Lecture

11

ECE 216B

Data-Flow Graph Model

Prof. Dejan Marković

ee216b@gmail.com

Agenda

- **DFG modeling**
- **Architecture transformations**
  - Retiming example
- **Optimization methods**
  - Scheduling
  - Retiming

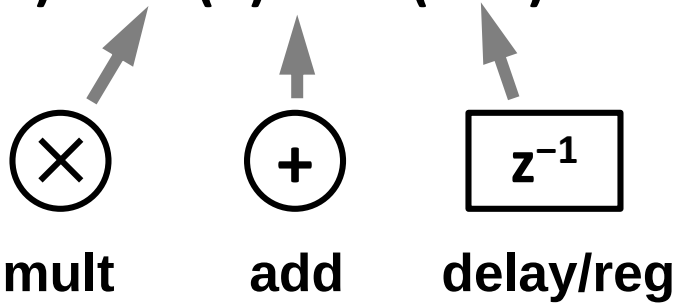
Iteration

- **Iterative nature of DSP algorithms**
  - Executes a set of operations in a defined sequence
  - One round of these operations constitutes an iteration
  - Algorithm output computed from result of these operations
- **Graphical representations of iterations** [1]
  - Block diagram (BD)
  - Signal-flow graph (SFG)
  - Data-flow graph (DFG)
  - Dependence graph (DG)
- **Example: 3-tap filter iteration**
  - $y(n) = a \cdot x(n) + b \cdot x(n-1) + c \cdot x(n-2), \quad n = \{0, 1, \ldots, \infty\}$
  - **Iteration:** 3 multipliers, 2 adders, 1 output $y(n)$

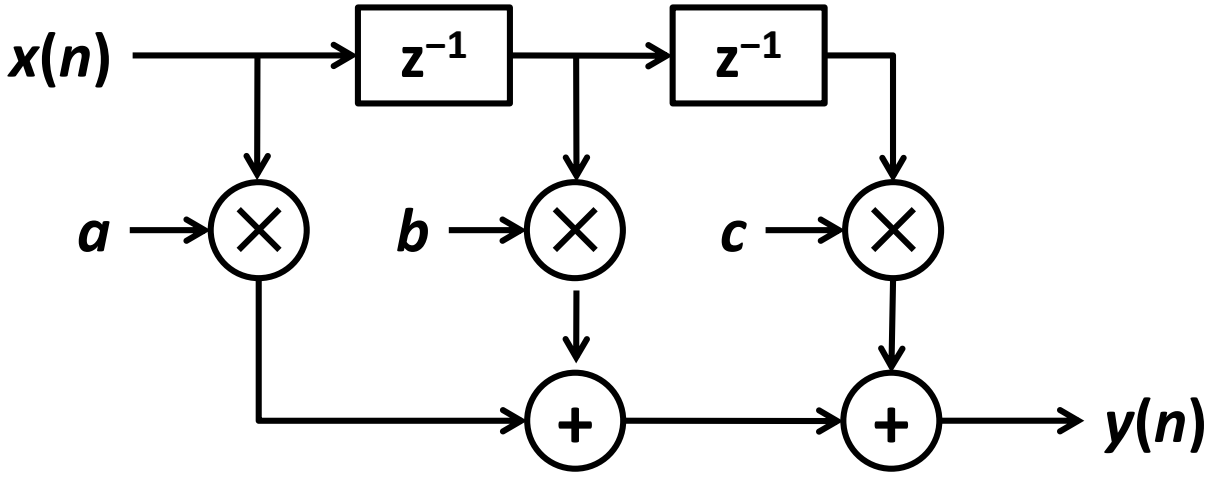
[1] K.K. Parhi, VLSI Digital Signal Processing Systems: Design and Implementation, John Wiley & Sons Inc., 1999.

Block Diagram Representation

$$y(n) = a \cdot x(n) + b \cdot x(n-1) + c \cdot x(n-2), \qquad n = \{0, 1, \ldots, \infty\}$$

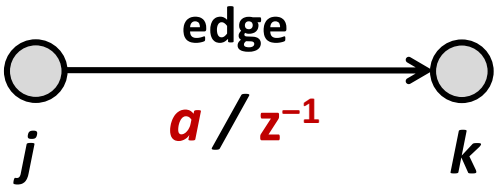


- **Block diagram of 3-tap FIR filter**

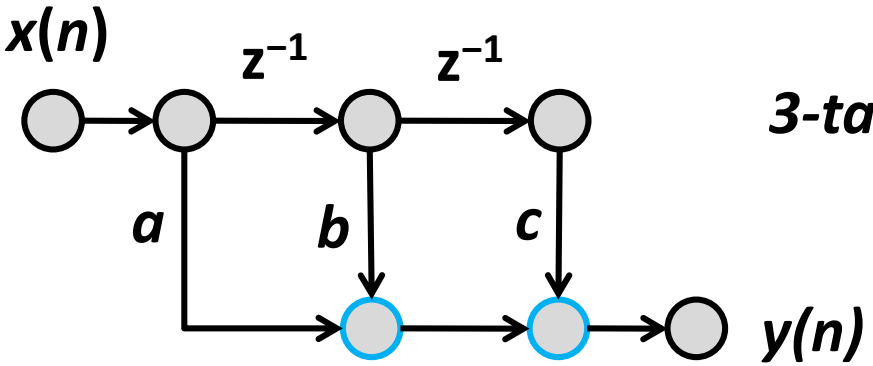


Signal-Flow Graph Representation

- **Network of nodes and edges**
  - **Edges** are signal flows or paths with non-negative # of regs
    - Linear transforms, multiplications or registers shown on edges
  - **Nodes** represent computations, sources, sinks
    - Adds (> 1 input), sources (no input), sinks (no output)



***constant multiplication* (*a*)**
or register* (z^{-1}) *on edges



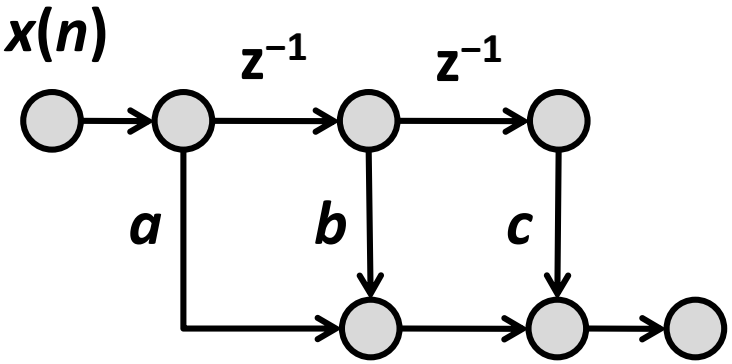
3-tap FIR filter signal-flow graph

source node: x(n)
sink node: y(n)

Transposition of a SFG

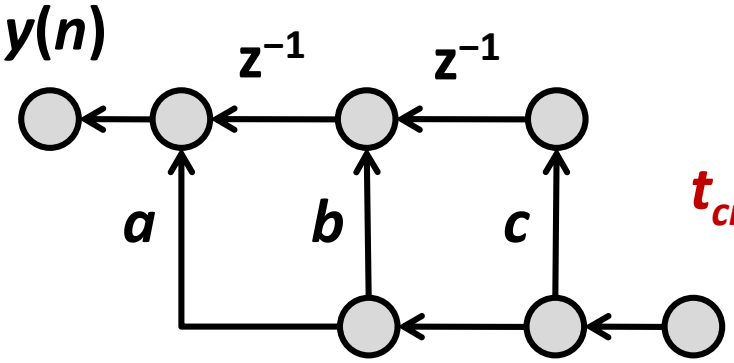
- **Transposed SFG functionally equivalent**
  - Reverse direction of signal flow edges
  - Exchange input and output nodes
- **Commonly used to reduce critical path in design**

Original SFG



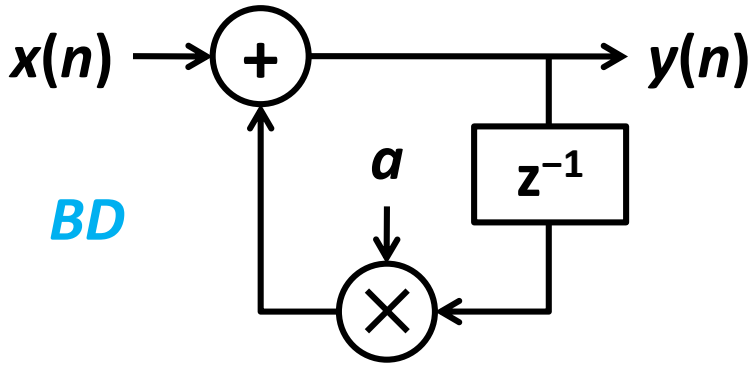
$t_{crit} = 2t_{add} + t_{mult}$

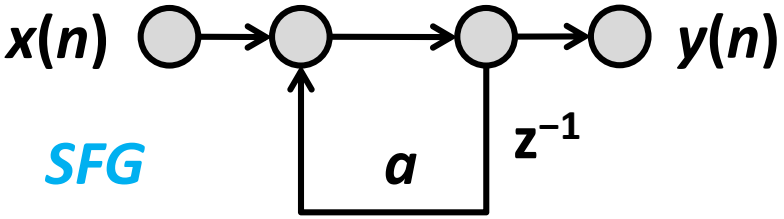
Transposed SFG

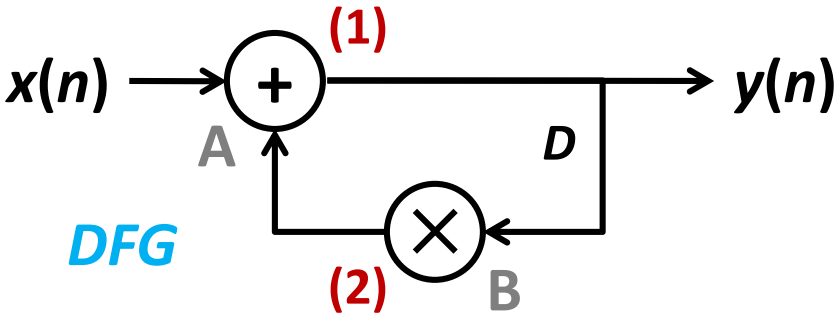


$t_{crit} = t_{add} + t_{mult}$

Different Representations



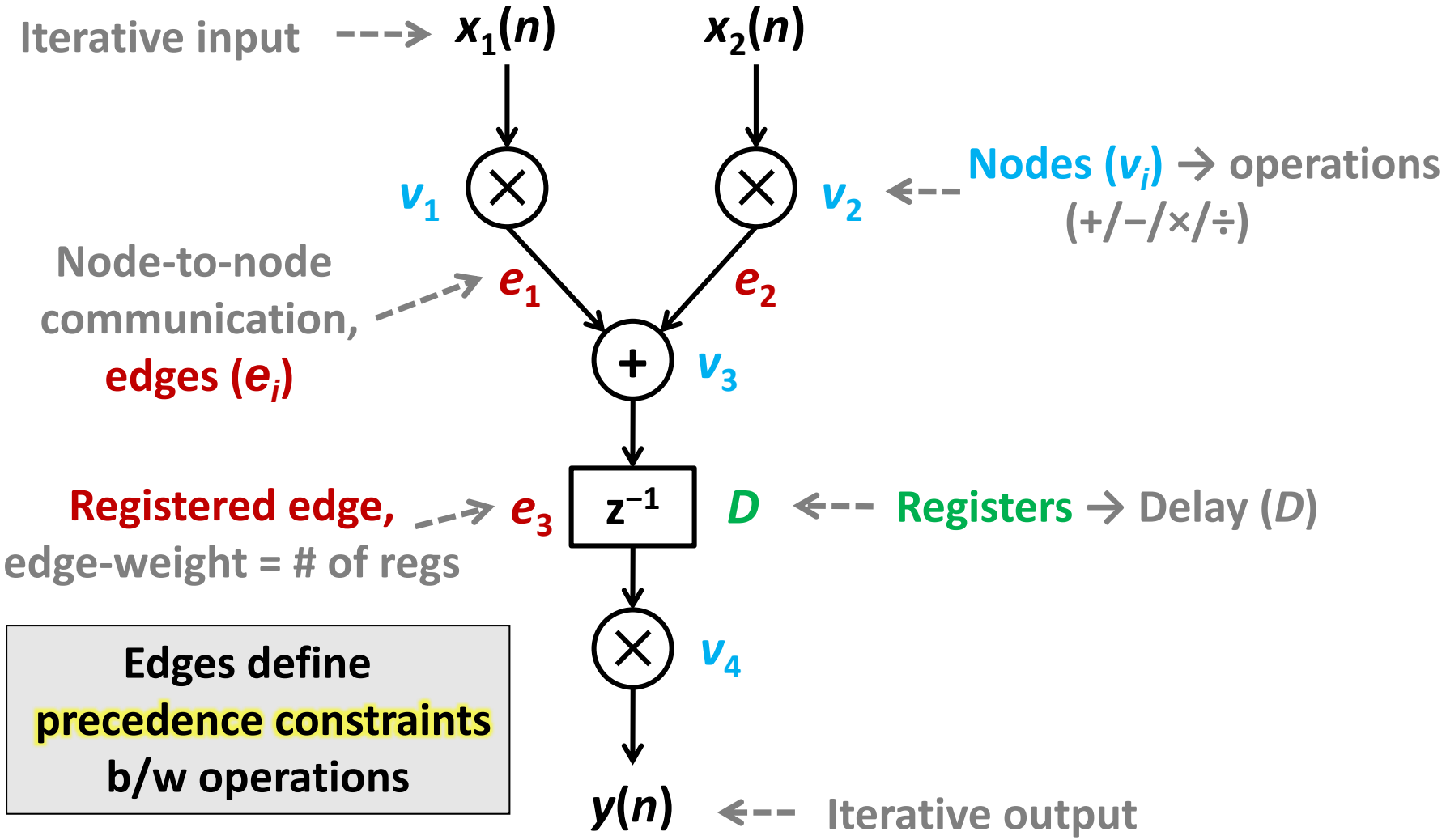




- **Block Diagram (BD)**
  - Close to hardware
  - Computations, delays shown through blocks
- **Signal-flow Graph (SFG)**
  - Multiplications, delays shown on edges
  - Source, sink, add are nodes
- **Data-flow Graph (DFG)**
  - Computations on nodes A, B
  - (delays) shown on edges
  - Computation time in brackets next to the nodes

Data-Flow Graphs

Graphical representation of signal flow in an algorithm

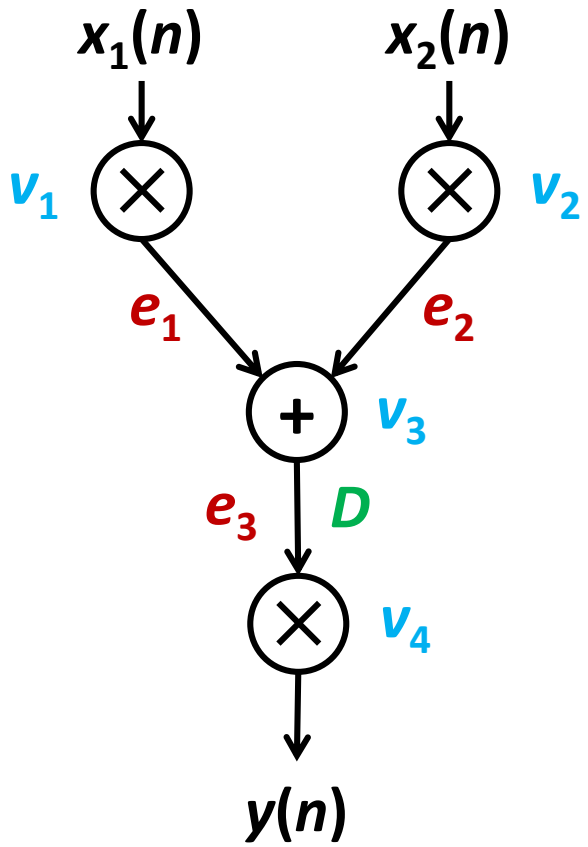


Edges define precedence constraints b/w operations

Formal Definition of DFGs

A directed DFG is denoted as G = <V, E, d, w>

V	*Set of vertices (nodes) of **G**. The vertices represent operations.*
d	*Vector of logic delay of vertices. **d(v)** is the logic delay of vertex **v**.*
E	***Set** of directed edges of **G**. A directed **edge e** from vertex **u** to vertex **v** is denoted as **e:u → v.***
w(e)	*Number of sequential delays (registers) on the edge e, also referred to as the **weight of the edge.***
p:u→ v	*Path starting from vertex **u**, ending in vertex **v**.*
D	*Denotes **register** on an edge.*

