Chapter 2
Major Components of Computer "Engines"

Norman Matloff
University of California at Davis

February 27, 2002

Contents

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 the IBM PC's Intel 80386 CPU certainly won't run on an Apple Macintosh's Motorola PowerPC CPU. 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?

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 2.1.

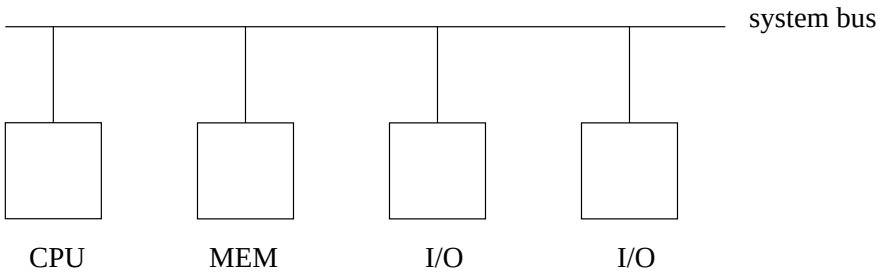

Figure 2.1

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

This was described in Chapter 1. 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.[1] 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.[2]

[1] 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.

[2] Do not confuse ROM with CD-ROMs, in spite of the similar names. A ROM is a series of words with addresses within the

I/O Devices

A typical computer system will have several **input/output** devices, possibly even hundreds of them (Figure 2.1 shows two of them). Typical examples are keyboards/monitor screens, floppy and fixed disks, CD-ROMs, 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.

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 2.1—the 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.

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:**

  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.

computer's memory space, just like RAM, whereas a CD-ROM is an input/output device, like a keyboard or disk.

- **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,

```
00000000000000001101000000100100110
```

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

We will now look in more detail at the components in a CPU. Figure 2.2 shows the components that make up a typical CPU. Included are an **arithmetic and logic unit** (ALU), and various **registers**.

