EE-334
Digital System Design

Custom Digital Circuits

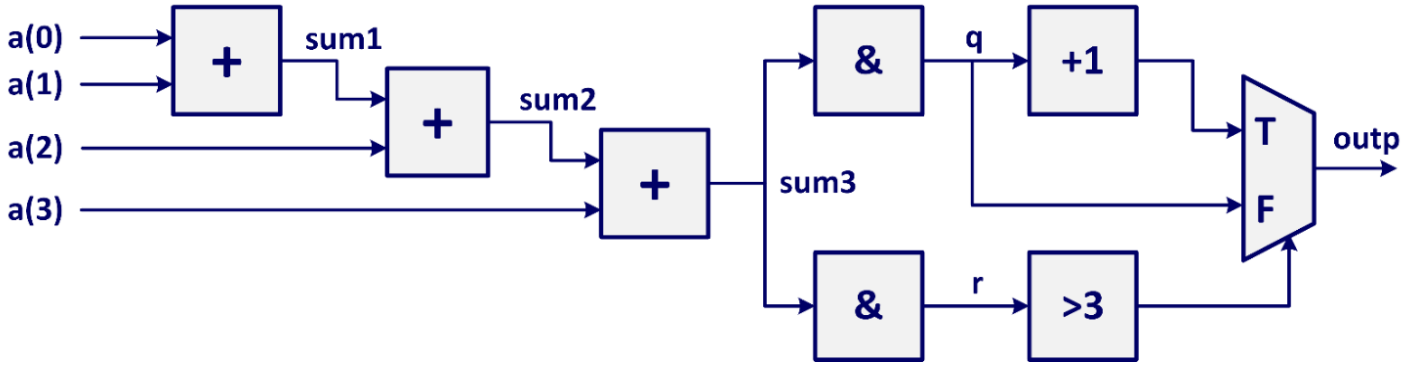
From Algorithms to Architectures

Andreas Burg
(Alain Vachoux)

Isomorphic Architecture

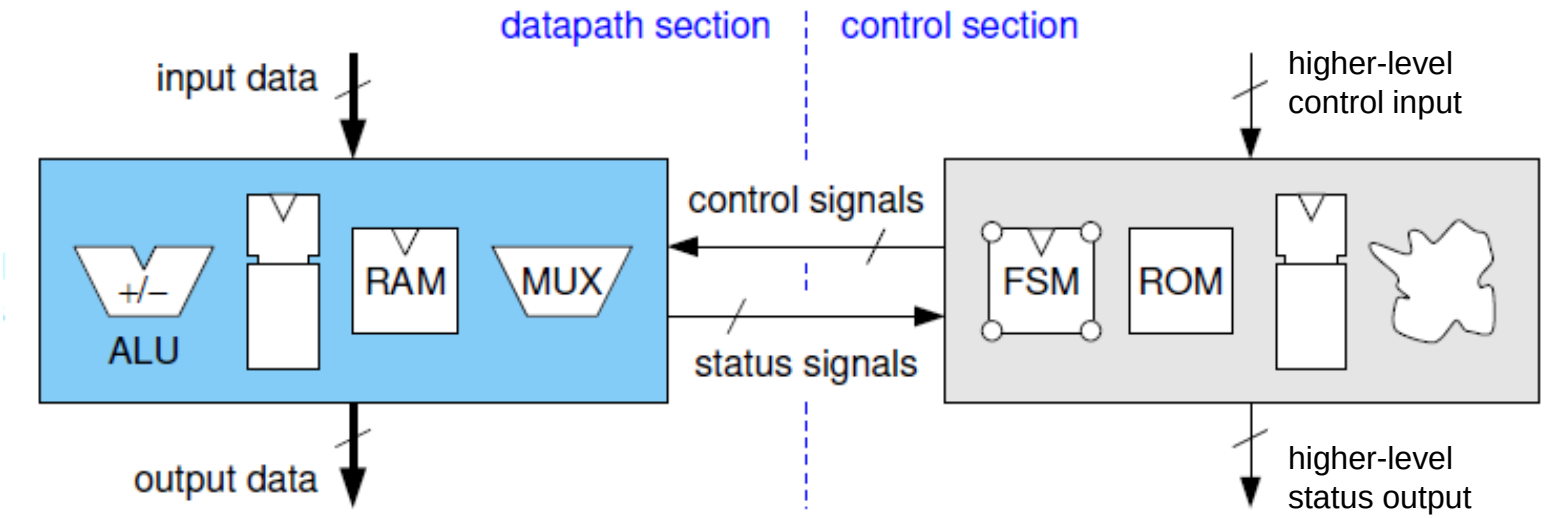
- **Isomorphic architecture:** straightforward mapping of an algorithm to hardware
  - Corresponds to the data flow graph of the algorithm
    - Vertices correspond to operations, edges route intermediate results to other vertices
  - **Every operation** is **mapped to a dedicated** combinational hardware **unit**
    - **Loops are unrolled**: loop counter must be computable (known) at design time
    - **Conditional statements become multiplexers**

```
int sigproc(int a[]) {
   int q, r;
   int sum = a[0];
   for (int i=1; i<=3;i++) {
      sum += a[i];
   }
   q = sum/8;
   r = sum % 8;
   if (r > 3) q += 1;
   return q;
}
```



Sequential Processing in Custom Hardware

- **Separation between computations and control**
  - **Datapath** : performs (arithmetic or other) operations on data and keeps intermediate resultsin memory elements. Control inputs (from control logic) define what should be done in each cycle
  - **Control** : controls datapath and manages the sequence of operations (potentially based on feedback from the datapath), but does not manipulate data itself



data processing (arithmetic) units, data switches, random logic for data manipulation, data storage

Finite state machines, (programmable or hardwired) instruction sequences, counters, hardwired control logic

Assessing Efficiency of the Hardware

- To assess the efficiency of a hardware architecture **we need** some **metrics**
- The most relevant metrics are:
  - **Timing and throughput:**
    - **Time per data item or throughput:** defined by
      - **Cycles per data item**: number of computation cycles between releasing two subsequent data items.
      - **Minimum clock period (max. freq.)**: defined by the longest path (Static Timing Analysis)
    - **Latency**: number of computation cycles from a data item entering a circuit until the pertaining result becomes available.
  - **Circuit complexity**: usually proportional to circuit size or area (for FPGAs: resource utilization)
  - **Size-time AT-product (efficiency)**: the hardware resources spent to obtain a given throughput
    - Combines circuit complexity and throughput in a **figure of merit (FOM)**
  - **Power and energy** consumption
    - Very important, but also very difficult to estimate

Characteristics of the Isomorphic Architecture

- **Generic statements are difficult,** but some general remarks are straightforward assuming an algorithm with N operations
  - **Timing and throughput:** $O(1) - O(N)$
    - **Critical path length:** determined by the algorithm, but generally long: $O(1) - O(N)$
    - **Cycles per data item:** purely combinational logic, hence, single cycle: $O(1)$
  - **Circuit complexity**: often prohibitively high: $O(N)$
    - (dedicated resources for every operation)
  - **Size-time product AT (efficiency)**: $O(N) - O(N^2)$
  - **Power and energy** consumption: don't even try to estimate…
    - Good: no power for registers, bad: lots of glitching due to long combinational paths

Isomorphic architecture is rarely attractive due to large area and only OK timing,

BUT it provides a great starting point for optimization

Systematic Approach to Steps 2-4: optimization

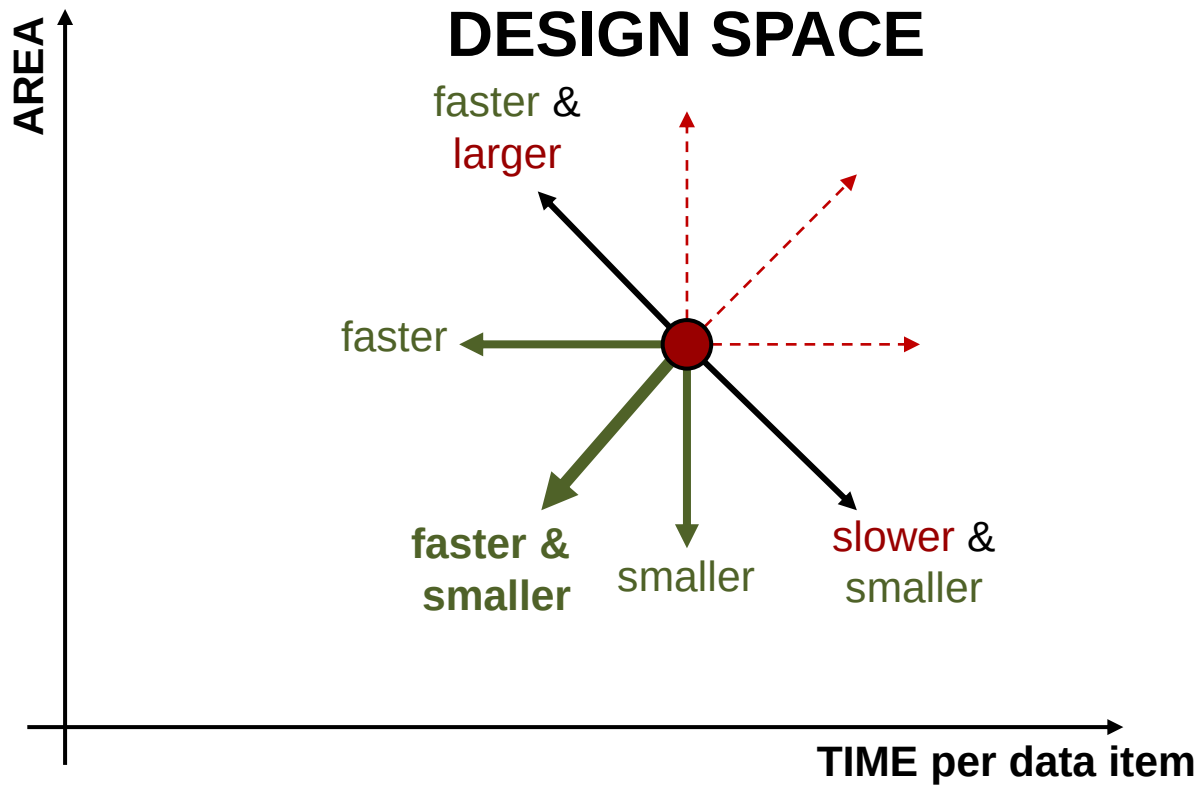
- **Defining hardware resources**, followed by scheduling, binding, and register allocation **is a crucial optimization process**
  - Determines the area of the circuit (more resources, more area)
  - Determines the number of cycles and the cycle time, i.e., the time per data item (throughput)

