Parallelism

Parallelism: execute multiple operations simultaneously

Some Types

--- Instruction-Level Parallelism **===>** execute multiple instructions from same job **(ILP)**

--- Data Parallelism **===>** operate on multiple data items from same job **(SIMD, MIMD, SPMD)**

--- Thread-Level Parallelism **===>** execute multiple jobs but same program

--- Task-Level Parallelism **===>** execute multiple jobs, multiple programs

ILP, Pipeline

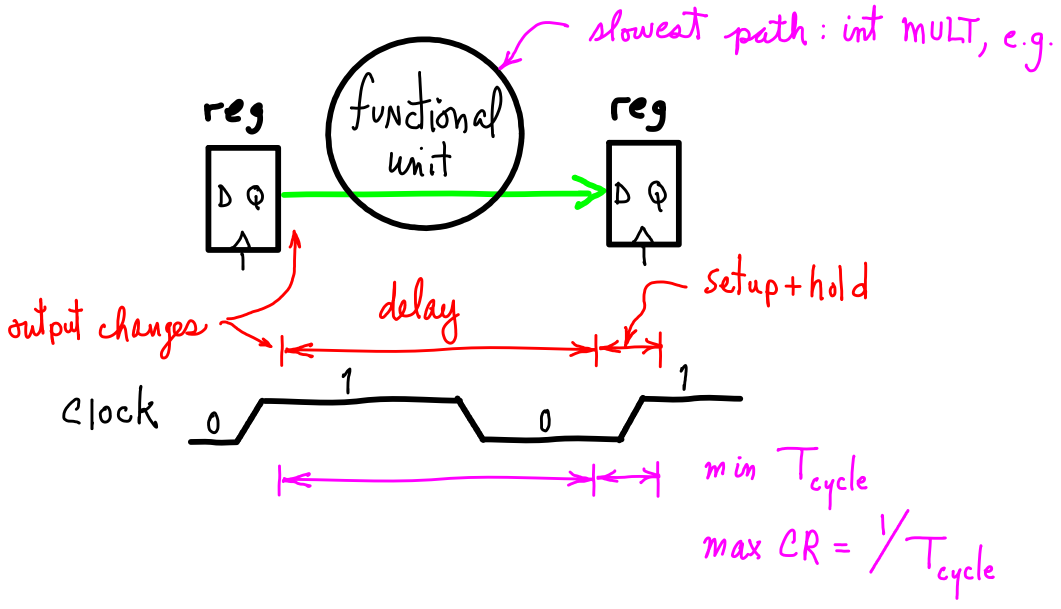
$$Perf = \frac{n \text{ instructions}}{time} = \frac{n}{n\, CPI\, (1/CR)} = CR/CPI \qquad CR\uparrow \Rightarrow perf\uparrow$$

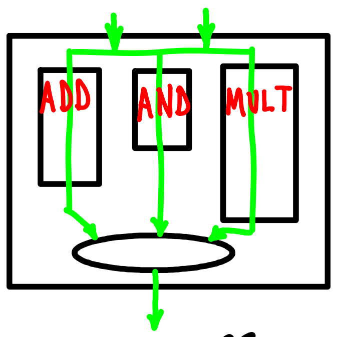
CR limited by slowest path.

--- Other, faster units could be clocked faster.

--- But cannot because of slow path.

--- Faster units waiting/idle.





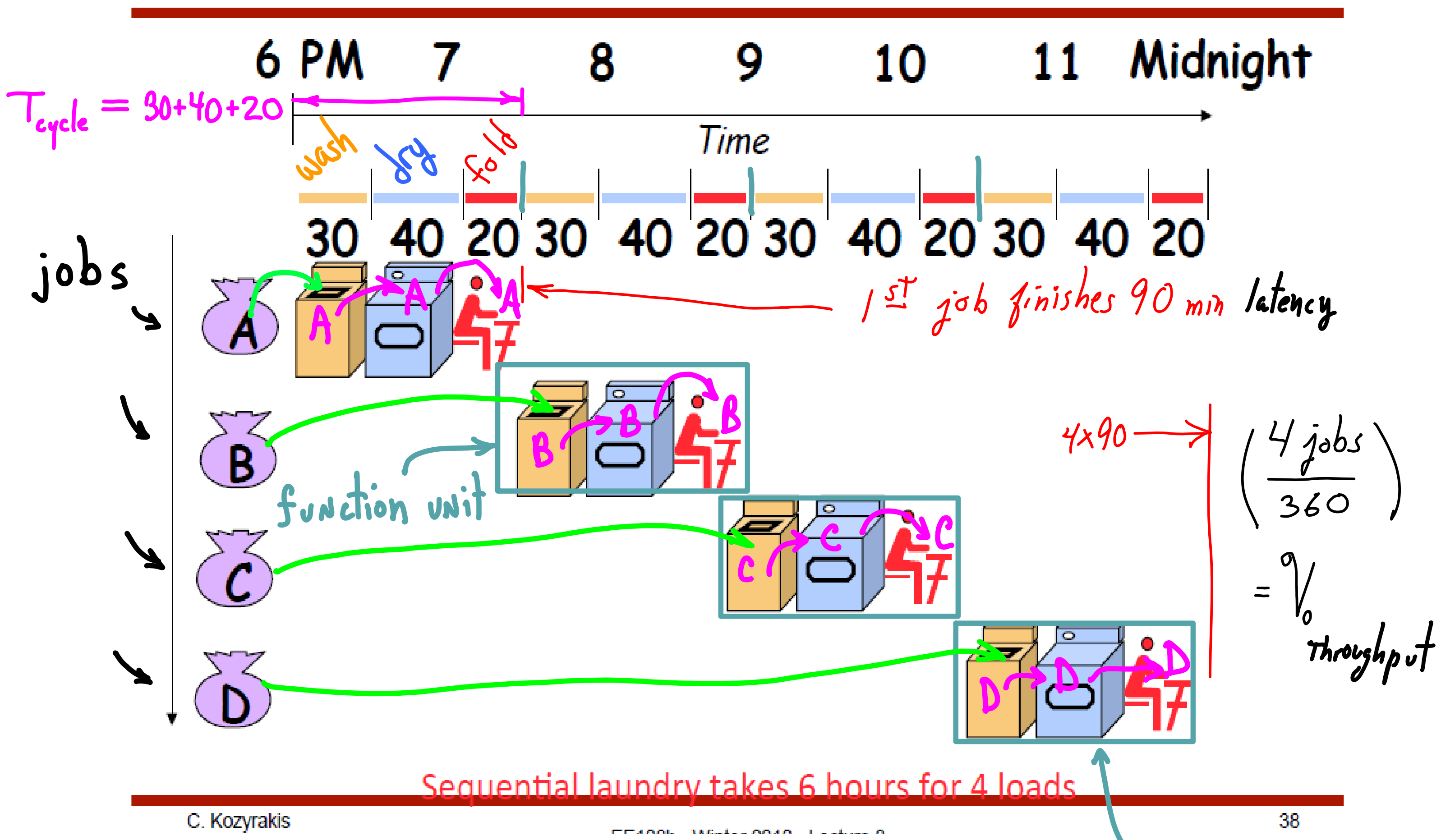
ALU

- Only 1 unit used
- every instruction takes same time

Amdahl's Law says attack the common case.
Every instruction faster w/ increased **CR**.
--- **MULT** chopped into k fast pieces.
--- More cycles for MULT, **but** faster CR.

