High-Performance Architecture Lectures

1. Basic Computer Organization – *What is a processor and how it works?*
   – Design of PDcLX-1 processor
2. Program Execution – *How does a Code run on a Processor?*
   – Programming PDcLX-1 processor
3. **Pipelined Processor – *Increase Performance of our Processor***
   – **How much speed-up with pipelined processor? What it is the cost of it?**
4. Scalar Processor – *Increase Performance of our Processor*
   – PDcLX-2 and why ISA is important
5. On the way to Supercomputers – *Caches, Multicore Processor, Networks*
   – Beskow Supercomputer

Pipelined Processor

Stefano Markidis and Erwin Laure
KTH Royal Institute of Technology

Goal of This Lecture

Pipelined execution is a technique that enables microprocessor designers to increase the speed at which a processor operates.

This lecture will first introduce the concept of pipelining

We will then learn how to evaluate the benefits of pipelining, before I conclude with a discussion of the technique's limitations and costs.

The Life Cycle of an Instruction (Fetch-Execute Loop)

In the previous lecture, we learned that a computer repeats (3 + 1) basic steps:

1. *Fetch* the next instruction from the address stored in the program counter and load that instruction into the instruction register. Increment the program counter
2. *Decode* the instruction in the instruction register
3. *Execute* the instruction in the instruction register
   - *Read* the contents of registers
   - *Operate on* contents of registers
4. *Write* the result back to register

Four Stages for Pipeline = different stages of Fetch-Execute Loop

1. Fetch
2. Decode
3. Execute
4. Write (or “write-back”)

Are the four stages in a classic RISC1 pipeline

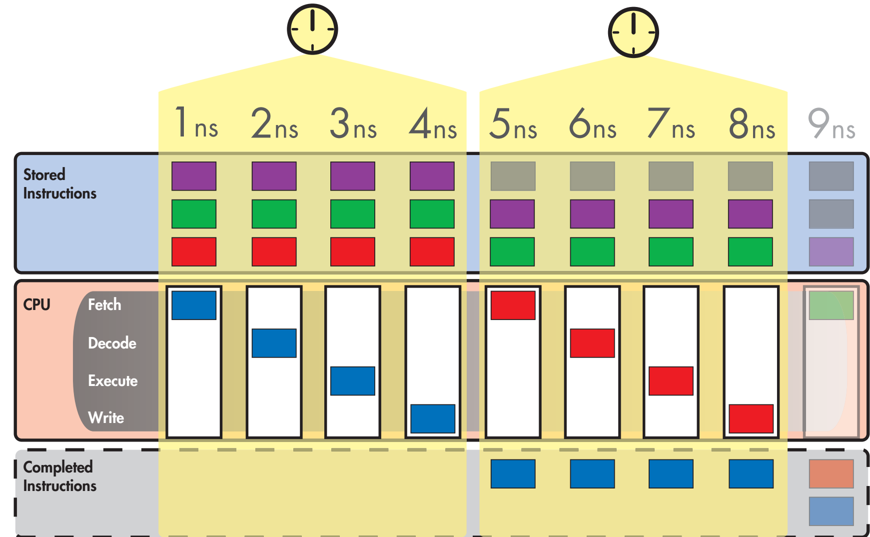
- **pipeline** = series of stages that each instruction in the code stream must pass through when the code stream is being executed

A Non-Pipelined Processor (our first PDcLX-1)

- Non-pipelined processors, also called *single-cycle* processors, work on one instruction at a time, **moving each instruction through all four phases of its lifecycle during the course of one clock cycle.**
  - We want the CPU's clock to run as fast as possible
  - On the hypothetical example CPU, the four phases of the instruction's lifecycle take a total of 4 ns to complete.
    - We should set the duration of the CPU clock cycle to 4 ns.

