Throughput** or sample time T_S
3. **Latency**: clock cycles between the input and associated output change
4. **Power** consumption
5. **Energy** of executing a task
6. …



Fixed-Coefficient Multiplication

Conventional Multiplication

$Z = X \cdot Y$

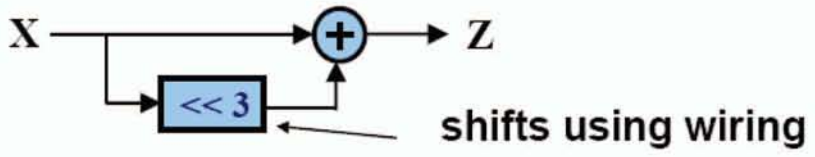
$$
\begin{array}{cccccccc}
 & & & & X_3 & X_2 & X_1 & X_0 \\
 & & & & Y_3 & Y_2 & Y_1 & Y_0 \\
\hline
 & & & & X_3 \cdot Y_0 & X_2 \cdot Y_0 & X_1 \cdot Y_0 & X_0 \cdot Y_0 \\
 & & & X_3 \cdot Y_1 & X_2 \cdot Y_1 & X_1 \cdot Y_1 & X_0 \cdot Y_1 & \\
 & & X_3 \cdot Y_2 & X_2 \cdot Y_2 & X_1 \cdot Y_2 & X_0 \cdot Y_2 & & \\
 & X_3 \cdot Y_3 & X_2 \cdot Y_3 & X_1 \cdot Y_3 & X_0 \cdot Y_3 & & & \\
\hline
Z_7 & Z_6 & Z_5 & Z_4 & Z_3 & Z_2 & Z_1 & Z_0
\end{array}
$$

Constant multiplication (become hardwired shifts and adds)

$Z = X \cdot (1001)_2$

$$
\begin{array}{cccccccc}
 & & & & X_3 & X_2 & X_1 & X_0 \\
 & & & & 1 & 0 & 0 & 1 \\
\hline
 & & & & X_3 & X_2 & X_1 & X_0 \\
 & X_3 & X_2 & X_1 & X_0 & & & \\
\hline
Z_7 & Z_6 & Z_5 & Z_4 & Z_3 & Z_2 & Z_1 & Z_0
\end{array}
$$

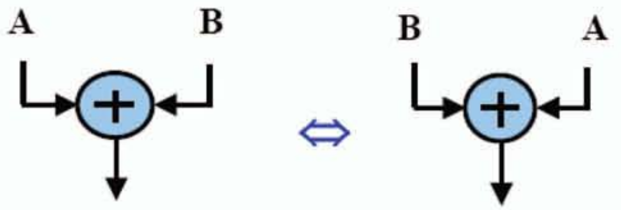
$Y = (1001)_2 = 2^3 + 2^0$



- Example - FFT

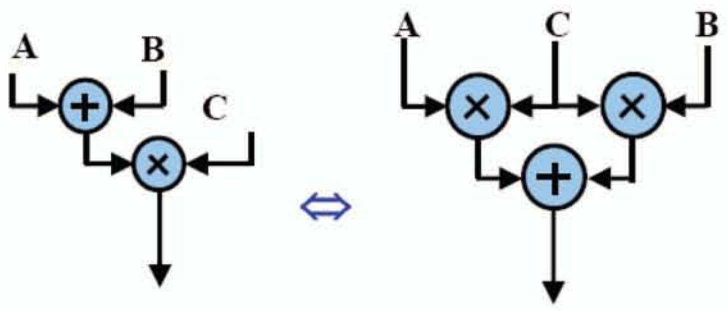
Algebraic transformations

Commutativity



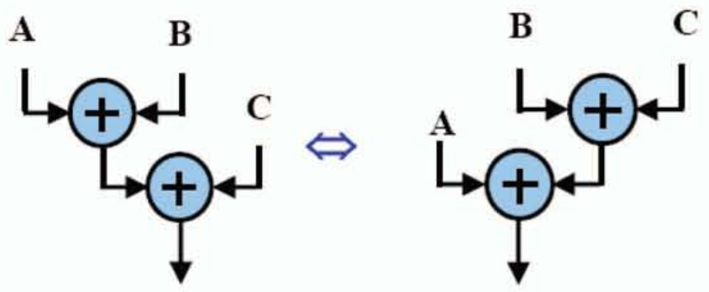
A + B = B + A

Distributivity



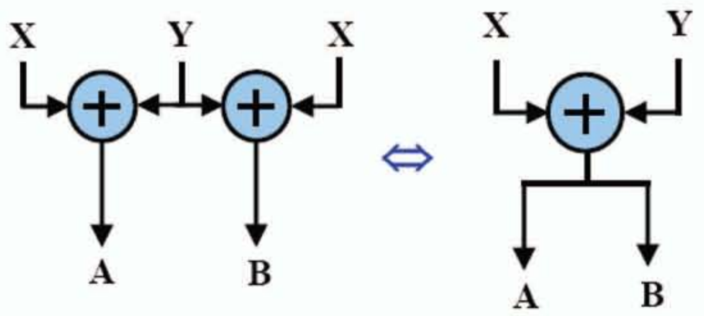
(A + B) C = AB + BC

Associativity

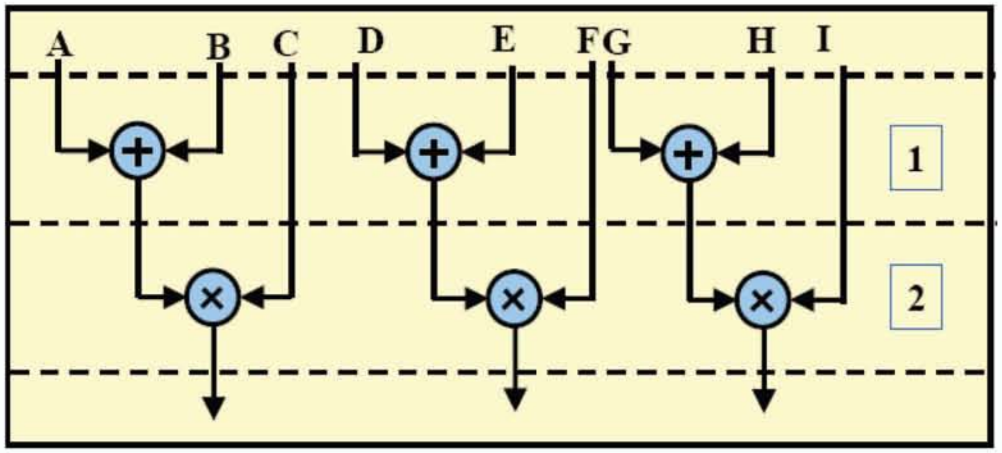


(A + B) + C = A + (B+C)