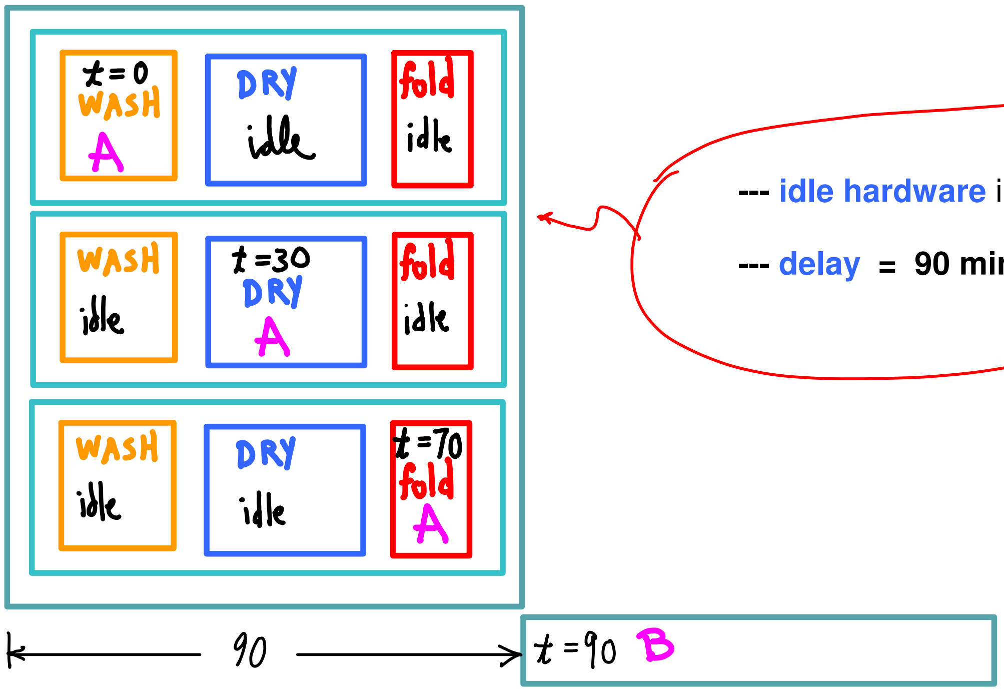
Also, see if we can get k pieces of MULT to run in Parallel

Sequential: 1-piece functional unit


6 PM
7
8
9
10
11
Midnight
Time
$T_{cycle} = 30+40+20$
wash
dry
fold
30 40 20 30 40 20 30 40 20 30 40 20
jobs
A
B
C
D
function unit
1st job finishes 90 min latency
4x90
$\left(\frac{4 \text{ jobs}}{360}\right) = \frac{1}{90}$ Throughput
Sequential laundry takes 6 hours for 4 loads
C. Kozyrakis
38

functional unit at $t \in [3\times90, 4\times90]$

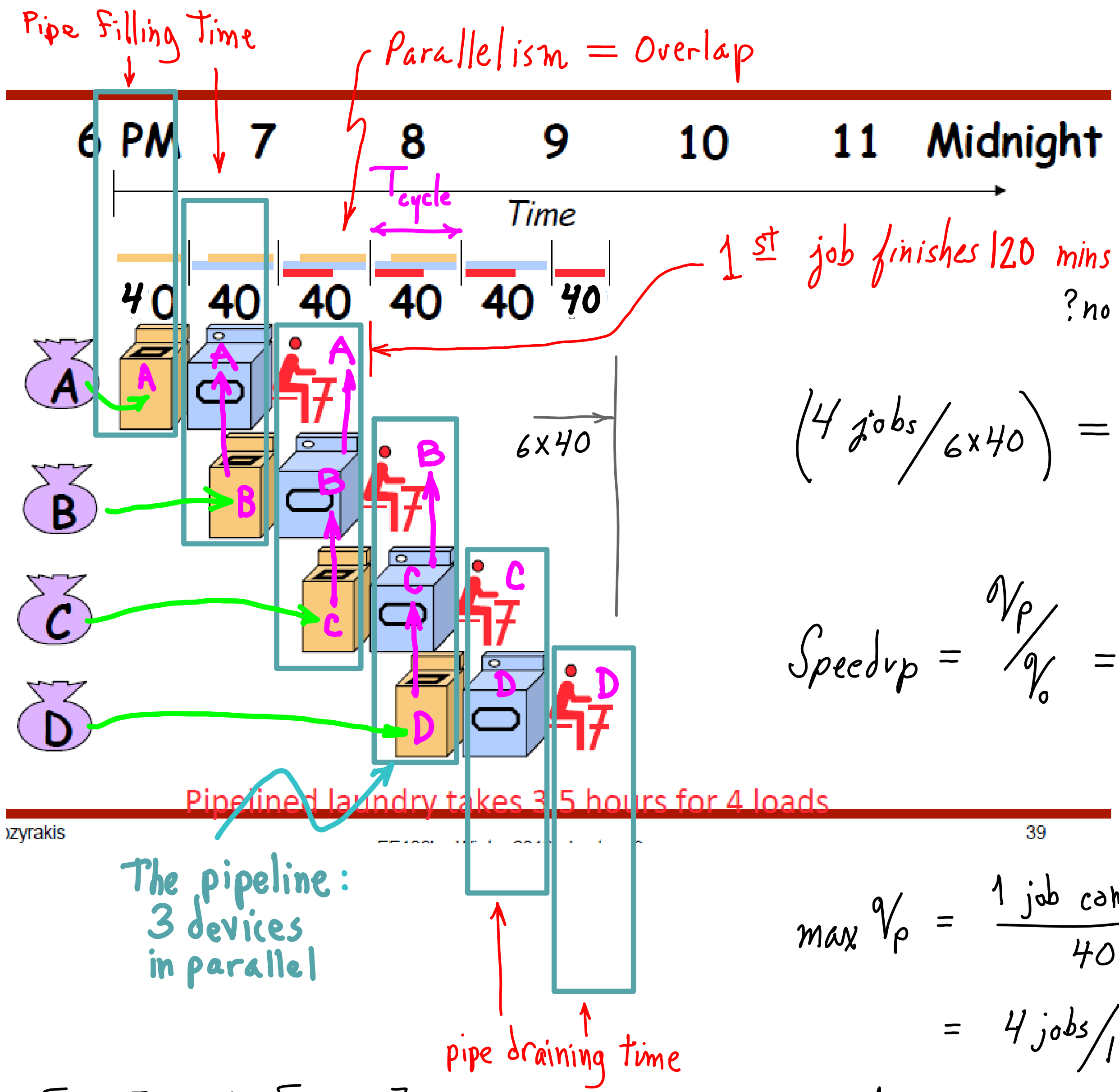
Laundry unit = [wash + dry + fold]



