4.1 Chapter Overview

This chapter discusses history of the 80x86 CPU family and the major improvements occuring along the line. The historical background will help you better understand the design compromises they made as well as understand the legacy issues surrounding the CPU s design. This chapter also discusses the major advances in computer architecture that Intel employed while improving the x86[1].

4.2 The History of the 80x86 CPU Family

Intel developed and delivered the first commercially viable microprocessor way back in the early 1970 s: the 4004 and 4040 devices. These four-bit microprocessors, intended for use in calculators, had very little power. Nevertheless, they demonstrated the future potential of the microprocessor — an entire CPU on a single piece of silicon[2]. Intel rapidly followed their four-bit offerings with their 8008 and 8080 eight-bit CPUs. A small outfit in Santa Fe, New Mexico, incorporated the 8080 CPU into a box they called the Altair 8800. Although this was not the world s first "personal computer" (there were some limited distribution machines built around the 8008 prior to this), the Altair was the device that sparked the imaginations of hobbyists the world over and the personal computer revolution was born.

Intel soon had competition from Motorola, MOS Technology, and an upstart company formed by disgruntled Intel employees, Zilog. To compete, Intel produced the 8085 microprocessor. To the software engineer, the 8085 was essentially the same as the 8080. However, the 8085 had lots of hardware improvements that made it easier to design into a circuit. Unfortunately, from a software perspective the other manufacturer s offerings were better. Motorola s 6800 series was easier to program, MOS Technologies 65xx family was easier to program and very inexpensive, and Zilog s Z80 chip was upwards compatible with the 8080 with lots of additional instructions and other features. By 1978 most personal computers were using the 6502 or Z80 chips, not the Intel offerings.

Sometime between 1976 and 1978 Intel decided that they needed to leap-frog the competition and produce a 16-bit microprocessor that offered substantially more power than their competitor s eight-bit offerings. This initiative led to the design of the 8086 microprocessor. The 8086 microprocessor was not the world s first 16-bit microprocessor (there were some oddball 16-bit microprocessors prior to this point) but it was certainly the highest performance single-chip 16-bit microprocessor when it was first introduced.

During the design timeframe of the 8086 memory was very expensive. Sixteen Kilobytes of RAM was selling above $200 at the time. One problem with a 16-bit CPU is that programs tend to consume more memory than their counterparts on an eight-bit CPU. Intel, ever cogniscent of the fact that designers would reject their CPU if the total system cost was too high, made a special effort to design an instruction set that had a high memory density (that is, packed as many instructions into as little RAM as possible). Intel achieved their design goal and programs written for the 8086 were comparable in size to code running on eight-bit microprocessors. However, those design decisions still haunt us today as you ll soon see.

1. Note that Intel wasn t the inventor of most of these new technological advances. They simply duplicated research long since commercially employed by mainframe designers.
2. Prior to this point, commerical computer systems used multiple semiconductor devices to implement the CPU.

At the time Intel designed the 8086 CPU the average lifetime of a CPU was only a couple of years. Their experiences with the 4004, 4040, 8008, 8080, and 8085 taught them that designers would quickly ditch the old technology in favor of the new technology as long as the new stuff was radically better. So Intel designed the 8086 assuming that whatever compromises they made in order to achieve a high instruction density would be fixed in newer chips. Based on their experience, this was a reasonable assumption.

Intel s competitors were not standing still. Zilog created their own 16-bit processor that they called the Z8000, Motorola created the 68000, their own 16-bit processor, and National Semicondutor introduced the 16032 device (later to be renamed the 32016). The designers of these chips had different design goals than Intel. Primarily, they were more interested in providing a reasonable instruction set for programmers even if their code density wasn t anywhere near as high as the 8086. The Motorola and National offers even provided 32-bit integer registers, making programming the chips even easier. All in all, these chips were much better (from a software development standpoint) than the Intel chip.

Intel wasn t resting on its laurels with the 8086. Immediately after the release of the 8086 they created an eight-bit version, the 8088. The purpose of this chip was to reduce system cost (since a minimal system could get by with half the memory chips and cheaper peripherals since the 8088 had an eight-bit data bus). In the very early 1980 s, Intel also began work on their intended successor to the 8086 — theiAPX432 CPU. Intel fully expected the 8086 and 8088 to die away and that system designers who were creating general purpose computer systems would choose the 432 chip instead.

Then a major event occurred that would forever change history: in 1980 a small group at IBM got the go-ahead to create a "personal computer" along the likes of the Apple II and TRS-80 computers (the most popular PCs at the time). IBM s engineers probably evaluated lots of different CPUs and system designs. Ultimately, they settled on the 8088 chip. Most likely they chose this chip because they could create a minimal system with only 16 Kilobytes of RAM and a set of cheap eight-bit peripheral devices. So Intel s design goals of creating CPUs that worked well in low-cost systems landed them a very big "design win" from IBM.

Intel was still hard at work on the (ill-fated) iAPX432 project, but a funny thing happened — IBM PCs started selling far better than anyone had ever dreamed. As the popularity of the IBM PCs increased (and as people began "cloning" the PC), lots of software developers began writing software for the 8088 (and 8086) CPU, mostly in assembly language. In the meantime, Intel was pushing their iAPX432 with the Ada programming language (which was supposed to be the next big thing after Pascal, a popular language at the time). Unfortunately for Intel, no one was interested in the 432. Their PC software, written mostly in assembly language wouldn t run on the 432 and the 432 was notoriously slow. It took a while, but the iAPX432 project eventually died off completely and remains a black spot on Intel s record to this day.

Intel wasn t sitting pretty on the 8086 and 8088 CPUs, however. In the late 1970 s and early 1980 s they developed the 80186 and 80188 CPUs. These CPUs, unlike their previous CPU offerings, were fully upwards compatible with the 8086 and 8088 CPUs. In the past, whenever Intel produced a new CPU it did not necessarily run the programs written for the previous processors. For example, the 8086 did not run 8080 software and the 8080 did not run 4040 software. Intel, recognizing that there was a tremendous investment in 8086 software, decided to create an upgrade to the 8086 that was superior (both in terms of hardware capability and with respect to the software it would execute). Although the 80186 did not find its way into many PCs, it was a very popular chip in embedded applications (i.e., non-computer devices that use a CPU to control their functions). Indeed, variants of the 80186 are in common use even today.

The unexpected popularity of the IBM PC created a problem for Intel. This popularity obliterated the assumption that designers would be willing to switch to a better chip when such a chip arrived, even if it meant rewriting their software. Unfortunately, IBM and tens of thousands of software developers weren t willing to do this to make life easy for Intel. They wanted to stick with the 8086 software they d written but they also wanted something a little better than the 8086. If they were going to be forced into jumping ship to a new CPU, the Motorola, Zilog, and National offerings were starting to look pretty good. So Intel did something that saved their

bacon and has infuriated computer architects ever since: they started creating upwards compatible CPUs that continued to execute programs written for previous members of their growing CPU family while adding new features.

As noted earlier, memory was very expensive when Intel first designed the 8086 CPU. At that time, computer systems with a megabyte of memory usually cost megabucks. Intel was expecting a typical computer system employing the 8086 to have somewhere between 4 Kilobytes and 64 Kilobytes of memory. So when they designed in a one megabyte limitation, they figured no one would ever install that much memory in a system. Of course, by 1983 people were still using 8086 and 8088 CPUs in their systems and memory prices had dropped to the point where it was very common to install 640 Kilobytes of memory on a PC (the IBM PC design effectively limited the amount of RAM to 640 Kilobytes even though the 8086 was capable of addressing one megabyte). By this time software developers were starting to write more sophisticated programs and users were starting to use these programs in more sophisticated ways. The bottom line was that everyone was bumping up against the one megabyte limit of the 8086. Despite the investment in existing software, Intel was about to lose their cash cow if they didn t do something about the memory addressing limitations of their 8086 family (the 68000 and 32016 CPUs could address up to 16 Megbytes at the time and many system designers [e.g., Apple] were defecting to these other chips). So Intel introduced the 80286 which was a big improvement over the previous CPUs. The 80286 added lots of new instructions to make programming a whole lot easier and they added a new "protected" mode of operation that allowed access to as much as 16 megabytes of memory. They also improved the internal operation of the CPU and bumped up the clock frequency so that the 80286 ran about 10 times faster than the 8088 in IBM PC systems.

IBM introduced the 80286 in their IBM PC/AT (AT = "advanced technology"). This change proved enourmously popular. PC/AT clones based on the 80286 started appearing everywhere and Intel s financial future was assured.

Realizing that the 80x86 (x = "", "1", or "2") family was a big money maker, Intel immediately began the process of designing new chips that continued to execute the old code while improving performance and adding new features. Intel was still playing catch-up with their competitors in the CPU arena with respect to features, but they were definitely the king of the hill with respect to CPUs installed in PCs. One significant difference between Intel s chips and many of their competitors was that their competitors (noteably Motorola and National) had a 32-bit internal architecture while the 80x86 family was stuck at 16-bits. Again, concerned that people would eventually switch to the 32-bit devices their competitors offered, Intel upgraded the 80x86 family to 32 bits by adding the 80386 to the product line.

The 80386 was truly a remarkable chip. It maintained almost complete compatibility with the previous 16-bit CPUs while fixing most of the real complaints people had with those older chips. In addition to supporting 32-bit computing, the 80386 also bumped up the maximum addressablility to four gigabytes as well as solving some problems with the "segmented" organization of the previous chips (a big complaint by software developers at the time). The 80386 also represented the most radical change to ever occur in the 80x86 family. Intel more than doubled the total number of instructions, added new memory management facilities, added hardware debugging support for software, and introduced many other features. Continuing the trend they set with the 80286, the 80386 executed instructions faster than previous generation chips, even when running at the same clock speed plus the new chip ran at a higher clock speed than the previous generation chips. Therefore, it ran existing 8088 and 80286 programs faster than on these older chips. Unfortunately, while people adopted the new chip for its higher performance, they didn t write new software to take advantage of the chip s new features. But more on that in a moment.

Although the 80386 represented the most radical change in the 80x86 architecture from the programmer s view, Intel wasn t done wringing all the performance out of the x86 family. By the time the 80386 appeared, computer architects were making a big noise about the so-called RISC (Reduced Instruction Set Computer) CPUs. While there were several advantages to these new RISC chips, a important advantage of these chips is

that they purported to execute one instruction every clock cycle. The 80386 instructions required a wildly varying number of cycles to execute ranging from a few cycles per instruction to well over a hundred. Although comparing RISC processors directly with the 80386 was dangerous (because many 80386 instructions actually did the work of two or more RISC instructions), there was a general perception that, at the same clock speed, the 80386 was slower since it executed fewer instructions in a given amount of time.

The 80486 CPU introduced two major advances in the x86 design. First, the 80486 integrated the floating point unit (or FPU) directly onto the CPU die. Prior to this point Intel supplied a separate, external, chip to provide floating point calculations (these were the 8087, 80287, and 80387 devices). By incorporating the FPU with the CPU, Intel was able to speed up floating point operations and provide this capability at a lower cost (at least on systems that required floating point arithmetic). The second major architectural advance was the use of *pipelined instruction execution*. This feature (which we will discuss in detail a little later in this chapter) allowed Intel to overlap the execution of two or more instructions. The end result of pipelining is that they effectively reduced the number of cycles each instruction required for execution. With pipelining, many of the simpler instructions had an aggregate throughput of one instruction per clock cycle (under ideal conditions) so the 80486 was able to compete with RISC chips in terms of clocks per instruction cycle.

While Intel was busy adding pipelining to their x86 family, the companies building RISC CPUs weren t standing still. To create ever faster and faster CPU offerings, RISC designers began creating *superscalar* CPUs that could actually execute more than one instruction per clock cycle. Once again, Intel s CPUs were perceived as following the leaders in terms of CPU performance. Another problem with Intel s CPU is that the integrated FPU, though faster than the earlier models, was significantly slower than the FPUs on the RISC chips. As a result, those designing high-end engineering workstations (that typically require good floating point hardware support) began using the RISC chips because they were faster than Intel s offerings.

From the programmer s perspective, there was very little difference between an 80386 with an 80387 FPU and an 80486 CPU. There were only a handful of new instructions (most of which had very little utility in standard applications) and not much in the way of other architectural features that software could use. The 80486, from the software engineer s point of view, was just a really fast 80386/80387 combination.

So Intel went back to their CAD[3] tools and began work on their next CPU. This new CPU featured a superscalar design with vastly improved floating point performance. Finally, Intel was closing in on the performance of the RISC chips. Like the 80486 before it, this new CPU added only a small number of new instructions and most of those were intended for use by operating systems, not application software.

Intel did not designate this new chip the 80586. Instead, they called it the *Pentium“ Pr ocessor*[4]. The reason they discontinued referring to processors by number and started naming them was because of confusion in the marketplace. Intel was not the only company producing x86 compatible CPUs. AMD, Cyrix, and a host of others were also building and selling these chips in direct competition with Intel. Until the 80486 came along, the internal design of the CPUs were relatively simple and even small companies could faithfully reproduce the functionality of Intel s CPUs. The 80486 was a different story altogether. This chip was quite complex and taxed the design capabilities of the smaller companies. Some companies, like AMD, actually licensed Intel s design and they were able to produce chips that were compatible with Intel s (since they were, effectively, Intel s chips). Other companies attempted to create their own version of the 80486 and fell short of the goal. Perhaps they didn t integrate an FPU or the new instructions on the 80486. Many didn t support pipelining. Some chips lacked other features found on the 80486. In fact, most of the (non-Intel) chips were really 80386 devices with some very slight improvements. Nevertheless, they called these chips 80486 CPUs.

---

3. Computer aided design.
4. Pentium Processor is a registered trademark of Intel Corporation. For legal reasons Intel could not trademark the name Pentium by itself, hence the full name of the CPU is the "Pentium Processor".

This created massive confusion in the marketplace. Prior to this, if you d purchased a computer with an 80386 chip you knew the capabilities of the CPU. All 80386 chips were equivalent. However, when the 80486 came along and you purchased a computer system with an 80486, you didn t know if you were getting an actual 80486 or a remarked 80386 CPU. To counter this, Intel began their enormously successful "Intel Inside" campaign to let people know that there was a difference between Intel CPUs and CPUs from other vendors. This marketing campaign was so successful that people began specifying Intel CPUs even though some other vendor s chips (i.e., AMD) were completely compatible.

Not wanting to repeat this problem with the 80586 generation, Intel ditched the numeric designation of their chips. They created the term "Pentium Processor" to describe their new CPU so they could trademark the name and prevent other manufacturers from using the same designation for their chip. Initially, of course, savvy computer users griped about Intel s strong-arm tactics but the average user benefited quite a bit from Intel s marketing strategy. Other manufacturers release their own 80586 chips (some even used the "586" designation), but they couldn t use the Pentium Processor name on their parts so when someone purchased a system with a Pentium in it, they knew it was going to have all the capabilities of Intel s chip since it had to be Intel s chip. This was a good thing because most of the other 586 class chips that people produced at that time were not as powerful as the Pentium.

