Chapter 3.

Circuit Synthesis

The digital abstraction used to describe a CMOS circuit treats the transistor as a switch. The low voltage level (ground, often abbreviated *GND*) is treated as Boolean **false** (or logic 0), and the high voltage level (supply voltage, often abbreviated *Vdd*) is treated as Boolean **true** (or logic 1). However, voltages are continuous quantities and they can take on intermediate values.

Figure 3.1 shows how simple clocked circuits enforce the digital abstraction by using the clock signal to discretize the time domain. When the circuit is exhibiting non-ideal behavior, i.e., when signals are changing, the clock is used to ignore the transient state of the system. The light grey regions in Figure 3.1 show the times at which the circuit can exhibit transient behavior. Once that period is over, the voltages of all signals must be in the dark grey bands representing logic levels 0 and 1. The width of each band corresponds to the *noise margin* of the circuit.

An asynchronous circuit cannot ignore the intermediate states that occur when signals are changing. An important observation is that we do not need transistors to behave as discrete devices; voltages can be viewed as continuous quantities, as long as changes from a stable value in

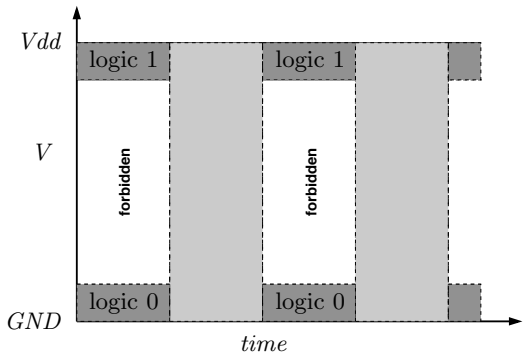

Figure 3.1. Clocked system showing discretization in time.

one logic region to another are *monotonic*. This monotonicity assumption is implemented during the circuit design process.


A change in voltage of the input to a gate can be detected when the voltage passes the switching threshold of the gate. Therefore, switching thresholds of operators determine the noise margin of their inputs.


Strict monotonicity is not necessary—which is fortunate because it is impossible to implement in the presence of noise. All that is required is that fluctuations in the input near the switching threshold of an operator are transient. The property of *stability* guarantees that the network implementing the input will eventually change the value of the input node to *Vdd* or *GND*.

3.1. Production Rules

Our digital abstraction for CMOS circuits is the *production rule set*, which consists of a set of *production rules*. A production rule is a construct of the form $G \mapsto S$, where S is either a simple assignment statement or a list of simple assignment statements, and G is a Boolean expression called the *guard* of the production rule. A simple assignment is one that assigns a constant value (**true** or **false**) to a Boolean-valued variable.

Example. The following are examples of production rules:

$$x \wedge y \;\mapsto\; z\uparrow$$
$$\neg x \;\mapsto\; u\downarrow, v\uparrow$$

Note that the only assignment statements on the right hand side of a production rule arrow are of the form $v := \textbf{true}$ or $v := \textbf{false}$ (where v is a variable), and are abbreviated $v\uparrow$ and $v\downarrow$ respectively. ∎

A production rule $G \mapsto s_1, s_2, \ldots, s_n$ where all the s_i's are simple assignment statements is an abbreviation for the production rule set $\{G \mapsto s_1, G \mapsto s_2, \ldots, G \mapsto s_n\}$. Therefore, we only consider production rules that have a simple assignment statement on the right hand side of the arrow.

Example. The production rule $\neg x \mapsto u\downarrow, v\uparrow$ shown above is equivalent to the production rule set

$$\neg x \;\mapsto\; u\downarrow \qquad\qquad \neg x \;\mapsto\; v\uparrow$$

∎

The *execution* of a production rule $G \mapsto t$ where t is a simple assignment statement (sometimes called a *transition*) is defined as an unbounded sequence of *firings*. A firing of $G \mapsto t$ with $G \equiv \textbf{true}$ amounts to the execution of t; A firing of $G \mapsto t$ with $G \equiv \textbf{false}$ amounts to **skip**. The execution of a production rule set is the weakly fair concurrent composition of the execution of the individual production rules in the set.

