CE653 – Asynchronous Circuit Design

Instructor: C. Sotiriou

http://inf-server.inf.uth.gr/courses/CE653/

Contents

Pipeline Performance and Metrics

- Synchronous Design
  - Latency of a pipeline
    - Measured in terms of # of clock cycles
  - Throughput
    - Typically measured in terms of results per second
    - Inverse of Clock Cycle Time for systems that generate a result each cycle
- Asynchronous Pipelined Design (i.e., using pipelined handshaking)
  - Latency
    - Time between tokens consumed at inputs and generated outputs
    - Inputs tokens spread apart to avoid congestion slowing down results
  - Cycle Time
    - Taken as long-term average of time between successive output tokens
  - Throughput
    - Results (tokens) per second - inverse of cycle time
  - Data-dependent delays
    - Block-level delays and data-flow may be data-dependent

Forward Latency

- Forward latency (FL) – block level
  - Time between tokens consumed at inputs and generated outputs
  - Inputs tokens spread apart to avoid congestion slowing down results
  - May be data-dependent
- (Forward) latency – system
  - A sum of forward latencies through blocks
  - Must account for causality of output tokens within blocks
  - Earlier/latest arriving token may cause output token
  - I.e., notion of critical path exists

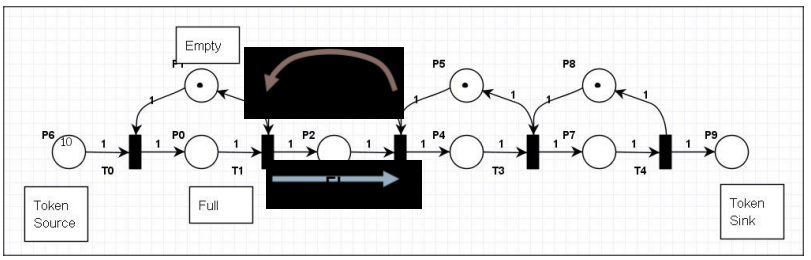
```
module OR(L1, L2, R);
 `INPORT(L1,1);
 `INPORT(L2,1);
 `OUTPORT(R,1);
 `USES_CHANNEL
parameter FL = 2;
reg d1, d2;
always
begin
    fork
     `RECEIVE(L1,d1);
     `RECEIVE(L2,d2);
      join
      #FL;
     `SEND(R,d1 | d2);
  end
endmodule
```

Local Cycle Time + Backward Latency

- Local cycle time
  - Shortest time to complete a handshake with its neighbours
    - Cycle may involve three neighbours for half-buffers
    - Lower-bound on performance
  - Equals FL + BL
- Backward latency (BL)
  - Time needed to reset before accepting new tokens
    - Time between generated output and earliest time of subsequent input

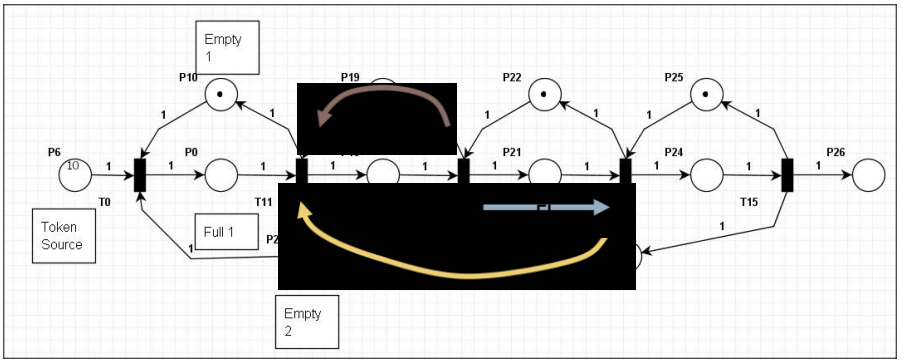
```
module BUF(L, R);
parameter width = 8;
parameter FL = 2;
parameter BL = 4;
 `INPORT(L,width);
 `OUTPORT(R,width);
 `USES_CHANNEL
  reg [width-1:0] d;
  always
  begin
      `RECEIVE(L,d);
      #FL;
      `SEND(R,d);
      #BL;
  end
endmodule
```


Full-Buffer PTnet Model



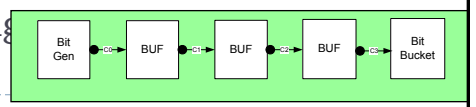
- How do FL/BL determine pipeline performance?