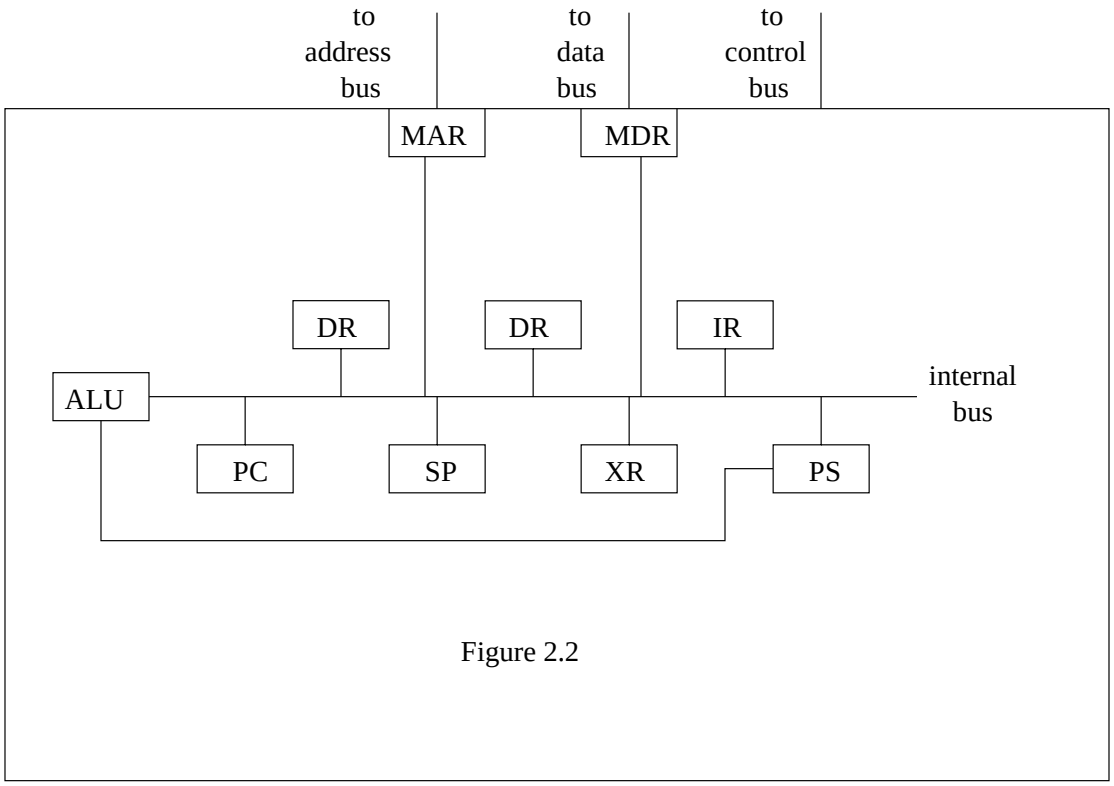

Figure 2.2

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

```
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. (For convenience, we are assuming 16-bit words and addresses here.) 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 2.2.

- **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.

- **SP:** 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.

- **XR:** An **index register** helps programs access arrays. Its name comes from the fact that in an array element, say y[i] for an **int** array y, the subscript i is often called the **index**. The instruction itself would specify the address of y, and isizeof(int) would be placed in XR. The circuitry in the CPU adds these two quantities, thus producing the proper location to access y[i].[3]

- **PS:** The **processor status register** 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 PS (shown as just one line in Figure 2.2). 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 Chapter 1 that 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.

- **DRs: Data registers** are usually used as "fast memory," i.e. as temporary places to store data to which we need quick access. Because they are in the CPU, an instruction executing within the CPU can access them much faster than it can access memory, since memory is outside the CPU (see Figure 2.1). Different CPU types have different numbers of DRs; two are pictured in Figure 2.2.

- **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.

[3]This is why arrays in the C language begin with the subscript 0 rather than 1 (the latter is used in the Pascal language, for instance). If the first array element had the subscript 1, then the address of y[i] would be (i-1)sizeof(int), thus requiring an extra subtraction operation. The C language was developed with the philosophy that it be close to the hardware, and this is an example.

A note on the sizes of the various registers: The PC, SP, XR and MAR all contain addresses, and thus typically have sizes equal to the address size of the machine. Similarly, DRs and the MDR typically have 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 2.2 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.

2.2.1 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,[4] 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 has been 32 bits. The registers were again extended in size, to 32 bits, and renamed EAX, EBX and so on ('E' for "extended").

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 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[5] 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.

[4] The first IBM PC used the 8088.

[5] concatenation is equivalent to multiplying by 16.

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 98 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.

2.3 The CPU Fetch/Execute Cycle

After its power is turned on, the typical CPU pictured in Figure 2.2 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**.

2.3.1 Example

The Mac-1 CPU is intended to be simple but representative of CISC machines. It is a mythical machine designed by Andrew Tanenbaum, a well-known computer science professor and author in the Netherlands. If you are interested, you can learn more about it in the writeup of a Mac-1 simulator written by Professor Rosalee Nerheim-Wolfe at DePaul University in Chicago; see `http://heather.cs.ucdavis.edu/~matloff/depaulmicmac.html`

The Mac-1 has only three programmer-visible registers,[6] a PC and SP, and an **accumulator** register, AC. This is a rather small number of registers, even for a CISC machine, so SP and AC tend to play multiple roles. SP works not only like a stack pointer but also as a kind of an index register. Like most accumlators, AC works as a data register, but it too plays other roles, as we will see. (We will go into more detail in the upcoming chapters.) Mac-1 has 16-bit words and 12-bit (i.e. three hex digits) addresses. Mac-1 has no instructions capable of addressing individual bytes, so only words have addresses; the addresses of contiguous words therefore differ by 1.

Let us illustrate the fetch/execute cycle with an example of Mac-1's **addd** ("add-direct") instruction. Applied to a memory location x, **addd** adds c(x) to the current value in AC, and stores the sum back in AC, i.e. **addd**'s action is

```
c(AC)  <--  c(AC) + c(x)
```

The coding for **addd** consists of the bit string 0010, i.e. 0x2, concatenated with the 12-bit coding for x. If **addd** is to access memory location 0x1c4, for instance, then 0x21c4, i.e.

