جامعة ألأنبار

كلية علوم الحاسوب وتكنولوجيا المعلومات

قسم أنظمة شبكات الحاسوب

المرحلة الثانيه

Computer Architecture

التدريسي: أ.م.د عمر منذر حسين

A Closer Look at Instruction Set Architectures

5.5 Instruction Level Pipelining:

- Some CPUs break the fetch-decode execute cycle down into smaller steps, where some of these smaller steps can be performed in parallel.
- This overlapping speeds up execution. This method, used by all current CPUs, is known as **pipelining**.
- Suppose the fetch-decode-execute cycle were broken into the following mini-steps:
  1. Fetch instruction
  2. Decode opcode
  3. Calculate effective address of operands
  4. Fetch operands
  5. Execute instruction
  6. Store result
- Suppose we have a six-stage pipeline. S1 fetches the instruction, S2 decodes it, S3 determines the address of the operands, S4 fetches them, S5 executes the instruction, and S6 stores the result.
- Each step in a computer pipeline completes a part of an instruction.
- Like the automobile assembly line, different steps are completing different parts of different instructions in parallel.
- Each of the steps is called a pipeline stage. The stages are connected to form a pipe. Instructions enter at one end, progress through the various stages, and exit at the other end.
- The goal is to balance the time taken by each pipeline stage (i.e., more or less the same as the time taken by any other pipeline stage).
- If the stages are not balanced in time, after a while, faster stages will be waiting on slower ones.
- To see an example of this imbalance in real life, consider the stages of doing laundry. If you have only one washer and one dryer, you usually

end up waiting on the dryer. If you consider washing as the first stage and drying as the next, you can see that the longer drying stage causes clothes to pile up between the two stages.

- If you add folding clothes as a third stage, you soon realize that this stage would consistently be waiting on the other, slower stages
- Figure 4.1 provides an illustration of computer pipelining with overlapping stages.
- We see each clock cycle and each stage for each instruction (where S1 represents the fetch, S2 represents the decode, S3 is the calculate state, S4 is the operand fetch, S5 is the execution, and S6 is the store).

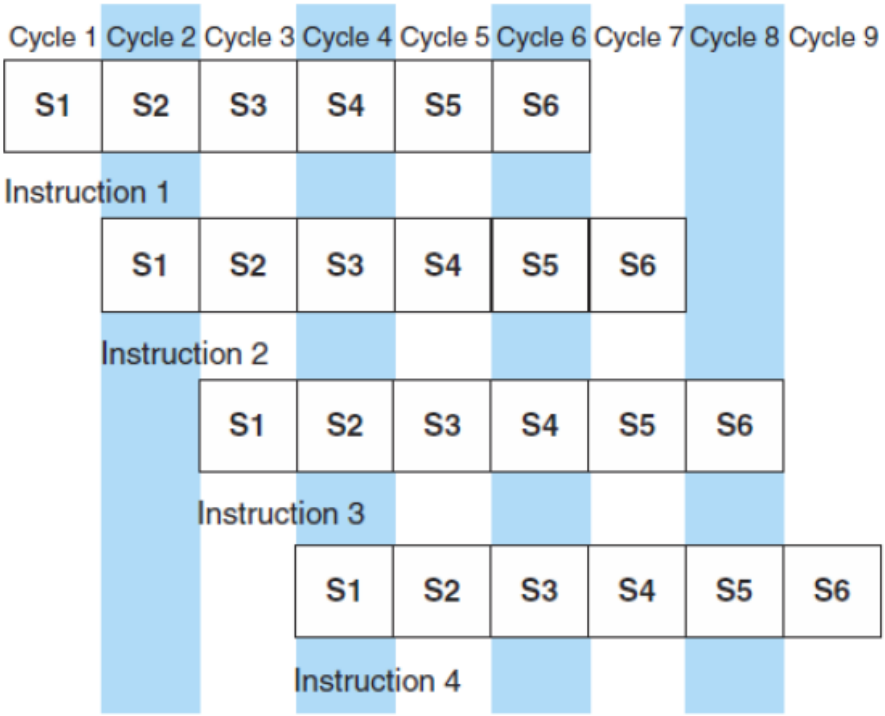

FIGURE 4.1 Four Instructions Going through a 6-Stage Pipeline

- We see from Figure 4.1 that once instruction 1 has been fetched and is in the process of being decoded, we can start the fetch on instruction 2.

