such as channel length modulation and DIBL. Figure 7-22b plots the transient response for different device sizes and confirms that an individual *W/L* ratio of greater than 3 is required to overpower the feedback and switch the state of the latch.

7.3 Dynamic Latches and Registers

Storage in a static sequential circuit relies on the concept that a cross-coupled inverter pair produces a bistable element and can thus be used to memorize binary values. This approach has the useful property that a stored value remains valid as long as the supply voltage is applied to the circuit—hence the name *static*. The major disadvantage of the static gate, however, is its complexity. When registers are used in computational structures that are constantly clocked (such as a pipelined datapath), the requirement that the memory should hold state for extended periods of time can be significantly relaxed.

This results in a class of circuits based on temporary storage of charge on parasitic capacitors. The principle is exactly identical to the one used in dynamic logic—charge stored on a capacitor can be used to represent a logic signal. The absence of charge denotes a 0, while its presence stands for a stored 1. No capacitor is ideal, unfortunately, and some charge leakage is always present. A stored value can thus only be kept for a limited amount of time, typically in the range of milliseconds. If one wants to preserve signal integrity, a periodic *refresh* of the value is necessary; hence, the name *dynamic* storage. Reading the value of the stored signal from a capacitor without disrupting the charge requires the availability of a device with a high-input impedance.

7.3.1 Dynamic Transmission-Gate Edge-Triggered Registers

A fully dynamic positive edge-triggered register based on the master–slave concept is shown in Figure 7-23. When $CLK = 0$, the input data is sampled on storage node 1, which has an equivalent capacitance of C_1, consisting of the gate capacitance of I_1, the junction capacitance of T_1, and the overlap gate capacitance of T_1. During this period, the slave stage is in a hold mode, with node 2 in a high-impedance (floating) state. On the rising edge of clock, the transmission gate T_2 turns on, and the value sampled on node 1 right before the rising edge propagates to the output Q (note that node 1 is stable during the high phase of the clock, since the first transmission gate is

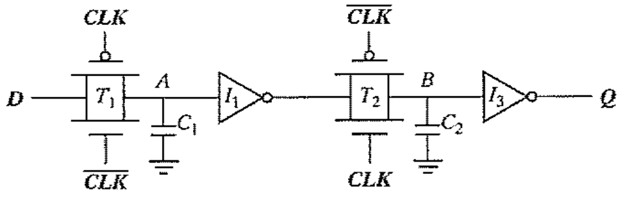


Figure 7-23 Dynamic edge-triggered register.

turned off). Node 2 now stores the inverted version of node 1. This implementation of an edge-triggered register is very efficient because it requires only eight transistors. The sampling switches can be implemented using NMOS-only pass transistors, resulting in an even simpler six transistor implementation. The reduced transistor count is attractive for high-performance and low-power systems.

The setup time of this circuit is simply the delay of the transmission gate, and it corresponds to the time it takes node 1 to sample the D input. The hold time is approximately zero, since the transmission gate is turned off on the clock edge and further inputs changes are ignored. The propagation delay (t_{c-q}) is equal to two inverter delays plus the delay of the transmission gate T_2.

One important consideration for such a dynamic register is that the storage nodes (i.e., the state) have to be refreshed at periodic intervals to prevent
losses due to charge leakage, diode leakage, or subthreshold currents. In datapath circuits, the refresh rate is not an issue, since the registers are periodically clocked, and the storage nodes are constantly updated.

Clock overlap is an important concern for this register. Consider the clock waveforms shown in Figure 7-24. During the 0–0 overlap period, the NMOS of T_1 and the PMOS of T_2 are simultaneously on, creating a direct path for data to flow from the D input of the register to the Q output. In other words, a *race condition* occurs. The output Q can change on the falling edge if the overlap period is large—obviously an undesirable effect for a positive edge-triggered register. The same is true for the 1–1 overlap region, where an input-output path exists through the PMOS of T_1 and the NMOS of T_2. The latter case is taken care of by enforcing a *hold* time constraint. That is, the data must be stable during the high-overlap period. The former situation (0–0 overlap) can be addressed by making sure that there is enough delay between the D input and node B, ensuring that new data sampled by the master stage does not propagate through to the slave stage. Generally, the built-in single inverter delay should be sufficient. The overlap period constraint is given by

$$t_{overlap0-0} < t_{T1} + t_{I1} + t_{T2} \tag{7.5}$$

Similarly, the constraint for the 1–1 overlap is given as:

$$t_{hold} > t_{overlap1-1} \tag{7.6}$$

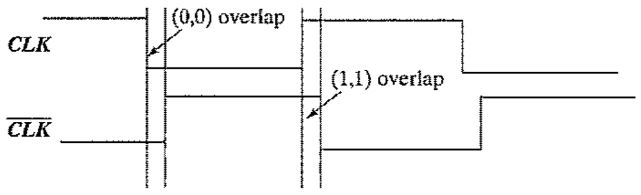


Figure 7-24 Impact of nonoverlapping clocks.

WARNING: The dynamic circuits shown in this section are very appealing from the perspective of complexity, performance, and power. Unfortunately, robustness considerations limit their use. In a fully dynamic circuit like that shown in Figure 7-23, a signal net that is capacitively coupled to the internal storage node can inject significant noise and destroy the state. This is especially important in ASIC flows, where there is little control over coupling between signal nets and internal dynamic nodes. Leakage currents cause another problem: Most modern processors require that the clock can be slowed down or completely halted, to conserve power in low-activity periods. Finally, the internal dynamic nodes do not track variations in power supply voltage. For example, when *CLK* is high for the circuit in Figure 7-23, node *A* holds its state, but it does not track variations in the power supply seen by I_1. This results in reduced noise margins.

Most of these problems can be adequately addressed by adding a weak feedback inverter and making the circuit *pseudostatic* (Figure 7-25). While this comes at a slight cost in delay, it improves the noise immunity significantly. Unless registers are used in a highly-controlled environment (for instance, a custom-designed high-performance datapath), they should be made pseudostatic or static. This holds for all latches and registers discussed in this section.

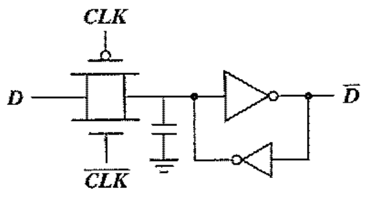


Figure 7-25 Making a dynamic latch pseudostatic.

7.3.2 C²MOS—A Clock-Skew Insensitive Approach

The C²MOS Register

Figure 7-26 shows an ingenious positive edge-triggered register that is based on a master–slave concept insensitive to clock overlap. This circuit is called the *C^2MOS* (Clocked CMOS) *register* [Suzuki73], and operates in two phases:

1. $CLK = 0$ ($\overline{CLK} = 1$): The first tristate driver is turned on, and the master stage acts as an inverter sampling the inverted version of *D* on the internal node *X*. The master stage is in the evaluation mode. Meanwhile, the slave section is in a high-impedance mode, or in a hold mode. Both transistors M_7 and M_8 are off, decoupling the output from the input. The output *Q* retains its previous value stored on the output capacitor C_{L2}.
2. The roles are reversed when $CLK = 1$: The master stage section is in hold mode (M_3-M_4 off), while the second section evaluates (M_7–M_8 on). The value stored on C_{L1} propagates to the output node through the slave stage, which acts as an inverter.

The overall circuit operates as a positive edge-triggered master–slave register very similar to the transmission-gate-based register presented earlier. However, there is an important difference:

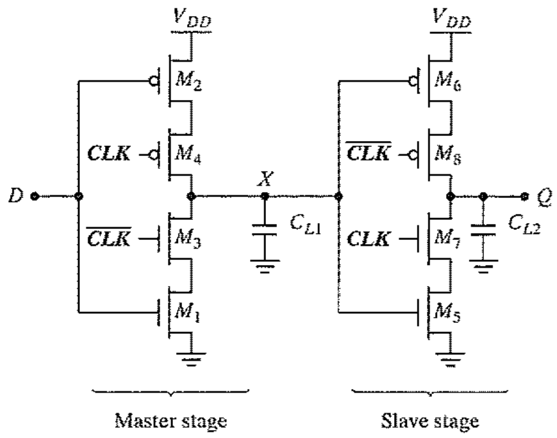


Figure 7-26 C^2MOS master–slave positive edge-triggered register.

A C^2MOS register with CLK–$\overline{CLK}$ clocking is insensitive to overlap, as long as the rise and fall times of the clock edges are sufficiently small.

To prove this statement, we examine both the (0–0) and (1–1) overlap cases (see Figure 7-24). In the (0–0) overlap case, the circuit simplifies to the network shown in Figure 7-27a in which both PMOS devices are *on* during this period. To operate correctly, none of the new data sampled during the overlap window should propagate to the output Q, since data should not change on the negative edge of a positive edge-triggered register. Indeed, new data is sampled on node X through the series PMOS devices M_2–M_4, and node X can make a 0-to-1 transition during the overlap period. However, this data cannot propagate to the output since the NMOS device M_7 is turned off. At the end of the overlap period, $\overline{CLK} = 1$ and both M_7 and M_8 turn off, putting the slave stage in the hold mode. Therefore, any new data sampled on the falling clock edge is not seen at the slave output Q, since the slave state is off till the next rising edge of the clock. As the circuit consists of a cascade of inverters, signal propagation requires one pull-up followed by a pull-down, or vice versa, which is not feasible in the situation presented.

The (1–1) overlap case where both NMOS devices M_3 and M_7 are turned on, is somewhat more contentious (see Figure 7-27b). The question is again if new data sampled during the overlap period (right after clock goes high) propagates to the Q output. A positive edge-triggered register may only pass data that is presented at the input before the rising edge. If the D input changes during the overlap period, node X can make a 1-to-0 transition, but cannot propagate further. However, as soon as the overlap period is over, the PMOS M_8 turns on and the 0 propagates tooutput, which is not desirable. The problem is fixed by imposing a hold-time constraint on the input data, D; or, in other words, the data D should be stable during the overlap period.

In sum, it can be stated that the C^2MOS latch is insensitive to clock overlaps because those overlaps activate either the pull-up or the pull-down networks of the latches, but never both of them simultaneously. If the rise and fall times of the clock are sufficiently slow, however, there

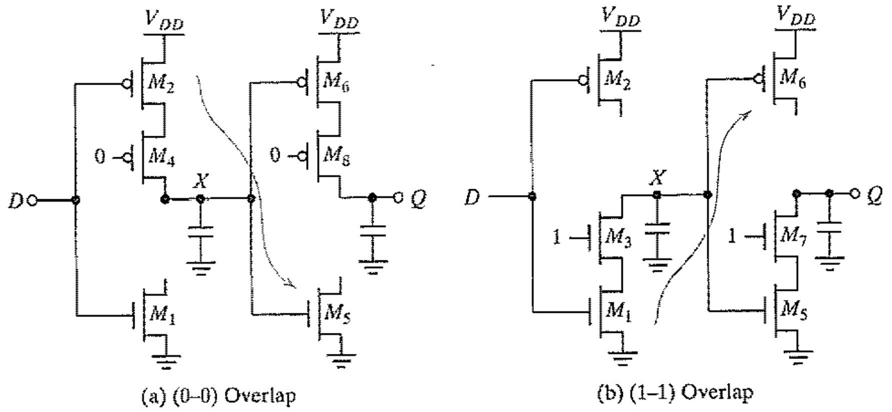


Figure 7-27 C^{2}MOS *D* FF during overlap periods. No feasible signal path can exist between *In* and *D*, as illustrated by the arrows.

exists a time slot where both the NMOS and PMOS transistors are conducting. This creates a path between input and output that can destroy the state of the circuit. Simulations have shown that the circuit operates correctly as long as the clock rise time (or fall time) is smaller than approximately five times the propagation delay of the register. This criterion is not too stringent, and it is easily met in practical designs. The impact of the rise and fall times is illustrated in Figure 7-28, which plots the simulated transient response of a C^{2}MOS *D* FF for clock slopes of, respectively, 0.1 and 3 ns. For slow clocks, the potential for a *race condition* exists.

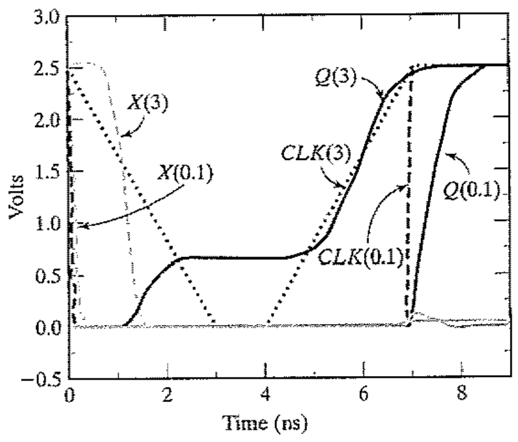


Figure 7-28 Transient response of C^{2}MOS FF for 0.1-ns and 3-ns clock rise/fall times, assuming *In* = 1.

Dual-Edge Registers

So far, we have focused on edge-triggered registers that sample the input data on only one of the clock edges (rising or falling). It also is possible to design sequential circuits that sample the input on both edges. The advantage of this scheme is that a lower frequency clock—half the original rate—is distributed for the same functional throughput, resulting in power savings in the clock distribution network. Figure 7-29 shows a modification of the C^2MOS register enabling sampling on both edges. It consists of two parallel master–slave edge-triggered registers, whose outputs are multiplexed by using tristate drivers.

When clock is high, the positive latch composed of transistors M_1–M_4 is sampling the inverted D input on node X. Node Y is held stable, since devices M_9 and M_{10} are turned off. On the falling edge of the clock, the top slave latch M_5–M_8 turns on, and drives the inverted value of X to the Q output. During the low phase, the bottom master latch (M_1, M_4, M_9, M_{10}) is turned on, sampling the inverted D input on node Y. Note that the devices M_1 and M_4 are reused, reducing the load on the D input. On the rising edge, the bottom slave latch conducts and drives the inverted version of Y on node Q. Data thus changes on both edges. Note that the slave latches operate in a complementary fashion—that is, only one of them is turned on during each phase of the clock.

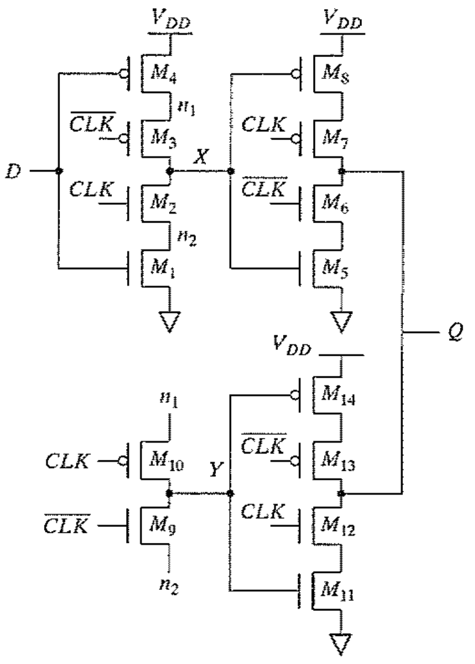


Figure 7-29 C^2MOS-based dual-edge triggered register.

Problem 7.5 Dual-Edge Registers

Determine how the adoption of dual-edge registers influences the power dissipation in the clock-distribution network.

7.3.3 True Single-Phase Clocked Register (TSPCR)

In the two-phase clocking schemes described earlier, care must be taken in routing the two clock signals to ensure that overlap is minimized. While the C^2MOS provides a skew-tolerant solution, it is possible to design registers that only use a single phase clock. The *True Single-Phase Clocked Register* (TSPCR), proposed by Yuan and Svensson, uses a **single clock** [Yuan89]. The basic single-phase positive and negative latches are shown in Figure 7-30. For the positive latch, when *CLK* is high, the latch is in the transparent mode and corresponds to two cascaded inverters; the latch is noninverting, and propagates the input to the output. On the other hand, when $CLK = 0$, both inverters are disabled, and the latch is in hold mode. Only the pull-up networks are still active, while the pull-down circuits are deactivated. As a result of the dual-stage approach, no signal can ever propagate from the input of the latch to the output in this mode. A register can be constructed by cascading positive and negative latches. The clock load is similar to a conventional transmission gate register, or C^2MOS register. The main advantage is the use of a single clock phase. The disadvantage is the slight increase in the number of transistors—12 transistors are now required.

As a reminder, note that a dynamic circuit in the style of Figure 7-30 must be used with caution. When the clock is low (for the positive latch), the output node may be floating, and it is exposed to coupling from other signals. Also, charge sharing can occur if the output node drives transmission gates. Dynamic nodes should be isolated with the aid of static inverters, or made pseudostatic for improved noise immunity.

As with many other latch families, TSPC offers an additional advantage that we have not explored so far: The possibility of embedding logic functionality into the latches. This reduces

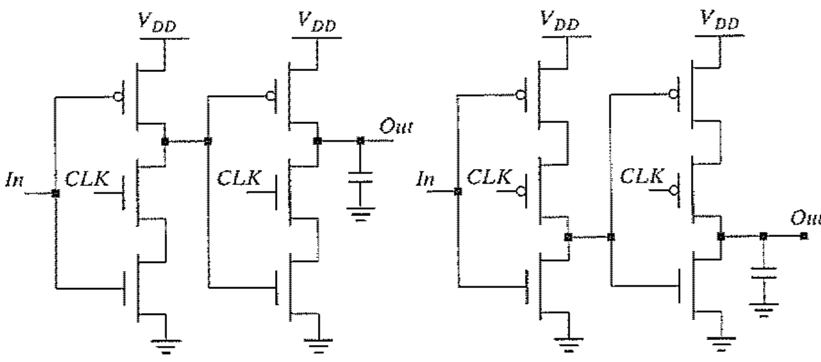


Figure 7-30 True Single-Phase Latches.

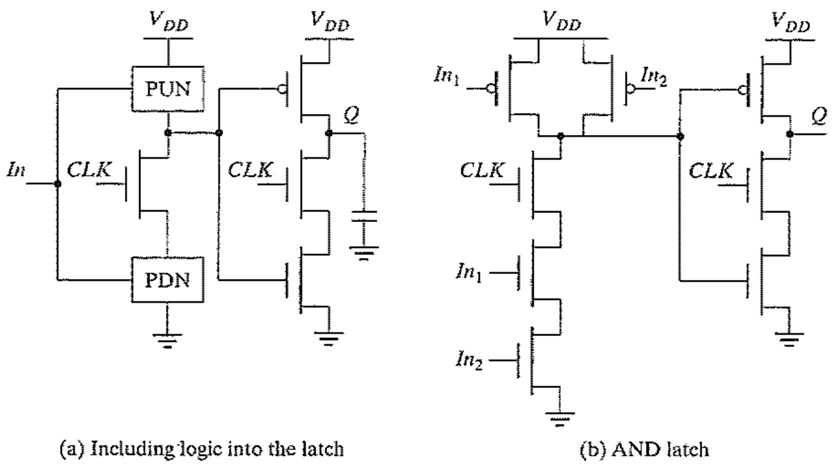


Figure 7-31 Adding logic to the TSPC approach.

the delay overhead associated with the latches. Figure 7-31a outlines the basic approach for embedding logic, while Figure 7-31b shows an example of a positive latch that implements the AND of In_1 and In_2 in addition to performing the latching function. While the setup time of this latch has increased over the one shown in Figure 7-30, the overall performance of the digital circuit (that is, the clock period of a sequential circuit) has improved: The increase in setup time typically is smaller than the delay of an AND gate. This approach of embedding logic into latches has been used extensively in the design of the EV4 DEC Alpha microprocessor [Dobberpuhl92] and many other high-performance processors.

Example 7.3 Impact of Embedding Logic into Latches on Performance

Consider embedding an AND gate into the TSPC latch, as shown in Figure 7-31b. In a 0.25-μm technology, the setup time of such a circuit, using minimum-size devices is 140 ps. A conventional approach, composed of an AND gate followed by a positive latch, has an effective setup time of 600 ps (we treat the AND plus latch as a black box that performs both functions). The embedded logic approach thus results in significant performance improvements.

The TSPC latch circuits can be further reduced in complexity, as illustrated in Figure 7-32, where only the first inverter is controlled by the clock. Besides the reduced number of transistors, these circuits have the advantage that the clock load is reduced by half. On the other hand, not all node voltages in the latch experience the full logic swing. For instance, the voltage at node A (for V_{in} = 0 V) for the positive latch maximally equals $V_{DD} - V_{Tn}$, which results in a reduced drive for the output NMOS transistor and a loss in performance. Similarly, the voltage

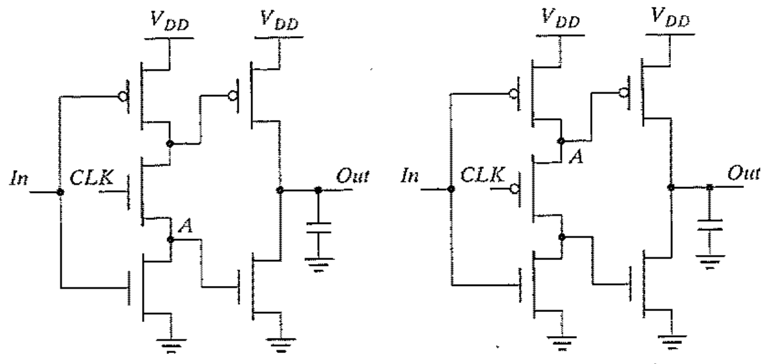


Figure 7-32 Simplified TSPC latch (also called split output).

on node A (for $V_{in} = V_{DD}$) for the negative latch is only driven down to $|V_{TP}|$ .This also limits the amount of V_{DD} scaling possible on the latch.

Figure 7-33 shows the design of a specialized single-phase edge-triggered register. When $CLK = 0$, the input inverter is sampling the inverted D input on node X. The second (dynamic) inverter is in the precharge mode, with M_6 charging up node Y to V_{DD}. The third inverter is in the hold mode, since M_8 and M_9 are off. Therefore, during the low phase of the clock, the input to the final (static) inverter is holding its previous value and the output Q is stable. On the rising edge of the clock, the dynamic inverter M_4–M_6 evaluates. If X is high on the rising edge, node Y discharges. The third inverter M_7–M_9 is on during the high phase, and the node value on Y is passed to the output Q. On the positive phase of the clock, note that node X transitions to a low if the D input transitions to a high level. Therefore, the input must be kept stable until the value on node X before the rising edge of the clock propagates to Y. This represents the hold time of the register (note that the hold time is less than 1 inverter delay, since it takes 1 delay for the input to affect node X). The propagation delay of the register is essentially three inverters, because the

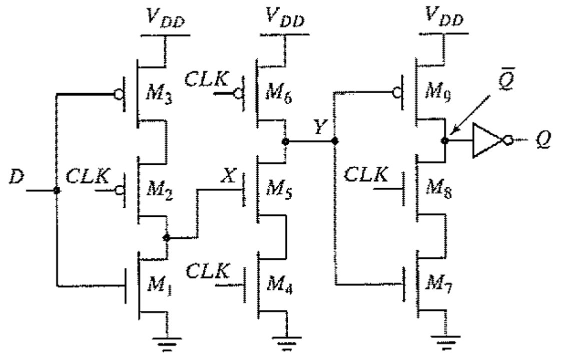


Figure 7-33 Positive edge-triggered register in TSPC.

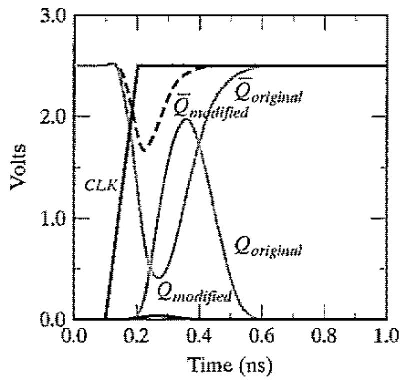
value on node X must propagate to the output Q. Finally, the setup time is the time for node X to be valid, which is one inverter delay.

WARNING: Similar to the C^2MOS latch, the TSPC latch malfunctions when the *slope of the clock* is not sufficiently steep. Slow clocks cause both the NMOS and PMOS clocked transistors to be on simultaneously, resulting in undefined values of the states and race conditions. The clock slopes should therefore be carefully controlled. If necessary, local buffers must be introduced to ensure the quality of the clock signals.

Example 7.4 TSPC Edge-Triggered Register

Transistor sizing is critical for achieving correct functionality in the TSPC register. With improper sizing, glitches may occur at the output due to a *race condition* when the clock transitions from low to high. Consider the case where D is low and $\overline{Q} = 1$ ($Q = 0$). While CLK is low, Y is precharged high turning on M_7. When CLK transitions from low to high, nodes Y and $\overline{Q}$ start to discharge simultaneously (through M_4–M_5 and M_7–M_8, respectively). Once Y is sufficiently low, the trend on $\overline{Q}$ is reversed and the node is pulled high again through M_9. In a sense, this sequence of events is comparable to what happens when we chain dynamic logic gates. Figure 7-34 shows the transient response of the circuit of Figure 7-33 for different sizes of devices in the final two stages.

This glitch may be the cause of fatal errors, because it may create unwanted events (for instance, when the output of the latch is used as a clock signal input to another register). It also reduces the contamination delay of the register. The problem can be corrected by resizing the relative strengths of the pull-down paths through M_4–M_5 and M_7–M_8, so that Y discharges much faster than $\overline{Q}$. This is accomplished by reducing the strength of the M_7–M_8 pull-down path, and by speeding up the M_4–M_5 pull-down path.



	M_4, M_5	M_7, M_8
Original Width	0.5 μm	2 μm
Modified Width	1 μm	1 μm

Figure 7-34 Transistor sizing issues in TSPC (for the register of Figure 7-33).

7.4 Alternative Register Styles*

7.4.1 Pulse Registers

Until now, we have used the master–slave configuration to create an edge-triggered register. A fundamentally different approach for constructing a register uses *pulse signals*. The idea is to construct a short pulse around the rising (or falling) edge of the clock. This pulse acts as the clock input to a latch (for example, Figure 7-35a), sampling the input only in a short window. Race conditions are thus avoided by keeping the opening time (i.e, the transparent period) of the latch very short. The combination of the glitch-generation circuitry and the latch results in a positive edge-triggered register.

Figure 7-35b shows an example circuit for constructing a short intentional glitch on each rising edge of the clock [Kozu96]. When $CLK = 0$, node X is charged up to V_{DD} (M_N is off since $CLKG$ is low). On the rising edge of the clock, there is a short period of time when both inputs of the AND gate are high, causing $CLKG$ to go high. This in turn activates M_N, pulling X and eventually $CLKG$ low (Figure 7-35c). The length of the pulse is controlled by the delay of the AND gate and the two inverters. Note that there exists also a delay between the rising edges of the input clock (*CLK*) and the glitch clock (*CLKG*), which also is equal to the delay of the AND gate and the two inverters. If every register on the chip uses the same clock generation mechanism, this sampling delay does not matter. However, process variations and load variations may

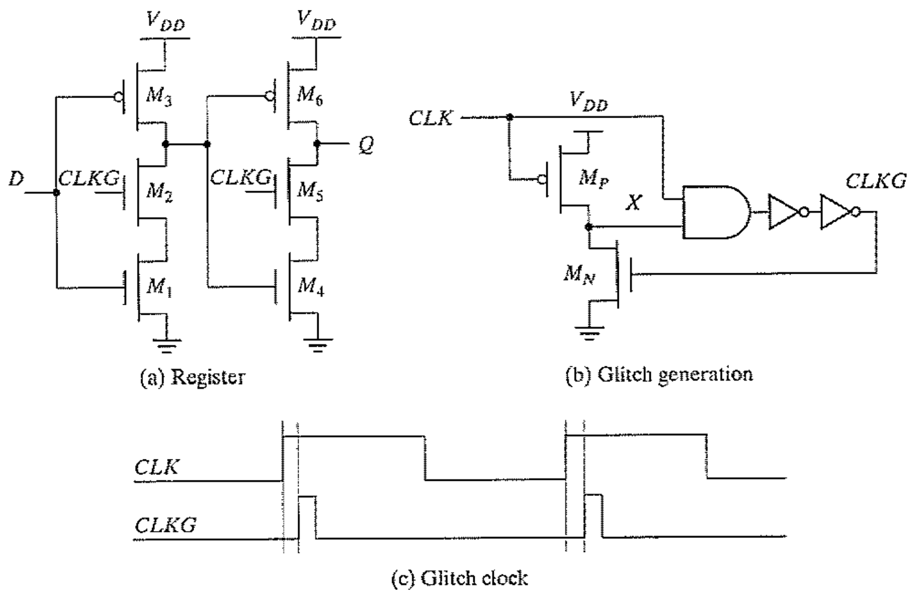


Figure 7-35 TSPC-based glitch latch-timing generation and register.

Dell Ex. 1025
Page 243

cause the delays through the glitch clock circuitry to be different. This must be taken into account when performing timing verification and clock skew analysis (the topics of Chapter 10).

If the setup time and hold time are measured in reference to the rising edge of the glitch clock, the setup time is essentially zero, the hold time is essentially equal to the length of the pulse, and the propagation delay (t_{c-q}) equals two gate delays. The advantage of the approach is the **reduced clock load and the small number of transistors** required. The glitch-generation circuitry can be amortized over multiple register bits. The disadvantage is a substantial increase in verification complexity. For this circuit to function properly, simulations must be performed across all corners to ensure that the clock pulse always exists (i.e., that the glitch-generation circuit works reliably). Despite the increased complexity, such registers do provide an alternate approach to conventional schemes, and they have been adopted in a number of high-performance processors (e.g., [Kozu96]).

Another version of the pulsed register is shown in Figure 7-36 (as used in the AMD-K6 processor [Partovi96]). When the clock is low, M_3 and M_6 are off, and device P_1 is turned on. Node X is precharged to V_{DD}, the output node (Q) is decoupled from X and is held at its previous state. $\overline{CLKD}$ is a delay-inverted version of CLK. On the rising edge of the clock, M_3 and M_6 turn on while devices M_1 and M_4 stay on for a short period, determined by the delay of the three inverters. During this interval, the circuit is transparent and the input data D is sampled by the latch. Once $\overline{CLKD}$ goes low, node X is decoupled from the D input and is either held or starts to precharge to V_{DD} through PMOS device P_2. On the falling edge of the clock, node X is held at V_{DD} and the output is held stable by the cross-coupled inverters.

Note that this circuit also uses a *pulse generator*, but it is integrated into the register. The transparency period also determines the hold time of the register. The window must be wide enough for the input data to propagate to the Q output. In this particular circuit, the setup time can be negative. This is the case if the transparency window is longer than the delay from input to output. This is attractive, as data can arrive at the register even after the clock goes high, which means that time is borrowed from the previous cycle.

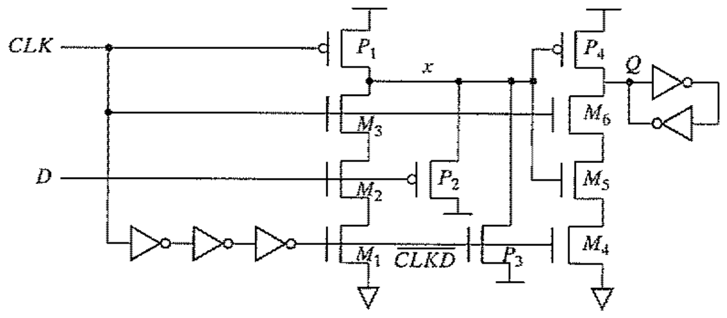


Figure 7-36 Flow-through positive edge-triggered register.

Example 7.5 Setup Time of Glitch Register

The glitch register of Figure 7-36 is transparent during the (1–1) overlap of *CLK* and $\overline{CLKD}$. As a result, the input data can actually change after the rising edge of the clock, resulting in a negative setup time (Figure 7-37). The *D*-input transitions to low after the rising edge of the clock, and transitions to high before the falling edge of $\overline{CLKD}$ (i.e., during the transparency period). Observe how the output follows the input. The output *Q* does go to the correct value of V_{DD} as long as the input *D* is set up correctly some time before the falling edge of $\overline{CLKD}$. When the negative setup time is exploited, there can be no guarantees on the monotonic behavior of the output. That is, the output can have multiple transitions around the rising edge, and therefore, the output of the register should not be used for driving dynamic logic or as a clock as a clock to other registers.

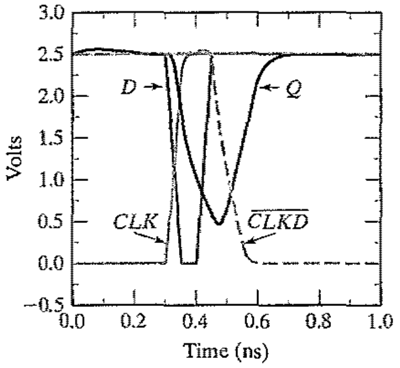


Figure 7-37 Simulation showing a negative setup time for the glitch register.

Problem 7.6 Converting a Glitch Register to a Conditional Glitch Register

Modify the circuit in Figure 7-36 so that it takes an additional *Enable* input. The goal is to convert the register to a conditional register which latches only when the enable signal is asserted.

7.4.2 Sense-Amplifier-Based Registers

In addition to the *master–slave* and the *glitch* approaches to implement an edge-triggered register, a third technique based on *sense amplifiers* can be used, as introduced in Figure 7-38 [Montanaro96].[3] Sense-amplifier circuits accept small input signals and amplify them to generate rail-to-rail swings. They are used extensively in memory cores and in low-swing bus drivers to either improve performance or reduce power dissipation. There are many techniques to construct these amplifiers. A common approach is to use feedback—for instance, through a set of

[3]In a sense, these sense-amplifier-based registers are similar in operation to the glitch registers—that is, the first stage generates the pulse, and the second latches it.

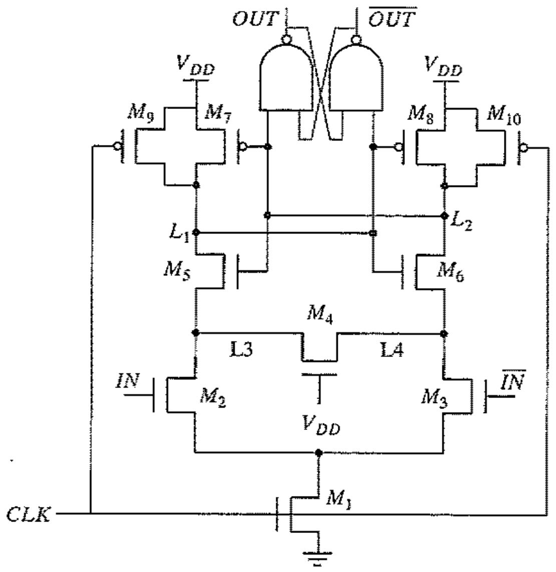


Figure 7-38 Positive edge-triggered register based on sense amplifier.

cross-coupled inverters. The circuit shown in Figure 7-38 uses a precharged front-end amplifier that samples the differential input signal on the rising edge of the clock signal. The outputs of front end are fed into a NAND cross-coupled SR *flip-flop* that holds the data and guarantees that the differential outputs switch only once per clock cycle. The differential inputs in this implementation don't have to have rail-to-rail swing.

The core of the front end consists of a cross-coupled inverter (M_5–M_8), whose outputs (L_1 and L_2) are precharged by using devices M_9 and M_{10} during the low phase of the clock. As a result, PMOS transistors M_7 and M_8 are turned off and the NAND flip-flop is holding its previous state. Transistor M_1 is similar to an evaluate switch in dynamic circuits and is turned off to ensure that the differential inputs do not affect the output during the low phase of the clock. On the rising edge of the clock, the evaluate transistor turns on and the differential input pair (M_2 and M_3) is enabled, and the difference between the input signals is amplified on the output nodes on L_1 and L_2. The cross-coupled inverter pair flips to one of its stable states based on the value of the inputs. For example, if *IN* is 1, L_1 is pulled to 0, and L_2 remains at V_{DD}. Due to the amplifying properties of the input stage, it is not necessary for the input to swing all the way up to V_{DD}, which enables the use of low-swing signaling on the input wires.