The Pentium cemented Intel s position as champ of the personal computer. It had near RISC performance and ran tons of existing software. Only the Apple Macintosh and high-end UNIX workstations and servers went the RISC route. Together, these other machines comprised less than 10% of the total desktop computer market.

Intel still was not satisfied. They wanted to control the server market as well. So they developed the Pentium Pro CPU. The Pentium Pro had a couple of features that made it ideal for servers. Intel improved the 32-bit performance of the CPU (at the expense of its 16-bit performance), they added better support for multiprocessing to allow multiple CPUs in a system (high-end servers usually have two or more processors), and they added a handful of new instructions to improve the performance of certain instruction sequences on the pipelined architecture. Unfortunately, most application software written at the time of the Pentium Pro s release was 16-bit software which actually ran slower on the Pentium Pro than it did on a Pentium at equivalent clock frequencies. So although the Pentium Pro did wind up in a few server machines, it was never as popular as the other chips in the Intel line.

The Pentium Pro had another big strike against it: shortly after the introduction of the Pentium Pro, Intel s engineers introduced an upgrade to the standard Pentium chip, the MMX (multimedia extension) instruction set. These new instructions (nearly 60 in all) gave the Pentium additional power to handle computer video and audio applications. These extensions became popular overnight, putting the last nail in the Pentium Pro s coffin. The Pentium Pro was slower than the standard Pentium chip and slower than high-end RISC chips, so it didn t see much use.

Intel corrected the 16-bit performance in the Pentium Pro, added the MMX extensions and called the result the Pentium II[5]. The Pentium II demonstrated an interesting point. Computers had reached a point where they were powerful enough for most people s everyday activities. Prior to the introduction of the Pentium II, Intel (and most industry pundits) had assumed that people would always want more power out of their computer systems. Even if they didn t need the machines to run faster, surely the software developers would write larger (and slower) systems requiring more and more CPU power. The Pentium II proved this idea wrong. The average user needed email, word processing, Internet access, multimedia support, simple graphics editing capabilities, and a spreadsheet now and then. Most of these applications, at least as home users employed them, were fast enough on existing CPUs. The applications that were slow (e.g., Internet access) were generally beyond the control of the CPU (i.e., the modem was the bottleneck not the CPU). As a result, when Intel introduced their pricey Pen-

5. Interestingly enough, by the time the Pentium II appeared, the 16-bit efficiency was no longer a facter since most software was written as 32-bit code.

tium II CPUs, they discovered that system manufacturers started buying other people s x86 chips because they were far less expensive and quite suitable for their customer s applications. This nearly stunned Intel since it contradicted their experience up to that point.

Realizing that the competition was capturing the low-end market and stealing sales away, Intel devised a low-cost (lower performance) version of the Pentium II that they named *Celeron*[6]. The initial Celerons consisted of a Pentium II CPU without the on-board level two cache. Without the cache, the chip ran only a little bit better than half the speed of the Pentium II part. Nevertheless, the performance was comparable to other low-cost parts so Intel s fortunes improved once more.

While designing the low-end Celeron, Intel had not lost sight of the fact that they wanted to capture a chunk of the high-end workstation and server market as well. So they created a third version of the Pentium II, the Xeon Processor with improved cache and the capability of multiprocessor more than two CPUs. The Pentium II supports a two CPU multiprocessor system but it isn t easy to expand it beyond this number; the Xeon processor corrected this limitation. With the introduction of the Xeon processor (plus special versions of Unix and Windows NT), Intel finally started to make some serious inroads into the server and high-end workstation markets.

You can probably imagine what followed the Pentium II. Yep, the Pentium III. The Pentium III introduced the SIMD (pronounced SIM-DEE) extensions to the instruction set. These new instructions provided high performance floating point operations for certain types of computations that allow the Pentium III to compete with high-end RISC CPUs. The Pentium III also introduced another handful of integer instructions to aid certain applications.

With the introduction of the Pentium III, nearly all serious claims about RISC chips offering better performance were fading away. In fact, for most applications, the Intel chips were actually faster than the RISC chips available at the time. Next, of course, Intel introduced the Pentium IV chip (it was running at 2 GHz as this was being written, a much higher clock frequency than its RISC contemporaries). An interesting issues concerning the Pentium IV is that it does not execute code faster than the Pentium III when running at the same clock frequency (it runs slower, in fact). The Pentium IV makes up for this problem by executing at a much higher clock frequency than is possible with the Pentium III. One would think that Intel would soon own it all. Surely by the time of the Pentium V, the RISC competition wouldn t be a factor anymore.

There is one problem with this theory: even Intel is admiting that they ve pushed the x86 architecture about as far as they can. For nearly 20 years, computer architects have blasted Intel s architecture as being gross and bloated having to support code written for the 8086 processor way back in 1978. Indeed, Intel s design decisions (like high instruction density) that seemed so important in 1978 are holding back the CPU today. So-called "clean" designs, that don t have to support legacy applications, allow CPU designers to create high-performance CPUs with far less effort than Intel s. Worse, those decisions Intel made in the 1976-1978 time frame are beginning to catch up with them and will eventually stall further development of the CPU. Computer architects have been warning everyone about this problem for twenty years; it is a testament to Intel s design effort (and willingness to put money into R&D) that they ve taken the CPU as far as they have.

The biggest problem on the horizon is that most RISC manufacturers are now extending their architectures to 64-bits. This has two important impacts on computer systems. First, arithmetic calculations will be somewhat faster as will many internal operations and second, the CPUs will be able to directly address more than four gigabytes of main memory. This last factor is probably the most important for server and workstation systems. Already, high-end servers have more than four gigabytes installed. In the future, the ability to address more than four gigabytes of physical RAM will become essential for servers and high-end workstations. As the price of a gigabyte or more of memory drops below $100, you ll see low-end personal computers with more than four gigabytes installed. To effectively handle this kind of memory, Intel will need a 64-bit processor to compete with the RISC chips.

---

6. The term "Celeron Processor" is also an Intel trademark.

Perhaps Intel has seen the light and decided it s time to give up on the x86 architecture. Towards the middle to end of the 1990 s Intel announced that they were going to create a partnership with Hewlet-Packard to create a new 64-bit processor based around HP s PA-RISC architecture. This new 64-bit chip would execute x86 code in a special "emulation" mode and run native 64-bit code using a new instruction set. It s too early to tell if Intel will be successful with this strategy, but there are some major risks (pardon the pun) with this approach. The first such CPUs (just becoming available as this is being written) run 32-bit code far slower than the Pentium III and IV chips. Not only does the emulation of the x86 instruction set slow things down, but the clock speeds of the early CPUs are half the speed of the Pentium IVs. This is roughly the same situation Intel had with the Pentium Pro running 16-bit code slower than the Pentium. Second, the 64-bit CPUs (the IA64 family) rely heavily on compiler technology and are using a commercially untested architecture. This is similar to the situation with the iAPX432 project that failed quite miserably. Hopefully Intel knows what they re doing and ten years from now we ll all be using IA64 processors and wondering why anyone ever stuck with the IA32. On the other hand, hopefully Intel has a back-up plan in case the IA64 intiative fails.

Intel is betting that people will move to the IA64 when they need 64-bit computing capabilities. AMD, on the other hand, is betting that people would rather have a 64-bit x86 processor. Although the details are sketchy, AMD has announced that they will extend the x86 architecture to 64 bits in much the same way that Intel extend the 8086 and 80286 to 32-bits with the introduction of the the 80386 microprocessor. Only time will tell if Intel or AMD (or both) are successful with their visions.

Processor	Date of Introduction	Transistors on Chip	Maximum MIPS at Introduction[a]	Maximum Clock Frequency at Introduction[b]	On-chip Cache Memory	Maximum Addressable Memory
8086	1978	29K	0.8	8 MHz		1 MB
80286	1982	134K	2.7	12.5 MHz		16 MB
80386	1985	275K	6	20 MHz		4 GB
80486	1989	1.2M	20	25 MHz[c]	8K Level 1	4 GB
Pentium	1993	3.1M	100	60MHz	16K Level 1	4 GB
Pentium Pro	1995	5.5M	440	200 MHz	16K Level 1, 256K/ 512K Level 2	64 GB
Pentium II	1997	7M	466	266 MHz	32K Level 1, 256/ 512K Level 2	64 GB
Pentium III	1999	8.2M	1,000	500 MHz	32K Level 1, 512K Level 2	64 GB

a. By the introduction of the next generation this value was usually higher.
b. Maximum clock frequency at introduction was very limited sampling. Usually, the chips were available at the next lower clock frequency in Intel's scale. Also note that by the introduction of the next generation this value was usually much higher.
c. Shortly after the introduction of the 25MHz 80486, Intel began using "Clock doubling" techniques to run the CPU twice as fast internally as the external clock. Hence, a 50 MHz 80486 DX2 chip was really running at 25 MHz externally and 50 MHz internally. Most chips after the 80486 employ a different internal clock frequency compared to the external (or "bus") frequency.

4.3 A History of Software Development for the x86

A section on the history of software development may seem unusual in a chapter on CPU Architecture. However, the 80x86 s architecture is inexorably tied to the development of the software for this platform. Many architectural design decisions were a direct result of ensuring compatibility with existing software. So to fully understand the architecture, you must know a little bit about the history of the software that runs on the chip.

From the date of the very first working sample of the 8086 microprocessor to the latest and greatest IA-64 CPU, Intel has had an important goal: as much as possible, ensure compatibility with software written for previous generations of the processor. This mantra existed even on the first 8086, before there was a previous generation of the family. For the very first member of the family, Intel chose to include a modicum of compatibilty with their previous eight-bit microprocessor, the 8085. The 8086 was not capable of running 8085 software, but Intel designed the 8086 instruction set to provide almost a one for one mapping of 8085 instructions to 8086 instructions. This allowed 8085 software developers to easily translate their existing assembly language programs to the 8086 with very little effort (in fact, software translaters were available that did about 85% of the work for these developers).

Intel did not provide *object code compatibility*[7] with the 8085 instruction set because the design of the 8085 instruction set did not allow the expansion Intel needed for the 8086. Since there was very little software running on the 8085 that needed to run on the 8086, Intel felt that making the software developers responsible for this translation was a reasonable thing to do.

When Intel introduced the 8086 in 1978, the majority of the world s 8085 (and Z80) software was written in Microsoft s BASIC running under Digital Research s CP/M operating system. Therefore, to "port" the majority of business software (such that it existed at the time) to the 8086 really only required two things: porting the CP/M operating system (which was less than eight kilobytes long) and Microsoft s BASIC (most versions were around 16 kilobytes a the time). Porting such small programs may have seemed like a bit of work to developers of that era, but such porting is trivial compared with the situation that exists today. Anyway, as Intel expected, both Microsoft and Digital Research ported their products to the 8086 in short order so it was possible for a large percentage of the 8085 software to run on 8086 within about a year of the 8086 s introduction.

Unfortunately, there was no great rush by computer hobbyists (the computer users of that era) to switch to the 8086. About this time the Radio Shack TRS-80 and the Apple II microcomputer systems were battling for supremacy of the home computer market and no one was really making computer systems utilizing the 8086 that appealed to the mass market. Intel wasn t doing poorly with the 8086; its market share, when you compared it with the other microprocessors, was probably better than most. However, the situation certainly wasn t like it is today (circa 2001) where the 80x86 CPU family owns 85% of the general purpose computer market.

7. That is, the ability to run 8085 machine code directly.

The 8086 CPU, and it smaller sibling, the eight-bit 8088, was happily raking in its portion of the microprocessor market and Intel naturally assumed that it was time to start working on a 32-bit processor to replace the 8086 in much the same way that the 8086 replaced the eight-bit 8085. As noted earlier, this new processor was the ill-fated iAPX 432 system. The iAPX 432 was such a dismal failure that Intel might not have survived had it not been for a big stroke of luck — IBM decided to use the 8088 microprocessor in their personal computer system.

To most computer historians, there were two watershed events in the history of the personal computer. The first was the introduction of the Visicalc spreadsheet program on the Apple II personal computer system. This single program demonstrated that there was a real reason for owning a computer beyond the nerdy "gee, I ve got my own computer" excuse. Visicalc quickly (and, alas, briefly) made Apple Computer the largest PC company around. The second big event in the history of personal computers was, of course, the introduction of the IBM PC. The fact that IBM, a "real" computer company, would begin building PCs legitimized the market. Up to that point, businesses tended to ignore PCs and treated them as toys that nerdy engineers liked to play with. The introduction of the IBM PC caused a lot of businesses to take notice of these new devices. Not only did they take notice, but they liked what they saw. Although IBM cannot make the claim that they started the PC revolution, they certainly can take credit for giving it a big jumpstart early on in its life.

Once people began buying lots of PCs, it was only natural that people would start writing and selling software for these machines. The introduction of the IBM PC greatly expanded the marketplace for computer systems. Keep in mind that at the time of the IBM PC s introduction, most computer systems had only sold tens of thousands of units. The more popular models, like the TRS-80 and Apple II had only sold hundreds of thosands of units. Indeed, it wasn t until a couple of years after the introduction of the IBM PC that the first computer system sold one million units; and that was a Commodore 64 system, not the IBM PC.

For a brief period, the introduction of the IBM PC was a godsend to most of the other computer manufacturers. The original IBM PC was underpowered and quite a bit more expensive than its counterparts. For example, a dual-floppy disk drive PC with 64 Kilobytes of memory and a monochrome display sold for $3,000. A comparable Apple II system with a color display sold for under $2,000. The original IBM PC with it s 4.77 MHz 8088 processor (that s four-point-seven-seven, not four hundred seventy-seven!) was only about two to three times as fast as the Apple II with its paltry 1 MHz eight-bit 6502 processor. The fact that most Apple II software was written by expert assembly language programmers while most (early) IBM software was written in a high level language (often interpreted) or by inexperienced 8086 assembly language programmers narrowed the gap even more.

Nonetheless, software development on PCs accelerated. The wide range of different (and incompatible) systems made software development somewhat risky. Those who did not have an emotional attachment to one particular company (and didn t have the resources to develop for more than one platform) generally decided to go with IBM s PC when developing their software.

One problem with the 8086 s architecture was beginning to show through by 1983 (remember, this is five years after Intel introduced the 8086). The *segmented memory architecture* that allowed them to extend their 16-bit addressing scheme to 20 bits (allowing the 8086 to address a megabyte of memory) was being attacked on two fronts. First, this segmented addressing scheme was difficult to use in a program, especially if that program needed to access more than 64 kilobytes of data or, worse yet, needed to access a single data structure that was larger than 64K long. By 1983 software had reached the level of sophistication that most programs were using this much memory and many needed large data structures. The software community as a whole began to grumble and complain about this segmented memory architecture and what a stupid thing it was.

The second problem with Intel s segmented architecture is that it only supported a maximum of a one megabyte address space. Worse, the design of the IBM PC effectively limited the amount of RAM the system could have to 640 kilobytes. This limitation was also beginning to create problems for more sophisticated programs

running on the PC. Once again, the software development community grumbled and complained about Intel s segmented architecture and the limitations it imposed upon their software.

