Computer Systems II

Li Lu

Room 319, Yifu Business and Management Building

Yuquan Campus

li.lu@zju.edu.cn

https://person.zju.edu.cn/lynnluli

Pipelining

Pipelining?

You already knew!

Pipelining

- What is pipelining?
- How is the pipelining implemented?
- What makes pipelining hard to implement?

Cafeteria:

- Did you wait until all others finish?

Cafeteria: Order

Cafeteria: Pay

Cafeteria: Enjoy

- While some others are ordering or paying

Cafeteria: Observations?

- Besides eating…

Cafeteria: Observations?

- co-use **dependent function areas** speed up the dining process of all

Cafeteria: Observations?

- Individual perspective?

…… a potentially very, very long queue

fastest if only one to serve

order | enjoy | pay

Cafeteria: Observations?

- Average - faster
- Individual – slower *(service time)*

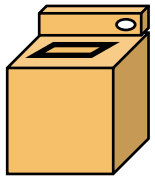
  but much less time in queue

- Individual – faster: queue + service

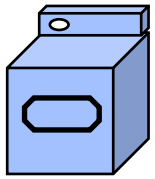
(classic) laundry example

Laundry Example

- **A**nn, **B**rian, **C**athy, **D**ave
- Each has one load of clothes to
- wash, dry, fold



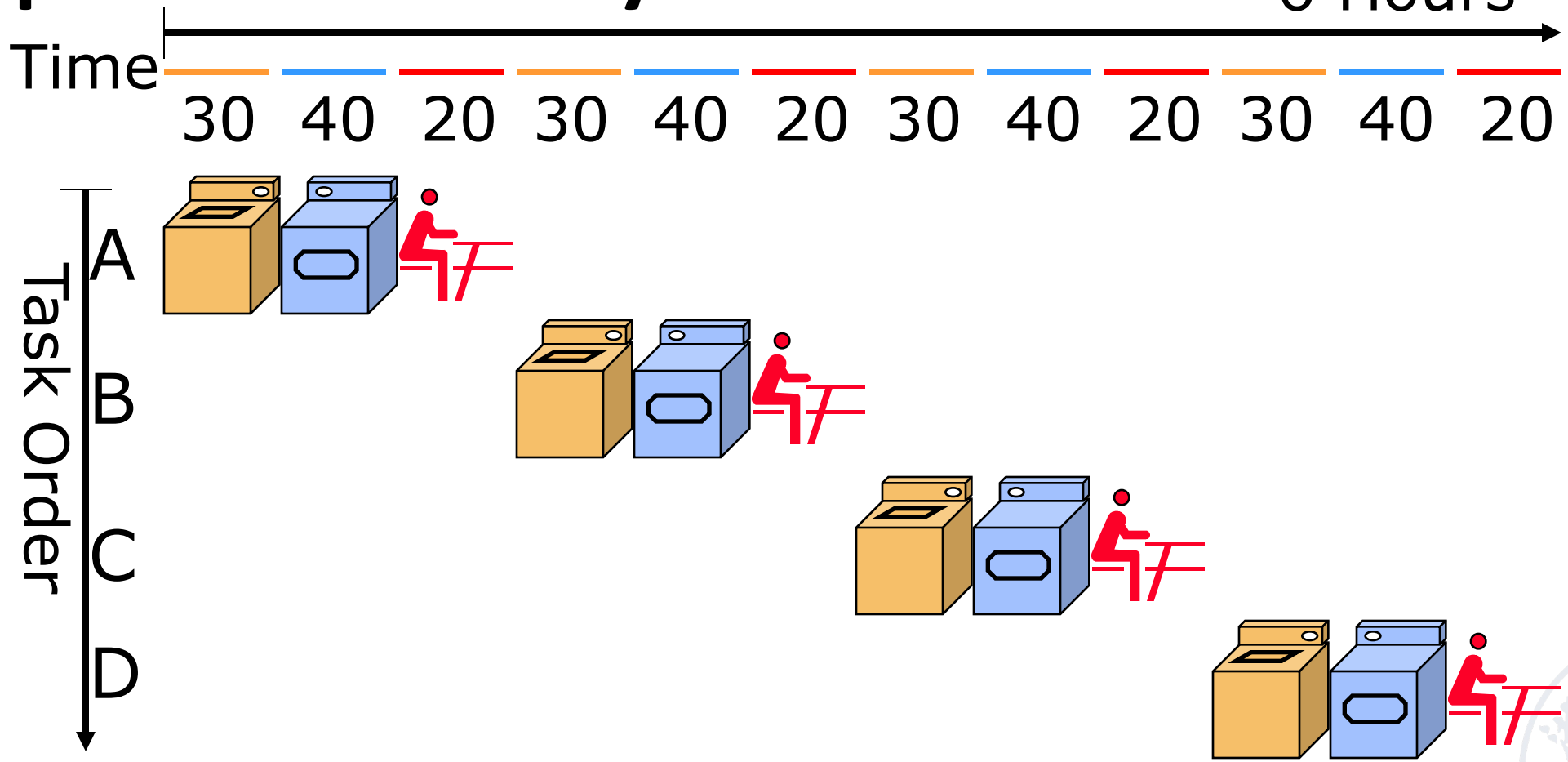
washer
30 mins



dryer
40 mins

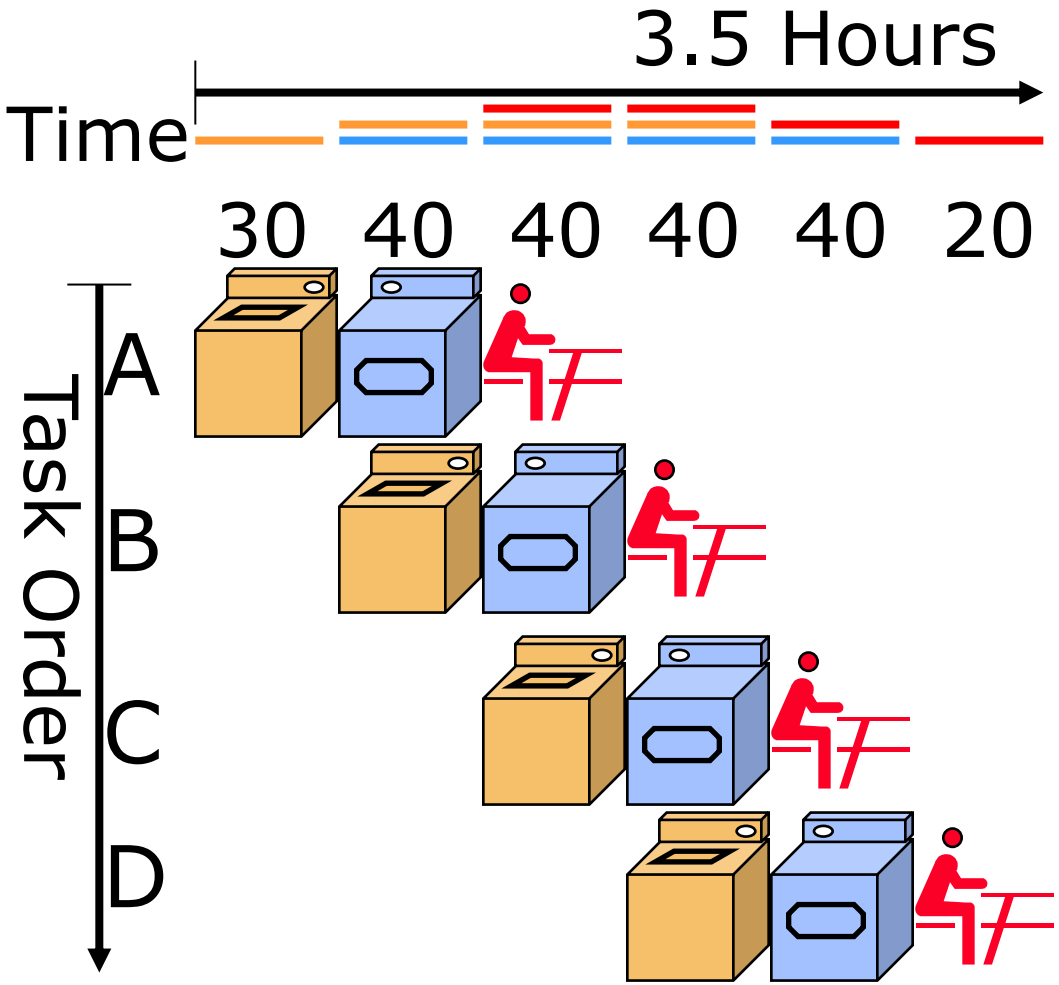
folder
20 mins

Sequential Laundry

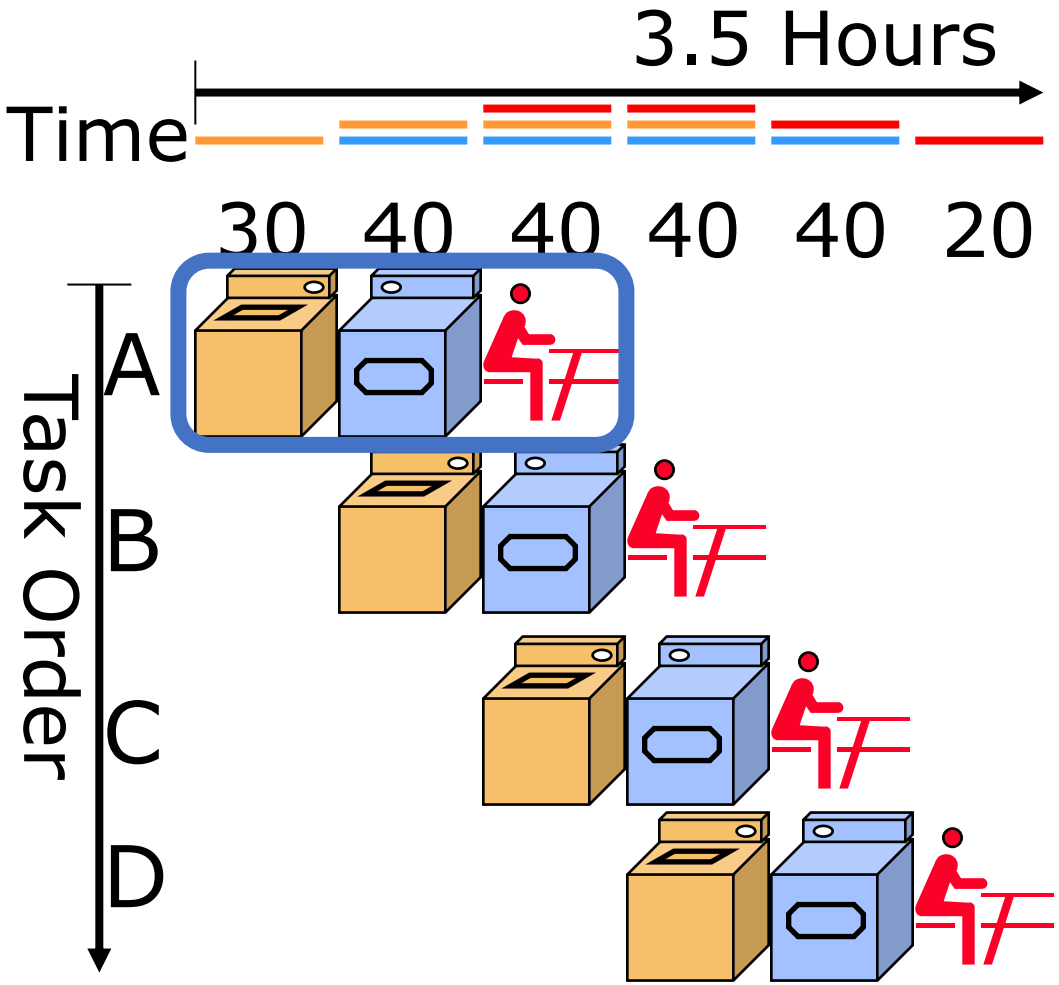


What would you do?

Pipelining Laundry



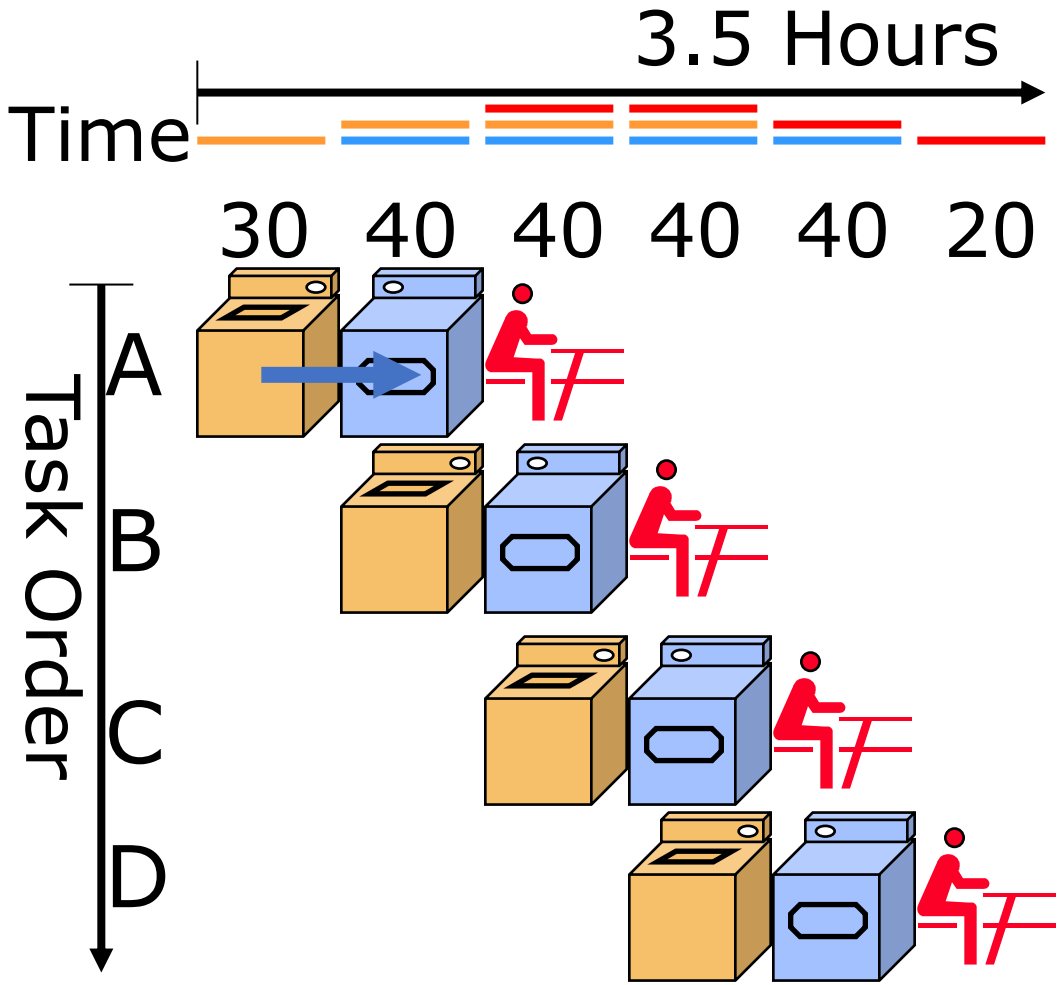
Pipelining Laundry



Observations

- A task has a series of stages;