- How to design and optimize a datapath for different area/throughput objectives?

- We use a **transform approach** to optimization
  - **Isomorphic architecture is** the **starting point**: easy to obtain and unique
    - Straightforward representation as a block diagram with no overhead or control
  - **Systematic** (reversible) **transformations** improve some design metrics, while worsening others
    - Each transformation results in a new block diagram
    - Evaluation of the main performance metrics shows if the transformation was beneficial and hints to further transformations

Datapath Representation as Data Flow Graph

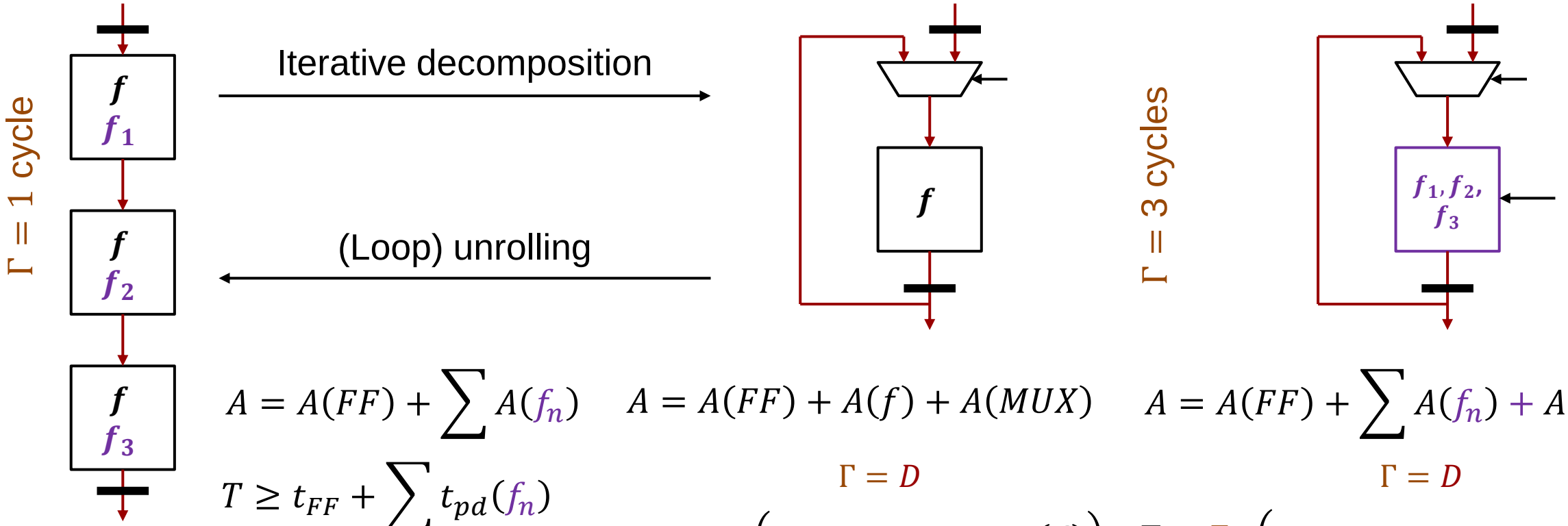
- There are six main transformations that allow for moving around in the design space, while **optimizing area (A), time per data item (T), and the AT product**.
  - **Iterative decomposition**
  - **Loop unrolling**
  - **Resource sharing**
  - **Replication**
  - **Pipelining**
  - *Retiming*



- **Pareto-optimal design**: a design where you can not make one metric better without making another one worse

Iterative Decomposition and (Loop) Unrolling

- Applicable to **sequences of dependent** identical (or similar) **operations**
  - **Iterative decomposition**: spread subsequent combinational operations across multiple cycles
  - **(Loop) unrolling**: map operations from multiple cycles into a single combination operation



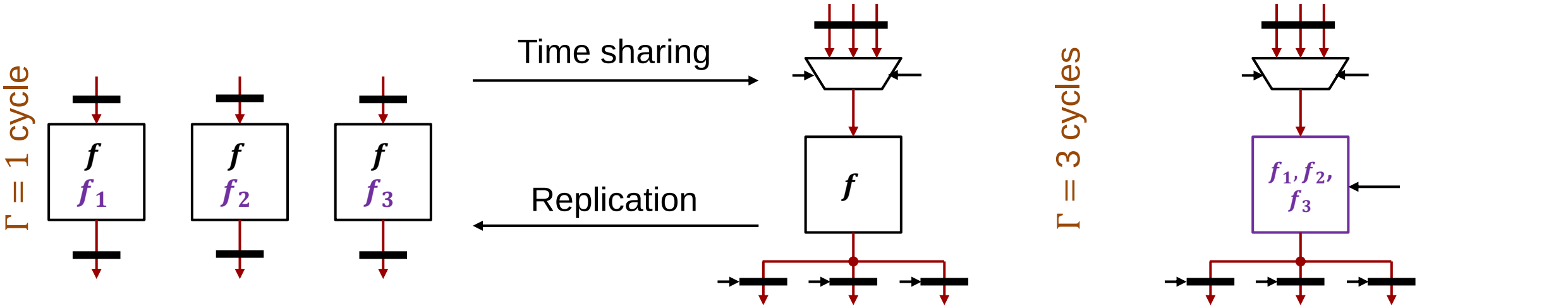
$$A = A(FF) + \sum A(f_n) \qquad T \geq t_{FF} + \sum t_{pd}(f_n)$$

$$A = A(FF) + A(f) + A(MUX) \qquad \Gamma = D \qquad T = \Gamma \cdot \left(t_{FF} + t_{MUX} + t_{pd}(f)\right)$$

$$A = A(FF) + \sum A(f_n) + A(MUX) \qquad \Gamma = D \qquad T \geq \Gamma \cdot \left(t_{FF} + t_{MUX} + \max_n t_{pd}(f_n)\right)$$

Time (Resource) Sharing and Replication

- Applicable to **independent** (parallel) identical (or similar) **operations**
  - **Time (resource) sharing:** distribute parallel operations across multiple cycles
  - **Replication:** replicate a resource that is shared for independent operations over multiple cycles to carry these out in parallel