About the time people began complaining about Intel s architecture, Intel began running an ad campaign bragging about how great their chip was. They quoted top executives at companies like Visicorp (the outfit selling Visicalc) who claimed that the segmented architecture was great. They also made a big deal about the fact that over a billion dollars worth of software had been written for their chip. This was all marketing hype, of course. Their chip was not particularly special. Indeed, the 8086 s contemporaries (Z8000, 68000, and 16032) were archiecturally superior. However, Intel was quite right about one thing — people had written a lot of software for the 8086 and most of the really good stuff was written in 8086 assembly language and could not be easily ported to the other processors. Worse, the software that people were writing for the 8086 was starting to get large; making it even more difficult to port it to the other chips. As a result, software developers were becoming locked into using the 8086 CPU.

About this time, Intel undoubtedly realized that they were getting locked into the 80x86 architecture, as well. The iAPX 432 project was on its death bed. People were no more interested in the iAPX 432 than they were the other processors (in fact, they were less interested). So Intel decided to do the only reasonable thing — extend the 8086 family so they could continue to make more money off their cash cow.

The first real extension to the 8086 family that found its way into general purpose PCs was the 80286 that appeared in 1982. This CPU answered the second complaint by adding the ability to address up to 16 MBytes of RAM (a formidable amount in 1982). Unfortunately, it did not extend the segment size beyond 64 kilobytes. In 1985 Intel introduced the 80386 microprocessor. This chip answered most of the complaints about the x86 family, and then some, but people still complained about these problems for nearly ten years after the introduction of the 80386.

Intel was suffering at the hands of Microsoft and the installed base of existing PCs. When IBM introduced the floppy disk drive for the IBM PC they didn t choose an operating system to ship with it. Instead, they offered their customers a choice of the widely available operating systems at the time. Of course, Digital Research had ported CP/M to the PC, UCSD/Softech had ported UCSD Pascal (a very popular language/operating system at the time) to the PC, and Microsoft had quickly purchased a CP/M knock-off named QD DOS (for Quick and Dirty DOS) from Seattle Microsystems, relabelled it "MS-DOS", and offered this as well. CP/M-86 cost somewhere in the vicenity of $595. UCSD Pascal was selling for something like $795. MS-DOS was selling for $50. Guess which one sold more copies! Within a year, almost no one ran CP/M or UCSD Pascal on PCs. Microsoft and MS-DOS (also called IBM DOS) ruled the PC.

MS-DOS v1.0 lived up to its "quick and dirty" heritage. Working furiously, Microsoft s engineers added lots of new features (many taken from the UNIX operating system and shell program) and MS-DOS v2.0 appeared shortly thereafter. Although still crude, MS-DOS v2.0 was a substantial improvement and people started writing tons of software for it.

Unfortunately, MS-DOS, even in its final version, wasn t the best operating system design. In particular, it left all but rudimentary control of the hardware to the application programmer. It provided a file system so application writers didn t have to deal with the disk drive and it provided mediocre support for keyboard input and character display. It provided nearly useless support for other devices. As a result, most application programmers (and most high level languages) bypassed MS-DOS device control and used MS-DOS primarily as a file system module.

In addition to poor device management, MS-DOS provided nearly non-existant memory management. For all intents and purposes, once MS-DOS started a program running, it was that program s responsibility to manage the system s resources. Not only did this create extra work for application programmers, but it was one of the main reasons most software could not take advantage of the new features Intel was adding to their microprocessors.

When Intel introduced the 80286 and, later, the 80386, the only way to take advantage of their extra addressing capabilities and the larger segments of the 80386 was to operate in a so-called *protected mode*. Unfortunately, neither MS-DOS nor most applications (that managed memory themselves) were capable of operating in protected mode without substantial change (actually, it would have been easy to modify MS-DOS to use protected mode, but it would have broken all the existing software that ran under MS-DOS; Microsoft, like Intel, couldn t afford to alienate the software developers in this manner).

Even if Microsoft could magically make MS-DOS run under protected mode, they couldn t afford to do so. When Intel introduced the 80386 microprocessor it was a very expensive device (the chip itself cost over $1,000 at initial introduction). Although the 80286 had been out for three years, systems built around the 8088 were still extremely popular (since they were much lower cost than systems using the 80386). Software developers had a choice: they could solve their memory addressing problems and use the new features of the 80386 chip but limit their market to the few who had 80386 systems, or they could continue to suffer with the 64K segment limitation imposed by the 8088 and MS-DOS and be able to sell their software to millions of users who owned one of the earlier machines. The marketing departments of these companies ruled the day, all software was written to run on plain 8088 boxes so that it had a larger market. It wasn t until 1995, when Microsoft introduced Windows 95 that people finally felt they could abandon processors earlier than the 80386. The end result was the people were still complaining about the Intel architecture and its 64K segment limitation ten years after Intel had corrected the problem. The concept of upwards compatibility was clearly a double-edged sword in this case.

Segmentation had developed such a bad name over the years that Microsoft abandoned the use of segments in their 32-bit versions of Windows (95, 98, NT, 2000, ME, etc.). In a couple of respects, this was a real shame because Intel finally did segmentation right (or, at least, pretty good) in the 80386 and later processors. By not allowing the use of segmentation in Win32 programs Microsoft limited the use of this powerful feature. They also limited their users to a maximum address space of 4GB (the Pentium Pro and later processors were capable of addressing 64GB of physical memory). Considering that many applications are starting to push the 4GB barrier, this limitation on Microsoft s part was ill-considered. Nevertheless, the "flat" memory model that Microsoft employs is easier to write software for, undoubtedly a big part of their decision not to use segmentation.

The introduction of Windows NT, that actually ran on CPUs other than Intel s, must have given Intel a major scare. Fortunately for Intel, NT was an asbysmal failure on non-Intel architectures like the Alpha and the PowerPC. On the other hand, the new Windows architecture does make it easier to move existing applications to 64-bit processors like the IA-64; so maybe WinNT s flexibility will work to Intel s advantage after all.

The 8086 software legacy has both advanced and retarded the 80x86 architecture. On the one hand, had software developers not written so much software for the 80x86, Intel would have abandoned the family in favor of something better a long time ago (not an altogether bad thing, in many people s opinions). On the other hand, however, the general acceptance of the 80386 and later processors was greatly delayed by the fact that software developers were writing software for the installed base of processors.

Around 1996, two types of software actually accellerated the design and acceptance of Intel s newer processors: multimedia software and games. When Intel introduced the MMX extensions to the 80x86 instruction set, software developers ignored the installed base and immediately began writing software to take advantage of these new instructions. This change of heart took place because the MMX instructions allowed developers to do things they hadn t been able to do before - not simply run faster, but run fast enough to display actual video and quick render 3D images. Combined with a change in pricing policy by Intel on new processor technology, the public quickly accepted these new systems.

Hard-core gamers, multimedia artists, and others quickly grabbed new machines and software as it became available. More often than not, each new generation of software would only run on the latest hardware, forcing these individuals to upgrade their equipment far more rapidly than ever before.

Intel, sensing an opportunity here, began developing CPUs with additional instruction targetted at specific applications. For example, the Pentium III introduced the SIMD (pronounced SIM-DEE) instructions that did

for floating point calculations what the MMX instructions did for integer calculations. Intel also hired lots of software engineers and began funding research into topic areas like speech recognition and (visual) pattern recognition in order to drive the new technologies that would require the new instructions their Pentium IV and later processors would offer. As this is being written, Intel is busy developing new uses for their specialized instructions so that system designers and software developers continue to use the 80x86 (and, perhaps, IA-64) family chips.

However, this discussion of fancy instruction sets is getting way ahead of the game. Let s take a long step back to the original 8086 chip and take a look at how system designers put a CPU together.

4.4 Basic CPU Design

A fair question to ask at this point is How exactly does a CPU perform assigned chores? This is accomplished by giving the CPU a fixed set of commands, or instructions, to work on. Keep in mind that CPU designers construct these processors using logic gates to execute these instructions. To keep the number of logic gates reasonably small, CPU designers must necessarily restrict the number and complexity of the commands the CPU recognizes. This small set of commands is the CPU s instruction set.

Programs in early (pre-Von Neumann) computer systems were often hard-wired into the circuitry . That is, the computer s wiring determined what problem the computer would solve. One had to rewire the circuitry in order to change the program. A very difficult task. The next advance in computer design was the programmable computer system, one that allowed a computer programmer to easily rewire the computer system using a sequence of sockets and plug wires. A computer program consisted of a set of rows of holes (sockets), each row representing one operation during the execution of the program. The programmer could select one of several instructions by plugging a wire into the particular socket for the desired instruction (see Figure 4.1).

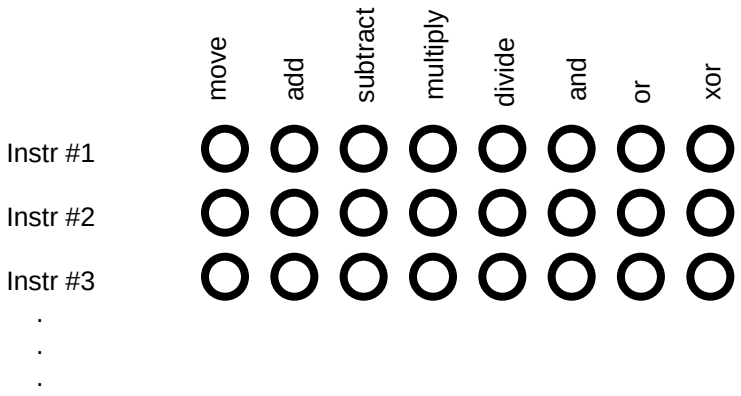

Figure 4.1 Patch Panel Programming

Of course, a major difficulty with this scheme is that the number of possible instructions is severely limited by the number of sockets one could physically place on each row. However, CPU designers quickly discovered that with a small amount of additional logic circuitry, they could reduce the number of sockets required from n holes for n instructions to $\log_2(n)$ holes for n instructions. They did this by assigning a numeric code to each instruction and then encode that instruction as a binary number using $\log_2(n)$ holes (see Figure 4.2).

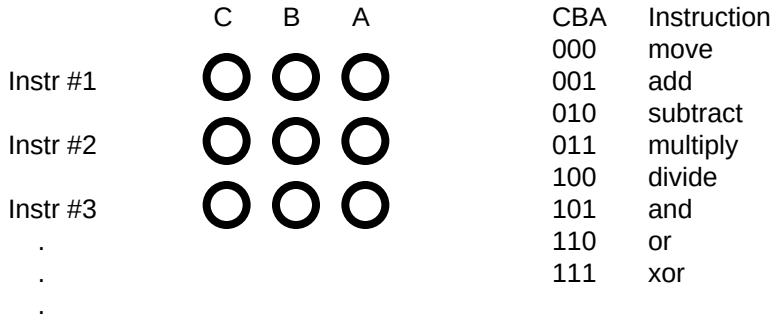

Figure 4.2 Encoding Instructions

This addition requires eight logic functions to decode the A, B, and C bits from the patch panel, but the extra circuitry is well worth the cost because it reduces the number of sockets that must be repeated for each instruction (this circuitry, by the way, is nothing more than a single three-line to eight-line decoder).

Of course, many CPU instructions are not stand-alone. For example, the move instruction is a command that moves data from one location in the computer to another (e.g., from one register to another). Therefore, the move instruction requires two operands: a source operand and a destination operand. The CPU s designer usually encodes these source and destination operands as part of the machine instruction, certain sockets correspond to the source operand and certain sockets correspond to the destination operand. Figure 4.3 shows one possible combination of sockets to handle this. The move instruction would move data from the source register to the destination register, the add instruction would add the value of the source register to the destination register, etc.

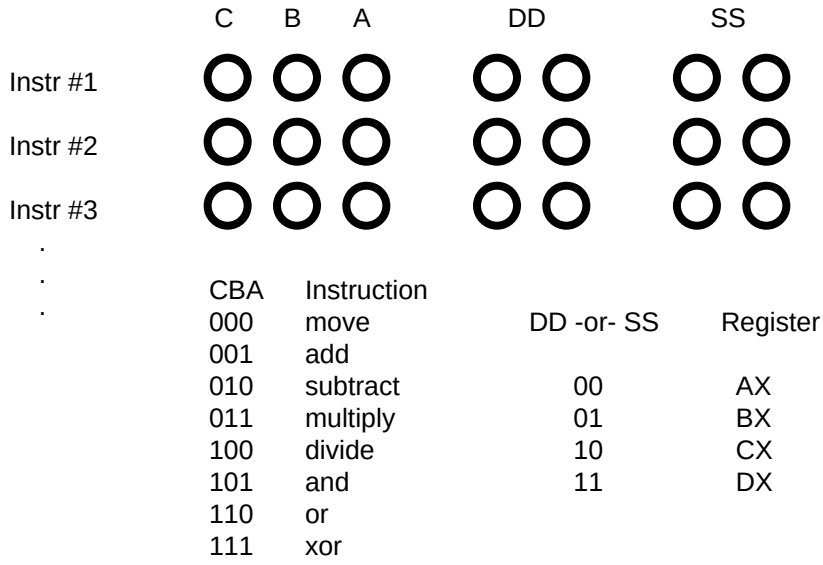

Figure 4.3 Encoding Instructions with Source and Destination Fields

One of the primary advances in computer design that the VNA provides is the concept of a stored program. One big problem with the patch panel programming method is that the number of program steps (machine instructions) is limited by the number of rows of sockets available on the machine. John Von Neumann and oth-

ers recognized a relationship between the sockets on the patch panel and bits in memory; they figured they could store the binary equivalents of a machine program in main memory and fetch each program from memory, load it into a special decoding register that connected directly to the instruction decoding circuitry of the CPU.

The trick, of course, was to add yet more circuitry to the CPU. This circuitry, the control unit (CU), fetches instruction codes (also known as operation codes or opcodes) from memory and moves them to the instruction decoding register. The control unit contains a special register, the instruction pointer that contains the address of an executable instruction. The control unit fetches this instruction s opcode from memory and places it in the decoding register for execution. After executing the instruction, the control unit increments the instruction pointer and fetches the next instruction from memory for execution, and so on.

When designing an instruction set, the CPU s designers generally choose opcodes that are a multiple of eight bits long so the CPU can easily fetch complete instructions from memory. The goal of the CPU s designer is to assign an appropriate number of bits to the instruction class field (move, add, subtract, etc.) and to the operand fields. Choosing more bits for the instruction field lets you have more instructions, choosing additional bits for the operand fields lets you select a larger number of operands (e.g., memory locations or registers). There are additional complications. Some instructions have only one operand or, perhaps, they don t have any operands at all. Rather than waste the bits associated with these fields, the CPU designers often reuse these fields to encode additional opcodes, once again with some additional circuitry. The Intel 80x86 CPU family takes this to an extreme with instructions ranging from one to almost 15 bytes long[8].

4.5 Decoding and Executing Instructions: Random Logic Versus Microcode

Once the control unit fetches an instruction from memory, you may wonder "exactly how does the CPU execute this instruction?" In traditional CPU design there have been two common approaches: hardwired logic and emulation. The 80x86 family uses both of these techniques.

