Major Components of Computer "Engines"

Norman Matloff
University of California at Davis
©2001-2007, N. Matloff

December 11, 2006

Contents

1 Introduction **3**

2 Major Hardware Components of the Engine **4**

2.1 System Components 4

2.2 CPU Components 6

2.2.1 Intel/Generic Components 6

2.2.2 History of Intel CPU Structure 9

2.3 The CPU Fetch/Execute Cycle 10

3 Software Components of the Computer "Engine" **10**

4 Speed of a Computer "Engine" **12**

4.1 CPU Architecture 12

4.2 Parallel Operations 12

4.3 Clock Rate 13

4.4 Memory Caches 14

4.4.1 Need for Caching 14

4.4.2 Basic Idea of a Cache 14

4.4.3 Blocks and Lines 15

4.4.4 Direct-Mapped Policy 16

4.4.5 What About Writes? 16

4.4.6 Programmability 17

4.4.7 Details on the Tag and Misc. Line Information 17

4.4.8 Why Caches Usually Work So Well 17

4.5 Disk Caches 17

4.6 Web Caches 18

1 Introduction

Recall from Chapter 0 that we discussed the metaphor of the "engine" of a computer. This engine has two main components:

- the hardware, including the central processing unit (CPU) which executes the computer's machine language, and
- the low-level software, consisting of various services that the operating system makes available to programs.

This chapter will present a broad overview of both components of this engine. The details will then unfold in the succeeding chapters.

One of the major goals of this chapter, and a frequent theme in the following chapters, is to develop an understanding of the functions of these two components. In particular, questions which will be addressed both in this chapter and in the chapters which follow concern the various functions of computer systems:

- What functions are typically implemented in hardware?
- What functions are typically implemented in software?
- For which functions are both hardware and software implementations common? Why is hardware implementation generally faster, but software implementation more flexible?

Related to these questions are the following points, concerning the **portability** of a program, i.e. the ability to move a program developed under one computing environment to another. What dependencies, if any, does the program have to that original environment? Potential dependencies are:

- Hardware dependencies:

  A program written in machine language for an older PC's Intel 80386 CPU[1] certainly won't run on Motorola's PowerPC CPU.[2] But that same program *will* run on PC using the Intel Pentium, since the Intel family is designed to be **upward compatible**, meaning that programs written for earlier members of the Intel CPU family are guaranteed to run on later members of the family (though not vice versa).

- Operating system (OS) dependencies:

  A program written to work on a PC under the Windows OS will probably not run on the same machine under the Linux OS (and vice versa), since the program will probably call OS functions.

And finally, we will discuss one of the most important questions of all: What aspects of the hardware and software components of the engine determine the overall speed at which the system will run a given program?

[1]Or clones of Intel chips, such as those by AMD. Throughout this section, our term *Intel* will include the clones.

[2]This is what the Apple Macintosh used until 2005.

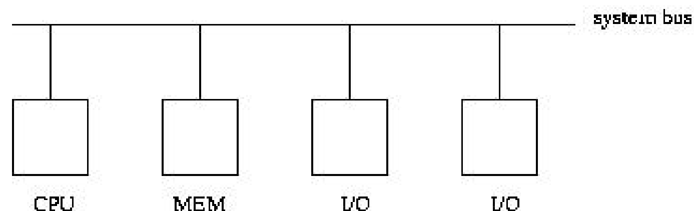

Figure 1: system structure

2 Major Hardware Components of the Engine

2.1 System Components

A block diagram of the basic setup of a typical computer system appears in Figure 1.

The major components are as follows:

CPU

As mentioned earlier, this is the **central processing unit**, often called simply the **processor**, where the actual execution of a program takes place. (Since only machine language programs can execute on a computer, the word *program* will usually mean a machine language program. Recall that we might write such a program directly, or it might be produced indirectly, as the result of compiling a source program written in a high-level language (HLL) such as C.)

Memory

A program's data and machine instructions are stored here during the time the program is executing. Memory consists of cells called **words**, each of which is identifiable by its **address**.

If the CPU fetches the contents of some word of memory, we say that the CPU **reads** that word. On the other hand, if the CPU stores a value into some word of memory, we say that it **writes** to that word. Reading is analogous to watching a video cassette tape, while writing is analogous to recording onto the tape.

Ordinary memory is called **RAM**, for Random Access Memory, a term which means that the access time is the same for each word.[3] There is also **ROM** (Read-Only Memory), which as its name implies, can be read but not written. ROM is used for programs which need to be stored permanently in main memory, staying there even after the power is turned off. For example, an autofocus camera typically has a computer in it, which runs only one program, a program to control the operation of the camera. Think of how inconvenient—to say the least—it would be if this program had to be loaded from a disk drive everytime you took a picture! It is much better to keep the program in ROM.[4]

I/O Devices

A typical computer system will have several **input/output** devices, possibly even hundreds of them (Figure 1 shows two of them). Typical examples are keyboards/monitor screens, floppy and fixed disks, CD-ROMs,

[3] In the market for personal computers, somehow the word "memory" has evolved to mean disk space, quite different from the usage here. In that realm, what we refer to here as "memory" is called "RAM." The terms used in this book are the original ones, and are standard in the computer industry, for instance among programmers and computer hardware designers. However, the reader should keep in mind that the terms are used differently in some other contexts.

