UNIVERSITY OF THESSALY

TECHNION
Israel Institute of Technology

USC Viterbi School of Engineering
University of Southern California

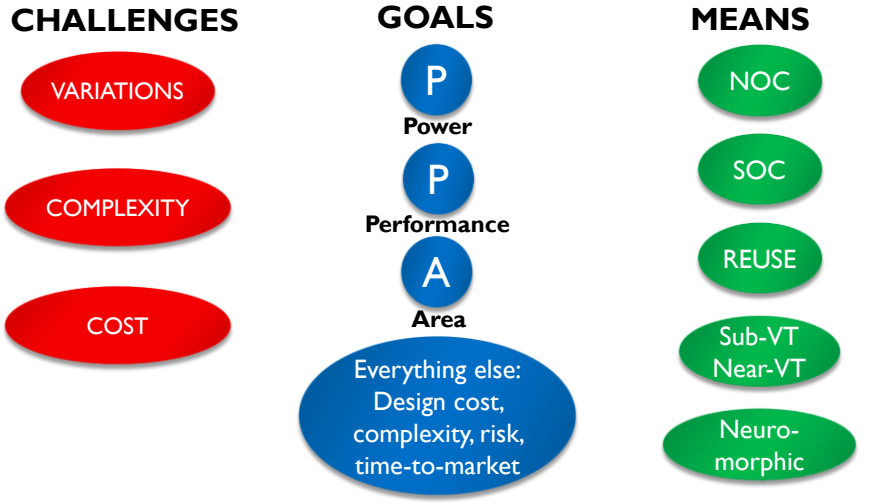
Asynchronous Design Seminar at University of Verona – Lecture Notes 1

Asynchronous Design Fundamentals

Lecture Notes 1

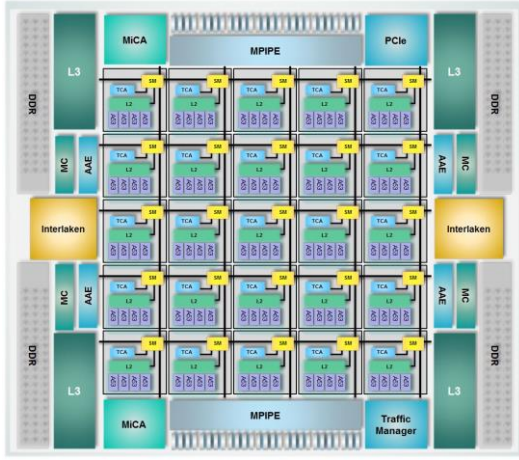
- Motivation for Asynchronous Design
  - Challenges in Traditional Synchronous Design
  - Benefits of Asynchronous Design
  - Startup and other Industrial Efforts (Past and Current)
  - Challenges and Opportunities Moving Forward
- Data Channels and Channel Based Design
- Handshaking and its impact
- Asynchronous Pipelines
  - Data Token Movement and Control
  - Characteristics and Performance
- Modelling using PTnets
- Latch Controller Design

Motivation



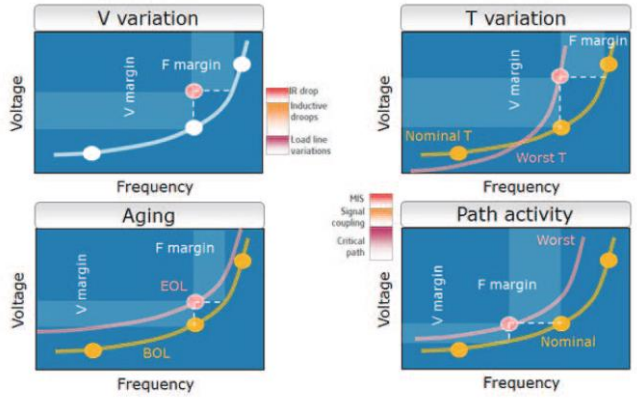
Motivation I: Network on Chip

- Scale of chip / designs increasing
- Design efficiency demands network on chip
- Synchronous NoCs challenging
  - Span large portion of chips
  - Hard to clock gate
  - Synchronization penalties
  - Impacted severely by global PVT variations
  - Buffering to reduce contention is costly



Tilera's TILE-Mx100

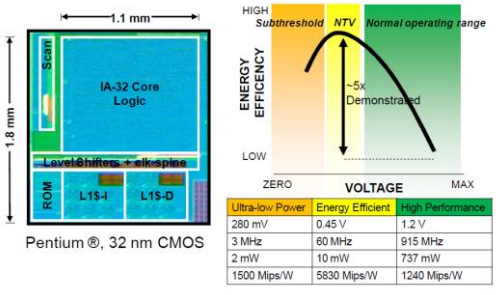
Motivation II: Dynamic Variations



[De, Asian IIEEE ASSCC 2014]

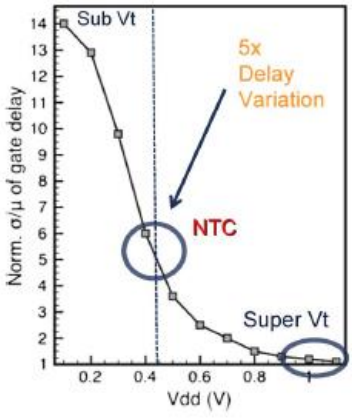
- Dynamic variations force margins on clock frequency

Motivation III: Near Threshold Computing



Ultra-low Power	Energy Efficient	High Performance
280 mV	0.45 V	1.2 V
3 MHz	60 MHz	915 MHz
2 mW	10 mW	737 mW
1500 Mips/W	5830 Mips/W	1240 Mips/W

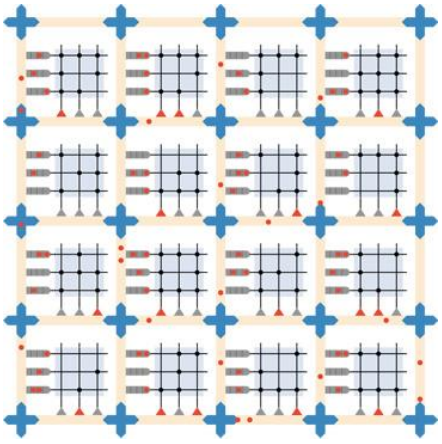
[Borkar, et al. DAC 2012]



[Dreslinksi et al., IEEE Proc. 2010]

- NTV computing provides ~5X energy efficiency
- But margins grow by ~5x forcing constrained libraries and flows

Motivation IV: Neuromorphic Computing

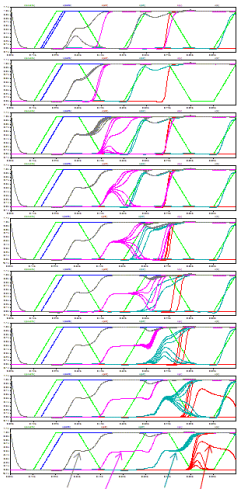


- Neuromorphic cores
- Large scale
  - hard to route a global clock everywhere
- Low activity
- Asynchronous becoming default approach

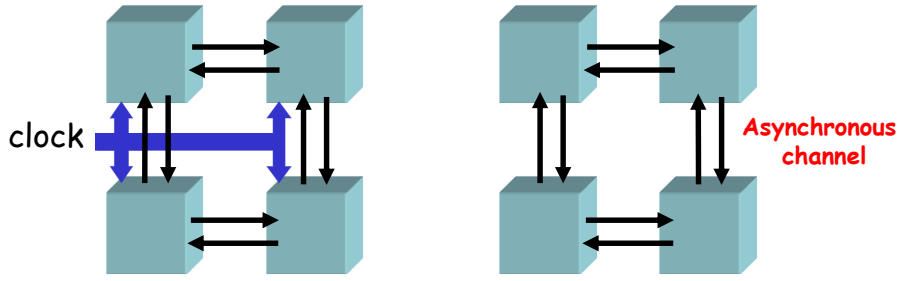
IBM

Motivation V: Metastabilty

- Multiple clock domains, and dynamic frequency scaling
- Crossings are hard, challenging, risky
- Subject to variations and modes of use
- Understanding comes from theory of asynchronous circuits
- Solutions
  - Synchronizer design and verification
  - Arbiters
  - Encapsulate the issues



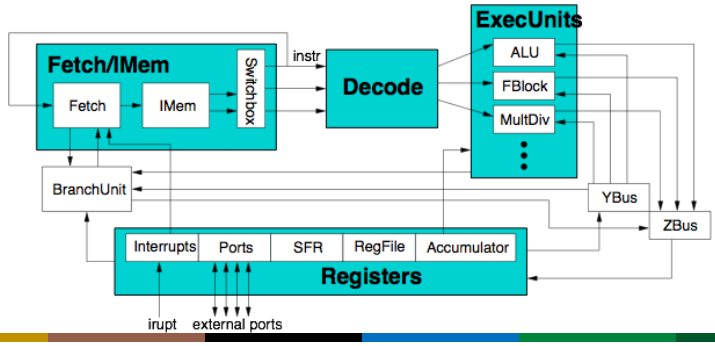
The Asynchronous Alternative



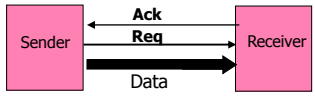
Synchronization and communication between blocks implemented with asynchronous channels that send and receive tokens

Concurrent Communicating Hardware

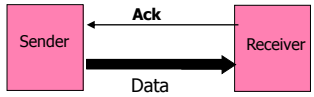
- System is a collection of **Processes** linked by **Channels**
- Channels pass messages with **guaranteed delivery**
- Processes synchronize
- Processes can be **decomposed** into smaller processes



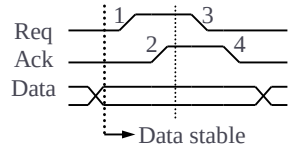
Asynchronous Channels

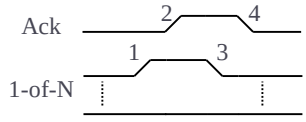


Bundled-Data Channel



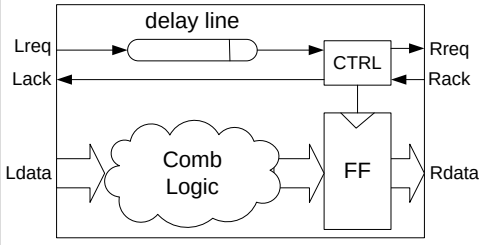
Dual-Rail (1-of-N) Channel

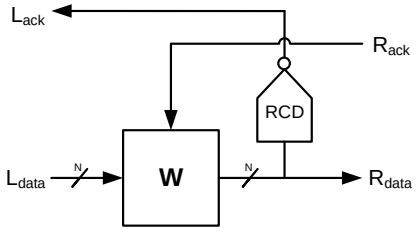




1 of the N wires rises
(N-1 remains zero)

Asynchronous Blocks





- Bundled-Data
  - Delay line Matched to Worst-Case Combinational Logic Delay
  - May be Post-silicon Tunable
  - Uses Standard-Cell Libraries
- QDI
  - Data and Completion integrated
  - Relies on Completion Sensing Logic
  - Uses Proprietary Cell Libraries
    - Domino-Style Dynamic Logic

Performance Advantages

- Freedom from global clock(s)
  - Local versus global control supports higher frequencies with less effort
  - Flexible pipelining – block- to gate-level
  - Efficient Support for Globally Asynchronous Locally Synchronous (GALS) Design
- Average-case delay
  - Architectural – Optimize Architecture for the Common Data Flow
  - Micro-architectural – Optimize Blocks for Average-case Input Data
  - Gate-level – Control Responds to Average Delays (*e.g.* Metastability)
  - Process – Design Adapts to Process Variation (*e.g.* Tunable Delay Lines)
  - Voltage – Design Adapts to Voltage (*e.g.* Dynamic Voltage Scaling)
  - Temperature – Design Adapts to Temperature (*e.g.* Wide Temperature range)
  - Also leads to Power Advantages!

Power Advantages

- Automatically adapts to operating conditions and variations
  - Combines: voltage/frequency scaling, clock-gating, tuning
  - More power savings, than the sum of all these techniques
- Clock-gating inherent in most asynchronous designs
  - Only expend energy in blocks when blocks are used
  - Inherent in architectural level rather than as an after-thought
- Blocks can be designed at ideal throughput
  - Rather than dictated by global clock period
  - Less frequently used blocks can be designed at lower throughput
- Immediate start-up (no need to wait for clock to stabilize)
  - Go quiet; Run fast; Go quiet again

Other Advantages

- Reduced Electromagnetic Noise
  - No global clock
  - Frequency spread out much more
  - Co-locate with sensitive analog
  - Less noise - does not effect analog circuitry
- Ease of Modular Composition
  - Reduces design complexity, cost, risk, iterations, time to market
  - Supports GALS design for multi-frequency disparate SoC designs
  - Handshaking offers plug-and-play IP

Asynchronous Challenges (1 of 2)

- CAD tools
  - Support from major EDA companies
- Cell/IP Core Libraries for Asynchronous Design
  - Most design styles could benefit from at least a few asynchronous cells
    - e.g., C-elements, Mutual Exclusion Elements
  - Some design styles heavily rely on custom gates
    - QDI PCHB, NCL
- High-performance asynchronous design
  - Some blocks can be 2-5x larger due to dual-rail design
  - May consume more peak power than desired
- Low-power asynchronous design
  - Can be slower than desired, if control overhead not managed

Asynchronous Challenges (2 of 2)

- Debug
  - Circuit can't be slowed via clock to aid in debugging
  - …but the circuit can be single-stepped more easily!
- Asynchronous test
  - Testers geared toward synchronous
  - Standards do not exist and test methodologies still evolving
  - Automatic test pattern generation in infancy

Asynchronous Commercialization Efforts (1 of 2)

- Fulcrum Microsystems (`www.fulcrummicro.com`)
  - High-performance computing and networking markets
  - Founded out of Caltech in 2000, Sold to Intel in 2011
  - Several asynchronous products shipping, Full-custom flow still in use today
- TimeLess Design Automation
  - High-Performance ASIC Flow for Asynchronous Design
  - Founded out of USC in 2008
  - Sold to Fulcrum Microsystems in 2010
  - Semi-custom flow still in use today
- Achronix (`www.achronix.com`)
  - High-performance async FPGA core with synchronous interfaces
  - Founded out of Cornell research in 2006
  - Speedster®22i HP is shipping and includes fine-grain asynchronous pipelines

Asynchronous Commercialization Efforts (2 of 2)

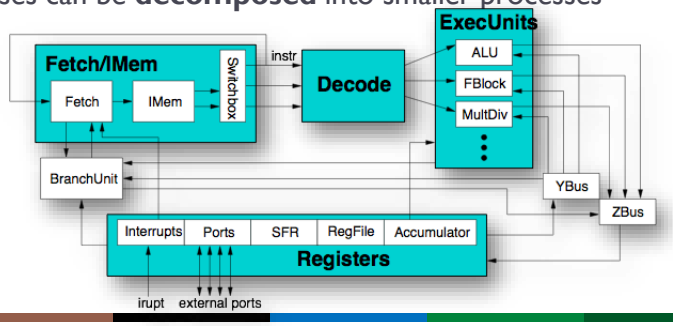
- Tiempo (`www.tiempo-ic.com`)
  - IP Cores and ASIC Flow (Power/Performance Tradeoff)
  - Smartcards …
- vSync Circuits (`www.vsyncc.com`)
  - EDA and IP for Multi-Clock Domain design integration and verification
- Reduced Energy Microsystems (`remicro.systems`)
  - Low Power ASIC microprocessors using asynchronous resiliency
  - Founded in 2015 – Newest kid on the block
- IBM
  - Neuromorphic True North core based on QDI technology (EXPAND)
- Several Past Start-ups
  - Handshake Solutions,
  - Silistix,
  - Elastix, Nanochronous,
  - Cogency

