This observation means that only LOAD and STORE instructions should reference memory. All other instructions should operate only on registers.

Provide Plenty of Registers

Since accessing memory is relatively slow, many registers (at least 32) need to be provided, so that once a word is fetched, it can be kept in a register until it is no longer needed. Running out of registers and having to flush them back to memory only to later reload them is undesirable and should be avoided as much as possible. The best way to accomplish this is to have enough registers.

2.1.5 Instruction-Level Parallelism

Computer architects are constantly striving to improve performance of the machines they design. Making the chips run faster by increasing their clock speed is one way, but for every new design, there is a limit to what is possible by brute force at that moment in history. Consequently, most computer architects look to parallelism (doing two or more things at once) as a way to get even more performance for a given clock speed.

Parallelism comes in two general forms, namely, instruction-level parallelism and processor-level parallelism. In the former, parallelism is exploited within individual instructions to get more instructions/sec out of the machine. In the latter, multiple CPUs work together on the same problem. Each approach has its own merits. In this section we will look at instruction-level parallelism; in the next one, we will look at processor-level parallelism.

Pipelining

It has been known for years that the actual fetching of instructions from memory is a major bottleneck in instruction execution speed. To alleviate this problem, computers going back at least as far as the IBM Stretch (1959) have had the ability to fetch instructions from memory in advance, so they would be there when they were needed. These instructions were stored in a special set of registers called the **prefetch buffer**. This way, when an instruction was needed, it could usually be taken from the prefetch buffer rather than waiting for a memory read to complete.

In effect, prefetching divides instruction execution into two parts: fetching and actual execution. The concept of a **pipeline** carries this strategy much further. Instead of being divided into only two parts, instruction execution is often divided into many (often a dozen or more) parts, each one handled by a dedicated piece of hardware, all of which can run in parallel.

Figure 2-4(a) illustrates a pipeline with five units, also called **stages**. Stage 1 fetches the instruction from memory and places it in a buffer until it is needed. Stage 2 decodes the instruction, determining its type and what operands it needs.

Stage 3 locates and fetches the operands, either from registers or from memory. Stage 4 actually does the work of carrying out the instruction, typically by running the operands through the data path of Fig. 2-2. Finally, stage 5 writes the result back to the proper register.

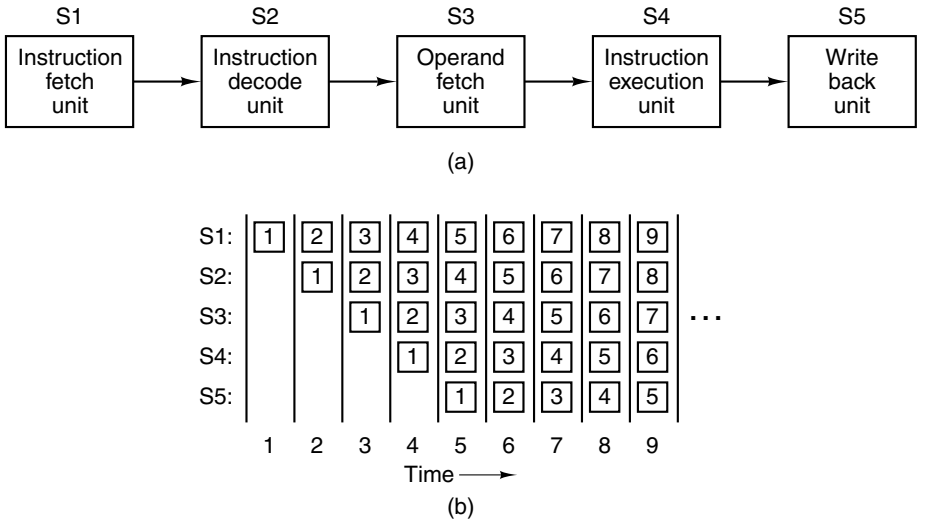

Figure 2-4. (a) A five-stage pipeline. (b) The state of each stage as a function of time. Nine clock cycles are illustrated.