[4] Do not confuse ROM with CD-ROMs, in spite of the similar names. A ROM is a series of words with addresses within the computer's memory space, just like RAM, whereas a CD-ROM is an input/output device, like a keyboard or disk.

modems, printers, mice and so on.

Specialized applications may have their own special I/O devices. For example, consider a vending machine, say for tickets for a regional railway system such as the San Franciso Bay Area's BART, which is capable of accepting dollar bills. The machine is likely to be controlled by a small computer. One of its input devices might be an optical sensor which senses the presence of a bill, and collects data which will be used to analyze whether the bill is genuine. One of the system's output devices will control a motor which is used to pull in the bill; a similar device will control a motor to dispense the railway ticket. Yet another output device will be a screen to give messages to the person buying the ticket, such as "please deposit 25 cents more."

The common feature of all of these examples is that they serve as interfaces between the computer and the "outside world." Note that in all cases, they are communicating with a *program* which is running on the computer. Just as you have in the past written programs which input from a keyboard and output to a monitor screen, programs also need to be written in specialized applications to do input/output from special I/O devices, such as the railway ticket machine application above. For example, the optical sensor would collect data about the bill, which would be input by the program. The program would then analyze this data to verify that the bill is genuine.

Note carefully the difference between memory and disk.[5] Memory is purely electronic, while disk is electromechanical, the latter referring to the fact that a disk is a rotating platter. Both memory and disk store bits, but because of the mechanical nature of disk, it is very much slower than memory—memory access is measured in nanoseconds (billionths of seconds) while disk access is measured in milliseconds (thousandths of seconds). The good thing about disk is that it is nonvolatile, so we can store files permanently.[6] Also, disk storage is a lot cheaper, per bit stored, than memory.

System Bus

A **bus** is a set of parallel wires (usually referred to as "lines"), used as communication between components. Our **system bus** plays this role in Figure 1— he CPU communicates with memory and I/O devices via the bus. It is also possible for I/O devices to communicate directly with memory, an action which is called **direct memory access** (DMA), and again this is done through the bus.[7]

The bus is broken down into three sub-buses:

- **Data Bus:**

  As its name implies, this is used for sending data. When the CPU reads a memory word, the memory sends the contents of that word along the data bus to the CPU; when the CPU writes a value to a memory word, the value flows along the data bus in the opposite direction.

  Since the word is the basic unit of memory, a data bus usually has as many lines as there are bits in a memory word. For instance, a machine with 32-bit word size would have a data bus consisting of 32 lines.

- **Address Bus:**

[5] Often confused by the fact that salespeople at computer stores erroneously call disk "memory."

[6] We couldn't do that with ROM, since we want to be able to modify the files.

[7] This is the basic view, but technically it applies more to older or smaller computers. In a PC, for instance, there will be extra chips which serve as "agents" for the CPU's requests to memory. For a description of how some common chips like this work, and their implications for software execution speed, see Chapter 5 of *Code Optimization: Effective Memory Usage*, by Kris Kaspersky, A-LIST, 2003.

When the CPU wants to read or write a certain word of memory, it needs to have some mechanism with which to tell memory *which* word it wants to read or write. This is the role of the address bus. For example, if the CPU wants to read Word 504 of memory, it will put the value 504 on the address bus, along which it will flow to the memory, thus informing memory that Word 504 is the word the CPU wants.

The address bus usually has the same number of lines as there are bits in the computer's addresses.

- **Control Bus:**

  How will the memory know whether the CPU wants to read or write? This is one of the functions of the control bus. For example, the control bus in typical PCs includes lines named MEMR and MEMW, for "memory read" and "memory write." If the CPU wants to read memory, it will **assert** the MEMR line, by putting a low voltage on it, while for a write, it will assert MEMW. Again, this signal will be noticed by the memory, since it too is connected to the control bus, and so it can act accordingly.

As an example, consider a machine with both address and word size equal to 32 bits. Let us denote the 32 lines in the address bus as A_{31} through A_0, corresponding to Bits 31 through 0 of the 32-bit address, and denote the 32 lines in the data bus by D_{31} through D_0, corresponding to Bits 31 through 0 of the word being accessed. Suppose the CPU executes an instruction to fetch the contents of Word 0x000d0126 of memory. This will involve the CPU putting the value 0x000d0126 onto the address bus. Remember, this is hex notation, which is just a shorthand abbreviation for the actual value,

```
00000000000000011010000000100100110
```

So, the CPU will put 0s on lines A_{31} through A_{20}, a 1 on Line A_{19}, a 1 on Line A_{18}, a 0 on Line A_{17}, and so on. At the same time, it will assert the MEMR line in the control bus. The memory, which is attached to these bus lines, will sense these values, and "understand" that we wish to read Word 0x000d0126. Thus the memory will send back the contents of that word on the data bus. If for instance c(0x000d0126) = 0003, then the memory will put 0s on Lines D_{31} through D_2, and 1s on Lines D_1 and D_0, all of which will be sensed by the CPU.

Some computers have several buses, thus enabling more than one bus transaction at a time, improving performance.

2.2 CPU Components

2.2.1 Intel/Generic Components

We will now look in more detail at the components in a CPU. Figure 2 shows the components that make up a typical CPU. Included are an **arithmetic and logic unit** (ALU), and various **registers**.

