

# Digital Circuits and Systems

## Lecture 7 Pipeline and Parallel

---

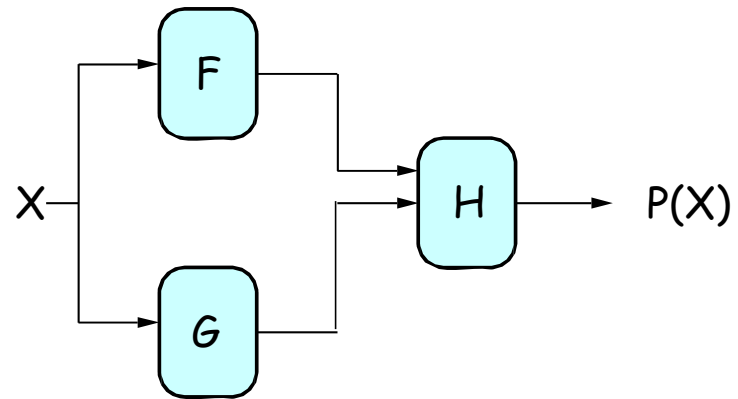
Tian Sheuan Chang

想要變得更快，要怎麼做呢？

# Outline [Dally. Ch. 23]

- 如何設計一個系統
- 讓設計變快一點: 增加一點平行度 (pipeline or duplicate unit)

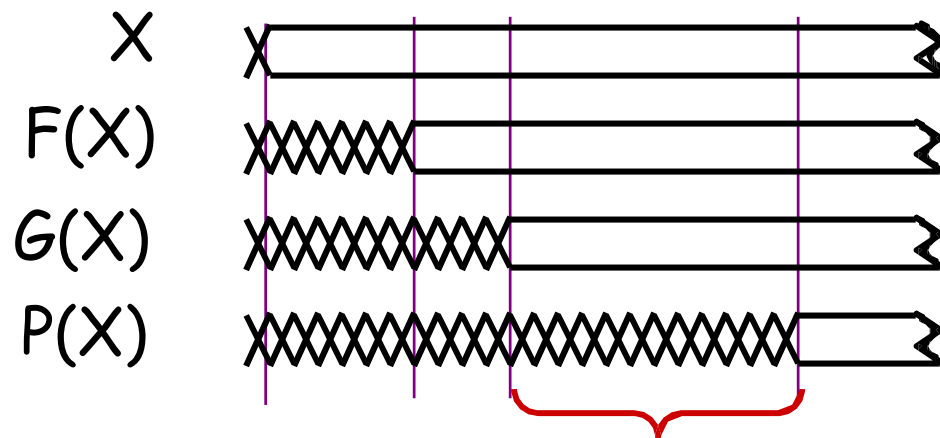
# Performance of Combinational Circuits



For combinational logic:

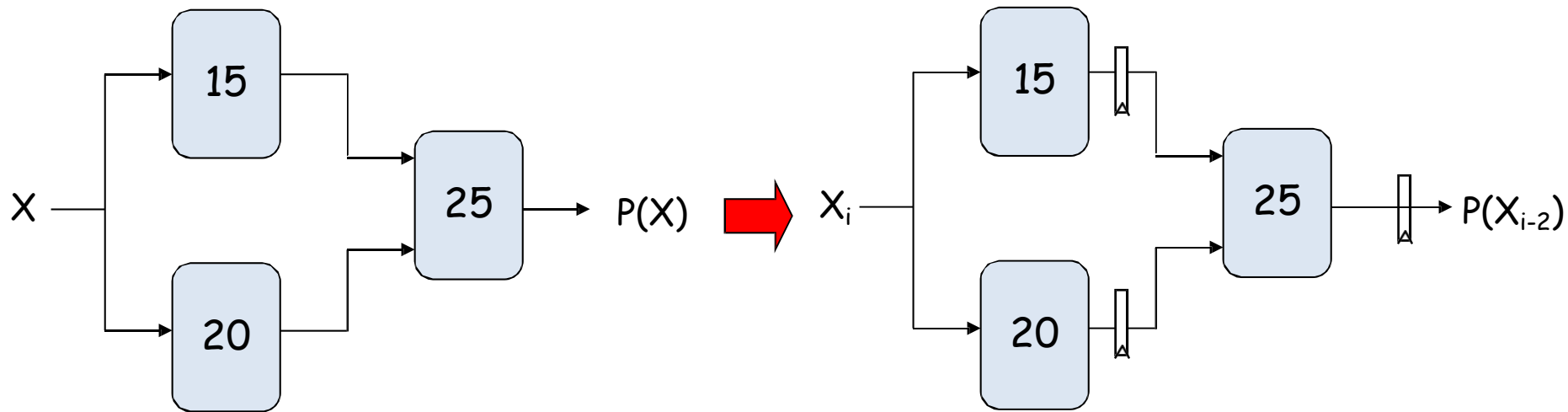
$$L = t_{PD},$$
$$T = 1/t_{PD}.$$

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

# Retiming Combinational Circuits aka “Pipelining”



$$L = 45$$
$$T = 1/45$$

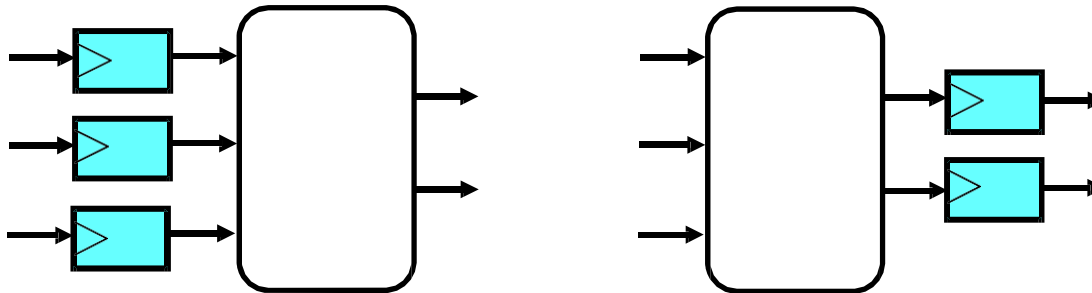
Assuming ideal registers:  
i.e.,  $t_{PD} = 0$ ,  $t_{SETUP} = 0$

$$\rightarrow t_{CLK} = 25$$
$$L = 2 * t_{CLK} = 50$$
$$T = 1/t_{CLK} = \mathbf{1/25}$$

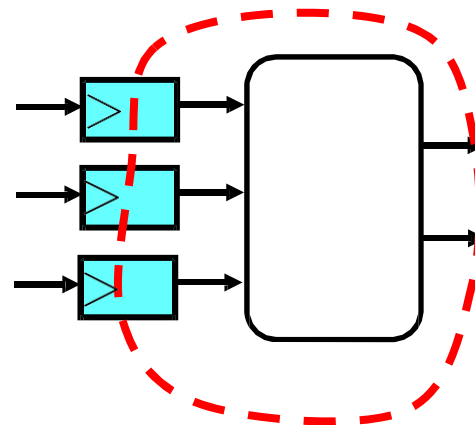
# Retiming (Optional)

Retiming is the action of moving registers around in the system

- Registers have to be moved from ALL inputs to ALL outputs or vice versa



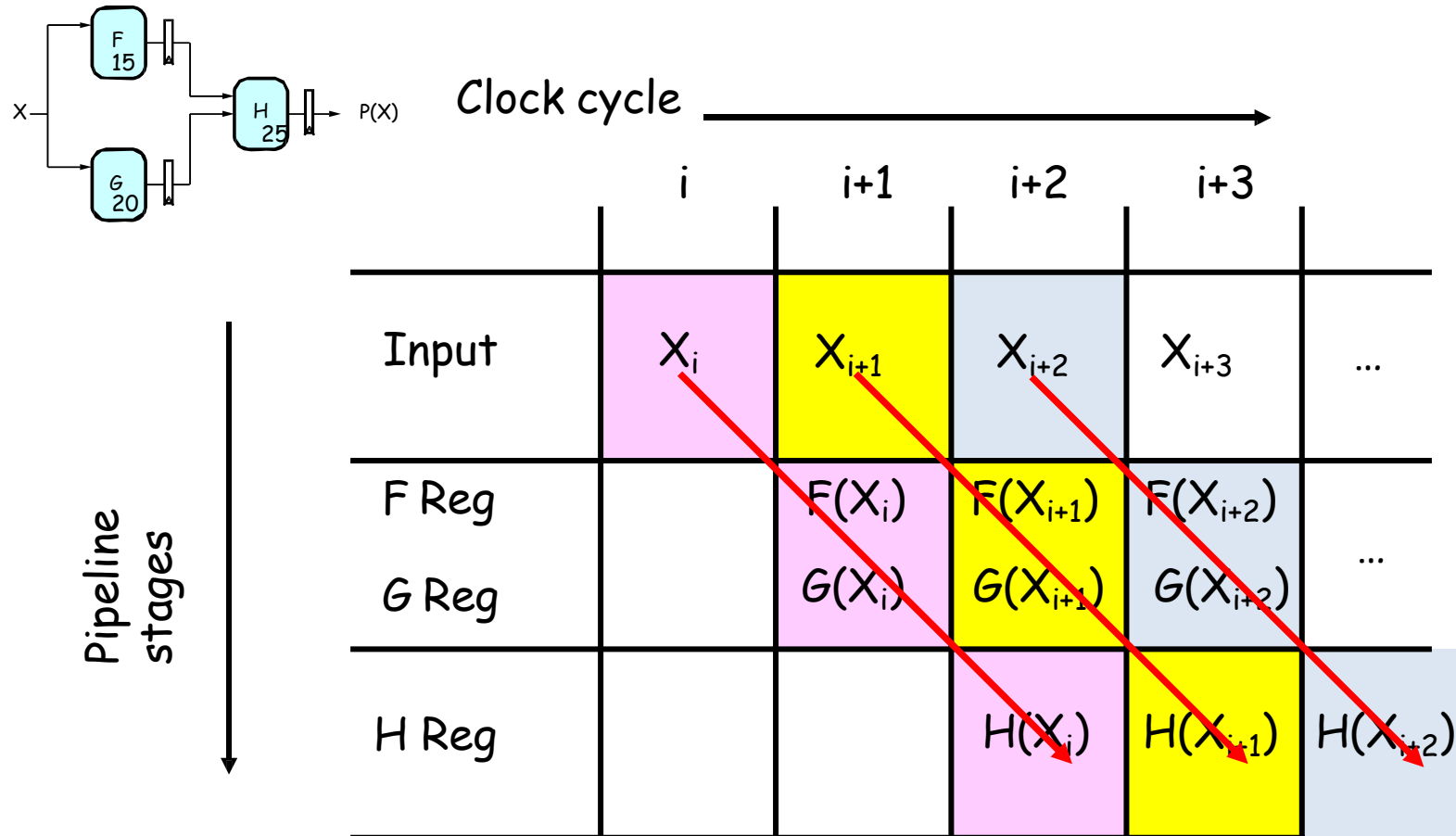
**Cutset retiming:** A cutset intersects the edges, such that this would result in two disjoint partitions of the edges being cut. To retime, delays are moved from the ingoing to the outgoing edges or vice versa.



**Benefits of retiming:**

- Modify critical path delay
- Reduce total number of registers

# 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.

# Pipeline Conventions

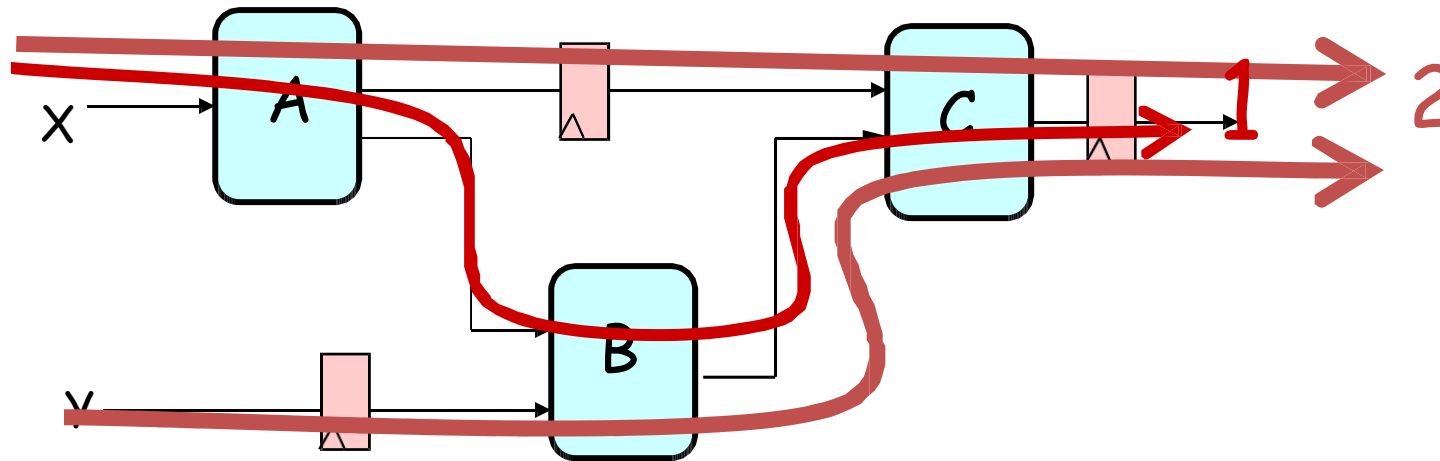
- DEFINITION:
  - a **K-Stage Pipeline** (“K-pipeline”) is an acyclic circuit having exactly K registers on every path from an input to an output.
  - a COMBINATIONAL CIRCUIT is thus an 0-stage pipeline.
- CONVENTION:
  - Every pipeline stage, hence every K-Stage pipeline, has a register on its OUTPUT (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? \_\_\_\_\_ none

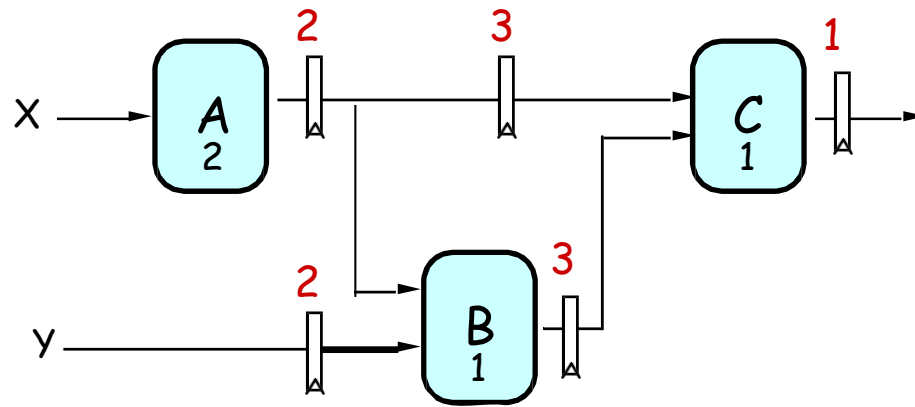
Problem:

*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!



# Pipeline Example

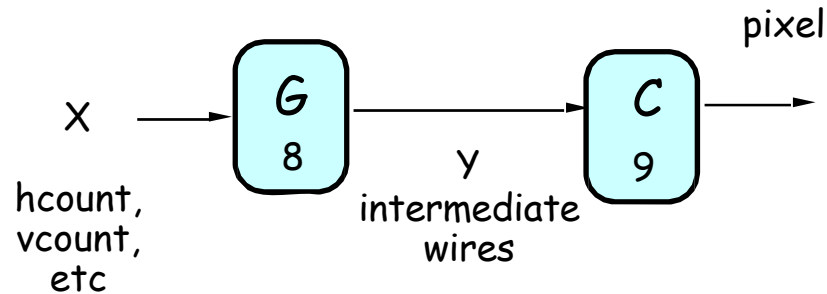


## OBSERVATIONS:

- 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:	4	1/4
1-pipe:	4	1/4
2-pipe:	4	1/2
3-pipe:	6	1/2

# Pipeline Example - Verilog



- $G$  = game logic 8ns tpd
- $C$  = draw round puck, use multiply with 9ns tpd
- System clock 65mhz = 15ns period - opps

No pipeline

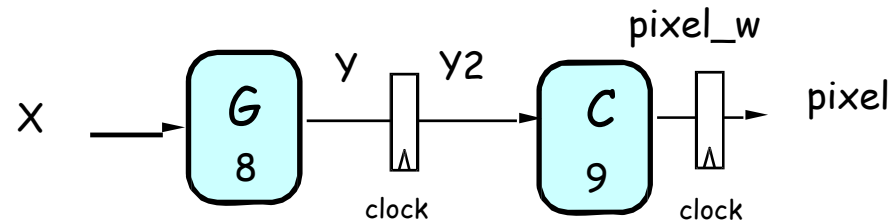
```

assign y = G(x);      // logic for y
assign pixel = C(y)    // logic for pixel
  
```



```

reg [N:0] x,y;
reg [23:0] pixel
always @ * begin
    y=G(x);
    pixel = C(y);
end
  
```



Pipeline

```

always @(posedge clock) begin
    ...
    y2 <= G(x);      // pipeline y
    pixel <= C(y2)    // pipeline pixel
end
  
```

Latency = 2 clock cycles!  
Implications?

