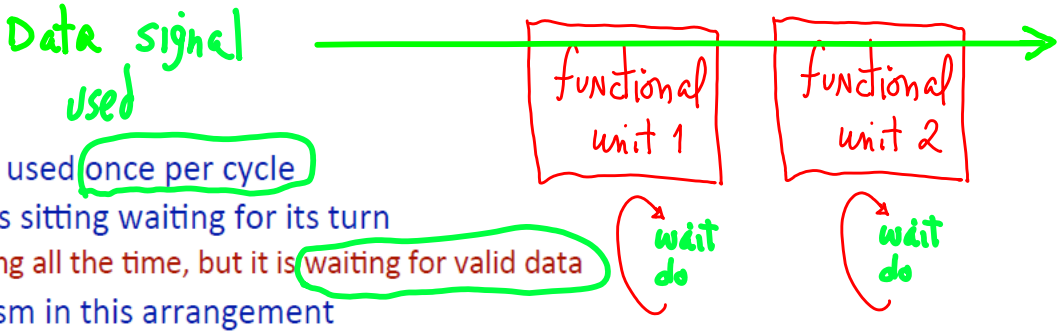


- Problem?
  - Each functional unit used once per cycle
  - Most of the time it is sitting waiting for its turn
    - Well it is calculating all the time, but it is waiting for valid data
  - There is no parallelism in this arrangement