The shorting transistor, M_4, is used to provide a DC-leakage path from either node L_3, or L_4, to ground. This is necessary to accommodate the case in which the inputs change their value after the positive edge of *CLK* has occurred, resulting in either L_3 or L_4 being left in a high-impedance state with a logical low-voltage level stored on the node. Without the leakage path, that node would be susceptible to charging by leakage currents. The latch could then actually change state prior to the next rising edge of *CLK*! This is best illustrated graphically, as in Figure 7-39.

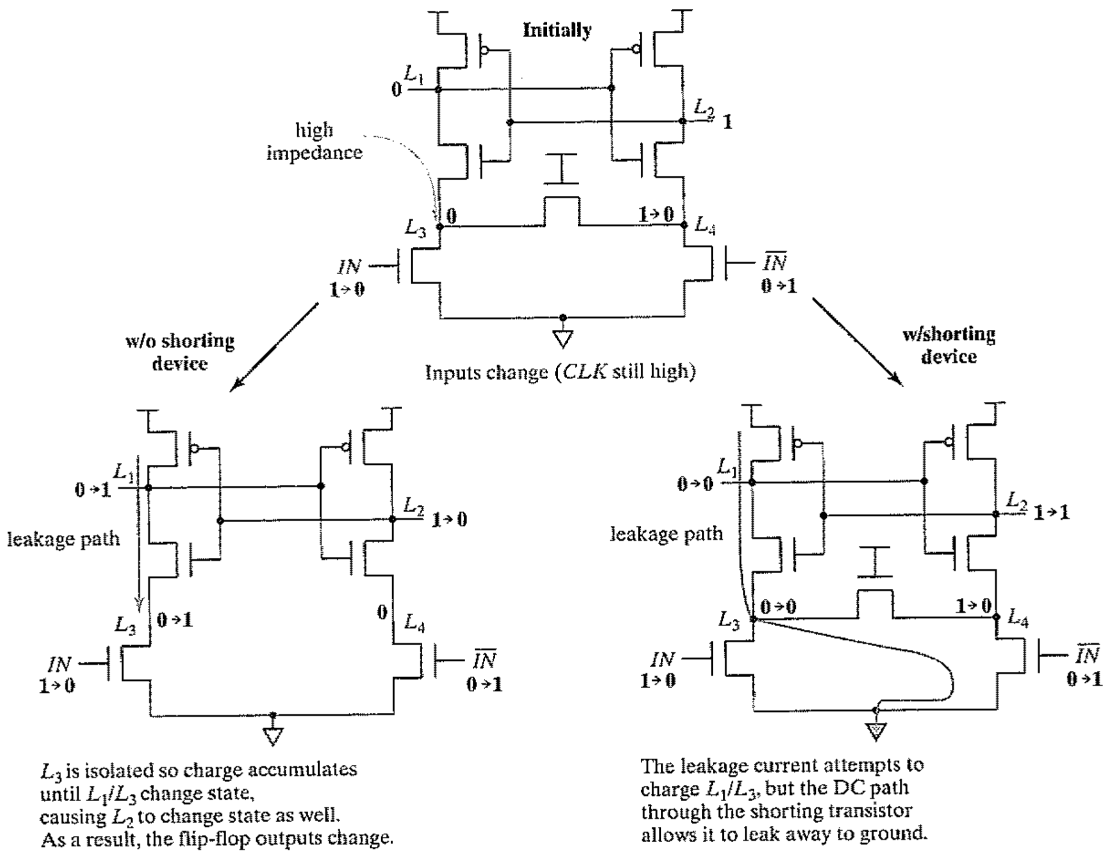

Initially
L_1
0
L_2 1
high impedance
L_3 0
1→0 L_4
IN 1→0
$\overline{IN}$ 0→1
w/o shorting device
Inputs change (CLK still high)
w/shorting device
0→1 L_1
leakage path
L_2 1→0
L_3 0→1
0 L_4
IN 1→0
$\overline{IN}$ 0→1
0→0 L_1
leakage path
L_2 1→1
L_3 0→0
1→0 L_4
IN 1→0
$\overline{IN}$ 0→1
L_3 is isolated so charge accumulates until L_1/L_3 change state, causing L_2 to change state as well. As a result, the flip-flop outputs change.
The leakage current attempts to charge L_1/L_3, but the DC path through the shorting transistor allows it to leak away to ground.

Figure 7-39 The need for the shorting transistor M_4.

7.5 Pipelining: An Approach to Optimize Sequential Circuits

Pipelining is a popular design technique often used to accelerate the operation of datapaths in digital processors. The concept is explained with the example of Figure 7-40a. The goal of the presented circuit is to compute log(|$a + b$|), where both a and b represent streams of numbers (i.e., the computation must be performed on a large set of input values). The minimal clock period T_{min} necessary to ensure correct evaluation is given as

$$T_{min} = t_{c-q} + t_{pd,logic} + t_{su} \tag{7.7}$$

where t_{c-q} and t_{su} are the propagation delay and the setup time of the register, respectively. We assume that the registers are edge-triggered D registers. The term $t_{pd,logic}$ stands for the worst case delay path through the combinational network, which consists of the adder, absolute value, and logarithm functions. In conventional systems (that don't push the edge of technology), the

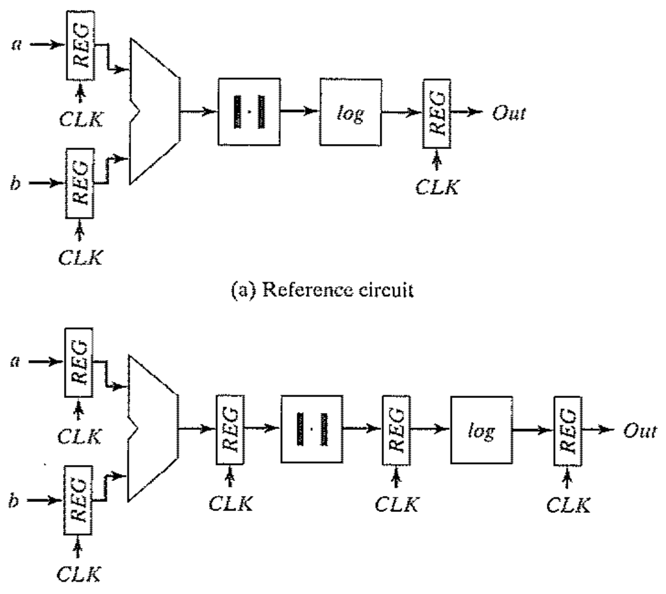


Figure 7-40 Datapath for the computation of log(|*a* + *b*|).

latter delay is generally much larger than the delays associated with the registers and dominates the circuit performance. Assume that each logic module has an equal propagation delay. We note that each logic module is then active for only one-third of the clock period (if the delay of the register is ignored). For example, the adder unit is active during the first third of the period and remains idle (no useful computation) during the other two-thirds of the period. Pipelining is a technique to improve the resource utilization, and increase the functional through-put. Assume that we introduce registers between the logic blocks, as shown in Figure 7-40b. This causes the computation for one set of input data to spread over a number of clock-periods, as shown in Table 7-1. The result for the data set (a_1, b_1) only appears at the output after three clock periods.

Table 7-1 Example of pipelined computations.

Clock Period	Adder	Absolute Value	Logarithm
1	$a_1 + b_1$		
2	$a_2 + b_2$	$\lvert a_1 + b_1 \rvert$	
3	$a_3 + b_3$	$\lvert a_2 + b_2 \rvert$	$\log(\lvert a_1 + b_1 \rvert)$
4	$a_4 + b_4$	$\lvert a_3 + b_3 \rvert$	$\log(\lvert a_2 + b_2 \rvert)$
5	$a_5 + b_5$	$\lvert a_4 + b_4 \rvert$	$\log(\lvert a_3 + b_3 \rvert)$

At that time, the circuit has already performed parts of the computations for the next data sets, (a_2, b_2) and (a_3, b_3). The computation is performed in an assembly-line fashion—hence the name *pipeline*.

The advantage of pipelined operation becomes apparent when examining the minimum clock period of the modified circuit. The combinational circuit block has been partitioned into three sections, each of which has a smaller propagation delay than the original function. This effectively reduces the value of the minimum allowable clock period:

$$T_{min,pipe} = t_{c-q} + \max(t_{pd,add}, t_{pd,abs}, t_{pd,log}) + t_{su} \tag{7.8}$$

Suppose that all logic blocks have approximately the same propagation delay, and that the register overhead is small with respect to the logic delays. The pipelined network outperforms the original circuit by a factor of three under these assumptions (i.e., $T_{min,pipe} = T_{min}/3$). The increased performance comes at the relatively small cost of two additional registers and an increased latency.[4] This explains why pipelining is popular in the implementation of very high-performance datapaths.

7.5.1 Latch- versus Register-Based Pipelines

Pipelined circuits can be constructed by using level-sensitive latches instead of edge-triggered registers. Consider the pipelined circuit of Figure 7-41. The pipeline system is implemented using pass-transistor-based positive and negative latches instead of edge-triggered registers. That is, logic is introduced between the master and slave latches of a master–slave system. In the following discussion, we use the CLK–$\overline{CLK}$ notation to denote a two-phase clock system without loss of generality. Latch-based systems give significantly more flexibility in implementing a pipelined system, and they often offer higher performance. When the CLK and $\overline{CLK}$ clocks are nonoverlapping, correct pipeline operation is obtained. Input data is sampled on C_1 at the negative edge of CLK and the computation of logic block F starts; the result of the logic block F is stored on C_2 on the falling edge of $\overline{CLK}$, and the computation of logic block G starts. The nonoverlapping of the clocks ensures correct operation. The value stored on C_2 at the end of the CLK low phase is the result of passing the previous input (stored on the falling edge of CLK on C_1) through the logic function F. When overlap exists between CLK and $\overline{CLK}$, the next input is already being applied to F, and its effect might propagate to C_2 before $\overline{CLK}$ goes low (assuming that the contamination delay of F is small). In other words, a *race* develops between the previous input and the current one. Which value wins depends upon the logic and is often a function of the applied inputs. The latter factor makes the detection and elimination of race conditions nontrival in nature.

[4] *Latency* is defined here as the number of clock cycles it takes for the data to propagate from the input to the output. For the example at hand, pipelining increases the latency from 1 to 3. An increased latency is generally acceptable, but it can cause a global performance degradation if not treated with care.

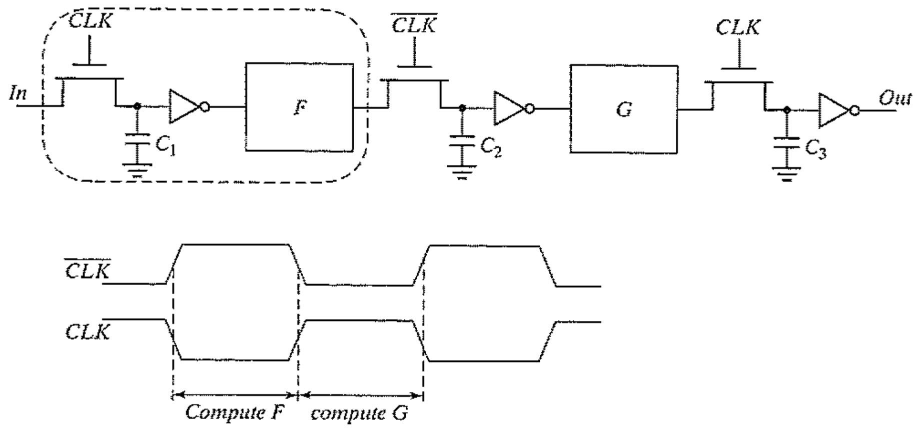


Figure 7-41 Operation of two-phase pipelined circuit, using dynamic registers.

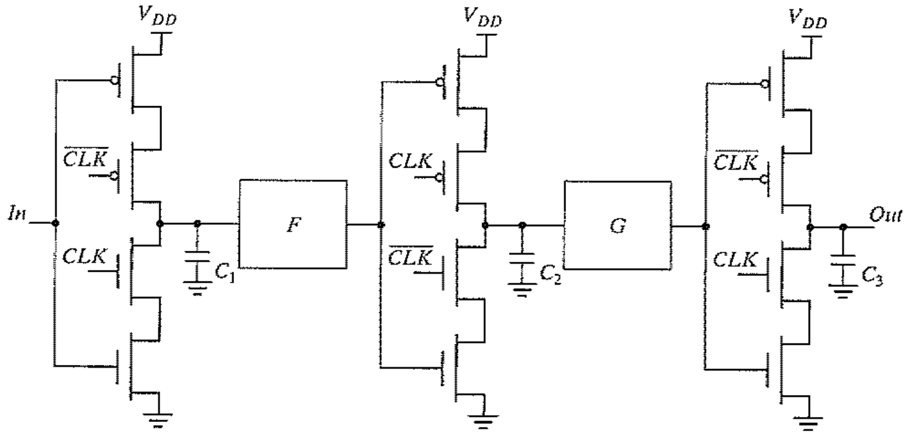


Figure 7-42 Pipelined datapath, using C^2MOS latches.

7.5.2 NORA–CMOS—A Logic Style for Pipelined Structures

The latch-based pipeline circuit can also be implemented by using C^2MOS latches, as shown in Figure 7-42. The operation is similar to the one discussed in Section 7.5.1. This topology has one additional important property:

A C^2MOS-based pipelined circuit is race free as long as all the logic functions *F* (implemented by using static logic) between the latches are noninverting.

The reasoning for the preceding argument is similar to the argument made in the construction of a C^2MOS register. During a (0–0) overlap between *CLK* and $\overline{CLK}$, all C^2MOS latches simplify to pure pull-up networks (see Figure 7-27). The only way a signal can race

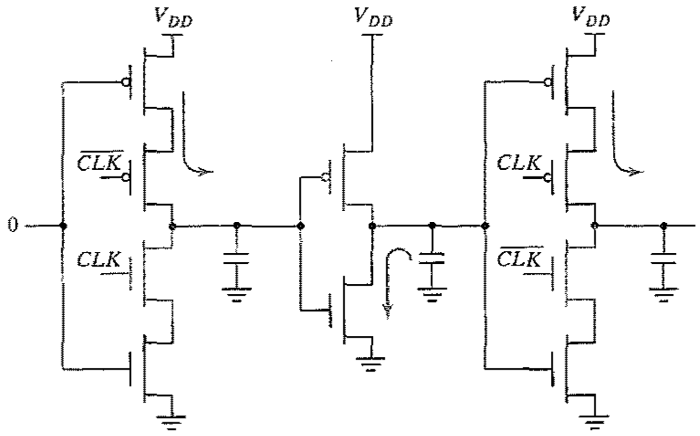


Figure 7-43 Potential race condition during (0–0) overlap in C^2MOS-based design.

from stage to stage under this condition is when the logic function F is inverting, as illustrated in Figure 7-43, where F is replaced by a single, static CMOS inverter. Similar considerations are valid for the (1–1) overlap.

Based on this concept, a logic circuit style called NORA–CMOS was conceived [Gonçalves83]. It combines C^2MOS pipeline registers and NORA dynamic logic function blocks. Each module consists of a block of combinational logic that can be a mixture of static and dynamic logic, followed by a C^2MOS latch. Logic and latch are clocked in such a way that both are simultaneously in either evaluation, or hold (precharge) mode. A block that is in evaluation during $CLK = 1$ is called a *CLK module*, while the inverse is called a $\overline{CLK}$ *module*. Examples of both classes are shown in Figure 7-44a and 7-44b, respectively. The operation modes of the modules are summarized in Table 7-2.

A NORA datapath consists of a chain of alternating CLK and $\overline{CLK}$ modules. While one class of modules is precharging with its output latch in hold mode, preserving the previous output value, the other class is evaluating. Data is passed in a pipelined fashion from module to module. NORA offers designers a wide range of design choices. Dynamic and static logic

Table 7-2 Operation modes for NORA logic modules.

	CLK block		$\overline{CLK}$ block	
	Logic	**Latch**	**Logic**	**Latch**
$CLK = 0$	Precharge	Hold	Evaluate	Evaluate
$CLK = 1$	Evaluate	Evaluate	Precharge	Hold

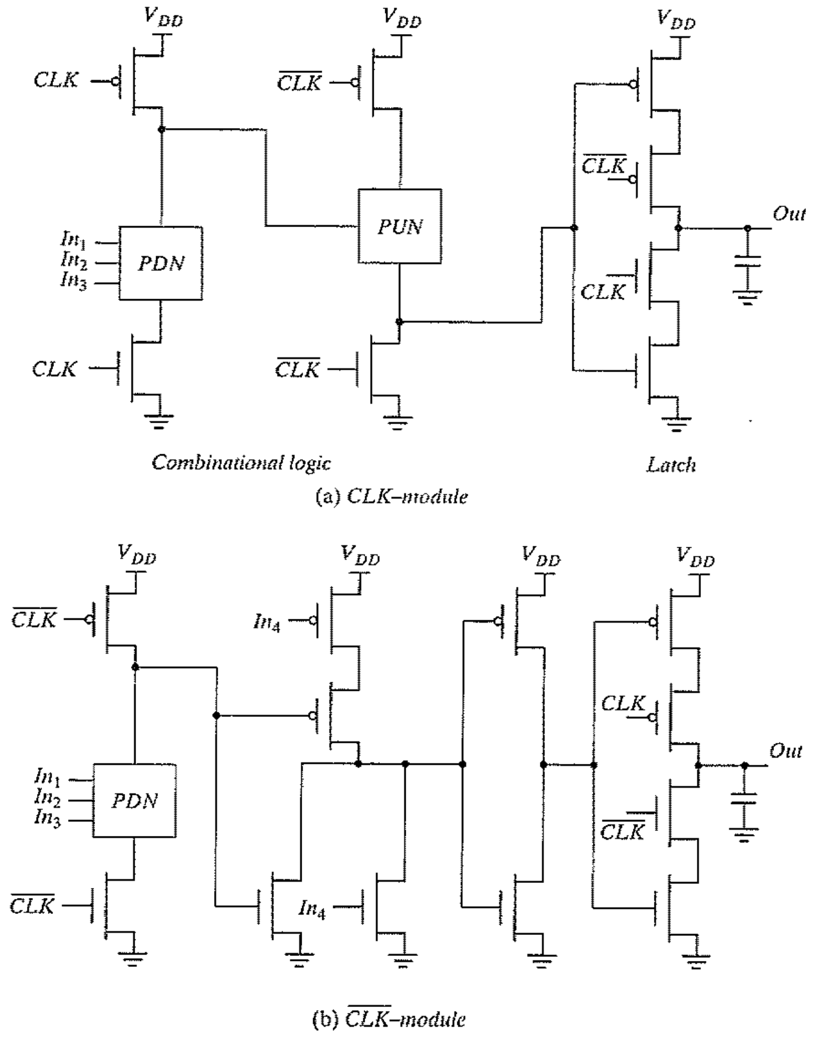


Figure 7-44 Examples of NORA–CMOS modules.

can be mixed freely, and both CLK_p and CLK_n dynamic blocks can be used in cascaded or in pipelined form. Although this style of logic avoids the extra inverter required in domino CMOS, there are many rules that must be followed to achieve reliable and race-free operation. As a result of this added complexity, the use of NORA has been limited to high-performance applications.

7.6 Nonbistable Sequential Circuits

In the preceding sections, we have focused on a single type of sequential element: the latch (and its sibling, the register). The most important property of such a circuit is that it has two stable states—hence, the term *bistable*. The bistable element is not the only sequential circuit of interest. Other regenerative circuits can be catalogued as *astable* and *monostable*. The former act as oscillators and can, for instance, be used for on-chip clock generation. The latter serve as pulse generators, also called *one-shot circuits*. Another interesting regenerative circuit is the *Schmitt trigger*. This component has the useful property of showing hysteresis in its dc characteristics—its switching threshold is variable and depends upon the direction of the transition (low to high or high to low). This peculiar feature can come in handy in noisy environments.

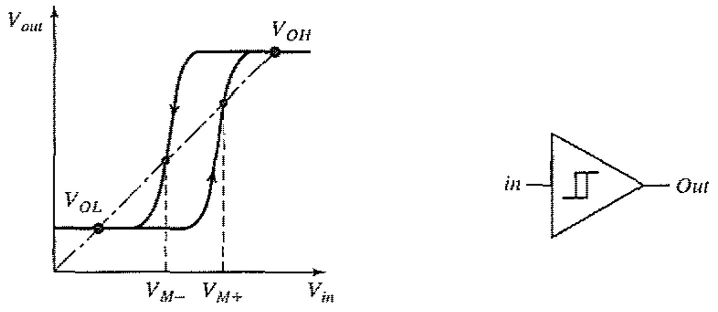
7.6.1 The Schmitt Trigger

Definition

A *Schmitt trigger* [Schmitt38] is a device with two important properties:

1. It responds to a slowly changing input waveform with a **fast transition time at the output.**
2. The voltage-transfer characteristic of the device displays *different switching thresholds for positive- and negative-going input signals.* This is demonstrated in Figure 7-45, where a typical voltage-transfer characteristic of the Schmitt trigger is shown (and its schematics symbol). The switching thresholds for the low-to-high and high-to-low transitions are called V_{M+} and V_{M-}, respectively. The *hysteresis voltage* is defined as the difference between the two.

One of the main uses of the Schmitt trigger is to turn a noisy or slowly varying input signal into a clean digital output signal. This is illustrated in Figure 7-46. Notice how the hysteresis suppresses the ringing on the signal. At the same time, the fast low-to-high (and high-to-low) transi-



(a) Voltage-transfer characteristic (b) Schematic symbol

Figure 7-45 Noninverting Schmitt trigger.

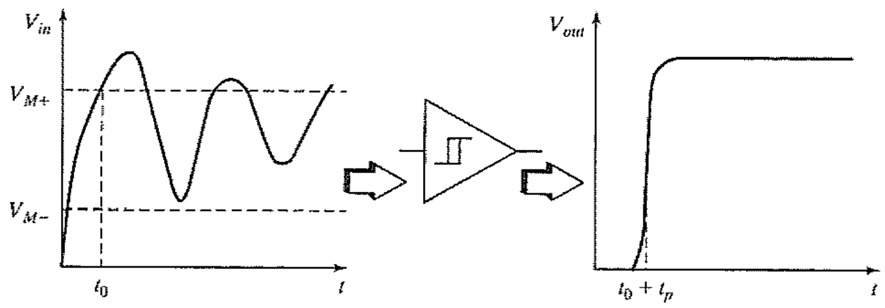


Figure 7-46 Noise suppression, using a Schmitt trigger.

tions of the output signal should be observed. Steep signal slopes are beneficial in general, for instance for reducing power consumption by suppressing direct-path currents. The "secret" behind the Schmitt trigger concept is the use of positive feedback.

CMOS Implementation

One possible CMOS implementation of the Schmitt trigger is shown in Figure 7-47. The idea behind this circuit is that the switching threshold of a CMOS inverter is determined by the (k_n/k_p) ratio between the PMOS and NMOS transistors. Increasing the ratio raises the threshold, while decreasing it lowers V_M. Adapting the ratio depending upon the direction of the transition results in a shift in the switching threshold and a hysteresis effect. This adaptation is achieved with the aid of feedback.

Suppose that V_{in} is initially equal to 0, so that $V_{out} = 0$ as well. The feedback loop biases the PMOS transistor M_4 in the conductive mode, while M_3 is off. The input signal effectively connects to an inverter consisting of two PMOS transistors in parallel (M_2 and M_4) as a pull-up network, and a single NMOS transistor (M_1) in the pull-down chain. This modifies the effective transistor ratio of the inverter to $k_{M1}/(k_{M2} + k_{M4})$, which moves the switching threshold upwards.

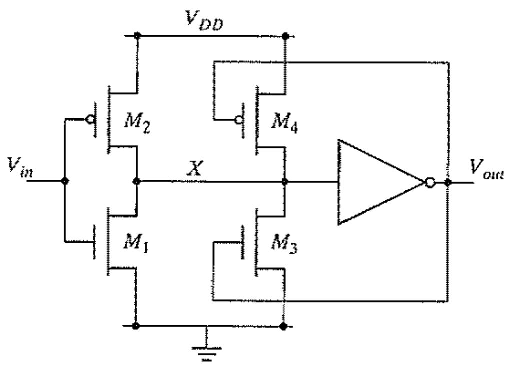


Figure 7-47 CMOS Schmitt trigger.

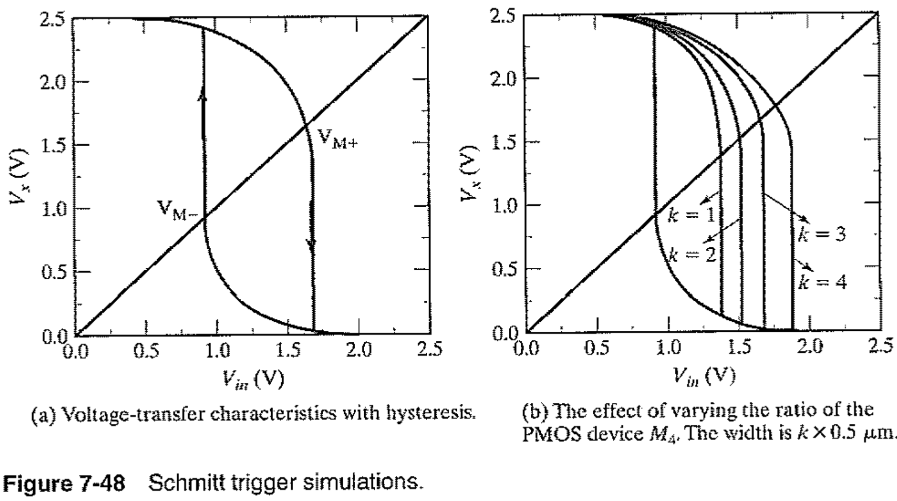
Once the inverter switches, the feedback loop turns off M_4, and the NMOS device M_3 is activated. This extra pull-down device speeds up the transition and produces a clean output signal with steep slopes.

A similar behavior can be observed for the high-to-low transition. In this case, the pull-down network originally consists of M_1 and M_3 in parallel, while the pull-up network is formed by M_2. This reduces the value of the switching threshold to V_{M-}.

Example 7.6 CMOS Schmitt Trigger

Consider the Schmitt trigger of Figure 7-47, with M_1 and M_2 sized at 1 μm/0.25 μm, and 3 μm/0.25 μm, respectively. The inverter is designed such that the switching threshold is around $V_{DD}/2$ (= 1.25 V). Figure 7-48a shows the simulation of the Schmitt trigger assuming that devices M_3 and M_4 are 0.5 μm/0.25 μm and 1.5 μm/0.25 μm, respectively. As apparent from the plot, the circuit exhibits hysteresis. The high-to-low switching point (V_{M-} = 0.9 V) is lower than $V_{DD}/2$, while the low-to-high switching threshold (V_{M+} = 1.6 V) is larger than $V_{DD}/2$.

It is possible to shift the switching point by changing the sizes of M_3 and M_4. For example, to modify the low-to-high transition, we need to vary the PMOS device. The high-to-low threshold is kept constant by keeping the device width of M_3 at 0.5 μm. The device width of M_4 is varied as $k \times 0.5$ μm. Figure 7-48b demonstrates how the switching threshold increases with raising values of k.



(a) Voltage-transfer characteristics with hysteresis.

(b) The effect of varying the ratio of the PMOS device M_4. The width is $k \times 0.5$ μm.

Figure 7-48 Schmitt trigger simulations.

Problem 7.7 An Alternative CMOS Schmitt Trigger

Another CMOS Schmitt trigger is shown in Figure 7-49. Discuss the operation of the gate, and derive expressions for V_{M-} and V_{M+}.

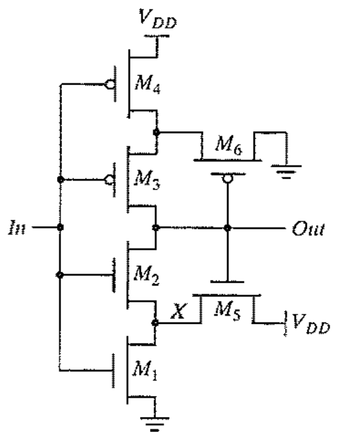


Figure 7-49 Alternate CMOS Schmitt trigger.

7.6.2 Monostable Sequential Circuits

A monostable element is a circuit that generates a pulse of a predetermined width every time the quiescent circuit is triggered by a pulse or transition event. It is called *monostable* because it has only one stable state (the quiescent one). A trigger event, which is either a signal transition or a pulse, causes the circuit to go temporarily into another quasi-stable state. This means that it eventually returns to its original state after a time period determined by the circuit parameters. This circuit, also called a *one-shot*, is useful in generating pulses of a known length. This functionality is required in a wide range of applications. We have already seen the use of a one-shot in the construction of glitch registers. Another well-known example is the *address transition detection* (ATD) circuit, used for the timing generation in static memories. This circuit detects a change in a signal or group of signals, such as the address or data bus, and produces a pulse to initialize the subsequent circuitry.

The most common approach to the implementation of one-shots is the use of a simple delay element to control the duration of the pulse. The concept is illustrated in Figure 7-50. In the quiescent state, both inputs to the XOR are identical, and the output is low. A transition on the input causes the XOR inputs to differ temporarily and the output to go high. After a delay t_d (of the delay element), this disruption is removed, and the output goes low again. A pulse of length t_d is created. The delay circuit can be realized in many different ways, such as an *RC*-network or a chain of basic gates.

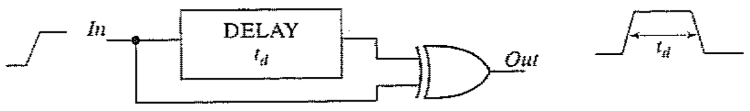


Figure 7-50 Transition-triggered one shot.

7.6.3 Astable Circuits

An astable circuit has no stable states. The output oscillates back and forth between two quasi-stable states, with a period determined by the circuit topology and parameters (delay, power supply, etc.). One of the main applications of oscillators is the on-chip generation of clock signals. (This application is discussed in detail in a later chapter on timing.)

The ring oscillator is a simple example of an astable circuit. It consists of an odd number of inverters connected in a circular chain. Due to the odd number of inversions, no stable operation point exists, and the circuit oscillates with a period equal to $2 \times t_p \times N$, where N is the number of inverters in the chain and t_p is the propagation delay of each inverter.

Example 7.7 Ring Oscillator

The simulated response of a ring oscillator with five stages is shown in Figure 7-51 (all gates use minimum-size devices). The observed oscillation period approximately equals 0.5 ns, which corresponds to a gate propagation delay of 50 ps. By tapping the chain at various points, different phases of the oscillating waveform are obtained. (Phases 1, 3, and 5 are displayed in the plot.) A wide range of clock signals with different duty-cycles and phases can be derived from those elementary signals, using simple logic operations.

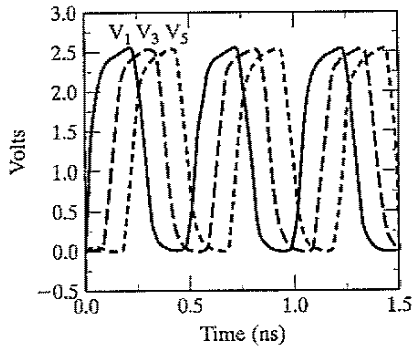


Figure 7-51 Simulated waveforms of five-stage ring oscillator. The outputs of stages 1, 3, and 5 are shown.

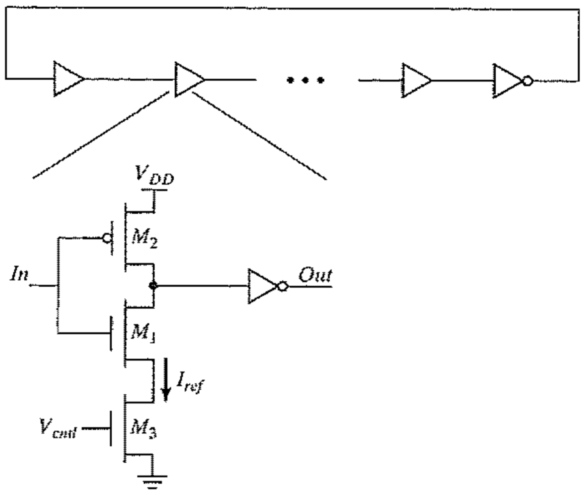


Figure 7-52 Voltage-controlled oscillator based on current-starved inverters.

The ring oscillator composed of cascaded inverters produces a waveform with a fixed oscillating frequency determined by the delay of an inverter in the CMOS process. In many applications, it is necessary to control the frequency of the oscillator. An example of such a circuit is the *voltage-controlled oscillator (VCO)*, whose oscillation frequency is a function (typically, nonlinear) of a control voltage. The standard ring oscillator can be modified into a *VCO* by replacing the standard inverter with a *current-starved inverter* like the one shown in Figure 7-52 [Jeong87]. The mechanism for controlling the delay of each inverter is to limit the current available to discharge the load capacitance of the gate.

In this modified inverter circuit, the maximal discharge current of the inverter is limited by adding an extra series device. Note that the low-to-high transition on the inverter can also be controlled by adding a PMOS device in series with M_2. The added NMOS transistor M_3, is controlled by an analog control voltage V_{cntl}, which determines the available discharge current. Lowering V_{cntl} reduces the discharge current and, hence, increases t_{pHL}. The ability to alter the propagation delay per stage allows us to control the frequency of the ring structure. The control voltage is generally set by using feedback techniques. Under low-operating current levels, the current-starved inverter suffers from slow fall times at its output. This can result in significant short-circuit current. We solve this problem by feeding its output into a CMOS inverter or, better yet, a Schmitt trigger. An extra inverter is needed at the end to ensure that the structure oscillates.

Example 7.8 Current-Starved Inverter Simulation

Figure 7-53 shows the simulated delay of the current-starved inverter as a function of the control voltage V_{cntl}. The delay of the inverter can be varied over a large range. When the control voltage is smaller than the threshold, the device enters the subthreshold region.

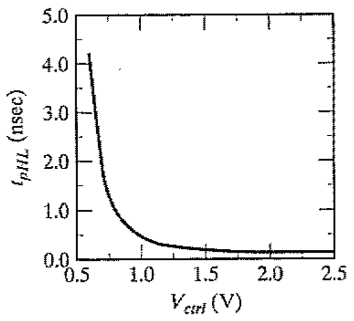


Figure 7-53 t_{pHL} of current-starved inverter as a function of the control voltage.

This results in large variations of the propagation delay, as the drive current is exponentially dependent on the drive voltage. When operating in this region, the delay is very sensitive to variations in the control voltage and hence to noise.

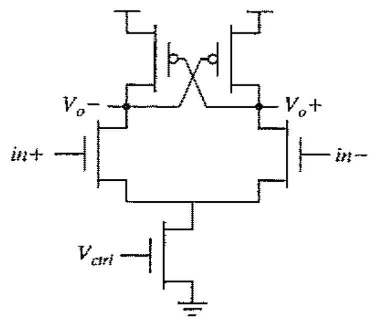
Another approach to implement the delay cell is to use a differential element as shown in Figure 7-54a. Since the delay cell provides both inverting and noninverting outputs, an oscillator with an even number of stages can be implemented. Figure 7-54b shows a two-stage differential *VCO*, where the feedback loop provides 180° phase shift through two gate delays, one noninverting and the other inverting, therefore forming an oscillation. The simulated waveforms of this two-stage *VCO* are shown in Figure 7-54c. The in-phase and quadrature phase outputs are available simultaneously. The differential-type *VCO* has better immunity to common mode noise (for example, supply noise) compared with the common ring oscillator. However, it consumes more power due to its increased complexity and its static current.

7.7 Perspective: Choosing a Clocking Strategy

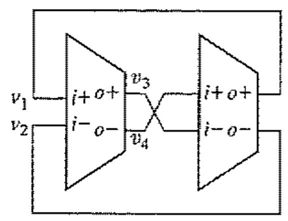
A crucial decision that must be made in the earliest phases of chip design is to select the appropriate clocking methodology. The reliable synchronization of the various operations occurring in a complex circuit is one of the most intriguing challenges facing the digital designer of the next decade. Choosing the right clocking scheme affects the functionality, speed, and power of a circuit.