分開comb, seq logic

```

always @ * begin
    y=G(x);
    pixel_w = C(y);
end
always @(posedge clock)begin
    y2 <= y;          // pipeline y
    pixel <= pixel_w; // pipeline pixel
end
  
```

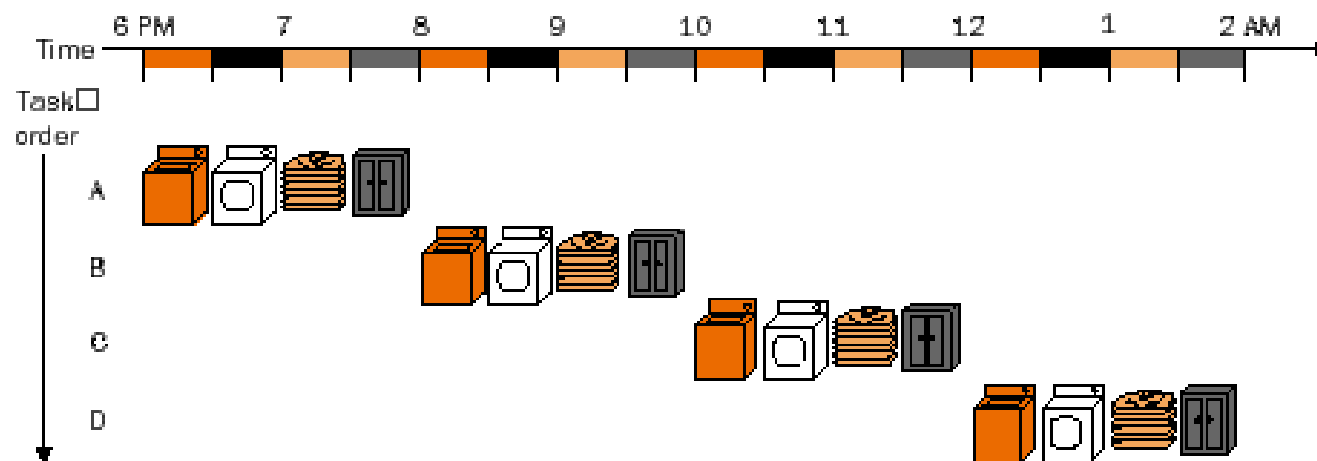


# PIPELINE AND PARALLEL

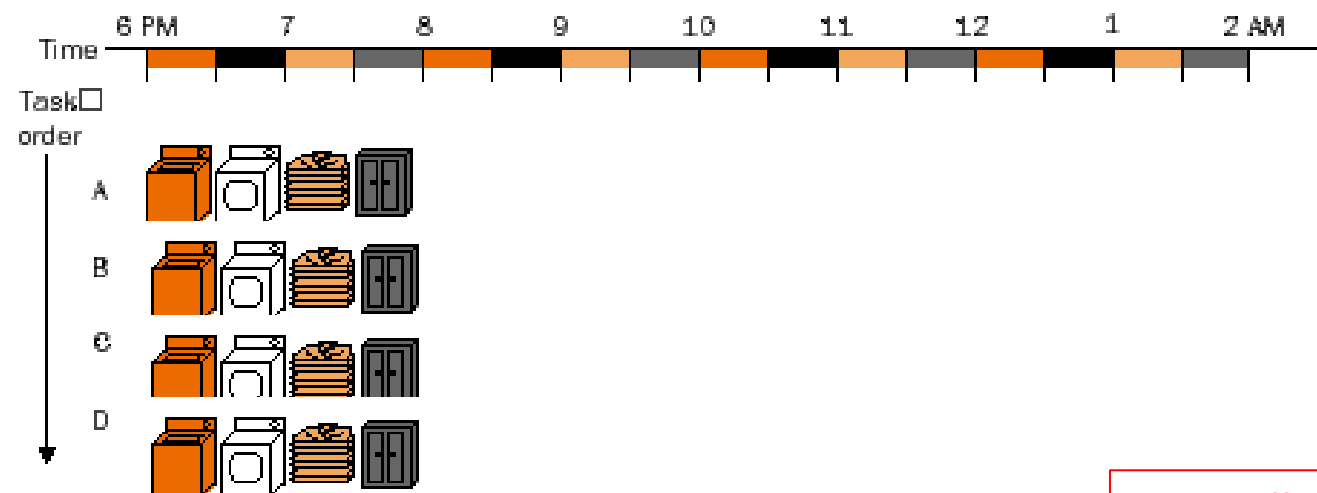
# 變快的兩招：平行多套 或與 pipeline

原來執行方式  
一個資料全部執行完，  
才換下一個資料

Q. 缺點



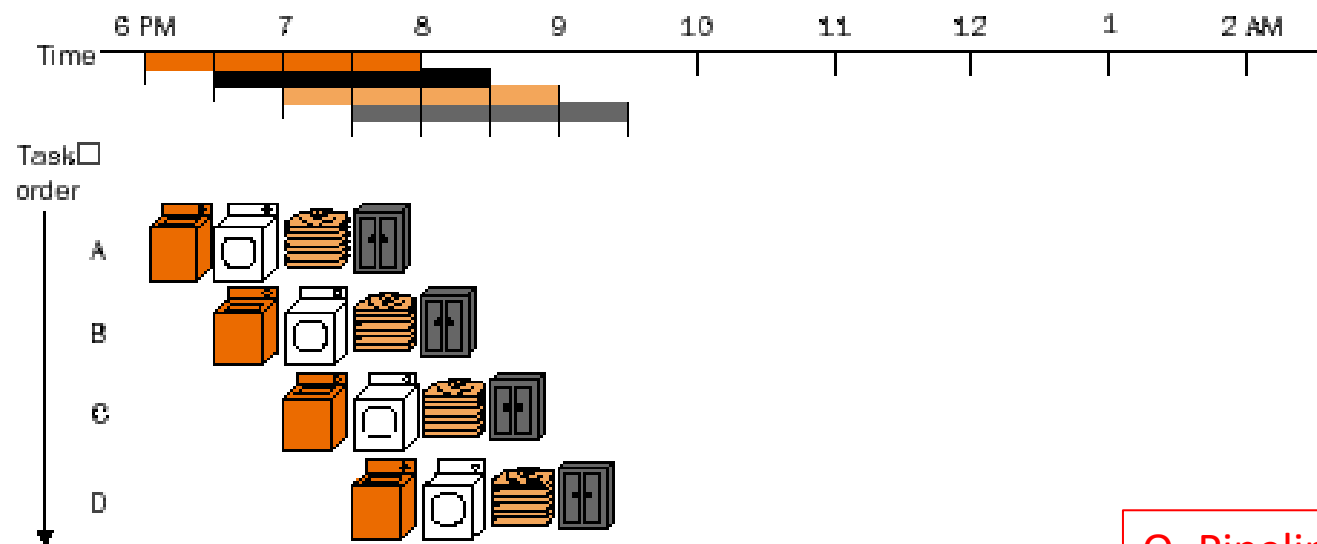
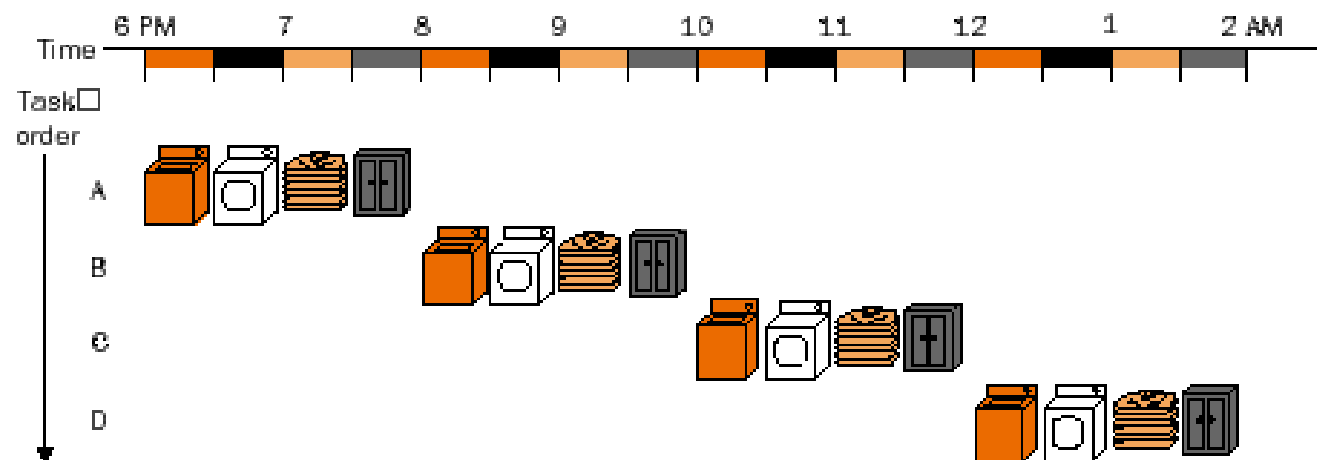
Parallel  
使用多套硬體，  
同時處理多個資料  
輸出變快



Q. Parallel 極限在哪裡

# 變快的兩招：平行多套 或與 pipeline

原來執行方式  
一個資料全部執行完，  
才換下一個資料

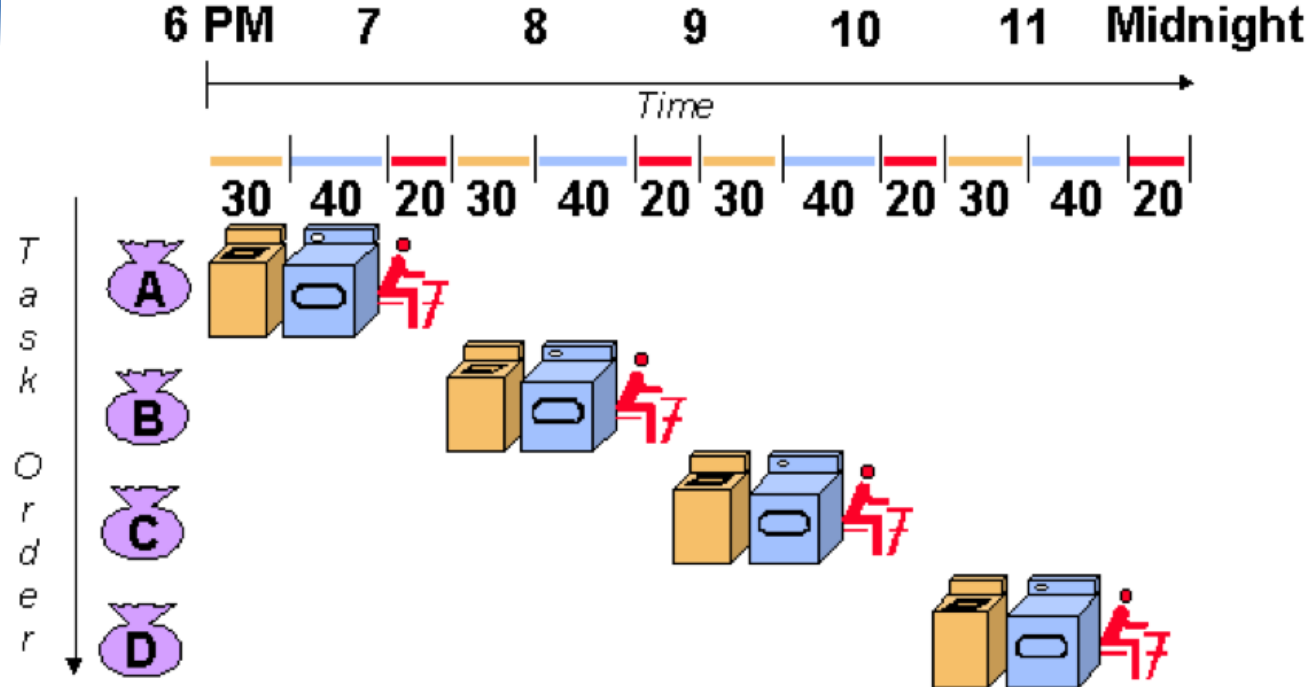


## Pipeline

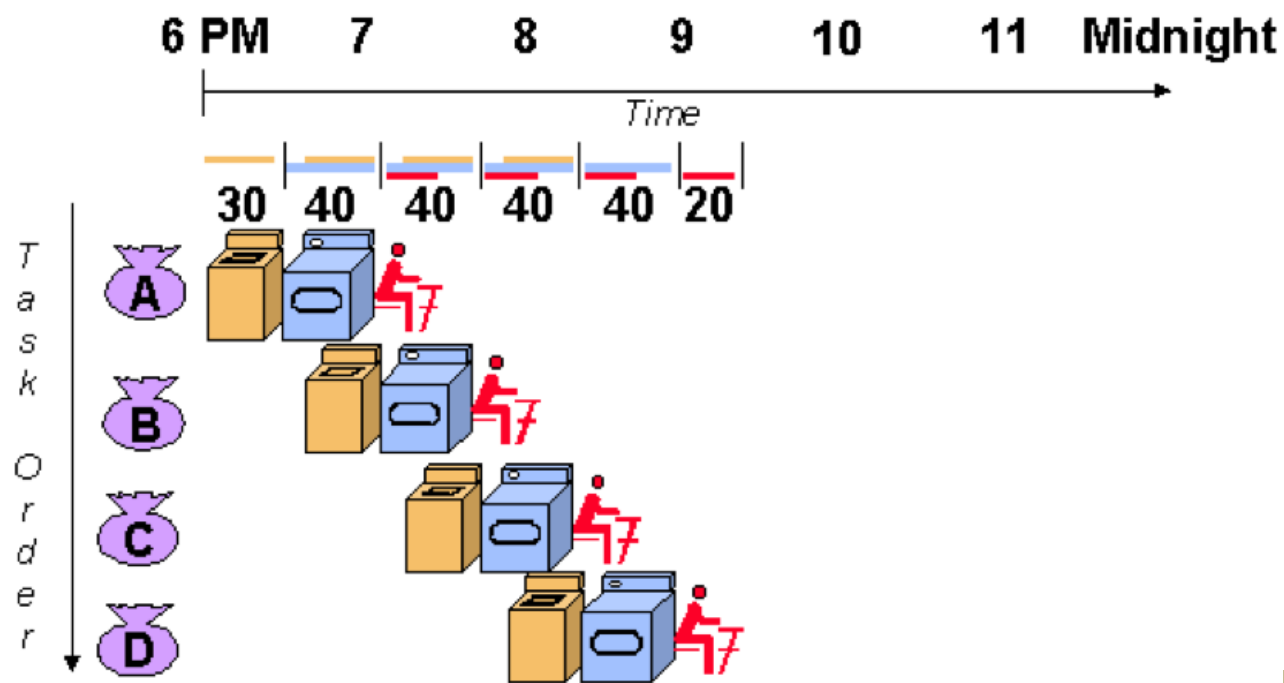
分成多級，執行完一級，  
下一個資料就可以進行處理  
把一級所需時間  
切得更細更少，頻率變快  
輸出變快

Q. Pipeline 極限在哪裡

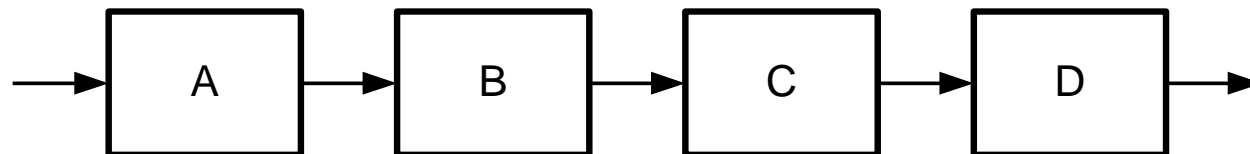
各級速度不一樣



會被最慢的哪一級限制住速度

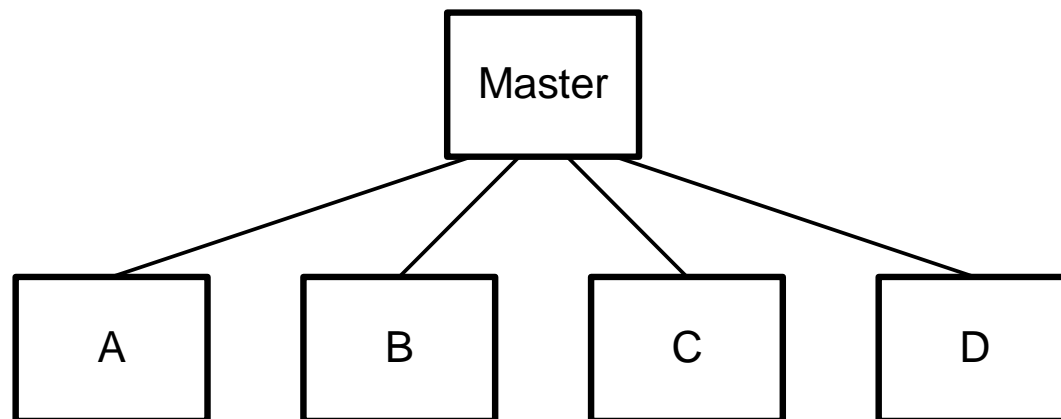


# pipeline and parallel 適合用在哪？



//for computation-oriented, e.g.  $Z = \sum |X_i - Y_i|$   
in motion/pattern detection

計算類比較規則，適合pipeline  
Or parallel



//for control-oriented, e.g. if or while  
Statement in decision making

控制類較不規則，pipeline  
容易空轉，適合parallel

# Pipeline a 32-bit ripple carry adder

- 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

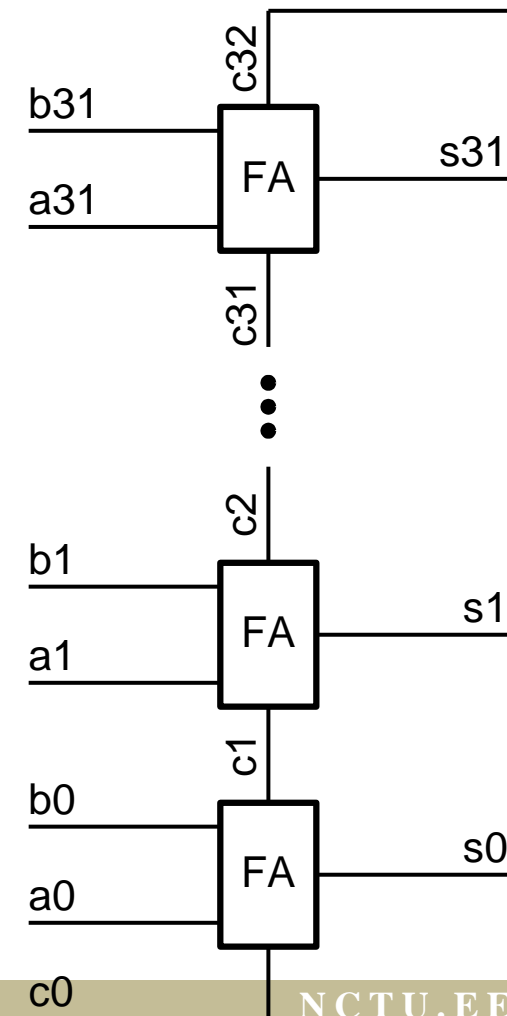
**Exercise:** What is the critical Path and area of this 32-bit Ripple adder?

Assume one-bit full adder (FA)

Has the following:

Delay:  $T_{1bFA}$

Area:  $A_{1bFA}$



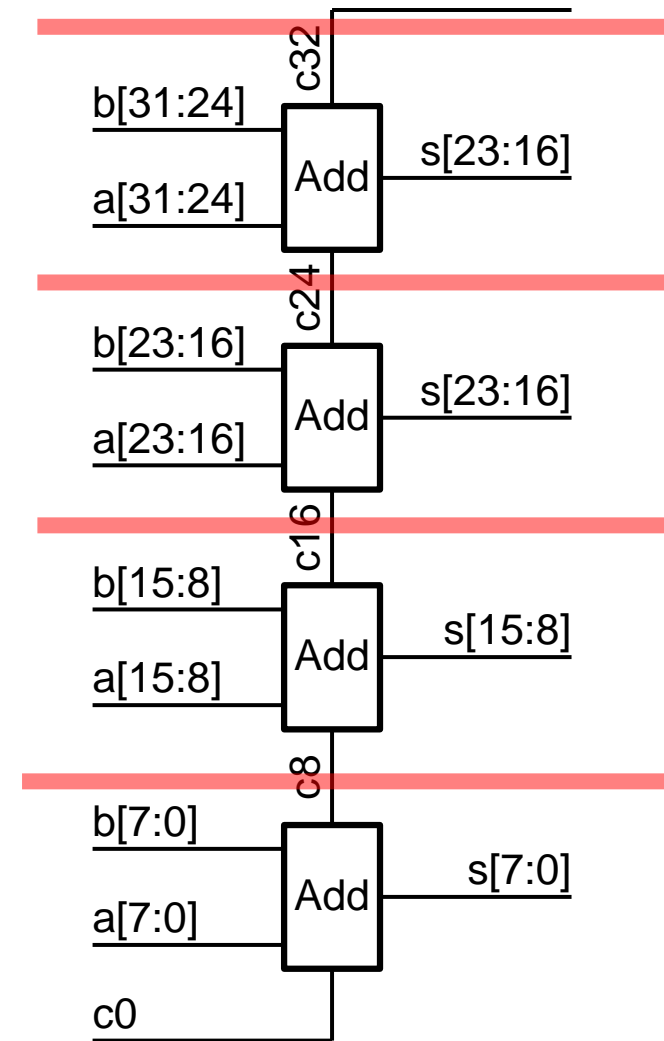


# Split into 4 8-bit adders

Note: the purpose of exploiting Pipelining is to speed up circuit Performance. As a result, each 8-bit Adder can be improved

By those carry propagation Reduction methods mentioned Earlier.