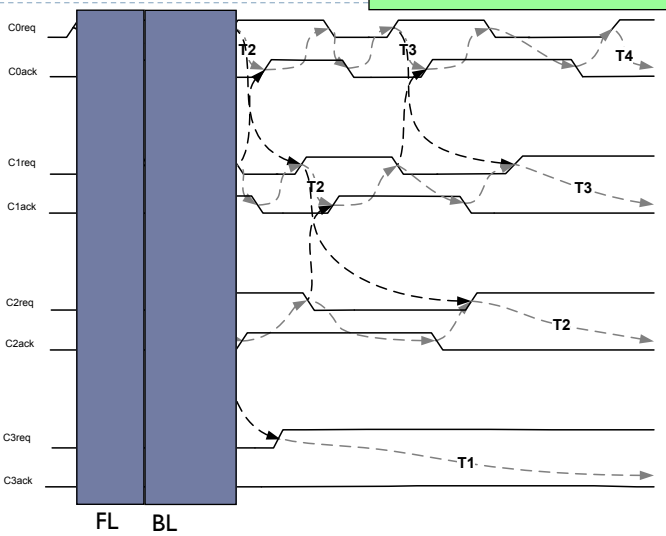
Half-Buffer PTnet Model



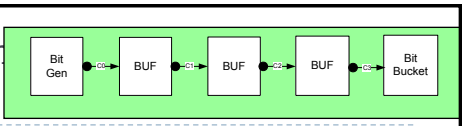
- What parameters determine the performance in the Half-Buffer case?

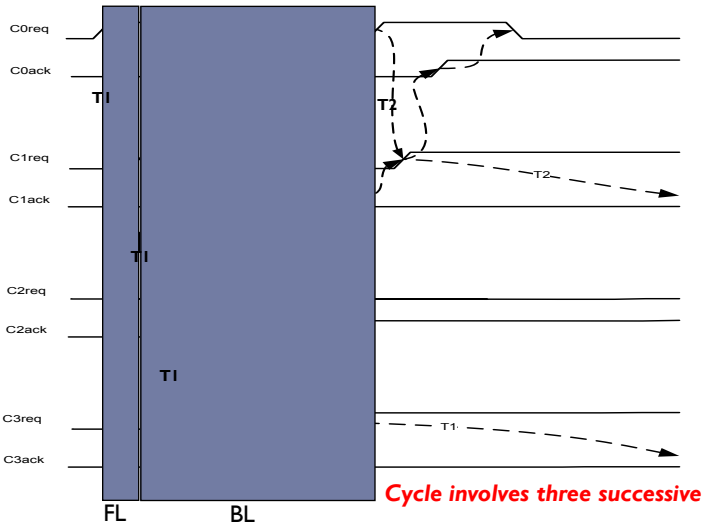
Full-Buffer Handshaking & Backward Latency





Half-Buffer Handshakin Backward Latency

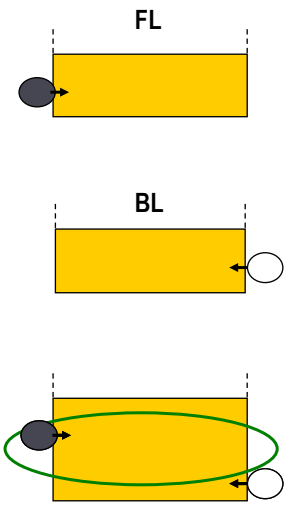




Cycle involves three successive blocks

Performance Metrics - Animated

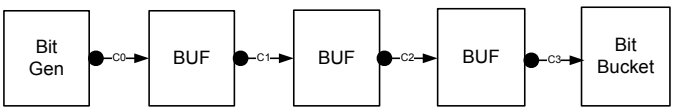
- Forward Latency (FL)
- Backward Latency (BL)
- Local Cycle Time
  - FL+BL