A number of widely used clocking schemes were introduced in this chapter. The most robust and conceptually simple scheme is the two-phase master–slave design. The predominant approach is to use the multiplexer-based register, and to generate the two clock phases locally by simply inverting the clock. More exotic schemes such as the glitch register are also used in practice. However, these schemes require significant fine-tuning and must only be used in specific situations. An example of such is the need for a negative setup time to cope with clock skew.

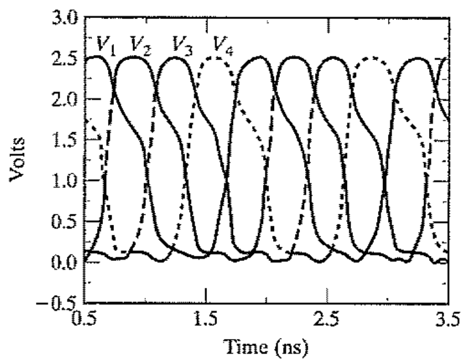
The general trend in high-performance CMOS VLSI design is therefore to **use simple clocking schemes**, even at the expense of performance. Most automated design methodologies



(a) Delay cell



(b) Two-stage *VCO*



(c) Simulated waveforms of two-stage *VCO*

Figure 7-54 Differential delay element and *VCO* topology.

such as standard cell employ a single-phase, edge-triggered approach, based on static flip-flops. Nevertheless, the tendency towards simpler clocking approaches also is apparent in high-performance designs such as microprocessors. The use of latches between logic to improve circuit performance is common as well.

7.8 Summary

This chapter has explored the subject of sequential digital circuits. The following topics were discussed:

- The cross coupling of two inverters creates a *bistable* circuit, also known as a *flip-flop*. A third potential operation point turns out to be metastable; that is, any diversion from this bias point causes the flip-flop to converge to one of the stable states.
- A latch is a *level-sensitive* memory element that samples data on one phase and holds data on the other phase. A register, on the other hand, samples the data on the rising or falling edge. A register has three important parameters: *the setup time, the hold time, and the*

propagation delay. These parameters must be carefully optimized, because they may account for a significant portion of the clock period.

- Registers can be *static* or *dynamic*. A static register holds state as long as the power supply is turned on. It is ideal for memory that is accessed infrequently (e.g., reconfiguration registers or control information). Static registers use either multiplexers or overpowering to enable the writing of data.
- Dynamic memory is based on temporary charge storage on capacitors. The primary advantage is reduced complexity, higher performance, and lower power consumption. However, charge on a dynamic node leaks away with time, and dynamic circuits thus have a minimum clock frequency. Pure dynamic memory is hardly used anymore. Register circuits are made pseudostatic to provide immunity against capacitive coupling and other sources of circuit induced noise.
- Registers can also be constructed by using the *pulse or glitch concept*. An intentional pulse (using a one-shot circuit) is used to sample the input around an edge. Sense-amplifier-based schemes also are used to construct registers; they should be used as well when high-performance or low-signal-swing signalling is required.
- Choice of *clocking style* is an important consideration. Two-phase design can result in race problems. Circuit techniques such as C^2MOS can be used to eliminate race conditions in two-phase clocking. Another option is to use true single-phase clocking. However, the rise time of clocks must be carefully optimized to eliminate races.
- The combination of dynamic logic with dynamic latches can produce extremely fast computational structures. An example of such an approach, the NORA logic style, is very effective in *pipelined datapaths*.
- *Monostable structures* have only one stable state; thus, they are useful as pulse generators.
- *Astable multivibrators*, or oscillators, possess no stable state. The ring oscillator is the best-known example of a circuit of this class.
- *Schmitt triggers* display hysteresis in their dc characteristic and fast transitions in their transient response. They are mainly used to suppress noise.

7.9 To Probe Further

The basic concepts of sequential gates can be found in many logic design textbooks (e.g., [Mano82] and [Hill74]). The design of sequential circuits is amply documented in most of the traditional digital circuit handbooks. [Partovi01] and [Bernstein98] provide in-depth overviews of the issues and solutions in the design of high-performance sequential elements.

References

[Bernstein98] K. Bernstein et al., *High-Speed CMOS Design Styles*, Kluwer Academic Publishers, 1998.

[Dopperpuhl92] D. Dopperpuhl et al., "A 200 MHz 64-b Dual Issue CMOS Microprocessor," *IEEE Journal of Solid-State Circuits*, vol. 27, no. 11, Nov. 1992, pp. 1555–1567.

[Gieseke97] B. Gieseke et al., "A 600 MHz Superscalar RISC Microprocessor with Out-of-Order Execution," *IEEE International Solid-State Circuits Conference*, pp. 176–177, Feb. 1997.

[Gonçalves83] N. Gonçalves and H. De Man, "NORA: a racefree dynamic CMOS technique for pipelined logic structures," *IEEE Journal of Solid-State Circuits*, vol. SC-18, no. 3, June 1983, pp. 261–266.

[Hill74] F. Hill and G. Peterson, *Introduction to Switching Theory and Logical Design*, Wiley, 1974.

[Jeong87] D. Jeong et al., "Design of PLL-based clock generation circuits," *IEEE Journal of Solid-State Circuits*, vol. SC-22, no. 2, April 1987, pp. 255–261.

[Kozu96] S. Kozu et al., "A 100 MHz 0.4 W RISC Processor with 200 MHz Multiply-Adder, using Pulse-Register Technique," *IEEE ISSCC*, pp. 140–141, February 1996.

[Mano82] M. Mano, *Computer System Architecture*, Prentice-Hall, 1982.

[Montanaro96] J. Montanaro et al., "A 160-MHz, 32-b, 0.5-W CMOS RISC Microprocessor," *IEEE Journal of Solid-State Circuits*, pp. 1703–1714, November 1996.

[Mutoh95] S. Mutoh et al., "1-V Power Supply High-Speed Digital Circuit Technology with Multithreshold-Voltage CMOS," *IEEE Journal of Solid State Circuits*, pp. 847–854, August 1995.

[Partovi96] H. Partovi, "Flow-Through Latch and *Edge-Triggered* Flip-Flop Hybrid Elements," *IEEE ISSCC*, pp. 138–139, February 1996.

[Partovi01] H. Partovi, "Clocked Storage Elements," in *Design of High-Performance Microprocessor Circuits*, Chandakasan et al., ed., Chapter 11, pp. 207–233, 2001.

[Schmitt38] O. H. Schmitt, "A Thermionic Trigger," *Journal of Scientific Instruments*, vol. 15, January 1938, pp. 24–26.

[Suzuki73] Y. Suzuki, K. Odagawa, and T. Abe, "Clocked CMOS calculator circuitry," *IEEE Journal of Solid State Circuits*, vol. SC-8, December 1973, pp. 462–469.

[Yuan89] J. Yuan and Svensson C., "High-Speed CMOS Circuit Technique," *IEEE JSSC*, vol. 24, no. 1, February 1989, pp. 62–70.

PART

3

A System Perspective

"Art, it seems to me, should simplify. That, indeed, is very nearly the whole of the higher artistic process; finding what conventions of form and what of detail one can do without and yet preserve the spirit of the whole."

Willa Sibert Cather,
On the Art of Fiction (1920).

"Simplicity and repose are the qualities that measure the true value of any work of art"

Frank Lloyd Wright.

CHAPTER

8

Implementation Strategies for Digital ICs

Semicustom and structured design methodologies

ASIC and system-on-a-chip design flows

Configurable hardware

8.1 Introduction

The dramatic increase in complexity of contemporary integrated circuits poses an enormous design challenge. Designing a multimillion-transistor circuit and ensuring that it operates correctly when the first silicon returns is a daunting task that is virtually impossible without the help of computer aids and well-established design methodologies. In fact, it has often been suggested that technology advancements might be outpacing the absorption bandwidth of the design community. This is articulated in Figure 8-1, which shows how IC complexity (in logic transistors) is growing faster than the productivity of a design engineer, creating a "design gap." One way to address this gap is to increase steadily the size of the design teams working on a single project. We observe this trend in the high-performance processor world, where teams of more than 500 people are no longer a surprise.

Obviously, this approach cannot be sustained in the long term—just imagine all the design engineers in the world working on a single design. Fortunately, about once in a decade we witness the introduction of a novel design methodology that creates a step function in design productivity, helping to bridge the gap temporarily. Looking back over the past four decades, we can identify a number of these productivity leaps. Pure custom design was the norm in the early integrated circuits of the 1970s. Since then, programmable logic arrays (PLAs), standard cells, macrocells, module compilers, gate arrays, and reconfigurable hardware have steadily helped to ease the time and cost of mapping a function onto silicon. In this chapter, we provide a description of some commonly used design implementation approaches. Due to the extensive nature of the field, we cannot be comprehensive—doing so would require a textbook of its own. Instead, we present *a user perspective* that provides a basic perception and insight into what is offered and can be expected from the different design methodologies.

The preferred approach to mapping a function onto silicon depends largely upon the function itself. Consider, for instance, the simple digital processor of Figure 8-2. Such a processor

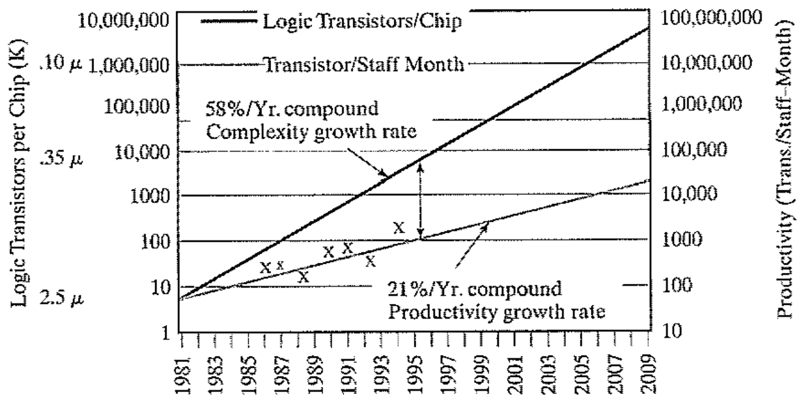


Figure 8-1 The design productivity gap. Technology (in logic transistors/chip) outpaces the design productivity (in transistors designed by a single design engineer per month). Source: SIA [SIA97].

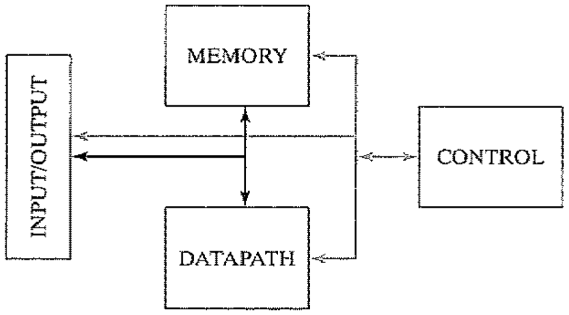


Figure 8-2 Composition of a generic digital processor. The arrows represent the possible interconnections.

could be the brain of a personal computer (PC), or the heart of a compact-disc player or cellular phone. It is composed of a number of building blocks that occur in one form or another in almost every digital processor:

- **The datapath** is the core of the processor; it is where all computations are performed. The other blocks in the processor are support units that either store the results produced by the datapath or help to determine what will happen in the next cycle. A typical datapath consists of an interconnection of basic combinational functions, such as logic (AND, OR, EXOR) or arithmetic operators (addition, multiplication, comparison, shift). Intermediate results are stored in registers. Different strategies exist for the implementation of datapaths—structured custom cells versus automated standard cells, or fixed hard-wired versus flexible field-programmable fabric. The choice of the implementation platform is mostly influenced by the trade-off between different design metrics such as area, speed, energy, design time, and reusability.
- **The control module** determines what actions happen in the processor at any given point in time. A controller can be viewed as a finite state machine (FSM). It consists of registers and logic, and thus is a sequential circuit. The logic can be implemented in different ways—either as an interconnection of basic logic gates (standard cells), or in a more structured fashion using programmable logic arrays (PLAs) and instruction memories.
- **The memory module** serves as the centralized data storage area. A broad range of different memory classes exist. The main difference between those classes is in the way data can be accessed, such as "read only" versus "read–write," sequential versus random access, or single-ported versus multiported access. Another way of differentiating between memories is related to their data-retention capabilities. Dynamic memory structures must be refreshed periodically to keep their data, while static memories keep their data as long as the power source is turned on. Finally, nonvolatile memories such as flash memories conserve the stored data even when the supply voltage is removed. A single processor might combine different memory classes. For example, random access memory can be used to store data, and read-only memory may store instructions.

- **The interconnect** network joins the different processor modules to one another, while the **input/output circuitry** connects to the outside world. For a long time, interconnections were an afterthought in the design process. Unfortunately, the wires composing the interconnect network are less than ideal and present a capacitive, resistive, and inductive load to the driving circuitry. As die sizes grow larger, the length of the interconnect wires also tends to grow, resulting in increasing values for these parasitics. Today, automated or structured design methodologies are being introduced that ease the deployment of these interconnect structures. Examples include *on-chip busses*, *mesh interconnect* structures, and even complete *networks on a chip*. Some components of the interconnect network typically are abstracted away on schematic block diagrams, such as the one shown in Figure 8-2, yet are of critical importance to the well-being of the design. These include the power- and clock-distribution networks. Early planning of these "service" networks can go a long way toward ensuring the correct operation of the integrated circuit.

The structure of Figure 8-2 may be repeated many times on a single die. Figure 8-3 shows an example of a *system on a chip*, which combines all the functions needed for the realization of a complete high-definition digital TV set. It combines two processors, memory units, specialized accelerators for functions such as MPEG (de)coding and data filtering, as well as a range of

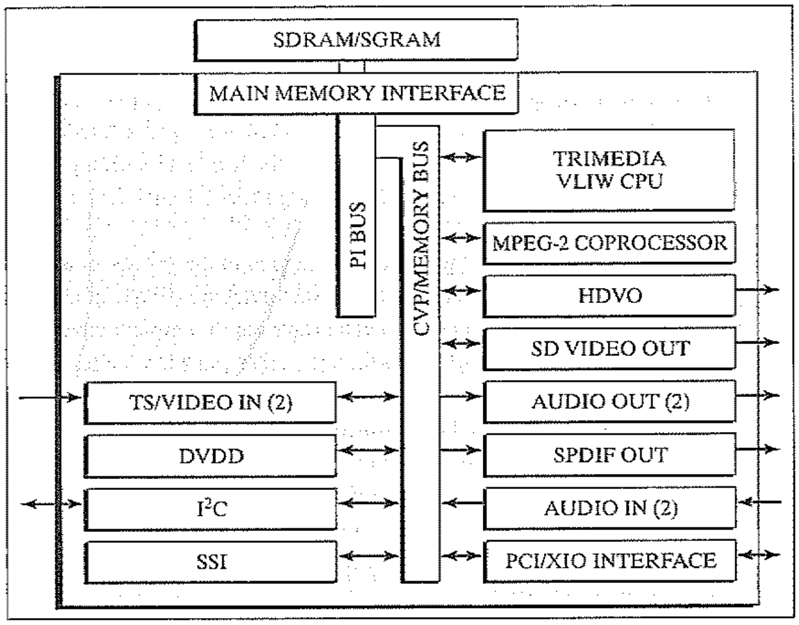


Figure 8-3 The "Nexperia" system on a chip [Philips99]. This single chip combines a general-purpose microprocessor core, a VLIW (very large instruction word) signal processor, a memory system, an MPEG coprocessor, multiple accelerator units, and input/output peripherals, as well as two system busses.

peripheral units. Other applications such as wireless transceivers or hard-disk read/write units may even include some sizable analog modules.

Choosing an effective implementation approach strongly depends upon the function of the modules under consideration. For example, memory units tend to be very regular and structured. A module compiler that stacks cells in an arraylike fashion is thus the preferred implementation approach. Controllers, on the other hand, tend to be unstructured, and other implementation approaches are desirable. The choice of the implementation strategy can have a tremendous effect on the quality of the final product. The challenge for the designer is to pick the style that meets the product specifications and constraints. What works well for one design may well be a disaster for another one.

Example 8.1 Trading Off Energy Efficiency and Flexibility

A design that embraces flexibility (or programmability) is very attractive from an application perspective. It allows for "late binding," in which the application can still be changed after the chip has gone to fabrication. Flexibility makes it possible to reuse a single design for multiple applications, or to upgrade the firmware of a component in the field, reducing the risk for the manufacturer. In contrast, a hard-wired component is totally fixed at manufacturing time and cannot be modified afterwards.

So, why not use flexible or programmable components for every possible design? As always, there is no free lunch. Flexibility comes at a price in both performance and energy efficiency. Providing programmability means adding overhead to implementation. For example, a programmable processor uses stored instructions and an instruction decoder to make a single datapath perform multiple functions. Most designers are not aware of the large cost of flexibility. The impact is illustrated in Figure 8-4, which compares the *energy*

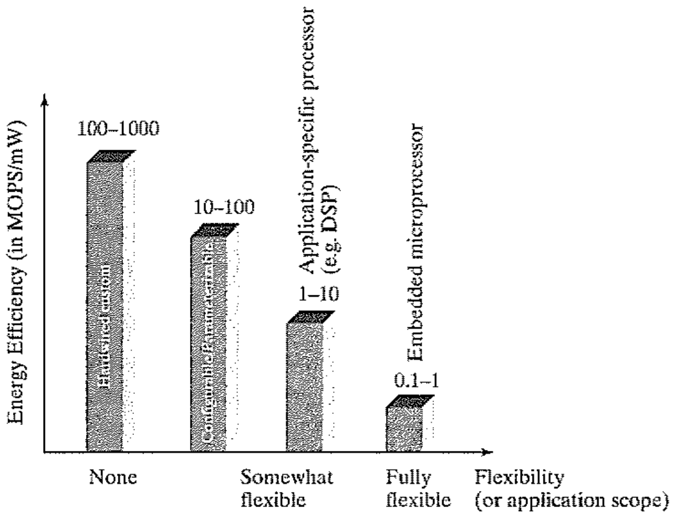


Figure 8-4 Trading off flexibility versus energy efficiency (in MOPS/mW or millions of operations per mJ of energy) for different implementation styles. The numbers were collected for a 0.25 μm CMOS process [Rabaey00].

efficiency—the number of operations that can be performed for a given amount of energy—of various implementation styles versus their *flexibility*—that is, the range of applications that can be mapped onto them. A staggering **three orders of magnitude** in variation can be observed. This clearly demonstrates that hard-wired or implementation styles with limited flexibility (such as configurable or parameterizable modules) are preferable when energy efficiency is a must.

In this and the following three chapters, we discuss, respectively, implementation techniques for random logic and controllers (this chapter), interconnect (Chapter 9), datapaths (Chapter 11), and memories (Chapter 12). Observe that the choice of the implementation approach can have a tremendous effect on the quality of the final product. Important aspects in the design of complex systems consisting of multiple blocks and thus deserving special attention are synchronization and timing (Chapter 10) and the power distribution network (Chapter 9). The distribution of clock signals and supply current has become one of the dominant problems in the design of state-of-the-art processors. A number of Design Methodology Inserts, interspersed between the chapters, address the design challenge posed by these complex components, and introduce the advanced design automation tools that are available to the designer. Inserts F, G, and H discuss design synthesis, verification, and test, respectively.

8.2 From Custom to Semicustom and Structured-Array Design Approaches

The viability of a microelectronics design depends on a number of (often) conflicting factors, such as performance in terms of speed or power consumption, cost, and production volume. For example, to be competitive in the market, a microprocessor has to excel in performance at a low cost to the customer. Achieving both goals simultaneously is only possible through large sales volumes. The high development cost associated with high-performance design is then amortized over many parts. Applications such as supercomputing and some defense applications present another scenario. With ultimate performance as the primary design goal, high-performance custom design techniques often are desirable. The production volume is small, but the cost of electronic parts is only a fraction of the overall system costs and thus not much of an issue. Finally, reducing the system size through integration, not performance, is the major objective in most consumer applications. Under these circumstances, the design cost can be reduced substantially by using advanced design-automation techniques, which compromise performance, but minimize design time. As noted in Chapter 1, the cost of a semiconductor device is the sum of two components:

- The *nonrecurring expense* (NRE), which is incurred only once for a design and includes the cost of designing the part.
- The *production cost per part*, which is a function of the process complexity, design area, and process yield.

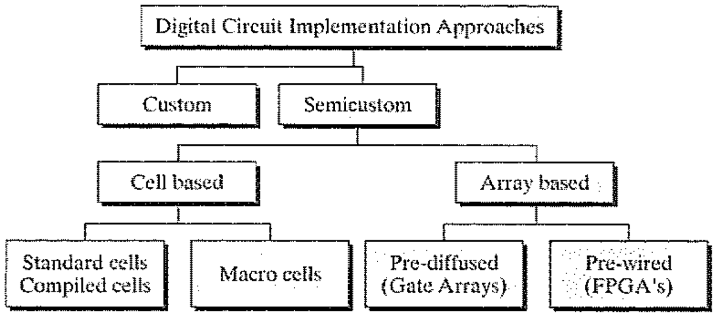

Digital Circuit Implementation Approaches
Custom
Semicustom
Cell based
Array based
Standard cells Compiled cells
Macro cells
Pre-diffused (Gate Arrays)
Pre-wired (FPGA's)

Figure 8-5 Overview of implementation approaches for digital integrated circuits (after [DeMicheli94]).

These economic considerations have spurred the development of a number of distinct implementation approaches ranging from high-performance, handcrafted design to fully programmable, medium-to-low performance designs. Figure 8-5 provides an overview of the different methodologies. In the sections that follow, we discuss first the custom design methodology, followed by the semicustom and array-based approaches.

8.3 Custom Circuit Design

When performance or design density is of primary importance, handcrafting the circuit topology and physical design seems to be the only option. Indeed, this approach was the only option in the early days of digital microelectronics, as is adequately demonstrated in the design of the Intel 4004 microprocessor (see Figure 8-5a). The labor-intensive nature of custom design translates into a high cost and a long *time to market*. Therefore, it can only be justified economically under the following conditions:

- The custom block can be reused many times (for example, as a library cell).
- The cost can be amortized over a large volume. Microprocessors and semiconductor memories are examples of applications in this class.
- Cost is not the prime design criterion, as it is in supercomputers or hypersupercomputers.

With continuous progress in the design-automation arena, the share of custom design reduces from year to year. Even in the most advanced high-performance microprocessors, such as the Intel Pentium® 4 processor (see Figure 8-6), virtually all portions are designed automatically using semicustom design approaches. Only the most performance-critical modules such as the phase locked-loops and the clock buffers are designed manually. In fact, library cell design is the only area where custom design still thrives today.

The amount of design automation in the custom-design process is minimal, yet some design tools have proven indispensable. In concert with a wide range of verification, simulation, extraction and modeling tools, layout editors, design-rule and electrical-rule checkers—as

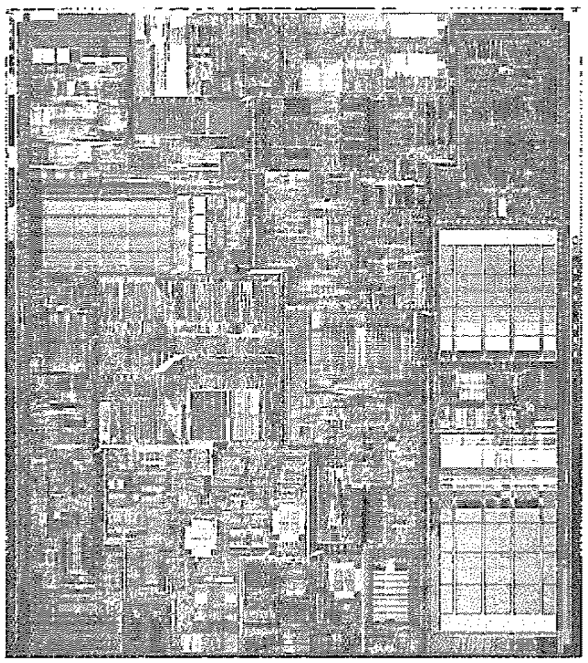


Figure 8-6 Chip microphotograph of Intel Pentium® 4 processor. It contains 42 million transistors, designed in a 0.18-µm CMOS technology. Its first generation runs at a clock speed of 1.5 GHz (Courtesy Intel Corp.).

described earlier in Design Methodology Insert A—are at the core of every custom-design environment. A excellent discussion of the opportunities and challenges of custom design can be found in [Grundman97].

8.4 Cell-Based Design Methodology

Since the custom-design approach proves to be prohibitively expensive, a wide variety of design approaches have been introduced over the years to shorten and automate the design process. This automation comes at the price of reduced integration density and/or performance. The following rule tends to hold: **the shorter the design time, the larger is the penalty incurred.** In this section, we discuss a number of design approaches that still require a full run through the manufacturing process for every new design. The *array-based design* approach discussed in the next section cuts the design time and cost even further by requiring only a limited set of extra processing steps or by eliminating processing completely.

The idea behind cell-based design is to reduce the implementation effort by *reusing* a limited library of cells. The advantage of this approach is that the cells only need to be designed and verified once for a given technology, and they can be reused many times, thus amortizing the design cost. The disadvantage is that the constrained nature of the library reduces the possibility of fine-tuning the design. Cell-based approaches can be partitioned into a number of classes depending on the granularity of the library elements.

8.4.1 Standard Cell

The standard-cell approach standardizes the design entry level at the logic gate. A library containing a wide selection of logic gates over a range of fan-in and fan-out counts is provided. Besides the basic logic functions, such as inverter, AND/NAND, OR/NOR, XOR/XNOR, and flip-flops, a typical library also contains more complex functions, such as AND-OR-INVERT, MUX, full adder, comparator, counter, decoders, and encoders. A design is captured as a schematic containing only cells available in the library, or is generated automatically from a higher level description language. The layout is then automatically generated. This high degree of automation is made possible by placing strong restrictions on the layout options. In the standard-cell philosophy, cells are placed in rows that are separated by routing channels, as illustrated in Figure 8-7. To be effective, this requires that all cells in the library have identical heights. The width of the cell can vary to accommodate for the variation in complexity between the cells. As illustrated in the drawing, the standard-cell technique can be intermixed with other layout approaches to allow for the introduction of modules such as memories and multipliers that do not adapt easily or efficiently to the logic-cell paradigm.

An example of a design implemented in an early standard-cell design style is shown in Figure 8-8a. A substantial fraction of the area is devoted to signal routing. The minimization of the interconnect overhead is the most important goal of the standard-cell placement and routing tools. One approach to minimizing the wire length is to introduce feed-through cells (Figure 8-7) that make it possible to connect between cells in different rows without having to route around a complete row. A far more important reduction in wiring overhead is obtained by adding more

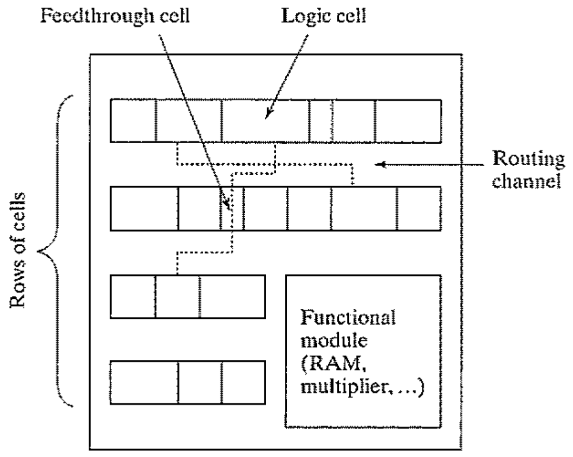


Figure 8-7 Standard-cell layout methodology.

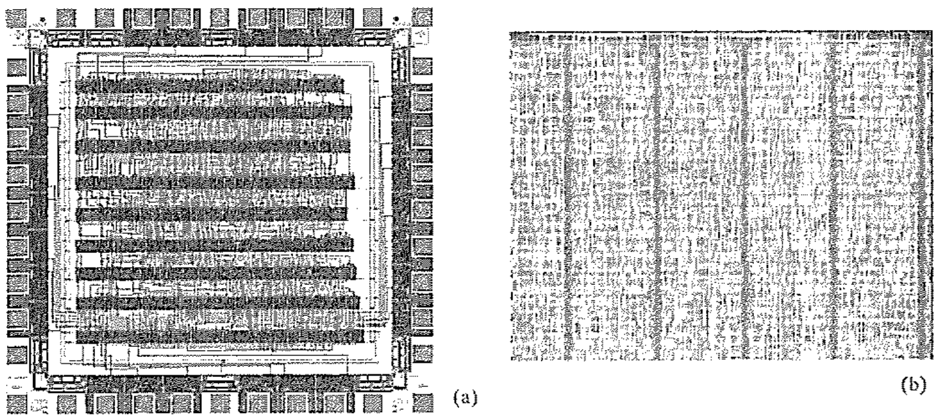


Figure 8-8 The evolution of standard-cell design. (a) Design in a three-layer metal technology. Wiring channels represent a substantial amount of the chip area. (b) Design in a seven-layer metal technology. Routing channels have virtually disappeared, and all interconnection is laid on top of the logic cells.

interconnect layers. The seven or more metal layers that are available in contemporary CMOS processes make it possible to all but eliminate the need for routing channels. Virtually all signals can be routed on top of the cells, creating a truly three-dimensional design. Figure 8-8b shows a fraction of a standard-cell design, implemented by using seven metal layers. The design achieves more than 90% density, which means that virtually all of the chip area is covered by logic cells, and that only a limited amount of the area is wasted for interconnect.

The design of a standard-cell library is a time-intensive undertaking that, fortunately, can be amortized over a large number of designs. Determining the composition of the library is a nontrivial task. A pertinent question is, Are we better off with a small library in which most cells have a limited fan-in, or is it more beneficial to have a large library with many versions of every gate (e.g., containing two-, three-, and four-input NAND gates, and different sizes for each of these gates)? Since the fan-out and load capacitance due to wiring are not known in advance, it used to be common practice to ensure that each gate had large current-driving capabilities, (i.e., employs large output transistors). While this simplifies the design procedure, it has a detrimental effect on area and power consumption. Today's libraries employ many versions of each cell, sized for different driving strengths, as well as performance and power consumption levels. It is left to the synthesis tool to select the correct cells, given speed and area requirements.

To make the library-based approach work, a detailed documentation of the cell library is an absolute necessity. The information should not only contain the layout, a description of functionality and terminal positioning, but it also must accurately characterize the delay and power consumption of the cell as a function of load capacitance and the input rise and fall times. Generating this information accounts for a large portion of the library generation effort. How to characterize logic and sequential cells is the topic of "Design Methodology Insert E."

Example 8.2 A Three-Input NAND-Gate Cell

To illustrate some of the preceding observations, the design of a three-input NAND standard-cell gate, implemented in a 0.18 μm CMOS technology, is depicted in Figure 8-9. The library actually contains five versions of the cell, supporting capacitive loads from 0.18 pF up to 0.72 pF and ranging in area from 16.4 μm^2 to 32.8 μm^2. The cell shown represents the low-performance, energy-efficient design corner, and uses high-threshold transistors to reduce leakage. The NMOS and PMOS transistors in the pull-down (-up) networks are both sized at a (W/L) ratio of approximately 8.

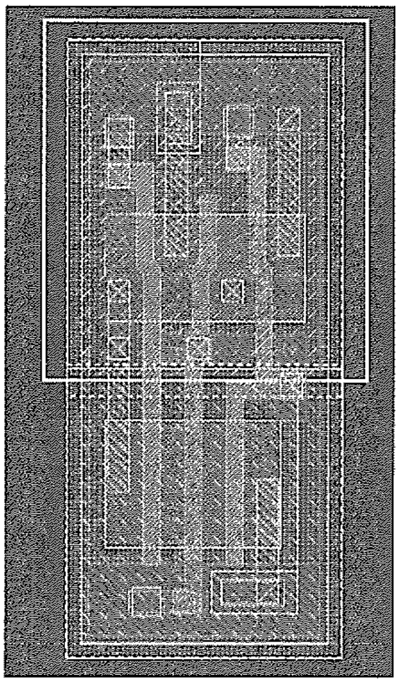


Figure 8-9 Three-input NAND standard cell (Courtesy ST Microelectronics).

Observe how the layout strategy follows the approach outlined in Figure D-2. Supply lines are distributed horizontally and shared between cells in the same row. Input signals are wired vertically using polysilicon. The input/output terminals are located throughout the cell body (as exemplified by the *pin* terminal in the layout drawing), in line with the over-the-cell wiring approach of today's standard-cell methodology.

The standard-cell approach has become immensely popular, and is used for the implementation of virtually all logic elements in today's integrated circuits. The only exceptions are when extreme high performance or low energy consumption is needed, or when the structure of the targeted function is very regular (such as a memory or a multiplier). The success of the standard-cell approach can be attributed to a number of developments, including the following:

- The **increased quality of the automatic cell placement and routing tools** in conjunction with the availability of multiple routing layers. In fact, it has been shown in a number of studies that the automated approach of today rivals if not surpasses manual design for complex, irregular logic circuits. This is a major departure from a couple of years ago, when automated layout carried a large overhead.
- **The advent of sophisticated *logic-synthesis* tools.** The logic-synthesis approach allows for the design to be entered at a high level of abstraction using Boolean equations, state machines, or register-transfer languages such as VHDL or Verilog. The synthesis tools automatically translate this specification into a gate netlist, minimizing a specific cost function such as area, delay, or power. Early synthesis tools—such as those used in the first half of the 1980s—focused mostly on two-level logic minimization. While this enabled automatic design mapping for the first time, it limited the area efficiency and the performance of the generated circuits. It is only with the arrival of *multilevel logic synthesis* in the late 1980s that automated design generation has really taken off. Today, virtually no designer uses the standard-cell approach without resorting to automatic synthesis. A more detailed description of the design synthesis process can be found in "Design Methodology Insert F" which follows this chapter.

Historical Perspective: The Programmable Logic Array

In the early days of MOS integrated circuit design, logic design and optimization was a manual and labor-intensive task. Karnaugh maps and Quine–McCluskey tables were the techniques of choice at that time. In the late 1970s, a first approach toward automating the tedious process of designing logic circuits emerged, triggered by two important developments:

- Rather than using the ad hoc approach to laying out logic circuits, a regular structured design approach was adopted called the *Programmable Logic Array* or PLA. This methodology enabled the automatic layout generation of two-level logic circuits, and, more importantly, it did so in a predictable fashion in terms of area and performance.
- The emergence of automated logic synthesis tools for two-level logic [Brayton84] made it possible to translate any possible Boolean expression into an optimized two-level (sum-of-products or product-of-sums) logic structure. Tools for the synthesis of sequential circuits followed shortly thereafter.

The idea of structured logic design gained a rapid foothold, and already in the mid-1980s it was adopted by major microprocessor design companies such as Intel and DEC. While PLAs are only sparingly used in today's semicustom logic design, the topic deserves some discussion (especially since PLAs might be poised for a come-back).

The concept is best explained with the aid of an example. Consider the following logic functions, for which we have transformed the equations into the sum-of-products format by using logic manipulations:

$$\begin{aligned} f_0 &= x_0x_1 + \bar{x}_2 \\ f_1 &= x_0x_1x_2 + \bar{x}_2 + \bar{x}_0x_1 \end{aligned} \tag{8.1}$$

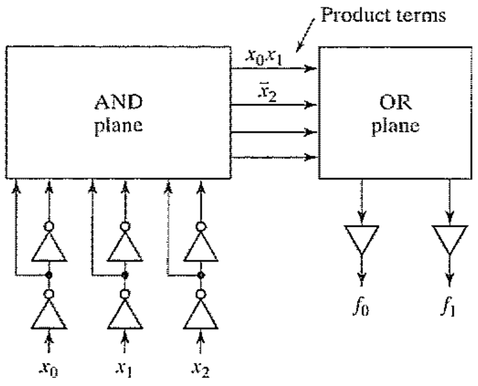


Figure 8-10 Regular two-level implementation of Boolean functions.

