```
Instance name        in pin-->out pin    tr      total   incr    cell
--------------------------------------------------------------------
END_OF_PATH
D.a_r_ff_b2                              R       4.52    0.00    DF1
INBUF_24           : PAD--->Y            R       4.52    4.52    INBUF
a_2_                                     R       0.00    0.00
BEGIN_OF_PATH

---------------------CLOCK to SETUP longest path---------------------
Rise delay, Worst case

Instance name        in pin-->out pin    tr      total   incr    cell
--------------------------------------------------------------------
END_OF_PATH
D.sel_r_ff                               R       9.99    0.00    DF1
I_1_CM8            : S10--->Y            R       9.99    0.00    CM8
I_3_CM8            : S00--->Y            R       9.99    4.40    CM8
a_r_ff_b1          : CLK--->Q            R       5.60    5.60    DF1
BEGIN_OF_PATH


---------------------CLOCK to OUTPAD longest path--------------------
Rise delay, Worst case

Instance name        in pin-->out pin    tr      total   incr    cell
--------------------------------------------------------------------
END_OF_PATH
outp_2_                                  R       11.95
OUTBUF_31          : D--->PAD            R       11.95   7.55    OUTBUF
outp_ff_b2         : CLK--->Q            R       4.40    4.40    DF1
BEGIN_OF_PATH
```

The timing analyzer has examined the following:

1. Paths that start at an input pad and end on the data input of a sequential logic cell (the D input to a D flip-flop, for example). We might call this an **entry path** (or input-to-D path) to a pipelined design. The longest **entry delay** (or input-to-setup delay) is 4.52 ns.
2. Paths that start at a clock input to a sequential logic cell and end at the data input of a sequential logic cell. This is a **stage path** (register-to-register path or clock-to-D path) in a pipeline stage. The longest **stage delay** (clock-to-D delay) is 9.99 ns.
3. Paths that start at a sequential logic cell output and end at an output pad. This is an **exit path** (clock-to-output path) from the pipeline. The longest **exit delay** (clock-to-output delay) is 11.95 ns.

By pipelining the design we added three clock periods of latency, but we increased the estimated operating speed. The longest prelayout critical path is now an exit delay, approximately 12 ns—more than doubling the maximum operating frequency. Next, we route the registered version of the design. The Actel software informs us that the postroute maximum stage delay is 11.3 ns (close to the preroute estimate of 9.99 ns). To check this figure we can perform another timing analysis. This time we shall measure the stage delays (the start points are all clock pins, and the end points are all inputs to sequential cells, in our case the D input to a D flip-flop). We need to define the **sets** of nodes at which to start and end the timing analysis (similar to the path clusters we used to specify timing constraints in logic synthesis). In the Actel timing analyzer we can use predefined sets `'clock'` (flip-flop clock pins) and `'gated'` (flip-flop inputs) as follows:

```
timer> startset clock
timer> endset gated
timer> longest
 1st  longest path to all endpins
Rank Total Start pin      First Net      End Net        End pin
  0   11.3 a_r_ff_b2:CLK  a_r_2_         block_0_OUT1   sel_r_ff:D
  1    6.6 sel_r_ff:CLK   sel_r          DEF_NET_50     outp_ff_b0:D
... 8 similar lines omitted ...
```

We could try to reduce the long stage delay (11.3 ns), but we have already seen from the preroute timing estimates that an exit delay may be the critical path. Next, we check some other important timing parameters.

13.7.1 Hold Time

Hold-time problems can occur if there is clock skew between adjacent flip-flops, for example. We first need to check for the shortest exit delays using the same sets that we used to check stage delays,

```
timer> shortest
 1st shortest path to all endpins
Rank Total Start pin      First Net      End Net        End pin
  0    4.0 b_rr_ff_b1:CLK b_rr_1_        DEF_NET_48     outp_ff_b1:D
  1    4.1 a_rr_ff_b2:CLK a_rr_2_        DEF_NET_46     outp_ff_b2:D
... 8 similar lines omitted ...
```

The shortest path delay, 4 ns, is between the clock input of a D flip-flop with instance name `b_rr_ff_b1` (call this `X`) and the D input of flip-flop instance name `outp_ff_b1` (`Y`). Due to clock skew, the clock signal may not arrive at both flip-flops simultaneously. Suppose the clock arrives at flip-flop `Y` 3 ns earlier than at flip-flop `X`. The D input to flip-flop `Y` is only stable for $(4-3) = 1$ ns after the clock edge. To check for hold-time violations we thus need to find the clock skew corresponding to each clock-to-D path. This is tedious and normally timing-analysis tools check hold-time requirements automatically, but we shall show the steps to illustrate the process.

13.7.2 Entry Delay

Before we can measure clock skew, we need to analyze the entry delays, including the clock tree. The synthesis tools automatically add I/O pads and the clock cells. This means that extra nodes are automatically added to the netlist with automatically generated names. The EDIF conversion tools may then modify these names. Before we can perform an analysis of entry delays and the clock network delay, we need to find the input node names. By looking for the EDIF `'rename'` construct in the EDIF netlist we can associate the input and output node names in the behavioral Verilog model, `comp_mux_rrr`, and the EDIF names,

```
piron% grep rename comp_mux_rrr_o.edn
          (port (rename a_2_ "a[2]")  (direction INPUT))
... 8 similar lines renaming ports omitted ...
          (net (rename a_rr_0_ "a_rr[0]")  (joined
... 9 similar lines renaming nets omitted ...
piron%
```

Thus, for example, the EDIF conversion program has renamed input port `a[2]` to `a_2_` because the design tools do not like the Verilog bus notation using square brackets. Next we find the connections between the ports and the added I/O cells by looking for `'PAD'` in the Actel format netlist, which indicates a connection to a pad and the pins of the chip, as follows:

```
piron% grep PAD comp_mux_rrr_o.adl
NET DEF_NET_148;  outp_2_, OUTBUF_31:PAD.
NET DEF_NET_151;  outp_1_, OUTBUF_32:PAD.
NET DEF_NET_154;  outp_0_, OUTBUF_33:PAD.
NET DEF_NET_127;  a_2_, INBUF_24:PAD.
NET DEF_NET_130;  a_1_, INBUF_25:PAD.
NET DEF_NET_133;  a_0_, INBUF_26:PAD.
NET DEF_NET_136;  b_2_, INBUF_27:PAD.
NET DEF_NET_139;  b_1_, INBUF_28:PAD.
NET DEF_NET_142;  b_0_, INBUF_29:PAD.
NET DEF_NET_145;  clock, CLKBUF_30:PAD.
piron%
```

This tells us, for example, that the node we called `clock` in our behavioral model has been joined to a node (with automatically generated name) called `CLKBUF_30:PAD`, using a net (connection) named `DEF_NET_145` (again automatically generated). This net is the connection between the node `clock` that is dangling in the behavioral model and the clock-buffer pad cell that the synthesis tools automatically added.

13.7.3 Exit Delay

We now know that the clock-pad input is CLKBUF_30:PAD, so we can find the exit delays (the longest path between clock-pad input and an output) as follows (using the clock-pad input as the start set):

```
timer> startset clockpad
Working startset 'clockpad' contains 0 pins.

timer> addstart CLKBUF_30:PAD
Working startset 'clockpad' contains 2 pins.
```

I shall explain why this set contains two pins and not just one presently. Next, we define the end set and trace the longest exit paths as follows:

```
timer> endset outpad
Working endset 'outpad' contains 3 pins.

timer> longest
 1st  longest path to all endpins
Rank Total Start pin      First Net     End Net       End pin
  0   16.1 CLKBUF_30/U0:PAD DEF_NET_144   DEF_NET_154   OUTBUF_33:PAD
  1   16.0 CLKBUF_30/U0:PAD DEF_NET_144   DEF_NET_151   OUTBUF_32:PAD
  2   16.0 CLKBUF_30/U0:PAD DEF_NET_144   DEF_NET_148   OUTBUF_31:PAD
3 pins
```

This tells us we have three paths from the clock-pad input to the three output pins (outp[0], outp[1], and outp[2]). We can examine the longest exit delay in more detail as follows:

```
timer> expand 0
 1st longest path to OUTBUF_33:PAD (rising) (Rank: 0)
Total Delay Typ Load Macro      Start pin        Net name
 16.1    3.7 Tpd    0  OUTBUF     OUTBUF_33:D      DEF_NET_154
 12.4    4.5 Tpd    1  DF1        outp_ff_b0:CLK   DEF_NET_1530
  7.9    7.9 Tpd   16  CLKEXT_0   CLKBUF_30/U0:PAD DEF_NET_144
```

The input-to-clock delay, t_{IC}, due to the clock-buffer cell (or macro) CLKEXT_0, instance name CLKBUF_30/U0, is 7.9 ns. The clock-to-Q delay, t_{CQ}, of flip-flop cell DF1, instance name outp_ff_b0, is 4.5 ns. The delay, t_{QO}, due to the output buffer cell OUTBUF, instance name OUTBUF_33, is 3.7 ns. The longest path between clock-pad input and the output, t_{CO}, is thus

$$t_{CO} = t_{IC} + t_{CQ} + t_{QO} = 16.1 \text{ ns} . \tag{13.23}$$

This is the critical path and limits the operating frequency to (1 / 16.1 ns) ≈ 62 MHz.

When we created a start set using CLKBUF_30:PAD, the timing analyzer told us that this set consisted of two pins. We can list the names of the two pins as follows:

```
timer> showset clockpad
Pin name                    Net name                    Macro name
CLKBUF_30/U0:PAD            <no net>                    CLKEXT_0
CLKBUF_30/U1:PAD            DEF_NET_145                 CLKTRI_0
2 pins
```

The clock-buffer instance name, CLKBUF_30/U0, is hierarchical (with a '/' hierarchy separator). This indicates that there is more than one instance inside the clock-buffer cell, CLKBUF_30. Instance CLKBUF_30/U0 is the input driver, instance CLKBUF_30/U1 is the output driver (which is disabled and unused in this case).

13.7.4 External Setup Time

Each of the six chip data inputs must satisfy the following set-up equation:

$$t_{SU}\,(\text{external}) > t_{SU}\,(\text{internal}) - (\text{clock delay}) + (\text{data delay}) \qquad (13.24)$$

(where both clock and data delays end at the same flip-flop instance). We find the clock delays in Eq. 13.24 using the clock input pin as the start set and the end set 'clock'. The timing analyzer tells us all 16 clock path delays are the same at 7.9 ns in our design, and the clock skew is thus zero. Actel's clock distribution system minimizes clock skew, but clock skew will not always be zero. From the discussion in Section 13.7.1, we see there is no possibility of internal hold-time violations with a clock skew of zero.

Next, we find the data delays in Eq, 13.24 using a start set of all input pads and an end set of 'gated',

```
timer> longest
... lines omitted ...
 1st  longest path to all endpins
Rank Total Start pin     First Net     End Net       End pin
 10   10.0 INBUF_26:PAD  DEF_NET_1320  DEF_NET_1320  a_r_ff_b0:D
 11    9.7 INBUF_28:PAD  DEF_NET_1380  DEF_NET_1380  b_r_ff_b1:D
 12    9.4 INBUF_25:PAD  DEF_NET_1290  DEF_NET_1290  a_r_ff_b1:D
 13    9.3 INBUF_27:PAD  DEF_NET_1350  DEF_NET_1350  b_r_ff_b2:D
 14    9.2 INBUF_29:PAD  DEF_NET_1410  DEF_NET_1410  b_r_ff_b0:D
 15    9.1 INBUF_24:PAD  DEF_NET_1260  DEF_NET_1260  a_r_ff_b2:D
16 pins
```

We are only interested in the last six paths of this analysis (rank 10–15) that describe the delays from each data input pad (a[0], a[1], a[2], b[0], b[1], b[2]) to the D input of a flip-flop. The maximum data delay, 10 ns, occurs on input buffer instance name INBUF_26 (pad 26); pin INBUF_26:PAD is node a_0_ in the EDIF file or input `a[0]` in our behavioral model. The six t_{SU} (external) equations corresponding to Eq, 13.24 may be reduced to the following worst-case relation:

$$
\begin{aligned}
t_{SU}\,(\text{external})_{\max} &> t_{SU}\,(\text{internal}) - 7.9\text{ ns} + \max\,(9.1\text{ ns}, 10.0\text{ ns}) \\
&> t_{SU}\,(\text{internal}) + 2.1\text{ ns}
\end{aligned}
\qquad (13.25)
$$

We calculated the clock and data delay terms in Eq. 13.24 separately, but timing analyzers can normally perform a single analysis as follows:

$$
t_{SU}\,(\text{external})_{\max} > t_{SU}\,(\text{internal}) - \,(\text{clock delay} - \text{data delay})_{\min}. \qquad (13.26)
$$

Finally, we check that there is no external hold-time requirement. That is to say, we must check that t_{SU} (external) is never negative or

$$
\begin{aligned}
t_{SU}\,(\text{external})_{\min} &> t_{SU}\,(\text{internal}) - \,(\text{clock delay} - \text{data delay})_{\max} > 0 \\
&> t_{SU}\,(\text{internal}) + 1.2\text{ ns} > 0.
\end{aligned}
\qquad (13.27)
$$

Since t_{SU} (internal) is always positive on Actel FPGAs, $t_{SU}\,(\text{external})_{\min}$ is always positive for this design. In large ASICs, with large clock delays, it is possible to have external hold-time requirements on inputs. This is the reason that some FPGAs (Xilinx, for example) have programmable delay elements that deliberately increase the data delay and eliminate irksome external hold-time requirements.

13.8 Formal Verification

Using logic synthesis we move from a behavioral model to a structural model. How are we to know (other than by trusting the logic synthesizer) that the two representations are the same? We have already seen that we may have to alter the original reference model because the HDL acceptable to a synthesis tool is a subset of HDL acceptable to simulators. **Formal verification** can prove, in the mathematical sense, that two representations are equivalent. If they are not, the software can tell us why and how two representations differ.

13.8.1 An Example

We shall use the following VHDL entity with two architectures as an example:[3]

```
entity Alarm is                                                   --1
  port(Clock, Key, Trip : in bit; Ring : out bit);                --2
end Alarm;                                                        --3
```

[3] By one of the architects of the Compass VFormal software, Erich Marschner.

The following behavioral architecture is the **reference model**:

```
architecture RTL of Alarm is
  type States is (Armed, Off, Ringing); signal State : States;
begin
  process (Clock) begin
  if Clock = '1' and Clock'EVENT then
    case State is
      when Off => if Key = '1' then State <= Armed; end if;
      when Armed => if Key = '0' then State <= Off;
                    elsif Trip = '1' then State <= Ringing;
                    end if;
      when Ringing => if Key = '0' then State <= Off; end if;
    end case;
  end if;
  end process;
  Ring <= '1' when State = Ringing else '0';
end RTL;
```

The following synthesized structural architecture is the **derived model**:

```
library cells; use cells.all; // ...contains logic cell models
architecture Gates of Alarm is
component Inverter port(i : in BIT;z : out BIT) ; end component;
component NAnd2 port(a,b : in BIT;z : out BIT) ; end component;
component NAnd3 port(a,b,c : in BIT;z : out BIT) ; end component;
component DFF port(d,c : in BIT; q,qn : out BIT) ; end component;
signal State, NextState : BIT_VECTOR(1 downto 0);
signal s0, s1, s2, s3 : BIT;
begin
  g2: Inverter port map ( i => State(0), z => s1 );
  g3: NAnd2 port map ( a => s1, b => State(1), z => s2 );
  g4: Inverter port map ( i => s2, z => Ring );
  g5: NAnd2 port map ( a => State(1), b => Key, z => s0 );
  g6: NAnd3 port map ( a => Trip, b => s1, c => Key, z => s3 );
  g7: NAnd2 port map ( a => s0, b => s3, z => NextState(1) );
  g8: Inverter port map ( i => Key, z => NextState(0) );
  state_ff_b0: DFF port map
  ( d => NextState(0), c => Clock, q => State(0), qn => open );
  state_ff_b1: DFF port map
  ( d => NextState(1), c => Clock, q => State(1), qn => open );
end Gates;
```

To compare the reference and the derived models (two representations), formal verification performs the following steps: (1) the HDL is parsed, (2) a **finite-state machine compiler** extracts the states present in any sequential logic, (3) a **proof generator** automatically generates formulas to be proved, (4) the **theorem prover** attempts to prove the formulas. The results from the last step are as follows:

```
formulas to be proved:    8
formulas proved VALID:    8
```

By constructing and then proving formulas the software tells us that `architecture RTL` *implies* `architecture  Gates` (implication is the default proof mechanism—we could also have asked if the architectures are exactly equivalent). Next, we shall explore what this means and how formal verification works.

13.8.2 Understanding Formal Verification

The **formulas** to be proved are generated in a separate file of **proof statements**:

```
# axioms
Let Axiom_ref = Axioms Of alarm-rtl
Let Axiom_der = Axioms Of alarm-gates
ProveNotAlwaysFalse (Axiom_ref)
Prove (Axiom_ref => Axiom_der)
# assertions
Let Assert_ref = Asserts Of alarm-rtl
Let Assert_der = Asserts Of alarm-gates
Prove (Axiom_ref => (Assert_ref => Assert_der))
# clocks
Let ClockEvents_ref = Clocks Of alarm-rtl
Let ClockEvents_der = Clocks Of alarm-gates
Let Master__clock_event_ref =
  Value (master__clock'event Of alarm-rtl)
Prove (Axiom_ref => (ClockEvents_ref <=> ClockEvents_der))
# next state of memories
Prove ((Axiom_ref And Master__clock_event_ref) =>
(Transition (state(1) Of alarm-rtl) <=>
Transition (state_ff_b1.t Of alarm-gates)))
Prove ((Axiom_ref And Master__clock_event_ref) =>
(Transition (state(0) Of alarm-rtl) <=>
Transition (state_ff_b0.t Of alarm-gates)))
# validity value of outbuses
Prove (Axiom_ref => (Domain (ring Of alarm-rtl) <=>
Domain (ring Of alarm-gates)))
Prove (Axiom_ref => (Domain (ring Of alarm-rtl) =>
(Value (ring Of alarm-rtl) <=>
Value (ring Of alarm-gates))))
```

Formal verification makes strict use of the terms *axiom* and *assertion*. An **axiom** is an explicit or implicit fact. For example, if a VHDL signal is declared to be type `BIT`, an implicit axiom is that this signal may only take the logic values `'0'` and `'1'`. An **assertion** is derived from a statement placed in the HDL code. For example, the following VHDL statement is an assertion:

```
assert Key /= '1' or Trip /= '1' or NextState = Ringing
  report "Alarm on and tripped but not ringing";
```

A VHDL `assert` statement prints only if the condition is `FALSE`. We know from de Morgan's theorem that `(A+B+C)'=A'B'C'`. Thus, this statement checks for a burglar alarm that does not ring when it is on and we are burgled.

In the proof statements the symbol `'=>'` means **implies**. In logic calculus we write $A \Rightarrow B$ to mean A implies B. The symbol `'<=>'` means **equivalence**, and this is stricter than implication. We write $A \Leftrightarrow B$ to mean: A is equivalent to B. Table 13.13 show the truth tables for these two logic operators.

TABLE 13.13 Implication and equivalence.

A	B	A => B	A <=> B
F	F	T	T
F	T	T	F
T	F	F	F
T	T	T	T

13.8.3 Adding an Assertion

If we include the `assert` statement from the previous section in `architecture RTL` and repeat formal verification, we get the following message from the FSM compiler:

```
<E> Assertion may be violated
SEVERITY: ERROR
REPORT: Alarm on and tripped but not ringing
FILE: .../alarm-rtl3.vhdl
FSM: alarm-rtl3
STATEMENT or DECLARATION: line8
.../alarm-rtl3.vhdl (line 8)
Context of the message is:
(key And trip And memoryofdriver__state(0))
```

This message tells us that the `assert` statement that we included may be triggered under a certain condition: `(key And trip And state(0))`. The prefix `'memoryofdriver__'` is used by the theorem prover to refer to the memory element used for `state(0)`. The state `'off'` in the reference model corresponds to `state(0)` in the encoding that the finite-state machine compiler has used (and also to `state(0)` in the derived model). From this message we can isolate the problem to the following `case` statement (the line numbers follow the original code in `architecture RTL`):

```
case State is                                                   --6
  when Off => if Key = '1' then State <= Armed; end if;         --7
  when Armed => if Key = '0' then State <= Off;                 --8
                elsif Trip = '1' then State <= Ringing;         --9
                end if;                                         --10
  when Ringing => if Key = '0' then State <= Off; end if;       --11
end case;                                                       --12
```

When we start in state `Off` and the two inputs are `Trip = '1'` and `Key = '1'`, we go to state `Armed`, and not to state `Ringing`. On the subsequent clock cycle we will go state `Ringing`, but only if `Trip` does not change. Since we have all seen "Mission Impossible" and the burglar who exits the top-secret computer room at the Pentagon at the exact moment the alarm is set, we know this is perfectly possible and the software is warning us of this fact. Continuing on, we get the following results from the theorem prover:

```
Prove (Axiom_ref => (Assert_ref => Assert_der))
Formula is NOT VALID
But is VALID under Assert Context of alarm-rtl3
```

We included the `assert` statement in the reference model (`architecture RTL`) but not in the derived model (`architecture Gates`). Now we are really mixed up: The `assertion` statement in the reference model says one thing, but the `case` statement in the reference model describes another. The theorem prover retorts: "The axioms of the reference model do not imply that the assertions of the reference model imply the assertions of the derived model." Translation: "These two architectures differ in some way." However, if we assume that the assertion is true (despite what the `case` statement says) then the formula is true. The prover is also saying: "Make up your mind, you cannot have it both ways." The prover goes on to explain the differences between the two representations:

```
***Difference is:
(Not state(1) And key And state(0) And trip)
There are 1 cubes and 4 literals in the complete equation

***Local Variable Assert_der is:
Not key Or Not state(0) Or Not trip
There are 3 cubes and 3 literals in the complete equation

***Local Variable Assert_ref is: 1

***Local Variable Axiom_ref is:
Not state(1) Or Not state(0)
```

```
There are 2 cubes and 2 literals in the complete equation

formulas to be proved:    8
formulas proved VALID:    7
formulas VALID under assert context of der.model:    1
```

Study these messages hard and you will see that the differences between the two models are consistent with our explanation.

13.8.4 Completing a Proof

To fix the problem we change the code as follows:

```
...
case State is
  when Off =>    if Key = '1' then
                   if Trip = '1' then NextState <= Ringing;
                   else NextState <= Armed;
                   end if;
                 end if;
  when Armed => if Key = '0' then NextState <= Off;
                 elsif Trip = '1' then NextState <= Ringing;
                 end if;
  when Ringing => if Key = '0' then NextState <= Off; end if;
end case;
...
```

This results in a minor change in the synthesized netlist,

```
g2: Inverter port map ( i => State(0), z => s1 );
g3: NAnd2    port map ( a => s1, b => State(1), z => s2 );
g4: Inverter port map ( i => s2, z => Ring );
g5: NAnd2    port map ( a => State(1), b => Key, z => s0 );
g6: NAnd3    port map ( a => Trip, b => s1, c => Key, z => s3 );
g7: NAnd2    port map ( a => s0, b => s3, z => NextState(1) );
g8: Inverter port map ( i => Key, z => NextState(0) );
state_ff_b0: DFF port map ( d => NextState(0), c => Clock, q =>
State(0), qn => open );
state_ff_b1: DFF port map ( d => NextState(1), c => Clock, q =>
State(1), qn => open );
```

Repeating the formal verification confirms and formally proves that the derived model will operate correctly. Strictly, we say that the operation of the derived model is implied by the reference model.

13.9 Switch-Level Simulation

The **switch-level simulator** is a more detailed level of simulation than we have discussed so far. Figure 13.1 shows the circuit schematic of a **true single-phase flip-flop** using **true single-phase clocking** (**TSPC**). TSPC has been used in some full-custom ICs to attempt to save area and power.

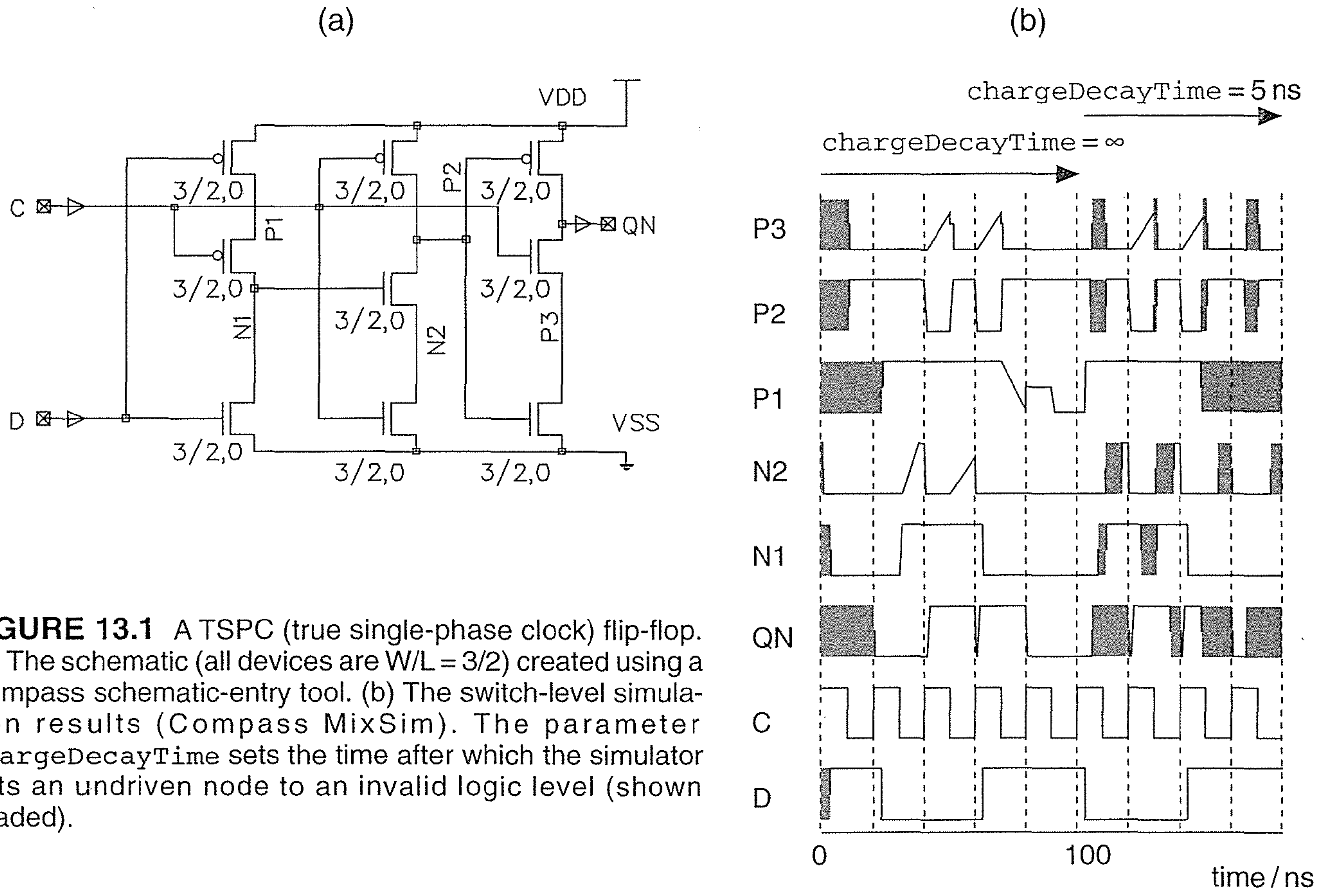


FIGURE 13.1 A TSPC (true single-phase clock) flip-flop. (a) The schematic (all devices are W/L = 3/2) created using a Compass schematic-entry tool. (b) The switch-level simulation results (Compass MixSim). The parameter `chargeDecayTime` sets the time after which the simulator sets an undriven node to an invalid logic level (shown shaded).

