Lecture 12

Asynchronous Circuits

Computer Systems Laboratory
Stanford University
horowitz@stanford.edu

Overview

- Readings
  - (To be announced)
- Today's topics
  - Asynchronous circuits benefits
  - Data exchange as a fundamental operation
  - Data-bundling
  - Self-timed circuits

Why Use Clocks?

- We've seen how hard it is to distribute a global clock over a die
  - Takes careful engineering to minimize the delay difference of clocks
  - Overhead associated with latches and flops
  - What if we got rid of the global clock altogether?
- Asynchronous circuits don't use a global clock; they handshake
  - Tell the next block that you have data ready
  - The next block acknowledges that it can process the data
  - Only then will you move your data onto the next block
- Asynchrony can happen over different levels of granularity
  - System-on-Chip with asynchronous interfaces (CPU, $$, Co-proc)
  - Functional blocks within a compute core
  - Datapaths within a functional block
  - Individual bitslices

The Devil You Know vs. The Devil You Don't

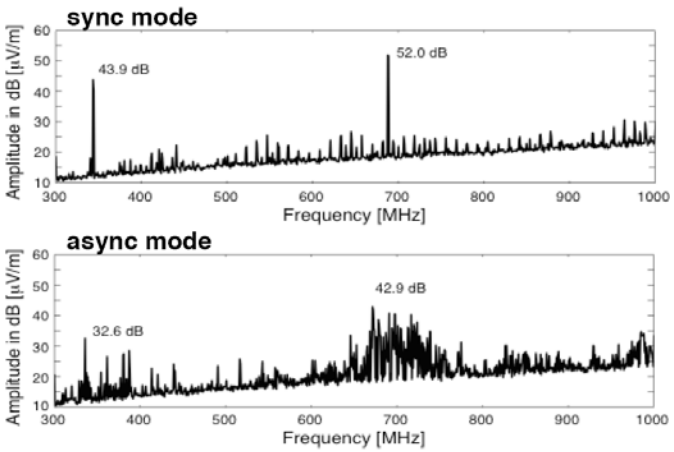
- When you evaluate a new technique / technology
  - Remember there are no perfect systems
  - There are no perfect circuits
- Remember to ask questions about:
  - What assumptions are they making that enable this method
  - What are they not telling us
  - Where are the weaknesses of the approach
- We will take a critical look at asynchronous circuits
  - Always start with the advantages
  - Then look at the stuff that is problematic
  - And end with a discussion of trade-offs between sync and async

Potential Benefits of Asynchronous Circuits

- Average-case behavior, not worst-case behavior
  - A synchronous circuit's cycle time must exceed the worst-case path
  - An asynchronous circuit runs as fast as it can
- Insensitivity to variations in data delays and clock delivery
  - Depends on the kind of asynchronous circuit style
  - Some forms of async circuits use the "moral equivalent of a clock"
    - They still care about "clock" uncertainty
- Automatic handling of overflow/underflow conditions
  - Overflow: If full, the next stage won't give me the OK to send data
  - Underflow: If I have no data, I won't send requests to the next stage

More Benefits of Asynchronous Circuits

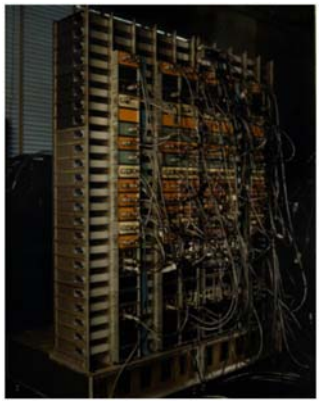
- Spread-spectrum EMI without spikes from a clock
  - Measured noise from a FIFO testchip from Sun Labs



Source: Ebergen, SunLabs

More Benefits of Asynchronous Circuits

- Modularity and replaceability



Macromodules ('66-'74), built at Washington University by Wes Clark and Charlie Molnar.

Any system that was mechanically connect-able was timing-correct-by-construction, using asynchronous circuits.

Very upgradeable: plug in some faster parts, and the entire machine would speed up, a little bit.

Source: Ebergen, SunLabs

- Potentially lower energy; burn power only during computation
  - An asynch circuit is a synch circuit with **optimally** gated clocks

With All These Advantages …

Why haven't asynchronous circuits taken over the world?

- Because they have some "issues" of their own
  - We will talk about these when we describe the circuits
- Because there is a lot of money being made in sync design
  - And thus, there is an entire CAD infrastructure behind sync design

When To Use Asynchronous Circuits

- To interface between asynchronous clock domains
  - For example, between two differently-clocked chips
  - UltrasparcIIIi has an asynchronous FIFO b/w the CPU and memory
- To save as much power as possible
  - Smart cards get a burst of energy when waved inside a field
  - Do as much computation as possible with that energy
  - Handshake (Philips spinoff) doing asynch design for smart cards
- To implement a long-latency operation that doesn't need clocks
  - Floating-point division takes many cycles
  - HaL, Fujitsu, AMD all used variants of a Williams self-timed divider

Basic Asynchronous Event

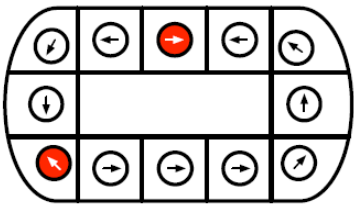
- An data-transfer event is fundamental to asynchronous circuits
  - When I have new data ("request")
  - And my successor is ready for new data ("acknowledge")
  - Then I push my data to my successor
  - And tell my predecessor I'm ready for new data ("acknowledge")
- Compare this to a synchronous system
  - When the clock arrives
  - I push my data to my successor (whether or not I am ready)
  - And grab data from my predecessor (whether or not it is ready)
- "Faith" versus "Measurement"
  - Negotiate in advance, or negotiate each cycle

Asynchronous Data Exchange

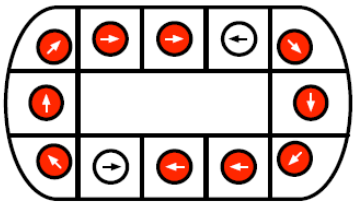
- This event can be thought of as a data exchange
  - I exchange my data with a “bubble” at my successor
- Data flows forward, and bubbles flow backwards
  - Data can only flow forward if a bubble is ahead of it
- Consider a FIFO (first-in, first-out) buffer
  - It’s full of data and has no bubbles: no throughput (overflow)
  - It has no data and is full of bubbles: no throughput (underflow)
  - Maximum throughput: half data, half bubbles, and they alternate

Data Exchanges as Bubbles and Tokens

- Consider a simple FIFO wrapped into a ring
  - Few tokens (red circles) can run freely, but with low throughput



  - Few holes (white circles) can also run freely; again low throughput

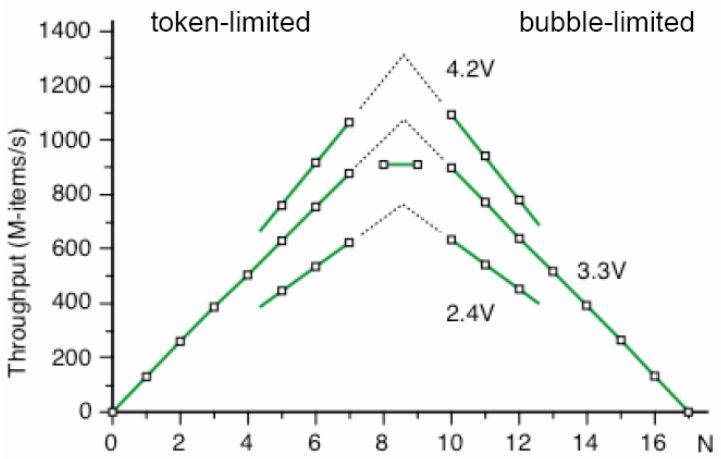


- Neither case is particularly fast

Source: Ebergen, SunLabs

Data Exchange Experiment

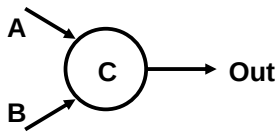
- Plot throughput versus input occupancy
  - Power plot looks just like this, too
  - Sun Labs testchip (350nm, 3.3V) measured results



