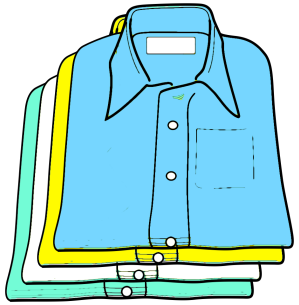
7. Performance Measures

6.004x Computation Structures
Part 1 – Digital Circuits

Forget circuits… Let's Solve a Real Problem

INPUT:
dirty laundry

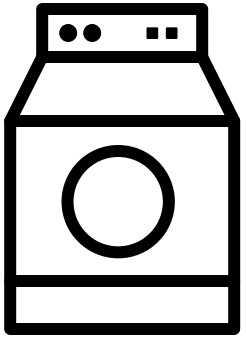
OUTPUT:
6 more weeks



Device: Washer

Function: Fill, Agitate, Spin

Washer_{PD} = 30 mins



Device: Dryer

Function: Heat, Spin

Dryer_{PD} = 60 mins

One Load At a Time

Everyone knows that the real reason that we put off doing laundry so long is not because we procrastinate, are lazy, or even have better things to do.

The fact is, doing one load at a time is not smart.

Step 1:

Step 2:

$Total_{PD} = Washer_{PD} + Dryer_{PD}$

$= \underline{\quad 90 \quad}$ mins

Doing N Loads of Laundry

Here's how they do laundry at Harvard, the "combinational" way.

Of course, this is just an urban legend. No one at Harvard actually *does* laundry. The butlers all arrive on Wednesday morning, pick up the dirty laundry and return it all pressed and starched in time for afternoon tea.

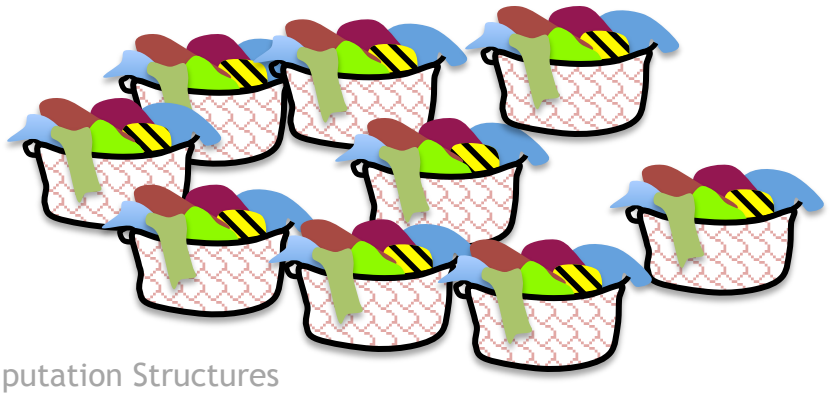
Step 1:

Step 2:

Step 3:

Step 4:

• • •



$Total_{PD} = N*(Washer_{PD} + Dryer_{PD})$

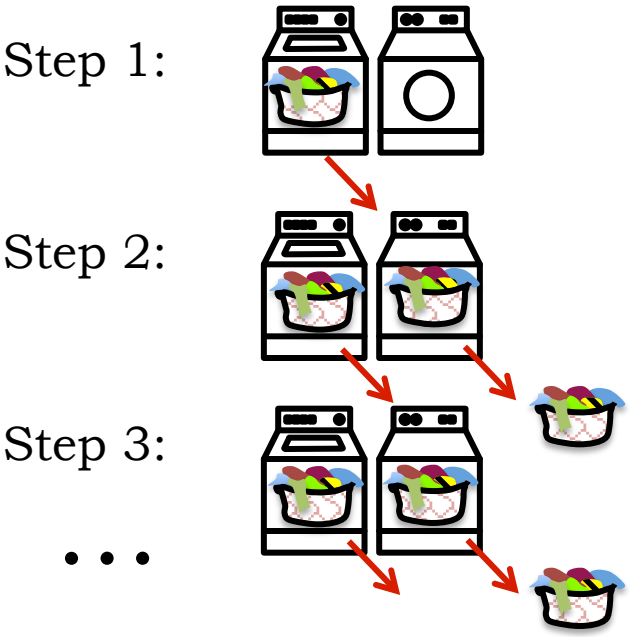
$= \underline{N*90}$ mins

Doing N Loads… The 6.004 Way

6.004 students "pipeline" the laundry process.

That's why we wait!

Actually, it's more like N*60 + 30 if we account for the startup transient correctly. When doing pipeline analysis, we're mostly interested in the "steady state" where we assume we have an infinite supply of inputs.



$$\text{Total}_{PD} = N * \text{Max}(\text{Washer}_{PD}, \text{Dryer}_{PD})$$

$$= \underline{\quad N*60 \quad} \text{ mins}$$

Performance Measures

Latency:

The delay from when an input is established until the output associated with that input becomes valid.

Harvard Laundry = 90 mins

6.004 Laundry = 120 mins ←

Assuming that the wash is started as soon as possible and waits (wet) in the washer until dryer is available.

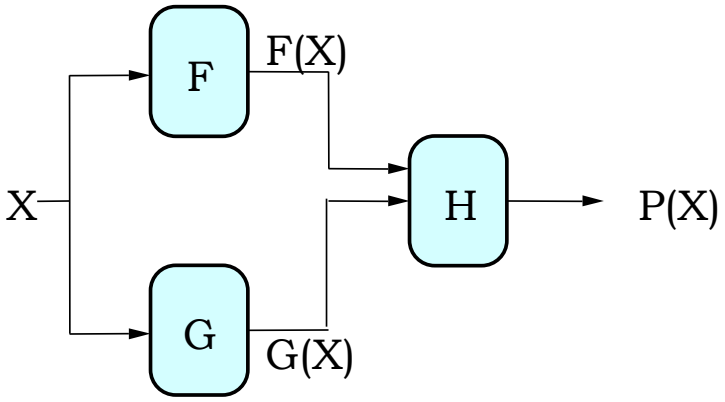
Throughput:

The *rate* at which inputs or outputs are processed.

Harvard Laundry = 1/90 outputs/min

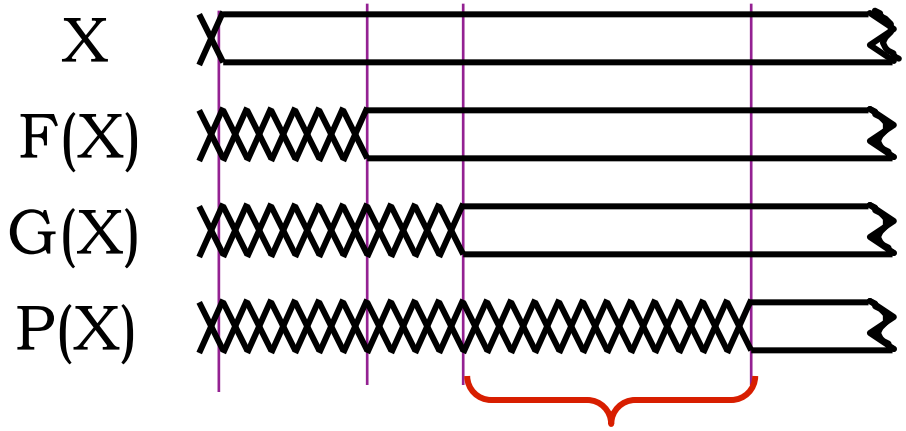
6.004 Laundry = 1/60 outputs/min

Okay, Back To Circuits...

