An Energy-Complexity Model for VLSI Computations

Thesis by

José Andrés Tierno

In Partial Fulfillment of the Requirements
for the Degree of
Doctor of Philosophy

California Institute of Technology
Pasadena, California
1995
(Submitted January 11)

Mamá, Papá, Rosana y Jorge.

Acknowledgements

After six years of toil, the journey is over. Caltech was for me an incredible experience, for its people, its traditions, and its enormous accumulated wealth of scientific knowledge. I must thank all of the Caltech community, for all that I learned from them, either directly or indirectly.

From none did I learn more, however, than from Alain Martin. For six years he patiently guided my research, and taught me most of what I know about concurrency and asynchronous design.

A lot of other people has to be especially mentioned, because things would have been different without them. I want to thank Chuck Seitz, for having introduced me to the black magic of VLSI. I want to thank Drazen Borkovic, for having patiently listened to my ramblings for so many years. I want to thank Marcel van der Goot, for his programs, his LaTeXmacros, and many very stimulating conversations around a dictionary. I want to thank Tony Lee, for his wonderful CAD tools, and his incredible energy to fix what didn't work right or do enough. I want to thank Peter Hofstee, for all the lunch-time discussions. I want to thank Jessie Xu, for teaching me some Chinese characters and a little patience. I want to thank Wen-King Su for the Friday-night movies. I want to thank all of the CS185 students, for all of what they taught me.

Finally, a very special thanks to Mr. Oshima, for teaching me that of a one thousand mile journey, nine hundred and ninety nine is only half.

Abstract

An energy complexity model for CSP programs to be implemented in CMOS VLSI is developed. This model predicts with some accuracy the energy dissipation of the "standard" asynchronous VLSI implementation of a CSP program, associated to a given trace of that program. This energy complexity is used in the analysis of CSP programs, in order to optimize this high level representation of asynchronous circuits for energy efficiency. A lower bound to the energy complexity of a CSP program is derived, based on the information theoretical entropy per symbol of the input/output behavior of the CSP program. This lower bound abstracts the specification of the circuit (that is, its input/output behavior), from the implementation of the specification (that is, the text of the program), and therefore applies to any program that meets the specification. A number of techniques are presented to write programs of low energy complexity, and are applied to several examples.

To link the high level representation of circuits to the CMOS representation, several circuits are analyzed to provide standard translations for basic CSP operators into CMOS. In particular, a method for pipelining bus transfers using the sense-amplifier of the bus as a register is proposed.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this chapter we justify the need for a methodology to design energy-efficient digital circuits. Three main reasons are given: it is hard to supply integrated circuits with substantial amounts of power, it is hard to extract from an integrated circuit a substantial amount of heat, and, if the integrated circuit is to operate from batteries, the total amount of energy available is limited. We then explain what type of approach we have taken towards energy-efficient design, and then describe the contents of this thesis.

There are several aspects to be considered when designing an energy-efficient circuit. We can put our effort on the circuits we use to implement logic gates, we can put our effort on the technology we use to fabricate those circuits, or we can put our effort on the algorithm that the circuit uses to perform the computation. In this thesis, the main focus will be directed to improving the algorithms used in the computation, analyzing their performance based on their "energy complexity." This energy complexity is based on the high level algorithmic description of the computation performed by a circuit, and is an abstraction of the actual energy required by the circuit to operate.

In the end, the actual energy dissipation of the circuit is going to be a product, the average energy dissipation per transition, times the number of transitions. Energy complexity addresses the number of transitions required to perform a computation; circuit optimization and technology optimization address the average energy per transition. Both terms of the product can be optimized separately for better energy performance.

1.1 Why Energy Efficiency?

Electric energy, once spent, is transformed into heat that has to be taken from the chip surface. Heat dissipation is a relatively efficient process. DEC's *alpha* microprocessor, for example, dissipates 36 W and can be cooled with a heat sink and forced air circulation. For larger power requirements, Cray Computer's freon-cooling technology can eliminate the 150 kW that a Cray-2 requires for operation. However, the more sophisticated liquid-cooling technologies are expensive, and, as much as possible, we want to be able to use exclusively air cooling.

Air cooling cooling is efficient primarily because of conduction from the chip surface to the heat sink of the package, and convection from the heat sink into the cooling fluid. Radiation plays only a small role, since surface temperature is low — about 70°C — and radiated heat warms up the chassis around the heat sink. This heat also has to be removed by the cooling fluid.

The situation in a satellite is quite different. Because the available mass is finite, any heat put out by the devices on board heats the satellite. The only way of removing this heat is to radiate it into space. Radiation is inefficient at low temperature. Heat pumps can be used, but they require energy to operate and add to the weight of the satellite.

Another problem related to power is getting the energy to the chip's surface. For example, a chip that requires 22 W at 2.2 V uses 10 A of current. This large amount of current will force a high number of the pins in the package to be dedicated to power connections, reducing the number of pins available for signals. In some cases, this problem can be a more serious limit than the amount of heat that can be eliminated from the package.

Both problems are related to packaging technology. In the present state of the art, we can cool any circuit that we can build. However, sophisticated packaging has an elevated cost, and it pays off to design circuits that — for a given function and level of performance — use the minimum possible amount of energy.

If the circuit has to work out of batteries, as in portable computers, satellites, or pacemakers, every bit of power saved translates directly into extended battery life. Portable applications are becoming more important, especially in the area of portable computing and wireless networking. Current portable computers have a limited battery life, about one hour of operation per pound of battery. A practical amount of energy is between 5 to 8 hours of continuous operation: 5 hours for a transcontinental flight, 8 hours for a full day's work away from the office. Extra requirements on portable computers, like network connections, cellular phone connections, and more powerful processors and larger memories, will increase, in the future, the amount of energy required for operation, unless these portable computers are designed in a very different way than they have been designed so far.

1.2 Architectural Energy Optimization

Power optimization at the gate level is important, but if a circuit is inherently wasteful in energy, this optimization will only mitigate a bad problem. The situation is similar to the huge performance improvement of microprocessors over the last 20 years. An Intel 8080 from 1974, operating at 2.5 MHz, will execute about 0.5 MIPS, while an Intel 80486 from 1989, operating at 25 MHz will execute about 15 MIPS. There is a factor of three in circuit performance improvement, which is due only to better system level design. In the same way, low-power has to be designed-in at the system level.

Asynchronous circuits such as the Caltech Asynchronous Microprocessor [21,22] were designed with high performance in mind. It was not until the design was fabricated and tested that an interesting side effect of asynchronous design was noticed: the power delay figures for this processor were very good, executing 200 MIPS per watt at 5 V power supply, 600 MIPS per watt at 2 V power supply. In an asynchronous circuit, in principle, every transition contributes to the computation, and there are no hazards or spurious transitions. Inactive circuits only consume static power, which, in CMOS technology, is a small fraction of the total. On the other hand, it is not unusual for a clocked circuit to spend about a third of the total power just driving the clock lines (this has to be done irrespective of the level of activity of the circuit). On the down side, asynchronous operation may require extra transitions for synchronization, which do not directly contribute to the computation, as well as dual-rail implementation of many logic blocks.

This architectural efficiency is not exclusive of asynchronous design. They can be, and have been, used in the design of low-power synchronous systems.

However, in asynchronous designs the implementation comes for "free": no extra circuitry is needed to shut down the idle parts of the system.

This thesis has, as a central idea, the concept that the sources of energy dissipation can be pinpointed in the high level description of the circuit; this description can then be optimized to eliminate those sources whenever possible. Not all high level descriptions or design methodologies are suitable for such a manipulation. There has to be some correspondence between the operators described in the high level specification language, and the circuits actually generated. There has to be a match as well between the operations described in the specification language, and the activity in the different parts of the circuit.

1.3 Low-Energy vs. Low-Power

Electrical power dissipation has been used as a figure of merit for energy efficient applications. Power is a convenient measure for synchronous circuits with no power management, where power dissipation is very much independent of the level of activity of the circuit. In this context, reducing power will reduce energy consumption. Asynchronous operation is better described in terms of reactive programs: energy is dissipated only when the circuit is active. For asynchronous systems, a proper measure of performance is the "energy per operation." This metric measures the energy required to execute an instruction, fetch a piece of data from memory, service an interrupt, etc. To maximize the battery life, we can minimize the average energy per operation, that is, we maximize the number of instructions that we can execute with one battery charge.

Energy per operation is an additive quantity. Given a computation described in terms of some more elementary operations, we can calculate the energy required to execute that computation by adding the energies required by the elementary operations. In this way, we can compare different algorithms to execute the same computation in terms of energy efficiency, independent of timing considerations. A comparison based on power requires precise knowledge of timing and the way in which the computation is going to be implemented.

1.4 Energy Model for High-Level Programs

The first step towards tying the high level description of a system to the energy performance of the circuit itself, is having an energy model, not of the circuit, but of the high level constructs — a model that can predict accurately enough what is going to happen in the circuit before the circuit is even designed. Accuracy is necessary in relative terms; at this level, we want to compare high-level designs, or, given a design, see how we can improve on that design. This model must somehow express the trade-offs that are possible at the circuit-level design stage. For example, for each delay we can design the most efficient circuit in terms of energy per operation; this delay-to-energy function becomes part of the model. We build the model for more complex operators using the models for the parts of those operators.

This energy model is expressed as the energy complexity of the CSP program that we use as the specification for the circuit. This energy complexity counts the number of transitions executed by the program to compute a given trace, or set of traces, taking into account the fan-in and fan-out of these transitions to give them weights.

1.5 Low-Energy Programming Techniques

Given the ideas presented in the previous sections, asynchronous circuits designed using Martin synthesis [18,19,20], appear to be an ideal target for the techniques to be presented in this thesis. In the Martin synthesis method, the circuit to be designed is specified as a CSP program [14]. This CSP program is then successively refined by semantics-preserving program transformations, expanded into handshaking variables, and finally translated into a network of boolean operators. Each CSP operator (send, receive, add, etc.) corresponds to some specific circuitry; every time that a CSP statement is executed, the corresponding circuitry is activated.

The techniques to be presented here are of general use; they can be applied to other design methodologies, even to synchronous circuits. There is, however, an underlying assumption that the target technology is CMOS, or any other technology where the static power dissipation (that is, the power that is independent of the level of activity of the circuit) is only a small part of the total power dissipation. The energy complexity techniques assume that energy has to be spent only to change the state of the circuit, and no energy is required to maintain the state of the circuit.

Low-energy programs are reactive: work is done only when requested by the environment. In a CSP program, this characteristic applies to individual processes and excludes busy-waiting or input-polling strategies.

Low-energy programs are "lazy": work is postponed as much as possible, reducing preparatory work to a minimum and hoping to avoid unnecessary computation. Lazy programs exhibit a very good average-delay average-power product, at the cost of an increased worst-case delay. The trade-off between energy per computation and worst-case delay is the most important consequence of lazy programming.

1.6 Shorter Delays for the Same Energy

Low-energy circuits will almost always have worse delays than circuits built for speed. It is important to see how to make these delays shorter, without paying too much of a price in energy.

Techniques such as pipelining, caching, or concurrency, have been well exploited to increase the throughput of digital systems. We look at these techniques from the low-energy point of view, to see what their cost or benefit is, which ones are preferable, or when to use each of them.

In asynchronous systems, communication reshuffling can increase throughput, by making signal dependencies closer to what data dependencies are for the registers in the circuits. This communication reshuffling is an important optimization because it almost never has an energy overhead.

1.7 Energy and Entropy

The energy complexity of a CSP program is used as an indication of how efficient this program is, and can be used to compare two solutions to the same problem. There is, however, the following question: given a specification, what is the lowest energy complexity of any program that satisfies that specification? If we can compute this lower bound, we can compare our solutions not only to each other, but to the best possible solution.

Such a lower bound can be computed from the input/output behavior of the specification. If the sequence of input/output symbols is completely predictable, then there is no need to compute it (we already know the result), and the minimum energy complexity is zero. If the sequence of input/output symbols is completely random, then there are no "easy" ways of computing that sequence, and the minimum energy complexity is high. Based on these intuitive considerations, we look at the information-theoretic entropy of the sequences of input/output actions that the specification requires. Entropy can be used as a measure of the "randomness" of these sequences. As it turns out, this entropy can be used as a lower bound for the energy complexity of a CSP program; in some cases, this lower bound is tight.

There are other consequences of this lower bound. Because of the way it is constructed, we can apply information-theory results to the construction of CSP programs, using the statistics of the input/output sequences. Process decomposition, concurrency, pipelining, and control/data decomposition can be directed using entropy arguments. Also, entropy can be used to refine and simplify the specification of the problem.

1.8 Transistor Level Techniques

The models used for evaluating the performance of CSP programs make some assumptions about how the corresponding circuit looks. For some types of operations, like register arrays, buses, completion trees, and control-signal buffering, the interaction between the parts is so important that they have to be treated as a unit.

These operations have to be designed with a great level of detail, going all the way to the transistor description. Even layout issues are of importance, and they will influence the parameters of the model. High-level optimization is very powerful, but the final product still will be a transistor circuit, and the characteristics of transistors have to be taken into consideration.

A good collection of datapath operators is also important, since it can be used to verify some of the assumptions made by the energy complexity model, and to compute parameter values in the model. We look, in particular, at efficient implementations for bus transfers, which are an expensive on-chip data movement operation.

1.9 Contributions

This thesis makes the following contributions:

- It shows how to generate an energy model from the high-level description of a circuit. Because this model separates the high-level description of a circuit from the details of the implementation, it makes it possible to compare programs, or choose architectural parameters.

- It demonstrates a number of high level programming techniques to obtain low-energy circuits, or to improve on existing circuits.

- It demonstrates a number of high level programming techniques to improve the throughput of a program with a small cost in energy.

- It introduces a lower bound to the attainable energy complexity of a CSP program that has to satisfy a given specification. It also demonstrates how to use this lower bound to direct the process decomposition so that the energy complexity of the final program is as close as possible to the lower bound.

- It demonstrates a number of specialized datapath circuits for low-energy asynchronous design, in particular a pipelined bus transfer, using the sense amplifiers on the bus as an extra register.

- These techniques are applied to the design of a microprocessor, to show with an example the impact they can have.

1.10 Contents

Chapter 1 is a brief introduction to the contents of this thesis.
Chapter 2 explains the causes of power dissipation in CMOS and proposes an energy model for CSP programs that takes into account trade-offs between energy, delay, and power supply voltage. A performance index is proposed that captures the essence of those trade-offs, enabling the designer to better compare alternative circuits for the same computation.
Chapter 3 analyzes the energy complexity of CSP programs, based on the input/output behavior of the specification of the program. As a result, we

derive a lower bound to the attainable energy complexity of any solution to a given specification. This lower bound is based on the information-theoretic entropy of the sequences of input/output symbols.

Chapter 4 takes a detailed look at low-energy programming. Reactive programming, lazy programming, energy/worst-case-delay trade-off, caching, pipelining, concurrency, and non-causal probes, are analyzed to evaluate their relative merits.

Chapter 5 shows that the minimum-delay transistor-sizing problem is equivalent to the minimum-energy transistor-sizing problem. We propose a gate energy/delay model for accurate minimum-energy sizing based on least-squares posynomial approximation of the gate equations.

Chapter 6 goes back to the transistor level issues of low-energy design. Some specialized circuits are proposed to deal with buses, register arrays, completion trees, and other datapath structures.

Chapter 7 discusses self-limitation as a way to prevent high-power asynchronous circuits from exceeding the maximum allowable chip temperature. We also show how we can select the supply voltage so that the circuit works at the highest possible speed.

Chapter 8 Shows how to apply some of the techniques described in this thesis to the design of an asynchronous microprocessor based on the architecture of the Caltech Asynchronous Microprocessor.

Chapter 9 makes some concluding remarks about this work.

Chapter 2

Energy Model for CSP Programs in CMOS

In this chapter we present an energy model for asynchronous circuits derived from the CSP specification of the circuit. The model is based on the energy dissipated per operation (i.e. executing one instruction of a processor, accessing one element of a memory array, etc.).

The CSP description abstracts the notion of timing by providing the partial ordering of actions that the circuit has to perform. Likewise, to calculate the energy per operation we do not need timing information; we only have to add the energy required by each of the sub-parts of that operation.

The energy dissipation of a CMOS circuit is dependent on the power supply: the speed of operation and the energy required to charge capacitors increases at higher voltages. In this chapter we derive an energy-based index of performance that is independent of the power-supply voltage and we use it to justify an energy model for asynchronous circuits based on the energy cost of communication actions. This model is based on counting transitions, in the sense that we tally all the actions in the trace of an execution of the CSP program. Just counting transitions is, however, not enough, since not all transitions are equivalent. Variables that are accessed from several places in the circuit are more expensive in energy than localized, non-shared variables. The extra read and write circuits and wires used to distribute the value of the variable increase the cost of switching that variable to a new value. It is possible to reduce the energy cost of a program and at the same time increase the number of transitions, either by making each of those transitions cheaper, or by simplifying the average-case performance at the expense of the worst-case performance. Therefore, we take into account the fan-in and fan-out of all variables in the energy model.

As an example of the use of the energy model, we analyze the design of asynchronous memories. Memory subsystems are usually designed for speed and density, with secondary consideration given to energy. Memory is slow compared to processors; high throughput in memory is achieved through parallelism (wide data-words) and prediction (memory caching). These same design techniques can be used to improve energy performance.

First, we show how to partition a memory array to minimize access energy under the assumption that all addresses are equally probable. Second, we show how to use the statistics of long sequences of addresses to further reduce the average energy per access. These techniques result in a trade-off between area and energy per access. This analysis shows that conventional commercial architectures are not optimal from the point of view of energy efficiency.

2.1 Energy Index

The energy dissipation of a CMOS circuit is dependent on the supply voltage: the speed of operation and the energy required to charge capacitors increases at higher voltages. In order to evaluate the energy efficiency of a high level circuit description, we need a measure of energy dissipation that is independent of the supply voltage. In this section, we derive such an index of performance and use it in the following section to justify an energy model for asynchronous circuits based on the energy cost of communication actions and synchronization primitives.

2.1.1 Sources of Energy Dissipation

CMOS circuits have three main sources of energy dissipation: leakage currents, short-circuit currents, and dynamic currents. The total energy dissipated during the execution of one operation, E_T, can be calculated as:

$$E_T = E_s + E_d + E_{sc} \tag{2.1}$$

where E_s is the energy dissipated by the sub-threshold leakage currents, E_d is the energy used for charging and discharging capacitors, and E_{sc} is the energy dissipated by the short-circuit currents.

Leakage currents come from the sub-threshold behavior of MOSFET's. For $V_{GS} < V_{th}$, the channel conductance, g_c, can be modeled by [33]:

$$g_c = I_c \frac{q}{kT} \exp\left(\frac{q(V_{GS} - V_{th})}{kT}\right) \tag{2.2}$$

All these currents add up and are responsible for an energy dissipation of the form:

$$E_s = \int V_{DD}^2 I_c \frac{q}{kT} \exp\left(-\frac{qV_{th}}{kT}\right) dt \tag{2.3}$$

where $V_{GS} = 0$ is assumed. At the present state of the technology, energy dissipation due to leakage currents represents only a small fraction of the total power of a CMOS circuit.

Short-circuit currents originate in the short transients, as in the case of a CMOS inverter, when both pull-up and pull-down transistors conduct while the input signal switches between V_{thn} and $V_{DD} - V_{thp}$. This energy dissipation has the form [34]:

$$E_{sc} = \sum s_i \left(V_{DD} - 2V_{th}\right)^3 \tag{2.4}$$

where the s_i's are proportionality constants, and the sum is made over all transitions executed in one operation. Short-circuit currents also play a significant role in storing a value into a flip-flop built from cross-coupled inverters.

Dynamic energy dissipation, E_d, comes from the energy used to charge the capacitors in the circuit. The capacitors are then discharged to ground, and the energy is not recuperated. E_d can be computed as:

$$E_d = \sum_{C_i} n_i C_i V_{DD}^2 \tag{2.5}$$

where the C_i's are all the capacitors in the circuit, and n_i is the number of times the capacitor is switched in the execution of one operation. We rewrite Eq. 2.5 as:

$$E_d = K_L V_{DD}^2 \tag{2.6}$$

2.1.2 Linear Energy Model

Using Eqs. 2.4 and 2.6, and neglecting the effect of sub-threshold currents, we rewrite the energy equation as:

$$E_T = \left(K_L + K_S \frac{(V_{DD} - 2V_{th})^3}{V_{DD}^2} \right) V_{DD}^2 \tag{2.7}$$

Outside the sub-threshold region, ($V_{DD} \gg V_{th}$), Eq. 2.7 simplifies to:

$$E_T = (K_L + K_S V_{DD}) V_{DD}^2 \tag{2.8}$$

Fig. 2.1 shows E_T/V_{DD}^2 as a function of V_{DD} for a 4-bit counter (SPICE simulation), and for the Caltech Asynchronous Microprocessor [21,22] and for a $3x + 1$ engine [17] (measurement). This figure shows that the linear approximation of Eq. 2.7 is indeed accurate.

Based on these results, we propose, as an index of performance for an asynchronous CMOS circuit, the corresponding constants K_L and K_s. These indices are independent of the power-supply voltage and the speed of operation; furthermore, K_L and K_s are additive: we can calculate the index corresponding to an operation by adding the indices of all of its sub-operations.

As a first-order approximation, we assume $K_s = 0$, and use K_L as the energy performance index.

2.2 Energy Model for CSP Programs

The CSP specification of an asynchronous circuit corresponds very closely to its implementation. For each assignment, communication and function evaluation executed by the CSP program there will be a corresponding assignment, communication, function evaluation computed by the CMOS implementation. In fact, it is possible to do a purely syntactic translation from CSP into CMOS [3]. The CMOS implementation will dissipate energy only during the execution of the assignment, etc. This energy can be assimilated to the energy required to execute the corresponding CSP statement. To calculate the energy required to execute a CSP program, we add the energy required to execute each statement in a "canonical" trace of that program; we can also use the relative

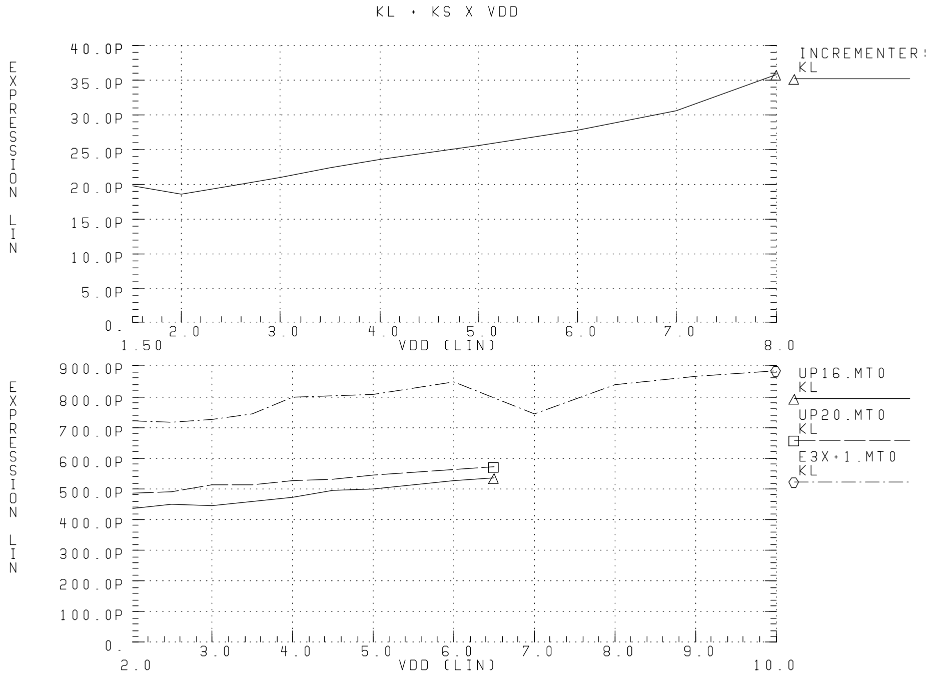


Figure 2.1: Graph of E_T/V_{DD}^2 against V_{DD} for a 4-bit counter (SPICE simulation), and for the 3x+1 engine, and the 1.6μm and 2.0μm processors.

frequencies of occurrence of each statement in the program on a reasonably large set of typical traces.

We would like to be able to map each statement into an energy performance index, independently of the other statements in the program. In general, it is not possible to do so; layout constraints result in the length — and therefore capacitance — of wires being affected by the connectivity of the whole circuit, not just the local connections. A detailed energy model would have to take into consideration the program as a whole, instead of individual statements.

The purpose of the model is, however, to study architectural trade-offs (e.g., compare bit-serial and parallel implementation of a function) or determine architectural parameters (e.g., determine the optimal width of a cache memory). A very detailed model with a large number of parameters can be intractable and not that much more accurate if those parameters are layout-dependent (and, therefore, not well known before the layout is finished). At the architectural design stage a simpler model is desirable; we will base this model on the cost of communication, assignment, and selection.

The model proposed is based on the energy performance index. To each type of statement, we assign a capacitance that is representative of the energy that we expect that operation to cost in a typical implementation.

As much as possible, we want to remove syntactic dependencies from the energy model; similar programs that result in identical implementations should have the same energy performance. To this effect, we define an energy index for just a few constructs that can be used to implement any CSP program.

2.2.1 Synchronization

The synchronization primitives of CSP are parallel composition ('$\|$'), sequential composition (';'), guarded choice ('$\square$'), repetition, and bullet synchronization between communication actions ('$\bullet$'). Some of these primitives have zero energy cost, such as parallel composition. Some of these primitives require extra hardware to be implemented, such as guarded choice. We make an estimate of the extra hardware to assign a cost to each primitive.

Concurrency:

A basic postulate of this model is that parallel composition is free: no extra circuits are required in the implementation. If there is no synchronization between the P_i's, we can write:

$$\mathcal{C}(\langle \| i : 1..n : P_i \rangle) = \sum_{i=1}^{n} \mathcal{C}(P_i) \tag{2.9}$$

where $\mathcal{C}()$ is the cost function that assigns an energy index to a program.

Sequencing:

Sequencing synchronizes the end of an action with the beginning of the next action. If the previous action is the parallel composition of several actions, the end of those actions has to be synchronized with a tree, which has a linear energy cost. If the next action is the parallel composition of several actions, the start signal has to be distributed to them (maybe with a tree), which also has a linear cost (see Fig. 2.2). We can express these costs with the following equation:

$$
\begin{aligned}
&\mathcal{C}(\langle \| i : 1..n : P_i\rangle ; \langle \| j : 1..m : Q_j\rangle) = \\
&\qquad \mathcal{C}(\text{join}(n)) + \mathcal{C}(\text{fork}(m)) + \mathcal{C}(;) + \mathcal{C}(\langle \| i : 1..n : P_i\rangle) + \mathcal{C}(\langle \| j : 1..m : Q_j\rangle) \\
&= \quad K_j(n-1) + K_f(m-1) + K_{sc} + \sum_{i=1}^{n} \mathcal{C}(P_i) + \sum_{j=1}^{m} \mathcal{C}(Q_j) \qquad (2.10)
\end{aligned}
$$

where K_j, K_f, and K_{sc} are technology dependent constants. If $n = 1$ and $m = 1$, then the fork and join circuits are not needed, and the cost of sequencing is just the constant overhead K_{sc}.

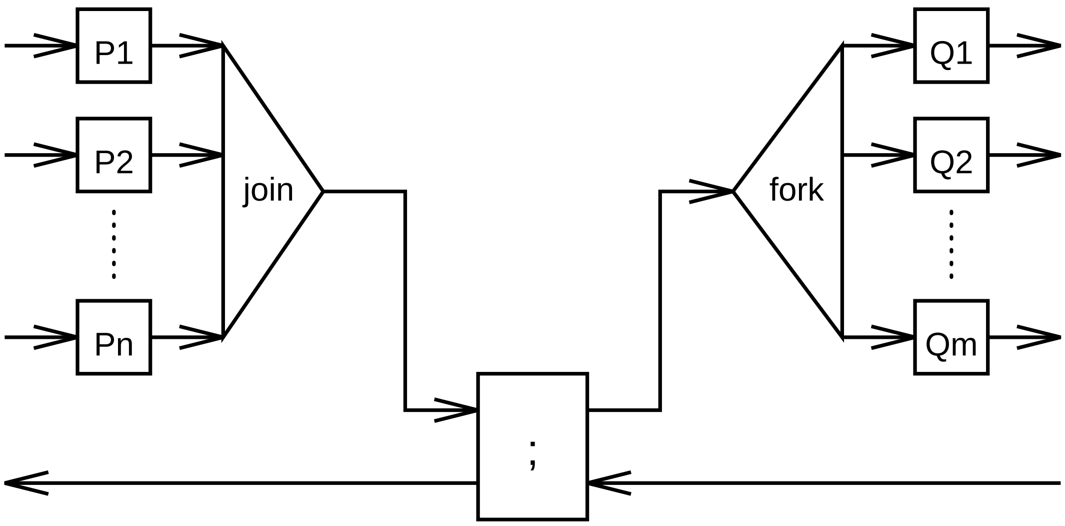


Figure 2.2: Extra circuits required to implement sequencing between two blocks of concurrent processes. The *fork* and *join* trees can be configured in several ways.

Consider, for example, the following programs:

$P_1 \equiv (A_1; ..; A_n) \parallel (B_1; ..; B_n)$

$P_2 \equiv (A_1 \parallel B_1); ..; (A_n \parallel B_n)$

Using Eqs. 2.9 and 2.10 we can compute the cost of P_1 and P_2 as:

$$\mathcal{C}(P_1) = 2K_{sc}(n-1) + \sum_{i=1}^{n}(\mathcal{C}(A_i) + \mathcal{C}(B_i)) \tag{2.11}$$

$$\mathcal{C}(P_2) = (K_{sc} + K_f + K_j)(n-1) + \sum_{i=1}^{n}(\mathcal{C}(A_i) + \mathcal{C}(B_i)) \tag{2.12}$$

$$\mathcal{C}(P_1) - \mathcal{C}(P_2) = (K_{sc} - (K_f + K_j))(n-1) \tag{2.13}$$

The relative efficiency of these two programs depends on the value of $K_{sc} - (K_f + K_j)$. Program P_2 reduces the number of sequencing operators at the expense of reduced concurrency and extra join and merge circuits.

Choice:

Guarded choice can be implemented in a number of ways. We consider the cost of selection as the difference in cost between the following two programs:

$PAR \equiv \langle \parallel i : 1..n : \ *[[\ G_i \longrightarrow A_i \ ]] \rangle$

and,

$CHOOSE \equiv *[[\langle \ [] \ i : 1..n : \ G_i \longrightarrow A_i \ \rangle]]$

Program $CHOOSE$ can be transformed into program PAR using the P and V operations on a semaphore:

$CHOOSE \equiv \langle \parallel i : 1..n : *[[\ G_i \longrightarrow P; [G_i \longrightarrow A_i [] \neg G_i \longrightarrow \textbf{skip}]; V \ ]] \rangle$

If the guards G_i are stable, we can simplify the implementation to:

$CHOOSE \equiv \langle \parallel i : 1..n : *[[\ G_i \longrightarrow P; A_i; V \ ]] \rangle$

The cost of choice is, therefore, the cost of implementing that semaphore. A number of implementations are possible and practical; a state variable per choice and an extra process can be used to that effect:

$CHOOSE \equiv \ \langle \parallel i : 1..n : \ *[[\ \neg u \ \wedge \ G_i \longrightarrow u_i\uparrow; [u]; A_i; u_i\downarrow \ ]] \rangle$

$$\parallel *[[\langle \vee \ i : 1..N : u_i]; u\uparrow; [\langle \wedge \ i : 1..N : \neg u_i]; u\downarrow]$$

The cost of selection can be reduced to the cost of an Or-gate with fan-out proportional to the number of choices. If statement i is chosen, we can express the energy required to execute the selection plus statement i as:

$$\begin{aligned}&\mathcal{C}(\,[\langle \Box : i : 1..n : G_i \to A_i \rangle]\,, i) = \\ &\qquad K_o \log_2 n + K_f n + \mathcal{C}(\,[\neg u \wedge G_i]\,; u_i{\uparrow};\, [u]\,; A_i; u_i{\downarrow}) \end{aligned} \tag{2.14}$$

where K_o and K_f are technology constants, corresponding to the energy cost of computing an n-input 'or' and forking the result to n receivers.

We can get rid of the indexation of the cost function by using the frequencies with which each statement is taken. We can write:

$$\begin{aligned}&\mathcal{C}(\,[\langle \Box : i : 1..n : G_i \to A_i \rangle]\,) = \\ &\qquad K_o \log_2 n + K_f n + \sum_{i=1}^{n} p_i \mathcal{C}(\,[\neg u \wedge G_i]\,; u_i{\uparrow};\, [u]\,; A_i; u_i{\downarrow}) \end{aligned} \tag{2.15}$$

where p_i is the conditional frequency of selection of statement i in the traces of the program, given the history of the computation.

Alternatively, if the guards are stable, the semaphore may be implemented with a selection tree:

$$\begin{aligned} \mathit{CHOOSE} \equiv\ & \langle\, \| \ i : \mathit{1..N} :\ \ *[[\ \ G_i \longrightarrow U_i; A_i; U_i\ \ ]]\rangle \\ & \| \ *[[\langle\ \ \Box\ \ i : \mathit{1..N/2} :\ \ \overline{U}_i \longrightarrow L; U_i; U_i; L\ \ \rangle]] \\ & \| \ *[[\langle\ \ \Box\ \ i : N/\mathit{2} + \mathit{1}..N :\ \ \overline{U}_i \longrightarrow H; U_i; U_i; H\ \ \rangle]] \\ & \| \ *[[\ \ \overline{L} \longrightarrow L; L\ \ \Box\ \ \overline{H} \longrightarrow H; H\ \ ]] \end{aligned}$$

We apply this transformation recursively, and we get a cost of selection that is logarithmic on the number of choices and can be expressed as:

$$\begin{aligned}&\mathcal{C}(\,[\langle \Box : i : 1..n : G_i \to A_i \rangle]\,) = \\ &\qquad K_c \log_2 n + \sum_{i=1}^{n} p_i \mathcal{C}(\,[G_i]\,; U_i; A_i; U_i) \end{aligned} \tag{2.16}$$

Either Eq. 2.15 or 2.16 can be used to compute the cost of guarded selection, depending on the expected implementation of the command. The linear cost implementation is more efficient for low values of n, while the logarithmic cost implementation has more concurrency in the guard evaluation, and scales better for large values of n.

We can use the frequencies of the guarded commands to improve the average cost by using the Huffman tree for the p_i's instead of a symmetric tree. We will see more about this on Chapter 3.

The remaining question is whether a better implementation (i.e., with a worst-case better that $\log_2 n$) of the choice statement exists. We will prove that this is not true under some fairly general assumptions that are justified in terms of the CMOS energy model.

We implement the mutual exclusion between the branches of the selection statement with shared variables. Let n be the number of branches in the selection statement, s_i be the number of variables that have to be set in branch i to enter the critical region, and q_i the number of variables that have to be consulted in branch i to enter the critical region; each of these variables is written by w_{ij} and read by r_{ij} different branches respectively, with $w_{ij} \geq 2$ and $r_{ij} \geq 2$ for all i, j.

Now, the s_i written variables have to cover all branches to ensure mutual exclusion; therefore, for all i, $\sum_{j=1}^{s_i} w_{ij} \geq n$; the same is true for the q_i read variables, and therefore $\sum_{j=1}^{q_i} r_{ij} \geq n$.

Writing into a variable with m write ports can be accomplished with $K_w \log_2 m$ energy cost, using a tree multiplexer to write into the variable; reading from a variable with m read ports can be accomplished with $K_r \log_2 m$ energy cost, using a tree demultiplexer. The average cost per branch of mutual exclusion can be expressed as:

$$\mathcal{C}(n) = \frac{1}{n} \sum_{i=1}^{n} \left(K_w \sum_{j=1}^{s_i} \log_2 w_{ij} + K_r \sum_{j=1}^{q_i} \log_2 r_{ij} \right) \tag{2.17}$$

If $a_i \geq 2$ for all i, then $\sum_i \log a_i \geq \log \sum_i a_i$, and we can write the following inequality:

$$\begin{aligned} \mathcal{C}(n) &\geq \frac{1}{n} \sum_{i=1}^{n} \left(K_w \log_2 \sum_{j=1}^{s_i} w_{ij} + K_r \log_2 \sum_{j=1}^{q_i} r_{ij} \right) \\ &\geq \frac{1}{n} \sum_{i=1}^{n} \left(K_w \log_2 n + K_r \log_2 n \right) \end{aligned}$$

$$= \quad (K_w + K_r)\log_2 n \tag{2.18}$$

Equality is achieved for a single variable shared by all branches.

In conclusion, assuming that we cannot implement access to a shared variable with less than logarithmic cost, the worst-case cost per branch of a selection statement is always at least logarithmic in the number of branches. Exact logarithmic cost is achieved if we implement mutual exclusion with a single variable.

2.2.2 Assignments and Communication

The datapath of a circuit very often consumes the bulk of the energy required for operation. This is especially true for wide datapaths, where the overhead of control is relatively small. In the energy cost of the datapath, we will also include the cost of the control lines that drive it.

There are three main datapath operations: assignments, communications, and guard and function evaluation. Guard and function evaluation is treated later, assignment and communication is analyzed in this section.

A CSP assignment can be implemented with data communications. In fact, the usual implementation of an assignment is as follows:

$$\ldots; x := y; \ldots \;\; \triangleright \;\; \ldots; (A!x \,\|\, A?y); \ldots$$

where channel A is a new channel.

A CSP data communication involves two actions: first, copying the data into the wires that implement the communication channel, and second, copying the data from the communication channel into a register; the second part may not be present if the data is to be tested on the channel wires.

We assign an energy cost to the two parts of the communication, send and receive. In this case, copying data into the channel wires has a cost which is proportional to the number of bits of data. Copying data into the channel wires has an extra cost related to the capacitance on those wires. If the channel is shared by several registers, the capacitance of the channel wires will have contributions from the write ports to the channel, from the read ports to the channel, and from the wire itself. As a simplifying assumption, we will consider that the length of the wire scales with the number of writers, plus the number

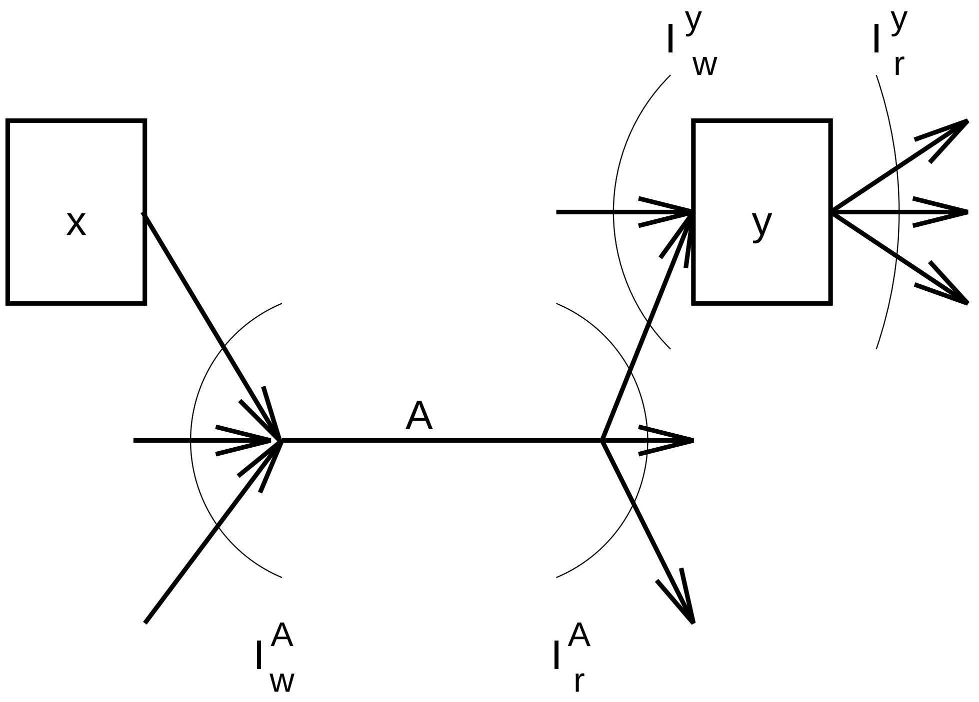


Figure 2.3: Energy cost of an assignment. I_w^A and I_r^A are the fan-in and fan-out of channel A; I_w^y and I_r^y are the fan-in and fan-out of register y.

of receivers. Under these conditions, we can write:

$$\mathcal{C}(A!x) = (K_r I_r^A + K_w I_w^A + K_s) \log_2 N \tag{2.19}$$

where I_r^A and I_w^A are the number of read and write ports, respectively, on channel A; K_r, K_w, and K_s are technology constants; and N is the number of different values that the variable x can take (see Fig. 2.3). K_r corresponds to the contribution of each read port to the capacitance of the channel, K_w corresponds to the contribution from each write port to the capacitance of the channel, and K_s corresponds to the contribution from the pre-charge circuit and other fixed costs (sense-amplifier if present, for example) to the capacitance of the channel.