Basic Asynchronous Exchange Circuit

- We need a circuit element to coordinate my data exchange
  - When I am full and successor is empty, then exchange
  - Otherwise do nothing

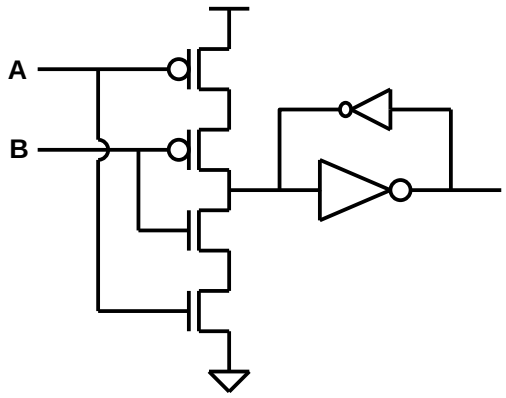
- David Muller (1960s) proposed a “completion element”
  - Two input gate that calculates consensus
  - If both inputs are 1, then output goes to 1
  - If both inputs are 0, then output goes to 0
  - Otherwise holds to last value



A	B	Out
0	0	0
0	1	(Hold)
1	0	(Hold)
1	1	1

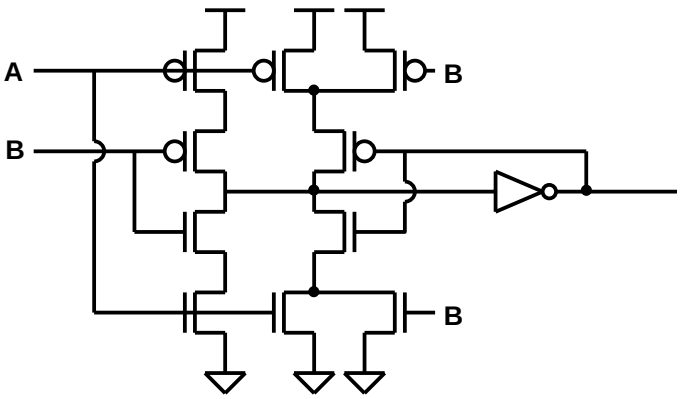
C-Element Implementation

- Basic dynamic C-element stores old data on inverters
  - Like a jam-latch: overpower weak feedback on a new consensus



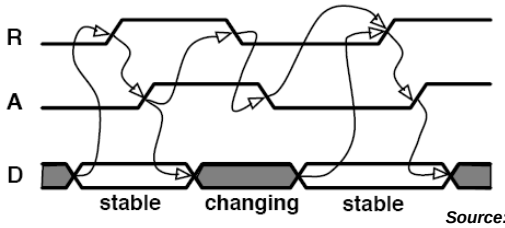
No-Fight C-Element Implementation

- Gated feedback prevents a fight when A=B



Level-Sensitive Control Schemes

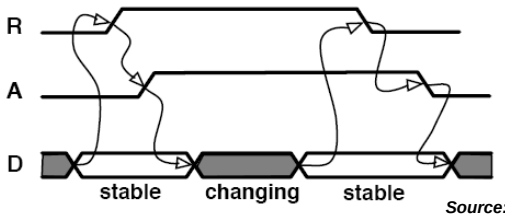
- Simplest level sensitive control is called “two-wire, RZ” control
  - Return-to-zero encoding; not very efficient
- Four phases must occur for each transaction
  - I have data, so I assert the Request line and send data
  - Receiver sees this, asserts its Acknowledge line, and grabs data
  - Once I see the Acknowledge line, I lower my Request line
  - Once the receiver sees my Request drop, it lowers Acknowledge



Source: Ebergen, SunLabs

Two Wire Transition Control

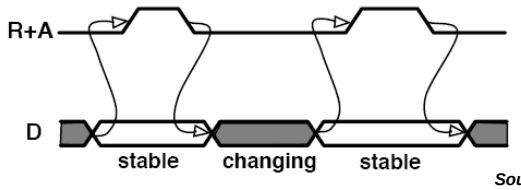
- Transition encoding (micropipelines) are one method of control
  - Also known as “two-wire, NRZ” control
  - Request and acknowledge each has its own wire
- A complete data transfer takes two phase transitions
  - I make a request when I have data by sending a transition on Req
  - I must keep the data “live” until I see a transition on Ack
- Some complexity in maintaining transition encoding paradigm



Source: Ebergen, SunLabs

Optimized Level-Sensitive Control

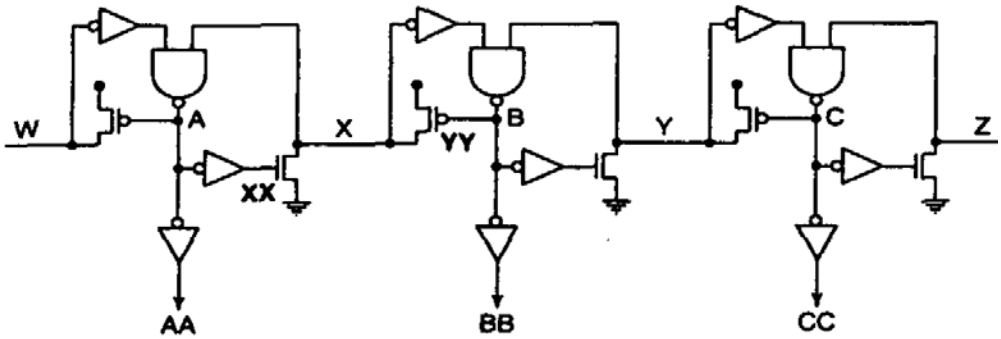
- Eliminate a wire and some transitions: “single-track, RZ”
  - Put the request and acknowledge signals on the same wire
- Two events occur for each transaction
  - I have data, so I pull up on the Request line and send data
    - Then I “let go” of the high request line; weak keepers hold it stable
  - Receiver sees Request go high and processes data
  - Receiver then pulls down on the Request line (acknowledgement)
    - Then it “lets go” of the low request line; same keepers hold it stable



Source: Ebergen, SunLabs

Example of This Control: GasP

- Circuit family from Sun Labs, using convention that low=Full
  - Based on asP (asynchronous symmetric Pulse) family
  - Not shown are the weak keepers on X, Y, and Z to hold state
  - NAND is built as a 2-high nMOS stack and a self-resetting pMOS
  - Built and tested on a number of prototype chips: runs quite fast



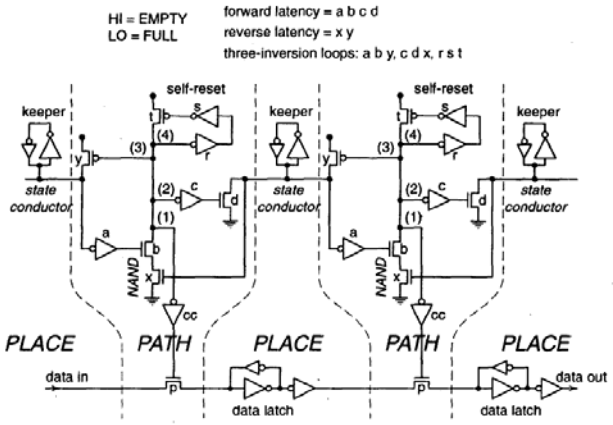
Source: Sutherland, ASYNC ’01

GasP Example Con't

- Expand the cells to see more closely why this is fast
  - Forward effort model = 2/3 for "Nand" and 1/3 for pulldown (d)
  - Backwards effort model = 2/3 for "Nand" and 2/3 for pullup (y)

- Fast, but at what cost?
  - Control wire is dynamic
    - So it's noise-sensitive
    - Yet it's routed along data

- Works well for FIFOs
  - Complex circuits are hard
  - Branch, merge, fork

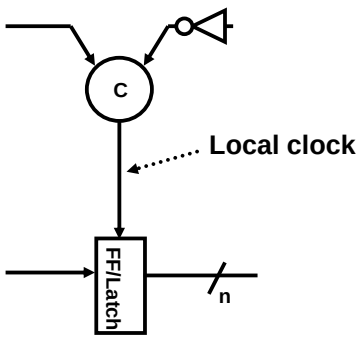


Source: Sutherland, ASYNC '01

But wait… Aren't We Generating Clocks?