--- **idle hardware** in **LAUNDRY** unit

--- **delay** = **90 min**

Pipelined : pipelined functional unit



$$\left(4 \text{ jobs} / 6 \times 40 \right) = \mathcal{V}_p \text{ Throughput}$$

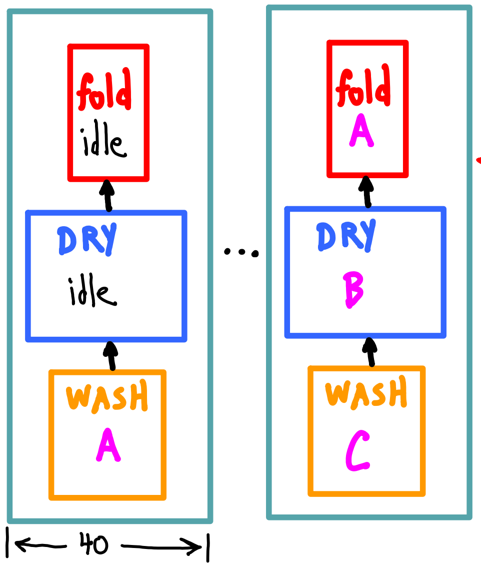
$$\text{Speedup} = \frac{\mathcal{V}_p}{\mathcal{V}_0} = \frac{360}{240} = \frac{3}{2}$$

$$\max \mathcal{V}_p = \frac{1 \text{ job completes}}{40}$$

$$= 4 \text{ jobs} / 160$$

$$S = 360/160 = 2^+$$

$t \in [0,40]$ $t \in [80,120]$



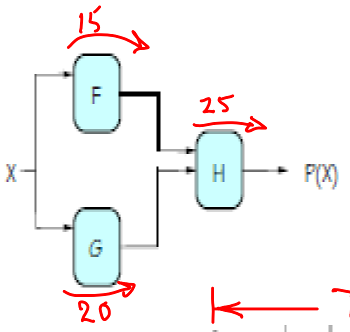
$$\frac{CR_{new}}{CR_{old}} = \frac{T^{old}_{cycle}}{T^{new}_{cycle}} = 90/40 = 2^+$$

Other instructions $S = 2^+$

$$\% \text{ overhead} = \frac{\text{fill time} + \text{drain time}}{\text{total time}}$$

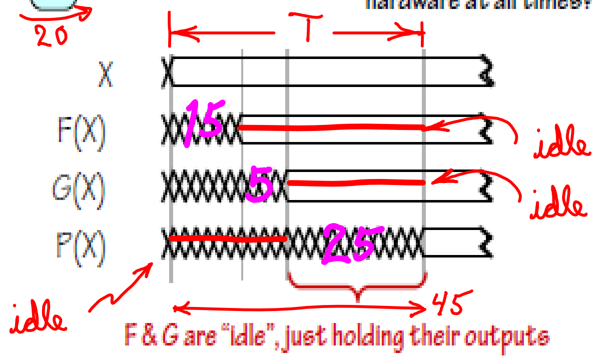
decreases as total time increases

General pipelines



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

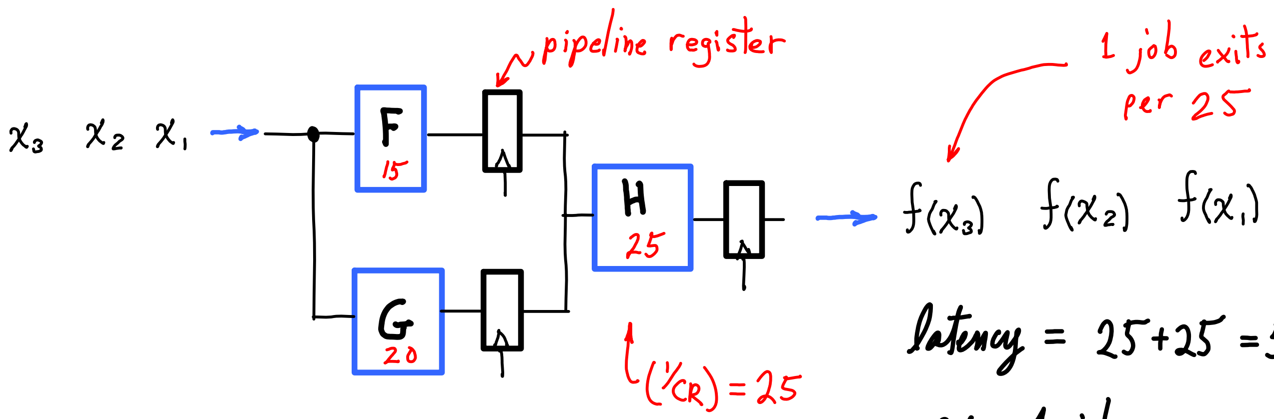
We can't get the answer faster, but are we making effective use of our hardware at all times?