- When instruction 1 is fetching operands, and instruction 2 is being decoded, we can start the fetch on instruction 3. Notice these events can occur in parallel, very much like an automobile assembly line.
- Suppose we have a k-stage pipeline. Assume the clock cycle time is $\mathbf{t_p}$, that is, it takes $\mathbf{t_p}$ time per stage.
- Assume also we have **n** instructions (often called tasks) to process.
- Task 1 (T1) requires ($k \times \mathbf{t_p}$) time to complete.
- The remaining (n - 1) tasks emerge from the pipeline one per cycle, which implies a total time for these tasks of [(n - 1) × $\mathbf{t_p}$].
- Therefore, the time to complete **n** tasks using a k-stage pipeline requires:

$$T_p = (k \times t_p) + (n-1) \times t_p = (k+n-1) \times t_p$$

Or [*k* + (*n* - 1)] clock cycles

The execution time without pipeline:

$$T_{np} = n \times t_n \qquad t_n = k \times t_p$$

Let's calculate the speedup we gain using a pipeline. Without a pipeline, the time required is ($n*t_n$) cycles, where ($t_n = k * t_p$). Therefore, the speedup (time without a pipeline divided by the time using a pipeline) is:

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

If we take the limit of this as *n* approaches infinity, we see that (*k* + *n* - 1) approaches *n*, which results in a theoretical speedup of:

$$Speedup = \frac{k \times t_p}{t_p} = k$$

- The theoretical speedup, **k**, is the number of stages in the pipeline.
- Let′s look at an example. Suppose we have a 4-stage pipeline, where:
  - S1 = fetch instruction
  - S2 = decode and calculate effective address
  - S3 = fetch operand
  - S4 = execute instruction and store results
- We must also assume the architecture provides a means to fetch data and
- instructions in parallel. This can be done with separate instruction and data paths; however, most memory systems do not allow this.
- Instead, they provide the operand in cache, which, in most cases, allows the instruction and operand to be fetched simultaneously.
- There are several conditions that result in "pipeline conflicts," which keep us from reaching the goal of executing one instruction per clock cycle.
- These include:
  - Resource conflicts
  - Data dependencies
  - Conditional branch statements
- Resource conflicts are a major concern in instruction-level parallelism.
- For example, if one instruction is storing a value to memory while another is being fetched from memory, both need access to memory.
- Typically, this is resolved by allowing the instruction executing to continue, while forcing the instruction fetch to wait.
- Certain conflicts can also be resolved by providing two separate pathways:
- one for data coming from memory and another for instructions coming from memory.
- Specialized hardware can also be used to detect these conflicts and route data through special paths that exist between various stages of the pipeline.

- This reduces the time necessary for the instruction to access the required operand.

Example:

- A linear pipe with 4 stages process 1000 tasks the clock period is 60 nsec calculate.

1-The execution time

2-The speed up

Sol:

1-The execution time:

$$T_{np} = n \times t_n \qquad t_n = k \times t_p$$

$$T_{np} = 1000 \times 4 \times 60 \times 10^{-9} \qquad T_{np} = 240 \mu \text{sec}$$

$$T_p = (k \times t_p) + (n-1) \times t_p = (k + n - 1) \times t_p$$

$$T_p = (4 + 1000 - 1) \times 60 \times 10^{-9} = 60.18 \mu \text{sec}$$

2-The speed up:

$$Speedup \quad S = \frac{n t_n}{(k + n - 1) \times t_p}$$

$$S = \frac{1000 \times 4 \times 60 \times 10^{-9}}{(4 + 1000 - 1) \times 60 \times 10^{-9}} = 3.988$$