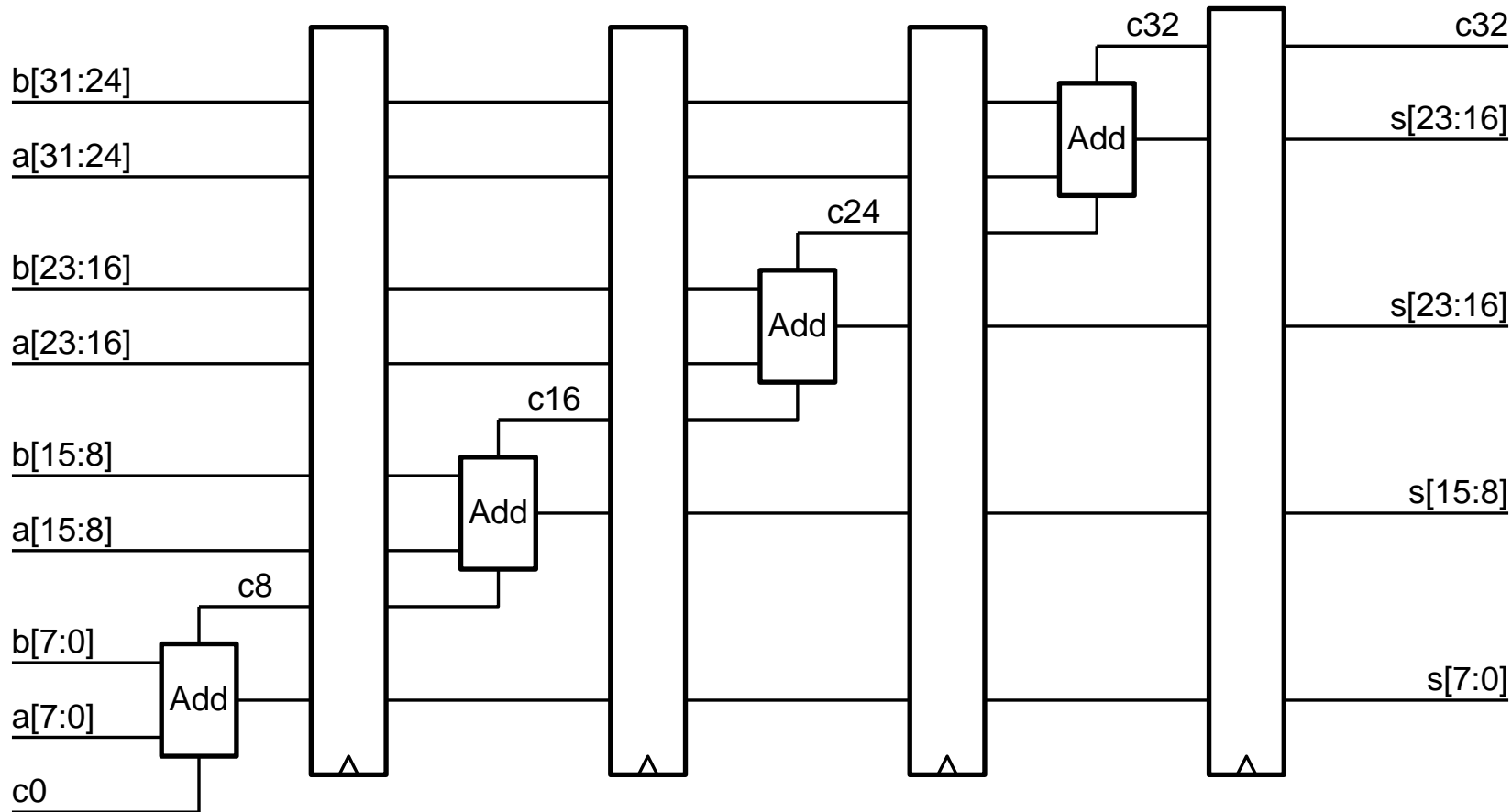
For example, carry-look-ahead (CLA) with carry-select-adder (CSA) Can be combined to speed up carry Propagation.



# Split into stages

## 4 problems 'in process' at once

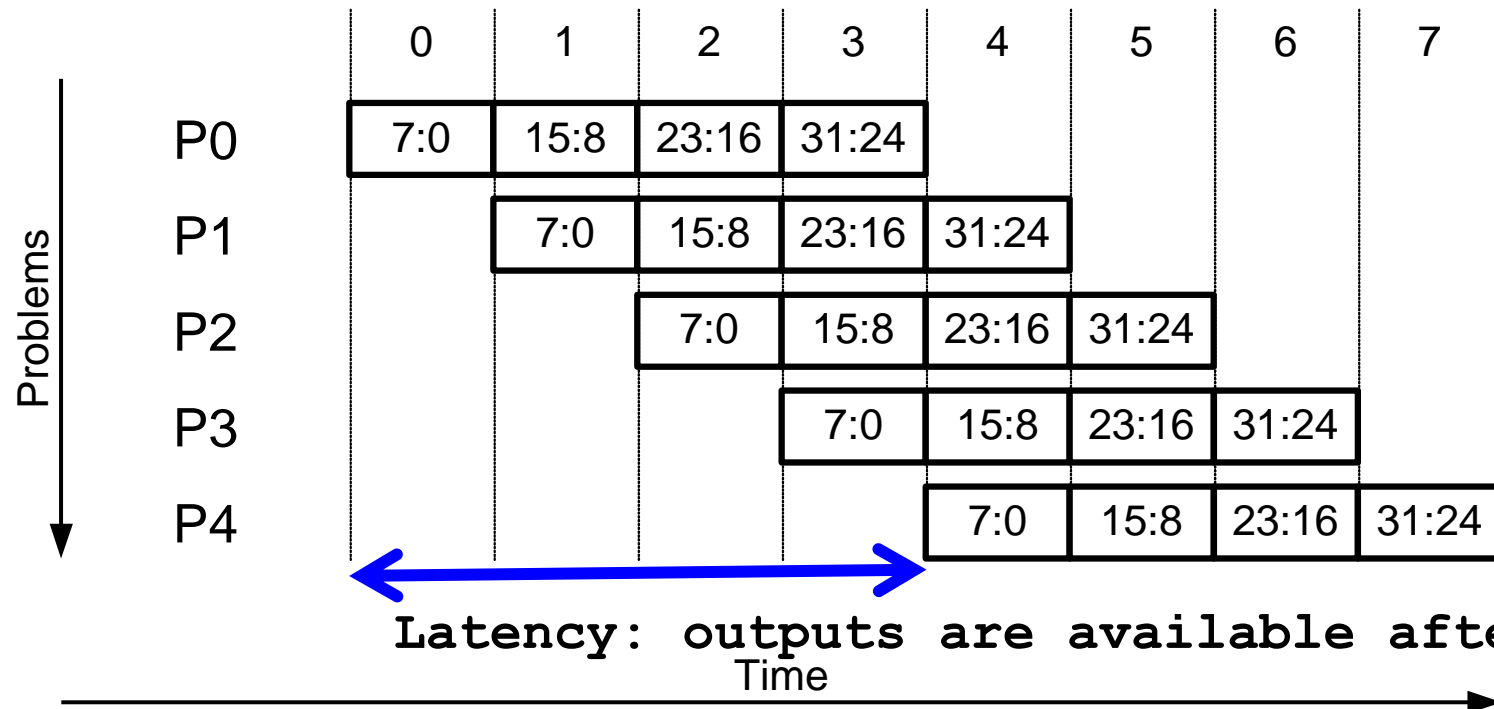
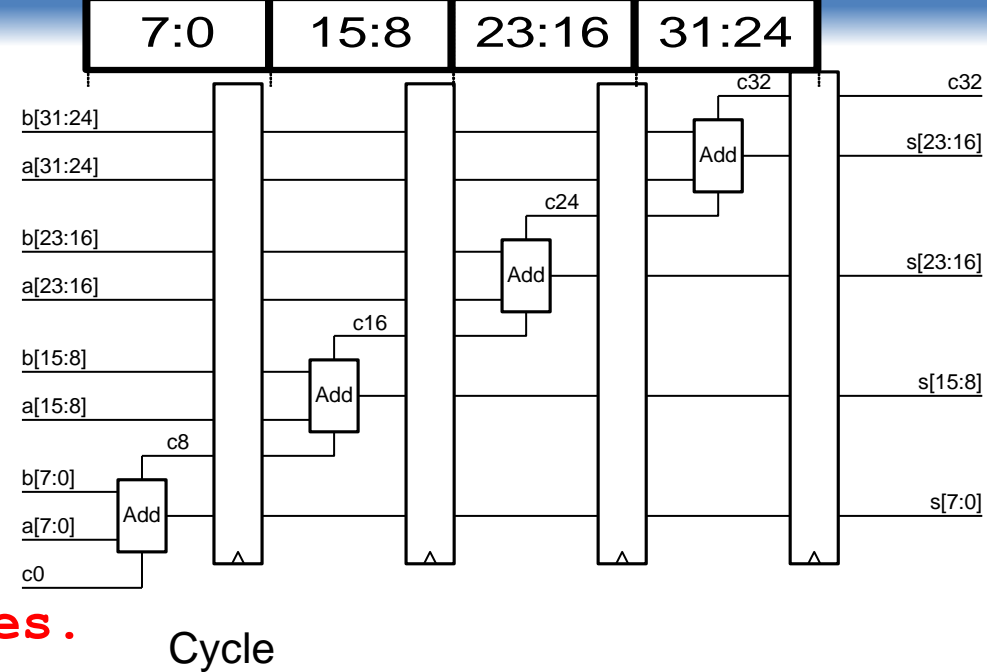
**Exercise:** what is the clock rate for this pipelined adder?  
How many extra areas are demanded to realize this adder?



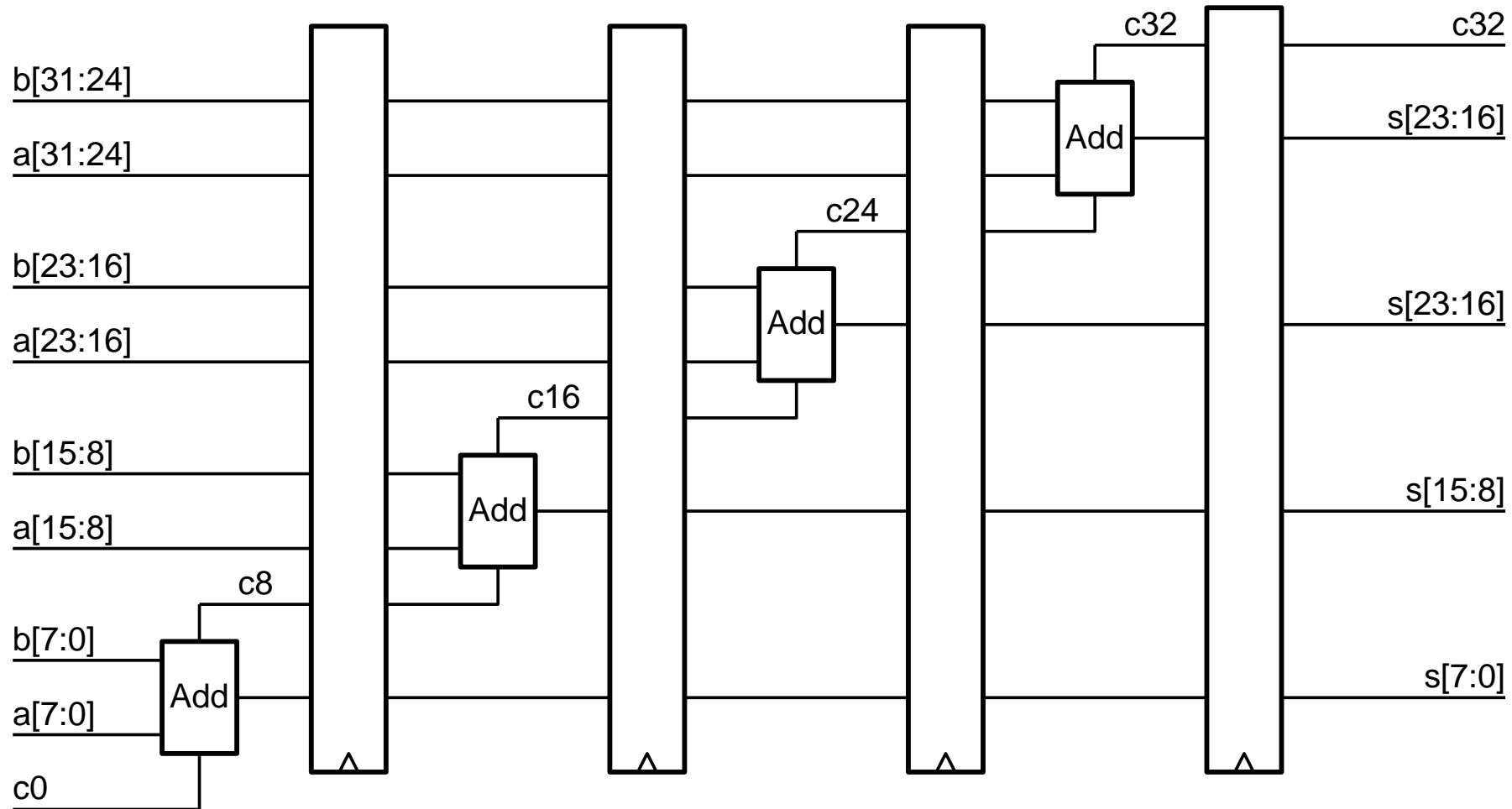
# Pipeline Diagram

## Illustrates pipeline timing

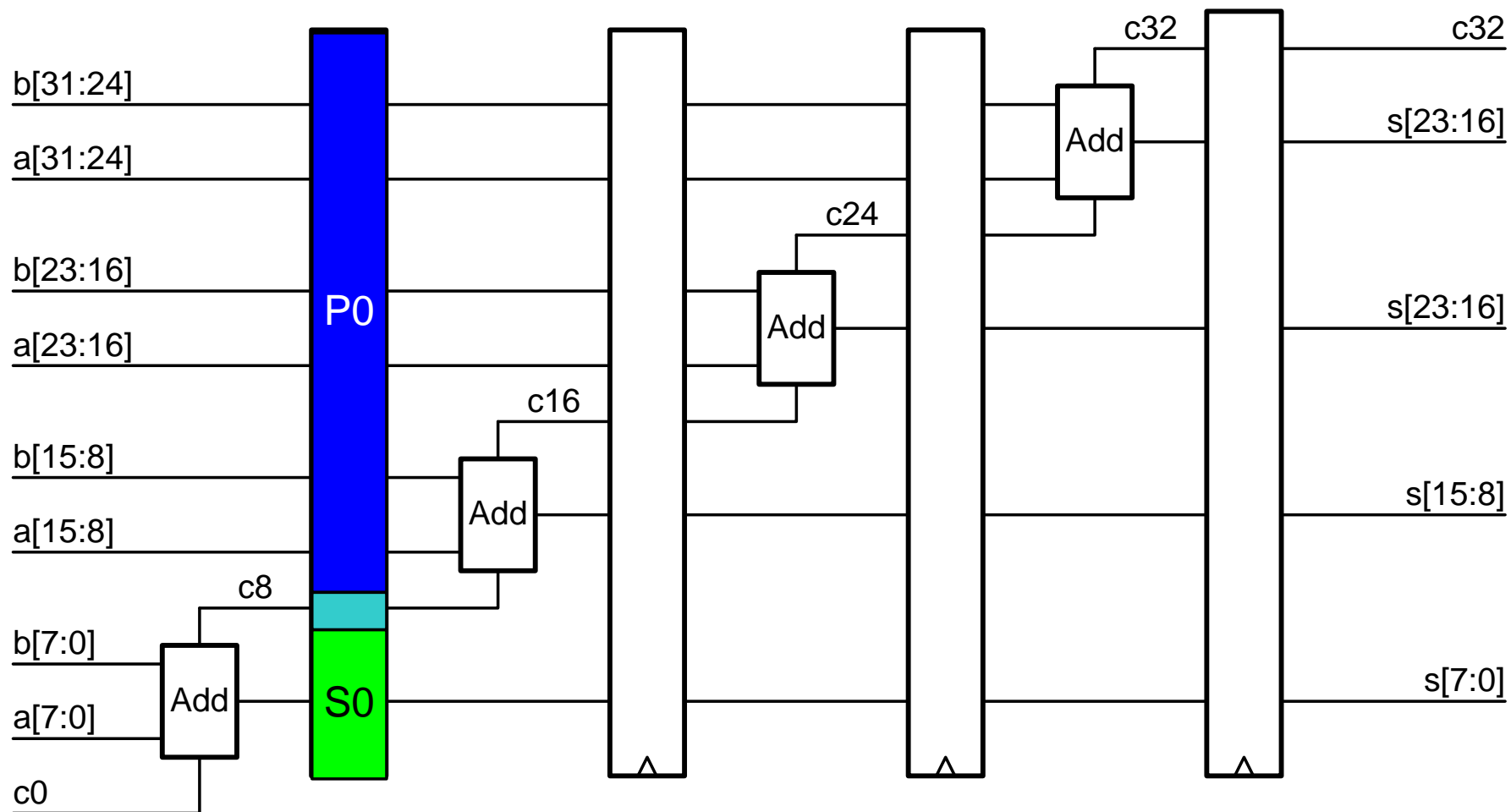
Latency depends on the  
Number of pipelining stages.