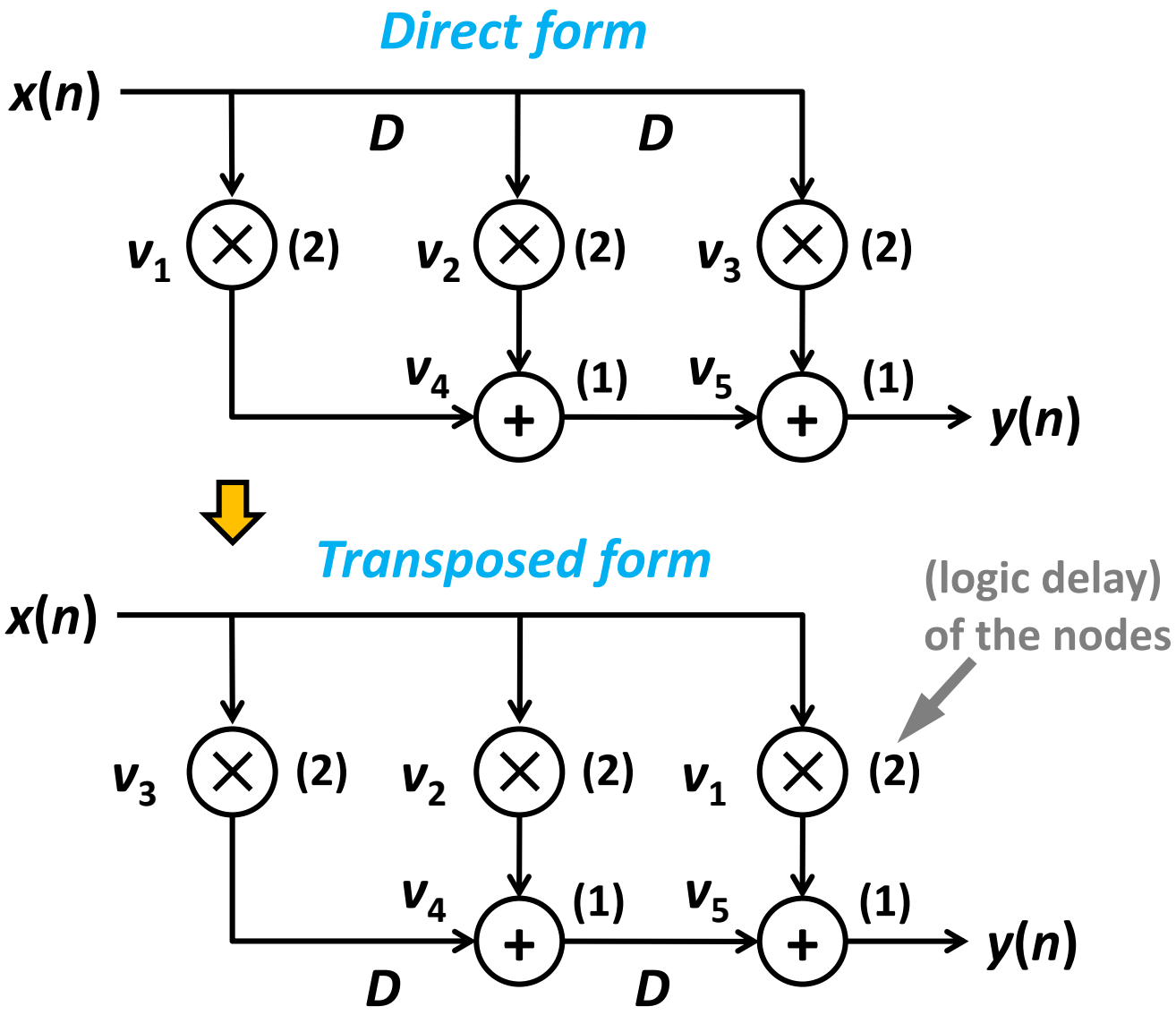


e_1 : ***Intra-iteration*** **edge**

e_3 : ***Inter-iteration*** **edge**

$w(e_1) = 0$, $w(e_3) = 1$

Example 11.1: DFGs for a 3-tap FIR Filter



Matrix Representation

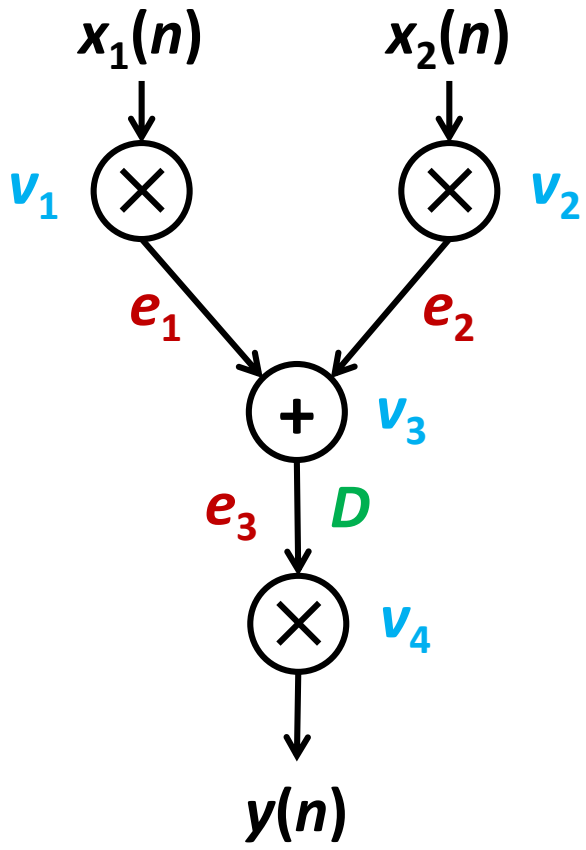
- **DFG matrix A, dimension $|V| \times |E|$**
  - $a_{ij} = 1$, *if edge* e_j *starts from node* v_i
  - $a_{ij} = -1$, *if edge* e_j *ends in node* v_i
  - $a_{ij} = 0$, *if edge* e_j *neither starts, nor ends in node* v_i

edges

nodes

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -1 & -1 & 1 \\ 0 & 0 & -1 \end{bmatrix}$$

Matrix A for graph *G*

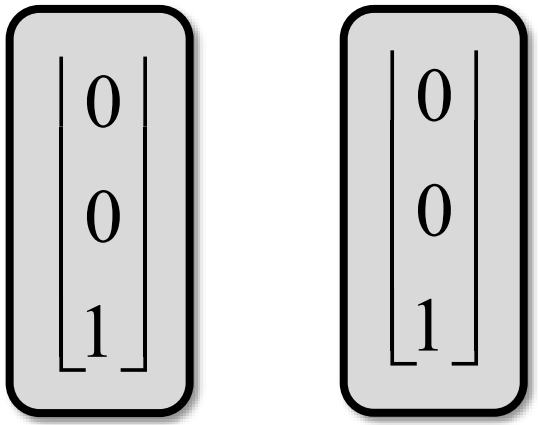


Data-flow graph *G*

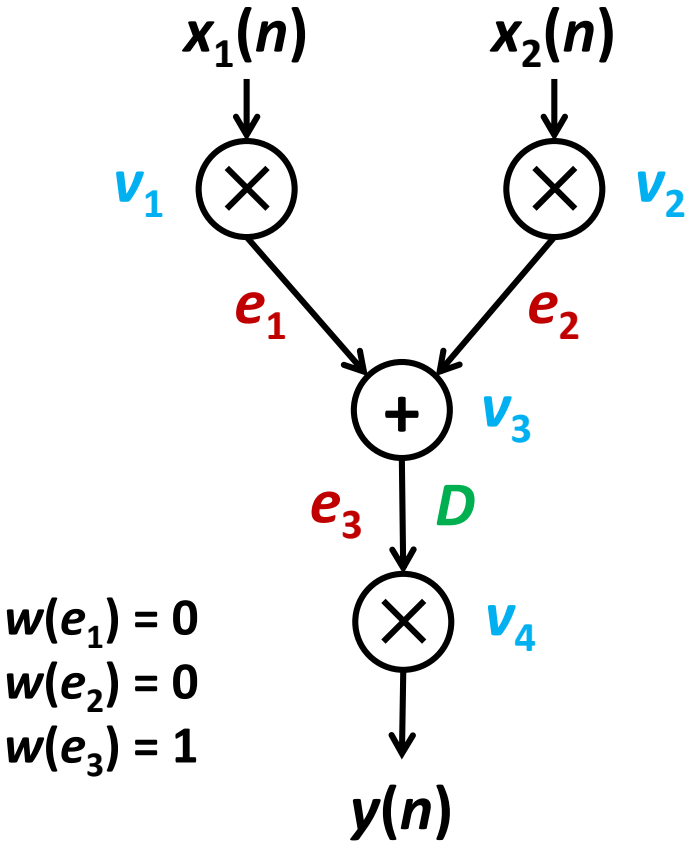
[2] R. Nanda, DSP Architecture Optimization in MATLAB/Simulink Environment, M.S. Thesis, University of California, Los Angeles, June 2008.

Matrix Representation

- **Weight vector *w***
  - dimension $|E| \times |1|$
  - $w_j = w(e_j)$, *weight of edge* e_j
- **Pipeline vector *du***
  - dimension $|E| \times |1|$
  - du_j = *pipeline depth of source node u of edge* e_j

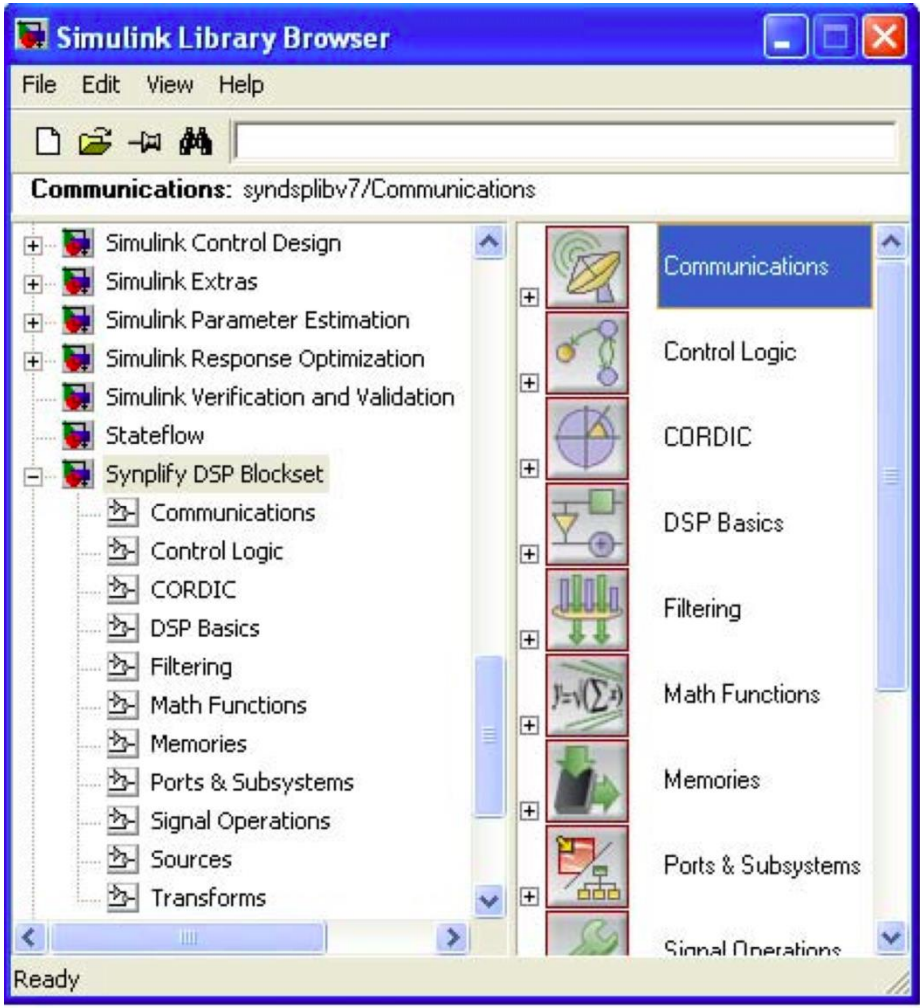


Vector *w* Vector *du*



Data-flow graph *G*

Simulink DFG Modeling

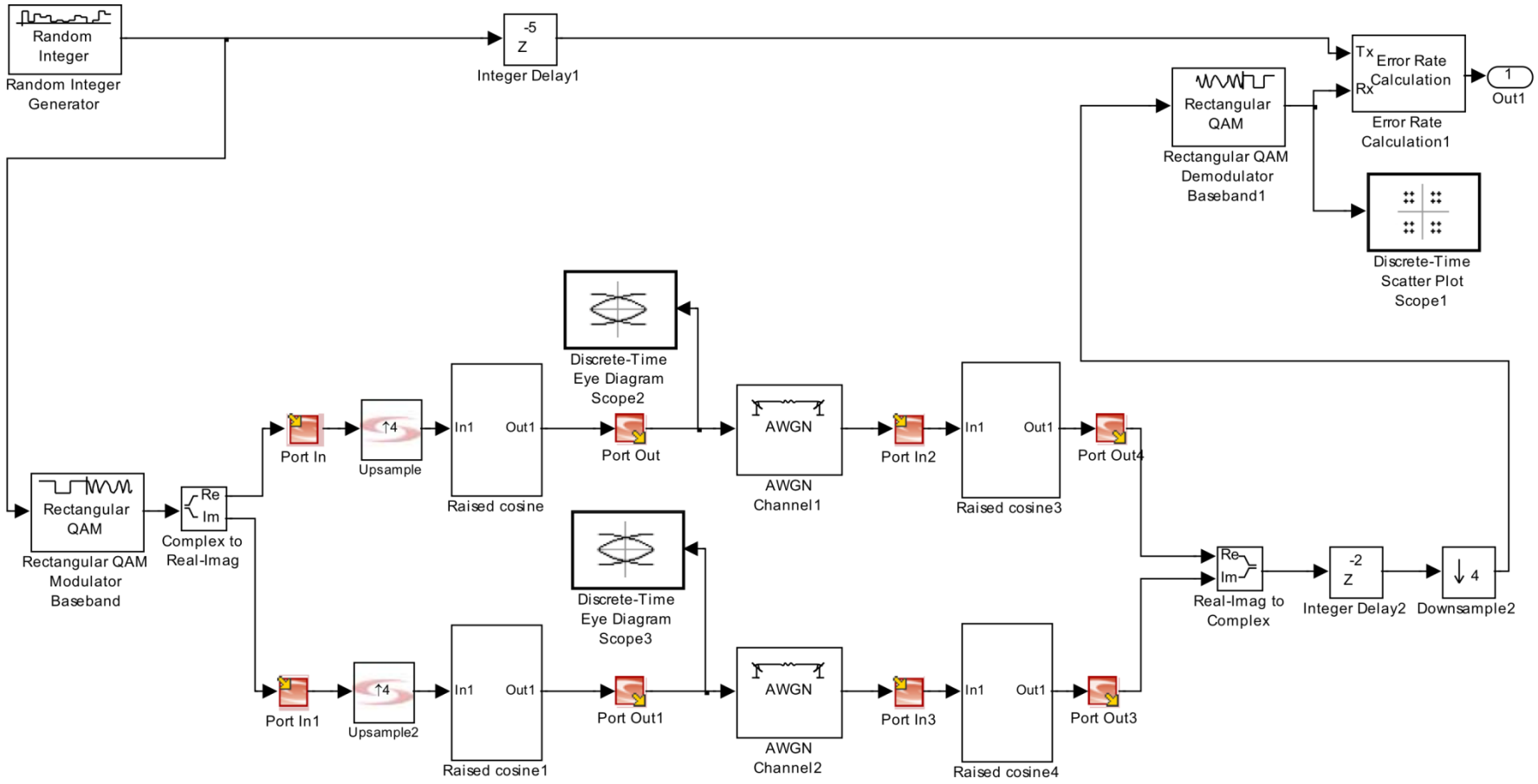


Synplify DSP block library

- **Drag-and-drop Simulink flow**
- **Allows easy modeling**
- **Predefined libraries contain DSP macros**
  - Xilinx XSG
  - Synplify DSP
- **Simulink goes a step beyond modeling macros**
  - Functional simulation of complex systems possible
  - On-the-fly RTL generation through Synplify DSP

DFG Example

- QAM modulation and demodulation
- Combination of Simulink and Synplify DSP blocks