Likewise, The cost of assigning to a register increases with the complexity of that register. More read and write ports represent added capacitance to the storage nodes of the register; we can write:

$$\mathcal{C}(A?y) = (K_r^l I_r^y + K_w^l I_w^y + K_s^l)\log_2 N \tag{2.20}$$

where I_r^y and I_w^y are the number of read and write ports respectively on variable y; K_r^l, K_w^l, and K_s^l are technology constants; and N is the number of different values that the variable y can take (see Fig. 2.3).

The technology parameters K_r, K_w, K_s, K_r^l, K_w^l, and K_s^l can be measured, for example, from a SPICE simulation of a test circuit. To this effect we designed a standard register with multiple read and write ports and measured the energy index of several transfers between registers, separating the energy required to assign to the bus from that required to assign to the register. The register energy is the energy required to change the value of the register (if the register doesn't change value, the energy is zero). The measurement was made at several different voltages, and the parameters were computed using a least-squares approximation. The values of the parameters in fF for the $1.2\mu m$ HP process are:

$$\mathcal{C}(A?y) = (22.3I_r^y + 22.4I_w^y + 66.8)\log_2 N \tag{2.21}$$

$$\mathcal{C}(A!x) = (15.6I_r^A + 9.6I_w^A + 10.7)\log_2 N \tag{2.22}$$

The raw data is shown on Table 2.1.

	Bus				Register			
		I_w^A				I_r^y		
	I_r^A	1	2	3	I_w^y	1	2	3
SPICE	1	35.5	45.7	55.6	1	112.8	131.3	157.5
Model		36.0	45.6	55.2		111.6	133.9	156.2
SPICE	2	51.8	61.7	70.2	2	135.4	153.6	179.9
Model		51.6	61.2	70.9		134.0	156.3	178.6

Table 2.1: Measured (SPICE) and predicted (Model) energy index for several register/bus configurations (results in fF). The worst-case relative error is less than 2%.

The numbers I_r^A, I_w^A, I_r^y, I_w^y can be derived syntactically from the text of the program by counting in how many places a variable is used, how many different variables can be sent over a channel, number of assignments, guard evaluations, etc. There is, however, the possibility of optimizing the channel assignment by adding extra intermediate registers and other high level optimization steps that cannot be derived from the program text, but are dependent on the optimization algorithm used. Since this type of optimization can be expressed in CSP, we only consider explicit channels and assignments.

We will show next how to assign channels to optimize the cost of data communications. Consider, for example, a single-bit register with multiple read and write ports, where the mutual exclusion between the reads and writes is guaranteed by the environment:

$$\begin{array}{lll} \mathit{REG} \equiv & & \langle \parallel \ i : \mathit{1..r} : \ *[[\ \overline{R}_i \longrightarrow R_i!x \ ]] \ \rangle \\ & \parallel & \langle \parallel \ i : \mathit{1..w} : \ *[[\ \overline{W}_i \longrightarrow W_i?x \ ]] \ \rangle \end{array}$$

and the environment can be modeled by:

$$\begin{array}{l} \mathit{ENV} \equiv *[[\langle \ [] \ i : \mathit{1..r} : \quad \mathbf{true} \longrightarrow R_i?y_i \ \rangle \\ \qquad\qquad \langle \ [] \ i : \mathit{1..w} : \quad \mathbf{true} \longrightarrow W_i!z_i \ \rangle \\ \qquad]] \end{array}$$

From the traces of the program we compute the frequencies of each communication action; these frequencies are p_i^r for the read ports, and p_i^w for the write ports, with $\sum_i p_i^r = p_r$, $\sum_j p_j^w = p_w$, and $p_r + p_w = 1$. The average cost per communication can be computed as:

$$\begin{aligned}
p_r\mathcal{C}(R) &= p_r(K_r + K_w + K_s) + \sum_i p_i^r(K_r^l I_r^{y_i} + K_w^l I_w^{y_i} + K_s^l) \\
p_w\mathcal{C}(W) &= p_w(K_r^l r + K_w^l w + K_s^l) + p_w(K_r + K_w + K_s) \\
\mathcal{C}(RW) &= p_r\mathcal{C}(R) + p_w\mathcal{C}(W) \\
&= (K_r + K_w + K_s) + p_w(K_r^l r + K_w^l w + K_s^l) + \\
&\quad \sum_i p_i^r(K_r^l I_r^{y_i} + K_w^l I_w^{y_i} + K_s^l)
\end{aligned} \tag{2.23}$$

To improve on this energy cost, we reduce the number of channels to variable x by merging the channels W_1 and W_2:

$$\begin{aligned}
REG \equiv\ & \langle \| \ i : 1..r :\ *[[\ \overline{R}_i \longrightarrow R_i!x\ ]]\ \rangle \\
\| \ & \qquad\qquad\quad *[[\ \overline{W}_{12} \longrightarrow W_{12}?x\ ]] \\
\| \ & \langle \| \ i : 3..w :\ *[[\ \overline{W}_i \longrightarrow W_i?x\ ]]\ \rangle
\end{aligned}$$

$$\begin{aligned}
ENV \equiv *[[\ & \mathbf{true} \longrightarrow W_{12}!z_1 \\
[]\ & \mathbf{true} \longrightarrow W_{12}!z_2 \\
\langle\ []\ & i : 1..r :\ \ \mathbf{true} \longrightarrow R_i?y_i\ \rangle \\
\langle\ []\ & i : 3..w :\ \ \mathbf{true} \longrightarrow W_i!z_i\ \rangle \\
]]&
\end{aligned}$$

The new energy cost per communication action from register x, $\mathcal{C}'(RW)$, can be computed as:

$$\begin{aligned}
p_r\mathcal{C}'(R) &= p_r(K_r + K_w + K_s) + \sum_i p_i^r(K_r^l I_r^{y_i} + K_w^l I_w^{y_i} + K_s^l) \\
&= p_r\mathcal{C}(R) \\
p_w\mathcal{C}'(W) &= p_w(K_r^l r + K_w^l(w-1) + K_s^l) + \\
&\quad (p_w - p_1^w - p_2^w)(K_r + K_w + K_s) + (p_1^w + p_2^w)(K_r + 2K_w + K_s) \\
&= p_w\mathcal{C}(W) - p_w K_w^l + (p_1^w + p_2^w)K_w \\
\mathcal{C}'(RW) &= p_r\mathcal{C}'(R) + p_w\mathcal{C}'(W) \\
&= \mathcal{C}(RW) - p_w\left(K_w^l - \frac{p_1^w + p_2^w}{p_w}K_w\right)
\end{aligned} \tag{2.24}$$

To maximize the energy savings, we merge the two channels with the lowest frequencies ($p_1^w + p_2^w < p_w 2/w$), and the energy savings are:

$$\mathcal{C}(RW) - \mathcal{C}'(RW) = p_w \left(K_w' - \frac{p_1^w + p_2^w}{p_w} K_w \right) \geq p_w \left(K_w' - \frac{2}{w} K_w \right) \quad (2.25)$$

Eq. 2.25 represents the trade-off between more complex registers and more complex buses. From a previous measurement we know that $K_w' \approx K_w$, and therefore this transformation improves the energy cost as long as $w > 2$. The same transformation can be used iteratively as long as the energy improvement is positive.

If all ports have the same frequency, we can build those ports as a tree of multiplexors and de-multiplexors. This results in an average and a worst-case energy cost logarithmic in the number of read ports for the read actions, and logarithmic in the number of write ports for the write actions.

The global optimization problem of channel assignment is more complex. Local optimization per variable does not give a global optimum, and other considerations, such as the complexity of the resulting wiring problem or the introduction of extra intermediate variables, may have a strong impact on energy cost. Global optimization is beyond the scope of this section.

2.2.3 Function Evaluation

Function evaluation can hide part of the computation executed by the program. To incorporate that cost into the energy model, we have to make the evaluation of that function explicit in the CSP specification, or otherwise use a worst-case cost for the evaluation of an arbitrary boolean function.

Given the program:

$$\ldots; F!f(x); \ldots$$

we want to express the cost of the evaluation of $f(x)$. To estimate the worst-case cost we give a specific implementation for f and calculate the cost of that implementation based on the energy model described so far. This way we know that the cost of evaluating a function is consistent with the rest of the model.

If the range of x is $\{\mathrm{x}_1, \ldots, \mathrm{x}_n\}$, and $f(\mathrm{x}_i) = f_i$, we can express the function evaluation as:

```
...; [⟨ i : 1..n :  x = x_i ⟶ F!f_i⟩]; ...
```

The cost of this program scales with n, which can be a large number. To obtain a more efficient implementation, we encode x as an array of $N = \lceil \log_2 n \rceil$ bits, and eliminate one bit from the function evaluation by currying:

```
...;
[   x[0] ⟶ X_t!x[1..N − 1];  F!(F_t?)
[] ¬x[0] ⟶ X_f!x[1..N − 1];  F!(F_f?)
]; ...

|| *[X_t?y;  F_t!f_t(y)]
|| *[X_f?y;  F_f!f_f(y)]
```

From the previous decomposition we see that the cost of evaluating a function of N bits, $C_f(N)$, is, at worst, the cost of communicating $N-1$ bits (X_{tf} channel), $K_c(N-1)$) plus the cost of evaluating an $N-1$ bit function, $C_f(N-1)$ plus the cost of merging the F_t and F_f channels, K_M:

$$\mathcal{C}_f(N) = K_c(N-1) + \mathcal{C}_f(N-1) + K_M \tag{2.26}$$

Solving for $C_f(N)$,

$$\mathcal{C}_f(N) = K_c\frac{N(N-1)}{2} + \mathcal{C}_f(0) + NK_M \tag{2.27}$$

This equation can be further refined. If the range of the function f has m different values that can be expressed as an array of $M = \lceil \log_2 m \rceil$ bits, the cost of merging F_t and F_f can be expressed as $K_M = MK_m$, and we have:

$$\mathcal{C}_f(N) = K_c\frac{N(N-1)}{2} + \mathcal{C}_f(0) + NMK_m \tag{2.28}$$

In general, the cost of evaluating a function of N inputs and M outputs, $\mathcal{C}_f(N, M)$ can be expressed as:

$$\mathcal{C}_f(N, M) \approx K_1N^2 + K_2NM \tag{2.29}$$

2.3 Example: Counter

In this section we develop an example — a self incrementing register — and show how to predict the energy consumption of the incrementer using the above energy model.

The following CSP process will serve as the specification for the counter:

$$
\begin{array}{lll}
PC \equiv *[[& \overline{I} \longrightarrow & x := x + 1; I \\
 & [] \ \overline{R} \longrightarrow & x := 0; R \\
 & [] \ \overline{L} \longrightarrow & L?x \\
 & [] \ \overline{S} \longrightarrow & S!x \\
 &]] &
\end{array}
$$

In most applications, the environment would be responsible for ensuring the mutual exclusion between the I, R, L, and S channels. We can rewrite PC as:

$$
\begin{array}{ll}
PC \equiv (& *[[\ \overline{I} \longrightarrow \ x := x + 1; I \ ]] \\
 & \| \ *[[\ \overline{R} \longrightarrow \ x := 0; R \ ]] \\
 & \| \ *[[\ \overline{L} \longrightarrow \ L?x \ ]] \\
 & \| \ *[[\ \overline{S} \longrightarrow \ S!x \ ]] \\
 &)
\end{array}
$$

To improve the performance of the incrementer, we separate the I communication in two parts:

$$INC \equiv *[[\ \overline{Iu} \longrightarrow y := x + 1, \ \ Iu; \ \ [\overline{Id}]; \ \ x := y; \ \ Id \ ]]$$

The value of x is stable until Id is started, and stable again after Id is completed. Iu can execute concurrently with S.

Notice that when x is even, it is trivial to calculate $x + 1$. We separate process INC into two parts, one that takes care of the least significant bit of x, and one that takes care of the rest:

$$INC \equiv (\ INC0 \ \| \ INCR)$$

$$
\begin{array}{lll}
INC0 \equiv *[[& \overline{Iu} \ \wedge \ \neg x_0 \ \longrightarrow & Iu; \ \ [\overline{Id}]; \ \ x_0\uparrow; \ \ Id \\
 & [] \ \overline{Iu} \ \wedge \ x_0 \ \longrightarrow & I1u, \ Iu; \ \ [\overline{Id}]; \ \ x_0\downarrow, \ I1d; \ \ Id \\
 &]] &
\end{array}
$$

$$INCR \equiv *[[\ \overline{I1u} \longrightarrow y1 := x1 + 1, \ \ I1u; \ \ [\overline{I1d}]; \ \ x1 := y1; \ \ I1d \ ]]$$

$INC0$ is a one-bit incrementer, and $INCR$ is an $n-1$ bit incrementer. We apply to $INCR$ the same transformation, until we end up with one-bit counters only.

2.3.1 Handshaking Expansion, Production Rules

The following is a handshaking expansion for the program *INC0*. We rename the channel *Iu*, *Id* as *I*, and the channel *I1u*, *I1d* as *O*.

$$\begin{array}{llll} INC \equiv *[[& Ii \wedge \neg x & \longrightarrow & Io\uparrow;\ [\neg Ii];\ x\uparrow;\ Io\downarrow \\ & [] \ Ii \wedge x & \longrightarrow & Oo\uparrow,\ Io\uparrow;\ [\neg Ii];\ x\downarrow;\ [Oi]; \\ & & & Oo\downarrow;\ [\neg Oi];\ Io\downarrow \\ &]] & & \end{array}$$

To generate the production rules, we change the comma into a semicolon, and add the state variable s:

$$\begin{array}{llll} INC \equiv *[[& Ii \wedge \neg x & \longrightarrow & s\uparrow;\ Io\uparrow;\ [\neg Ii];\ x\uparrow;\ s\downarrow;\ Io\downarrow \\ & [] \ Ii \wedge x & \longrightarrow & Oo\uparrow;\ Io\uparrow;\ [\neg Ii];\ x\downarrow;\ [Oi]; \\ & & & Oo\downarrow;\ [\neg Oi];\ Io\downarrow \\ &]] & & \end{array}$$

We derive the production rules for this handshaking expansion:

$$\begin{array}{lllllll} I_i \wedge \neg x & \to & s\uparrow & \quad & O_o \vee s & \to & I_o\uparrow \\ x & \to & s\downarrow & & \neg O_i \wedge \neg O_o \wedge \neg s & \to & I_o\downarrow \end{array}$$

$$\begin{array}{lllllll} I_i \wedge x & \to & O_o\uparrow & \quad & \neg I_i \wedge s & \to & x\uparrow \\ O_i \wedge \neg x & \to & O_o\downarrow & & \neg I_i \wedge O_o & \to & x\downarrow \end{array}$$

If x is implemented as cross-coupled inverters, the previous production rules are directly implementable.

2.3.2 Average Energy and Latency

We will derive the average energy cost per increment of the counter, under the assumption that all register values are equally probable. Therefore, half of the time the register contains an even number, and only the first bit of the counter is utilized:

$$E_n = E_1 + \frac{1}{2}E_{n-1} \tag{2.30}$$

where E_j is the average energy dissipated by a j-bit counter.

From the characteristic equation, E_n has the form $A2^{-n} + B$. Substituting in the above expression, we derive A and B:

$$E_n = (2 - \frac{1}{2^{n-1}})E_1 < 2E_1 \tag{2.31}$$

The average energy dissipation can be bounded by a constant, irrespective of the word size.

A similar computation can be done for the average latency of this circuit. Again, assuming that x is even half of the time, we get the following equation:

$$L_n = L_1 + \frac{1}{2} L_{n-1} \tag{2.32}$$

This is the same equation as Eq. 2.30. The average latency is therefore bounded by two times the latency of one stage, no matter what the word size is.

Worst-case delay and energy are linear in the word size, of course, but this is not worse than a simple incrementer design with no carry-tree.

To finish this computation, we calculate the value of E_1 based on the previously defined energy model, and then compare it with a SPICE simulation of the circuit.

We can compute E_1 as the energy cost of the I communication. The I channel is one-to-one, and the target, register x, has 3 write ports (I, R, and L) and two read ports (S, and guard evaluation). The value of the register is changed in all I communications. We can approximate E_1 by:

$$E_1 = (K_r + K_w + K_s) + (2K'_r + 3K'_w + K'_s) + K_c \tag{2.33}$$

In this case selection is implemented with a shared variable with two read and two write ports, and we can write $K_c = 2K'_r + 2K'_w + K'_s$. Using the constants previously computed, we get $E_1 \approx 36 + 179 + 156 = 371$ fF. This number can be measured directly on a SPICE simulation of an implementation of the incrementer based on the same register circuits, and we get $E_1 = 350$fF. The difference is due to an optimization in the implementation of the shared variable where only one of the branches of the choice statement needs to read the variable (see in the handshaking expansion, variable s). In this case, $E_1 \approx 36 + 179 + 134 = 349$.

This example shows that the energy cost for CSP programs can be predicted with a good measure of accuracy, based on some fairly general assumptions about the implementation methodology.

2.4 Example: Memory Array

In this section we examine the design of a memory array from the energy-efficiency point of view. The energy model previously developed is used to express the architectural trade-offs in the partitioning of such an array [32].

In CSP, a memory is an array, and reading from memory is one of the two operations: $x := M[a]$ or $X!M[a]$; writing to memory is one of the two operations $M[a] := y$ or $Y?M[a]$, where $M[a]$ is an array of n words of b bits. A program that describes a memory array with one read and one write port is:

$$
\begin{array}{l}
MEM \equiv *[[\ \overline{R} \longrightarrow \ A?a;\ R!M[a] \\
\qquad\qquad\ \ [\!]\ \overline{W} \longrightarrow \ A?a;\ W?M[a] \\
\qquad\qquad\ \ ]]
\end{array}
$$

The indexing $M[a]$ is removed by breaking up the memory array into a decoder and an array of registers:

$$
\begin{array}{l}
DECODER \equiv \langle \|\ i : 0..n-1 : \\
\qquad\qquad *[[\ \overline{R} \wedge \left(\overline{A?} = i\right) \longrightarrow \ A?;\ R_i! \\
\qquad\qquad\quad [\!]\ \overline{W} \wedge \left(\overline{A?} = i\right) \longrightarrow \ A?;\ W_i! \\
\qquad\qquad\quad]] \\
\qquad\qquad \rangle \\
ARRAY \equiv \langle \|\ i : 0..n-1 : \\
\qquad\qquad *[[\ \overline{R_i} \longrightarrow R!x_i \bullet R_i\ ]] \\
\qquad\quad \|\ *[[\ \overline{W_i} \longrightarrow W?x_i \bullet W_i\ ]] \\
\qquad\quad \rangle
\end{array}
$$

To read one word from the array, we have to execute an A communication (one sender, n receivers, $\log_2 n$ bits wide), an R_i communication (one sender, one receiver, data-less), and an R communication (n senders, one receiver, b bits wide). The costs of these operations are summarized in table 2.2. The energy cost of reading one word is the sum of the energy costs of executing each of these communications, that is:

$$E_{1D,R}(n,b) = K_A n \log_2 n + K_{R_i} + K_R nb \tag{2.34}$$

where K_A, K_{R_j}, and K_R are geometry dependent proportionality constants.

Channel	Type	Width	Cost
A	1-to-n	$\log_2 n$	$K_A n \log_2 n$
R	n-to-1	b	$K_R nb$
W	1-to-n	b	$K_W nb$

Table 2.2: Cost of the communications involved in accessing an $n \times b$ memory array.

A one dimensional array is a viable solution only for small arrays; for large n, the energy cost scales like $n \log_2 n$. One way of improving on this cost is by mapping the one-dimensional array into a two-dimensional array. We represent the double indexing by splitting the address in two:

$$
\begin{aligned}
&\mathit{MEM} \equiv *[[\ \overline{R} \longrightarrow\ A?(a_w, a_l);\ R!M[a_l][a_h] \\
&\qquad\quad [\!]\ \overline{W} \longrightarrow\ A?(a_w, a_l);\ W?M[a_l][a_h] \\
&\qquad\quad]]
\end{aligned}
$$

The first indexing is removed by extracting a row decoder:

$$
\begin{aligned}
&\mathit{DEC} \equiv \langle \|\ i : \mathit{0..l-1} : \\
&\qquad *[[\ \left(\overline{Al?} = i\right) \longrightarrow\ Al? \,\|\, S_i!\ ]] \\
&\quad \rangle
\end{aligned}
$$

The second indexing is removed by extracting a column decoder:

$$
\begin{aligned}
&\mathit{MUX} \equiv \langle \|\ j : \mathit{0..w-1} : \\
&\qquad *[[\ \overline{R} \wedge \left(\overline{Aw?} = j\right) \longrightarrow\ Aw? \,\|\, R!(R_j?) \\
&\qquad\quad [\!]\ \overline{W} \wedge \left(\overline{Aw?} = j\right) \longrightarrow\ Aw? \,\|\, W_j!(W?) \\
&\qquad\quad]] \\
&\quad \rangle
\end{aligned}
$$

where l and w are such that $l \times w = n$. Finally, the register processes:

$$
\begin{aligned}
&\mathit{ARRAY} \equiv \langle \|\ i : \mathit{0..l-1} : j : \mathit{0..w-1} : \\
&\qquad\qquad *[[\ \overline{S_i} \wedge \overline{R_j} \longrightarrow R_j!x_{ij} \bullet S_i\ ]] \\
&\qquad\quad \|\ *[[\ \overline{S_i} \wedge \overline{W_j} \longrightarrow W_j?x_{ij} \bullet S_i\ ]] \\
&\quad \rangle
\end{aligned}
$$

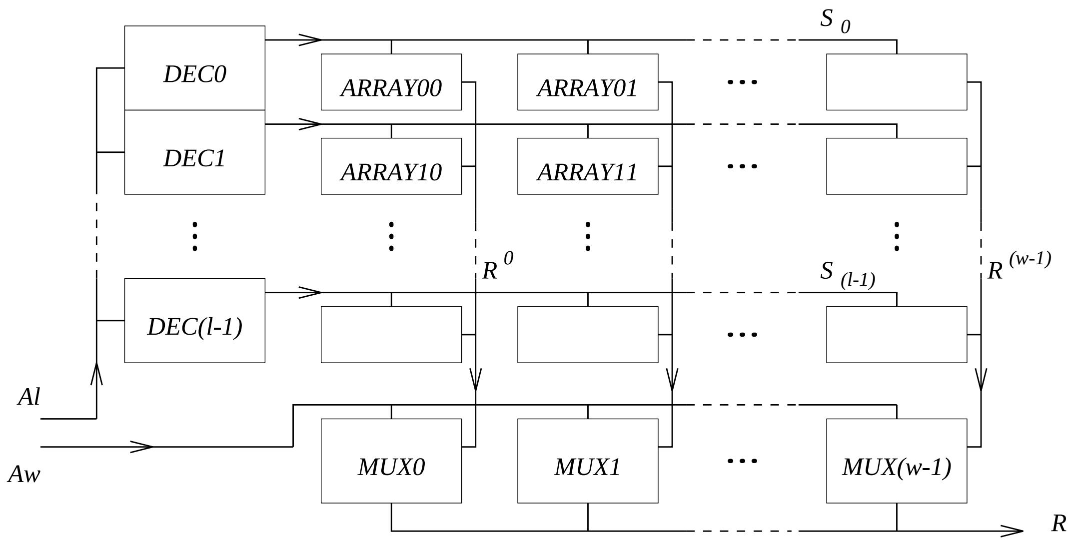


Figure 2.4: Process decomposition of MEM as a two-dimensional array. Only the channels corresponding to a read operation are shown.

The process decomposition and channel interconnection are shown in Fig. 2.4.

In the following section, we show how to choose l and w from a simple energy model.

2.4.1 Energy Model and Optimization

The energy cost of accessing one element of the array is calculated as the sum of the costs of the communications executed by the DEC, MUX, and $ARRAY$ processes. A read from memory requires executing communication R (w senders, one receiver, b bits wide), communication Al (one sender, l receivers, $\log_2 l$ bits wide), communication Aw (one sender, w receivers, $\log_2 w$ bits wide), communication S_i (one sender, w receivers), and communication R_j (l senders, one receiver, b bits wide). Table 2.3 summarizes the energy costs for all of the communication actions in MEM.

The total energy cost E_r of reading a memory location is:

$$E_{2D,R}(n,b) = K_{Al} l \log_2 l + K_{Aw} w \log_2 w + K_S w + K_{R_j} lb + K_R wb \quad (2.35)$$

Channel	Type	Width	Cost
Al	1-to-l	$\log_2 l$	$K_{Al} l \log_2 l$
Aw	1-to-w	$\log_2 w$	$K_{Aw} w \log_2 w$
S_i	1-to-w	dataless	$K_S w$
R_j	l-to-1	b	$K_{R_j} lb$
R	w-to-1	b	$K_R wb$
W_j	1-to-l	b	$K_{W_j} lb$
W	1-to-w	b	$K_W wb$

Table 2.3: Cost of the communications involved in accessing an $l \times w \times b$ memory array.

We simplify (2.35) by assuming all constant equal to one. This approximation is acceptable for most technologies; if a more accurate model is needed, the parameters can be calculated from the layout, and the optimization is done with those values of the parameters.

$$E_{2D,R}(n, b) = l \log_2 l + w \log_2 w + (w + l)b + w \tag{2.36}$$

We minimize $E_{2D,R}$ with respect to l and w under the constraint $l \times w = n$, using Lagrange multipliers:

$$U = l \log_2 l + w \log_2 w + (w + l)b + w + \lambda(n - lw) \tag{2.37}$$

We take derivatives with respect to l, w, and λ:

$$\frac{\partial U}{\partial l} = \log_2 l + \log_2 e + b - \lambda w \tag{2.38}$$

$$\frac{\partial U}{\partial w} = \log_2 w + \log_2 e + b + 1 - \lambda l \tag{2.39}$$

$$\frac{\partial U}{\partial \lambda} = n - lw \tag{2.40}$$

Assuming that $\frac{1}{2} \log_2 n + b + \log_2 e \gg 1$, we solve for l, w, and λ, and $l_{opt} =$

$w_{opt} = \sqrt{n}$. The optimum energy per access, $E_{2D,R}(n, b)$ is:

$$E_{2D,R}(n, b) = \sqrt{n}\,(\log_2 n + 2b + 1) \tag{2.41}$$

Memory designed for speed usually has $l = b \times w$ [7]. A completely square bit-array optimizes the access time per bit, but does not take into account the energy savings derived from selecting only the bits that are part of the desired word. This extra selection step takes time and area, and saves energy.

If we compare the optimal energy for a two-dimensional array with the energy used by a one-dimensional array (assuming that all constants are equal to one), we get:

$$\frac{E_{1D}}{E_{2D}} = \frac{n(\log_2 n + b + 1)}{\sqrt{n}\,(\log_2 n + 2b + 1)} \approx \sqrt{n} \tag{2.42}$$

The number of words in a memory chip is usually very large, in the order of 2^{20}, making the two dimensional arrangement far better in energy. We can, in principle, generalize this argument to multidimensional arrays, to get an even greater improvement in energy per access. This cannot be done, however, by simply increasing the number of indices in the array. The memory has to be laid-out on a two-dimensional surface; mapping a multidimensional array on this surface will make all wires much longer, and the results will not be comparable with (2.41). In the next section, we make that mapping explicit in the CSP program for the memory so that a realistic energy model can be derived from that program.

2.4.2 Multi-bank Memory Array

We can further reduce the energy per access by breaking up the memory into several sub-arrays so that only one of the smaller sub-arrays is accessed in each memory reference; this technique is also known as the divided word-line method [35]. We obtain the CSP for the multi-banked memory by applying a *divide-and-conquer* strategy to the *MEM* program.

$MEM2B \equiv \; (MERGE \;\|\; MEMO \;\|\; MEME)$

$$
\begin{array}{l}
MERGE \equiv *[[\ \overline{R} \longrightarrow\ A?a;\ [\ odd(a) \longrightarrow\ Ao!a/2;\ R!(Ro?) \\
\qquad\qquad\qquad\qquad\qquad\quad\ \ []\ even(a) \longrightarrow\ Ae!a/2;\ R!(Re?) \\
\qquad\qquad\qquad\qquad\qquad\quad\ \ ] \\
\qquad\quad\ []\ \overline{W} \longrightarrow\ A?a;\ [\ odd(a) \longrightarrow\ Ao!a/2 \parallel Wo!(W?) \\
\qquad\qquad\qquad\qquad\qquad\qquad []\ even(a) \longrightarrow\ Ae!a/2 \parallel We!(W?) \\
\qquad\qquad\qquad\qquad\qquad\qquad] \\
\qquad\quad\ ]]
\end{array}
$$

$$
\begin{array}{l}
MEMO \equiv *[[\ \overline{Ro} \longrightarrow\ Ao?a;\ Ro!MO[a] \\
\qquad\quad\ \ []\ \overline{Wo} \longrightarrow\ Ao?a;\ Wo?MO[a] \\
\qquad\quad\ \ ]]
\end{array}
$$

$$
\begin{array}{l}
MEME \equiv *[[\ \overline{Re} \longrightarrow\ Ae?a;\ Re!ME[a] \\
\qquad\quad\ \ []\ \overline{We} \longrightarrow\ Ae?a;\ We?ME[a] \\
\qquad\quad\ \ ]]
\end{array}
$$

where ME and MO are $n/2 \times b$ arrays.

To read one word from $MEM2B$, we have to execute communication A (one sender, one receiver, $\log_2 n$ bits wide) and communication R (two senders, one receiver, b bits wide), plus we have to execute either *MEMO* or *MEME*. The energy cost of reading one word from a memory of size $n \times b$ can, therefore, be expressed as:

$$E_{2B,R}(n,b) = 2K_R b + K_A \log_2 n + E_{2B,R}(n/2,b) \tag{2.43}$$

Or, for $n = 2^N$,

$$E_{2B,R}(2^N,b) = E_{2B,R}(2^{N-1},b) + 2K_R b + K_A N \tag{2.44}$$

We apply the same transformation to the sub-arrays until the indexing is completely removed. We get:

$$E_{2B,R}(2^N,b) = E_{2B,R}(1,b) + 2K_R N b + K_A \frac{N(N+1)}{2} \tag{2.45}$$

Or, in terms of n,

$$E_{2B,R}(n,b) \approx 2K_R b \log_2 n + K_A \frac{\log_2 n(\log_2 n - 1)}{2} \tag{2.46}$$

Depending on the cost of merging the results from the two sub-arrays, it may be convenient to stop the divide-and-conquer process after fewer than N steps, and implement the remaining array as a two-dimensional array. After $N-J$ divide steps, the energy cost is:

$$E_{2B,R}(2^N, b) = E_{2D,R}(2^J, b) + 2K_R(N-J)b + K_A\frac{N(N+1)-J(J+1)}{2} \tag{2.47}$$

Minimizing equation (2.47) with respect to J we obtain the optimum bank size, $2^{J_{opt}}$.

For example, if all constants are equal to 1, $N = 20$, and $b = 32$, we obtain an optimum for $J = 3$, $E_{2B,R}(2^{20}, 32) = 1493$. For $J = 20$ (no break-up in banks), the energy cost is $E_{2D,R}(2^{20}, 32) = 87040$.

2.5 Summary & Conclusion

In this chapter we have explored the causes of energy dissipation in CMOS circuits. A first-order approximation shows that most of this energy is spent in charging capacitors. Energy dissipation occurs only when some computation is being performed, and this allows us to tie the energy dissipation of the CMOS circuit to an energy-cost measure for the CSP program that specifies that circuit.

This energy-cost measure corresponds to the "energy complexity" of the corresponding program and is computed relative to a trace of the execution of the program, or to frequencies of occurrence of the different statements in a number of traces. The energy complexity reflects the sum of all transitions required to execute the trace, where each transition has been weighed according to its fan-in and fan-out.

At the CSP level, we have given an energy cost to assignments, communication actions, function computation, and synchronization. A number of technology-dependent constants have to be computed to be able to compare the relative costs of statements of different types. We have shown, with an example, how these constants can be computed in the case of an assignment to a multi-ported register.

These constants, though, are not always required to get meaningful conclusions, as was shown in the memory design example. In this example, we have presented several memory designs, and we have shown how to choose the design parameters to obtain the optimum energy cost. These results show that commercial memory designs, optimized for delay and density, can be greatly improved in energy performance.

Chapter 3

Entropy and Energy of Reactive Computations

In this chapter we present a lower bound to the attainable energy cost of a CSP program, based on the entropy of the source that generates the input/output behavior of the environment. We show, as well, how to use this lower bound to direct the synthesis procedure.

A CSP program is not only the specification of what an asynchronous circuit does, but also of *how* it does it. From the minimum energy point of view, we want to abstract the how from the specification, and given the computation performed by the circuit, find the "best" possible implementation for that computation.

A very large class of programs can be described by their input/output behavior; for these programs, it is possible to consider the set of all CSP programs that have the same input/output behavior and pick the one with the lowest energy cost. Of course, it is not practical to list all these programs. Instead, we can try to find a lower bound to the energy cost and look at the programs that get close to that lower bound.

The energy cost of a CSP program, as defined in Chapter 2, can be interpreted as the energy complexity of the program, and is similar to the time complexity of the sequential execution of that program. Using that interpretation, we can relate the energy complexity of a CSP program with the information-theoretic complexity of the sequence of input/output symbols. We show how to give a lower bound to the energy cost of a CSP program and under what conditions the lower bound is attained.

In general, a computation engine can be treated as a dynamical system. From the thermodynamic equations of this system, we can compute the energy required to perform a given computation as:

$$\Delta Q = \int_{\text{initial state}}^{\text{final state}} T dS \tag{3.1}$$

where ΔQ is the heat dissipated, T is the absolute temperature of the system, and S is the thermodynamic entropy of the system. Several models of reversible computation where the computation proceeds along an isentropic line [2,12,10], have been proposed. According to these models, any computation can be performed at an arbitrarily low energy cost, as long as the speed of forward progress is low enough (or at any speed in the ideal billiard-ball computer model). Zero energy dissipation is achieved by making each step of the computation infinitely efficient and, therefore, infinitely slow.

If the computation is to progress at any finite speed, we will have to dissipate some non-zero amount of energy, because the quasi-static requirement for the dynamical system is not met. It is in our interest, therefore, to make the computation as simple as possible, or, in the context of digital circuits, to minimize the "number" of transitions (that is, the transition-count taking into account fan-in and fan-out of operators). To this effect, we estimate the complexity of the computation by looking at the information theoretical entropy of the strings generated by that computation [5,1]. Based on this entropy, we find a lower bound to the energy complexity of any CSP program that implements that computation.

3.1 Flat CSP

A CSP process P is called *flat* if it has the form:

$$
\begin{array}{ll}
P \equiv *[[& G_0 \longrightarrow \ A_0;\ SA_0 \\
 & [\!]\ G_1 \longrightarrow \ A_1;\ SA_1 \\
 & \ldots \\
 & [\!]\ G_{n-1} \longrightarrow \ A_{n-1};\ SA_{n-1} \\
 &]]
\end{array}
$$

where the guards G_i are stable (that is, once they become true, they remain true at least until the first action of the guarded command is executed). Commands A_i are either data-less or boolean communications, and commands SA_i are constant assignments to state variables. This restriction is introduced so that all the complexity of the computation is handled by the selection mechanism, instead of the data-assignment mechanism.

3.1.1 Flattening a CSP Process

A CSP process can be transformed into flat form by applying the following rules recursively. The variables that are added when applying a rule are new; that is, they did not appear in the original program and are initially false. S_A represents a sequence of state variable assignments.

Process. The first transformation converts the whole process into a repetition, if it was not one already:

$P \triangleright *[[\ \neg s \longrightarrow\ P;\ s\uparrow\ ]]$

Sequencing. This transformation removes sequential composition. Other state assignments are possible to enforce sequencing.

$\square\ G \longrightarrow\ A_0; A_1; \ldots; A_n;\ S_A \triangleright$

$$\begin{array}{l}
\square\ \neg s_G \wedge G \longrightarrow\ s_{A_0}\uparrow; s_G\uparrow \\
\square\ s_G \wedge s_{A_0} \longrightarrow\ A_0; s_{A_0}\downarrow; s_{A_1}\uparrow \\
\ldots \\
\square\ s_G \wedge s_{A_n} \longrightarrow\ A_n; s_{A_n}\downarrow; s_G\downarrow; S_A
\end{array}$$

Choice. This transformation removes choice composition.

$\square\ G \longrightarrow\ [G_0 \longrightarrow A_0 \square \ldots \square G_n \longrightarrow A_n];\ S_A \triangleright$

$$\begin{array}{l}
\square\ \neg s_G \wedge G \longrightarrow\ s_G\uparrow \\
\square\ s_G \wedge G_0 \longrightarrow\ A_0;\ s_G\downarrow; S_A \\
\square\ s_G \wedge G_1 \longrightarrow\ A_1;\ s_G\downarrow; S_A \\
\ldots \\
\square\ s_G \wedge G_n \longrightarrow\ A_n;\ s_G\downarrow; S_A
\end{array}$$

Repetition. This transformation removes repetition.

$$\Box\ G \longrightarrow\ *[G_0 \longrightarrow A_0 \Box \ldots \Box G_n \longrightarrow A_n];\ S_A \triangleright$$

$$\begin{array}{l}
\Box\ \neg s_G \wedge G \longrightarrow\ s_G\uparrow \\
\Box\ s_G \wedge G_0 \longrightarrow\ A_0 \\
\Box\ s_G \wedge G_1 \longrightarrow\ A_1 \\
\ldots \\
\Box\ s_G \wedge G_n \longrightarrow\ A_n \\
\Box\ s_G \wedge \neg G_0 \wedge \ldots \wedge \neg G_n \longrightarrow\ s_G\downarrow;\ S_A
\end{array}$$

Bitwise Decomposition. This transformation removes concurrency between assignments to different bits on the same word, different bits on the same channel, etc.

$$\Box\ G \longrightarrow\ A_0, A_1, \ldots, A_n;\ S_A \triangleright$$

$$\begin{array}{l}
\Box\ \neg s_G \wedge G \longrightarrow\ s_G\uparrow \\
\Box\ s_G \wedge \neg s_{A_0} \longrightarrow\ A_0; s_{A_0}\uparrow \\
\Box\ s_G \wedge \neg s_{A_1} \longrightarrow\ A_1; s_{A_1}\uparrow \\
\ldots \\
\Box\ s_G \wedge \neg s_{A_n} \longrightarrow\ A_n; s_{A_n}\uparrow \\
\Box\ s_G \wedge s_{A_0} \wedge \ldots \wedge s_{A_n} \longrightarrow s_{A_0}\downarrow; \ldots; s_G\downarrow; S_A
\end{array}$$

3.1.2 Flat Process Decomposition

A single flat process is an inefficient implementation of a CSP program; all the guards have to be evaluated every time a command is executed. Hierarchical evaluation of guards can improve the average cost per command by making the commands most frequently executed cheaper at the expense of the more infrequent ones.

Flat process decomposition transforms one flat process into two, in the following way:

$$*[[G_0 \longrightarrow A_0; SA_0 [] G_1 \longrightarrow A_1; SA_1 \cdots [] G_{n-1} \longrightarrow A_{n-1}; SA_{n-1}]] \triangleright$$

$$\begin{array}{l}
\quad *[[\ G_0 \vee G_1 \vee \ldots \vee G_{j-1} \longrightarrow \ H \\
\quad\ \ [] \ G_j \longrightarrow \ A_j; \ SA_j \\
\quad\ \ \ldots \\
\quad\ \ [] \ G_{n-1} \longrightarrow \ A_{n-1}; \ SA_{n-1} \\
\quad\ \ ]] \\
\| *[[\ \overline{H} \wedge \neg G_0 \wedge \neg G_1 \wedge \ldots \wedge \neg G_{j-1} \longrightarrow \ H \\
\quad\ \ [] \ \overline{H} \wedge G_0 \longrightarrow \ A_0; \ SA_0 \\
\quad\ \ \ldots \\
\quad\ \ [] \ \overline{H} \wedge G_{j-1} \longrightarrow \ A_{j-1}; \ SA_{j-1} \\
\quad\ \ ]]
\end{array}$$

where H is a new channel.

The two resulting processes are also flat, and we can re-apply the procedure to each of them to obtain a tree of flat subprocesses.

3.2 Energy and Entropy

In this section we define the energy complexity of the hierarchical decomposition of a flat process and relate this energy complexity to the information-theoretic entropy of the sequence of input/output symbols.

Hierarchical decomposition of a flat process can drastically reduce the average energy cost of executing a CSP process. Two mechanisms are at play: First, we can choose the decomposition so that the more frequently executed commands are higher up in the tree, and, second, the state of the hierarchical tree of processes stores information about the history of the computation, modifying the cost of each command according to this history.

