
Digital Design: A Systems Approach

Lecture 11: Pipelining

Readings

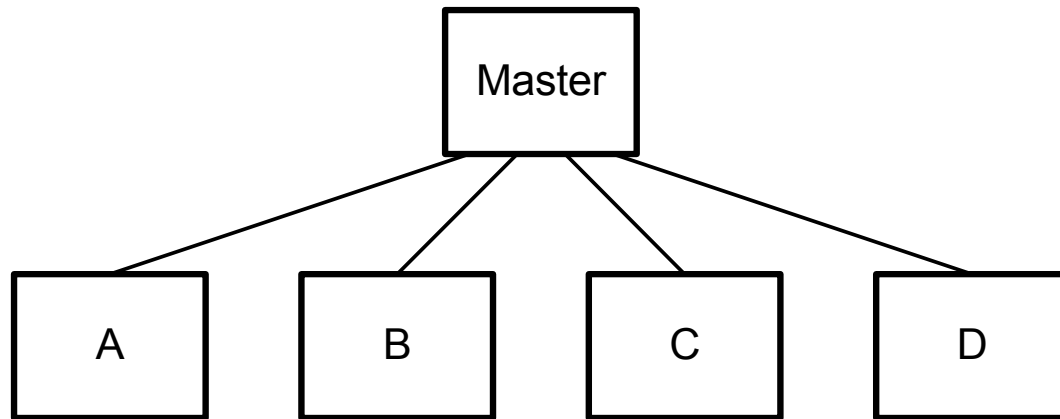
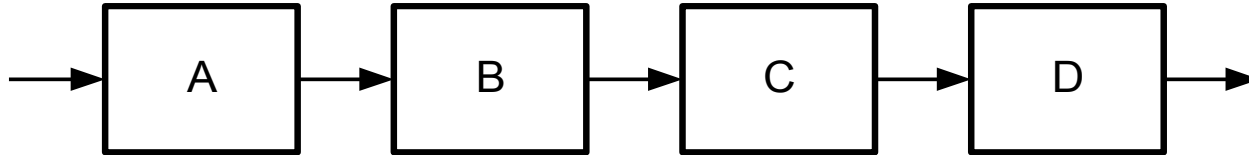
- L11: Chapter 23
- L12: Chapter 15

System Design – a process

- Specification
 - Understand what you need to build
- Divide and conquer
 - Break it down into manageable pieces
- Define interfaces
 - Clearly specify every signal between pieces
 - Hide implementation
 - Choose representations
- Timing and sequencing
 - Overall timing – use a table
 - Timing of each interface – use a simple convention (e.g., valid – ready)
- Add parallelism as needed (pipeline or duplicate units)
- Timing and sequencing (of parallel structures)
- Design each module
- Code
- Verify

Iterate back to the top at any step as needed.

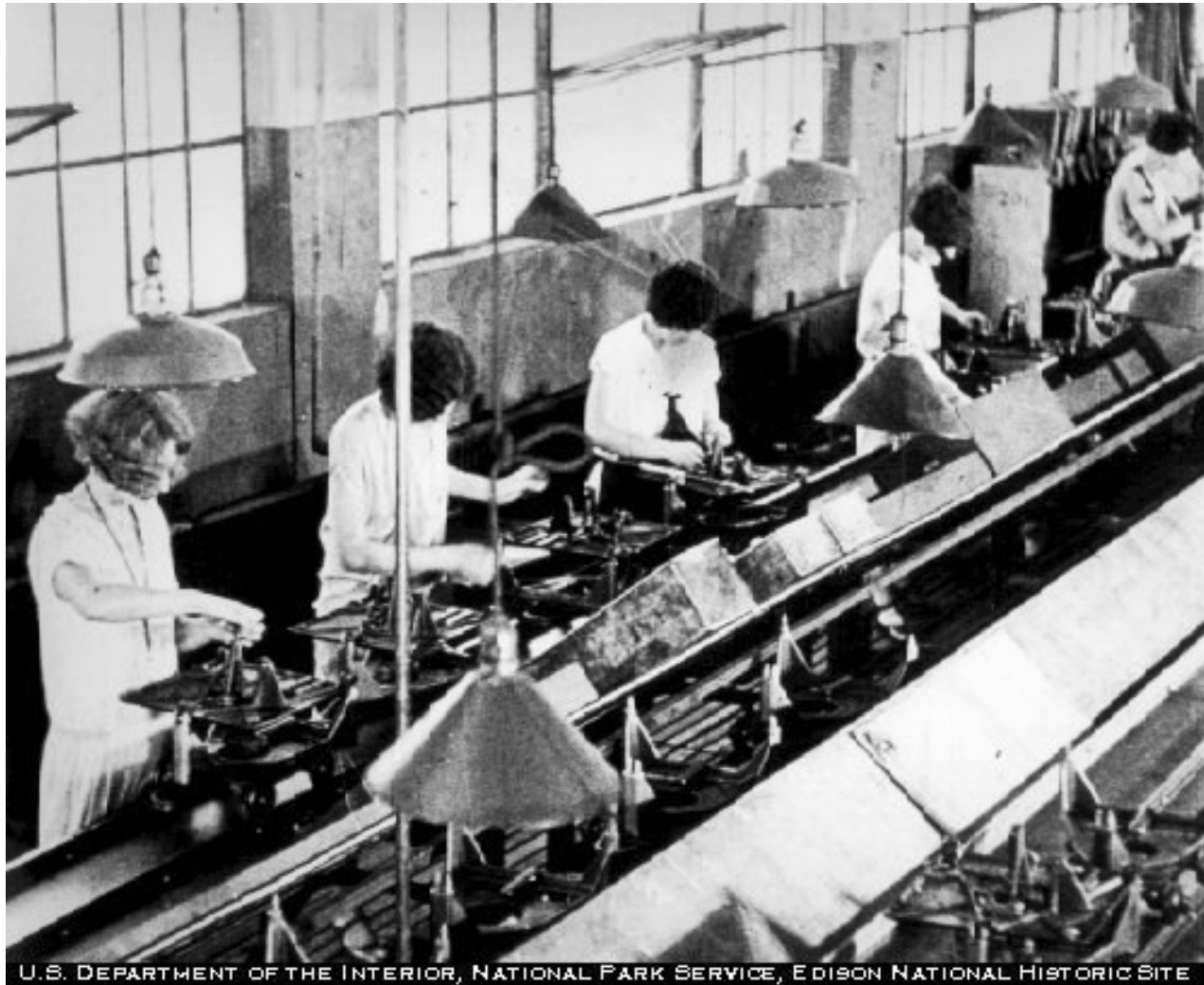
Two ways to make things faster: pipeline and parallel



Pipelines



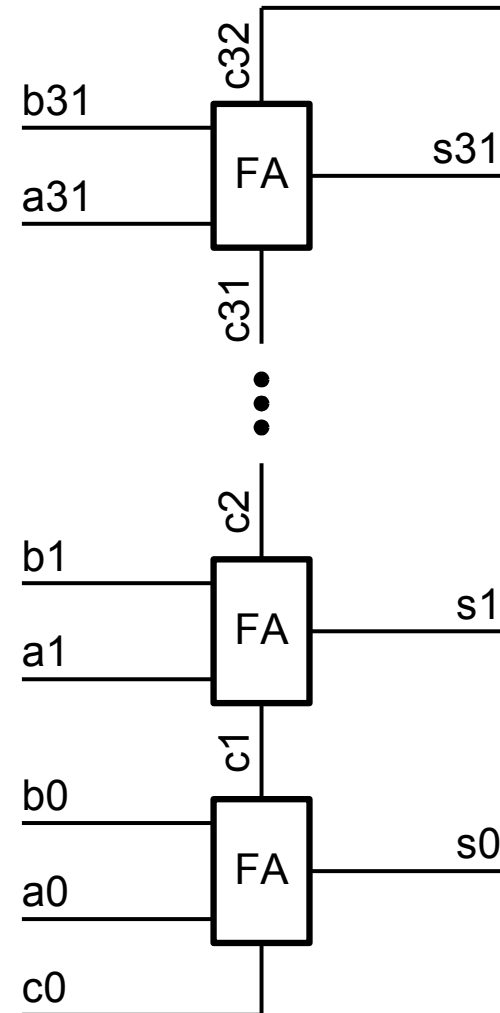
More Like an Assembly Line



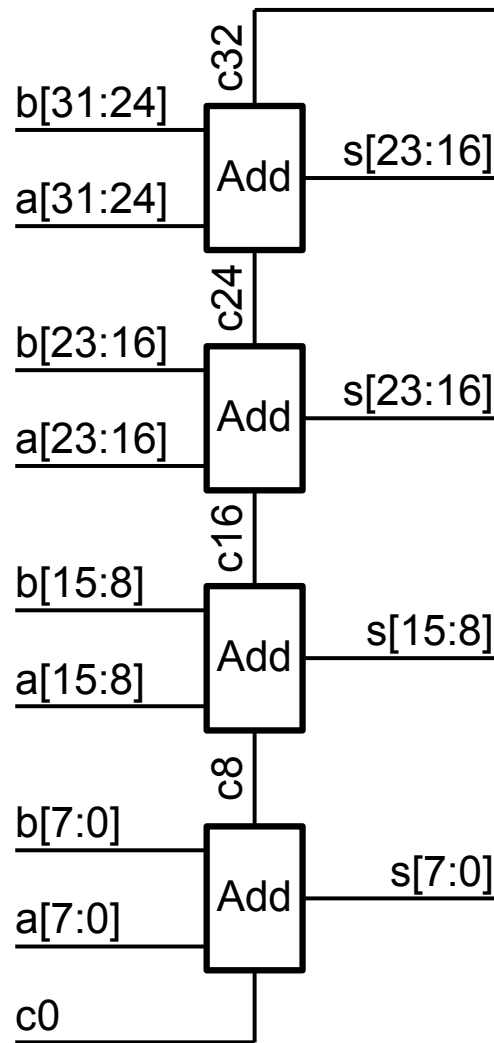
U.S. DEPARTMENT OF THE INTERIOR, NATIONAL PARK SERVICE, EDISON NATIONAL HISTORIC SITE