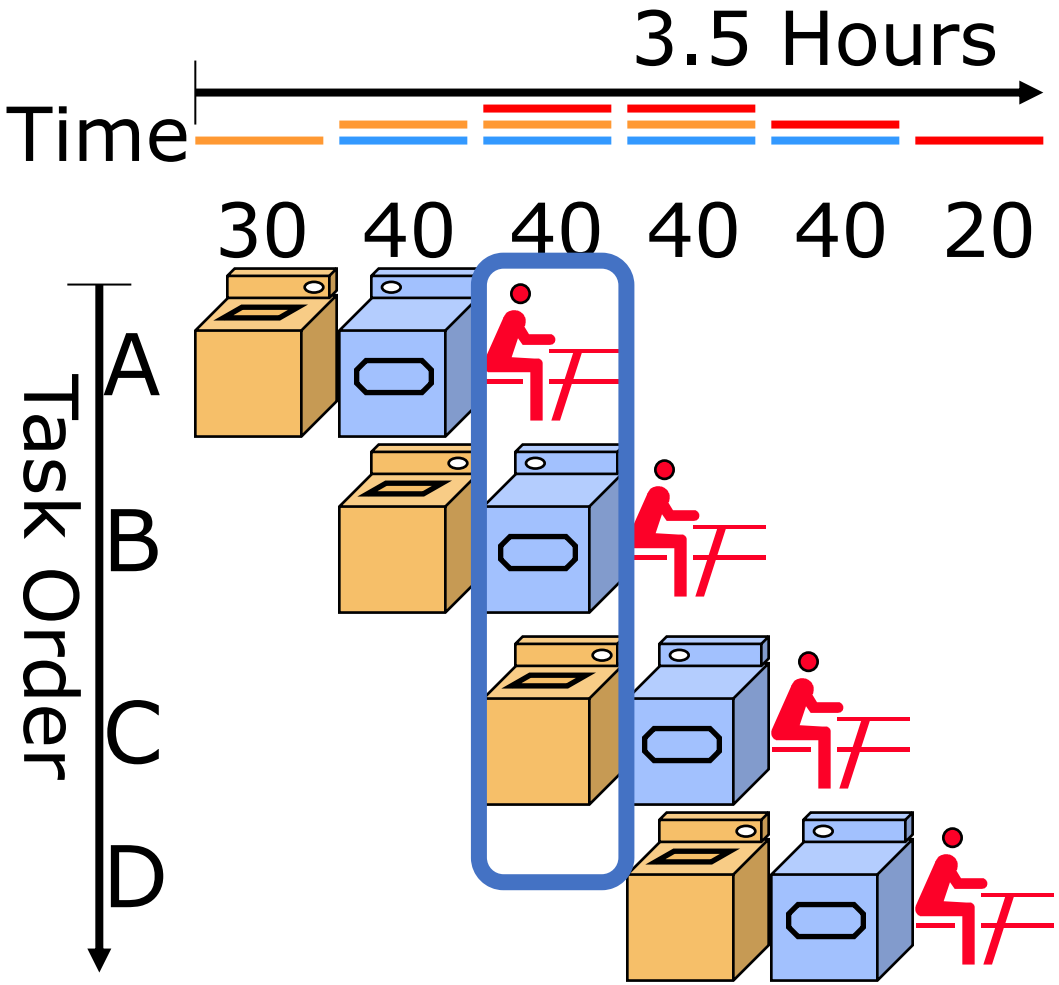
Pipelining Laundry



Observations

- A task has a series of stages;
- Stage dependency:
  e.g., wash before dry

Pipelining Laundry

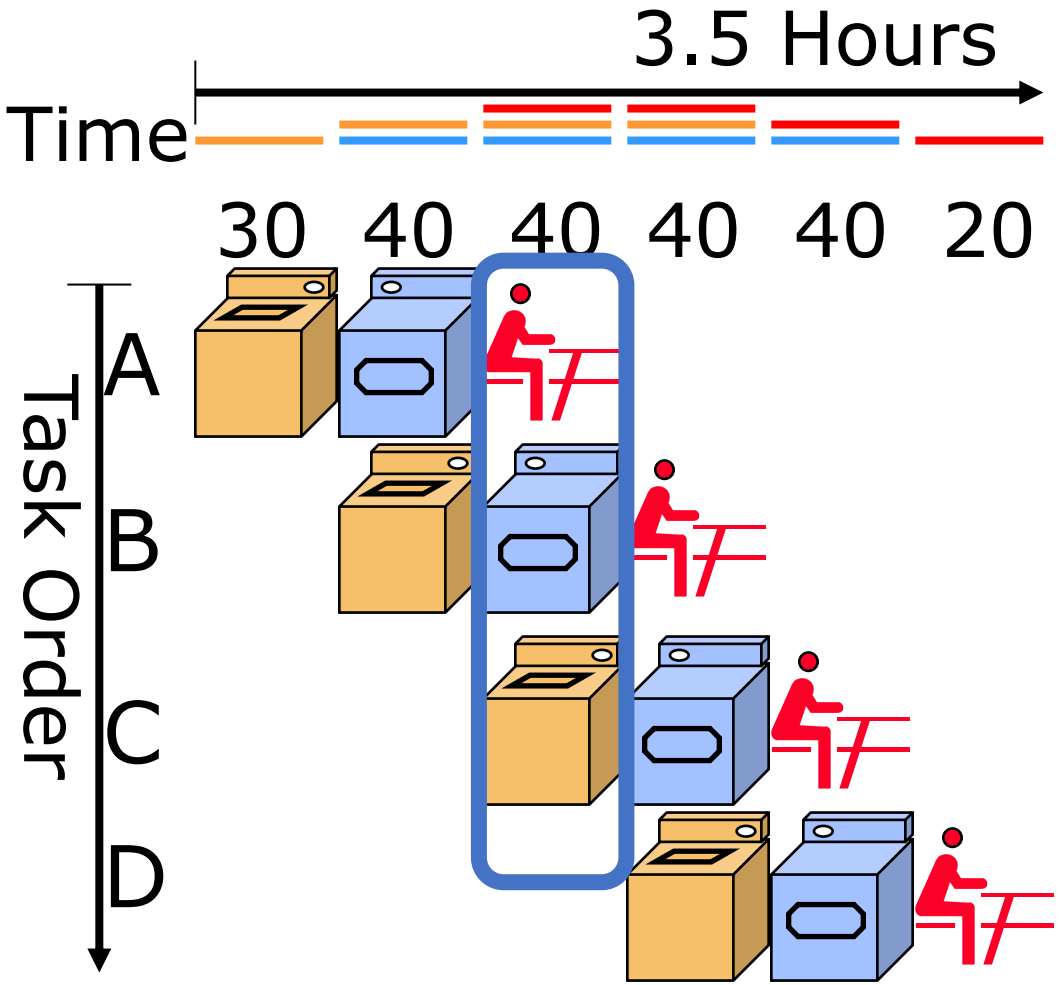


Observations

- A task has a series of stages;
- Stage dependency:

  e.g., wash before dry;
- Multi tasks with overlapping stages;

Pipelining Laundry

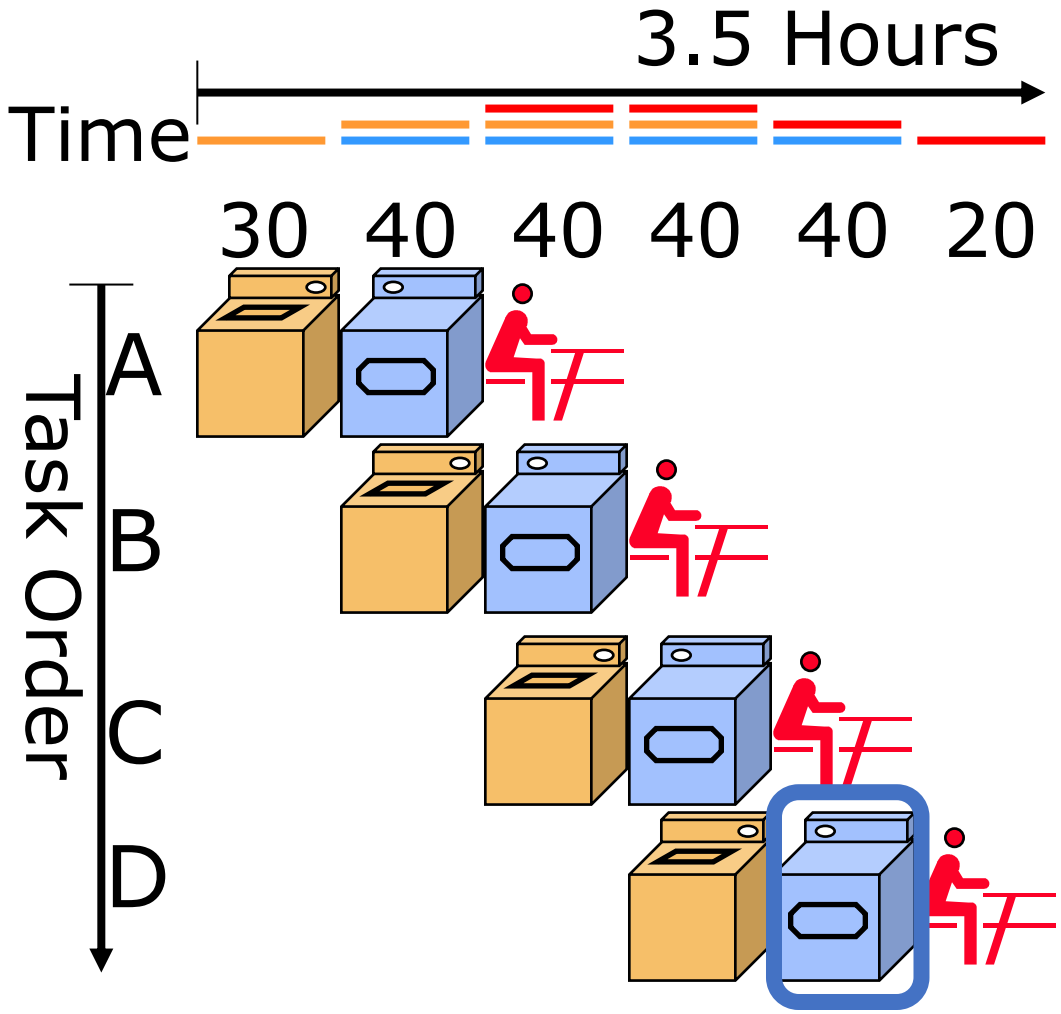


Observations

- A task has a series of stages;
- Stage dependency:

  e.g., wash before dry;
- Multi tasks with overlapping stages;
- Simultaneously use diff resources to speed up;

Pipelining Laundry

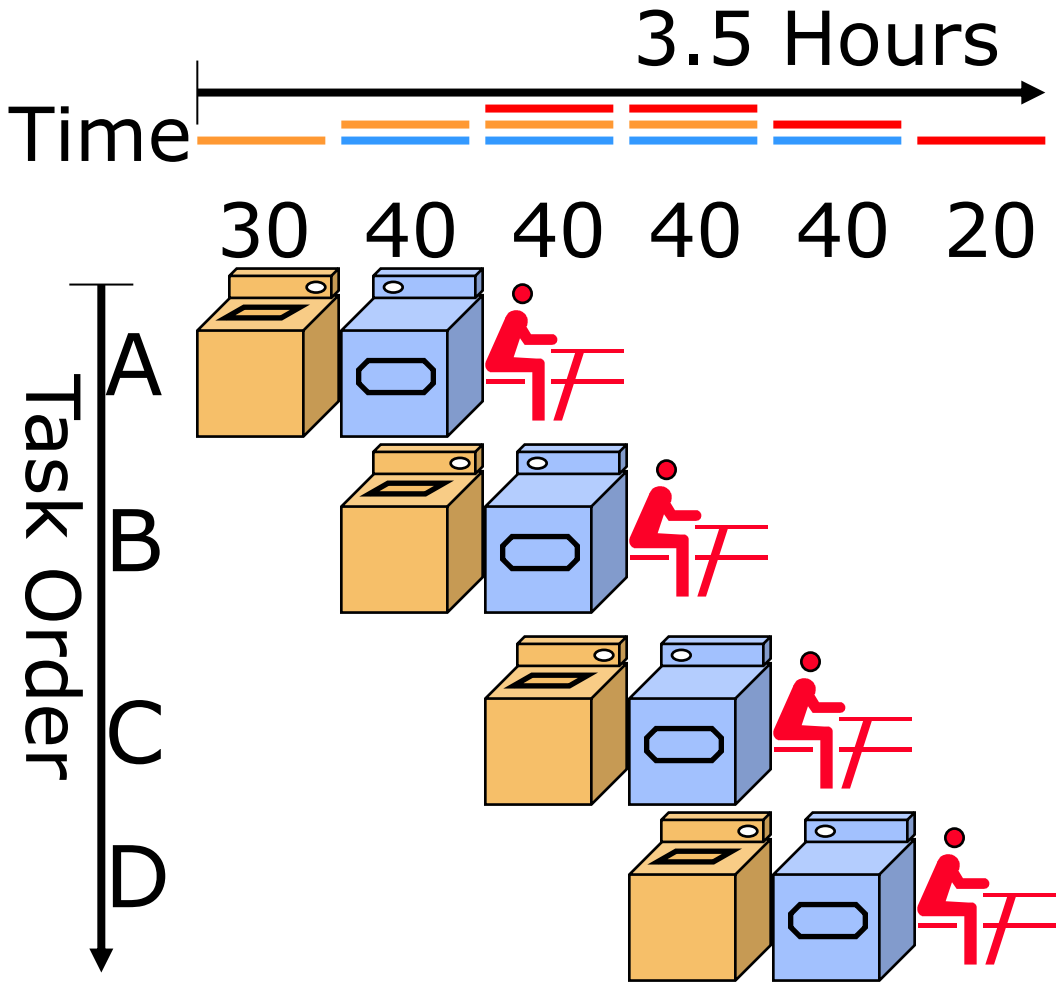


Observations

- A task has a series of stages;
- Stage dependency:

  e.g., wash before dry;
- Multi tasks with overlapping stages;
- Simultaneously use diff resources to speed up;
- Slowest stage determines the finish time;

Pipelining Laundry

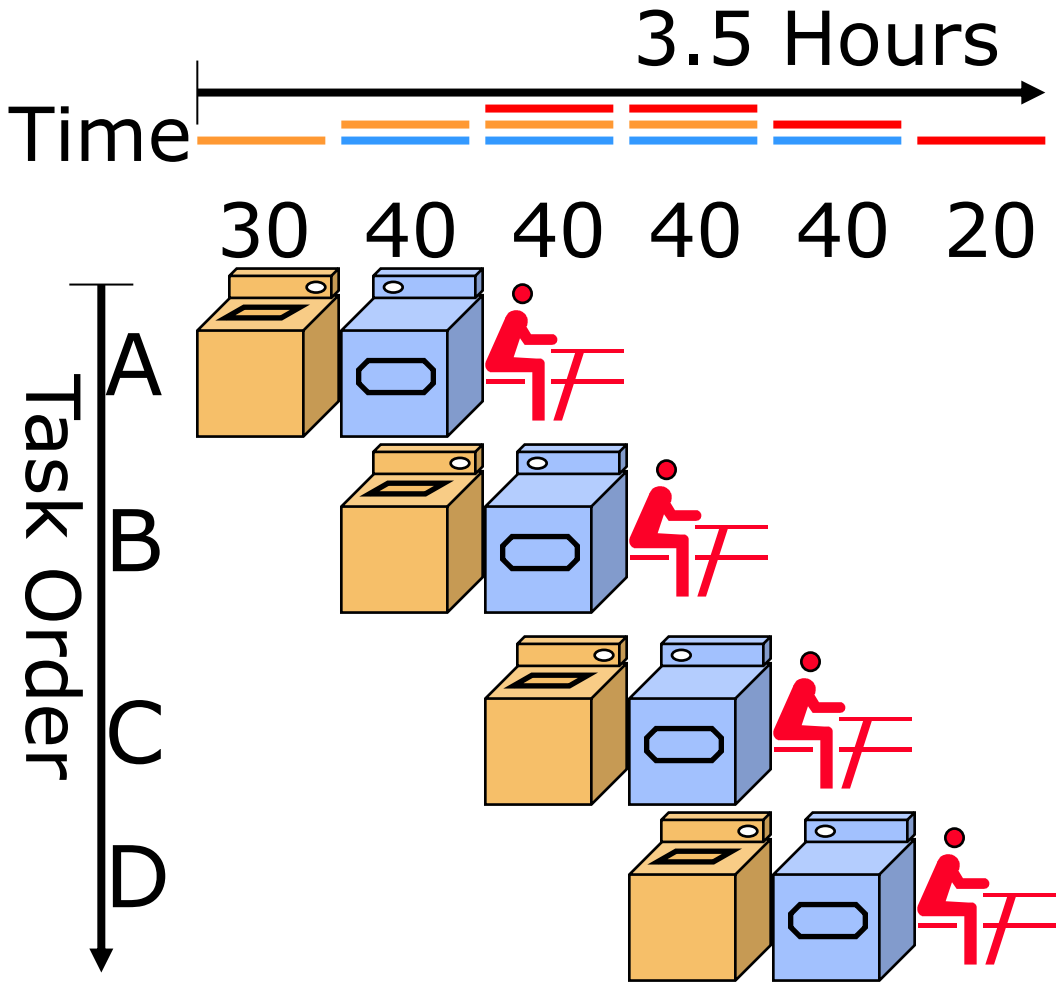


Observations

- No speed up for individual task;

  e.g., A still takes 30+40+20=90

Pipelining Laundry



Observations

- No speed up for individual task;

  e.g., A still takes 30+40+20=90

- But speed up for average task execution time;

  e.g., 3.5*60/4=52.5 < 30+40+20=90

Pipelining Elsewhere: Assembly Line

Auto

Cola

What exactly is pipelining in computer arch?