- 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

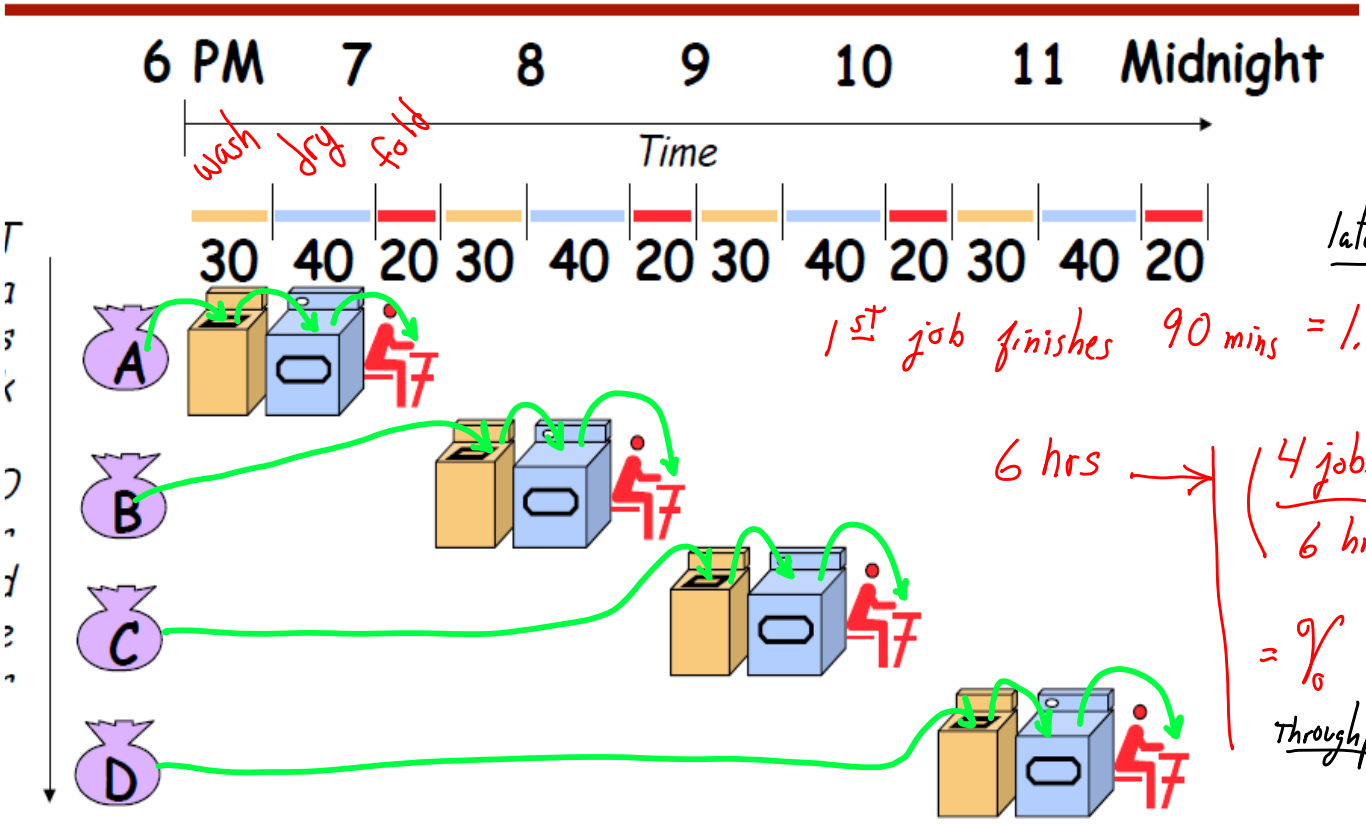
CPI ↑ CR ↑

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

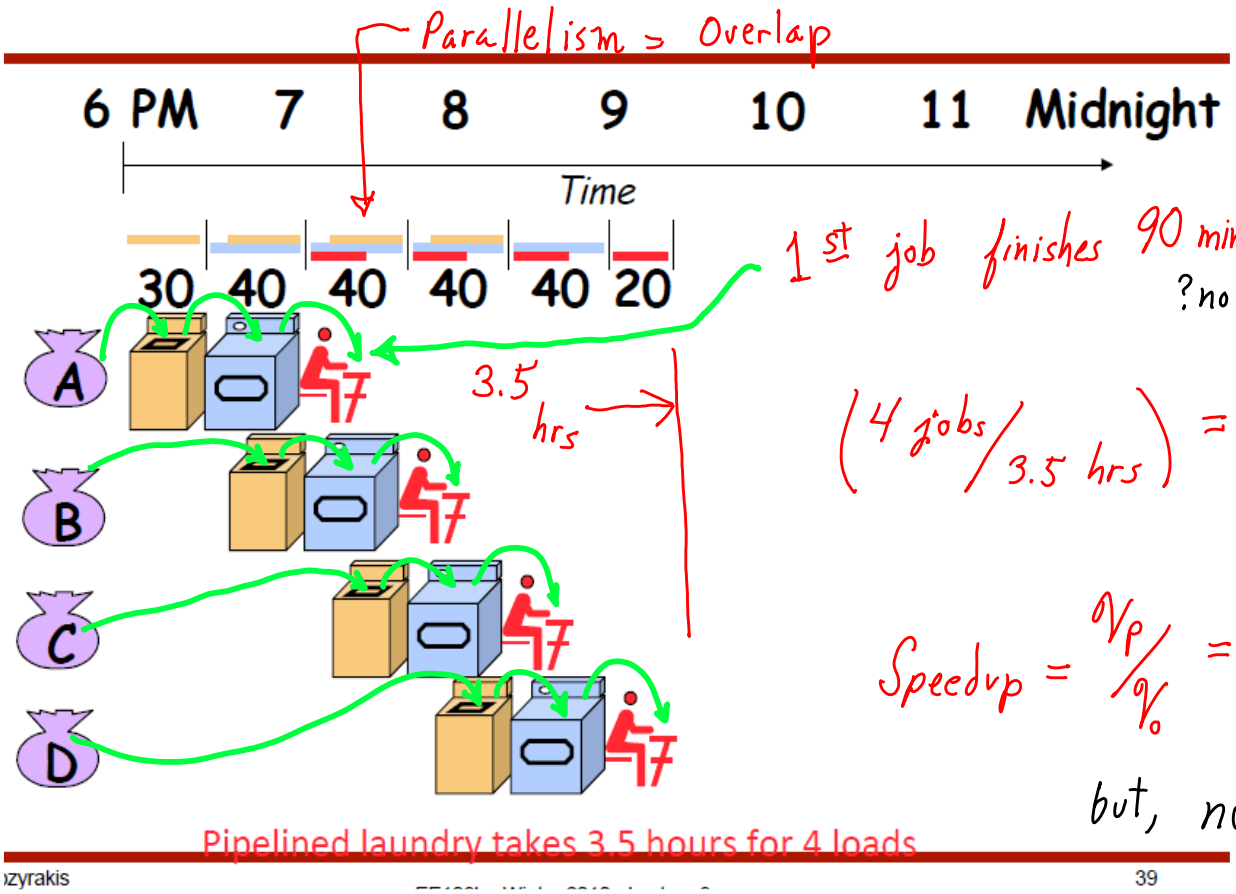
→ ↑CR/CPI ↑

hmm, ... ?

Sequential Laundry



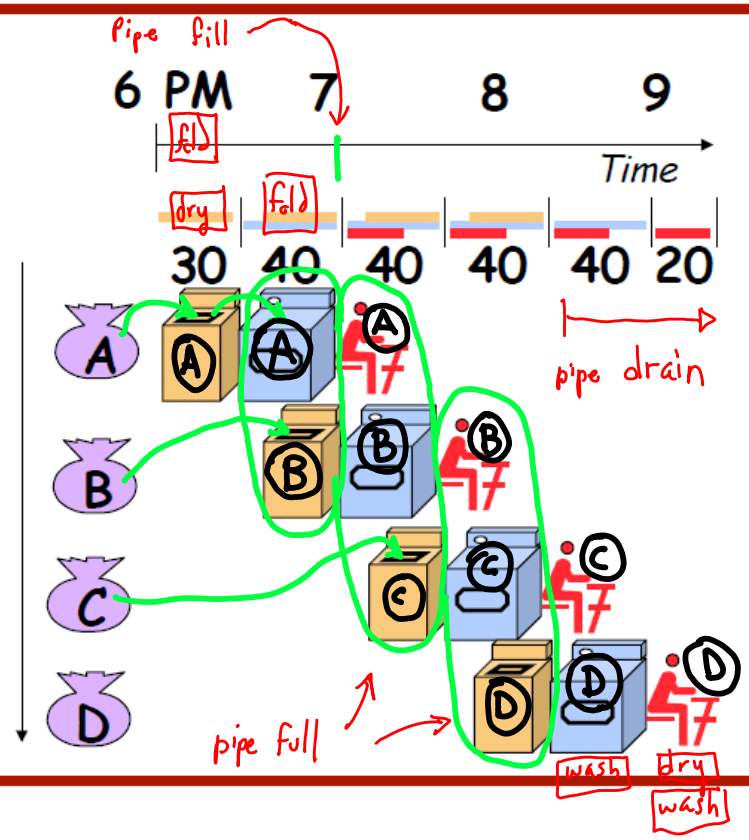
Sequential laundry takes 6 hours for 4 loads



