Gordon L. Johnson
Informatics 101
11.20.06

How the Microprocessor Works

The Microprocessor: The microprocessor, (or CPU, central processing unit) is the “brain” of the computer to place it into layman's terms. Despite what type of processor you may have, albeit a Pentium, Athlon XP, or Sempron, all of them perform the same operations. The idea of a simple CPU is that it has a “complete computation engine” that is on a single chip, rather than having multiple chips to complete one action, or wired one at a time with transistors. A transistor is comprised of two diodes back to back, but what makes it so special is that it contains a central layer in between those two diodes, which enables transfer of bits, or electric current, (depending on what it may be used for). Something that holds these transistors would be a silicon chip, which can hold literally thousands. Through these transistors, the “bits” of data are sent through to another destination within the CPU itself, or exported to another station within the computer. The definition of a “bit” is a measurement of data, whether they are kilobits, megabits, etc, (which in this case are sizes of bandwidth speed through an ISP or LAN). It may also be referred to bytes as well. The very first CPU's transferred data at a blazing speed of up to 4 bits per second! This was complex enough to perform simple calculations within 15 cycles of the CPU, such as adding, subtracting, dividing and multiplying. These were eventually placed into the first portable calculators, which were quite large, and heavy. One stops to think, (if you have limited knowledge of this subject) how CPU’s might actually understand how to carry data from point A, to point B, (let alone retrieving information from the BIOS, and placing stored data into the RAM, etc). All of these processes revolve around an assembly language, which is implemented into the CPU so it can grasp what needs to occur. First, some background information about how the assembly language works, and looks like.

The Assembly Language: Every instruction within the assembly language is formed as “bitpatterns” which consist of binary, (00010001, etc). But it is obviously impossibly for humans to remember binary fluently, so we have developed short words to represent such code. Here are a few examples of such code.

“LOADA mem Load register A from memory address
LOADB mem Load register B from memory address
CONB con Load a constant value into register B
SAVEB mem Save register B to memory address
SAVEC mem Save register C to memory address
ADD Add A and B and store the result in C
SUB Subtract A and B and store the result in C
MUL Multiply A and B and store the result in C
DIV Divide A and B and store the result in C
COM Compare A and B and store the result in test
JUMP addr Jump to an address
JEQ addr Jump, if equal, to address

JNEQ addr Jump, if not equal, to address
JG addr Jump, if greater than, to address
JGE addr Jump, if greater than or equal, to address
JL addr Jump, if less than, to address
JLE addr Jump, if less than or equal, to address
STOP Stop execution”

Imagine how annoying and complex it would be to write each line of code with binary, (which is read backwards and consists of adding numbers and what not). Assembly code is fairly similar to C, (a programming language that is an overcomplicated C++ language where every single line is written out in full, while C++ makes numerous shortcuts). If you understand C, or any widely used language, then assembly language would be a cinch to understand, maybe not write, but to grasp what is trying to be executed. Here is an example of C in its most simplistic form.

```
a=1;
b=1;
while (a <= 5)
{
b = b * a;
a = a + 1;
}
```

If you do not understand the code, then allow me to explain it, roughly. A and B are variables which represent the number 1. The brackets either end each statement or the class as a whole, (depending on how many brackets there are, and what formation they may be in). The asterisk means multiply, and the parenthesis means to isolate that statement. The semicolon means the end of each given line, after its variable or whatever is given. In this case, this bit of code will execute the factorial of 5!, which means 5*4*3*2*1 = 120. When compiled, in a console, it would display 120 as the answer. This is somewhat similar to how assembly code works. To make it easier, people essentially write assembly code in C first, and then with a C compiler, it is converted into assembly language. This is quite handy, since the majorities know C, and not the native CPU coding. Plus, C can be used for almost any sort of program, whether it is GUI or not. In the following example, it is assumed that the address starts at 128 in the CPU. As an “FYI” most CPU's have their assembly language, (however it may be compiled) to start at address line 128. And of course the ROM starts at 0, since this is where the beginning instructions start.

```
// Assume a is at address 128
// Assume F is at address 129
0 CONB 1 // a=1;
1 SAVEB 128
2 CONB 1 // f=1;
3 SAVEB 129
4 LOADA 128 // if a > 5 the jump to 17
5 CONB 5
6 COM
```

```
7 JG 17
8 LOADA 129 // f=f*a;
9 LOADB 128
10 MUL
11 SAVEC 129
12 LOADA 128 // a=a+1;
13 CONB 1
14 ADD
15 SAVEC 128
16 JUMP 4 // loop back to if
17 STOP
```

