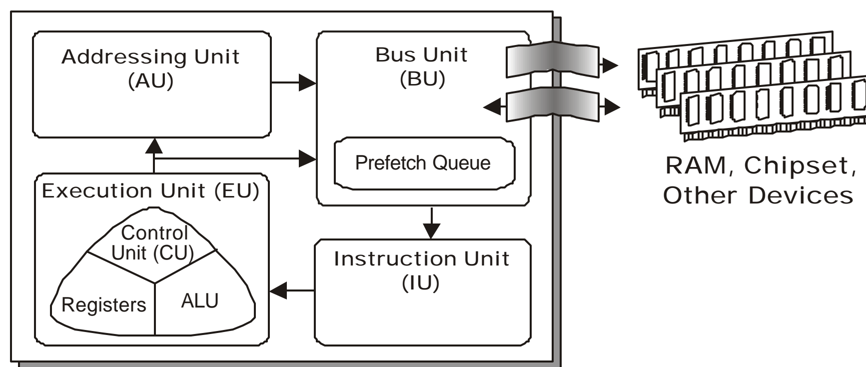
***(pages 184-200** (4th), **115-131** (3rd))*

The phrase "Central Processing Unit" clearly describes the responsibility of the CPU for processing all data within the computer system. In reality, not all data is processed or passes through the CPU, but it is the driving force behind all activity observed by the user. The CPU may not actually write to the hard disk, or produce the image on the display screen, but it is the component that processed and provided the data that eventually was given to those devices.

The CPU is the *core* of the entire system, so fundamental that its qualities are one of the key measurements for judging the performance potential of the entire system—*the system is only as good as the CPU controlling it* (if you ignore disk speed, video throughput, bus transfer time, and memory access speed).

As with all other aspects associated with computers, technology has progressed **very** quickly since the first *microprocessors* design by Intel in the late 1970's (see "History of Intel Processors" at the end).

Sections of CPU



[Generic] Central Processing Unit

The diagram above describes the CPU as being composed of separate sections, or components. Each component serves a special purpose, with no one component being more important than another. In a carefully choreographed sequence, binary words flow in and out of the CPU, changing paths depending on whether the words are <u>instruction</u> or <u>data</u> (for processing).

The CPU's *deepest* components (having existed from the first CPU designs),

- <u>Control Unit</u> (CU)
  - co-ordinates the flow of data through the ALU and Registers
- <u>Arithmetic-Logic Unit</u> (ALU)
  - performs arithmetic (addition, subtraction) and logic (greater than, less than, equal to, left/right-shift) operations on data stored in CPU registers
- <u>Registers</u>
  - local memory used by the ALU and CU
  - for CU as *input* and *output data* buffers, and for ALU as *operand* and *result* storage
  - registers are sometimes incorrectly classified as part of the <u>L1 Cache</u> in the CPU; registers limited, special "calculation processing" memory, whereas Cache is used a buffer for the <u>entire</u> CPU

The *supporting* components (servicing the *functionality* of the CPU towards the main bus and rest of the system),

- <u>Bus Unit</u> (BU)
  - controls the interaction of the CPU with the internal system bus (ISA, PCI) and memory bus
  - the *Prefetch Queue* stores the next instruction that is planned for execution (using the idea of *Cache Memory*, this queue may store more than one instruction)

- Execution Unit (EU)
  - responsible for data processing (adding, comparing, etc.) through the *Control Unit (CU)*, *Registers*, and *Arithmetic-Logic Unit (ALU);* some instructions may be given to the *Floating-Point Unit (FPU)*
  - [in modern and capable CPUs] also determines which instructions can be executed in parallel, or "out-of-sequence"
- Instruction Unit (IU)
  - the identification of current instruction in *Prefetch Queue* (removes it from queue) and decodes it for the *Execution Unit (EU)*
  - uses the *Instruction Lookup Table* to translate (decode) the *Machine Language Instruction* (used outside the CPU) to a *Microcode Instruction* (used within the CPU) {very simple instructions are the same}
- Addressing Unit (AU)
  - controls the addressing of memory (Cache, RAM) and I/O devices for the Bus Unit (BU)
  - also includes a *Memory Management Unit (MMU)* to co-ordinate the proper address of *paged memory* (through the *Segmentation Unit (SU)* and *Page Unit* (PU))
  - the width (in bits) of the AU defines (and limits) the maximum potentially addressable memory
- Floating-Point Unit (FPU) or Numeric Processing Unit (NPU) or Math Co-processor
  - originally a separate processor that worked via control from the *Execution Unit (EU)*
  - controls any advanced floating-point operations and higher-level mathematics that are not performed by the *Arithmetic-Logic Unit (ALU)*
  - instructions for the FPU are redirected by the *Execution Unit (EU)*

The Execution Cycle

The components described above work together as an *assembly line* to input, decode, process, and output the results of the instructions a program (usually, only the BIOS and Operating System).

