Dataflow Design

Montek Singh
UNC Chapel Hill

ASYNC 2024 Summer School
Week 1: July 1

Outline

- Dataflow basics
  - Pipelining primitives
- Performance estimation
  - “Canopy Graph” analysis

Dataflow basics

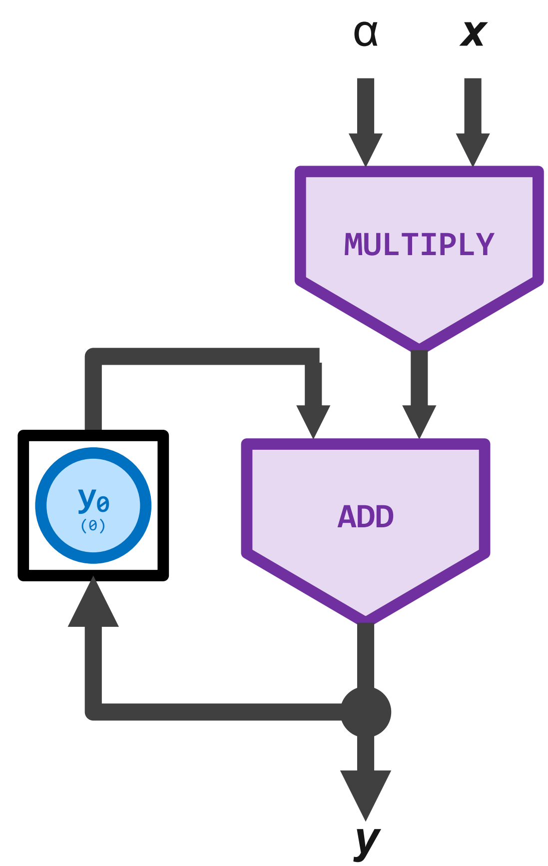
What is Dataflow?

- Graphical description of operations in a computation
- Sequencing is determined by data dependencies
  - inputs trigger a function
  - … instead of an overall control structure
- Intuitive, natural representation for:
  - data-driven algorithms, e.g. DSPs
  - stream processing
- Implementation is not necessarily asynchronous
  - but async is often a natural match

Example: multiply-accumulate

Motivation: linear algebra core operation

$$\boldsymbol{y} \leftarrow \alpha \boldsymbol{x} + \boldsymbol{y} \qquad \text{(SAXPY)}$$

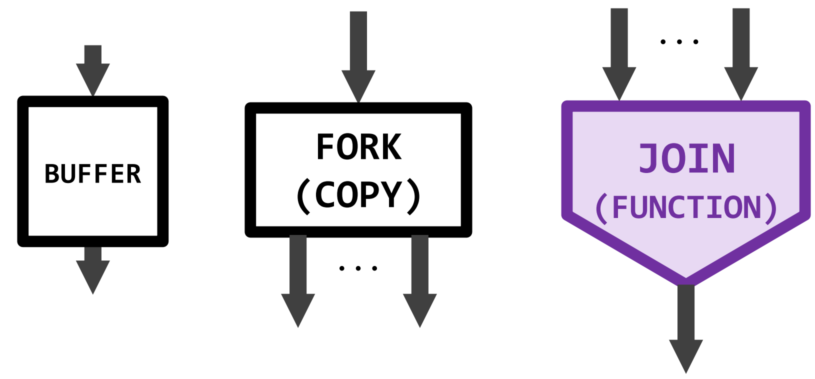


If you care about DSP, HPC, AI/deep learning… this is a useful kernel to implement

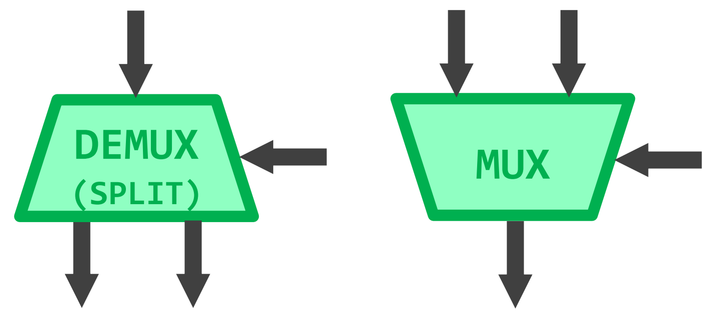
[Original slide by Benjamin Hill]

Dataflow primitives

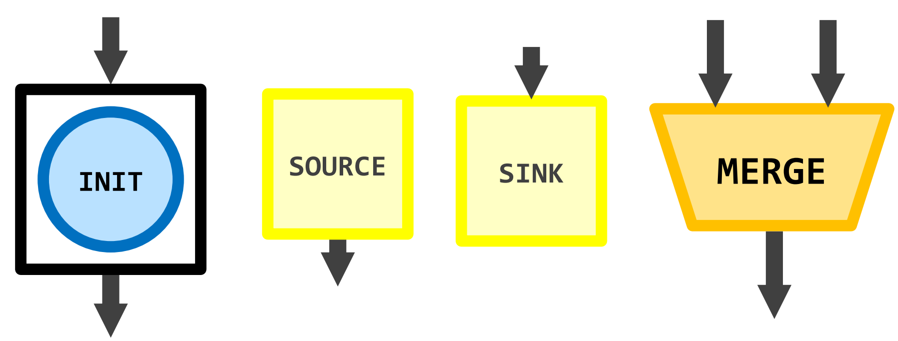
- Reading from all input channels, writing to all output channels



- Reading from 1, writing to one-of-*N* (demux)
- Reading from one-of-*N*, writing to 1 (mux / conditional merge)



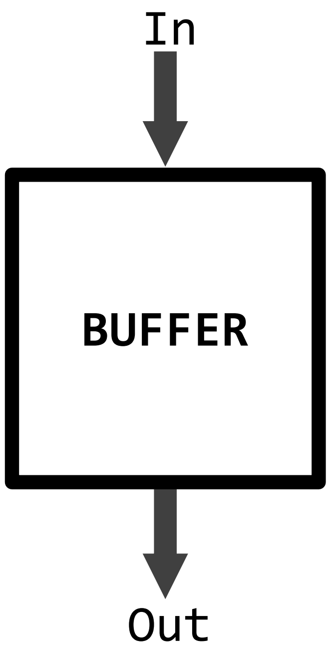
- Other misc useful blocks:
  - initialization
  - source/sink
  - merging/arbitration



[Modified from original slide by Benjamin Hill]

BUFFER

- Transmit token from input to output with storage and handshaking flow control
  - one pipeline stage (FIFO stage)
  - latch + handshake control



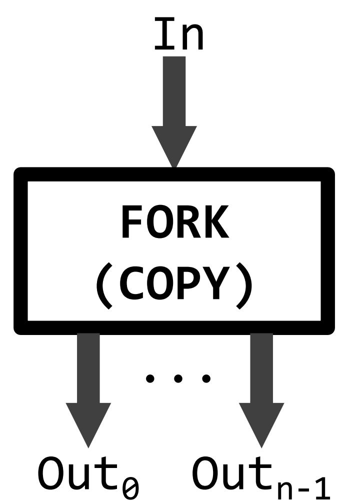
Also known as: slack buffer, one-place FIFO, latch

```
*[In?x; Out!x]
```

[Original slide by Benjamin Hill]

FORK / COPY

Copy input token to multiple destinations



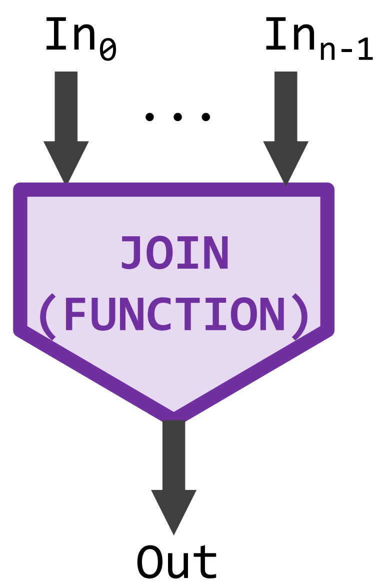
Also known as: n-way link

`*[In?x; Out0!x, …, Outn-1!x]`

[Original slide by Benjamin Hill]

JOIN / FUNCTION

Read values from all inputs, compute result and send on output



Example functions: arithmetic, logic, decoding, etc.

