Remark 15.4 The step sizes α_k^P and α_k^D are chosen to keep $\mathbf{x}_{k+1}$ and $\mathbf{z}_{k+1}$ strictly positive. The ability in the primal–dual scheme to choose separate step sizes for the primal and dual variables is a major advantage that this method has over the pure primal or dual methods. Empirically this advantage translates to a significant reduction in the number of iterations.

Remark 15.5 The stopping condition essentially checks for primal and dual feasibility and near complementary slackness. Exact complementary slackness is not possible with interior solutions. It is possible to maintain primal and dual feasibility through the algorithm, but this would require a phase I construction via artificial variables. Empirically, this feasible variant has not been found to be worthwhile. In any case, when the algorithm terminates with an interior solution, a post-processing step is usually invoked to obtain optimal extreme point solutions for the primal and dual. This is usually called the *purification* of solutions and is based on a clever scheme described by Megiddo [1991].

Remark 15.6 Instead of using Newton steps to drive the solutions to satisfy the optimality conditions of (D_μ), Mehrotra [1992] suggested a predictor–corrector approach based on power series approximations. This approach has the added advantage of providing a rational scheme for reducing the value of μ. It is the predictor–corrector based primal–dual interior method that is considered the current winner in interior point methods. The OB1 code of Lustig et al. [1994] is based on this scheme.

Remark 15.7 CPLEX 6.5 [1999], a general purpose linear (and integer) programming solver, contains implementations of interior point methods. A computational study of parallel implementations of simplex and interior point methods on the SGI power challenge (SGI R8000) platform indicates that on all but a few small linear programs in the NETLIB linear programming benchmark problem set, interior point methods dominate the simplex method in run times. New advances in handling Cholesky factorizations in parallel are apparently the reason for this exceptional performance of interior point methods. For the simplex method, CPLEX 6.5 incorporates efficient methods of solving triangular linear systems and faster updating of reduced costs for identifying improving edge directions. For the interior point method, the same code includes improvements in computing Cholesky factorizations and better use of level-two cache available in modern computing architectures. Using CPLEX 6.5 and CPLEX 5.0, Bixby et al. [2001] in a recent study have done extensive computational testing comparing the two codes with respect to the performance of the Primal simplex, Dual simplex and Interior Point methods as well as a comparison of the performance of these three methods. While CPLEX 6.5 considerably outperformed CPLEX 5.0 for all the three methods, the comparison among the three methods is inconclusive. However, as stated by Bixby et al. [2001], the computational testing was biased against interior point method because of the inferior floating point performance of the machine used and the nonimplementation of the parallel features on shared memory machines.

Remark 15.8 Karmarkar [1990] has proposed an interior-point approach for integer programming problems. The main idea is to reformulate an integer program as the minimization of a quadratic energy function over linear constraints on continuous variables. Interior-point methods are applied to this formulation to find local optima.

15.3 Large-Scale Linear Programming in Combinatorial Optimization

Linear programming problems with thousands of rows and columns are routinely solved either by variants of the simplex method or by interior point methods. However, for several linear programs that arise in combinatorial optimization, the number of columns (or rows in the dual) are too numerous to be enumerated explicitly. The columns, however, often have a structure which is exploited to generate the columns as and when required in the simplex method. Such an approach, which is referred to as **column**

generation, is illustrated next on the *cutting stock problem* (Gilmore and Gomory [1963]), which is also known as the *bin packing problem* in the computer science literature.

15.3.1 Cutting Stock Problem

Rolls of sheet metal of standard length L are used to cut required lengths $l_i, i = 1, 2, \ldots, m$. The jth cutting pattern should be such that a_{ij}, the number of sheets of length l_i cut from one roll of standard length L, must satisfy $\sum_{i=1}^{m} a_{ij} l_i \leq L$. Suppose $n_i, i = 1, 2, \ldots, m$ sheets of length l_i are required. The problem is to find cutting patterns so as to minimize the number of rolls of standard length L that are used to meet the requirements. A linear programming formulation of the problem is as follows.

Let $\mathbf{x}_j$, $j = 1, 2, \ldots, n$, denote the number of times the jth cutting pattern is used. In general, $\mathbf{x}_j$, $j = 1, 2, \ldots, n$ should be an integer but in the next formulation the variables are permitted to be fractional.

$$
\begin{aligned}
\text{(P1)} \quad & \text{Min} \sum_{j=1}^{n} \mathbf{x}_j \\
\text{Subject to} \quad & \sum_{j=1}^{n} a_{ij} \mathbf{x}_j \geq n_i \qquad i = 1, 2, \ldots, m \\
& \mathbf{x}_j \geq 0 \qquad j = 1, 2, \ldots, n \\
\text{where} \quad & \sum_{i=1}^{m} l_i a_{ij} \leq L \qquad j = 1, 2, \ldots, n
\end{aligned}
$$

The formulation can easily be extended to allow for the possibility of p standard lengths L_k, $k = 1, 2, \ldots, p$, from which the n_i units of length $l_i, i = 1, 2, \ldots, m$, are to be cut.

The cutting stock problem can also be viewed as a bin packing problem. Several bins, each of standard capacity L, are to be packed with n_i units of item i, each of which uses up capacity of l_i in a bin. The problem is to minimize the number of bins used.

15.3.1.1 Column Generation

In general, the number of columns in (P1) is too large to enumerate all of the columns explicitly. The simplex method, however, does not require all of the columns to be explicitly written down. Given a basic feasible solution and the corresponding simplex multipliers $\mathbf{w}_i, i = 1, 2, \ldots, m$, the column to enter the basis is determined by applying dynamic programming to solve the following knapsack problem:

$$
\begin{aligned}
\text{(P2)} \quad & z = \text{Max} \sum_{i=1}^{m} \mathbf{w}_i a_i \\
\text{Subject to} \quad & \sum_{i=1}^{m} l_i a_i \leq L \\
& a_i \geq 0 \text{ and integer}, \quad \text{for } i = 1, 2, \ldots, m
\end{aligned}
$$

Let a_i^*, $i = 1, 2, \ldots, m$, denote an optimal solution to (P2). If $z > 1$, the kth column to enter the basis has coefficients $a_{ik} = a_i^*$, $i = 1, 2, \ldots, m$.

Using the identified columns, a new improved (in terms of the objective function value) basis is obtained, and the column generation procedure is repeated. A major iteration is one in which (P2) is solved to identify, if there is one, a column to enter the basis. Between two major iterations, several minor iterations may be performed to optimize the linear program using only the available (generated) columns.

If $z \leq 1$, the current basic feasible solution is optimal to (P1). From a computational point of view, alternative strategies are possible. For instance, instead of solving (P2) to optimality, a column to enter the basis can be indentified as soon as a feasible solution to (P2) with an objective function value greater than 1 has been found. Such an approach would reduce the time required to solve (P2) but may increase the number of iterations required to solve (P1).

A column once generated may be retained, even if it comes out of the basis at a subsequent iteration, so as to avoid generating the same column again later on. However, at a particular iteration some columns, which appear unattractive in terms of their reduced costs, may be discarded in order to avoid having to store a large number of columns. Such columns can always be generated again subsequently, if necessary. The rationale for this approach is that such unattractive columns will rarely be required subsequently.

The dual of (P1) has a large number of rows. Hence column generation may be viewed as row generation in the dual. In other words, in the dual we start with only a few constraints explicitly written down. Given an optimal solution $\mathbf{w}$ to the current dual problem (i.e., with only a few constraints which have been explicitly written down) find a constraint that is violated by $\mathbf{w}$ or conclude that no such constraint exists. The problem to be solved for identifying a violated constraint, if any, is exactly the separation problem that we encountered in the section on algorithms for linear programming.

15.3.2 Decomposition and Compact Representations

Large-scale linear programs sometimes have a block diagonal structure with a few additional constraints linking the different blocks. The linking constraints are referred to as the master constraints and the various blocks of constraints are referred to as subproblem constraints. Using the representation theorem of polyhedra (see, for instance, Nemhauser and Wolsey [1988]), the decomposition approach of Dantzig and Wolfe [1961] is to convert the original problem to an equivalent linear program with a small number of constraints but with a large number of columns or variables. In the cutting stock problem described in the preceding section, the columns are generated, as and when required, by solving a knapsack problem via dynamic programming. In the Dantzig–Wolfe decomposition scheme, the columns are generated, as and when required, by solving appropriate linear programs on the subproblem constraints.

It is interesting to note that the reverse of decomposition is also possible. In other words, suppose we start with a statement of a problem and an associated linear programming formulation with a large number of columns (or rows in the dual). If the column generation (or row generation in the dual) can be accomplished by solving a linear program, then a *compact* formulation of the original problem can be obtained. Here compact refers to the number of rows and columns being bounded by a polynomial function of the input length of the original problem. This result due to Martin [1991] enables one to solve the problem in the polynomial time by solving the compact formulation using interior point methods.

15.4 Integer Linear Programs

Integer linear programming problems (ILPs) are linear programs in which all of the variables are restricted to be integers. If only some but not all variables are restricted to be integers, the problem is referred to as a mixed integer program. Many combinatorial problems can be formulated as integer linear programs in which all of the variables are restricted to be 0 or 1. We will first discuss several examples of combinatorial optimization problems and their formulation as integer programs. Then we will review a general representation theory for integer programs that gives a formal measure of the expressiveness of this algebraic approach. We conclude this section with a representation theorem due to Benders [1962], which has been very useful in solving certain large-scale combinatorial optimization problems in practice.

15.4.1 Example Formulations

15.4.1.1 Covering and Packing Problems

A wide variety of location and scheduling problems can be formulated as set covering or set packing or set partitioning problems. The three different types of **covering and packing** problems can be succinctly stated as follows: Given (1) a finite set of elements $\mathcal{M} = \{1, 2, \ldots, m\}$, and (2) a family F of subsets of $\mathcal{M}$ with each member F_j, $j = 1, 2, \ldots, n$ having a profit (or cost) $\mathbf{c}_j$ associated with it, find a collection, S,

of the members of F that maximizes the profit (or minimizes the cost) while ensuring that every element of $\mathcal{M}$ is in one of the following:

(P3): at most one member of S (set packing problem)
(P4): at least one member of S (set covering problem)
(P5): exactly one member of S (set partitioning problem)

The three problems (P3), (P4), and (P5) can be formulated as ILPs as follows:

Let A denote the $m \times n$ matrix where

$$A_{ij} = \begin{cases} 1 & \text{if element } i \in F_j \\ 0 & \text{otherwise} \end{cases}$$

The decision variables are $\mathbf{x}_j$, $j = 1, 2, \ldots, n$ where

$$\mathbf{x}_{ij} = \begin{cases} 1 & \text{if } F_j \text{ is chosen} \\ 0 & \text{otherwise} \end{cases}$$

The set packing problem is

$$\begin{aligned} & \text{(P3)} \quad \text{Max } \mathbf{cx} \\ \text{Subject to} \quad & A\mathbf{x} \leq \mathbf{e}_m \\ & \mathbf{x}_j = 0 \quad \text{or} \quad 1, \quad j = 1, 2, \ldots, n \end{aligned}$$

where $\mathbf{e}_m$ is an m-dimensional column vector of ones.

The set covering problem (P4) is (P3) with less than or equal to constraints replaced by greater than or equal to constraints and the objective is to minimize rather than maximize. The set partitioning problem (P5) is (P3) with the constraints written as equalities. The set partitioning problem can be converted to a set packing problem or set covering problem (see Padberg [1995]) using standard transformations. If the right-hand side vector $\mathbf{e}_m$ is replaced by a nonnegative integer vector $\mathbf{b}$, (P3) is referred to as the generalized set packing problem.

The airline crew scheduling problem is a classic example of the set partitioning or the set covering problem. Each element of $\mathcal{M}$ corresponds to a flight segment. Each subset F_j corresponds to an acceptable set of flight segments of a crew. The problem is to cover, at minimum cost, each flight segment exactly once. This is a set partitioning problem. If *dead heading* of crew is permitted, we have the set covering problem.

15.4.1.2 Packing and Covering Problems in a Graph

Suppose A is the node-edge incidence matrix of a graph. Now, (P3) is a weighted matching problem. If in addition, the right-hand side vector $\mathbf{e}_m$ is replaced by a nonnegative integer vector $\mathbf{b}$, (P3) is referred to as a weighted $\mathbf{b}$-matching problem. In this case, each variable $\mathbf{x}_j$ which is restricted to be an integer may have a positive upper bound of u_j. Problem (P4) is now referred to as the weighted edge covering problem. Note that by substituting for $\mathbf{x}_j = 1 - \mathbf{y}_j$, where $\mathbf{y}_j = 0$ or 1, the weighted edge covering problem is transformed to a weighted $\mathbf{b}$-matching problem in which the variables are restricted to be 0 or 1.

Suppose A is the edge-node incidence matrix of a graph. Now, (P3) is referred to as the weighted vertex packing problem and (P4) is referred to as the weighted vertex covering problem. The *set packing* problem can be transformed to a weighted vertex packing problem in a graph G as follows:

> G contains a node for each $\mathbf{x}_j$ and an edge between nodes j and k exists if and only if the columns $A_{\cdot j}$ and $A_{\cdot k}$ are not orthogonal. G is called the *intersection graph* of A. The set packing problem is equivalent to the weighted vertex packing problem on G. Given G, the complement graph $\overline{G}$ has the same node set as G and there is an edge between nodes j and k in $\overline{G}$ if and only if there is no such corresponding edge in G. A clique in a graph is a subset, k, of nodes of G such that the subgraph induced by k is complete. Clearly, the weighted vertex packing problem in G is equivalent to finding a maximum weighted clique in $\overline{G}$.

15.4.1.3 Plant Location Problems

Given a set of customer locations $N = \{1, 2, \ldots, n\}$ and a set of potential sites for plants $M = \{1, 2, \ldots, m\}$, the plant location problem is to identify the sites where the plants are to be located so that the customers are served at a minimum cost. There is a fixed cost $\mathbf{f}_i$ of locating the plant at site i and the cost of serving customer j from site i is $\mathbf{c}_{ij}$. The decision variables are: $\mathbf{y}_i$ is set to 1 if a plant is located at site i and to 0 otherwise; $\mathbf{x}_{ij}$ is set to 1 if site i serves customer j and to 0 otherwise.

A formulation of the problem is

$$\text{(P6)} \quad \text{Min} \sum_{i=1}^{m} \sum_{j=1}^{n} \mathbf{c}_{ij}\mathbf{x}_{ij} + \sum_{i=1}^{m} \mathbf{f}_i \mathbf{y}_i$$

$$\begin{aligned}
\text{subject to} \quad & \sum_{i=1}^{m} \mathbf{x}_{ij} = 1 && j = 1, 2, \ldots, n \\
& \mathbf{x}_{ij} - \mathbf{y}_i \leq 0 && i = 1, 2, \ldots, m; \quad j = 1, 2, \ldots, n \\
& \mathbf{y}_i = 0 \quad \text{or} \quad 1 && i = 1, 2, \ldots, m \\
& \mathbf{x}_{ij} = 0 \quad \text{or} \quad 1 && i = 1, 2, \ldots, m; \quad j = 1, 2, \ldots, n
\end{aligned}$$

Note that the constraints $\mathbf{x}_{ij} - \mathbf{y}_i \leq 0$ are required to ensure that customer j may be served from site i only if a plant is located at site i. Note that the constraints $\mathbf{y}_i = 0$ or 1 force an optimal solution in which $\mathbf{x}_{ij} = 0$ or 1. Consequently, the $\mathbf{x}_{ij} = 0$ or 1 constraints may be replaced by nonnegativity constraints $\mathbf{x}_{ij} \geq 0$.

The linear programming relaxation associated with (P6) is obtained by replacing constraints $\mathbf{y}_i = 0$ or 1 and $\mathbf{x}_{ij} = 0$ or 1 by nonnegativity contraints on $\mathbf{x}_{ij}$ and $\mathbf{y}_i$. The upper bound constraints on $\mathbf{y}_i$ are not required provided $\mathbf{f}_i \geq 0, i = 1, 2, \ldots, m$. The upper bound constraints on $\mathbf{x}_{ij}$ are not required in view of constraints $\sum_{i=1}^{m} \mathbf{x}_{ij} = 1$.

Remark 15.9 It is frequently possible to formulate the same combinatorial problem as two or more different ILPs. Suppose we have two ILP formulations (F1) and (F2) of the given combinatorial problem with both (F1) and (F2) being minimizing problems. Formulation (F1) is said to be stronger than (F2) if (LP1), the the linear programming relaxation of (F1), always has an optimal objective function value which is greater than or equal to the optimal objective function value of (LP2), which is the linear programming relaxation of (F2).

It is possible to reduce the number of constraints in (P6) by replacing the constraints $\mathbf{x}_{ij} - \mathbf{y}_i \leq 0$ by an aggregate:

$$\sum_{j=1}^{n} \mathbf{x}_{ij} - n\mathbf{y}_i \leq 0 \quad i = 1, 2, \ldots, m$$

However, the disaggregated (P6) is a stronger formulation than the formulation obtained by aggregrating the constraints as previously. By using standard transformations, (P6) can also be converted into a set packing problem.

15.4.1.4 Satisfiability and Inference Problems:

In propositional logic, a truth assignment is an assignment of true or false to each atomic proposition $\mathbf{x}_1, \mathbf{x}_2, \ldots \mathbf{x}_n$. A literal is an atomic proposition $\mathbf{x}_j$ or its negation $\neg\mathbf{x}_j$. For propositions in conjunctive normal form, a clause is a disjunction of literals and the proposition is a conjunction of clauses. A clause is obviously satisfied by a given truth assignment if at least one of its literals is true. The satisfiability problem consists of determining whether there exists a truth assignment to atomic propositions such that a set S of clauses is satisfied.

Let T_i denote the set of atomic propositions such that if any one of them is assigned true, the clause $i \in S$ is satisfied. Similarly, let F_i denote the set of atomic propositions such that if any one of them is assigned false, the clause $i \in S$ is satisfied.

The decision variables are

$$\mathbf{x}_j = \begin{cases} 1 & \text{if atomic proposition } j \text{ is assigned true} \\ 0 & \text{if atomic proposition } j \text{ is assigned false} \end{cases}$$

The satisfiability problem is to find a feasible solution to

$$\text{(P7)} \quad \sum_{j \in T_i} \mathbf{x}_j - \sum_{j \in F_i} \mathbf{x}_j \geq 1 - |F_i| \quad i \in S$$
$$\mathbf{x}_j = 0 \quad \text{or} \quad 1 \quad \text{for } j = 1, 2, \ldots, n$$

By substituting $\mathbf{x}_j = 1 - \mathbf{y}_j$, where $\mathbf{y}_j = 0$ or 1, for $j \in F_i$, (P7) is equivalent to the set covering problem

$$\text{(P8)} \quad \text{Min} \sum_{j=1}^{n} (\mathbf{x}_j + \mathbf{y}_j) \tag{15.3}$$

$$\text{subject to} \quad \sum_{j \in T_i} \mathbf{x}_j + \sum_{j \in F_i} \mathbf{y}_j \geq 1 \qquad i \in S \tag{15.4}$$

$$\mathbf{x}_j + \mathbf{y}_j \geq 1 \quad j = 1, 2, \ldots, n \tag{15.5}$$

$$\mathbf{x}_j, \mathbf{y}_j = 0 \quad \text{or} \quad 1 \quad j = 1, 2, \ldots, n \tag{15.6}$$

Clearly (P7) is feasible if and only if (P8) has an optimal objective function value equal to n.

Given a set S of clauses and an additional clause $k \notin S$, the logical inference problem is to find out whether every truth assignment that satisfies all of the clauses in S also satisfies the clause k. The logical inference problem is

$$\text{(P9)} \quad \text{Min} \quad \sum_{j \in T_k} \mathbf{x}_j - \sum_{j \in F_k} \mathbf{x}_j$$
$$\text{subject to} \quad \sum_{j \in T_i} \mathbf{x}_j - \sum_{j \in F_i} \mathbf{x}_j \geq 1 - |F_i| \quad i \in S$$
$$\mathbf{x}_j = 0 \quad \text{or} \quad 1 \quad j = 1, 2, \ldots, n$$

The clause k is implied by the set of clauses S, if and only if (P9) has an optimal objective function value greater than $-|F_k|$. It is also straightforward to express the MAX-SAT problem (i.e., find a truth assignment that maximizes the number of satisfied clauses in a given set S) as an integer linear program.

15.4.1.5 Multiprocessor Scheduling

Given n jobs and m processors, the problem is to allocate each job to one and only one of the processors so as to minimize the make span time, i.e., minimize the completion time of all of the jobs. The processors may not be identical and, hence, job j if allocated to processor i requires p_{ij} units of time. The multiprocessor scheduling problem is

$$\text{(P10)} \quad \text{Min } T$$
$$\text{subject to} \quad \sum_{i=1}^{m} \mathbf{x}_{ij} = 1 \quad j = 1, 2, \ldots, n$$
$$\sum_{j=1}^{n} \mathbf{p}_{ij} \mathbf{x}_{ij} - T \leq 0 \quad i = 1, 2, \ldots, m$$
$$\mathbf{x}_{ij} = 0 \quad \text{or} \quad 1$$

Note that if all $\mathbf{p}_{ij}$ are integers, the optimal solution will be such that T is an integer.

15.4.2 Jeroslow's Representability Theorem

Jeroslow [1989], building on joint work with Lowe in 1984, characterized subsets of n-space that can be represented as the feasible region of a mixed integer (Boolean) program. They proved that a set is the feasible region of some mixed integer/linear programming problem (MILP) if and only if it is the union of finitely many polyhedra having the same recession cone (defined subsequently). Although this result is not widely known, it might well be regarded as the fundamental theorem of mixed integer modeling.

The basic idea of Jeroslow's results is that any set that can be represented in a mixed integer model can be represented in a disjunctive programming problem (i.e., a problem with either/or constraints). A *recession direction* for a set S in n-space is a vector $\mathbf{x}$ such that $s + \alpha\mathbf{x} \in S$ for all $s \in S$ and all $\alpha \geq 0$. The set of recession directions is denoted $rec(S)$. Consider the general mixed integer constraint set

$$\begin{gathered} \mathbf{f}(\mathbf{x}, \mathbf{y}, \boldsymbol{\lambda}) \leq \mathbf{b} \\ \mathbf{x} \in \Re^n, \qquad \mathbf{y} \in \Re^p \\ \boldsymbol{\lambda} = (\lambda_1, \ldots, \lambda_k), \quad \text{with} \quad \boldsymbol{\lambda}_j \in \{0, 1\} \quad \text{for } j = 1, \ldots, k \end{gathered} \tag{15.7}$$

Here $\mathbf{f}$ is a vector-valued function, so that $\mathbf{f}(\mathbf{x}, \mathbf{y}, \boldsymbol{\lambda}) \leq \mathbf{b}$ represents a set of constraints. We say that a set $S \subset \Re^n$ is *represented* by Eq. (15.6) if,

$$\mathbf{x} \in S \quad \text{if and only if } (\mathbf{x}, \mathbf{y}, \boldsymbol{\lambda}) \text{ satisfies Eq. (15.6) for some } y, \lambda.$$

If $\mathbf{f}$ is a linear transformation, so that Equation 15.6 is a MILP constraint set, we will say that S is *MILP representable*. The main result can now be stated.

Theorem 15.2 [Jeroslow and Lowe 1984, Jeroslow 1989]. *A set in n-space is MILP representable if and only if it is the union of finitely many polyhedra having the same set of recession directions.*

15.4.3 Benders's Representation

Any mixed integer linear program can be reformulated so that there is only one continuous variable. This reformulation, due to Benders [1962], will in general have an exponential number of constraints. Analogous to column generation, discussed earlier, these rows (constraints) can be generated as and when required.

Consider the (MILP)

$$\max \{\mathbf{cx} + \mathbf{dy} : \ A\mathbf{x} + G\mathbf{y} \ \leq \ \mathbf{b}, \ \mathbf{x} \geq \mathbf{0}, \ \mathbf{y} \geq \mathbf{0} \text{ and integer}\}$$

Suppose the integer variables $\mathbf{y}$ are fixed at some values, then the associated linear program is

$$\text{(LP)} \quad \max \{\mathbf{cx} : \mathbf{x} \in \mathcal{P} = \{\mathbf{x} : \ A\mathbf{x} \leq \mathbf{b} - G\mathbf{y}, \ \mathbf{x} \geq \mathbf{0}\}\}$$

and its dual is

$$\text{(DLP)} \quad \min \{\mathbf{w}(\mathbf{b} - G\mathbf{y}) : \ \mathbf{w} \in \mathcal{Q} = \{\mathbf{w} : \ \mathbf{w}A \geq \mathbf{c}, \ \mathbf{w} \geq \mathbf{0}\}\}$$

Let $\{\mathbf{w}^k\}$, $k = 1, 2, \ldots, K$ be the extreme points of Q and $\{\mathbf{u}^j\}$, $j = 1, 2, \ldots, J$ be the extreme rays of the recession cone of Q, $\mathcal{C}_Q = \{\mathbf{u} : \ \mathbf{u}A \geq \mathbf{0}, \ \mathbf{u} \geq \mathbf{0}\}$. Note that if Q is nonempty, the $\{\mathbf{u}^j\}$ are all of the extreme rays of Q.

From linear programming duality, we know that if Q is empty and $\mathbf{u}^j(\mathbf{b} - G\mathbf{y}) \geq 0$, $j = 1, 2, \ldots, J$ for some $\mathbf{y} \geq \mathbf{0}$ and integer then (LP) and consequently (MILP) have an unbounded solution. If Q is nonempty and $\mathbf{u}^j(\mathbf{b} - G\mathbf{y}) \geq 0$, $j = 1, 2, \ldots, J$ for some $\mathbf{y} \geq \mathbf{0}$ and integer then (LP) has a finite optimum given by

$$\min_k \{\mathbf{w}^k(\mathbf{b} - G\mathbf{y})\}$$

Hence an equivalent formulation of (MILP) is

$$\begin{aligned} \text{Max } & \alpha \\ & \alpha \le \mathbf{d}\mathbf{y} + \mathbf{w}^k(\mathbf{b} - G\mathbf{y}), \quad k = 1, 2, \ldots, K \\ & \mathbf{u}^j(\mathbf{b} - G\mathbf{y}) \ge 0, \quad j = 1, 2, \ldots, J \\ & y \ge \mathbf{0} \text{ and integer} \\ & \alpha \quad \text{unrestricted} \end{aligned}$$

which has only one continuous variable α as promised.

15.5 Polyhedral Combinatorics

One of the main purposes of writing down an algebraic formulation of a combinatorial optimization problem as an integer program is to then examine the linear programming relaxation and understand how well it represents the discrete integer program. There are somewhat special but rich classes of such formulations for which the linear programming relaxation is sharp or tight. These correspond to linear programs that have integer valued extreme points. Such polyhedra are called **integral polyhedra**.

15.5.1 Special Structures and Integral Polyhedra

A natural question of interest is whether the LP associated with an ILP has only integral extreme points. For instance, the linear programs associated with matching and edge covering polytopes in a bipartite graph have only integral vertices. Clearly, in such a situation, the ILP can be solved as LP. A polyhedron or a polytope is referred to as being integral if it is either empty or has only integral vertices.

Definition 15.3 A $0, \pm 1$ matrix is totally unimodular if the determinant of every square submatrix is 0 or ± 1.

Theorem 15.3 [Hoffman and Kruskal 1956]. *Let*

$$A = \begin{pmatrix} A_1 \\ A_2 \\ A_3 \end{pmatrix}$$

be a $0, \pm 1$ *matrix and*

$$\mathbf{b} = \begin{pmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \\ \mathbf{b}_3 \end{pmatrix}$$

be a vector of appropriate dimensions. Then A is totally unimodular if and only if the polyhedron

$$P(A, \mathbf{b}) = \{\mathbf{x} : A_1\mathbf{x} \le \mathbf{b}_1; A_2\mathbf{x} \ge \mathbf{b}_2; A_3\mathbf{x} = \mathbf{b}_3; \mathbf{x} \ge 0\}$$

is integral for all integral vectors $\mathbf{b}$.

The constraint matrix associated with a network flow problem (see, for instance, Ahuja et al. [1993]) is totally unimodular. Note that for a given integral $\mathbf{b}$, $P(A, \mathbf{b})$ may be integral even if A is not totally unimodular.

Definition 15.4 A polyhedron defined by a system of linear constraints is totally dual integral (TDI) if for each objective function with integral coefficient the dual linear program has an integral optimal solution whenever an optimal solution exists.

Theorem 15.4 [Edmonds and Giles 1977]. *If* $P(A) = \{\mathbf{x} : A\mathbf{x} \leq \mathbf{b}\}$ *is TDI and* $\mathbf{b}$ *is integral, then* $P(A)$ *is integral.*

Hoffman and Kruskal [1956] have, in fact, shown that the polyhedron $P(A, \mathbf{b})$ defined in Theorem 15.3 is TDI. This follows from Theorem 15.3 and the fact that A is totally unimodular if and only if A^T is totally unimodular.

Balanced matrices, first introduced by Berge [1972] have important implications for packing and covering problems (see also Berge and Las Vergnas [1970]).

Definition 15.5 A 0, 1 matrix is balanced if it does not contain a square submatrix of odd order with two ones per row and column.

Theorem 15.5 [Berge 1972, Fulkerson et al. 1974]. *Let* A *be a balanced* 0, 1 *matrix. Then the set packing, set covering, and set partitioning polytopes associated with* A *are integral, i.e., the polytopes*

$$P(A) = \{\mathbf{x} : \mathbf{x} \geq \mathbf{0}; A\mathbf{x} \leq \mathbf{1}\}$$
$$Q(A) = \{\mathbf{x} : \mathbf{0} \leq \mathbf{x} \leq \mathbf{1}; A\mathbf{x} \geq \mathbf{1}\}$$
$$R(A) = \{\mathbf{x} : \mathbf{x} \geq \mathbf{0}; A\mathbf{x} = \mathbf{1}\}$$

are integral.

Let

$$A = \begin{pmatrix} A_1 \\ A_2 \\ A_3 \end{pmatrix}$$

be a balanced 0, 1 matrix. Fulkerson et al. [1974] have shown that the polytope $P(A) = \{\mathbf{x} : A_1\mathbf{x} \leq \mathbf{1}; A_2\mathbf{x} \geq \mathbf{1}; A_3\mathbf{x} = \mathbf{1}; \mathbf{x} \geq \mathbf{0}\}$ is TDI and by the theorem of Edmonds and Giles [1977] it follows that $P(A)$ is integral.

Truemper [1992] has extended the definition of balanced matrices to include $0, \pm 1$ matrices.

Definition 15.6 A $0, \pm 1$ matrix is balanced if for every square submatrix with exactly two nonzero entries in each row and each column, the sum of the entries is a multiple of 4.

Theorem 15.6 [Conforti and Cornuejols 1992b]. *Suppose* A *is a balanced* $0, \pm 1$ *matrix. Let* $\mathbf{n}(A)$ *denote the column vector whose* i*th component is the number of* -1*s in the* i*th row of* A*. Then the polytopes*

$$P(A) = \{\mathbf{x} : A\mathbf{x} \leq \mathbf{1} - \mathbf{n}(A); \mathbf{0} \leq \mathbf{x} \leq \mathbf{1}\}$$
$$Q(A) = \{\mathbf{x} : A\mathbf{x} \geq \mathbf{1} - \mathbf{n}(A); \mathbf{0} \leq \mathbf{x} \leq \mathbf{1}\}$$
$$R(A) = \{\mathbf{x} : A\mathbf{x} = \mathbf{1} - \mathbf{n}(A); \mathbf{0} \leq \mathbf{x} \leq \mathbf{1}\}$$

are integral.

Note that a $0, \pm 1$ matrix A is balanced if and only if A^T is balanced. Moreover, A is balanced (totally unimodular) if and only if every submatrix of A is balanced (totally unimodular). Thus, if A is balanced (totally unimodular) it follows that Theorem 15.6 (Theorem 15.3) holds for every submatrix of A.

Totally unimodular matrices constitute a subclass of balanced matrices, i.e., a totally unimodular $0, \pm 1$ matrix is always balanced. This follows from a theorem of Camion [1965], which states that a $0, \pm 1$ is totally unimodular if and only if for every square submatrix with an even number of nonzero entries in each row and in each column, the sum of the entries equals a multiple of 4. The 4×4 matrix in Figure 15.1

$$A = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix} \qquad A = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

FIGURE 15.1 A balanced matrix and a perfect matrix. (From Chandru, V. and Rao, M. R. Combinatorial optimization: an integer programming perspective. ACM *Comput. Surveys*, 28, 1. March 1996.)

illustrates the fact that a balanced matrix is not necessarily totally unimodular. Balanced $0, \pm 1$ matrices have implications for solving the satisfiability problem. If the given set of clauses defines a balanced $0, \pm 1$ matrix, then as shown by Conforti and Cornuejols [1992b], the satisfiability problem is trivial to solve and the associated MAXSAT problem is solvable in polynomial time by linear programming. A survey of balanced matrices is in Conforti et al. [1994].

Definition 15.7 A $0, 1$ matrix A is perfect if the set packing polytope $P(A) = \{\mathbf{x} : A\mathbf{x} \leq \mathbf{1}; \mathbf{x} \geq \mathbf{0}\}$ is integral.

The chromatic number of a graph is the minimum number of colors required to color the vertices of the graph so that no two vertices with the same color have an edge incident between them. A graph G is perfect if for every node induced subgraph H, the chromatic number of H equals the number of nodes in the maximum clique of H. The connections between the integrality of the set packing polytope and the notion of a perfect graph, as defined by Berge [1961, 1970], are given in Fulkerson [1970], Lovasz [1972], Padberg [1974], and Chvátal [1975].

Theorem 15.7 [Fulkerson 1970, Lovasz 1972, Chvátal 1975] *Let A be $0, 1$ matrix whose columns correspond to the nodes of a graph G and whose rows are the incidence vectors of the maximal cliques of G. The graph G is perfect if and only if A is perfect.*

Let G_A denote the intersection graph associated with a given $0, 1$ matrix A (see Section 15.4). Clearly, a row of A is the incidence vector of a clique in G_A. In order for A to be perfect, every maximal clique of G_A must be represented as a row of A because inequalities defined by maximal cliques are facet defining. Thus, by Theorem 15.7, it follows that a $0, 1$ matrix A is perfect if and only if the undominated (a row of A is dominated if its support is contained in the support of another row of A) rows of A form the clique-node incidence matrix of a perfect graph.

Balanced matrices with $0, 1$ entries, constitute a subclass of $0, 1$ perfect matrices, i.e., if a $0, 1$ matrix A is balanced, then A is perfect. The 4×3 matrix in Figure 15.1 is an example of a matrix that is perfect but not balanced.

Definition 15.8 A $0, 1$ matrix A is ideal if the set covering polytope

$$Q(A) = \{\mathbf{x} \ : \ A\mathbf{x} \geq \mathbf{1}; \mathbf{0} \leq \mathbf{x} \leq \mathbf{1}\}$$

is integral.

Properties of ideal matrices are described by Lehman [1979], Padberg [1993], and Cornuejols and Novick [1994]. The notion of a $0, 1$ perfect (ideal) matrix has a natural extension to a $0, \pm 1$ perfect (ideal) matrix. Some results pertaining to $0, \pm 1$ ideal matrices are contained in Hooker [1992], whereas some results pertaining to $0, \pm 1$ perfect matrices are given in Conforti et al. [1993].

An interesting combinatorial problem is to check whether a given $0, \pm 1$ matrix is totally unimodular, balanced, or perfect. Seymour's [1980] characterization of totally unimodular matrices provides a polynomial-time algorithm to test whether a given matrix $0, 1$ matrix is totally unimodular. Conforti

et al. [1999] give a polynomial-time algorithm to check whether a 0, 1 matrix is balanced. This has been extended by Conforti et al. [1994] to check in polynomial time whether a $0, \pm 1$ matrix is balanced. An open problem is that of checking in polynomial time whether a 0, 1 matrix is perfect. For linear matrices (a matrix is linear if it does not contain a 2×2 submatrix of all ones), this problem has been solved by Fonlupt and Zemirline [1981] and Conforti and Rao [1993].

15.5.2 Matroids

Matroids and submodular functions have been studied extensively, especially from the point of view of combinatorial optimization (see, for instance, Nemhauser and Wolsey [1988]). Matroids have nice properties that lead to efficient algorithms for the associated optimization problems. One of the interesting examples of a matroid is the problem of finding a maximum or minimum weight spanning tree in a graph. Two different but equivalent definitions of a matroid are given first. A greedy algorithm to solve a linear optimization problem over a matroid is presented. The matroid intersection problem is then discussed briefly.

Definition 15.9 Let $N = \{1, 2, \cdot, n\}$ be a finite set and let $\mathcal{F}$ be a set of subsets of N. Then $I = (N, \mathcal{F})$ is an independence system if $S_1 \in \mathcal{F}$ implies that $S_2 \in \mathcal{F}$ for all $S_2 \subseteq S_1$. Elements of $\mathcal{F}$ are called independent sets. A set $S \in \mathcal{F}$ is a maximal independent set if $S \cup \{j\} \notin \mathcal{F}$ for all $j \in N \backslash S$. A maximal independent set T is a maximum if $|T| \geq |S|$ for all $S \in \mathcal{F}$.

The rank $r(Y)$ of a subset $Y \subseteq N$ is the cardinality of the maximum independent subset $X \subseteq Y$. Note that $r(\phi) = 0$, $r(X) \leq |X|$ for $X \subseteq N$ and the rank function is nondecreasing, i.e., $r(X) \leq r(Y)$ for $X \subseteq Y \subseteq N$.

A matroid $M = (N, \mathcal{F})$ is an independence system in which every maximal independent set is a maximum.

Example 15.2

Let $G = (V, E)$ be an undirected connected graph with V as the node set and E as the edge set.

1. Let $I = (E, \mathcal{F})$ where $F \in \mathcal{F}$ if $F \subseteq E$ is such that at most one edge in F is incident to each node of V, that is, $F \in \mathcal{F}$ if F is a matching in G. Then $I = (E, \mathcal{F})$ is an independence system but not a matroid.
2. Let $M = (E, \mathcal{F})$ where $F \in \mathcal{F}$ if $F \subseteq E$ is such that $G_F = (V, F)$ is a forest, that is, G_F contains no cycles. Then $M = (E, \mathcal{F})$ is a matroid and maximal independent sets of M are spanning trees.

An alternative but equivalent definition of matroids is in terms of submodular functions.

Definition 15.10 A nondecreasing integer valued submodular function r defined on the subsets of N is called a matroid rank function if $r(\phi) = 0$ and $r(\{j\}) \leq 1$ for $j \in N$. The pair (N, r) is called a matroid.

A nondecreasing, integer-valued, submodular function f, defined on the subsets of N is called a polymatroid function if $f(\phi) = 0$. The pair (N, r) is called a polymatroid.

15.5.2.1 Matroid Optimization

To decide whether an optimization problem over a matroid is polynomially solvable or not, we need to first address the issue of representation of a matroid. If the matroid is given either by listing the independent sets or by its rank function, many of the associated linear optimization problems are trivial to solve. However, matroids associated with graphs are completely described by the graph and the condition for independence. For instance, the matroid in which the maximal independent sets are spanning forests, the graph $G = (V, E)$ and the independence condition of no cycles describes the matroid.

Most of the algorithms for matroid optimization problems require a test to determine whether a specified subset is independent. We assume the existence of an oracle or subroutine to do this checking in running time, which is a polynomial function of $|N| = n$.

Maximum Weight Independent Set. Given a matroid $M = (N, \mathcal{F})$ and weights w_j for $j \in N$, the problem of finding a maximum weight independent set is $\max_{F \in \mathcal{F}} \left\{ \sum_{j \in F} w_j \right\}$. The greedy algorithm to solve this problem is as follows:

Procedure 15.3 Greedy:

0. **Initialize:** Order the elements of N so that $w_i \geq w_{i+1},\ i = 1, 2, \ldots, n-1$. Let $T = \phi, i = 1$.
1. **If** $w_i \leq 0$ or $i > n$, **stop** T is optimal, i.e., $x_j = 1$ for $j \in T$ and $x_j = 0$ for $j \notin T$. If $w_i > 0$ and $T \cup \{i\} \in \mathcal{F}$, add element i to T.
2. **Increment** i by 1 and return to step 1.

Edmonds [1970, 1971] derived a complete description of the *matroid polytope*, the convex hull of the characteristic vectors of independent sets of a matroid. While this description has a large (exponential) number of constraints, it permits the treatment of linear optimization problems on independent sets of matroids as linear programs. Cunningham [1984] describes a polynomial algorithm to solve the separation problem for the matroid polytope. The matroid polytope and the associated greedy algorithm have been extended to polymatroids (Edmonds [1970], McDiarmid [1975]).

The separation problem for a polymatroid is equivalent to the problem of minimizing a submodular function defined over the subsets of N (see Nemhauser and Wolsey [1988]). A class of submodular functions that have some additional properties can be minimized in polynomial time by solving a maximum flow problem [Rhys 1970, Picard and Ratliff 1975]. The general submodular function can be minimized in polynomial time by the ellipsoid algorithm [Grötschel et al. 1988].