What does Amdahl's Law say?

Pipelining Lessons

throughput ↑



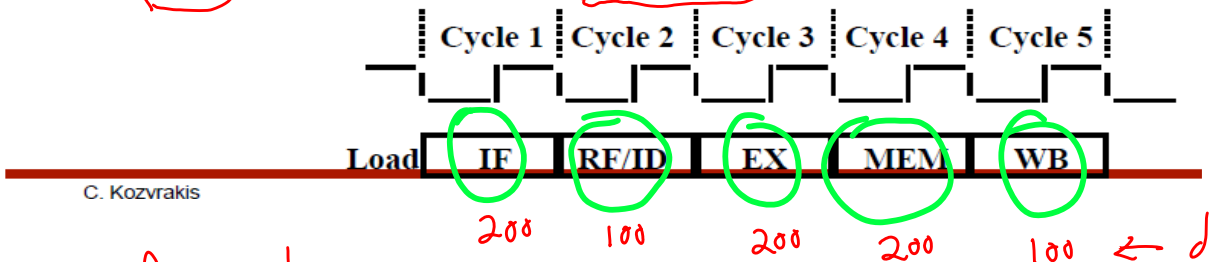
- Pipelining doesn't help latency of single task, it helps throughput of entire workload
- Multiple tasks operating simultaneously
- Potential speedup = Number pipe stages (max)
- Pipeline rate limited by slowest pipeline stage
- Unbalanced lengths of pipe stages reduces speedup
- Time to "fill" pipeline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

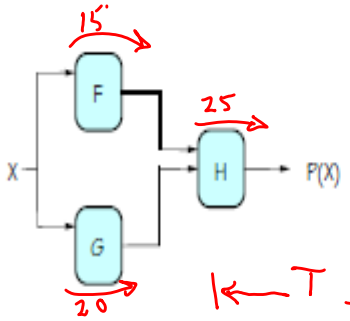
calc. address



200 100 200 200 100 ← delays, ps

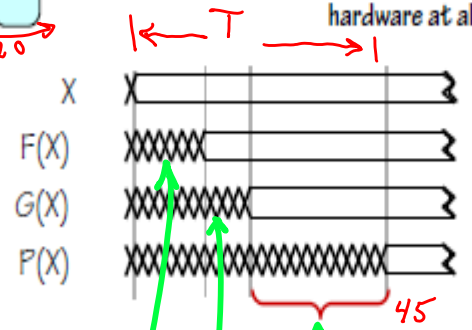
$(1/f_{CK}) \geq 200_{ps}$

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?

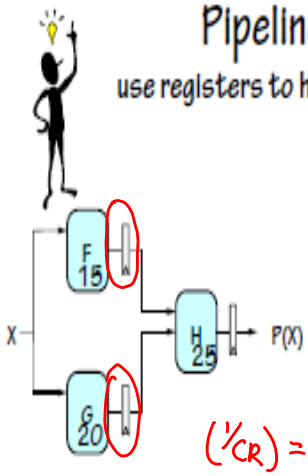


F & G are "idle", just holding their outputs stable while H performs its computation

15 20 25

$T = 20 + 25 = 45$

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



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.