```
0010000111000100
```

Again, a program's instructions are stored in main memory during the program's execution. Say this instruction 0x21c4 is stored at memory location 0x803, and suppose that currently AC and memory location 0x1c4 contain 0x0002 and 0x0003, respectively. Then the sequence of actions which will take place during the execution of the instruction is as follows:

Step A: To fetch the instruction, the CPU will copy c(PC) to the MAR, so c(MAR) becomes 0x803. The CPU also increments the PC by 1, to 0x804, so that when the next Step A comes around, the next instruction after the one at 0x803 will be fetched. The CPU asserts a read line in the control bus, and the memory responds by sending c(0x803) along the data bus, from which it flows into the MDR. The value in MDR now will be 0x21c4, the fetched instruction. The CPU will now look as in Figure 2.3:

[6] In our model above, MAR, MDR and IR are not considered to be visible to the programmer, because machine instructions cannot explicitly access them. Mac-1 has such registers too.

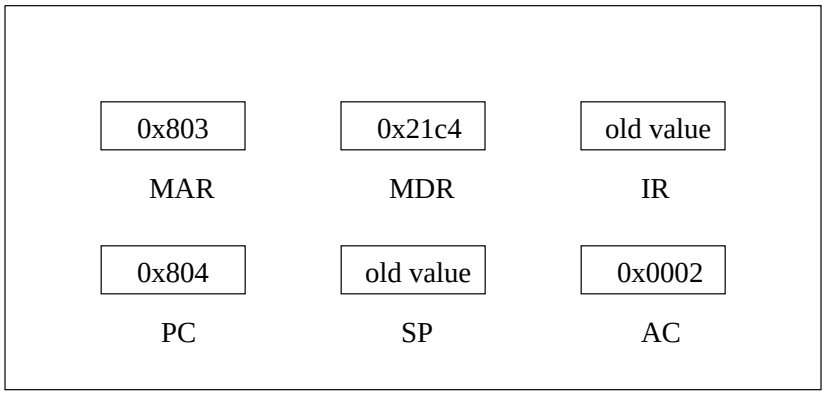

Figure 2.3

(The figure merely reports "old value" for c(SP) and c(IR), meaning that those contents have not changed from their old values, values which are irrelevant to the discussion.)

Step B: The CPU copies c(MDR) to the IR, and starts the decoding. For the latter, the CPU notices that the first four bits of the instruction are 0010, the **op code** for **addd**. The CPU will now look as in Figure 2.4:

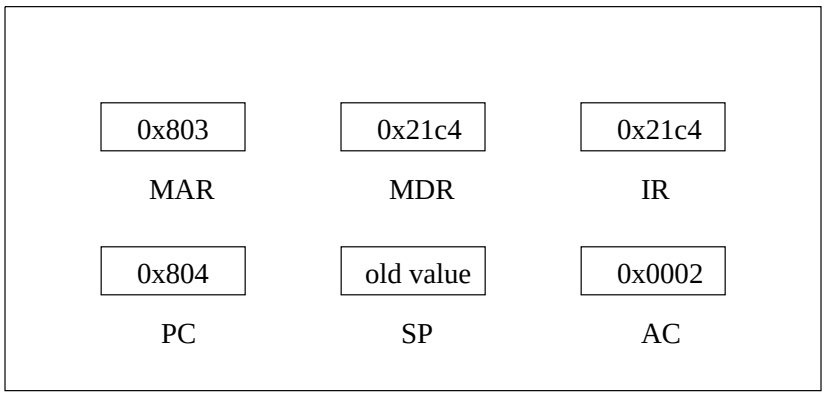

Figure 2.4

Step C: The CPU looks at the lower 12 bits of the instruction, and finds them to have the value 0x1c4. Since the action of **addd** is to add the contents of main memory at that location to AC, the CPU must fetch those contents. Thus the CPU copies those 12 bits to the MAR, and asserts the read line in the control bus. The memory responds by sending c(0x124) down the data bus; c(MDR) will now be 3, i.e. 0000000000000011. The CPU will now input c(MDR) and c(AC) into the ALU, and have the ALU do an addition. The sum, 5, will now be routed back to the AC, and Step C and the instruction are finished. The CPU will now look as in Figure 2.5:

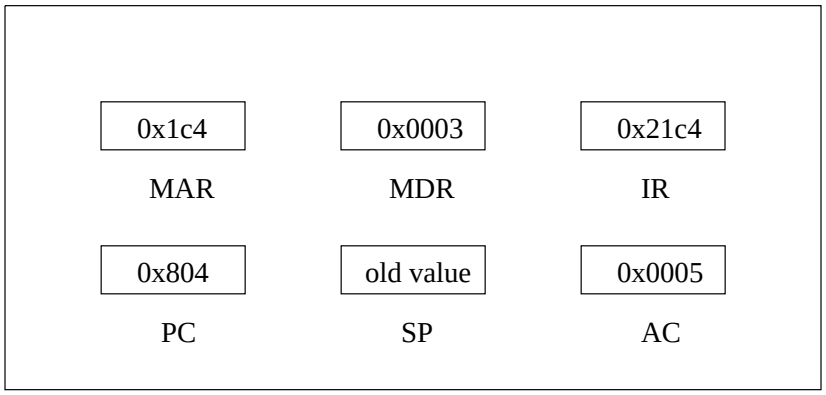

Figure 2.5

There now will be another Step A, starting the execution of the next instruction, and so on.

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.

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.[7] 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.

[7] 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.

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.[8] 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 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. The hardware—especially the CPU and the bus—must be set up to enable such coordination, by the way.

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

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

one microstep.

In 1996, clock rates of over 100 **megahertz**, i.e. 100 million 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 100 megahertz, so that a clock cycle takes 10^{-8} seconds, i.e. 10 **nanoseconds**.[9] Then the instruction would take 20 nanoseconds to execute.

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:

- (a) 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.
- (b) 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 Access Time

In recent years CPU speeds have been increasing at very impressive rates, but memory access speeds have not kept pace with these increases. There are a number of reasons for this, such as the fact that an electrical signal propagates more slowly when it leaves a CPU chip, and the fact that control buses often must include **handshaking** lines, in which the various components attached to the bus assert to coordinate their actions. These considerations are beyond the scope of this text, but the important point in our context here is that memory access speed is a major bottleneck to today's computers. Note, for instance, that if we have two CPUs which are completely identical in all respects except that one has, say, a 50 MHz clock and the other 100 MHz, it will not be the case that the latter system will run twice as fast as the former.