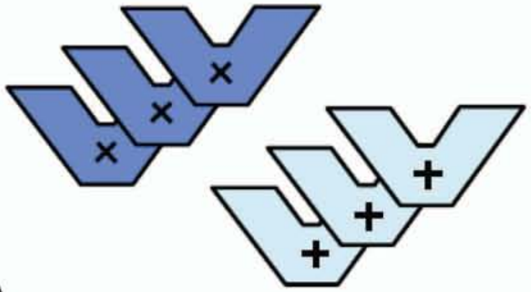
Common sub-expressions



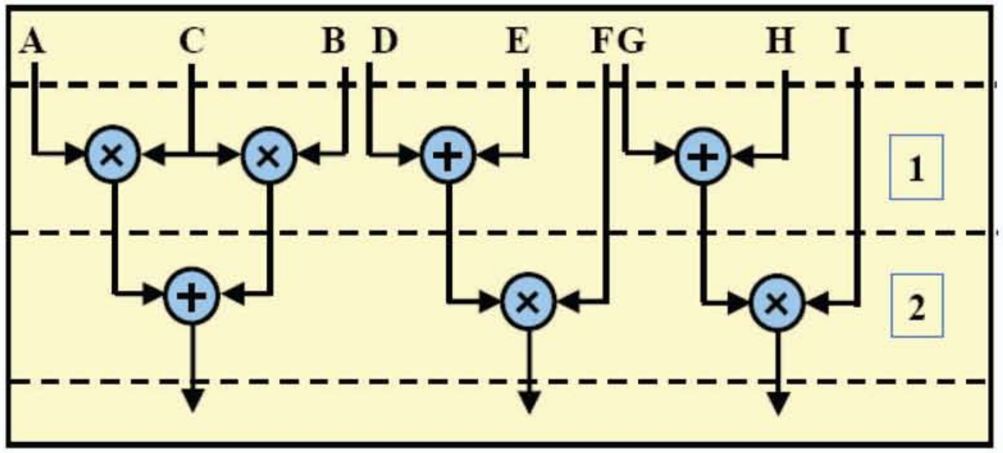
Transforms for efficient resource utilization



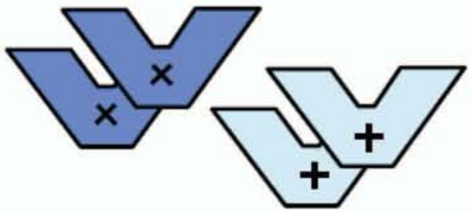
Time multiplexing: mapped to 3 multipliers and 3 adders



distributivity



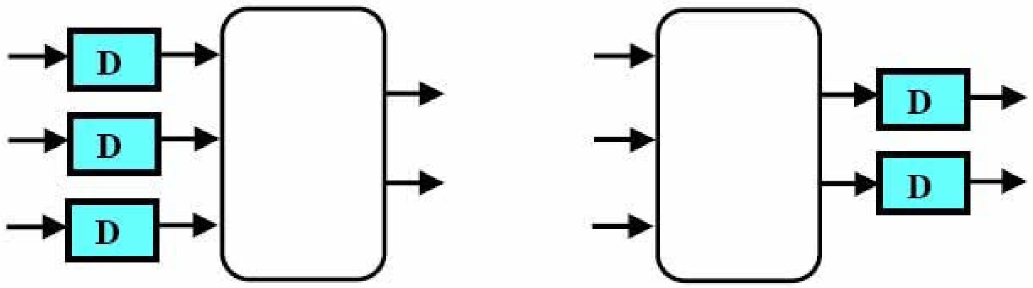
Reduce number of operators to 2 multipliers and 2 adders



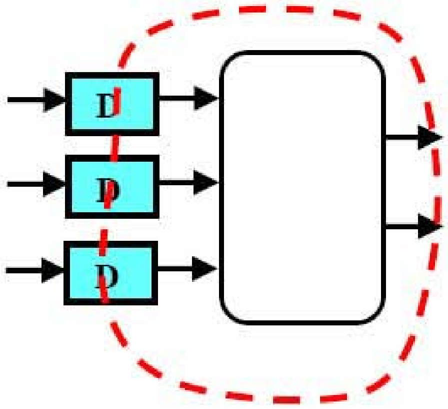
Essential transform: Retiming

Retiming is the action of moving delay around in the systems

- **Delays have to be moved from ALL inputs to ALL outputs or vice versa**



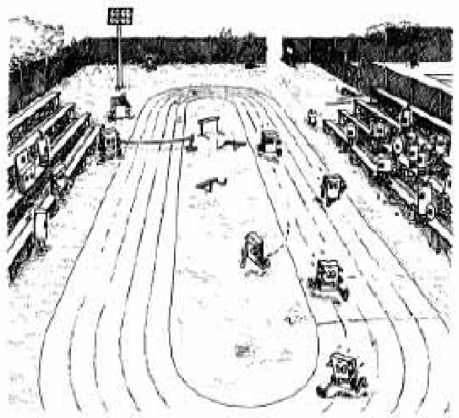
Cutset retiming: A cutset intersects the edges, such that this would result in two disjoint partitions of these edges being cut. To retime, delays are moved from the ingoing to the outgoing edges or vice versa.



Retiming Synchronous Circuitry

Charles E. Leiserson and James B. Saxe

August 20, 1990.

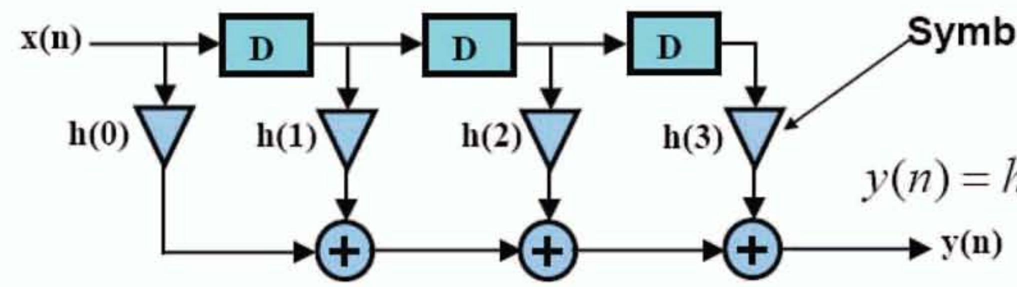


Benefits of retiming:

- Modify critical path delay
- Reduce total number of registers

(Courtesy of Prof. Charles E. Leiserson. Used with permission.)

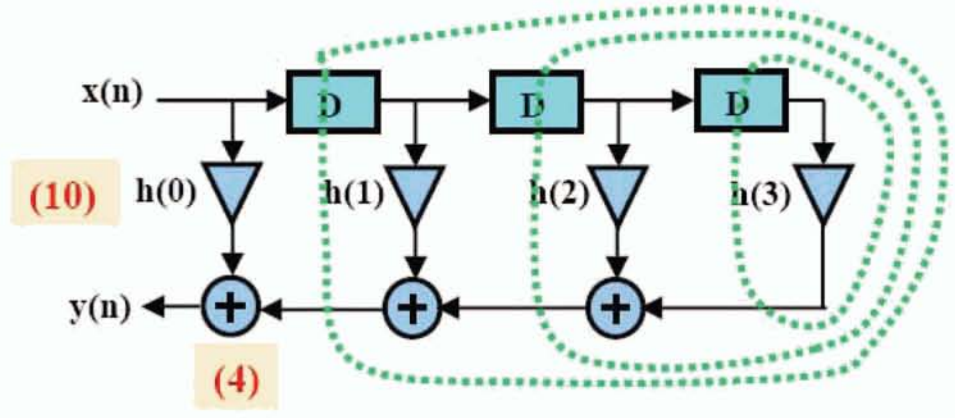
Retiming example: FIR filter



Symbol for multiplication

Direct form

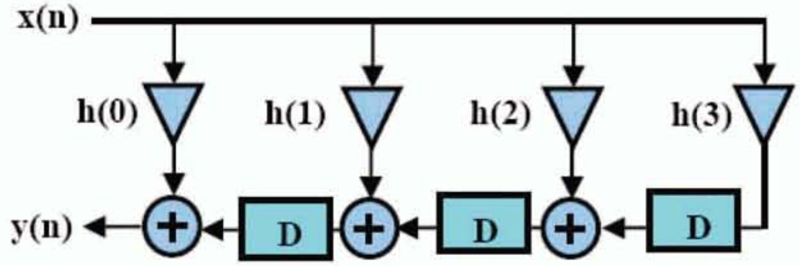
$$y(n) = h(n) \otimes x(n) = \sum_{i=0}^{K} x(n-i) \cdot h(i)$$