Channel-Based Design

Top-Level View of Asynchronous Circuits

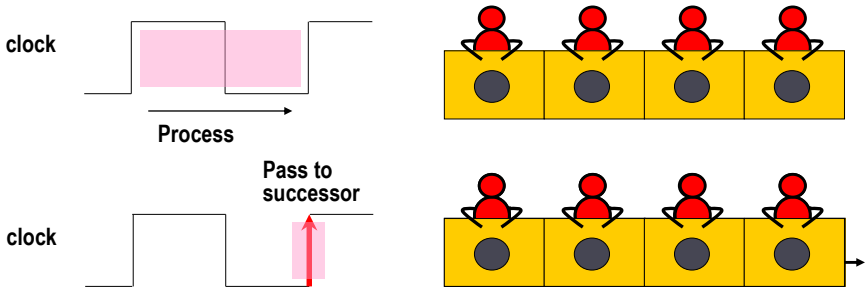
Hardware Abstraction

- System:
  - Collection of "**Processes**" linked by **Channels**
  - Channels pass messages with **guaranteed delivery**
  - Processes synchronize
  - Processes can be **decomposed** into smaller processes



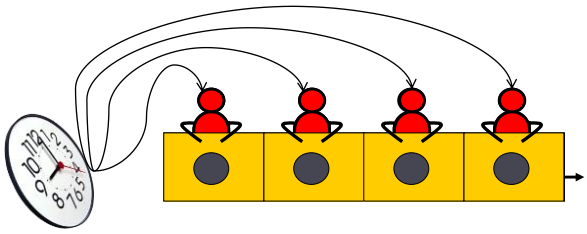
Synchronous Version

- In case of edge triggered stages
  - During the cycle: Process
  - At the edge of the clock: Pass to successor



Synchronous Version

- Central synchronizer
  - `SYNC(clk)



```
module  ff(clk, left, right)
input           clk;
input             left;
output reg      right;

always
begin
  `SYNC(clk);
   right = left;
end
endmodule
```


Synchronous FF Stage

- Abstract synchronization
  - `SYNC(clk)

```
module ff(clk, left, right)
input     clk;
input               left;
output reg          right;

always @(posedge clk) begin
 right = left;
end
endmodule
```

```
module  ff(clk, left, right)
input           clk;
input             left;
output reg      right;

always
begin
  `SYNC(clk);
 right = left;
end
endmodule
```

Asynchronous Version

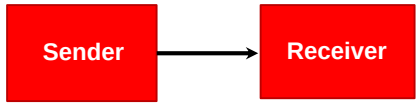
- Distributed Synchronization
- Sender
  - Provides data
  - Synchronize
- Receiver
  - Synchronize
  - Samples data

```
module Sender(right)
output reg      right;
output reg      data;

always
begin
 right = data;
`SYNC(right);
end
endmodule
```

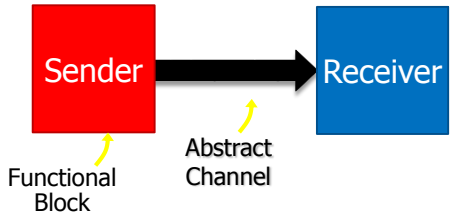
```
module Receiver(left)
input           left;
output reg      data;

always
begin
 `SYNC(left);
  data= left;
end
endmodule
```

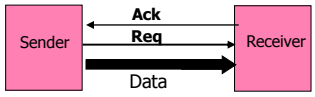


Asynchronous Channels

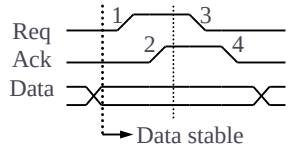
- **Channel:** A bundle of wires and a protocol for communicating data/control called a **token**
  - Data/control encoding: Dual-rail or Single-rail encoded
    - tells what the data is, and when it is valid
  - Communication protocol: Handshake over request/acknowledgement wires

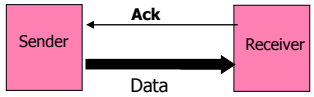


Asynchronous Channel Types

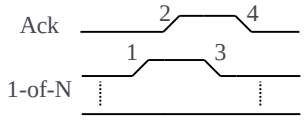


Bundled-Data Channel



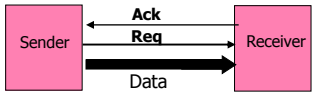


Dual-Rail (1-of-N) Channel

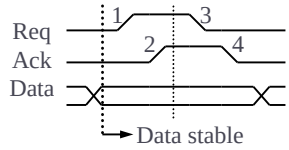


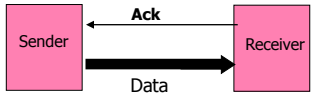
1 of the N wires is risen
(N-1 remains zero)

Asynchronous Channel Types

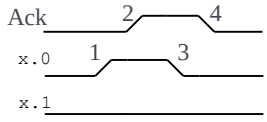


Bundled-Data Channel



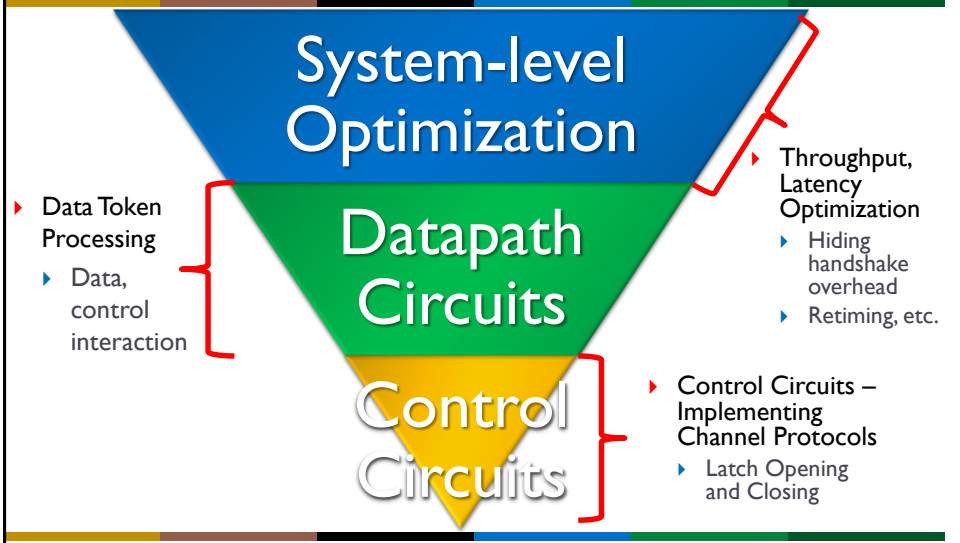


Dual-Rail (1-of-2) Channel



1 of the 2 wires rises

Asynchronous Design Levels of Abstraction

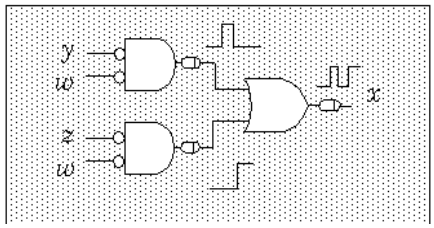


Asynchronous Design Levels

- **Block-Level Design – Micro-architectural Level**
  - **Control Circuits – Implementing Channel Protocols**
    - Handshake-based Token Movement, *i.e.* H/S Protocol Implementation
    - Latch Opening and Closing
    - Synchronization of Multiple Requests and Acknowledgments, … etc.
  - **Datapath Circuits – Data Token Processing**
    - Asynchronous Implementation of Combinational Clouds and Interaction with the H/S-based Control Circuits
      - Bundled-data, *i.e.* delay-element based
      - Data-Dependent Latency – Multiplexed Delay, based on Operands/Operation
      - Dual-Rail/Indicating Logic (Encoded)
- **System-level Optimization – Architecture Level**
  - Optimizing Token Cycle Time, Throughput (Tokens/sec)
  - Trading-off Token Latency for Token Throughput
  - Hiding Handshake Delays to achieve comparable cycle time to sync.
  - Retiming, *i.e.* Splitting Combinational Logic, adding Token Buffers (Latches)

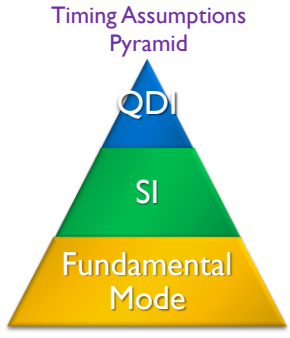
Logic Hazards (Glitches)

- Synchronous Circuits
  - Glitches tolerated because outputs sampled only after signals settle
  - Clocking constraints
    - Clock edge occurs only after data settles
    - Limits clock frequency
- Asynchronous Circuits
  - Control circuits
    - Avoided completely
    - Hazard-free logic synthesis techniques
  - Datapath (Either)
    - Outputs sampled after signal settles OR
    - Avoided completely

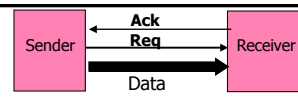


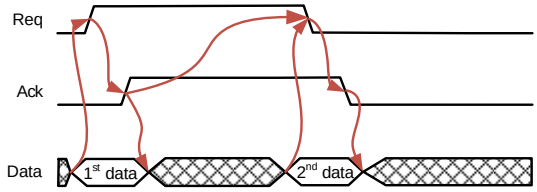
Asynchronous Circuits - Classes

- Timing Model (or Class) is used to define specific timing assumptions with respect to correct circuit operation
  - **DI**
    - Arbitrary gate and wire delays (unbounded)
  - **QDI**
    - DI except for Isochronic Forks
      - No need to acknowledge fanouts
  - **SI** (Muller) circuits
    - Arbitrary gate delay no wire delay
    - Only applicable to small-scale control circuits
  - **Fundamental Mode** (Huffman) circuits
    - Outputs and Local State (internal) stabilize
      - before a new input change from the circuit's environment



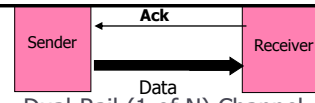
2-Phase Bundled-Data

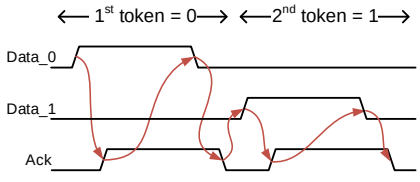




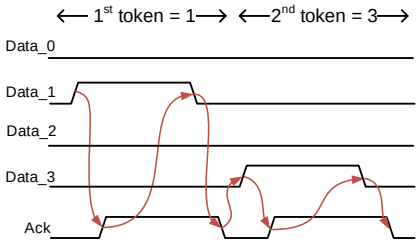
- Two-phase Bundled-Data Protocol
  - Both rising and falling transitions on Req
    - *Means new data are available*
- Both rising and falling transitions on Ack
  - *Means data have been acknowledged*
- Sometimes called transition signaling
  - *It is the transition that is meaningful (stateful), not the level values*
- Push Channels
  - Pull Channels shortly

1-of-N Protocols





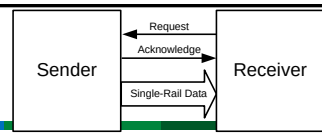
(a)

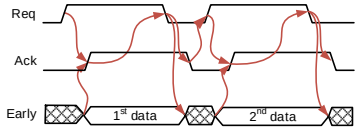


(b)

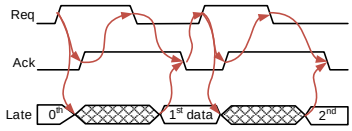
- Dual-Rail
  - 4 phase
  - 2 wires per bit
- 1-of-4
  - 4 phase
  - 4 wires per 2 bits

Pull Channels

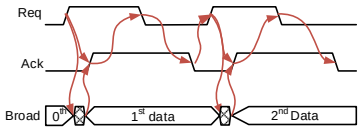




(a)



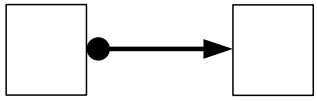
(b)



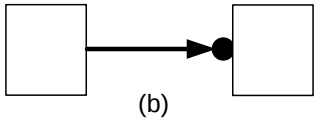
(c)

- Early:
  - Data stable after Ack+
  - Data stable until Req-
- Late:
  - Data stable after Ack-
  - Data stable until Req+
- Broad:
  - Data stable after Ack+
  - Data stable until Req-

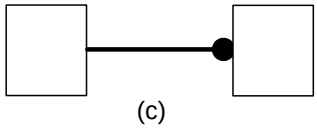
Abstract Channel Diagrams



(a)



(b)



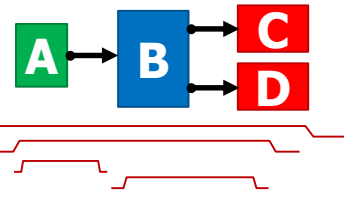
(c)

- Push Channel
- Pull Channel
- Synchronization channel – no data attached/bundled
  - No data – Control only
  - Active on right side

Handshaking details omitted

Sequencing and Concurrency

- Enclosed Handshaking
  - B completes handshake with C before starting handshake with D
    - Operation associated with C occurs before operation associated with D
  - B can enclose both handshakes in handshake with A
    - Completion of handshake with A is acknowledgement that the tasks of C and D are done

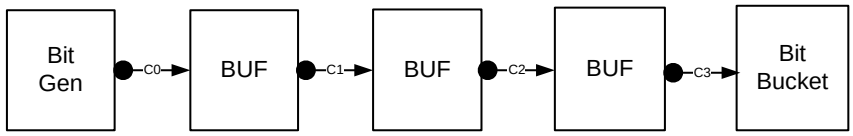


- Pipelining Handshake
  - B overlaps handshake with C and handshake with A
  - Creates pipeline behavior
    - Tokens flow on both channels
  - Increases throughput

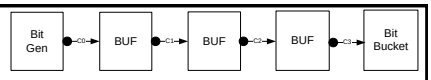
Token Buffer Pipelines

Pipelining Data Tokens

Pipelined Handshaking

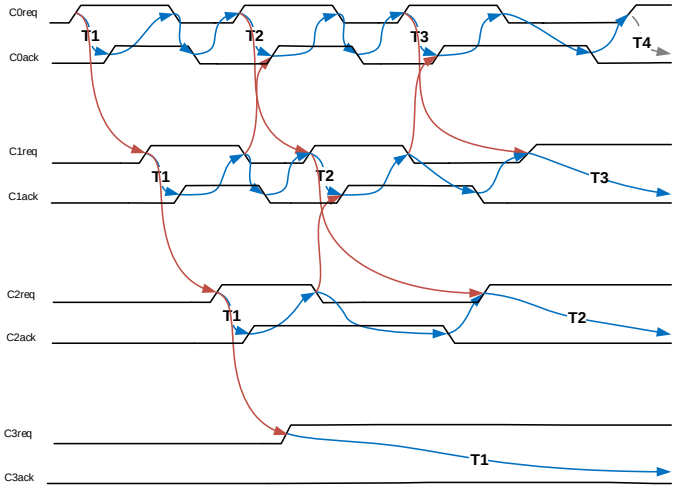


- Pipeline handshaking enables multiple tokens to exist in pipeline
  - Each token represents intermediate result of different problem instance
  - Increases throughput of system
- No tokens lost despite relative speed of stages – has implicit flow control
- Two types
  - Full buffers can support distinct tokens on inputs/output channels
  - Half buffers cannot support distinct tokens on inputs/outputs
    - N-stage pipeline of half-buffers can support a maximum of N/2 tokens

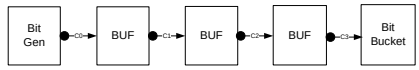


Full-Buffer Handshaking

- Pipeline can store tokens at every buffer



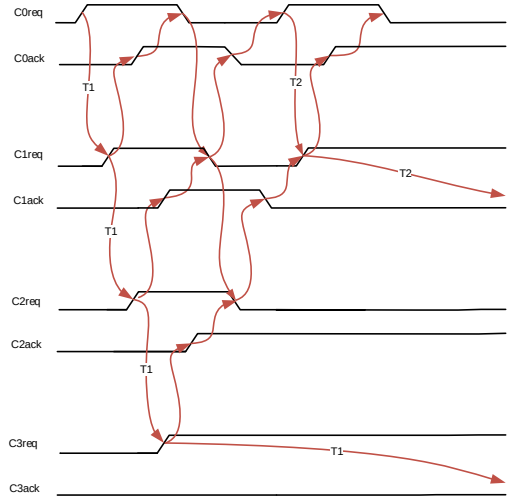
Handshaking assuming Bit Bucket is Stalled