$*[\ In_0?arg_0,\ In_1?arg_1,\ \dots\ ,\ In_{n1}?arg_{n-1};$
$\quad Out!func(arg_0,arg_1,\dots,arg_{n-1})$
$]$

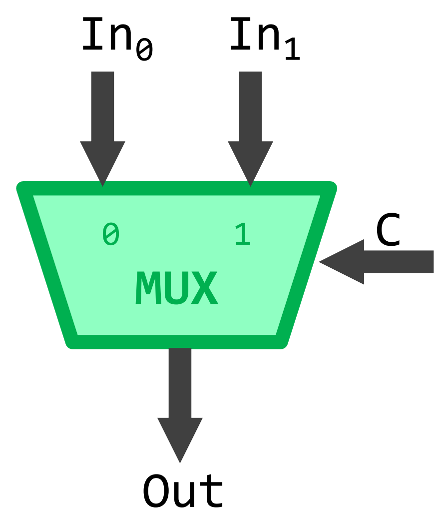
Also known as: OPERATOR

[Original slide by Benjamin Hill]

Multiplexer (MUX)

Select one input to send to output based on control signal

- ignore other input (do not consume)
- generalizable to *N* inputs



Not to be confused with combinational MUX:

- same basic behavior, but this is a dataflow operator
- unused input channel is not consumed; its data is still available

```
*[C?c;
   [  c=0 -> In0?x
   [] c=1 -> In1?x
   ];
  Out!x
 ]
```

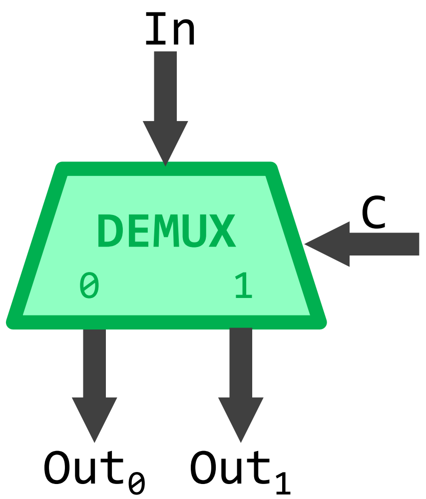
Also known as: controlled merge, conditional join

[Original slide by Benjamin Hill]

DEMUX

Steer/route input to one of two outputs

- based on value of control signal
- generalizable to *N* outputs



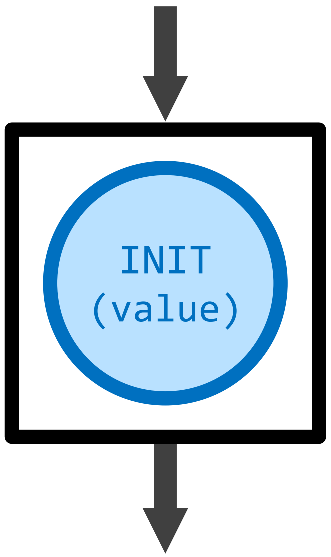

```
*[In?x, C?c;
   [  c=0 -> Out0!x
   [] c=1 -> Out1!x
   ]
 ]
```

Also known as: SPLIT

[Original slide by Benjamin Hill]

Initial token buffer

Send one initial value token, then behave as a normal buffer



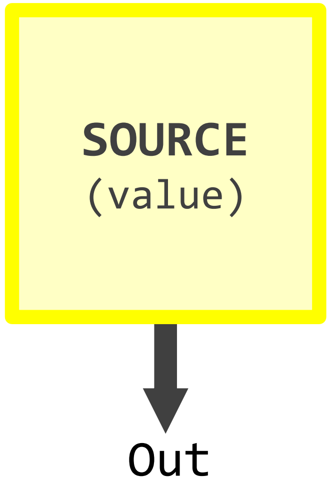
Also known as: INITIALIZER

```
Out!value; *[In?x; Out!x]
```

[Original slide by Benjamin Hill]

SOURCE

Repeatedly send tokens with same constant value



Also known as: bit/token generator

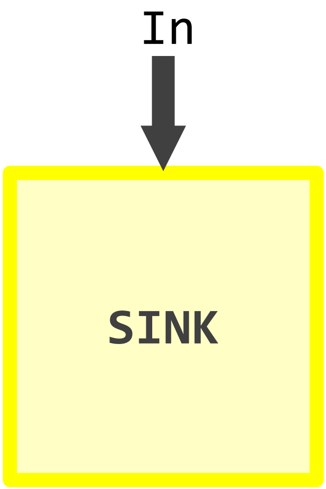
```
*[Out!value]
```

[Original slide by Benjamin Hill]

SINK

Consume and discard input token

- Not particularly useful by itself, but in combination with other dataflow primitives



Also known as: (bit) bucket

`*[In?value]`

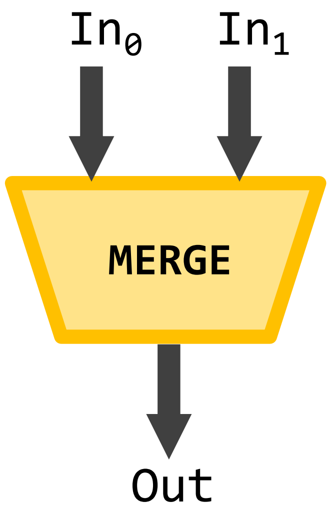
[Original slide by Benjamin Hill]

Uncontrolled merge

Combine two input streams to one output

Depending on system design, selection is either:

- **deterministic** – only one input will arrive at a time (ensured by environment)
- **non-deterministic** – requires arbitration to choose if both inputs can arrive close together



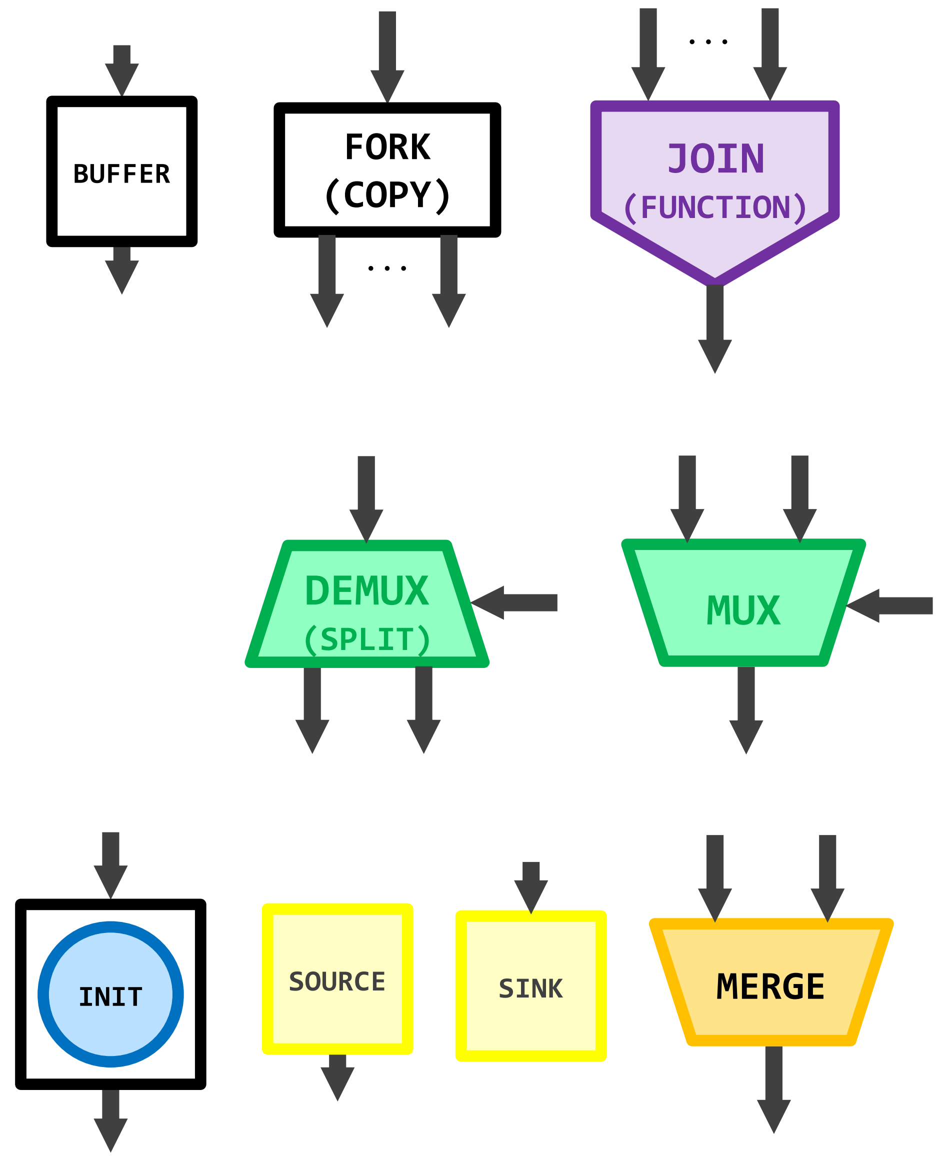

```
*[ [  #In0 -> In0?x
   [] #In1 -> In1?x
   ];
   Out!x
 ]
```

