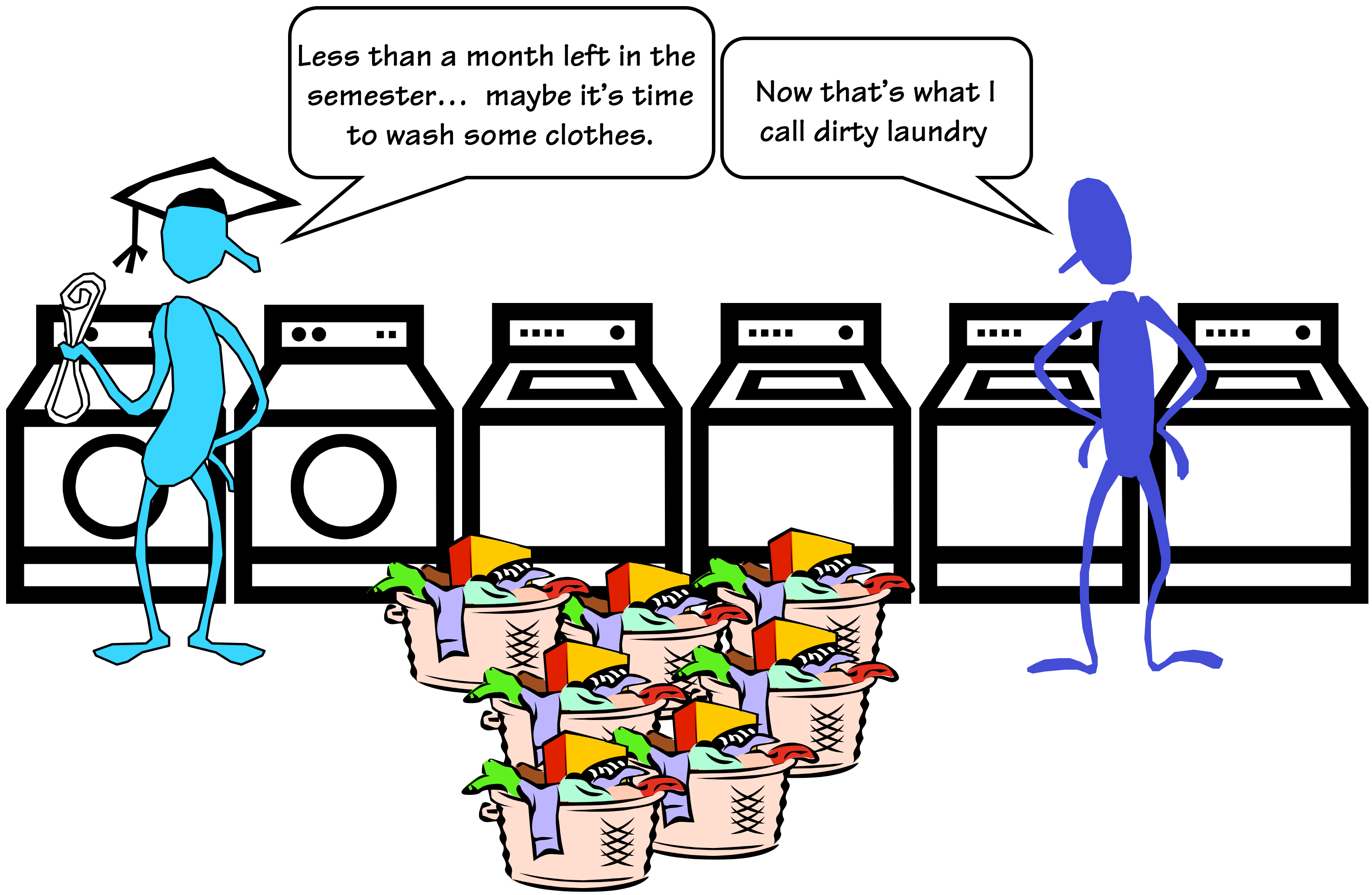
Pipelining


Less than a month left in the semester… maybe it's time to wash some clothes.
Now that's what I call dirty laundry

The Goal of Pipelining

- Recall our measure of processor performance

Millions of Instructions per Second → MIPS

Frequency in MHz → clocks/sec

$$MIPS = \frac{clocks/\sec}{AVE(clocks/instruction)}$$

CPI (Average Clocks Per Instruction) → AVE(clocks/instruction)

- How can we crank up the clock rate?