The ALU, as its name implies, does arithmetic operations, such as addition, subtraction, multiplication and division, and also several **logical** operations. The latter category of operations are similar to the `&&`, `||` and ! operators in the C language) used in logical expressions such as

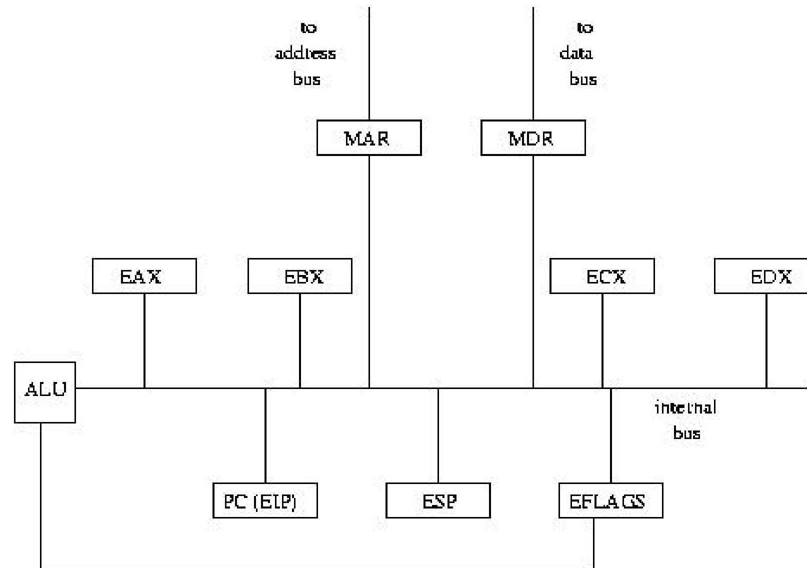

Figure 2: CPU internals

```
if (a < b && c == 3) x = y;
```

The ALU does not *store* anything. Values are input to the ALU, and results are then output from it, but it does not store anything, in contrast to memory words, which do store values. An analogy might be made to telephone equipment. A telephone inputs sound, in the form of mechanical vibrations in the air, and converts the sounds to electrical pulses to be sent to the listener's phone, but it does not store these sounds. A telephone tape-recording answering machine, on the other hand, does store the sounds which are input to it.

Registers are storage cells similar in function to memory words. The number of bits in a register is typically the same as that for memory words. We will even use the same c( ) notation for the contents of a register as we have used for the contents of a memory word. For example, c(PC) will denote the contents of the register PC described below, just as, for instance, c(0x22c4) means the contents of memory word 0x22c4. Keep in mind, though, that registers are not in memory; they are inside the CPU. Here are some details concerning the registers shown in Figure 2.

- **PC:** This is the **program counter**. Recall that a program's machine instructions must be stored in memory while the program is executing. The PC contains the address of the currently executing instruction. The term *PC* is a generic term; on Intel, the register is called EIP (Extended Instruction Pointer).
- **ESP:** The **stack pointer** contains the address of the "top" of a certain memory region which is called the **stack**. A stack is a type of data structure which the machine uses to keep track of function calls and other information, as we will see in Chapter 5.
- **EAX, EBX, etc.:** These are **data registers**, used for temporary storage of data.

- **EFLAGS:** The **processor status register** (PS), called EFLAGS on Intel machines, contains miscellaneous pieces of information, including the **condition codes**. The latter are indicators of information such as whether the most recent computation produced a negative, positive or zero result. Note that there are wires leading out of the ALU to the EFLAGS (shown as just one line in Figure 2). These lines keep the condition codes up to date. Each time the ALU is used, the condition codes are immediately updated according to the results of the ALU operation.

  Generally the PS will contain other information in addition to condition codes. For example, it was mentioned in MIPS and PowerPC processors give the operating system a choice as to whether the machine will run in big-endian or little-endian mode. A bit in the PS will record which mode is used.

- **MAR:** The **memory address register** is used as the CPU's connection to the address bus. For example, if the currently executing instruction needs to read Word 0x0054 from memory, the CPU will put 0x0054 into MAR, from which it will flow onto the address bus.

- **MDR:** The **memory data register** is used as the CPU's connection to the data bus. For example, if the currently executing instruction needs to read Word 0x0054 from memory, the memory will put c(0x0054) onto the data bus, from which it will flow into the MDR in the CPU. On the other hand, if we are writing to Word 0x0054, say writing the value 0x0019, the CPU will put 0x0019 in the MDR, from which it will flow out onto the data bus and then to memory. At the same time, we will put 0x0054 into the MAR, so that the memory will know to which word the 0x0019 is to be written.

- **IR:** This is the **instruction register**. When the CPU is ready to start execution of a new instruction, it fetches the instruction from memory. The instruction is returned along the data bus, and thus is deposited in the MDR. The CPU needs to use the MDR for further accesses to memory, so it enables this by copying the fetched instruction into the IR, so that the original copy in the MDR may be overwritten.

A note on the sizes of the various registers: The PC, ESP, and MAR all contain addresses, and thus typically have sizes equal to the address size of the machine. Similarly, the MDR typically has sizes equal to the word size of the machine. The PS stores miscellaneous information, and thus its size has no particular relation to the machine's address or word size. The IR must be large enough to store the longest possible instruction for that machine.

