returned by reads are consistent with some total ordering on all the operations that respects the relative orderings of the operations. These definitions were introduced by Lamport [Lam86].

More specifically, we are interested in *one-write algorithms*—implementations with the property that every WRITE to the logical register requires only one write to a physical register and no reads of physical registers[1]. Since bounds on the number of physical accesses per logical access can be converted into time bounds for the logical access, a one-write algorithm would have time-efficient logical WRITEs, perhaps an important characteristic for applications in which WRITEs outnumber READs.

In this part of our thesis, we present a one-write algorithm for implementing a k-ary regular register out of binary regular registers. Clearly this algorithm is optimal in the number of physical writes per logical WRITE. The best previous upper bound was $\lceil \log k \rceil$ writes per WRITE, due to Chaudhuri and Welch [CW90]. The algorithm is simple to describe using the complete graph whose nodes are labeled with the logical values. Its correctness proof is based on properties of paths in this graph. The same algorithm also implements a k-ary atomic register out of binary atomic registers.

One drawback of our algorithm is that it requires $C(k,2) = k(k-1)/2 = O(k^2)$ physical registers. The best previous lower bound on the number of physical registers for a k-ary regular implementation was k [CW90], for any number of physical writes per logical WRITE. Theoretically, binary to k-ary regular implementations are inherently expensive in the amount of hardware required. In this chapter we show two improved lower bounds on the number of physical registers in any one-write algorithm in which the writer does not read physical registers. Each lower bound holds for a natural class of regular implementations. The first lower bound, $C(k,2)$, holds for a restricted class of implementations satisfying the *toggle* property. Since the lower bound matches the upper bound given by our algorithm, which satisfies the toggle property, our algorithm is optimal in the number of physical registers for this class. The second lower bound, $2k - 1 - \lfloor \log k \rfloor$, holds for a

[1]The names of logical operations will be capitalized in the remainder of the chapters concerning register implementations, and the names of physical operations will remain in lower case.

more general and reasonably unrestrictive class of implementations satisfying the *symmetric* property.

Our lower bounds are proved by contradiction; in both cases, the ultimate contradiction reached is a violation of the regular property. We have formalized a general technique for proving that the regular property does not hold by "fooling the reader", or constructing a scenario wherein a reader returns an incorrect value. We also developed a general transformation to convert a one-write algorithm for k values into a one-write algorithm for $k-1$ values using fewer physical registers. This transformation is used in the inductive proof of our symmetric lower bound. In proving these two lower bounds, we have developed considerable understanding of such one-write algorithms. We can prove that, for any one-write algorithm (in which the writer does not read physical registers), there is no advantage, in terms of number of physical registers, to be gained if readers write, or if different readers follow different protocols, or if a reader's protocol depends on its history. Furthermore, for symmetric algorithms, there is no advantage if a reader reads some physical registers more than once. Thus our lower bound proofs are simpler, since we assume the reader does none of the above. These results are shown with transformation techniques similar to the ones mentioned previously.

Our lower bounds also apply for the two corresponding classes of atomic implementations which prohibit readers from doing any of the above. However, in the atomic case, the restrictions on the readers are possibly too strong. We have not yet obtained any general lower bounds for atomic one-write algorithms.

In Section 2.2, we present our basic definitions. In Chapter 3 we describe the algorithm and prove it is correct with respect to regularity and atomicity. Chapter 4 consists of our lower bounds on the number of registers required by one-write algorithms. Chapter 5 contains an investigation of the necessity of the toggle or symmetric properties in one-write algorithms. We conclude in Chapter 6.

2.2 Definitions

We use a simplified form of the I/O automaton model [LT87] to describe our system.

To implement a logical register with value set V, where $|V| = k$, we compose a collection of physical registers X_j, $1 \leq j \leq m$, each with value set $\{0,1\}$, a collection of read processes RP_i, $1 \leq i \leq n$, and a single write process WP. The read and write processes implement the protocols used by the readers and writer of the logical register. Each such protocol consists of accessing certain of the physical registers and doing some local computation.

The components of our logical register implementation are the outside world, the X_j's, the RP_i's, and WP. Communication between these components takes place via **actions**. Each action is an output of one component (the component that generates it) and an input to another component. Components are modeled as state machines in which actions trigger transitions. Components have no control over when inputs occur, and thus must have a transition for every input in every state. Components do have control over when outputs occur; if an output labels a transition from a state, then the output is **enabled** in that state.

An **execution** of the implementation consists of a sequence in which state tuples (one entry for the state of each component) and actions alternate, beginning with a tuple of initial states. For each action π in the execution, π must be enabled in the preceding state of the component for which it is an output. In the following state tuple, the states of the two components for which π is an input and an output must change according to the transition functions, while the remaining components' states are unchanged.

A **schedule** is the sequence of actions in an execution.

The **logical actions** are $\mathrm{READ}(i)$, $\mathrm{RETURN}(i,v)$, $\mathrm{WRITE}(v)$, and ACK, $1 \leq i \leq n$ and $v \in V$. $\mathrm{READ}(i)$ is an input to RP_i from the outside world and $\mathrm{RETURN}(i,v)$ is an output from RP_i to the outside world. $\mathrm{WRITE}(v)$ is an input to WP from the outside world and ACK is an output from WP to the outside world. Although we do not explicitly model the outside world with a component, we do assume that for each i, the outside world and RP_i cooperate so that $\mathrm{READ}(i)$'s and $\mathrm{RETURN}(i,*)$'s strictly alternate, beginning with a READ, and that the outside world and WP cooperate so that $\mathrm{WRITE}(v)$'s and ACK's strictly alternate. This means that at most one operation is pending at a time at a given read or write process.

The **physical actions** are $\text{read}_j(i)$, $\text{return}_j(i,v)$, $\text{write}_j(v)$, and ack_j. The subscript j is between 1 and m; it indicates that X_j is the physical register being read or written. The parameter v is either 0 or 1 and indicates the value being read from or written to X_j. The parameter i is between 1 and n and indicates which of the read and write processes is reading X_j. For a fixed j, there is no parameter i for writes and acks, since there is a unique read or write process that writes X_j.

A READ(i) and its following RETURN(i,v) form a **logical operation**, as do a WRITE(v) and its following ACK. **Physical operations** are defined analogously. An operation is **pending** if its first half is present but not its second half.

We assume that the read and write processes cooperate with the physical registers so that for each i, $0 \leq i \leq n$, and each j, $1 \leq j \leq m$, $\text{read}_j(i)$ and $\text{return}_j(i,*)$ alternate beginning with a read, and analogously for writes. We also assume that no read or write process has a physical operation pending unless it has a logical operation pending.

Each physical register X_j satisfies this liveness property:

- Immediately after an input action occurs, the matching output is enabled.

