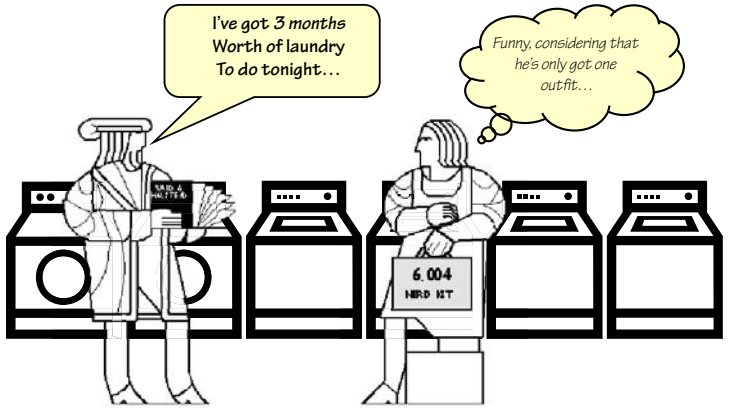
6.004 Computation Structures
Spring 2009

Pipelining

what Seymour Cray taught the laundry industry



Due Thursday: Lab #3

Forget circuits… lets 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

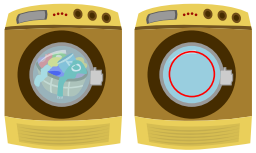
Figure by MIT OpenCourseware.

One load at a time

Everyone knows that the real reason that MIT 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 one load at a time is not smart.

Step 1:



Step 2:

Figure by MIT OpenCourseware.

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

= 90 mins

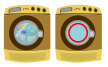
Doing N loads of laundry

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)

Image by MIT OpenCourseWare.

Step 1:



Step 2:

Step 3:

Step 4:
…

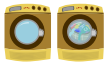


Figure by MIT OpenCourseware.

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

= N*90 mins

Doing N Loads… the MIT way

MIT students "pipeline" the laundry process.

That's why we wait!

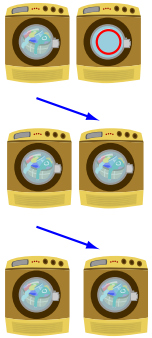


Figure by MIT OpenCourseware.

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

Performance Measures

Latency:

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

(Harvard Laundry = 90 mins)
(MIT Laundry = 120 mins)

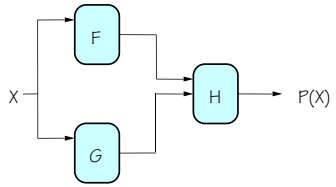
← Assuming that the wash is started as soon as possible and waits (wet) in the washer until dryer is available.

Throughput:

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

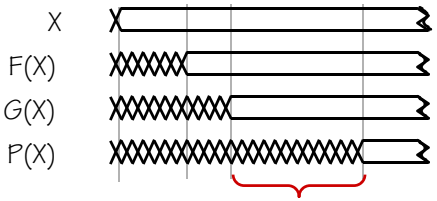
(Harvard Laundry = 1/90 outputs/min)
(MIT Laundry = 1/60 outputs/min)

Okay, back to circuits…



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

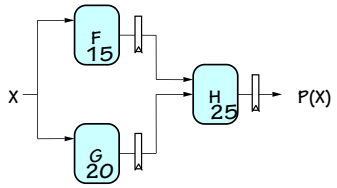
We can't get the answer faster, but are we making effective use of our hardware at all times?



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

Pipelined Circuits

use registers to hold H's input stable!

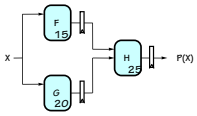


Now F & G can be working on input X_{i+1} while H is performing its computation on X_i. We've created a 2-stage *pipeline*: if we have a valid input X during clock cycle j, P(X) is valid during clock j+2.

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
	worse	better

Pipeline diagrams



Clock cycle →

Pipeline stages ↓

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

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 *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.*

CONVENTION:

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

ALWAYS:

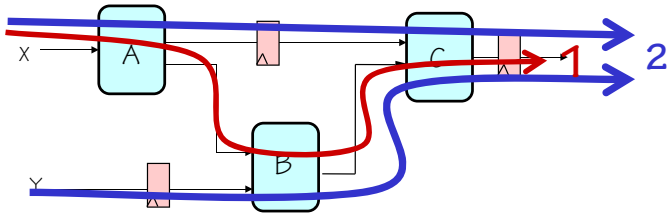
The CLOCK common to all registers must have a period sufficient to cover propagation over combinational paths PLUS (input) register t_{PD} PLUS (output) register t_{SETUP}.

The LATENCY of a K-pipeline is K times the period of the 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 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.