Also known as: MIXER, JOIN

[Original slide by Benjamin Hill]

Recap: Dataflow primitives

- Reading from all input channels, writing to all output channels
- Reading from 1, writing to one-of-*N* (demux)
- Reading from one-of-*N*, writing to 1 (mux / conditional merge)
- Other misc useful blocks:
  - initialization
  - source/sink
  - merging/arbitration



[Modified from original slide by Benjamin Hill]

Some useful design patterns

Wagging or Multithreading

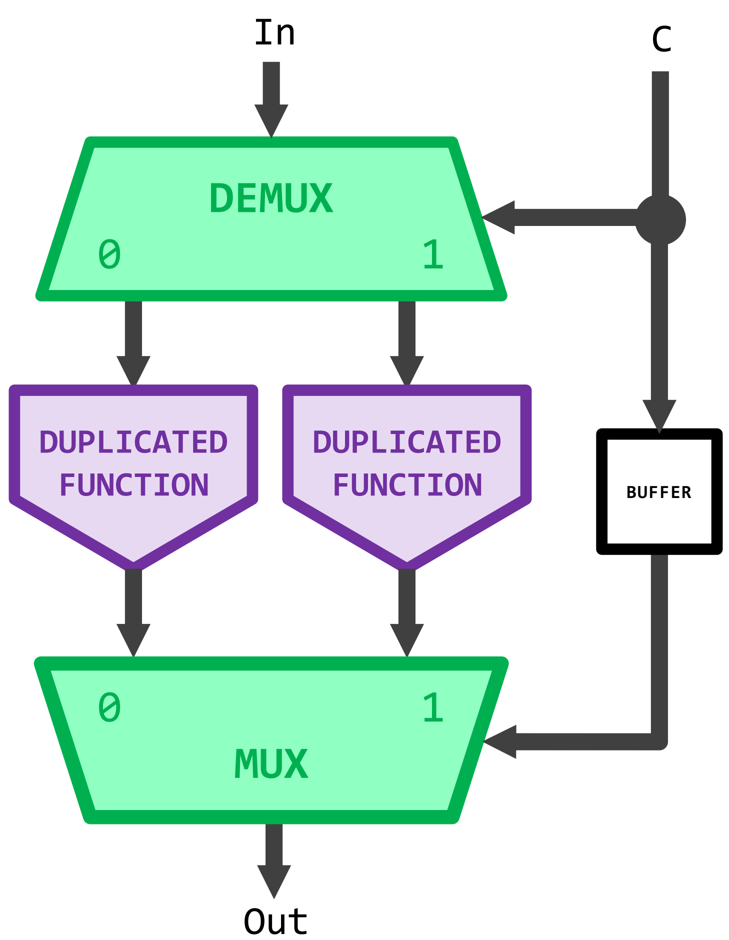
Problem: Slow function block

Solution: Duplicate function block and interleave data between them

➔ Improves throughput at the cost of area

Example: large arithmetic block where it is difficult to add internal pipelining

Not just for compute, could also be storage (e.g. tree FIFO)

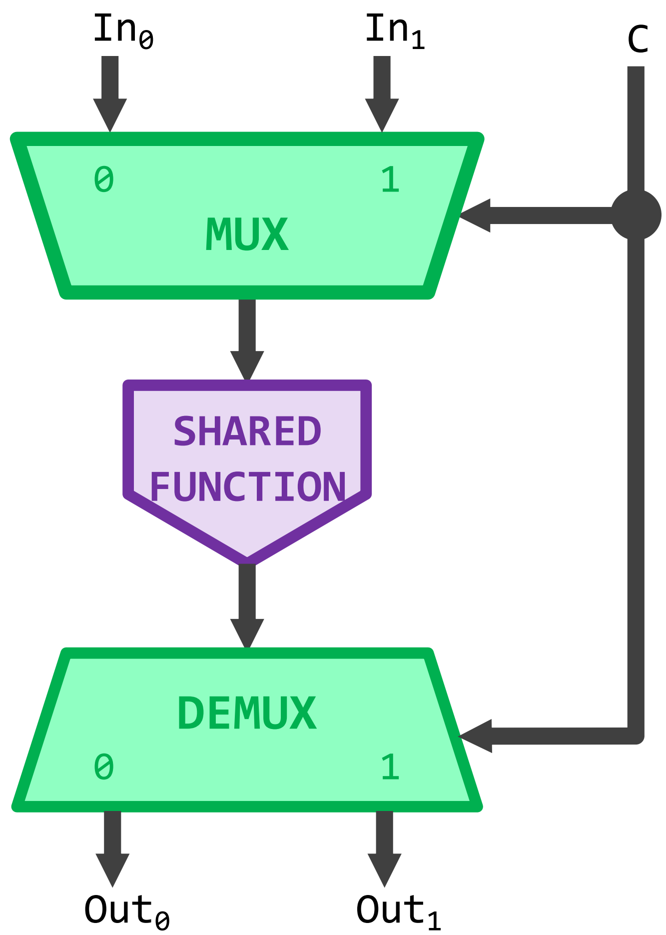


[Original slide by Benjamin Hill]

Resource sharing

Idea: share one expensive or unique resource between multiple users

Improves area at the cost of throughput

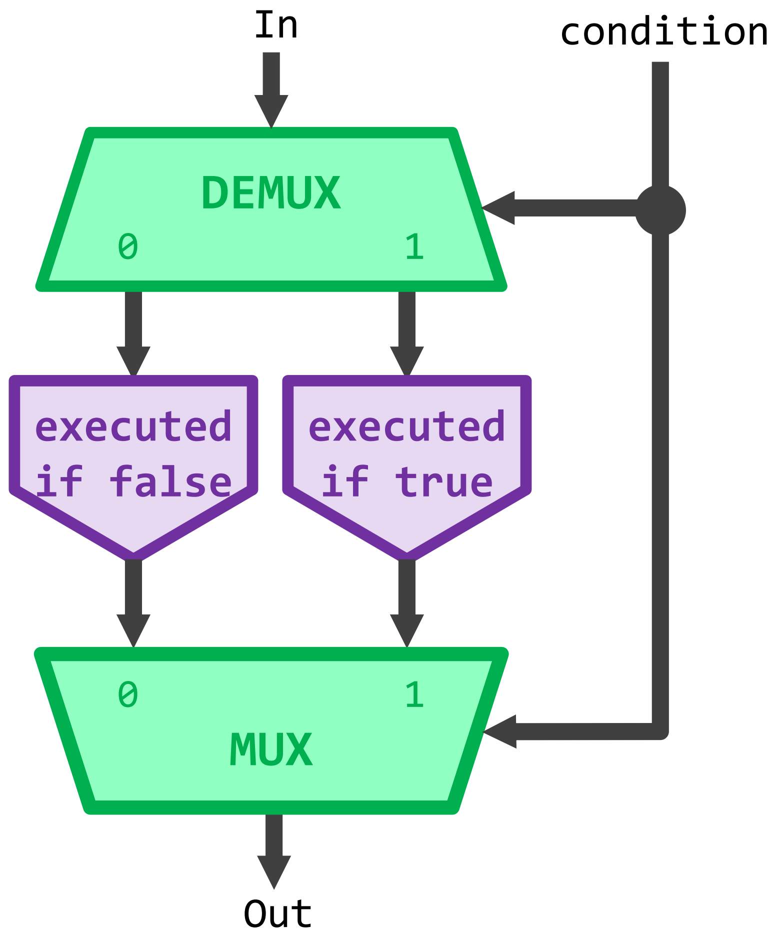


[Original slide by Benjamin Hill]