In a CMOS logic cell every node is driven to a strong `'1'` or a strong `'0'`. This is not true in TSPC, some nodes are left floating, so we ask the switch-level simulator to model charge leakage or charge decay (normally we need not worry about this low-level device issue). Figure 13.1 shows the waveform results. After five clock cycles, or 100 ns, we set the charge decay time to 5 ns. We notice two things. First, some of the node waveforms have values that are between logic `'0'` and `'1'`. Second, there are shaded areas on some node waveforms that represent the fact that, during the period of time marked, the logic value of the node is unknown. We can see that initially, before $t = 100$ ns (while we neglect the effects of charge decay), the circuit functions as a flip-flop. After $t = 100$ ns (when we begin including the effects of charge decay), the simulator tells us that this circuit may not function correctly. It is unlikely that all the charge would leak from a node in 5 ns, but we could not stop the clock in a design that uses a TSPC flip-flop. In ASIC design we do not use dangerous techniques such as TSPC and therefore do not normally need to use switch-level simulation.

A switch-level simulator keeps track of voltage levels as well as logic levels, and it may do this in several ways. The simulator may use a large possible set of discrete values or the value of a node may be allowed to vary continuously.

13.10 Transistor-Level Simulation

Sometimes we need to simulate a logic circuit with more accuracy than provided by switch-level simulation. In this case we turn to simulators that can solve circuit equations exactly, given models for the nonlinear transistors, and predict the analog behavior of the node voltages and currents in continuous time. This type of **transistor-level simulation** or **circuit-level simulation** is costly in computer time. It is impossible to simulate more than a few hundred logic cells using a circuit-level simulator. Virtually all circuit-level simulators used for ASIC design are commercial versions of the **SPICE** (or **Spice**, **Simulation Program with Integrated Circuit Emphasis**) developed at UC Berkeley.

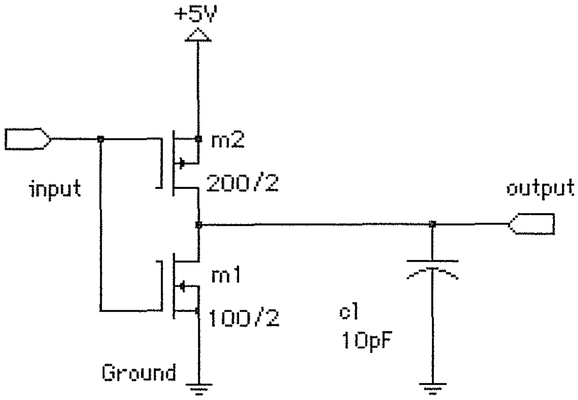


FIGURE 13.2 Output buffer (OB.IN) schematic (created using Capilano's DesignWorks)

13.10.1 A PSpice Example

Figure 13.2 shows the schematic for the output section of a CMOS I/O buffer driving a 10 pF output capacitor representing an off-chip load. The **PSpice** input file that follows is called a **deck** (from the days of punched cards):

```
OB September 5, 1996 17:27
.TRAN/OP 1ns 20ns
.PROBE
 cl  output  Ground  10pF
 VIN  input  Ground   PWL(0us 5V 10ns 5V 12ns 0V 20ns 0V)
 VGround  0  Ground   DC 0V
 Vdd  +5V  0   DC 5V
 m1  output  input  Ground Ground NMOS W=100u L=2u
 m2  output  input  +5V +5V PMOS W=200u L=2u
.model nmos nmos level=2 vto=0.78 tox=400e-10 nsub=8.0e15 xj=-0.15e-6
+ ld=0.20e-6 uo=650 ucrit=0.62e5 uexp=0.125 vmax=5.1e4 neff=4.0
+ delta=1.4 rsh=37 cgso=2.95e-10 cgdo=2.95e-10 cj=195e-6 cjsw=500e-12
+ mj=0.76 mjsw=0.30 pb=0.80
.model pmos pmos level=2 vto=-0.8 tox=400e-10 nsub=6.0e15 xj=-0.05e-6
+ ld=0.20e-6 uo=255 ucrit=0.86e5 uexp=0.29 vmax=3.0e4 neff=2.65
+ delta=1 rsh=125 cgso=2.65e-10 cgdo=2.65e-10 cj=250e-6 cjsw=350e-12
+ mj=0.535 mjsw=0.34 pb=0.80
.end
```

Figure 13.3 shows the input and output waveforms as well as the current flowing in the devices.We can quickly check our circuit simulation results as follows. The total charge transferred to the 10 pF load capacitor as it charges from 0 V to 5 V is 50 pC (equal to 5 V × 10 pF). This total charge should be very nearly equal to the integral of the drain current of the pull-up (*p*-channel) transistor I_L(m2). We can get a quick estimate of the integral of the current by approximating the area under the waveform for id(m2) in Figure 13.3 as a triangle—half the base (about 12 ns) multiplied by the height (about 8 mA), so that

$$\int_{10\text{ns}}^{22\text{ns}} I_L(m2)\,dt = 0.5\,(8\text{mA})\,(12\text{ns}) \approx 50\text{pC} = 5\,(10\text{pF})\,. \qquad (13.28)$$

Notice that the two estimates for the transferred charge are equal.

Next, we can check the time derivative of the pull-up current. (We can also do this by using the Probe program and requesting a plot of did(m2); the symbol dn represents the time derivative of quantity n for Probe. The symbol id(m2) requests Probe to plot the drain current of m2.) The maximum derivative should be roughly equal to the maximum change of the drain current ($\Delta I_L(m2) = 8$ mA) divided by the time taken for that change (about $\Delta t = 2$ ns from Figure 13.3) or

$$\frac{|\Delta I_L(m2)|}{\Delta t} = \frac{8\text{mA}}{2\text{ns}} = 4 \times 10^6\text{As}^{-1}\,. \qquad (13.29)$$

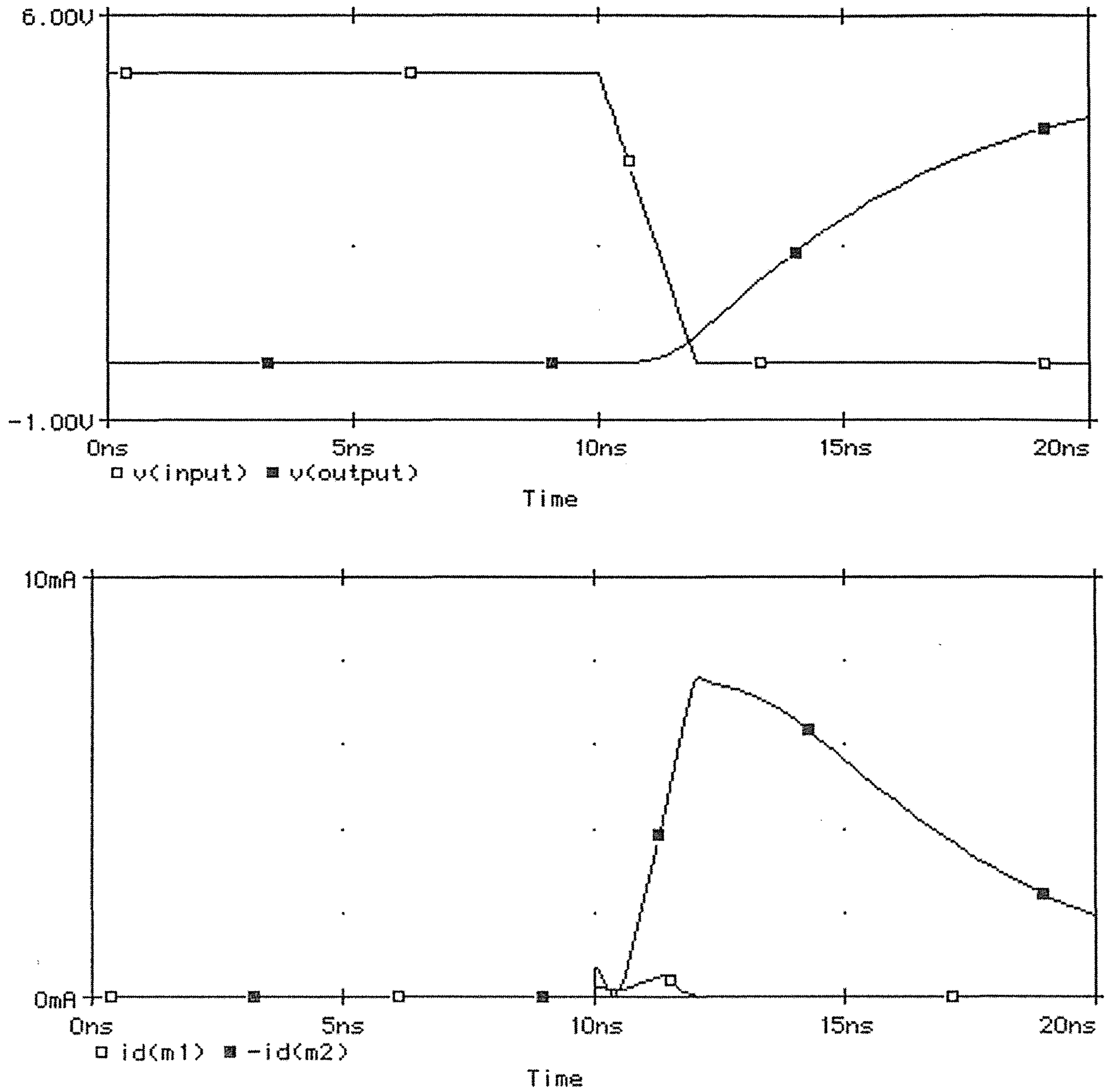


FIGURE 13.3 Output Buffer (OB.IN). (Top) The input and output voltage waveforms. (Bottom) The current flowing in the drains of the output devices.

The large time derivative of the device current, here 4 MAs^{-1}, causes problems in high-speed CMOS I/O. This sharp change in current must flow in the supply leads to the chip, and through the inductance associated with the bonding wires to the chip which may be of the order of 10 nanohenrys. An electromotive force (emf), V_P, will be generated in the inductance as follows,

$$V_P = -L\frac{dI}{dt} = -10\mathrm{nH}\,(4\times10^6)\,\mathrm{As}^{-1} = -40\ \mathrm{mV}\,. \tag{13.30}$$

The result is a glitch in the power supply voltage during the buffer output transient. This is known as **supply bounce** or **ground bounce**. To limit the amount of bounce we may do one of two things:

1. Limit the power supply lead inductance (minimize L)
2. Reduce the current pulse (minimize dI/dt)

We can work on the first solution by careful design of the packages and by using parallel bonding wires (inductors add in series, reduce in parallel).

13.10.2 SPICE Models

Table 13.14 shows the SPICE parameters for the typical 0.5 μm CMOS process (0.6 μm drawn gate length), G5, that we used in Section 2.1. These LEVEL = 3 parameters may be used with Spice3, PSpice, and HSPICE (see also Table 2.1 and Figure 2.4).

There are several levels of the SPICE MOSFET models, the following is a simplified overview (a huge number of confusing variations, fixes, and options have been added to these models—see Meta Software's HSPICE User's Manual, Vol. II, for a comprehensive description [1996]):

1. LEVEL = 1 (**Schichman–Hodges model**) uses the simple square-law I_{DS}–V_{DS} relation we derived in Section 2.1 (Eqs. 2.9 and 2.12).
2. LEVEL = 2 (**Grove–Frohman model**) uses the 3/2 power equations that result if we include the variation of threshold voltage across the channel.
3. LEVEL = 3 (**empirical model**) uses empirical equations.
4. The UCB **BSIM1** model (~1984, PSpice LEVEL = 4, HSPICE LEVEL = 13) focuses on modeling observed device data rather than on device physics. A commercial derivative (HSPICE LEVEL = 28) is widely used by ASIC vendors.
5. The UCB **BSIM2** model (~1991, the commercial derivative is HSPICE LEVEL = 39) improves modeling of subthreshold conduction.
6. The UCB **BSIM3** model (~1995, the commercial derivative is HSPICE LEVEL = 49) corrects potential nonphysical behavior of earlier models.

Table 13.15 shows the BSIM1 parameters (in the PSpice LEVEL = 4 format) for the G5 process. The **Berkeley short-channel IGFET model** (**BSIM**) family models capacitance in terms of charge. In Sections 2.1 and 3.2 we treated the gate–drain

TABLE 13.14 SPICE transistor model parameters (LEVEL = 3).

SPICE parameter	n-channel value	p-channel value (if different)	Units	Explanation
CGBO	4.0E-10	3.8E-10	Fm^{-1}	Gate–bulk overlap capacitance (CGBoh, not CGBzero)
CGDO	3.0E-10	2.4E-10	Fm^{-1}	Gate–drain overlap capacitance (CGDoh, not CGDzero)
CGSO	3.0E-10	2.4E-10	Fm^{-1}	Gate–source overlap capacitance (CGSoh, not CGSzero)
CJ	5.6E-4	9.3E-4	Fm^{-2}	Junction area capacitance
CJSW	5E-11	2.9E-10	Fm^{-1}	Junction sidewall capacitance
DELTA	0.7	0.29	m	Narrow-width factor for adjusting threshold voltage
ETA	3.7E-2	2.45E-2	1	Static-feedback factor for adjusting threshold voltage
GAMMA	0.6	0.47	$\text{V}^{0.5}$	Body-effect factor
KAPPA	2.9E-2	8	V^{-1}	Saturation-field factor (channel-length modulation)
KP	2E-4	4.9E-5	AV^{-2}	Intrinsic transconductance (μC_{ox}, not $0.5\mu C_{ox}$)
LD	5E-8	3.5E-8	m	Lateral diffusion into channel
LEVEL	3		none	Empirical model
MJ	0.56	0.47	1	Junction area exponent
MJSW	0.52	0.50	1	Junction sidewall exponent
NFS	6E11	6.5E11	$\text{cm}^{-2}\text{V}^{-1}$	Fast surface-state density
NSUB	1.4E17	8.5E16	cm^{-3}	Bulk surface doping
PB	1	1	V	Junction area contact potential
PHI	0.7		V	Surface inversion potential
RSH	2		Ω/square	Sheet resistance of source and drain
THETA	0.27	0.29	V^{-1}	Mobility-degradation factor
TOX	1E-8		m	Gate-oxide thickness
TPG	1	-1	none	Type of polysilicon gate
U0	550	135	$\text{cm}^2\text{V}^{-1}\text{s}^{-1}$	Low-field bulk carrier mobility (Uzero, not Uoh)
XJ	0.2E-6		m	Junction depth
VMAX	2E5	2.5E5	ms^{-1}	Saturated carrier velocity
VTO	0.65	-0.92	V	Zero-bias threshold voltage (VTzero, not VToh)

Meta Software's HSPICE User's Manual [1996], p. 15-36 and pp.16-13 to 16-15, explains these parameters. Note that m or M both represent milli or 10^{-3} in SPICE, not mega or 10^{6} (u or U = micro or 10^{-6} and so on).

TABLE 13.15 PSpice parameters for process G5 (PSpice LEVEL = 4).

```
.MODEL NM1 NMOS LEVEL=4
+ VFB=-0.7, LVFB=-4E-2, WVFB=5E-2
+ PHI=0.84, LPHI=0, WPHI=0
+ K1=0.78, LK1=-8E-4, WK1=-5E-2
+ K2=2.7E-2, LK2=5E-2, WK2=-3E-2
+ ETA=-2E-3, LETA=2E-02, WETA=-5E-3
+ MUZ=600, DL=0.2, DW=0.5
+ U0=0.33, LU0=0.1, WU0=-0.1
+ U1=3.3E-2, LU1=3E-2, WU1=-1E-2
+ X2MZ=9.7, LX2MZ=-6, WX2MZ=7
+ X2E=4.4E-4, LX2E=-3E-3, WX2E=9E-4
+ X3E=-5E-5, LX3E=-2E-3, WX3E=-1E-3
+ X2U0=-1E-2, LX2U0=-1E-3, WX2U0=5E-3
+ X2U1=-1E-3, LX2U1=1E-3, WX2U1=-7E-4
+ MUS=700, LMUS=-50, WMUS=7
+ X2MS=-6E-2, LX2MS=1, WX2MS=4
+ X3MS=9, LX3MS=2, WX3MS=-6
+ X3U1=9E-3, LX3U1=2E-4, WX3U1=-5E-3
+ TOX=1E-2, TEMP=25, VDD=5
+ CGDO=3E-10, CGSO=3E-10, CGBO=4E-10
+ XPART=1
+ N0=1, LN0=0, WN0=0
+ NB=0, LNB=0, WNB=0
+ ND=0, LND=0, WND=0
* n+ diffusion
+ RSH=2.1, CJ=3.5E-4, CJSW=2.9E-10
+ JS=1E-8, PB=0.8, PBSW=0.8
+ MJ=0.44, MJSW=0.26, WDF=0
*, DS=0
```

```
.MODEL PM1 PMOS LEVEL=4
+ VFB=-0.2, LVFB=4E-2, WVFB=-0.1
+ PHI=0.83, LPHI=0, WPHI=0
+ K1=0.35, LK1=-7E-02, WK1=0.2
+ K2=-4.5E-2, LK2=9E-3, WK2=4E-2
+ ETA=-1E-2, LETA=2E-2, WETA=-4E-4
+ MUZ=140, DL=0.2, DW=0.5
+ U0=0.2, LU0=6E-2, WU0=-6E-2
+ U1=1E-2, LU1=1E-2, WU1=7E-4
+ X2MZ=7, LX2MZ=-2, WX2MZ=1
+ X2E= 5E-5, LX2E=-1E-3, WX2E=-2E-4
+ X3E=8E-4, LX3E=-2E-4, WX3E=-1E-3
+ X2U0=9E-3, LX2U0=-2E-3, WX2U0=2E-3
+ X2U1=6E-4, LX2U1=5E-4, WX2U1=3E-4
+ MUS=150, LMUS=10, WMUS=4
+ X2MS=6, LX2MS=-0.7, WX2MS=2
+ X3MS=-1E-2, LX3MS=2, WX3MS=1
+ X3U1=-1E-3, LX3U1=-5E-4, WX3U1=1E-3
+ TOX=1E-2, TEMP=25, VDD=5
+ CGDO=2.4E-10, CGSO=2.4E-10, CGBO=3.8E-10
+ XPART=1
+ N0=1, LN0=0, WN0=0
+ NB=0, LNB=0, WNB=0
+ ND=0, LND=0, WND=0
* p+ diffusion
+ RSH=2, CJ=9.5E-4, CJSW=2.5E-10
+ JS=1E-8, PB=0.85, PBSW=0.85
+ MJ=0.44, MJSW=0.24, WDF=0
*, DS=0
```

PSpice LEVEL = 4 is almost exactly equivalent to the UCB BSIM1 model, and closely equivalent to the HSPICE LEVEL = 13 model (see Table 14-1 and pp. 16–86 to 16-89 in Meta Software's HSPICE User's Manual [1996].

capacitance, C_{GD}, for example, as if it were a **reciprocal capacitance**, and could be written assuming there was charge associated with the gate, Q_G, and the drain, Q_D, as follows:

$$C_{GD} = -\frac{\delta Q_G}{\delta V_D} = C_{DG} = -\frac{\delta Q_D}{\delta V_G} . \tag{13.31}$$

Equation 13.31 (the **Meyer model**) would be true if the gate and drain formed a parallel plate capacitor and $Q_G = -Q_D$, but they do not. In general, $Q_G \neq -Q_D$ and Eq. 13.31 is not true. In an MOS transistor we have four regions of charge: Q_G

(gate), Q_D (channel charge associated with the drain), Q_S (channel charge associated with the drain), and Q_B (charge in the bulk depletion region). These charges are not independent, since

$$Q_G + Q_D + Q_S + Q_B = 0. \tag{13.32}$$

We can form a 4×4 matrix, **M**, whose entries are $\partial Q_i / \partial V_j$, where $V_j = V_G$, V_S, V_D, and V_B. Then $C_{ii} = M_{ii}$ are the terminal capacitances; and $C_{ij} = -M_{ij}$, where $i \neq j$, is a **transcapacitance**. Equation 13.32 forces the sum of each column of **M** to be zero. Since the charges depend on voltage differences, there are only three independent voltages ($V_{GB,}$ V_{DB}, and V_{SB}, for example) and each row of **M** must sum to zero. Thus, we have nine (=16 − 7) independent entries in the matrix **M**. In general, C_{ij} is not necessarily equal to C_{ji}. For example, using PSpice and a LEVEL = 4 BSIM model, there are nine independent partial derivatives, printed as follows:

```
Derivatives of gate (dQg/dVxy) and bulk (dQb/dVxy) charges
DQGDVGB        1.04E-14
DQGDVDB       -1.99E-15
DQGDVSB       -7.33E-15
DQDDVGB       -1.99E-15
DQDDVDB        1.99E-15
DQDDVSB        0.00E+00
DQBDVGB       -7.51E-16
DQBDVDB        0.00E+00
DQBDVSB       -2.72E-15
```

From these derivatives we may compute six **nonreciprocal capacitances**:

$$\begin{aligned}
C_{GB} &= \partial Q_G / \partial V_{GB} + \partial Q_G / \partial V_{DB} + \partial Q_G / \partial V_{SB} \\
C_{BG} &= -\partial Q_B / \partial V_{GB} \\
C_{GS} &= -\partial Q_G / \partial V_{SB} \\
C_{SG} &= \partial Q_G / \partial V_{GB} + \partial Q_B / \partial V_{GB} + \partial Q_D / \partial V_{GB} \\
C_{GD} &= -\partial Q_G / \partial V_{DB} \\
C_{DG} &= -\partial Q_D / \partial V_{GB}
\end{aligned} \tag{13.33}$$

and three terminal capacitances:

$$\begin{aligned}
C_{GG} &= \partial Q_G / \partial V_{GB} \\
C_{DD} &= \partial Q_D / \partial V_{DB} \\
C_{SS} &= -(\partial Q_G / \partial V_{SB} + \partial Q_B / \partial V_{SB} + \partial Q_D / \partial V_{SB})
\end{aligned} \tag{13.34}$$

Nonreciprocal transistor capacitances cast a cloud over our analysis of gate capacitance in Section 3.2, but the error we made in neglecting this effect is small compared to the approximations we made in the sections that followed. Even though we now find the theoretical analysis was simplified, the conclusions in our treatment of logical effort and delay modeling are still sound. Sections 7.3 and 9.2 in the book on transistor modeling by Tsividis [1987] describe nonreciprocal capacitance in detail. Pages 15-42 to 15-44 in Vol. II of Meta Software's HSPICE User Manual [1996] also gives an explanation of transcapacitance.

13.11 Summary

We discussed the following types of simulation (from high level to low level):

- Behavioral simulation includes no timing information and can tell you only if your design will not work.
- Prelayout simulation of a structural model can give you estimates of performance, but finding a critical path is difficult because you need to construct input vectors to exercise the model.
- Static timing analysis is the most widely used form of simulation. It is convenient because you do not need to create input vectors. Its limitations are that it can produce false paths—critical paths that may never be activated.
- Formal verification is a powerful adjunct to simulation to compare two different representations and formally prove if they are equal. It cannot prove your design will work.
- Switch-level simulation is required to check the behavior of circuits that may not always have nodes that are driven or that use logic that is not complementary.
- Transistor-level simulation is used when you need to know the analog, rather than the digital, behavior of circuit voltages.

There is a trade-off in accuracy against run time. The high-level simulators are fast but are less accurate.

13.12 Problems

* = Difficult, ** = Very difficult, *** = Extremely difficult

13.1 (Errors, 30 min.) Change `a<=b` to `a>=b` in line 4 in module `reference` in Section 13.2.1. Simulate the testbench (write models for the five logic cell models not shown in Section 13.2.1). How many errors are there, and why? *Answer:* 56.

13.2 (False paths, 15 min.) The following code forces an output pin to a constant value. Perform a timing analysis on this model and comment on the results.

```
module check_critical_path_2 (a, z);                                     //1
input a; output z; supply1 VDD; supply0 VSS;                             //2
nd02d0 b1_i3 (.a1(a), .a2(VSS), .zn(z)); // 2-input NAND                 //3
endmodule                                                                //4
```

13.3 (Timing loops, 30 min.) The following code models a set–reset latch with feedback to implement a memory element. Perform a timing analysis on this model and comment on the results.

```
module check_critical_path_3 (s, r, q, qn);                              //1
input s, r; output q, qn; supply1 VDD; supply0 VSS;                      //2
nr02d0 b1_i1 (.a1(s), .a2(qn), .zn(q)); // 2-input NOR                   //3
nr02d0 b1_i2 (.a1(r), .a2(q), .zn(qn)); // 2-input NOR                   //4
endmodule                                                                //5
```

13.4 (Simulation script, 30 min.) Perform a gate-level simulation of the comparator/MUX in Section 13.2.3. Write a script to set input values and so on.

13.5 (Verilog loops, 30 min.) Change the index from `integer` to `reg` (width three) in each loop in `testbench.v` from Section 13.2. Explain the simulation result.

13.6 (Verilog time, 30 min.) Remove `'#1'` from line 15 in `testbench.v` from Section 13.2. Explain carefully the simulation result.

13.7 (Infinite loops, 30 min.) Construct an HDL program that loops infinitely on a UNIX machine (with no output file!) and explain how the following helps:

```
<293> ps
  PID TT STAT   TIME COMMAND
...
28920 p1 R      0:30 verilog infinite_loop.v
...
<294> kill -9 28920
```

13.8 (Verilog graphics, 30 min.) Experiment with graphical waveform dumps from Verilog. For example, in VeriWell you need to include the following statement:

```
initial $dumpvars;
```

The file Dump file `veriwell.dmp` should appear. Next, select `File...`, then `Convert Dumpvar...` Write a cheat sheet on how to use and display simulation results from a hierarchical model.

13.9 (Unknowns, 30 min.) Explain, using truth tables, the function of primitive `G6` in module `mx21d1` from Section 13.2.1. *Hint:* Consider unknown propagation. Eliminate primitive `G6` as follows and use simulation to compare the two models:

```
  not G3(N3,s); and G4(N4,i0,N3), G5(N5,s,i1); or G7(z,N4,N5);
```

13.10 (Data books, 10 min.) Explain carefully what you safely can and cannot deduce from the data book figures in Table 13.16.