One important advantage of this representation is that a regular realization is easily conceived, as illustrated in Figure 8-10. A first layer of gates implements the AND operations—also called *product terms* or *minterms*—while a second layer realizes the OR functions, called the *sumterms*. Hence, a PLA is a rectangular macrocell, consisting of an array of transistors aligned to form rows in correspondence with product terms, and columns in correspondence with inputs and outputs. The input and output columns partition the array into two subarrays, called AND and OR planes, respectively.

The schematic of Figure 8-10 is not directly realizable since single-layer logic functions in CMOS are always inverting. With a few simple Boolean manipulations, Eq. (8.1) can be rewritten into a NOR–NOR format:

$$\begin{aligned} \bar{f}_0 &= \overline{\overline{(\bar{x}_0 + \bar{x}_1)} + \bar{x}_2} \\ \bar{f}_1 &= \overline{\overline{(\bar{x}_0 + \bar{x}_1 + \bar{x}_2)} + \bar{x}_2 + \overline{(x_0 + \bar{x}_1)}} \end{aligned} \tag{8.2}$$

Problem 8.1 Two-Level Logic Representations

It is equally conceivable to represent Eq. (8.1) in a NAND–NAND format. In general, the NOR–NOR representation is preferred due to the prohibitively slow speed of large fan-in NAND gates. The NAND–NAND configuration is very dense, however, and thus can help to reduce power consumption. Derive the NAND–NAND representation for the example of Eq. (8.2).

Translating a set of two-level logic functions into a physical design now boils down to a "programming" task—that is, deciding where to place transistors in both the AND and the OR planes. This task is easily automated—hence, the early success of PLAs. An automatically generated PLA implementation of the logic functions described by Eq. (8.2) is shown in Figure 8-11. Unfortunately, the regular structure, while predictable,

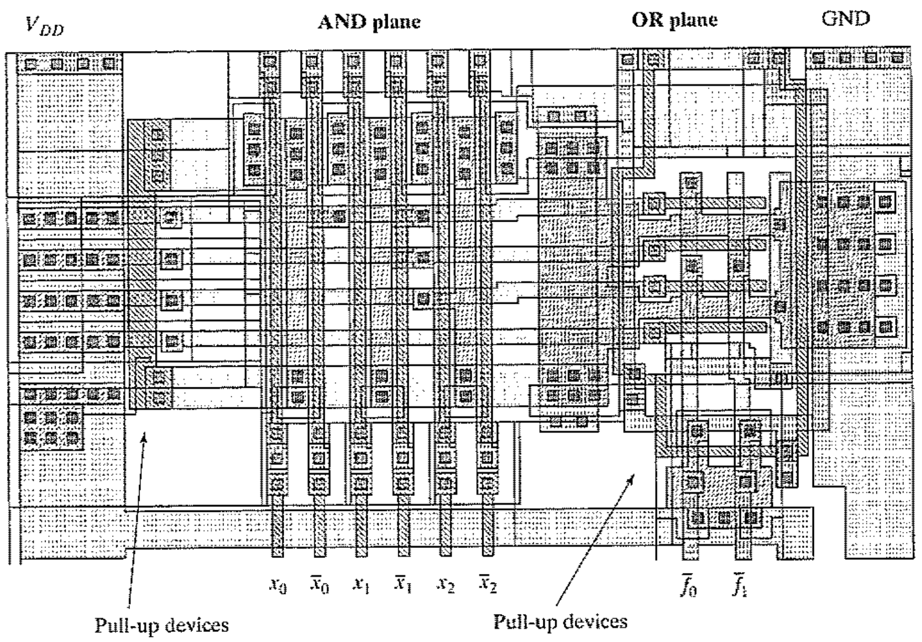


Figure 8-11 PLA layout implementing Eq. (8.2).

brings with it a lot of overhead in area and delay (as is quite visible in the layout), which was its ultimate demise in the semicustom design world. Those who are curious on how these AND and OR planes are actually implemented must wait until we get to Chapter 12, where we discuss the transistor-level implementation of PLAs. ■

8.4.2 Compiled Cells

The cost of implementing and characterizing a library of cells should not be underestimated. Today's libraries contain from several hundred to more than a thousand cells. These cells have to be redesigned with every migration to a new technology. Moreover, changes happen during the development of a single technology generation. For example, minimum metal widths or contact rules often are changed to improve yield. As a result, the complete library has to be laid out and characterized again. In addition, even an extensive library has the disadvantage of being discrete, which means that the number of design options is limited. When targeting performance or power, customized cells with optimized transistor sizes are attractive. With the increased impact of interconnect load, providing cells with adjusted driver sizes is an absolute necessity from both a performance and a power perspective [Sylvester98]—hence, the quest for automated (or compiled) cell generation.

A number of automated approaches have been devised that generate cell layouts on the fly, given the transistor netlists, but high-quality automatic cell layout has remained elusive. Earlier approaches relied on fixed topologies. Later approaches allowed for more flexibility in the transistor placement (e.g., [Hill85]). Layout densities close to what can be accomplished by a human designer are now within reach, and a number of cell-generation tools are commercially available—for example [Cadabra01, Prolific01]:

Example 8.3 Automatic Cell Generation

The flow of a typical cell-generation process is illustrated with the example of a simple inverter (using the Abracad tool [Cadabra01]).

- The cell schematics are developed first. The Spice netlist is the starting point for the automatic layout generation. The generator examines the netlist and starts with transistor geometries. In case of a CMOS inverter, the cell contains just two transistors (see Figure 8-12a) .
- The tool proceeds along the same lines that a designer would follow. The transistors are placed in a cell architecture with predefined topology rules (Figure 8-12b). This architecture is common for all the cells in the library, including the cell height, power rails, pin placements, routing and contact styles.
- The cell is routed symbolically (Figure 8-12c).
- The routing is rearranged, and the cell is compacted to meet design rules and library preferences (Figure 8-12d).
- The final step cleans the cell of any remaining design rule errors and produces the final layout (Figure 8-12e).

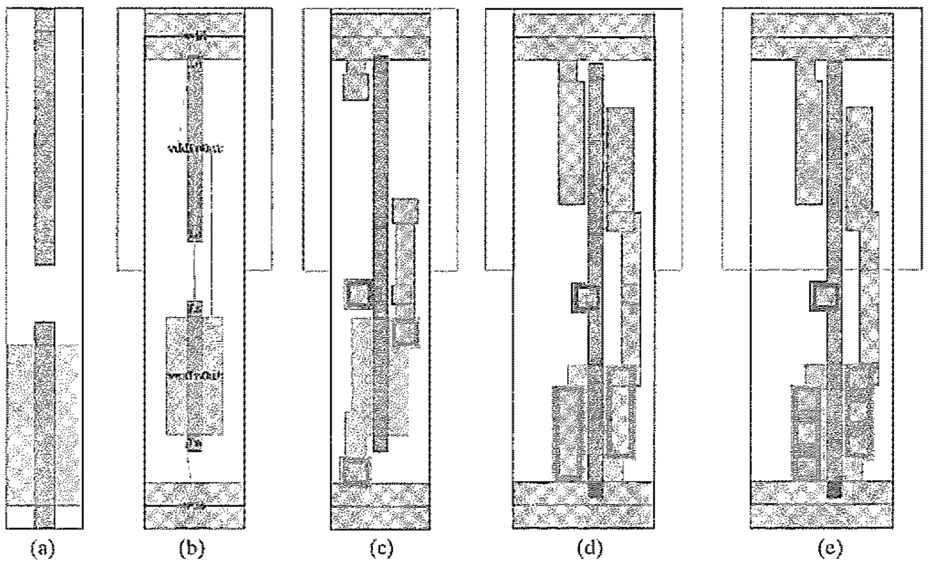

(a) (b) (c) (d) (e)

Figure 8-12 Automatic cell layout (a) initial transistor geometries, (b) placed transistors with flylines indicating intended interconnections, (c) initially routed cell, and (d) compacted cell, (e) finished cell.

8.4.3 Macrocells, Megacells and Intellectual Property

Standardizing at the logic-gate level is attractive for random logic functions, but it turns out to be inefficient for more complex structures such as multipliers, data paths, memories, and embedded microprocessors and DSPs. By capturing the specific nature of these blocks, implementations can be obtained that outperform the results of the standard ASIC design process by a wide margin. Cells that contain a complexity that surpasses what is found in a typical standard-cell library are called *macrocells* (or, sometimes, *megacells*). Two types of macrocells can be identified:

The Hard Macro represents a module with a given functionality and a predetermined physical design. The relative location of the transistors and the wiring within the module is fixed. In essence, a hard macro represents a custom design of the requested function. In some cases, the macro is parameterized, which means that versions with slightly different properties are available or can be generated. Multipliers and memories are examples: A hard multiplier macro may not only generate a 32×16 multiplier, but also an 8×8 one.

The advantage of the hard macro is that it brings with it all the good properties of custom design: dense layout, and optimized and predictable performance and power dissipation. By encapsulating the function into a macromodule, it can be reused over and over in different designs. This reuse helps to offset the initial design cost. The disadvantage of the hard macro is that it is hard to port the design to other technologies or to other manufacturers. For every new technology, a major redesign of the block is necessary. For this reason, hard macros are used less and less, and are employed mainly when the automated generation approach is far inferior or even impossible. Embedded memories and microprocessors are good examples of hard macros. They typically are provided by the IC manufacturer (who also provides the standard cell library), or the semiconductor vendor who has a particularly desirable function to offer (such as a standard microprocessor or DSP).

In the case of a macro that can be parameterized, a generator called the *module compiler* is used to create the actual physical layout. Regular structures such as PLAs, memories, and multipliers are easily constructed by abutting predesigned leaf cells in a two-dimensional array topology. All interconnections are made by abutment, and no or little extra routing is needed if the cells are designed correctly, which minimizes the parasitic capacitance. The PLA of Figure 8-11 is an example of such a configuration. The whole array can be constructed with a minimal number of cells. The generator itself is a simple software program that determines the relative positioning of the various leaf cells in the array.

Example 8.4 A Memory Macromodule

Figure 8-13 shows an example of a "hard" memory macrocell. The 256×32 SRAM block is generated by a parameterizable module generator. Besides creating the layout, the generator also provides accurate timing and power information. Modern memory generators also include an amount of redundancy to deal with defects.

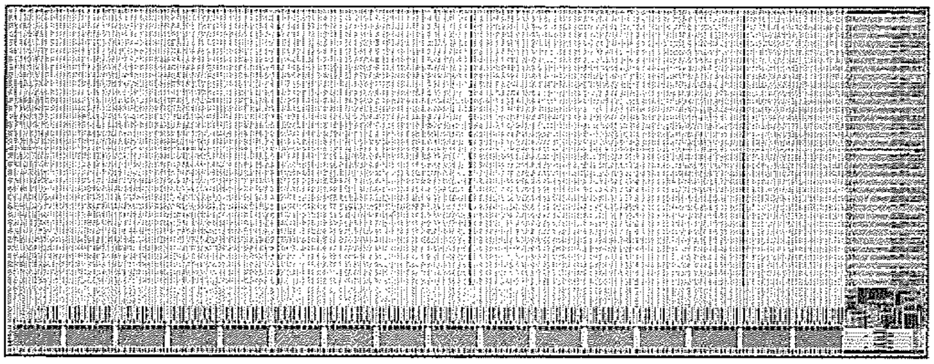


Figure 8-13 Parameterizable memory "hard" macrocell. This particular instance stores 256 × 32 (or 8192) bits. The decoders are located on the bottom. All eight address bits, as well as the 32 data input and output ports are placed on the right side of the cell. The total area of the memory module, implemented in a 0.18-μm CMOS technology, equals a mere 0.094 mm^2 (courtesy ST Microelectronics).

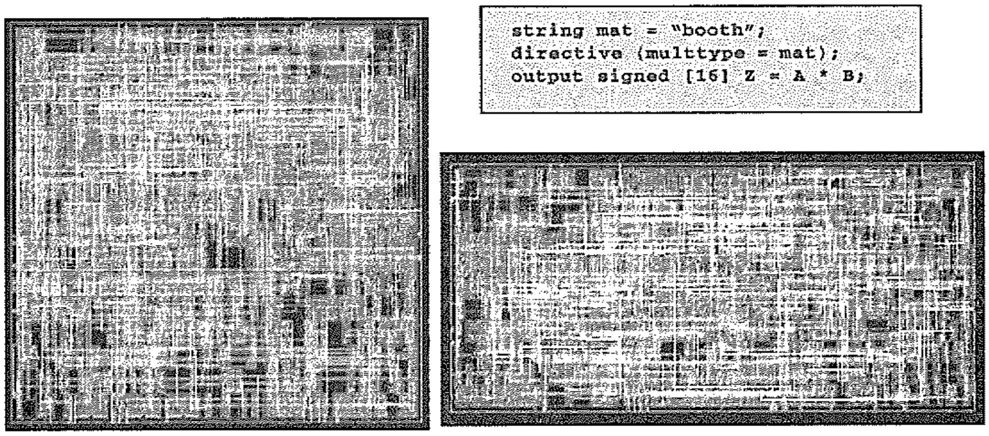
A Soft Macro represents a module with a given functionality, but without a specific physical implementation. The placement and the wiring of a soft macro may vary from instance to instance. This means that the timing data can only be determined after the final synthesis and placement and routing steps—in other words, the process is unpredictable. Yet, through intrinsic knowledge of the internal structure of the module, and by imposing precise timing and placement constraints on the physical generation process, soft macros most often succeed in offering well-defined timing guarantees. While stepping away from the advantages of the custom design process and relying on the semicustom physical design process, soft macros have the major advantage that they can be ported over a wide range of technologies and processes. This amortizes the design effort and cost over a wide set of designs.

Soft-macrocell generators come in different styles depending on the type of function they target. Virtually all of them can be classified as *structural generators*: Given the desired function and values for the requested parameters, the generator produces a netlist, which is an enumeration of the standard cells used and their interconnections. It also provides a set of timing constraints that the placement and routing tools should meet. The advantage of this approach is that the generator exploits its knowledge of the function under consideration to come up with clever structures that are more efficient than what logic synthesis would produce. For example, the design of fast and area-efficient multipliers has been the topic of decades of research.[1] The multiplier generator just incorporates the best of what the multiplier literature has to offer into an automated generation tool.

[1]Multiplier design is explained more thoroughly in Chapter 11, which discusses the design of arithmetic structures.

Example 8.5 Multiplier Macromodule

Two instances of an 8 × 8 multiplier module with different aspect ratios are shown in Figure 8-14. The modules are generated using the ModuleCompiler tool from Synopsys [ModuleCompiler01]. As can observed from the layout, a common standard-cell methodology is used to generate the physical artwork. The contribution of the macrocell generator is to translate the compact input description into an optimized connection of standard cells that meets the timing constraints. This "soft" approach has the advantage that modules with different aspect ratios can easily be generated. Also, porting between different manufacturing technologies is relatively easy.



```
string mat = "booth";
directive (multtype = mat);
output signed [16] Z = A * B;
```

Figure 8-14 Multiplier "soft" macro modules. Both layouts implement an 8 × 8 booth multiplier, but with different aspects ratios. The compact input description to the compiler is shown in the gray box on top.

The availability of macromodules has substantially changed the semicustom design landscape in the 21st century. With the complexity of ICs going up exponentially, the idea of building every new IC from scratch becomes an uneconomic and nonplausible proposition. More and more, circuits are being built from reusable building blocks of increasing complexity and functionality. Typically, these modules are acquired from third-party vendors, who make the functions available through royalty or licensing agreements. Macromodels distributed in this style are called *intellectual property* (or IP) modules. This approach is somewhat comparable to the software world, where a large programming project typically makes intensive use of reusable software libraries. Good examples of commonly available intellectual property modules are embedded microprocessors and microcontrollers, DSP processors, bus interfaces such as PCI, and several special-purpose functional modules such as FFT and filter modules for DSP applications, error-correction coders for wireless communications, and MPEG decoding and encoding for video. Obviously, for an IP module to be useful, it has to not only deliver the hardware, but it also has to come with the appropriate software tools (such as compilers and debuggers for embedded processors), prediction models, and test benches. The latter are quite important because they represent the only means for the end user to verify that the module delivers the promised functionality and performance.

The design of a system on a chip is rapidly becoming an exercise in reuse at different levels of granularity. At the lowest level, we have the standard cell library; at a level higher, we have the functional modules such as multipliers, datapaths and memories; next, we have the embedded processors; and finally, the application-specific megacells. With more and more of the system functionality migrating onto a single die, it is not surprising to see that a typical ASIC consists of a blend of design styles and modules, embedding a number of hard or soft macrocells within a sea of standard cells.

Example 8.6 A Processor for Wireless Communications

Figure 8-15 shows an integrated circuit implementing the protocol stack for a wireless indoor communication system [Silva01]. The majority of the area is occupied by the embedded microprocessor (the Tensilica Xtensa processor [Xtensa01]) and its memory system. This processor allows for a flexible implementation of the higher levels of the protocol stack (Application/Network), and enables changes in the functionality of the chip, even after fabrication. The memory modules are generated using module compilers provided by the process vendor. The processor core itself is automatically generated from a higher level description in Verilog, and uses standard cells for its physical implementation. The advantage of using the "soft-core" approach is that the processor instruction set can be

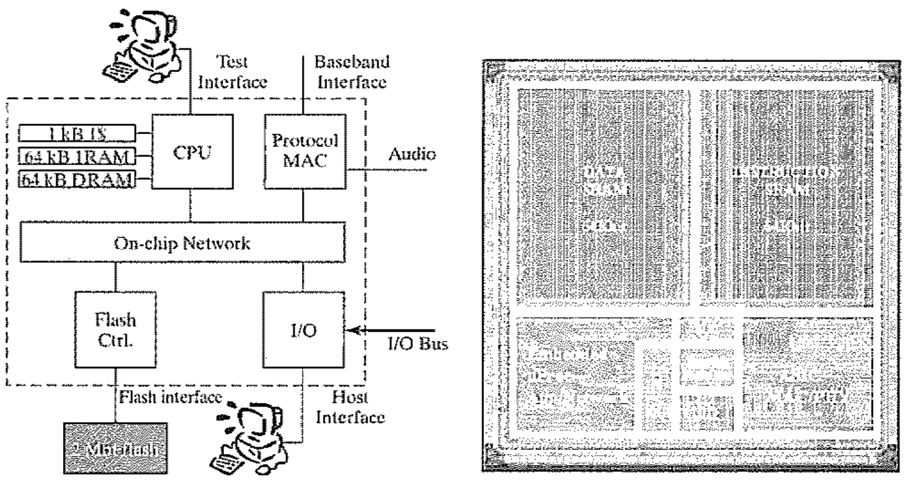


Figure 8-15 Wireless communications processor—an example of a hybrid ASIC design methodology. The processor combines an embedded microprocessor and its memory system with dedicated hardware accelerators and I/O modules. Observe also the on-chip network module [Silva01].

tailored to the application, and that the processor itself can easily be ported to different technologies and fabrication processes.

Implementing the computation-intensive parts of the protocol (MAC/PHY) on the microprocessor would require very high clock speeds and would unnecessarily increase the power dissipation of the chip. Fortunately, these functions are fixed and typically do not require a flexible implementation. Hence, they are implemented as an accelerator module in standard cells. The hard-wired implementation accomplishes the task of implementing a huge number of computations at a relatively low power level and clock frequency. The designer of a system on a chip is continuously faced with the challenge of deciding what is more desirable—after-the-fabrication flexibility versus higher performance at lower power levels. Fortunately, tools are emerging that help the designer to explore the overall design space and analyze the trade-offs in an informed fashion [Silva01]. Observe also that the chip contains a set of I/O interfaces, as well as an embedded network module, which helps to orchestrate the traffic between processor and the various accelerator and I/O modules.

The generation process of a macro module depends on the hard or soft nature of the block, as well as the level of design entry. In the following sections, we briefly discuss some commonly: used approaches.

8.4.4 Semicustom Design Flow

So far, we have defined the components that make up the cell-based design methodology. In this section, we discuss how it all comes together. Figure 8-16 details the traditional sequence of steps to design a semicustom circuit. The steps of what we call the design flow are enumerated in the figure, with a brief description of each:

1. **Design Capture** enters the design into the ASIC design system. A variety of methods can be used to do so, including schematics and block diagrams; hardware description languages (HDLs) such as VHDL, Verilog, and, more recently, C-derivatives such as SystemC; behavioral description languages followed by high-level synthesis; and imported intellectual property modules.
2. **Logic Synthesis** tools translate modules described using an HDL language into a *netlist.* Netlists of reused or generated macros can then be inserted to form the complete netlist of the design.
3. **Prelayout Simulation and Verification**. The design is checked for correctness. Performance analysis is performed based on estimated parasitics and layout parameters. If the design is found to be nonfunctional, extra iterations over the design capture or the logic synthesis are necessary.
4. **Floor Planning**. Based on estimated module sizes, the overall outlay of the chip is created. The global-power and clock-distribution networks also are conceived at that time.

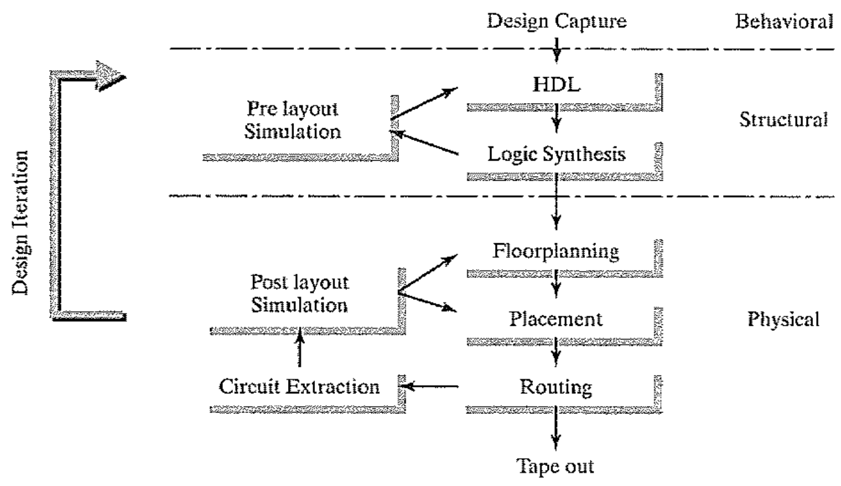


Figure 8-16 The Semicustom (or ASIC) design flow.

5. **Placement.** The precise positioning of the cells is decided.
6. **Routing.** The interconnections between the cells and blocks are wired.
7. **Extraction.** A model of the chip is generated from the actual physical layout, including the precise device sizes, devices parasitics, and the capacitance and resistance of the wires.
8. **Postlayout Simulation and Verification.** The functionality and performance of the chip is verified in the presence of the layout parasitics. If the design is found to be lacking, iterations on the floorplanning, placement, and routing might be necessary. Very often, this might not solve the problem, and another round of the structural design phase might be necessary.
9. **Tape Out.** Once the design is found to be meeting all design goals and functions, a binary file is generated containing all the information needed for mask generation. This file is then sent out to the ASIC vendor or foundry. This important moment in the life of a chip is called *tape out.*

While the design flow just described has served us well for many years, it was found to be severely lacking once technology reached the 0.25-μm CMOS boundary. With design technology proceeding into the deep submicron region, layout parasitics—especially from the interconnect—are playing an increasingly important role. The prediction models used by the logic and structural synthesis tools have a hard time providing accurate estimates for these parasitics. The chances that the generated design meets the timing constraints at the first try are thus very small (Figure 8-17a). The designer (or design team) is then forced to go through a number of costly iterations of synthesis followed by layout generation until an acceptable artwork that meets the timing constraints is obtained (Figure 8-17b and c). Each of these iterations may take several days—just routing a complex chip can take a week on the most advanced computers! The

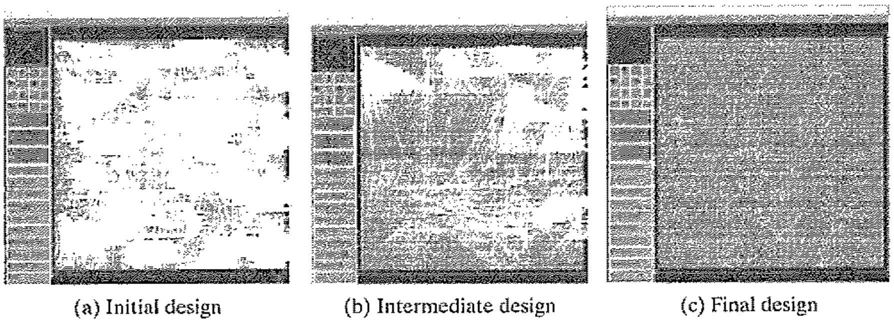


Figure 8-17 The timing closure process. The white lines indicate nets with timing violations. In each iteration of the design process, timing errors are removed by optimizing the logic, by insertion of buffers, by constraining the placement, or by streamlining the routing until an error-free design is obtained [Avanti01].

number of needed iterations continues to grow with the scaling of technology. This problem, called *timing closure*, made it obvious that new solutions and a change in design methodology were required.

The common answer is to create a tighter integration between the logical and physical design processes. If the logic synthesis tool, for example, also performs some part of the placement—or directs the placement—more precise estimates of the layout parameters can be obtained. Figure 8-18 shows an example of a design environment that merges RTL synthesis with first-order placement and routing. The resulting netlist is then fed into an optimization tool that performs the detailed placement and routing, while guaranteeing the timing constraints are met. While this approach has shown to be quite successful in reducing the number of design iter-

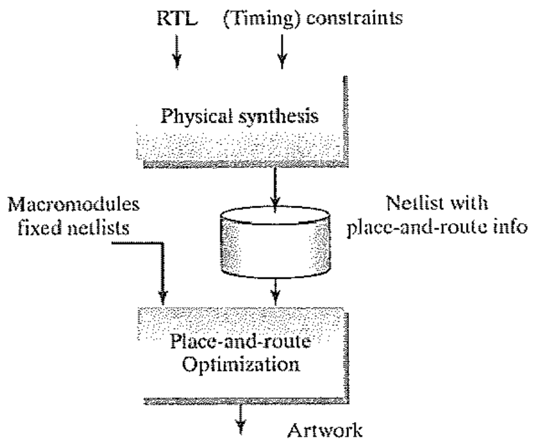

RTL
(Timing) constraints
Physical synthesis
Macromodules fixed netlists
Netlist with place-and-route info
Place-and-route Optimization
Artwork

Figure 8-18 Integrated synthesis place-and-route reduces the number of iterations to reach timing closure in deep submicron.

ations, it throws quite a challenge at the design-tool developers. With the number of parasitic effects increasing with every round of technology scaling, the design optimization process that must take all this into account becomes exponentially complex as well. As a result, other approaches might be required as well. In the coming chapters, we will highlight "design solutions" that can help to alleviate some of these problems. An example is the use of regular and predictable structures, both at the logical and the physical level.

8.5 Array-Based Implementation Approaches

While design automation can help reduce the design time, it does not address the time spent in the manufacturing process. All of the design methodologies discussed thus far require a complete run through the fabrication process.This can take from three weeks to several months, and it can substantially delay the introduction of a product. Additionally, with ever-increasing mask costs, a dedicated process run is expensive, and product economics must determine if this is a viable route.

Consequently, a number of alternative implementation approaches have been devised that do not require a complete run through the manufacturing process, or they avoid dedicated processing completely. These approaches have the advantage of having a lower NRE (nonrecurring expense) and are, therefore, more attractive for small series. This comes at the expense of lower performance, lower integration density, or higher power dissipation.

8.5.1 Prediffused (or Mask-Programmable) Arrays

In this approach, batches of wafers containing arrays of primitive cells or transistors are manufactured by the vendors and stored. All the fabrication steps needed to make transistors are standardized and executed without regard to the final application.

To transform these uncommitted wafers into an actual design, only the desired interconnections have to be added, determining the overall function of the chip with only a few metallization steps. These layers can be designed and applied to the premanufactured wafers much more rapidly, reducing the turnaround time to a week or less.

This approach is often called the *gate-array* or the *sea-of-gates* approach, depending on the style of the prediffused wafer. To illustrate the concept, consider the gate-array primitive cell shown in Figure 8-19a. It comprises four NMOS and four PMOS transistors, polysilicon gate connections, and a power and ground rail. There are two possible contact points per diffusion area and two potential connection points for the polysilicon strips. We can turn this cell, which does not implement any logic function so far, into a real circuit by adding some extra wires on the metal layer and contact holes. This is illustrated in Figure 8-19b, where the cell is turned into a four-input NOR gate.

The original *gate-array* approach[2] places the cells in rows separated by wiring channels, as shown in Figure 8-20a. The overall look is similar to the traditional standard-cell technique. With the advent of extra metallization layers, the routing channels can be eliminated, and routing can

[2]This approach is often called the *channeled* gate array.

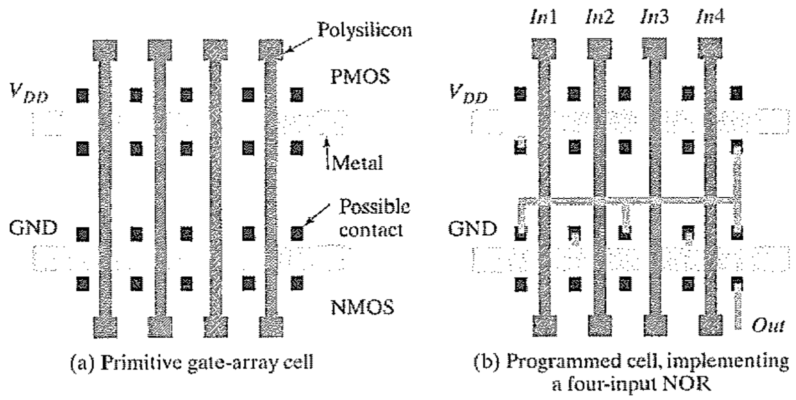


Figure 8-19 An example of the gate-array approach.

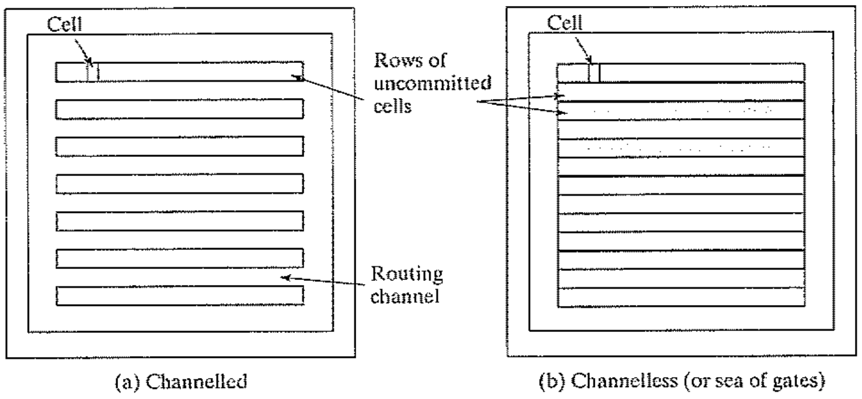


Figure 8-20 Gate-array architectures.

be performed on top of the primitive cells—occasionally leaving a cell unused. This channelless architecture, also called *sea of gates* (Figure 8-20b), yields an increased density, and makes it possible to achieve integration levels of millions of gates on a single die. Another advantage of the sea-of-gates approach is that it customizes the contact layer between metal-1 and diffusion and/ or polysilicon, in contrast to the standard gate-array approach where the contacts are predefined (see Figure 8-19a). This extra flexibility leads to a further reduction in cell size.

The primary challenge when designing a gate-array (or sea-of-gates) template is to determine the composition of the primitive cell and the size of the individual transistors. A sufficient number of wiring tracks must be provided to minimize the number of cells wasted to interconnect. The cell should be chosen so that the prefabricated transistors can be utilized to a maximal extent over a wide range of designs. For example, the configuration of Figure 8-19 is well suited for the realization of four-input gates, but wastes devices when implementing two-input gates.

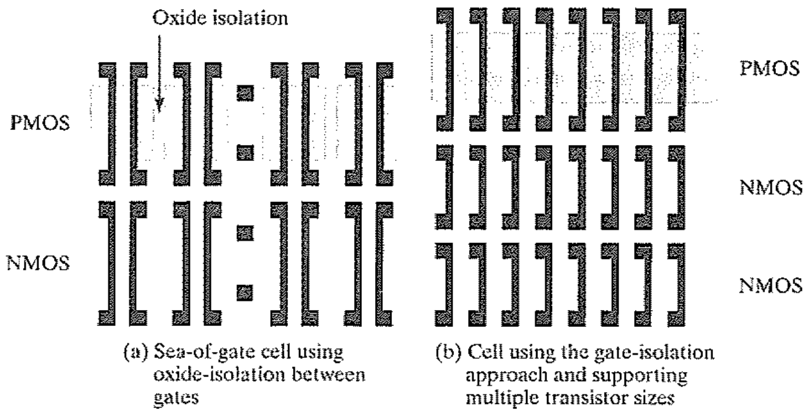

Oxide isolation
PMOS
NMOS
PMOS
NMOS
NMOS
(a) Sea-of-gate cell using oxide-isolation between gates
(b) Cell using the gate-isolation approach and supporting multiple transistor sizes

Figure 8-21 Examples of sea-of-gates primitive cells (from [Veendrick92]).

Multiple cells are needed when implementing a flip-flop. A number of alternative cell structures are pictured in Figure 8-21 in a simplified format. In one approach, each cell contains a limited number of transistors (four to eight). The gates are isolated by means of *oxide isolation* (also called *geometry isolation*). The "dog-bone" terminations on the poly gates allow for denser routing. A second approach provides long rows of transistors, all sharing the same diffusion area. In this architecture, it is necessary to electrically turn off some devices to provide isolation between neighboring gates by tying NMOS and PMOS transistors to *GND* and V_{DD}, respectively. This technique is called *gate isolation*. This approach wastes a number of transistors to provide the isolation, but provides an overall higher transistor density.

Figure 8-22 shows the base cell for a gate-isolated gate array (from [Smith97]). The cell is one routing track wide, and contains one p-channel and one n-channel transistor. Also shown is a base cell containing all possible contact positions. There is room for 21 contacts in the vertical direction, which means that the cell has a height of 21 tracks.

It is worth observing that the cell in Figure 8-21b provides two rows of smaller NMOS transistors that can be connected in parallel if needed. Smaller transistors come in handy when implementing pass-transistor logic or memory cells. Sizing the transistors in the cells is a clear challenge. Due to the interconnect-oriented nature of the array-based design methodology, the propagation delay is generally dominated by the interconnect capacitance. This seems to favor larger device sizes that cause a larger area loss when unused. On the other hand, it is possible to construct larger transistors by putting several smaller devices in parallel.

Mapping a logic design onto an array of cells is a largely automated process, involving logic synthesis followed by placement and routing. The quality of these tools has an enormous impact on the final density and performance of a sea-of-gates implementation. Utilization factors in sea-of-gates structures are a strong function of the type of application being implemented. Utilization factors of nearly 100% can be obtained for regular structures such as memories. For

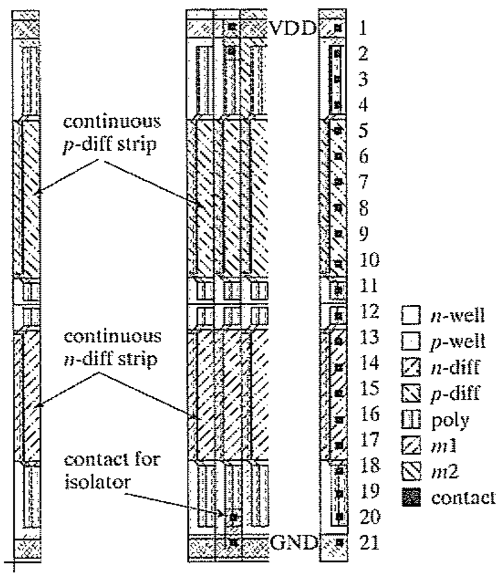


Figure 8-22 Base cell of gate-isolated gate array (from [Smith97]).