You may use the prior given table that explains each command of the assembly language to make more sense of this. I did not explain though, that every single time you see double slashes, (//) it means a comment is made. Such as line 8 of the assembly code means that line 129 of the CPU is variable f=f times a, end comment/statement, (;). If you were to view this code in the ROM, then it would be viewed as binary, such as the following table.

```
LOADA – 0100110001001111010000010100010001000001
LOADB – 0100110001001111010000010100010001000010
CONB – 01000011010011110100111001000010
SAVEB 0101001101000001010101100100010101000010
Line 128 001100010011001000111000
```

And for the sake of repetition, here is an example of what it would look like, (somewhat) in ROM.

```
0 01000011010011110100111001000010 // CONB 1
1 0100110001001111010000010100010001000001
2 0101001101000001010101100100010101000010 // SAVEB 128
3 001100010011001000111000
4 01000011010011110100111001000010 // CONB 1
```

For the sake of simplicity, not everything was written in binary, but the point is made. As you can see, bytes of code in C, when converted to assembly code in ROM can nearly double in length of bytes.

Assembly language may be one of the most basic languages, but at the same time is the most powerful language in making your computer run faster, and more efficiently. A CPU is given a set of instructions every instant to tell it what to perform next, such as each letter that is being typed at this moment is being copied over into a temp folder since it is currently not being saved at the moment. Now if I were to save this document by holding ctrl +s, I would be giving the CPU an instruction to redirect the new information to my file_name.doc on the hard drive disk to be overwritten with the new data. Through an instruction given to the CPU, there are three basic things that are involved. It contains an ALU, (Arithmetic/Logic Unit) that can perform basic math computations, but with "floating point ALU" it may compute highly sophisticated algorithms and what not with such technology revolving around large floating point numbers. The second element that a CPU has is the ability to move data from its memory, to another desired location. And lastly, the ability to make decisions essentially for itself, such as the desire to jump to a new set of numbers to proceed to a different set of instructions, (of course based on its original instructions given by the assembly code). In the physical sense, (rather than capabilities) a CPU has the following within: a reset, clock, read, and write line, and an address and data bus. The reset line is self-explanatory in the sense that it simply resets the action back to zero to proceed with a different action, or to start over again. A clock line is something that lets the clock "pulse sequence" the CPU. The RD and WR, (read and write consecutively) are lines in the code that allow the CPU to either read the information sent to it, or to write data to a specific place outside of the CPU. The address bus sends an address to the memory, and finally the data bus is the manager that allows data to be sent or received from the memory. The components of a very simple microprocessor are made up by the following. A few registers made up of "edge triggered latches," by using "Boolean Logic," a program counter which can either increase by one, or reset back to zero when given the proper command, an ALU (which was defined earlier), a TriState Buffer, which has the ability to either set itself to a 0, a 1, or completely disconnect itself from the output line, and finally the instruction register and decoder are held responsible for the control of all the other devices. There are many other commands that are not mentioned in this idea of a simple microprocessor, so do not think that this is simply it. The RD and RW, and the address and data buses coordinate with the RAM and ROM. These are probably the most used instructions within a microprocessor. RAM stands for random access memory, which means that this component retains bytes of information that it is sent, and chooses either to overwrite it read it, delete it, or move all of the specified bytes to a new location within the computer. Which all of this depends on which line it is given through instruction, (RD or WR). RAM, to give you an image relation would be the DDR, SDRAM, SODIMM, etc types of sticks of "memory" that are placed in a groove on a motherboard to make it easier on a CPU, so it can process things faster by loosening up the memory. The only flaw in such a device is that it contains no continuous memory, such as the clock on your desktop. Have you ever noticed that your clock remains accurate even though it is turned off for periods at a time? Imagine if the clock was stored through your RAM. If it was, then it would be reset back to the default, 12:00. This is why we have the wonderful ROM. ROM stands for read-only memory. A ROM is a chip on the motherboard that is given a preset amount of bytes that remain permanent. In this case, the address bus tells the ROM chip which byte it wants the data bus to retrieve. When dealing with a ROM chip on a PC based architecture of a CPU, it is called the BIOS, the basic input/output system. This is where all of the basic information is stored about your computer, such as the boot order of the disks, internal time, AGP aperture size, what devices are hooked up and running, CPU clock information, etc. Whenever the CPU is started up, such as each time you press the button to turn on your computer, the CPU retrieves instructions from the BIOS to explain what it needs to do on the boot up. The first instruction found in the BIOS is normally to find which boot sector to start with, within the hard drive disk. The boot sector is a program that tells where to start the HDD. After any instruction is made, it is stored within the RAM thereafter. After that, the CPU goes back to the RAM to retrieve the information on how to execute the

