CS 101, Mock
Computer Architecture

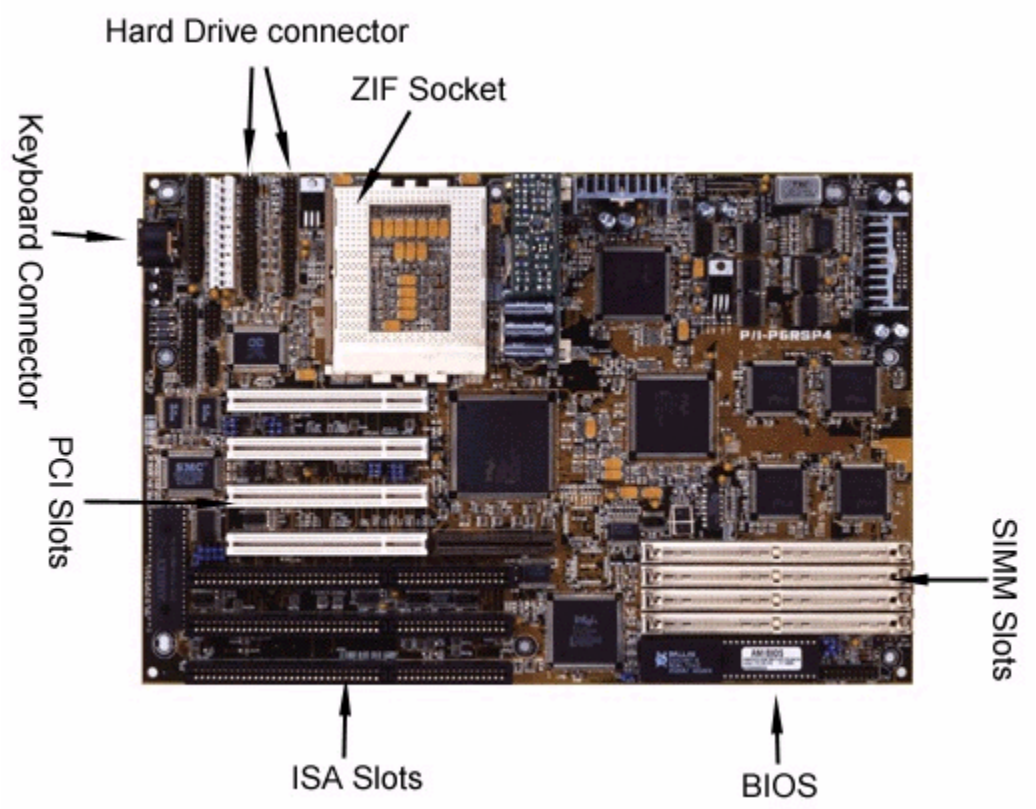
Computer organization and architecture refers to the actual hardware used to construct the computer, and the way that the hardware operates both physically and logically (logically is what the hardware does when executed, physically is how everything fits together and what is connected to what).

Computer architecture is discussed in chapter 5 of the textbook. We will also be presenting a simple, hypothetical machine architecture in these notes.

Under The Cover

Opening up the cover of a computer reveals the major physical (but not really the logical) components. Immediately you should see your power supply, floppy or hard drives, and then a motherboard containing the Central Processing Unit (CPU), main memory, and then any extra circuit boards inserted into slots on the motherboard.

The guts of the computer is on the motherboard, so let's focus on that:



Motherboards of today include a lot of functionality that previously was included as an expansion card. For example, chips for sound input and output, chips for interfacing with other devices (e.g., USB, universal serial bus), and connections for mice and the keyboard and even network connections are now standard. Nevertheless, the

motherboard is designed to accommodate the addition of hardware. You can add additional RAM using the SIMM (Single Inline Memory Module) slots, although many systems now use the DIMM format instead (Dual Inline Memory Module). These are memory modules mounted on small circuit boards that you can just snap into the slots to add new memory.

The ISA and PCI slots interface peripheral cards with the computer's bus. We'll say more about the bus later, but it is a way to connect devices (e.g., the CPU and a peripheral card).

The motherboard also contains connectors to attach storage devices, such as hard drives or CD-ROMs, along with ROM and BIOS chips for initial bootup.

Finally, the motherboard has either a socket or a slot for the CPU itself. The CPU will likely have a fan on it in order to dissipate some of the heat it generates.