A **regular physical** register satisfies:

- **Physical Regular Property.** Every physical read operation returns a value written by an overlapping write operation or by the most recent preceding write (or the initial value if there is no preceding write).

An **atomic physical** register satisfies:

- **Physical Atomic Property.** For every execution, there exists a linearization [HW90] of the execution. A **linearization** of an execution e is a sequence of operations T such that T is a permutation of the operations in e and the ordering of non-overlapping operations in T is the same as their ordering in e (two operations do not overlap if the response for one occurs before the invocation of the other one). T must be such that each read in T returns the value written by the latest preceding write in T (or the initial value if there is no preceding write).

The read and write processes must work together to implement a logical register. The liveness property for a logical register differs from that for a physical register, as discussed below. A **regular logical** register satisfies:

- **Logical Regular Property.** Every logical READ operation returns a value written by an overlapping WRITE operation or by the most recent preceding WRITE (or the initial value if there is no preceding WRITE).

An **atomic logical** register satisfies:

- **Logical Atomic Property.** For every execution e, there exists a linearization T of the operations in e. T must be such that each READ in T RETURNs the value written by the latest preceding WRITE in T (or the initial value if there is no preceding WRITE).

The liveness property for a logical register is that the implementation must be *wait-free*. Informally, an implementation is wait-free if any logical operation initiated by a process can complete in a finite number of steps regardless of the actions of the other processes in the system. However, the wait-free property involves fairness considerations because a process cannot complete an operation if it is not allowed to take steps in its protocol. An execution is **fair** to a process if every physical operation initiated by the process eventually completes and if no output action by the process is continuously enabled without occurring. We finally define an implementation to be **wait-free** if for any fixed process, in any execution which is fair to that process, every logical invocation by that process has a matching response. Our algorithms actually provide a bounded number of actions, while our lower bounds hold for algorithms satisfying the weaker definition.

A natural question is why the liveness property is different for physical and logical registers. The wait-free definition for the logical register implies that every logical operation must complete using only physical operations initiated by that logical operation. In the case of the physical register, where we don't model the internal actions, this wait-free property reduces to the physical liveness property given.

To describe a register implementation algorithm, it is sufficient to describe the code for the readers and the writer. An algorithm is a **one-write algorithm** if, in every execution, every logical WRITE uses at most one physical write and no physical reads.

We now define several terms which will be used in the discussion of one-write algorithms.

Let A be a one-write algorithm that uses m binary registers. A **configuration** of A is an element C of $\{0,1\}^m$; let $C[i]$ denote the i^{th} bit of C for $i \in \{1,\ldots,m\}$. The **distance** between two configurations C_1 and C_2, denoted $d(C_1,C_2)$, is the number of bits that differ in C_1 and C_2. Configurations C_1 and C_2 are **neighbors** if $d(C_1,C_2) = 1$. A configuration C is **initial** if $C[i]$ is the value of the i^{th} binary register in the initial state of A for all $i \in \{1,\ldots,m\}$. A configuration C is **reachable** if there exists a state in an execution of A where no physical write is pending such that $C[i]$ is the value of the i^{th} binary register in the state for all $i \in \{1,\ldots,m\}$. (If a physical write is pending, the value of that physical register is ambiguous.)

Chapter 3

Upper Bounds

3.1 The Algorithm

We first present our one-write algorithm.

Let V be the value set of the logical register, where $|V| = k$ and $v_0 \in V$ is the initial value. Let K_V be the complete graph with k nodes and $r = C(k,2)$ edges in which each edge is labeled with a distinct integer from the set $\{1, \ldots, r\}$ and each node is labeled with a distinct element from V. The **special bit set** corresponding to $v \in V$ is defined as $s(v) = \{l : l$ labels an edge incident to the node labeled v in $K_V\}$. Since K_V is a complete graph, $|s(v)| = k - 1$ for all $v \in V$.

Our algorithm uses r binary regular registers (bits). Each bit corresponds to an edge of K_V. A reader reads all r bits and returns the value of a function f applied to the configuration obtained. The function f is defined below. The writer changes a bit only when the value of the logical register changes; when the value is changed from v to w, the bit whose label is contained in $s(v) \cap s(w)$ is changed. There is exactly one such bit because there is exactly one edge connecting v and w in K_V. Figure 3.1 is a formal description of our algorithm.

We now define f. For each $v \in V$ and configuration C, let $count(C, v) = |\{i \in s(v) : C[i] = 1\}|$. Configuration C is **valid** if either (1) $count(C, v)$ is even for all $v \in V$, or (2) $count(C, v_0)$ is odd and $count(C, w)$ is odd for exactly one $w \neq v_0$. Otherwise, C is invalid. First we define f for valid configuration C. If $count(C, v)$ is

even for all $v \in V$, then let $f(C) = v_0$. Otherwise, let $f(C) = v$, where $v \neq v_0$ and $count(C, v)$ is odd. Now we define f for invalid configurations. Let c be the **closest valid configuration function**, where $c(C)$ is defined to be the first configuration in lexicographic order in the set $\{D : D \text{ is valid and } d(C, D) \text{ is a minimum}\}$. Define $f(C)$, for C not valid, to be $f(D)$, where $D = c(C)$.

If a configuration C is valid, then there is a path in K_V, not necessarily edge-disjoint, starting from the node labeled with v_0 and corresponding to initial configuration 0^r such that when the path is traversed and the appropriate bits are changed, then the resulting configuration is C. The resulting node is labeled v, where $v = f(C)$. For each $i \in \{1, \ldots, r\}$, $C[i]$ is the parity of the number of times edge i is traversed in this path. Suppose the path corresponding to valid configuration C does not end at the node labeled with v_0. The two endpoints of the path are adjacent to an odd number of edges in the path, while all internal nodes are adjacent to an even number. The last node in the path is entered one more time than it is left; thus, the count for that node is odd. The first node in the path is left one more time than it is entered; thus, the count for that node is odd. All other nodes are entered and left the same number of times; thus, the counts for those nodes are even. C satisfies condition (2) of the definition of valid. Suppose the path corresponding to valid configuration C ends at the node labeled with v_0, thus forming a cycle. All nodes in the cycle are adjacent to an even number of edges in the cycle. All nodes in the cycle are entered and left the same number of times; thus, the counts for all the nodes are even. C satisfies condition (1) of the definition of valid.

3.2 Proofs of Correctness

We first prove that our algorithm implements a k-ary regular register from binary regular registers in Subsection 3.2.1. We then prove in Subsection 3.2.2 that our algorithm implements a k-ary atomic register from binary atomic registers.

