Chapter 2

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

2.1 Introduction

The tradeoff 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 roadmap, 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 [5]. 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 (nm); older generations are referred to in terms of microns. So the previous generations are 65 nm, 90 nm, 0.13 microns and 0.18 microns.

<table>
<thead>
<tr>
<th>Year</th>
<th>2010</th>
<th>2013</th>
<th>2016</th>
</tr>
</thead>
<tbody>
<tr>
<td>Technology generation (nm)</td>
<td>45</td>
<td>32</td>
<td>22</td>
</tr>
<tr>
<td>Wafer size, diameter (cm)</td>
<td>30</td>
<td>45</td>
<td>45</td>
</tr>
<tr>
<td>Defect density (per cm²)</td>
<td>0.14</td>
<td>0.14</td>
<td>0.14</td>
</tr>
<tr>
<td>µP die size (cm²)</td>
<td>1.9</td>
<td>2.6</td>
<td>2.6</td>
</tr>
<tr>
<td>Chip Frequency (GHz)</td>
<td>5.9</td>
<td>7.3</td>
<td>9.2</td>
</tr>
<tr>
<td>Million Transistors per cm²</td>
<td>1203</td>
<td>3403</td>
<td>6806</td>
</tr>
<tr>
<td>Max Power (W) High Performance</td>
<td>146</td>
<td>149</td>
<td>130</td>
</tr>
</tbody>
</table>

Table 2.1 Technology roadmap projections.
2.1.1 Design Tradeoffs

With increases in chip frequency and especially in transistor density, the designer must be able to find the best set of tradeoffs 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 tradeoffs, we have five different considerations. The first is time, which includes partitioning instructions into events or cycles, basic pipelining mechanisms used in speeding up 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 tradeoff. Third, power consumption 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 non-recurring design costs.

Five big issues in 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 see in Chapter 1, in SOC design the non-recurring design costs can dominate the total project cost. Making a design flexible through reconfgurability 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 [7], 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, $T$). VLSI (Very Large Scale Integration) complexity theorists have shown that an $A \times T^n$ bound exists for processor designs, where $n$ usually falls between 1 and 2 [11]. It is also possible to trade off time $T$ for power $P$ with a $P \times T^3$ bound. Figure 2.2 shows the possible trade-off involving area, time, and power in a processor design [2]. 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 tradeoffs. It begins with the issue
Section 2.1 Introduction

Figure 2.1 Makimoto’s Wave.

Figure 2.2 Processor design tradeoffs.

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. The first factor we deal with 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 tradeoffs. This model is restricted to on-chip or processor-type tradeoffs, but it illustrates a type of system design model. As mentioned in chapter one 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 over all 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 tradeoffs 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 tradeoff 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 tradeoff item to derive corresponding specifications. This chapter, when coupled with the essential understanding of the system components which we 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; but 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 flows 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 a 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 OP code) specifies the values for the registers in the system. These control 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.
Figure 2.3 Possible sequence of actions within a cycle.

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).

If we ignore the problem of fitting actions into an integer number of cycles, we can derive an optimal cycle time, $\Delta t$, 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 $T/S = T_{seg}$. Associated with each segment is partitioning overhead. This clock overhead time $C$ (in ns), includes clock skew and any register requirements for data setup and hold.

Now, the actual cycle time (Figure 2.4c) of the pipelined processor is the ideal cycle...
Section 2.2 Cycle Time

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:

$$G = \frac{\Delta t}{\text{performance}} \text{instructions/ns}$$

$$= \left( \frac{1}{1 + (S - 1)b} \right) \times \left( \frac{1}{(T/S) + C} \right).$$

If we find the $S$ for which

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

we can find $S_{\text{opt}}$, 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 ($T_{\text{instr}}$) 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 $T = 12.0$ ns and $b = 0.2$, $C = 0.5$ ns. Then $S_{\text{opt}} \approx 10$ stages.

This $S_{\text{opt}}$ as determined is simplistic—functional units cannot be arbitrarily divided, integer cycle boundaries must be observed, etc. Still, determining $S_{\text{opt}}$ 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 ($S_{\text{opt}}$) 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 a fan-out of four (FO4) gate delay.