For example, consider a program that executes commands a_1, a_2, a_3, a_4, with $\Pr(a_1) = 1/2$, $\Pr(a_2) = 1/4$, $\Pr(a_3) = 1/8$, and $\Pr(a_4) = 1/8$. The flat representation of this program is:

$$\begin{array}{l}
E \equiv *[[\ G_1 \longrightarrow a_1 \\
\qquad\ \ [] \ G_2 \longrightarrow a_2 \\
\qquad\ \ [] \ G_3 \longrightarrow a_3 \\
\qquad\ \ [] \ G_4 \longrightarrow a_4 \\
\qquad\ ]]
\end{array}$$

The average cost per command for this program is:

$$\mathcal{C}(E) = \sum_{i=1}^{4} \Pr(a_i) \log_2 4 = 2 \tag{3.2}$$

We can rewrite this program in the following way:

$$\begin{array}{llllll}
EH \equiv *[[& G_1 \longrightarrow a_1 & & & & \\
& [] \; \neg G_1 \longrightarrow [& G_2 \longrightarrow a_2 & & & \\
& & [] \; \neg G_2 \longrightarrow [& G_3 \longrightarrow a_3 & & \\
& & & [] \; G_4 \longrightarrow a_4 & & \\
]] & &] &] & &
\end{array}$$

The average cost per command of this program can be computed as:

$$\begin{aligned}
\mathcal{C}(EH) &= \Pr(a_1)\log_2 2 + \Pr(a_2)(\log_2 2 + \log_2 2) + \\
&\quad (\Pr(a_3) + \Pr(a_4))(\log_2 2 + \log_2 2 + \log_2 2) \\
&= 1/2 + 1/2 + 3/4 = 1.75
\end{aligned} \tag{3.3}$$

The new program has a better cost per command because the more frequently executed command has become cheaper.

Consider next the program:

$S \equiv *[\langle ; \;\; i : 1..n : \;\; a_i \;\; \rangle]$

The a_i commands all have the same frequency, and therefore the following program should be efficient:

$S \equiv *[[\langle [] \;\; i : 1..n : \;\; p_i \longrightarrow a_1; p_i\!\downarrow, p_{i+1}\!\uparrow \;\; \rangle]]$

where all p_i are initially false, except p_1, and $n + 1 = 1$.

The average cost per command of S is $\mathcal{C}(S) = \log_2 n$. There is, however, a better way of encoding S:

$$\begin{array}{lll}
SH \equiv *[& p_1 \longrightarrow a_1; p_1\!\downarrow, p_2\!\uparrow & \\
& [] \; \neg p_1 \longrightarrow *[& p_2 \longrightarrow a_2; p_2\!\downarrow, p_3\!\uparrow \\
& & [] \; \neg p_2 \wedge \neg p_1 \longrightarrow *[\; \ldots \;] \\
] &] &
\end{array}$$

Now, for each a_i, we execute two guarded commands, each of cost 1; the cost per command of SH is $\mathcal{C}(SH) = 2$. The difference in cost is due to the fact that the decomposition tree stored the previous history of the computation, so at any time the non-relevant choices have been discarded.

The previous observations can be formalized. Given a hierarchical decomposition of a flat process, an execution of that process corresponds to a path in the tree of subprocesses. This path can be encoded by giving the sequence of cardinals of the guarded command selected within each subprocess. With this sequence and the program text, we can reconstruct the computation without needing to know the sequence of input symbols and reproduce the sequence of commands that would have been executed by the original process.

For example, consider the program:

$$
\begin{array}{l}
P \equiv *[[\ G_1 \longrightarrow\ H_1 \\
\qquad\quad [\!]\ G_2 \longrightarrow\ A_1 \\
\qquad\quad [\!]\ G_3 \longrightarrow\ A_2 \\
\qquad\ ]] \\
\qquad \| \\
\qquad *[[\ \overline{H_1} \wedge G_4 \longrightarrow\ A_3 \\
\qquad\quad [\!]\ \overline{H_1} \wedge G_5 \longrightarrow\ H_1 \\
\qquad\ ]]
\end{array}
$$

where the A_i's are input/output symbols. Then the cardinal sequence 1, 1, 2, 2, 3 corresponds to the sequence A_3, A_1, A_2; the cardinal sequence 1, 2, 2, 2, 3 corresponds to the sequence A_1, A_1, A_2; etc.

As was shown in Chapter 2, the cost of executing a guarded command from a one-of-n selection scales with $\log_2 n$. To simplify the notation and the proofs on this chapter, we use $\lceil \log_2 n \rceil$[1] as the energy complexity of one-to-n selection.

To formalize, let P be a process, FP be the flat representation of that process, HP be a hierarchical decomposition of FP, $a_{1..m} = a_1, \ldots, a_m$ be the first m commands executed by process P. Let $s_1, \ldots, s_{k(\mathrm{HP}, a_{1..m})}$ be the sequence of cardinals of the selected guarded commands required to reconstruct $a_{1..m}$ from HP, and $k(\mathrm{HP}, a_{1..m})$ be the length of that sequence. This sequence can be encoded as a list of $l(\mathrm{HP}, a_{1..m})$ bits, $K_1, \ldots, K_{l(\mathrm{HP}, a_{1..m})}$. Finally, let $\mathcal{C}(\mathrm{HP}, a_{1..m})$ be the cost of running process HP until $a_{1..m}$ has been executed.

[1] $\lceil x \rceil$ represents the ceiling function of x, the smallest integer that is at least x.

We can relate the energy cost to the length of the code with the following theorem:

Theorem 3.1 *$\mathcal{C}(\mathrm{HP}, a_{1..m}) = l(\mathrm{HP}, a_{1..m})$.*

Proof: Given n_i, the number of guarded commands in the i^{th} selection and $\lceil \log_2 n_i \rceil$ the cost of that selection, we have:

$$\mathcal{C}(\mathrm{HP}, a_{1..m}) = \sum_{i=1}^{k(\mathbf{HP}, a_{1..m})} \lceil \log_2 n_i \rceil \tag{3.4}$$

We need $\lceil \log_2 n_i \rceil$ bits to encode the i^{th} cardinal; therefore,

$$l(\mathrm{HP}, a_{1..m}) = \sum_{i=1}^{k(\mathbf{HP}, a_{1..m})} \lceil \log_2 n_i \rceil = \mathcal{C}(\mathrm{HP}, a_{1..m}) \quad \blacksquare \tag{3.5}$$

From Theorem 3.1 we conclude that to optimize the energy cost of a CSP process we have to find the HP that best encodes the command sequence $a_1, \ldots, a_m$. First we compute a lower bound of the optimum encoding, using some results from information theory.

Let A_i be a random variable that takes as value the i^{th} command executed by process P. The command sequences of length m have a probability distribution $\Pr(A_{1..m})$, which can be calculated either deterministically (for example, assuming that all input sequences are possible and equiprobable), or statistically, by looking at actual traces of the execution of the program. Let S_m be the set of all command sequences with non-zero probability.

To define the cost per command, we take an average of the energy cost of the process over a very large number of commands. The limit of the average cost when the number of commands goes to infinity may not exist, or be unbounded (as would be the case in a busy-waiting loop). To avoid those problems, we use the lim sup [2] in the following definition:

[2]

$$\limsup_{n \to +\infty} a_n = \lim_{n \to +\infty} \sup_{i > n} (a_i)$$

Definition 3.1 *The cost per command of a process* HP, $\mathcal{C}(\mathrm{HP})$, *is defined as:*

$$\mathcal{C}(\mathrm{HP}) = \limsup_{m \to +\infty} \sum_{(a_{1..m}) \in S_m} \Pr(a_1, \ldots, a_m) \frac{1}{m} \mathcal{C}(\mathrm{HP}, a_{1..m}) \tag{3.6}$$

The following theorem gives a sufficient, though not necessary, condition for the convergence of this limit:

Theorem 3.2 *If every loop of HP (that is, every sequence of commands from HP that has the same initial state and final state) contains at least one command from P, then* $\mathcal{C}(\mathrm{HP})$ *converges.*

Proof: Let K be the length of the longest loop, counted as the number of selections made in that loop, and n be the number of commands in P. Then we can write $k(\mathrm{HP}, a_{1..m}) \leq K \times m$; therefore:

$$\mathcal{C}(\mathrm{HP}, a_{1..m}) = \sum_{i=1}^{k(\mathrm{HP}, a_{1..m})} \lceil \log_2 n_i \rceil \leq K \times m \lceil \log_2 n \rceil \tag{3.7}$$

Therefore,

$$\begin{aligned} \mathcal{C}(\mathrm{HP}) &= \limsup_{m \to +\infty} \sum_{(a_{1..m}) \in S_m} \Pr(a_1, \ldots, a_m) \frac{1}{m} \mathcal{C}(\mathrm{HP}, a_{1..m}) \\ &\leq \limsup_{m \to +\infty} \sum_{(a_{1..m}) \in S_m} \Pr(a_1, \ldots, a_m) K \lceil \log_2 n \rceil \\ &= K \lceil \log_2 n \rceil \end{aligned} \tag{3.8}$$

The lim sup is bounded and, therefore, converges. ∎

The entropy of the command sequences of length m, $\mathcal{H}(A_1, \ldots, A_m)$, is defined in the usual way [28]:

Definition 3.2 *The entropy of a sequence* $(A_1, \ldots, A_m)$ *of random variables,* $\mathcal{H}(A_1, \ldots, A_m)$, *is defined as:*

$$\mathcal{H}(A_1, \ldots, A_m) = \sum_{(a_{1..m}) \in S_m} \Pr(a_1, \ldots, a_m) \log_2 \frac{1}{\Pr(a_1, \ldots, a_m)} \tag{3.9}$$

We use the following theorem to define the entropy of a process P:

Theorem 3.3 *The limit,*

$$\limsup_{m\to+\infty} \frac{1}{m}\mathcal{H}(A_1,\ldots,A_m)$$

always exists.

Proof: If A_i can take n different values, we have

$$0 \le \frac{1}{m}\mathcal{H}(A_1,\ldots,A_m) \le \frac{1}{m}\left(\mathcal{H}(A_1)+\cdots+\mathcal{H}(A_m)\right) \le \log_2 n \tag{3.10}$$

The sequence is bounded by a constant; therefore, the lim sup exists. ∎

Definition 3.3 *The entropy of a process* P*,* $\mathcal{H}(\mathrm{P})$*, is defined as:*

$$\mathcal{H}(\mathrm{P}) = \limsup_{m\to+\infty} \frac{1}{m}\mathcal{H}(A_1,\ldots,A_m) \tag{3.11}$$

Now we are ready to prove the basic theorem that gives us a lower bound to the energy complexity of a hierarchical decomposition of a process P:

Theorem 3.4 *For every process* P*, and every hierarchical decomposition* HP *of* P*, we have:*

$$\mathcal{H}(\mathrm{P}) \le \mathcal{C}(\mathrm{HP}) \tag{3.12}$$

Proof: $K_1,\ldots,K_{l(\mathrm{HP},m)}$ is a prefix-code[3] for $A_1,\ldots,A_m$; therefore we know that the average length of the code is at least the entropy of the source of symbols [28], and we can write:

$$\begin{aligned}\mathcal{H}(A_1,\ldots,A_m) &\le \sum \Pr(a_1,\ldots,a_m)\, l(\mathrm{HP},a_{1..m}) \\ &= \sum \Pr(a_1,\ldots,a_m)\,\mathcal{C}(\mathrm{HP},a_{1..m})\end{aligned} \tag{3.13}$$

Dividing by m and taking lim sup on both sides of the inequality, we get the thesis. ∎

Theorem 3.4 gives a lower bound to the energy cost of a hierarchical decomposition. The next question to be answered is under what conditions the lower bound can be reached. The following theorem gives a partial answer:

[3] A prefix-code is a code such that no codeword is a prefix of another codeword.

Theorem 3.5 *If for every sequence of commands $a_1, \ldots, a_i$ executed by a hierarchical decomposition* HP *of a process* P*, we have the following conditions:*

1. $\Pr(s_{k(\mathrm{HP},a_{1..i})} | s_1, \ldots, s_{k(\mathrm{HP},a_{1..i})-1}) = \frac{1}{n_{k(\mathrm{HP},a_{1..i})}}$
2. $\Pr(s_1, \ldots, s_{k(\mathrm{HP},a_{1..i})}) = \Pr(a_1, \ldots, a_i)$
3. $k(\mathrm{HP}, a_{1..i}) < K \times i$

where K is a constant, then $\mathcal{H}(\mathrm{P}) \leq \mathcal{C}(\mathrm{HP}) \leq \mathcal{H}(\mathrm{P}) + K$ holds.

Theorem 3.5 can be interpreted as follows. The first condition means that all choices in a computation of HP are equally probable. The second condition is automatically verified if, for each sequence of commands from the original process P, there is a unique sequence of choices from HP. The third condition puts a fixed bound to the overhead introduced by the hierarchical decomposition. It is satisfied if each loop contains at least one command of P. This condition excludes busy-waiting.

Proof: The cost of executing $a_{1..m}$ can be computed as:

$$\begin{aligned}
\mathcal{C}(\mathrm{HP}, a_{1..m}) &= \sum_{j=1}^{k(\mathrm{HP},a_{1..m})} \lceil \log_2 n_j \rceil \\
&= \sum_{j=1}^{k(\mathrm{HP},a_{1..m})} \left\lceil \log_2 \frac{1}{\Pr(s_j | s_1, \ldots, s_{j-1})} \right\rceil \\
&< k(\mathrm{HP}, a_{1..m}) + \sum_{j=1}^{k(\mathrm{HP},a_{1..m})} \log_2 \frac{1}{\Pr(s_j | s_1, \ldots, s_{j-1})} \\
&= k(\mathrm{HP}, a_{1..m}) + \log_2 \prod_{j=1}^{k(\mathrm{HP},a_{1..m})} \frac{1}{\Pr(s_j | s_1, \ldots, s_{j-1})} \qquad (3.14)
\end{aligned}$$

We use the formula:

$$\prod_{j=1}^{n} \Pr(x_j | x_1, \ldots, x_{j-1}) = \Pr(x_1, \ldots, x_n) \qquad (3.15)$$

and Eq. 3.14 becomes:

$$\begin{aligned}
\mathcal{C}(\mathrm{HP}, a_{1..m}) &< k(\mathrm{HP}, a_{1..m}) + \log_2 \frac{1}{\Pr(s_1, \ldots, s_{k(\mathrm{HP},a_{1..m})})} \\
&= k(\mathrm{HP}, a_{1..m}) + \log_2 \frac{1}{\Pr(a_1, \ldots, a_m)}
\end{aligned}$$

$$< \quad K \times m + \log_2 \frac{1}{\Pr(a_1, \ldots, a_m)} \tag{3.16}$$

We compute next $\mathcal{C}(\mathrm{HP})$:

$$\begin{aligned}
\mathcal{C}(\mathrm{HP}) &= \limsup_{m \to +\infty} \sum_{(a_{1..m}) \in S_m} \Pr(a_1, \ldots, a_m) \frac{1}{m} \mathcal{C}(\mathrm{HP}, a_{1..m}) \\
&\leq \limsup_{m \to +\infty} \sum_{(a_{1..m}) \in S_m} \Pr(a_1, \ldots, a_m) \frac{1}{m} \left(K \times m + \log_2 \frac{1}{\Pr(a_1, \ldots, a_m)} \right) \\
&= \limsup_{m \to +\infty} \left(K + \frac{1}{m} \sum_{(a_{1..m}) \in S_m} \Pr(a_1, \ldots, a_m) \log_2 \frac{1}{\Pr(a_1, \ldots, a_m)} \right) \\
&= K + \mathcal{H}(\mathrm{P}) \ \blacksquare
\end{aligned} \tag{3.17}$$

The entropy $\mathcal{H}(\mathrm{P})$ of a program P was defined based on the entropy of the sequences of commands from the original program P. We can restrict this definition to the input symbols or the output symbols, exclusively, and all the theorems proved so far hold as well.

3.3 Process Decomposition

The entropy of the input/output symbols appears in Theorem 3.4 as a lower-bound to the best achievable energy dissipation under a simplified energy model. The bound is, however, not tight. In this section we investigate how the program can be decomposed so that its energy cost comes closer to its lower bound.

3.3.1 Program Approximation

One way of simplifying the computation executed by a program is to allow that program to make "errors": On some type of inputs the program responds with an error condition, and the function has to be re-evaluated by the environment. The idea is that the program that computes the approximation of the function may have a much simpler implementation than the original program, with an energy cost closer to the lower bound. Depending on the probability distribution of the input sequence, this strategy may reduce the energy cost of the computation.

Consider a program P of the type:

$P \equiv *[\ \ I?x; O!f(x)\ \ ]$

We can replace P by:

$P \equiv\ \ P_\epsilon \parallel P'$

$$
\begin{array}{l}
P_\epsilon \equiv *[\ \ I?x; y := f_\epsilon(x); [y = \mathbf{error} \longrightarrow U!x;\ \ O!(D?) \\
\qquad\qquad\qquad\qquad\qquad\ \ \ \ [\!]\, y \neq \mathbf{error} \longrightarrow O!y \\
\qquad\ ] \qquad\qquad\qquad\qquad\ \ ]
\end{array}
$$

$P' \equiv *[\ \ U?x; D!f(x)\ \ ]$

P' looks the same as P, but the statistics of the input have changed significantly since the input of P' has been filtered by P_ϵ. Therefore, good implementations of P are not necessarily good implementations of P', and conversely.

To compute the energy cost of $P_\epsilon \| P'$, we take the average of executing only P_ϵ, in case of a hit, and executing P_ϵ and P', in case of an error:

$$\mathcal{C}(P_\epsilon \| P') = \mathcal{C}(P_\epsilon) + \epsilon\mathcal{C}(P') \tag{3.18}$$

where $\epsilon = \Pr(f_\epsilon(x) = \mathbf{error}) = \Pr(f_\epsilon(x) \neq f(x))$.

If P is a deterministic program, its input/output behavior is totally determined by the sequence of inputs; therefore, $\mathcal{H}(P)$ is the entropy of the input sequence. The same can be said for P_ϵ, and since both programs have the same input, we have:

$$\mathcal{H}(P) = \mathcal{H}(P_\epsilon) \tag{3.19}$$

$\mathcal{H}(P')$ is, in general, hard to compute because it depends heavily on how the input sequence was modified by P_ϵ, as we show in the two following examples.

First, consider the following probability distribution for the input to program P:

$$\Pr(x) = \begin{cases} \frac{1-\epsilon}{n-1} & \text{if } x \neq x_\epsilon \\ \epsilon & \text{if } x = x_\epsilon \end{cases} \tag{3.20}$$

where x can take n different values. If P_ϵ makes an error for $x = x_\epsilon$ only, then we have $\Pr(f_\epsilon(x) = \mathbf{error}) = \epsilon$, and $\mathcal{H}(P') = 0$.

Second, consider the following probability distribution for the input to program P:

$$\Pr(x) = \begin{cases} 1-\epsilon & \text{if } x = x_0 \\ \frac{\epsilon}{n-1} & \text{if } x \neq x_0 \end{cases} \tag{3.21}$$

If P_ϵ makes an error for $x \neq x_0$ only, then we have $\Pr(f_\epsilon(x) = \mathbf{error}) = \epsilon$, and $\mathcal{H}(\mathrm{P}') = \log_2(n-1)$.

Let m be the size of the set of values of the input that generate an error:

$$m = |\{x : f_\epsilon(x) = \mathbf{error}\}| \tag{3.22}$$

Then the best we can say about $\mathcal{H}(P')$ is $\mathcal{H}(P') \leq \log_2 m$. The inefficiency of splitting P into $P_\epsilon \| P'$ — quantized as the increase in entropy of the resulting program — is not worse than $\epsilon \log_2 m$. Since m decreases when ϵ decreases, $\epsilon \log_2 m$ can be made arbitrarily small. By using an approximation of the original program, we can greatly increase the number of efficient implementations of the program, with an arbitrarily low theoretical cost in energy.

An alternative way of looking at this problem is with the source coding function. As before, S_m is the set of input sequences of m symbols.

Definition 3.4 *The source coding function for a sequence $(A_1, \ldots, A_m)$ of random variables, $\hat{\mathcal{H}}_m(\epsilon)$ is the smallest positive number $\hat{\mathcal{H}}$ such that S_m can be partitioned into two sets, S_{mT} (typical sequences) and S_{mA} (atypical sequences), with $|S_{mT}| \leq 2^{\hat{\mathcal{H}}}$ and $\Pr(S_{mA}) \leq \epsilon$.*

As before, we can take the limit for a very large number of inputs and have the following definition:

Definition 3.5 *The source coding function for a process P is defined as:*

$$\hat{\mathcal{H}}_\epsilon(P) = \limsup_{m \to +\infty} \frac{1}{m} \hat{\mathcal{H}}_m(\epsilon) \tag{3.23}$$

Using Definition 3.5, we can partition the input into typical and atypical sets, letting P_ϵ handle the typical set, and P' the atypical set. In the worst-case, P_ϵ has to make a choice out of $2^{\hat{\mathcal{H}}_\epsilon(P)}$, and P' has to make a choice out of $n - 2^{\hat{\mathcal{H}}_\epsilon(P)}$. Therefore, $\mathcal{H}(P') \leq \log_2\left(n - 2^{\hat{\mathcal{H}}_\epsilon(P)}\right)$. In the worst-case, using Eq. 3.18, we can bound the cost of $P_\epsilon \| P'$ by:

$$\mathcal{C}(P_\epsilon \| P') \geq \hat{\mathcal{H}}_\epsilon(P) + \epsilon \log_2\left(n - 2^{\hat{\mathcal{H}}_\epsilon(P)}\right) \tag{3.24}$$

We relate next the source coding function to the entropy using the following two theorems:

Theorem 3.6 (Shannon–McMillan Theorem) *Given a sequence of independent, equally distributed random variables* $(A_1, \ldots, A_n, \ldots)$*, with entropy* $\mathcal{H}(A)$ *and any* $\epsilon > 0$*, we can choose* n *large enough so that the set* S *of all possible sequences of length* n *can be partitioned into two sets,* S_A *and* S_T*, such that:*

1. $\Pr(S_A) < \epsilon$
2. *If* $(a_1, \ldots, a_n) \in S_T$*, then* $\mathcal{H}(A) - \epsilon < -\frac{1}{n}\log_2 \Pr(a_1, \ldots, a_n) < \mathcal{H}(A) + \epsilon$
3. $(1-\epsilon)2^{n(\mathcal{H}(A)-\epsilon)} \leq |S_T| \leq 2^{n(\mathcal{H}(A)+\epsilon)}$

Theorem 3.7 *If the inputs to a process* P *verify the Shannon–McMillan hypothesis, and* $\mathcal{H}(A) > \epsilon$*, then we have* $|\mathcal{H}(P) - \hat{\mathcal{H}}_\epsilon(P)| \leq \epsilon$

For some types of sources, and for small enough error rates, ϵ, the source coding function corresponds roughly to the entropy of the source.
Proof: From the Shannon–McMillan Theorem we know that there exists a sufficiently large n such that the set of valid sequences of n symbols, S_n, can be partitioned into a typical and an atypical set, S_{nT} and S_{nA}, such that $\Pr(S_{nA}) < \epsilon$, and $|S_{nT}| \leq 2^{n(\mathcal{H}(A)+\epsilon)}$. Therefore, $\hat{\mathcal{H}}_n(\epsilon) \leq n(\mathcal{H}(A) + \epsilon)$

Now, assume that $\hat{\mathcal{H}}_n(\epsilon) < n(\mathcal{H}(A) - \epsilon)$. We can encode the source by assigning codewords of length $\hat{\mathcal{H}}_n(\epsilon)$ to the typical sequences and length n to the atypical sequences. The average length of such a code is

$$\begin{aligned} \hat{\mathcal{H}}_n(\epsilon)(1-\epsilon) + n\epsilon &\leq n(\mathcal{H}(A) - \epsilon)(1-\epsilon) + n\epsilon \\ &= n(\mathcal{H}(A) - \epsilon(\mathcal{H}(A) - \epsilon)) \\ &< n\mathcal{H}(A) \end{aligned} \tag{3.25}$$

The average length of this code per symbol is less than the entropy of the source, and this is not allowed by the coding theorem; we have $\hat{\mathcal{H}}_n(\epsilon) \geq n(\mathcal{H}(A) - \epsilon)$ and, therefore, $|\hat{\mathcal{H}}_n(\epsilon) - n\mathcal{H}(A)| \leq n\epsilon$; taking the appropriate lim sup we derive the claim. ∎

Using Theorem 3.7, if ϵ is small enough and the input source is such that it verifies the hypothesis of Theorem 3.6, we can rewrite Eq. 3.24 as:

$$\mathcal{C}(P_\epsilon \| P') \geq \hat{\mathcal{H}}_\epsilon(P) + \epsilon \log_2\left(n - 2^{\hat{\mathcal{H}}_\epsilon(P)}\right) \approx \mathcal{H}(P) + \epsilon \log_2\left(n - 2^{\mathcal{H}(P)}\right) \tag{3.26}$$

which is the expected entropy result.

Consider, for example, a memory system with a cache. The entropy of the input is the entropy of the sequence of memory addresses; the cache can be considered an approximation of the memory process, with ϵ being the miss ratio of the cache. To achieve a miss ratio of ϵ, the cache size has to be approximately $2^{\hat{\mathcal{H}}_\epsilon}$. Theorem 3.7 implies that very low miss ratios are possible with a cache size not much larger than $2^{\mathcal{H}}$ (making some assumptions about the sequence of memory addresses).

3.3.2 Breaking-Up the Input

Combining data from different sources in a single channel can increase the entropy of that channel, if some information is lost about the source of the data. Keeping this information around can decrease the complexity of the program that processes the data on the channel. In general, the more specific a program is to a particular type of data, the more efficient this program will be. The question is whether the input stream can be split in such a way that a net gain is achieved in the lower bound.

Consider a program P of the type:

$P \equiv *[\ I?x; [G_1(x) \longrightarrow O_1!f_1(x) \,[]\, G_2(x) \longrightarrow O_2!f_2(x)]]$

Suppose that $G_1(x)$ and $G_2(x)$ are known by the environment. Then P can be replaced by:

$P' \equiv\ P_1\ \|\ P_2$

$P_1 \equiv *[\ I_1?x; O_1!f_1 x\ ]$
$P_2 \equiv *[\ I_2?x; O_2!f_2 x\ ]$

where $G_1(x)$ holds for channel I_1, and $G_2(x)$ holds for channel I_2.

Let δ be the fraction of I_1 communications relative to the total $I_1 + I_2$. Then a lower bound for the parallel composition of P_1 and P_2 is given by:

$$\delta\mathcal{H}(P_1) + (1-\delta)\mathcal{H}(P_2) \leq \mathcal{C}(P_1\|P_2) \tag{3.27}$$

$P_1\|P_2$ is not strictly equivalent to P; some extra information about the input is preserved and that can translate in energy savings.

To compute the entropy of P, we model the sequences of communication commands on I_1 and I_2 as independent random variables, and the interleaving of actions in I as a coin toss of probability δ for I_1 and $1-\delta$ for I_2. In this case, splitting channel I into I_1 and I_2 destroys an amount of information corresponding to the result of the coin toss, $H_2(\delta) = -\delta \log_2 \delta - (1-\delta)\log_2(1-\delta)$ and, therefore:

$$\mathcal{H}(P) = \mathcal{H}(P_1 \| P_2) + H_2(\delta) \tag{3.28}$$

Observe that $H_2(\delta)$ is as well a lower bound to the cost of computing G_1 and G_2.

Processor memory accesses can be treated the same way as the previous example. The processor may be accessing data, instructions, stack, video-memory, data from different processes, etc. There is an entropy advantage to keeping those access channels separate; however, multi-ported memory is expensive by almost any measure. An alternative is to provide a caching mechanism for each type of access. A cache is an approximation of the memory system; a cache miss corresponds to an error, which can be recovered by the main memory process.

3.3.3 Control/Data Separation

Control/data separation is used in the synthesis method as a practical way of reducing the complexity of the design. Data is, in general, very regular and can be built from standard parts — such as multi-ported registers, adders and buses — while control has to be synthesized all the way to the transistor level and is very specific to the circuit being designed.

From the energy complexity point of view, control/data separation is a good idea. This is because operations that affect a variable in the program usually affect all bits of the variable equally — for example, reads and writes from a register —, and therefore actions on all bits are equiprobable, independently of the history of the computation. According to Theorem 3.5, grouping together these actions should help in getting closer to the lower bound.

In some cases, however, the operation on the register is such that not all bits of the register behave the same way. There may be some advantage to keeping part of the register in the control circuit. Consider, for example, the following program:

$$
\begin{aligned}
INC \equiv *[[\;& \overline{I} \longrightarrow x := x + 1; I \\
\| \;& \overline{S} \longrightarrow \; S!x \\
]] &
\end{aligned}
$$

where x is an n-bit variable. We flatten INC by expanding x into $x_0, \ldots, x_{n-1}$

$$
\begin{aligned}
INC_F \equiv *[[\;& \overline{I} \wedge \neg x_0 \longrightarrow x_0\uparrow; I \\
\| \;& \overline{I} \wedge x_0 \wedge \neg x_1 \longrightarrow x_0\downarrow, x_1\uparrow; I \\
& \ldots \\
\| \;& \overline{I} \wedge x_0 \wedge \ldots \wedge x_{n-2} \wedge \neg x_{n-1} \longrightarrow x_0\downarrow, \ldots, x_{n-2}\downarrow, x_{n-1}\uparrow; I \\
\| \;& \overline{S} \longrightarrow \; S!x \\
]] &
\end{aligned}
$$

Expanding the parallel assignments to the x_i variables into separate branches, we get order n^2 guarded commands; not counting the S communication, we get:

$$\mathcal{C}(INC_F) \approx 2\log_2 n \tag{3.29}$$

We use Theorem 3.5 to direct the hierarchical decomposition of INC_F; at each level, we require that all choices have the same probability. Observe that x_0 is toggled every time that an I communication is executed. We have, therefore, two equiprobable cases, $x_0 = \mathbf{true}$, and $x_0 = \mathbf{false}$:

$$INC_H \equiv INC_0 \| INC_1$$

$$
\begin{aligned}
INC_0 \equiv *[[\;& \overline{I} \wedge \neg x_0 \longrightarrow x_0\uparrow; I \\
\| \;& \overline{I} \wedge x_0 \longrightarrow x_0\downarrow, I_1; I \\
\| \;& \overline{S} \longrightarrow \; S!x \\
]] &
\end{aligned}
$$

$$
\begin{aligned}
INC_1 \equiv *[[\;& \overline{I_1} \wedge \neg x_1 \longrightarrow x_1\uparrow; I_1 \\
\| \;& \overline{I_1} \wedge x_1 \wedge \neg x_2 \longrightarrow x_1\downarrow, x_2\uparrow; I_1 \\
& \ldots \\
\| \;& \overline{I_1} \wedge x_1 \wedge \ldots \wedge x_{n-2} \wedge \neg x_{n-1} \longrightarrow x_1\uparrow, \ldots, x_{n-2}\uparrow, x_{n-1}\downarrow; I_1 \\
]] &
\end{aligned}
$$

In this decomposition INC_0 is the control and INC_1 is the datapath. Again, not counting the S communication, we get:

$$\mathcal{C}(INC_H) = \mathcal{C}(INC_0) + \frac{1}{2}\mathcal{C}(INC_1) \approx 1 + \log_2 n \tag{3.30}$$

This cost is half as much as $\mathcal{C}(INC_F)$. Keeping x_0 inside the control circuit will improve the energy cost of the program.

The previous example shows that control/data decomposition has to be done with some care. The main reason to remove data operations from sequencing is data parallelism. A command on a data word typically involves a command on each bit of that data word; given that the word-command has to be performed, the probability that one of the bit-commands will be performed next is the same for all bits and, therefore, according to Theorem 3.5, it makes sense to group those bit-commands together. In the incrementer example, some bit-commands are much more frequent than others, and a flat decomposition of the data is inefficient.

3.3.4 Pipelining/Parallelism

Pipelining has been used for a long time as a technique for reducing the cycle time of digital circuits. This reduced cycle time can be traded-off for reduced energy consumption, for example, by lowering the power supply voltage of the circuit — energy per operation is quadratic in the power supply voltage. Of course, a pipelined circuit will have a more complex behavior than its non-pipelined equivalent, and we can expect the entropy of the pipeline to be higher.

Consider, for example, a circuit designed to compute the function $h(x)$, where x can take n different values:

$P_h \equiv *[L?x;\, R!h(x)]$

Input and output commands alternate, and the function to be computed, $h(x)$, is deterministic; therefore, the sequence of input/output commands is completely determined by the sequence of input commands. In the worst-case, all values of the input are equiprobable, and we have:

$$\mathcal{H}(P_h) = \log_2 n \leq \mathcal{C}(P_h) \tag{3.31}$$

Assume that the function h can be expressed as $h = g \circ f$, where both g and f can take n different values. We pipeline the computation of $h(x)$ with the following program:

$P_{g \circ f} \equiv P_f \| P_g$

$P_f \equiv *[L?x; M!f(x)]$
$P_g \equiv *[M?y; R!g(y)]$

If we implement P_f and P_g separately, then each of those programs looks like P_h, and the cost of the parallel composition will have, as a lower bound:

$$\mathcal{C}(P_f \| P_g) \geq 2 \log_2 n \tag{3.32}$$

However, the cost of pipelining should be lower. Consider the transition diagram of Fig. 3.1. This transition diagram generates all the sequences of input/output alternations allowed by $P_f \| P_g$, I being the initial state. A path through this diagram corresponds to a sequence of coin-flips, one for every two commands. To describe the sequence of input and output commands completely, we need the sequence of input commands plus the sequence of results for the coin-flips. Using again the coding theorem, the entropy per input symbol of $P_f \| P_g$ can be bounded by:

$$\mathcal{H}(P_f \| P_g) \leq \log_2 n + 1 \tag{3.33}$$

From Eqs. 3.32 and 3.33 we conclude that there may be more efficient implementations of $P_{g \circ f}$. Consider, for example, the following program, where we replace pipelining with parallelism:

$P_p \equiv P_1 \| P_2$

$P_1 \equiv *[L?x; I; R!f(x); O]$
$P_2 \equiv *[I; L?x; O; R!g(x)]$

Program P_p is equivalent to $P_{g \circ f}$; both programs compute the same function and have a slack of 1 from L to R. The cost per input symbol of P_p can be computed as:

$$\mathcal{C}(P_p) = \frac{1}{2}\left(\mathcal{C}(P_1) + \mathcal{C}(P_2)\right) \leq 1 + \log_2 n \tag{3.34}$$

The previous result can be generalized to an m stage pipeline. The state of the pipeline can be modeled by an integer $p = cL? - cR!$. p is the difference between the number of completed $L?$ commands and the number of completed $R!$ commands and, therefore, $0 \leq p \leq m$. The sequence of values of p, plus the input sequence, completely determine the input/output behavior of the

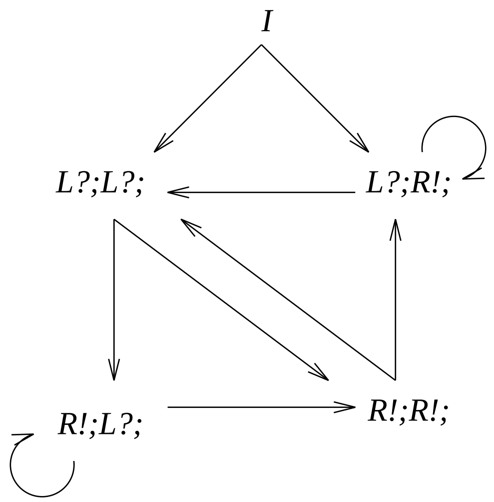


Figure 3.1: Transition diagram for $P_f \| P_g$. A path on this diagram corresponds to an allowed sequence of input/output symbols.

pipeline. In the worst-case, p can be modeled by a random walk between 0 and m, for which we need, at most, one coin flip per input symbol. For an m stage pipeline P_m, we can write:

$$\mathcal{H}(P_m) \leq \log_2 n + 1 \tag{3.35}$$

Eq. 3.35 implies that an arbitrary amount of pipelining can be achieved at almost no cost. That is true under the energy model described in this chapter; the program P_p can be extended to an arbitrary number of parallel processes. In practice, there is a hidden cost in the implementation of the multiple-receiver channel L and multiple-sender channel R, that is not captured by the model. If we restrict the valid implementations to single-sender, single-receiver channels, the comparison between pipelining and parallelism becomes a comparison between the costs of copying information and the cost of splitting a channel. Copying information increases the number of transitions, while splitting channels increases the cost of each transition; nevertheless, splitting a channel in two can be done almost for free by using two-phase signaling protocols and may reduce the entropy of each channel. Other reasons to prefer parallelism over pipelining are given in other chapters.

3.4 Summary & Conclusion

In this chapter we have shown how to abstract the complexity of the specification of a circuit from the actual implementation. The complexity of the specification is derived from its input/output behavior and is expressed as a lower limit to the achievable energy per instruction performance of any CSP process that satisfies the specification. This lower bound is based on the entropy of the communication symbols with the environment, which expresses with a single number how hard it is to generate that sequence.

This lower bound is not necessarily tight. Theorem 3.5 provides sufficient conditions for getting close to that limit. These conditions can be used to direct the synthesis procedure; in particular, they have consequences in the way data and control are separated.

Other strategies for improving energy performance have been discussed. It is possible to improve the lower bound by changing the specification of the environment — splitting the input channels, for example — or by selecting alternative implementations of equivalent circuits that get closer to the lower bound — choosing parallelism over pipelining — or by increasing the number of candidates for an efficient solution — program approximation.

The results presented here are limited in scope by the validity of the energy model selected. Their usefulness resides in allowing us to search a large design space at the high level when the design costs are still low. Among other areas, we can look at the specification of the circuit and try to simplify or change this specification for one that has a better lower bound; we can direct the first division of the problem into subproblems, so that the entropy of the subproblems is as close as possible to the original entropy.

Chapter 4

Low-Energy Programs

In this chapter we describe some programming techniques that result in energy efficient programs. These techniques are justified in terms of the energy model for the resulting program structure.

The energy model for a program defines the energy complexity of the algorithm being implemented; we use complexity arguments to decide which programs are best. This works well for particular instances, but in general we would like to have a catalog of program templates, or program styles, that we know are energy efficient. Using these templates as a starting point for algorithm development, we can reduce the size of the design space without throwing away interesting solutions.

Next we present four different techniques for reducing energy consumption. These techniques try to reduce or eliminate the amount of useless work, or improve the time complexity without hurting the energy complexity, or replace circuits with cheaper, equivalent circuits.

4.1 Reactive Programs

Reactive programs alternate computation with waiting for input. Computation is done only on demand, and waiting for input does not require energy.

One of the reasons that asynchronous circuits are energy efficient is that the design methodology promotes a reactive programming style. As a result, all transitions contribute directly to the computation, and no energy is wasted in busy waiting or in checking that there is no more work to be done.

An example where busy waiting is employed is the ring of mutual exclusion. A token is passed around a ring, and the processes that want access to a restricted resource have to obtain the token from the ring before being granted access.

$TkRng \equiv *[\ L?; [\ \overline{U} \longrightarrow U?; U? \ [\!]\neg\overline{U} \longrightarrow \textbf{skip}\ ]; R!\ ]$

$TkRngO \equiv *[[\ \overline{U} \longrightarrow U?; U? \ [\!]\neg\overline{U} \longrightarrow \textbf{skip}\ ]; R!; L?\ ]$

$Master \equiv \ldots; U!; Critical\ \ Section; U!; \ldots$

Fig. 4.1 shows how these processes are interconnected. The number of *TkRngO* processes in the ring determines how many *Master* processes can be in the critical section simultaneously.

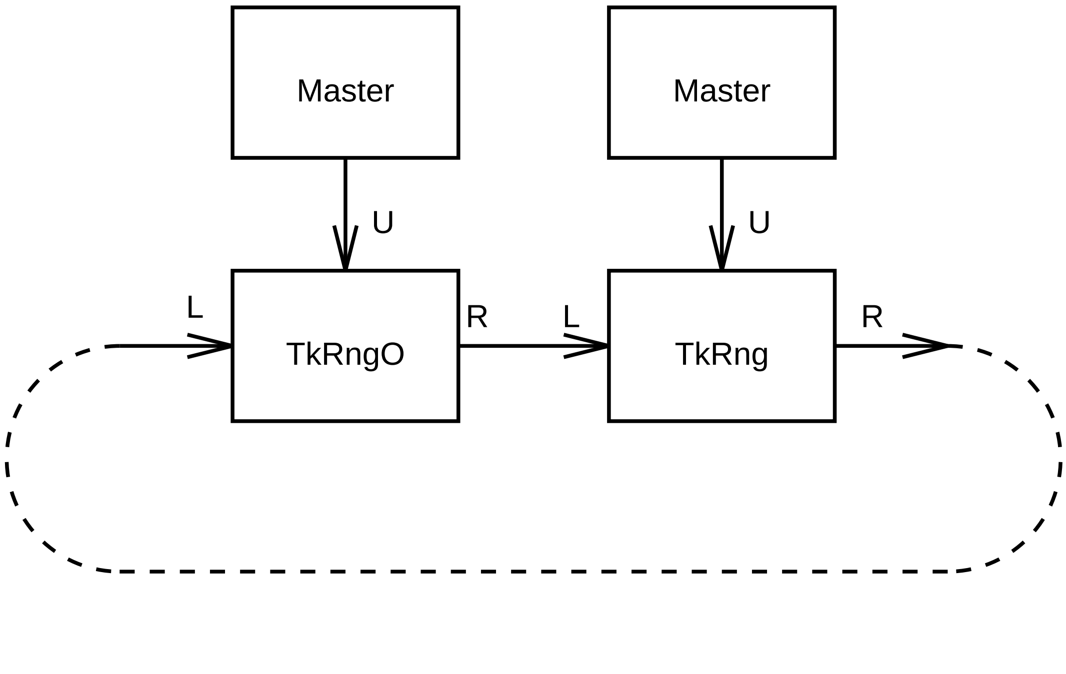


Figure 4.1: Channel connections for a mutual exclusion token ring.