IF statement

Useful for high-level synthesis

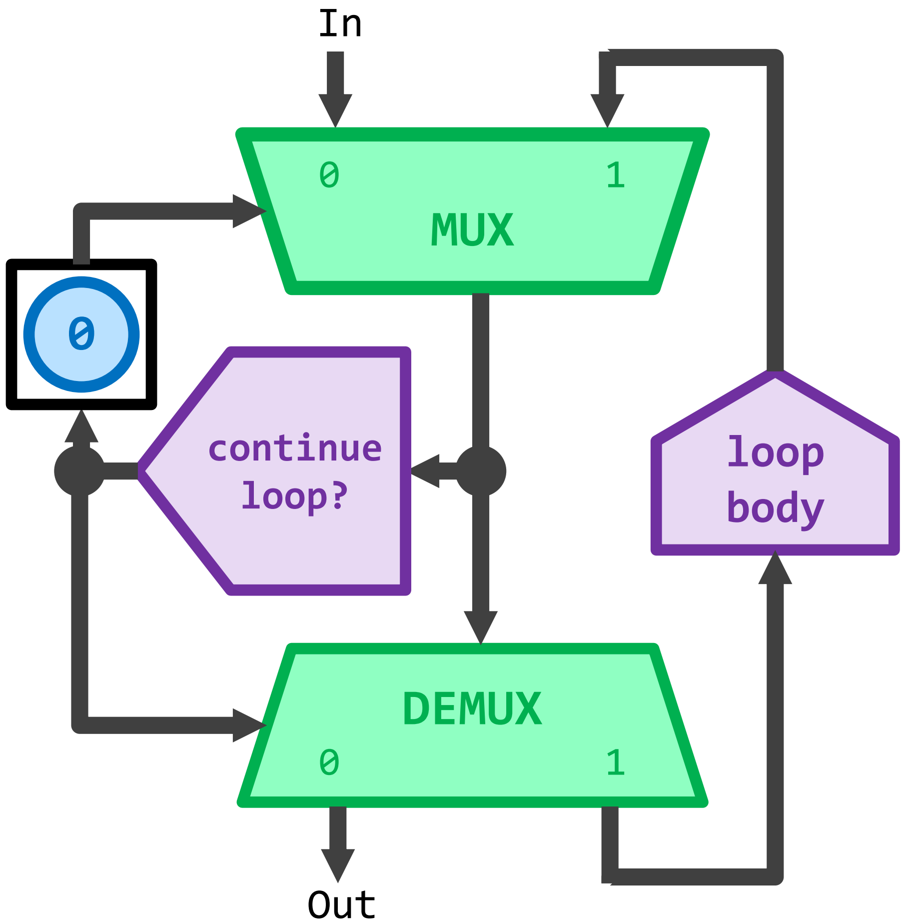
Shown with FUNCTION blocks but can also be other dataflow graphs (e.g. nested IF statements)



[Original slide by Benjamin Hill]

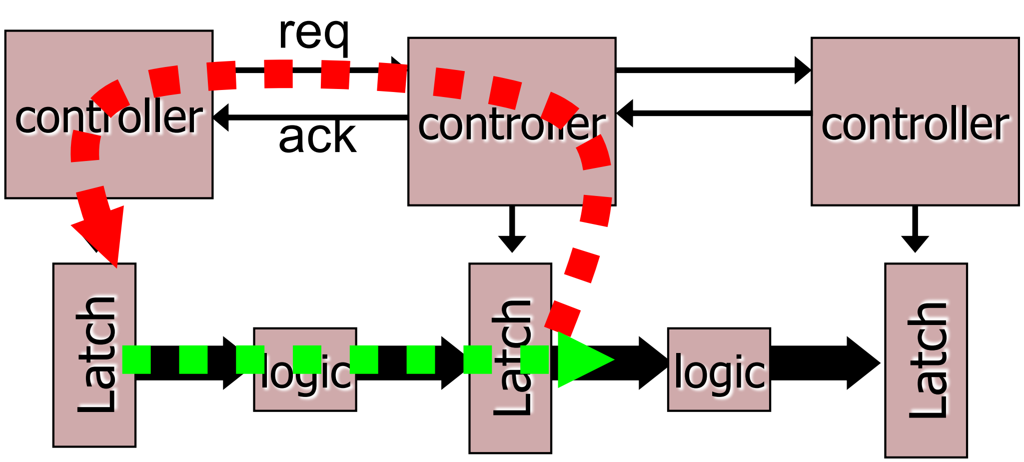
WHILE loop

Can also implement other loop constructs with a similar pattern

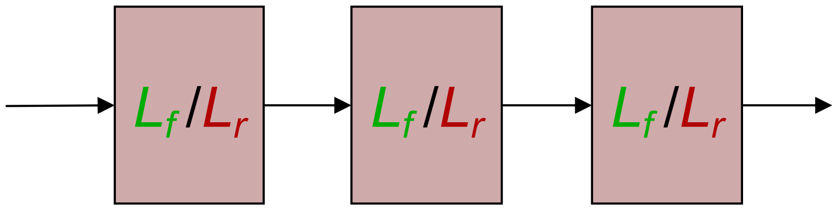


Performance Estimation

Performance Basics: pipeline stages



Cycle time in an asynchronous pipeline



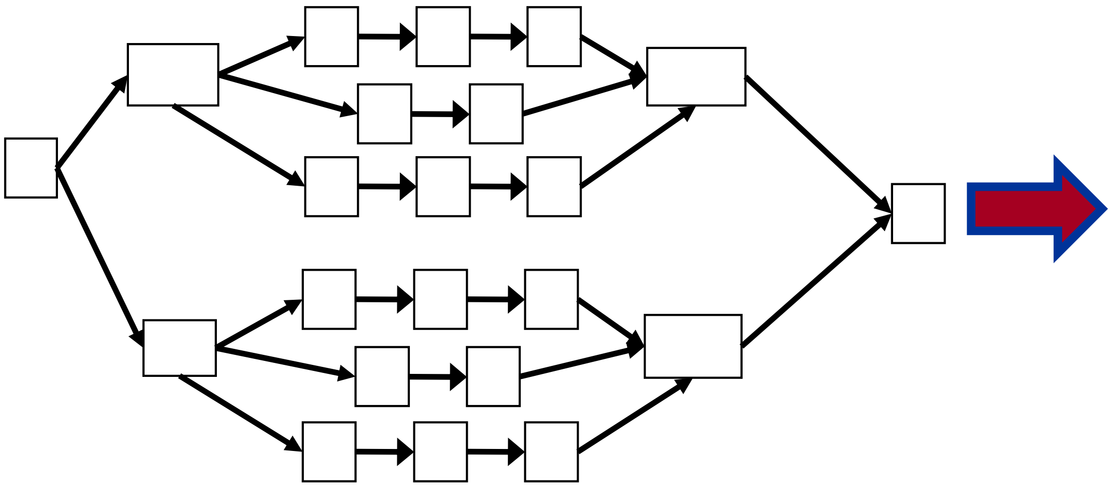
An abstracted view of the pipeline

Each stage characterized by three delays:

- Forward latency, L_f
  - Time for data to propagate forward
- Reverse latency, L_r
  - Time for a stage to receive and process ack
  - Time for a 'hole' to travel backward
- Cycle time, $T = L_f + L_r$
  - Throughput, tpt = 1 / cycle time

Goal

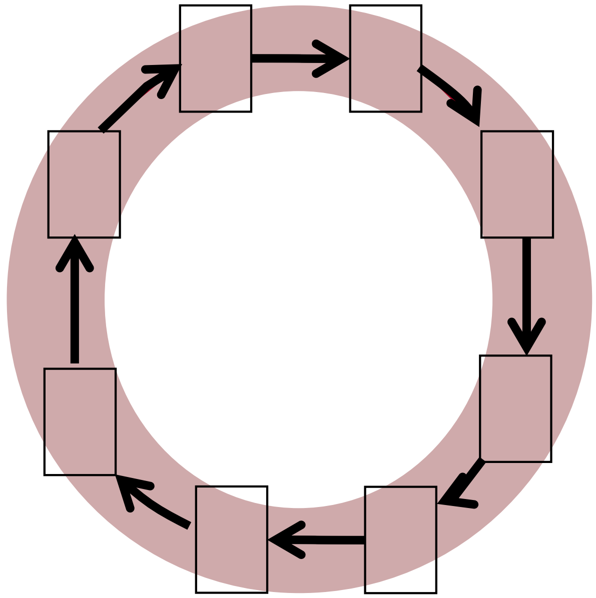
Input: pipelined system-level implementation