Forget 411… Let's Solve a "Relevant Problem"

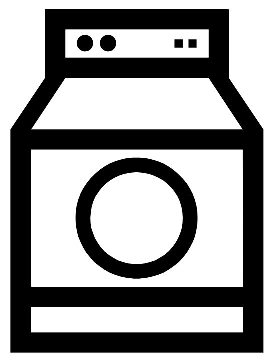
INPUT:
dirty laundry

Device: Washer

Function: Fill, Agitate, Spin

Washer_{PD} = 30 mins

OUTPUT:
4 more weeks



Device: Dryer

Function: Heat, Spin

Dryer_{PD} = 60 mins

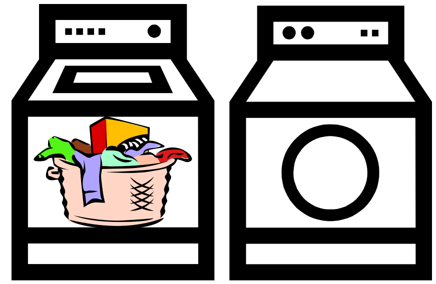
One Load at a Time

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

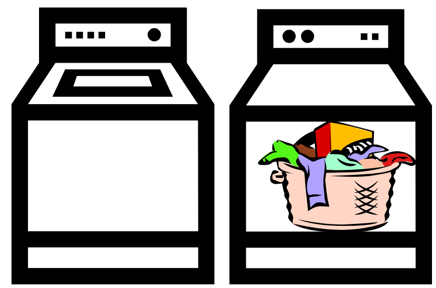
The fact is, doing laundry one load at a time is not smart.

(Sorry Mom, but you were wrong about this one!)

Step 1:



Step 2:



$$\text{Total} = \text{Washer}_{PD} + \text{Dryer}_{PD}$$

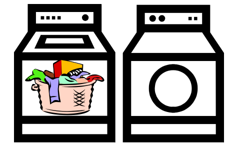
$$= \underline{\quad 90 \quad} \text{ mins}$$

Doing N Loads of Laundry

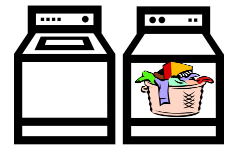
Here's how they do laundry at Duke, the "combinational" way.

(Actually, this is just an urban legend. No one at Duke actually does laundry. The butler's all arrive on Wednesday morning, pick up the dirty laundry and return it all pressed and starched by dinner)

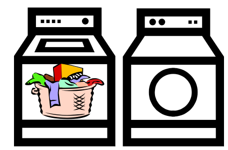
Step 1:



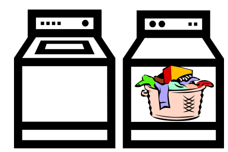
Step 2:



Step 3:



Step 4:



...

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

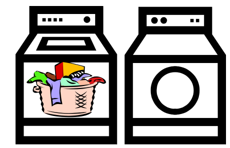
= ____N*90____ mins

Doing N Loads… the UNC way

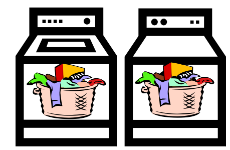
UNC students "pipeline" the laundry process.

That's why we wait!

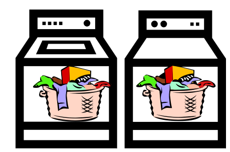
Step 1:



Step 2:



Step 3:



…

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.

Total $= N * Max(Washer_{PD}, Dryer_{PD})$

$=$ N*60 mins

Recall Our Performance Measures

Latency:

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

(Duke Laundry = ___90___ mins)
(UNC Laundry = ___120___ mins)