The uncapacitated plant location problem formulated in Section 15.4 can be reduced to maximizing a submodular function. Hence, it follows that maximizing a submodular function is $\mathcal{NP}$-hard.

15.5.2.2 Matroid Intersection

A matroid intersection problem involves finding an independent set contained in two or more matroids defined on the same set of elements.

Let $G = (V_1, V_2, E)$ be a bipartite graph. Let $M_i = (E, \mathcal{F}_i), i = 1, 2$, where $F \in \mathcal{F}_i$ if $F \subseteq E$ is such that no more than one edge of F is incident to each node in V_i. The set of matchings in G constitutes the intersection of the two matroids $M_i, i = 1, 2$. The problem of finding a maximum weight independent set in the intersection of two matroids can be solved in polynomial time [Lawler 1975, Edmonds 1970, 1979, Frank 1981]. The two (poly) matroid intersection polytope has been studied by Edmonds [1979].

The problem of testing whether a graph contains a Hamiltonian path is $\mathcal{NP}$-complete. Since this problem can be reduced to the problem of finding a maximum cordinality independent set in the intersection of three matroids, it follows that the matroid intersection problem involving three or more matroids is $\mathcal{NP}$-hard.

15.5.3 Valid Inequalities, Facets, and Cutting Plane Methods

Earlier in this section, we were concerned with conditions under which the packing and covering polytopes are integral. But, in general, these polytopes are not integral, and additional inequalities are required to have a complete linear description of the convex hull of integer solutions. The existence of finitely many such linear inequalities is guaranteed by Weyl's [1935] Theorem.

Consider the feasible region of an ILP given by

$$P_I = \{\mathbf{x} : A\mathbf{x} \leq \mathbf{b}; \mathbf{x} \geq \mathbf{0} \text{ and integer}\} \tag{15.8}$$

Recall that an inequality $\mathbf{f}\mathbf{x} \leq f_0$ is referred to as a valid inequality for P_I if $\mathbf{f}\mathbf{x}^* \leq f_0$ for all $\mathbf{x}^* \in P_I$. A valid linear inequality for $P_I(A, \mathbf{b})$ is said to be facet defining if it intersects $P_I(A, \mathbf{b})$ in a face of dimension one less than the dimension of $P_I(A, \mathbf{b})$. In the example shown in Figure 15.2, the inequality $\mathbf{x}_2 + \mathbf{x}_3 \leq 1$ is a facet defining inequality of the integer hull.

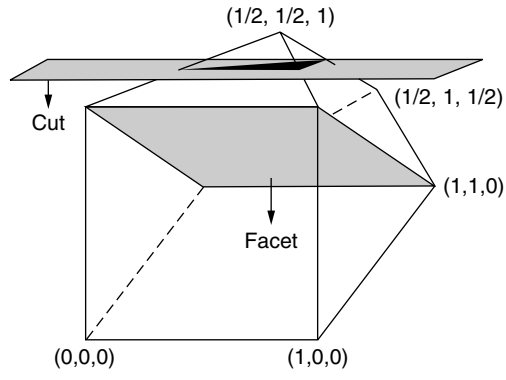


FIGURE 15.2 Relaxation, cuts, and facets (From Chandru, V. and Rao, M. R. Combinatorial optimization: an integer programming perspective. *ACM Comput. Surveys,* 28, 1. March 1996.)

Let $\mathbf{u} \geq 0$ be a row vector of appropriate size. Clearly $\mathbf{u}A\mathbf{x} \leq \mathbf{ub}$ holds for every $\mathbf{x}$ in P_I. Let $(\mathbf{u}A)_j$ denote the jth component of the row vector $\mathbf{u}A$ and $\lfloor(\mathbf{u}A)_j\rfloor$ denote the largest integer less than or equal to $(\mathbf{u}A)_j$. Now, since $\mathbf{x} \in P_I$ is a vector of nonnegative integers, it follows that $\sum_j \lfloor(\mathbf{u}A)_j\rfloor \mathbf{x}_j \leq \lfloor \mathbf{ub} \rfloor$ is a valid inequality for P_I. This scheme can be used to generate many valid inequalities by using different $\mathbf{u} \geq \mathbf{0}$. Any set of generated valid inequalities may be added to the constraints in Equation 15.7 and the process of generating them may be repeated with the enhanced set of inequalities. This iterative procedure of generating valid inequalities is called Gomory–Chvátal (GC) rounding. It is remarkable that this simple scheme is complete, i.e., every valid inequality of P_I can be generated by finite application of GC rounding (Chvátal [1973], Schrijver [1986]).

The number of inequalities needed to describe the convex hull of P_I is usually exponential in the size of A. But to solve an optimization problem on P_I, one is only interested in obtaining a partial description of P_I that facilitates the identification of an integer solution and prove its optimality. This is the underlying basis of any cutting plane approach to combinatorial problems.

15.5.3.1 The Cutting Plane Method

Consider the optimization problem

$$\max\{\mathbf{cx} : \mathbf{x} \in P_I = \{\mathbf{x} : A\mathbf{x} \leq \mathbf{b};\ \mathbf{x} \geq \mathbf{0} \text{ and integer}\}\}$$

The generic **cutting plane** method as applied to this formulation is given as follows.

Procedure 15.4 Cutting Plane:

1. Initialize $A' \leftarrow A$ and $\mathbf{b}' \leftarrow \mathbf{b}$.
2. Find an optimal solution $\bar{\mathbf{x}}$ to the linear program

$$\max\{\mathbf{cx} : A'\mathbf{x} \leq \mathbf{b}';\ \mathbf{x} \geq \mathbf{0}\}$$

   If $\bar{\mathbf{x}} \in P_I$, stop and return $\bar{\mathbf{x}}$.
3. Generate a valid inequality $\mathbf{fx} \leq f_0$ for P_I such that $\mathbf{f}\bar{\mathbf{x}} > f_0$ (the inequality "cuts" $\bar{\mathbf{x}}$).
4. Add the inequality to the constraint system, update

$$A' \leftarrow \begin{pmatrix} A' \\ \mathbf{f} \end{pmatrix}, \qquad \mathbf{b}' \leftarrow \begin{pmatrix} \mathbf{b}' \\ f_0 \end{pmatrix}$$

   Go to step 2.

In step 3 of the cutting plane method, we require a suitable application of the GC rounding scheme (or some alternative method of identifying a cutting plane). Notice that while the GC rounding scheme will generate valid inequalities, the identification of one that cuts off the current solution to the linear programming relaxation is all that is needed. Gomory [1958] provided just such a specialization of the rounding scheme that generates a cutting plane. Although this met the theoretical challenge of designing a sound and complete cutting plane method for integer linear programming, it turned out to be a weak method in practice. Successful cutting plane methods, in use today, use considerable additional insights into the structure of facet-defining cutting planes. Using facet cuts makes a huge difference in the speed of convergence of these methods. Also, the idea of combining cutting plane methods with search methods has been found to have a lot of merit. These branch and cut methods will be discussed in the next section.

15.5.3.2 The *b*-Matching Problem

Consider the **b**-matching problem:

$$\max\{\mathbf{cx} : A\mathbf{x} \leq \mathbf{b},\ \mathbf{x} \geq \mathbf{0} \text{ and integer}\} \tag{15.9}$$

where A is the node-edge incidence matrix of an undirected graph and $\mathbf{b}$ is a vector of positive integers. Let G be the undirected graph whose node-edge incidence matrix is given by A and let $W \subseteq V$ be any subset of nodes of G (i.e., subset of rows of A) such that

$$\mathbf{b}(W) = \sum_{i \in W} \mathbf{b}_i$$

is odd. Then the inequality

$$\mathbf{x}(W) = \sum_{e \in E(W)} \mathbf{x}_e \leq \frac{1}{2}(\mathbf{b}(W) - 1) \tag{15.10}$$

is a valid inequality for integer solutions to Equation 15.8 where $E(W) \subseteq E$ is the set of edges of G having both ends in W. Edmonds [1965] has shown that the inequalities Equation 15.8 and Equation 15.9 define the integral **b**-matching polytope. Note that the number of inequalities Equation 15.9 is exponential in the number of nodes of G. An instance of the successful application of the idea of using only a partial description of P_I is in the blossom algorithm for the matching problem, due to Edmonds [1965].

As we saw, an implication of the ellipsoid method for linear programming is that the linear program over P_I can be solved in polynomial time if and only if the associated separation problem (also referred to as the constraint identification problem, see Section 15.2) can be solved in polynomial time, see Grötschel et al. [1982], Karp and Papadimitriou [1982], and Padberg and Rao [1981]. The separation problem for the **b**-matching problem with or without upper bounds was shown by Padberg and Rao [1982], to be solvable in polynomial time. The procedure involves a minor modification of the algorithm of Gomory and Hu [1961] for multiterminal networks. However, no polynomial (in the number of nodes of the graph) linear programming formulation of this separation problem is known. A related unresolved issue is whether there exists a polynomial size (compact) formulation for the **b**-matching problem. Yannakakis [1988] has shown that, under a symmetry assumption, such a formulation is impossible.

15.5.3.3 Other Combinatorial Problems

Besides the matching problem, several other combinatorial problems and their associated polytopes have been well studied and some families of facet defining inequalities have been identified. For instance, the set packing, graph partitioning, plant location, max cut, traveling salesman, and Steiner tree problems have been extensively studied from a polyhedral point of view (see, for instance, Nemhauser and Wolsey [1988]).

These combinatorial problems belong to the class of $\mathcal{NP}$-complete problems. In terms of a worst-case analysis, no polynomial-time algorithms are known for these problems. Nevertheless, using a cutting plane approach with branch and bound or branch and cut (see Section 15.6), large instances of these problems

have been successfully solved, see Crowder et al. [1983], for general $0-1$ problems, Barahona et al. [1989] for the max cut problem, Padberg and Rinaldi [1991] for the traveling salesman problem, and Chopra et al. [1992] for the Steiner tree problem.

15.6 Partial Enumeration Methods

In many instances, to find an optimal solution to integer linear programing problems (ILP), the structure of the problem is exploited together with some sort of partial enumeration. In this section, we review the branch and bound (B-and-B) and branch and cut (B-and-C) methods for solving an ILP.

15.6.1 Branch and Bound

The branch bound (B-and-B) method is a systematic scheme for implicitly enumerating the finitely many feasible solutions to an ILP. Although, theoretically the size of the enumeration tree is exponential in the problem parameters, in most cases, the method eliminates a large number of feasible solutions. The key features of branch and bound method are:

1. **Selection/removal** of one or more problems from a candidate list of problems
2. **Relaxation** of the selected problem so as to obtain a lower bound (on a minimization problem) on the optimal objective function value for the selected problem
3. **Fathoming**, if possible, of the selected problem
4. **Branching** strategy is needed if the selected problem is not fathomed. Branching creates subproblems, which are added to the candidate list of problems.

The four steps are repeated until the candidate list is empty. The B-and-B method sequentially examines problems that are added and removed from a candidate list of problems.

15.6.1.1 Initialization

Initially, the candidate list contains only the original ILP, which is denoted as

$$(P) \quad \min\{\mathbf{cx} : A\mathbf{x} \leq \mathbf{b},\ \mathbf{x} \geq \mathbf{0} \text{ and integer}\}$$

Let $F(P)$ denote the feasible region of (P) and $z(P)$ denote the optimal objective function value of (P). For any $\bar{\mathbf{x}}$ in $F(P)$, let $z_P(\bar{\mathbf{x}}) = \mathbf{c}\bar{\mathbf{x}}$.

Frequently, heuristic procedures are first applied to get a good feasible solution to (P). The best solution known for (P) is referred to as the current incumbent solution. The corresponding objective function value is denoted as z_I. In most instances, the initial heuristic solution is neither optimal nor at least immediately certified to be optimal. Thus, further analysis is required to ensure that an optimal solution to (P) is obtained. If no feasible solution to (P) is known, z_I is set to ∞.

15.6.1.2 Selection/Removal

In each iterative step of B-and-B, a problem is selected and removed from the candidate list for further analysis. The selected problem is henceforth referred to as the candidate problem (CP). The algorithm terminates if there is no problem to select from the candidate list. Initially, there is no issue of selection since the candidate list contains only the problem (P). However, as the algorithm proceeds, there would be many problems on the candidate list and a selection rule is required. Appropriate selection rules, also referred to as branching strategies, are discussed later. Conceptually, several problems may be simultaneously selected and removed from the candidate list. However, most sequential implementations of B-and-B select only one problem from the candidate list and this is assumed henceforth. Parallel aspects of B-and-B on $0-1$ integer linear programs are discussed in Cannon and Hoffman [1990] and for the case of traveling salesman problems in Applegate et al. [1994].

The computational time required for the B-and-B algorithm depends crucially on the order in which the problems in the candidate list are examined. A number of clever heuristic rules may be employed in devising such strategies. Two general purpose selection strategies that are commonly used are as follows:

1. Choose the problem that was added last to the candidate list. This last-in–first-out rule (LIFO) is also called depth first search (DFS) since the selected candidate problem increases the depth of the active enumeration tree.
2. Choose the problem on the candidate list that has the least lower bound. Ties may be broken by choosing the problem that was added last to the candidate list. This rule would require that a lower bound be obtained for each of the problems on the candidate list. In other words, when a problem is added to the candidate list, an associated lower bound should also be stored. This may be accomplished by using ad hoc rules or by solving a relaxation of each problem before it is added to the candidate list.

Rule 1 is known to empirically dominate rule 2 when storage requirements for candidate list and computation time to solve (P) are taken into account. However, some analysis indicates that rule 2 can be shown to be superior if minimizing the number of candidate problems to be solved is the criterion (see Parker and Rardin [1988]).

15.6.1.3 Relaxation

In order to analyze the selected candidate problem (CP), a **relaxation** (CP_R) of (CP) is solved to obtain a lower bound $z(CP_R) \leq z(CP)$. (CP_R) is a relaxation of (CP) if:

1. $F(CP) \subseteq F(CP_R)$
2. For $\bar{\mathbf{x}} \in F(CP)$, $z_{CP_R}(\bar{\mathbf{x}}) \leq z_{CP}(\bar{\mathbf{x}})$
3. For $\bar{\mathbf{x}}, \hat{\mathbf{x}} \in F(CP)$, $z_{CP_R}(\bar{\mathbf{x}}) \leq z_{CP_R}(\hat{\mathbf{x}})$ implies that $z_{CP}(\bar{\mathbf{x}}) \leq z_{CP}(\hat{\mathbf{x}})$

Relaxations are needed because the candidate problems are typically hard to solve. The relaxations used most often are either linear programming or Lagrangian relaxations of (CP), see Section 15.7 for details. Sometimes, instead of solving a relaxation of (CP), a lower bound is obtained by using some ad hoc rules such as penalty functions.

15.6.1.4 Fathoming

A candidate problem is fathomed if:

(FC1) analysis of (CP_R) reveals that (CP) is infeasible. For instance, if $F(CP_R) = \phi$, then $F(CP) = \phi$.
(FC2) analysis of (CP_R) reveals that (CP) has no feasible solution better than the current incumbent solution. For instance, if $z(CP_R) \geq z_I$, then $z(CP) \geq z(CP_R) \geq z_I$.
(FC3) analysis of (CP_R) reveals an optimal solution of (CP). For instance, if the optimal solution, $\mathbf{x}_R$, to (CP_R) is feasible in (CP), then $(\mathbf{x}_R)$ is an optimal solution to (CP) and $z(CP) = \mathbf{c}\mathbf{x}_R$.
(FC4) analysis of (CP_R) reveals that (CP) is dominated by some other problem, say, CP^*, in the candidate list. For instance, if it can shown that $z(CP^*) \leq z(CP)$, then there is no need to analyze (CP) further.

If a candidate problem (CP) is fathomed using any of the preceding criteria, then further examination of (CP) or its descendants (subproblems) obtained by separation is not required. If (FC3) holds, and $z(CP) < z_I$, the incumbent is updated as $\mathbf{x}_R$ and z_I is updated as $z(CP)$.

15.6.1.5 Separation/Branching

If the candidate problem (CP) is not fathomed, then CP is separated into several problems, say, (CP_1), $(CP_2), \ldots, (CP_q)$, where $\bigcup_{t=1}^{q} F(CP_t) = F(CP)$ and, typically,

$$F(CP_i) \cap F(CP_j) = \phi \; \forall \, i \neq j$$

For instance, a separation of (CP) into $(CP_i), i = 1, 2, \ldots, q$, is obtained by fixing a single variable, say, $\mathbf{x}_j$, to one of the q possible values of $\mathbf{x}_j$ in an optimal solution to (CP). The choice of the variable

to fix depends on the separation strategy, which is also part of the branching strategy. After separation, the subproblems are added to the candidate list. Each subproblem (CP_t) is a restriction of (CP) since $F(CP_t) \subseteq F(CP)$. Consequently, $z(CP) \leq z(CP_t)$ and $z(CP) = \min_t z(CP_t)$.

The various steps in the B-and-B algorithm are outlined as follows.

Procedure 15.5 B-and-B:

0. **Initialize:** Given the problem (P), the incumbent value z_I is obtained by applying some heuristic (if a feasible solution to (P) is not available, set $z_I = +\infty$). Initialize the candidate list $C \leftarrow \{(P)\}$.
1. **Optimality:** If $C = \emptyset$ and $z_I = +\infty$, then (P) is infeasible, stop. Stop also if $C = \emptyset$ and $z_I < +\infty$, the incumbent is an optimal solution to (P).
2. **Selection:** Using some candidate selection rule, select and remove a problem $(CP) \in C$.
3. **Bound:** Obtain a lower bound for (CP) by either solving a relaxation (CP_R) of (CP) or by applying some ad-hoc rules. If (CP_R) is infeasible, return to Step 1. Else, let x_R be an optimal solution of (CP_R).
4. **Fathom:** If $z(CP_R) \geq z_I$, return to step 1. Else if x_R is feasible in (CP) and $z(CP) < z_I$, set $z_I \leftarrow z(CP)$, update the incumbent as x_R and return to step 1. Finally, if x_R is feasible in (CP) but $z(CP) \geq z_I$, return to step 1.
5. **Separation:** Using some separation or branching rule, separate (CP) into $(CP_i), i = 1, 2, \ldots, q$ and set $C \leftarrow C \cup \{CP_1), (CP_2), \ldots, (CP_q)\}$ and return to step 1.
6. **End Procedure.**

Although the B-and-B method is easy to understand, the implementation of this scheme for a particular ILP is a nontrivial task requiring the following:

1. A relaxation strategy with efficient procedures for solving these relaxations
2. Efficient data-structures for handling the rather complicated bookkeeping of the candidate list
3. Clever strategies for selecting promising candidate problems
4. Separation or branching strategies that could effectively prune the enumeration tree

A key problem is that of devising a relaxation strategy, that is, to find *good relaxations*, which are significantly easier to solve than the original problems and tend to give sharp lower bounds. Since these two are conflicting, one has to find a reasonable tradeoff.

15.6.2 Branch and Cut

In the past few years, the branch and cut (B-and-C) method has become popular for solving NP-complete combinatorial optimization problems. As the name suggests, the B-and-C method incorporates the features of both the branch and bound method just presented and the cutting plane method presented previously. The main difference between the B-and-C method and the general B-and-B scheme is in the bound step (step 3).

A distinguishing feature of the B-and-C method is that the relaxation (CP_R) of the candidate problem (CP) is a linear programming problem, and, instead of merely solving (CP_R), an attempt is made to solve (CP) by using cutting planes to tighten the relaxation. If (CP_R) contains inequalities that are valid for (CP) but not for the given ILP, then the GC rounding procedure may generate inequalities that are valid for (CP) but not for the ILP. In the B-and-C method, the inequalities that are generated are always valid for the ILP and hence can be used globally in the enumeration tree.

Another feature of the B-and-C method is that often heuristic methods are used to convert some of the fractional solutions, encountered during the cutting plane phase, into feasible solutions of the (CP) or more generally of the given ILP. Such feasible solutions naturally provide upper bounds for the ILP. Some of these upper bounds may be better than the previously identified best upper bound and, if so, the current incumbent is updated accordingly.

We thus obtain the B-and-C method by replacing the bound step (step 3) of the B-and-B method by steps 3(a) and 3(b) and also by replacing the fathom step (step 4) by steps 4(a) and 4(b) given subsequently.

3(a) **Bound:** Let (CP_R) be the LP relaxation of (CP). Attempt to solve (CP) by a cutting plane method which generates valid inequalities for (P). Update the constraint system of (P) and the incumbent as appropriate.

Let $F\mathbf{x} \leq \mathbf{f}$ denote all of the valid inequalities generated during this phase. Update the constraint system of (P) to include all of the generated inequalities, i.e., set $A^T \leftarrow (A^T, F^T)$ and $\mathbf{b}^T \leftarrow (\mathbf{b}^T, \mathbf{f}^T)$. The constraints for all of the problems in the candidate list are also to be updated.

During the cutting plane phase, apply heuristic methods to convert some of the identified fractional solutions into feasible solutions to (P). If a feasible solution, $\bar{\mathbf{x}}$, to (P), is obtained such that $\mathbf{c}\bar{\mathbf{x}} < z_I$, update the incumbent to $\bar{\mathbf{x}}$ and z_I to $\mathbf{c}\bar{\mathbf{x}}$. Hence, the remaining changes to B-and-B are as follows:

3(b) **If** (CP) is solved go to step 4(a). **Else**, let $\hat{\mathbf{x}}$ be the solution obtained when the cutting plane phase is terminated, (we are unable to identify a valid inequality of (P) that is violated by $\hat{\mathbf{x}}$). Go to step 4(b).

4(a) **Fathom by Optimality:** Let $\mathbf{x}^*$ be an optimal solution to (CP). If $z(CP) < z_I$, set $\mathbf{x}_I \leftarrow z(CP)$ and update the incumbent as $\mathbf{x}^*$. Return to step 1.

4(b) **Fathom by Bound:** If $\mathbf{c}\hat{\mathbf{x}} \geq z_I$, return to Step 1.
Else go to step 5.

The incorporation of a cutting plane phase into the B-and-B scheme involves several technicalities which require careful design and implementation of the B-and-C algorithm. Details of the state of the art in cutting plane algorithms including the B-and-C algorithm are reviewed in Jünger et al. [1995].

15.7 Approximation in Combinatorial Optimization

The inherent complexity of integer linear programming has led to a long-standing research program in approximation methods for these problems. Linear programming relaxation and Lagrangian relaxation are two general approximation schemes that have been the real workhorses of computational practice. Semidefinite relaxation is a recent entrant that appears to be very promising. In this section, we present a brief review of these developments in the approximation of combinatorial optimization problems.

In the past few years, there has been significant progress in our understanding of performance guarantees for approximation of $\mathcal{NP}$-hard combinatorial optimization problems. A **ρ-approximate** algorithm for an optimization problem is an approximation algorithm that delivers a feasible solution with objective value within a factor of ρ of optimal (think of minimization problems and $\rho \geq 1$). For some combinatorial optimization problems, it is possible to *efficiently* find solutions that are arbitrarily close to optimal even though finding the true optimal is hard. If this were true of most of the problems of interest, we would be in good shape. However, the recent results of Arora et al. [1992] indicate exactly the opposite conclusion.

A polynomial-time approximation scheme (PTAS) for an optimization problem is a family of algorithms, A_ρ, such that for each $\rho > 1$, A_ρ is a polynomial-time ρ-approximate algorithm. Despite concentrated effort spanning about two decades, the situation in the early 1990s was that for many combinatorial optimization problems, we had no PTAS and no evidence to suggest the nonexistence of such schemes either. This led Papadimitriou and Yannakakis [1991] to define a new complexity class (using reductions that preserve approximate solutions) called MAXSNP, and they identified several complete languages in this class. The work of Arora et al. [1992] completed this agenda by showing that, assuming $\mathcal{P} \neq \mathcal{NP}$, there is no PTAS for a MAXSNP-complete problem.

An implication of these theoretical developments is that for most combinatorial optimization problems, we have to be quite satisfied with performance guarantee factors ρ that are of some small fixed value. (There are problems, like the general traveling salesman problem, for which there are no ρ-approximate algorithms

for any finite value of ρ, assuming of course that $\mathcal{P} \neq \mathcal{NP}$.) Thus, one avenue of research is to go problem by problem and knock ρ down to its smallest possible value. A different approach would be to look for other notions of good approximations based on probabilistic guarantees or empirical validation. Let us see how the polyhedral combinatorics perspective helps in each of these directions.

15.7.1 LP Relaxation and Randomized Rounding

Consider the well-known problem of finding the *smallest weight vertex cover* in a graph. We are given a graph $G(V, E)$ and a nonnegative weight $\mathbf{w}(v)$ for each vertex $v \in V$. We want to find the smallest total weight subset of vertices S such that each edge of G has at least one end in S. (This problem is known to be MAXSNP-hard.) An integer programming formulation of this problem is given by

$$\min \left\{ \sum_{v \in V} \mathbf{w}(v)\mathbf{x}(v) : \mathbf{x}(u) + \mathbf{x}(v) \geq 1,\ \forall (u, v) \in E,\ \ \mathbf{x}(v) \in \{0, 1\}\ \forall v \in V \right\}$$

To obtain the linear programming relaxation we substitute the $\mathbf{x}(v) \in \{0, 1\}$ constraint with $\mathbf{x}(v) \geq 0$ for each $v \in V$. Let $\mathbf{x}^*$ denote an optimal solution to this relaxation. Now let us round the fractional parts of $\mathbf{x}^*$ in the usual way, that is, values of 0.5 and up are rounded to 1 and smaller values down to 0. Let $\hat{\mathbf{x}}$ be the 0–1 solution obtained. First note that $\hat{\mathbf{x}}(v) \leq 2\mathbf{x}^*(v)$ for each $v \in V$. Also, for each $(u, v) \in E$, since $\mathbf{x}^*(u) + \mathbf{x}^*(v) \geq 1$, at least one of $\hat{\mathbf{x}}(u)$ and $\hat{\mathbf{x}}(v)$ must be set to 1. Hence $\hat{\mathbf{x}}$ is the incidence vector of a vertex cover of G whose total weight is within twice the total weight of the linear programming relaxation (which is a lower bound on the weight of the optimal vertex cover). Thus, we have a 2-approximate algorithm for this problem, which solves a linear programming relaxation and uses rounding to obtain a feasible solution.

The deterministic rounding of the fractional solution worked quite well for the vertex cover problem. One gets a lot more power from this approach by adding in randomization to the rounding step. Raghavan and Thompson [1987] proposed the following obvious randomized rounding scheme. Given a $0-1$ integer program, solve its linear programming relaxation to obtain an optimal $\mathbf{x}^*$. Treat the $\mathbf{x}_j{}^* \in [0, 1]$ as probabilities, i.e., let probability $\{\mathbf{x}_j = 1\} = \mathbf{x}_j{}^*$, to randomly round the fractional solution to a $0 - 1$ solution. Using Chernoff bounds on the tails of the binomial distribution, Raghavan and Thompson [1987] were able to show, for specific problems, that with high probability, this scheme produces integer solutions which are close to optimal. In certain problems, this rounding method may not always produce a feasible solution. In such cases, the expected values have to be computed as conditioned on feasible solutions produced by rounding. More complex (nonlinear) randomized rounding schemes have been recently studied and have been found to be extremely effective. We will see an example of nonlinear rounding in the context of semidefinite relaxations of the max-cut problem in the following.

15.7.2 Primal–Dual Approximation

The linear programming relaxation of the vertex cover problem, as we saw previously, is given by

$$(P_{VC}) \quad \min \left\{ \sum_{v \in V} \mathbf{w}(v)\mathbf{x}(v) : \mathbf{x}(u) + \mathbf{x}(v) \geq 1,\ \forall (u, v) \in E,\ \ \mathbf{x}(v) \geq 0\ \forall v \in V \right\}$$

and its dual is

$$(D_{VC}) \quad \max \left\{ \sum_{(u,v) \in E} \mathbf{y}(u, v) : \sum_{(u,v) \in E} \mathbf{y}(u, v) \leq w(v),\ \forall v \in V,\ \ \mathbf{y}(u, v) \geq 0\ \forall (u, v) \in E \right\}$$

The primal–dual approximation approach would first obtain an optimal solution $\mathbf{y}^*$ to the dual problem (D_{VC}). Let $\hat{V} \subseteq V$ denote the set of vertices for which the dual constraints are tight, i.e.,

$$\hat{V} = \left\{ v \in V : \sum_{(u,v)\in E} \mathbf{y}^*(u,v) = \mathbf{w}(v) \right\}$$

The approximate vertex cover is taken to be $\hat{V}$. It follows from complementary slackness that $\hat{V}$ is a vertex cover. Using the fact that each edge (u, v) is in the star of at most two vertices (u and v), it also follows that $\hat{V}$ is a 2-approximate solution to the minimum weight vertex cover problem.

In general, the primal–dual approximation strategy is to use a dual solution to the linear programming relaxation, along with complementary slackness conditions as a heuristic to generate an integer (primal) feasible solution, which for many problems turns out to be a good approximation of the optimal solution to the original integer program.

Remark 15.10 A recent survey of primal-dual approximation algorithms and some related interesting results are presented in Williamson [2000].

15.7.3 Semidefinite Relaxation and Rounding

The idea of using semidefinite programming to solve combinatorial optimization problems appears to have originated in the work of Lovász [1979] on the Shannon capacity of graphs. Grötschel et al. [1988] later used the same technique to compute a maximum stable set of vertices in perfect graphs via the ellipsoid method. Lovasz and Schrijver [1991] resurrected the technique to present a fascinating theory of semidefinite relaxations for general 0–1 integer linear programs. We will not present the full-blown theory here but instead will present a lovely application of this methodology to the problem of finding the maximum weight cut of a graph. This application of semidefinite relaxation for approximating MAXCUT is due to Goemans and Williamson [1994].

We begin with a quadratic Boolean formulation of MAXCUT

$$\max \left\{ \frac{1}{2} \sum_{(u,v)\in E} \mathbf{w}(u,v)(1 - \mathbf{x}(u)\mathbf{x}(v)) : \mathbf{x}(v) \in \{-1, 1\} \; \forall \; v \in V \right\}$$

where $G(V, E)$ is the graph and $\mathbf{w}(u, v)$ is the nonnegative weight on edge (u, v). Any $\{-1, 1\}$ vector of $\mathbf{x}$ values provides a bipartition of the vertex set of G. The expression $(1 - \mathbf{x}(u)\mathbf{x}(v))$ evaluates to 0 if u and v are on the same side of the bipartition and to 2 otherwise. Thus, the optimization problem does indeed represent exactly the MAXCUT problem.

Next we reformulate the problem in the following way:

- We square the number of variables by substituting each $\mathbf{x}(v)$ with $\boldsymbol{\chi}(v)$ an n-vector of variables (where n is the number of vertices of the graph).
- The quadratic term $\mathbf{x}(u)\mathbf{x}(v)$ is replaced by $\boldsymbol{\chi}(u) \cdot \boldsymbol{\chi}(v)$, which is the inner product of the vectors.
- Instead of the $\{-1, 1\}$ restriction on the $\mathbf{x}(v)$, we use the Euclidean normalization $\|\boldsymbol{\chi}(v)\| = 1$ on the $\boldsymbol{\chi}(v)$.

Thus, we now have a problem

$$\max \left\{ \frac{1}{2} \sum_{(u,v)\in E} \mathbf{w}(u,v)(1 - \boldsymbol{\chi}(u) \cdot \boldsymbol{\chi}(v)) : \|\boldsymbol{\chi}(v)\| = 1 \; \forall \; v \in V \right\}$$

which is a relaxation of the MAXCUT problem (note that if we force only the first component of the $\boldsymbol{\chi}(v)$ to have nonzero value, we would just have the old formulation as a special case).

The final step is in noting that this reformulation is nothing but a semidefinite program. To see this we introduce $n \times n$ Gram matrix Y of the unit vectors $\boldsymbol{\chi}(v)$. So $Y = X^T X$ where $X = (\boldsymbol{\chi}(v) : v \in V)$. Thus, the relaxation of MAXCUT can now be stated as a semidefinite program,

$$\max \left\{ \frac{1}{2} \sum_{(u,v)\in E} \mathbf{w}(u,v)(1 - Y_{(u,v)}) : Y \succeq 0, \quad Y_{(u,v)} = 1 \; \forall \, v \in V \right\}$$

Recall from Section 15.2 that we are able to solve such semidefinite programs to an additive error ϵ in time polynomial in the input length and $\log 1/\epsilon$ by using either the ellipsoid method or interior point methods.

Let $\boldsymbol{\chi}^*$ denote the near optimal solution to the semidefinite programming relaxation of MAXCUT (convince yourself that $\boldsymbol{\chi}^*$ can be reconstructed from an optimal Y^* solution). Now we encounter the final trick of Goemans and Williamson. The approximate maximum weight cut is extracted from $\boldsymbol{\chi}^*$ by randomized rounding. We simply pick a random hyperplane H passing through the origin. All of the $v \in V$ lying to one side of H get assigned to one side of the cut and the rest to the other. Goemans and Williamson observed the following inequality.

Lemma 15.1 *For $\boldsymbol{\chi}_1$ and $\boldsymbol{\chi}_2$, two random n-vectors of unit norm, let $\mathbf{x}(1)$ and $\mathbf{x}(2)$ be ± 1 values with opposing signs if H separates the two vectors and with same signs otherwise. Then $\tilde{E}(1 - \boldsymbol{\chi}_1 \cdot \boldsymbol{\chi}_2) \leq 1.1393 \cdot \tilde{E}(1 - \mathbf{x}(1)\mathbf{x}(2))$ where $\tilde{E}$ denotes the expected value.*

By linearity of expectation, the lemma implies that the expected value of the cut produced by the rounding is at least 0.878 times the expected value of the semidefinite program. Using standard conditional probability techniques for derandomizing, Goemans and Williamson show that a deterministic polynomial-time approximation algorithm with the same margin of approximation can be realized. Hence we have a cut with value at least 0.878 of the maximum cut value.

Remark 15.11 For semidefinite relaxations of mixed integer programs in which the integer variables are restricted to be 0 or 1, Iyengar and Cezik [2002] develop methods for generating Gomory–Chavatal and disjunctive cutting planes that extends the work of Balas et al. [1993]. Ye [2000] shows that strengthened semidefinite relaxations and mixed rounding methods achieve superior performance guarantee for some discrete optimization problems. A recent survey of semidefinite programming and applications is in Wolkowicz et al. [2000].

15.7.4 Neighborhood Search

A combinatorial optimization problem may be written succinctly as

$$\min\{ f(x) \; : \; x \in X \}$$

The traditional neighborhood method starts at a feasible point x_0 (in X), and iteratively proceeds to a neighborhood point that is better in terms of the objective function f until a specified termination condition is attained. While the concept of neighborhood $N(x)$ of a point x is well defined in calculus, the specification of $N(x)$ is itself a matter of consideration in combinatorial optimization. For instance, for the traveling salesman problem the so-called *k-opt heuristic* (see Lin and Kernighan [1973]) is a neighborhood search method which for a given tour considers "neighborhood tours" in which k variables (edges) in the given tour are replaced by k other variables such that a tour is maintained. This search technique has proved to be effective though it is quite complicated to implement when k is larger than 3.

A neighborhood search method leads to a local optimum in terms of the neighborhood chosen. Of course, the chosen neighborhood may be large enough to ensure a global optimum but such a procedure is typically not practical in terms of searching the neighborhood for a better solution. Recently Orlin [2000]

has presented very large-scale neighborhood search algorithms in which the neighborhood is searched using network flow or dynamic programming methods. Another method advocated by Orlin [2000] is to define the neighborhood in such a manner that the search process becomes a polynomially solvable special case of a hard combinatorial problem.

To avoid getting trapped at a local optimum solution, different strategies such as tabu search (see, for instance, Glover and Laguna [1997]), simulated annealing (see, for instance, Aarts and Korst [1989]), genetic algorithms (see, for instance, Whitley [1993]), and neural networks have been developed. Essentially these methods allow for the possibility of sometimes moving to an inferior solution in terms of the objective function or even to an infeasible solution. While there is no guarantee of obtaining a global optimal solution, computational experience in solving several difficult combinatorial optimization problems has been very encouraging. However, a drawback of these methods is that performance guarantees are not typically available.

15.7.5 Lagrangian Relaxation

We end our discussion of approximation methods for combinatorial optimization with the description of Lagrangian relaxation. This approach has been widely used for about two decades now in many practical applications. Lagrangian relaxation, like linear programming relaxation, provides bounds on the combinatorial optimization problem being relaxed (i.e., lower bounds for minimization problems).

Lagrangian relaxation has been so successful because of a couple of distinctive features. As was noted earlier, in many hard combinatorial optimization problems, we usually have embedded some nice tractable subproblems which have efficient algorithms. Lagrangian relaxation gives us a framework to *jerry-rig* an approximation scheme that uses these efficient algorithms for the subproblems as subroutines. A second observation is that it has been empirically observed that well-chosen Lagrangian relaxation strategies usually provide very tight bounds on the optimal objective value of integer programs. This is often used to great advantage within partial enumeration schemes to get very effective pruning tests for the search trees.

Practitioners also have found considerable success with designing heuristics for combinatorial optimization by starting with solutions from Lagrangian relaxations and constructing good feasible solutions via so-called *dual ascent* strategies. This may be thought of as the analogue of rounding strategies for linear programming relaxations (but with no performance guarantees, other than empirical ones).

Consider a representation of our combinatorial optimization problem in the form

$$(P) \quad z = \min\{\mathbf{cx} : A\mathbf{x} \geq \mathbf{b},\ \mathbf{x} \in X \subseteq \Re^n\}$$

Implicit in this representation is the assumption that the explicit constraints ($A\mathbf{x} \geq \mathbf{b}$) are *small* in number. For convenience, let us also assume that that X can be replaced by a finite list $\{\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^T\}$.

The following definitions are with respect to (P):

- Lagrangian. $L(\mathbf{u}, \mathbf{x}) = \mathbf{u}(A\mathbf{x} - \mathbf{b}) + \mathbf{cx}$ where $\mathbf{u}$ are the Lagrange multipliers.
- Lagrangian-dual function. $\mathcal{L}(\mathbf{u}) = \min_{\mathbf{x}\in X}\{L(\mathbf{u}, \mathbf{x})\}$.
- Lagrangian-dual problem. $(D) \quad d = \max_{\mathbf{u}\geq 0}\{\mathcal{L}(\mathbf{u})\}$.

It is easily shown that (D) satisfies a weak duality relationship with respect to (P), i.e., $z \geq d$. The discreteness of X also implies that $\mathcal{L}(\mathbf{u})$ is a piecewise linear and concave function (see Shapiro [1979]). In practice, the constraints X are chosen such that the evaluation of the Lagrangian dual function $\mathcal{L}(\mathbf{u})$ is easily made (i.e., the *Lagrangian subproblem* $\min_{\mathbf{x}\in X}\{L(\mathbf{u}, \mathbf{x})\}$ is easily solved for a fixed value of $\mathbf{u}$).

Example 15.3

Traveling salesman problem (TSP). For an undirected graph G, with costs on each edge, the TSP is to find a minimum cost set H of edges of G such that it forms a Hamiltonian cycle of the graph. H is a Hamiltonian cycle of G if it is a simple cycle that spans all the vertices of G. Alternatively, H must satisfy:

(1) exactly two edges of H are adjacent to each node, and (2) H forms a connected, spanning subgraph of G.

Held and Karp [1970] used these observations to formulate a Lagrangian relaxation approach for TSP that relaxes the degree constraints (1). Notice that the resulting subproblems are minimum spanning tree problems which can be easily solved.

The most commonly used general method of finding the optimal multipliers in Lagrangian relaxation is subgradient optimization (cf. Held et al. [1974]). Subgradient optimization is the non differentiable counterpart of steepest descent methods. Given a dual vector $\mathbf{u}^k$, the iterative rule for creating a sequence of solutions is given by:

$$\mathbf{u}^{k+1} = \mathbf{u}^k + t_k \boldsymbol{\gamma}(\mathbf{u}^k)$$

where t_k is an appropriately chosen step size, and $\boldsymbol{\gamma}(\mathbf{u}^k)$ is a subgradient of the dual function $\mathcal{L}$ at $\mathbf{u}^k$. Such a subgradient is easily generated by

$$\boldsymbol{\gamma}(\mathbf{u}^k) = A\mathbf{x}^k - \mathbf{b}$$

where $\mathbf{x}^k$ is a maximizer of $\min_{\mathbf{x}\in X}\{L(\mathbf{u}^k, \mathbf{x})\}$.

Subgradient optimization has proven effective in practice for a variety of problems. It is possible to choose the step sizes $\{t_k\}$ to guarantee convergence to the optimal solution. Unfortunately, the method is not finite, in that the optimal solution is attained only in the limit. Further, it is not a pure descent method. In practice, the method is heuristically terminated and the best solution in the generated sequence is recorded. In the context of nondifferentiable optimization, the ellipsoid algorithm was devised by Shor [1970] to overcome precisely some of these difficulties with the subgradient method.