A CPU also has internal buses, similar in function to the system bus, which serve as pathways with which transfers of data from one register to another can be made. Figure 2 shows a CPU having only one such bus, but some CPUs have two or more. Internal buses are beyond the scope of this book, and thus any reference to a "bus" from this point onward will mean the system bus.

The reader should pay particular attention to the MAR and MDR. They will be referred to at a number of points in the following chapters, both in text and in the exercises—not because they are so vital in their own right, but rather because they serve as excellent vehicles for clarifying various concepts that we will cover in this book. In particular, phrasing some discussions in terms of the MAR and MDR will clarify the fact that some CPU instructions access memory while others do not.

Again, the CPU structure shown above should only be considered "typical," and there are many variations. RISC CPUs, not surprisingly, tend to be somewhat simpler than the above model, though still similar.

2.2.2 History of Intel CPU Structure

The earliest widely-used Intel processor chip was the 8080. Its word size was 8 bits, and it included registers named A, B, C and D (and a couple of others). Address size was 16 bits.

The next series of Intel chips, the 8086/8088,[8] and then the 80286, featured 16-bit words and 20-bit addresses. The A, B, C and D registers were accordingly extended to 16-bit size and renamed AX, BX, CX and DX ('X' stood for "extended"). Other miscellaneous registers were added. The lower byte of AX was called AL, the higher byte AH, and similarly for BL, BH, etc.

Beginning with the 80386 and extending to the Pentium series, both word and address size were 32 bits. The registers were again extended in size, to 32 bits, and renamed EAX, EBX and so on ('E' for "extended"). As of December 2006, the 64-bit generation is now becoming commonplace.

The pre-32-bit Intel CPUs, starting with 8086/8088, replaced the *single* register PC with a *pair* of registers, CS (for *code segment*) and IP (for *instruction pointer*). A rough description is that the CS register pointed to the **code segment**, which is the place in memory where the program's instructions start, and the IP register then specified the *distance* in bytes from that starting point to the current instruction. Thus by combining the information given in c(CS) and c(IP), we obtained the absolute address of the current instruction.

This is still true today when an Intel CPU runs in 16-bit mode, in which case it generates 20-bit addresses. The CS register is only 16 bits, but it represents a 20-bit address whose least significant four bits are implicitly 0s. (This implies that code segments are allowed to begin only at addresses which are multiples of 16.) The CPU generates the address of the current instruction by concatenating c(CS) with four 0 bits[9] and then adding the result to c(IP).

Suppose for example the current instruction is located at 0x21082, and the code segment begins at 0x21040. Then c(CS) and c(IP) will be 0x2104 and 0x0042, respectively, and when the instruction is to be executed, its address will be generated by combining these two values as shown above.

The situation is similar for stacks and data. For example, instead of having a single SP register as in our model of a typical CPU above, the earlier Intel CPUs (and current CPUs when they are running in 16-bit mode) use a pair of registers, SS and SP. SS specifies the start of the **stack segment**, and SP contains the distance from there to the current top-of-stack. For data, the DS register points to the start of the **data segment**, and a 16-bit value contained in the instruction specifies the distance from there to the desired data item.

Since IP, SP and the data-item distance specified within an instruction are all 16-bit quantities, it follows that the code, stack and data segments are limited to $2^{16} = 65,536$ bytes in size. This can make things quite inconvenient for the programmer. If, for instance, we have an array of length, say, 100,000 bytes, we could not fit the array into one data segment. We would need two such segments, and the programmer would have to include in the program lines of code which change the value of c(DS) whenever it needs to access a part of the array in the other data segment.

These problems are avoided by the newer operating systems which run on Intel machines today, such as Windows and Linux, since they run in 32-bit mode. Addresses are also of size 32 bits in that mode, and IP, SP and data-item distance are 32 bits as well. Thus a code segment, for instance, can fill all of memory, and segment switching as illustrated above is unnecessary.

[8] The first IBM PC used the 8088.

[9] This is one 0 hex digit. Note too that this concatenation is equivalent to multiplying by 16.

This continues to be the case for 64-bit machines, e.g. with 64-bit addresses.

2.3 The CPU Fetch/Execute Cycle

After its power is turned on, the typical CPU pictured in Figure 2 will enter its **fetch/execute cycle**, repeatedly cycling through these three steps, i.e. Step A, Step B, Step C, Step A, Step B, and so on:

- **Step A:** The CPU will perform a memory read operation, to fetch the current instruction. This involves copying the contents of the PC to the MAR, asserting a memory-read line in the control bus, and then waiting for the memory to send back the requested instruction along the data bus to the MDR. While waiting, the CPU will update the PC, to point to the following instruction in memory, in preparation for Step A in the next cycle.
- **Step B:** The CPU will copy the contents of MDR to IR, so as to free the MDR for further memory accesses. The CPU will inspect the instruction in the IR, and **decode** it, i.e. decide what kind of operation this instruction performs—addition, subtraction, jump, and so on.
- **Step C:** Here the actual execution of the operation specified by the instruction will be carried out. If any of the operands required by the instruction are in memory, they will be fetched at this time, by putting their addresses into MAR and waiting for them to arrive at MDR. Also, if this instruction stores its result back to memory, this will be accomplished by putting the result in MDR, and putting into MAR the memory address of the word we wish to write to.