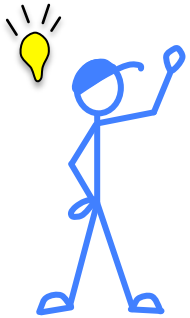


For combinational logic:
latency = t_{PD},
throughput = $1/t_{PD}$.

We can't get the answer faster, but are we making effective use of our hardware at all times?

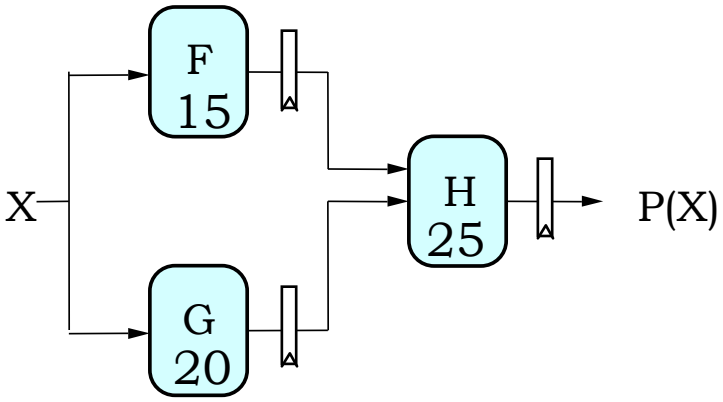


F & G are "idle", just holding their outputs stable while H performs its computation



Pipelined Circuits

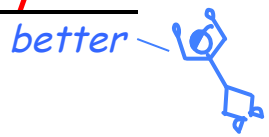
use registers to hold H's input stable!



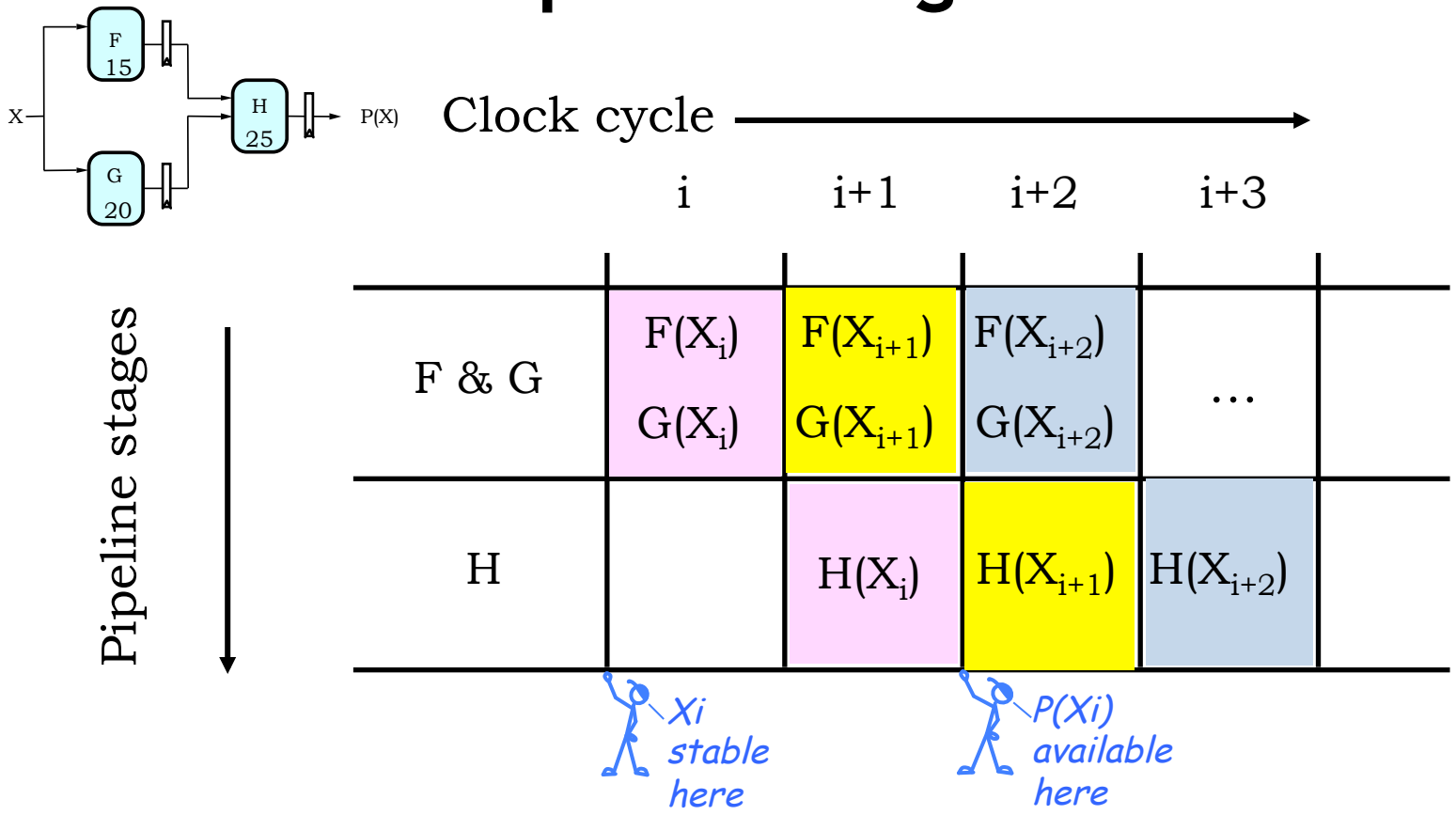
Now F & G can be working on input X_{i+1} while H is performing its computation on X_i. We've created a 2-stage *pipeline*: if we have a valid input X during clock cycle j, P(X) is valid during clock j+2.

Suppose F, G, H have propagation delays of 15, 20, 25 ns and we are using ideal zero-delay registers:

	latency	throughput
unpipelined	45	1/45
2-stage pipeline	50	1/25



Pipeline Diagrams



The results associated with a particular set of input data moves *diagonally* through the diagram, progressing through one pipeline stage each clock cycle.

Pipeline Conventions

DEFINITION:

A well-formed *K-Stage Pipeline* ("K-pipeline") is an acyclic circuit having exactly K registers on *every* path from an input to an output.

a COMBINATIONAL CIRCUIT is thus an 0-stage pipeline.

COMPOSITION CONVENTION:

Every pipeline stage, hence every K-Stage pipeline, has a register on its *OUTPUT* (not on its input).