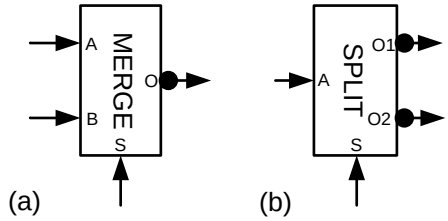
Half-Buffer Handshaking

- Constraint that leads to a half-buffer:
  - output channel must be acknowledged before input channel completes handshake
    - *e.g.*, `c1ack+` ***before*** `c0ack-`
- Pipeline can store tokens at every other buffer

Handshaking assuming Bit Bucket is Stalled

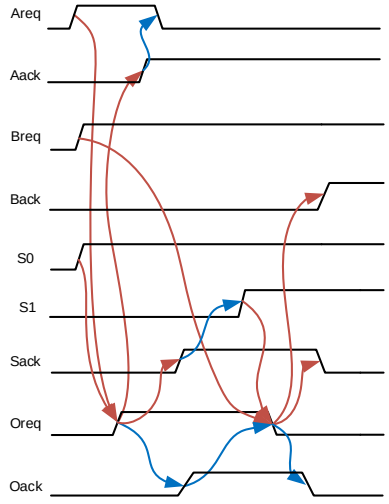


Conditional and Non-Linear Pipeines



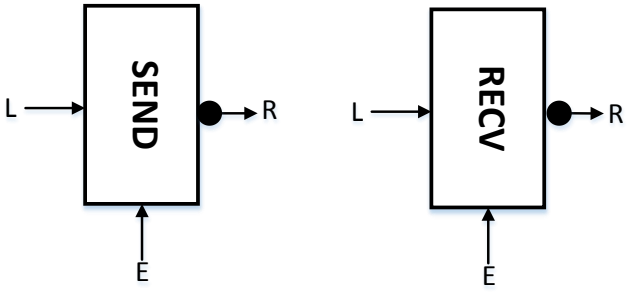
- MERGE
  - Wait for token on S.
  - Depending on value, wait for token on either A or B and send onto O
- SPLIT
  - Wait for token on S and A.
  - Dependent upon value of S, send copy of token on A to O1 or O2

Timing Diagram of Merge



- Assumptions (in this example)
  - Full-buffer Two-phase Handshaking
  - dual-rail select signal
- Functionality – Handshake MUX
  - Token on A consumed first
    - After token on S = 0
    - *i.e.*, S0 changes
  - Token on B stalled until consumed second
    - After token on S = 1
    - *i.e.*, Once S1 changes
  - Result: two tokens on O
    - First = Oreq+
    - Second = Oreq-

Send and Receive Cells



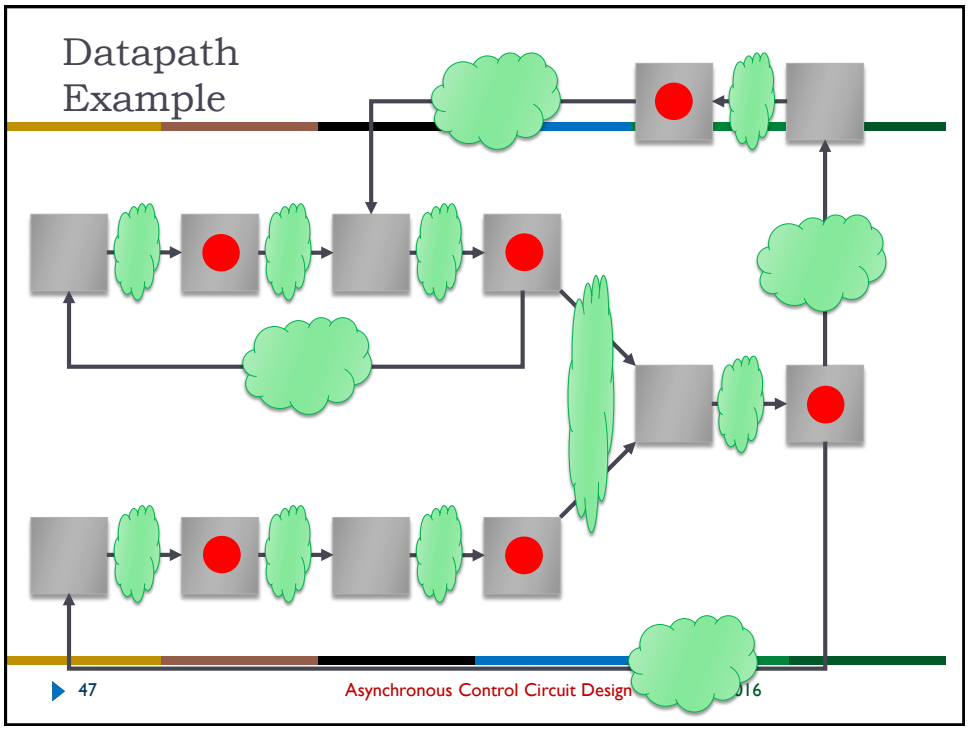
- SEND
  - Always receive on L and E.
  - Conditionally send on R if E==1.
- RECV
  - Always receive on E and send on R.
  - Conditionally receive on L if E==1.

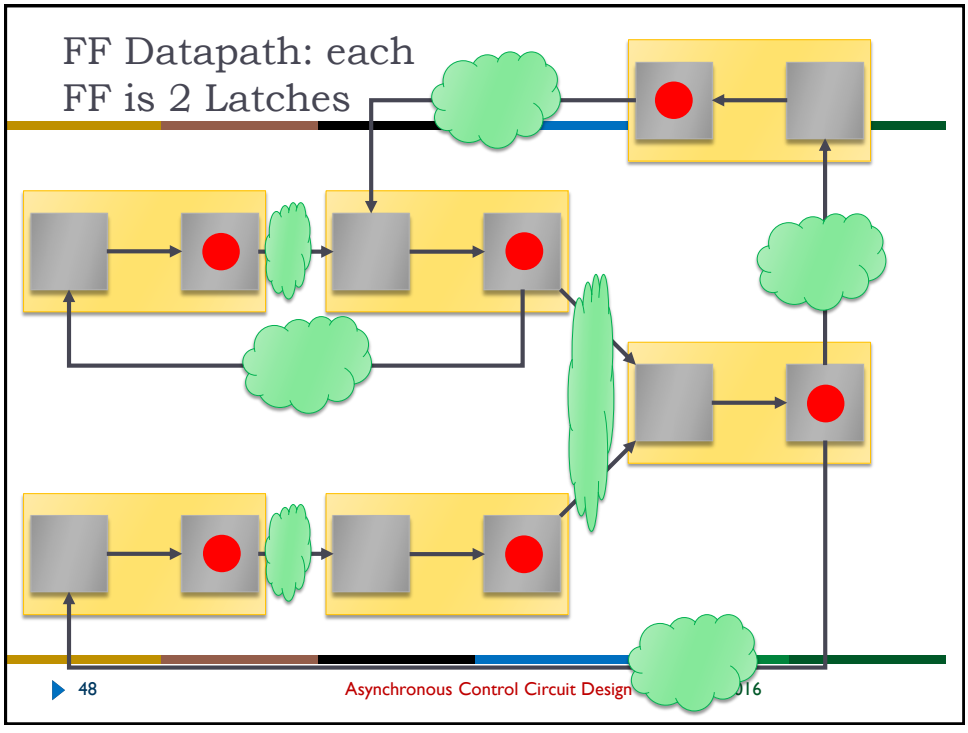
Asynchronous Pipelines

Dynamic behavior is very different from synchronous design

Synchronous vs. Asynchronous Pipelines

- Intuitive Understanding of the Differences

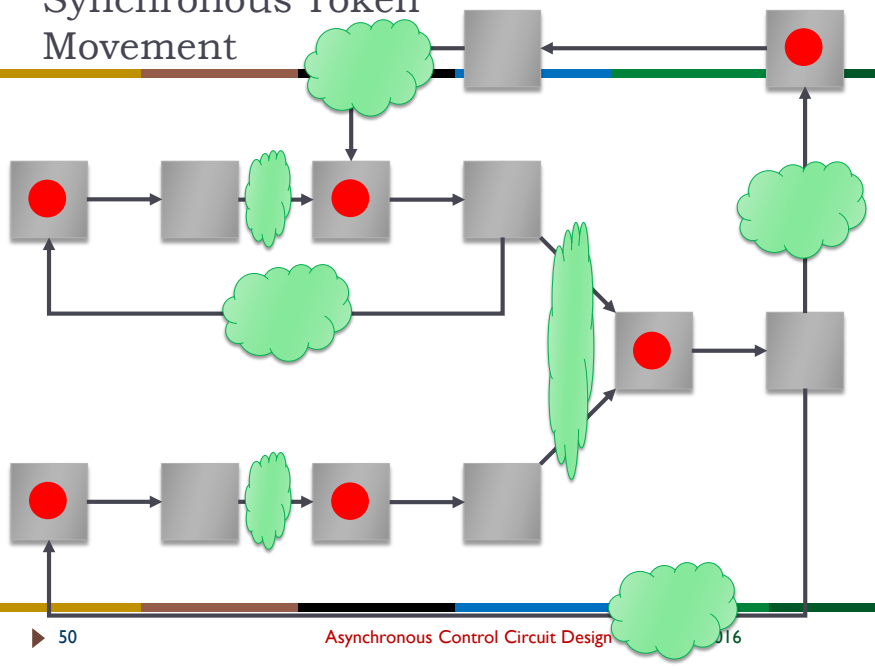

Datapath
Example
47
Asynchronous Control Circuit Design

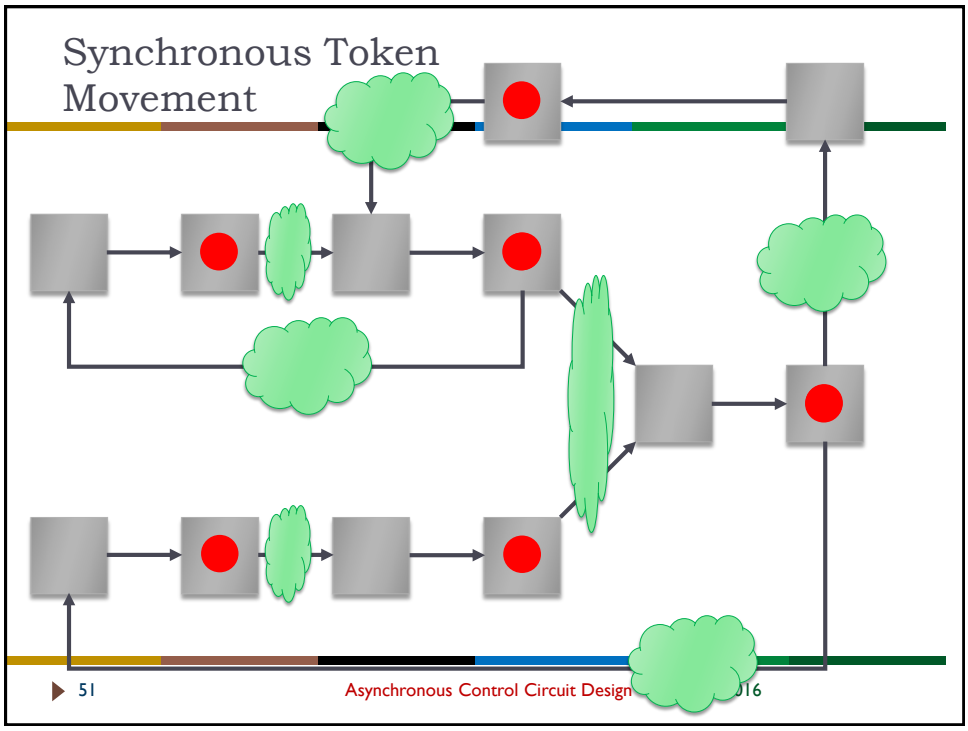

FF Datapath: each
FF is 2 Latches
48
Asynchronous Control Circuit Design

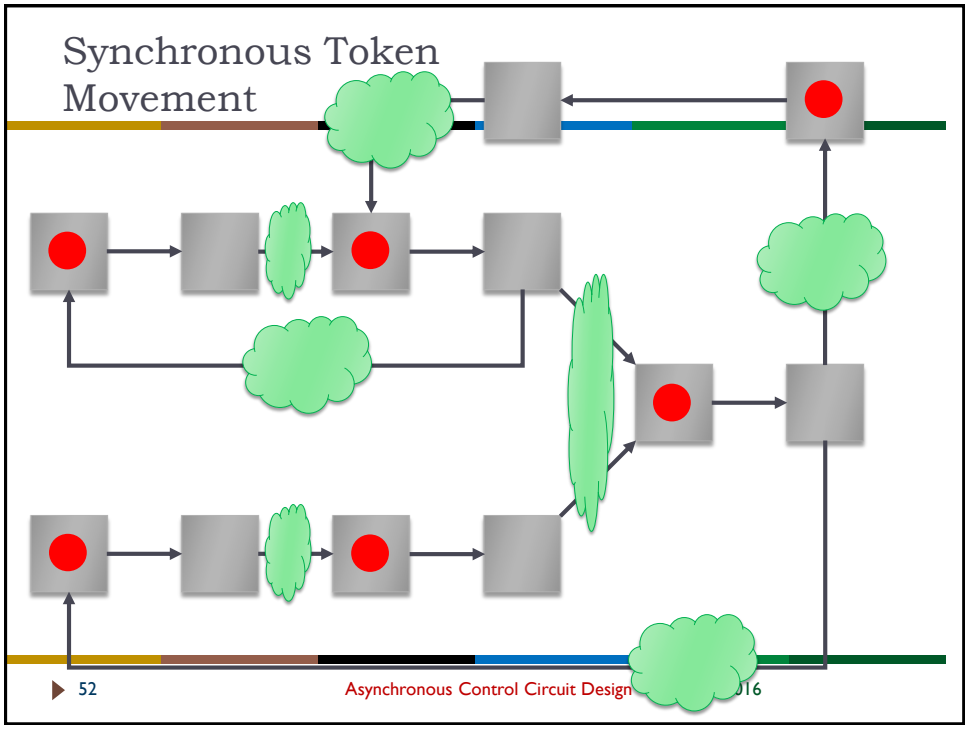
Synchronous vs. Asynchronous

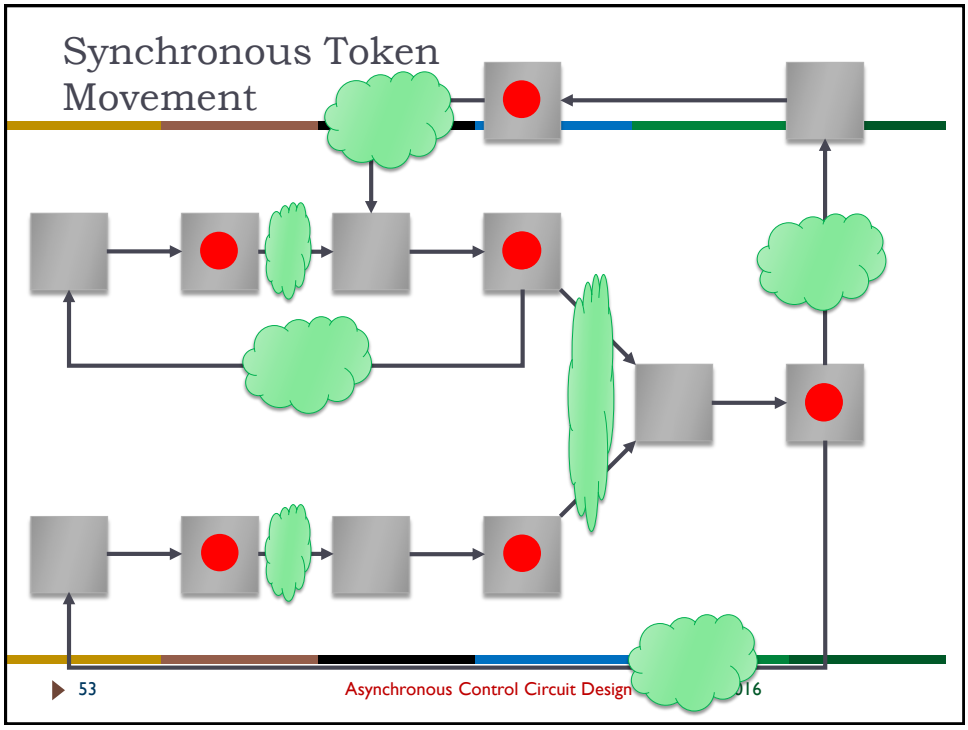
- Tokens represent Data
  - Data Tokens
- We now compare the Synchronous and Asynchronous Data Flows in the simplest version of this datapath
- Synchronous version first…