The ellipsoid algorithm may be viewed as a scaled subgradient method in much the same way as variable metric methods may be viewed as scaled steepest descent methods (cf. Akgül [1984]). And if we use the ellipsoid method to solve the Lagrangian dual problem, we obtain the following as a consequence of the polynomial-time equivalence of optimization and separation.

Theorem 15.8 *The Lagrangian dual problem is polynomial-time solvable if and only if the Lagrangian subproblem is. Consequently, the Lagrangian dual problem is $\mathcal{NP}$-hard if and only if the Lagrangian subproblem is.*

The theorem suggests that, in practice, if we set up the Lagrangian relaxation so that the subproblem is tractable, then the search for optimal Lagrangian multipliers is also tractable.

15.8 Prospects in Integer Programming

The current emphasis in software design for integer programming is in the development of shells (for example, CPLEX 6.5 [1999], MINTO (Savelsbergh et al. [1994]), and OSL [1991]) wherein a general purpose solver like branch and cut is the driving engine. Problem-specific codes for generation of cuts and facets can be easily interfaced with the engine. Recent computational results (Bixby et al. [2001]) suggests that it is now possible to solve relatively large size integer programming problems using general purpose codes. We believe that this trend will eventually lead to the creation of general purpose problem solving languages for combinatorial optimization akin to AMPL (Fourer et al. [1993]) for linear and nonlinear programming.

A promising line of research is the development of an empirical science of algorithms for combinatorial optimization (Hooker [1993]). Computational testing has always been an important aspect of research on the efficiency of algorithms for integer programming. However, the standards of test designs and empirical analysis have not been uniformly applied. We believe that there will be important strides in this aspect of integer programming and more generally of algorithms. J. N. Hooker argues that it may be useful to

stop looking at algorithmics as purely a deductive science and start looking for advances through repeated application of "hypothesize and test" paradigms, i.e., through empirical science. Hooker and Vinay [1995] developed a science of selection rules for the Davis–Putnam–Loveland scheme of theorem proving in propositional logic by applying the empirical approach.

The integration of logic-based methodologies and mathematical programming approaches is evidenced in the recent emergence of constraint logic programming (CLP) systems (Saraswat and Van Hentenryck [1995], Borning [1994]) and logico-mathematical programming (Jeroslow [1989], Chandru and Hooker [1991]). In CLP, we see a structure of Prolog-like programming language in which some of the predicates are constraint predicates whose truth values are determined by the solvability of constraints in a wide range of algebraic and combinatorial settings. The solution scheme is simply a clever orchestration of constraint solvers in these various domains and the role of conductor is played by resolution. The clean semantics of logic programming is preserved in CLP. A bonus is that the output language is symbolic and expressive. An orthogonal approach to CLP is to use constraint methods to solve inference problems in logic. Imbeddings of logics in mixed integer programming sets were proposed by Williams [1987] and Jeroslow [1989]. Efficient algorithms have been developed for inference algorithms in many types and fragments of logic, ranging from Boolean to predicate to belief logics (Chandru and Hooker [1999]).

A persistent theme in the integer programming approach to combinatorial optimization, as we have seen, is that the representation (formulation) of the problem deeply affects the efficacy of the solution methodology. A proper choice of formulation can therefore make the difference between a successful solution of an optimization problem and the more common perception that the problem is insoluble and one must be satisfied with the best that heuristics can provide. Formulation of integer programs has been treated more as an art form than a science by the mathematical programming community. (See Jeroslow [1989] for a refreshingly different perspective on representation theories for mixed integer programming.) We believe that progress in representation theory can have an important influence on the future of integer programming as a broad-based problem solving methodology in combinatorial optimization.

Defining Terms

Column generation: A scheme for solving linear programs with a huge number of columns.

Cutting plane: A valid inequality for an integer polyhedron that separates the polyhedron from a given point outside it.

Extreme point: A corner point of a polyhedron.

Fathoming: Pruning a search tree.

Integer polyhedron: A polyhedron, all of whose extreme points are integer valued.

Linear program: Optimization of a linear function subject to linear equality and inequality constraints.

Mixed integer linear program: A linear program with the added constraint that some of the decision variables are integer valued.

Packing and covering: Given a finite collection of subsets of a finite ground set, to find an optimal subcollection that is pairwise disjoint (packing) or whose union covers the ground set (covering).

Polyhedron: The set of solutions to a finite system of linear inequalities on real-valued variables. Equivalently, the intersection of a finite number of linear half-spaces in $\Re^n$.

ρ-Approximation: An approximation method that delivers a feasible solution with an objective value within a factor ρ of the optimal value of a combinatorial optimization problem.

Relaxation: An enlargement of the feasible region of an optimization problem. Typically, the relaxation is considerably easier to solve than the original optimization problem.

References

Aarts, E.H.L. and Korst, J.H. 1989. *Simulated annealing and Boltzmann machines: A stochastic approach to Combinatorial Optimization and neural computing*, Wiley, New York.

Ahuja, R. K., Magnati, T. L., and Orlin, J. B. 1993. *Network Flows: Theory, Algorithms and Applications*. Prentice–Hall, Englewood Cliffs, NJ.

Akgül, M. 1984. *Topics in Relaxation and Ellipsoidal Methods, Research Notes in Mathematics,* Pitman.
Alizadeh, F. 1995. Interior point methods in semidefinite programming with applications to combinatorial optimization. *SIAM J. Optimization* 5(1):13–51.
Applegate, D., Bixby, R. E., Chvátal, V., and Cook, W. 1994. Finding cuts in large TSP's. *Tech. Rep.* Aug.
Arora, S., Lund, C., Motwani, R., Sudan, M., and Szegedy, M. 1992. Proof verification and hardness of approximation problems. In *Proc. 33rd IEEE Symp. Found. Comput. Sci.* pp. 14–23.
Balas, E., Ceria, S. and Cornuejols, G. 1993. A lift and project cutting plane algorithm for mixed 0-1 programs. *Mathematical Programming* 58: 295–324.
Barahona, F., Jünger, M., and Reinelt, G. 1989. Experiments in quadratic $0-1$ programming. *Math. Programming* 44:127–137.
Benders, J. F. 1962. Partitioning procedures for solving mixed-variables programming problems. *Numerische Mathematik* 4:238–252.
Berge, C. 1961. Farbung von Graphen deren samtliche bzw. deren ungerade Kreise starr sind (Zusammenfassung). Wissenschaftliche Zeitschrift, Martin Luther Universitat Halle-Wittenberg, Mathematisch-Naturwiseenschaftliche Reihe. pp. 114–115.
Berge, C. 1970. Sur certains hypergraphes generalisant les graphes bipartites. In *Combinatorial Theory and its Applications I.* P. Erdos, A. Renyi, and V. Sos eds., Colloq. Math. Soc. Janos Bolyai, 4, pp. 119–133. North Holland, Amsterdam.
Berge, C. 1972. Balanced matrices. *Math. Programming* 2:19–31.
Berge, C. and Las Vergnas, M. 1970. Sur un theoreme du type Konig pour hypergraphes. pp. 31–40. In *Int. Conf. Combinatorial Math.,* Ann. New York Acad. Sci. 175.
Bixby, R. E. 1994. Progress in linear programming. *ORSA J. Comput.* 6(1):15–22.
Bixby, R.E., Fenelon, M., Gu, Z., Rothberg, E., and Wunderling, R. 2001. M.I.P: Theory and practice: Closing the gap. Paper presented at Padberg-Festschrift, Berlin.
Bland, R., Goldfarb, D., and Todd, M. J. 1981. The ellipsoid method: a survey. *Operations Res.* 29:1039–1091.
Borning, A., ed. 1994. *Principles and Practice of Constraint Programming,* LNCS Vol. 874, Springer–Verlag.
Camion, P. 1965. Characterization of totally unimodular matrices. *Proc. Am. Math. Soc.* 16:1068–1073.
Cannon, T. L. and Hoffman, K. L. 1990. Large-scale zero-one linear programming on distributed workstations. *Ann. Operations Res.* 22:181–217.
Černikov, R. N. 1961. The solution of linear programming problems by elimination of unknowns. *Doklady Akademii Nauk* 139:1314–1317 (translation in 1961. *Soviet Mathematics Doklady* 2:1099–1103).
Chandru, V. and Hooker, J. N. 1991. Extended Horn sets in propositional logic. *JACM* 38:205–221.
Chandru, V. and Hooker, J. N. 1999. *Optimization Methods for Logical Inference,* Wiley Interscience.
Chopra, S., Gorres, E. R., and Rao, M. R. 1992. Solving Steiner tree problems by branch and cut. *ORSA J. Comput.* 3:149–156.
Chvátal, V. 1973. Edmonds polytopes and a hierarchy of combinatorial problems. *Discrete Math.* 4:305–337.
Chvátal, V. 1975. On certain polytopes associated with graphs. *J. Combinatorial Theory B* 18:138–154.
Conforti, M. and Cornuejols, G. 1992a. Balanced 0, ± 1 matrices, bicoloring and total dual integrality. Preprint, Carnegie Mellon University.
Conforti, M. and Cornuejols, G. 1992b. A class of logical inference problems solvable by linear programming. *FOCS* 33:670–675.
Conforti, M., Cornuejols, G., and De Francesco, C. 1993. Perfect 0, ± 1 matrices. Preprint, Carnegie Mellon University.
Conforti, M., Cornuejols, G., Kapoor, A., and Vuskovic, K. 1994. Balanced 0, ± 1 matrices. Pts. I–II. Preprints, Carnegie Mellon University.
Conforti, M., Cornuejols, G., Kapoor, A. Vuskovic, K., and Rao, M. R. 1994. Balanced matrices. In *Mathematical Programming, State of the Art 1994.* J. R. Birge and K. G. Murty, Eds., University of Michigan.
Conforti, M., Cornuejols, G., and Rao, M. R. 1999. Decomposition of balanced 0, 1 matrices. *Journal of Combinatorial Theory* Series B 77:292–406.

Conforti, M. and Rao, M. R. 1993. Testing balancedness and perfection of linear matrices. *Math. Programming* 61:1–18.

Cook, W., Lovász, L., and Seymour, P., eds. 1995. *Combinatorial Optimization: Papers from the DIMACS Special Year*. Series in discrete mathematics and theoretical computer science, Vol. 20, AMS.

Cornuejols, G. and Novick, B. 1994. Ideal 0, 1 matrices. *J. Combinatorial Theory* 60:145–157.

CPLEX 6.5. 1999. Using the CPLEX Callable Library and CPLEX Mixed Integer Library, Ilog Inc.

Crowder, H., Johnson, E. L., and Padberg, M. W. 1983. Solving large scale 0–1 linear programming problems. *Operations Res.* 31:803–832.

Cunningham, W. H. 1984. Testing membership in matroid polyhedra. *J. Combinatorial Theory* 36B:161–188.

Dantzig, G. B. and Wolfe, P. 1961. The decomposition algorithm for linear programming. *Econometrica* 29:767–778.

Ecker, J. G. and Kupferschmid, M. 1983. An ellipsoid algorithm for nonlinear programming. *Math. Programming* 27.

Edmonds, J. 1965. Maximum matching and a polyhedron with 0–1 vertices. *J. Res. Nat. Bur. Stand.* 69B:125–130.

Edmonds, J. 1970. Submodular functions, matroids and certain polyhedra. In *Combinatorial Structures and their Applications,* R. Guy, Ed., pp. 69–87. Gordon Breach, New York.

Edmonds, J. 1971. Matroids and the greedy algorithm. *Math. Programming* 127–136.

Edmonds, J. 1979. Matroid intersection. *Ann. Discrete Math.* 4:39–49.

Edmonds, J. and Giles, R. 1977. A min-max relation for submodular functions on graphs. *Ann. Discrete Math.* 1:185–204.

Edmonds, J. and Johnson, E. L. 1970. Matching well solved class of integer linear polygons. In *Combinatorial Structure and Their Applications.* R. Guy, Ed., Gordon and Breach, New York.

Fonlupt, J. and Zemirline, A. 1981. A polynomial recognition algorithm for $K_4 \backslash e$-free perfect graphs. *Res. Rep.,* University of Grenoble.

Fourer, R., Gay, D. M., and Kernighian, B. W. 1993. *AMPL: A Modeling Language for Mathematical Programming,* Scientific Press.

Fourier, L. B. J. 1827. In: Analyse des travaux de l'Academie Royale des Sciences, pendant l'annee 1824, Partie mathematique, *Histoire de l'Academie Royale des Sciences de l'Institut de France 7* (1827) xlvii-lv. (Partial English translation Kohler, D. A. 1973. *Translation of a Report by Fourier on his Work on Linear Inequalities. Opsearch* 10:38–42.)

Frank, A. 1981. A weighted matroid intersection theorem. *J. Algorithms* 2:328–336.

Fulkerson, D. R. 1970. The perfect graph conjecture and the pluperfect graph theorem. pp. 171–175. In *Proc. 2nd Chapel Hill Conf. Combinatorial Math. Appl.* R. C. Bose et al., Eds.

Fulkerson, D. R., Hoffman, A., and Oppenheim, R. 1974. On balanced matrices. *Math. Programming Study* 1:120–132.

Gass S. and Saaty, T. 1955. The computational algorithm for the parametric objective function. *Naval Research Logistics Quarterly* 2:39–45.

Gilmore, P. and Gomory, R. E. 1963. A linear programming approach to the cutting stock problem. Pt. I. *Operations Res.* 9:849–854; Pt. II. *Operations Res.* 11:863–887.

Glover, F. and Laguna, M. 1997. *Tabu Search*, Kluwer Academic Publishers.

Goemans, M. X. and Williamson, D. P. 1994. .878 approximation algorithms MAX CUT and MAX 2SAT. pp. 422–431. In *Proc. ACM STOC.*

Gomory, R. E. 1958. Outline of an algorithm for integer solutions to linear programs. *Bull. Am. Math. Soc.* 64:275–278.

Gomory, R. E. and Hu, T. C. 1961. Multi-terminal network flows. *SIAM J. Appl. Math.* 9:551–556.

Grötschel, M., Lovasz, L., and Schrijver, A. 1982. The ellipsoid method and its consequences in combinatorial optimization. *Combinatorica* 1:169–197.

Grötschel, M., Lovász, L., and Schrijver, A. 1988. *Geometric Algorithms and Combinatorial Optimization*. Springer–Verlag.

Haĉijan, L. G. 1979. A polynomial algorithm in linear programming. *Soviet Math. Dokl.* 20:191–194.
Held, M. and Karp, R. M. 1970. The travelling-salesman problem and minimum spanning trees. *Operations Res.* 18:1138–1162, Pt. II. 1971. *Math. Programming* 1:6–25.
Held, M., Wolfe, P., and Crowder, H. P. 1974. Validation of subgradient optimization. *Math. Programming* 6:62–88.
Hoffman, A. J. and Kruskal, J. K. 1956. Integral boundary points of convex polyhedra. In *Linear Inequalities and Related Systems,* H. W. Kuhn and A. W. Tucker, Eds., pp. 223–246. Princeton University Press, Princeton, NJ.
Hooker, J. N. 1988. Resolution vs cutting plane solution of inference problems: some computational experience. *Operations Res. Lett.* 7:1–7.
Hooker, J. N. 1992. Resolution and the integrality of satisfiability polytopes. Preprint, GSIA, Carnegie Mellon University.
Hooker, J. N. 1993. Towards and empirical science of algorithms. *Operations Res.* 42:201–212.
Hooker, J. N. and Vinay, V. 1995. Branching rules for satisfiability. In *Automated Reasoning* 15:359–383.
Huynh, T., Lassez C., and Lassez, J.-L. 1992. Practical issues on the projection of polyhedral sets. *Ann. Math. Artif. Intell.* 6:295–316.
IBM. 1991. *Optimization Subroutine Library—Guide and Reference (Release 2),* 3rd ed.
Iyengar, G. and Cezik, M. T. 2002. Cutting planes for mixed 0-1 semidefinite programs. *Proceedings of the VIII IPCO conference*.
Jeroslow, R. E. 1987. Representability in mixed integer programming, I: characterization results. *Discrete Appl. Math.* 17:223–243.
Jeroslow, R. E. and Lowe, J. K. 1984. Modeling with integer variables. *Math. Programming Stud.* 22:167–184.
Jeroslow, R. G. 1989. *Logic-Based Decision Support: Mixed Integer Model Formulation*. Ann. discrete mathematics, Vol. 40, North–Holland.
Jünger, M., Reinelt, G., and Thienel, S. 1995. Practical problem solving with cutting plane algorithms. In *Combinatorial Optimization: Papers from the DIMACS Special Year.* Series in discrete mathematics and theoritical computer science, Vol. 20, pp. 111–152. AMS.
Karmarkar, N. K. 1984. A new polynomial-time algorithm for linear programming. *Combinatorica* 4:373–395.
Karmarkar, N. K. 1990. An interior-point approach to NP-complete problems—Part I. In *Contemporary Mathematics,* Vol. 114, pp. 297–308.
Karp, R. M. and Papadimitriou, C. H. 1982. On linear characterizations of combinatorial optimization problems. *SIAM J. Comput.* 11:620–632.
Lawler, E. L. 1975. Matroid intersection algorithms. *Math. Programming* 9:31–56.
Lehman, A. 1979. On the width-length inequality, mimeographic notes (1965). *Math. Programming* 17:403–417.
Lin, S. and Kernighan, B.W. 1973. An effective heuristic algorithm for the travelling salesman problem. *Operations Research* 21: 498-516.
Lovasz, L. 1972. Normal hypergraphs and the perfect graph conjecture. *Discrete Math.* 2:253–267.
Lovasz, L. 1979. On the Shannon capacity of a graph. *IEEE Trans. Inf. Theory* 25:1–7.
Lovász, L. 1986. *An Algorithmic Theory of Numbers, Graphs and Convexity,* SIAM Press.
Lovasz, L. and Schrijver, A. 1991. Cones of matrices and setfunctions. *SIAM J. Optimization* 1:166–190.
Lustig, I. J., Marsten, R. E., and Shanno, D. F. 1994. Interior point methods for linear programming: computational state of the art. *ORSA J. Comput.* 6(1):1–14.
Martin, R. K. 1991. Using separation algorithms to generate mixed integer model reformulations. *Operations Res. Lett.* 10:119–128.
McDiarmid, C. J. H. 1975. Rado's theorem for polymatroids. *Proc. Cambridge Philos. Soc.* 78:263–281.
Megiddo, N. 1991. On finding primal- and dual-optimal bases. *ORSA J. Comput.* 3:63–65.
Mehrotra, S. 1992. On the implementation of a primal-dual interior point method. *SIAM J. Optimization* 2(4):575–601.

Nemhauser, G. L. and Wolsey, L. A. 1988. *Integer and Combinatorial Optimization*. Wiley.
Orlin, J. B. 2000. Very large-scale neighborhood search techniques. Featured Lecture at the International Symposium on Mathematical Programming, Atlanta, Georgia.
Padberg, M. W. 1973. On the facial structure of set packing polyhedra. *Math. Programming* 5:199–215.
Padberg, M. W. 1974. Perfect zero-one matrices. *Math. Programming* 6:180–196.
Padberg, M. W. 1993. Lehman's forbidden minor characterization of ideal 0, 1 matrices. *Discrete Math.* 111:409–420.
Padberg, M. W. 1995. *Linear Optimization and Extensions.* Springer–Verlag.
Padberg, M. W. and Rao, M. R. 1981. The Russian method for linear inequalities. Part III, bounded integer programming. Preprint, New York University, New York.
Padberg, M. W. and Rao, M. R. 1982. Odd minimum cut-sets and b-matching. *Math. Operations Res.* 7:67–80.
Padberg, M. W. and Rinaldi, G. 1991. A branch and cut algorithm for the resolution of large scale symmetric travelling salesman problems. *SIAM Rev.* 33:60–100.
Papadimitriou, C. H. and Yannakakis, M. 1991. Optimization, approximation, and complexity classes. *J. Comput. Syst. Sci.* 43:425–440.
Parker, G. and Rardin, R. L. 1988. *Discrete Optimization.* Wiley.
Picard, J. C. and Ratliff, H. D. 1975. Minimum cuts and related problems. *Networks* 5:357–370.
Pulleyblank, W. R. 1989. Polyhedral combinatorics. In *Handbooks in Operations Research and Management Science.* Vol. 1, Optimization, G. L. Nemhauser, A. H. G. Rinooy Kan, and M. J. Todd, eds., pp. 371–446. North–Holland.
Raghavan, P. and Thompson, C. D. 1987. Randomized rounding: a technique for provably good algorithms and algorithmic proofs. *Combinatorica* 7:365–374.
Rhys, J. M. W. 1970. A selection problem of shared fixed costs and network flows. *Manage. Sci.* 17:200–207.
Saraswat, V. and Van Hentenryck, P., eds. 1995. *Principles and Practice of Constraint Programming,* MIT Press, Cambridge, MA.
Savelsbergh, M. W. P., Sigosmondi, G. S., and Nemhauser, G. L. 1994. MINTO, a mixed integer optimizer. *Operations Res. Lett.* 15:47–58.
Schrijver, A. 1986. *Theory of Linear and Integer Programming.* Wiley.
Seymour, P. 1980. Decompositions of regular matroids. *J. Combinatorial Theory* B 28:305–359.
Shapiro, J. F. 1979. A survey of lagrangian techniques for discrete optimization. *Ann. Discrete Math.* 5:113–138.
Shmoys, D. B. 1995. Computing near-optimal solutions to combinatorial optimization problems. In *Combinatorial Optimization: Papers from the DI'ACS special year.* Series in discrete mathematics and theoretical computer science, Vol. 20, pp. 355–398. AMS.
Shor, N. Z. 1970. Convergence rate of the gradient descent method with dilation of the space. *Cybernetics* 6.
Spielman, D. A. and Tang, S.-H. 2001. Smoothed analysis of algorithms: Why the simplex method usually takes polynomial time. *Proceedings of the The Thirty-Third Annual ACM Symposium on Theory of Computing*, 296–305.
Truemper, K. 1992. Alpha-balanced graphs and matrices and GF(3)-representability of matroids. *J. Combinatorial Theory* B 55:302–335.
Weyl, H. 1935. Elemetere Theorie der konvexen polyerer. *Comm. Math. Helv.* Vol. pp. 3–18 (English translation 1950. *Ann. Math. Stud.* 24, Princeton).
Whitley, D. 1993. *Foundations of Genetic Algorithms 2*, Morgan Kaufmann.
Williams, H. P. 1987. Linear and integer programming applied to the propositional calculus. *Int. J. Syst. Res. Inf. Sci.* 2:81–100.
Williamson, D. P. 2000. The primal-dual method for approximation algorithms. *Proceedings of the International Symposium on Mathematical Programming*, Atlanta, Georgia.

Wolkowicz, W., Saigal, R. and Vanderberghe, L. eds. 2000. *Handbook of semidefinite programming*. Kluwer Acad. Publ.

Yannakakis, M. 1988. Expressing combinatorial optimization problems by linear programs. pp. 223–228. In *Proc. ACM Symp. Theory Comput.*

Ye, Y. 2000. Semidefinite programming for discrete optimization: Approximation and Computation. *Proceedings of the International Symposium on Mathematical Programming*, Atlanta, Georgia.

Ziegler, M. 1995. *Convex Polytopes.* Springer–Verlag.

II

Architecture and Organization

Computer architecture is the design and organization of efficient and effective computer hardware at all levels — from the most fundamental aspects of logic and circuit design to the broadest concerns of RISC, parallel, and high-performance computing. Individual chapters cover the design of the CPU, memory systems, buses, disk storage, and computer arithmetic devices. Other chapters treat important subjects such as parallel architectures, the interaction between computers and networks, and the design of computers that tolerate unanticipated interruptions and failures.

16 Digital Logic *Miriam Leeser*
Introduction • Overview of Logic • Concept and Realization of a Digital Gate • Rules and Objectives in Combinational Design • Frequently Used Digital Components • Sequential Circuits • ASICs and FPGAs — Faster, Cheaper, More Reliable Logic

17 Digital Computer Architecture *David R. Kaeli*
Introduction • The Instruction Set • Memory • Addressing • Instruction Execution • Execution Hazards • Superscalar Design • Very Long Instruction Word Computers • Summary

18 Memory Systems *Douglas C. Burger, James R. Goodman, and Gurindar S. Sohi*
Introduction • Memory Hierarchies • Cache Memories • Parallel and Interleaved Main Memories • Virtual Memory • Research Issues • Summary

19 Buses *Windsor W. Hsu and Jih-Kwon Peir*
Introduction • Bus Physics • Bus Arbitration • Bus Protocol • Issues in SMP System Buses • Putting It All Together — CCL-XMP System Bus • Historical Perspective and Research Issues

20 Input/Output Devices and Interaction Techniques *Ken Hinckley, Robert J. K. Jacob, and Colin Ware*
Introduction • Interaction Tasks, Techniques, and Devices • The Composition of Interaction Tasks • Properties of Input Devices • Discussion of Common Pointing Devices • Feedback and Perception — Action Coupling • Keyboards, Text Entry, and Command Input • Modalities of Interaction • Displays and Perception • Color Vision and Color Displays • Luminance, Color Specification, and Color Gamut • Information Visualization • Scale in Displays • Force and Tactile Displays • Auditory Displays • Future Directions

21 Secondary Storage Systems *Alexander Thomasian*
Introduction • Single Disk Organization and Performance • RAID Disk Arrays • RAID1 or Mirrored Disks • RAID5 Disk Arrays • Performance Evaluation Studies • Data Allocation and Storage Management in Storage Networks • Conclusions and Recent Developments

22 High-Speed Computer Arithmetic *Earl E. Swartzlander Jr.*
Introduction • Fixed Point Number Systems • Fixed Point Arithmetic Algorithms • Floating Point Arithmetic • Conclusion

23 Parallel Architectures *Michael J. Flynn and Kevin W. Rudd*
Introduction • The Stream Model • SISD • SIMD • MISD • MIMD • Network Interconnections • Afterword

24 Architecture and Networks *Robert S. Roos*
Introduction • Underlying Principles • Best Practices: Physical Layer Examples • Best Practices: Data-Link Layer Examples • Best Practices: Network Layer Examples • Research Issues and Summary

25 Fault Tolerance *Edward J. McCluskey and Subhasish Mitra*
Introduction • Failures, Errors, and Faults • Metrics and Evaluation • System Failure Response • System Recovery • Repair Techniques • Common-Mode Failures and Design Diversity • Fault Injection • Conclusion • Further Reading

16 Digital Logic

16.1 Introduction
16.2 Overview of Logic
16.3 Concept and Realization of a Digital Gate
CMOS Binary Logic Is Low Power • CMOS Switching Model for NOT, NAND, and NOR • Multiple Inputs and Our Basic Primitives • Doing It All with NAND
16.4 Rules and Objectives in Combinational Design
Boolean Realization: Half Adders, Full Adders, and Logic Minimization • Axioms and Theorems of Boolean Logic • Design, Gate-Count Reduction, and SOP/POS Conversions • Minimizing with Don't Cares • Adder/Subtractor • Representing Negative Binary Numbers
16.5 Frequently Used Digital Components
Elementary Digital Devices: ENC, DEC, MUX, DEMUX • The Calculator Arithmetic and Logical Unit
16.6 Sequential Circuits
Concept of a Sequential Device • The Data Flip-Flop and the Register • From DFF to Data Register, Shift Register, and Stack • Datapath for a 4-bit Calculator
16.7 ASICs and FPGAs — Faster, Cheaper, More Reliable Logic
FPGA Architecture • Higher Levels of Complexity

Miriam Leeser
Northeastern University

16.1 Introduction

This chapter explores combinational and sequential Boolean logic design as well as technologies for implementing efficient, high-speed digital circuits. Some of the most common devices used in computers and general logic circuits are described. Sections 16.2 through 16.4 introduce the fundamental concepts of logic circuits and in particular the rules and theorems upon which *combinational logic*, logic with no internal memory, is based. Section 16.5 describes in detail some frequently used combinational logic components, and shows how they can be combined to build the Arithmetic and Logical Unit (ALU) for a simple calculator. Section 16.6 introduces the subject of *sequential logic*, logic in which feedback and thus internal memory exist. Two of the most important elements of sequential logic design, the *data flip-flop* and the *register*, are introduced. Memory elements are combined with the ALU to complete the design of a simple calculator. The final section of the chapter examines field-programmable gate arrays that now provide fast, economical solutions for implementing large logic designs for solving diverse problems.

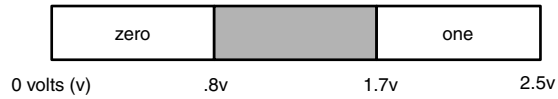


FIGURE 16.1 The states zero and one as defined in 2.5V CMOS logic.

16.2 Overview of Logic

Logic has been a favorite academic subject, certainly since the Middle Ages and arguably since the days of the greatness of Athens. That use of *logic* connoted the pursuit of orderly methods for defining theorems and proving their consistency with certain accepted propositions. In the middle of the 19th century, George Boole put the whole subject on a sound mathematical basis and spread "logic" from the Philosophy Department into Engineering and Mathematics. (Boole's original writings have recently been reissued [Boole 1998].) Specifically, what Boole did was to create an algebra of two-valued (*binary*) variables. Initially designated as *true* or *false*, these two values can represent any parameter that has two clearly defined states. Boolean algebras of more than two values have been explored, but the original binary variable of Boole dominates the design of circuitry for reasons that we will explore. This chapter presents some of the rules and methods of binary Boolean algebra and shows how it is used to design digital hardware to meet specific engineering applications.

One of the first things that must strike a reader who sees *true* or *false* proposed as the two identifiable, discrete states is that we live in a world with many half-truths, with hung juries that end somewhere between *guilty* and *not guilty*, and with "not bad" being a response that does not necessarily mean "good." The answer to the question: "Does a two-state variable really describe anything?" is properly: "Yes and no." This apparent conflict between the continuum that appears to represent life and the underlying reality of atomic physics, which is inherently and absolutely discrete, never quite goes away at any level. We use the words "quantum leap" to describe a transition between two states with no apparent state between them. Yet we know that the leaper spends some time between the two states.

A system that is well adapted to digital (discrete) representation is one that spends little time in a state of ambiguity. All digital systems spend some time in indeterminate states. One very common definition of the two states is made for systems operating between 2.5 volts (V) and ground. It is shown in Figure 16.1. One state, usually called *one*, is defined as any voltage greater than 1.7V. The other state, usually called *zero*, is defined as any voltage less than 0.8V.

The gray area in the middle is *ambiguous*. When an input signal is between 0.8 and 1.7V in a 2.5V CMOS (complementary metal–oxide–semiconductor) digital circuit, you cannot predict the output value. Most of what you will read in this chapter assumes that input variables are clearly assigned to the state *one* or the state *zero*. In real designs, there are always moments when the inputs are ambiguous. A good design is one in which the system never makes decisions based on ambiguous data. Such requirements limit the speed of response of real systems; they must wait for the ambiguities to settle out.

16.3 Concept and Realization of a Digital Gate

A *gate* is the basic building block of digital circuits. A gate is a circuit with one or more inputs and a single output. From a logical perspective in a binary system, any input or output can take on only the values *one* and *zero*. From an analog perspective (a perspective that will vanish for most of this chapter), the gates make transitions through the ambiguous region with great rapidity and quickly achieve an unambiguous state of *oneness* or *zeroness*.

In Boolean algebra, a good place to begin is with three operations: NOT, AND, and OR. These have similar meaning to their meaning in English. Given two input variables, called A and B, and an output variable X, X = NOT A is true when A is false, and false when A is true. X = A AND B is true when both inputs are true, and X = A OR B is true when either A or B is true (or both are true). This is called an

TABLE 16.1 The Boolean Operators of Two Input Variables

Inputs A B	True	False	A	NOT(A)	AND	OR	XOR	NAND	NOR	XNOR
0 0	1	0	0	1	0	0	0	1	1	1
0 1	1	0	0	1	0	1	1	1	0	0
1 0	1	0	1	0	0	1	1	1	0	0
1 1	1	0	1	0	1	1	0	0	0	1

TABLE 16.2 The Boolean Operators Extended to More than Two Inputs

Operation	Input Variables	Operator Symbol	Output = 1 if
NOT	A	$\overline{A}$	$A = 0$
AND	$A, B, \ldots$	$A \cdot B \cdot \cdots$	All of the set $[A, B, \ldots]$ are 1.
OR	$A, B, \ldots$	$A + B + \cdots$	Any of the set $[A, B, \ldots]$ are 1.
NAND	$A, B, \ldots$	$\overline{(A \cdot B \cdot \cdots)}$	Any of the set $[A, B, \ldots]$ are 0.
NOR	$A, B, \ldots$	$\overline{(A + B + \cdots)}$	All of the set $[A, B, \ldots]$ are 0.
XOR	$A, B, \ldots$	$A \oplus B \oplus \cdots$	The set $[A, B, \ldots]$ contains an odd number of 1's.
XNOR	$A, B, \ldots$	$A \odot B \odot \cdots$	The set $[A, B, \ldots]$ contains an even number of 1's.

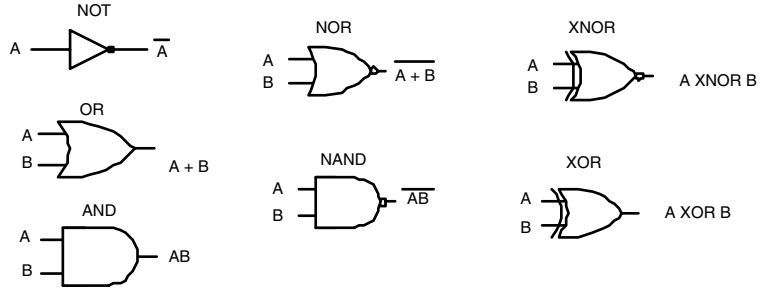


FIGURE 16.2 Commonly used graphical symbols for seven of the gates defined in Table 16.1.

"inclusive or" function because it includes the case where A and B are both true. There is another Boolean operator, *exclusive or*, that is true when either A or B, but not both, is true. In fact, there are 16 Boolean functions of two variables. The more useful functions are shown in truth table form in Table 16.1. These functions can be generalized to more than one variable, as is shown in Table 16.2.

AND, OR, and NOT are sufficient to describe all Boolean logic functions. Why do we need all these other operators?

Logic gates are themselves an abstraction. The actual physical realization of logic gates is with transistors. Most digital designs are implemented in CMOS technology. In CMOS and most other transistor technologies, logic gates are naturally inverting. In other words, it is very natural to build NOT, NAND, and NOR gates, even if it is more natural to think about positive logic: AND and OR. Neither XOR nor XNOR are natural building blocks of CMOS technology. They are included for completeness. AND and OR gates are implemented with NAND and NOR gates as we shall see.

One question that arises is: How many different gates do we really need? The answer is one. We normally admit three or four to our algebra, but one is sufficient. If we pick the right gate, we can build all the others. Later we will explore the minimal set.

There are widely used graphical symbols for these same operations. These are presented in Figure 16.2. The symbol for NOT includes both a buffer (the triangle) and the actual inversion operation (the open

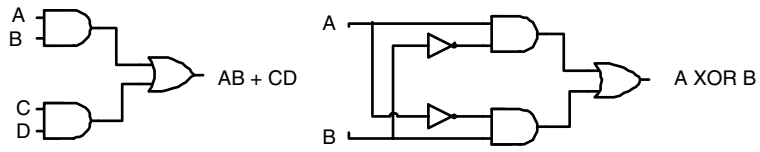


FIGURE 16.3 Two constructs built from the gates in column 1 of Figure 16.2. The first is a common construct in which if either of two paired propositions is TRUE, the output is TRUE. The second is XOR constructed from the more primitive gates, AND, OR, and NOT.

circle). Often, the inversion operation alone is used, as seen in the outputs of NAND, NOR, and XNOR. In writing Boolean operations we use the symbols $\overline{A}$ for NOT A, $A + B$ for A OR B, and $A \cdot B$ for A AND B. $A + B$ is called the sum of A and B and $A \cdot B$ is called the product. The operator for AND is often omitted, and the operation is implied by adjacency, just like in multiplication. To illustrate the use of these symbols and operators and to see how well these definitions fit common speech, Figure 16.3 shows two constructs made from the gates of Figure 16.2. These two examples show how to build the expression $AB + CD$ and how to construct an XOR from the basic gates AND, OR, and NOT.

The first construct of Figure 16.3 would fit the logic of the sentence: "I will be content if my federal and state taxes are lowered (A and B, respectively), or if the money that I send is spent on reasonable things and spent effectively (C and D, respectively)." You would certainly expect the speaker to be content if either pair is TRUE and most definitely content if both are TRUE. The output on the right side of the construct is TRUE if either or both of the inputs to the OR is TRUE. The outputs of the AND gates are TRUE when both of their inputs are TRUE. In other words, both state and federal taxes must be reduced to make the top AND's output TRUE.

The right construct in Figure 16.3 gives an example of how one can build one of the basic logic gates, in this case the XOR gate, from several of the others. Let us consider the relationship of this construct to common speech. The sentence: "With the time remaining, we should eat dinner or go to a movie." The implication is that one cannot do both. The circuit on the right of Figure 16.3 would indicate an acceptable decision (TRUE if acceptable) if either movie or dinner were selected (*asserted* or made TRUE) but an unacceptable decision if both or neither were asserted.

What makes logic gates so very useful is their speed and remarkably low cost. On-chip logic gates today can respond in less than a nanosecond and can cost less than 0.0001 cent each. Furthermore, a rather sophisticated decision-making apparatus can be designed by combining many simple-minded binary decisions. The fact that it takes many gates to build a useful apparatus leads us back directly to one of the reasons why binary logic is so popular. First we will look at the underlying technology of logic gates. Then we will use them to build some useful circuits.

16.3.1 CMOS Binary Logic Is Low Power

A modern microcomputer chip contains more than 10 million logic gates. If all of those gates were generating heat at all times, the chip would melt. Keeping them cool is one of the most critical issues in computer design. Good thermal designs were significant parts of the success of Cray, IBM, Intel, and Sun. One of the principal advantages of CMOS binary logic is that it can be made to expend much less energy to generate the same amount of calculation as other forms of circuitry.

Gates are classified as **active logic** or **saturated logic**, depending on whether they control the current continuously or simply switch it on or off. In active logic, the gate has a considerable voltage across it and conducts current in all of its states. The result is that power is continually being dissipated. In saturated logic, the TRUE–FALSE dichotomy has the gate striving to be perfectly connected to the power bus when the output voltage is high and perfectly connected to the ground bus when the voltage is low. These are zero-dissipation ideals that are not achieved in real gates, but the closer one gets to the ideal, the better the

gate. When you start with more than 1 million gates per chip, small reductions in power dissipation make the difference between usable and unusable chips.

Saturated logic is *saturated* because it is driven hard enough to ensure that it is in a minimum-dissipation state. Because it takes some effort to bring such logic out of saturation, it is a little slower than active logic. Active logic, on the other hand, is always dissipative. It is very fast, but it is always getting hot. Although it has often been the choice for the most active circuits in the fastest computers, active logic has never been a major player, and it owns a diminishing role in today's designs. This chapter focuses on today's dominant family of binary, saturated logic, which is CMOS: Complementary Metal Oxide Semiconductor.

16.3.2 CMOS Switching Model for NOT, NAND, and NOR

The metal–oxide–semiconductor (MOS) transistor is the oldest transistor in concept and still the best in one particular aspect: its control electrode — also called a *gate* but in a different meaning of that word from *logic gate* — is a purely capacitive load. Holding it at constant voltage takes no energy whatsoever. These MOS transistors, like most transistors, come in two types. One turns on with a positive voltage; the other turns off with a positive voltage. This pairing allows one to build *complementary* gates, which have the property that they dissipate no energy except when switching. Given the large number of logic gates and the criticality of energy dissipation, zero dissipation in the static state is enormously compelling. It is small wonder that the complementary metal–oxide–semiconductor (CMOS) gate dominates today's digital technology.

Consider how we can construct a set of primitive gates in the CMOS family. The basic element is a pair of switches in series, the NOT gate. This basic building block is shown in Figure 16.4. The switching operation is shown in the two drawings to the right. If the input is low, the upper switch is closed and the lower one is open — complementary operation. This connects the output to the high side. Apart from voltage drops across the switch itself, the output voltage becomes the voltage of the high bus. If the input now goes high, both switches flip and the output is connected, through the resistance of the switch, to the ground bus. High–in, low–out, and vice versa. We have an inverter. Only while the switches are switching is there significant current flowing from one bus to the other. Furthermore, if the loads are other CMOS switches, only while the gates are charging is any current flowing from bus to load. Current flows when charging or discharging a load. Thus, in the static state, these devices dissipate almost no power at all. Once one has the CMOS switch concept, it is easy to show how to build NAND and NOR gates with multiple inputs.