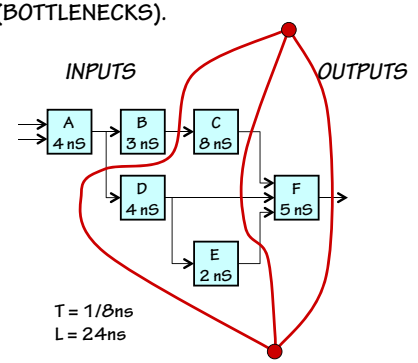
Step 2:

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.*

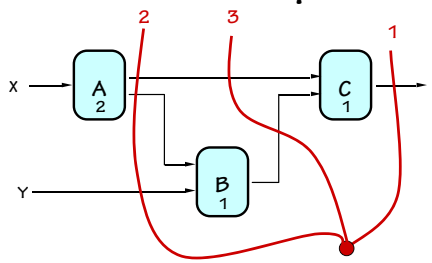
Adding a pipeline register at every point where a separating line crosses a connection will always generate a valid pipeline.

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.

Pipelining Summary

Advantages:

- Allows us to increase thruput, by breaking up long combinational paths and (hence) increasing clock frequency

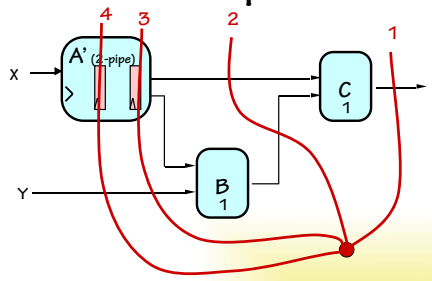
Disadvantages:

- May increase latency...
- Only as good as the weakest link: slowest step constrains system thruput.

This bottleneck is the only problem

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

Pipelined Components



4-stage pipeline, thruput=1

Pipelined systems can be hierarchical:

- Replacing a slow combinational component with a k-pipe version may increase clock frequency
- Must account for new pipeline stages in our plan

but... but...
How can I pipeline
a *clothes dryer*???

How do 6.004 Aces do Laundry?

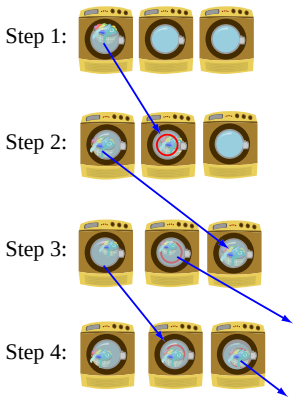


Figure by MIT OpenCourseware.

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

Back to our bottleneck...

Recall our earlier example...

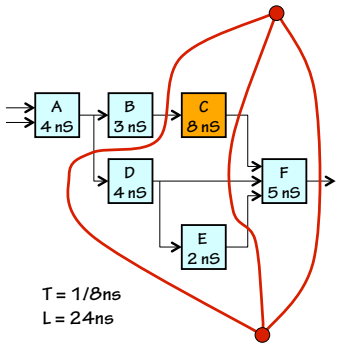
- C – the slowest component – limits clock period to 8 ns.
- HENCE throughput limited to 1/8ns.

We could improve throughput by

- Finding a pipelined version of C;

OR ...

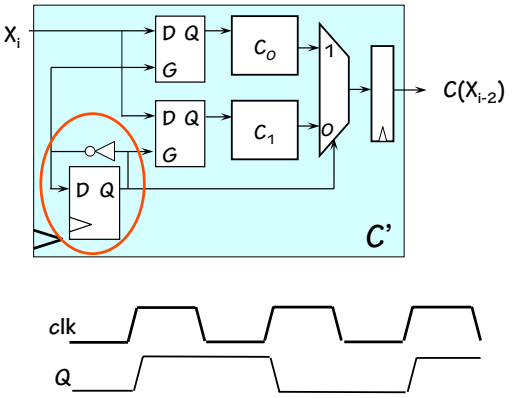
- *interleaving* multiple copies of C!



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

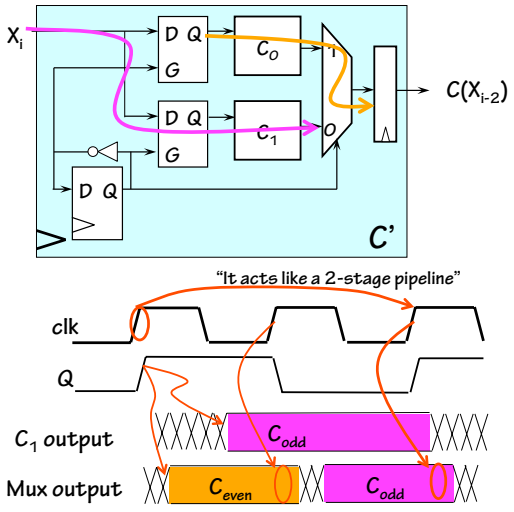


Circuit Interleaving

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

When Q is 1 the lower path is combinational (the latch is open), yet the output of the upper path will be enabled onto the input of the output register ready for the NEXT clock edge.