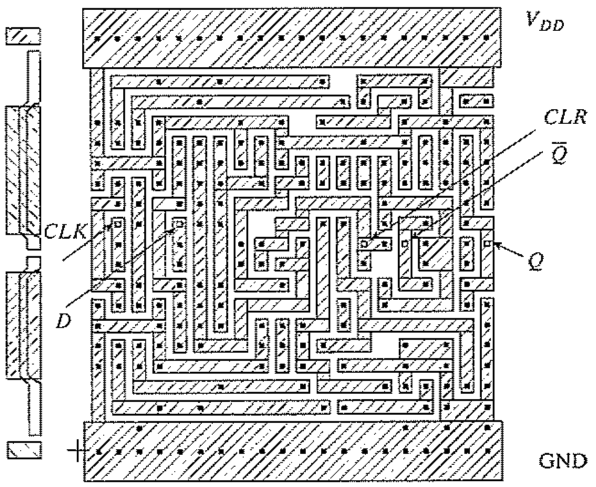


Figure 8-23 Flip-flop implemented in a gate-isolated gate-array library. The base cell is shown on the left (from [Smith97]).

other applications, utilization factors can be substantially lower (< 75%), due largely to wiring restrictions. Figure 8-23 shows an example of a flip-flop macrocell, implemented in a gate-isolated, gate-array library.

Similar to the scenarios unfolding in the standard-cell arena, designers of sea-of-gate arrays discovered that a design with a large number of gates also has large memory needs. Implementing these memory cells on top of the gate-array base-cells is possible, but not very efficient. A more efficient approach is to set aside some area for dedicated memory modules. The mixing of gate arrays with fixed macros is called the *embedded gate-array* approach. Other modules such as microprocessor and microcontrollers are also ideal candidates for embedding.

Example 8.7 Sea-of-Gates

An example of a sea-of-gates implementation is shown in Figure 8-24. The array has a maximum capacity of 300 K gates and is implemented in a 0.6-µm CMOS technology. The upper left part of the array implements a memory subsystem, which results in a regular modular layout. The rest of the array implements random logic.

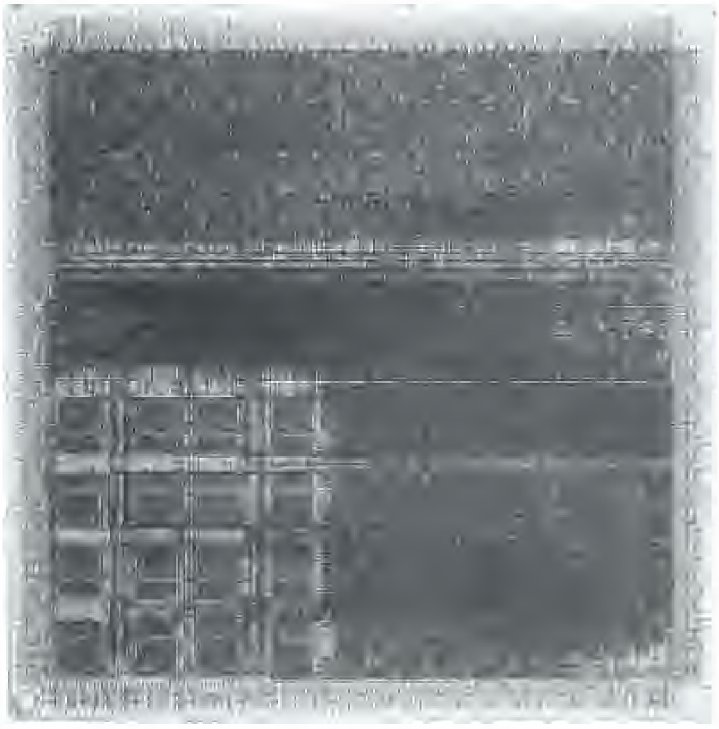


Figure 8-24 Gate-array die microphotograph (LEA300K) (Courtesy of LSI Logic.)

Design Consideration—Gate Arrays versus Standard Cells

In the 1980s and 1990s, when the majority of the chips were less than 50,000 gates, design cycles often could be measured in weeks or a few months. The two- or three-week savings in turnaround time for a gate-array design was then a significant portion of the total design cycle, more than enough to offset the additional die size. With today's deep-submicron processes and multimillion-gate complexities come longer design times, and the small reduction in turnaround time is no longer much of an issue. Furthermore, metallization has become the most time-consuming and yield-impacting part of the semiconductor manufacturing process, reducing further the advantage that gate arrays had to offer. Consequently, gate arrays have lost a lot of their luster. Another alternative for rapid prototyping—the prewired arrays discussed in the next section—has arisen, and it has taken a large portion out of the gate-array market.

Still, beware of dismissing the idea of the mask-programmable logic module as a thing of the past. A regular and fixed layout style has the advantage that load factors, wiring parasitics, and cross-coupling noise are easily and accurately estimated. This is in contrast to the standard-cell approach, where these values are ultimately only known after placement, routing, and extraction. One may consider populating sections of a large chip with a regular logic array consisting of uncommitted (prediffused) logic cells superimposed by a wiring grid. The actual programming of the module is performed by placing vias at predefined positions. As shown in Figure 8-25, the use of a via-programmable cross-point switch makes it possible to overlay a wide variety of wiring patterns on a regular repetitive wiring grid. It is the opinion of the authors that prediffused arrays have quite some life left into them.

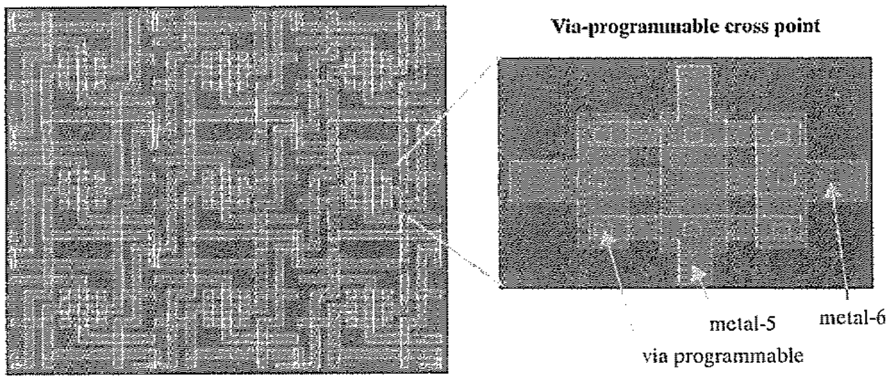


Figure 8-25 Via-programmable gate array. Vias are used to dedicate a generic wiring grid to a specific wiring pattern, resulting in predictable arrays [Pileggi02].

■

8.5.2 Prewired Arrays

While the prediffused arrays offer a fast road to implementation, it would be even more efficient if dedicated manufacturing steps could be avoided altogether. This leads to the concept of the preprocessed die that can be programmed in the field (i.e., outside the semiconductor foundry) to implement a set of given Boolean functions. Such a programmable, prewired array of cells is called a *field-programmable gate array (FPGA)*. The advantage of this approach is that the manufacturing process is completely separated from the implementation phase and can be amortized over a large number of designs. The implementation itself can be performed at the user site with negligible turnaround time. The major drawback of this technique is a loss in performance and design density, compared with the more customized approaches.

Two main issues have to be addressed when attempting to implement a set of Boolean functions on top of a regular array of cells without requiring any processing steps:

1. How do we implement "programmable" logic—that is, logic that can committed to perform any possible Boolean function?
2. How and where do we store the *program*—also called the configuration—that dedicates the programmable array to a certain logic function?

The answer to the second question depends on the memory technology used. Since memory technology is the topic of a later chapter, we limit ourselves here to a high-level overview. In general, three different techniques can be identified:

- **The write-once or fuse-based FPGA.** The logic array is committed to a particular function by blowing "fuses" or by short-circuiting "antifuses." A fuse is a connection element that is short-circuited by default. A large current causes it to blow, and then it becomes an open circuit. The antifuse has the opposite behavior. An example of an antifuse implementation is shown in Figure 8-26 [El-Ayat89]. The advantage of the write-once approach is that the area overhead of the program memory (i.e., the fuses) is very small. But it has the important disadvantage of being *one-time programmable*. Circuit corrections or extensions are not possible, and new components are required for every design change.
- **The nonvolatile FPGA.** The program is stored in nonvolatile memory, which is memory that retains its value even when the supply voltage is turned off. Examples include EEPROM (*Electrically Erasable Programmable Read-Only Memory*) or Flash memories. Once programmed, the logic remains functional and fixed until a new programming round. The disadvantage of this approach is that nonvolatile memories require special steps in the manufacturing process, such as the deposition of ultrathin oxides. Also, high voltages

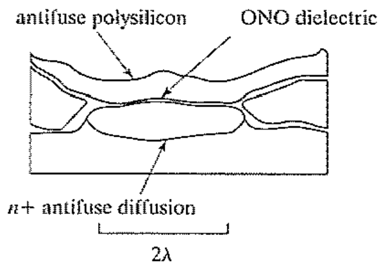


Figure 8-26 Example of antifuse. A 10-nm-thin layer (< 10 nm) of ONO (oxide–nitride–oxide) dielectric is deposited between conducting polysilicon and diffusion layers. The circuit is open by default, unless a large programming current is forced through it. This causes the dielectric to melt, and a permanent connection with fixed resistance is formed (from [Smith97]).

(> 10 V) are needed for the programming and erasure of the memory cells. Generating these high voltages and distributing them through the logic array adds extra complexity to the design.

- **The Volatile or RAM-Based FPGA.** This popular approach to programming the logic array employs volatile static RAM (random-access memory) cells for the storage of the program. Since these memories lose their stored contents when the FPGA is powered down, a reloading of the configuration from an external permanent memory is necessary every time the part is turned on. To program the component at start-up time, programming data is shifted serially into the part over a single line (or pin). For all practical purposes, one can consider the FPGA RAM cells to be configured as a giant shift register during that period. Once all memories are loaded, normal execution is started. The configuration time is proportional to the number of programmable elements. This can become excessive for today's larger FPGAs, which often feature more than one million gates. Recent parts therefore rely more and more on a parallel programming interface, allowing multiple cells to be programmed at the same time.

  In contrast to their nonvolatile counterparts, volatile FPGAs do not have special manufacturing process requirements, and can be implemented in a regular CMOS process. In addition, designers can reuse chips during prototyping. Logic can be modified and upgraded once deployed in the field—a customer can be sent a new configuration file to upgrade the chip, instead of sending a new chip. In addition, logic can be dynamically modified on the fly during execution. The latter approach is called *reconfiguration*, and it became quite popular in the late 1990s. In some sense, this brings a paradigm that was extremely successful in the world of programming (as embodied by the microprocessor) to the domain of logic design.

As for the first question, the answer is somewhat more extensive. Implementing a complex circuit in a programmable fashion requires that both the logic functions as well as the interconnect between them are realized in a configurable fashion. In the coming sections, we first discuss different ways of implementing programmable logic, followed by an overview of programmable interconnection. Finally, we detail a number of specific ways of putting the two together.

Programmable Logic

Similar to the situation in semicustom design, two fundamentally different approaches towards programmable logic are currently in vogue: array based and cell based.

Array-Based Programmable Logic Earlier we discussed how a *programmable logic array* (PLA) implements arbitrary Boolean logic functions in a regular fashion (see page 388). A similar approach can be applied to field-programmable devices as well. Consider, for example, the logic structure of Figure 8-27. A circle (o) at an intersection indicates a programmable connection—that is, an interconnect point that is either enabled or not. An inspection of the diagram reveals that it is equivalent to a PLA, where both the AND and OR planes can be programmed by selectively enabling connections. This approach allows for the implementation of arbitrary

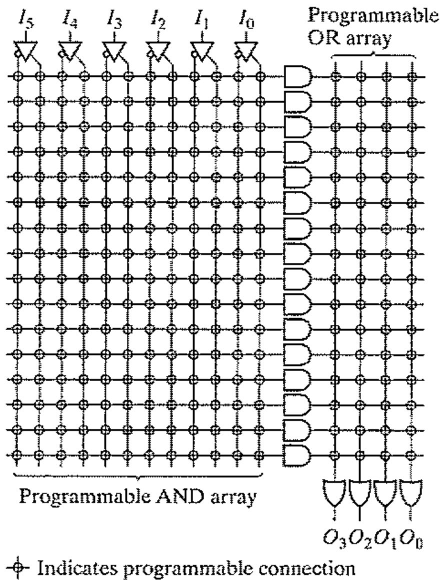

I_5 I_4 I_3 I_2 I_1 I_0
Programmable OR array
Programmable AND array
$O_3 O_2 O_1 O_0$
Indicates programmable connection

Figure 8-27 Fuse-programmable logic array (PLA).

logic functions in a two-level *sum-of-products* format. The AND plane creates the required minterms, while the OR plane takes the sum of a selected set of products to form the outputs. To include a given input variable (for instance, I_1) in a specific minterm, just close the switch at the intersection of the input signal and the minterm. Similarly, a minterm is included into an output by closing the appropriate connection in the OR plane. The functionality of PLA is restricted by the number of inputs, outputs, and minterms.

We can envision variations on this theme, some of which are represented in Figure 8-28. The dot (•) at the intersection of two lines represents a nonfusible, hard-wired link. The first structure represents the PROM architecture, in which the AND plane is fixed and enumerates all possible minterms. The second structure, called a *programmable array logic device* (PAL), is located at the other end of the spectrum, where the OR plane is fixed, and the AND plane is programmable. The PLA architecture is the most generic one for the implementation of arbitrary logic functions. The PROM and PAL structures, on the other hand, trade off flexibility for density and performance. Which structure to select depends strongly on the nature of the Boolean functions to be implemented. All these approaches are generally classified under the common term of *programmable logic devices* (or PLDs).

The single-array architecture of the PLA, PROM, and PAL structures in Figure 8-27 and Figure 8-28 becomes less attractive in the era of higher integration density. First of all, implementing very complex logic functions on a single, large array results in a loss of programming density and performance. Secondly, the arrays shown implement only combinational logic. To realize complete, sequential subdesigns, the presence of registers and/or flip-flops is an absolute requirement. These deficiencies can be addressed as follows:

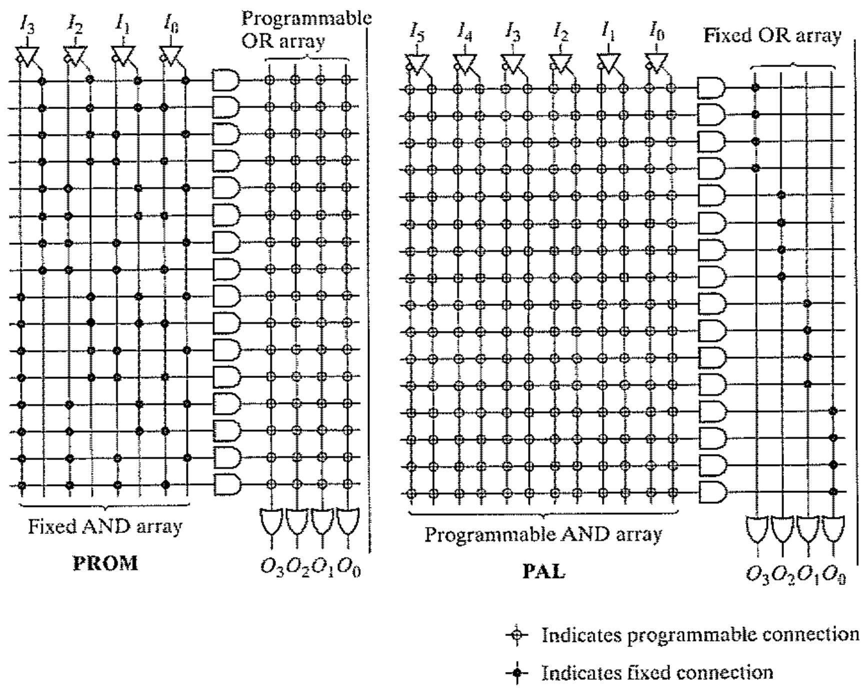


Figure 8-28 Alternative fuse-based programmable logic devices (or PLDs).

1. Partition the array into a number of smaller sections, often called macrocells.
2. Introduce flip-flops and provide a potential feedback from output signals to the inputs.

One example of how this can be accomplished is shown in Figure 8-29. The PAL consists of *k* macrocells, each of which can select from *i* inputs and features, at most, *j* product terms. Each macrocell contains a single register, which also is programmable—it can be configured as a *D*, *T*, *J-K*, or a clocked *S-R* flip-flop. The *k* output signals are fed back to the input bus, and thus form a subset of the *i* input signals.

The PLA approach to configurable logic has two distinct advantages:

- The structure is very regular, which makes the estimation of the parasitics quite easy, and enables accurate predictions of area, speed, and power dissipation.
- It provides an efficient implementation for logic functions that map well into a two-level logic description. Functions with a large fan-in fall into that category. Examples of such are finite-state machines used in controllers and sequencers.

On the other hand, the array structure has the disadvantage of higher overhead. Every intermediate node has a sizable capacitance, which negatively affects performance and power. This is especially true when parts of the array are underutilized—that is, if only some of the minterms are actively used.

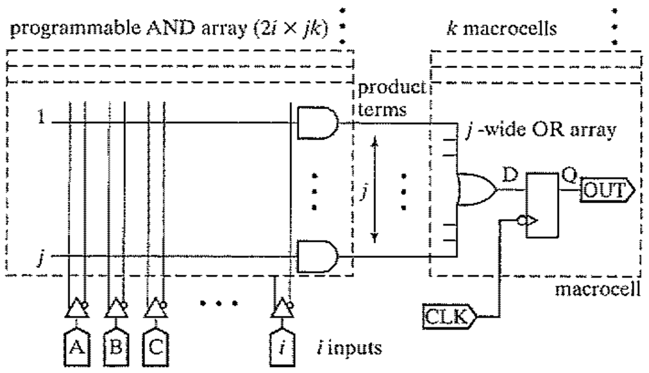


Figure 8-29 Schematic diagram of a PAL with *i* inputs, *j* minterms/macrocell and *k* macrocells (or outputs) [Smith97].

Example 8.8 Example of Programmed Macrocell

Figure 8-30 shows an example of how to program a PROM module. The structure is programmed to realize the logical functions used earlier during the discussion on PLAs (Eq. (8.1)):

$$f_0 = x_0x_1 + \overline{x_2}$$
$$f_1 = x_0x_1x_2 + \overline{x_2} + \overline{x_0}x_1$$

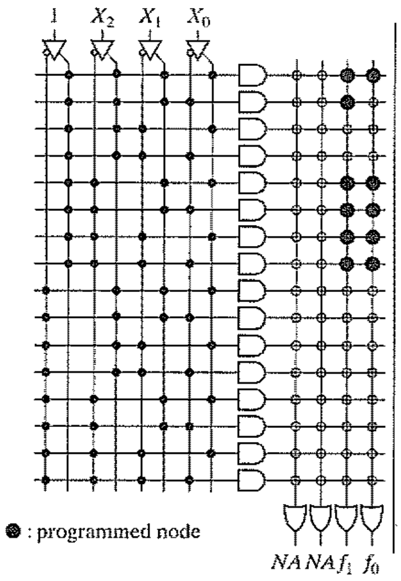


Figure 8-30 Programming a PROM.

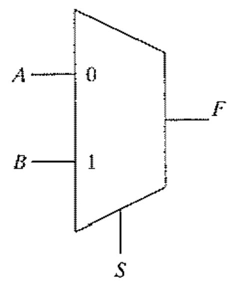
Observe that only a fraction of the array is used as the number of input (3) and output (2) variables are smaller than the dimensions of the 4 × 4 array. Unused input variables are tied either to 0 or 1. The large dots in the output planes represent programmed nodes. The reader is invited to repeat the exercise for the PLA and PAL modules presented in Figure 8-28 and Figure 8-29.

Cell-Based Programmable Logic The sum-of-products approach results in regular structures, and is very effective for logic functions that have a large fan-in such as finite-state machines. On the other hand, it performs rather poorly for logic that features a large fan-out, or that benefits from a multilevel logic implementation. (Arithmetic operations such as addition and multiplication are an example of such). Other approaches can be conceived that are more in line with the multilevel approach favored in the standard-cell and sea-of-gate approaches.

There are many ways to design a logic block that can be configured to perform a wide range of logic functions. One approach is to use *multiplexers as function generators.* Consider the two-input multiplexer of Figure 8-31, which implements the logic function F:

$$F = A \cdot \bar{S} + B \cdot S \tag{8.3}$$

By carefully choosing the connections between the variables X and Y and the input ports A, B, and S of the multiplexer, we can program it to perform ten useful logic operations on one or more of those inputs (see Figure 8-31).



(a)

Configuration			
A	B	S	$F =$
0	0	0	0
0	X	1	X
0	Y	1	Y
0	Y	X	XY
X	0	Y	$X\overline{Y}$
Y	0	X	$\overline{X}Y$
Y	1	X	$X+Y$
1	0	X	$\overline{X}$
1	0	Y	$\overline{Y}$
1	1	1	1

(b)

Figure 8-31 Using a two-input multiplexer (a) as a configurable logic block. By properly connecting the inputs A, B, and S to the input variables X or Y, or to 0 or 1, 10 different logic functions can be obtained (b).

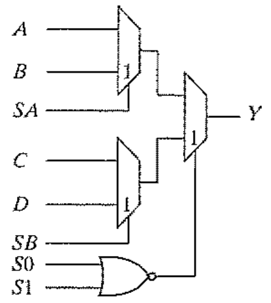


Figure 8-32 Logic cell as used in the Actel fuse-based FPGA.

A number of multiplexers can be combined to form more complex logic gates. Consider, for example, the logic cell of Figure 8-32, which is used in the Actel ACT family of FPGAs. It consists of three two-input multiplexers and a two-input NOR gate. The cell can be programmed to realize any two- and three-input logic functions, some four-input Boolean functions, and a latch.

Example 8.9 Programmable Logic Cell

It can be verified that the logic cell of Figure 8-32 acts as a two-input XOR under the programming conditions that follow. Assume the multiplexers select the bottom input signal when the control signal is high. We have the following:

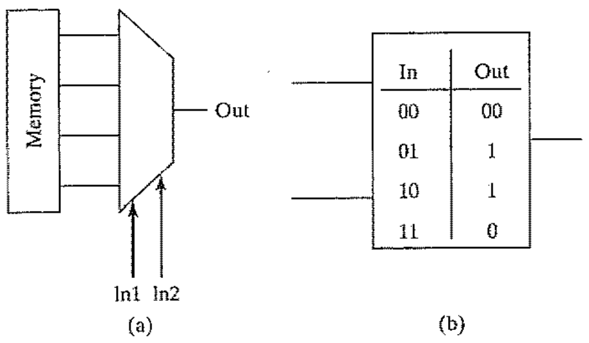
$$A = 1;\ B = 0;\ C = 0;\ D = 1;\ SA = SB = In1;\ S0 = S1 = In2$$

As an exercise, determine the programming required for the two-input XNOR function. A three-input AND gate can be realized as follows:

$$A = 0;\ B = In1;\ C = 0;\ D = 0;\ SA = In2;\ SB = 0;\ S0 = S1 = In3$$

Finally, the largest function that can be realized is the four-input multiplexer. A, B, C, and D act as inputs, while SA, SB, and $(S0 + S1)$ are control signals.

The "multiplexer-as-functional-block" approach provides configurability through programmable interconnections. The *lookup table* (LUT) method employs a vastly different strategy. To configure a fully programmable module with fan-in of i for a specific function, a two-bit large memory, called the lookup table, is programmed to capture the truth table of that function. The input variables serve as control inputs to a multiplexer, which picks the appropriate value from the memory. The idea is illustrated in Figure 8-33 for a two-input cell. To implement an EXOR function, the lookup table is loaded with the output column of the EXOR truth table, this is "0 1 1 0". For an input value of "0 0", the multiplexer selects the first value in the table ("0"), etc. With this approach, any logic function of two inputs can be realized by a simple (re)programming of the memory.



In	Out
00	00
01	1
10	1
11	0

Figure 8-33 Configurable logic cell based on lookup table. (a) cell schematic; (b) programming the cell to implement an EXOR function.

As in the case of the multiplexer-based approach, more complex gates can be constructed. This is accomplished by either combining a number of LUTs, or by increasing the LUT sizes, or a combination of both. Additional functionality is provided by incorporating flip-flops.

Example 8.10 LUT-Based Programmable Logic Cell

Figure 8-34 shows the basic cell, called a *Configurable Logic Block* or CLB, used in the Xilinx 4000 FPGA series [Xilinx4000]. It combines two four-input LUTs feeding a three-

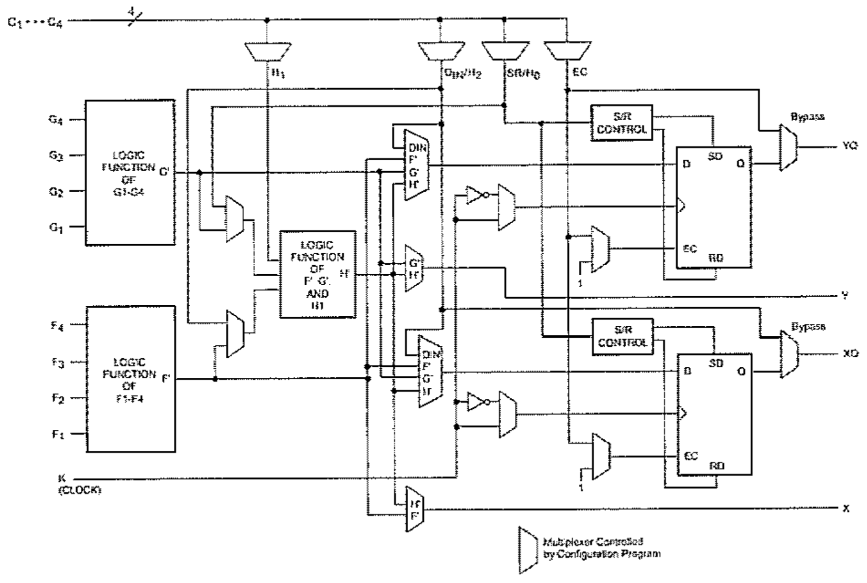


Figure 8-34 Simplified block diagram of XC4000 Series CLB (RAM and Carry-logic functions not shown) [Xilinx4000].

input LUT. The cell features two flip-flops, whose inputs can be any one of the LUT outputs F, G, or H, or an extra external input D_{in}, and whose outputs are available at the XQ and YQ output pins. The X and Y outputs export the outputs of the LUTs and make it possible to build more complex combinational functions. The cell has four extra inputs ($C1 \ldots C4$) that either can be used as inputs or as set/reset and clock-enable signals for the flip-flops.

Programmable Interconnect

So far, we have discussed in some depth how to make logic programmable. A compelling question is how to make interconnections between those gates changeable or programmable as well. To fully utilize the available logic cells, the interconnect network must be flexible and routing bottlenecks must be avoided. Speed is another prerequisite, since interconnect delay tends to dominate the performance in this style of design. At the same time, the reader should be aware that programmable interconnect comes at a substantial cost in performance in area, performance, and power. In fact, most of the power dissipation in field-programmable architectures is attributable to the interconnect network [George01].

Once again, we can differentiate between mask-programmable, one-time programmable and reprogrammable approaches. It also is worth differentiating between local cell-to-cell interconnections and global signals, such as clocks, that have to be distributed over the complete chip with low delay. In the local-area class, programmable wiring can be classified into two major groupings: array and switchbox routers.

Array-Based Programmable Wiring In this approach, wiring is grouped into routing channels, each of which contains a complete grid of horizontal and vertical wires. An interconnect wire can then be programmed into the structure by short-circuiting some of the intersections between horizontal and vertical wires (see Figure 8-35). This can be accomplished by providing a pass transistor at each of the cross points. Closing the interconnection means raising the control signal—by programming a "1" into the connected memory cell M (see Figure 8-36). This approach is prohibitive and expensive because it requires a large number of transistors and control signals. Also, the large number of transistors connected to each wire leads to a high fan-out, translating into delay and power consumption. A fuse is a more effective programmable connector. In this approach, each routing channel as a fully connected grid of horizontal and vertical interconnect wires, and a fuse is blown whenever a connection is not needed. Unfortunately, interconnect networks tend to be sparsely populated, which requires the interruption of an excessive number of switches and results in prohibitively long programming times.

To circumvent this problem, an *antifuse* can be used (as in Figure 8-26). Antifuses only need to be enabled when a connection is required in the routing channel. This represents a small fraction of the overall grid. Notice in Figure 8-35 how only two antifuses are needed to set up a connection. Be aware that this figure hides the programming circuitry. This operation is a one-time event and cannot be undone. The array-based wiring approach has thus been most successful in the write-once class of FPGAs. Circuit corrections or extensions are not possible, and new

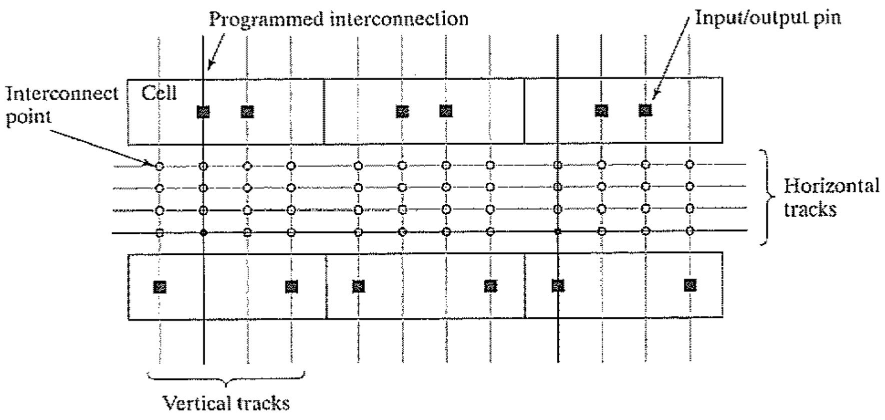


Figure 8-35 Array-based programmable wiring.

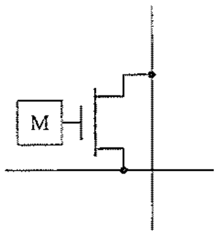


Figure 8-36 Programmable interconnect point. The memory cell controls the interconnection. A stored 0 and 1 mean an open or a closed circuit, respectively. The memory cell can be nonvolatile (EEPROM) or volatile (SRAM).

components are required for every design change. Providing true field (re)programmability requires a more efficient routing strategy.

Switch-Box-Based Programmable Wiring It's easy to imagine more efficient programmable-routing approach once we realize that the fully connected wiring grid represents major overkill. By restricting the number of routing resources and interconnect points, we can still manage to wire the desired interconnections, while drastically reducing the overhead. The disadvantage of this approach is that occasionally an interconnection cannot be routed. Most often, this can be addressed by remapping the design—for instance, by choosing another group of logic cells for a given function.

A large number of local interconnections can be accounted for by providing a mesh-like interconnection between neighboring cells. For instance, the outputs of each logic cell (LC) can be distributed to its neighbors to the north, east, south, and west. To account for interconnections between disjoint cells or to provide global interconnections, routing channels are placed between the cells containing a fixed number of uncommitted vertical and horizontal routing wires (Figure 8-37). At the junctions of the horizontal and vertical wires, RAM-programmable switching matrices (S-boxes) are

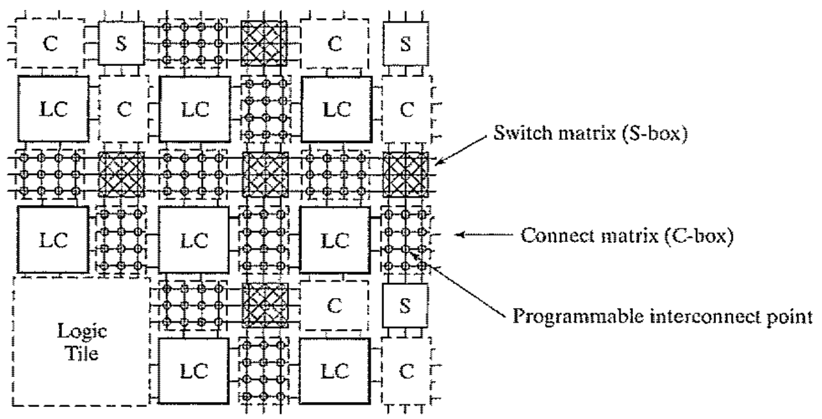


Figure 8-37 Programmable mesh-based interconnect network (Courtesy Andre Dehon and John Wawrzyniek.).

provided that direct the routing of the data. Cell inputs and outputs are connected to the global interconnect network by RAM-programmable interconnect points (C-box). Figure 8-38 provides a more detailed view, showing the transistor implementation of the switch and interconnect boxes. Be aware that the single pass-transistor implementation of the switches comes with a threshold-voltage drop. While advantageous from a power perspective, this reduced signal swing has a negative impact on the performance. Special design techniques such as zero-threshold devices, level restorers, or boosted control signals might be required.

The mesh architecture provides a flexible and scalable means for connecting a large number of components. It is quite efficient for local connections, as the number of switches traversed by a single interconnection is small and the fan-out is small. However, the mesh network does not lend itself well to global interconnections. The delay caused by the combination of the many

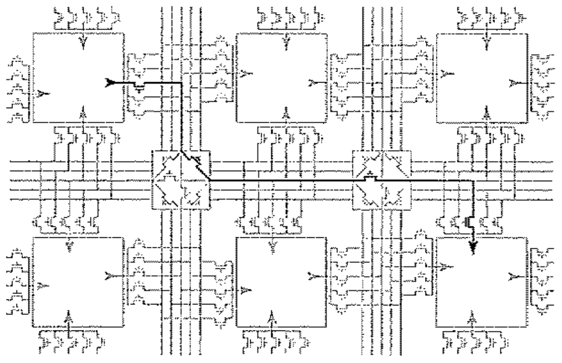


Figure 8-38 Transistor-level schematic diagram of mesh-based programmable routing network (Courtesy Andre Dehon and John Wawrzyniek.).

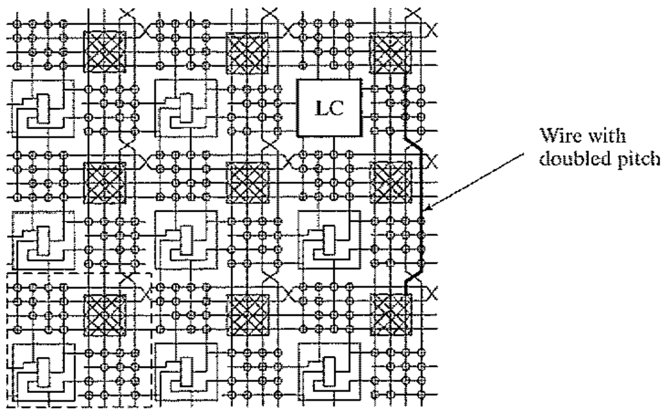


Figure 8-39 Programmable mesh-based interconnect architecture with overlaid 2 x 2 grid (Courtesy Andre Dehon and John Wawrzyniek.).

switches and the large capacitive load becomes excessive. Most mesh-based FPGA architectures therefore offer alternative wiring resources that allow for effective global wiring. One approach for accomplishing this task is shown in Figure 8-39. In addition to the standard S-box-to-S-box wiring, the network also includes wires connecting S-boxes that are two steps away from each other. Eliminating one S-box from an interconnection decreases the resistance. Similarly, we can include long wires that connect every 4th, 8th, or 16th S-box. What we are creating, in fact, is a number of overlaying meshes with different granularity (single pitch, double pitch, etc.). Long wires are, by preference, mapped on the wiring meshes with the larger pitch.

Putting It All Together

A complete field-programmable gate array can now be assembled by joining logic-cell and interconnect approaches. Many alternative architectures can be (and have been) conceived. The most important decision to make at the start is the configuration style (write once, nonvolatile, volatile). This puts some constraints on the types of cells and interconnects that can be used. Giving a complete overview is out of the scope of this textbook, so we limit ourselves to two popular architectures, which are illustrative for the field. The interested reader can find more information in [Trimberger94], [Smith97], [Betz99], and [George01].

