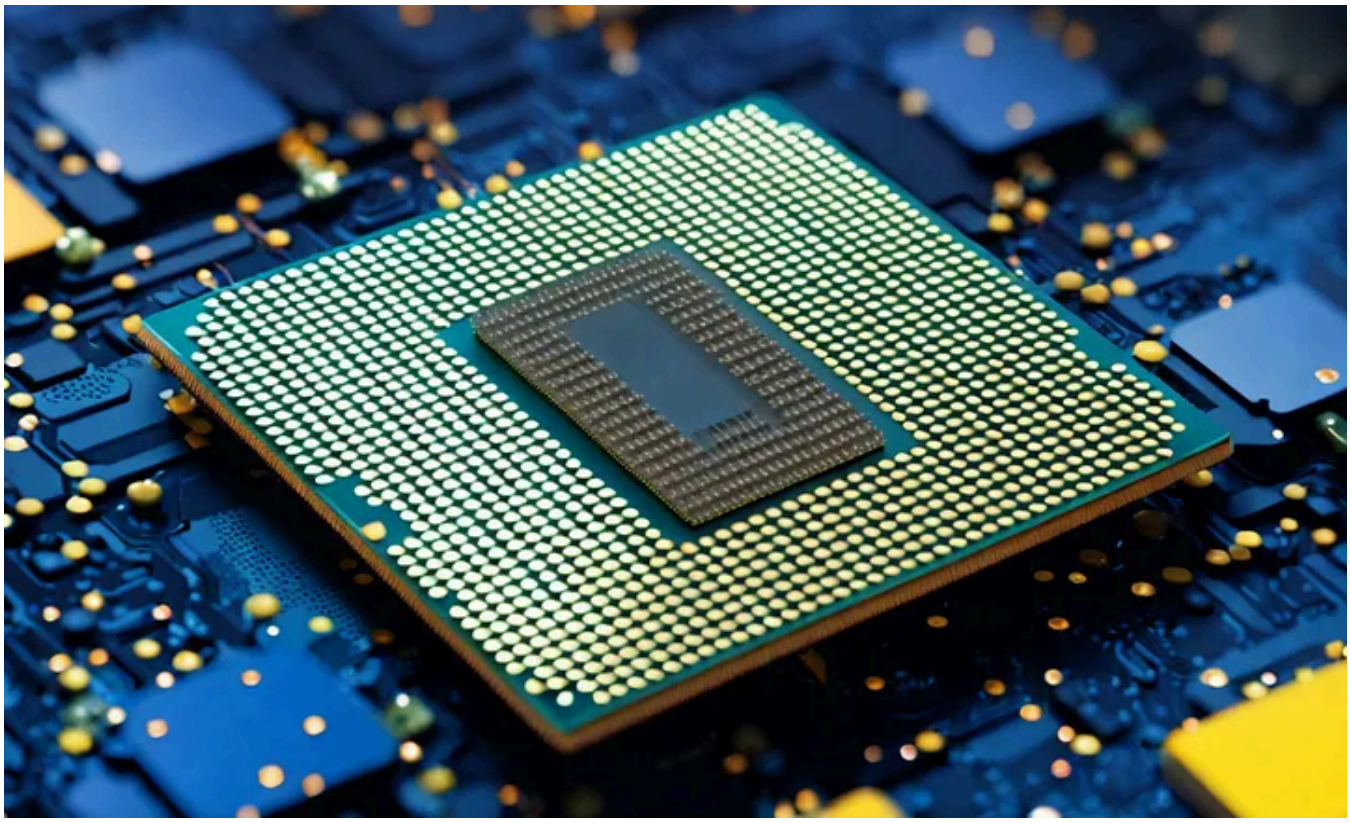
Anatomy of a CPU

The Core of Digital Evolution: Inside a Processor's Design

Razvan Badescu · Follow

19 min read · Oct 12, 2023

Listen Share More



CPU Visualization

As the brainpower behind our devices, the CPU has undergone revolutionary changes over the years, driven by the relentless miniaturization of transistors and innovations in design. Let's peel back the layers and explore the insides of a CPU.

CPU Performance and Transistors

At the core of CPU performance lies the transistor, a tiny switch that can either allow a signal to pass (on state) or not (off state). In the context of digital circuits, this behavior represents the binary states: 1 (on) and 0 (off).

All components of a CPU, including all its processing units, cache memory and everything else, are built using transistors.