Physical Registers (Bits): $X_1, \ldots, X_r$, initially $X_j = 0$, for all $j \in \{1, \ldots, r\}$

Reader i, $1 \leq i \leq n$: variables $x_1, \ldots, x_r$

READ(i):

for $j := 1$ to r do

read$_j(i)$

return$_j(i, x_j)$

endfor

RETURN($i, f(x_1 \ldots x_r)$)

Writer: variables $x_1, \ldots, x_r$, initially $x_j = 0$, for all $j \in \{1, \ldots, r\}$, and old, initially $old = v_0$

WRITE(v):

if $v \neq old$ then

pick the unique i from $s(v) \cap s(old)$

write$_i(\overline{x_i})$

ack$_i$

$x_i := \overline{x_i}$

$old := v$

endif

ACK

Figure 3.1: One-Write Algorithm

3.2.1 Proof of Regularity

In this subsection, we prove that our algorithm implements a k-ary regular register from binary regular registers. The bulk of this subsection is devoted to showing that the logical register satisfies the regular property.

Lemma 3.1 shows that any reachable configuration is valid and is mapped by f to the value which was written to the register by the last completed WRITE.

Lemma 3.1 *Let C be a reachable configuration resulting from a sequence of m physical writes corresponding to the logical values $v_1, v_2, \ldots, v_m$. Then C is valid,*

and $f(C) = v_m$.

Proof We proceed by induction on m.

Basis: ($m = 0$.) Then C is the initial configuration and is valid, and $f(C) = v_0$.

Inductive step: ($m > 0$.) Suppose the lemma is true for $m-1$. Now we show that it is true for m. Suppose the sequence of logical values is $v_1, v_2, \ldots, v_{m-1}, v_m$ and the sequence of corresponding reachable configurations is $C_1, C_2, \ldots, C_{m-1}, C_m$. By the inductive hypothesis, C_{m-1} is valid, and $f(C_{m-1}) = v_{m-1}$. If $v_{m-1} = v_m$, then C_m trivially is valid, and $f(C_{m-1}) = f(C_m)$ because $C_m = C_{m-1}$. Thus, suppose that $v_{m-1} \neq v_m$. There are two possibilities for v_{m-1}. Either $v_{m-1} = v_0$, or $v_{m-1} \neq v_0$.

Case 1: $v_{m-1} = v_0$. Then $count(C_{m-1}, v)$ is even for all $v \in V$. When the write for v_m is performed, the unique bit $b \in s(v_0) \cap s(v_m)$ is changed. Thus $count(C_m, v_0)$ and $count(C_m, v_m)$ become odd, and $count(C_m, v)$ remains even for all $v \in V - \{v_0, v_m\}$. Therefore C_m is valid, and $f(C_m) = v_m$.

Case 2: $v_{m-1} \neq v_0$. Then $count(C_{m-1}, v_0)$ and $count(C_{m-1}, v_{m-1})$ are odd, and $count(C_{m-1}, v)$ is even for all $v \in V - \{v_0, v_{m-1}\}$. When the write for v_m is performed, the unique bit $b \in s(v_{m-1}) \cap s(v_m)$ is changed. There are two possibilities for v_m. Either $v_m = v_0$, or $v_m \neq v_0$. First suppose that $v_m = v_0$. Thus $count(C_m, v_0)$ and $count(C_m, v_{m-1})$ become even, and $count(C_m, v)$ remains even for all $v \in V - \{v_0, v_{m-1}\}$. Therefore C_m is valid, and $f(C_m) = v_0$. Now suppose that $v_m \neq v_0$. Thus $count(C_m, v_m)$ becomes odd, $count(C_m, v_0)$ remains odd, and $count(C_m, v)$ is even for all $v \in V - \{v_0, v_m\}$. Therefore C_m is valid, and $f(C_m) = v_m$. ■

We need to show that the logical register implemented by our algorithm satisfies the regular property. If a reader RETURNs value v, we must show that v was actually written to the register by some WRITE overlapping the READ or by the last WRITE preceding the READ. This is nontrivial because a slow reader can read either a reachable or a nonreachable configuration by noticing traces from many WRITEs to the logical register by a fast writer. Lemma 3.2 shows that a WRITE(v) operation has occurred during an interval in an execution if a bit in $s(v)$ is changed during that interval. Lemma 3.3 shows that if two valid configurations agree in all bits of $s(v)$ for some v and one is mapped to v by f, our *value extraction function*,

then the other must be mapped to v by f. Lemma 3.4 shows that an invalid configuration C agrees with its closest valid configuration C_N in the special bits corresponding to $f(C_N)$. Lemma 3.5, which shows that the reader will RETURN a correct value of the register no matter what configuration it reads, is the main result of this section. The proof of Lemma 3.5 uses Lemma 3.2 initially to deduce that if a value is not written to the logical register, then all elements of its special bit set remain unchanged. If the reader reads a reachable configuration, then Lemma 3.3 is applied to deduce the correctness of the value RETURNed. Otherwise, Lemmas 3.4 and 3.3 are applied to deduce the correctness of the value RETURNed.

Lemma 3.2 *For any interval in any execution, if no WRITE(v) operation overlaps the interval or occurs as the last preceding WRITE, then the bits in $s(v)$ are not changed during the interval.*

Proof Suppose in contradiction that a bit in $s(v)$ is changed during the interval. Then the value in the register changed from some w to v or the value in the register changed from v to some w. This is impossible because no WRITE(v) operation overlapped the interval or occurred as the last preceding WRITE. Therefore, the lemma is true. ■

Lemma 3.3 *Let C and D be valid configurations. If $f(D) = v$ and $C[i] = D[i]$ for all $i \in s(v)$, then $f(C) = v$.*

Proof There are two cases to consider. Either $v = v_0$, or $v \neq v_0$.

Case 1: $v = v_0$. Thus $count(D, w)$ is even for all $w \in V$. Since $C[i] = D[i]$ for all $i \in s(v_0)$, $count(C, v_0) = count(D, v_0)$. Thus $count(C, w)$ is even for all $w \in V$ because C is valid. This implies that $f(C) = v_0$.

Case 2: $v \neq v_0$. Thus $count(D, v)$ is odd. Since $C[i] = D[i]$ for all $i \in s(v)$, $count(C, v) = count(D, v)$. Thus $count(C, v_0)$ is odd and $count(C, w)$ is even for all $w \in V - \{v_0, v\}$ because C is valid. This implies that $f(C) = v$. ■

Lemma 3.4 *Let C be an invalid configuration, where $D = c(C)$ and $v = f(D)$. Then $C[i] = D[i]$ for all $i \in s(v)$.*

Proof Suppose in contradiction that there exists at least one bit $b \in s(v)$ such that C and D are not equal in that bit. Thus $d(C, D) = l \geq 1$. Change bit b in D to yield C_D. C_D is valid and $C_D[b] = C[b]$. So $d(C, C_D) = l - 1$. This is a contradiction, because D was supposed to be the closest valid configuration to C. Therefore, the lemma is true. ■

Lemma 3.5 *Let C be the configuration obtained by a reader during some execution of the READ protocol. Suppose $f(C) = v$. Then the value v was written by a WRITE which overlapped the READ or the value v was the result of the last WRITE preceding the READ.*

Proof Assume for contradiction that the value v was not written by a WRITE which overlapped the READ and the value v was not the result of the last WRITE preceding the READ. Thus no state of the algorithm during the READ has the physical registers in a configuration with value v. By Lemma 3.2, the bits in $s(v)$ are never changed during the READ. Let D be any reachable configuration resulting from either the last preceding WRITE or any overlapping WRITE. D is valid by Lemma 3.1, and $D[i] = C[i]$ for all $i \in s(v)$. There are two cases to consider. Either C is a valid configuration, or C is an invalid configuration.

