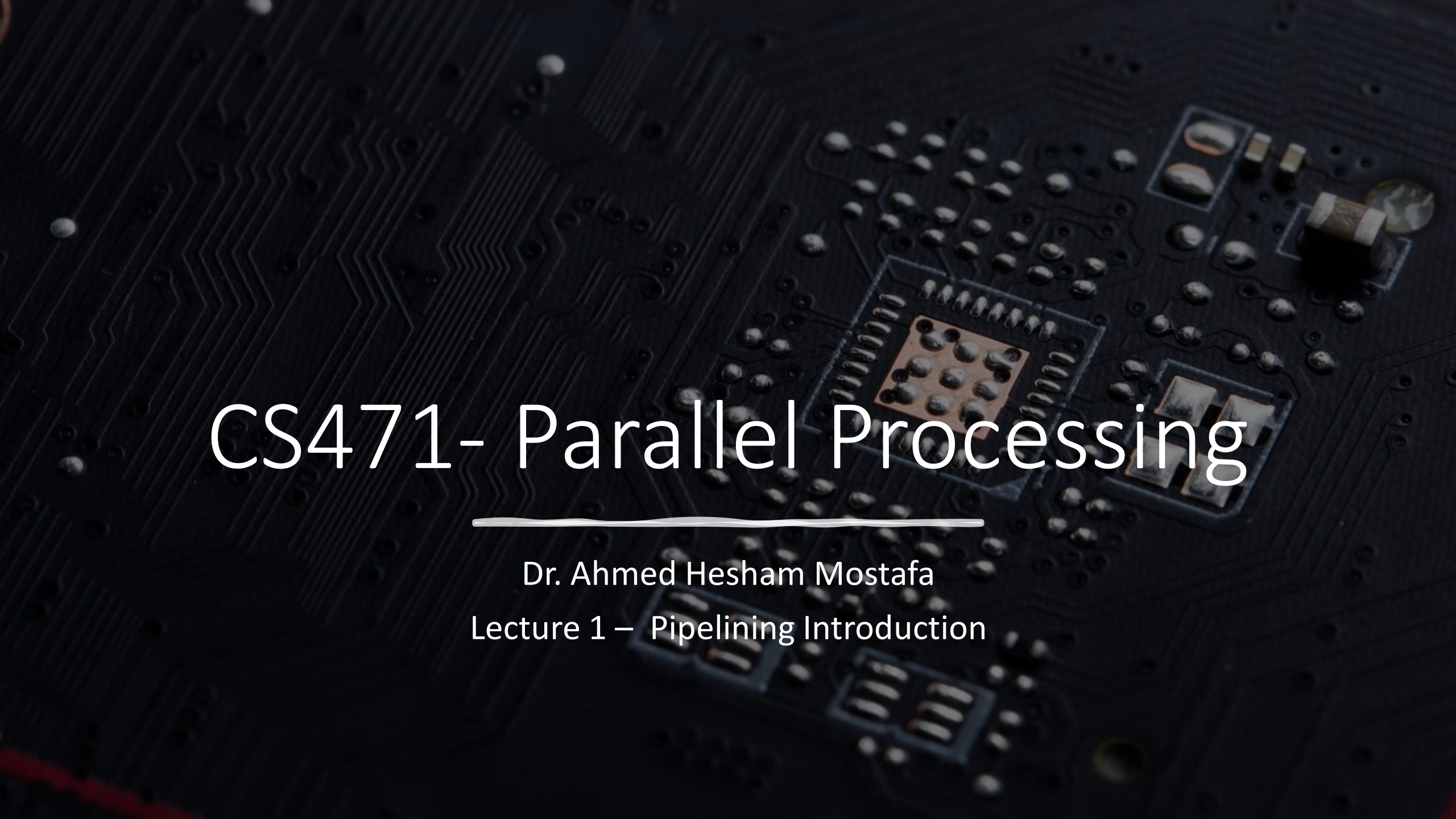


CS471- Parallel Processing

Dr. Ahmed Hesham Mostafa

Lecture 1 – Pipelining Introduction

Parallel Processing

- Execution of Concurrent Events in the computing process to achieve faster Computational Speed
- **The purpose of parallel processing is to speed up the computer processing capability and increase its throughput, that is, the amount of processing that can be done during a given interval of time.**
- **The amount of hardware increases with parallel processing, and with it, the cost of the system increases.**
- **However, technological developments have reduced hardware costs to the point where parallel processing techniques are economically feasible.**

Parallel processing according to levels of complexity

- **At the lower level**
- Serial Shift register VS parallel load registers
- **At the higher level**
- Multiple functional units that perform identical or different operations at the same time.

Parallel processing classification

- There are a variety of ways that parallel processing can be classified.
- It can be considered from the internal organization of the processors, from the interconnection structure between processors, or from the flow of information through the system.
- One classification introduced by M. J. Flynn considers the organization of a computer system by the number of instructions and data items that are manipulated at the same time.

Parallel processing classification

- The normal operation of a computer is to fetch instructions from memory and execute them in the processor.
- The sequence of instructions read from the memory → **instruction stream.**
- The operations performed on the data in the processor → **a data stream.**
- Parallel processing may occur in the instruction stream, in the data stream, or in both.

Parallel processing classification

In 1966 Michael J Flynn proposed a classification system where architecture can be divided into 4 types:

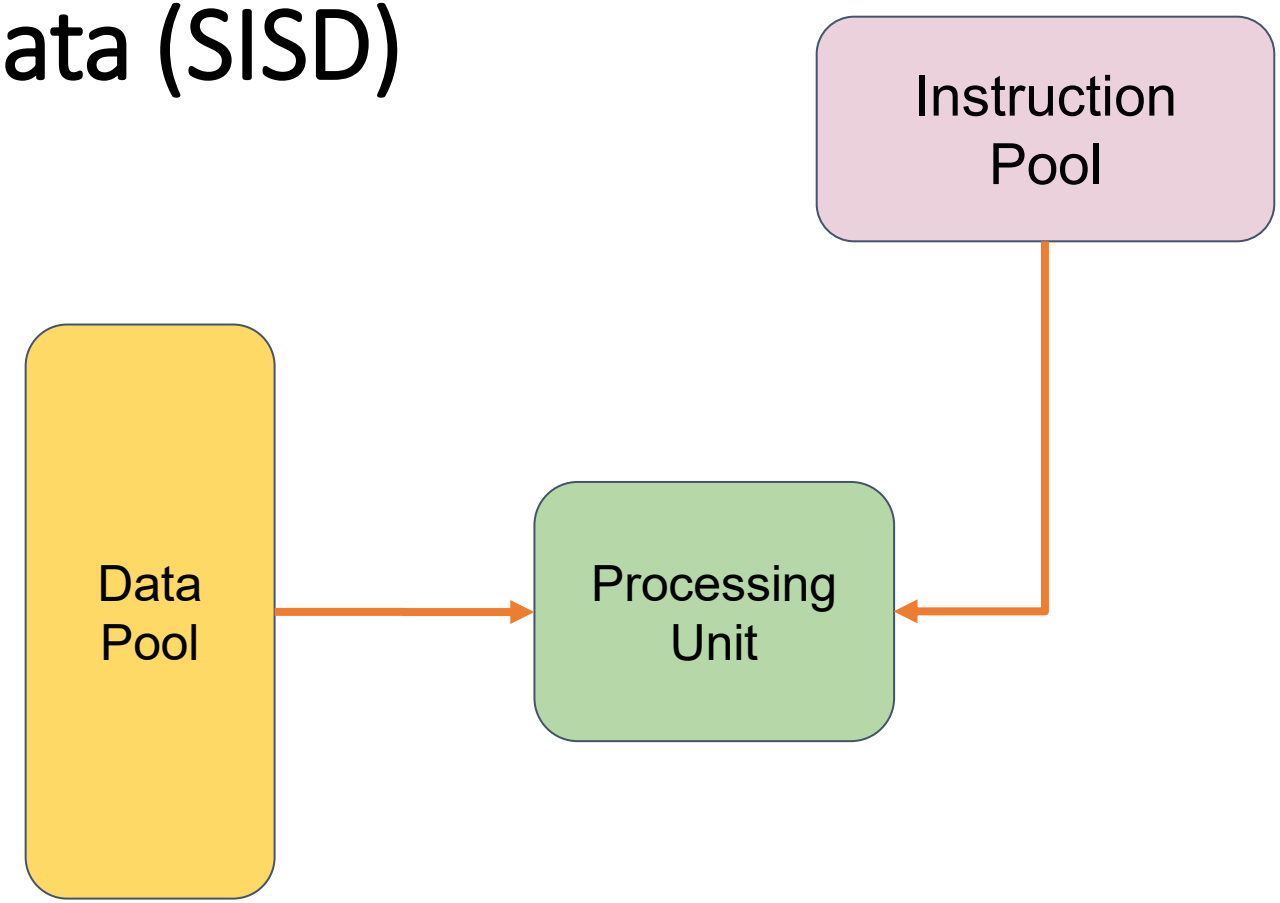
- SISD - Single Instruction Single Data
- SIMD - Single Instruction Multiple Data
- MISD - Multiple Instruction Single Data
- MIMD - Multiple Instruction Multiple Data