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

latency = 45

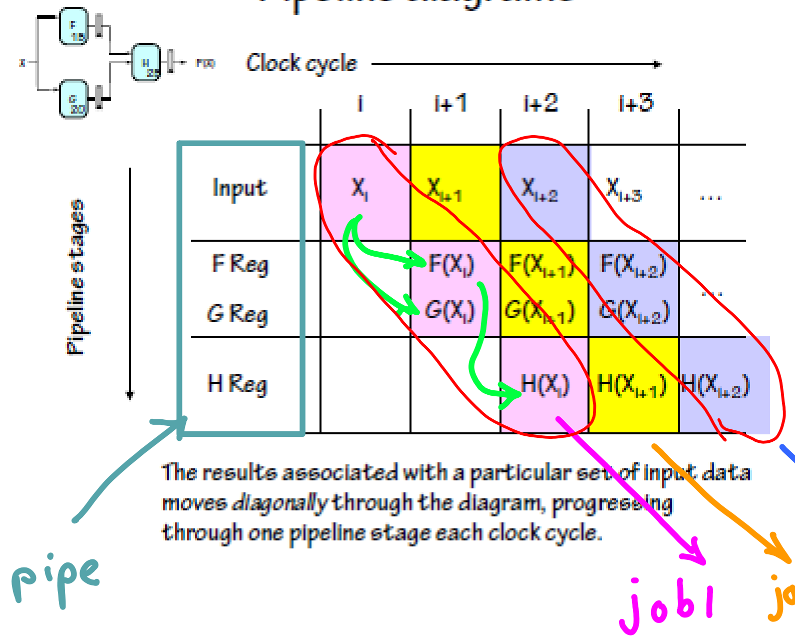
$\gamma = \frac{1 \text{ job}}{45}$



latency = 25+25 = 50

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

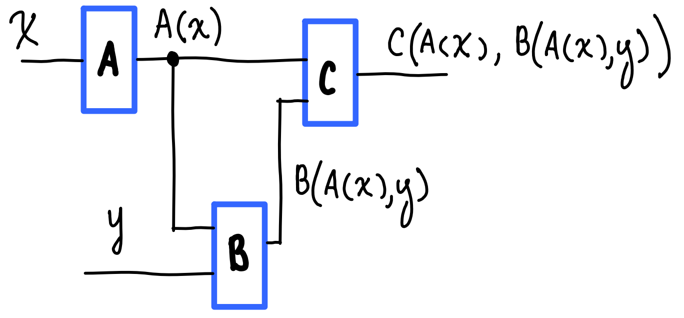
different
Pipeline diagrams

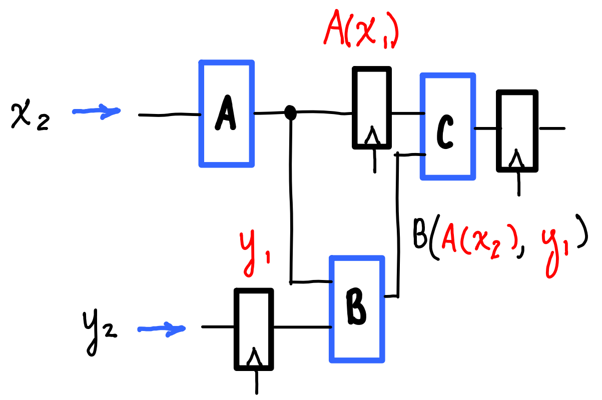


	i	i+1	i+2	i+3	
Input	X_i	X_{i+1}	X_{i+2}	X_{i+3}	...
F Reg G Reg		$F(X_i)$ $G(X_i)$	$F(X_{i+1})$ $G(X_{i+1})$	$F(X_{i+2})$ $G(X_{i+2})$	...
H Reg			$H(X_i)$	$H(X_{i+1})$	$H(X_{i+2})$

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

Bad pipelining





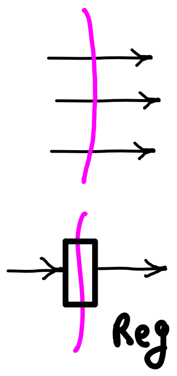
What's wrong? How to fix?