After Step C is done, the next cycle is started, i.e. another Step A, then another Step B, and so on. Again, keep in mind that the CPU will continually cycle through these three steps as long as the power remains on. Note that all of the actions in these steps are functions of the *hardware*; the circuitry is designed to take these actions, while the programmer merely takes advantage of that circuitry, by choosing the proper machine instructions for his/her program.

A more detailed description of CPU operations would involve specifying its **microsteps**. These are beyond the scope of this book, and the three-step cycle described above will give sufficient detail for our purposes, though we *will* use them later to illustrate the term **clock speed**.

3 Software Components of the Computer "Engine"

There are many aspects of a computer system which people who are at the learning stage typically take for granted as being controlled by hardware, but which are actually controlled by software. An example of this is the backspace action when you type the backspace key on the keyboard. You are accustomed to seeing the last character you typed now disapppear from the screen, and the cursor moving one position to the left. You might have had the impression that this is an inherent property of the keyboard and the screen, i.e. that their circuitry was designed to do this. However, for most computer systems today this is not the case. The bare hardware will not take any special action when you hit the backspace key. Instead, the special action is taken by whichever **operating system** (OS) is being used on the computer.

The OS is software—a *program*, which a person or group of people wrote to provide various services to user programs. One of those services is to monitor keystrokes for the backspace key, and to take special actions

(move the cursor leftward one position, and put a blank where it used to be) when encountering that key. When you write a program, say in C, you do not have to do this monitoring yourself, which is a tremendous convenience. Imagine what a nuisance it would be if you were forced to handle the backspace key yourself: You would have to include some statements in each program you write to check for the backspace key, and to update the screen if this character is encountered. The OS relieves you of this burden.

But if you want this "burden"—say you are writing a game or a text editor and need to have characters read as they are typed, instead of waiting for the user to hit the Enter key—then the OS also will turn off the backspace, echo etc. if you request it. In UNIX, this is done via the ioctl() system call.

This backspace-processing is an example of one of the many services that an OS provides. Another example is maintenance of a file system. Again the theme is convenience. When you create a file, you do not have to burden yourself with knowing the physical location of your file on the disk. You merely give the file a name. The OS finds unused space on the disk to store your file, and enters the name and physical location in a table that the OS maintains. Subsequently, you may access the file merely by specifying the name, and the OS service will translate that into the physical location and access the file on your behalf. In fact, a typical OS will offer a large variety of services for accessing files.

So, a user program will make use of many OS services, usually by calling them as functions. For example, consider the C-language function **scanf()**. Though of course you did not write this function yourself, someone did, and and in doing so that person (or group of people) relied heavily on calls to an OS subroutine, **read()**. In terms of our "look under the hood" theme, we might phrase this by saying that a look under the hood of the C **scanf()** source code would reveal system calls to the OS function **read()**. For this reason, the OS is often referred to as "low-level" software. Also, this reliance of user programs on OS services shows why the OS is included in our "computer engine" metaphor—the OS is indeed one of the sources of "power" for user programs, just as the hardware is the other source of power.

To underscore that the OS services do form a vital part of the computer's "engine," consider the following example. Suppose we have a machine-language program—which we either wrote ourselves or produced by compiling from C—for a DEC computer with a MIPS CPU. Could that program be run without modification on a Silicon Graphics machine, which also uses the MIPS chip? The answer is no. Even though both machines do run a Unix OS, there are many different "flavors" of Unix. The DEC version of Unix, called Ultrix, differs somewhat from the SGI version, called IRIX. The program in question would probably include a number of calls to OS services—recall from above that even reads from the keyboard and writes to the screen are implemented as OS services—and those services would be different under the two OSs.

Thus even though individual instructions of the program written for the DEC would make sense on the SGI machine, since both machines would use the same type of CPU, some of those instructions would be devoted to OS calls, which would differ.

Since an OS consists of a program, written to provide a group of services, it follows that several different OSs—i.e. several different programs which offer different groups of services—could be run on the same hardware. For instance, this is the case for PCs. The most widely used OS for these CPUs is Microsoft Windows, but there are also several versions of Unix for PCs, notably the free, public-domain Linux and the commercial SCO.

4 Speed of a Computer "Engine"

What factors determine the speed capability of a computer engine? This is an extremely complex question which is still a subject of hot debate in both academia and the computer industry. However, a number of factors are clear, and will be introduced here. The presentation below will just consist of overviews, and the interested reader should pursue further details in books on computer architecture and design. However, it is important that the reader get some understanding of the main issues now; at the very least enough to be able to understand newspaper PC ads! The discussion below is aimed at that goal.

4.1 CPU Architecture

Different CPU types have different instruction and register sets. The Intel family, for example, has a very nice set of character-string manipulation instructions, so it does such operations especially quickly. This is counterbalanced somewhat by the fact that CPUs in this family have fewer registers than do CPUs in some other families, such as those in most of the newer machines.[10] Since registers serve as local—and thus fast—memory, the more of them we have, the better, and thus the Intel family might be at a disadvantage in this respect.

4.2 Parallel Operations

One way to get more computing done per unit time is do several things at one time, i.e. in parallel. Most modern machines, even inexpensive ones such as personal computers, include some forms of parallelism.