Speed up the execution of instructions

- Shorten the execution time of each instruction
  - More high-speed devices
  - Better calculation methods
  - Improve the parallelism of each microoperation in the instruction
  - reduce the number of beats needed in the interpretation process

- Reduces the execution time of the entire program (machine language)
  - Through the control mechanism, the interpretation of the entire program can be speeded up by means of **simultaneous interpretation of two, multiple or even whole programs**

- *Pipelining is an **implementation** technique whereby multiple instructions are **overlapped** in execution; it takes advantage of **parallelism** that exists among the actions needed to execute an instruction*

- *Today, pipelining is the key implementation technique used to make fast CPUs*

Pipelining

- *"A technique designed into some computers to increase speed by starting the execution of one instruction before completing the previous one."*

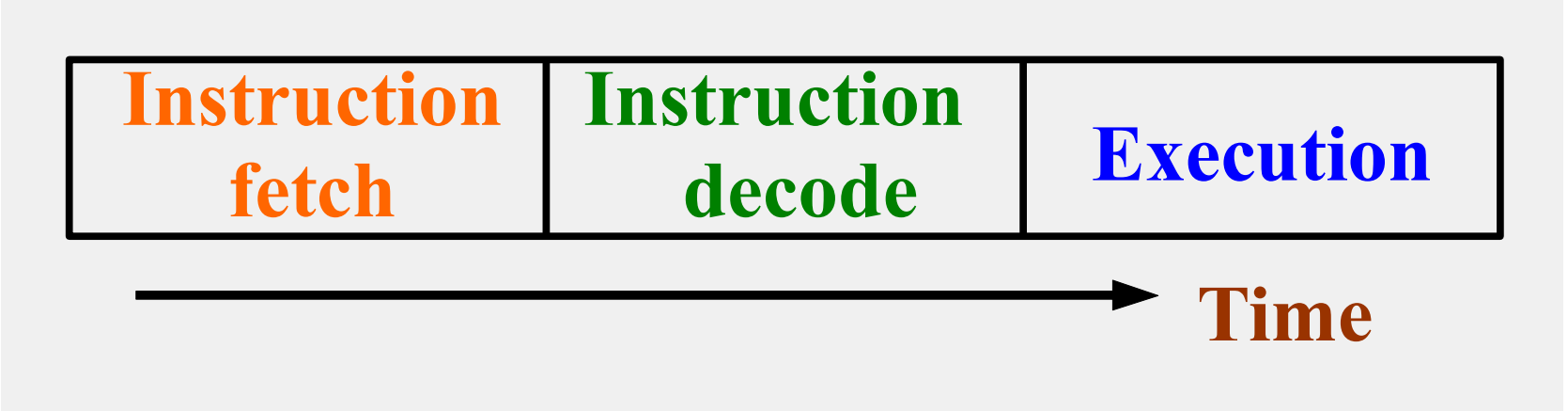
  ----Modern English-Chinese Dictionary

- implementation technique whereby different instructions are overlapped in execution at the same time
- implementation technique to make fast CPUs

Three modes of execution

- Sequential execution
- Single overlapping execution
- Twice overlapping execution

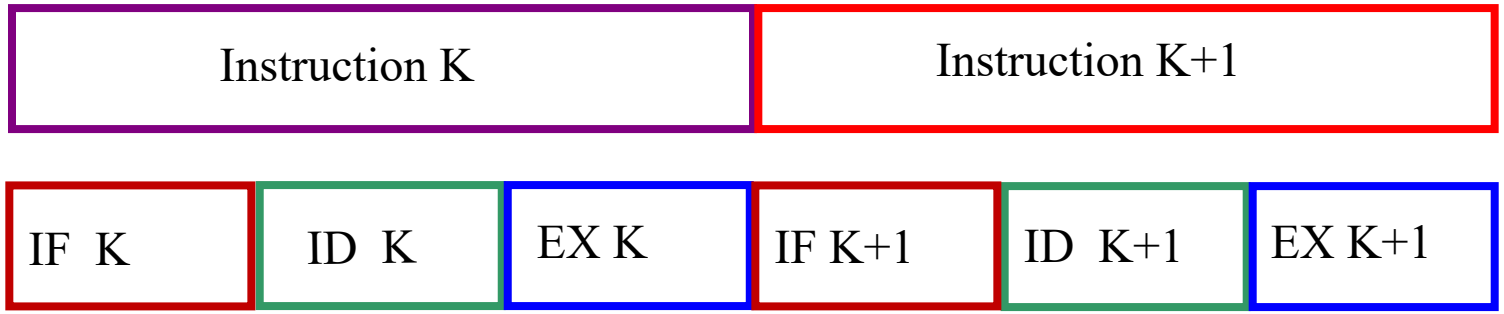
The execution of an instruction is divided into three stages：



The execution of an instruction

Sequential execution

- The execution of the instructions

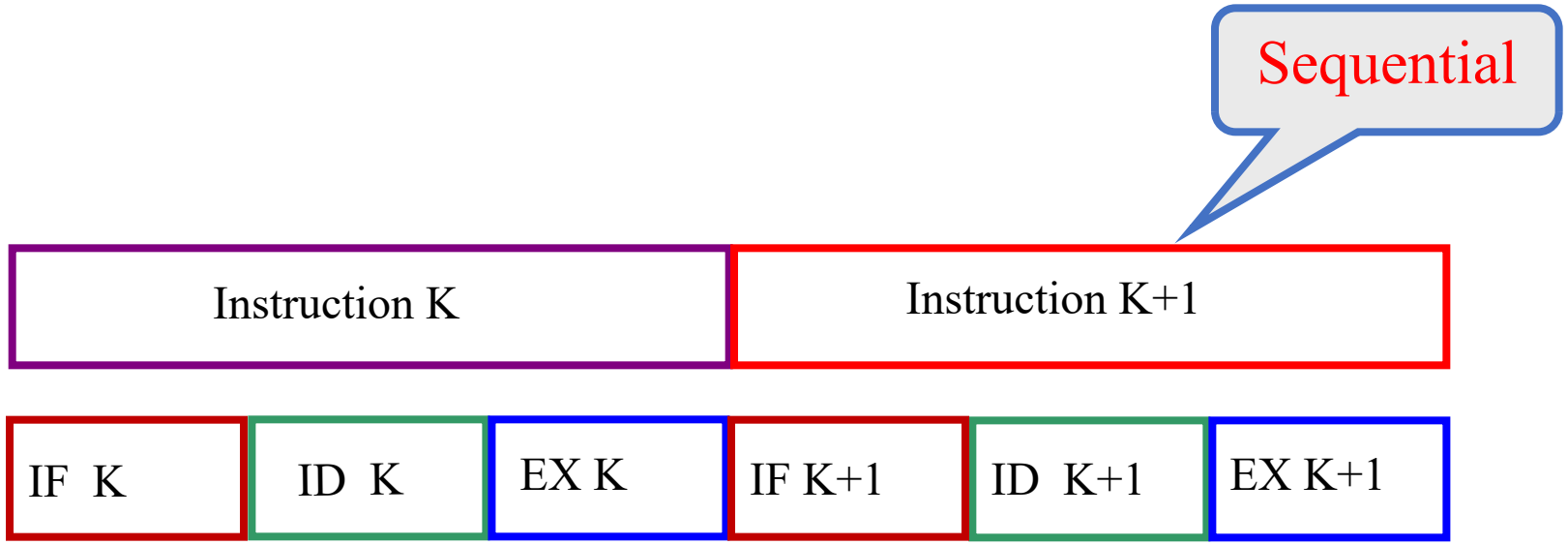


- Instructions are executed sequentially, and each microoperation in the instruction is also executed sequentially

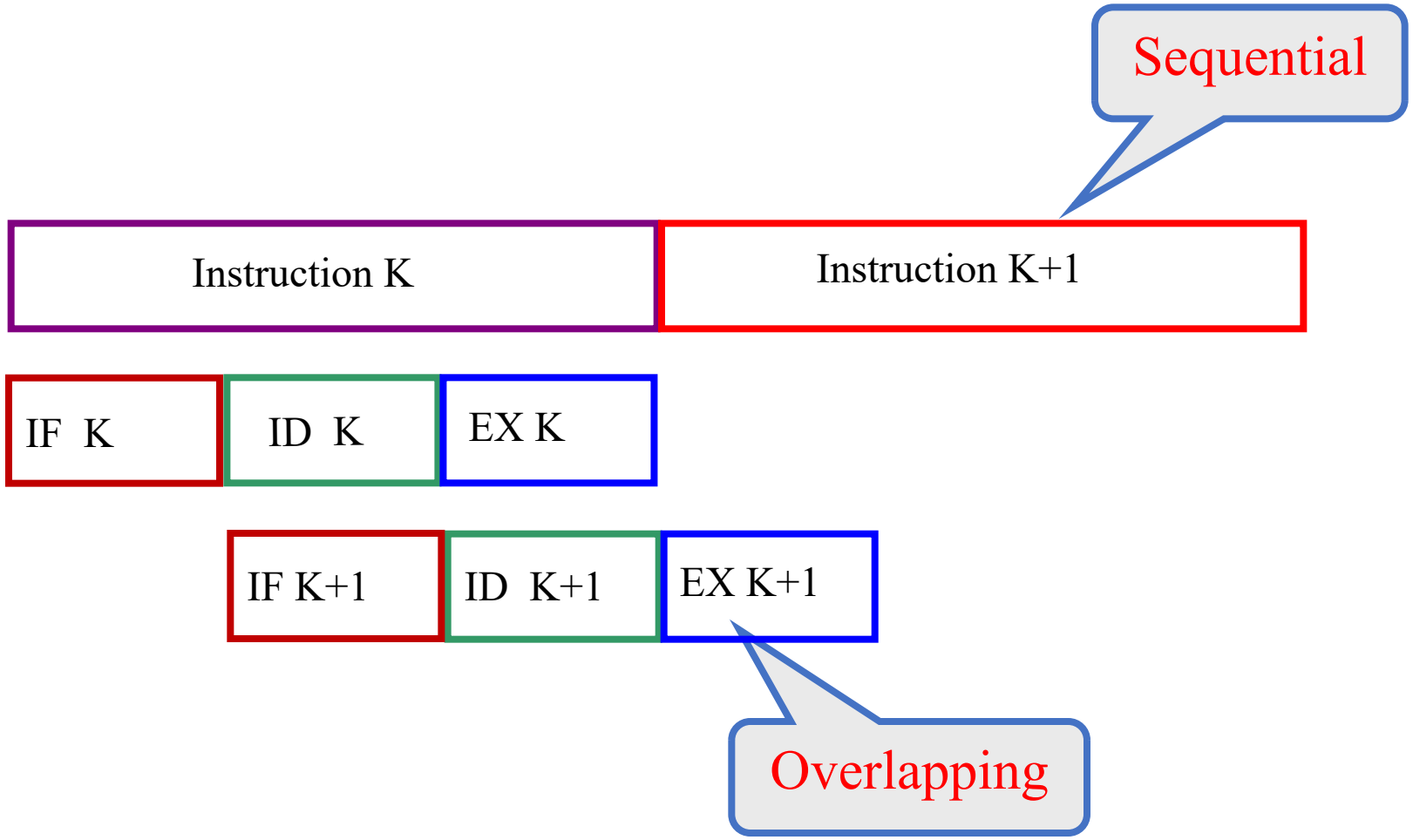
- Execution Time:

$$T = \sum_{i=1}^{n} (t_{IFi} + t_{IDi} + t_{EXi})$$

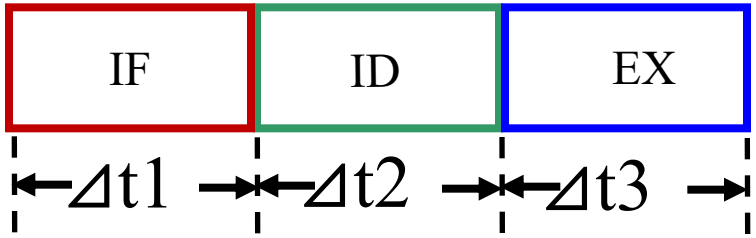
Overlapping execution



Overlapping execution



Overlapping execution

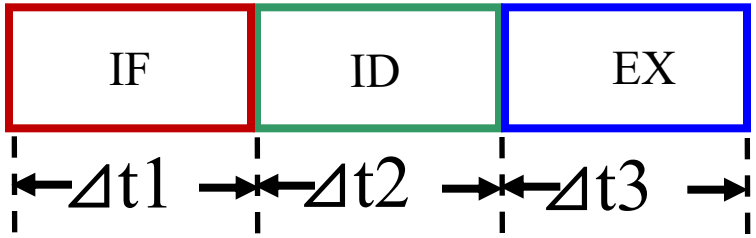


- ⊿t1 =⊿t2 =⊿t3 = ⊿t

- Execute time for *n* instructions in sequence

$$T = \sum_{i=0}^{n} (t_{IFi} + t_{IDi} + t_{EXi}) = 3n\Delta t$$

Overlapping execution

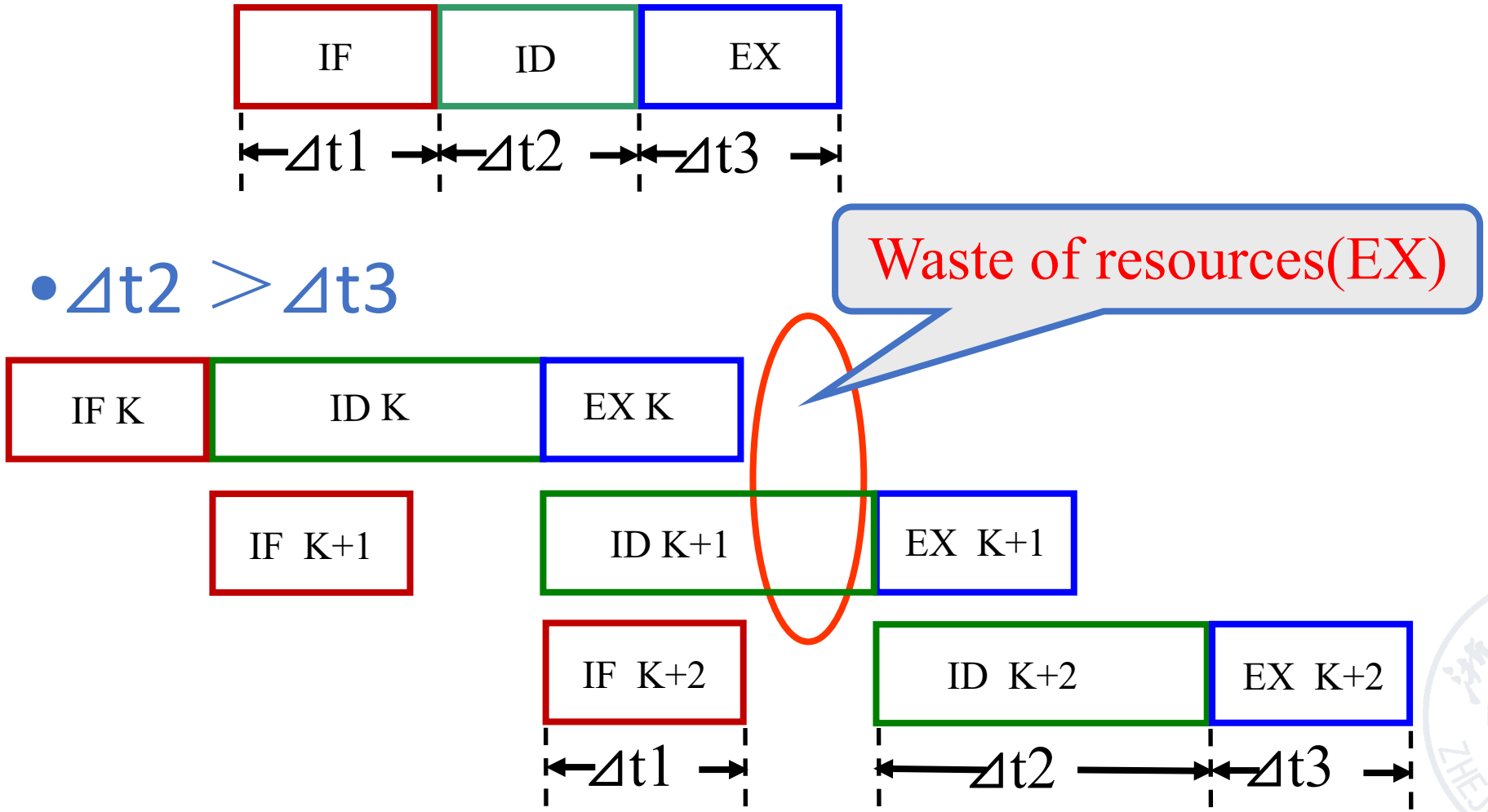


- ⊿t1 =⊿t2 =⊿t3 = ⊿t

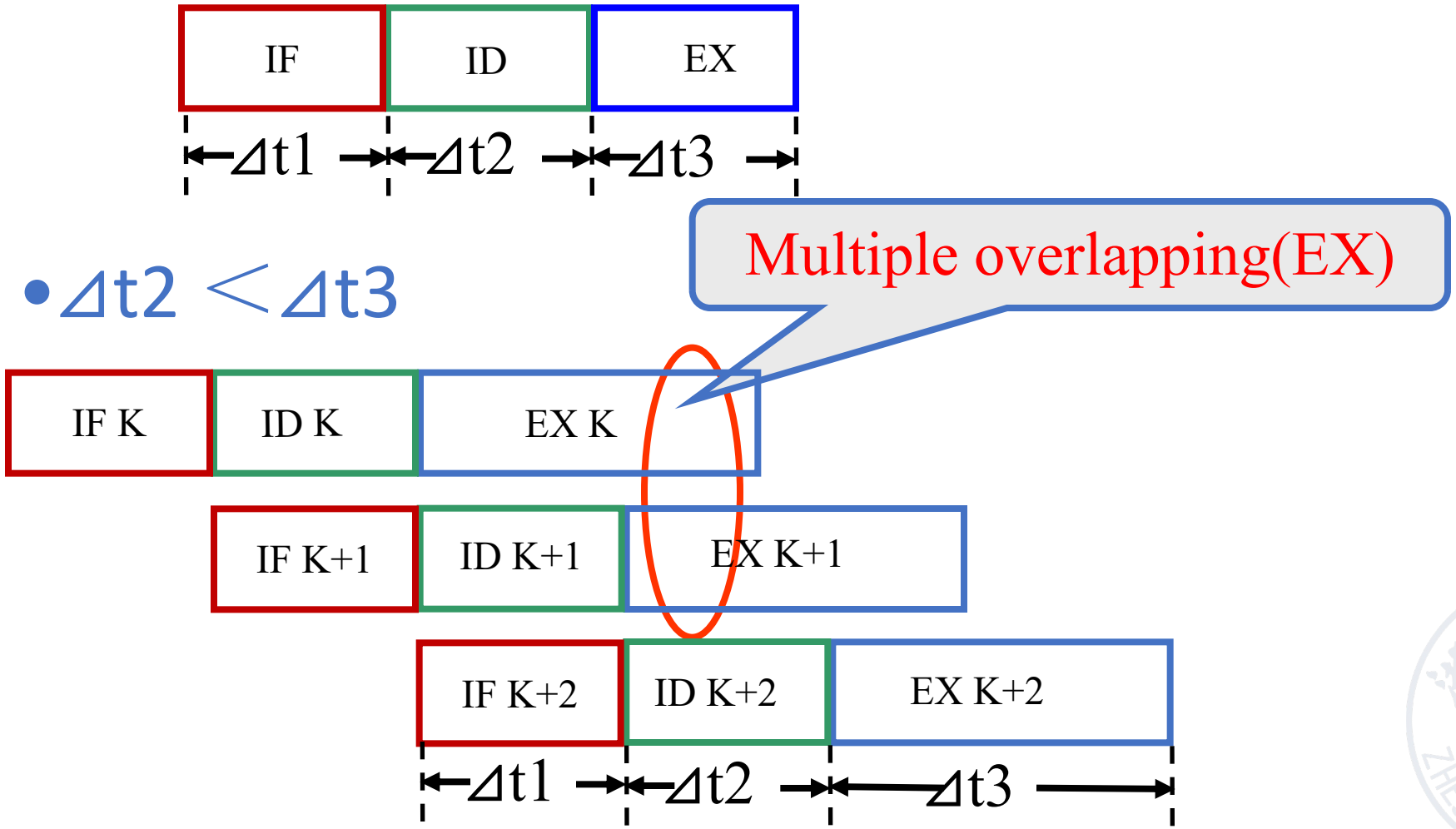
- Execute time for *n* instructions in overlapped sequence

$$T = t_{IFi} + \max(t_{IFi}, t_{IDi}) + (n-2)\max(t_{IFi}, t_{IDi}, t_{EXi}) + \max(t_{IDi}, t_{EXi}) + t_{EXi}$$
$$= (n+2)\Delta t$$