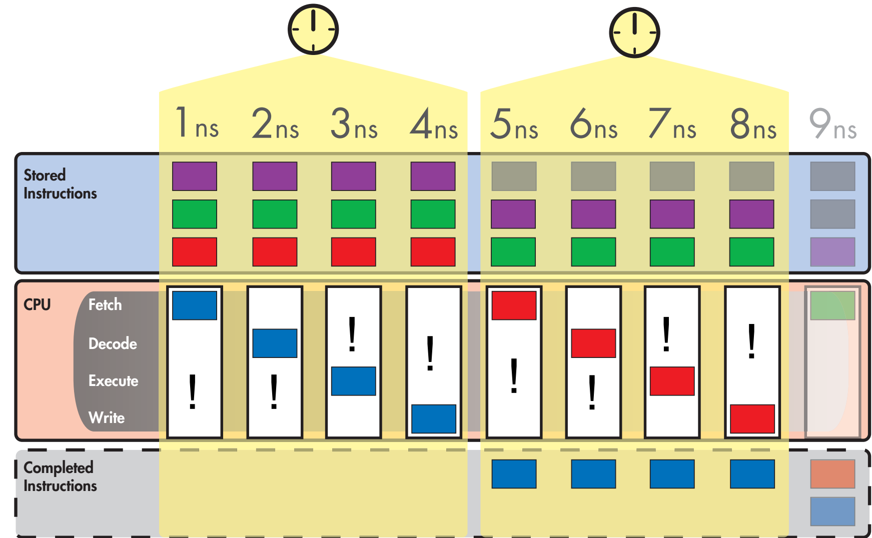
What is the instructions/s performance?

- The blue instruction leaves the code storage area, enters the processor, and then advances through the phases of its lifecycle over the course of the **4 ns clock period**
  - The end of the fourth ns is also the end of the first clock cycle, the red instruction can enter the processor at the start of a new clock cycle and go through the same process.
- This 4 ns sequence of steps is repeated until, after a total of 16 ns (or four clock cycles)
  - the processor has completed all four instructions at a completion rate of **0.25 instructions/ns**



Pros/Cons of A Non-Pipelined Processor

- Pro: Single-cycle processors are **simple to design**
- Con: **they waste a lot of hardware resources**
  - All of that white space in the diagram represents processor hardware that's sitting idle!

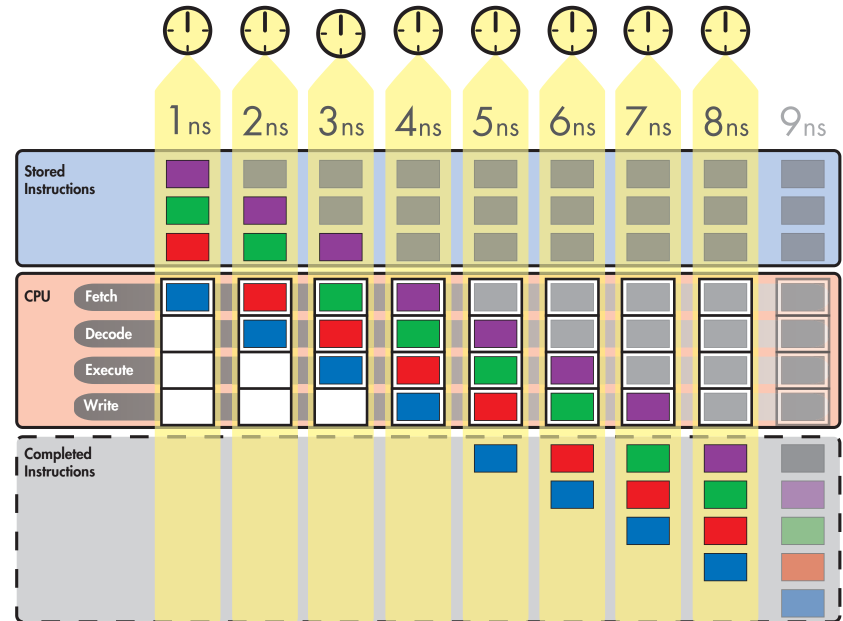


A Pipelined Processor

- Pipelining a processor means **breaking down** its **instruction execution** process into a series of **discrete *pipeline stages*** that can be completed in sequence by specialized hardware.
- Each pipeline stage corresponds to a phase in the standard instruction lifecycle:
  - **Stage 1:** Fetch the instruction from code storage.
  - **Stage 2:** Decode the instruction.
  - **Stage 3:** Execute the instruction.
  - **Stage 4:** Write the results of the instruction back to the register file.
- Note that the number of pipeline stages is called the ***pipeline depth***.
  - So the four-stage pipeline has a pipeline depth of four.