TABLE 13.16 Input capacitances—AOIabcd family (Problem 13.10).

	1X drive	2X drive	4X drive
Area	0.034 pF	0.069 pF	0.138 pF
Performance	0.145 pF	0.294 pF	0.588 pF

13.11 (Synthesis, 30 min.) Synthesize `comp_mux_rrr.v` in Section 13.7. What type and how many sequential elements result? *Answer:* 16.

13.12 (Place and route, 60 min.) Route both `comp_mux.v` (Section 13.2) and `comp_mux_rrr.v` (Section 13.7) using an FPGA. What fraction of the chip is used? *Answer:* For an Actel 1415 FPGA, `comp_mux_rrr.v` uses about 10 percent of the available logic.

13.13 (Timing analysis, 60 min.) Perform timing analysis on a routed version of `comp_mux.v` from Section 13.2. Use worst-case commercial conditions.

13.14 (***NAND gate delay, 120 min.) The following example of a six-input NAND gate illustrates the difference between transistor-level and other levels of simulation. A designer once needed a delay element (do not ask why!). Looking at the data book they found a six-input NAND gate had the right delay, but they did not know what to do with the other five inputs. So they tied all six inputs together. This is a horrendous error, but why? *Hint:* You might have to simulate a structural model using both digital simulation and a circuit-level simulation in order to explain.

13.15 (Logic systems, 30 min.) Compare the 12 value system of Table 13.5 with the IEEE 1164 standard and explain: Which logic values are equivalent in both systems, which logic values have no equivalents, and why there is a difference in the number of values (12 versus 9) when both systems have the same number of logic levels and logic strengths?

13.16 (VHDL overloaded functions, 30 min.) Write a definition for the type `stdlogic_table` used in the `and` function in Section 13.3.2,

```
constant and_table:stdlogic_table
```

Compile, simulate, and test the `and` function.

13.17 (**Scheduling transactions in VHDL, 60 min.) (From an example in the VHDL LRM.) Consider this assignment to an integer `S` in a VHDL process:

```
S <= reject 15 ns inertial 12 after 20 ns, 18 after 41 ns;
```

Assume that at the time this signal assignment is executed, the driver for `s` in the process has the following contents (the first entry is the current driving value):

1	2	2	12	5	8
now	+3ns	+12ns	+13ns	+20ns	+42ns

This is called the **projected output waveform** (times are relative to the current time). The LRM states the rule for updating a projected output waveform consists of the deletion of zero or more previously computed transactions (called old transactions) from the projected output waveform, and the addition of the new transactions, as follows:

1. All old transactions that are projected to occur at or after the time at which the earliest new transaction is projected to occur are deleted from the projected output waveform.
2. The new transactions are then appended to the projected output waveform in the order of their projected occurrence.

If the initial delay is inertial delay, the projected output waveform is further modified as follows:

1. All of the new transactions are marked.
2. An old transaction is marked if the time at which it is projected to occur is less than the time at which the first new transaction is projected to occur minus the pulse rejection limit.
3. For each remaining unmarked, old transaction, the old transaction is marked if it immediately precedes a marked transaction and its value component is the same as that of the marked transaction.
4. The transaction that determines the current value of the driver is marked.
5. All unmarked transactions (all of which are old transactions) are deleted from the projected output waveform.

For the purposes of marking transactions, any two successive null transactions in a projected output waveform are considered to have the same value component. Using these rules compute the new projected output waveform.

13.18 (***`awk`, 120 min.) Write an `awk` program with the following specification to compare two simulations:

```
# program to check two files with the format:
#    time signal value
# to check agreement within time tolerance delta (by default 0.1)
#
# Use: check file1 file2 [delta]
```

13.19 (VITAL, 60 min.) Simulate the model, `sdf_testbench`, shown in Section 13.5.5, with and without back-annotation timing information in `SDF_b.sdf`.

13.20 (Formal verification, 60 min.) Write a cheat sheet explaining how to run your formal verification tool. Repeat the example in Section 13.8.1.

13.21 (***Beetle problem) (Based on a problem by Seitz.) A planet has many geological gem mazes: A maze covers a square km or so, on a 10 mm grid; a maze cell is 10 mm by 10 mm and gems lie at cell centers; there is a path from every maze cell to every other; on average one in 64 cells has an overhead opening; on average one in seven cells has a single gem; there are no gems under overhead openings.

You are to design a gem-mining beetle ASIC with the following inputs: a nominal 1 MHz single-phase clock, `CLK`; wall sensors: `WL`, `WR`, `WF`, `WB` (wall to left/right/forward/behind); light sensors: `LL`, `LR`, `LF`, `LB` (light left/right/forward/behind); low-battery indicator: `BLOW`; gem sensor: `GEM` (directly over a gem); opening sensor: `OPEN` (when under an opening).

All signals are active high and the light sensor outputs are mutually exclusive. The beetle ASIC must produce the following (mutually exclusive) signals: move forward, `MF`; move backward, `MB`; turn 90 degrees clockwise, `TC`; turn 90 degrees anticlockwise, `TA`; pick up a gem, `PICKUP`; throw gem up and out of overhead opening, `THROWUP`; jump up to surface and shut down, `SHUTUP`.

The beetle specifications and limitations are as follows: Beetles are dropped into the maze to find the gems; beetles must find gems and carry them to an opening; beetles can eject gems through openings; beetles can carry only one gem at a time.

A beetle move is one of the following: taking one step (moving to an adjacent cell), turning 90 degrees, picking up a gem, ejecting a gem, jumping out of opening—all take the same time and energy. A battery can provide energy for about 200 moves before the low-battery signal comes on. After the low-battery warning is signaled the battery has energy for 50 moves to find an overhead opening, and the beetle must then eject itself for recharging. The cost of the beetle determines that we would like the probability of losing a beetle be below 0.01.

The following describes a state machine to drive a beetle. Jim Rowson used a state-machine language that he developed—along with the first CAD tool that could automatically create state machines:

```
# Jim Rowson's beetle
sm smbtl;
clock clk;
reset res                  --> resetState;
inputs WL WL WR WB GEM LF LL LR LB OPEN BLOW;
outputs MF=0 MB=0 TC=0 TA=0 PICKUP=0 THROWUP=0 SHUTUP=0;
outputs haveAgem SHUTUP;
let getout = (BLOW|haveAgem) & (LL|LF|LR|LB);

state resetState           --> searchState haveAgem=0 SHUTUP=0;
state searchState
  BLOW & OPEN              --> jumpState,
  haveAgem & OPEN          --> ejectstate,
   getout & LL              --> turnLstate,
   getout & LF              --> goFwdState,
   getout & LR              --> turnRstate,
   getout & LB              --> turnAroundState,
   !haveAgem & GEM          --> getGemState,
   !WL                      --> turnLState,
   !WF                      --> goFwdState,
   !WR                      --> turnRState,
   !WB                      --> turnAroundState;
state goFwdState            --> MF searchState;
state turnLState            --> TA goFwdState;
state turnRState            --> TC goFwdState;
state turnAroundState       --> TC turnAgainState;
state ejectState            --> THROWUP !haveAgem searchState;
state jumpState             --> SHUTUP shutdownState;
state getGemState           --> PICKUP haveAgem searchState;
state shutDownState         --> SHUTUP shutDownState;
state turnAgainState        --> TC searchState;
end
```

a. (120 min.) Draw Jim's state machine diagram and translate it to an HDL.

b. (120 min.) Build a model for the maze that will work with Jim's design.

c. (120 min.) Simulate the operation of Jim's beetle using your maze model.

d. (Hours) Can you do better than Jim?

13.22 (Switch-level simulation, 120 min.) Perform the switch-level simulation shown in Section 13.9.

13.23 (**Simulation, 60 min.) (From a question posed by Ray Ryan to the VITAL timing group.) Suppose we have a two-input NAND gate (inputs I1 and I2, and output Q) with separate path delays from I1 to Q and from I2 to Q with delays as follows:

```
tpd_I1_Q =>
(  tr01 => 10 ns, -- falling I1 -> rising  Q
   tr10 => 7 ns ) -- rising  I1 -> falling Q
tpd_I2_Q =>
(  tr01 => 5 ns, -- falling I2 -> rising  Q
   tr10 => 3 ns ) -- rising  I1 -> falling Q
```

a. For inputs: (I1:0->1, 9 ns; I2:0->1, 10 ns), should Q fall at:

12 ns, 13 ns, 16 ns, 17 ns or other?

b. For inputs: (I2:0->1, 9 ns; I1:0->1, 10 ns), should Q fall at:

12 ns, 13 ns, 16 ns, 17 ns or other?

c. For inputs: (I2:0->1, 10 ns; I1:0->1, 10 ns), should Q fall at:

13 ns, 15 ns, 17 ns, 20 ns or other?

d. For inputs: (I1:1->0, 9 ns; I2:1->0, 10 ns), should Q rise at:

14 ns, 15 ns, 19 ns, 20 ns or other?

e. For inputs: (I2:1->0, 9 ns; I1:1->0, 10 ns), should Q rise at:

14 ns, 15 ns, 19 ns, 20 ns or other?

f. For inputs: (I1:0->1, 10 ns; I2:0->1, 10 ns), should Q fall at:

13 ns, 15 ns, 17 ns, 20 ns or other?

In each case explain your answer using actual simulation results to help you.

13.24 (VHDL trace, 30 min.) Write a simple testbench and trace through the following VHDL behavioral simulation.

```
library IEEE;                                                  --1
use IEEE.std_logic_1164.all; use IEEE.NUMERIC_STD.all;         --2

entity comp_mux is                                             --3
  generic (TPD : TIME := 1 ns);                                --4
  port (A, B : in STD_LOGIC_VECTOR (2 downto 0);               --5
  Y : out STD_LOGIC_VECTOR (2 downto 0));                      --6
end;                                                           --7

architecture Behave of comp_mux is                             --8
begin                                                          --9
   Y <= A after TPD when (A <= B) else B after TPD;           --10
end;                                                          --11
```

13.25 (VHDL simulator, 30 min.) Explain the steps in using your VHDL simulator. Are there separate compile, analyze, elaborate, initialization, and simulate phases. Where and when do they occur. How do you know?

13.26 (Debugging VHDL, 60 min.) Correct the errors in the following code:

```
entity counter8 is port (
  rset, updn, clock : in bit; carry : out bit; count : buffer integer
range 0 to 255 );
end counter8;

architecture behave of counter8 is
begin process
  begin
  wait until clock'event and clock = '1';
  if (rset = '1') then count <= 0; carry <= '0';
  else case updn
    when '1' => count <= count + 1;
    if (count = 255) then carry <= '1'; else carry <= '0'; end if;
    when '0' => count <= count - 1;
    if (count = 0) then carry <= 1; else carry <= 0; end if;
    end case;
  end if;
end process;
end behave;
```

13.27 (***VITAL flip-flop) The following VITAL code models a D flip-flop:

```
LIBRARY ieee; USE ieee.Std_Logic_1164.all;                          --1
USE ieee.Vital_Timing.all; USE ieee.Vital_Primitives.all;           --2
ENTITY dff IS                                                       --3
  GENERIC (                                                         --4
  TimingChecksOn : BOOLEAN := TRUE;                                 --5
  XGenerationOn : BOOLEAN := TRUE;                                  --6
  InstancePath : STRING := "*";                                     --7
  tipd_Clock :  DelayType01 := (0 ns, 0 ns);                        --8
  tipd_Data :  DelayType01 := (0 ns, 0 ns);                         --9
  tsetup_Data_Clock : DelayType01 := (0 ns, 0 ns);                 --10
  thold_Data_Clock : DelayType01 := (0 ns, 0 ns);                  --11
  tpd_Clock_Q : DelayType01 := (0 ns, 0 ns);                       --12
  tpd_Clock_Qbar : DelayType01 := (0 ns, 0 ns));                   --13
  PORT (Clock, Data: Std_Logic; Q,Qbar:OUT Std_Logic);             --14
END dff;                                                           --15

ARCHITECTURE Gate OF dff IS                                         --1
  ATTRIBUTE Vital_Level1 of gate : ARCHITECTURE IS TRUE;            --2
  SIGNAL Clock_ipd : Std_Logic := 'X';                              --3
  SIGNAL Data_ipd  : Std_Logic := 'X';                              --4
BEGIN                                                               --5
Wire_Delay:BLOCK BEGIN --  INPUT PATH DELAYs                        --6
  VitalPropagateWireDelay                                           --7
    (Clock_ipd, Clock, VitalExtendToFillDelay(tipd_Clock));         --8
  VitalPropagateWireDelay                                           --9
    (Data_ipd, Data, VitalExtendToFillDelay(tipd_Data));           --10
END BLOCK;                                                         --11
VitalBehavior : PROCESS (Clock_ipd, Data_ipd)                      --12
  CONSTANT Dff_tab:VitalStateTableType:= (                         --13
--Vio   CLOCK DATA  IQ    Q    QBAR                                --14
( 'X',  '-',  '-',  '-',  'X',  'X' ), -- Timing Violation         --15
( '-',  '\',  '0',  '-',  '0',  '1' ), -- Active Clock Edge        --16
( '-',  '\',  '1',  '-',  '1',  '0' ),                             --17
( '-',  '\',  'X',  '-',  'X',  'X' ),                             --18
( '-',  '-',  '0',  '0',  '0',  '1' ), -- X Reduction              --19
( '-',  '-',  '1',  '1',  '1',  '0' ),                             --20
( '-',  'D',  '-',  '-',  'X',  'X' ), -- X Generation             --21
( '-',  'B',  '-',  '-',  'S',  'S' ), -- Non-Active Clock Edge    --22
( '-',  'X',  '-',  '-',  'S',  'S' ));                            --23
-- Anything else generates X on Q and QBAR                         --24
-- Timing Check Results                                            --25
  VARIABLE Tviol_Data_Clock : X01 := '0';                          --26
  VARIABLE Tmkr_Data_Clock  : TimeMarkerType;                      --27
-- Functionality Results                                           --28
  VARIABLE Violation:X01:='0';                                     --29
  VARIABLE PrevData:Std_Logic_Vector(1 to 3):=(OTHERS=>'X');       --30
  VARIABLE Results:Std_Logic_Vector(1 to 2):=(OTHERS =>'X');
  ALIAS Q_zd:Std_Logic IS Results(1);
  ALIAS Qbar_zd:Std_Logic IS Results(2);
-- Output Glitch Detection Variables
  VARIABLE Q_GlitchData : GlitchDataType;
  VARIABLE Qbar_GlitchData : GlitchDataType;
BEGIN --  Timing Check Section
  IF (TimingChecksOn) THEN
  VitalTimingCheck (
  Data_ipd, "Data", Clock_ipd, "Clock",
  t_setup_hi => tsetup_Data_Clock(tr01),
  t_setup_lo => tsetup_Data_Clock(tr10),
  t_hold_hi => thold_Data_Clock(tr01),
  t_hold_lo => thold_Data_Clock(tr10),
  CheckEnabled  => TRUE,
  RefTransition => (Clock_ipd = '0'),
  HeaderMsg => InstancePath & "/DFF",
  TimeMarker => Tmkr_Data_Clock,
  Violation => Tviol_Data_Clock);
  END IF;
--  Functionality Section
  Violation := Tviol_Data_Clock ;
  VitalStateTable(StateTable => Dff_tab,
  DataIn => (Violation, Clock_ipd, Data_ipd),
  NumStates  => 1,
  Result => Results,
  PreviousDataIn => PrevData);
--  Path Delay Section
  VitalPropagatePathDelay (Q, "Q", Q_zd,
  Paths => (0 => (Clock_ipd'LAST_EVENT,
  VitalExtendToFillDelay(tpd_Clock_Q), TRUE),
    1 => (Clock_ipd'LAST_EVENT,
  VitalExtendToFillDelay(tpd_Clock_Q), TRUE)),
    GlitchData => Q_GlitchData,
    GlitchMode => MessagePlusX,
    GlitchKind => OnEvent );
  VitalPropagatePathDelay ( Qbar, "Qbar", Qbar_zd,
    Paths => (0 => (Clock_ipd'LAST_EVENT,
  VitalExtendToFillDelay(tpd_Clock_Qbar), TRUE),
    1 => (Clock_ipd'LAST_EVENT,
  VitalExtendToFillDelay(tpd_Clock_Qbar), TRUE)),
    GlitchData => Qbar_GlitchData,
    GlitchMode => MessagePlusX,
    GlitchKind => OnEvent );
  END PROCESS;
END Gate;
```

a. (120 min.) Build a testbench for this model.

b. (30 min.) Simulate and check the model using your testbench.

c. (60 min.) Explain the function of each line.

d. (60 min.) Explain the glitch detection.

e. (120 min.) Explain the unknown propagation behavior.

13.28 (VCD, 30 min.) Verilog can create a **value change dump (VCD)** file:

```
module waves; reg clock; integer count;
initial begin clock = 0; count = 0; $dumpvars; #340 $finish; end
always #10 clock = ~ clock;
always begin @ (negedge clock); if (count == 7) count = 0;
     else count = count + 1; end
endmodule
```

A VCD file contains header information, variable definitions, and the value changes for variables [Verilog LRM 15]. Try and explain the format of the file that results.

13.29 (*Formal verification, 60 min.) (Based on an example by Browne, Clarke, Dill, and Mishra.) A designer needs to fold an 8-bit ripple-carry adder into a small space on an ASIC and check the circuit extracted from the layout. With two 8-bit inputs, A and B, and a 1-bit carry Cin, exhaustively testing all possible inputs requires 2^{17} or over 128,000 input vectors. Instead the designer selects a subset of tests. The three tests in Table 13.17 check that all the bits of the output can be '0' or '1'. The two tests in Table 13.18 make sure that the carry propagates through the adder, and that the adder can handle the largest numbers. The designer then repeats all of these five tests with the carry-in Cin set to '1' instead of '0'. Next the designer performs a series of 24 tests using the three patterns shown in Table 13.19, with all eight possible combinations of '0' and '1' for x, y, and z. These patterns test each full adder in isolation for all possible sum inputs (A and B) and carry input (Cin). These tests appear comprehensive and reduce the number of vectors required from over 128,000 to 34. Confident, the designer releases the chip for fabrication. Unfortunately, the chip does not work. Which connections between adders did the designer's tests fail to check?

TABLE 13.17 Test to check for output toggling (Problem 13.29).

A	00000000	00000000	01010101
B	00000000	11111111	10101010
Cin	0	0	0
Sum	000000000	011111111	011111111

TABLE 13.18 Test to check for carry propagation (Problem 13.29).

A	00000001	11111111
B	11111111	11111111
Cin	0	0
Sum	100000000	111111110

TABLE 13.19 Test with all possible combinations of `'0'` and `'1'` for `x`, `y`, and `z` (Problem 13.29).

`A`	`xz0xz0xz`	`z0xz0xz0`	`0xz0xz0x`
`B`	`yz0yz0yz`	`z0yz0yz0`	`0yz0yz0y`
`C`	`0`	`z`	`z`
`Sum`	`cs0cs0cs0`	`z0cs0cs0z`	`0cs0cs0cs`

13.30 (*BSIM1 parameters, 120 min.) SPICE models are tangled webs. For example, there are two formats for the 69 UCB BSIM1 transistor parameters: (1) using parameter names (Table 13.15); and (2) without parameter names (Table 13.20). MOSIS uses format (2). The first 58 parameters starting with `VFB` (19 rows of 3 parameters plus one, `XPART`) are the same order in format (1) and (2). Format (2) uses two dummy parameters (zero) following `XPART`. The final nine parameters (`NO` to `WND`) are the same order in both formats. There are 10 additional parameters that follow the n- and p-channel transistor parameters that model the n- and p-diffusion, respectively (in the same order in both formats). To complicate things further: (i) HSPICE, SPICE, and PSpice use different names for some parameters (for example PSpice uses `VFB`, HSPICE uses `VFB0`); (ii) HSPICE accepts names for the dummy parameters (`DUM1` and `DUM2`), but PSpice does not; (iii) HSPICE accepts the final parameter `DS` (though it is ignored), but PSpice does not (`DS` models mask bias and can be neglected without too much fear).

Convert the models shown in Table 13.20 to format (2) for your chosen simulator (PSpice `LEVEL=4`, HSPICE `LEVEL=13`). Compare the resulting I_{DS}–V_{DS} characteristics with the `LEVEL=3` parameters (Table 13.14) and the (PSpice) `LEVEL=4` parameters (Table 13.15) for the G5 process. MOSIS has stored the results of its process runs in the format shown in Table 13.20.

13.31 (**Nonreciprocal capacitance, 120 min.)

a. Starting from the equation for transient current flowing into the gate,

$$i_G = \frac{\partial Q_G}{\partial V_D}\frac{dV_D}{dt} + \frac{\partial Q_G}{\partial V_G}\frac{dV_G}{dt} + \frac{\partial Q_G}{\partial V_S}\frac{dV_S}{dt} + \frac{\partial Q_G}{\partial V_B}\frac{dV_B}{dt}, \tag{13.35}$$

(where $\delta Q_i/\delta V_j$ are elements of matrix **M**), show

$$i_G = -C_{GD}\frac{dV_D}{dt} + C_{GG}\frac{dV_G}{dt} - C_{GS}\frac{dV_S}{dt} - C_{GB}\frac{dV_B}{dt} \tag{13.36}$$

and thus that the rows of **M** sum to zero by showing that

$$C_{GG} = C_{GD} + C_{GS} + C_{GB} \tag{13.37}$$

TABLE 13.20 MOSIS SPICE parameters (Problem 13.30).

*NMOS PARAMETERS	*PMOS PARAMETERS
-7.05628E-01,-3.86432E-02, 4.98790E-02	-2.02610E-01, 3.59493E-02,-1.10651E-01
8.41845E-01, 0.00000E+00, 0.00000E+00	8.25364E-01, 0.00000E+00, 0.00000E+00
7.76570E-01,-7.65089E-04,-4.83494E-02	3.54162E-01,-6.88193E-02, 1.52476E-01
2.66993E-02, 4.57480E-02,-2.58917E-02	-4.51065E-02, 9.41324E-03, 3.52243E-02
-1.94480E-03, 1.74351E-02,-5.08914E-03	-1.07507E-02, 1.96344E-02,-3.51067E-04
5.75297E+02,1.70587E-001,4.75746E-001	1.37992E+02,1.92169E-001,4.68470E-001
3.30513E-01, 9.75110E-02,-8.58678E-02	1.89331E-01, 6.30898E-02,-6.38388E-02
3.26384E-02, 2.94349E-02,-1.38002E-02	1.31710E-02, 1.44096E-02, 6.92372E-04
9.73293E+00,-5.62944E+00, 6.55955E+00	6.57709E+00,-1.56096E+00, 1.13564E+00
4.37180E-04,-3.07010E-03, 8.94355E-04	4.68478E-05,-1.09352E-03,-1.53111E-04
-5.05012E-05,-1.68530E-03,-1.42701E-03	7.76679E-04,-1.97213E-04,-1.12034E-03
-1.11542E-02,-9.58423E-04, 4.61645E-03	8.71439E-03,-1.92306E-03, 1.86243E-03
-1.04401E-03, 1.29001E-03,-7.10095E-04	5.98941E-04, 4.54922E-04, 3.11794E-04
6.92716E+02,-5.21760E+01, 7.00912E+00	1.49460E+02, 1.36152E+01, 3.55246E+00
-6.41307E-02, 1.37809E+00, 4.15455E+00	6.37235E+00,-6.63305E-01, 2.25929E+00
8.86387E+00, 2.06021E+00,-6.19817E+00	-1.21135E-02, 1.92973E+00, 1.00182E+00
9.02467E-03, 2.06380E-04,-5.20218E-03	-1.16599E-03,-5.08278E-04, 9.56791E-04
9.60000E-003, 2.70000E+01, 5.00000E+00	9.60000E-003, 2.70000E+01, 5.00000E+00
3.60204E-010,3.60204E-010,4.37925E-010	4.18427E-010,4.18427E-010,4.33943E-010
1.00000E+000,0.00000E+000,0.00000E+000	1.00000E+000,0.00000E+000,0.00000E+000
1.00000E+000,0.00000E+000,0.00000E+000	1.00000E+000,0.00000E+000,0.00000E+000
0.00000E+000,0.00000E+000,0.00000E+000	0.00000E+000,0.00000E+000,0.00000E+000
0.00000E+000,0.00000E+000,0.00000E+000	0.00000E+000,0.00000E+000,0.00000E+000
*N+ diffusion::	*P+ diffusion::
2.1, 3.5e-04, 2.9e-10, le-08, 0.8	2, 9.4529e-04, 2.4583e-10, le-08, 0.85
0.8, 0.44, 0.26, 0, 0	0.85, 0.439735, 0.237251, 0, 0

Source: MOSIS, process = HP-NID, technology = scn05h, run = n5bo, wafer = 42, date = 1-Feb-1996.

and three other similar equations for C_{DD}, C_{SS}, and C_{BB}.

b. Show

$$i_G = -C_{GD}\frac{dV_{DB}}{dt} + C_{GG}\frac{dV_{GB}}{dt} - C_{GS}\frac{dV_{SB}}{dt} \tag{13.38}$$

and derive similar equations for the transient currents i_S, i_D, and i_B.

c. Using the fact that $i_G + i_S + i_D + i_B = 0$, show

$$C_{GG} = C_{DG} + C_{SG} + C_{BG} \tag{13.39}$$

Capilano Computing. 1997. *LogicWorks Verilog Modeler: Interactive Circuit Simulation Software for Windows and Macintosh.* Menlo Park, CA: Capilano Computing, 102 p. ISBN 0201895854. TK7888.4.L64.

Carey, G. F., et al. 1996. *Circuit, Device, and Process Simulation: Mathematical and Numerical Aspects.* New York: Wiley, 425 p. ISBN 0471960195. TK7867.C4973. 31 pages of references.

Cheng, K.-T., and V. D. Agrawal. 1989. *Unified Methods for VLSI Simulation and Test Generation.* Norwell, MA: Kluwer, 148 p. ISBN 0-7923-9025-3. TK7874.C525. 377 references. The first three chapters give a good introduction to fault simulation and test-vector generation.

Ciccarelli, F. A. 1995. *Circuit Modeling: Exercises and Software.* 3rd ed. Englewood Cliffs: Prentice-Hall, 190 p. ISBN 0023224738. TK454.C59. Includes BreadBoard, an IBM-PC compatible circuit analysis computer program.

Conant, R. 1993. *Engineering Circuit Analysis with PSpice and Probe: Macintosh Version.* New York: McGraw-Hill, 176 p. ISBN 0079116795. TK454.C674.

Divekar, D. 1988. *FET Modeling for Circuit Simulation.* Boston: Kluwer, 183 p. ISBN 0898382645. TK7871.95.D58. 12 pages of references.

Fenical, L. H. 1992. *PSpice: A Tutorial.* Englewood Cliffs, NJ: Prentice-Hall, 344 p. ISBN 0136811493. TK454.F46.

Fjeldly, T. A., T. Ytterdal, and M. Shur. 1997. *Introduction to Device Modeling and Circuit Simulation.* New York: Wiley, ISBN 0471157783. TK7871.85.F593.

