First part of topic 13

„Modern processor solutions (pipeline, hazard, out-of-order execution, speculative execution, superscalar-, VLIW- and vector processors)”

During their operation, the processors basically repeat a so-called the fetch-execute. This sequence of steps can be divided into 5 phases in a simple approach.

1. Instruction fetch (IF): Reading the following machine code instruction from the memory address currently have in the program counter register.
2. Instruction decoding (ID): Interpretation of the scanned machine code instruction. What is the operation? Where are the operands? (In which register or at which memory address? What addressing method should be used to determine their memory address at all?) Where will the result be saved?
3. Operand fetch (OF): Loading of operands into the input registers of the operation unit (e.g. ALU) using the appropriate addressing mode.
4. Execution (EX): Execution of the operation during which the desired result is determined (e.g. in the output register of the ALU) based on the values of the operands.
5. Write back (WB): Saveing the result to the memory address or general-purpose register defined by the machine code. Update the content of the program counter register (usually incremented by the length of the executed instruction).

In modern processors, this does not work sequentially (as opposed to Neumann principles). The reason for this is that the implementation of each phase takes place in separate circuits and in the case of serial execution, only one circuit would always work and the others would be idle.

Pipeline execution

To increase efficiency, pipeline implementation can be used as an option for ILP (instruction level parallelism). In this way, several elementary instructions (machine code, not used for high-level programming languages, but assembly-level) are executed at the same time, but each in different phase. For example, while the operands of an instruction are loaded, we can decode the next instruction while starting to load a third, and so on. Thus, an instruction is completed in each step (clock cycle), although the execution of each takes more than one clock time. In some systems, the execution of an instruction can be divided into up to 20-30 phases.

In some cases, problems can occur if an instruction starts sooner than the preceding instruction is terminated. Such a situation is called a **hazard**. There are 3 types:

- Data hazard: For example, we need data that has not yet been generated (data dependency) or data wants to overwrite an existing one that we will still need (name dependency).

- Structural hazard: In this case, one part of the execution of two instructions cannot be executed in parallel for architectural reasons.
- Control hazard: In a decision situation (branching, loop organization) the value of the condition is not yet known, when the following instructions have to be processed. However, we do not know which one is necessary. (True or false branch? Loop core or post-loop part?)

Various techniques can be used to deal with such situations:

- Pipeline bubble: A "bubble" is placed on the conveyor belt. Thus, by idling, one of the instructions "waits" until the dangerous situation, i.e. the hazard, ceases. Execution slows down a bit, but not as much as if there were no pipeline at all.
- Operand/result bypassing: In some cases (e.g., data dependency) the required delay (number of bubbles) can be reduced by making the result in the EX phase available immediately in the OF phase, not only after the WB phase.
- Renaming a register: Name dependency could be avoided with an infinite number of registers, but sometimes it is enough to use the available registers properly. In a processor using such a technique, the values may not be in the register specified by the programmer/assembler but in an available register, thus reducing the number of hazards.
- **Out-of-order execution** (OoOE): Changes the order of machine code instructions in a program during a kind of runtime hardware recompilation. Using a small instruction window, the processor reads the next few instructions in advance and schedules their execution so that, as far as possible, no hazard develops, but the final result remains unchanged. Thus, the implementation may remain approximately bubble-free, but requires much more complex processor circuits.
- **Branch prediction**: Used to avoid control hazards. The processor tries to estimate which branch you will need to execute and places its instructions on the "conveyor belt". If the estimate was correct, no bubble was needed, in case of an incorrect estimate, the results are discarded, and execution of the appropriate branch begins. A good estimation mechanism is very important, especially for deep pipelines.

Some processors have sufficient resources to be able to handle not just one but multiple instructions in each pipeline phase in a single clock, thus multiplying throughput. These are **superscalar** processors. In this case, the chances of hazards are particularly high due to completely parallel execution. In addition to ILP, such systems mostly use OoOE, so that instructions in the same phase can be selected to be as independent of each other as possible. They are capable of running normal sequential programs. They may also be able to implement greedy execution. This is another option for speculative execution (in addition to branch estimation). In this case of a control hazard, both execution branches are added to the pipeline, but the result of one is discarded, which determines which branch is to be executed.

Very long instruction word (VLIW) processor

One way to implement ILP and OoOE is to compile a long, complex instruction (called a bundle) from the elementary instructions during the compilation of the source code. By properly

rearranging the instructions, the bundles contain instructions that, when executed in complete parallel, do not occur in a hazard. The translation will of course be slower (and a special compiler is required), but the implementation is very efficient. Hardware can also be much simpler than a superscalar processor because there is no need for runtime recompilation to avoid hazards. A further development of this is EPIC (Explicitly Parallel Instruction Computing).

Vector processor

This is a SIMD system implementing data concurrency. It contains large registers that can contain more than one piece of data at a time and can execute a given instruction on all stored values at once. (For example, in a 128-bit register we can store 2 double or 4 float or 4 int or 8 short or 16 char values.) Special instructions must be used to use special registers.

In the history, various realizations have appeared:

- MultiMedia eXtension (MMX)
- 3DNow!
- Streaming SIMD Extension (SSE, SSE2, SSE3, SSE4)
- Advanced Vector eXtension (AVX, AVX2, AVX-512)

In these processors, the size of the registers increased steadily, and more and more instructions appeared, but in the meantime they also sought backward compatibility. For this reason, for example, the lower half of the 64-byte registers of the AVX-512 (`ZMM0-ZMM31`) can be used as a 32-byte AVX register (`YMM0-YMM15`), the lower 128-bit portion of which can be used as an SSE register (`XMM0-XMM15`) and its lower half as MMX register (`MM0-MM7`).

This solution can be very efficient when processing arrays (vectors). Graphics processing units (GPUs) also use such logic in their operation.

Further technologies

Additional technologies are available in modern processors to increase performance:

- Dynamic clock management (Intel Turbo Boost, AMD Turbo Core)
- Multilevel (integrated) cache
- Hyper-threading
- Multicore and manycore processors
- Supporting virtualization (VT-x, AMD-V)
- General-purpose GPU, integrated FPGA

Comment:

The ability to use the programming tools associated with the topic may be required. Wider familiarity with the subject is not a disadvantage. This is just a short summary.

Related subjects:

- *Computer architectures*

Further suggested readings:

- John Paul Shen, Mikko H. Lipasti: *Modern processor design* (Waveland Press, 2013) chapter 2, 4 and 5.
- Sarah L. Harris, David Money Harris: *Digital Design and Computer Architecture* (Morgan Kaufmann, 2016)
  chapter 7.
- Christopher J. *Hughes: Single-Instruction Multiple-Data Execution (Synthesis Lectures on Computer Architecture)* (Morgan & Claypoolm 2015)
  chapter 2 and 3.