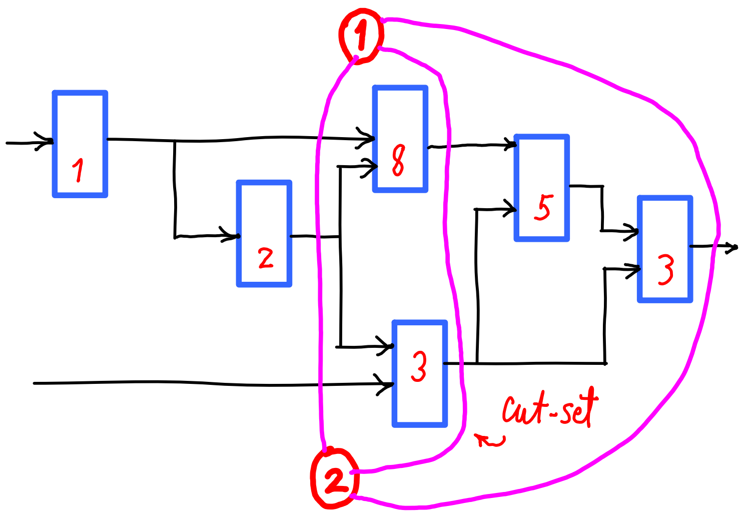
General Method

Cut all paths
in **same direction**
bisect graph

===> cut set

For each cut
add Reg to path

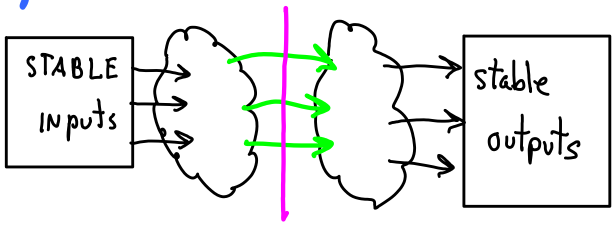


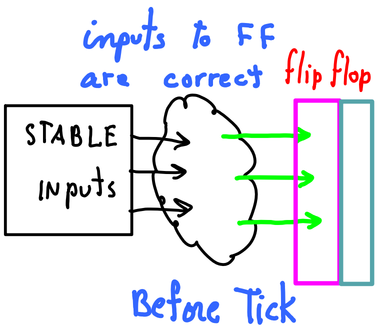


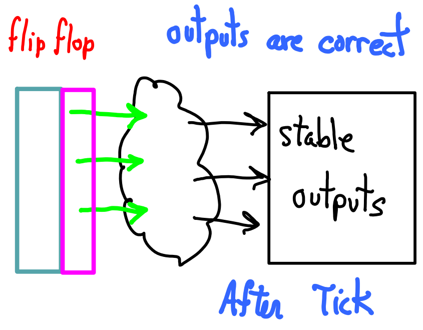
balance stage delays

Inductive proof

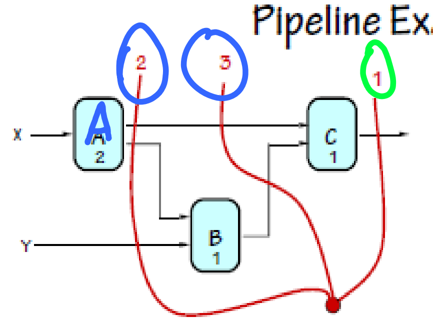
signals in cut-set are correct







Pipeline Example

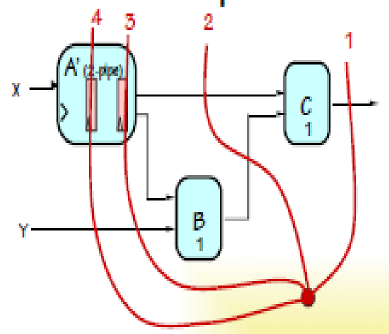


	LATENCY	THROUGHPUT
CR = 1/4 — 0-pipe:	$\frac{2+1+1}{4}$	1/4
~~1-pipe:~~	~~4~~	~~1/4~~
CR = 1/2 — 1, 2-pipe:	$\frac{2+2}{4}$	1/2
CR = 1/2 — 1, 2, 3-pipe:	$\frac{2+2+2}{6}$	1/2

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:

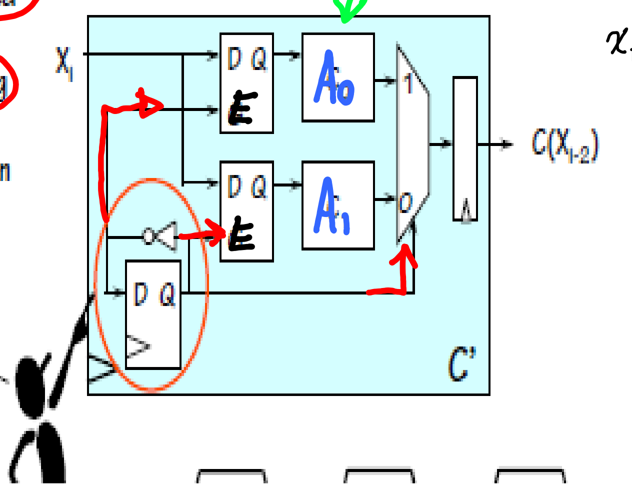
- Replacing a slow combinational component with a k-pipe version may increase clock frequency
- Must account for new pipeline stages in our plan

Can't split A in two w/ T=1 ? Fake it!

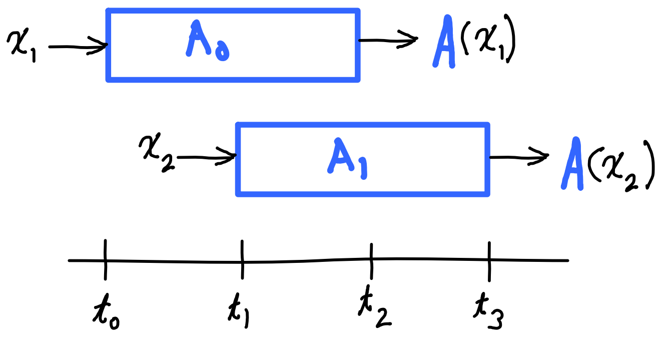
Circuit Interleaving

duplicate A

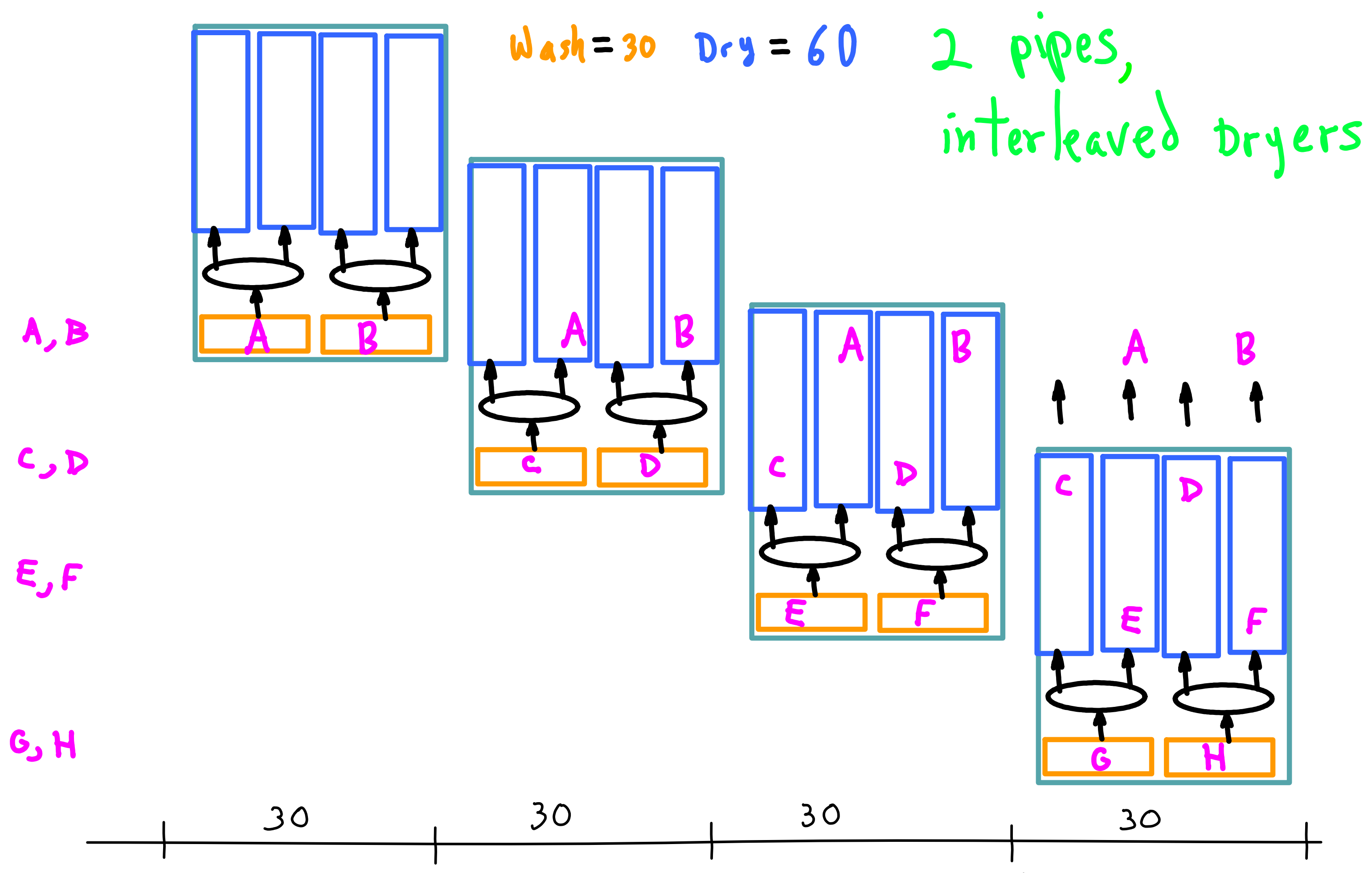
We can simulate a pipelined version of a slow component by replicating the critical element and alternate inputs between the various copies.