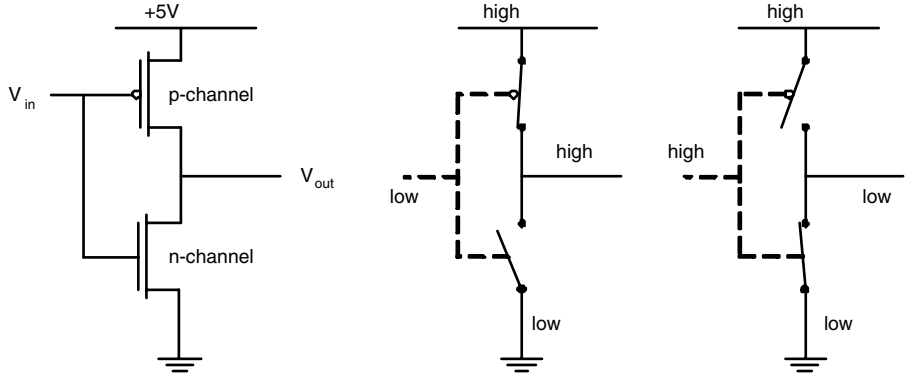


FIGURE 16.4 A CMOS inverter shown as a pair of transistors with voltage and ground and also as pairs of switches with logic levels. The open circle indicates logical negation (NOT).

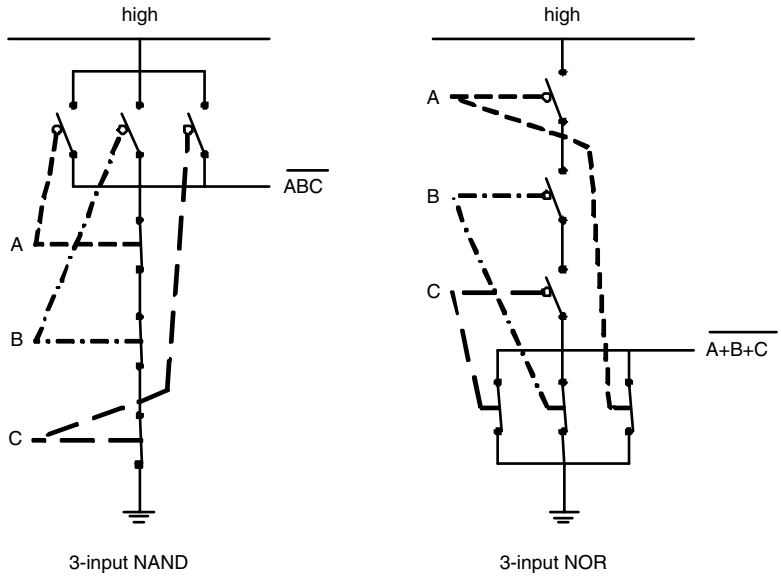


FIGURE 16.5 Three pairs of CMOS switches arranged on the left to execute the three-input NAND function and on the right the three-input NOR. The switches are shown with all the inputs high, putting the output in the low state.

16.3.3 Multiple Inputs and Our Basic Primitives

Let us look at the switching structure of a 3-input NAND and 3-input NOR, just to show how multiple-input gates are created. The basic inverter, or NOT gate of Figure 16.4 is our paradigm; if the lower switch is closed, the upper one is open, and vice versa. To go from NOT to an N-input NAND, make the single lower switch in the NOT a series of N switches, so only one of these need be open to open the circuit. Then change the upper complementary switch in the NOT into N parallel switches. With these, only one switch need be closed to connect the circuit. Such an arrangement with $N = 3$ is shown on the left in Figure 16.5. On the left, if any input is low, the output is high. On the right is the construction for NOR. All three inputs must be low to drive the output high.

An interesting question at this point is: How many inputs can such a circuit support? The answer is called the *fan-in* of the circuit. The fan-in depends mostly on the resistance of each switch in the series string. That series of switches must be able to *sink* a certain amount of current to ground and still hold the output voltage at 0.8V or less over the entire temperature range specified for the particular class of gate. In most cases, six or seven inputs would be considered a reasonable limit. The analogous question at the output is: How many gates can this one gate drive? This is the *fan-out* of the gate. It too needs to sink a certain amount of current through the series string. This minimum sink current represents a central design parameter. Logic gates can be designed with a considerably higher fan-out than fan-in.

16.3.4 Doing It All with NAND

We think of the basic logic operators as being NOT, AND, and OR, because these seem to be the most natural. When it comes to building logic gates out of CMOS transistor technology, as we have just seen, the "natural" logic gates are NOTs, NANDs, and NORs.

To build an AND or an OR gate, you take a NAND or NOR and add an inverter. The more primitive nature of NAND and NOR comes about because transistor switches are inherently inverting. Thus, a single-stage gate will be NAND or NOR; AND and OR gates require an extra stage. If this is the way one were to implement a design with a million gates, a million extra inverters would be required. Each extra stage requires extra area and introduces longer propagation delays. Simplifying logic to eliminate delay

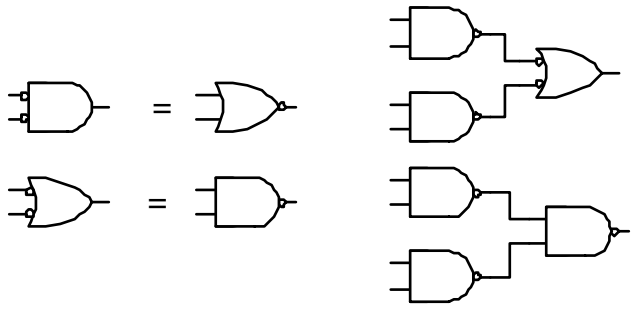


FIGURE 16.6 On the left, the two forms of De Morgan's theorem in logic gates. On the right, the two forms of the circuit on the left of Figure 16.3. In the upper form, we have replaced the lines between the ANDs and OR with two inverters in series. Then, we have used the lower form of De Morgan's theorem to replace the OR and its two inverters with a NAND. The resulting circuit is all-NAND and is simpler to implement than the construction from AND and OR in Figure 16.3.

and unnecessary heat are two of the most important objectives of logic design. Instead of using an inverter after each NAND or NOR gate, most designs use the inverting gates directly. We will see how Boolean logic helps us do this. Consider the declaration: "Fred and Jack will come over this afternoon." This is equivalent to saying: "Fred will stay away or Jack will stay away, NOT." This strange construct in English is an exact formulation of one of two relationships in Boolean logic known as *De Morgan's theorems*. More formally:

$$\overline{A \cdot B} = \overline{A} + \overline{B}$$

$$\overline{A + B} = \overline{A} \cdot \overline{B}$$

In other words, the NAND of A and B is equivalent to the OR of (not A) and (not B). Similarly, the NOR of A and B is equivalent to the AND of (not A) and (not B). These two statements can be represented at the gate level as shown in Figure 16.6.

De Morgan's theorems show that a NAND can be used to implement a NOR if we have inverters. It turns out that a NAND gate is the only gate required. Next we will show that a NOT gate (inverter) can be constructed from a NAND. Once we have shown that NORs and NOTs can be constructed out of NANDs, only NAND gates are required. An AND gate is a NAND followed by a NOT and an OR gate is a NOR followed by a NOT. Thus, all other logic gates can be implemented from NANDs. The same is true of NOR gates; all other logic gates can be implemented from NORs.

Take a NAND gate and connect both inputs to the same input A. The output is the function $\overline{(A \cdot A)}$. Since A AND A is TRUE only if A is TRUE ($AA = A$), we have just constructed our inverter. If we actually wanted an inverter, we would not use a two-input gate where a one-input gate would do. But we could. This exercise shows that the minimal number of logic gates required to implement all Boolean logic functions is one. In reality, we use AND, OR, and NOT when using positive logic, and NAND, NOR, and NOT when using negative logic or thinking about how logic gates are implemented with transistors.

16.4 Rules and Objectives in Combinational Design

Once the concept of a logic gate is established, the next natural question is: What useful devices can you build with them? We will look at a few basic building blocks and show how you can put them together to build a simple calculator.

The components of digital circuits can be divided into two classes. The first class of circuits has outputs that are simply some logical combination of their inputs. Such circuits are called **combinational**. Examples include the gates we have just looked at and those that we will examine in this section and in Section 16.5. The other class of circuits, constructed from combinational gates, but with the addition of internal feedback, have the property of *memory*. Thus, their output is a function not only of their inputs but also of their

previous state(s). Because such circuits go through a sequence of states, they are called **sequential**. These will be discussed in Section 16.6.

The two principal objectives in digital design are functionality and minimum cost. Functionality requires not only that the circuit generates the correct outputs for any possible inputs, but also that those outputs be available quickly enough to serve the application. Minimum cost must include both the design effort and the cost of production and operation. For very small production runs ($<10,000$), one wants to "program" off-the-shelf devices. For very large runs, costs focus mostly on manufacture and operation. The *operation* costs are dominated by cooling or battery drain, where these necessary peripherals add weight and complexity to the finished product. To fit in off-the-shelf devices, to reduce delays between input and output, and to reduce the gate count and thus the dissipation for a given functionality, designs must be realized with the smallest number of gates possible. Many design tools have been developed for achieving designs with minimum gate count. In this section and the next, we will develop the basis for such minimization in a way that assures the design achieves logical functionality.

16.4.1 Boolean Realization: Half Adders, Full Adders, and Logic Minimization

One of the basic units central to a calculator or microprocessor is a binary adder. We will consider how an adder is implemented from logic gates. A straightforward way to specify a Boolean logic function is by using a truth table. This table enumerates the output for every possible combinations of inputs. Truth tables were used in Table 16.1 to specify different Boolean functions of two variables. Table 16.3 shows the truth table for a Boolean operation that adds two one bit numbers A and B and produces two outputs: the sum bit S and the carry-out C. Because binary numbers can only have the values 1 or 0, adding two binary numbers each of value 1 will result in there being a carry-out. This operation is called a *half adder*.

To implement the half adder with logic gates, we need to write Boolean logic equations that are equivalent to the truth table. A separate Boolean logic equation is required for each output. The most straightforward way to write an equation from the truth table is to use Sum of Products (SOP) form to specify the outputs as a function of the inputs. An SOP expression is a set of "products" (ANDs) that are "summed" (ORed) together. Note that any Boolean formula can be expressed in SOP or POS (Product of Sums) form.

Let's consider output S. Every line in the truth table that has a 1 value for an output corresponds to a term that is ORed with other terms in SOP form. This term is formed by ANDing together all of the input variables. If the input variable is a 1 to make the output 1, the variable appears as is in the AND term. If the input is a zero to make the output 1, the variable appears negated in the AND term. Let's apply these rules to the half adder. The S output has two combinations of inputs that result in its output being 1; therefore, its SOP form has two terms ORed together. The C output only has one AND or product term, because only one combination of inputs results in a 1 output. The entire truth table can be summarized as:

$$S = \overline{A} \cdot B + A \cdot \overline{B}$$

$$C = A \cdot B$$

Note that we are implicitly using the fact that A and B are Boolean inputs. The equation for C can be read "C is 1 when A and B are both 1." We are assuming that C is zero in all other cases. From the Boolean

TABLE 16.3 Truth Table for a Half Adder

Inputs		Outputs	
A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

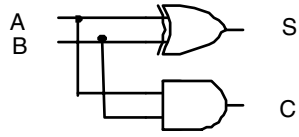


FIGURE 16.7 The gate level implementation of a half adder.

TABLE 16.4 Binary Representation of Decimal Numbers

Decimal	Binary
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

TABLE 16.5 Adding 3-Bit Binary Numbers

1	0		carry bits
0	1	0	2
0	1	1	+3
1	0	1	= 5

logic equations, it is straightforward to implement S and C with logic gates, as shown as in Figure 16.7. The logical function for S is that of an XOR gate, so we show S as an XOR gate in the figure.

The half adder is a building block in an n-bit binary adder. An n-bit binary adder adds n-bit numbers represented in base 2. Table 16.4 shows the representation of 3-bit, unsigned binary numbers. The leftmost bit is the most significant bit. It is in the 4's place. The middle bit represents the 2's place and the rightmost bit represents the 1's place. The largest representable number, 111_2 represents $4 + 2 + 1$ or 7 in decimal.

Let's examine adding two binary numbers. Table 16.5 shows the operation $2 + 3 = 5$ in binary. The top row is the carry-out from the addition in the previous bit location. Notice that there is a carry-out bit with value 1 from the second position to the third (leftmost) bit position.

The half adder described above has two inputs: A and B. This can be used for the rightmost bit where there is no carry-in bit. For other bit positions we use a *full adder* with inputs A, B, and C_{in} and outputs S and C_{out}. The truth table for the full adder is given in Table 16.6. Note that a full adder adds one bit position; it is *not* an n-bit adder.

To realize the full adder as a circuit we need to design it using logic gates. We do this in the same manner as with the half adder, by writing a logic equation for each of the outputs separately. For each 1 in the truth table on the output of the function, there is an AND term in the Sum of Products representation. Thus, there are four AND terms for the S equation and four AND terms for the C_{out} equation. These equations are given below:

$$S = \overline{A} \cdot \overline{B} \cdot C_{in} + \overline{A} \cdot B \cdot \overline{C_{in}} + A \cdot \overline{B} \cdot \overline{C_{in}} + ABC_{in}$$

$$C_{out} = \overline{A} \cdot BC_{in} + A \cdot \overline{B} \cdot C_{in} + AB \cdot \overline{C_{in}} + ABC_{in}$$

These equations for S and C_{out} are logically correct, but we would also like to use the minimum number of logic gates to implement these functions. The fewer gates used, the fewer gates that need to switch, and

TABLE 16.6 Truth Table for a Full Adder

	Inputs			Outputs	
	A	B	C_{in}	S	C_{out}
0	0	0	0	0	0
1	0	0	1	1	0
2	0	1	0	1	0
3	0	1	1	0	1
4	1	0	0	1	0
5	1	0	1	0	1
6	1	1	0	0	1
7	1	1	1	1	1

hence the smaller amount of power that is dissipated. Next, we will look at applying the rules of Boolean logic to minimize our logic equations.

16.4.2 Axioms and Theorems of Boolean Logic

Our goal is to use the minimum number of logic gates to implement a design. We use logic rules or axioms. These were first described by George Boole, hence the term Boolean algebra. Many of the axioms and theorems of Boolean algebra will seem familiar because they are similar to the rules you learned in algebra in high school. Let us be formal here and state the axioms:

1. Variables are binary. This means that every variable in the algebra can take on one of two values and these two values are not the same. Usually, we will choose to call the two values 1 and 0, but other binary pairs, such as TRUE and FALSE, and HIGH and LOW, are widely used and often more descriptive. Two binary operators, AND ($\cdot$) and OR ($+$), and one unary operator, NOT, can transform variables into other variables. These operators were defined in Table 16.2.
2. Closure: The AND or OR of any two variables is also a binary variable.
3. Commutativity: $A \cdot B = B \cdot A$ and $A + B = B + A$.
4. Associativity: $(A \cdot B) \cdot C = A \cdot (B \cdot C)$ and $(A + B) + C = A + (B + C)$.
5. Identity elements: $A \cdot 1 = 1 \cdot A = A$ and $A + 0 = 0 + A = A$.
6. Distributivity: $A \cdot (B + C) = A \cdot B + A \cdot C$ and $A + (B \cdot C) = (A + B) \cdot (A + C)$. (The usual rules of algebraic hierarchy are used here where $\cdot$ is done before $+$.)
7. Complementary pairs: $A \cdot \overline{A} = 0$ and $A + \overline{A} = 1$.

These are the axioms of this algebra. They are used to prove further theorems. Each algebraic relationship in Boolean algebra has a *dual*. To get the dual of an axiom or a theorem, one simply interchanges AND and OR as well as 0 and 1. Because of this principle of *duality*, Boolean algebra axioms and theorems come in pairs. The principle of duality tells us that if a theorem is true, then its dual is also true.

In general, one can prove a Boolean theorem by exhaustion — that is, by listing all of the possible cases — although more abstract algebraic reasoning may be more efficient. Here is an example of a pair of theorems based on the axioms given above:

Theorem 16.1 (Idempotency). $A \cdot A = A$ *and* $A + A = A$.

***Proof* 16.1** The definition of AND in Table 16.1 can be used with exhaustion to complete the proof for the first form.

$$A \text{ is } 1: \quad 1 \cdot 1 = 1 = A$$
$$A \text{ is } 0: \quad 0 \cdot 0 = 0 = A$$

The second form follows as the dual of the first.

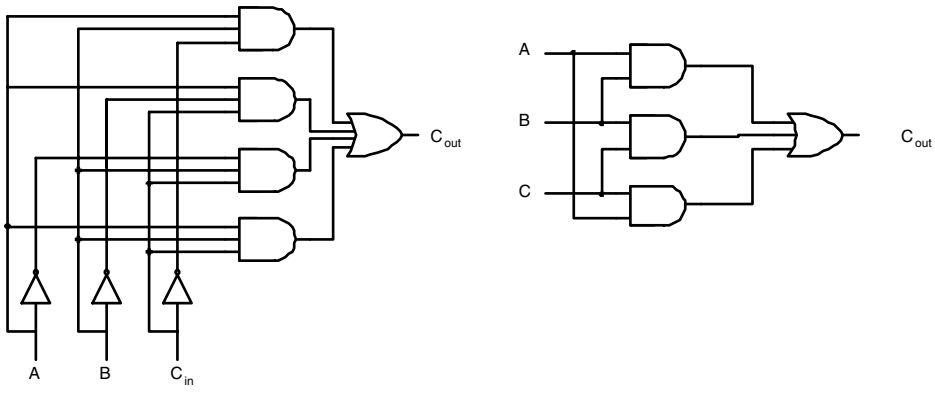


FIGURE 16.8 The direct and reduced circuits for computing the carry-out from the three inputs to the full adder.

Now let us consider reducing the expression from the previous section:

$$C_{out} = \overline{A} \cdot BC_{in} + A \cdot \overline{B} \cdot C_{in} + AB \cdot \overline{C_{in}} + ABC_{in}$$

First we apply idempotency twice to triplicate the last term on the right and put the extra terms after the first and second terms by repeated application of axiom 3:

$$C_{out} = \overline{A} \cdot BC_{in} + ABC_{in} + A \cdot \overline{B} \cdot C_{in} + ABC_{in} + AB \cdot \overline{C_{in}} + ABC_{in}$$

Now we apply axioms 4, 3, and 6 to obtain:

$$C_{out} = (\overline{A} + A)BC_{in} + A(\overline{B} + B)C_{in} + AB(\overline{C} + C)$$

And finally, we apply axioms 7 and 5 to obtain:

$$C_{out} = AB + AC_{in} + BC_{in}$$

The reduced equation certainly looks simpler; let's consider the gate representation of the two equations. This is shown in Figure 16.8. From four 3-input ANDs to three 2-input ANDs and from a 4-input OR to a 3-input OR is a major saving in a basically simple circuit.

The reduction is clear. The savings in a chip containing more than a million gates should build some enthusiasm for gate simplification. What is probably not so clear is how you could know that the key to all of this saving was knowing to make two extra copies of the fourth term in the direct expression. It turns out that there is a fairly direct way to see what you have to do, one that takes advantage of the eye's remarkable ability to see a pattern. This tool, the **Karnaugh map**, is the topic of the next section. □

16.4.3 Design, Gate-Count Reduction, and SOP/POS Conversions

The *truth table* for the full adder was given is Table 16.6. All possible combinations of the three input bits appear in the second through fourth columns. Note that the first column is the *numerical value* if I interpret those three bits as an unsigned binary number. So, 000 is the value 0, 101 is the value 5, etc.

Let's rearrange the rows of the truth table so that rather than being in increasing numerical order, the truth table values are listed in a way that each row differs from its neighbors by only one bit value. (Note that the fourth and fifth rows (entries for 2 and 4) differ by more than one bit value.) It should become apparent soon why you would want to do this. The result will be Table 16.7.

Consider the last two lines in Table 16.7, corresponding to 6 and 7. Both have $C_{out} = 1$. On the input side, the pair is represented as $AB\overline{C}_{in} + ABC_{in} = AB(\overline{C}_{in} + C_{in})$.

TABLE 16.7 Truth Table for Full Adder with Rows Rearranged

Input	ABC_{in}	S	C_{out}
0	000	0	0
1	001	1	0
3	011	0	1
2	010	1	0
4	100	1	0
5	101	0	1
7	111	1	1
6	110	0	1

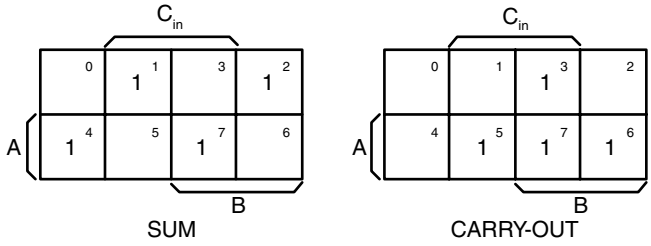


FIGURE 16.9 Karnaugh maps for SUM and CARRY-OUT. The numbers in the cell corners give the bit patterns of ABC_{in}. The cells whose outputs are 1 are marked; those whose outputs are 0 are left blank.

The algebraic reduction operation shows up as adjacency in the table. In the same way, the 5,7 pair can be reduced. The two are adjacent and both C_{out} outputs are 1. It is less obvious in the truth table, but notice that 3,7 also forms just such a pair. In other words, all of the steps proposed in algebra are "visible" in this truth table. To make adjacency even clearer, we arrange the groups of four, one above the other, in a table called a Karnaugh map after its inventor, M. Karnaugh [1953]. In this map, each possible combination of inputs is represented by a box. The contents of the box are the output for that combination of inputs. Adjacent boxes all have numerical values exactly one bit different from their neighbors on any side. It is customary to mark the asserted outputs (the 1's) but to leave the unasserted cells blank (for improved readability). The tables for S and C_{out} are shown in Figure 16.9. The two rows are just the first and second group of four from the truth table with the output values of the appropriate column. First convince yourself that each and every cell differs from any of its neighbors (no diagonals) by precisely one bit. The neighbors of an outside cell include the opposite outside cell. That is, they wrap around. Thus, 2 and 0 or 4 and 6 are neighbors. The Karnaugh map (or K-map) simply shows the relationships of the outputs of conjugate pairs, which are sets of inputs that differ in exactly one bit location. The item that most people find difficult about K-maps is the meaning and arrangement of the input variables around the map. If you think of these input variables as the bits in a binary number, the arrangement is more logical. The difference between the first four rows of the truth table and the second four is that A has the value 0 in the first four and the value 1 in the second four. In the map, this is shown by having A indicated as asserted in the second row. In other words, where the input parameter is placed, it is asserted. Where it is not placed, it is unasserted. Accordingly, the middle two columns are those cells that have C_{in} asserted. The right two columns have B asserted. Column 3 has both B and C_{in} asserted.

Let us look at how the carry-out map implies gate reduction while sum's K-map shows that no reduction is possible. Because we are looking for conjugate pairs of asserted cells, we simply look for adjacent pairs of 1's. The carry-out map has three such pairs; sum has none. We take pairs, pairs of pairs, or pairs of pairs of pairs — any rectangular grouping of 2^n cells with all 1's. With carry-out, this gives us the groupings shown in Figure 16.10.

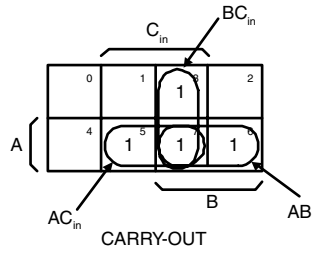


FIGURE 16.10 The groupings of conjugate pairs in CARRY-OUT.

The three groupings do the three things that we must always achieve:

1. The groups must cover all of the 1's (and none of the 0's).
2. Each group must include at least one cell not included in any other group.
3. Each group must be as large a rectangular box of 2^n cells as can be drawn.

The last rule says that in Figure 16.10 none of these groups can cover only one cell. Once we fulfill these three rules, we are assured of a minimal set, which is our goal. Although there is no ambiguity in the application of these rules in this example, there are other examples where more than one set of groups results in a correct, minimal set. K-maps can be used for functions of up to six input variables, and are useful aids for humans to minimize logic functions. Computer-aided design programs use different techniques to accomplish the same goal.

Writing down the solution once you have done the groupings is done by reading the specification of the groups. The vertical pair in Figure 16.10 is BC_{in}. In other words, that pair of cells is uniquely defined as having B and C_{in} both 1. The other two groups are indicated in the figure. The sum of those three (where "+" is OR) is the very function we derived algebraically in the last section. Notice how you could know to twice replicate cell 7. It occurs in three different groups. It is important to keep in mind that the Karnaugh map simply represents the algebraic steps in a highly visual way. It is not magical or intrinsically different from the algebra.

We have used the word "cell" to refer to a single box in the K-map. The formal name for a cell whose value is 1 is the *minterm* of the function. Its counterpart, the *maxterm*, comprises all the cells that represent an output value of 0. Note that all cells are both possible minterms and possible maxterms.

Two more examples will complete our coverage of K-maps. One way to specify a function is to list the minterms in the form of a summation. For example, $C_{out} = \sum(2, 5, 6, 7)$. Consider the arbitrary four-input function $F(X, Y, Z, T) = \sum(0, 1, 2, 3, 4, 8, 9, 12, 15)$. With four input variables, there are 16 possible input states, and every minterm must contact four neighbors. That can be accomplished in a 4×4 array of cells as shown in Figure 16.11. Convince yourself that each cell is properly adjacent to its neighbors. For example, 11_{10} (1011) is adjacent to 15 (1111), 9 (1001), 10_{10} (1010), and 3 (0011) with each neighbor differing by one bit. Now consider the groupings. Minterm 15 has no neighbors whose value is 1. Hence, it forms a group on its own, represented by the AND of all four inputs. The top row and first columns can each be grouped as a pair of pairs. It takes only two variables to specify such a group. For example, the top row includes all terms of the form $00xx$, and the first column includes all the terms of the form $xx00$. This leaves us but one uncovered cell, 9. You might be tempted to group it with its neighbor, 8, but rule 3 demands that we make as large a covering as possible. We can make a group of four by including the neighbors 0 and 1 on top. Had we not done that, the bottom pair would be $X \cdot \overline{Y} \cdot \overline{Z}$; but by increasing the coverage, we get that down to $\overline{Y} \cdot \overline{Z}$, a 2-input AND vs. a 3-input AND. The final expression is

$$F(X, Y, Z, T) = \overline{X} \cdot \overline{Y} + \overline{Y} \cdot \overline{Z} + \overline{Z} \cdot \overline{T} + XYZT$$

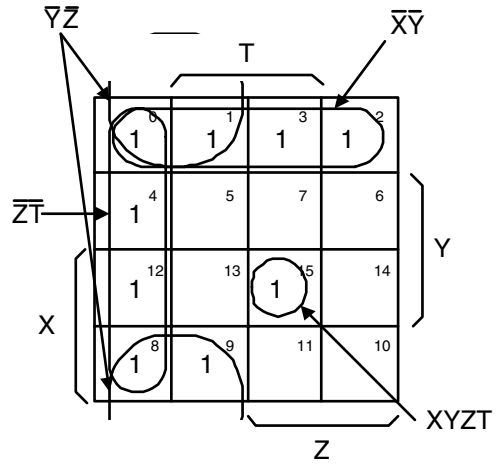


FIGURE 16.11 The K-map for $F(X, Y, Z, T) = \sum(0, 1, 2, 3, 4, 8, 9, 12, 15)$ with the minterm groupings shown.

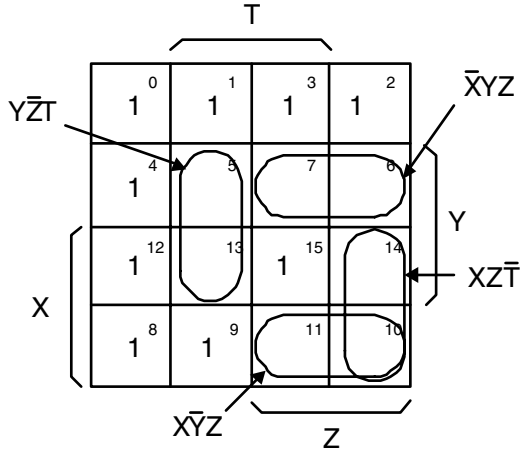


FIGURE 16.12 The K-map for the complement of F from Figure 16.11.

The above function is in Sum of Products (SOP) form because it is a set of products that are summed together. It is just as easy to generate the function with a Product of Sums (POS) where several OR gates are joined by a single AND. To get to that expression, we find $\overline{F}$, the complement of F, and then convert to F using De Morgan's theorem. $\overline{F}$ is obtained by grouping the cells where F is not asserted — the zero cells. This is shown in Figure 16.12, where we get the expression $\overline{F} = \overline{X}YZ + XZ\overline{T} + X\overline{Y}Z + Y\overline{Z}T$.

Let us convert from $\overline{F}$ to F using De Morgan's theorem to get the POS form:

$$\begin{aligned} F &= \overline{(\overline{X}YZ + XZ\overline{T} + X\overline{Y}Z + Y\overline{Z}T)} \\ &= (X + \overline{Y} + \overline{Z})(\overline{X} + \overline{Z} + T)(\overline{X} + Y + \overline{Z})(\overline{Y} + Z + \overline{T}) \end{aligned}$$

Why would one want to do this? Economy of gates. Sometimes the SOP form has fewer gates, sometimes the POS form does. In this example, the SOP form is somewhat more economical.

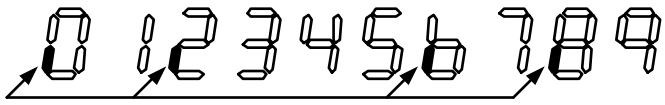


FIGURE 16.13 Segment *e* of the seven-segment display whose decoder we are going to minimize.

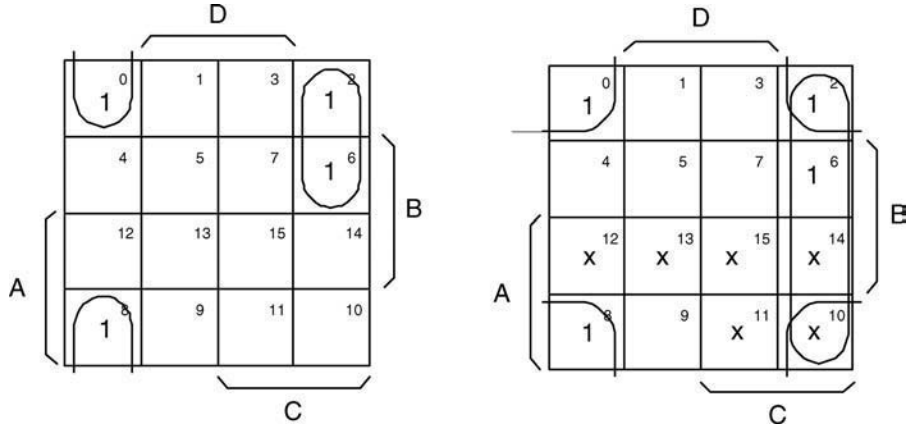


FIGURE 16.14 Minimization of Se without and with deliberate assignment of the *don't cares*.

16.4.4 Minimizing with Don't Cares

Sometimes, we can guarantee that some combination of inputs will never occur. I don't care what the output value is for that particular combination of inputs because I know that the output can never occur. This is known as an "output" don't care. I can set these outputs to any value I want. The best way to do this is to set these outputs to values that will minimize the gate count of the entire circuit.

An example is the classic seven-segment numerical display that is common in watches, calculators, and other digital displays. The input to a seven-segment display is a number coded in binary-coded-decimal (BCD), a 4-bit representation with 16 possible input combinations, but only the 10 numbers $0, \ldots, 9$ ever occur. The states $10, \ldots, 15$ are called **don't cares**. One can assign them to achieve minimum gate count. Consider the entire number set that one can display using seven line segments. We will consider the one line segment indicated by the arrows in Figure 16.13. It is generally referred to as "segment *e*," and it is asserted only for the numbers 0, 2, 6, and 8.

Now we will minimize Se(A, B, C, D) with and without the use of the *don't cares*. We put an "X" wherever the *don't cares* may lie in the K-map and then treat each one as either 0 or 1 in such a way as to minimize the gate count. This is shown in Figure 16.14.

We are not doing something intrinsically different on the right and left. On the left, all of the *don't cares* are assigned to 0. In other words, if someone enters a 14 into this 0 : 9 decoder, it will not light up segment *e*. But because this is a *don't care* event, we examine the map to see if letting it light up on 14 will help. The grouping with the choice of *don't care* values is decidedly better. We choose to assert *e* only for *don't cares* 10 and 14, but those assignments reduce the gates required from two 3-input ANDs to two 2-input ANDs. For this little circuit, that is a substantial reduction.

16.4.5 Adder/Subtractor

Let's return to the design of the full adder. A full (one-bit) adder can be implemented out of logic gates by implementing the equations for C and S. As we have seen, the simplified version for C is:

$$C_{out} = AB + AC_{in} + BC_{in}$$

TABLE 16.8 Truth Table for 3-Input XOR

Inputs			Outputs	
A	B	C	$A \oplus B$	$A \oplus B \oplus C$
0	0	0	0	0
0	0	1	0	1
0	1	0	1	1
0	1	1	1	0
1	0	0	1	1
1	0	1	0	0
1	1	0	0	0
1	1	1	0	1

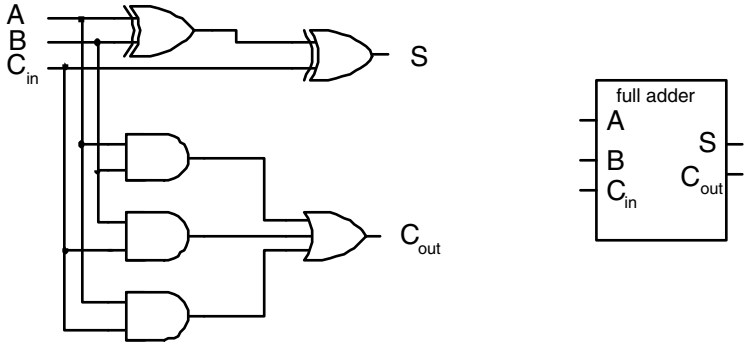


FIGURE 16.15 Implementation of full adder from logic gates on the left. Symbol of full adder on the right.

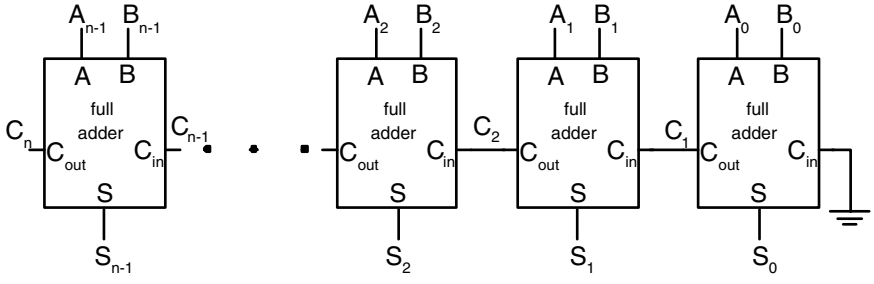


FIGURE 16.16 Implementation of an n-bit adder from full adders.

S cannot be simplified using K-maps. Instead, we will simplify S by inspection of the truth table for the full-adder given in Table 16.6. Note that S is high when exactly one of the three inputs is high, or when all the inputs are high. This is the same functionality as a 3-input XOR gate, as shown in Table 16.8. Thus we can implement S with the equation:

$$S = A \oplus B \oplus C_{in}$$

This completes our design of a full adder. Its implementation in logic gates is shown in Figure 16.15.

We would like to add n-bit binary numbers. To accomplish this we will connect N full adders as shown in Figure 16.16. This configuration connects the carry-out of one bit to the carry-in of the next bit, and is called a ripple carry adder. This design is small but slow because the carry must ripple through from the least significant bit to the most significant bit. For designs where speed is of the essence, such as modern high-speed microprocessors, various techniques are used to speed up the calculation of the carry chain. Such techniques are beyond the scope of this chapter.

TABLE 16.9 Binary Representation of Negative Numbers

Decimal	Sign-Magnitude	1's Complement	2's Complement
7	0111	0111	0111
6	0110	0110	0110
5	0101	0101	0101
4	0100	0100	0100
3	0011	0011	0011
2	0010	0010	0010
1	0001	0001	0001
0	0000	0000	0000
−0	1000	1111	
−1	1001	1110	1111
−2	1010	1101	1110
−3	1011	1100	1101
−4	1100	1011	1100
−5	1101	1010	1011
−6	1110	1001	1010
−7	1111	1000	1001
−8			1000

16.4.6 Representing Negative Binary Numbers

We would like to add negative numbers as well as positive numbers, and we would like to subtract numbers as well as add them. To do this we need a way of representing negative numbers in base 2. Two common methods are used: *sign-magnitude* and *2's complement*. A third method, *1's complement* will also be described to aid in the explanation of 2's complement.

In Table 16.9, four bits are used to represent the values 7 to 0. The three different methods, for representing negative numbers are shown. Note that for *all* three methods, the positive numbers have the same representation. Also, the leftmost bit is always the sign bit. It is 0 for a positive number and 1 for a negative number. It is important to note that all these representations are different ways that a *human* interprets the bit patterns. The bit patterns are not what is changing; the interpretations are. Given a bit pattern, you cannot tell which system is being used unless someone tells you. This discussion can be extended to numbers represented with any number of bits.

The sign-magnitude method is the closest to the method we use for representing positive and negative numbers in decimal. The sign bit indicates whether it is positive or negative, and the remaining bits represent the magnitude or value of the number. So, for example, to get the binary value of negative 3, you take the positive value 0011 and flip the sign bit to get 1011. While this is the most intuitive for humans to understand, it is not the easiest representation for computers to manipulate. This is true for several reasons. One is that there are two representations for 0: +0 and −0. A very common operation to perform is to compare a number to 0. With two representations of zero, this operation becomes more complex.

Instead, a representation called 2's complement is more frequently used. 2's complement has the feature that it has only one representation for zero. It has other advantages as well, including the fact that addition and subtraction are straightforward to implement; subtraction is the same as adding a negative number. If you add a number and its complement, the result is zero with no carry-out, as one would expect. To form the negative value of a positive number in 2's complement, simply invert all the bits and add 1. Inverting the bits results in the 1's complement, as shown in Table 16.7. 1's complement has many of the advantages of 2's complement, except that it still has two representations of zero. The 2's complement is formed by adding 1 to the 1's complement of a number. 3 is 0011. Its one's complement is 1100 and its 2's complement is 1101. Given a negative number, how can I tell its value? Due to properties of 2's complement numbers, if I take the 2's complement of a negative number, I get its positive value. Given 1101, I invert the bits to get 0010 and then add 1 to get 0011, its positive value 3. Note that the 2's complement representation has one representation for zero, and it is the value represented by all zeros. Because I can represent 16 values

TABLE 16.10 Choosing the B Input for an Adder/Subtractor

SB	B_i	Result
0	0	0
0	1	1
1	0	1
1	1	0

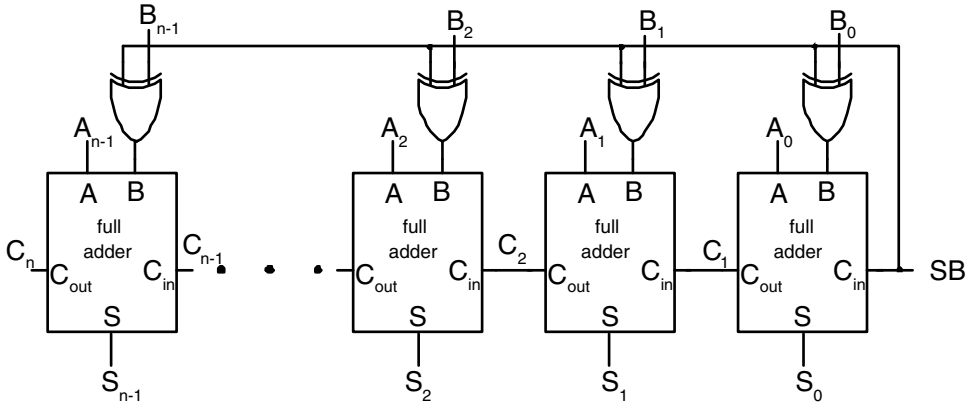


FIGURE 16.17 Connection of n full adders to form an N-bit ripple-carry adder/subtractor. At the rightmost adder, the subtract line (SB) is connected to Cin_0.

with 4 bits, this leaves me with a non-symmetric range. In other words, I can represent one more negative number 1000 than positive number. This number is -8. Its positive value cannot be represented in 4 bits. What happens if I take the 2's complement of -8 ? The 1's complement is 0111. When I add 1 to form the 2's complement, I get 1000. What is really happening is that the true value, $+8$, cannot be represented in the number of bits I have available. It overflows the range for the representation I am using.

Representing numbers in 2's complement makes it easy to do subtraction. To subtract two n-bit numbers $A - B$, you simply invert the bits of B and add 1 when you are summing $A + \overline{B}$.