Dynamic Pipeline Behaviour

- Cycle Time
  - *T* = ***FL + BL*** (simplest case)
- Latency
  - input to output delay = ***N * FL***
- Throughput
  - # of tokens flowing per unit time, generally = ***1/T = 1/(FL + BL)***
  - Depends on throughput of sender/receiver
- Static Slack or Static Token Occupancy
  - Maximum token capacity of the pipeline
- Spread
  - distance between successive tokens in a full pipeline
  - token distance travelled for 1 *T* = ***(FL + BL)/FL***
- Dynamic Slack or Dynamic Occupancy
  - Average token capacity in the pipeline during operation
  - N*1/Spread = ***(N*FL)/(FL + BL)***

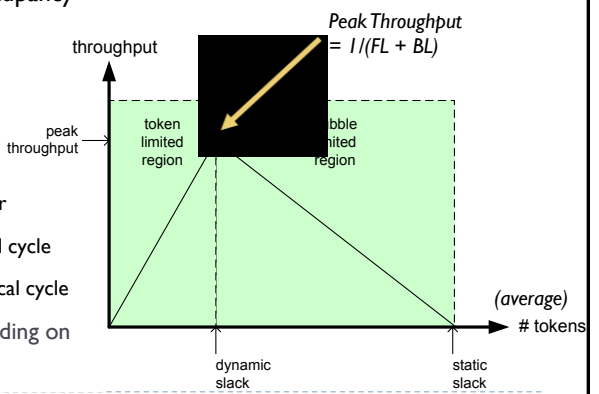
Dynamic Slack



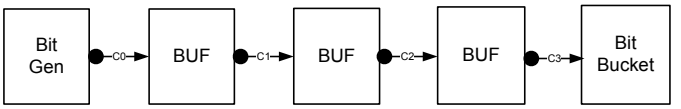
- Dynamic Slack or Dynamic Occupancy Formula for N buffers:

$$\frac{N * FL}{FL + BL} = \frac{N}{1 + \frac{BL}{FL}}$$

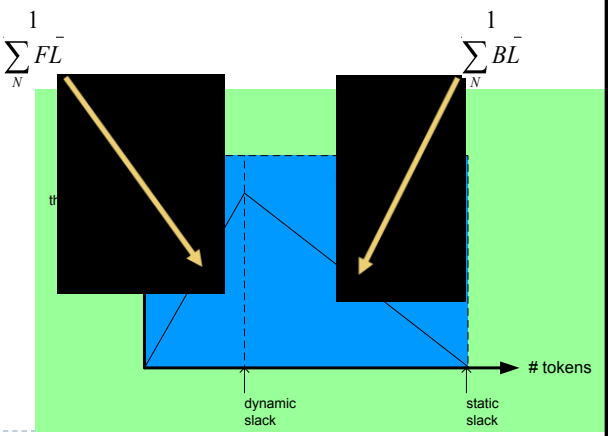
- Assumptions:
  - Tokens not stalled by buffers or BitBucket resetting
  - Tokens inserted at rate of local cycle time (*FL + BL*)
  - Tokens consumed at rate of local cycle time
- Can be as small as N/9 (depending on circuit type)



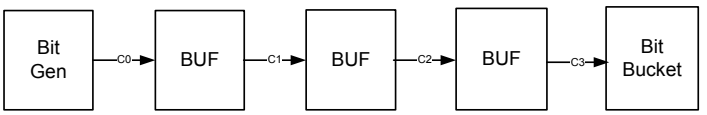
Throughput vs. Tokens Graph

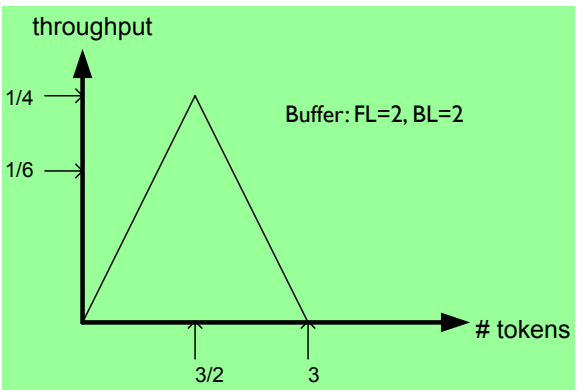


- Throughput is zero
  - When no tokens in pipe
  - When pipeline is full of tokens
- Peak throughput in-between
  - Token limited region
    - Faster BitGen improves throughput
  - Bubble limited region
    - Faster BitBucket improves performance