From Transistors to Logic

Beyond acting as mere switches, transistors combine to form logical circuits:

- **Logic Gates**: At the basic level, transistors are configured to create logic gates (AND, OR, NOT, NAND, NOR, XOR, etc.). These gates perform logical operations based on their inputs.
- **Combinational Circuits**: Logic gates are combined to form more complex structures called combinational circuits. These circuits produce a specific output for a given set of inputs. Examples include adders (used in ALUs), multiplexers, and decoders.
- **Sequential Circuits**: When you add elements that have memory, like flip-flops, to combinational circuits, you get sequential circuits. These circuits consider both current and past inputs to produce their output. Registers, counters, and memory elements are examples of sequential circuits.
- **Functional Units**: Units like ALUs, FPUs, and AGUs are essentially combinations of many combinational and sequential circuits to perform arithmetic, floating-point, and addressing operations, respectively.
- **Control Structures**: The control unit, which fetches and decodes instructions and sends control signals to other parts of the CPU, is also built using a combination of logic gates and other circuit elements.
- **Memory Elements**: Cache memory, registers, and other storage elements in the CPU are built using flip-flops and other transistor-based structures.
- **Interconnects**: Transistors are also used in driving and receiving signals over the various buses and interconnects within the CPU and between the CPU and other components.

Facing Performance Challenges

The transistors amount that can be compacted in a CPU influences the CPU performance.

In 1975, Gordon Moore (co-founder of Intel) has observed that the number of transistors was doubling approximatively once every two years. This observation is known as Moore's Law.

Rising transistor counts have made it harder to keep up with Moore's Law. Physical, economic, and technological barriers are some of the main challenges. Issues such as power consumption and heat dissipation further complicate matters. This complexity is also captured in the economic version of Moore's Law, which posits that the cost of chip fabrication doubles every four years. To navigate these challenges, the semiconductor industry has pivoted to architectural advancements, software enhancements, specialized co-processors (like GPUs and TPUs), and breakthroughs in quantum and neuromorphic computing (for a comprehensive overview of these cutting-edge technologies, check out our article on 'Beyond Classical Computing: Exploring Quantum and Neuromorphic Frontiers')

CPU's Components

A modern CPU isn't just a monolithic block. Since it has several cores, it makes sense to distinguish which CPU components are located in the core and which outside of the core.

Among the CPU's components you'll find:

- Multiple cores.
- **Integrated Graphics Processing Units (iGPU):** While many CPUs, especially in consumer laptops and desktop, have integrated graphics processors, the high-performance computing or server CPUs might not always integrate graphics to maximize other performance aspects.
- **System Management Mode (SMM) Components:** For low-level operations, like hardware monitoring and power management.
- **Northbridge/Southbridge:** In older architectures, these were distinct chips responsible for fast and slow I/O operations, respectively, Many of these functions are now integrated directly into the CPU, especially with Systems on a Chip (SoCs)

- **Physical Interfaces:** Such as pins or contact points for interfacing with the motherboard on the chipset. These interfaces can change based on CPU socket design, and newer designs prioritize more efficient power delivery and faster data communication.

- **Memory Controller Unit (MCU):**
  – It manages interactions with the main system memory (RAM). When data is not available in any of the CPU's cache levels, due to a cache miss, the MCU retrieves it from RAM. Similarly, when data needs to be written back to RAM (either due to a cache eviction policy or because the data is being flushed from the cache), the MCU handles this operation.
  – By controlling the flow of data to and from RAM it ensures that data is retrieved and stored in an orderly and efficient manner. This includes handling read and write operations, managing memory refresh cycles, and dealing with potential conflicts or access timings.
  – Older systems had MCUs as a separate component (usually on the northbridge of the motherboard). Modern CPUs. Especially in multi-core architectures, have integrated the MCU into the CPU chip. This allows faster access to RAM.

- **I/O Controller:** Manages input/output operations, interfacing with peripherals.

- **Interconnects/Busses:** They are communication pathways to transmit data between various CPU components, RAM, and other parts of the computer. In multi-core CPU designs, sophisticated interconnects like Intel's QuickPath Interconnect (QPI) or AMD's Infinity Fabric were developed to facilitate faster and more efficient communication between cores, memory and other components.

- **L2 Cache:**
  – Some CPU architectures have a dedicated L2 cache for each core, while other share the L2 cache between a few cores.
  – Bigger and slower than L1 but smaller and faster than L3.

- **L3 Cache (and beyond):** Bigger and slower than L2, often shared across all cores.

Core Components

In a core, you can find components such as:

- **Control Unit (CU):** Orchestrates the CPU's operations by fetching instructions from the memory (RAM), decoding them, and then managing execution of instructions by generating necessary control signals for other CPU components to execute the instructions.

- **Fetch and Decode Units (FDU):**
  – Work in tandem with the CU to retrieve and interpret instructions.
  – Instruction Fetch Unit or IFU, a component of the larger Fetch-Decode Unit or FDU) is a regular, sequential operation that fetches instructions one after the other, unless there's a jump, branch, or other non-sequential instruction.
  Observation: *Not AGU computes the memory address of the instructions to be fetched from memory. For regular sequential instruction fetches, a simple program counter (PC) register increment can provide the address of the next instruction. The PC is incremented by the instruction size (like 4 bytes for 32-bit instructions) each cycle. However, for jumps, branches, and the like, more sophisticated logic is needed to determine the next instruction address, but this logic is often part of the instruction fetch or branch prediction mechanisms, not the AGU.*

- **AGUs (Adress Generation Unit):**
  – A dedicated PU that calculates the addresses used by the CPU to access the main memory (RAM). The computation involves integer arithmetic operations such as: addition, subtraction, modulo, bit shifts. etc.
  – Modern CPUs integrate multiple AGUs in each core to compute the memory addresses in parallel, and often the result comes out in a single CPU cycle. E.g. Intel's Sandy Bridge or Haswell CPU architectures incorporates multiple AGUs.
  – As all dedicated PUs, it operates independent of ALU.
  Observation: *AGU doesn't compute addresses for instruction fetches (instructions also resides in RAM), but for the data fetches and stores.*

- **L1 cache:** The smallest and fastest memory, the L1 cache is only one per core.