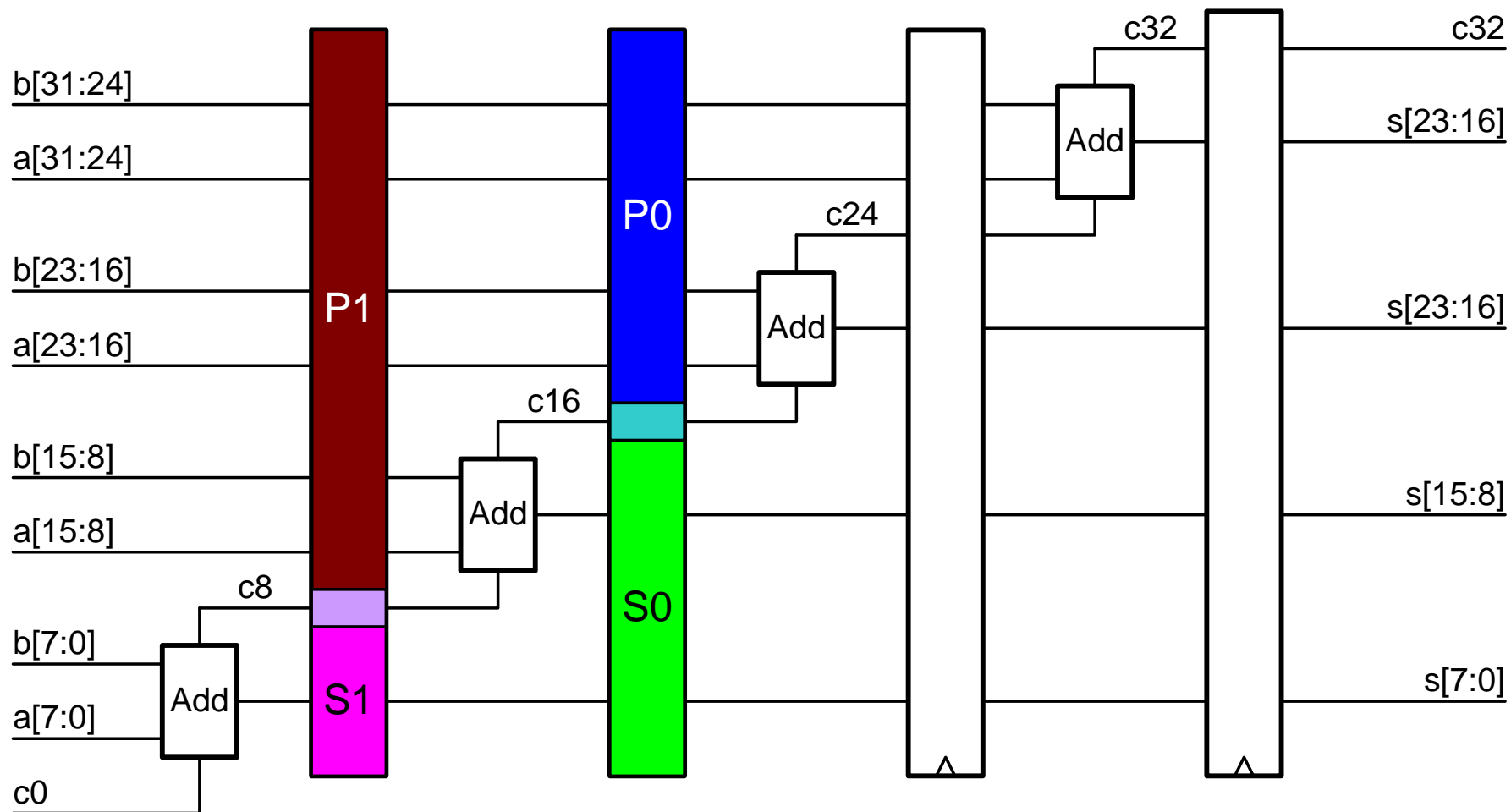
# Movie Illustration of Pipelined Adder



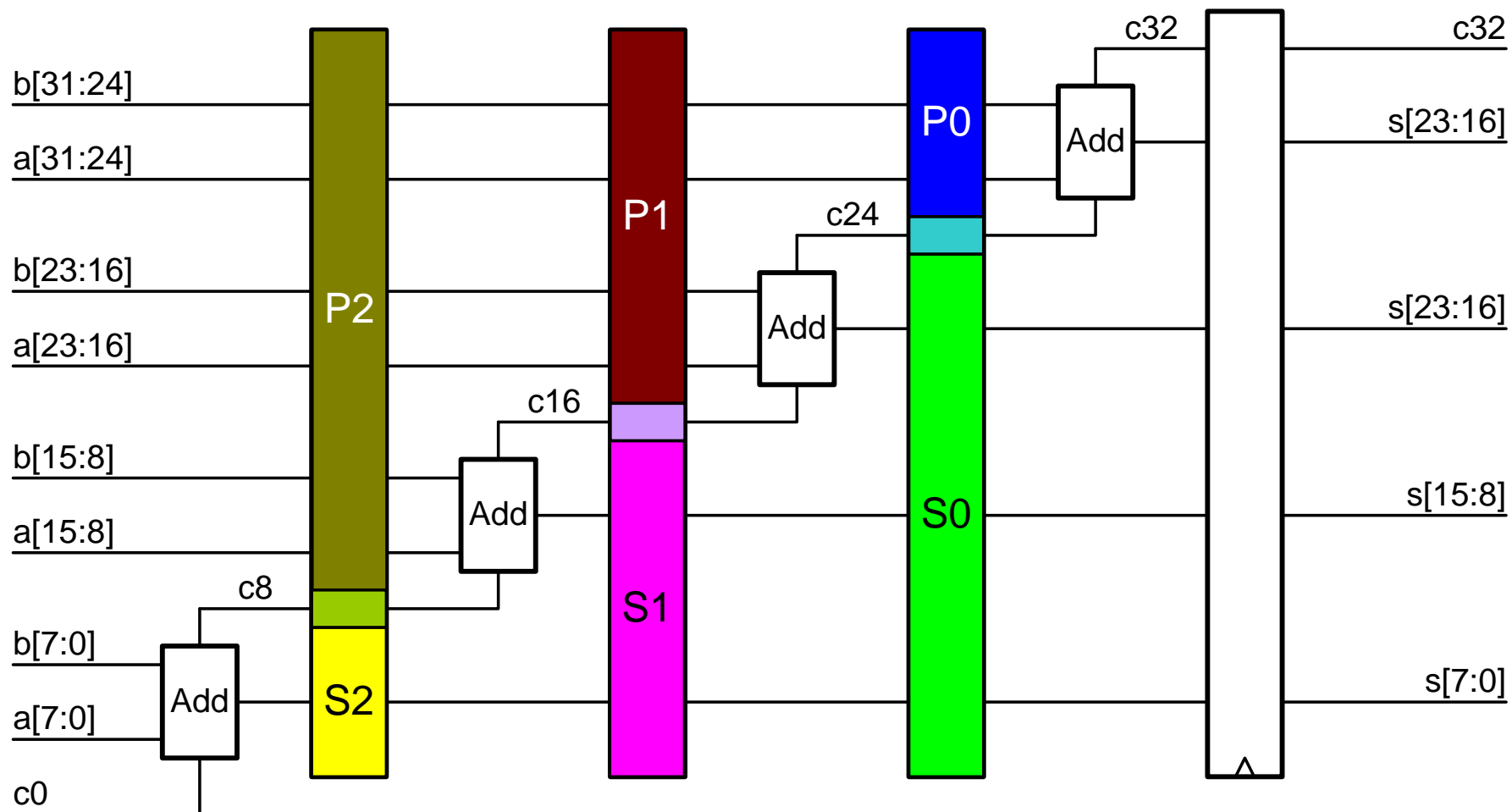
# Cycle 1



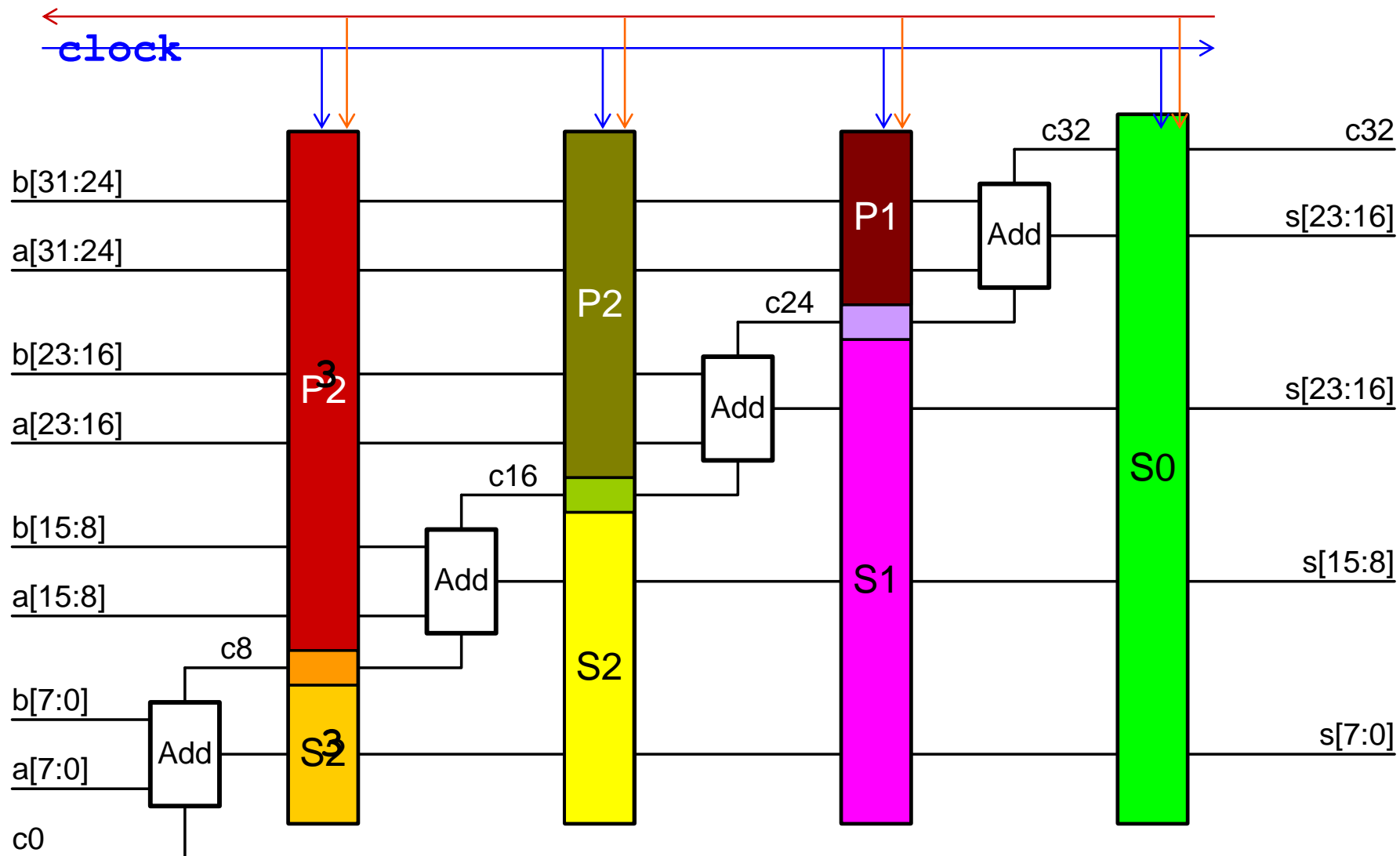
# Cycle 2



# Cycle 3



# Cycle 3



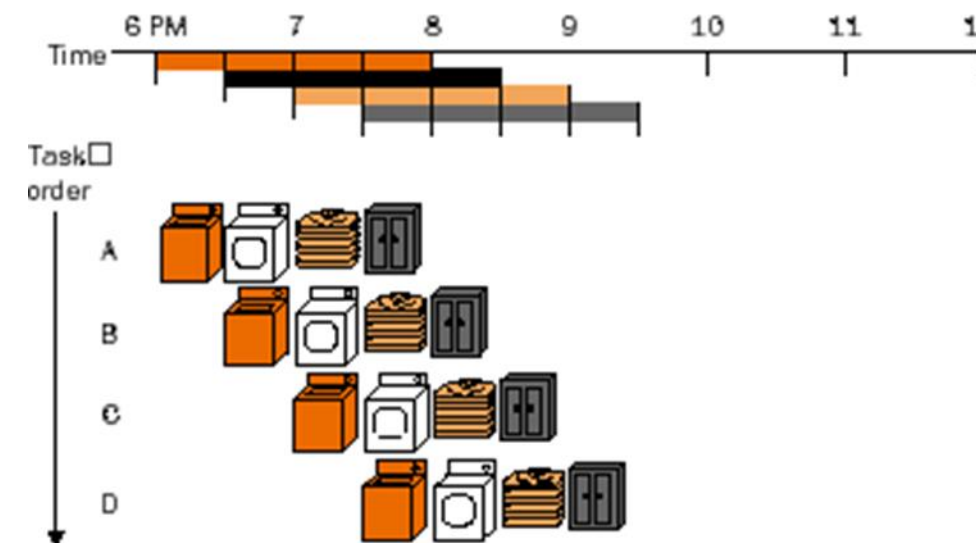


# Pipeline可以跑多快: Latency and throughput

- Latency != delay
  - 1 add per 10 ns => latency = 10ns
  - Pipeline 從第一級執行到最後一級所花的時間
  - Latency =  $\Sigma$ (一級組合邏輯時間+pipeline overhead)
  - Pipeline overhead = register (DFF)所需要的時間
  - Pipeline 不會加快單一工作的latency



- Throughput(Pipeline會增加throughput)
  - 單位時間內可以處理完的工作
    - 1 add per 10ns => 100MOPs
  - Throughput = 1/(pipeline 的工作頻率)
  - Pipeline的工作頻率 =  
1/(最長那一級組合邏輯時間+pipeline overhead)

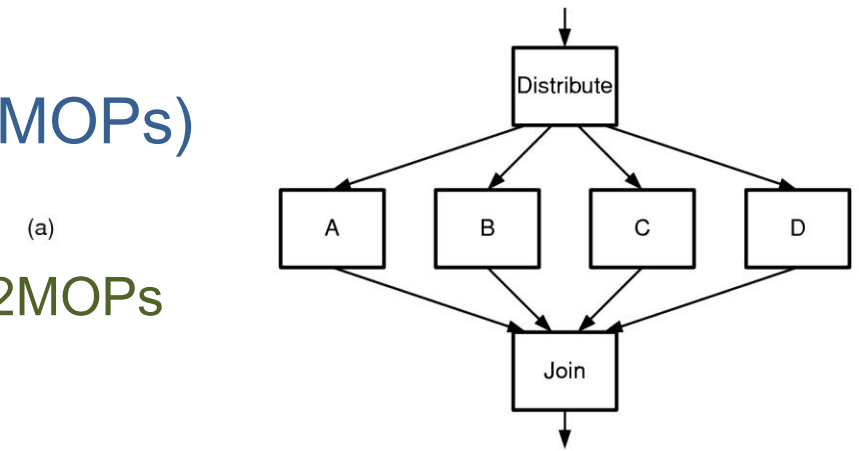


# Example 算算可以跑多快

- Increase throughput of a module with delay  $T = 10\text{ns}$  by  $\times 4$

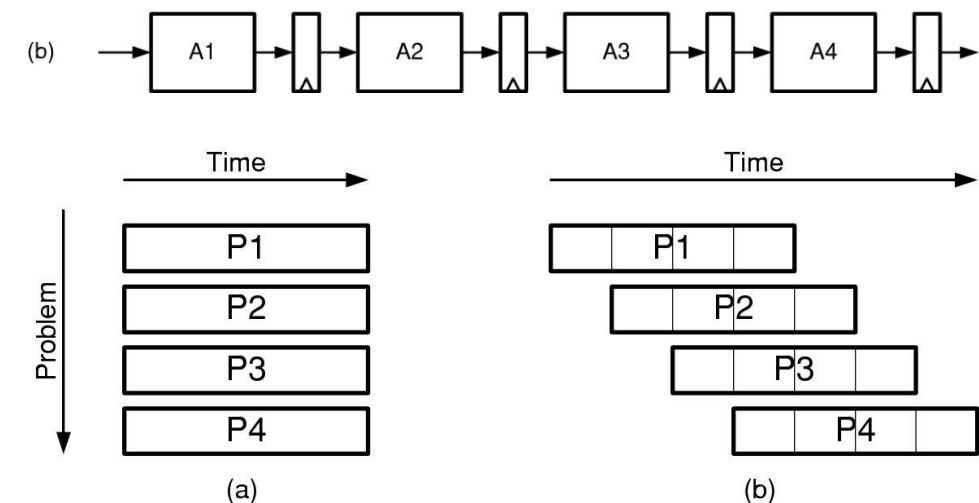
- **Parallel:**

- four copies.  $T = 10\text{ns}$ , but with 4 output (400MOPs)
- With register overhead ( $0.2\text{ns}$ ),
  - latency =  $10 + 0.2 = 10.2\text{ns}$ , throughput =  $4 / 10.2 = 392\text{MOPs}$



- **pipeline:**

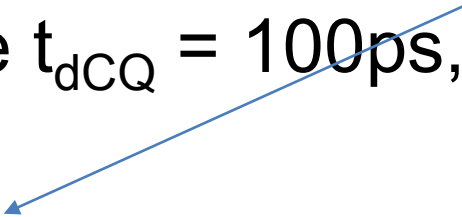
- Four stages, each stage delay =  $10/4 = 2.5$
- Ideally, (without register overhead)
  - One output per  $2.5\text{ns}$
  - $\Rightarrow 10\text{ns} \Rightarrow 4$  output  $\Rightarrow 4$  times throughput
- With Pipeline overhead ( $0.2\text{ns}$ )
  - Latency  $10 + 0.2 \times 4 = 10.8\text{ns}$
  - Throughput  $1 / (2.5 + 0.2) = 370\text{MOPs}$



# Example: Pipeline a Ripple Carry Adder

## 算算可以跑多快

- 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)


$$t_{\text{pipe}} = n(t_{\text{stage}} + t_{\text{dCQ}} + t_s + t_k) \text{ // with register overhead}$$

$$\Theta = n/t_{\text{pipe}} = 1/(t_{\text{stage}} + t_{\text{dCQ}} + t_s + t_k) \text{ // throughput}$$

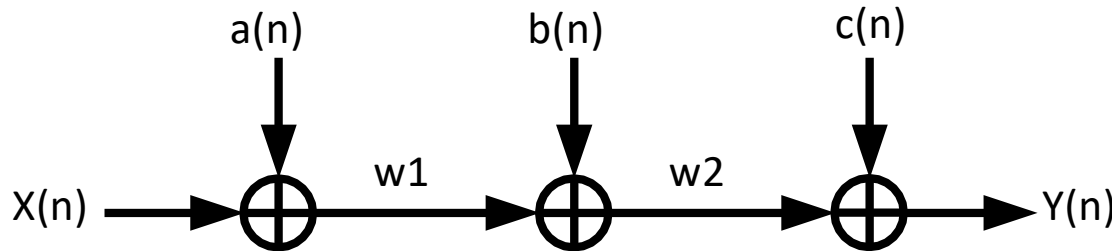
$$t_{\text{stage}} = 3200/4 = 800\text{ps} \quad t_{\text{dCQ}} + t_s + t_k = 100 + 50 + 50 = 200\text{ps}$$

$$t_{\text{pipe}} = 4(800 + 200) = 4000\text{ps} \quad \Theta = 4/4000\text{ps} = 1 \text{ GOPs}$$

# MORE EXAMPLES

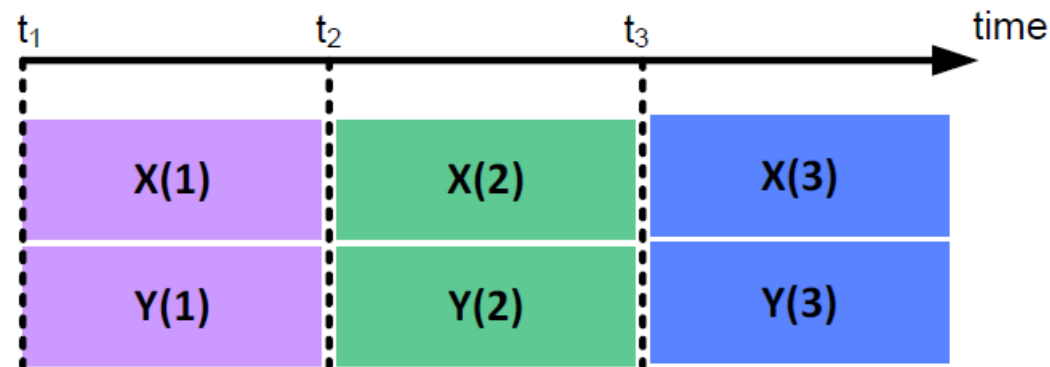
# Architectural Techniques : Pipeline depth

- Pipeline depth: 0 (No Pipeline)
  - Critical path: 3 Adders



```
wire w1, w2; assign  
w1 = X + a; assign  
w2 = w1 + b; assign  
Y = w2 + c;
```

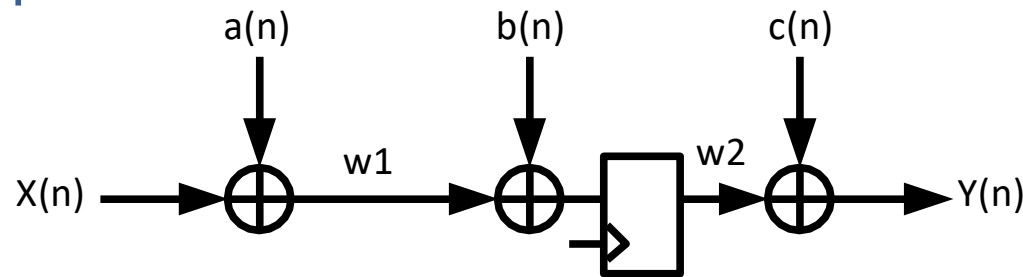
- Latency : 0



# Architectural Techniques : Pipeline depth

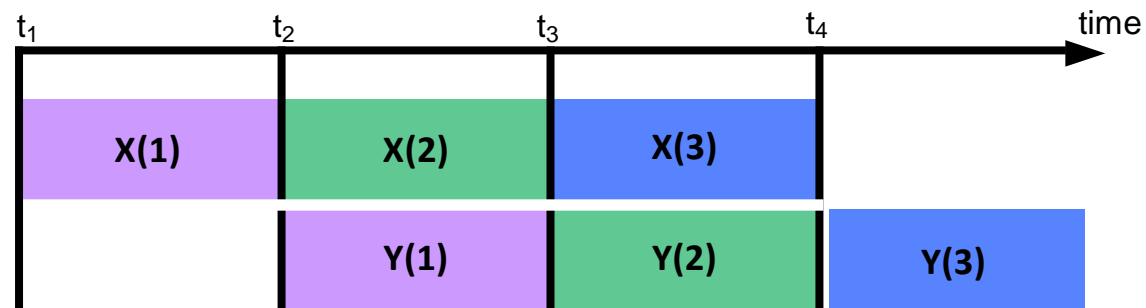
- Pipeline depth: 1 (One Pipeline register Added)