Pipelines

- Like an assembly line – each pipeline stage does part of the work and passes the ‘workpiece’ to the next stage
- Example 1: Pipelined 32b Adder

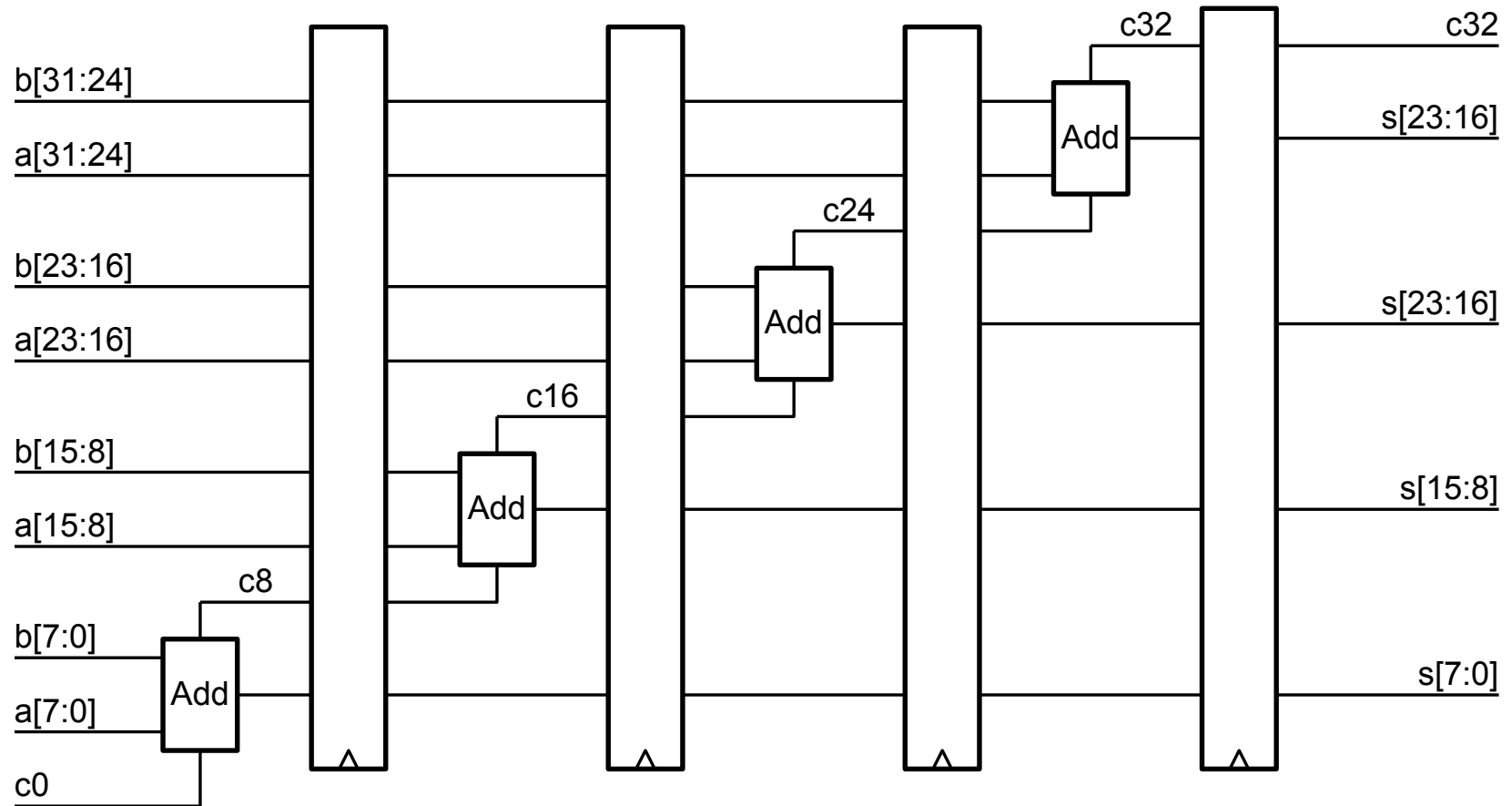


Split into 4 8-bit adders



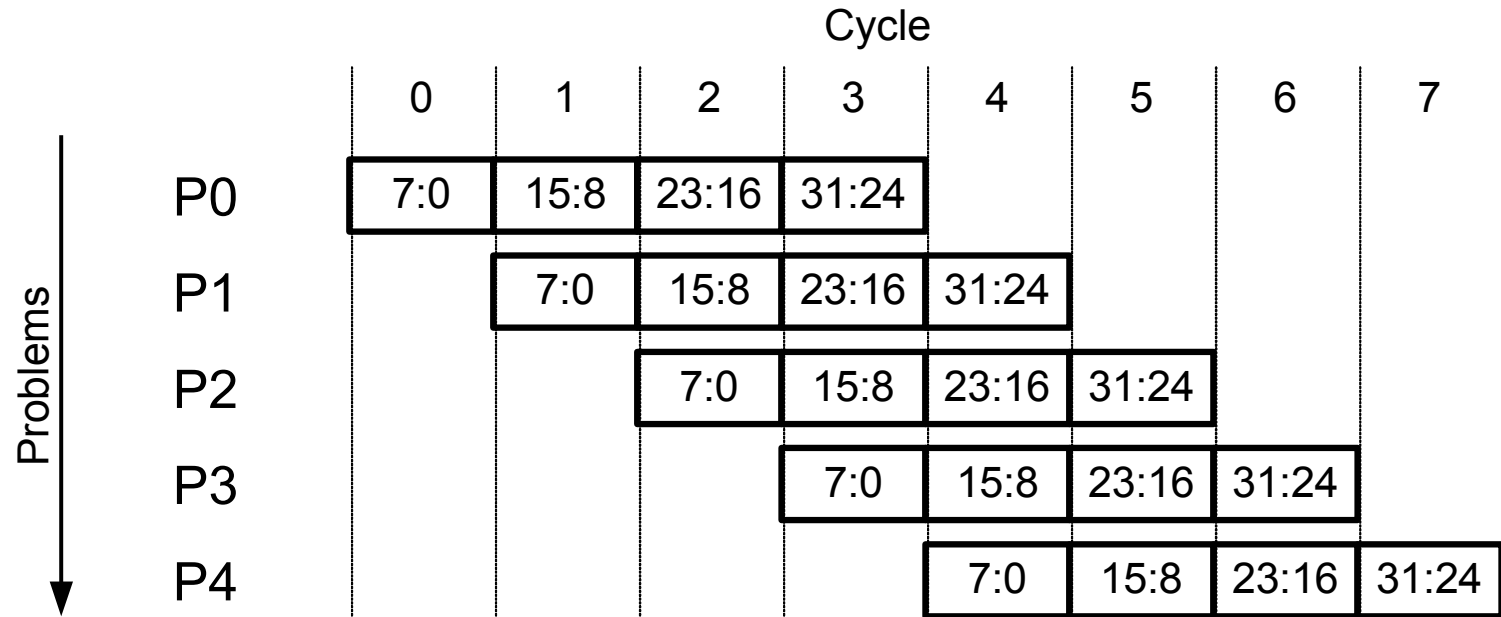
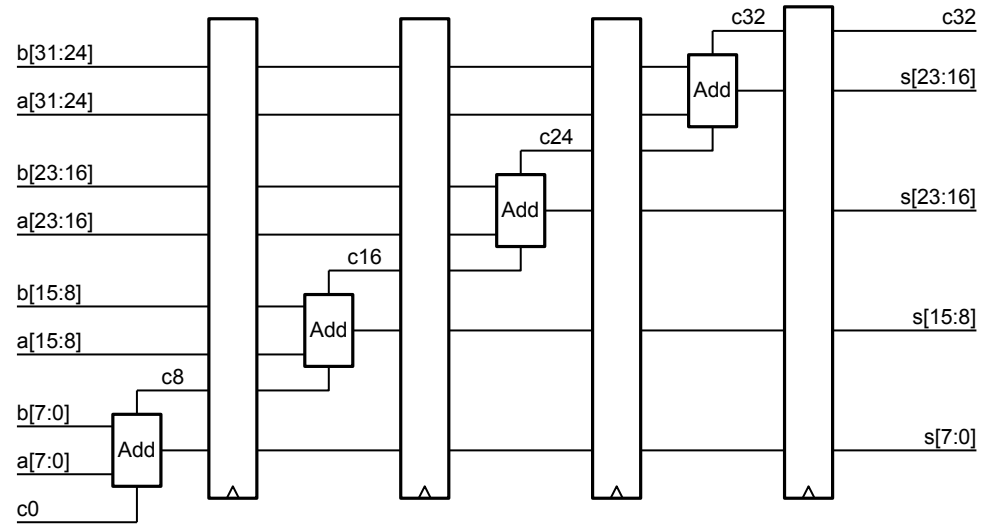
Split into stages