- **Translation Lookaside Buffer (TLB):** A cache specifically for virtual address to physical address translations.

- **Load-Store Unit (LSU):**
  – The LSU loads data from cache into registers and stores the data from registers back to cache. It deals the immediate needs of the CPU's execution units to load and store data: It will first look in the cache. If the data is present in the cache (a cache hit), it's used directly. If not (a cache miss), the request gets passed to a

higher level of memory hierarchy (like MCU).
– Since each core contains the executing units that requires loads and stores, each core has its own LSU.
Observation: *In order not to confuse LSU with MCU, I will briefly outline the main responsibilities of the two entities that manage r/w operations in memory spaces. While the LSU handles r/w operations between the CPU's cache levels and registers, the MCU handles r/w operations to and from the RAM. For instance, when an execution unit needs data, the LSU retrieves it from cache. When the data isn't available in the cache and needs to be fetched from the main memory, or when data needs to be written back to memory, the MCU handles this.*

- **ALU(s) (Arithmetic-Logic Unit):** Are used to perform logic operations and typically uses the general-purpose registers (GPRs). Modern CPUs, especially those designed for high-performance or parallel processing tasks, often incorporate multiple ALU within a single core to allow greater instruction-level parallelism.

- **FPU(s) (Floating-Point Unit):** It handles operations on floating-point numbers like, addition, subtraction, multiplication, division, square roots and trigonometric functions more efficiently than if these were emulated in software using the ALU. It works with the floating-point specialized registers to read operands from and write results to. Modern high-performance CPUs can have more than one FPU per core.

- **SIMD Unit(s):** While traditional functional processing units (FPUs) work perform scalar operations (ALU on integers and FPU on floating-point numbers), the SIMD units are more powerful, performing *vector operations* on both integer and floating-point numbers enabling parallel computation on multiple data at once (in the same CPU cycle).
They operate on wide specialized registers that can hold multiple data elements (stores vectors of data). Modern CPUs can have more SIMD units per core.

- **Instruction Pipeline:** In general, there's only one per core, but it's segmented into multiple stages (fetch, decode, execute, etc.). These stages work in tandem, allowing the core to work on multiple instructions simultaneously at different stages.

- **Branch Prediction Unit:** Its role is vital in ensuring that the instruction pipeline remains filled and that the CPU can speculate ahead, guessing the outcome of

branches (like if-else branches). Typically, there's one BPU per core.

- **Multiple registers:** Used by the PUs (Processing Units) to temporary store the operands.

Registers

Registers are the fastest storage space used by the CPU keep data it needs to access quickly (in a single CPU cycle).

Quantity in CPU:
The number of registers is determined by the CPU architecture and design, but do not scale directly with the number of cores or threads, even though each core has its own set of registers. E.g. Intel CPUs have multiple general-purpose registers (GPR):

- 8 in the 32-bit version
- 16 in the 64-bit version

Classification:
The registers can be classified as general purpose and specialized registers, and we can define some of them bellow:

- **General-Purpose Registers (GPR):** used for arithmetic, logic, and data transfer operations;
- **Program Counter (PC) or Instruction Pointer (IP):** Holds the address of the next instruction to be executed.
- **Stack Pointer (SP):** Points to the top of the current stack in memory.
- **Base Pointer (BP) or Frame Pointer (FP):** Often used to point to a fixed location within a stack frame, aiding in referencing local variables and function call return addresses.
- **Status or Flag Registers:** Contain flags that provide information about the outcome of the most recent arithmetic or logic operation (e.g. zero flag, carry flag, overflow flag).
- **Control Registers**: Controls various CPU operations. E.g. in x86 CPUs:
  – **CR0**: Controls system mode and state, including protected mode, paging, and coprocessor emulation.

– **CR2**: Stores the address that caused a page fault.
– **CR3**: Page directory base register, critical for virtual memory and paging.

- **Statement Registers**: In architectures that use memory segmentation, these registers store pointers to memory segments.
- **Floating-Point Registers**: Store floating-point values used in fp. operations.
- **Special Purpose Array (SPA)**: Found in some DSPs (Digital Signal Processors), these registers hold multiple values for parallel operations.
- **Loop Counters**: used to handle loop iterations efficiently.
- **Predicate Registers**: Some architectures, like Itanium, predicate registers store Boolean values that determine if the associated instruction will be executed.
- **Registers used for SIMD (Single Instruction, Multiple Data) operations**:
  – **MMX**: Used for the older MMX instruction set, operating on integer values.
  – **XMM**: Used for SSE and SSE2 instruction set.
  – **YMM**: Introduced with Advanced Vector Extension (AVX) instruction set, they're wider than XMM registers.
  – **ZMM**: Introduced with AVX-512 instruction set, they're even wider than YMM and can hold up to 512 bits of data.
  – **Mask Registers**: Introduced with AVX-512, the mask registers (k0 through k7) are 64-bit registers used to conditionally mask off or include certain vector lanes during SIMD operations.