The Altera MAX Series [Altera01] The MAX family of devices (Figure 8-40) belongs to the class of nonvolatile FPGAs (often called EPLDs, or *Electrically Programmable Logic Devices*). It uses a PAL module, (as introduced in Figure 8-29) as the basic logic module. The module (called the *Logic Array Block* or LAB in Altera language) varies little over the members of the family: a wide programmable AND array followed by a narrow fixed OR array and programmable inversion. A LAB typically contains 16 macrocells.

The major differentiation lies in the interconnect architecture between the LABs. The smaller devices (MAX5000, MAX7000) use an array-based routing architecture. The back-

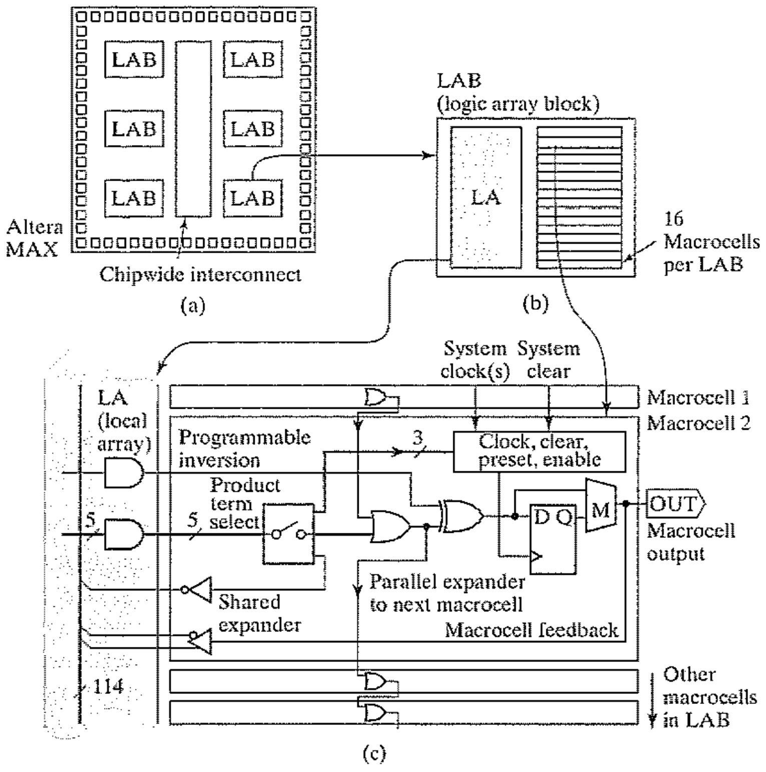


Figure 8-40 The Altera MAX Architecture. (a) Organization of logic and interconnect; (b) LAB module; (c) a MAX family macrocell. The expanders increase the number of products available by taking another pass through the logic array (from [Smith97]).

bone of the routing channel is formed by the outputs of all the macrocells, complemented with the direct chip inputs. These can be connected to the inputs of the LABs through programmable interconnect points. The advantage of this architecture, called the *Programmable Interconnect Array* or *PIA*, is that it is simple, and the routing delay between the blocks is totally predictable and fixed (see Figure 8-41). The disadvantage is that it does not scale very well. This is why the larger members of the series (MAX9000) have to resort to another scheme. With the number of macrocells reaching up to 560, the single-channel approach runs out of steam, and becomes slow. A mesh-based routing architecture has been opted for instead. Individual macrocells can connect to both row and column channels, which are quite wide (48 to 96 wires).

The EPLD approach delivers up to 15,000 logic gates, and typically is used when high performance is a necessity. Other architectures become desirable when more complex functions have to be implemented.

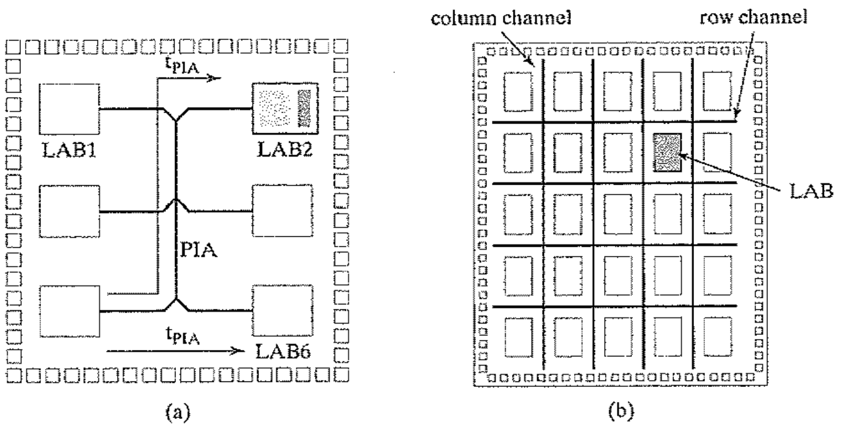


Figure 8-41 Interconnect architectures used in the Altera MAX series. (a) Array-based architecture used in MAX 3000-7000; (b) Mesh architecture of the MAX9000.

The Xilinx XC40xx Series This popular RAM-programmable device family combines the lookup table approach for the implementation of the logic cells, with a mesh-based interconnect network. The largest part in the series (XC4085) supports almost 100,000 gates using a 56 × 56 CLB array. The architecture of the CLB was shown in Figure 8-34. An interesting feature is that the CLB can also be configured as an array of Read/Write memory cells, using the memory lookup tables in the F' and G' blocks. Depending on the selected mode, a single CLB can be configured as either a 16 × 2, 32 × 1, or 16 × 1 bit array. This feature comes in handy, because it is typical for large modules of logic to need comparable amounts of storage.

The interconnect architecture is also quite rich, and combines a wide variety of wiring resources, as shown in Figure 8-42. The overlaid meshes consist of wire segments of lengths 1, 2, and 4. Some components also support direct connections, which link adjacent CLBs without using general wiring resources. Signals routed on the direct interconnect experience minimum wiring delay, as the fan-out is small. These *Directs* are especially effective in the implementation of fast arithmetic modules, which feature many critical local connections. To address global wiring, *long lines* are provided that form a grid of metal interconnect segments that run the entire length or width of the array. These are intended for high fan-out, time-critical signal nets, or nets that are distributed over long distances (such as buses). In addition, special wires are provided for the routing of the clocks.

One topic we have ignored so far in our discussion of configurable array structures is the input/output architecture. For maximum usability, it is crucial that the I/O pins of the component are flexible, and that they provide a wide range of options in terms of direction, logic levels, and drive strengths. One style of input/output block (IOB), used in the XC4000 series, is shown in Figure 8-43. It can be programmed to act as an input, output, or bidirectional port. It includes a flip-flop that can be programmed to be either edge triggered or level sensitive. The slew-rate

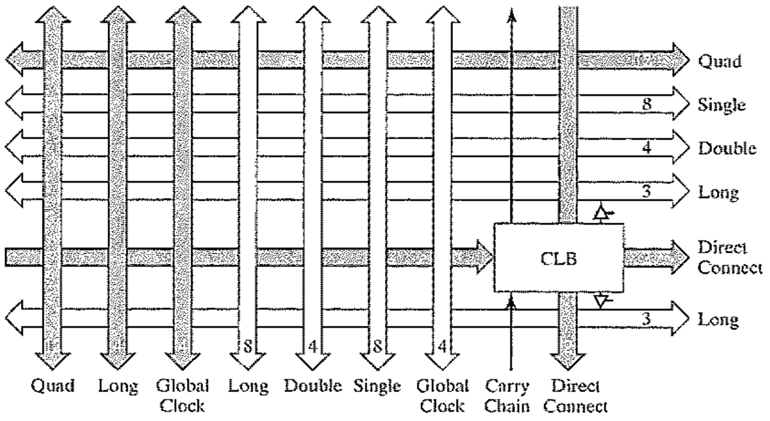


Figure 8-42 Interconnect architecture of the Xilinx XC4000 series. The numbers annotated on the diagram indicate the amount of each of the resources available.

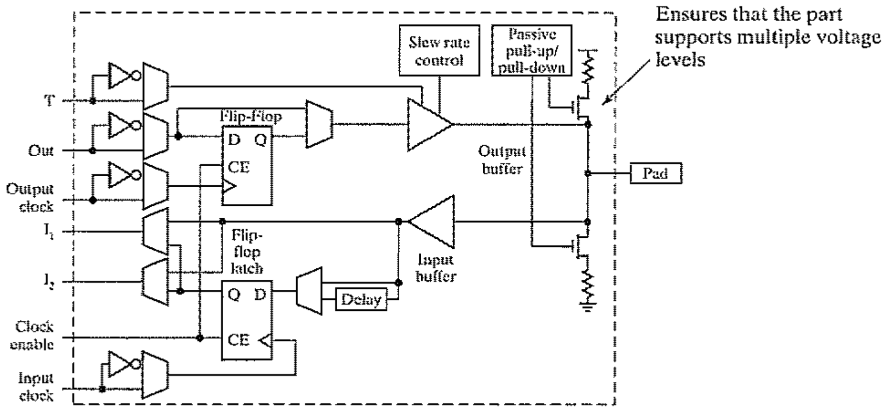


Figure 8-43 Programmable input/output Block of XC4000 series.

control provides variable drive strengths and allows for a reduction in the rise–fall time for non-critical signals.

Example 8.11 FPGA Complexity and Performance

To get an impression of what can be achieved with the volatile field-programmable components, consider the Xilinx 4025. It contains approximately 1000 CLBs organized in a 32×32 array. This translates into a maximum equivalent gate count of 25,000 gates. The chip contains 422 Kbits of RAM, used mostly for programming. A single CLB is specified to operate at 250 MHz. When taking into account the interconnect network and attempting more complex logic configurations such as adders, clock speeds between 20 and 50 MHz

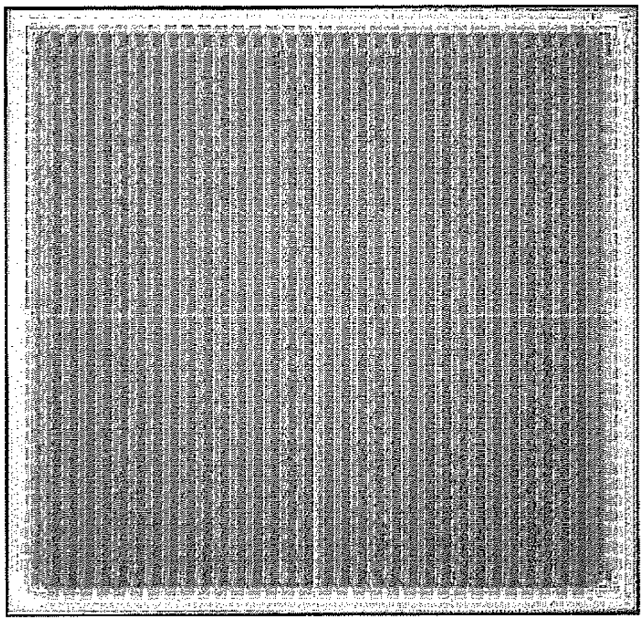


Figure 8-44 Chip microphotograph of XC4025 volatile FPGA (Courtesy of Xilinx, Inc.).

are attainable. To put the integration complexity in perspective, a 32-bit adder requires approximately 62 CLBs. A chip microphotograph of the XC4025 part is shown in Figure 8-44. The horizontal and vertical routing channels are easily recognizable.

Prewired logic arrays have rapidly claimed a significant part of the logic component market. Their arrival has effectively ended the era of logic design using discrete components represented by the TTL logic family. It is generally believed that the impact of these components is increasing with a further scaling of the technology. To make this approach successful, however, advanced software support in terms of cell placement, signal routing, and synthesis are required. Also, one should not ignore the overhead that flexibility brings with it. Programmable logic is at least 10 times less efficient in terms of energy and performance with respect to ASIC solutions. Hence, its scope has been mostly restricted to prototyping and small-volume applications so far. Yet, flexibility and reuse are alluring. Field-programmable components are bound to see a substantial growth in the years to come.

8.6 Perspective—The Implementation Platform of the Future

The designer of today's advanced systems-on-a-chip is offered a broad range of implementation choices. What approach is ultimately chosen is determined by a broad range of factors:

- performance, power and cost constraints
- design complexity
- testability
- time to market, or more precisely, time to revenue
- uncertainty of the market, or late changes in the design
- application range to be covered by the design
- prior experiences of the design team

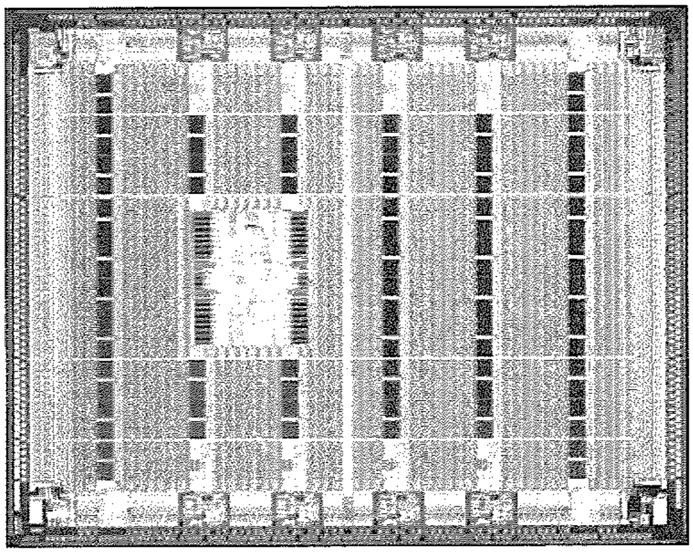
A number of these factors seem to imply a trend towards more flexible, programmable components that can be reused and that can be modified even after manufacturing. At the same time, solutions that offer the best "bang for the buck" most often end up the winners. Too much flexibility often results in ineffective and expensive solutions, which rapidly end up on the dust heap. Finding the balance between the two extremes is the ultimate challenge of the chip architects of today.

On the basis of these observations, it seems logical to assume that the implementation platform of the future will be a combination of the strategies we have discussed in this chapter, providing implementation efficiency and flexibility when and where needed. The system on a chip is becoming a combination of embedded microprocessors with their memory subsystems, DSPs, fixed ASIC-style hardware accelerators, parameterizable modules, and flexible logic implemented in FPGA style. How these components are balanced is a function of the application requirements and the intended market.

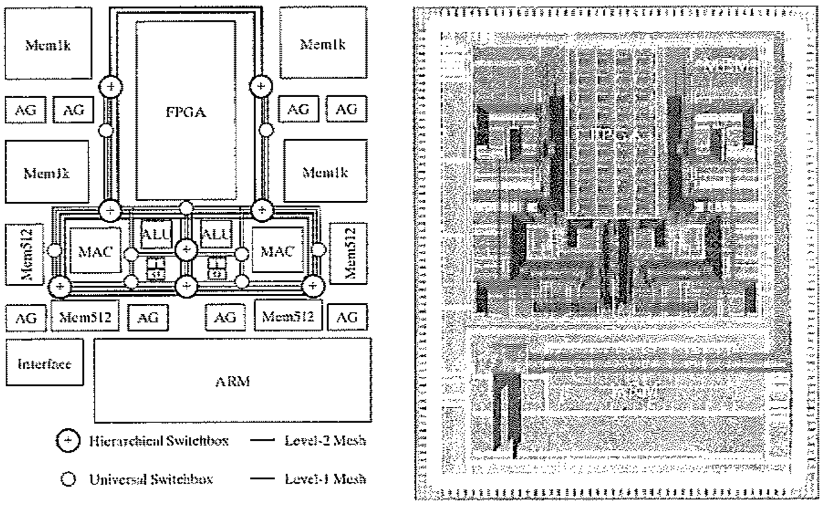
Example 8.12 Examples of Hybrid Implementation Platforms

Figure 8-45 shows two contrasting implementation platforms for wireless applications. The first device, the Virtex-II Pro from Xilinx [XilinxVirtex01] is centered around a large FPGA array. A PowerPC microprocessor is embedded in the center of the array. The processor provides an effective implementation approach for application-level functionality and system-level control. To provide higher performance for signal processing applications, an array of embedded 18×18 multipliers is added. These dedicated components offer a significant performance, power, and area advantage over a pure FPGA implementation of the same function. Finally, a number of very fast 3.125-Gbps transceivers are provided, allowing for high-speed serial communication off chip.

A somewhat contrasting approach is offered in the design of Figure 8-45b [Zhang00]. The center of this device is an ARM-7 embedded microprocessor, acting as the overall chip manager. Functions that need high performance and energy efficiency are off-loaded to a configurable array of functional units such as multipliers, ALUs, memories, and address generators. These components can be combined dynamically into application-specific processors. The chip also provides an embedded FPGA array for functions that need bit-level granularity.



(a) The Xilinx Virtex-II Pro embeds a PowerPC microprocessor into an FPGA fabric (Courtesy Xilinx, Inc.).



(b) The Maia chip combines embedded microprocessor, configurable accelerators, and FPGA [Zhang00].

Figure 8-45 Examples of hybrid implementation platforms.

8.7 Summary

In this chapter, we have briefly scanned the complex world of design implementation strategies for digital integrated circuits. New implementation styles have rapidly emerged over the last few decades, presenting the designer with a wide variety of options. These design techniques and the accompanying tools are having a major impact on the way design is performed today, and make possible the exciting and impressive processors and application-specific circuits to which we have become so accustomed. We have touched on the following issues in this chapter:

- *Custom design*, where each transistor is individually handcrafted, offers the implementation from an area and performance perspective. This approach has become prohibitively expensive, and should be reserved for the design of the few critical modules in which extreme performance is required, or for often-reused library cells.
- The *semicustom* approach, based on the standard-cell methodology, is the workhorse of today's digital design industry. The advantage is the high degree of automation. The challenge is to deal with the impact of deep-submicron technologies.
- To deal with the increasing complexity of integrated circuits, designers increasingly rely on the availability of large *macrocells* such as memories, multipliers, and microprocessors. These modules are often provided by third-party vendors, and they have spurned a new industry focused on "*intellectual property*."
- Starting a new design for every new emerging application has become prohibitively expensive. The majority of the semiconductor market now focuses on flexible solutions that allow a single component to be used for a variety of applications, either through software programming or reconfiguration. *Configurable hardware* delays the time when the required function is actually committed to the hardware. Different approaches toward late binding also have been discussed. Delaying the binding time comes with an efficiency penalty: The more flexibility that is provided, the larger the impact on performance and power dissipation.

Undoubtedly, new design styles will come on the scene in the near future. Becoming familiar with the available options is an essential part of the learning experience of the beginning digital designer. We hope this chapter, although compressed, entices the reader to further explore the numerous possibilities. One final observation is as follows: Even with the increasing automation of the digital circuit design process, new challenges are continuously emerging—challenges that require the profound insight and intuition offered only by a human designer.

8.8 To Probe Further

The literature on design methodologies and automation for digital integrated circuits has exploded in the last few decades. Several reference works are worth mentioning:

- ASIC and FPGA design methodologies: [Smith97]
- FPGA architectures: [Trimberger94], [George01]

- System on a Chip: [Chang99]
- Design methodology and technology: [Bryant01]
- Design synthesis: [DeMicheli94]

State-of-the-art developments in the design automation domain are generally reported in the *IEEE Transactions on CAD*, the *IEEE Transactions on VLSI Systems*, and the *IEEE Design and Test Magazine*. Premier conferences are, among others, the Design Automation Conference (DAC) and the International Conference on CAD (ICCAD). The web sites of the major Electronic Design Automation Companies (Cadence, Synopsys, Mentor, etc.) provide a treasure of information as well.

References

[Altera01] Altera Device Index, *http://www.altera.com/products/devices/dev-index.html*, 2001.

[Avanti01] Saturn Efficient and Concurrent Logical and Physical Optimization of SoC Timing, Area and Power, *http://www.synopsis.com/product/avmrg/saturn_ds.html.*

[Betz99] V. Betz, J. Rose, and A. Marquardt, *Architecture and CAD for Deep-Submicron FPGAs*, Kluwer International Series in Engineering and Computer Science, Kluwer Academic Publishers, 1999.

[Brayton84] R. Brayton et al., *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publishers, 1984.

[Bryant01] R. Bryant, T. Cheng, A. Kahng, K. Keutzer, W. Maly, R. Newton, L. Pileggi, J. Rabaey, and A. Sangiovanni-Vincentelli, "Limitations and Challenges of Computer-Aided Design Technology for CMOS VLSI," *IEEE Proceedings*, pp. 341–365, March 2001.

[Cadabra01] AbraCAD Automated Layout Creation, *http://www.cadabratech.com/?id=145products*, Cadabra Design Automation.

[Chang99] H. Chang et al., "Surviving the SOC Revolution: A Guide to Platform-Based Design," Kluwer Academic Publishers, 1999.

[DeMicheli94] G. De Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, 1994.

[El-Ayat89] K. El-Ayat, "A CMOS Electrically Configurable Gate Array," *IEEE Journal of Solid State Circuits*, vol. SC-24, no. 3, pp. 752–762, June 1989.

[George01] V. George and J. Rabaey, *Low-Energy FPGAs*, Kluwer Academic Publishers, 2001.

[Grundman97] W. Grundmann, D. Dobberpuhl, R. Allmon, and N. Rethman "Designing High-Performance CMOS Processors Using Full Custom Techniques," *Proceedings Design Automation Conference*, pp. 722–727, Anaheim, June 1997.

[Hill85] D. Hill, "S2C—A Hybrid Automatic Layout System," *Proc. ICCAD-85*, pp. 172–174, November 1985.

[ModuleCompiler01] Synopsys Module Compiler Datasheet, *http://www.synopsys.com/products/datapath/datapath.html*, Synopsys, Inc.

[Philips99] The Nexperia System Silicon Implementation Platform, *http://www.semiconductors.philips.com/platforms/nexperia/*, Philips Semiconductors.

[Pileggi02] Pileggi, Schmit et al., "*Via Patterned Gate Array*," CMU Center for Silicon System Implementation Technical Report Series, no. CSSI 02-15, April 2002.

[Prolific01] The ProGenensis Cell Compiler, *http://www.prolificinc.com/progenesis.html*, Prolific, Inc.

[Rabaey00] J. Rabaey, "Low-Power Silicon Architectures for Wireless Applications," *Proceedings ASPDAC Conference*, Yokohama, January 2000.

[Silva01] J. L. da Silva Jr., J. Shamberger, M. J. Ammer, C. Guo, S. Li, R. Shah, T. Tuan, M. Sheets, J. M. Rabaey, B. Nikolic, A. Sangiovanni-Vincentelli, P. Wright, "Design Methodology for PicoRadio Networks," *Proc. DATE Conference*, Munich, March 2000.

[Smith97] M. Smith, *Application-Specific Integrated Circuits*, Addison-Wesley, 1997.

[Sylvester98] D. Sylvester and K. Keutzer, "Getting to the Bottom of Deep Submicron," *Proc. ICCAD Conference*, pp. 203, San Jose, November 1998.

[Trimberger94] S. Trimberger, *Field-Programmable Gate Array Technology*, Kluwer Academic Publishers, 1994.

[Veendrick92] H. Veendrick, *MOS IC's: From Basics to ASICS*, Wiley-VCH, 1992.

[Xilinx4000] The Xilinx-4000 Product Series, *http://www.xilinx.com/apps/4000.htm*, Xilinx, Inc.

[XilinxVirtex01] Virtex-II Pro Platform FPGAs, *http://www.xilinx.com/xlnx/xil_prodcat_landing page.jsp?title=Virtex-II+Pro+FPGAs*, Xilinx, Inc.

[Xtensa01] Xtensa Configurable Embedded Processor Core, *http://www.tensilica.com/technology.html*, Tensilica.

[Zhang00] H. Zhang, V. Prabhu, V. George, M. Wan, M. Benes, A. Abnous, and J. Rabaey, "A 1 V Heterogeneous Reconfigurable Processor IC for Baseband Wireless Applications," *Proc. ISSCC*, pp. 68–69, February 2000.

Exercises

For problems and exercises on design methodology, please check **http://bwrc.eecs.berkeley.edu/IcBook**.

DESIGN METHODOLOGY INSERT

E

Characterizing Logic and Sequential Cells

The challenge of library characterization

Characterization methods for logic cells and registers

Cell parameters

The Importance and Challenge of Library Characterization

The quality of the results produced by a logic synthesis tool is a strong function of the level of detail and accuracy with which the individual cells were characterized. To estimate the delay of a complex module, a logic synthesis program must rely on higher level delay models of the individual cells—falling back to a full circuit- or switch-level timing model for each delay estimation is simply not possible because it takes too much compute time. Hence, an important component in the development process of a standard-cell library is the generation of the delay models. In previous chapters, we learned that the delay of a complex gate is a function of the fan-out (consisting of connected gates and wires), and the rise and fall times of the input signals. Furthermore, the delay of a cell can vary between manufacturing runs as a result of process variations.

In this insert, we first discuss the models and characterization methods that are commonly used for logic cells. Sequential registers require extra timing parameters and thus deserve a separate discussion.

Characterization of Logic Cells

Unfortunately, no common delay model for standard cells has been adopted. Every vendor has his own favored methods of cell characterization. Even within a single tool, various delay models often can be used, trading off accuracy for performance. The basic concepts are, however, quite similar, and they are closely related to the ones we introduced in Chapters 5 and 6. We therefore opt to concentrate on a single set of models in this section—more precisely, those used in the Synopsys Design Compiler [DesignCompiler01], one of the most popular synthesis tools. Once a model has been adopted, it has to be adopted for all the cells in the block; in other words, it cannot be changed from cell to cell.

The total delay consists of four components, as illustrated in Figure E-1:

$$D_{total} = D_I + D_T + D_S + D_C. \tag{E.1}$$

D_I represents the *intrinsic delay*, which is the delay with no output loading. D_T is the transition component, or the part of the delay caused by the output load. D_S is the fraction of the delay due to the *input slope*. Finally, D_C is the *delay of the wire* following the gate. All delays are characterized for both rising and falling transitions.

The simplest model for the transition delay is the linear delay model of Chapter 5. We have

$$D_T = R_{driver}(\Sigma C_{gate} + C_{wire}), \tag{E.2}$$

where ΣC_{gate} is the sum of all input pin capacitances of gates connected to the output of this gate, and C_{wire} is the estimated wire capacitance. The slope delay D_S is approximated as a linear function of the transition delay D_T of the previous gate, written as

$$D_S = S_S\, D_{Tprev} \tag{E.3}$$

where S_S is the *slope-sensitivity factor*, and D_{Tprev} is the transition delay of the previous stage.

The characterization of a library cell must therefore provide the following components, each of them for both rising and falling transitions, and with respect to each of the input pins:

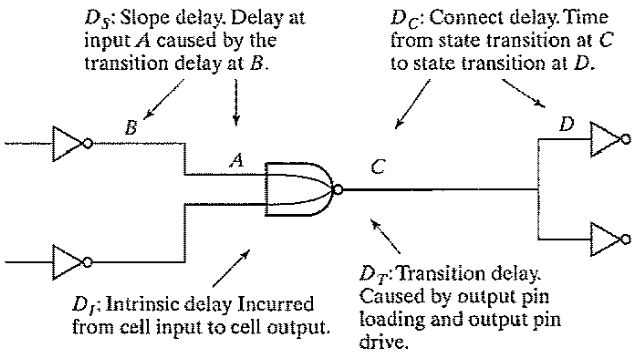

D_S: Slope delay. Delay at input A caused by the transition delay at B.
D_C: Connect delay. Time from state transition at C to state transition at D.
B
A
C
D
D_I: Intrinsic delay Incurred from cell input to cell output.
D_T: Transition delay. Caused by output pin loading and output pin drive.

Figure E-1 Delay components of a combinational gate [DesignCompiler01].

- intrinsic delay
- input pin capacitance
- equivalent output driving resistance
- slope sensitivity

In addition to the cell models, the synthesis tools also must have access to a wire model. Since the length of the wires is unknown before the placement of the cells, estimates of C_{wire} and R_{wire} are made on the basis of the size of the block and the fan-out of the gate. The length of a wire is most often proportional to the number of destinations it has to connect.

Example E.1 Three-Input NAND Gate Cell

The characterization of the three-input NAND standard cell gate, presented earlier in Example 8.2, is given in Table E-1. The table characterizes the performance of the cell as a function of the load capacitance and the input-rise (fall) time for two different supply voltages and operating temperatures. The cell is designed in a 0.18-µm CMOS technology.

Table E-1 Delay characterization of a three-input NAND gate (in ns) as a function of the input node for two operation corners (supply-voltage–temperature pairs of 1.2 V–125°C, and 1.6 V–40°C). The parameters are the load capacitance *C* and the input rise (fall) time *T*. (*Courtesy ST Microelectronics.*)

Path	1.2 V–125°C	1.6 V–40°C
$In1—t_{pLH}$	$0.073 + 7.98C + 0.317T$	$0.020 + 2.73C + 0.253T$
$In1—t_{pHL}$	$0.069 + 8.43C + 0.364T$	$0.018 + 2.14C + 0.292T$
$In2—t_{pLH}$	$0.101 + 7.97C + 0.318T$	$0.026 + 2.38C + 0.255T$
$In2—t_{pHL}$	$0.097 + 8.42C + 0.325T$	$0.023 + 2.14C + 0.269T$
$In3—t_{pLH}$	$0.120 + 8.00C + 0.318T$	$0.031 + 2.37C + 0.258T$
$In3—t_{pHL}$	$0.110 + 8.41C + 0.280T$	$0.027 + 2.15C + 0.223T$

While linear delay models offer good first-order estimates, more precise models are often used in synthesis, especially when the real wire lengths are back annotated onto the design. Under those circumstances, nonlinear models have to be adopted. The most common approach is to capture the nonlinear relations as lookup tables for each of these parameters. To increase computational efficiency and minimize storage and characterization requirements, only a limited set of loads and slopes are captured, and linear interpolation is used to determine the missing values.

Example E.2 Delay Models Using Lookup Tables

A (partial) characterization of a two-input AND cell (AND2), designed in a 0.25-μm CMOS technology (*Courtesy ST Microelectronics*) follows. The delays are captured for output capacitances of 7 fF, 35 fF, 70 fF, and 140 fF, and input slopes of 40 ps, 200 ps, 800 ps, and 1.6 ns, respectively.

```
cell(AND) {
 area : 36 ;
 pin(Z) {
  direction : output ;
  function : "A*B";
  max_capacitance : 0.14000 ;

  timing() {
   related_pin : "A" ; /* delay between input pin A and output pin Z */
cell_rise {
    values( "0.10810, 0.17304, 0.24763, 0.39554", \
        "0.14881, 0.21326, 0.28778, 0.43607", \
        "0.25149, 0.31643, 0.39060, 0.53805", \
        "0.35255, 0.42044, 0.49596, 0.64469" ); }
   rise_transition {
    values( "0.08068, 0.23844, 0.43925, 0.84497", \
        "0.08447, 0.24008, 0.43926, 0.84814", \
       "0.10291, 0.25230, 0.44753, 0.85182", \
        "0.12614, 0.27258, 0.46551, 0.86338" );}
   cell_fall(table_1) {
    values( "0.11655, 0.18476, 0.26212, 0.41496", \
        "0.15270, 0.22015, 0.29735, 0.45039", \
        "0.25893, 0.32845, 0.40535, 0.55701", \
        "0.36788, 0.44198, 0.52075, 0.67283" );}
   fall_transition(table_1) {
    values( "0.06850, 0.18148, 0.32692, 0.62442", \
        "0.07183, 0.18247, 0.32693, 0.62443", \
        "0.09608, 0.19935, 0.33744, 0.62677", \
        "0.12424, 0.22408, 0.35705, 0.63818" );}
   intrinsic_rise : 0.13305 ; /* unloaded delays */
   intrinsic_fall : 0.13536 ;
  }
 timing() {
   related_pin : "B" ; /* delay between input pin A and output pin Z */
   ...
intrinsic_rise : 0.12426 ;
   intrinsic_fall : 0.14802 ;
  }
 }
 pin(A) {
  direction : input ;
  capacitance : 0.00485 ; /* gate capacitance */
 }
 pin(B) {
  direction : input ;
  capacitance : 0.00519 ;
 }
}
```

Characterization of Registers

In Chapter 7, we identified the three important timing parameters of a register. The *setup time* (t_{su}) is the time that the data inputs (D input) must be valid before the clock transition (in other words, the 0 to 1 transition for a *positive edge-triggered* register). The *hold time* (t_{hold}) is the time the data input must remain valid after the clock edge. Finally, the propagation delay (t_{c-q}) equals the time it takes for the data to be copied to the Q output after a clock event. The latter parameter is illustrated in Figure E-2a.

Latches have a bit more complex behavior, and thus require an extra timing parameter. While t_{C-Q}, corresponds to the delay of relaunching of data that arrived to a closed latch, t_{D-Q} equals the delay between D and Q terminals when the latch is in transparent mode (Figure E-2b).

The characterization of the t_{C-Q} (t_{D-Q}) delay is fairly straightforward. It consists of a delay measurement between the 50% transitions of *Clk* (D) and Q, for different values of the input slopes and the output loads, not unlike the case of combinational logic cells.

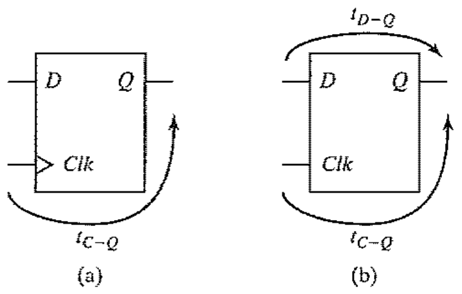


Figure E-2 Propagation delay definitions for sequential components: (a) register; (b) latch.

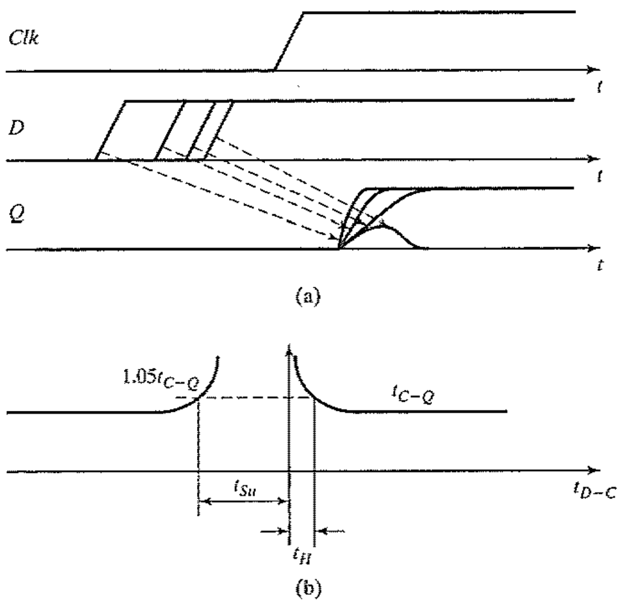


Figure E-3 Characterization of sequential elements: (a) determining the setup time of a register; (b) definition of setup and hold times.

The characterization of setup and hold times is more elaborate, and depends on what is defined as "valid" in the definitions of both setup and hold times. Consider the case of the setup time. Narrowing the time interval between the arrival of the data at the *D* input and the *Clk* event does not lead to instantaneous failure (as assumed in the first-order analysis in Chapter 7), but rather to a gradual degradation in the delay of the register. This is documented in Figure E-3a, which illustrates the behavior of a register when the data arrives close to the setup time. If *D* changes long before the clock edge, the t_{C-Q} delay has a constant value. Moving the data transition closer to the clock edge causes t_{C-Q} to increase. Finally, if the data changes too close to the clock edge, the register fails to register the transition altogether.