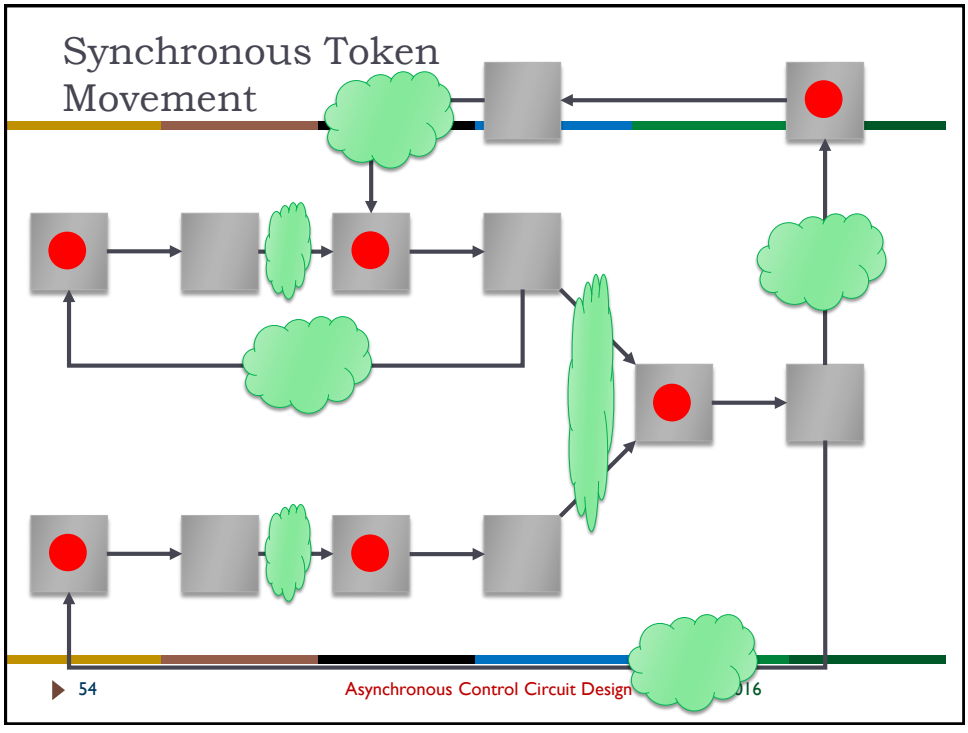
Synchronous Token Movement

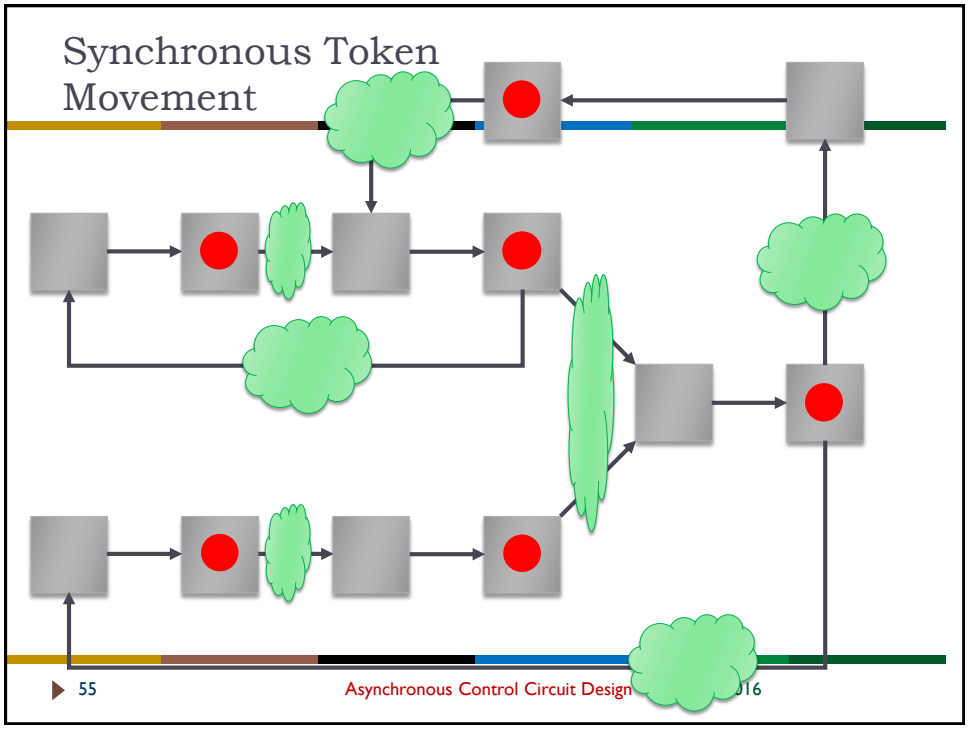


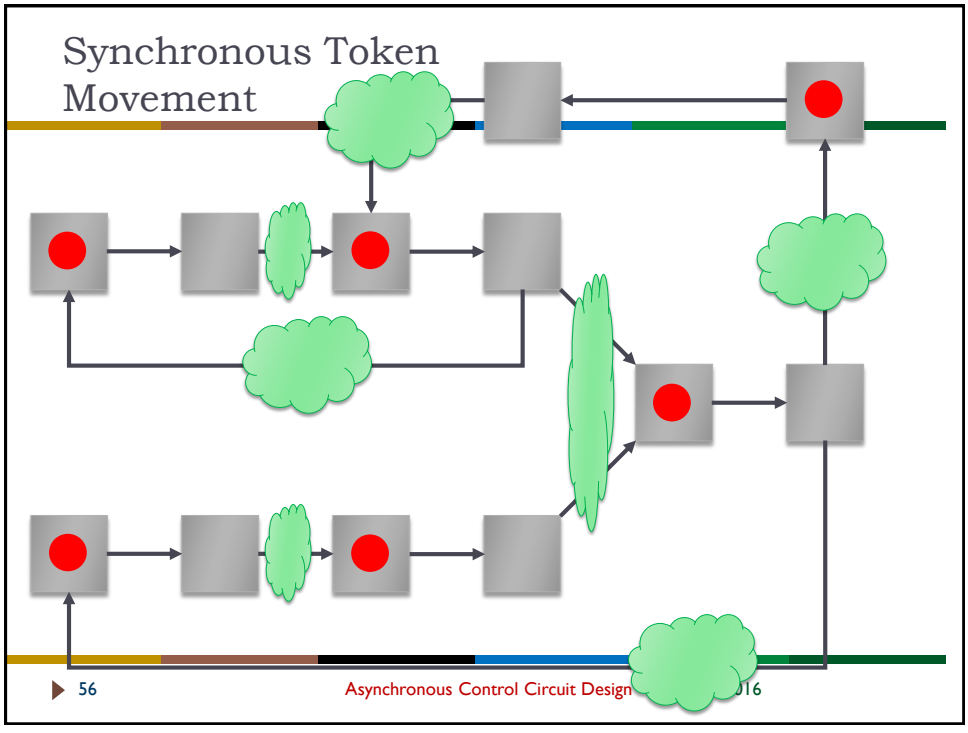

Synchronous Token Movement
51
Asynchronous Control Circuit Design

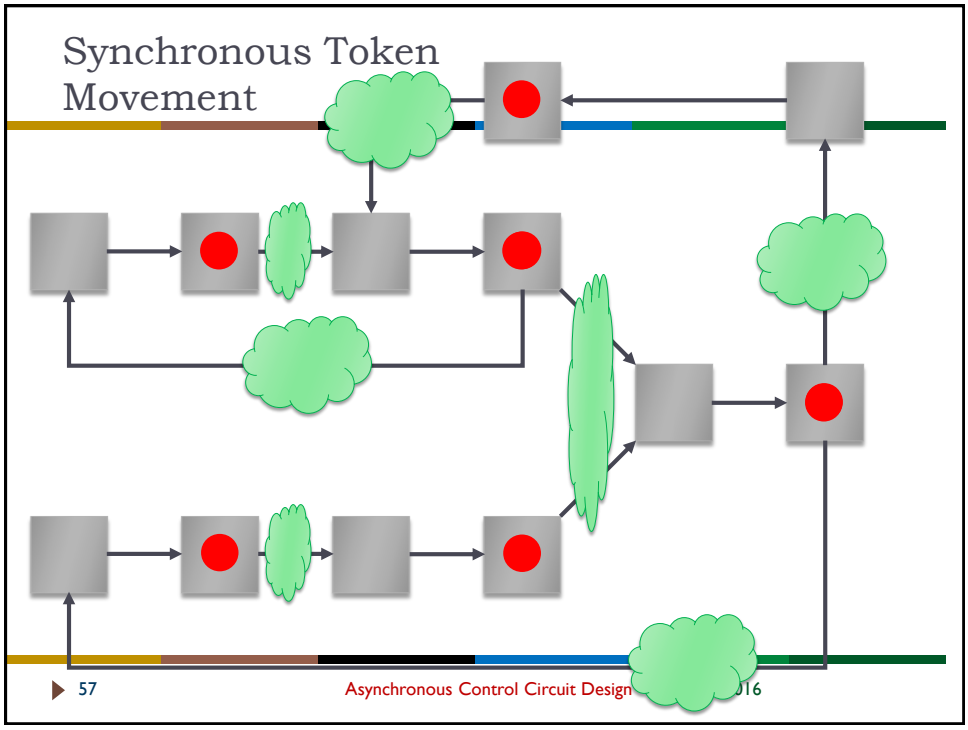

Synchronous Token Movement
52
Asynchronous Control Circuit Design

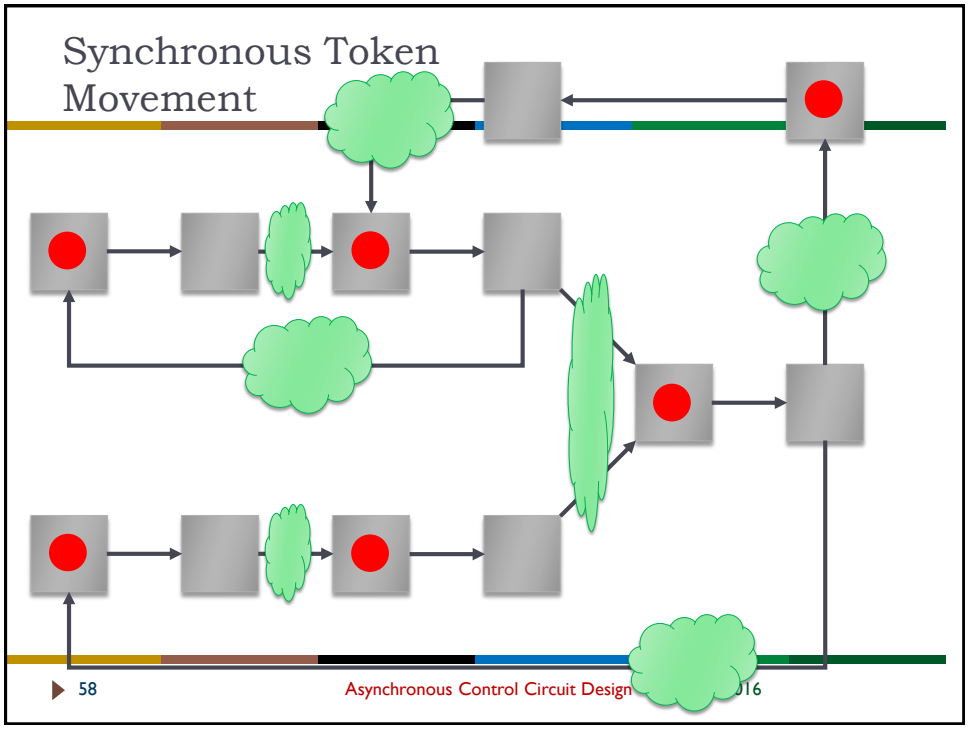

Synchronous Token Movement
53
Asynchronous Control Circuit Design


Synchronous Token Movement
54
Asynchronous Control Circuit Design

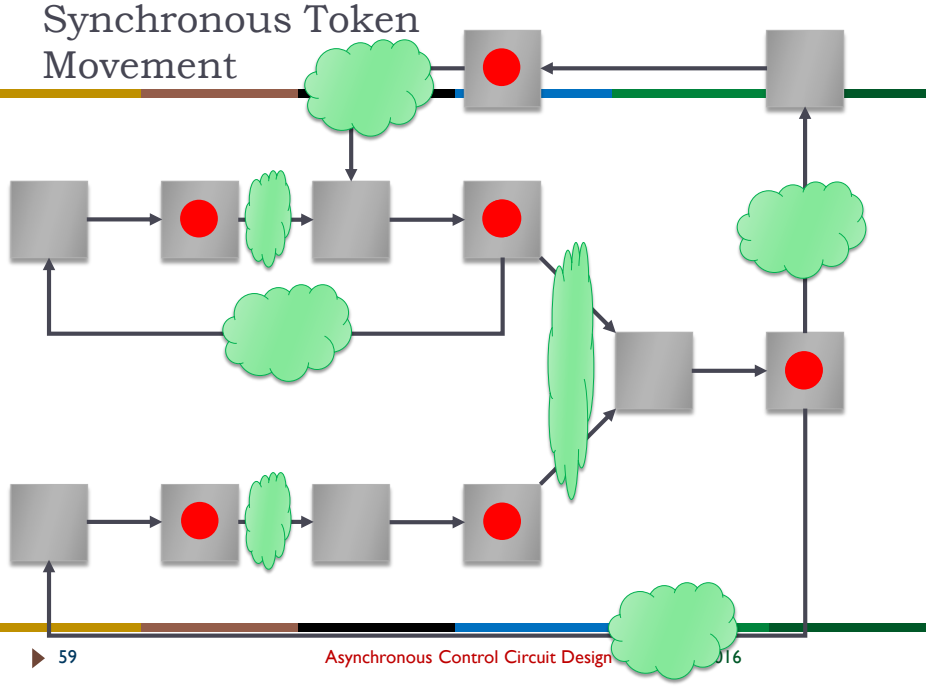

Synchronous Token Movement
55
Asynchronous Control Circuit Design


Synchronous Token Movement
56