A hardwired, or *random logic*[9], approach uses decoders, latches, counters, and other logic devices to move data around and operate on that data. The microcode approach uses a very fast but simple internal processor that uses the CPU s opcodes as an index into a table of operations (the *microcode*) and executes a sequence of *microinstructions* that do the work of the *macroinstruction* (i.e., the CPU instruction) they are emulating.

The random logic approach has the advantage that it is possible to devise faster CPUs if typical CPU speeds are faster than typical memory speeds (a situation that has been true for quite some time). The drawback to random logic is that it is difficult to design CPUs with large and complex instruction sets using a random logic approach. The logic to execute the instructions winds up requiring large percentage of the chip s real estate and it becomes difficult to properly lay out the logic so that related circuits are close to one another in the two-dimensional space of the chip,

CPUs based on microcode contain a small, very fast, execution unit that fetches instructions from the microcode bank (which is really nothing more than fast ROM on the CPU chip). This microcode executes one microinstruction per clock cycle and a sequence of microinstructions decode the instruction, fetch its operands, move the operands to appropriate functional units that do whatever calculations are necessary, store away necessary results, and then update appropriate registers and flags in anticipation of the next instruction.

8. Though this is, by no means, the most complex instruction set. The VAX, for example, has instructions up to 150 bytes long!
9. There is actually nothing random about this logic at all. This design technique gets its name from the fact that if you view a photomicrograph of a CPU die that uses microcode, the microcode section looks very regular; the same photograph of a CPU that utilizes random logic contains no such easily discernable patterns.

The microcode approach may appear to be substantially slower than the random logic approach because of all the steps involved. Actually, this isn t necessarily true. Keep in mind that with a random logic approach to instruction execution, part of the random logic is often a sequencer that steps through several states (one state per clock cycle). Whether you use your clock cycles executing microinstructions or stepping through a random logic state machine, you re still burning up clock cycles.

One advantage of microcode is that it makes better reuse of existing silicon on the CPU. Many CPU instructions (macroinstructions) execute some of the same microinstructions as many other instructions. This allows the CPU designer to use microcode subroutines to implement many common operations, thus saving silicon on the CPU. While it is certainly possible to share circuitry in a random logic device, this is often difficult if two circuits could otherwise share some logic but are across the chip from one another.

Another advantage of microcode is that it lets you create some very complex instructions that consist of several different operations. This provides programmers (especially assembly language programmers) with the ability to do more work with fewer instructions in their programs. In theory, this lets them write faster programs since they now execute half as many instructions, each doing twice the work of a simpler instruction set (the 80x86 MMX instruction set extension is a good example of this theory in action, although the MMX instructions do not use a microcode implementation).

Microcode does suffer from one disadvantage compared to random logic: the speed of the processor is tied to the speed of the internal microcode execution unit. Although the "microengine" itself is usually quite fast, the microengine must fetch its instruction from the microcode ROM. Therefore, if memory technology is slower than the execution logic, the microcode ROM will slow the microengine down because the system will have to introduce wait states into the microcode ROM access. Actually, microengines generally don t support the use of wait states, so this means that the microengine will have to run at the same speed as the microcode ROM. This effectively limits the speed at which the microengine, and therefore the CPU, can run.

Which approach is better for CPU design? That depends entirely on the current state of memory technology. If memory technology is faster than CPU technology, then the microcode approach tends to make more sense. If memory technology is slower than CPU technology, then random logic tends to produce the faster CPUs.

When Intel first began designing the 8086 CPU sometime between 1976 and 1978, memory technology was faster so they used microcode. Today, CPU technology is much faster than memory technology, so random logic CPUs tend to be faster. Most modern (non-x86) processors use random logic. The 80x86 family uses a combination of these technologies to improve performance while maintaining compatibility with the complex instruction set that relied on microcode way back in 1978.

4.6 RISC vs. CISC vs. VLIW

In the 1970 s, CPU designers were busy extending their instruction sets to make their chips easier to program. It was very common to find a CPU designer poring over the assembly output of some high level language compiler searching for common two and three instruction sequences the compiler would emit. The designer would then create a single instruction that did the work of this two or three instruction sequence, the compiler writer would modify the compiler to use this new instruction, and a recompilation of the program would, presumably, produce a faster and shorter program than before.

Digital Equipment Corporation (now part of Compaq Computer who is looking at merging with Hewlett Packard as this is being written) raised this process to a new level in their VAX minicomputer series. It is not surprising, therefore, that many research papers appearing in the 1980 s would commonly use the VAX as an example of what not to do.

The problem is, these designers lost track of what they were trying to do, or to use the old cliche, they couldn t see the forest for the trees. They assumed that there were making their processors faster by executing a

single instruction that previously required two or more. They also assumed that they were making the programs smaller, for exactly the same reason. They also assumed that they were making the processors easier to program because programmers (or compilers) could write a single instruction instead of using multiple instructions. In many cases, they assumed wrong.

In the early 80 s, researchers at IBM and several institutions like Stanford and UC Berkeley challenged the assumptions of these designers. They wrote several papers showing how complex instructions on the VAX mini-computer could actually be done faster (and sometimes in less space) using a sequence of simpler instructions. As a result, most compiler writers did not use the fancy new instructions on the VAX (nor did assembly language programmers). Some might argue that having an unused instruction doesn t hurt anything, but these researchers argued otherwise. They claimed that any unnecessary instructions required additional logic to implement and as the complexity of the logic grows it becomes more and more difficult to produce a high clock speed CPU.

This research led to the development of the RISC, or Reduced Instruction Set Computer, CPU. The basic idea behind RISC was to go in the opposite direction of the VAX. Decide what the smallest reasonable instruction set could be and implement that. By throwing out all the complex instructions, RISC CPU designers could use random logic rather than microcode (by this time, CPU speeds were outpacing memory speeds). Rather than making an individual instruction more complex, they could move the complexity to the system level and add many on-chip features to improve the overall system performance (like caches, pipelines, and other advanced mainframe features of the time). Thus, the great "RISC vs. CISC[10]" debate was born.

Before commenting further on the result of this debate, you should realize that RISC actually means "(Reduced Instruction) Set Computer," not "Reduced (Instruction Set) Computer." That is, the goal of RISC was to reduce the complexity of individual instructions, not necessarily reduce the number of instructions a RISC CPU supports. It was often the case that RISC CPUs had fewer instructions than their CISC counterparts, but this was not a precondition for calling a CPU a RISC device. Many RISC CPUs had more instructions than some of their CISC contemporaries, depending on how you count instructions.

First, there is no debate about one thing: if you have two CPUs, one RISC and one CISC and they both run at the same clock frequency and they execute the same average number of instructions per clock cycle, CISC is the clear winner. Since CISC processors do more work with each instruction, if the two CPUs execute the same number of instructions in the same amount of time, the CISC processor usually gets more work done.

However, RISC performance claims were based on the fact that RISC s simpler design would allow the CPU designers to reduce the overall complexity of the chip, thereby allowing it to run at a higher clock frequency. Further, with a little added complexity, they could easily execute more instructions per clock cycle, on the average, than their CISC contemporaries.

One drawback to RISC CPUs is that their code density was much lower than CISC CPUs. Although memory devices were dropping in price and the need to keep programs small was decreasing, low code density requires larger caches to maintain the same number of instructions in the cache. Further, since memory speeds were not keeping up with CPU speeds, the larger instruction sizes found on the RISC CPUs meant that the system spent more time bringing in those instructions from memory to cache since they could transfer fewer instructions per bus transaction. For many years, CPU architects argued to and fro about whether RISC or CISC was the better approach. With one big footnote, the RISC approach has generally won the argument. Most of the popular CISC systems, e.g., the VAX, the Z8000, the 16032/32016, and the 68000, have quitely faded away to be replaced by the likes of the PowerPC, the MIPS CPUs, the Alpha, and the SPARC. The one footnote here is, of course, the 80x86 family. Intel has proven that if you really want to keep extending a CISC architecture, and you re willing to throw a lot of money at it, you can extend it far beyond what anyone ever expected. As of late 2001/early 2002 the 80x86 is the raw performance leader. The CPU runs at a higher clock frequency than the competing RISC

10.CISC stands for Complex Instruction Set Computer and defines those CPUs that were popular at the time like the VAX and the 80x86.

chips; it executes fairly close to the same number of instructions per clock cycle as the competing RISC chips; it has about the same "average instruction size to cache size" ratio as the RISC chips; and it is a CISC, so many of the instructions do more work than their RISC equivalents. So overall, the 80x86 is, on the average, faster than contemporary RISC chips[11].

To achieve this raw performance advantage, the 80x86 has borrowed heavily from RISC research. Intel has divided the instruction set into a set of simple instructions that Intel calls the "RISC core" and the remaining, complex instructions. The complex instructions do not execute as rapidly as the RISC core instructions. In fact, it is often the case that the task of a complex instruction can be accomplished faster using multiple RISC core instructions. Intel supports the complex instructions to provide full compatibility with older software, but compiler writers and assembly language programmers tend to avoid the use of these instructions. Note that Intel moves instructions between these two sets over time. As Intel improves the processor they tend to speed up some of the slower, complex, instructions. Therefore, it is not possible to give a static list of instructions you should avoid; instead, you will have to refer to Intel s documentation for the specific processor you use.

Later Pentium processors do not use an interpretive engine and microcode like the earlier 80x86 processors. Instead, the Pentium core processors execute a set of "micro-operations" (or "micro-ops"). The Pentium processors translate the 80x86 instruction set into a sequence of micro-ops on the fly. The RISC core instructions typically generate a single micro-op while the CISC instructions generate a sequence of two or more micro-ops. For the purposes of determining the performance of a section of code, we can treat each micro-op as a single instruction. Therefore, the CISC instructions are really nothing more than "macro-instructions" that the CPU automatically translates into a sequence of simpler instructions. This is the reason the complex instructions take longer to execute.

Unfortunately, as the x86 nears its 25th birthday, it s clear (to Intel, at least) that it s been pushed to its limits. This is why Intel is working with HP to base their IA-64 architecture on the PA-RISC instruction set. The IA-64 architecture is an interesting blend. On the one hand, it (supposedly) supports object-code compatibility with the previous generation x86 family (though at reduced performance levels). Obviously, it s a RISC architecture since it was originally based on Hewlet-Packard s PA-RISC (PA=Precision Architecture) design. However, Intel and HP have extended on the RISC design by using another technology: Very Long Instruction Word (VLIW) computing. The idea behind VLIW computing is to use a very long opcode that handle multiple operations in parallel. In some respects, this is similar to CISC computing since a single VLIW "instruction" can do some very complex things. However, unlike CISC instructions, a VLIW instruction word can actually complete several independent tasks simultaneously. Effectively, this allows the CPU to execute some number of instructions in parallel.

Intel s VLIW approach is risky. To succeed, they are depending on compiler technology that doesn t yet exist. They made this same mistake with the iAPX 432. It remains to be seen whether history is about to repeat itself or if Intel has a winner on their hands.

4.7 Instruction Execution, Step-By-Step

To understand the problems with developing an efficient CPU, let’s consider four representative 80x86 instructions: MOV, ADD, LOOP, and JNZ (jump if not zero). These instructions will allow us to explore many of the issues facing the x86 CPU designer.

You’ve seen the MOV and ADD instructions in previous chapters so there is no need to review them here. The LOOP and JNZ instructions are new, so it’s probably a good idea to explain what they do before proceeding. Both of these instructions

11.Note, by the way, that this doesn t imply that 80x86 systems are faster than computer systems built around RISC chips. Many RISC systems gain additional speed by supporting multiple processors better than the x86 or by having faster bus throughput. This is one reason, for example, why Internet companies select Sun equipment for their web servers.

are *conditional jump* instructions. A conditional jump instruction tests some condition and jumps to some other instruction in memory if the condition is true and they fall through to the next instruction if the condition is false. This is basically the opposite of HLA's IF statement (which falls through if the condition is true and jumps to the else section if the condition is false). The JNZ (jump if not zero) instruction tests the CPU's zero flag and transfers control to some target location if the zero flag contains zero; JNZ falls through to the next instruction if the zero flag contains one. The program specifies the target instruction to jump to by specifying the distance from the JNZ instruction to the target instruction as a small signed integer (for our purposes here, we'll assume that the distance is within the range ±128 bytes so the instruction uses a single byte to specify the distance to the target location).

The last instruction of interest to us here is the LOOP instruction. The LOOP instruction decrements the value of the ECX register and transfers control to a target instruction within ±128 bytes if ECX does not contain zero (after the decrement). This is a good example of a CISC instruction since it does multiple operations: (1) it subtracts one from ECX and then it (2) does a conditional jump if ECX does not contain zero. That is, LOOP is equivalent to the following two 80x86 instructions[12]:

```
        loop SomeLabel;

-is roughly equivalent to-

        dec( ecx );
        jnz SomeLabel;
```

Note that *SomeLabel* specifies the address of the target instruction that must be within about ±128 bytes of the LOOP or JNZ instructions above. The LOOP instruction is a good example of a complex (vs. RISC core) instruction on the Pentium processors. It is actually faster to execute a DEC and a JNZ instruction[13] than it is to execute a LOOP instruction. In this section we will not concern ourselves with this issue; we will assume that the LOOP instruction operates as though it were a RISC core instruction.

The 80x86 CPUs do not execute instructions in a single clock cycle. For example, the MOV instruction (which is relatively simple) could use the following execution steps[14]:

- Fetch the instruction byte from memory.
- Update the EIP register to point at the next byte.
- Decode the instruction to see what it does.
- If required, fetch a 16-bit instruction operand from memory.
- If required, update EIP to point beyond the operand.
- If required, compute the address of the operand (e.g., EBX+disp) .
- Fetch the operand.
- Store the fetched value into the destination register

If we allocate one clock cycle for each of the above steps, an instruction could take as many as eight clock cycles to complete (note that three of the steps above are optional, depending on the MOV instruction's addressing mode, so a simple MOV instruction could complete in as few as five clock cycles).

The ADD instruction is a little more complex. Here's a typical set of operations the ADD(reg, reg) instruction must complete:

- Fetch the instruction byte from memory.
- Update EIP to point at the next byte.
- Decode the instruction.
- Get the value of the source operand and send it to the ALU.
- Fetch the value of the destination operand (a register) and send it to the ALU.
- Instruct the ALU to add the values.
- Store the result back into the first register operand.
- Update the flags register with the result of the addition operation.

If the source operand is a memory location, the operation is slightly more complicated:

12.This sequence is not exactly equivalent to LOOP since this sequence affects the flags while LOOP does not.

13.Actually, you ll see a little later that there is a *decrement* instruction you can use to subtract one from ECX. The decrement instruction is better because it is shorter.

14.It is not possible to state exactly what steps each CPU requires since many CPUs are different from one another.

- Fetch the instruction byte from memory.
- Update EIP to point at the next byte.
- Decode the instruction.
- If required, fetch a displacement for use in the effective address calculation
- If required, update EIP to point beyond the displacement value.
- Get the value of the source operand from memory and send it to the ALU.
- Fetch the value of the destination operand (a register) and send it to the ALU.
- Instruct the ALU to add the values.
- Store the result back into the register operand.
- Update the flags register with the result of the addition operation.