We use the predicate R on transitions to denote the result of a transition: $R(v\uparrow) \equiv v$ and $R(v\downarrow) \equiv \neg v$. A firing of a production rule $G \mapsto t$ in a state where $G \wedge \neg R(t)$ holds is called an *effective firing*. An effective firing changes the state of the computation. A firing of $G \mapsto t$ in a state where $G \wedge R(t)$ holds is called a *vacuous firing*.

This definition of production rules and their execution was chosen so as to make them easily implementable in CMOS. Consider a production rule of the form $G \mapsto v\uparrow$. If we implement a switching network that is conducting just when G is **true**, and connect v to one end and *Vdd* (which represents **true**) to the other end, we will have implemented the production rule $G \mapsto v\uparrow$ correctly. Whenever the switching network is not conducting, i.e., $G \equiv$ **false**, v will not be changed by the action of $G \mapsto v\uparrow$. If the switching network is conducting, i.e., $G \equiv$ **true**, then v will be connected to *Vdd* (logic value **true**), and will therefore eventually be set to **true**.

3.1.1. Stability and Non-interference

Two production rules are said to be *complementary* when they set the same variable to two different values. Complementary production rules are of the form $G^+ \mapsto v\uparrow$ and $G^- \mapsto v\downarrow$. Two complementary production rules $G^+ \mapsto v\uparrow$ and $G^- \mapsto v\downarrow$ are said to be *non-interfering* when $\neg G^+ \vee \neg G^-$ is invariant. A production rule set is said to be non-interfering when all pairs of complementary production rules in the set are non-interfering.

Example. The production rules

$$x \mapsto u\uparrow$$
$$y \mapsto u\downarrow$$

are interfering if $x \wedge y$ holds at any point in the computation. ∎

A production rule $G \mapsto t$ is *stable* in a computation just when G can change from **true** to **false** only in states where $R(t)$ holds. If the computation can change state from $G \wedge \neg R(t)$ to $\neg G \wedge \neg R(t)$, the production rule $G \mapsto t$ is *unstable*. A production rule set is said to be stable when all production rules in it are stable.

Example. Consider the production rule set

$$a \mapsto b\downarrow \qquad c \mapsto a\downarrow$$
$$\neg a \mapsto b\uparrow \qquad \neg c \mapsto a\uparrow$$

$$b \mapsto c\downarrow \qquad a \mapsto x\uparrow$$
$$\neg b \mapsto c\uparrow \qquad \neg a \mapsto x\downarrow$$

Assume the initial state is $\neg a \wedge \neg x \wedge b \wedge \neg c$. The production rule $a \mapsto x\uparrow$ is unstable, because the system can change from state $a \wedge \neg b \wedge c \wedge \neg x$ to $\neg a \wedge \neg b \wedge c \wedge \neg x$, making the guard of $a \mapsto x\uparrow$

change from **true** to **false** in a state where x is **false**. ∎

The only valid production rule sets are those that are stable and non-interfering. The examples demonstrate that stability and non-interference do not hold for arbitrary production rule sets. The synthesis method described in this chapter will guarantee these two properties.

It can be shown that, under the stability and non-interference of a production rule set, the execution of the production rules of a set is equivalent to the following sequential execution:

```
*[ select a production rule with a true guard;
   fire the production rule
 ]
```

where the selection operation is weakly fair. This equivalence simplifies the analysis of production rule sets. Circuits generated from stable, non-interfering production rule sets are known as *quasi delay-insensitive* (QDI) circuits. Chapter 9 describes the difference between these circuits are purely delay-insensitive circuits in detail.

3.2. CMOS Implementation of Production Rules

A CMOS circuit is a network of electrical *nodes*, interconnected by transistors and wires. The circuit contains the special node *Vdd* that provides the constant high-voltage value used to represent **true**/logic 1, and the special node *GND* that provides the constant low-voltage value used to represent **false**/logic 0. A node in the circuit can take on voltage values between the high voltage and low voltage.

Our design methodology will guarantee that the production rules generated are stable and non-interfering. These two properties simplify the CMOS implementation of production rules. Non-interference will guarantee that the resulting CMOS circuit has no stable states where *Vdd* and *GND* are connected (shorted), and stability will ensure that the circuit has no switching *hazards*. The result is that asynchronous QDI circuits are robust to variations in the underlying fabrication technology, and can be mapped to different fabrication processes with little or no modification.