$(1/f_{CK}) = 25$

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-stage pipeline	50	1/25

worse better

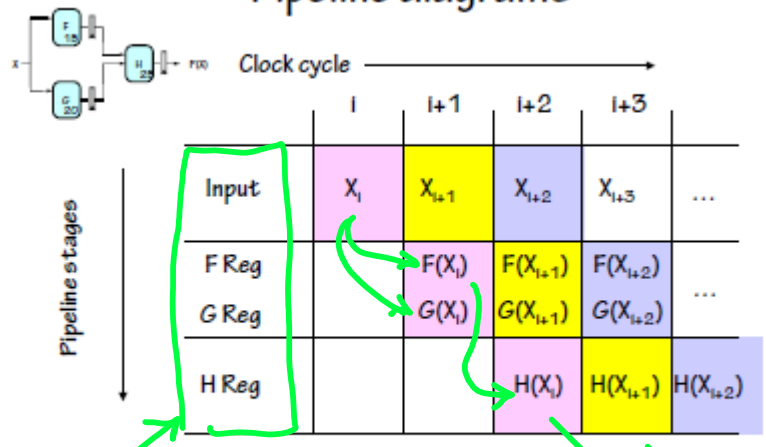
2 tics @ 25 → 1 job exits per 25

$T = 25 + 25 = 50$

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

different

Pipeline diagrams

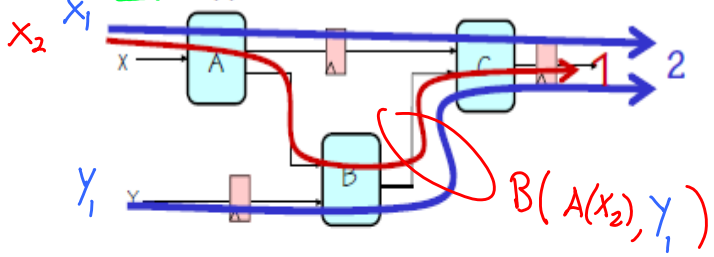


The results associated with a particular set of input data moves *diagonally* through the diagram, progressing through one pipeline stage each clock cycle.

pipe

job 1 job 2 job 3

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

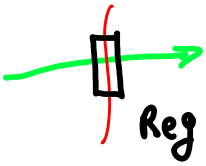
Step 1:

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.

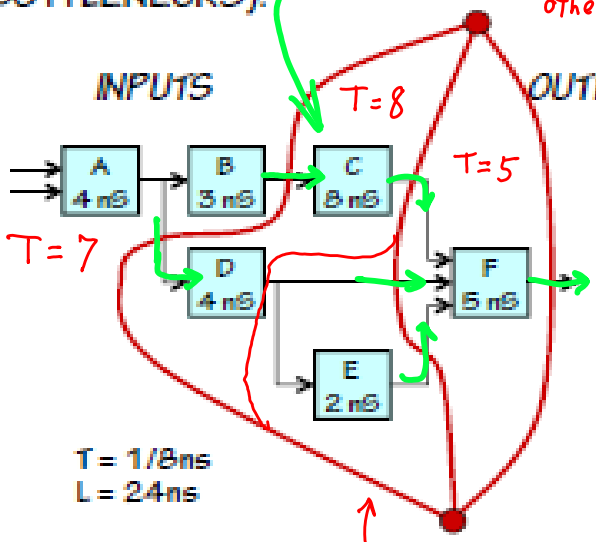
Adding a pipeline register at every point where a separating line crosses a connection will always generate a valid pipeline.



STRATEGY:

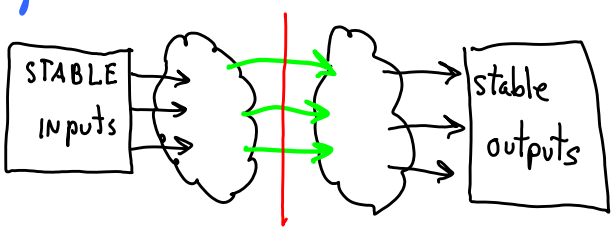
Focus your attention on placing pipelining registers around the slowest circuit elements (BOTTLENECKS).

isolation from other delays

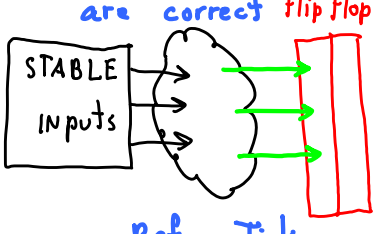


Inductive proof

signals in cut-set are correct

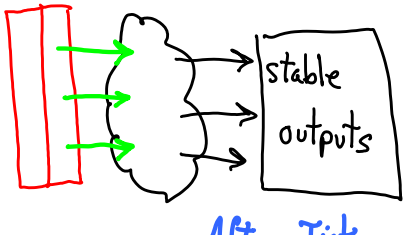


inputs to FF are correct



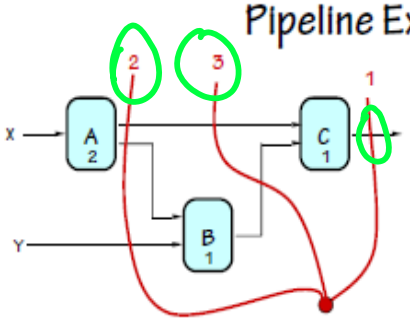
Before Tick

flip flop outputs are correct



After Tick

Pipeline Example

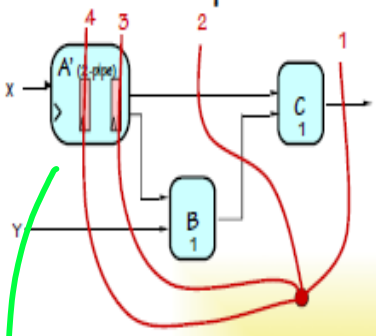


	LATENCY	THROUGHPUT
0-pipe:	2+1+1 = 4	1/4
1-pipe:	4	1/4
2-pipe:	2+2 = 4	1/2
3-pipe:	2+2+2 = 6	1/2

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.