Digital Data Representation and Memory Organization

Recall that everything in the computer is stored as 1's or 0's. Numbers might be represented in unsigned binary, or perhaps in the IEEE 754 format if the number is a floating point number. Characters might be stored in the ASCII code. All of these are represented as the 1's and 0's and stored in RAM. The actual memory chips contain transistors that can store the 1's and 0's as electrical charges.

How do we store data in RAM? To control where data goes, we need to have an **address** for memory. Typically each memory address will reference 8 bits, or 1 byte. For example, let's say that we have a very simple memory system with 16 addresses:

Memory Address	Memory Contents
0	00000000
1	00000001
2	01000001
3	01000010
4	…
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	
15	

I've left some data in the contents of the memory addresses. In memory address 0, we have the byte 00000000. As shown here, this is just some raw data. But if we wanted to treat these 8 bits like a number, this would represent the number 0. Similarly, the contents of memory address 1 contains the data 00000001. If treated as a number, this could contain the value 1. Memory address 2 contains the data 01000001. If we treat this as a number, it is 65. However, we might want to treat it as an ASCII character. If you look at the ASCII chart, this code corresponds to the letter 'A'.

What might be in memory address 3? It could be the ASCII code for 'B', the number 66, or something else! It is going to be up to our program to correctly interpret what data is stored in memory.

Von Neumann Architecture

All of today's computers, despite their many differences, are based on the design of John Von Neumann (pronounced as: von noimän). His design was proposed in 1946 and the first computer, the EDVAC, was built according to his design in 1950. The main ideas of the von Neumann architecture are:

- Four major subsystems: Memory, Input/Output (I/O), Arithmetic and Logic Unit (ALU), and the Control Unit (CU). We'll cover these in more detail coming up.

- Stored Program Concept. This is the idea that a program can be stored in memory, just like data is stored in memory. This idea might sound obvious since you load programs in memory all the time today, but prior to this concept computers were programmed by hard-wiring them! That is, to "load" a new program one had to manually wire and set switches to get the desired behavior.

- Sequential program execution. This concept goes along with the stored program concept. A program is the sequential execution of instructions we will store in memory.

Memory-Mapped I/O

A common usage of memory is to "map" a portion of memory to some I/O device. For example, memory address 15 above might really correspond to the printer's serial port. By storing something into memory address 15, we could be sending data to the printer to be printed. This concept is typically used for many input and output devices, such as the monitor and the graphics screen.

Moving Data Around

We'll want to store different values in memory and operate on them. The way we will generally do this is to move a value from memory to the CPU, the CPU might do some

operation on it (e.g., maybe treat the data as a number and increment it or add two numbers together), and then we will store the result back into memory. To do this we need some way to move data from memory to the CPU, and vice versa.

The device used to move data is called a bus. A bus is a communication pathway connecting two or more devices. A key characteristic of a bus is that it is a shared transmission medium, so multiple devices connected to the bus have to wait until the bus is free before it can transmit.

In many cases, the bus actually consists of multiple communication pathways, or multiple wires/lines. For example, an 8 bit unit of data can be transmitted over eight bus lines.

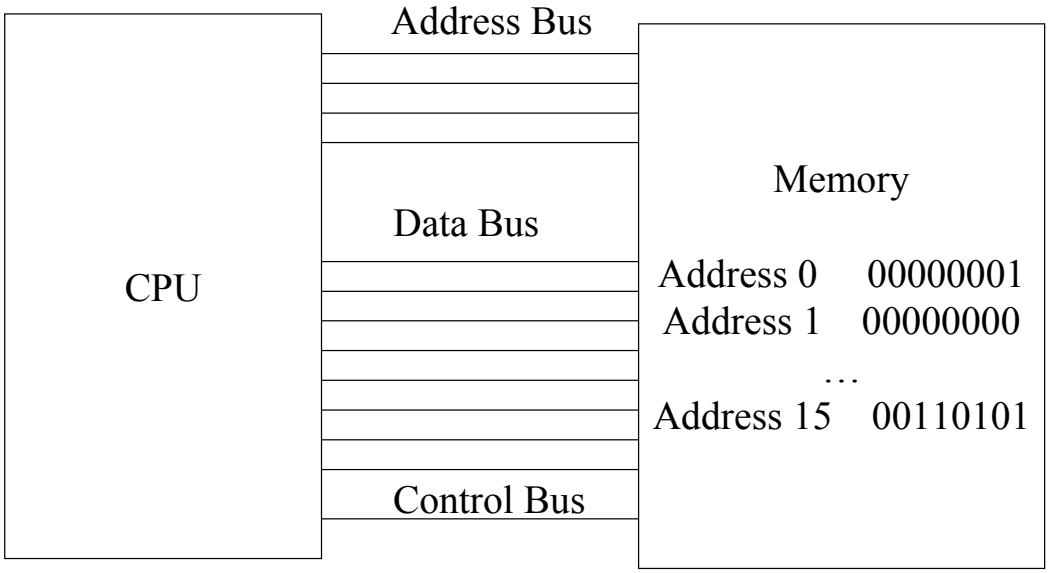
Computer systems contain many different buses to connect different components. Some of the buses we are concerned with are:

- Data Bus: connects data from RAM to the CPU. This consists of generally 8 or more lines so that we can transfer at least one byte simultaneously to the CPU. The number of lines is referred to as the bus width.

- Address Bus: connects addresses from RAM to the CPU. If we are going to store some data into memory, somehow we have to tell memory what address we want to store it into. We do this by putting the address we are storing to / reading from into the address bus.

- Control Bus : This is going to control the timing and operations on the address and data bus (e.g., execute a read or a write to memory depending on what we want).

Here is a simplified diagram of a CPU and memory:



Why did I use 4 lines for the address bus? Because there are 16 memory addresses in this system, and we need four bits to specify which one of the 16 addresses we are interested in. Each line will carry one bit, so we need four lines for the address bus.

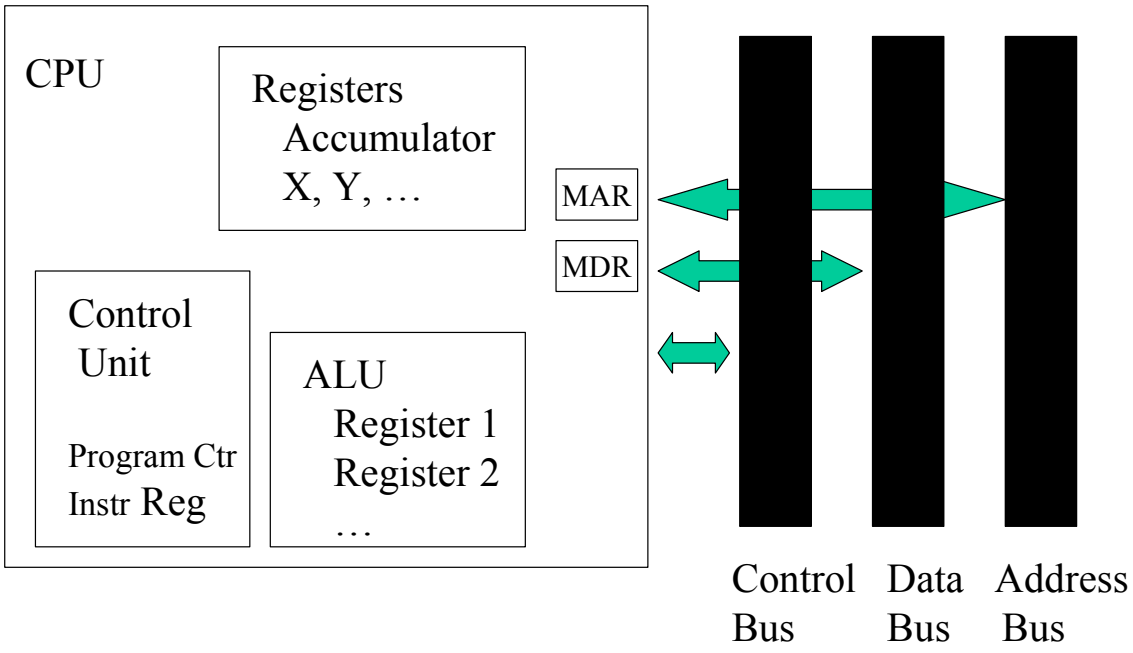
Let's say that the CPU wants to read the location of memory address 1. It would load the address it wants onto the address bus: 0001. Then it would send the proper signal on the control bus that corresponded to "read the data that's in the address bus". Main memory would then read this and load the contents of address 1 onto the data bus (00000000) and then send the proper signal on the control bus that corresponded to "send the data out on the data bus". The CPU would then be able to read the result on the data bus.