4 problems 'in process' at once

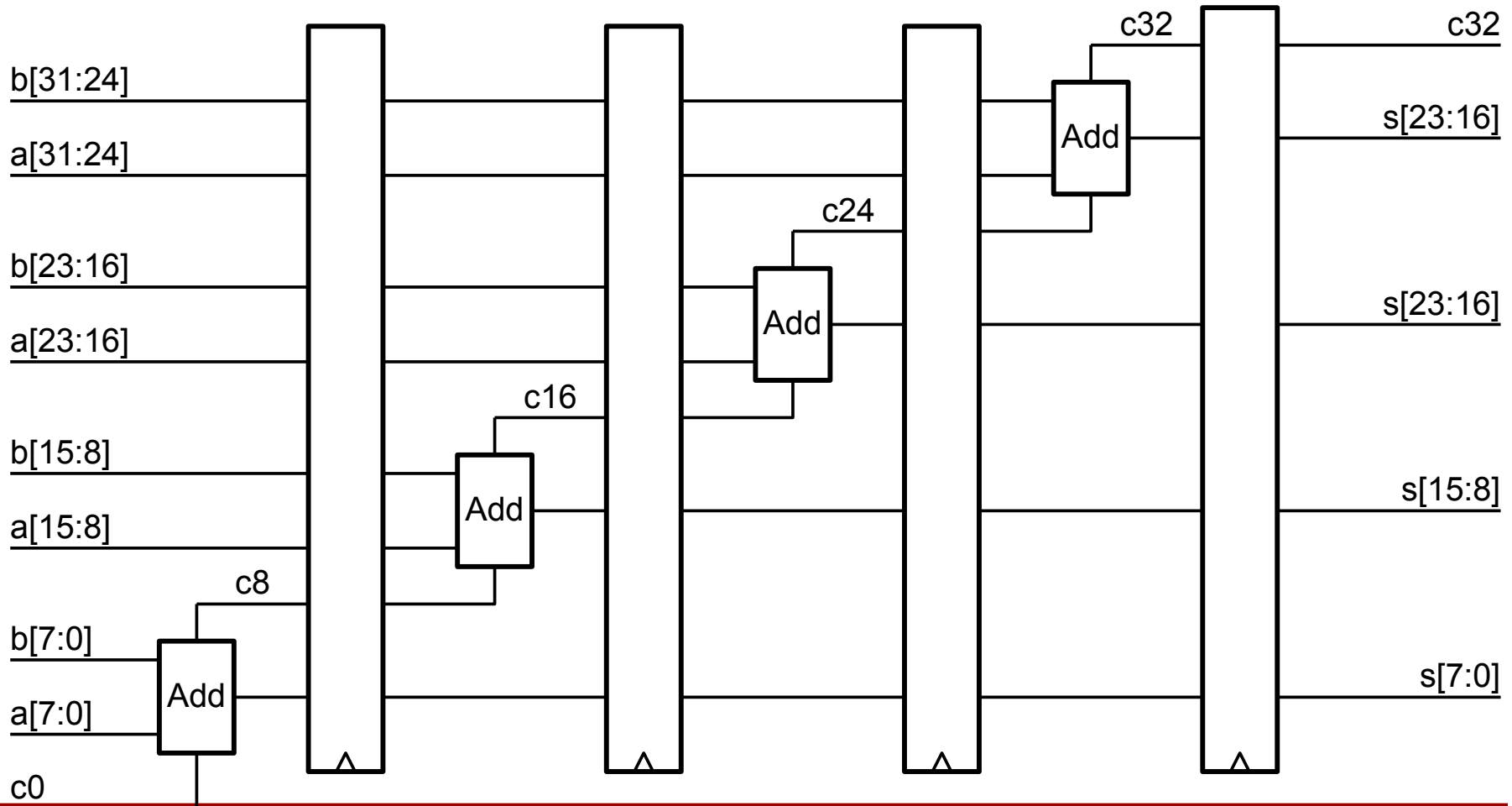


Pipeline Diagram

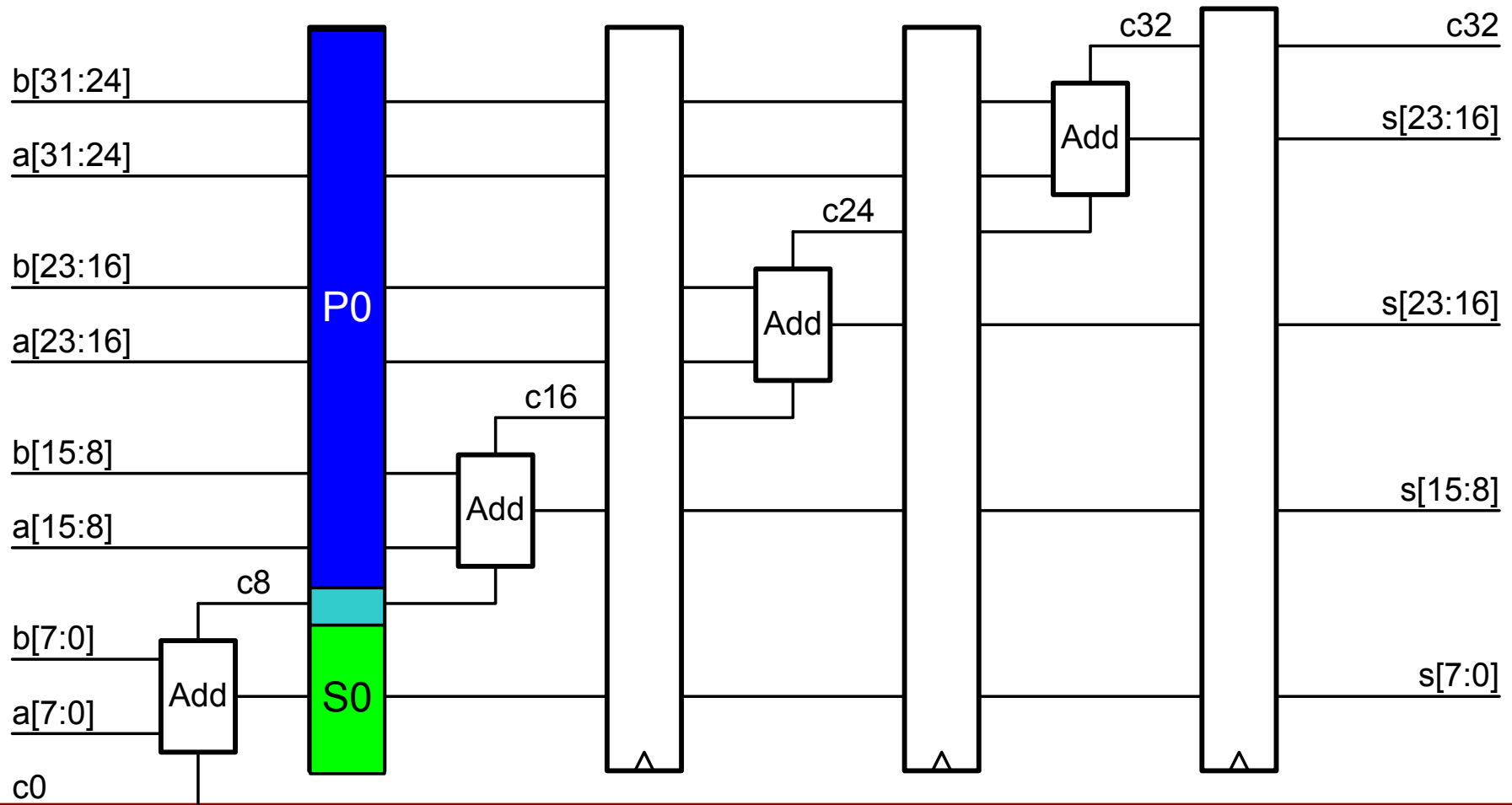
Illustrates pipeline timing



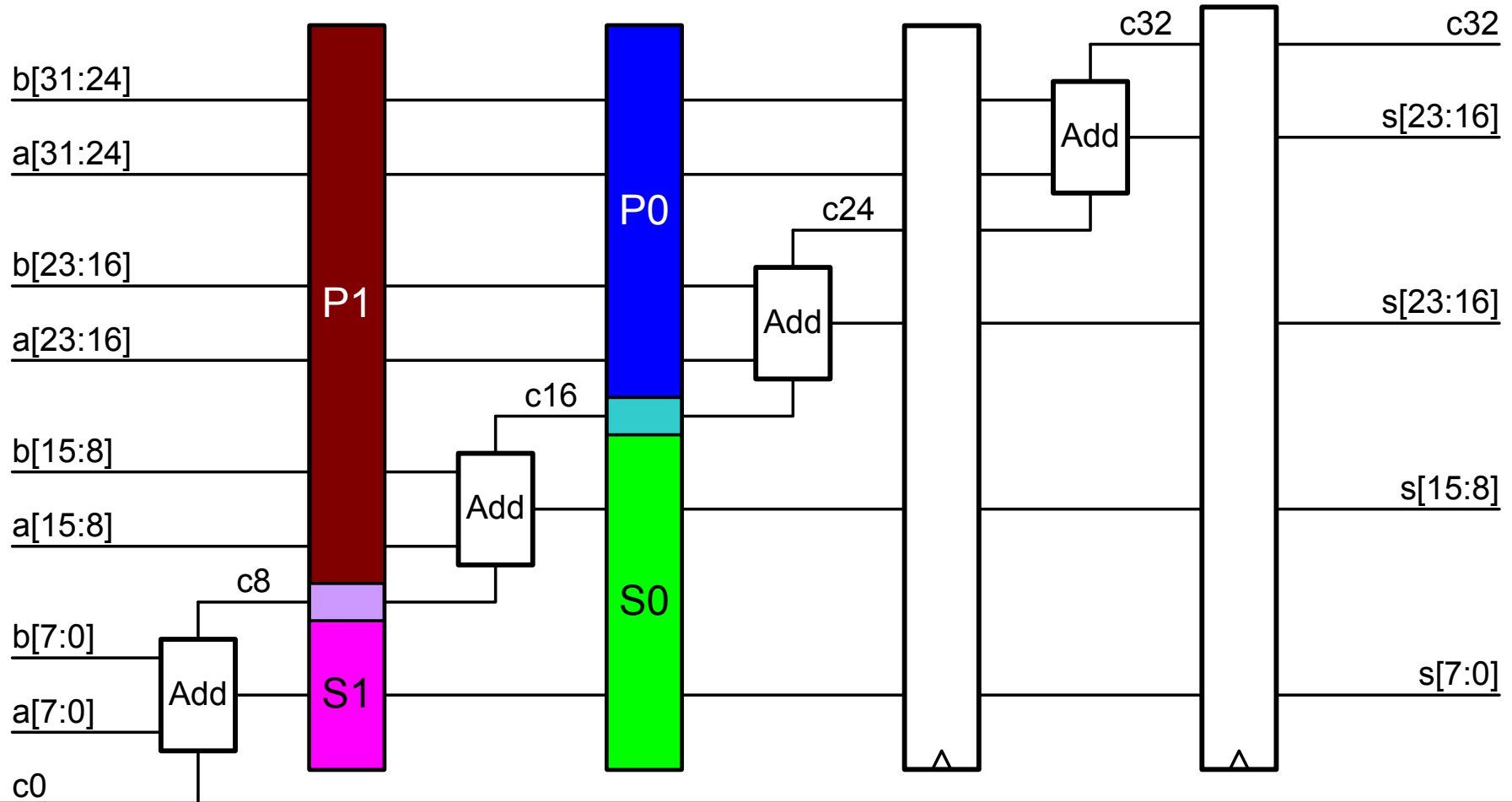
Movie



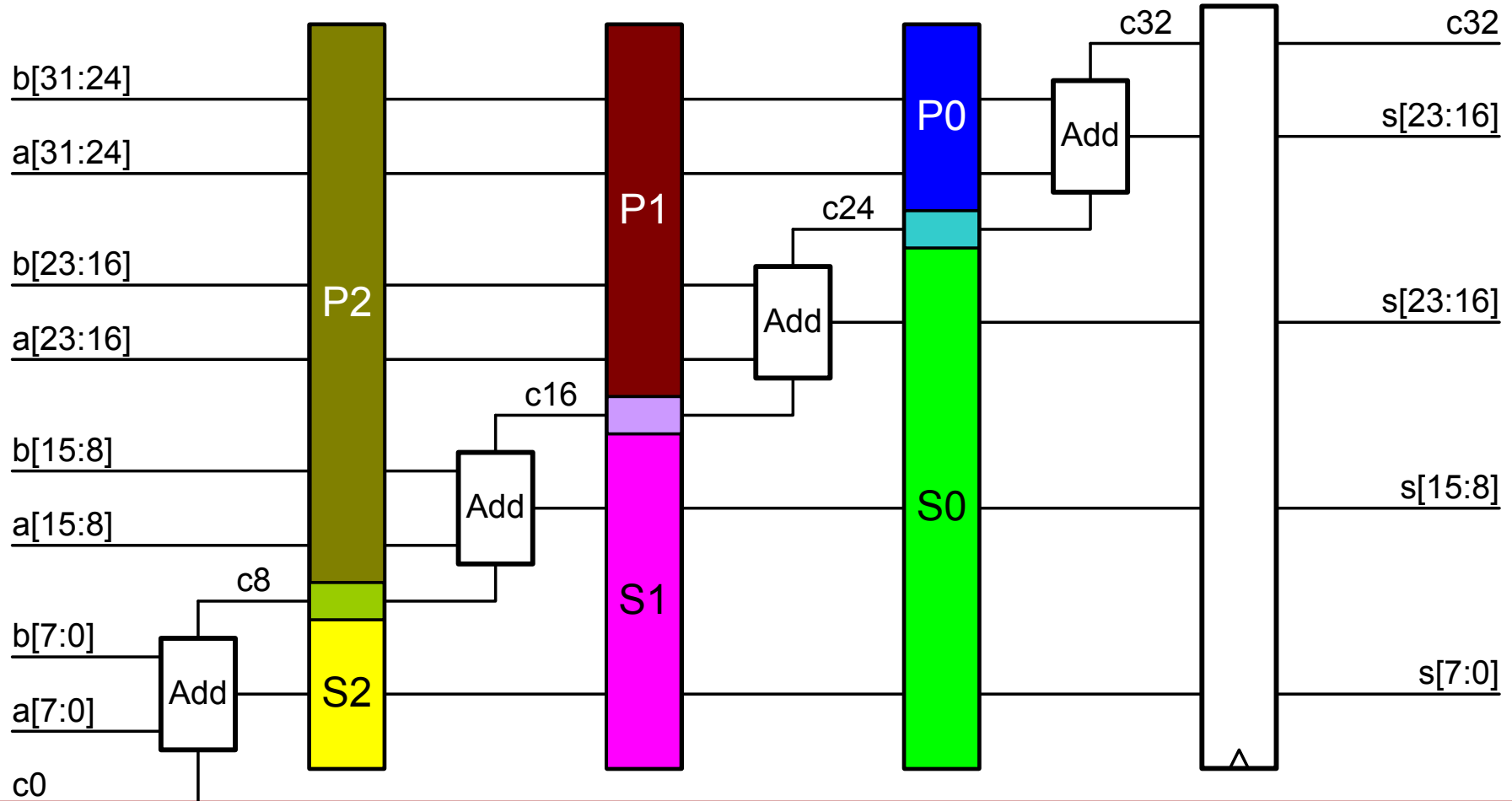
Cycle 1



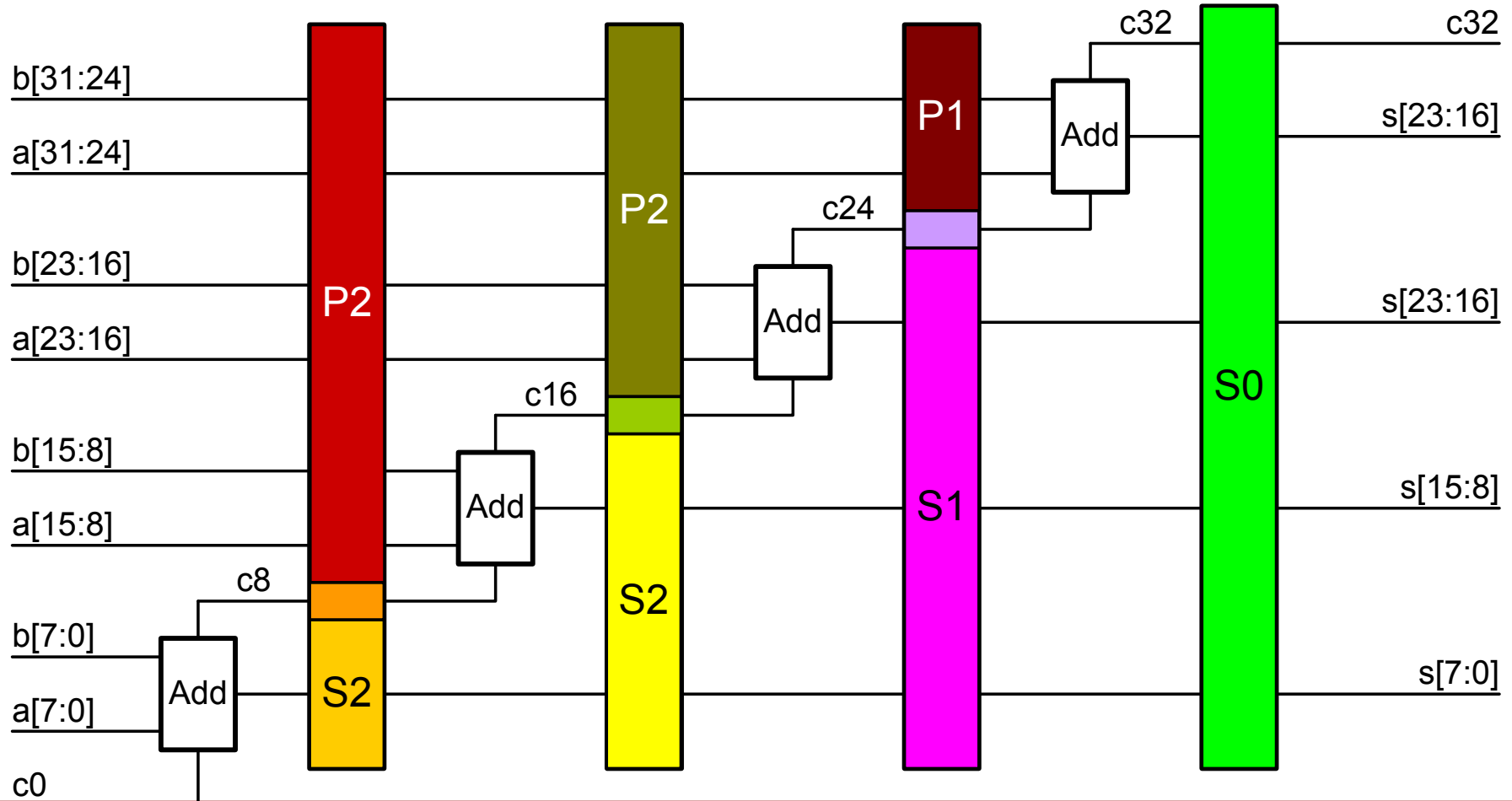
Cycle 2



Cycle 3



Cycle 3



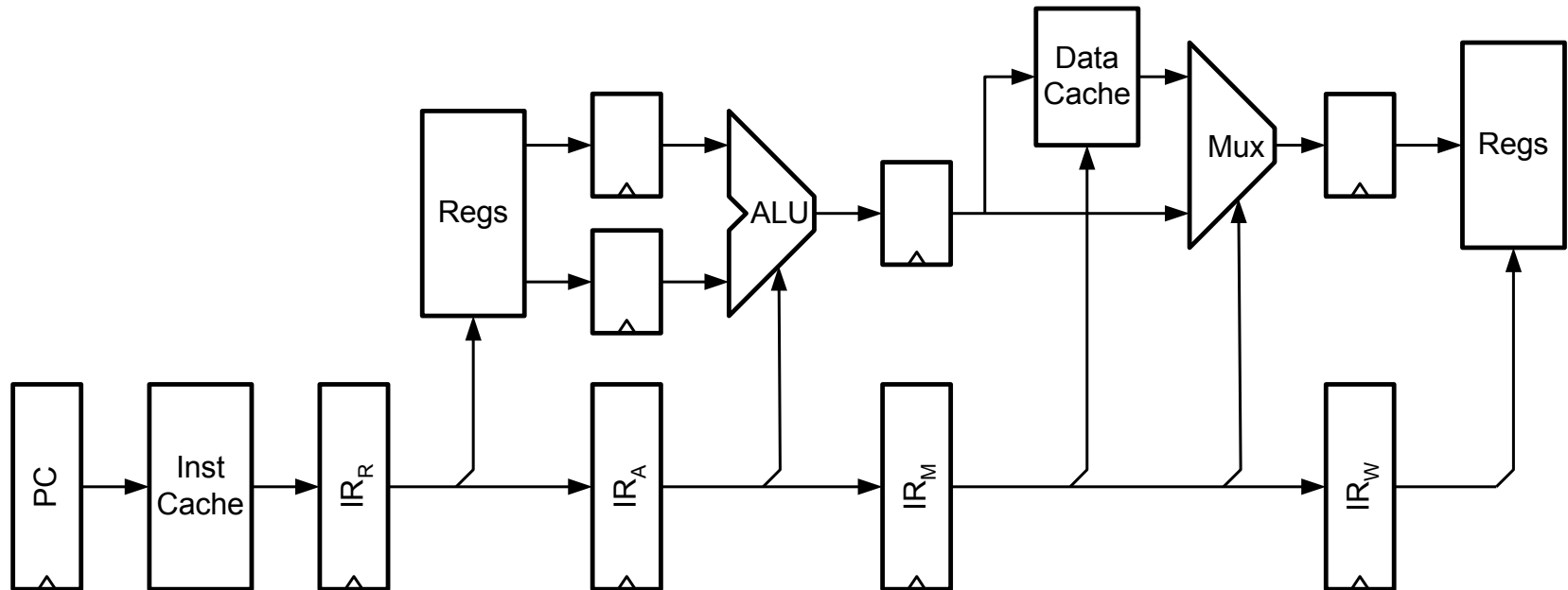
Latency and Throughput of a Pipeline

- Suppose before pipelining the delay of our 32b adder is 3200ps (100ps per bit) and this adder can do one problem each 3200ps for a *throughput* of $1/3200\text{ps} = 312\text{Mops}$
- What is the delay (latency) and throughput of the adder with pipelining?

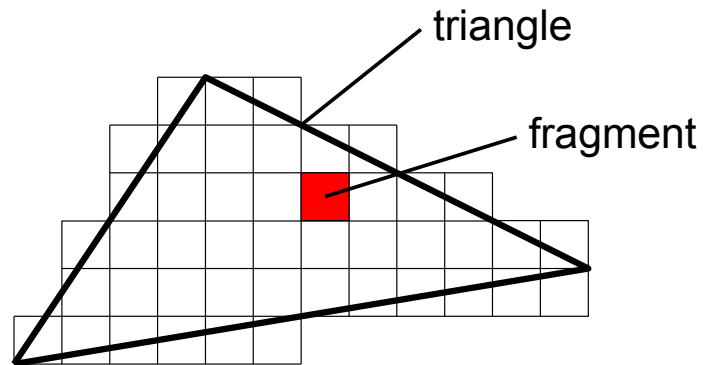