Pipelined Components



4-stage pipeline, thruput=1

Pipelined systems can be hierarchical:

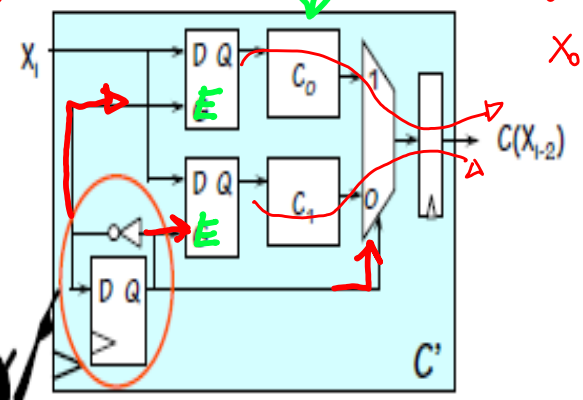
- Replacing a slow combinational component with a k-pipe version may increase clock frequency
- Must account for new pipeline stages in our plan

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

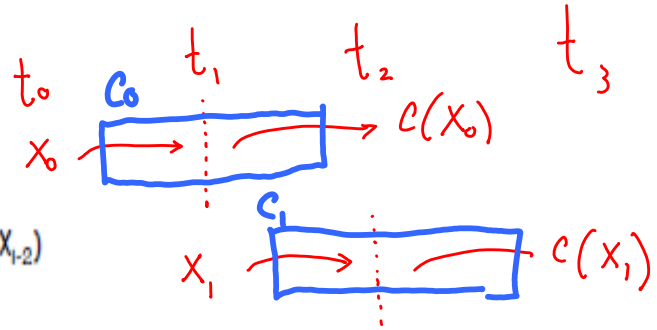
Circuit Interleaving

duplicate HW

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



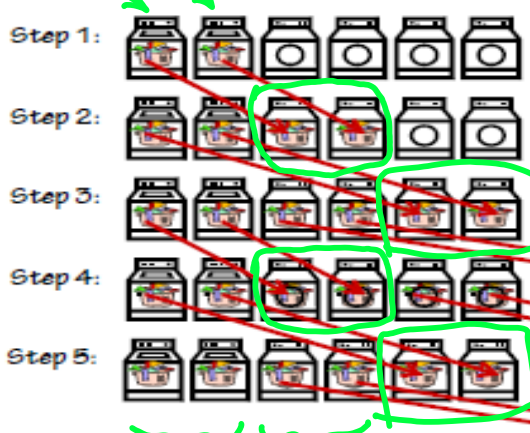
This is a simple 2-state FSM that alternates between 0 and 1 on each clock



Wash = 30 Dry = 2×30

2 loads And a little parallelism...

parallel pipes w/ interleave



We can combine interleaving and pipelining with parallelism.

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

Latency = 90 min

T = 90

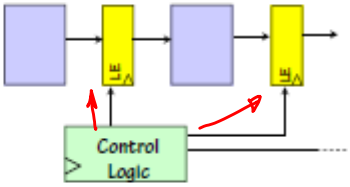
$\eta = \frac{2 \text{ jobs}}{30}$

2 pipes interleaved Dryers

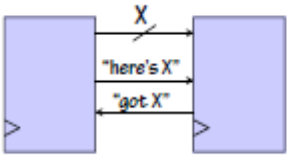
$S = \frac{2/30}{1/90} = 6$

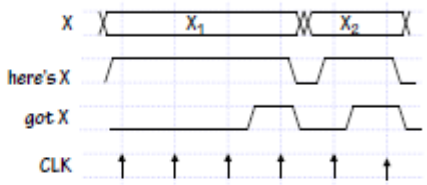
Control Structure Alternatives

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



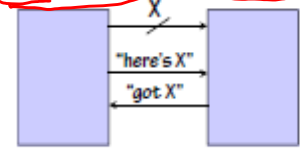
Synchronous, locally-timed:
Local circuitry, "handshake" controls
flow of data:

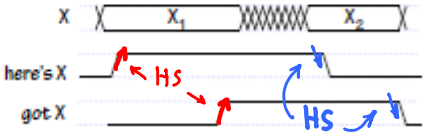




I/O busses w/ clk
both use same clock

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!

	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).