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

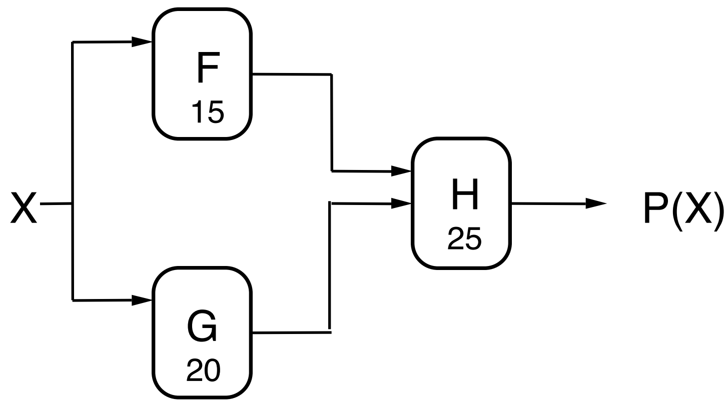
Throughput:

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

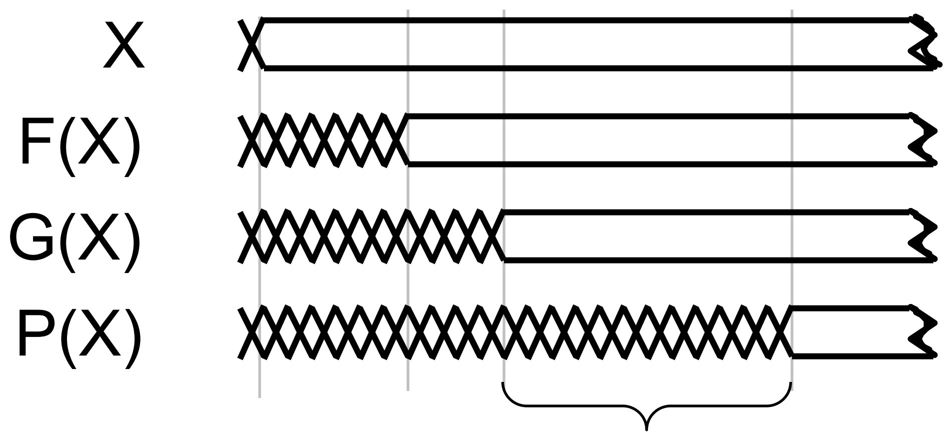
(Duke Laundry = ___1/90___ outputs/min)
(UNC Laundry = ___1/60___ outputs/min)

Even though we increase latency, it takes less time per load

Okay, Back to Circuits…



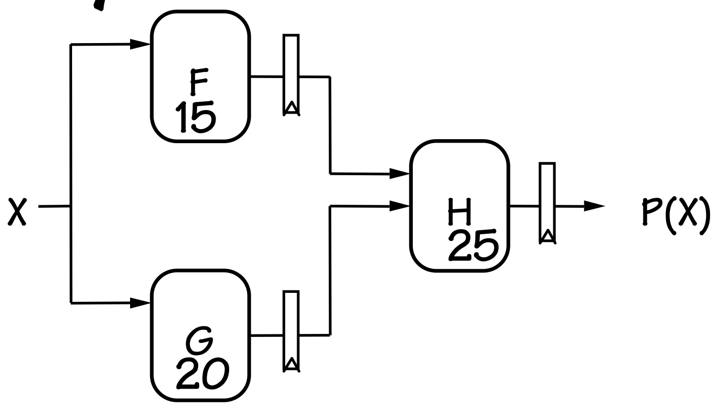
For combinational logic:
latency = t_{PD},
throughput = $1/t_{PD}$.
We can't get the answer faster, but are we making effective use of our hardware at all times?



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

Pipelined Circuits

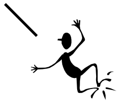
use registers to hold H's input stable!



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.

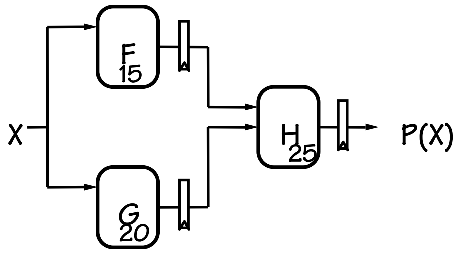
Suppose F, G, H have propagation delays of 15, 20, 25 ns and we are using ideal zero-delay registers ($t_s = 0$, $t_{pd} = 0$):

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



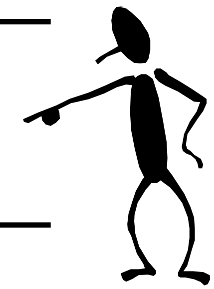
Pipelining uses registers to improve the throughput of combinational circuits

Pipeline Diagrams



Clock cycle →

This is an example of parallelism. At any instant we are busy computing 2 results.