$$A = A(FF) + \sum A(f_n)$$

$$T = t_{FF} + \max_n t_{pd}(f_n)$$

$$A = A(FF) + A(f) + A(MUX)$$

$$\Gamma = D$$

$$T = \Gamma \cdot \left(t_{FF} + t_{MUX} + t_{pd}(f)\right)$$

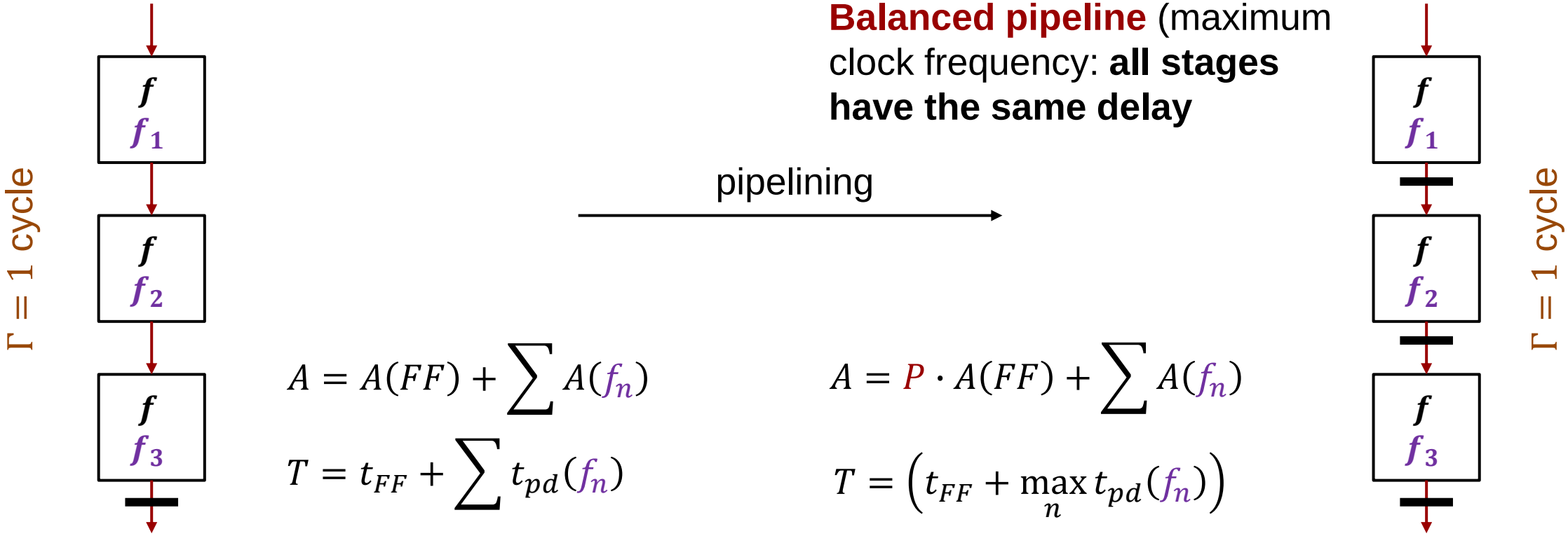
$$A = A(FF) + \sum A(f_n) + A(MUX)$$

$$\Gamma = D$$

$$T \geq \Gamma \cdot \left(t_{FF} + t_{MUX} + \max_n t_{pd}(f_n)\right)$$

Pipelining

- Applicable to **long** (slow) **sequences of combinational logic**
  - **Pipelining:** split a combinational path into segments that operate independently and in parallel on different data by inserting pipeline registers

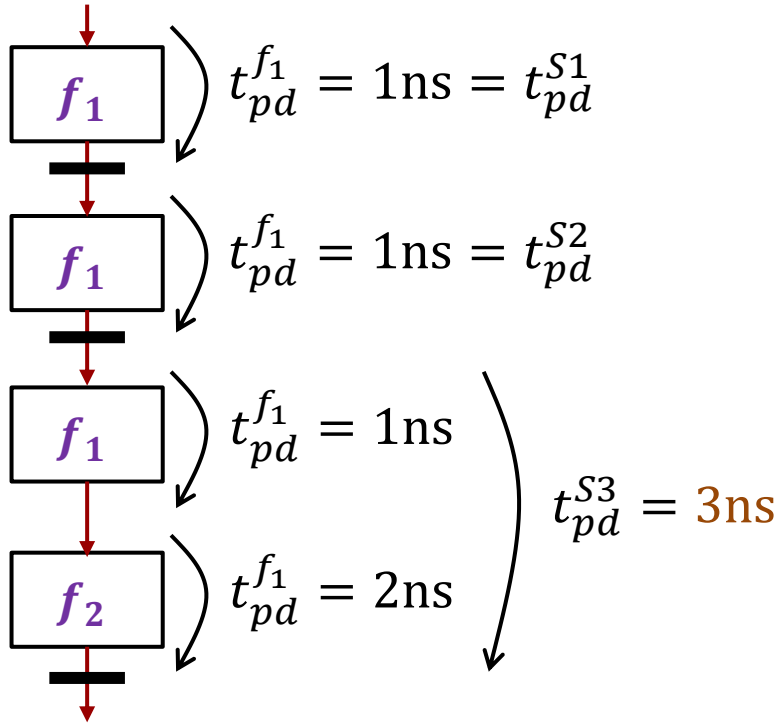


Retiming

- **Pipelines are not always well balanced**: not all stages have the same delay

- Unbalanced pipelines originate from
  - Careless placement of the registers
  - Attempts to minimize the number of registers to be inserted when pipelining
  - From new critical paths created when inserting shimming registers

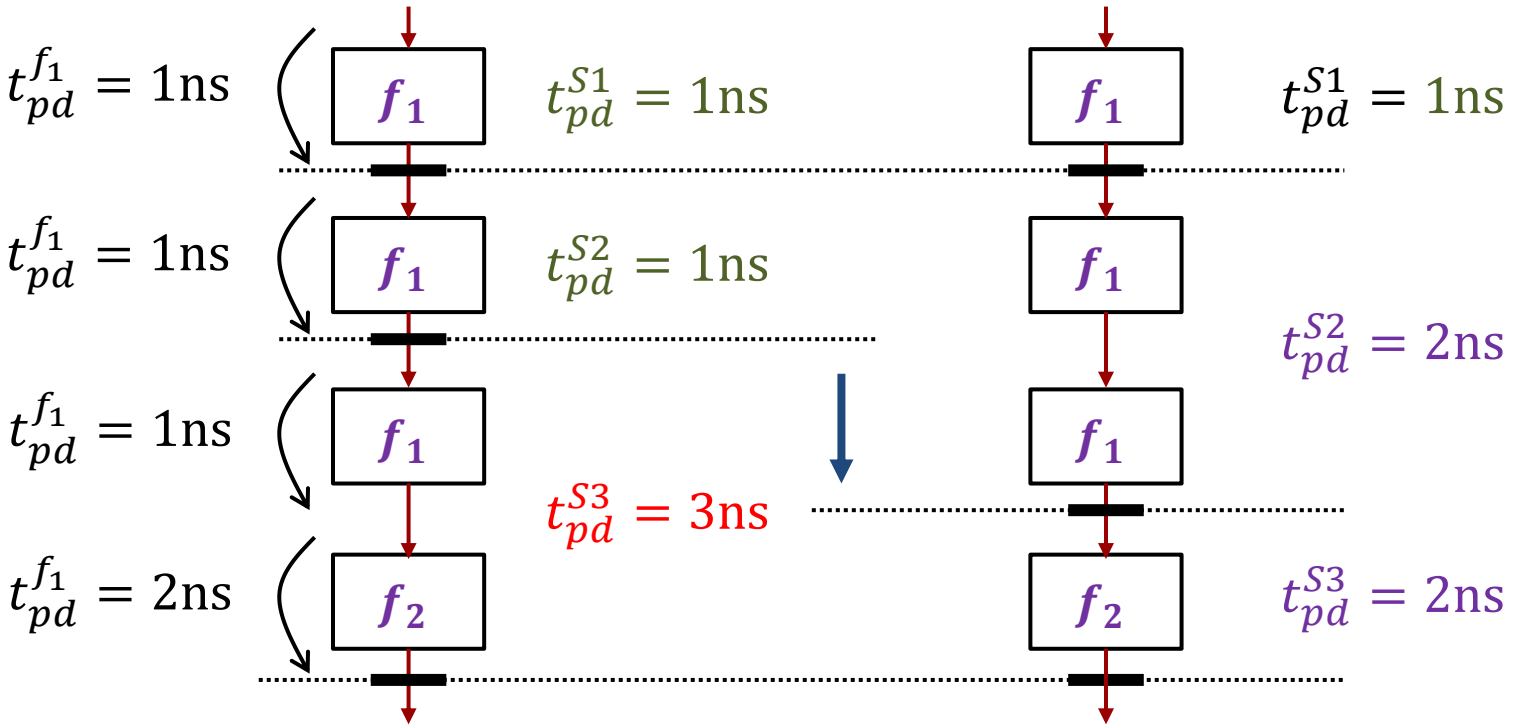
- **Clock period** of a pipelined design is **determined by the longest pipeline stage** (pipeline stage with the longest path)



$$T = \left(t_{FF} + \max_n\{t_{pd}^{S1}, t_{pd}^{S2}, t_{pd}^{S3}\}\right) = t_{FF} + 3\text{ns}$$

Retiming

- **Moving pipeline registers to reduce the length of the relevant longest stage** while increasing the delay of shorter (irrelevant) stages **reduces clock period**

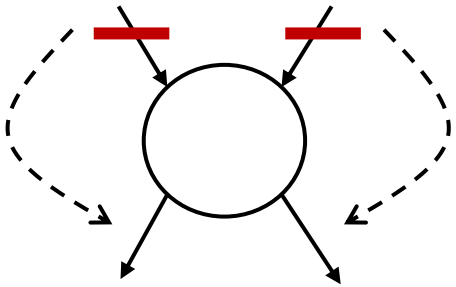


$$T = \left(t_{FF} + \max_n\{t_{pd}^{S1}, t_{pd}^{S2}, t_{pd}^{S3}\}\right) = t_{FF} + 3\text{ns}$$

$$T = \left(t_{FF} + \max_n\{t_{pd}^{S1}, t_{pd}^{S2}, t_{pd}^{S3}\}\right) = t_{FF} + 2\text{ns}$$

