

1 Introduction to the Systems Approach

1.1 SYSTEM ARCHITECTURE: AN OVERVIEW

The past 40 years have seen amazing advances in silicon technology and resulting increases in transistor density and performance. In 1966, Fairchild Semiconductor [84] introduced a quad two input NAND gate with about 10 transistors on a die. In 2008, the Intel quad-core Itanium processor has 2 billion transistors [226]. Figures 1.1 and 1.2 show the unrelenting advance in improving transistor density and the corresponding decrease in device cost.

The aim of this book is to present an approach for computer system design that exploits this enormous transistor density. In part, this is a direct extension of studies in computer architecture and design. However, it is also a study of system architecture and design.

About 50 years ago, a seminal text, *Systems Engineering—An Introduction to the Design of Large-Scale Systems* [111], appeared. As the authors, H.H. Goode and R.E. Machol, pointed out, the system's view of engineering was created by a need to deal with complexity. As then, our ability to deal with complex design problems is greatly enhanced by computer-based tools.

A system-on-chip (SOC) architecture is an ensemble of processors, memories, and interconnects tailored to an application domain. A simple example of such an architecture is the Emotion Engine [147, 187, 237] for the Sony PlayStation 2 (Figure 1.3), which has two main functions: behavior simulation and geometry translation. This system contains three essential components: a main processor of the reduced instruction set computer (RISC) style [118] and two vector processing units, VPU0 and VPU1, each of which contains four parallel processors of the single instruction, multiple data (SIMD) stream style [97]. We provide a brief overview of these components and our overall approach in the next few sections.

While the focus of the book is on the system, in order to understand the system, one must first understand the components. So, before returning to the issue of system architecture later in this chapter, we review the components that make up the system.

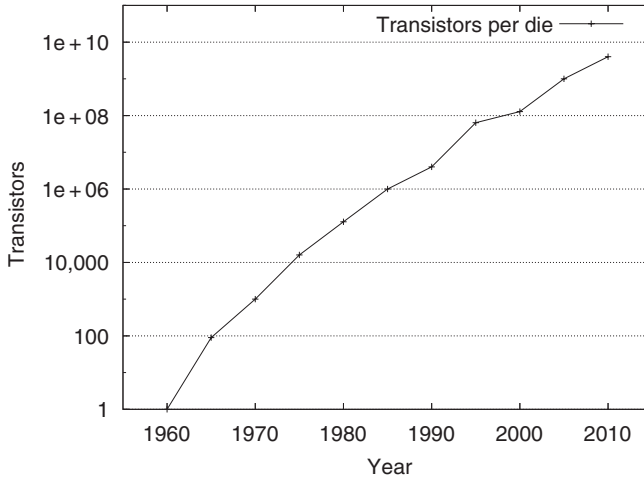


Figure 1.1 The increasing transistor density on a silicon die.

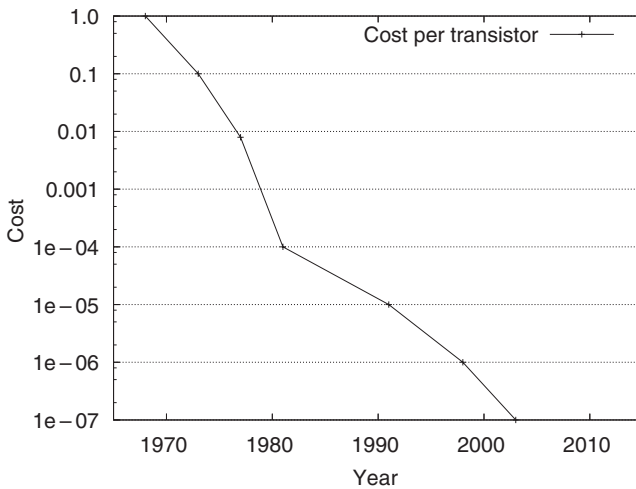


Figure 1.2 The decrease of transistor cost over the years.

1.2 COMPONENTS OF THE SYSTEM: PROCESSORS, MEMORIES, AND INTERCONNECTS

The term *architecture* denotes the operational structure and the user's view of the system. Over time, it has evolved to include both the functional specification and the hardware implementation. The system architecture defines the system-level building blocks, such as processors and memories, and the

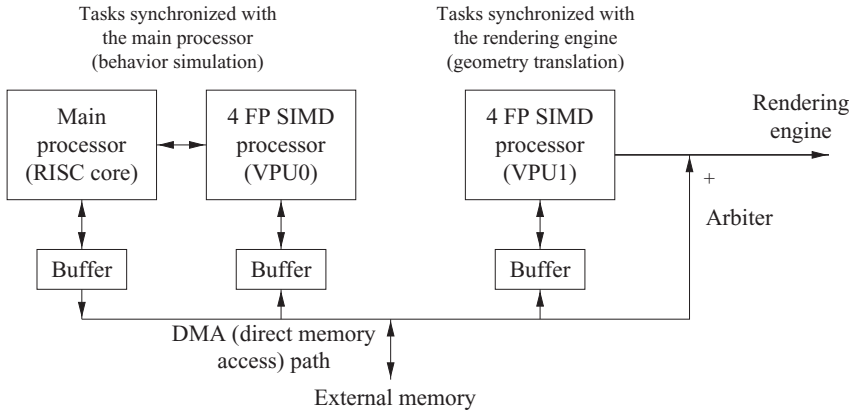


Figure 1.3 High-level functional view of a system-on-chip: the Emotion Engine of the Sony PlayStation 2 [147, 187].

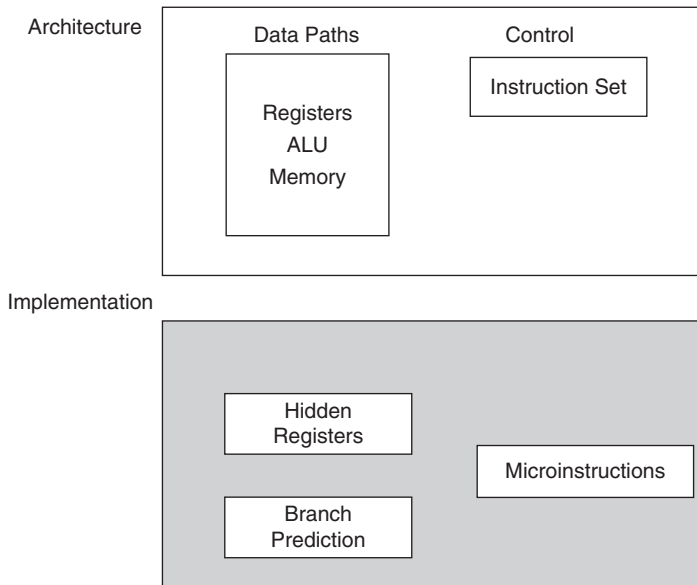


Figure 1.4 The processor architecture and its implementation.

interconnection between them. The processor architecture determines the processor's instruction set, the associated programming model, its detailed implementation, which may include hidden registers, branch prediction circuits and specific details concerning the ALU (arithmetic logic unit). The implementation of a processor is also known as *microarchitecture* (Figure 1.4).

The system designer has a programmer's or user's view of the system components, the system view of memory, the variety of specialized processors, and

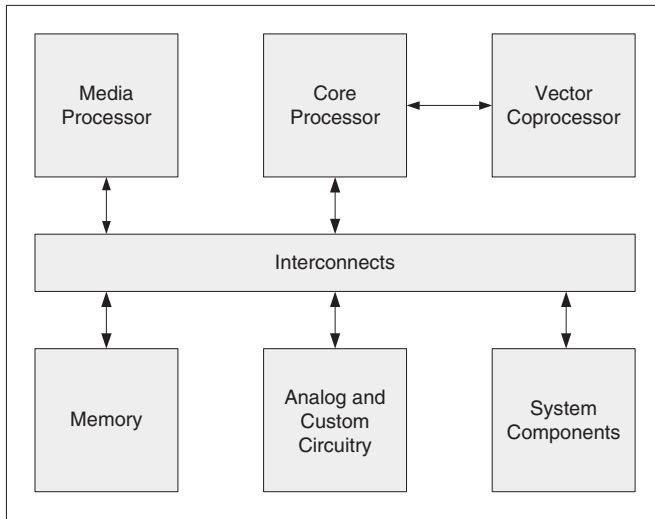


Figure 1.5 A basic SOC system model.

their interconnection. The next sections cover basic components: the processor architecture, the memory, and the bus or interconnect architecture.

Figure 1.5 illustrates some of the basic elements of an SOC system. These include a number of heterogeneous processors interconnected to one or more memory elements with possibly an array of reconfigurable logic. Frequently, the SOC also has analog circuitry for managing sensor data and analog-to-digital conversion, or to support wireless data transmission.

As an example, an SOC for a smart phone would need to support, in addition to audio input and output capabilities for a traditional phone, Internet access functions and multimedia facilities for video communication, document processing, and entertainment such as games and movies. A possible configuration for the elements in Figure 1.5 would have the core processor being implemented by several ARM Cortex-A9 processors for application processing, and the media processor being implemented by a Mali-400MP graphics processor and a Mali-VE video engine. The system components and custom circuitry would interface with peripherals such as the camera, the screen, and the wireless communication unit. The elements would be connected together by AXI (Advanced eXtensible Interface) interconnects.

If all the elements cannot be contained on a single chip, the implementation is probably best referred to as a system on a board, but often is still called a SOC. What distinguishes a system on a board (or chip) from the conventional general-purpose computer plus memory on a board is the specific nature of the design target. The application is assumed to be known and specified so that the elements of the system can be selected, sized, and evaluated during the design process. The emphasis on selecting, parameterizing, and configuring system components tailored to a target application distinguishes a system architect from a computer architect.

In this chapter, we primarily look at the higher-level definition of the processor—the programmer’s view or the instruction set architecture (ISA), the basics of the processor microarchitecture, memory hierarchies, and the interconnection structure. In later chapters, we shall study in more detail the implementation issues for these elements.

1.3 HARDWARE AND SOFTWARE: PROGRAMMABILITY VERSUS PERFORMANCE

A fundamental decision in SOC design is to choose which components in the system are to be implemented in hardware and in software. The major benefits and drawbacks of hardware and software implementations are summarized in Table 1.1.

A software implementation is usually executed on a general-purpose processor (GPP), which interprets instructions at run time. This architecture offers flexibility and adaptability, and provides a way of sharing resources among different applications; however, the hardware implementation of the ISA is generally slower and more power hungry than implementing the corresponding function directly in hardware without the overhead of fetching and decoding instructions.

Most software developers use high-level languages and tools that enhance productivity, such as program development environments, optimizing compilers, and performance profilers. In contrast, the direct implementation of applications in hardware results in custom application-specific integrated circuits (ASICs), which often provides high performance at the expense of programmability—and hence flexibility, productivity, and cost.

Given that hardware and software have complementary features, many SOC designs aim to combine the individual benefits of the two. The obvious method is to implement the performance-critical parts of the application in hardware, and the rest in software. For instance, if 90% of the software execution time of an application is spent on 10% of the source code, up to a 10-fold speedup is achievable if that 10% of the code is efficiently implemented in hardware. We shall make use of this observation to customize designs in Chapter 6.

Custom ASIC hardware and software on GPPs can be seen as two extremes in the technology spectrum with different trade-offs in programmability and

TABLE 1.1 Benefits and Drawbacks of Software and Hardware Implementations

	Benefits	Drawbacks
Hardware	Fast, low power consumption	Inflexible, unadaptable, complex to build and test
Software	Flexible, adaptable, simple to build and test	Slow, high power consumption

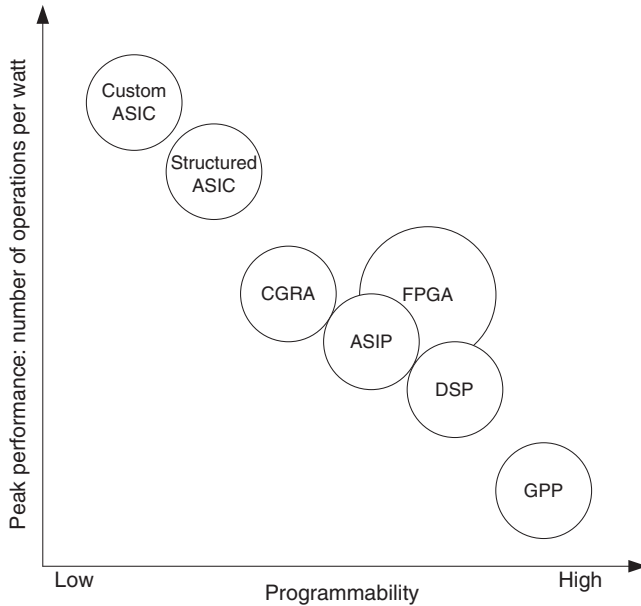


Figure 1.6 A simplified technology comparison: programmability versus performance. GPP, general-purpose processor; CGRA, coarse-grained reconfigurable architecture.

performance; there are various technologies that lie between these two extremes (Figure 1.6). The two more well-known ones are application-specific instruction processors (ASIPs) and field-programmable gate arrays (FPGAs).

An ASIP is a processor with an instruction set customized for a specific application or domain. Custom instructions efficiently implemented in hardware are often integrated into a base processor with a basic instruction set. This capability often improves upon the conventional approach of using standard instruction sets to fulfill the same task while preserving its flexibility. Chapters 6 and 7 explore further some of the issues involving custom instructions.

An FPGA typically contains an array of computation units, memories, and their interconnections, and all three are usually programmable in the field by application builders. FPGA technology often offers a good compromise: It is faster than software while being more flexible and having shorter development times than custom ASIC hardware implementations; like GPPs, they are offered as off-the-shelf devices that can be programmed without going through chip fabrication. Because of the growing demand for reducing the time to market and the increasing cost of chip fabrication, FPGAs are becoming more popular for implementing digital designs.

Most commercial FPGAs contain an array of fine-grained logic blocks, each only a few bits wide. It is also possible to have the following:

- *Coarse-Grained Reconfigurable Architecture (CGRA)*. It contains logic blocks that process byte-wide or multiple byte-wide data, which can form building blocks of datapaths.
- *Structured ASIC*. It allows application builders to customize the resources before fabrication. While it offers performance close to that of ASIC, the need for chip fabrication can be an issue.
- *Digital Signal Processors (DSPs)*. The organization and instruction set for these devices are optimized for digital signal processing applications. Like microprocessors, they have a fixed hardware architecture that cannot be reconfigured.

Figure 1.6 compares these technologies in terms of programmability and performance. Chapters 6–8 provide further information about some of these technologies.

1.4 PROCESSOR ARCHITECTURES

Typically, processors are characterized either by their application or by their architecture (or structure), as shown in Tables 1.2 and 1.3. The requirements space of an application is often large, and there is a range of implementation options. Thus, it is usually difficult to associate a particular architecture with a particular application. In addition, some architectures combine different implementation approaches as seen in the PlayStation example of Section 1.1. There, the graphics processor consists of a four-element SIMD array of vector processing functional units (FUs). Other SOC implementations consist of multiprocessors using very long instruction word (VLIW) and/or superscalar processors.

TABLE 1.2 Processor Examples as Identified by Function

Processor Type	Application
Graphics processing unit (GPU)	3-D graphics; rendering, shading, texture
Digital signal processor (DSP)	Generic, sometimes used with wireless
Media processor	Video and audio signal processing
Network processor	Routing, buffering

TABLE 1.3 Processor Examples as Identified by Architecture

Processor Type	Architecture/Implementation Approach
SIMD	Single instruction applied to multiple functional units (processors)
Vector (VP)	Single instruction applied to multiple pipelined registers
VLIW	Multiple instructions issued each cycle under compiler control
Superscalar	Multiple instructions issued each cycle under hardware control

From the programmer's point of view, sequential processors execute one instruction at a time. However, many processors have the capability to execute several instructions concurrently in a manner that is transparent to the programmer, through techniques such as pipelining, multiple execution units, and multiple cores. Pipelining is a powerful technique that is used in almost all current processor implementations. Techniques to extract and exploit the inherent parallelism in the code at compile time or run time are also widely used.

Exploiting program parallelism is one of the most important goals in computer architecture.

Instruction-level parallelism (ILP) means that multiple operations can be executed in parallel within a program. ILP may be achieved with hardware, compiler, or operating system techniques. At the loop level, consecutive loop iterations are ideal candidates for parallel execution, provided that there is no data dependency between subsequent loop iterations. Next, there is parallelism available at the procedure level, which depends largely on the algorithms used in the program. Finally, multiple independent programs can execute in parallel.

Different computer architectures have been built to exploit this inherent parallelism. In general, a computer architecture consists of one or more interconnected processor elements (PEs) that operate concurrently, solving a single overall problem.

1.4.1 Processor: A Functional View

Table 1.4 shows different SOC designs and the processor used in each design. For these examples, we can characterize them as general purpose, or special purpose with support for gaming or signal processing applications. This functional view tells little about the underlying hardware implementation. Indeed, several quite different architectural approaches could implement the same generic function. The graphics function, for example, requires shading, rendering, and texturing functions as well as perhaps a video function. Depending

TABLE 1.4 Processor Models for Different SOC Examples

SOC	Application	Base ISA	Processor Description
Freescale e600 [101]	DSP	PowerPC	Superscalar with vector extension
ClearSpeed CSX600 [59]	General	Proprietary ISA	Array processor of 96 processing elements
PlayStation 2 [147, 187, 237]	Gaming	MIPS	Pipelined with two vector coprocessors
ARM VFP11 [23]	General	ARM	Configurable vector coprocessor



Figure 1.7 Instruction execution sequence.

on the relative importance of these functions and the resolution of the created images, we could have radically different architectural implementations.

1.4.2 Processor: An Architectural View

The architectural view of the system describes the actual implementation at least in a broad-brush way. For sophisticated architectural approaches, more detail is required to understand the complete implementation.

Simple Sequential Processor Sequential processors directly implement the sequential execution model. These processors process instructions sequentially from the instruction stream. The next instruction is not processed until all execution for the current instruction is complete and its results have been committed.

The semantics of the instruction determines that a sequence of actions must be performed to produce the specified result (Figure 1.7). These actions can be overlapped, but the result must appear in the specified serial order. These actions include

1. fetching the instruction into the instruction register (IF),
2. decoding the opcode of the instruction (ID),
3. generating the address in memory of any data item residing there (AG),
4. fetching data operands into executable registers (DF),
5. executing the specified operation (EX), and
6. writing back the result to the register file (WB).

A simple sequential processor model is shown in Figure 1.8. During execution, a sequential processor executes one or more operations per clock cycle from the instruction stream. An instruction is a container that represents the smallest execution packet managed explicitly by the processor. One or more operations are contained within an instruction. The distinction between instructions and operations is crucial to distinguish between processor behaviors. Scalar and superscalar processors consume one or more instructions per cycle, where each instruction contains a single operation.

Although conceptually simple, executing each instruction sequentially has significant performance drawbacks: A considerable amount of time is spent on overhead and not on actual execution. Thus, the simplicity of directly implementing the sequential execution model has significant performance costs.

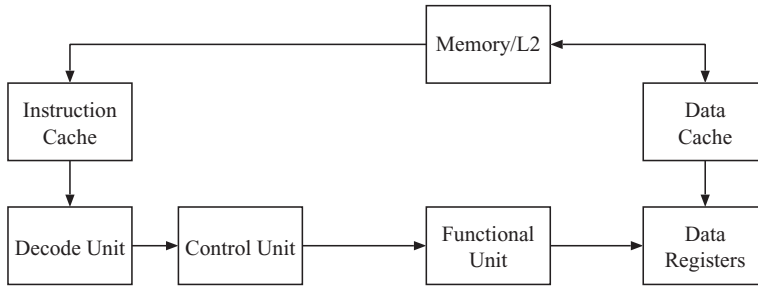


Figure 1.8 Sequential processor model.

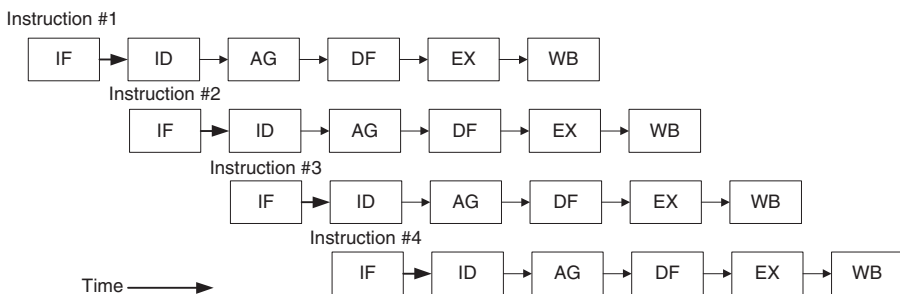


Figure 1.9 Instruction timing in a pipelined processor.

Pipelined Processor Pipelining is a straightforward approach to exploiting parallelism that is based on concurrently performing different phases (instruction fetch, decode, execution, etc.) of processing an instruction. Pipelining assumes that these phases are independent between different operations and can be overlapped—when this condition does not hold, the processor stalls the downstream phases to enforce the dependency. Thus, multiple operations can be processed simultaneously with each operation at a different phase of its processing. Figure 1.9 illustrates the instruction timing in a pipelined processor, assuming that the instructions are independent.

For a simple pipelined machine, there is only one operation in each phase at any given time; thus, one operation is being fetched (IF); one operation is being decoded (ID); one operation is generating an address (AG); one operation is accessing operands (DF); one operation is in execution (EX); and one operation is storing results (WB). Figure 1.10 illustrates the general form of a pipelined processor. The most rigid form of a pipeline, sometimes called the static pipeline, requires the processor to go through all stages or phases of the pipeline whether required by a particular instruction or not. A dynamic pipeline allows the bypassing of one or more pipeline stages, depending on the requirements of the instruction. The more complex dynamic pipelines allow instructions to complete out of (sequential) order, or even to initiate out of order. The out-of-order processors must ensure that the sequential consistency of the program is preserved. Table 1.5 shows some SOC pipelined “soft” processors.

TABLE 1.5 SOC Examples Using Pipelined Soft Processors [177, 178]. A Soft Processor Is Implemented with FPGAs or Similar Reconfigurable Technology

Processor	Word Length (bit)	Pipeline Stages	I/D-Cache* Total (KB)	Floating-Point Unit (FPU)	Usual Target
Xilinx MicroBlaze	32	3	0–64	Optional	FPGA
Altera Nios II fast	32	6	0–64	—	FPGA
ARC 600 [19]	16/32	5	0–32	Optional	ASIC
Tensilica Xtensa LX	16/24	5–7	0–32	Optional	ASIC
Cambridge XAP3a	16/32	2	—	—	ASIC

*Means configurable I-cache and/or D-cache.

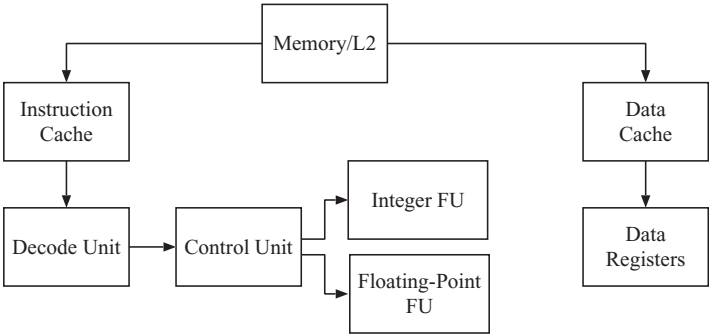


Figure 1.10 Pipelined processor model.

ILP While pipelining does not necessarily lead to executing multiple instructions at exactly the same time, there are other techniques that do. These techniques may use some combination of static scheduling and dynamic analysis to perform concurrently the actual evaluation phase of several different operations, potentially yielding an execution rate of greater than one operation every cycle. Since historically most instructions consist of only a single operation, this kind of parallelism has been named ILP (instruction level parallelism).

Two architectures that exploit ILP are *superscalar* and *VLIW* processors. They use different techniques to achieve execution rates greater than one operation per cycle. A superscalar processor dynamically examines the instruction stream to determine which operations are independent and can be executed. A VLIW processor relies on the compiler to analyze the available operations (OP) and to schedule independent operations into wide instruction words, which then execute these operations in parallel with no further analysis.

Figure 1.11 shows the instruction timing of a pipelined superscalar or VLIW processor executing two instructions per cycle. In this case, all the instructions are independent so that they can be executed in parallel. The next two sections describe these two architectures in more detail.

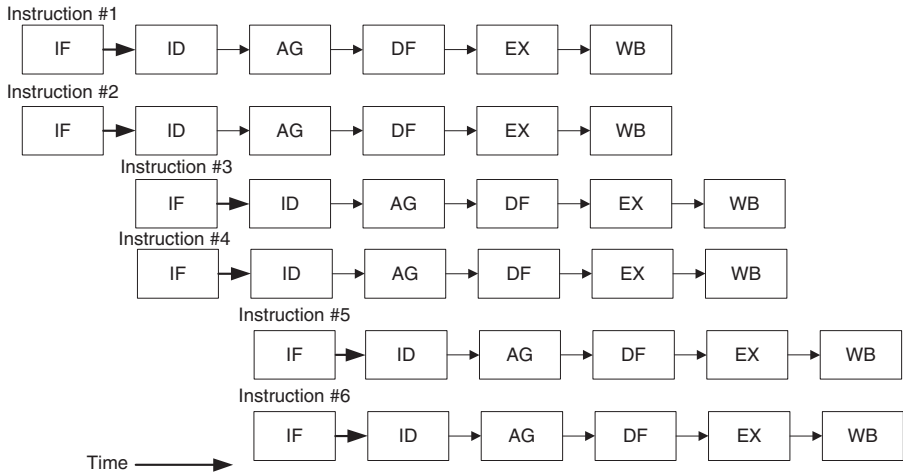


Figure 1.11 Instruction timing in a pipelined ILP processor.