3.2.1. Digital Abstraction

Boolean variables in our computation are implemented as nodes in the CMOS circuit. The value of the variable is represented by the voltage of the corresponding node. Voltages that are greater than a certain value v_1 are interpreted as logic 1 (**true**), and voltages below a certain value v_0 are interpreted as logic 0 (**false**). The value of v_0 and v_1 vary from node to node.

The following is a highly simplified presentation of CMOS transistors. For a comprehensive

treatment, the reader is referred to other standard texts.[33,40]

A CMOS transistor is either of n-type or of p-type. An n-type transistor can be considered to be a three-terminal device (with the fourth terminal, the "bulk," being held at *GND*), having terminals g, the "gate", and two other terminals x and y. When the voltage of the gate $v(g)$ exceeds $\min(v(x), v(y)) + v_{tn}$, the region between x and y becomes conducting and a current flows between x and y until $v(x) = v(y)$, or $v(g) \leq \min(v(x), v(y)) + v_{tn}$. The value v_{tn} is called the *threshold voltage* of the n-type transistor. For a p-type transistor, when the voltage $v(g)$ is greater than $\max(v(x), v(y)) - v_{tp}$, the region between x and y becomes conducting. The value v_{tp} is the threshold voltage of the p-type transistor. The values of v_{tn} and v_{tp} are technology dependent.

Hence, a transistor approximates a switch that either opens or closes a connection between two electrical nodes. The state of the switch is controlled by the voltage of the gate of the transistor. If a node is connected to *Vdd* or *GND* through a network of transistors, we say that the node is *driven*; otherwise the node is said to be *floating*. A node in a CMOS circuit is said to be *dynamic* if the circuit can enter a state where the node is floating.

Given an n-type transistor with gate g and two other terminals x and y, the effect of $v(g)$ being set to *Vdd* can be easily determined if x is driven and y is floating (or vice-versa). If $v(x) = GND$, then setting $v(g)$ to *Vdd* sets $v(y)$ to *GND*. If $v(x) = Vdd$, the result is to set $v(y)$ to $Vdd - v_{tn}$. For p-type transistors, setting $v(g)$ to *GND* sets $v(y)$ to *Vdd* if $v(x) = Vdd$, and $v(y)$ to v_{tp} if $v(x) = GND$. Observe that even with our simple model, an n-transistor can only raise the voltage of a node to $Vdd - v_{tn}$, and a p-transistor can only lower the voltage of a node to v_{tp}. To prevent such voltage drops, we will use n-transistors exclusively to lower the voltage of a node—in *pull-down* networks, and p-transistors exclusively to raise the voltage of a node—in *pull-up* networks.

3.2.2. Switching Networks

Consider a stable production rule

$$B \mapsto z\downarrow$$

The variables of B are used as control voltages for the gates of a switching network S of transistors that implements the condition B. Since we have to set z to **false** when B is **true**, the two ends of the switching network S are connected to *GND* and z respectively. To avoid any voltage drops, we implement the switching network S exclusively with n-type transistors. We can implement B with a standard series/parallel combination of transistors. Similarly, a production rule of the form

$$B \mapsto z\uparrow$$

is implemented with a switching network that connects z to *Vdd*, where the switching network comprises only p-type transistors.

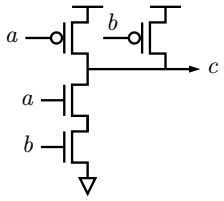

Figure 3.2. CMOS implementation of a NAND gate.

Example. Figure 3.2 shows the CMOS implementation of a NAND gate. The NAND gate is a valid implementation of the following production rule set:

$$\neg a \vee \neg b \;\mapsto\; c\uparrow$$
$$a \wedge b \;\mapsto\; c\downarrow$$

The switching network for the pull-up implements the Boolean expression $\neg a \vee \neg b$ using p-transistors, and the switching network for the pull-down implements expression $a \wedge b$ using n-transistors. ∎

Consider the production rule

$$a \;\mapsto\; c\uparrow$$

We cannot directly implement a switching network with p-type transistors that only uses variable a as a gate voltage. The problem is that we need the *negated version of* a to build the switching network. If $\_a$ were the negated version of a, we could implement the production rule $\neg\_a \mapsto c\uparrow$ without difficulty.