ALWAYS:

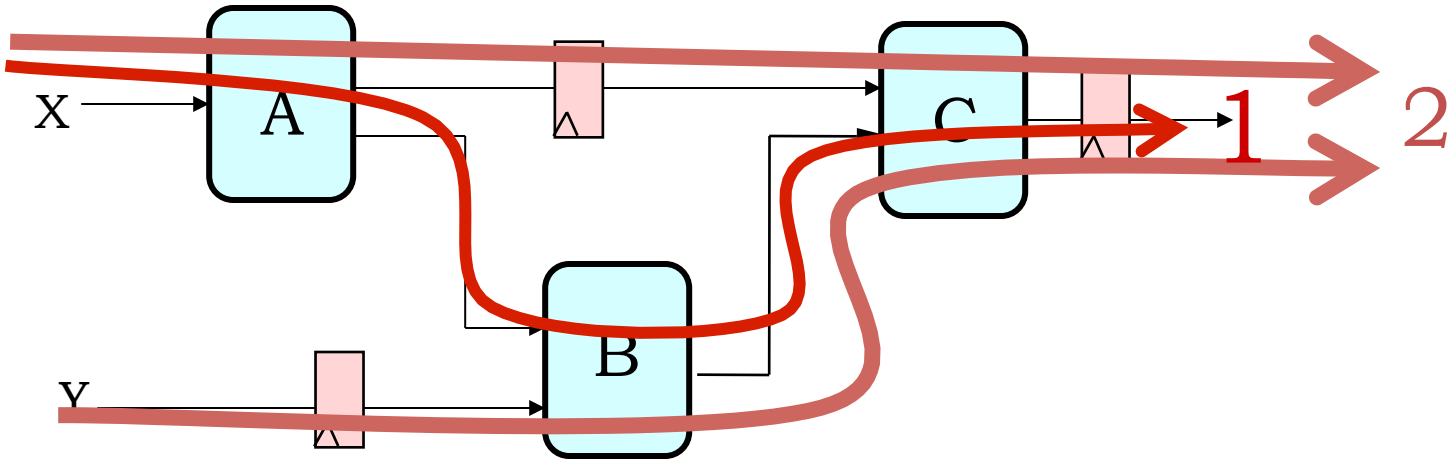
The CLOCK common to all registers must have a period sufficient to cover propagation over combinational paths PLUS (input) register t_{PD} PLUS (output) register t_{SETUP}.

The LATENCY of a K-pipeline is K times the period of the system's clock.

The THROUGHPUT of a K-pipeline is the frequency of the clock.

Ill-formed Pipelines

Consider a BAD job of pipelining:



For what value of K is the following circuit a K-Pipeline?

ANS: none

Problem:

Successive inputs get mixed: e.g., $B(A(X_{i+1}), Y_i)$. This happened because some paths from inputs to outputs have 2 registers, and some have only 1!

This CAN'T HAPPEN on a well-formed K pipeline!

A Pipelining Methodology

Step 1:

Draw a line that crosses every output in the circuit, and mark the endpoints as terminal points.

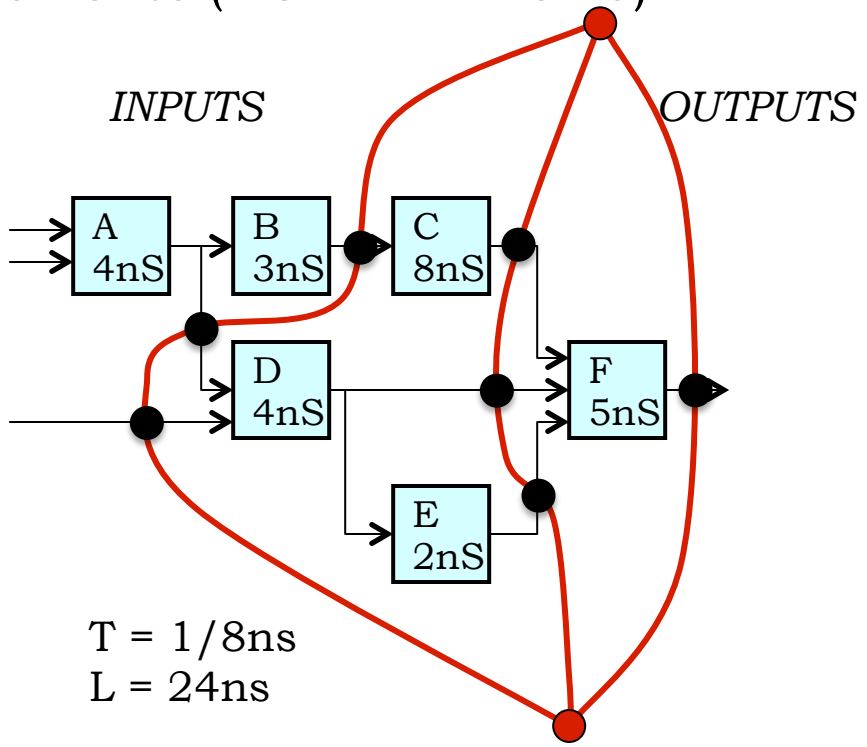
Step 2:

Continue to draw new lines between the terminal points across various circuit connections, ensuring that every connection crosses each line in the same direction. These lines demarcate *pipeline stages*.

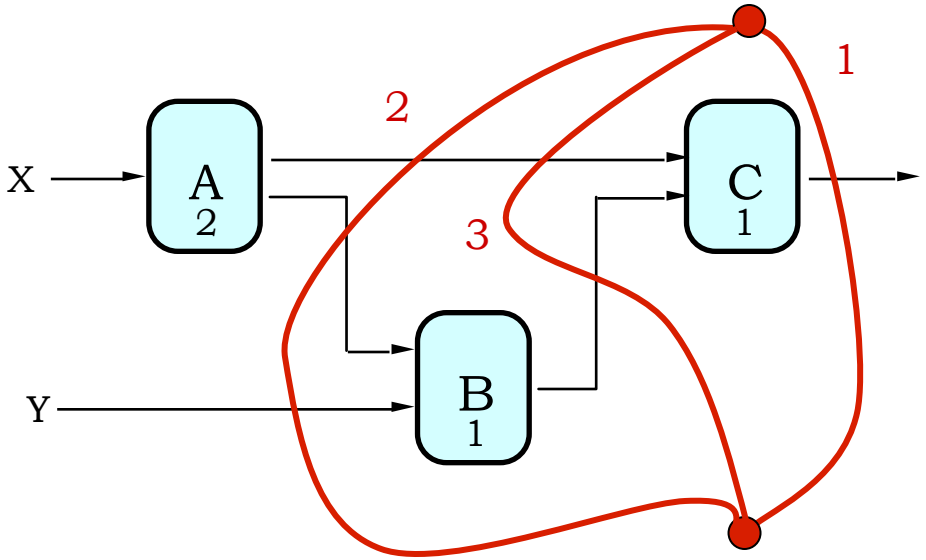
Adding a pipeline register at every point where a separating line crosses a connection will always generate a valid pipeline.

STRATEGY:

Focus your attention on placing pipelining registers around the slowest circuit elements (BOTTLENECKS).



Pipeline Example



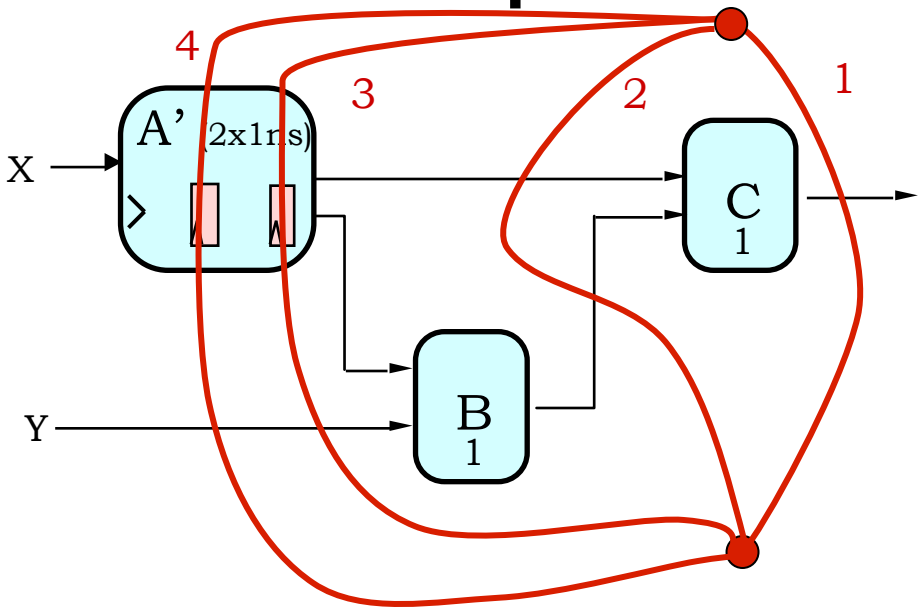
	LATENCY	THROUGHPUT
0-pipe:	4	1/4
1-pipe:	4	1/4
2-pipe:	4	1/2
3-pipe:	6	1/2

OBSERVATIONS:

- 1-pipeline improves neither L or T.
- T improved by breaking long combinational paths, allowing faster clock.
- Too many stages cost L, don't improve T.
- Back-to-back registers are often required to keep pipeline well-formed.

