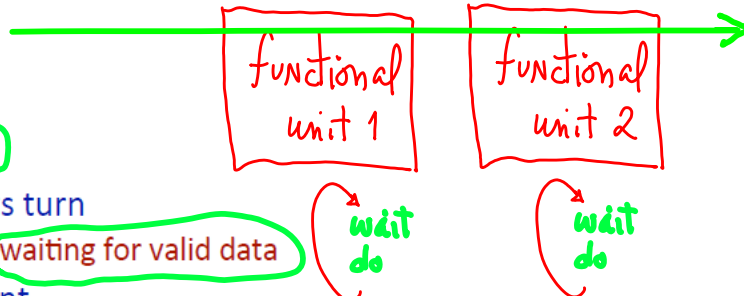


Data signal used



- Problem?
 - Each functional unit used **once per cycle**
 - Most of the time it is sitting waiting for its turn
 - Well it is calculating all the time, but it is **waiting for valid data**
 - There is no parallelism in this arrangement

- Making instructions take **more cycles** can make machine **faster!?!**
 - Each instruction takes roughly the same time
 - While the CPI is much worse, the **clock freq is much higher**
 - **Overlap execution** of multiple instructions at the same time
 - Different instructions will be active at the same time
 - This is called "Pipelining"
 - We will look at a 5 stage pipeline
 - Modern machines (**Core 2**) have order **15 cycles/instruction**