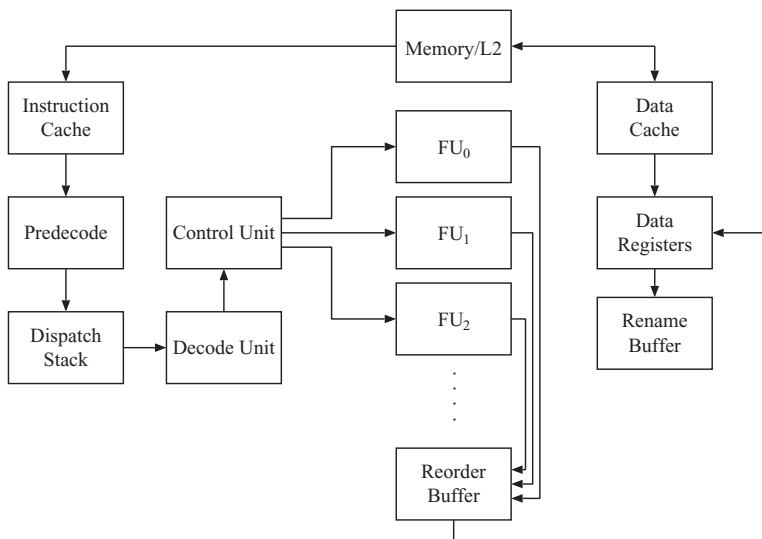


Figure 1.12 Superscalar processor model.

Superscalar Processors Dynamic pipelined processors remain limited to executing a single operation per cycle by virtue of their scalar nature. This limitation can be avoided with the addition of multiple functional units and a dynamic scheduler to process more than one instruction per cycle (Figure 1.12). These superscalar processors [135] can achieve execution rates of several instructions per cycle (usually limited to two, but more is possible depending on the application). The most significant advantage of a superscalar processor is that processing multiple instructions per cycle is done transparently to the

user, and that it can provide binary code compatibility while achieving better performance.

Compared to a dynamic pipelined processor, a superscalar processor adds a scheduling instruction window that analyzes multiple instructions from the instruction stream in each cycle. Although processed in parallel, these instructions are treated in the same manner as in a pipelined processor. Before an instruction is issued for execution, dependencies between the instruction and its prior instructions must be checked by hardware.

Because of the complexity of the dynamic scheduling logic, high-performance superscalar processors are limited to processing four to six instructions per cycle. Although superscalar processors can exploit ILP from the dynamic instruction stream, exploiting higher degrees of parallelism requires other approaches.

VLIW Processors In contrast to dynamic analyses in hardware to determine which operations can be executed in parallel, VLIW processors (Figure 1.13) rely on static analyses in the compiler.

VLIW processors are thus less complex than superscalar processors and have the potential for higher performance. A VLIW processor executes operations from statically scheduled instructions that contain multiple independent operations. Because the control complexity of a VLIW processor is not significantly greater than that of a scalar processor, the improved performance comes without the complexity penalties.

VLIW processors rely on the static analyses performed by the compiler and are unable to take advantage of any dynamic execution characteristics. For applications that can be scheduled statically to use the processor resources effectively, a simple VLIW implementation results in high performance. Unfortunately, not all applications can be effectively scheduled statically. In many applications, execution does not proceed exactly along the path defined

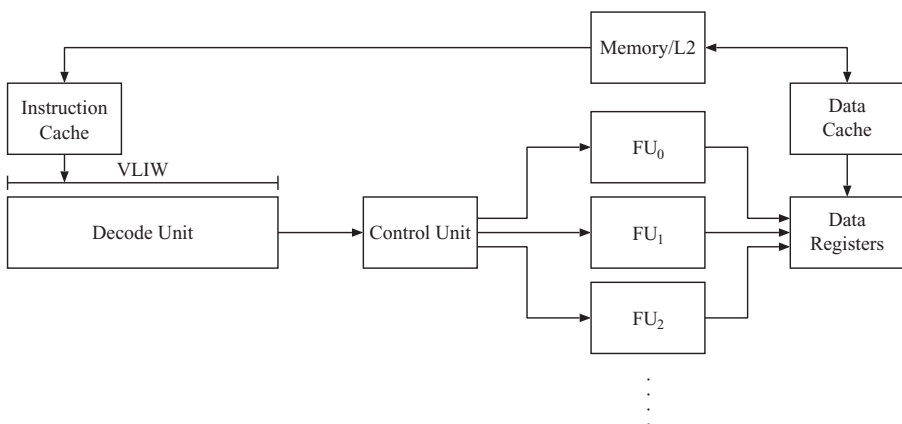


Figure 1.13 VLIW processor model.

by the code scheduler in the compiler. Two classes of execution variations can arise and affect the scheduled execution behavior:

1. delayed results from operations whose latency differs from the assumed latency scheduled by the compiler and
2. interruptions from exceptions or interrupts, which change the execution path to a completely different and unanticipated code schedule.

Although stalling the processor can control a delayed result, this solution can result in significant performance penalties. The most common execution delay is a data cache miss. Many VLIW processors avoid all situations that can result in a delay by avoiding data caches and by assuming worst-case latencies for operations. However, when there is insufficient parallelism to hide the exposed worst-case operation latency, the instruction schedule has many incompletely filled or empty instructions, resulting in poor performance.

Tables 1.6 and 1.7 describe some representative *superscalar* and *VLIW processors*.

SIMD Architectures: Array and Vector Processors The SIMD class of processor architecture includes both array and vector processors. The SIMD processor is a natural response to the use of certain regular data structures, such as vectors and matrices. From the view of an assembly-level programmer, programming SIMD architecture appears to be very similar to programming a simple processor except that some operations perform computations on aggregate data. Since these regular structures are widely used in scientific programming, the SIMD processor has been very successful in these environments.

The two popular types of SIMD processor are the array processor and the vector processor. They differ both in their implementations and in their data

TABLE 1.6 SOC Examples Using Superscalar Processors

Device	Number of Functional Units	Issue Width	Base Instruction Set
MIPS 74K Core [183]	4	2	MIPS32
Infineon TriCore2 [129]	4	3	RISC
Freescale e600 [101]	6	3	PowerPC

TABLE 1.7 SOC Examples Using VLIW Processors

Device	Number of Functional Units	Issue Width
Fujitsu MB93555A [103]	8	8
TI TMS320C6713B [243]	8	8
CEVA-X1620 [54]	30	8
Philips Nexperia PNX1700 [199]	30	5

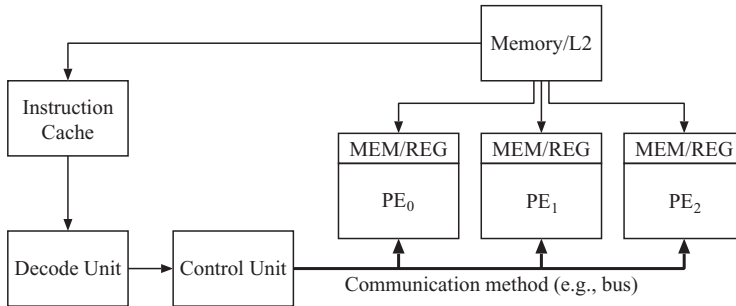


Figure 1.14 Array processor model.

organizations. An array processor consists of many interconnected processor elements, each having their own local memory space. A vector processor consists of a single processor that references a global memory space and has special function units that operate on vectors.

An array processor or a vector processor can be obtained by extending the instruction set to an otherwise conventional machine. The extended instructions enable control over special resources in the processor, or in some sort of coprocessor. The purpose of such extensions is to enable increased performance on special applications.

Array Processors The array processor (Figure 1.14) is a set of parallel processor elements connected via one or more networks, possibly including local and global interelement communications and control communications. Processor elements operate in lockstep in response to a single broadcast instruction from a control processor (SIMD). Each processor element (PE) has its own private memory, and data are distributed across the elements in a regular fashion that is dependent on both the actual structure of the data and also the computations to be performed on the data. Direct access to global memory or another processor element's local memory is expensive, so intermediate values are propagated through the array through local interprocessor connections. This requires that the data be distributed carefully so that the routing required to propagate these values is simple and regular. It is sometimes easier to duplicate data values and computations than it is to support a complex or irregular routing of data between processor elements.

Since instructions are broadcast, there is no means local to a processor element of altering the flow of the instruction stream; however, individual processor elements can conditionally disable instructions based on local status information—these processor elements are idle when this condition occurs. The actual instruction stream consists of more than a fixed stream of operations. An array processor is typically coupled to a general-purpose control processor that provides both scalar operations as well as array operations that are broadcast to all processor elements in the array. The control processor performs the scalar sections of the application, interfaces with the outside

TABLE 1.8 SOC Examples Based on Array Processors

Device	Processors per Control Unit	Data Size (bit)
ClearSpeed CSX600 [59]	96	32
Atsana J2211 [174]	Configurable	16/32
Xelerator X10q [257]	200	4

world, and controls the flow of execution; the array processor performs the array sections of the application as directed by the control processor.

A suitable application for use on an array processor has several key characteristics: a significant amount of data that have a regular structure, computations on the data that are uniformly applied to many or all elements of the data set, and simple and regular patterns relating the computations and the data. An example of an application that has these characteristics is the solution of the Navier–Stokes equations, although any application that has significant matrix computations is likely to benefit from the concurrent capabilities of an array processor.

Table 1.8 contains several array processor examples. The ClearSpeed processor is an example of an array processor chip that is directed at signal processing applications.

Vector Processors A vector processor is a single processor that resembles a traditional single stream processor, except that some of the function units (and registers) operate on vectors—sequences of data values that are seemingly operated on as a single entity. These function units are deeply pipelined and have high clock rates. While the vector pipelines often have higher latencies compared with scalar function units, the rapid delivery of the input vector data elements, together with the high clock rates, results in a significant throughput.

Modern vector processors require that vectors be explicitly loaded into special vector registers and stored back into memory—the same course that modern scalar processors use for similar reasons. Vector processors have several features that enable them to achieve high performance. One feature is the ability to concurrently load and store values between the vector register file and the main memory while performing computations on values in the vector register file. This is an important feature since the limited length of vector registers requires that vectors longer than the register length would be processed in segments—a technique called strip mining. Not being able to overlap memory accesses and computations would pose a significant performance bottleneck.

Most vector processors support a form of result bypassing—in this case called chaining—that allows a follow-on computation to commence as soon as the first value is available from the preceding computation. Thus, instead of waiting for the entire vector to be processed, the follow-on computation can be significantly overlapped with the preceding computation that it is dependent on. Sequential computations can be efficiently compounded to behave as

if they were a single operation, with a total latency equal to the latency of the first operation with the pipeline and chaining latencies of the remaining operations, but none of the start-up overhead that would be incurred without chaining. For example, division could be synthesized by chaining a reciprocal with a multiply operation. Chaining typically works for the results of load operations as well as normal computations.

A typical vector processor configuration (Figure 1.15) consists of a vector register file, one vector addition unit, one vector multiplication unit, and one vector reciprocal unit (used in conjunction with the vector multiplication unit to perform division); the vector register file contains multiple vector registers (elements).

Table 1.9 shows examples of vector processors. The IBM mainframes have vector instructions (and support hardware) as an option for scientific users.

Multiprocessors Multiple processors can cooperatively execute to solve a single problem by using some form of interconnection for sharing results. In

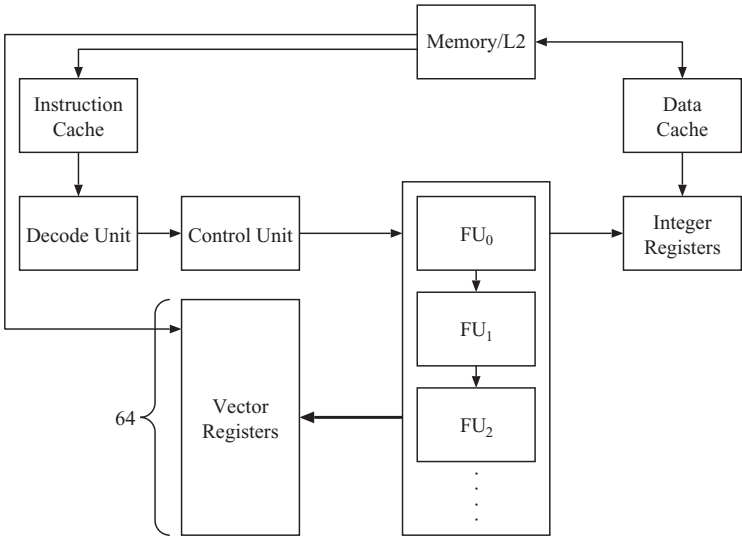


Figure 1.15 Vector processor model.

TABLE 1.9 SOC Examples Using Vector Processor

Device	Vector Function Units	Vector Registers
Freescape e600 [101]	4	32 Configurable
Motorola RSVP [58]	4 (64 bit partitionable at 16 bits)	2 streams (each 2 from, 1 to) memory
ARM VFP11 [23]	3 (64 bit partitionable to 32 bits)	4 × 8, 32 bit

Configurable implies a pool of N registers that can be configured as p register sets of N/p elements.

TABLE 1.10 SOC Multiprocessors and Multithreaded Processors

SOC	Machanic [162]	IBM Cell [141]	Philips PNX8500 [79]	Lehtoranta [155]
Number of CPUs	4	1	2	4
Threads	1	Many	1	1
Vector units	0	8	0	0
Application	Various	Various	HDTV	MPEG decode
Comment	Proposal only		Also called Viper 2	Soft processors

this configuration, each processor executes completely independently, although most applications require some form of synchronization during execution to pass information and data between processors. Since the multiple processors share memory and execute separate program tasks (MIMD [multiple instruction stream, multiple data stream]), their proper implementation is significantly more complex than the array processor. Most configurations are homogeneous with all processor elements being identical, although this is not a requirement. Table 1.10 shows examples of SOC multiprocessors.

The interconnection network in the multiprocessor passes data between processor elements and synchronizes the independent execution streams between processor elements. When the memory of the processor is distributed across all processors and only the local processor element has access to it, all data sharing is performed explicitly using messages, and all synchronization is handled within the message system. When the memory of the processor is shared across all processor elements, synchronization is more of a problem—certainly, messages can be used through the memory system to pass data and information between processor elements, but this is not necessarily the most effective use of the system.

When communications between processor elements are performed through a shared memory address space—either global or distributed between processor elements (called distributed shared memory to distinguish it from distributed memory)—there are two significant problems that arise. The first is maintaining memory consistency: the programmer-visible ordering effects on memory references, both within a processor element and between different processor elements. This problem is usually solved through a combination of hardware and software techniques. The second is cache coherency—the programmer-invisible mechanism to ensure that all processor elements see the same value for a given memory location. This problem is usually solved exclusively through hardware techniques.

The primary characteristic of a multiprocessor system is the nature of the memory address space. If each processor element has its own address space (distributed memory), the only means of communication between processor elements is through message passing. If the address space is shared (shared memory), communication is through the memory system.

The implementation of a distributed memory machine is far easier than the implementation of a shared memory machine when memory consistency and cache coherency are taken into account. However, programming a distributed memory processor can be much more difficult since the applications must be written to exploit and not to be limited by the use of message passing as the only form of communication between processor elements. On the other hand, despite the problems associated with maintaining consistency and coherency, programming a shared memory processor can take advantage of whatever communications paradigm is appropriate for a given communications requirement, and can be much easier to program.

1.5 MEMORY AND ADDRESSING

SOC applications vary significantly in memory requirements. In one case, the memory structure can be as simple as the program residing entirely in an on-chip read-only memory (ROM), with the data in on-chip RAM. In another case, the memory system might support an elaborate operating system requiring a large off-chip memory (system on a board), with a memory management unit and cache hierarchy.

Why not simply include memory with the processor on the die? This has many attractions:

1. It improves the accessibility of memory, improving both memory access time and bandwidth.
2. It reduces the need for large cache.
3. It improves performance for memory-intensive applications.

But there are problems. The first problem is that DRAM memory process technology differs from standard microprocessor process technology, and would cause some sacrifice in achievable bit density. The second problem is more serious: If memory were restricted to the processor die, its size would be correspondingly limited. Applications that require very large real memory space would be crippled. Thus, the conventional processor die model has evolved (Figure 1.16) to implement multiple robust homogeneous processors sharing the higher levels of a two- or three-level cache structure with the main memory off-die, on its own multidie module.

From a design complexity point of view, this has the advantage of being a “universal” solution: One implementation fits all applications, although not necessarily equally well. So, while a great deal of design effort is required for such an implementation, the production quantities can be large enough to justify the costs.

An alternative to this approach is clear. For specific applications, whose memory size can be bounded, we can implement an integrated memory SOC. This concept is illustrated in Figure 1.17 (also recall Figure 1.3).

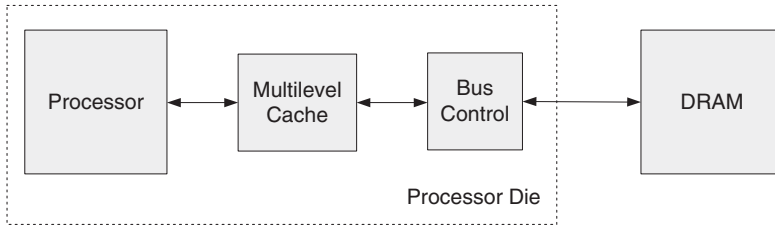


Figure 1.16 Processors with memory off-die.

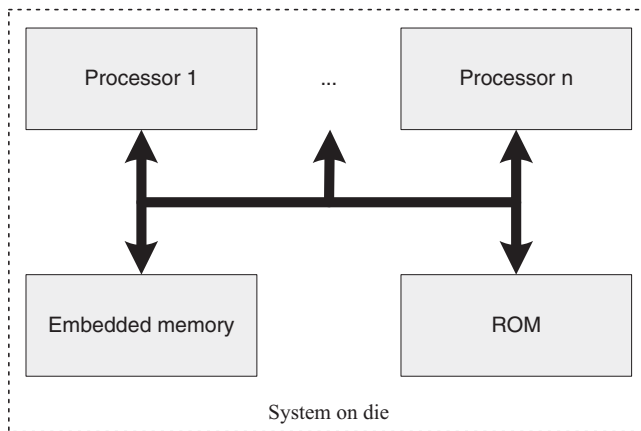


Figure 1.17 System on a chip: processors and memory.

A related but separate question is: Does the application require virtual memory (mapping disk space onto memory) or is all real memory suitable? We look at the requirement for virtual memory addressing in the next section.

Finally, the memory can be centralized or distributed. Even here, the memory can appear to the programmer as a single (centralized) shared memory, even though it is implemented in several distributed modules. Several memory considerations are listed in Table 1.11.

The *memory system* comprises the physical storage elements in the memory hierarchy. These elements include those specified by the instruction set (registers, main memory, and disk sectors) as well as those elements that are largely transparent to the user's program (buffer registers, cache, and page mapped virtual memory).

1.5.1 SOC Memory Examples

Table 1.12 shows a number of different SOC designs and their cache and memory configuration. It is important for SOC designers to consider whether to put RAM and ROM on-die or off-die. Table 1.13 shows various examples of SOC embedded memory macro cell.

TABLE 1.11 SOC Memory Considerations

Issue	Implementation	Comment
Memory placement	On-die Off-die	Limited and fixed size System on a board, slow access, limited bandwidth
Addressing	Real addressing Virtual addressing	Limited size, simple OS Much more complex, require TLB, in-order instruction execution support
Arrangement (as programmed for multiple processors)	Shared memory Message passing	Requires hardware support Additional programming
Arrangement (as implemented)	Centralized Distributed	Limited by chip considerations Can be clustered with a processor or other memory modules

TABLE 1.12 Memory Hierarchy for Different SOC Examples

SOC	Application	Cache Size	On-Die/ Off-Die	Real/ Virtual
NetSilicon NET + 40 [184]	Networking	4-KB I-cache, 4-KB D-cache	Off	Real
NetSilicon NS9775 [185]	Printing	8-KB I-cache, 4-KB D-cache	Off	Virtual
NXP LH7A404 [186]	Networking	16-KB I-cache, 8KB D-Cache	On	Virtual
Motorola RSVP [58]	Multimedia	Tile buffer memory	Off	Real

1.5.2 Addressing: The Architecture of Memory

The user's view of memory primarily consists of the addressing facilities available to the programmer. Some of these facilities are available to the application programmer and some to the operating system programmer. Virtual memory enables programs requiring larger storage than the physical memory to run and allows separation of address spaces to protect unauthorized access to memory regions when executing multiple application programs. When virtual addressing facilities are properly implemented and programmed, memory can be efficiently and securely accessed.

Virtual memory is often supported by a memory management unit. Conceptually, the physical memory address is determined by a sequence of (at least) three steps:

TABLE 1.13 Example SOC Embedded Memory Macro Cell (See Chapter 4 for the Discussion on Cell Types)

Vendor	Cell Type (Typical)	SOC User (Typical)
Virage Logic	6T (SRAM)	SigmaTel/ARM
ATMOS	1T (eDRAM)	Philips
IBM	1T (eDRAM)	IBM

Note: T refers to the number of transistors in a 1-bit cell.

1. The application produces a *process address*. This, together with the *process or user ID*, defines the *virtual address*: $\text{virtual address} = \text{offset} + (\text{program}) \text{ base} + \text{index}$, where the *offset* is specified in the instruction while the *base* and *index* values are in specified registers.
2. Since multiple processes must cooperate in the same memory space, the process addresses must be coordinated and relocated. This is typically done by a segment table. Upper bits of the *virtual address* are used to address a segment table, which has a (predetermined) *base* and *bound* values for the process, resulting in a *system address*: $\text{system address} = \text{virtual address} + (\text{process}) \text{ base}$, where the *system address* must be less than the *bound*.
3. *Virtual versus real*. For many SOC applications (and all generic systems), the memory space exceeds the available (real) implemented memory. Here the memory space is implemented on disk and only the recently used regions (pages) are brought into memory. The available pages are located by a page table. The upper bits of the system address access a page table. If the data for this page have been loaded from the disk, the location in memory will be provided as the upper address bits of the “real” or physical memory address. The lower bits of the real address are the same as the corresponding lower bits of the virtual address.

Usually, the tables (segment and page) performing address translation are in memory, and a mechanism for the translation called the translation lookaside buffer (TLB) must be used to speed up this translation. A TLB is a simple register system, usually consisting of between 64 and 256 entries, that saves recent address translations for reuse. A small number of (hashed) virtual address bits address the TLB. The TLB entry has both the real address and the complete virtual address (and ID). If the virtual address matches, the real address from the TLB can be used. Otherwise, a *not-in-TLB* event occurs and a complete translation must occur (Figure 1.18).

1.5.3 Memory for SOC Operating System

One of the most critical decisions (or requirements) concerning an SOC design is the selection of the operating system and its memory management function-

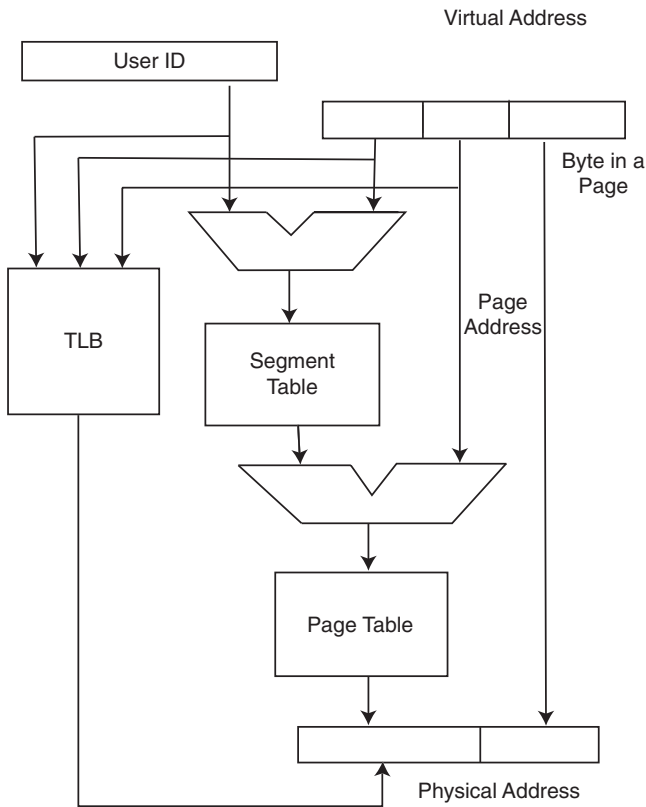


Figure 1.18 Virtual-to-real address mapping with a TLB bypass.

TABLE 1.14 Operating Systems for SOC Designs

OS	Vendor	Memory Model
uClinux	Open source	Real
VxWorks (RTOS) [254]	Wind River	Real
Windows CE	Microsoft	Virtual
Nucleus (RTOS) [175]	Mentor Graphics	Real
MQX (RTOS) [83]	ARC	Real

ality. Of primary interest to the designer is the requirement for virtual memory. If the system can be restricted to a real memory (physically, not virtually addressed) and the size of the memory can be contained to the order of 10s of megabytes, the system can be implemented as a true system on a chip (all memory on-die). The alternative, virtual memory, is often slower and significantly more expensive, requiring a complex memory management unit. Table 1.14 illustrates some current SOC designs and their operating systems.

Of course, fast real memory designs come at the price of functionality. The user has limited ways of creating new processes and of expanding the application base of the systems.

1.6 SYSTEM-LEVEL INTERCONNECTION

SOC technology typically relies on the interconnection of predesigned circuit modules (known as intellectual property [IP] blocks) to form a complete system, which can be integrated onto a single chip. In this way, the design task is raised from a circuit level to a system level. Central to the system-level performance and the reliability of the finished product is the method of interconnection used. A well-designed interconnection scheme should have vigorous and efficient communication protocols, unambiguously defined as a published standard. This facilitates interoperability between IP blocks designed by different people from different organizations and encourages design reuse. It should provide efficient communication between different modules maximizing the degree of parallelism achieved.

SOC interconnect methods can be classified into two main approaches: buses and network-on-chip, as illustrated in Figures 1.19 and 1.20.

1.6.1 Bus-Based Approach

With the bus-based approach, IP blocks are designed to conform to published bus standards (such as ARM's Advanced Microcontroller Bus Architecture

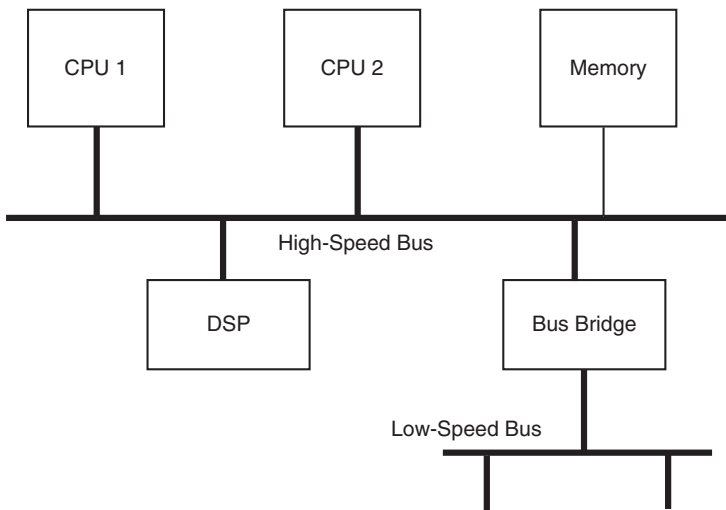


Figure 1.19 SOC system-level interconnection: bus-based approach.