+ increase throughput
– increase latency
– "bottleneck" problem

Pipelined Components



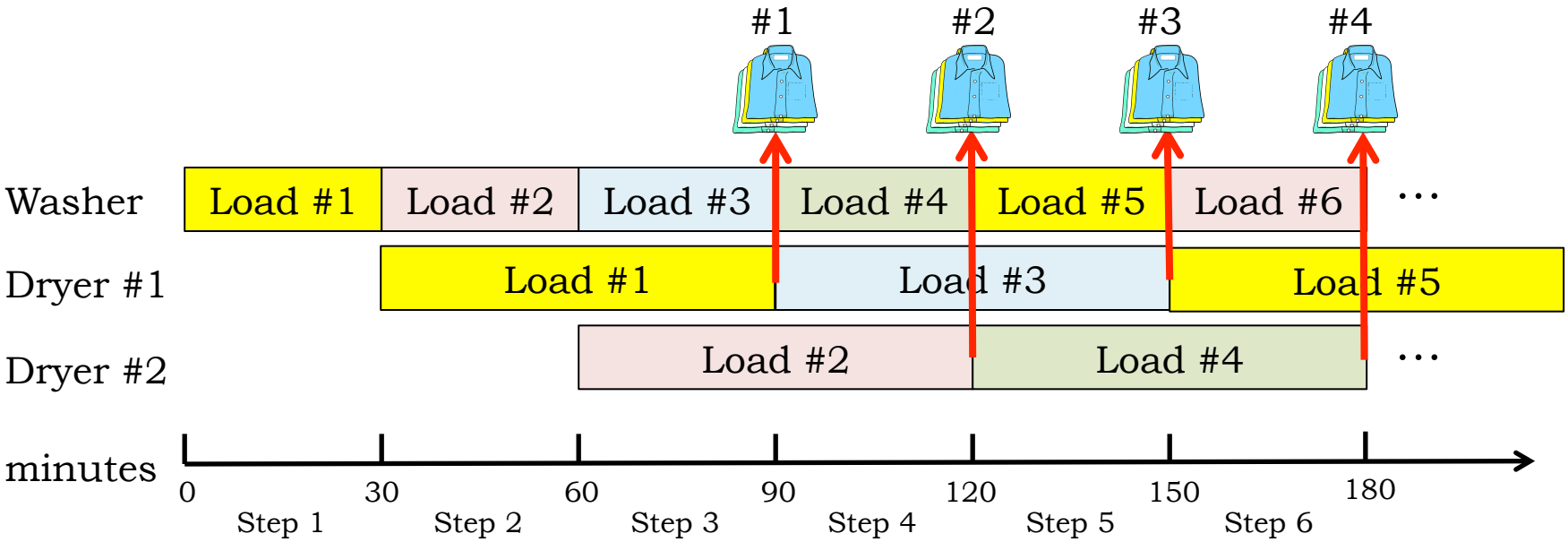
4-stage pipeline, throughput=1

Pipelined systems can be hierarchical:

- Replacing a slow combinational component with a k-pipe version may let us decrease the clock period
- Must account for new pipeline stages in our plan

How Do 6.004 Students Do Laundry?

They work around the bottleneck. First, they find a laundromat with two dryers for every washer. Then they use dryer #1 for odd-numbered wash loads and dryer #2 for even-numbered wash loads.



Throughput = 1/30 loads/min, Latency = 90 mins/load

Back To Our Bottleneck…

Recall our earlier example…

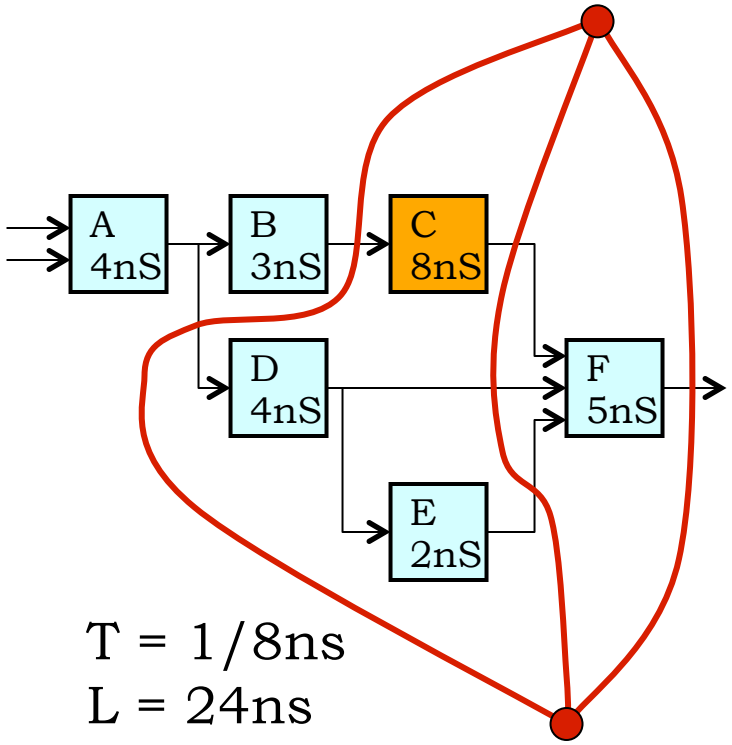
- C – the slowest component – limits clock period to 8 ns.
- HENCE throughput limited to 1/8ns.

We could improve throughput by

- Finding a pipelined version of C;

OR …

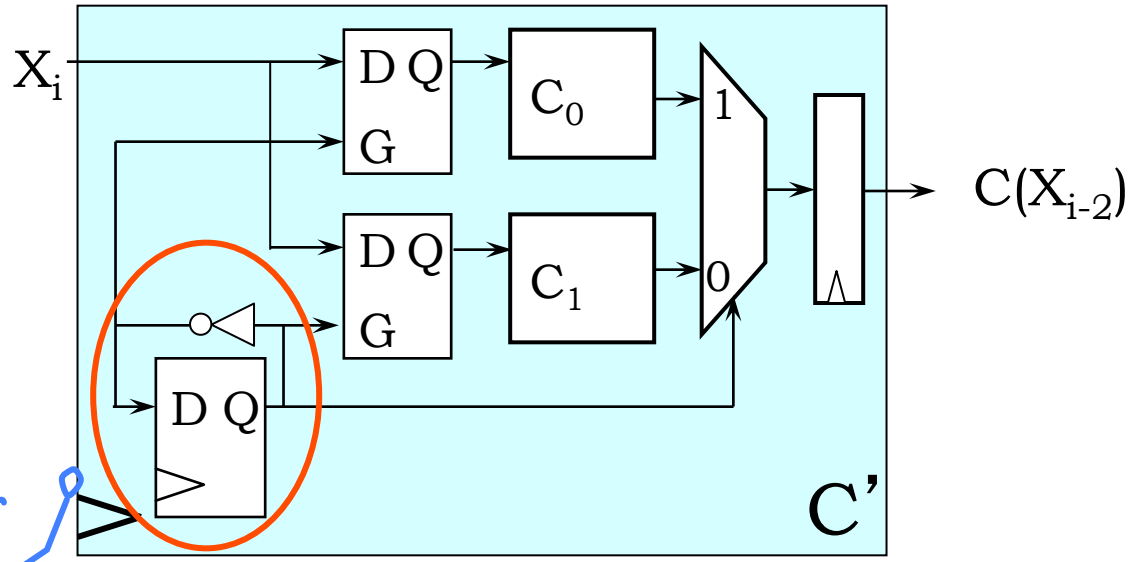
- *interleaving* multiple copies of C!