Overlapping execution



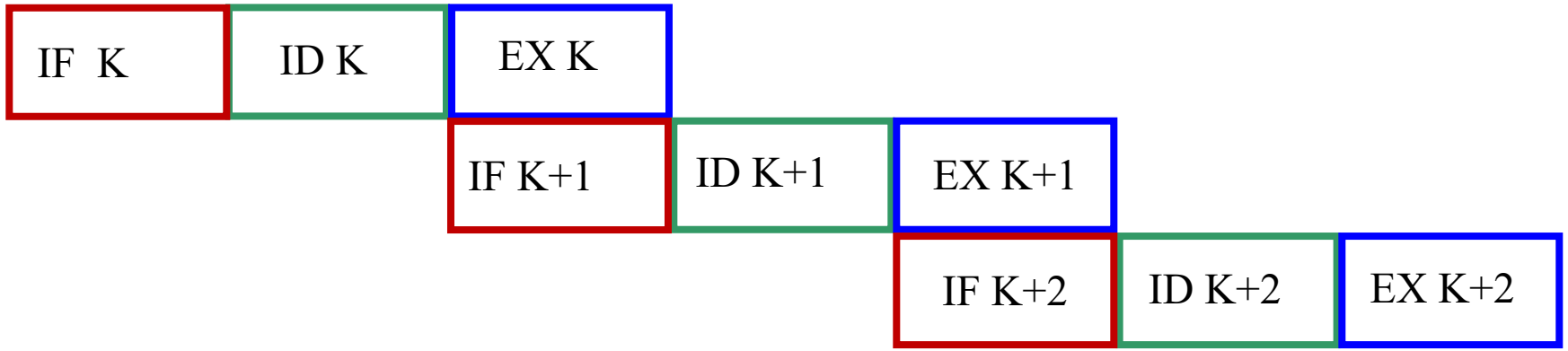
Overlapping execution



- Sequential execution
  - ✓ Simpler control, saving equipment
- Overlapping execution
  - ✓ Higher-usage of functional unit

Overlapping execution

- Single overlapping execution
- Execute the k^{th} instruction at the same time as fetch the $k+1^{th}$ instruction



- Execute time for n instructions by single overlapping execution

$$T = (2n + 1)\Delta t$$

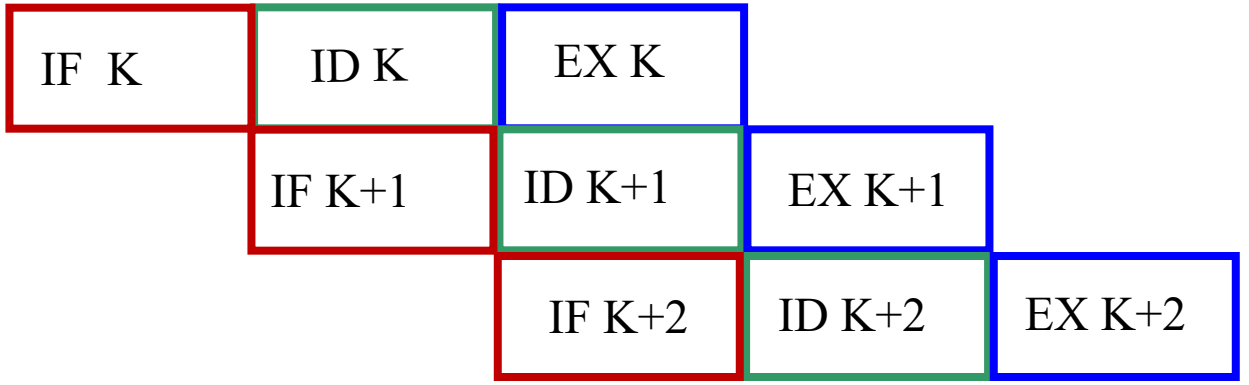
Overlapping execution

- Advantages
  - Execution time was reduced by nearly 1/3
  - The utilization rate of functional unit is improved obviously

- Disadvantages
  - Some additional hardware was needed, and the control process became complicated

Overlapping execution

- Twice overlapping execution
  - Decode the *k* instruction at the same time as fetch the *k+1* instruction
  - Execute the *k* instruction at the same time as decode the *k+1* instruction



- Execute time for n instructions by twice overlapping execution

$$T = (n + 2)\Delta t$$

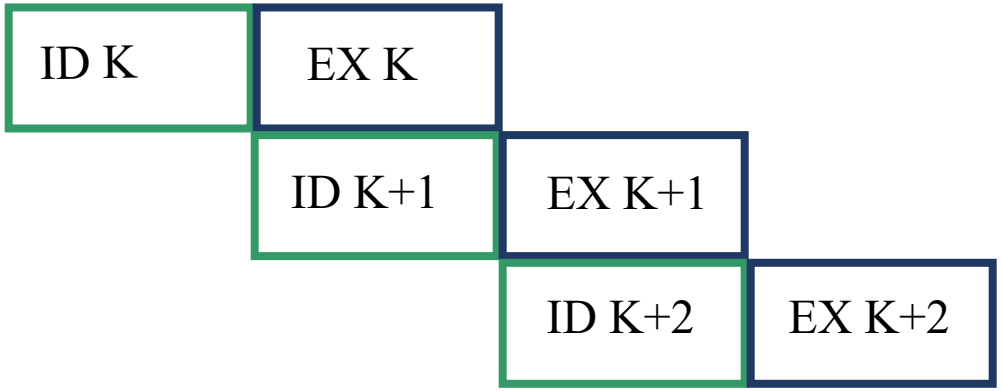
Twice overlapping execution

- Advantages
  - Execution time was reduced by nearly 2/3
  - The utilization rate of functional unit is improved obviously

- Disadvantages
  - Much more hardware was needed
  - Separate fetch, decode, and execution components are required

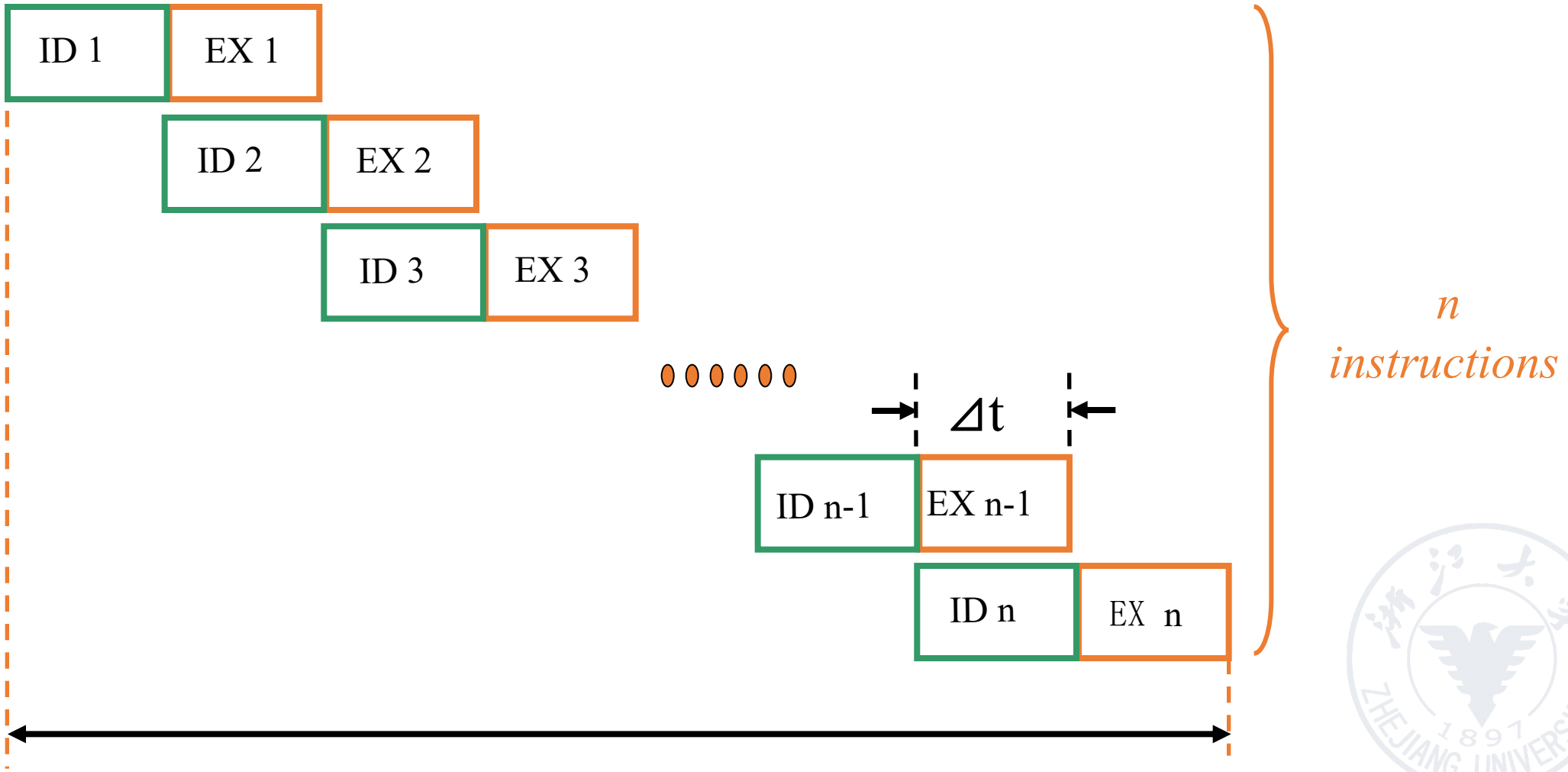
Single overlapping execution

- If the time of fetching instruction phase is very short, fetch operation can be incorporated into the decode operation. The twice overlapping becomes single overlapping

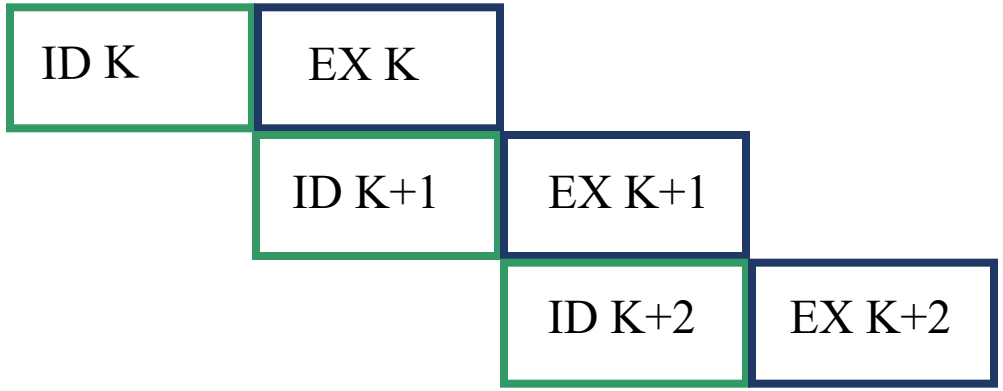


Single overlapping execution

Single overlapping execution

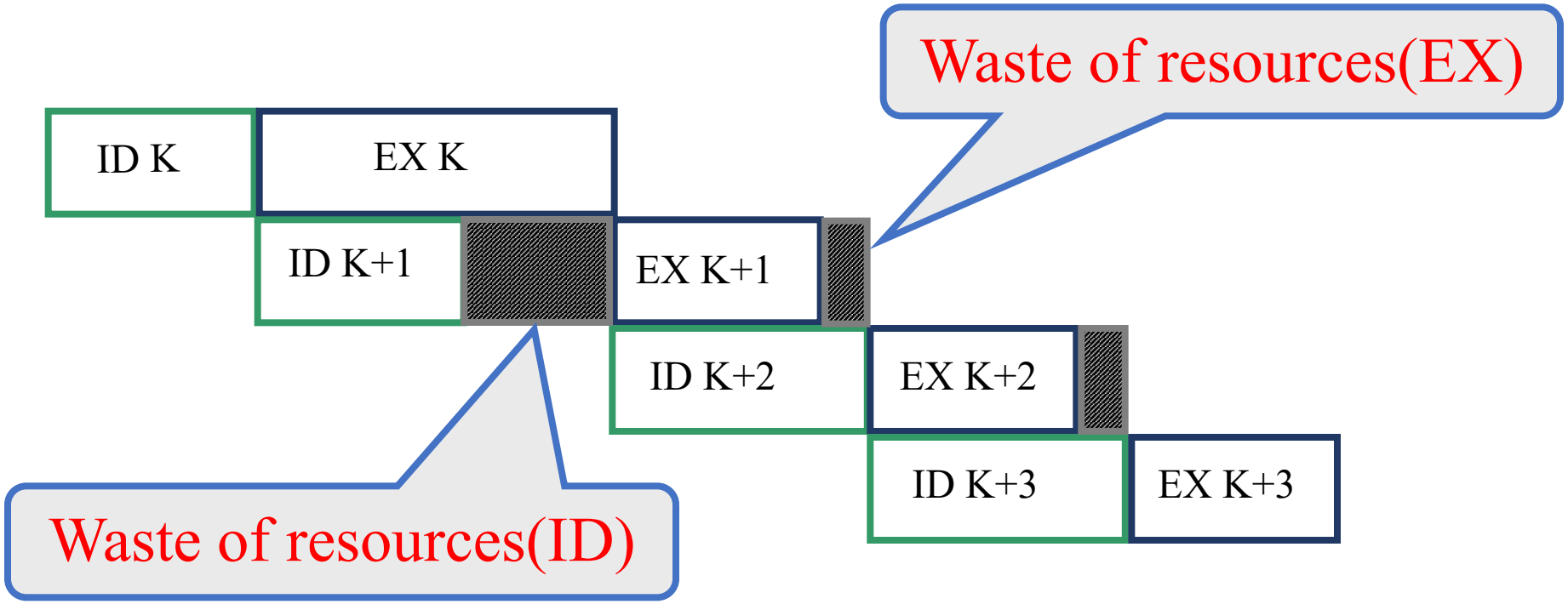


Single overlapping execution



- $\Delta t_{ID} = \Delta t_{EX}$

Single overlapping execution

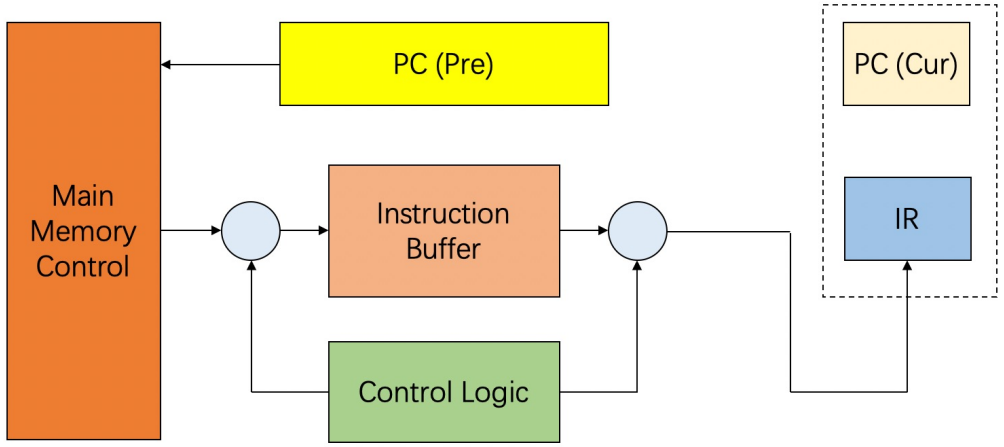


- $\Delta t_{ID} \neq \Delta t_{EX}$

Overlapping execution

- Conflict in access memory
  - Instruction memory & data memory: Harvard structure
  - Instruction cache & data cache (same memory)
  - Adding instruction buffer between memory and instruction decode unit

Instruction Buffer Structure

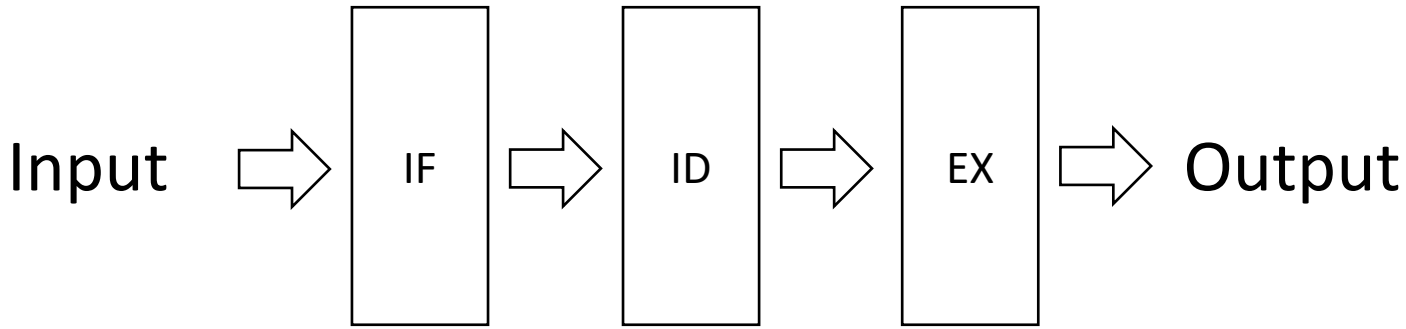


Besides overlapping, what is pipelining ?