- These control schemes all generate local clocks to drive flops
  - Still considered "asynchronous" since clocks are locally generated
  - Asynchrony doesn't mean we have no clocks, just no global clocks

- More exactly, these are called **data-bundling** schemes
  - The control/clk is explicit, and it rides alongside a "bundle" of data
  - But this leads to path matching constraints

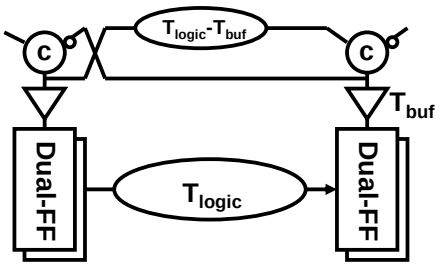


Data-Bundling Constraints

- What if the data path has delay due to gates or logic?
  - Clock path must track the logic delay, or else it gets “unbundled”
  - Too-early clock leads to failure; too-late clock slows system down
- Note that you still have max- and min-path timing issues
  - Only difference is that clocks are not globally generated
  - A few experienced clock designers vs many less experienced ones
- Locally generated clocks can also have some skew
  - If they span a wide datapath, the RC delay can be large

More Data-Bundling Constraints

- What if we have to drive lots of flops (say, 64)?
  - Clock path must be buffered, which costs delay
  - This delay must be accounted for in the clock path
- So launch the clock a little early, to give it time to buffer up
  - Called “kiting” the clock
  - “Kiting” = financial practice of writing checks before money arrives



More Data-Bundling

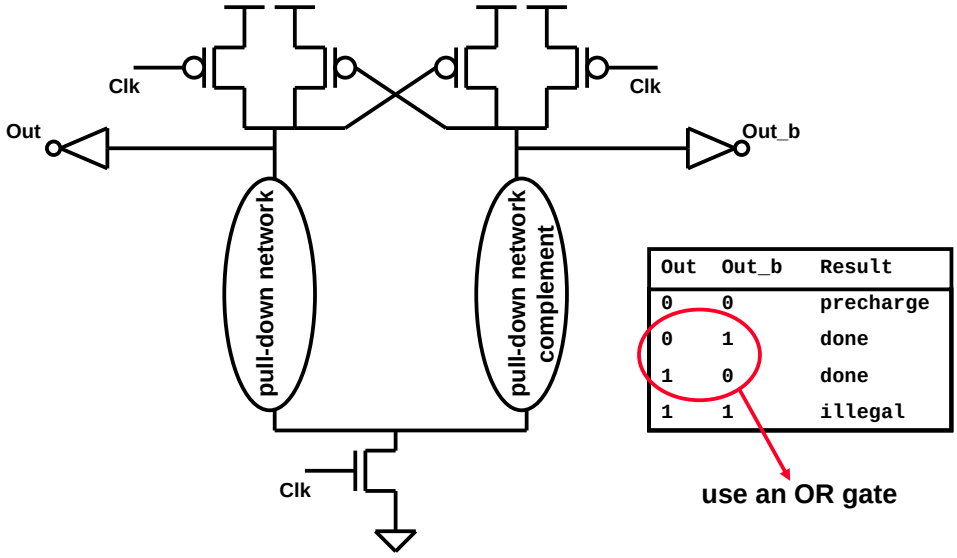
- Matching paths is tricky (it is what makes global clocks hard)
  - Data delays can depend on actual data values
    - So matching data paths is much harder than matching clock paths
  - Clock paths must NOT be early; late is okay
  - So usually use worst-case data delay for the matching clock delay
    - Asynchronous circuits lose some of their “average case” benefit
    - Worse device matching with technology means this gets harder
- FIFOs are still a great application for asynchronous circuits
  - Data delays are simple to match
  - But complex logic (ex: ALUs) are hard to match clock and data
- Note that we still rely a lot on faith. Too much?
  - What about direct measurement of individual data wires?
  - Avoid the data-bundling problems

Self-Timed Pipelines

- In self-timed pipelines we let each gate negotiate its timing
  - No separate control channels or local clocks
- We need a circuit style that will help us do this negotiation
  - One that allows direct measurement of its completion status
- Dual-rail domino is a candidate
  - Normally has two outputs, both driven low during precharge
  - Then only one output rises during the evaluation (never both)
  - Take the logical “or” of the two outputs
    - If still 0, then the gate is still in precharge
    - If 1, then the gate is done
  - It also holds its own state during input precharge
    - You don’t need to have any explicit latches in the design