associativity of the addition

$T_{clk} = 22$ ns

retime

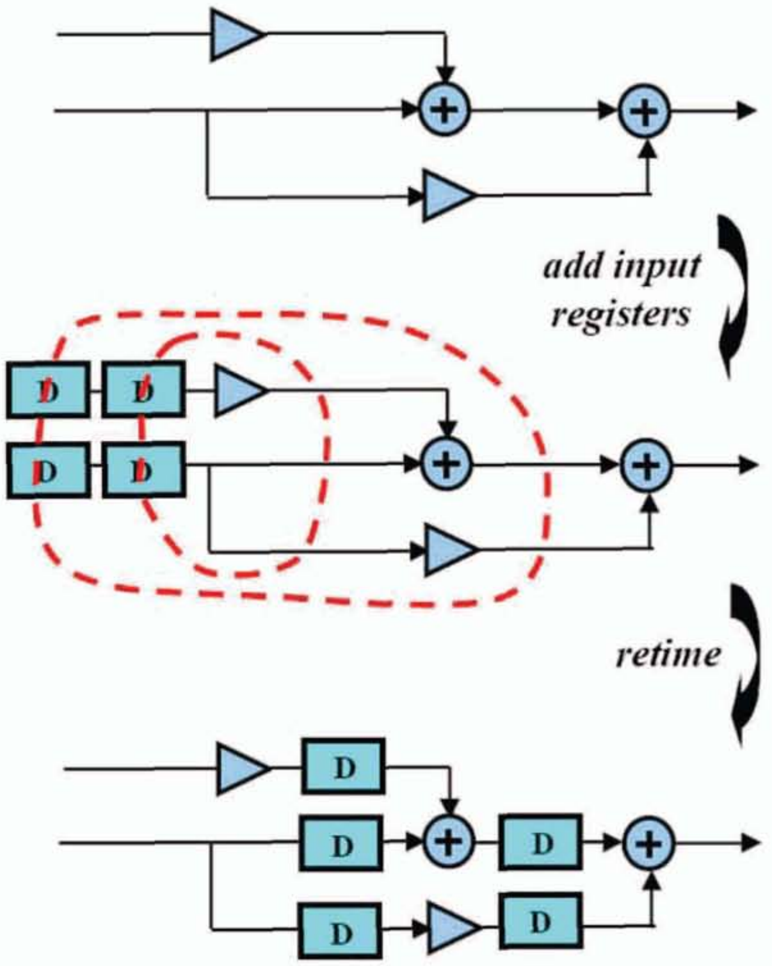


Transposed form $T_{clk} = 14$ ns

Note: here we use a first cut analysis that assumes the delay of a chain of operators is the sum of their individual delays. This is not accurate.

Pipelining

- Pipelining = Adding Delays + Retiming
  - Only works on feed-forward paths



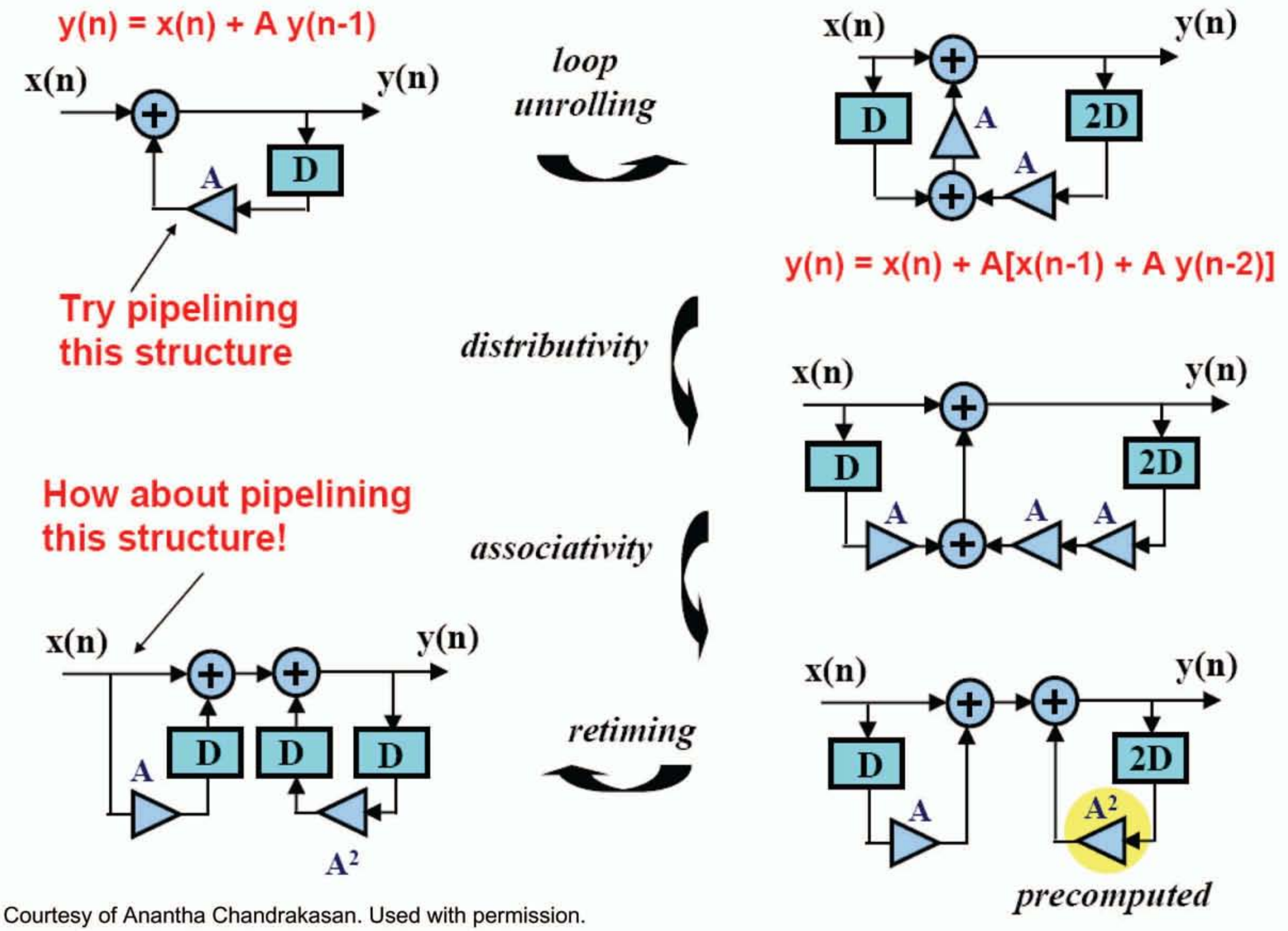
Contrary to retiming, pipelining adds extra registers to the system

How to pipeline:

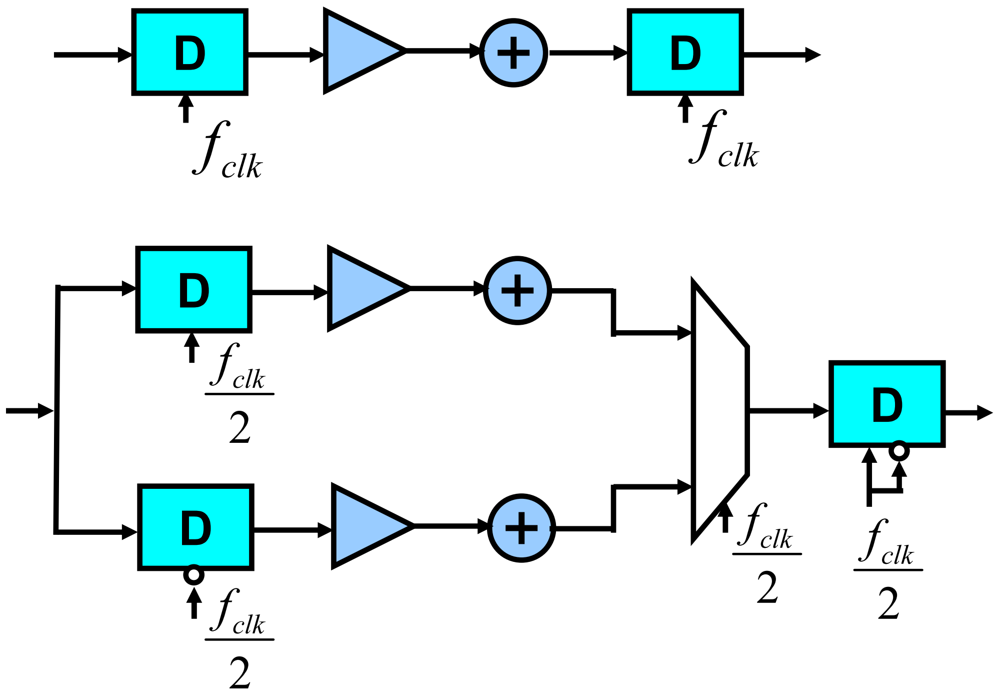
1. **Add extra registers at *all* inputs**
2. **Retime**

Courtesy of Anantha Chandrakasan. Used with permission.

Lookahead

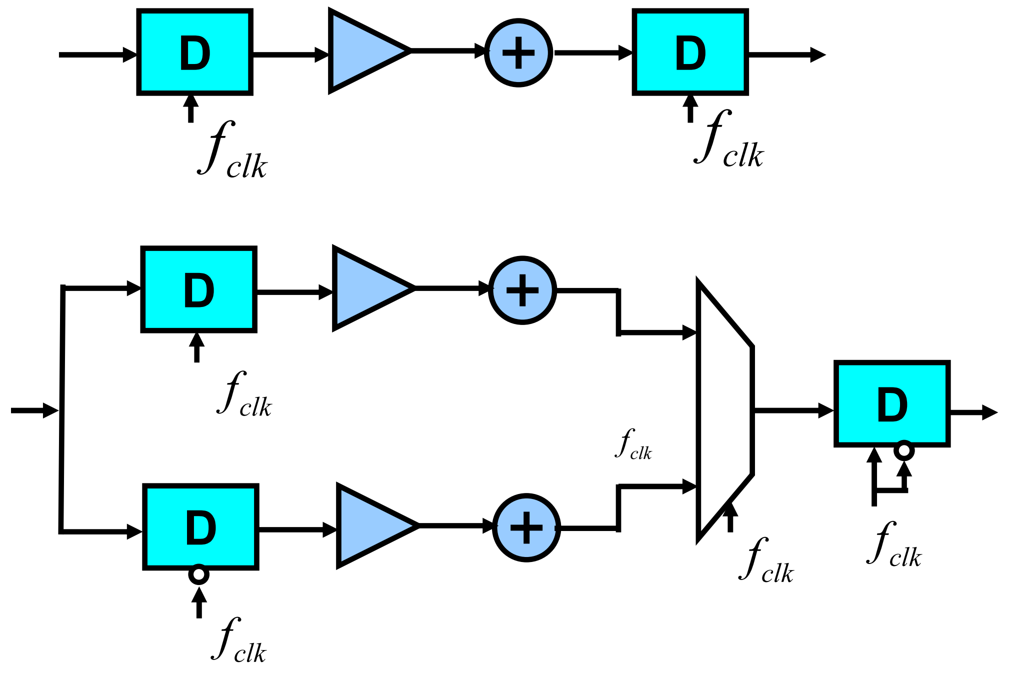

y(n) = x(n) + A y(n-1)
x(n)
y(n)
D
A
Try pipelining this structure
loop unrolling
x(n)
y(n)
D
A
2D
A
y(n) = x(n) + A[x(n-1) + A y(n-2)]
distributivity
x(n)
y(n)
D
2D
A
A
A
associativity
How about pipelining this structure!
x(n)
y(n)
D
D
D
A
A²
retiming
x(n)
y(n)
D
2D
A
A²
precomputed

Parallelism – saving power



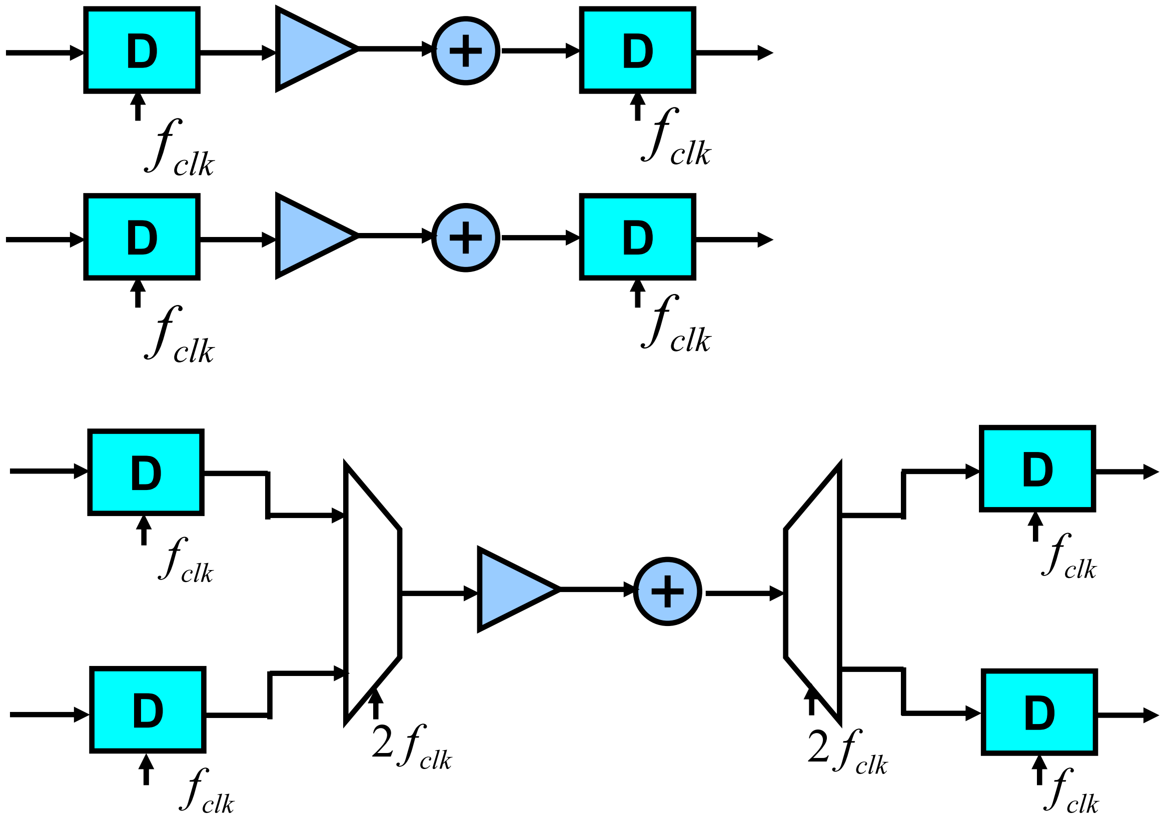
- Same throughput as nominal
  - Delay of each path relaxed 2x
  - Lower the supply to match 2x nominal delay
  - $P=f*C_{tot}*V_{dd}^2$
  - $f_{par}=f_{nom}/2$, $C_{par}=2*C_{nom}$, $V_{dd\_par}\sim=V_{dd\_nom}/2$ (sqrt(2))=> Ppar~=Pnom/4 (2)
    - Not quite right since Mux is additional overhead (but close)