Single Instruction Single Data (SISD)

- SISD represents the organization of a single computer containing a control unit, a processor unit, and a memory unit.
- Instructions are executed sequentially, and the system may or may not have internal parallel processing capabilities.
- Parallel processing in this case may be achieved by means of multiple functional units or by pipeline processing.

Single Instruction Single Data (SISD)

- A single processor takes data from a single address in memory and performs a single instruction on the data at a time
- Pipelining can be implemented, but only one instruction will be executed at a time.
- All single-processor systems are SISD.



Single Instruction Single Data (SISD)

Advantages

- Cheap
- Low power consumption

Disadvantages

- Limited speed due to being a single core

Uses

- Microcontrollers
- Older mainframes

Single Instruction Multiple Data (SIMD)

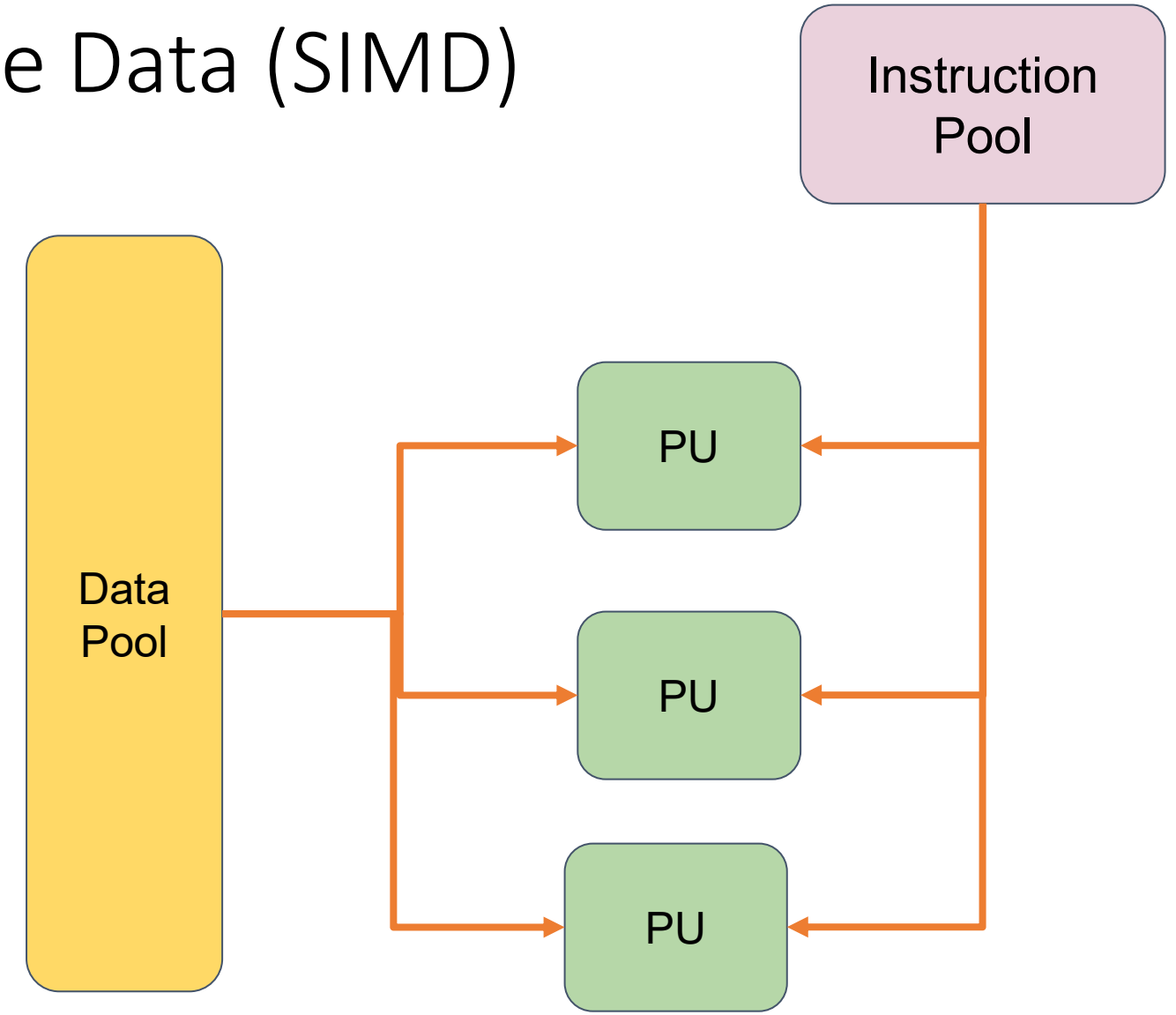
- SIMD represents an organization that includes many processing units under the supervision of a common control unit.
- All processors receive the same instruction from the control unit but operate on different items of data.