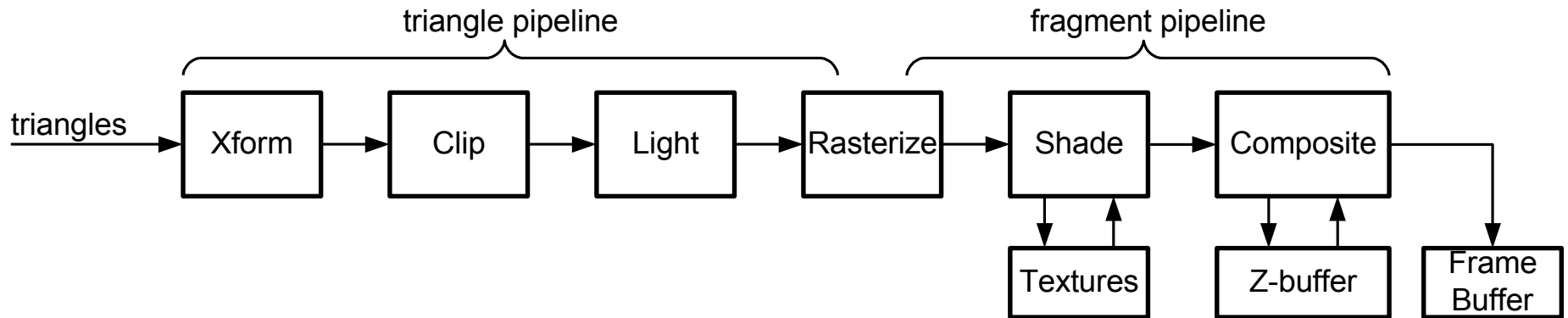
Suppose $t_{\text{dCQ}} = 100\text{ps}$, $t_s = 50\text{ps}$, $t_k = 50\text{ps}$ (200ps Overhead)

- $t_{\text{pipe}} = n(t_{\text{stage}} + t_{\text{dCQ}} + t_s + t_k)$ $4(1000) = 4000$
- $\Theta = n/t_{\text{pipe}} = 1/(t_{\text{stage}} + t_{\text{dCQ}} + t_s + t_k)$ $4/4000 = 1\text{Gops}$

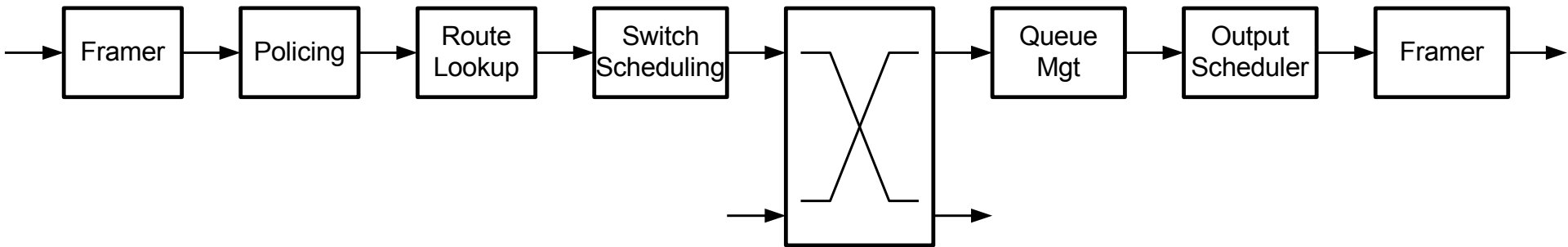
Example 2: Processor Pipeline



- Example 3: Graphics rendering pipeline



Example 4 – Packet Processing Pipeline



And each of these modules is internally pipelined

You get the idea. Lots of systems are organized this way.

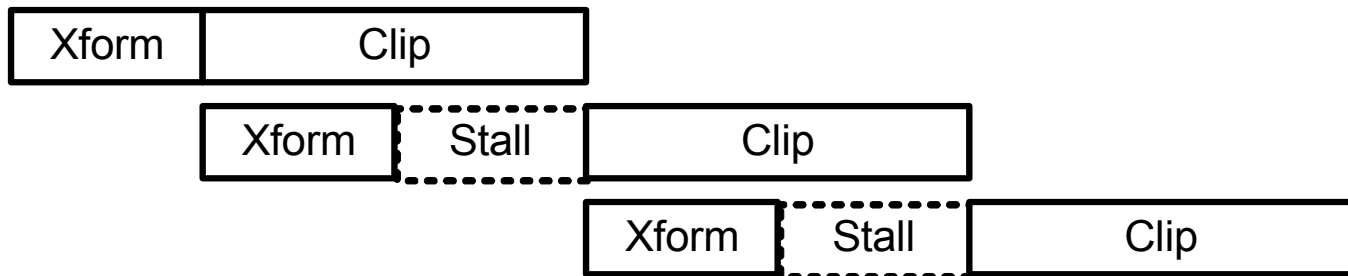
Issues with pipelines

(all deal with time per stage)