Parallelism – speeding up



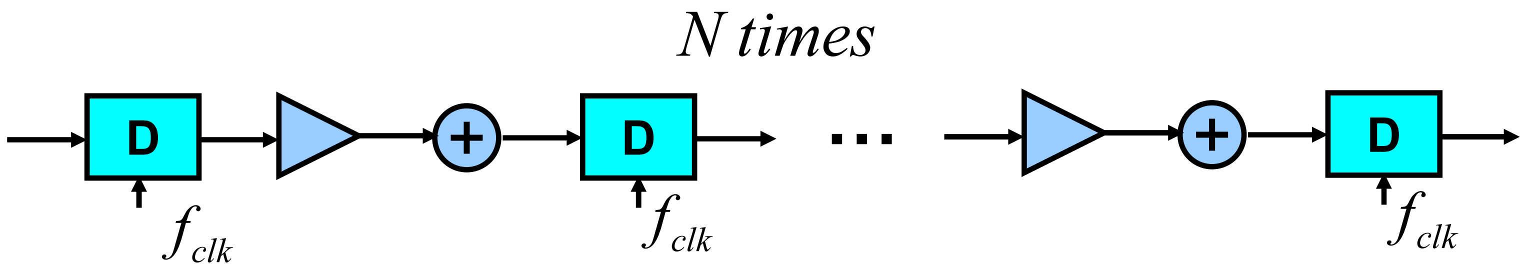
- Almost twice the throughput of the nominal design
  - Need to fit the extra mux

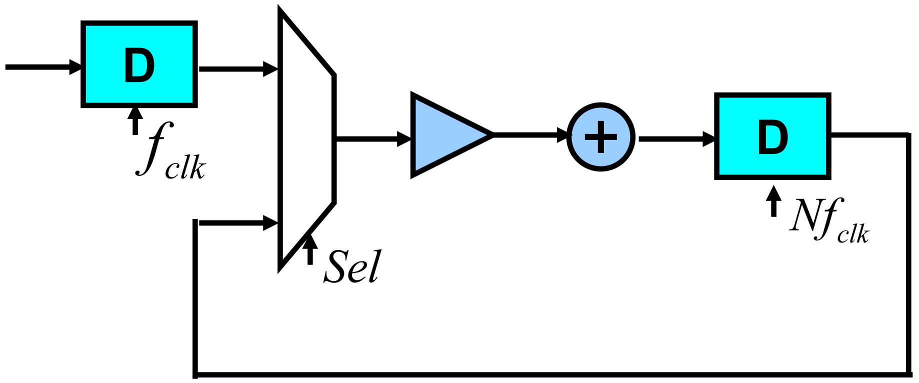
Improving area efficiency: Time multiplexing



- Save area by reusing resources
  - Need twice faster clock internally

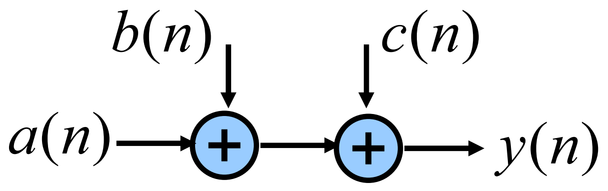
Improving area efficiency: Folding

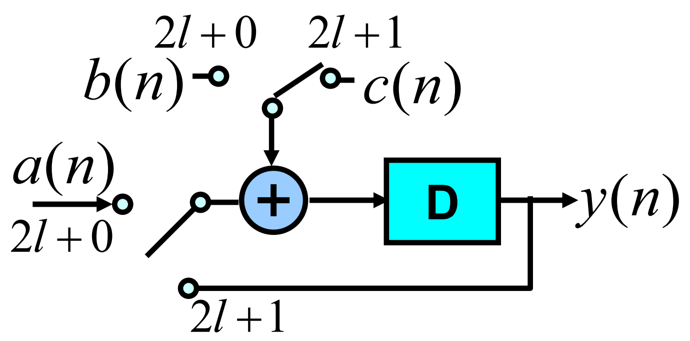




- Reuse logic and registers
  - Preserves the throughput
  - Saves area
  - Need to up-sample the internal data flow

Improving area efficiency: Folding

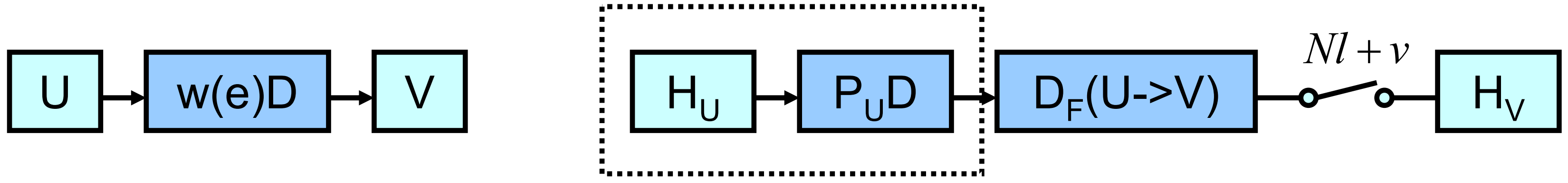




- Reuse logic and registers
  - Preserves the throughput
  - Saves area
  - Need to up-sample the internal data flow
- Easy for simple designs and folding ratios
  - Need a systematic way to do it

Systematic Folding

- Consider an edge e connecting U and V, with w(e) delays



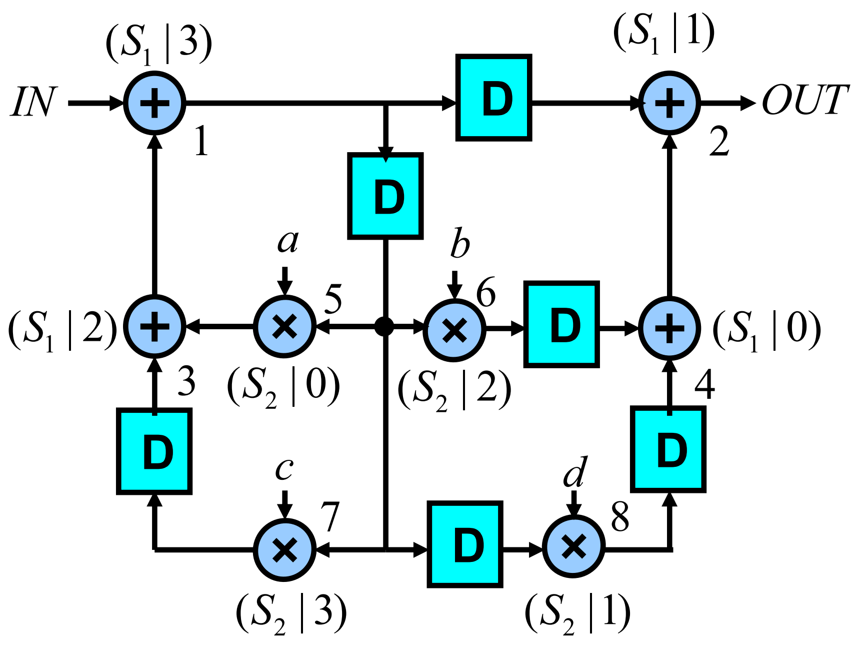
  - Nl+u and Nl+v, u and v are folding orders
    - Time partition to which the node is scheduled to execute in hardware
  - N – folding factor (number of operations folded onto a single unit)
  - If H_U pipelined by P_U stages, result available at $Nl+u+P_U$
  - The result of the l-th iteration of node U is used by l+w(e) iteration of node V
    - Executed at N(l+w(e))+v
  - Hence, the result must be stored for

