A First Look at the Intel ia32 Architecture

In this chapter, we are going to take an initial look at the architecture that Intel calls ia32. Now you may be more familiar with names like Pentium, Celeron, or Xeon, But these are names of particular chips or families of chips. The ia32 designation describes the set of instructions that is recognized by a processor. In fact there are many chips that implement very similar architectures which are all part of the ia32 family. This includes not only these Intel chips, but also chips from other manufacturers, such as AMD's Opteron chip.
All these chips implement similar enough instructions that the same machine language program can run unchanged on any one of them. On the other hand, the Mac uses a PowerPC architecture (with chip names like G4 and G5) that has a completely different set of instructions. It really does not matter what PC you buy, the same programs will run, perhaps slower, perhaps faster, depending on whether you are willing to shell out the bucks for the latest fastest chip. But if you try to run these programs on a Mac, they won't work (it is possible to buy a program for the Mac that will run them, basically by pretending to implement the ia32 instructions, we will examine that in more detail later).

Registers

But for now, back to the ia32. This architecture uses a fairly small set of registers. For now, we will concentrate on a subset of the registers that are available.

First, there are eight 32-bit integer registers, with the following (somewhat peculiar) names:

EAX
EBX
ECX
EDX
ESI
EDI
EBP
ESP

Later on we will learn more about why these registers have these funny names. For now, we should learn that ESP is the stack pointer (that's what the SP is) register, and for now, we should regard this register as being reserved and not available for use. Later on we will learn how the stack pointer register works and is used.

Another important register is the EIP, or instruction pointer register. The program never addresses this register directly, but implicitly it is the register that does the most work, since every single instruction references EIP. The value in EIP is the 32-bit address in memory of the start of the next instruction to be executed. Each instruction increments the EIP value so that the processor will execute the immediately following instruction when the execution of the current instruction is complete. Some special instructions, called jump instructions modify the value in EIP so that the next instruction is not in sequential order. We will look in more detail out how the jump instructions work later on.

Finally, there are four flag registers that contain a single bit

CF (carry flag)
OF (overflow flag)
SF (sign flag)
ZF (zero flag)

Certain instructions modify the values in these flags (the values are either 0 or 1) to record more information about the outcome of the instruction. We'll be taking a close look later on at how these flag registers are used.

Memory

The memory of an ia32 chip is arranged into 8-bit bytes. Each byte can hold an ASCII character, or an integer from 0-255, or indeed any arbitrary 8-bit pattern. Each byte in memory has an address. The address values are 32-bits long (no accident that this is the same as the length of the registers, since as we will see later we often put an address in a register). Note that the addresses are not stored as such in the memory, they are simply a method of identifying which byte you are talking about. As an analogy, if someone tells you to take the fifth turn on the right, you do not expect to see street signs saying "first turn on the right", "second turn on the right", etc. But having the number five allows you to locate the turn you need.

Similarly, a 32-bit address allows you to identify a unique byte in memory. The memory unit receives a 32-bit address and either loads or stores an 8-bit quantity in that address. The lowest address possible is all zero bits. The highest address is all one bits. This number is 2**32-1 which is about four billion, which is appropriate for addressing up to four gigabytes of memory. If you want to put more than four gigabytes of memory on a PC you are out of luck. More likely you have less memory than this, for example, perhaps only 512 megabytes. In that case, some of the memory address values may be unused.

A little digression here. Four gigabytes sounds like a lot. You might be tempted to think "no one could possibly need more than four gigabytes of memory". If you do think this, then you are in good company. The history of computing over the last fifty years records many instances of this kind of statement with steadily increasing values. For example, the 8080 chip had a maximum of 64K memory. When Intel designed the 8086 successor chip (which became the chip used on early PC's), Intel told the designer "well 64K is enough for anyone, but to be safe, please design the new chip to take a maximum of 128K .. no one could need more than 128K. The designer, Steve Morse, was suspicious of this, and he designed the chip to allow a maximum of one megabyte ("no one could need more than one megabyte). All these statements have proved wrong, and indeed the latest development is that four gigabytes is not enough for some applications, and 64-bit chips are now beginning to appear (no one could possibly need more than 16 exabytes of memory ☺).

Layout of Data

Now 8-bits is not very much. It would be uncomfortable to be limited to 255 pieces of gold in a typical video game, and anyone wanting to store the national dept as an integral number of dollars would be out of luck trying to store the value in one byte.

To get around this, many kinds of data that we store will take more than one byte. In particular, a 32-bit unsigned value (the kind of data that corresponds to the unsigned declaration in C), requires four consecutive bytes in memory. As we discussed in the previous chapter, there are two ways for laying out data. On a “big-endian” machine, the most significant byte is first, and on a “little-endan” machine, the least significant byte is first.

The ia32 is a little-endian architecture. When a chip is designed the choice between these two possibilities is pretty much arbitrary. In the case of the ia32, the decision was forced by considerations of compatibility, since all previous Intel chips are also little-endian, and Intel did not want to have incompatible data layouts between different chips, because otherwise transition from one chip to another would be more complicated. If you trace the history of Intel chips (ww.i-probe.com/i-probe/ip_intel.html has a nice account of this history), the very original chip was the 4004. The 4004 was little-endian, because it was the product of a research project which aimed to show that a single chip could duplicate the capabilities of an existing computer. The computer chosen happened to be little-endian, and the reason for that was interesting. This was from the early days of computing (over forty years ago), and the machine they were copying had delay line memories. This memory technology basically stores data on a rotary electronic device very much like a hard disk, in that you have to wait for the data stream to rotate till you can access the byte you want. When you are doing multi-byte additions, it is convenient to access the least significant byte first, so that you can propagate carries to more significant bytes. This makes a little endian arrangement more efficient, since otherwise you would have to wait a rotation between bytes. Of course this consideration is irrelevant for a modern RAM-based memory, but it is always interesting to see how historical considerations of compatibility can cause old decisions to stretch far into the future.