To check whether the guard of a production rule is CMOS-implementable, we write all guards in *negation-normal form*. A guard is said to be in negation-normal form just when negation symbols in the guard only appear on variables. We can convert a guard into negation-normal form using DeMorgan's laws.

Example. The Boolean expression $\neg(a \vee b)$ is not in negation-normal form since it contains a $\neg$ symbol attached to an expression. By applying DeMorgan's laws, we can rewrite it as $\neg a \wedge \neg b$, which is in negation-normal form since $\neg$ symbols only appear on variables. ∎

A production rule $B \mapsto z\uparrow$ with B in negation-normal form is CMOS-implementable just when every variable in B is negated. A production rule $B \mapsto z\downarrow$ with B in negation-normal form is CMOS-implementable just when every variable in B is non-negated.

Example. The production rule $x \wedge y \mapsto z\downarrow$ is CMOS-implementable because variables x and y are non-negated. However, $\neg x \wedge y \vee x \wedge u \mapsto z\downarrow$ is not CMOS-implementable because there is an instance of x that is negated. ∎

3.2.3. Operators

Complementary production rules are implemented as one *operator* (also known as a gate). Consider the set of complementary production rules

$$b_1 \mapsto z\uparrow$$
$$b_2 \mapsto z\downarrow$$

where both b_1 and b_2 are CMOS-implementable by switching networks s_1 and s_2 respectively. The two switching networks are connected by node z. Since the production rules are non-interfering, $\neg b_1 \vee \neg b_2$ is invariant, implying that there is no (digital) state in the computation where Vdd and GND are connected.

An operator is said to be *combinational* whenever $b_1 \vee b_2 \equiv \mathbf{true}$. An operator that is not combinational is said to be *state-holding*.

If an operator is combinational, either b_1 or b_2 holds in any state of the computation. Therefore, either switching network s_1 or switching network s_2 is always conducting. As a result, the node z is always driven. The CMOS implementation of a combinational operator is shown in Figure 3.3. For an operator to be combinational, $b_1 \equiv \neg b_2$, and so the set variables used in the pull-up and pull-down are identical. As a result, whenever $b_1 \mapsto z\uparrow$ ($b_2 \mapsto z\downarrow$) has an effective firing, it is because a variable occurring in b_1 (b_2) changed and made $b_1 \equiv \mathbf{true}$ ($b_2 \equiv \mathbf{true}$) and $b_2 \equiv \mathbf{false}$ ($b_1 \equiv \mathbf{false}$). Since the voltage of the changing variable changes continuously from one logic value to another, there is an intermediate, transient state where both switching networks are partially conducting. This results in short-circuit currents that dissipate power. If the output z is changing to **true** (**false**), there is some opposition to this change by the pull-down (pull-up) network for z because the network is partially conducting. By picking appropriate transistor sizes, we can determine the voltage a changing input must cross before it can affect the output z (see any standard textbook on CMOS design for details). When the p-transistors and n-transistors are sized to have comparable drive strengths, this voltage tends to be close to $Vdd/2$.

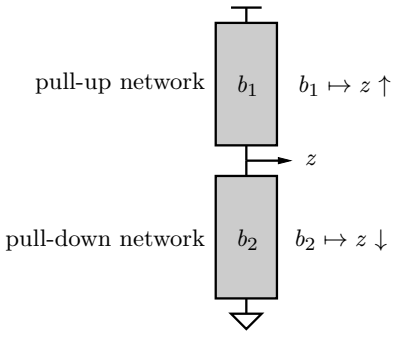

Figure 3.3. CMOS implementation of a combinational operator.

The values v_0 and v_1 for the input to an operator are determined by the transistor sizes used to implement the switching network for the operator.

Consider the case when the operator

$b_1 \mapsto z\uparrow$

$b_2 \mapsto z\downarrow$

is state-holding. If we use the circuit shown in Figure 3.3 as the CMOS implementation of the operator, node z is not driven in the state where $\neg b_1 \wedge \neg b_2$ holds. This is problematic if the state $\neg b_1 \wedge \neg b_2$ is persistent. If this state persists, the charge stored on z can change due to leakage currents thereby changing the value of z without b_1 or b_2 being **true** causing an erroneous execution. The node is also susceptible to noise when it is floating.