Case 1: Suppose C is valid. Since D has the same values as C for the bits in $s(v)$ and $f(C) = v$, $f(D) = v$ by Lemma 3.3, which is a contradiction.

Case 2: Suppose C is not valid. Let $C_N = c(C)$. Then $f(C_N) = v$. By Lemma 3.4, $C[i] = C_N[i]$ for all $i \in s(v)$. By the transitive property of equality, $C_N[i] = D[i]$ for all $i \in s(v)$. By Lemma 3.3, $f(D) = v$, which is a contradiction. ■

The result of Lemma 3.5 proves the following theorem. The logical register is seen to be wait-free by inspecting the code of the read and write processes.

Theorem 3.1 *A one-write algorithm for implementing a k-ary regular register from binary regular registers exists.*

3.2.2 Proof of Atomicity

The previous subsection showed that our k-ary register is regular, but we need to go one step further. We now assume that the constituent binary registers are atomic. Since atomicity implies regularity, our new register still satisfies the regular property. We must now show that our new register satisfies the atomic property by constructing a linearization for an arbitrary execution.

We define several terms which will be used in proving that our algorithm satisfies the atomic property. If W is a logical WRITE, then $value(W) = v$ if the call for W is a WRITE(v). If R is a logical READ, then $value(R) = v$ if the response for R is a RETURN(i, v) for some i. The **possible writes-to-read set** for READ R, denoted by PRS(R), is $\{W | W$ is a WRITE and W either immediately precedes R or overlaps $R\}$. Without loss of generality, we assume that each execution contains a special initializing WRITE which precedes all other operations in the execution. Thus, PRS(R) is always nonempty. The **same value set** for READ R, denoted by SVS(R), is $\{W | W \in$ PRS(R) and $value(W) = value(R)\}$.

Given execution e of the algorithm, we must give a linearization T of the operations in e. Since the logical register only has one writer, the WRITEs are already linearized. We just need to determine where to place each READ. We consider the READs in the order in which they end. This yields a total ordering of the READs, denoted $R_1, R_2, \ldots$.

Given WRITE W in e, define **last**(W) to be the last physical write in e that finishes before W ends, and define **next**(W) to be the first physical write in e that starts after W ends.

Since the physical registers are atomic, there exists a linearization L of the physical operations of e. Fix such an L. Relative to L, each physical read has a unique physical write that it **reads from**, namely the most recent preceding write to that register in the linearization.

We say that a READ R **observes** a write w to physical register b, relative to L, if R's read of b reads from w relative to L.

Lemma 3.6 shows that there is a correspondence between the READs and WRITEs in an execution which can be turned into a linearization.

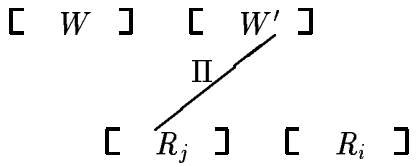

Figure 3.2: Relationships between Two Possibly Conflicting READs R_i and R_j

Lemma 3.6 *There exists a function* Π *from the READs in* e *to the WRITEs in* e *such that for all* $i \geq 1$*, the following are true.*

1. $\Pi(R_i)$ *is in* $SVS(R_i)$.
2. *For all* $j < i$*, if* R_j *strictly precedes* R_i *in* e*, then* $\Pi(R_j)$ *does not follow* $\Pi(R_i)$ *in* e.
3. R_i *observes* $last(\Pi(R_i))$ *or* $next(\Pi(R_i))$.

Proof We inductively (and non-constructively) define Π.

Basis: Let $\Pi(R_1)$ be any W in SVS(R_1) such that R_1 observes either last(W) or next(W). We now prove that such a W exists. By Theorem 3.1, SVS(R_1) is nonempty. Suppose in contradiction that such a W does not exist. Then all physical reads of bits in $s(value(R_1))$ occur before the first WRITE in SVS(R_1). If the first WRITE in PRS(R_1) is the first WRITE in SVS(R_1), then we have a contradiction because the first WRITE in PRS(R_1) precedes R_1 in e. Otherwise, let C be the configuration of the physical registers after the first WRITE in PRS(R_1). C is valid, and $f(C) \neq value(R_1)$ by Lemma 3.1. Let C_1 be the configuration read by R_1. $f(C_1) = value(R_1)$. Let $v = f(C_1)$. $C_1[b] = C[b]$ for all b in $s(v)$. Let $D = c(C_1)$ (if C_1 is valid, then $D = C_1$). $D[b] = C_1[b]$ for all b in $s(v)$ by Lemma 3.4. Thus, $D[b] = C[b]$ for all b in $s(v)$. By Lemma 3.3, $f(C) = f(D) = v$, which is a contradiction. Thus, we can find a W in SVS(R_1) such that R_1 observes either last(W) or next(W). Conditions 1 and 3 are true by construction and condition 2 is vacuously true.

Inductive step: Assume Π has been defined for $R_1, \ldots, R_{i-1}$. We show that there exists some W in SVS(R_i) satisfying conditions 2 and 3. Define $\Pi(R_i)$ to be W.

Claim 3.1 *If, for any W in PRS(R_i), there exists $j < i$ such that R_j strictly precedes R_i and $\Pi(R_j) = W'$ follows W in e, then*

(i) W is the first WRITE in PRS(R_i),

(ii) W' is the last WRITE in PRS(R_j), and

(iii) PRS(R_j)$\cap$PRS(R_i) $= \{W, W'\}$.

Proof By the inductive hypothesis, W' is in SVS(R_j). Figure 3.2 gives an illustration of the relationships among R_i, R_j, W, and W'.

End of Claim

Suppose in contradiction that there is no WRITE in SVS(R_i) satisfying conditions 2 and 3. Then for every W in SVS(R_i), either

(a) there exists $j < i$ such that R_j strictly precedes R_i and $\Pi(R_j)$ follows W in e, or

(b) R_i does not observe last(W) or next(W) relative to L.

Let W_1 be the first WRITE in PRS(R_i), and let $v = value(R_i)$.