Asynchronous Control Circuit Design


Synchronous Token Movement
57
Asynchronous Control Circuit Design


Synchronous Token Movement
58
Asynchronous Control Circuit Design

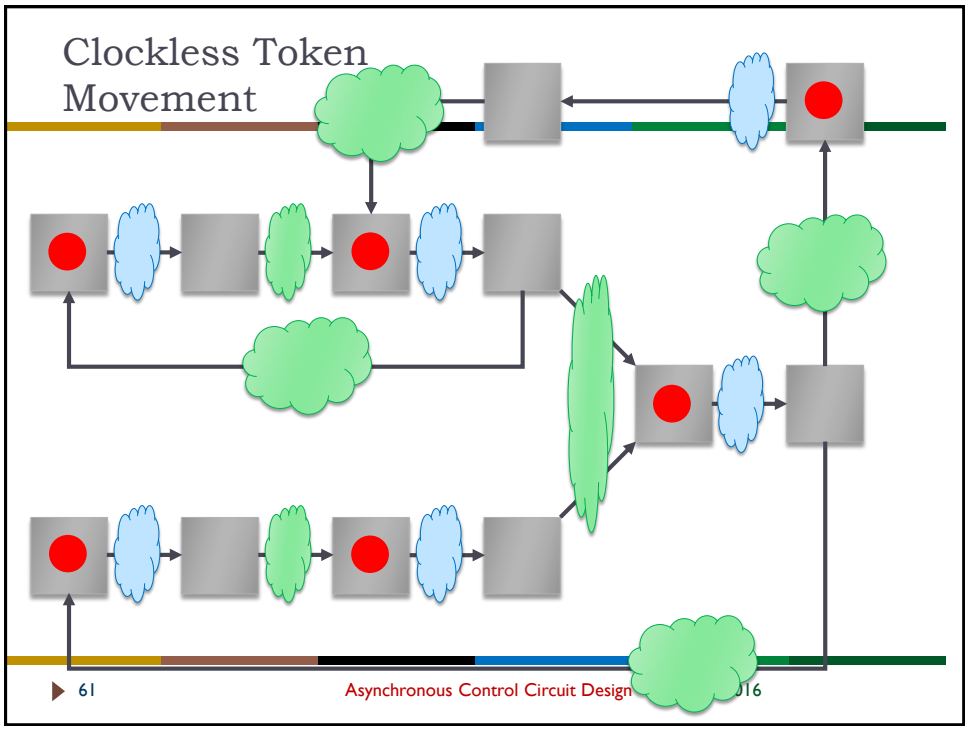
Synchronous Token Movement

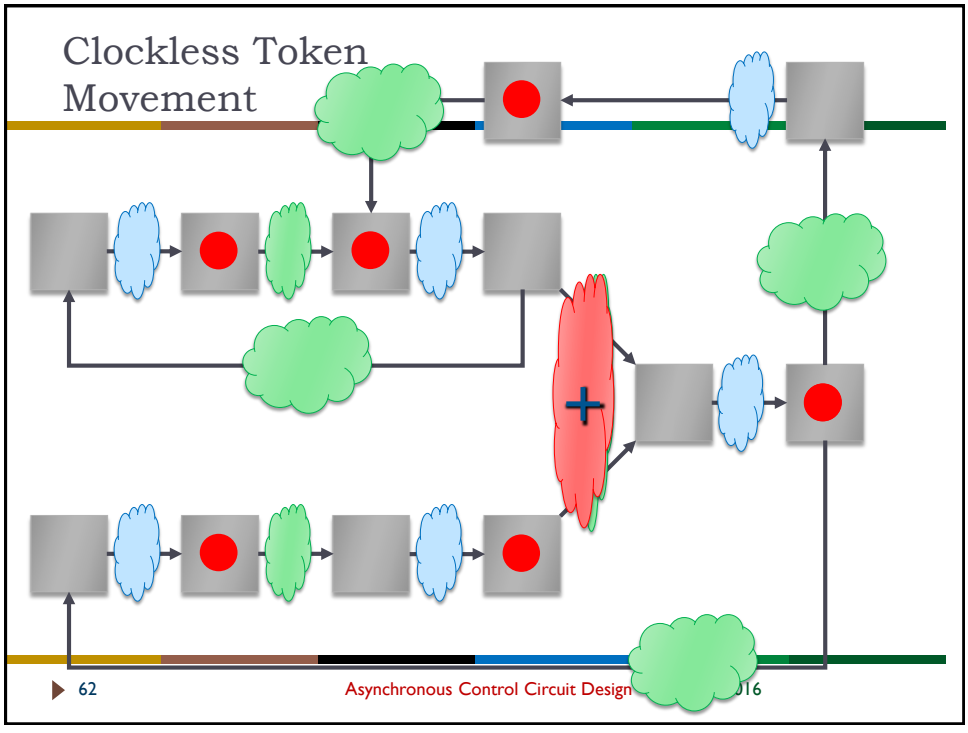


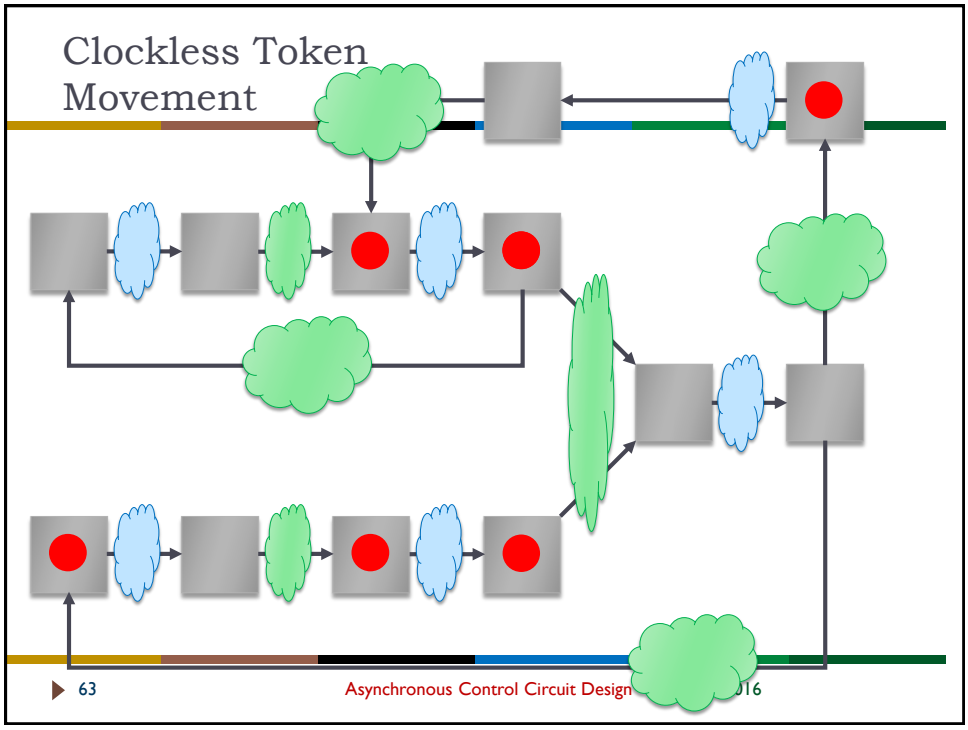
Synchronous Token Movement

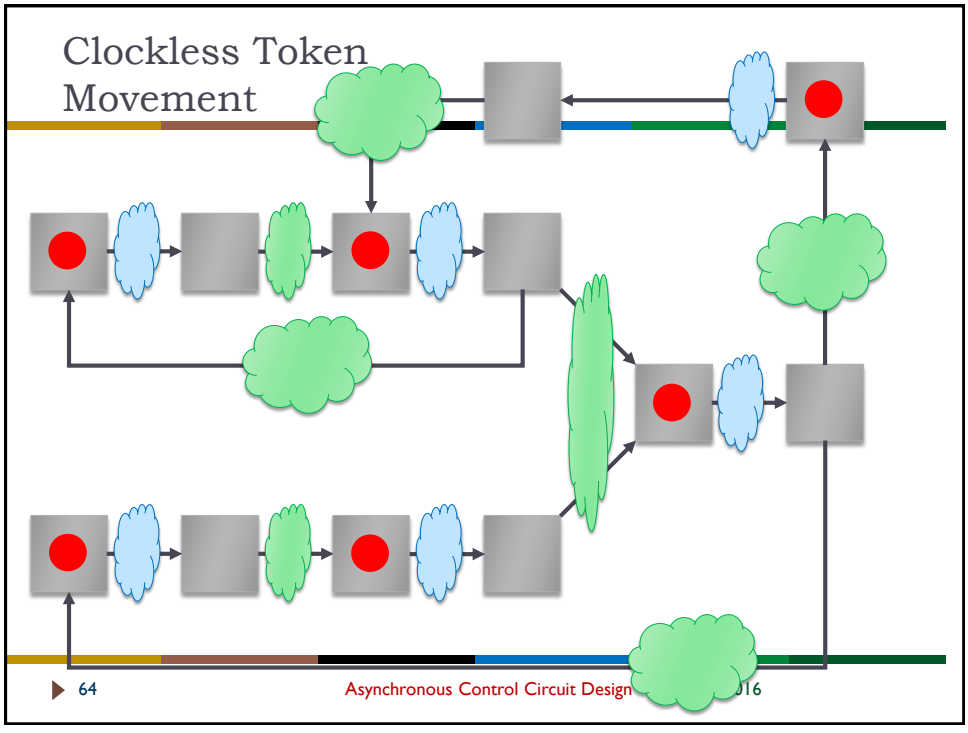
- Tokens are always separated by empty latches
  - Aka "Bubbles"
- Latency in strongly dependent on # of clock cycles
- Pipeline delays result in cycle-based Latency increase

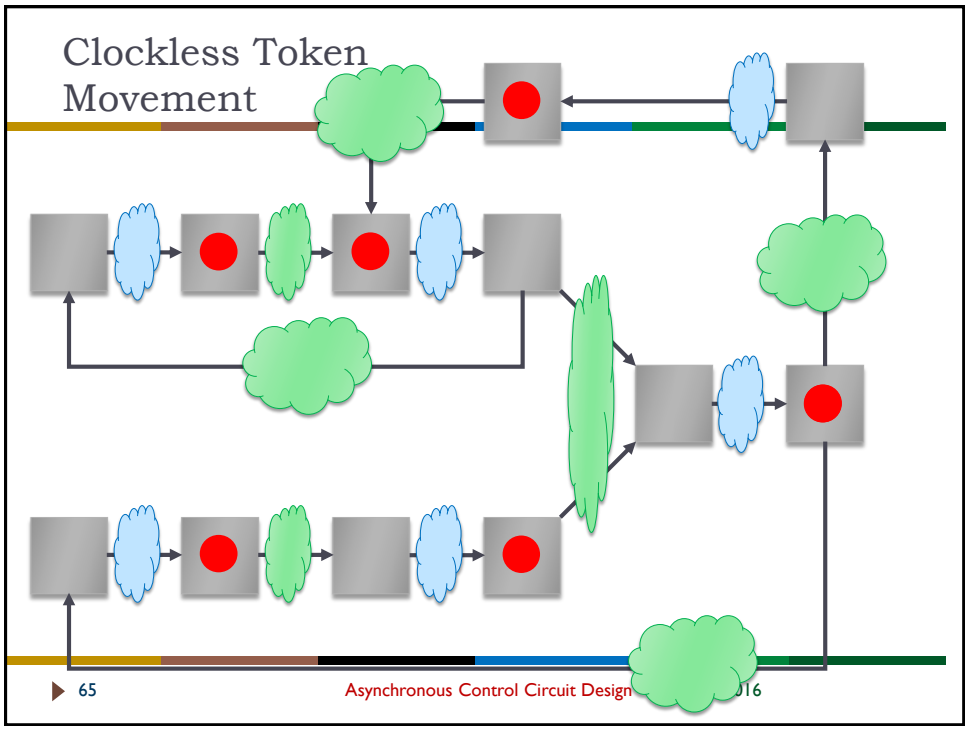
- Now, let's look at the Asynchronous version…