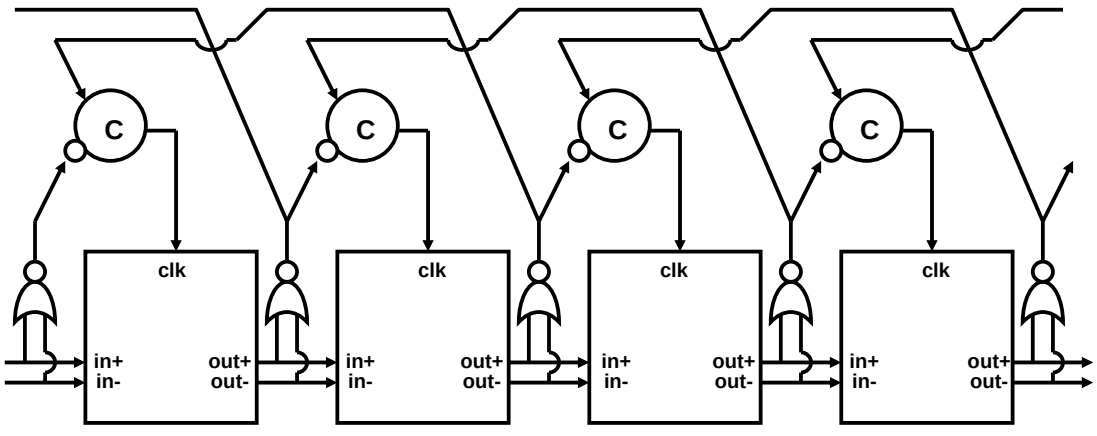
Dual-Rail Domino

- This circuit announces when it's done evaluating



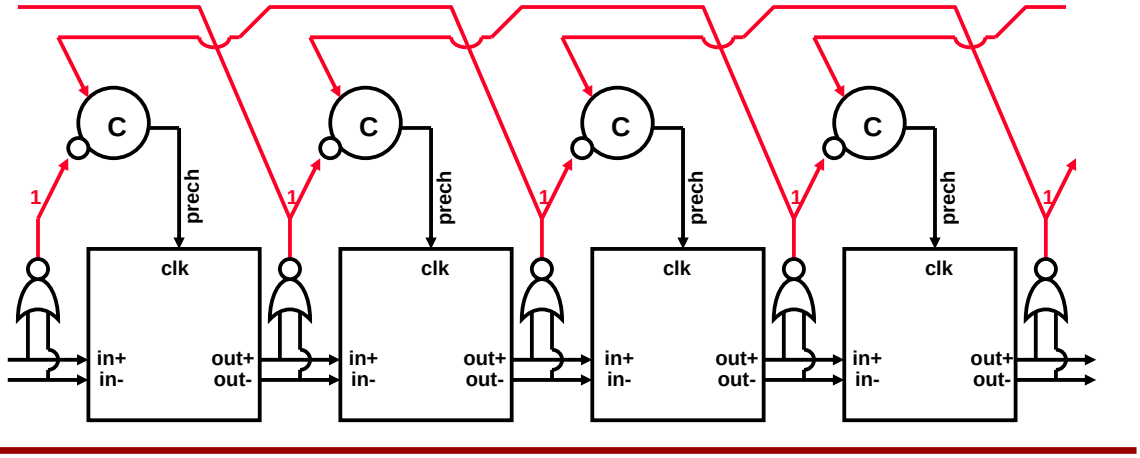
A Self-Timed Pipeline

- Each box is a logical function that signals completion
  - Signal enables the next gate ("You have data")
  - Signal resets a prior gate ("Your data has been consumed safely")
  - From Williams, '91



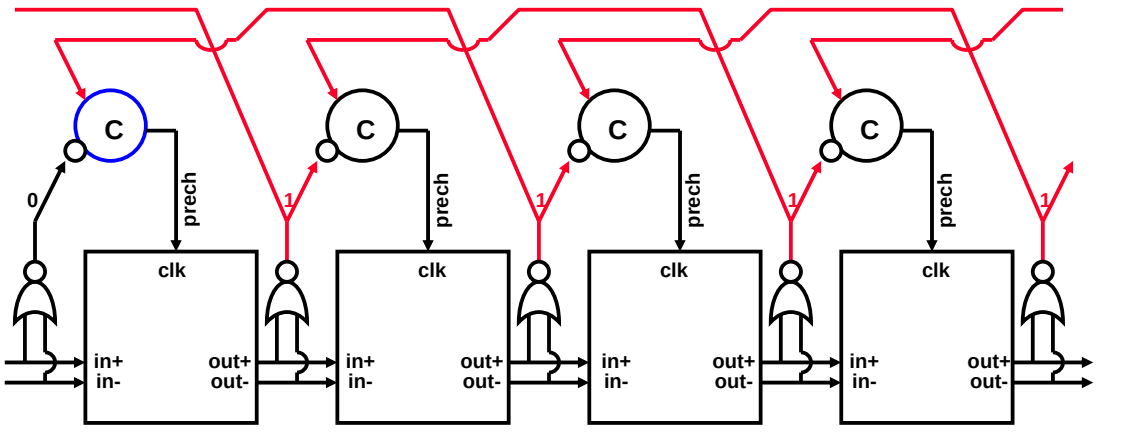
A Self-Timed Pipeline

- Starting condition puts all gates in precharge (clk is low)
- Assume basic inputs to the entire block are monotonic (rising)



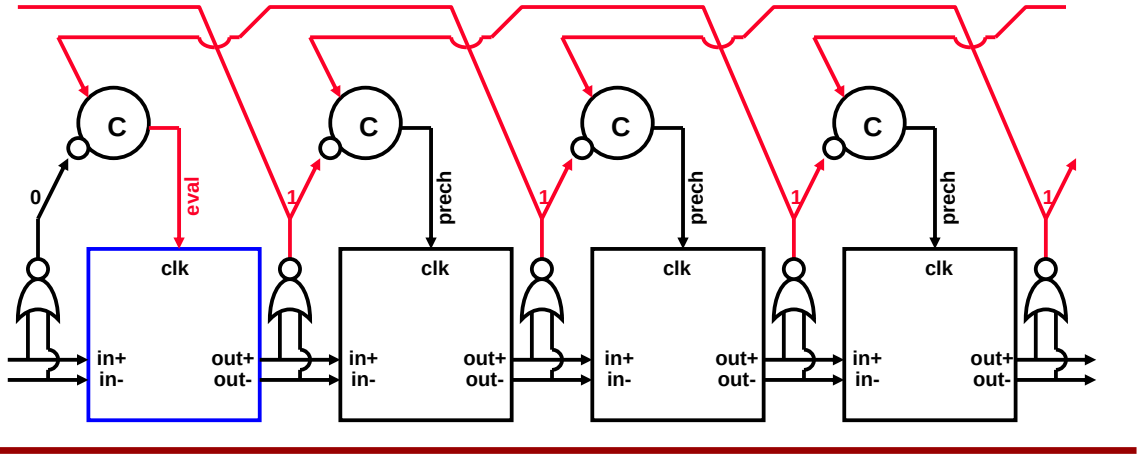
A Self-Timed Pipeline

- Data enters at the far left and the NOR gate flips
  - This activates the C-element



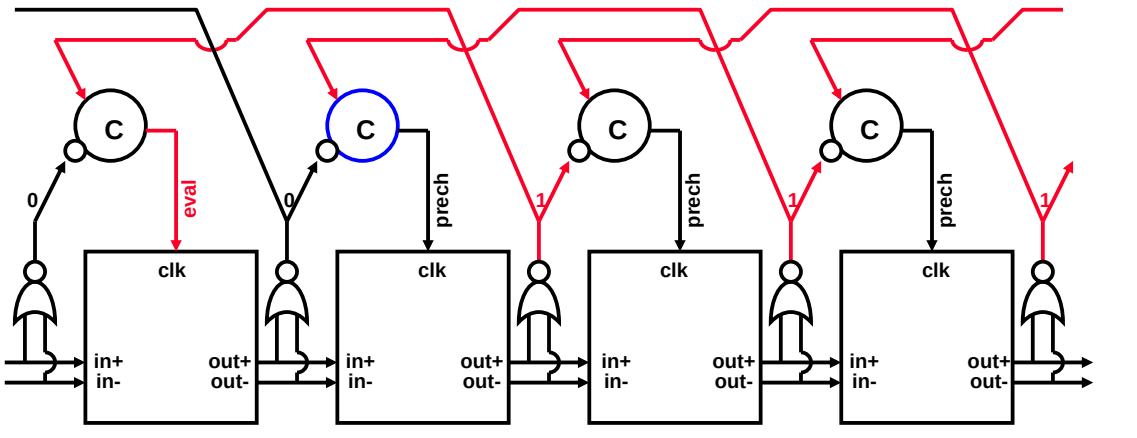
A Self-Timed Pipeline

- First logic block goes into evaluate



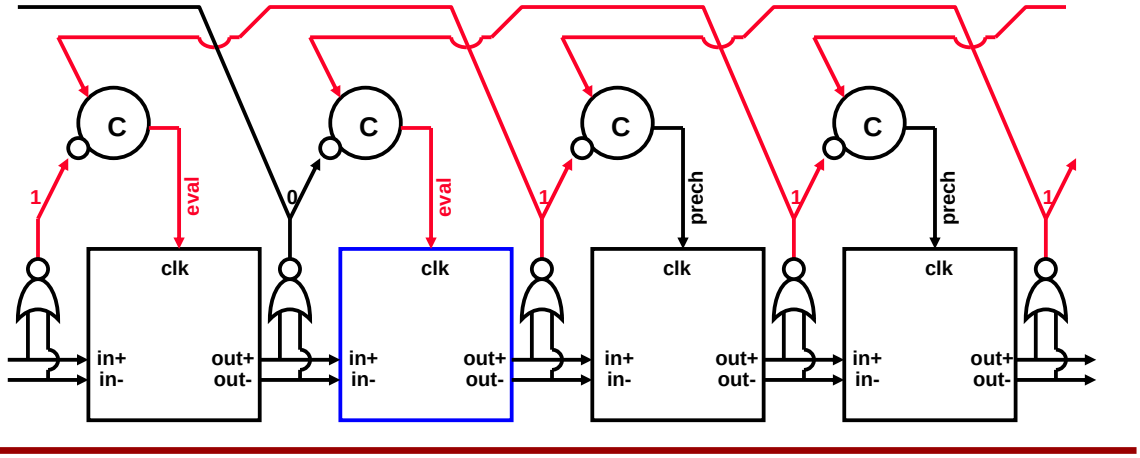
A Self-Timed Pipeline

- Second block's inputs are now valid, which flips its NOR gate
  - C-element flips
  - Primary inputs can be deasserted