- Load balance (across stages)
 - one stage takes longer to process each input than the others – becomes a ‘bottleneck’
 - Example
 - Rasterizing an ‘average’ triangle in a graphics pipeline takes more time than ‘lighting’ its vertices.
- Variable load (across data)
 - A given stage takes more time on some inputs than others
 - Example
 - The the time needed to rasterize a triangle is proportional to the number of fragments in the triangle. The average triangle may contain 20 fragments, but triangles range from 0 to over 1M
- Long latency
 - A stage may require a long latency operation (e.g., texture access)

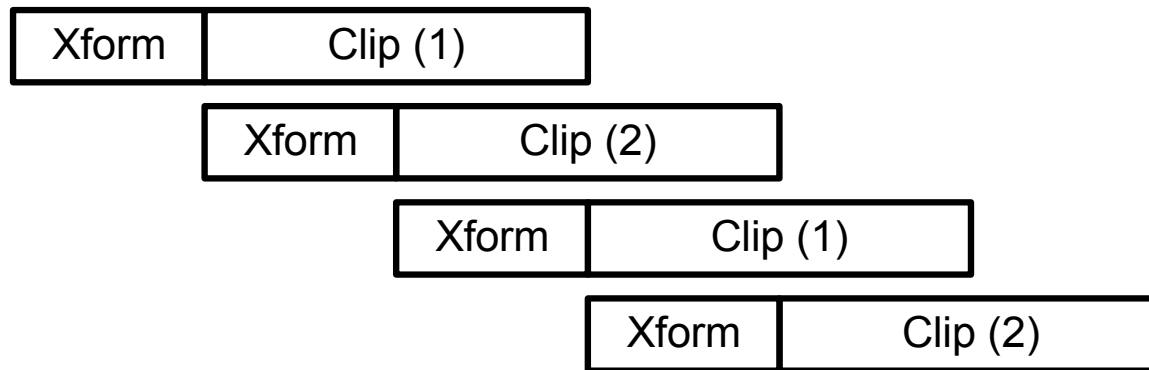
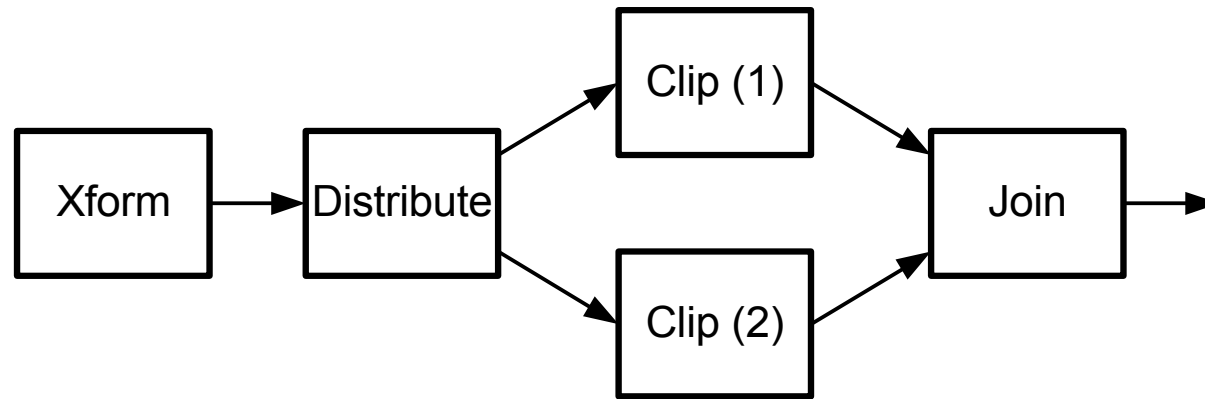
Load Balancing Pipelines