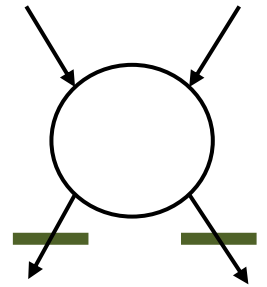
Systematic Retiming

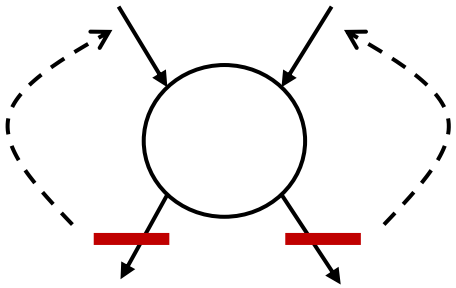
- **Pipelines registers can be moved (retiming)** “forward” and “backward”, step-by-step to rebalance an unbalanced pipeline
- **Retiming**: applied to a node in the schematic (data-dependency graph)



Forward

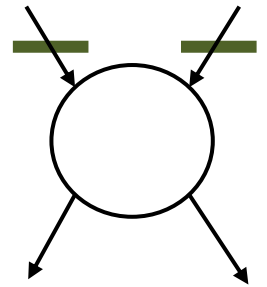
- **Remove** a registers from every **input** of the node
- **Insert** a register at every **output** of the same node





Backward

- **Remove** a registers from every **output** of the node
- **Insert** a register at every **input** of the same node



- **Nodes can only be retimed if all inputs/outputs have sufficient registers to remove one**

Comparison of Architectural Transformations

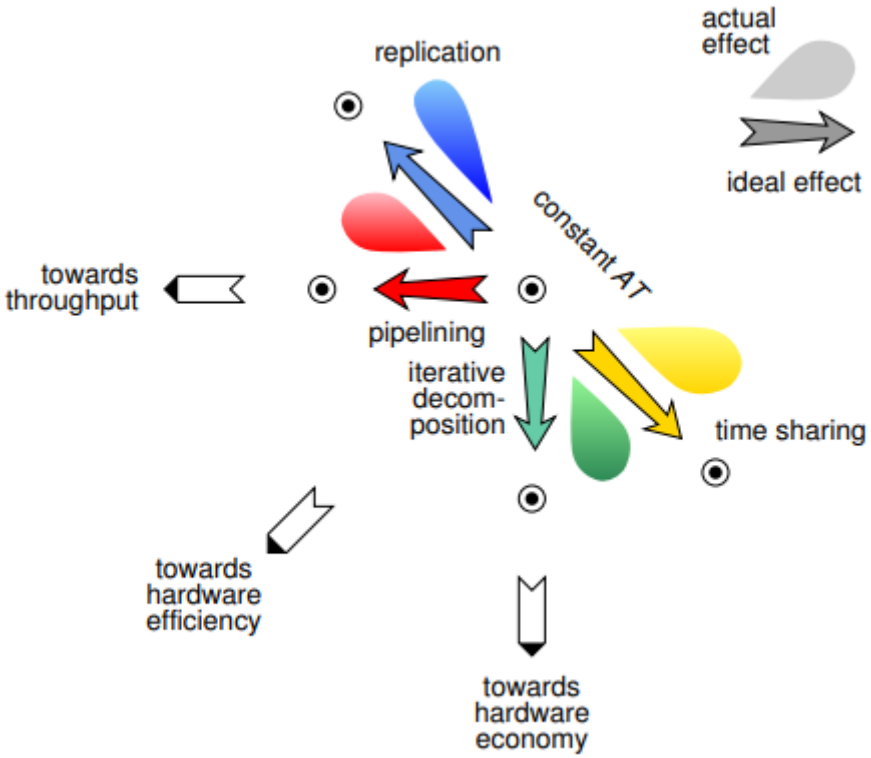
- Comparison with standard metrics for area, time per data item and efficiency (AT)

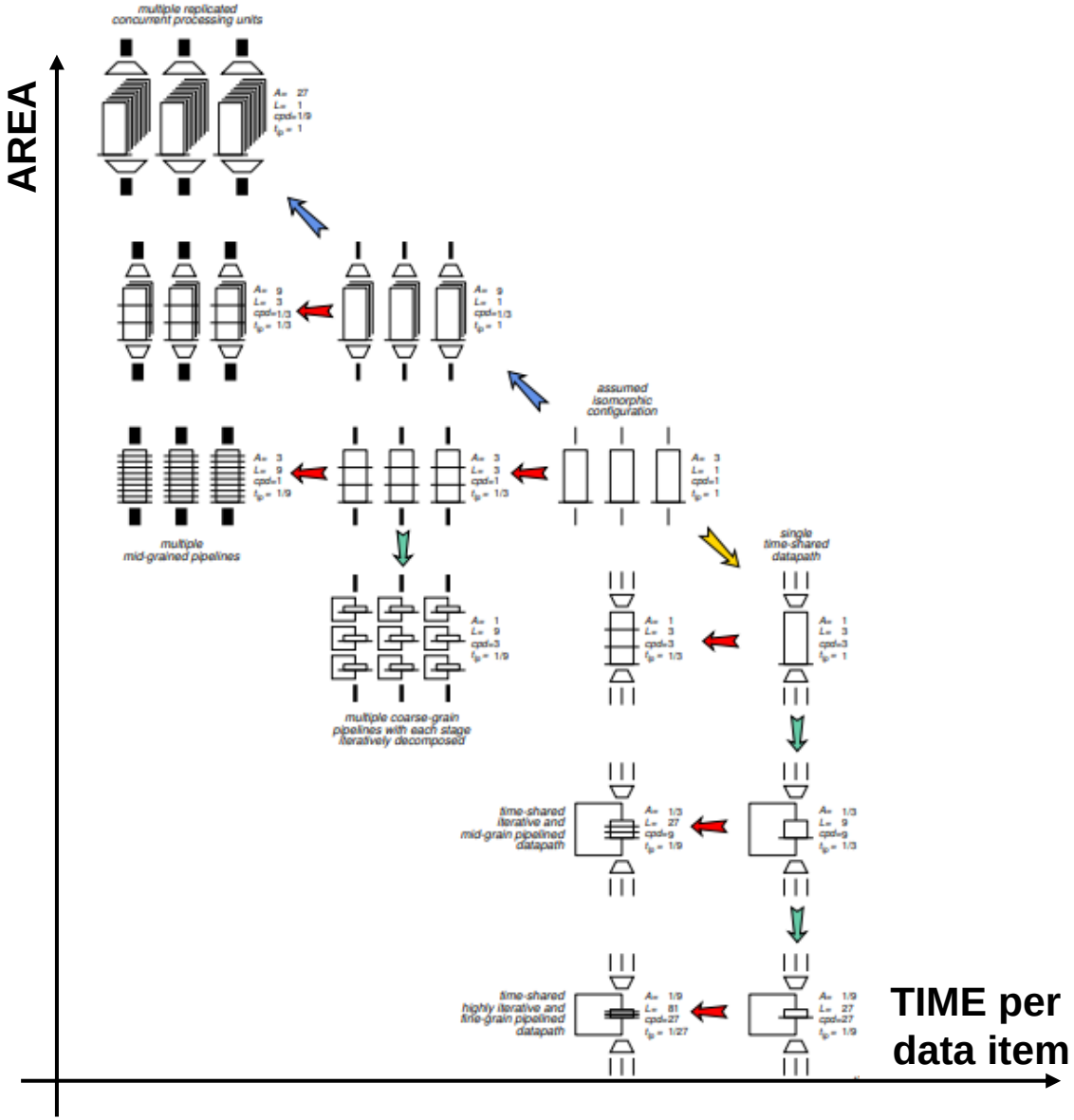
Operation	Factor	Cycles	Clock period		**Time per data item**		**Area**		**AT product**
			min	max	min	max	min	max	
It. Decomp.	D	D	$1/D+\delta$	$1+\delta$	$1+D\delta$	$D+D\delta$	$1/D+\delta$	$1+\delta$	$1/D+\delta$
Unrolling	D	1	$<D$	D	$<D$	D	D	D	D^2
Time sharing	D	D	1	$1+\delta$	D	$D+D\delta$	$1/D$	$1+\delta$	1
Replication	D	1	1	1	1	1	D	D	D
Pipelining	P	1	$1/P+\delta$	$1+\delta$	$1/P+\delta$	$1+\delta$	$1+\delta$	$1+\delta$	$1/P+\delta$

  - Most transformations have their pros and cons.
  - Strong differences even for a given transformation depending on how well it can be applied