There are several ways we can solve this problem. One method is known as the technique of *symmetrization* where we attempt to convert a state-holding operator into a combinational one. This technique is discussed later in this chapter when we talk about compilation. This technique suffers from two drawbacks: it is not always applicable, and it can result in circuits that are much more complicated than the original state-holding operator.

The standard way to make sure that node z is always driven is to use the circuit implementation shown in Figure 3.4. The additional circuitry consists of two cross-coupled inverters attached to node z. The transistors drawn in Figure 3.4 implement a feedback inverter that ensures that node z is always driven. Given the circuitry in Figure 3.4, assume the circuit is in a state in which $z \wedge \neg b_1 \wedge \neg b_2$ holds. The node z is at Vdd, and both switching networks s_1 and s_2 are not conducting. The node $\_z$ (shown in Figure 3.4) is at GND, and the feedback inverter is driving z to Vdd. If the system enters a state where $z \wedge \neg b_1 \wedge b_2$ holds, the pull-down network for z becomes conducting. At this point, the feedback p-transistor is still driving z high. Therefore, we have a state where z is being driven to Vdd and to GND. The final value of z is determined by the electrical properties of the circuit. Since the correct final value of z is **false**, we can ensure that

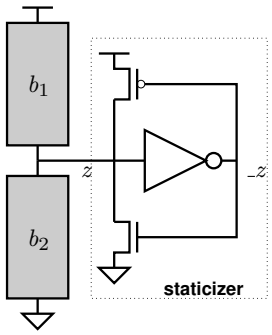

Figure 3.4. Standard implementation of a state-holding operator.

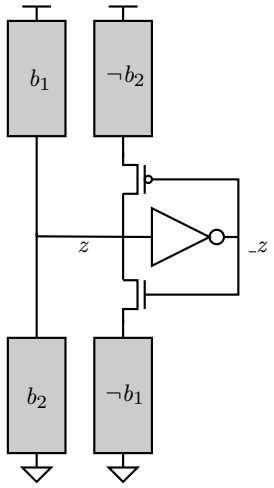

Figure 3.5. Implementation of state-holding operators using combinational feedback.

the network s_2 overcomes the opposing feedback inverter by making the feedback transistors weak compared to the switching networks s_1 and s_2. Given this circuit structure, there is always some opposition to any change in z by the feedback transistors. This opposition helps combat noise, because any perturbation in z will be opposed by the feedback inverter.

The actual change in input voltage necessary for z to change its value depends on the relative strengths of the switching network the input is connected to and the opposing feedback transistor on z. However, since we have to make the feedback transistor weak to ensure the circuit functions correctly, the change in input that is required to change z tends to be small compared to the change required for combinational operators. Therefore, the inputs to state-holding operators implemented as shown in Figure 3.4 are more susceptible to noise. We can change the strength of the feedback transistors to improve the noise margin of the input, but we cannot make the feedback transistors have arbitrary strength.

To avoid this problem, we can adopt the circuit implementation shown in Figure 3.5. In this circuit structure, the opposing feedback transistors are connected in series with a switching network that prevents z from being driven to *Vdd* and *GND* simultaneously. When both b_2 and b_1 are **false**, the switching networks connected in series with the feedback transistors are both conducting, thereby causing z to be driven. Whenever b_1 or b_2 become **true**, the opposing feedback path is cut-off. With this configuration, we can *pick* the voltage an input must cross before it can change z, thereby permitting us to set the noise margin for the inputs to arbitrary values. This solution is known as *combinational feedback*.

The drawback of combinational feedback is that the resulting circuit is much more complex

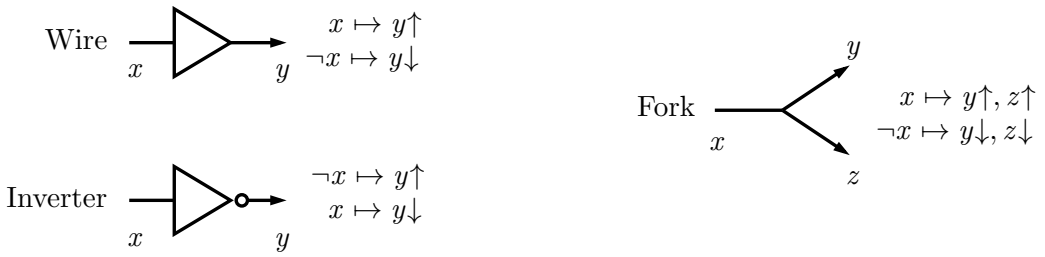

Figure 3.6. One-input operators.

than the simple staticizer shown in Figure 3.4. Therefore, we rarely use this solution. Our preferred implementation strategy for state-holding operators will be the one shown in Figure 3.4. We use combinational feedback only when we need to set the noise margin of the input to a state-holding operator. This level of paranoia is typically necessary when the inputs to the operator are being generated off-chip.