[AMBA] [21] or IBM's CoreConnect [124]). Communication between modules is achieved through the sharing of the physical connections of address, data, and control bus signals. This is a common method used for SOC system-level interconnect. Usually, two or more buses are employed in a system, organized in a hierarchical fashion. To optimize system-level performance and cost, the bus closest to the CPU has the highest bandwidth, and the bus farthest from the CPU has the lowest bandwidth.

1.6.2 Network-on-Chip Approach

A network-on-chip system consists of an array of switches, either dynamically switched as in a crossbar or statically switched as in a mesh.

The crossbar approach uses asynchronous channels to connect synchronous modules that can operate at different clock frequencies. This approach has the advantage of higher throughput than a bus-based system while making integration of a system with multiple clock domains easier.

In a simple statically switched network (Figure 1.20), each node contains processing logic forming the core, and its own routing logic. The interconnect scheme is based on a two-dimensional mesh topology. All communications between switches are conducted through data packets, routed through the router interface circuit within each node. Since the interconnections between switches have a fixed distance, interconnect-related problems such as wire delay and cross talk noise are much reduced. Table 1.15 lists some interconnect examples used in SOC designs.

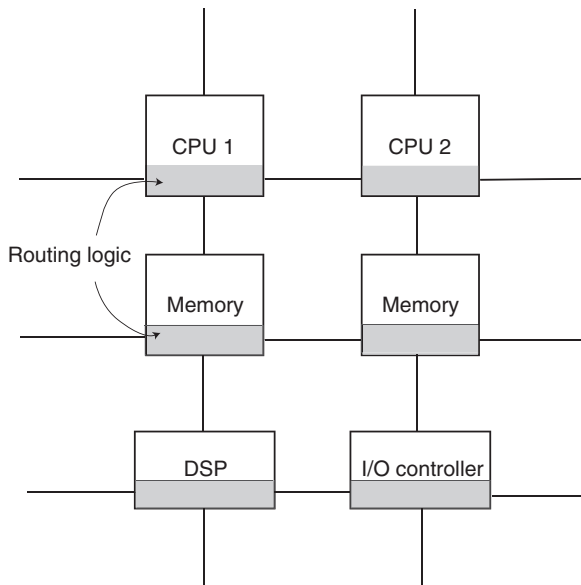


Figure 1.20 SOC system-level interconnection: network-on-chip approach.

TABLE 1.15 Interconnect Models for Different SOC Examples

SOC	Application	Interconnect Type
ClearSpeed CSX600 [59]	High Performance Computing	ClearConnect bus
NetSilicon NET +40 [184]	Networking	Custom bus
NXP LH7A404 [186]	Networking	AMBA bus
Intel PXA27x [132]	Mobile/wireless	PXBus
Matsushita i-Platform [176]	Media	Internal connect bus
Emulex InSpeed SOC320 [130]	Switching	Crossbar switch
MultiNOC [172]	Multiprocessing system	Network-on-chip

1.7 AN APPROACH FOR SOC DESIGN

Two important ideas in a design process are figuring out the requirements and specifications, and iterating through different stages of design toward an efficient and effective completion.

1.7.1 Requirements and Specifications

Requirements and specifications are fundamental concepts in any system design situation. There must be a thorough understanding of both before a design can begin. They are useful at the beginning and at the end of the design process: at the beginning, to clarify what needs to be achieved; and at the end, as a reference against which the completed design can be evaluated.

The system requirements are the largely externally generated criteria for the system. They may come from competition, from sales insights, from customer requests, from product profitability analysis, or from a combination. Requirements are rarely succinct or definitive of anything about the system. Indeed, requirements can frequently be unrealistic: “I want it fast, I want it cheap, and I want it now!”

It is important for the designer to analyze carefully the requirements expressions, and to spend sufficient time in understanding the market situation to determine all the factors expressed in the requirements and the priorities those factors imply. Some of the factors the designer considers in determining requirements include

- compatibility with previous designs or published standards,
- reuse of previous designs,
- customer requests/complaints,
- sales reports,
- cost analysis,
- competitive equipment analysis, and
- trouble reports (reliability) of previous products and competitive products.

The designer can also introduce new requirements based on new technology, new ideas, or new materials that have not been used in a similar systems environment.

The system specifications are the quantified and prioritized criteria for the target system design. The designer takes the requirements and must produce a succinct and definitive set of statements about the eventual system. The designer may have no idea of what the eventual system will look like, but usually, there is some “straw man” design in mind that seems to provide a feasibility framework to the specification. In any effective design process, it would be surprising if the final design significantly resembles the straw man design.

The specification does not complete any part of the design process; it initializes the process. Now the design can begin with the selection of components and approaches, the study of alternatives, and the optimization of the parts of the system.

1.7.2 Design Iteration

Design is always an iterative process. So, the obvious question is how to get the very first, initial design. This is the design that we can then iterate through and optimize according to the design criteria. For our purposes, we define several types of designs based on the stage of design effort.

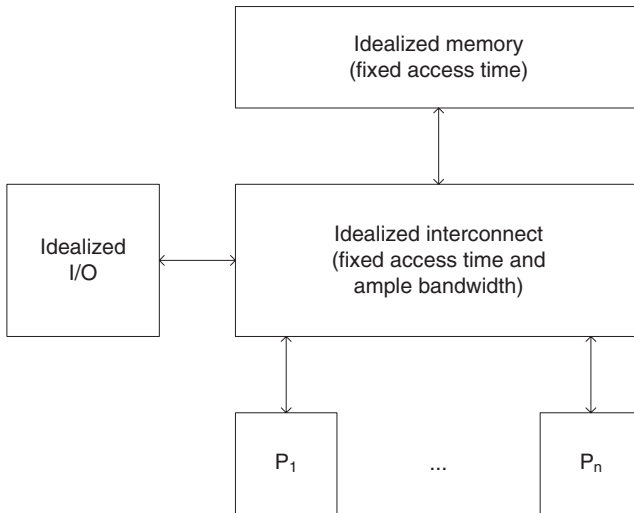
Initial Design This is the first design that shows promise in meeting the key requirements, while other performance and cost criteria are not considered. For instance, processor or memory or input/output (I/O) should be sized to meet high-priority real-time constraints. Promising components and their parameters are selected and analyzed to provide an understanding of their expected idealized performance and cost. Idealized does not mean ideal; it means a simplified model of the expected area occupied and computational or data bandwidth capability. It is usually a simple linear model of performance, such as the expected million instructions per second (MIPS) rate of a processor.

Optimized Design Once the base performance (or area) requirements are met and the base functionality is ensured, then the goal is to minimize the cost (area) and/or the power consumption or the design effort required to complete the design. This is the iterative step of the process. The first steps of this process use higher-fidelity tools (simulations, trial layouts, etc.) to ensure that the initial design actually does satisfy the design specifications and requirements. The later steps refine, complete, and improve the design according to the design criteria.

Figure 1.21 shows the steps in creating an initial design. This design is detailed enough to create a component view of the design and a corresponding projection of the component’s expected performance. This projection is, at this

1. Understand functional, cost, and real-time requirements.
2. Identify key requirements that the design must meet.
3. Prioritize the selection of processor, memory, and interconnect components based on their impact on key requirements.
4. Evaluate whether key requirements are met.
5. If yes, then initial design is completed.
6. If no, then try a different component selection; go to step 3.

Figure 1.21 The SOC initial design process.



n idealized processors (P) selected by function

Figure 1.22 Idealized SOC components.

step, necessarily simplified and referenced to here as the idealized view of the component (Figure 1.22).

System performance is limited by the component with the least capability. The other components can usually be modeled as simply presenting a delay to the critical component. In a good design, the most expensive component is the one that limits the performance of the system. The system's ability to process transactions should closely follow that of the limiting component. Typically, this is the processor or memory complex.

Usually, designs are driven by either (1) a specific real-time requirement, after which functionality and cost become important, or (2) functionality and/or throughput under cost–performance constraints. In case (1), the real-time constraint is provided by I/O consideration, which the processor–memory–interconnect system must meet. The I/O system then determines the performance, and any excess capability of the remainder of the system is usually used to add functionality to the system. In case (2), the object is to improve task

throughput while minimizing the cost. Throughput is limited by the most constrained component, so the designer must fully understand the trade-offs at that point. There is more flexibility in these designs, and correspondingly more options in determining the final design.

The purpose of this book is to provide an approach for determining the initial design by

- (a) describing the range of components—processors, memories, and interconnects—that are available in building an SOC;
- (b) providing examples of requirements for various domains of applications, such as data compression and encryption; and
- (c) illustrating how an initial design, or a reported implementation, can show promise in meeting specific requirements.

We explain this approach in Chapters 3–5 on a component by component basis to cover (a), with Chapter 6 covering techniques for system configuration and customization. Chapter 7 contains application studies to cover (b) and (c).

As mentioned earlier, the designer must optimize each component for processing and storage. This optimization process requires extensive simulation. We provide access to basic simulation tools through our associated web site.

1.12 PROBLEM SET

1. Suppose the TLB in Figure 1.18 had 256 entries (directly addressed). If the virtual address is 32 bits, the real memory is 512MB and the page size is 4KB, show the possible layout of a TLB entry. What is the purpose of the user ID in Figure 1.18 and what is the consequence of ignoring it?
2. Discuss possible arrangement of addressing the TLB.
3. Find an actual VLIW instruction format. Describe the layout and the constraints on the program in using the applications in a single instruction.
4. Find an actual vector instruction for vector ADD. Describe the instruction layout. Repeat for vector load and vector store. Is overlapping of vector instruction execution permitted? Explain.
5. For the pipelined processor in Figure 1.9, suppose instruction #3 sets the CC (condition code that can be tested by following a branch instruction) at the end of WB and instruction #4 is the condition branch. Without additional hardware support, what is the delay in executing instruction #5 if the branch is taken and if the branch is not taken?
6. Suppose we have four different processors; each does 25% of the application. If we improve two of the processors by 10 times, what would be the overall application speedup?
7. Suppose we have four different processors and all but one are totally limited by the bus. If we speed up the bus by three times and assume the processor performance also scales, what is the application speedup?
8. For the pipelined processor in Figure 1.9, assume the cache miss rate is 0.05 per instruction execution and the total cache miss delay is 20 cycles. For this processor, what is the achievable cycle per instruction (CPI)? Ignore other delays, such as branch delays.
9. Design validation is a very important SOC design consideration. Find several approaches specific to SOC designs. Evaluate each from the perspective of a small SOC vendor.
10. Find (from the Internet) two new VLIW DSPs. Determine the maximum number of operations issued in each cycle and the makeup of the operations (number of integer, floating point, branch, etc.). What is the stated maximum performance (operations per second)? Find out how this number was computed.
11. Find (from the Internet) two new, large FPGA parts. Determine the number of logic blocks (configurable logic blocks [CLBs]), the minimum cycle time, and the maximum allowable power consumption. What soft processors are supported?

2 Chip Basics: Time, Area, Power, Reliability, and Configurability

2.1 INTRODUCTION

The trade-off between cost and performance is fundamental to any system design. Different designs result either from the selection of different points on the cost–performance continuum or from differing assumptions about the nature of cost or performance.

The driving force in design innovation is the rapid advance in technology. The Semiconductor Industry Association (SIA) regularly makes projections, called the SIA road map, of technology advances, which become the basis and assumptions for new chip designs. While the projections change, the advance has been and is expected to continue to be formidable. Table 2.1 is a summary of the roadmap projections for the microprocessors with the highest performance introduced in a particular year [133]. With the advances in lithography, the transistors are getting smaller. The minimum width of the transistor gates is defined by the process technology. Table 2.1 refers to process technology generations in terms of nanometers; older generations are referred to in terms of microns (μm). So the previous generations are 65 and 90 nm, and 0.13 and 0.18 μm .

2.1.1 Design Trade-Offs

With increases in chip frequency and especially in transistor density, the designer must be able to find the best set of trade-offs in an environment of rapidly changing technology. Already the chip frequency projections have been called into question because of the resulting power requirements.

In making basic design trade-offs, we have five different considerations. The first is *time*, which includes partitioning instructions into events or cycles, basic pipelining mechanisms used in speeding up the instruction execution, and cycle time as a parameter for optimizing program execution. Second, we discuss *area*. The cost or area occupied by a particular feature is another important aspect of the architectural trade-off. Third, *power consumption*

TABLE 2.1 Technology Roadmap Projections

Year	2010	2013	2016
Technology generation (nm)	45	32	22
Wafer size, diameter (cm)	30	45	45
Defect density (per cm ²)	0.14	0.14	0.14
μ P die size (cm ²)	1.9	2.6	2.6
Chip frequency (GHz)	5.9	7.3	9.2
Million transistors per square centimeter	1203	3403	6806
Max power (W) high performance	146	149	130

affects both performance and implementation. Instruction sets that require more implementation area are less valuable than instruction sets that use less—unless, of course, they can provide commensurately better performance. Long-term *cost-performance ratio* is the basis for most design decisions. Fourth, *reliability* comes into play to cope with deep submicron effects. Fifth, *configurability* provides an additional opportunity for designers to trade off recurring and nonrecurring design costs.

FIVE BIG ISSUES IN SYSTEM-ON-CHIP (SOC) DESIGN

Four of the issues are obvious. *Die area* (manufacturing cost) and *performance* (heavily influenced by cycle time) are important basic SOC design considerations. Power consumption has also come to the fore as a design limitation. As technology shrinks feature sizes, reliability will dominate as a design consideration.

The fifth issue, configurability, is less obvious as an immediate design consideration. However, as we saw in Chapter 1, in SOC design, the non-recurring design costs can dominate the total project cost. Making a design flexible through reconfigurability is an important issue to broaden the market—and reduce the per part cost—for SOC design.

Configurability enables programmability in the field and can be seen to provide features that are “standardized in manufacturing while customized in application.” The cyclical nature of the integrated circuit industry between standardization and customization has been observed by Makimoto [163] and is known as Makimoto’s wave, as shown in Figure 2.1.

In terms of complexity, various trade-offs are possible. For instance, at a fixed feature size, area can be traded off for performance (expressed in term of execution time, T). Very large scale integration (VLSI) complexity theorists have shown that an $A \times T^n$ bound exists for processor designs, where n usually falls between 1 and 2 [247]. It is also possible to trade off time T for power P

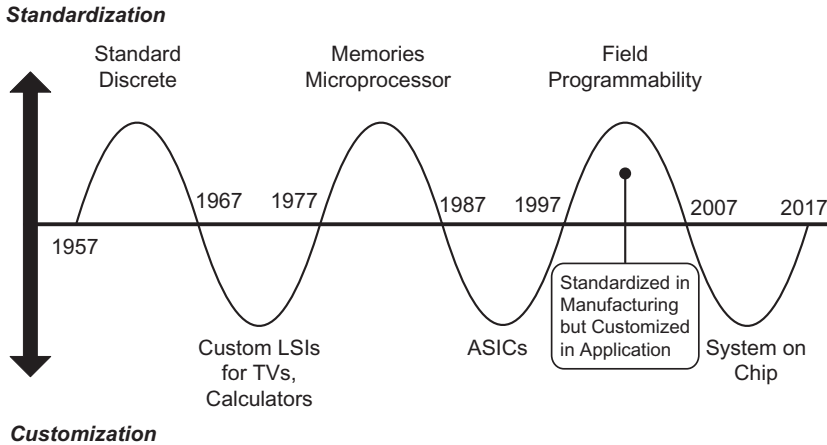


Figure 2.1 Makimoto's wave.

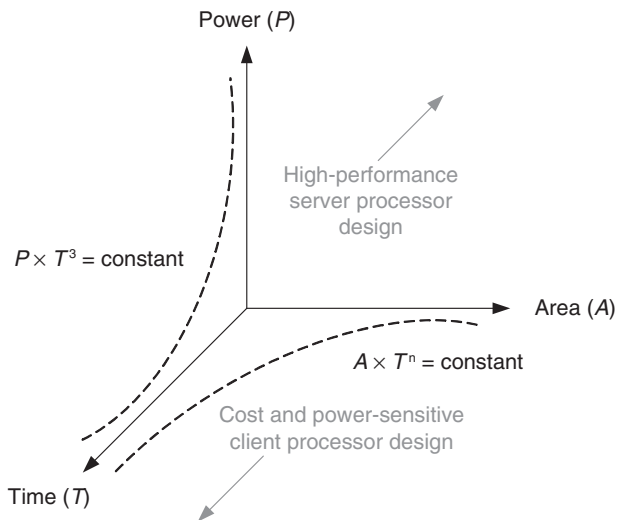


Figure 2.2 Processor design trade-offs.

with a $P \times T^3$ bound. Figure 2.2 shows the possible trade-off involving area, time, and power in a processor design [98]. Embedded and high-end processors operate in different design regions of this three-dimensional space. The power and area axes are typically optimized for embedded processors, whereas the time axis is typically for high-end processors.

This chapter deals with design issues in making these trade-offs. It begins with the issue of time. The ultimate measure of performance is the time required to complete required system tasks and functions. This depends on two factors: first, the organization and size of the processors and memories,

and the second, the basic frequency or clock rate at which these operate. We deal with the first factor in the next two chapters. In this chapter, we only look at the basic *processor cycle*—specifically, how much delay is incurred in a cycle and how instruction execution is partitioned into cycles. As almost all modern processors are pipelined, we look at the cycle time of pipelined processors and the partitioning of instruction execution into cycles. We next introduce a cost (area) model to assist in making manufacturing cost trade-offs. This model is restricted to on-chip or processor-type trade-offs, but it illustrates a type of system design model. As mentioned in Chapter 1, die cost is often but a small part of the total cost, but an understanding of it remains essential. Power is primarily determined by cycle time and the overall size of the design and its components. It has become a major constraint in most SOC designs. Finally, we look at reliability and reconfiguration and their impact on cost and performance.

2.1.2 Requirements and Specifications

The five basic SOC trade-offs provide a framework for analyzing SOC requirements so that these can be translated into specifications. Cost requirements coupled with market size can be translated into die cost and process technology. Requirements for wearables and weight limits translate into bounds on power or energy consumption, and limitations on clock frequency, which can affect heat dissipation. Any one of the trade-off criteria can, for a particular design, have the highest priority. Consider some examples:

- High-performance systems will optimize time at the expense of cost and power (and probably configurability, too).
- Low-cost systems will optimize die cost, reconfigurability, and design reuse (and perhaps low power).
- Wearable systems stress low power, as the power supply determines the system weight. Since such systems, such as cell phones, frequently have real-time constraints, its performance cannot be ignored.
- Embedded systems in planes and other safety-critical applications would stress reliability, with performance and design lifetime (configurability) being important secondary considerations.
- Gaming systems would stress cost—especially production cost—and, secondarily, performance, with reliability being a lesser consideration.

In considering requirements, the SOC designer should carefully consider each trade-off item to derive corresponding specifications. This chapter, when coupled with the essential understanding of the system components, which we will see in later chapters, provides the elements for SOC requirements translation into specifications and the beginning of the study of optimization of design alternatives.

2.2 CYCLE TIME

The notion of time receives considerable attention from processor designers. It is the basic measure of performance; however, breaking actions into cycles and reducing both cycle count and cycle times are important but inexact sciences.

The way actions are partitioned into cycles is important. A common problem is having unanticipated “extra” cycles required by a basic action such as a cache miss. Overall, there is only a limited theoretical basis for cycle selection and the resultant partitioning of instruction execution into cycles. Much design is done on a pragmatic basis.

In this section, we look at some techniques for instruction partitioning, that is, techniques for breaking up the instruction execution time into manageable and fixed time cycles. In a pipelined processor, data flow through stages much as items flow on an assembly line. At the end of each stage, a result is passed on to a subsequent stage and new data enter. Within limits, the shorter the cycle time, the more productive the pipeline. The partitioning process has its own overhead, however, and very short cycle times become dominated by this overhead. Simple cycle time models can optimize the number of pipeline stages.

THE PIPELINED PROCESSOR

At one time, the concept of *pipelining* in a processor was treated as an advanced processor design technique. For the past several decades, pipelining has been an integral part of any processor or, indeed, controller design. It is a technique that has become a basic consideration in defining cycle time and execution time in a processor or system.

The trade-off between cycle time and number of pipeline stages is treated in the section on *optimum pipeline*.

2.2.1 Defining a Cycle

A cycle (of the clock) is the basic time unit for processing information. In a synchronous system, the clock rate is a fixed value and the cycle time is determined by finding the maximum time to accomplish a frequent operation in the machine, such as an add or register data transfer. This time must be sufficient for data to be stored into a specified destination register (Figure 2.3). Less frequent operations that require more time to complete require multiple cycles.

A cycle begins when the instruction decoder (based on the current instruction opcode) specifies the values for the registers in the system. These control

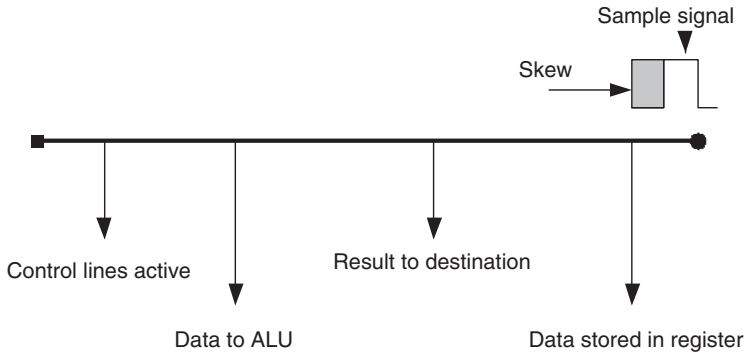


Figure 2.3 Possible sequence of actions within a cycle.

values connect the output of a specified register to another register or an adder or similar object. This allows data from source registers to propagate through designated combinatorial logic into the destination register. Finally, after a suitable setup time, all registers are sampled by an edge or pulse produced by the clocking system.

In a synchronous system, the cycle time is determined by the sum of the worst-case time for each step or action within the cycle. However, the clock itself may not arrive at the anticipated time (due to propagation or loading effects). We call the maximum deviation from the expected time of clock arrival the (uncontrolled) clock skew.

In an asynchronous system, the cycle time is simply determined by the completion of an event or operation. A completion signal is generated, which then allows the next operation to begin. Asynchronous design is not generally used within pipelined processors because of the completion signal overhead and pipeline timing constraints.

2.2.2 Optimum Pipeline

A basic optimization for the pipeline processor designer is the partitioning of the pipeline into concurrently operating segments. A greater number of segments allow a higher maximum speedup. However, each new segment carries clocking overhead with it, which can adversely affect performance.

If we ignore the problem of fitting actions into an integer number of cycles, we can derive an optimal cycle time, Δt , and hence the level of segmentation for a simple pipelined processor.

Assume that the total time to execute an instruction without pipeline segments is T nanoseconds (Figure 2.4a). The problem is to find the optimum number of segments S to allow clocking and pipelining. The ideal delay through a segment is $T/S = T_{\text{seg}}$. Associated with each segment is partitioning overhead. This clock overhead time C (in nanoseconds), includes clock skew and any register requirements for data setup and hold.

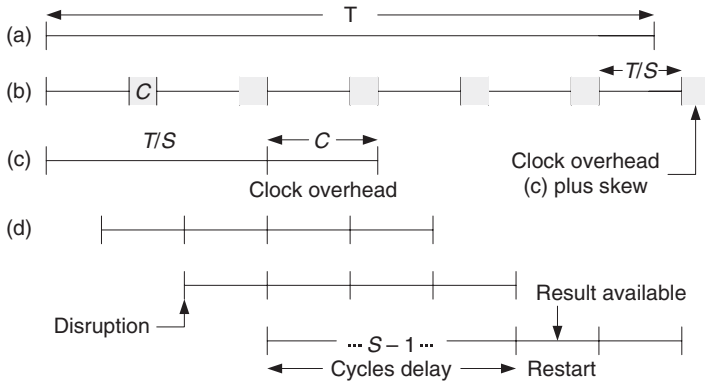


Figure 2.4 Optimal pipelining. (a) Unclocked instruction execution time, T . (b) T is partitioned into S segments. Each segment requires C clocking overhead. (c) Clocking overhead and its effect on cycle time, T/S . (d) Effect of a pipeline disruption (or a stall in the pipeline).

Now, the actual cycle time (Figure 2.4c) of the pipelined processor is the ideal cycle time T/S plus the overhead:

$$\Delta t = \frac{T}{S} + C.$$

In our idealized pipelined processor, if there are no code delays, it processes instructions at the rate of one per cycle, but delays can occur (primarily due to incorrectly guessed or unexpected branches). Suppose these interruptions occur with frequency b and have the effect of invalidating the $S - 1$ instructions prepared to enter, or already in, the pipeline (representing a “worst-case” disruption, Figure 2.4d). There are many different types of pipeline interruption, each with a different effect, but this simple model illustrates the effect of disruptions on performance.

Considering pipeline interruption, the performance of the processor is

$$\text{Performance} = \frac{1}{1 + (S - 1)b} \text{ instructions per cycle.}$$

The throughput (G) can be defined as

$$\begin{aligned} G &= \frac{\text{performance}}{\Delta t} \text{ instructions/ns} \\ &= \left(\frac{1}{1 + (S - 1)b} \right) \times \left(\frac{1}{(T/S) + C} \right). \end{aligned}$$

If we find the S for which

$$\frac{dG}{dS} = 0,$$

we can find S_{opt} , the optimum number of pipeline segments:

$$S_{\text{opt}} = \sqrt{\frac{(1-b)T}{bC}}.$$

Once an initial S has been determined, the total instruction execution latency (T_{instr}) is

$$T_{\text{instr}} = T + S \times (\text{clocking overhead}) = T + SC, \text{ or } S(T_{\text{seg}} + C) = S\Delta t.$$

Finally, we compute the throughput performance G in (million) instructions per second.

Suppose $T = 12.0\text{ns}$ and $b = 0.2$, $C = 0.5\text{ns}$. Then, $S_{\text{opt}} = 10$ stages.

This S_{opt} as determined is simplistic—functional units cannot be arbitrarily divided, integer cycle boundaries must be observed, and so on. Still, determining S_{opt} can serve as a design starting point or as an important check on an otherwise empirically optimized design.

