Processor Architecture

- **Superscalar execution**: already discussed.
- **Out-of-order execution**: Out-of-order execution happens in **multiprocessors**.
  - In this approach, the instructions are executed in an order of **availability** of **operands** and the **execution units**, rather than by their original order in a program.
  - By doing so, the processor can **avoid being idle.** While waiting for the preceding instruction to complete, in the meantime, the processor processes the next instructions that are able to run immediately and independently.
  - It is also known as **dynamic execution**.
  - This approach is followed in the most high-performance CPUs.

See Stallings Chapter: Instruction Level Parallelism and Superscalar Processors

Processor Architecture

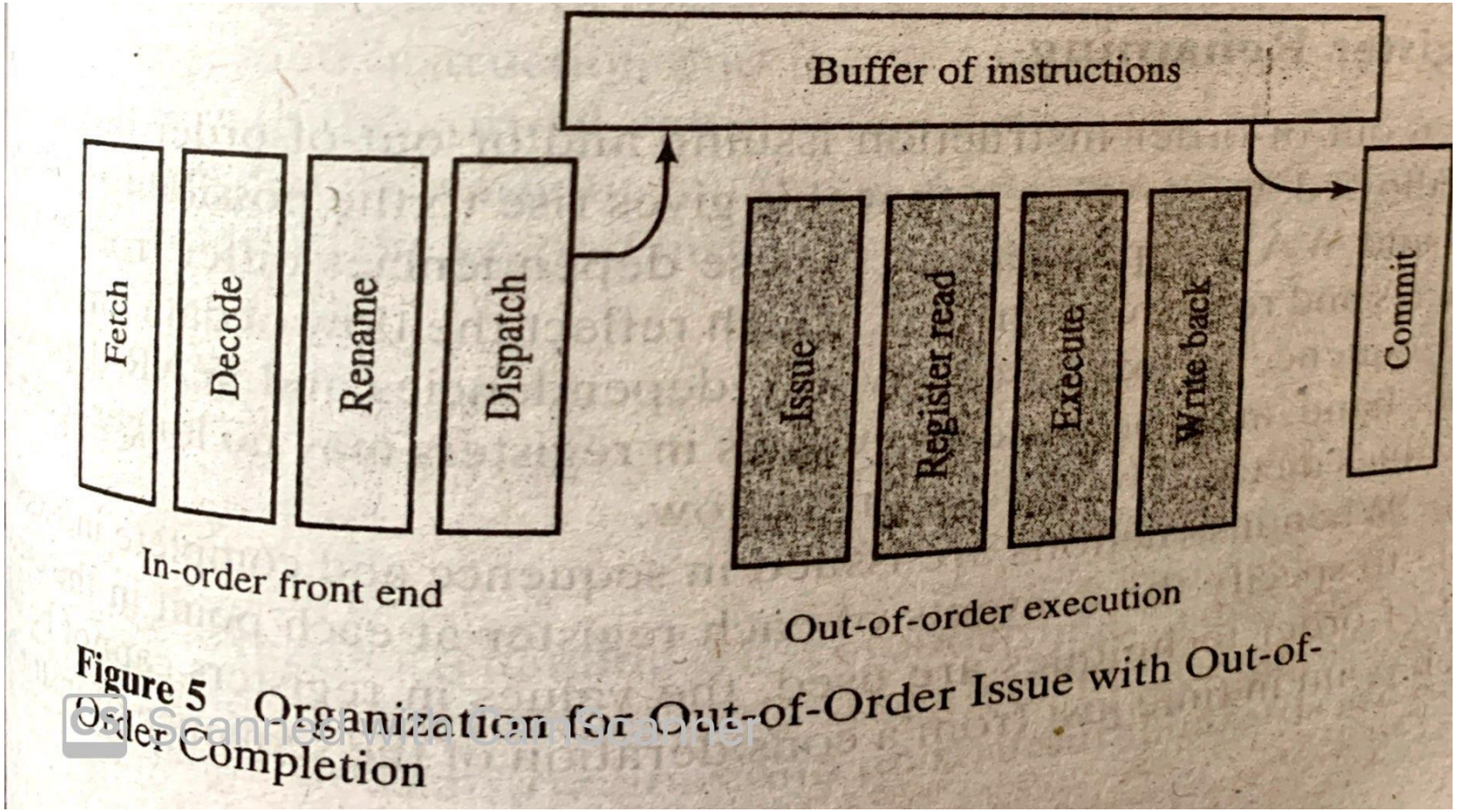

Figure 5 Organization for Out-of-Order Issue with Out-of-Order Completion

Processor Architecture

- The **out-of-order approach** performs the execution of instructions into the following steps:
  - **Step 1**: Instruction is **fetched** from the memory.
  - **Step 2**: Instruction is **dispatched** to an **instruction queue** (instruction buffer/reservation station).
  - **Step 3**: The instruction waits in the queue until its input operands are available. The **instruction** is allowed to **leave** the **queue** for the execution **before** the **older instructions**. The instruction does not need to wait in the queue until its turn. Whenever the operand is available the instruction will leave the queue or buffer for execution.
  - **Step 4**: Then, the **instruction** is sent to the appropriate **functional/execution unit** and **executed** by that unit.
  - **Step 5**:The **results** are **queued**.
  - **Step 6**: When all the **older instructions** have their results **written** back to the **register**, then **this result** is **written** back to the **register**. This is called the **graduation** or **retire stage**.

Processor Architecture

- **Register renaming**
  - When **instructions** are issued in a **sequence** and **complete** in a **sequence**, it is **possible** to **specify** the **content** of each **register** at each point of the execution.
  - When the **out-of-order execution** is used, it is **not** always **possible** to know the contents of the registers at each point of execution as the **sequence** is **not maintained**. This results the issue of **data dependency** (WAW and WAR).
  - The values are in conflict for the use of the registers. **Multiple instructions** also **compete** for the use of the **same register**. The processor must resolve this conflict by occasionally stalling a pipeline stage.
  - One **solution** to this conflict is **duplication of resources**.