ADD(const, memory) is the messiest of all, this code sequence looks something like the following:

- Fetch the instruction byte from memory.
- Update EIP to point at the next byte.
- Decode the instruction.
- If required, fetch a displacement for use in the effective address calculation
- If required, update EIP to point beyond the displacement value.
- Fetch the constant value from memory and send it to the ALU.
- Update EIP to point beyond the constant's value (at the next instruction in memory).
- Get the value of the source operand from memory and send it to the ALU.
- Instruct the ALU to add the values.
- Store the result back into the memory operand.
- Update the flags register with the result of the addition operation.

Note that there are other forms of the ADD instruction requiring their own special processing. These are just representative examples. As you see in these examples, the ADD instruction could take as many as ten steps (or cycles) to complete. Note that this is one advantage of a RISC design. Most RISC design have only one or two forms of the ADD instruction (that add registers together and, perhaps, that add constants to registers). Since register to register adds are often the fastest (and constant to register adds are probably the second fastest), the RISC CPUs force you to use the fastest forms of these instructions.

The JNZ instruction might use the following sequence of steps:

- Fetch the instruction byte from memory.
- Update EIP to point at the next byte.
- Decode the instruction.
- Fetch a displacement byte to determine the jump distance send this to the ALU
- Update EIP to point at the next byte.
- Test the zero flag to see if it is clear.
- If the zero flag was clear, copy the EIP register to the ALU.
- If the zero flag was clear, instruct the ALU to add the displacement and EIP register values.
- If the zero flag was clear, copy the result of the addition above back to the EIP register.

Notice how the JNZ instruction requires fewer steps if the jump is not taken. This is very typical for conditional jump instructions. If each step above corresponds to one clock cycle, the JNZ instruction would take six or nine clock cycles, depending on whether the branch is taken. Because the 80x86 JNZ instruction does not allow different types of operands, there is only one sequence of steps needed for this application.

The 80x86 LOOP instruction might use an execution sequence like the following:

- Fetch the instruction byte from memory.
- Update EIP to point at the next byte.
- Decode the instruction.
- Fetch the value of the ECX register and send it to the ALU.
- Instruct the ALU to decrement the value.
- Send the result back to the ECX register. Set a special internal flag if this value is non-zero.
- Fetch a displacement byte to determine the jump distance send this to the ALU
- Update EIP to point at the next byte.
- Test the special flag to see if ECX was non-zero.
- If the flag was set, copy the EIP register to the ALU.
- If the flag was set, instruct the ALU to add the displacement and EIP register values.

- If the flag was set, copy the result of the addition above back to the EIP register.

Although a given 80x86 CPU might not execute the steps for the instructions above, they all execute some sequence of operations. Each operation requires a finite amount of time to execute (generally, one clock cycle per operation or *stage* as we usually refer to each of the above steps). Obviously, the more steps needed for an instruction, the slower it will run. This is why complex instructions generally run slower than simple instructions, complex instructions usually have lots of execution stages.

4.8 Parallelism – the Key to Faster Processors

An early goal of the RISC processors was to execute one instruction per clock cycle, on the average. However, even if a RISC instruction is simplified, the actual execution of the instruction still requires multiple steps. So how could they achieve this goal? And how do later members the 80x86 family with their complex instruction sets also achieve this goal? The answer is parallelism.

Consider the following steps for a MOV(reg, reg) instruction:

- Fetch the instruction byte from memory.
- Update the EIP register to point at the next byte.
- Decode the instruction to see what it does.
- Fetch the source register.
- Store the fetched value into the destination register

There are five stages in the exection of this instruction with certain dependencies between each stage. For example, the CPU must fetch the instruction byte from memory before it updates EIP to point at the next byte in memory. Likewise, the CPU must decode the instruction before it can fetch the source register (since it doesn't know it needs to fetch a source register until it decodes the instruction). As a final example, the CPU must fetch the source register before it can store the fetched value in the destination register.

Most of the stages in the execution of this MOV instruction are *serial*. That is, the CPU must execute one stage before proceeding to the next. The one exception is the "Update EIP" step. Although this stage must follow the first stage, none of the following stages in the instruction depend upon this step. Therefore, this could be the third, forth, or fifth step in the calculation and it wouldn't affect the outcome of the instruction. Further, we could execute this step concurrently with any of the other steps and it wouldn't affect the operation of the MOV instruction, e.g.,

- Fetch the instruction byte from memory.
- Decode the instruction to see what it does.
- Fetch the source register and update the EIP register to point at the next byte.
- Store the fetched value into the destination register

By doing two of the stages in parallel, we can reduce the execution time of this instruction by one clock cycle. Although the remaining stages in the "mov(reg, reg);" instruction must remain serialized (that is, they must take place in exactly this order), other forms of the MOV instruction offer similar opportunities to overlapped portions of their execution to save some cycles. For example, consider the "mov([ebx+disp], eax);" instruction:

- Fetch the instruction byte from memory.
- Update the EIP register to point at the next byte.
- Decode the instruction to see what it does.
- Fetch a displacement operand from memory.
- Update EIP to point beyond the displacement.
- Compute the address of the operand (e.g., EBX+disp) .
- Fetch the operand.
- Store the fetched value into the destination register

Once again there is the opportunity to overlap the execution of several stages in this instruction, for example:

- Fetch the instruction byte from memory.
- Decode the instruction to see what it does and update the EIP register to point at the next byte.
- Fetch a displacement operand from memory.
- Compute the address of the operand (e.g., EBX+disp) and update EIP to point beyond the displacement..
- Fetch the operand.

- Store the fetched value into the destination register

In this example, we reduced the number of execution steps from eight to six by overlapping the update of EIP with two other operations.

As a last example, consider the "add(const, [ebx+disp]);" instruction (the instruction with the largest number of steps we've considered thus far). It's non-overlapped execution looks like this:

- Fetch the instruction byte from memory.
- Update EIP to point at the next byte.
- Decode the instruction.
- Fetch a displacement for use in the effective address calculation
- Update EIP to point beyond the displacement value.
- Fetch the constant value from memory and send it to the ALU.
- Compute the address of the memory operand (EBX+disp).
- Get the value of the source operand from memory and send it to the ALU.
- Instruct the ALU to add the values.
- Store the result back into the memory operand.
- Update the flags register with the result of the addition operation.
- Update EIP to point beyond the constant's value (at the next instruction in memory).

We can overlap at least three steps in this instruction by noting that certain stages don't depend on the result of their immediate predecessor

- Fetch the instruction byte from memory.
- Decode the instruction and update EIP to point at the next byte.
- Fetch a displacement for use in the effective address calculation
- Update EIP to point beyond the displacement value.
- Fetch the constant value from memory and send it to the ALU.
- Compute the address of the memory operand (EBX+disp).
- Get the value of the source operand from memory and send it to the ALU.
- Instruct the ALU to add the values.
- Store the result back into the memory operand and update the flags register with the result of the addition operation and update EIP to point beyond the constant's value.

Note that we could not merge one of the "Update EIP" operations because the previous stage and following stages of the instruction both use the value of EIP before and after the update.

Unlike the MOV instruction, the steps in the ADD instruction above are not all dependent upon the previous stage in the instruction's execution. For example, the sequence above fetches the constant from memory and then computes the effective address (EBX+disp) of the memory operand. Neither operation depends upon the other, so we could easily swap their positions above to yield the following:

- Fetch the instruction byte from memory.
- Decode the instruction and update EIP to point at the next byte.
- Fetch a displacement for use in the effective address calculation
- Update EIP to point beyond the displacement value.
- Compute the address of the memory operand (EBX+disp).
- Fetch the constant value from memory and send it to the ALU.
- Get the value of the source operand from memory and send it to the ALU.
- Instruct the ALU to add the values.
- Store the result back into the memory operand and update the flags register with the result of the addition operation and update EIP to point beyond the constant's value.

This doesn't save any steps, but it does reduce some dependencies between certain stages and their immediate predecessors, allowing additional parallel operation. For example, we can now merge the "Update EIP" operation with the effective address calculation:

- Fetch the instruction byte from memory.
- Decode the instruction and update EIP to point at the next byte.
- Fetch a displacement for use in the effective address calculation

- Compute the address of the memory operand (EBX+disp) and update EIP to point beyond the displacement value.
- Fetch the constant value from memory and send it to the ALU.
- Get the value of the source operand from memory and send it to the ALU.
- Instruct the ALU to add the values.
- Store the result back into the memory operand and update the flags register with the result of the addition operation and update EIP to point beyond the constant's value.

Although it might seem possible to fetch the constant and the memory operand in the same step (since their values do not depend upon one another), the CPU can't actually do this (yet!) because it has only a single data bus. Since both of these values are coming from memory, we can't bring them into the CPU during the same step because the CPU uses the data bus to fetch these two values. In the next section you'll see how we can overcome this problem.

By overlapping various stages in the execution of these instructions we've been able to substantially reduce the number of steps (i.e., clock cycles) that the instructions need to complete execution. This process of executing various steps of the instruction in parallel with other steps is a major key to improving CPU performance without cranking up the clock speed on the chip. In this section we've seen how to speed up the execution of an instruction by doing many of the internal operations of that instruction in parallel. However, there's only so much to be gained from this approach. In this approach, the instructions themselves are still serialized (one instruction completes before the next instruction begins execution). Starting with the next section we'll start to see how to overlap the execution of adjacent instructions in order to save additional cycles.

4.8.1 The Prefetch Queue – Using Unused Bus Cycles

The key to improving the speed of a processor is to perform operations in parallel. If we were able to do two operations on each clock cycle, the CPU would execute instructions twice as fast when running at the same clock speed. However, simply deciding to execute two operations per clock cycle is not so easy. Many steps in the execution of an instruction share *functional units* in the CPU (functional units are groups of logic that perform a common operation, e.g., the ALU and the CU). A functional unit is only capable of one operation at a time. Therefore, you cannot do two operations that use the same functional unit concurrently (e.g., incrementing the EIP register and adding two values together). Another difficulty with doing certain operations concurrently is that one operation may depend on the other's result. For example, the two steps of the ADD instruction that involve adding two values and then storing their sum. You cannot store the sum into a register until after you've computed the sum. There are also some other resources the CPU cannot share between steps in an instruction. For example, there is only one data bus; the CPU cannot fetch an instruction opcode at the same time it is trying to store some data to memory. The trick in designing a CPU that executes several steps in parallel is to arrange those steps to reduce conflicts or add additional logic so the two (or more) operations can occur simultaneously by executing in different functional units.

Consider again the steps the MOV(mem/reg, reg) instruction requires:

- Fetch the instruction byte from memory.
- Update the EIP register to point at the next byte.
- Decode the instruction to see what it does.
- If required, fetch a displacement operand from memory.
- If required, update EIP to point beyond the displacement.
- Compute the address of the operand, if required (i.e., EBX+xxxx) .
- Fetch the operand.
- Store the fetched value into the destination register

The first operation uses the value of the EIP register (so we cannot overlap incrementing EIP with it) and it uses the bus to fetch the instruction opcode from memory. Every step that follows this one depends upon the opcode it fetches from memory, so it is unlikely we will be able to overlap the execution of this step with any other.

The second and third operations do not share any functional units, nor does decoding an opcode depend upon the value of the EIP register. Therefore, we can easily modify the control unit so that it increments the EIP register at the same time it decodes the instruction. This will shave one cycle off the execution of the MOV instruction.

The third and fourth operations above (decoding and optionally fetching the displacement operand) do not look like they can be done in parallel since you must decode the instruction to determine if it the CPU needs to fetch an operand from memory. However, we could design the CPU to go ahead and fetch the operand anyway, so that it's available if we need it. There is one problem with this idea, though, we must have the address of the operand to fetch (the value in the EIP register) and if we

must wait until we are done incrementing the EIP register before fetching this operand. If we are incrementing EIP at the same time we're decoding the instruction, we will have to wait until the next cycle to fetch this operand.

Since the next three steps are optional, there are several possible instruction sequences at this point:

#1 (step 4, step 5, step 6, and step 7) — e.g., MOV([ebx+1000], eax)

#2 (step 4, step 5, and step 7) — e.g., MOV(disp, eax) -- assume disp s address is 1000

#3 (step 6 and step 7) — e.g., MOV([ebx], eax)

#4 (step 7) — e.g., MOV(ebx, eax)

In the sequences above, step seven always relies on the previous steps in the sequence. Therefore, step seven cannot execute in parallel with any of the other steps. Step six also relies upon step four. Step five cannot execute in parallel with step four since step four uses the value in the EIP register, however, step five can execute in parallel with any other step. Therefore, we can shave one cycle off the first two sequences above as follows:

#1 (step 4, step 5/6, and step 7)

#2 (step 4, step 5/7)

#3 (step 6 and step 7)

#4 (step 7)

Of course, there is no way to overlap the execution of steps seven and eight in the MOV instruction since it must surely fetch the value before storing it away. By combining these steps, we obtain the following steps for the MOV instruction:

- Fetch the instruction byte from memory.
- Decode the instruction and update ip
- If required, fetch a displacement operand from memory.
- Compute the address of the operand, if required (i.e., ebx+xxxx) .
- Fetch the operand, if required update EIP to point beyond xxxx.
- Store the fetched value into the destination register

By adding a small amount of logic to the CPU, we've shaved one or two cycles off the execution of the MOV instruction. This simple optimization works with most of the other instructions as well.

Consider what happens with the MOV instruction above executes on a CPU with a 32-bit data bus. If the MOV instruction fetches an eight-bit displacement from memory, the CPU may actually wind up fetching the following three bytes after the displacement along with the displacement value (since the 32-bit data bus lets us fetch four bytes in a single bus cycle). The second byte on the data bus is actually the opcode of the next instruction. If we could save this opcode until the execution of the next instruction, we could shave a cycle of its execution time since it would not have to fetch the opcode byte. Furthermore, since the instruction decoder is idle while the CPU is executing the MOV instruction, we can actually decode the next instruction while the current instruction is executing, thereby shaving yet another cycle off the execution of the next instruction. This, effectively, overlaps a portion of the MOV instruction with the beginning of the execution of the next instruction, allowing additional parallelism.

Can we improve on this? The answer is yes. Note that during the execution of the MOV instruction the CPU is not accessing memory on every clock cycle. For example, while storing the data into the destination register the bus is idle. During time periods when the bus is idle we can pre-fetch instruction opcodes and operands and save these values for executing the next instruction.

The hardware to do this is the prefetch queue. Figure 4.4 shows the internal organization of a CPU with a prefetch queue. The Bus Interface Unit, as its name implies, is responsible for controlling access to the address and data busses. Whenever some component inside the CPU wishes to access main memory, it sends this request to the bus interface unit (or BIU) that acts as a "traffic cop" and handles simultaneous requests for bus access by different modules (e.g., the execution unit and the prefetch queue).

