Eight Great Ideas

1. Design for Moore's Law

The one constant for computer designers is rapid change, which is driven largely by Moore's Law. It states that integrated circuit resources double every 18–24 months. Moore's Law resulted from a 1965 prediction of such growth in IC capacity made by Gordon Moore, one of the founders of Intel. As computer designs can take years, the resources available per chip can easily double or quadruple between the start and finish of the project. Like a skeet shooter, computer architects must anticipate where the technology will be when the design finishes rather than design for where it starts. We use an "up and to the right" Moore's Law graph to represent designing for rapid change.

2. Use Abstraction to Simplify Design

Both computer architects and programmers had to invent techniques to make themselves more productive, for otherwise design time would lengthen as dramatically as resources grew by Moore's Law. A major productivity technique for hardware and soft ware is to use abstractions to represent the design at different levels of representation; lower-level details are hidden to off er a simpler model at higher levels. We'll use the abstract painting icon to represent this second great idea.

3. Make the common case fast

Making the common case fast will tend to enhance performance better than optimizing the rare case. Ironically, the common case is oft en simpler than the rare case and hence is oft en easier to enhance. This common sense advice implies that you know what the common case is, which is only possible with careful experimentation and measurement. We use a sports car as the icon for making the common case fast, as the most common trip has one or two passengers, and it's surely easier to make a fast sports car than a fast minivan.

4. Performance via parallelism

Since the dawn of computing, computer architects have offered designs that get more performance by performing operations in parallel. We'll see many examples of parallelism in this book. We use multiple jet engines of a plane as our icon for parallel performance.

5. Performance via pipelining

A particular pattern of parallelism is so prevalent in computer architecture that it merits its own name: pipelining. For example, before fire engines, a "bucket brigade" would respond to a fire, which many cowboy movies show in response to a dastardly act by the villain. Th e townsfolk form a human chain to carry a water source to fi re, as they could much more quickly move buckets up the chain instead of individuals running back and forth. Our pipeline icon is a sequence of pipes, with each section representing one stage of the pipeline.

6. Performance via prediction

Following the saying that it can be better to ask for forgiveness than to ask for permission, the next great idea is prediction. In some cases it can be faster on average to guess and start working rather than wait until you know for sure, assuming that the mechanism to recover from a misprediction is not too expensive and your prediction is relatively accurate. We use the fortune-teller's crystal ball as our prediction icon.

7. Hierarchy of memories

Programmers want memory to be fast, large, and cheap, as memory speed often shapes performance, capacity limits the size of problems that can be solved, and the cost of memory today is often the majority of computer cost. Architects have found that they can address these conflicting demands with a hierarchy of memories, with the fastest, smallest, and most expensive memory per bit at the top of the hierarchy and the slowest, largest, and cheapest per bit at the bottom. Caches give the programmer the illusion that main memory is nearly as fast as the top of the hierarchy and nearly as big and cheap as the bottom of the hierarchy. We use a layered triangle icon to represent the memory hierarchy. The shape indicates speed, cost, and size: the closer to the top, the faster and more expensive per bit the memory; the wider the base of the layer, the bigger the memory.

8. Dependability via redundancy

Computers not only need to be fast; they need to be dependable. Since any physical device can fail, we make systems dependable by including redundant components that can take over when a failure occurs and to help detect failures. We use the tractor-trailer as our icon, since the dual tires on each side of its rear axels allow the truck to continue driving even when one tire fails. (Presumably, the truck driver heads immediately to a repair facility so the fl at tire can be fixed, thereby restoring redundancy!)