3.2.4. Standard Operators

We list all the standard state-holding and combinational operators in this section, together with their circuit symbols. These operators are frequently encountered in circuit synthesis.

Figure 3.6 shows all the one-input operators together with their circuit symbols. The explicit *wire* between x and y is written $wire(x, y)$. The introduction of an explicit production rule for the wire implies that we can add arbitrary delay between a change in x and the corresponding change in y without affecting the correctness of the computation. The other one-input operators are the inverter and the fork, shown in Figure 3.6.

Figure 3.7 shows all two-input state-holding operators along with their circuit symbols. The C-element (consensus element) is one of the most common state-holding circuit elements. The operator waits for both its inputs to be equal, and then sets its output to be equal to its input.

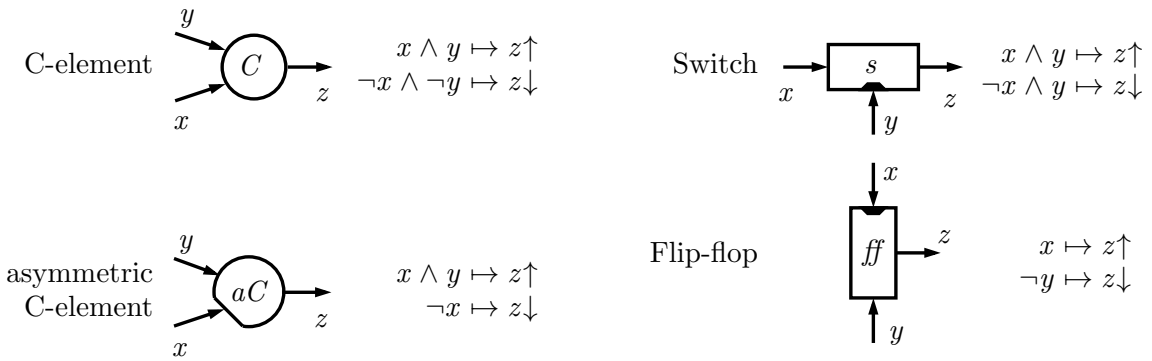

Figure 3.7. Two-input state-holding operators.

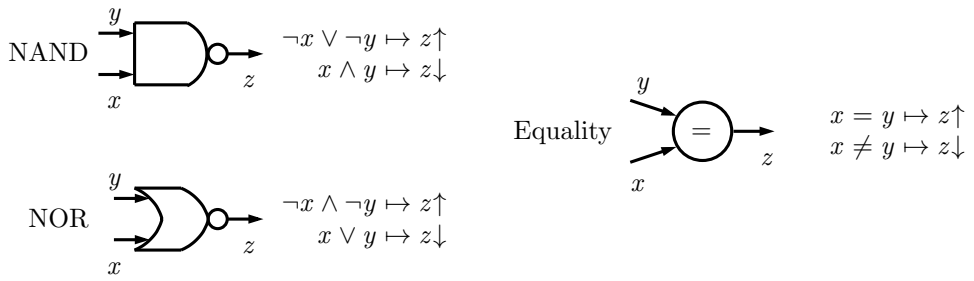

Figure 3.8. Two-input combinational operators.

The asymmetric C-element is similar to a C-element, except one of the inputs only occurs in the production rule for $z\uparrow$. Note that these two operators are not directly CMOS-implementable. However, if we replace $z\uparrow$ with $z\downarrow$ and $z\downarrow$ with $z\uparrow$ in the two production rules, they are CMOS-implementable. This modification results in the *inverting* C-element and *inverting* asymmetric C-element respectively; their circuit symbols have a "bubble" on the output. The other two state-holding operators that are possible are the switch and the flip-flop, shown in Figure 3.7. The C-element, the asymmetric C-element, and the switch are guaranteed to be non-interfering. If a circuit uses the flip-flop, non-interference implies that $\neg x \vee y$ is invariant.