  - The method of **duplication of resources** is known as **register renaming**. **Register names** are **replaced** by **value names** dynamically for each instruction destination operand.
  - Registers are allocated dynamically by the processor hardware, and they are associated with the values needed by the instructions at various points in time.
  - When a new register value is created as the destination operand, a new register is created for that value. Any subsequent instruction that has that value as a source operand, must rename that register.
  - It is the **abstraction** of the **logical registers** from the **physical registers**.
  - Program runs faster.

Processor Architecture

- For example, consider the following instructions:

  I1: R3 ← R3 op R5

  I2: R4 ← R3 + 1

  I3: R3 ← R5 + 1

  I4: R7 ← R3 op R4

  After register renaming,

  I1: $R3_b \leftarrow R3_a$ op $R5_a$

  I2: $R4_b \leftarrow R3_b + 1$

  I3: $R3_c \leftarrow R5_a + 1$

  I4: $R7_b \leftarrow R3_c$ op $R4_b$

Processor Architecture

- Another example of register renaming:

  R1 ← M[1024]
  R1 ← R1 + 2
  M[1032] ← R1
  R1 ← M[2048]
  R1 ← R1 + 4
  M[2056] ← R1

  After register renaming,

  R1 ← M[1024]
  R1 ← R1 + 2
  M[1032] ← R1
  R2 ← m[2048]
  R2 ← R2 + 4
  M[2056] ← R2

Processor Architecture

- **Memory disambiguation**
  - This technique is used by the high performance out-of-order processors that execute memory access instructions (LOAD and STORE) out of program order.
  - The mechanism for performing memory disambiguation, implemented using digital logic inside the microprocessor core, detects true dependencies between memory operations at execution time and allows the processor to recover when a dependence has been violated.
  - For example, in the following two instruction, the processor can not determine, before the execution, if the two memory locations are the same or not. Because, it depends on the value of R2 and R3.

    M[R2+2] ← R1 (STORE operation)

    R4 ← M[R3+1] (LOAD operation)
  - If the values are different, then the instructions are independent and can be successfully executed out-of-order.
  - If the locations are same, then the Load operation is dependent on the Store operation, this known as **ambiguous dependence**.

Processor Architecture

- This dependence comes in three forms:
  - ✓ **Read-After-Write** (RAW) dependencies/true dependencies: this arise when a load operation reads a value from the memory that was produced by the most recent preceding store operation to that same address.
  - ✓ **Write-After-Read** (WAR) dependencies/anti dependencies: this arise when a store operation writes a value to the memory that a preceding load operation reads.
  - ✓ **Write-After-Write** (WAW) dependencies/output dependencies: this arise when two store operations write values to the same memory address.