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



Parallel pipes + interleave



latency = 90

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

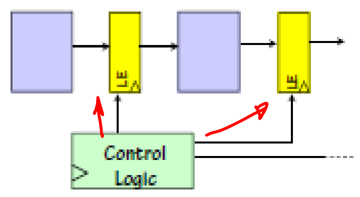
$\overline{\text{CPI}} = ?$ $\quad T_{cycle} = 30$

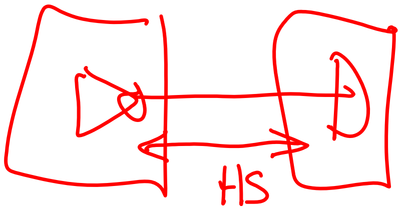
$1 \text{ cycle} / 2 \text{ instr} = 1/2$ Provided pipe is full.

superscalar

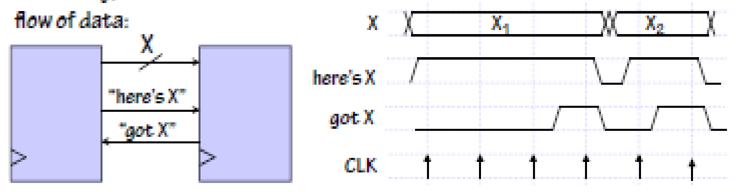
Control Structure Alternatives

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



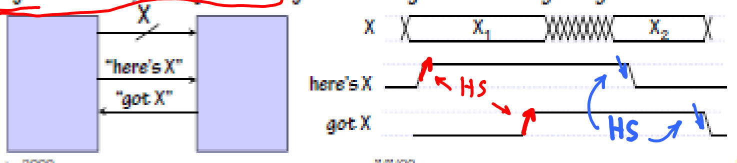


Synchronous, locally-timed:
Local circuitry, "handshake" controls flow of data:



I/O busses w/ clk
both use same clock

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



independent clocks
slow I/O device
w/ HS protocols

(or no clocks, Self-timed)
(fast Logic)

Control Structure Taxonomy

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

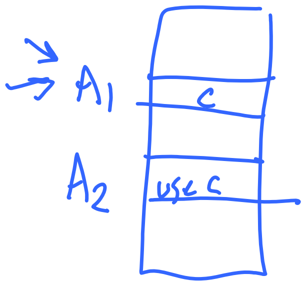
Large systems lead to very complicated timing generators… just say no!

	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).

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

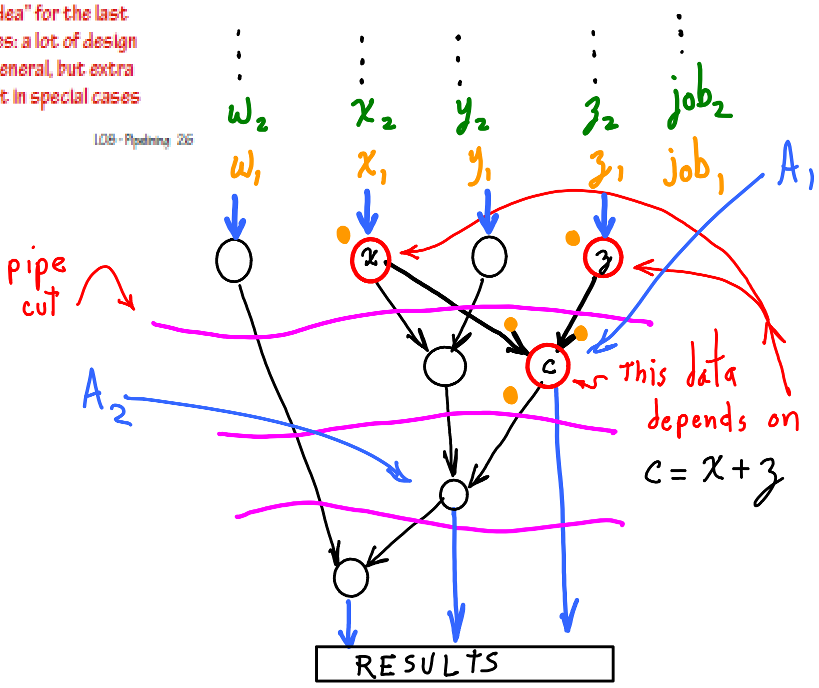
The "next big idea" for the last several decades: a lot of design work to do in general, but extra work is worth it in special cases