For example, most CPUs perform **instruction prefetch**: During execution of the current instruction, the CPU will attempt to fetch one or more of the instructions which follow it sequentially—i.e. at increasing addresses—in memory.[11] For example, consider an Intel chip running in 16-bit mode. In this mode the chip has 20-bit addresses. Consider a two-byte instruction at location 0x21082. During the time we are executing that instruction the CPU might attempt to start fetching the next instruction, at 0x21084. The success or failure of this attempt will depend on the duration of the instruction at 0x21082. If this attempt is successful, then Step A can be skipped in the next cycle, i.e. the instruction at 0x21084 can be executed earlier.

Of course, the last statement holds only if we actually do end up executing the instruction at 0x21084. If the instruction at 0x21082 turns out to be a jump instruction, which moves us to some other place in memory, we will not execute the instruction at 0x21084 at all. In this case, the prefetching of this instruction during the execution of the one at 0x21082 would turn out to be wasted. Modern CPUs have elaborate jump-prediction mechanisms to try to deal with this problem.

Intel CPUs will often fetch *several* downstream instructions, while for RISC CPUs we might be able to fetch only one. The reason for this is that RISC instructions, being so simple, have such short duration that there just is not enough time to fetch more than one instruction.

Instruction prefetching is a special case of a more general form of parallelism, called **pipelining**. This concept treats the actions of an instruction as being like those of an assembly line in a factory. Consider an

[10] This does not include the newer Intel chips, such as the Pentium, because these chips had to be designed for compatibility with the older ones.

[11] The prefetch may insted be from the **cache**, which we will discuss later.

automobile factory, for instance. Its operation is highly parallel, which the construction of many cars being done simultaneously. At any given time, one car, at any early stage in the assembly line, might be having its engine installed, while another car, at a later stage in the line, is having its transmission installed. Pipelined CPUs (which includes virtually every modern CPU) break instruction execution down into several stages, and have several instructions executing at the same time, in different stages. In the simple case of instruction prefetch, there would be only two stages, one for the fetch (Step A) and the other for execution (Steps B and C).

Most recently-designed CPUs are **superscalar**, meaning that a CPU will have several ALUs. This is another way in which we can get more than one instruction executing at a time.

Yet another way to do this is to have multiple CPUs! In **multiprocessor** systems with n CPUs, we can execute n instructions at once, instead of one, thus potentially improving system speed by a factor of n. Typical speedup factors in real applications are usually much less than n, due to such overhead as the need for the several CPUs to coordinate actions with each other, and not get in each other's way as they access memory.

The classical way of constructing multiprocessor systems was to connect several CPUs to the system bus. However, as of 2006, it is common for even many low-end CPU chips to be **dual core**, meaning that the chip actually contains *two* CPUs. This has brought multiprocessor computing into the home.

It used to be very expensive to own a multiprocessor machine. The CPUs would all be connected to the bus, which increases manufacturing costs, etc. But by having more than once CPU on a chip, the expense is small.

Note that chip manufacturers have found that it makes better economic sense now for them to use chip space for extra processors, rather than continuing to increase processor speed.

4.3 Clock Rate

Recall that each machine instruction is implemented as a series of **microsteps**. Each microstep has the same duration, namely one **clock cycle**, which is typically set as the time needed to transfer a value from one CPU register to another. The CPU is paced by a **CPU clock**, a crystal oscillator; each pulse of the clock triggers one microstep.

In 2002, clock rates of over 1 **gigahertz**, i.e. 1 billion cycles per second, became common in PCs. This is quite a contrast to the 4.77 megahertz clock speed of the first IBM PC, introduced in 1981.

Each instruction takes a certain number of clock cycles. For example, an addition instruction with both operands in CPU registers might take 2 clock cycles on a given CISC CPU, while a multiply instruction with these operands takes 21 clock cyles on the same CPU. If one of the operands is in memory, the time needed will increase beyond these values.

RISC machines, due to the fact that they perform only simple operations (and usually involving only registers), tend to have instructions which operate in a single clock cylce.

The time to execute a given instruction will be highly affected by clock rate, even among CPUs of the same type. For example, as mentioned above, a register-to-register addition instruction might typically take 2 microsteps on a CISC CPU. Suppose the clock rate is 1 gigahertz, so that a clock cycle takes 10^{-9} seconds, i.e. 1 **nanosecond**.[12] Then the instruction would take 2 nanoseconds to execute.

[12] A nanosecond is a billionth of a second.

Within a CPU family, say the Intel family, the later members of the family typically run a given program much faster than the earlier members, for two reasons related to clock cycles:

- Due to advances in fabrication of electronic circuitry, later members of a CPU family tend to have much faster clock rates than do the earlier ones.
- Due to more clever algorithms, pipelining, and so on, the later members of the family often can accomplish the same operation in fewer microsteps than can the earlier ones.

4.4 Memory Caches

4.4.1 Need for Caching

In recent years CPU speeds have been increasing at very impressive rates, but memory access speeds have not kept pace with these increases. Remember, memory is *outside* the CPU, not in it. This causes slow access, for various reasons.

- The physical distance from the CPU is on the order of inches, whereas it is typically less than an inch within the CPU. Even though signals travel at roughly 2/3 the speed of light, there are so many signals sent per second that even this distance is a factor causing slowdown.
- An electrical signal propagates more slowly when it leaves a CPU chip and goes onto the bus toward memory. This is due to the bus wires being far thicker than the very fine wires within the CPU.
- Control buses typically must include **handshaking** lines, in which the various components attached to the bus assert to coordinate their actions. One component says to the other, "Are you ready," and that component cannot proceed until it gets a response from the other. This causes delay.