The preceding discussion considers a number of pipeline segments, S , on the basis of performance. Each time a new pipeline segment is introduced, additional cost is added, which is not factored into the analysis. Each new segment requires additional registers and clocking hardware. Because of this, the optimum number of pipeline segments (S_{opt}) ought to be thought of as a probable upper limit to the number of useful pipeline segments that a particular processor can employ.

2.2.3 Performance

High clock rates with small pipeline segments may or may not produce better performance. Indeed, given problems in wire delay scaling, there is an immediate question of how projected clock rates are to be achieved. There are two basic factors enabling clock rate advances: (1) increased control over clock overhead and (2) an increased number of segments in the pipelines. Figure 2.5 shows that the length (in gate delays) of a pipeline segment has decreased significantly, probably by more than five times, measured in units of a standard gate delay. This standard gate has one input and drives four similar gates as output. Its delay is referred to as a fan-out of four (FO4) gate delay.

Low clock overhead (small C) may enable increased pipeline segmentation, but performance does not correspondingly improve unless we also decrease the probability of pipeline disruption, b . In order to accomplish this high clock rate, processors also employ large branch table buffers and branch vector prediction tables, significantly decreasing delays due to branching. However, disruptions can also come from cache misses, and this requires another strat-

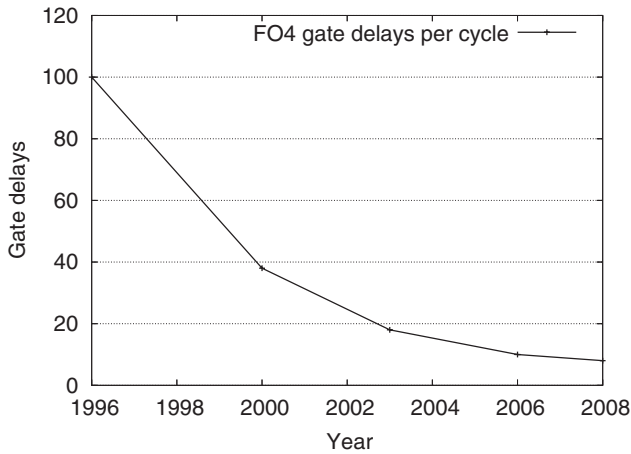


Figure 2.5 Number of gate delays (FO4) allowed in a cycle.

egy: multilevel, very large on-die caches. Often these cache structures occupy 80–90% of the die area. The underlying processor is actually less important than the efficiency of the cache memory system in achieving performance.

2.3 DIE AREA AND COST

Cycle time, machine organization, and memory configuration determine machine performance. Determining performance is relatively straightforward when compared to the determination of overall cost.

A good design achieves an optimum cost–performance trade-off at a particular target performance. This determines the *quality* of a processor design.

In this section, we look at the marginal cost to produce a system as determined by the die area component. Of course, the system designer must be aware of significant side effects that die area has on the fixed and other variable costs. For example, a significant increase in the complexity of a design may directly affect its serviceability or its documentation costs, or the hardware development effort and time to market. These effects must be kept in mind, even when it is not possible to accurately quantify their extent.

2.3.1 Processor Area

SOCs usually have die sizes of about 10–15 mm on a side. This die is produced in bulk from a larger wafer, perhaps 30 cm in diameter (about 12 in.). It might seem that one could simply expand the chip size and produce fewer chips from the wafer, and these larger chips could readily accommodate any function that the designer might wish to include. Unfortunately, neither the silicon wafers nor processing technologies are perfect. Defects randomly occur over the

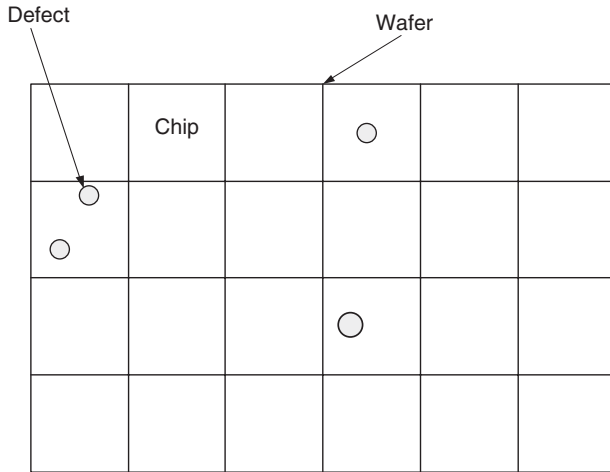


Figure 2.6 Defect distribution on a wafer.

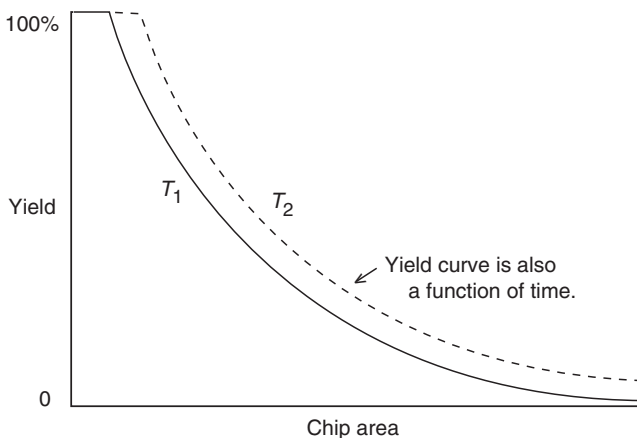


Figure 2.7 Yield versus chip area at various points in time.

wafer surface (Figure 2.6). Large chip areas require an absence of defects over that area. If chips are too large for a particular processing technology, there will be little or no yield. Figure 2.7 illustrates yield versus chip area.

A good design is not necessarily the one that has the maximum yield. Reducing the area of a design below a certain amount has only a marginal effect on yield. Additionally, small designs waste area because there is a required area for pins and for separation between the adjacent die on a wafer.

The area available to a designer is a function of the manufacturing processing technology. This includes the purity of the silicon crystals, the absence of dust and other impurities, and the overall control of the process technology. Improved manufacturing technology allows larger dice to be realized with higher yields. As photolithography and process technology improve, their

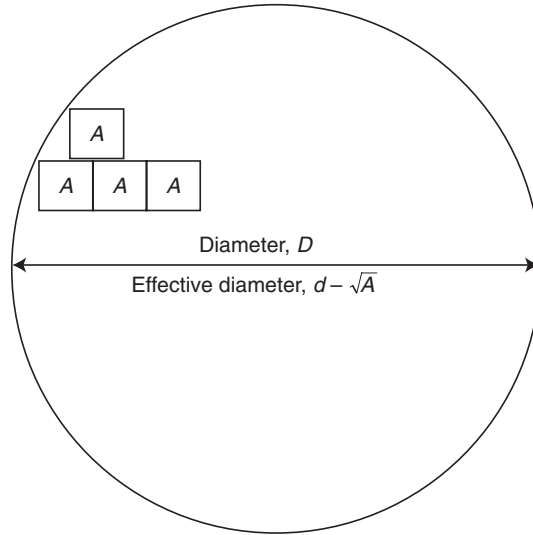


Figure 2.8 Number of die (of area A) on a wafer of diameter d .

design parameters do not scale uniformly. The successful designer must be aggressive enough in anticipating the movement of technology so that, although early designs may have low yield, with the advance of technology, the design life is extended and the yield greatly improves, thus allowing the design team to amortize fixed costs over a broad base of products.

Suppose a die with square aspect ratio has area A . About N of these dice can be realized in a wafer of diameter d (Figure 2.8):

$$N \approx \frac{\pi}{4A} (d - \sqrt{A})^2.$$

This is the wafer area divided by the die area with diameter correction. Now suppose there are N_G good chips and N_D point defects on the wafer. Even if $N_D > N$, we might expect several good chips since the defects are randomly distributed and several defects would cluster on defective chips, sparing a few.

Following the analysis of Ghandi [109], suppose we add a random defect to a wafer; N_G/N is the probability that the defect ruins a good die. Note that if the defect hits an already bad die, it would cause no change to the number of good die. In other words, the change in the number of good die (N_G), with respect to the change in the number of defects (N_D), is

$$\begin{aligned} \frac{dN_G}{dN_D} &= -\frac{N_G}{N} \\ \frac{1}{N_G} dN_G &= -\frac{1}{N} dN_D. \end{aligned}$$

Integrating and solving

$$\ln N_G = -\frac{N_D}{N} + C.$$

To evaluate C, note that when $N_G = N$, then $N_D = 0$; so, C must be $\ln(N)$.
Then the yield is

$$\text{Yield} = \frac{N_G}{N} = e^{-N_D/N}.$$

This describes a Poisson distribution of defects.

If ρ_D is the defect density per unit area, then

$$N_D = \rho_D \times (\text{wafer area}).$$

For large wafers $d \gg \sqrt{A}$, the diameter of the wafer is significantly larger than the die side and

$$(d - \sqrt{A})^2 \approx d^2$$

and

$$\frac{N_D}{N} = \rho_D A,$$

so that

$$\text{Yield} = e^{-\rho_D A}.$$

Figure 2.9 shows the projected number of good die as a function of die area for several defect densities. Currently, a modern fab facility would have ρ_D between 0.15–0.5, depending on the maturity of the process and the expense of the facility.

Large die sizes are very costly. Doubling the die area has a significant effect on yield for an already large $\rho_D \times A$ (≈ 5 –10 or more). Thus, the large die designer gambles that technology will lower ρ_D in time to provide a sufficient yield for a profitable product.

2.3.2 Processor Subunits

Within a system or processor, the amount of area that a particular subunit of a design occupies is a primary measure of its cost. In making design choices or in evaluating the relative merits of a particular design choice, it is frequently useful to use the principle of marginal utility: Assume we have a complete base design and some additional pins/area available to enhance the design. We

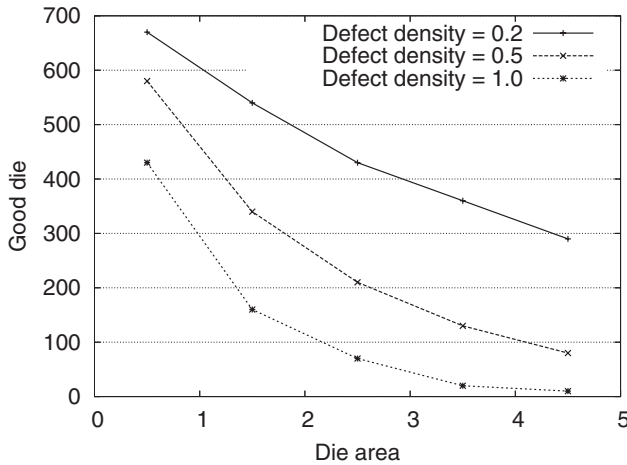


Figure 2.9 Number of good die versus die area for several defect densities.

select the design enhancement that best uses the available pins and area. In the absence of pinout information, we assume that area is a dominant factor in a particular trade-off.

The obvious unit of area is *millimeter square*, but since photolithography and geometries' resulting minimum feature sizes are constantly shifting, a dimensionless unit is preferred. Among others, Mead and Conway [170] used the unit λ , the fundamental resolution, which is the distance from which a geometric feature on any one layer of mask may be positioned from another. The minimum size for a diffusion region would be 2λ with a necessary allowance of 3λ between adjacent diffusion regions.

If we start with a device $2\lambda \times 2\lambda$, then a device of nominal $2\lambda \times 2\lambda$ can extend to $4\lambda \times 4\lambda$. We need at least 1λ isolation from any other device or $25\lambda^2$ for the overall device area. Thus, a single transistor is $4\lambda^2$, positioned in a minimum region of $25\lambda^2$.

The minimum feature size (f) is the length of one polysilicon gate, or the length of one transistor, $f = 2\lambda$. Clearly, we could define our design in terms of λ^2 , and any other processor feature (gate, register size, etc.) can be expressed as a number of transistors. Thus, the selection of the area unit is somewhat arbitrary. However, a better unit represents primary architectural trade-offs. One useful unit is the register bit equivalent (rbe). This is defined to be a six-transistor register cell and represents about $2700\lambda^2$. This is significantly more than six times the area of a single transistor, since it includes larger transistors, their interconnections, and necessary inter-bit isolating spaces.

A static RAM (SRAM) cell with lower bandwidth would use less area than an rbe, and a DRAM bit cell would use still less. Empirically, they would have the relationship shown in Table 2.2.

In the table, the area for the register file is determined by the number of register bits and the number of ports (P) available to access the file:

TABLE 2.2 Summary of Technology-Independent Relative Area Measures, rbe and A (These Can Be Converted to True Area for Any Given Feature Size, f)

Item: Size in rbe	
1 register bit (rbe)	1.0 rbe
1 static RAM bit in an on-chip cache	0.6 rbe
1 DRAM bit	0.1 rbe
rbe corresponds to (in feature size: f)	1 rbe = $675f^2$
Item: Size in A Units	
A corresponds to 1 mm^2 with $f = 1\text{ }\mu\text{m}$.	
1 A	$=f^2 \times 10^6$ (f in μm)
or about	≈ 1481 rbe
A simple integer file (1 read + 1 read/write) with 32 words of 32 bits per word	$= 1444$ rbe
or about	$\approx 1 A$ ($= 0.975 A$)
A 4-KB direct mapped cache	$= 23,542$ rbe
or about	$\approx 16 A$
Generally a simple cache (whose tag and control bits are less than one-fifth the data bits) uses	$= 4 A/KB$
Simple Processors (Approximation)	
A 32-bit processor (no cache and no floating point)	$= 50 A$
A 32-bit processor (no cache but includes 64-bit floating point)	$= 100 A$
A 32-bit (signal) processor, as above, with vector facilities but no cache or vector memory	$= 200 A$
Area for interunit latches, buses, control, and clocking	Allow an additional 50% of the processor area.
Xilinx FPGA	
A slice (2 LUTs + 2 FFs + MUX)	$= 700$ rbe
A configurable logic block (4 slices) Virtex 4	$= 2800$ rbe $\approx 1.9 A$
A 18-KB block RAM	$= 12,600$ rbe $\approx 8.7 A$
An embedded PPC405 core	$\approx 250 A$

Area of register file = (number of regs + $3P$)(bits per reg + $3P$) rbe.

The cache area uses the SRAM bit model and is determined by the total number of cache bits, including the array, directory, and control bits.

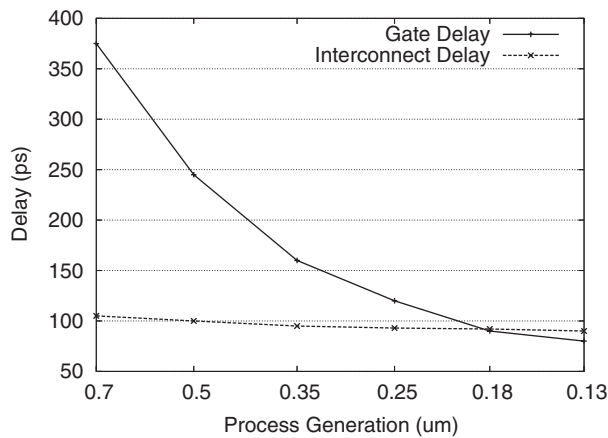
The number of rbe on a die or die section rapidly becomes very large, so it is frequently easier to use a still larger unit. We refer to this unit simply as A and define it as 1 mm^2 of die area at $f = 1\text{ }\mu\text{m}$. This is also the area occupied by a 32×32 bit three-ported register file or 1481 rbe.

Transistor density, rbe, and A all scale as the square of the feature size. As seen from Table 2.3, for feature size f , the number of A in 1 mm^2 is simply $(1/f)^2$. There are almost 500 times as many transistors of rbe in 1 mm^2 of a technology with a feature size of 45 nm as there are with the reference $1\text{-}\mu\text{m}$ feature size.

TABLE 2.3 Density in A Units for Various Feature Sizes

Feature Size (μm)	Number of A per mm^2
1.000	1.00
0.350	8.16
0.130	59.17
0.090	123.46
0.065	236.69
0.045	493.93

One A is 1481 rbe.

**Figure 2.10** The dominance of wire delay over gate delay.

2.4 IDEAL AND PRACTICAL SCALING

As feature sizes shrink and transistors get smaller, one expects the transistor density to improve with the square of the change in feature size. Similarly, transistor delay (or gate delay) should decrease linearly with feature size (corresponding to the decrease in capacitance). Practical scaling is different as wire delay, and wire density does not scale at the same rate as transistors scale. Wire delay remains almost constant as feature sizes shrink since the increase in resistance offsets the decrease in length and capacitance. Figure 2.10 illustrates the increasing dominance of wire delay over gate delay especially in feature sizes less than $0.10\mu\text{m}$. Similarly for feature sizes below $0.20\mu\text{m}$, transistor density improves at somewhat less than the square of the feature size. A suggested scaling factor of 1.5 is commonly considered more accurate, as shown in Figure 2.11; that is, scaling occurs at $(f_1/f_2)^{1.5}$ rather than at $(f_1/f_2)^2$. What actually happens during scaling is more complex. Not only does the feature size shrink but other aspects of a technology also change and usually

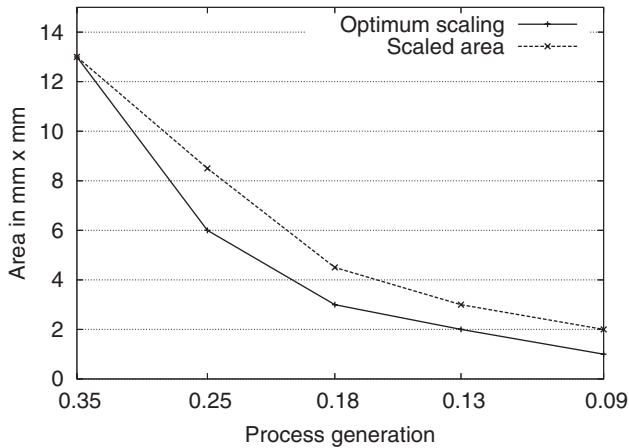


Figure 2.11 Area scaling with optimum and “practical” shrinkage.

improve. Thus, copper wires become available as well as many more wiring layers and improved circuit designs. Major technology changes can affect scaling in a discontinuous manner. The effects of wire limitations can be dramatically improved, so long as the designer is able to use all the attributes of the new technology generation. The simple scaling of a design might only scale as 1.5, but a new implementation taking advantage of all technology features could scale at 2. For simplicity in the remainder of the text, we will use ideal scaling with the understanding as above.

Study 2.1 A Baseline SOC Area Model

The key to efficient system design is chip floor planning. The process of chip floor planning is not much different from the process of floor-planning a residence. Each functional area of the processor must be allocated sufficient room for its implementation. Functional units that frequently communicate must be placed close together. Sufficient room must be allocated for connection paths.

To illustrate possible trade-offs that can be made in optimizing the chip floor plan, we introduce a baseline system with designated areas for various functions. The area model is based upon empirical observations made of existing chips, design experience, and, in some cases, logical deduction (e.g., the relationship between a floating-point adder and an integer ALU). The chip described here ought not to be considered optimal in any particular sense, but rather a typical example of a number of designs in the marketplace today.

The Starting Point. The design process begins with an understanding of the parameters of the semiconductor process. Suppose we expect to be able to use a manufacturing process that has a defect density of 0.2 defect per square centimeter; for economic reasons, we target an initial yield of about 95%:

$$Y = e^{-\rho_D A},$$

where $\rho_D = 0.2$ defect per square centimeter, $Y = 0.95$. Then,

$$A = 25 \text{ mm}^2$$

or approximately 0.25 cm^2 .

So the chip area available to us is 25 mm^2 . This is the total die area of the chip, but such things as pads for the wire bonds that connect the chip to the external world, drivers for these connections, and power supply lines all act to decrease the amount of chip area available to the designer. Suppose we allow 12% of the chip area—usually around the periphery of the chip—to accommodate these functions, then the net area will be 22 mm^2 (Figure 2.12).

Feature Size. The smaller the feature size, the more logic that can be accommodated within a fixed area. At $f = 65 \text{ nm}$, we have about $5200A$ or area units in 22 mm^2 .

The Architecture. Almost by definition, each system is different with different objectives. For our example, assume that we need the following:

- a small 32-bit core processor with an 8 KB I-cache and a 16 KB D-cache;
- two 32-bit vector processors, each with 16 banks of $1 \text{ K} \times 32b$ vector memory; an 8 KB I-cache and a 16 KB D-cache for scalar data;
- a bus control unit;
- directly addressed application memory of 128 KB ; and
- a shared L2 cache.

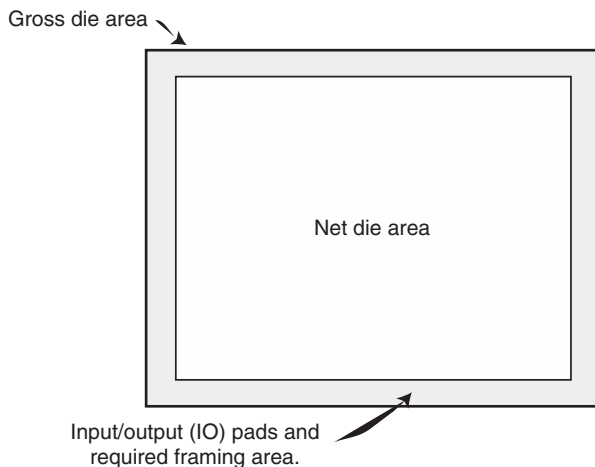


Figure 2.12 Net die area.

An Area Model. The following is a breakdown of the area required for various units used in the system.

Unit	Area (A)
Core processor (32^b)	100
Core cache (24 KB)	96
Vector processor #1	200
Vector registers and cache #1	$256 + 96$
Vector processor #2	200
Vector registers and cache #2	352
Bus and bus control (50%)	See below 650
Application memory (128 KB)	512
Subtotal	2462

Latches, Buses, and (Interunit) Control. For each of the functional units, there is a certain amount of overhead to accommodate nonspecific storage (latches), interunit communications (buses), and interunit control. This is allocated as 10% overhead for latches and 40% overhead for buses, routing, clocking, and overall control.

Total System Area. The designated processor elements and storage occupy $2462A$. This leaves a net of $5200 - 2462 = 2738A$ available for cache. Note that the die is highly storage oriented. The remaining area will be dedicated to the L2 cache.

Cache Area. The net area available for cache is $2738A$. However, bits and pieces that may be unoccupied on the chip are not always useful to the cache designer. These pieces must be collected into a reasonably compact area that accommodates efficient cache designs.

For example, where the available area has a large height/width (aspect) ratio, it may be significantly less useful than a more compact or square area. In general, at this early stage of microprocessor floor planning, we allocate another 10% overhead to aspect ratio mismatch. This leaves a net available area for cache of about $2464A$.

This gives us about 512 KB for the L2 cache. Is this reasonable? At this point, all we can say is that this much cache fits on the die. We now must look to the application and determine if this allocation gives the best performance. Perhaps a larger application storage or another vector processor and a smaller L2 would give better performance. Later in the text we consider such performance issues.

An example baseline floor plan is shown in Figure 2.13. A summary of area design rules follow:

1. Compute the target chip size from the target yield and defect density.
2. Compute the die cost and determine whether it is satisfactory.

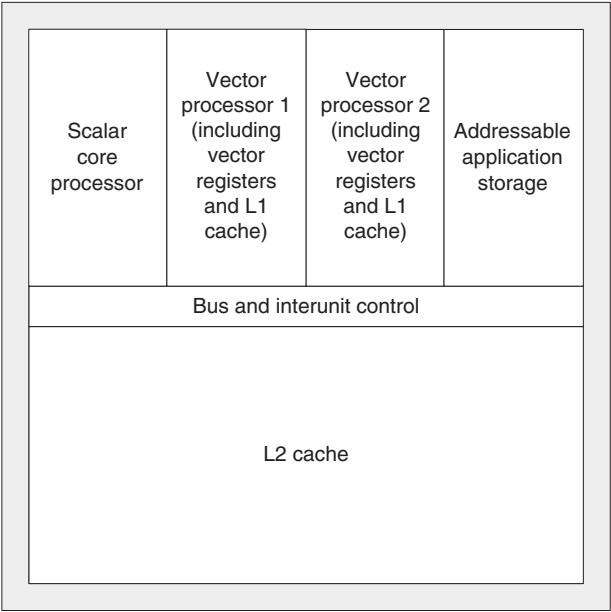


Figure 2.13 A baseline die floor plan.

- 3. Compute the net available area. Allow 10–20% (or other appropriate factor) for pins, guard ring, power supplies, and so on.
- 4. Determine the **rbe** size from the minimum feature size.
- 5. Allocate the area based on a trial system architecture until the basic system size is determined.
- 6. Subtract the basic system size (5) from the net available area (3). This is the die area available for cache and storage optimization.

Note that in this study (and more surely with much small feature sizes), most of the die area is dedicated to storage of one type or another. The basic processor area is around 20%, allowing for a partial allocation of bus and control area. Thus, however rough our estimate of processor core and vector processor area, it is likely to have little effect on the accuracy of the die allocation so long as our storage estimates are accurate. There are a number of commercial tools available for chip floor planning in specific design situations.

2.5 POWER

Growing demands for wireless and portable electronic appliances have focused much attention recently on power consumption. The SIA road map points to

TABLE 2.4 Some Power Operating Environments [133]

Type	Power/Die	Source and Environment
Cooled high power	70.0W	Plug-in, chilled
High power	10.0–50.0W	Plug-in, fan
Low power	0.1–2.0W	Rechargeable battery
Very low power	1.0–100.0mW	AA batteries
Extremely low power	1.0–100.0 μ W	Button battery

increasingly higher power for microprocessor chips because of their higher operating frequency, higher overall capacitance, and larger size. Power scales indirectly with feature size, as its primary determinate is frequency.

Some power environments are shown in Table 2.4.

At the device level, total power dissipation (P_{total}) has two major sources: dynamic or switching power and static power caused by leakage current:

$$P_{\text{total}} = \frac{CV^2\text{freq}}{2} + I_{\text{leakage}}V,$$

where C is the device capacitance; V is the supply voltage; freq is the device switching frequency; and I_{leakage} is the leakage current. Until recently, switching loss was the dominant factor in dissipation, but now static power is increasing. On the other hand, gate delays are roughly proportional to $CV/(V - V_{\text{th}})^2$, where V_{th} is the threshold voltage (for logic-level switching) of the transistors.