Thus solutions to this problem are of great importance. We have already seen two such solutions. One of these solutions is the instruction prefetching described above. This does not make memory access physically faster, but it does something even better: It makes the memory access time *appear* to be 0, since Step A is skipped, by "hiding" the instruction fetch activity behind the execution of the previous instruction. Another solution to the memory access time bottleneck is that of data registers, described earlier in this section. Most CPUs include a few data registers. These act like memory, but since they are internal to the CPU, they can be accessed quite quickly, avoiding the problems described in the last paragraph.

The bus size may affect memory access speed too. For instance, consider a CPU which operates on 32-bit words. The "normal" width of the data bus to which such a CPU is attached would be 32 lines, but a 64-line

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

data bus could be used to fetch two (consecutive) words from memory at a time, thus greatly increasing average memory-access speed.

Secondary memory speed plays a role in programs which use it. A slow disk drive, for example, will substantially impair the performance of programs which do a lot of disk accesses.

4.4.1 Memory Caches

An extension of the idea of coping with slow memory by storing things in registers is the following: 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.

To do this, the memory is first partitioned into **blocks**, say of 512 bytes each (Bytes 00000-00511 form Block 0, Bytes 00512-01023 form Block 1, and so on). A memory location's block number is its address divided by the block size. Note that there the partitioning is just conceptual; there is no "fence" or any other physical boundary separating one block from another.

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 the block number is stored. 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. At any given time, the cache contains copies of some blocks from memory, with different blocks being involved at different times.

The cache itself is also memory.[10] Though variations are possible in some senses, we will assume here that it is therefore merely a long sequence of words. Say the block size is 512 bytes, i.e. 128 words on a 32-bit machine. Then the first word in the cache would contain the block number, if any, of the block which is currently in Line 0 of the cache, and the next 128 words would be that block itself.[11] In other words, the first 129 words would be devoted to Line 0. The next 129 words would be similarly devoted to Line 1, and so on.

When the CPU needs to access a word in memory, it will check the cache first. To do so, it will calculate the block number of the desired word, and then look in the cache to see if that block is in the cache. If the CPU is lucky enough for this to be the case(a **cache hit**), it will access the copy of the word there in the cache, thus saving a time-consuming trip to memory for that access. If the block is not currently in the cache (a **cache miss**), the *entire* block containing the desired word – not just the word itself – is copied in from memory, and placed in one of the lines in the cache.

That of course means that we need to have at least one line already empty, in order to receive that new block. Some block currently in the cache must be evicted to make room for the new block. What becomes of the old block?

There are two approaches to handling writes to a cache. Consider a C statement like