Register Files:
In modern CPUs, especially those with multiple execution units, registers are grouped in what's called a "register file". This is a small, highly optimized block of memory within the CPU core where all the GPR and specialized registers are stored. The design ensures rapid parallel access to multiple registers at once.

Location in the Core:
To minimize the latency, they're located in the core, typically close to the ALU and other execution units to ensure data can be quickly moved from the units that process it.

Physical Design:
A register is made up of semiconductor devices specifically using transistors.

Modern CPUs use CMOS (Complementary Metal-Oxide-Semiconductor) technology, which is a combination of nMOS and pMOS transistors.

Logical design:

- The building block of a register (and memory cells in general) is called a flip-flop. There are several types of flip-flops, like the D-type (Data Type) flip-flop, which is commonly used in registers.
- A flip-flop can store a bit of data. To store, say, a 32-bit word, you'd essentially have an array of 32 flip-flops grouped together and managed such that they can be read from or written to simultaneously.

Quantity Transistors in a Data Flip-Flop:
A standard D-type flip-flop can be logically projected using 4 NAND or 4 NOR gates. Since a standard NOR or NAND gate requires 4 transistors (in CMOS technology: 2 nMOS and 2 pMOS), a standard data flip-flop contains 16 transistors.

Observation: *The number of transistors can vary based on the specific of the design or if additional inputs are added, but for the standard 2-input NOR/NAND gate, 4 transistors are required.*

Standard Registers vs SIMD Registers:
While a standard 64-bit register contains 64 flip-flops (each flip-flop can store 1 bit of data) and requires 64 x 16 = 1024 transistors, a SIMD register requires 16 x 256 = 4096 transistors.

CPU Components Interaction

To better understand how the CPU components interacts, let's follow two simple scenarios step by step.

Addition of two integers A and B

1. **Fetch Instruction:** CU fetches the machine code instruction ADD A, B from memory.
2. **Decode Instruction:** CU decodes the previously fetched instruction to determine which operation to perform and on which operands.
3. **Compute Memory Address:** If A and B are stored in memory (and not already in registers), the CU instructs the AGU to compute the memory address for A and B. The AGU computes the memory address based on the instruction given (this

could involve adding offsets, or using base addresses, etc.).

Note:

*– **Base address**: It's a fixed memory location. For instance, an array might start at a particular memory location, this starting address is the base address of the array. Since A and B are scalars, the base address would be the start of the integer variable in memory.*

*– **Offset**: It is the distance (usually in terms of memory locations) from the base address. As an example, a 4-lanes vector has an offset of 3, which is the index (position) of the vector's last element. For scalar values like A and B there will be no offsets because they aren't part of a longer structure like an array. But remember, operations might occur on different sizes of data (byte, word, double-word, etc.) and the CPU needs to know where each byte of the data resides.*

– In memory access, particularly with arrays or structures, the address of the data is often computed (by the AGU) as the sum of the base address and the offset.

4. **Fetching Data:** Once the AGU provides the memory addresses for "A" and "B", CU instructs the memory controller unit (MCU) interfacing with the CPU to fetch the values of the operands.

5. **Load into Registers:** The CU orchestrates the loading of the fetched data (values of A and B) into the GPRs.

6. **Execution:** With A and B loaded into GPRs, the CU sends a control signal to the ALU to execute the "add" operation.
   ALU performs the addition and places the result in another GPR (or sometimes, depending on the CPU architecture, one of the operand registers can be overwritten with the result).
   Observations:
   – Do not confuse the control instructions sent by the CU to other CPU components with the assembly instructions (which is specific to each type of processor) While the assembly instruction looks like LD R1, A (instructs to load data from the memory address of the operand "A" into the register "R1") the control signal from the CU is more specific.
   – Modern CPUs have complex control logic where different operations correspond to different patterns of bits in the control signal. It is not as simple as 1 means "add" and 0 means "multiply".