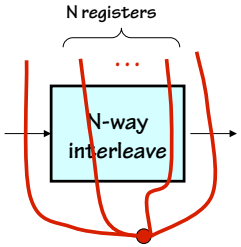
Meanwhile, the other latch maintains the input from the last clock.

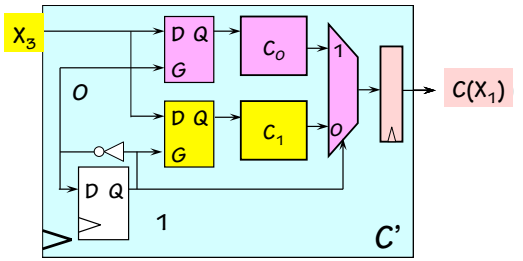


Circuit Interleaving

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

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





Latency = 2 clocks

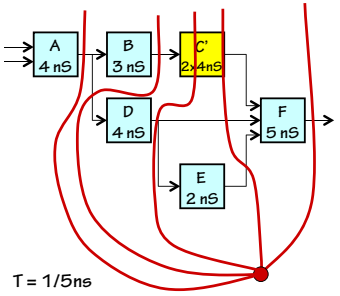
- Clock period 0: X_0 presented at input, propagates thru upper latch, C_0.
- Clock period 1: X_1 presented at input, propagates thru lower latch, C_1. $C_0(X_0)$ propagates to register inputs.
- Clock period 2: X_2 presented at input, propagates thru upper latch, C. $C_0(X_0)$ loaded into register, appears at output.

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 1/4 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

And a little parallelism…

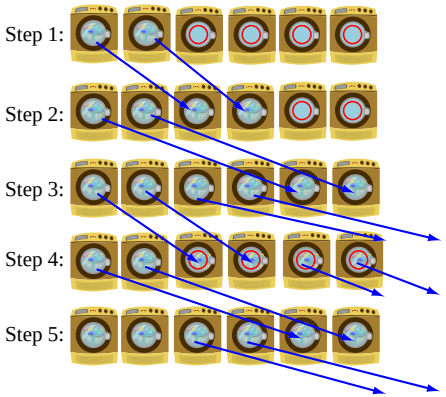


Figure by MIT OpenCourseware.

We can combine interleaving and pipelining with parallelism.

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

Latency = 90 min

Control Structure Approaches

Synchronous

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

RIGID

Globally Timed

Timing dictated by centralized FSM according to a fixed schedule.

Laid Back

Asynchronous

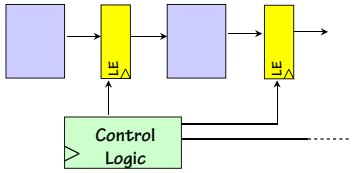
Events -- eg the loading of a register -- can happen at at arbitrary times.

Locally Timed

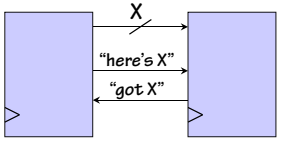
Each module takes a START signal, generates a FINISHED signal. Timing is dynamic, data dependent.

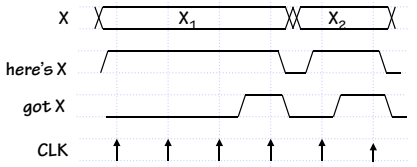
Control Structure Alternatives

Synchronous, globally-timed:
Control signals (e.g., load enables) From FSM controller

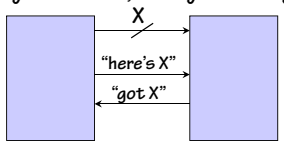


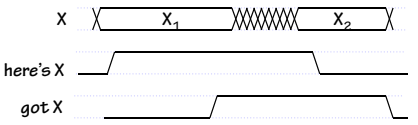
Synchronous, locally-timed:
Local circuitry, "handshake" controls flow of data:





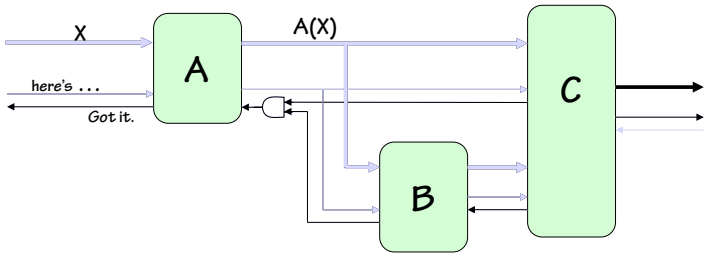
Asynchronous, locally-timed system using *transition signaling:*





Self-timed Example

a glimpse of an asynchronous, locally-time discipline



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 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
  - Inputs available shortly after clock i, outputs available shortly after clock (i+K)
  - 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