$$D_F(U \xrightarrow{e} V) \quad [N(l+w(e))+v]-[Nl+P_U+u] \quad Nw(e)-P_U+v-u$$

Folding - example

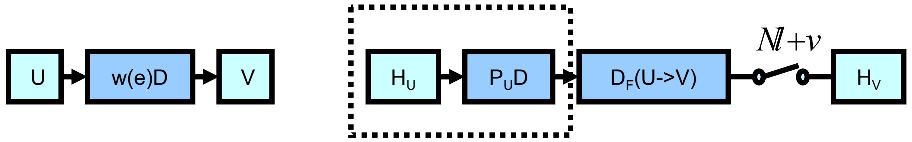
- Folding set
  - ordered set of operations executed by the same functional unit (e.g. S1={A1, 0, A2}, N=3)
    - A1 belongs to folding set S1 with folding order 0 – S1|0
    - A2 belongs to folding set S1 with folding order 2 – S1|2
    - Unit is not utilized at time instances 3l+1 due to null operation at position 1 within S1 – S1|1
- Biquad filter example (add 1, multiply 2)
  - N=4 (folded 4 times, now iteration period is 4)
    - Each node of the filter executed once every 4 time units when folded
  - Pa=1, Pm=2 (one stage pipelined add, 2-stage pipelined multiply) – units can be clocked at unit time
  - Functional units in the folded architecture execute 4 operations before the next period

Folding – biquad filter example

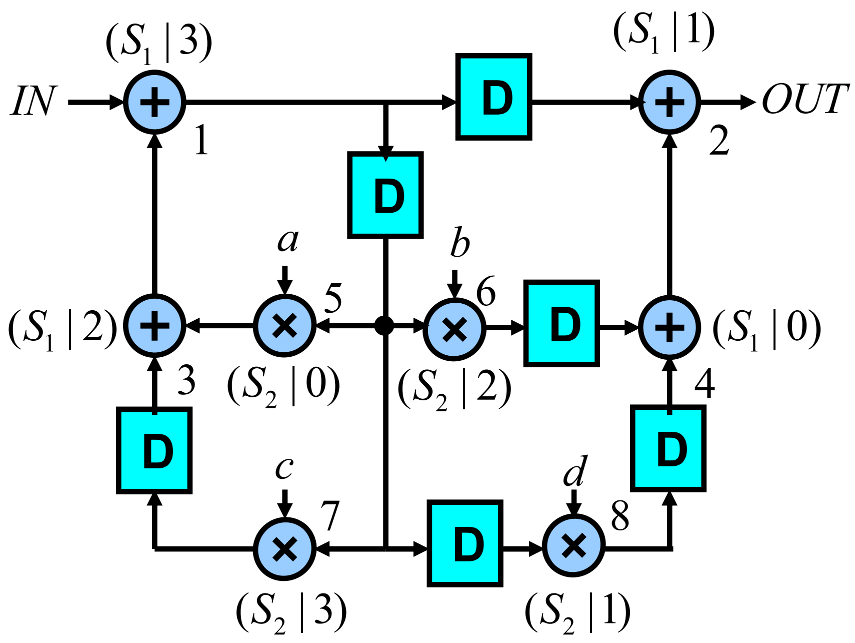


- Folding sets S1={4,2,3,1}, S2={5,8,6,7}
  - e.g. node 3 executed in the folded architecture at time instances 4l+2 (S1|2)

Folding – biquad filter example



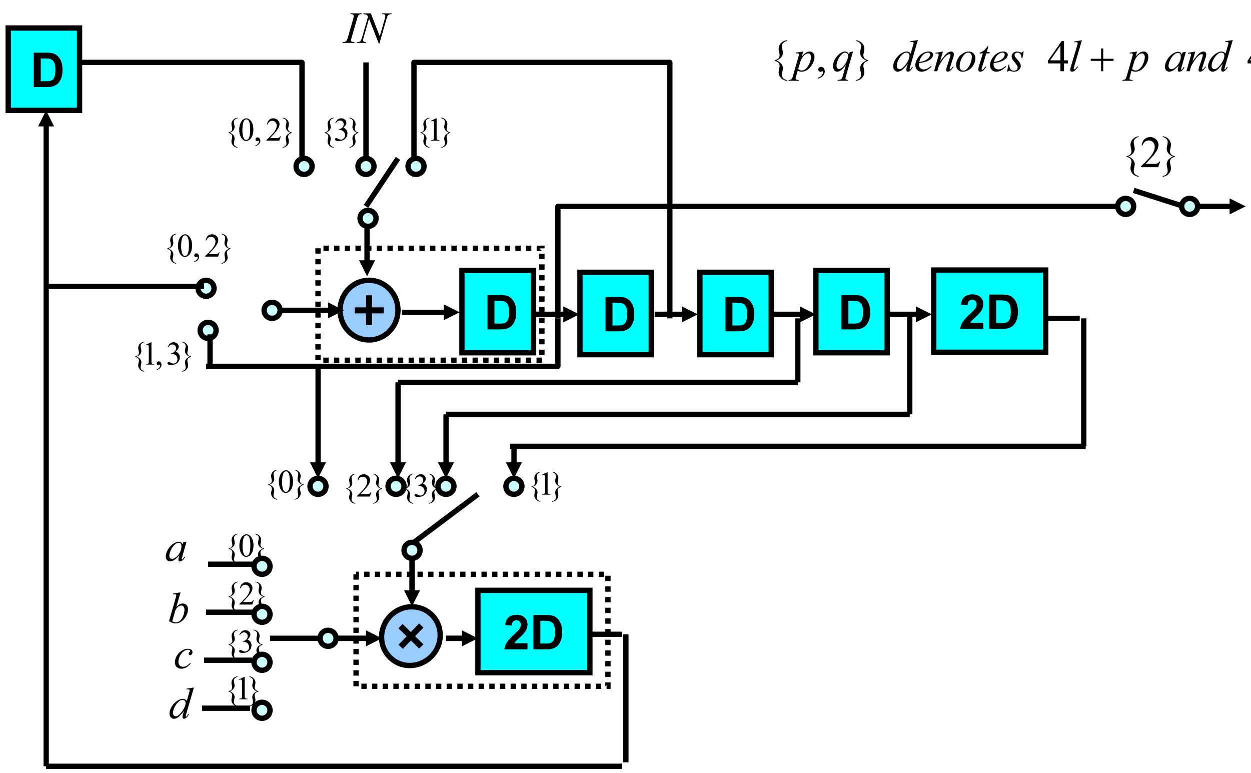
$D_F(U \xrightarrow{e} V) \quad [N(l+w(e))+v]-[Nl+P_U+u] \quad Nw(e)-P_U+v-u$