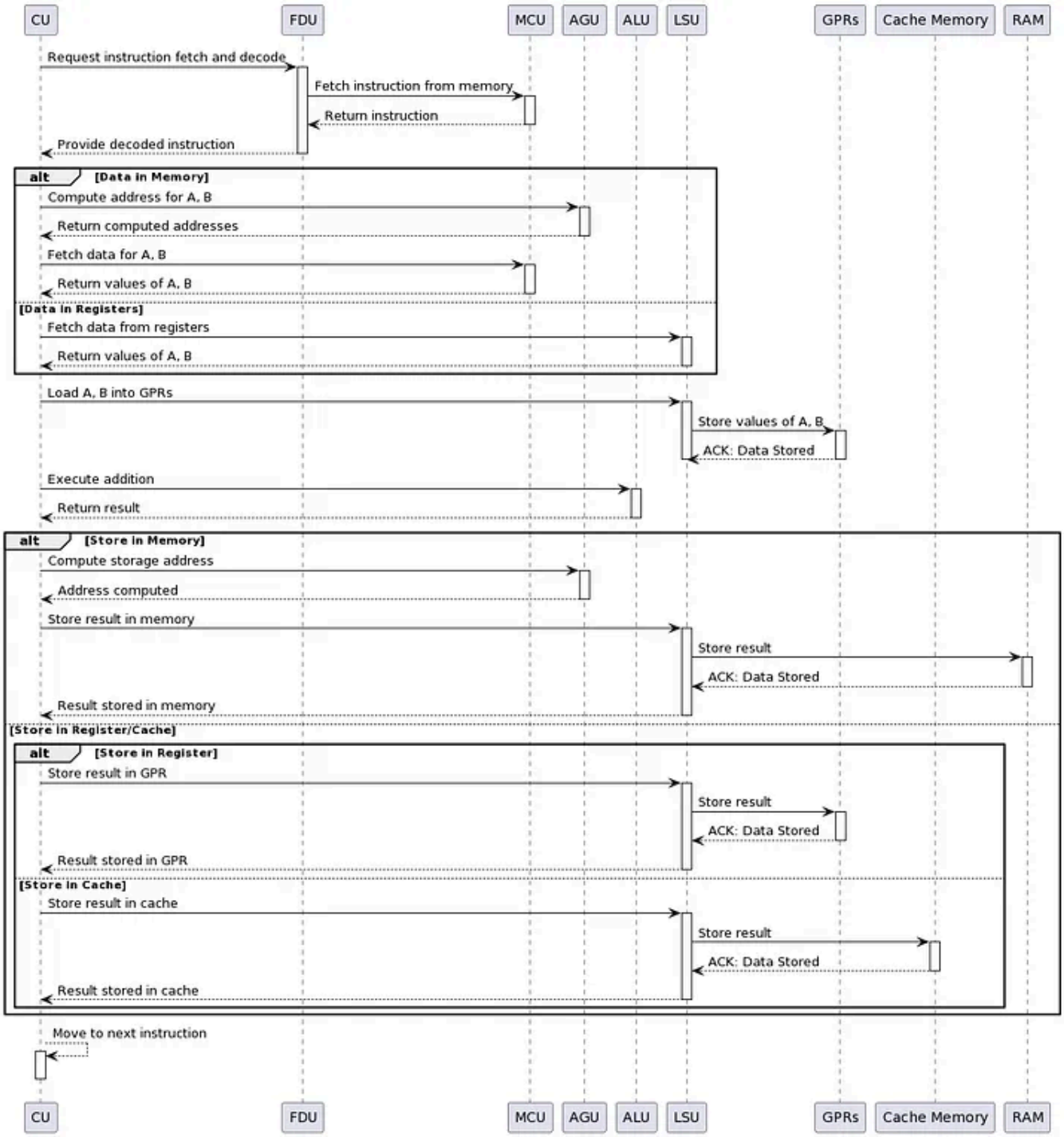
7. **Store Result:** If the instruction specifies to store the result in memory, the CU will again instruct the AGU to compute the address where the result should be

stored. Once the storage address is determined, the result is written back to memory from the GPR. This operation is often called a “store” operation in machine instructions.

8. **Completion:** The CU marks the instruction as complete and moves to the next instruction in the sequence.

Observation: *This is a high-level sequence, and real-works scenarios involve a lot of optimizations and additional considerations (like branch prediction, cache hits/misses, pipelining, etc.) that can influence this basic sequence. But for the purpose of understanding, this sequence should provide a clear picture of how data moves through the CPU during a simple arithmetic operation.*

To gain a clearer, step-by-step understanding of how the components interact, refer to the sequence diagram below:



Breaking Down the Addition of Integers A and B

Add A and B and Save Result in Registry

Imagine the assembly instruction is ADD R1, R2, R3, which means to add the values in registers “R2” and “R3” and store the result in “R1”. Let’s see how the CU orchestrates the processing of this assembly instruction:

1. **Instruction Fetch:** The CU fetches this instruction from memory (with the help of memory controllers and AGUs for addressing).
2. **Instruction Decode:** The CU decodes ADD R1, R2, R3. It recognizes the ADD operation and the involved registers.

3. **Emit Control Signals:**
   – It sends a control signal to the Register File to fetch the contents of “R2” and “R3”.
   – It sends another control signal to the ALU to prepare it for an addition operation.

4. **Execution**: Once “R2” and “R3” values are fetched, they’re sent to the ALU. The ALU, upon receiving the earlier control signal, knows it has to perform addition. It adds the numbers.