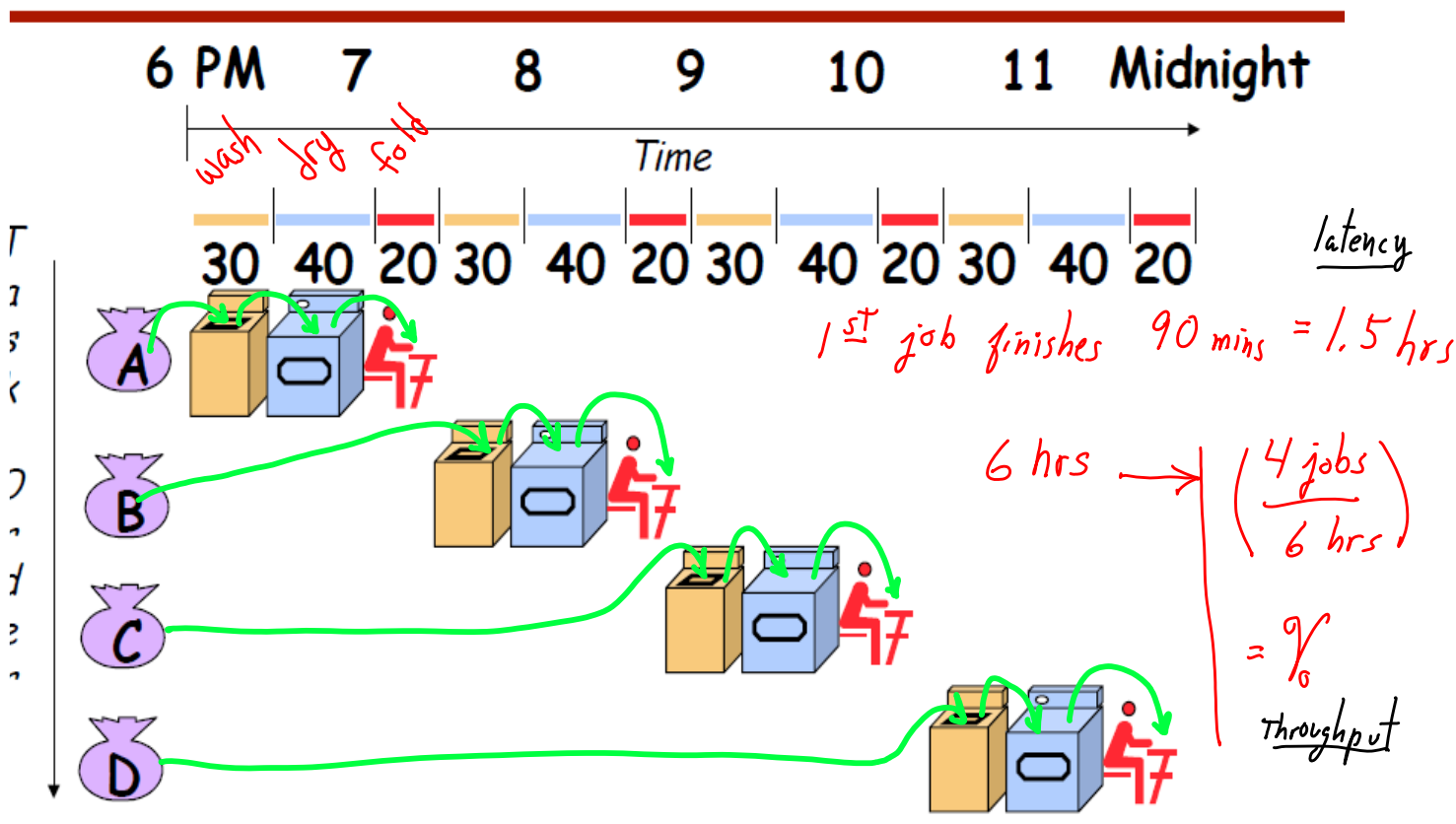
$CPI \uparrow \quad CR \uparrow$

$$Perf = \frac{n}{T} = \frac{n}{n \cdot CPI \cdot (1/CR)}$$

$\rightarrow \uparrow CR / CPI \uparrow$

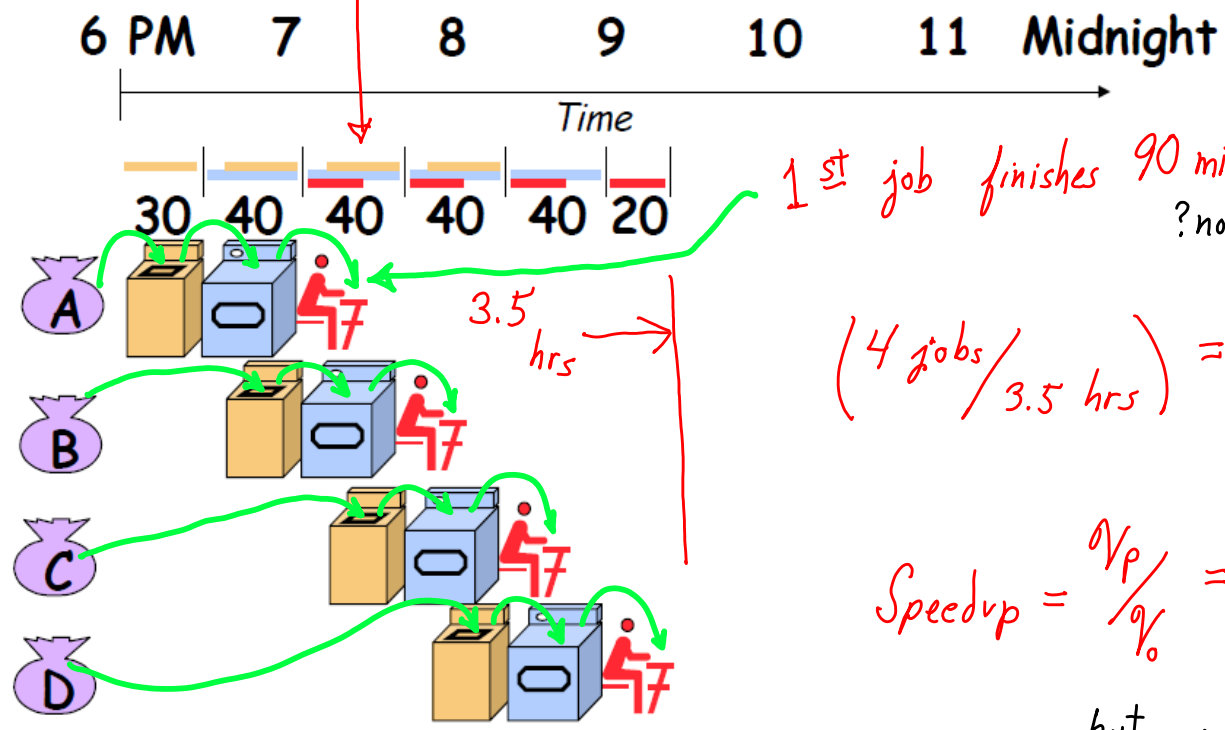
hmm, ... ?

Sequential Laundry



Sequential laundry takes 6 hours for 4 loads

Parallelism = Overlap



1st job finishes 90 mins = 1.5 hrs
? no improvement?

Throughput
 $(4 \text{ jobs} / 3.5 \text{ hrs}) = \rho_p$

Speedup = $\rho_p / \rho_s = \frac{6}{3.5} \approx 2$

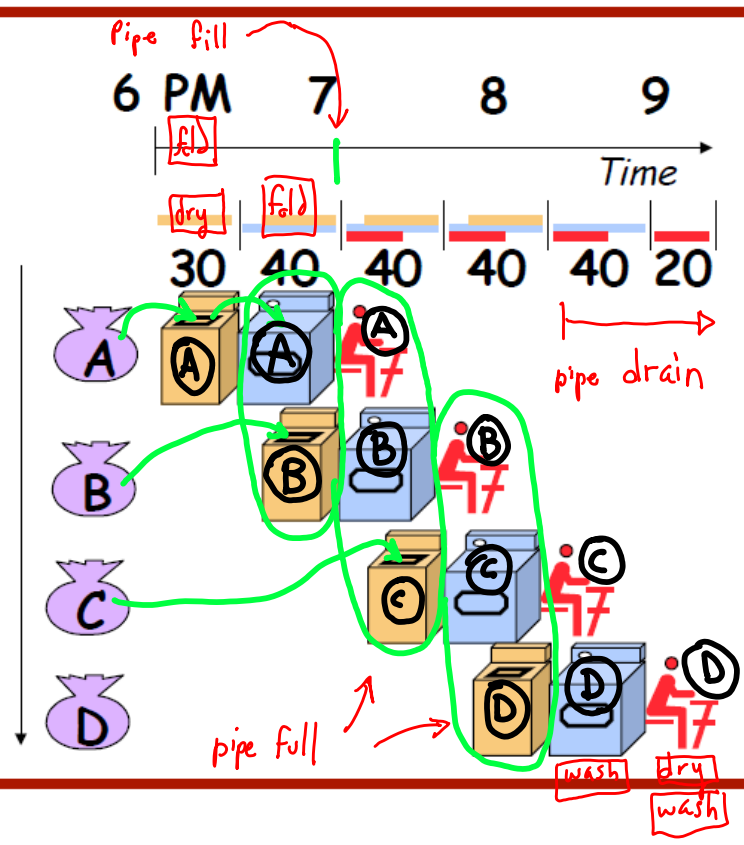
but, no CR ↑

Pipelined laundry takes 3.5 hours for 4 loads

What does Amdahl's Law say?