Low clock overhead (small $C$) may enable increased pipeline segmentation, but performance doesn’t 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 strategy: 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 with the determination of overall cost.

A good design achieves an optimum cost–performance tradeoff 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 size 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 inches). It might seem that one could simply expand the chip size and produce fewer chips from the wafer, and these larger chips could realize 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 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 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 die to be realized with higher yields. As photolithography and process technology improve their 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.
Yield curve is also a function of time.

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

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

$$d - \sqrt{A}$$
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):

\[
N \approx \frac{\pi}{4A} \left( d - \sqrt{A} \right)^2
\]

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 [4], 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:

\[
\frac{dN_G}{dN_D} = -\frac{N_G}{N} \quad \frac{1}{N_G} dN_G = -\frac{1}{N} dN_D.
\]

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

\[
Yield = \frac{N_G}{N} = e^{-N_D/N}.
\]

This describes a Poisson distribution of defects.

If \( \rho_D \) 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

\[
\left( d - \sqrt{A} \right)^2 \approx d^2
\]

and

\[
\frac{N_D}{N} = \rho_D A,
\]

so that

\[
Yield = e^{-\rho_D A}.
\]
Chapter 2  Chip Basics: Time, Area, Power, Reliability and Configurability

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

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 $\rho_D$ 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 already large $\rho_D \times A$ (≈ 5–10 or more). Thus, the large die designer gambles that technology will lower $\rho_D$ in time to provide a sufficient yield for a profitable product.

2.3.2 Processor Sub-Units

Within a system or processor, the amount of area that a particular sub-unit 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 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 tradeoff.

The obvious unit of area is millimeters square (mm$^2$), but since photolithography and geometries’ resulting minimum feature sizes are constantly shifting, a dimensionless unit is preferred. Among others, Mead and Conway [8] used the unit $\lambda$, 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 $2\lambda$ with a necessary allowance of $3\lambda$ 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 $1\lambda$ 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 $\lambda^2$ and any other
Section 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 2.11 illustrates the increasing dominance of wire delay over gate delay especially in feature sizes less than 0.13 μm. Similarly for feature sizes below 0.20 μm, 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 2.10. 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 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. So that the effects of wire limitations are 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’ll use ideal scaling with the understanding as above.
<table>
<thead>
<tr>
<th>Item Size in $rbe$</th>
<th>Size in $A$ Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 register bit ($rbe$)</td>
<td>$1.0 \text{ rbe}$</td>
</tr>
<tr>
<td>1 static RAM bit in an on-chip cache</td>
<td>$0.6 \text{ rbe}$</td>
</tr>
<tr>
<td>1 DRAM bit</td>
<td>$0.1 \text{ rbe}$</td>
</tr>
<tr>
<td>$rbe$ corresponds to (in feature size: $f$)</td>
<td>$1 \text{ rbe} = 675f^2$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Item</th>
<th>$A$ corresponds to 1 mm$^2$ with $f = 1\mu$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1A$ or about</td>
<td>$= f^2 \times 10^6 \ (f \text{ in microns})$</td>
</tr>
<tr>
<td>A simple integer file (1 read + 1 read/write) with 32 words of 32 bits/word or about</td>
<td>$= 1444 \text{ rbe}$</td>
</tr>
<tr>
<td>A 4KB direct mapped cache or about</td>
<td>$= 23,542 \text{ rbe}$</td>
</tr>
<tr>
<td>Generally a simple cache (whose tag and control bits are less than 1/5th the data bits) uses</td>
<td>$= 4A/KB$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Simple processors (approximation)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A 32-bit processor (no cache and no floating point)</td>
<td>$= 50A$</td>
</tr>
<tr>
<td>A 32-bit processor (no cache but includes 64-bit floating point)</td>
<td>$= 100A$</td>
</tr>
<tr>
<td>A 32b (signal) processor, as above, with vector facilities but no cache or vector memory</td>
<td>$= 200A$</td>
</tr>
<tr>
<td>Area for inter-unit latches, buses, control and clocking</td>
<td>allow an additional 50% of the processor area.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Xilinx FPGA</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A slice (2 LUTs + 2 FFs + MUX)</td>
<td>$= 700 \text{ rbe}$</td>
</tr>
<tr>
<td>A configurable logic block (4 slices) Virtex 4</td>
<td>$= 2,800 \text{ rbe} \approx 1.9A$</td>
</tr>
<tr>
<td>A 18Kb BlockRAM</td>
<td>$= 12,600 \text{ rbe} \approx 8.7A$</td>
</tr>
<tr>
<td>An Embedded PPC405 core</td>
<td>$\approx 250A$</td>
</tr>
</tbody>
</table>