Summary

- **Graphical representations of DSP algorithms**
  - Block diagrams
  - Signal-flow graphs
  - Data-flow graphs
- **Matrix abstraction of data-flow graph properties**
  - Useful for modeling architectural transformations
- **Simulink DSP modeling**
  - Construction of block diagrams in Simulink
  - Functional simulation, RTL generation
  - Data-flow property extraction

Architecture Transformations

DFG Realizations

- **DFGs can be realized with several architectures**
  - Change graph structure without changing functionality
  - Observe transformations in energy-area-delay space

- **DFG Transformations**
  - Retiming
  - Pipelining
  - Time-multiplexing/folding
  - Parallelism

  } **Scheduling & retiming**

- **Choice of the architecture**
  - Dictated by system specifications

Retiming [1]

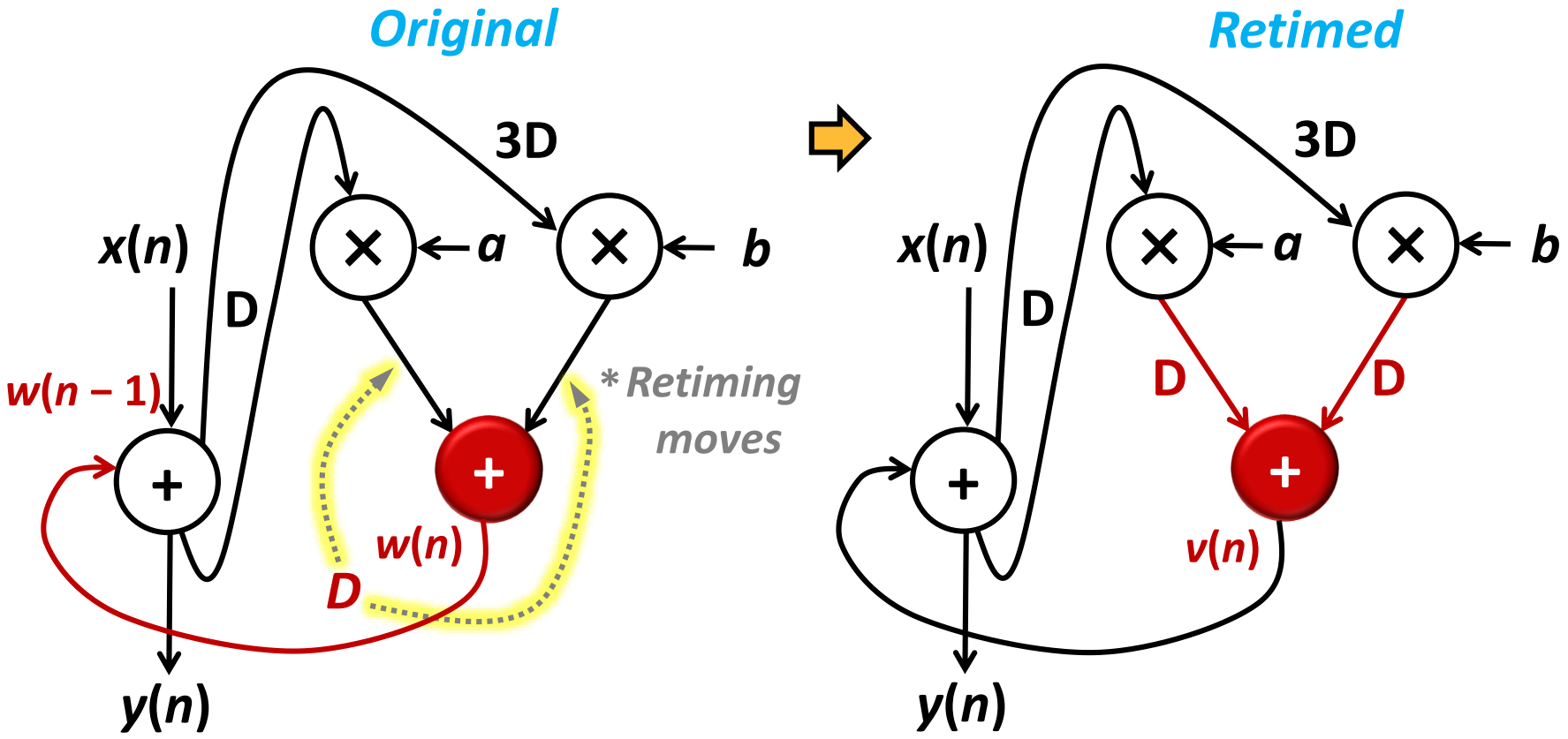
- **Registers in a flow graph can be moved across edges**
- **Movement should not alter DFG functionality**
- **Benefits**
  - Higher speed
  - Lower power through V_{DD} scaling
  - Not very significant area increase
  - Efficient automation using polynomial-time CAD algorithms [2]

[1] C. Leiserson and J. Saxe, "Optimizing synchronous circuitry using retiming," A*lgorithmica,* vol. 2, no. 3, pp. 211–216, 1991.

[2] R. Nanda, DSP Architecture Optimization in MATLAB/Simulink Environment, M.S. Thesis, University of California, Los Angeles, 2008.

Retiming

- **Register movement in the flow graph without functional change**

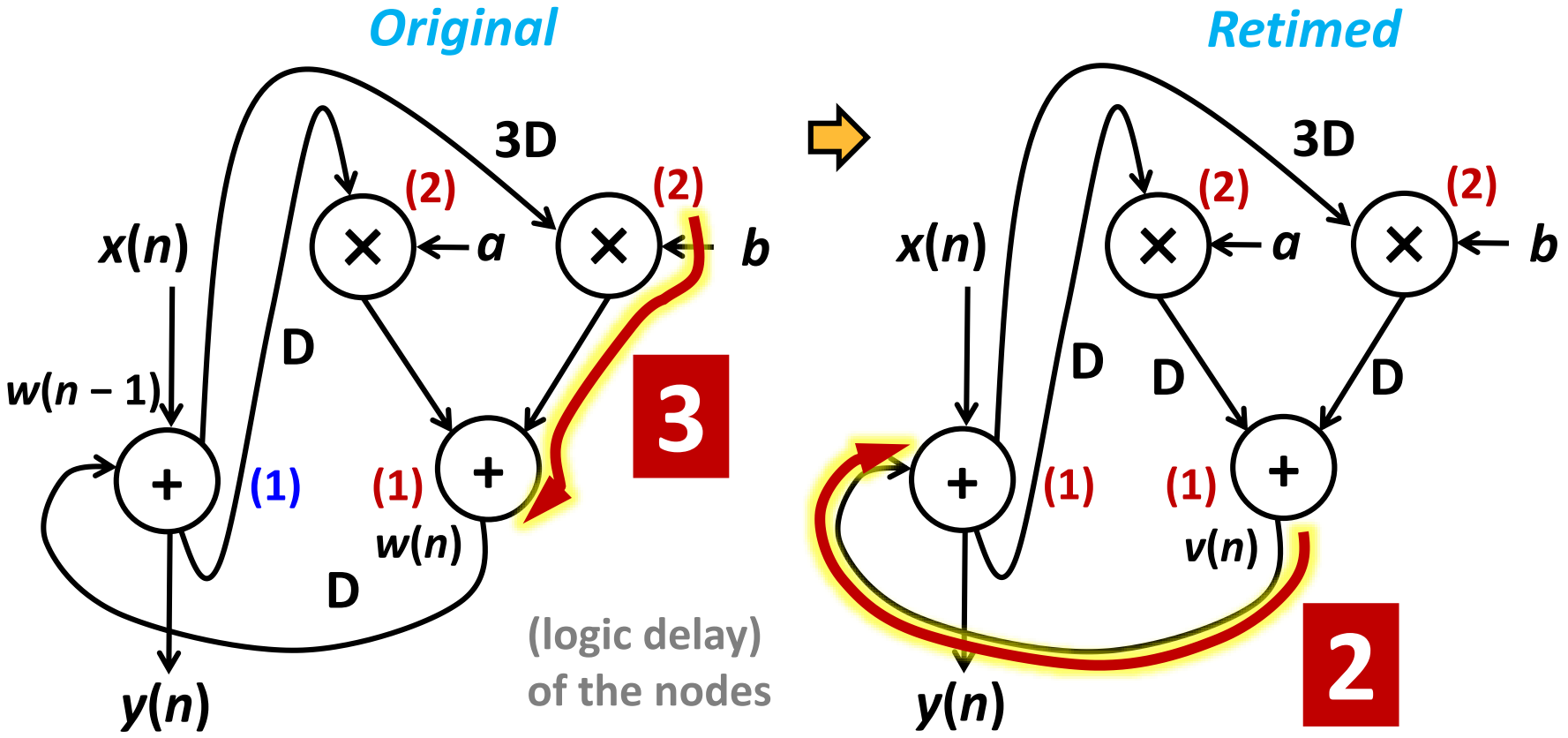


$w(n) = a \cdot y(n-1) + b \cdot y(n-3)$
$y(n) = x(n) + w(n-1)$
$y(n) = x(n) + a \cdot y(n-2) + b \cdot y(n-4)$

$v(n) = a \cdot y(n-2) + b \cdot y(n-4)$
$y(n) = x(n) + v(n)$
$y(n) = x(n) + a \cdot y(n-2) + b \cdot y(n-4)$

Retiming for Higher Throughput

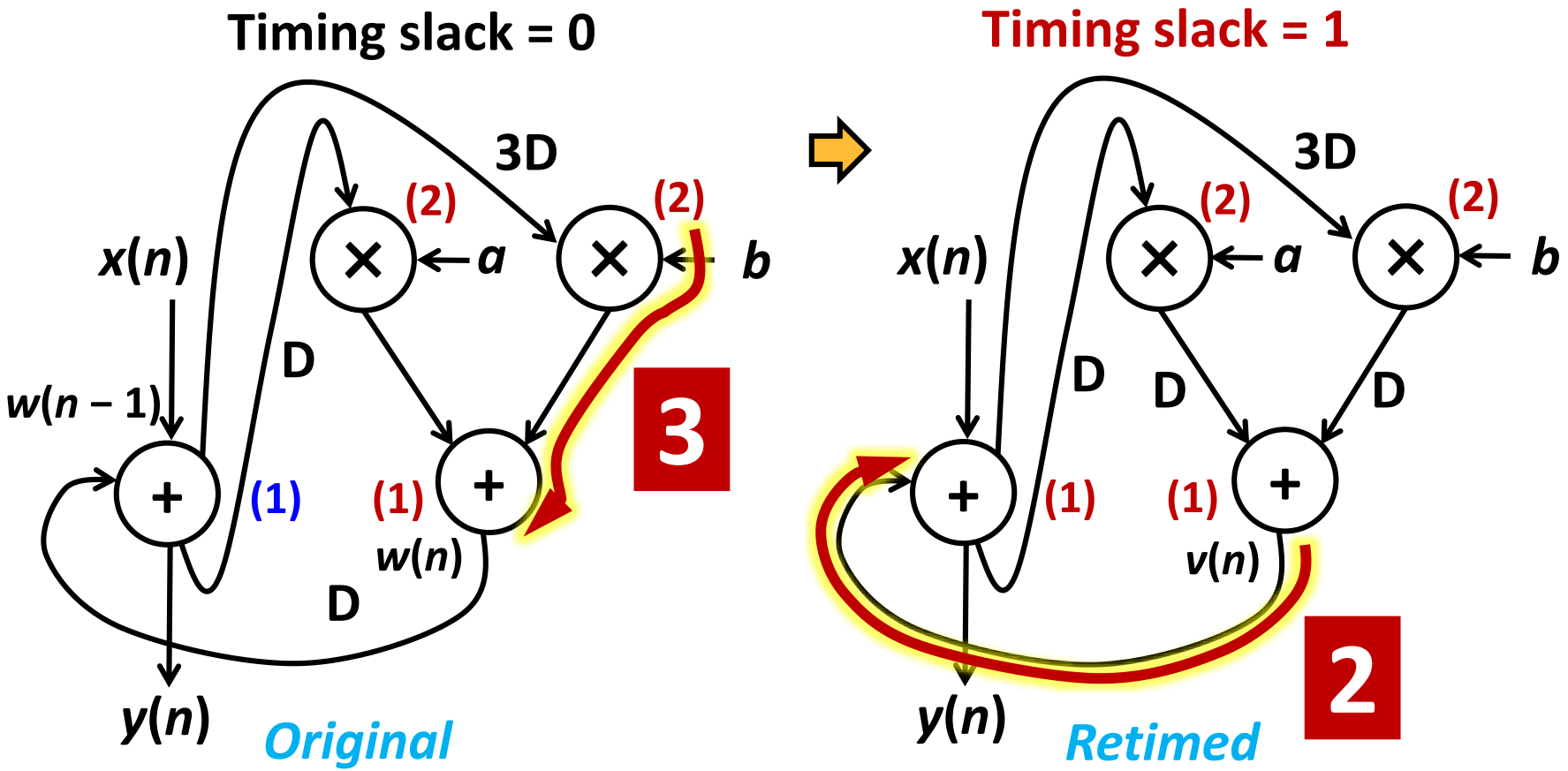
Register movement can shorten the critical path of the circuit



Critical path reduced from 3 time units to 2 time units

Retiming for Lower Power

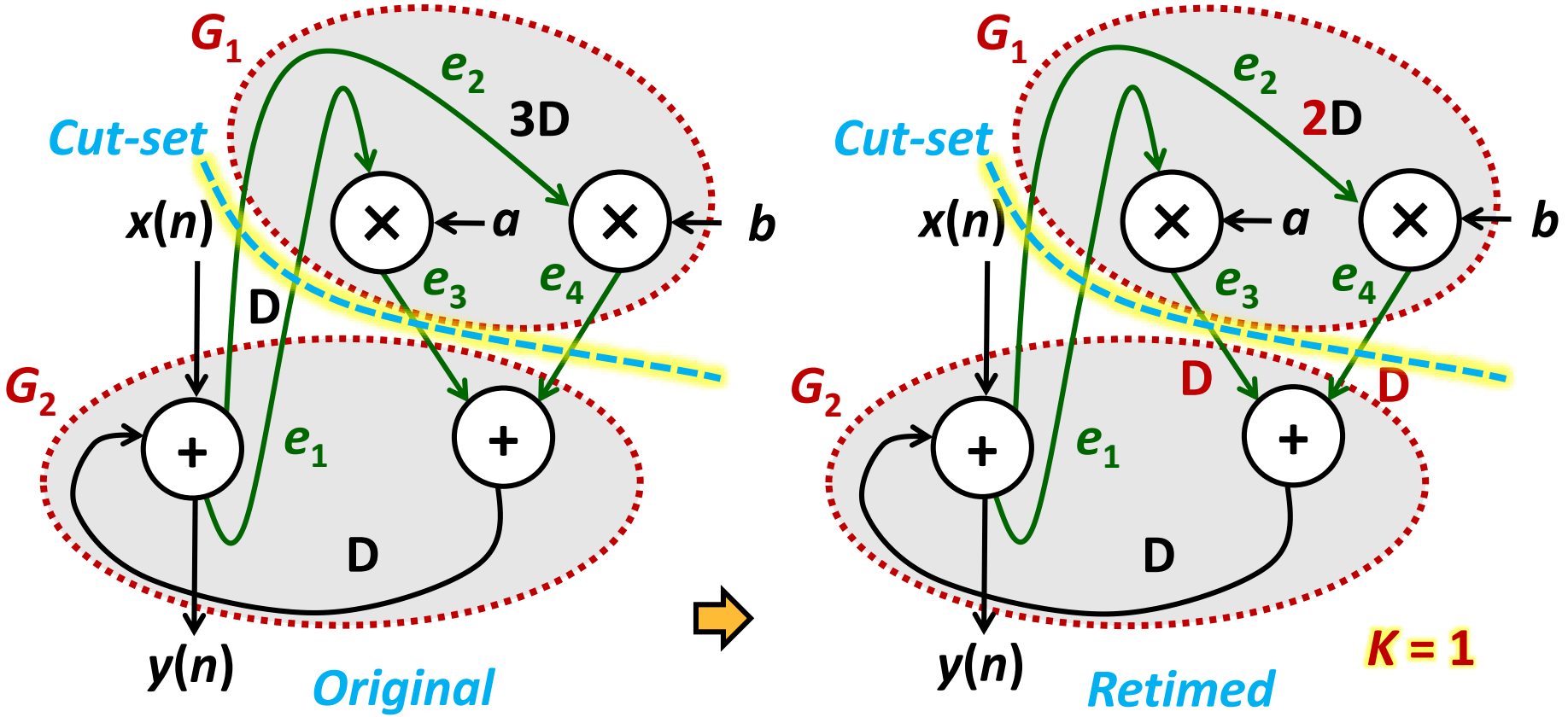
Desired throughput: 1/3



Exploit additional combinational slack for voltage scaling

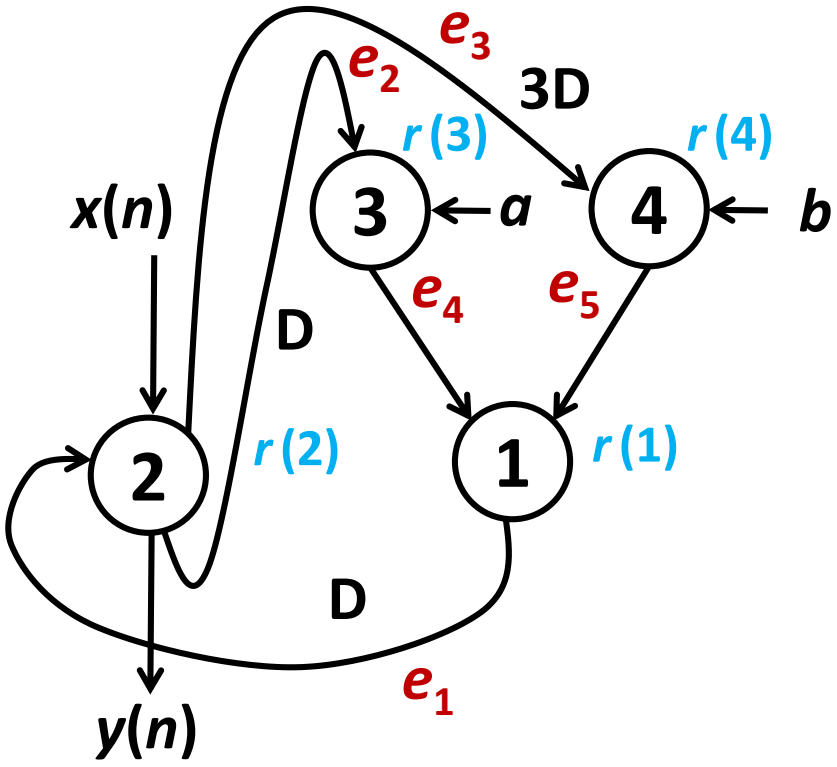
Retiming Cut-sets

- **Make cut-sets which divide the DFG in two disconnected halves**
  - Add ***K*** delays to each edge from G_1 to G_2
  - Remove ***K*** delays from each edge from G_2 to G_1