Concrete Example: 3-stage Pipeline

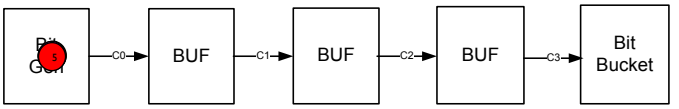




Example: Pipeline Performance

- **Slow left environment**
  - Bitgen: LCT = 6
  - Buffer: FL=2, BL=2
  - Bucket: LCT = 2

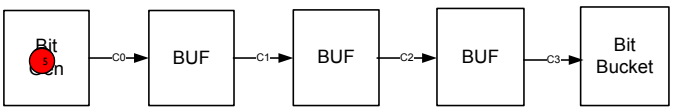
t=0: Token 1 generated
t=6: Token 2 generated
t=12: Token 3 generated
t=18: Token 4 generated
t=24: Token 5 generated



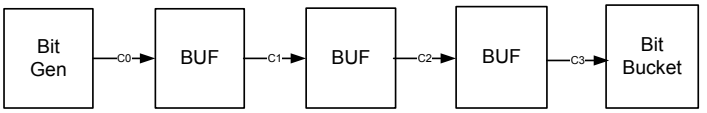
Example: Pipeline Performance

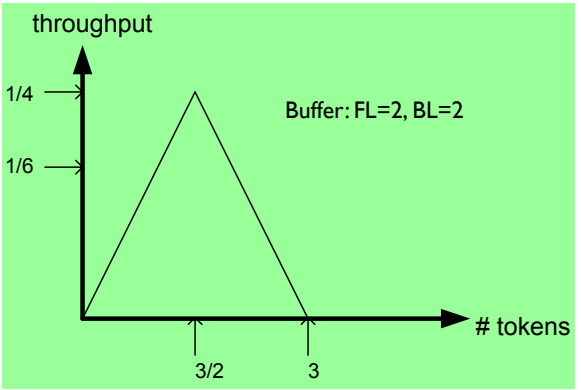
- **Slow right environment**
  - Bitgen: LCT = 2
  - Buffer: FL=2, BL=2
  - Bucket: LCT = 6

t=6: Token 1 consumed
t=12: Token 2 consumed
t=18: Token 3 consumed
t=24: Token 4 consumed

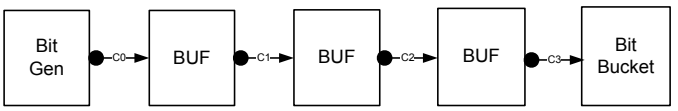


Concrete Example: 3-stage Pipeline

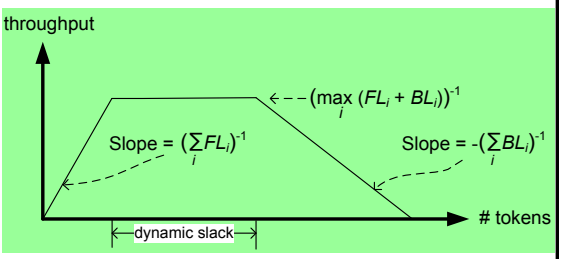




Non-homogeneous Pipelines



- *Peak throughput is limited by:*
  - **Worst-case (largest)** local cycle time
- *Dynamic slack/Occupancy*
  - Becomes a range of # of tokens
- *Data-limited slope*
  - Inverse of sum of FLs (?)
- *Bubble-limited slope*
  - Inverse of sum of BLs (?)