In Fig. 2-4(b) we see how the pipeline operates as a function of time. During clock cycle 1, stage S1 is working on instruction 1, fetching it from memory. During cycle 2, stage S2 decodes instruction 1, while stage S1 fetches instruction 2. During cycle 3, stage S3 fetches the operands for instruction 1, stage S2 decodes instruction 2, and stage S1 fetches the third instruction. During cycle 4, stage S4 executes instruction 1, S3 fetches the operands for instruction 2, S2 decodes instruction 3, and S1 fetches instruction 4. Finally, in cycle 5, S5 writes the result of instruction 1 back, while the other stages work on the following instructions.

Let us consider an analogy to clarify the concept of pipelining. Imagine a cake factory in which the baking of the cakes and the packaging of the cakes for shipment are separated. Suppose that the shipping department has a long conveyor belt with five workers (processing units) lined up along it. Every 10 sec (the clock cycle), worker 1 places an empty cake box on the belt. The box is carried down to worker 2, who places a cake in it. A little later, the box arrives at worker 3's station, where it is closed and sealed. Then it continues to worker 4, who puts a label on the box. Finally, worker 5 removes the box from the belt and puts it in a large container for later shipment to a supermarket. Basically, this is the way computer pipelining works, too: each instruction (cake) goes through several processing steps before emerging completed at the far end.

Getting back to our pipeline of Fig. 2-4, suppose that the cycle time of this machine is 2 nsec. Then it takes 10 nsec for an instruction to progress all the way through the five-stage pipeline. At first glance, with an instruction taking 10 nsec, it might appear that the machine can run at 100 MIPS, but in fact it does much better than this. At every clock cycle (2 nsec), one new instruction is completed, so the actual rate of processing is 500 MIPS, not 100 MIPS.

Pipelining allows a trade-off between **latency** (how long it takes to execute an instruction), and **processor bandwidth** (how many MIPS the CPU has). With a cycle time of T nsec, and n stages in the pipeline, the latency is nT nsec because each instruction passes through n stages, each of which takes T nsec.

Since one instruction completes every clock cycle and there are $10^9/T$ clock cycles/second, the number of instructions executed per second is $10^9/T$. For example, if $T = 2$ nsec, 500 million instructions are executed each second. To get the number of MIPS, we have to divide the instruction execution rate by 1 million to get $(10^9/T)/10^6 = 1000/T$ MIPS. Theoretically, we could measure instruction execution rate in BIPS instead of MIPS, but nobody does that, so we will not either.

Superscalar Architectures

If one pipeline is good, then surely two pipelines are better. One possible design for a dual pipeline CPU, based on Fig. 2-4, is shown in Fig. 2-5. Here a single instruction fetch unit fetches pairs of instructions together and puts each one into its own pipeline, complete with its own ALU for parallel operation. To be able to run in parallel, the two instructions must not conflict over resource usage (e.g., registers), and neither must depend on the result of the other. As with a single pipeline, either the compiler must guarantee this situation to hold (i.e., the hardware does not check and gives incorrect results if the instructions are not compatible), or conflicts must be detected and eliminated during execution using extra hardware.

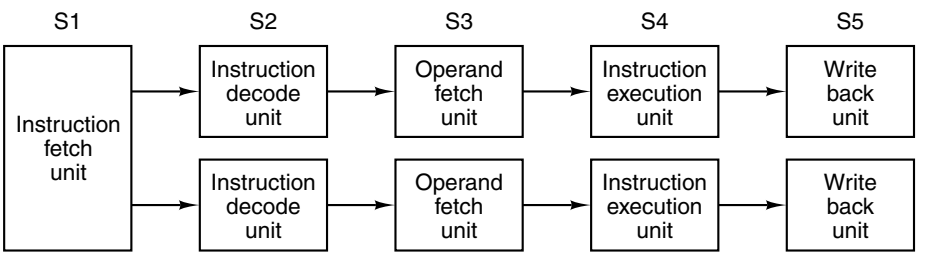

Figure 2-5. Dual five-stage pipelines with a common instruction fetch unit.