HDD. This is a continuing process throughout the computer, such as the aforementioned example describing my process of saving file_name.doc. This revolves around anything that you see being executed on your monitor, especially the loading of your Operating System, (such as a Linux distribution, Mac, or Windows). And technically, Windows should not be apart of this list since it does not run entirely off of the hard drive like a proper OS is supposed to run, but runs its kernel off of the RAM entirely, which is highly inefficient.

Getting back to the instruction decoder, (mentioned earlier as an essential component to the CPU) this is what is used to decode the assembly code. To essentially make sense of what it is given, from the ROM. It will be much easier to understand if we label the process of the instruction decoder in steps.

1. First clock cycle is executed; the instruction decoder first activates the TriState buffer, RD line, datain TriState buffer and “latches” the instruction to the instruction register.
2. The second clock cycle of the CPU occurs, which encompasses the decoding of the ADD instruction. It does so by adjusting the ALU to addition, and then latches the output of the ALU to the C register.
3. On the third and final clock cycle, the program counter is incremented.

In the end, every single sequence that is being run through a CPU can be laid out as methodically as the prior set of operations, but each *may* require more clock cycles. Which brings up the topic of speed with a CPU. Everyone has heard of “how fast” a certain CPU is by simply remarking that it is, oh say, 3.4 “gigahertz.” This represents the speed in clock cycles of how fast it can churn out data per second. In this case, a “gigahertz” is a unit of measurement in “gigabytes.” Such as, 1 kilobyte = 1024 bytes, 1 Megabyte = 1024 kilobytes, 1 gigabyte = 1024 megabytes, 1 terabyte = 1024 gigabytes, and so on. But of course these are just measurements of size, but the same scale, (number wise) can be used to measure the speed in hertz, such as, 3.4 GHz = 3481.6 MHz. But when in reference to clock cycles per second, a GHz = 1,000,000,000. So, in this case, a 3.4 GHz capable CPU = 3,400,000,000. This is quite astounding in comparison to the very first CPU in 1974 by Intel where it could do basic arithmetic that took roughly 15 cycles per calculation! But the true definition of a “hertz” just to get technical, (maybe this will help your understanding) is an indicator of frequency of UHF, (ultrahigh frequency) and microwave EM signals. The wavelength of 1 GHz has a length of nearly 300 millimeters, which is roughly a little less than a foot. Since we are on the topic of pure “performance” of the CPU, like stated earlier, not only is the assembly code important within this realm, so is the number of transistors. The numbers of transistors make it easier and faster to transport data more effectively. The more transistors, the more multipliers can be assigned, which means the higher the clock rate. Such as, 9 multipliers * the FSB (front side bus) of 250 MHz would equal 2250 MHz CPU clock speed, or 2.25 GHz. Also, the presence of more transistors also opens up the door to pipelining. In pipelining, multiple processes can execute over each other in 5 clock cycles, so it appears as though one execution is occurring per one clock cycle, therefore enhancing the CPU's performance.