We are now ready to expand our n-bit ripple carry adder to an n-bit adder/subtractor. Just as we do addition one digit at a time, the adder circuit handles two input bits, A_i and B_i, plus a carry-in C_{in_i}. We can arrange as many of these circuits in parallel as we have bits. The ith circuit gets the ith bits of two operands plus the carry-out of the previous stage. It is straightforward to modify the full-adder to be a one-bit adder/subtractor. A one-bit adder/subtractor performs the following tasks:

1. Choose B_i or the complement of B_i as the B input.
2. Form the sum of the three input bits, $S_i = A_i + B_i + C_{in_i}$.
3. Form the carry-out of the three bits, $C_{out_i} = f(A_i, B_i, C_{in_i})$.

On a bit-by-bit basis, the complement of B_i is just $\overline{B_i}$. For an n-bit number, the 2's complement is formed by taking the bit-by-bit complement and adding 1. I can add 1 to an n-bit subtraction by setting the carry-in bit C_{in_0} to 1. In other words, I want $C_{in_0} = 1$ when subtract is true, and $C_{in_0} = 0$ when subtract is false. This is accomplished by connecting the control signal for subtracting (SB) to C_{in_0}. Similarly when SB = 0, I want to use B_i as the input to my full adder. When SB = 1, I want to use $\overline{B_i}$ as input. This is summarized in Table 16.10. By inspection, the desired B input bit to the ith full adder is the XOR of SB and B_i. If I put all the components together — n full adders, the carry-in of the LSB set to SB, and the XOR of SB and B_i to form the complement of B — I get an n-bit adder/subtractor as shown in Figure 16.17.

TABLE 16.11 Truth Table for a 4-to-2 Encoder

D_0	D_1	D_2	D_3	Q_1	Q_0	V
0	0	0	0	0	0	0
1	0	0	0	0	0	1
0	1	0	0	0	1	1
0	0	1	0	1	0	1
0	0	0	1	1	1	1

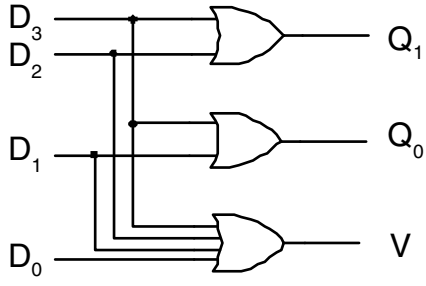


FIGURE 16.18 A 4-to-2 encoder with outputs Q_0 and Q_1 and valid signal.

16.5 Frequently Used Digital Components

Many components such as full adders, half adders, 4-bit adders, 8-bit adders, etc., are used over and over again in digital logic design. These are usually stored in a design library to be used by designers. In some libraries these components are parameterized. For example, a generator for creating an n-bit adder may be stored. When the designer wants a 6-bit adder, he or she must instantiate the specific bit width for the component.

Many other, more complex components are stored in these libraries as well. This allows components to be designed efficiently once and reused many times. These include encoders, multiplexers, demultiplexers, and decoders. Such designs are described in more detail below. Later, we will use them in the design of a calculator datapath.

16.5.1 Elementary Digital Devices: ENC, DEC, MUX, DEMUX

16.5.1.1 ENC

An ENCODER circuit has 2^n input lines and n output lines. The output is the number of the input line that is asserted. Such a circuit *encodes* the asserted line. The truth table of a 4-to-2 encoder is shown in Table 16.11. The inputs are D_0, D_1, D_2, and D_3, and the outputs are Q_0 and Q_1. Note that we assume at most one input can be high at any given time. More complicated encoders, such as priority encoders, that allow more than one input to be high at a time, are described below.

An encoder, like all other components, is built out of basic logic gates. The equations for the Q outputs can be determined by inspection. Q_0 is 1 if D_1 is 1 or D_3 is 1. Q_1 is 1 if D_2 is 1 or D_3 is 1.

Note that there is no difference in the outputs of this circuit if *no* input is asserted, or if input D_0 is asserted. To distinguish between these two cases, an output that indicates the Q outputs are valid, V is added. V is 1 if any of the inputs are 1, and 0 otherwise. V is considered a control output rather than a data output. A logic diagram of an encoder is shown in Figure 16.18.

16.5.1.2 DEC

A decoder performs the opposite function of an encoder. Exactly one of the outputs is true if the circuit is enabled. That output is the *decoded* value of the input, if I think of the input as a binary number. When

TABLE 16.12 Truth Table for a 2-to-4 Decoder with Enable

EN	A	B	Q_0	Q_1	Q_2	Q_3
0	X	X	0	0	0	0
1	0	0	1	0	0	0
1	1	0	0	1	0	0
1	0	1	0	0	1	0
1	1	1	0	0	0	1

the enable input (*EN*) is 0, the circuit is disabled, and all outputs are 0. The truth table of a 2-to-4 decoder is given in Table 16.12.

Note the use of X values for inputs in the first line of the truth table. Here, X stands for "don't care." In other words, I don't care what the value of the inputs A and B are. If $EN = 0$, the outputs will always have the value 0. I am using the *don't care* symbol, X, as a shorthand for the four combinations of input values for A and B. We have already used *don't cares* for minimizing circuits with K-maps above. In that case, the *don't cares* were "output" *don't cares*. The *don't cares* in the truth table for the encoder are "input" *don't cares*. They are shorthand for several combinations of inputs.

16.5.1.3 MUX

Many systems have multiple inputs which are handled one at a time. *Call waiting* is an example. You are talking on one connection when a clicking noise signals that someone else is calling. You switch to the other, talk briefly, and then switch back. You can toggle between calls as often as you like. Because you are using one phone to talk on two different circuits, you need a **multiplexer** or MUX to choose between the two. There is also an inverse MUX gate called either a DEMUX or a DECODER. Again, a telephone example is the selection of an available line among, say, eight lines between two exchanges. That is, you have one line in and eight possible out, but only one output line is connected at any time. An algorithm based on which lines are currently free determines the choice. Let us design these two devices, beginning with a 2-to-1 MUX.

What we want in a two-input MUX is a circuit with one output. The value of that output should be the same as the input that we select. We will call the two inputs A and B and the output Q. The select input S chooses which input to steer to the output.

Logically I can think of Q being equal to A when $S = 0$, and Q being equal to B when $S = 1$. I can write this as the Boolean equation: $Q = (A \cdot \overline{S}) + (B \cdot S)$. You can use a truth table to convince yourself that this equation captures the behavior of a two-input MUX. Note that we now have a new dichotomy of inputs. We call some of them *inputs* and the others *controls*. They are not inherently different, but from the human perspective, we would like to separate them. In logic circuit drawings, *inputs* come in from the left and outputs go out to the right. *Controls* are brought in from top or bottom. The select input for our multiplexer is a control input. Note that, although I talk about S being a control, in the logic equation it is treated the same as an input. An enable is another kind of control input. A *valid* signal can be viewed as a control output. A realization of a 2-to-1 multiplexer with enable is shown in Figure 16.19.

The 2-to-1 MUX circuit is quite useful and is found in many design libraries. Other similar circuits are the 4-to-1 MUX and the 8-to-1 MUX. In general, you can design n-to-1 MUX circuits, where n is a power of 2. The number of select bits needed for an n-to-1 MUX is $log_2(n)$.

Figure 16.20 shows a 4-to-1 multiplexer on the left, and a 1-to-4 demultiplexer on the right. The MUX chooses one of four inputs using the two select lines: $S1$ and $S0$. If we view the values on the select lines as the binary numbers $0, \ldots, 3$, we understand the selection process as enabling the top AND when the input is 00 and then progressively lower ANDs as the numbers become 01, 10, and 11. Essentially, there is a decoder circuit within the 4-to-1 multiplexer.

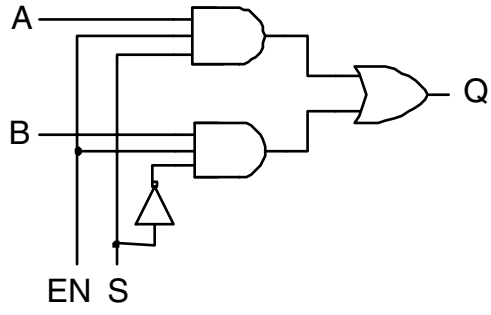


FIGURE 16.19 A 2-to-1 MUX with enable. If the enable is asserted, this circuit delivers at its output, *Q*, the value of *A* or the value of *B*, depending on the value of *S*. In this sense, the output is "connected" to one of the input lines. If the enable is not asserted, the output *Q* is low.

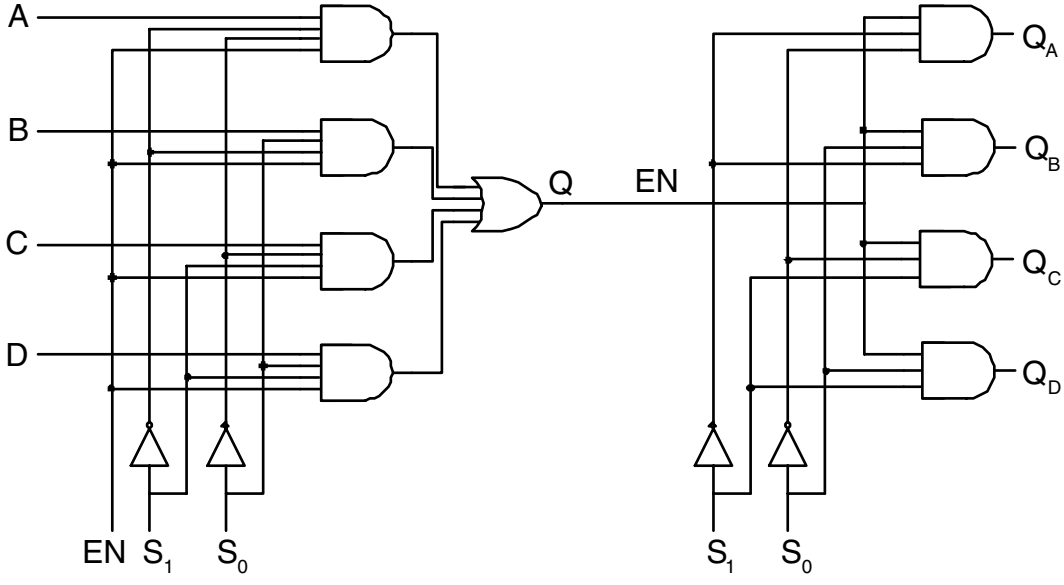


FIGURE 16.20 A 4-to-1 MUX feeding a 1-to-4 DEMUX. The value on MUX select lines S1:S0 determines the input connected to *Q*. *EN*, in turn, is connected to the output of choice by S1:S0 on the DEMUX.

16.5.1.4 DEMUX/DECODER

The inverse circuit to a multiplexer is a demultiplexer, or DEMUX. A DEMUX has one line in, which it switches to one of n possible output lines. A 1-to-4 DEMUX, used in conjunction with the 4-to-1 MUX, is shown on the right in Figure 16.20.

Note that the *Q* output line on the multiplexer is labeled as the *EN* input line on the DEMUX. If I treat this line as an *enable*, the DEMUX becomes a DECODER, in the sense that, when *EN* is asserted, one and only one of the four outputs is asserted, that being the output selected by the number on S1:S0. So a decoder and a demux are the same circuit. You usually do not find a demux in a design library. Rather, what you find is called a decoder, or sometimes a demux/decoder.

Decoding is an essential function in many places in computer design. For example, random-access memory (RAM) is fed an address — a number — and must return data based on that number. It does this by decoding the address to assert lines that enable the output of the selected data. Similarly, computer

TABLE 16.13 Truth Table for a 4-to-2 Priority Encoder

D_0	D_1	D_2	D_3	Q_0	Q_1	V
0	0	0	0	0	0	0
1	0	0	0	0	0	1
X	1	0	0	1	0	1
X	X	1	0	0	1	1
X	X	X	1	1	1	1

TABLE 16.14 ALU Instructions for Calculator

I_1	I_0	Result
0	0	A AND B
0	1	A OR B
1	0	$A + B$
1	1	$A - B$

instructions are numbers that must be decoded to assert the lines which enable the specific hardware that each instruction requires.

16.5.1.5 Priority Encoder

The encoder we started this section with assumed that exactly one input was asserted at any given time. An encoder that could deal with more than one asserted input would be even more useful, but how would we define the output if more than one line were asserted? One simple choice is to have the encoder deliver the value of the highest-ranking line that is asserted. Thus, it is a **priority encoder**.

The truth table for the priority encoder is given in Table 16.13. This truth table has a lot of similarities to the simple encoder we started this section with. The valid output V tells us if any input is asserted. The output Q_0 is true if the only input asserted is D_1. The circuit differs in that more than one input may be asserted. In this case, the output encodes the value of the highest input that is asserted. So, for example, if D_0 and D_1 are both asserted, the output Q_0 is asserted. I don't care if the D_0 input is asserted or not, because the D_1 input has higher priority. Here, once again, the *don't cares* are used as shorthand to cover several different inputs. If I listed all possible combinations of inputs in the truth table, my truth table would have $2^4 = 16$ lines. Using *don't cares* makes the truth table more compact and readable.

16.5.2 The Calculator Arithmetic and Logical Unit

Let's look at putting some of these components together to do useful work. The core of a calculator or microprocessor is its Arithmetic and Logic Unit (ALU). This is the part of the calculator that implements the arithmetic functions, such as addition, subtraction, and multiplication. In addition, logic functions are also implemented such as ANDing inputs and ORing inputs. A microprocessor may have several, sophisticated ALUs. We will examine the design of a very simple ALU for a 4-bit calculator. Our ALU will perform four different operations: AND, OR, addition, and subtraction on two 4-bit inputs, A and B. Two input control signals, I_1 and I_0, will be used to choose between the operations, as shown in Table 16.14. We will call the 4-bit result R.

A common way to implement such an ALU is to implement the various functions in parallel, then choose the requested result based on the setting of the control inputs. The 4-bit AND requires four AND gates, one for each bit position. Similarly, the 4-bit OR requires four OR gates. We will implement the adder and subtractor using a single adder/subtractor unit, since using two would waste area. The adder/subtractor unit in Figure 16.17 is perfect for our purposes here. Note, in Table 16.14, that I_0 is 0 for adding A and B and 1 for subtracting A and B, so we can use I_0 for the SB input to the adder/subtractor. These units will always operate on the A and B inputs; a multiplexer will select the correct output based on the values of I_1 and I_0. Four 4-to-1 multiplexers are used, one for each output bit. 4-to-1 multiplexers are used because

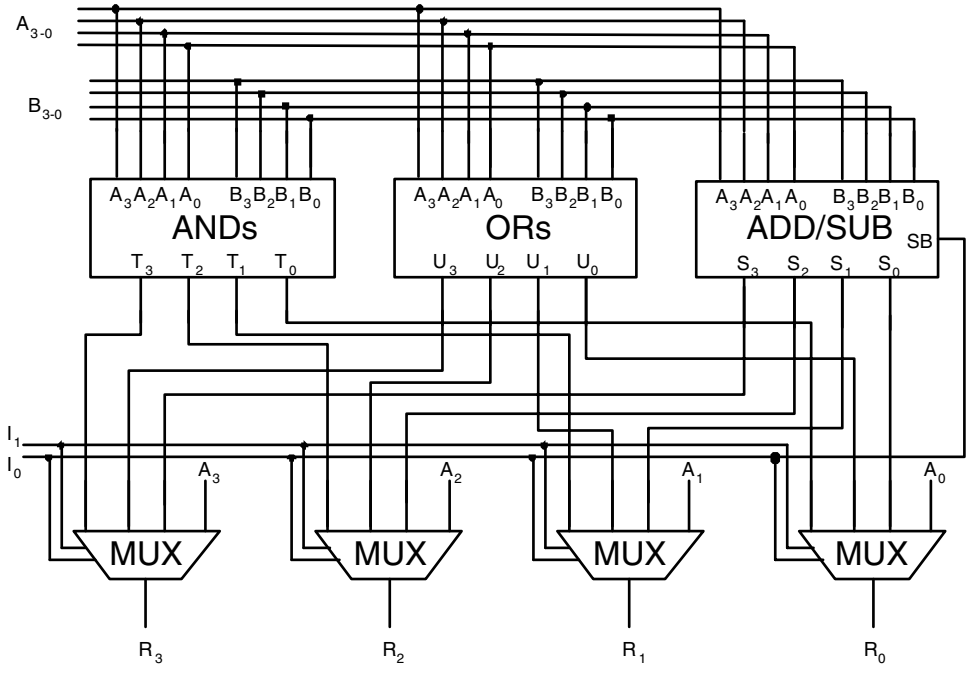


FIGURE 16.21 Implementation of an ALU from other components.

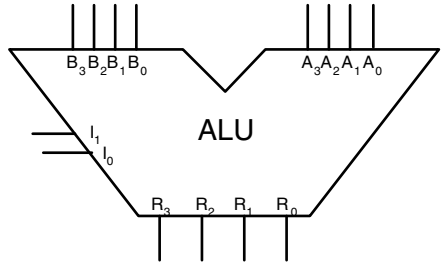


FIGURE 16.22 Symbol of an ALU component.

there are no 3-to-1 muxes. We will use the fourth input to pass the *A* input to the output *R*. The reason for doing this will become apparent when we use the ALU in a calculator datapath. To keep the diagram readable, we use the convention that signals with the same name are wired together. The resulting ALU implementation is shown in Figure 16.21. The symbol for this ALU is shown in Figure 16.22. We will use this symbol when we incorporate the ALU into a calculator datapath.

16.6 Sequential Circuits

16.6.1 Concept of a Sequential Device

So far, all the circuits we have discussed have been combinational. The current output can be determined by knowing the current inputs. Sequential circuits differ from combinational circuits because they have memory. For a circuit with memory, the current outputs depend on the current inputs *and* on the past history of the circuit. Memory elements dramatically change the way a circuit operates.

One of the oldest and most familiar sequential devices is a clock. In its mechanical implementation, ticks from a mechanical oscillator — pendulum or spring and balance wheel — are tallied in a complex, base-12 counter. Typically, the counter recycles every 12 hours. All the states are specifiable, and they form an orderly sequence. Except during transitions from one state to its successor, the clock is always in a discrete state. To be in a discrete state requires some form of memory. I can only know the current output of my clock if I know what its previous output was.

One of the most ubiquitous and essential memory elements in the digital world is the latch, or flip-flop. It snaps from one position to the other (storing a 1 or storing a 0) and retains memory of its current position. We shall see how to build such a latch out of logic gates.

Like clocks, computers and calculators are *finite-state machines*. All of the states of a computer can be enumerated. Saying this does not in any way restrict what you can compute anymore than saying you can completely describe the states of a clock limits the life of the universe. The states of a finite-state machine capture the history of the behavior of the circuit up to the current state.

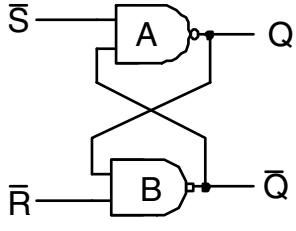
By linking memory elements together, we can build predictable sequential machines that do important and interesting tasks. Only the electronic "latch" and the datapath of our simple calculator are included in this short chapter; but from the sequential elements presented here, complex machines can be built. There are two kinds of sequential circuits, called *clocked* or synchronous circuits and *asynchronous* circuits. The clocked circuits are built from components such as the flip-flop, which are synchronized to a common clock signal. In asynchronous circuits, the "memory" is the intrinsic delay between input and output. To maintain an orderly sequence of events, they depend on knowing precisely how long it takes for a signal to get from input to output. Although that sounds difficult to manage in a very complex device, it turns out that keeping a common clock synchronized over a large and complex circuit is nontrivial as well. We will limit our discussion to clocked sequential circuits. They are more common, but as computer speeds become faster, the asynchronous approach is receiving greater attention.

16.6.2 The Data Flip-Flop and the Register

16.6.2.1 The *SR* Latch: Set, Reset, Hold, and Muddle

In all the circuits we have looked at so far, there was a clear distinction between inputs and outputs. Now we will erase this distinction by introducing positive feedback; we will *feed back* the outputs of a circuit to the inputs of the same circuit. In an electronic circuit, positive feedback can be used to force the circuit into a "stable state." Because saturated logic goes into such states quite normally, it is a very small step to generate an electronic latching circuit from a pair of NAND or NOR gates. The simplest such circuit is shown in Figure 16.23.

Analyzing Figure 16.23 requires walking through the behavior of the circuit. Let's assume that Q has the value 1 and $\overline{Q}$ has the value 0. Start with both $\overline{S}$ and $\overline{R}$ deasserted. In other words, both have value 1 because they are active low signals. The inputs to B will be high, so $\overline{Q}$ will be low. This is a "steady state"



A	B	NAND
0	0	1
0	1	1
1	0	1
1	1	0

FIGURE 16.23 The basic set/reset (SR) latch is shown on the left. If $\overline{S}$ is asserted, Q is asserted (set). If $\overline{R}$ is asserted, Q is deasserted (reset). If neither $\overline{S}$ nor $\overline{R}$ is asserted (both high), the latch retains its current state. If both are asserted, the latch goes into a *muddle* state where Q is asserted and $\overline{Q}$ is deasserted (both high); but upon simultaneous release of the inputs, the next state is unpredictable. The truth table for an NAND gate is shown on the right.

of this circuit; the circuit will stay in this state for some time. This state is called "storing 1," or sometimes just "1" because Q has the value 1. You could toggle $\overline{S}$ (i.e., change its value to 0 and then back to 1) and no other change would take place in the circuit.

Now, with $\overline{S}$ high, let's assert $\overline{R}$ by setting it to 0. First, $\overline{Q}$ will go high because $\overline{R}$ is one of the inputs to B, and the NAND of 0 with anything is 1. This makes both of the inputs to A high, so Q goes low. Now the upper input to B is low, so deasserting $\overline{R}$ (setting it to 1) will have no effect. Thus, asserting $\overline{R}$ has reset the latch. The latch is in the other steady state, "storing 0" or "0."

At this point, asserting $\overline{S}$ will set the latch, or put it back into the state "1." For this reason, the S input is the "set" input to the latch, and the R input is the "reset" input.

What happens if both $\overline{S}$ and $\overline{R}$ are asserted at the same time? The initial result is to have both Q and $\overline{Q}$ go high simultaneously. Now, deassert both inputs simultaneously. What happens? You cannot tell. It may go into either the set or the reset state. Occasionally, the circuit may even oscillate, although this behavior is rare. For this reason, it is usually understood that the designer is *not allowed* to assert both $\overline{S}$ and $\overline{R}$ at the same time. This means that, if the designer asserts both $\overline{S}$ and $\overline{R}$ at the same time, the future behavior of the circuit cannot be guaranteed, until it is set or reset again into a known state.

There is another problem with this circuit. To hold its value, both $\overline{S}$ and $\overline{R}$ must be continuously deasserted. Glitches and other noise in a circuit might cause the state to flip when it should not.

With a little extra logic, we can improve upon this basic latch to build circuits less likely to go into an unknown state, oscillate, or switch inadvertently. These better designs eliminate the muddle state.

16.6.2.2 The Transparent *D*-Latch

A simple way to avoid having someone press two buttons at once is to provide them with a toggle switch. You can push it only one way at one time. We can also provide a single line to enable the latch. This enable control signal is usually called the **clock**. We will modify the SR latch above. First we will combine the S and R inputs into one input called the data, or D input. When the D line is a 1, we will set the latch. When the D line is a 0, we will reset the latch. Second, we will add a clock signal (CLK) to control when the latch updates. With the addition of two NANDs and an inverter, we can accomplish both purposes, as shown in Figure 16.24.

Note that we tie the data line, D, to the top NAND gate, and the inverse or $\overline{D}$ to the bottom NAND gate. This assures that only one of the two NAND outputs can be low at one time. The CLK signal allows us to open the latch (let data through) or latch the data at will. This device is called a transparent D-latch, and is found in many digital design libraries. This latch is called *transparent* because the current value of D appears at Q if the CLK signal is high. If CLK is low, then the latch retains the last value D had when CLK was high.

Has this device solved all the problems we described for the *SR* latch? No. Consider what might happen if D changes from low to high just as the clock changes from high to low. For the brief period before the change has propagated through the D-inverter, both NANDs see both inputs high. Thus, at least briefly,

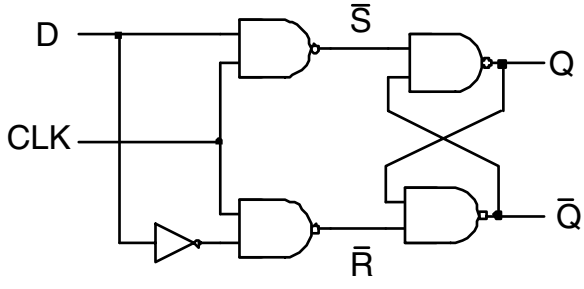


FIGURE 16.24 The transparent D-latch. The circuit is transparent when CLK is high (that is, the current value D appears at Q) and latched when CLK is low (the value of D when the clock went low is held at Q.)

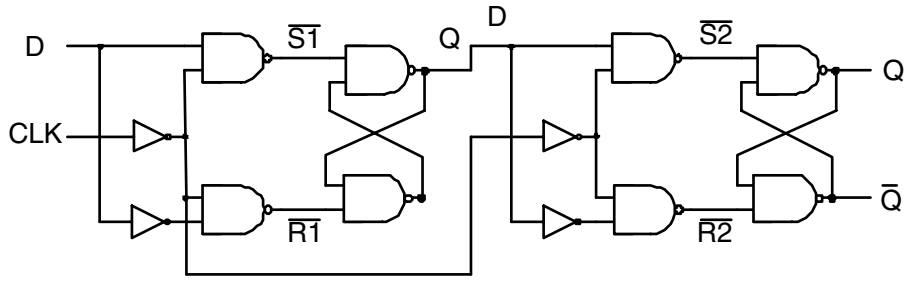


FIGURE 16.25 The master–slave data flip-flop constructed of two D-latches in series.

both $\overline{R}$ and $\overline{S}$ are asserted. This is the very situation we wanted to avoid. This muddle situation would last only for the propagation time of the inverter, but then the CLK signal arrives and drives both $\overline{S}$ and $\overline{R}$ high. The latch might oscillate or flip either way. In any case, it will be unpredictable.

There is another problem with this circuit. It is indeed transparent during the high clock signal. This means that Q will mirror D while CLK is high. If D changes rapidly, so will Q. Sometimes you may want transparency. However, frequently you want to be able to guarantee when the output will change and to only allow one transition on the output per clock cycle. In that case, you do not want transparency; what you really want is a different circuit: a flip-flop (FF).

16.6.2.3 Master–Slave DFF to Eliminate Transparency

The problem with transparent gates is not a new one. A solution that first appeared in King Solomon's time (9th century B.C.E.) will work here as well. The Solomonic gate was a series pair of two quite ordinary city gates. They were arranged so that both were never open at the same time. You entered the first and it was shut behind you. While you were stuck between the two gates, a well-armed, suspicious soldier asked your business. Only if you satisfied him was the second gate opened. The solution of putting out-of-phase transparent latches between input and output is certainly one obvious solution to generating a **data flip-flop** (DFF). Such an arrangement of two D-latches is shown in Figure 16.25.

The latch on the left is called the **master**; that on the right is called the **slave**. This master–slave (MS) DFF solves the transparency problem but does nothing to ameliorate the timing problems. While timing problems are not entirely solvable in any FF, accommodating the number of delays in this circuit tends to make the MSFF a slow device and thus a less attractive solution. Why should it be slow? The issue is that to be sure that you do not put either of these devices into a metastable or oscillatory state, you must hold D constant for a relatively long setup time (before the clock edge) and continue it past the clock transition for a sufficient hold time. This accommodation limits the speed with which the whole system can switch.

Can we do better? Yes, not perfect, but better. The device of choice is the edge-triggered DFF. We will not go into the details of the implementation of an edge-triggered flip-flop as it is considerably more complicated than the circuits considered so far. An edge-triggered flip-flop is designed to pass the input datum to the output during a very short amount of time defined by a clock edge. Edge-triggered flip-flops can either be *rising edge* triggered or *falling edge* triggered. A rising edge triggered flip-flop will only change its output on the rising edge of a clock. There is still a setup time and a hold time; a small amount of time right around the clock edge during which the input datum must be stable in order to avoid the flip-flop becoming metastable. The advantage of the edge-triggered design is that there is only one brief moment when any critical changes take place. This improves synchronization and leads to faster circuits.

There are other types of flip-flops as well, but the DFF is widely used and useful for building more complex circuits. We will consider these as we build more complex circuits. Most designs of a positive edge-triggered DFF include two additional **asynchronous** inputs, *preset* and *clear*. Asynchronous inputs cause the output to change independently of the clock input. The D input is a *synchronous* input; changes on the D input are only reflected at the output during a clock edge. An active signal on the preset input

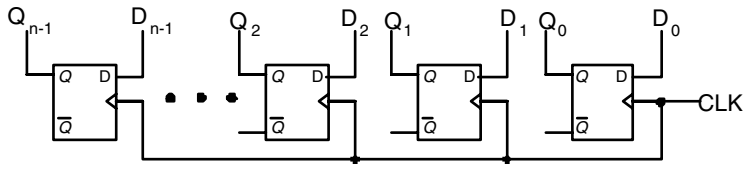


FIGURE 16.26 An *n*-bit data register built from *n* DFFs.

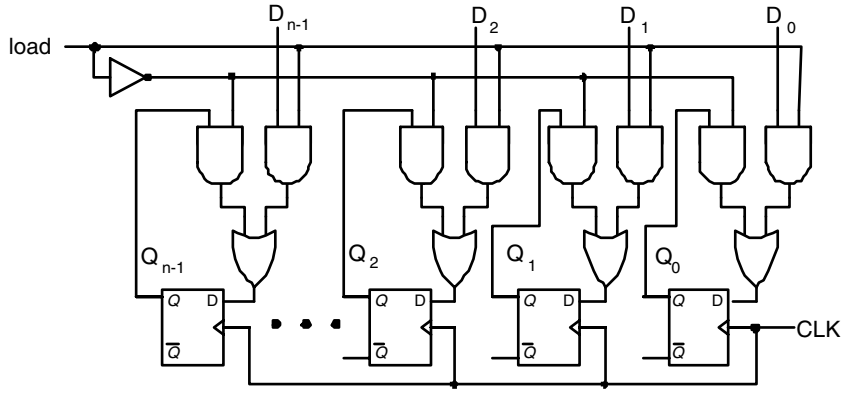


FIGURE 16.27 An *n*-bit shift register with load input. The upper layer is a set of *n* two-input MUXs. The bottom layer is a set of *n* positive-edge-triggered DFFs.

causes the output of the FF to be set to 1; an active signal on the clear input causes the output to be cleared, or set to 0. These inputs are useful for putting flip-flops in a design into a known state.

16.6.3 From DFF to Data Register, Shift Register, and Stack

The simplest and most useful device we can build with DFFs is a data register. A data register stores data. The simplest data register stores new data on every clock edge. Its design is shown in Figure 16.26. It is useful to control when the data register stores new data. We can extend the data register by adding a control signal called *load*. Now our register will load new data only when *load* is high. Otherwise it will store its old data. The new register design is shown in Figure 16.27.

We can also build a *shift* register, which shifts data, one bit at a time, into a parallel register. Shift registers are useful for converting serial data to parallel data. For example, you may receive data one bit at a time from a serial connection to your computer, but want to operate on the data in parallel. Shift registers can sometimes be loaded in parallel also, just like the data register. We will keep things simple; a serial-in parallel-out shift register is shown in Figure 16.28. This shift register also has a serial output.

16.6.3.1 A Stack for Holding Data

Our calculator requires memory to store variables that need to be manipulated. We could implement a random access memory with addressing, read, and write capabilities. Instead, to keep things simple, we will implement a *stack* for storing variables and results. A stack is a group of memory locations with limited access to its contents. At any given time, only the top of the stack can be read. This makes its implementation simple because general addressing does not need to be supported. Our stack supports three operations: push, pop, and hold. When a value is pushed onto the stack, it becomes the value at the top of the stack. All values already in the stack are pushed down by one. If the stack was full *before* the push operation, then the oldest (first in) value on the stack is lost. In a pop operation, the top of the stack is deleted, and all other values move up in the stack by one position. Note that this operation differs from a software *pop*

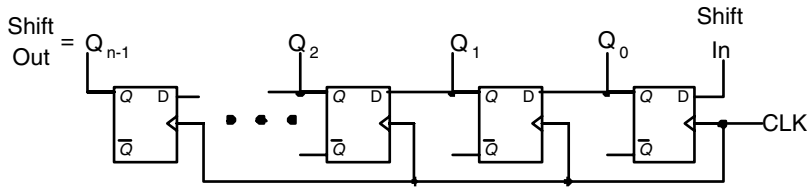


FIGURE 16.28 An *n*-bit shift register with serial input, parallel and serial output. This register shifts one bit to the left every clock cycle.

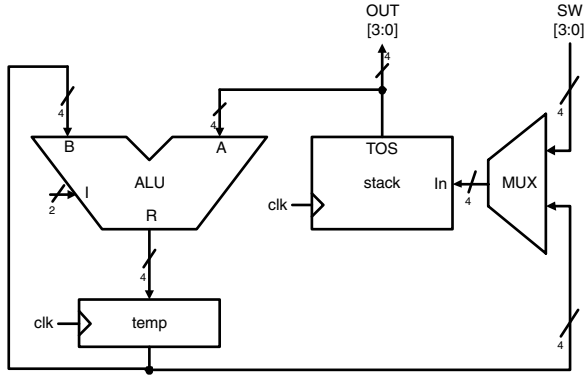


FIGURE 16.29 Calculator datapath. The temp register stores the ALU results every CLK edge.

operation. In software, the popped value is stored in a register. In our hardware implementation, there is no register storing the removed value, so this value is lost. A stack is sometimes called a Last In First Out (LIFO) because that is the order in which values are accessed. The hold operation ensures that the current contents of the stack are retained. It is important to explicitly support this so that the contents of the stack are not changed during operation. Push, pop, and hold all happen on a clock edge. By default, the stack holds its contents when there is no clock edge.

We will implement a stack to hold 4-bit variables. Our stack will contain four locations. One can implement this stack as four shift registers (one for each bit position) with each shift register containing four flip-flops. The total memory contents of our stack is held in 16 DFFs. In a real calculator implementation, the stack would be implemented with memory cells which use fewer transistors and consume less power than DFFs, but for our small design, DFFs will suffice.

16.6.4 Datapath for a 4-bit Calculator

Let's put the pieces together and implement the datapath of a 4-bit calculator. We will use the stack described above as our memory, and the ALU described in the previous section to implement logical operations. We need to add connections and a way to input data values. We also need an *instruction* to tell the calculator datapath what to do. We will use a 7-bit input that can be connected to toggle switches to provide the data and instructions. The datapath for the calculator is shown in Figure 16.29. The output is the top of stack (TOS). This is represented as the four bits `OUT[3:0]`. These could be hooked to a seven-segment display to display the results.

Our calculator will support the instructions: **push**, **pop**, **and**, **or**, **add**, and **sub**tract. Note that the last four are the operations that our ALU supports. These are operations that take two 4-bit input values and produce a 4-bit result. The operands will be found on the stack. The input values will be popped off the stack, and the output value will be pushed onto the top of the stack. For a push operation, the value to be

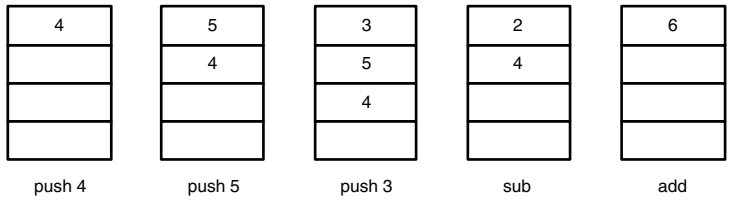


FIGURE 16.30 Stack contents for calculating $4 + (5 - 3)$.

TABLE 16.15 Switch Settings for Calculator Instructions

Instruction	$SW_6 \ldots SW_4$	$SW_3 \ldots SW_0$
AND	000	unused
OR	001	unused
ADD	010	unused
SUB	011	unused
PUSH	101	data
POP	110	unused

pushed onto the stack will be entered via the input switches. Suppose we want to calculate: $4 + (5 - 3)$. The operations required are:

```
PUSH 4
PUSH 5
PUSH 3
SUB
ADD
```

The contents of the stack for this sequence of operations is shown in Figure 16.30.

The format of the instructions and data on the input switches is shown in Table 16.15. Note that only the PUSH instruction uses external data.

What is missing from our calculator design? A controller to make sure that the right operations are performed on the datapath in the correct sequence for each instruction. While the design of such a controller is beyond the scope of this chapter, we will specify its behavior. For each instruction, the datapath goes through a sequence of operations or states. There are a finite number of such states to describe the behavior of the controller, hence the controller is a finite-state machine. Such a finite-state machine can be implemented with flip-flops to hold the state, and combinational logic to generate the next state and the outputs required. The inputs to the state machine are the current state and, for the calculator, the instruction that is entered via the switches. There is also a button the user needs to be able to press that indicates that the next instruction is ready to execute. The outputs are the control signals for the datapath. Our controller needs to output two bits for the ALU to tell it what to do; two bits for the stack to generate the push, pop, or hold inputs, and one bit for the MUX to tell it which input to use. The temporary register requires no control signals — it will be updated every clock cycle. The entire calculator design is shown in Figure 16.31.

Let's look at how the instructions are executed on our calculator. There are three *classes* of instruction: **push**, **pop**, and ALU (**add, sub, and, or**). The **pop** instruction is the simplest. All that is needed is one state to pop the stack. In that state, the control inputs to the stack should be "pop." We don't care what the ALU or MUX control inputs are set to **push** is also a one state instruction. In that state, the data on the switches is pushed onto the stack. So the control input for the MUX should be set to select the input switches and the control input for the stack should be set to "push."

The ALU instructions are more complicated; they require three control states. We assume that our calculator has the two operands on the stack before an ALU instruction is executed. In the first state, the data value

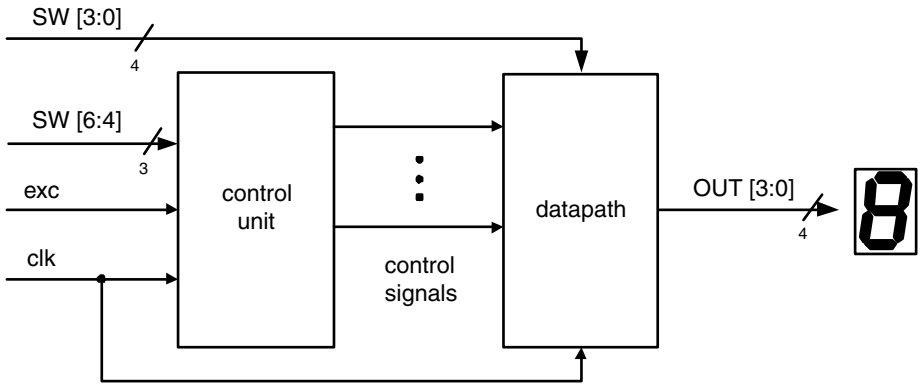


FIGURE 16.31 Connecting the controller, datapath, inputs, and outputs of the calculator.

on the top of the stack is stored in the temp register and the stack is popped. How can we accomplish this in one state? Note that the ALU is a combinational circuit and the TOS is already the A input to the ALU. If we select the ALU operation to pass A through to the output of the ALU, then on the clock edge, we will simultaneously store the current TOS value in the temp register, and pop the value off the stack. Now, at the start of the second state, the first operand is at input B of the ALU because the temp register output is connected to input B. The second operand is at input A of the ALU because the TOS is connected to input A. In this state we will execute the correct ALU instruction, store the result in the temp register, and pop the A operand off the stack. Finally, in the third state the contents of the temp register (i.e., the results of the ALU operation) are pushed onto the stack. At the end of an ALU operation, the two operands have been popped from the stack, and the result pushed on the stack as required. To summarize, the three states of an ALU operation are:

```
state 1:  temp <- operand A;  pop
state 2:  temp <- A op B   ;  pop
state 3:                      push temp
```

We have discussed what happens when an instruction is executing. What happens between instructions? The stack should "hold" its contents. It does not matter what the ALU is doing, or what input the MUX passes through. The memory of this design is in the stack. As long as the stack holds its contents, the memory is maintained. The temporary register is updated every clock cycle, but the results are not saved, so this does not affect the correct operation of the calculator datapath. Note that the output is always active, and it always shows what is currently on the top of the stack.

We have presented basic digital components, both combinational and sequential, and put them together to build a useful design: a simple calculator. Using essentially the same techniques, much more complicated devices can be constructed. Many of the digital devices we take for granted — digital watches, antilock braking systems, microwave oven controllers, etc. — are the implementations of designs using these techniques. In the final section of this chapter, we will consider how such devices are physically realized.