Although pipelines, single or double, were originally used on RISC machines (the 386 and its predecessors did not have any), starting with the 486 Intel began

introducing data pipelines into its CPUs. The 486 had one pipeline and the original Pentium had two five-stage pipelines roughly as in Fig. 2-5, although the exact division of work between stages 2 and 3 (called decode-1 and decode-2) was slightly different than in our example. The main pipeline, called the **u pipeline**, could execute an arbitrary Pentium instruction. The second pipeline, called the **v pipeline**, could execute only simple integer instructions (and also one simple floating-point instruction—FXCH).

Fixed rules determined whether a pair of instructions were compatible so they could be executed in parallel. If the instructions in a pair were not simple enough or incompatible, only the first one was executed (in the u pipeline). The second one was then held and paired with the instruction following it. Instructions were always executed in order. Thus Pentium-specific compilers that produced compatible pairs could produce faster-running programs than older compilers. Measurements showed that a Pentium running code optimized for it was exactly twice as fast on integer programs as a 486 running at the same clock rate (Pountain, 1993). This gain could be attributed entirely to the second pipeline.

Going to four pipelines is conceivable, but doing so duplicates too much hardware (computer scientists, unlike folklore specialists, do not believe in the number three). Instead, a different approach is used on high-end CPUs. The basic idea is to have just a single pipeline but give it multiple functional units, as shown in Fig. 2-6. For example, the Intel Core architecture has a structure similar to this figure. It will be discussed in Chap. 4. The term **superscalar architecture** was coined for this approach in 1987 (Agerwala and Cocke, 1987). Its roots, however, go back more than 40 years to the CDC 6600 computer. The 6600 fetched an instruction every 100 nsec and passed it off to one of 10 functional units for parallel execution while the CPU went off to get the next instruction.

The definition of "superscalar" has evolved somewhat over time. It is now used to describe processors that issue multiple instructions—often four or six—in a single clock cycle. Of course, a superscalar CPU must have multiple functional units to hand all these instructions to. Since superscalar processors generally have one pipeline, they tend to look like Fig. 2-6.

Using this definition, the 6600 was technically not superscalar because it issued only one instruction per cycle. However, the effect was almost the same: instructions were issued at a much higher rate than they could be executed. The conceptual difference between a CPU with a 100-nsec clock that issues one instruction every cycle to a group of functional units and a CPU with a 400-nsec clock that issues four instructions per cycle to the same group of functional units is very small. In both cases, the key idea is that the issue rate is much higher than the execution rate, with the workload being spread across a collection of functional units.

Implicit in the idea of a superscalar processor is that the S3 stage can issue instructions considerably faster than the S4 stage is able to execute them. If the S3 stage issued an instruction every 10 nsec and all the functional units could do their work in 10 nsec, no more than one would ever be busy at once, negating the whole

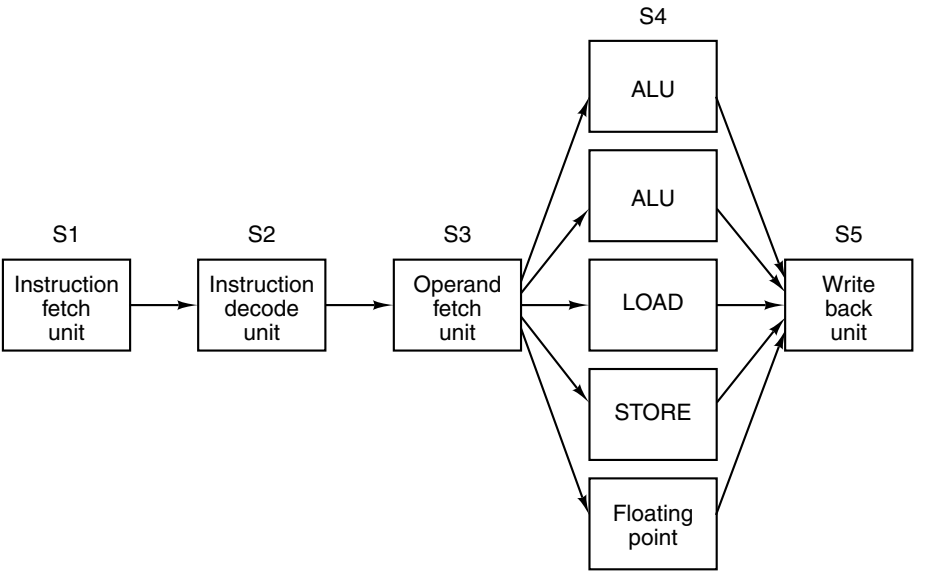

Figure 2-6. A superscalar processor with five functional units.