Architectural Transformations in the Design Space

- Architectural transformations allow to move around in the design space



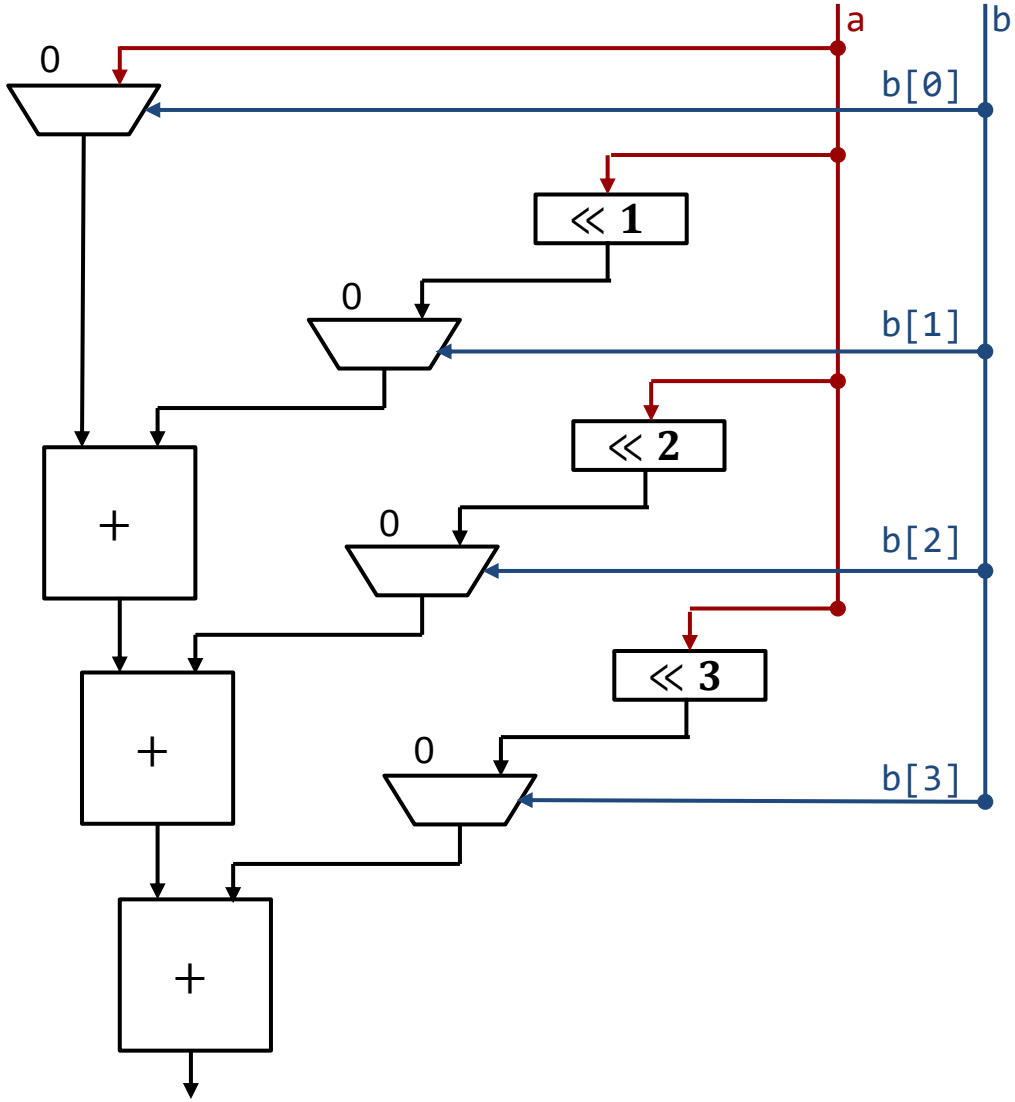


Mult-4 Example: Isomorphic Architecture

- Array multiplier algorithm

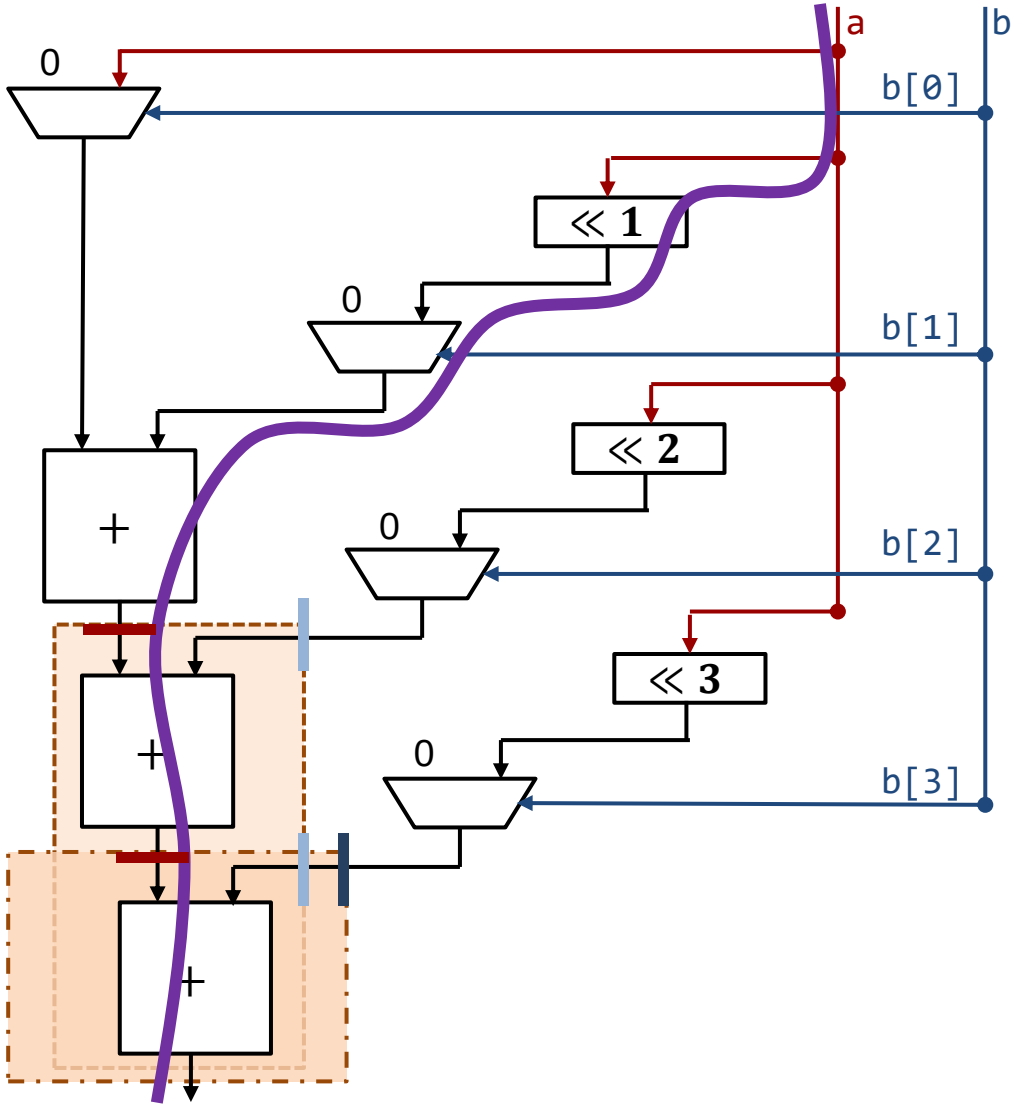
				a_3	a_2	a_1	a_0	
			x	b_3	b_2	b_1	b_0	
				a_3b_0	a_2b_0	a_1b_0	a_0b_0	-- bp_0
			a_3b_1	a_2b_1	a_1b_1	a_0b_1		-- bp_1
		a_3b_2	a_2b_2	a_1b_2	a_0b_2			-- bp_2
	a_3b_3	a_2b_3	a_1b_3	a_0b_3				-- bp_3
p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0	

```
int mult-4(a,b) {
   if(b[0] == 1) q = a;
      else q = 0;
   for (i=1; i<=3;i++) {
      if(b[i] == 1) q += a<<i;
         else q += 0;
   }
   return q;
}
```