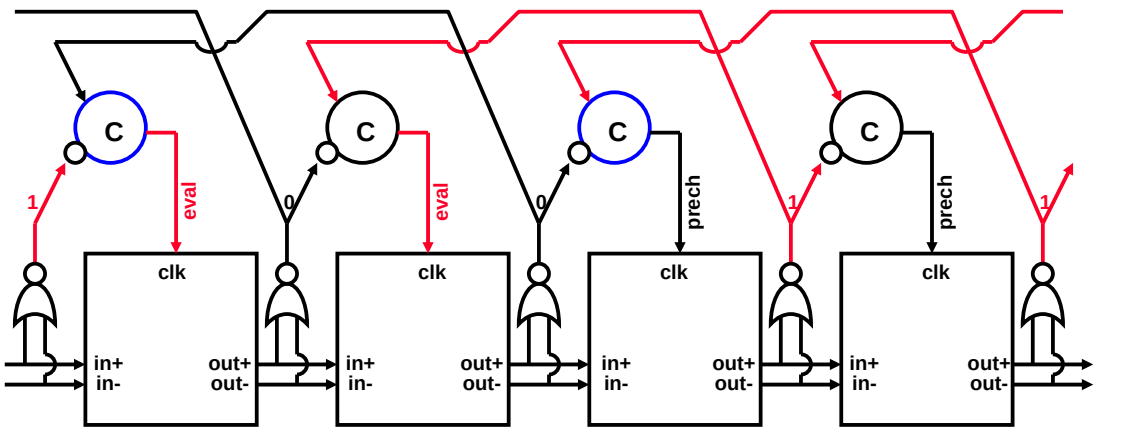
A Self-Timed Pipeline

- Second block goes into evaluate
- Primary inputs are deasserted, flipping the first NOR gate



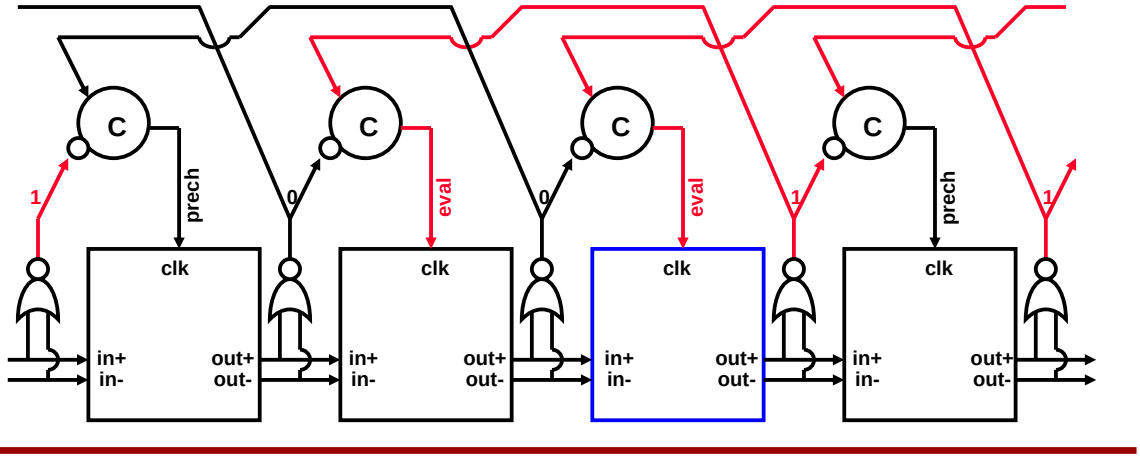
A Self-Timed Pipeline

- Third block's inputs are valid, so the third NOR gate flips
  - And the third C-element flips
- Also, the first C-element flips



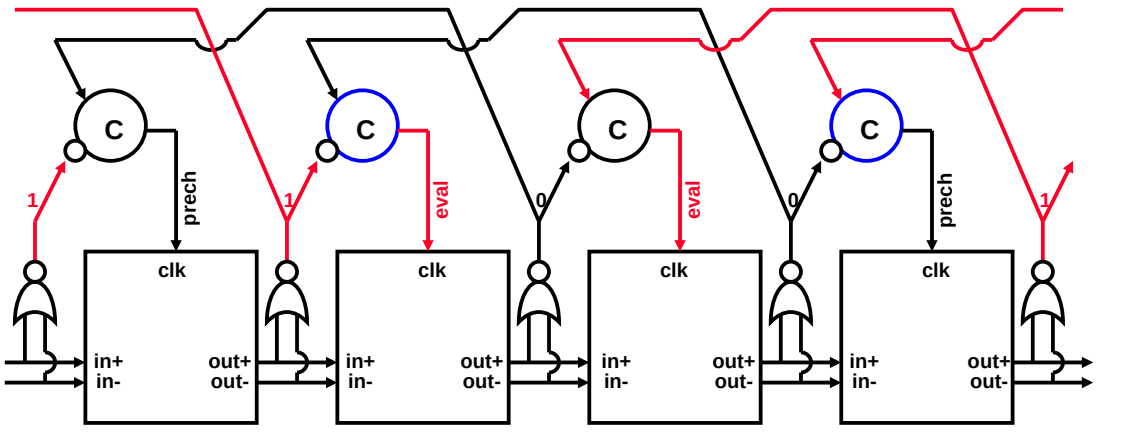
A Self-Timed Pipeline

- Third block goes into evaluate
- First block drops back into precharge



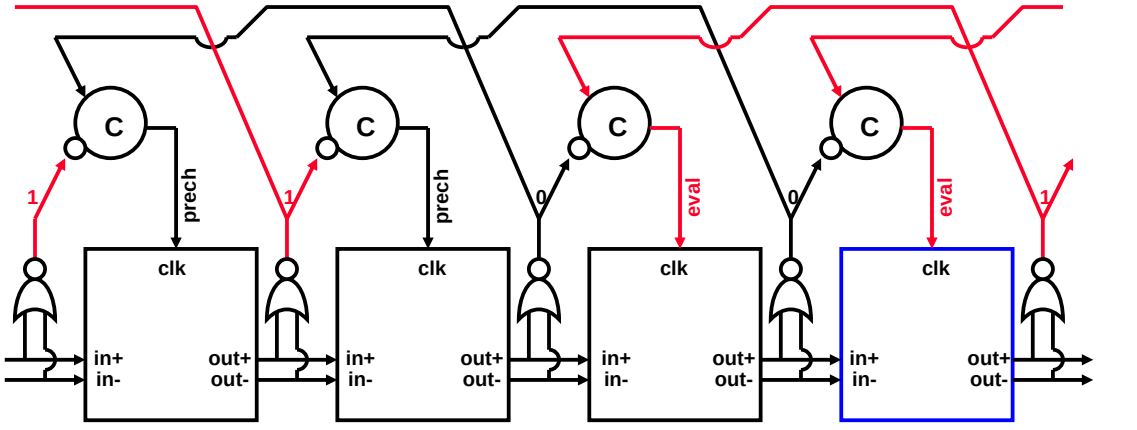
A Self-Timed Pipeline

- Fourth blocks inputs are valid, tripping the fourth NOR gate
- First block outputs are now precharged, tripping second NOR
  - Both C-elements fire