idea. In reality, most of the functional units in stage 4 take appreciably longer than one clock cycle to execute, certainly the ones that access memory or do floating-point arithmetic. As can be seen from the figure, it is possible to have multiple ALUs in stage S4.

2.1.6 Processor-Level Parallelism

The demand for ever faster computers seems to be insatiable. Astronomers want to simulate what happened in the first microsecond after the big bang, economists want to model the world economy, and teenagers want to play 3D interactive multimedia games over the Internet with their virtual friends. While CPUs keep getting faster, eventually they are going to run into the problems with the speed of light, which is likely to stay at 20 cm/nanosecond in copper wire or optical fiber, no matter how clever Intel's engineers are. Faster chips also produce more heat, whose dissipation is a huge problem. In fact, the difficulty of getting rid of the heat produced is the main reason CPU clock speeds have stagnated in the past decade.

Instruction-level parallelism helps a little, but pipelining and superscalar operation rarely win more than a factor of five or ten. To get gains of 50, 100, or more, the only way is to design computers with multiple CPUs, so we will now take a look at how some of these are organized.

Data Parallel Computers

A substantial number of problems in computational domains such as the physical sciences, engineering, and computer graphics involve loops and arrays, or otherwise have a highly regular structure. Often the same calculations are performed repeatedly on many different sets of data. The regularity and structure of these programs makes them especially easy targets for speed-up through parallel execution. Two primary methods have been used to execute these highly regular programs quickly and efficiently: SIMD processors and vector processors. While these two schemes are remarkably similar in most ways, ironically, the first is generally thought of as a parallel computer while the second is considered an ex extension to a single processor.

Data parallel computers have found many successful applications as a consequence of their remarkable efficiency. They are able to produce significant computational power with fewer transistors than alternative approaches. Gordon Moore (of Moore's law) famously noted that silicon costs about $1 billion per acre (4047 square meters). Thus, the more computational muscle that can be squeezed out of that acre of silicon, the more money a computer company can make selling silicon. Data parallel processors are one of the most efficient means to squeeze performance out of silicon. Because all of the processors are running the same instruction, the system needs only one "brain" controlling the computer. Consequently, the processor needs only one fetch stage, one decode stage, and one set of control logic. This is a huge saving in silicon that gives data parallel computers a big edge over other processors, as long as the software they are running is highly regular with lots of parallelism.

A **Single Instruction-stream Multiple Data-stream** or **SIMD processor** consists of a large number of identical processors that perform the same sequence of instructions on different sets of data. The world's first SIMD processor was the University of Illinois ILLIAC IV computer (Bouknight et al., 1972). The original ILLIAC IV design consisted of four quadrants, each quadrant having an 8×8 square grid of processor/memory elements. A single control unit per quadrant broadcast a single instruction to all processors, which was executed by all processors in lockstep each using its own data from its own memory. Owing to funding constraints only one 50 megaflops (million floating-point operations per second) quadrant was ever built; had the entire 1-gigaflop machine been completed, it would have doubled the computing power of the entire world.