The token is continuously circulating around the ring, even if there are no requests for the token. The distinguishing property of this ring of mutual exclusion is that if all critical sections eventually terminate, selection among the processes requesting for access is fair. Fair selection, however, can be achieved without busy waiting. Consider the program:

$$Mutex \equiv *[[\ \overline{U_1} \longrightarrow U_1?;\, U_1? \ \square \ldots \square \overline{U_n} \longrightarrow U_n?;\, U_n?\ ]]$$

Requestor i enters the critical region between the two U_i communications. This program ensures mutual exclusion, but it is not fair. We transform the program in the following way:

$$Mutex \equiv Top \,\|\, Left \,\|\, Right$$

$$\begin{array}{l} Top \equiv *[[\ \overline{L} \longrightarrow L?;\, L?;\, [\ \overline{R} \longrightarrow R?;\, R? \\ \qquad\qquad\qquad\qquad\qquad\quad \square \neg\overline{R} \longrightarrow \mathbf{skip} \\ \qquad\qquad\qquad\qquad\qquad\quad] \\ \qquad\qquad \square\ \overline{R} \longrightarrow R?;\, R?;\, [\ \overline{L} \longrightarrow L?;\, L? \\ \qquad\qquad\qquad\qquad\qquad\quad \square \neg\overline{L} \longrightarrow \mathbf{skip} \\ \qquad\qquad\qquad\qquad\qquad\quad] \\ \qquad\quad]] \end{array}$$

$$Left \equiv *[[\ \overline{U_1} \longrightarrow L!;\, U_1?;\, U_1?;\, L!\ \square \ldots \square \overline{U_j} \longrightarrow L;\, !U_j?;\, U_j?;\, L!\ ]]$$

$$\begin{array}{l} Right \equiv *[[\ \overline{U_{j+1}} \longrightarrow R!;\, U_{j+1}?;\, U_{j+1}?;\, R!\ \square \ldots \square \overline{U_n} \longrightarrow R!;\, U_n?;\, U_n?;\, R! \\ \qquad\quad]] \end{array}$$

Selections between the requests from *Left* and the requests from *Right* are fair. We apply the same transformation recursively, and we achieve fair selection between all requests. Fairness is obtained at the expense of an extra check on the non-selected channels. If no requests are being made, however, the previous program becomes idle, and no energy is dissipated.

4.2 Lazy Programs

There are a number of choices concerning when to execute data-communication actions in a program. The earliest possible time is when the data is known by the sender; the latest possible time is when the data is known to be needed by the receiver. The earlier the data is sent, the better the delay performance of the program as a whole. There is, however, the risk that the data sent is not always needed by the receiver, and, in those cases, we can save energy by postponing sending data until it is known to be needed on the other end.

A typical example of this situation is the linear stack. A linear stack process of size N can be defined recursively as follows:

$$
\begin{aligned}
STACK(N, \mathbf{in}\ put, \mathbf{out}\ get) \equiv\ & SE(\mathbf{in}\ put, \mathbf{out}\ get, \mathbf{out}\ push, \mathbf{in}\ pop) \\
& \|\ STACK(N-1, \mathbf{in}\ putp, \mathbf{out}\ getp) \\
& \mathbf{connect}\ \ push, putp \\
& \mathbf{connect}\ \ pop, getp
\end{aligned}
$$

Last-in, first-out (LIFO) order is preserved between *put* and *get*; the environment will not try to *put* an element into a full stack, nor *get* an element from an empty stack. The actual definition of *SE* and *STACK(0)* determines the type of stack. To facilitate describing the processes, we will represent the stack as a left-to-right queue, with the leftmost element being the top of the stack.

We are looking for a stack element *SE* with the following restrictions: *SE* contains only one register, and for any sequence of *put*s and *get*s the number of assignments is the minimum for all one-register stack elements.

With only one register per stack element, we cannot re-order the data stored in those registers. As long as we do not destroy any information — that is, we communicate the data before we overwrite the register — the LIFO property will be preserved.

To minimize the number of data movements to the right, the stack element has to execute a *push* only when absolutely necessary. This condition occurs when the stack element is holding data, all the stack elements to the left are holding data, and the environment is trying to *put* into the stack. The same condition can be applied to the stack element on the left; this stack element will try to *push* only when it and the rest of the stack to the left are holding data and the environment is trying to *put*. Each stack element has to keep track of whether it is full or empty. In CSP, these conditions can be expressed as:

$$
\begin{aligned}
*[[\ & \overline{put} \wedge \ \ empty \longrightarrow \ put?x, empty\!\downarrow \\
[]\ & \overline{put} \wedge \neg empty \longrightarrow \ push!x; put?x \\
]]
\end{aligned}
$$

To minimize the number of data movements to the left, the stack element has to execute a *pop* only when absolutely necessary. This condition occurs when the stack element is not holding data, all the stack elements to the left are not holding data, and the environment is trying to *get* from the stack. Again, the same condition can be applied to the stack element to the left; this stack element will try to *pop* only when it and the rest of the stack to the left are not holding data and the environment is trying to *get*. In CSP, these conditions can be expressed as:

$$
\begin{array}{lllll}
*[[& \overline{get} \wedge & empty & \longrightarrow & pop?x; get!x \\
\ \square & \overline{get} \wedge & \neg empty & \longrightarrow & get!x, empty\uparrow \\
]] & & & &
\end{array}
$$

Putting these two programs together, we get the code for the lazy stack:

$$
\begin{array}{lllll}
\multicolumn{5}{l}{SE\_lazy \equiv \{empty = \mathbf{true}\}} \\
*[[& \overline{put} \wedge & empty & \longrightarrow & put?x, empty\downarrow \\
\ \square & \overline{put} \wedge & \neg empty & \longrightarrow & push!x; put?x \\
\ \square & \overline{get} \wedge & empty & \longrightarrow & pop?x; get!x \\
\ \square & \overline{get} \wedge & \neg empty & \longrightarrow & get!x, empty\uparrow \\
]] & & & &
\end{array}
$$

The full/empty condition can be encoded in many ways. An alternate form of encoding is the following:

$$
\begin{array}{l}
SE\_lazy \equiv *[[\ \overline{put} \longrightarrow put?x \ \square\ \overline{get} \longrightarrow pop?x\ ]; \\
\qquad\qquad\quad [\ \overline{put} \longrightarrow push!x \ \square\ \overline{get} \longrightarrow get!x\ ] \\
\qquad\qquad\ ]
\end{array}
$$

The lazy stack is not CRT[1]; it takes $\frac{N(N+1)}{2}$ time-steps to *put* N elements into the stack and $\frac{N(N+1)}{2}$ time-steps to *get* them out. However, this is a worst-case; performance improves greatly if *put*s and *get*s are intermixed. In the best case, where *put* and *get* alternate, it takes $2N$ time-steps to execute N *put*s and N *get*s.

In order for *put* to be CRT, the order of the communications $push!x; put?x$ has to be inverted [16]. To make the *get* operation CRT, the $pop?x$ communication has to be done ahead of time and whenever the stack element becomes empty.

[1] CRT = Constant Response Time

These two conditions can be met with an extra register in the stack element. A bottom marker is used to avoid propagating *put*s and *get*s into the empty part of the stack.

$$
\begin{array}{l}
SE\_eager \equiv \ \{\ x = y = \perp\ \} \\
\qquad *[[\ \overline{put} \longrightarrow\ put?x,\ [y \neq \perp \longrightarrow push!y\ \ [\!]\ \ y = \perp \longrightarrow \mathbf{skip}] \\
\qquad\quad [\!]\ \overline{get} \longrightarrow\ get!y,\ \ [x \neq \perp \longrightarrow pop?x\ \ [\!]\ \ x = \perp \longrightarrow \mathbf{skip}] \\
\qquad\quad]; \\
\qquad\quad [\ \overline{put} \longrightarrow\ put?y,\ [x \neq \perp \longrightarrow push!x\ \ [\!]\ \ x = \perp \longrightarrow \mathbf{skip}] \\
\qquad\quad [\!]\ \overline{get} \longrightarrow\ get!x,\ \ [y \neq \perp \longrightarrow pop?y\ \ [\!]\ \ y = \perp \longrightarrow \mathbf{skip}] \\
\qquad]]
\end{array}
$$

At best, half of the registers in the stack will contain data.

The previous encoding presents a symmetry between x and y that can be exploited. A simple analysis of *SE_eager* shows that the stack behaves like two separate stacks, one with the x registers and one with the y registers. If we split channels *pu*, *get*, *push*, and *pop* we can recode the stack element in the following way:

$$
\begin{array}{l}
SE\_eager \equiv \ \{\ x = y = \perp\ \} \\
\qquad *[[\ \overline{putx} \longrightarrow\ ypush!\bullet putx?x \\
\qquad\quad [\!]\ \overline{getx} \longrightarrow\ ypop!\bullet getx!x \\
\qquad\quad [\!]\ \overline{xpop} \longrightarrow\ [x \neq \perp \longrightarrow xpop?, popx?x\ \ [\!]\ \ x = \perp \longrightarrow xpop?] \\
\qquad\quad [\!]\ \overline{xpush} \longrightarrow\ [x \neq \perp \longrightarrow xpush?, pushx!x\ \ [\!]\ \ x = \perp \longrightarrow xpush?] \\
\qquad]]\ \| \\
\qquad *[[\ \overline{puty} \longrightarrow\ xpush!\bullet puty?y \\
\qquad\quad [\!]\ \overline{gety} \longrightarrow\ xpop!\bullet gety!y \\
\qquad\quad [\!]\ \overline{ypop} \longrightarrow\ [y \neq \perp \longrightarrow ypop?, popy?y\ \ [\!]\ \ y = \perp \longrightarrow ypop?] \\
\qquad\quad [\!]\ \overline{ypush} \longrightarrow\ [y \neq \perp \longrightarrow ypush?, pushy!y\ \ [\!]\ \ y = \perp \longrightarrow ypush?] \\
\qquad]]
\end{array}
$$

The top of the stack has to split the data into the x and y stacks:

$$
\begin{array}{l}
ST\_eager \equiv \{\ x = y = \perp\ \} \\
\quad *[[\ \overline{put} \longrightarrow\ put?x,\ [y \neq \perp \longrightarrow pushy!y\ \ [\!]\ \ y = \perp \longrightarrow \mathbf{skip}] \\
\quad\ \ [\!]\ \overline{get} \longrightarrow\ get!y,\ \ [x \neq \perp \longrightarrow popx?x\ \ [\!]\ \ x = \perp \longrightarrow \mathbf{skip}] \\
\quad\ \ ]; \\
\quad\ \ [\ \overline{put} \longrightarrow\ put?y,\ [x \neq \perp \longrightarrow pushx!x\ \ [\!]\ \ x = \perp \longrightarrow \mathbf{skip}] \\
\quad\ \ [\!]\ \overline{get} \longrightarrow\ get!x,\ \ [y \neq \perp \longrightarrow popy?y\ \ [\!]\ \ y = \perp \longrightarrow \mathbf{skip}] \\
\quad]]
\end{array}
$$

This encoding of the eager stack has the advantage that data channels always have the same source and destination registers, which reduces the cost of those communication actions and allows for interesting optimizations at the circuit level.

4.2.1 Non-Causal Probe

Using knowledge about the future, we can improve the efficiency of a program by doing operations in advance or eliminating unnecessary operations. The non-causal probe, represented as $\overline{\overline{X}}$ (double overbar), introduces this type of knowledge in the programming notation.

The non-causal probe can be used to probe a channel in the guard of an IF statement. The non-causal probe becomes true if the probe on the same channel is guaranteed to become true if the execution of the IF statement were suspended forever. Clearly, the regular probe is an implementation of the non-causal probe; other implementations are possible.

We can use the non-causal probe to improve the efficiency of the lazy stack:

$$
\begin{array}{l}
SE\_lazy \equiv \\
\quad *[[\ \overline{\overline{put}} \longrightarrow\ put?x\ \ [\!]\ \overline{\overline{get}} \longrightarrow\ pop?x\ ]; \\
\quad\ \ [\ \overline{\overline{put}} \longrightarrow\ push!x\ \ [\!]\ \overline{\overline{get}} \longrightarrow\ get!x\ \ ] \\
\quad\ ]
\end{array}
$$

In the case of the lazy stack, the non-causal probe allows $push!x$ and $pop?x$ to execute before *put* and *get* become pending. The stack works in both directions as a left-right buffer and will, therefore, have CRT. Since *put* and *get* are mutually exclusive, the order and total number of *put* and *get* operations is going to be the same as in the case of the lazy stack; therefore, this new lazy stack also uses minimum energy.

The non-causal probe has to have a practical implementation to be useful. It is not possible to know all the future of the computation; otherwise, the result of the computation would be known in advance. It is possible, however, to know the near future: a stack machine will execute two *get*s and one *put* for each ALU operation; a digital signal processor will execute a fixed sequence of *put*s and *get*s to simulate a digital filter (this sequence can be pre-calculated, even though the data is not known in advance).

If the guards which contain non-causal probes do not involve data, the difference in propagation delay between control signals and data can be used to determine the value of the non-causal probe in advance (even though little is known about the next operation). Another approach is to guess the value of the probe and undo the work in case of an incorrect guess. If the value is guessed right most of the time, this strategy can improve performance, but may increase the energy requirement.

The non-causal probe can be implemented as a data channel with slack, where choices between guards are pre-calculated and queued for execution. We modify the code for the lazy stack in this way. We introduce a new channel, nxt, to get the value of the non-causal probe.

$$
\begin{array}{lll}
SE\_lazy \equiv *[[& (\overline{nxt?} = \text{put}) \longrightarrow & nxt, put?x \\
& [\!]\ (\overline{nxt?} = \text{get}) \longrightarrow & pop?x \\
&]; & \\
& [\ (\overline{nxt?} = \text{put}) \longrightarrow & push!x \\
& [\!]\ (\overline{nxt?} = \text{get}) \longrightarrow & nxt, get!x \\
&]] &
\end{array}
$$

In this case, this transformation has the advantage of eliminating the probes on the data channels, allowing freedom of choice between active and passive implementations.

If the choice is deterministic, it is always valid to replace $\overline{\overline{A}}$ by $\overline{A} \vee \overline{\overline{A}}$. This transformation can be used to reduce the latency introduced by the slack on the nxt channel, replacing $(\overline{nxt?} = \text{A})$ by $\overline{A} \vee (\overline{nxt?} = \text{A})$.

The data on the nxt channel can be calculated by simulating the stack with a dataless stack. The ordering between *push* and *get* or *put* and *get* does not have to be maintained: data is not involved, and these communications will be re-ordered by the stack element.

$$
\begin{array}{lllll}
\mathit{SEN\_lazy} \equiv & \{\mathit{empty} = \mathbf{true}\} & & & \\
\quad *[[& \overline{\mathit{nput}} \wedge & \mathit{empty} & \longrightarrow & \mathit{nput}?, \mathit{empty}\!\downarrow, \mathit{nxt}!(\mathrm{put}) \\
\quad\ \ [] & \overline{\mathit{nput}} \wedge & \neg\mathit{empty} & \longrightarrow & \mathit{npush}!, \mathit{nput}?, \mathit{nxt}!(\mathrm{put}) \\
\quad\ \ [] & \overline{\mathit{nget}} \wedge & \mathit{empty} & \longrightarrow & \mathit{npop}?, \mathit{nget}!, \mathit{nxt}!(\mathrm{get}) \\
\quad\ \ [] & \overline{\mathit{nget}} \wedge & \neg\mathit{empty} & \longrightarrow & \mathit{nget}!, \mathit{empty}\!\uparrow, \mathit{nxt}!(\mathrm{get}) \\
\quad\ ]] & & & &
\end{array}
$$

As long as there is enough slack in the *nxt* channel, the data-less stack will have CRT. This slack has to store as much of the sequence of *get*s and *put*s as we can pre-calculate. Additional slack will only increase the latency and energy consumption.

4.3 Worst-Case Delay/Average Energy

In the design of a digital computer, we are not so much interested in the time required to complete each individual operation, as in the time required to complete large tasks. Making each operation very fast helps, but making operations in average very fast gets the same result and is a weaker requirement.

This is partially the approach taken in RISC processor architecture. The instruction set is simplified (types of instructions, addressing modes) so that instructions execute very fast. Some operations require several instructions, but the average duration of the operation is shorter than in an equivalent CISC processor.

In an asynchronous design we can go one step further by using the data-dependent delays of some operations in our favor. Making sure that the worst-case delay of an instruction execution fits within one clock cycle requires some extra hardware (for example, using a tree adder instead of a ripple carry adder). If the average case data-dependent delay is low enough, then we can save the extra hardware and some energy dissipation as well. In general, the fewer restrictions we put on the worst-case delay, the better average delay and average energy we can obtain.

Arithmetic circuits have large, data-dependent variations in delay due to the variable length of carry chains. Carry chains can be made of uniform length, using, for example, tree adders, or carry select adders. The worst-case and average-case delay of a tree adder will be logarithmic in the number of bits in the input data, but twice as many adders are needed to compute the addition. A ripple-carry adder will have a worst-case delay linear on the number of bits in the input data, but the average case (assuming that all inputs are equiprobable) is logarithmic delay. The average energy-per-addition for the ripple carry adder is better than the larger, more complex tree adder for roughly the same average delay.

The main reason this trade-off is possible is that, in many interesting cases, the circuit to be implemented can be approximated by another simpler circuit that computes the same function in the most common cases and generates an error otherwise. Ch. 3 goes in depth into this trade-off; we give here only a few examples.

We look first at zero-detection for a large number of bits (for example, a floating point number with 80 bits). We can determine in logarithmic time and linear energy in the number of bits that the result is zero; however, most of the time the number is different from zero, and it should take substantially less energy to do that determination.

We assume that all bits are equally distributed independent random variables, with equal probability of being 1 or 0. We represent the number as a vector, $B[1..n]$. Then we can write:

$$B[1..n] = 0 \Leftrightarrow B[1..j] = 0 \wedge B[j+1..n] = 0 \tag{4.1}$$

We start with the following specification for the zero detection circuit:

$Zero(n) \equiv *[[\ \overline{Z} \longrightarrow Z!(B[1..n] = 0)\ ]]$

The following program uses lazy evaluation to reduce energy consumption:

$Zero(n) \equiv Zero1 \parallel Zero(n-1)$

$Zero1 \equiv *[[\ \overline{Z} \wedge B[1] \longrightarrow Z!\textbf{false}$
$\quad [\!]\ \overline{Z} \wedge \neg B[1] \longrightarrow Z!(Z1?)$
$\quad]]$

$Zero(n-1) \equiv *[[\ \overline{Z1} \longrightarrow Z1!(B[2..n] = 0)\ ]]$

The energy consumption of $Zero(n)$ can be computed as:

$$\begin{aligned} C(Zero(n)) &= C(Zero1) + \Pr(B[1] = \textbf{false})C(Zero(n-1)) \\ &= K_1 + \frac{1}{2}C(Zero(n-1)) \end{aligned} \tag{4.2}$$

where K_1 is the energy cost of executing *Zero1*, which is independent of n. If we apply the same transformation recursively, until all of the vector is broken up in bits, we get:

$$C(Zero(n)) < 2K_1 \tag{4.3}$$

We compute similarly the average-case delay, $D(Zero(n))$, and the worst-case delay, $WD(Zero(n))$:

$$\begin{aligned} D(Zero(n)) &< 2T_1 && (4.4)\\ WD(Zero(n)) &= nT_1 && (4.5)\end{aligned}$$

where T_1 is the delay of $Zero(1)$.

The worst-case delay can be reduced at the expense of some energy. The following program evaluates the zero condition of all bits simultaneously:

$Zero(n) \equiv *[[\ \overline{Z} \longrightarrow Z!(Z0? \ \ \wedge \ \ Z1?)\ ]]$

$Zero(1..j) \equiv *[[\ \overline{Z0} \longrightarrow Z0!(B[1..j] = 0)\ ]]$

$Zero(j+1..n) \equiv *[[\ \overline{Z1} \longrightarrow Z1!(B[j+1..n] = 0)\ ]]$

The energy and delay equations in this case are:

$$\begin{aligned} C(Zero(1..n)) &= K_m + 2C(Zero(1..n/2)) && (4.6)\\ WD(Zero(1..n)) &= D_m + D(Zero(1..n/2)) && (4.7)\end{aligned}$$

Applying the transformation recursively, until all of the vector is broken up in bits, we get:

$$\begin{aligned} C(Zero(1..n)) &\approx 2nK_m && (4.8)\\ WD(Zero(1..n)) &\approx D_m \log_2 n && (4.9)\end{aligned}$$

The worst-case delay has improved significantly, but both the average-case delay and average cost have increased. An alternative is to combine both programs so that the worst-case delay can be reduced without such a high penalty. Assume $n = 2^N$; we apply the second transformation J times, $J < N$, and we are left with 2^J vectors of length 2^{N-J} to check against zero.

Combining the equations for both cases, we obtain:

$$C(J) \approx (2^J - 1)K_m + 2^{J+1}K_1 \quad (4.10)$$

$$D(J) \approx JT_m + 2T_1 \quad (4.11)$$

$$WD(J) \approx JT_m + 2^{N-J}T_1 \quad (4.12)$$

The previous equations can be used to select J given the largest allowable worst-case delay, average delay, or energy cost.

4.4 Concurrency

In the previous sections we reduced the energy consumption by increasing the sequentiality of the program, thus avoiding useless work. In the examples presented (both the lazy stack and the zero detection circuit) there was enough inter-symbol dependency in the input stream that it made sense to postpone some computations until the results from other computations were ready.

If the computations to be performed on the input stream are independent on each symbol (as, for example, on a vector operation), we have two immediate choices for efficient implementation: pipelining of the operation and parallelism. Pipelining improves the cycle time, but it requires a lot of copying of data down the pipe. Parallelism splits the input stream into several sub-streams and performs the computation in parallel on each sub-stream at a much-reduced rate. There is less copying of information (except for the split and possibly the merge of the streams at the end of the computation), but the area requirement for a given throughput might be larger than for the pipelined circuit because of the replication of the computing circuit and the routing of signals.

Many other applications can be parallelized efficiently. The following is an example of a serial-to-parallel converter:

$S2P \equiv *[[\ L?x;\ \ U!x;\ \ L?x;\ \ D!x]]$

Fig. 4.2 shows the channel interconnection for the converter.

In this case, we have the liberty to give different implementations for the first few stages of the split rather than for the last stages. The throughput will be high only in the first stages, which imposes extra requirements on the

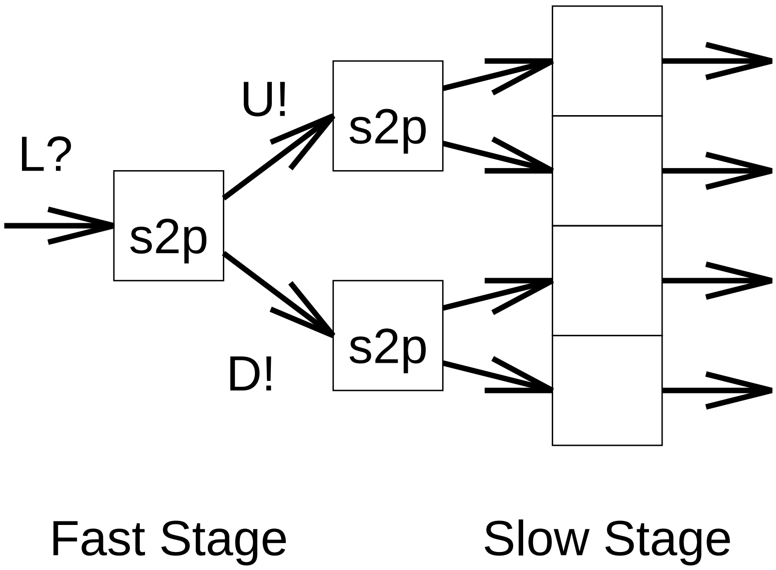


Figure 4.2: Channel interconnection for a serial-to-parallel converter.

circuit design. Even though all stages have the same CSP code, it is reasonable to expect that the energy dissipation of fast and slow stages will be different in actual implementations. We can take care of this problem by selecting different constants for the communication actions of each stage, though this approach involves mixing delay considerations with the energy model.

A different type of concurrency can be extracted from computations that have some predictability, such as instruction memory accesses. Instruction memory accesses are, most of the time, accesses to consecutive memory locations. This fact can be exploited in two ways to make a more efficient memory in terms of speed and energy cost. First, the address does not need to be communicated all of the time; it can be calculated locally. Second, several consecutive words can be read in parallel at one time, thus reducing the number of memory references.

The following program describes a read-only memory with sequential access:

$$MEMS \equiv *[[\ \overline{A} \longrightarrow A?a \ \ [] \ \overline{D} \longrightarrow D!M[a{+}{+}]\ ]]$$

where M is an $n \times b$ array (a++ means post-increment a with wrap-around). After an address a is sent to the memory, several data requests are executed.

Sequencing between the A and D communications is maintained by the environment.

We can reduce the number of accesses to the array M by reading several words in parallel. We replace M by an $(n/m) \times mb$ array, MP. After receiving an address, the variable *line* is read from the array, and subsequent data requests are satisfied with data from *line* until $a.w$ overflows and new data is read into *line*.

$$MEMR \equiv \ (PREF \ \| \ MEMP)$$

$$
\begin{array}{l}
PREF \equiv *[[\ \overline{A} \longrightarrow \ A.w?a.w, \ \ B!A.l?, \ \ v\downarrow \\
\qquad\quad [\!]\ \overline{D} \wedge v \longrightarrow \ \ D!line[a.w{+}{+}]; \\
\qquad\qquad\qquad\qquad\quad [a.w = 0 \longrightarrow v\downarrow \\
\qquad\qquad\qquad\qquad\quad [\!]\ \ a.w \neq 0 \longrightarrow v\uparrow \\
\qquad\qquad\qquad\qquad\quad] \\
\qquad\quad [\!]\ \overline{D} \wedge \neg v \longrightarrow \ \ L?line, \ \ v\uparrow \\
\qquad\quad]]
\end{array}
$$

$$MEMP \equiv *[[\ \overline{B} \ \longrightarrow \ B?b \quad [\!]\ \overline{L} \ \longrightarrow \ L!MP[b{+}{+}]\ ]]$$

Notice that *MEMP* has the same form as *MEMS*.

We compute next the average energy cost of reading a word from *MEMR*, and compare that cost with the energy cost of reading a word from a memory array. Let k be the number of consecutive memory references. To satisfy those k requests, the program *MEMR* has to execute one $A.w$, $A.l$, and B-communication; k D-communications; about $\lceil \frac{k}{m} \rceil$ L-communications; about $\lceil \frac{k}{m} \rceil$ reads from an $(n/m) \times mb$ array (MP); and k reads from a $m \times b$ array (*line*). The energy cost of executing those k requests is:

$$
\begin{aligned}
k \times E_S \ &= \ K_A \log_2 n + K_B \log_2 \frac{n}{m} + k(E_R(m,b) + K_D) + \\
&\quad \left\lceil \frac{k}{m} \right\rceil (E_R(n/m, mb) + K_L) \qquad (4.13)
\end{aligned}
$$

where $E_R(n,b)$ is the energy cost of reading an $n \times b$ array. We take expected value on both sides of the equation to obtain the expected energy cost of reading one word from *MEMR*, $E_S(n,m,b)$:

$$E_S(n,m,b) =$$

$$\Big(K_A \log_2 n + K_B \log_2 \frac{n}{m} + \overline{k}(E_R(m,b) + K_D) + \\ \overline{\left\lceil \frac{k}{m} \right\rceil}\,(E_R(n/m, mb) + K_L)\Big)\Big/\overline{k} \tag{4.14}$$

To optimize the previous equation, we use Eq. 2.47 for $E_R(n/m, mb)$; $\overline{k}$ and $\overline{\left\lceil \frac{k}{m} \right\rceil}$ are determined from program traces.

For example, if all constants are equal to 1, $n = 2^{20}$, $b = 32$, $\overline{k} = 8$, we obtain the minimum energy cost for $m = 8$, $E_S = 1134$. Compared to the minimum energy cost of accessing a memory array, $E_R = 1493$, we do not obtain a significant improvement. However, the optimal block size for this parameter set is eight words per block; with this block size, most of the silicon area occupied by the memory will be dedicated to routing of data and address, resulting in very poor memory density. We can choose a sub-optimal block size to improve in density; for a block size of 2^{10} words, we get $E_R = 3195$, and $E_S = 2590$. The pre-fetch mechanism allows us to use a denser memory with a smaller energy penalty.

We use a simple sequential access model to compute the hit ratio for the *line* array. If $a[i]$ is the i^{th} address in the sequence of accesses, then $\Pr\left(a[i+1] = a[i] + 1\right) = \lambda$; that is, the probability that the next address is in sequence with the previous one is λ, $0 \le \lambda \le 1$. Also, if $a[i+1] \neq a[i] + 1$, then $a[i+1]$ is uniformly distributed over the address space. If n is the number of memory locations, we have:

$$\Pr(a[i+1] = x) = \begin{cases} \lambda + \frac{1-\lambda}{n}, & \text{if } x = a[i] + 1 \\ \frac{1-\lambda}{n}, & \text{otherwise} \end{cases} \tag{4.15}$$

With this model we compute how many words are used in average from the line retrieved from memory. If the line has m words, and l is the number of words used from the line, then the expected value of l, $E(l)$, can be calculated as:

$$E(l) = \sum_{i=0}^{+\infty} i \Pr(l = i) = \sum_{i=0}^{+\infty} \Pr(l > i) = \sum_{i=0}^{m-1} \Pr(l > i) \tag{4.16}$$

Let y be the index within the line of the first word in the sequence. Let P_0 be the probability of having overflow from one line to the next. Using the uniform distribution assumption, we have:

$$\Pr(y=i)=\begin{cases} P_0+\frac{1-P_0}{m}, & \text{if } i=0 \\ \frac{1-P_0}{m}, & \text{if } 0<i<m \\ 0, & \text{otherwise} \end{cases} \tag{4.17}$$

We can calculate the probability of overflow, P_0 as

$$\begin{aligned} P_0 &= \sum_{i=1}^{m} \Pr(y=m-i)\lambda^i \\ &= P_0\lambda^m + \frac{1-P_0}{m}\sum_{i=1}^{m}\lambda^i \\ &= P_0\lambda^m + \lambda\frac{1-P_0}{m}\frac{1-\lambda^m}{1-\lambda} \end{aligned} \tag{4.18}$$

Solving for P_0,

$$P_0=\frac{\lambda}{m(1-\lambda)+\lambda} \tag{4.19}$$

The probability of $l>i$ can be calculated as the probability of getting at least $i+1$ addresses in sequence, times the probability that the first address in the sequence falls in the first $m-i$ words of the line. That is,

$$\Pr(l>i)=\Pr(y<m-i)\lambda^i \tag{4.20}$$

$$\Pr(y<m-i)=P_0+(1-P_0)\frac{m-i}{m}=1-\frac{(1-\lambda)i}{m(1-\lambda)+\lambda} \tag{4.21}$$

$$\begin{aligned} E(l) &= \sum_{i=0}^{m-1}\left(\lambda^i-\frac{(1-\lambda)i\lambda^i}{m(1-\lambda)+\lambda}\right) \\ &= \frac{\left(1-\lambda-2\lambda^m+2\lambda^{m+1}\right)m+2\lambda-2\lambda^{m+1}}{(1-\lambda)\left(m(1-\lambda)+\lambda\right)} \end{aligned} \tag{4.22}$$

The hit ratio for this one-line cache is $h(m,\lambda)=E(l)/m$,

$$h(m,\lambda)=\frac{\left(1-\lambda-2\lambda^m+2\lambda^{m+1}\right)m+2\lambda-2\lambda^{m+1}}{m(1-\lambda)\left(m(1-\lambda)+\lambda\right)} \tag{4.23}$$

Notice that $h(m,0)=1/m$ (addresses are random; only one word is used per line) and $h(m,1)=1$ (addresses are sequential; all words are used from every line). For $m = 1/(1-\lambda)$ (average length of a sequence of consecutive addresses), we can easily show that:

$$h(\frac{1}{1-\lambda}, \lambda) > \frac{3}{2} - \frac{2}{e} \approx 0.76 \tag{4.24}$$

Given the energy per access for a memory of size $(n/m) \times mb$, $E_a(n/m, mb)$, we can calculate the average energy per access using the previous pre-fetch scheme as the energy cost of retrieving a line from a m-word wide memory, divided by the average number of words used from a line:

$$E_p(n, b, \lambda) = \frac{E_a(n/m, mb)}{m\, h(m, \lambda)} \tag{4.25}$$

where we have assumed that the energy required to select one word out of m is a very small part of $E_p(n, b, \lambda)$. The reduction in energy per access, ρ, is:

$$\rho = \frac{E_p(n, b, \lambda)}{E_a(n, b)} = \frac{E_a(n/m, mb)}{m\, h(m, \lambda) E_a(n, b)} \tag{4.26}$$

We use Eq. 2.41 to substitute E_a:

$$\rho = \frac{\log_2(n/m) + 2mb + 1}{m\, h(m, \lambda)\sqrt{m}(\log_2 n + 2b + 1)} \tag{4.27}$$

Minimizing ρ with respect to m gives the optimum size of the pre-fetch line. The equation that results is transcendental, and we cannot give a closed form expression. We can, however, determine numerically that if $n \gg m$, a good approximation for the optimum is $m = 1/(1-\lambda)$, and we get:

$$\rho \approx \frac{(1-\lambda)^{\frac{3}{2}}}{0.76} \tag{4.28}$$

For example, if in the execution of a program we have a branch instruction every eight instructions in average (that is, $\lambda = \frac{7}{8}$), we can expect an improvement of $\rho \approx 1/17$.

4.5 Summary & Conclusion

We have shown in this section some of the consequences of the correspondence between energy complexity of programs and energy dissipation of circuits. These translate into programming techniques to achieve low energy complexity and several trade-offs that can be made between energy complexity and time complexity.

Reactive programs come naturally as a programming style for CSP programs, and that is the reason why many circuits designed so far without low-energy performance in mind have very good energy characteristic. Reactive programs avoid wasting energy in busy-waiting loops.

Lazy programs postpone actions whose result may be invalidated by a later action as much as possible, until it is known that the result will be used. Lazy programming improves the energy efficiency of the algorithm at the expense of the delay performance. To recover some of the lost performance, we introduced the non-causal probe to look ahead in the future and determine whether the action will be needed. The non-causal probe is not implementable in general, but a good approximation can be implemented in many interesting cases. The non-causal probe corresponds, in a sense, to letting the control circuit run ahead of the data and, thus, increase the concurrency between the data operations.

Worst-case delay can be traded-off for energy dissipation without affecting the average-case delay much. Reducing the worst-case delay involves extra circuitry that increases the energy cost of the average case and does not necessarily improve in the same degree the performance of the system as a whole. In some examples, such as the zero-detection circuit, reducing the worst-case delay increases the energy cost and the average-delay and may degrade system performance.

Finally, we have shown that concurrency is an important source of efficiency. If there is some amount of data-independence in the input stream, it is convenient to replicate the hardware and split the input stream to keep delay performance at a lower energy cost. In this way, we trade-off area for energy cost.

Chapter 5

Energy/Delay Sizing

In this chapter we show that transistor sizing for minimum-energy and transistor sizing for minimum-delay are equivalent problems. We look at energy/delay models for optimum sizing, and how to simplify these models using local optimizations at the gate level, that do not alter the global optimum.

Transistor sizing is one of the final stages of the design before the physical layout. The purpose of transistor sizing is to select the size of all transistors in a circuit so that some measure of performance is optimized or some minimum requirement of that measure is met. Two problems have to be solved: first, choosing a relevant measure of performance; second, solving the optimization problem. Choosing a relevant measure of performance is far from trivial; asynchronous circuits have no clock period to optimize, and worse-case delays are not relevant if they are very different from average-case delays. Also, power dissipation varies substantially with the level of activity. A more elaborate measure of performance will be, in general, harder to evaluate and optimize. At the transistor level, the number of parameters to be calculated is very large (one per transistor), and computationally simple transistor models are not accurate for sub-micron CMOS technology.

As a consequence, transistor sizing requires some trade-offs between accuracy and computation time. The weak point of sizing methods such as [11,4, 27] is the use of the Elmore delay model [9]. In exchange, the global measure of performance is computationally simple, and the optimization problem turns out to be convex in many interesting cases. The accuracy of the energy/delay model can be increased significantly by using more elaborate transistor models. To keep the optimization problem simple, the number of parameters has to be reduced with local optimization — restrictions on the number of different transistor sizes, per-gate optimizations that do not affect the global optimum, use of symmetries in datapath circuits, etc. — or the global measure of performance has to be broken up so that the optimization can be performed independently on smaller sub-circuits. The global optimum can be replaced by a local, easy-to-compute optimum, leaving to the high-level design optimization the responsibility of eliminating bad local optima.

Transistor sizing is usually done to optimize delays. In general, minimum delays require infinitely large transistors (so that delays due to parasitic wiring capacitance are eliminated). To get an implementable solution, some constraints have to be added: area, power, or some other transistor-size related constraint. Other sizing strategies are possible. We can, for example, minimize the energy required for a given operation. Without any further constraints, for a CMOS circuit the optimum corresponds to all transistors being minimum size. To obtain a meaningful solution, we have to add a constraint, for example an upper bound to the delay, cycle time, or some other speed-related constraint.

5.1 Delay vs. Energy Optimization

Given a circuit C to be sized, we define two functions, $D(S)$, and $E(S)$; where $S = (s_1, \ldots, s_n)$ are the transistor sizes to be determined, $D(S)$ is the time required to perform a certain computation, and $E(S)$ is the energy required to perform the same computation. These are not two arbitrary functions; they are related by the circuit itself. We will assume that $D(S)$ and $E(S)$ have the following properties; these properties are derived from the physical nature of energy and delay and have to be verified for specific examples.

Property 5.1 *$D(S)$ and $E(S)$ are continuous functions of S.*

Property 5.2 *Delay can be traded-off for energy, and vice-versa. That is, given S and $\epsilon > 0$, there exist s and t such that $|s| < \epsilon$, $|t| < \epsilon$, and:*

1. $\epsilon > E(S + s) - E(S) > 0$
2. $\epsilon > D(S) - D(S + s) > 0$
3. $\epsilon > E(S) - E(S + t) > 0$

4. $\epsilon > D(S+t) - D(S) > 0$

The meaning of this property is that there are no flat regions anywhere on the $E(S)$ and $D(S)$ functions. The upper bound to the function variation is a consequence of the continuity of the energy and delay functions.

Consider, for example, as a function $D(S)$ the sum of Elmore delays in the critical cycle of an asynchronous circuit, and as $E(S)$ the sum of all transistor widths. These two functions are continuous, so property 5.1 is verified. If we multiply all transistor widths by a factor $\lambda > 1$, all delays will decrease since parasitic wiring capacitances become relatively less important; therefore, the cycle time will decrease and the sum of all transistor widths will increase. The converse happens when $\lambda < 1$; therefore, property 5.2 is verified as well.

We define next the delay and energy optimization problems.

Definition 5.1 *Given a delay function $D(S)$ and an energy function $E(S)$, and two positive numbers D_m and E_m, the delay optimization problem is to find S_0 such that $D(S)$ has a local minimum at $S = S_0$, with the constraint $E(S) \leq E_m$; the energy optimization problem is to find S_1 such that $E(S)$ has a local minimum at $S = S_1$, with the constraint $D(S) \leq D_m$.*

Theorem 5.1 *Given $D(S)$, $E(S)$, D_m, E_m, S_0, and S_1 as in the previous definition, we have $E(S_0) = E_m$ and $D(S_1) = D_m$.*

Proof: Assume that $E(S_0) < E_m$. From property 5.2, and using $\epsilon = E_m - E(S_0)$, there exists s such that $E(S_0 + s) < E_m$ and $D(S_0 + s) > D(S_0)$. This contradicts the definition of S_0. Therefore, $E(S_0) = E_m$. A similar proof holds for $D(S_1) = D_m$. ∎

Theorem 5.2 *The delay optimization problem is the dual of the energy optimization problem. That is, given $D(S)$, $E(S)$, D_m, and E_m, then*

1. *if S_0 is a solution to the delay optimization problem, it is also a solution to the energy optimization problem with $D_m = D(S_0)$.*
2. *if S_1 is a solution to the energy optimization problem, it is also a solution to the delay optimization problem with $E_m = E(S_1)$.*

Proof: Let S_0 be the solution to the delay optimization problem. From Th. 5.1 we know that $E_m = E(S_0)$. Then there exists $\delta > 0$ such that, for all s, if

$|s| < \delta$ and $E(S_0 + s) \le E(S_0)$, then $D(S_0 + s) \ge D(S_0)$. By contraposition, there exists $\delta > 0$ such that, for all s, if $|s| < \delta$ and $D(S_0 + s) < D(S_0)$, then $E(S_0 + s) > E(S_0)$.