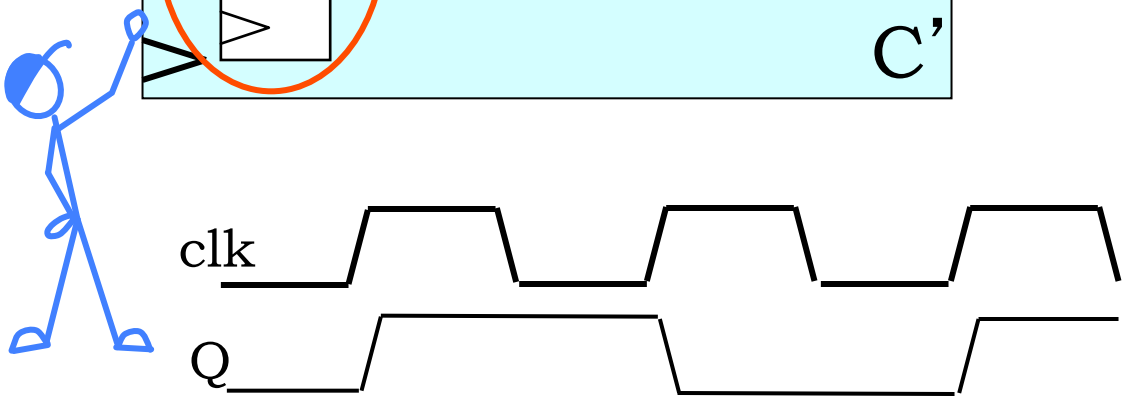
T = 1/8ns
L = 24ns

Circuit Interleaving

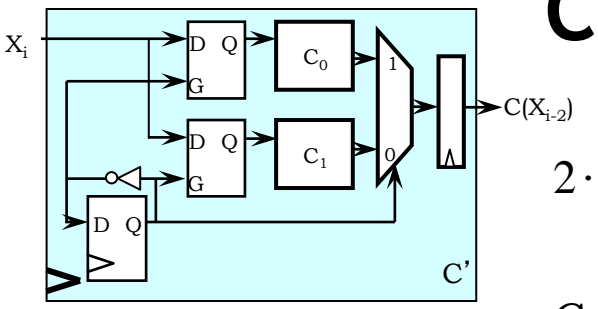
We can simulate a pipelined version of a slow component by replicating the critical element and alternate inputs between the various copies.



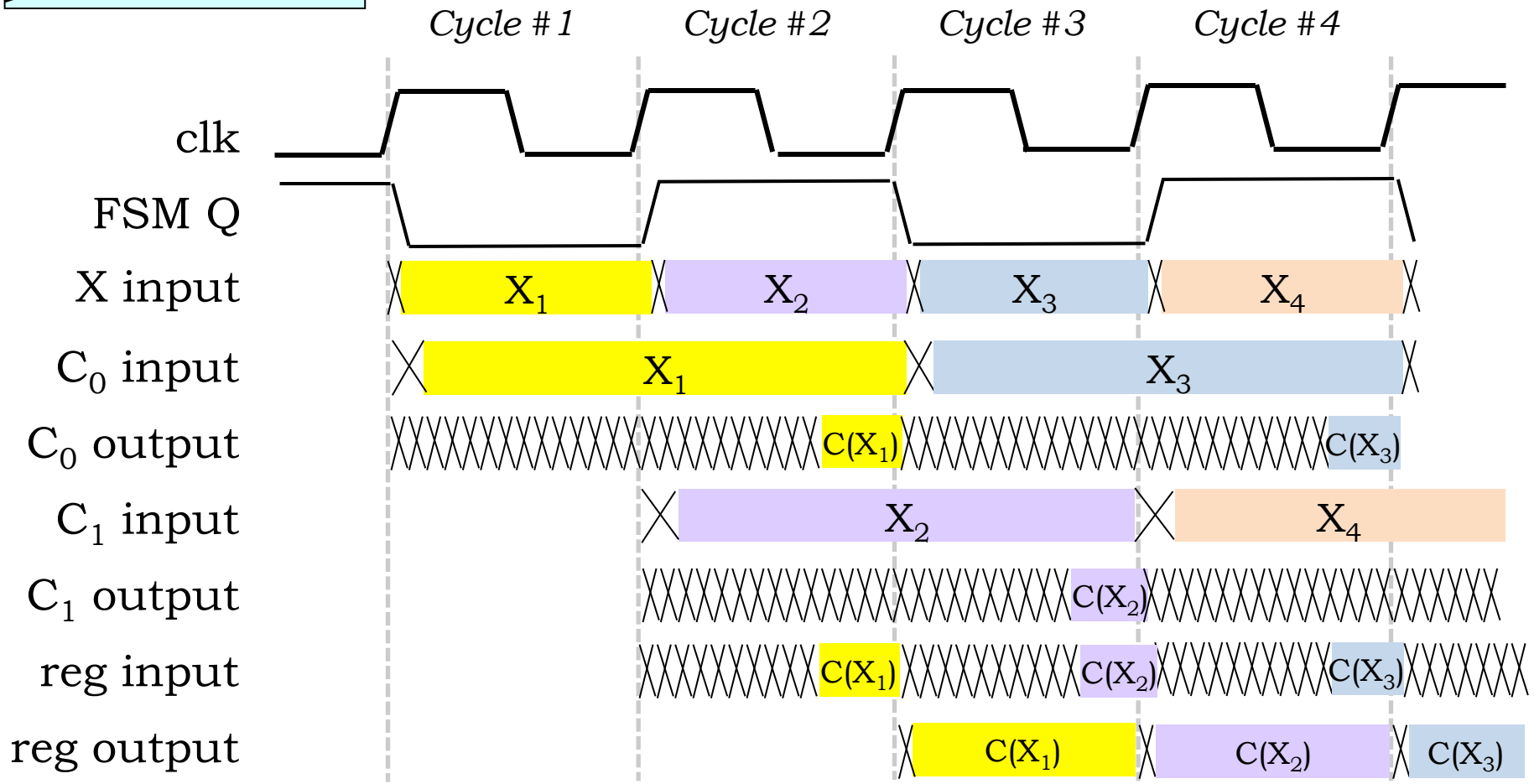
This is a simple 2-state FSM that alternates between 0 and 1 on each clock