- Critical path: 2 Adders



```
wire w1;  
reg w2;  
assign w1 = X + a;  
assign Y = w2 + c;  
  
always @(posedge Clk)  
    w2 <= w1 + b;
```

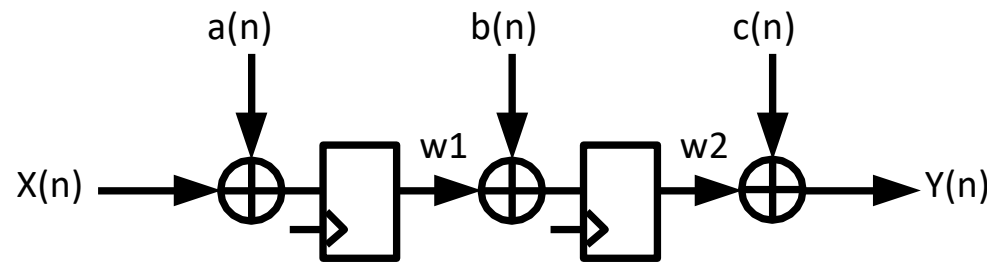
- Latency : 1



# Architectural Techniques : Pipeline depth

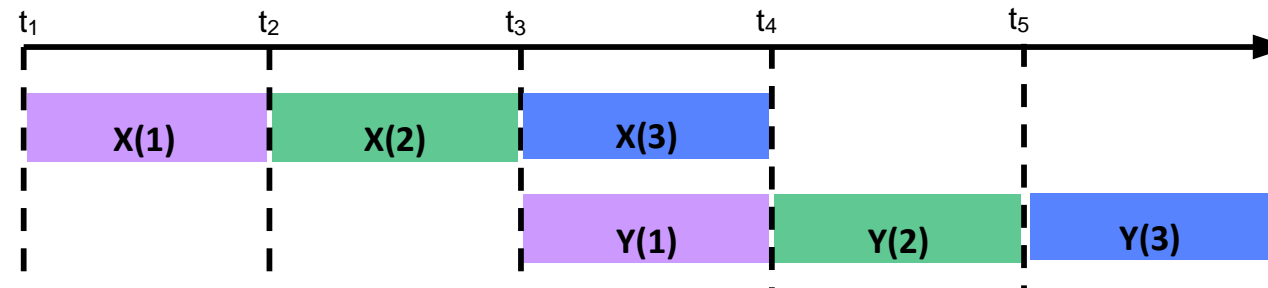
- Pipeline depth: 2 (One Pipeline register Added)

- Critical path: 1 Adder



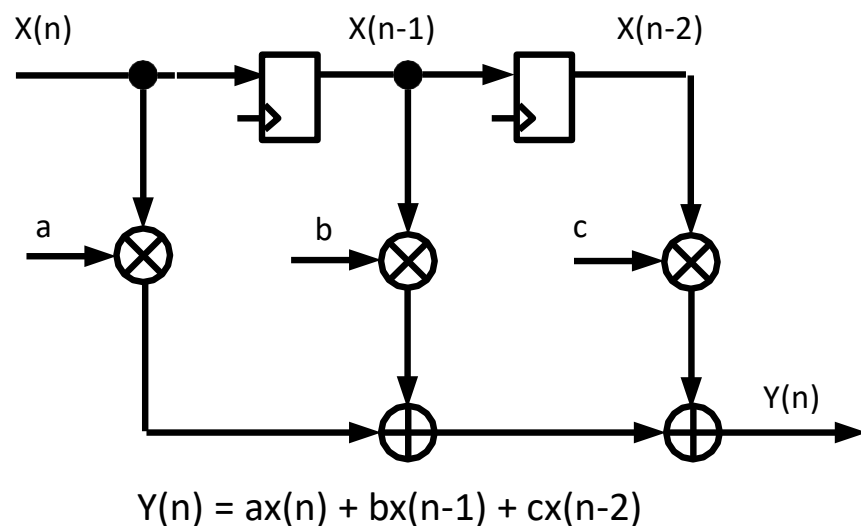
```
reg w1, w2;  
assign Y = w2 + c;  
  
always @(posedge Clk)  
begin  
    w1 <= X + a;  
    w2 <= w1 + b;  
end
```

- Latency : 2

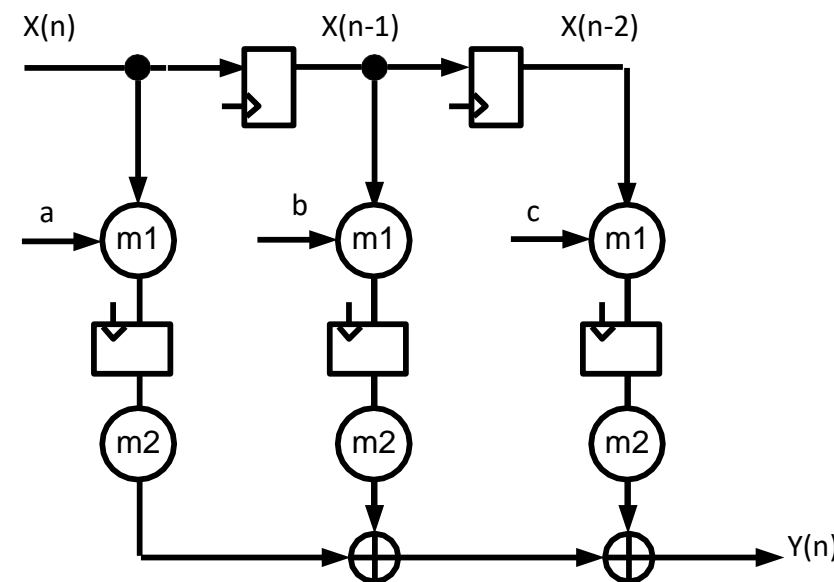


# Architectural Techniques : Fine-Grain Pipelining

- Pipelining at the operation level
  - Break the multiplier into two parts



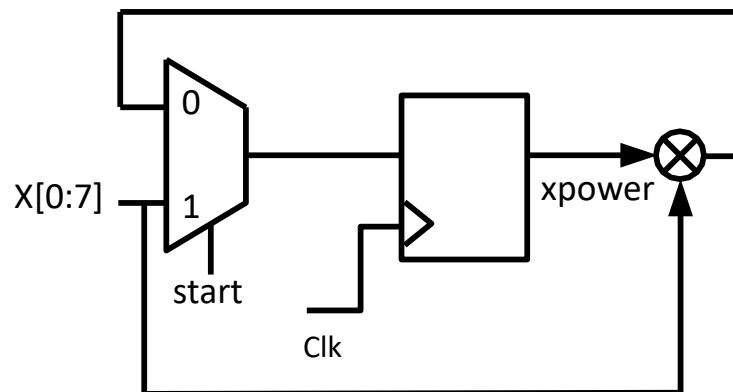
Fine-Grain  
Pipelining





# Unrolling the Loop Using Pipelining

- Calculation of X3
  - Throughput =  $8/3$ , or 2.7 bits/clock
  - Latency = 3 clocks
  - Timing = One multiplier in the critical path
  - Iterative implementation:
    - No new computations can begin until the previous computation has completed

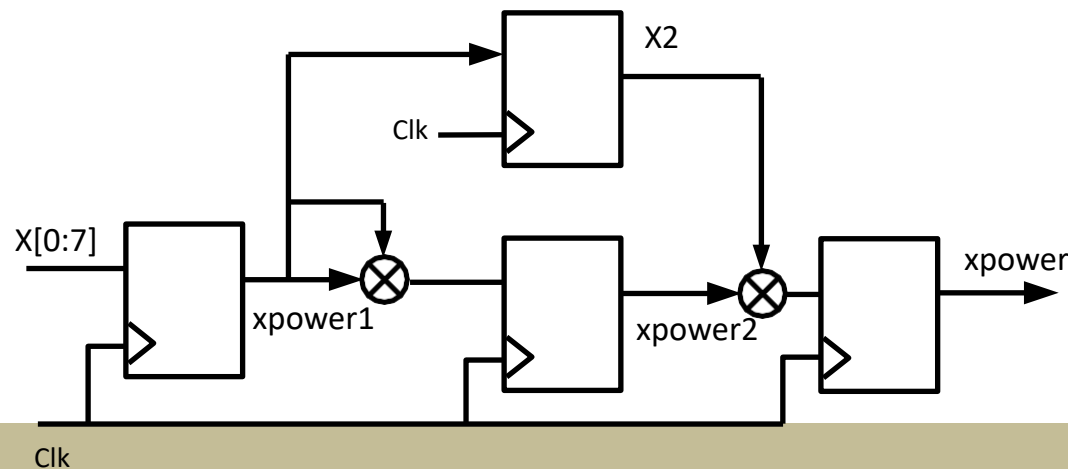


```
module power3(  
    output reg [7:0] X3,  
    output finished,  
    input [7:0] X,  
    input clk, start);  
    reg [7:0] ncount;  
    reg [7:0] Xpower, Xin;  
    assign finished = (ncount == 0);  
    always@(posedge clk)  
        if (start) begin  
            XPower <= X; Xin<=X;  
            ncount <= 2;  
            X3 <= XPower;  
        end  
        else if(!finished) begin  
            ncount <= ncount - 1;  
            XPower <= XPower * Xin;  
        End  
endmodule
```

# Unrolling the Loop Using Pipelining

- Calculation of X3
  - Throughput = 8/1, or 8 bits/clock (3X improvement)
  - Latency = 3 clocks
  - Timing = One multiplier in the critical path
- Penalty: More Area

Unrolling an algorithm with n iterative loops increases throughput by a factor of n

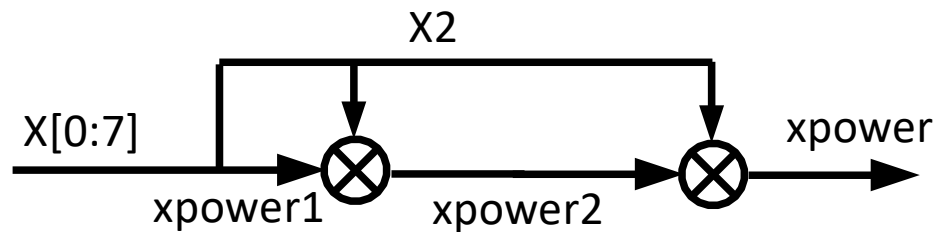


```
module power3(  
  output reg [7:0] XPower,  
  input clk,  
  input [7:0] X;  
  reg [7:0] XPower1, XPower2;  
  reg [7:0] X2;  
  always @(posedge clk) begin  
    // Pipeline stage 1  
    XPower1 <= X;  
    // Pipeline stage 2  
    XPower2 <= XPower1 * XPower1 ;  
    X2 <= XPower1 ;  
    // Pipeline stage 3  
    XPower <= XPower2 * X2;  
  end  
endmodule
```

# Removing Pipeline Registers (to Improve Latency)

- Calculation of X3
  - Throughput = 8 bits/clock (3X improvement)
  - Latency = 0 clocks
  - Timing = Two multipliers in the critical path

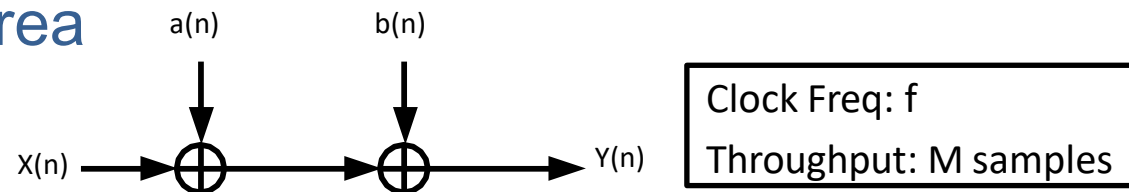
Latency can be reduced by removing pipeline registers



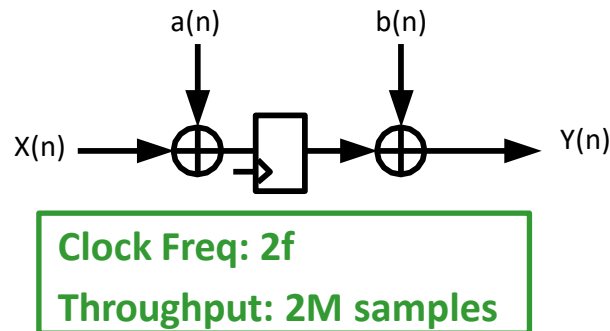
```
module power3(  
    Output [7:0] XPower,  
    input [7:0] X;  
    reg [7:0] XPower1, XPower2;  
    reg [7:0] X1, X2;  
    always @(*)  
        XPower1 = X;  
    always @(*)  
    begin  
        X2 = XPower1;  
        XPower2 = XPower1*XPower1;  
    end  
  
    assign XPower = XPower2 * X2;  
  
endmodule
```

# Architectural Techniques : Parallel Processing

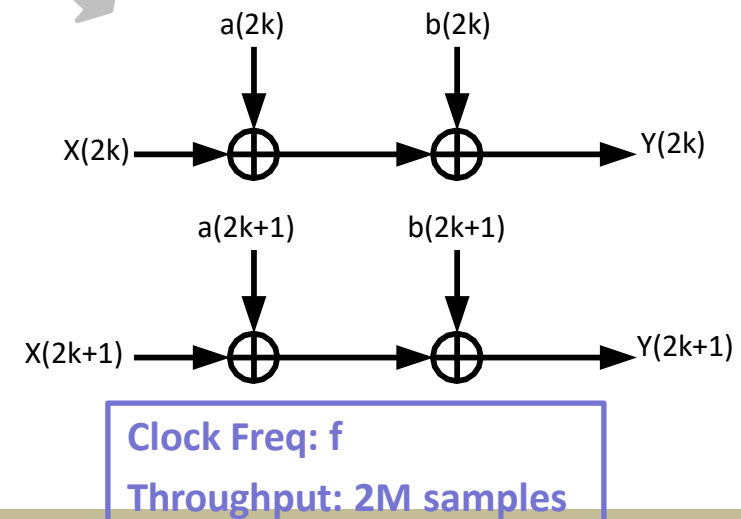
- In parallel processing the same hardware is duplicated to
  - Increases the throughput without changing the critical path
  - Increases the silicon area



Pipelining

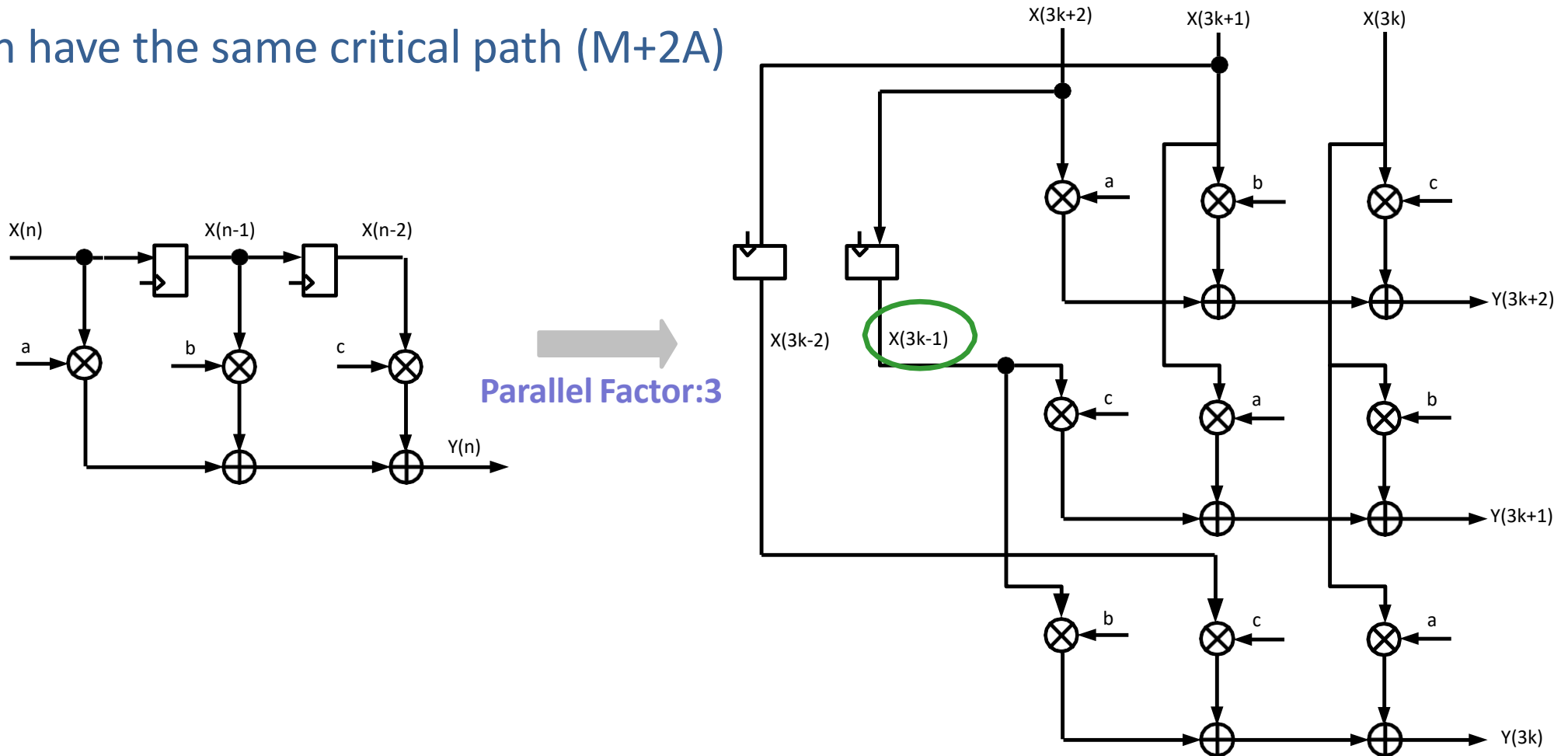


Parallel Processing



# Architectural Techniques : Parallel Processing

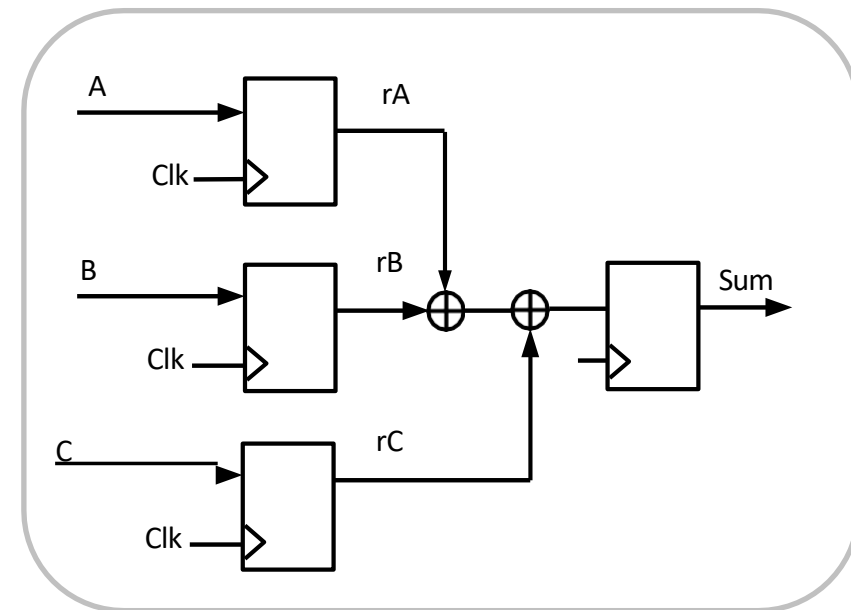
- Parallel processing for a 3-tap FIR filter
  - Both have the same critical path ( $M+2A$ )