Case 1: W_1 is not in SVS(R_i). By Claim 3.1, all WRITEs in SVS(R_i) satisfy (b). Since R_i does not observe last(W) or next(W) for any W in SVS(R_i), R_i's read of physical register b must occur in L before the earliest write of b within a WRITE in PRS(R_i) $- \{W_1\}$, for all b in $s(v)$. Clearly all reads of R_i occur after W_1. Thus $C[b] = C_i[b]$ for all b in $s(v)$, where C is the configuration after W_1 (C is valid by Lemma 3.1) and C_i is the configuration that R_i reads. Let $D = c(C_i)$ (if C_i is valid, then $D = C_i$). By Lemma 3.4, $D[b] = C_i[b]$ for all b in $s(v)$. Thus, $D[b] = C[b]$ for all b in $s(v)$. By Lemma 3.3, $f(C) = f(D) = v$, a contradiction.

Case 2: W_1 is in SVS(R_i) and satisfies (b). By Claim 3.1, all WRITEs in SVS(R_i) satisfy (b). Thus, R_i does not observe last(W) or next(W) for any W in SVS(R_i). last(W_1) occurs before R_i begins, which implies that R_i must observe last(W_1) or some later write of the same physical register written by last(W_1), due to a WRITE in SVS(R_i) or a WRITE following a WRITE in SVS(R_i). This is a contradiction.

Case 3: W_1 is in SVS(R_i) and does not satisfy (b). By our supposition, W_1 satisfies (a). Thus, there exists $j < i$ such that R_j precedes R_i in e, but $\Pi(R_j)$ follows W. Let $W_2 = \Pi(R_j)$.

Since by Claim 3.1 W_2 is not the first WRITE in PRS(R_j), R_j observes last(W_2) or next(W_2) by the inductive hypothesis. But R_j cannot observe next(W_2) because next(W_2) does not overlap R_j (since W_2 is the last WRITE in PRS(R_j)). Thus, R_j observes last(W_2).

Since all of R_i's reads follow all of R_j's reads in L, R_i must observe last(W_2) or some later write of the same physical register, due to a WRITE in SVS(R_i) or a WRITE following a WRITE in SVS(R_i). But note that last(W_2) is equal to either last(W_1) or next(W_1). Thus, some WRITE in SVS(R_i) satisfies (b), a contradiction.

■

We can use Lemma 3.6 to determine where to place each READ in our proposed linearization. As usual, we consider the READs in the order in which they end. Define Σ_i inductively as follows. Let Σ_0 be the sequence of WRITEs in e, in order. For $i > 0$, let Σ_i be obtained from Σ_{i-1} by placing R_i immediately before the first WRITE following $\Pi(R_i)$. (If no WRITE follows $\Pi(R_i)$, then place R_i at the end of Σ_{i-1}.)

If e contains only a finite number of READs, then let $T = \Sigma_k$, where k is the number of READs in e. We need to show that if e contains an infinite number of READs, then the sequence of sequences produced by our method has a limit. We take this limit to be the linearization. If e contains an infinite number of READs, then we define T as follows. For $j \leq i$, R_j is **stable** in Σ_i if the prefix of Σ_i through R_j is also a prefix of Σ_k, for all $k > j$. If R_m precedes R_j in Σ_i and R_j is stable

in Σ_i, then clearly R_m is also stable in Σ_i. Let T_i be the smallest prefix of Σ_i containing all the READs that are stable in Σ_i. Let $T = \lim_{i \to \infty} T_i$. T is clearly well-defined. We must show that it contains all the operations of e. It suffices to show the following lemma.

Lemma 3.7 *For all i, there exists $j \geq i$ such that R_i is stable in Σ_j.*

Proof: Suppose in contradiction that there exists an i such that R_i is not stable in Σ_j for all $j \geq i$. This means that each R_j, where $j > i$, is placed before R_i in the proposed linearization. We now show that each R_j overlaps R_i. Since $j > i$, R_j ends after R_i ends in e. Thus, R_j does not precede R_i in e. If R_i precedes R_j in e, then our method for placing READs ensures that R_j would be placed after R_i in the proposed linearization. We can deduce that R_j overlaps R_i. Thus, R_i has infinitely many READs overlapping it, which is impossible.

□

Now we can prove that our algorithm satisfies the atomic property.

Theorem 3.2 *T is a linearization of e.*

Proof: Since each READ R is placed after a WRITE W in SVS(R) and before any WRITE which W precedes, the value RETURNed by R is correct. Thus, all READs RETURN correct values. We need to show that the order of non-overlapping operations is preserved. We have four cases to consider:

- Suppose W_1 precedes W_2 in e, where W_1 and W_2 are WRITEs. W_1 precedes W_2 in T by construction.
- Suppose W precedes R in e, where W is a WRITE and R is a READ. W precedes R in T because R is always placed after a WRITE in SVS(R) which wrote the value it RETURNs by construction and Lemma 3.6, and that WRITE is either W or a later WRITE.

- Suppose R precedes W in e, where R is a READ and W is a WRITE. By construction and Lemma 3.6, R is placed after some WRITE W' which wrote the value it RETURNs, where W' either immediately precedes or overlaps R, and before any WRITE which W' precedes. W' precedes W because the logical register only has one writer. Thus, the following ordering holds in T: W', R, W.

- Suppose R_1 precedes R_2 in e, where R_1 and R_2 are READs. R_1 precedes R_2 in T by construction and condition 2 of Lemma 3.6.

□

Corollary 3.1 *A one-write algorithm for implementing a k-ary atomic register from binary atomic registers exists.*

Chapter 4

Lower Bounds on Number of Registers

We have proven the existence of a one-write algorithm for implementing a k-ary regular (respectively, atomic) register from binary regular (respectively, atomic) registers. The number of registers used by our algorithm is very large, $C(k,2) = O(k^2)$. The best previously known lower bound on the number of registers for this problem is k, shown by Chaudhuri and Welch [CW90].

Section 4.1 gives lower bounds on the number of registers required by two classes of one-write algorithms for implementing a k-ary regular register from binary regular registers. We call these algorithms **regular one-write algorithms**. Section 4.2 gives lower bounds on the number of registers required by two classes of one-write algorithms for implementing k-ary atomic registers from binary atomic registers. We call these algorithms **atomic one-write algorithms**.

4.1 Lower Bounds for Regularity

In this section we establish lower bounds on the number of registers required by two classes of regular one-write algorithms. Subsection 4.1.1 gives a lower bound of $C(k,2)$ for the class of regular one-write algorithms satisfying the toggle property.

Subsection 4.1.2 gives a lower bound of $2k - 1 - \lfloor \log k \rfloor$ for the class of regular one-write algorithms satisfying the symmetric property. These properties are defined below.

A one-write algorithm with the following properties is a **normal form** algorithm.