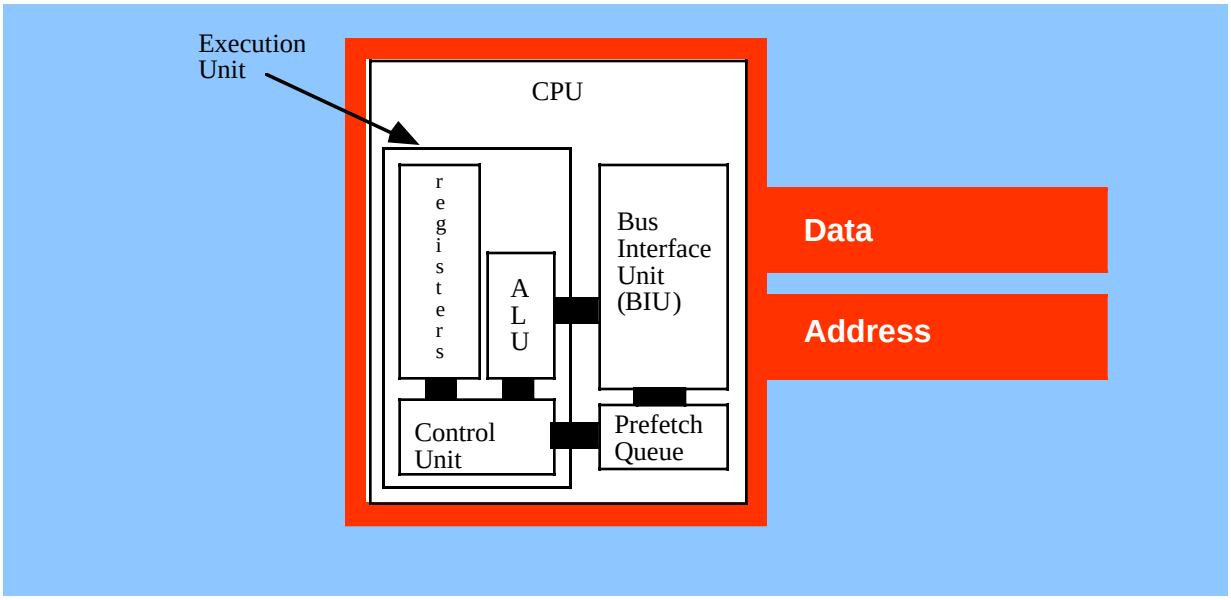

Figure 4.4 CPU Design with a Prefetch Queue

Whenever the execution unit is not using the Bus Interface Unit, the BIU can fetch additional bytes from the instruction stream. Whenever the CPU needs an instruction or operand byte, it grabs the next available byte from the prefetch queue. Since the BIU grabs four bytes at a time from memory (assuming a 32-bit data bus) and it generally consumes fewer than four bytes per clock cycle, any bytes the CPU would normally fetch from the instruction stream will already be sitting in the prefetch queue.

Note, however, that we're not guaranteed that all instructions and operands will be sitting in the prefetch queue when we need them. For example, consider the "JNZ Label;" instruction, if it transfers control to *Label*, will invalidate the contents of the prefetch queue. If this instruction appears at locations 400 and 401 in memory (it is a two-byte instruction), the prefetch queue will contain the bytes at addresses 402, 403, 404, 405, 406, 407, etc. If the target address of the JNZ instruction is 480, the bytes at addresses 402, 403, 404, etc., won't do us any good. So the system has to pause for a moment to fetch the double word at address 480 before it can go on.

Another improvement we can make is to overlap instruction decoding with the last step of the previous instruction. After the CPU processes the operand, the next available byte in the prefetch queue is an opcode, and the CPU can decode it in anticipation of its execution. Of course, if the current instruction modifies the EIP register then any time spent decoding the next instruction goes to waste, but since this occurs in parallel with other operations, it does not slow down the system (though it does require extra circuitry to do this).

The instruction execution sequence now assumes that the following events occur in the background:

CPU Prefetch Events:

- If the prefetch queue is not full (generally it can hold between eight and thirty-two bytes, depending on the processor) and the BIU is idle on the current clock cycle, fetch the next double word from memory at the address in EIP at the beginning of the clock cycle[15].

- If the instruction decoder is idle and the current instruction does not require an instruction operand, begin decoding the opcode at the front of the prefetch queue (if present), otherwise begin decoding the byte beyond the current operand in the prefetch queue (if present). If the desired byte is not in the prefetch queue, do not execute this event.

15.This operation fetches only a byte if ip contains an odd value.

Now let's reconsider our "mov(reg, reg);" instruction from the previous section. With the addition of the prefetch queue and the bus interface unit, fetching and decoding opcode bytes, as well as updating the EIP register, takes place in parallel with the previous instruction. Without the BIU and the prefetch queue, the "mov(reg, reg);" requires the following steps:

- Fetch the instruction byte from memory.
- Decode the instruction to see what it does.
- Fetch the source register and update the EIP register to point at the next byte.
- Store the fetched value into the destination register

However, now that we can overlap the instruction fetch and decode with the previous instruction, we now get the following steps:

- Fetch and Decode Instruction - overlapped with previous instruction
- Fetch the source register and update the EIP register to point at the next byte.
- Store the fetched value into the destination register

The instruction execution timings make a few optimistic assumptions, namely that the opcode is already present in the prefetch queue and that the CPU has already decoded it. If either case is not true, additional cycles will be necessary so the system can fetch the opcode from memory and/or decode the instruction.

Because they invalidate the prefetch queue, jump and conditional jump instructions (when actually taken) are much slower than other instructions. This is because the CPU cannot overlap fetching and decoding the opcode for the next instruction with the execution of the jump instruction since the opcode is (probably) not in the prefetch queue. Therefore, it may take several cycles after the execution of one of these instructions for the prefetch queue to recover and the CPU is decoding opcodes in parallel with the execution of previous instructions. The has one very important implication to your programs: if you want to write fast code, make sure to avoid jumping around in your program as much as possible.

Note that the conditional jump instructions only invalidate the prefetch queue if they actually make the jump. If the condition is false, they fall through to the next instruction and continue to use the values in the prefetch queue as well as any predecoded instruction opcodes. Therefore, if you can determine, while writing the program, which condition is most likely (e.g., less than vs. not less than), you should arrange your program so that the most common case falls through and conditional jump rather than take the branch.

Instruction size (in bytes) can also affect the performance of the prefetch queue. The longer the instruction, the faster the CPU will empty the prefetch queue. Instructions involving constants and memory operands tend to be the largest. If you place a string of these in a row, the CPU may wind up having to wait because it is removing instructions from the prefetch queue faster than the BIU is copying data to the prefetch queue. Therefore, you should attempt to use shorter instructions whenever possible since they will improve the performance of the prefetch queue.

Usually, including the prefetch queue improves performance. That's why Intel provides the prefetch queue on many models of the 80x86 family, from the 8088 on up. On these processors, the BIU is constantly fetching data for the prefetch queue whenever the program is not actively reading or writing data.

Prefetch queues work best when you have a wide data bus. The 8086 processor runs much faster than the 8088 because it can keep the prefetch queue full with fewer bus accesses. Don't forget, the CPU needs to use the bus for other purposes than fetching opcodes, displacements, and immediate constants. Instructions that access memory compete with the prefetch queue for access to the bus (and, therefore, have priority). If you have a sequence of instructions that all access memory, the prefetch queue may become empty if there are only a few bus cycles available for filling the prefetch queue during the execution of these instructions. Of course, once the prefetch queue is empty, the CPU must wait for the BIU to fetch new opcodes from memory, slowing the program.

A wider data bus allows the BIU to pull in more prefetch queue data in the few bus cycles available for this purpose, so it is less likely the prefetch queue will ever empty out with a wider data bus. Executing shorter instructions also helps keep the prefetch queue full. The reason is that the prefetch queue has time to refill itself with the shorter instructions. Moral of the story: when programming a processor with a prefetch queue, always use the shortest instructions possible to accomplish a given task.

4.8.2 Pipelining – Overlapping the Execution of Multiple Instructions

Executing instructions in parallel using a bus interface unit and an execution unit is a special case of pipelining. The 80x86 family, starting with the 80486, incorporates pipelining to improve performance. With just a few exceptions, we'll see that pipelining allows us to execute one instruction per clock cycle.

The advantage of the prefetch queue was that it let the CPU overlap instruction fetching and decoding with instruction execution. That is, while one instruction is executing, the BIU is fetching and decoding the next instruction. Assuming you're willing to add hardware, you can execute almost all operations in parallel. That is the idea behind pipelining.

4.8.2.1 A Typical Pipeline

Consider the steps necessary to do a generic operation:

- Fetch opcode.
- Decode opcode and (in parallel) prefetch a possible displacement or constant operand (or both)
- Compute complex addressing mode (e.g., [ebx+xxxx]), if applicable.
- Fetch the source value from memory (if a memory operand) and the destination register value (if applicable).
- Compute the result.
- Store result into destination register.

Assuming you're willing to pay for some extra silicon, you can build a little "mini-processor" to handle each of the above steps. The organization would look something like Figure 4.5.

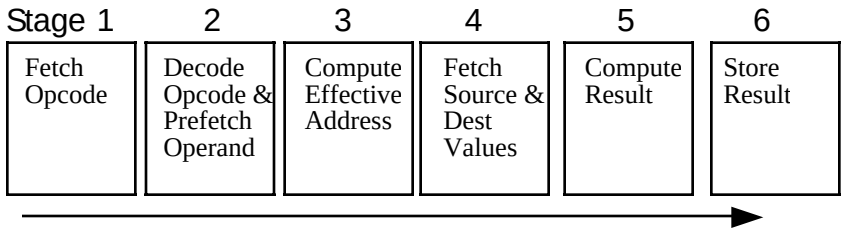

Figure 4.5 A Pipelined Implementation of Instruction Execution

Note how we've combined some stages from the previous section. For example, in stage four of Figure 4.5 the CPU fetches the source and destination operands in the same step. You can do this by putting multiple data paths inside the CPU (e.g., from the registers to the ALU) and ensuring that no two operands ever compete for simultaneous use of the data bus (i.e., no memory-to-memory operations).

If you design a separate piece of hardware for each stage in the pipeline above, almost all these steps can take place in parallel. Of course, you cannot fetch and decode the opcode for more than one instruction at the same time, but you can fetch one opcode while decoding the previous instruction. If you have an n-stage pipeline, you will usually have n instructions executing concurrently.

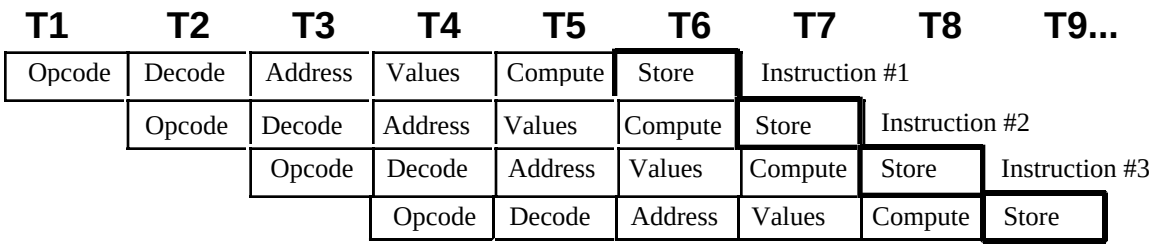

Figure 4.6 Instruction Execution in a Pipeline

Figure 4.6 shows pipelining in operatoin. T1, T2, T3, etc., represent consecutive “ticks” of the system clock. At T=T1 the CPU fetches the opcode byte for the first instruction.

At T=T2, the CPU begins decoding the opcode for the first instruction. In parallel, it fetches a block of bytes from the prefetch queue in the event the instruction has an operand. Since the first instruction no longer needs the opcode fetching circuitry, the CPU instructs it to fetch the opcode of the second instruction in parallel with the decoding of the first instruction. Note there is a minor conflict here. The CPU is attempting to fetch the next byte from the prefetch queue for use as an operand, at the same time it is fetching operand data from the prefetch queue for use as an opcode. How can it do both at once? You’ll see the solution in a few moments.

At T=T3 the CPU computes an operand address for the first instruction, if any. The CPU does nothing on the first instruction if it does not use an addressing mode requiring such computation. During T3, the CPU also decodes the opcode of the second instruction and fetches any necessary operand. Finally the CPU also fetches the opcode for the third instruction. With each advancing tick of the clock, another step in the execution of each instruction in the pipeline completes, and the CPU fetches yet another instruction from memory.

This process continues until at T=T6 the CPU completes the execution of the first instruction, computes the result for the second, etc., and, finally, fetches the opcode for the sixth instruction in the pipeline. The important thing to see is that after T=T5 the CPU completes an instruction on every clock cycle. Once the CPU fills the pipeline, it completes one instruction on each cycle. Note that this is true even if there are complex addressing modes to be computed, memory operands to fetch, or other operations which use cycles on a non-pipelined processor. All you need to do is add more stages to the pipeline, and you can still effectively process each instruction in one clock cycle.

A bit earlier you saw a small conflict in the pipeline organization. At T=T2, for example, the CPU is attempting to prefetch a block of bytes for an operand and at the same time it is trying to fetch the next opcode byte. Until the CPU decodes the first instruction it doesn’t know how many operands the instruction requires nor does it know their length. However, the CPU needs to know this information to determine the length of the instruction so it knows what byte to fetch as the opcode of the next instruction. So how can the pipeline fetch an instruction opcode in parallel with an address operand?

One solution is to disallow this. If an instruction as an address or constant operand, we simply delay the start of the next instruction (this is known as a *hazard* as you shall soon see). Unfortunately, many instructions have these additional operands, so this approach will have a substantial negative impact on the execution speed of the CPU.

The second solution is to throw (a lot) more hardware at the problem. Operand and constant sizes usually come in one, two, and four-byte lengths. Therefore, if we actually fetch three bytes from memory, at offsets one, three, and five, beyond the current opcode we are decoding, we know that one of these bytes will probably contain the opcode of the next instruction. Once we are through decoding the current instruction we know how long it will be and, therefore, we know the offset of the next opcode. We can use a simple data selector circuit to choose which of the three opcode bytes we want to use.

In actual practice, we have to select the next opcode byte from more than three candidates because 80x86 instructions take many different lengths. For example, an instruction that moves a 32-bit constant to a memory location can be ten or more bytes long. And there are instruction lengths for nearly every value between one and fifteen bytes. Also, some opcodes on the 80x86 are longer than one byte, so the CPU may have to fetch multiple bytes in order to properly decode the current instruction. Nevertheless, by throwing more hardware at the problem we can decode the current opcode at the same time we’re fetching the next.

4.8.2.2 Stalls in a Pipeline

Unfortunately, the scenario presented in the previous section is a little too simplistic. There are two drawbacks to that simple pipeline: bus contention among instructions and non-sequential program execution. Both problems may increase the average execution time of the instructions in the pipeline.

Bus contention occurs whenever an instruction needs to access some item in memory. For example, if a "mov(reg, mem);" instruction needs to store data in memory and a "mov(mem, reg);" instruction is reading data from memory, contention for the address and data bus may develop since the CPU will be trying to simultaneously fetch data and write data in memory.

One simplistic way to handle bus contention is through a *pipeline stall*. The CPU, when faced with contention for the bus, gives priority to the instruction furthest along in the pipeline. The CPU suspends fetching opcodes until the current instruction fetches (or stores) its operand. This causes the new instruction in the pipeline to take two cycles to execute rather than one (see Figure 4.7).