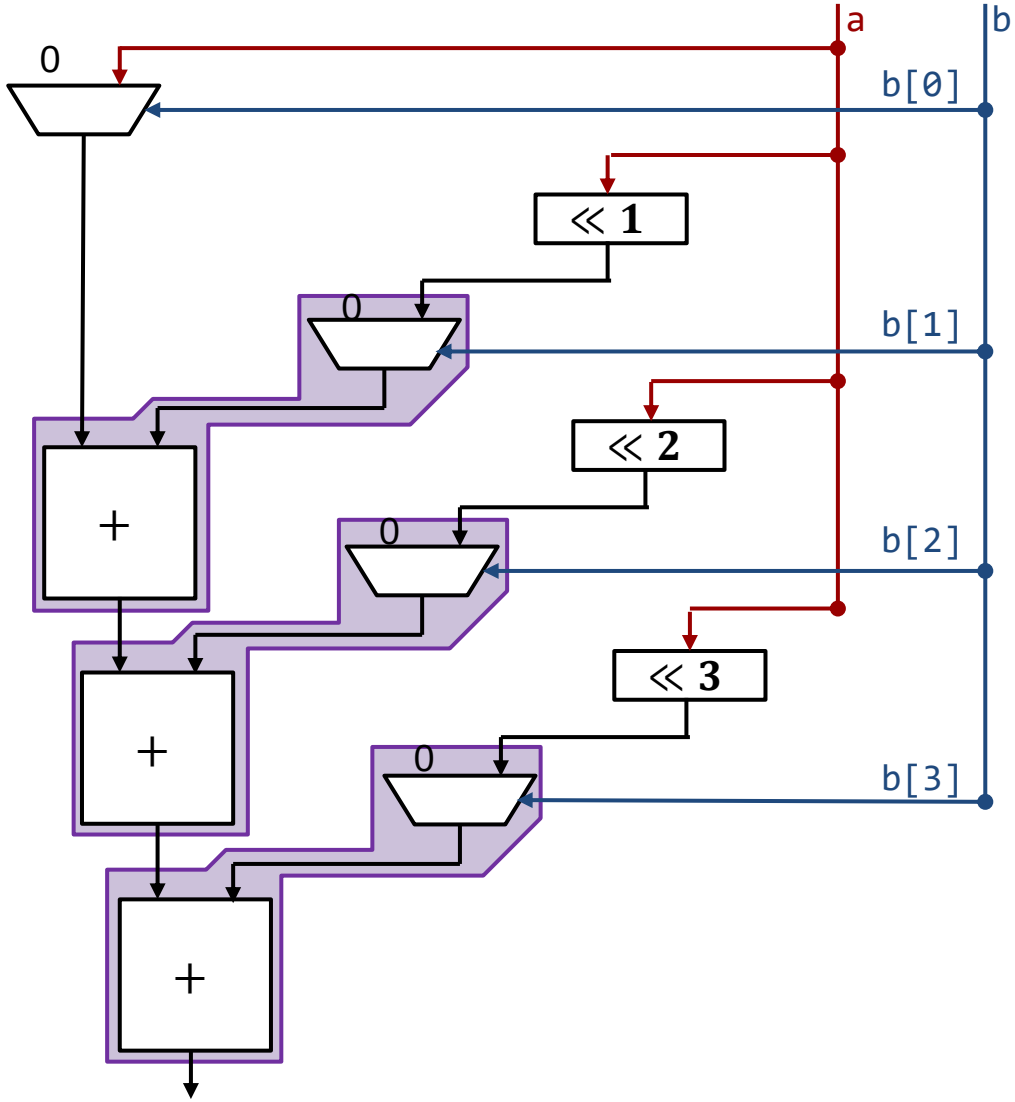
Mult-4 Example: Pipelining

- Array multiplier **PIPELINING**
  1. Identify **critical path**
  2. Cut the critical path into P ~equal pieces by **inserting pipeline registers**
  3. **Equalize the latency of all paths** by
     - Enclosing all down-stream vertices of each pipeline register in a separate box
     - **Inserting** additional **shimming registers** (for each pipeline register) on the input of each box



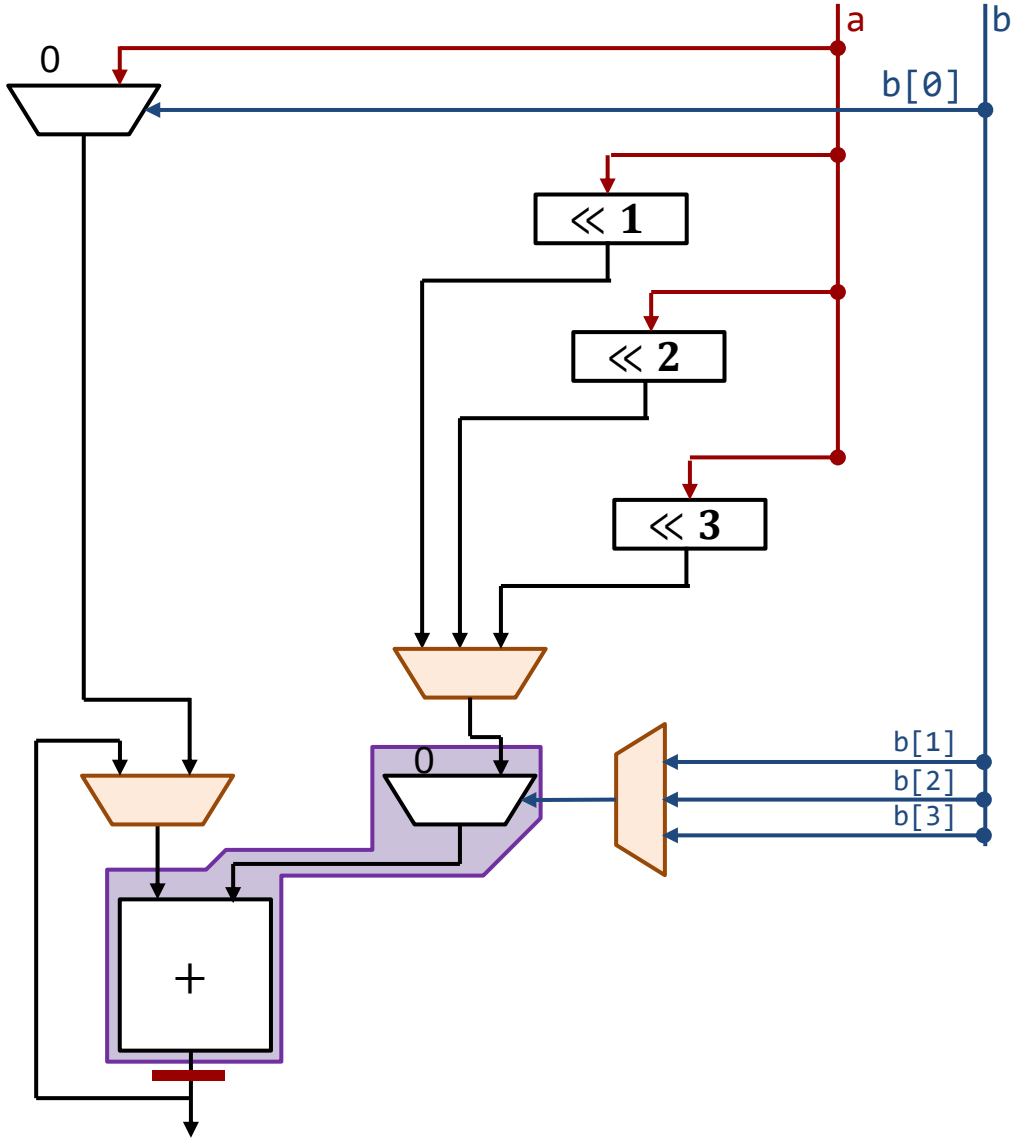
Mult-4 Example: Iterative Decomposition (1/2)

- Array multiplier **ITERATIVE DECOMPOSITION**
  1. Identify **largest common sub-circuits** across subsequent logic stages



Mult-4 Example: Iterative Decomposition (2/2)

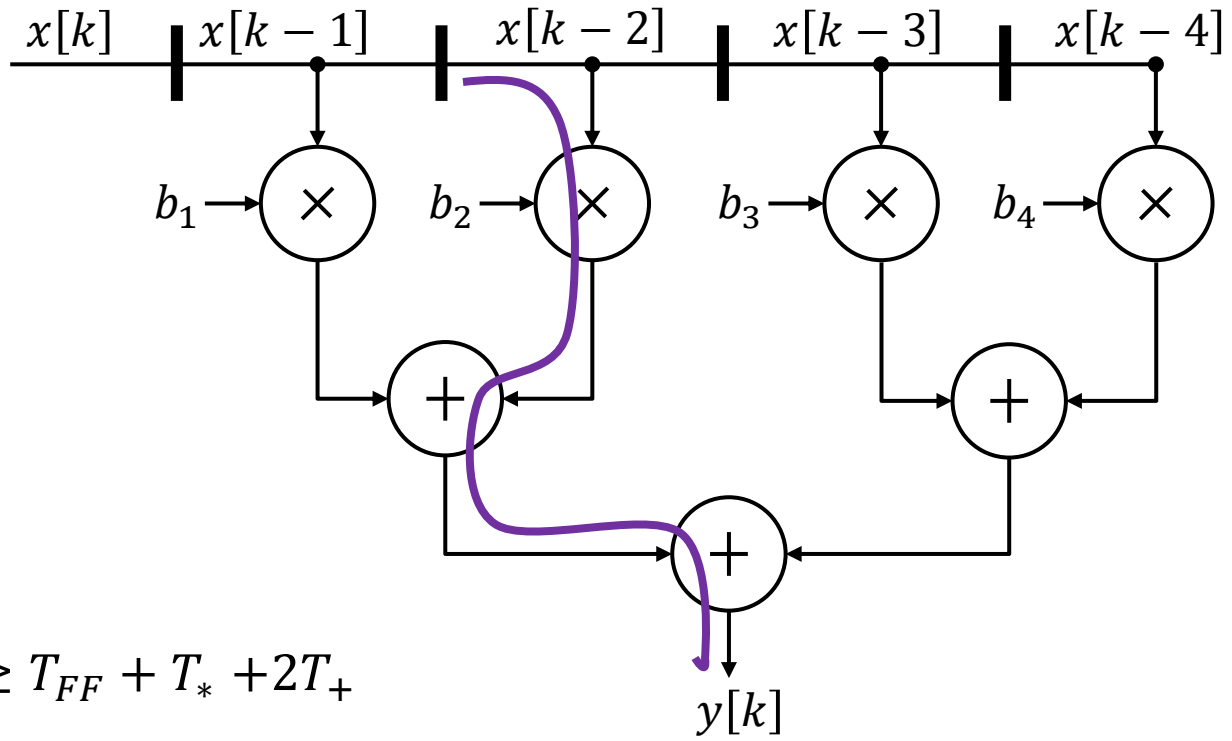
- Array multiplier **ITERATIVE DECOMPOSITION**
  1. Identify **largest common sub-circuits** across subsequent logic stages
  2. Keep only a single instance
     - **Allocate/add registers** to store the output
     - **Add MUXes** to select the inputs from the ones used previously for the different instances