Single Instruction Multiple Data (SIMD)

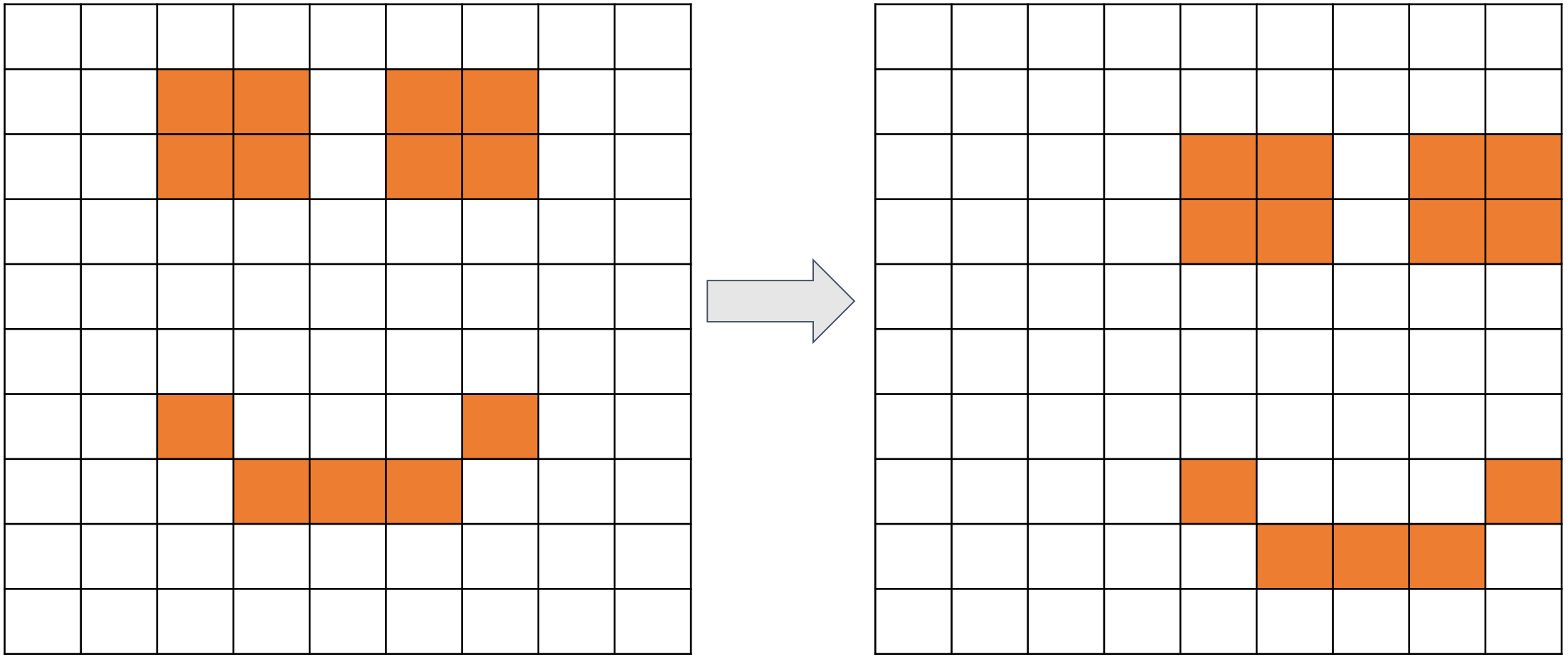
A single instruction is executed on multiple different pieces of data

These instructions can be performed sequentially, taking advantage of pipelining, or in parallel using multiple processors.

Modern GPUs, containing Vector processors and array processors, are commonly SIMD systems.



SIMD Example - moving a game character



X+2 Y-1

Single Instruction Multiple Data (SIMD)

Advantages

- Very efficient where you need to perform the same instruction on large amounts of data.

Disadvantages

- Limited to specific applications

Uses

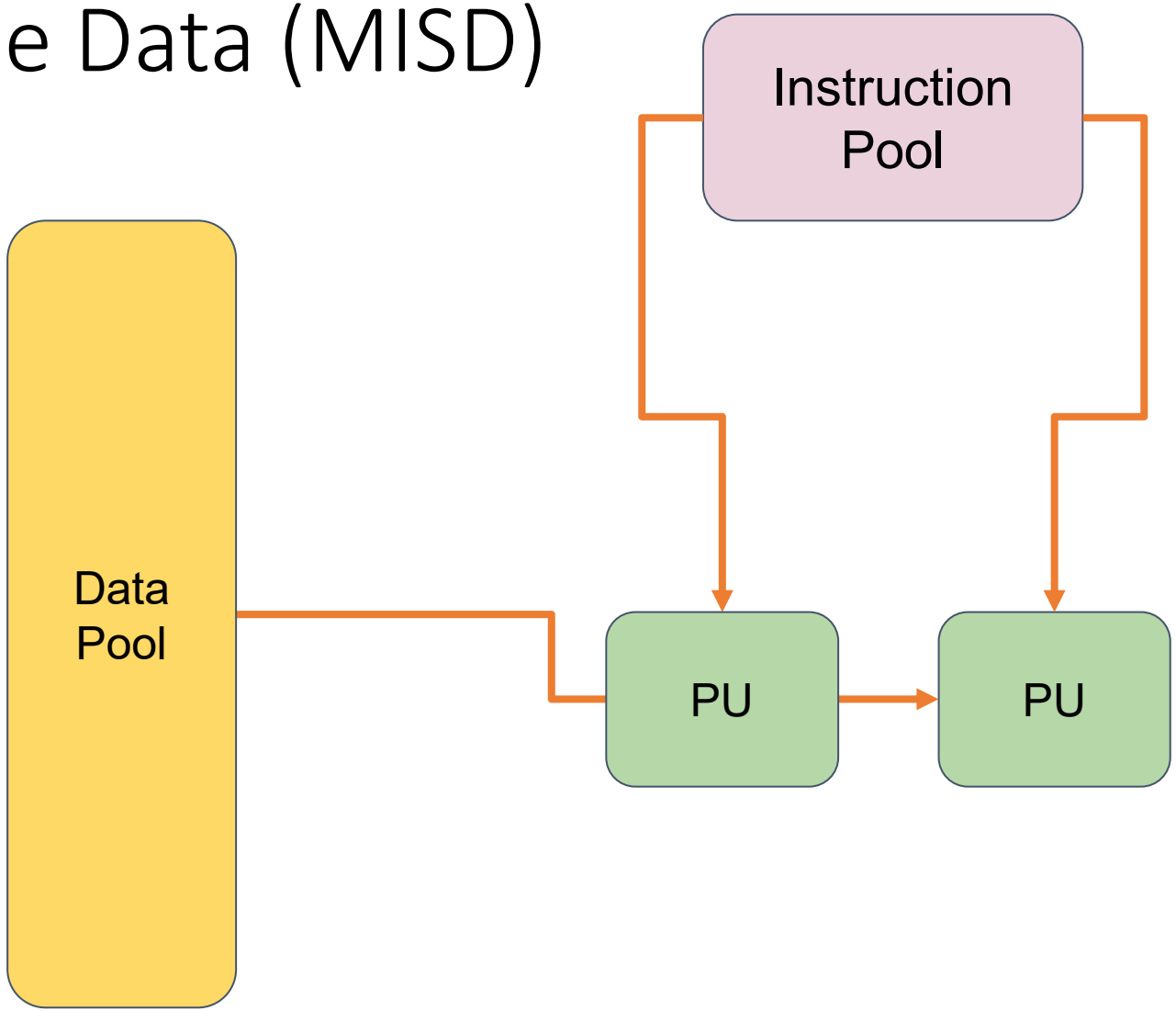
- GPUs
- Scientific processing

Multiple Instruction Single Data (MISD)

- MISD structure is only of theoretical interest since no practical system has been constructed using this organization.

Multiple Instruction Single Data (MISD)

This is where multiple processors work on the same data set, performing the same instructions at the same time.



Multiple Instruction Single Data (MISD)

Advantages

- Useful where real-time fault detection is critical

Disadvantages

- Very limited application so not available commercially

Uses

- Space shuttle flight control systems.