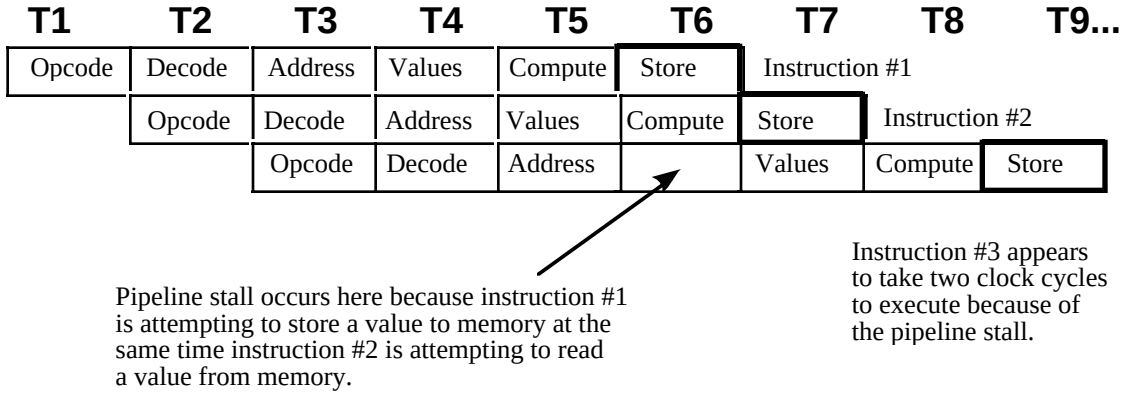

Figure 4.7 A Pipeline Stall

This example is but one case of bus contention. There are many others. For example, as noted earlier, fetching instruction operands requires access to the prefetch queue at the same time the CPU needs to fetch an opcode. Given the simple scheme above, it's unlikely that most instructions would execute at one clock per instruction (CPI).

Fortunately, the intelligent use of a cache system can eliminate many pipeline stalls like the ones discussed above. The next section on caching will describe how this is done. However, it is not always possible, even with a cache, to avoid stalling the pipeline. What you cannot fix in hardware, you can take care of with software. If you avoid using memory, you can reduce bus contention and your programs will execute faster. Likewise, using shorter instructions also reduces bus contention and the possibility of a pipeline stall.

What happens when an instruction *modifies* the EIP register? This, of course, implies that the next set of instructions to execute do not immediately follow the instruction that modifies EIP. By the time the instruction

```
JNZ     Label;
```

completes execution (assuming the zero flag is clear so the branch is taken), we've already started five other instructions and we're only one clock cycle away from the completion of the first of these. Obviously, the CPU must not execute those instructions or it will compute improper results.

The only reasonable solution is to *flush* the entire pipeline and begin fetching opcodes anew. However, doing so causes a severe execution time penalty. It will take six clock cycles (the length of the pipeline in our examples) before the next instruction completes execution. Clearly, you should avoid the use of instructions which interrupt the sequential execution of a program. This also shows another problem – pipeline length. The longer the pipeline is, the more you can accomplish per cycle in the system. However, lengthening a pipeline may slow a program if it jumps around quite a bit. Unfortunately, you cannot control the number of stages in the pipeline[16]. You can, however, control the number of transfer instructions which appear in your programs. Obviously you should keep these to a minimum in a pipelined system.

4.8.3 Instruction Caches – Providing Multiple Paths to Memory

System designers can resolve many problems with bus contention through the intelligent use of the prefetch queue and the cache memory subsystem. They can design the prefetch queue to buffer up data from the instruction stream, and they can design the cache with separate data and code areas. Both techniques can improve system performance by eliminating some conflicts for the bus.

The prefetch queue simply acts as a buffer between the instruction stream in memory and the opcode fetching circuitry. The prefetch queue works well when the CPU isn't constantly accessing memory. When the CPU isn't accessing memory, the BIU can fetch additional instruction opcodes for the prefetch queue. Alas, the pipelined 80x86 CPUs are constantly accessing memory since they fetch an opcode byte on every clock cycle. Therefore, the prefetch queue cannot take advantage of any "dead" bus cycles to fetch additional opcode bytes – there aren't any "dead" bus cycles. However, the prefetch queue is still valuable for a very simple reason: the BIU fetches multiple bytes on each memory access and most instructions are shorter. Without the prefetch queue, the system would have to explicitly fetch each opcode, even if the BIU had already "accidentally" fetched the opcode along with the previous instruction. With the prefetch queue, however, the system will not refetch any opcodes. It fetches them once and saves them for use by the opcode fetch unit.

For example, if you execute two one-byte instructions in a row, the BIU can fetch both opcodes in one memory cycle, freeing up the bus for other operations. The CPU can use these available bus cycles to fetch additional opcodes or to deal with other memory accesses.

Of course, not all instructions are one byte long. The 80x86 has a large number of different instruction sizes. If you execute several large instructions in a row, you're going to run slower. Once again we return to that same rule: *the fastest programs are the ones which use the shortest instructions*. If you can use shorter instructions to accomplish some task, do so.

Suppose, for a moment, that the CPU has two separate memory spaces, one for instructions and one for data, each with their own bus. This is called the *Harvard Architecture* since the first such machine was built at Harvard. On a Harvard machine there would be no contention for the bus. The BIU could continue to fetch opcodes on the instruction bus while accessing memory on the data/memory bus (see Figure 4.8),

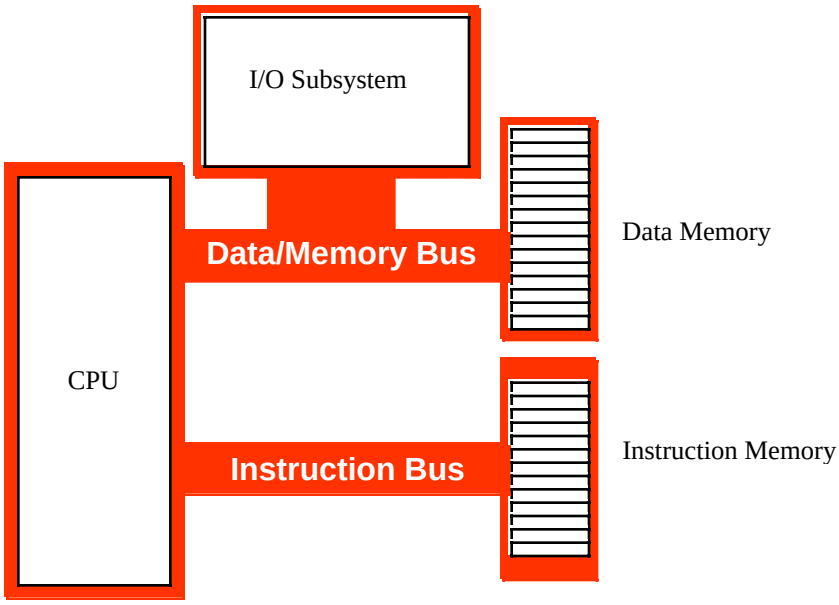

Figure 4.8 A Typical Harvard Machine

In the real world, there are very few true Harvard machines. The extra pins needed on the processor to support two physically separate busses increase the cost of the processor and introduce many other engineering problems. However, micropro-

16.Note, by the way, that the number of stages in an instruction pipeline varies among CPUs.

cessor designers have discovered that they can obtain many benefits of the Harvard architecture with few of the disadvantages by using separate on-chip caches for data and instructions. Advanced CPUs use an internal Harvard architecture and an external Von Neumann architecture. Figure 4.9 shows the structure of the 80x86 with separate data and instruction caches.

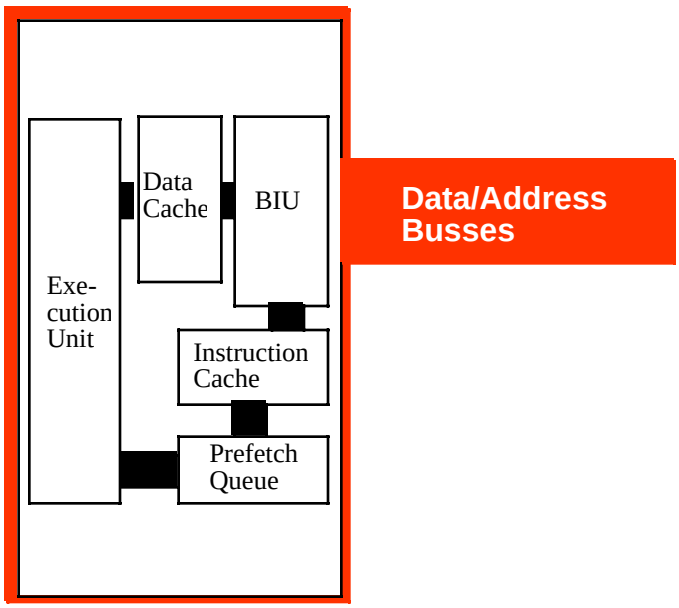

Figure 4.9 Using Separate Code and Data Caches

Each path inside the CPU represents an independent bus. Data can flow on all paths concurrently. This means that the prefetch queue can be pulling instruction opcodes from the instruction cache while the execution unit is writing data to the data cache. Now the BIU only fetches opcodes from memory whenever it cannot locate them in the instruction cache. Likewise, the data cache buffers memory. The CPU uses the data/address bus only when reading a value which is not in the cache or when flushing data back to main memory.

Although you cannot control the presence, size, or type of cache on a CPU, as an assembly language programmer you must be aware of how the cache operates to write the best programs. On-chip level one instruction caches are generally quite small (8,192 bytes on the 80486, for example). Therefore, the shorter your instructions, the more of them will fit in the cache (getting tired of "shorter instructions" yet?). The more instructions you have in the cache, the less often bus contention will occur. Likewise, using registers to hold temporary results places less strain on the data cache so it doesn't need to flush data to memory or retrieve data from memory quite so often. *Use the registers wherever possible!*

4.8.4 Hazards

There is another problem with using a pipeline: the data hazard. Let's look at the execution profile for the following instruction sequence:

```
mov( Somevar, ebx );
mov( [ebx], eax );
```

When these two instructions execute, the pipeline will look something like shown in Figure 4.10:

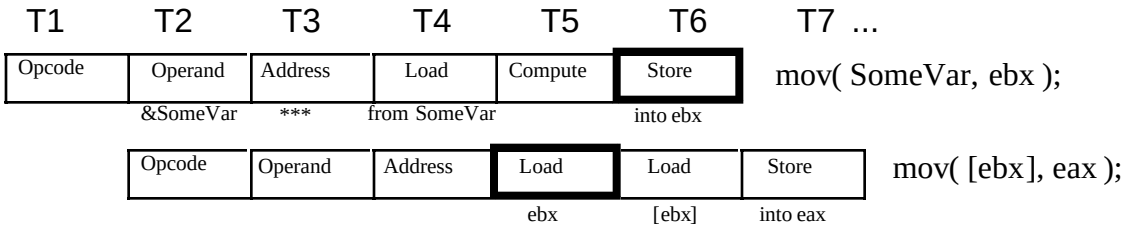

Figure 4.10 A Data Hazard

Note a major problem here. These two instructions fetch the 32 bit value whose address appears at location &*SomeVar* in memory. *But this sequence of instructions won't work properly!* Unfortunately, the second instruction has already used the value in EBX before the first instruction loads the contents of memory location &*SomeVar* (T4 & T6 in the diagram above).

CISC processors, like the 80x86, handle hazards automatically[17]. However, they will stall the pipeline to synchronize the two instructions. The actual execution would look something like shown in Figure 4.11.

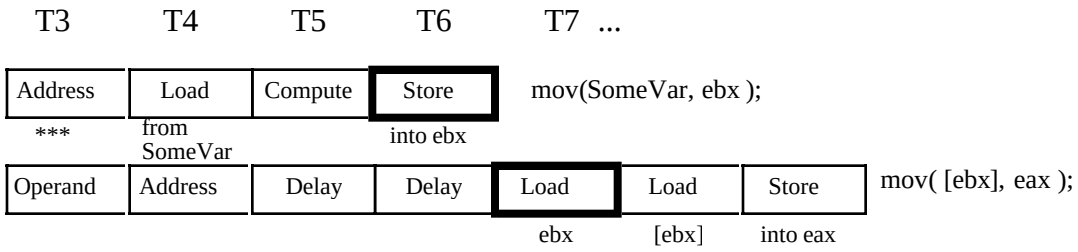

Figure 4.11 How the 80x86 Handles a Data Hazard

By delaying the second instruction two clock cycles, the CPU guarantees that the load instruction will load EAX from the proper address. Unfortunately, the second load instruction now executes in three clock cycles rather than one. However, requiring two extra clock cycles is better than producing incorrect results. Fortunately, you can reduce the impact of hazards on execution speed within your software.

Note that the data hazard occurs when the source operand of one instruction was a destination operand of a previous instruction. There is nothing wrong with loading EBX from *SomeVar* and then loading EAX from [EBX], *unless they occur one right after the other.* Suppose the code sequence had been:

```
mov( 2000, ecx );
mov( SomeVar, ebx );
mov( [ebx], eax );
```

We could reduce the effect of the hazard that exists in this code sequence by simply *rearranging the instructions.* Let's do that and obtain the following:

```
mov( SomeVar, ebx );
mov( 2000, ecx );
mov( [ebx], eax );
```

Now the "mov([ebx], eax);" instruction requires only one additional clock cycle rather than two. By inserting yet another instruction between the "mov(SomeVar, ebx);" and the "mov([ebx], eax);" instructions you can eliminate the effects of the hazard altogether[18].

17.Some RISC chips do not. If you tried this sequence on certain RISC chips you would get an incorrect answer.

On a pipelined processor, the order of instructions in a program may dramatically affect the performance of that program. Always look for possible hazards in your instruction sequences. Eliminate them wherever possible by rearranging the instructions.

In addition to data hazards, there are also *control hazards*. We've actually discussed control hazards already, although we did not refer to them by that name. A control hazard occurs whenever the CPU branches to some new location in memory and the CPU has to flush the following instructions following the branch that are in various stages of execution.

4.8.5 Superscalar Operation– Executing Instructions in Parallel

With the pipelined architecture we could achieve, at best, execution times of one CPI (clock per instruction). Is it possible to execute instructions faster than this? At first glance you might think, "Of course not, we can do at most one operation per clock cycle. So there is no way we can execute more than one instruction per clock cycle." Keep in mind however, that a single instruction is *not* a single operation. In the examples presented earlier each instruction has taken between six and eight operations to complete. By adding seven or eight separate units to the CPU, we could effectively execute these eight operations in one clock cycle, yielding one CPI. If we add more hardware and execute, say, 16 operations at once, can we achieve 0.5 CPI? The answer is a qualified "yes." A CPU including this additional hardware is a *superscalar* CPU and can execute more than one instruction during a single clock cycle. The 80x86 family began supporting superscalar execution with the introduction of the Pentium processor.

A superscalar CPU has, essentially, several execution units (see Figure 4.12). If it encounters two or more instructions in the instruction stream (i.e., the prefetch queue) which can execute independently, it will do so.