We still need to know what happens if $D(S_0 + s) = D(S_0)$. Assume that there exists s, $|s| < \delta$, such that $D(S_0+s) = D(S_0)$ and $E(S_0+s) < E(S_0)$. We apply property 5.2 with $\epsilon = \min(\delta - |s|, E(S_0) - E(S_0+s))$. Then there exists t, $|t| < \epsilon$, such that $D(S_0+s+t) < D(S_0)$ and $E(S_0+s) < E(S_0+s+t) < E(S_0)$. But $|s + t| < \delta$, therefore $E(S_0 + s + t) > E(S_0)$. This contradicts the assumption and, therefore, implies the first claim. A similar proof holds for the second claim. ∎

Definition 5.2 *Given a delay function $D(S)$ and an energy function $E(S)$, the delay sizing function $S_D(S)$ is the function that assigns to S the closest solution to the delay minimization problem with the constraint $E_m \le E(S)$; the energy sizing function $S_E(S)$ is the function that assigns to S the closest solution to the delay minimization problem with the constraint $D_m \le D(S)$*[1]*.*

Theorem 5.3 *The following properties hold:*

1. *If $S_D(S)$ is defined, then $S_D(S_D(S)) = S_D(S)$.*
2. *If $S_E(S)$ is defined, then $S_E(S_E(S)) = S_E(S)$.*
3. *If $S_D(S)$ is defined, then $S_E(S_D(S)) = S_D(S)$.*
4. *If $S_E(S)$ is defined, then $S_D(S_E(S)) = S_E(S)$.*
5. *If $S_D(S)$ and $S_E(S)$ are defined, then $(S_D(S) = S_E(S)) \Leftrightarrow (S_E(S) = S \vee S_D(S) = S)$.*

Proof:

1. From the definition of $S_D(S)$ and Th. 5.1, we have $E(S_D(S)) = E(S)$. The constraint is the same; therefore, $S_D(S)$ is a valid solution for the new delay optimization problem and is the only solution at distance 0 from itself. ∎

2. Similar to 1.

[1] If more than one solution lies at the same distance, a suitable total order on S is used to choose one solution.

3. From the definition of $S_D(S)$ and Th. 5.2, $S_D(S)$ is a solution to the energy optimization problem with constraint $D(S) \leq D(S_D(S))$, and is the only solution at distance 0 from itself. ∎

4. Similar to 3.

5. ($\Rightarrow$) Assume $S \neq S_E(S)$. Then $E(S) > E(S_E(S))$; but from Th. 5.1 we have $E(S_D(S)) = E(S)$, and therefore $S_D(S) \neq S_E(S)$.

   ($\Leftarrow$) If $S_D(S) = S$, then $S_E(S_D(S)) = S_E(S)$; from 3 we have $S_D(S) = S_E(S)$. A similar proof holds for the case $S_E(S) = S$. ∎

The previous results show that energy minimization and delay minimization are very closely related, and techniques used to solve one problem can be used to solve the other. Delay minimization techniques try to provide a good delay function so that the sizing obtained corresponds to the best possible measured performance on the fabricated circuit. The energy function used in the constraint will not be, in general, an accurate prediction of the energy consumption of the fabricated circuit because some other numerical properties of the constraint are more important for the optimization algorithm (it is convenient that the constraint be computationally simple and a convex function of S).

If we want to guarantee that the optimal sizing results in a good measured energy performance, we have to provide the sizing algorithm with an accurate energy model. The added complexity will be reduced by doing some amount of local optimization or by simplifying the timing function, which now becomes a constraint.

5.2 Gate Modeling

The energy and delay functions studied in the previous section are very general; properties 5.1 and 5.2 are the only restrictions imposed on them. The names do suggest, however, that the delay function have some relation to the time performance of the circuit and that the energy function have some relation to the energy performance of the circuit.

In this section we narrow the choices on energy/delay functions by defining an energy delay model for individual gates in the circuit, and by considering only those energy/delay functions that can be built from the gate models in the circuit.

Gate-level modeling is convenient because it can be developed independent of the global timing analysis of the circuit. Also, in most design techniques, circuits are naturally divided into gates; the function of a gate can be abstracted and modeled, and the transistor circuit of the gate can be optimized for that particular function, thus reducing the total number of parameters.

The influence of a gate on the rest of the circuit is determined by the function it computes, and by the electrical interaction with other parts of the circuit. For CMOS gates, this electrical interaction is reduced to its input capacitance, delay per transition, and energy dissipation per transition. Given these parameters for each gate, we can predict the behavior of the circuit to some degree of accuracy, without knowledge of the actual transistor netlist.

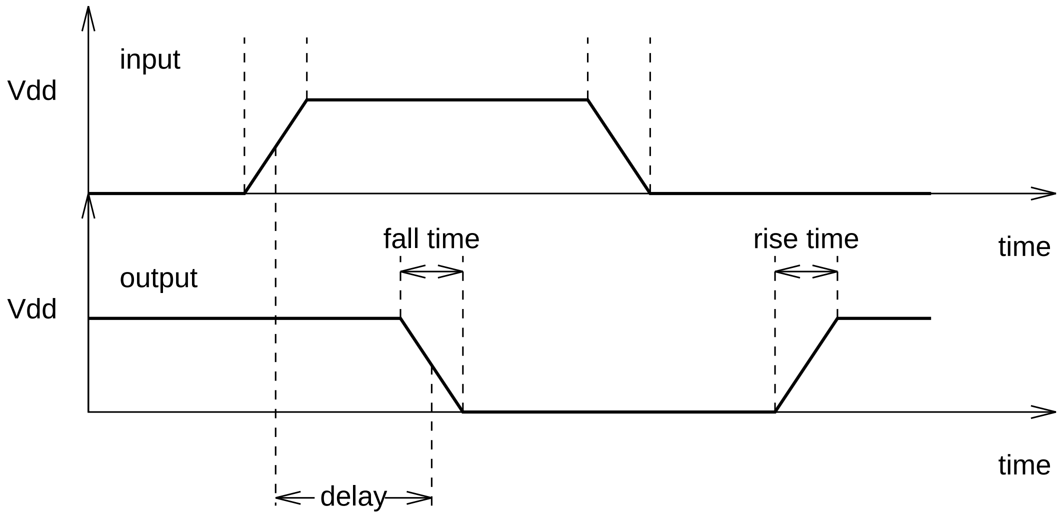


Figure 5.1: Input/output voltage relationships.

Gate delay is an abstraction of when and how the gate switches its output. The how is important in case the inputs to the gate are unstable (that is, the input condition that makes the gate switch is negated before the gate finishes switching). Instability generates transients whose behavior may be important to model with accuracy.

Quasi delay-insensitive (QDI) circuits, for example, do not have any unstable guards, except those appearing in arbiters. In these cases, the arbitration is solved by using a circuit similar to the one shown in Fig. 5.2. If both inputs $I1$ and $I2$ come in at the same time, the circuit formed by the cross-coupled nand gates may go into a metastable state; transistors $T1$ and $T2$ wait for the outputs of the nand gates to differ in at least one threshold voltage before producing an output, by which time the metastable state has been resolved into one of the two stable states.

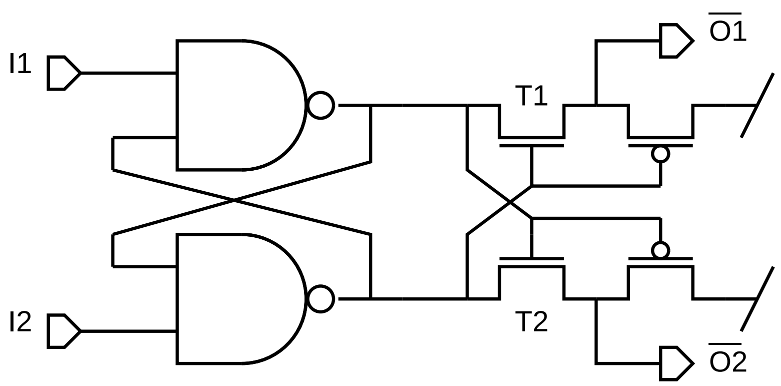


Figure 5.2: Arbiter Implementation for QDI circuits.

The delay model for QDI circuits has to guarantee the correct operation of metastable circuits. However, since arbiters are used very sparingly and can be identified easily from the circuit specification, it is more convenient to use a simpler delay model for the rest of the circuit and a specific delay model for arbiters.

The delay on a gate depends on the transistor circuit for that gate, the output load, and the shape of the input signal. Of all, input signal shape is the hardest to quantize; the order in which inputs occur and the rise time of those inputs, all seem to affect the actual shape of the output signal. To account for most of those effects some assumptions have to be made about "reasonable" input shapes. Since inputs are the outputs of other gates, input and output shapes have to be similar. From the stability requirement we know that there will be no runts as outputs, that is, all signals are allowed to make full transitions.

Several models for CMOS delay estimation have been proposed in the past. These models vary in the way that the delay through the gate is computed, and in the nature of the circuit that is assumed to generate that delay. We can classify those models in two categories: those that model the behavior of individual transistors and those that model the behavior of individual gates.

5.2.1 Transistor Modeling

The most accurate way to compute gate delay is to solve the transistor equations for each gate in the same way that, for example, SPICE2 [24] simulates a circuit. This type of transistor modeling will generate the right answer with very few assumptions about the input signal. Even though this approach can be used for small circuits, its computational complexity is very high, especially if we use it as a model for global sizing optimization.

Increasing the accuracy of the equations that model the transistor does not necessarily improve the overall accuracy of the prediction. More precise equations rely on a larger number of process dependent parameters that may not be very well known or that may have large variations from one fabrication run to the next. Also, because the modeling is done to a finer level of detail, the effect of parasitic capacitors, resistors, and inductances becomes noticeable, and therefore these parasitics have to be extracted or predicted with better accuracy.

Adding constraints to the input shape, we can simplify the transistor equations and still get a fairly accurate prediction of the timing function. The transistor network of the gate can be replaced by an RC-network plus ideal

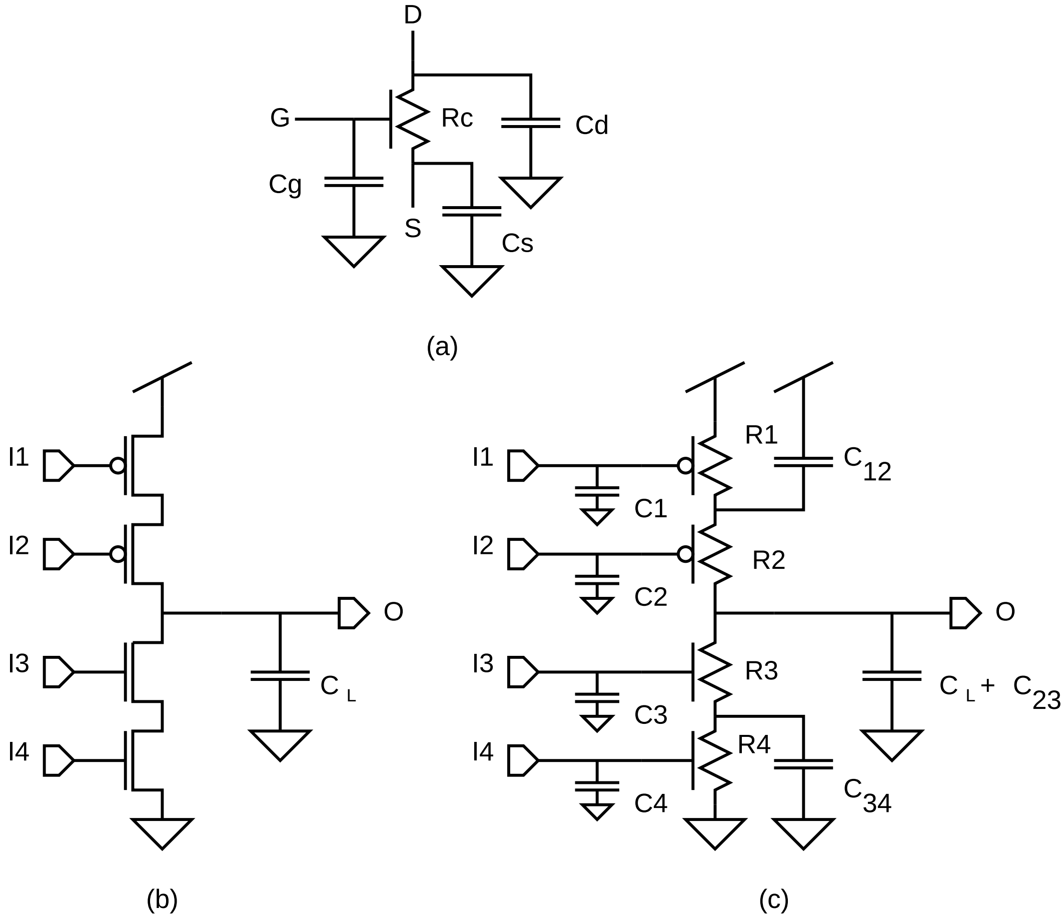


Figure 5.3: *RC*-model for CMOS gates. (a) transistor model, (b) CMOS gate, and (c) equivalent *RC*-network model for that gate.

switches, as in Fig. 5.3. The source, drain, and gate capacitances, Cs, Cd, and Cg, are proportional to the transistor channel width, while the channel resistance, Rc, is inversely proportional to the channel width. Based on this model, it is possible to make predictions on the upper and lower bound to the signal delay through the *RC*-network [26].

The *RC*-model for transistors is not very accurate for submicron devices because of velocity-saturation effects. The relationship between transistor sizes and timing parameters becomes very complex and is not suitable for global optimization.

5.2.2 Macro Modeling

Gate modeling, or macro modeling [23,25,6], attempts to reduce the complexity of the optimization problem by providing timing models for logic gates. All internal nodes to the gate are eliminated, and the gate is replaced by an equivalent and simpler electrical model.

Gate modeling can be done in many different ways. In [23], complex gates are mapped into simple gates as inverters. Sizes are computed and optimized on those inverters and then mapped back to the original gate. The reduction in complexity corresponds to the reduction in the number of degrees of freedom of the complex gate, to the number of degrees of freedom of the inverter. The accuracy of the method is dependent on the accuracy of the mapping function. A similar approach is followed by [30], where the concept of logical effort [29] is used to separate the logic function that the gate is computing from the physical modeling of that gate with an inverter. This type of modeling is more adequate for asynchronous design because it can deal better with feedback loops.

In [25], a look-up table approach is used. The delay computation is tabulated for different gate types and configurations. This method is very accurate, but it requires large tables to be useful, especially if each gate has several parameters. The size of the look-up table can be reduced by combining the effect of some of the parameters into scaling laws.

The method presented in [15] derives the delay equations from the non-linear equations for the transistor circuit, and the theoretical parameters are adjusted from SPICE simulations. These equations are developed for simple combinational gates (nand, nor, AOI, OAI, inverters) under all possible triggering conditions. The resulting model is very accurate (to within 10% of SPICE), but it is restricted to those gates for which a model was derived.

5.2.3 Gate Model for Optimum-Energy Sizing

Gate models have traditionally been optimized for very accurate delay prediction at the expense of the energy function. This is an acceptable strategy if the purpose of optimization is to obtain the best delays at a reasonable price in energy dissipation. The energy measure most commonly employed is the sum of all transistor sizes. This energy measure is positively correlated to the total energy consumption, to the area of the circuit (in the case of a datapath, not true for control logic), and to the probability of transistor failure (assuming that fabrication flaws are the most critical on transistor gates); it also has the fundamental property of being a posynomial[2] on the transistor widths and, therefore, a convex function of the logarithms of the transistor widths [11].

When the purpose of the optimization is to minimize the energy dissipation in absolute terms, we have to use a better model. On Ch. 2 we have shown that the energy consumption of a CMOS circuit could be approximated by the equation:

$$E_T = (K_L + K_S V_{DD})\, V_{DD}^2 \tag{5.1}$$

The same equation is valid for individual gates or individual transitions in each gate. We can derive K_L and K_S for the circuit, adding the coefficients corresponding to each of the transitions occurring in the operation of the circuit. If there are n different transitions in the circuit, with K_{Li}, K_{Si} being the coefficients corresponding to the i^{th} transition, and f_i being the relative frequency of the i^{th} transition, we have:

$$\begin{aligned} E_T &= \sum_{i=1}^{n} f_i\,(K_{Li} + K_{Si} V_{DD})\, V_{DD}^2 \\ &= \left(\sum_{i=1}^{n} f_i K_{Li} + \sum_{i=1}^{n} f_i K_{Si} V_{DD}\right) V_{DD}^2 \end{aligned} \tag{5.2}$$

$$K_L = \sum_{i=1}^{n} f_i K_{Li} \tag{5.3}$$

$$K_S = \sum_{i=1}^{n} f_i K_{Si} \tag{5.4}$$

Observe that the f_i's can be computed independently of the scenario chosen for the timing simulation. This way, the energy function can correspond to average energy dissipation, while the timing function corresponds to a specific simulation. This is especially interesting for circuits with data dependencies, where not all gates are used with the same frequency and, therefore, do not contribute in the same way to the energy dissipation.

Next, we assume that the delay function for the circuit is based on the

[2] A posynomial is a polynomial with positive coefficients, positive variables, and integer exponents

timed simulation of the circuit [4]. In that case, the other gate parameters that are relevant to the sizing problem are delay, transition time, and input capacitance. Delays are used to compute the timing function. To insure that isochronic forks do not misfire, we have to impose maximum restrictions on transition times; finally, input capacitance plays an important role in the interaction between gates.

Each gate will have at least two different transitions, one up and one down. More transitions per gate are possible: transitions are identified by the input condition that causes them, or the production rule that fires, or the transistor path that ties the output node to ground or V_{DD}, etc. Identifying the transitions is part of the logic specification of the gate.

The sizes of the transistors in a gate can be fixed arbitrarily within certain limits that ensure that the gate works correctly. These transistor sizes represent the degrees of freedom of the gate; we can, however, choose a different set of independent variables that allow us to compute the transistor sizes and that are more convenient in the context of the optimization problem.

To summarize, the characteristics of the gate model gate are:

1. An output load, C_L.
2. m degrees of freedom, $k_1, \ldots, k_m$.
3. p inputs, $I_1, \ldots, I_p$.
4. For each input I_i, an input capacitance, $C_i(k_1, \ldots, k_m)$.
5. n transitions, $T_1, \ldots, T_n$.
6. For each transition, T_j;
   (a) An energy equation, $E_j(k_1, \ldots, k_m, C_L)$.
   (b) A delay equation, $D_j(k_1, \ldots, k_m, C_L)$.

The independent variables $k_1, \ldots, k_m$ may not have physical meaning; the input and output capacitances and energy and delay functions, however, do, and we can derive some properties for these functions from their physical meaning.

Property 5.3 *The gate delay and energy functions are continuous functions of $k_1, \ldots, k_m, C_L$, and differentiable functions of C_L, and verify the following equations:*

1. $\frac{\partial E_j}{\partial C_L} > 0$
2. $\frac{\partial D_j}{\partial C_L} > 0$

Again, these properties have to be verified for the specific energy and delay function chosen to do the optimization.

5.2.4 Parameter Reduction

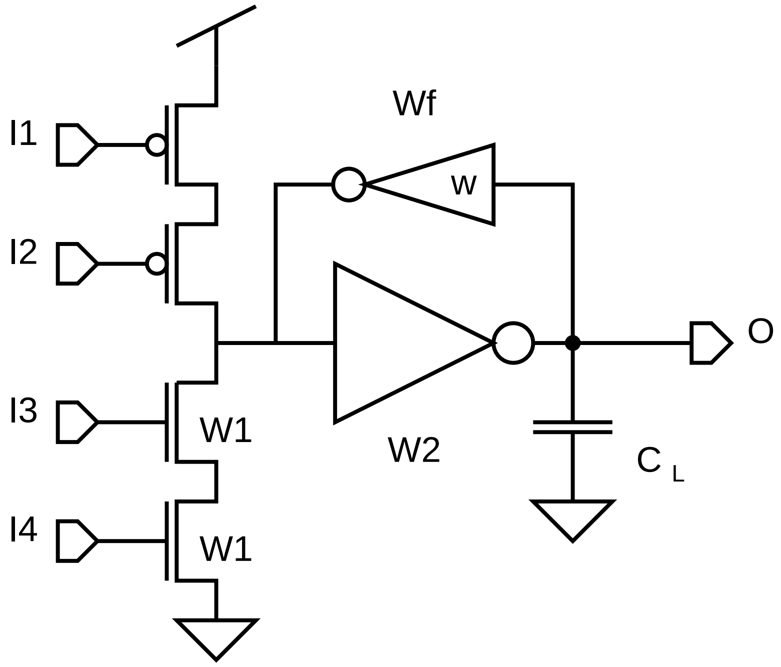


Figure 5.4: Two-level circuit for a generalized C-element with weak inverter feedback.

Consider the two-stage implementation for a generalized C-element shown in Fig. 5.4. To simplify the example, we assume that all transistors in the same pull-up or pull-down chain have the same width. The up-going transition corresponding to the production rule $I3 \wedge I4 \rightarrow O\uparrow$ can be modeled with 3 parameters, $W1$, $W2$, and Wf, plus the output load C_L. This gate has more degrees of freedom than we need; given an input capacitance and a delay, there are many assignments to the gate parameters that achieve that input capacitance and delay. Nevertheless, the only interesting assignment is the one that minimizes the energy function.

We show next that the number of parameters can always be reduced to, at most, two per transition, plus the output load C_L, without altering the global optimum.

Consider the function,

$$U_j(k_1, \ldots, k_m, C_L) = \Big(E_j(k_1, \ldots, C_L), D_j(k_1, \ldots, C_L), C_j(k_1, \ldots), C_L\Big) \quad (5.5)$$

where E_j, D_j, and C_j are are the energy, delay, and input capacitance function for a particular transition, T_j. Let $\mathcal{D}_j$ and $\mathcal{R}_j$ be the domain and range of U_j. We assume that $\mathcal{D}_j$ is a closed set. We further assume that $\mathcal{R}_j$ is a closed set[3] and that both $\mathcal{D}_j$ and $\mathcal{R}_j$ are connected sets. We show next that not all points in $\mathcal{R}_j$ are interesting.

Definition 5.3 *We define the following functions:*

1. $f_E(D, C, C_L) = \min_{(E,d,c,c_L)\in\mathcal{R}_j,\, d\leq D,\, c\leq C,\, c_L\geq C_L} E$
2. $f_D(E, C, C_L) = \min_{(e,D,c,c_L)\in\mathcal{R}_j,\, e\leq E,\, c\leq C,\, c_L\geq C_L} D$
3. $f_C(E, D, C_L) = \min_{(e,d,C,c_L)\in\mathcal{R}_j,\, e\leq E,\, d\leq D,\, c_L\geq C_L} C$
4. $f_{C_L}(E, D, C) = \max_{(e,d,c,C_L)\in\mathcal{R}_j,\, e\leq E,\, d\leq D,\, c\geq C} C_L$

Theorem 5.4 *Let $P = (E, D, C, C_L) \in \mathcal{R}_j$ be an optimum solution to the energy-sizing problem. Then we have:*

$$(E, D, C, C_L) =$$

[3] If the optimization procedure selects a point in the closure of $\mathcal{R}_j$ without a pre-image in $\mathcal{D}_j$, then, given the granularity of the actual implementation of the gate, and the continuity of U_j, we can map that point into another close by point in the interior of $\mathcal{R}_j$.

$$\Big(f_E(D,C,C_L), f_D(E,C,C_L), f_C(E,D,C_L), f_{C_L}(E,D,C)\Big) \quad (5.6)$$

Proof: Assume that $E > f_E(D,C,C_L)$. Then we can choose the parameters $k_1,\ldots,k_m$ such that the transition T_j achieves a better energy dissipation, $f_E(D,C,C_L)$, with an equal or better delay, an equal or better input capacitance, under output load that is either equal or more strict. This new selection of parameters verifies the constraints of the global optimization problem, and has a smaller contribution in energy. It is, therefore, a better solution, and we can replace the transition T_j with this new transition.

A similar argument holds for D, C, and C_L. ∎

Theorem 5.5 *Given C_L, two more parameters are required, at most, to describe a transition that is a solution to the global optimization problem.*

Proof: Given C_L and the parameters D and C, we know, from Th. 5.4, $E = f_E(D,C,C_L)$. C_L, D, and C are enough to describe the set of optimum solutions; other more interesting parameter selections are possible. ∎

Fig. 5.5 shows a parametric representation of the relationship between E, D, and C, under optimum conditions, for a two-stage generalized C-element. Two parameters plus the output load are necessary in this case.

5.2.5 Posynomial Interpolation

The reduction in the number of parameters is important, because it allows us to use tables derived from accurate (SPICE) simulations in the sizing algorithm. However, even with only three parameters left, the tables can still be fairly large if we desire good accuracy — unless we have an efficient way of interpolating into that table.

Many sizing algorithms rely on expressing the energy/delay functions as posynomials of the transistor widths. It is advantageous, therefore, to use posynomials as interpolating functions, to be able to use the same sizing algorithms. The table becomes a table of posynomial coefficients.

We start with a function $f(k_1,\ldots,k_m)$, that we want to approximate with a posynomial of which we know everything but the coefficients, λ_I's:

$$f(k_1,\ldots,k_m) \approx \mathrm{P}(k_1,\ldots,k_m) = \sum_{0<I\leq q} \lambda_I k_1^{\alpha_I^1} \ldots k_m^{\alpha_I^m} \quad (5.7)$$

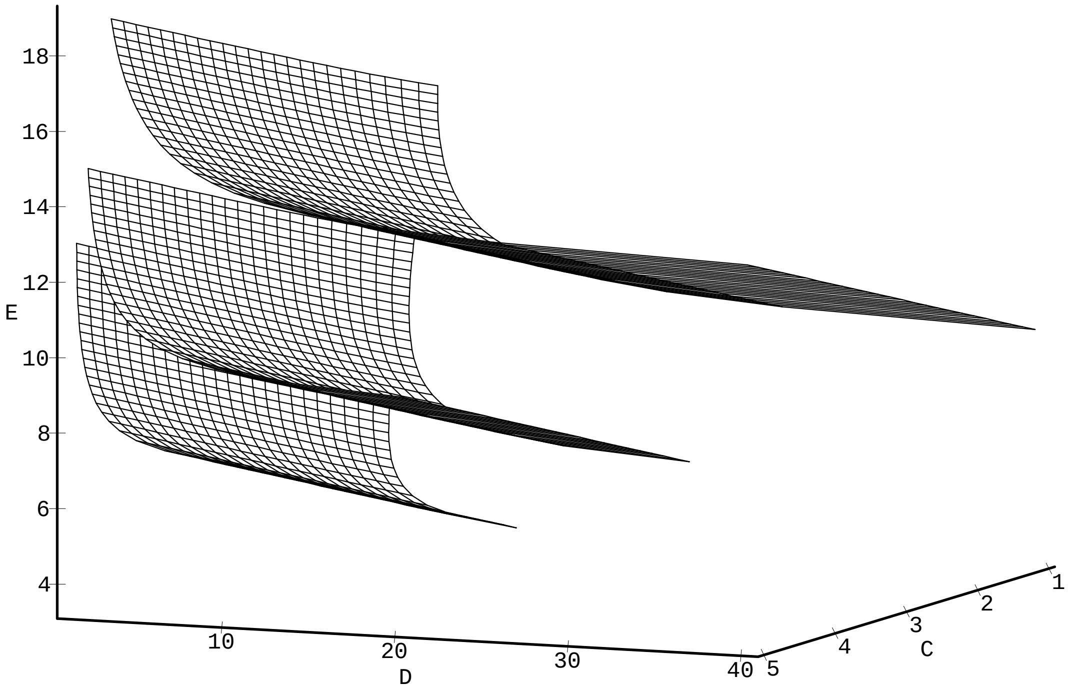


Figure 5.5: E-D-C surface for a two-stage generalized C-element, for several values of C_L.

We need to model the value of the function and the functions derivatives with accuracy. It is the derivatives of the function that specify the trade-off between the different parameters of the gate. To this effect, we compute the λ_I's by equating the differentials of f and P to as high a degree as it is necessary to make the system of equations determinate. We end up with a system of linear equations on the λ_I's:

$$\begin{pmatrix} f \\ \frac{\partial f}{\partial k_1} \\ \vdots \\ \frac{\partial^2 f}{\partial k_1^2} \\ \vdots \end{pmatrix} = \begin{pmatrix} k_1^{\alpha_1^1} \dots k_m^{\alpha_1^m} & \cdots & k_1^{\alpha_q^1} \dots k_m^{\alpha_q^m} \\ \alpha_1^1 k_1^{\alpha_1^1 - 1} \dots k_m^{\alpha_1^m} & \cdots & \alpha_q^1 k_1^{\alpha_q^1 - 1} \dots k_m^{\alpha_q^m} \\ \vdots & \ddots & \vdots \\ \alpha_1^1(\alpha_1^1 - 1) k_1^{\alpha_1^1 - 2} \dots k_m^{\alpha_1^m} & \cdots & \alpha_q^1(\alpha_q^1 - 1) k_1^{\alpha_q^1 - 2} \dots k_m^{\alpha_1^m} \\ \vdots & \ddots & \vdots \end{pmatrix} \begin{pmatrix} \lambda_1 \\ \vdots \\ \lambda_q \end{pmatrix} \tag{5.8}$$

or, in matrix notation,

$$\mathbf{D} = \mathbf{A}\Lambda \tag{5.9}$$

$\mathbf{D}$ is a $h \times 1$ matrix and is either measured or derived from a SPICE simulation. $\mathbf{A}$ is a $h \times q$ matrix and is derived from the theoretical posynomial model for the gate (Elmore delay model, for example). Λ is a $q \times 1$ matrix and the unknown to be determined. Because we have to take whole differentials for the posynomial approximation, the value of h is not arbitrary but has some fixed values, $h = 1+m, 1+m+\frac{1}{2}m(m-1), \ldots$. In general, we will have $h > q$, and the system will be overdetermined. This means that we will not be able to find Λ such that $\mathbf{D} = \mathbf{A}\Lambda$. We define the error vector of the approximation, $\mathbf{E} = \mathbf{D} - \mathbf{A}\Lambda$, and the error $\epsilon = ||\mathbf{E}||$. We choose Λ so that ϵ is minimized. Since $\mathbf{A}\Lambda \in \text{Image}(\mathbf{A})$, this minimum is achieved when $\mathbf{E}$ is orthogonal to Image($\mathbf{A}$) (see Fig. 5.6).

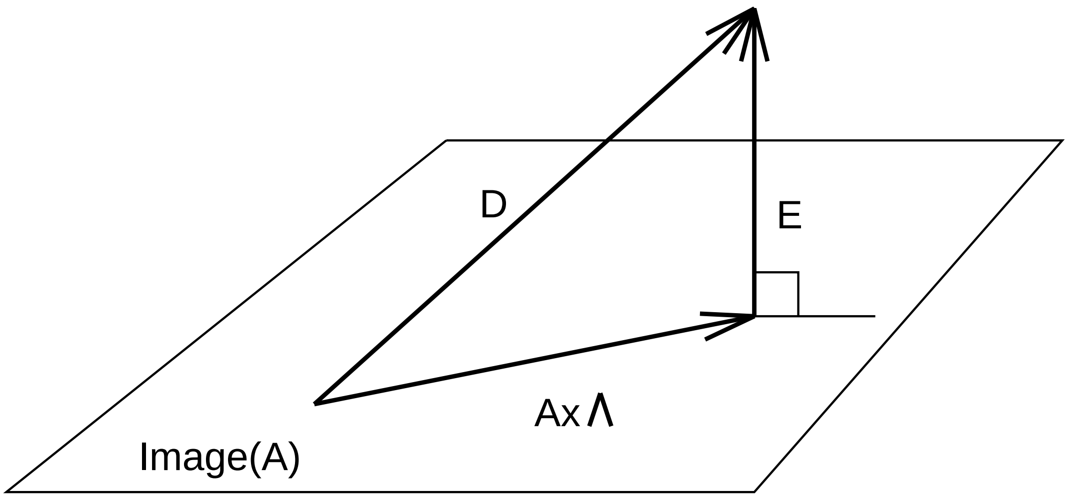


Figure 5.6: Graphical representation of the error vector. The norm of the error vector is minimized when $\mathbf{A}\Lambda$ is the orthogonal projection of $\mathbf{D}$ on Image($\mathbf{A}$).

From the orthogonality relation, we have $\mathbf{A}^T\mathbf{E} = \mathbf{0}$; therefore,

$$\mathbf{A}^T\mathbf{E} = \mathbf{A}^T\mathbf{D} - \mathbf{A}^T\mathbf{A}\Lambda = \mathbf{0} \tag{5.10}$$

If $\mathbf{A}\Lambda = \mathbf{D}$ is overdetermined, the matrix $\mathbf{A}^T\mathbf{A}$ is invertible, and we have:

$$\Lambda = \left(\mathbf{A}^T\mathbf{A}\right)^{-1}\mathbf{A}^T\mathbf{D} \tag{5.11}$$

The vector norm used so far is $\|\mathbf{X}\| = \sqrt{\mathbf{X}^T\mathbf{X}}$. Other vector norms can be used if we want to assign different weights to the error contributions from different equations. In general, if $\mathbf{N}^T\mathbf{N}$ is a positive definite matrix[4], we can define the associated norm $\|\mathbf{X}\|_{\mathbf{N}} = \sqrt{\mathbf{X}^T\mathbf{N}^T\mathbf{N}\mathbf{X}}$. In that case, Eq. 5.11 becomes:

$$\Lambda = \left(\mathbf{A}^T\mathbf{N}^T\mathbf{N}\mathbf{A}\right)^{-1}\mathbf{A}^T\mathbf{N}^T\mathbf{N}\mathbf{D} \tag{5.12}$$

Observe that if $\mathbf{A}$ is invertible, Eq. 5.12 becomes $\Lambda = \mathbf{A}^{-1}\mathbf{D}$, as would be expected.

The posynomial approximation for the energy/delay can be used for sizing in a number of ways, depending whether we pre-compute the approximation as a table, or we compute the approximation after some preliminary sizing has been done. In both cases, the objective is to increase the accuracy of the sizing algorithm by adjusting the gate parameters around a point close to the optimum. This reduction in the domain of validity of the model is what allows us to provide a model that is very accurate for those particular circuit configurations. In the next two subsections we describe these two sizing methods.

5.2.6 Sizing with a Pre-Computed Table

Many sizing algorithms rely on choosing from a fixed set of gate sizes. There are good reasons to do so; with a reduced number of gates we can do extensive simulations on those gates, or characterize them after fabrication. In this way, the characteristics of those gates are very well known, and the energy/delay functions can be computed with a high degree of accuracy. The gates can be taken from well-tested library, which increases the reliability of the circuit.

[4] A positive definite matrix is a symmetric matrix with positive eigenvalues.

For control-type circuits that are going to be laid-out in a "standard-cell place-and-route" style, the wiring capacitance becomes a larger part of the load capacitance (compared to gate capacitance), as the circuit becomes larger [8]. As a consequence, the placement algorithm becomes more critical than sizing. This, together with the use of two-stage gates for generalized C-elements, greatly reduces the sensitivity of the delay with respect to individual gate sizes.

Assigning a discreet number of sizes to the gates in a circuit, however, converts a continuous optimization problem into an integer optimization problem, and it is, therefore, computationally harder. The algorithms described in [4] to compute the performance of asynchronous circuits, generate an integer linear programming problem in this case, which only has exponential-time known solutions.

The table can be used to bridge the gap between the integer and continuous optimization problem. Given a solution to the integer optimization problem, we can improve this solution by using the posynomial approximation to the energy/delay functions around that solution. Since the energy/delay functions are going to be approximated around a few points, the posynomial coefficients can be pre-computed and tabulated.

There are a number of advantages to this approach:

Because the table is based on numbers obtained from very precise simulations or measurements, we are not dependent on the accuracy of simplified theoretical models used to reduce the complexity of the computation.

The integer programming problem is used to compute an initial solution. If the table is large enough, this initial solution will be close to the global optimum, and we prevent the continuous optimization algorithm from getting stuck in a bad local optimum.

It is expected that the continuous solution will not be very far away from the integer solution. If this is the case, the posynomial approximation of the energy/delay functions will be fairly accurate.

5.2.7 Sizing with a Post-Computed Approximation

The main drawback of the previous approach is the complexity of the integer optimization problem. The size of the input can be fairly large — several hundreds of gates — which may require that we use small tables (three or four different gate sizes) to obtain results in a reasonable amount of time.

A different approach is to obtain the initial solution by solving the continuous optimization problem. We then compute the posynomial approximation around this point and re-compute the optimum with the new model. Computing the posynomial approximation is numerically intensive, but computation time scales linearly with the number of gates, instead of exponentially.

Also, with the pre-sizing for the initial solution we can generate layout that will not be very different from the final solution. From this layout we can obtain a more accurate prediction of the parasitics and use them for the second round of sizing.

5.3 Summary & Conclusion

We have shown in this section that low-energy sizing can be approached the same way as low-delay sizing. In fact, both problems are dual of each other. This fact underlines the necessity to treat both energy consumption and delay at the same time; transformations that improve performance can be used to improve energy dissipation, and vice-versa.

Typically, a great effort is spent in correctly modeling the delay characteristic of gates. To obtain a meaningful energy optimization, energy consumption has to be modeled better. Global energy measures — as compared to delay measures — are easy to compute based on energy models for gates, since there is no temporal interaction to be accounted for. We can compute the energy dissipation of a circuit as the sum of the energy dissipation of the gates in the circuit, while circuit delay requires a much more complicated operation on the delays of the gates of the circuit. We have directed the effort in this chapter to the better modeling of CMOS gates, and base on those models any circuit-level optimization.

Given that only a limited number of properties from the gate are interesting (input capacitance, energy/delay characteristic), we have shown that the number of degrees of freedom in each gate can be reduced so that the gate characteristics are optimized locally without altering the global optimum. This allows us to reduce the complexity of the global optimization problem by doing some of the optimization at the gate level and, in this way, reducing the total number of parameters in the circuit.

The traditional posynomial function modeling for gates can be used as a first-order approximation for gate delays. To improve the accuracy of such a model, we have proposed a way of fitting a posynomial function to the gate properties to be modeled. The posynomial function is based on the Elmore-delay model for the gate; the coefficients of the posynomial are fitted using least-squares approximation for the best available model for the gate and its differentials (a SPICE model, for example). This approach is directed towards either table-based sizing algorithms, where the posynomial approximation can be used to reduce the size of the table, or continuous sizing algorithms, where the posynomial approximation is used to refine the accuracy of the energy model after some preliminary sizing has been done.

Chapter 6

Datapath Techniques

In this chapter we study how to minimize the energy dissipation of datapath circuits, using efficient implementations for the more common datapath constructs. We look into circuits for register-to-register transfers and for multiple sender/multiple receiver buses.

Algorithmic low-energy techniques try to minimize the number of energy-consuming steps required to perform a computation, in particular, register-to-register transfers, and function evaluation. The next step in improving the energy efficiency of the circuit, is to come up with efficient, circuit-level implementations for the building blocks of asynchronous design.

6.1 Register-to-Register Transfers

Register-to-register transfers account for a large part of the energy dissipation in processor-type circuits. Self-timed register transfers require either spacer tokens and completion signals or cumbersome logic to implement the more efficient (i.e. fewer transitions per symbol) protocols. We can considerably reduce the complexity of the logic by giving up some of the self-timed properties of the code and by introducing timing assumptions.

In this section we make an analysis of the following encodings: dual-rail, one-hot, and bundled data. For each type of transfer, we calculate an index based on the total amount of charge that has to be spent in the transfer. This charge is computed using the following assumptions:

All transistors have the same size, and their gate has one unit of capacitance (minimum-size transistors will result in minimum energy for the transfer).

The charge in the diffusion area of each transistor can be assimilated to the charge in the gate and, therefore, will not be counted.

Short-circuit currents are small compared to dynamic currents and will not be counted.

The energy cost index will be computed for an n-bit register-to-register transfer.

6.1.1 Dual-Rail/One-Hot Encoding

Dual-rail, four-phase encoding results in the simplest implementation of registers and boolean functions, even though not always the smallest implementation. Dual-rail encoding uses two wires to transmit one bit (one for each value); one-hot encoding uses one wire for each value to be transmitted. In between data values, all wires are reset to the neutral state; this is a self-timed code. In practice, only one-of-two, one-of-four, and one-of-eight codes are useful; larger codes have too much overhead or are used only for very specific functions (e.g., array decoders).