Clearly, a more precise definition of the "setup time" concept is necessary. An unambiguous specification can be obtained by plotting the t_{C-Q} delay against the data-to-clock offset, as shown in Figure E-3b. The degradation of the delay for smaller values of the offset can be observed. The actual definition of the setup time is rather precarious. If it were defined as the minimum *D-Clk* offset that causes the flip-flop to fail, the logic following the register would suffer from excessive delay if the offset is close to, but larger than, that point of failure. Another option is to place it at the operation point of the register that minimizes the sum of the data-clock offset and the t_{C-Q} delay. This point, which minimizes the overall flip-flop overhead, is reached when the slope of the delay curve in Figure E-3b equals 45 degrees [Stojanovic99].

While custom design can take advantage of driving flip-flops close to their point of failure—and take all the risk that comes with it—semicustom design must take a more conservative approach. For the characterization of registers in a standard cell library, both setup and hold times are commonly defined as data-clock offsets that correspond to **some fixed percentage increase in** t_{C-Q}, typically set **at 5%**, as indicated in Figure E-3b. Note that these curves are different for 0–1 and 1–0 transitions, resulting in different setup (an hold) times for 0 and 1 values. As with clock-to-output delays, setup times also are dependent on clock and data slopes, and they are represented as a two-dimensional table in nonlinear delay models. Identical definitions hold for latches.

Example E.3 Register Setup and Hold Times

In this example, we examine setup and hold behavior of the transmission gate master-slave register introduced in Chapter 7. (See Figure 7.18.) The register is loaded with a 100-fF capacitor, and its setup and hold times are examined for clock and data slopes of 100 ps. The simulation results are plotted in Figure E-4. When data settles a "long time" before the clock edge, the clock-to-output delay equals 193 ps. Moving the data transition closer to the clock edge causes the t_{C-Q} delay to increase. This becomes noticeable at an offset between data and clock of about 150 ps. The register completely fails to latch the data when data precedes the clock by 77 ps. The sum of D-Q offset and the $t_{C\text{-}Q}$ is minimal at 93 ps. A 5% increase in t_{C-Q} is observed at 125 ps, and this time is entered in the library as the setup time for this particular slope of data and clock. This characterization of setup time adds a margin to the design of about 30 ps. From these simulations, we also can determine that this register has a hold time of –15 ps.

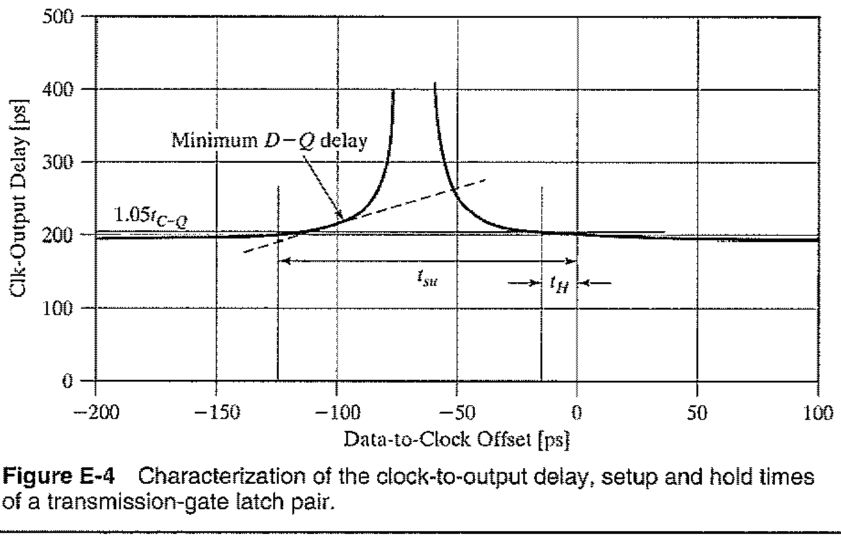


Figure E-4 Characterization of the clock-to-output delay, setup and hold times of a transmission-gate latch pair.

References

[DesignCompiler01] Design Compiler, Product Information, *http://www.synopsys.com/products/logic/design_compiler.html*, Synopsys, Inc.

[Stojanovic99] V. Stojanovic, V.G. Oklobdzija, "Comparative analysis of master-slave latches and flip-flops for high-performance and low-power systems," *IEEE Journal of Solid-State Circuits*, vol. 34, no. 4, April 1999.

DESIGN METHODOLOGY INSERT

F

Design Synthesis

Circuit, Logic, and Architectural Synthesis

One of the most enticing proposals one can make to a designer who has to generate a circuit with tough specifications in a short time is to offer him a tool that automatically translates his specifications into a working circuit that meets all the requirements. One of the main reasons that semiconductor circuits have reached the mind-boggling complexity they have today, is that such synthesis tools actually exist—at least to a certain extent. Synthesis can be defined as the transformation between two different design views. Typically, it represents a translation from a *behavioral* specification of a design entity into a *structural* description. In simple terms, it translates a description of the function a module should perform (the behavior) into a composition—that is, an interconnection of elements (the structure). Synthesis approaches can be defined at each level of abstraction: circuit, logic, and architecture. An overview of the various synthesis levels and their impact is given in Figure F-1. The synthesis procedures may differ depending on the targeted implementation style. For example, logic synthesis translates a logic description given by a set of Boolean equations into an interconnection of gates. The techniques involved in this process strongly depend on the choice of either a two-level (PLA) or a multilevel (standard-cell or gate-array) implementation style. We briefly describe the synthesis tasks at each of the different modeling levels. Refer to [DeMicheli94] for more information and a deeper insight into design synthesis.

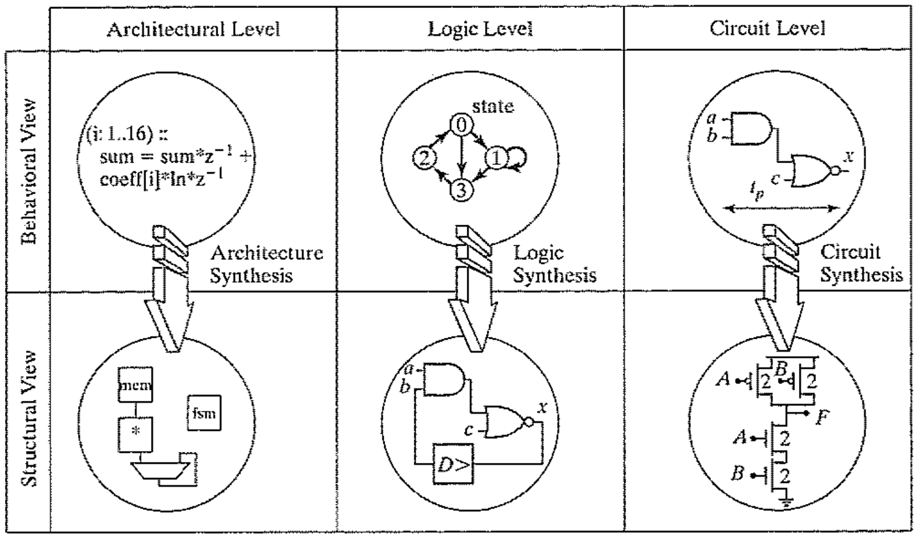


Figure F-1 A taxonomy of synthesis tasks.

Circuit Synthesis

The task of circuit synthesis is to translate a logic description of a circuit into a network of transistors that meets a set of timing constraints. This process can be divided into two stages:

1. Derivation of *transistor schematics* from the logic equations. This requires the selection of a circuit style (complementary static, pass transistors, dynamic, DCVSL, etc.) and the construction of the logic network. The former task is usually up to the designer, while the latter depends upon the chosen style. For instance, the logic graph technique introduced in Design Methodology Insert D can be used to derive the complementary pull-down and pull-up networks of a static CMOS gate. Similarly, automated techniques have been developed to generate the pull-down trees for the DCVSL logic style so that the number of required transistors is minimized [Chu86].
2. *Transistor sizing* to meet performance constraints. This has been a recurring subject throughout this book. The choice of the transistor dimensions has a major impact on the area, performance, and power dissipation of a circuit. We have also learned that this is a subtle process. For instance, the performance of a gate is sensitive to a number of layout parasitics, such as the size of the diffusion area, fan-out, and wiring capacitances. Notwithstanding these daunting challenges, some powerful transistor-sizing tools have been developed [e.g., Fishburn85, AMPS99, Northrop01]. The key to the success of these tools is the accurate modeling of the performance of the circuit using *RC* equivalent circuits and a detailed knowledge of the subsequent layout-generation process. The latter allows for an accurate estimation of the values of the parasitic capacitances.

While circuit synthesis has proven to be a powerful tool, it has not penetrated the design world as much as we might expect. One of the main reasons for this is that the quality of the cell library has a strong influence on the complete design, and designers are reluctant to pass this important task to automatic tools that might produce inferior results. Yet, the need for ever-larger libraries and the impact of transistor-sizing on circuit performance and power dissipation is providing a strong push for a more pervasive introduction of circuit-synthesis tools.

Logic Synthesis

Logic synthesis is the task of generating a structural view of a logic-level model. This model can be specified in many different ways, such as state-transition diagrams, state charts, schematic diagrams, Boolean equations, truth tables, or HDL (Hardware Description Language) descriptions.

The synthesis techniques differ according to the nature of the circuit (combinational or sequential) or the intended implementation architecture (multilevel logic, PLA, or FPGA). The synthesis process consists of a sequence of optimization steps, the order and nature of which depend on the chosen cost function—area, speed, power, or a combination of these. Typically, logic optimization systems divide the task into two stages:

1. A *technology-independent phase*, where the logic is optimized using a number of Boolean or algebraic manipulation techniques.
2. A *technology-mapping phase*, which takes into account the peculiarities and properties of the intended implementation architecture. The technology-independent description resulting from the first phase is translated into a gate netlist or a PLA description.

The *two-level minimization* tools were the first logic-synthesis techniques to become widely available. The Espresso program developed at the University of California at Berkeley [Brayton84] is an example of a popular two-level minimization program. For some time, the wide availability of these tools made regular, array-based architectures like PLAs and PALs the prime choice for the implementation of random logic functions.

At the same time, the groundwork was laid for sequential or state-machine synthesis. Tasks involved include the *state minimization* that aims at reducing the number of machine states, and the *state encoding* that assigns a binary encoding to the states of a finite state machine [DeMicheli94].

The emergence of *multilevel logic synthesis* environments such as the Berkeley MIS tool [Brayton87] swung the pendulum towards the standard-cell and FPGA implementations that offer higher performance or integration density for a majority of random-logic functions.

The combination of these techniques with sequential synthesis has opened the road to complete register-transfer (RTL) synthesis environments that take as an input an HDL description (in VHDL or Verilog—see Design Methodology Insert C) of a sequential circuit and produce a gate netlist [Carlson91, Kurup97]. Saying the logic synthesis has fundamentally altered the digital circuit design landscape is by no means an understatement. It also is fair to say that the tool set that made this major paradigm change in design methodology ultimately happen is the *Design Compiler* environment from Synopsys. Even after being in place for almost two decades, Design Compiler continues to dominate the market and represents the synthesis tool of choice for the majority of the digital ASIC designers. Built around a core of Boolean optimization and technology mapping, Design Compiler incorporates advanced techniques such as timing, area and power optimization, cell-based sizing, and test insertion [Kurup97, DesignCompiler].

Example F.1 Logic Synthesis

To demonstrate the difference between two-level and multilevel logic synthesis, both approaches were applied to the following full-adder equations, which will be treated in substantial detail in Chapter 11.

$$\begin{aligned} S &= (A \oplus B) \oplus C_i \\ C_o &= A \cdot B + A \cdot C_i + B \cdot C_i \end{aligned} \tag{F.1}$$

The MIS-II logic synthesis environment was employed for both the two-level and multilevel synthesis. The minimized truth table representing the PLA implementation is shown in Table F-1. It can be verified that the resulting network corresponds to the preceding full-adder equations. The PLA counts three inputs, seven product terms, and two outputs. Observe that no product terms can be shared between the sum and carry outputs. A NOR-NOR implementation requires 26 transistors in the PLA array (17 and 9 in the OR plane and AND planes, respectively). This count does not include the input and output buffers.

Table F-1 Minimized PLA truth table for full adder. The dashes (–) mean that the corresponding input does not appear in the product term.

A	B	C_i	S	C_o
1	1	1	1	–
0	0	1	1	–
0	1	0	1	–
1	0	0	1	–
1	1	–	–	1
1	–	1	–	1
–	1	1	–	1

Figure F-2 shows the multilevel implementation as generated by MIS-II. In the technology-mapping phase, a generic standard-cell library was targeted. Implementation of the adder requires only six standard cells. This corresponds to 34 (!) transistors in a static CMOS implementation.[1] Observe the usage of complex logic gates such as EXOR and OR-AND-INVERT. For this case study, minimization of the area was selected as the prime optimization target. Other implementations can be obtained by targeting performance instead. For instance, the critical timing path from C_i to C_o can be reduced by signal reordering. This requires the designer to identify this path as the most critical, a fact that is not obvious from a simple inspection of the full-adder equations.

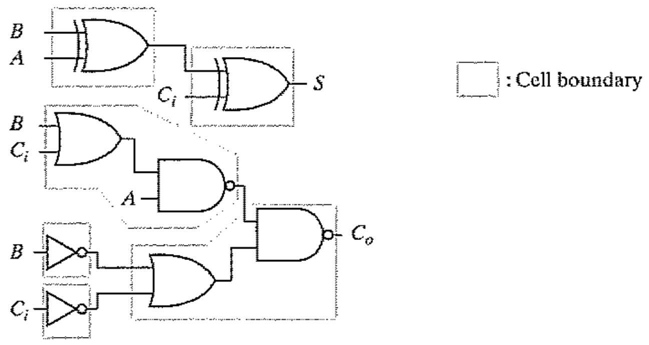


Figure F-2 Standard-cell implementation of full adder, as generated by multilevel logic synthesis.

Architecture Synthesis

Architecture synthesis is the latest development in the synthesis area. It is also referred to as *behavioral* or *high-level synthesis*. Its task is to generate a structural view of an architecture design, given a behavioral description of the task to be executed, and a set of performance, area, and/or power constraints. This corresponds to determining what architectural resources are needed to perform the task (execution units, memories, busses, and controllers), binding the behavioral operations to hardware resources, and determining the execution order of the operations on the produced architecture. In synthesis jargon, these functions are called *allocation, assignment*, and *scheduling* [Gajski92, DeMicheli94]. While these operations represent the core of architecture synthesis, other steps can have a dramatic impact on the quality of the solution. For example, optimizing transformations manipulate the initial behavioral description so that a superior solution can be obtained in terms of area or speed. *Pipelining* is a typical example of such a transformation. In a sense, this component of the synthesis process is similar to the use of optimizing transformations in software compilers.

[1] How to implement a static complementary CMOS EXOR gate with only nine transistors is left as an exercise for the reader.

Example F.2 Architecture Synthesis

To illustrate the concept and capabilities of architecture synthesis, consider the simple computational flowgraph of Figure F-3. It describes a program that inputs three numbers *a*, *b*, and *c* from off-chip and produces their sum *x* at the output.

Two possible implementations, as generated by the HYPER synthesis system ([Rabaey91]), are shown in Figure F-4. The first instance requires four clock cycles and

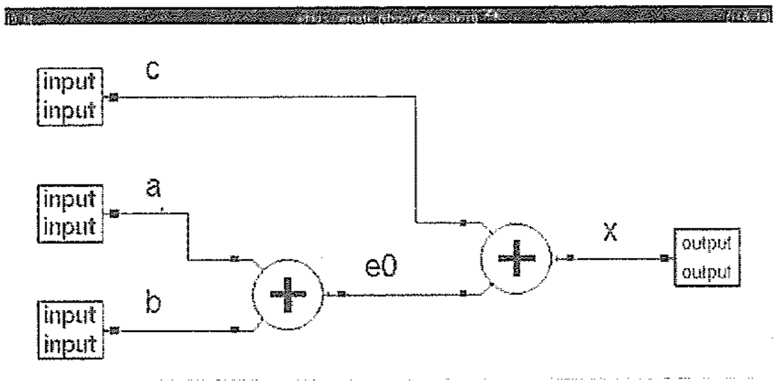
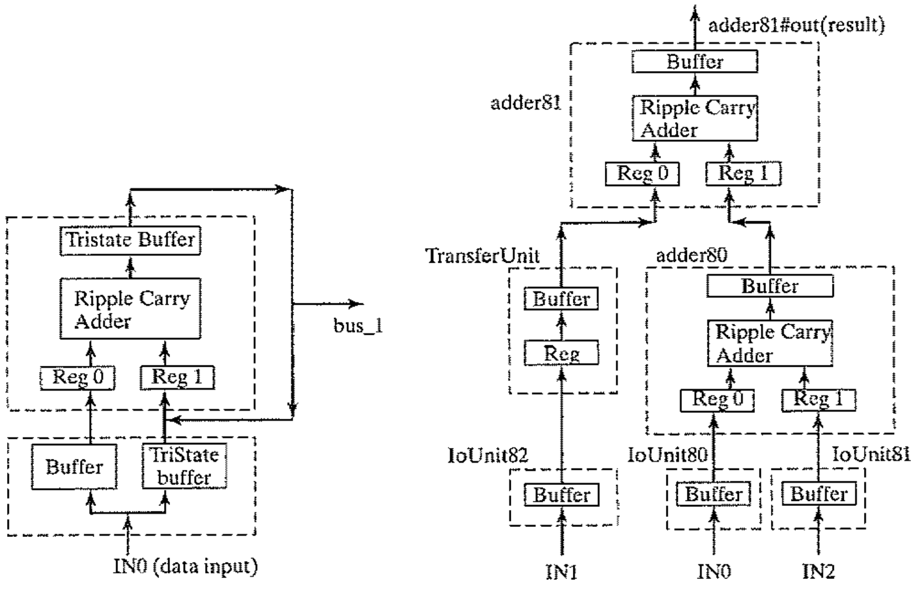


Figure F-3 Simple program performing the sum of three numbers.



(a) Four-cycle implementation (b) One-cycle implementation

Figure F-4 Two alternative architectures implementing the sum program.

time-shares the input bus as well as the adder. The second architecture performs the program in a single clock cycle. To achieve this performance, it was necessary to pipeline the algorithm; that is, multiple iterations of the computation overlap. The increased speed translates as expected to a higher hardware cost—one extra adder, extra registers, and a more dedicated bus architecture, including three input ports. Both architectures were produced automatically, given the behavioral description and the clock-cycle constraint. This includes the pipelining transformation.

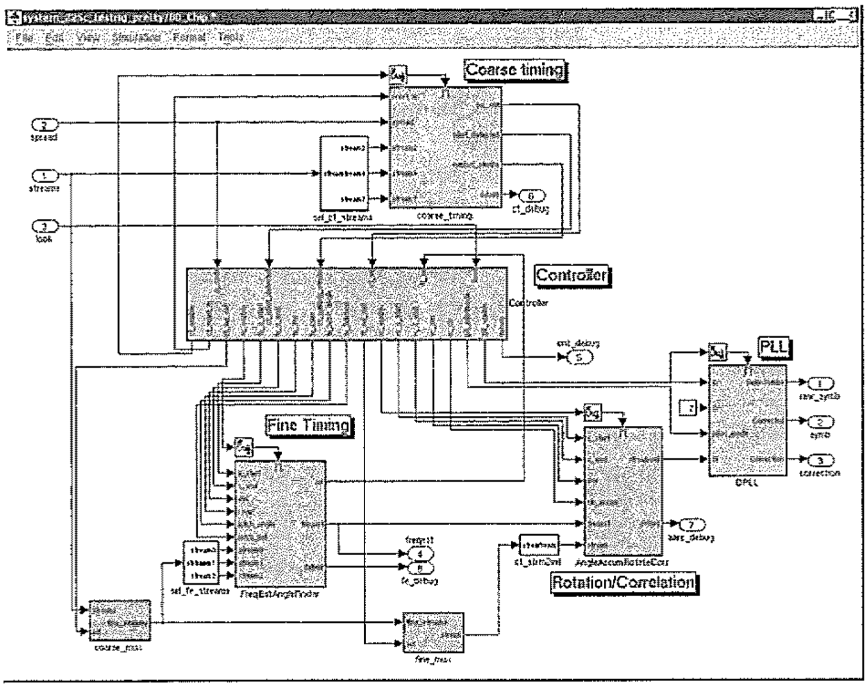
While architecture compilers have been extensively researched in the academic community (e.g., [DeMan86], [Rabaey91]), their overall impact has remained limited. Commercial introductions have been largely unsuccessful. A number of reasons for this slow penetration can be enumerated:

- Behavioral synthesis assumes the availability of an established synthesis approach at the register-transfer level. This has only recently come to a widespread acceptance. In addition, the discussion about the appropriate input language at the behavioral level has created a lot of confusion. The emergence of widely accepted input languages such as SystemC can change the momentum.
- For a long time, architecture synthesis has concentrated on a limited aspect of the overall design process. The impact of interconnect on the overall design cost, for example, was long ignored. Also, limitations on the architectural scope resulted in inferior solutions apparent to every experienced designer.
- Most importantly, the revolutionary advent of the system-on-a-chip has outstripped the evolutionary progress of the synthesis world. The hybrid nature of embedded system architectures that combine embedded processors with ASIC accelerators ultimately limits the usability of architectural synthesis. Current logic and sequential synthesis tools probably suffice for the accelerators. The challenge has shifted to the synthesis of the software that runs on the embedded processors, chip-level operation systems, driver generators, interconnect network synthesis, and architectural exploration.

Notwithstanding these observations, architectural synthesis has proven to be very successful in a number of application-specific areas. The design of high-performance accelerator units in areas such as wireless communications, storage, imaging, and consumer electronics has benefited greatly from compilers that translate high-level algorithmic functions into hard-wired dedicated solutions.

Example F.3 Architectural Synthesis of Wireless-Communications Processor [Silva01]

An advanced baseband processor for a wireless modem is generated automatically from a high-level description in the Simulink environment ([Mathworks01]). Simulink and Mathlab are tools used extensively in the world of communications design. Capturing the



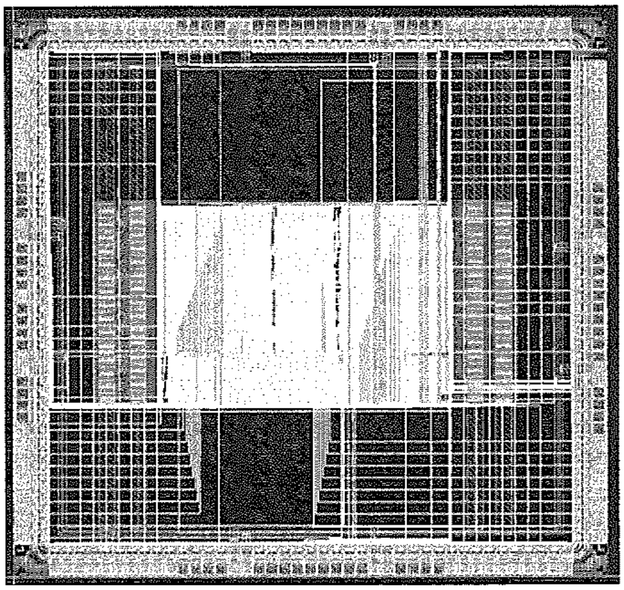


Figure F-5 Architectural synthesis of wireless baseband processor from Simulink (a) to silicon (b). The core area of the chip, which is pad limited, measures only 2 mm^2 in a 0.18 μm CMOS technology, and counts 600,000 transistors. The high transistor density (0.3 transistor/μm^2) demonstrates the effectiveness of today's physical design tools.

design specifications in that environment is a major help in bridging the chasm between systems and implementation engineer. The translation process from Simulink to implementation is managed by the "*Chip-in-a-day*" design environment [Davis01]. This tool manages the synthesis of the individual blocks from behavior to gate level, introduces the chip floorplan, performs the clock tree generation, and oversees the execution of the physical synthesis. The overall generation and verification process takes little more than 24 hours. A similar approach has also proven to be very successful in the mapping of high-level signal-processing functions on rapid-prototyping platforms such as FPGAs. The *System Generator* tool from Xilinx, Inc, for instance, maps modules such as filters, modulators, and correlators, described in the Mathworks Simulink environment, directly onto an FPGA module [SystemGenerator].

To Probe Further

For an in-depth overview of design synthesis, please refer to [DeMicheli94].

References

[AMPS99] AMPS, Intelligent Design Optimization, *http://www.synopsys.com/products/analysis/amps_ds.html*, Synopsys, Inc.

[Brayton84] R. Brayton et al., *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publishers, 1984.

[Brayton87] R. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang, "MIS: A Multilevel Logic Optimization System," *IEEE Trans. on CAD*, CAD-6, pp. 1062–81, November 1987.

[Carlson91] S. Carlson, *Introduction to HDL-Based Design Using VHDL*, Synopsys, Inc. 1991.

[Chu86] K. Chu and D. Pulfrey, "Design Procedures for Differential Cascode Logic," *IEEE Journal of Solid State Circuits*, vol. SC-21, no. 6, Dec. 1986, pp. 1082–1087.

[Davis01] W.R. Davis, N. Zhang, K. Camera, F. Chen, D. Markovic, N. Chan, B. Nikolic, R.W. Brodersen, "A Design Environment for High Throughput, Low Power, Dedicated Signal Processing Systems," *Proceedings CICC 2001*, San Diego, 2001.

[DesignCompiler] Design Compiler Technical Datasheet, *http://www.synopsys.com/products/logic/design_compiler.html*, Synopsys, Inc.

[DeMan86] H. De Man, J. Rabaey, P. Six, and L. Claesen, "Cathedral-II: A Silicon Compiler for Digital Signal Processing," *IEEE Design and Test*, vol. 3, no. 6, pp. 13–25, December 1986.

[DeMicheli94] G. De Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, 1994.

[Fishburn85] J. Fishburn and A. Dunlop, "TILOS: A Polynomial Programming Approach to Transistor Sizing," *Proceedings ICCAD-85*, pp. 326–328, Santa Clara, 1985.

[Gajski92] D. Gajski, N. Dutt, A. Wu, and S. Lin, *High-Level Synthesis—Introduction to Chip and System Design*, Kluwer Academic Publishers, 1992.

[Kurup97] P. Kurub and T. Abassi, *Logic Synthesis using Synopsys*, Kluwer Academic Publishers, 1997.

[Mathworks01] Matlab and Simulink, *http://www.mathworks.com*, The Mathworks

[Northrop01] G. Northrop, P. Lu, "A Semicustom Design Flow in High-Performance Microprocessor Design," *Proceedings 38th Design Automation Conference*, Las Vegas, June 2001.

[Rabaey91] J. Rabaey, C. Chu, P. Hoang and M. Potkonjak, "Fast Prototyping of Datapath-Intensive Architectures," *IEEE Design and Test*, vol. 8, pp. 40–51, 1991.

[Silva01] J.L. da Silva Jr., J. Shamberger, M.J. Ammer, C. Guo, S. Li, R. Shah, T. Tuan, M. Sheets, J.M. Rabaey, B. Nikolic, A. Sangiovanni-Vincentelli, P. Wright, "Design Methodology for PicoRadio Networks," *Proceedings DATE Conference*, Munich, March 2000.

[SystemGenerator] The Xilinx System Generator for DSP, *http://www.xilinx.com/xlnx/xil_prodcat_product.jsp?title=system_generator*, Xilinx, Inc.

CHAPTER

10

Timing Issues in Digital Circuits

Impact of clock skew and jitter on performance and functionality

Alternative timing methodologies

Synchronization issues in digital IC and board design

Clock generation

10.1 Introduction

All sequential circuits have one property in common—a well-defined ordering of the switching events must be imposed if the circuit is to operate correctly. If this were not the case, wrong data might be written into the memory elements, resulting in a functional failure. The *synchronous* system approach, in which all memory elements in the system are simultaneously updated using a globally distributed periodic synchronization signal (that is, a global clock signal), represents an effective and popular way to enforce this ordering. Functionality is ensured by imposing some strict constraints on the generation of the clock signals and their distribution to the memory elements distributed over the chip; noncompliance often leads to malfunction.

This chapter starts with an overview of the different timing methodologies. The majority of the text is devoted to the popular *synchronous approach*. We analyze the impact of spatial variations of the clock signal, called *clock skew*, and temporal variations of the clock signal, called *clock jitter*, and introduce techniques to cope with both. These variations fundamentally limit the performance that can be achieved using a conventional design methodology.

At the other end of the spectrum is an approach called *asynchronous design*, which avoids the problem of clock uncertainty altogether by eliminating the need for globally distributed clocks. After discussing the basics of asynchronous design approach, we analyze the associated overhead and identify some practical applications. The important issue of synchronization between different *clock domains* and *interfacing* between asynchronous and synchronous systems also deserve in-depth treatment. Finally, the fundamentals of on-chip clock generation using feedback are introduced, along with trends in timing.

10.2 Timing Classification of Digital Systems

In digital systems, signals can be classified depending on how they are related to a local clock [Messerschmitt90][Dally98]. Signals that transition only at predetermined periods in time can be classified as *synchronous*, *mesochronous*, or *plesiochronous* with respect to a system clock. A signal that can transition at arbitrary times, on the other hand, is considered *asynchronous*.

10.2.1 Synchronous Interconnect

A synchronous signal is one that has the exact same frequency as the local clock and maintains a known fixed phase offset to that clock. In such a timing framework, the signal is "synchronized" with the clock, and the data can be sampled directly without any uncertainty. In digital logic

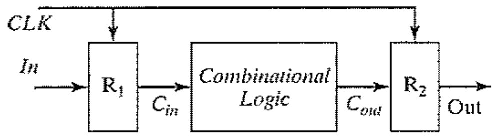


Figure 10-1 Synchronous interconnect methodology.

design, synchronous systems are the most straightforward type of interconnect. The flow of data in such a circuit proceeds in lockstep with the system clock, as illustrated in Figure 10-1.

Here, the input data signal *In* is sampled with register R_1 to produce signal C_{in}, which is synchronous with the system clock, and then it is passed along to the combinational logic block. After a suitable setting period, the output C_{out} becomes valid. Its value is sampled by R_2 which synchronizes the output with the clock. In a sense, the *certainty period* of signal C_{out}—the period during which data are valid—is synchronized with the system clock. This allows register R_2 to sample the data with complete confidence. The **length of the uncertainty period**, or the period during which data are not valid, **places an upper bound on how fast a synchronous system can be clocked**.

10.2.2 Mesochronous Interconnect

A *mesochronous* signal—*meso* is Greek for "middle"—is a signal that not only has the same frequency as the local clock, but also has an unknown phase offset with respect to that clock. For example, if data are being passed between two different clock domains, the data signal transmitted from the first module can have an unknown phase relationship to the clock of the receiving module. In such a system, it is not possible to directly sample the output at the receiving module because of the uncertainty in the phase offset. A (mesochronous) synchronizer can be used to synchronize the data signal with the receiving clock, as shown in Figure 10.2. The synchronizer serves to adjust the phase of the received signal to ensure proper sampling.

In Figure 10-2, signal D_1 is synchronous with respect to Clk_A. However, D_1 and D_2 are mesochronous with Clk_B because of the unknown phase difference between Clk_A and Clk_B and the unknown interconnect delay in the path between Block A and Block B. The role of the synchronizer is to adjust the variable delay line such that the data signal D_3 (a delayed version of D_2) is aligned properly with the system clock of Block B. In this example, the variable delay element is adjusted by measuring the phase difference between the received signal and the local clock. Register R_2 samples the incoming data during the certainty period, after which the signal D_4 becomes synchronous with Clk_B.

10.2.3 Plesiochronous Interconnect

A *plesiochronous* signal is one that has a frequency that is nominally the same as that of the local clock, yet is slightly different. (In Greek, *plesio* means "near.") This causes the phase difference to drift in time. This scenario can easily arise when two interacting modules have independent clocks generated from separate crystal oscillators. Since the transmitted signal can arrive at the receiving module at a different rate than the local clock, one needs to utilize a buffering scheme to ensure that all data are received. Typically, plesiochronous interconnect occurs only in distributed

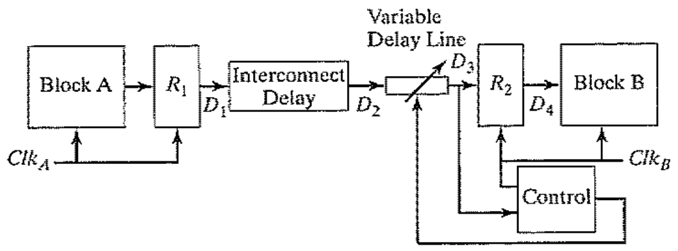


Figure 10-2 Mesochronous communication approach using variable delay line.

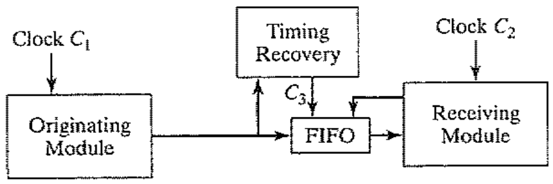


Figure 10-3 Plesiochronous communications by using a FIFO.

systems that contain long-distance communications, since chip- or even board-level circuits typically utilize a common oscillator to derive local clocks. A possible framework for plesiochronous interconnect is shown in Figure 10-3.

In this digital communications framework, the originating module issues data at some unknown rate C_1, which is plesiochronous with respect to C_2. The timing recovery unit is responsible for deriving clock C_3 from the data sequence and buffering the data in a FIFO. As a result, C_3 will be synchronous with the data at the input of the FIFO and will be mesochronous with C_1. Since the clock frequencies from the originating and receiving modules are mismatched, data might have to be dropped if the transmit frequency is faster, or data can be duplicated if the transmit frequency is slower than the receive frequency. However, by making the FIFO large enough, as well as periodically resetting the system whenever an overflow condition occurs, robust communication can be achieved.

10.2.4 Asynchronous Interconnect

Asynchronous signals can transition arbitrarily at any time, and they are not slaved to any local clock. As a result, it is not straightforward to map these arbitrary transitions into a synchronized data stream. It is possible to synchronize asynchronous signals by detecting events and by introducing latencies into the data stream synchronized to a local clock. A more natural way to handle asynchronous signals, however, is simply to eliminate the use of local clocks and utilize a self-timed asynchronous design approach. In such an approach, communication between modules is controlled through a handshaking protocol that ensures the proper ordering of operations.

When a logic block completes an operation (Figure 10-4), it will generate a completion signal DV to indicate that output data are valid. The handshaking signals then initiate a data transfer to the next block, which latches in the new data and begins a new computation by asserting the initialization signal I. Asynchronous designs are advantageous because computations are

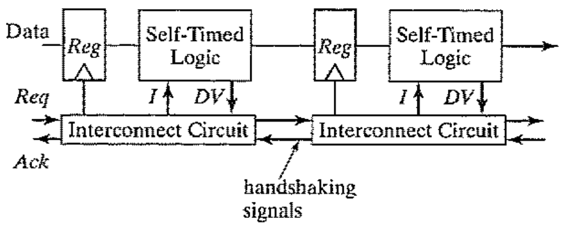


Figure 10-4 Asynchronous design methodology for simple pipeline interconnect.

performed at the native speed of the logic, and block computations occur whenever data become available. There is no need to manage clock *skew*, and the design methodology leads to a very modular approach in which interaction between blocks simply occurs through a handshaking procedure. However, these protocols result in increased complexity and overhead in communication, which impacts performance.

10.3 Synchronous Design—An In-Depth Perspective

10.3.1 Synchronous Timing Basics

Virtually all systems designed today use a periodic *synchronization* signal or clock. The generation and distribution of a clock has a significant impact on the performance and power dissipation of the system. For the time being, let us assume a positive *edge-triggered* system, in which the rising edge of the clock denotes the beginning and completion of a clock cycle. In an ideal world, the phase of the clock (i.e., the position of the clock edge relative to the reference) at various points in the system is exactly equal, assuming that the clock paths from the central distribution point to each register are perfectly balanced. Figure 10-5 shows the basic structure of a synchronous pipelined datapath. In the ideal scenario, the clocks at registers 1 and 2 have the same period and transition at the exact same time.