Hëorbst, E. (Ed.). 1986. *Logic Design and Simulation.* New York: Elsevier Science. ISBN 0-444-87892-0. TK7868.L6L624.

Hill, D. D., and D. R. Coelho. 1987. *Multi-Level Simulation for VLSI Design.* Boston: Kluwer, 206 p. ISBN 0-89838-184-3. TK7874.H525.

IEEE 1076.4-1995. *IEEE Standard VITAL Application-Specific Integrated Circuit (ASIC) Modeling Specification.* 96p. ISBN 1-55937-691-0. IEEE Ref. SH94382-NYF. The Institute of Electrical and Electronics Engineers. Available from The IEEE, 345 East 47th Street, New York, NY 10017 USA. [cited on p. 664 of this chapter]

Kielkowski, R. M. 1994. *Inside SPICE: Overcoming the Obstacles of Circuit Simulation.* New York: McGraw-Hill, 188 p. ISBN 0-07-911525-X. TK454.K48.

Lamey, R. 1995. *The Illustrated Guide to PSpice.* Albany, NY: Delmar, 219 p. ISBN 0827365241. TK454.L35.

Massobrio, G., and P. Antognetti. 1993. *Semiconductor Device Modeling with SPICE.* New York: McGraw-Hill, 479 p. ISBN 0-07-002469-3. TK7871.85.S4454. Contains a more detailed analysis of the SPICE models than other introductory texts.

McCalla, W. J. 1988. *Fundamentals of Computer-Aided Circuit Simulation.* Boston: Kluwer, 175 p. ISBN 0-89838-248-3. TK7874.M355.

Meta Software. 1996. *HSPICE User's Manual.* No catalog information. Available from Customer Service, 1300 White Oaks Road, Campbell, CA 95008, `cs@metasw.com`. This is a three-volume paperback set that is available separately from the HSPICE program. Volume I, Simulation and Analysis, explains the operation of the HSPICE program. Volume II, Elements and Device Models, contains a comprehensive description of all device models used in HSPICE. Volume III, Analysis and Methods, details input control, types of analysis, output format, optimization, filter and system design, statistical and worst-case analysis, characterization, behavioral applications, and signal integrity (packaging). [cited on p. 692, p. 693, p. 694, p. 696 of this chapter]

Miczo, A. 1994. *Digital Logic Testing and Simulation.* New York: Harper & Row, 414 p. ISBN 0-06-044444-4. TK7868.D5M49.

TEST

14

ASICs are tested at two stages during manufacture using **production tests**. First, the silicon die are tested after fabrication is complete at **wafer test** or **wafer sort**. Each wafer is tested, one die at a time, using an array of probes on a **probe card** that descend onto the bonding pads of a single die. The **production tester** applies signals generated by a **test program** and measures the ASIC **test response**. A test program often generates hundreds of thousands of different **test vectors** applied at a frequency of several megahertz over several hundred milliseconds. Chips that fail are automatically marked with an ink spot. Production testers are large machines that take up their own room and are very expensive (typically well over $1 million). Either the customer, or the ASIC manufacturer, or both, develops the test program.

A diamond saw separates the die, and the good die are bonded to a lead carrier and packaged. A second, **final test** is carried out on the packaged ASIC (usually with the same test vectors used at wafer sort) before the ASIC is shipped to the customer. The customer may apply a **goods-inward test** to incoming ASICs if the customer has the resources and the product volume is large enough. Normally, though, parts are directly assembled onto a bare **printed-circuit board** (**PCB** or **board**) and then the board is tested. If the board test shows that an ASIC is bad at this point, it is difficult to replace a surface-mounted component soldered on the board, for example. If there are several board failures due to a particular ASIC, the board manufacturer typically ships the defective chips back to the ASIC vendor. ASIC vendors have sophisticated **failure analysis** departments that take packaged ASICs apart and can often determine the failure mechanism. If the ASIC production tests are adequate, failures are often due to the soldering process, electrostatic damage during handling, or other problems that can occur between the part being shipped and board test. If the problem is traced to defective ASIC fabrication, this indicates that the test program may be inadequate. As we shall see, failure and diagnosis at the board level is very expensive. Finally, ASICs may be tested and replaced (usually by swapping boards) either by a customer who buys the final product or by servicing—this is **field repair**. Such system-level diagnosis and repair is even more expensive.

Programmable ASICs (including FPGAs) are a special case. Each programmable ASIC is tested to the point that the manufacturer can guarantee with a high degree of confidence that if your design works, and if you program the FPGA correctly, then your ASIC will work. Production testing is easier for some programmable ASIC architectures than others. In a reprogrammable technology the manufacturer can test the programming features. This cannot be done for a one-time programmable antifuse technology, for example. A programmable ASIC is still tested in a similar fashion to any other ASIC and you are still paying for test development and design. Programmable ASICs also have similar test, defect, and manufacturing problems to other members of the ASIC family. Finally, once a programmable ASIC is soldered to a board and part of a system, it looks just like any other ASIC. As you will see in the next section, considering board-level and system-level testing is a very important part of ASIC design.

14.1 The Importance of Test

One measure of **product quality** is the **defect level**. If the ABC Company sells 100,000 copies of a product and 10 of these are defective, then we say the defect level is 0.1 percent or 100 ppm. The **average quality level** (**AQL**) is equal to one minus the defect level (ABC's AQL is thus 99.9 percent).

Suppose the semiconductor division of ABC makes an ASIC, the bASIC, for the PC division. The PC division buys 100,000 bASICs, tested by the semiconductor division, at $10 each. The PC division includes one surface-mounted bASIC on each PC motherboard it assembles for the aPC computer division. The aPC division tests the finished motherboards. Rejected boards due to defective bASICs incur an average $200 board repair cost. The board repair cost as a function of the ASIC defect level is shown in Table 14.1. A defect level of 5 percent in bASICs costs $1 million dollars in board repair costs (the same as the total ASIC part cost). Things are even worse at the system level, however.

TABLE 14.1 Defect levels in printed-circuit boards (PCB).[1]

ASIC defect level	Defective ASICs	Total PCB repair cost
5%	5000	$1million
1%	1000	$200,000
0.1%	100	$20,000
0.01%	10	$2,000

[1] Assumptions: The number of parts shipped is 100,000; part price is $10; total part cost is $1 million; the cost of a fault in an assembled PCB is $200.

Suppose the ABC Company sells its aPC computers for $5,000, with a profit of $500 on each. Unfortunately the aPC division also has a defect level. Suppose that 10 percent of the motherboards that contain defective bASICs that passed the chip test also manage to pass the board tests (10 percent may seem high, but chips that have hard-to-test faults at the chip level may be very hard to find at the board level—catching 90 percent of these rogue chips would be considered good). The system-level repair cost as a function of the bASIC defect level is shown in Table 14.2. In this example a 5 percent defect level in a $10 bASIC part now results in a $5 million cost at the system level. From Table 14.2 we can see it would be worth spending $4 million (i.e., $5 million – $1 million) to reduce the bASIC defect density from 5 percent to 1 percent.

TABLE 14.2 Defect levels in systems.[1]

ASIC defect level	Defective ASICs	Defective boards	Total repair cost at system level
5%	5000	500	$5 million
1%	1000	100	$1 million
0.1%	100	10	$100,000
0.01%	10	1	$10,000

[1] Assumptions: The number of systems shipped is 100,000; system cost is $5,000; total cost of systems shipped is $500 million; the cost of repairing or replacing a system due to failure is $10,000; profit on 100,000 systems is $50 million.

14.2 Boundary-Scan Test

It is possible to test ICs in dual-in-line packages (DIPs) with 0.1 inch (2.5 mm) lead spacing on low-density boards using a **bed-of-nails tester** with probes that contact test points underneath the board. Mechanical testing becomes difficult with board trace widths and separations below 0.1 mm or 100 µm, package-pin separations of 0.3 mm or less, packages with 200 or more pins, surface-mount packages on both sides of the board, and multilayer boards [Scheiber, 1995].

In 1985 a group of European manufacturers formed the **Joint European Test Action Group (JETAG)** to study board testing. With the addition of North American companies, JETAG became the **Joint Test Action Group (JTAG)** in 1986. The JTAG 2.0 test standard formed the basis of the **IEEE Standard 1149.1 Test Port and Boundary-Scan Architecture** [IEEE 1149.1b, 1994], approved in February 1990 and also approved as a standard by the American National Standards Institute (ANSI) in August 1990 [Bleeker, v. d. Eijnden, and de Jong, 1993; Maunder and Tulloss, 1990; Parker, 1992]. The IEEE standard is still often referred to as JTAG, although there are important differences between the last JTAG specification (version 2.0) and the IEEE 1149.1 standard.

Boundary-scan test (BST) is a method for testing boards using a four-wire interface (five wires with an optional master reset signal). A good analogy would be the RS-232 interface for PCs. The BST standard interface was designed to test boards, but it is also useful to test ASICs. The BST interface provides a standard means of communicating with test circuits on-board an ASIC. We do need to include extra circuits on an ASIC in order to use BST. This is an example of increasing the cost and complexity (as well as potentially reducing the performance) of an ASIC to reduce the cost of testing the ASIC and the system.

Figure 14.1(a) illustrates failures that may occur on a PCB due to shorts or opens in the copper traces on the board. Less frequently, failures in the ASIC package may also arise from shorts and opens in the wire bonds between the die and the package frame (Figure 14.1b). Failures in an ASIC package that occur during ASIC fabrication are caught by the ASIC production test, but stress during automated handling and board assembly may cause package failures. Figure 14.1(c) shows how a group of ASICs are linked together in boundary-scan testing. To detect the failures shown in Figure 14.1(a) or (b) manufacturers use boundary scan to test every connection between ASICs on a board. During boundary scan, test data is loaded into each ASIC and then driven onto the board traces. Each ASIC monitors its inputs, captures the data received, and then shifts the captured data out. Any defects in the board or ASIC connections will show up as a discrepancy between expected and actual measured continuity data.

In order to include BST on an ASIC, we add a special logic cell to each ASIC I/O pad. These cells are joined together to form a chain and create a boundary-scan shift register that extends around each ASIC. The input to a boundary-scan shift register is the **test-data input (TDI)**. The output of a boundary-scan shift register is

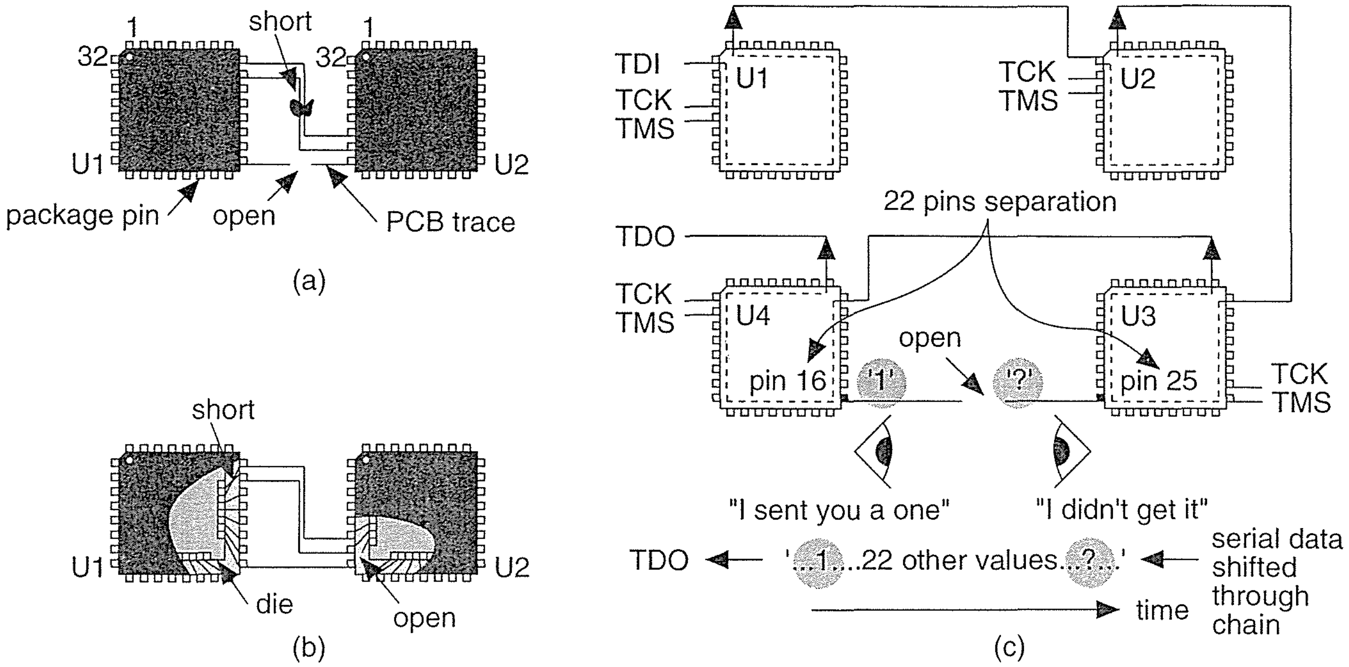


FIGURE 14.1 IEEE 1149.1 boundary scan. (a) Boundary scan is intended to check for shorts or opens between ICs mounted on a board. (b) Shorts and opens may also occur inside the IC package. (c) The boundary-scan architecture is a long chain of shift registers allowing data to be sent over all the connections between the ICs on a board.

the **test-data output** (**TDO**). These boundary-scan shift registers are then linked in a serial fashion with the boundary-scan shift registers on other ASICs to form one long boundary-scan shift register. The boundary-scan shift register in each ASIC is one of several **test-data registers** (**TDR**) that may be included in each ASIC. All the TDRs in an ASIC are connected directly between the TDI and TDO ports. A special register that decodes instructions provides a way to select a particular TDR and control operation of the boundary-scan test process.

Controlling all of the operations involved in selecting registers, loading data, performing a test, and shifting out results are the **test clock** (**TCK**) and **test-mode select** (**TMS**). The boundary-scan standard specifies a four-wire test interface using the four signals: TDI, TDO, TCK, and TMS. These four dedicated signals, the **test-access port** (**TAP**), are connected to the TAP controller inside each ASIC. The TAP controller is a state machine clocked on the rising edge of TCK, and with state transitions controlled by the TMS signal. The **test-reset input signal** (**TRST***, **nTRST**, or **TRST**—always an active-low signal) is an optional (fifth) dedicated interface pin to reset the TAP controller.

Normally the boundary-scan shift-register cells at each ASIC I/O pad are transparent, allowing signals to pass between the I/O pad and the core logic. When an ASIC is put into boundary-scan test mode, we first tell the TAP controller which

TDR to select. The TAP controller then tells each boundary-scan shift register in the appropriate TDR either to capture input data, to shift data to the neighboring cell, or to output data.

There are many acronyms in the IEEE 1149.1 standard (referred to as "**dot one**"); Table 14.3 provides a list of the most common terms.

TABLE 14.3 Boundary-scan terminology.

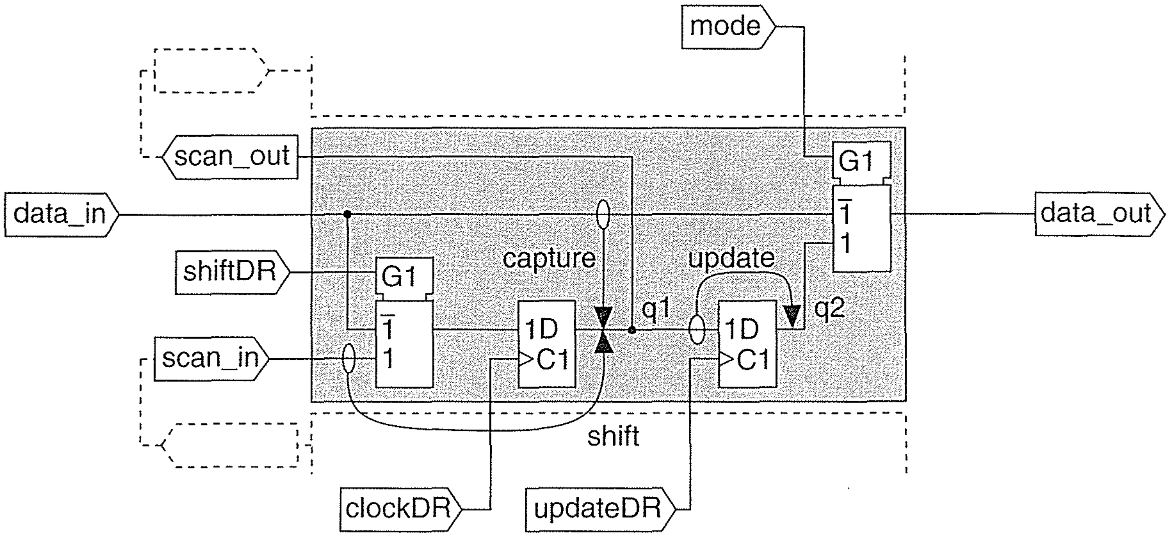
Acronym	Meaning	Explanation
BR	Bypass register	A TDR, directly connects TDI and TDO, bypassing BSR
BSC	Boundary-scan cell	Each I/O pad has a BSC to monitor signals
BSR	Boundary-scan register	A TDR, a shift register formed from a chain of BSCs
BST	Boundary-scan test	Not to be confused with BIST (built-in self-test)
IDCODE	Device-identification register	Optional TDR, contains manufacturer and part number
IR	Instruction register	Holds a BST instruction, provides control signals
JTAG	Joint Test Action Group	The organization that developed boundary scan
TAP	Test-access port	Four- (or five-)wire test interface to an ASIC
TCK	Test clock	A TAP wire, the clock that controls BST operation
TDI	Test-data input	A TAP wire, the input to the IR and TDRs
TDO	Test-data output	A TAP wire, the output from the IR and TDRs
TDR	Test-data register	Group of BST registers: IDCODE, BR, BSR
TMS	Test-mode select	A TAP wire, together with TCK controls the BST state
TRST* or nTRST	Test-reset input signal	Optional TAP wire, resets the TAP controller (active-low)

14.2.1 BST Cells

Figure 14.2 shows a **data-register cell** (**DR cell**) that may be used to implement any of the TDRs. The most common DR cell is a **boundary-scan cell** (**BS cell**, or **BSC**), or **boundary-register cell** (this last name is not abbreviated to BR cell, since this term is reserved for another type of cell) [IEEE 1149.1b-1994, p. 10-18, Fig. 10-16].

A BSC contains two sequential elements. The **capture flip-flop** or **capture register** is part of a shift register formed by series connection of BSCs. The **update flip-flop**, or **update latch**, is normally drawn as an edge-triggered D flip-flop, though it may be a transparent latch. The inputs to a BSC are: **scan in** (**serial in** or **SI**); **data in** (**parallel in** or **PI**); and a control signal, **mode** (also called **test/normal**). The BSC outputs are: **scan out** (**serial out** or **SO**); **data out** (**parallel**

out or **PO**). The BSC in Figure 14.2 is **reversible** and can be used for both chip inputs and outputs. Thus `data_in` may be connected to a pad and `data_out` to the core logic cr vice versa.



```
entity DR_cell is port (mode, data_in, shiftDR, scan_in, clockDR, updateDR: BIT;      --1
  data_out, scan_out: out BIT ); end DR_cell;                                          --2
architecture behave of DR_cell is signal q1, q2 : BIT; begin                           --3
CAP : process(clockDR) begin if clockDR = '1' then                                     --4
  if shiftDR = '0' then q1 <= data_in; else q1 <= scan_in; end if; end if;             --5
end process;                                                                           --6
UPD : process(updateDR) begin if updateDR = '1' then q2 <= q1; end if; end process;    --7
data_out <= data_in when mode = '0' else q2; scan_out <= q1;                           --8
end behave;                                                                            --9
```

FIGURE 14.2 A DR (data register) cell. The most common use of this cell is as a boundary-scan cell (BSC).

The IEEE 1149.1 standard shows the sequential logic in a BSC controlled by the gated clocks: `clockDR` (whose positive edge occurs at the positive edge of TCK) and `updateDR` (whose positive edge occurs at the negative edge of TCK). The IEEE 1149.1 schematics illustrate the standard but do not define how circuits should be implemented. The function of the circuit in Figure 14.2 (and its model) follows the IEEE 1149.1 standard and many other published schematics, but this is not necessarily the best, or even a safe, implementation. For example, as drawn here, signals `clockDR` and `updateDR` are gated clocks—normally to be avoided if possible. The update sequential element is shown as an edge-triggered D flip-flop but may be implemented using a latch.

Figure 14.3 [IEEE 1149.1b-1994, Chapter 9] shows a **bypass-register cell (BR cell)**. The BR inputs and outputs, scan in (serial in, SI) and scan out (serial out, SO), have the same names as the DR cell ports, but DR cells and BR cells are not directly connected.

```
entity BR_cell is port (
 clockDR,shiftDR,scan_in : BIT; scan_out : out BIT );
end BR_cell;
architecture behave of BR_cell is
signal t1 : BIT; begin t1 <= shiftDR and scan_in;
process (clockDR) begin
  if (clockDR = '1') then scan_out <= t1; end if;
end process;
end behave;
```

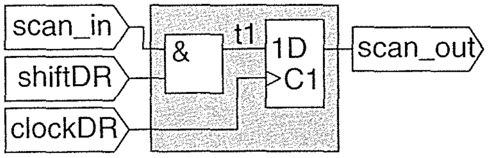


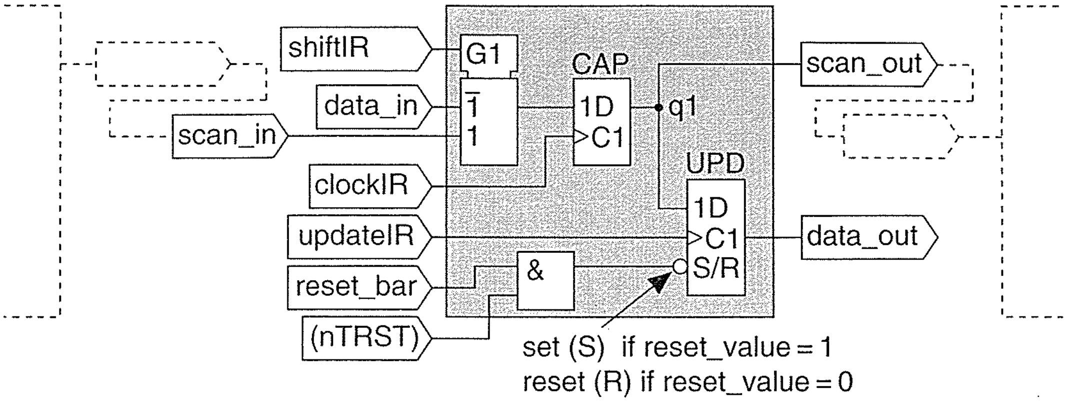
FIGURE 14.3 A BR (bypass register) cell.

Figure 14.4 shows an **instruction-register cell (IR cell)** [IEEE 1149.1b-1994, Chapter 6]. The IR cell inputs are: scan_in, data_in; as well as clock, shift, and update signals (with names and functions similar to those of the corresponding signals in the BSC). The reset signals are nTRST and reset_bar (active-low signals often use an asterisk, reset* for example, but this is not a legal VHDL name). The two LSBs of data_in must permanently be set to '01' (this helps in checking the integrity of the scan chain during testing). The remaining data_in bits are status bits under the control of the designer. The update sequential element (sometimes called the **shadow register**) in each IR cell may be set or reset (depending on reset_value). The IR cell outputs are: data_out (the instruction bit passed to the instruction decoder) and scan_out (the data passed to the next IR cell in the IR).

14.2.2 BST Registers

Figure 14.5 shows a **boundary-scan register (BSR)**, which consists of a series connection, or chain, of BSCs. The BSR surrounds the ASIC core logic and is connected to the I/O pad cells. The BSR monitors (and optionally controls) the inputs and outputs of an ASIC. The direction of information flow is shown by an arrow on each of the BSCs in Figure 14.5. The control signal, mode, is decoded from the IR. Signal mode is drawn as common to all cells for the BSR in Figure 14.5, but that is not always the case.

Figure 14.6 shows an **instruction register (IR)**, which consists of at least two IR cells connected in series. The IEEE 1149.1 standard specifies that the IR cell is reset to '00...01' (the optional IDCODE instruction). If there is no IDCODE TDR, then the IDCODE instruction defaults to the BYPASS instruction.



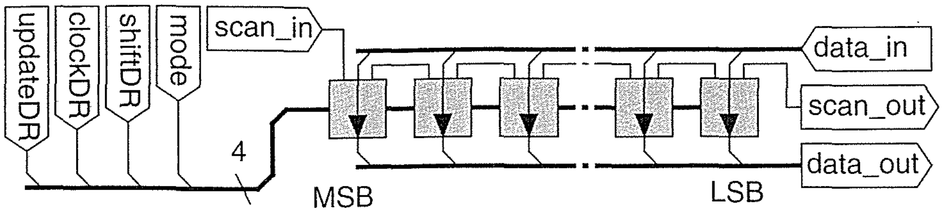
```
entity IR_cell is port(
  shiftIR, data_in, scan_in, clockIR, updateIR, reset_bar, nTRST, reset_value : BIT;
  data_out, scan_out : out BIT); end IR_cell;
architecture behave of IR_cell is signal q1, SR : BIT; begin
scan_out <= q1; SR <= reset_bar and nTRST;
CAP:process (clockIR) begin
  if (clockIR = '1') then
    if (shiftIR = '0') then q1 <= data_in; else q1 <= scan_in; end if;
  end if;
end process;
UPD:process (updateIR, SR) begin
  if (SR = '0') then data_out <= reset_value;
  elsif ((updateIR = '1') and updateIR'EVENT) then data_out <= q1;
  end if;
end process;
end behave;
```

FIGURE 14.4 An IR (instruction register) cell.