1. no reader performs a physical write
2. every reader has the same program
3. every reader starts in the same state at the beginning of every READ

In Subsection 4.1.3, we show how any one-write algorithm can be converted to a normal form algorithm without increasing the number of physical registers. Thus we can, without loss of generality, restrict our attention to normal form algorithms.

If one-write algorithm A uses m binary registers, A has 2^m configurations. These configurations are nodes in a directed m-dimensional hypercube H_A. If configurations C_1 and C_2 are neighbors, then both (C_1, C_2) and (C_2, C_1) are edges of H_A. An edge (C_1, C_2) of H_A is an **algorithm edge** if C_1 and C_2 are reachable configurations and C_2 can be derived from C_1 after one WRITE operation. An edge (C_1, C_2) of H_A is labeled with i, where i is the bit in which C_1 and C_2 differ.

A one-write algorithm A has the **symmetric property** if for all configurations C_1, C_2, (C_1, C_2) is an algorithm edge of H_A if and only if (C_2, C_1) is an algorithm edge of H_A. If A satisfies the symmetric property, the two directed edges connecting any pair of neighboring configurations are either both algorithm edges or both non-algorithm edges. Thus the two directed edges can be replaced by one edge which is either an algorithm edge or a non-algorithm edge. Therefore, H_A can be considered an undirected graph. In Subsection 4.1.3, we show how an arbitrary symmetric algorithm can be transformed into a symmetric algorithm using no more registers in which every reader reads each physical register at most once during a READ. Thus we can assume without loss of generality that in a symmetric algorithm every reader reads each physical register at most once during a READ. The symmetric property seems reasonably unrestrictive and it may allow for implementations requiring fewer physical registers.

A one-write algorithm has the **toggle property** if for each pair of distinct $v, w \in V$, there exists a unique bit l such that whenever the value of the logical register is changed from v to w or from w to v, bit l is written. A one-write algorithm satisfying the toggle property trivially satisfies the symmetric property. Our algorithm satisfies this property. We will show that our algorithm is optimal in the class of algorithms satisfying this property with respect to the number of physical registers. However, when proving lower bounds on the number of physical registers required by general one-write algorithms, the toggle property is an overly restrictive property for a one-write algorithm.

In our lower bound proofs, we want to deduce the value which must be RETURNed by a reader given a particular configuration of the physical registers. This mapping from configurations to values is given by a "value extraction function", such as the function f from our algorithm in Section 3.1. We now define a value extraction function for the more general class of symmetric algorithms in which a reader does not have to read every physical register. We first define the term "consistent". Bit i is **consistent** with configuration C if the value of bit i is $C[i]$. Let A be a symmetric algorithm for implementing a k-ary regular register from m binary regular registers. For algorithm A we define a **value extraction function** $f_A : \{0,1\}^m \rightarrow V$. If no reader ever reads bits consistent with configuration C, then $f_A(C)$ is undefined. If all the bits that a reader reads are consistent with configuration C and the reader RETURNs v, then $f_A(C) = v$. Thus f_A is a partial function. We now discuss why f_A is well-defined. Consider two logical READs. Suppose the reader performing the first logical READ reads a subset S_1 of the physical registers, RETURNing v_1, and the reader performing the second logical READ reads a different subset S_2 of the physical registers, RETURNing v_2, where $v_1 \neq v_2$. Suppose all bits in $S_1 \cup S_2$ are consistent with C. This is impossible because the readers have the same program and start their READs in the same initial state. For the readers to read two different sets of physical registers, there must be some physical register for which the first reader obtained 1 and the second reader obtained 0 (or vice versa). Thus one of the readers did not read bits consistent with configuration C. Therefore, f_A is well-defined.

We now define terms which will be used in the formalization of our general

technique for "fooling the reader", which is Lemma 4.1. Let A be a one-write algorithm for implementing a k-ary regular register from m binary regular registers that satisfies the symmetric property. Let S be a set of reachable configurations and C be a configuration. C is **constructible** from S if for each $i \in \{1, \ldots, m\}$, there exists a $C' \in S$ such that $C'[i] = C[i]$. (A similar definition was given in [JSL91].) Let $f_A(S) = \{f_A(C) : C \in S\}$. S is **connected** if for all distinct $D, E \in S$, there exists a path from D to E in H_A consisting only of algorithm edges in which every configuration on the path is an element of S.

Given a configuration C which is constructible from a connected set of configurations S, Lemma 4.1 states that $f_A(C)$ must be in $f_A(S)$; otherwise, the reader could be fooled into returning a wrong value. In our lower bound proofs, we obtain a contradiction to Lemma 4.1 by identifying a connected set S of configurations and showing how there is a constructible C with a wrong value.

Lemma 4.1 *Let A be a one-write algorithm that satisfies the symmetric property. For all configurations C and connected sets of reachable configurations S, if C is constructible from S, then $f_A(C) \in f_A(S)$.*

Proof Suppose in contradiction that $f_A(C) \notin f_A(S)$. Consider the following execution of A. First the writer executes a sequence of WRITEs so that the resulting configuration of the physical registers is in S. This sequence exists because S is reachable. Then a logical READ starts. For all i, whenever the reader is about to read bit i, the writer executes a sequence of WRITEs with the following properties: (1) the configuration of the physical registers after each WRITE is in S, and (2) the final configuration D is such that $C[i] = D[i]$. Since S is connected, this sequence exists. Thus the reader returns $f_A(C)$, which violates the regular property because $f_A(C)$ was not the value of any overlapping WRITE or of the preceding WRITE. ■

This lemma is true for nonsymmetric algorithms if S is a strongly connected set and the definition of f_A is appropriately modified. In the general case, we can define $f_A(C)$ to be the value RETURNed by a reader if all the bits that a reader reads are consistent with configuration C and if the reader never sees two different

values for the same bit during the READ. The general lemma might be useful in proving lower bounds for nonsymmetric algorithms.

4.1.1 Toggle Property

We can show that the upper bound of $C(k,2)$ is tight for the class of algorithms satisfying the toggle property (which includes our algorithm). Every algorithm A with the toggle property can be represented by the complete graph on k nodes, in which each node is labeled with a distinct element from V and the edge between v and w is labeled with some $l \in \{1,\ldots,m\}$ (when the value of the logical register is changed from v to w or vice versa, bit l is changed), where m is the number of binary registers used by A. Call this graph G_A.

When $k = 3$, $k = C(k,2)$; thus our algorithm is trivially optimal in the number of binary regular registers used. Theorem 4.1 shows that $C(k,2)$ binary regular registers are necessary for any $k \geq 4$.