16.7 ASICs and FPGAs — Faster, Cheaper, More Reliable Logic

The circuits we have considered so far — encoders, decoders, muxes, flip-flops, etc. — can be bought in packages with several to a chip. For example, a 14-pin package containing four 2-input NANDs per package has been available for more than 30 years. These chips are called small-scale integrated circuits (SSI). No one who has watched the astonishing decline in the cost of digital electronics brought about by Very Large Scale Integrated circuits (VLSI) would expect to find engineers generating circuits by hooking

up vast arrays of such packages. In a world where powerful computer chips roll off the line with more than 10 million transistors all properly connected and functioning at clock speeds in excess of 1 GHz, why would we be manually hooking up hundreds of these small packages with 16 transistors in a chip that takes 15 to 20 ns to get a signal through a single NAND? Today, the equivalent of many pages of random logic circuit diagrams can be implemented in a single chip. There are many different ways to specify such designs and many different ways to physically realize them. The dominant method of design entry is using a Hardware Description Language (HDL). HDLs resemble software programming languages with added features specifically for describing hardware such as bitwidth, I/O ports, and controller state specifications. Design tools translate HDL descriptions of a circuit to the final implementation. One of the goals of an HDL design is to separate the design from the implementation technology. The same HDL description, in theory at least, can be mapped to different target technologies.

There are also many technologies for physically realizing digital logic designs. Application Specific Integrated Circuits (ASICs) can be implemented as VLSI circuits where millions of transistors are realized on a single chip resulting in very high speeds and very low power dissipation. Such designs are manufactured at a foundry and cannot be changed after they have been implemented. Because high-performance VLSI designs are very expensive to manufacture, they are increasingly used only for very large volume designs and designs where low power is critical. For example, VLSI ASIC chips are found in mobile phones and other handheld devices that meet these criteria.

Designers are increasingly turning to *programmable* and *reconfigurable* devices for realizing their designs. These devices are manufactured in large quantities using VLSI techniques with the latest technology. They are specialized to a particular design after the fabrication process, and hence are programmable. Devices that can be reprogrammed, to fix errors or update functionality, are also called reconfigurable. One of the most popular of these devices is the Field Programmable Gate Array (FPGA). Modern FPGAs can implement designs with the equivalent of millions of transistors on a single chip and operate with clock speeds of several hundred MegaHertz. FPGAs are much more cost-effective than ASICs for many designs, and enjoy an increasing market share in digital hardware products.

For both FPGAs and ASICs, all of the steps that take the initial design to finished chip can be automated. The initial design can be described as a schematic, similar to the diagrams in this chapter, or using an HDL. In the case of FPGAs, design tools translate this specification into programming data that can be downloaded to the chip. As we shall see, FPGA chips are based on memory technology. Rather than downloading a data file to memory, you download a configuration file to an FPGA that changes the way the hardware functions. This *programming* of the chip is very rapid. One can make a change in a complex design and have a working realization in less than an hour. By comparison, ASIC fabrication can take several weeks. By tightening the design cycle, such rapid prototyping has dramatically reduced the cost of designing and producing complex circuits for specific applications.

The underlying technology of an FPGA, and what makes it programmable and reconfigurable, is memory. Writing HDL programs and programming the FPGA makes the design process sound more like software than hardware development. The major difference is that the underlying structures being programmed implement the hardware structures we have been discussing in this chapter. In this section we introduce FPGA technology and explain how digital designs are mapped onto the underlying structures. There are several companies that design and manufacture FPGAs. We will use the architecture of the Xilinx FPGA as an example.

16.7.1 FPGA Architecture

Let's consider the architecture of Xilinx FPGAs. Our objective is not to learn how to program them — a task normally accomplished by software — but rather to show the relationship of these sophisticated chips to the logic we have already developed. The Xilinx chip is made up of three basic building blocks:

1. **CLB.** The configurable logic blocks are where the computation of the user's circuit takes place.
2. **IOB.** The input/output blocks connect I/O pins to the circuitry on the chip.
3. **Interconnect.** Interconnect is essential for wiring between CLBs and from IOBs to CLBs.

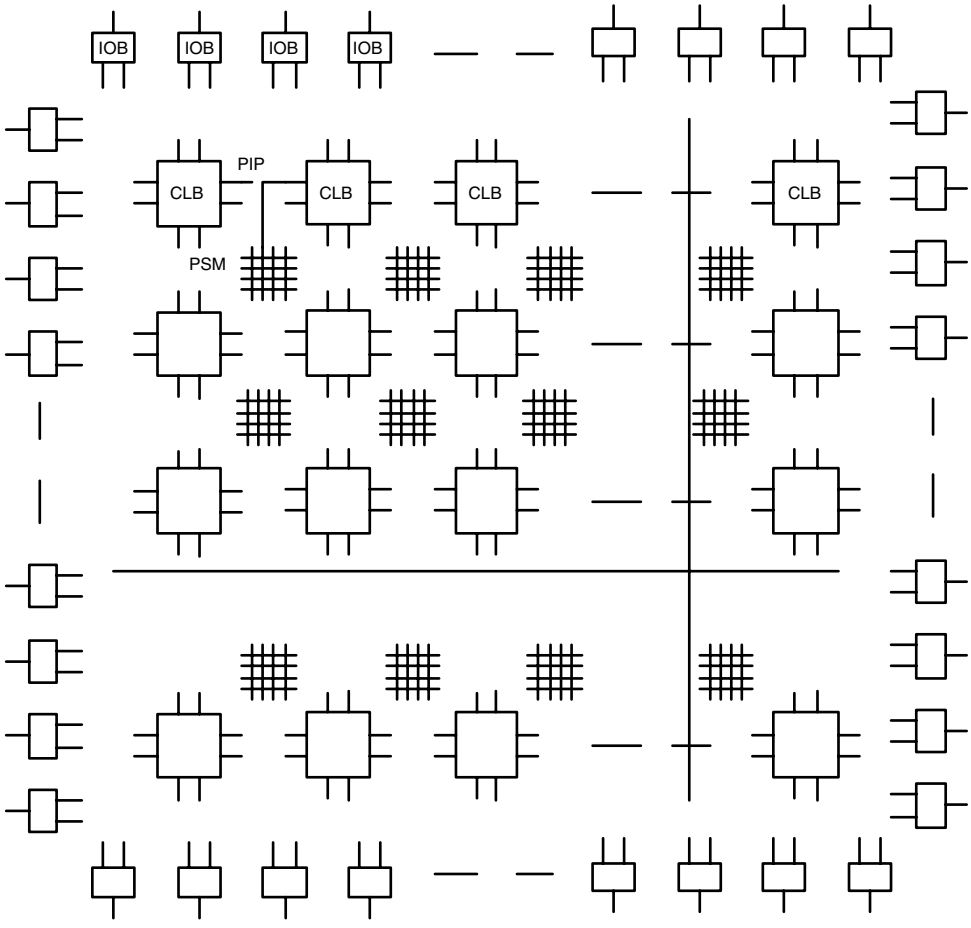


FIGURE 16.32 Overview of the Xilinx FPGA. I/O blocks (IOBs) are connected to pads on the chip, which are connected to the chip-carrier pins. Several different types of interconnect are shown, including Programmable Interconnect Points (PIPs), Programmable Switch Matrices (PSMs), and long line interconnect.

The Xilinx chip is organized with its configurable logic blocks (CLBs) in the middle, its I/O blocks (IOBs) on the periphery, and lots of different types of interconnect. Wiring is essential to support the versatility of the chips to implement different designs efficiently and to ensure that the resources on the FPGA can be utilized efficiently. The overview of a Xilinx chip is presented in Figure 16.32. Each CLB is programmable and can implement combinational or sequential logic or both. Data enter or exit the chip through the IOBs. The interconnect can be programmed so that the desired connections are made. Distributed configuration memory (not shown in the figure) controls the functionality of the CLBs and IOBs, as well as the wiring connections. Implementation of the CLBs, interconnect, and IOBs are described in more detail below.

16.7.1.1 The Configurable Logic Block

How do you use memory to implement different Boolean logic functions? You download the truth table of the function to the memory. Changing the contents of the memory cells changes the functionality of the hardware. One set of memory cells for a one-bit output is referred to as a lookup table (LUT) because you "look up" the result. The basic Xilinx logic slice contains a four-input LUT for realizing combinational logic. The result of this combinational function may or may not be stored in a D flip-flop. The implementation of a logic slice, with four-input LUT and optional flip-flop on the LUT output, is shown on the left in

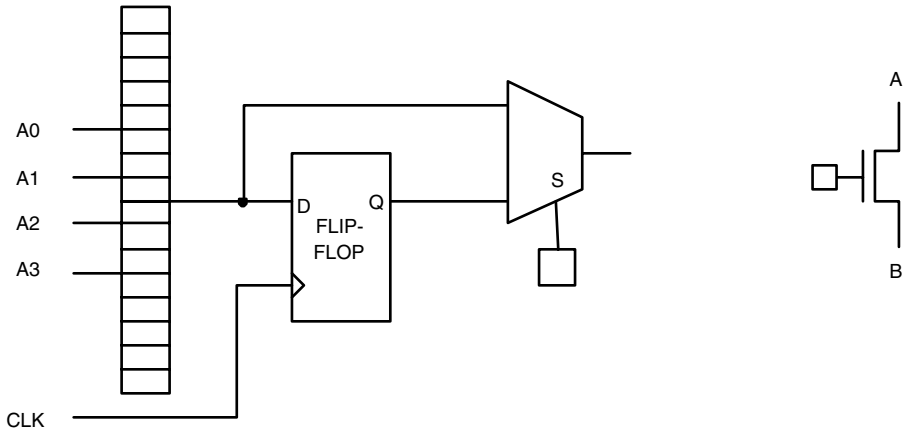


FIGURE 16.33 On the left, a simplified CLB logic slice containing one 4-input lookup table (LUT) and optional DFF. The 16 one-bit memory locations on the left implement the LUT. One additional bit of memory is used to configure the MUX so the output comes either directly from the LUT or from the DFF. On the right is a programmable interconnect point (PIP). LUTs, PIPs, and MUXes are three of the components that make FPGA hardware programmable.

Figure 16.33. Note that the multiplexer can be configured to output the combinational result of the LUT or the result of the LUT after it has been stored in the flip-flop by setting the memory bit attached to the MUX's select line. The logic is configured by downloading 17 bits of memory: the 16 bits in the lookup table and the one select bit for the multiplexer. Using this simple structure, all of the designs described so far in this chapter can be implemented.

The real Xilinx 4000 family CLB is considerably more complicated; however, the way it is programmed is the same. It essentially contains two of the logic slices shown in Figure 16.33. In addition, there is a third LUT whose three inputs can include the outputs of the two 4-input LUTs. The CLB is designed to be able to implement any Boolean function of five input variables, or two Boolean functions each of four input variables. With the third LUT, some functions of up to nine input variables can be realized. Extra logic and routing is also added to speed up the carry chain for an adder, because adders are such common digital components. In addition, additional routing and MUXes allow the flip-flops to be used independent of the LUTs as well as in conjunction with them.

16.7.1.2 Interconnect

Once the CLBs have been configured to implement combinational and sequential logic components, they need to be connected to implement larger circuits, just as we wired together the ALU, register, and MUXes to implement the calculator datapath. This requires programmable interconnect, so that an FPGA can be programmed with different connections depending on the circuit being implemented. The key is the programmable interconnect point (PIP) shown on the right in Figure 16.33. This simple device is a pass transistor with its gate connected to a memory bit. If that memory bit contains a 1, the two ends of the transistor are logically connected; if the memory bit contains a 0, no connection is made. By appropriately loading these memory bits, different wiring connections can be realized. Note that there is considerably more delay across the PIP than across a simple metal wire on a chip. This is the trade-off when using programmable interconnect.

Our FPGA architecture has CLBs arranged in a matrix over the surface of a chip, with routing channels for wiring between the CLBs. Programmable Switch Matrices (PSMs) are implemented at the intersection between a row and column of routing. These switch matrices support multiple connections, including signals on a row and a column, signals passing through on a row, and signals passing through on a column. Figure 16.34 shows a signal output from one CLB connecting to the inputs of two others. This signal passes through two PSMs and three PIPs, one for each CLB connection.

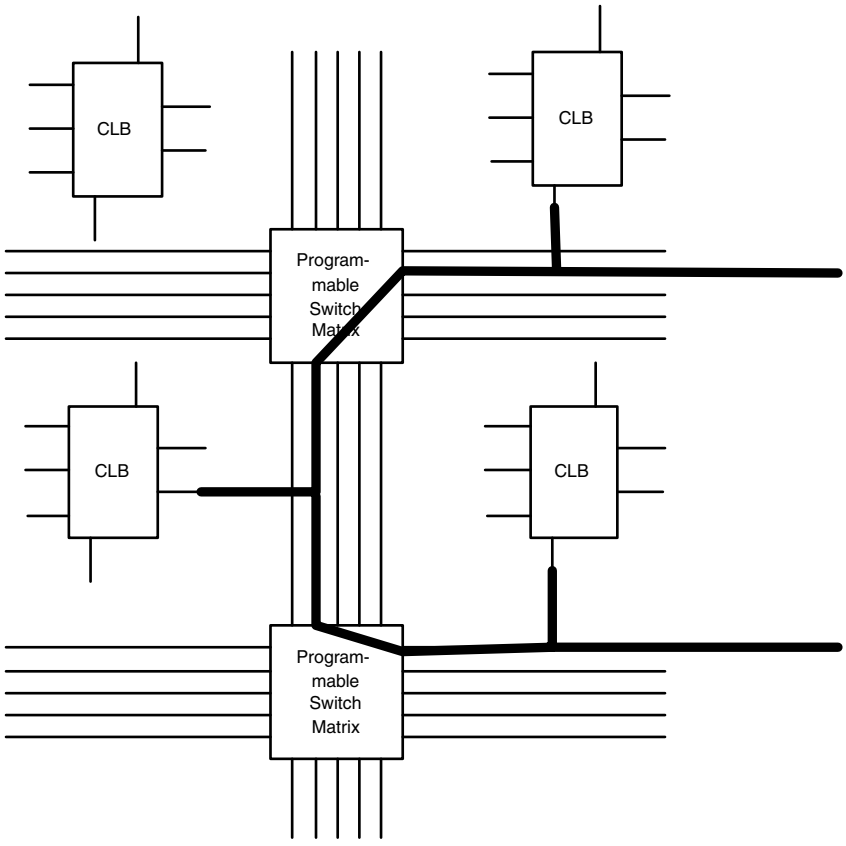


FIGURE 16.34 Programmable interconnect, including two programmable switch matrices (PSMs) for connecting the output of one CLB to the input of two other CLBs.

While programmable interconnect makes the FPGA versatile, each active device in the interconnection fabric slows the signal being routed. For this reason, early FPGA devices, where all the interconnect went through PIPs and PSMs, implemented designs that were considerably slower than their ASIC counterparts. More recent FPGA architectures have recognized the fact that high-speed interconnect is essential to high-performance designs. In addition to PIPs and PSMs, many other types of interconnect have been added. Many architectures have nearest neighbor connections, where wires connect from one CLB to its neighbors without going through a PIP. Lines that skip PSMs have been added. For example, double lines go through every other PSM in a row or a column. Long lines have been added to support signals that span the chip. Special channels for fast carry chains are available. Finally, global lines that transmit clock and reset signals are provided to ensure these signals are propagated with little delay. All of these types of interconnect are provided to support both versatility and performance.

16.7.1.3 The Xilinx Input/Output Block

Finally, we need a way to get signals into and out of the chip. This is done with I/O blocks (IOBs) that can be configured as input blocks, output blocks, or both (but not at the same time). The Output Enable (OE) signal enables the IOB as an output. If the OE signal is high, the output buffer drives its signal out to the I/O pad. If the OE signal is low, the output function is disabled, and the IOB does not interfere with reading the input from the pad. The OE signal can be produced from a CLB, thus allowing the IOB to sometimes be enabled as an output and sometimes not. In addition, IOBs contain DFFs for latching

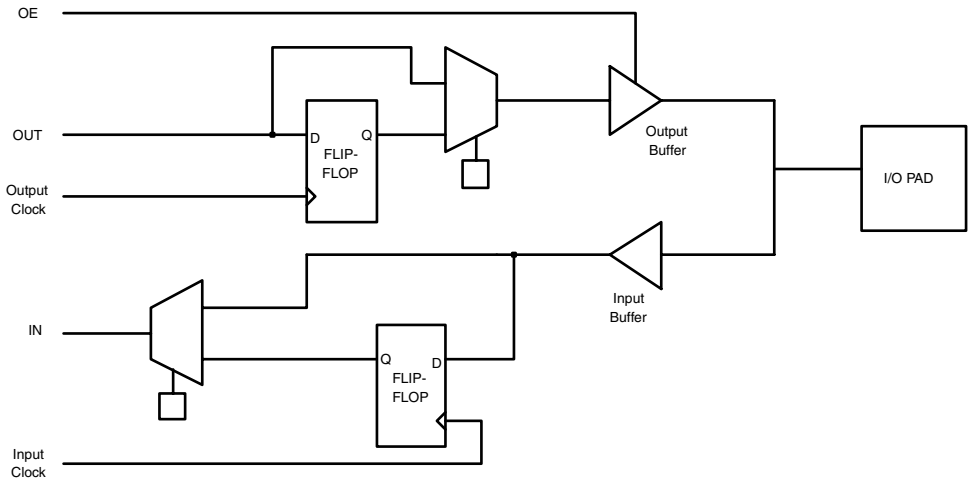


FIGURE 16.35 Simplified version of the IOB. IOBs can be configured to input or output signals to the FPGA. When OE is high, the output buffer is enabled so the output signal is driven on the I/O pad. When OE is low, the IOB functions as an input block. Buffers handle electrical issues with signals from the I/O pad.

the input and output signals. The latches can be bypassed by appropriately programming multiplexers. A simplified version of the IOB is shown in Figure 16.35. The actual IOB contains additional circuitry to properly deal with such electrical issues as voltage and current levels, ringing, and glitches, that are important when interfacing the chip to signals on a circuit board.

CLBs, IOBs, and interconnect form the basic architecture for implementing many different designs in a single FPGA. The configuration memory locations, distributed across the chip, need to be loaded to implement the appropriate design. For a Xilinx FPGA, these memory bits are SRAM, and are loaded on power-up. Special I/O pins that are not user configurable are provided to download the configuration bits that define the design to the FPGA. Other devices use different underlying technologies than SRAM to provide programmability and reconfigurability.

16.7.1.4 Mapping the Simple Calculator to an FPGA

Let's look at how the calculator design (Figure 16.31) is mapped onto a board containing an FPGA. The board used contains a Xilinx 4028E FPGA, switches, push-buttons, and seven-segment displays. The calculator was designed to map to this board, with switches used for entering instructions and data, a push-button for the EXC command, and a seven-segment display used to show the top of stack. The logic of the calculator is mapped to CLBs. The controller is made up of Boolean logic and DFFs to hold the state. The datapath is made up of the components developed in this chapter and mapped to LUTs and DFFs.

This calculator was developed as an undergraduate laboratory experiment. Students enter the design using a schematic capture tool, which involves drawing diagrams like the ones in this chapter. Synthesis tools translate the design to LUTs and flip-flops, breaking the logic up into four-input chunks of logic, each of which is implemented with one truth table. Alternatively, the design can be described using a hardware description language to specify behavior. A different synthesis tool is involved, but the end result is a set of LUTs and DFFs that implement the design. Placement tools map these components to CLBs on the chip, and routing tools route the connections, making use of the various kinds of interconnect available. The tools translate the logic design to a bitstream that is downloaded to the board from a PC through a download cable. The result is a functioning calculator on an FPGA board.

An advantage of this design flow is that designers can migrate their designs to the newest chip architecture without changing the specification. Only the tools need to change to target a faster or cheaper device.

16.7.2 Higher Levels of Complexity

Integrating functionality on a single chip allows for higher performance and smaller packages. As more and more transistors can be realized on a single chip, and functionality increases, it also has become increasingly clear that one particular structure for implementing a design does not suit all needs. While many digital designs can be implemented using FPGA structures, others are less well suited to this technology. For example, hardware multipliers are particularly inefficient when mapped to CLBs. Certain functions perform better on a microprocessor or a programmable digital signal processor (DSP) than in digital hardware. For this reason, FPGA manufacturers have begun integrating large functional blocks on FPGAs. For example, both Xilinx and Altera, two of the major FPGA manufacturers, have introduced FPGAs with embedded multipliers, embedded RAM blocks, and embedded processors. Altera calls this approach "System on a Programmable Chip." Similarly, to support reconfigurability after manufacturing, ASIC designers are increasingly adding blocks of FPGA logic to their designs. It is clear that the future will bring more complex chips with more functionality, higher clock speeds, and more types of logic integrated on a single chip. Digital logic and reconfigurable hardware will be part of these designs for the foreseeable future.

Acknowledgment

The author acknowledges the significant contribution of James Feldman, author of the original version of the chapter, to the current organization and content.

Defining Terms

Active logic: Digital logic that operates all of the time in the active, dissipative region of the electronic amplifiers from which it is constructed. Such logic is generally faster than saturated logic, but it dissipates much more energy. *See* **saturated logic**.

ASIC: Application-specific integrated circuit. Integrated circuits that are designed for a specific application. The term is used to describe VLSI circuits that can be configured before manufacturing to meet the specific needs of the application. High-performance ASICs are low power and high speed, but also expensive.

CLB: Configurable logic block in a Xilinx FPGA. This is where the computation occurs in an FPGA. CLBs implement truth tables and D flip-flops.

Clock: The input that provides the timing signal for a circuit. In general, the oscillator circuit that generates a synchronization signal.

CMOS: Complementary metal–oxide–semiconductor. The dominant family of binary, saturated logic. CMOS circuits are built from MOS transistors in complementary pairs. When one transistor is open, its complement is closed, ensuring that no current flows through the switch itself. Such a configuration has minimum power dissipation in any static state.

Combinational circuit: A logic circuit whose output is a function only of its inputs. Apart from propagation delays, the output always represents a logical combination of its present inputs. *See* **sequential circuit**.

Critical race: In a sequential circuit, a situation in which the "next state" is determined by which of two (or more) internal gates is first to reach saturated state. Such a race is dependent on minor variations in circuit parameters and on temperature, making the circuit unpredictable and possibly even unstable.

Decoder: A logic circuit with N inputs and 2^N outputs, one and only one of which is asserted to indicate the numerical value of the N input lines read as a binary number. A decoder and demultiplexer use the same internal circuitry.

Demultiplexer: (DEMUX) A logic circuit with K control inputs that steers the data input to the one of the 2^K outputs selected by the control inputs.

D flip-flop (DFF): Data flip-flop. A fundamental sequential circuit whose output changes only upon a clock signal and whose output represents the data on its input at the time of the last clock.

Don't care: In a truth table or Karnaugh-map, a state that is irrelevant to the correct functioning of the circuit (e.g., because it never occurs in the intended application). Thus, the designer "doesn't care" whether that state is asserted, and he or she may choose the output that best minimizes the number of gates.

Edge-triggered FF: A flip-flop that changes state on a clock transition from low to high or high to low rather than responding to the level of the clock signal. Contrast to **master–slave FF**.

Encoder: A logic circuit with 2^N inputs and N outputs, the outputs indicating the binary number of the one input line that is asserted. *See also* **priority encoder**.

Flip-flop: Any of several related bistable circuits that form the memory elements in clocked, sequential circuits.

FPGA: Field-programmable gate array. VLSI chips with a large number of reconfigurable gates that can be "programmed" to function as complex logic circuits. The programming can be done on site and in some cases may be dynamically (in circuit) reprogrammable.

Glitch: A transient transition between logic states caused by different delays through parallel paths in a logic circuit. They are unintentional transitions, so they do not correctly represent the logic of the intended design.

HDL: Hardware Description Language. A language that resembles a programming language with added features for specifying hardware designs.

IOB: I/O block in a Xilinx FPGA. Block on the periphery of an FPGA that supports the input and/or output of a signal from the FPGA to its external environment.

Karnaugh map: A mapping of a truth table into a rectangular array of cells in which the nearest neighbors of any cell differ from that cell by exactly one binary input variable. K-maps are useful for minimizing Boolean logic functions.

Master–slave FF: A flip-flop that changes state when the clock voltage reaches a threshold level. Contrast to **edge-triggered FF**.

Multiplexer: (MUX) A circuit with N control inputs to select among one of 2^N data inputs, and connect the appropriate data input to the single output line.

PIP: Programmable Interconnect Point on a Xilinx FPGA. A pass transistor with a memory bit connected to its gate terminal. If the memory is loaded with a "1" the two ends are connected; if loaded with a "0," the two ends are not connected. This is the basis of programmable interconnect.

Priority encoder: An encoder with the additional property that if several inputs are asserted simultaneously, the output number indicates the numerically highest input that is asserted. For example, if lines 1 and 3 were both asserted, the output value would be 3.

Saturated logic: Logic gates whose output is fully on or fully off. Saturated logic dissipates no power except while switching. The opposite of ***saturated logic*** is ***active logic***.

Sequential circuit: A circuit that goes through a sequence of stable states, transitioning between such states at times determined by a clock signal. The output of a sequential circuit depends both on its current inputs and its history, which is captured in the states. Contrast with **combinational** circuit.

Transparent latch: Essentially, a flip-flop that continuously passes the input to the output (thus *transparent*) when the clock is high (low) but holds the last output during any interval when the clock is low (high). The circuit is said to have *latched* when it is holding its output constant regardless of the value of the input.

VLSI: Very Large Scale Integrated Circuit. A semiconductor device that integrates millions of transistors on a single chip. VLSI chips are typically very high speed and have very high power dissipation.

References

Ashenden, P.J. 2001. *The Designer's Guide to VHDL*, 2nd ed. Morgan Kaufmann.

Boole, G. 1998. *The Mathematical Analysis of Logic*. St. Augustine Press, Inc.

Karnaugh, M. 1953. A map method for synthesis of combinational circuits. *Trans. AIEE. Comm. and Electron.*, 72(1):593–599.

Katz, R.H. 2003. *Contemporary Logic Design*, 2nd ed. Addison-Wesley Publishing.
Mano, M.M. and Kime, C.R. 2000. *Logic and Computer Design Fundamentals*, 2nd ed. Prentice Hall.
Moorby, P.R. and Thomas, D.E. 2002. *The Verilog Hardware Description Language*, 5th ed. Kluwer Academic Publishers.
Salcic, Z. and Smailagic, A. 2000. *Digital Systems Design and Prototyping Using Field Programmable Logic and Hardware Description Language*, 2nd ed. Kluwer Academic Publishers.
Wakerly, J.F. 2000. *Digital Design: Principles and Practice*, 3rd ed. Prentice Hall.
Zeidman, R. 2002. *Designing with FPGAs and CPLDs*. CMP Books.

Further Information

This is a very quick pass through digital circuit design. What has been covered in this chapter provides a good overview of the principles as well as information to help the reader understand the chapters on computer architecture in this volume.

There are many textbooks devoted to the subject of digital logic design. Wakerly [2000] emphasizes basic principles and the underlying technologies. Other digital logic texts emphasize logic design tools [Katz 2003] and computer design fundamentals [Mano and Kime 2000].

There have also been many volumes published on design entry, tools for automating the digital desgin process, and mapping designs onto Field Programmable Logic (FPL). The Hardware Description Languages (HDLs) most widely used today are VHDL [Ashenden 2001] and Verilog [Moorby and Thomas 2002]. The interested reader may also wish to pursue the topic of design with field programmable logic [Zeidman 2002]. Salcic and Smailagic [2000] brings the subjects of logic design, FPL, and HDLs together in one volume.

Ongoing research in this area is concerned with design entry, automation of the design process, and new architectures and technologies for implementing digital designs. The research in design entry is focused on raising the level of specification of digital logic designs. New HDLs based on high-level languages such as Java and C are being developed. Another approach is design environments that incorporate sophisticated libraries of very complex, parameterized components such as digital filters, ALUs, and Ethernet controllers. The user can customize these blocks for a specific design.

Along with higher levels of design specification, researchers are investigating more sophisticated design automation tools. The goal is to have designers specify the functionality of their designs, and to use synthesis tools to automatically translate that functionality to efficient hardware implementations.

17

Digital Computer Architecture

17.1 Introduction
The Processor-Program Interface
17.2 The Instruction Set
ALU Instructions • Memory and Memory Referencing Instructions • Control Transfer Instructions
17.3 Memory
Register File • Main Memory • Cache Memory • Memory and Architecture • Secondary Storage
17.4 Addressing
Addressing Format • Physical and Virtual Memory
17.5 Instruction Execution
Instruction Fetch Unit • Instruction Decode Unit • Execution Unit • Storeback Unit
17.6 Execution Hazards
Data Hazards • Control Hazards • Structural Hazards
17.7 Superscalar Design
17.8 Very Long Instruction Word Computers
17.9 Summary

David R. Kaeli
Northeastern University

17.1 Introduction

A computer architecture is a specification which defines the interface between the hardware and software. This specification is a contract, describing the features of the hardware which the software writer can depend upon and identifying design implementation issues which need to be supported in the hardware.

While the term computer architecture has commonly been used to define the interface between the instruction set of the processing element and the software that runs on on the processsing element, the term has also been applied more generally to define the overall structure of the computing system. This structure generally includes a processing element, memory subsystem, and input/output devices. We will first discuss the more traditional use of the term, focusing on how a program interacts with a processing element, and then discuss design issues associated with the broader definition of this term.

17.1.1 The Processor-Program Interface

Tasks are carried out on a computer by software specific to each program task. A simple program, written in the high-level language C, is shown in Figure 17.1. This program computes the difference of two integers. Even within a simple programming example we are able to identify some of the necessary elements in a computer architecture (e.g., arithmetic and assignment operations, integers).

```
x = 5;          /* Initialize x to 5 */
y = 3;          /* Initialize y to 3 */
z = x - y;      /* Compute the difference */
```

FIGURE 17.1 High-level language program.

```
x:    .int 5
y:    .int 3
z:    .
      load r1, x
      load r2, y
      sub r3, r1, r2
      store r3, z
```

FIGURE 17.2 Assembly-language version of HLL program.

High-level languages passes through a series of phases of transformation, including compilation, assembly, and linking. The result is a program that can run on an *execution processor*.

An assembly-code version of the subtraction machine code is shown in Figure 17.2. After the assembly code (stored in an object file) is then compiled, linking is performed. Linking merges all compiled elements of the program and constructs the final machine-language representation of the tasks to be performed.

The machine code of the computer system comprises a set of primitive operations which are performed with great rapidity. We refer to these operations as instructions. The set of instructions provided is defined in the specification of the architecture. An instruction set is just one aspect of defining an architecture.

Instructions are executed on the processor (commonly called the CPU or microprocessor), which may comprise a number of units, including (1) an *arithmetic logical unit* (ALU), (2) a *floating-point unit* (FPU), (3) local memory, and (4) external bus control. A particular computer architecture can be realized in hardware in a wide variety of hardware organizations. What remains constant across these different implementations of the architecture is a common software interface that programmers can depend upon. The Intel Pentium IV and SPARCV9 architectures are two good examples of well-defined computer architectures.

In this chapter various aspects of a computer architecture are presented. The design of a digital computer will also include the supporting memory system and input/output devices necessary. We will begin our discussion by describing the fundamentals of an instruction set.

17.2 The Instruction Set

An instruction set includes the machine-language primitives which can be directly executed on a processor. Instruction classes present in most instruction sets include:

1. ALU instructions (e.g., integer add/subtract, shift left/right, logical and/or/xor/inversion).
2. Memory accessing instructions (e.g., load and store).
3. Control transfer instructions (e.g., conditional/unconditional branch, call/return, and interrupt).

Two prevalent instruction set implementation paradigms are *reduced instruction set computers* (RISC) [Patterson 1985] and *complex instruction set computers* (CISC) [IBM 1981]. RISC microprocessors are designed around the concept that by keeping instructions simple, the compiler will have greater freedom to produce optimized code (i.e., code which will execute faster). CISC architectures use a very different principle, attempting to perform operations specified in the high-level language in a single instruction (commonly referred to as reducing the semantic gap). While RISC architectures have become the popular paradigm, the Intel *x*86 and Motorola 680*x*0 architectures continue to employ a CISC architecture (actually

the more recent Intel *x*86 processors utilize a CISC instruction set, but are implemented as RISC machines at the hardware level).

While these CISC and RISC models are based on different principles, they both contain instructions from the three classes above. The underlying principles are that RISC instruction sets are simple (or reduced) and that CISC instruction sets are complicated (or complex).

Most architectures include floating-point instructions. Those implementations of the architecture which contain a floating-point unit (FPU) are able to execute floating-point operations at speeds approaching integer operations. If a floating-point unit is not provided, then floating-point instructions are emulated by the integer processor (using a software program). When the hardware encounters a floating-point instruction, and if no FPU is present, a message (i.e., a software interrupt) is presented to the operating system. In response to this message, the floating-point instruction is executed using a number of integer instructions (i.e., it is emulated). The performance of emulated floating-point instructions is typically 3–4 orders of magnitude slower than if an FPU were present. Specific instructions may also be provided to manipulate decimal or string data formats. Most modern architectures include instructions for graphics and multimedia (e.g., the Intel *x*86 provides an architectural extension for multimedia called MMX) [Peleg 1997].

17.2.1 ALU Instructions

An ALU, which builds on the adder/subtracter presented in the chapter on digital logic, is used to perform simple operations upon data values. The ALU performs a variety of operations on input data fed from its two input registers and stores the result in a third register. Figure 17.3 shows an example of an ALU which might be found in the CPU.

The ALU is supplied data from a pair of registers *a* and *b* (registers are typically designed using D-flip-flops). The resulting answer is placed into register *c*. The function performed is determined based on the value of the select lines, which is directly derived by the instruction currently being executed. Table 17.1 shows the possible values for the three control lines shown in Figure 17.3.

The select lines are decoded (e.g., with a 3-to-8 decoder) to specify the desired operation. For example, the machine code format for the subtract instruction's operation code would contain (or be decoded into) the bit values 001. The assignment of these values is defined by the architecture and generally appears in a programmer's reference manual for the particular instruction set.

17.2.2 Memory and Memory Referencing Instructions

To run our program, we must have a place to store the machine code, and also a place to act as a scratch pad for intermediate results. Main memory provides a place where we can temporarily store both the machine instructions and data values associated with our program.

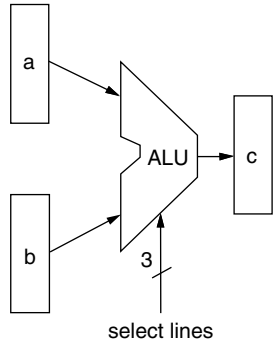


FIGURE 17.3 ALU with input and output registers.

TABLE 17.1 Sample ALU Operations

Operation Selected	Select-Line Value
$c = a + b$	000
$c = a - b$	001
$c = a$ shifted left b bits	010
$c = a$ shifted right b bits	011
$c = a$ OR b	100
$c = a$ AND b	101
$c = a$ XOR b	110
$c =$ NOT a	111

When a program begins execution, it is loaded into memory by the operating system. In our example, the values of the variables **x** and **y** in our high-level language program need to be initially stored in memory. The compiler will reserve memory locations to store these values and will provide instructions which will retrieve these values to initialize **x** and **y.** Then **x** and **y** are supplied to the input of the ALU via instructions that load them from registers **r1** and **r2.** The subtract instruction will tell the ALU to produce the difference **(r1**−**r2)** in register **r3.**

A computer architecture defines how data are stored in memory and how they are retrieved. To retrieve the values of **x** and **y** from memory, a load instruction is used. A load retrieves data values from memory and loads them into registers (e.g., data values **x** and **y,** and registers **r1** and **r2).** A store instruction takes the contents of a register (e.g., register **r3)** and stores it to the specified memory location (e.g., the memory location which the compiler assigned to **z).** These instructions are necessary for obtaining the data upon which the CPU will operate and to store away results produced by the CPU.

The CPU needs a way to differentiate between different data. Just as we are assigned Social Security numbers to differentiate one taxpayer from another, memory locations are assigned unique numbers called memory addresses. Using a memory address, the CPU can retrieve the desired datum. We will discuss addressing a little later and will focus on the aspects of addressing which are defined by the computer architecture.

RISC CPUs provide individual instructions for loading from and storing to memory. Pure RISC architectures provide only two memory-referencing instructions, load and store. All ALU instructions can have only input and output operands specified as registers or integer values. In contrast, CISC architectures provide a variety of instructions which both access memory and perform operations in a single instruction. A good discussion comparing the implications of these capabilities can be found in Colwell et al. [1985].

17.2.3 Control Transfer Instructions

Control transfer instructions cause a break in the sequential flow of program execution. The transfer can be performed unconditionally or conditionally. When an unconditional control transfer occurs (commonly referred to as a jump), the execution processor will begin processing instructions from a different portion of the program. Jumps include unconditional jump instructions, call instructions, and return instructions.

Interrupts are another form of jump. They break the sequential instruction flow unconditionally and transfer control not to another part of the current application but instead to an interrupt service routine. Interrupts can be programmed (inserted by the programmer or compiler) or they can occur due to hardware or software events (e.g., input/output, timers). Interrupts transfer control over to the operating system. The event which caused the interruption is handled, and then control is later passed back to the program.

A conditional control transfer (commonly referred to as a conditional branch) is dependent on the execution processor state determined by the execution of past instructions. Conditional branches are used for making decisions in control logic, checking for errors or overflows, or a variety of other situations. For instance, we may want to check if the result of adding our two integer numbers produced an overflow (a result which cannot be represented in the range of values provided for in the result).

Conditional branch instructions can perform both the decision making and the control transfer in a single branch instruction, or a separate comparison instruction (i.e., an ALU operation) can perform the comparison upon which the branch decision is dependent. Next, we explore the memory elements provided in the support of the execution processor.

17.3 Memory

The interface between the memory system and CPU is also defined by the architecture. This includes defining the amount of space addressable, the addressing format, and the management of memory at various levels of the memory hierarchy.

In traditional computer systems, memory is used for storing both the instructions and the data for the program to be executed. Instructions are fetched from memory by the CPU, and once decoded (instructions are typically in an encoded format, similar to that found in Table 17.1 for ALU instructions), the data operands to be operated upon by the instruction are retrieved from, or stored to, memory. Memory is typically organized in a hierarchy. The closer the memory is to the CPU, the faster (and more expensive) it is. The memory hierarchy shown in Figure 17.4 includes the following levels:

1. Register file.
2. Cache memory.
3. Main memory.
4. Secondary memory (disk, tapes, etc.).

Above each level in Figure 17.4 is a measure of its typical size and access speed in contemporary technology. Notice we include multiple levels of the cache, which are commonly found as on-chip elements in today's CPUs.

17.3.1 Register File

The register file is an integral part of the CPU and, as such, is clearly defined by the architecture. The register file can provide operands directly to the ALU. Memory referencing instructions either load to or

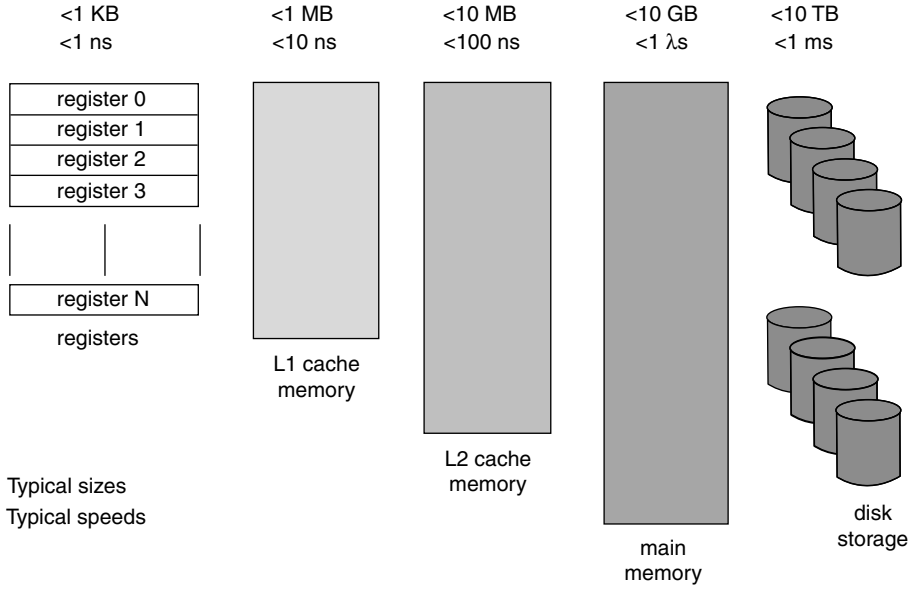


FIGURE 17.4 The memory hierarchy.

store from the registers contained in the register file. The register file can contain both general-purpose registers (GPRs) and floating-point registers (FPRs). Additional registers for managing the CPU state and addressing information are generally provided.

The register file represents the lowest level in the memory hierarchy, since it is closest to the processor. The registers are constructed out of fast flip-flops. The typical size of a GPR in current designs is 32 or 64 bits, as defined by the architecture. Registers are typically accessible on either a bit, byte, halfword, or fullword granularity (a word refers to either 32 or 64 bits). While a processor many have many hundred hardware registers, the architecture generally defines a smaller set of general purpose registers. In many instruction sets, use of particular registers is reserved for use by selected instructions (e.g., in the Intel *x*86 architecture, register CX holds the count value used by the *LOOP* instruction).