As feature sizes decrease, so do device sizes. Smaller device sizes result in reduced capacitance. Decreasing the capacitance decreases both the dynamic power consumption and the gate delays. As device sizes decrease, the electric field applied to them becomes destructively large. To increase the device reliability, we need to reduce the supply voltage V . Reducing V effectively reduces the dynamic power consumption but results in an increase in the gate delays. We can avoid this loss by reducing V_{th} . On the other hand, reducing V_{th} increases the leakage current and, therefore, the static power consumption. This has an important effect on design and production; there are two device designs that must be accommodated in production:

1. the high-speed device with low V_{th} and high static power; and
2. the slower device maintaining V_{th} and V at the expense of circuit density and low static power.

In either case, we can reduce switching loss by lowering the supply voltage, V . Chen et al. [55] showed that the drain current is proportional to

$$I = (V - V_{\text{th}})^{1.25},$$

where again V is the supply voltage.

From our discussion above, we can see that the signal transition time and frequency scale with the charging current. So, the maximum operating frequency is also proportional to $(V - V_{th})^{1.25}/V$. For values of V and V_{th} of interest, this means that frequency scales with the supply voltage, V .

Assume V_{th} is 0.6V; suppose we reduce the supply voltage by one-half, say, from 3.0 to 1.5V, the operating frequency is also reduced by about one-half. So, reducing the supply voltage by half also reduces the operating frequency by half.

Now by the power equation (since the voltage and frequency were halved), the total power consumption is one-eighth of the original. Thus, if we take an existing design optimized for frequency and modify that design to operate at a lower voltage, the frequency is reduced by approximately the cube root of the original (dynamic) power:

$$\frac{\text{freq}_1}{\text{freq}_2} = \sqrt[3]{\frac{P_2}{P_1}}.$$

It is important to understand the distinction between scaling the frequency of an existing design and that of a power-optimized implementation. Power-optimized implementations differ from performance-optimized implementations in several ways.

Power-optimized implementations use less chip area not only because of reduced requirements for power supply and clock distributions but also, and more importantly, because of reduced performance targets. Performance-oriented designs use a great deal of area to achieve marginally improved performance, as in very large floating-point units, minimum-skew clock distribution networks, or maximally sized caches. Power dissipation, not performance, is the most critical issue for applications such as portable and wireless processors running on batteries. Some battery capacities are shown in Table 2.5.

For SOC designs to run on battery power for an extended period, the entire system power consumption must remain very small (in the order of a milliwatt). As a result, power management must be implemented from the system architecture and operating system down to the logic gate level.

There is another power constraint, *peak power*, which the designer cannot ignore. In any design, the power source can only provide a certain current at the specified voltage; going beyond this, even as a transient, can cause logic errors or worse (damaging the power source).

TABLE 2.5 Battery Capacity and Duty Cycle

Type	Energy Capacity (mAh)	Duty Cycle/Lifetime	At Power
Rechargeable	10,000	50 h (10–20% duty)	400 mW–4 W
2 × AA	4000	0.5 year (10–20% duty)	1–10 mW
Button	40	5 years (always on)	1 μW

2.6 AREA-TIME-POWER TRADE-OFFS IN PROCESSOR DESIGN

Processor design trade-offs are quite different for our two general classes of processors:

1. *Workstation Processor.* These designs are oriented to high clock frequency and AC power sources (excluding laptops). Since they are not area limited as the cache occupies most die area, the designs are highly elaborated (superscalar with multithreading).
2. *Embedded Processor Used in SOC.* Processors here are generally simpler in control structure but may be quite elaborate in execution facilities (e.g., digital signal processor [DSP]). Area is a factor as is design time and power.

2.6.1 Workstation Processor

To achieve a general-purpose performance, the designer assumes ample power. The most basic trade-off is between high clock rates and the resulting power consumption. Up until the early 1990s, emitter coupled logic (ECL) using bipolar technology was dominant in high-performance applications (mainframes and supercomputers). At power densities of $80\text{W}/\text{cm}^2$, the module package required some form of liquid cooling. An example from this period is the Hitachi M-880 (Figure 2.14). A $10 \times 10\text{ cm}$ module consumed 800W. The

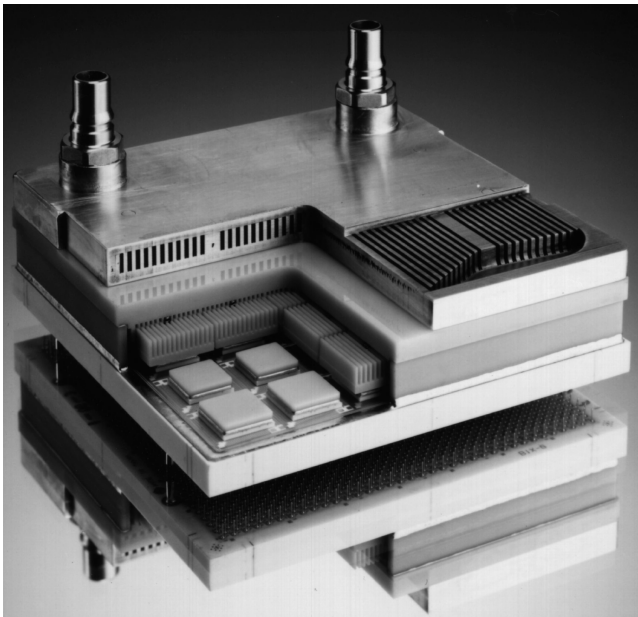


Figure 2.14 Hitachi processor module. The Hitachi M-880 was introduced about 1991 [143]. Module is $10.6 \times 10.6\text{ cm}$, water-cooled and dissipated at 800W.

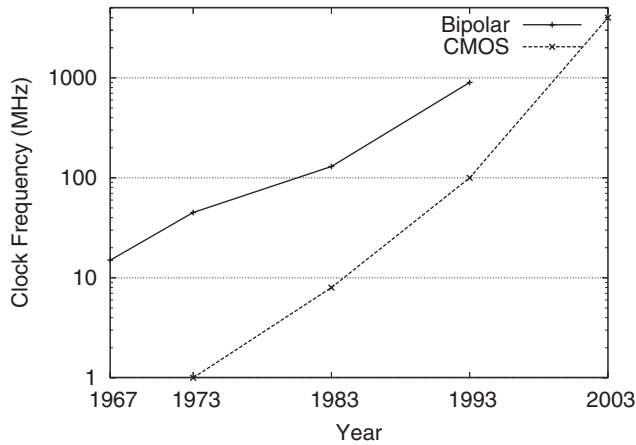


Figure 2.15 Processor frequency for bipolar and CMOS over time. Generally, CMOS frequency scaling ceased in around 2003 at around 3.5 GHz due to power limitations.

module contained 40 dice, sealed in helium gas with chilled water pumped across a water jacket at the top of the module. As CMOS performance approached bipolar's, the extraordinary cost of such a cooling system could no longer be sustained, and the bipolar era ended (see Figure 2.15). Now CMOS has reached the same power densities, and similar cooling techniques would have to be reconsidered if chip frequencies were to continue to increase. In fact, after 2003 the useful chip frequency stabilized at about 3.5 GHz.

2.6.2 Embedded Processor

System-on-a-chip-type implementations have a number of advantages. The requirements are generally known. So, memory sizes and real-time delay constraints can be anticipated. Processors can be specialized to a particular function. In doing so, usually clock frequency (and power) can be reduced as performance can be regained by straightforward concurrency in the architecture (e.g., use of a simple very long instruction word [VLIW] for DSP applications). The disadvantages of SOC compared to processor chips are available design time/effort and intra-die communications between functional units. In SOC, the market for any specific system is relatively small; hence, the extensive custom optimization used in processor dies is difficult to sustain, so off-the-shelf core processor designs are commonly used. As the storage size for programs and data may be known at design time, specific storage structures can be included on-chip. These are either SRAM or a specially designed DRAM (as ordinary DRAM uses an incompatible process technology). With multiple storage units, multiple processors (some specialized, some generic), and specialized controllers, the problem is designing a robust bus hierarchy to ensure timely communications. A comparison between the two design classes is shown in Table 2.6.

TABLE 2.6 A Typical Processor Die Compared with a Typical SOC Die

	Processor on a Chip	SOC
Area used by storage	80% cache	50% ROM/RAM
Clock frequency	3.5 GHz	0.5 GHz
Power	≥50 W	≤10 W
Memory	≥1-GB DRAM	Mostly on-die

2.7 RELIABILITY

The fourth important design dimension is reliability [218], also referred to as dependability and fault tolerance. As with cost and power, there are many more factors that contribute to reliability than what is done on a processor or SOC die.

Reliability is related to die area, clock frequency, and power. Die area increases the amount of circuitry and the probability of a fault, but it also allows the use of error correction and detection techniques. Higher clock frequencies increase electrical noise and noise sensitivity. Faster circuits are smaller and more susceptible to radiation.

Not all failures or errors produce faults, and indeed not all faults result in incorrect program execution. Faults, if detected, can be masked by error-correcting codes (ECCs), instruction retry, or functional reconfiguration.

First, some definitions:

1. A *failure* is a deviation from a design specification.
2. An *error* is a failure that results in an incorrect signal value
3. A *fault* is an error that manifests itself as an incorrect logical result.
4. A *physical fault* is a failure caused by the environment, such as aging, radiation, temperature, or temperature cycling. The probability of physical faults increases with time.
5. A *design fault* is a failure caused by a design implementation that is inconsistent with the design specification. Usually, design faults occur early in the lifetime of a design and are reduced or eliminated over time.

2.7.1 Dealing with Physical Faults

From a system point of view, we need to create processor and subunit configurations that are robust over time.

Let the probability of a fault occurrence be $P(t)$, and let T be the *mean time between faults* (MTBF). So, if λ is the fault rate, then

$$\lambda = \frac{1}{T}.$$

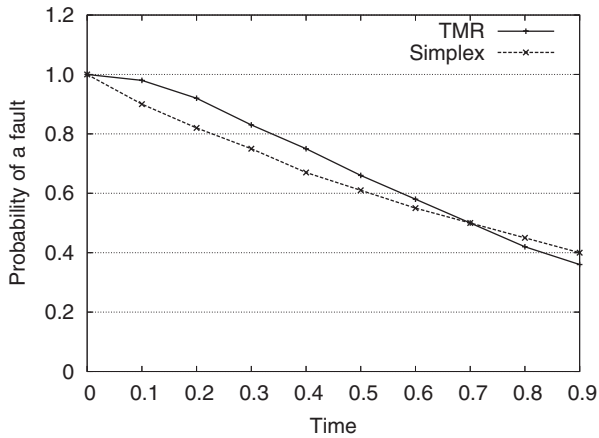


Figure 2.16 TMR reliability compared to simplex reliability.

Now imagine that faults occur on the time axis in particular time units, separated with mean, T . Using the same reasoning that we used to develop the Poisson yield equation, we can get the Poisson fault equation:

$$P(t) = e^{\frac{-t}{T}} = e^{-t\lambda}.$$

Redundancy is an obvious approach to improved reliability (lower $P(t)$). A well-known technique is *triple modular redundancy* (TMR). Three processors execute the same computation and compare results. A voting mechanism selects the output on which at least two processors agree. TMR works but only up to a point. Beyond the obvious problem of the reliability of the voting mechanism, there is a problem with the sheer amount of hardware. Clearly, as time t approaches T , we expect to have more faults in the TMR system than in a simple simplex system (Figure 2.16). Indeed, the probability of a TMR fault (any two out of three processor faults) exceeds the simplex system when

$$t = T \times \log_e 2.$$

Most fault-tolerant designs involve simpler hardware built around the following:

- *Error Detection.* The use of parity, residue, and other codes are essential to reliable system configurations.
- *Instruction (Action) Retry.* Once a fault is detected, the action can be retried to overcome transient errors.

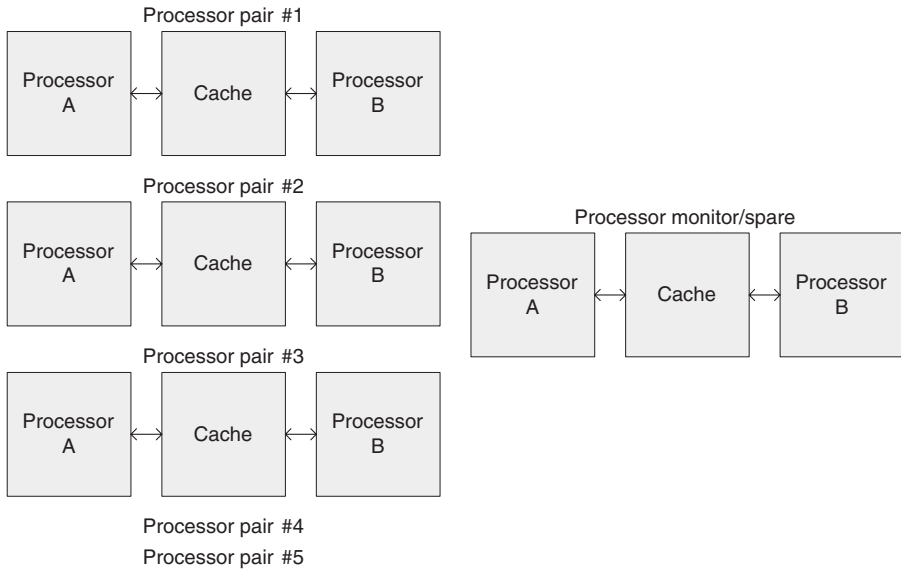


Figure 2.17 A duplex approach to fault tolerance using error detection.

- *Error Correction.* Since most of the system is storage and memory, an ECC can be effective in overcoming storage faults.
- *Reconfiguration.* Once a fault is detected, it may be possible to reconfigure parts of the system so that the failing subsystem is isolated from further computation.

Note that with error detection, efficient, reliable system configurations are limited. As a minimum, most systems should incorporate error detection on all parts of essential system components and should selectively use ECC and other techniques to improve reliability.

The IBM mainframe S/390 (Figure 2.17) is an example of a system oriented to reliability. One model provides a module of 12 processors. Five pairs in duplex configuration (5×2) run five independent tasks, and two processors are used as monitor and spare. Within a duplex, the processor pairs share a common cache and storage system. The processor pairs run the same task and compare results. The processors use error detection wherever possible. The cache and storage uses ECC, usually single error correction, double error detection (SECCDED).

Recent research addresses reliability for multiprocessor SOC technology. For instance, to improve reliability due to single-event upsets due to cosmic rays, techniques involving voltage scaling and application task mapping can be applied [214].

2.7.2 Error Detection and Correction

The simplest type of error detection is parity. A bit is added (a check bit) to every stored word or data transfer, which ensures that the sum of the number of 1's in the word is even (or odd, by predetermined convention). If a single error occurs to any bit in the word, the sum modulo two of the number of 1's in the word is inconsistent with the parity assumption, and the memory word is known to have been corrupted.

Knowing that there is an error in the retrieved word is valuable. Often, a simple reaccessing of the word may retrieve the correct contents. However, often the data in a particular storage cell have been lost and no amount of reaccessing can restore the true value of the data. Since such errors are likely to occur in a large system, most systems incorporate hardware to automatically correct single errors by making use of ECCs.

The simplest code of this type consists of a geometric block code. The message bits to be checked are arranged in a roughly square pattern, and the message is augmented by a parity bit for each row and for each column. If a row and a column indicate a flaw when the message is decoded at the receiver, the intersection is the damaged bit, which may be simply inverted for correction. If only a single row or a column or multiple rows or columns indicate a parity failure, a multiple-bit error is detected and a noncorrectable state is entered.

For 64 message bits, we need to add 17 parity bits: eight for each of the rows and columns and one additional parity bit to compute parity on the parity row and column (Figure 2.18).

It is more efficient to consider the message bits as forming a hypercube, for each message combination forms a particular point in this hypercube. If the

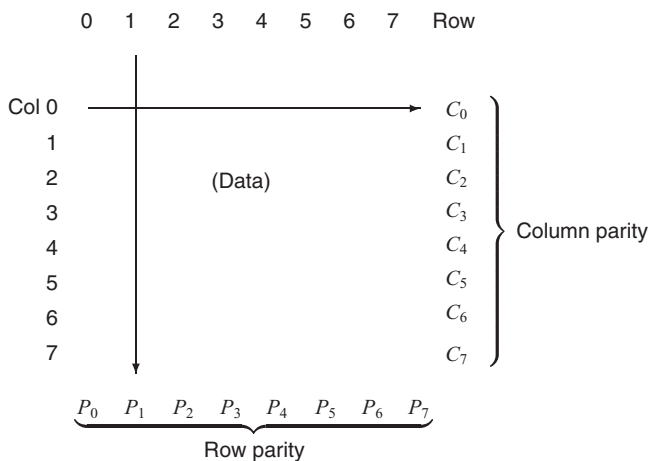


Figure 2.18 Two-dimensional error-correcting codes (ECCs).

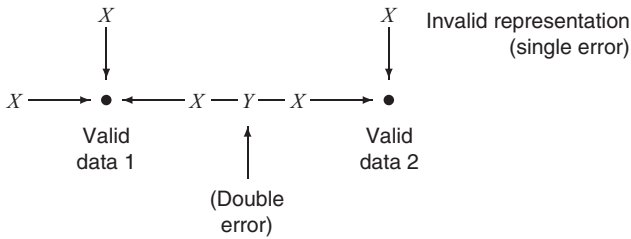


Figure 2.19 ECC code distance.

hypercube can be enlarged so that each valid data point is surrounded by associated invalid data points that are caused by a single-bit corruption in the message, the decoder will recognize that the invalid data point belongs to the valid point and will be able to restore the message to its original intended form. This can be extended one more step by adding yet another invalid point between two valid data combinations (Figure 2.19). The minimum number of bits by which valid representations may differ is the code distance. This third point indicates that two errors have occurred. Hence, either of two valid code data points is equally likely, and the message is detectably flawed but noncorrectable. For a message of 64 bits, and for single-bit error correction, each of the 2^{64} combinations must be surrounded by, or must accommodate, a failure of any of the 64 constituent bits ($2^6 = 64$). Thus, we need 2^{64+6} total code combinations to be able to identify the invalid states associated with each valid state, or a total of 2^{64+6+1} total data states. We can express this in another way:

$$2^k \geq m + k + 1,$$

where m is the number of message bits and k is the number of correction bits that must be added to support single error correction.

Hamming codes represent a realization of ECC based on hypercubes. Just as in the block code before, a pair of parity failures addresses the location of a flawed bit. The k correction bits determine the address of a flawed bit in a Hamming code. The message bits must be arranged to provide an orthogonal basis for the code (as in the case of the columns and rows of the block code). Further, the correction bits must be included in this basis. An orthogonal basis for 16 message bits is shown in Example 2.1, together with the setting of the five correction bits. Adding another bit, a sixth bit, allows us to compute parity on the entire $m + k + 1$ bit message. Now if we get an indication of a correctable error from the k correct bits, and no indication of parity failure from this new d bit, we know that there is a double error and that any attempt at correction may be incorrect and should not be attempted. These codes are commonly called SECDED.

EXAMPLE 2.1 A HAMMING CODE EXAMPLE

Suppose we have a 16-bit message, $m = 16$.

$2^k \geq 16 + k + 1$; therefore, $k = 5$.

Thus, the message has $16 + 5 = 21$ bits. The five correction bits will be defined by parity on the following groups, defined by base 2 hypercubes:

k_5	bits 16–21.
k_4	bits 8–15.
k_3	bits 4–7, 12–15, and 20–21.
k_2	bits 2–3, 6–7, 10–11, 14–15, and 18–19.
k_1	bits 1, 3, 5, 7, 9 . . . , 19, 21.

In other words, the 21-bit formatted message bits $f_1 - f_{21}$ consist of original message bits $m_1 - m_{16}$ and correction bits $k_1 - k_5$. Each correction bit is sited in a location within the group it checks.

Suppose the message consists of $f_1 - f_{21}$ and $m_1 - m_{16} = 01010101010101$. For simplicity of decoding, let us site the correction bits at locations that are covered only by the designated correction bit (e.g., only k_5 covers bit 16):

$$k_1 = f_1.$$

$$k_2 = f_2.$$

$$k_3 = f_4.$$

$$k_4 = f_8.$$

$$k_5 = f_{16}.$$

Now we have (m_1 is at f_3 , m_2 at f_5 , etc.)

$$\begin{array}{cccccccccccccccccccccccc} f_1 & f_2 & f_3 & f_4 & f_5 & f_6 & f_7 & f_8 & f_9 & f_{10} & f_{11} & f_{12} & f_{13} & f_{14} & f_{15} & f_{16} & f_{17} & f_{18} & f_{19} & f_{20} & f_{21} \\ k_1 & k_2 & 0 & k_3 & 1 & 0 & 1 & k_4 & 0 & 1 & 0 & 1 & 0 & k_5 & 1 & 0 & 1 & 0 & 1 & & & \end{array}$$

Thus, with even parity,

$$k_5 = 1.$$

$$k_4 = 1.$$

$$k_3 = 1.$$

$$k_2 = 0.$$

$$k_1 = 1.$$

Suppose this message is sent but received with $f_8 = 0$ (when it should be $f_8 = k_4 = 1$). When parity is recomputed at the receiver for each of the five correction groups, only one group covers f_8 .

In recomputing parity across the groups, we get

$$k'_5 = 0 \text{ (i.e., there is no error in bits 16–21).}$$

$$k'_4 = 1.$$

$$k'_3 = 0.$$

$$k'_2 = 0.$$

$$k'_1 = 0.$$

The failure pattern 01000 is the binary representation for the incorrect bit (bit 8), which must be changed to correct the message.

2.7.3 Dealing with Manufacturing Faults

The traditional way of dealing with manufacturing faults is through testing. As transistor density increases and the overall die transistor count increases proportionally, the problem of testing increases even faster. The testable combinations increase exponentially with transistor count. Without a testing breakthrough, it is estimated that within a few years, the cost of die testing will exceed the remaining cost of manufacturing.

Assuring the integrity of a design *a priori* is a difficult, if not impossible, task. Depending on the level at which the design is validated, various design automation tools can be helpful. When a design is complete, the logical model of the design can, in some cases, be *validated*. Design validation consists of comparing the logical output of a design with the logical assertions specifying the design. In areas such as storage (cache) or even floating-point arithmetic, it is possible to have a reasonably comprehensive validation. More generalized validation is a subject of ongoing research.

Of course, the hardware designer can help the testing and validation effort, through a process called *design for testability* [104]. Error detection hardware, where applicable, is an obvious test assist. A technique to give testing access to interior (not accessible from the instruction set) storage cells is called *scan*. A scan chain in its simplest form consists of a separate entry and exit point from each storage cell. Each of these points is MUXed (multiplexed) onto a serial bus, which can be loaded from/to storage independent of the rest of the system. Scan allows predetermined data configurations to be entered into storage, and the output of particular configurations can be compared with known correct output configurations. Scan techniques were originally developed in the 1960s as part of mainframe technology. They were largely abandoned later only to be rediscovered with the advent of high-density dice.

Scan chains require numerous test configurations to cover large design; hence, even scan is limited in its potential for design validation. Newer techniques extend scan by compressing the number of patterns required and by incorporating various *built-in self-test* features.

2.7.4 Memory and Function Scrubbing

Scrubbing is a technique that tests a unit by exercising it when it would otherwise be idle or unavailable (such as on startup). It is most often used with memory. When memory is idle, the memory cells are cycled with write and read operations. This potentially detects damaged portions of memory, which are then declared unavailable, and processes are relocated to avoid it.

In principle, the same technique can be applied to functional units (such as floating-point units). Clearly, it is most effective if there is a possibility of reconfiguring units so that system operation can continue (at reduced performance).

2.8 CONFIGURABILITY

This section covers two topics involving configurability, focusing on designs that are reconfigurable. First, we provide a number of motivations for reconfigurable designs and include a simple example illustrating the basic ideas. Second, we estimate the area cost of current reconfigurable devices based on the rbe model developed earlier in this chapter.

2.8.1 Why Reconfigurable Design?

In Chapter 1, we describe the motivation for adopting reconfigurable designs, mainly from the point of view of managing complexity based on high-performance intellectual properties (IPs) and avoiding the risks and delays associated with fabrication. In this section, we provide three more reasons for using reconfigurable devices, such as FPGAs, based on the topics introduced in the previous sections of this chapter: time, area, and reliability:

Time. Since FPGAs, particularly the fine-grained ones, contain an abundance of registers, they support highly pipelined designs. Another consideration is parallelism: Instead of running a sequential processor at a high clock rate, an FPGA-based processor at a lower clock rate can have similar or even superior performance by having customized circuits executing in parallel. In contrast, the instruction set and the pipeline structure of a microprocessor may not always fit a given application. We shall illustrate this point by a simple example later.

Area. While it is true that the programmability of FPGAs would incur area overheads, the regularity of FPGAs simplifies the adoption of more aggressive manufacturing process technologies than the ones for application-specific integrated circuits (ASICs). Hence, FPGAs tend to be able to exploit advances in process technologies more readily than other forms of circuits. Furthermore, a small FPGA can support a large design by time-division multiplex and run-time reconfiguration, enabling trade-off in execution time and the amount of resources required. In the

next section, we shall estimate the size of some FPGA designs based on the rbe model that we introduced earlier this chapter.

Reliability. The regularity and homogeneity of FPGAs enable the introduction of redundant cells and interconnections into their architecture. Various strategies have been developed to avoid manufacturing or run-time faults by means of such redundant structures. Moreover, the reconfigurability of FPGAs has been proposed as a way to improve their circuit yield and timing due to variations in the semiconductor fabrication process [212].

To illustrate the opportunity of using FPGAs for accelerating a demanding application, let us consider a simplified example comparing HDTV processing for microprocessors and for FPGAs. The resolution of HDTV is 1920×1080 pixels, or around 2 million pixels. At 30 Hz, it corresponds to 60 million pixels per second. A particular application involves 100 operations, so the amount of processing required is 6000 million operations per second.

Consider a 3-GHz microprocessor that takes, on average, five cycles to complete an operation. It can support 0.2 operation per cycle and, in aggregate, only 600 million operations per second, 10 times slower than the required processing rate.

In contrast, consider a 100-MHz FPGA design that can cover 60 operations in parallel per cycle. This design meets the required processing rate of 6000 million operations per second, 10 times more than the 3 GHz microprocessor, although its clock rate is only 1/30th of that of the microprocessor. The design can exploit reconfigurability in various ways, such as making use of instance-specific optimization to improve area, speed, or power consumption for specific execution data, or reconfiguring the design to adapt to run-time conditions. Further discussions on configurability can be found in Chapter 6.

2.8.2 Area Estimate of Reconfigurable Devices

To estimate the area of reconfigurable devices, we use the rbe, discussed earlier as the basic measure. Recall, for instance, that in practical designs, the six-transistor register cell takes about $2700\lambda^2$.