14.2.3 Instruction Decoder

Table 14.4 on page 722 shows an **instruction decoder**. This model is capable of decoding the following minimum set of boundary-scan instructions:

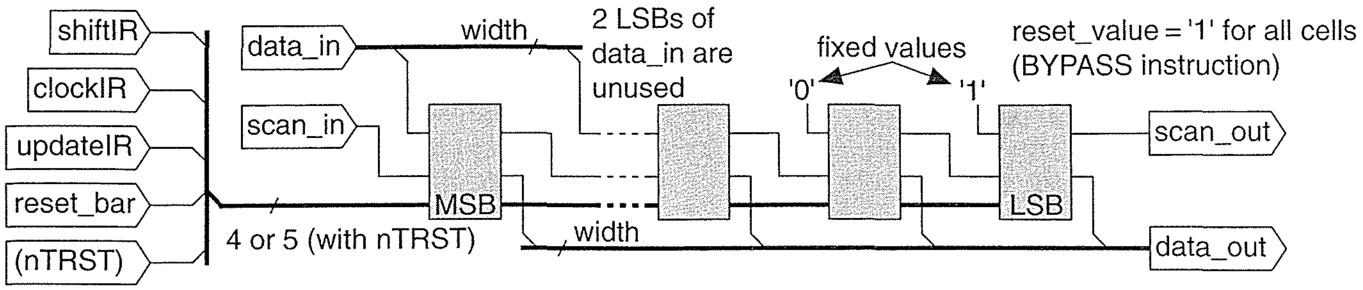
1. EXTEST, external test. Drives a known value onto each output pin to test connections between ASICs.
2. SAMPLE/PRELOAD (often abbreviated to SAMPLE). Performs two functions: first sampling the present input value from input pad during capture; and then preloading the BSC update register output during update (in preparation for an EXTEST instruction, for example).
3. IDCODE. An optional instruction that allows the **device-identification register** (IDCODE) to be shifted out. The IDCODE TDR is an optional register that



```
entity BSR is
generic (width : INTEGER := 3);
port (shiftDR, clockDR, updateDR, mode, scan_in : BIT;
  scan_out : out BIT;
  data_in : BIT_VECTOR(width-1 downto 0);
  data_out : out BIT_VECTOR(width-1 downto 0));
end BSR;

architecture structure of BSR is
component DR_cell port (
  mode, data_in, shiftDR, scan_in, clockDR, updateDR : BIT;
  data_out, scan_out : out BIT);
end component;
for all : DR_cell use entity WORK.DR_cell(behave);
signal int_scan : BIT_VECTOR (data_in'RANGE);
begin
BSR : for i in data_in'LOW to data_in'HIGH generate
  RIGHT : if (i = 0) generate
  BSR_LSB : DR_cell port map (mode, data_in(i), shiftDR,
  int_scan(i), clockDR, updateDR, data_out(i), scan_out);
  end generate;
  MIDDLE : if ((i > 0) and (i < data_in'HIGH)) generate
  BSR_i : DR_cell port map (mode, data_in(i), shiftDR,
  int_scan(i), clockDR, updateDR, data_out(i), int_scan(i-1));
  end generate;
  LFET : if (i = data_in'HIGH) generate
  BSR_MSB : DR_cell port map (mode, data_in(i), shiftDR,
  scan_in, clockDR, updateDR, data_out(i), int_scan(i-1));
  end generate;
end generate;
end structure;
```

FIGURE 14.5 A BSR (boundary-scan register). An example of the component data-register (DR) cells (used as boundary-scan cells) is shown in Figure 14.2.

allows the tester to query the ASIC for the manufacturer's name, part number, and other data that is shifted out on TDO. IDCODE defaults to the BYPASS instruction if there is no IDCODE TDR.



```
entity IR is generic (width : INTEGER := 4); port (                           --1
  shiftIR, clockIR, updateIR, reset_bar, nTRST, scan_in : BIT; scan_out : out BIT;  --2
  data_in : BIT_VECTOR (width-1 downto 0) ;                                  --3
  data_out : out BIT_VECTOR (width-1 downto 0) );                            --4
end IR;                                                                       --5

architecture structure of IR is                                               --6
component IR_cell port (shiftIR, data_in, scan_in, clockIR,                   --7
  updateIR, reset_bar, nTRST, reset_value : BIT ; data_out, scan_out : out BIT );  --8
end component;                                                                --9
for all : IR_cell use entity WORK.IR_cell(behave);                            --10
signal int_scan : BIT_VECTOR (data_in'RANGE);                                 --11
signal Vdd : BIT := '1'; signal GND : BIT := '0';                             --12
begin                                                                         --13
IRGEN : for i in data_in'LOW to data_in'HIGH generate                        --14
FIRST : if (i = 0) generate                                                   --15
  IR_LSB: IR_cell port map (shiftIR, Vdd, int_scan(i),                        --16
  clockIR, updateIR, reset_bar, nTRST, Vdd, data_out(i), scan_out);          --17
end generate;                                                                 --18
SECOND : if ((i = 1) and (data_in'HIGH > 1)) generate                        --19
  IR1 : IR_cell port map (shiftIR, GND, int_scan(i),                          --20
  clockIR, updateIR, reset_bar, nTRST, Vdd, data_out(i), int_scan(i-1));     --21
end generate;                                                                 --22
MIDDLE : if ((i < data_in'HIGH) and (i > 1)) generate                        --23
  IRi : IR_cell port map (shiftIR, data_in(i), int_scan(i),                  --24
  clockIR, updateIR, reset_bar, nTRST, Vdd, data_out(i), int_scan(i-1));     --25
end generate;                                                                 --26
LAST : if (i = data_in'HIGH) generate                                         --27
  IR_MSB : IR_cell port map (shiftIR, data_in(i), scan_in,                   --28
  clockIR, updateIR, reset_bar, nTRST, Vdd, data_out(i), int_scan(i-1));     --29
end generate; end generate ;                                                  --30
end structure;                                                                --31
```

FIGURE 14.6 An IR (instruction register).

4. BYPASS. Selects the single-cell bypass register (instead of the BSR) and allows data to be quickly shifted between ASICs.

The IEEE 1149.1 standard predefines additional optional instructions and also defines the implementation of custom instructions that may use additional TDRs.

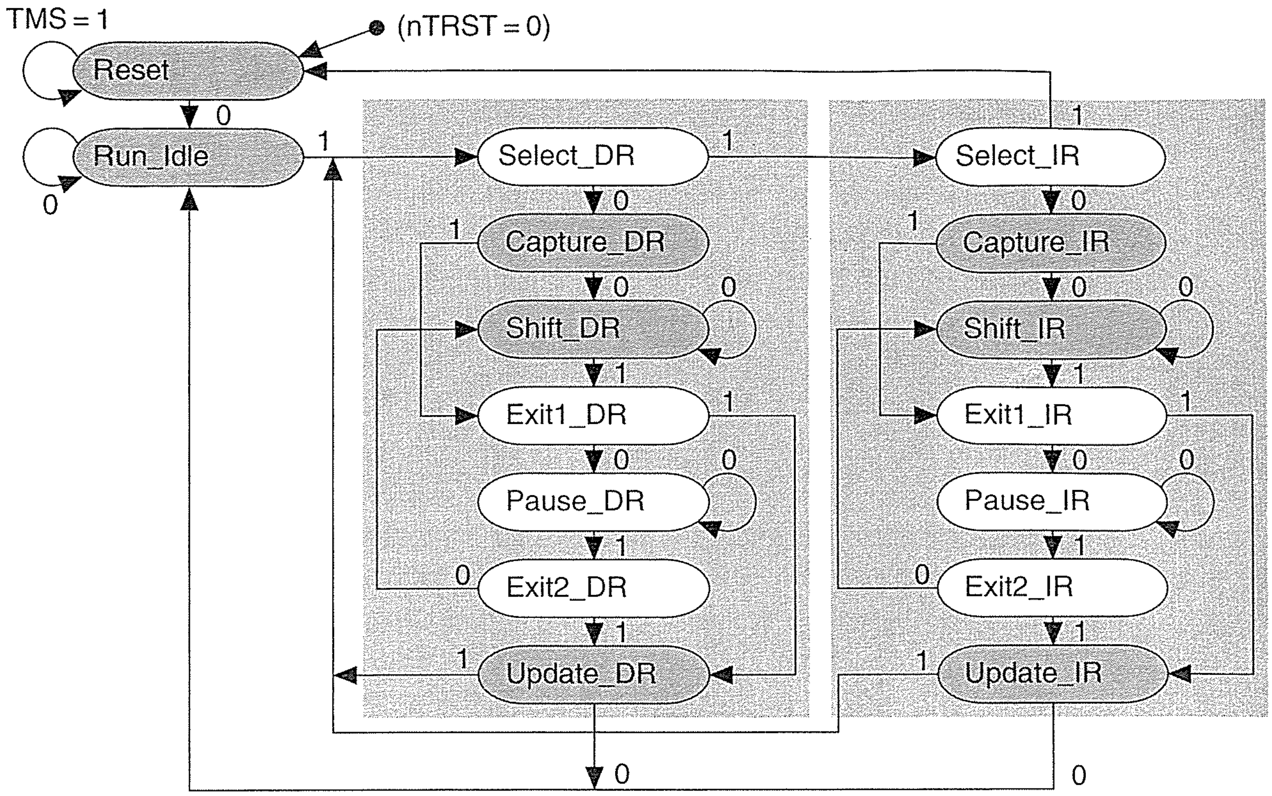
TABLE 14.4 An IR (instruction register) decoder.

```
entity IR_decoder is generic (width : INTEGER := 4); port (
  shiftDR, clockDR, updateDR : BIT; IR_PO : BIT_VECTOR (width-1 downto 0) ;
  test_mode, selectBR, shiftBR, clockBR, shiftBSR, clockBSR, updateBSR : out BIT );
end IR_decoder;

architecture behave of IR_decoder is
type INSTRUCTION is (EXTEST, SAMPLE_PRELOAD, IDCODE, BYPASS);
signal I : INSTRUCTION;
begin process (IR_PO) begin case BIT_VECTOR'( IR_PO(1), IR_PO(0) ) is
  when "00" => I <= EXTEST; when "01" => I <= SAMPLE_PRELOAD;
  when "10" => I <= IDCODE; when "11" => I <= BYPASS;
end case; end process;
test_mode <= '1' when I = EXTEST else '0';
selectBR  <= '1' when (I = BYPASS or I = IDCODE) else '0';
shiftBR <= shiftDR;
clockBR <= clockDR when (I = BYPASS or I = IDCODE) else '1';
shiftBSR <= shiftDR;
clockBSR <= clockDR when (I = EXTEST or I = SAMPLE_PRELOAD) else '1';
updateBSR <= updateDR when (I = EXTEST or I = SAMPLE_PRELOAD) else '0';
end behave;
```

14.2.4 TAP Controller

Figure 14.7 shows the TAP controller finite-state machine. The 16-state diagram contains some symmetry: states with suffix '_DR' operate on the data registers and those with suffix '_IR' apply to the instruction register. All transitions between states are determined by the TMS (test mode select) signal and occur at the rising edge of TCK, the boundary-scan clock. An optional active-low reset signal, nTRST or TRST*, resets the state machine to the initial state, Reset. If the dedicated nTRST is not used, there must be a power-on reset signal (POR)—not an existing system reset signal.

The outputs of the TAP controller are not shown in Figure 14.7, but are derived from each TAP controller state. The TAP controller operates rather like a four-button digital watch that cycles through several states (alarm, stopwatch, 12 hr / 24 hr, countdown timer, and so on) as you press the buttons. Only the shaded states in Figure 14.7 affect the ASIC core logic; the other states are intermediate steps. The pause states let the controller jog in place while the tester reloads its memory with a new set of test vectors, for example.



```
use work.TAP.all; entity TAP_sm_states is                                    --1
  port (TMS, TCK, nTRST : in BIT; S : out TAP_STATE); end TAP_sm_states;     --2

architecture behave of TAP_sm_states is                                     --3
type STATE_ARRAY is array (TAP_STATE, 0 to 1) of TAP_STATE;                  --4
constant T : STATE_ARRAY := ( (Run_Idle, Reset),                            --5
(Run_Idle, Select_DR), (Capture_DR, Select_IR), (Shift_DR, Exit1_DR),        --6
(Shift_DR, Exit1_DR), (Pause_DR, Update_DR), (Pause_DR, Exit2_DR),           --7
(Shift_DR, Update_DR), (Run_Idle, Select_DR), (Capture_IR, Reset),           --8
(Shift_IR, Exit1_IR), (Shift_IR, Exit1_IR), (Pause_IR, Update_IR),           --9
(Pause_IR, Exit2_IR), (Shift_IR, Update_IR), (Run_idle, Select_DR) );       --10
begin process (TCK, nTRST) variable S_i: TAP_STATE; begin                   --11
  if ( nTRST = '0' ) then S_i := Reset;                                     --12
  elsif ( TCK = '1' and TCK'EVENT ) then -- transition on +VE clock edge    --13
    if ( TMS = '1' ) then S_i := T(S_i, 1); else S_i := T(S_i, 0); end if;  --14
  end if; S <= S_i; -- update signal with already updated internal variable --15
end process;                                                                --16
end behave;                                                                 --17
```

FIGURE 14.7 The TAP (test-access port) controller state machine.

Table 14.5 shows the output control signals generated by the TAP state machine. I have taken the unusual step of writing separate entities for the state machine and its outputs. Normally this is bad practice because it makes it difficult for synthesis tools to extract and optimize the logic, for example. This separation of functions reflects the fact that the operation of the TAP controller state machine is precisely defined by the IEEE 1149.1 standard—independent of the implementation of the register cells and number of instructions supported. The model in Table 14.5 contains the following combinational, registered, and gated output signals and will change with different implementations:

- `reset_bar`. Resets the IR to IDCODE (or BYPASS in absence of IDCODE TDR).
- `selectIR`. Connects a register, the IR or a TDR, to `TDO`.
- `enableTDO`. Enables the three-state buffer that drives `TDO`. This allows data to be shifted out of the ASIC on `TDO`, either from the IR or from the DR, in states `shift_IR` or `shift_DR` respectively.
- `shiftIR`. Selects the serial input to the capture flip-flop in the IR cells.
- `clockIR`. Causes data at the input of the IR to be captured or the contents of the IR to be shifted toward `TDO` (depending on `shiftIR`) on the *negative* edge of `TCK` following the *entry* to the states `shift_IR` or `capture_IR`. This is a dirty signal.
- `updateIR`. Clocks the update sequential element on the *positive* edge of `TCK` at the same time as the *exit* from state `update_IR`. This is a dirty signal.
- `shiftDR`, `clockDR`, and `updateDR`. Same functions as corresponding IR signals applied to the TDRs. These signals may be gated to the appropriate TDR by the instruction decoder.

The signals `reset_bar`, `enableTDO`, `shiftIR`, and `shiftDR` are registered or clocked by `TCK` (on the positive edge of `TCK`). We say these signals are **clean** (as opposed to being dirty gated clocks).

14.2.5 Boundary-Scan Controller

Figure 14.8 shows a boundary-scan controller. It contains the following four parts:

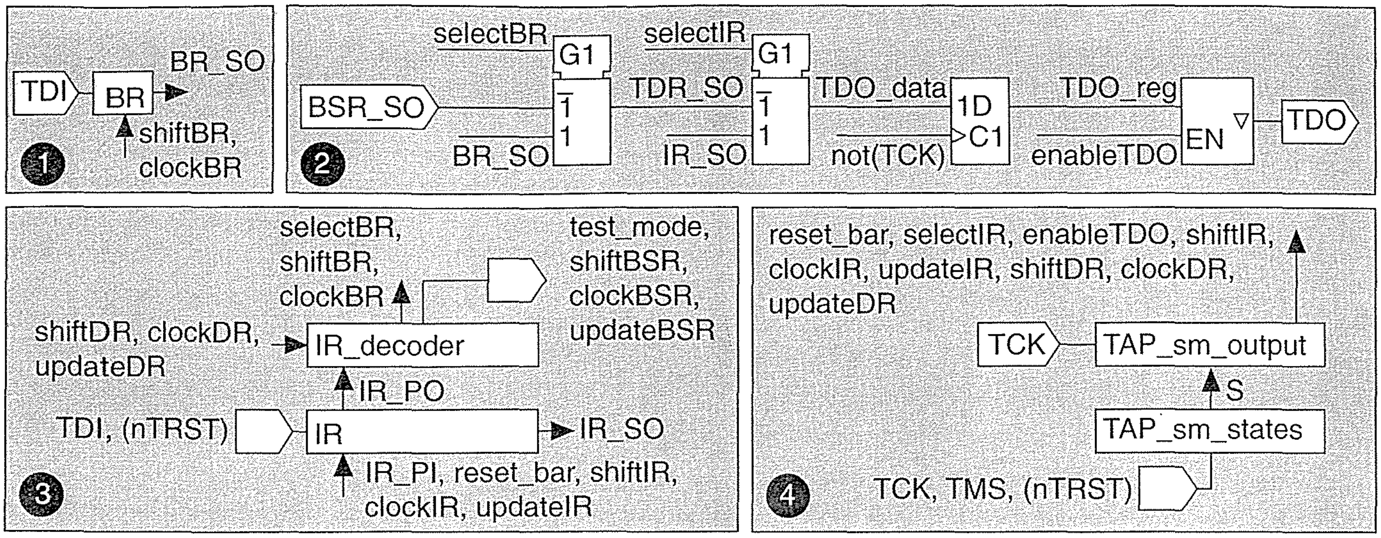
1. Bypass register.
2. `TDO` output circuit. The data to be shifted out of the ASIC on `TDO` is selected from the serial outputs of bypass register (`BR_SO`), instruction register (`IR_SO`), or boundary-scan register (`BSR_SO`). Notice the registered output means that data appears on `TDO` at the *negative* edge of `TCK`. This prevents race conditions between ASICs.
3. Instruction register and instruction decoder.
4. TAP controller.

TABLE 14.5 The TAP (test-access port) control.[1]

State Output	Reset	Run_Idle	Select_DR	Capture_DR	Shift_DR	Exit1_DR	Pause_DR	Exit2_DR	Update_DR	Select_IR	Capture_IR	Shift_IR	Exit1_IR	Pause_IR	Exit2_IR	Update_IR
reset_bar	0R															
selectIR	1	1								1	1	1	1	1	1	1
enableTDO					1R							1R				
shiftIR												1R				
clockIR											0G	0G				
updateIR																1G
shiftDR					1R											
clockDR				0G	0G											
updateDR									1G							

```
use work.TAP.all; entity TAP_sm_output is
port (TCK : in BIT; S : in TAP_STATE; reset_bar, selectIR, enableTDO, shiftIR,
  clockIR, updateIR, shiftDR, clockDR, updateDR : out BIT);
end TAP_sm_output;

architecture behave_1 of TAP_sm_output is begin -- registered outputs
process (TCK) begin if ( (TCK = '0') and TCK'EVENT ) then
  if S = Reset then reset_bar <= '0'; else reset_bar <= '1'; end if;
  if S = Shift_IR or S = Shift_DR then enableTDO <= '1'; else enableTDO <= '0'; end if;
  if S = Shift_IR then ShiftIR <= '1'; else shiftIR <= '0'; end if;
  if S = Shift_DR then ShiftDR <= '1'; else shiftDR <= '0'; end if;
end if;
end process;
process (TCK) begin -- dirty outputs gated with not(TCK)
if (TCK = '0' and (S = Capture_IR or S = Shift_IR))
  then clockIR <= '0'; else clockIR <= '1'; end if;
if (TCK = '0' and (S = Capture_DR or S = Shift_DR))
  then clockDR <= '0'; else clockDR <= '1'; end if;
if TCK = '0' and S=Update_IR then updateIR <= '1'; else updateIR <= '0'; end if;
if TCK = '0' and S=Update_DR then updateDR <= '1'; else updateDR <= '0'; end if;
end process;
selectIR <= '1' when (S = Reset or S = Run_Idle or S = Capture_IR or S = Shift_IR
  or S = Exit1_IR or S = Pause_IR or S = Exit2_IR or S = Update_IR) else '0';
end behave_1;
```

[1] Outputs: G = gated with –TCK, R = registered on falling edge of TCK. Only active levels are shown in the table.



```
library IEEE; use IEEE.std_logic_1164.all; use work.TAP.all;                      --1
entity Control is generic (width : INTEGER := 2); port (TMS, TCK, TDI, nTRST : BIT; --2
  TDO: out STD_LOGIC; BSR_SO : BIT; BSR_PO : BIT_VECTOR (width-1 downto 0);       --3
  shiftBSR, clockBSR, updateBSR, test_mode : out BIT); end Control;               --4

architecture mixed of Control is use work.BST_components.all;                     --5
signal reset_bar, selectIR, enableTDO, shiftIR, clockIR, updateIR, shiftDR,       --6
  clockDR, updateDR, IR_SO, BR_SO, TDO_reg, TDO_data, TDR_SO, selectBR,           --7
  clockBR, shiftBR : BIT;                                                         --8
signal IR_PI, IR_PO : BIT_VECTOR (1 downto 0); signal S : TAP_STATE;              --9
begin                                                                             --10
IR_PI <= "01";                                                                    --11
TDO <= TO_STDULOGIC(TDO_reg) when enableTDO = '1' else 'Z';                       --12
R1 : process (TCK) begin if (TCK='0') then TDO_reg <= TDO_data; end if; end process; --13
TDO_data <= IR_SO when selectIR = '1' else TDR_SO;                                --14
TDR_SO <= BR_SO when selectBR = '1' else BSR_SO;                                  --15
TC1 : TAP_sm_states port map (TMS, TCK, nTRST, S);                                --16
TC2 : TAP_sm_output port map (TCK, S, reset_bar, selectIR, enableTDO,             --17
  shiftIR, clockIR, updateIR, shiftDR, clockDR, updateDR);                        --18
IR1 : IR generic map (width => 2) port map (shiftIR, clockIR, updateIR,           --19
  reset_bar, nTRST, TDI, IR_SO, IR_PI, IR_PO);                                    --20
DEC1 : IR_decoder generic map (width => 2) port map (shiftDR, clockDR, updateDR,  --21
  IR_PO, test_mode, selectBR, shiftBR, clockBR, shiftBSR, clockBSR, updateBSR);   --22
BR1 : BR_cell port map (clockBR, shiftBR, TDI, BR_SO);                            --23
end mixed;                                                                        --24
```

FIGURE 14.8 A boundary-scan controller.

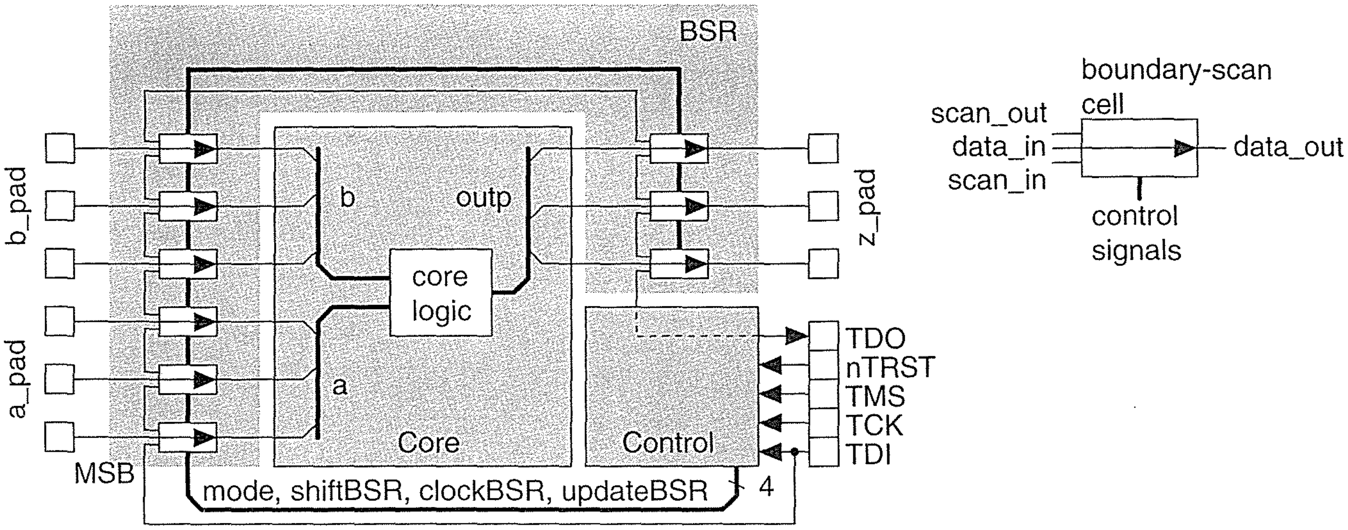
The BSR (and other optional TDRs) are connected to the ASIC core logic outside the BST controller.