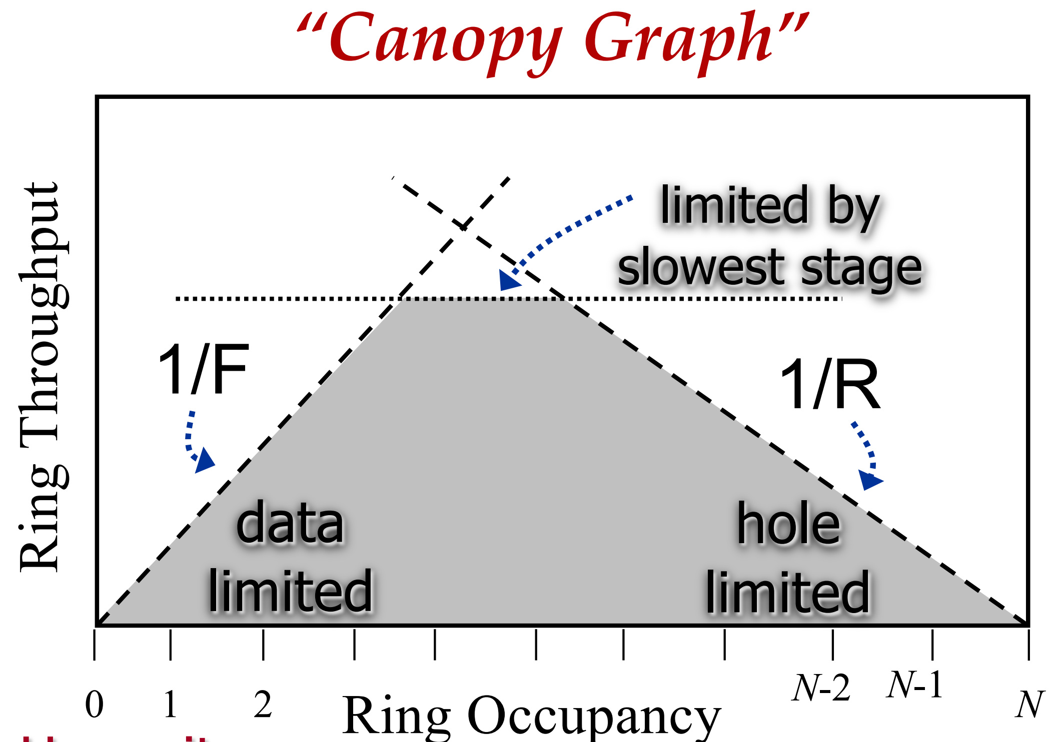


Output: system-level throughput

- Motivation: crucial part of an optimizing design flow
  - Used repeatedly in an optimization loop
  - Requires <u>low runtime</u> and <u>good accuracy</u>

Early work: Pipeline Rings



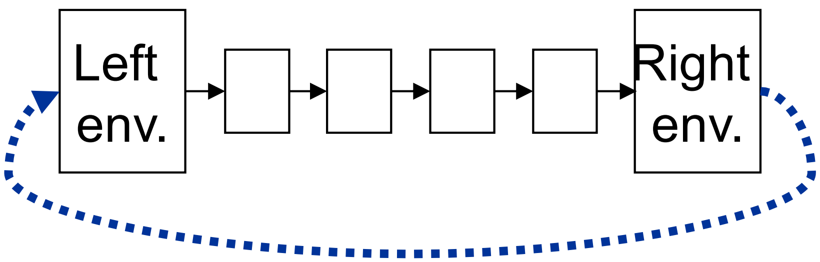


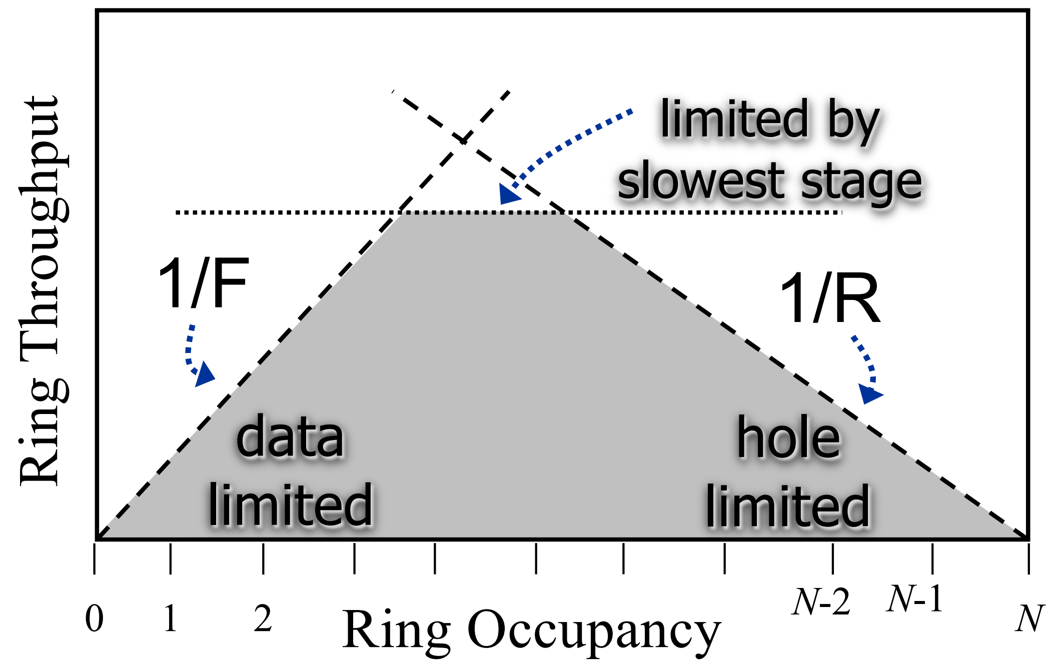
Classic work by T. Williams and M. Horowitz [ISSCC-91]

Ring throughput depends on its occupancy (#items)

- **For small number of items**: under-utilization limits throughput
- **For small number of holes**: congestion limits throughput
- Throughput also limited by the **slowest stage**
- Graph is a convex shape: *"Canopy Graph"*
  - [term coined by Singh et al. ASYNC-02 and Gill/Singh ICCAD-08]

Canopy Graphs for linear pipeline

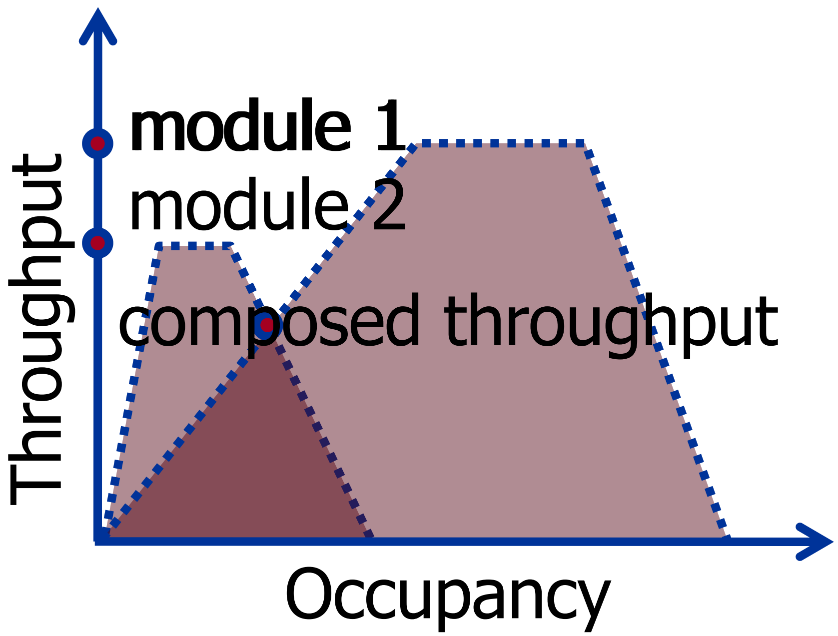


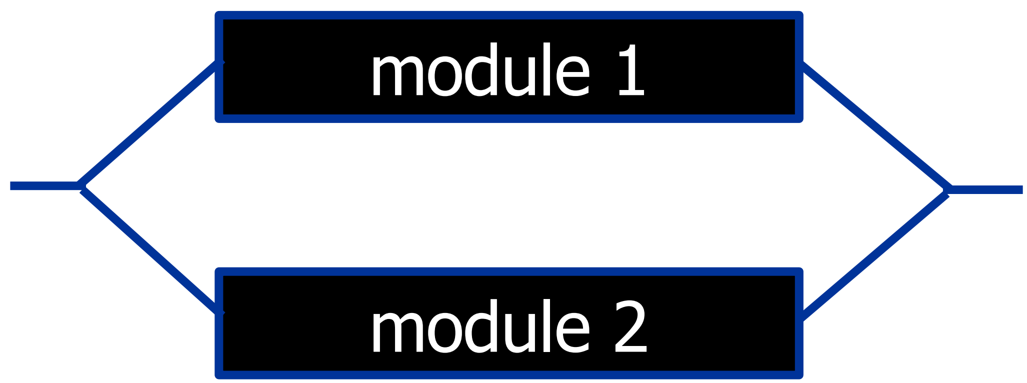


- Canopy graph: also useful approximation for *linear* pipelines
  - In steady state: linear pipeline can be modeled as ring
    - Rate at which data enters and leaves is identical
    - *i.e.* one token leaves ➔ one token enters