# Register Balancing (to Improve Timing)

- Redistribute logic evenly between registers to minimize the worst-case delay between any two registers
  - b/c clock is limited by only the worst-case delay

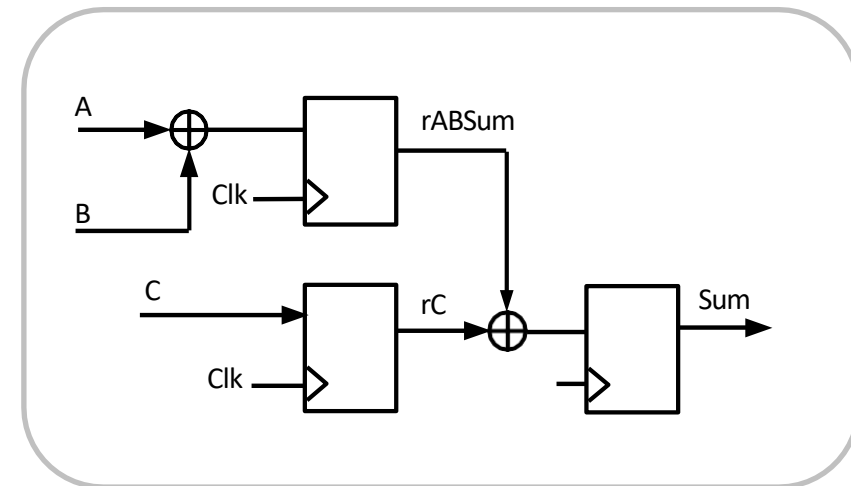
```
module adder(  
  output reg [7:0] Sum,  
  input [7:0] A, B, C,  
  input clk);  
  reg [7:0] rA, rB, rC;  
  always @(posedge clk) begin  
    rA <= A;  
    rB <= B;  
    rC <= C;  
    Sum <= rA + rB + rC;  
  end  
endmodule
```



# Register Balancing (to Improve Timing)

- Redistribute logic evenly between registers to minimize the worst-case delay between any two registers
  - b/c clock is limited by only the worst-case delay

```
module adder(  
  output reg [7:0] Sum,  
  input [7:0] A, B, C,  
  input clk);  
  reg [7:0] rABSum, rC;  
  always @(posedge clk) begin  
    rABSum <= A + B;  
    rC <= C;  
    Sum <= rABSum + rC;  
  end  
endmodule
```



# Speed-related Techniques: Summary

- Throughput-related:
  - A high-throughput architecture is one that maximizes the number of bits per second that can be processed by a design.
  - Unrolling an iterative loop increases throughput.
  - The penalty for unrolling an iterative loop is an increase in area.
- Latency-related:
  - A low-latency architecture is one that minimizes the delay from the input of a module to the output.
  - Latency can be reduced by removing pipeline registers.
  - The penalty for removing pipeline registers is an increase in combinatorial delay between registers.



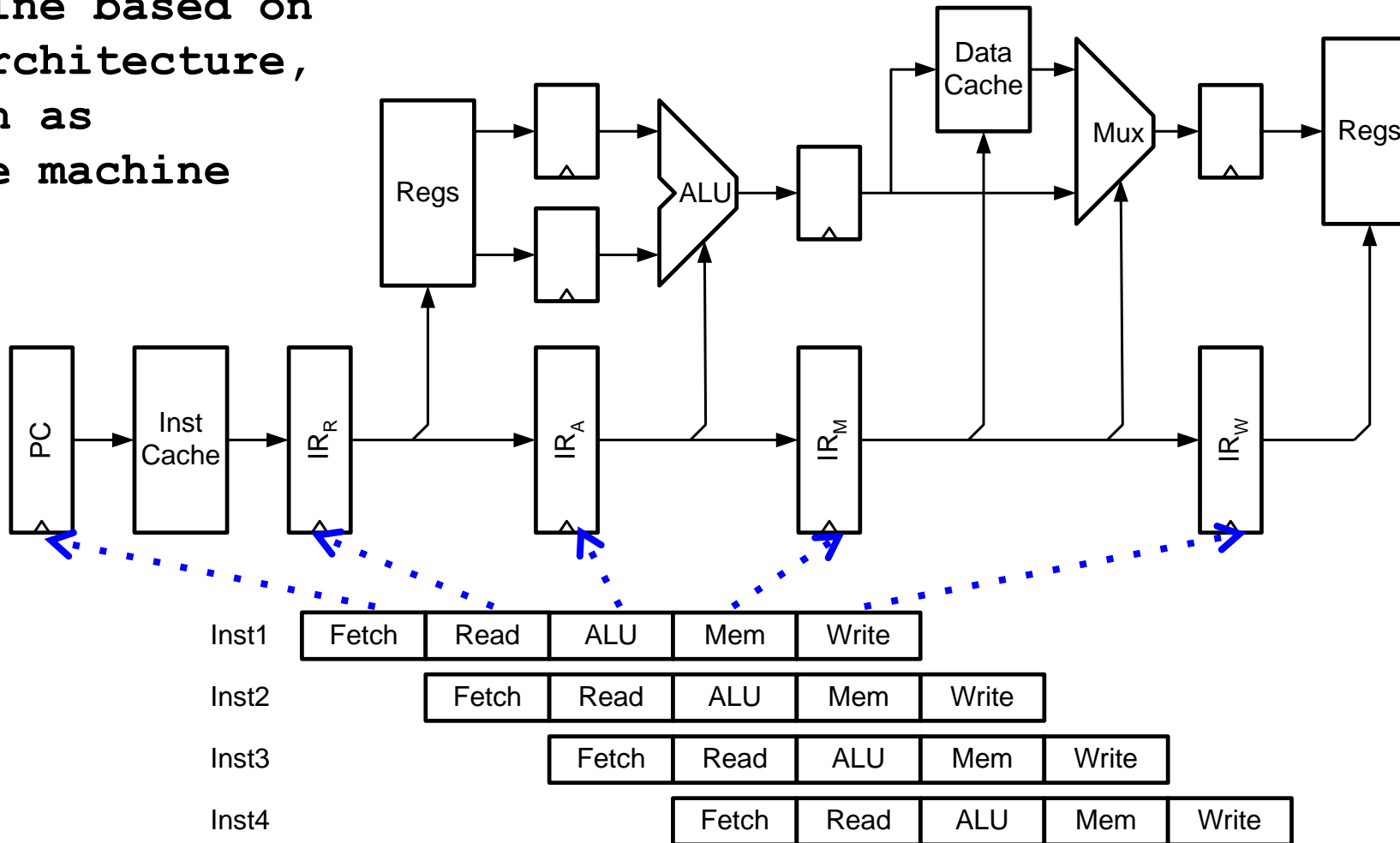
# Speed-related Techniques: Summary

- Timing-related:
  - Timing refers to the clock speed of a design. A design meets timing when the maximum delay between any two sequential elements is smaller than the minimum clock period.
  - Adding register layers improves timing by dividing the critical path into two paths of smaller delay.
  - Separating a logic function into a number of smaller functions that can be evaluated in parallel reduces the delay to the longest of the substructures.
  - By removing priority encodings where they are not needed, the logic structure is flattened, and the path delay is reduced.
  - Register balancing improves timing by moving combinatorial logic from the critical path to an adjacent path.

# **SIMPLE AND COMPLEX PIPELINE**

# Example 2: Processor Pipeline

RISC machine based on  
Harvard architecture,  
Also known as  
Load-Store machine

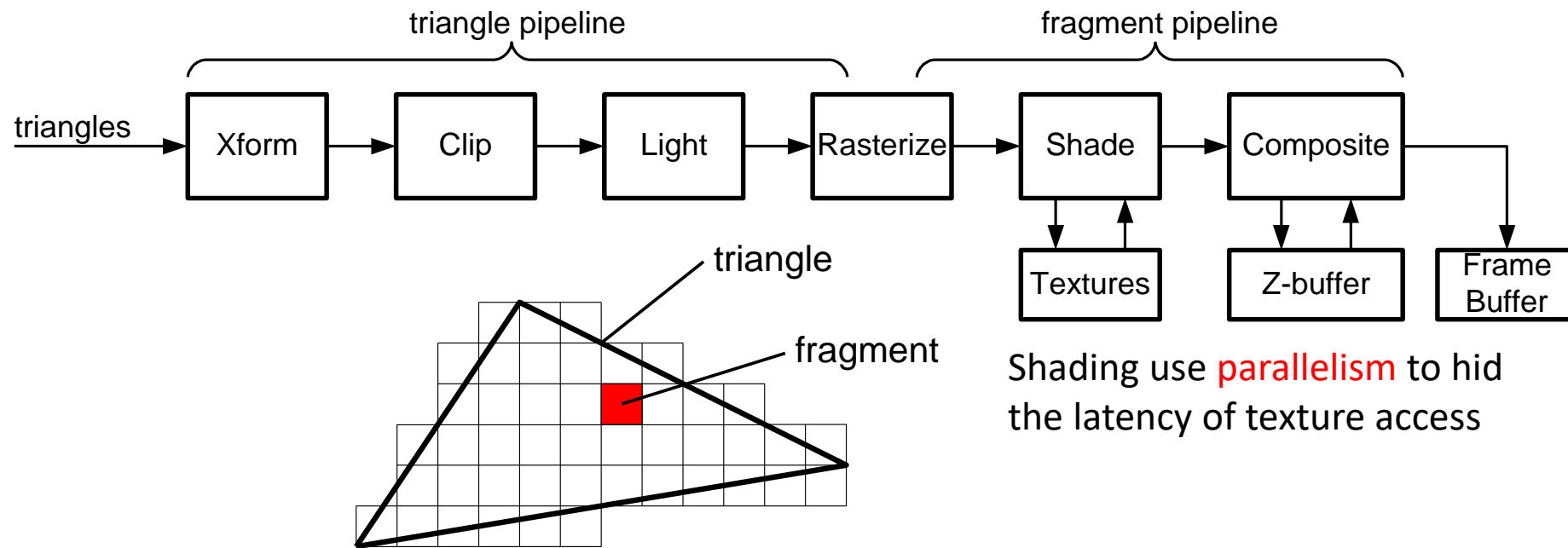


# Example 3: Graphics rendering pipeline

Graphic processor, dominated by Nvidia

Large amounts of Graphic processing units (GPU's)

To deal with pixel-based operations.

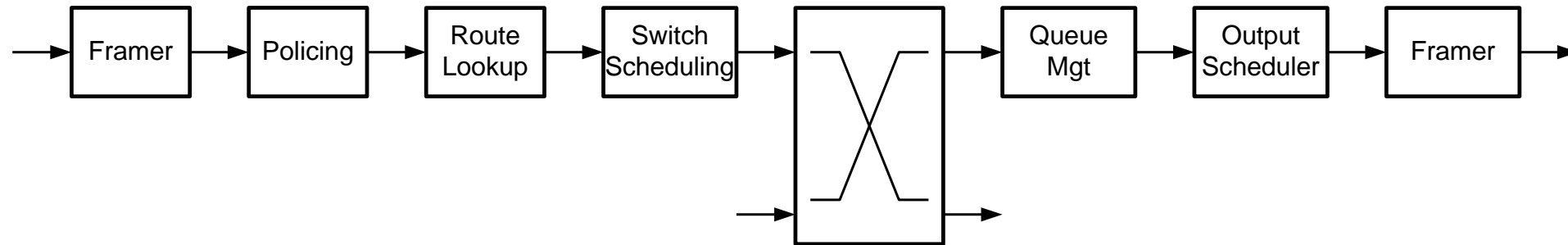


Shading use **parallelism** to hid the latency of texture access

- Pipeline 也可以做成階層式  
分成幾個大的pipeline stage  
每個大的stage再拆成更小的pipeline stage

- Pipeline 和 parallel 常常同時一起用

# Example 4 – Packet Processing Pipeline



- Pipelined with parallel pipelines for all inputs and output ports
- Some blocks takes variable amount of time, and needs **FIFOs** to smooth I/O throughput
- And each of these modules is **internally pipelined**
- You get the idea. Lots of systems are organized this way.

以大的block 來看，各block所需時間和資料順序不一，加FIFO使各級輸出入較平滑  
另外各block再切成較小的pipeline stage，使所有stage所需時間盡量一致

# PIPELINE設計問題: 空轉的PIPELINE與解法

勞逸不均造成的問題

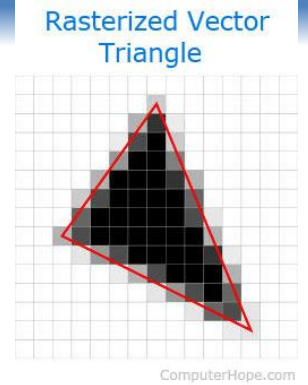
避免STALL解法: 大家都一樣快

容忍STALL解法: FIFO OR DOUBLE BUFFER

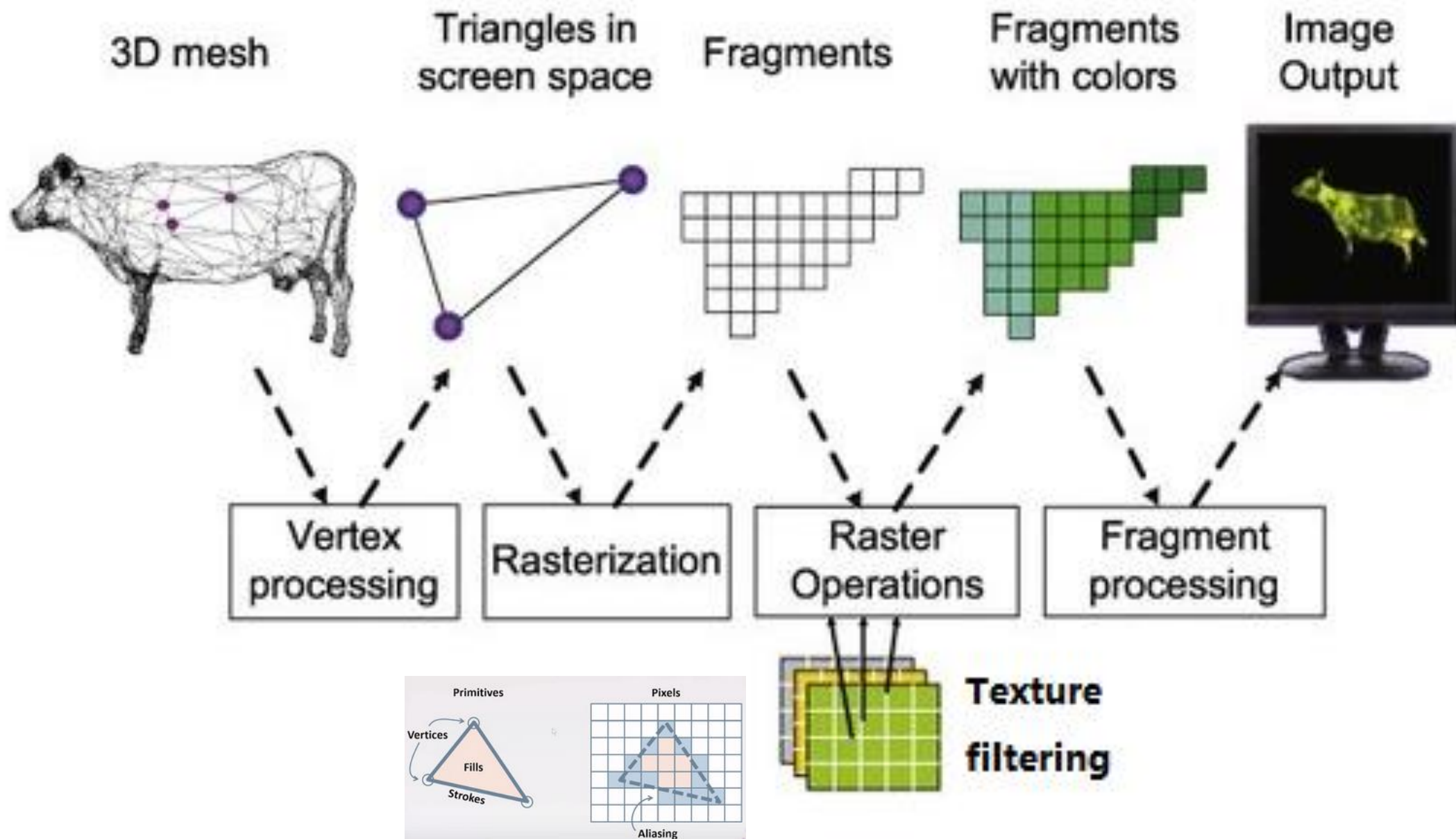
*這裡是針對比較複雜的PIPELINE (HIERARCHICAL PIPELINE)*

# Issues with pipelines 空轉的原因

## (all deal with time per stage)



- **Load balance** (across stages) 各級要處理資料數量不一樣
  - 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) 同一級某些資料花的時間比較多
  - A given stage takes more time on some inputs than others
  - Example
    - 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** 各級算的時間不一樣
  - A stage may require a long latency operation (e.g., texture access)

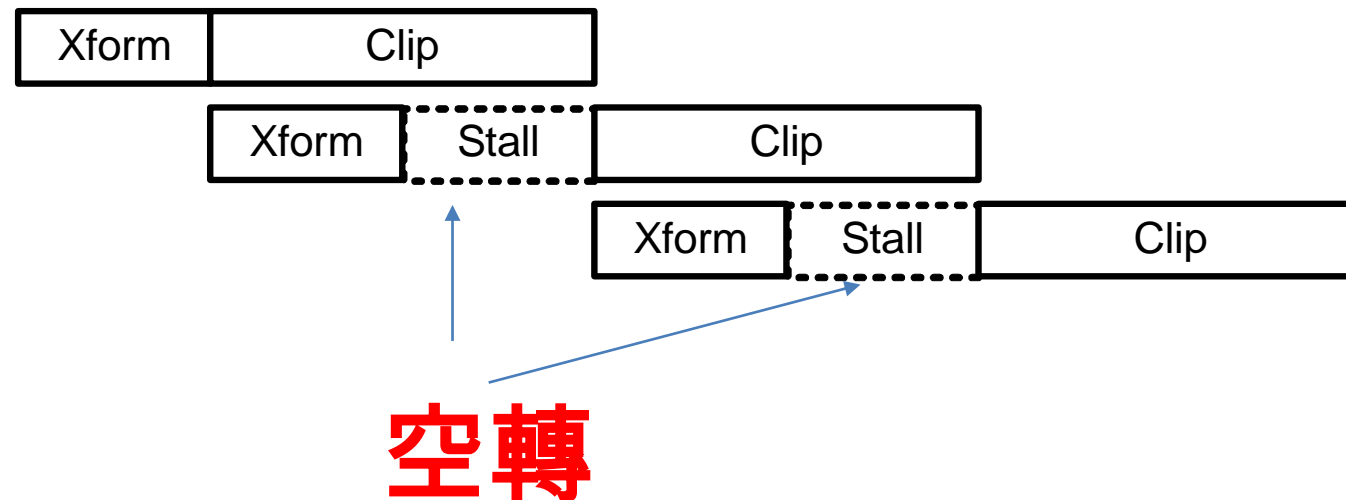


三角形的栅格化方法



# 1. Load Balancing Pipelines

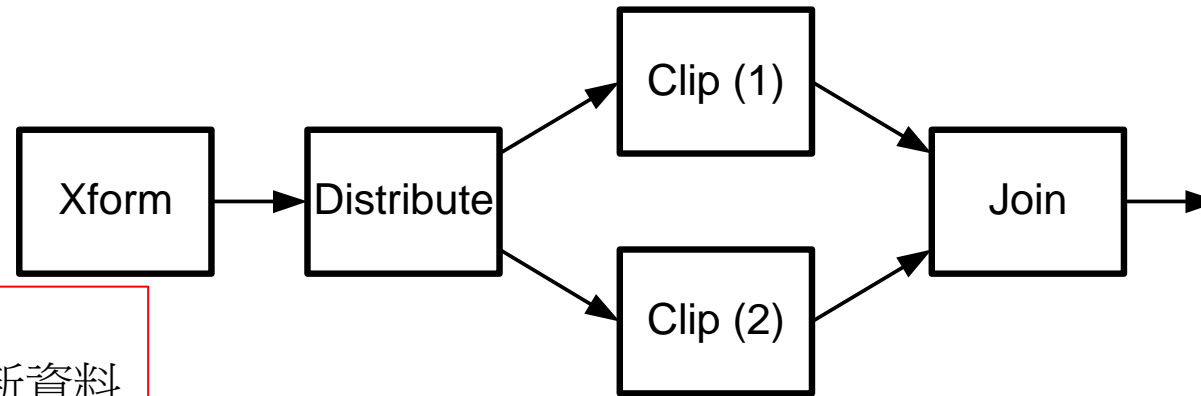
- Suppose transform takes 2 cycles and clip 4 cycles
- Clip is a 'bottleneck' pipeline stage
- Xform unit is busy only half the time