14.2.6 A Simple Boundary-Scan Example

Figure 14.9 shows an example of a simple ASIC (our comparator/MUX example) containing boundary scan. The following two packages define the TAP states and the components (these are not essential to understanding what follows, but are included so that the code presented here forms a complete BST model):

```
package TAP is
type TAP_STATE is (reset, run_idle, select_DR, capture_DR,
  shift_DR, exit1_DR, pause_DR, exit2_DR, update_DR, select_IR,
  capture_IR, shift_IR, exit1_IR, pause_IR, exit2_IR, update_IR);
end TAP;
```

```
use work.TAP.all; library IEEE; use IEEE.std_logic_1164.all;
package BST_Components is

component DR_cell port (
  mode, data_in, shiftDR, scan_in, clockDR, updateDR: BIT;
  data_out, scan_out : out BIT );
end component;

component IR_cell port (
  shiftIR, data_in, scan_in, clockIR, updateIR, reset_bar,
  nTRST, reset_value : BIT; data_out, scan_out : out BIT);
end component;

component BR_cell port (
  clockDR,shiftDR,scan_in : BIT; scan_out: out BIT );
end component;

component BSR
  generic (width : INTEGER := 5); port (
  shiftDR, clockDR, updateDR, mode, scan_in : BIT;
  scan_out : out BIT;
  data_in : BIT_VECTOR(width-1 downto 0);
  data_out : out BIT_VECTOR(width-1 downto 0));
end component;

component IR generic (width : INTEGER := 4); port (
  shiftIR, clockIR, updateIR, reset_bar, nTRST,
  scan_in : BIT; scan_out : out BIT;
  data_in : BIT_VECTOR (width-1 downto 0) ;
  data_out : out BIT_VECTOR (width-1 downto 0) );
end component;

component IR_decoder generic (width : INTEGER := 4); port (
  shiftDR, clockDR, updateDR : BIT;
  IR_PO : BIT_VECTOR (width-1 downto 0);
```



```
entity Core is port (a, b : BIT_VECTOR (2 downto 0);
  outp : out BIT_VECTOR (2 downto 0)); end Core;
architecture behave of Core is begin outp <= a when a < b else b;
end behave;

library IEEE; use IEEE.std_logic_1164.all;
entity BST_ASIC is port (TMS, TCK, TDI, nTRST : BIT; TDO : out STD_LOGIC;
  a_PAD, b_PAD : BIT_VECTOR (2 downto 0); z_PAD : out BIT_VECTOR (2 downto 0));
end BST_ASIC;

architecture structure of BST_ASIC is use work.BST_components.all;
component Core port (a, b: BIT_VECTOR (2 downto 0);
  outp: out BIT_VECTOR (2 downto 0)); end component;
for all : Core use entity work.Core(behave);
constant BSR_width : INTEGER := 9;
signal BSR_SO, test_mode, shiftBSR, clockBSR, updateBSR : BIT;
signal BSR_PI, BSR_PO : BIT_VECTOR (BSR_width-1 downto 0);
signal a, b, z : BIT_VECTOR (2 downto 0);
begin BSR_PI <= a_PAD & b_PAD & z ;
  a <= BSR_PO(8 downto 6); b <= BSR_PO(5 downto 3); z_pad <= BSR_PO(2 downto 0);
CORE1 : Core port map (a, b, z);
C1 : Control generic map (width => BSR_width) port map (TMS, TCK, TDI, nTRST,
TDO, BSR_SO, BSR_PO, shiftBSR, clockBSR, updateBSR, test_mode);
BSR1 : BSR generic map (width => BSR_width) port map (shiftBSR, clockBSR,
  updateBSR, test_mode, TDI, BSR_SO, BSR_PI, BSR_PO);
end structure;
```

FIGURE 14.9 A boundary-scan example.

```
  test_mode, selectBR, shiftBR, clockBR, shiftBSR, clockBSR,
  updateBSR: out BIT );
end component;

component TAP_sm_states port (
  TMS, TCK, nTRST : in BIT; S : out TAP_STATE); end component;

component TAP_sm_output port (
  TCK: BIT; S : TAP_STATE; reset_bar, selectIR,
  enableTDO, shiftIR, clockIR, updateIR, shiftDR, clockDR,
  updateDR : out BIT);
end component;

component Control generic (width : INTEGER := 2); port (
  TMS, TCK, TDI, nTRST : BIT; TDO : out STD_LOGIC;
  BSR_SO : BIT; BSR_PO : BIT_VECTOR (width-1 downto 0);
  shiftBSR, clockBSR, updateBSR, test_mode : out BIT);
end component;

component BST_ASIC port (
  TMS, TCK, TDI : BIT; TDO : out STD_LOGIC;
  a_PAD, b_PAD : BIT_VECTOR (2 downto 0);
  z_PAD : out BIT_VECTOR (2 downto 0));
end component;

end;
```

The following testbench, Test_BST, performs these functions:

1. Resets the TAP controller at $t = 10$ ns using nTRST.
2. Continuously clocks the BST clock, TCK, at a frequency of 10 MHz. Rising edges of TCK occur at 100 ns, 200 ns, and so on.
3. Drives a series of values onto the TAP inputs TDI and TMS. The sequence shifts in instruction code '01'(SAMPLE/PRELOAD), followed by '00'(EXTEST).

```
library IEEE; use IEEE.std_logic_1164.all;
library STD; use STD.TEXTIO.all;
entity Test_BST is end;
architecture behave of Test_BST is
component BST_ASIC port (TMS, TCK, TDI, nTRST: BIT;
  TDO : out STD_LOGIC; a_PAD, b_PAD : BIT_VECTOR (2 downto 0);
  z_PAD : out BIT_VECTOR (2 downto 0));
end component;
for all : BST_ASIC use entity work.BST_ASIC(behave);
signal TMS, TCK, TDI, nTRST : BIT; signal TDO : STD_LOGIC;
signal TDI_TMS : BIT_VECTOR (1 downto 0);
signal a_PAD, b_PAD, z_PAD : BIT_VECTOR (2 downto 0);
begin
TDI <= TDI_TMS(1) ; TMS <= TDI_TMS(0) ;
ASIC1 : BST_ASIC port map
```

```
  (TMS, TCK, TDI, nTRST, TDO, a_PAD, b_PAD, z_PAD);
nTRST_DRIVE : process begin
  nTRST <= '1', '0' after 10 ns, '1' after 20 ns; wait;
PAD_DRIVE : process begin
  a_PAD <= ('0', '1', '0'); b_PAD <= ('0', '1', '1'); wait;
end process;
end process;
TCK_DRIVE : process begin -- rising edge at 100 ns
  TCK <= '0' after 50 ns, '1' after 100 ns; wait for 100 ns;
  if (now > 3000 ns) then wait; end if;
end process;
BST_DRIVE : process begin  TDI_TMS <=
                                 -- State after +VE edge:
     ('0', '1') after 0 ns,      -- Reset
     ('0', '0') after 101 ns,    -- Run_Idle
     ('0', '1') after 201 ns,    -- Select_DR
     ('0', '1') after 301 ns,    -- Select_IR
     ('0', '0') after 401 ns,    -- Capture_IR
     ('0', '0') after 501 ns,    -- Shift_IR
     ('1', '0') after 601 ns,    -- Shift_IR
     ('0', '1') after 701 ns,    -- Exit1_IR
     ('0', '1') after 801 ns,    -- Update_IR, 01 = SAMPLE/PRELOAD
     ('0', '1') after 901 ns,    -- Select_DR
     ('0', '0') after 1001 ns,   -- Capture_DR
     ('0', '0') after 1101 ns,   -- Shift_DR
-- shift 111111101 into BSR, TDI(time) = 101111111 starting now
     ('1', '0') after 1201 ns,   -- Shift_DR
     ('0', '0') after 1301 ns,   -- Shift_DR
     ('1', '0') after 1401 ns,   -- Shift_DR -- shift 4 more 1's
     ('1', '0') after 1901 ns,   -- Shift_DR -- in-between
     ('1', '1') after 2001 ns,   -- Exit1_DR
     ('0', '1') after 2101 ns,   -- Update_DR
     ('0', '1') after 2201 ns,   -- Select_DR
     ('0', '1') after 2301 ns,   -- Select_IR
     ('0', '0') after 2401 ns,   -- Capture_IR
     ('0', '0') after 2501 ns,   -- Shift_IR
     ('0', '0') after 2601 ns,   -- Shift_IR
     ('0', '1') after 2701 ns,   -- Exit1_IR
     ('0', '1') after 2801 ns,   -- Update_IR, 00=EXTEST
     ('0', '0') after 2901 ns;   -- Run_Idle
     wait;
end process;
process (TDO, a_pad, b_pad, z_pad) variable L : LINE; begin
  write (L, now, RIGHT, 10); write (L, STRING'(" TDO="));
  if TDO = 'Z' then write (L, STRING'("Z")) ;
    else write (L, TO_BIT(TDO)); end if;
  write (L, STRING'(" PADS=")); write (L, a_pad & b_pad & z_pad);
  writeline (output, L);
```

```
end process;                                                      --64
end behave;                                                       --65
```

Here is the output from this testbench:

```
#      0 ns TDO=0 PADS=000000000
#      0 ns TDO=Z PADS=010011000
#      0 ns TDO=Z PADS=010011010
#    650 ns TDO=1 PADS=010011010
#    750 ns TDO=0 PADS=010011010
#    850 ns TDO=Z PADS=010011010
#   1250 ns TDO=0 PADS=010011010
#   1350 ns TDO=1 PADS=010011010
#   1450 ns TDO=0 PADS=010011010
#   1550 ns TDO=1 PADS=010011010
#   1750 ns TDO=0 PADS=010011010
#   1950 ns TDO=1 PADS=010011010
#   2050 ns TDO=0 PADS=010011010
#   2150 ns TDO=Z PADS=010011010
#   2650 ns TDO=1 PADS=010011010
#   2750 ns TDO=0 PADS=010011010
#   2850 ns TDO=Z PADS=010011010
#   2950 ns TDO=Z PADS=010011101
```

This trace shows the following activities:

- All changes to TDO and at the pads occur at the negative edge of TCK.
- The core logic output is z_pad = '010' and appears at the I/O pads at $t = 0$ ns. This is the smaller of the two inputs, a_pad = '010' and b_pad = '011', and the correct output when the pads are connected to the core logic.
- At $t = 650$ ns the IDCODE instruction '01' is shifted out on TDO (with '1' appearing first). If we had multiple ASICs in the boundary-scan chain, this would show us that the chain was intact.
- At $t = 850$ ns the TDO output is floated (to 'Z') as we exit the shift_IR state.
- At $t = 1200$ ns the TAP controller begins shifting the serial data input from TDI ('111111101') into the BSR.
- At $t = 1250$ ns the BSR data starts shifting out. This is data that was captured during the SAMPLE/PRELOAD instruction from the device input pins, a_pad and b_pad, as well as the driver of the output pins, z_pad. The data appears as the pattern '010011010'. This pattern consists of a_pad = '010', b_pad = '011', followed by z_pad = '010' (notice that TDO does not change at $t = 1650$ ns or 1850 ns).
- At $t = 2150$ ns, TDO is floated after we exit the shift_DR state.
- At $t = 2650$ ns the IDCODE instruction '01' (loaded into the IR as we passed through capture_IR the second time) is again shifted out as we shift the EXTEST instruction from TDI into the IR.

- At $t = 2650$ ns the TDO output is floated after we exit the shift_IR state.
- At $t = 2950$ ns the output, z_pad, is driven with '101'. The inputs a_pad and b_pad remain unchanged since they are driven from outside the chip. The change on z_pad occurs on the negative edge of TCK because the IR is loaded with the instruction EXTEST on the negative edge of TCK. When this instruction is decoded, the signal mode changes (this signal controls the MUX at the output of the BSCs).

Figure 14.10 shows a signal trace from the MTI simulator for the last four negative edges of TCK. Notice that we shifted in the test pattern on TDI in the order '101111111'. The output z_pad (3 bits wide) is last in the BSR (nearest TDO) and thus is driven with the first 3 bits of this pattern, '101'. Forcing '101' onto the ASIC output pins would allow us to check that this pattern is correctly received at inputs of other connected ASICs through the bonding wires and board traces. In a later test cycle we can force '010' onto z_pad to check that both logic levels can be transmitted and received. We may also capture other signals (which are similarly being forced onto the outputs of neighboring ASICs) at the inputs.

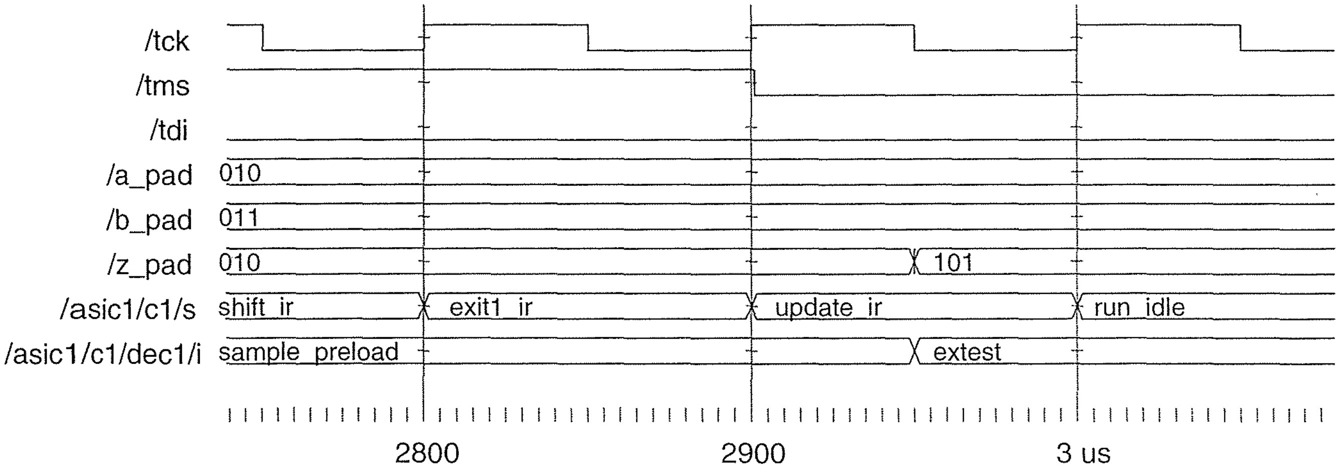


FIGURE 14.10 Results from the MTI simulator for the boundary-scan testbench.

14.2.7 BSDL

The **boundary-scan description language** (**BSDL**) is an extension of IEEE 1149.1 but without any overlap. BSDL uses a subset of VHDL. The BSDL for an ASIC is part of an imaginary data sheet; it is not intended for simulation and does not include models for any boundary-scan components. BSDL is a standard way to describe the features and behavior of an ASIC that includes IEEE 1149.1 boundary scan and a standard way to pass information to test-generation software. Using BSDL, test software can also check that the BST features are correct. As an example, test software can use the BSDL to check that the ASIC uses the correct boundary-scan cells for the instructions that claim to be supported. BSDL cannot prove that an implementation works, however.

The following example BSDL description corresponds to our halfgate ASIC example with BST (this code was generated automatically by the Compass tools):

```
entity asic_p is
generic (PHYSICAL_PIN_MAP : STRING := "DUMMY_PACKAGE");
port (
    pad_a:in BIT_VECTOR (0 to 0);
    pad_z:buffer BIT_VECTOR (0 to 0);
    TCK:in BIT;
    TDI:in BIT;
    TDO:out BIT;
    TMS:in BIT;
    TRST:in BIT);
use STD_1149_1_1994.all;
attribute PIN_MAP of asic_p : entity is PHYSICAL_PIN_MAP;
-- CUSTOMIZE package pin mapping.
constant DUMMY_PACKAGE : PIN_MAP_STRING :=
    "pad_a:(1)," &
    "pad_z:(2)," &
    "TCK:3," &
    "TDI:4," &
    "TDO:5," &
    "TMS:6," &
    "TRST:7";
attribute TAP_SCAN_IN of TDI : signal is TRUE;
attribute TAP_SCAN_MODE of TMS : signal is TRUE;
attribute TAP_SCAN_OUT of TDO : signal is TRUE;
attribute TAP_SCAN_RESET of TRST : signal is TRUE;
-- CUSTOMIZE TCK max freq and safe stop state.
attribute TAP_SCAN_CLOCK of TCK : signal is (20.0e6, BOTH);
attribute INSTRUCTION_LENGTH of asic_p : entity is 3;
attribute INSTRUCTION_OPCODE of asic_p : entity is
    "IDCODE  (001)," &
    "STCTEST  (101)," &
    "INTEST  (100)," &
    "BYPASS  (111)," &
    "SAMPLE  (010)," &
    "EXTEST  (000)";
attribute INSTRUCTION_CAPTURE of asic_p : entity is "001";
-- attribute INSTRUCTION_DISABLE of asic_p : entity is " "
-- attribute INSTRUCTION_GUARD of asic_p : entity is " "
-- attribute INSTRUCTION_PRIVATE of asic_p : entity is " "
attribute IDCODE_REGISTER of asic_p : entity is
    "0000" &              -- 4-bit version
    "0000000000000000" &   -- 16-bit part number
    "00000101011" &        -- 11-bit manufacturer                        --43
    "1";                   -- mandatory LSB                              --44
-- attribute USERCODE_REGISTER of asic_p : entity is " "                 --45
attribute REGISTER_ACCESS of asic_p : entity is                          --46
    "BOUNDARY (STCTEST)";                                                --47
attribute BOUNDARY_CELLS of asic_p : entity is                           --48
    "BC_1, BC_2";                                                        --49
attribute BOUNDARY_LENGTH of asic_p : entity is 2;                       --50
attribute BOUNDARY_REGISTER of asic_p : entity is                        --51
  --  num   cell      port    function  safe [ccell disval rslt]         --52
   "    1 ( BC_2, pad_a(0),     input,  X)," &                           --53
   "    0 ( BC_1, pad_z(0),   output2,  X)";                             --54
  -- " 98 ( BC_1, OE, input, X), " &                                     --55
  -- " 98 ( BC_1, *, control, 0), " &                                    --56
  -- " 99 ( BC_1, myport(0), output3, X, 98, 0, Z);                      --57
end asic_p;                                                              --58
```

The functions of the lines of this BSDL description are as follows:

- Line 2 refers to the ASIC package. We can have the same part (with identical pad numbers on the silicon die) in different ASIC packages. We include the name of the ASIC package in line 2 and the pin mapping between bonding pads and ASIC package pins in lines 14–21.
- Lines 3–10 describe the signal names of inputs and outputs, the TAP pins, and the optional fifth TAP reset signal. The BST signals do not have to be given the names used in the standard: TCK, TDI, and so on.
- Line 11 refers to the VHDL package, STD_1149_1_1994. This is a small VHDL package (just over 100 lines) that contains definitions of the constants, types, and attributes used in a BSDL description. It does not contain any models for simulation.
- Lines 22–25 attach signal names to the required TAP pins and the optional fifth TAP reset signal.
- Lines 26–27 refer to the maximum test clock frequency in hertz, and whether the clock may be stopped in both states or just the low state (just the high state is not valid).
- Line 28 describes a 3-bit IR (in the comparator/MUX example we used a 2-bit IR). Length must be greater than or equal to 2.
- Lines 29–35 describe the three required instruction opcodes and mnemonics (BYPASS, SAMPLE, EXTEST) and three optional instructions: IDCODE, STCTEST (which is a scan test mode), and INTEST (which supports internal testing in the same fashion as EXTEST supports external testing). EXTEST must be all ones; BYPASS must be all zeros. A mnemonic may have more than one opcode (and opcodes may be specified using 'x'). Other instructions that may appear here include CLAMP and HIGHZ, both optional instruc-

tions that were added to 1149.1 (see Supplement A, 1149.1a). String concatenation is used in BSDL to avoid line-break problems.

- Lines 37–39 include instruction attributes INSTRUCTION_DISABLE (for HIGHZ), INSTRUCTION_GUARD (for CLAMP), as well as INSTRUCTION_PRIVATE (for user-defined instructions) that are not used in this example.
- Lines 40–44 describe the IDCODE TDR. The 11-bit manufacturer number is determined from codes assigned by JEDEC Publication 106-A.
- Line 45 describes the USERCODE TDR in a similar fashion to IDCODE, but is not used here.
- Lines 46–47 describe the TDRs for user-defined instructions. In this case the existing BOUNDARY TDR is inserted between TDI and TDO during STCTEST. User-defined instructions listed here may use the other existing IDCODE and BYPASS TDRs or define new TDRs.
- Lines 48–49 list the boundary-scan cells used in the ASIC. These may be any of the following cells defined in the 1149.1 standard and defined in the VHDL package, STD_1149_1_1994: BC_1 (Figs. 10-18, 10-29, 10-31c, 10-31d, and 10-33c), BC_2 (Figs. 10-14, 10-30, 10-32c, 10-32d, 10-35c), BC_3 (Fig. 10-15), BC_4 (Figs. 10-16, 10-17), BC_5 (Fig. 10-41c), BC_6 (Fig. 10-34d). The figure numbers in parentheses here refer to the IEEE 1149.1 standard [IEEE 1149.1b-1994]. Alternatively the cells may be user-defined (and must then be declared in a package).
- Line 50 must be an integer greater than zero and match the number defined by the following register description.
- Lines 51–54 are an array of records, numbered by cell, with seven fields: four required and three that only appear for certain cells. Field 1 specifies the scan cell name as defined in the STD_1149_1_1994 or user-defined package. Field 2 is the port name, with a subscript if the type is BIT_VECTOR. An '*' denotes no connection. Field 3 is one of the following cell functions (with figure or page numbers from the IEEE standard [IEEE 1149.1b-1994]): input (Fig. 10-18), clock (Fig. 10-17), output2 (two-state output, Fig. 10-29), output3 (three-state, Fig. 10-31d), internal (p. 33, 1149.1b), control (Fig. 10-31c), controlr (Fig. 10-33c), bidir_in (a reversible cell acting as an input, Fig. 10-34d), bidir_out (a reversible cell acting as an output, Fig. 10-34d). Field 4, safe, contains the safe value to be loaded into the update flip-flop when otherwise unspecified, with 'X' as a don't care value.
- Lines 55–57 illustrate the use of the optional three fields. Field 5, ccell or control cell, refers to the cell number (98 in this example) of the cell that controls an output or bidirectional cell. The control cell number 98 is a merged cell in this example with an input cell, input signal name OE, also labeled as cell number 98. The ASIC input OE (for output enable) thus directly controls (enables) the ASIC three-state output, myport(0).

The boundary-scan standard may seem like a complicated way to test the connections *outside* an ASIC. However, the IEEE 1149.1 standard also gives us a method to communicate with test circuits *inside* an ASIC. Next, we turn our attention from problems at the board level to problems that may occur within the ASIC.

14.3 Faults

Fabrication of an ASIC is a complicated process requiring hundreds of processing steps. Problems may introduce a **defect** that in turn may introduce a **fault** (Sabnis [1990] describes **defect mechanisms**). Any problem during fabrication may prevent a transistor from working and may break or join interconnections. Two common types of defects occur in metallization [Rao, 1993]: either underetching the metal (a problem between long, closely spaced lines), which results in a **bridge** or short circuit (**shorts**) between adjacent lines, or overetching the metal and causing **breaks** or open circuits (**opens**). Defects may also arise after chip fabrication is complete—while testing the wafer, cutting the die from the wafer, or mounting the die in a package. Wafer probing, wafer saw, die attach, wire bonding, and the intermediate handling steps each have their own defect and failure mechanisms. Many different materials are involved in the packaging process that have different mechanical, electrical, and thermal properties, and these differences can cause defects due to corrosion, stress, adhesion failure, cracking, and peeling. Yield loss also occurs from human error—using the wrong mask, incorrectly setting the implant dose—as well as from physical sources: contaminated chemicals, dirty etch sinks, or a troublesome process step. It is possible to repeat or **rework** some of the reversible steps (a lithography step, for example—but not etching) if there are problems. However, reliance on rework indicates a poorly controlled process.

14.3.1 Reliability

It is possible for defects to be nonfatal but to cause failures early in the life of a product. We call this **infant mortality**. Most products follow the same kinds of trend for failures as a function of life. Failure rates decrease rapidly to a low value that remains steady until the end of life when failure rates increase again; this is called a **bathtub curve**. The end of a product lifetime is determined by various **wearout mechanisms** (usually these are controlled by an exponential energy process). Some of the most important wearout mechanisms in ASICs are hot-electron wearout, electromigration, and the failure of antifuses in FPGAs.

We can catch some of the products that are susceptible to early failure using **burn-in**. Many failure mechanisms have a failure rate proportional to exp $(-E_a/\mathrm{kT})$. This is the **Arrhenius equation**, where E_a is a known **activation energy** (k is Boltzmann's constant, 8.62×10^{-5} eVK^{-1}, and T the absolute temperature). Operating an ASIC at an elevated temperature accelerates this type of failure mechanism. Depending on the physics of the failure mechanism, additional stresses, such as elevated current or voltage, may also accelerate failures. The longer and harsher the burn-in conditions, the more likely we are to find problems, but the more costly the process and the more costly the parts.

We can measure the overall **reliability** of any product using the **mean time between failures** (**MTBF**) for a repairable product or **mean time to failure** (**MTTF**) for a fatal failure. We also use **failures in time** (**FITs**) where 1 FIT equals a single failure in 10^9 hours. We can sum the FITs for all the components in a product to determine an overall measure for the product reliability. Suppose we have a system with the following components:

- Microprocessor (standard part) 5 FITs
- 100 TTL parts, 50 parts at 10 FITs, 50 parts at 15 FITs
- 100 RAM chips, 6 FITs

The overall failure rate for this system is $5 + 50 \times 10 + 50 \times 15 + 100 \times 6 = 1855$ FITs. Suppose we could reduce the component count using ASICs to the following:

- Microprocessor (custom) 7 FITs
- 9 ASICs, 10 FITs
- 5 SIMMs, 15 FITs

The failure rate is now $10 + 9 \times 10 + 5 \times 15 = 175$ FITs, or about an order of magnitude lower. This is the rationale behind the Sun SparcStation 1 design described in Section 1.3, "Case Study."

14.3.2 Fault Models

Table 14.6 shows some of the causes of faults. The first column shows the **fault level**—whether the fault occurs in the logic gates on the chip or in the package. The second column describes the **physical fault**. There are too many of these and we need a way to reduce and simplify their effects—by using a fault model.

There are several types of **fault model**. First, we simplify things by mapping from a physical fault to a **logical fault**. Next, we distinguish between those logical faults that degrade the ASIC performance and those faults that are fatal and stop the ASIC from working at all. There are three kinds of logical faults in Table 14.6: a degradation fault, an open-circuit fault, and a short-circuit fault.

A **degradation fault** may be a **parametric fault** or **delay fault** (**timing fault**). A parametric fault might lead to an incorrect switching threshold in a TTL/CMOS level converter at an input, for example. We can test for parametric faults using a production tester. A **delay fault** might lead to a critical path being slower than specification. Delay faults are much harder to test in production. An **open-circuit fault** results from physical faults such as a bad contact, a piece of metal that is missing or overetched, or a break in a polysilicon line. These physical faults all result in failure to transmit a logic level from one part of a circuit to another—an open circuit. A **short-circuit fault** results from such physical faults as: underetching of metal; spiking, pinholes or shorts across the gate oxide; and diffusion shorts. These faults result

TABLE 14.6 Mapping physical faults to logical faults.

		Logical fault		
Fault level	**Physical fault**	**Degradation fault**	**Open-circuit fault**	**Short-circuit fault**
Chip				
	Leakage or short between package leads	●		●
	Broken, misaligned, or poor wire bonding		●	
	Surface contamination, moisture	●		
	Metal migration, stress, peeling		●	●
	Metallization (open or short)		●	●
Gate				
	Contact opens		●	
	Gate to S/D junction short	●		●
	Field-oxide parasitic device	●		●
	Gate-oxide imperfection, spiking	●		●
	Mask misalignment	●		●

in a circuit being accidentally connected—a short circuit. Most short-circuit faults occur in interconnect; often we call these **bridging faults** (BF). A BF usually results from **metal coverage** problems that lead to shorts. You may see reference to **feedback bridging faults** and **nonfeedback bridging faults**, a useful distinction when trying to predict the results of faults on logic operation. Bridging faults are a frequent problem in CMOS ICs.

14.3.3 Physical Faults

Figure 14.11 shows the following examples of physical faults in a logic cell:

- F1 is a short between m1 lines and connects node n1 to VSS.
- F2 is an open on the poly layer and disconnects the gate of transistor t1 from the rest of the circuit.
- F3 is an open on the poly layer and disconnects the gate of transistor t3 from the rest of the circuit.
- F4 is a short on the poly layer and connects the gate of transistor t4 to the gate of transistor t5.
- F5 is an open on m1 and disconnects node n4 from the output Z1.
- F6 is a short on m1 and connects nodes p5 and p6.
- F7 is a nonfatal defect that causes necking on m1.