**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, $f$. 
Table 2.3  Density in $A$ units for various feature sizes. One $A$ is 1481 rbe.

![Table 2.3](image)

Figure 2.10  Area scaling with optimum and “practical” shrinkage.

![Figure 2.10](image)

Figure 2.11  The dominance of wire delay over gate delay.

![Figure 2.11](image)
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 tradeoffs 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 80%.

\[ Y = e^{-\rho_D A}, \]

where \( \rho_D = 0.2 \) defect/cm\(^2\), \( Y = 0.8 \). Then

\[ A = 110 \text{ mm}^2 \]

or approximately 1 cm\(^2\).

So the chip area available to us is 110 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 20% of the chip area—usually around the periphery of the chip—to accommodate these functions, then the net area will be 88 mm\(^2\).

Feature Size  The smaller the feature size, the more logic that can be accommodated within a fixed area. At \( f = 0.13\mu \) we have about 5,200\( \Lambda \) or area units in 88 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 32b core processor with an 8KB I cache and a 16KB D cache
- Two 32b vector processors, each with 16 banks of 1K \( \times \) 32b vector memory; an 8KB I cache and a 16KB D cache for scalar data.
A bus control unit.

- Directly addressed application memory of $128K B$.

- A shared L2 cache.

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

<table>
<thead>
<tr>
<th>Unit</th>
<th>Area</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core Processor ($32^b$)</td>
<td>100A</td>
</tr>
<tr>
<td>Core cache ($24KB$)</td>
<td>96A</td>
</tr>
<tr>
<td>Vector Processor #1</td>
<td>200A</td>
</tr>
<tr>
<td>Vector Registers &amp; cache #1</td>
<td>256 + 96A</td>
</tr>
<tr>
<td>Vector Processor #2</td>
<td>200A</td>
</tr>
<tr>
<td>Vector Registers &amp; cache #2</td>
<td>352A</td>
</tr>
<tr>
<td>Bus and bus control (50%)</td>
<td>see below 650A</td>
</tr>
<tr>
<td>Application memory ($128KB$)</td>
<td>512A</td>
</tr>
<tr>
<td><strong>Subtotal</strong></td>
<td><strong>2,462A</strong></td>
</tr>
</tbody>
</table>

**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 2,462\(A\). This leaves a net of 5200 \(\rightarrow\) 2462 = 2738\(A\) 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 2,738\(A\). 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-to-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 2,464\(A\).

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 floorplan is shown in Figure 2.13. A summary of area design rules follow:

1. Compute target chip size from target yield and defect density.
2. Compute die cost and determine whether satisfactory.
3. Compute net available area. Allow 20\% (or other appropriate factor) for pins, guard ring, power supplies, etc.
4. Determine the \textbf{rbe} size from the minimum feature size.
Section 2.5  Power

Growing demands for wireless and portable electronic appliances have focused much attention recently on power consumption. The SIA roadmap points to 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 \( P_{\text{total}} \) 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, \( \text{freq} \) is the device switching frequency, \( I_{\text{leakage}} \) 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 \( V_{\text{th}} \) 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.
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 [1] showed that the drain current is proportional to

\[
I = (V - V_{th})^{1.25}
\]

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 1/2, say from 3V to 1.5V, the operating frequency is also reduced by about 1/2. 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 1/8 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:

\[
\frac{freq_1}{freq_2} = \sqrt[3]{\frac{P_2}{P_1}}
\]