FIR Filter Example

- Construct a Finite Impulse Response (FIR) filter from its equation

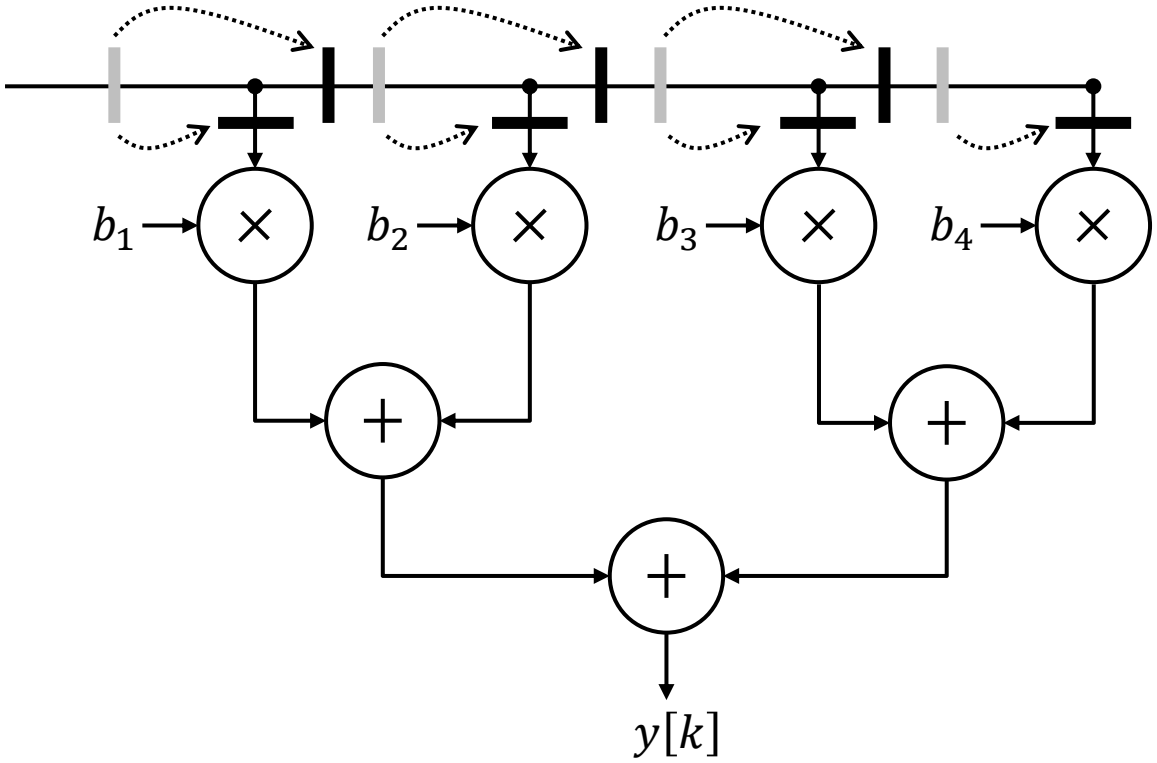
$$y[k] = \sum_{n=1}^{4} b_n \cdot x[k-n]$$



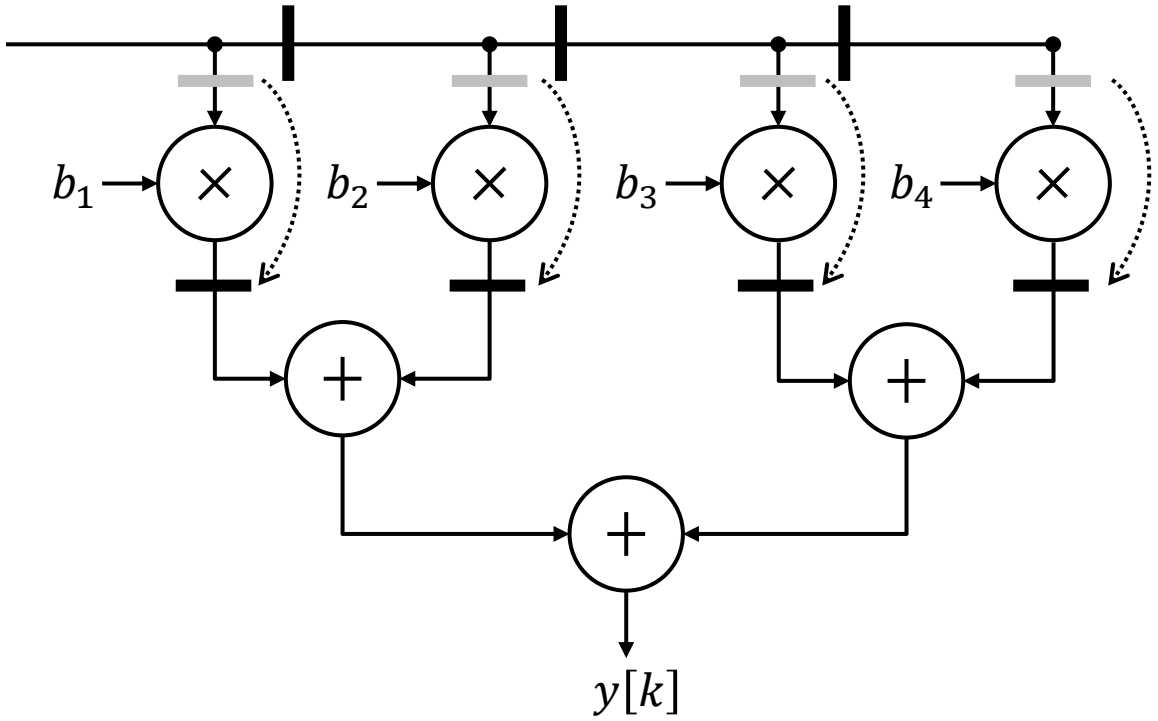
$$T \geq T_{FF} + T_{*} + 2T_{+}$$

FIR Filter Example (Retimed)