There are around 7000 transistors required for configuration, routing, and logic for a “slice” in a Xilinx FPGA, and around 12,000 transistors in a logic element (LE) of an Altera device. Empirically, each rbe contains around 10 logic transistors, so each slice contains 700rbe. A large Virtex XC2V6000 device contains 33,792 slices, or 23.65 million rbe or 16,400A.

An 8×8 multiplier in this technology would take about 35 slices, or 24,500rbe or 17A. In contrast, given that a 1-bit multiplier unit containing a full adder and an AND gate has around 60 transistors in VLSI technology, the same multiplier would have $64 \times 60 = 3840$ transistors, or around 384rbe, which is around 60 times smaller than the reconfigurable version.

Given that multipliers are used often in designs, many FPGAs now have dedicated resources for supporting multipliers. This technique frees up recon-

figurable resources to implement other functions rather than multipliers, at the expense of making the device less regular and wasting area when the design cannot use them.

2.9 CONCLUSION

Cycle time is of paramount importance in processor design. It is largely determined by technology but is significantly influenced by secondary considerations, such as clocking philosophy and pipeline segmentation.

Once cycle time has been determined, the designer's next challenge is to optimize the cost–performance of a design by making maximum use of chip area—using chip area to the best possible advantage of performance. A technology-independent measure of area called the rbe provides the basis for storage hierarchy trade-offs among a number of important architectural considerations.

While efficient use of die area can be important, the power that a chip consumes is equally (and sometime more) important. The performance–power trade-off heavily favors designs that minimize the required clock frequency, as power is a cubic function of frequency. As power enables many environmental applications, particularly those wearable or sensor based, careful optimization determines the success of a design, especially an SOC design.

Reliability is usually an assumed requirement, but the ever smaller feature sizes in the technology make designs increasingly sensitive to radiation and similar hazards.

Depending on the application, the designer must anticipate hazards and incorporate features to preserve the integrity of the computation.

The great conundrum in SOC design is how to use the advantages the technology provides within a restricted design budget. Configurability is surely one useful approach that has been emerging, especially the selected use of FPGA technology.

2.10 PROBLEM SET

1. A four-segment pipeline implements a function and has the following delays for each segment ($b = 0.2$):

Segment #	Maximum delay*
1	1.7 ns
2	1.5 ns
3	1.9 ns
4	1.4 ns

*Excludes clock overhead of 0.2 ns.

- (a) What is the cycle time that maximizes performance without allocating multiple cycles to a segment?
 - (b) What is the total time to execute the function (through all stages)?
 - (c) What is the cycle time that maximizes performance if each segment can be partitioned into sub-segments?
2. Repeat problem 1 if there is a 0.1 ns clock skew (uncertainty of ± 0.1 ns) in the arrival of each clock pulse.
 3. We can generalize the equation for S_{opt} by allowing for pipeline interruption delay of $S - a$ cycles (rather than $S - 1$), where $S > a \geq 1$. Find the new expression for S_{opt} .
 4. A certain pipeline has the following functions and functional unit delays (without clocking overhead):

Function	Delay
A	0.6
B	0.8
C	0.3
D	0.7
E	0.9
F	0.5

Function units B , D , and E can be subdivided into two equal delay stages. If the expected occurrence of pipeline breaks is $b = 0.25$ and clocking overhead is 0.1 ns:

- (a) What is the optimum number of pipeline segments (round down to integer value)?
 - (b) What cycle time does this give?
 - (c) Compute the pipeline performance with this cycle time.
5. A processor die ($1.4 \text{ cm} \times 1.4 \text{ cm}$) will be produced for five years. Over this period, defect densities are expected to drop linearly from 0.5 defects/cm² to 0.1 defects/cm². The cost of 20 cm wafer production will fall linearly from \$5,000 to \$3,000, and the cost of 30 cm wafer production will fall linearly from \$10,000 to \$6,000. Assume production of good devices is constant in each year. Which production process should be chosen?
 6. DRAM chip design is a specialized art where extensive optimizations are made to reduce cell size and data storage overhead. For a cell size of $135\lambda^2$, find the capacity of a DRAM chip. Process parameters are: yield = 80%, $\rho_D = 0.3$ defects/cm², feature size = $0.1 \mu\text{m}$, overhead consists of 10% for drivers and sense amps. Overhead for pads, drivers, guard ring, etc., is 20%. There are no buses or latches.

Since memory must be sized as an even power of 2, find the capacity and resize the die to the *actual gross area* (eliminating wasted space) and find the corresponding yield.

7. Compute the cost of a $512^M \times 1^b$ die, using the assumptions of problem 6. Assume a 30cm diameter wafer costs \$15,000.
8. Suppose a 2.3 cm^2 die can be fabricated on a 20cm wafer at a cost of \$5,000, or on a 30cm wafer at a cost of \$8,000. Compare the effective cost per die for defect densities of 0.2 defects/ cm^2 and 0.5 defects/ cm^2 .
9. Following the reasoning of the yield equation derivation, show

$$P(t) = e^{\frac{-t}{T}}$$

10. Show that, for the triple modular system the expected time, t , for 2 modules failure is

$$t = T \times \log_e 2$$

Hint: there are 3 modules, if any 2 (3 combinations) or all 3 fail, the system fails.

11. Design a Hamming code for a 32 bit message. Place the check bits in the resulting message.
12. Suppose we want to design a Hamming code for double error correct for a 64-bit message. How many correct bits are required? Explain.

7 Application Studies

7.1 INTRODUCTION

This chapter describes various applications to illustrate the opportunities and trade-offs in SOC design. It also shows how some of the techniques described in earlier chapters can be applied.

We first present an approach for developing SOC designs. Then we illustrate the proposed techniques in analyzing simple designs for the Advanced Encryption Standard (AES). Next, we have a look at 3-D computer graphics, showing the use of analysis and prototyping techniques and their application to a simplified PS2 system. After that we describe compression methods for still image and real-time video, as well as a few other applications to illustrate the variety of requirements and architectures of SOC.

Our discussion separates *requirements* from *designs* that can be shown to meet the requirements. Requirements cover what are needed, while a design includes sufficient implementation detail for it to be evaluated against the requirements.

7.2 SOC DESIGN APPROACH

Figure 7.1 shows a simplified SOC design approach based on the material from the preceding chapters. Chapter 2 introduces five big issues in SOC design: performance, die area, power consumption, reliability, and configurability. These issues provide the basis on which design specification and run-time requirements for SOC designs can be captured. An initial design can then be developed, which would show promise in meeting the key requirements. This initial design can then be systematically optimized by addressing issues related to memory (Chapter 4), interconnect (Chapter 5), processor (Chapter 3) and cache (Chapter 4), and customization and configurability (Chapter 6). This process is repeated until reaching a design that meets the specification and run-time requirements. More details will be given later in this section.

Following this approach, however, can appear to be a formidable task. System design is often more challenging than component or processor design,

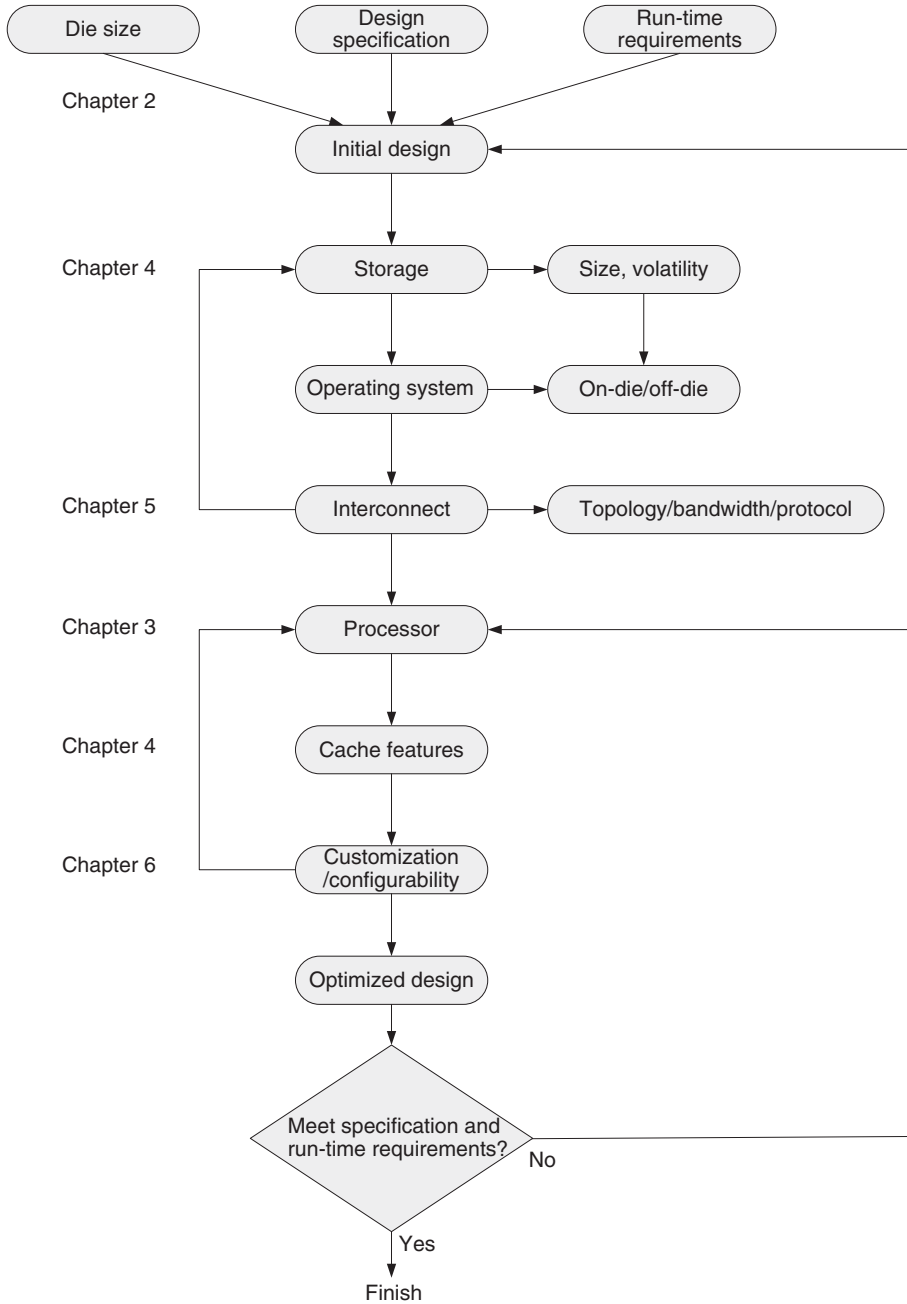


Figure 7.1 An approach for designing SOC devices.

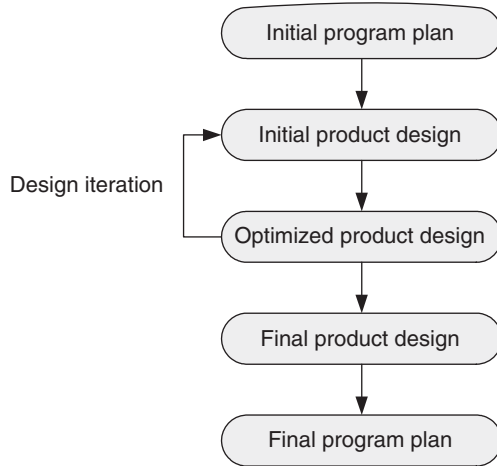


Figure 7.2 The system design process.

and it often takes many iterations through the design to ensure that (1) the design requirements are satisfied and (2) the design is close to optimal, where optimality includes considerations for overall cost (including design, manufacturing, and other costs) and performance (including market size and competition).

The usual starting point for a design is an initial project plan. As discussed in Chapter 1, this includes a budget allocation for product development, a schedule, a market estimate (and a corresponding competitive product analysis), and some measure of targeted product performance and cost. As shown in Figure 7.2, the next step is to create an initial product design. This design is merely a placeholder (or “straw man”) that has a good chance of meeting requirements in target product performance and cost. Further analysis may prove that it may or may not satisfy the requirements. An important part of this initial analysis is to develop a complete understanding of the performance and functional requirements and their inter-relationship. The various pieces of the application are specified and formally defined, and appropriate analytic and simulation models are developed. These models should provide an idea of the performance–functionality trade-off for the application and the implementation technology, which would be important in meeting run-time requirements such as those shown in Table 7.1.

With the design specification ready, we propose an initial system design (Figure 7.3). Design specifications usually anticipate the general product layout, addressing issues such as having all on one die or system on a board, operating system selection, total size of memory, and backing store. The development of the initial design then proceeds as follows, to ensure that the critical requirements are met:

REQUIREMENTS AND DESIGN

The input to the requirement understanding task is usually a requirement specification from customers or from a marketing study. The output is often a functional requirement specification for the design. The specification may be detailed and carefully written for review; this is the case in large companies. However, the specification may also be brief, captured, for example, in a spreadsheet; this is often the case in small companies and startups. The specification, whether detailed or brief, is essential for design review, documentation, and verification, and must be clear and consistent. Moreover, mathematical, executable, or diagrammatic descriptions can be used to capture data flow and control flow for the main operations or selected functions of the SOC, during various system design stages. Such descriptions can help designers to understand the functional requirements. Once the functional requirements are understood, these descriptions can be mapped to possible implementations of the main components and their interactions in the SOC.

1. Selection and allocation of memory and backing store. This generally follows the discussions in Chapter 4.
2. Once the memory has been allocated, the processor(s) are selected. Usually a simple base processor is selected to run the operating system and manage the application control functions. Time critical processes can be assigned to special processors (such as VLIW and SIMD processors discussed in Chapter 1 and Chapter 3) depending on the nature of the critical computation.
3. The layout of the memory and the processors generally defines the interconnect architecture covered in Chapter 5. Now the bandwidth requirements must be determined. Again the design specifications and processor target performance largely determine the overall requirement but cache memory can act as an important buffer element in meeting specifications.

TABLE 7.1 Run-Time Requirements Showing Various Constraints to Be Met for Some Video and Graphics Applications

Application	Real-Time Constraint (fps)	Other Constraints
Video conference	16	Frame size, error rate, missed frames
3-D graphics	30	Image size, shading, texture, color

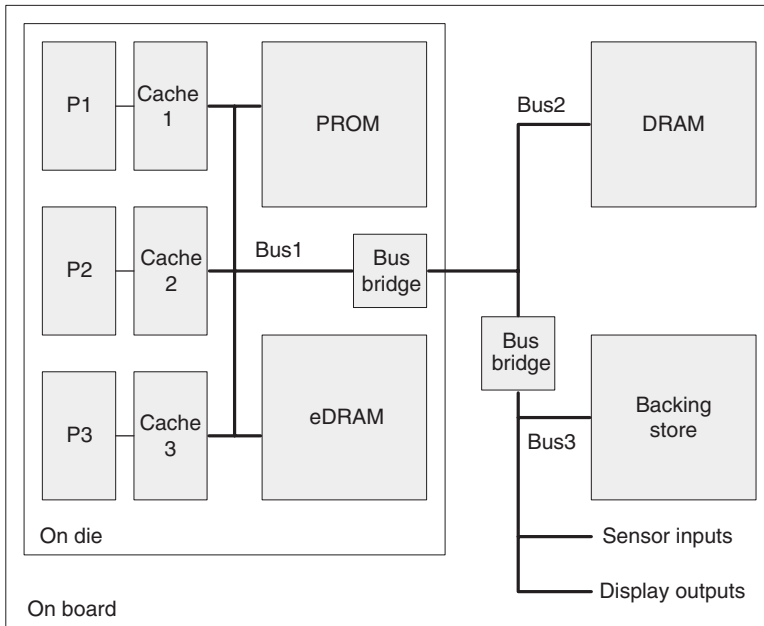


Figure 7.3 An example of an initial design, with three processors P1, P2, and P3.

Usually the initial design assumes that the interconnect bandwidth is sufficient to match the bandwidth of memory.

4. The memory elements are analyzed to assess their effects on latency and bandwidth. The caches or data buffers are sized to meet the memory and interconnect bandwidth requirements. Some details can be covered later, so for instance bus latency is usually determined without considering the effects of bus contention. Processor performance models are developed.
5. Some applications require peripheral selection and design, which must also meet bandwidth requirements. Example peripherals are shown in Section 7.5.1, which covers the JPEG system for a digital still camera, and also in Section 8.7, which covers radio frequency and optical communications for future autonomous SOCs.
6. Rough estimates of overall cost and performance are determined.

Following initial design, the design optimization and verification phase begins. This phase is supported by various tools, including profiling facilities and optimizing compilers. All components and allocations are reassessed with the view toward lowering cost (area) and improving both performance and functionality. For instance, customization and configurability techniques, such as the use of custom instructions or the adoption of run-time reconfiguration as discussed in Chapter 6, can be applied to enhance flexibility or performance. As another example, software optimizations, such as those that improve locality of refer-

ence, can often provide large performance improvement for little hardware cost. The optimization would, where applicable, involve repartitioning between hardware and software, which could affect the complexity of embedded software programs [151] and the choice of real-time operating systems [34].

After each optimization, its impact on accuracy, performance, resource usage, power and energy consumption, and so on needs to be analyzed. Also the design needs to be verified to make sure that its correctness would not be affected by the optimizations [53]. Such analysis and verification can often be supported by electronic system-level design tools (see box), as well as by prototyping.

ESL: ELECTRONIC SYSTEM LEVEL DESIGN AND VERIFICATION

There does not seem to be a standard description of what ESL covers. Wikipedia describes ESL design and verification to be “an emerging electronic design methodology that focuses on the higher abstraction level concerns first and foremost.” Another definition of ESL is the utilization of appropriate abstractions in order to increase comprehension about a system, and to enhance the probability of a successful implementation of functionality in a cost-effective manner [32]. Various ESL tools have been developed, which are capable of supporting a design flow that can generate systems across hardware and software boundaries from an algorithmic description [108].

When the design appears optimal after several iterations, another complete program plan is developed to understand any changes made to schedule fixed costs and to address issues involved in system integration and testing. Finally, the product market is assessed based on the final design and the overall program profitability can be assessed.

7.3 APPLICATION STUDY: AES

We adopt the AES as a case study to illustrate how techniques discussed in the preceding chapters can be used for exploring designs that meet specified requirements.

7.3.1 AES: Algorithm and Requirements

The AES cipher standard [69] has three block sizes: 128 (AES-128), 192 (AES-192), and 256 (AES-256) bits. The whole process from original data to

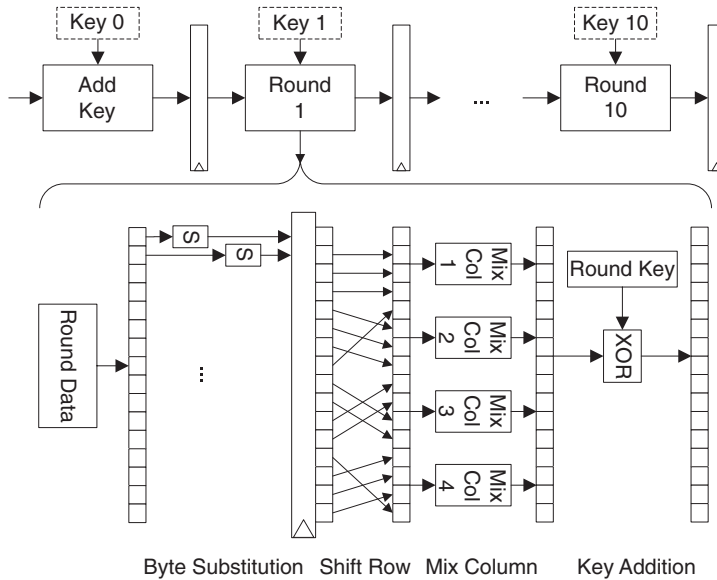


Figure 7.4 Fully pipelined AES architecture [107].

encrypted data involves one initial round, $r - 1$ standard rounds, and one final round. The major transformations involve the following steps (Figure 7.4):

- *SubBytes*. An input block is transformed byte by byte by using a special design substitution box (S-Box).
- *ShiftRows*. The bytes of the input are arranged into four rows. Each row is then rotated with a predefined step according to its row value.
- *MixColumns*. The arranged four-row structure is then transformed by using polynomial multiplication over $GF(2^8)$ per column basis.
- *AddRoundKey*. The input block is XOR-ed with the key in that round.

There is one round *AddRoundKey* operation in the initial round; the standard round consists of all four operations above; and the *MixColumns* operation is removed in the final round operation, while the other three operations remain. On the other hand, the inverse transformations are applied for decryption. The round transformation can be parallelized for fast implementation.

Besides the above four main steps, the AES standard includes three block sizes: 128 (AES-128), 192 (AES-192), and 256 (AES-256) bits. The whole block encryption is divided into different rounds. The design supporting AES-128 standard consists of 10 rounds.

Run-time requirements are shown in Table 7.2 for various applications, such as Wi-Fi and VoIP (Voice over Internet Protocol); our task is to find designs that meet one or more of these throughput requirements.

TABLE 7.2 Different Application Throughput Requirements, PAN: Personal Area Network

Application	Throughput requirement
Wi-Fi 802.11b	11 Mbps
Wi-Fi 802.11g	54 Mbps
Wi-Fi 802.11i/802.11n	500 Mbps
Metropolitan area network (MAN) 802.16a	75 Mbps
PAN 802.15 TG4 (low rate)	250 Kbps
PAN 802.15 TG3 (high rate)	55 Mbps
VoIP	64 Kbps
Cisco PIX firewall router	370 Mbps

7.3.2 AES: Design and Evaluation

Our initial design starts with a die size, design specification, and run-time requirement (Figure 7.1). We assume that the requirements specify the use of a PLCC68 (Plastic Leaded Chip carrier) package, with a die size of $24.2 \times 24.2 \text{ mm}^2$.

Our task is to select a processor that meets the area constraint while capable of performing a required function. Let us consider ARM7TDMI, a 32-bit RISC processor. Its die size is 0.59 mm^2 for a 180-nm process, and 0.18 mm^2 for a 90-nm process. Clearly, both processors can fit into the initial area requirement for the PLCC68 package. The cycle count for executing AES from the SimpleScalar tool set is 16,511, so the throughput, given an 115-MHz clock (as advertised by the vendor) with the 180-nm device, is $(115 \times 32)/16,511 = 222.9 \text{ Kbps}$; for a 236-MHz clock with the 90-nm device, the throughput is 457.4 Kbps. Hence the 180-nm ARM7 device is likely to be capable of performing only VoIP in Table 7.2, while the 90-nm ARM7 device should be able to support PAN 802.15 TG4 as well.

Let us explore optimization of this SOC chip such that we can improve the total system throughput without violating the initial area constraint. We would apply the technique used in Chapter 4 for modifying the cache size and evaluate its effect, using facilities such as the SimpleScalar tool set [30] if a software model for the application is available.

Using SimpleScalar with an AES software model, we explore the effects of doubling the block size of a 512-set L1 direct mapped instruction cache from 32 bytes to 64 bytes; the AES cycle count reduces from 16,511 to 16,094, or 2.6%. Assume that the initial area of the processor with the basic configuration without cache is 60K rbe, and the L1 instruction cache has 8K rbe. If we double the size of the cache, we get a total of 76K rbe instead of 68K. The total area increase is over 11%, which does not seem worthwhile for a 2.6% speed improvement.

The ARM7 is already a pipelined instruction processor. Other architectural styles, such as parallel pipelined datapaths, have much potential as shown in Table 7.3; these FPGA designs meet the throughput requirements for all the

TABLE 7.3 Performance and Area Trade-off on a Xilinx Virtex XCV-1000 FPGA [107]

	Basic Iterative	Inner-Round Pipelined	Fully Pipelined
Maximum clock frequency (MHz)	47	80	95
Encrypt/decrypt throughput (128 bits) (Mbps)	521	888	11,300
Area (number of slices)	1228	2398	12,600
Area (number of BRAM)	18	18	80
Slices and BRAM usages	10% and 56%	19% and 56%	103% and 250%

BRAM: block RAM. Note that a fully pipelined design requires more resources than the XCV-1000 device can support.

applications in Table 7.2, at the expense of larger area and power consumption than ASICs [146]. Another alternative, mentioned in Chapter 6, is to extend the instruction set of a processor by custom instructions [28]; in this case they would be specific to AES.

A note of caution: the above discussion is intended to illustrate what conclusions can be drawn given a set of conditions. In practice, various other factors should be taken into account, such as how representative are the numbers derived from, for instance, benchmark results or application scenarios based on the SimpleScalar tool set. In any case, such analysis should only be used for producing evaluations similar to those from back-of-envelope estimates—useful in providing a feel for promising solutions, which should then be confirmed by detailed design using appropriate design tools.

Two further considerations. First, as shown in Figure 7.4, an AES design can be fully pipelined and fitted into an FPGA device. To achieve over 21 Gbit/s throughput, the implementation exploits technology-specific architectures in FPGAs, such as block memories and block multipliers [107].

Second, AES cores are often used as part of a larger system. Figure 7.5 shows one such possibility for implementing the AES core, in the ViaLink FPGA fabric on a QuickMIPS device [116]. This device has a separate 32-bit MIPS 4Kc processor core and various memory and interface elements. Another possibility is to use AES in the implementation of designs involving secure hash methods [88].

7.4 APPLICATION STUDY: 3-D GRAPHICS PROCESSORS

This section considers 3-D graphics accelerators, similar to the Sony PS2 architecture [237]. Our study illustrates two useful techniques in deriving an initial design:

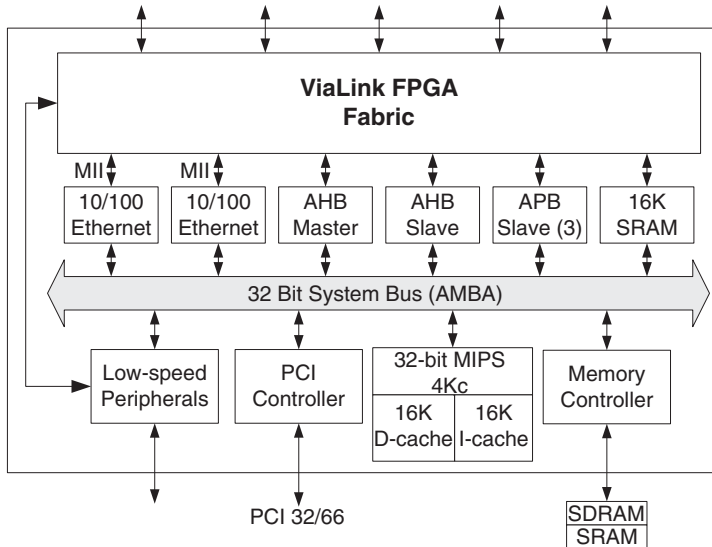


Figure 7.5 QuickMIPS block diagram for the AES SOC system [116].