Maybe the CPU did some operation, and it now wants to store the value 00010001 back into address 1. In this case it would load the bit pattern 00010001 onto the data bus, and the address 0001 onto the address bus. When the "Store the data" signal is given on the control bus, main memory would then store 00010001 into memory address 1.

In practice, the control bus is more complex than a signal line depicted above. There are also numerous timing issues that need to be resolved, that we are skipping here!

The Central Processing Unit

So far we've been treating the CPU as a black box. Let's look inside the CPU itself now to see what is there. The major components of a CPU are depicted below:



The CPU contains the Arithmetic and Logic Unit (ALU), the Control Unit (CU), and Registers. The whole shebang interfaces with main memory through the control, data, and address bus described earlier.

The registers are minimal units of internal memory that operate very quickly. There is a special register called the *accumulator* which is typically the "main" register that is used. Other special registers include the Memory Address Register (MAR) and the Memory Data Register (MDR). The MAR holds the memory address we want to either fetch data from or store data to. Consequently, it is connected to the address bus. The MDR holds the actual data we want to write to or read from main memory. Consequently, it is connected to the data bus.

The ALU does the actual computation or processing of data when we do some type of logical operation or mathematical operation. For example, the ALU might add together the contents of two registers and store the result back in a different register. The ALU will generally contain its own set of registers in addition to the CPU's registers.

The Control unit controls the movement of data and instructions into and out of the CPU and also controls the operation of the ALU. Once piece of the CU is a special register called the Instruction Pointer. This is a pointer to the memory address of the current instruction in the program we are now running. The Instruction register contains the actual instruction we are to execute (e.g., add, subtract…) Note: The names of these registers vary on different computer architectures!

Let's say that we want to add two numbers together. First, the CPU must fetch these two numbers from main memory using the process we described above. One of these numbers might go into register X, and the other into the Accumulator. Next, the contents of register X and the Accumulator can be sent into registers in the ALU. The ALU then performs the addition operation, and stores the result in the Accumulator. Finally, the value in the Accumulator is stored back into main memory. All of these steps are controlled by the Control Unit!

Instructions

How does the Control Unit know if it should add, subtract, fetch, store, or what? Your program controls all of this! The control unit contains a Program Counter and an Instruction Register. The Program Counter is the address of the current instruction we are executing in our program. The Instruction Register contains the actual instruction itself.

Recall that in the Von Neumann architecture, main memory contains data. This data will store both the instructions for the program, and the data the program operates on. For example let's say that our simple computer memory contains the following:

Memory Address	Memory Contents	
0	00000010	
1	00000011	
2	00000010	
3	00000000	
4	00000000	
5	00000000	
6	00000000	
7	00110000	; Instruction for LOAD Accumulator
8	00000000	; with memory address 0
9	00110001	; Instruction for LOAD X Register
10	00000001	; with memory address 1
11	10101010	; Instruction for Multiply Acc, X
12	11101010	; Instruction for Store Accumulator
13	00000000	; into memory address 0
14	11111110	; Jump to memory location
15	00001111	; jump destination is address 7

Here, mixed in with our memory is data that our program will operate on, along with instructions for the computer. The instructions are coded as sequences of bits as defined by the designer of the CPU. In this example, the instructions begin at memory location 7. The bit pattern 00110000 might stand for "LOAD the Accumulator with the data in the following memory address". The value stored in the next memory slot holds the memory address to load. Similarly, the bit pattern 00110001 might stand for "LOAD the X Register", and some other pattern might correspond with Multiple, Divide, Add, etc. Each of these instructions is called an *op code*, and the data being operated on is the *operand*.

If we program directly using these 1's and 0's then we are programming in *machine code*. It is inconvenient to program in machine code, so one level up is to assign short codes or *mnemonics* to each sequence of bits.

For example, the mnemonic of LDA could correspond to the machine code 00110000 for "LoaDing the Accumulator". MUL could be the mnemonic that corresponds to the machine code 10101010 to MULtiply, etc. It is important to remember that these mnemonics correspond to patterns of bits; the English mnemonics are there just to make it easier for people to understand.