- Pipelining: The process of an instruction is divided into *m* (*m* > *2*) sub processes with equal time, and the process of *m* adjacent instructions are staggered and overlapped in the same time

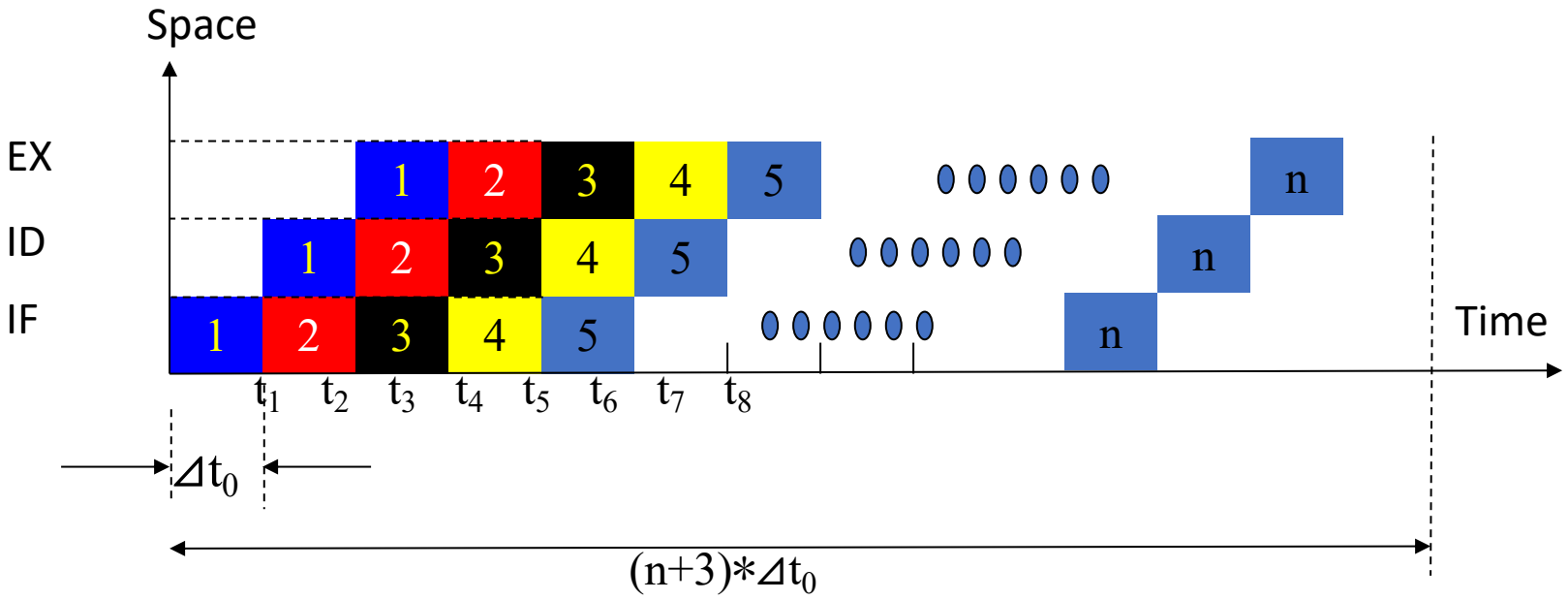
- Pipelining can be regarded as the extension of overlapping execution
- Each subprocess and its functional components in the pipelining are called *stages* or *segments* of the pipelining, which are connected to form a pipelining
- The number of segments in a pipelining is called the *depth* of pipelining

What is pipelining ?



Pipelining

What is pipelining ?



How much faster?

Characteristics of pipelining

- The pipelining divides a process into several sub processes, each of which is implemented by a special functional unit
- The time of each stage in the pipelining should be equal as much as possible, otherwise the pipelining will be blocked and cut off. A longest stage will become the *bottleneck* of the pipelining
- Every functional part of the pipelining must have a buffer register (latch), which is called *pipelining register*

- Function: transfer data between two adjacent stages to ensure the data to be used later, and separate the processing work of each stage from each other

Learning from multi-cycle design

Characteristics of pipelining

- Pipelining technology is suitable for a large number of *repetitive sequential* processes. Only when tasks are *continuously provided* at the input, the efficiency of pipelining can be brought into full play.

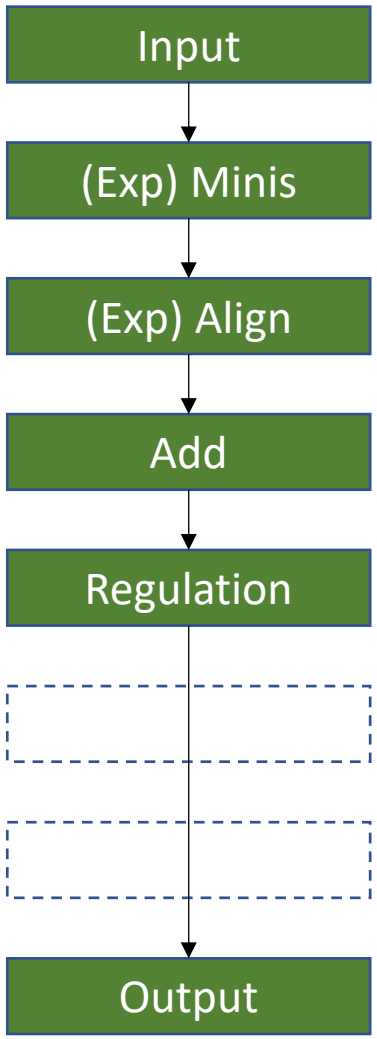
- The pipelining needs the pass time and the empty time
  - *Pass time*: the time for the first task from beginning (entering the pipelining) to ending.
  - *Empty time*: the time for the last task from entering the pipelining to have the result.

Classes of pipelining

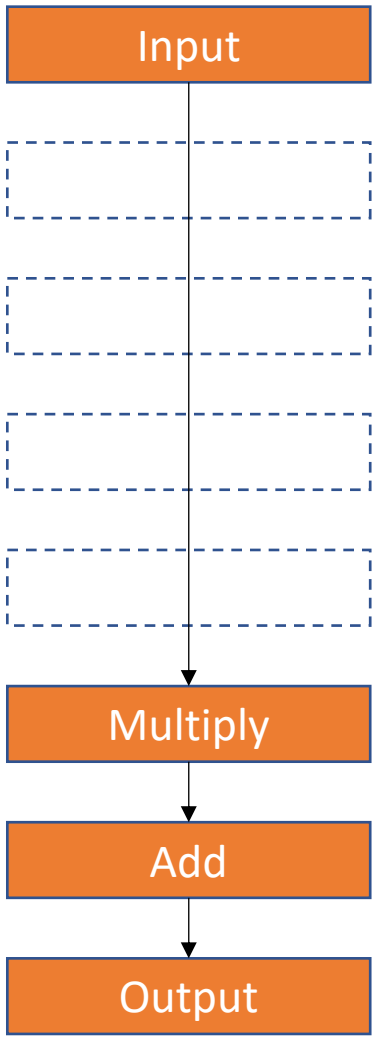
- Single function pipelining: only one fixed function pipelining.
- Multi function pipelining: each stage of the pipelining can be connected differently for several different functions.

§2.3 Classes of pipelining

Segmentation



Floating Point Arithmetic

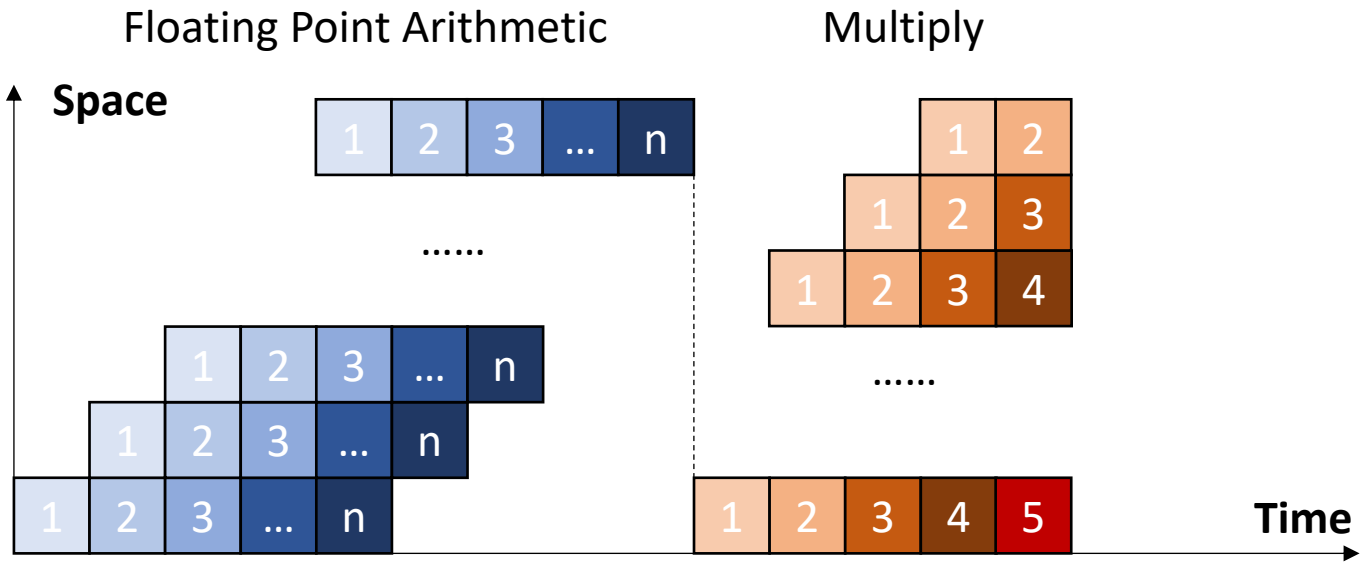


Multiply

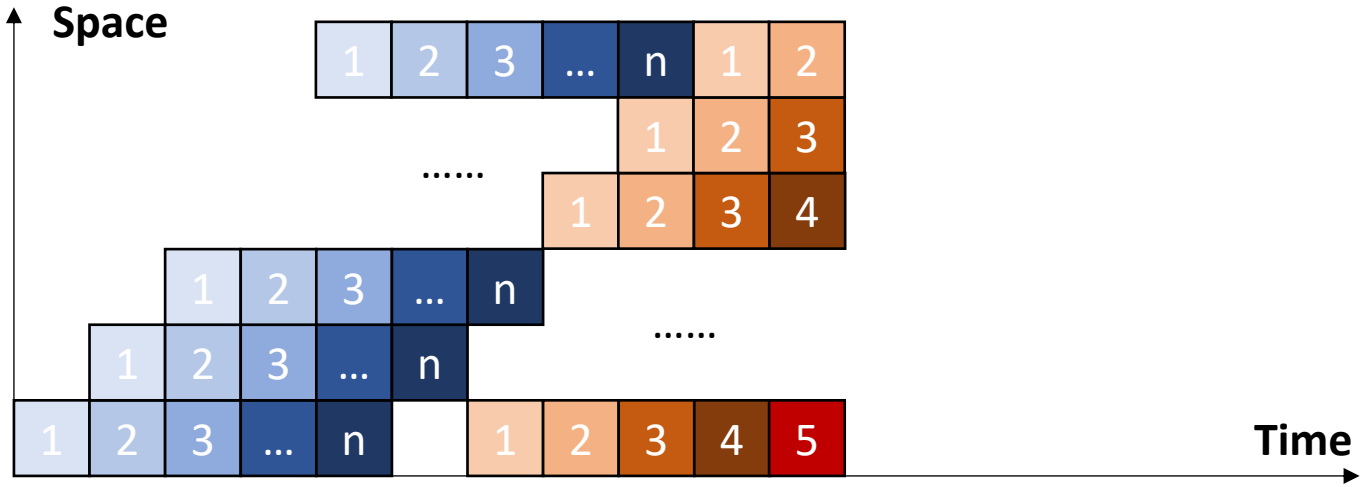
Classes of pipelining

- Static pipelining: In the same time, each segment of the multi-functional pipelining can only work according to the connection mode of the same function
  - For static pipelining, only if the input is a series of the same operation tasks, the efficiency of pipelining can be brought into full play
- Dynamic pipelining: In the same time, each segment of the multi-functional pipelining can be connected in different ways and perform multiple functions at the same time
  - It is flexible but with complex control
  - It can improve the availability of functional units

Static Pipelining



Dynamic Pipelining



Component level pipelining (in component - operation pipelining) : The **arithmetic and logic operation components** of the processor are divided into segments, so that various types of operation can be carried out by pipelining

Processor level pipelining (inter component - instruction pipelining): The **interpretation and execution of instructions** are implemented through pipelining. The execution process of an instruction is divided into several sub processes, each of which is executed in an independent functional unit

Inter processor pipelining (inter processor - macro pipelining): It is a **serial connection of two or more processors** to process the same data stream, and each processor completes a part of the whole task

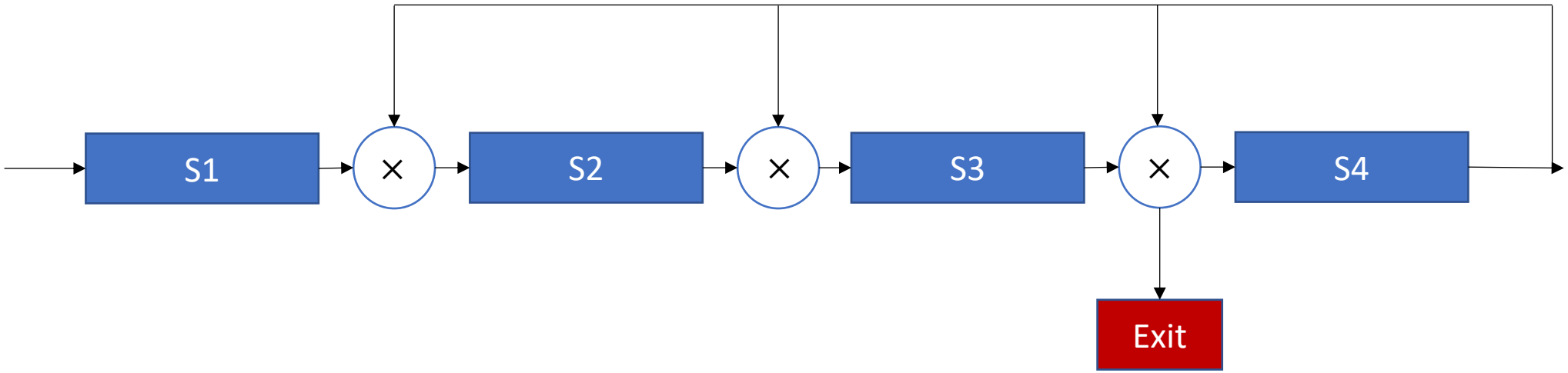
Linear pipelining: Each stage of the pipelining is connected serially without feedback loop. When data passes through each segment in the pipelining, each segment can only flow once at most

Nonlinear pipelining: In addition to the serial connection, there is also a feedback loop in the pipelining

- Scheduling problem of nonlinear pipelining

  Determine when to introduce a new task to the pipelining, so that the task will not conflict with the task previously entering the pipelining

Nonlinear pipelining



Task: → S1 → S2 → S3 → S4 → S2 → S3 → S4 → S3 →

Ordered pipelining: In the pipelining, the outflow order of tasks is exactly the same as the inflow order. Each task flows by sequence in each segment of the pipelining

Disordered pipelining: In the pipelining, the outflow order of tasks is not the same as the inflow order. The later tasks are allowed completed first

Scalar processor: The processor does **NOT** have vector data representation and vector instructions, and only deal with scalar data through pipelining

Vector pipelining processor: The processor has vector data representation and vector instructions. It is the combination of vector data representation and pipelining technology