- Suppose transform takes 2 cycles and clip 4 cycles
- Clip is a 'bottleneck' pipeline stage
- Xform unit is busy only half the time



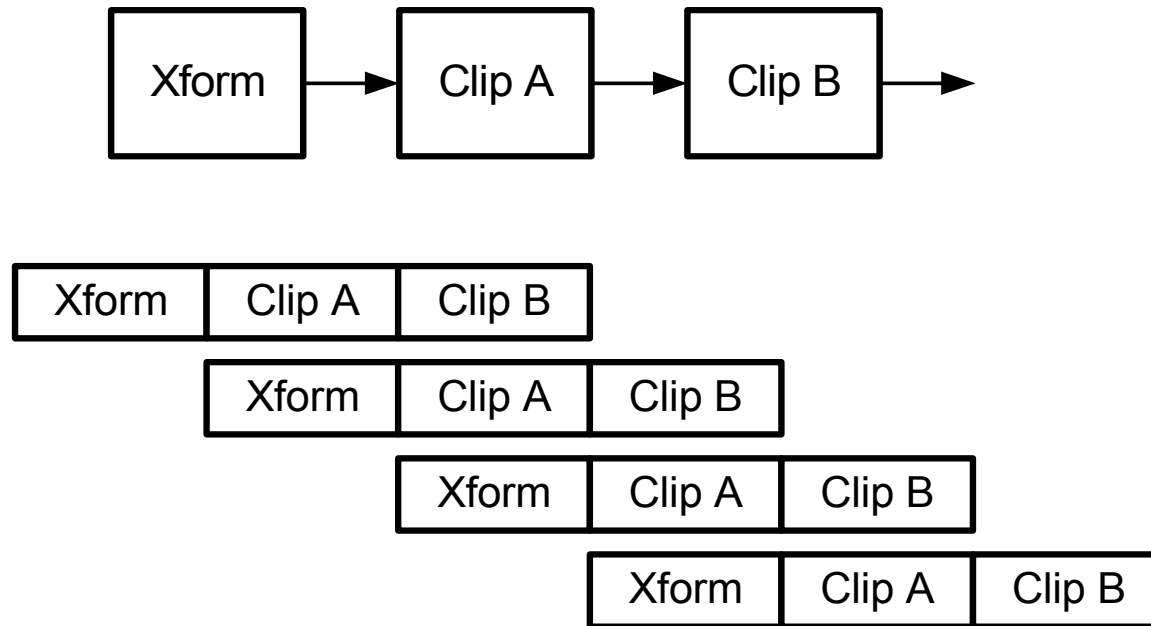
Load Balancing Solutions

1 – Parallel copies of slow unit

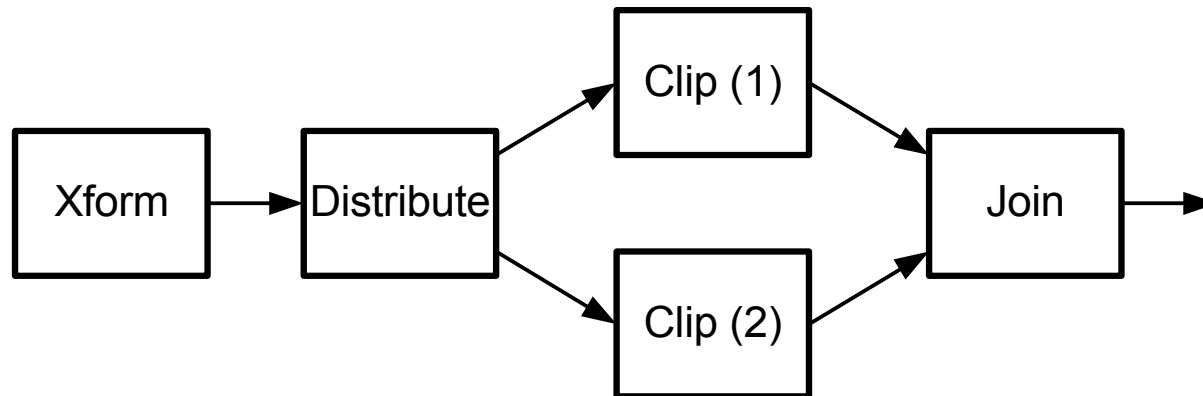
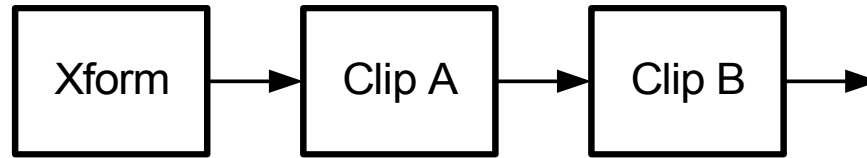


Load Balancing Solutions

2 – Split slow pipeline stage



When is it better to split? To copy?
Throughput and latency are the same

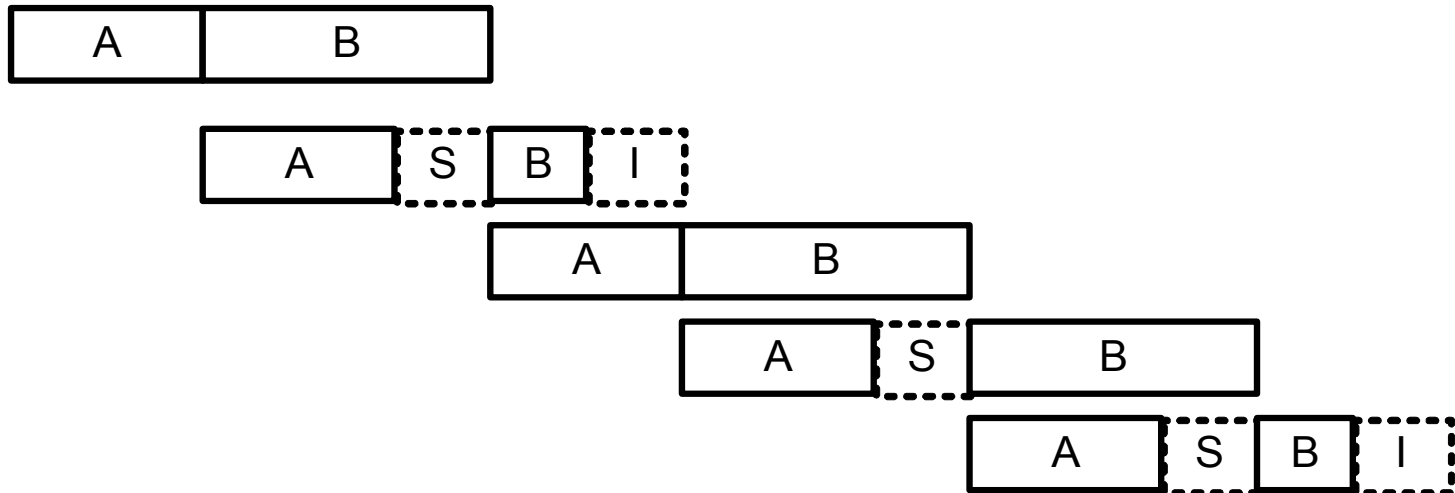
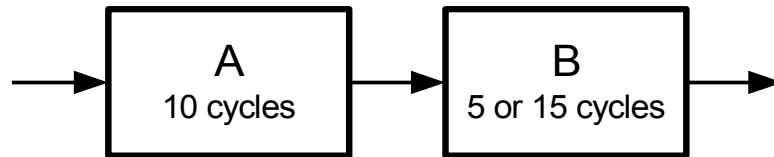


Variable load

Stage A always takes 10 cycles.