Global
Local

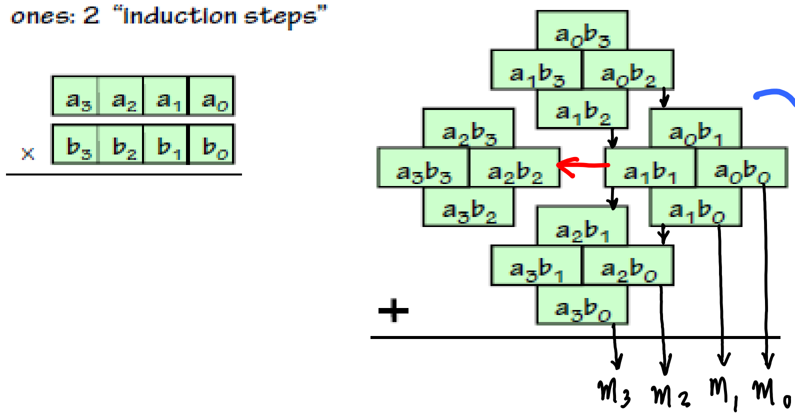


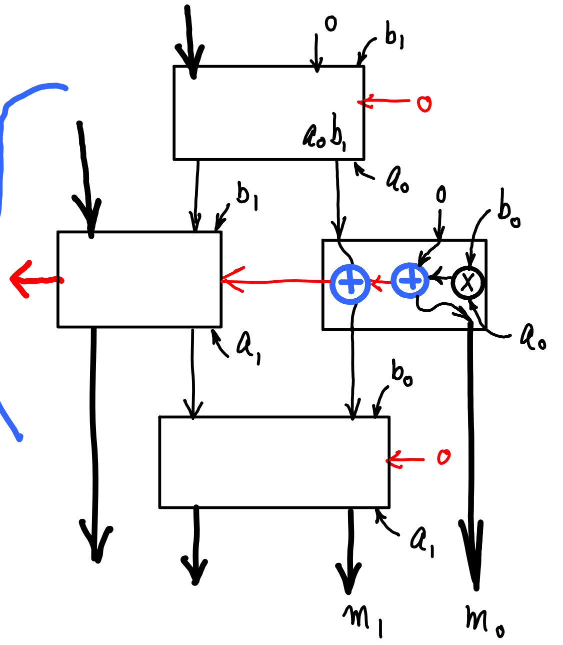
Data Dependency Graphs and **Pipelining**

--- **Entire algorithm** is **dependency graph**
--- Data-Flow: nodes fire when inputs ready
--- **Multiple jobs**
--- **High throughput**
--- Is parallelism **as high as possible**?
--- Does **timing depend on data**?



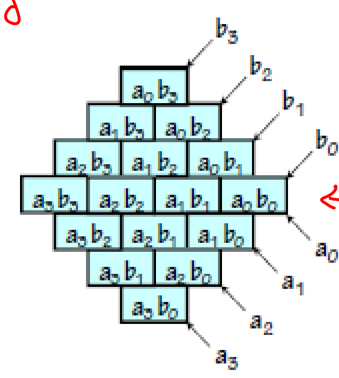
Making 4n-bit multipliers from n-bit ones: 2 "induction steps"





Design of 1-bit multiplier "Brick":

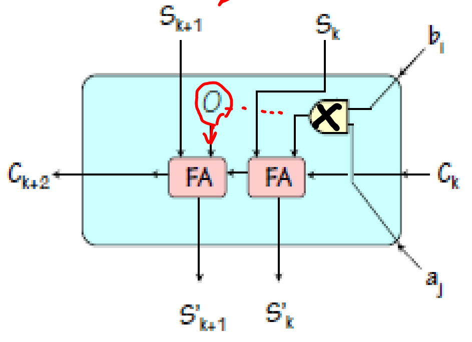
add



partial product

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

add by columns



1-bit multiply ×

2-bit result

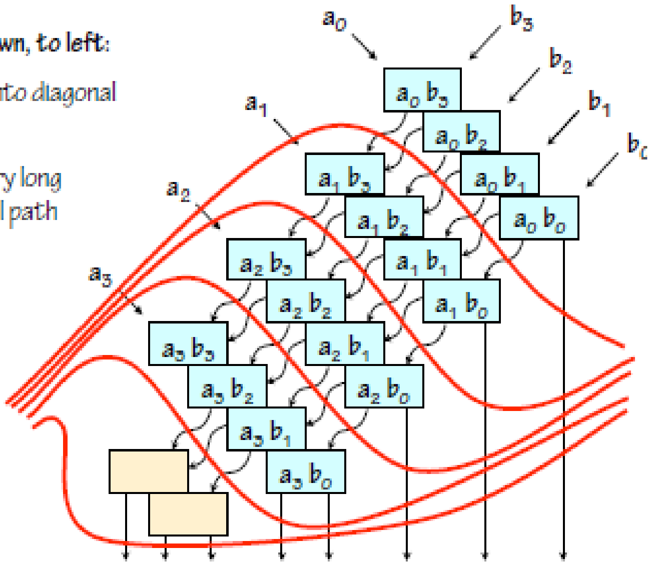
Column sums

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!

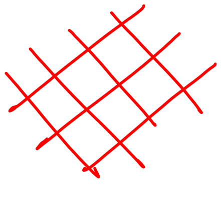
Breaking O(n) combinational paths

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



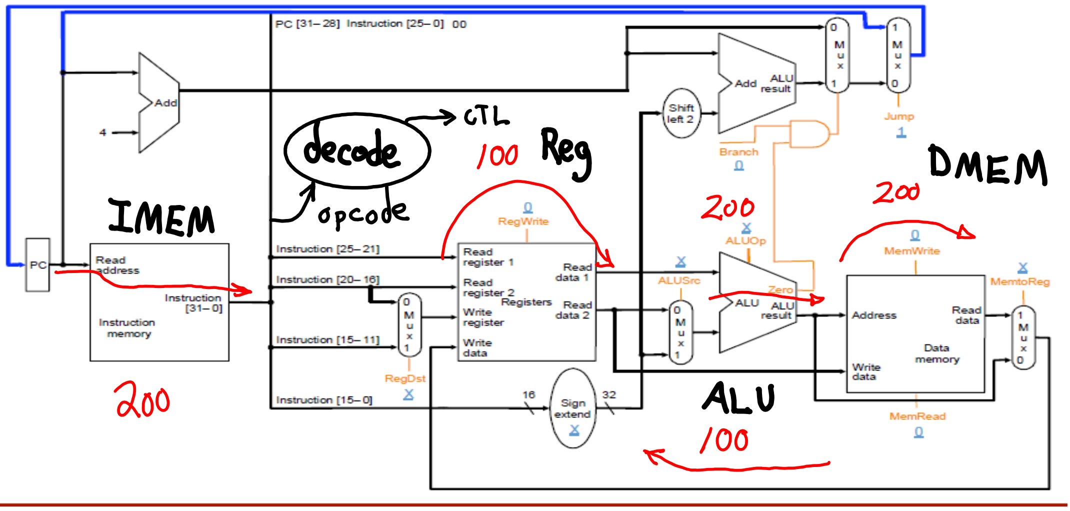
cut both ways?

⟹ systolic array



GOAL: Θ(n) stages; Θ(1) clock period!

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}

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

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

what if we let clock trigger delay by opcode?