RISC-V Pipelining

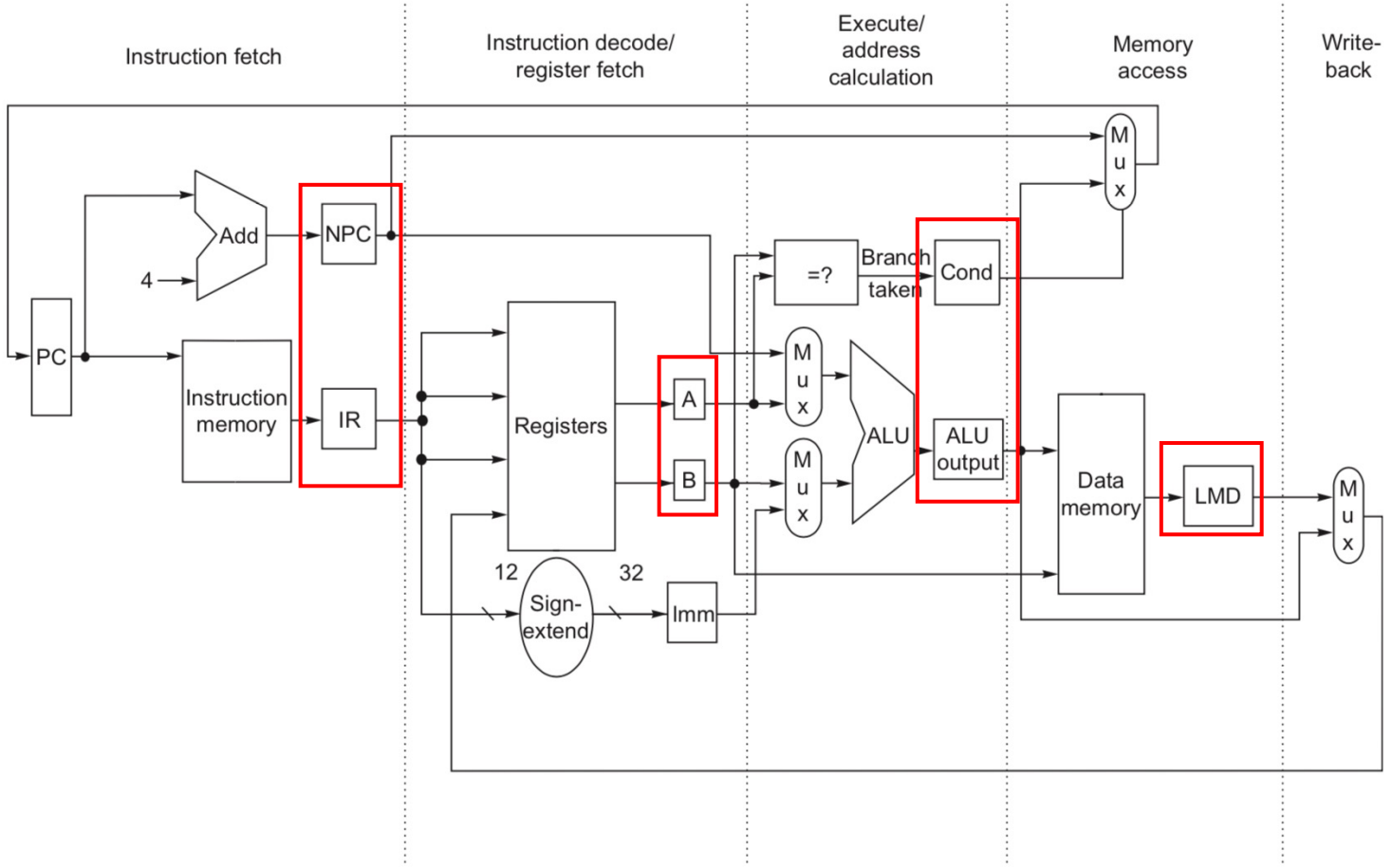
- Five stages, one step per stage

1. IF: Instruction fetch from memory
2. ID: Instruction decode & register read
3. EX: Execute operation or calculate address
4. MEM: Access memory operand
5. WB: Write result back to register

Pipelining and ISA Design

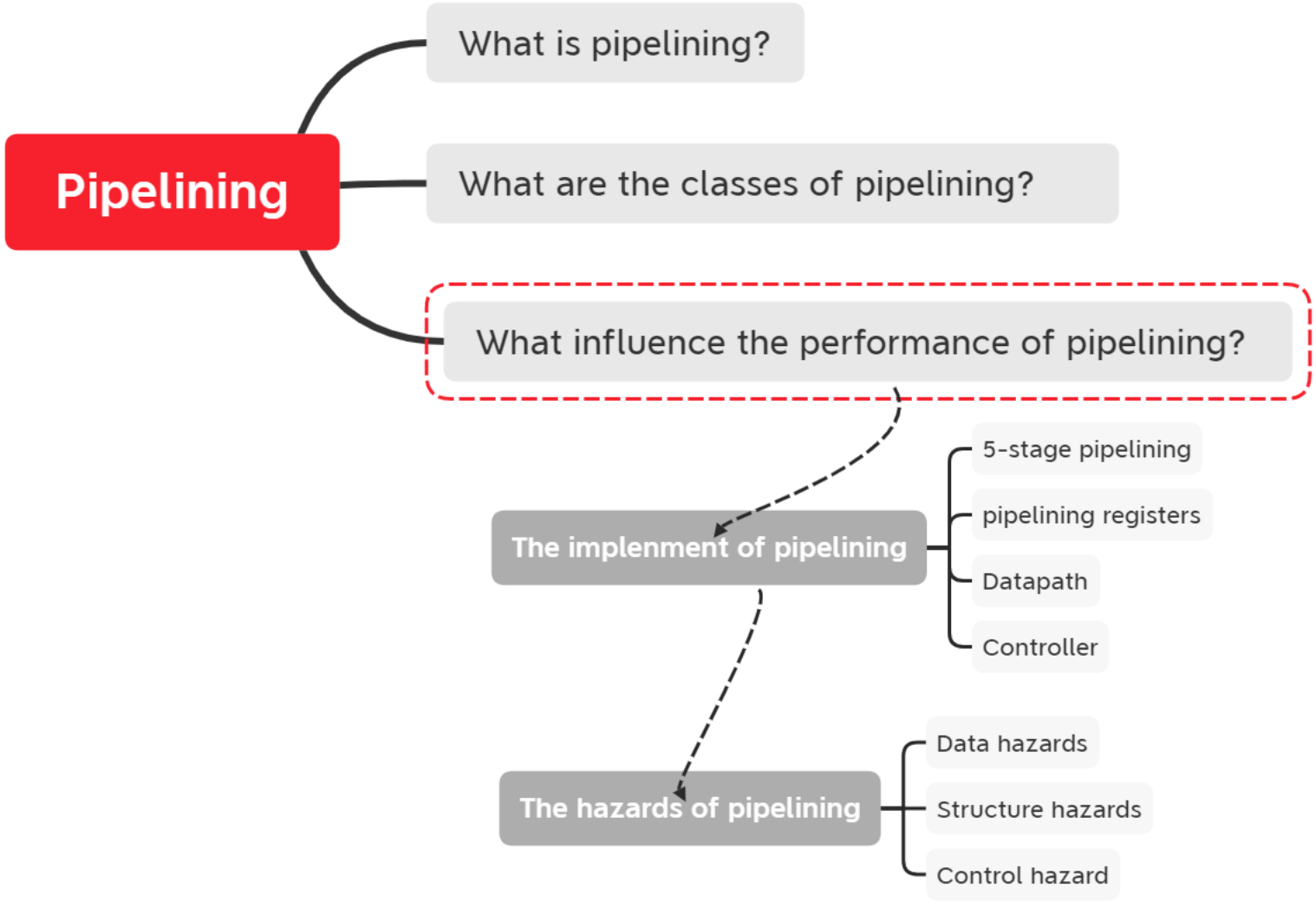
- RISC-V ISA designed for pipelining
  - All instructions are 32-bits
    - Easier to fetch and decode in one cycle
    - c.f. x86: 1- to 17-byte instructions
  - Few and regular instruction formats
    - Can decode and read registers in one step
  - Load/store addressing
    - Can calculate address in 3rd stage, access memory in 4th stage
  - Alignment of memory operands
    - Memory access takes only one cycle

An Implementation of Pipelining



Main Difference: Adding new registers to separate stages physically!

Stage		Any instruction		
IF		IF/ID.IR←Mem[PC] IF/ID.NPC,PC←(if ((EX/MEM.opcode == branch) & EX/MEM.cond){EX/MEM.ALUOutput} else {PC+4});		
ID		ID/EX.A←Regs[IF/ID.IR[rs1]]; ID/EX.B←Regs[IF/ID.IR[rs2]]; ID/EX.NPC←IF/ID.NPC; ID/EX.IR←IF/ID.IR; ID/EX.Imm←sign-extend(IF/ID.IR[immediate field]);		
		ALU instruction	**Load instruction**	**Branch instruction**
EX	R	EX/MEM.IR←ID/EX.IR; EX/MEM.ALUOutput← ID/EX.A func ID/EX.B; or	EX/MEM.IR to ID/EX.IR EX/MEM.ALUOutput← ID/EX.A+ID/EX.Imm;	EX/MEM.ALUOutput← ID/EX.NPC + (ID/EX.Imm<< 2);
	I	EX/MEM.ALUOutput← ID/EX.A *op* ID/EX.Imm;		
			EX/MEM.B←ID/EX.B;	EX/MEM.cond← (ID/EX.A == ID/EX.B);
MEM		MEM/WB.IR←EX/MEM.IR; MEM/WB.ALUOutput← EX/MEM.ALUOutput;	MEM/WB.IR←EX/MEM.IR; MEM/WB.LMD← Mem[EX/MEM.ALUOutput];	
	I		or	
	S		Mem[EX/MEM.ALUOutput]← EX/MEM.B;	
WB		Regs[MEM/WB.IR[rd]]← MEM/WB.ALUOutput;	For load only: Regs[MEM/WB.IR[rd]]← MEM/WB.LMD;	



Single-Cycle vs. Pipelined Performance

- Example: consider a system with seven instructions
  - load word (lw)
  - store word (sw)
  - add (add)
  - subtract (sub)
  - AND (and)
  - OR (or)
  - branch if equal(beq)

Total Time for Each Instruction

Instruction Class	Instruction Fetch	Register Read	ALU Operation	Data Access	Register Write	Total Time
lw	200ps	100ps	200ps	200ps	100ps	800ps
sw	200ps	100ps	200ps	200ps		700ps
add, sub, and, or	200ps	100ps	200ps		100ps	600ps
beq	200ps	100	200ps			500ps

Question: Consider the Clock Period for the Single-Cycle and the Pipeline?

Total Time for Each Instruction

Instruction Class	Instruction Fetch	Register Read	ALU Operation	Data Access	Register Write	Total Time
lw	200ps	100ps	200ps	200ps	100ps	800ps
sw	200ps	100ps	200ps	200ps		700ps
add, sub, and, or	200ps	100ps	200ps		100ps	600ps
beq	200ps	100	200ps			500ps

Clock Period for Single-Cycle = MAX(instructions total times) = 800ps

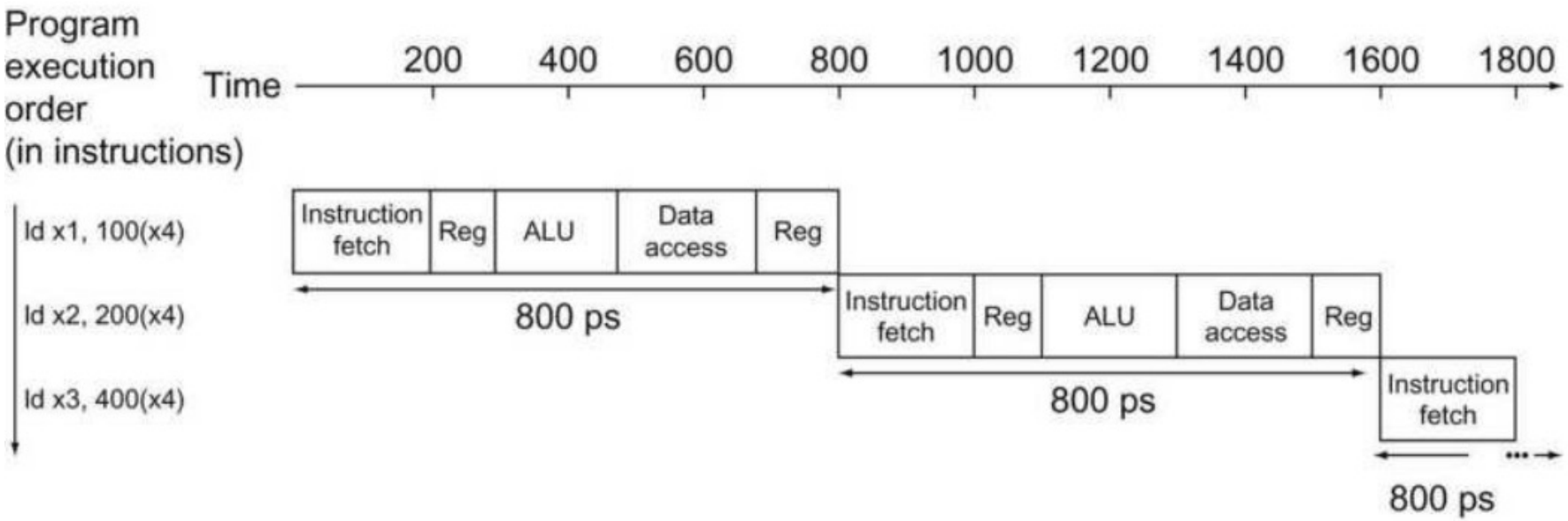
Total Time for Each Instruction

Instruction Class	Instruction Fetch	Register Read	ALU Operation	Data Access	Register Write	Total Time
lw	200ps	100ps	200ps	200ps	100ps	800ps
sw	200ps	100ps	200ps	200ps		700ps
add, sub, and, or	200ps	100ps	200ps		100ps	600ps
beq	200ps	100	200ps			500ps

Clock Period for Single-Cycle = MAX(instructions total times) = 800ps

Clock Period for Pipelined = MAX(operations times) = 200ps

Single-Cycle, Nonpipelined Execution

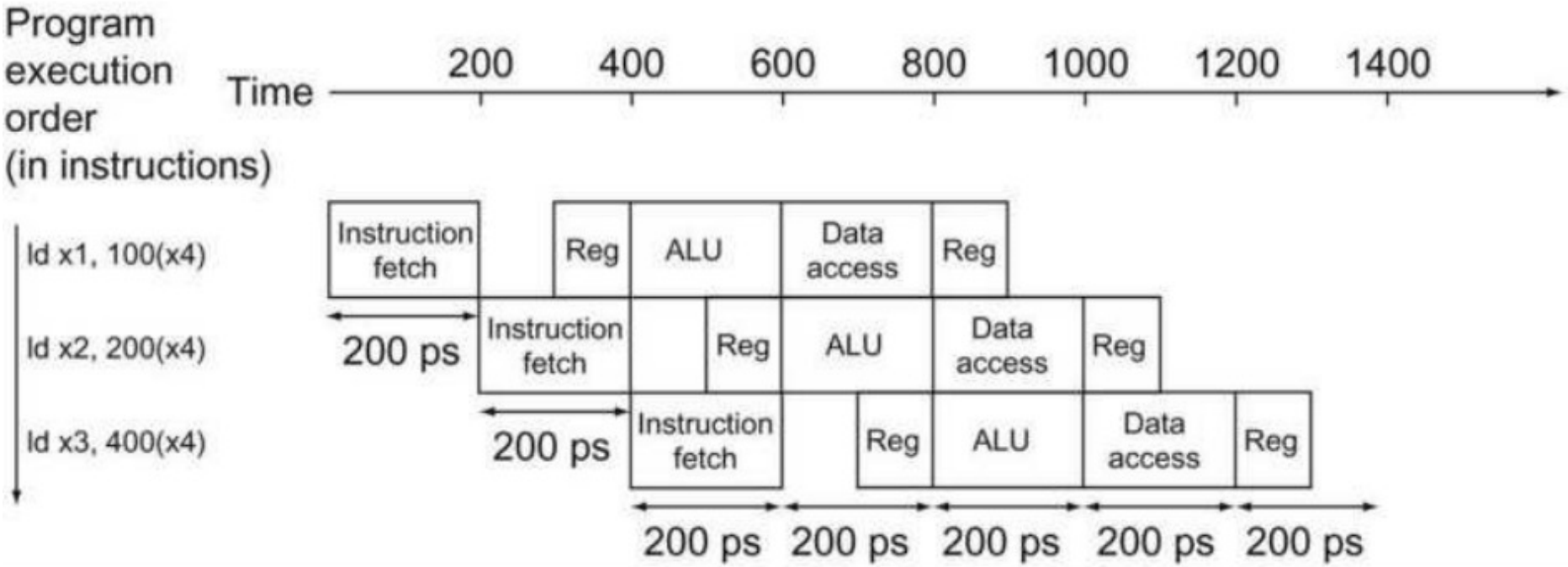


Total execution time for the three instructions = 800ps * 3 = 2400ps

How about Pipeline?