Modern graphics processing units (GPUs) heavily rely on SIMD processing to provide massive computational power with few transistors. Graphics processing lends itself to SIMD processors because most of the algorithms are highly regular, with repeated operations on pixels, vertices, textures, and edges. Fig. 2-7 shows the SIMD processor at the core of the Nvidia Fermi GPU. A Fermi GPU contains up to 16 SIMD stream multiprocessors (SM), with each SM containing 32 SIMD processors. Each cycle, the scheduler selects two threads to execute on the SIMD

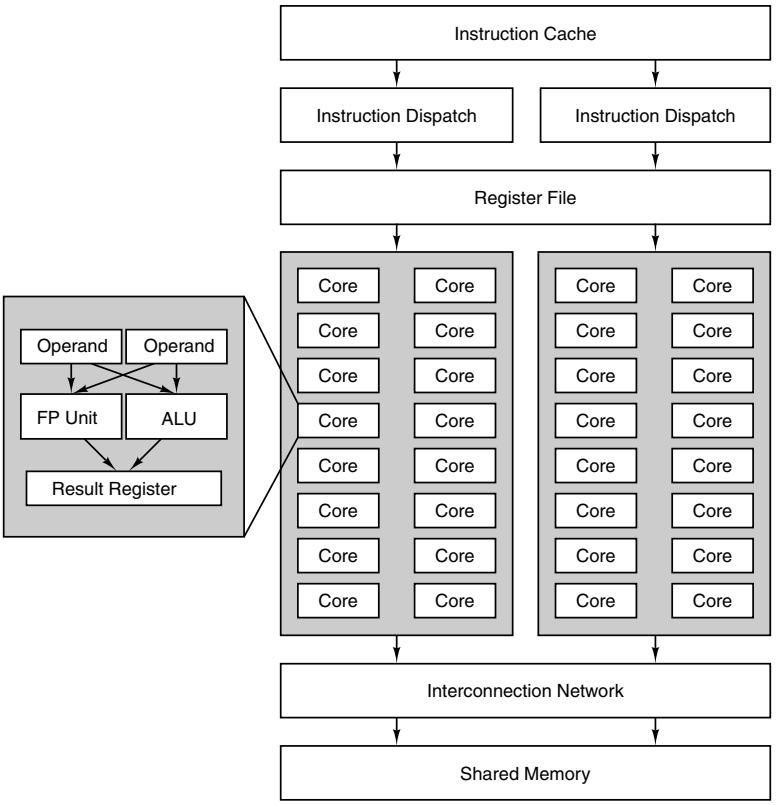

Figure 2-7. The SIMD core of the Fermi graphics processing unit.

processor. The next instruction from each thread then executes on up to 16 SIMD processors, although possibly fewer if there is not enough data parallelism. If each thread is able to perform 16 operations per cycle, a fully loaded Fermi GPU core with 32 SMs will perform a whopping 512 operations per cycle. This is an impressive feat considering that a similar-sized general purpose quad-core CPU would struggle to achieve 1/32 as much processing.

A **vector processor** appears to the programmer very much like a SIMD processor. Like a SIMD processor, it is very efficient at executing a sequence of operations on pairs of data elements. But unlike a SIMD processor, all of the operations are performed in a single, heavily pipelined functional unit. The company Seymour Cray founded, Cray Research, produced many vector processors, starting with the Cray-1 back in 1974 and continuing through current models.

Both SIMD processors and vector processors work on arrays of data. Both execute single instructions that, for example, add the elements together pairwise for two vectors. But while the SIMD processor does it by having as many adders as elements in the vector, the vector processor has the concept of a **vector register**,

which consists of a set of conventional registers that can be loaded from memory in a single instruction, which actually loads them from memory serially. Then a vector addition instruction performs the pairwise addition of the elements of two such vectors by feeding them to a pipelined adder from the two vector registers. The result from the adder is another vector, which can either be stored into a vector register or used directly as an operand for another vector operation. The SSE (Streaming SIMD Extension) instructions available on the Intel Core architecture use this execution model to speed up highly regular computation, such as multimedia and scientific software. In this respect, the Intel Core architecture has the ILLIAC IV as one of its ancestors.

Multiprocessors

The processing elements in a data parallel processor are not independent CPUs, since there is only one control unit shared among all of them. Our first parallel system with multiple full-blown CPUs is the **multiprocessor**, a system with more than one CPU sharing a common memory, like a group of people in a room sharing a common blackboard. Since each CPU can read or write any part of memory, they must coordinate (in software) to avoid getting in each other's way. When two or more CPUs have the ability to interact closely, as is the case with multiprocessors, they are said to be tightly coupled.