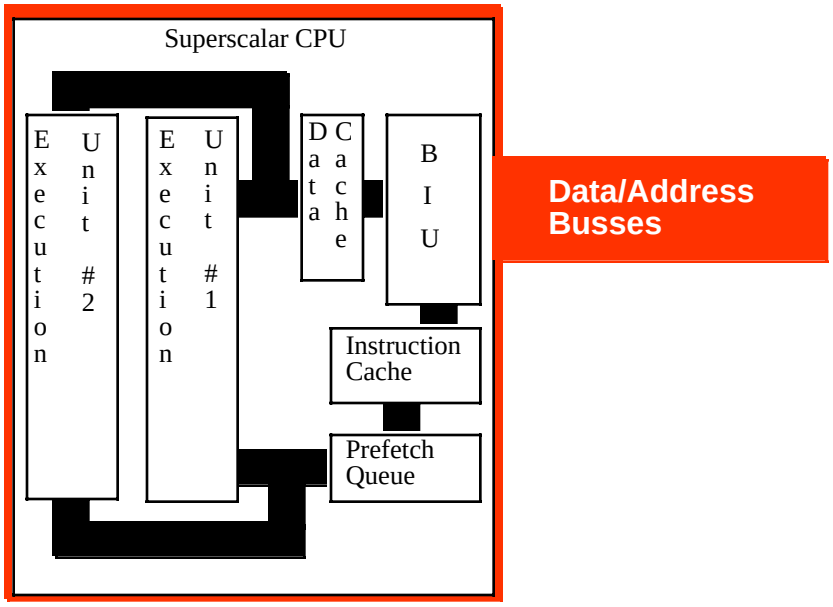

Figure 4.12 A CPU that Supports Superscalar Operation

There are a couple of advantages to going superscalar. Suppose you have the following instructions in the instruction stream:

```
mov( 1000, eax );
mov( 2000, ebx );
```

18. Of course, any instruction you insert at this point must *not* modify the values in the eax and ebx registers. Also note that the examples in this section are contrived to demonstrate pipeline stalls. Actual 80x86 CPUs have additional circuitry to help reduce the effect of pipeline stalls on the execution time.

If there are no other problems or hazards in the surrounding code, and all six bytes for these two instructions are currently in the prefetch queue, there is no reason why the CPU cannot fetch and execute both instructions in parallel. All it takes is extra silicon on the CPU chip to implement two execution units.

Besides speeding up independent instructions, a superscalar CPU can also speed up program sequences that have hazards. One limitation of superscalar CPU is that once a hazard occurs, the offending instruction will completely stall the pipeline. Every instruction which follows will also have to wait for the CPU to synchronize the execution of the instructions. With a superscalar CPU, however, instructions following the hazard may continue execution through the pipeline as long as they don't have hazards of their own. This alleviates (though does not eliminate) some of the need for careful instruction scheduling.

As an assembly language programmer, the way you write software for a superscalar CPU can dramatically affect its performance. First and foremost is that rule you're probably sick of by now: *use short instructions*. The shorter your instructions are, the more instructions the CPU can fetch in a single operation and, therefore, the more likely the CPU will execute faster than one CPI. Most superscalar CPUs do not completely duplicate the execution unit. There might be multiple ALUs, floating point units, etc. This means that certain instruction sequences can execute very quickly while others won't. You have to study the exact composition of your CPU to decide which instruction sequences produce the best performance.

4.8.6 Out of Order Execution

In a standard superscalar CPU it is the programmer's (or compiler's) responsibility to schedule (arrange) the instructions to avoid hazards and pipeline stalls. Fancier CPUs can actually remove some of this burden and improve performance by automatically rescheduling instructions while the program executes. To understand how this is possible, consider the following instruction sequence:

```
mov( SomeVar, ebx );
mov( [ebx], eax );
mov( 2000, ecx );
```

A data hazard exists between the first and second instructions above. The second instruction must delay until the first instruction completes execution. This introduces a pipeline stall and increases the running time of the program. Typically, the stall affects every instruction that follows. However, note that the third instruction's execution does not depend on the result from either of the first two instructions. Therefore, there is no reason to stall the execution of the "mov(2000, ecx);" instruction. It may continue executing while the second instruction waits for the first to complete. This technique, appearing in later members of the Pentium line, is called "out of order execution" because the CPU completes the execution of some instruction prior to the execution of previous instructions appearing in the code stream.

Clearly, the CPU may only execute instruction out of sequence if doing so produces exactly the same results as in-order execution. While there a lots of little technical issues that make this problem a little more difficult than it seems, with enough engineering effort it is quite possible to implement this feature.

Although you might think that this extra effort is not worth it (why not make it the programmer's or compiler's responsibility to schedule the instructions) there are some situations where out of order execution will improve performance that static scheduling could not handle.

4.8.7 Register Renaming

One problem that hampers the effectiveness of superscalar operation on the 80x86 CPU is the 80x86's limited number of general purpose registers. Suppose, for example, that the CPU had four different pipelines and, therefore, was capable of executing four instructions simultaneously. Actually achieving four instructions per clock cycle would be very difficult because most instructions (that can execute simultaneously with other instructions) operate on two register operands. For four instructions to execute concurrently, you'd need four separate destination registers and four source registers (and the two sets of registers must be disjoint, that is, a destination register for one instruction cannot be the source of another). CPUs that have lots of registers can handle this task quite easily, but the limited register set of the 80x86 makes this difficult. Fortunately, there is a way to alleviate part of the problem: through *register renaming*.

Register renaming is a sneaky way to give a CPU more registers than it actually has. Programmers will not have direct access to these extra registers, but the CPU can use these additional register to prevent hazards in certain cases. For example, consider the following short instruction sequence:

```
mov( 0, eax );
mov( eax, i );
mov( 50, eax );
mov( eax, j );
```

Clearly a data hazard exists between the first and second instructions and, likewise, a data hazard exists between the third and fourth instructions in this sequence. Out of order execution in a superscalar CPU would normally allow the first and third instructions to execute concurrently and then the second and fourth instructions could also execute concurrently. However, a data hazard, of sorts, also exists between the first and third instructions since they use the same register. The programmer could have easily solved this problem by using a different register (say EBX) for the third and fourth instructions. However, let's assume that the programmer was unable to do this because the other registers are all holding important values. Is this sequence doomed to executing in four cycles on a superscalar CPU that should only require two?

One advanced trick a CPU can employ is to create a bank of registers for each of the general purpose registers on the CPU. That is, rather than having a single EAX register, the CPU could support an array of EAX registers; let's call these registers EAX[0], EAX[1], EAX[2], etc. Similarly, you could have an array of each of the registers, so we could also have EBX[0]..EBX[n], ECX[0]..ECX[n], etc. Now the instruction set does not give the programmer the ability to select one of these specific register array elements for a given instruction, but the CPU can automatically choose a different register array element if doing so would not change the overall computation and doing so could speed up the execution of the program. For example, consider the following sequence (with register array elements automatically chosen by the CPU):

```
mov( 0, eax[0] );
mov( eax[0], i );
mov( 50, eax[1] );
mov( eax[1], j );
```

Since EAX[0] and EAX[1] are different registers, the CPU can execute the first and third instructions concurrently. Likewise, the CPU can execute the second and fourth instructions concurrently.

The code above provides an example of *register renaming*. Dynamically, the CPU automatically selects one of several different elements from a register array in order to prevent data hazards. Although this is a simple example, and different CPUs implement register renaming in many different ways, this example does demonstrate how the CPU can improve performance in certain instances through the use of this technique.

4.8.8 Very Long Instruction Word Architecture (VLIW)

Superscalar operation attempts to schedule, in hardware, the execution of multiple instructions simultaneously. Another technique that Intel is using in their IA-64 architecture is the use of very long instruction words, or VLIW. In a VLIW computer system, the CPU fetches a large block of bytes (41 in the case of the IA-64 Itanium CPU) and decodes and executes this block all at once. This block of bytes usually contains two or more instructions (three in the case of the IA-64). VLIW computing requires the programmer or compiler to properly schedule the instructions in each block (so there are no hazards or other conflicts), but if properly scheduled, the CPU can execute three or more instructions per clock cycle.

The Intel IA-64 Architecture is not the only computer system to employ a VLIW architecture. Transmeta's Crusoe processor family also uses a VLIW architecture. The Crusoe processor is different than the IA-64 architecture insofar as it does not support native execution of IA-32 instructions. Instead, the Crusoe processor dynamically translates 80x86 instructions to Crusoe's VLIW instructions. This "code morphing" technology results in code running about 50% slower than native code, though the Crusoe processor has other advantages.

We will not consider VLIW computing any further since the IA-32 architecture does not support it. But keep this architectural advance in mind if you move towards the IA-64 family or the Crusoe family.

4.8.9 Parallel Processing

Most of the techniques for improving CPU performance via architectural advances involve the parallel (overlapped) execution of instructions. Most of the techniques of this chapter are transparent to the programmer. That is, the programmer does not have to do anything special to take minimal advantage of the parallel operation of pipelines and superscalar operations. True, if programmers are aware of the underlying architecture they can write code that runs even faster, but these architectural advances often improve performance even if programmers do not write special code to take advantage of them.

The only problem with this approach (attempting to dynamically parallelize an inherently sequential program) is that there is only so much you can do to parallelize a program that requires sequential execution for proper operation (which covers most programs). To truly produce a parallel program, the programmer must specifically write parallel code; of course, this does require architectural support from the CPU. This section and the next touches on the types of support a CPU can provide.

Typical CPUs use what is known as the SISD model: *Single Instruction, Single Data*. This means that the CPU executes one instruction at a time that operates on a single piece of data[19]. Two common parallel models are the so-called SIMD (*Single Instruction, Multiple Data*) and MIMD (*Multiple Instruction, Multiple Data*) models. As it turns out, x86 systems can support both of these parallel execution models.

In the SIMD model, the CPU executes a single instruction stream, just like the standard SISD model. However, the CPU executes the specified operation on multiple pieces of data concurrently rather than a single data object. For example, consider the 80x86 ADD instruction. This is a SISD instruction that operates on (that is, produces) a single piece of data; true, the instruction fetches values from two source operands and stores a sum into a destination operand but the end result is that the ADD instruction will only produce a single sum. An SIMD version of ADD, on the other hand, would compute the sum of several values simultaneously. The Pentium III's MMX and SIMD instruction extensions operate in exactly this fashion. With an MMX instruction, for example, you can add up to eight separate pairs of values with the execution of a single instruction. The aptly named SIMD instruction extensions operate in a similar fashion.

Note that SIMD instructions are only useful in specialized situations. Unless you have an algorithm that can take advantage of SIMD instructions, they're not that useful. Fortunately, high-speed 3-D graphics and multimedia applications benefit greatly from these SIMD (and MMX) instructions, so their inclusion in the 80x86 CPU offers a huge performance boost for these important applications.

The MIMD model uses multiple instructions, operating on multiple pieces of data (usually one instruction per data object, though one of these instructions could also operate on multiple data items). These multiple instructions execute independently of one another. Therefore, it's very rare that a single program (or, more specifically, a single thread of execution) would use the MIMD model. However, if you have a multiprogramming environment with multiple programs attempting to execute concurrently in memory, the MIMD model does allow each of those programs to execute their own code stream concurrently. This type of parallel system is usually called a multiprocessor system. Multiprocessor systems are the subject of the next section.

The common computation models are SISD, SIMD, and MIMD. If you're wondering if there is a MISD model (Multiple Instruction, Single Data) the answer is no. Such an architecture doesn't really make sense.

4.8.10 Multiprocessing

Pipelining, superscalar operation, out of order execution, and VLIW design are techniques CPU designers use in order to execute several operations in parallel. These techniques support *fine-grained parallelism*[20] and are useful for speeding up adjacent instructions in a computer system. If adding more functional units increases parallelism (and, therefore, speeds up the system), you might wonder what would happen if you added two CPUs to the system. This technique, known as *multiprocessing*, can improve system performance, though not as uniformly as other techniques. As noted in the previous section, a multiprocessor system uses the MIMD parallel execution model.

The techniques we've considered to this point don't require special programming to realize a performance increase. True, if you do pay attention you will get better performance; but no special programming is necessary to activate these features. Multiprocessing, on the other hand, doesn't help a program one bit unless that program was specifically written to use multi-

19. We will ignore the parallelism provided by pipelining and superscalar operation in this discussion.
20. For our purposes, fine-grained parallelism means that we are executing adjacent program instructions in parallel.

processor (or runs under an O/S specfically written to support multiprocessing). If you build a system with two CPUs, those CPUs cannot trade off executing alternate instructions within a program. In fact, it is very expensive (timewise) to switch the execution of a program from one processor to another. Therefore, multiprocessor systems are really only effective in a system that execute multiple programs concurrently (i.e., a multitasking system)[21]. To differentiate this type of parallelism from that afforded by pipelining and superscalar operation, we'll call this kind of parallelism *coarse-grained parallelism*.

Adding multiple processors to a system is not as simple as wiring the processor to the motherboard. A big problem with multiple processors is the *cache coherency* problem. To understand this problem, consider two separate programs running on separate processors in a multiprocessor system. Suppose also that these two processor communicate with one another by writing to a block of shared physical memory. Unfortunately, when CPU #1 writes to this block of addresses the CPU caches the data up and might not actually write the data to physical memory for some time. Simultaneously, CPU #2 might be attempting to read this block of shared memory but winds up reading the data out of its local cache rather than the data that CPU #1 wrote to the block of shared memory (assuming the data made it out of CPU #1's local cache). In order for these two functions to operate properly, the two CPU's must communicate writes to common memory addresses in cache between themselves. This is a very complex and involved process.

Currently, the Pentium III and IV processors directly support cache updates between two CPUs in a system. Intel also builds a more expensive processor, the XEON, that supports more than two CPUs in a system. However, one area where the RISC CPUs have a big advantage over Intel is in the support for multiple processors in a system. While Intel systems reach a point of diminishing returns at about 16 processors, Sun SPARC and other RISC processors easily support 64-CPU systems (with more arriving, it seems, every day). This is why large databases and large web server systems tend to use expensive UNIX-based RISC systems rather than x86 systems.

4.9 Putting It All Together

The performance of modern CPUs is intrinsically tied to the architecture of that CPU. Over the past half century there have been many major advances in CPU design that have dramatically improved preformance. Although the clock frequency has improved by over four orders of magnitude during this time period, other improvements have added one or two orders of magnitude improvement as well. Over the 80x86's lifetime, performance has improved 10,000-fold.

Unfortunately, the 80x86 family has just about pushed the limits of what it can achieve by extending the architecture. Perhaps another order of manitude is possible, but Intel is reaching the point of diminishing returns. Having realized this, Intel has chosen to implement a new architecture using VLIW for their IA-64 family. Only time will prove whether their approach is the correct one, but most people believe that the IA-32 has reached the end of its lifetime. On the other hand, people have been announcing the death of the IA-32 for the past decade, so we'll see what happens now.

21. Technically, it only needs to execute multiple threads concurrently, but we ll ignore this distinction here since the technical distinction between threads and programs/processes is beyond the scope of this chapter.