- *Analysis.* The application is viewed as a high-level algorithm for back-of-envelope-type estimates about, for instance, the amount of computation and communication so that a preliminary choice of design styles and components can be made.
- *Prototyping.* A simplified version of the application is developed using common software tools and off-the-shelf hardware platforms, for example using a standard PC or a general-purpose FPGA platform. The experience gained will suggest areas that are likely to require attention, such as performance bottlenecks. It may also help identify noncritical components that do *not* need to be optimized, saving development time.

The analysis and prototyping activities can often be assisted by ESL techniques and tools [32, 108].

7.4.1 Analysis: Processing

In 3-D graphics, objects are represented by a collection of triangles in 3-D space, and there are lighting and texture effects on each picture element—or pixel—to make objects look realistic. Such 3-D representations are transformed into a 2-D space for viewing. Animation consists of providing successive frames of pixels over time.

There are two main stages in the graphics pipeline (Figure 7.6): transformation and lighting, and rasterization. We cover them in turn.

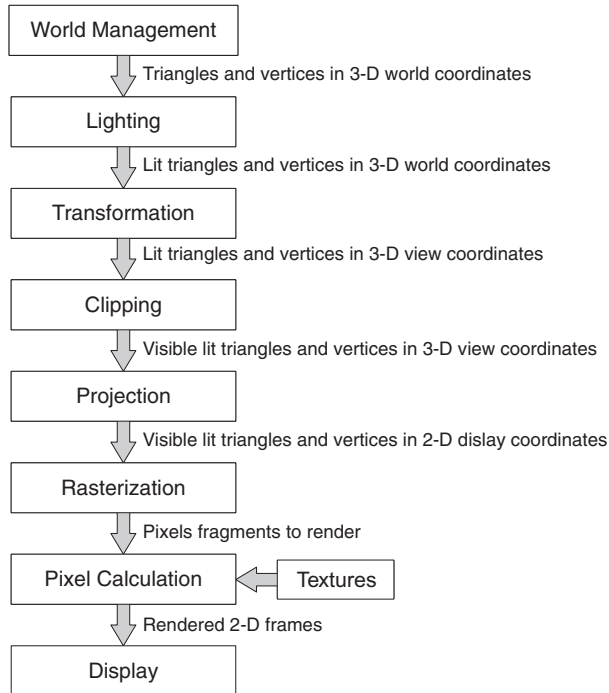
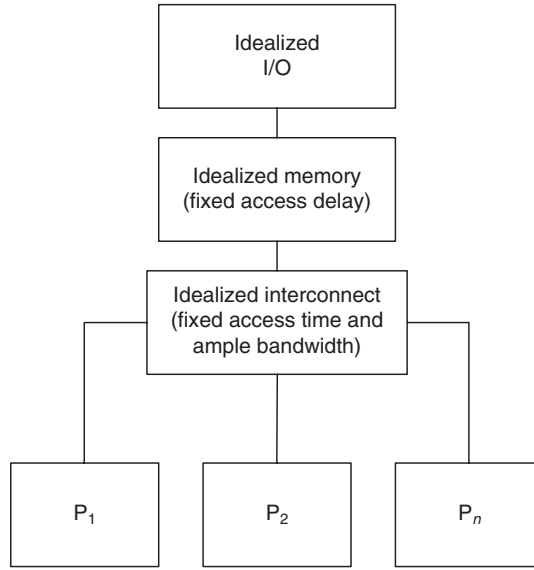


Figure 7.6 3-D graphics pipeline.

Requirements For transformation and lighting, consider v visible triangles per frame, and l light sources; the realism and complexity of the objects improve with larger v and l . To facilitate perspective projection, a system of four coordinates is used: three space dimensions and one homogeneous component capturing the location of the plane to which the 3-D image is projected.

During transformation and lighting, each triangle vertex is transformed from world space to view space, requiring a 4×4 matrix multiplication; then projected into 2-D, requiring a division and a further four multiplies for perspective correction. This can be approximated as about 24 FMACs (floating-point multiply and accumulate) per vertex, where a floating-point division (FDIV) is assumed to take the equivalent of four FMACs. The lighting process requires another 4×4 multiplication to rotate the vertex normal, followed by a dot product and some further calculations. This is approximated as 20 FMACs per light source. This results in $24 + 20l$ FMACs per vertex. In the worst case of three distinct vertices per triangle, $v(72 + 60l)$ FMACs are needed per frame; if vertices between adjacent triangles can be shared, we would only need $v(24 + 20l)$ in the best case.

Let n be the number of triangles processed per second and m be the number of FMAC per second. Given there are f frames per second (fps), $n = fv$ and:



N idealized processors selected by function

Figure 7.7 Idealized SOC architecture.

$$n \times (24 + 20l) \leq m \leq n \times (72 + 60l). \quad (7.1)$$

If $n = 50M$ ($M = 10^6$) triangles per second and no lights ($l = 0$), then $1200M \leq m \leq 3600M$; if $n = 30 \times 10^6$ triangles per second and with one light ($l = 1$), then $1320M \leq m \leq 3960M$.

Design Let us describe how we come up with a design that meets the requirements. Our initial design is based on the simple structure in Figure 7.7, introduced in Chapter 1.

The proposed design is inspired by the Emotion Engine, which contains two groups of processors as shown in Figure 7.8. The first group of processors includes:

- CPU, a MIPS-III processor with 32 registers, 128 bits, dual issue;
- Floating-point unit (FPU), supporting basic FMAC and floating-point division;
- VPU0, a vector processing unit that can operate as a slave to the MIPS or as an independent SIMD/VLIW processor.
- IPU, an image processing unit for decoding compressed video streams.

These components are connected by a 128-bit bus at 150MHz.

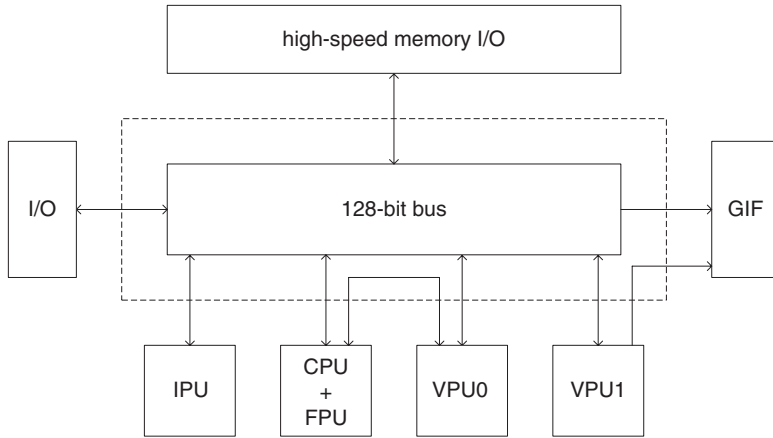


Figure 7.8 Initial design for 3-D graphics engine.

The second group of processors includes:

- VPU1, the same as VPU0, but only operates as a SIMD/VLIW processor.
- GIF, a graphics interface that mainly shunts data onto the graphics synthesizer using a dedicated 128-bit bus.

Since VPU0 and VPU1 each contains four floating-point multipliers at 300MHz, their performance is given by $300\text{MHz} \times 8 = 2400M$ FMAC/s; this value is within the range of $1200M \leq m \leq 3960M$ derived earlier.

There are other components that we do not cover in detail. An example is IPU, an image processing unit for decoding compressed video streams.

Requirements The rasterization process needs to scan convert each 2-D triangle, calculating the set of output pixels corresponding to each triangle. This is usually performed by stepping vertically along the edges of the triangles using DDA (digital differential analyzer) or another line-drawing method, allowing each horizontal span to be handled at once. Within each span, it is necessary to use more DDAs to interpolate the values of Z (for occlusion testing), color, and texture coordinates where appropriate. We ignore the vertical DDAs and approximate this as requiring $2 + 2t$ DDA steps per pixel.

Design If each DDA step requires four integer instructions, and the operation at each pixel (such as comparing updating frame and z -buffer) also requires four integer instructions, then the number of instructions for each rendered pixel is:

$$4 + 8(1 + t). \quad (7.2)$$

These steps must be performed each time a pixel is rendered, even if the pixel has already been rendered. Given there are o output pixels in each frame and each output pixel needs to be recalculated p times due to overlapping shapes, the total number of instructions required per frame is

$$(12 + 8t) \times o \times p. \quad (7.3)$$

We would use this result in the prototyping process below. Note that we have ignored the time taken to perform the vertical DDAs, so we would expect there to be some additional computation time that varies with v , but overall computation time is dominated by the per-pixel operation time.

7.4.2 Analysis: Interconnection

In our model of the 3-D graphics pipelines, there are two main intertask logical communications channels: lists of 3-D triangles passed from the creation/management task to the transformation task, and lists of 2-D triangles passed from the transformation task to the rendering task. Both these channels are essentially unidirectional: once a task has passed a set of triangles (2-D or 3-D) onto the next stage, there is no substantial flow of data needed in the other direction apart from obvious data flow signals such as status indicators.

Requirements In the first channel, between world management and transformation, all the 3-D coordinates consist of three single-precision floating point components, requiring $4 \times 3 = 12$ bytes. The minimal triangle size, where each triangle consists only of three coordinates, requires $3 \times 12 = 36$ bytes. However, in most cases there will need to be additional information, such as texture information and lighting information. In order to support lighting, it is sufficient to store the surface normal at each vertex. This applies no matter how many lights are applied, so the size of each vertex then becomes $3 \times (12 + 12 \times \min(l, 1))$. Texture information additionally requires the storage of a 2-D texture coordinate for each vertex; assuming floating-point texture coordinates, this adds 8 bytes to each vertex per applied texture. With n visible triangles, the total bandwidth on the channel is:

$$3n \times (12 + 12 \times \min(l, 1) + 8t). \quad (7.4)$$

As an example, given $n = 50 \times 10^6$, $l = 0$, $t = 1$, the bandwidth required is 3 GB/s; for $n = 30 \times 10^6$, $l = 1$, $t = 1$, the bandwidth required is 2.88 GB/s.

In the second channel, each point is now in screen coordinates, so each point can be represented as two 16-bit integers. In addition the depth of each pixel is needed, but this is stored in greater precision, so the total size is 4 bytes per vertex. Assuming that lights have been applied, the vertices also require a color intensity, requiring 1 byte per channel, or approximately 4 bytes per vertex: $4 + 4 \min(l, 1)$. Each texture coordinate must still be applied individually at

rasterization, so a 4-byte (2×16 bit integers) coordinate must be retained. This results in a total required bandwidth for the second channel as:

$$3n \times (4 + 4 \times \min(l, 1) + 4t). \quad (7.5)$$

This time, given $n = 50 \times 10^6$, $l = 0$, $t = 1$, the bandwidth required is 1.2 GB/s; for $n = 30 \times 10^6$, $l = 1$, $t = 1$, the bandwidth required is 1.08 GB/s.

Design As shown in Figure 7.8, the interconnect consists of a 128-bit bus. Since the peak transfer rate of a 128-bit bus at 150 MHz is 2.4 GB/s, which meets the bandwidth required for the second channel but not the first, an additional 64-bit dedicated interface to the rendering engine is included.

7.4.3 Prototyping

A prototype 3-D renderer is written in the C language. This prototype incorporates a world management stage that creates a random pattern of triangles, a transformation stage that projects triangles into 2-D using a single-precision floating point, and a z-buffer-based renderer using integer DDAs. Among the parameters of the renderer that can be varied are:

- number of triangles;
- size of triangles;
- width and height of output.

By adjusting these parameters, it is possible to selectively adjust parameters such as o , v , and p . For example, p can be varied by increasing the size of triangles, as this increases the chance that each triangle is covered by another.

Figure 7.9 shows the changes in execution time on an Athlon 1200 when the number of output pixels are increased. As expected, both the creation and transformation stages show no significant variation as the number of output pixels is increased. In contrast, the rendering stage's execution time is clearly increasing linearly with output pixel count. The fitted line has a correlation coefficient of 0.9895, showing a good linear fit. The fitted linear relationship between rasterization time and pixel count is given by Equation 7.3: $t = o \times 5 \times 10^{-8} + 0.0372$. The large offset of 0.0372 is caused by the per-triangle setup time mentioned (and ignored) in the analysis section. Based on this prototype we might now feel it appropriate to include the effects, as they are clearly more significant than expected.

According to Equation 7.2, the instructions per output pixel is $p \times 12$ in the case that no textures are applied ($t = 0$). In this experiment $P = 1.3$, so the instructions per frame should be 15.6. The reported performance in million instructions per second (MIPS) of the Athlon 1200 is 1400, so according to the model each extra pixel should require $15.6/1.4 \times 10^{-9} = 1.1 \times 10^{-8}$. Comparing the predicted growth of 1.1×10^{-8} with the observed growth of 5×10^{-8} , we see that they differ by a factor of 5. The majority of this error can probably be

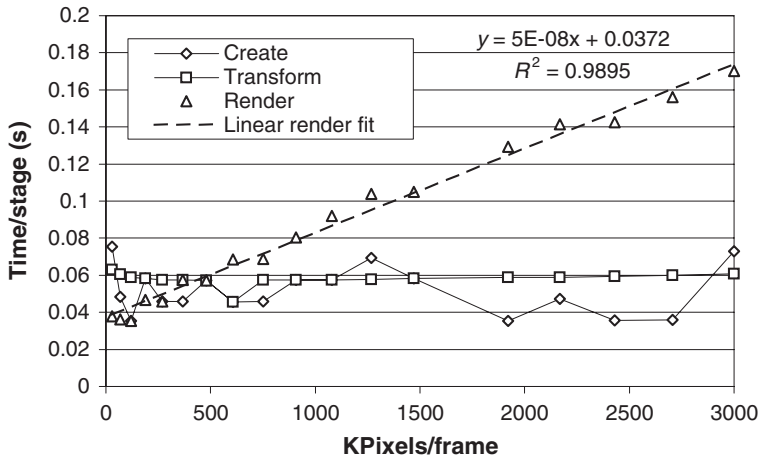


Figure 7.9 Increase in execution times of stages as pixel count increases.

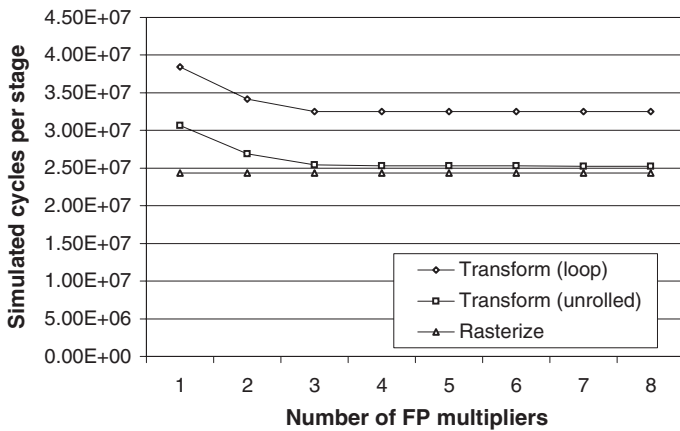


Figure 7.10 Graph of transform stage execution time for different numbers of FPU's when original version is compared to fully unrolled version. The size of the queues, issue width, and commit width are held at a constant (fairly large) value.

attributed to the unoptimized nature of the prototype and to the approximations in our models.

In Figure 7.10 and Figure 7.11 the performance of the transformation stage for different numbers of FPU's is tested using the PISA simulator [30]. When four multipliers are used the performance increases significantly, although there is little benefit when more than four are used. This suggests that a VLIW or SIMD processor that can perform four floating-point operations at once would be efficient. Also shown in the graph is the performance when the matrix multiply loop is unrolled, rather than being implemented as a doubly nested loop. The unrolling allows the processor to use the FPU's to better advantage, but the cost is still significant compared to the speedup. Finally,

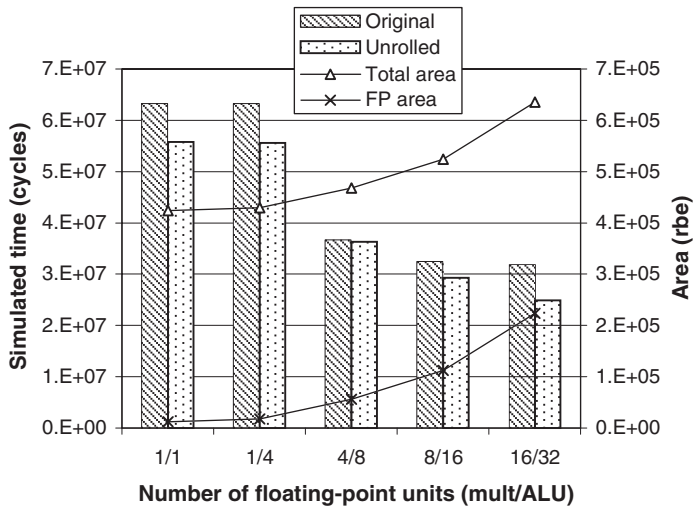


Figure 7.11 Graph of transform stage execution time for different numbers of FPUs when the original version is compared with the fully unrolled version. The size of the queues, issue width, and commit width are also scaled with the number of FPUs. The approximate area in rbes is also shown, for the whole processor and for just the FPUs.

Figure 7.12 shows that the highest performance per unit area is achieved with eight floating-point multipliers and 16 ALUs.

This section has demonstrated that application modeling can provide useful predictions of the broad computational characteristics of a 3-D engine. While the predicted times may not be accurate due to the need for estimating instruction counts, the overall trends can usually be used as a basis for further development. The only trend not predicted by our simple analysis is the growth in rasterizer time due to increasing numbers of triangles.

7.5 APPLICATION STUDY: IMAGE COMPRESSION

A number of intraframe operations are common to both still image compression methods such as JPEG, and video compression methods such as MPEG and H.264. These include color space transformation and entropy coding (EC). Video compression methods usually also include interframe operations, such as motion compensation (MC), to take advantage of the fact that successive video frames are often similar; these will be described in Section 7.6.

7.5.1 JPEG Compression

The JPEG method involves 24 bits per pixel, eight each of red, green, and blue (RGB). It can deal with both lossy and lossless compression. There are three main steps [42].

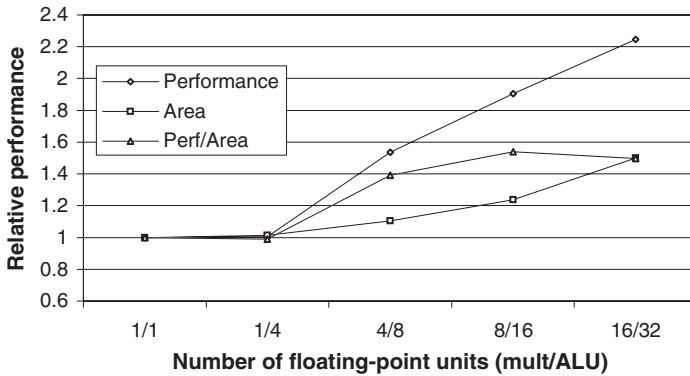


Figure 7.12 Comparison of increases in area and performance for unrolled transform stage. The maximum performance per area is achieved with eight floating-point multipliers (mult) and 16 ALUs.

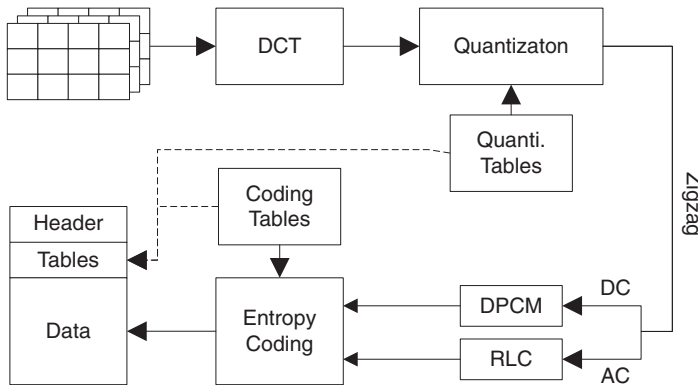


Figure 7.13 Block diagram for JPEG compression. Color space transformation is not shown.

First, color space transformation. The image is converted from RGB into a different color space such as YCbCr. The Y component represents the brightness of a pixel, while the Cb and Cr components together represent the chrominance or color. The human eye can see more detail in the Y component than in Cb and Cr, so the latter two are reduced by downsampling. The ratios at which the downsampling can be done on JPEG are 4:4:4 (no downsampling), 4:2:2 (reduce by factor of 2 in horizontal direction), and most commonly 4:2:0 (reduce by factor of 2 in horizontal and vertical directions). For the rest of the compression process, Y, Cb, and Cr are processed separately in a similar manner. These three components form the input in Figure 7.13.

Second, discrete cosine transform (the DCT block in Figure 7.13). Each component (Y, Cb, Cr) of the image is arranged into tiles of 8×8 pixels each, then each tile is converted to frequency space using a two-dimensional forward

DCT (DCT, type II) by multiplication with an 8×8 matrix. Since much information is covered by the low-frequency pixels, one could apply quantization—another matrix operation—to reduce the high-frequency components.

Third, EC. EC is a special form of lossless data compression. It involves arranging the image components in a “zigzag” order accessing low-frequency components first, employing run-length coding (RLC) algorithm to group similar frequencies together in the AC component and differential pulse code modulation (DPCM) on the DC component, and then using Huffman coding or arithmetic coding on what is left. Although arithmetic coding tends to produce better results, the decoding process is more complex.

As an example for estimating the amount of operations, consider a 2-D DCT involving $k \times k$ blocks. We need to compute:

$$y_i = \sum_{0 \leq j \leq k} c_{i,j} \times j$$

for $0 < i \leq k$, input x , DCT coefficients c , and output y . We would need k image data loads, k coefficient data loads, k multiply accumulations, and 1 data store. So in total there are $3k + 1$ operations/pixel. Hence each $k \times k$ block DCT with row-column decomposition has $2k^2(3k + 1)$ operations.

For frames of $n \times n$ resolution at f fps, the number of operations is $2fn(3k + 1)$. Two common formats are CIF (Common Intermediate Format) and QCIF (Quarter CIF), which correspond, respectively, to a resolution of 352×288 pixels and 176×144 pixels.

For a YCbCr QCIF frame with 4:2:0 sampling ratio, which has 594 tiles of 8×8 blocks, at 15 fps the total number of operations is: $2 \times 15 \times 594 \times 8 \times 8 \times (24 + 1) = 28.5$ million operations per second (MOPS). For a CIF frame, 114 MOPS are required.

Typically, lossless compression can achieve up to three times reduction in size, while lossy compression can achieve up to 25 times reduction.

7.5.2 Example JPEG System for Digital Still Camera

A typical imaging pipeline for a still image camera is shown in Figure 7.14 [128]. The TMS320C549 processor, receiving 16×16 blocks of pixels from SDRAM, implements this imaging pipeline.

Since the TMS320C549 has 32K of 16-bit RAM and 16K of 16-bit ROM, all imaging pipeline operations can be executed on chip since only a small 16×16 block of the image is used. In this way, the processing time is kept short, because there is no need for slow external memory.

This device offers performance up to 100MIPS, with low power consumption in the region of 0.45mA/MIPS. Table 7.4 illustrates a detailed cycle count for the different stages of the imaging pipeline software. The entire imaging pipeline, including JPEG, takes about 150 cycles/pixel, or about 150 instructions/pixel given a device of 100MIPS at 100MHz.

A TMS320C54x processor at 100MHz can process 1 megapixel CCD (charge coupled devices) image in 1.5 second. This processor supports a

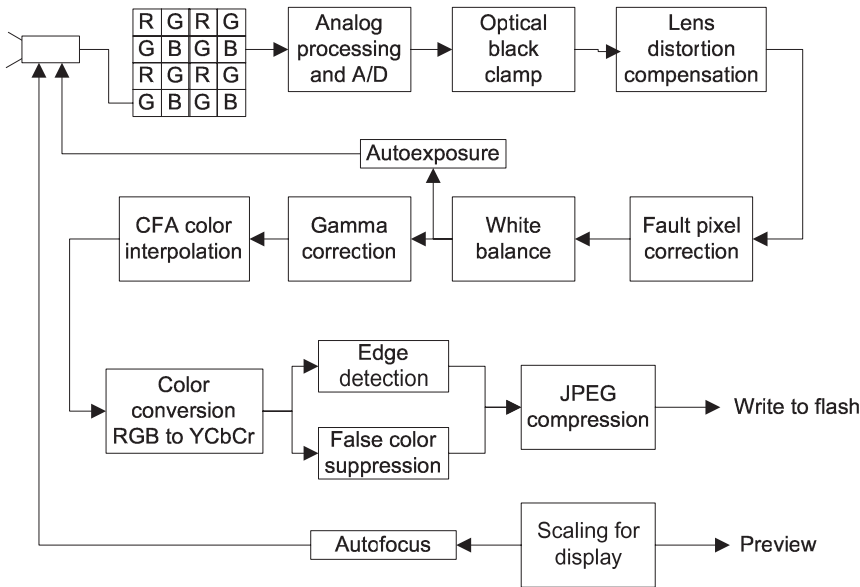


Figure 7.14 Block diagram for a still image camera [128]. A/D: analog to digital conversion; CFA: color filter array.

TABLE 7.4 TMS320C54X Performance [128]

Task	Cycles/Pixel
Preprocessing: for example, gain, white balancing	22
Color space conversion	10
Interpolation	41
Edge enhancement, false color suppression	27
4:1:1 decimation, JPEG encoding	62
Total	152

2 second shot-to-shot delay, including data movement from external memory to on-chip memory. Digital cameras should also allow users to display the captured images on the LCD screen on the camera, or on an external TV monitor. Since the captured images are stored on a flash memory card, playback-mode software is also needed on this SOC.

If the images are stored as JPEG bitstreams, the playback-mode software would decode them, scale the decoded images to appropriate spatial resolutions, and display them on the LCD screen and/or the external TV monitor. The TMS320C54x playback-mode software can execute 100cycles/pixel to support a 1 second playback of a megapixel image.