Mathematical Modeling [2]

- Assign retiming weight $r(v)$ to every node in the DFG
- Define edge-weight $w(e)$ = number of registers on the edge
- Retiming changes $w(e)$ into $w_r(e)$, the retimed weight



$w(e_1) = 1$
$w(e_2) = 1$
$w(e_3) = 3$
$w(e_4) = 0$
$w(e_5) = 0$

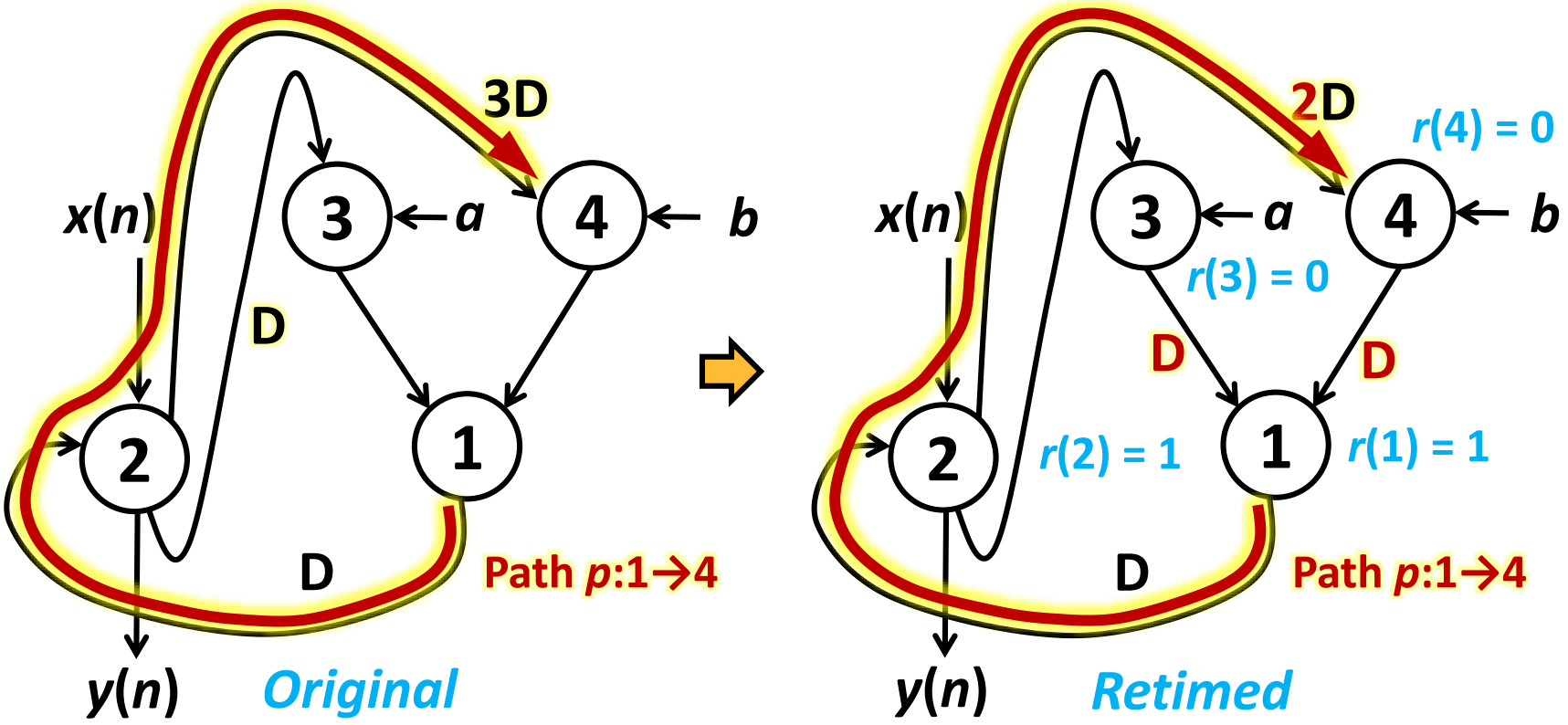
Retiming equation

$$w_r(e) = w(e) + r(v) - r(u)$$

[2] R. Nanda, DSP Architecture Optimization in MATLAB/Simulink Environment, M.S. Thesis, University of California, Los Angeles, 2008.

Path Retiming

- **# of registers inserted in a path *p:* $v_1 \rightarrow v_2$ given by $r(v_2) - r(v_1)$**
  - If $r(v_2) - r(v_1) > 0$, registers added to the path
  - If $r(v_2) - r(v_1) < 0$, registers removed from the path

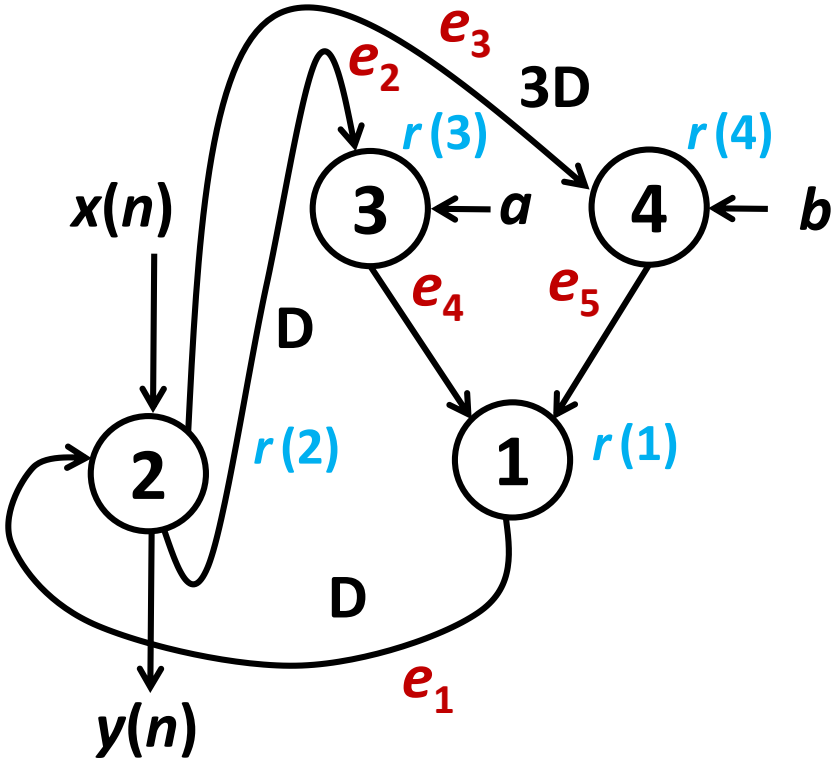


$w_r(p) = w(p) + r(4) - r(1) = 4 - 1$ (one register removed from path p)

Optimization Methods:
Scheduling & Retiming

Mathematical Modeling

- **Feasible retiming solution for $r(v_i)$ must ensure**
  - Non-negative edge weights $w_r(e)$
  - Integer values of $r(v)$ and $w_r(e)$



Feasibility constraints
$w_r(e_1) = w(e_1) + r(2) - r(1) \geq 0$
$w_r(e_2) = w(e_2) + r(3) - r(2) \geq 0$
$w_r(e_3) = w(e_3) + r(4) - r(2) \geq 0$
$w_r(e_4) = w(e_4) + r(1) - r(3) \geq 0$
$w_r(e_5) = w(e_5) + r(1) - r(4) \geq 0$

Integer solutions to feasibility constraints constitute a retiming solution

Retiming with Timing Constraints

- **Find retiming solution which guarantees critical path in DFG ≤ *T***
  - Paths with logic delay > ***T*** must have at least one register
- **Define**
  - ***W*(*u*,*v*)**: minimum number of registers over all paths b/w nodes ***u*** and ***v*, *min* {*w*(*p*) | *p* : *u* → *v*}**
  - If no path exists between the vertices, then ***W*(*u*,*v*) = 0**
  - ***Ld*(*u*,*v*):** maximum logic delay over all paths b/w nodes ***u*** and ***v***
  - If no path exists between vertices ***u*** and ***v*** then ***Ld*(*u*,*v*) = −1**
- **Constraints**
  - Non-negative weights for all edges, $W_r(v_i, v_j) \geq 0, \forall$ i,j
  - Look for nodes **(*u*,*v*)** with ***Ld*(*u*,*v*) > *T***
  - Define inequality constraint $W_r(u,v) \geq 1$ for such nodes

Leiserson-Saxe Algorithm [1]

Algorithm for feasible retiming solution with timing constraints

$$\underline{Algorithm\ \{r(v_i), flag\} \leftarrow Retime(G,d,T)}$$

$k \leftarrow 1$

for $u = 1$ *to* $|V|$

  for $v = 1$ *to* $|V|$ *do*

    if $Ld(u,v) > T$ *then*

      Define inequality $I_k : W(u,v) + r(v) - r(u) \geq 1$

    else if $Ld(u,v) > -1$ *then*

      Define inequality $I_k : W(u,v) + r(v) - r(u) \geq 0$

    endif

    $k \leftarrow k + 1$

  endfor

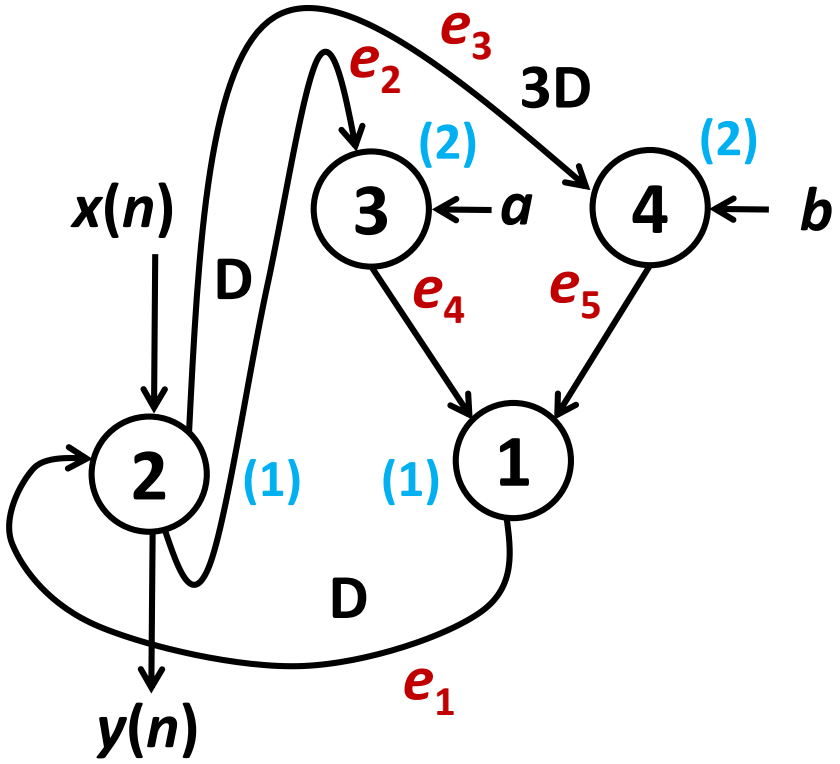
endfor

[1] C. Leiserson and J. Saxe, "Optimizing synchronous circuitry using retiming," *Algorithmica,* vol. 2, no. 3, pp. 211-216, 1991.

[2] R. Nanda, DSP Architecture Optimization in MATLAB/Simulink Environment, M.S. Thesis, University of California, Los Angeles, 2008.

Use Bellman-Ford algorithm to solve the inequalities I_k [2]

Retiming with Timing Constraints



Feasibility + Timing constraints

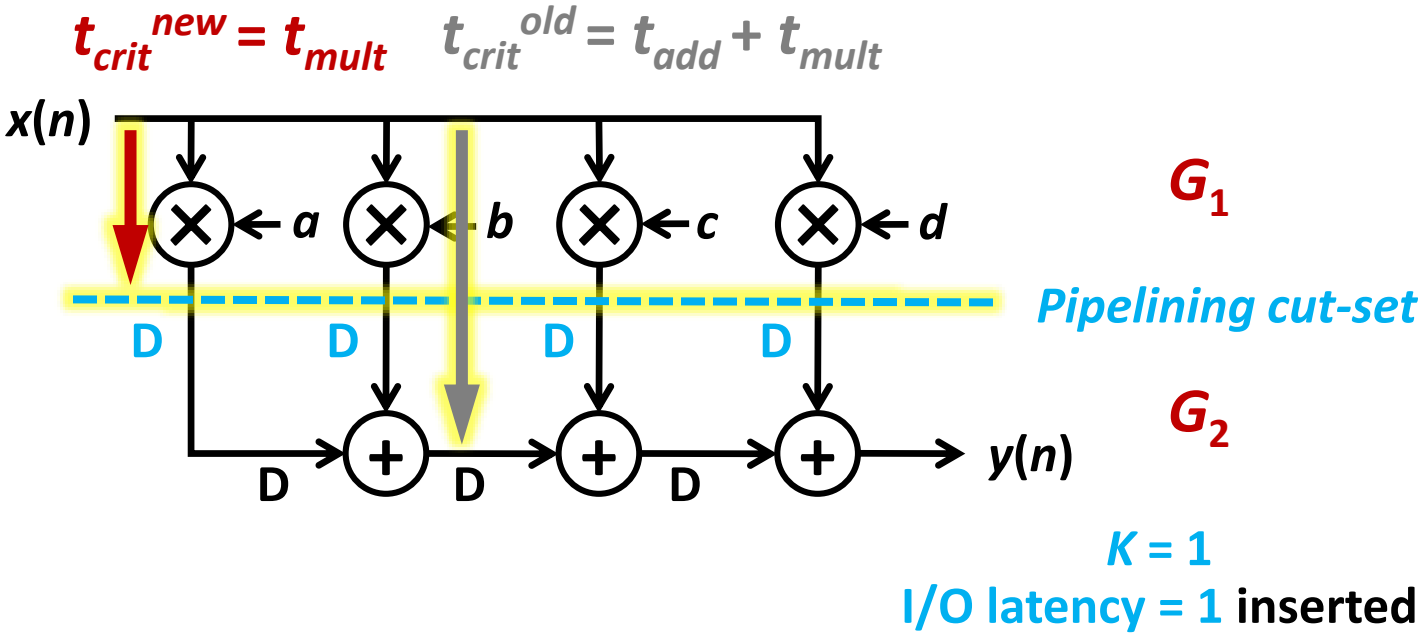
T = 2 time units

$W(1,2) + r(2) - r(1) \geq 0,\ W(1,2) = 1$
$W(2,1) + r(1) - r(2) \geq 1,\ W(2,1) = 1$
$W(4,2) + r(2) - r(4) \geq 1,\ W(4,2) = 1$
$W(2,4) + r(4) - r(2) \geq 1,\ W(2,4) = 3$
$W(4,1) + r(1) - r(4) \geq 1,\ W(4,1) = 0$
$W(1,4) + r(4) - r(1) \geq 1,\ W(1,4) = 4$
$W(3,1) + r(1) - r(3) \geq 1,\ W(3,1) = 0$
$W(1,3) + r(3) - r(1) \geq 1,\ W(1,3) = 2$
$W(4,3) + r(3) - r(4) \geq 1,\ W(4,3) = 2$
$W(3,4) + r(4) - r(3) \geq 1,\ W(3,4) = 4$
$W(2,3) + r(3) - r(2) \geq 1,\ W(2,3) = 1$
$W(3,2) + r(2) - r(3) \geq 1,\ W(3,2) = 1$