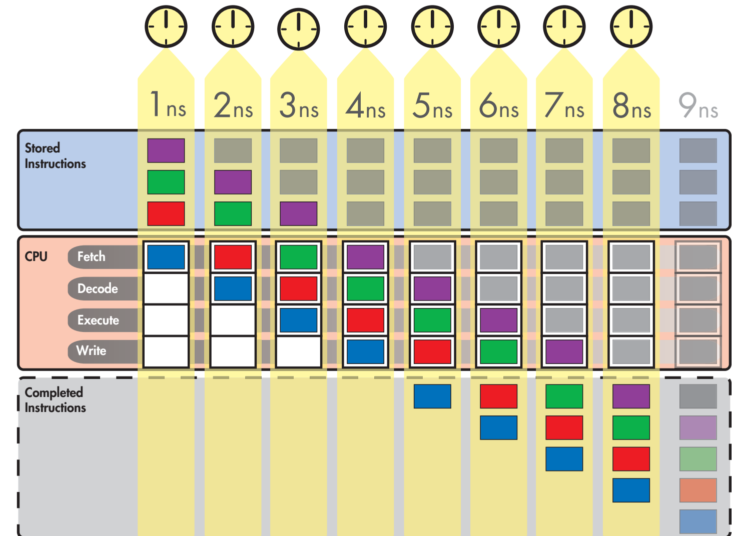
A Pipelined Processor – First 4 ns

- **0:** the blue instruction enters the fetch stage.
- **1 ns:** the blue instruction moves on to the decode stage, while the red instruction enters the fetch stage
- **2 ns:** the blue instruction advances to the execute stage, the red instruction advances to the decode stage, and the green instruction enters the fetch stage.
- **3 ns:** the blue instruction advances to the write stage, the red instruction advances to the execute stage, the green instruction advances to the decode stage, and the purple instruction advances to the fetch stage.
- **4 ns:** the blue instruction has passed from the pipeline and is now finished executing.
  - At the end of 4 ns, **the pipelined processor has completed one instruction.**



What is the instructions/s performance?

- At start of the fifth ns, the pipeline is now full and the processor can begin completing instructions at a rate of **one instruction per ns!**
- This **1 instruction/ns** completion rate is a **four-fold improvement** over the single-cycle processor's completion rate of 0.25 instructions/ns



Shrinking the Clock

- Because all of the pipeline stages must now work together simultaneously, **the clock is needed to coordinate the activity of the whole pipeline.**
  - **Shrink the clock cycle time to match the time it takes each stage to complete its work** so that at the start of each clock cycle, each pipeline stage hands off the instruction it was working on to the next stage in the pipeline.
- Because each pipeline stage in the example processor takes **1 ns to complete its work, we can set the clock cycle to be 1 ns in duration.**

Shrinking Instruction Execution Time

- Note that the total execution time for each individual instruction is not changed by pipelining.
  - It still takes an instruction 4 ns to make it all the way through the processor
    - that 4 ns can be split up into four clock cycles of 1 ns each, or it can cover one longer clock cycle, but it’s still the same 4 ns.
- Thus pipelining doesn’t speed up instruction execution time, but **it does speed up *program execution time* by parallel execution**

The Speedup from Pipelining

- Ideally, t**he speedup** in completion rate versus a single-cycle implementation that's gained from pipelining is **equal to the number of pipeline stages**
  - A four-stage pipeline yields a fourfold speedup in the completion rate versus a single-cycle pipeline, a five-stage pipeline yields a fivefold speedup, …
- This speedup is possible because the more pipeline stages there are in a processor, **the more instructions the processor can work on simultaneously**

Program Execution Time and Completion Rate

In general, a program’s execution time is equal to the total number of instructions in the program divided by the processor’s instruction completion rate (number of instructions completed per ns)

```
program execution time = number of instructions in program / instruction
completion rate
```