4.4.2 Basic Idea of a Cache

To solve these problems, a storage area, either within the CPU or near it or both, is designed to act as a **memory cache**. The cache functions as a temporary storage area for a *copy* of some small subset of memory, which can be accessed locally, thus avoiding a long trip to memory for the desired byte or word.[13]

Say for example the program currently running on the CPU had in its source file the line

```
char x;
```

and suppose that **x** is stored in byte 1205 of memory, i.e. **&x** is 1205. Consider what occurs in each instruction in the program that reads **x**. Without a cache, the CPU would put 1205 on the address bus, and assert the MEMR line in the control bus. The memory would see these things on the bus, and copy whatever is in location 1205 to the data bus, from which the CPU would receive it.

[13]Modern computers typically have two caches, an **L1** cache inside the CPU, and an **L2** cache just outside it. When the CPU wants to access a byte or word, it looks in L1 first; if that fails, it looks in L2; and if that fails, it goes to memory. Here, though, we will assume only an L1 cache.

If we did have a cache, the CPU would look first to see if the cache contains a copy of location 1205. If so, the CPU does not have to access memory, a huge time savings. This is called a cache **hit**. If the CPU does not find that the cache contains a copy of location 1205—a cache **miss**—then the CPU must go to memory in the normal manner.

4.4.3 Blocks and Lines

Memory is partitioned into **blocks**, of fixed size. For concreteness, we will assume here that the block size is 512 bytes. Then Bytes 00000-000511 form Block 0, Bytes 00512-001023 form Block 1, and so on. A memory byte's block number is therefore its address divided by the block size. Equivalently, since $log_2 512 = 9$, if our address size is 32 bits, then the location's block number is given in the most-significant 23 bits of the address, i.e. all the bits of the address except for the lower 9. The lower 9 bits then give location of the byte within the block.[14] Note that the partitioning is just conceptual; there is no "fence" or any other physical boundary separating one block from another.

This organization by blocks comes into play in the following manner. Consider our example above in which the CPU wanted to read location 1205. The block number is $\lfloor 1205/512 \rfloor = 2$. The CPU would not really search the cache for a copy of byte 1205 or even word 1205. Instead, the CPU would search for a copy of the entire block containing location 1205, i.e. a copy of block 2.

The cache is partitioned into a fixed number of slots, called **lines**. Each slot has room for a copy of one block of memory, plus an extra word in which additional information is stored. This information includes a **tag**, which states which block has a copy in memory right now. This is vital, of course, as otherwise when the CPU looked in the cache, it wouldn't know whether this line contains a copy of the desired block, as opposed to a copy of some other block.

At any given time, the cache contains copies of some blocks from memory, with different blocks being involved at different times. For example, suppose again the CPU needs to read location 1205. It then searches the cache for a copy of block 2. In each line in which it searches, the CPU would check the tag for that line. If the tag says "This line contains a copy of block 2," then the CPU would see that this is a hit. The CPU would then get byte 1205 from this line as follows. As we saw above, byte 1205 is in block $\lfloor 1205/512 \rfloor = 2$, and using the same reasoning,[15] *within* that block, this byte is byte number 1205 mod 512 = 181. So, the CPU would simply look at the 181st byte in this line.[16]

If on the other hand, the CPU finds that block 2 does not have a copy in the cache, the CPU will read the *entire* block 2 from memory, and put it into some cache line. It may have to remove a copy of some other block in doing so; the term used for this is that that latter block copy is **evicted**.

The total number of lines varies with the computer; the number could be as large as the thousands, or on the other extreme less than ten. The larger the cache, the higher the hit rate. On the other hand, bigger caches are slower, take up too much space on a CPU chip in the case of on-chip caches, and are more expensive in the case of off-chip caches.

[14] Suppose we have 10 classrooms, each with 10 kids, sitting in 10 numbered seats, according to the kids' ID numbers. Kids 0-9 are in classroom 0, 10-19 are in classroom 1, and so on. So for instance kid number 28 will be in classroom 2, sitting in seat 8 of that room. Note how the 2 comes from the upper 1 digit of the ID number, and the 8 comes from the lower 1 digit—just like the block number of a byte is given by the upper 23 bits in the address, while the byte number within block is given by the lower 9 bits.

[15] Recall note 14. Make SURE you understand this point.

[16] This means the block begins with its "0th" byte.

4.4.4 Direct-Mapped Policy

One question is which lines the CPU should search in checking whether the desired block has a copy in the cache. In a **fully-associative** cache, the CPU must inspect all lines. This gives maximum flexibility, but if it is large then it is expensive to implement and the search action is slower, due to the overhead of complex circuitry.

The other extreme is called **direct-mapped**.[17] Here the CPU needs to look in only *one* cache line. By design, the desired block copy will either be in that particular line or not in any line at all. The number of the particular line to be checked is the block number mod the number of lines. Let us suppose for illustration that there are 32 lines in the cache. In our example above, byte 1205 was in block 2, and since 2 mod 32 = 2, then to check whether block 2 has a copy in the cache, the CPU needs to look only at line 2. If the tag in line 2 says "This is a copy of block 2," then the CPU sees we have a hit. If not, the CPU knows it is a miss. The basic design of this direct-mapped cache is that a copy of block 2 will either be in the cache at line 2 or else the block will not have a copy in the cache at all.