Pipeline Rings

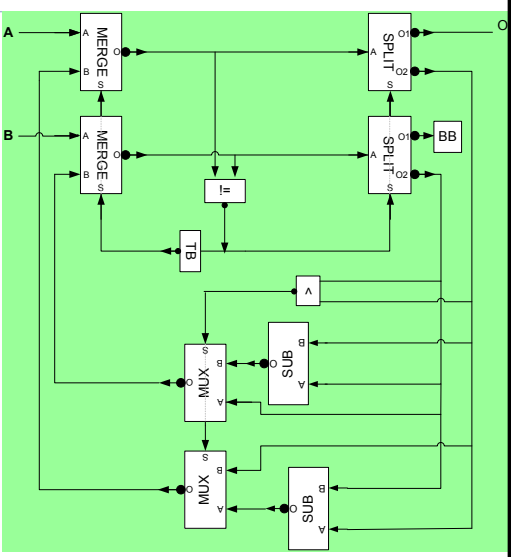
- Definition (informal)
  - Pipeline buffers configured in a loop
  - Can be combined with forks, joins
- Used in implementing iterative algorithms
  - Each iteration implemented by a token traversing the loop
- Multiple tokens in loop possible
  - Each token independent of others
  - Implements "multi-threading", *i.e.* **pipelining**

```
function gcd(A, B)
    while A ≠ B
        if A > B
            A := A - B
        else
            B := B - A
    return O = A
```

Euclid's Algorithm for Greatest Common Divisor (GCD)

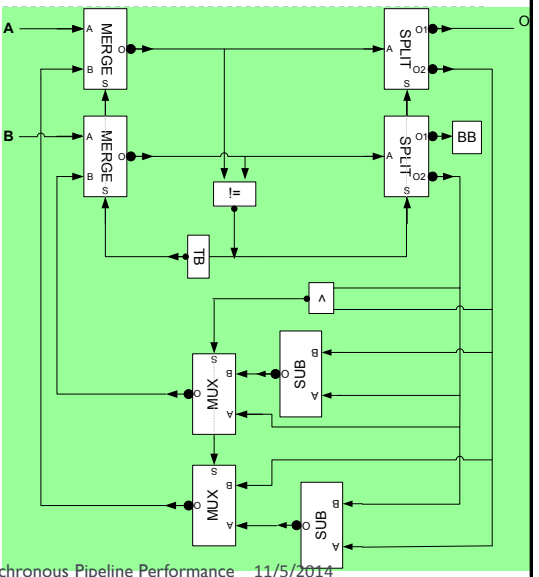
A GCD Implementation

- Implementation Notes
  - MUXs are same as MERGEs but consume both input tokens
  - TB is a token buffer
    - Generates a token on initialization with configurable value
    - Acts as a buffer afterwards
  - FORK cells implied by branching channels (for clarity)
  - All cells use pipeline handshaking
- Architectural Feature
  - Contains many pipeline rings
  - Single Token around the ring
- Does one GCD at a time!



A GCD Implementation

- Operation
  - TB generates tokens to select input tokens come in on PIs A and B
  - Tested for equality which controls how they are routed
    - If != routed to SUBs & '<'
    - Otherwise, A is routed to output
  - SUBs concurrently generate differences.
  - Specific difference routed back to merge controlled by '<' and MUXs



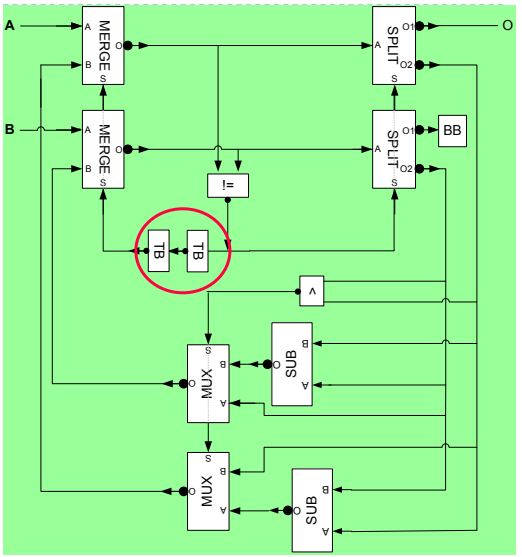
Token Buffer – VerilogCSP Model

```
module TOK_BUF(L, R);
parameter width = 8; parameter init = 8'b0;
 `INPORT(L,width);
 `OUTPORT(R,width);
 `USES_CHANNEL
parameter FL = 2; parameter BL = 4;
reg [width-1:0] d;
initial
begin
    `SEND(R,init);
end
  always
  begin
      `RECEIVE(L,d);
      #FL;
      `SEND(R,d);
      #BL;
  end
endmodule
```