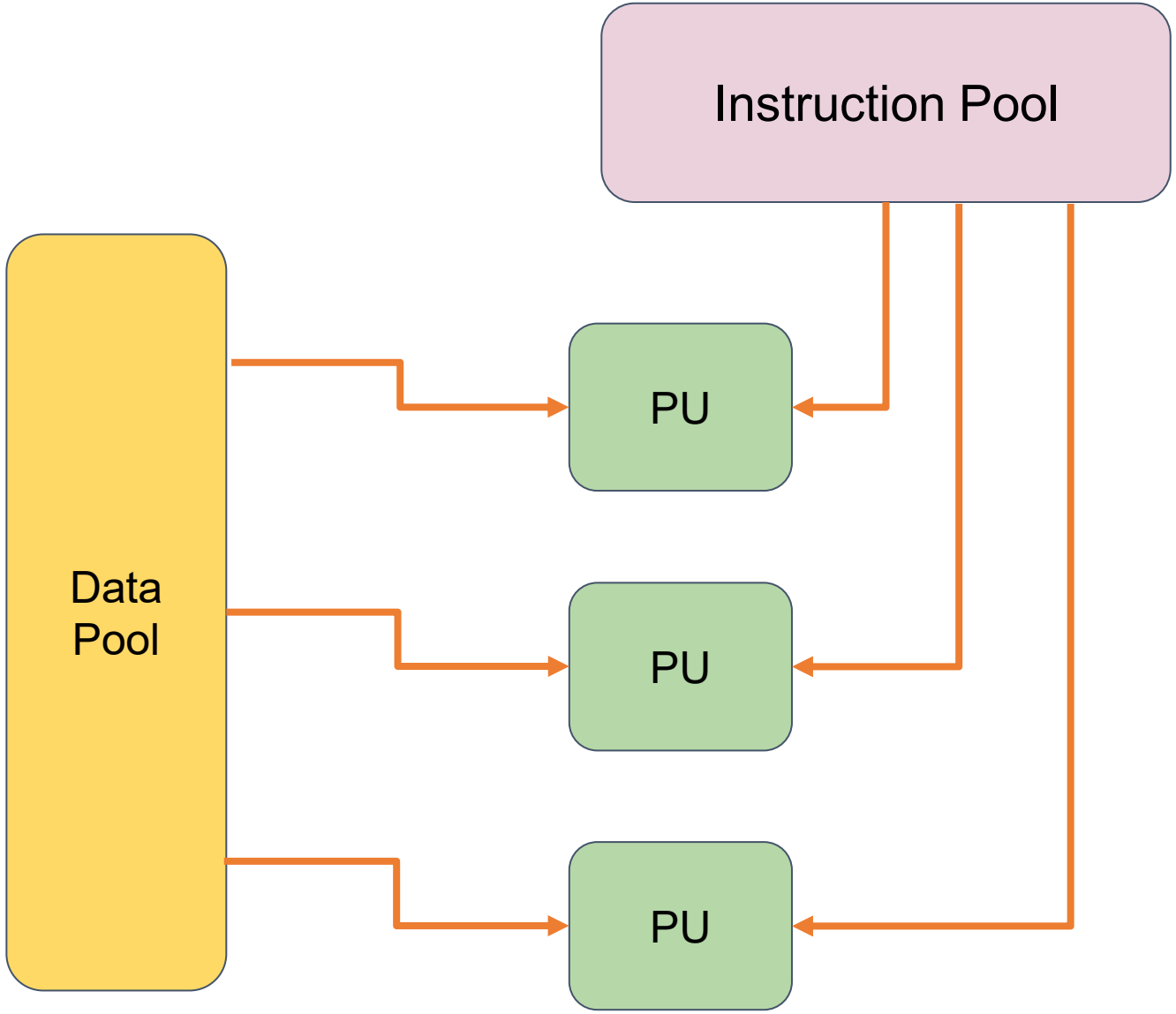
Multiple Instruction Multiple Data (MIMD)

- MIMD organization refers to a computer system capable of processing several programs at the same time.
- Most multiprocessor and multicomputer systems can be classified in this category.

Multiple Instruction Multiple Data (MIMD)

Multiple processors perform operations on different pieces of data, either independently or as part of shared memory space.

This means that several different instructions can be executed at the same time, using different data streams.



Multiple Instruction Multiple Data (MIMD)

Advantages

- Great for situations where multitasking is required

Disadvantages

- Much more complicated architecture so more expensive

Uses

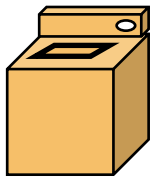
- Most modern PCs, Laptops, and Smart Phones

What’s Pipelining

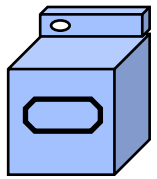
- You may not have heard of this concept, but you must’ve known how it works.

Laundry Example

- Say four students **A**nn, **B**rian, **C**athy, **D**ave each with a load of clothes to wash, dry, and fold.
- The washer takes 30 mins, dryer 40 minis, and folder 20 mins.