Key Idea: Generalize Canopy Graphs

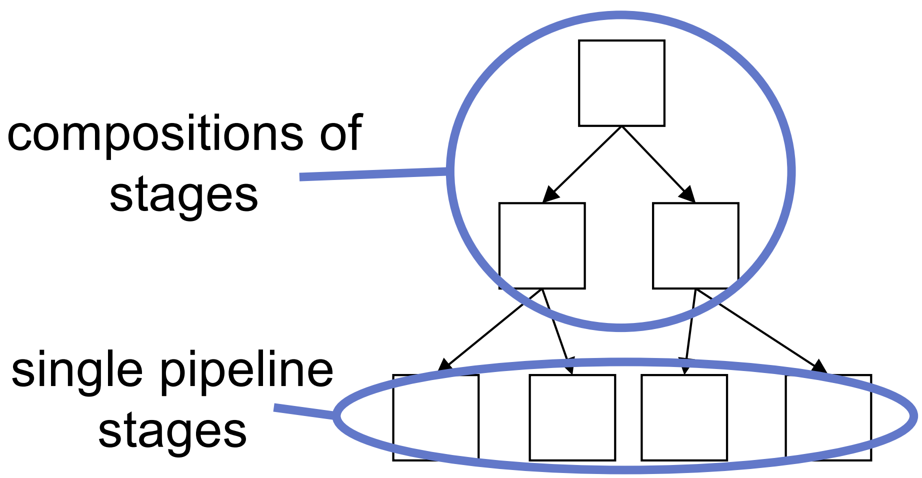
- Goal: Find the system-level throughput for an async dataflow system
  - Use a modular, "divide-and-conquer" method
- Challenge: Throughput is not composable
  - Complex interdependencies dictate throughput
- Take problem to higher dimension to make decomposable
  - One-dimensional throughput is not composable
  - Two-dimensional throughput-occupancy pairs are

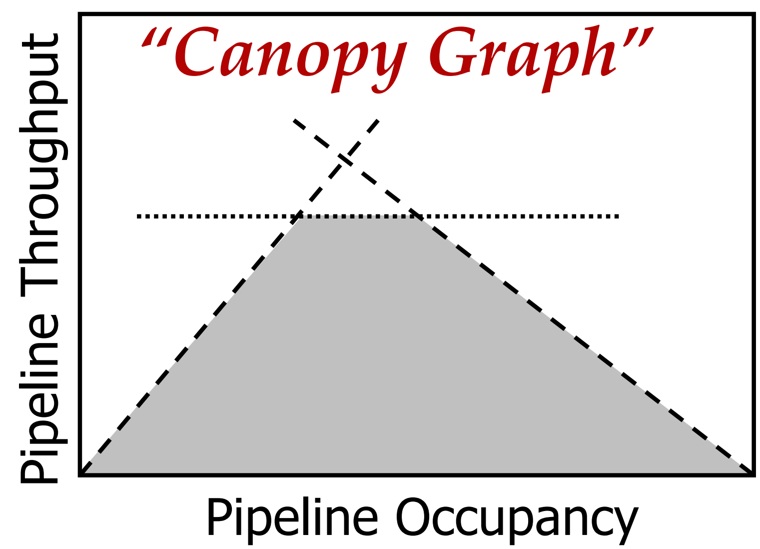




Performance Analysis: Method

- Modular method for performance analysis
  - Exploits <u>system hierarchy</u> with "divide-and-conquer" method
  - First: calculate canopy graph at each <u>leaf node</u>
    - Each leaf node is a single stage
  - Next: <u>compose</u> canopy graphs at each level of the hierarchy
  - Finally: canopy graph for <u>root node</u> gives system-level performance
- Requires composition algorithm for common circuit structures
  - **Parallel, sequential, conditional, and iterative**

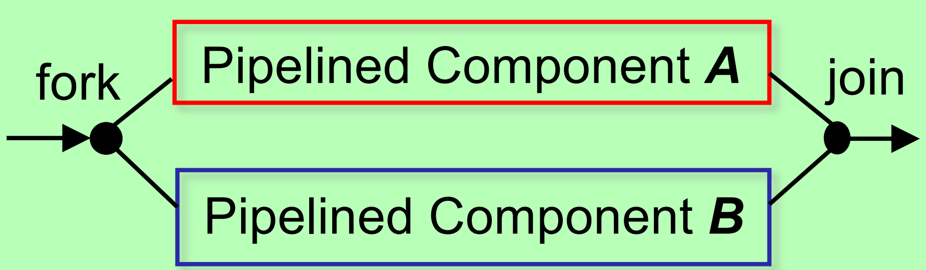




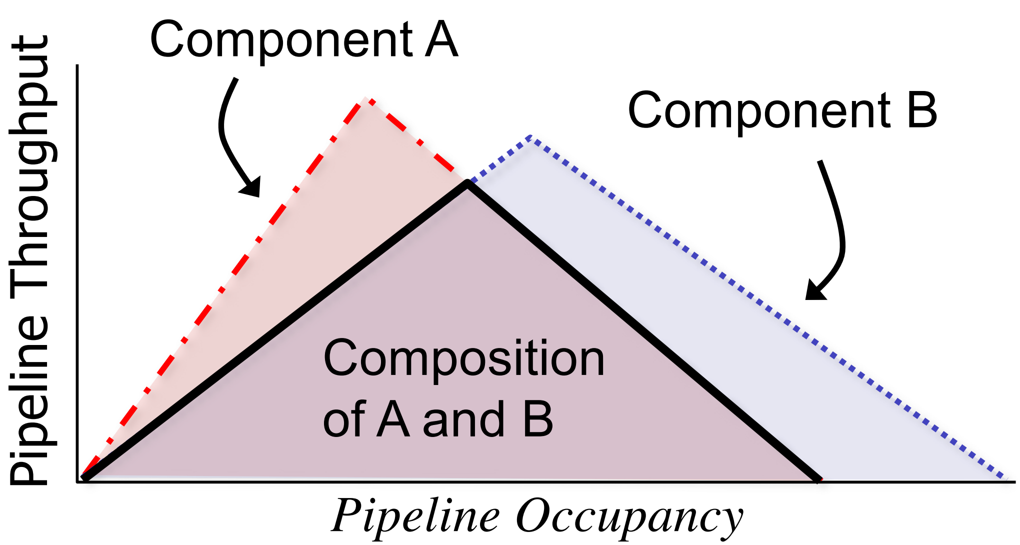
Gennette Gill and Montek Singh, "*Performance Estimation and Slack Matching for Pipelined Asynchronous Architectures with Choice,*" International Conference on Computer-Aided Design (ICCAD) (November 2008).

1) Parallel Composition

Parallel Composition of *A* and *B*



Canopy Graph of Composed Structure



Parallel structures [Lines98]

- Data copied at fork
- A and B compute in parallel
- Results recombined at join

Operation invariants under composition:

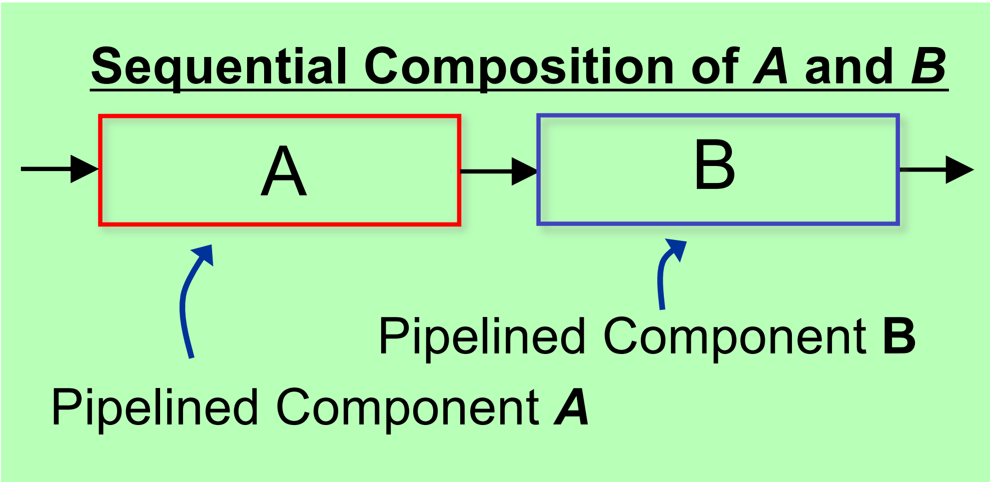
1) # of items in each branch equal
2) Branches have same throughput

Throughput of structure:

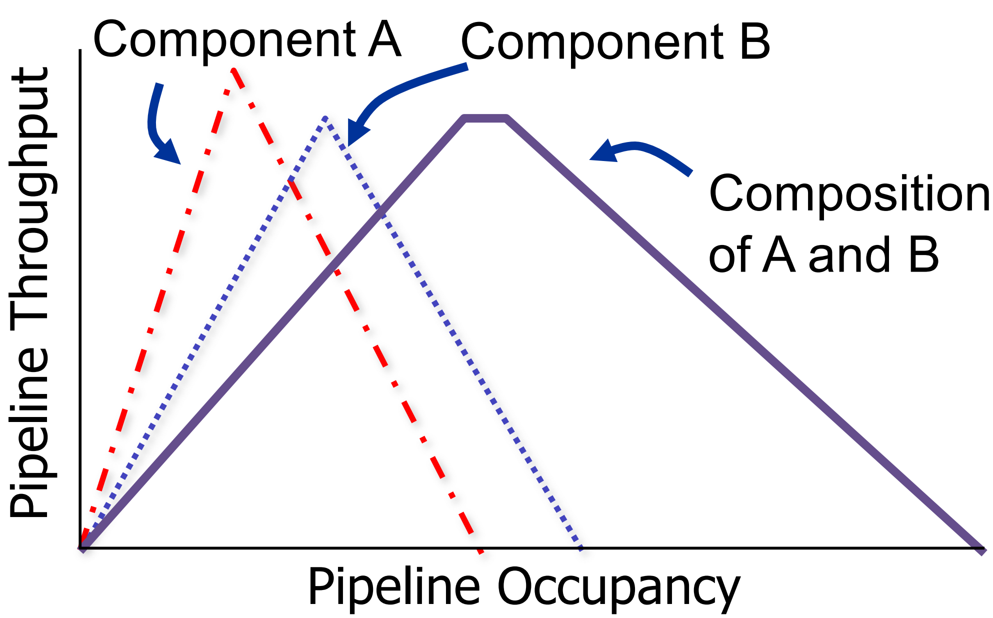
- Intuition: at each occupancy, throughput limited by slower branch

⇒ Intersection of canopy graphs of A and B

2) Sequential Composition



Canopy Graph of Composed Structure



Sequential Structures [Lines98]

- Data transmitted through A, then through B

Operation invariants under composition:

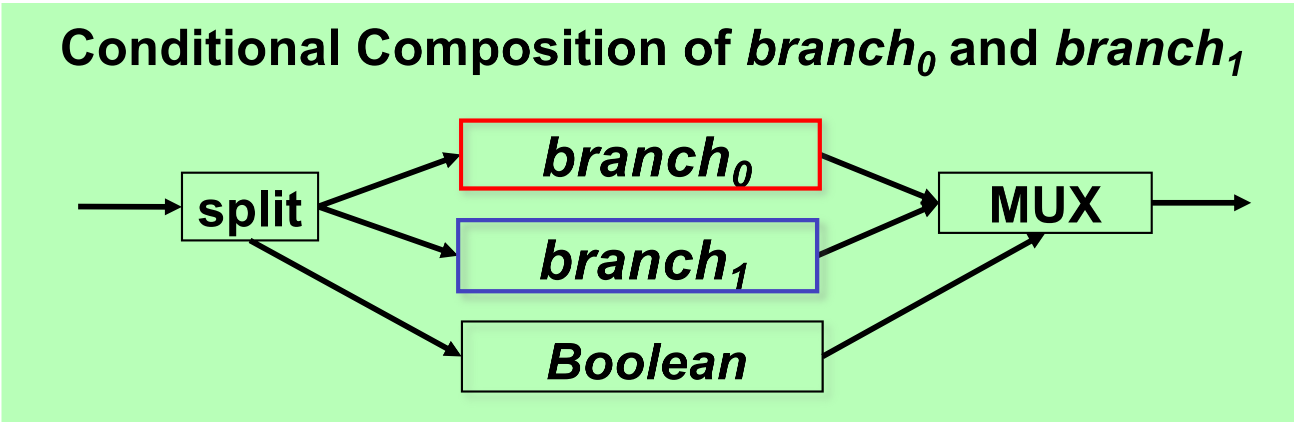
1) Find total # items: sum of # items in both pipes
2) Throughput *A* = Throughput *B*
3) Max throughput: limited by slower pipeline

Throughput of structure:

⇒ "horizontal sum" of canopy graphs of A and B

- At each throughput, add the occupancies of the two pipelines

3) Conditional Composition



Example:
p0 = 2/3 and p1 = 1/3

2 items enter branch$_0$
➔
1 item enters branch$_1$

- Operation invariants under composition:
  - Ratio of # items in each branch = ratio of probabilities

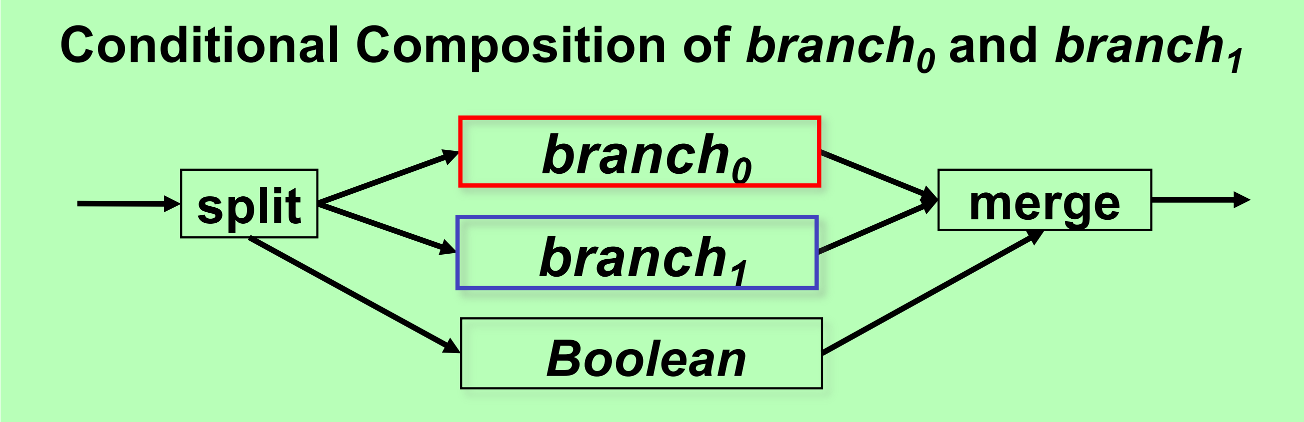
$$\frac{Occupancy_0}{p_0} = \frac{Occupancy_1}{p_1}$$

  - Ratio of throughput of each branch = ratio of probabilities

$$\frac{TPT_0}{p_0} = \frac{TPT_1}{p_1}$$

- Throughput of conditional structure:
  - Divide each branch's canopy graph by its probability p_i
  - Compute intersection of scaled canopy graphs
- "Bursty" inputs cause additional bottlenecks (see ICCAD-08 paper for details)

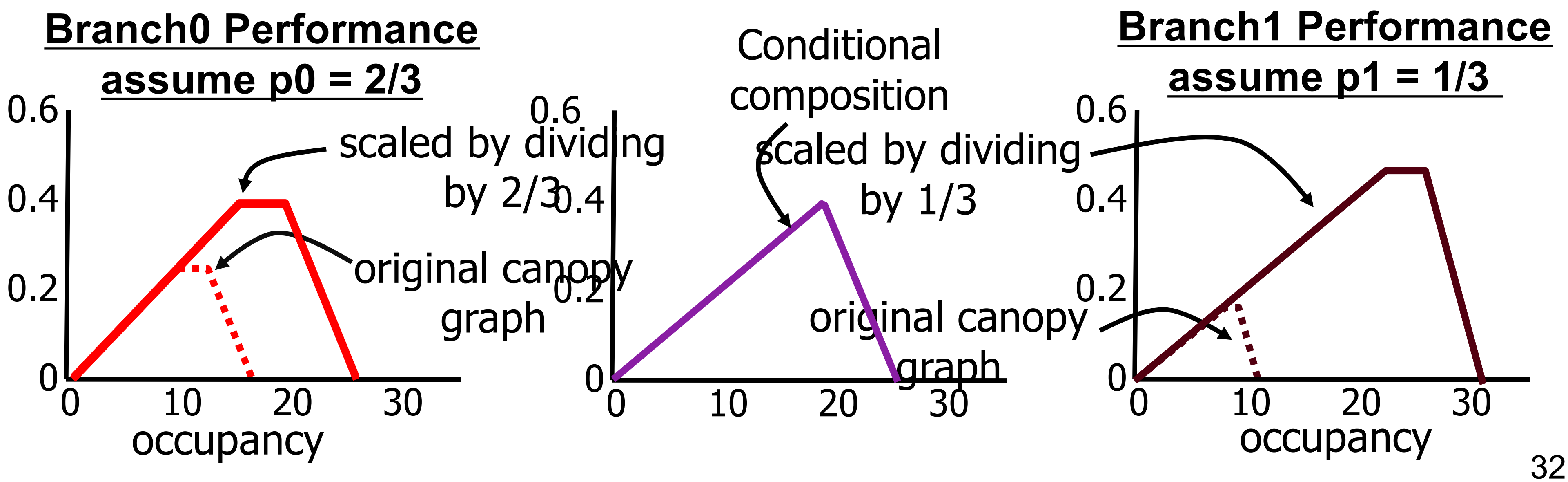
3) Conditional Composition (cont'd)