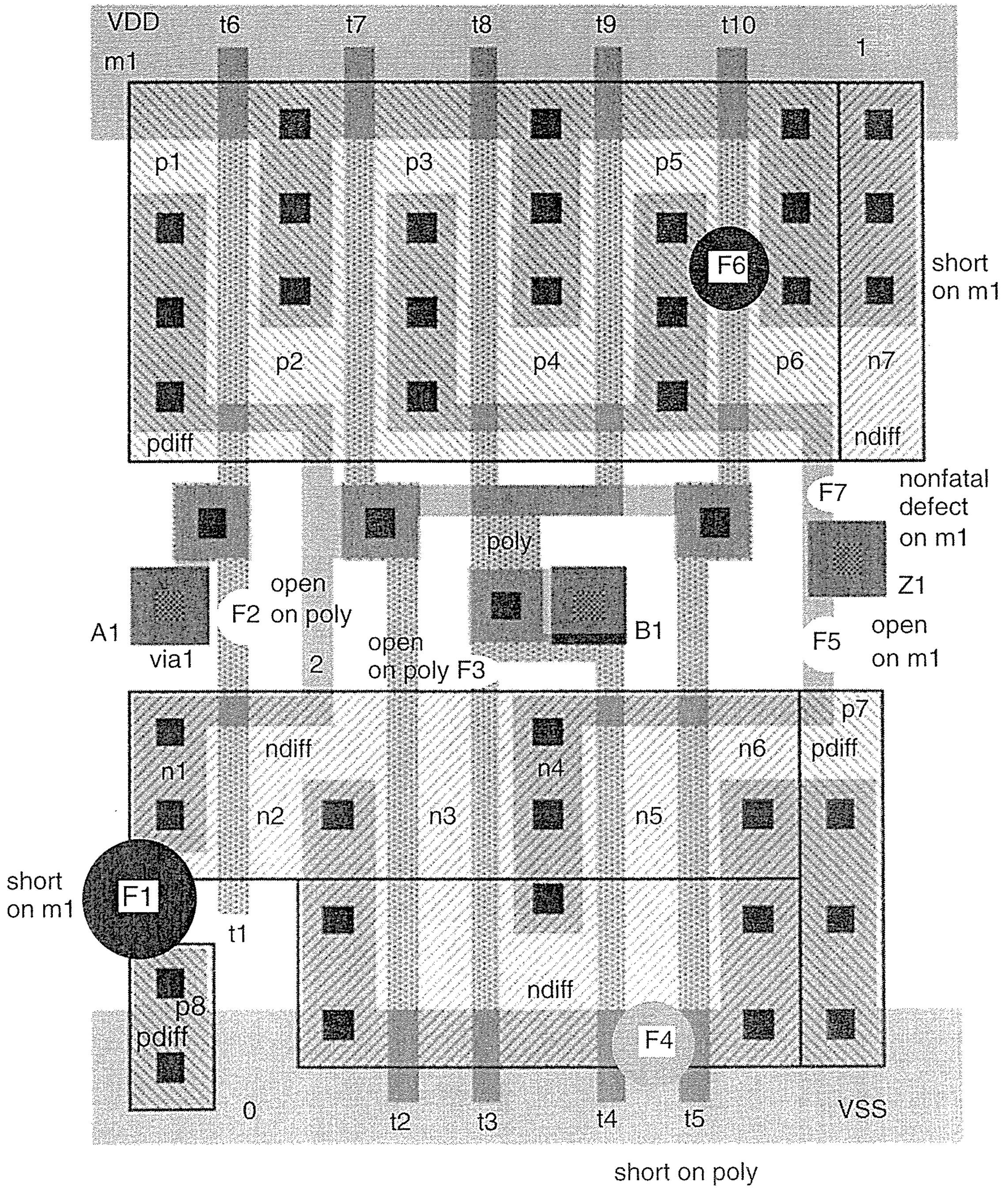


FIGURE 14.11 Defects and physical faults. Many types of defects occur during fabrication. Defects can be of any size and on any layer. Only a few small sample defects are shown here using a typical standard cell as an example. Defect density for a modern CMOS process is of the order of 1 cm^{-2} or less across a whole wafer. The logic cell shown here is approximately $64 \times 32 \lambda^2$, or 250 μm^2 for a $\lambda = 0.25$ μm process. We would thus have to examine approximately 1 cm^{-2}/250 μm^2 or 400,000 such logic cells to find a single defect.

Once we have reduced the large number of physical faults to fewer logical faults, we need a model to predict their effect. The most common model is the **stuck-at fault model**.

14.3.4 Stuck-at Fault Model

The **single stuck-at fault** (SSF) model assumes that there is just one fault in the logic we are testing. We use a single stuck-at fault model because a **multiple stuck-at fault model** that could handle several faults in the logic at the same time is too complicated to implement. We hope that any multiple faults are caught by single stuck-at fault tests [Agarwal and Fung, 1981; Hughes and McCluskey, 1986]. In practice this seems to be true.

There are other fault models. For example, we can assume that faults are located in the transistors using a **stuck-on fault** and **stuck-open fault** (or **stuck-off fault**). Fault models such as these are more realistic in that they more closely model the actual physical faults. However, in practice the simple SSF model has been found to work—and work well. We shall concentrate on the SSF model.

In the SSF model we further assume that the effect of the physical fault (whatever it may be) is to create only two kinds of logical fault. The two types of logical faults or **stuck-at faults** are: a **stuck-at-1 fault** (abbreviated to SA1 or s@1) and a **stuck-at-0 fault** (SA0 or s@0). We say that we **place faults** (**inject faults**, **seed faults**, or **apply faults**) on a node (or net), on an input of a circuit, or on an output of a circuit. The location at which we place the fault is the **fault origin**.

A **net fault** forces all the logic cell inputs that the net drives to a logic `'1'` or `'0'`. An **input fault** attached to a logic cell input forces the logic cell input to a `'1'` or `'0'`, but does not affect other logic cell inputs on the same net. An **output fault** attached to the output of a logic cell can have different strengths. If an output fault is a **supply-strength fault** (or **rail-strength** fault) the logic-cell output node and every other node on that net is forced to a `'1'` or `'0'`—as if all these nodes were connected to one of the supply rails. An alternative assigns the same strength to the output fault as the drive strength of the logic cell. This allows contention between outputs on a net driving the same node. There is no standard method of handling **output-fault strength**, and no standard for using types of stuck-at faults. Usually we do not inject net faults; instead we inject only input faults and output faults. Some people use the term **node fault**—but in different ways to mean either a net fault, input fault, or output fault.

We usually inject stuck-at faults to the inputs and outputs, the pins, of logic cells (AND gates, OR gates, flip-flops, and so on). We do not inject faults to the internal nodes of a flip-flop, for example. We call this a **pin-fault model** and say the fault level is at the **structural level**, gate level, or cell level. We could apply faults to the internal logic of a logic cell (such as a flip-flop) and (the fault level would then be at the transistor level or switch level. We do not use transistor-level or switch-level fault models because there is often no need. From experience, but not from any theoretical reason, it turns out that using a fault model that applies faults at the logic-cell level is sufficient to catch the bad chips in a production test.

When a fault changes the circuit behavior, the change is called the **fault effect**. Fault effects travel through the circuit to other logic cells causing other fault effects. This phenomenon is **fault propagation**. If the fault level is at the structural level, the phenomenon is **structural fault propagation**. If we have one or more large functional blocks in a design, we want to apply faults to the functional blocks only at the inputs and outputs of the blocks. We do not want to place (or cannot place) faults inside the blocks, but we do want faults to propagate through the blocks. This is **behavioral fault propagation**.

Designers adjust the fault level to the appropriate level at which they think there may be faults. Suppose we are performing a fault simulation on a board and we have already tested the chips. Then we might set the fault level to the chip level, placing faults only at the chip pins. For ASICs we use the logic-cell level. You have to be careful, though, if you mix behavioral level and structural level models in a **mixed-level fault simulation**. You need to be sure that the behavioral models propagates faults correctly. In particular, if the behavioral model responds to faults on its inputs by propagating too many unknown `'X'` values to its outputs, this will decrease the fault coverage, because the model is hiding the logic beyond it.

14.3.5 Logical Faults

Figure 14.12 and the following list show how the defects and physical faults of Figure 14.11 translate to logical faults (not all physical faults translate to logical faults—most do not):

- F1 translates to node n1 being stuck at 0, equivalent to A1 being stuck at 1.
- F2 will probably result in node n1 remaining high, equivalent to A1 being stuck at 0.
- F3 will affect half of the *n*-channel pull-down stack and may result in a degradation fault, depending on what happens to the floating gate of T3. The cell will still work, but the fall time at the output will approximately double. A fault such as this in the middle of a chain of logic is extremely hard to detect.
- F4 is a bridging fault whose effect depends on the relative strength of the transistors driving this node. The fault effect is not well modeled by a stuck-at fault model.
- F5 completely disables half of the *n*-channel pulldown stack and will result in a degradation fault.
- F6 shorts the output node to VDD and is equivalent to Z1 stuck at 1.
- Fault F7 could result in infant mortality. If this line did break due to electromigration the cell could no longer pull Z1 up to VDD. This would translate to a Z1 stuck at 0. This fault would probably be fatal and stop the ASIC working.

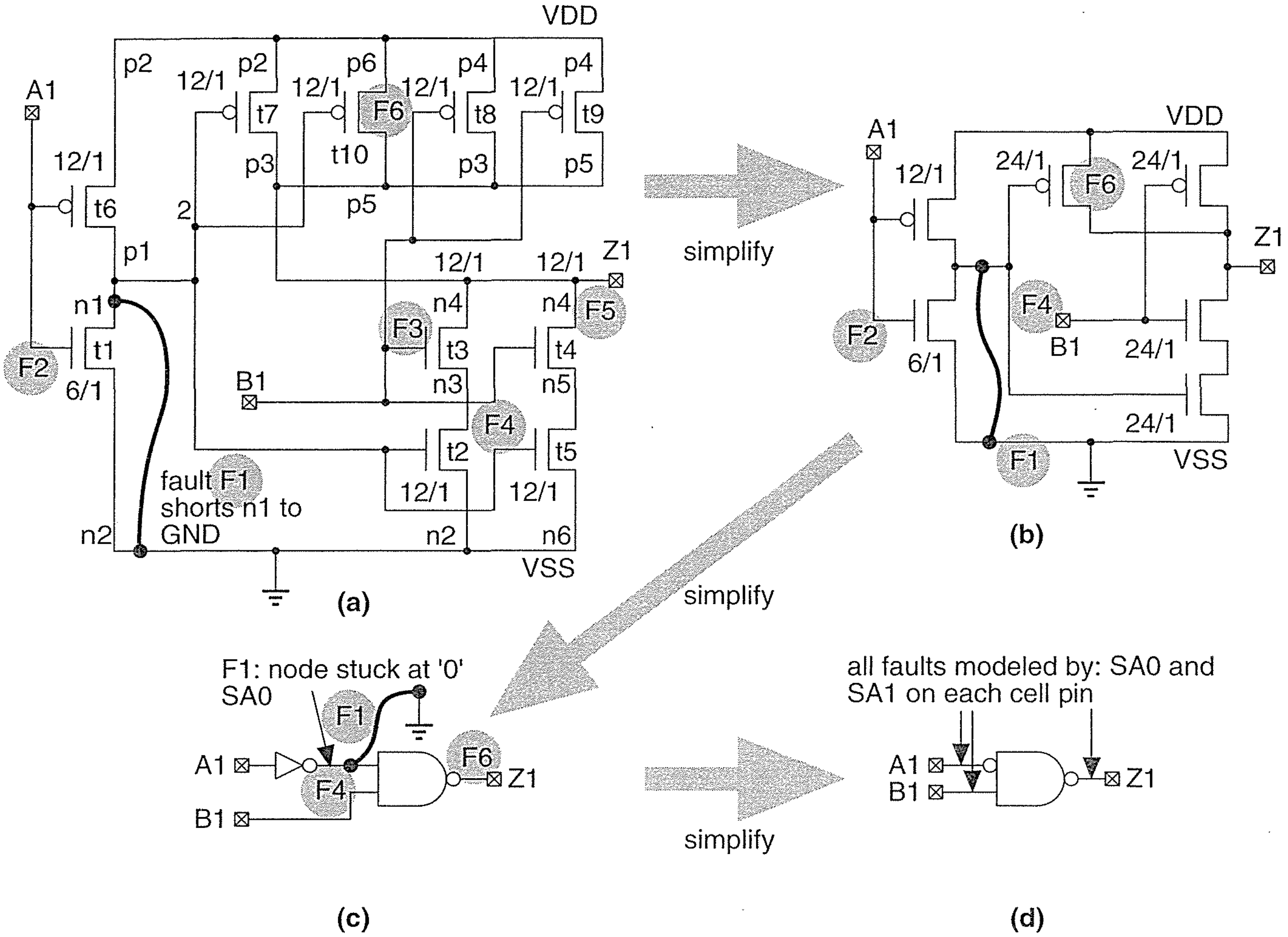


FIGURE 14.12 Fault models. (a) Physical faults at the layout level (problems during fabrication) shown in Figure 14.11 translate to electrical problems on the detailed circuit schematic. The location and effect of fault F1 is shown. The locations of the other fault examples from Figure 14.11 (F2–F6) are shown, but not their effect. (b) We can translate some of these faults to the simplified transistor schematic. (c) Only a few of the physical faults still remain in a gate-level fault model of the logic cell. (d) Finally at the functional-level fault model of a logic cell, we abandon the connection between physical and logical faults and model all faults by stuck-at faults. This is a very poor model of the physical reality, but it works well in practice.

14.3.6 IDDQ Test

When they receive a prototype ASIC, experienced designers measure the resistance between VDD and GND pins. Providing there is not a short between VDD and GND, they connect the power supplies and measure the power-supply current. From experience they know that a supply current of more than a few milliamperes indicates a bad chip. This is exactly what we want in production test: Find the bad chips quickly, get them off the tester, and save expensive tester time. An **IDDQ** (IDD stands for the supply current, and Q stands for quiescent) test is one of the first production tests applied to a chip on the tester, after the chip logic has been initialized [Gulati and Hawkins, 1993; Rajsuman, 1994]. High supply current can result from bridging faults that we described in Section 14.3.2. For example, the bridging fault F4 in Figure 14.11 and Figure 14.12 would cause excessive IDDQ if node n1 and input B1 are being driven to opposite values.

14.3.7 Fault Collapsing

Figure 14.13(a) shows a test for a stuck-at-1 output of a two-input NAND gate. Figure 14.13(b) shows tests for other stuck-at faults. We assume that the NAND gate still works correctly in the **bad circuit** (also called the **faulty circuit** or **faulty machine**) even if we have an input fault. The input fault on a logic cell is presumed to arise either from a fault from a preceding logic cell or a fault on the connection to the input.

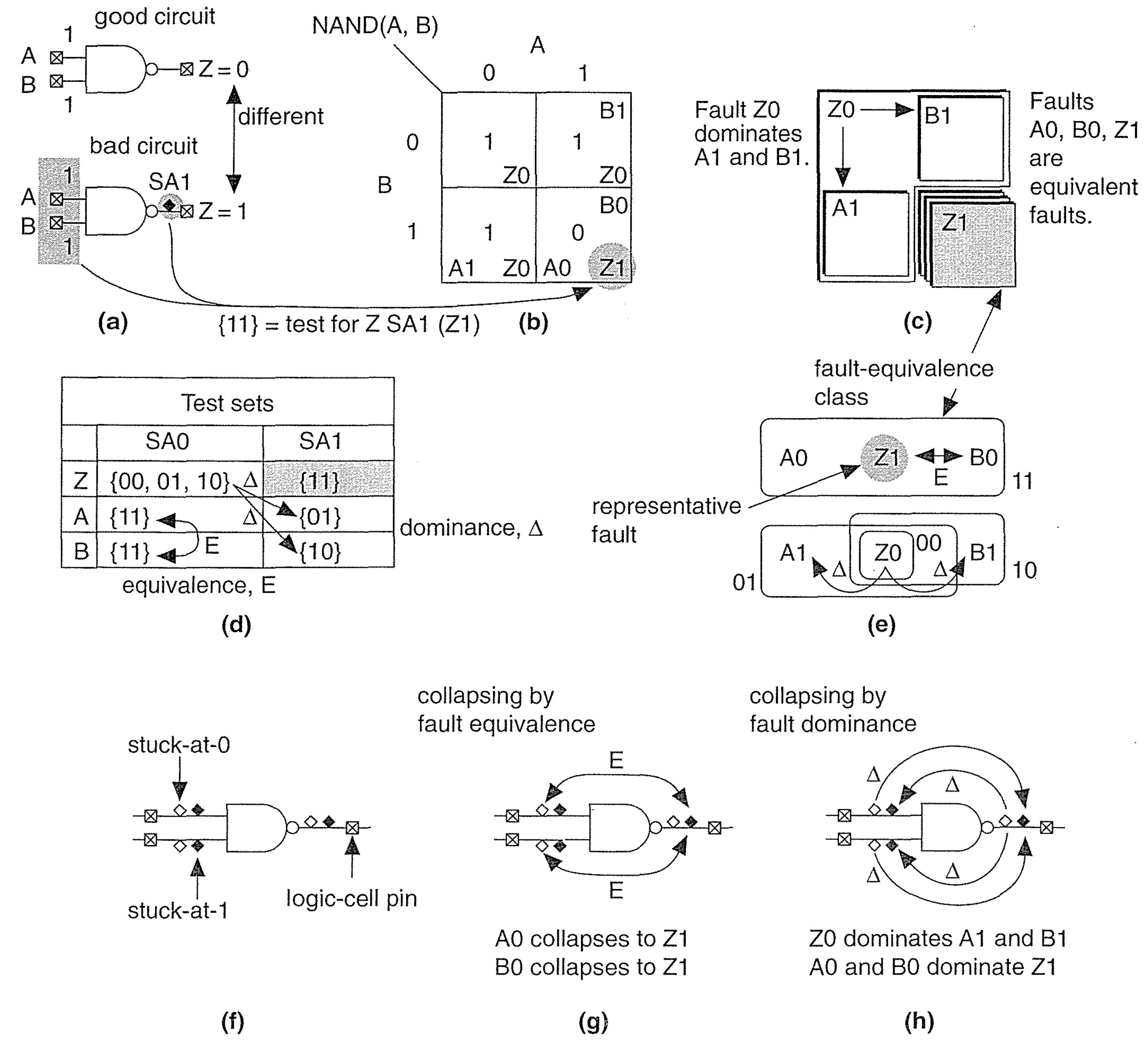
Stuck-at faults attached to different points in a circuit may produce identical fault effects. Using **fault collapsing** we can group these **equivalent faults** (or **indistinguishable faults**) into a **fault-equivalence class**. To save time we need only consider one fault, called the **prime fault** or **representative fault**, from a fault-equivalence class. For example, Figure 14.13(a) and (b) show that a stuck-at-0 input and a stuck-at-1 output are equivalent faults for a two-input NAND gate. We only need to check for one fault, Z1 (output stuck at 1), to catch any of the equivalent faults.

Suppose that any of the tests that detect a fault B also detects fault A, but only some of the tests for fault A also detect fault B. W say A is a **dominant fault**, or that fault A dominates fault B (this the definition of fault dominance that we shall use, some texts say fault B dominates fault A in this situation). Clearly to reduce the number of tests using **dominant fault collapsing** we will pick the test for fault B. For example, Figure 14.13(c) shows that the output stuck at 0 dominates either input stuck at 1 for a two-input NAND. By testing for fault A1, we automatically detect the fault Z1. Confusion over dominance arises because of the difference between focusing on faults (Figure 14.13d) or test vectors (Figure 14.13e).

Figure 14.13(f) shows the six stuck-at faults for a two-input NAND gate. We can place SA1 or SA0 on each of the two input pins (four faults in total) and SA1 or SA0 on the output pins. Using fault equivalence (Figure 14.13g) we can collapse six faults to four: SA1 on each input, and SA1 or SA0 on the output. Using fault dominance (Figure 14.13h) we can collapse six faults to three. There is no way to tell the difference between equivalent faults, but if we use dominant fault collapsing we may lose information about the fault location.

14.3.8 Fault-Collapsing Example

Figure 14.14 shows an example of fault collapsing. Using the properties of logic cells to reduce the number of faults that we need to consider is called **gate collapsing**. We can also use **node collapsing** by examining the effect of faults on the same



	SA0	SA1
Z	{00, 01, 10} Δ	{11}
A	{11} Δ	{01}
B	{11} E	{10}

FIGURE 14.13 Fault dominance and fault equivalence. (a) We can test for fault Z0 (Z stuck at 0) by applying a test vector that makes the bad (faulty) circuit produce a different output than the good circuit. (b) Some test vectors provide tests for more than one fault. (c) A test for A stuck at 1 (A1) will also test for Z stuck at 0; Z0 dominates A1. The fault effects of faults: A0, B0 and Z1 are the same. These faults are equivalent. (d) There are six sets of input vectors that test for the six stuck-at faults. (e) We only need to choose a subset of all test vectors that test for all faults. (f) The six stuck-at faults for a two-input NAND logic cell. (g) Using fault equivalence we can collapse six faults to four. (h) Using fault dominance we can collapse six faults to three.

node. Consider two inverters in series. An output fault on the first inverter collapses with the node fault on the net connecting the inverters. We can collapse the node fault in turn with the input fault of the second inverter. The details of fault collapsing depends on whether the simulator uses net or pin faults, the fanin and fanout of nodes, and the output fault-strength model used.

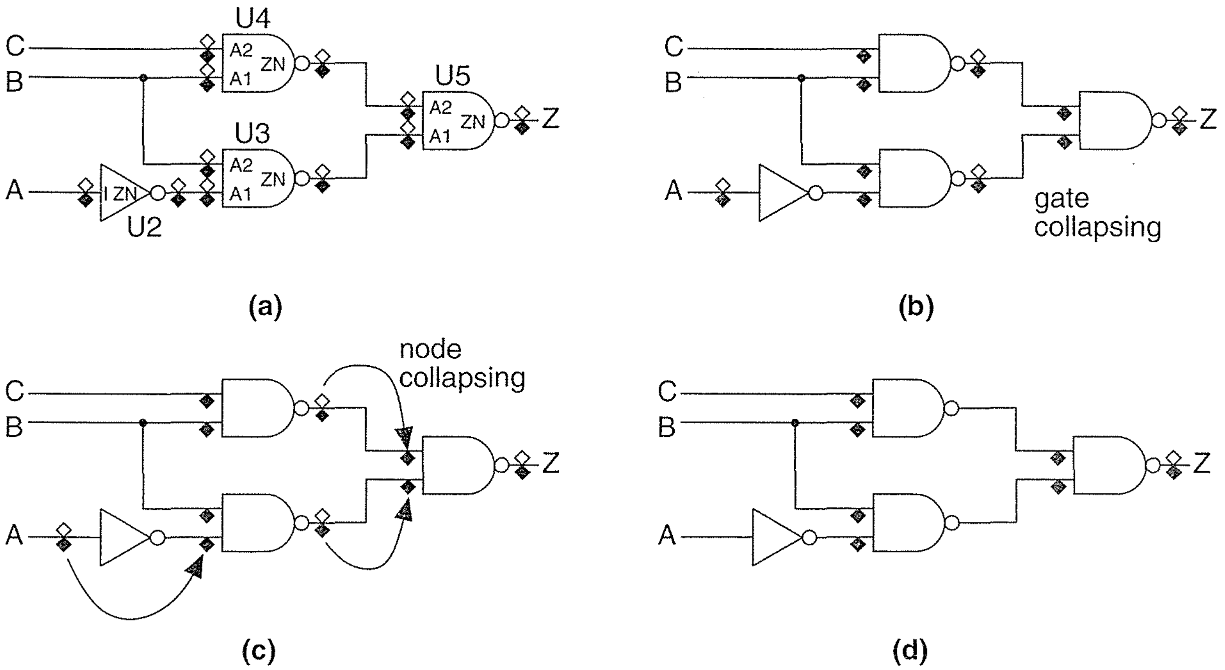


FIGURE 14.14 Fault collapsing for A'B + BC. (a) A pin-fault model. Each pin has stuck-at-0 and stuck-at-1 faults. (b) Using fault equivalence the pin faults at the input pins and output pins of logic cells are collapsed. This is gate collapsing. (c) We can reduce the number of faults we need to consider further by collapsing equivalent faults on nodes and between logic cells. This is node collapsing. (d) The final circuit has eight stuck-at faults (reduced from the 22 original faults). If we wished to use fault dominance we could also eliminate the stuck-at-0 fault on Z. Notice that in a pin-fault model we cannot collapse the faults U4.A1.SA1 and U3.A2.SA1 even though they are on the same net.

14.4 Fault Simulation

We use **fault simulation** after we have completed logic simulation to see what happens in a design when we deliberately introduce faults. In a production test we only have access to the package pins—the **primary inputs** (**PIs**) and **primary outputs** (**POs**). To test an ASIC we must devise a series of sets of input patterns that will detect any faults. A **stimulus** is the application of one such set of inputs (a **test**

vector) to the PIs of an ASIC. A typical ASIC may have several hundred PIs and therefore each test vector is several hundred bits long. A **test program** consists of a set of test vectors. Typical ASIC test programs require tens of thousands and sometimes hundreds of thousands of test vectors.

The **test-cycle time** is the period of time the tester requires to apply the stimulus, sense the POs, and check that the actual output is equal to the expected output. Suppose the test cycle time is 100 ns (corresponding to a test frequency of 10 MHz), in which case we might **sense** (or **strobe**) the POs at 90 ns after the beginning of each test cycle. Using fault simulation we mimic the behavior of the production test. The fault simulator deliberately introduces all possible faults into our ASIC, one at a time, to see if the test program will find them. For the moment we dodge the problem of how to create the thousands of test vectors required in a typical test program and focus on fault simulation.

As each fault is inserted, the fault simulator runs our test program. If the fault simulation shows that the POs of the faulty circuit are different than the PIs of the good circuit at any strobe time, then we have a **detected fault**; otherwise we have an **undetected fault**. The list of **fault origins** is collected in a file and as the faults are inserted and simulated, the results are recorded and the faults are marked according to the result. At the end of fault simulation we can find the **fault coverage**,

$$\text{fault coverage} = \text{detected faults} / \text{ detectable faults.} \tag{14.1}$$

Detected faults and detectable faults will be defined in Section 14.4.5, after the description of fault simulation. For now assume that we wish to achieve close to 100 percent fault coverage. How does fault coverage relate to the ASIC defect level?

Table 14.7 shows the results of a typical experiment to measure the relationship between single stuck-at fault coverage and AQL. Table 14.7 completes a circle with test and repair costs in Table 14.1 and defect levels in Table 14.2. These experimental results are the only justification (but a good one) for our assumptions in adopting the SSF model. We are not quite sure why this model works so well, but, being engineers, as long as it continues to work we do not worry too much.

TABLE 14.7 Average quality level as a function of single stuck-at fault coverage.

Fault coverage	Average defect level	Average quality level (AQL)
50%	7%	93 %
90%	3%	97 %
95%	1%	99 %
99%	0.1%	99.9 %
99.9%	0.01%	99.99 %

There are several algorithms for fault simulation: serial fault simulation, parallel fault simulation, and concurrent fault simulation. Next, we shall discuss each of these types of fault simulation in turn.

14.4.1 Serial Fault Simulation

Serial fault simulation is the simplest fault-simulation algorithm. We simulate two copies of the circuit, the first copy is a good circuit. We then pick a fault and insert it into the faulty circuit. In test terminology, the circuits are called **machines**, so the two copies are a **good machine** and a **faulty machine**. We shall continue to use the term *circuit* here to show the similarity between logic and fault simulation (the simulators are often the same program used in different modes). We then repeat the process, simulating one faulty circuit at a time. Serial simulation is slow and is impractical for large ASICs.

14.4.2 Parallel Fault Simulation

Parallel fault simulation takes advantage of multiple bits of the words in computer memory. In the simplest case we need only one bit to represent either a `'1'` or `'0'` for each node in the circuit. In a computer that uses a 32-bit word memory we can simulate a set of 32 copies of the circuit at the same time. One copy is the good circuit, and we insert different faults into the other copies. When we need to perform a logic operation, to model an AND gate for example, we can perform the operation across all bits in the word simultaneously. In this case, using one bit per node on a 32-bit machine, we would expect parallel fault simulation to be about 32 times faster than serial simulation. The number of bits per node that we need in order to simulate each circuit depends on the number of states in the logic system we are using. Thus, if we use a four-state system with `'1'`, `'0'`, `'X'` (unknown), and `'Z'` (high-impedance) states, we need two bits per node.