$D_F(1 \to 2)$	$4(1)-1+1-3$	1
$D_F(1 \to 5)$	$4(1)-1+0-3=0$	
$D_F(1 \to 6)$	$4(1)-1+2-3$	2
$D_F(1 \to 7)$	$4(1)-1+3-3$	3
$D_F(1 \to 8)$	$4(2)-1+1-3$	5
$D_F(3 \to 1)$	$4(0)-1+3-2$	0
$D_F(4 \to 2)$	$4(0)-1+1-0$	0
$D_F(5 \to 3)$	$4(0)-2+2-0$	0
$D_F(6 \to 4)$	$4(1)-2+0-2$	0
$D_F(7 \to 3)$	$4(1)-2+2-3=1$	
$D_F(8 \to 4)$	$4(1)-2+0-1$	1

- D_F(1->8)=5 means there is an edge from the adder to the multiplier in the folded DFG with 5 delays

Folding – biquad folded architecture

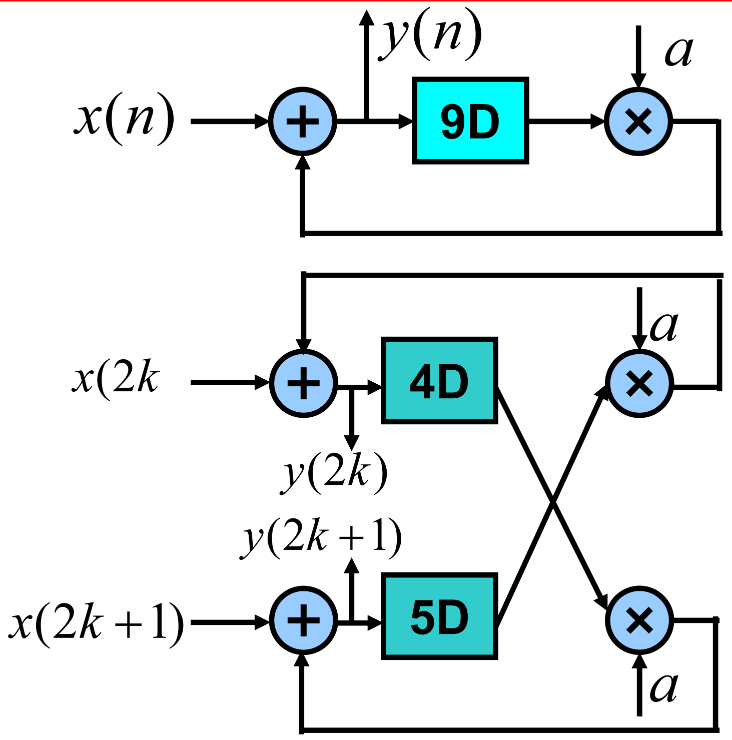


$\{p,q\}$ *denotes* $4l+p$ *and* $4l+q$

$D_F(1\rightarrow 2)$	$4(1)-1+1-3$	1
$D_F(1\rightarrow 5)$	$4(1)-1+0-3$	0
$D_F(1\rightarrow 6)$	$4(1)-1+2-3$	2
$D_F(1\rightarrow 7)$	$4(1)-1+3-3$	3
$D_F(1\rightarrow 8)$	$4(2)-1+1-3$	5
$D_F(3\rightarrow 1)$	$4(0)-1+3-2$	0
$D_F(4\rightarrow 2)$	$4(0)-1+1-0$	0
$D_F(5\rightarrow 3)$	$4(0)-2+2-0$	0
$D_F(6\rightarrow 4)$	$4(1)-2+0-2$	0
$D_F(7\rightarrow 3)$	$4(1)-2+2-3=1$	
$D_F(8\rightarrow 4)$	$4(1)-2+0-1$	1

- D_F(1->8)=5 means there is an edge from the adder to the multiplier in the folded DFG with 5 delays
  - Since this edge ends at node 8, which has folding order 1, folded edge is switched at the input of the multiplier in the folded DFG at 4l+1

Unfolding



$$y(n) \quad ay(n-9)+x(n)$$

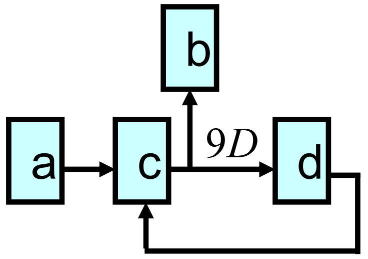
$$y(2k) \quad ay(2k-9)+x(2k)$$

$$y(2k+1) \quad ay(2k+1-9)+x(2k+1)$$

- Increase throughput
  - Can reveal hidden dependencies – schedule to a smaller iteration period
- Design parallel architectures at the word or bit level
  - Word(bit)-parallel architectures from word(bit)-serial
    - Increase throughput or decrease power consumption
- Each delay is J-slow (for J-unfolded system)
  - y(2k)=ay(2(k-5)+1)+x(2k) and y(2k+1)=ay(2(k-4)+0)+x(2k+1)

Unfold J-times

- Data flow graph (DFG)
- For each node in DFG, J nodes in J-unfolded DFG
- For each edge in DFG, J edges in J-unfolded DFG
- Unfolding algorithm
  - For each node U in the original DFG
    - Draw J nodes U_0, U_1, …, U_{J-1}
  - For each edge U->V with w delays in the original DFG
    - Draw J edges U_i->$V_{(i+w) \bmod J}$ with floor((i+w)/J) delays for i=0…J-1