- **Step 1) uniform scaling**: enlarge each branch's canopy graph

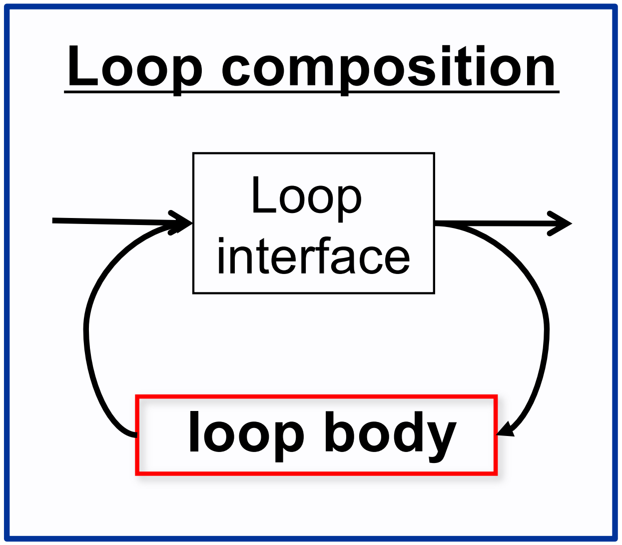
  Example: $\mathbf{p_0 = 2/3}$ and $\mathbf{p_1 = 1 - p_0 = 1/3}$

- **Step 2) intersection**: finds system-level performance

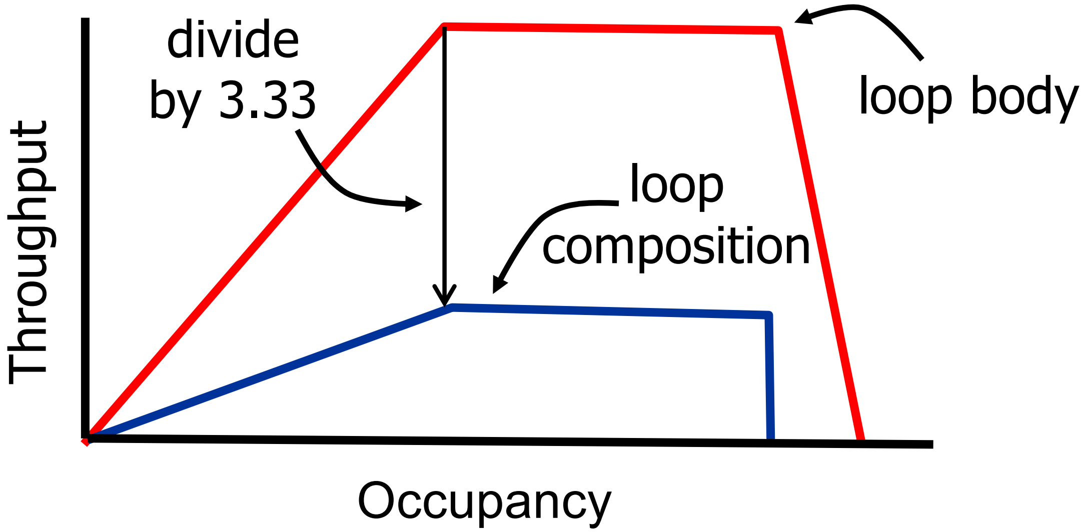


4) Iterative Loop Composition

- Operation invariants under composition:
  - Each item passes through the loop multiple times
  - Loop can handle multiple items simultaneously
- Throughput of composition
  - # data items processed <u>decreases</u> as iteration count increases
  - ⇒ Scale down based on expected number of iterations



Throughput vs. Occupancy
with Expected Iterations = 3.33



Analysis: Benchmark Examples

- Analysis algorithm demonstrated on 8 benchmarks
  - Chosen to represent a variety of circuit constructs

	Composition Type			
Example	**Parallel**	**Sequential**	**Conditional**	**Iteration**
CORDIC	✔	✔	✔	
CRC		✔	✔	
DIFFEQ	✔	✔		✔
GCD		✔		✔
Ray-tracing		✔	✔	✔
MULT	✔	✔		
JPEG	✔	✔	✔	✔

- Evaluated several circuit implementations of some
  - Naive implementation vs. hand-optimized version
  - Different choice models: uniform random vs. correlated

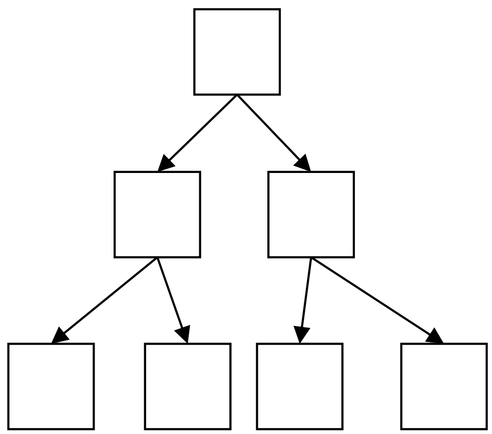
Performance Analysis: Results

- Total of 12 different circuit examples tested
  - Error < 4% for all examples, runtime ≤ 10 ms for all examples

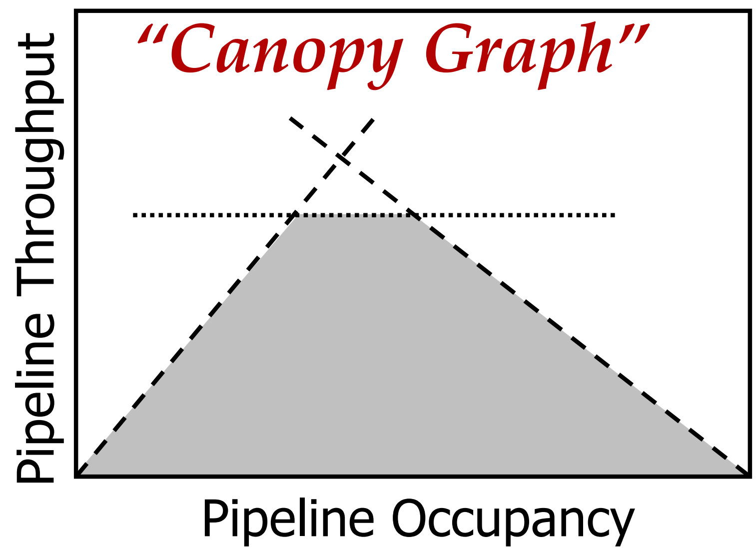
Example	Version	Size # stages	Throughput simulated	Throughput predicted	Error (%)	Runtime (ms)
CORDIC	*original*	31	90.9	90.9	0.00	~10
	optimized	44	167	167	0.00	~10
	bursty inputs	44	83	83	0.00	~10
CRC	*original*	23	292	286	2.05	~10
	optimized	27	352	357	1.42	~10
	bursty inputs	27	305	300	1.64	~10
DIFFEQ	*original*	10	18.3	18.2	0.55	<10
GCD	*original*	21	49	50	2.04	~10
Ray-tracing	*original*	21	161	167	3.73	~10
	optimized	166	222	222	0.00	~10
MULT	*original*	13	38.7	38.4	0.78	<10
	optimized	21	167	167	0.00	<10

Performance Analysis: Summary

circuit hierarchy



system-level performance



- **Fast**: restriction to hierarchical systems yielded fast runtimes
  - Divide-and-conquer approach with linear runtime
  - Modular canopy graph analysis for many constructs
    - Sequential, parallel, conditional, and loop
  - Expressive subset: modeled real-world applications
    - CORDIC, CRC, ray intersection algorithm, etc.
- **Accurate**: tested on several many non-trivial examples
  - Throughput estimates within 4% of simulation results

References

- Gennette Gill. Analysis and Optimization for Pipelined Asynchronous Systems. PhD thesis. UNC Chapel Hill. 2010.
- Gennette Gill and Montek Singh. *"Performance Estimation and Slack Matching for Pipelined Asynchronous Architectures with Choice."* ICCAD 2008.
- Montek Singh and Steven Nowick. *"MOUSETRAP: Ultra-High-Speed Transition-Signaling Asynchronous Pipelines."* ICCD 2001.
- Montek Singh and Steven Nowick. *"MOUSETRAP: High-Speed Transition-Signaling Asynchronous Pipelines."* TVLSI 2007.
- Gennette Gill, J. Hansen, A. Agiwal, L. Vicci and M, Singh. *"A High-Speed GCD Chip: A Case Study in Asynchronous Design."* ISVLSI 2009.