Pipeline stages ↓

	i	i+1	i+2	i+3	
Input	X_i	X_{i+1}	X_{i+2}	X_{i+3}	…
F Reg G Reg		$F(X_i)$ $G(X_i)$	$F(X_{i+1})$ $G(X_{i+1})$	$F(X_{i+2})$ $G(X_{i+2})$	…
H Reg			$H(X_i)$	$H(X_{i+1})$	$H(X_{i+2})$

A pipeline diagram is just a depiction of what inputs are being processed during a given clock period. The results associated with a particular set of input data move *diagonally* through the diagram, progressing through one pipeline stage on each clock cycle.

Pipeline Conventions

DEFINITION:

a *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 a 0-stage pipeline.

CONVENTION:

Every pipeline stage, hence every K-Stage pipeline, has a register on its *OUTPUTS* (as opposed to, alternatively, its inputs).

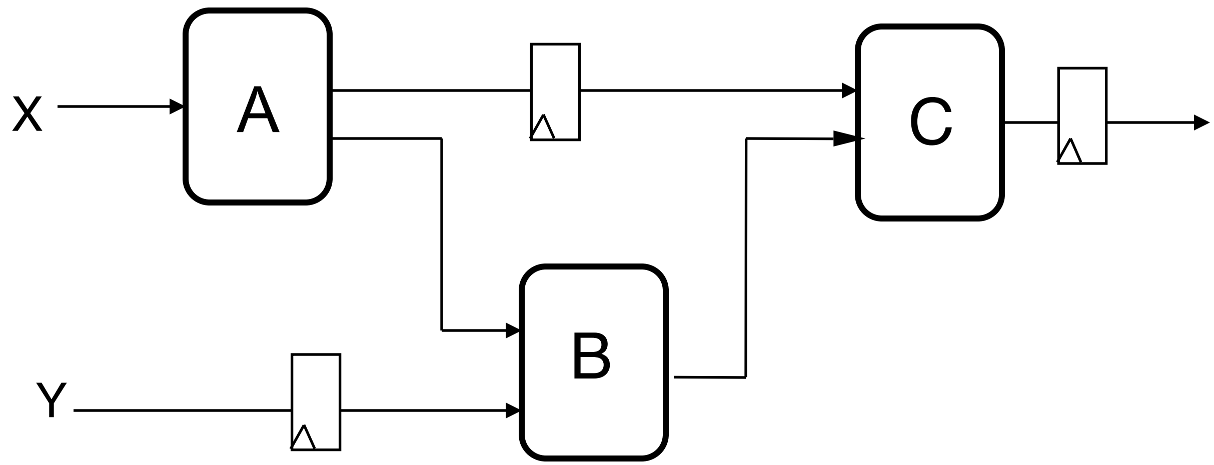
ALWAYS:

The CLOCK common to all registers *must* have a period sufficient to allow for the propagation delays of all combinational paths PLUS (input) register's t_{PD} PLUS (output) register's t_{SETUP}.

> The LATENCY of a K-pipeline is K times the period of the clock common to all registers.
>
> 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 had 2 registers, and some had only 1!

Can this happen on a well-formed K pipeline?

A Pipelining Methodology

Step 1:

Draw a line that crosses every output in the circuit, and select one endpoint as an origin.