Another excursion on the subject of historical compatibility. If you took a commuter train to work or school today, you rode on a set of rails that are exactly 1.44 meters apart (a bit more than 48 inches). Why this value? Well the US copied this value from the British, since both expertise and some of the early equipment came from England, and once you have chosen a particular width, you can easily see that it would be amazingly difficult to change. So why did the Brits choose 1.44 meters? The answer is that this was very close to the standard distance between the wheels on a horse drawn carriage. There was some thought early on in the history of developing rail technology that it might be useful to fit carriages with special wheels and pull them along the tracks. Of course this never happened, but at the time it seemed like a good idea. So why were the wheels of horse drawn carriages 48 inches apart? That’s a simple question to answer, the main roads in England had deep ruts this distance apart. If your carriage did not fit in the ruts, you would be in big trouble very quickly. So how did these ruts get there? The answer is that these roads were built by the Romans, and the Roman war chariots had wheels separated by this amount because it was just a perfect distance for two horses to pull one chariot. So next time you are riding a train, you can think about Roman war chariots on the one hand, and the amazing strait jacket that compatibility considerations put on technology.

Indeed, the ia32 architecture itself is an example of the power of compatibility. Any designer of computer architectures will tell you that this design is simply horrible. If it were to be turned in as a class project for a computer architecture class, the professor would conclude that the student had learned nothing in the class. As we study the ia32 architecture, we will quite often comment on how horrible it is in a number of different ways (starting with the miserably small set of registers). So was the design incompetent? Not at all, it was dictated by considerations of compatibility with previous Intel chips. Why is the chip still in such wide use if it is so horrible? That's easy to answer, the computing world has scads of software that runs on this architecture, and machines that cannot run this software have a hard time competing (Apple knows this only too well). Also, to be fair, Intel and other companies like AMD, have done a simply amazing job of building blazingly fast chips despite the difficult architectural barriers that the ia32 puts in the way of efficiency.

So, returning to the issue that sent us off on a side discussion of compatibility, the ia32 is little-endian. This means that for example, the number 258, which is 00000102 in hexadecimal, will be stored in four consecutive bytes containing (in hexadecimal)

02 01 00 00

where the 02 is in the lowest addressed memory byte, and 01 is in the next higher addressed memory byte etc. For example, the bytes might be laid out like this

address	contents
00001000	02
00001001	01
00001002	00
00001003	00

A four-byte quantity like this is always addressed using the lowest addressed byte, so in the above example, the address of the value would be 00001000. The addresses of the remaining three bytes are then found by simply taking the four consecutive byte locations starting with the given address. Two-byte and eight-byte integer values would similarly use two or eight consecutive bytes, with the same little-endian arrangement, where the least significant byte comes first.

Instruction Layout

Memory contains both data and instructions. On many well designed architectures, every instruction is a fixed length. 32-bits is a typical instruction length for a machine with 8-bit byte addressability, and indeed many common architectures, including SPARC, MIPS, and the Power PC chip found on the Mac all follow this design principle. A fixed instruction length is valuable because then you immediately know given a block of instructions where each instruction starts. As we will see later on, modern architectures often work on several instructions at the same time, and knowing how to immediately cut up a packet of instructions without looking inside the instructions is very valuable.

A recurring theme as we look at the ia32 architecture is that if there is a sound design principle that aids efficient execution, this architecture will not only fail to follow the principle, but will do so in a pretty convincing and spectacular manner. Instruction layout on the ia32 is no exception to this rule. Far from being fixed length, the instructions on the ia32 range in length from one byte to ten or more bytes. To tell how long an instruction is, you have to perform a fairly complex analysis of the bit patterns. Luckily, this won't have the slightest effect on we humans, even if we write assembly language, since the assembler is in charge of understanding these complex instruction formats. But for the writer of the assembler, and far more so, the designer of the microprocessor, this is a huge pain. We will examine later on how Intel and AMD manage to get around this thorny problem,

The EIP register always contains the lowest addressed (first) byte of the next instruction to be executed. Each instruction knows how long it is, and as it is executed, one of the actions that is taken is to add the appropriate amount to EIP. For example, if we have an instruction at location 000010AF that is five bytes long, then as part of the execution instruction, we add 5 to this value, so that the following instruction starting at 000010B4 will be the next instruction to be executed.

Setting the Stage

For now, that's a complete enough picture of the basic architecture of the ia32 in terms of register and memory layout. The big missing ingredient is what the various instructions look like and what they do. Now if we were interested in building an ia32 chip, we would be worrying about what these instructions look like at the binary bit string level. That would be hugely painful, especially on this architecture. However, for the purpose of these chapters, we are not interested in building a chip, just in programming it. Luckily the assembler will hide this detail from us, and even an assembler programmer who has written hundreds of thousands of lines of assembly code may have very little detailed knowledge of how the instructions are laid out at the bit level. The next few chapters will examine the instruction set at the assembler level. Occasionally we will give some idea of what these instructions look like as binary bit strings, but only in very broad detail.