Integer solutions to these constraints constitute a retiming solution

Pipelining

- **Special case of retiming**
  - Small functional change with additional I/O latency
  - Insert *K* delays at cut-sets, all cut-set edges uni-directional
  - Exploits additional latency to minimize critical path

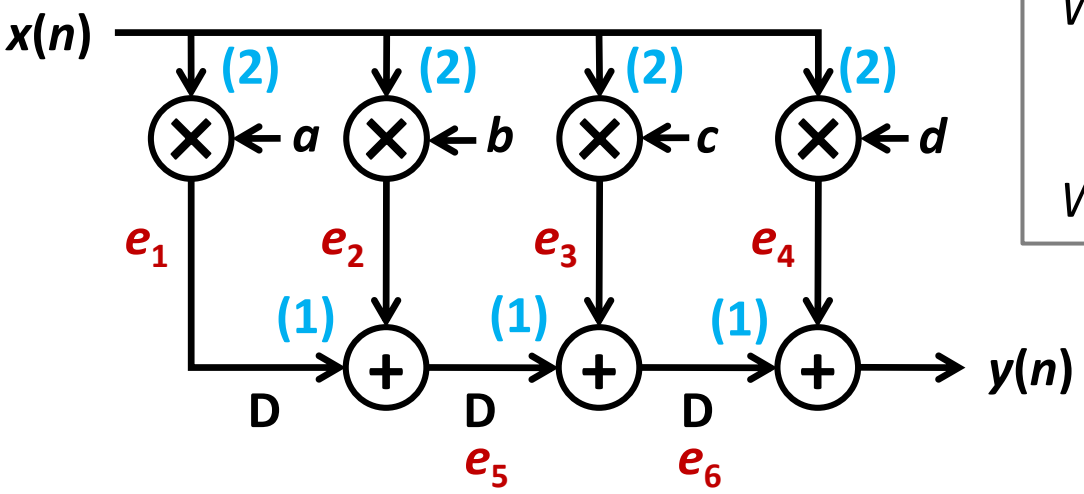


Modeling Pipelining

- **Same model as retiming with timing constraints**
- **Additional constraints to limit the added I/O latency**
  - Latency inserted b/w input node v_1 and output node v_2 is given by difference between retiming weights, $r(v_2) - r(v_1)$

$t_{critical,desired}$ = 2 time units

Max additional I/0 latency = 1



(logic delay) of the nodes

Feasibility + Timing constraints

$W_r(1,5) = W(1,5) + r(5) - r(1) \geq 1$

$W_r(1,6) = W(1,6) + r(6) - r(1) \geq 1$

$\vdots$

$W_r(4,7) = W(4,7) + r(7) - r(4) \geq 1$

$r(7) - r(4) \leq 1$

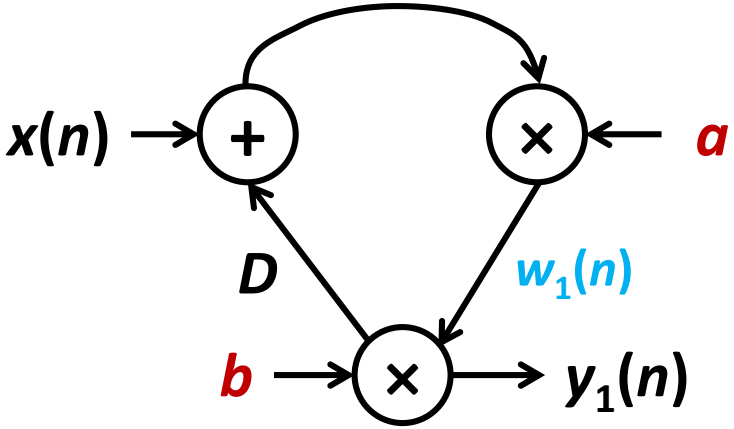
$r(7) - r(3) \leq 1$

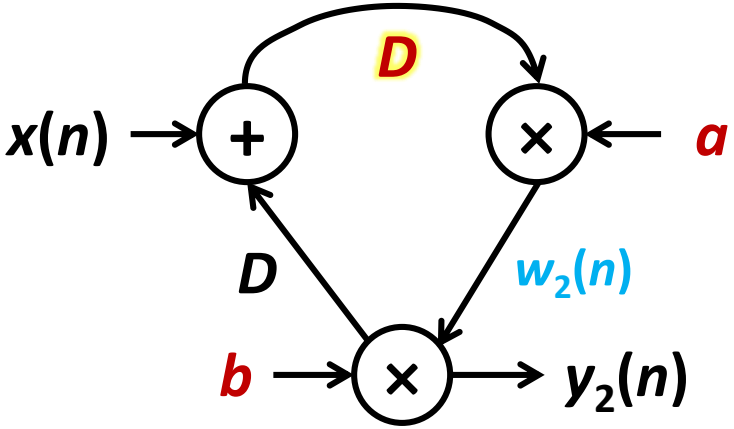
$r(7) - r(2) \leq 1$

$r(7) - r(1) \leq 1$

Recursive-Loop Bottlenecks

- **Pipelining loops not possible**
  - # registers in feedback loops must remain fixed





$w_1(n) = a \cdot (y_1(n-1) + x(n))$

$y_1(n) = b \cdot a \cdot y_1(n-1) + b \cdot a \cdot x(n)$

$w(n) = a \cdot (y_2(n-2) + x(n-1))$

$y_2(n) = b \cdot a \cdot y_2(n-2) + b \cdot a \cdot x(n-1)$

$y_1(n) \neq y_2(n)$

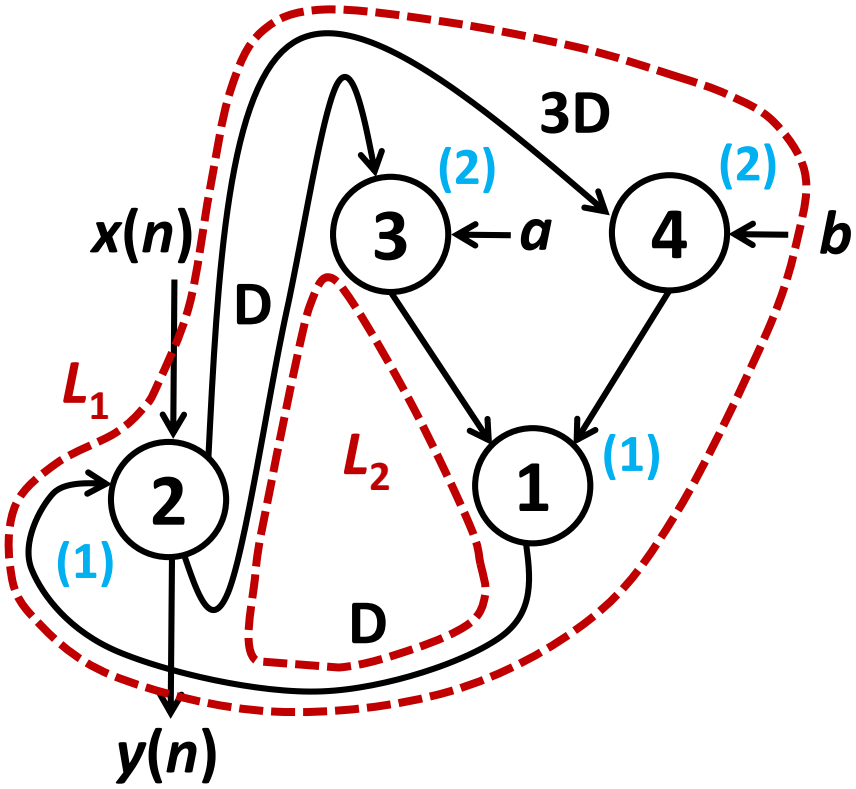
Changing the # delays in a loop alters functionality

Iteration Bound = Max{Loop Bound}

- **Loops limit the maximum achievable throughput**
  - Achieved when registers in a loop balance the logic delay

$$\frac{1}{f_{max}} = \max_{\text{all loops}} \left\{ \frac{\text{Combinational delay of loop}}{\text{Number of registers in loop}} \right\}$$

Loop bound



Loop L_1: 2 → 4 → 1

Loop L_2: 3 → 1 → 2

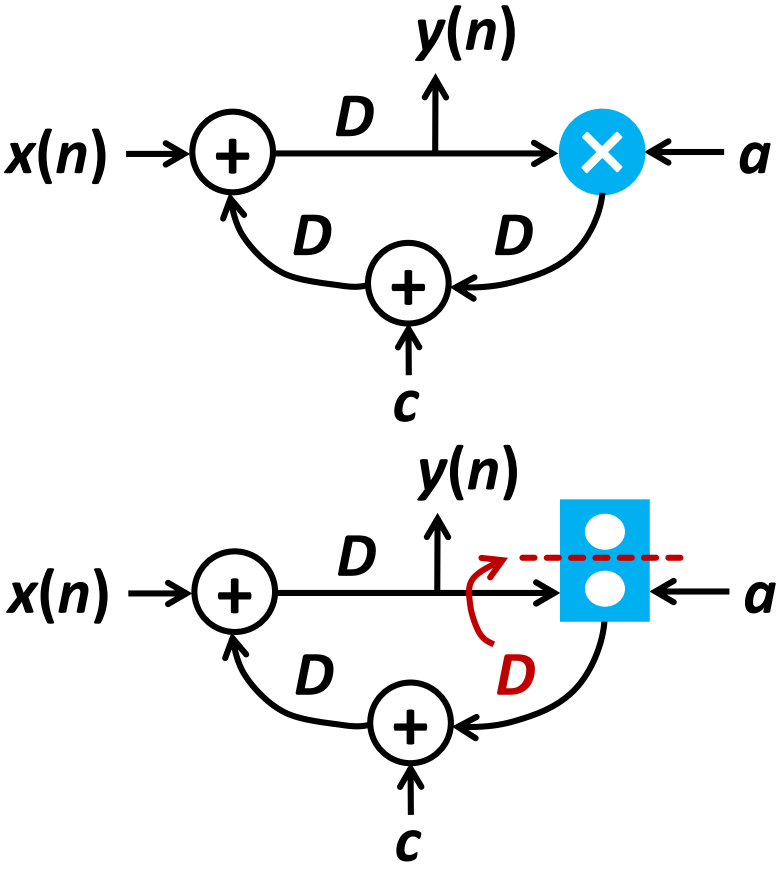
Loop bound $L_1 = \frac{4}{4} = 1$

Loop bound $L_2 = \frac{4}{2} = 2$

IB = 2 time units

Fine-Grain Pipelining

- **Achieving the iteration bound**
  - Requires finer level of granularity of operations



$t_{critical} = t_{mult}$

$t_{critical} = t_{mult}/2 + t_{add}$

Gate-level granularity can be achieved during logic synthesis

Parallelism [2]

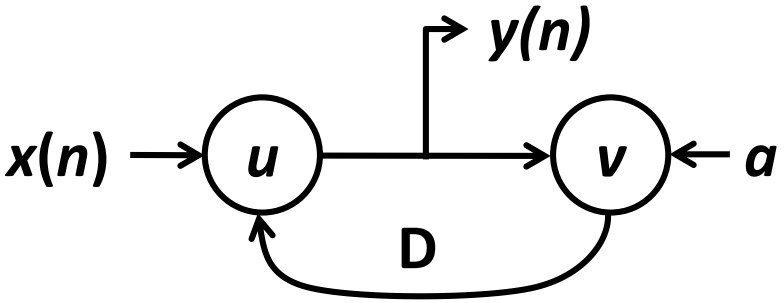
- **Unfolding of the operations in a flow-graph**
  - Parallelizes the flow-graph
  - Higher speed, lower power via V_{DD} scaling
  - Larger area
- **Describes multiple iterations of the DFG signal flow**
  - Symbolize the multiple number of iterations by P
  - Unfolded DFG constructed from the following P equations: $y_i = y(Pm + i), i \in \{0, 1, \ldots, P - 1\}$
  - DFG takes the inputs $x(Pm), x(Pm + 1), \ldots, x(Pm + P - 1)$
  - Outputs are $y(Pm), y(Pm + 1), \ldots, y(Pm + P - 1)$

[2] K.K. Parhi, VLSI Digital Signal Processing Systems: Design and Implementation, John Wiley & Sons Inc., 1999.

Unfolding

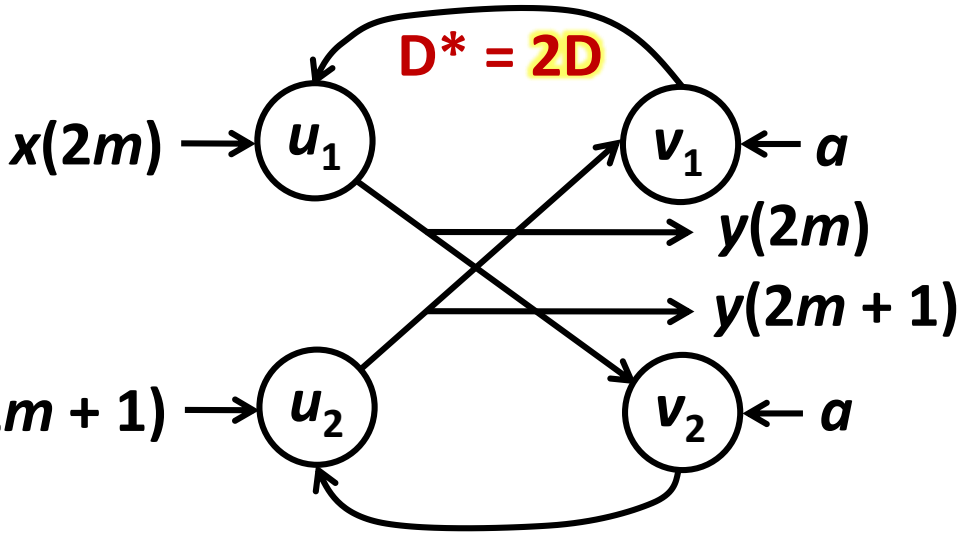
- **To construct *P*-unfolded DFG**
  - Draw *P* copies of all the nodes in the original DFG
  - The *P* input nodes take in values $x(Pm), \ldots, x(Pm + P - 1)$
  - Connect the nodes based on precedence constraints of DFG
  - Each delay in unfolded DFG is ***P*-slow**
  - Tap outputs $x(Pm), \ldots, x(Pm + P - 1)$ from the *P* output nodes

Original



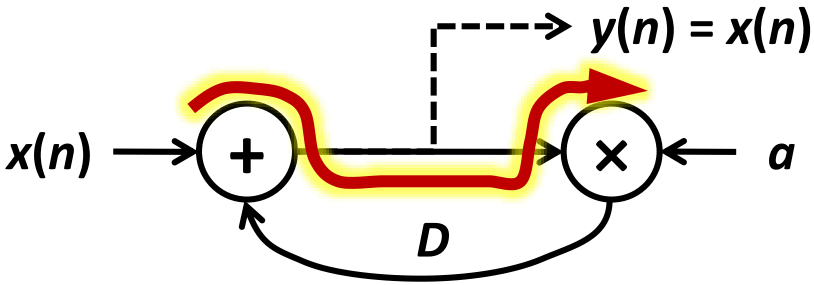
$$y(2m) = a \cdot y(2m - 1) + x(2m)$$
$$y(2m + 1) = a \cdot y(2m) + x(2m + 1)$$