Pipelining Lessons

throughput ↑

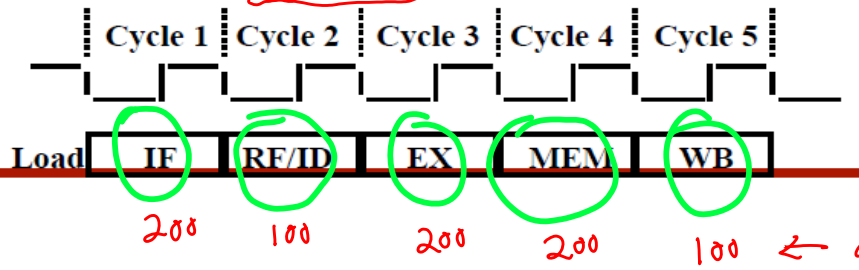


- Pipelining doesn't help latency of single task, it helps throughput of entire workload
- Multiple tasks operating simultaneously
- Potential speedup = Number pipe stages
- Pipeline rate limited by slowest pipeline stage
- Unbalanced lengths of pipe stages reduces speedup
- Time to "fill" pipeline and time to "drain" it reduces speedup

- **IF:** Instruction Fetch
 - Fetch the instruction from memory
 - Increment the PC
- **RF/ID:** Register Fetch and Instruction Decode
 - Fetch base register
- **EX:** Execute
 - Calculate base + sign-extended offset
- **MEM:** Memory
 - Read the data from the data memory
- **WB:** Write back
 - Write the results back to the register file

lw (slowest instr.)

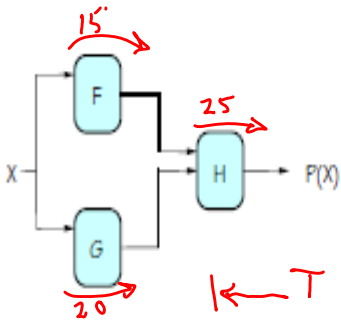
Pipe stages



$$(1/CR) \geq 200 \text{ ps}$$

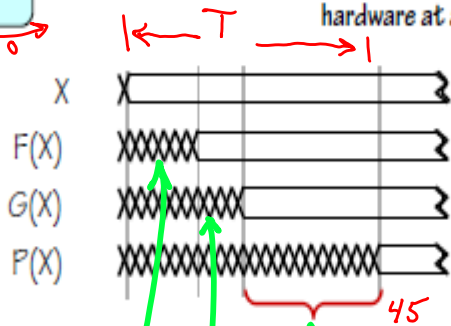
C. Kozvrakis

General pipelines



For combinational logic:
 latency = t_{PD}
 throughput = $1/t_{PD} = \left(\frac{1 \text{ job}}{T \text{ sec}}\right)$

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



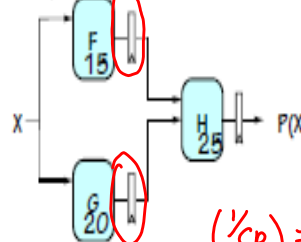
F & G are "idle", just holding their outputs stable while H performs its computation

$$T = 15 + 20 + 25 = 45$$

$$v = \frac{1 \text{ job}}{45}$$

Pipelined Circuits

use registers to hold H's input stable!



Now F & G can be working on input X_{j+1} while H is performing its computation on X_j . 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.

$$(1/CR) = 25$$

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

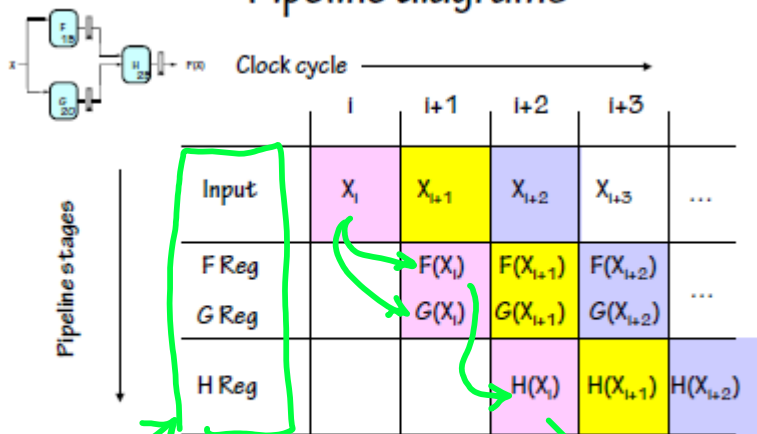
	latency	throughput
unpipelined	45	1/45
2 times @ 25 = 2-stage pipeline	50	1/25
	worse	better