The complete list of instructions that the CPU is able to execute is called the *instruction set*. A small sample of some common instructions with hypothetic mnemonics are listed below:

Op Code	Operation	Example
CLA	Clear the accumulator to 0	CLA
MAM	Move Accumulator to memory	MAM 4
MMA	Move memory to Accumulator	MMA 6
ADD	Add two registers, result in Acc	ADD
SUB	Subtract, result in Accumulator	SUB
INC	Increment the register value by 1	INC
CMP	Compare registers; if equal,	
	Put 1 in the Accumulator, else 0	CMP REG1 REG2
JMP	Branch to a new memory location	JMP 5
JPZ	Jump if accumulator is zero	JPZ 5

Instruction Cycle

Let's actually simulate running through the sequence of instructions shown above. The process that the computer follows is to:

1. Fetch the instruction located at the address stored in the Program Counter register, and put its contents into the Instruction Register.

2. Decode the instruction (figure out what it is)

3. Execute the instruction.

4. Increment the contents of the instruction pointer, and go back to step 1.

Let's say that we begin our process with the Program Counter containing the number 7, the Accumulator contains 0, and the X register contains 0:

PC = 7, Acc = 0, X = 0

In our example, we will first fetch the contents of memory address 7 and put it into the Instruction Register. The Instruction Register now holds 00110000. In decoding this instruction, the Control Unit learns that this is the "Load" operation and that it has one operand. Therefore we need to fetch the operand from memory. To do this we can increment the Program Counter to 8, and fetch its contents into the Instruction Register. We now have all the information we need, and can execute the instruction: Load the contents of memory location 0 into the accumulator (00000010 or the number 2). After this is done, the Program Counter is incremented by one and we go on to the next instruction. Our three registers now contains:

PC = 9, Acc = 2, X = 0

In this case, the next instruction performs a similar process but with memory location 1 into register X (00000011 or the number 3). The registers now contain:

PC = 11, Acc = 2, X = 3

The next instruction is a multiply instruction, which will cause the CU to store the contents of the two registers into the ALU, and then direct the ALU to multiply the values together. Typically the results of these operations will be stored back into the Accumulator. Note that this instruction does not have an operand, so we don't need to make another trip to memory in decoding or executing the instruction. After we have stored the result of the multiplication back into the accumulator, the registers contain:

PC = 12, Acc = 6, X = 3

The next instruction is a store instruction, which will cause the CPU to store the results of the accumulator into memory address 0. The value 00000010 in memory address 0 will be overwritten with the number 6, or 00000110 in binary.

Finally, we fetch the last instruction, which is a JUMP instruction to memory address 7. Once these instructions have been loaded, this instruction is executed by changing the contents of the Program Counter to 7. This will then cause the computer to start executing the instructions at memory location 7, which is where we just started! This program is an infinite loop which keeps multiplying memory location 1 to memory location 0 and storing the results back to memory location 0. (Note after the Jump we have to not increment the Program Counter; alternately we could have jumped to location 6 and then incremented the Program Counter).

Increasing CPU Performance

There are several ways we can speed up operation of the computer. The simplest way is to increase the clock rate. The CPU operates off a clock, e.g. many computers today operate at 1.5 Ghz or 1.5 billion cycles per second. Each cycle corresponds to a period of time where the computer can do some work – such as transfer data among registers or add numbers together. If we can increase this rate, then we will perform all of our instructions faster.

However, increasing clock rate is a bit misleading – even if the CPU operates faster, this doesn't mean memory operates any faster! The computer is only as fast as its weakest link. Consequently, the speed of your disk, RAM, and other devices may limit actual performance.

Word size refers to the number of bits that the CPU can manipulate at once. Older computers operated on 8 bits at once, but today's computers operate on 32 bits. The Alpha and the Itanium operate on 64 bits. In general, processing more bits at once contributes to increased performance.

CPU Cache (pronounced cash) refers to a special high-speed memory that exists between main memory and the CPU. The cache will temporarily store instructions and data. It is much faster to retrieve data from the cache than it is to retrieve data from main memory

because the cache is constructed using high-speed transistors that are much faster than the way main RAM is created. We could use this high-speed memory in main memory RAM, but the computer would then be incredibly expensive. When the CPU needs to get something from memory, it first checks the cache. If the data we want is in the cache, then the value is used without getting it from main memory. If the data is not in the cache, it is retrieved from main memory and also stored in the cache. Since the cache is much smaller than main memory, when something new is added to the cache, something old needs to be thrown out! There are many subtleties to caching that are discussed in more detail in a computer architecture course, but caching is responsible for a great increase in computing performance.

