Amongst significant reasons z196 is faster than previous generation mainframe models is that the z196 has a superscalar microprocessor with **out-of-order (OOO)** execution. With OOO, instructions may not execute in the original *program* order, although results are presented in the original order. OOO allows, for instance, several instructions to complete while another is waiting.

> A scalar processor is a processor that is based on a single-issue architecture, which means that only a single instruction is executed at a time. A superscalar processor allows concurrent execution of instructions by adding additional resources onto the microprocessor to achieve more parallelism by creating multiple pipelines, each working on its own set of instructions.
>
> A superscalar processor is based on a multi-issue architecture. In such a processor, where multiple instructions can be executed at each cycle, a higher level of complexity is reached, because an operation in one pipeline stage might depend on data in another pipeline stage. Therefore, a superscalar design demands careful consideration of which instruction sequences can successfully operate in a long pipeline environment. On the z196, up to three instructions can be decoded per cycle and up to five instructions/operations can be executed per cycle. Execution can occur out of (program) order.
>
> Many challenges exist in creating an efficient superscalar processor. The superscalar design of the PU has made big strides in avoiding address generation interlock (AGI) situations. Instructions requiring information from memory locations can suffer multi-cycle delays to get the desired memory content, and because high-frequency processors wait faster, the cost of getting the information might become prohibitive.

Generally, out of order execution is a paradigm used in most high-performance microprocessors to make use of instruction cycles that would otherwise be wasted by a certain type of costly delay.

Arguably the first machine to use out-of-order execution was probably the CDC 6600 (1964), which used a scoreboard to resolve conflicts. In modern usage, such score boarding is considered to be in-order execution, not out-of-order execution, since such machines stall on the first RAW (Read After Write) conflict. Strictly speaking, such machines initiate execution in-order, although they may complete execution out-of-order.

About three years later, the IBM 360/91 (1966) introduced Tomasulo's algorithm, supporting full out-of-order execution. In 1990, IBM introduced the first out-of-order microprocessor, the POWER1, although out-of-order execution was limited to floating point instructions only.

Basic Concepts

In-order processors

In earlier processors, the processing of instructions is normally done in these steps:

1. Instruction fetch.
2. If input operands are available (in registers for instance), the instruction is dispatched to the appropriate functional unit. If one or more operand is unavailable during the current

clock cycle (generally because they are being fetched from memory), the processor stalls until they are available.
3. The instruction is executed by the appropriate functional unit.
4. The functional unit writes the results back to the register file.

Out-of-order processors

This paradigm breaks up the processing of instructions into these steps:

1. Instruction fetch.
2. Instruction dispatch to an instruction queue (also called instruction buffer or reservation stations).
3. The instruction waits in the queue until its input operands are available. The instruction is then allowed to leave the queue before earlier, older instructions.
4. The instruction is issued to the appropriate functional unit and executed by that unit.
5. The results are queued.
6. Only after all older instructions have their results written back to the register file, then this result is written back to the register file. This is called the graduation or retire stage.

The key concept of OOO processing is to allow the processor to avoid a class of stalls that occur when the data needed to perform an operation are unavailable. In the outline above, the OOO processor avoids the stall that occurs in step (2) of the in-order processor when the instruction is not completely ready to be processed due to missing data.

OOO processors fill these "slots" in time with other instructions that *are* ready, then re-order the results at the end to make it appear that the instructions were processed as normal. The way the instructions are ordered in the original computer code is known as *program order*, in the processor they are handled in *data order*, the order in which the data, operands, become available in the processor's registers. Fairly complex circuitry is needed to convert from one ordering to the other and maintain a logical ordering of the output; the processor itself runs the instructions in seemingly random order.

The benefit of OOO processing grows as the instruction pipeline deepens and the speed difference between main memory (or cache memory) and the processor widens. On modern machines, the processor runs many times faster than the memory, so during the time an in-order processor spends waiting for data to arrive, it could have processed a large number of instructions.

System z CMOS uses OOO

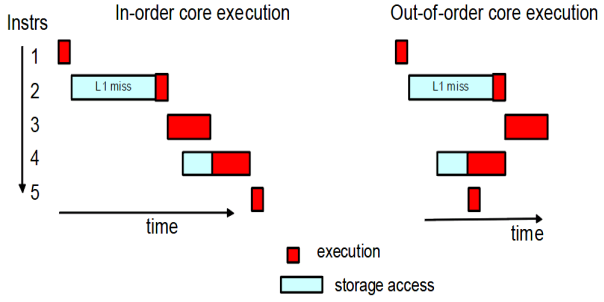
The z196 is the first System z CMOS to implement an out-of-order (OOO) core. OOO yields significant performance benefit for compute intensive applications through re-ordering instruction execution, allowing later (younger) instructions to be executed ahead of a stalled instruction, and re-ordering storage accesses and parallel storage accesses. OOO maintains good performance growth for traditional applications. Out-of-order (OOO) execution can improve performance by**:**

* Re-ordering instruction execution
  Instructions stall in a pipeline because they are waiting for results from a previous instruction or the execution resource they require is busy. In an in-order core, this stalled

instruction stalls all later instructions in the code stream. In an out-of-order core, later instructions are allowed to execute ahead of the stalled instruction.

* Re-ordering storage accesses
  Instructions which access storage can stall because they are waiting on results needed to compute storage address. In an in-order core, later instructions are stalled. In an out-of-order core, later storage-accessing instructions which can compute their storage address are allowed to execute.

* Hiding storage access latency
  Many instructions access data from storage. Storage accesses can miss the L1 and require 10 to 500 additional cycles to retrieve the storage data. In an in-order core, later instructions in the code stream are stalled. In an out-of-order core, later instructions that are not dependent on this storage data are allowed to execute.



In the example, the left side is showing an in-order core execution. Instruction 2 has a big delay due to an L1 miss, and next instructions wait until instruction 2 finishes.

On usual in-order execution, next instruction waits until the previous one finishes. Using OOO core execution, shown on the right side of the example, instruction 4 could start its storage access and execution while the instruction 2 is waiting for data, if no dependencies exist between both instructions. So, when L1 miss is solved, instruction 2 could also start its execution while instruction 4 is executing, instruction 5 could need the same storage data required by instruction 4, and as soon as this data is on L1, instruction 5 starts execution at the same time. The z196 superscalar PU core can execute up to five instructions/operations per cycle.

OOO execution does not change any program results. Execution can occur out of (program) order, but all program dependencies are honored, ending up with same results of the in order (program) execution. This implementation requires special circuitry to make execution and memory accesses appear in order to software.

Memory address generation and memory accesses can occur out of (program) order. This capability can provide a greater exploitation of the z196 superscalar core, and can improve system performance.

- - -