1 job exits per 25

$$T = 25 + 25 = 50$$

$$v_p = \frac{1 \text{ job}}{25}$$

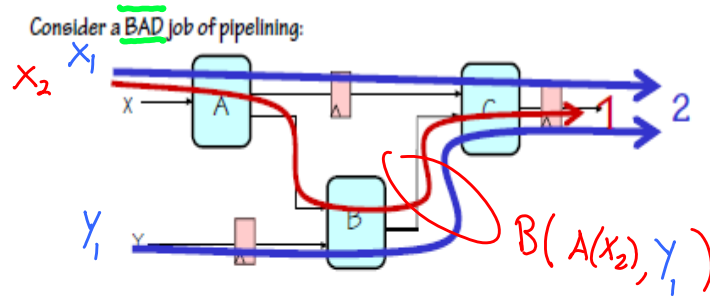
different Pipeline diagrams



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

pipe

job 1 job 2 job 3



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 CANT 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.

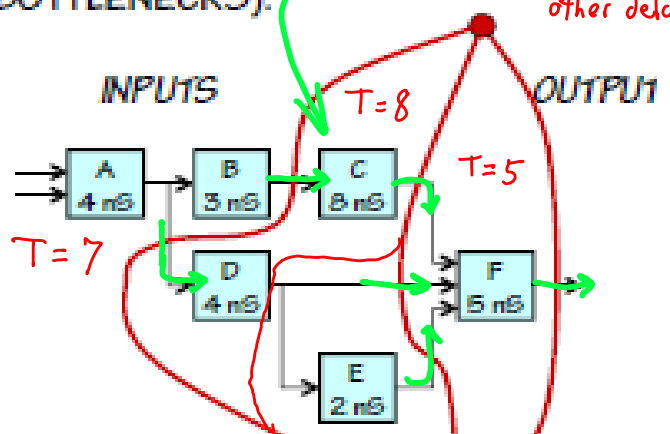
Step 2:

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.

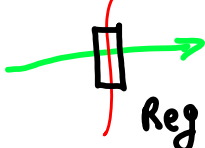
Adding a pipeline register at every point where a separating line crosses a connection will always generate a valid pipeline.

STRATEGY:

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

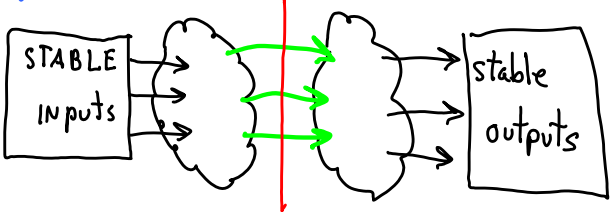


$T = 1/8ns$
 $L = 24ns$

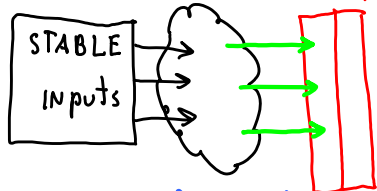


Inductive proof

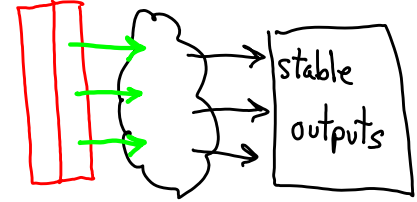
signals in cut-set are correct



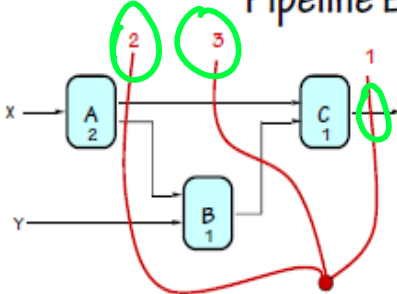
inputs to FF are correct flip flop



flip flop outputs are correct



Pipeline Example

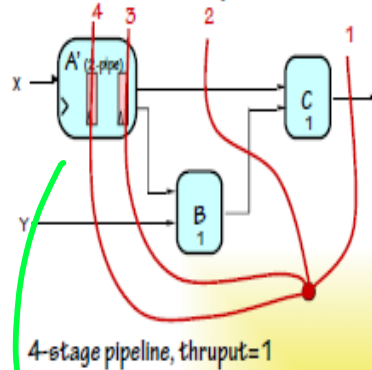


OBSERVATIONS:

- 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.