There are three combinational two-input operators: NAND, NOR, and equality. The three operators are shown in Figure 3.8. The inverting equality operator is exclusive-or. Note that while the NAND and NOR operators are CMOS-implementable, the equality operator is not directly implementable in CMOS.

The NAND gate, NOR gate, and C-element can be generalized to N-input operators in the obvious way. The circuit symbol is the same, except there are more than two inputs drawn. The production rules for the N-input NAND, NOR, and C-element are given below:

$$(\vee i :: \neg x_i) \mapsto z\uparrow \qquad (\wedge i :: x_i) \mapsto z\uparrow$$
$$(\wedge i :: x_i) \mapsto z\downarrow \qquad (\wedge i :: \neg x_i) \mapsto z\downarrow$$

$$(\wedge i :: \neg x_i) \mapsto z\uparrow$$
$$(\vee i :: x_i) \mapsto z\downarrow$$

The *generalized C-element* is a special multi-input operator. The pull-up for the generalized C-element is the conjunction of N terms, where each term is either a variable or a negated variable. The pull-down is the conjunction of M terms, each term being a variable or a negated variable. The C-element is a special case of a generalized C-element with the pull-up consisting of the conjunction of non-negated variables, and the pull-down consisting of the conjunction of the negated version of the variables occurring in the pull-up.

Example. The operator

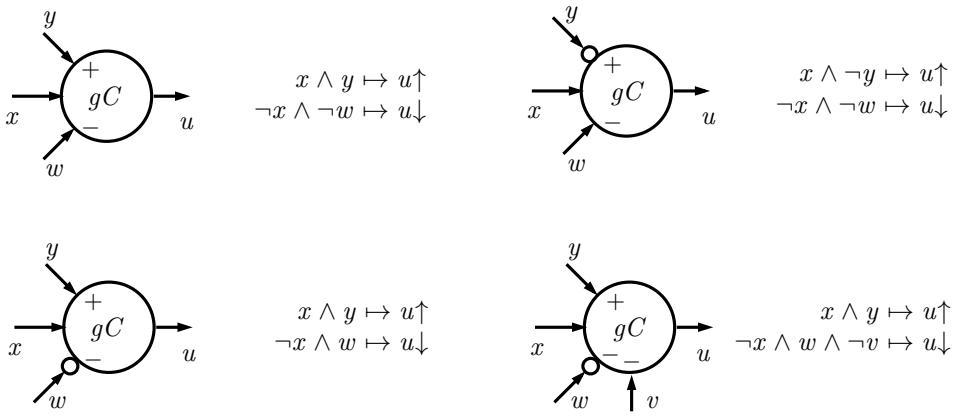

Figure 3.9. Several generalized C-elements and their circuit symbols.

$$a \wedge b \wedge \neg c \;\mapsto\; d\uparrow$$
$$\neg b \wedge e \;\mapsto\; d\downarrow$$

is a generalized C-element. ∎

The circuit symbol for some generalized C-elements are shown in Figure 3.9. When an input is only used in a pull-up, it is drawn with a "+" symbol; if it is only used in the pull-down, it is drawn with a "−" symbol. If the pull-up uses a negated input variable, or the pull-down uses a non-negated input variable, then the input is drawn with a "bubble." If a variable is used in both the pull-up and pull-down and is used with a negation in one and without a negation in the other, then it is shown without either a "+" or a "−."

3.3. Handshaking Expansions

The high-level design of asynchronous VLSI systems is done in CHP; the final target of the synthesis is production rules. *Handshaking expansions* (HSE) are an intermediate form used to aid the compilation of CHP into production rules.

Handshaking expansions are CHP programs with the following restrictions:

- All variables are Boolean-valued;
- There are no communication actions;
- All assignment statements only have constant expressions (**true** or **false**).

The purpose of these restrictions is to close the gap between production rules and CHP. Note that handshaking expansions still permit the use of selection statements, loops, and sequential composition.

The restriction on assignment statements is more a matter of syntax than a restriction. Con-