Unfolded with *P* = 2

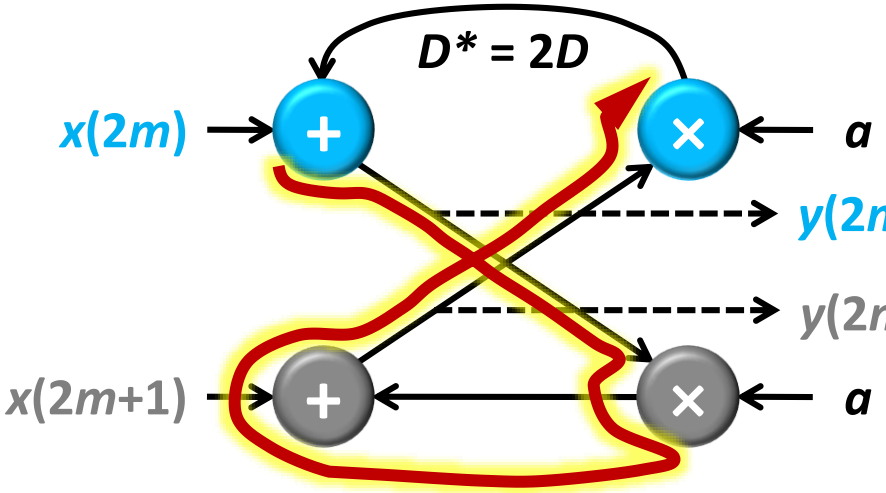


Unfolding IIR for Constant Throughput

- Maximum throughput limited by iteration bound (IB)
- Unfolding does not help if IB is already achieved



Critical path = $t_{add} + t_{mult}$

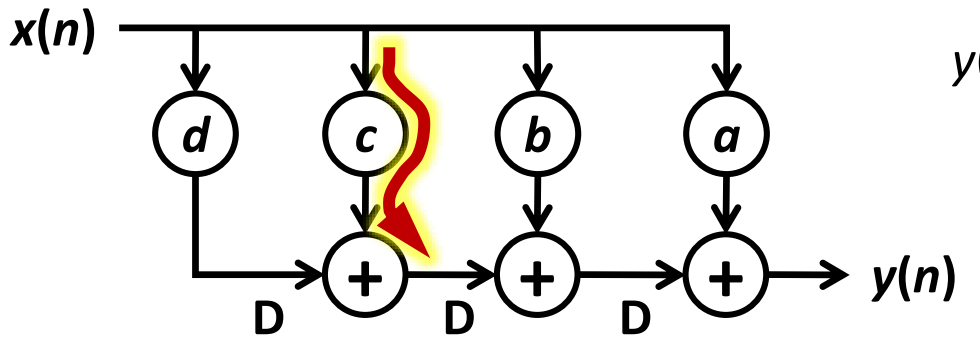


Critical path = $2t_{add} + 2t_{mult}$

Per iteration: $t_{add} + t_{mult}$

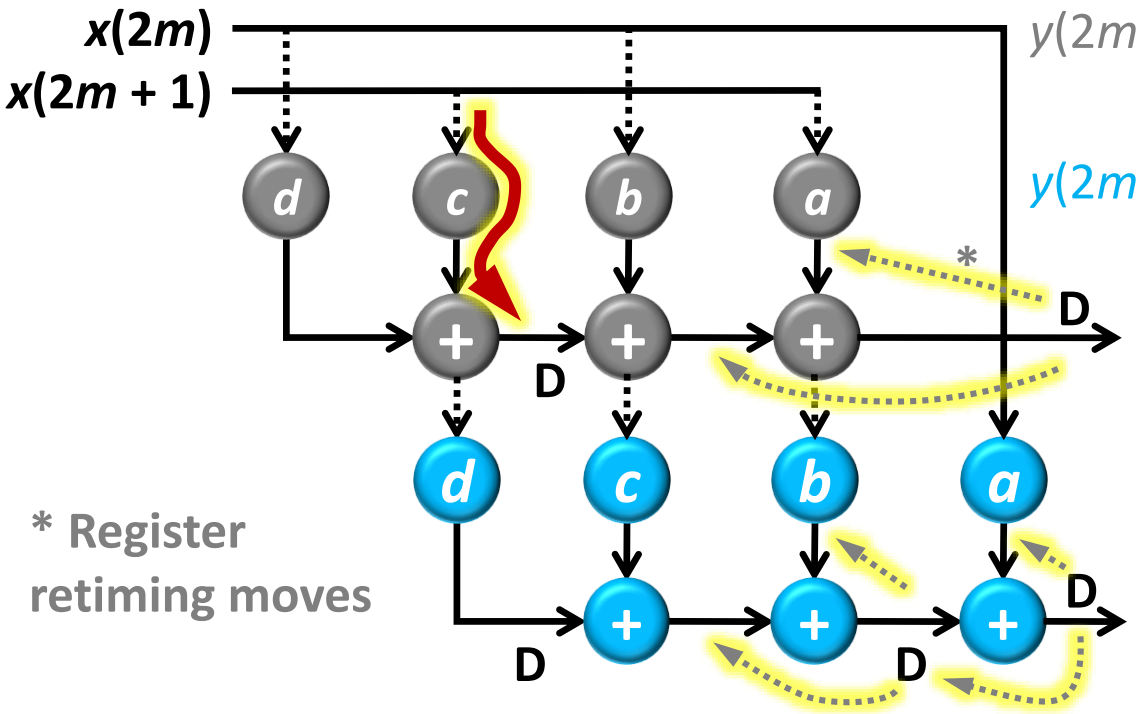
Unfolding FIR for Higher Throughput

- **Throughput can be increased with effective pipelining**



$$y(n) = a \cdot x(n) + b \cdot x(n-1) + c \cdot x(n-2) + d \cdot x(n-3)$$

$$t_{critical} = t_{add} + t_{mult}$$



$$y(2m-1) = a \cdot x(2m-1) + b \cdot x(2m-2) + c \cdot x(2m-3) + d \cdot x(2m-4)$$

$$y(2m-2) = a \cdot x(2m-2) + b \cdot x(2m-3) + c \cdot x(2m-4) + d \cdot x(2m-5)$$

$$t_{critical} = t_{add} + t_{mult}$$

$$t_{critical/iter} = t_{critical} / 2$$

* Register retiming moves

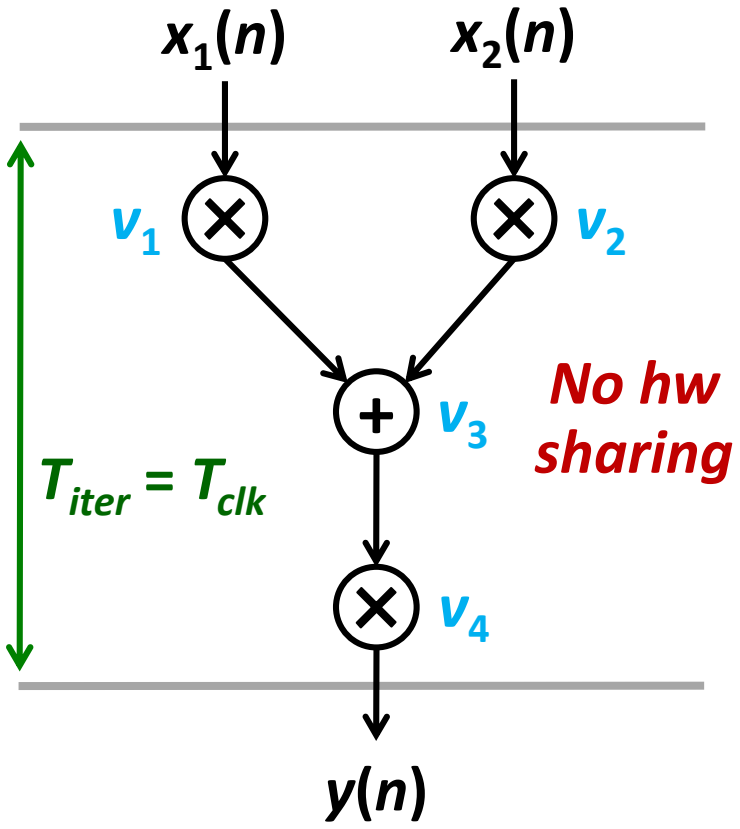
Throughput doubles!!

Introduction to Scheduling

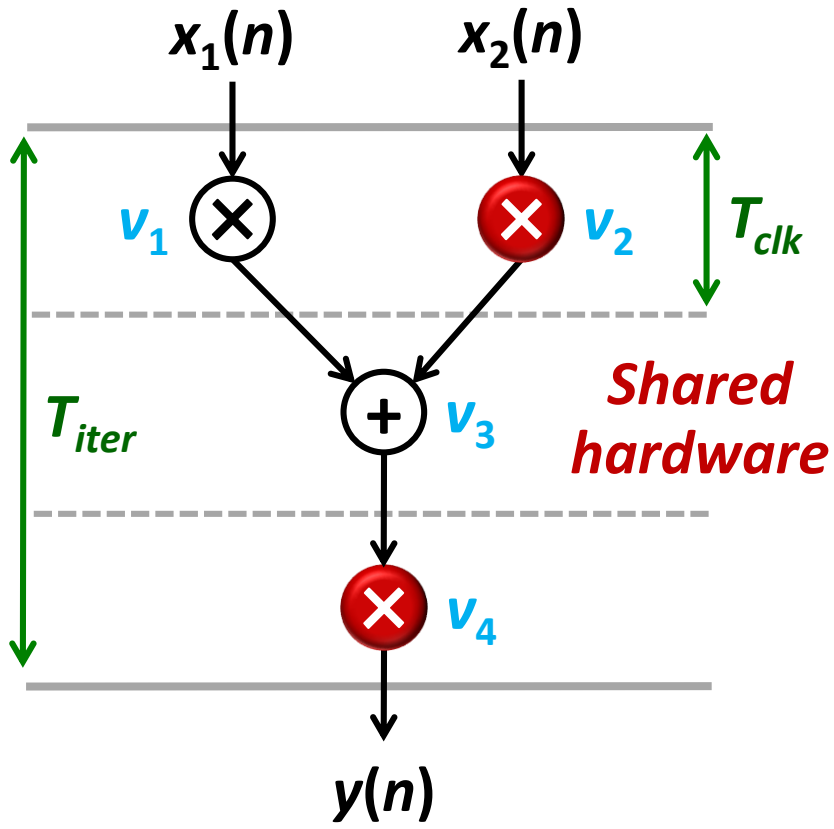
- **Dictionary definition**
  - The coordination of multiple related tasks into a time sequence
  - To solve the problem of satisfying time and resource constraints between a number of tasks
- **Data-flow-graph scheduling**
  - Data-flow-graph iteration
    - Execute all operations in a sequence
    - Sequence defined by the signal flow in the graph
  - One iteration has a finite time of execution T_{iter}
  - Constraints on T_{iter} given by throughput requirement
  - If required T_{iter} is long
    - T_{iter} can be split into several smaller clock cycles
    - Operations can be executed in these cycles
    - Operations executing in different cycles can share hardware

Area-Throughput Tradeoff

- **Scheduling provides a means to tradeoff throughput for area**
  - If $T_{iter} = T_{clk}$ all operations required dedicated hardware units
  - If $T_{iter} = N \cdot T_{clk}$, $N > 1$, operations can share hardware units



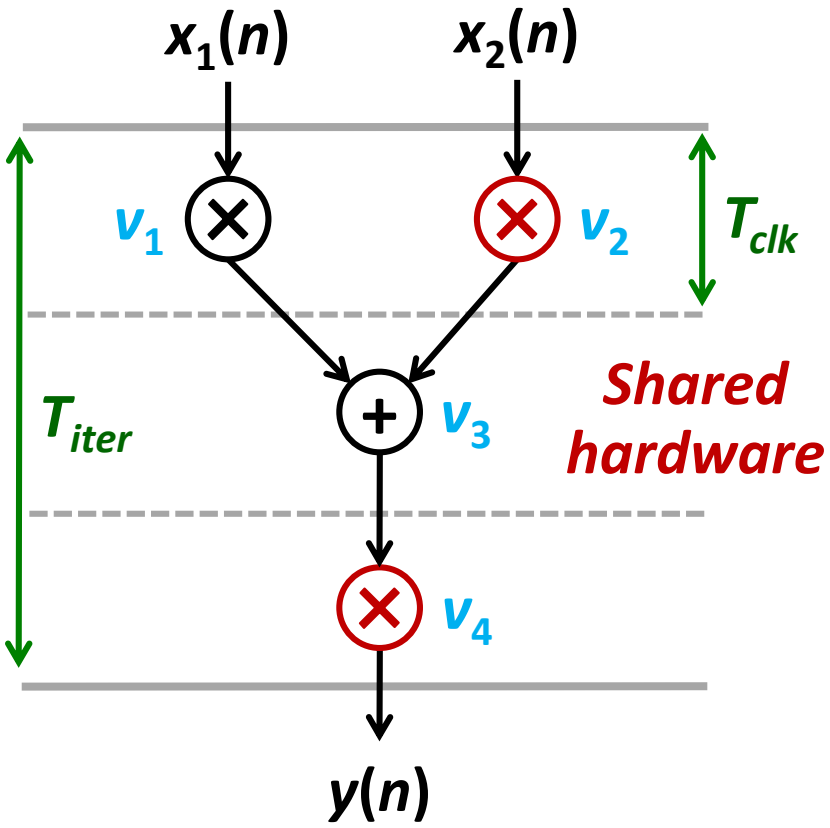
3 multipliers and 1 adder



2 multipliers and 1 adder

Schedule Assignment

- **Available: hardware units *H* and *N* clock cycles for execution**
  - For each operation, schedule table records
    - Assignment of hardware unit for execution, $H(v_i)$
    - Assignment of time of execution, $p(v_i)$



Schedule Table

Schedule	Add 1	Mult 1	Mult 2
Cycle 1	x	v_1	v_2
Cycle 2	v_3	x	x
Cycle 3	x	x	v_4

$H(v_1)$ = Multiplier 1
$H(v_2)$ = Multiplier 2
$H(v_3)$ = Adder 1
$H(v_4)$ = Multiplier 1

$p(v_1) = 1$
$p(v_2) = 1$
$p(v_3) = 2$
$p(v_4) = 3$

Problem Statement

- **Given a data-flow graph G, T_{iter} and T_{clk}**
  - Find a schedule assignment $H(v_i)$, $p(v_i)$ which:
    - Executes all DFG operations in N clock cycles
    - Sequence of execution should not alter DFG functionality
    - Minimizes the area A of the hardware resources required for execution

$$\min A = N_a \cdot Area_{adder} + N_m \cdot Area_{multiplier}$$

Schedule	Add 1	Mult 1	Mult 2
Cycle 1	x	v_1	v_2
Cycle 2	v_3	x	x
Cycle 3	x	x	v_4

Number of adders: N_a
Number of multipliers: N_m

v_1, v_2, v_3 executed in N = 3 cycles

$$A = 1 \cdot Area_{adder} + 2 \cdot Area_{multiplier}$$

ASAP: As Soon As Possible Scheduling [4]

$Algorithm\ \{H(v_i), p(v_i)\} \leftarrow ASAP(G)$

$u \leftarrow v_i$ // v_i *is any "ready" operation, operation is "ready"*
// *if all its preceding operations have been scheduled*

$q_i \in V \leftarrow$ *operations immediately preceding* u

$e_i \leftarrow$ *execution of* q_i *ends in this cycle*

$S_{\min} \leftarrow$ *first available cycle for execution of* $u = \max\{e_i + 1\}$

$S \leftarrow$ *first available cycle* $\geq S_{\min}$ *with*
available hardware resource H_i

$H(u) \leftarrow H_i$

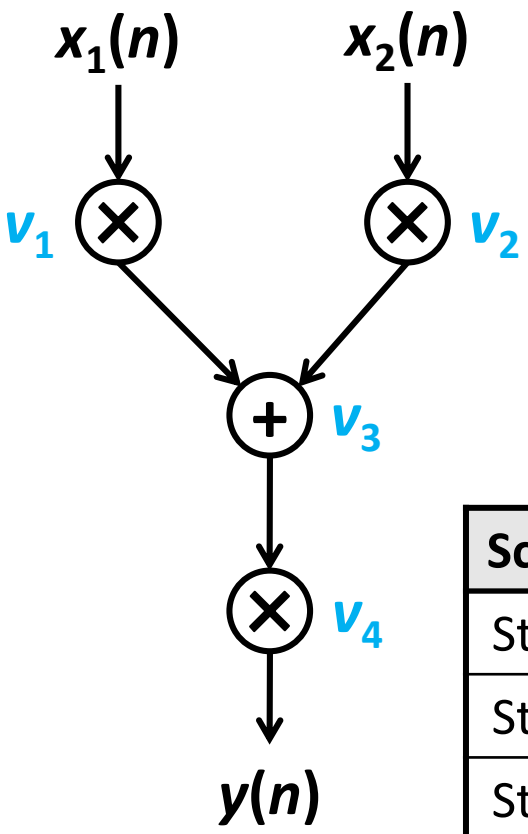
$p(u) \leftarrow S$

[4] C. Tseng and D.P. Siewiorek, "Automated synthesis of datapaths in digital systems," *IEEE Trans. Computer-Aided Design,* vol. CAD-5, no. 3, pp. 379-395, July 1986.