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

Pipelined Components



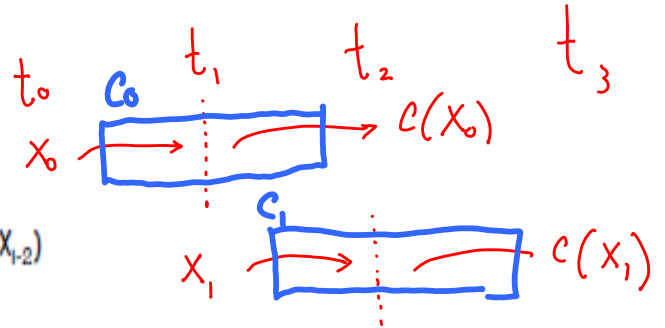
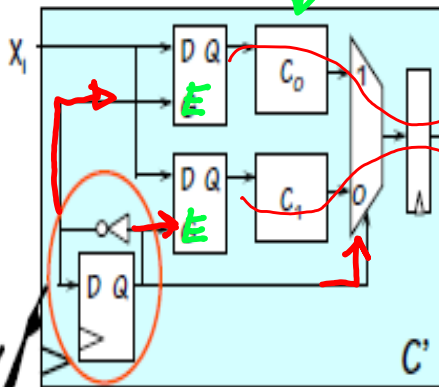
Pipelined systems can be hierarchical:

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

split A in 2 w/ T=1
? How?

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

Wash = 30 Dry = 2 x 30

2 loads And a little parallelism...

parallel pipes w/ interleave

2 pipes
interleaved
dryers



We can combine interleaving and pipelining with parallelism.

Throughput = $\frac{2}{30} = 1/15$ load/min

Latency = 90 min

T = 90

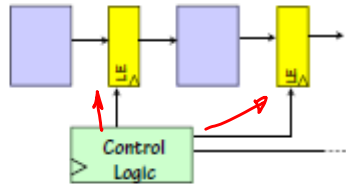
$\eta = \frac{2 \text{ jobs}}{30}$

$S = \frac{2/30}{1/90} = 6$

wash dry₁ dry₂

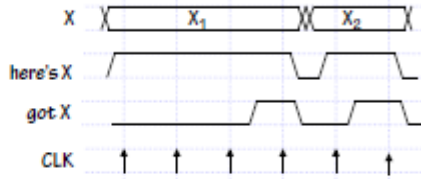
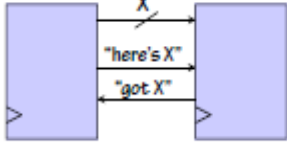
Control Structure Alternatives

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



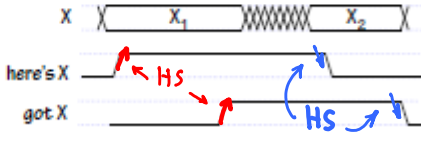
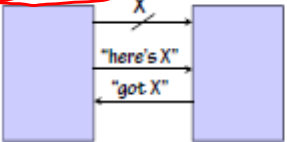
Synchronous, locally-timed:
Local circuitry, "handshake" controls

flow of data:



I/O busses w/ clk
both use same clock

Asynchronous, locally-timed system using transition signaling:

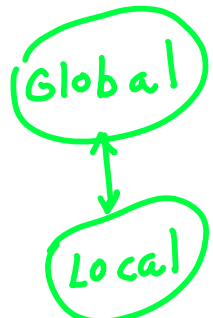
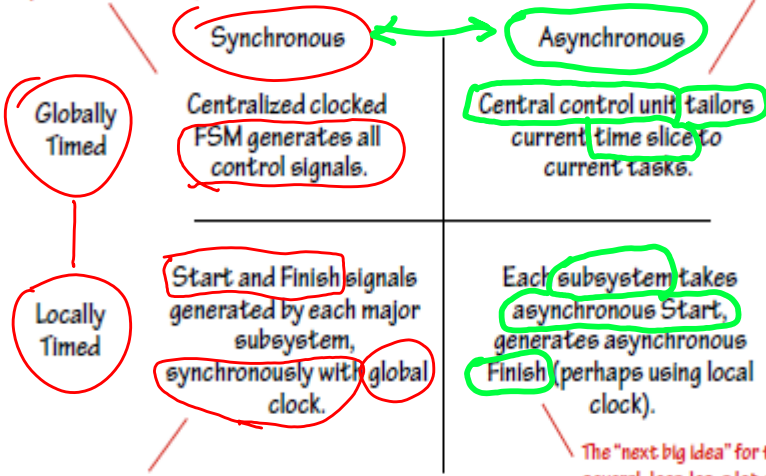


independent clocks (or no clocks, self-timed)
slow I/O device w/ HS protocols (fast Logic)

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!

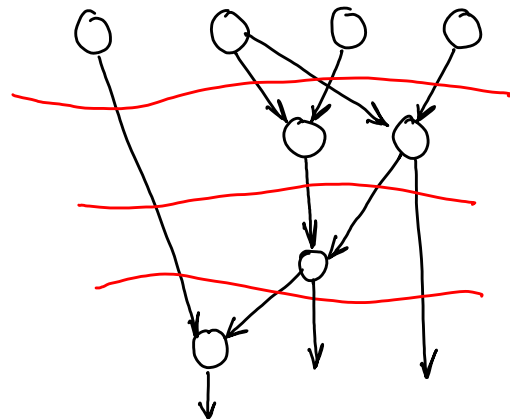


The best way to build large systems that have independently-timed components.