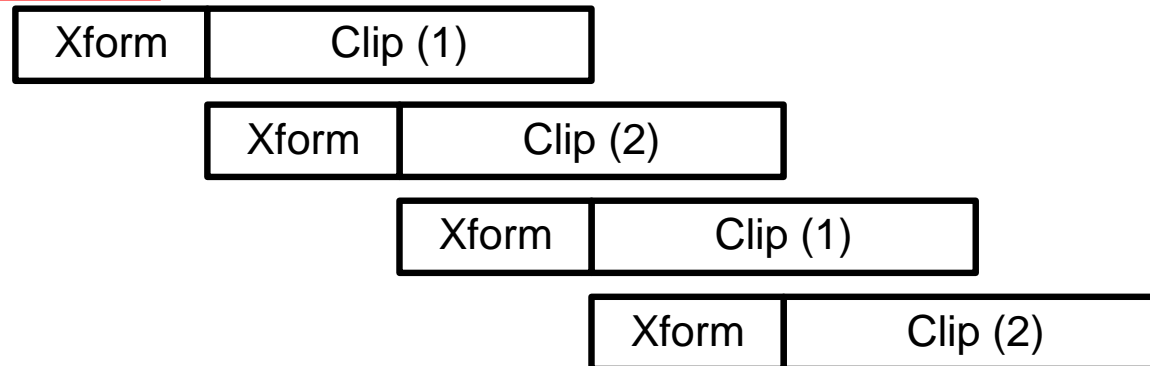
# Load Balancing Solutions 解法:慢的要加快

## 1 – Parallel copies of slow unit

Both Distribute and Join modules are required to make this parallel processing structure work.



平行兩套，相當於  
每2個cycle就可以處理一筆新資料  
符合Xform輸出速度

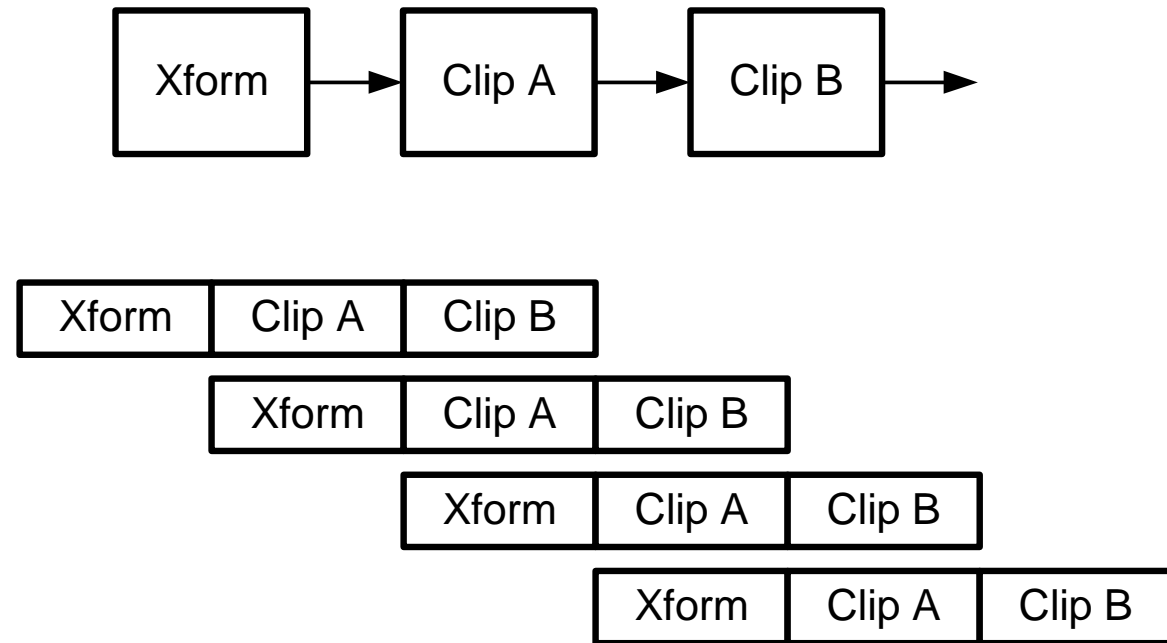


# Load Balancing Solutions 解法:慢的要加快

## 2 – Split slow **pipeline** stage

If timing bottleneck modules can be divided evenly,  
fully pipelining can be achieved without any stall.

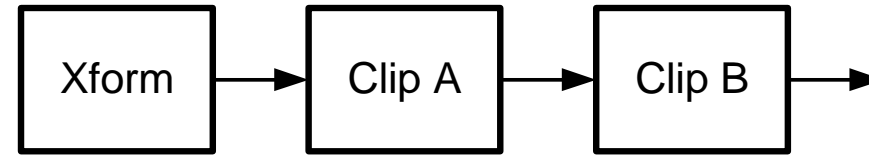
要讓Clip(4cycle)速度與Xform(2cycle)一樣，  
就把**Clip分成兩級pipeline**，相當於  
每級也是2個cycle就可以處理一筆新資料  
符合Xform輸出速度



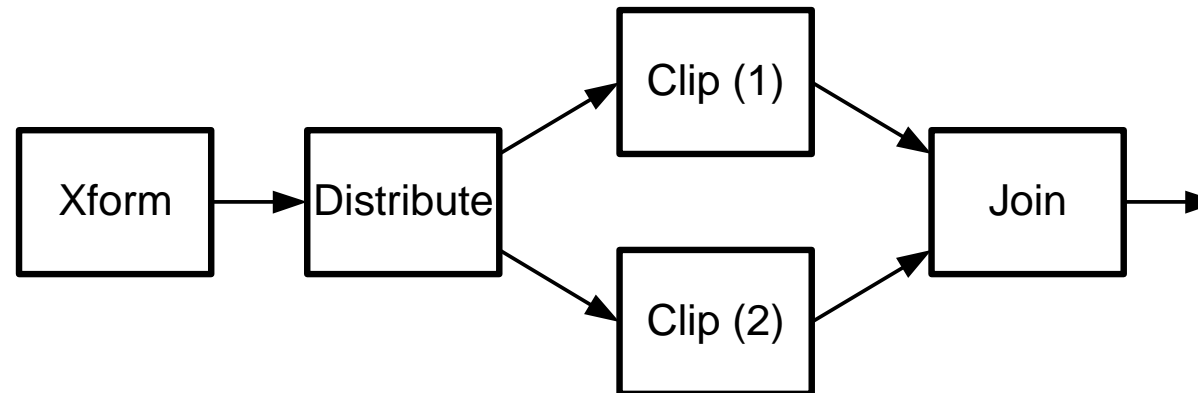
# When is it better? to split? Or to copy?

## Throughput and latency are the same

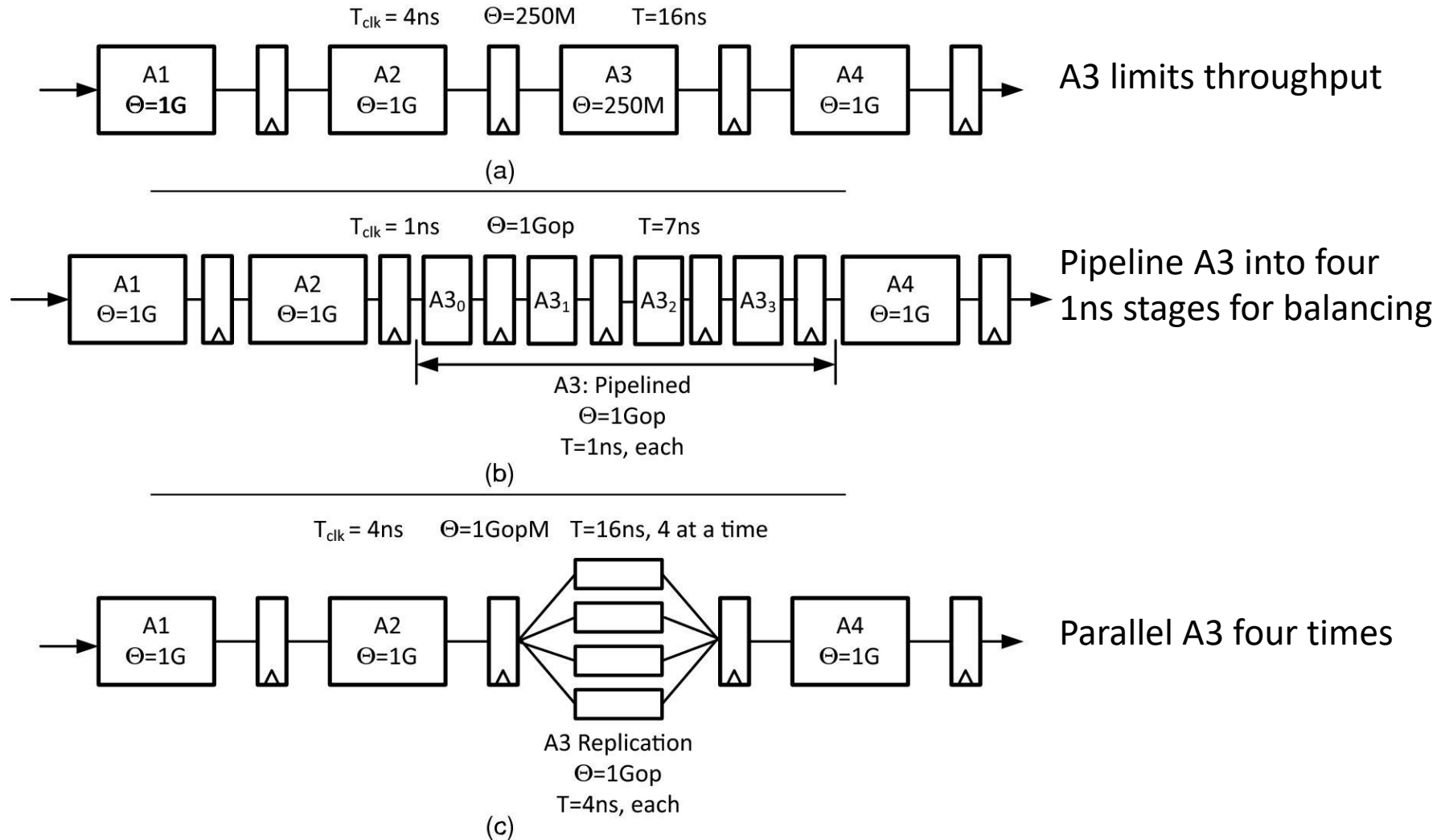
Less area but long latency



More area but less latency

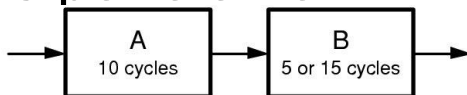


# Load Balancing



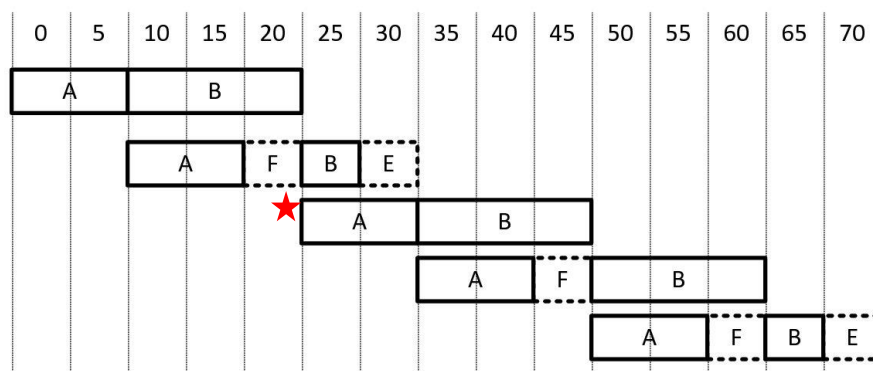
## 2. Variable load 解法: FIFO or double buffer

- Stage A always takes 10 cycles.  
Stage B takes 5 or 15 cycles – averages 10 cycles  
Pipeline averages \_\_\_\_\_ cycles per element



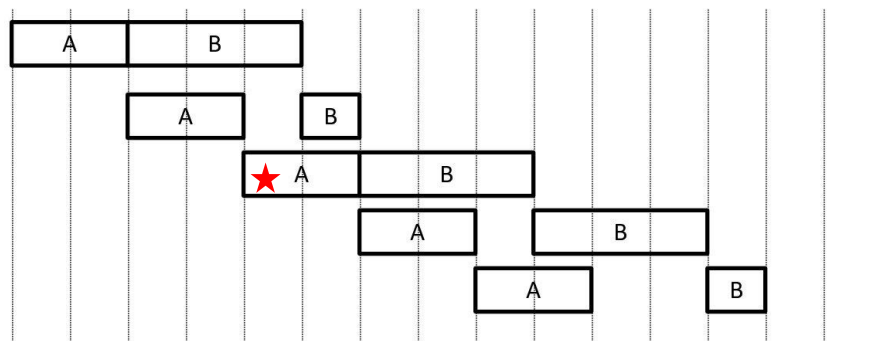
(a)

Cycle



(b)

(b) Only a single register sits btw A and B, caused A or B to stall when one is not ready



(c)

(c) With FIFO to eliminates stalls

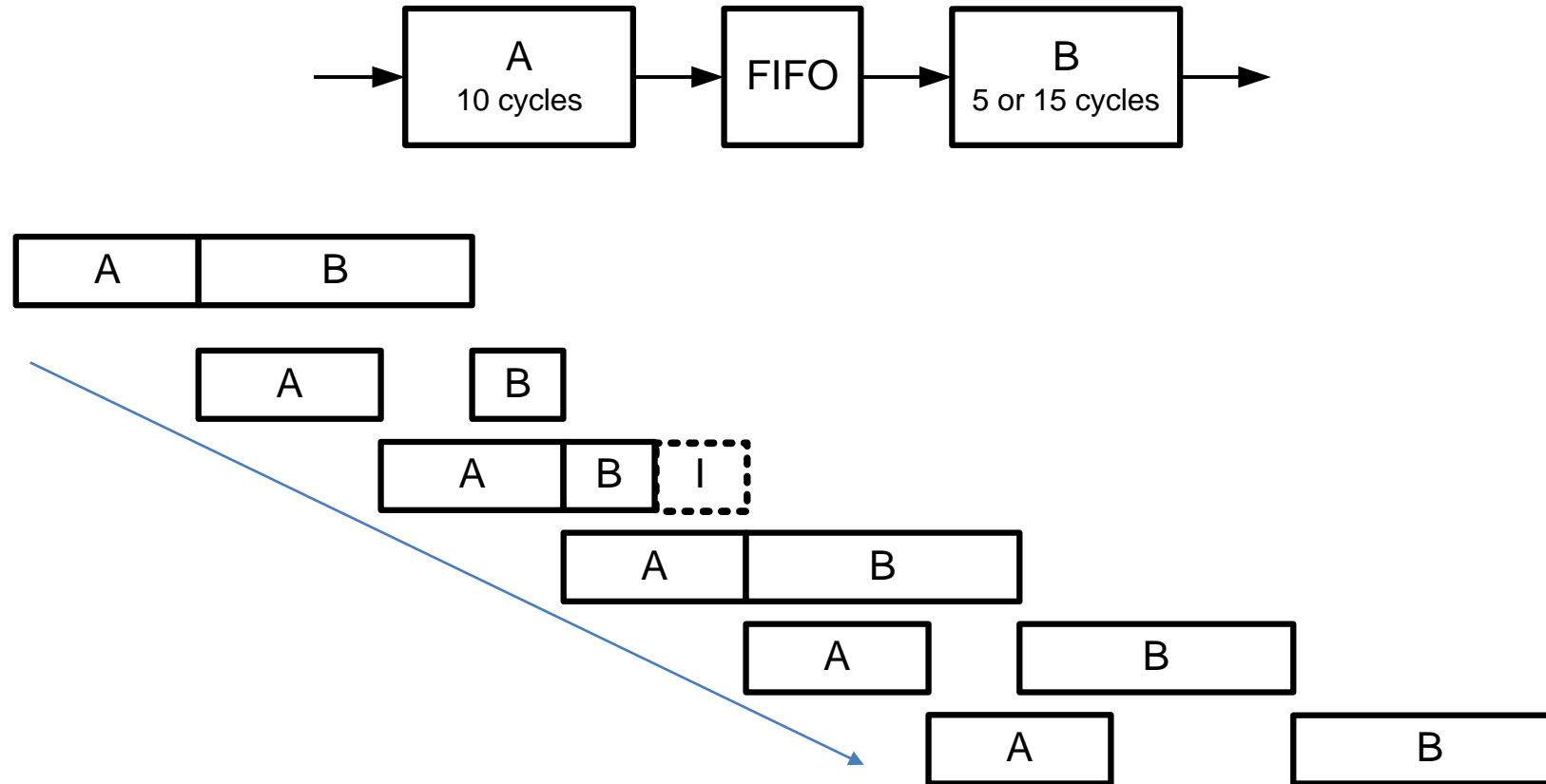
AB間放一個pipeline register  
A 或B還沒算完，就要等

不等待作法: AB間放 FIFO  
(變形: double buffer)  
不管B有沒有算完，A都繼續  
算新資料，A算完的就先存在  
FIFO，B直接從FIFO拿資料算

# Elastic Pipelines

A FIFO between stages decouples timing

Allows stages to operate at their 'average' speed

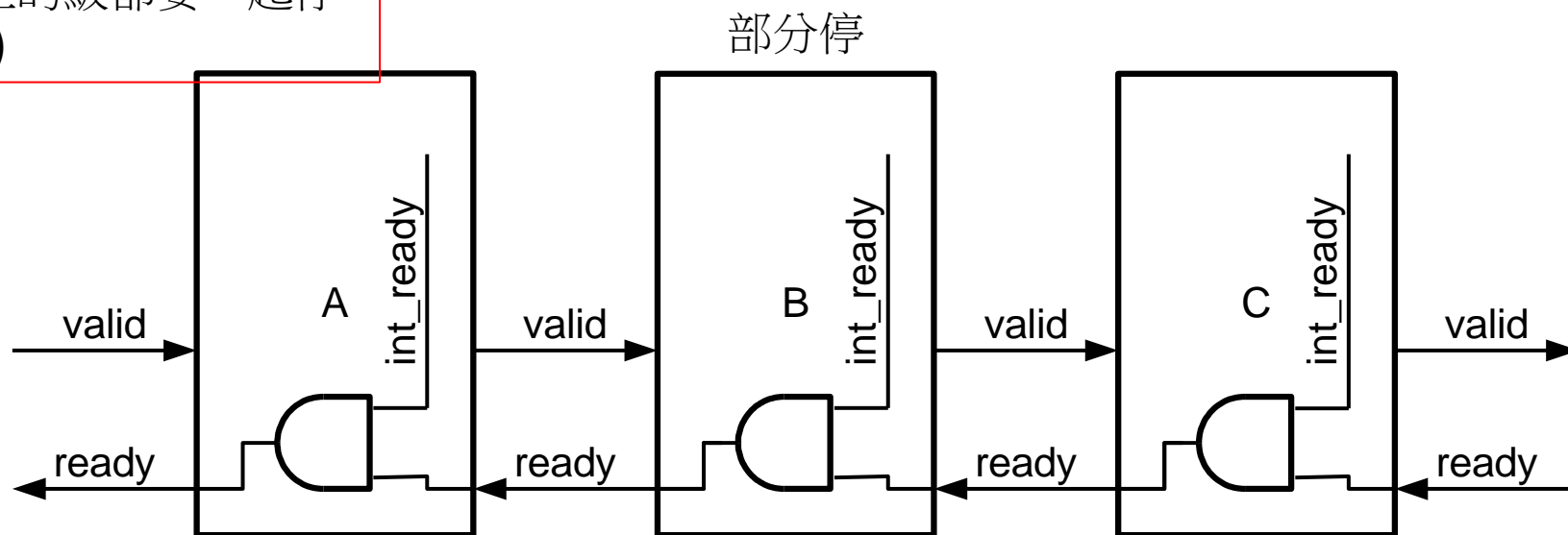


### 3. Stalling a rigid pipeline

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

如何停pipeline?