Instruction Class	Instruction Fetch	Register Read	ALU Operation	Data Access	Register Write	Total Time
lw	200ps	100ps	200ps	200ps	100ps	800ps
sw	200ps	100ps	200ps	200ps		700ps
add, sub, and, or	200ps	100ps	200ps		100ps	600ps
beq	200ps	100	200ps			500ps

Pipelined Execution



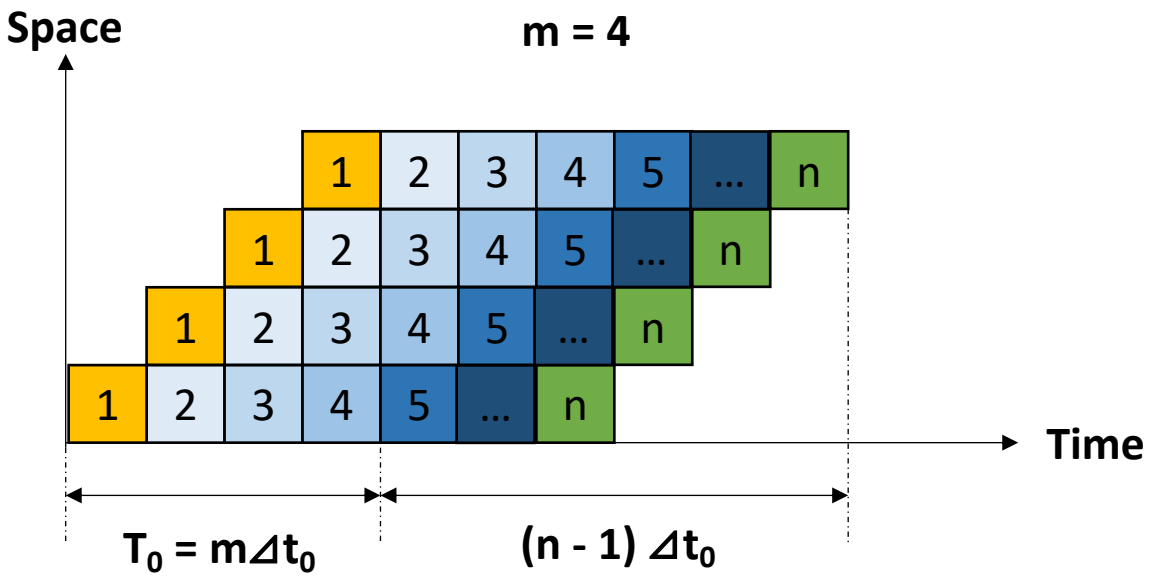
Total execution time for the three instructions = 200ps * 7 = 1400ps

Throughput (TP)

$$TP = \frac{n}{T}$$

$$TP < TP_{max}$$

If $n>>m$,
$TP \approx TP_{max}$



$$T = (m + n - 1) \times \Delta t_0$$

$$TP = n / (m + n - 1)\ \Delta t_0$$

$$TP_{max} = 1 / \Delta t_0$$

Throughput (TP)

$$TP = \frac{n}{n + m - 1} TP_{max}$$

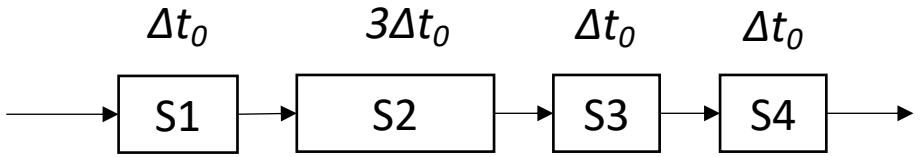
- The actual throughput of the pipeline is **less than** the maximum throughput, which is not only related to the time of each segment, but also related to *m* and *n*

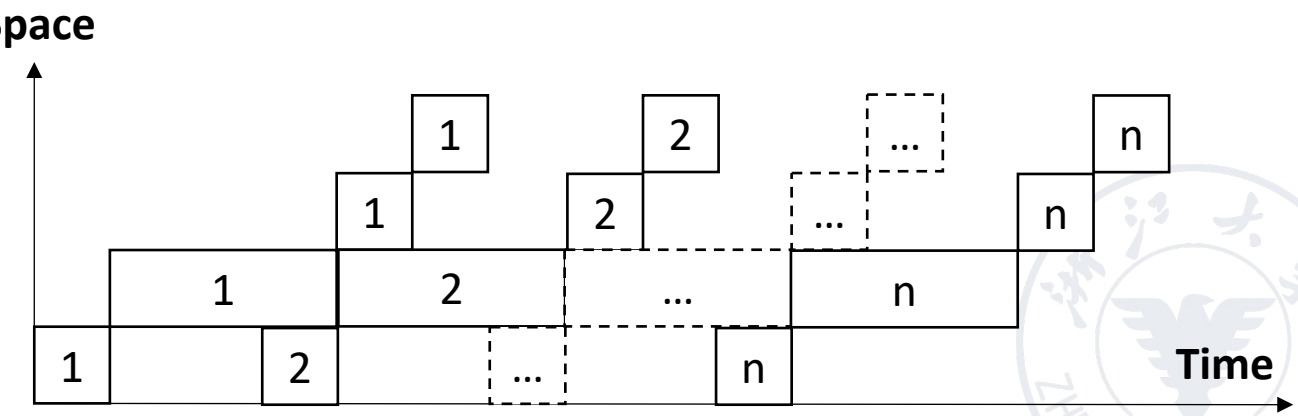
- If $n >> m$, $TP \approx TP_{max}$

Throughput under Practical Case

- Suppose the time of segments are different in pipelining,
  - $m = 4$
  - Time of *S1, S3, S4:* Δt
  - Time of *S2:* $3\Delta t$ (**Bottleneck**)

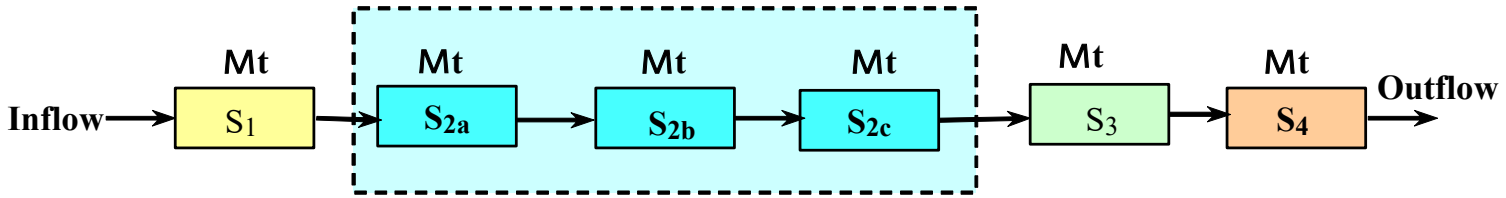
The longest segment in the pipelining is called the bottleneck segment





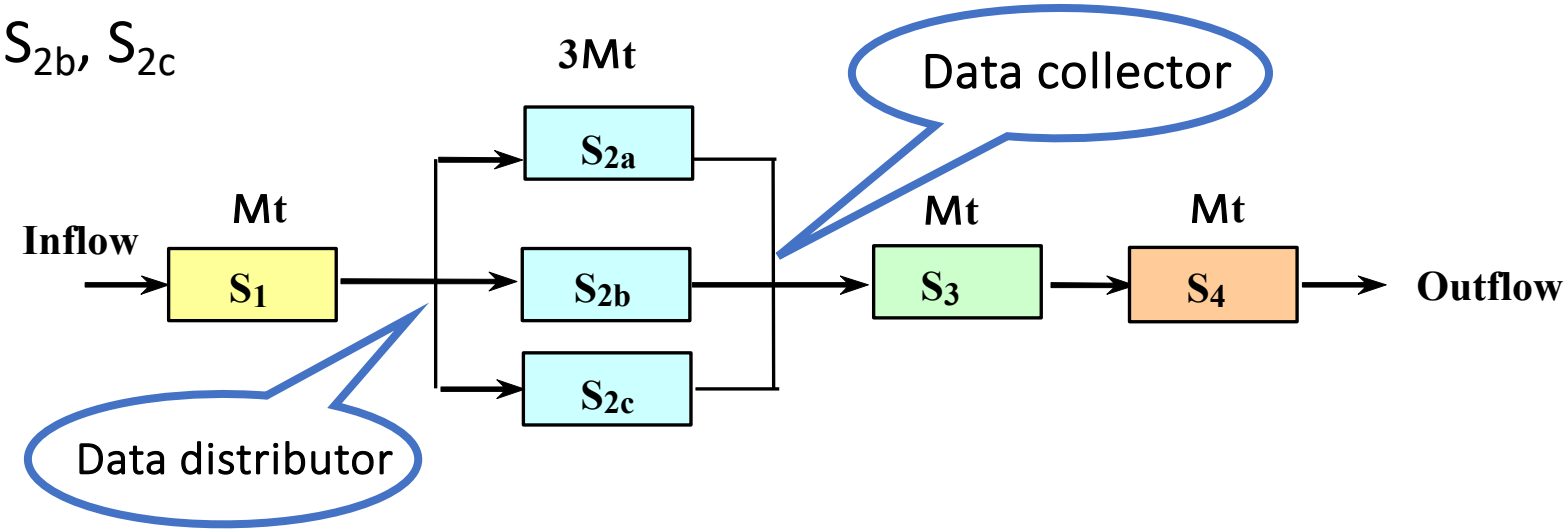
Common methods to solve pipeline bottleneck

- Subdivision
  - Divide S_2 into 3 subsegments： S_{2a}, S_{2b}, S_{2c}

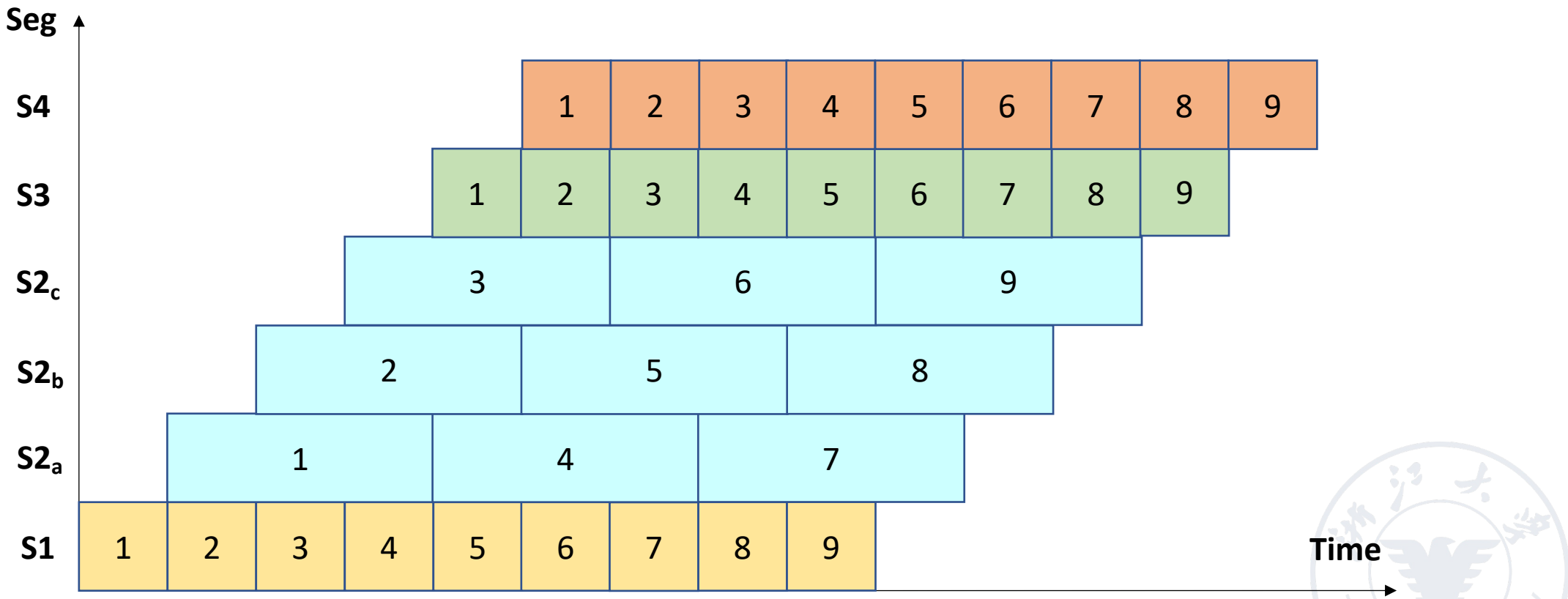


$$TP_{max} = \frac{1}{\Delta t}$$

- Repetition
  - S_2: S_{2a}, S_{2b}, S_{2c}



Time space diagram with repetition bottleneck segment



Speedup (Sp)

In the example (execute 3 instructions):

- Non-pipelined execution time **vs.** Pipelined execution time

  2,400ps **vs.** 1,400ps **1.7:1**

What would happen if we increase the number of instructions?

Extend the pervious example to 1,000,003 instructions

- Non-pipelined execution time **vs.** Pipelined execution time

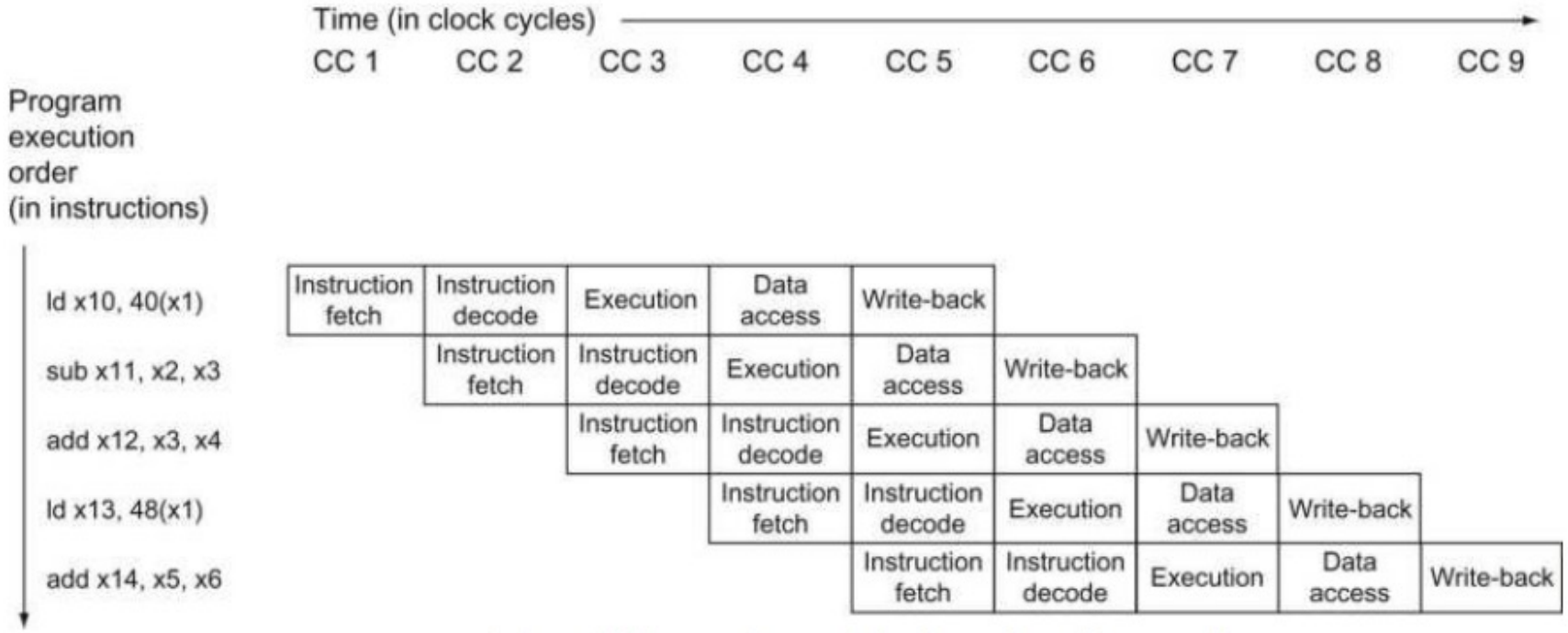
  1,000,000 ×800ps +2,400ps **vs**. 1,000,000 ×200ps +1,400ps

  800,002,400ps **vs.** 200,001,400ps

  4:1

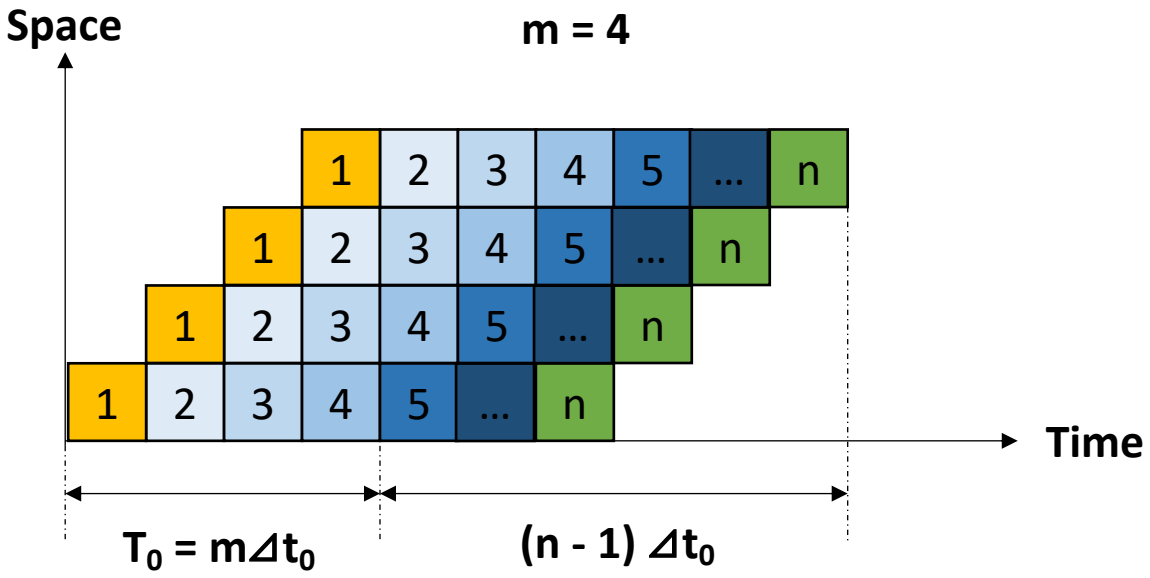
Speedup (Sp)

$$Execution\ Time_{\boldsymbol{pipelined}} = \frac{Execution\ Time_{\boldsymbol{non-pipelined}}}{Number\ of\ Pipestages}$$