The dual-rail four-phase protocol is used as a reference to evaluate the other protocols and to illustrate how the energy index is obtained. The handshaking expansion of the dual-rail four-phase protoco is as follows:

$$
\begin{array}{l}
\mathit{SENDER} \equiv *[[\ \neg x \wedge r \longrightarrow\ b0\uparrow;\ [\neg r];\ b0\downarrow \\
\qquad\qquad\quad [\!]\ x \wedge r \ \longrightarrow\ b1\uparrow;\ [\neg r];\ b1\downarrow \\
\qquad\qquad\ ]]
\end{array}
$$

$$
\begin{array}{l}
\mathit{RECEIVER} \equiv *[[\ b0 \longrightarrow\ y0\downarrow;\ a\uparrow;\ [\neg b0];\ a\downarrow \\
\qquad\qquad\qquad [\!]\ b1 \longrightarrow\ y0\uparrow;\ a\uparrow;\ [\neg b1];\ a\downarrow \\
\qquad\qquad\quad\ ]]
\end{array}
$$

Fig. 6.1 shows a transistor circuit for a dual-rail, four-phase channel. From this diagram and the handshaking expansion we can derive the cost per bit, as follows:

3 gates for the $rw\_$ signal,

4 gates for the b_i signal,

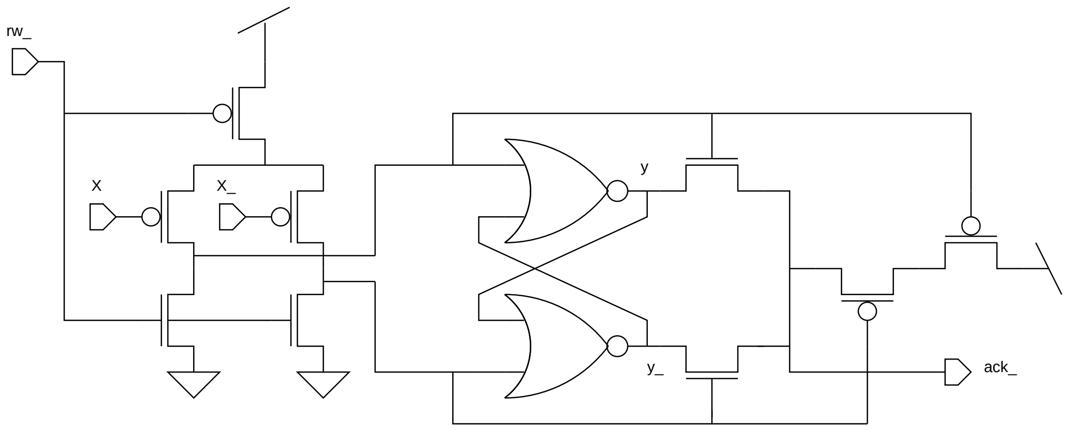


Figure 6.1: Transistor circuit for a dual-rail, four-phase channel.

2 gates for the y signal, half of the time,

2 gates for the $ack\_$ signal,

$2(n-1)$ gates for the completion tree,

Total: $10n + 2(n-1) = 12n - 2$.

We look next at the one-of-four encoding. The handshaking expansion for this protocol is as follows:

$$
\begin{array}{lllll}
\mathit{SENDER} \equiv *[[& \neg x0 \wedge \neg x1 \wedge r & \longrightarrow & b0\uparrow; & [\neg r];\ b0\downarrow \\
& [\!]\ x0 \wedge \neg x1 \wedge r & \longrightarrow & b1\uparrow; & [\neg r];\ b1\downarrow \\
& [\!]\ \neg x0 \wedge x1 \wedge r & \longrightarrow & b2\uparrow; & [\neg r];\ b2\downarrow \\
& [\!]\ x0 \wedge x1 \wedge r & \longrightarrow & b3\uparrow; & [\neg r];\ b3\downarrow \\
&]] & & &
\end{array}
$$

$$
\begin{array}{lllll}
\mathit{RECEIVER} \equiv *[[& b0 & \longrightarrow & y0\downarrow, y1\downarrow; & a\uparrow;\ [\neg b0];\ a\downarrow \\
& [\!]\ b1 & \longrightarrow & y0\uparrow, y1\downarrow; & a\uparrow;\ [\neg b1];\ a\downarrow \\
& [\!]\ b2 & \longrightarrow & y0\downarrow, y1\uparrow; & a\uparrow;\ [\neg b2];\ a\downarrow \\
& [\!]\ b3 & \longrightarrow & y0\uparrow, y1\uparrow; & a\uparrow;\ [\neg b3];\ a\downarrow \\
&]] & & &
\end{array}
$$

We derive the production rules for the *SENDER* and *RECEIVER*:

$$
\begin{array}{lcl}
\neg x_0 \wedge \neg x_1 \wedge r & \to & b_0\uparrow \\
\neg r & \to & b_0\downarrow
\end{array}
\qquad
\begin{array}{lcl}
b_1 \vee b_3 & \to & y_0\uparrow \\
b_0 \vee b_2 & \to & y_0\downarrow
\end{array}
$$

$$
\begin{array}{lcl}
x_0 \wedge \neg x_1 \wedge r & \to & b_1\uparrow \\
\neg r & \to & b_1\downarrow
\end{array}
\qquad
\begin{array}{lcl}
b_2 \vee b_3 & \to & y_1\uparrow \\
b_0 \vee b_1 & \to & y_1\downarrow
\end{array}
$$

$$
\begin{array}{lcl}
\neg x_0 \wedge x_1 \wedge r & \to & b_2\uparrow \\
\neg r & \to & b_2\downarrow \\
 & & \\
x_0 \wedge x_1 \wedge r & \to & b_3\uparrow \\
\neg r & \to & b_3\downarrow
\end{array}
\qquad
\begin{array}{lcl}
b_0 \wedge \neg y_0 \wedge \neg y_1 & \to & a\uparrow \\
b_1 \wedge y_0 \wedge \neg y_1 & \to & a\uparrow \\
b_2 \wedge \neg y_0 \wedge y_1 & \to & a\uparrow \\
b_3 \wedge y_0 \wedge y_1 & \to & a\uparrow \\
\neg b_0 \wedge \neg b_1 \wedge \neg b_2 \wedge \neg b_3 & \to & a\downarrow
\end{array}
$$

These production rules are not directly implementable[1]. Depending on the choices we make for the registers x_0, x_1, y_0, and y_1, we get very different energy costs.

On the sender side, the x registers can be implemented either as two separate registers with cross-coupled inverters or as double registers (quad-flop). The generation of the b_i signals can be done in one stage, requiring more power from the control line, or in two stages, generating an extra transition.

If we use quad-flops, the implementable production rules are as follows:

$$
\begin{array}{lcl}
x_{00} \wedge r & \to & b_{0}\_\downarrow \\
\neg r & \to & b_{0}\_\uparrow \\
 & & \\
x_{01} \wedge r & \to & b_{1}\_\downarrow \\
\neg r & \to & b_{1}\_\uparrow
\end{array}
\qquad
\begin{array}{lcl}
x_{10} \wedge r & \to & b_{2}\_\downarrow \\
\neg r & \to & b_{2}\_\uparrow \\
 & & \\
x_{11} \wedge r & \to & b_{3}\_\downarrow \\
\neg r & \to & b_{3}\_\uparrow
\end{array}
$$

The control line r goes to two and a half transistors per bit (which is better than the three transistors per bit for the dual-rail implementation), and the pull-down chains are two transistors long.

If we use two registers instead of one quad-flop, we get:

[1] That is, extra inverters are needed to implement this circuit with CMOS logic.

$$
\begin{array}{lcllcl}
x_{0f} \wedge x_{1f} \wedge r & \rightarrow & b_0\_\!\downarrow & \quad x_{0f} \wedge x_{1t} \wedge r & \rightarrow & b_2\_\!\downarrow \\
\neg r & \rightarrow & b_0\_\!\downarrow & \quad \neg r & \rightarrow & b_2\_\!\downarrow \\
\\
x_{0t} \wedge x_{1f} \wedge r & \rightarrow & b_1\_\!\downarrow & \quad x_{0t} \wedge x_{1t} \wedge r & \rightarrow & b_3\_\!\downarrow \\
\neg r & \rightarrow & b_1\_\!\downarrow & \quad \neg r & \rightarrow & b_3\_\!\downarrow
\end{array}
$$

The control signal r still goes to two and a half transistors per bit, but now the pull-down chains are three transistors long, requiring larger transistors. Fig. 6.2 shows a transistor schematic for this circuit. Even though a quad-flop looks like a better solution for the source register, the choice between flip-flop and quad-flop can be more strictly determined by the other write ports that the source register may have.

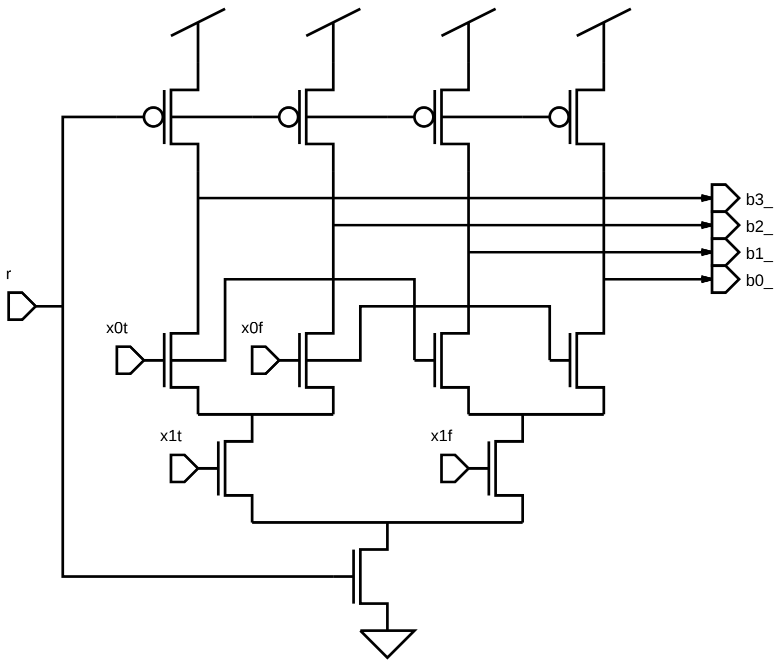


Figure 6.2: Best transistor circuit for one-of-four sender, one stage.

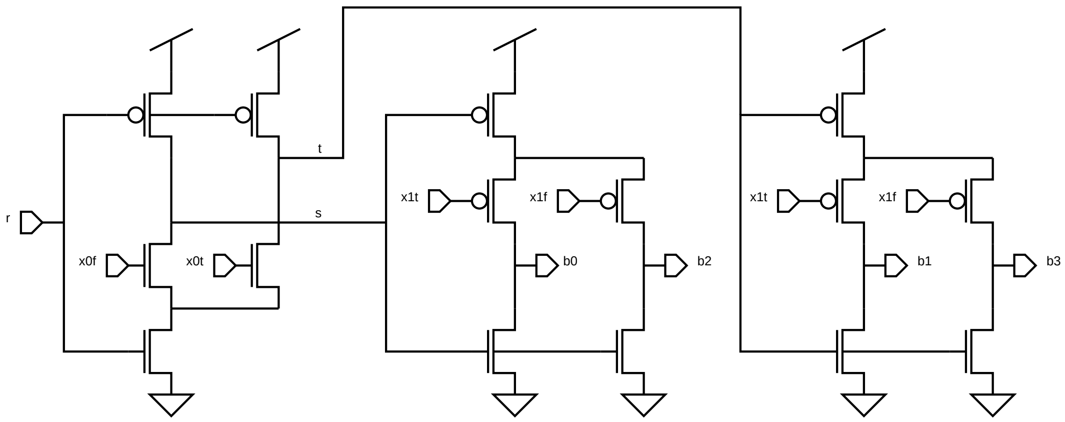


Figure 6.3: Best transistor circuit for one-of-four sender, two stages.

We can improve on this design by generating the b_i signals in two stages:

$$
\begin{array}{lcl@{\qquad}lcl}
\neg x_{1t} \wedge \neg s & \rightarrow & b_0\uparrow & \neg x_{1f} \wedge \neg t & \rightarrow & b_3\uparrow \\
s & \rightarrow & b_0\downarrow & t & \rightarrow & b_3\downarrow \\
 & & & & & \\
\neg x_{1t} \wedge \neg t & \rightarrow & b_1\uparrow & x_{0f} \wedge r & \rightarrow & s\downarrow \\
t & \rightarrow & b_1\downarrow & x_{0t} \wedge r & \rightarrow & t\downarrow \\
 & & & \neg r & \rightarrow & s\uparrow, t\uparrow \\
\neg x_{1f} \wedge \neg s & \rightarrow & b_2\uparrow & & & \\
s & \rightarrow & b_2\downarrow & & &
\end{array}
$$

The control signal r goes only to one and a half transistors per bit, and these transistors can be made smaller because they have to switch just the local signals s and t. An extra one and a half transistors are switched per bit by the second stage. Also, the pull-up and pull-down chains are just two transistors long, and the channel uses positive logic, which results in a better implementation of the receiver side. The price to be paid is in the two extra transitions on the variables s or t, making this implementation as expensive as the dual-rail circuit. Fig. 6.3 shows a transistor schematic for this circuit. Also, this scheme scales well for a one-of-eight code or for a lazy-active datapath where one extra transistor is required in the pull-up and pull-down chains:

$$
\begin{array}{lcl}
\neg x_{1t} \wedge \neg s \wedge \neg b_2 & \rightarrow & b_0\uparrow \\
s & \rightarrow & b_0\downarrow \\
\\
\neg x_{1t} \wedge \neg t \wedge \neg b_3 & \rightarrow & b_1\uparrow \\
t & \rightarrow & b_1\downarrow \\
\\
\neg x_{1f} \wedge \neg s \wedge \neg b_0 & \rightarrow & b_2\uparrow \\
s & \rightarrow & b_2\downarrow
\end{array}
$$

$$
\begin{array}{lcl}
\neg x_{1f} \wedge \neg t \wedge \neg b_1 & \rightarrow & b_3\uparrow \\
t & \rightarrow & b_3\downarrow \\
\\
x_{0f} \wedge r \wedge t & \rightarrow & s\downarrow \\
x_{0t} \wedge r \wedge s & \rightarrow & t\downarrow \\
\neg r & \rightarrow & s\uparrow, t\uparrow
\end{array}
$$

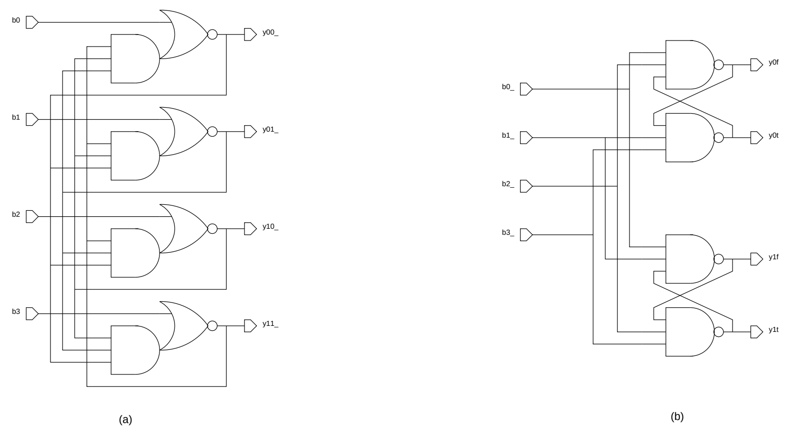


Figure 6.4: (a) Quad-flop with one write port, positive inputs; and (b) double flip-flop, one write port, negative inputs. Completion circuit not shown.

On the receiver side, the y registers can be implemented either as a single double register (quad-flop) or as two separate registers (see Fig. 6.4).

The production rules for the quad-flop are:

$$
\begin{array}{lcl}
b_0 \vee (y_{1-} \wedge y_{2-} \wedge y_{3-}) & \rightarrow & y_{0-}\downarrow \\
\neg b_0 \wedge (\neg y_{1-} \vee \neg y_{2-} \vee \neg y_{3-}) & \rightarrow & y_{0-}\uparrow \\
\\
b_1 \vee (y_{0-} \wedge y_{2-} \wedge y_{3-}) & \rightarrow & y_{1-}\downarrow \\
\neg b_1 \wedge (\neg y_{0-} \vee \neg y_{2-} \vee \neg y_{3-}) & \rightarrow & y_{1-}\uparrow \\
\\
b_2 \vee (y_{1-} \wedge y_{0-} \wedge y_{3-}) & \rightarrow & y_{2-}\downarrow \\
\neg b_2 \wedge (\neg y_{1-} \vee \neg y_{0-} \vee \neg y_{3-}) & \rightarrow & y_{2-}\uparrow \\
\\
b_3 \vee (y_{0-} \wedge y_{2-} \wedge y_{1-}) & \rightarrow & y_{3-}\downarrow \\
\neg b_3 \wedge (\neg y_{0-} \vee \neg y_{2-} \vee \neg y_{1-}) & \rightarrow & y_{3-}\uparrow
\end{array}
$$

$$
\begin{array}{lcl}
b_0 \wedge y_{1-} \wedge y_{2-} \wedge y_{3-} & \rightarrow & ack_{-}\downarrow \\
b_1 \wedge y_{0-} \wedge y_{2-} \wedge y_{3-} & \rightarrow & ack_{-}\downarrow \\
b_2 \wedge y_{1-} \wedge y_{0-} \wedge y_{3-} & \rightarrow & ack_{-}\downarrow \\
b_3 \wedge y_{1-} \wedge y_{2-} \wedge y_{0-} & \rightarrow & ack_{-}\downarrow \\
\neg b_0 \wedge \neg b_1 \wedge \neg b_2 \wedge \neg b_3 & \rightarrow & ack_{-}\uparrow
\end{array}
$$

To write a new value into the quad-flop, six gates have to be switched three quarters of the time, or two and a quarter gates per bit. This implementation of the quad-flop, shown in Fig. 6.4(a), does not scale well for a large number of write ports due to the complex write-acknowledge gate. It is possible to simplify this gate by making a reasonably safe timing assumption on the values of the y variables and replacing the acknowledge gate by:

$$
\begin{array}{lcl}
b_0 \wedge \neg y_{0-} \vee b_1 \wedge \neg y_{1-} \vee b_2 \wedge \neg y_{2-} \vee b_3 \wedge \neg y_{3-} & \rightarrow & ack_{-}\downarrow \\
\neg b_0 \wedge \neg b_1 \wedge \neg b_2 \wedge \neg b_3 & \rightarrow & ack_{-}\uparrow
\end{array}
$$

This gate is smaller than the previous one, requires only one extra connection per y variable, is faster, scales well to a one-of-eight code, and can be broken up easily in two gates. Fig. 6.5 shows three different transistor circuits for this gate. Fig. 6.5(c) is the fastest of the three (all transistor chains are two-long), and (b) has the lowest transistor count but is very slow for positive-logic b_i signals.

The production rules for the double flip-flop are:

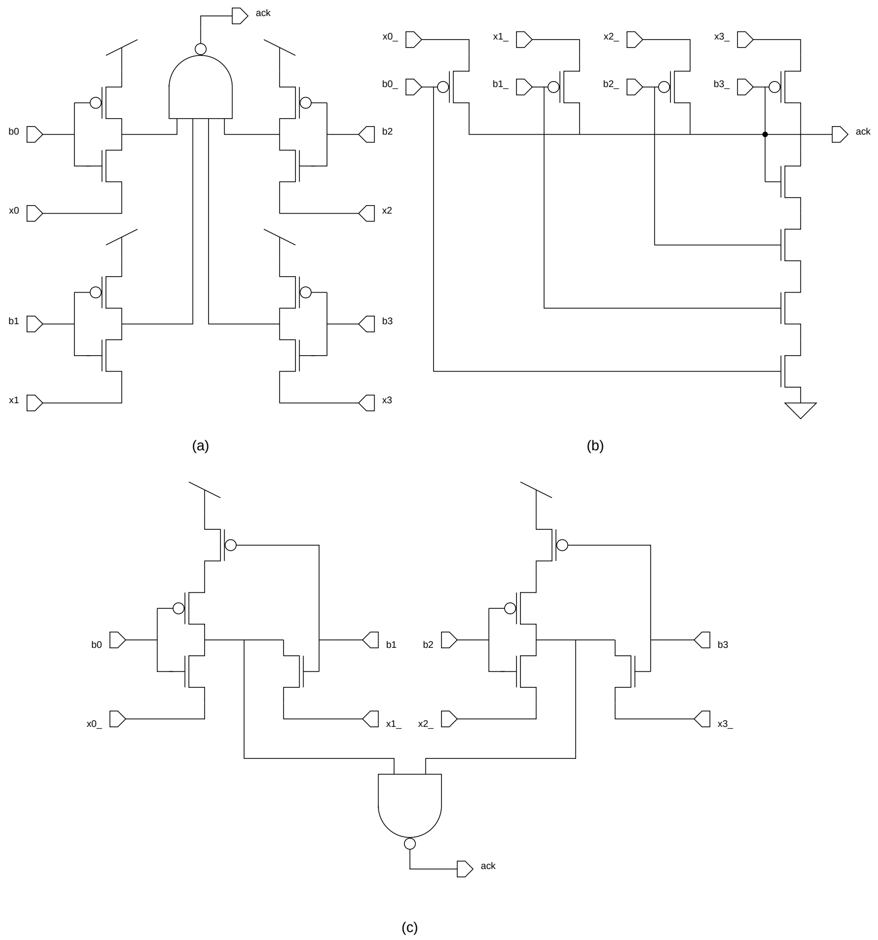


Figure 6.5: Three possible transistor circuits for the non-safe acknowledge of a quad-flop.

$$
\begin{aligned}
\neg b_{0\_} \vee \neg b_{2\_} \vee \neg y_0 t &\quad \rightarrow \quad y_0 f\uparrow \\
b_{0\_} \wedge b_{2\_} \wedge y_0 t &\quad \rightarrow \quad y_0 f\downarrow
\end{aligned}
$$

$$
\begin{aligned}
\neg b_{1\_} \vee \neg b_{3\_} \vee \neg y_0 f &\quad \rightarrow \quad y_0 t\uparrow \\
b_{1\_} \wedge b_{3\_} \wedge y_0 f &\quad \rightarrow \quad y_0 t\downarrow
\end{aligned}
$$

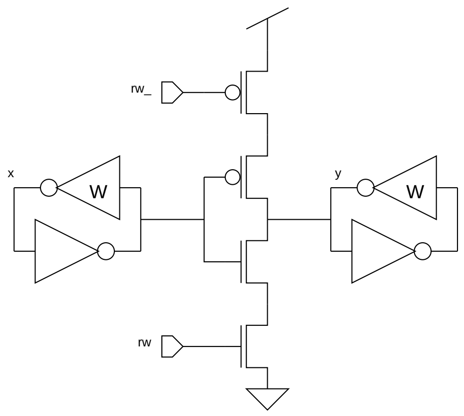
$$
\begin{array}{lcl}
\neg b_{0\_} \vee \neg b_{1\_} \vee \neg y_1 t & \rightarrow & y_1 f\uparrow \\
b_{0\_} \wedge b_{1\_} \wedge y_1 t & \rightarrow & y_1 f\downarrow \\
 & & \\
\neg b_{2\_} \vee \neg b_{3\_} \vee \neg y_1 f & \rightarrow & y_1 t\uparrow \\
b_{2\_} \wedge b_{3\_} \wedge y_1 f & \rightarrow & y_1 t\downarrow \\
(\neg b_{0\_} \vee \neg b_{2\_}) \wedge \neg y_0 t & \rightarrow & ack_0\uparrow \\
(\neg b_{1\_} \vee \neg b_{2\_}) \wedge \neg y_0 f & \rightarrow & ack_0\uparrow \\
b_{0\_} \wedge b_{1\_} \wedge b_{2\_} \wedge b_{3\_} & \rightarrow & ack_0\downarrow \\
 & & \\
(\neg b_{0\_} \vee \neg b_{1\_}) \wedge \neg y_1 t & \rightarrow & ack_1\uparrow \\
(\neg b_{2\_} \vee \neg b_{3\_}) \wedge \neg y_2 f & \rightarrow & ack_1\uparrow \\
b_{0\_} \wedge b_{1\_} \wedge b_{2\_} \wedge b_{3\_} & \rightarrow & ack_1\downarrow
\end{array}
$$

The double flip-flop, shown in Fig. 6.4(b) has the advantage of representing both bits separately, which is convenient if the register is going to have read and write ports of different types. Also, this register converts the one-of-four code back to dual rail. Two write acknowledge signals are generated, one for each flip-flop, and the gates are more complex than for a standard one-bit register. It turns out that the energy saved from having two less transitions on the data wires does not make up for the extra complexity of the acknowledge, and the capacitance of the data wires is higher due to the higher fan-in of this register.

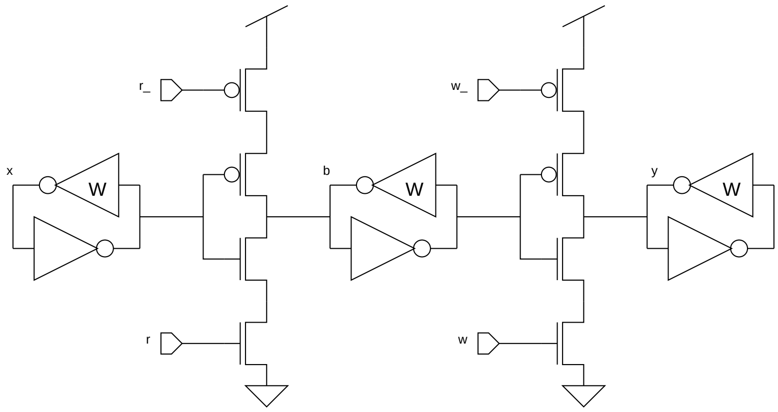
6.1.2 Bundled-Data

The bundled data protocol relies on timing information from a control signal to determine the validity of the data on the channel wires. Only one wire is needed per bit; if the bus is not precharged, only half of the wires, in average, will have to switch for each data transmission. This protocol has, therefore, one-fourth as many data-wire transitions than the dual-rail protocol and half as many data-wire transitions as the one-of-four protocol. It does, however, violate the self-timed properties of the circuit; we have to be sure that we get an energy improvement in return and assertain that the circuit timing is not compromised critically.

A simple implementation of a bundled data transfer is shown in Fig. 6.6. It requires two separate control signals for the read circuit, but each signal



(a)



(b)

Figure 6.6: Bundled data, register transfer circuit, non precharged. (a) Direct copy; (b) indirect copy.

goes to only one transistor per bit. This circuit is ideally suited for micropipeline style design [31]. The transfer can be done either in one step or two. The one-step transfer is simpler and faster, but it requires that both registers be placed close to each other. If this is not possible (because of the other read/write ports that these registers may have), we can use the two-step transfer shown in Fig. 6.6(b). The value of the source register is first copied to the intermediate variable b and then copied from b to the destination. The indirect copy makes the destination register into a master-slave register and allows for better pipelining of the operation at a cost in energy.

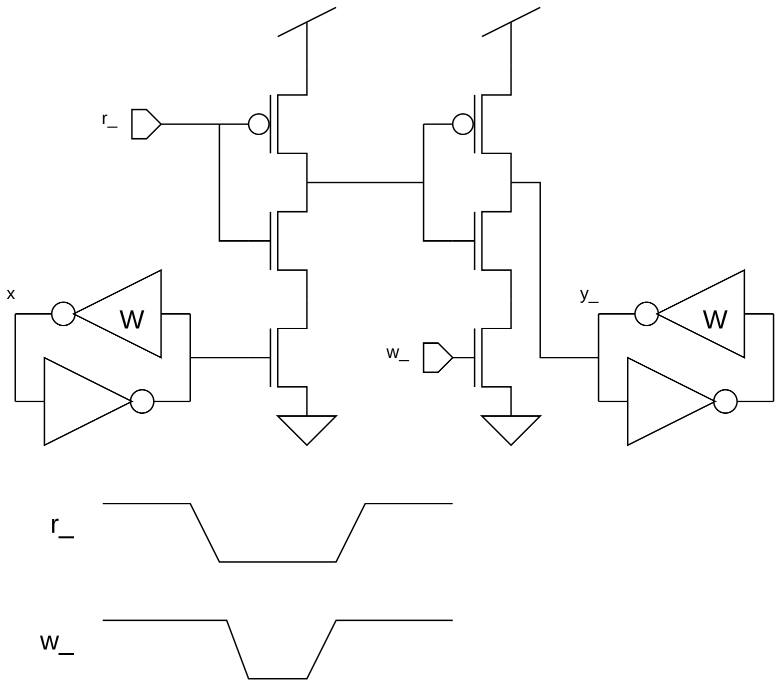


Figure 6.7: Bundled data register-to-register transfer circuit, precharged.

An alternative implementation of the bundled data protocol is with a precharged intermediate variable, as shown in Fig. 6.7. The precharged protocol has one transition per bit, on average, and is very efficient if the data transfer includes function evaluation. The write signal can be derived from the read signal using the circuit of Fig. 6.8. This circuit generates a fixed-width pulse. The timing assumption is that this width is enough to complete the write action. The width of the write pulse is controlled by the delay element in the feedback path.

6.2 Buses

A bus is an efficient way of reducing the number of point-to-point communication channels in an architecture with a large number of registers. Transfers over buses are more expensive than point-to-point transfers, but using buses results in simpler registers — fewer read and write ports — thus reducing the

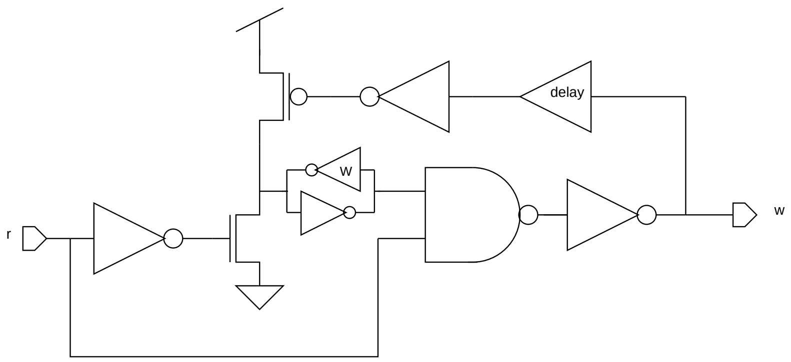


Figure 6.8: Write pulse generation from the read pulse for a precharged register-to-register transfer.

overall cost of each assignment.

A bus is a communication channel with at least three ports connected to it. Buses come in three flavors: single-sender, multiple-receiver; multiple-sender, single-receiver; and multiple-sender, multiple-receiver. We will assume that access to the bus is mutually exclusive, so no arbitration is necessary.

A bus is different from a point-to-point channel in the way the design scales with the number of senders and receivers. A large number of receivers increases the speed and area penalty of completion detection; a large number of senders increases the speed penalty of the data transfer.

6.2.1 Multiple-Sender Channel

The capacitance of the bus wires of a multiple sender channel scales linearly with the number of senders on the channel and, therefore, the delay and energy dissipation on the bus will also increase with the number of senders. Increasing the size of the driving transistors increases, in the same proportion, the capacitance of the buses, with little net gain on the delay.

6.2.2 Bus with Sense-Amplifier

A standard solution to improving delays on a bus is to use sense-amplifiers. Sense-amplifiers are usually very expensive in power, unless they are switched-

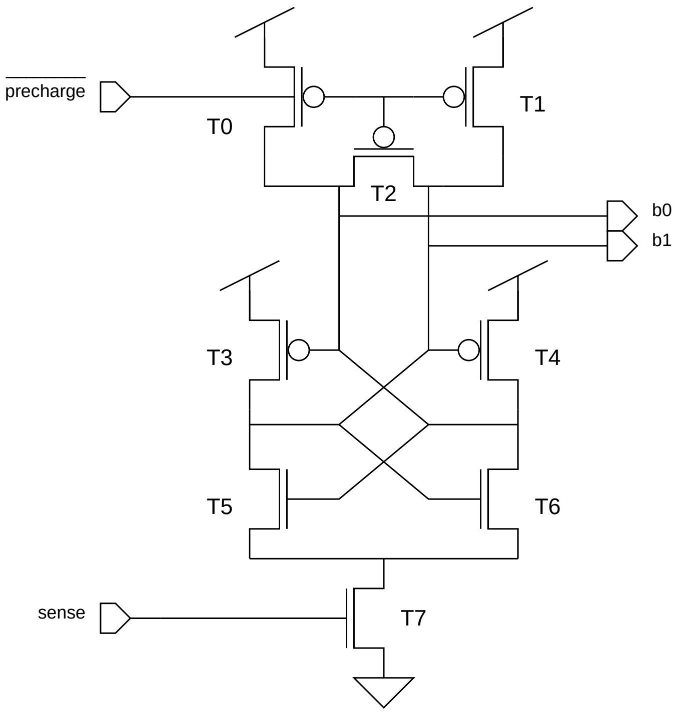


Figure 6.9: Sense-amplifier circuit for asynchronous buses.

off while not in use. Fig. 6.9 shows the circuit of a switched sense-amplifier. When *sense* and $\overline{precharge}$ are low, the buses are precharged high and equalized. When *sense* and $\overline{precharge}$ are high, the sense amplifier behaves like two cross-coupled inverters, and switches to one of its stable states. The sender trying to write into this channel has biased the bus wires towards the value to be sent, making this state the preferred equilibrium state of the amplifier.

Thanks to the sense-amplifier, we can use small transistors as bus driving circuits. As a consequence, several sources of energy dissipation become smaller: the node capacitance of the sending register is reduced, and it is, therefore, cheaper and faster to write into this register; the select signal for the sending register goes to small transistors; the registers themselves will be smaller in area, making the length of the bus wires shorter.

The main energy cost associated with this sense amplifier is the short-circuit currents during the sense operation. If the timing of $\overline{precharge}$ and *sense* is not correct, the amplifier can go into a metastable state, with a high cost in energy, and potentially generate the wrong output.

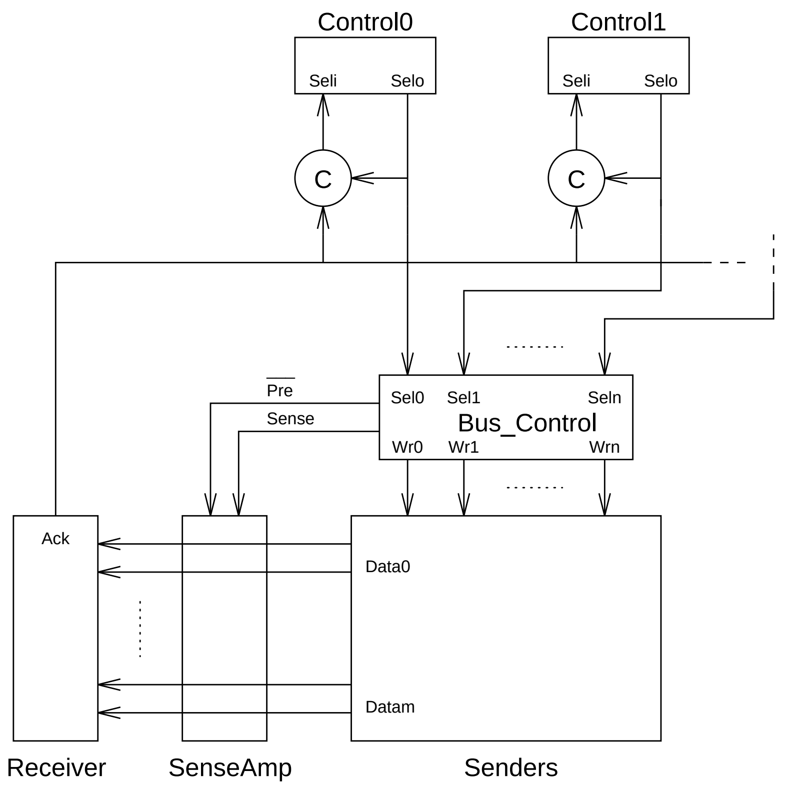


Figure 6.10: Signal interconnection for a non-pipelined asynchronous bus with sense-amplifier.

The following handshaking expansion gives a possible implementation for the bus control that guarantees the proper operation of the sense-amplifier. Signal Sel_j selects register j to write into the bus; signal Wr_j allows register j to write onto the bus; signal $\overline{Pre}$ precharges the buses when low; and signal $Sense$ triggers the sense-amplifier circuit. Fig. 6.10 shows how signals are interconnected.

$BUS\_CONTROL \equiv$

$$
\begin{array}{l}
*\texttt{[[}\ \langle\ \texttt{[]}\ j:\ \ Sel_j \longrightarrow\ \ \overline{Pre}\uparrow; Wr_j\uparrow; Sense\uparrow; \\
\qquad\qquad\qquad\qquad\qquad \texttt{[}\neg Sel_j\texttt{]}; Wr_j\downarrow; Sense\downarrow; \overline{Pre}\downarrow\ \ \rangle \\
\ \texttt{]]}
\end{array}
$$

$$
\begin{array}{lcl}
Wr_0 \lor \ldots \lor Wr_n & \rightarrow & Sense\uparrow \\
\neg\, Wr_0 \land \ldots \land \neg\, Wr_n & \rightarrow & Sense\downarrow
\end{array}
$$

$$
\begin{array}{lcl}
\overline{Pre} \land Sel_j & \rightarrow & Wr_j\uparrow \\
\neg Sel_j & \rightarrow & Wr_j\downarrow
\end{array}
$$

$$
\begin{array}{lcl}
Sel_0 \lor \ldots \lor Sel_n & \rightarrow & \overline{Pre}\uparrow \\
\neg Sel_0 \land \ldots \land \neg Sel_n \land \neg Sense & \rightarrow & \overline{Pre}\downarrow
\end{array}
$$

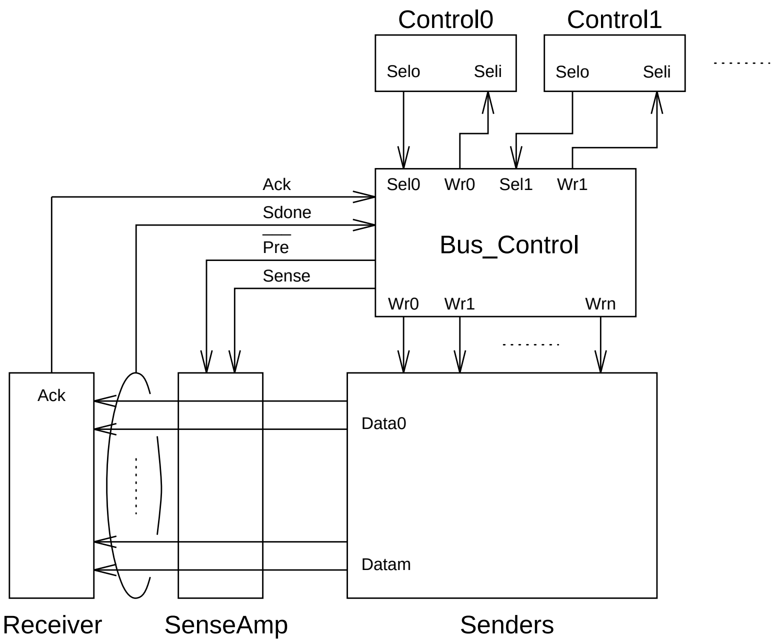


Figure 6.11: Signal interconnection for a pipelined asynchronous bus with sense-amplifier.

6.2.3 Pipelined Bus Transfer

The cycle time of the previous scheme can be improved by observing that the sense-amplifier behaves like a register. We can use this register to pipeline the data transfer, thus decoupling sender and receiver and increasing the concurrency of the circuit. To use the sense-amplifier as a register, we have to add a completion signal, *Sdone*, to the bus, to determine whether the bus is in a valid state or the neutral state. A possible handshaking expansion for this type of transfer follows. Fig. 6.11 shows how signals are interconnected

$$
\begin{array}{l}
BUS\_PIPE \equiv \\
\quad *[[\ \langle\ []\ j:\ \neg Sdone\ \wedge\ Sel_j \longrightarrow \overline{Pre}\uparrow; [\neg ack]; Wr_j\uparrow; Sense\uparrow; \\
\qquad\qquad\qquad\qquad\qquad [Sdone \wedge \neg Sel_j]; Wr_j\downarrow; [ack]; Sense\downarrow; \overline{Pre}\downarrow \\
\qquad \rangle \\
\quad]]
\end{array}
$$

$$
\begin{array}{lcl}
(Sel_0 \vee \ldots \vee Sel_n) \wedge \neg Sdone & \rightarrow & \overline{Pre}\uparrow \\
\neg W \wedge \neg Sense & \rightarrow & \overline{Pre}\downarrow
\end{array}
$$

$$
\begin{array}{lcl}
\neg Sense \wedge \neg ack \wedge \overline{Pre} \wedge Sel_j & \rightarrow & Wr_j\uparrow \\
Sdone \wedge \neg Sel_j & \rightarrow & Wr_j\downarrow
\end{array}
$$

$$
\begin{array}{lcl}
Wr_0 \vee \ldots \vee Wr_n & \rightarrow & W\uparrow \\
\neg Wr_0 \wedge \ldots \wedge \neg Wr_n \wedge ack & \rightarrow & W\downarrow
\end{array}
$$

$$
\begin{array}{lcl}
W & \rightarrow & Sense\uparrow \\
\neg W \wedge ack & \rightarrow & Sense\downarrow
\end{array}
$$

With this reshuffling, the data will be held on the bus until the receiver is ready to read it in. The signal *Sdone* is required to maintain mutual exclusion in the access to the bus resource. A slack of one is created between the sender and receiver, so care has to be taken that this does not affect the correctness of the circuit.