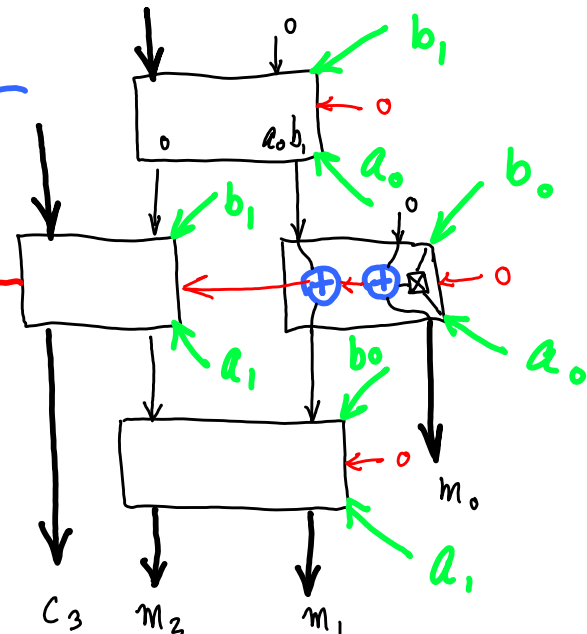
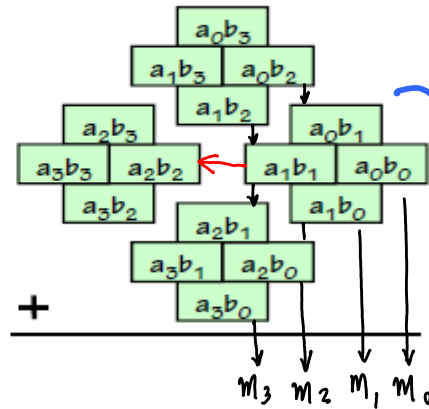
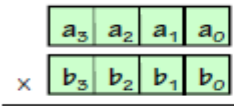
The "next big idea" for the last several decades: a lot of design work to do in general, but extra work is worth it in special cases

Data Dependency Graphs and Pipelining

- Multiple jobs
- High throughput
- Is parallelism as high as possible?
- Is timing data dependent?

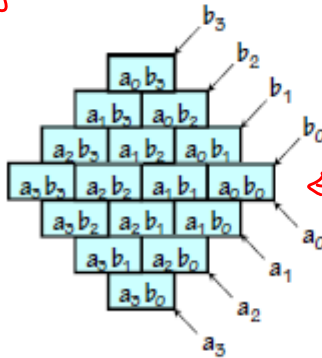


Making 4n-bit multipliers from n-bit ones: 2 "induction steps"



Design of 1-bit multiplier "Brick":

add



Array Layout:

- operand bits bused diagonally
- Carry bits propagate right-to-left
- Sum bits propagate down

partial product

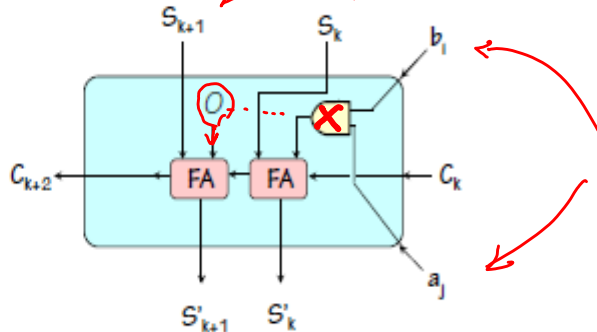
add by columns

1-bit multiply x
2-bit result

Brick design:

- AND gate forms 1x1 product
- 2-bit sum propagates from top to bottom
- Carry propagates to left

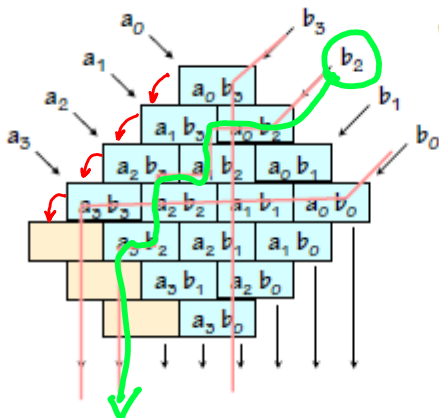
Wastes some gates... but consider (say) optimized 4x4-bit brick!



Column sums

Here's our combinational multiplier:

carry



What's its propagation delay?

Naive (but valid) bound:

- $O(n)$ additions
- $O(n)$ time for each addition
- Hence $O(n^2)$ time required

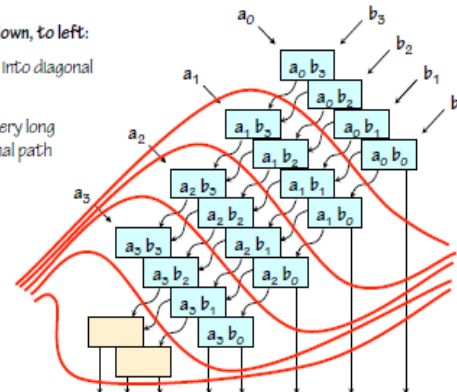
On closer inspection:

- Propagation only toward left, bottom
- Hence longest path bounded by length + width of array:
 $O(n+n) = O(n)$

Breaking $O(n)$ combinational paths

LONG PATHS go down, to left:

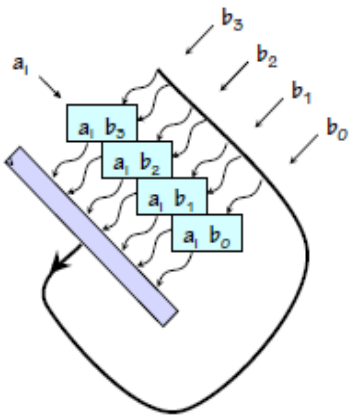
- Break array into diagonal slices
- Segment every long combinational path



GOAL: $\Theta(n)$ stages; $\Theta(1)$ clock period!

cut both ways \Rightarrow systolic

Multiplier Cookbook: Chapter 4



Sequential Multiplier:

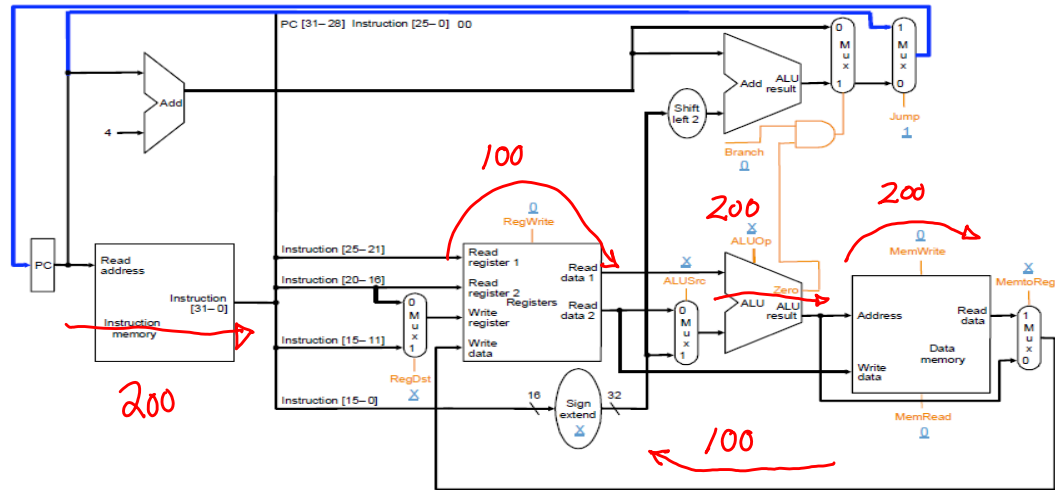
- Re-uses a single n-bit "slice" to emulate each pipeline stage
- a operand entered serially
- Lots of details to be filled in...

Stages: 1
Clock Period: $\Theta(1)$ (constant!)
Hardware cost for n by n bits: $\Theta(n)$
Latency: $\Theta(n)$
Throughput: $\Theta(1/n)$

Can Pipelining Lead to an Arbitrary Short Clock Cycle?

- Min clock cycle = longest combinatorial delay + FF setup + clock skew
- Pipelining reduces the combinatorial delay
 - Less work per pipeline stage
 - Ideally, N stages reduce delay to $1/N$
 - Best you can achieve is Clock cycle \rightarrow FF setup + clock skew
 - Diminishing returns from ever longer pipelines...
- Imbalance between stages also reduces benefits from subdividing
- Even if you could continuously improve clock frequency
 - Power consumption \propto Frequency

- 1-cycle MIPS processor
- Harvard Architecture (two memories)
- clocking PC initiates cycle



Single Cycle Processor Performance

- Functional unit delay
 - Memory: 200ps
 - ALU and adders: 200ps
 - Register file: 100 ps

$ps = 10^{-12} sec$

Instruction Class	Instruction memory	Register read	ALU operation	Data memory	Register write	Total
R-type	200	100	200		100	600
load	200	100	200	200	100	800
store	200	100	200	200		700
branch	200	100	200			500
jump	200					200

max delay = T_{clock}
 $\frac{1}{T_{clock}} = \frac{1}{0.8 ns} = 1.25 GHz$

- CPU clock cycle = 800 ps = 0.8ns (1.25GHz)

what if we let clock trigger delay by opcode?

- Instruction Mix
 - 45% ALU
 - 25% loads
 - 10% stores
 - 15% branches
 - 5% jumps

Instruction Class	Instruction memory	Register read	ALU operation	Data memory	Register write	Total
R-type	200	100	200		100	600
load	200	100	200	200	100	800
store	200	100	200	200		700
branch	200	100	200			500
jump	200					200

ps → 0.6 ns
 0.8
 0.7
 0.5
 0.2

CPU clock cycle = $0.6 \times 45\% + 0.8 \times 25\% + 0.7 \times 10\% + 0.5 \times 15\% + 0.2 \times 5\%$
 = 0.625 ns (1.6GHz)