- **Initial block**
  - Mechanism to send out initial token
- Init
  - Value of initial token sent out
  - Configurable via Verilog *parameter* feature
  - After initial block, never used again
- Always block
  - Performs steady-state behavior
  - Just like BUF cell

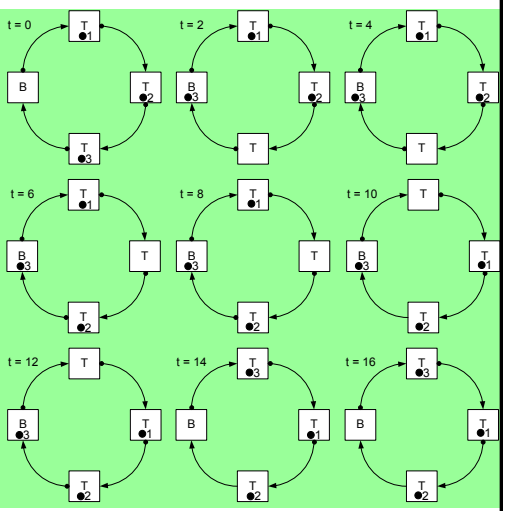
"Multi-Threading"/**Pipelined** Variant

- Add another Token Buffer (TBs)
  - Pipelines design by enabling two sets of tokens in loop simultaneously
    - Second set enters immediately, before first set completes algorithm
  - Each set represents a thread and moves around loop independent of other set
  - No interference due to handshaking
- The Purpose
  - Multiple instances of GCD algorithm solved simultaneously
  - Can improve throughput
    - Tokens on O per second
- Completion may be **Out-Of-Order (OOO)**
  - depending on # of iterations per input
    - Use ROB or problem/tag



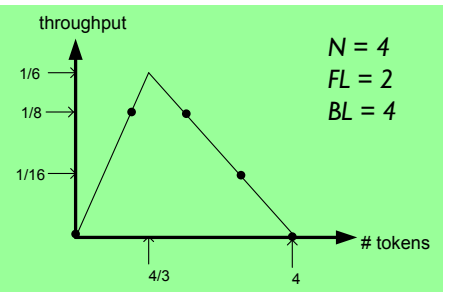
Pipeline Loops – Bottleneck Example

- Buffer Performance
  - Assume: FL = 2; BL = 4
- Ring Architecture
  - Three tokens with four full-buffers
  - N.B. # of tokens in ring is constant
- Performance Analysis
  - Each token can move forward every 3 * BL = 12 time units
    - Example: token #3 moves forward at t = 2 and t = 14
  - Each token completes an iteration every 12 * 4 = 48 time units
  - Three tokens do this concurrently
    - Completing 3 iterations every 48 time units
  - Yields, throughput of 1 iteration every 48/3 = 16 time units



Pipeline ring – Performance Analysis

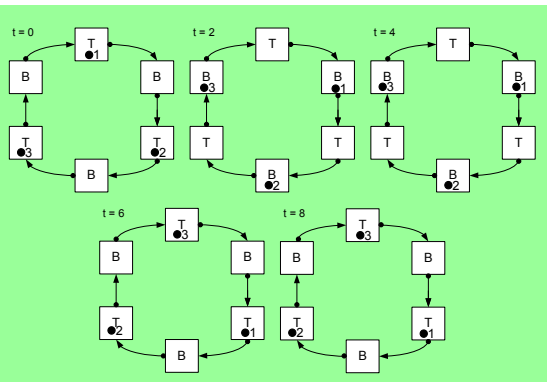
- Intuition
  - Pipeline ring is like a linear pipeline with output channel tied to input channel
- Optimal performance
  - No pipeline buffer starved
  - No pipeline buffer stalled
- Dynamic slack/Occupancy (for full-buffers): $\frac{N}{1+\frac{BL}{FL}}$