Other reshufflings are possible, depending on whether the sender or the receiver is active or passive. The bus can be used as an active-active converter, allowing the designer to choose the gender of the channel on both ends independently. Some of this options will be explored further in the implementation of a multiple sender, multiple receiver channel.

6.2.4 Multiple-Receiver Channel

In a multiple-receiver channel, the generation of the completion signal is especially inefficient because it has to be implemented as yet another bus transfer. One completion tree per register is more efficient in energy because only one bus wire is required to gather all the completion signals from each register (instead of one bus wire per bit).

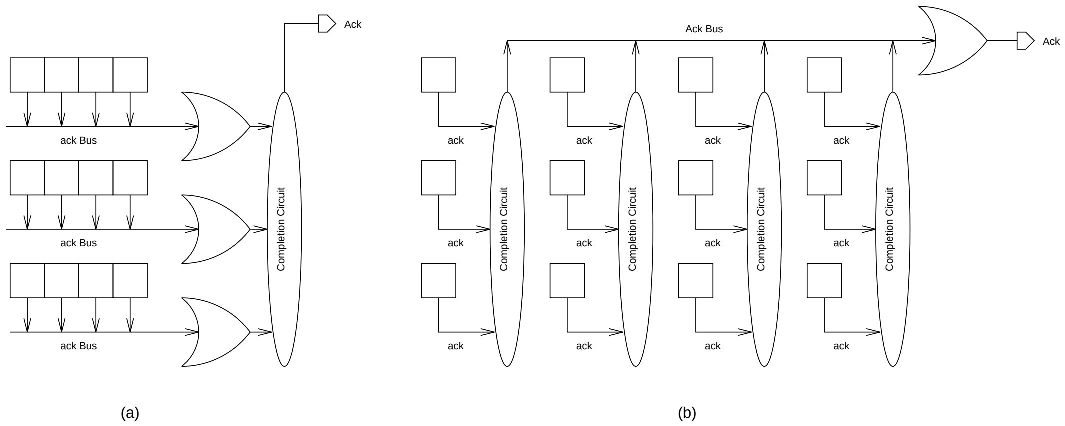


Figure 6.12: Safe completion schemes for multiple receiver channels. (a) shared completion tree, and (b) local completion tree.

6.2.5 Safe Completion

Fig. 6.12 shows two basic schemes for the generation of the completion signal. The first alternative (Fig. 6.12 (a)) is to have a single completion tree per channel and share the inputs to the completion tree using acknowledge buses. This scheme minimizes the amount of circuitry required, but it is slow and expensive in energy because of the extra bus transfer. We can make an estimate of the energy cost of completion to compare to other schemes. If the channel has m receivers, and n bits per receiver, then the cost of completion can be estimated as:

$$E_c^a = K_b mn + K_T(n) \tag{6.1}$$

where K_b is a proportionality constant for the energy required for a bus transition, and $K_T(n)$ is the cost of an n input completion tree.

The second alternative (Fig. 6.12 (b)) is to provide each register with a completion tree and gather the output of the completion trees in a bus. The amount of circuitry required for completion increases considerably; however, only one or-gate is required to compute completion. The cost of computing completion can be estimated as:

$$E_c^b = K_b n + K_T(n) + K_o(m) \tag{6.2}$$

where K_o(m) is the cost of an m input or-gate. This or-gate can be placed outside of the datapath. We have, therefore, no wiring restriction, and we can use a tree structure for the or-gate, which has a logarithmic cost. Eq. 6.2 becomes:

$$E_c^b = K_b n + K_T(n) + K_o \log_2 m \tag{6.3}$$

For reasonable sizing of transistors, we will have $K_o \gg K_b$, since the transistors on the bus will be small, and the transistors on the or-gate will be large to minimize the delay of the or-gate. For large enough values of m, it is clear, however, that $E_c^a > E_c^b$.

To avoid having to pay the cost of adding one completion tree per target register, we can use a hybrid solution, in which each completion tree is shared by several of the target registers. If we use p completion trees, then the energy cost becomes:

$$E_c^h = K_b n \frac{m}{p} + K_T(n) + K_o \log_2 p \tag{6.4}$$

Differentiating Eq. 6.4 with respect to p, the optimum number of completion trees is given by:

$$p = m \min(1, n \frac{K_b}{K_o}) \tag{6.5}$$

A smaller number may be preferable to reduce the area overhead of computing completion.

6.2.6 Unsafe Completion

If the number of receivers is large, the time overhead of computing completion can be unacceptable. As in the case of a multiple sender channel, we can use the bus as a pipeline register to reduce the cycle time of the operation and decouple sender from receivers. The multiple sender circuit of Fig. 6.11 can be used to that effect.

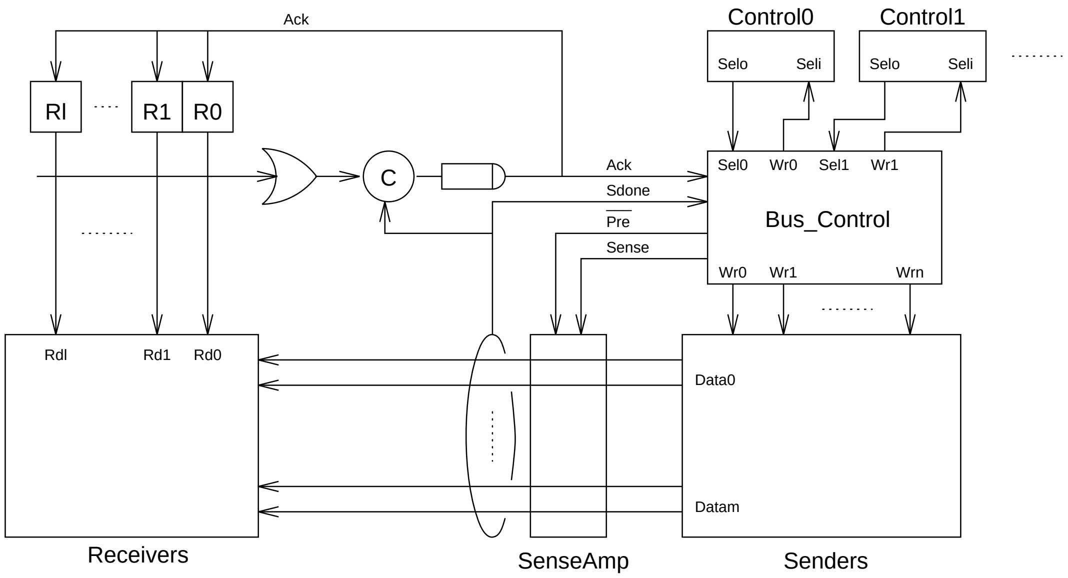


Figure 6.13: Unsafe completion scheme for multiple sender, multiple receiver channels. The bus transfer is pipelined to reduce the cycle time and decouple senders from receivers.

Given the amount of circuitry that has to be added to compute completion in a multiple receiver channel, there is a good deal to be gained in energy and delay by replacing those circuits with an appropriate timing assumption. We assume that the write operation is completed a fixed delay δ after the bus becomes valid and the target register is selected. The delay has to model the time it takes for a register to be written to; this delay is data-independent and comparable to two gate delays. Fig. 6.13 shows how signals are interconnected. The unsafe completion scheme results in significant savings in energy, area, and delay.

6.3 Summary & Conclusion

In this chapter we have compared several alternative ways of implementing register-to-register transfers. We looked at two types of transfers, point-to-point and buses.

We have shown how to make point to point transfers more efficient by using multi-valued registers. Fewer transitions are necessary to copy the same amount of information between multi-valued registers than between equivalent single valued registers, at the expense of more complex circuitry. There is a potential for some energy savings in reducing the number of transitions, but these savings are especially important if they are combined with a reduction in the complexity of function evaluation — some functions can be evaluated more efficiently if the input data is in one-of-n form.

We have shown in other chapters that pipelining is a bad strategy for low-energy design. The reason is that pipelining implies copying of information, and there is an alternate strategy — concurrency — that achieves the same effect without having to copy information. Sometimes, however, information is copied anyway because it results in a smaller and more efficient implementation. An example of this is datapath buses, where data is copied from one register to the bus and from the bus into another register. In this case we can consider the bus to be an intermediate register and use it to pipeline the register-to-register transfer. We have shown in this chapter how to use the sense-amplifier on the bus to increase the speed of the bus transfer and pipeline this transfer. The objective is to reduce or eliminate the overhead introduced by completion detection in such a transfer.

Chapter 7

Self-Limiting Circuits

One of the targets of low-energy design is to reduce the average power consumption of high performance circuits. The effect of this action is two-fold: it is easier and cheaper to cool the circuits, and we need to dedicate fewer pins to the power and ground connections. Present-day high-performance CMOS circuits are designed to the limit of available air-cooling technology; it is unlikely that in the near future we are going to overtake the capabilities of liquid-cooling technology, but liquid-cooling is completely inadequate for applications like desk-top workstations, that owe their success to their excellent price/performance ratio.

The cooling system guarantees that nowhere on the machine will the silicon temperature exceed the maximum allowable for that particular silicon technology. This is a worst-case design that takes into account the maximum power-output of all systems. As a result, the cooling system has to be over-designed, and chips will run, in average, at a lower temperature than they could, with the corresponding loss in circuit performance (the design could have been less conservative) and efficiency of the cooling process (the temperature differential available to extract heat from the chip is lower).

In this chapter we show how to use temperature feedback to slow down the hot parts of the circuit so that the maximum temperature specification, T_M, is never exceeded. This way, we eliminate the need for a worst-case cooling design; instead, the cooling power limits the maximum sustained performance of the system.

7.1 Heat Equation

Heat transfer occurs through three different mechanisms: radiation, conduction, and convection. Surface temperature and area are much too low for radiation to play a significant role in cooling; almost all of the heat is removed by conduction from the chip surface to the package and heat sinks and convection from the package and heat sink to the cooling fluid (air, water, freon, etc.).

The total effect of the cooling process can be represented by the thermal resistivity of the package, R_θ, and the thermal mass of the chip, M_θ. If T_s is the surface temperature on the chip, T_a is the ambient temperature, and P is the instantaneous electric power dissipated by the chip, then the heat transfer equation for the chip is:

$$M_\theta \frac{dT_s}{dt} = P - \frac{T_s - T_a}{R_\theta} \tag{7.1}$$

The real situation is a little more complicated than shown above because the thermal mass of the package, heat-sink, air, etc., has to be taken into consideration. We will model these effects by lumping them into the value of M_θ. Observe that the steady state solution of Eq. 7.1 does not depend on the value of M_θ.

Assuming constant power output, Eq. 7.1 can be solved; for initial conditions $T_s(0) = T_a$ we get:

$$T_s - T_a = PR_\theta \left(1 - e^{-\frac{t}{M_\theta R_\theta}}\right) \tag{7.2}$$

The steady state solution is independent of the initial conditions, $T_s = T_a + PR_\theta$. Under worst-case power assumptions, we have to guarantee that

$$P < \frac{T_M - T_a}{R_\theta} \tag{7.3}$$

Eq. 7.3 puts an upper bound to the peak performance of the system.

In synchronous systems, where power dissipation is a fairly constant function of time, we can use this equation to design the cooling system. It is, however, the wrong way to design an asynchronous system, where there may be a very large difference between the worst-case power dissipation and the average power dissipation. We will show in the next section how to replace the worst-case analysis with "self-limitation": chip temperature is used in a feedback loop to reduce circuit activity and ensure that the temperature specification is never exceeded.

7.2 Temperature Feed-Back

CMOS circuits slow down when the temperature increases. This occurs because at higher temperatures more electrons jump between the valence and conduction band, emitting a phonon. The extra phonons in the lattice reduce the mean-free-path of the electrons and, therefore, reduce electron mobility. This effect has to be considered in synchronous design when we do worst-case timing analysis. In an asynchronous system, the effect will be that the chip runs slower when it warms up, but can it still run a little faster if it is cold.

7.2.1 Linear Feedback

We will model the slow-down as a linear function of temperature. Instantaneous power is proportional to circuit activity and, therefore, a linear function of temperature as well. The heat equation becomes:

$$M_\theta \frac{dT_s}{dt} = P\frac{T_s - T_H}{T_a - T_H} - \frac{T_s - T_a}{R_\theta} \tag{7.4}$$

where $T_H \gg T_a$ and $T_H \gg T_M$. The constraint on power becomes

$$P < \frac{T_M - T_a}{R_\theta} \times \frac{T_H - T_a}{T_H - T_M} \tag{7.5}$$

This restriction is weaker than Eq. 7.3 by a factor $\frac{T_H - T_a}{T_H - T_M}$. The steady state performance is the same as before, since the limitation on constant power dissipation is the same. If power is not constant, however, the chip will run a little faster in average.

We can increase the strength of the temperature feedback by measuring the temperature of the chip and deliberately slowing down the speed of operation (for example by inserting a temperature controlled delay on a key handshake, as shown in Figure 7.1(b)). We can then choose the best value for the parameter T_H.

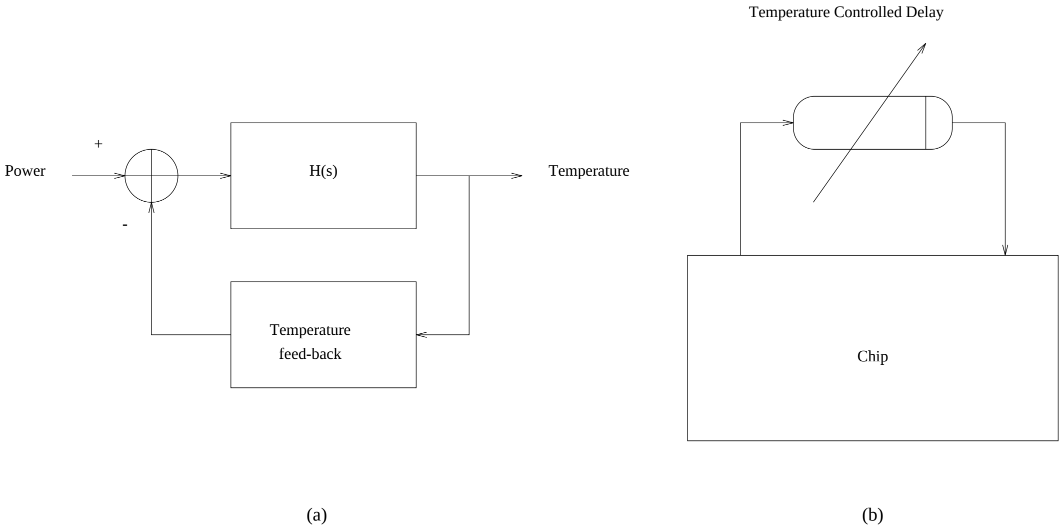


Figure 7.1: Negative temperature feedback. H(s) is the transfer from power to temperature for the system composed of the chip, package, and cooling fluid. Temperature feedback can be implemented as a temperature-controlled delay inserted in a critical loop.

7.2.2 Non-Linear Feedback

The main advantage of linear feedback is that the underlying mathematical theory is simple and well understood. We can use a more sophisticated linear model of the chip and packaging without significantly increasing the complexity of the design. In particular, it is easy to make sure that the feedback loop does not introduce instabilities that could cause large temperature oscillations.

We cannot use linear feedback, however, to guarantee that the chip will not overheat under any conditions, including variations on all parameters. Also, from the point of view of performance, linear feedback starts slowing down the chip well before getting to a dangerous temperature.

One of the remaining parameters that has to be designed for worst-case is the thermal resistance between chip surface and ambient. If we desire to use the cooling power to the maximum efficiency, the design has to make the minimum assumptions about this parameter. In the linear feedback section, we needed R_θ to calculate the maximum power that can be dissipated on chip. We can use the temperature measurement to guarantee that the surface temperature does not increase beyond a critical point, without having R_θ enter the equation. This way, we can increase the system performance by increasing the efficiency of the cooling system.

We will show in the following sections non-linear feedback mechanisms that achieve the following objectives: no slow-down unless the temperature becomes critical; under no circumstances is the temperature allowed to exceed the maximum specification — even under a failure of the cooling system; and the thermal system is stable.

7.2.3 Exponential Control

We can use a temperature-exponential delay to control the speed of the chip. The exponential law is interesting for two reasons: at low temperatures it is fairly constant, and the feedback control has very little effect; and there are several physical mechanisms that behave exponentially in temperature (for example, sub-threshold currents).

The heat equation becomes:

$$M_\theta \frac{dT_s}{dt} = Pe^{-\frac{T_s - T_a}{T_0}} - \frac{T_s - T_a}{R_\theta} \tag{7.6}$$

The steady state solution requires that we impose a restriction on P,

$$P < \frac{T_M - T_a}{R_\theta} \times e^{\frac{T_M - T_a}{T_0}} \tag{7.7}$$

Because the factor $e^{\frac{T_M - T_a}{T_0}}$ can be fairly large, Eq. 7.7 is a weak restriction on the worst-case power dissipation, and we can choose a very pessimistic R_θ without a great penalty in performance.

Stability of the solution can be proved by linearization of the heat equation around the steady-state solution. Performance is affected at low temperatures, but due to the slow start of the exponential function, this effect will be smaller than the equivalent linear system. Temperature may exceed specification if the cooling system were to be turned off, but performance will increase with increased cooling power.

7.2.4 On/Off Control

The simplest way of controlling temperature is the "thermostat" algorithm: the chip is completely stopped at temperature T_1 and re-started at temperature T_2, with $T_1 > T_2$. The hardware required to implement a thermostat is minimal: one temperature sensor/comparator with hysteresis, and a synchronizer for the delay line. The temperature will never go over T_1, irrespective of the value of P and R_θ. The condition $T_1 > T_2$ ensures the stability of the system.

The circuit will always work at maximum speed, at all temperatures, while the limiting mechanism is not working. Maximum performance is achieved by choosing T_2 as close to T_1 as possible; if $T_2 = T_1$ the functioning of the circuit will resemble the control described in the previous section.

If the thermal mass of the chip is high, once the chip is turned-off, it may take some time before the chip is cold enough to be turned on again. During that time, high-priority requests cannot be taken care of. If it is important that the system have a guaranteed maximum latency, this approach is not adequate.

We will show later how this method can be combined with global power-supply voltage control to avoid this problem and maximize energy performance.

7.3 Current Feedback

In the previous section we showed how to control the temperature of the hot spots in a system on a per-chip basis. Slowing down the hot chips may, as a consequence, slow down other parts of the system, because these parts spend more time waiting for data coming from the hot chips. This is inefficient in terms of energy: the system is working too fast and paying too much for operations that could be spread over time.

One way of globally controlling the speed of the system is by varying the power-supply voltage. Reducing the power-supply voltage reduces the speed of operation, but it also reduces the energy dissipation. Reducing the voltage reduces the speed of operation of all chips, but does not necessarily reduce the performance of the system. If some of the chips are being limited in speed because of temperature, they may run just as fast at a lower voltage (and, therefore, a lower temperature).

7.3.1 Sub-Optimal Voltage

We will use as a measure of performance time to completion for a specific task (running a program, routing a message, etc.). This task can be measured by the total capacitance that has to be switched to complete the task — which corresponds to the E/V_{DD}^2 index at a fixed power-supply voltage. Let V be the power-supply voltage, I be the power-supply current, C_c be the total capacitance to be switched, and T_c be the time to completion. During a period dt, the amount of charge supplied to the circuit is Idt; the total switched capacitance is dC, and the charge stored in that capacitance is, therefore, $VdC = Idt$. We can, therefore, write:

$$\int_0^{T_c} \frac{I}{V} dt = \int_0^{C_c} dC = C_c \qquad (7.8)$$

The optimization problem can be posed as a variational problem: minimize T_c under the constraint $\int_0^{T_c} \frac{I}{V} dt = C_c$. In general, it is very hard to give an optimal solution to this problem, it will depend strongly on what the I/V space looks like. There is, however, an interesting sub-optimal solution. I/V can be measured externally at any instant, and we adjust V so that I/V is maximized. At any moment, the system is running as fast as possible. We only need to make global measurements (power-supply voltage and power supply current), and we do not need to know what the actual task being performed by the processor is (neither C_c or T_c is evaluated or measured).

7.3.2 Current and Temperature Feedback

It is interesting to combine the on-off temperature control with the current feedback voltage control; this allows us to run all chips just hot enough so that the system is working at peak performance.

Consider a system consisting of a single chip. This chip is active a fraction δ of the time, $0 < \delta \leq 1$, and is blocked by the temperature control a fraction $1-\delta$. The average power dissipation of this chip can be computed as $P = kV^2\delta$. If $\delta < 1$, the power dissipation is constant and equal to the maximum power dissipation allowed by that package, $P_{\max}$. The average power supply current can be computed as $I = P/V = kV\delta$. Therefore, $I/V = k\delta$; the sub-optimal algorithm that maximizes I/V also maximizes the duty-cycle of the system.

The same argument holds when the circuit is idle not because of temperature control, but because the environment is not supplying data fast enough. In this case, too, the voltage control will slow the circuit down until it matches the environment.

7.4 Summary & Conclusion

In this section we have shown how to control the activity of an asynchronous circuit so that the circuit never overheats. Temperature feedback is used to control the throughput of individual circuits; this control removes from the design worst-case power constraints that would adversely affect average-case performance.

On/off control and exponential control were explained as ways of implementing temperature limitation. On/off control is simpler and prevents overheating under all circumstances, but it completely stops all circuit activity for long periods (thermal time constants are long). Exponential control is somewhat harder to implement and still requires some limits on the maximum allowable power dissipation for the circuit, but the system will always be active.

Self-limitation can be combined with power-supply voltage variation to slow down the system when it is working too hot (or when the environment is working slower), to stretch operations over the available time so that they can be performed more efficiently. The power-supply voltage is adjusted so that the "resistance" of the circuit V_{DD}/I_{DD} is minimized.

Chapter 8

Example: Processor Design

In this chapter we apply the techniques introduced previously in this thesis to the design of a low-energy-per-instruction microprocessor. The instruction set and architectural definition are based on low-energy requirements.

The specification is in terms of the energy required to execute a specific mix of instructions, plus a minimum requirement in performance (time to completion for those programs). To make the specification of the programs independent of the selection of the instruction set, those programs are given in a high level notation. A way of trading-off energy for delay has to be provided as well.

The type of programs to be run on this computer are reactive. When a program has terminated, the machine stops and waits for another program to become active. This mode of operation is important, since the processor operating in this mode will dissipate energy only when executing a program.

For the purpose of clarity, the example will be kept small. Care will be taken, though, that this design can be scaled up (wider datapath, more registers, more functional units).

8.1 Specification

The initial specification for the processor is given by the following program:

$\mu P \equiv *[\ P?Program;\ \ Execute(Program)\ ]$

The text of the program is read and executed, and when the program is finished, the process is re-started. This initial specification is in the form of a reactive loop, which ensures that all the energy that is spent by the circuit goes into execution of the program.

We reduce the generality of the specification by making some assumptions about the possible implementation. If the program is going to be stored in memory, as a sequence of instructions, then we do not need to read the whole program, just the address of the first instruction of the program. At the same time, the *Execute* function can be expressed as the execution of the instructions in the program. We now have some new variables, pc points to the next instruction to be executed; S represents the state of the processor (register contents, alu flags, etc.); Im is an array that contains the program instructions; and $Execute()$ is a function that executes an instruction and returns the address of the next instruction and the new state of the processor. There is a special instruction, *done*, that indicates that the program has finished.

$$\mu P \equiv *[\ P?pc;\ \ *[\ Im[pc] \neq done \longrightarrow pc, S := Execute(Im[pc], S)\ ]]$$

We have seen in previous chapters that there are many alternative ways of implementing indexing. In this case in particular, there many possible optimizations on the access to the Im array, due to the relative predictability of the string of instruction memory references: prefetch buffer, instruction memory cache, memory management unit, etc.. We remove the instruction memory form the processor process, so that we can treat the instruction memory separately. The processor and instruction memory processes become:

$$\begin{array}{l} \mu P \equiv *[\ P?pc; d\uparrow; *[\ d \longrightarrow A!pc \parallel I?i; \\ \qquad\qquad\qquad\qquad\qquad\qquad [\ i = done \longrightarrow d\downarrow \\ \qquad\qquad\qquad\qquad\qquad\qquad [\!]\ i \neq done \longrightarrow pc, S := Execute(i, S) \\ \qquad\quad] \qquad\qquad\qquad] \qquad\quad] \end{array}$$

$$IM \equiv *[A?a; I!Im[a]]$$

In the previous program we have made the assumption that there is one instruction per word in Im. This does not necessarily imply that all instructions have the same length; words in Im could have different lengths. For practical implementations, however, it is more convenient that all words in memory are equal, so we will deal with variable-length instructions in the processor specification. Since we have not yet defined what $Execute()$ is supposed to do, we can assume that all words in instruction memory have the same length, and $Execute()$ will handle multiple word instructions if there are any.

The previous program represents the basic specification of the circuit to be designed. The *Execute*() function is left unspecified. We deal with the instruction set next.

8.2 Instruction Set

In this section, we define the instruction set for the processor. This instruction set has to be complete, that is, we have to be able to implement all programs from the specification. While it is enough to make a machine equivalent to a Universal Turing Machine, we want an instruction set that executes efficiently the more frequent operations from the specification. We assume that these operations are:

Control-transfer: conditional branch, sub-routine call.

Arithmetic/logic operations: signed and unsigned integer arithmetic.

Memory operations: loading operands from memory, storing results back in memory, moving data.

At this level of the specification, the input/output operations of the processor are exclusively the sequences of instruction addresses and instructions. Since the entropy of these sequences is a lower bound to the energy complexity of the processor, we choose an instruction set that minimizes this entropy.

8.2.1 Control-Transfer

Control-transfer instructions alter the contents of the pc register. Technically, all instructions do this; each instruction specifies which is the next instruction to be executed. Nevertheless we differentiate between the instructions that only increment pc to the next consecutive memory location, and those that replace pc with a new value. The reason is based on the entropy of the instruction address sequence.

Let $A[1], \ldots, A[i], \ldots$ be this sequence. We assume that the probability distribution for $A[i]$ can be modeled by:

$$\Pr(A[i] = x) = \begin{cases} \lambda + \frac{1-\lambda}{n}, & \text{if } x = A[i-1]+1 \\ \frac{1-\lambda}{n}, & \text{otherwise} \end{cases} \tag{8.1}$$

where $0 < \lambda < 1$, and n is the number of instructions in the program. The parameter λ represents the probability that the next address in the sequence is the present address plus 1. For actual programs, $\lambda \approx \frac{5}{6}$, and $n \gg 1$. We can compute the entropy of $A[i]$, as:

$$\begin{aligned}
\mathcal{H}(A[i]) &= \left(\lambda + \frac{1-\lambda}{n}\right) \log \frac{1}{\lambda + \frac{1-\lambda}{n}} + (n-1)\frac{1-\lambda}{n} \log \frac{n}{1-\lambda} \\
&\approx \lambda \log \frac{1}{\lambda} + (1-\lambda) \log \frac{1}{1-\lambda} + (1-\lambda) \log n \\
&\approx (1-\lambda) \log n
\end{aligned} \tag{8.2}$$

To decrease the entropy of the address sequence, we have to increase λ or, similarly, increase the predictability of the instruction sequence. Delayed branching has this effect; the address of the following instruction is always known before decoding the current instruction. Delayed branching is inefficient, however, if we are not successful in filling the shadow with a useful instruction. In a clocked system, if we do not use the shadow we always lose at least one clock cycle, and delayed branching pays-off in performance. In an asynchronous system, the shadow may be less than one instruction cycle, and filling it with useless work (as a no-operation, for example) may waste, in average, more time than it saves. Also, delayed branching requires that more than one *pc* register be kept, since the instructions in the shadow are not necessarily sequential. It is estimated that delayed branching can be filled with a useful instruction about 80% of the time [13]. Assuming that one out of 6 instructions is a branch, we have that one out of 30 instructions is overhead in energy for a delayed branching strategy.

In this Chapter we will explore a different strategy for removing entropy from the instruction stream. We saw in Ch. 3 that if we can split an input channel into several sub-channels, then the average entropy per symbol of the multiple channels can be made lower than the average entropy per symbol of the single channel. In this case, the natural way of splitting the instruction channel I, is according to instruction type. The new program for the processor becomes:

$$
\begin{array}{l}
\mu P \equiv *[\ \ P?pc;\, d\uparrow;\, *[\ \ d \longrightarrow A!pc; \\
\qquad\qquad\qquad\qquad\qquad [\ \ \overline{I_d} \longrightarrow I_d?;\, d\downarrow \\
\qquad\qquad\qquad\qquad\qquad [\!]\ \ \overline{I_1} \longrightarrow I_1?i_1;\, pc, S := Execute_1(i_1, S) \\
\qquad\qquad\qquad\qquad\qquad [\!]\ \ \overline{I_2} \longrightarrow I_1?i_2;\, pc, S := Execute_2(i_2, S) \\
\qquad\qquad\qquad\qquad\qquad [\!]\ \ \overline{I_3} \longrightarrow I_1?i_3;\, pc, S := Execute_2(i_3, S) \\
\qquad\quad] \qquad\qquad\quad] \qquad\quad]
\end{array}
$$

where three different instruction types have been represented.

We show next an efficient implementation for the instruction channel split. The determination of the type of the instruction can be encoded on the previous instruction, so that before the processor fetches the next instruction, it already knows something about it.

In principle, the instruction code can be split over several instruction words. However, each instruction can fork the instruction stream, and the amount of information that has to be kept increases exponentially with the length of the split.

There are two ways of getting to an instruction; either by fall-through from the previous sequential instruction, or by a jump to that address. In the first case, the extra information on the current instruction can be encoded in the previous instruction; in the second case, the extra information can be encoded in the jump address.

We introduce this split in the μP program:

$$
\begin{array}{l}
\mu P \equiv *[\ \ P?(di, pc);\, d\uparrow; \\
\qquad\quad *[\ \ d \longrightarrow A!pc; \\
\qquad\qquad\qquad [\ \ di = done \longrightarrow d\downarrow \\
\qquad\qquad\qquad [\!]\ \ di = 1 \longrightarrow I?(i_1, ni);\, pc, S, di := Execute_1(i_1, di, ni, S) \\
\qquad\qquad\qquad [\!]\ \ di = 2 \longrightarrow I?(i_2, ni);\, pc, S, di := Execute_2(i_2, di, ni, S) \\
\qquad\qquad\qquad [\!]\ \ di = 3 \longrightarrow I?(i_3, ni);\, pc, S, di := Execute_3(i_3, di, ni, S) \\
\qquad] \quad] \qquad]
\end{array}
$$

The split between instruction type and opcode can be used to gain an extra stage of pipelining as well, without having to spend energy copying information.

For the moment we will postpone the decision on how to split the instruction channel, until we have all of the instruction types of the processor. We keep around the variable di, and rewrite the μP process as:

$$
\begin{array}{l}
\mu P \equiv *[\ P?(di, pc);\ d\uparrow; \\
\qquad *[\ d \longrightarrow A!pc \,\|\, I?(i, ni) \\
\qquad\qquad [\ di = done \longrightarrow d\downarrow \\
\qquad\qquad [\!]\ di \neq done \longrightarrow pc, S, di := Execute(i, di, ni, S) \\
\quad]\quad]\qquad]
\end{array}
$$

Next we discuss the addressing modes for control-transfer instructions. A few of the choices are:

Immediate: The destination address is encoded in the instruction. Immediate addressing is easy to implement, and does not use the register bank. It does, however, require either longer instructions to store the address, and this address has to be pre-computed at compile-time or load-time. To keep the processor simple, we use this type of addressing for conditional branch instructions.

Register indirect: The target address is the contents of register. This type of addressing is necessary to use jump tables.

Stack indirect: The target address is taken from the top of the hardware stack. The stack is used to implement sub-routine calls and returns (the alternative option is to use a register and save or restore the register into a software stack). Return from sub-routine uses this addressing mode.

We have tried to minimize register access for address computations. Registers will be highly overloaded by the arithmetic and memory instructions; extra buses to the registers cost extra time and energy for each register access. An alternative is to provide a register bank exclusively for addresses. We do not choose this option in order to keep the processor simple.

The control-transfer instructions are:

branch if cc: Branch to the immediate address if condition cc is true. The conditions are the usual arithmetic and logic conditions.

pshpc: Used to implement sub-routine calls in combination with the branch instruction. Save the address of the return instruction in the hardware stack.

return: Return from sub-routine. Pull the address of the next instruction from the hardware stack.

The following program specifies these instructions. The stack process has been left out.

$$
\begin{array}{l}
\mu P \equiv *[\;\; P?(di, pc);\, d\uparrow; \\
\qquad\quad *[\;\; d \longrightarrow A!pc \,\|\, I?(i, ni); \\
\qquad\qquad\qquad [\;\; i : di = done \longrightarrow d\downarrow \\
\qquad\qquad\qquad [\!]\;\; i : di = pshpc\;\; ri \longrightarrow Push!(ri, pc + 3); \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad di, pc := ni, pc + 1 \\
\qquad\qquad\qquad [\!]\;\; i : di = return \longrightarrow Pull?(di, pc) \\
\qquad\qquad\qquad [\!]\;\; (i : di = br\;\; \mathrm{cc}) \wedge \mathrm{cc}(flags) \longrightarrow A!(pc + 1) \,\|\, I?(di, pc) \\
\qquad\qquad\qquad [\!]\;\; (i : di = br\;\; \mathrm{cc}) \wedge \neg\mathrm{cc}(flags) \longrightarrow di, pc := ni, pc + 2 \\
\qquad\qquad\qquad [\!]\;\; else \;\; \longrightarrow pc, S, di := Execute(i, di, ni, S) \\
\qquad\quad] \quad] \quad]
\end{array}
$$

8.2.2 Arithmetic/Logic Operations

The operands for alu operations can be either registers in the processor or data memory. There are good reasons to use memory operands, especially for vector operations, where memory access can be done very efficiently. Vector operations are, however, more appropriate for co-processor hardware, and we will restrict the specification to register-to-register operations (two source registers, one destination register).

As a generic specification for alu operations, we use:

$$
\begin{array}{l}
[\!]\;\; i : di = aluop\;\; \mathrm{r0}\;\; \mathrm{r1}\;\; \mathrm{r2} \longrightarrow \\
\qquad\qquad\qquad\qquad R[\mathrm{r2}] := alu(aluop, flags, R[\mathrm{r0}], R[\mathrm{r1}]); \\
\qquad\qquad\qquad\qquad di, pc := ni, pc + 1
\end{array}
$$

The *alu* function is left unspecified at this moment. The result flags are assigned as a side-effect of the *alu* function call.

8.2.3 Memory Addressing

Most of the operations executed by a processor are data-movement operations. Data movement requires address computation and the actual loading and storing of the data.

We first decide on the addressing modes for data operations:

Immediate: The data to be loaded is part of the instruction. This addressing is convenient for loading constants.

Register: The data to be loaded/stored is the contents of a register. All data movements have as source or destination a register.

Register indirect: The address of the data to be loaded/stored is the contents of a register.

Many more addressing modes are possible. Addressing modes are justified in terms of the reduction of instructions required to compute the source/target address. For the moment, we restrict the addressing modes to those presented above. The data movement instructions are:

Load immediate: Load a constant into a register.

Load indirect into a register: Use the value of a register as an address, and load a second register with the corresponding memory location.

Store indirect from a register: Use the value of a register as an address, and store the contents of a third register in the corresponding memory location.

We introduce at this point the data memory array, $Dm[]$. The CSP specification for the memory instructions is:

$$
\begin{array}{l}
[\!]\ \ i : di = ldi\ \ \mathrm{r2} \longrightarrow A!(pc+1) \,\|\, I?R[\mathrm{r2}];\, di, pc := ni; pc+2 \\
[\!]\ \ i : di = load\ \ \mathrm{r0}\ \ \mathrm{r2} \longrightarrow DA!R[\mathrm{r0}];\, DR?R[\mathrm{r2}]; \\
\qquad\qquad\qquad\qquad\qquad\qquad di, pc := ni, pc+1 \\
[\!]\ \ i : di = store\ \ \mathrm{r0}\ \ \mathrm{r1} \longrightarrow DA!R[\mathrm{r0}] \,\|\, DW!R[\mathrm{r1}]; \\
\qquad\qquad\qquad\qquad\qquad\qquad di, pc := ni, pc+1
\end{array}
$$

$$
\begin{array}{l}
DM \equiv *[\,[\overline{DR} \longrightarrow DA?da;\, DR!Dm[da] \\
\qquad\quad [\!]\overline{DW} \longrightarrow DA?da;\, DW?Dm[da] \\
\qquad\ ]]
\end{array}
$$

8.3 Process Refinement

In this section we refine the specification of the processor given previously into a CSP program suitable for implementation.

The transformations to the processor program have the goal of improving the energy complexity without increasing the time complexity, and improving the time complexity without increasing the energy complexity.

One refinement has already been introduced, instruction splitting. In the present specification instruction, splitting has not been used either for energy or for performance. The objective of instruction splitting is to do predecoding of the next instruction to be executed — improving performance — and increase the hit ratio of the prefetch mechanism — improving both performance and energy.

We start from the following program that we have derived from the specification of the instruction set:

$$
\begin{array}{l}
\mu P \equiv *[\;\; P?(di,pc);\, d\uparrow; \\
\qquad\qquad *[\;\; d \longrightarrow A!pc \,\|\, I?(i,ni); \\
\qquad\qquad\qquad [\;\; i : di = done \longrightarrow d\downarrow \\
\qquad\qquad\qquad [\!]\;\; i : di = pshpc\;\; ri \longrightarrow Push!(ri, pc+3);\, di, pc := ni, pc+1 \\
\qquad\qquad\qquad [\!]\;\; i : di = return \longrightarrow Pull?(di,pc) \\
\qquad\qquad\qquad [\!]\;\; (i : di = br\;\; \mathrm{cc}) \wedge \mathrm{cc}(flags) \longrightarrow A!(pc+1) \,\|\, I?(di,pc) \\
\qquad\qquad\qquad [\!]\;\; (i : di = br\;\; \mathrm{cc}) \wedge \neg\mathrm{cc}(flags) \longrightarrow di, pc := ni, pc+2 \\
\qquad\qquad\qquad [\!]\;\; i : di = aluop\;\; \mathrm{r0}\;\; \mathrm{r1}\;\; \mathrm{r2} \longrightarrow \\
\qquad\qquad\qquad\qquad\qquad R[\mathrm{r2}] := alu(aluop, flags, R[\mathrm{r0}], R[\mathrm{r1}]); \\
\qquad\qquad\qquad\qquad\qquad\quad di, pc := ni, pc+1 \\
\qquad\qquad\qquad [\!]\;\; i : di = ldi\;\; \mathrm{r2} \longrightarrow A!(pc+1);\, I?R[\mathrm{r2}]; \\
\qquad\qquad\qquad\qquad\qquad\quad di, pc := ni; pc+2 \\
\qquad\qquad\qquad [\!]\;\; i : di = load\;\; \mathrm{r0}\;\; \mathrm{r2} \longrightarrow DA!R[\mathrm{r0}];\, DR?R[\mathrm{r2}]; \\
\qquad\qquad\qquad\qquad\qquad\quad di, pc := ni, pc+1 \\
\qquad\qquad\qquad [\!]\;\; i : di = store\;\; \mathrm{r0}\;\; \mathrm{r1} \longrightarrow DA!R[\mathrm{r0}] \,\|\, DW!R[\mathrm{r1}]; \\
\qquad\qquad\qquad\qquad\qquad\quad di, pc := ni, pc+1 \\
\qquad\qquad\qquad [\!]\;\; else \longrightarrow pc, S, di := Execute(i, di, ni, S) \\
\qquad]\quad]\quad]
\end{array}
$$

where the *else* statement has been included for possible extensions to the instruction set.

8.3.1 Instruction Fetch

The pc register has to be communicated to the instruction memory for every instruction fetch. We have, as a trade-off, the cost of communicating the value of the pc register, which scales with the size of pc, against the cost of re-computing the pc in two places. Most of the time, computing the next pc is a very simple operation, with constant cost independent of the size of pc. In this case, it is better to keep two program counters and update one from the other when necessary.