Various implementation schemes are possible. The simplest one is to have a single bus with multiple CPUs and one memory all plugged into it. A diagram of such a bus-based multiprocessor is shown in Fig. 2-8(a).

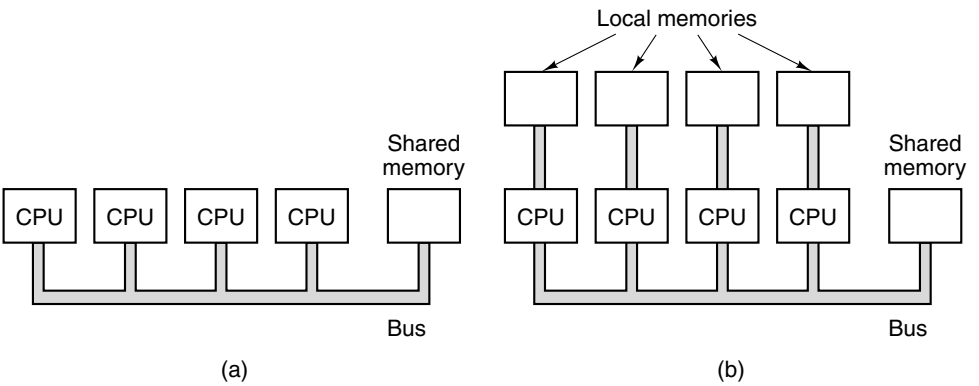

Figure 2-8. (a) A single-bus multiprocessor. (b) A multicomputer with local memories.

It does not take much imagination to realize that with a large number of fast processors constantly trying to access memory over the same bus, conflicts will result. Multiprocessor designers have come up with various schemes to reduce this

contention and improve performance. One design, shown in Fig. 2-8(b), gives each processor some local memory of its own, not accessible to the others. This memory can be used for program code and those data items that need not be shared. Access to this private memory does not use the main bus, greatly reducing bus traffic. Other schemes (e.g., caching—see below) are also possible.

Multiprocessors have the advantage over other kinds of parallel computers that the programming model of a single shared memory is easy to work with. For example, imagine a program looking for cancer cells in a photograph of some tissue taken through a microscope. The digitized photograph could be kept in the common memory, with each processor assigned some region of the photograph to hunt in. Since each processor has access to the entire memory, studying a cell that starts in its assigned region but straddles the boundary into the next region is no problem.

Multicomputers

Although multiprocessors with a modest number of processors (≤ 256) are relatively easy to build, large ones are surprisingly difficult to construct. The difficulty is in connecting so many the processors to the memory. To get around these problems, many designers have simply abandoned the idea of having a shared memory and just build systems consisting of large numbers of interconnected computers, each having its own private memory, but no common memory. These systems are called **multicomputers**. The CPUs in a multicomputer are said to be **loosely coupled**, to contrast them with the **tightly coupled** multiprocessor CPUs.

The CPUs in a multicomputer communicate by sending each other messages, something like email, but much faster. For large systems, having every computer connected to every other computer is impractical, so topologies such as 2D and 3D grids, trees, and rings are used. As a result, messages from one computer to another often must pass through one or more intermediate computers or switches to get from the source to the destination. Nevertheless, message-passing times on the order of a few microseconds can be achieved without much difficulty. Multicomputers with over 250,000 CPUs, such as IBM's Blue Gene/P, have been built.

Since multiprocessors are easier to program and multicomputers are easier to build, there is much research on designing hybrid systems that combine the good properties of each. Such computers try to present the illusion of shared memory without going to the expense of actually constructing it. We will go into multiprocessors and multicomputers in detail in Chap. 8.

2.2 PRIMARY MEMORY

The **memory** is that part of the computer where programs and data are stored. Some computer scientists (especially British ones) use the term **store** or **storage** rather than memory, although more and more, the term "storage" is used to refer