The CPU performs the instructions of a program in order, completing each instruction, storing the result, then repeating with the next instruction. And there is no end to the series of instructions—*the CPU always has something to do.*

This sequence is called the processing (or execution) cycle. A very simplified breakdown of the cycle,

1. Read the next instruction in the queue (current instruction), point to next instruction in the queue.
2. Decode the current instruction, determine nature of instruction (does it require data from memory?)
3. Load any required data from memory and store in Registers (this might take two or more clock cycles)
4. Execute instruction
5. Store any result in Registers and/or write to memory (if needed)
6. Go back to step 1.

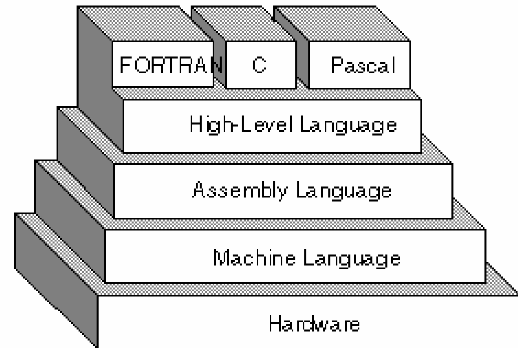
Obviously, a lot more happens within the CPU, and in the entire computer system, during a single execution cycle such as,

- interrupt request handling for hardware and software
- reading and writing to L1 (level 1-internal) and L2 (level 2-external) Cache
- pre-processing of instructions,
  - to pre-load data, determine parallel instruction execution, and determine "out-of-order" (out-of-sequence) instruction execution
- floating-point instructions
- extended instructions (MMX (multimedia extensions), 3Dnow!, SSE (streaming SIMD Extension)

Further, some instructions may warrant conversion from CISC to RISC, which makes the interpretation of the instruction longer, but the overall instruction execution much faster (see below, "RISC and CISC").

Machine Language, Assembly Language

Most programmers are familiar with *high-level languages*, since these are languages they use on a regular basis (such as, C/C++, VB, Java, COBOL, FORTRAN, and Pascal).



Programs written in these languages must be compiled or translated into *machine language* in order for the CPU to understand and perform them. But the level diagram seems to indicate a language in-between: *assembly language*.

For many computer-literate users, *machine language* and *assembly language* are one-in-the-same, using the terms interchangeably. Although both languages are *semantically* similar, the programming format and approach are different.

Machine Language

The *binary code* used to identify specific commands for the CPU based on the Instruction Set (each CPU, or compatible group of CPUs, has its own machine language). Since coding directly in binary is not possible, or desired, programmers use either hexadecimal or octal (and a translation tool) to compose programs.

The main disadvantage of machine language is that each CPU instruction and data is a binary code (as hex or octal), that must be referenced in a programming manual, or just remembered.

Assembly Language

To ease the burden on programmers having to remember (or reference) binary code, each machine language instruction is given a *mnemonic*, or "name" (in general, a *mnemonic* is anything used in place of an object, to help remember that object). This allows programs to be "assembled" from symbols, and although the number of instructions is not reduced by using Assembly Language, programs can be written using more "human-language" codes. The advantage is programs are *readable*.

The disadvantage of Assembly Language is that a compiler is required to transform the mnemonic-written code into machine code (binary) for the CPU. Below is a sample of assembly language.

```
;this is a simple program which displays "Hello World!"
;on the screen.

.model small
.stack

.data
Message db "Hello World!$"    ;message to be display

.code
mov dx,OFFSET Message         ;offset of Message is in DX
mov ax,SEG Message            ;segment of Message is in AX
mov ds,ax                     ;DS:DX points to string
mov ah,9                      ;DOS function 9 - display string
int 21h                       ;call DOS service software interrupt
mov ax,4c00h                  ;return to DOS
int 21h
END start                     ;end here
```

(*Note: The DOS program execution and debugging tool (the DEBUG utility) is capable of representing binary code machine language programs in assembly (called "unassembly"), as well as allowing machine language programming in hex. DEBUG does not include an assembly language compiler, but such compilers are available freely on the Internet for almost all platforms (CPUs and operating systems).)*

RISC and CISC

***(page 194** (4th), **125** (3rd))*

The language of a CPU is defined by its *Instruction Set (IS)*, which defines all possible instructions.