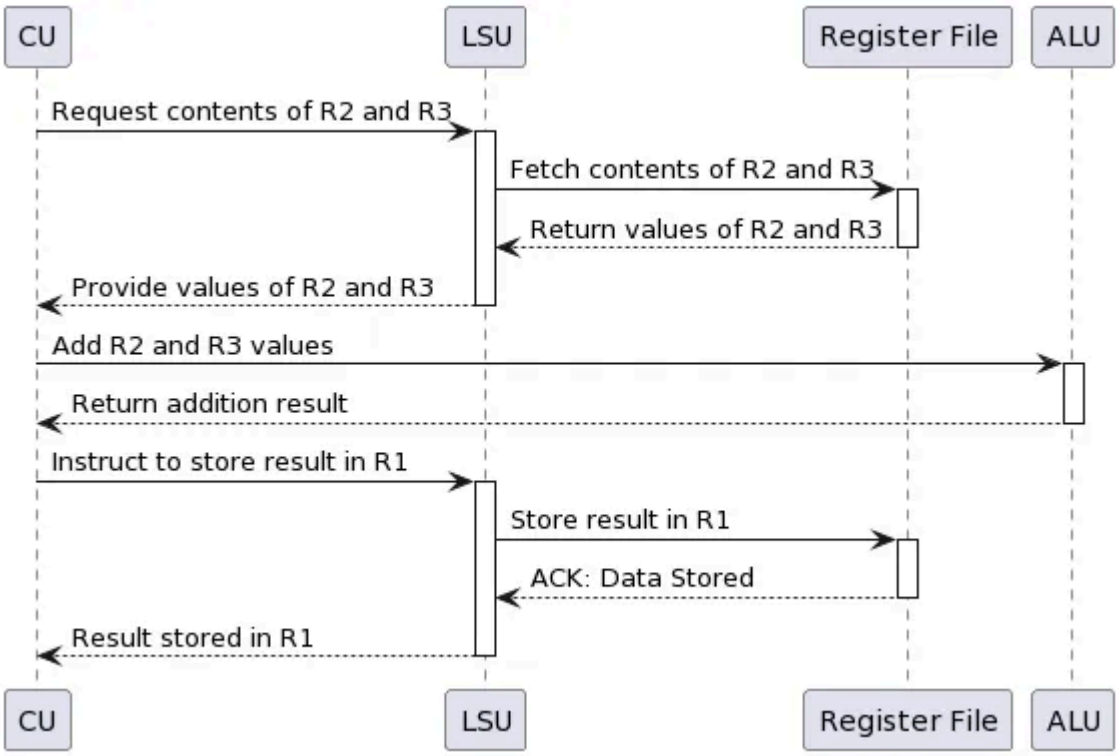
5. **Store Result**: The result from the ALU is then sent back to the Register File, and the CU sends another control signal to store this result in R1.

If we had to conceptualize control signals for the above scenario we would have:

- “Fetch contents of R2 and R3.”
- “Prepare for addition.”
- “Store result in R1.”

Note: *These aren’t “instructions” in the traditional sense. They’re a simplified representation of the electronic control signals the CU sends to coordinate CPU operations. In reality, these signals are a combination of electrical voltages on various lines that each component is designed to understand and act upon.*

The following sequence diagram might help to better understand the scenario:



Add Values of Registers R1 and R2, Store Result in Register R3

Note: *The operation of a CPU is a complex dance of many components working together, driven by both the architecture's design and the specific sequence of instructions in the program being run. The actual specifics can vary widely between architectures and even between different models in the same architecture family.*

You might wonder how does your code get understood and executed by the CPU. "Program Compilation: From Source To Machine Code" sheds light on the fascinating journey of transforming your code into machine-executable instructions.

CPU Architectures

Modern CPUs employ a concept called superscalar architecture. This means they can execute more than one instruction per clock cycle by dispatching multiple instructions to appropriate functional units in the CPU. Those functional units can be ALUs, AGUs, FPUs, SIMD units, etc.

The CPU architecture refer to the design principles and technologies underpinning a processor's operation.

1. **CISC (Complex Instruction Set Computing):** These architectures have a wide variety of instructions, some of which might perform complex operations. This

aims to minimize the number of instructions per program, sacrificing clock speed. Examples: Intel's x86, Motorola's 68k

2. **RISC (Reduced Instruction Set Computing):** Focuses on highly optimized set of frequently used instructions for faster performance, aiming for more instructions per program but faster clock speed per instruction. Examples: ARM, MIPS, PowerPC

3. **EPIC (Explicitly Parallel Instruction Computing):** Developed by Intel and Hewlett Packard, this architecture focuses on parallel processing, with the compiler playing a central role in enhancing instruction-level parallelism. Example: Itanium

4. **VLIW (Very Long Instruction Word):** Similar to EPIC, VLIW relies heavily on compilers to optimize parallel instruction execution. Examples: Some DSPs (Digital Signal Processors)

5. **MISC (Minimal Instruction Set Computing)**: As the name implies, MISC architectures have an extremely limited set of instructions. Example: Stack machines like the Burroughs Large Systems.