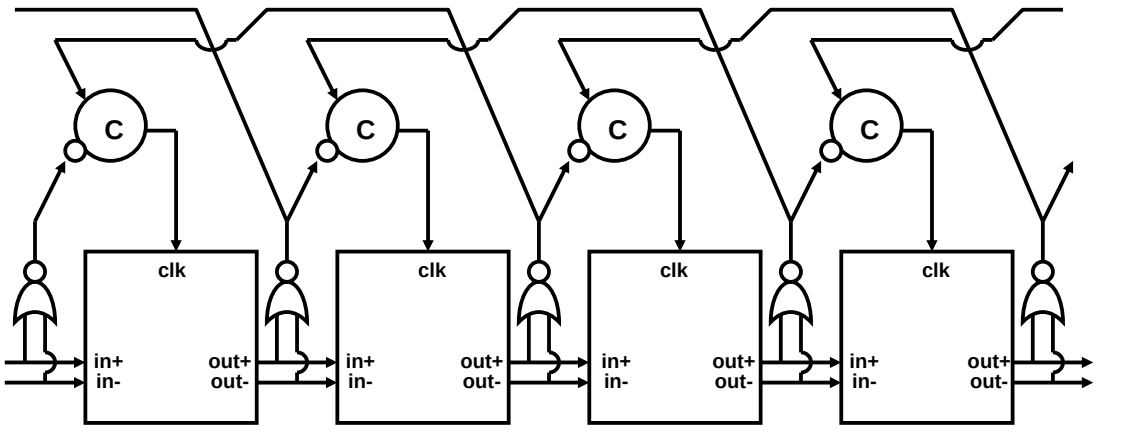
A Self-Timed Pipeline

- Fourth block goes into evaluate
- Second block goes into precharge
- …and so on and so forth



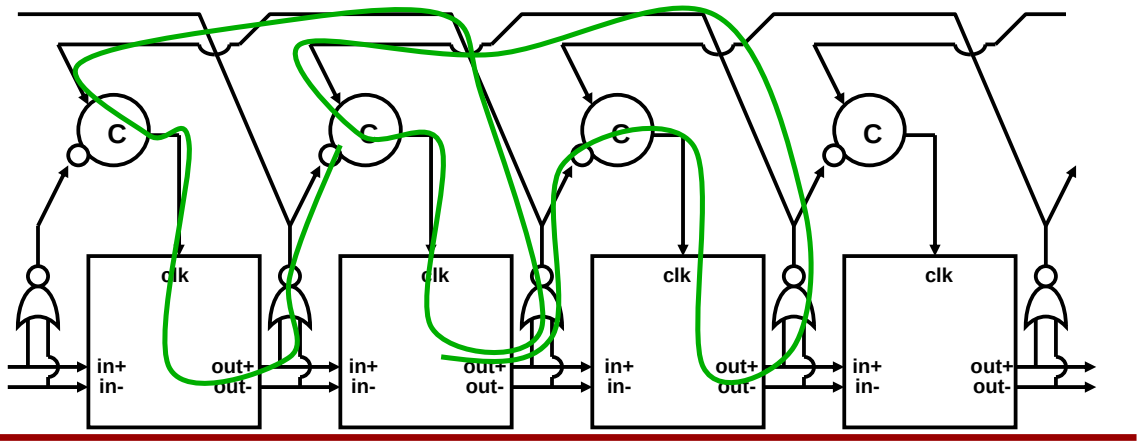
A “Safe” Pipeline

- A gate won’t precharge until its inputs have been consumed
  - Until the next block’s outputs are valid
- A gate can’t evaluate until its inputs are ready



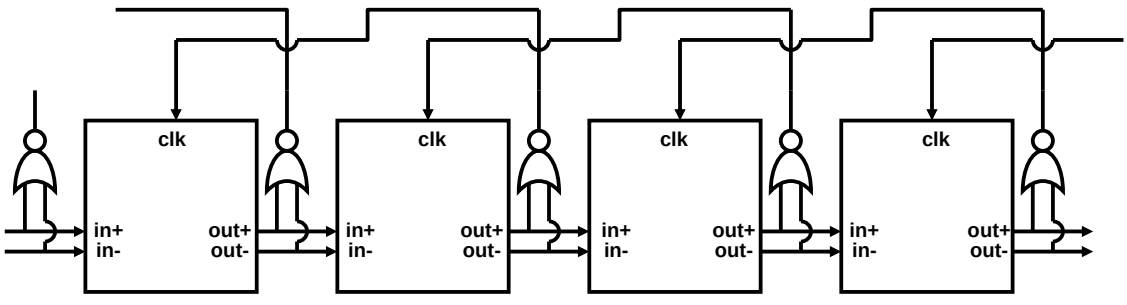
A "Slow" Pipeline

- Forward delay includes some control elements: $T_{eval} + T_{nor} + T_c$
- Data cycle time consists of lots of elements
  - $T_{eval} + T_{nor} + T_c + T_{eval} + T_{nor} + T_c + T_{prech} + T_{nor} + T_c + T_{eval} + T_{nor} + T_c$
  - $3T_{eval} + 4T_{nor} + 4T_c + T_{prech}$



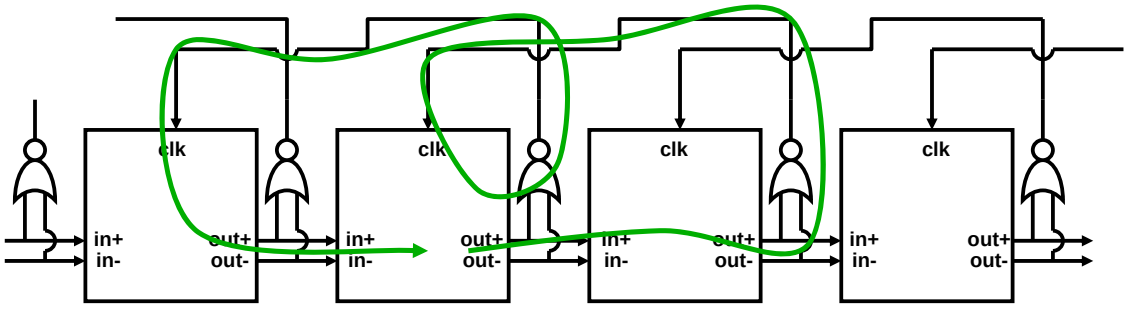
A Faster Pipeline

- A variant, called "PS0," of the self-timed pipeline
  - Relaxes the reset constraint



A Faster Pipeline

- Forward delay contains only T_{eval}
  - Often called “zero-overhead” self-timing
- Data cycle time is shorter, too
  - $T_{eval} + T_{eval} + T_{nor} + T_{prech} + T_{nor} + T_{eval} = 3T_{eval} + 2T_{nor} + T_{prech}$



Self-Timed Pipeline Control Overhead

- At what level do you do the sequence control
  - At the bit/gate level has the lowest time overhead
    - Just need to look at the NOR’s of your inputs
    - But you have the highest gate overhead
  - At the block level, the logical overhead is not high
    - But the delay to know input has changed is high
    - Bubbles move backward very slowly
    - So pipeline must be mostly empty for bubbles not to limit performance
- The design requires dual rail, with a transition probability of 0.5
  - Need to know that the gate has evaluated
  - Concern about power issues