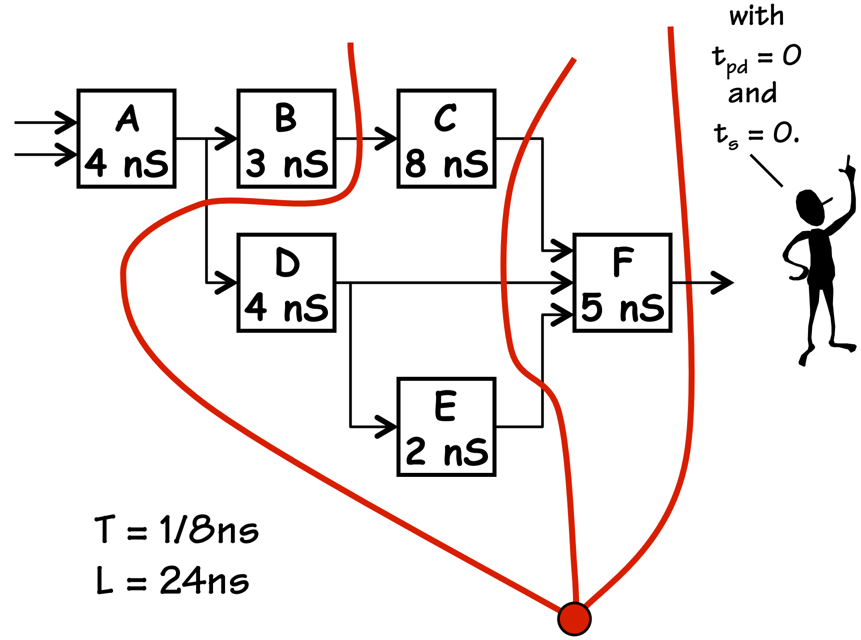
Step 2:

Continue to draw new lines from the origin across various circuit connections such that these new lines partition the inputs from the outputs.

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

STRATEGY:

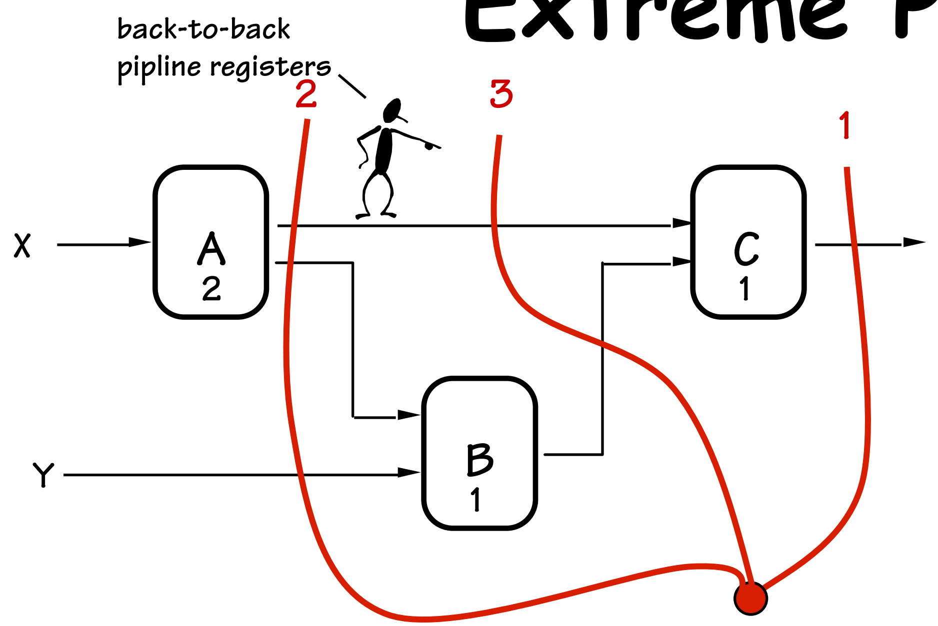
Focus your attention on placing pipelining registers around the slowest circuit elements (BOTTLENECKS).



T = 1/8ns

L = 24ns

Extreme Pipelining



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

OBSERVATIONS:

- 1-pipelines improves neither Latency nor Throughput.
- Throughput is improved by breaking long combinational paths, allowing faster clock.
- Too many stages increase Latency , and don't improve Throughput.
- Back-to-back registers (without logic in-between) are sometimes required to keep a pipeline well-formed.