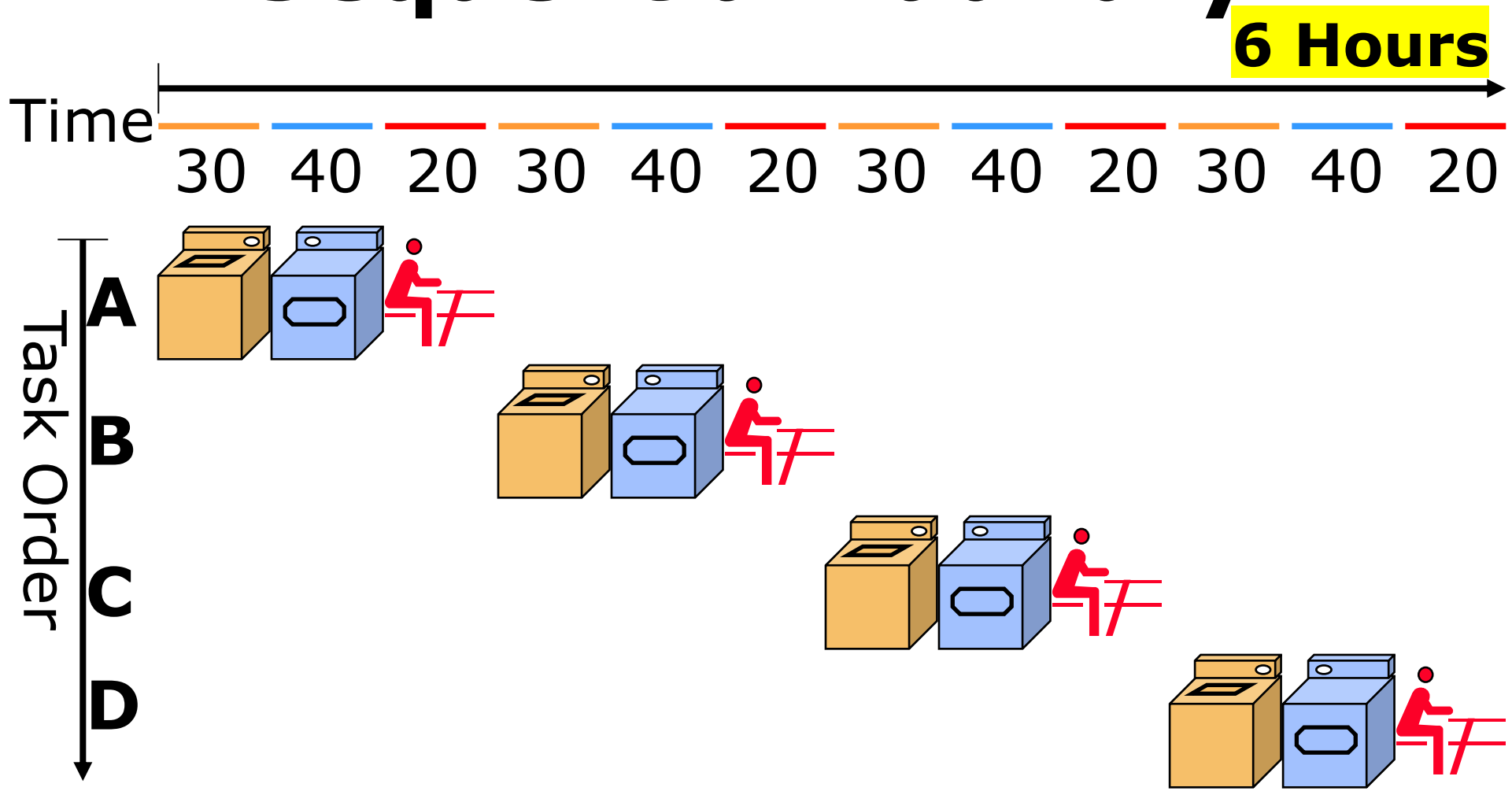
washer
30 mins



dryer
40 mins

folder
20 mins

Sequential Laundry

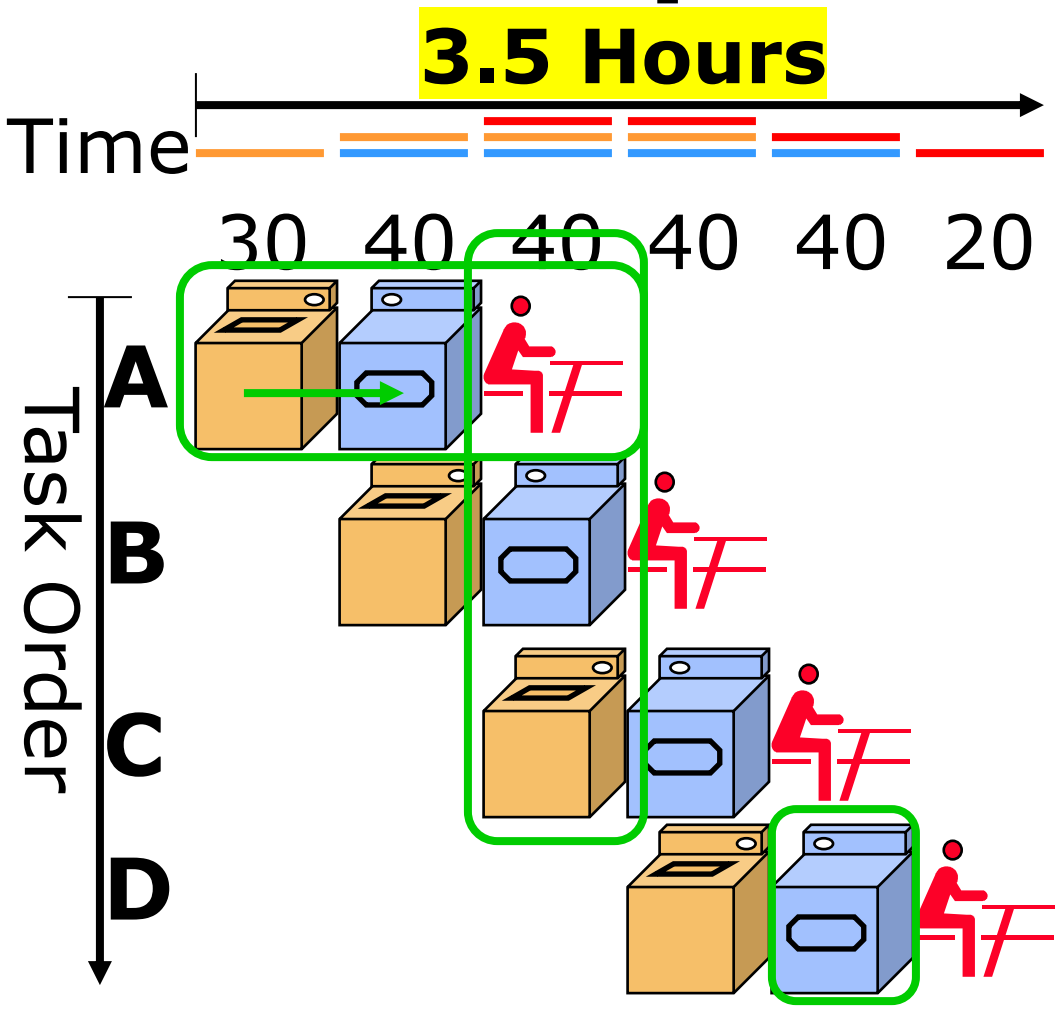


What would you do?

Sequential Laundry

- When they perform the laundry tasks in a sequential fashion, A does the laundry first, and only when A completes can B start. Similarly, C after B, and D after C.
- In this case, the sequential laundry will take up to 6 hours.
- But the question is, will anybody do it like this in real life? Actually, after A uses the washer, B can immediately start washing without further waiting. And then B can start drying after A finishes drying and start folding after A finishes folding.

Pipelined Laundry



Observations

- A task has a series of stages;
- Stage dependency:
  e.g., wash before dry;
- Multi tasks with overlapping stages;
- Parellel use diff resources to speed up;
- Slowest stage determines the finish time;