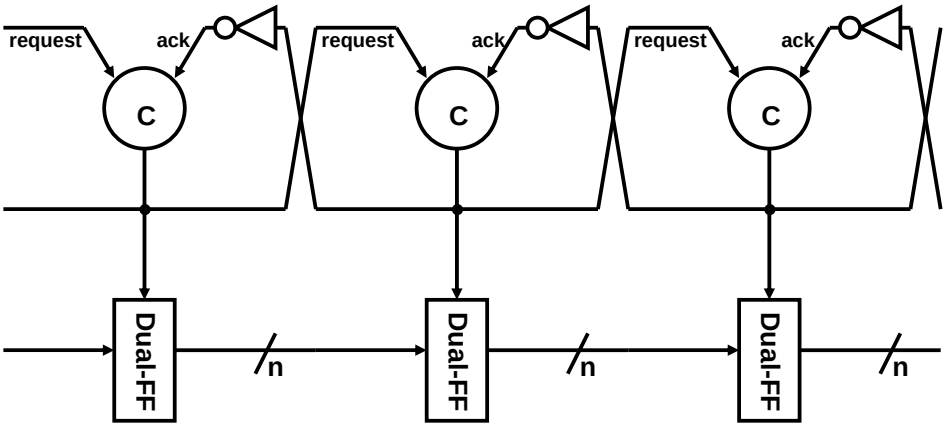
Bottom Line

- Async designs are used all the time
  - Generally in part of a larger synchronous block
    - Ex: SRAMS, FIFOs,
- Distributing global clock is not hard enough
  - To make most people want to leave their synchronous tools
  - And their tape-off check lists
- Asynchronous design has some promising features
  - And some delay matching / overhead issues of its own
  - Has not made it to the mainstream
  - But has a loyal following

Backup slides

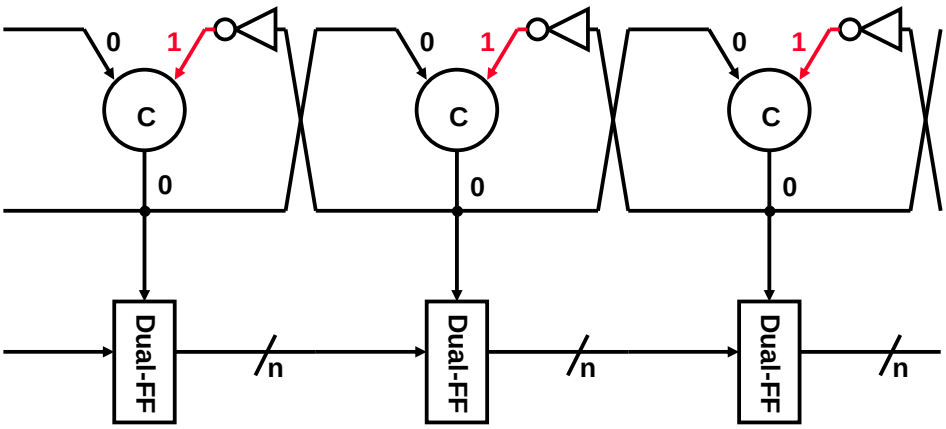
Controlling Exchanges Using C-Elements

- A “micropipeline” (or “Chain of Rendezvous”) control
  - If stages n and n+1 differ, then exchange
  - A transition-encoded control scheme