6. **SISD, SIMD, MISD, MIMD (Flynn's Taxonomy):** This classification is based on data and instruction stream parallelism:
   – SISD: Single Instruction stream, Single Data stream (traditional sequential computers)
   – SIMD: Single Instruction stream, Multiple Data streams (vector processors, graphics units)
   – MISD: Multiple Instruction streams, Single Data stream (rarely used)
   – MIMD: Multiple Instruction streams, Multiple Data streams (parallel processors, multi-core)

7. **Harvard vs. Princeton (Von Neumann) Architectures**
   – Harvard: Separate memory storage for data and instructions, allowing the CPU to fetch both at once. Common in modern microcontrollers.
   – **Princeton/Von Neumann**: Single memory storage for both data and instructions.

8. **Quantum Computing**: A nascent field, quantum computers use quantum bits or qubits. They can represent multiple states simultaneously, offering potential computational speed-ups.

9. **Neuromorphic Computing:** Designs inspired by the structure and operation of the human brain, aiming for high efficiency in AI-related tasks.

10. **Asynchronous (Clockless) Architectures:** Most CPUs use a clock to synchronize operations. Asynchronous architectures don't; operations commence as soon as the inputs are ready.

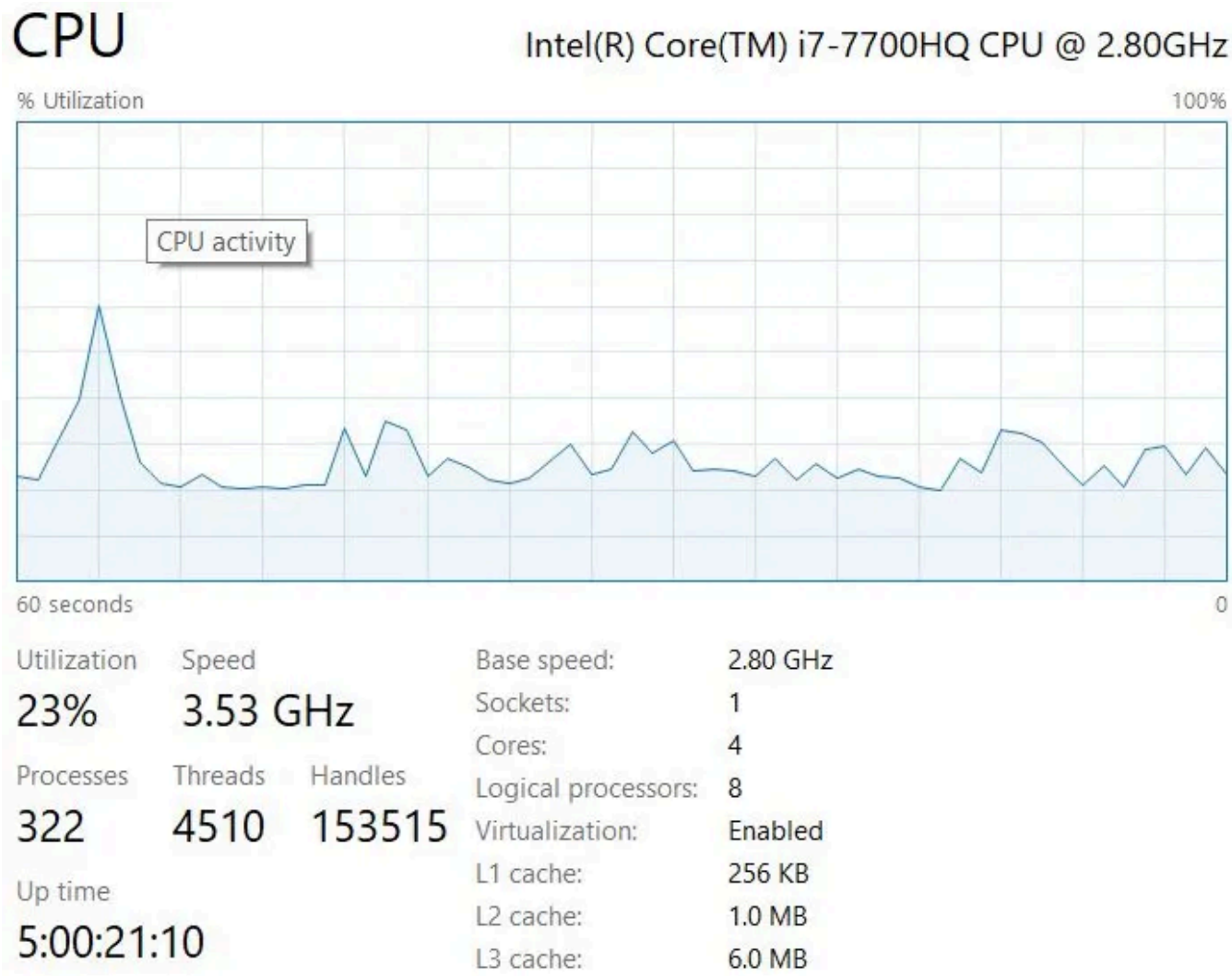
This is a high-level overview. Each category contains a vast depth of details and variations, and ongoing research ensures the landscape of CPU architectures continues to evolve.