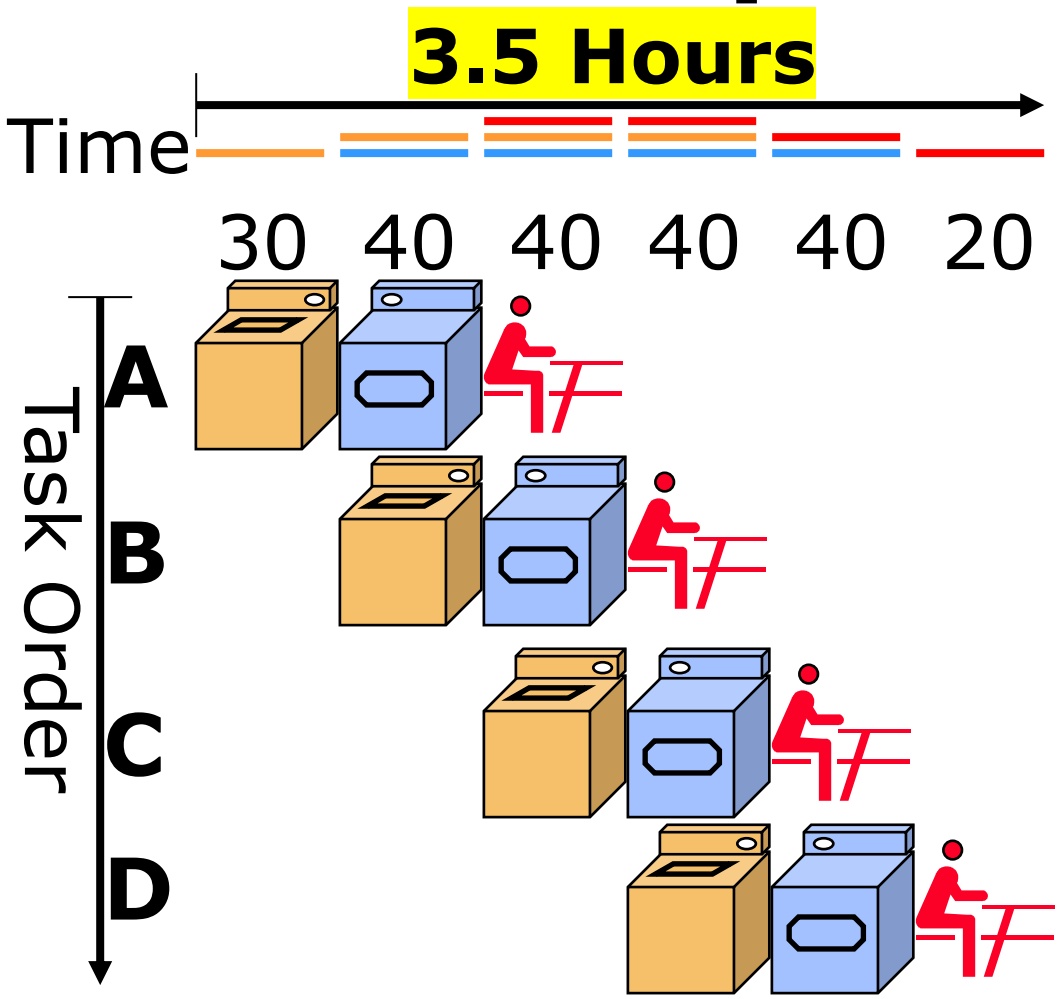
Pipelined Laundry

- Following this way, the laundry task execution will look like this. We call such laundry execution pipelined laundry, in which a laundry task can start before the previous one completes.
- The pipelined laundry takes only 3 and a half hours, which is much shorter than the 6 hours taken by the sequential laundry.
- Before we delve into the definition of pipelining in computer architecture, let’s see what observations we can get from this pipelined laundry example.
- First, a task has a series of stages. For example, a laundry task has three stages, washing, drying, and folding.

Pipelined Laundry

- Second, connecting stages have a dependency on each other. For example, you need to wash clothes before you dry them.
- When there are multiple tasks to run, simultaneously using different resources can accelerate the execution.
- A deeper observation is that the slowest stage determines the finish time.
- In this example, the dryer takes the longest time of 40 mins. And it’s obvious that the dryer decides when the last task will finish.

Pipelined Laundry



Observations

- No speed-up for the individual task; e.g., A takes 30+40+20=110 minutes(min)
- But speed up for average task execution time;

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

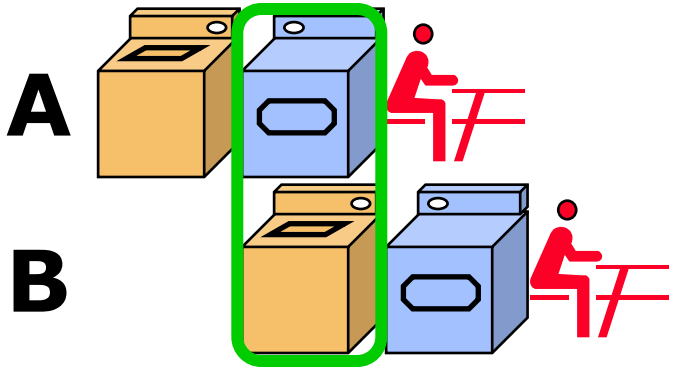
Pipelined Laundry

- Another important observation is that an individual task doesn’t become faster.
- For example. A takes 110 mins for laundry.
- On the contrary, it’s the average task execution time that becomes much shorter.
- In this case, four tasks take 3 and a half hours, with each taking about 52 mins, which is much shorter than the original 90 mins.

Pipelining

- An implementation technique whereby multiple instructions are overlapped in execution.

  e.g., B wash while A dry
- **Essence:** Start executing one instruction before completing the previous one.
- **Significance:** Make fast CPUs.



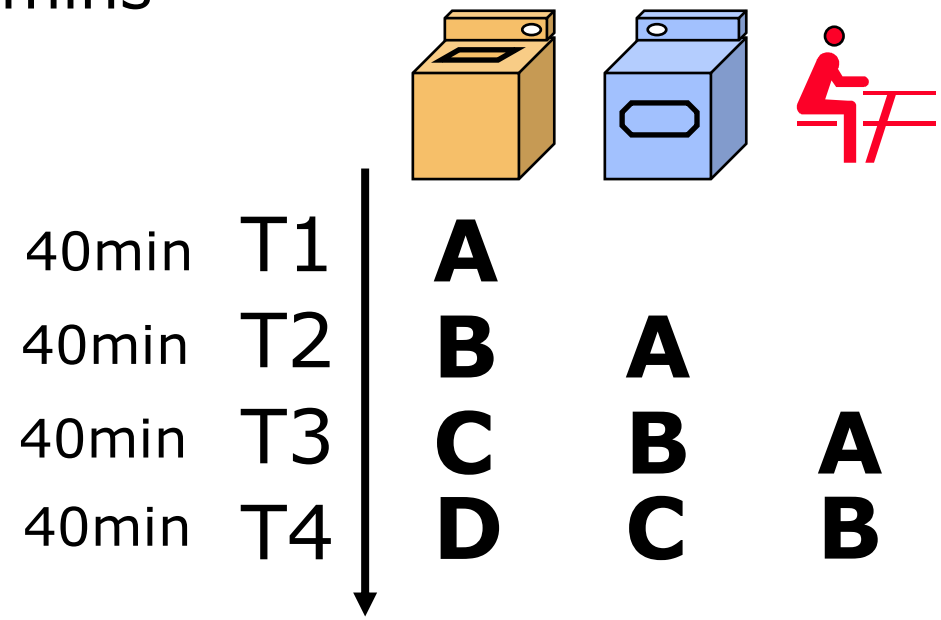
Balanced Pipeline

- Equal-length pipe stages

  e.g., Wash, dry, fold = 40 mins

  per unpipelined laundry time = 40x3 mins

  3 pipe stages – wash, dry, fold



Balanced Pipeline

- The ideal case for pipelining is a balanced pipeline. In a balanced pipeline, all pipe stages have equal duration.
- Consider again the laundry example with each stage taking 40 mins. Then the unpipelined laundry will take 40 times 3 mins to finish one task.
- Now let’s recap how pipelined laundry will perform the four tasks.

Balanced Pipeline

- In the first time duration T1, A uses the washer; In T2, A uses the dryer while B uses the washer; In T3, A uses the folder, B uses the dryer, while C uses the washer. Starting from T3, all resources are fully used in the same duration. Then the pipelined laundry takes 40 mins on average to complete one task.
- Based on this observation, we have two performance properties of the balanced pipeline.
- One is the time per instruction by pipeline is equal to time per instruction on the unpipelined machine over the number of pipe stages.
- The other is speed up by the pipeline is equal to the number of pipe stages.