17.3.2 Main Memory

Main memory is physical (versus virtual) memory which typically resides off the CPU chip. The memory is usually organized in banks and supplies instructions and data to the processor. Main memory is typically byte-addressable (meaning that the smallest addressable quantity is 8 bits of instruction or data). Main memory is generally implemented in dynamic random-access memory (DRAM) to take advantage of DRAM's low cost, low power drain, and high storage density. The costs of using this memory technology include reduced storage response time and increased design complexity (DRAM needs to be periodically refreshed, since it is basically a tiny capacitor).

Main memory is typically organized to provide efficient access to sequential memory addresses. This technique of accessing many memory locations in parallel is called interleaving. Interleaved memory allows memory references to be multiplexed between different banks of memory. Since memory references tend to be sequential in nature, allowing the processor to obtain multiple addresses in a single access cycle can be advantageous. Multiplexing can also provide a substantial benefit when using cache memories.

17.3.3 Cache Memory

Cache memory is used to hold a small subset of the main memory. The cache is typically developed in static random access memory (SRAM), which is faster than DRAM, but is more expensive, more power-hungry, and less dense. SRAM technology does not need to be refreshed. The cache contains the most recently accessed code and data. The cache memory is used to provide instructions and data to the processor faster than would be possible if only main memory were used. In current CPUs, separate caches for instructions and data are provided on the CPU chip.

In most processors produced today, the first level cache (referred to as the L1-cache) is designed to be on the CPU chip. In many designs, a second level of caching is provided on chip. Generally, most processors provide an on-chip L1 instruction cache and an on-chip L1 data cache. L2 caches (either seperate or unified) are also found on-chip on many designs today. It is also common to see an L3 cache on multi-CPU systems, that is located off chip.

17.3.4 Memory and Architecture

The amount of the memory system defined by the architectural specification varies greatly. The design and layout of the cache(s) and main memory are typically not defined by the architecture. The architecture specifies the bit and byte arrangement of data and the addressing formats within instructions. The interface (e.g., address lines) to external (main) memory can also be specified in the architecture. The virtual memory systems may also place some restrictions on the addressing scheme used for the main memory system. In some architectures, instructions have been provided to manipulate the state of cache memory to ensure the memory coherency of the system (memory coherency refers to the issue of having only a single valid copy of a datum in the system).

Since the performance gap between accessing cache and main memory is so great, maximizing the probability of finding a memory request resident in cache when requested is of great interest. Some of the tradeoffs in the design of a cache include the block size (minimum unit of transfer between main memory and the cache), associativity (used to define the mapping between main memory and the cache), handling of cache writes (for data and mixed caches), number of entries, and mapping strategies. Handy provides a thorough discussion on a number of these topics [Handy 1993]. While many of these design parameters are important, they typically are not included in the architectural definition and are left as part of the design not specified.

17.3.5 Secondary Storage

When you first turn on your computer, the program code will reside in secondary storage (disk storage). When the program is run, it will be transferred to main memory, then to cache, and then to the processor (generally these last two transfers are performed simultaneously). Disk storage is commonly used for secondary storage, since it is nonvolatile storage (the contents are maintained even when power is turned off).

A disk is designed using magnetic media, and data are stored in the form of a magnetic polarization. The disk comprises one or more platters which are always spinning. When a request is made for instructions or data, the disk must rotate to the proper location in order for the read heads to be able to access the information at that location. Because disk rotation is a mechanical operation, disk accesses are many orders of magnitude slower than the access time of the DRAM used for main memory.

After a program has been run once, it will reside for a period of time in either the cache or main memory. Access to the program will be faster upon subsequent runs if the execution processor does not have to wait for the program to be reloaded from disk.

The architecture of a system does not typically impose any limitations on the organization of secondary storage besides defining the smallest addressable unit (typically called a block) and the total amount of addressable storage.

17.4 Addressing

All instructions and data in registers and memory are accessed by an address. Next we look at various aspects of addressing: addressing format, physical addressing, virtual addressing, and byte ordering.

17.4.1 Addressing Format

Defined within all architectures are the various addressing formats which are permissible with particular instructions. Addresses define where input and output operands will located in the system. Standard addressing formats provided with most architectures include:

1. *Register:* The register number is specified in the instruction. The register number is used directly to access the register file.
2. *Immediate:* A constant field is specified in the instruction. This field is used directly as an operand. While this format does not fit nicely into our concept of addressing, immediate addressing is a commonly used address format for specifying the operands.
3. *Direct:* The full address is provided in the instruction. No address calculation is necessary. The address field can be used directly (thus, direct addressing). Direct addressing is commonly used for accessing static data.
4. *Register indirect:* The number of the GPR which contains the memory address is specified. Register addressing is commonly used for accessing via a pointer value. The format is also referred to as *register deferred*.
5. *Memory indirect:* The number of the GPR is specified which contains the address where the address of the desired data is located. In this case, two memory references are performed to obtain the desired data. Indirect addressing is commonly used to dereference pointers. This format is also referred to as *memory deferred*.

6. *Base displacement:* The number of the GPR which contains the base address of some data structure is specified. The base value is added to a displacement field to obtain the final memory address. Base-displacement addressing is commonly used for addressing sequential data patterns (e.g., arrays, structures).
7. *Indexed:* A GPR is added to a base register (GPR) to obtain the memory address. Some architectures provide separate index registers for this purpose. Other architectures add an index to a base register and possibly even include a displacement field. Indexed addressing is commonly used for traversing complex data structures such as link lists of structures.

Register, immediate, and base displacement addressing are the most commonly used addressing formats. Some other addressing modes that are found in selected instruction sets include:

1. *Auto-increment/auto-decrement*. Similar to register indirect, with the addition that an index register is also incremented/decremented. This format is commonly used when accessing arrays within a loop.
2. *Scaled*. Similar to register indexed, except that a second index register is multiplied by a constant (typically the data type size) and this second index is added to the base and the index. This allows for efficiently traversing arrays with nonstandard data sizes.

17.4.2 Physical and Virtual Memory

Virtual memory refers to the ability of a computer system to address a larger address range than the amount of physically installed memory (DRAM). This is a very cost-effective approach to computing. We need to have only a small amount of DRAM memory installed on our system, compared to the actual addressable space. The range of the virtual address space places a limit on the amount of addressable secondary storage (including disks, printers, devices, etc.). Prior to the introduction of the concept of virtual memory, the programmer had to insert explicit commands to load programs in from disk storage. One segment supplanted another in memory (this was called and *overlay*). With the introduction of virtual memory, memory overlays were no longer necessary. Some motivating factors behind using virtual memory include: (1) providing efficient usage of physical memory without the need for explicit overlays, and (2) allowing full flexibility of placement of instructions and data.

A virtual memory address generally refers to any memory location in the available address range as defined by the architecture. A physical memory address refers to a memory location in main memory. The physical-memory-address range is defined by the amount of installed main memory and is bounded by the number of bits provided for in the address scheme of the architecture. The virtual-address range is defined by the largest permissible address value. Figure 17.5 provides an example of how a 32-bit virtual address can address over 4 billion different memory locations. Since the virtual-address range is generally 2 or more orders of magnitude greater than the installed physical memory, a mapping from virtual to physical memory is performed. See Figure 17.5 for an example of how different regions (i.e., pages) in the virtual address space map to the physical address space.

The operating system performs the mapping between the virtual and physical address spaces. The mapping is performed on either a fixed-size or variable-sized memory blocks (pages versus segments, respectively). In Figure 17.5 we see that a fixed-size page (4096 bytes) is being mapped to the physical memory address space. This mapping is stored in a table in memory (called a page table). The page table will hold virtual-to-physical mapping for all pages present in the physical memory. In Figure 17.5 we see that virtual page A' is mapped to physical memory location A and that virtual page M' is mapped to physical page M. Thus, sequentiality is limited to a page and generally does not extend past the page boundary, which is to say, two sequential pages in the virtual address space need not be mapped to two sequential physical pages.

Pages are brought from secondary storage into physical (main) memory when they are requested by the execution processor. This is referred to as *demand paging*. The size of a page is typically fixed (e.g., 4096 bytes in our example). Segmented virtual memory systems provide a variable-sized unit of

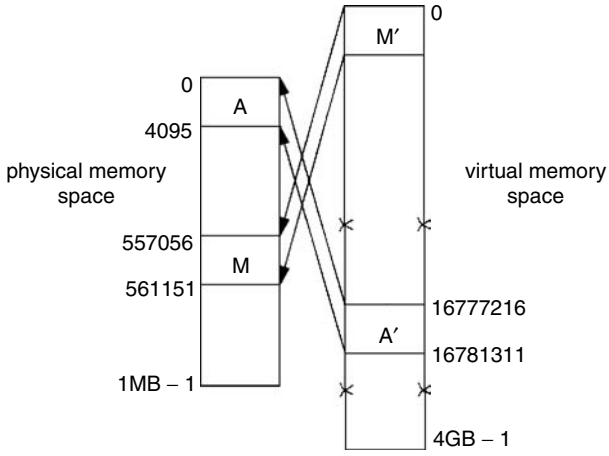


FIGURE 17.5 Mapping of physical to virtual memory addresses.

transfer between the virtual and physical memory spaces. A good discussion of the different types of virtual memory systems can be found in Feldman and Retter [1995] as well as in Chapter 85.

Since every instruction execution in the CPU would require that at least two memory accesses be performed to obtain instructions (one access to obtain the physical address stored in the page table, and a second access to obtain the instruction stored at that physical address), a hardware feature called a *translation lookaside buffer* (TLB) was proposed. The TLB caches the recently accessed portions of the page table and quickly provides a virtual-to-physical translation of the address. The TLB is generally located on the CPU to provide fast translation capability. Further discussion on TLBs can be found in Teller [1991].

Now that we know a little bit more about instructions, addressing, and memory, we can begin to understand how instructions are processed by the CPU.

17.5 Instruction Execution

To this point we have focused on *architected* features of a digital computer system. These will be used to document the interface to which compiler and application developers must program and the features which the hardware designer must provide. Next we will describe some *nonarchitected* features of a digital computer system which are typically left to the implementer to decide upon. The first feature looks at how we organize elements of the CPU in order to execute an instruction efficiently.

Instructions are requested from the memory system. This involves sending an address out to the memory and waiting until the memory system produces the requested address. Once retrieved, the instruction enters the processor. The execution of an instruction involves a number of steps. Figure 17.6 provides the buses and logic to be used to describe the steps for nonpipelined instruction execution.

In a nonpipelined system, a single instruction at a time enters the processor and waits the necessary amount of time for all of the steps associated with an instruction to be completed. For instance, in Figure 17.6 the subtract instruction is loaded into the instruction register and decoded by the control logic. Then the input registers to the ALU are enabled and the ALU is programmed to perform a subtract. After a sufficient delay to allow all of these operations to be performed and for the output of the ALU to reach a steady-state value, the output register is latched. This all happens during a single processor clock cycle (the CPU clock is typically used to synchronously capture the new state of memory devices).

If instructions were allowed to flow through the execution processor only one at a time, the throughput of the execution processor (the rate at which instructions exit the processor) would be low. This is because a majority of the elements of the CPU would remain idle while different phases of the instruction execution progressed.

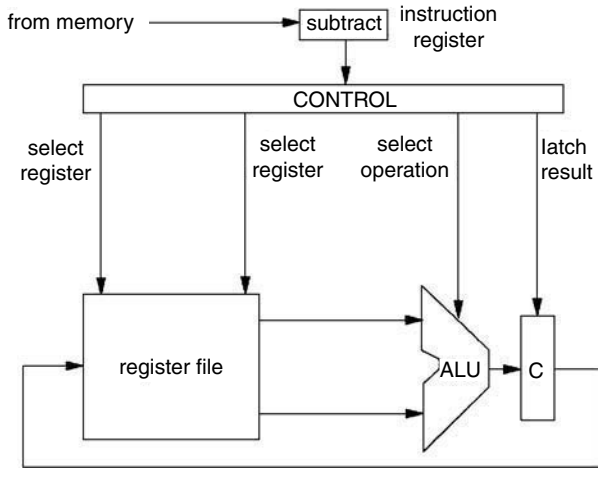


FIGURE 17.6 Nonpipelined instruction execution.

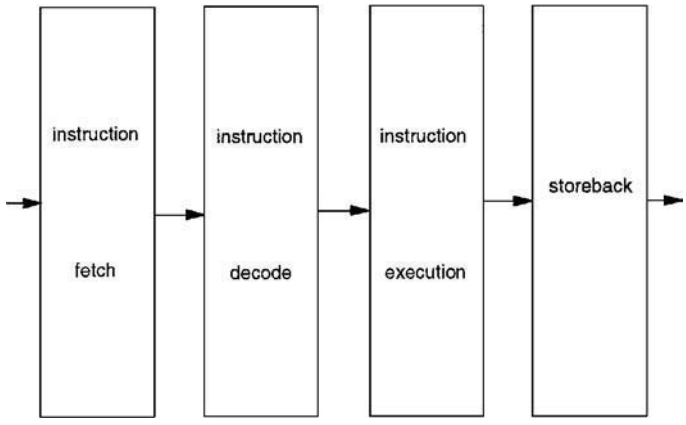


FIGURE 17.7 Pipelined execution of the subtraction example.

Instead, consider the execution of the same instruction broken up into a number of stages, as is shown in Figure 17.7. A single clock is used to latch results at the end of each stage (in practice, different clock edges may be used to latch particular elements in a stage). The key ideas here are that all stages work synchronously and that multiple instructions are being processed simultaneously (similar in concept to Henry Ford's assembly line). This is called *pipelining*. Given a pipeline of n stages, n instructions can be in process simultaneously (each at different stages of the pipeline). This can dramatically increase instruction throughput.

To complete the execution of an instruction, a series of tasks are performed by a number of functional units. To begin with, the instruction must be present in the processor. We will label this stage the instruction fetch stage of the pipeline, since it is fetching the next instruction to be executed.

The second stage of the pipeline is instruction decode. During this stage, the machine language is decoded and the operation to be performed is discovered. The decoding process generates the control bit values which will enable buses, latch registers, and program ALUs. These are represented in Figure 17.6 by the vertical lines produced by the control logic. Also during this stage, the necessary operands (e.g., **x** and **y** for the subtract) may also be discovered.

The third stage of the pipeline is the instruction execution stage. During this stage, the operation specified in the instruction operation code is actually performed (e.g., in our example program, the subtract will take place during this stage and the result will be latched into the ALU result register **c)**.

The final stage is the storeback stage. During this stage, the results of the execution stage are stored back to the specified register or storage location. While the discussion here has been for ALU-type instructions, it can be generalized for memory referencing and control-transfer instructions. Kogge describes a number of pipeline organizations [Kogge 1981].

Next we discuss the individual contents of each of these stages.

17.5.1 Instruction Fetch Unit

An executable computer program is a contiguous block of instructions which is stored in memory (copies may reside simultaneously in secondary, main, and cache memory). The *instruction fetch unit* (IFU) is responsible for retrieving the instructions which are next to be executed. To start (or restart) a program, the dispatch portion of the operating system will point the execution processor's *program counter* (PC) to the beginning (or current) address of the program which is to be executed. The IFU will begin to retrieve instructions from memory and feed them to the next stage of the pipeline, the *instruction decode unit* (IDU).

17.5.2 Instruction Decode Unit

Instructions are stored in memory in an encoded format. Encoding is done to reduce the length of instructions (common RISC architectures use a 32-bit instruction format). Shorter instructions will reduce the demands on the memory system, but then encoded instructions must be decoded to determine the desired control bit values and identify the accompanying operands. The instruction decode unit performs this decoding and will generate the necessary control signals, which will be fed to the *execution unit*.

17.5.3 Execution Unit

The execution unit will perform the operation specified in the instruction decoded operation code. The operands upon which the operation will be performed are present at the inputs of the ALU. If this is a memory referencing instruction (we will assume we are using a RISC processor for this discussion, so ALU operations are performed only on either immediate or register operands), address calculations will be performed during this stage. The execution unit will also perform any comparisons needed to execute conditional branch instructions. The result of the execution unit is then fed to the *storeback unit.*

17.5.4 Storeback Unit

Once the result of the requested operation is available, it must be stored away so that the next instruction can utilize the execution unit. The storeback unit is used to store the results of ALU operation to the register file (again, we are considering only RISC processors in this discussion), to update a register with a new value from memory (for LOAD instructions), and to update memory with a register value (for STORE instructions). The storeback unit is also used to update the program counter on branch instructions.

Figure 17.8 shows our subtraction code flowing through both a nonpipelined and a pipelined execution processor. The width of the boxes in the figures is meant to depict the length of a processor clock cycle. The nonpipelined clock cycle is longer (slower), since all of the work accomplished in the four separate stages of the instruction execution are completed in a single clock tick. The pipelined execution clock cycle is dominated by the time to stabilize and latch the result at the end of each stage. This is why a single pipelined instruction execution will typically take longer to execute than a nonpipelined instruction. The advantages of pipelining are reaped only when instructions are overlapped. As we can see, the time to execute the subtraction program is significantly smaller for the pipelined example (but not nearly four times smaller).

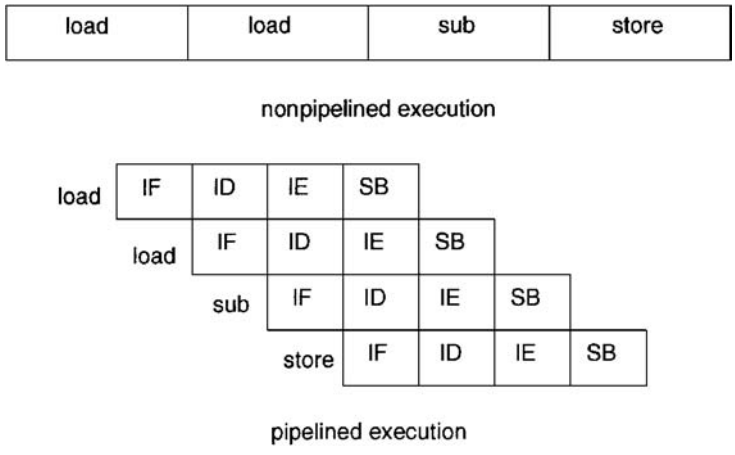


FIGURE 17.8 Comparison of nonpipelined and pipelined execution.

Also, other executions can be overlapped with this example code (i.e., instructions in the pipeline prior to the first load and after the store instruction). This is not the case for nonpipelined execution.

Note that in our examples in Figure 17.8 we are assuming that all nonpipelined instructions and pipelined stages take a single clock cycle. This is one of the underlying principles in RISC architectures. If instructions are kept simple enough, they can be executed in a single cycle.

Pipelining can provide an advantage only if the pipeline is supplied with instructions which can be issued without any delay or uncertainty. The benefits of pipelining can be greatly reduced if stalls occur due to different types of hazards. We will discuss this topic next.

17.6 Execution Hazards

Consider attempting to process the high-level language example in Figure 17.9, which builds upon our subtraction program. If we look at what the compiler would do with this code, it might look something like the instruction sequence shown in Figure 17.10.

A problem occurs if we try to execute this on the pipelined model. The cause of the problem is illustrated in Figure 17.11, which shows the instruction sequence given in Figure 17.10 flowing through the pipeline. The multiply instruction needs to get the most recent update of the variable **z** (which will reside in **r3**). Given the pipeline model described above, the multiply instruction will be attempting to direct the contents of **r3** to the inputs of the ALU during its instruction decode stage. In the same cycle the subtract instruction will be trying to store the results of the subtraction to **r3** during the storeback stage. This is just one example of a *hazard*, called a data hazard. There are three classes of hazards:

1. *data hazards*, which include read-after-write (RAW), write-after-read (WAR), and write-after-write hazards
2. *control hazards*, which include any instructions or interruptions which break sequential instruction execution
3. *structural hazards*, which occur when multiple instructions vie for a single functional element (e.g., an ALU)

17.6.1 Data Hazards

Data hazards occur when there exist dependencies between instructions. For the hazard to occur, the instructions need to be close enough in the execution stream to coexist in the pipeline. The example provided in Figure 17.10 is just one type of interinstruction dependence, called a *read-after-write* hazard.

```
x = 5;       /* Initialize x to 5 */
y = 3;       /* Initialize y to 3 */
z = x - y;   /* Compute the difference */
w = z * z;   /* compute the square */
```

FIGURE 17.9 Subtraction program which also computes the square of the difference.

```
w:   .
x:   .int 5
y:   .int 3
z:   .
     load r1, x
     load r2, y
     sub r3, r1, r2
     store r3, z
     mult r4, r3, r3
```

FIGURE 17.10 Assembly-language version of Figure 17.9.

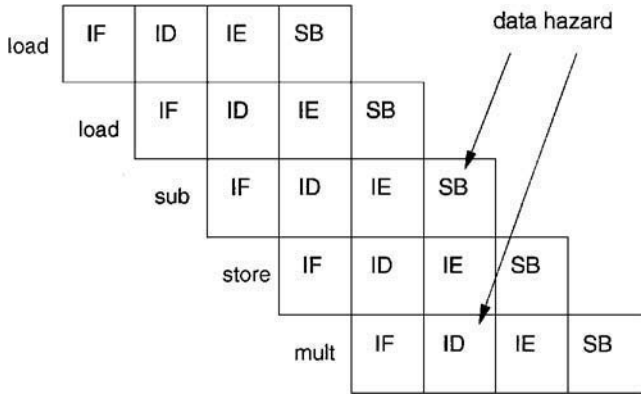


FIGURE 17.11 Example of a RAW hazard.

The multiply is attempting to read **r3** as the subtract is writing **r3.** This is the most commonly encountered type of data hazard.

A second type of data hazard occurs when a subsequent instruction modifies an operand of the current instruction before it is read by the current instruction. This is called a *write-after-read* hazard. This will occur only if writes are allowed to be performed early in the pipeline and reads are performed late in the pipeline.

A third type of data hazard occurs when a subsequent instruction modifies an operand of the current instruction before it is written by the current instruction. This is called a *write-after-write* hazard. This will occur only if writes are allowed to be performed both early and late in the pipeline.

A number of solutions to detecting data hazards have been proposed. Hardware techniques-of-forwarding data values can overcome a large number of these hazards which frequently occur. This technique is integrated into the decoding logic and keeps track of the source and destination of all instructions currently active in the pipeline. Hazard detection then becomes part of the instruction decoding stage. Compilers can also be tuned to eliminate the possibility of data hazards occurring.

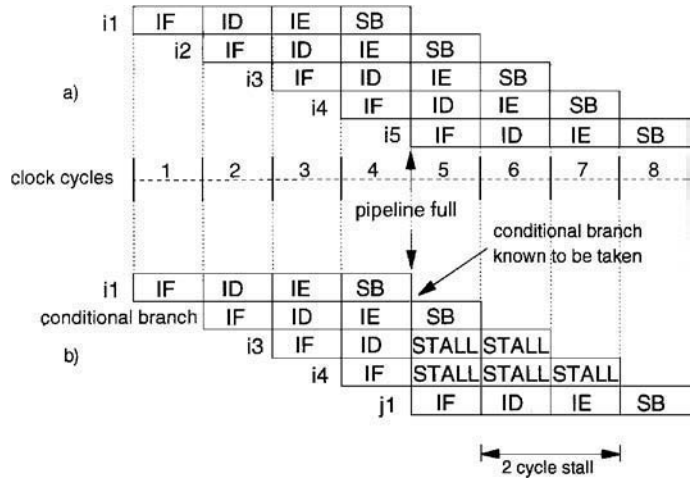


FIGURE 17.12 a) Ideal execution. b) A taken conditional branch.

17.6.2 Control Hazards

Another type of hazard which can affect pipeline throughput is a control hazard. This class of hazard includes any instructions which break the sequential pipeline flow. These include branches (both conditional and unconditional), jumps, calls and returns, and interrupts. Since the instruction fetch stage of the pipeline assumes that the sequential instruction path will be followed, when a change in the path takes place, the instructions which have entered the pipeline need to be flushed and the program counter needs to be adjusted to point to the new instruction stream. Once the instruction fetch unit redirects instruction fetching, the pipeline will be refilled. The result is that the benefits of pipelining will be dramatically reduced.

Figure 17.12a shows an ideal execution which does not contain any breaks in sequential execution. After the pipeline fills (i.e., after cycle 4) an instruction exits the pipeline each cycle. Figure 17.12b shows how a taken conditional branch will degrade pipeline performance. After clock cycle 4, the pipeline is full of valid instructions and the instruction **i2** is a taken conditional branch instruction. The pipeline will know that this instruction is a branch after it has passed through the ID stage (instruction decode). The pipeline forges ahead, hoping that the branch will not be taken. The pipeline detects that the conditional branch is taken when it completes the IE stage (at the end of clock cycle 4). Instructions **i3** and **i4** have already entered the pipeline and will need to be flushed. This introduces a 2-cycle stall into the pipeline. The branch is taken to instruction **j1**, and the pipeline is refilled. There is a 2-cycle delay in this example, since 2 cycles pass before a valid instruction (i.e., instruction **j1**) exits the pipeline. The average number of cycles per instruction (once the pipeline has filled) is one for the ideal execution and two for the code containing the conditional branch.

To handle control hazards many techniques have been proposed. History-based branch prediction is one mechanism which is now commonly used in most microprocessors to reduce the negative effects of control hazards. This type of mechanism attempts to predict the future behavior of branches by recording history of past execution. A table of entries, each containing information for a particular branch, is maintained. History for each branch is recorded (e.g., taken vs. not-taken, and branch destination). Then when the branch instruction is again instruction fetched, a lookup in the history table finds the outcome of the last execution(s) for this branch and then predicts that history will repeat itself. It has been found that past history is a very good predictor of future behavior. Using history-based branch prediction, a majority of the pipeline stalls encountered due to conditional branches can be eliminated. Cragon describes a number of these mechanisms [Cragon 1992].

Another method for hiding the effects of control hazards is called *branch predication*. Each predicated instruction is accompanied by a predication bit. Using predication, instructions from both paths of a

conditional branch can be issued, though only those instructions on correct path are committed. The predicate bits indicate which instructions are committed, and which instructions are *squashed*. Interest in predicated execution has recently been renewed with the introduction of the Intel IA-64 architecture [Gwennap 1998].

17.6.3 Structural Hazards

Structural hazards are the third class of pipeline delays which can occur. They occur when multiple instructions active in the pipeline vie for shared resources. Some examples of structural hazards include: two instructions trying to compute a memory address in the same cycle when a single address-generation ALU is provided or two instructions both attempting to access the data cache in the same cycle when only a single data-cache access port is provided.

A number of approaches can be taken to alleviate the delays introduced by structural hazards. One approach is to further exploit the principle of pipelining by employing pipelined stages within each pipeline unit. This technique is called *superpipelining*. This will allow multiple instructions which are active in the pipeline to coexist in a single pipeline stage.

Another approach is to provide multiple functional units (e.g., two cache ports or multiple copies of the register file). This approach is commonly used when high performance is critical or when we want to be able to issue multiple instructions in a single clock cycle. Multiple issue processors, also called *superscalar* processors, are discussed next.

17.7 Superscalar Design

If we can solve all the problems associated with hazards, an instruction should be exiting the pipeline every processor clock cycle. While this level of performance is seldom achieved (mainly due to latencies in the memory system and the limitations of effectively handling control hazards), we would like to be able to see multiple instructions exit the pipeline in a single clock cycle if possible. This approach has been labeled *superscalar* design. The idea is that if the compiler can produce groups of instructions which can be issued in parallel (which do not contain any data or control dependencies), then we can attain our goal of having multiple instructions exit the pipeline in a single cycle.

Some of the initial ideas which have motivated this direction date back to the 1960s and were initially implemented in early IBM [Anderson et al. 1967] and CDC machines [Thornton 1964]. The problem with this approach is finding a large number of instructions which are independent of one another. The compiler cannot exploit the scheduling to perfection because some conflicts are data-dependent. We can instead design complex hazard detection logic in our execution processor. This has been the approach taken by most superscalar designers.

Two issues occur in superscalar execution. First, can we issue nonsequential instructions in parallel? This is referred to as *out-of-order issue*. A second question is whether we can allow instructions to exit the pipeline in nonsequential order. This is referred to as *out-of-order completion*. A thorough discussion of the tradeoffs associated with superscalar execution and issue/completion design can be found in Johnson [1990].

17.8 Very Long Instruction Word Computers

In contrast to superscalar execution, which using *dynamic scheduling* to select at runtime the sequence of instructions to issue to the functional unit, Very Long Instruction Word (VLIW) architectures employ *static scheduling*, relying on the compiler to produce an efficient sequence of instructions to execute. A VLIW processor packs multiple, RISC-like, instructions into a long instruction word. If the compiler can find multiple instructions that can be issued in the same cycle, we should be able to expose high instruction-level parallelism. A number of designs have been developed based on a VLIW architecture, including the Cydra 5 [Beck 1993] and the Intel Itanium processors [Gwennap 1998].

17.9 Summary

This chapter has introduced many of the features provided in a digital computer architecture. The instruction set, memory hierarchy, and memory addressing elements, which are central to the definition of a computer architecture, were covered. Then some optimization techniques, which attempt to improve the efficiency of instruction execution, were presented. The hope is that this introductory material provides enough background for the nonspecialist to gain an appreciation for the pipelining and superscalar techniques that are currently used in today's CPU designs.

Defining Terms

Branch prediction: A mechanism used to predict the outcome of branches prior to their execution.

Cache memory: Fast memory, located between the CPU and main storage, that stores the most recently accessed portions of memory for future use.

Control hazards: Breaks in sequential instruction execution flow.

Data hazards: Dependencies between instructions that coexist in the pipeline.

Memory coherency: Ensuring that there is only one valid copy of any memory address at any time.

Pipelining: Splitting the CPU into a number of stages, which allows multiple instructions to be executed concurrently.

Predication: Conditionally executing instructions and only committing results for those instructions with enable predicates.

Structural hazards: A situation where shared resources are simultaneously accessed by multiple instructions.

Superpipelining: Dividing each pipeline stage into substages, providing for further overlap of multiple instruction execution.

Superscalar: Having the ability to simultaneously issue multiple instructions to separate functional units in a CPU.

Very Long Instruction Word: Specifies a multiple (although fixed) number of primitive operations that are issued together and executed upon multiple functional units. VLIW relies upon effecive static (compile-time) scheduling.

References

Anderson, D. W., Sparacio, F. J., and Tomasulo, R. M. 1967. The IBM 360 Model 91: processor philosophy and instruction handling. *IBM J. Res. Dev.* 11(1):8–24, Jan.

Beck, G. R., Yen, D. W., and Anderson, T. L. 1993. The Cydra 5 Mini-Supercomputer: Architecture and Implemementation. *Journal of Supercomputing*, 7:140–180.

Colwell, R. P., Hitchcock, C. Y., Jensen, E. D., Sprunt, H. M. B., and Kollar, C. P. 1985. Computers, complexity, and controversy. *IEEE Comput. Mag.* Sept., pp. 8–19.

Cragon, H. C. 1992. *Branch Strategy Taxonomy and Performance Models.* IEEE Computer Society Press, Los Alamitos, CA.

Feldman, J. and Retter, C. 1995. *Computer Architecture: A Designer's Guide to a Generic RISC*. Prentice–Hall, Reading, MA.

Gwennap, L. 1998. Intel's Merced and IA-64: Technology and Market Forecast. MDR Technical Library Report.

Handy, J. 1993. *The Cache Memory Book*. Academic Press, Boston.

IBM Corp. 1981. *IBM System/370 Principles of Operation*. Document No. GA22-7000-7, IBM Corp. New York, Mar.

Intel Corporation. 2001. Desktop Performance and Optimization of the Intel Pentium 4 Processor. Intel Corporation, Order Number 249438-01, Feb.

Johnson, M. 1990. *Superscalar Microprocessor Design*. Prentice–Hall, Englewood Cliffs, NJ.

Kogge, P. M. 1981. *The Architecture of Pipelined Computers*. McGraw–Hill, New York.

Patterson, D. 1985. Reduced instruction set computers. *Commun. ACM* 28(1):8–21, Jan.
Peleg, A., Wilkie, S., and Weiser, U. 1997. Intel MMX for Multimedia PCs, 40(1):24–38.
Sun Microsystems. 2002. V9 (64-bit SPARC) Architecture Book. www.sparc.com/standards.html.
Teller, P. 1991. *Translation Lookaside Buffer Consistency in Highly-Parallel Shared Memory Multiprocessors*, Ph.D. dissertation. New York University, May. Also available as an *IBM Research Report*, RC 16858, #74685, May 14.
Thornton, J. E. 1964. Parallel operation in the Control Data 6600. In *AFIPS Proc. Fall Joint Computer Conf.* No. 27.
Wang, P. H., Wang, H., Collins, J. D., Grochowski, E., Kling, R. M., and Shen, J. P. 2002. Memory Latency-Tolerance Approaches for Itanium Processors: Out-of-Order Execution vs. Speculative Precomputation, *Proc. of HPCA-8*, Feb., pp. 187–196.

Further Information

To learn more about the recent advances in computer architecture, you will find articles on a variety of related subjects in the following list of IEEE and ACM publications:

IEEE Transactions on Computers.
IEEE Computer Architecture Letters.
ACM SIGARCH Newsletter.
IEEE TCCA Newsletter.
Proceedings of the International Symposium on Computer Architecture, IEEE Computer Society Press.
Proceedings of the International Conference on High-Performance Computer Architecture, IEEE Computer Society Press.
Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems, ACM.

18

Memory Systems

Douglas C. Burger
University of Wisconsin at Madison

James R. Goodman
University of Wisconsin at Madison

Gurindar S. Sohi
University of Wisconsin at Madison

18.1 Introduction
18.2 Memory Hierarchies
18.3 Cache Memories
18.4 Parallel and Interleaved Main Memories
18.5 Virtual Memory
18.6 Research Issues
18.7 Summary

18.1 Introduction

The *memory system* serves as the repository of information (data) in a computer system. The processor (also called the central processing unit, or CPU) accesses (reads or loads) data from the memory system, performs computations on them, and stores (writes) them back to memory. The memory system is a collection of storage locations. Each storage location, or *memory word*, has a numerical *address*. A collection of storage locations form an *address space*. Figure 18.1 shows the essentials of how a processor is connected to a memory system via address, data, and control lines.

When a processor attempts to load the contents of a memory location, the request is very urgent. In virtually all computers, the work soon comes to a halt (in other words, the processor *stalls*) if the memory request does not return quickly. Modern computers are generally able to continue briefly by overlapping memory requests, but even the most sophisticated computers will frequently exhaust their ability to process data and stall momentarily in the face of long memory delays. Thus, a key performance parameter in the design of any computer, fast or slow, is the effective speed of its memory.

Ideally, the memory system must be both infinitely large, so that it can contain an arbitrarily large amount of information and infinitely fast, so that it does not limit the processing unit. Practically, however, this is not possible. There are three properties of memory that are inherently in conflict: speed, capacity, and cost. In general, technology tradeoffs can be employed to optimize any two of the three factors at the expense of the third. Thus it is possible to have memories that are (1) large and cheap, but not fast; (2) cheap and fast, but small; or (3) large and fast, but expensive. The last of the three is further limited by physical constraints. A large-capacity memory that is very fast is also physically large, and speed-of-light delays place a limit on the speed of such a memory system.

The **latency** (L) of the memory is the delay from when the processor first requests a word from memory until that word arrives and is available for use by the processor. The latency of a memory system is one attribute of performance. The other is **bandwidth** (BW), which is the rate at which information can be transferred from the memory system. The bandwidth and the latency are closely related. If R is the number of requests that the memory can service simultaneously, then

$$\text{BW} = \frac{R}{L} \tag{18.1}$$

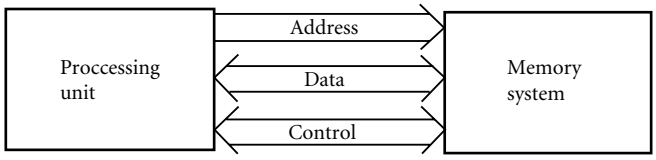


FIGURE 18.1 The memory interface. (*Source:* Dorf, R. C. 1992. *The Electrical Engineering Handbook*, 1st ed., p. 1928. CRC Press, Inc., Boca Raton, FL.)

From Equation (18.1) we see that a decrease in the latency will result in an increase in bandwidth, and vice versa, if R is unchanged. We can also see that the bandwidth can be increased by increasing R, if L does not increase proportionately. For example, we can build a memory system that takes 20 ns to service the access of a single 32-bit word. Its latency is 20 ns per 32-bit word, and its bandwidth is

$$\frac{32}{20 \times 10^{-9}} \text{ bits/s}$$

or 200 Mbyte/s. If the memory system is modified to accept a new (still 20-ns) request for a 32-bit word every 5 ns by overlapping requests, then its bandwidth is

$$\frac{32}{5 \times 10^{-9}} \text{ bits/s}$$

or 800 Mbyte/s. This memory system must be able to handle four requests at a given time.

Building an ideal memory system (infinite capacity, zero latency, and infinite bandwidth, with affordable cost) is not feasible. The challenge is, given the cost and technology constraints, to engineer a memory system whose abilities match the abilities that the processor demands of it. That is, engineering a memory system that performs as close to an ideal memory system (for the given processing unit) as is possible. For a processor that stalls when it makes a memory request (some current microprocessors are in this category), it is important to engineer a memory system with the lowest possible latency. For those processors that can handle multiple outstanding memory requests (vector processors and high-end CPUs), it is important not only to reduce latency but also to increase bandwidth (over what is possible by latency reduction alone) by designing a memory system that is capable of servicing multiple requests simultaneously.

Memory hierarchies provide decreased average latency and reduced bandwidth requirements, whereas parallel or **interleaved** memories provide higher bandwidth.

18.2 Memory Hierarchies

Technology does not permit memories that are cheap, large, and fast. By recognizing the nonrandom nature of memory requests, and emphasizing the *average* rather than worst-case latency, it is possible to implement a hierarchical memory system that performs well. A small amount of very fast memory, placed in front of a large, slow memory, can be designed to satisfy most requests at the speed of the small memory. This, in fact, is the primary motivation for the use of registers in the CPU: in this case, the programmer makes sure that the most commonly accessed variables are allocated to registers.

A variety of techniques, using hardware, software, or a combination of the two, can be employed to ensure that most memory references are satisfied by the faster memory. The foremost of these techniques is the exploitation of the *locality of reference* principle. This principle captures the fact that some memory locations are referenced much more frequently than others. *Spatial locality* is the property that an access to a given memory location greatly increases the probability that neighboring locations will be accessed immediately. This is largely, but not exclusively, a result of the tendency to access memory locations sequentially. *Temporal locality* is the property that an access to a given memory location greatly increases the probability that the same location will be accessed again soon. This is largely, but not exclusively, a result

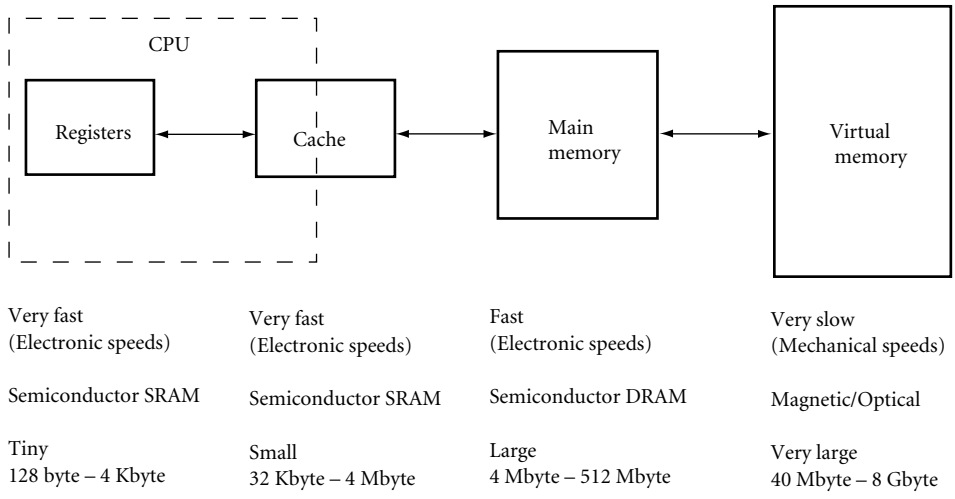


FIGURE 18.2 A memory hierarchy. (*Source:* Dorf, R. C. 1992. *The Electrical Engineering Handbook*, 1st ed., p. 1932. CRC Press, Inc., Boca Raton, FL.)

of the high frequency of programs' looping behavior. Particularly for temporal locality, a good predictor of the future is the past; the longer a variable has gone unreferenced, the less likely it is to be accessed soon.

Figure 18.2 depicts a common construction of a memory hierarchy. At the top of the hierarchy are the CPU registers, which are small and extremely fast. The next level down in the hierarchy is a special, high-speed semiconductor memory, known as a **cache memory**. The cache can actually be divided into multiple distinct levels; most current systems have between one and three levels of cache. Some of the levels of cache may be on the CPU chip itself, they may be on the same module as the CPU, or they may all be entirely distinct. Below the cache is the conventional memory, referred to as *main memory*, or *backing storage*. Like a cache, main memory is semiconductor memory, but it is slower, cheaper, and denser than a cache. Below the main memory is the virtual memory, which is generally stored on magnetic or optical disks. Accessing the virtual memory can be tens of thousands of times slower than accessing the main memory, since it involves moving, mechanical parts.