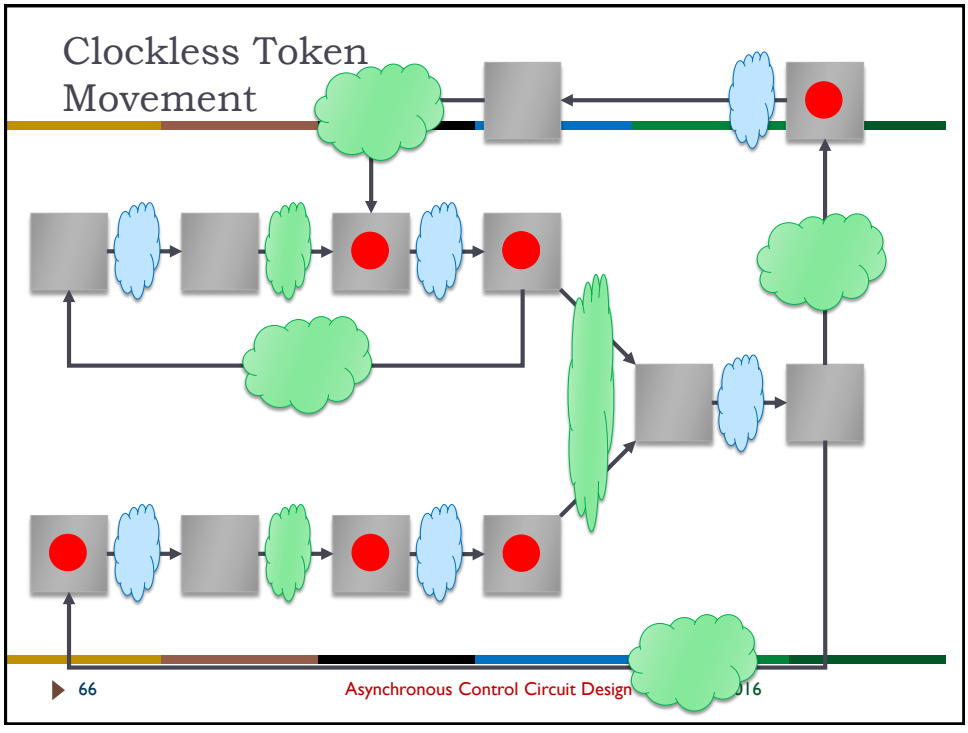

Clockless Token Movement
61
Asynchronous Control Circuit Design

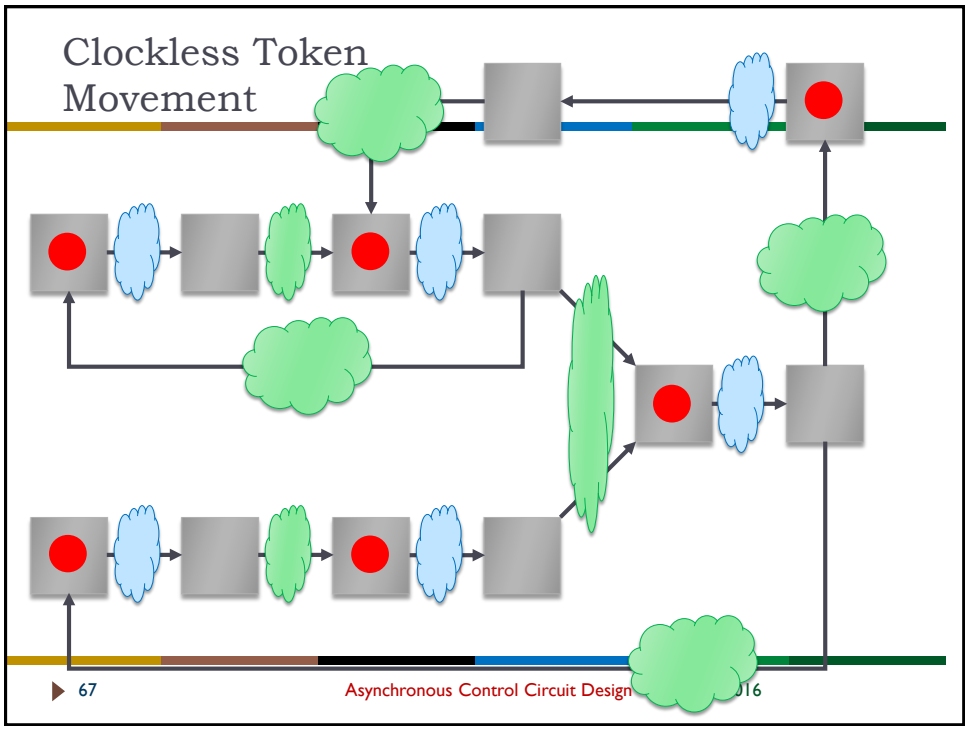

Clockless Token Movement
62
Asynchronous Control Circuit Design

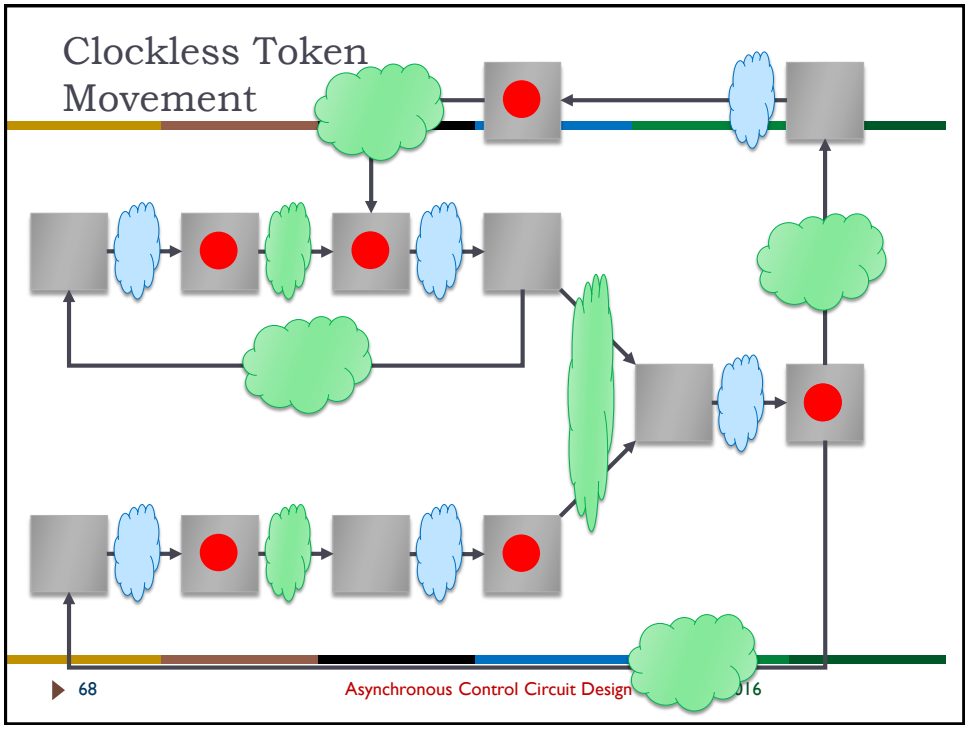

Clockless Token Movement
63
Asynchronous Control Circuit Design

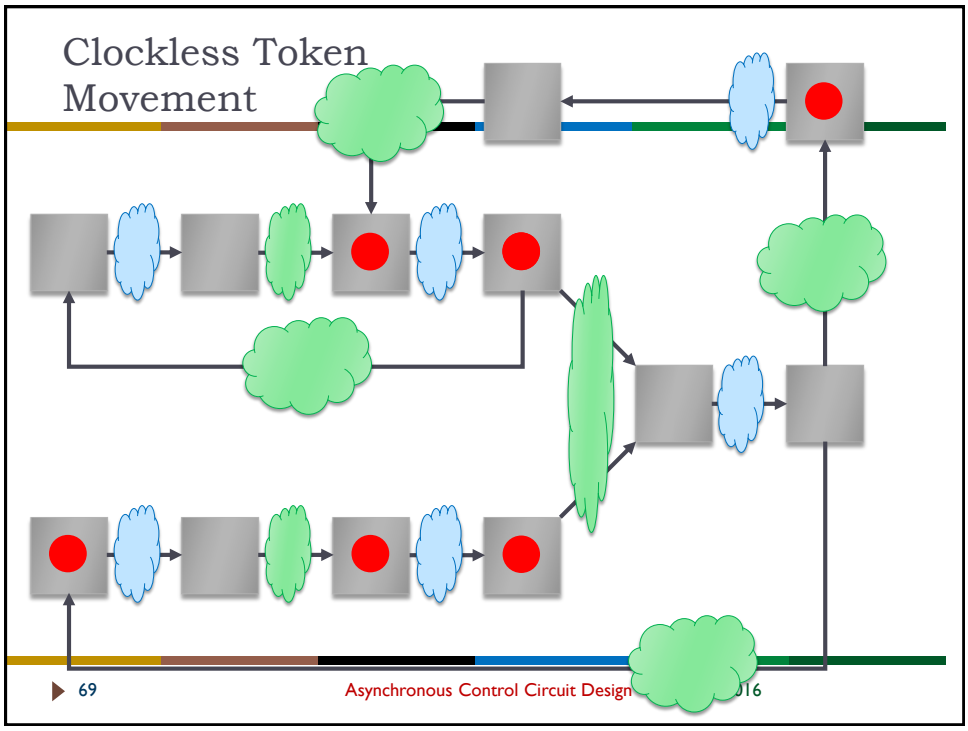

Clockless Token Movement
64
Asynchronous Control Circuit Design

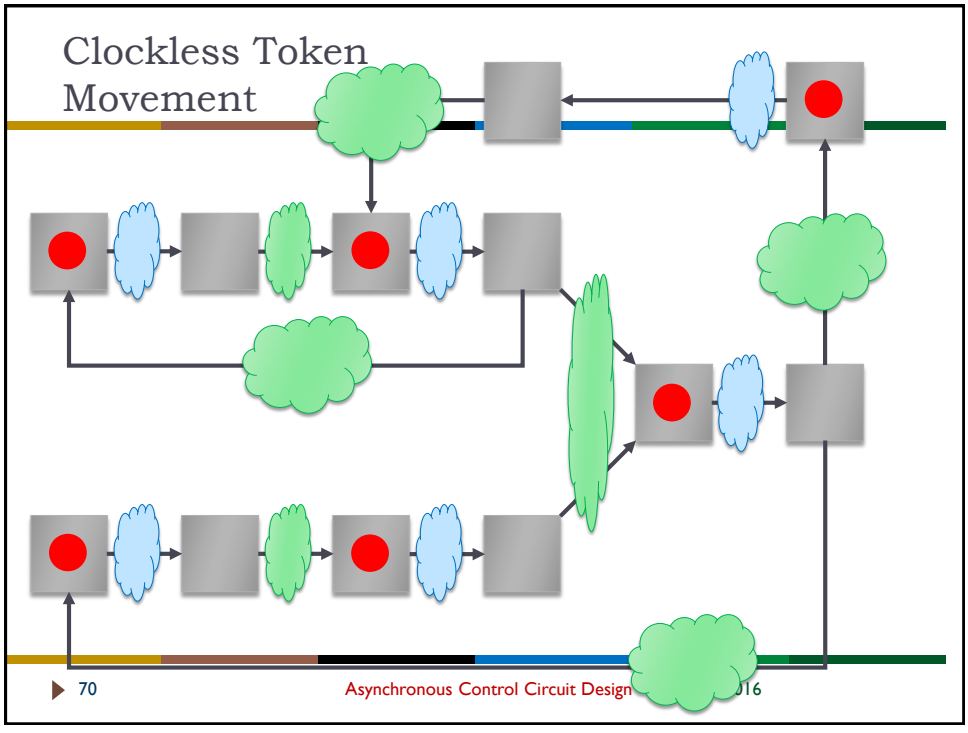

Clockless Token Movement
65
Asynchronous Control Circuit Design

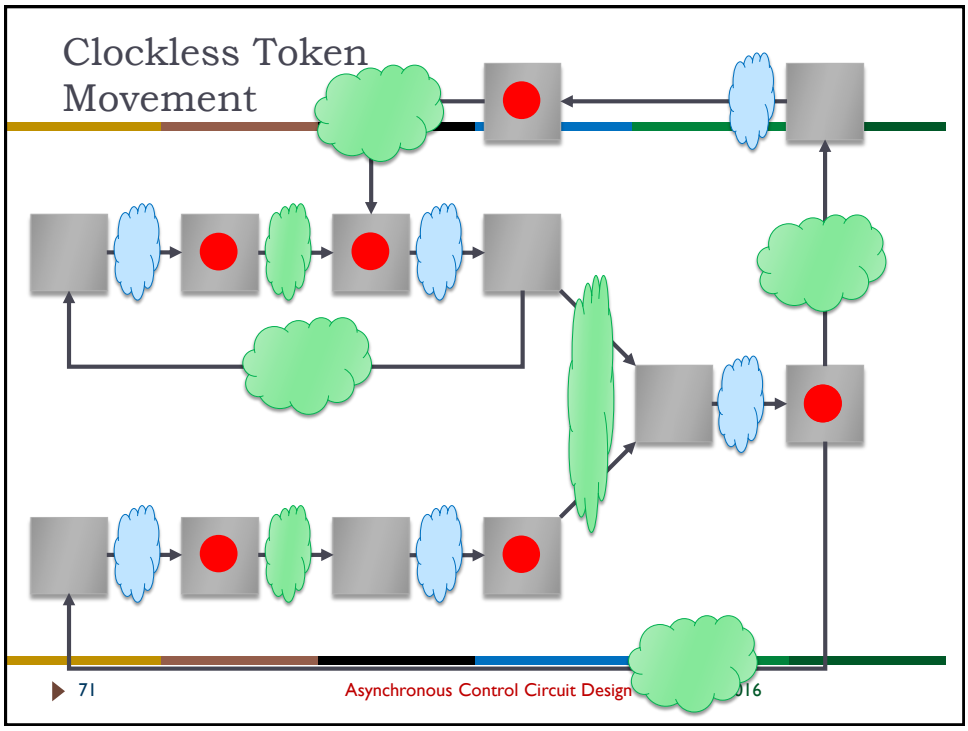

Clockless Token Movement
66
Asynchronous Control Circuit Design

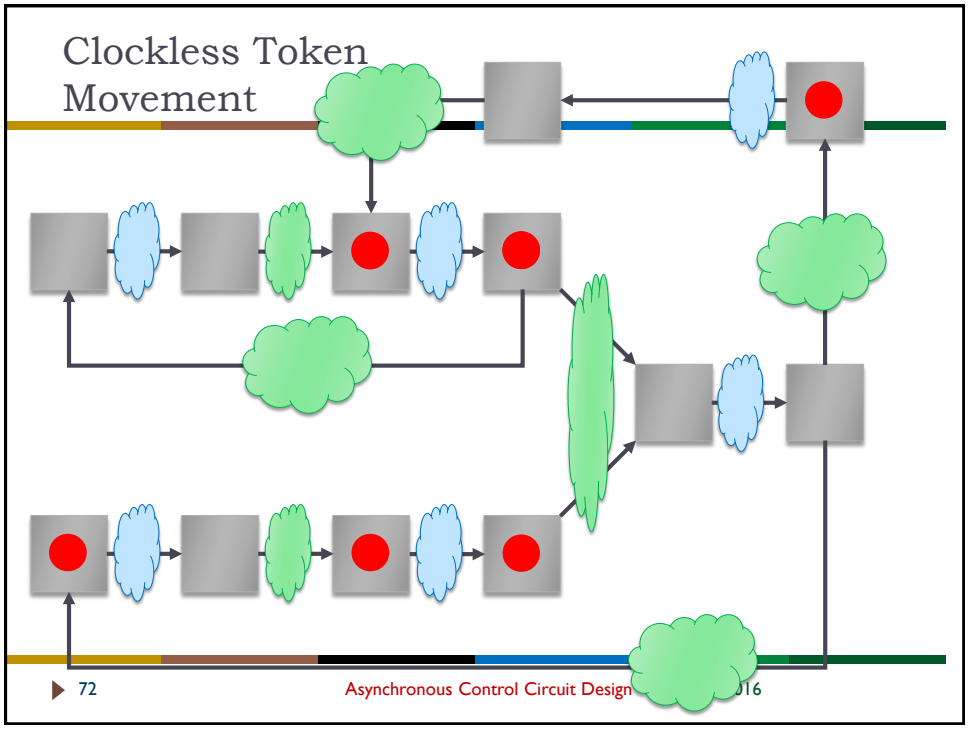

Clockless Token Movement
67
Asynchronous Control Circuit Design


Clockless Token Movement
68
Asynchronous Control Circuit Design


Clockless Token Movement
69
Asynchronous Control Circuit Design


Clockless Token Movement
70
Asynchronous Control Circuit Design


Clockless Token Movement
71
Asynchronous Control Circuit Design


Clockless Token Movement
72
Asynchronous Control Circuit Design

Asynchronous Token Movement

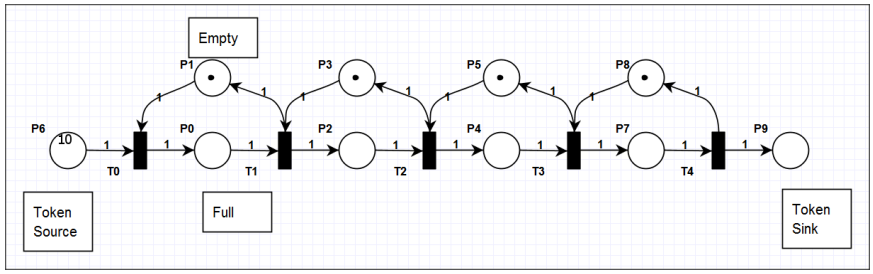
- Latency depends on combinational cloud delays
  - Not multiple of a clock, data-dependent
- Token Speed depends on available space in pipeline stages
  - Contention reduces performance!
    - Similarly to people in a queue (line)
  - Increasing pipe stages can:
    - Increase Tokens/sec, *i.e.* Performance
    - Reduce Latency, if Local Delays are Improved,
      - *e.g.* wire buffering
- Notion of total (or max/min) Tokens in the system, and total (or max/min) empty token buffers (bubbles)
  - System-Level optimization issue

Petri Nets (PN) in a Nutshell

- A PN is a graph of **Places** and **Transitions**
  - Allowed connections
    - Places are connected to Transitions
    - Transitions are connected to Places
- **Places**
  - Places hold a token (in general can be more than 1)
  - A place with a token in active (marked)
- **Transitions**
  - A transition is activated (fires)
    - when tokens are available on all of its input places
  - When it fires
    - it creates new tokens on all of its output places
- PN can model both
  - Choice, Return from Choice
  - Fork and Join

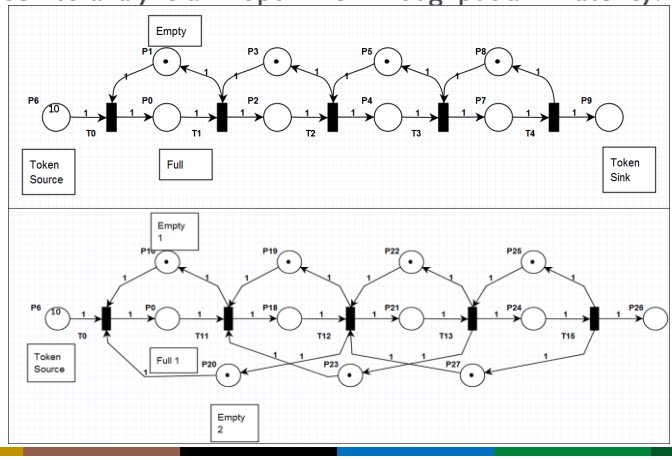
High-Level Asynchronous Pipeline: PTnet Models

- Full Buffer High-Level Model
  - Used to analyze and optimize Throughput and Latency:



High-Level Asynchronous Pipeline PTnet Models

- Full Buffer and Half Buffer High-Level Models
  - Used to analyze and optimize Throughput and Latency:



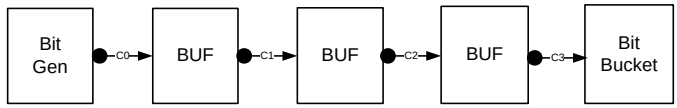
Dynamic Pipeline Behavior

- 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 an optimally-filled pipeline
  - token distance travelled for 1 *T* = ***(FL + BL)/FL***
- Dynamic Slack or Dynamic Occupancy
  - Average token capacity in the pipeline at optimal throughput
  - N*1/Spread = ***(N*FL)/(FL + BL)***