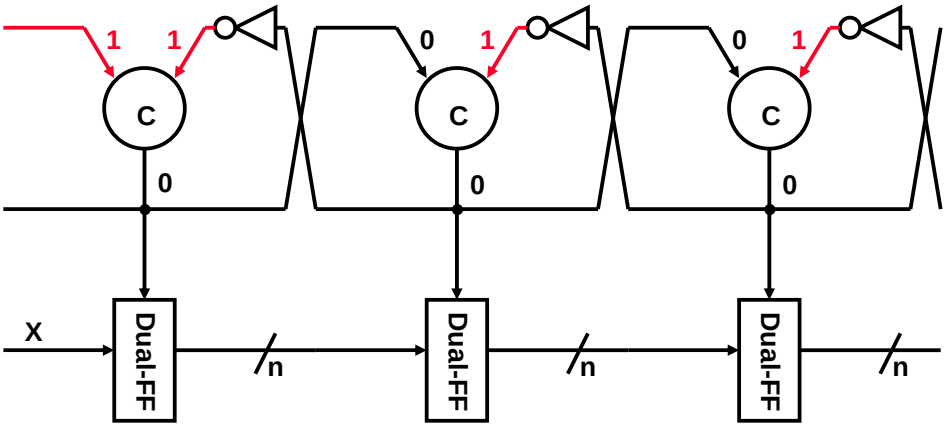
Controlling Exchanges Using C-Elements

- No action, waiting for input from the left



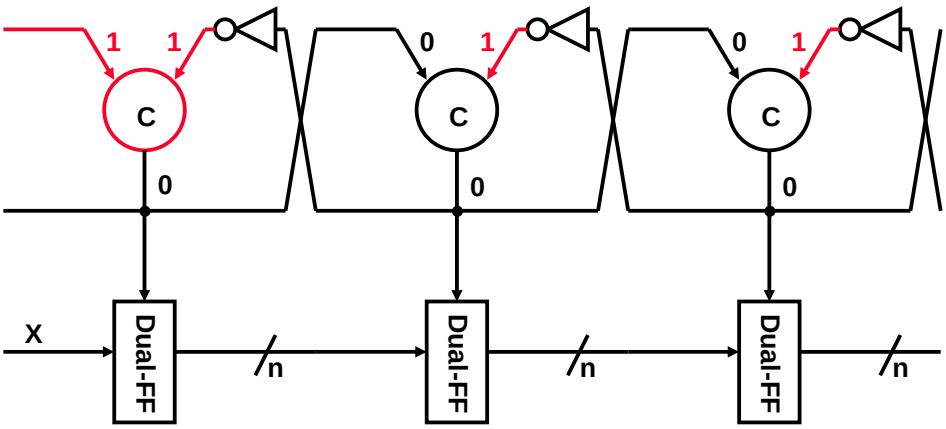
Controlling Exchanges Using C-Elements

- Input “X” arrives with a request transition



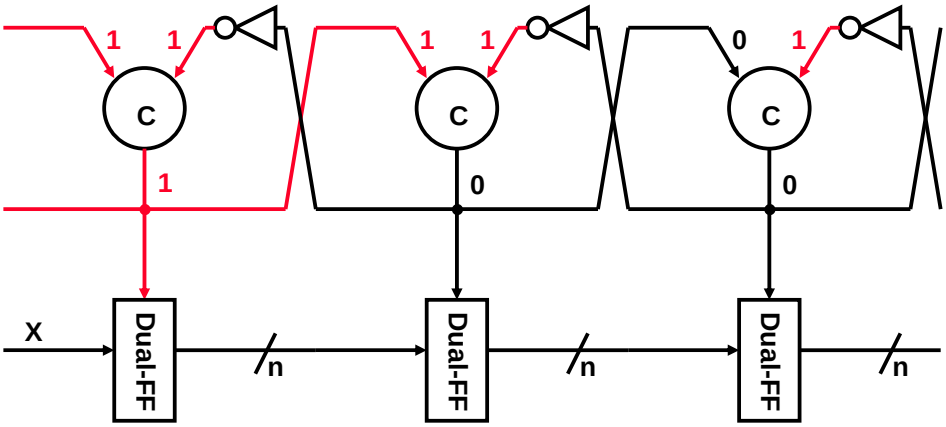
Controlling Exchanges Using C-Elements

- C-element fires…



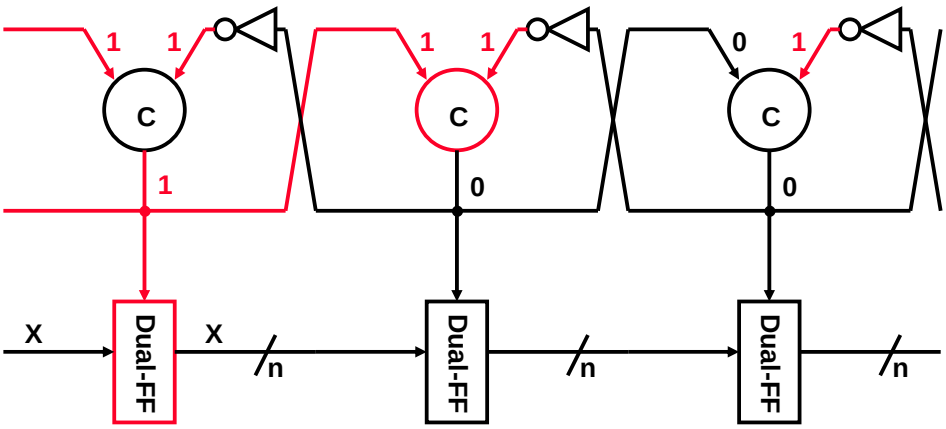
Controlling Exchanges Using C-Elements

- Clock to the flop fires
  - Sends a request transition to the next stage
  - Sends an acknowledgement transition backwards



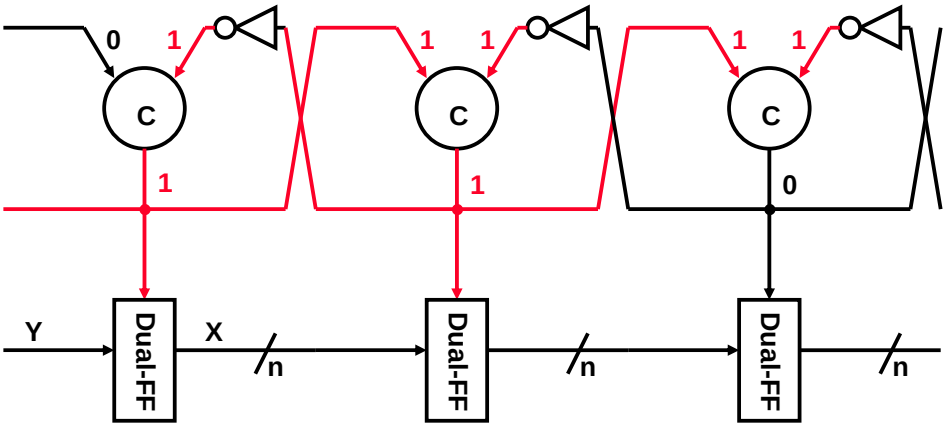
Controlling Exchanges Using C-Elements

- Flop fires and “X” moves ahead
- Next c-element fires



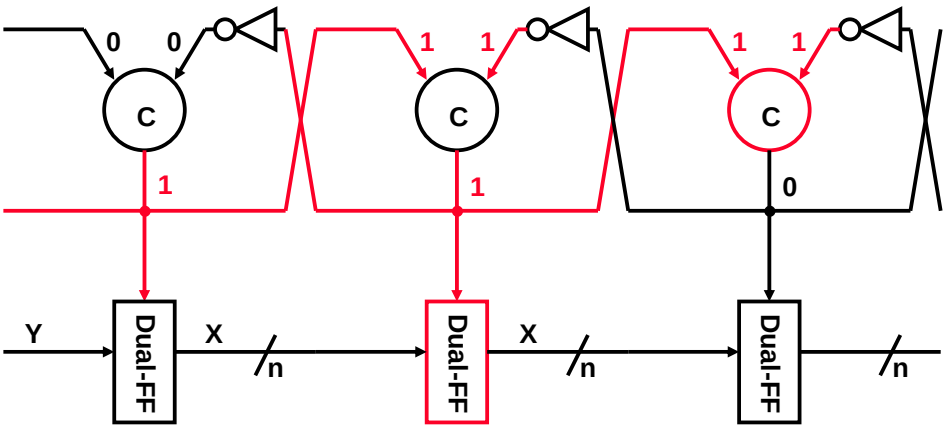
Controlling Exchanges Using C-Elements

- Next stage's flop clock fires
  - Sends request and acknowledgement transitions out
- Also let's say that new data appears at left (1→0 transition)



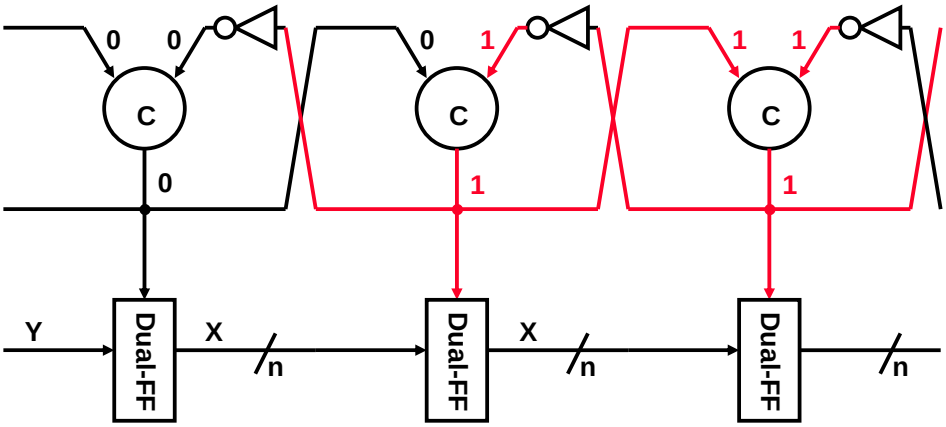
Controlling Exchanges Using C-Elements

- Second stage flop fires and propagates "X" ahead
- Third stage C-element fires
- First stage's inverter pulls low and fires the C-element



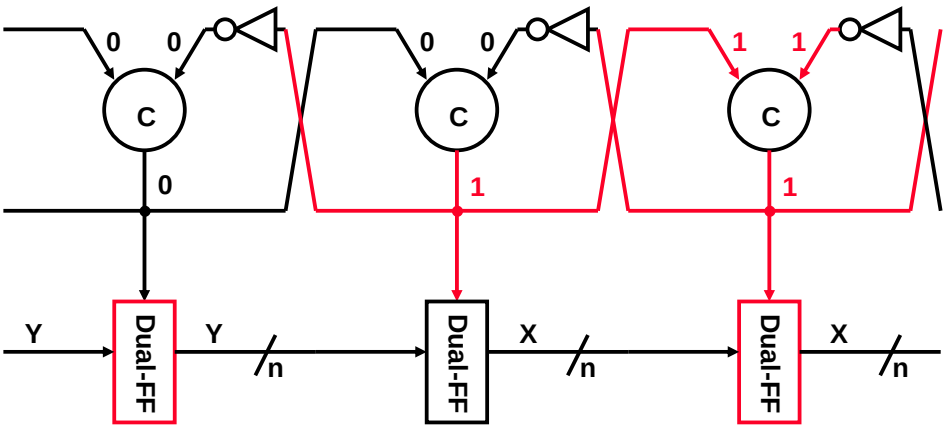
Controlling Exchanges Using C-Elements

- First stage's flop clock fires (1→0 transition)
- Third stage's flop clock fires (0→1 transition)



Controlling Exchanges Using C-Elements

- First and third flops fire; "X" and "Y" both propagate
- Second stage's c-element fires (1→0 transition)



Controlling Exchanges Using C-Elements

- Second flop clock fires (1→0 transition)
- …and so on and so forth