Circuit Interleaving



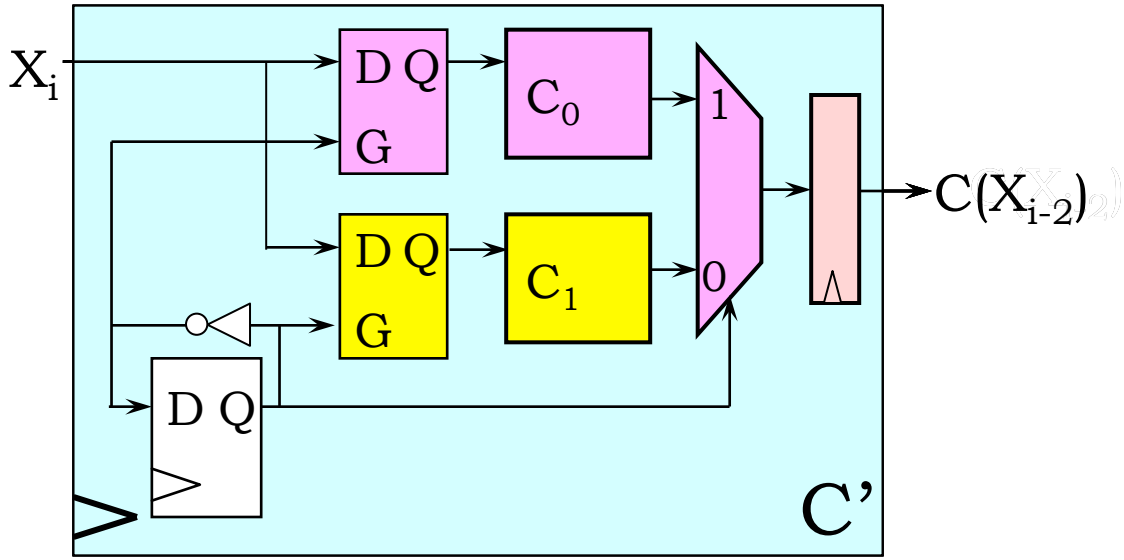
$$2 \cdot t_{CLK} \geq \left(t_{PD,upstreamREG} + t_{PD,LATCH} + t_{PC,C} + t_{PD,MUX} + t_{SETUP,REG} \right)$$



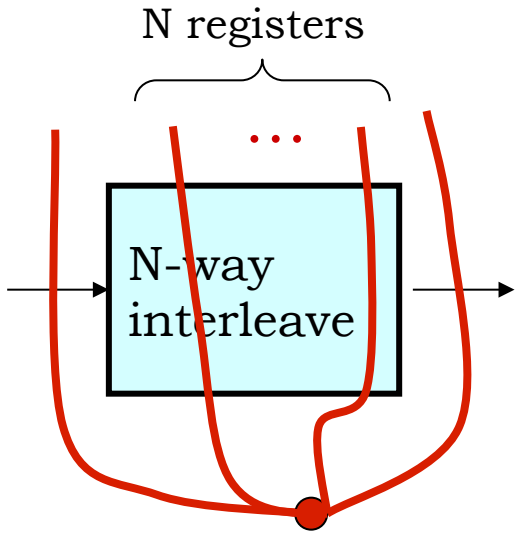
Circuit Interleaving

2-Clock Martinizing
"In by t_i, out by t_{i+2}"

Throughput = 1/clock
Latency = 2 clocks



N-way interleaving
is equivalent to
N pipeline Stages...

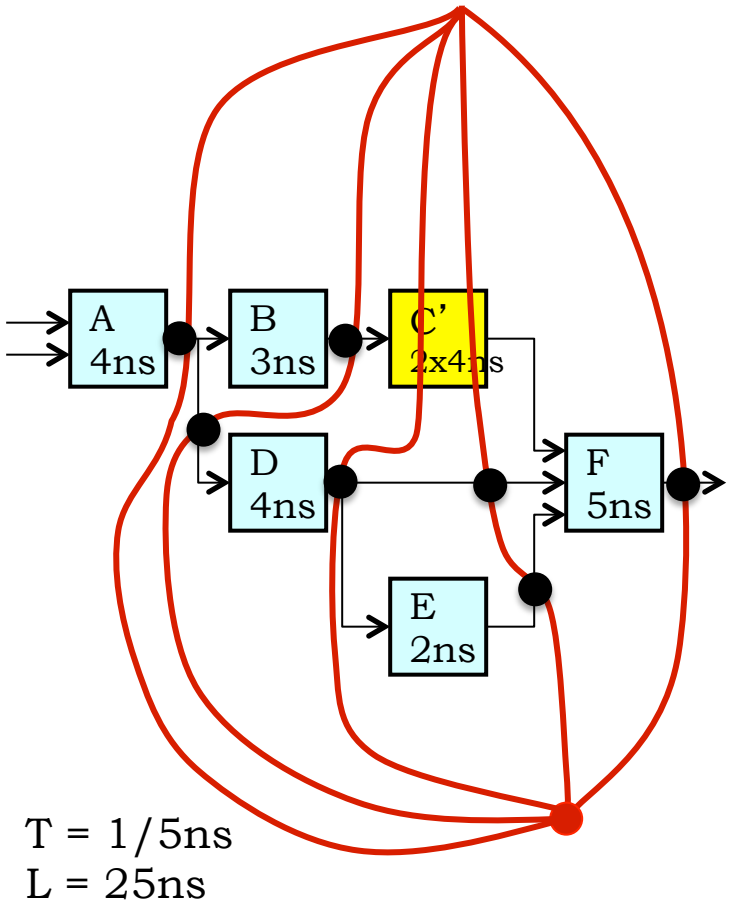


Combine Techniques

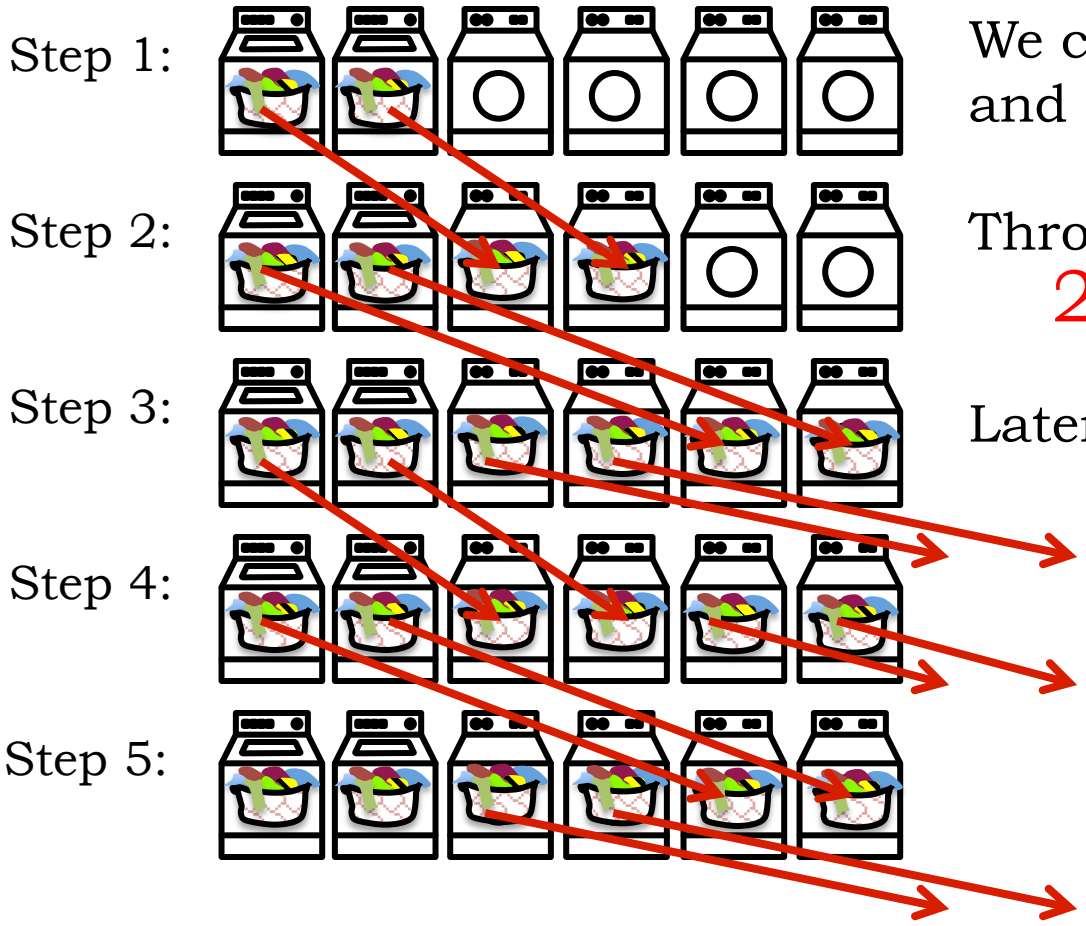
We can combine interleaving and pipelining. Here, C' interleaves two C elements and has an effective tCLK of 4 ns and a latency of 8 ns.