The New Wave – 64bit Processing: 64bit processors have been around since 1991, and seem to be reappearing now with their 64bit technology perfected. Most CPU's consist of 32-bit ALU, where now the trend is moving much more rapidly towards 64bit ALU's. This would obviously increase computing to such a new level, but seems to be utterly useless, (meaning no difference would be noticed, except if your clock cycles increased) until the architecture of the CPU can be written properly for programs able to utilize this new coding. Then a difference will be seen. Until that point in time, all will remain about the same clock rate wise. Also, within the past 10 years, new performance boosters have been added, such as L1, and L2 cache devices which increase performance, as well as special instructions such as 3d Now! and MMX, etc. which make 3d programs excel, and so on. Besides the point of enhanced 64bit ALU structures, is the increased amount of space in the address spaces. Which, from reading what an address space is earlier in this document explains why it is such a plus. And, of course, this cannot be used until programmers begin to realize that writing programs for 64bit architecture will enhance performance dramatically, (if the person has a 64bit operating system along with a 64bit CPU). One other plus with the 64bit architecture is that it can support, (in theory) an infinite amount of RAM, which is wonderful for those servers that require large amounts of RAM that support heavy-duty game/HTTP servers.

Dual Core (or Duo Core): Dual core CPU's, is having essentially two CPU's in one, thus doubling everything you can imagine; process wise of course. Such as, double the L1 and L2 cache, L1 now is 128*2 and L2 cache is now 512*2. The FSB is double, and so on.

How **does Dual Core (or multiple core technology in general) work?** After taking all of the prior knowledge into consideration, basically what is happening when a Dual or Multi-Core CPU is put together, two lower-graded CPU's are placed together on a single die, thus improving the overall performance of the CPU. Since the majority of applications run as single threads, they do not utilize the multi-core feature yet. Each CPU can be thought up as a master thread, and runs processes on each CPU. The operating system may run on one thread, while running parallel with the other thread, which may be devoted to a game, or some other application. In certain scenarios, one may think of it as having a separate CPU for your operating system to execute all of its commands, and another standard CPU to direct your single threaded game/application. For the time being, there are not many, (if any) applications that are double or quadruple threaded, thus not enabling full utilization of the processing power. For the time being, its almost as if you are given extra physical memory, such as doubling your RAM. As of now, it is virtually pointless to have multiple cores for the average end-user.

Much debate has been brought up in regards to if more CPU's in one is actually necessary for the everyday end-user. It is completely understandable for large server farms such as Google or Pixar to possess such technology, considering the fact that they both need to produce massive amounts of data in a limited amount of time, (upstream or in Pixar's case, animating 3 dimensional models). Ever since the release of the Core 2 Quadro, (quad core microprocessor on a single die made by the infamous Intel, many articles have been produced arguing over the idea that multiple cores are becoming almost absurd. Of course the idea of a faster processor is necessary; it would be stating the same question back when 286's were the "hot new thing" out on the market. But the major problem is not the idea if we *need* this particular piece of hardware in our computers, its whether or not we need to apply capital to CPU's of such stature *if* applications are not utilizing them yet. 64-bit processors alone have not been harnessed yet towards today's applications and they have been around since 1991, developed by MIPS Technologies, used for SGI graphics workstations. For a 64-bit application to function properly, it must have a couple of obvious requirements; the proper CPU architecture, (in this case, for AMD, x64 for example) and an operating system that is written with 64-bit CPU system calls so that the CPU can interpret the extended room in the address spaces. Then, the application may be executed within this environment. After 15 years between the first development of the 64-bit single core CPU, one would think that some progress on the Microsoft end, (application wise) would have more support for 64-bit applications, drivers, and so on. From what I ascertain, not much progress has been made towards making more 64-bit applications, albeit Linux or Windows based. Hardly anyone remembers the Windows x64 XP Pro. Why? Because it was a complete and utter failure, almost false advertising. It was one of Microsoft's largest flops since Microsoft Windows ME, where it generated a blue screen error on debut. It was almost as if Microsoft swept it under the rug, gave no support whatsoever, (especially plug and play driver wise) and failed to interpret 32-bit utilities, (such as Sygate firewall, SAV, etc). The WOW32.exe emulator is short for Windows-on-Windows emulator that supposedly interprets all mixed bit software. This is the one and only component that is used when interpreting the prior stated applications; it was poorly designed and barely ran many of the most commonly used applications. This is one of the many reasons why Microsoft chose to ignore that the operating system ever existed, and lasted on the shelves for only a few months. Technology is supposedly a huge part of the industry's agenda, which it is, only to a certain extent. There is a major lacking in support, which is always overlooked by any major software industry. The prior stated reasons are why 64-bit support is lacking quite a bit, since Microsoft, sad as this may be, is the leading company that sets software manufacturer's standards. If Microsoft falls behind, so does all progress in programming for 64-bit applications, and utilization of multiple core CPU's on a single die. Making reference to the problem of lack of support for multi-core CPU on a single die, certain software cannot interpret the CPU calls as well as it should. For example, on multiple forums, when particular 3-dimensional games are being played in a 32-bit environment, the game is literally sped up beyond its normal use. This is because the program is having a difficult time interpreting the system calls, thus resulting in error. This is one of many examples why the technology is not necessarily ready to be used by the end-user, and that more development needs to occur. How can we use more multi-core CPU's if the software industry is not using it properly yet? One metaphor that comes to mind is the development of yet another bridge connecting Kentucky to Indiana, however; the local government is incapable of maintaining the safety and aesthetics of their current bridges.