部分停 (被停以上的級都要一起停  
或是一級一級停)

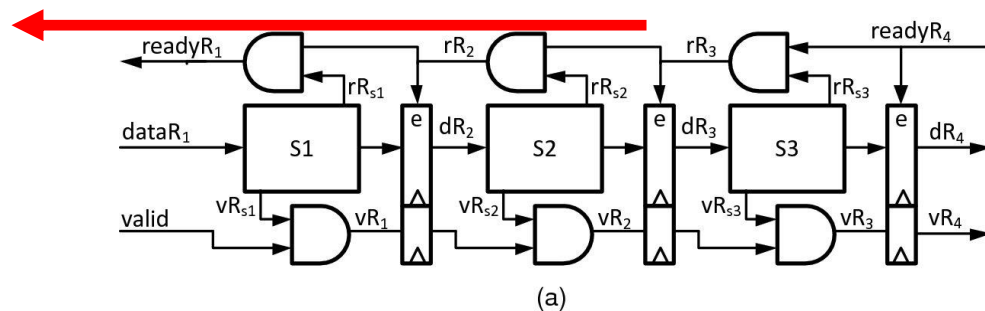


Q. What if we stopped all stages, not just upstream stages? 全部停?上游停?

Q. How does the delay of this structure *scale* with the number of stages? 上面停止電路的問題?

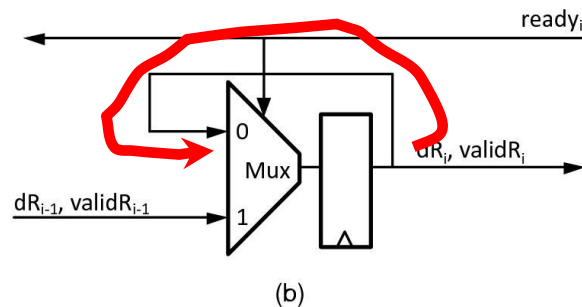


# Pipeline with Global Stall Signals



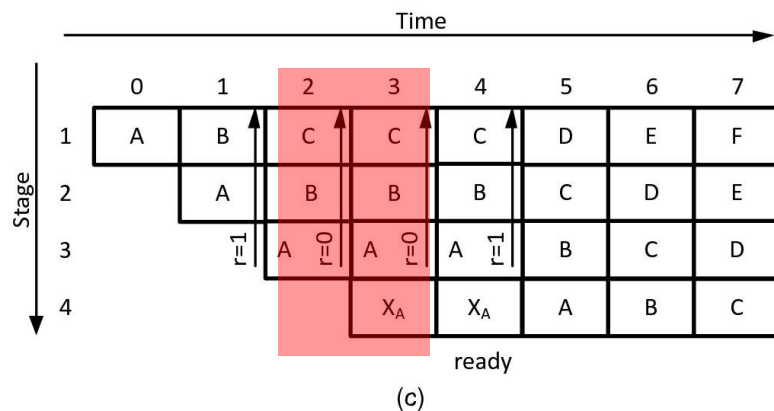
Stall 訊號一次傳給所有上游stage

Pipeline with stall signals  
(注意critical path)



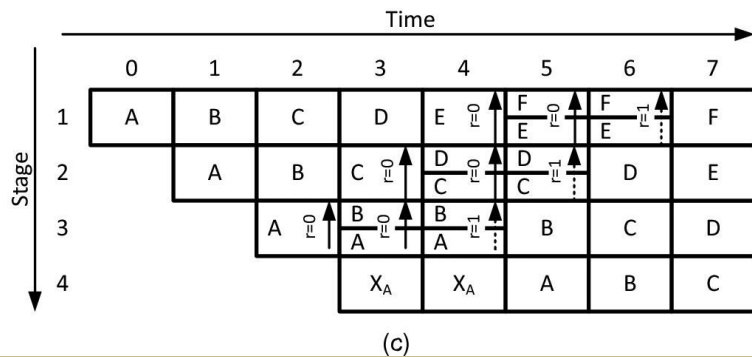
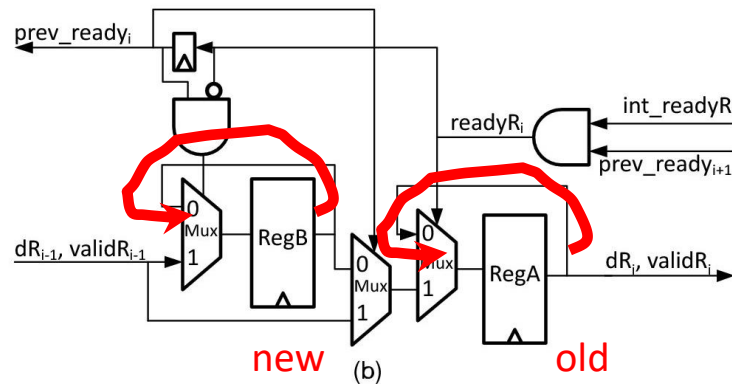
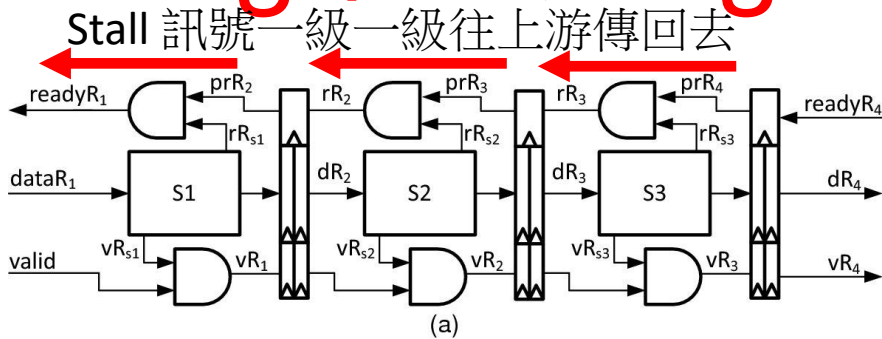
使pipeline register 輸入不改變，維持pipeline 的計算值

Stall logic  
(注意如何暫存值，如何正常運行)



Sample execution

# Pipeline with Local Stall Signals (Double Buffering 兩套Registers)



- Clock the ready signals into a DFF at every stage and propagate it upstream one cycle at a time
  - To avoid losing, need **double buffer**: one for old data and one for new data arriving (注意critical path)

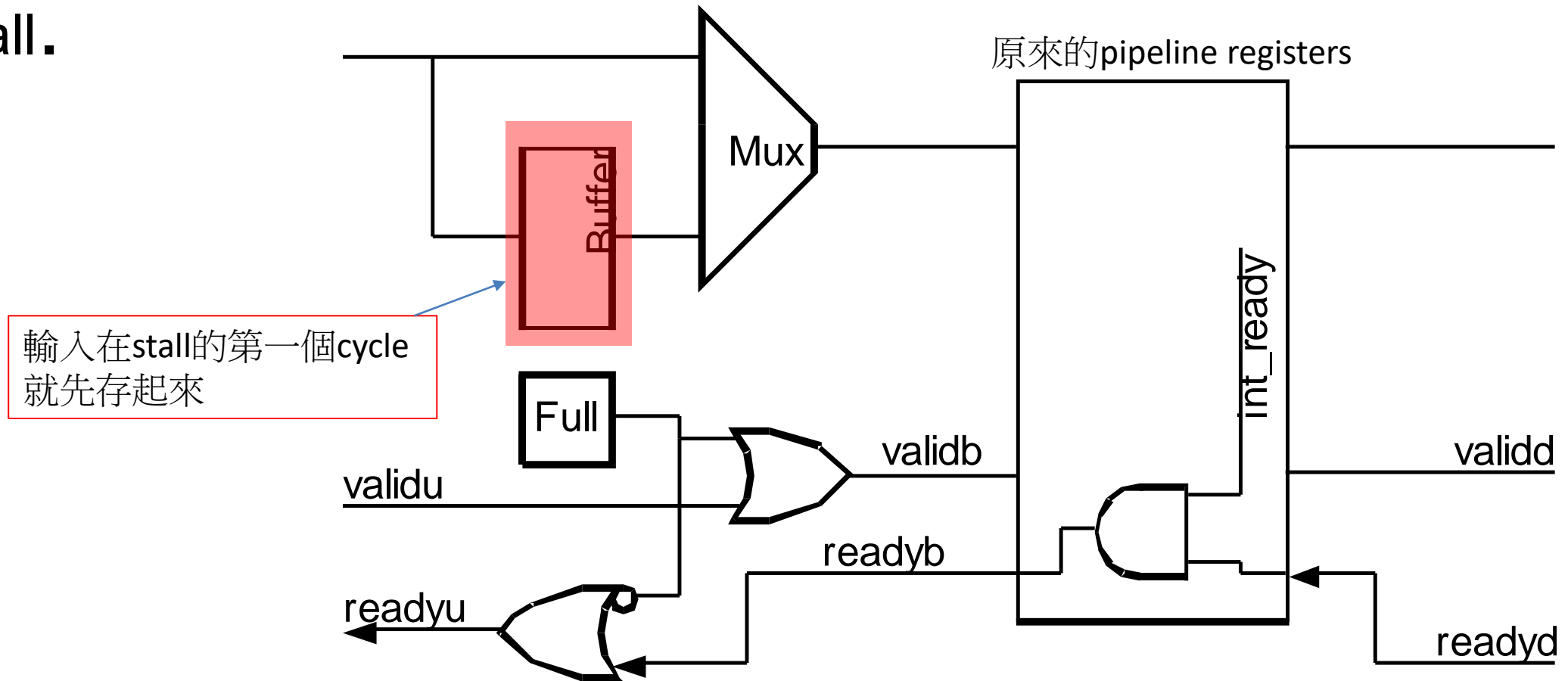
When stage  $i$  not ready, it sets internal ready to 0, stall old data (stage  $i-1$  has not received “not ready”). Also force another mux control to 0, stall new input data

把原來這級的pipeline register 輸入先暫存，前面一級新產生的資料也要暫存

部分停，慢慢停，有些還在算

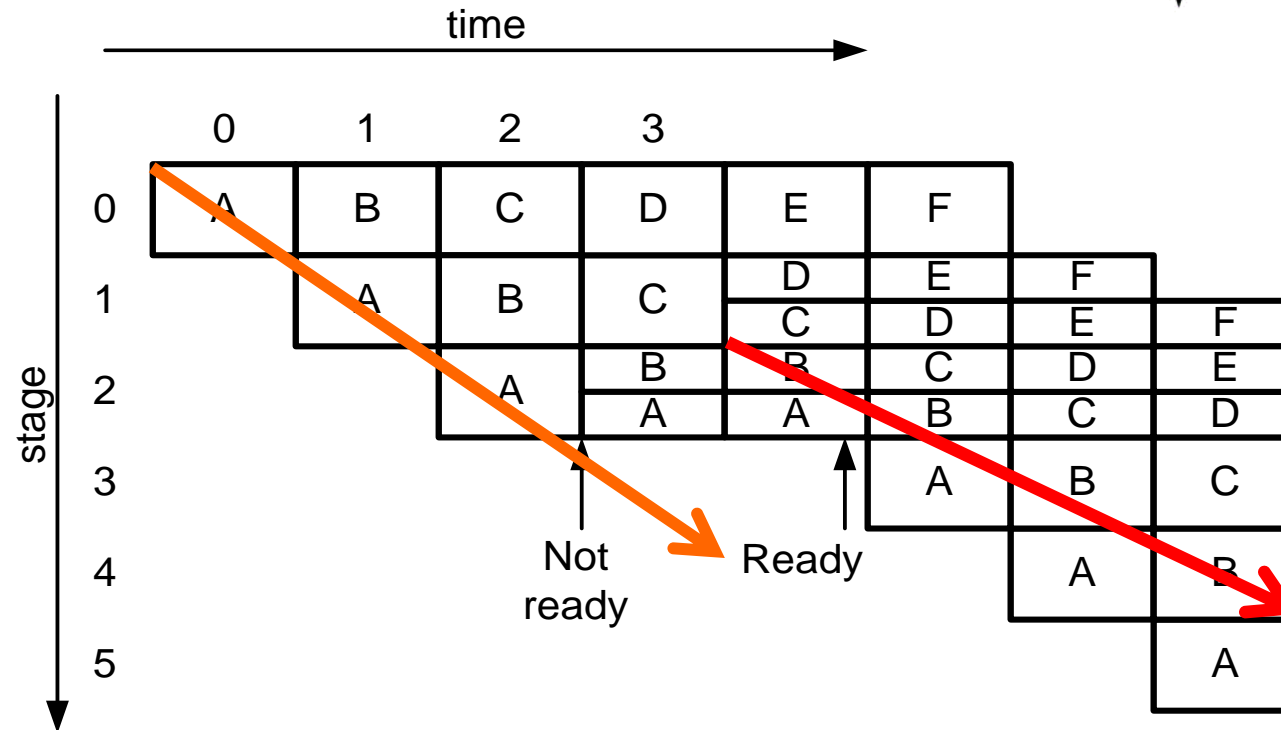
# Double Buffer 另一作法

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

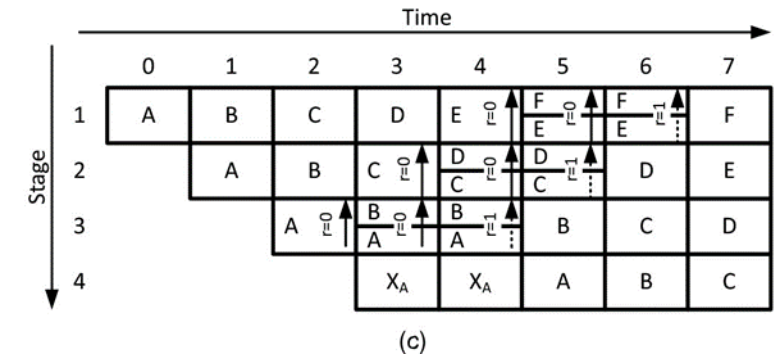


Extra logics need to be added into each pipelined module.  
Buffer size depends on the stall # allowed in specs.

# Double Buffer Timing



Quiz: when Not ready is activated, All modules need to buffer inputs till Ready appears. How to reach this goal?



Stall (not ready)

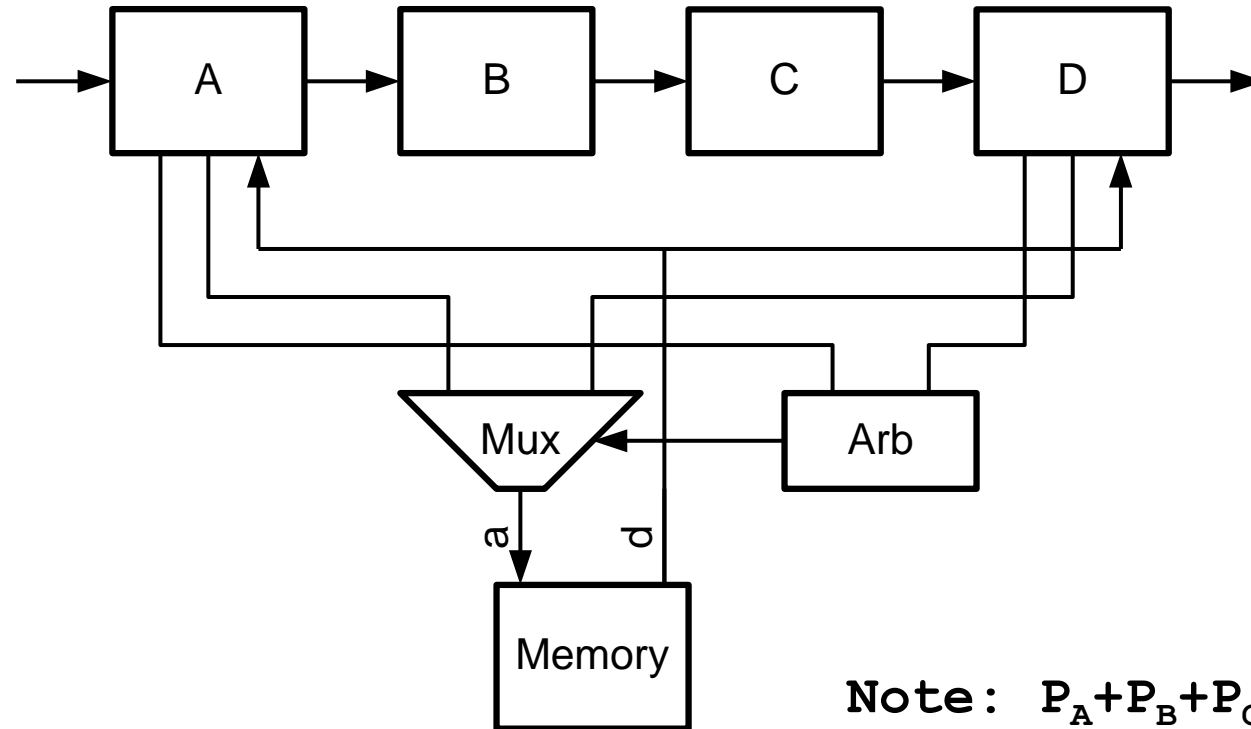
部分停，慢慢停，有些還在算

Ready

慢慢一級一級放開

# 4. Resource Sharing

- Suppose two pipeline stages need to access the same memory



Note:  $P_A + P_B + P_C + P_D = 1$

How would you set the priority on the arbiter?

優先給下游stage，要被停的stage 會比較少，也避免deadlock

若給上游stage,下游沒資料，也是要停，會造成大量停，而下游要不到資料，無法解除stall，會造成pipeline無法進行下去

# Pipeline Summary

- 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
    - \*also related to low-power design by scaling down Supply voltage.

# Pipeline Hazard

- More at computer organization
  - RAW ... (Read after Write)

sub ~~\$2~~, \$1, \$3      sub \$2, \$1, \$3  
and \$12, ~~\$2~~, \$5      nop  
add \$14, ~~\$2~~, \$2      => nop  
sw \$15, 100(\$2)      and \$12, \$2, \$5  
                         add \$14, \$2, \$2  
                         sw \$15, 100(\$2)