Logical Processors

A logical processor is a hardware component that represent the interface provided by the physical CPU core for the OS and software to run threads. It can run one hardware thread.

Modern CPUs spend a lot of time waiting, especially RAM access to complete. A CPU core with two logical processors can switch to another thread and execute its instruction while other thread is waiting for the RAM.
The technique of splitting a single physical CPU core into multiple logical processors is known as Simultaneous Multi-Threading (SMT). While the execution resources (ALUs, FPUs, etc.) aren't duplicated, certain pards of the CPU's architecture (like registers) are. Essentially, SMT increases the utilization of a CPU core's resources by allowing it to work on multiple tasks (threads) almost simultaneously.

Bellow you see the intel i7 having 4 cores, each being split into 2 logical processors:



CPU Activity — A Windows Chart

Note: *Hyper-Threading (HT) is Intel's proprietary implementation of SMT. The term "Hyper-Threading" is specific to Intel processors. When you see a modern Intel CPU that supports two threads per core, it's using Hyper-Threading.*

Benefits:

- **Increased Throughput:** A core with SMTP can deliver more completed tasks in a given time than a non-SMT core. This isn't doubling the performance, but can be a significant boost in multi-threaded tasks.
  A common rule of thumb is that SMT can give a performance boost of about 30%, but this varies widely depending on the workload.
- **Efficient Resource Utilization:** Without SMT, CPU cores often remain underutilized. SMT ensures that more parts of the CPU are working at a given time.

- **Improved Responsiveness:** For multitasking environments, having SMT can make systems more responsive, as one logical core remain dedicated to a user's immediate task, while background tasks use the other

Hardware Threads vs Software Threads

Software Thread:

- The Smallest sequence of programmed instructions that can be managed independently by an operating system scheduler
- It is a subset of a process, and multiple threads within a process share the same resources like memory address space but have their own registers, program counters and stack.
- They allow for concurrent operations within a single process, which can lead to better utilization of modern multicore CPUs.

Hardware Thread:

- A hardware thread often refers to the capabilities of a logical processor. The number of hardware-threads a CPU core can handle simultaneously is given by the number of logical processors per core
- A hardware thread (corresponding to a logical processor) can process instructions of only one software thread at a time. A SMT core with two logical processors can run two threads simultaneously.
  Observation: *Saying that each logical processor (or hardware thread) can handle one software thread at a time doesn't mean it's limited to just that software thread forever. The operating system scheduler can swap out software threads based on priority, resource needs, etc. But at any given cycle, each logical processor is processing instructions from one software thread.*

Conclusion

In our journey into the anatomy of a CPU, we've unpacked the intricate orchestration of transistors, circuits, and various functional units that come together to form the brain of our computational devices. These complex structures, continually evolving, underscore the remarkable advancements in technology and engineering. With CPUs set to play an even more central role in future technologies, understanding their intricate workings becomes not just an academic exercise but a window into the future of computing itself.

If you're curious about what the future holds beyond the capabilities of current CPUs, don't miss 'Beyond Classical Computing: Exploring Quantum and Neuromorphic Frontiers'. It's a glimpse into the next frontier of computational science

Acronyms

- ACU = Address Computation Unit
- AGU = Address Generation Unit
- ALU = Arithmetic Logic Unit
- ARM = Advanced RISC Machine
- CISC = Complex Instruction Set Computing
- CMOS = Complementary Metal Oxide Conductor
- CPU = Central Processing Unit
- FPU = Floating Point Unit
- EPIC = Explicitly Parallel Instruction Computing
- GPU = Graphical Processing Unit
- MIMD = Multiple Instruction streams, Multiple Data streams
- MIPS = Microprocessor without Interlocked Pipeline Stages
- MISD = Multiple Instruction streams, Single Data stream
- MOS = Metal Oxide Conductor
- nMOS = Negatively Doped Metal Oxide Conductor
- pMOS = Positively Doped Metal Oxide Conductor
- RISC = Reduced Instruction Set Computing
- SIMD = Single Instruction, Multiple Data
- SISD = Single Instruction stream, Single Data stream
- TPU = Tensor Processing Unit

- VILW = Very Long Instruction Word

References

- CPU
- List of CPUs
- Control Unit
- Arithmetic Logic Unit
- Floating-Point Unit
- Address Generation Unit
- Load-Store Unit
- Memory Controller
- Instruction Set Architecture
- Comparing Instruction Set Architectures
- Comparing CPU Architectures
- Nehalem Microarchitecture
- Flip-Flop
- SIMD
- NMOS & PMOS Transistors

👏If you enjoyed reading this article, don't forget to give it a round of applause to show your appreciation.

🔔 Follow me to stay updated with more insightful and practical articles.

Cpu | Computer Science | Processors | Computer Engineering | Microprocessor