Since C' behaves as a 2-stage pipeline, two of our pipelining contours must pass through the C' component.

By combining interleaving with pipelining we move the bottleneck from the C element to the F element.



And Add A Little Parallelism…



1 step = 30 minutes

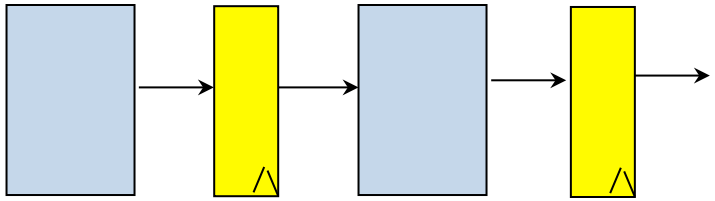
We can combine interleaving and pipelining with parallelism.

Throughput = 2/30 = 1/15 load/min

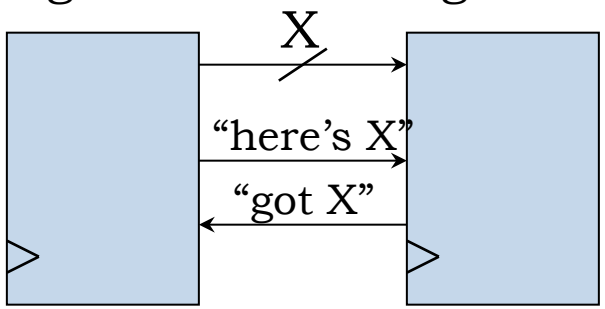
Latency = 90 min

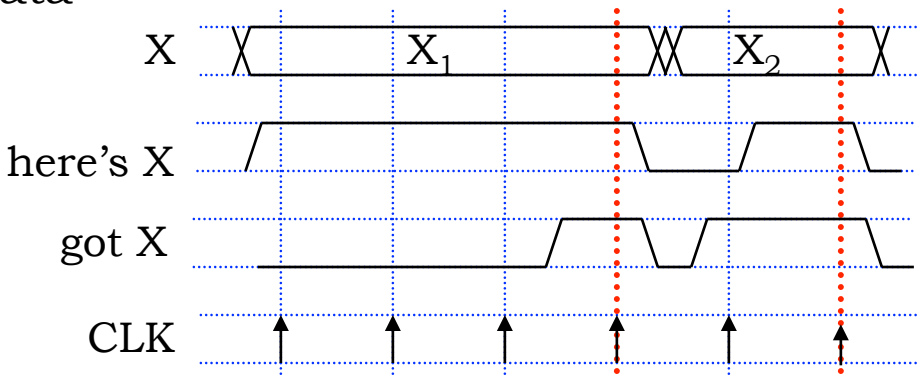
Control Structure Alternatives

Synchronous, globally-timed:

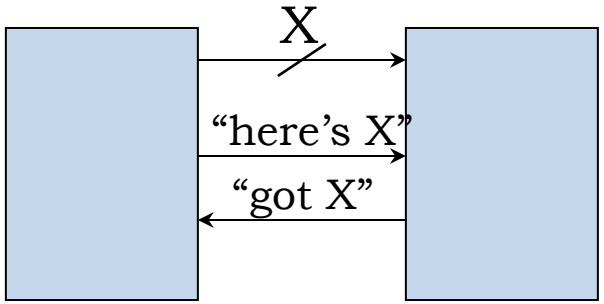


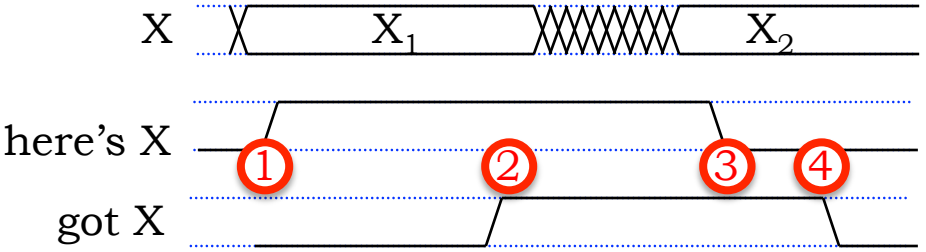
Synchronous, locally-timed:
Local FSMs control flow of data using "handshake" signals