N.B. Performance is at discrete points only because cannot have a fractional number of tokens in pipeline ring

Pipeline Loops – Improving Performance

- Bubble-limited loops
  - Can improve performance by adding pipeline buffers
- Intuition
  - Bubbles need to flow backwards less distance for tokens to flow forward
- Data-limited loops
  - Increase multi-threading
  - Shorten loop latency

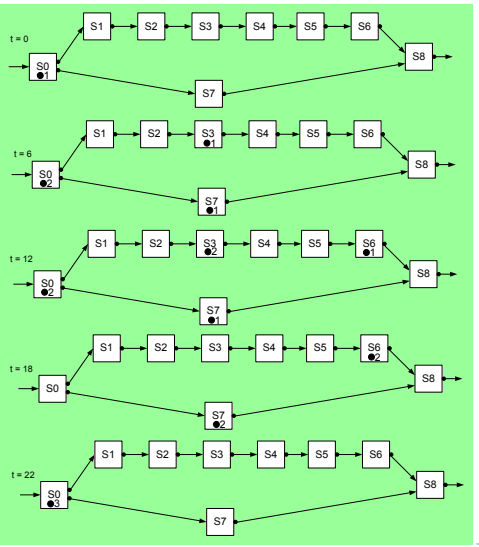


N = 6
FL = 2
BL = 4

Throughput = 8

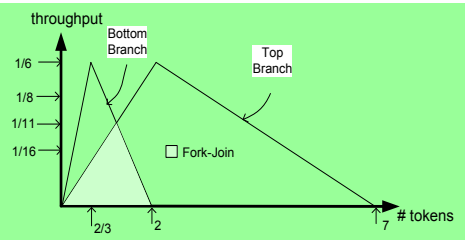
Fork – Join Pipeline - Bottlenecks

- Slowest fork-join path determines input-output latency

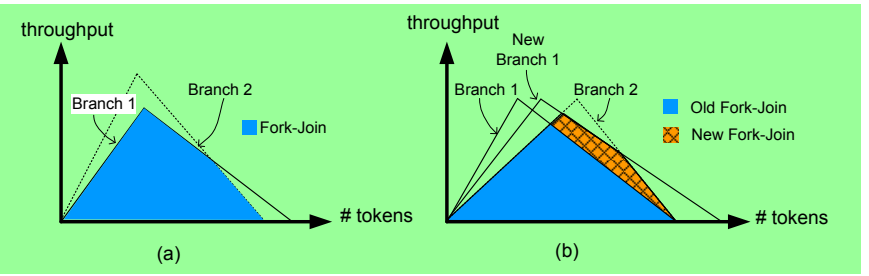


Fork – Join: Performance Analysis

- Intuition
  - Number of tokens in each branch of fork-join is identical
- Throughput versus # tokens
  - Lower-bounded by triangle graphs of individual pipelines
- Performance characteristics
  - Static slack is minimum of two individual pipelines
  - Peak throughput can be lower than either of individual pipelines



Fork – Join: Other Characteristics



- Shorter pipeline (lower static slack) may not be bottleneck
  - Can happen if peak throughout of shorter pipeline is larger than longer pipeline
  - See Figure (a) above
- Equal static slack is not always optimal
  - *i.e.*, adding buffers may improve peak throughput despite causing a static slack imbalance
  - See Figure (b) above

Summary and Conclusions

- Performance of asynchronous pipelines is complex
  - Largely due to presence of backward latency of pipeline buffers
- Throughput versus # of tokens graph
  - Effective way to analyze simple pipeline structures
- Provides good intuition of many issues
  - More complex pipeline structures popular
  - For example, forks / joins / conditional in pipeline loops
- *e.g.*, GCD example
  - Need more powerful methods of analyzing and optimizing pipelining
  - Covered later on using performance models