To this effect, we change the specification of the instruction memory:

$$
\begin{array}{l}
IM \equiv *[[\ \overline{A} \longrightarrow A?a \\
\qquad\quad [\!]\ \overline{I} \longrightarrow I!Im[a]\,;\, a := a + 1 \\
\qquad]]
\end{array}
$$

The pc needs to be sent on A only when a new value is loaded into it. We add this modification to the processor program:

$$
\begin{array}{l}
\mu P \equiv *[\ \ P?(di, pc);\, d\uparrow;\, A!pc; \\
\qquad\quad *[\ \ d \longrightarrow I?(i, ni); \\
\qquad\qquad\quad [\ \ i : di = done \longrightarrow d\downarrow \\
\qquad\qquad\quad [\!]\ \ i : di = pshpc\ \ ri \longrightarrow Push!(ri, pc + 3);\, di, pc := ni, pc + 1 \\
\qquad\qquad\quad [\!]\ \ i : di = return \longrightarrow Pull?(di, pc);\, A!pc \\
\qquad\qquad\quad [\!]\ \ (i : di = br\ \ \mathrm{cc}) \wedge \mathrm{cc}(flags) \longrightarrow I?(di, pc);\, A!pc \\
\qquad\qquad\quad [\!]\ \ (i : di = br\ \ \mathrm{cc}) \wedge \neg\mathrm{cc}(flags) \longrightarrow di, pc := ni, pc + 2 \\
\qquad\qquad\quad [\!]\ \ i : di = aluop\ \ \mathrm{r0}\ \ \mathrm{r1}\ \ \mathrm{r2} \longrightarrow \\
\qquad\qquad\qquad\qquad\qquad R[\mathrm{r2}] := alu(aluop, flags, R[\mathrm{r0}], R[\mathrm{r1}]); \\
\qquad\qquad\qquad\qquad\qquad\quad di, pc := ni, pc + 1 \\
\qquad\qquad\quad [\!]\ \ i : di = ldi\ \ \mathrm{r2} \longrightarrow I?R[\mathrm{r2}];\, di, pc := ni;\, pc + 2 \\
\qquad\qquad\quad [\!]\ \ i : di = load\ \ \mathrm{r0}\ \ \mathrm{r2} \longrightarrow DA!R[\mathrm{r0}];\, DR?R[\mathrm{r2}]; \\
\qquad\qquad\qquad\qquad\qquad\quad di, pc := ni, pc + 1 \\
\qquad\qquad\quad [\!]\ \ i : di = store\ \ \mathrm{r0}\ \ \mathrm{r1} \longrightarrow DA!R[\mathrm{r0}] \parallel DW!R[\mathrm{r1}]; \\
\qquad\qquad\qquad\qquad\qquad\quad di, pc := ni, pc + 1 \\
\qquad\qquad\quad [\!]\ \ else \longrightarrow pc, S, di := Execute(i, di, ni, S) \\
\qquad]\quad]\quad]
\end{array}
$$

Instruction	offset	branch	group
done	**false**	**true**	1
pshpc	**false**	**false**	0
return	**false**	**true**	1
branch	**true**	**true**	3
alu	**false**	**false**	0
ldi	**true**	**false**	2
load	**false**	**false**	0
store	**false**	**false**	0

Table 8.1: Instruction types according to their effect on the *pc* register.

8.3.2 Instruction Decoding

The next step in the transformation is to decide what information will be contained in *ni*. The idea is to be able to do some predecoding so that the decoding step is simpler.

From the energy point of view, the variables *i* and *di* are shared in too many places; with predecoding, we can copy these variables to the proper place so that they don't have to be shared.

From the performance point of view, the predecoding can be done concurrently with the decoding of the previous instruction so that the predecoding does not introduce a delay penalty. This makes decoding simpler, potentially reducing the cycle time. The type of information in *di* may be useful not only for predecoding, but to determine data dependencies between consecutive operations.

We will consider two choices for *di*. First, we minimize the size of *di* to contain information exclusively about updating the *pc* register. Second, we maximize the size of *di* to contain information about the exact type of instruction to be executed next.

We assign two bits to *di*: *di.offset* (the instruction is an offset instruction) and *di.branch* (the instruction is a branch instruction). Table 8.1 shows the classification of the processor instructions according to this criteria.

We can now re-write the μP program with predecoding, and we introduce

4 separate instruction registers, $i0$, $i1$, $i2$, and $i3$.

$$
\begin{array}{l}
\mu P1 \equiv \\
\quad *[\ P?(di, pc);\ d\uparrow;\ A!pc; \\
\quad\quad *[\ d \longrightarrow \\
\quad\quad\quad [\neg di.\mathit{offset} \wedge \neg di.\mathit{branch} \longrightarrow I?(i0, ni);\ di, pc := ni, pc + 1; \\
\quad\quad\quad\quad [\ i0 = \mathit{aluop}\ \mathrm{r0}\ \mathrm{r1}\ \mathrm{r2} \longrightarrow \\
\quad\quad\quad\quad\quad\quad\quad R[\mathrm{r2}] := \mathit{alu}(\mathit{aluop}, \mathit{flags}, R[\mathrm{r0}], R[\mathrm{r1}]) \\
\quad\quad\quad\quad [\!]\ i0 = \mathit{load}\ \mathrm{r0}\ \mathrm{r2} \longrightarrow DA!R[\mathrm{r0}];\ DR?R[\mathrm{r2}] \\
\quad\quad\quad\quad [\!]\ i0 = \mathit{store}\ \mathrm{r0}\ \mathrm{r1} \longrightarrow DA!R[\mathrm{r0}] \parallel DW!R[\mathrm{r1}] \\
\quad\quad\quad\quad [\!]\ i0 = \mathit{pshpc}\ \ ri \longrightarrow \mathit{Push}!(ri, pc\uparrow 2) \\
\quad\quad\quad\quad] \\
\quad\quad\quad [\!]\neg di.\mathit{offset} \wedge \ \ di.\mathit{branch} \longrightarrow I?(i1, ni); \\
\quad\quad\quad\quad [\ i1 = \mathit{done} \longrightarrow d\downarrow \\
\quad\quad\quad\quad [\!]\ i1 = \mathit{return} \longrightarrow \mathit{Pull}?(di, pc);\ A!pc \\
\quad\quad\quad\quad] \\
\quad\quad\quad [\!]\ di.\mathit{offset} \wedge \neg di.\mathit{branch} \longrightarrow I?(i2, ni);\ di, pc := ni, pc + 2; \\
\quad\quad\quad\quad [\ i2 = \mathit{ldi}\ \mathrm{r2} \longrightarrow I?R[\mathrm{r2}]\ ] \\
\quad\quad\quad [\!]\ di.\mathit{offset} \wedge \ \ di.\mathit{branch} \longrightarrow I?(i3, ni);\ di, pc := ni, pc + 2 \\
\quad\quad\quad\quad [\ (i3 = \mathit{br}\ \ \mathrm{cc}) \wedge \mathrm{cc}(\mathit{flags}) \longrightarrow I?(di, pc);\ A!pc \\
\quad\quad\quad\quad [\!]\ (i3 = \mathit{br}\ \ \mathrm{cc}) \wedge \neg\mathrm{cc}(\mathit{flags}) \longrightarrow A!pc \\
\quad]\quad]\quad]\quad]
\end{array}
$$

In this program, the instruction registers have become simpler, at the expense of the I channel.

The second approach is to encode in di the format of the instruction. In this instruction set, we have the following formats:

Alu operations: The instruction contains an alu opcode and three register names, r0, r1, and r2.

Load/store: The instruction contains three register names, r0, r1, and r2.

Done/return: No operands.

Load immediate: The instruction has one register name, r2, and an offset with the data.

Pshpc: The instruction contains the predecode information for the return instruction.

Conditional branch: The instruction contains an opcode with the condition to branch on, and an offset with the target destination.

With the previous assignment of values to di, the μP program becomes:

$$
\begin{array}{l}
\mu P2 \equiv *[\ P?(di,pc);\ d\uparrow;\ A!pc;\\
\quad *[\ d \longrightarrow\\
\quad\quad [\ di = alu \longrightarrow I?(aluop,\mathrm{r0},\mathrm{r1},\mathrm{r2},ni);\ di,pc := ni,pc+1;\\
\quad\quad\quad\quad R[\mathrm{r2}] := alu(aluop,flags,R[\mathrm{r0}],R[\mathrm{r1}])\\
\quad\quad [\!]\ di = mem \longrightarrow I?(ldst,\mathrm{r0},\mathrm{r1},\mathrm{r2},ni);\ di,pc := ni,pc+1;\\
\quad\quad\quad [\ ldst = load \longrightarrow DA!R[\mathrm{r0}];\ DR?R[\mathrm{r2}]\\
\quad\quad\quad [\!]\ ldst = store \longrightarrow DA!R[\mathrm{r0}] \parallel DW!R[\mathrm{r1}]\\
\quad\quad\quad]\\
\quad\quad [\!]\ di = dn/ret \longrightarrow I?(dnret);\\
\quad\quad\quad [\ dnret = done \longrightarrow d\uparrow\\
\quad\quad\quad [\!]\ dnret = return \longrightarrow Pull?(di,pc);\ A!pc\\
\quad\quad\quad]\\
\quad\quad [\!]\ di = ldi \longrightarrow I?(\mathrm{r2},ni);\ di,pc := ni,pc+2;\ I?R[\mathrm{r2}]\\
\quad\quad [\!]\ di = pshpc \longrightarrow I?(ri,ni);\ Push!(ri,pc+3);\\
\quad\quad\quad\quad\quad di,pc := ni,pc+1\\
\quad\quad [\!]\ di = branch \longrightarrow I?(\mathrm{cc},ni);\ di,pc := ni,pc+2;\\
\quad\quad\quad [\ \mathrm{cc}(flags) \longrightarrow I?(di,pc);\ A!pc\\
\quad\quad\quad [\!]\neg\mathrm{cc}(flags) \longrightarrow A!pc\\
]\ \ ]\ \ ]\ \ ]
\end{array}
$$

The di variable is shared in more places, but it only needs to be three bits long (instead of a full instruction word). The I channel is more complex now, but the registers it writes to are very simple.

We choose between $\mu P1$ and $\mu P2$ based on energy considerations and potential for pipelining. We assume that only the cost of decoding is different; the cost of executing the actual instruction is the same.

In the program $\mu P1$, the channel I has one source and six destinations. In the worst-case, destination $i0$ is shared by all registers (unless the register indexing operation is specified otherwise).

In the program $\mu P2$, we break up the channel I in several sub-fields. Effectively, the number of destinations is very small (between two and three for each bit). Again, the worst-case sharing is for the r0, r1, and r2 fields, which are shared by all the registers.

According to this analysis, $\mu P2$ looks more energy efficient, though the differences do not appear critical at this level of the design. More importantly, $\mu P2$ can be better pipelined. The reason is that there is less difference in complexity between predecoding and decoding; therefore, predecoding the next instruction overlaps better with decoding the present instruction.

In this example we choose $\mu P2$, and keep refining it for energy complexity before optimizing time complexity.

8.3.3 Offset Register

We can further reduce the number of destinations of the I channel by introducing an offset register. The reason is that it is better in energy complexity to do two simple assignments than to do one complex assignment. We use the offset register when the data in the instruction memory does not correspond to an instruction code.

For the branch instruction, we have removed the $A!pc$ communication on a branch not taken; in exchange, we read in the offset both for branch taken and not taken. The instruction following a *return* or a *done* is never executed; therefore, we do not have to read these instructions if *di* is enough to identify them.

Register *ni* can be removed and replaced with *di* since *ni* and *di*a are never used at the same time. The purpose of having two registers is to allow the pipelining of predecoding and decoding; however, pipelining can be achieved without *ni*, as we show later.

$$
\begin{array}{l}
\mu P2 \equiv *[\ P?(di, pc);\ d\uparrow;\ A!pc; \\
\quad *[\ d \longrightarrow \\
\qquad [\ di = alu \longrightarrow I?(aluop, \mathrm{r0}, \mathrm{r1}, \mathrm{r2}, di);\ pc := pc + 1; \\
\qquad\qquad R[\mathrm{r2}] := alu(aluop, flags, R[\mathrm{r0}], R[\mathrm{r1}]) \\
\qquad [\!]\ di = mem \longrightarrow I?(ldst, \mathrm{r0}, \mathrm{r1}, \mathrm{r2}, di);\ pc := pc + 1; \\
\qquad\quad [\ ldst = load \longrightarrow DA!R[\mathrm{r0}];\ DR?R[\mathrm{r2}] \\
\qquad\quad [\!]\ ldst = store \longrightarrow DA!R[\mathrm{r0}] \parallel DW!R[\mathrm{r1}] \\
\qquad\quad] \\
\qquad [\!]\ di = done \longrightarrow d\uparrow \\
\qquad [\!]\ di = return \longrightarrow Pull?(di, pc);\ A!pc \\
\qquad [\!]\ di = ldi \longrightarrow I?(\mathrm{r2}, di);\ I?R[\mathrm{r2}];\ pc := pc + 2 \\
\qquad [\!]\ di = pshpc \longrightarrow I?(ri, di);\ Push!(ri, pc + 3);\ pc := pc + 1 \\
\qquad [\!]\ di = branch \longrightarrow I?(\mathrm{cc}, di);\ I?or;\ pc := pc + 2; \\
\qquad\quad [\ \mathrm{cc}(flags) \longrightarrow di, pc := or.di, or.pc;\ A!pc \\
\qquad\quad [\!]\neg\mathrm{cc}(flags) \longrightarrow \mathbf{skip} \\
]\ \ ]\ \ \ ]\ \ \ \ ]
\end{array}
$$

8.3.4 Register-File Extraction

We remove the explicit indexing into the register-file by treating the register-file as memory. To do this, we have to decide on the number of read and write ports to this register-file. The more ports to the register file, the more units can operate in parallel. Having a large number of ports does, however, have some disadvantages; assignments take a longer time and/or more energy (there is more capacitance in each node of the register).

It is critical to tie the circuit level design of the register file to the high level specification since a good deal of low level optimization is necessary to obtain a fast and efficient register file. We have looked at several alternative implementations for low-energy buses in Ch. 6, and we will steer the derivation towards one of those designs.

With pipelining in mind, we consolidate all the read actions and all the write actions. In this case we need two read-ports and one write-port. Given that the register file is a relatively small array, we will use the variables r0, r1, and r2 as shared variables between the array and the processor. We define 3 channels, X and Y for read, Z for write.

$$
\begin{array}{l}
\mu P2 \equiv *[\ P?(di, pc);\ d\uparrow;\ A!pc;\\
\quad *[\ d \longrightarrow\\
\qquad [\ di = alu \longrightarrow I?(aluop, \mathrm{r0}, \mathrm{r1}, \mathrm{r2}, di);\ pc := pc + 1;\\
\qquad\qquad X?xa \parallel Y?ya;\ Z!alu(aluop, flags, xa, ya)\\
\qquad [\!]\ di = mem \longrightarrow I?(ldst, \mathrm{r0}, \mathrm{r1}, \mathrm{r2}, di);\ pc := pc + 1;\\
\qquad\quad [\ ldst = load \longrightarrow DA!(X?);\ Z!(DR?)\\
\qquad\quad [\!]\ ldst = store \longrightarrow DA!(X?) \parallel DW!(Y?)\\
\qquad\quad]\\
\qquad [\!]\ di = done \longrightarrow d\uparrow\\
\qquad [\!]\ di = return \longrightarrow Pull?(di, pc);\ A!pc\\
\qquad [\!]\ di = ldi \longrightarrow I?(\mathrm{r2}, di);\ Z!(I?);\ pc := pc + 2\\
\qquad [\!]\ di = pshpc \longrightarrow I?(ri, di);\ Push!(ri, pc + 3);\ pc := pc + 1\\
\qquad [\!]\ di = branch \longrightarrow I?(\mathrm{cc}, di);\ I?or;\ pc := pc + 2;\\
\qquad\quad [\ \mathrm{cc}(flags) \longrightarrow di, pc := or.di, or.pc;\ A!pc\\
\qquad\quad [\!]\neg\mathrm{cc}(flags) \longrightarrow \mathbf{skip}\\
]\ \ ]\ \ ]\ \ ]
\end{array}
$$

$$
\begin{array}{l}
Regfile \equiv \langle\parallel\ i:\ \ *[[\ \overline{X} \wedge (\mathrm{r0} = i) \longrightarrow X!R_i\ ]]\\
\qquad\qquad\quad \parallel\ *[[\ \overline{Y} \wedge (\mathrm{r1} = i) \longrightarrow Y!R_i\ ]]\\
\qquad\qquad\quad \parallel\ *[[\ \overline{Z} \wedge (\mathrm{r2} = i) \longrightarrow Z?R_i\ ]]\\
\qquad \rangle
\end{array}
$$

8.3.5 Pipelining and Concurrency

A common technique to reduce the cycle time of a processor is to introduce pipelining between successive functional units. Pipelining has a disadvantage: it requires copying of information and, therefore, increases the energy consumption per stage.

Some amount of information copying can be unavoidable; to preserve the low-energy characteristic of the design, and to improve the delay performance, we can introduce pipelining along the boundaries set by that information copying. In an alu operation, for example, it is just as natural natural and as efficient in energy to latch the input data while computing the output function. In general, we design programs so that semicolons separate data-dependencies; at each semicolon we can introduce a pipeline stage.

The lack of data dependencies also introduces some concurrency. For example, *pc* computations are independent of the rest of the computations of the processor, except when the *pc* register has to be updated in *branch* and *return* instructions. These computations can proceed in parallel and synchronized only in those cases.

Another source of concurrency is the split of the instruction decoding into two parts. We have on the processor at the same time part of two instructions, and the execution of both instructions overlaps. The concurrency we obtain from this overlap is derived from the data parallelism obtained from reading more than one bit at a time, and from the fact that not all of the instruction is needed to make decisions about decoding.

We use all these sources of concurrency to decrease the cycle time without increasing the energy required per instruction. To this effect, we group the actions in the program into several stages. All the actions in the same stage can proceed concurrently (no data dependencies between actions in the same stage). Actions in different stages can correspond to different instructions (pipelining).

$$\mu P \equiv *[[\ P?(di, pc);\ A!pc;\ G!\ ]]$$

$$\begin{array}{lll}
Predecode \equiv *[\ \overline{G} \longrightarrow & [& di = alu \longrightarrow Alu! \parallel Ipc! \\
 & [] & di = mem \longrightarrow Mem! \parallel Ipc! \\
 & [] & di = done \longrightarrow G? \\
 & [] & di = return \longrightarrow Pullpc! \\
 & [] & di = ldi \longrightarrow Ldi! \parallel I2pc! \\
 & [] & di = pshpc \longrightarrow Pshpc!; Ipc! \\
 & [] & di = branch \longrightarrow Br! \\
\quad] &] &
\end{array}$$

$$
\begin{array}{l}
\mathit{Decode} \equiv *[[\ \overline{Alu} \longrightarrow I?(aluop, \mathrm{r0}, \mathrm{r1}, \mathrm{r2}, di);\\
\qquad\qquad\qquad\qquad (Xb! \bullet Yb! \bullet Zb! \bullet A!aluop) \parallel Alu?\\
\qquad\qquad\quad [\!]\ \overline{Mem} \longrightarrow I?(ldst, \mathrm{r0}, \mathrm{r1}, \mathrm{r2}, di);\\
\qquad\qquad\qquad\qquad [\ \ ldst = load \longrightarrow (Xb! \bullet Zb! \bullet M!) \parallel Mem?\\
\qquad\qquad\qquad\qquad [\!]\ ldst = store \longrightarrow (Xb! \bullet Yb! \bullet M!) \parallel Mem?\\
\qquad\qquad\qquad\qquad]\\
\qquad\qquad\quad [\!]\ \overline{Ldi} \longrightarrow I?(\mathrm{r2}, di); Zb!; Z!(I?) \parallel Ldi?\\
\qquad\qquad\quad [\!]\ \overline{Br} \longrightarrow I?(cc, di); I?or \parallel F?f;\\
\qquad\qquad\qquad\qquad [\ \ cc(f) \longrightarrow Orpc!; Apc! \parallel Br?\\
\qquad\qquad\qquad\qquad [\!]\neg cc(f) \longrightarrow I2pc! \parallel Br?\\
\qquad\qquad\qquad\qquad]\\
\qquad\qquad\quad]]
\end{array}
$$

$$
\begin{array}{l}
\mathit{ALUunit} \equiv *[[\ \overline{A} \longrightarrow X?xa \parallel Y?ya; Z!alu(A?, flags, xa, ya)\\
\qquad\qquad\quad [\!]\ \overline{F} \longrightarrow F!flags\\
\qquad\qquad\quad]]
\end{array}
$$

$$
\begin{array}{l}
\mathit{MEMunit} \equiv *[[\ \overline{M} \wedge ldst = load \longrightarrow DA!(X?) \parallel Z!(DR?) \parallel M?\\
\qquad\qquad\quad [\!]\ \overline{M} \wedge ldst = store \longrightarrow DA!(X?) \parallel DW!(Y?) \parallel M?\\
\qquad\qquad\quad]]
\end{array}
$$

$$
\begin{array}{l}
\mathit{PCunit} \equiv *[[\ \overline{Ipc} \longrightarrow\ Ipc? \parallel pc := pc + 1\\
\qquad\qquad [\!]\ \overline{I2pc} \longrightarrow\ I2pc? \parallel pc := pc + 2\\
\qquad\qquad [\!]\ \overline{Pullpc} \longrightarrow\ Pullpc? \parallel Pull?(di, pc)\\
\qquad\qquad [\!]\ \overline{Pushpc} \longrightarrow\ Pushpc? \parallel Push!(ri, pc)\\
\qquad\qquad [\!]\ \overline{Apc} \longrightarrow\ Apc? \bullet A!pc\\
\qquad\qquad [\!]\ \overline{Orpc} \longrightarrow\ Orpc? \parallel di, pc := or.di, or, pc\\
\qquad\qquad [\!]\ \overline{Ppc} \longrightarrow\ Ppc? \parallel P?(di, pc)\\
\qquad\qquad]]
\end{array}
$$

$$
\begin{array}{l}
\mathit{Regfile} \equiv \langle \parallel\ i:\ *[[\ \neg b_i \wedge \overline{Xb} \wedge (\mathrm{r0} = i) \longrightarrow Xb! \bullet X!R_i\ ]]\\
\qquad\qquad\quad \parallel\ *[[\ \neg b_i \wedge \overline{Yb} \wedge (\mathrm{r1} = i) \longrightarrow Yb! \bullet Y!R_i\ ]]\\
\qquad\quad \rangle\\
\qquad\quad \parallel\ *[[\langle [\!]\ i:\ \neg b_i \wedge \overline{Zb} \wedge (\mathrm{r2} = i) \longrightarrow b_i\uparrow; Zb! \parallel Z?R_i; b_i\downarrow\ \rangle]]
\end{array}
$$

We have omitted the hardware stack and the $pc+3$ operation; this addition can be done on the top of the stack.

8.4 Process Decomposition of *PCunit*

In this section we take the *PCunit* process from the previous program and decompose it into simpler and more efficient processes. The decomposition is directed by the energy cost of the different operations on the *pc* register and by the frequency with which those operations occur.

The *pc* register, as is used in the previous program, is a fairly complicated register. It has 3 input ports, 2 output ports, and 2 different operations (+2 and +1) that have to be executed on the register. The +1 and +2 operations have to be executed very often compared to, for example, the $P?pc$ operation, and the overhead of that operation should not increase the cost of the increment.

Without loss of generality, we simplify the *PCunit* program by eliminating the d_i and r_i registers and express the assignment of *or* to *pc* as a communication action.

$$
\begin{array}{lll}
PCunit \equiv *[[& \overline{Ipc} \longrightarrow & Ipc? \parallel pc := pc + 1 \\
& [] \; \overline{I2pc} \longrightarrow & I2pc? \parallel pc := pc + 2 \\
& [] \; \overline{Pullpc} \longrightarrow & Pullpc? \parallel Pull?pc \\
& [] \; \overline{Pushpc} \longrightarrow & Pushpc? \parallel Push!pc \\
& [] \; \overline{Apc} \longrightarrow & Apc? \bullet A!pc \\
& [] \; \overline{Orpc} \longrightarrow & Orpc? \parallel Or?pc \\
& [] \; \overline{Ppc} \longrightarrow & Ppc? \parallel P?pc \\
&]] &
\end{array}
$$

Next, to reduce the cost of the increment actions, we reduce the number of ports on the *pc* register to one for read, and one for write; we then add multiplexors and de-multiplexors for the other ports. We increase the cost of the read and write operations, but the overall cost per operation is expected to improve.

$$
\begin{array}{lll}
PCunit \equiv *[[& \overline{Ipc} \longrightarrow & Ipc? \parallel pc := pc + 1 \\
& [] \; \overline{I2pc} \longrightarrow & I2pc? \parallel pc := pc + 2 \\
& [] \; \overline{Rpc} \longrightarrow & Rpc!pc \\
& [] \; \overline{Wpc} \longrightarrow & Wpc?pc \\
&]] &
\end{array}
$$

$$
\begin{array}{lll}
PCmux \equiv *[[& \overline{Pushpc} \longrightarrow & Pushpc? \bullet Push!(Rpc?) \\
[] & \overline{Apc} \longrightarrow & Apc? \bullet A!(Rpc?) \\
]] & &
\end{array}
$$

$$
\begin{array}{ll}
PCdmx \equiv *[[& \overline{Pullpc} \longrightarrow Pullpc? \bullet Wpc!(Pull?) \\
[] & \overline{Orpc} \longrightarrow Orpc? \bullet Wpc!(Or?) \\
[] & \overline{Ppc} \longrightarrow Ppc? \bullet Wpc!(P?) \\
]] &
\end{array}
$$

This transformation also attempts to make the probability of each statement in the guarded commands more equilibrated.

Next we break up the incrementer. Half of the time, the incrementer only has to flip one bit, and we use this to eliminate the assignment to pc in the first two guarded commands. We express pc as an array of bits, $pc[0..n-1]$.

$$
\begin{array}{ll}
PCunit \equiv *[[& \overline{Ipc} \wedge \neg pc[0] \longrightarrow pc[0]\uparrow \parallel Ipc \\
[] & \overline{Ipc} \wedge \ \ pc[0] \longrightarrow pc[0]\downarrow \parallel Ipc1 \parallel Ipc \\
[] & \overline{I2pc} \longrightarrow Ipc1 \parallel I2pc \\
[] & \overline{Rpc} \longrightarrow \ Rpc!pc \\
[] & \overline{Wpc} \longrightarrow \ Wpc?pc \\
]] &
\end{array}
$$

$$
PCinc1 \equiv *[[\ \overline{Ipc1} \longrightarrow pc := pc + 2; Ipc1\ ]]
$$

We can use the same transformation as before to improve the $PCinc1$ process:

$$
\begin{array}{ll}
PCinc1 \equiv *[[& \overline{Ipc1} \wedge \neg pc[1] \longrightarrow pc[1]\uparrow; Ipc1 \\
[] & \overline{Ipc1} \wedge \ \ pc[1] \longrightarrow pc[1]\downarrow; Ipc2; Ipc1 \\
]] &
\end{array}
$$

$$
PCinc2 \equiv *[[\ \overline{Ipc2} \longrightarrow pc := pc + 4; Ipc2\ ]]
$$

We transform $PCinc2$ in the same way recursively until all bits from pc have been taken care of.

8.5 Summary & Conclusion

We have shown in this chapter how to apply a number of different low-energy techniques to the design of an asynchronous microprocessor. Each aspect of the design is potentially affected by energy considerations. The instruction set was examined to provide all of the functionality of a processor, but implementing each function at a low cost. For example, an on-chip hardware stack was chosen to implement sub-routine calls because it can be used as a cache for the call stack in main memory with a very high hit ratio. Also, pipelining was used to take advantage of the registers already present in the processor, since we can then improve throughput without a cost in energy performance.

A characteristic of this processor design is the split instruction fetch. The instruction register contains at the same time part of the current instruction and part of the next instruction. This scheme allows us to split the instruction register in two and gain one pipeline stage without having to increase the number of registers, therefore without a cost in energy. It also allows efficient pre-decoding of the instruction and thus we are able to set-up the decoding circuit for the specific type of instruction contained in the next program location. Prefetch mechanisms are facilitated, since the processor knows ahead of time whether the next instruction contains a branch or not.

The trade-off between worst-case delay and average energy dissipation is used extensively to optimize certain parts of the architecture, like the PCunit.

Chapter 9

Conclusion & Future Work

9.1 Conclusion

In this thesis we have shown a number of results to help us design energy-efficient circuits. Energy efficiency was broken down in two parts: algorithmic efficiency and circuit efficiency. The idea is that to achieve a low-energy dissipation in a given computation, we can optimize independently both the algorithm used to make the computation and the circuit used to implement the algorithm.

This analysis leads to three questions:

How is energy spent? What are the physical mechanisms that dissipate energy during the computation?

Where is energy spent? What parts of the circuit spend energy and how much?

Why is energy spent? Why do I have to spend any energy to perform a computation? Is there a better circuit that I do not know about that is essentially better than the one I have?

9.1.1 How?

The first question has a simple answer. The transistor equations used in simulators such as SPICE are a good approximation of reality, and we can make a very precise circuit-level computation of energy dissipation. Factors like capacitor charge and discharge currents dominate the energy dissipation of CMOS logic circuits, which further simplifies the problem. As a result, we can design efficient CMOS gates and other basic building blocks for digital circuits; we can produce simple and accurate models for energy dissipation; and we can investigate other alternatives to the usual CMOS implementation of digital circuits.

9.1.2 Where?

Having good building blocks is, of course, not enough. We have to know how is it that they fit in the architecture of the circuit we are building. The approach we have taken to bridge this gap is very much related to the way we design asynchronous circuits, though it is general enough that it can be extended to other design methodologies.

An asynchronous circuit is first specified as a collection of sequential, communicating processes, a CSP program. This program is then refined through successive semantics-preserving transformations, and finally, when we get a satisfactory program, we translate this program into a CMOS circuit through a systematic compilation procedure. As a consequence, there is a very good match between the text of the program to be compiled and the CMOS circuit product of the compilation. Before the compilation takes place, we can already tell what the final circuit looks like and make an estimate of the energy consumption of each of the parts of that circuit based on the simple models derived from the circuit equations for CMOS gates.

We are now ready to answer the question "where is energy spent?" We can use a trace, or set of traces, of the execution of the program to estimate how often each part the circuit is in use. By tallying the energies required for each usage, we can profile the energy dissipation of the program we are going to implement and know exactly where that energy goes.

The answer to "where?" is the first step in the analysis of a system. Knowing exactly where most of the energy goes allows us to concentrate our effort on those parts of the computation that are the most wasteful. This answer is presented here as the energy complexity of the CSP program that performs the computation. The energy complexity of a program is similar to the time complexity of that program when all concurrency has been removed and each step of the computation is weighed to take into account the extra energy required to deal with fan-in and fan-out of operators.

We have now separated the two basic concerns in energy optimization. The energy required to perform a computation P, $E(\mathrm{P})$, can be expressed as:

$$E(\mathrm{P}) = E_t \times \mathcal{C}(\mathrm{P}) \tag{9.1}$$

where E_t is the energy required to perform an elementary transition and $\mathcal{C}(\mathrm{P})$ is the energy complexity of P.

The constant E_t depends on the CMOS technology employed and the design style for CMOS gates. To improve on E_t, we can try to make the technology better by reducing parasitic capacitances, increasing the number of wiring layers, and reducing threshold voltages to allow for lower power supplies, among other measures. Or we can try to make the gates that we employ better by utilizing pass gate transistor logic, quasi-adiabatic switching, reduced voltage swing logic, sense-amplifier schemes for buses, and so on. We do not deal in this thesis with technology aspects; some circuits are proposed, however, to implement some of the datapath constructs. In particular, we present a sense-amplifier circuit that can be used to take advantage of the implicit intermediate register in a bus transfer (the bus wires themselves), to pipeline the bus transfer, and reduce the cycle time without affecting the energy cost of the transfer.

Most of the focus of this thesis is directed towards the energy complexity of programs. QDI (quasi delay insensitive) circuits have shown to be energy efficient for algorithmic reasons: programs are designed as reactive loops, and energy is spent only in useful computation. Being careful about the design, we can further reduce the energy complexity of the algorithm that implements the computation and get more energy savings.

There is another good reason to target complexity instead of technology. Technology is continuously evolving, and it is possible that one day CMOS circuits will become obsolete when another new, better, technology comes about. Complexity results will then still be relevant because they are related to the mathematical representation of the computation, not to its physical embodiment.

We have shown how to derive a complexity model for CSP programs and how to use this model to help in the analysis and design of asynchronous circuits. In particular, if the algorithm is parametrized (as in the case of the memory design example), we can minimize the energy-complexity equation and, thus, determine the missing parameters.

9.1.3 Why?

So far, we have learned how energy is spent and where in our circuit we are spending it. The idea is that if we know how and where, we can reduce some of that energy dissipation without actually introducing more dissipation in the process.

But how much of that dissipation can be eliminated? We need to know why we are dissipating energy in a computation and whether it is unavoidable. In this case, our approach was to find a lower bound to the energy complexity of the computation. If the energy complexity of the program we have designed gets close to its lower bound, we know there is no room for improvement.

This lower bound is based on the input/output behavior of the specification of the problem. By looking at the input/output behavior, we have abstracted the specification from any choice that we might have made about the implementation. If the program we design has an energy complexity close to the lower bound, we know why we are spending that much energy: because we have to.

Knowing the lower bound helps us in other ways too. Every time we make a decision about the implementation of a specification, we restrict the number of possible solutions. We can validate those decisions by checking that we have not thrown away all of the interesting solutions (that is, check that the lower bound has not gone up by too much). We can thus validate implementation strategies: program approximation, proper separation of control and data, use of concurrency, removal of unnecessary synchronization, etc..

We can also make an analysis of the specification and try to simplify this specification. By knowing what is hard about the specification, we know what to relax and change so that the net effect is positive.

The arguments used to derive the lower bound are purely information-theoretical. The basic intuition behind them is that to generate a more complex string of input/output actions, we need a more complex machine, with more internal states, and therefore it is more expensive to run. This intuition can be quantized precisely in the equation:

$$\mathcal{H}(\mathrm{P}) \leq \mathcal{C}(\mathrm{P}) \tag{9.2}$$

where $\mathcal{H}(\mathrm{P})$ is the average entropy per symbol of the string of input/output symbols of process P, and $\mathcal{C}(\mathrm{P})$ is the average energy cost per symbol of running process P. We have, at this point, all the mathematical tools of information theory at our disposal to manipulate program specifications.

9.2 Future Work

The work on this thesis was aimed exclusively at energy cost of computation. Energy efficiency is achieved by simplifying the computation, so that fewer steps are necessary to finish it and, therefore, less energy has to be spent. The energy complexity approach assumes that time efficiency is going to follow from the energy efficiency as well, which may not be true under all circumstances.

An extension of this work would be to include a measure of time complexity into the equations, based as well on the CSP specification of the circuit, and combine both measures (E and T) into an ET^α index of performance.

Defining a time complexity index for CSP programs is simple; computing that index can be, however, a complex task because of the effect of concurrency. Time complexity is, however, well understood, and the reader can easily make this extension if, for example, he requires an ET^2 index of performance.

Another interesting point of research is to try to find an information-theoretical lower bound to the ET^α measures of performance. The bound for the energy was easy to find because of the additive nature of both energy dissipation and entropy. Time is not additive; two operations performed in parallel take as much time as the longest of the two, not as the sum, so a different approach would be necessary to account for this effect. We would have to consider the time cost of concurrency: starting n actions in parallel has a cost proportional to $\log_2 n$, that is required to distribute the *start* signal to all of the n processes, and collect the *done* signals form all of these processes. Based on this cost, we may be able to derive a similar lower bound for time complexity.

Finally, many of the results in this thesis are quantitative in nature, as the different energy complexity models, but are hard to compute. Some amount of CAD support is needed to simplify those computations and help direct the high level synthesis procedure. In particular, many aspects of high-level synthesis can be directed through energy-model considerations, as register and bus assignment or memory and cache dimensioning.

Bibliography

[1] Y. S. Abu-Mostafa, *Complexity of information extraction*, Ph.D. thesis, California Institute of Technology, 1983.

[2] C. H. Bennett, *The thermodynamics of computation—a review*, International J. of Theoretical Physics **21** (1982), no. 12, 905–940.

[3] S. M. Burns, *Automated compilation of concurrent programs into self-timed circuits*, Cs-tr-88-2, California Institute of Technology, December 1987.

[4] Steven M. Burns, *Performance analysis and optimization of asynchronous circuits*, Ph.D. thesis, California Institute of Technology, 1991.

[5] G. Chaitin, *Information-theoretic computational complexity*, IEEE Trans. on Information Theory **IT-20** (1974), 10–15.

[6] H. Y. Chen, *Design automation for high performance CMOS VLSI*, Ph.D. thesis, University of Illinois at Urbana-Champaign, December 1988.

[7] S. T. Chu, J. Dikken, C. D. Hartgring, F. J. List, J. G. Raemaekers, S. A. Bell, B. Walsh, and R. H. W. Salters, *A 25-ns low-power full-CMOS 1-Mbit (128K$\times$8) SRAM*, IEEE J. of Solid-State Circuits **SC-23** (1988), no. 5, 1078–1084.

[8] W. E. Donath, *Placement and average interconnection lengths of computer logic*, IEEE Transactions on Circuits and Systems **CAS-26** (1979), 272–277.

[9] W. C. Elmore, *The transient response of damped linear networks with particular regard to wideband amplifiers*, J. of Applied Physics **19** (1948), 55–63.

[10] R. P. Feynman, *Quantum mechanical computers*, Foundations of Physics **16** (1986), no. 6, 507–531.

[11] J. P. Fishburn and A. E. Dunlop, *TILOS: A posynomial approach to transistor sizing*, Proceedings of the 1985 International Conference on Computer-aided Design, November 1985, pp. 326–328.

[12] E. Fredkin and T. Toffoli, *Conservative logic*, International J. of Theoretical Physics **21** (1982), no. 3/4, 219–253.

[13] J. L. Hennessy and D. A. Patterson, Computer Architecture a Quantitative Approach, ch. 6, pp. 273–278, Morgan Kaufmann Publishers Inc., 1990, pp. 273–278.

[14] C. A. R. Hoare, *Communicating sequential processes*, Communications of the ACM **21** (1978), no. 8, 666–677.

[15] J. S. Hwang and C. Y. Wu, *Efficient techniques in the sizing and constrained optimisation of CMOS combinational logic circuits*, IEE Proceedings-E **138** (1991), no. 3, 154–164.

[16] T. K. Lee, *Communication behavior of linear arrays of processes*, Cs-tr-89-12, California Institute of Technology, December 1989.

[17] ———, *Energy and delay measurements of an asynchronous 3x+1 engine*, Personal Communication, 1993.

[18] Alain J. Martin, *Compiling communicating processes into delay-insensitive VLSI circuits*, Distributed Computing **1** (1986), no. 4, 226–234.

[19] Alain J. Martin, *The design of a delay-insensitive microprocessor: An example of circuit synthesis by program transformation*, Hardware Specification, Verification and Synthesis: Mathematical Aspects (M. Leeser and G. Brown, eds.), Lecture Notes in Computer Science, vol. 408, Springer-Verlag, 1989, pp. 244–259.

[20] Alain J. Martin, *Asynchronous datapaths and the design of an asynchronous adder*, Formal Methods in System Design **1** (1992), no. 1, 119–137.

[21] Alain J. Martin, Steven M. Burns, T. K. Lee, Drazen Borkovic, and Pieter J. Hazewindus, *The design of an asynchronous microprocessor*, Advanced Research in VLSI: Proceedings of the Decennial Caltech Conference on VLSI (Charles L. Seitz, ed.), MIT Press, 1989, pp. 351–373.

[22] ———, *The first asynchronous microprocessor: the test results*, Computer Architecture News **17** (1989), no. 4, 95–110.

[23] M. Matson, *Macromodeling and optimization of digital MOS VLSI circuits*, Ph.D. thesis, MIT, February 1985.

[24] L. Nagel, *SPICE2: A computer program to simulate semiconductor circuits.*, Tech. Report ERL–M520, University of California, Berkeley, May 1975.

[25] V. B. Rao, *Switch-level timing simulation of MOS VLSI circuits*, Ph.D. thesis, University of Illinois at Urbana Champaign, January 1985.

[26] J. Rubenstein, P. Penfield, and M. A. Horowitz, *Signal delay in RC tree networks*, IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems **2** (1983), no. 3, 202–211.

[27] S. S. Sapatnekar and S. M. Kang, Design Automation for Timing-Driven Layout Synthesis, ch. 4, pp. 113–140, Kluwer Academic Publishers, 1993, pp. 113–140.

[28] C. Shannon, *A mathematical theory of communication*, Bell Systems Tech. J. **27** (1948), 379–423.

[29] P. Single, *The theory of logical effort and overhead*, Proceedings 7^{th} Australian Microelectronics Conference, May 1988.

[30] I. E. Sutherland and R. E. Sproull, *Logical effort: Designing for speed on the back of an envelope*, Advanced Research in VLSI (C. H. Séquin, ed.), The MIT Press, 1991, pp. 1–16.

[31] Ivan E. Sutherland, *Micropipelines*, Communications of the ACM **32** (1989), no. 6, 720–738.

[32] José A. Tierno and Alain J. Martin, *Low-energy asynchronous memory design*, Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems, November 1994, pp. 176–185.

[33] R. R. Troutman, *Subthreshold design considerations for insulated gate field effect transistors*, IEEE J. of Solid State Circuits **SC-9** (1974), 55–60.

[34] H. J. M. Veendrick, *Short circuit dissipation of static CMOS circuitry and its impact on the design of buffer circuits*, IEEE J. of Solid-State Circuits **SC-19** (1984), no. 4, 468–473.

[35] M. Yoshimoto, K. Anami, H. Shinohara, T. Yoshihara, T. Takagi, and T. Nakano, *A divided word-line structure in the static RAM and its application to a 64K full CMOS RAM*, IEEE J. of Solid-State Circuits **SC-18** (1983), no. 5, 479–485.