This processor requires 1.7KB for program memory and 4.6KB for data memory to support the imaging pipeline and compress the image according to the JPEG standard. The complete imaging pipeline software is stored on-chip, which reduces external memory accesses and allows the use of slower

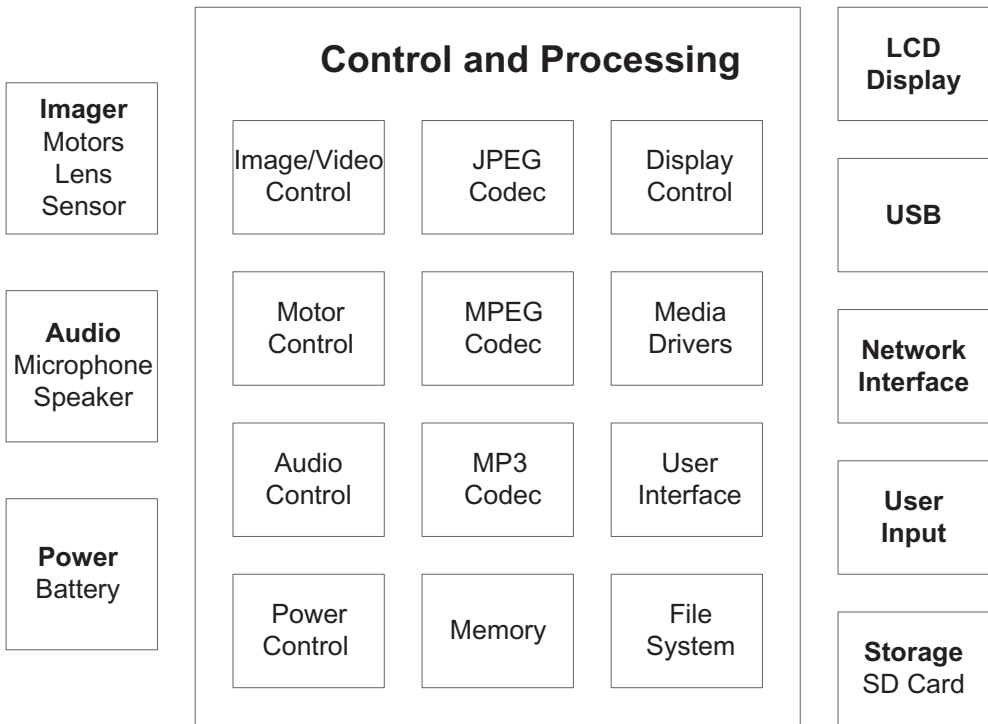


Figure 7.15 Block diagram for a camera chip with video, audio, and networking capabilities.

external memory. This organization not just improves performance, but it also lowers the system cost and enhances power efficiency.

More recent chips for use in digital cameras would need to support, in addition to image compression, also video compression, audio processing, and wireless communication [217]. Figure 7.15 shows some of the key elements in such a chip.

7.6 APPLICATION STUDY: VIDEO COMPRESSION

Table 7.5 summarizes the common video formats used in various applications, together with the associated compression methods such as MPEG1, MPEG2, and MPEG4. Video quality depends on the bitrate and the video resolution—higher bitrate and higher resolution generally mean better video quality, but requiring higher bandwidth.

There is another set of compression methods known as H.261, H.262, H.363, and H.264; some of these are related to the MPEG methods, for instance MPEG2 and H.262 are the same and H.264 corresponds to MPEG4/Part 10. Generally the more recent methods such as H.264 offer higher quality and

TABLE 7.5 Some Common Video Formats

Format	VCD	SVCD	DVD	HDDVD HDTV (WMVHD)	AVI DivX XviD WMV	MOV Quick-Time
Resolution	352 × 240	480 × 480	720 × 480*	1920 × 1080*	640 × 480*	640 × 480*
NTSC/PAL	352 × 288	480 × 576	720 × 576*	1280 × 720*		
Video compression	MPEG1	MPEG2	MPEG2, MPEG1	MPEG2 (WMV-MPEG4)	MPEG4	MPEG4 (from Sorenson Media)
Video bitrate	1150 Kbps	2000 Kbps†	5000 Kbps†	20 Mbps† (8 Mbps†)	1000 Kbps†	1000 Kbps†
Size/min	10 MB/min	10–20 MB/min	30–70 MB/min	150 MB/min† (60 MB/min†)	4–10 MB/min	4–20 MB/min

From <http://www.videohelp.com/svcd>.

*Approximate resolution, can be higher or lower.

†Approximate bitrate, can be higher or lower.

Kbps, thousand bits per second; Mbps, million bits per second; min, minutes.

higher compression ratio. In the following we shall provide an overview of some of these video compression methods [42, 270], without going into the details.

7.6.1 MPEG and H.26X Video Compression: Requirements

In addition to intraframe compression methods for still images described earlier, video compression methods also deploy interframe compression methods such as motion estimation. Motion estimation is one of the most demanding operations in standard-based video coding, as shown in the requirement below for H.261 involving CIF images with 352×288 pixels at 30 fps:

968MOPS for compression:

RGB to YCbCr	27
Motion estimation	608 (25 searches in 16×16 region)
Inter/intraframe coding	40
Loop filtering	55
Pixel prediction	18
2-D DCT	60
Quant., zigzag scanning	44
EC	17
Frame reconstruct	99

Most compression standards involve asymmetric computations, such that decompression is usually much less demanding than compression. For instance, for H.261, the amount of operations for decompression (around 200MOPS) is around 20% of that for compression (about 1000MOPS):

198MOPS for decompression:

Entropy decode	17
Inverse quantization	9
Inverse DCT (IDCT)	60
Loop filter	55
Prediction	30
YCbCr to RGB	27

The motion estimation method involves three kinds of frames (Figure 7.16). First, the intrapicture I, which does not include motion information, so it is like lossy JPEG. Second, the picture P, which covers motion prediction based on earlier I frames; it contains motion vectors (MVs) and error terms. Since error terms are small, quantizing gives good compression. Third, the bidirectional picture B, which supports motion prediction based on past and future I or P frames.

The goal of the motion estimator is to describe the information in the current frame based on information in its reference frame. The reference frame is typically the reconstructed version of the immediately preceding frame in the sequence, which is known to both the encoder and decoder.

- I frame is encoded as a still image and doesn't depend on any reference frame



- P frame depends on previously displayed reference frame



- B frame depends on previous and future reference frames

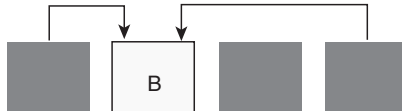


Figure 7.16 Three kinds of frames in MPEG motion estimation.

In the motion estimator for video compression methods such as H.264, pixel values are examined between these pairs of frames. Based on the operations of the motion estimator, the pixel values in the current frame can be alternately represented by a combination of two quantities: pixel values from a predictor based on the reference frame plus a prediction error that represents the difference between the predictor values and the actual pixel values in the current frame.

The function of the motion estimator can be interpreted in the following way. Suppose that the image of a moving object is represented by a group of pixels in the current frame of the original video sequence, as well as by a group of pixels in the previous frame of the reconstructed sequence. The reconstructed sequence is the designated reference frame. To achieve compression, the pixel representation of the object in the current frame is deduced from the pixel values in the reference frame. The pixel values representing the object in the reference frame is called the predictor, because it predicts the object's pixel values in the current frame. Some changes are usually needed in the predictor to attain the true pixel values of the object in the current frame; these differences are known as the prediction error.

In block-based motion estimation, the boundaries of objects represented in the current frame are assumed to approximately align along the boundaries of macroblocks. Based on this assumption, objects depicted in the frame can be represented well by one or more macroblocks. All pixels within a macroblock share the same motion characteristics. These motion characteristics are

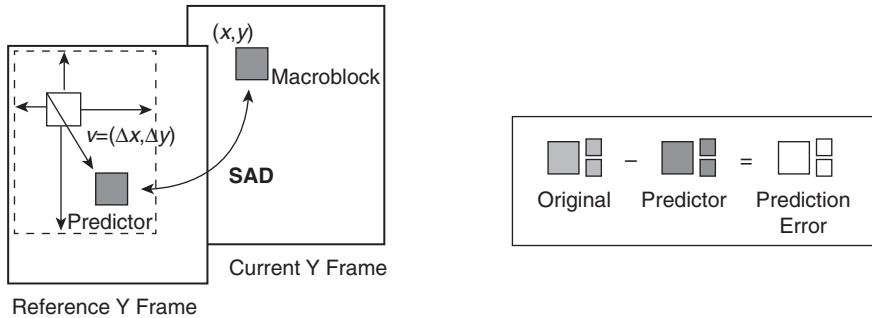


Figure 7.17 Motion estimation process.

described by the macroblock's MV, which is a vector that points from the pixel location of the macroblock in the current frame to the pixel location of its predictor. Video coding standards do not dictate how the MV and prediction error are obtained.

Figure 7.17 illustrates the process of motion estimation. Typically, the motion estimator examines only the luminance portion of the frame, so each macroblock corresponds to 16×16 luminance pixels. The current frame is subdivided into nonoverlapping macroblocks. To each macroblock, the motion estimator assigns a predictor, which must necessarily also be a square region of the size of 16×16 luminance pixels. The predictor can also be considered a macroblock. It is chosen based on the “similarity” between it and the macroblock in the current frame.

The similarity metric for the macroblocks is not specified by video coding standards such as H.264; a commonly used one is the SAD (sum of the absolute difference) metric, which computes the SAD between the values of corresponding luminance pixels in two macroblocks.

All macroblocks in a search region of the reference frame are evaluated against the current macroblock using the SAD metric. A larger SAD value indicates a greater difference between two macroblocks. The predictor macroblock is chosen to be the one which has the lowest SAD.

Many video compression standards require the search region to be rectangular and located about the coordinates of the original macroblock. The dimensions of the rectangle are adjustable but may not exceed a standard-specified maximum value.

The search strategy used to find the best-matching predictor macroblock, called the motion search, is usually not specified by the standard. Many motion search algorithms have been proposed. One possible motion search is an exhaustive (or full) search over all possible macroblocks in the search region; this strategy guarantees a global SAD minimum within the search region. However, exhaustive search is computationally expensive and is therefore primarily adopted by hardware designers, due to its regularity.

Figure 7.18 summarizes bandwidth and storage requirements of different compression methods for 90 minutes of DVD-quality video.

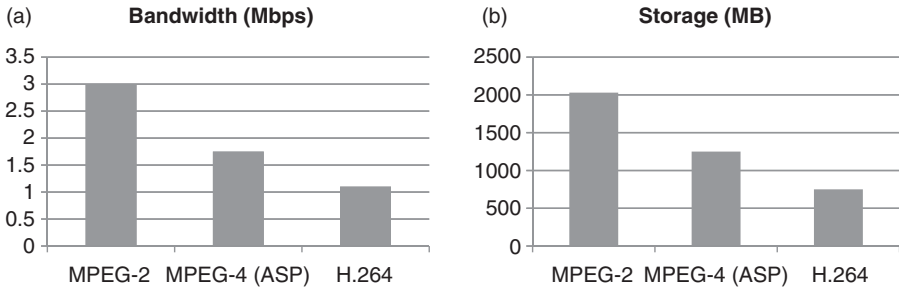


Figure 7.18 Bandwidth (a) and storage (b) of different video compression methods. ASP stands for Active Simple Profile, a version of MPEG-4.

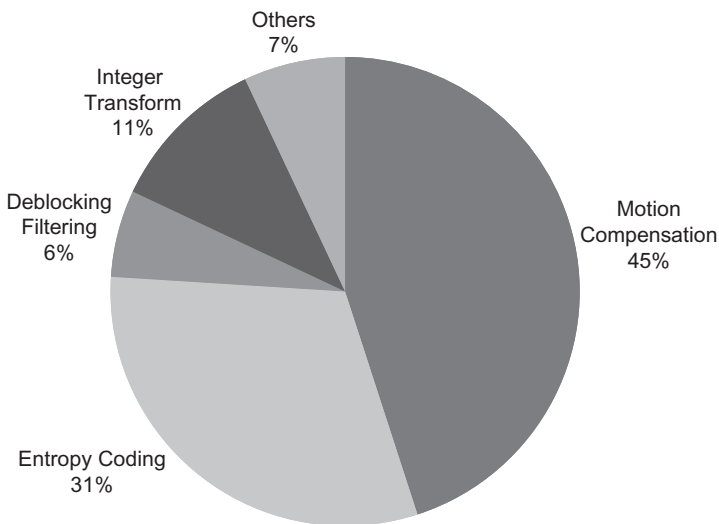


Figure 7.19 Comparing kernels in H.264/AVC decompression.

If we look at the percentage execution time of the operations in a compression or a decompression algorithm, it can be seen that motion compensation and motion estimation usually take the lion's share. For instance, H.264/AVC (Advanced Video Coding) decompression contains four major kernels: motion compensation, integer transform, entropy coding, and deblocking filtering. Motion compensation is the most time-consuming module; see Figure 7.19.

Similar results have been reported for compression as well. For instance, Figure 7.20 shows that motion estimation can take over 95% of execution time in a software H.263 encoder [270].

7.6.2 H.264 Acceleration: Designs

One of the most common digital video formats is H.264. It is an international standard that has been adopted by many broadcasting and mobile

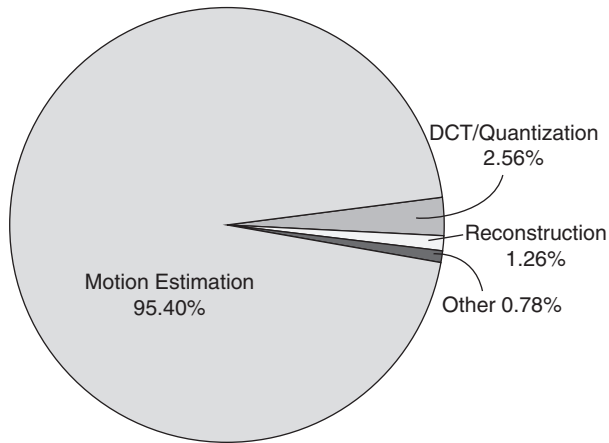


Figure 7.20 Comparing kernels in H.263 compression with a basic encoder using half-pel exhaustive motion search [270].

communication standards, such as DVB and 3G. Its coding efficiency has enabled new applications for streaming video.

Advanced algorithms are included in H.264 for motion estimation and for other methods for video compression. For motion estimation, blocks from 16×16 to 4×4 pixels are supported. Residual data transforms are applied to 4×4 blocks with modified integer DCT to prevent rounding errors.

H.264 provides better results than previous standards, due to the adoption of wider search ranges, multiple reference frames, and smaller macroblocks for motion estimation and motion compensation—at the expense of increased load in computation. It requires, for example, high-speed memory and highly pipelined designs, to meet the demand of various encoding tasks, such as those in motion estimation.

Let us consider two approaches. The first approach is to implement the tasks in programmable or dedicated hardware. For instance, a recent design of a baseline H.264/AVC encoder core [158] has been implemented in various hardware technologies, including:

1. 4CIF (704×576) at 30fps with low-cost FPGAs: Xilinx Spartan-3 and Altera Cyclone-II,
2. 720 pixels (1280×720) at 30fps with high-end FPGAs: Xilinx Virtex-4 and Altera Stratix-II,
3. 1080 pixels (1920×1080) at 30fps with $0.13\text{-}\mu\text{m}$ ASIC.

The features of these implementations are summarized in Table 7.6.

The second approach is to implement the tasks on a software configurable processor introduced in Section 6.5.3, which has an instruction set extension fabric (ISEF) to support demanding operations implemented as custom instructions (also called extension instructions). The one used here is a 300-MHz

TABLE 7.6 FPGA and ASIC Designs for H.264/AVC Encoding [158]

Technology	Approximate Area	Speed (MHz)	Video Throughput
0.13 μm LV 0.9V, 125 C	178K gates + 106 Kbits RAM, optimized for speed	~250	1920 $\times$ 1080 (1080p) at 30 fps
0.18 μm slow process	129K gates + 106 Kbits RAM, optimized for area	~50	4 CIF (704 $\times$ 576) at 30 fps
StratixII C3	17,511 ALUTs + 5 M512 + 51 M4K + 3 DSPs	~118	1280 $\times$ 720 (720p) at 32 fps
CycloneII C6	18,510 M4K + 5 M512 + 51 M4K + 3 DSPs	~65	4 CIF (704 $\times$ 576) at 40 fps
Virtex4-12	10,500 slices + 3 multipliers + 33 RAM blocks	~110	1280 $\times$ 720 (720p) at 30 fps
Spartan3-4	10,500 slices + 3 multipliers + 33 RAM blocks	~50	4 CIF (704 $\times$ 576) at 30 fps

ALUT, Adaptive Lookup Table; M4K, a configurable memory block with a total of 4,608 bits.

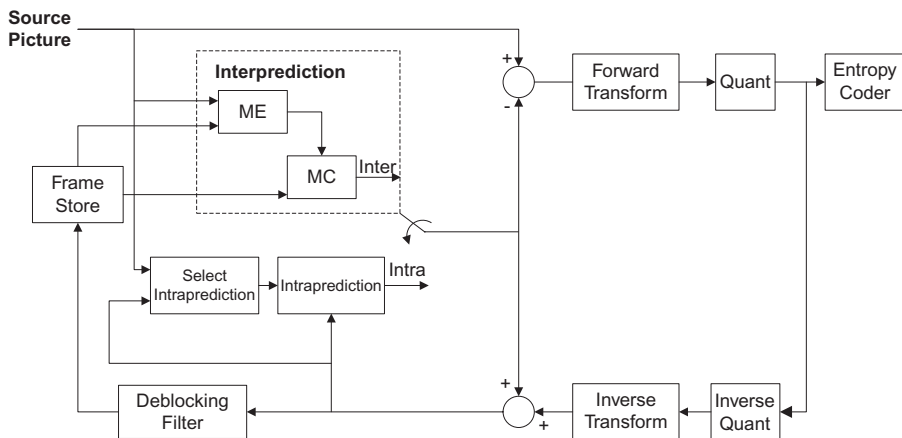


Figure 7.21 H.264 encoder architecture [152]. ME, motion estimation; MC, motion compensation; Quant, quantization.

Stretch S5 processor, which achieves H.264 encoding of Standard Definition (SD) video at 30 fps [152]. We shall look at this approach in more detail below.

Successful real-time video encoding applications have to deliver the best image quality feasible for a particular screen resolution, given real-world operating constraints. For example, an uncompressed video stream has an image size of 720 $\times$ 480 pixels and requires 1.5 bytes for color per pixel. Such a stream has 518 KB per frame, and at 30 fps it consumes 15.5 MB per second storage and bandwidth.

Figure 7.21 shows the architecture of an H.264 encoder. The efficiency of the encoder can be found in the implementation of the following functions:

1. forward DCT and IDCT,
2. intraprediction utilizing forward and inverse quantization,
3. deblocking filtering, and
4. motion estimation employing interframe comparisons.

The above are prime candidates for hardware acceleration because of the amount of computation required. Additional acceleration can be achieved by taking advantage of the inherent parallelism in these algorithms.

DCT Consider the processing of 4×4 blocks of luma pixels through a 2-D DCT and quantization step. The computations for the DCT portion of the matrix computation can be reduced to 64 add and subtract operations by taking advantage of symmetry and common subexpressions. All 64 operations can be combined into a single custom instruction for the ISEF.

Quantization (Q) This follows the DCT. The division operation, which is costly, is avoided by implementing quantization as a simple multiply and shift operation. Total processing required for luma encode and decode using DCT + Q + IDCT + IQ involves about 594 additions, 16 multiplications, and 288 decisions (using multiplexers).

Deblocking Filtering The 128-bit bus to the ISEF takes a single cycle to load a row of eight 16-bit prediction data. So one ISEF instruction can replace many conventional instructions, provided that the compiler can recognize the inherent parallelism in the function. The total number of cycles to perform these operations on a 4×4 block using a standard processor is over 1000 cycles. The same processing can be done in the software configurable processor in 105 cycles, offering more than 10 times acceleration. Hence a video stream of 720×480 pixels at 30fps would only require 14.2% utilization of the RISC processor, since the bulk of the task is off-loaded to the ISEF. Increasing sub-block sizes enhances parallelism: for example, operating on two 4×4 blocks in parallel reduces execution time in half, dropping the utilization of the RISC processor to 7.1% as the ISEF takes on a heavier load.

Accelerating deblocking requires developers to minimize conditional code. Instead of determining which values to calculate, it is often more efficient to create a single custom instruction that calculates all the results in hardware and then select the appropriate result.

Reordering the 128-bit result from the IDCT stage simplifies packing of 16 8-bit edge data pixels into a single 128-bit wide datum to feed the deblocking custom instruction. Precalculating macroblock parameters is another optimization option supported by the state registers inside the ISEF and the instruction.

The filter's inner loop loads the 128-bit register and executes the deblock-Filter() custom instruction, computing two edges per instruction. Because the same custom instruction can be used for both horizontal and vertical filtering, there is zero overhead.

This inner loop takes three cycles and is executed twice (horizontal and vertical), with about 20 cycles for loop overhead. With 64 edges in each MB of data, there are approximately 416 ($64/4 \times 26$) cycles required per MB. For a video stream of resolution 720×480 pixels at 30 fps, this results in 16.8 Mcycles/s, or approximately 5.2% processor utilization.

Motion Estimation This is known to consume much of the processor budget (50–60%). The key computation requirements are the repeated SAD calculations used in determining the best MV match.

The data calculations and comparisons are repetitive, with many of the intermediate results needing to be reused. These large data sets do not fit well within the limited register space of the traditional processor and digital signal processor (DSP) architectures. Also, these processors and DSP implementations struggle to feed the fixed arithmetic and multiplier units from the data cache.

With the Stretch S5 Software Configurable Processor, the ISEF custom processing unit is capable of performing computations in parallel and holding the intermediate results in the state registers while executing fully pipelined SAD instructions.

Motion estimation consists of potentially 41 SAD and 41 MVs calculations per macroblock. A full motion search on a single macroblock requires 262K operations for a video stream at 30fps, for a total of 10.6 giga operations per second (GOPS).

By using heuristic algorithms for many implementations, the application developer can minimize the computations to meet target image quality and/or bitrate requirements.

Custom algorithms optimized to perform estimates across different search areas, numbers of frames, or the number of MVs needing to be calculated can easily be converted to ISEF instructions. A single custom instruction can replace multiple computations, as well as pipeline many of the computations using intermediate results.

For example, a single custom instruction can perform 64 SAD calculations. The ISEF maintains the 64 partial sums to reduce the number of data transfers and to reuse the results in the next instruction. The ISEF instructions can be pipelined to improve compute capacity.

Motion estimation also involves various pixel predictions that require nine SADs computed in nine directions around the pixel. By using custom instructions, a 16×16 SAD calculation with quarter pixel precision takes 133 cycles, and a 4×4 SAD calculation with quarter pixel precision takes 50 cycles.

The above discussion covers the Stretch S5 processor. The Stretch S6 processor comes with a programmable accelerator that supports a dedicated hardware block for motion estimation, so there is no need to implement this operation in the ISEF for the S6 processor.

7.9 PROBLEM SET

1. How fast would a 32-bit processor with the ARM7 instruction set need to run to be able to support AES for Wi-Fi 802.11b? How about a 64-bit processor?
2. Estimate the number of operations per second involved in computing the DCT for high-resolution images of 1920×1080 pixels at 30 fps.
3. Explain how the JPEG system for the camera in Figure 7.14 can be revised to support 10 megapixel images.
4. Estimate the size in number of rbes of the FPGA and ASIC designs in Table 7.6, assuming that the FPGAs are produced in a 90 nm process.
5. Compare the pros and cons of the FPGA and ASIC designs in Table 7.6, assuming that the FPGAs are produced in a 90 nm process. How would your answer change when both the FPGAs and the ASIC are produced in a 45 nm process?
6. Consider a 3-D graphics application designed to deal with k nonclipped triangles, each covering an average of p pixels and a fraction α of which being obscured by other triangles. Ambient and diffuse illumination models and Gouraud shading are used. The display has a resolution of $m \times n$ pixels, updated at f fps. Estimate:
 - (a) the number of floating-point operations for geometry operations,
 - (b) the number of integer operations for computing pixel values, and
 - (c) the number of memory access for rasterization.
7. Table 7.10 shows data for the ARM1136J-S PXP system. The datapath runs at a maximum of 350 MHz for the 16K instruction cache plus 16K data

TABLE 7.10 MPEG4 Decode Performance on ARM1136J-S PXP System

L1 cache size	16K + 16K	32K + 32K	64K + 64K	16K + 16K	16K + 16K	16K + 16K	32K + 32K	32K + 32K
L2 cache size	—	—	—	128K	256K	512K	256K	512K
Speed (MHz)	350	324	277	350	350	324	324	324
Area (mm ²)	2.3	3.3	6	8.3	12.3	21	13.3	22
Run time (ms)	122.6	96.7	93.8	70.6	60.7	60.0	63.6	63.2

Area values include that of L2 where appropriate.

```

for  $y = 0$  to  $H_2 - H_1$  do
  for  $x = 0$  to  $W_2 - W_1$  do
     $I_{c,x,y} = f_3($ 
       $\sum_{i=0}^{H_1-1} \sum_{j=0}^{W_1-1} f_{12}(M1_{i,j}, I1_{i,j}, M2_{y+i,x+j}, I2_{y+i,x+j}),$ 
       $\sum_{i=0}^{H_1-1} \sum_{j=0}^{W_1-1} f_{11}(M1_{i,j}, I1_{i,j}),$ 
       $\sum_{i=0}^{H_1-1} \sum_{j=0}^{W_1-1} f_{22}(M1_{y+i,x+j}, I2_{y+i,x+j}))$ 
    end for
  end for

```

Figure 7.24 Convolution algorithm with resulting image I_c .

cache; the 32K + 32K and 64K + 64K implementations are limited in speed by their cache implementations.

- (a) Comment on the effect of increasing the size of L1 on performance.
 - (b) Suggest reasons to explain why the design with the largest cache area is not the fastest.
 - (c) Compare the fastest design and the one just behind that one. Which one is more cost effective and why?
8. Figure 7.24 shows a convolution algorithm between an $H_1 \times W_1$ image (I_1) and an $H_2 \times W_2$ image (I_2), which have optional masks M_1 and M_2 , respectively, of the same width and height. Given that $H_2 > H_1$ and $W_2 > W_1$, and f_3, f_{12}, f_{11} , and f_{22} are pure functions—that is, they have no internal state and their results depend only on their parameters:
- (a) What are the values of M_1, M_2 and f_3, f_{12}, f_{11} , and f_{22} for (1) SAD correlation, (2) normalized correlation, and (3) Gaussian blur?
 - (b) What is the resolution of the result image I_c ?
 - (c) How many cycles are needed to produce the resulting image I_c ?