Assume that the following timing parameters of the sequential circuit are available:

- The contamination or minimum delay ($t_{c-q,cd}$) and the maximum propagation delay of the register (t_{c-q}).
- The setup (t_{su}) and hold times (t_{hold}) for the registers.
- The contamination delay ($t_{logic,cd}$) and the maximum delay (t_{logic}) of the combinational logic.
- The positions of the rising edges of the clocks CLK_1 and CLK_2 (t_{clk1} and t_{clk2}, respectively), relative to a global reference.

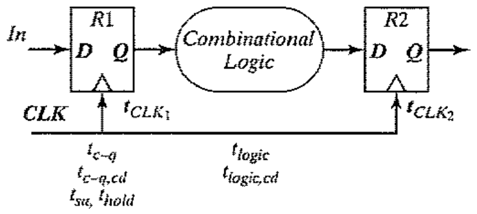


Figure 10-5 Pipelined datapath circuit and timing parameters.

Under the ideal condition that $t_{clk1} = t_{clk2}$, the minimum clock period required for this sequential circuit is determined solely by the worst case propagation delays. The period must be long enough for the data to propagate through the registers and logic and to be set up at the destination register before the next rising edge of the clock. As we saw in Chapter 7, this constraint is given by the following expression:

$$T > t_{c-q} + t_{logic} + t_{su} \tag{10.1}$$

At the same time, the hold time of the destination register must be shorter than the minimum propagation delay through the logic network:

$$t_{hold} < t_{c-q,cd} + t_{logic,cd} \tag{10.2}$$

Unfortunately, the preceding analysis is somewhat simplistic, since the clock is never ideal. The different clock events turn out to be neither perfectly periodic nor perfectly simultaneous. As a result of process and environmental variations, the clock signal can have both *spatial* and *temporal* variations, which lead to performance degradation and/or circuit malfunction.

Clock Skew

The spatial variation in arrival time of a clock transition on an integrated circuit is commonly referred to as *clock skew*. The *clock skew* between two points i and j on an IC is given by $\delta(i, j) = t_i - t_j$, where t_i and t_j are the positions of the rising edge of the clock with respect to the reference. Consider the transfer of data between registers $R1$ and $R2$ in Figure 10-5. The clock skew can be positive or negative depending upon the routing direction and position of the clock source. The timing diagram for the case with positive skew is shown in Figure 10-6. As the figure illustrates, the rising clock edge is delayed by a positive δ at the second register.

Clock skew is caused by static mismatches in the clock paths and differences in the clock load. By definition, skew is constant from cycle to cycle. That is, if in one cycle CLK_2 lagged CLK_1 by δ, then on the next cycle, it will lag it by the same amount. It is important to note that clock skew does not result in clock period variation, but only in phase shift.

The clock-skew phenomenon has strong implications for both the performance and the functionality of sequential systems. First, consider the impact of clock skew on performance. We can see from Figure 10-6 that a new input In sampled by $R1$ at edge ① will propagate through the com-

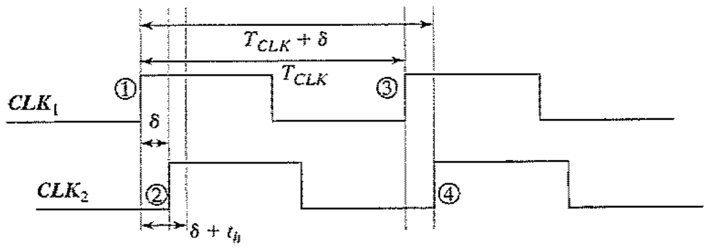


Figure 10-6 Timing diagram to study the impact of clock skew on performance and functionality. In this sample timing diagram, $\delta > 0$.

binational logic and be sampled by $R2$ on edge ④. If the clock skew is positive, the time available for a signal to propagate from $R1$ to $R2$ is increased by the skew δ. The output of the combinational logic must be valid one setup time before the rising edge of CLK_2 (point ④). The constraint on the minimum clock period can then be derived as follows:

$$T + \delta \geq t_{c-q} + t_{logic} + t_{su} \quad \text{or} \quad T \geq t_{c-q} + t_{logic} + t_{su} - \delta \tag{10.3}$$

This equation suggests that clock skew actually has the potential to improve the performance of the circuit. That is, the minimum clock period required to operate the circuit reliably reduces with increasing clock skew! This is indeed correct, but unfortunately, increasing skew makes the circuit more susceptible to race conditions, which may harm the correct operation of sequential systems.

This can be illustrated by the following example: Assume again that input In is sampled on the rising edge of CLK_1 at edge ① into $R1$. The new value at the output of $R1$ propagates through the combinational logic and should be valid before edge ④ at CLK_2. However, if the minimum delay of the combinational logic block is *small*, the inputs to $R2$ may change before the clock edge ②, resulting in incorrect evaluation. To avoid races, we must ensure that the minimum propagation delay through the register and logic is long enough that the inputs to $R2$ are valid for a hold time after edge ②. The constraint can be formally stated as

$$\delta + t_{hold} < t_{(c-q,cd)} + t_{(logic,cd)}$$

or (10.4)

$$\delta < t_{(c-q,cd)} + t_{(logic,cd)} - t_{hold}$$

Figure 10-7 shows the timing diagram for the case in which $\delta < 0$. For this case, the rising edge of CLK_2 happens before the rising edge of CLK_1. On the rising edge of CLK_1, a new input is sampled by $R1$. The new data propagate through the combinational logic, and they are sampled by $R2$ on the rising edge of CLK_2, which corresponds to edge ④. As Figure 10-7 and Eq. (10.3) clearly show, a negative skew adversely impacts the performance of a sequential system. However, assuming $t_{hold} + \delta < t_{(c-q,cd)} + t_{(logic,cd)}$, a negative skew implies that the system never fails, since edge ② happens before edge ①!

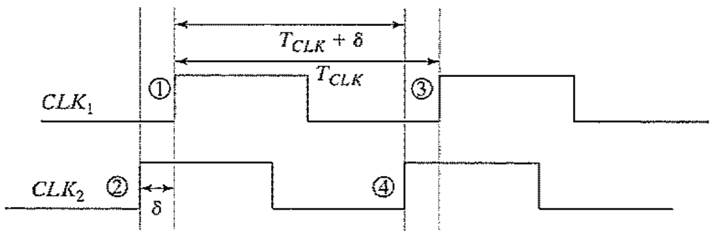


Figure 10-7 Timing diagram for the case when $\delta < 0$. The rising edge of CLK_2 arrives earlier than the edge of CLK_1.

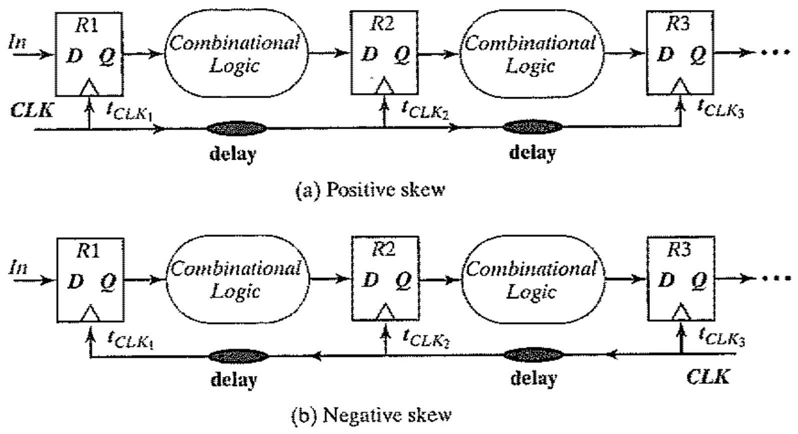


Figure 10-8 Positive and negative clock skew scenarios.

Example scenarios for positive and negative clock skew are shown in Figure 10-8.

- $\delta > 0$—This corresponds to a clock routed in the same direction as the flow of the data through the pipeline (Figure 10-8a). In this case, the skew has to be strictly controlled and satisfy Eq. . If the constraint is not met, the circuit malfunctions **independently of the clock period**. Reducing the clock frequency of an edge-triggered circuit does not help getting around skew problems! It is therefore necessary to satisfy the hold-time constraints at design time. On the other hand, positive skew increases the through put of the circuit as expressed by Eq. (10.3). The clock period can be shortened by δ. The extent of this improvement is limited, as large values of δ soon provoke violations of Eq. .
- $\delta < 0$—When the clock is routed in the opposite direction of the data (Figure 10-8b), the skew is negative and provides significant immunity to races; if the hold time is zero or negative, races are eliminated because Eq. is unconditionally met! The skew reduces the time available for actual computation so that the clock period has to be increased by $|\delta|$. In summary, routing the clock in the opposite direction of the data avoids disasters, but hampers the circuit performance.

Unfortunately, since a general logic circuit can have data flowing in both directions (for example, circuits with feedback), this solution to eliminate races does not always work. Figure 10-9 shows that the skew can assume both positive and negative values, depending on the direction of the data transfer. Under these circumstances, the designer has to account for the worst case skew condition. In general, routing the clock so that only negative skew occurs is not feasible. Therefore, the design of a low-skew clock network is essential.

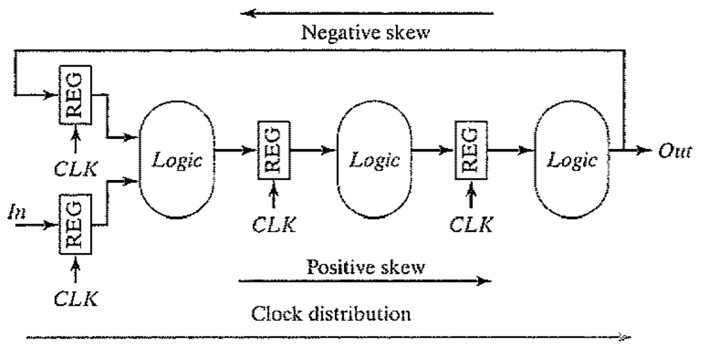


Figure 10-9 Datapath structure with feedback.

Example 10.1 Propagation and Contamination Delay Estimation

Consider the logic network shown in Figure 10-10. Determine the contamination and propagation delays of the network, given a worst case gate delay of t_{gate}. We also assume that the maximum and minimum delays of the gates are identical.

The contamination delay is easily found; it equals $2t_{gate}$, and is the delay through OR_1 and OR_2. On the other hand, computation of the worst case propagation delay is not as simple. At first glance, it would appear that the worst case corresponds to path ①, and its delay is $5t_{gate}$. However, when analyzing the data dependencies, it becomes obvious that path ① can never be exercised. Path ① is called a *false path*. If $A = 1$, the critical path goes through OR_1 and OR_2. If $A = 0$ and $B = 0$, the critical path is through I_1, OR_1 and OR_2 (corresponding to a delay of $3t_{gate}$). For the case in which $A = 0$ and $B = 1$, the longest path goes through I_1, OR_1, AND_3 and OR_2. In other words, for this simple (but contrived) network, the output does not even depend on inputs C and D (that is, there is **redundancy**). Therefore, the actual propagation delay is $4t_{gate}$. Given the propagation and contamination delay, the minimum and maximum allowable skew can be easily computed.

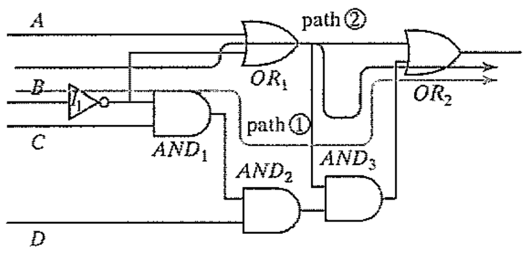


Figure 10-10 Logic network for computation of performance.

WARNING: The computation of the worst case propagation delay for combinational logic, due to the existence of *false paths*, cannot be obtained simply by adding the propagation delays of individual logic gates. The critical path is strongly dependent on circuit topology and data dependencies.

Clock Jitter

Clock jitter refers to the temporal variation of the clock period at a given point on the chip—that is, the clock period can reduce or expand on a cycle-by-cycle basis. It is strictly a temporal uncertainty measure, and it is often specified at a given point. **Jitter** can be measured and characterized in a number of ways and **is a zero-mean random variable.** The *absolute jitter* (t_{jitter}) refers to the worst case variation (absolute value) of a clock edge at a given location with respect to an ideally periodic reference clock edge. The *cycle-to-cycle jitter* (T_{jitter}) typically refers to the time-varying deviations of a single clock period relative to an ideal reference clock. For a given spatial location i, it is given as $T^i_{jitter}(n) = t^i_{clk,n+1} - t^i_{clk,n} - T_{CLK}$, where $t^i_{clk,n+1}$ and $t^i_{clk,n}$ represent the arrival time of the $n+1$th and the nth clock edges at node i, respectively, and T_{CLK} is the nominal clock period. Under the worst case conditions, the magnitude of the cycle-to-cycle jitter equals twice the absolute jitter ($2t^i_{jitter}$).

Jitter directly impacts the performance of a sequential system. Figure 10-11 shows the nominal clock period, as well as the variation in period. Ideally, the clock period starts at edge ② and ends at edge ⑤, with a nominal clock period of T_{CLK}. However, the worst case scenario happens when the leading edge of the current clock period is delayed by jitter (edge ③), while jitter causes the leading edge of the next clock period to occur early (edge ④). As a result, the total time available to complete the operation is reduced by $2t_{jitter}$ in the worst case and is given by

$$T_{CLK} - 2t_{jitter} \geq t_{c-q} + t_{logic} + t_{su} \text{ or } T \geq t_{c-q} + t_{logic} + t_{su} + 2t_{jitter} \tag{10.5}$$

Equation (10.5) illustrates that jitter directly reduces the performance of a sequential circuit. Keeping it within strict bounds is essential if one is concerned about performance.

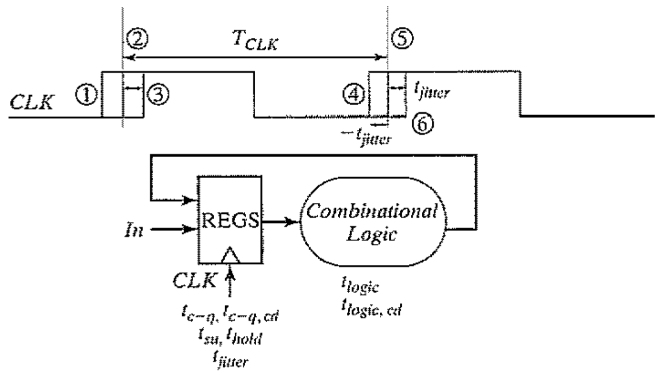


Figure 10-11 Circuit for studying the impact of jitter on performance.

The Combined Impact of Skew and Jitter

In this section, the combined impact of skew and jitter is studied for conventional edge-triggered clocking. Consider the sequential circuit shown in Figure 10-14.

Assume that as a result of the clock distribution, there is a static skew δ between the clock signals at the two registers (assume that $\delta > 0$). Furthermore, the two clocks experience a jitter of t_{jitter}. To determine the constraint on the minimum clock period, we must look at the minimum available time to perform the required computation. The worst case occurs when the leading edge of the current clock period on CLK_1 happens late (edge ③) and the leading edge of the next cycle of CLK_2 happens early (edge ⑩). This results in the following constraint:

$$T_{CLK} + \delta - 2t_{jitter} \geq t_{c-q} + t_{logic} + t_{su}$$

or (10.6)

$$T \geq t_{c-q} + t_{logic} + t_{su} - \delta + 2t_{jitter}$$

This equation illustrates that positive skew can provide a performance advantage. On the other hand, *jitter* always has a negative impact on the minimum clock period.[1]

To formulate the minimum delay constraint, consider the case in which the leading edge of the CLK_1 cycle arrives early (edge ①), and the leading edges the current cycle of CLK_2 arrives late (edge ⑥). The separation between edges ① and ⑥ should be smaller than the minimum delay through the network. This results in

$$\delta + t_{hold} + 2t_{jitter} < t_{(c-q,cd)} + t_{(logic,cd)}$$

or (10.7)

$$\delta < t_{(c-q,cd)} + t_{(logic,cd)} - t_{hold} - 2t_{jitter}$$

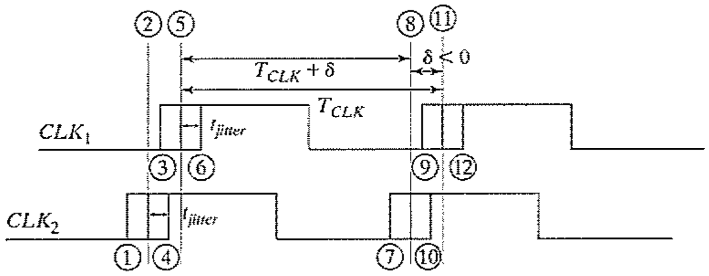


Figure 10-12 Sequence circuit with a negative clock skew (δ). The skew is assumed to be larger than the *jitter.*

[1]This analysis is definitely for the worst case. It assumes that the jitter values at the source and the destination nodes are independent statistical variables. In reality, the clock edges involved in the hold-time analysis are derived from the same clock edge and are statistically dependent. Taking this dependence into account reduces the timing constraints substantially.

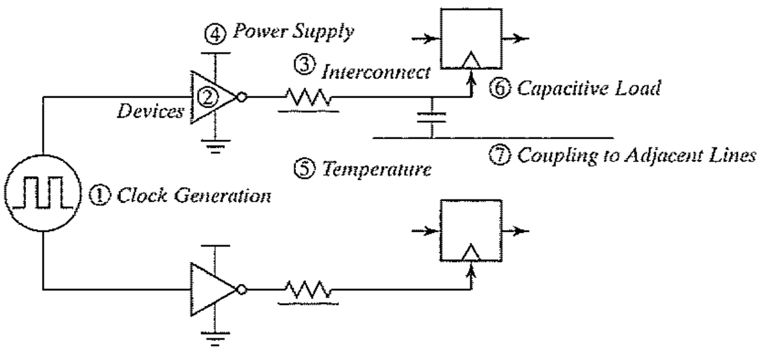


Figure 10-13 Skew and jitter sources in synchronous clock distribution.

This relation indicates that the acceptable skew is reduced by the *jitter* of the two signals.

Now consider the case in which the skew is negative ($\delta < 0$), as shown in Figure 10-12. Assume that $|\delta| > t_{jitter}$. It can be verified that the worst case timing is exactly the same as in the previous analysis, with δ taking a negative value. That is, negative skew reduces performance.

10.3.2 Sources of Skew and Jitter

A perfect *clock* is defined as a periodic signal that simultaneously triggers various memory elements on the chip. However, due to a variety of process and environmental variations, clocks are not ideal. To illustrate the sources of skew and jitter, consider a simplistic view of a typical clock generation and distribution network, as shown in Figure 10-13. A high-frequency clock is either provided from off chip or generated on chip. From a central point, the clock is distributed using multiple *matched* paths to low-level sequential elements. In this picture, two paths are shown. The clock paths include the wiring and the associated distributed buffers required to drive interconnect and loads. A key point to realize in clock distribution is that the **absolute delay through a clock distribution path is not important**; what matters is the relative arrival time at the register points at the end of each path. It is perfectly acceptable for the clock signal to take multiple cycles to get from a central distribution point to a low-level register as long as all clocks arrive at the same time at all the registers on the chip.

There are many reasons why the two parallel paths don't result in exactly the same delay. The sources of clock uncertainty can be classified in several ways. First, errors can be divided into two categories: *systematic* and *random*. *Systematic* errors are nominally identical from chip to chip and are predictable (for instance, variation in total load capacitance of each clock path). In principle, such errors can be modeled and corrected at design time, given sufficiently good models and simulators. Short of that, systematic errors can be deduced from measurements over a set of chips, and the design can be adjusted to compensate. *Random* errors are due to manufacturing variations that are difficult to model and eliminate (for instance, dopant fluctuations that result in threshold variations).

Mismatches may also be characterized as *static* or *time varying*. In practice, a continuum exists between changes that are slower than the time constant of interest and those that are faster. For example, temperature gradients on a chip vary on a millisecond time scale. A clock network

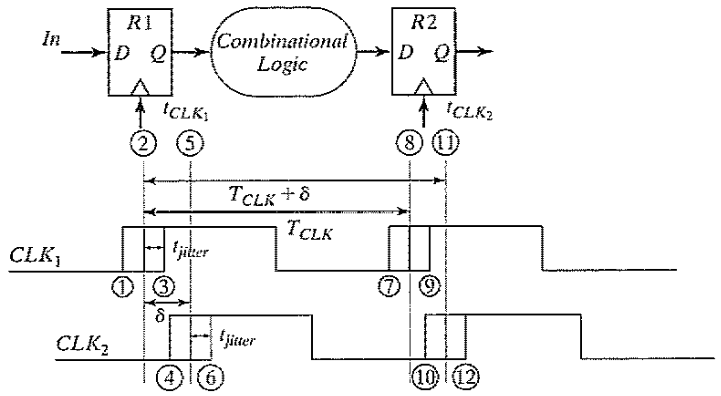


Figure 10-14 Sequential circuit to study the impact of skew and jitter on *edge-triggered* systems. In this example, a positive *skew* (δ) is assumed.

tuned by a one-time calibration is vulnerable to the time-varying mismatch caused by the varying thermal gradients. On the other hand, thermal changes appear essentially static to a feedback network with a bandwidth of several megahertz. Another example is fielded by power-supply noise. The clock net is usually by far the largest signal net on the chip, and simultaneous transitions on the clock drivers induce noise in the power supply. This high-speed effect does not create a time-varying mismatch, because it is the same at every clock cycle and affects each rising clock edge the same way. Of course, this power-supply glitch may still cause static mismatch if it is not the same throughout the chip. The various sources of skew and jitter introduced in Figure 10-13 are described and characterized in detail in the sections that follow.

Clock-Signal Generation (1)

The generation of the clock signal itself causes **jitter**. A typical on-chip clock generator, as described at the end of this chapter, takes a low-frequency reference clock signal and produces a high-frequency global reference for the processor. The core of such a generator is a *voltage-controlled oscillator* (VCO). This is an analog circuit, sensitive to intrinsic device noise and power-supply variations. A major problem is the coupling from the surrounding noisy digital circuitry through the substrate. This is especially a problem in modern fabrication processes that use a lightly doped epitaxy on the heavily doped substrate (to combat latch up). This causes substrate noise to travel over large distances on the chip [Maneatis00]. These noise sources cause temporal variations in the clock signal that propagate unfiltered through the clock drivers to the flip-flops, and result in *cycle-to-cycle* clock-period variations.

Manufacturing Device Variations (2)

Distributed buffers are integral components of the clock distribution networks. They are required to drive both the register loads and the global and local interconnects. The matching of devices in the buffers along multiple clock paths is critical to minimizing timing uncertainty. Unfortunately, as a result of process variations, device parameters in the buffers vary along different paths, resulting in *static skew*. There are many sources of variations that contribute, such as oxide variations (which affect the gain and threshold), dopant variations, and lateral dimension (width and length) variations. The doping variations can affect the depth of junction and dopant profiles and cause electrical parameters (such as device threshold and parasitic capacitances) to vary.

The orientation of polysilicon also can have some impact on the device parameters. Keeping the orientation the same across the chip for the clock drivers is therefore critical. Variation in the polysilicon critical dimension is particularly important, because it translates directly into MOS transistor channel length, impacting the drive current and switching characteristics. Spatial variation usually consists of a wafer-level (or within-wafer) variation and a die-level (or within-die) variation. At least part of this variation is systematic and therefore can be modeled and compensated for. The random variations, however, ultimately limit the matching and lower bound of the skew that can be achieved.

Interconnect Variations (3)

Vertical and lateral dimension variations cause the interconnect capacitance and resistance to vary across a chip. Since this variation is static, it causes **skew** between different paths. One important source of interconnect variation is the *Inter-layer Dielectric (ILD)* thickness variation. In the formation of aluminum interconnect, layers of silicon dioxide are interposed between layers of patterned metallization. Oxide is deposited over a layer of patterned metal features, generally resulting in some remaining step height or surface topography. *Chemical–mechanical polishing* (CMP) is used to "planarize" the surface and remove the topography resulting from deposition and etch (as described in Chapter 3 and shown in Figure 10-15a). While at the feature scale (i.e., over an individual metal line), CMP can achieve excellent planarity, there are limitations on it over a global range. This is due primarily to variations in the polish rate, which is a function of the circuit layout density and pattern effects. Figure 10-15b shows this effect—the polish rate is higher for the lower-spatial-density region, resulting in a smaller dielectric thickness and higher capacitance.

The assessment and control of variation is of critical importance in semiconductor process development and manufacturing. Significant advances have been made to develop analytical

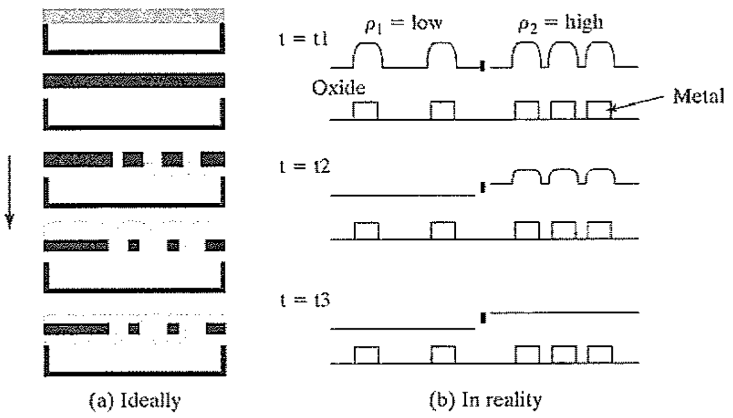


Figure 10-15 Inter-level Dielectric (ILD) thickness variation due to density (Courtesy of Duane Boning.).

models for estimating the ILD thickness variations, based on spatial density. Since this component is often predictable from the layout, it is possible to actually correct for the systematic component at design time (e.g., by adding appropriate delays or making the density uniform by adding "dummy fills"). Figure 10-16 shows the spatial pattern density and ILD thickness for a

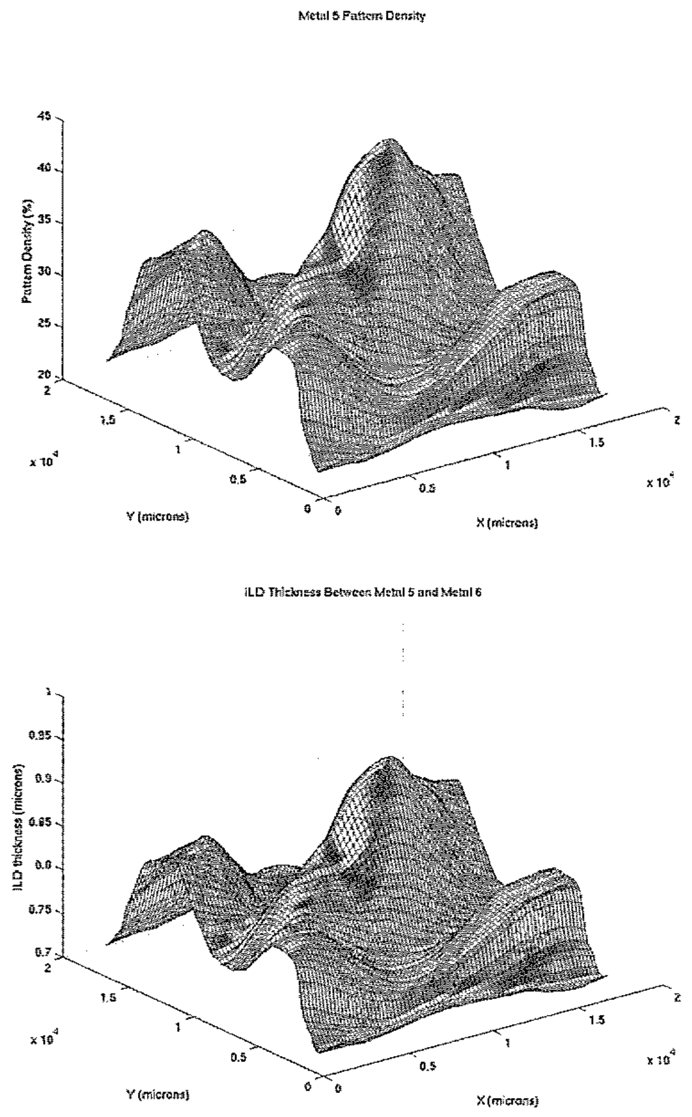


Figure 10-16 Pattern density and ILD thickness variation for a high-performance microprocessor. (Courtesy of Duane Boning)

high-performance microprocessor. The graphs show a clear correlation between the density and the thickness of the dielectric. Hence, clock distribution networks must exploit such information in order to reduce clock skew.

Other interconnect variations include deviations in the width of the wires and line spacing, which result from photolithography and etch dependencies. At the lower levels of the metallization hierarchy, lithographic effects are more important, while etch effects that depend on width and layout are dominant at the higher levels. The width is a critical parameter because it directly impacts the resistance of the line, and the wire spacing affects the wire-to-wire capacitance. A detailed review of device and interconnect variations is presented in [Boning00]. Recent processors use copper interconnects, in which line thickness variations are also seen to be highly pattern dependent due to CMP dishing and erosion effects [Park00].

Environmental Variations (4 and 5)

Environmental variations probably are the most significant contributors to **skew and jitter**. The two major sources of environmental variations are *temperature* and *power supply*. Temperature gradients across the chip result from variations in power dissipation across the die. These gradients can be quite large, as shown in Figure 10-17, which displays a snapshot of the surface temperature of the DEC 21064 microprocessor. Temperature variation has become an important issue with *clock gating*, where some parts of the chip may be idle, while other parts of the chip are fully active. Clock gating has become popular in recent years as a means to minimize power dissipation in idle modules (as described in a later section). Shutting off parts of the chip leads to large temperature variations. Since the device parameters (such as threshold and mobility) depend strongly on temperature, the buffer delay for a clock distribution network can vary drastically from path to path. More importantly, this component is time varying, since the temperature changes as the logic activity of the circuit varies. Hence, it is not sufficient to simulate the clock networks at worst case corners of temperature; instead, the worst case variation in temperature must be simulated. An interesting question is whether temperature variation contributes to skew or to jitter. Clearly, the difference in temperature is time varying, but the changes are rela-

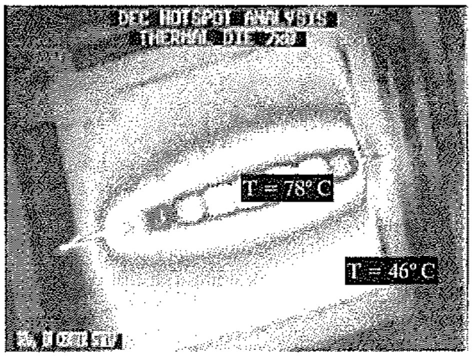


Figure 10-17 Temperature variation (snapshot) over DEC 21064 microprocessor. The highest temperature occurs at the central clock driver [Herrick00].

tively slow (typical time constants for temperature changes are on the order of milliseconds). Therefore, it is usually considered as a skew component and the worst case conditions are used. Fortunately, by using feedback, it is possible to calibrate the temperature and to compensate for this effect.

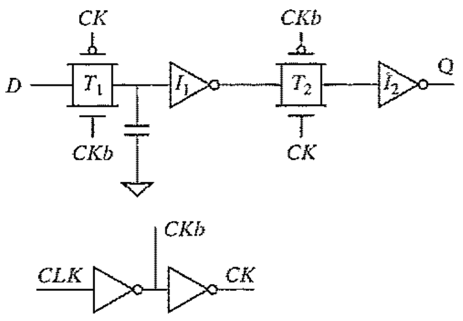
Power-supply variations, on the other hand, are the major source of **jitter** in clock distribution networks. The delay through buffers is a very strong function of power supply, as it directly affects the drive of the transistors. As with temperature, the power-supply voltage is a strong function of the switching activity. Therefore, the buffer delay varies strongly from path to path. Power-supply variations can be classified into slow- (or static) and high-frequency variations. Static power-supply variations may result from fixed currents drawn from various modules, while high-frequency variations result from instantaneous *IR* drops along the power grid due to fluctuations in switching activity. Inductive effects on the power supply also are a major concern since they cause voltage fluctuations. Again, clock gating has exacerbated this problem, because the logic transitions between the idle and active states can cause major changes in current drawn from the supply. Since the power supply can change rapidly, the period of the clock signal is modulated on a cycle-by-cycle basis, resulting in jitter. The jitter on two different clock points may be correlated or uncorrelated, depending on how the power network is configured and the profile of switching patterns. Unfortunately, high-frequency power-supply changes are difficult to compensate for, even with feedback techniques. Consequently, **power-supply noise fundamentally limits the performance of clock networks**. To minimize power-supply variations, high-performance designs add decoupling capacitance around major clock drivers.

Capacitive Coupling (6 and 7)

Changes in capacitive load also contribute to timing uncertainty. There are two major sources of capacitive-load variations: coupling between the clock lines and adjacent signal wires, and variation in gate capacitance. The clock network includes both the interconnect and the gate capacitance of latches and registers. Any coupling between the clock wire and adjacent signal results in timing uncertainty. Since the adjacent signal can transition in arbitrary directions and at arbitrary times, the exact coupling to the clock network is not fixed from cycle to cycle, causing **jitter**. Another major source of clock uncertainty is the variation in the gate capacitance contributed by the connecting sequential elements. The load capacitance is highly nonlinear and depends on the applied voltage. For many latches and registers, the clock load is a function of the current state of the latch/register (i.e., the values stored on the internal nodes of the circuit), as well as the next state. This causes the delay through the clock buffers to vary from cycle to cycle, which causes jitter.

Example 10.2 Data-Dependent Clock Jitter

Consider the circuit shown in Figure 10-18, where a minimum-sized local clock buffer drives a register. (Actually, each clock buffer drives four registers, though only one is shown here.) The simulation shows *CKb*, the output of the first inverter for four possible



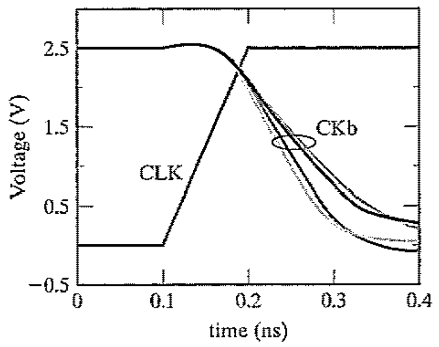


Figure 10-18 Impact of data-dependent clock load on clock jitter for transmission-gate register.

transitions ($0 \rightarrow 0$, $0 \rightarrow 1$, $1 \rightarrow 0$ and $1 \rightarrow 1$). The jitter on the clock based on data-dependent capacitance is illustrated. In general, the only way to deal with this problem is to use registers that do not exhibit a large variation in load as a function of data—for example, the differential sense-amplifier register shown in Chapter 7.

10.3.3 Clock-Distribution Techniques

It is clear from the previous discussion that clock skew and jitter are major issues in digital circuits, and can fundamentally limit the performance of a digital system. It is therefore necessary to design a clock network that minimizes both. While designing that clock network, a close eye should be kept on the associated power dissipation. In most high-speed digital processors, a majority of the power is dissipated in the clock network. To reduce power dissipation, clock networks must support clock conditioning—that is, the ability to shut down parts of the clock network. Unfortunately, clock gating results in additional clock uncertainty (as described earlier).

In this section, an overview of basic constructs in high-performance clock distribution techniques is presented, along with a case study of clock distribution in the Alpha microprocessors. There are many degrees of freedom in the design of a clock network, including the type of material used for wires, the basic topology and hierarchy, the sizing of wires and buffers, the rise and fall times, and the partitioning of load capacitances.

Fabrics for Clocking

Clock networks typically include a network that is used to distribute a **global reference** to various parts of the chip, and a final stage that is responsible for **local distribution** of the clock while considering the local load variations. Most clock distribution schemes exploit the fact that the absolute delay from a central clock source to the clocking elements is irrelevant—only the relative phase between two clocking points is important. Therefore, one common approach to distributing a clock is to use balanced paths (called *trees*).