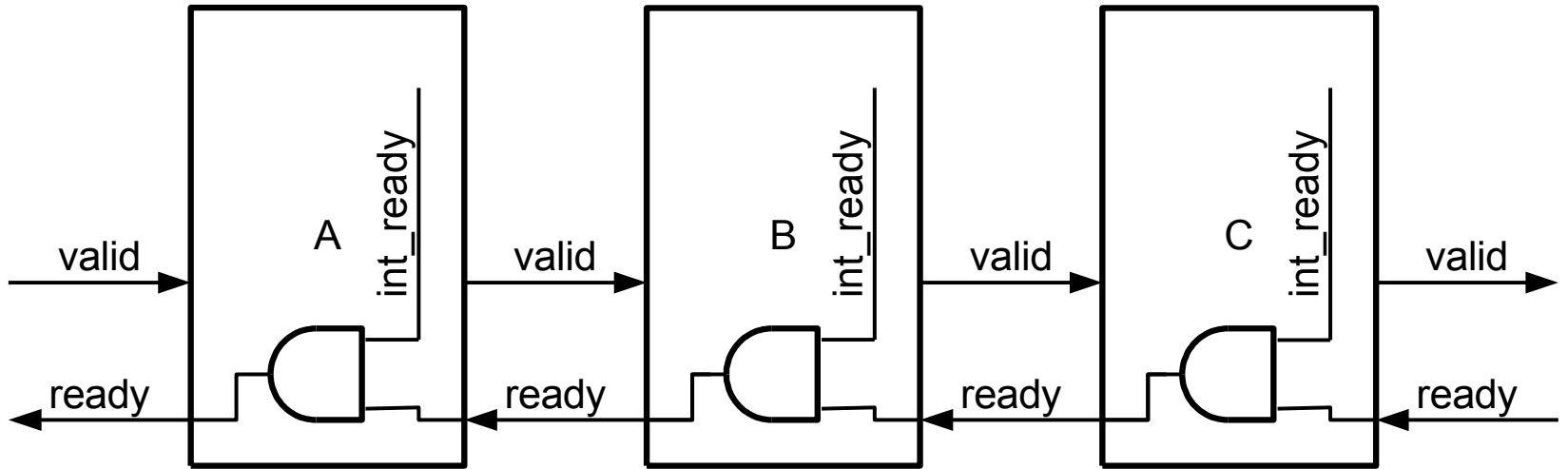
Stage B takes 5 or 15 cycles – averages 10 cycles

Pipeline averages ____ cycles per element



Stalling a *rigid* pipeline

A stall in any stage halts *all* stages upstream of the stall point *instantly* (on the next clock)

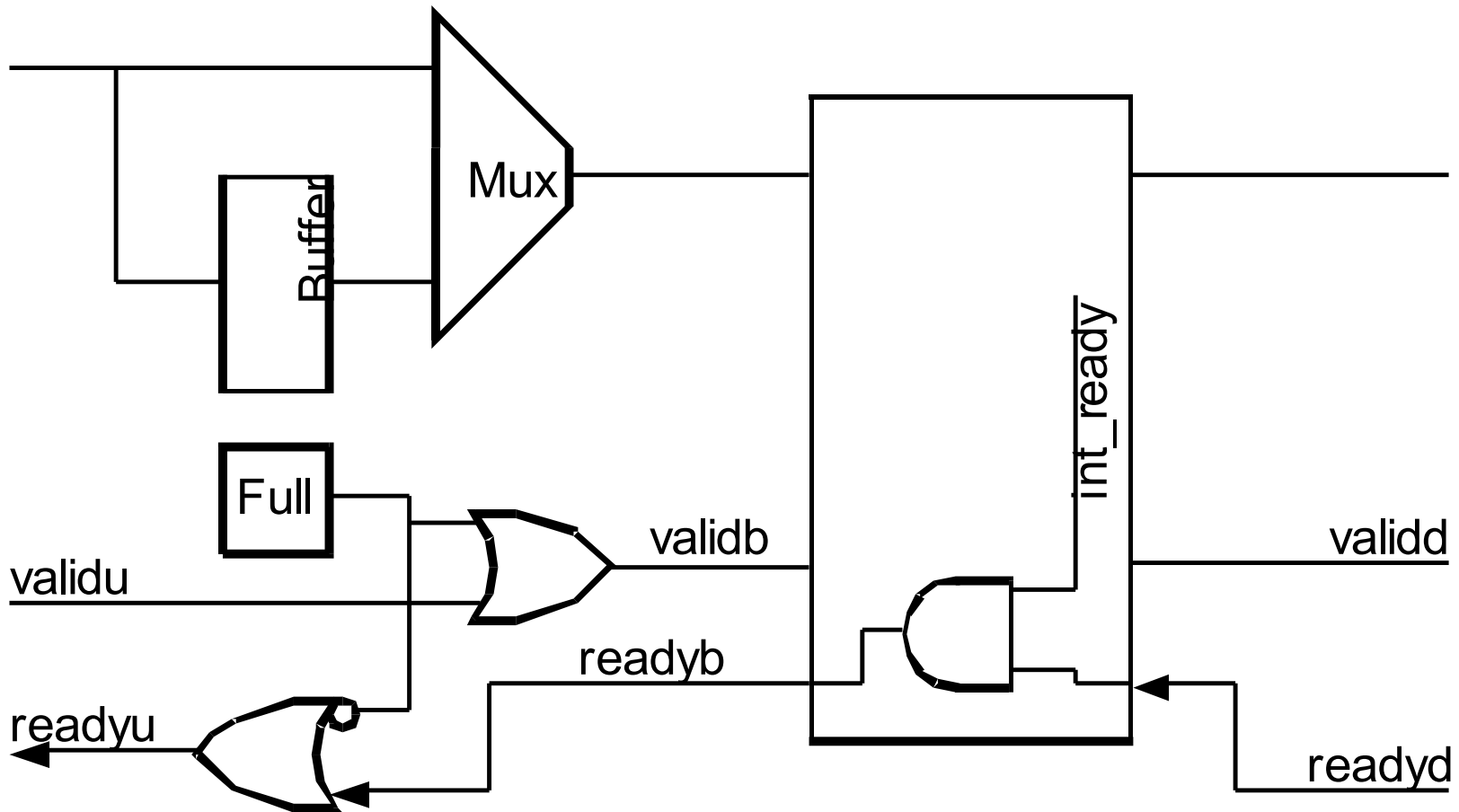


What if we stopped all stages, not just upstream stages?

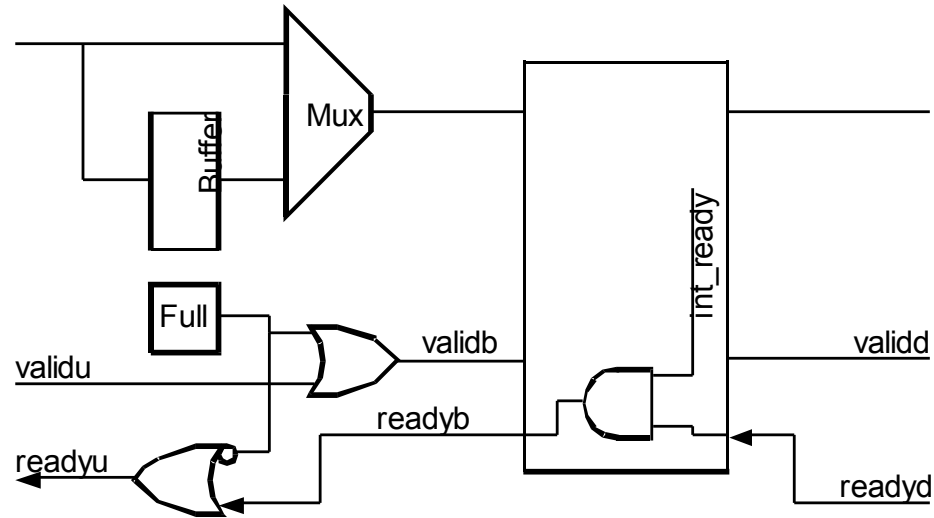
How does the delay of this structure *scale* with the number of stages?

Double Buffer

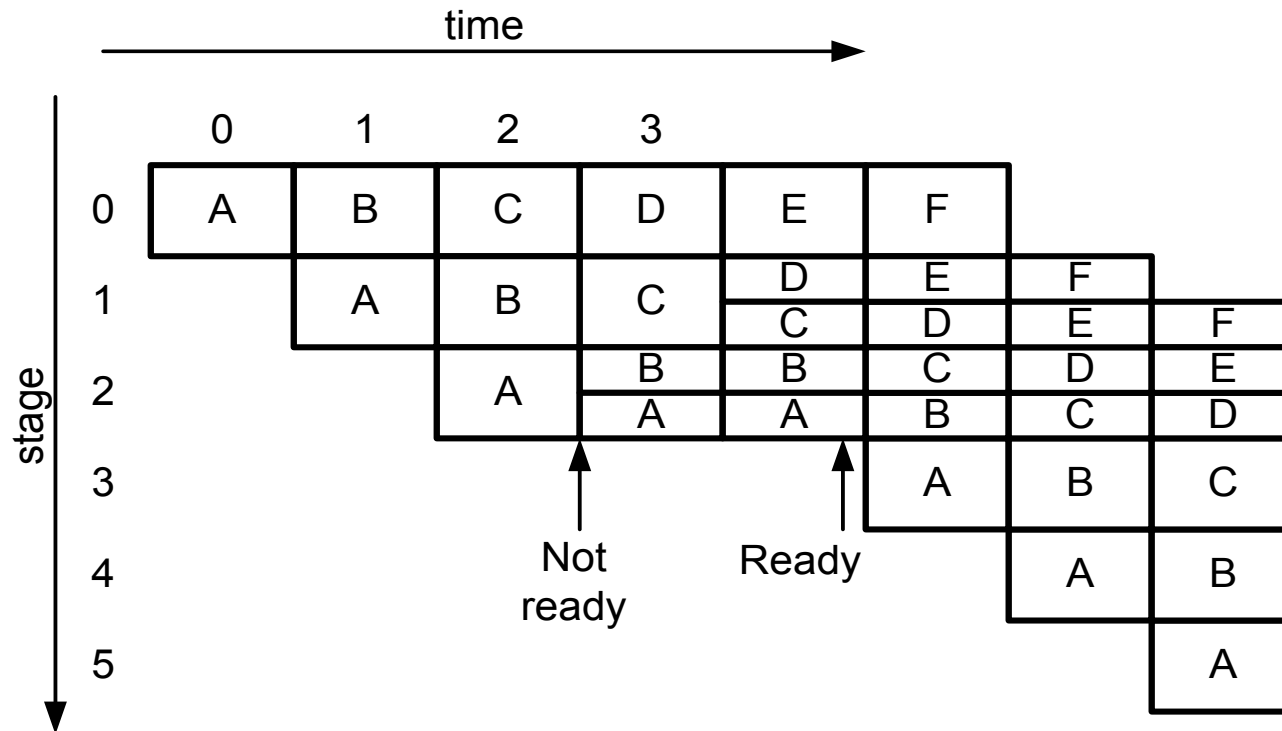
Add an extra buffer to each stage that is filled during the first cycle of a stall.