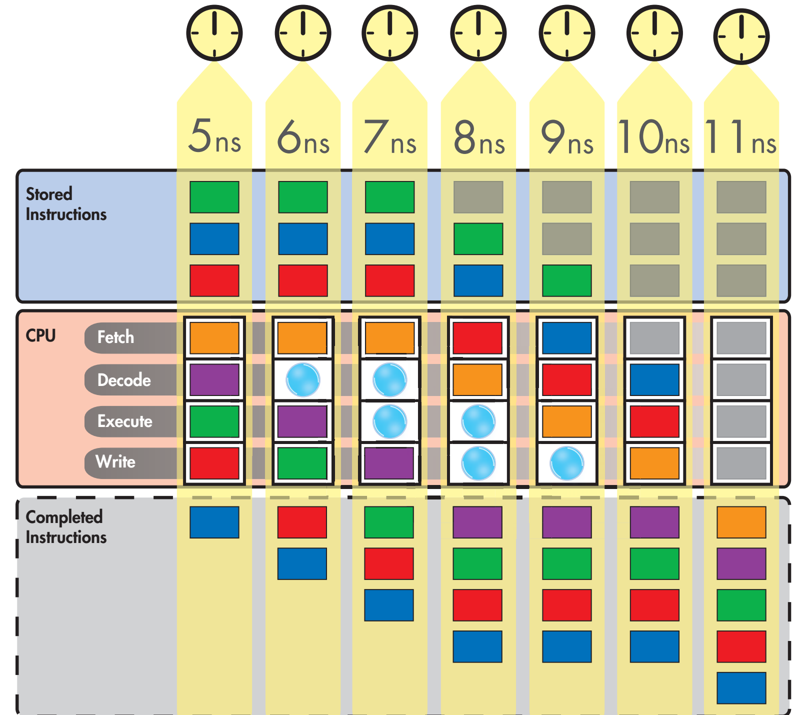
- If the program that the **single-cycle processor** is running consisted of only the four instructions depicted, that program would have a program execution time of **16 ns**, or 4 instructions / 0.25 instructions/ns
  - If the program consisted of, say, seven instructions, it would have a program execution time of 7 instructions / 0.25 instructions/ns = 28 ns

The Relationship Between Completion Rate and Program Execution Time

- If you look at the "Completed Instructions" box of the four-stage processor, you'll see that a total of **five instructions** have been completed at the start of the ninth nanosecond.
- The non-pipelined processor sports **two completed instructions** at the start of the ninth nanosecond.
  - not a fourfold improvement over two completed instructions in the same time period, why?
    - Remember that it took the pipelined processor 4 ns initially to **fill up with instructions**; the pipelined processor did not complete its first instruction until the end of the fourth nanosecond..

Possible Problem - Pipeline Stalls

- Sometimes, instructions get hung up in one pipeline stage for multiple cycles.
  - When it happens, the pipeline is said to ***stall***.
- When the pipeline stalls
  - all of the instructions in the stages **below** the one where the stall happened continue **advancing normally**
  - the stalled instruction just sits in its stage, and all the instructions behind it back up.
- **Pipeline stalls** - or **bubbles** - **reduce** a pipeline's average **instruction throughput**



Limits to Pipelining: Balance the Pipelining

- Each pipeline stage must take exactly one clock cycle to complete
  - The clock pulse that coordinates all the stages can be **no faster than the pipeline's slowest stage**
- As slice the pipeline more finely in order to add stages and increase throughput
  - the individual stages **get less and less uniform in length** and complexity, with the result that the processor's overall instruction execution time gets longer.
  - One of the most difficult challenges that the CPU designer faces is that of **balancing the pipeline** so that no one stage has to do more work to do than any other.

The Cost of Pipelining

- Pipelining requires a nontrivial amount of extra bookkeeping and buffering logic to implement, **so it incurs an overhead cost in transistors and die space**.
- This overhead cost increases with pipeline depth, so that a processor with a very deep pipeline spends a significant amount of its transistor budget on pipeline-related logic.

Key-Points

- Pipelining is a technique to parallelize different stages of the fetch-execute cycle
- Ideal speed-up wrt single-cycle processor performance is equal to the pipeline depth