Theorem 4.1 *For all one-write algorithms A for implementing a k-ary ($k \geq 4$) regular register from binary regular registers, if A has the toggle property, then the number of binary regular registers used by A is at least $C(k,2)$.*

Proof Suppose that A is a one-write algorithm for implementing a k-ary regular register from binary regular registers, where A has the toggle property and the number of registers used by A is less than $C(k,2)$. Then there is some register i such that i is the label of at least two edges in G_A, say (v_1,v_2) and (v_3,v_4). Suppose the edges have a common endpoint. Without loss of generality, assume $v_1 = v_3$. Then $v_2 \neq v_4$ because otherwise the edges would be the same. If the current value of the logical register is v_1 and bit i is changed, the new value of the logical register is both v_2 and v_4, which is ambiguous. Thus the edges are disjoint; v_1, v_2, v_3, and v_4 are distinct.

Let j, where $j \neq i$, label the edge (v_1,v_3) of G_A. Let C_1 be any configuration such that $f_A(C_1) = v_1$. Let C_2 be the configuration that differs from C_1 only in bit i. Let C_3 be the configuration that differs from C_1 only in bit j. Let C_4 be

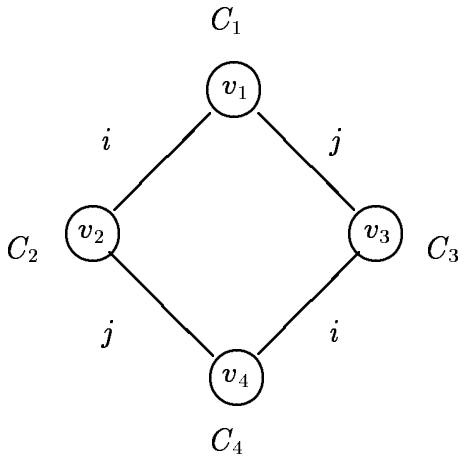

Figure 4.1: Relationships Among the Four Configurations in the Proof of Theorem 4.1

the configuration that differs from C_1 only in bits i and j. By the definition of G_A, C_2, C_3, and C_4 are reachable configurations, and $f_A(C_2) = v_2$, $f_A(C_3) = v_3$, and $f_A(C_4) = v_4$. Figure 4.1 shows the relationships among C_1,C_2,C_3, and C_4. C_2 is constructible from the connected set $\{C_1, C_3, C_4\}$. But $f_A(C_2) = v_2$ is not in $f_A(\{C_1, C_3, C_4\}) = \{v_1, v_3, v_4\}$, contradicting Lemma 4.1. ■

4.1.2 Symmetric Property

The symmetric property seems to be desirable since it is plausible that an algorithm with this property would use fewer registers. Also, it may simplify a lower bound proof since we can use the "fooling the reader" technique. Let $SYM(k)$ be the set of all one-write algorithms which implement a k-ary regular register from binary regular registers and satisfy the symmetric property. For an algorithm $A \in SYM(k)$, let $R_A(k)$ be the number of binary registers used by A. Let $R(k)$ be the minimum number of binary registers required by any one-write algorithm in $SYM(k)$. The main result of this section is Theorem 4.2, which states that $R(k) > 2k-2-\lfloor \log k \rfloor$.

The proof of Theorem 4.2 is inductive. Lemma 4.2, which shows that 4 binary regular registers cannot implement a 4-ary regular register, forms the base case for the proof. In the inductive step, either k is a power of 2, or k is not a power of 2. If k is a power of 2, then Lemma 4.4, which proves that $R(k) \geq R(k-1)+1$, is used. If k is not a power of 2, then Lemma 4.5, which proves that $R(k) \geq R(k-1)+2$, is used. Then some algebraic manipulations enable us to derive the desired lower bound. The proofs of Lemmas 4.4 and 4.5 use Lemma 4.3, which gives conditions under which a one-write algorithm can be converted into a one-write algorithm for fewer logical values using fewer physical registers. The proof of Lemma 4.3 consists of a general algorithm transformation.

Lemma 4.2 $R(4) > 4$.

Proof Suppose in contradiction that there exists an algorithm A such that $R_A(4) = 4$. Suppose without loss of generality that $V = \{R, G, B, Y\}$, the initial configuration is 0000, $f_A(0000) = R$, $f_A(1000) = G$, $f_A(0100) = B$, and $f_A(0010) = Y$. We now attempt to assign values to the remaining 12 configurations.

Figure 4.2 shows the current assignment of values to configurations and the possibilities for some currently unassigned configurations. Because A is a one-write algorithm, we only need to consider configurations which differ in one bit from the last assigned configuration 1000. We cannot assign two different values to the same configuration. Thus, we have six choices to consider:

1. $f_A(1010) = B$ and $f_A(1100) = Y$.
2. $f_A(1010) = B$ and $f_A(1001) = Y$.
3. $f_A(1001) = B$ and $f_A(1100) = Y$.
4. $f_A(1100) = B$ and $f_A(1010) = Y$.
5. $f_A(1100) = B$ and $f_A(1001) = Y$.
6. $f_A(1001) = B$ and $f_A(1010) = Y$.

We can eliminate choices 1 and 2 by showing that $f_A(1010) \neq B$. $f_A(1010) \neq B$ because otherwise 0010 is constructible from the connected set $\{0000, 1000, 1010\}$ and $f_A(0010) = Y$ is not in $f_A(\{0000, 1000, 1010\}) = \{R, G, B\}$, contradicting Lemma 4.1. We can eliminate choice 3 by showing that $f_A(1100) \neq Y$. $f_A(1100) \neq Y$ because otherwise 0100 is constructible from the connected set $\{0000, 1000, 1100\}$ and $f_A(0100) = B$ is not in $f_A(\{0000, 1000, 1100\}) = \{R, G, Y\}$, contradicting Lemma 4.1.

We now show how to eliminate choices 4, 5, and 6. We consider each of the three choices in turn.

Case 4. Figure 4.3 shows the current assignment of values to configurations and the possibilities for some currently unassigned configurations. $f_A(0110) \neq G$ because otherwise 0110 is constructible from the connected set $\{0000, 0010, 0100\}$ and $f_A(0110)$ is not in $f_A(\{0000, 0010, 0100\}) = \{R, B, Y\}$, contradicting Lemma 4.1. Thus, we only have one choice to consider: $f_A(0101) = G$ and $f_A(0110) = Y$. Figure 4.4 shows the current assignment of values to configurations and the possibilities for some currently unassigned configurations. f_A cannot map 0011 to both G and B. This case leads to a dead end.