Global

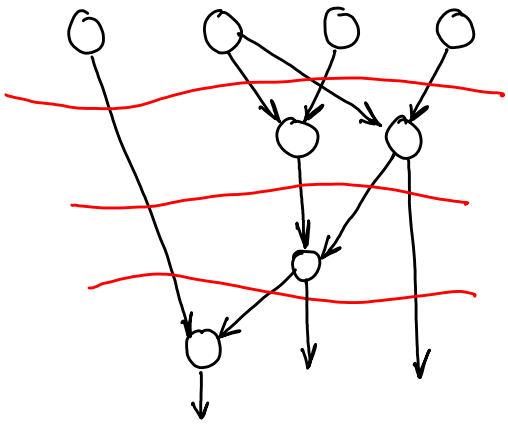
Local

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

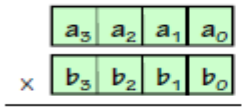
The "next big idea" for the last several decades: a lot of design work to do in general, but extra work is worth it in special cases

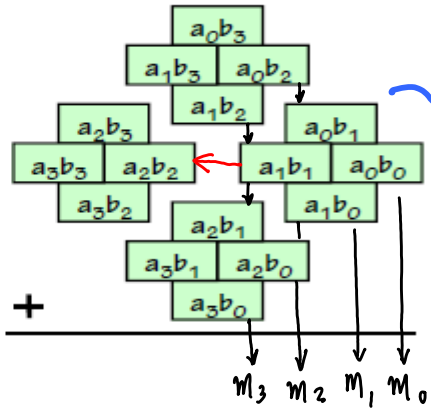
Data Dependency Graphs and Pipelining

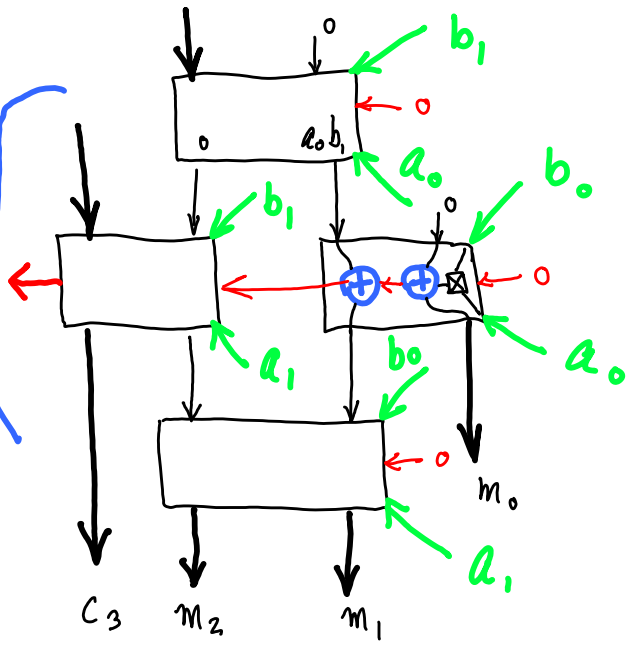
--- Multiple jobs
--- High throughput
--- Is parallelism as high as possible?
--- Is timing data dependent?



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

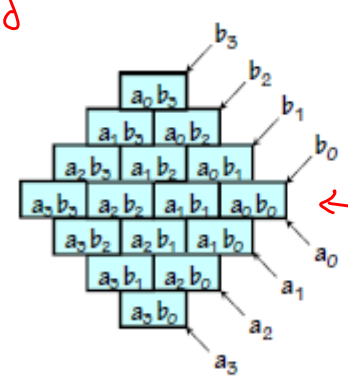






Design of 1-bit multiplier "Brick":

add



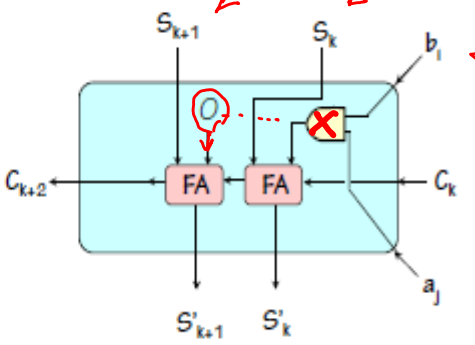
partial product

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

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!

add by columns

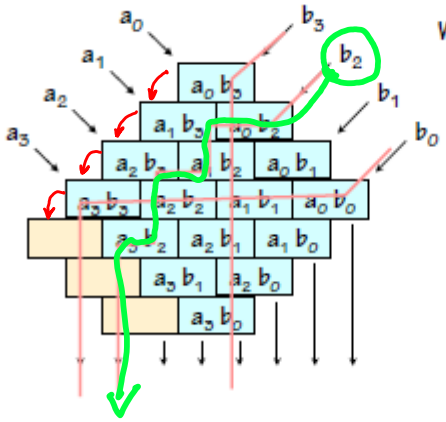


1-bit multiply ×
2-bit result

Column sums

Here's our combinational multiplier:

carry



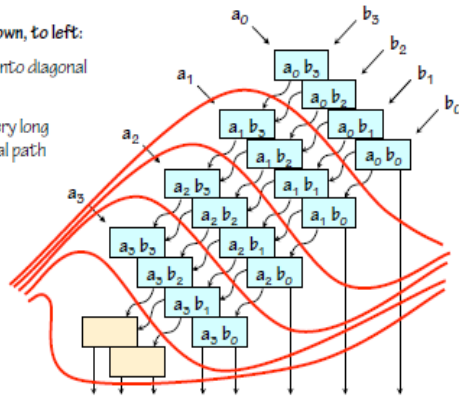
What's its propagation delay?