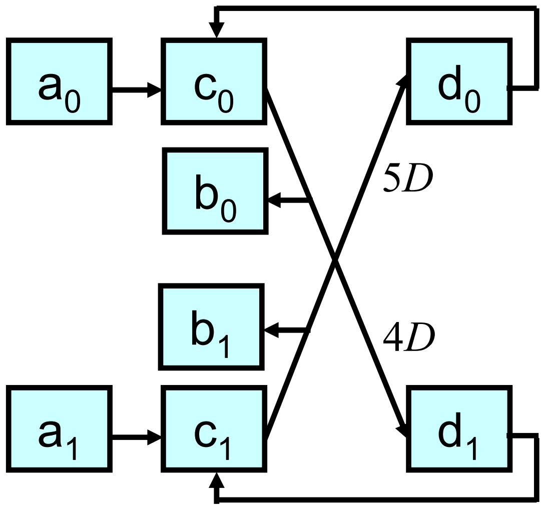


$a \to c$

$c \to b$

$c \xrightarrow{9D} d$

$d \to c$



$a_0 \to c_0$

$c_0 \to b_0$

$c_0 \xrightarrow{floor((0+9)/2)D} d_{(0+9)\bmod 2}$

$d_0 \to c_0$

$a_1 \to c_1$

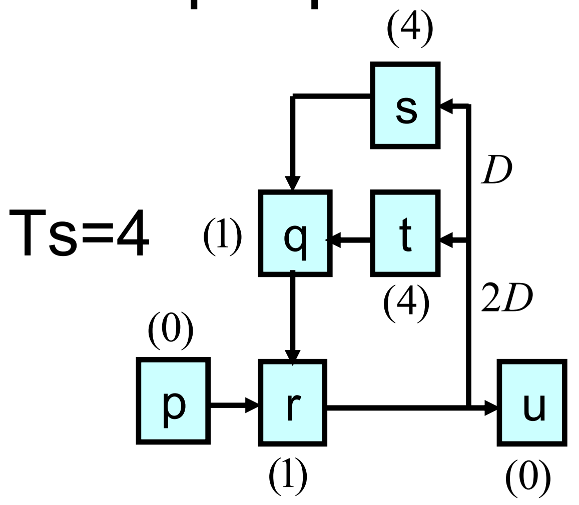
$c_1 \to b_1$

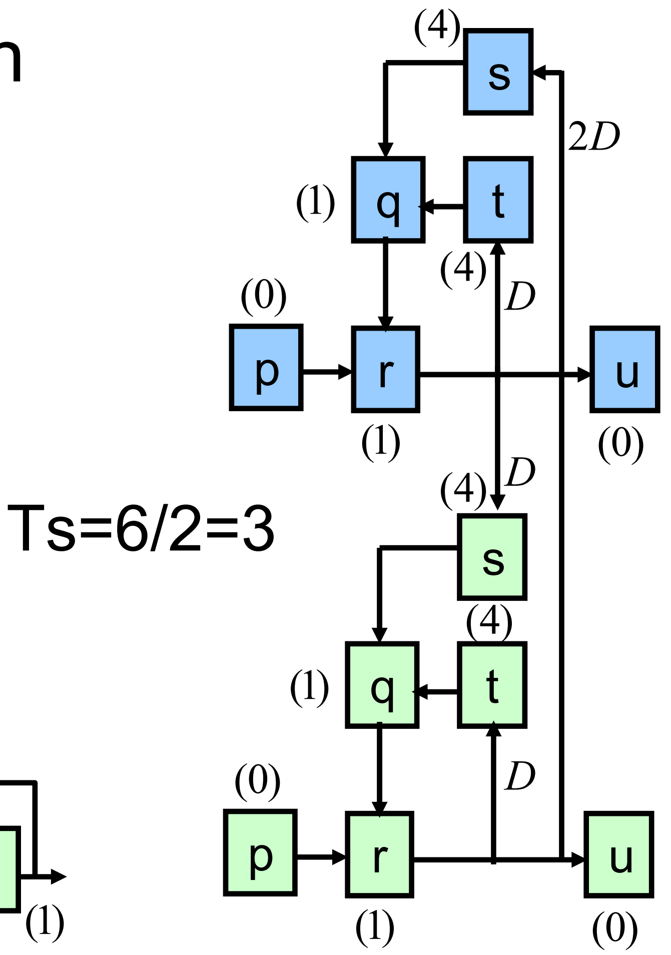
$c_1 \xrightarrow{floor((1+9)/2)D} d_{(1+9)\bmod 2}$

$d_1 \to c_1$

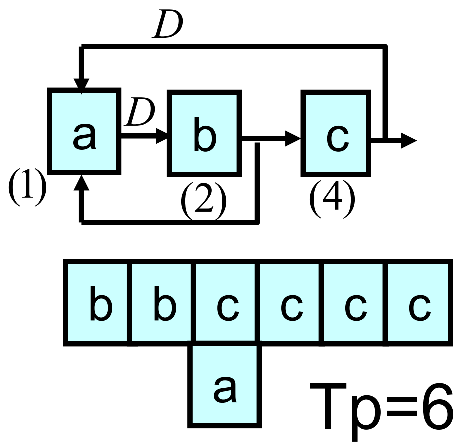
Unfolding applications

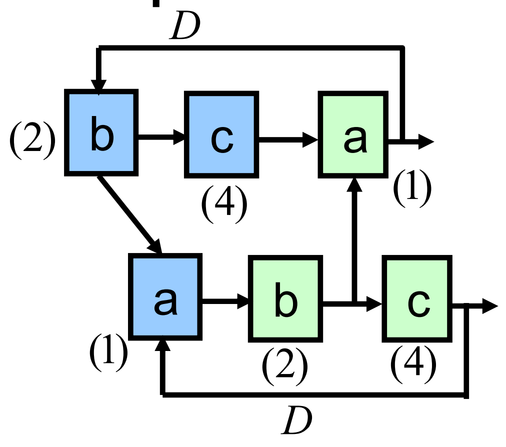
- Sample period reduction

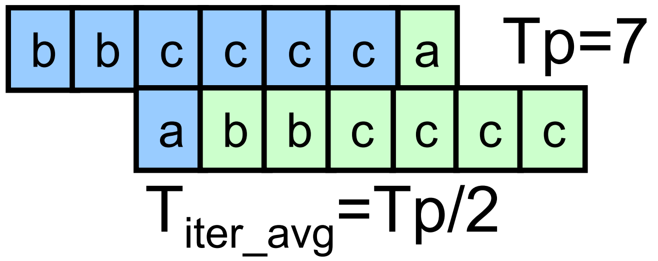




- Reduce iteration period

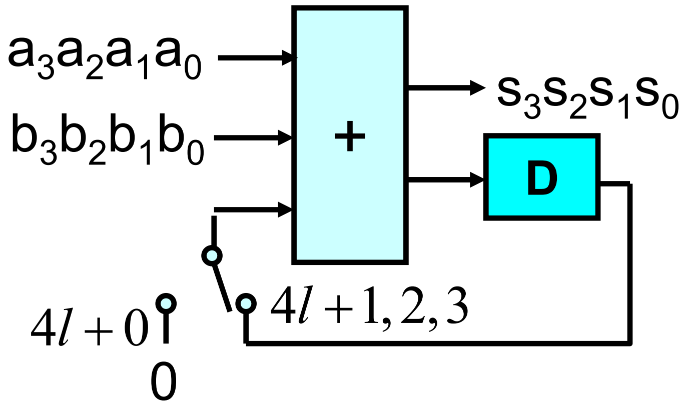


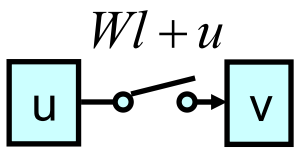




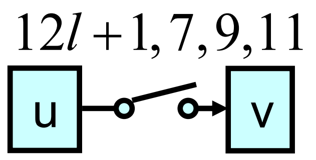
Unfolding applications

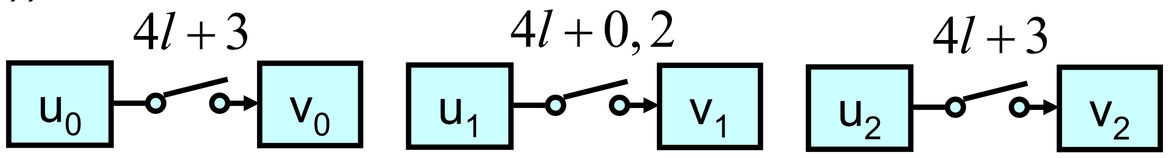
- Parallel processing
  - Turn bit-serial processing into bit-parallel





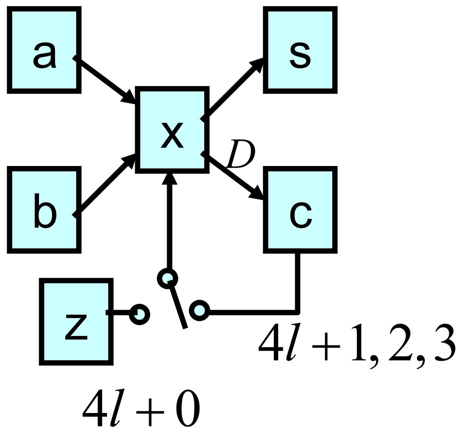
  - Unfolding with switches
    - Write the switching instance as $Wl+u = J(W'l+floor(u/J))+(u \bmod J)$
    - Draw an edge with no delays in the unfolded graph from node $U_{u \bmod J}$ to the node $V_{u \bmod J}$, which is switched at time $(W'l+floor(u/J))$





Bit-serial adder example

- Start from bit-serial adder DFG



References

- A. P. Chandrakasan, S. Sheng, and R. W. Brodersen, "Low-power CMOS digital design" IEEE Journal Solid-State Circuits, April 1992
  - Second most cited JSSC paper
- Keshab Parhi "VLSI Digital Signal Processing Systems"
  - Read: Chapters 3, 4, 5, 6 (10, 13 and 17 optional)