Case 5. Figure 4.5 shows the current assignment of values to configurations and the possibilities for some currently unassigned configurations. As in Choice 4, $f_A(0110) \neq G$ and thus $f_A(0101) = G$ and $f_A(0110) = Y$. Figure 4.6 shows the current assignment of values to configurations and the possibilities for some currently unassigned configurations. $f_A(1010) \neq B$ because otherwise 1000 is constructible from the connected set $\{0000, 0010, 1010\}$ and $f_A(1000) = G$ is not in $f_A(\{0000, 0010, 1010\}) = \{R, B, Y\}$, contradicting Lemma 4.1. Thus, we only have one choice to consider: $f_A(1010) = G$ and $f_A(0011) = B$. Figure 4.7 shows the current assignment of values to configurations and the possibilities for some currently unassigned configurations. $f_A(1101) \neq R$ because otherwise 1001 is constructible from the connected set $\{0000, 1000, 1100, 1101\}$ and $f_A(1001) = Y$ is not in $f_A(\{0000, 1000, 1100, 1101\}) = \{R, G, B\}$, contradicting Lemma 4.1. $f_A(1110) \neq R$ because otherwise 0110 is constructible from the connected set $\{0000, 1000, 1100, 1110\}$ and $f(0110) = Y$ is not in $f_A(\{0000, 1000, 1100, 1110\}) = \{R, G, B\}$, contradicting Lemma 4.1. This case leads to a dead end.

Case 6. Figure 4.8 shows the current assignment of values to configurations and the possibilities for some currently unassigned configurations. $f_A(0110) \neq G$ because otherwise 1010 is constructible from the connected set $\{0000, 1000, 0100, 0110\}$ and $f_A(1010) = Y$ is not in $f_A(\{0000, 1000, 0100, 0110\}) = \{R, G, B\}$, contradicting Lemma 4.1. $f_A(1100) \neq Y$ because otherwise 1000 is constructible from the connected set $\{0000, 0100, 1100\}$ and $f_A(1000) = G$ is not in $f_A(\{0000, 0100, 1100\}) = \{R, B, Y\}$, contradicting Lemma 4.1. Thus, we have three choices to consider:

6.1. $f_A(1100) = G$ and $f_A(0110) = Y$.

6.2. $f_A(1100) = G$ and $f_A(0101) = Y$.

6.3. $f_A(0101) = G$ and $f_A(0110) = Y$.

We consider each of the three choices in turn.

Case 6.1. Figure 4.9 shows the current assignment of values to configurations and the possibilities for some currently unassigned configurations. f_A cannot map 0011 to both G and B. This choice leads to a dead end.

Case 6.2. Figure 4.10 shows the current assignment of values to configurations and the possibilities for some currently unassigned configurations. $f_A(0110) \neq G$ because otherwise 0100 is constructible from the connected set $\{0000, 0010, 0110\}$ and $f_A(0100) = B$ is not in $f_A(\{0000, 0010, 0110\}) = \{R, G, Y\}$, contradicting Lemma 4.1. $f_A(0011) \neq G$ because otherwise 1001 is constructible from the connected set $\{0000, 1000, 0010, 0011\}$ and $f_A(1001) = B$ is not in $f_A(\{0000, 1000, 0010, 0011\}) = \{R, G, Y\}$, contradicting Lemma 4.1. This case leads to a dead end.

Case 6.3. Figure 4.11 shows the current assignment of values to configurations and the possibilities for some currently unassigned configurations. f_A cannot map 0011 to both G and B. We have nowhere else to backtrack.

Thus, $R(4) > 4$.

■

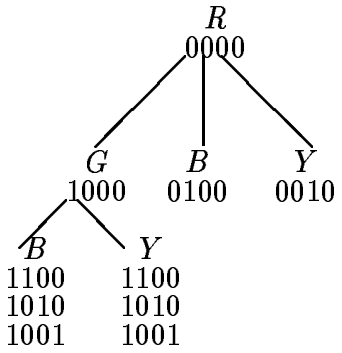

Figure 4.2: First Set of Choices

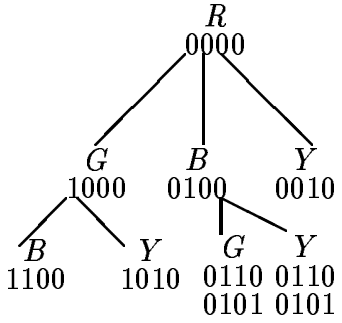

Figure 4.3: Case 4 and Second Set of Choices

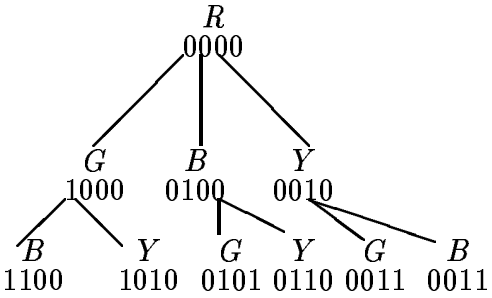

Figure 4.4: Case 4 - Remaining Choice

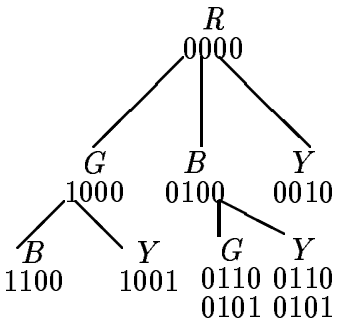

Figure 4.5: Case 5 and Second Set of Choices

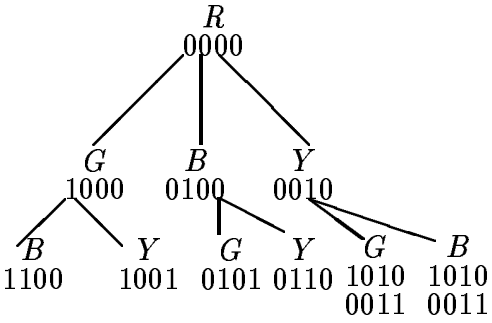

Figure 4.6: Case 5 and Third Set of Choices

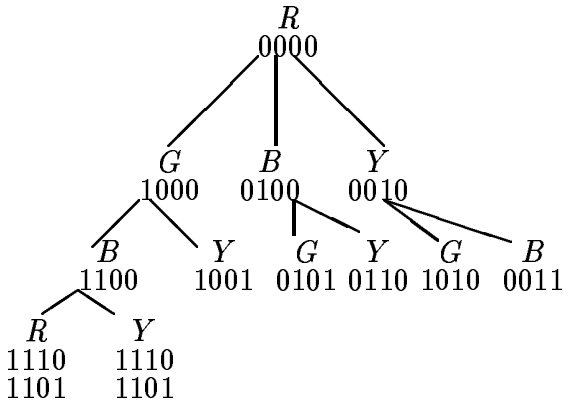

Figure 4.7: Case 5 and Fourth Set of Choices

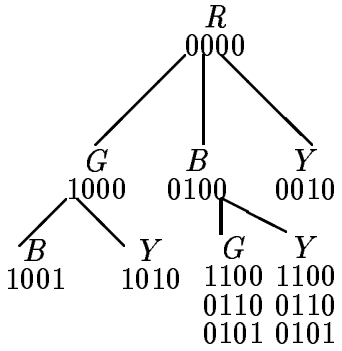

Figure 4.8: Case 6 and Second Set of Choices