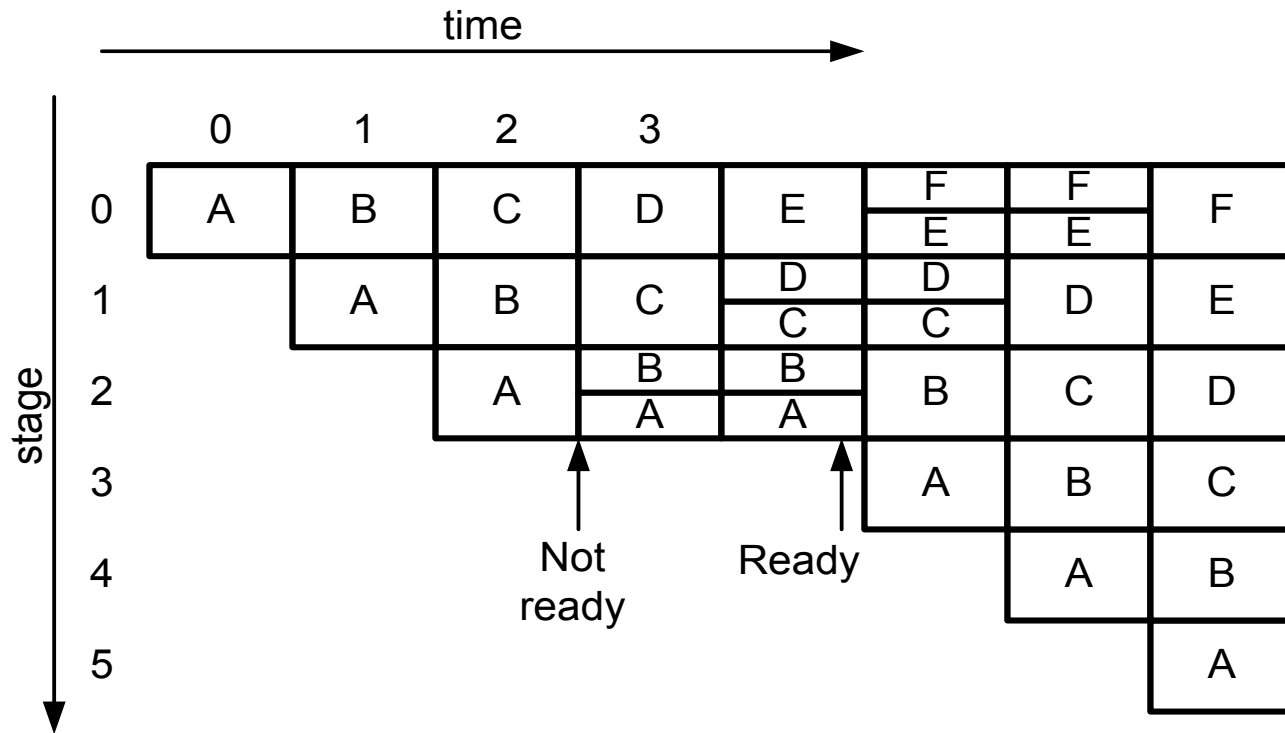
What is the logic equation for “next_full”? “next_buf”? “mux_sel”?



Double Buffer Timing



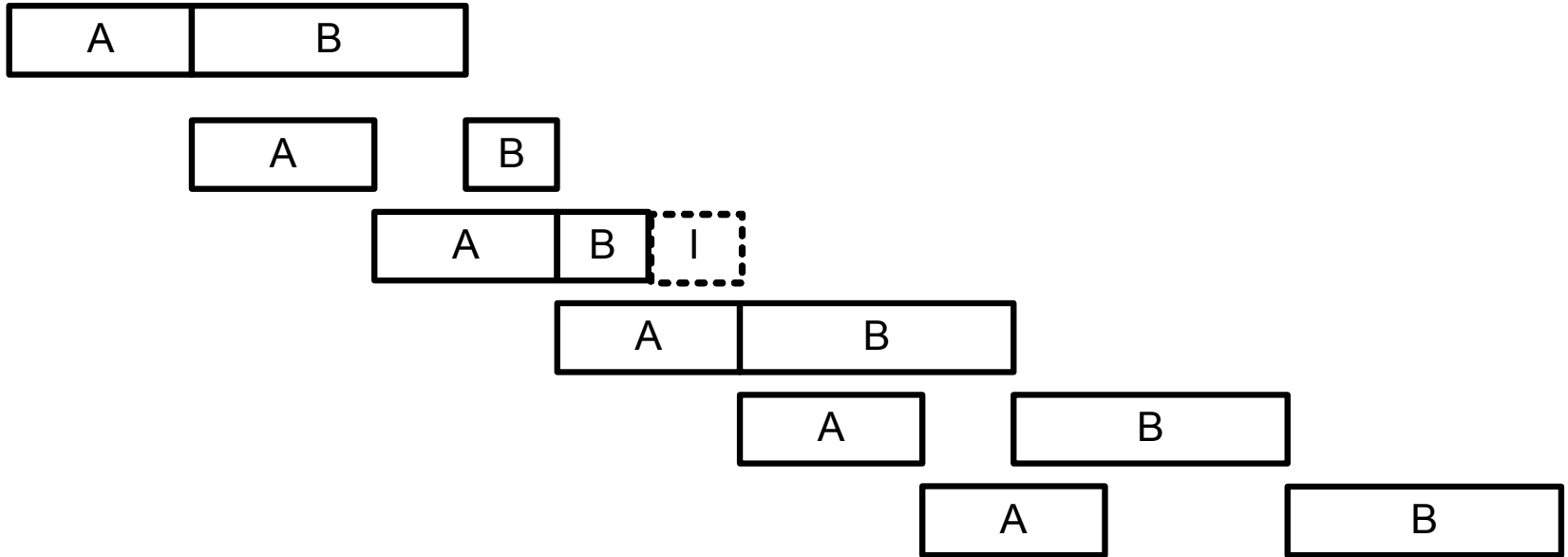
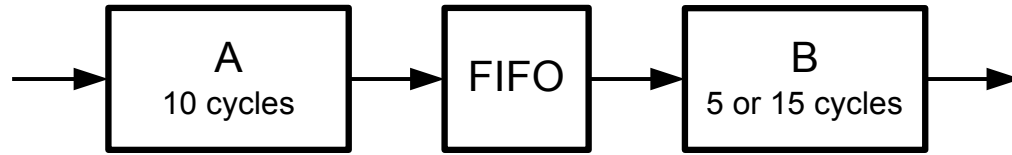
Double Buffer Alternate Timing (how do you make this happen?)



Elastic Pipelines

A FIFO between stages decouples timing

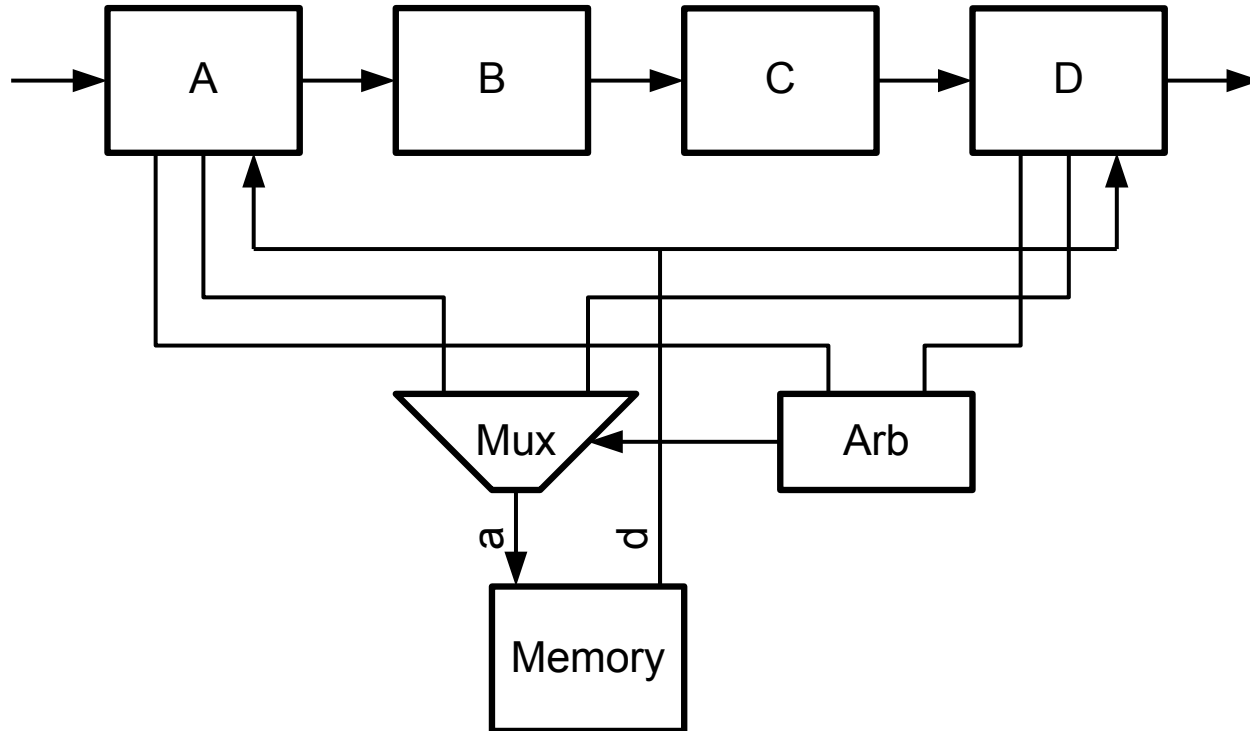
Allows stages to operate at their 'average' speed



Resource Sharing

Suppose two pipeline stages need to access the same memory

-



How would you set the priority on the arbiter?

Pipeline overview

- Divide large problem into *stages* assembly-line style
- Divide evenly or *load imbalance* will occur
 - Fix by splitting or copying *bottleneck* stage
- Rigid pipelines have no extra storage between stages
 - A *stall* on any stage halts all *upstream* stages
 - Hard to stop 100 stages at once
 - Make this scalable with double-buffering
- Variable load results in *stalls* and *idle* cycles on a rigid pipeline
 - Make pipeline *elastic* by adding FIFOs between key stages