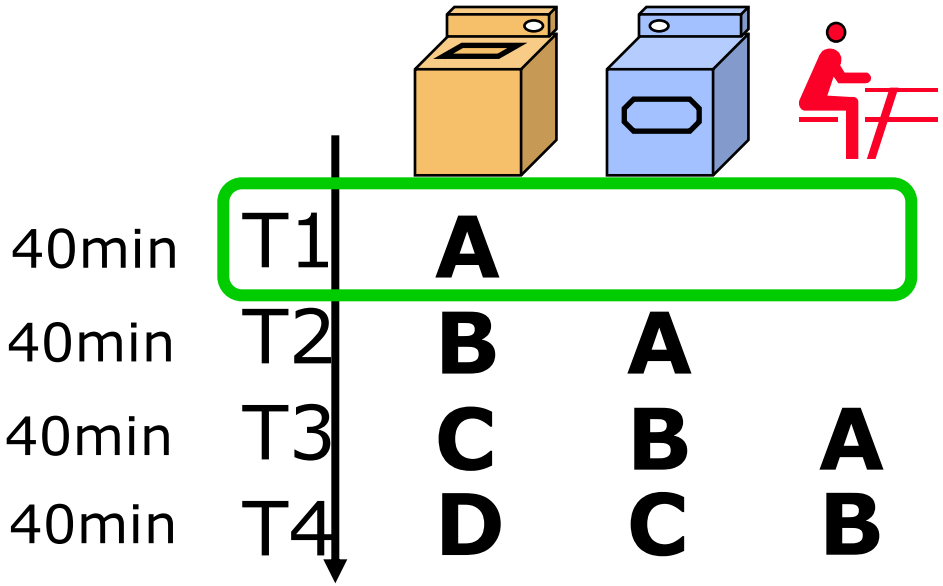
Balanced Pipeline

- Equal-length pipe stages

  e.g., Wash, dry, fold = 40 mins

  per unpipelined laundry time = 40x3 mins

  3 pipe stages – wash, dry, fold



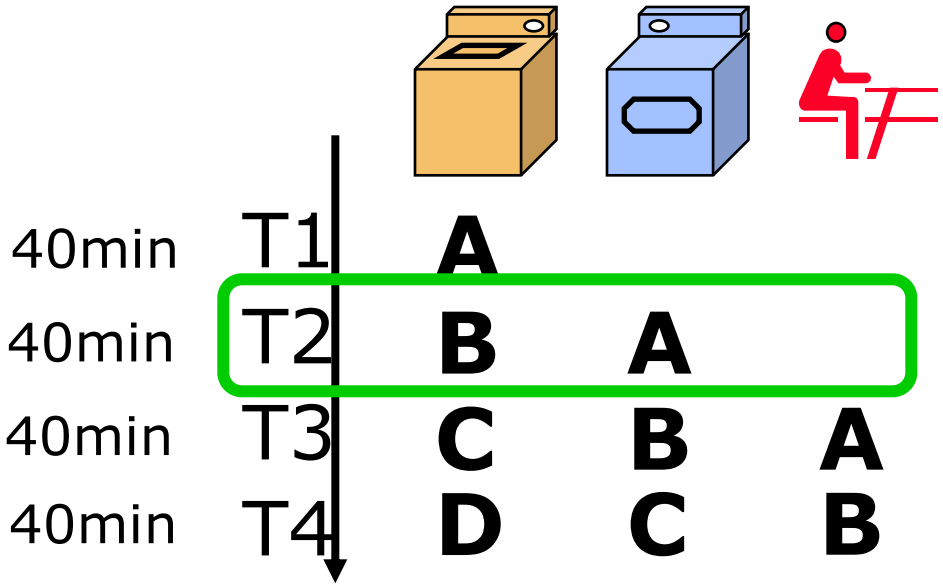
Balanced Pipeline

- Equal-length pipe stages

  e.g., Wash, dry, fold = 40 mins

  per unpipelined laundry time = 40x3 mins

  3 pipe stages – wash, dry, fold



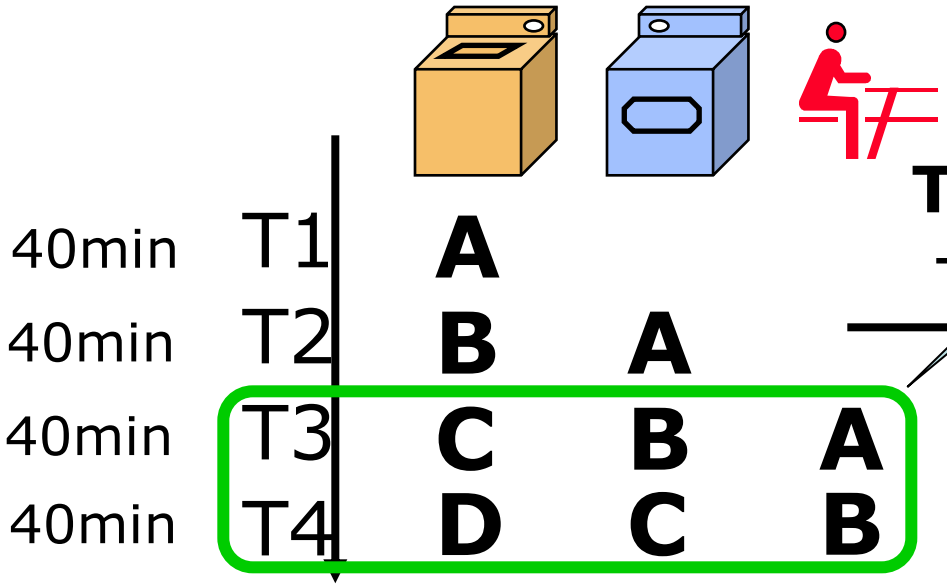
Balanced Pipeline

- Equal-length pipe stages

  e.g., Wash, dry, fold = 40 mins

  per unpipelined laundry time = 40x3 mins

  3 pipe stages – wash, dry, fold



One task/instruction per 40 mins

- Performance

Time per instruction by pipeline =

$$\frac{\text{Time per instr on unpipelined machine}}{\text{Number of pipe stages}}$$

Speed up by pipeline =

Number of pipe stages

Pipelining Terminology

- **Latency:** the time for an instruction to complete.
- **Throughput** of a CPU**:** the number of instructions completed per second.
- **Clock cycle:** everything in the CPU moves in lockstep; synchronized by the clock.
- **Processor Cycle: the** time required between moving an instruction one step down the pipeline;

  CPI: clock cycles per instruction

General Considerations

- Any operation that can be decomposed into a sequence of suboperations of about the same complexity can be implemented by a pipeline processor.
- The general structure of a four-segment pipeline is shown in next slide

General Considerations

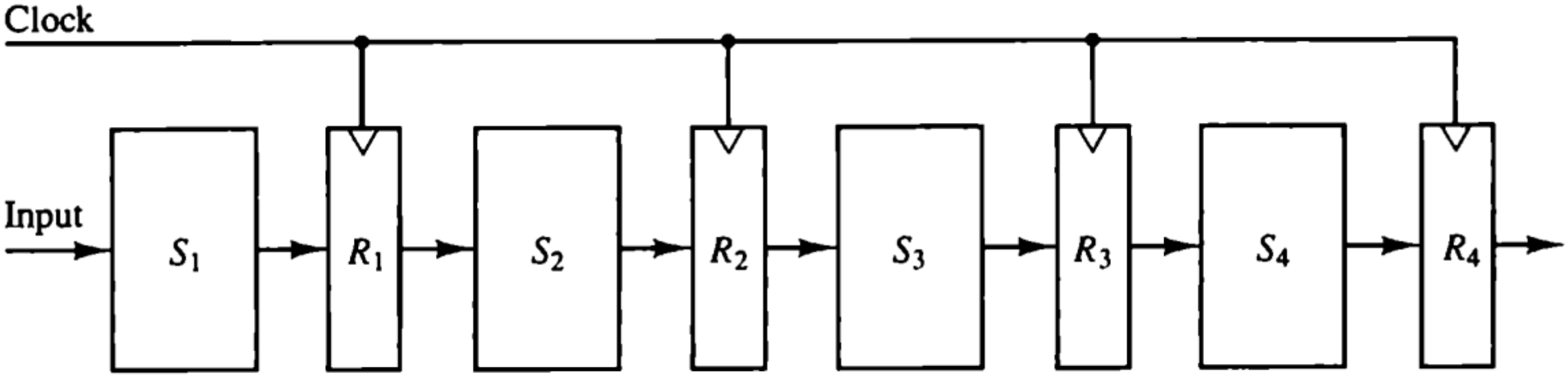


Figure 9-3 Four-segment pipeline.