Naive (but valid) bound:
- O(n) additions
- O(n) time for each addition
- Hence O(n²) time required

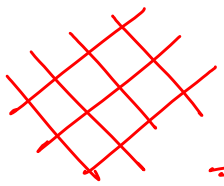
On closer inspection:
- Propagation only toward left, bottom
- Hence longest path bounded by length + width of array: O(n+n) = O(n)!

Breaking O(n) combinational paths

LONG PATHS go down, to left:
- Break array into diagonal slices
- Segment every long combinational path

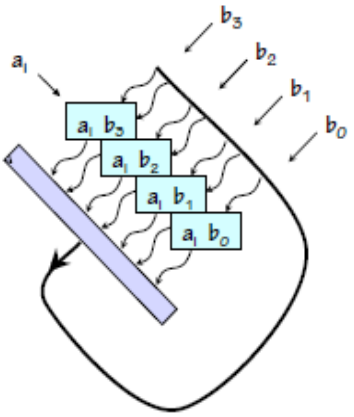


GOAL: Θ(n) stages; Θ(1) clock period!



cut both ways ⇒ systolic

Multiplier Cookbook: Chapter 4



Sequential Multiplier:

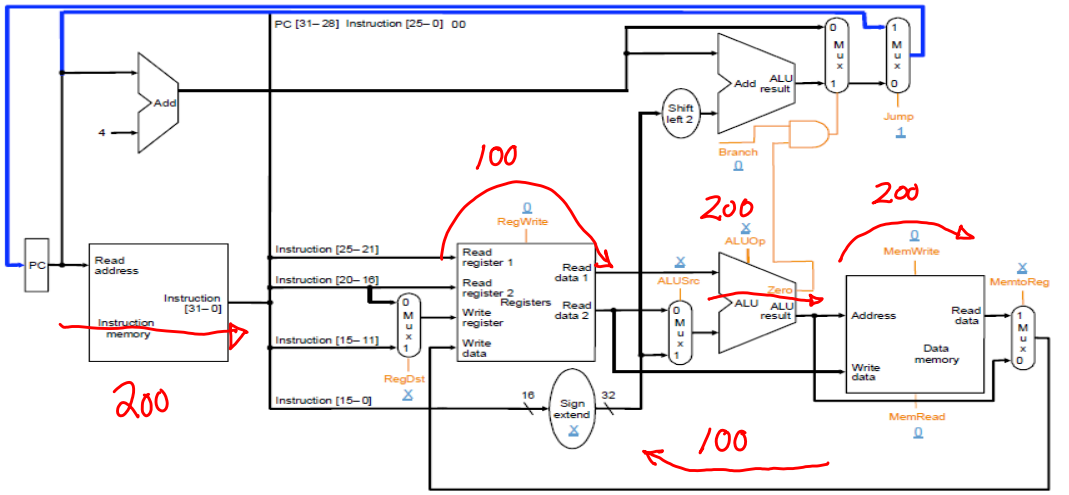
- Re-uses a single n-bit "slice" to emulate each pipeline stage
- a operand entered serially
- Lots of details to be filled in...

Stages:	1
Clock Period:	Θ (1) (constant!)
Hardware cost for n by n bits:	Θ (n)
Latency:	Θ (n)
Throughput:	Θ (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 > 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 ∞ Frequncy

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}

$1/T_{clock} = \frac{1}{0.8\,ns} = 1.25\,GHz$

- 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 = 0.6x45% + 0.8x25% + 0.7x10% + 0.5x15% + 0.2x5%
  = 0.625 ns (1.6GHz)

⇒ 1.6 GHz

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

Pipelining Load lw

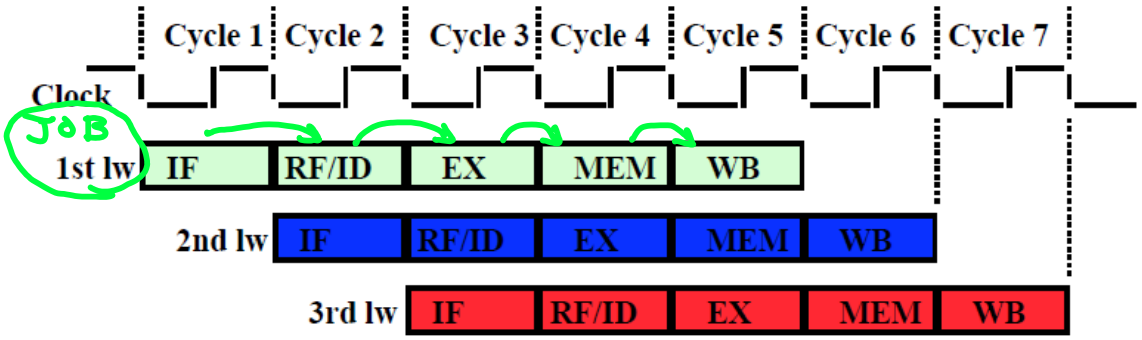
- Load instruction takes 5 stages
  - Five independent functional units work on each stage
    - Each functional unit used only once
  - Another load can start as soon as 1st finishes IF stage
  - Each load still takes 5 cycles to complete
  - The *throughput*, however, is much higher