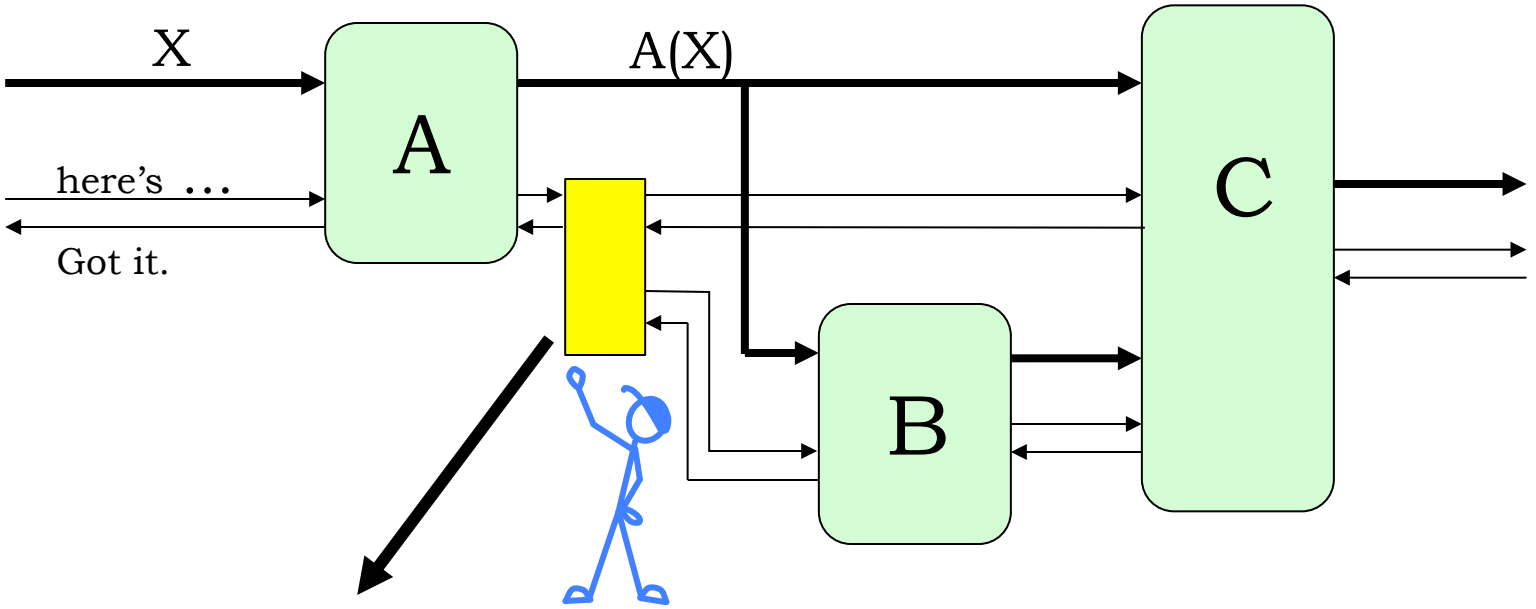


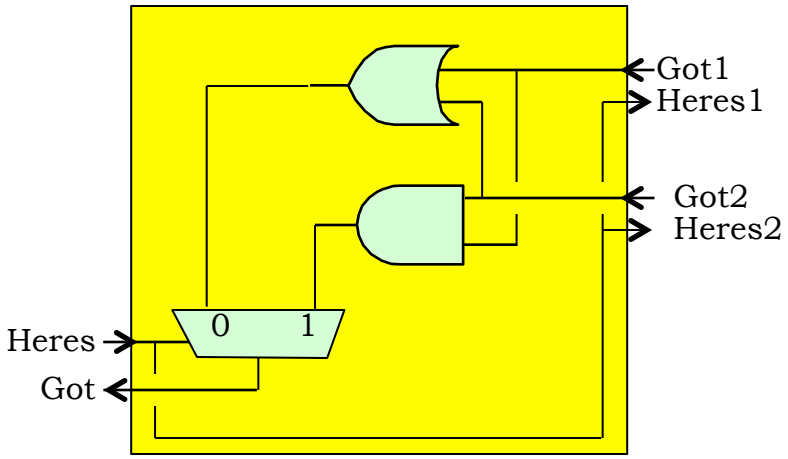
Asynchronous, locally-timed system using *transition signaling:*

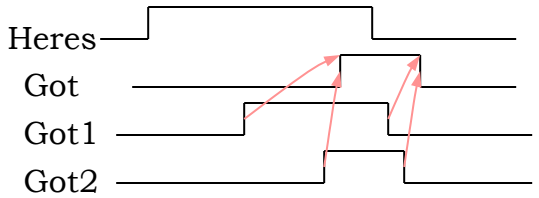


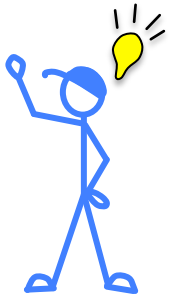


Self-timed Example



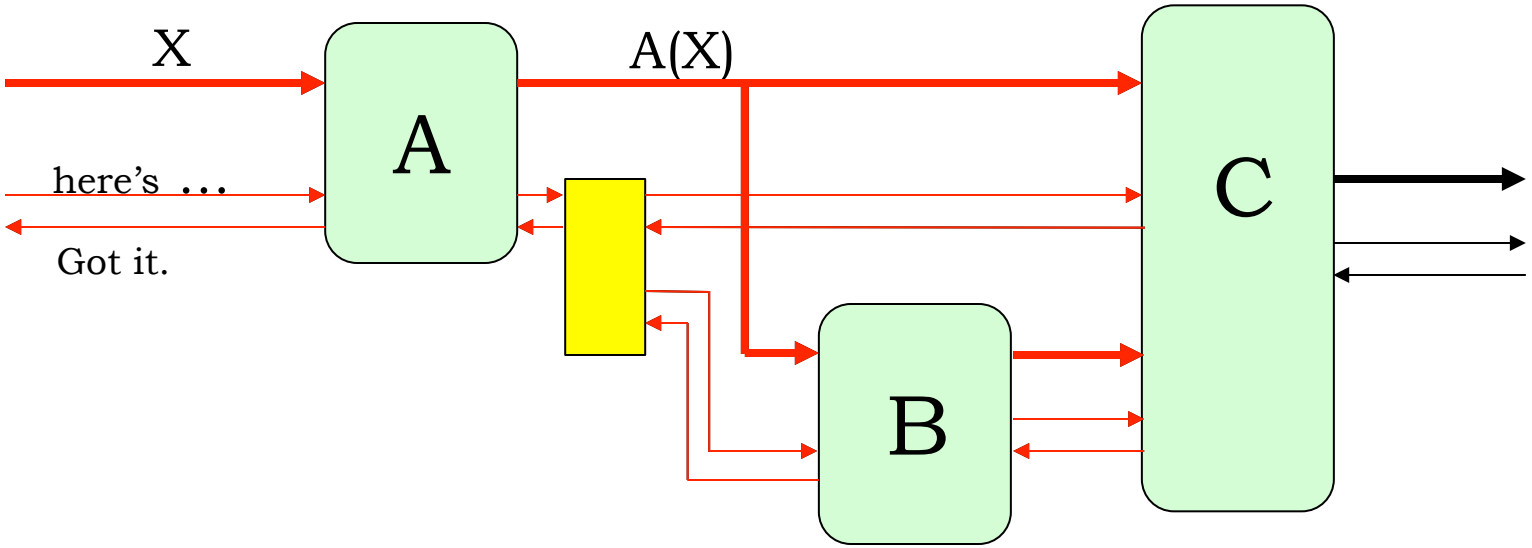






I get it! A sees "got it" as 1 when both B and C have asserted "got it". And then "got it" returns to 0 when both B and C have deasserted "got it".

Self-timed Example



Elegant, timing-independent design:

- Each component specifies its own time constraints
- Local adaptation to special cases (eg, multiplication by 0)
- Module performance improvements automatically exploited
- Can be made asynchronous (no clock at all!) or synchronous

Control Structure Taxonomy

Easy to design but fixed-sized interval can be wasteful (no data-dependencies in timing)

Large systems lead to very complicated timing generators… just say no!

	Synchronous	*Asynchronous*
Globally Timed	Centralized clocked FSM generates all control signals.	Central control unit tailors current time slice to current tasks.
Locally Timed	Start and Finish signals generated by each major subsystem, synchronously with global clock.	Each subsystem takes asynchronous Start, generates asynchronous Finish (perhaps using local clock).

The best way to build large systems that have independently-timed components.

The "next big idea" for the last several decades: a lot of design work to do in general, but extra work is worth it in special cases

Summary

Latency (L) = time it takes for given input to arrive at output

Throughput (T) = rate at which new outputs appear

For combinational circuits: L = t_{PD} of circuit, T = 1/L

For K-pipelines (K > 0):

- always have register on output(s)
- K registers on <u>every</u> path from input to output
- Inputs available shortly after clock i, outputs available shortly after clock (i+K)
- $t_{CLK} = t_{PD,REG} + t_{PD}$ of slowest pipeline stage + t_{SETUP}
- $T = 1/t_{CLK}$
  - more throughput ⇒ split slowest pipeline stage(s)
  - use replication/interleaving if no further splits possible
- $L = K*t_{CLK} = K / T$
  - pipelined latency ≥ combinational latency