RISC vs. CISC

Another way to increase CPU performance is to tinker with the instruction set. The original strategy that was pursued in CPU design was the CISC design, or Complex Instruction Set Computer. In the CISC design, lots of instructions were added to the instruction set. For example, we could have instructions that do relatively complex tasks, such as zeroing out portions of memory, copying blocks of memory, or even sorting bytes of data. While these instructions will run quickly, the problem is that most of these instructions are not used. However, a large amount of circuitry needs to be added to support these functions. Not only is this expensive, but all the extra circuitry actually slows down the basic computing cycle of the CPU! That is, the more instructions we add, the slower all instructions get.

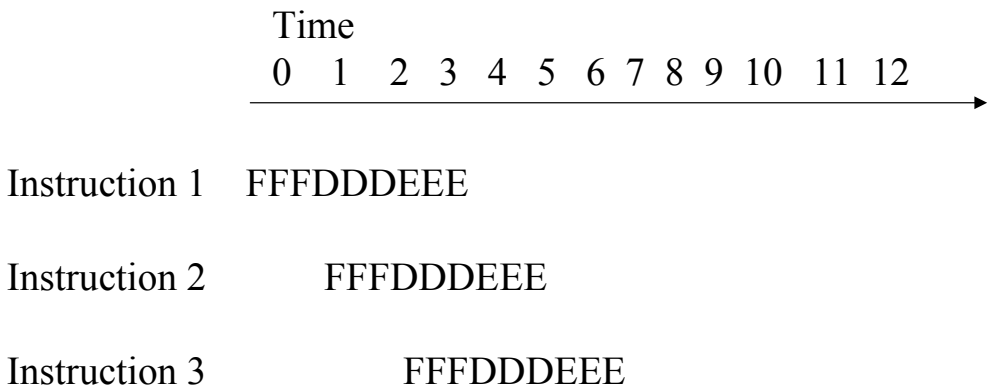
The alternate approach is the RISC design, or Reduced Instruction Set Computer. In RISC computers, the idea is to only supply the most primitive instructions needed to get the job done. Since there are not many instructions, all of them will run very fast. According to the theory, these instructions will also be the ones that are executed most frequently, resulting in overall faster CPU execution.

Many computers today actually strike a balance between both approaches.

Pipelining

Another approach that is implemented in virtually all computers is pipelining. If we look at the instruction cycle, we 1) fetch, 2) decode, and 3) execute the instruction. Each instruction we process has to go through all three of these stages.

However, why not start fetching the next instruction while we're decoding the current instruction? Similarly, why not decode the next instruction while we're executing the current instruction? This means that the next instruction will be ready to execute once we're done executing the current instruction. This process is shown in the picture below:



The pipelining process allows us to overlap the fetching and execution of each instruction. The result is that we can execute instruction one after the other, much like factory workers on an assembly line. In practice, pipelines are often many stages deep (7-10) stages.

Potential problem: What if Instruction 1 is a JPZ (Jump if Zero, or conditional branch) instruction? Then if we take the branch, instructions we have already fetched into the pipeline won't be used and have to be discarded, resulting in lower performance.

Parallel Processing

Finally, another way to increase CPU performance is to simply add more CPU's. If possible, these CPU's will operate at the same time on the data.

There are two general types of parallel processors:

SIMD = Single Instruction, Multiple Data. A SIMD machine applies a single instruction to lots of data. For example, it could add the number 1 to ten other numbers all simultaneously.

MIMD = Multiple Instruction, Multiple Data. A MIMD machine applies different instructions to different data. For example, it could be adding the number 1 to one memory location, while multiplying the number 5 to another memory location simultaneously.

Both types are applicable in certain situations and require careful design and setup to be effective. For example, many matrix operations can be efficiently implemented using SIMD instructions.

The Intel Pentium processors are uniprocessor machines, but actually have some SIMD instruction capabilities.

The Pentium II includes “MMX” capabilities, which allow for SIMD instructions on two 32 bit integers or on four 16 bit integers. As a simple example, maybe we want to perform a “fade” effect for the graphics screen. This might mean taking the brightness of each pixel on the screen, subtracting it, storing it back, and repeating the process until the brightness of all pixels is zero. Instead of doing this operation on one pixel at a time, we could use MMX and do it on four pixels at one time, increasing the speed of the operation.

The Pentium III includes “SSE” capabilities (Streaming SIMD Extensions) which allows for SIMD instructions on four 32 bit floats stored in the IEEE 754 format. This has the potential to increase the speed on compute-intensive tasks, but programs must be carefully written to take organize data in such a way as to take advantage of these types of instructions.