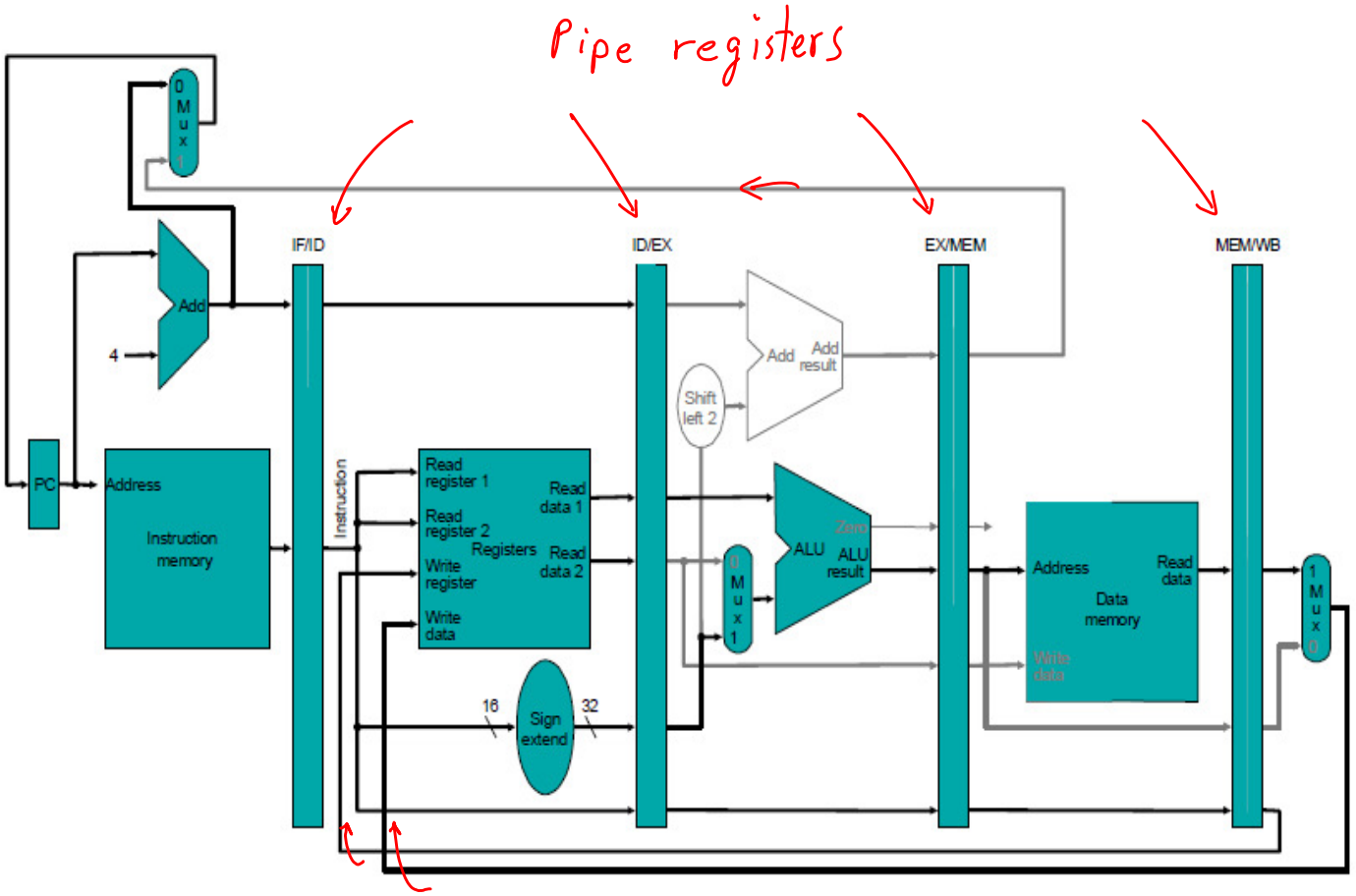
each stage busy

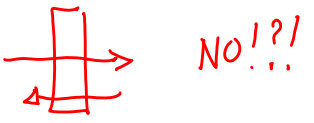
1 job exits per cycle

⇒ CPI = 1 job / 1 cycle

T = 5 cycles







NO!?!