Now suppose after successfully reading byte 1205, at some time later in the execution of this program we need to read byte 17589. This is block $\lfloor 17589/512 \rfloor = 34$. Since 34 mod 32 = 2, the CPU will again look at line 2 in the cache. But the tag there will say that this line currently contains a copy of block 2. Thus there will be a miss. The CPU will bring in a copy of block 34, which will replace the copy of block 2, and the tag will be updated accordingly.

4.4.5 What About Writes?

One issue to be resolved is what to do when we have a cache hit which is a write. The issue is that now the copy of the block in the cache will be different from the "real" block in memory. Eventually the two must be made consistent, but when? There are two main types of policies:

- Under a **write-through** policy, any write to the cache would also be accompanied by an immediate corresponding write to memory.
- Under a **write-back** policy, we do not make the memory block consistent with its cache copy until the latter is evicted. At that time, the entire block is written to memory.

Which policy is better will depend on the program being run. Suppose for example we only write to a cache line once before it is evicted. Then write-back would be rather disastrous, since the entire block would be written to memory even though the inconsistency consists of only one byte or word. On the other hand, in code such as

```
for (i = 0; i < 10000; i++)
   sum += x[i];
```

if **sum** is the only part of its cache line being used, we only care about the final value of **sum**, so updating it to memory 10,000 times under a write-through policy would be highly inefficient.

[17] A compromise scheme, called **set-associative**, is very common.

4.4.6 Programmability

As noted in Section 4.4.5, the behavior of a particular write policy depends on which program is being run. For this reason, many CPUs, e.g. the Pentium, give the programmer the ability to have some blocks subject to a write-through policy and others subject to write-back.

Similarly, the programmer may declare certain blocks not cacheable at all. The programmer may do this, for example, if the programmer can see that this block would have very poor cache behavior. Another reason for designating a block as noncacheable is if the block is also being accessed by a DMA device.

4.4.7 Details on the Tag and Misc. Line Information

Consider line 2 in the cache. We will search that line if the number of the block we want is equal to 2, mod 32. That means that the tag, which states the number of the block copy stored in this line, must have 00010 as its last 5 bits. In view of that, it would be wasteful to store those bits. So, we design the tag to consist only of the remainder of the block number, i.e. the upper 18 bits of the block number.

Let's look at our example 17589 above again. Recall that this is in block 34 = 00000000000000000100010. Since the last 5 bits are 00010, the CPU will examine the tag in cache line 2. The CPU will then see whether the tag is equal to 000000000000000001, the upper 18 bits of 34. If so, it is a hit.

At the beginning, a line might not contain any valid data at all. Thus we need another bit in the line, called the Valid Bit, which tells us whether there is valid data in that line (1 means valid).

If a write-back policy is used, we need to know whether a given cache line has been written to. If it has, then the line must be copied back to memory when it is evicted, but if not, we would want to skip that step. For this purpose, in addition to the tag, the line maintains a Dirty Bit, "dirty" meaning "has been written to."

Since we saved some bits in the storage of the tag, we use that space for the Valid and Dirty Bits.

4.4.8 Why Caches Usually Work So Well

Experience shows that most programs tend to reuse memory items repeatedly within short periods of time; for example, instructions within a program loop will be accessed repeatedly. Similarly, they tend to use items which neighbor each other (as is true for the data accesses in Version I above), and thus they stay in the same block for some time. This is called the principle of **locality of reference**, with the word "locality" meaning "nearness"—nearness in time or in space.

For these reasons, it turns out that cache misses tend to be rare. A typical hit rate is in the high 90s, say 97%. This is truly remarkable in light of the fact that the size of the cache is tiny compared to the size of memory. Again, keep in mind that this high hit rate is due to the locality, i.e. the fact that programs tend to NOT access memory at randomly-dispersed locations.

4.5 Disk Caches

We note also that one of the services an OS might provide is that of a **disk cache**, which is a software analog of the memory caches mentioned earlier. A disk is divided into **sectors**, which on PCs are 512 bytes each in size. The disk cache keeps copies of some sectors in main memory. If the program requests access to a certain disk sector, the disk cache is checked first. If that particular sector is currently present in the cache,

then a time-consuming trip to the disk, which is much slower than accessing main memory, can be avoided.[18] Again this is transparent to the programmer. The program makes its request to the disk-access service of the OS, and the OS checks the cache and if necessary performs an actual disk access. (Note therefore that the disk cache is implemented in software, as opposed to the memory cache, which is hardware, though hardware disk caches exist too.)

4.6 Web Caches

The same principle is used in the software for Web servers. In order to reduce network traffic, an ISP might cache some of the more popular Web pages. Then when a request for a given page reaches the ISP, instead of relaying it to the real server for that Web page, the ISP merely sends the user a copy. Of course, that again gives rise to an update problem; the ISP must have some way of knowing when the real Web page has been changed.

[18]Note, though, that the access to the cache will still be somewhat slower than the access to a register. Here is why:

Suppose we have 16 registers. Then the register number would be expressed as a 4-bit string, since $16 = 2^4$. By contrast, say our cache contains 1024 lines, which would mean that we need 10 bits to specify the line number. Digital circuity to decode a 10-bit ID is slower than that for a 4-bit ID. Moreover, there would be considerable internal circuitry delays within the cache itself.