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.
Section 2.6 Area-time-power tradeoffs in processor design

<table>
<thead>
<tr>
<th>Type</th>
<th>Energy Capacity</th>
<th>Duty cycle/lifetime</th>
<th>at Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rechargeable</td>
<td>10,000 mAh</td>
<td>50 Hours (10-20% duty)</td>
<td>400 mw-4 w</td>
</tr>
<tr>
<td>2 x AA</td>
<td>4,000 mAh</td>
<td>1/2 year (10-20% duty)</td>
<td>1-10 mw</td>
</tr>
<tr>
<td>Button</td>
<td>40 mAh</td>
<td>5 years (always on)</td>
<td>1 μW</td>
</tr>
</tbody>
</table>

Table 2.5 Battery capacity and duty cycle.

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).

2.6 Area-time-power tradeoffs in processor design

Processor design tradeoffs 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 multi threading).

2. Embedded processor used in SOC. Processors here are generally simpler in control structure but may be quite elaborate in execution facilities (e.g. DSP). Area is a factor as is design time and power.

2.6.1 Workstation processor

To achieve general purpose performance, the designer assumes ample power. The most basic tradeoff is between high clock rates and the resulting power consumption. Up until the early ’90s ECL (emitter coupled logic) using bipolar technology was dominant in high performance applications (mainframes and supercomputers). At power densities of 80 watts/cm² the module package required some form of liquid cooling. An example from this period is the Hitachi M-880 (Figure 2.14). A 10x10 cm module consumed 800 watts. The module contained 40 die, 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 approaches the same power densities, and similar cooling techniques may be reconsidered.

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
Figure 2.14  Hitachi processor module. The Hitachi M-880 was introduced about 1991 [6]. Module is $10.6 \times 10.6$ cm, water cooled and dissipated 800 watts.

Figure 2.15  Processor frequency for bipolar and CMOS over time.
straightforward concurrency in the architecture (e.g. use of a simple 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>
<thead>
<tr>
<th></th>
<th>Processor on a chip</th>
<th>SOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Area used by storage</td>
<td>80% cache</td>
<td>50% ROM/RAM</td>
</tr>
<tr>
<td>Clock frequency</td>
<td>4 GHz</td>
<td>1/2 GHz</td>
</tr>
<tr>
<td>Power</td>
<td>≥ 50 Watts</td>
<td>≤ 10 Watts</td>
</tr>
<tr>
<td>Memory</td>
<td>≥ 1 GB DRAM</td>
<td>mostly on die</td>
</tr>
</tbody>
</table>

Table 2.6 A typical processor die compared with a typical SOC die.

2.7 Reliability

The fourth important design dimension is reliability [10]; also referred to as dependability and fault-tolerance. As with cost and power there are many more factors which 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, 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 environment such as aging, radiation, temperature or temperature cycling, etc. The probability of physical faults increase 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 sub-unit 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* or the MTBF. So if \( \lambda \) is the fault rate, then:

\[
\lambda = \frac{1}{T}
\]

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.

\[
P(t) = e^{-\lambda} = e^{-t\lambda}
\]

Redundancy is an obvious approach to improved reliability (lower \( P(t) \)). A well known technique is *triple modular redundancy* or 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. 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 design involves simpler hardware built around:
Section 2.7 Reliability

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

- 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.

- Error correction. Since most of the system is storage and memory an error correction code (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 sub-system 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 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 $(5 \times 2)$, running five independent tasks and two processors used as monitors and spares. Within a duplex the processor pair share a common cache and storage system. The processor pair run the same task and compare results. The processors use error detection wherever possible. The cache and storage uses ECC, usually single error correct and double error detect, or SECDDED.

Recent research addresses reliability for multi-processor system-on-chip 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 [12].
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 has 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 error correcting codes (ECC).

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 non-correctable 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 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 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

![Figure 2.18](image-url) Two-dimensional error correcting codes (ECC).
point indicates that two errors have occurred. Hence, either of two valid code data points are equally likely and the message is detectably flawed but non-correctable. 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 address 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 (single error correction, double error detection).