⇒ 1.6 GHz

what is speedup? $S_{new-old} =$

Pipelining Load *lw*

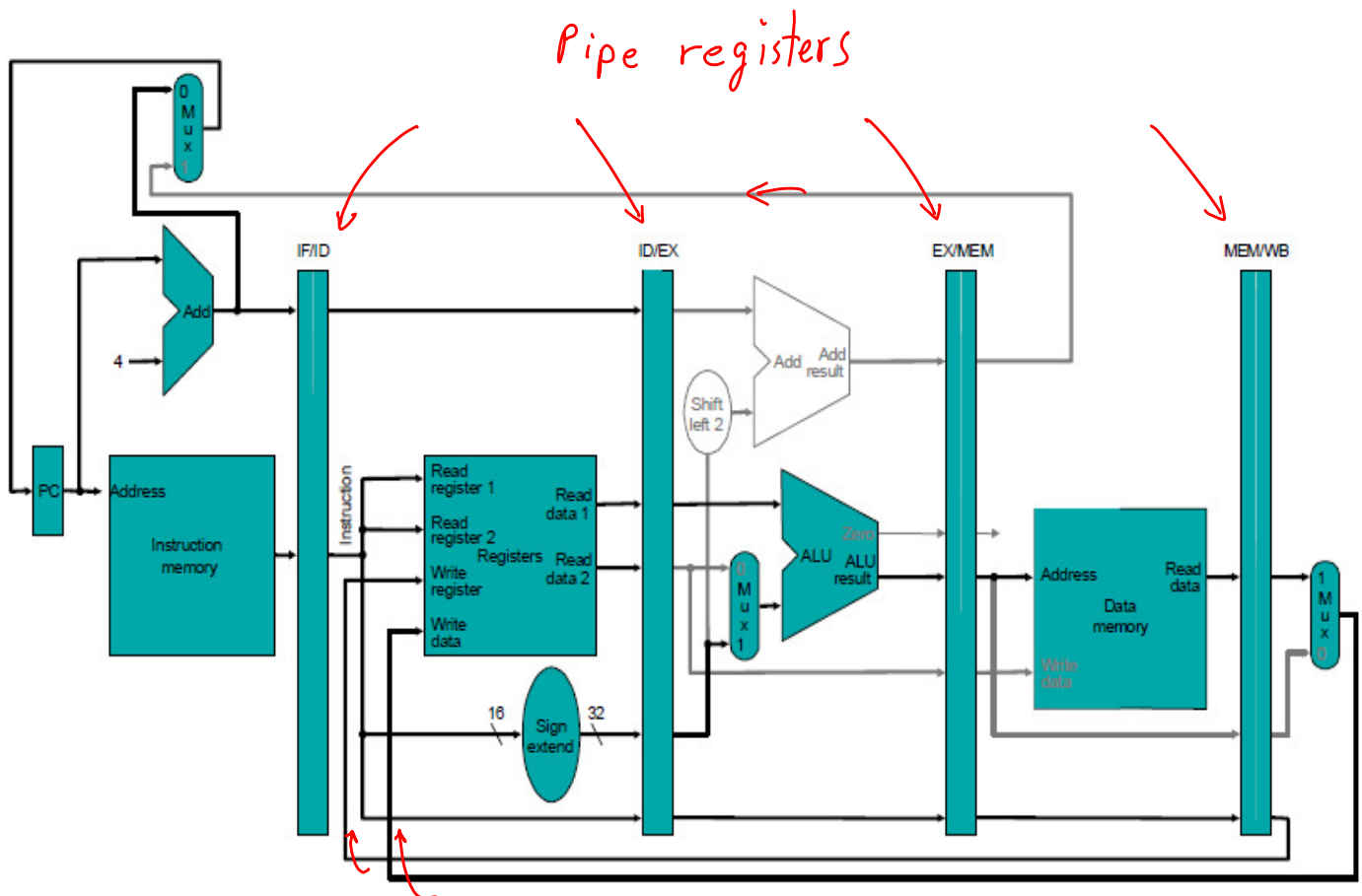
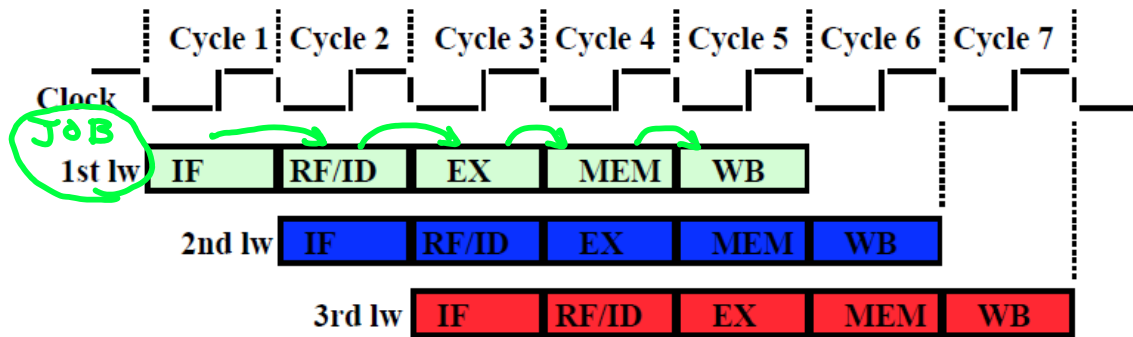
- Load instruction takes 5 stages
 - Five independent functional units work on each stage
 - Each functional unit used only once
 - Another load can start as soon as 1st finishes IF stage
 - Each load still takes 5 cycles to complete
 - The *throughput*, however, is much higher

*each stage
busy*

*1 job exits
per cycle*

*⇒ CPI
= 1 job
1 cycle*

T = 5 cycles



*NO!?!
→*