- **Schedules the operations top-down from input to output nodes**
- **Available hardware resource units specified by the user**
- **Operation scheduled in the first available cycle**

Example 11.2a: ASAP Scheduling

Graph *G*



- **Assumptions:**
  - $T_{iter} = 4 \cdot T_{clk}$, $N = 4$
  - Multiplier pipeline: 1
  - Adder pipeline: 1
  - Available hardware
    - 1 multiplier M_1
    - 1 adder A_1

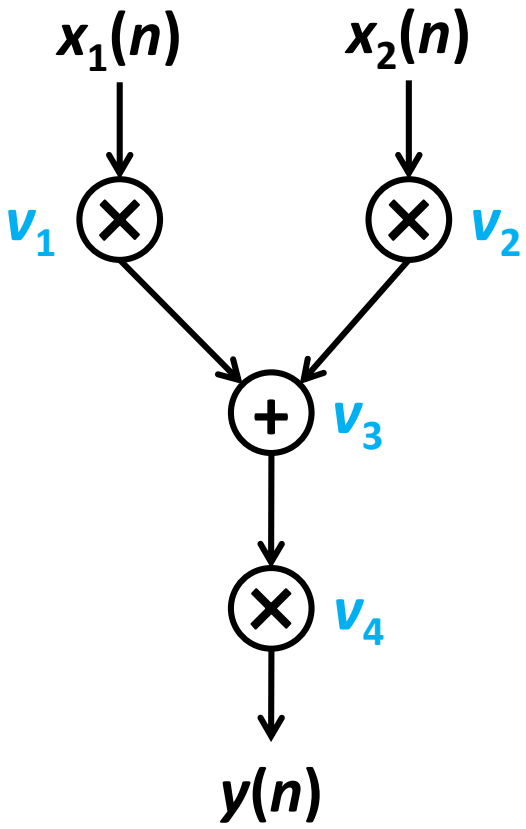
ASAP scheduling steps

Sched.	u	q	e	S_{min}	S	$p(u)$	$H(u)$
Step 1	v_1	null	0	1	1	1	M_1
Step 2	v_2	null	0	1	2	2	M_1
Step 3	v_3	v_1 , v_2	1	3	3	3	A_1
Step 4	v_4	v_3	3	4	4	4	M_1

Example 11.2b: ASAP Scheduling

- **Schedules "ready" operations in the first cycle with available resource**

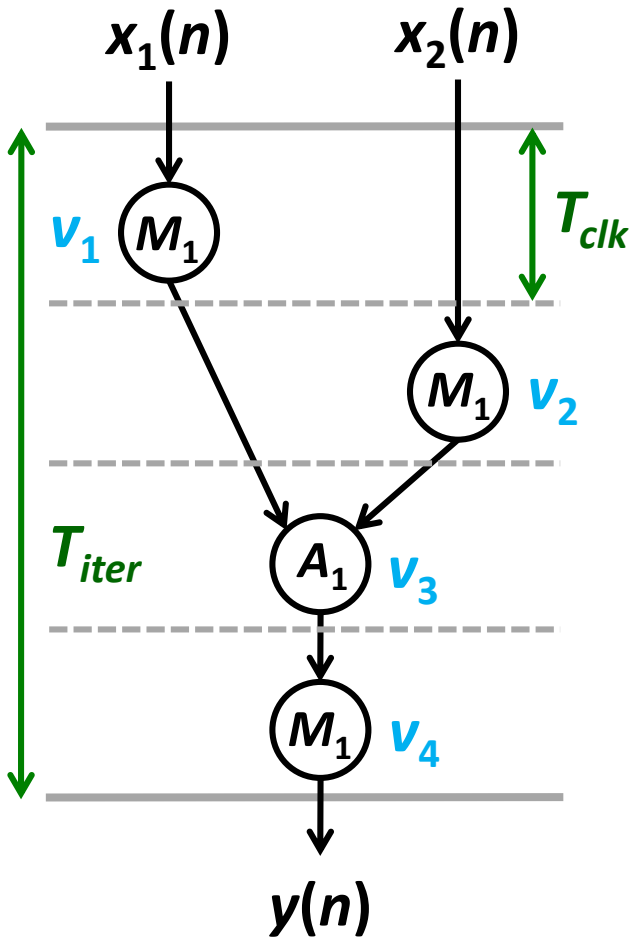
Graph *G*



Schedule Table

Schedule	M_1	A_1
Cycle 1	v_1	x
Cycle 2	v_2	x
Cycle 3	x	v_3
Cycle 4	v_4	x

Final ASAP schedule

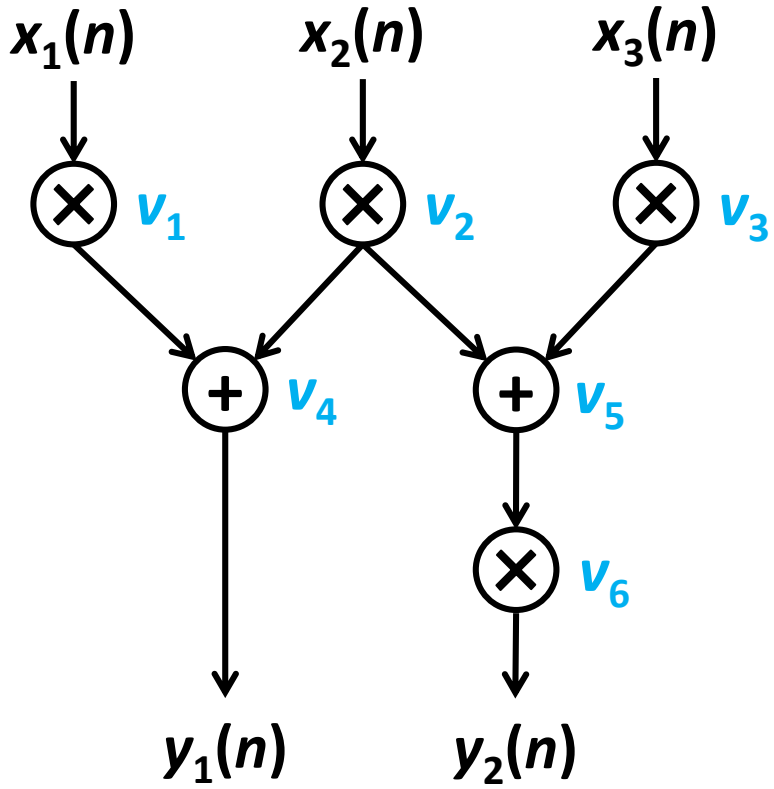


Scheduling Algorithms

- **More heuristics**
  - Heuristics vary in their selection of next operation to scheduled
  - This selection strongly determines the quality of the schedule
  - **ALAP:** As Late As Possible scheduling
    - Similar to ASAP except operations scheduled from output to input
    - Operation "ready" if all its succeeding operations scheduled
  - **ASAP, ALAP** do not give preference to timing-critical operations
    - Can result in timing violations for fixed set of resources
    - More resource/area required to meet the T_{iter} timing constraint
  - **List scheduling**
    - Selects the next operation to be scheduled from a list
    - The list orders the operations according to timing criticality

List Scheduling [5]

- **Assign precedence height $P_H(v_i)$ to each operation**
  - $P_H(v_i)$ = length of longest combinational path rooted by v_i
  - Schedule operations in descending order of precedence height



$t_{add} = 1, t_{mult} = 2$

$P_H(v_1) = T(v_4) = 1$
$P_H(v_2) = T(v_5) + T(v_6) = 3$
$P_H(v_3) = T(v_5) + T(v_6) = 3$
$P_H(v_5) = T(v_6) = 2$
$P_H(v_4) = 0, P_H(v_6) = 0$

Possible scheduling sequence	
ASAP	$v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow v_4 \rightarrow v_5 \rightarrow v_6$
LIST	$v_3 \rightarrow v_2 \rightarrow v_5 \rightarrow v_1 \rightarrow v_4 \rightarrow v_6$

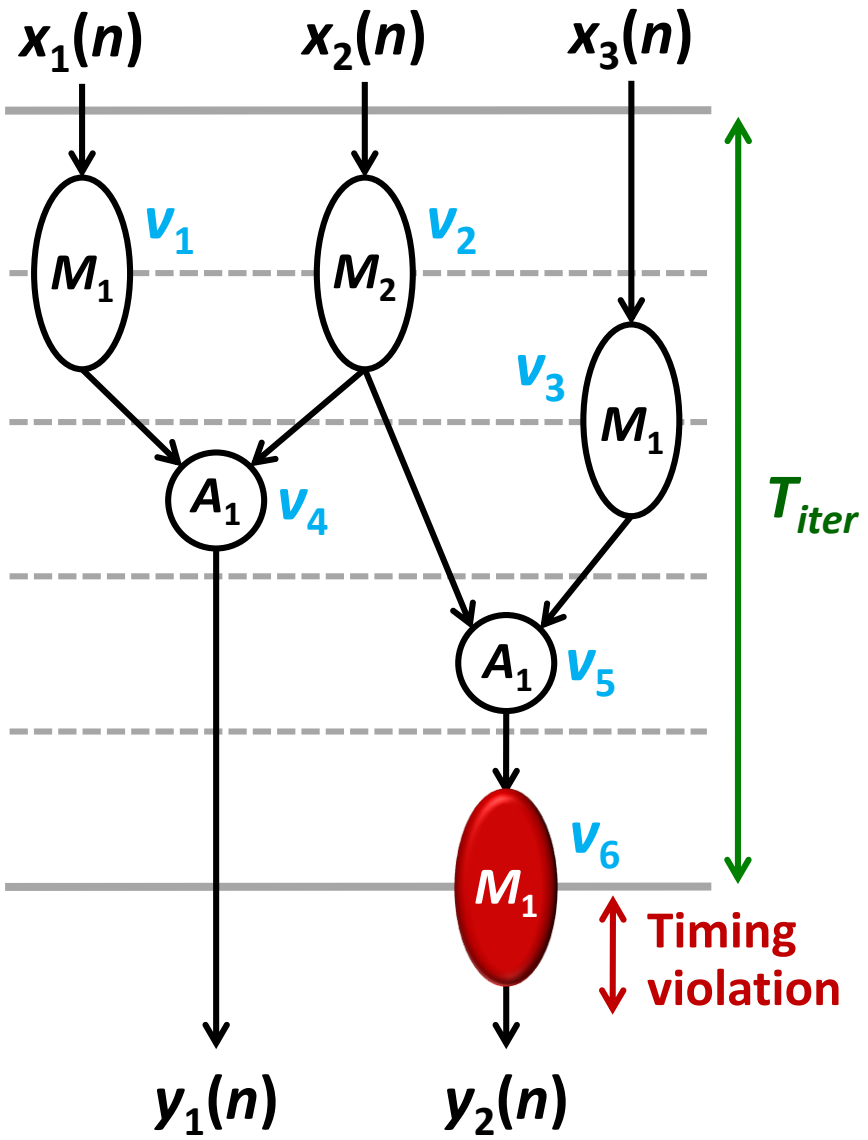
[5] S. Davidson *et. al.*, "Some experiments in local microcode compaction for horizontal machines," *IEEE Trans. Computers,* vol. C-30, no. 7, pp. 460-477, July 1981.

Comparing Scheduling Heuristics: ASAP

$T_{iter} = 5 \cdot T_{clk}$, $N = 5$

Pipeline depth
- Multiplier: 2
- Adder: 1

Available hardware
- 2 mult: M_1, M_2
- 1 add: A_1



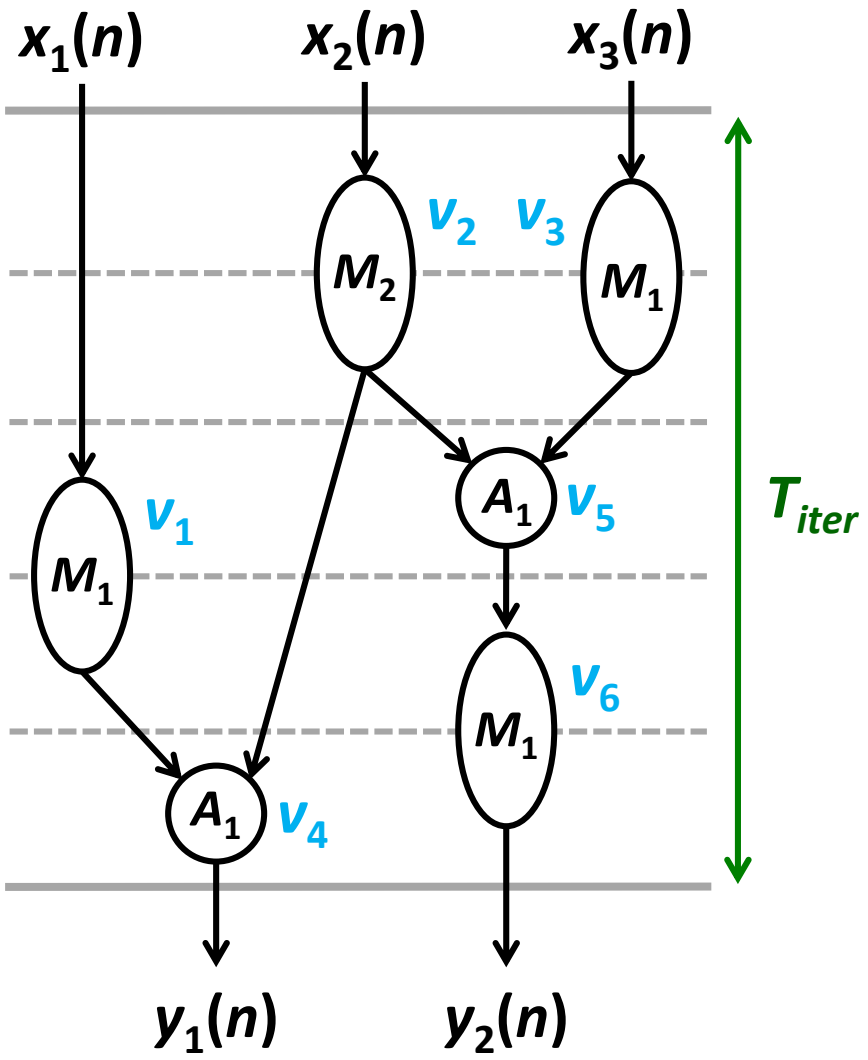
- **ASAP schedule infeasible**, more resources required to satisfy timing

Comparing Scheduling Heuristics: LIST

$T_{iter} = 5 \cdot T_{clk}$, $N = 5$

Pipeline depth
- Multiplier: 2
- Adder: 1

Available hardware
- 2 mult: M_1, M_2
- 1 add: A_1



- **LIST scheduling feasible**, with 1 adder and 2 multipliers in 5 time steps

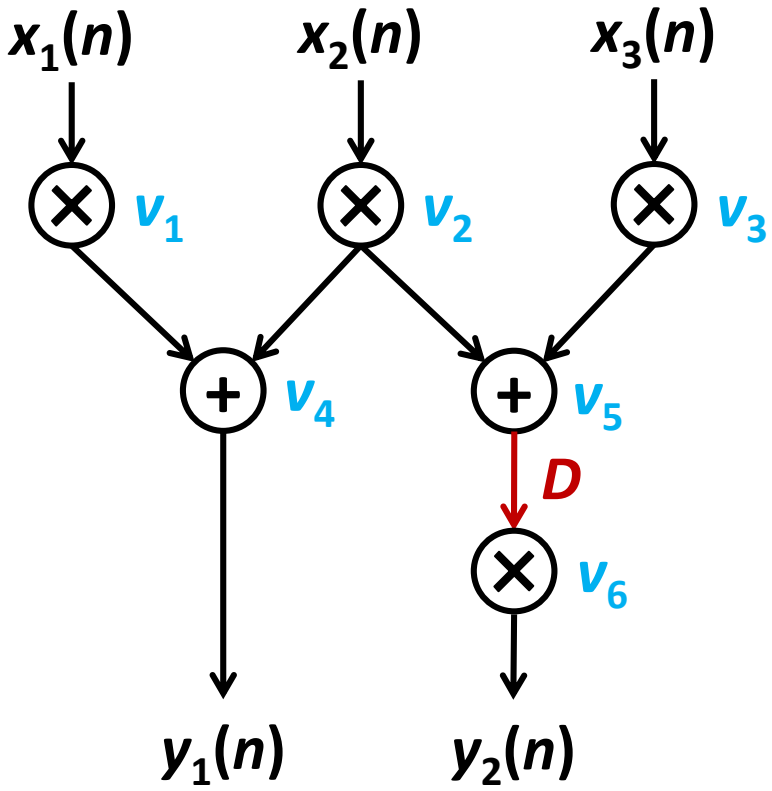
Inter-Iteration Edges: Timing Constraints