**Example 2.1**  A **Hamming Code Example**

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

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

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 and 12 – 15 and 20 – 21.
- \(k_2\) bits 2 – 3 and 6 – 7 and 10 – 11 and 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 \(m_1–m_{16} = 0101010101010101\). 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):

\[
\begin{align*}
  k_1 &= f_1, \\
  k_2 &= f_2, \\
  k_3 &= f_4, \\
  k_4 &= f_8, \\
  k_5 &= f_{16}.
\end{align*}
\]

Now we have \((m_1\text{ is at }f_3, m_2\text{ at }f_5, \text{ etc.}):\)

\[
\begin{align*}
  f_1 f_2 f_3 f_4 f_5 f_6 f_7 f_8 f_9 f_{10} f_{11} f_{12} f_{13} f_{14} f_{15} f_{16} f_{17} f_{18} f_{19} f_{20} f_{21} \\
  k_1 k_2 0 k_3 1 0 1 k_4 0 1 0 1 0 k_5 1 0 1 0 1.
\end{align*}
\]

Thus, with even parity:

\[
\begin{align*}
  k_5 &= 1, \\
  k_4 &= 1, \\
  k_3 &= 1, \\
  k_2 &= 0, \\
  k_1 &= 1.
\end{align*}
\]

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:

\[
\begin{align*}
  k'_5 &= 0 \text{ (i.e., there is no error in bits 16-21).} \\
  k'_4 &= 1. \\
  k'_3 &= 0. \\
  k'_2 &= 0. \\
  k'_1 &= 0.
\end{align*}
\]

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

\[\diamondsuit\]
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 [3]. 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 are MUXed 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 die.

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 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.

First, 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.

Second, 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.

Third, reliability. The regularity and homogeneity of FPGAs enable 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 semiconductor fabrication process [9].

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 by 1080 pixels, or around 2 million pixels. At 30Hz, 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 3GHz microprocessor that takes, on average, 5 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 100MHz 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 3GHz microprocessor, although its clock rate is only 1/30-th 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 *register bit equivalent, 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 12000 transistors in a logic element (LE) of an Altera device. Empirically each *rbe* contains around 10 logic transistors, so each slice contains 700 *rbe*. A large Virtex XC2V6000 device contains 33,792 slices, or 23.65 million *rbe* or 16400 A.

An 8 by 8 multiplier in this technology would take about 35 slices, or 24500 *rbe* or 17 A. 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 384 *rbe* 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 reconfigurable 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 register bit equivalent—the *rbe*—provides the basis for storage hierarchy tradeoffs 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 tradeoff 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 ($b = 0.2$):

<table>
<thead>
<tr>
<th>Segment #</th>
<th>Maximum delay*</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.7 ns</td>
</tr>
<tr>
<td>2</td>
<td>1.5 ns</td>
</tr>
<tr>
<td>3</td>
<td>1.9 ns</td>
</tr>
<tr>
<td>4</td>
<td>1.4 ns</td>
</tr>
</tbody>
</table>

* 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):

<table>
<thead>
<tr>
<th>Function</th>
<th>Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0.6</td>
</tr>
<tr>
<td>B</td>
<td>0.8</td>
</tr>
<tr>
<td>C</td>
<td>0.3</td>
</tr>
<tr>
<td>D</td>
<td>0.7</td>
</tr>
<tr>
<td>E</td>
<td>0.9</td>
</tr>
<tr>
<td>F</td>
<td>0.5</td>
</tr>
</tbody>
</table>

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 cm $\times$ 1.4 cm) will be produced for five years. Over this period, defect densities are expected to drop linearly from 0.5 defects/cm$^2$ to 0.1 defects/cm$^2$. 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/cm$^2$, feature size = 0.1 $\mu$, 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 (6). Assume a 30 cm diameter wafer costs $15,000.

8. Suppose a 2.3 cm$^2$ die can be fabricated on a 20 cm wafer at a cost of $5,000, or on a 30 cm 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.
Bibliography