As requests go deeper into the memory hierarchy, they encounter levels that are larger (in terms of capacity) and slower than the higher levels (moving left to right in Figure 18.2). In addition to size and speed, the bandwidth between adjacent levels in the memory hierarchy is smaller for the lower levels. The bandwidth between the registers and top cache level, for example, is higher than that between cache and main memory or between main memory and virtual memory. Since each level presumably intercepts a fraction of the requests, the bandwidth to the level below need not be as great as that to the intercepting level.

A useful performance parameter is the *effective latency*. If the needed word is found in a level of the hierarchy, it is a *hit*; if a request must be sent to the next lower level, the request is said to *miss*. If the latency L_{HIT} is known in the case of a hit and the latency in the case of a miss is L_{MISS}, the effective latency for that level in the hierarchy can be determined from the *hit ratio* (H), the fraction of memory accesses that are hits:

$$L_{\text{average}} = L_{\text{HIT}} H + L_{\text{MISS}} (1 - H) \tag{18.2}$$

The portion of memory accesses that miss is called the *miss ratio* ($M = 1 - H$). The hit ratio is strongly influenced by the program being executed, but it is largely independent of the ratio of cache size to memory size. It is not uncommon for a cache with a capacity of a few thousand bytes to exhibit a hit ratio greater than 90%.

18.3 Cache Memories

The basic unit of construction of a semiconductor memory system is a *module* or *bank*. A memory bank, constructed from several memory chips, can service a single request at a time. The time that a bank is busy servicing a request is called the *bank busy time*. The bank busy time limits the bandwidth of a memory bank. Both caches and main memories are constructed in this fashion, although caches have significantly shorter bank busy times than do main memory banks.

The hardware can dynamically allocate parts of the cache memory for addresses deemed most likely to be accessed soon. The cache contains only redundant copies of the address space, which is wholly contained in the main memory. The cache memory is *associative*, or *content-addressable*. In an associative memory, the address of a memory location is stored, along with its content. Rather than reading data directly from a memory location, the cache is given an address and responds by providing data which may or may not be the data requested. When a cache miss occurs, the memory access is then performed with respect to the backing storage, and the cache is updated to include the new data.

The cache is intended to hold the most active portions of the memory, and the hardware dynamically selects portions of main memory to store in the cache. When the cache is full, bringing in new data must be matched by deleting old data. Thus a strategy for cache management is necessary. Cache management strategies exploit the principle of locality. Spatial locality is exploited by the choice of what is brought into the cache. Temporal locality is exploited by the choice of which block is removed. When a cache miss occurs, hardware copies a large, contiguous block of memory into the cache, which includes the requested word. This fixed-size region of memory, known as a cache *line* or *block*, may be as small as a single word, or up to several hundred bytes. A block is a set of contiguous memory locations, the number of which is usually a power of two. A block is said to be *aligned* if the lowest address in the block is exactly divisible by the block size. That is to say, for a block of size B beginning at location A, the block is aligned if

$$A \bmod B = 0 \tag{18.3}$$

Conventional caches require that all blocks be aligned.

When a block is brought into the cache, it is likely that another block must be evicted. The selection of the evicted block is based on an attempt to capture temporal locality. Since prescience is difficult to achieve, other methods are generally used to predict future memory accesses. A least-recently-used (LRU) policy is often the basis for the replacement choice. Other replacement policies are sometimes used, particularly because true LRU replacement requires extensive logic and hardware bookkeeping.

The cache often comprises two conventional memories: the data memory and the tag memory, shown in Figure 18.3. The address of each cache line contained in the data memory is stored in the tag memory, as well as other information (*state* information), particularly the fact that a valid cache line is present. The state also keeps track of which cache lines the processor has modified. Each line contained in the data memory is allocated a corresponding entry in the tag memory to indicate the full address of the cache line.

The requirement that the cache memory be associative (content-addressable) complicates the design. Addressing data by content is inherently more complicated than by its address. All the tags must be compared concurrently, of course, because the whole point of the cache is to achieve low latency. The cache can be made simpler, however, by introducing a mapping of memory locations to cache cells. This mapping limits the number of possible cells in which a particular line may reside. The extreme case is known as *direct mapping*, in which each memory location is mapped to a single location in the cache. Direct mapping makes many aspects of the design simpler, since there is no choice of where the line might reside and no choice as to which line must be replaced. Direct mapping, however, can result in poor utilization of the cache when two memory locations are alternately accessed and must share a single cache cell.

A hashing algorithm is used to determine the cache address from the memory address. The conventional mapping algorithm consists of a function with the form

$$A_{\text{cache}} = \frac{A_{\text{memory}} \bmod \text{cache\_size}}{\text{cache\_line\_size}} \tag{18.4}$$

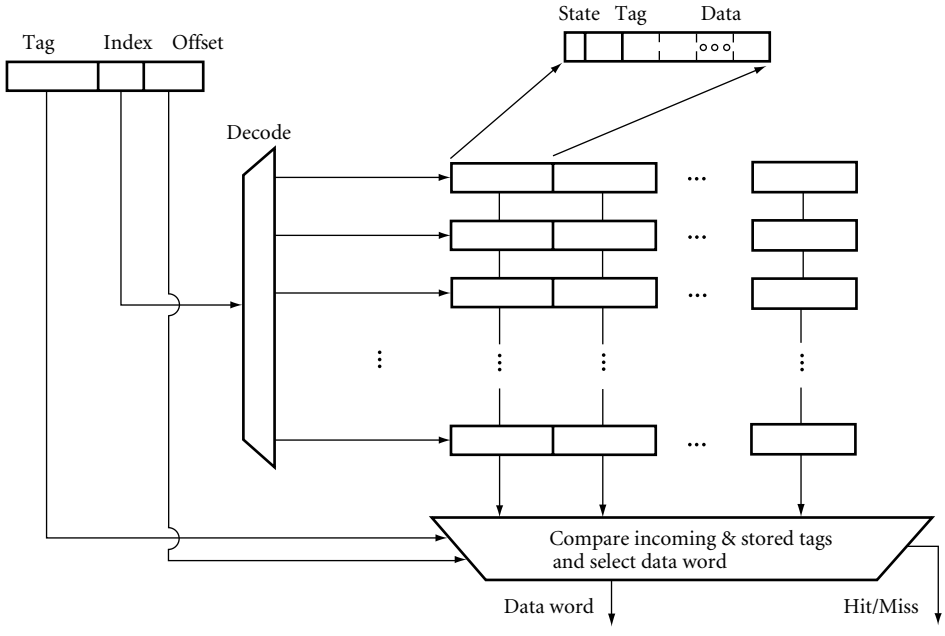


FIGURE 18.3 Components of a cache memory. (*Source:* Hill, M. D. 1988. A case for direct-mapped caches. *IEEE Comput.* 21(12):27. IEEE Computer Society, New York. With permission.)

where A_{cache} is the address within the cache for main memory location A_{memory}, cache_size is the capacity of the cache in addressable units (usually bytes), and cache_line_size is the size of the cache line in addressable units. Since the hashing function is simple bit selection, the tag memory need contain only the part of the address not implied by the result of the hashing function. That is,

$$A_{\text{tag}} = A_{\text{memory}} \text{ div size\_of\_cache} \tag{18.5}$$

where A_{tag} is stored in the tag memory and *div* is the integer divide operation. In testing for a match, the complete address of a line stored in the cache can be inferred from the tag and its storage location within the cache.

A *two-way set-associative* cache maps each memory location into either of two locations in the cache, and can be constructed essentially as two identical direct-mapped caches. However, both caches must be searched at each memory access and the appropriate data selected and multiplexed on a tag match (hit). On a miss, a choice must be made between the two possible cache lines as to which is to be replaced. A single LRU bit can be saved for each such pair of lines to remember which line has been accessed more recently. This bit must be toggled to the current state each time either of the cache lines is accessed.

In the same way, an *M-way associative* cache maps each memory location into any of M memory locations in the cache and can be constructed from M identical direct-mapped caches. The problem of maintaining the LRU ordering of M cache lines quickly becomes hard, however, since there are $M!$ possible orderings, so it takes at least

$$\lceil \log_2(M!) \rceil \tag{18.6}$$

bits to store the ordering. In practice, this requirement limits true LRU replacement to three- or four-way set associativity.

Figure 18.4 shows how a cache is organized into sets, blocks, and words. The cache shown is a 2-Kbyte, four-way set-associative cache, with 16 sets. Each set consists of four blocks. The cache block size in this example is 32 bytes, so each block contains eight 4-byte words. Also depicted at the bottom of Figure 18.4

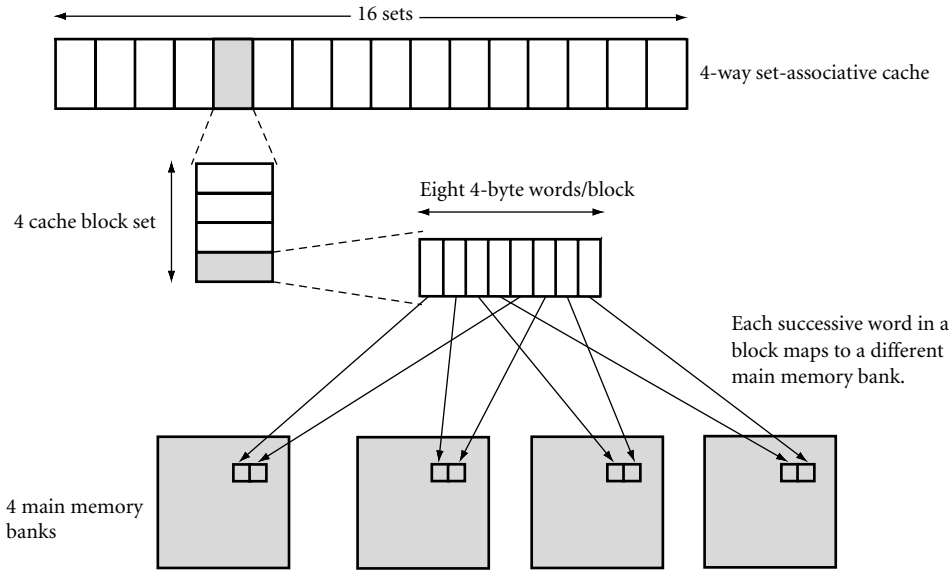


FIGURE 18.4 Organization of a cache.

is a four-way interleaved main memory system (see the next section for details). Each successive word in the cache block maps into a different main memory bank. Because of the cache's mapping restrictions, each cache block obtained from main memory will be loaded into its corresponding set, but it may appear anywhere within that set.

Write operations require special handling in the cache. If the main memory copy is updated with each write operation — a technique known as *write-through* or *store-through* — the writes may force operations to stall while the write operations are completing. This can happen after a series of write operations even if the processor is allowed to proceed before the write to the memory has completed. If the main memory copy is not updated with each write operation — a technique known as *write-back* or *copy-back* or *deferred writes* — the main memory locations become stale, that is, memory no longer contains the correct values and must not be relied upon to provide data. This is generally permissible, but care must be exercised to make sure that it is always updated before the line is purged from the cache and that the cache is never bypassed. Such a bypass could occur with *direct memory access* (DMA), in which the I/O system writes directly into main memory without the involvement of the processor.

Even for a system that implements a write-through policy, care must be exercised if memory requests may bypass the cache. While the main memory is never stale, a write that bypasses the cache, such as from I/O, could have the effect of making the cached copy stale. A later access by the CPU could then provide an incorrect value. This can be avoided only by making sure that cached entries are invalidated even if the cache is bypassed. The problem is relatively easy to solve for a single processor with I/O, but it becomes very difficult to solve for multiple processors, particularly so if multiple caches are involved as well. This is known in general as the cache *coherence* or *consistency* problem.

The cache exploits spatial locality by loading an entire cache line after a miss. This tends to result in bursty traffic to the main memory, since most accesses are filtered out by the cache. After a miss, however, the memory system must provide an entire line at once. Cache memory nicely complements an interleaved, high-bandwidth main memory (described in the next section), since a cache line can be interleaved across many banks in a regular manner — thus avoiding memory conflicts and being loaded rapidly into the cache. The example of main memory shown in Figure 18.4 can provide the entire cache line with two parallel memory accesses.

Conventional caches traditionally could not accept requests while they were servicing a miss request. In other words, they *locked-up* or *blocked* when servicing a miss. The growing penalty for cache misses has

made it necessary for high-end commodity memory systems to continue to accept (and service) requests from the processor while a miss is being serviced. Some systems are able to service multiple miss requests simultaneously. To allow this mode of operation, the cache design is *lockup-free* or *nonblocking* [Kroft 1981]. Lockup-free caches have one structure for each simultaneous outstanding miss that they can service. This structure holds the information necessary to correctly return the loaded data to the processor, even if the misses come back in a different order than that in which they were sent.

Two factors drive the existence of multiple levels of cache memory in the memory hierarchy: access times and a limited number of transistors on the CPU chip. Larger banks with greater capacity are slower than smaller banks. If the time needed to access the cache limits the clock frequency of the CPU, then the first-level cache size may need to be constrained. Much of the benefit of a large cache may be obtained by placing a small first-level cache above a larger second-level cache; the first is accessed quickly, and the second holds more data close to the processor. Since many modern CPUs have caches on the CPU chip itself, the size of the cache is limited by the CPU silicon real estate. Some CPU designers have assumed that system designers will add large off-chip caches to the one or two levels of caches on the processor chip. The complexity of this part of the memory hierarchy may continue to grow as main memory access penalties continue to increase.

Caches that appear on the CPU chip are manufactured by the CPU vendor. Off-chip caches, however, are a commodity part sold in large volume. An incomplete list of major cache manufacturers includes Hitachi, IBM Micro, Micron, Motorola, NEC, Samsung, SGS-Thomson, Sony, and Toshiba. Although most personal computers and all major workstations now contain caches, very high-end machines (such as multimillion-dollar supercomputers) do not usually have caches. These ultraexpensive computers can afford to implement their main memory in a comparatively fast semiconductor technology such as static RAM (SRAM) and can afford so many banks that cacheless bandwidth out of the main memory system is sufficient. Massively parallel processors (MPPs), however, are often constructed out of workstation-like nodes to reduce cost. MPPs therefore contain cache hierarchies similar to those found in the workstations on which the nodes of the MPPs are based.

Cache sizes have been steadily increasing on personal computers and workstations. Intel Pentium-based personal computers come with 8 Kbyte each of instruction and data caches. Two of the Pentium chip sets, manufactured by Intel and OPTi, allow level-two caches ranging from 256 to 512 Kbyte and 64 Kbyte to 2 Mbyte, respectively. The newer Pentium Pro systems also have 8-Kbyte first-level instruction and data caches, but they also have either a 256-Kbyte or a 512-Kbyte second-level cache on the same module as the processor chip. Higher-end workstations — such as DEC Alpha 21164-based systems — are configured with substantially more cache. The 21164 also has 8-Kbyte first-level instruction and data caches. Its second-level cache is entirely on chip and is 96 Kbyte. The third-level cache is off chip, and can have a size ranging from 1 to 64 Mbyte.

For all desktop machines, cache sizes are likely to continue to grow — although the rate of growth compared to processor speed increases and main memory size increases is unclear.

18.4 Parallel and Interleaved Main Memories

Main memories are composed of a series of semiconductor memory chips. A number of these chips, like caches, form a *bank*. Multiple memory banks can be connected together to form an **interleaved** (or parallel) memory system. Since each bank can service a request, an interleaved memory system with K banks can service K requests simultaneously, increasing the peak bandwidth of the memory system to K times the bandwidth of a single bank. In most interleaved memory systems, the number of banks is a power of two, that is, $K = 2^k$. An n-bit memory word address is broken into two parts: a k-bit bank number and an m-bit address of a word within a bank. Though the k bits used to select a bank number could be any k bits of the n-bit word address, typical interleaved memory systems use the low-order k address bits to select the bank number; the higher order $m = n - k$ bits of the word address are used to access a word in the selected bank. The reason for using the low-order k bits will be discussed shortly. An interleaved memory system which uses the low-order k bits to select the bank is referred to as a *low-order* or a *standard* interleaved memory.

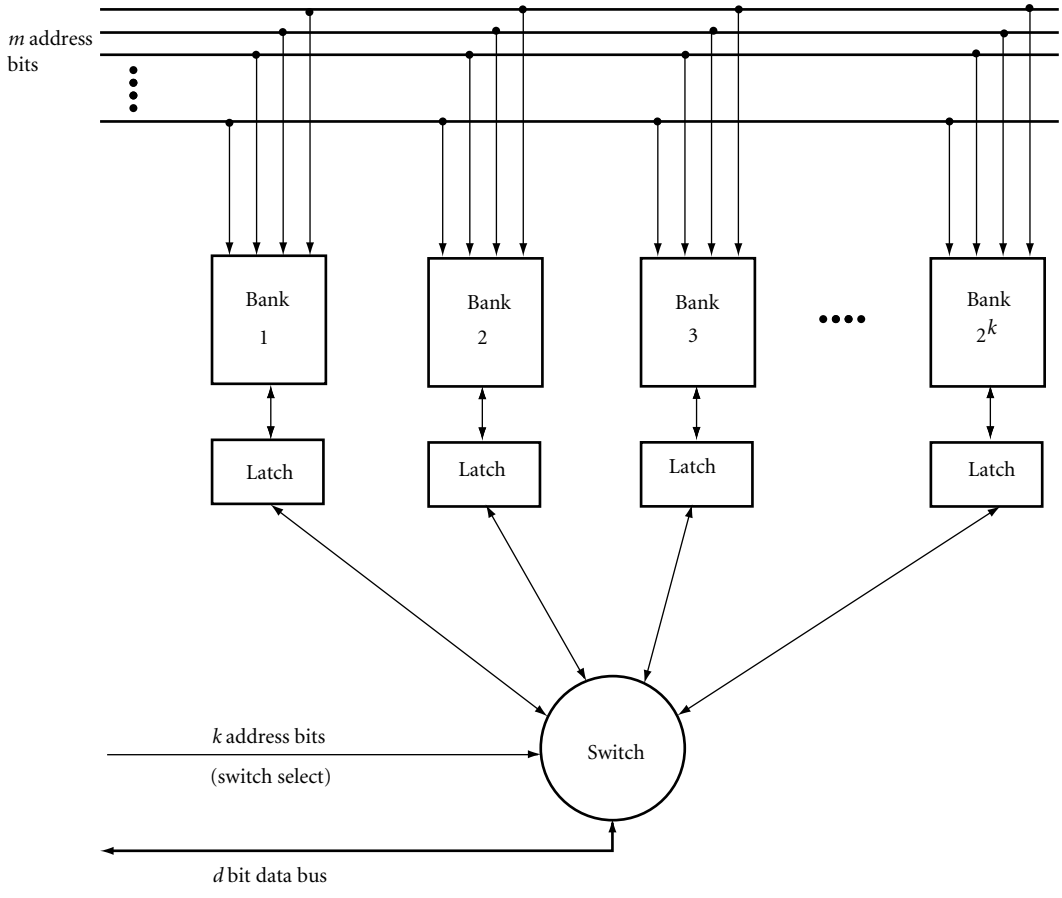


FIGURE 18.5 A simple interleaved memory system. (*Source:* Adapted from Kogge, P. M. 1981. *The Architecture of Pipelined Computers,* 1st ed., p. 41. McGraw–Hill, New York.)

There are two ways of connecting multiple memory banks: *simple interleaving* and *complex interleaving*. Sometimes simple interleaving is also referred to as *interleaving*, and complex interleaving is referred to as *banking*.

Figure 18.5 shows the structure of a simple interleaved memory system. m address bits are simultaneously supplied to every memory bank. All banks are also connected to the same read/write control line (not shown in Figure 18.5). For a read operation, the banks start the read operation and deposit the data in their latches. Data can then be read from the latches, one by one, by appropriately setting the switch. Meanwhile, the banks could be accessed again to carry out another read or write operation. For a write operation, the latches are loaded one by one. When all the latches have been written, their contents can be written into the memory banks by supplying m bits of address (they will be written into the same word in each of the different banks). In a simple interleaved memory, all banks are cycled at the same time; each bank starts and completes its individual operations at the same time as every other bank; a new memory cycle can start (for all banks) once the previous cycle is complete. Timing details of the accesses can be found in *The Architecture of Pipelined Computers* [Kogge 1981].

One use of a simple interleaved memory system is to back up a cache memory. To do so, the memory must be able to read blocks of contiguous words (a cache block) and supply them to the cache. If the low-order k bits of the address are used to select the bank number, then consecutive words of the block reside in different banks; they can all be read in parallel and supplied to the cache one by one. If some

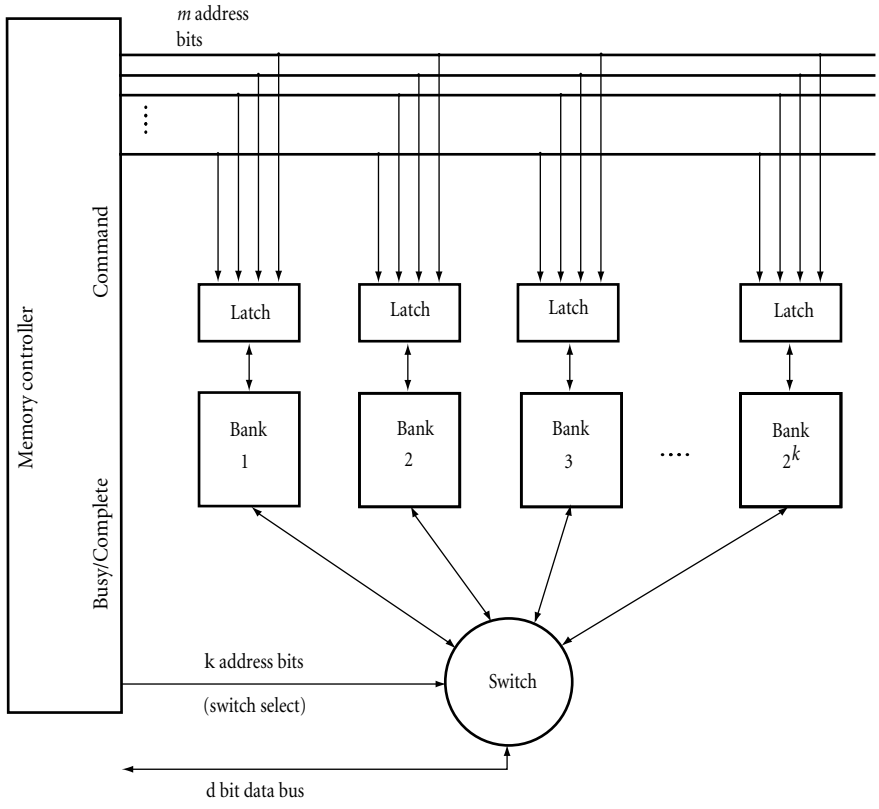


FIGURE 18.6 A complex interleaved memory system. (*Source:* Adapted from Kogge, P. M. 1981. *The Architecture of Pipelined Computers,* 1st ed., p. 42. McGraw–Hill, New York.)

other address bits are used for bank selection, then multiple words from the block might fall in the same memory bank, requiring multiple accesses to the same bank to fetch the block.

Figure 18.6 shows the structure of a complex interleaved memory system. In such a system, each bank is set up to operate on its own, independent of the other banks' operation. In this example, bank 1 could carry out a read operation on a particular memory address while bank 2 carries out a write operation on a completely unrelated memory address. (Contrast this with the operation in a simple interleaved memory where all banks are carrying out the same operation, read or write, and the locations accessed within each bank represent a contiguous block of memory.) Complex interleaving is accomplished by providing an address latch and a read/write command line for each bank. The *memory controller* handles the overall operation of the interleaved memory. The processing unit submits the memory request to the memory controller, which determines the bank that needs to be accessed. The controller then determines if the bank is busy (by monitoring a busy line for each bank). The controller holds the request if the bank is busy, submitting it later when the bank is available to accept the request. When the bank responds to a read request, the switch is set by the controller to accept the request from the bank and forward it to the processing unit. Timing details of the accesses can be found in *The Architecture of Pipelined Computers* [Kogge 1981].

A typical use of a complex interleaved memory system is in a *vector processor*. In a vector processor, the processing units operate on a vector, for example a portion of a row or a column of a matrix. If consecutive elements of a vector are present in different memory banks, then the memory system can sustain a bandwidth of one element per clock cycle. By arranging the data suitably in memory and using standard interleaving (for example, storing the matrix in row-major order will place consecutive elements

in consecutive memory banks), the vector can be accessed at the rate of one element per clock cycle as long as the number of banks is greater than the bank busy time.

Memory systems that are built for current machines vary widely, the price and purpose of the machine being the main determinant of the memory system design. The actual memory chips, which are the components of the memory systems, are generally commodity parts built by a number of manufacturers. The major commodity DRAM manufacturers include (but are certainly not limited to) Hitachi, Fujitsu, LG Semicon, NEC, Oki, Samsung, Texas Instruments, and Toshiba.

The low end of the price/performance spectrum is the personal computer, presently typified by Intel Pentium systems. Three of the manufacturers of Pentium-compatible chip sets (which include the memory controllers) are Intel, OPTi, and VLSI Technologies. Their controllers provide for memory systems that are simply interleaved, all with minimum bank depths of 256 Kbyte and maximum system sizes of 192 Mbyte, 128 Mbyte, and 1 Gbyte, respectively.

Both higher-end personal computers and workstations tend to have more main memory than the lower-end systems, although they usually have similar upper limits. Two examples of such systems are workstations built with the DEC Alpha 21164 and servers built with the Intel Pentium Pro. The Alpha systems, using the 21171 chip set, are limited to 128 Mbyte of main memory using 16 Mbit DRAMs, although they will be expandable to 512 Mbyte when 64-Mbit DRAMs are available. Their memory systems are eight-way simply interleaved, providing 128 bits per DRAM access. The Pentium Pro systems support slightly different features. The 82450KX and 82450GX chip sets include memory controllers that allow reads to bypass writes (performing writes when the memory banks are idle). These controllers can also buffer eight outstanding requests simultaneously. The 82450KX controller permits one- or two-way interleaving and up to 256 Mbyte of memory when 16-Mbit DRAMs are used. The 82450GX chip set is more aggressive, allowing up to four separate (complex-interleaved) memory controllers, each of which can be up to four-way interleaved and have up to 1 Gbyte of memory (again with 16-Mbit DRAMs).

Interleaved memory systems found in high-end *vector supercomputers* are slight variants on the basic complex interleaved memory system of Figure 18.6. Such memory systems may have hundreds of banks, with multiple memory controllers that allow multiple independent memory requests to be made every clock cycle. Two examples of modern vector supercomputers are the Cray T-90 series and the NEC SX series. The Cray T-90 models come with varying numbers of processors — up to 32 in the largest configuration. Each of these processors is coupled with 256 Mbyte of memory, split into 16 banks of 16 Mbyte each. The T-90 has complex interleaving among banks. The largest configuration (the T-932) has 32 processors, for a total of 512 banks and 8 Gbyte of main memory. The T-932 can provide a peak of 800-GByte/s bandwidth out of its memory system. NEC's SX-4 product line, their most recent vector supercomputer series, has numerous models. Their largest single-node model (with one processor per node) contains 32 processors, with a maximum of 8 Gbyte of memory, and a peak bandwidth of 512 Gbyte/s out of main memory. Although the sizes of the memory systems are vastly different between workstations and vector machines, the techniques that both use to increase total bandwidth and minimize bank conflicts are similar.

18.5 Virtual Memory

Cache memory contains portions of the main memory in dynamically allocated cache lines. Since the data portion of the cache memory is itself a conventional memory, each line present in the cache has two addresses associated with it: its main memory address and its cache address. Thus, the main memory address of a word can be divorced from a particular storage location and abstractly thought of as an element in the address space. The use of a two-level hierarchy — consisting of main memory and a slower, larger disk storage device — evolved by making a clear distinction between the address space and the locations in memory. An address generated during the execution of a program is known as a *virtual address*, which must be translated to a *physical address* before it can be accessed in main memory. The total address space is only an abstraction.

A **virtual memory** address is mapped to a physical address, which indicates the location in main memory where the data actually reside [Denning 1970]. The mapping is maintained through a structure called the *page table*, which is maintained in software by the operating system. Like the tag memory of a cache memory, the page table is accessed through a virtual address to determine the physical (main memory) address of the entry. Unlike the tag memory, however, the table is usually sorted by virtual addresses, making the translation process a simple matter of an extra memory access to determine the physical address of the desired item. A system maintaining the page table in a way analogous to a cache tag memory is said to have *inverted page tables*. In addition to the physical address mapped to a virtual page, and an indication of whether the page is present at all, a page table entry often contains other information. For example, the page table may contain the location on the disk where each block of data is stored when not present in main memory.

The virtual memory can be thought of as a collection of blocks. These blocks are often aligned and of fixed size, in which case they are known as *pages*. Pages are the unit of transfer between the disk and main memory and are generally larger than a cache line — usually thousands of bytes. A typical page size for machines in 1995 is 4 Kbyte. A page's virtual address can be broken into two parts: a virtual page number and an offset. The page number specifies which page is to be accessed, and the page offset indicates the distance from the beginning of the page to the indicated address.

A physical address can also be broken into two parts: a physical page number (also called a *page frame* number) and an offset. This mapping is done at the level of pages, so the page table can be indexed by means of the virtual page number. The page frame number is contained in the page table and is read out during the translation along with other information about the page. In most implementations the page offset is the same for a virtual address and the physical address to which it is mapped.

The virtual memory hierarchy is different than the cache/main memory hierarchy in a number of respects, resulting primarily from the fact that there is a much greater difference in latency between accesses to the disk and to main memory. While a typical latency ratio for cache and main memory is one order of magnitude (main memory has a latency ten times larger than the cache), the latency ratio between disk and main memory is often four orders of magnitude or more. This large ratio exists because the disk is a mechanical device — with a latency partially determined by velocity and inertia — whereas main memory is limited only by electronic and energy constraints. Because of the much larger penalty for a page miss, many design decisions are affected by the need to minimize the frequency of misses. When a miss does occur, the processor could be idle for a period during which it could execute tens of thousands of instructions. Rather than stall during this time, as may occur upon a cache miss, the processor invokes the operating system and may switch to a different task. Because the operating system is being invoked anyway, it is convenient to rely on it to set up and maintain the page table, unlike cache memory, where it is done entirely in hardware. The fact that this accounting occurs in the operating system enables the system to use virtual memory to enforce protection on the memory. This ensures that no program can corrupt the data in memory that belong to any other program.

Hardware support provided for a virtual memory system generally includes the ability to translate the virtual addresses provided by the processor into the physical addresses needed to access main memory. Thus, only upon a virtual-address miss is the operating system invoked. An important aspect of a computer that implements virtual memory, however, is the necessity of freezing the processor at the point at which a miss occurs, servicing the page table fault, and later returning to continue the execution as if no page fault had occurred. This requirement means either that it must be possible to halt execution at any point — including possibly in the middle of a complex instruction — or that it must be possible to guarantee that all memory accesses will be to pages resident in main memory.

As described above, virtual memory requires two memory accesses to fetch a single entry from memory: one into the page table to map the virtual address into the physical address, and the second to fetch the actual data. This process can be sped up in a variety of ways. First, a special-purpose cache memory to store the active portion of the page table can be used to speed up the first access. This special-purpose cache is usually called a *translation lookaside buffer* (TLB). Second, if the system also employs a cache memory, it may be possible to overlap the access of the cache memory with the access to the TLB, ideally

allowing the requested item to be accessed in a single cache access time. The two accesses can be fully overlapped if the virtual address supplies sufficient information to fetch the data from the cache before the virtual-to-physical address translation has been accomplished. This is true for an M-way set associative cache of capacity C if the following relationship holds:

$$\text{Page\_size} \geq \frac{C}{M} \tag{18.7}$$

For such a cache, the index into the cache can be determined strictly from the page offset. Since the virtual page offset is identical to the physical page offset, no translation is necessary, and the cache can be accessed concurrently with the TLB. The physical address must be obtained before the tag can be compared, of course.

An alternative method applicable to a system containing both virtual memory and a cache is to store the virtual address in the tag memory instead of the physical address. This technique introduces consistency problems in virtual memory systems that either permit more than a single address space, or allow a single physical page to be mapped to more than one single virtual page. This problem is known as the *aliasing* problem.

Chapter 85 is devoted to virtual memory and contains significantly more material on this topic for the interested reader.

18.6 Research Issues

Research is occurring on all levels of the memory hierarchy. At the register level, researchers are exploring techniques to provide more registers than are architecturally visible to the compiler. A large volume of work exists (and is occurring) for cache optimizations and alternative cache organizations. For instance, modern processors now commonly split the top level of the cache into separate physical caches, one for instructions (code) and one for program data. Due to the increasing cost of cache misses (in terms of processor cycles), some research trades off increasing the complexity of the cache for reducing the miss rate. Two examples of cache research from opposite ends of the hardware/software spectrum are *blocking* [Lam et al. 1991] and *skewed-associative caches* [Seznec 1993]. Blocking is a software technique in which the programmer or compiler reorganizes algorithms to work on subsets of data that are smaller than the cache instead of streaming entire large data structures repeatedly through the cache. This reorganization greatly improves temporal locality. The skewed-associative cache is one example of a host of hardware techniques that map blocks into the cache differently, with the goal of reducing misses from set conflicts. In skewed-associative caches, either one of two hashing functions may determine where a block should be placed in the cache, as opposed to just the one hashing function (low-order index bits) that traditional caches use. An important cache-related research topic is *prefetching* [Mowry et al. 1992], in which the processor issues requests for data well before the data are actually needed. Speculative prefetching is also a current research topic. In speculative prefetching, prefetches are issued based on guesses as to which data will be needed soon. Other cache-related research examines placing special structures in parallel with the cache, trying to optimize for workloads that do not lend themselves well to caches. Stream buffers [Jouppi 1990] are one such example. A stream buffer automatically detects when a linear access through a data structure occurs. The stream buffer issues multiple sequential prefetches upon detection of a linear array access.

Much of the ongoing research on main memory involves improving the bandwidth from the memory system without greatly increasing the number of banks. Multiple banks are expensive, particularly with the large and growing capacity of modern DRAM chips. Rambus [Rambus 1992] and Ramlink [IEEE Computer Society 1993] are two such examples.

Research issues associated with improving the performance of the virtual memory system fall into the domain of operating system research. One proposed strategy for reducing page faults allows each running program to specify its own page replacement algorithm, enabling each program to optimize the choice of page replacements based on its reference pattern [Engler et al. 1995]. Other recent research focuses on

improving the performance of the TLB. Two techniques for doing this are the use of a two-level TLB (the motivation is similar to that for a two-level cache) and the use of superpages [Talluri and Hill 1994]. With superpages, each TLB entry may represent a mapping for more than one consecutive page, thus increasing the total address range that a fixed number of TLB entries may cover.

18.7 Summary

A computer's memory system is the repository for all the information that the CPU uses and produces. A perfect memory system is one that can immediately supply any datum that the CPU requests. This ideal memory is not implementable, however, as the three factors of memory — capacity, speed, and cost — are directly in opposition.

By staging smaller, faster memories in front of larger, slower, and cheaper memories, the performance of the memory system may approach that of a perfect memory system — at a reasonable cost. The memory hierarchies of modern general-purpose computers generally contain registers at the top, followed by one or more levels of cache memory, main memory, and virtual memory on a magnetic or optical disk.

Performance of a memory system is measured in terms of latency and bandwidth. The latency of a memory request is how long it takes the memory system to produce the result of the request. The bandwidth of a memory system is the rate at which the memory system can accept requests and produce results. The memory hierarchy improves average latency by quickly returning results that are found in the higher levels of the hierarchy. The memory hierarchy generally reduces bandwidth requirements by intercepting a fraction of the memory requests at higher levels of the hierarchy. Some machines — such as high-performance vector machines — may have fewer levels in the hierarchy, increasing memory cost for better predictability and performance. Some of these machines contain no caches at all, relying on large arrays of main memory banks to supply very high bandwidth, with pipelined accesses of operands that mitigate the adverse performance impact of long latencies.

Cache memories are a general solution to improving the performance of a memory system. Although caches are smaller than typical main memory sizes, they ideally contain the most frequently accessed portions of main memory. By keeping the most heavily used data near the CPU, caches can service a large fraction of the requests without accessing main memory (the fraction serviced is called the hit rate). Caches assume locality of reference to work well transparently — they assume that accessed memory words will be accessed again quickly (temporal locality) and that memory words adjacent to an accessed word will be accessed soon after the access in question (spatial locality). When the CPU issues a request for a datum not in the cache (a cache miss), the cache loads that datum and some number of adjacent data (a cache block) into itself from main memory.

To reduce cache misses, some caches are associative — a cache may place a given block in one of several places, collectively called a set. This set is content-addressable; a block may or may not be accessed based on an address tag, one of which is coupled with each block. When a new block is brought into a set and the set is full, the cache's replacement policy dictates which of the old blocks should be removed from the cache to make room for the new block. Most caches use an approximation of least-recently-used (LRU) replacement, in which the block last accessed farthest in the past is the one that the cache replaces.

Main memory, or backing store, consists of banks of dense semiconductor memory. Since each memory chip has a small off-chip bandwidth, rows of these chips are placed together to form a bank, and multiple banks are used to increase the total bandwidth out of main memory. When a bank is accessed, it remains busy for a period of time, during which the processor may make no other accesses to that bank. By increasing the number of interleaved (parallel) banks, the chance that the processor issues two conflicting requests to the same bank is reduced.

Systems generally require a greater number of memory locations than are available in the main memory (i.e., a larger address space). The entire address space that the CPU uses is kept on large magnetic or optical disks; this is called the virtual address space, or virtual memory. The most frequently used sections of the virtual memory are kept in main memory (physical memory) and are moved back and forth in units

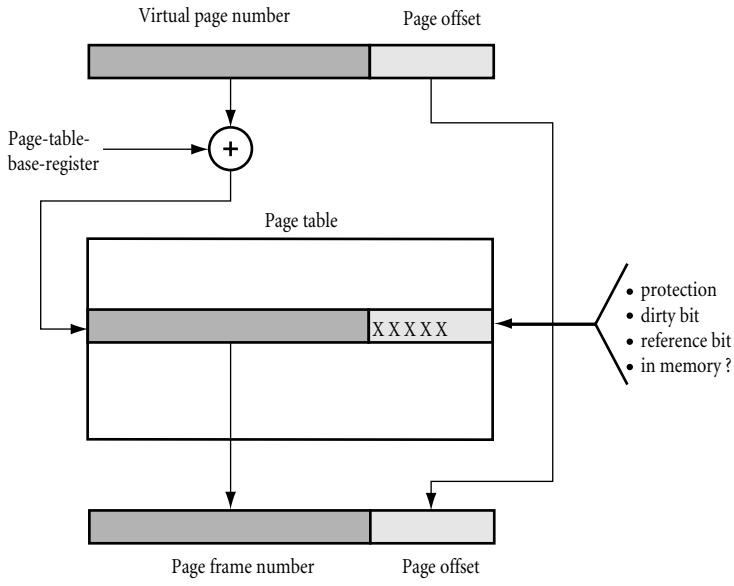


FIGURE 18.7 Virtual-to-physical address translation. (*Source:* Dorf, R. C. 1992. *The Electrical Engineering Handbook*, 1st ed., p. 1935. CRC Press, Inc., Boca Raton, FL.)

called pages. The place at which a virtual address lies in main memory is called its physical address. Since a much larger address space (virtual memory) is mapped onto a much smaller one (physical memory), the CPU must translate the memory addresses issued by a program (virtual addresses) into their corresponding locations in physical memory (physical addresses) (see Figure 18.7). This mapping is maintained in a memory structure called the page table. When the CPU attempts to access a virtual address that does not have a corresponding entry in physical memory, a page fault occurs. Since a page fault requires an access to a slow mechanical storage device (such as a disk), the CPU usually switches to a different task while the needed page is read from the disk.

Every memory request issued by the CPU requires an address translation, which in turn requires an access to the page table stored in memory. A translation lookaside buffer (TLB) is used to reduce the number of page table lookups. The most frequent virtual-to-physical mappings are kept in the TLB, which is a small associative memory tightly coupled with the CPU. If the needed mapping is found in the TLB, the translation is performed quickly and no access to the page table need be made. Virtual memory allows systems to run larger or more programs than are able to fit in main memory, enhancing the capabilities of the system.

Defining Terms

Bandwidth: The rate at which the memory system can service requests.

Cache memory: A small, fast, redundant memory used to store the most frequently accessed parts of the main memory.

Interleaving: Technique for connecting multiple memory modules together in order to improve the bandwidth of the memory system.

Latency: The time between the initiation of a memory request and its completion.

Memory hierarchy: Successive levels of different types of memory, which attempt to approximate a single large, fast, and cheap memory structure.

Virtual memory: A memory space implemented by storing the more frequently accessed data in main memory and less frequently accessed data on disk.