Consider the IS as a "closed vocabulary" (no new words are possible or can be added), and the *Instruction Lookup Table* (see *Instruction Unit (IU)* above) represents the CPU's vocabulary: its dictionary.

CISC (Complex Instruction Set Computer)

- the *instruction set* is very long, accommodating many similar but different instructions
- user programs (in Machine Language) are short because they are composed of comprehensive instructions, very exact instructions (rather than a large number of small, repeated instructions)
- very efficient for memory usage, since programs are short (few instructions required)
  not efficient for the CPU since it must look through a very long *lookup table* to find each instruction
- good for system designers writing programs on limited-memory computer, single-task systems (like microcontroller circuits and personal computers)

RISC (Reduced Instruction Set Computer)

- the *instruction set* is very small, with simple instructions (simpler than CISC)
- user programs are longer than CISC, since more instructions are required, but the programs are composed of short, quickly *decoded* instructions
- not efficient for memory, since programs are long, but efficient for CPU since the *lookup table* is very short, and instructions can be found quickly
- very popular now that program and data storage use multi-megabyte RAM and disks
- for CPU and OS designers, less machine language instructions implies less possible errors; which implies the possibility of producing more stable software

The RISC approach allows for the design of processors that may have a low clock cycle, but are very efficient and run very cool (less instructions→less circuitry→less heat).

SUN Microsystems has been using RISC technology in their SPARC CPUs from the beginning. And with the performance and reliability of Motorola's and IBM's RISC-based PowerPC CPUs in Apple's computers, Intel and AMD have decided to incorporate RISC designs in their CISC CPUs.

***Note**: Both Intel and AMD have stated that new CPU designs will be RISC-based, foregoing compatibility with older CISC CPUs or forcing Operating Systems companies to include CPU-emulators for legacy support—has this happened?*

Too much software (applications and operating systems) has been designed for CISC-based processors, and the shift will be extremely costly for the organisations involved to make such an investment.

One possible future has been offered by Transmeta and its Crusoe CPUs. The CPUs are RISC at the low-level, but provide a real-time CISC emulation. This means that a Crusoe CPU runs CISC instructions by translating them to RISC before execution. It may seem timely and complicated, and it is, but the processors require little energy and generate little heat—a perfect option in some cases.

History of Intel Processors

Processor	Reg. width	Addr. bus width	Data bus width	Address space	Clock speed (MHz)	Year	# of transistors
8088	16 bit	20 bit	8 bit	1 MB	4.7 to 8	1979	29,000
8086	16 bit	20 bit	16 bit	1 MB	4.7 to 10	1978	29,000
80188 *	16 bit	20 bit	8 bit	1 MB	5 to 16	1982	n/a
80186 *	16 bit	20 bit	16 bit	1 MB	5 to 16	1982	n/a
80286	16 bit	24 bit	16 bit	16 MB	12 to 25	1982	134,000
386DX, DX4	32 bit	32 bit	16 bit	4 GB	16 to 66	1985	275,000
486DX, DX4	32 bit	32 bit	32 bit	4 GB	25 to 100	1989	.8 to 1.2 million
Pentium	32 bit	32 bit	64 bit	4 GB	60 to 66	1993	3.1 million
Pentium MMX	32 bit	32 bit	64 bit	4 GB	75 to 233	1994	4.5 million
Pentium Pro	32 bit	36 bit	64 bit	64 GB	~ 150 to 200	1995	5.5 million
Pentium II	32 bit	36 bit	64 bit	64 GB	~ 233 to 400	1997	7.5 million
Pentium III	32 bit	36 bit	64 bit	64 GB	~ 500 to 1000	1999	24 million
Pentium IV	*32 bit*	*36 bit*	*64 bit*	*64 GB*	*1.5 to 2.8 GHz*	*2000*	*42 million*

** special processor used in advanced controller circuits*

Newer processors are now available from Intel and AMD that surpass the Pentium IV, with higher processing (clock) speeds and more potentially addressable,

the Intel Itanium has 64-bit register and data bus widths, and 44-bit address bus.

the AMD Athlon 64 has 64-bit register and data bus widths, and 52-bit address bus.

There are a lot of resources available on the Internet for the history and aspects of the *important* CPUs.

A good starting point is: **http://www3.sk.sympatico.ca/jbayko/cpu.html**

Along with **Intel**: **http://www.intel.com/intel/intelis/museum/exhibit/hist_micro/hof/tspecs.htm**