Dynamic Pipeline Behavior

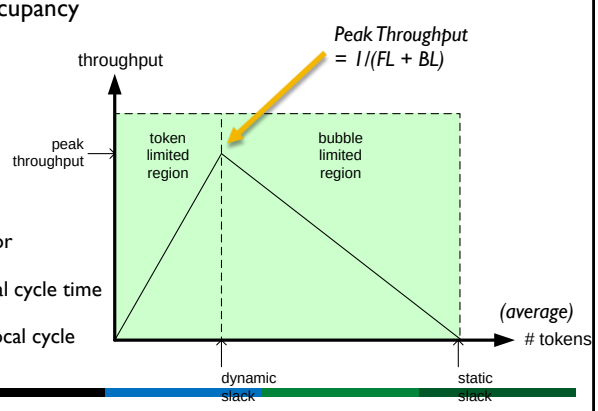
- Cycle Time
  - *T* = ***FL + BL*** (simplest case)
- Latency
  - input to output delay = ***N * FL***
- Throughput
  - # of tokens flowing per unit time, generally = ***1/T = 1/(FL + BL)***
  - Depends on throughput of sender/receiver
- Static Slack or Static Token Occupancy
  - Maximum token capacity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

Dynamic Occupancy, Throughput



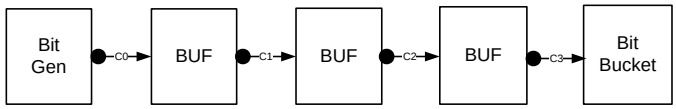
- Dynamic Slack or Dynamic Occupancy Formula for N buffers:

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

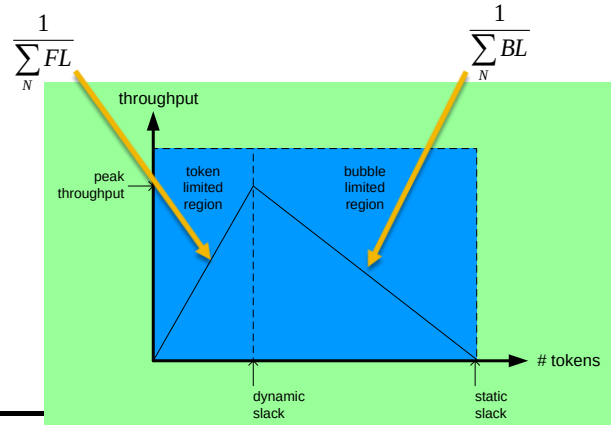


  - Assumptions:
    - Tokens not stalled by buffers or Bit Bucket resetting
    - Tokens inserted at rate of local cycle time (*FL* + *BL*)
    - Tokens consumed at rate of local cycle time

Throughput vs. Tokens Graph



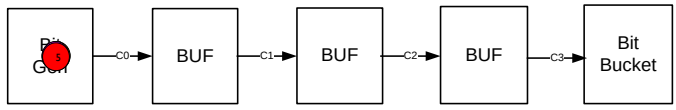
- Throughput is zero
  - When no tokens in pipe
  - When pipeline is full of tokens
- Peak throughput in-between
  - Token limited region
    - Faster Bit Gen improves throughput
  - Bubble limited region
    - Faster Bit Bucket improves performance



Example: Pipeline Performance

- Slow left environment
  - Bit Gen: LCT = 6
  - Buffer: FL=2, BL=2
  - Bit Bucket: LCT = 2

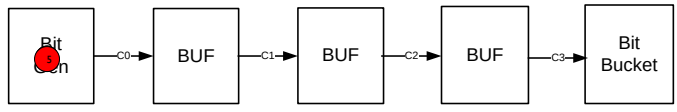
t=0: Token 1 generated
t=6: Token 2 generated
t=12: Token 3 generated
t=18: Token 4 generated
t=24: Token 5 generated



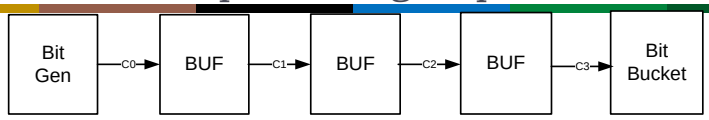
Example: Pipeline Performance

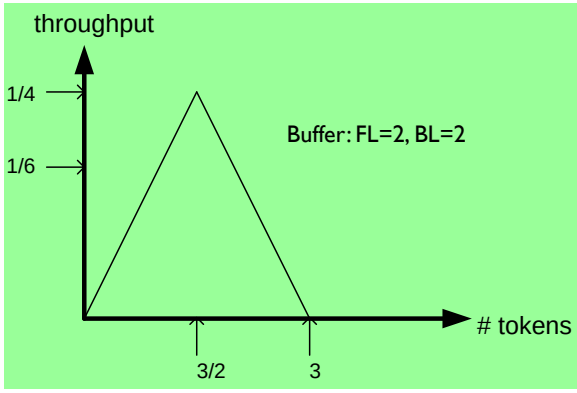
- Slow right environment
  - Bit Gen: LCT = 2
  - Buffer: FL=2, BL=2
  - Bit Bucket: LCT = 6

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

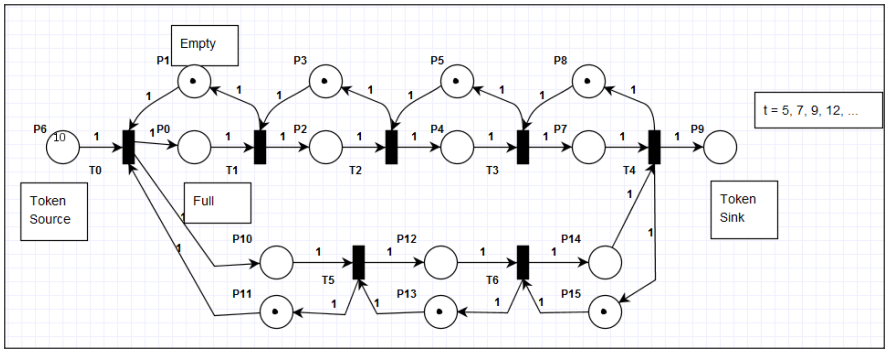




More Complex Example: Pipeline Performance

- What is the impact of fork/join on asynchronous pipelines?
  - Assume FL = 1, BL = 1

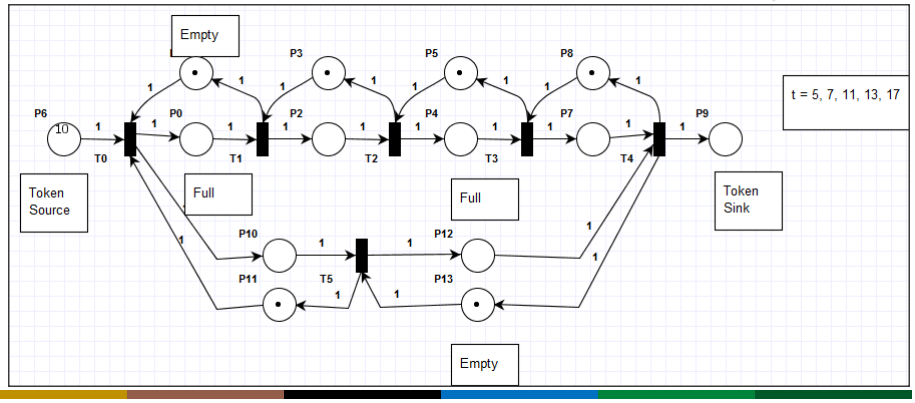
4, 3 Fork Join



More Complex Example: Pipeline Performance

- What is the impact of fork/join on asynchronous pipelines?
  - Assume FL = 1, BL = 1

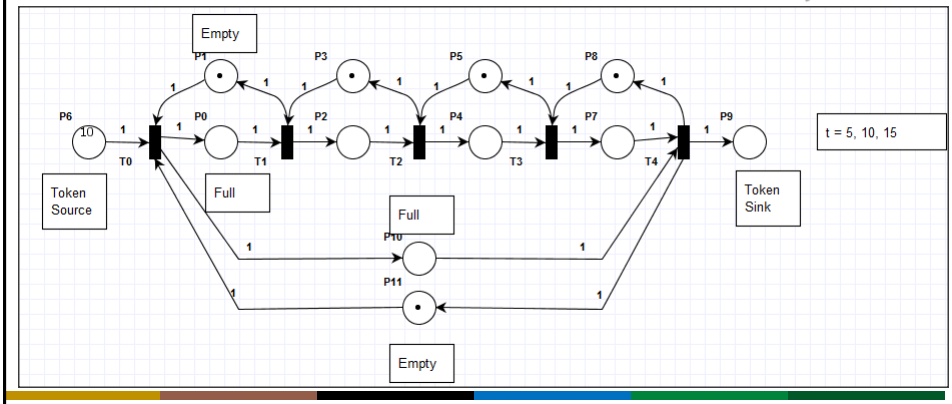
4, 2 Fork Join



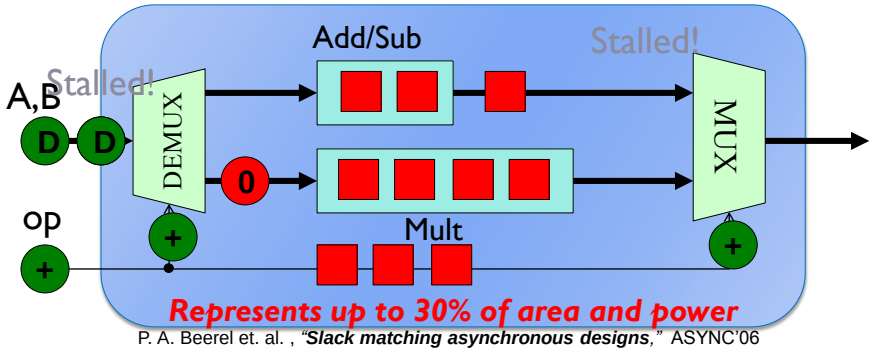
More Complex Example: Pipeline Performance

- What is the impact of fork/join on asynchronous pipelines?
  - Assume FL = 1, BL = 1

4, 1 Fork Join



Key to High Performance – Slack Matching



P. A. Beerel et. al. , *"Slack matching asynchronous designs,"* ASYNC'06

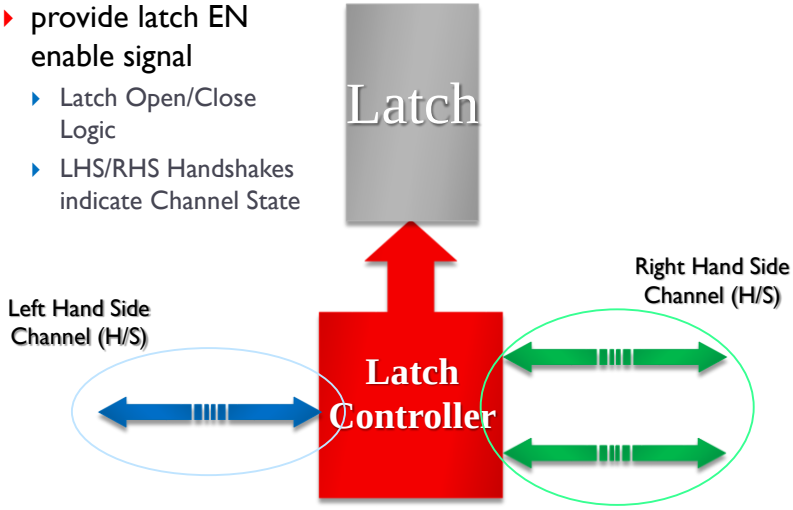
- **The Slack Matching Problem**:
  - Add minimum number of pipeline buffers to the circuit to meet a target T
- This problem is unique to Asynchronous Design
  - Unfortunately, often yields significant Area and Power Overhead!

Latch Control Circuits

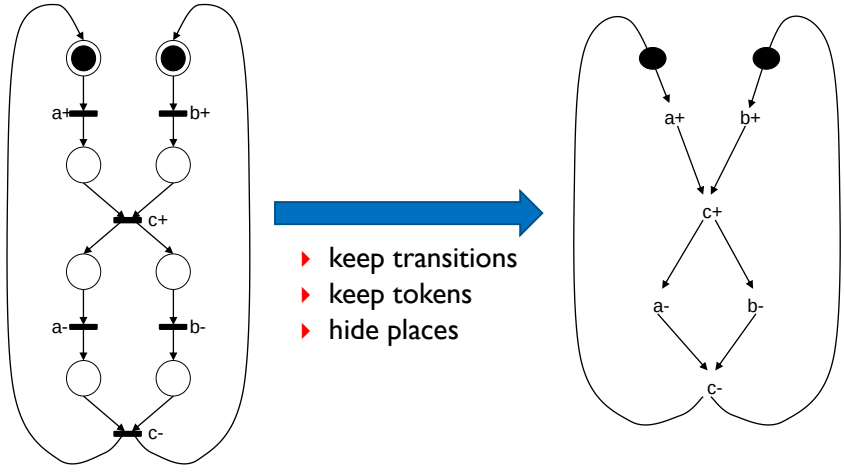
Basic Example of Latch Control

Bundled-Data Control Circuits – Basic Latch Control

- provide latch EN enable signal
  - Latch Open/Close Logic
  - LHS/RHS Handshakes indicate Channel State

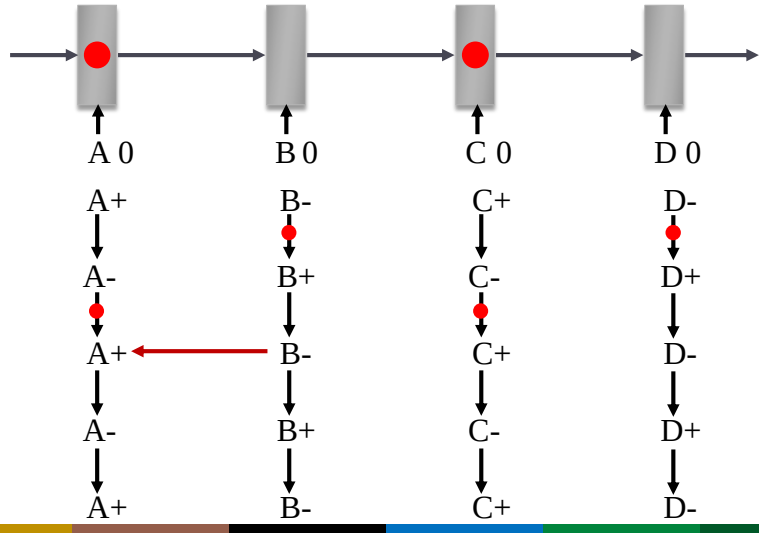


C Gate – PN to STG Conversion

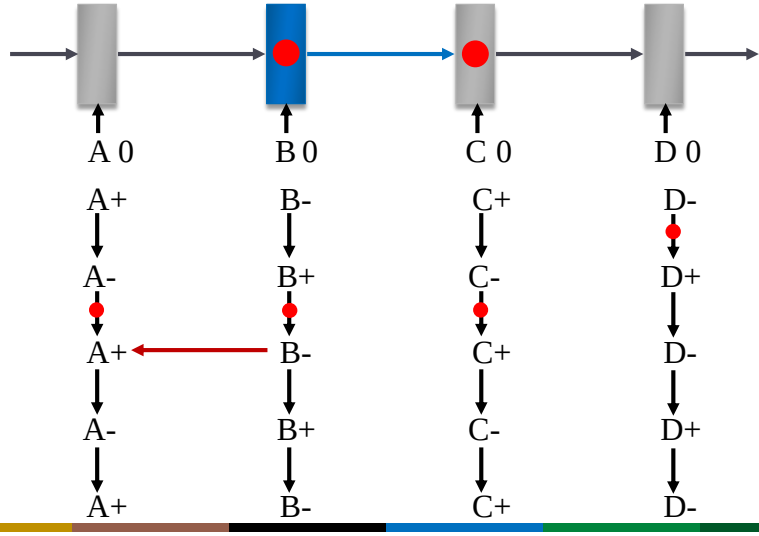


- keep transitions
- keep tokens
- hide places

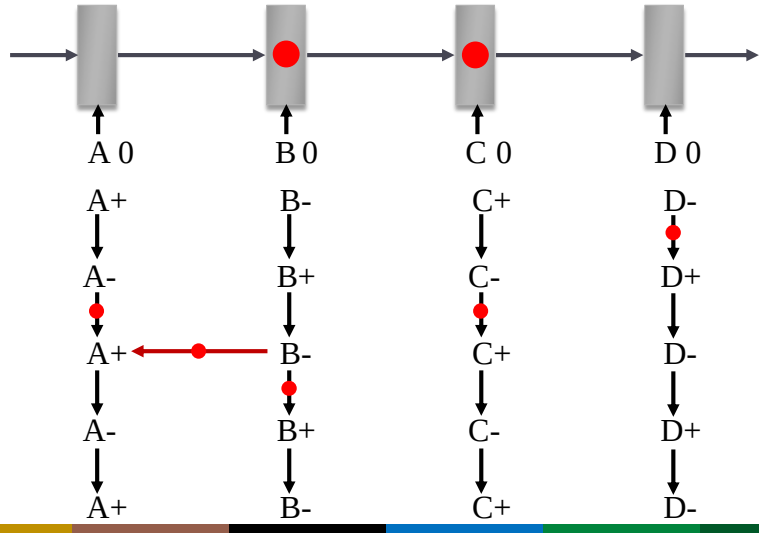
Condition 1: Latch Should Not Open until Successor Latch has Captured Data