```
X = 12;
```

[10] Since a cache is usually small, we can afford to construct it out of fast, expensive memory.

[11] We would also need to have something to indicate whether any memory block is contained in this line. To do this, we could, say, adopt the code -1 for an empty line. In other words, if Line 0 were empty, we would code its block number as -1.

Say X has a copy in the cache. Under the **write-through** approach, when this write to X is done by the CPU in the cache, the CPU also does the corresponding write to the real X in memory. Thus memory is always up to date.

On the other hand, some caches are designed to use the **write-back** approach. Here, the CPU does not immediately update memory. Instead, the CPU sets a **dirty bit** for that line, indicating that the copy of the block in that line is at least somewhat inconsistent with the real block in memory. Eventually, this block's copy in the cache will be evicted to make room for some new block. At that time, the CPU sees that the dirty bit for that line is set, and the the CPU copies the entire line back to the corresponding block in memory. (It has to copy the entire line, since it does not know which portions of the line have been written to.)

The write-through policy sounds more natural at first, but consider the situation in which the same few variables are written too repeatedly while a block has a copy in the cache. It would be wasteful if we were to update the memory each time. After all, the only change that really needs to be reflected in memory is the last one before the block is evicted.

When a new block is brought in from memory, which line is it put into? There are two main classes of policies for this too. Under an **associative** policy, the new block can be put into any line (evicting an existing block if necessary). At first this seems like it would make the subsequent cache searches very slow: Recall that when the CPU needs to access a memory word, it will calculate the block number of that word, and then search through the cache to see if that block is there. In our example above, for instance, that would mean looking at the first word in the cache, the 130th word in the cache, the 259th word in the cache, and so on. If we do this search sequentially, it could take a long time. But if we are willing to put in a lot of circuitry, we can search all those words simultaneously, which is called an "associative" search.

The other main approach uses **direct mapping**. Consider Block b in memory, and let k = b mod l, where l is the number of lines in the cache. Then if Block b has a copy in a direct-mapped type of cache, the copy will be in Line k. This is nice, because in a search, the CPU need only look at Line k; if Block b is there, fine, but if it is not there, we are guaranteed that it won't be in any other line either. Note in a direct-mapped cache, a cache miss may result in an eviction even if the cache is not full. This, plus the lack of flexibility,[12] implies that the miss rate might be higher.

An important issue in cache design is the **block replacement policy**. As mentioned above, when a cache miss occurs, the cache must remove one of its current blocks to make room for the one which is the subject of the current memory access request. Which block should be removed?

In the direct-mapped case, the answer is mandated: We must remove the block in Line k. But in associative caches, we have more choices. Hopefully the removed block will be one which will not be requested again for a long time into the future; since each block replacement is a time-consuming operation, we hope to postpone that as long as possible. But since future memory request patterns cannot be predicted, the designer of the cache must settle upon some policy that would seem to work "reasonably well" over a wide range of programs.

Most associative caches are designed to use a **Least Recently Used** (LRU) policy, or a variation of it. As the name implies, this policy chooses for replacement whichever block has gone the longest time without being accessed,[13] among all blocks currently in the cache. The motivation for this is an expectation that if a block has not been accessed for a long time, the probability is lower that it will be accessed in the near future, and thus can be "safely" replaced. Keep in mind, though, that the hardware is making this decision entirely on

[12] There is no way that two blocks with the same k could be in the cache at the same time.

[13] Or *approximately* so, in practical implementations.

its own, and carries out the replacement by itself too. The programmer does not have get involved with this process at all.

It is important to note that data registers and a memory cache, though both having the goal of avoiding memory access, are very different in their "visibility" to the programmer. The programmer directly accesses the registers, by including instructions in his/her program to access them. A cache, on the other hand, is **transparent** to the programmer, meaning that the CPU does all the management of the cache, i.e. checking for hit/miss, and doing the block replacement upon finding a miss. As far as the programmer is concerned, it *appears* as if all memory accesses really are to main memory, rather than some of them being intercepted by the cache. Note by the way that cache policies, e.g. write-through vs. write-back, are solely the province of the hardware designer; they are locked in to the hardware.

However, even though the actions of the cache are transparent to the programmer, it is important that the programmer recognize the existence of a cache in designing his/her code. For example, consider the following two equivalent pieces of C code, which both calculate the sum of all elements in a two-dimensional array declared to have M rows and N columns of integers:

```
Version I:

Sum = 0;
for (I = 0; I < M; I++)
   for (J = 0; J < N; J++)
      Sum += X[I][J];


Version II:

Sum = 0;
for (J = 0; J < N; J++)
   for (I = 0; I < M; I++)
      Sum += X[I][J];
```

Both versions are "correct," i.e. will give the same answer. However, they access memory in completely different patterns, that is, in different orders. Suppose, for simplicity: The machine has 16-bit words; block size is 64 bytes; the cache is associative and has 2 cache lines; M = 4 and N = 64; the compiler has placed the array X to begin at location 0x0c80, i.e. 3200 base-10, right at the beginning of Block 50[14], since 3200/64 = 50; the array X is global and the compiler stores it in nonreverse order, i.e. X[0][0] is at location 0x0c80, X[0][1] is at 0x0c82, and so on.

Assume that the cache is initially empty. Then the reader should verify that Version I of the code will result in eight cache misses, but Version II will generate a lot more—256, a hit rate of 0! Here are the first few actions in the case of Version II:

1. Cache is empty.

[14] change if the starting address of X were, say, 3204.

2. Program tries to access X[0][0], resulting in a cache miss.
3. CPU loads the whole block containing X[0][0], which will be X[0][0], X[0][1], X[0][2], ..., X[0][31], into the first cache line.
4. Program tries to access X[1][0], resulting in another cache miss.
5. CPU loads the whole block containing X[1][0], which will be X[1][0], X[1][1], X[1][2], ..., X[1][31], into the second cache line.
6. Program tries to access X[2][0], resulting in another cache miss.
7. CPU loads the whole block containing X[2][0], which will be X[2][0], X[2][1], X[2][2], ..., X[2][31], into the first cache line, since it is least recently used.
8. Et cetera.

So, two supposedly equivalent pieces of code might have very *different* run times. Once again, it does pay to "look under the hood."

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, around 5% of all accesses.

Most modern CPU chips have internal caches. In addition, it is common to also include a larger, secondary cache between the CPU and the system bus. The primary cache (on the CPU chip) contains a copy of a subset of the contents of the secondary cache, which in turn contains a copy of some subset of main memory. Whenever the CPU generates a memory request, the primary cache is checked first, and if a miss occurs there then the secondary cache is checked; only if the item is missing there too will main memory be accessed.

4.4.2 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. 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.)

4.4.3 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.