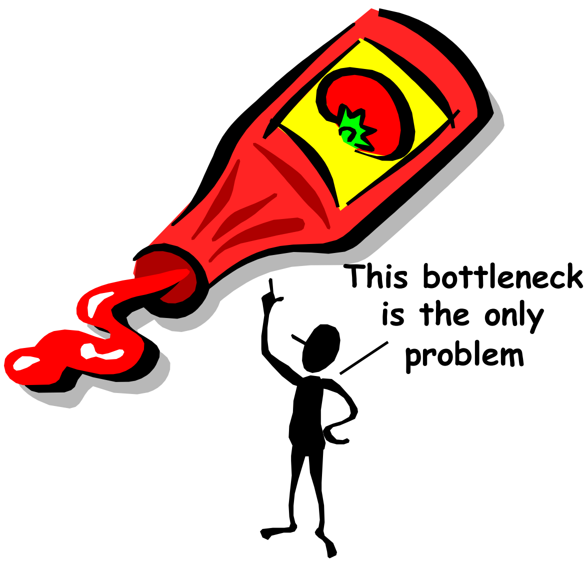
Pipelining Summary

Advantages:

- Higher throughput than combinational system
- Different parts of the logic work on different parts of the problem…

Disadvantages:

- Generally, increases latency
- Only as good as the *weakest* link
  (often called the pipeline's BOTTLENECK)



Isn't there a way around this "weak link" problem?

How do UNC students REALLY do Laundry?

Step 1:

Step 2:

Step 3:

Step 4:

They work around the bottleneck.

First, they find a place with twice as many dryers as washers.

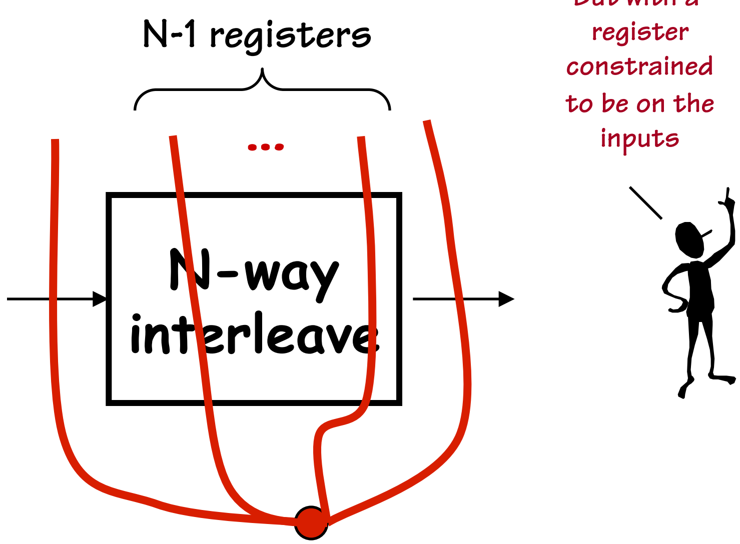
Throughput = 1/30 loads/min

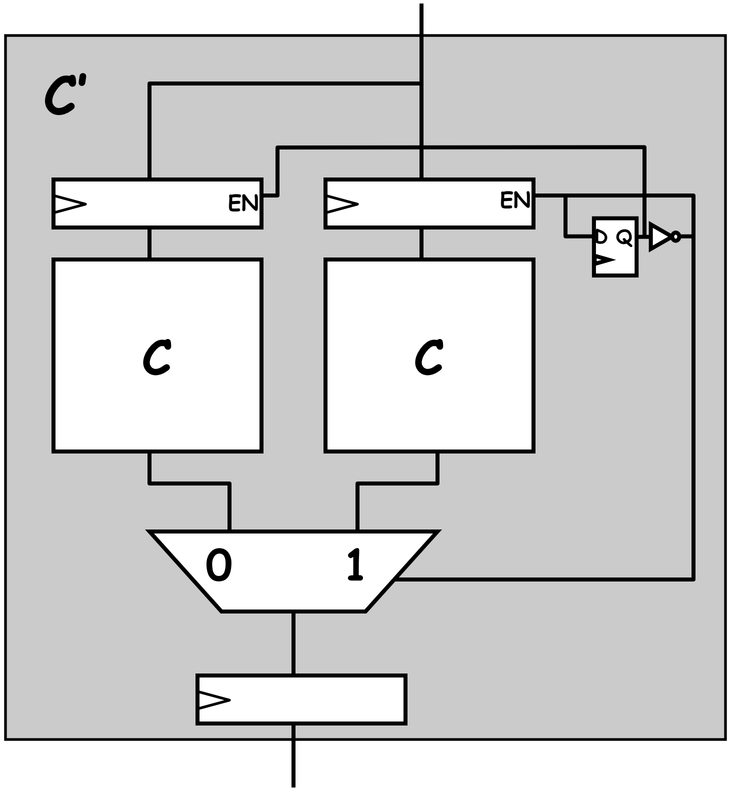
Latency = 90 mins/load

Circuit Interleaving

One way to overcome a pipeline bottleneck is to replicate the critical element as many times as needed and *alternate* inputs between the various copies.