- Construct a Finite Impulse Response (FIR) filter from its equation



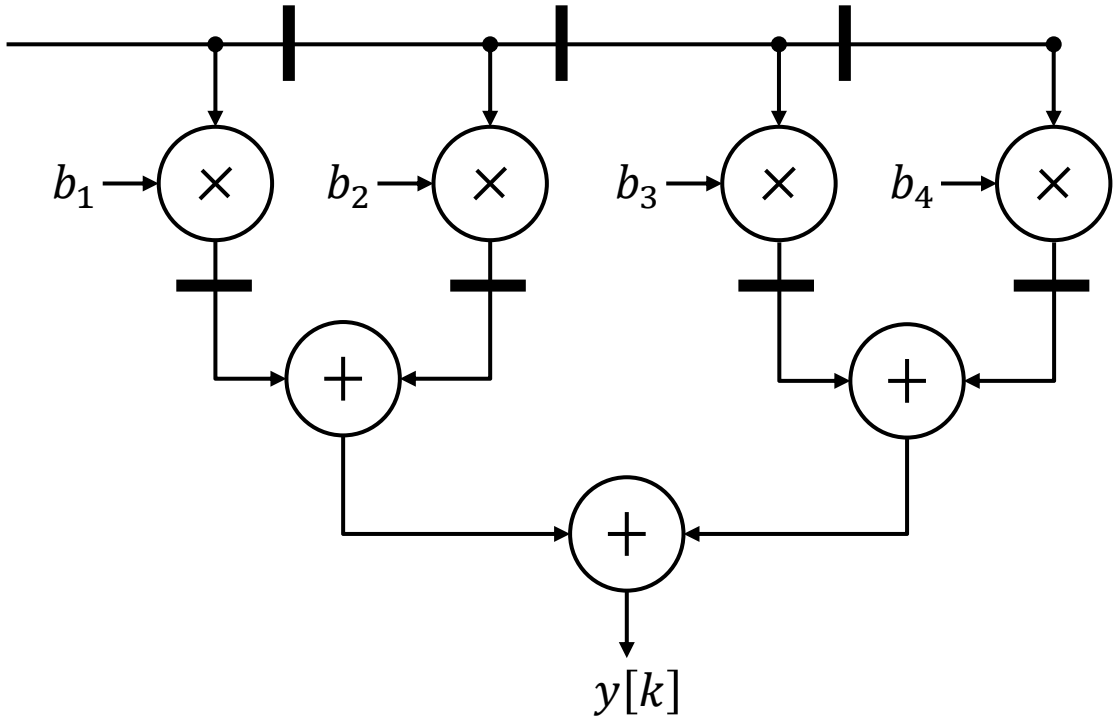
$$T \geq T_{FF} + T_* + 2T_+$$



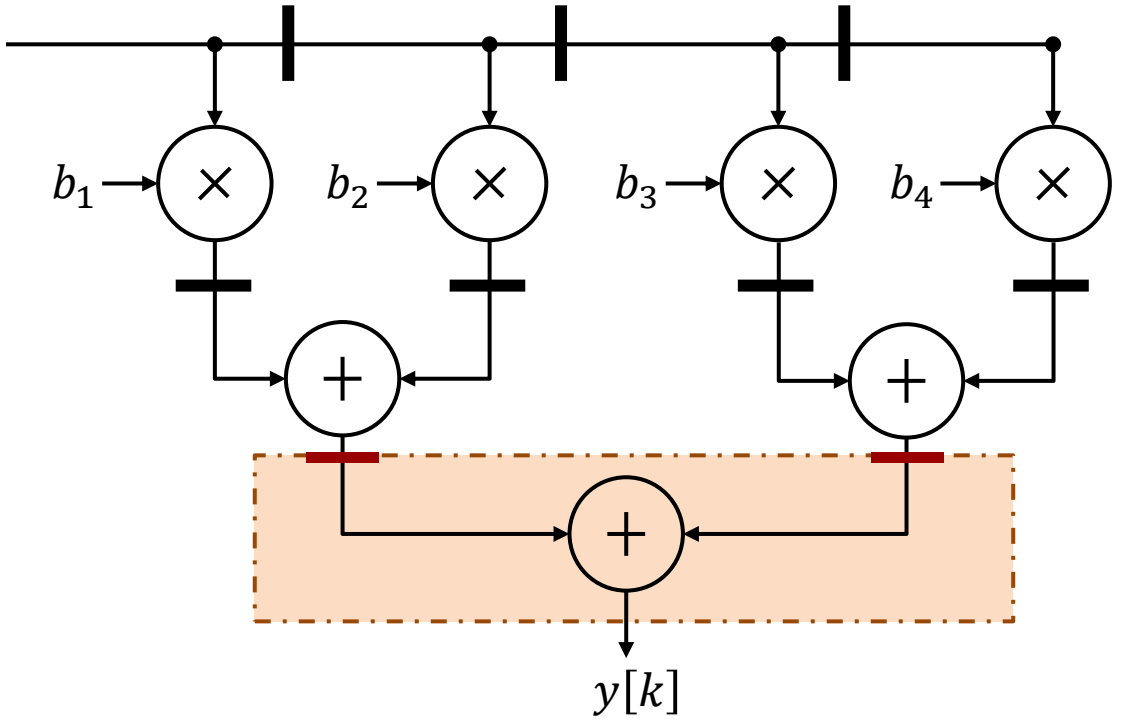
$$T \geq T_{FF} + \max\{T_*, 2T_+\}$$

FIR Filter Example (Retimed & Pipelined)

- Construct a Finite Impulse Response (FIR) filter from its equation



$$T \geq T_{FF} + \max\{T_*, 2T_+\}$$

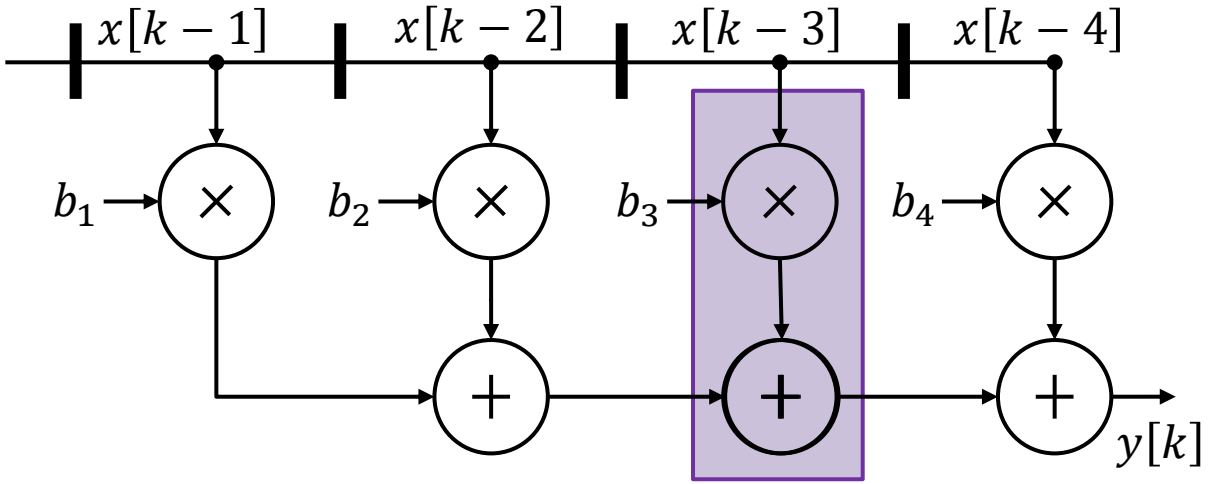


$$T \geq T_{FF} + \max\{T_*, T_+\}$$

FIR Filter Example (Iterative Decomposition)

- Construct a Finite Impulse Response (FIR) filter from its equation

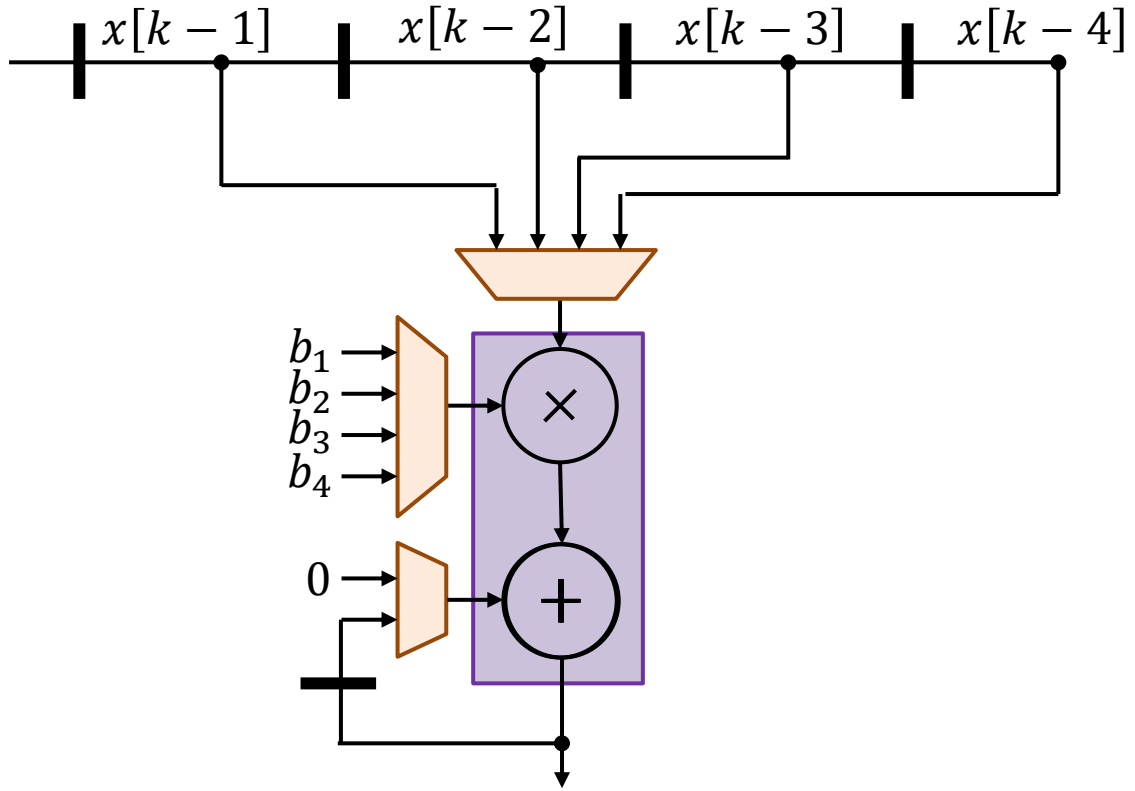
$$A = 4A_{FF} + 4A_{*} + 3A_{+}$$



$$y[k] = \sum_{n=1}^{4} b_n \cdot x[k-n]$$

$$T \geq T_{FF} + T_{*} + 2T_{+}$$

$$A = 5A_{FF} + 3A_{MUX} + A_{*} + A_{+}$$



$$T \geq 4(T_{FF} + T_{MUX} + T_{*} + T_{+})$$