The five-stage pipeline is nearly five times faster

Speedup (Sp)

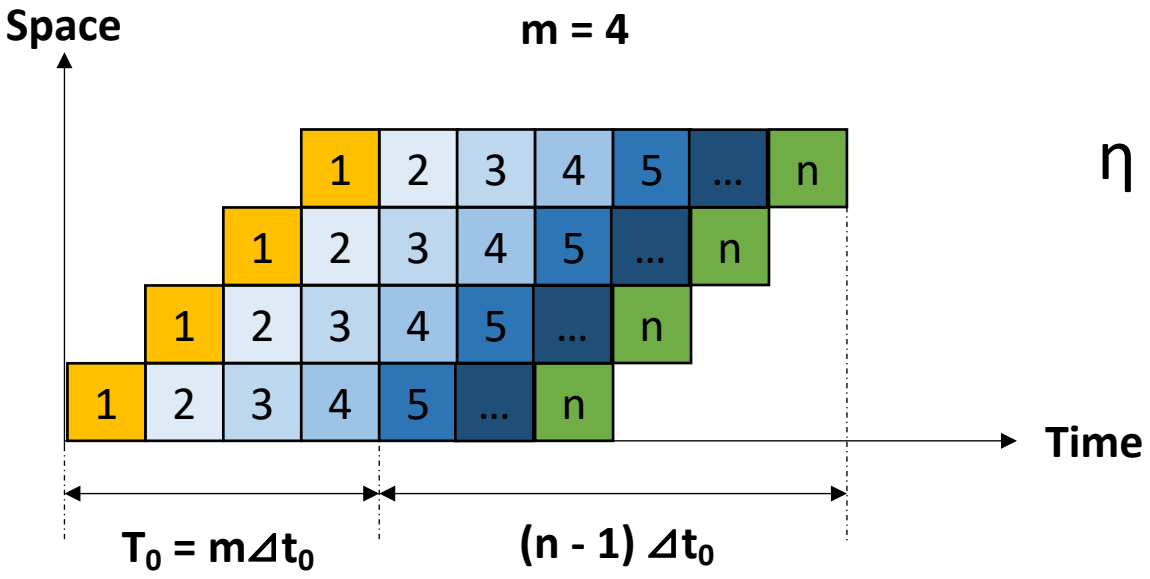


$$T = (m + n - 1) \times \Delta t_0$$

$$Sp = (n \times m \times \Delta t_0) / (m + n - 1) \Delta t_0$$
$$= (n \times m) / (m + n - 1)$$

If n>>m,
Sp ≈ m

Efficiency (η)



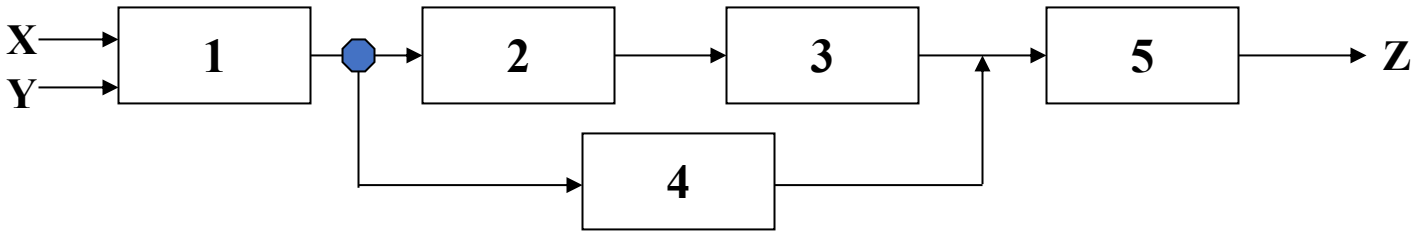
$$T = (m + n - 1) \times \Delta t_0$$

$$\eta = (n \times m \times \Delta t_0) / (m + n - 1)\,\Delta t_0 \times m$$
$$= n / (m + n - 1)$$

If n>>m,
$\eta \approx 1$

Pipeline Performance

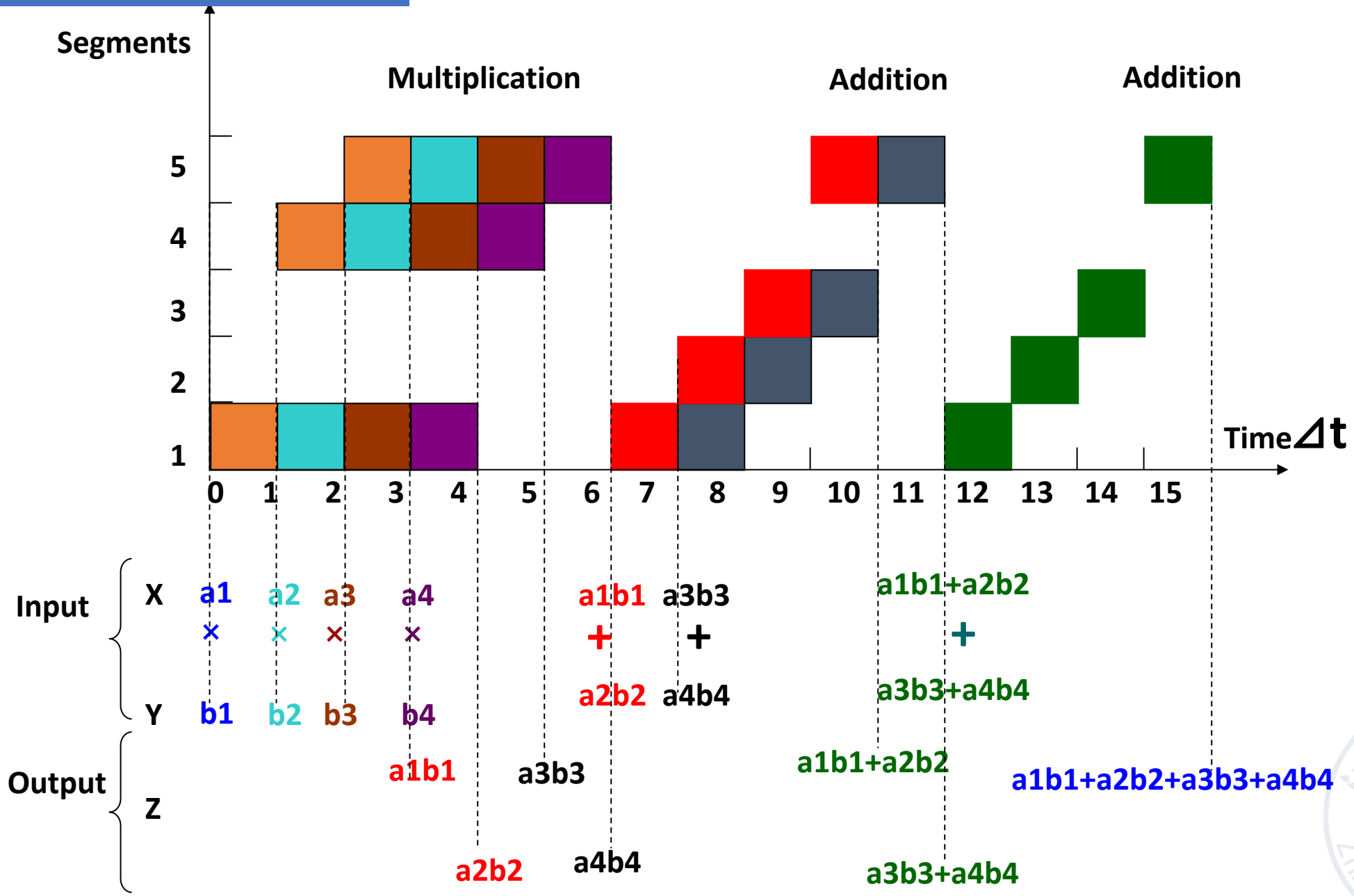
- Vector A(a1, a2,a3,a4)
- Vector B(b1,b2,b3,b4)
- Compute vector dot product (A·B) in the static dual-function pipelining

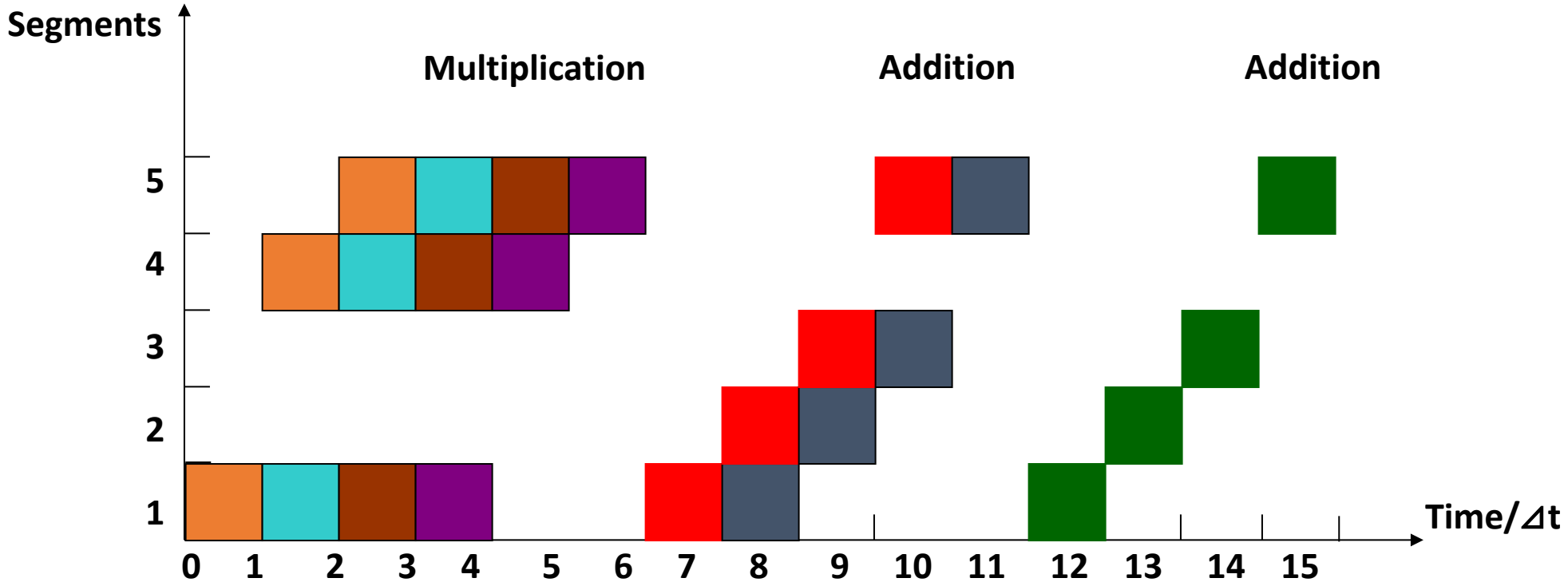


- 1→2→3→5 Addition pipelining
- 1→4→5 Multiplication pipelining
- The time of each segment in the pipelining is Δt

TP? Sp? η?

§2.5 Performance evaluation of pipelining





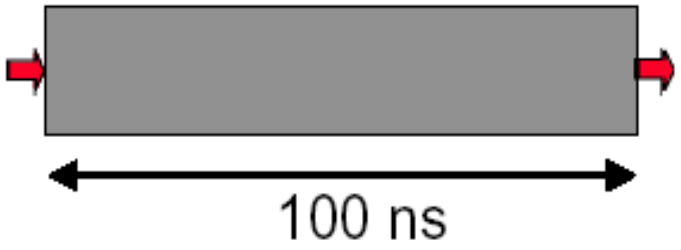
$$TP=7/(15\Delta t)=0.47/\Delta t$$

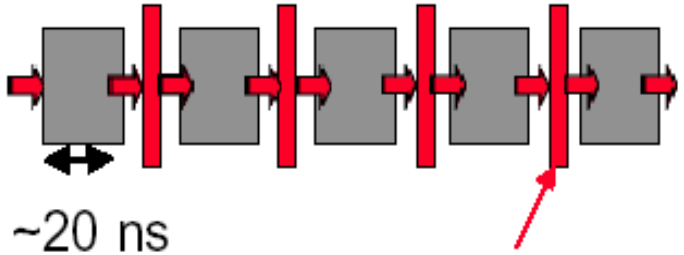
$$Sp=(4\times 3\Delta t+3\times 4\Delta t)/(15\Delta t)=1.6$$

$$\eta=(3\times 4\Delta t+4\times 3\Delta t)/(5\times 15\Delta t)=32\%$$

Discussion

- Why pipelining: overlapped





- Can "launch" a new computation every 100ns in this structure
- Can finish 10^7 computations per second

- Can launch a new computation every 20ns in pipelined structure
- Can finish 5×10^7 computations per second

Discussion

- Why pipelining?
  - The key implementation technique used to make fast CPU: decrease CPU time
  - Improving of throughput (rather than individual execution time)
  - Improving of efficiency for resources (functional unit)

Discussion

- Ideal Performance for Pipelining

- If the stages are perfectly balanced, the time per instruction on the pipelined processor equal to:

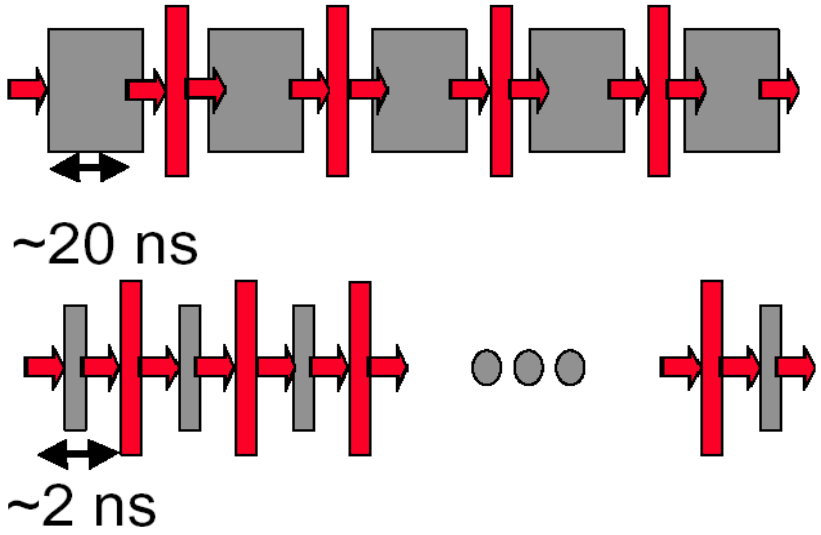
$$\frac{\textit{Time per instruction on unpipelined machine}}{\textit{Number of pipelined stages}}$$

- So, ideal speedup equal to

Number of pipeline stages

Discussion

- Why not just make a 50-stage pipelining ?



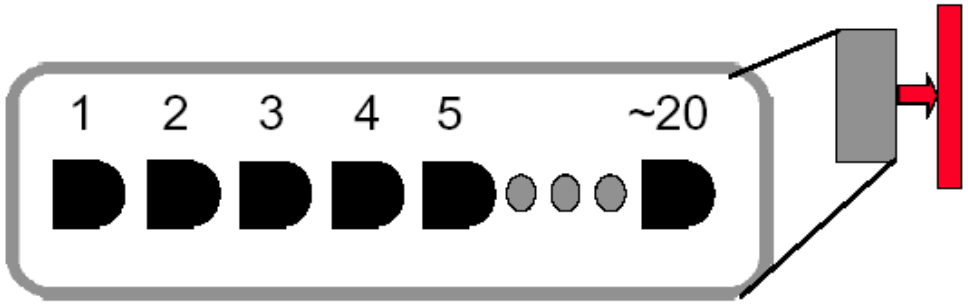
5 stages OK

50 stages NO. Sorry!

- Too many stages:
  - Lots of complications
  - Should take care of possible dependencies among in-flight instructions
  - Control logic is huge

Discussion

- Why not just make a 50-stage pipelining ?
- Those latches are NOT free, they take up area, and there is a real delay to go THRU the latch itself
  - Machine cycle > latch latency + clock skew
- In modern, deep pipeline (10-20 stages), this is a real effect
  - Typically see logic “depths” in one pipe stage of 10-20 “gates”



At these speeds, and with this few levels of logic, latch delay is important

Discussion

- What factors affect the **efficiency** of multi-functional pipeline?

1. When the multi-functional pipeline implements a certain function, there are always **some segments** for other functions in the **idle** state
2. In the process of **pipelining establishment**, some segments to be used are also idle
3. When **the segments are not equal**, the clock cycle depends on the time of the **bottleneck** segment
4. When the **functions are switch**, the pipelining needs to be **emptied**
5. The output of last operation is the input of the next operation (**Dependency**)
6. Extra cost: pipelining **register delay** & overhead of **clock skew**