As for Microsoft Window's Vista, supposedly, (and this is a large supposedly, considering I have run many benchmarks all throughout Longhorn 2 years ago, to Vista Beta 2, not noticing much utilization between 32 and 64-bit versions) the final release version will have better performance on the 64-bit version; obviously making absolutely no recognition of their first failure, Windows XP Pro x64. As Brookwood so eloquently states, "Vista will certainly take advantage of 64-bit code in ways that Windows XP does not. The belief is that 32-bit programs running on Windows Vista won't run as fast as their 64-bit equivalents." From viewing multiple benchmarks online, Windows Vista's latest 64-bit RC2 that is available for download does a worse job than Windows XP Pro, thus far. It has come to my attention that if a 64-bit version of this particular operating system in its current state as of now requires more CPU power to run the operating system alone, because of the staggering amounts of "eye-candy." It would be more worthwhile to have an impressive system with 32-bit environment software wise than to throw the majority of your CPU power away just to acquire some visuals that will never thank you, but only hog your system resources. Here are some benchmarks that display Windows Vista RC2 vs. the current Windows XP Pro.

iTunes encoding test
(Shorter bars indicate better performance)

Windows Vista beta 2 (build 5384)
246
Windows XP Pro SP2
247
Note: *Time in seconds*

Photoshop CS2 image-processing test
(Shorter bars indicate better performance)

Windows Vista beta 2 (build 5384)
378
Windows XP Pro SP2
400
Note: *Time in seconds*

3D games testing: F.E.A.R.
(Longer bars indicate better performance)
1,600x1,20 1,024x76
0 8
Windows Vista beta 2 (build 5384)
13
30
Windows XP Pro SP2
15
29
Note: *Scores in frames per second*

The prior tests were taken from CNET's website http://reviews.cnet.com/4531-10921_7-6543881.html?tag=blog. The set up is as follows; "We loaded Windows Vista beta 2 (build 5384) and Windows XP Professional SP2 on a 3.2GHz Pentium 4, with 1GB of DDR2 memory running at 664MHz and an ATI Radeon X850 XT graphics card."

It would be astounding if Microsoft made any more tweaks to the kernel so that it may interpret the 64-bit CPU system calls so that it may out-perform the legacy operating system. Since the release date is January of next year, this is *highly* unlikely. The only corrections that could conceivably be taking place would be corrections to the page-file "hack" patch, (http://theinvisiblethings.blogspot.com/2006/10/vista-rc2-vs-pagefile-attack-and-some.html) that made the operating system "Vista kernel fix, worse than useless," as stated by *The Register*, (please see link: http://www.theregister.co.uk/2006/10/24/vista_kernel_fix_controversy/). It seems as though Microsoft has more problems on their hands at the moment, as opposed to tweaking the kernel much more; now the problem is to simply keep it afloat.

In summary, development and utilization of technology essentially lies within the hands of the software engineers, and if the demand is prevalent. Just as the acquisition of non-oil gasoline is available, but there is not much of a demand yet considering the current condition of the more than abundant resources available. In a somewhat near future, multiple-core CPU's will be in high demand and single-cell CPU's will eventually fade out, and 64-bit will be the mainstream format of computer software; but for now, time is the largest factor, as well as demand.