- **Edge e : $v_1 \rightarrow v_2$ with zero delay forces precedence constraints**
  - Result of operation v_1 is input to operation v_2 in an iteration
  - Execution of v_1 must precede the execution of v_2

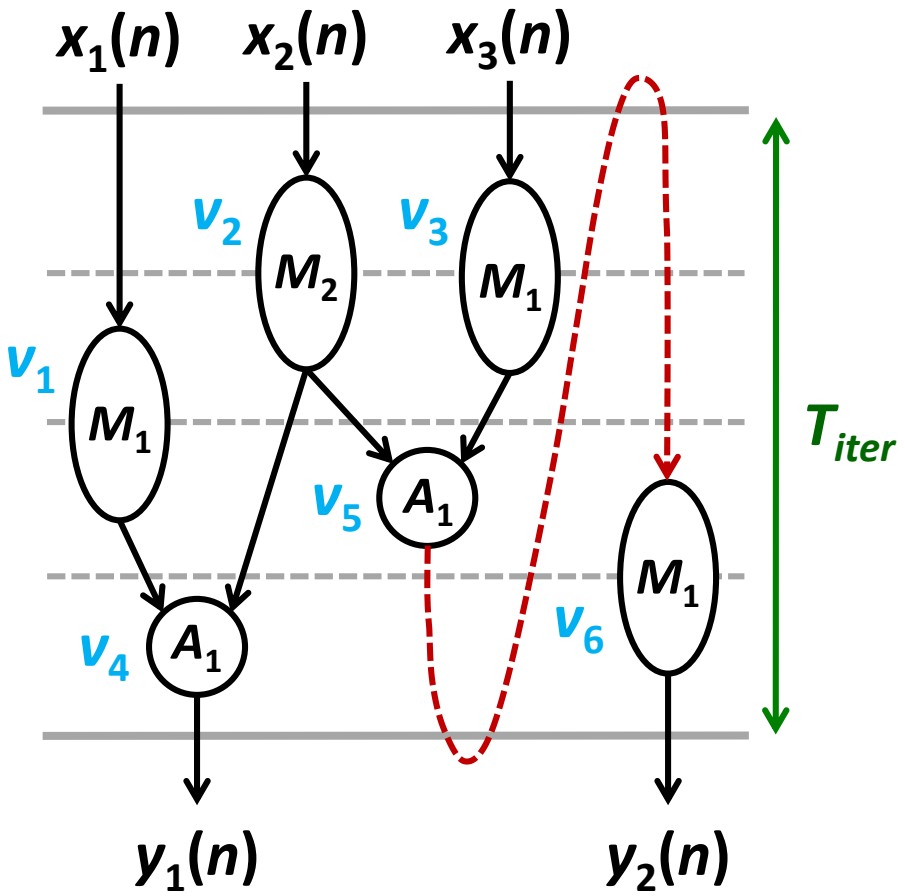
- **Edge e : $v_1 \rightarrow v_2$ with delays represent relaxed timing constraints**
  - If R delays present on edge e
  - Output of v_1 in I^{th} iteration is input to v_2 in $(I + R)^{th}$ iteration
  - v_1 not constrained to execute before v_2 in the I^{th} iteration

- **Delay insertion after scheduling**
  - Use folding equations to compute the number of delays/registers to be inserted on the edges after scheduling

Inter-Iteration Edges



Inter-iteration edge
$e : v_5 \rightarrow v_6$



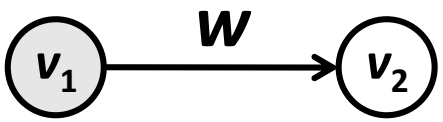
v_6 is not constrained to execute after v_5 in an iteration

Insert registers on edge *e* for correct operation

Folding

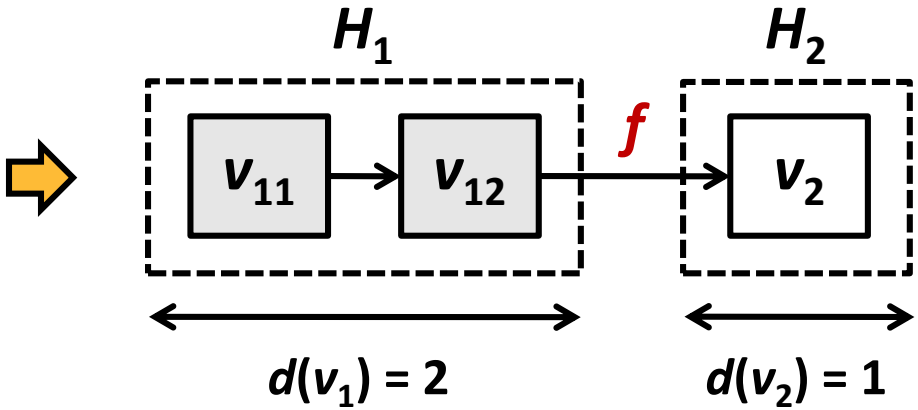
- **Maintain precedence constraints and functionality of DFG**
  - Route signals to hardware units at the correct time instances
  - Insert the correct number of registers on edges after scheduling

Original Edge



v_1 mapped to unit H_1
v_2 mapped to unit H_2
2 pipeline stages in H_1
1 pipeline stage in H_2

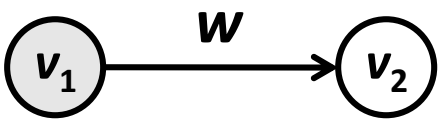
Scheduled Edge

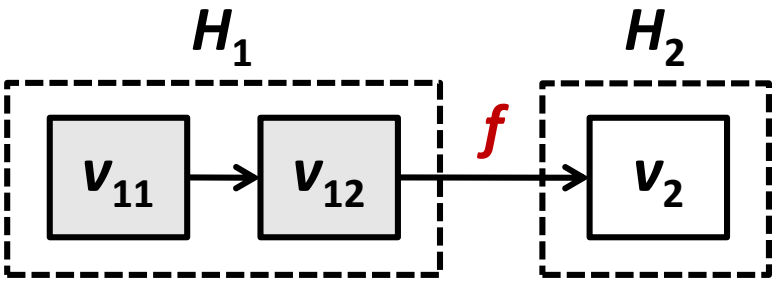


Compute value of f which maintains precedence

Folding Equation

- **Number of registers on edges after folding depends on**
  - Original number of delays ***w***, pipeline depth of source node
  - Relative time difference between execution of v_1 and v_2

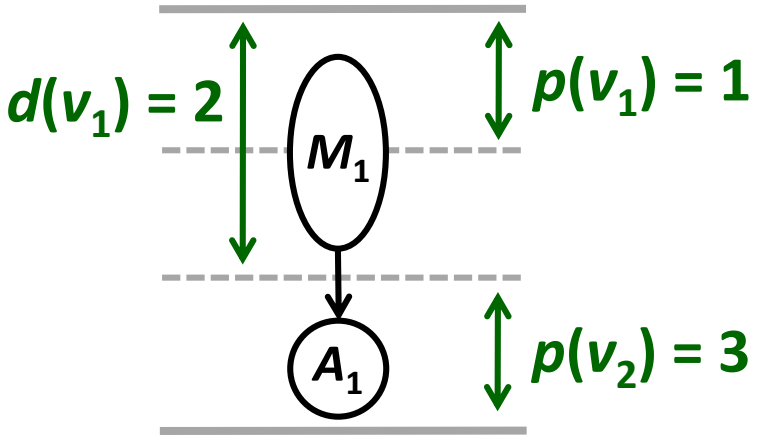




Legend ***d*: delay, *p*: schedule**

N clock cycles per iteration

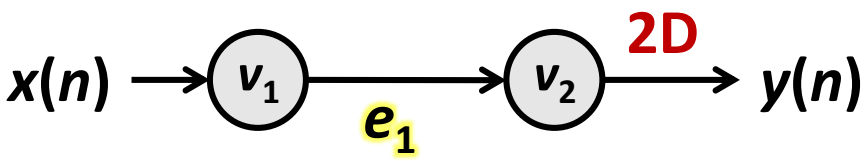
w delays → N·w delay in schedule



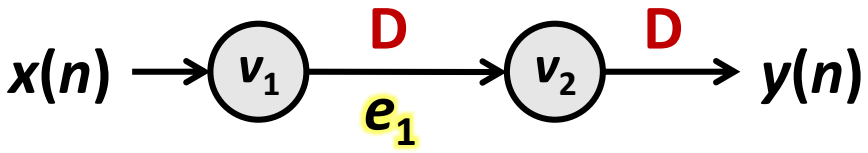
$$f = N \cdot w - d(v_1) + p(v_2) - p(v_1)$$

Example 11.3: Scheduling

- Edge scheduled using folding equations
- Folding factor (N) = 2
- Pipeline depth
  - $d(v_1) = 2$
  - $d(v_2) = 2$
- Schedule
  - $p(v_1) = 1$
  - $p(v_2) = 2$

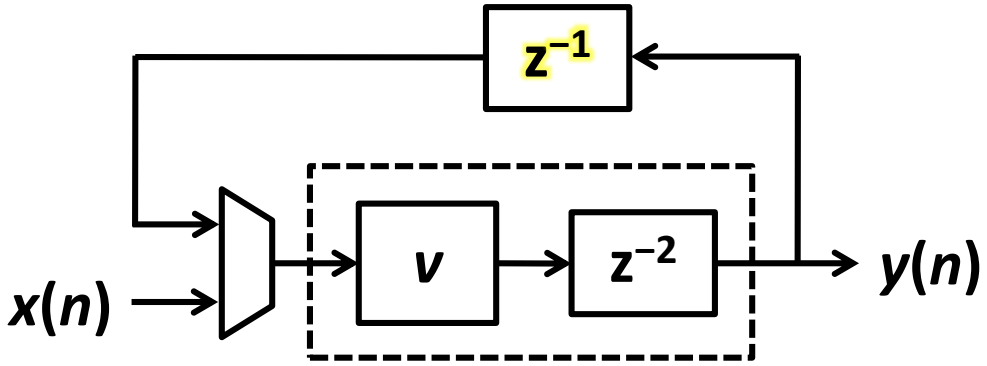


(a) Original edge



(b) Retimed edge

$f = N \cdot w - d(v_1) + p(v_2) - p(v_1)$ ⇨ $f = 2 \cdot 1 - 2 + 2 - 1 = 1$



(c) Retimed edge after scheduling

Efficient Retiming & Scheduling

- **Retiming with scheduling**
  - Additional degree of freedom associated with register movement results in less area or higher throughput schedules
- **Challenge: Retiming with scheduling**
  - Time complexity increases if retiming done with scheduling
- **Approach: Low-complexity retiming solution**
  - Pre-process data flow graph (DFG) prior to scheduling
  - Retiming algorithm converges quickly (polynomial time)
  - Time-multiplexed DSP designs can achieve faster throughput
  - Min-period retiming can result in reduced area as well
- **Result: Performance improvement**
  - An order of magnitude reduction in the worst-case time-complexity
  - Near-optimal solutions in most cases

Results: Area and Runtime

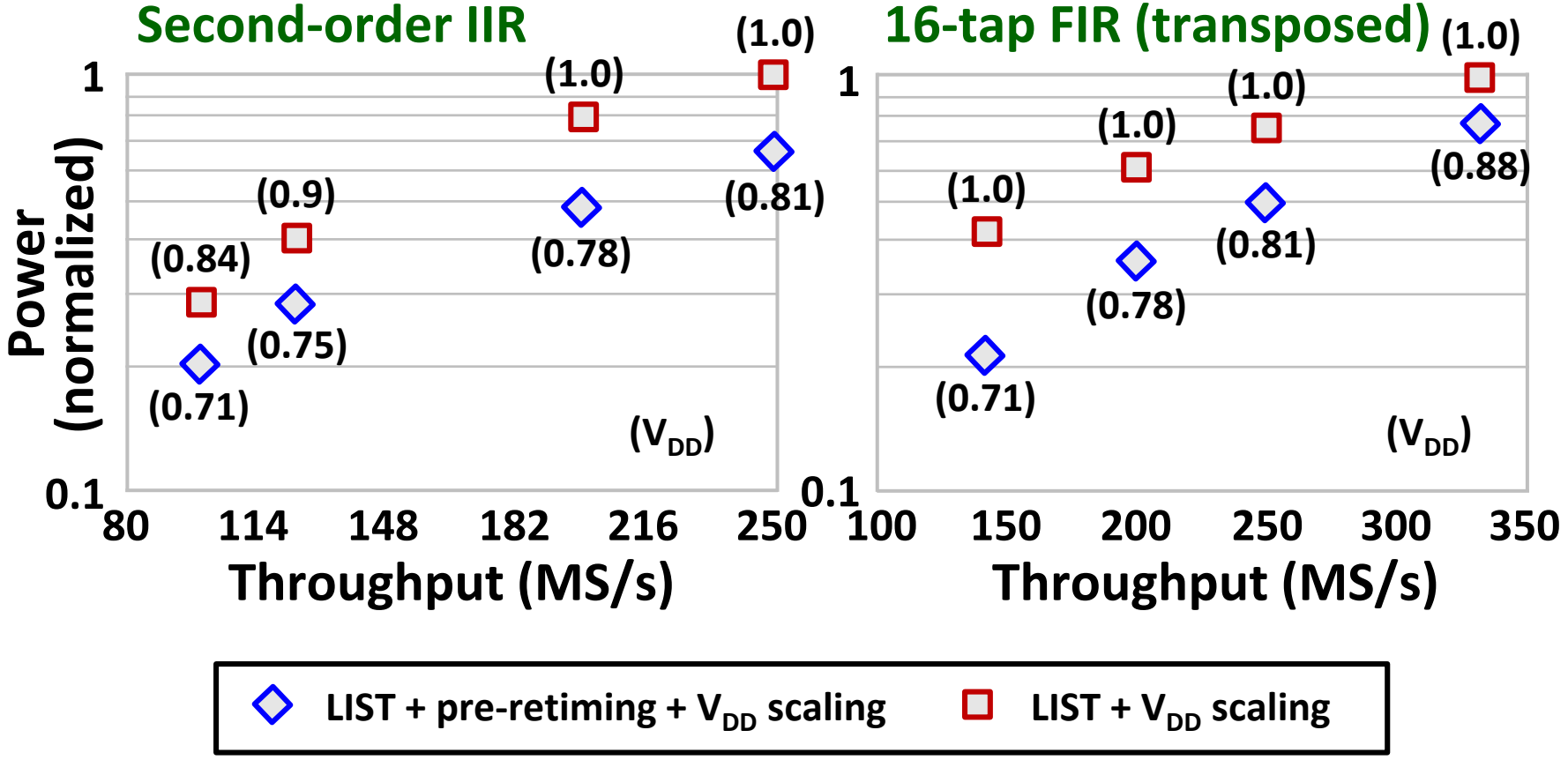
DSP Design	N	Scheduling (no retiming)		Scheduling (ILP) w/ BF retiming		Scheduling with pre-processed retiming	
		Area	CPU(s)	Area	CPU(s)	Area	CPU(s)
Wave filter	16	NA	NA	8	264	14	0.39
	17	13	0.20	7	777	8	0.73
Lattice filter	2	NA	NA	41	0.26	41	0.20
	4	NA	NA	23	0.30	23	0.28
8-point DCT	3	NA	NA	41	0.26	41	0.21
	4	NA	NA	28	0.40	28	0.39

NA – scheduling infeasible without retiming

Near-optimal solutions, significantly reduced runtime

Scheduling Comparison

- **Scheduling with pre-retiming outperforms scheduling**
  - Retiming before scheduling enables higher throughput
  - Lower power with V_{DD} scaling for same speed



Summary

- **DFG automation algorithms**
  - Retiming, pipelining
  - Scheduling
- **Simulink-based design optimization flow**
  - Parameterized architectural transformations
  - Optimized architecture available in Simulink
- **Energy, area, performance tradeoffs with**
  - Architectural optimizations
  - Carry-save arithmetic
  - Voltage scaling