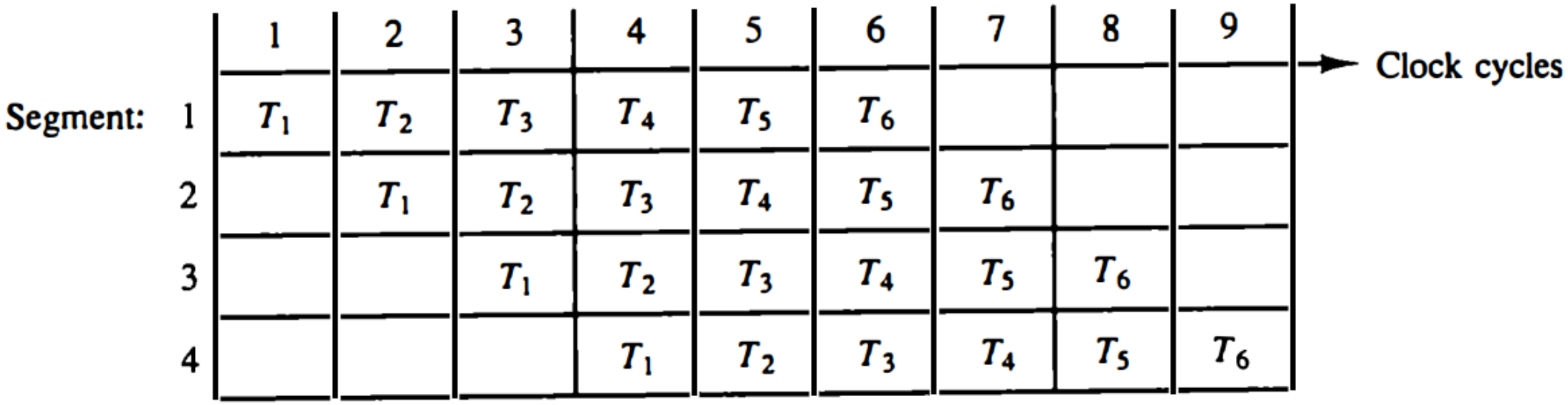
General Considerations

- The operands pass through all four segments in a fixed sequence. Each segment consists of a combinational circuit Si; that performs a suboperation over the data stream flowing through the pipe.
- The segments are separated by registers Ri; which hold the intermediate results between the stages.
- (Register like the person who carries the clothes (data) after finishing the wash stage to put them in the next stage (dry) and so on)
- Information flows between adjacent stages under the control of a common clock applied to all the registers simultaneously.

General Considerations

- The diagram shows six tasks T1 through T6 executed in four segments.

Figure 9-4 Space-time diagram for pipeline.



General Considerations

- Initially, task 11 is handled by segment 1. After the first clock, segment 2 is busy with T1, while segment 1 is busy with task T2
- Continuing in this manner, the first task T1 is completed after the fourth clock cycle.
- From then on, the pipe completes a task every clock cycle.
- No matter how many segments there are in the system, once the pipeline is full, it takes only one clock period to obtain an output.

General Considerations

- Now consider the case where a **k-segment** pipeline with a clock cycle time $\mathbf{t_p}$, is used to execute **n** tasks.
- The first task T1 requires a time equal to $\mathbf{kt_p}$, to complete its operation since there are **k** segments in the pipe.
- The remaining **n - 1** tasks emerge from the pipe at the **rate of one task per clock cycle** and they will be completed after a time equal to $\mathbf{(n - 1)t_p}$.

General Considerations

- Total time to complete **n** tasks using a **k-segment** pipeline requires

[k t_p + (n - 1) t_p]clock cycles = **t_p(k+n-1)**.

- Therefore, to complete **n** tasks using a **k-segment** pipeline requires **k + (n - 1)** clock cycles.

General Considerations

- For example, the diagram of Fig. 9-4 shows four segments and six tasks.
- The time required to complete all the operations is 4 + (6 - 1) = 9 clock cycles, as indicated in the diagram.

General Considerations

- Consider a non pipeline unit that performs the same operation and takes a time equal to $\mathbf{t_n}$. to complete each task.
- The total time required for **n** tasks is $\mathbf{nt_n}$
- The speedup of a pipeline processing over an equivalent non pipeline processing is defined by the ratio

$$S = \frac{nt_n}{(k + n - 1)t_p}$$

General Considerations

- As the number of tasks increases, **n** becomes much larger than **k - 1**, and **k + n - 1** approaches the value of **n**. Under this condition, the speedup becomes

$$S = \frac{t_n}{t_p}$$

General Considerations

- If we assume that **the time it takes to process a task is the same in the pipeline and non pipeline** circuits, we will have $\mathbf{t_n = kt_p}$,. Including this assumption, the speedup reduces to $S = \frac{kt_p}{t_p} = k$

- This shows that the theoretical maximum speedup that a pipeline can provide is **k**, where **k** is the number of segments in the pipeline.

Example

- Let the time it takes to process a suboperation in each segment be equal to $\mathbf{t_p = 20}$ ns.
- Assume that the pipeline has **k = 4** segments and executes **n = 100** tasks in sequence.
- The pipeline system will take $\mathbf{(k + n - 1)\ t_p = (4 + 99) \times 20 = 2060\ ns}$ to complete.
- Assuming that $\mathbf{t_n = kt_p = 4 \times 20 = 80\ ns}$, a non pipeline system requires $\mathbf{nkt_p = 100 \times 80 = 8000\ ns}$ to complete the **100** tasks.
- The speedup ratio is equal to **S= 8000/2060 = 3.88**.
- As the number of tasks increases, the speedup will approach **4**, which is equal to the number of segments in the pipeline.
- If we assume that t_n = 60 ns, the speedup becomes 60/20 = 3.

General Considerations

- There are various reasons why the pipeline cannot operate at its maximum theoretical rate.
- Different segments may take different times to complete their suboperation.
- The clock cycle must be chosen to equal the time delay of the segment with the maximum propagation time.
- This causes all other segments to waste time while waiting for the next clock.
- Moreover, it is not always correct to assume that a nonpipe circuit has the same time delay as that of an equivalent pipeline circuit.

General Considerations

- There are two areas of computer design where the pipeline organization is applicable.
- An arithmetic pipeline divides an arithmetic operation into suboperations for execution in the pipeline segments.
- An instruction pipeline operates on a stream of instructions by overlapping the fetch, decode, and execute phases of the instruction cycle.

References

- Computer System Architecture (3rd Edition) by M. Morris Mano
- Computer Architecture: A Quantitative Approach 5th Edition by John L. Hennessy
- SISD,SIMD,MISD,MIMD - A Level Computer Science (learnlearn.uk)

Thanks