N-way interleaving is equivalent to how many pipeline Stages? N



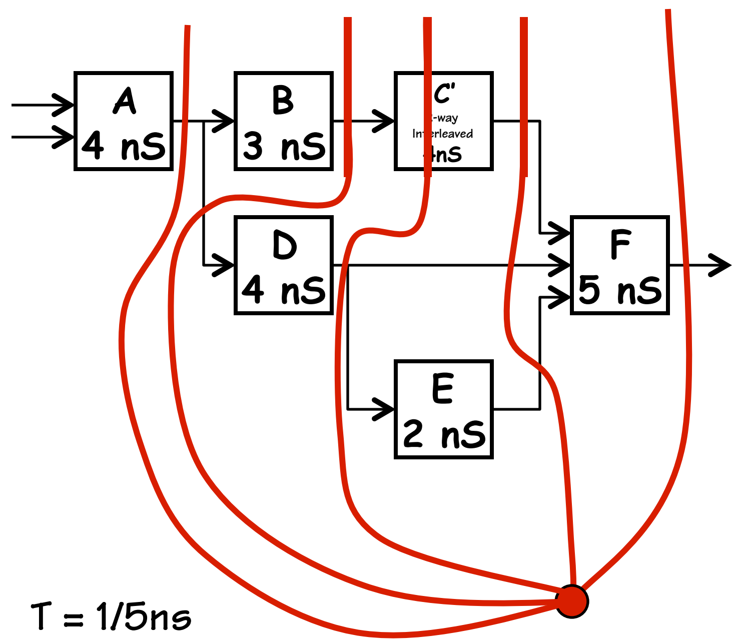


Latency = 2 clocks

Combining Techniques

We can combine interleaving and pipelining. Here, C' interleaves two C elements with a propagation delay of 8 nS. The resulting C' circuit has a throughput of 4 nS, and latency of 8 nS. This can be considered as an extra pipelining stage that passes through the middle of the C' module. One of our separation lines must pass through this pipeline stage.

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



T = 1/5ns
L = 25ns

Better Yet… Parallelism

We can combine interleaving and pipelining with parallelism.

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

Latency = 90 min

Other Control Structure Approaches

Synchronous

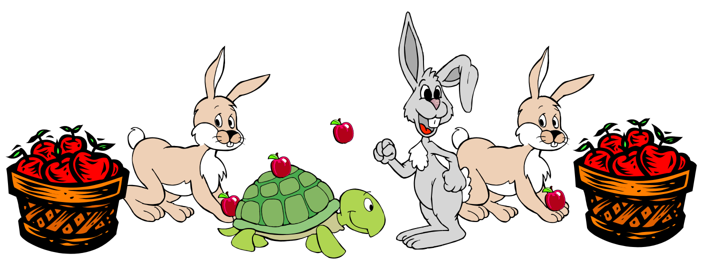
ALL computation “events” occur at active edges of a periodic clock: time is divided into fixed-size discrete intervals.

Globally Timed

Timing dictated by centralized FSM according to a fixed schedule.