BR 500 ps

LD 800 ps

- 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

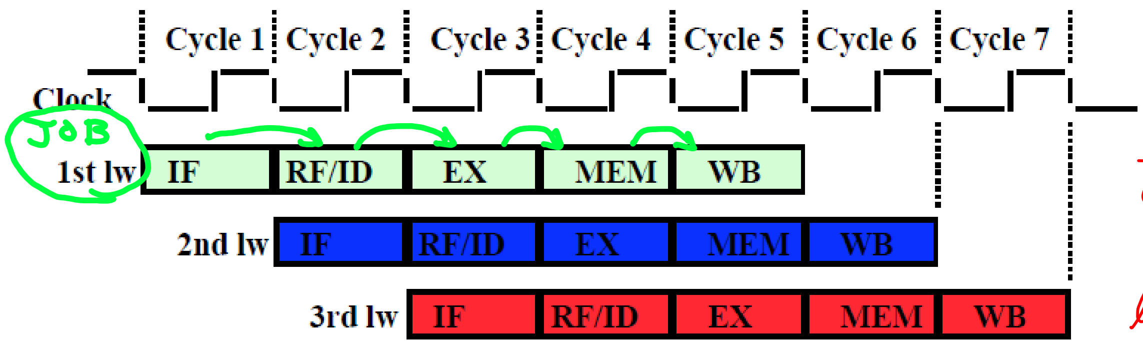
- CPU clock cycle = 0.6x45% + 0.8x25% + 0.7x10% + 0.5x15% + 0.2x5%
  = 0.625 ns (1.6GHz)

⟹ 1.6 GHz

speedup? $S_{new-old} = 1.6/1.25 = 1.28$

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

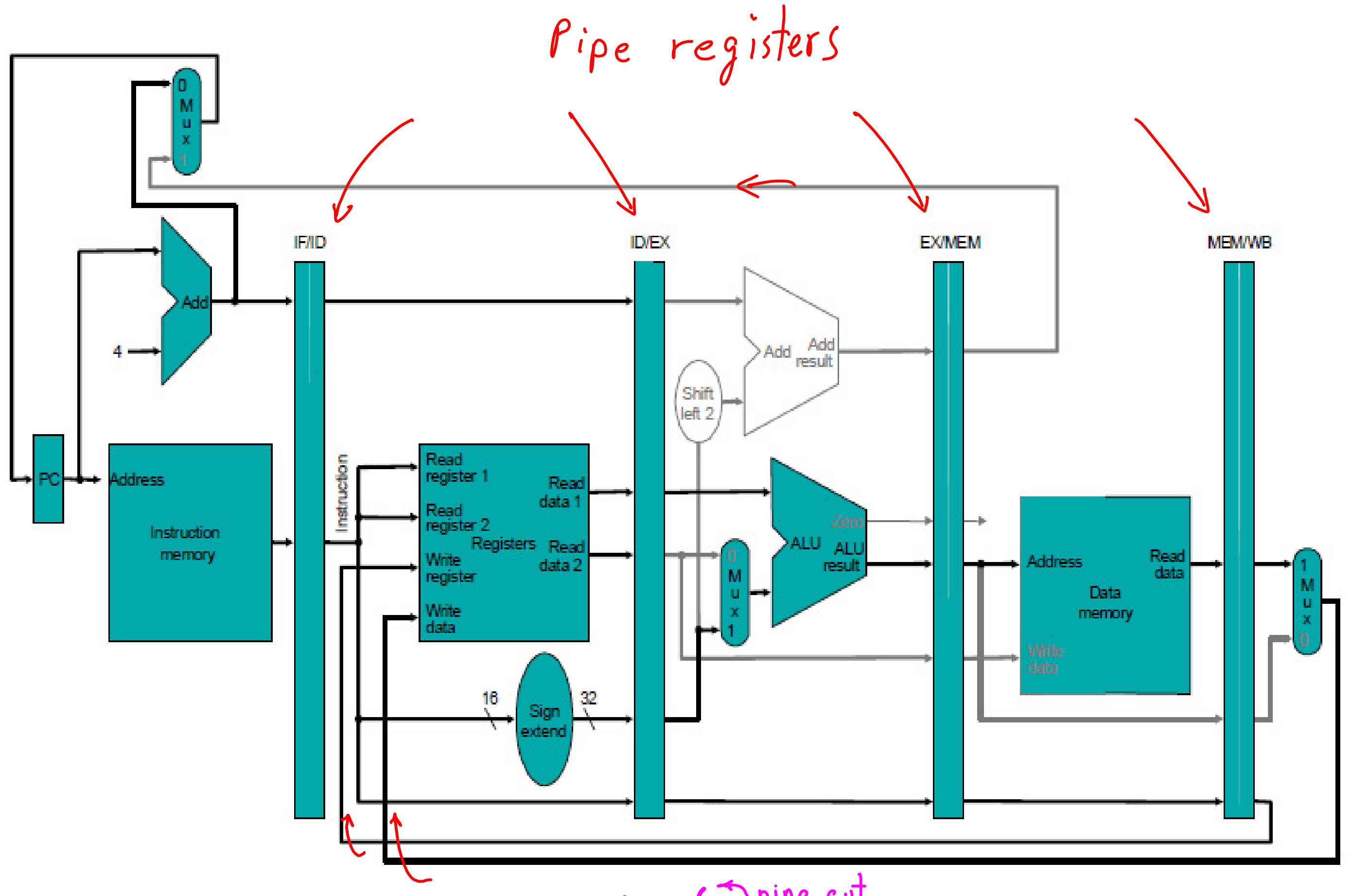


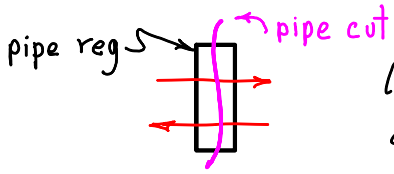
when all stages busy

1 instr. exits per cycle

$CPI = \frac{1 \text{ instr.}}{1 \text{ cycle}}$

latency = 5 cycles





Cannot (?) cut both directions. Can, but...

$$\boxed{\text{Max out pipeling?}}$$

$$\text{delay} \rightarrow \frac{\text{delay}}{n} \implies CR \rightarrow n \times CR \qquad (n \rightarrow \infty ?)$$

NO:

- $T_{cycle} > (\text{setup} + \text{hold}) \implies CR < 1/(\text{setup} + \text{hold})$
- small operations harder to divide into pieces
- n stages $\implies (n \text{ fill/drain overhead}) \times (T_{cycle} = 1/n \times CR)$

  looks like n cancels, but keeping pipe full gets harder → pay penalty more often
- $n \times CR \implies n \times \text{Power}$

≈ 20 stages limit for CPU (But elsewhere?)