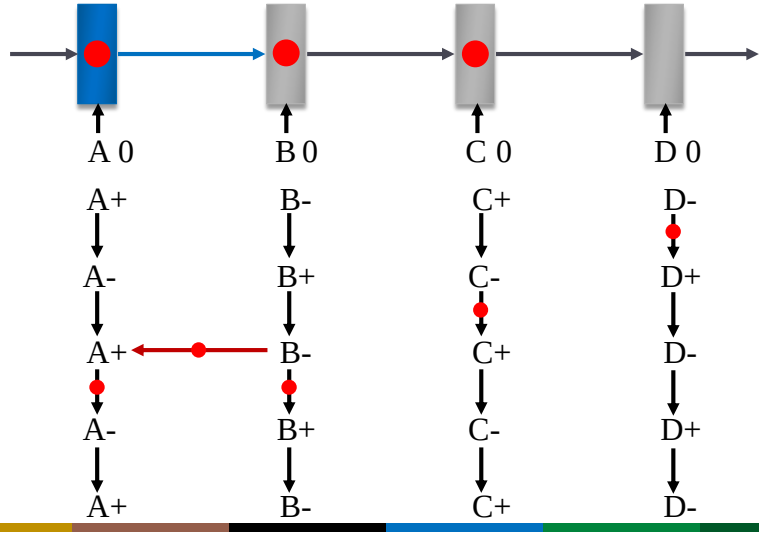
Condition 1: Latch Should Not Open until Successor Latch has Captured Data



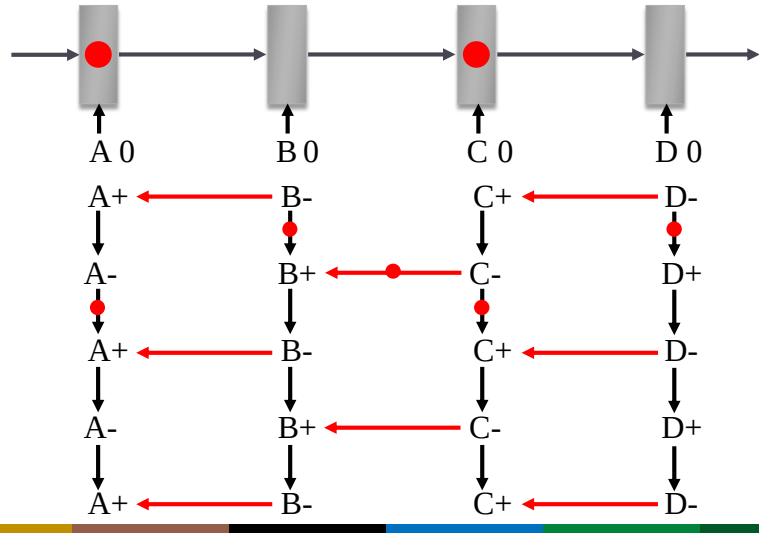
Condition 1: Latch Should Not Open until Successor Latch has Captured Data



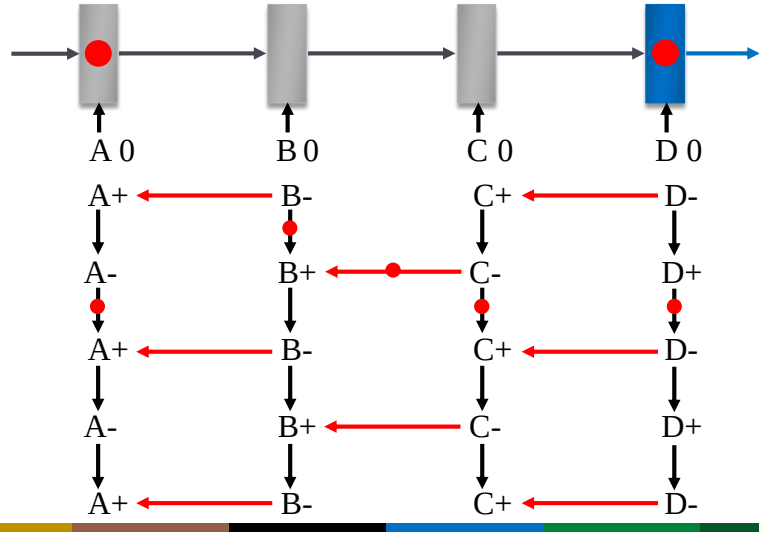
Condition 1: Latch Should Not Open until Successor Latch has Captured Data



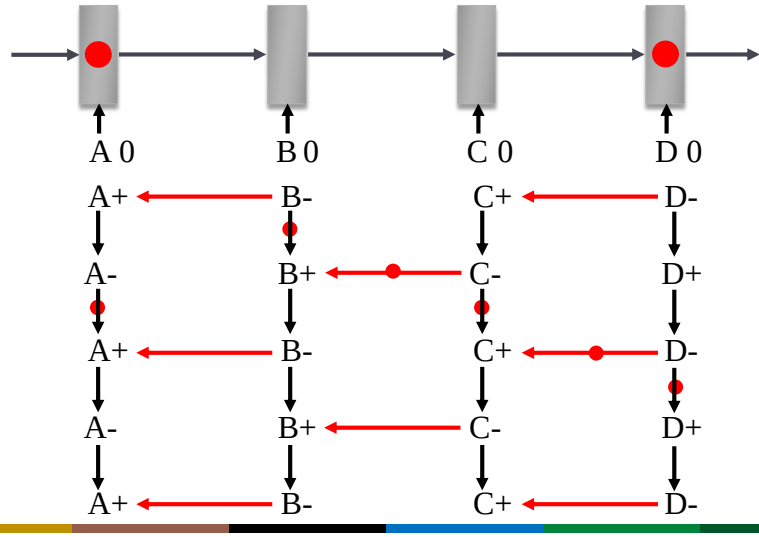
Latch Control Signal Dependencies Imposed by Condition 1



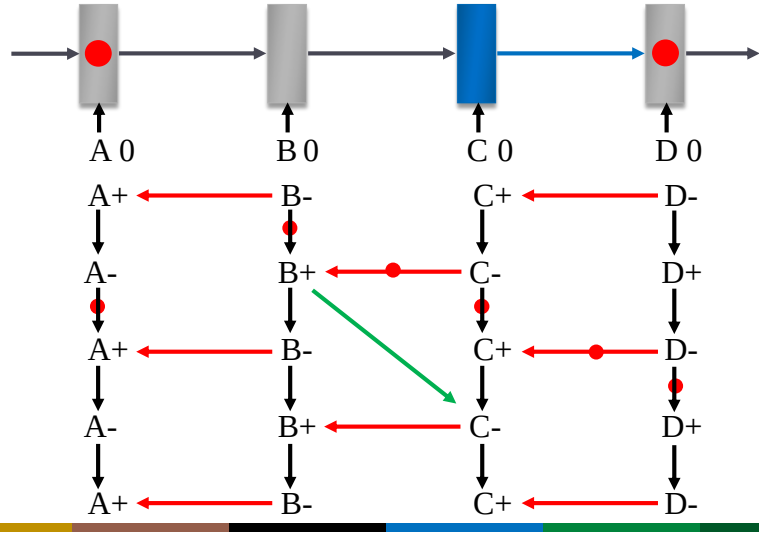
Latch Control Signal Dependencies Imposed by Condition 1



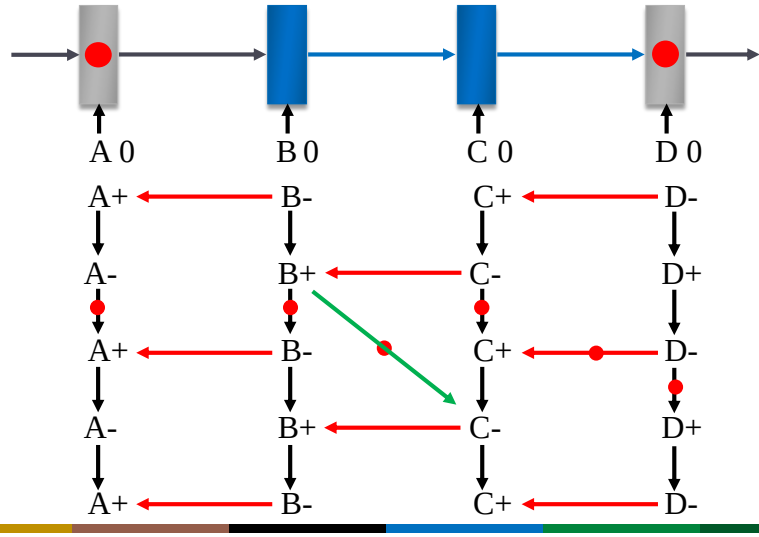
Latch Control Signal Dependencies Imposed by Condition 1



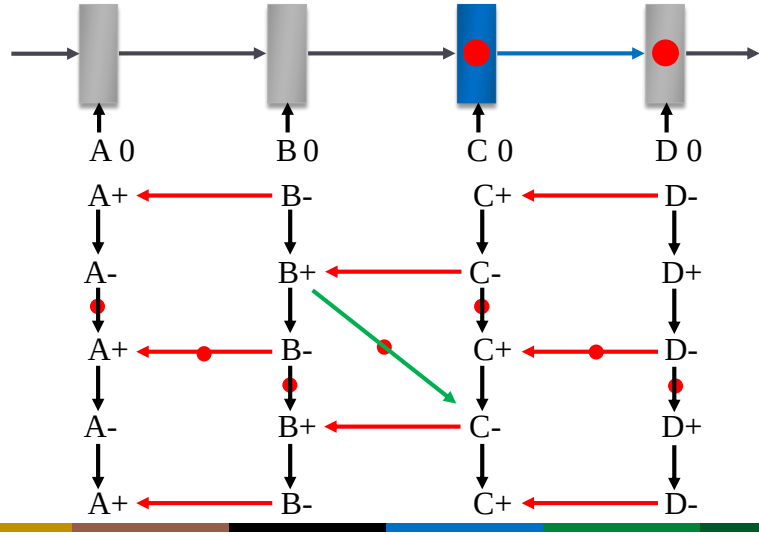
Condition 2: Latch Should Not Close until Captured Data from Predecessor



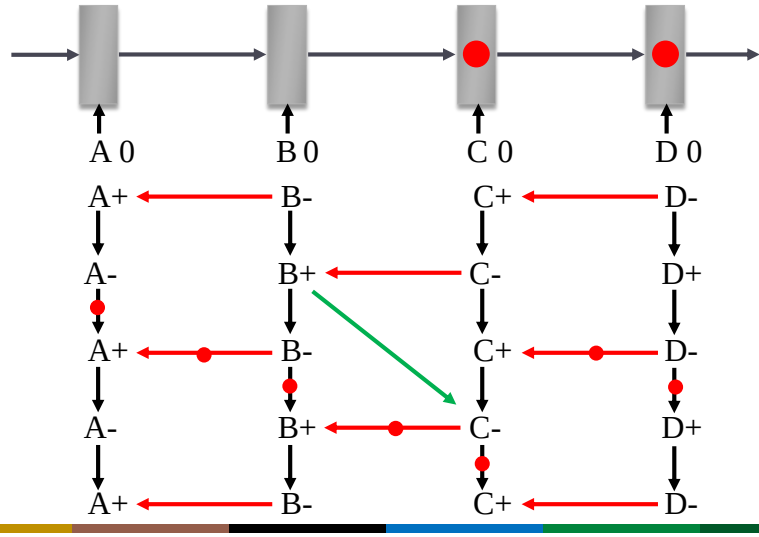
Condition 2: Latch Should Not Close until Captured Data from Predecessor



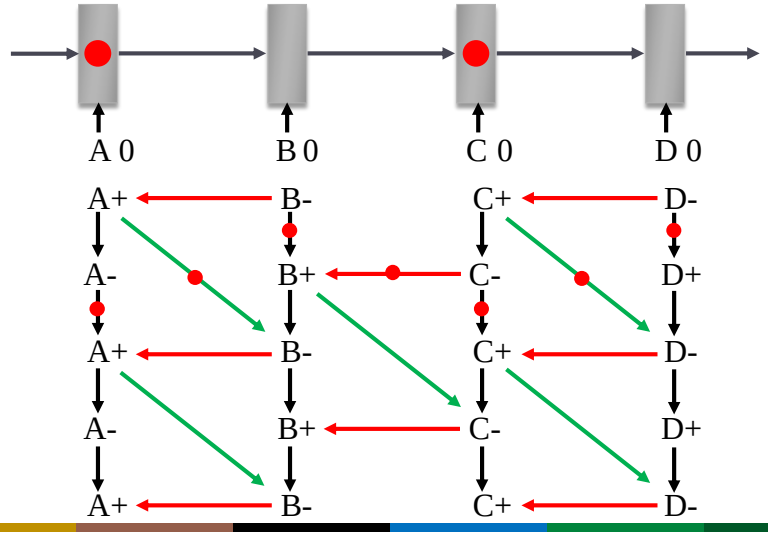
Condition 2: Latch Should Not Close until Captured Data from Predecessor



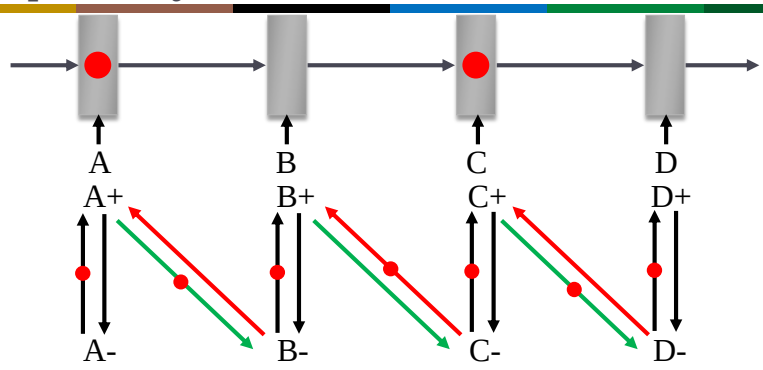
Condition 2: Latch Should Not Close until Captured Data from Predecessor



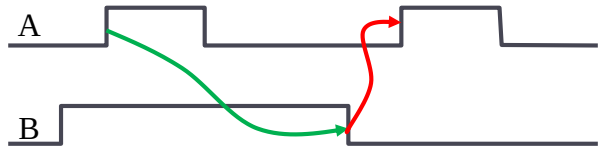
Latch Control Signal Dependencies Imposed by Conditions 1 and 2



Folded Latch Control Signal Dependencies Imposed by Conditions 1 and 2



- Latch Control Signal Graph Dependencies can also be visualized in the Time Domain



Formal Models For Asynchronous Control Circuit Design

- The graph model examples are really simplified **PN**
  - PN is Place, Transition Net (invented by Petri)
  - Represent Dependencies between signal transitions
    - Causality
  - Can Represent Choice
  - Multiple Signals (Places really) are active at one time
    - Represent Concurrency
- A PN is really a set of Multiple, Interacting FSMs all integrated and hidden within a single Graph!
  - Compact and Convenient
  - Ideal for Static Verification