RIGID

Laid Back



Asynchronous

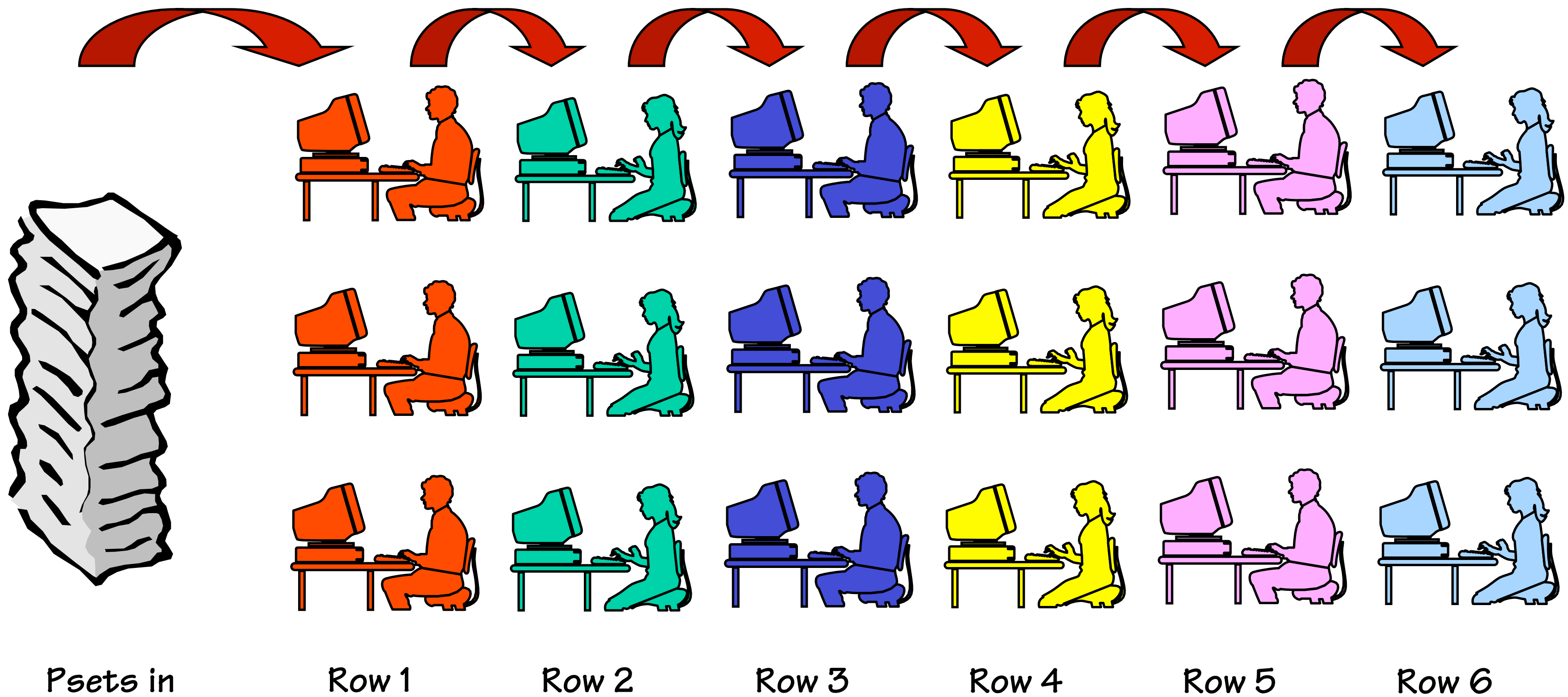
Events – e.g. the loading of a register -- can happen at at arbitrary times.

Locally Timed

Each module takes a START signal, generates a FINISHED signal. Timing is dynamic, data dependent.

"Classroom Computer"

There are lots of problem sets to grade, each with six problems. Students in Row 1 grade Problem 1 and then hand it back to Row 2 for grading Problem 2, and so on… Assuming we want to pipeline the grading, how do we time the passing of papers between rows?



Controls for "Classroom Computer"

	Synchronous	Asynchronous
Globally Timed	Teacher picks time interval long enough for worst-case student to grade toughest problem. Everyone passes psets at end of interval.	Teacher picks variable time interval long enough for *current students* to grade *current mix* of problems. Everyone passes psets at end of interval.
Locally Timed	Students raise hands when they finish grading current problem. Teacher checks every 10 secs, when all hands are raised, everyone passes psets to the row behind. Variant: students can pass when all students in a "column" have hands raised.	Students grade current problem, wait for student in next row to be free, and then pass the pset back.

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 a global clock.	Each subsystem takes asynchronous Start, generates asynchronous Finish (perhaps using local clock).

The best way to build large systems that have independent 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 the results from a given input to arrive at outputs
- Throughput (T) = rate at each 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
  - $T = 1/(t_{PD,REG} + t_{PD}$ of slowest pipeline stage $+ t_{SETUP})$
    - more throughput → split slowest pipeline stage(s)
    - use replication/interleaving if no further splits possible
  - $L = K / T$
    - pipelined latency ≥ combinational latency

Next Time: Let's Pipeline miniMIPS!