Parallel fault simulation is not quite as fast as our simple prediction because we have to simulate all the circuits in parallel until the last fault in the current set is detected. If we use serial simulation we can stop as soon as a fault is detected and then start another fault simulation. Parallel fault simulation is faster than serial fault simulation but not as fast as concurrent fault simulation. It is also difficult to include behavioral models using parallel fault simulation.

14.4.3 Concurrent Fault Simulation

Concurrent fault simulation is the most widely used fault-simulation algorithm and takes advantage of the fact that a fault does not affect the whole circuit. Thus we do not need to simulate the whole circuit for each new fault. In concurrent simulation we first completely simulate the good circuit. We then inject a fault and resimulate a copy of only that part of the circuit that behaves differently (this is the **diverged circuit**). For example, if the fault is in an inverter that is at a primary output, only the inverter needs to be simulated—we can remove everything preceding the inverter.

Keeping track of exactly which parts of the circuit need to be diverged for each new fault is complicated, but the savings in memory and processing that result allow hundreds of faults to be simulated concurrently. Concurrent simulation is split into several chunks, you can usually control how many faults (usually around 100) are simulated in each chunk or **pass**. Each pass thus consists of a series of test cycles. Every circuit has a unique **fault-activity signature** that governs the divergence that occurs with different test vectors. Thus every circuit has a different optimum setting for **faults per pass**. Too few faults per pass will not use resources efficiently. Too many faults per pass will overflow the memory.

14.4.4 Nondeterministic Fault Simulation

Serial, parallel, and concurrent fault-simulation algorithms are forms of **deterministic fault simulation.** In each of these algorithms we use a set of test vectors to simulate a circuit and discover which faults we can detect. If the fault coverage is inadequate, we modify the test vectors and repeat the fault simulation. This is a very time-consuming process.

As an alternative we give up trying to simulate every possible fault and instead, using **probabilistic fault simulation**, we simulate a subset or sample of the faults and extrapolate fault coverage from the sample.

In **statistical fault simulation** we perform a fault-free simulation and use the results to predict fault coverage. This is done by computing measures of observability and controllability at every node.

We know that a node is not stuck if we can make the node toggle—that is, change from a `'0'` to `'1'` or vice versa. A **toggle test** checks which nodes toggle as a result of applying test vectors and gives a statistical estimate of **vector quality**, a measure of faults detected per test vector. There is a strong correlation between high-quality test vectors, the vectors that will detect most faults, and the test vectors that have the highest **toggle coverage**. Testing for nodes toggling simply requires a single logic simulation that is much faster than complete fault simulation.

We can obtain a considerable improvement in fault simulation speed by putting the high-quality test vectors at the beginning of the simulation. The sooner we can detect faults and eliminate them from having to be considered in each simulation, the faster the simulation will progress. We take the same approach when running a production test and initially order the test vectors by their contribution to fault coverage. This assumes that all faults are equally likely. Test engineers can then modify the test program if they discover vectors late in the test program that are efficient in detecting faulty chips.

14.4.5 Fault-Simulation Results

The output of a fault simulator separates faults into several **fault categories**. If we can detect a fault at a location, it is a **testable fault**. A testable fault must be placed on a **controllable net**, so that we can change the logic level at that location from `'0'` to `'1'` and from `'1'` to `'0'`. A testable fault must also be on an **observable net**, so that we can see the effect of the fault at a PO. This means that **uncontrollable nets** and **unobservable nets** result in faults we cannot detect. We call these faults **untested faults, untestable faults**, or **impossible faults.**

If a PO of the good circuit is the opposite to that of the faulty circuit, we have a detected fault (sometimes called a **hard-detected fault** or a **definitely detected fault**). If the POs of the good circuit and faulty circuit are identical, we have an **undetected fault**. If a PO of the good circuit is a `'1'` or a `'0'` but the corresponding PO of the faulty circuit is an `'X'` (unknown, either `'0'` or `'1'`), we have a **possibly detected fault** (also called a **possible-detected fault**, **potential fault**, or **potentially detected fault**).

If the PO of the good circuit changes between a `'1'` and a `'0'` while the faulty circuit remains at `'X'`, then we have a **soft-detected fault**. Soft-detected faults are a subset of possibly detected faults. Some simulators keep track of these soft-detected faults separately. Soft-detected faults are likely to be detected on a real tester if this sequence occurs often. Most fault simulators allow you to set a **fault-drop threshold** so that the simulator will remove faults from further consideration after soft-detecting or possibly detecting them a specified number of times. This is called **fault dropping** (or **fault discarding**). The more often a fault is possibly detected, the more likely it is to be detected on a real tester.

A **redundant fault** is a fault that makes no difference to the circuit operation. A combinational circuit with no such faults is **irredundant**. There are close links between logic-synthesis algorithms and redundancy. Logic-synthesis algorithms can produce combinational logic that is irredundant and 100 % testable for single stuck-at faults by removing redundant logic as part of logic minimization.

If a fault causes a circuit to oscillate, it is an **oscillatory fault**. Oscillation can occur within feedback loops in combinational circuits with zero-delay models. A fault that affects a larger than normal portion of the circuit is a **hyperactive fault**. Fault simulators have settings to prevent such faults from using excessive amounts of computation time. It is very annoying to run a fault simulation for several days only to discover that the entire time was taken up by simulating a single fault in a RS flip-flop or on the clock net, for example. Figure 14.15 shows some examples of fault categories.

14.4.6 Fault-Simulator Logic Systems

In addition to the way the fault simulator counts faults in various fault categories, the number of detected faults during fault simulation also depends on the logic system used by the fault simulator. As an example, Cadence's VeriFault concurrent fault simulator uses a logic system with the six logic values: `'0'`, `'1'`, `'Z'`, `'L'`, `'H'`, `'X'`. Table 14.8 shows the results of comparing the faulty and the good circuit simulations.

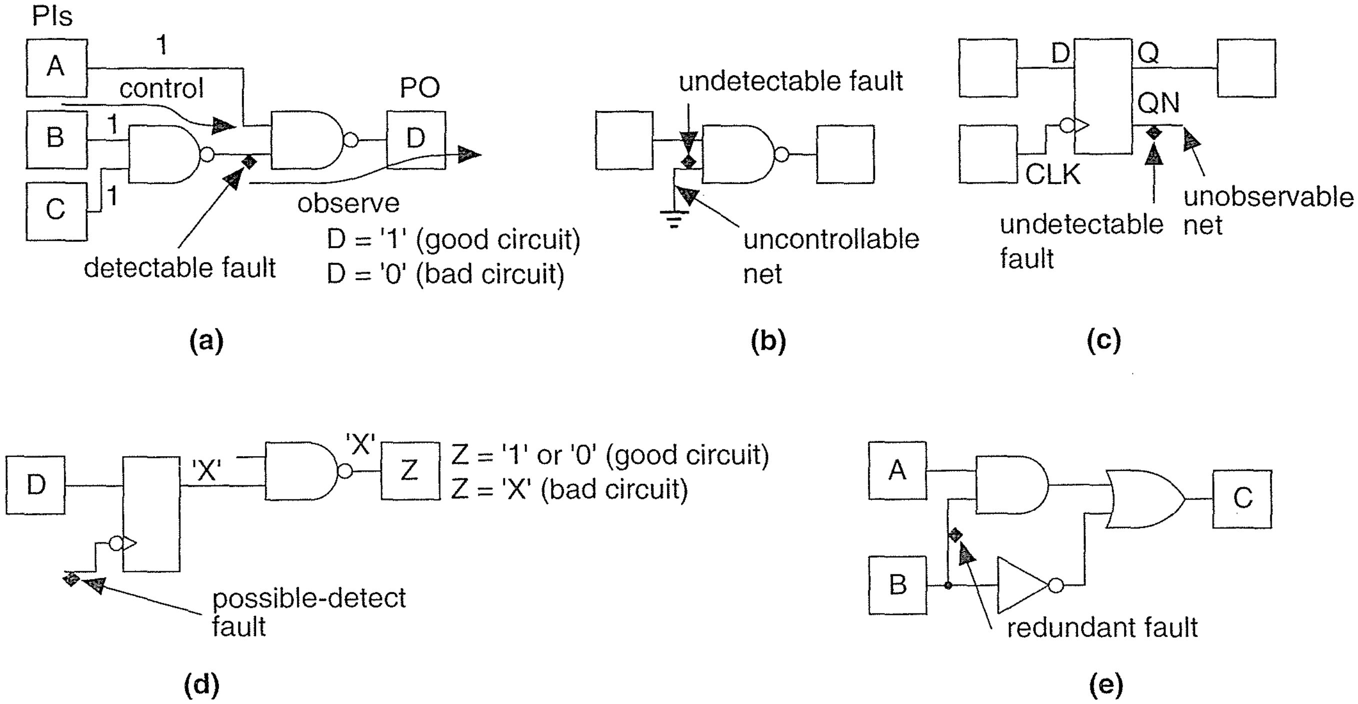


FIGURE 14.15 Fault categories. (a) A detectable fault requires the ability to control and observe the fault origin. (b) A net that is fixed in value is uncontrollable and therefore will produce one undetected fault. (c) Any net that is unconnected is unobservable and will produce undetected faults. (d) A net that produces an unknown 'X' in the faulty circuit and a '1' or a '0' in the good circuit may be detected (depending on whether the 'X' is in fact a '0' or '1'), but we cannot say for sure. At some point this type of fault is likely to produce a discrepancy between good and bad circuits and will eventually be detected. (e) A redundant fault does not affect the operation of the good circuit. In this case the AND gate is redundant since AB + B' = A + B'.

From Table 14.8 we can deduce that, in this logic system:

- Fault detection is possible only if the good circuit and the bad circuit both produce either a '1' or a '0'.
- If the good circuit produces a 'Z' at a three-state output, no faults can be detected (not even a fault on the three-state output).
- If the good circuit produces anything other than a '1' or '0', no faults can be detected.

A fault simulator assigns faults to each of the categories we have described. We define the fault coverage as:

$$\text{fault coverage} = \text{detected faults/detectable faults}. \qquad (14.2)$$

The number of detectable faults excludes any undetectable fault categories (untestable or redundant faults). Thus,

$$\text{detectable faults} = \text{faults} - \text{undetectable faults}, \qquad (14.3)$$

$$\text{undetectable faults} = \text{untested faults} + \text{redundant faults}. \qquad (14.4)$$

The fault simulator may also produce an analysis of **fault grading**. This is a graph, histogram, or tabular listing showing the cumulative fault coverage as a function of the number of test vectors. This information is useful to remove **dead test cycles**, which contain vectors that do not add to fault coverage. If you reinitialize the circuit at regular intervals, you can remove vectors up to an initialization without altering the function of any vectors after the initialization. The list of faults that the simulator inserted is the **fault list.** In addition to the fault list, a **fault dictionary** lists the faults with their corresponding primary outputs (the **faulty output vector**). The set of input vectors and faulty output vectors that uniquely identify a fault is the **fault signature**. This information can be useful to test engineers, allowing them to work backward from production test results and pinpoint the cause of a problem if several ASICs fail on the tester for the same reasons.

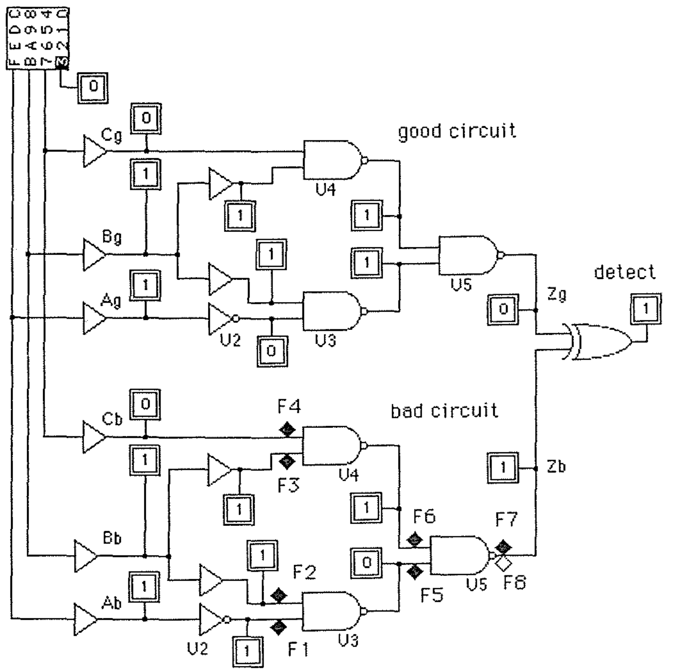
TABLE 14.8 The VeriFault concurrent fault simulator logic system.[1]

		Faulty circuit					
		0	**1**	**Z**	**L**	**H**	**X**
Good circuit	**0**	U	D	P	P	P	P
	1	D	U	P	P	P	P
	Z	U	U	U	U	U	U
	L	U	U	U	U	U	U
	H	U	U	U	U	U	U
	X	U	U	U	U	U	U

[1] L = 0 or Z; H = 1 or Z; Z = high impedance; X = unknown; D = detected; P = potentially detected; U = undetected.

14.4.7 Hardware Acceleration

Simulation engines or **hardware accelerators** use computer architectures that are tuned to fault-simulation algorithms. These special computers allow you to add multiple simulation boards in one chassis. Since each board is essentially a workstation produced in relatively low volume and there are between 2 and 10 boards in one accelerator, these machines are between one and two orders of magnitude more expensive than a workstation. There are two ways to use multiple boards for fault simulation. One method runs good circuits on each board in parallel with the same stimulus and generates faulty circuits concurrently with other boards. The acceleration factor is less than the number of boards because of overhead. This method is usually faster than distributing a good circuit across multiple boards. Some fault simulators allow you to use multiple circuits across multiple machines on a network in **distributed fault simulation**.



Fault	Type	[1]Vectors (hex)	Good output	Bad output
F1	SA1	**3**	0	1
F2	SA1	0, 4	0, 0	1, 1
F3	SA1	4, 5	0, 0	1, 1
F4	SA1	3	0	1
F5	SA1	2	1	0
F6	SA1	7	1	0
F7	SA1	0, 1, 3, 4, 5	0, 0, 0, 0, 0	1, 1, 1, 1, 1
F8	SA0	2, 6, 7	1, 1, 1	0, 0, 0

[1] Test vector format:
3 = 011, so that CBA = 011: C = '0', B = '1', A = '1'

FIGURE 14.16 Fault simulation of A'B + BC. The simulation results for fault F1 (U2 output stuck at 1) with test vector value hex **3** (shown in bold in the table) are shown on the LogicWorks schematic. Notice that the output of U2 is 0 in the good circuit and stuck at 1 in the bad circuit.

14.4.8 A Fault-Simulation Example

Figure 14.16 illustrates fault simulation using the circuit of Figure 14.14. We have used all possible inputs as a test vector set in the following order: `{000, 001, 010, 011, 100, 101, 110, 111}`. There are eight collapsed SSFs in this circuit, F1–F8. Since the good circuit is irredundant, we have 100 percent fault coverage. The following fault-simulation results were derived from a logic simulator rather than a fault simulator, but are presented in the same format as output from an automated test system.

```
Total number of faults: 22
Number of faults in collapsed fault list: 8

Test Vector         Faults detected        Coverage/%        Cumulative/%
-----------         ---------------        ----------        ------------
000                 F2, F7                       25.0              25.0
001                 F7                           12.5              25.0
```

```
010                 F5, F8                     25.0           50.0
011                 F1, F4, F7                 37.5           75.0
100                 F2, F3, F7                 37.5           87.5
101                 F3, F7                     25.0           87.5
110                 F8                         12.5           87.5
111                 F6, F8                     25.0          100.0

Total number of vectors   : 8
                          Noncollapsed          Collapsed
Fault counts:
Detected                            16                  8
Untested                             0                  0
                                ------             ------
Detectable                          16                  8

Redundant                            0                  0
Tied                                 0                  0
FAULT COVERAGE                  100.00 %           100.00 %
```

Fault simulation tells us that we need to apply seven test vectors in order to achieve full fault coverage. The highest-quality test vectors are {011} and {100}. For example, test vector {011} detects three faults (F1, F4, and F7) out of eight. This means if we were to reduce the test set to just {011} the fault coverage would be 3/8, or 37 percent. Proceeding in this fashion we reorder the test vectors in terms of their contribution to cumulative test coverage as follows: {011, 100, 010, 111, 000, 001, 101, 110}. This is a hard problem for large numbers of test vectors because of the interdependencies between the faults detected by the different vectors. Repeating the fault simulation gives the following fault grading:

```
Test Vector         Faults detected        Coverage/%       Cumulative/%
-----------         ---------------        ----------       ------------
011                 F1, F4, F7                 37.5            37.5
100                 F2, F3, F7                 37.5            62.5
010                 F5, F8                     25.0            87.5
111                 F6, F8                     25.0           100.0
000                 F2, F7                     25.0           100.0
001                 F7                         12.5           100.0
101                 F3, F7                     25.0           100.0
110                 F8                         12.5           100.0
```

Now, instead of using seven test vectors, we need only apply the first four vectors from this set to achieve 100 percent fault coverage, cutting the expensive production test time nearly in half. Reducing the number of test vectors in this fashion is called **test-vector compression** or **test-vector compaction**.

The fault signatures for faults F1–F8 for the last test sequence, {011, 100, 010, 111, 000, 001, 101, 110}, are as follows:

```
#  fail     good     bad
-- -------- -------- --------
F1 10000000 00110001 10110001
F2 01001000 00110001 01111001
F3 01000010 00110001 01110011
F4 10000000 00110001 10110001
F5 00100000 00110001 00010001
F6 00010000 00110001 00100001
F7 11001110 00110001 11111111
F8 00110001 00110001 00000000
```

The first pattern for each fault indicates which test vectors will fail on the tester (we say a test vector fails when it successfully detects a faulty circuit during a production test). Thus, for fault F1, pattern '10000000' indicates that only the first test vector will fail if fault F1 is present. The second and third patterns for each fault are the POs of the good and bad circuits for each test vector. Since we only have one PO in our simple example, these patterns do not help further distinguish between faults. Notice, that as far as an external view is concerned, faults F1 and F4 have identical fault signatures and are therefore indistinguishable. Faults F1 and F4 are said to be **structurally equivalent**. In general, we cannot detect structural equivalence by looking at the circuit. If we apply only the first four test vectors, then faults F2 and F3 also have identical fault signatures. Fault signatures are only useful in diagnosing fault locations if we have one, or a very few faults.

Not all fault simulators give all the information we have described. Most fault simulators drop hard-detected faults from consideration once they are detected to increase the speed of simulation. With dropped hard-detected faults we cannot independently grade each vector and we cannot construct a fault dictionary. This is the reason we used a logic simulator to generate the preceding results.

14.4.9 Fault Simulation in an ASIC Design Flow

At the beginning of this section we dodged the issue of test-vector generation. It is possible to automatically generate test vectors and test programs (with certain restrictions), and we shall discuss these methods in Section 14.5. A by-product of some of these automated systems is a measure of fault coverage. However, fault simulation is still used for the following reasons:

- Test-generation software is expensive, and many designers still create test programs manually and then grade the test vectors using fault simulation.
- Automatic test programs are not yet at the stage where fault simulation can be completely omitted in an ASIC design flow. Usually we need fault simulation to add some vectors to test logic not covered automatically, to check that

test logic has been inserted correctly, or to understand and correct fault coverage problems.

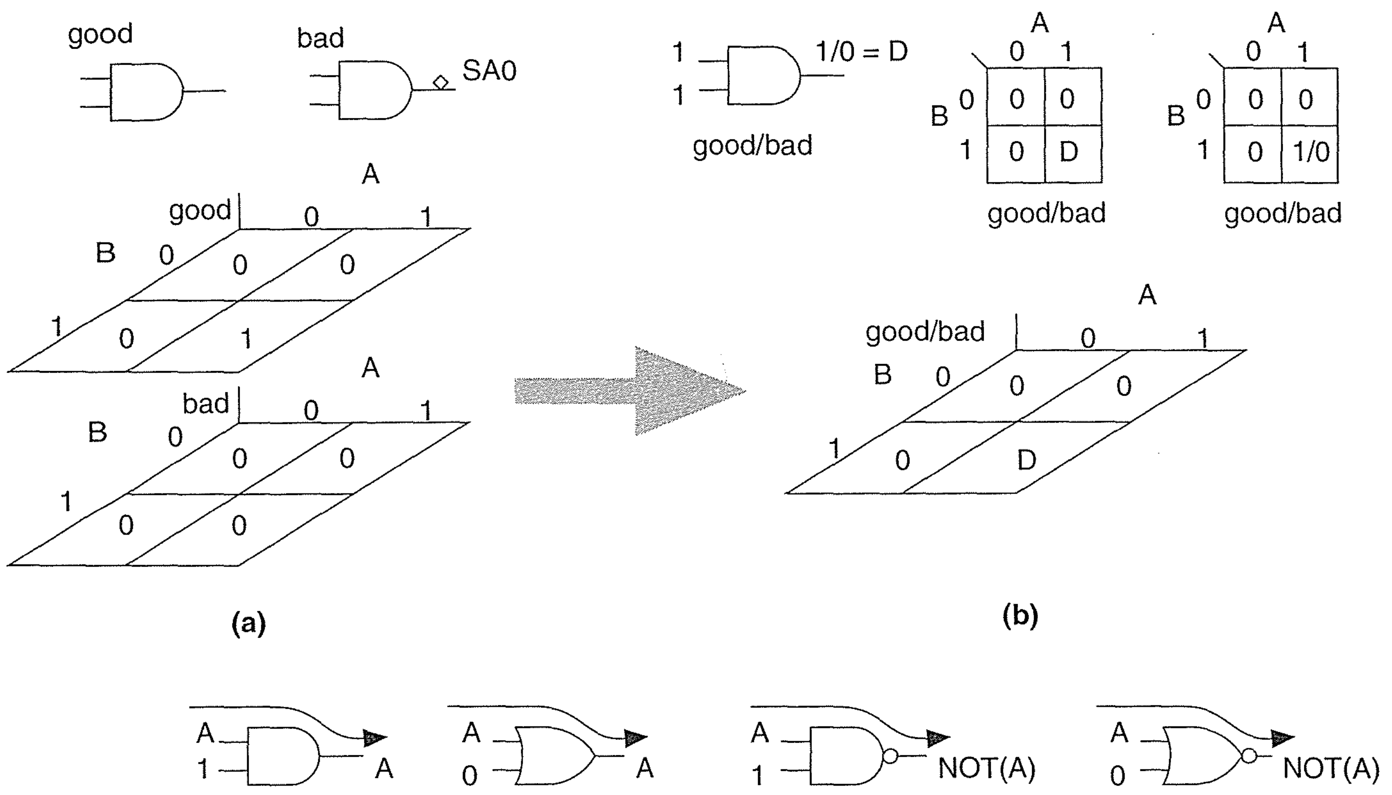
- It is far too expensive to use a production tester to debug a production test. One use of a fault simulator is to perform this function off line.
- The reuse and automatic generation of large cells is essential to decrease the complexity of large ASIC designs. Megacells and embedded blocks (an embedded microcontroller, for example) are normally provided with **canned test vectors** that have already been fault simulated and fault graded. The megacell has to be isolated during test to apply these vectors and measure the response. Cell compilers for RAM, ROM, multipliers, and other regular structures may also generate test vectors. Fault simulation is one way to check that the various embedded blocks and their vectors have been correctly glued together with the rest of the ASIC to produce a complete set of test vectors and a test program.
- Production testers are very expensive. There is a trend away from the use of test vectors to include more of the test function on an ASIC. Some internal test logic structures generate test vectors in a random or pseudorandom fashion. For these structures there is no known way to generate the fault coverage. For these types of test structures we will need some type of fault simulation to measure fault coverage and estimate defect levels.

14.5 Automatic Test-Pattern Generation

In this section we shall describe a widely used algorithm, PODEM, for **automatic test-pattern generation** (**ATPG**) or **automatic test-vector generation** (**ATVG**). Before we can explain the PODEM algorithm we need to develop a shorthand notation and explain some terms and definitions using a simpler ATPG algorithm.

14.5.1 The D-Calculus

Figure 14.17(a) and (b) shows a shorthand notation, the **D-calculus**, for tracing faults. The D-calculus was developed by Roth [1966] together with an ATPG algorithm, the **D-algorithm**. The symbol D (for detect) indicates the value of a node is a logic '0' in the good circuit and a logic '1' in the bad circuit. We can also write this as $D = 0/1$. In general we write *g*/*b*, a **composite logic value**, to indicate a node value in the good circuit is *g* and *b* in the bad circuit (by convention we always write the good circuit value first and the faulty circuit value second). The complement of D is $\overline{D} = 1/0$ ($\overline{D}$ is rarely written as D' since $\overline{D}$ is a logic value just like '1' and '0'). Notice that $\overline{D}$ does not mean *not* detected, but simply that we see a '0' in the good circuit and a '1' in the bad circuit. We can apply Boolean algebra to the composite logic values D and $\overline{D}$ as shown in Figure 14.17(c). The composite values 1/1 and 0/0 are equivalent to '1' and '0' respectively. We use the unknown logic value 'X' to



AND	0	1	D	D̄	X
0	0	0	0	0	0
1	0	1	0	D̄	X
D	0	D	0	0	X
D̄	0	D̄	0	D̄	X
X	0	X	X	X	X

OR	0	1	D	D̄	X
0	0	1	D	D̄	X
1	1	1	1	1	1
D	D	1	D	1	X
D̄	D̄	1	1	D̄	X
X	X	1	X	X	X

NAND	0	1	D	D̄	X
0	1	1	1	1	1
1	1	0	1	D	X
D	1	D̄	1	1	X
D̄	1	D	1	D	X
X	1	X	X	X	X

NOR	0	1	D	D̄	X
0	1	0	D̄	D	X
1	0	0	0	0	0
D	D̄	0	D̄	0	X
D̄	D	0	0	D	X
X	X	0	X	X	X

FIGURE 14.17 The D-calculus. (a) We need a way to represent the behavior of the good circuit and the bad circuit at the same time. (b) The composite logic value D (for detect) represents a logic '1' in the good circuit and a logic '0' in the bad circuit. We can also write this as D = 1/0. (c) The logic behavior of simple logic cells using the D-calculus. Composite logic values can propagate through simple logic gates if the other inputs are set to their enabling values.

represent a logic value that is one of '0', '1', D, or D̄, but we do not know or care which.

If we wish to **propagate** a signal from one or more inputs of a logic cell to the logic cell output, we set the remaining inputs of that logic cell to what we call the **enabling value**. The enabling value is '1' for AND and NAND gates and '0' for OR and NOR gates. Figure 14.17(c) illustrates the use of enabling values. In contrast, setting at least one input of a logic gate to the **controlling value**, the opposite of the enabling value for that gate, forces or **justifies** the output node of that logic gate to a fixed value. The controlling value of '0' for an AND gate justifies the output to '0' and for a NAND gate justifies the output to '1'. The controlling. values of '1' justifies the output of an OR gate to '1' and justifies the output of a NOR gate to '0'. To find controlling and enabling values for more complex logic cells, such as AOI and OAI logic cells, we can use their simpler AND, OR, NAND, and NOR gate representations.

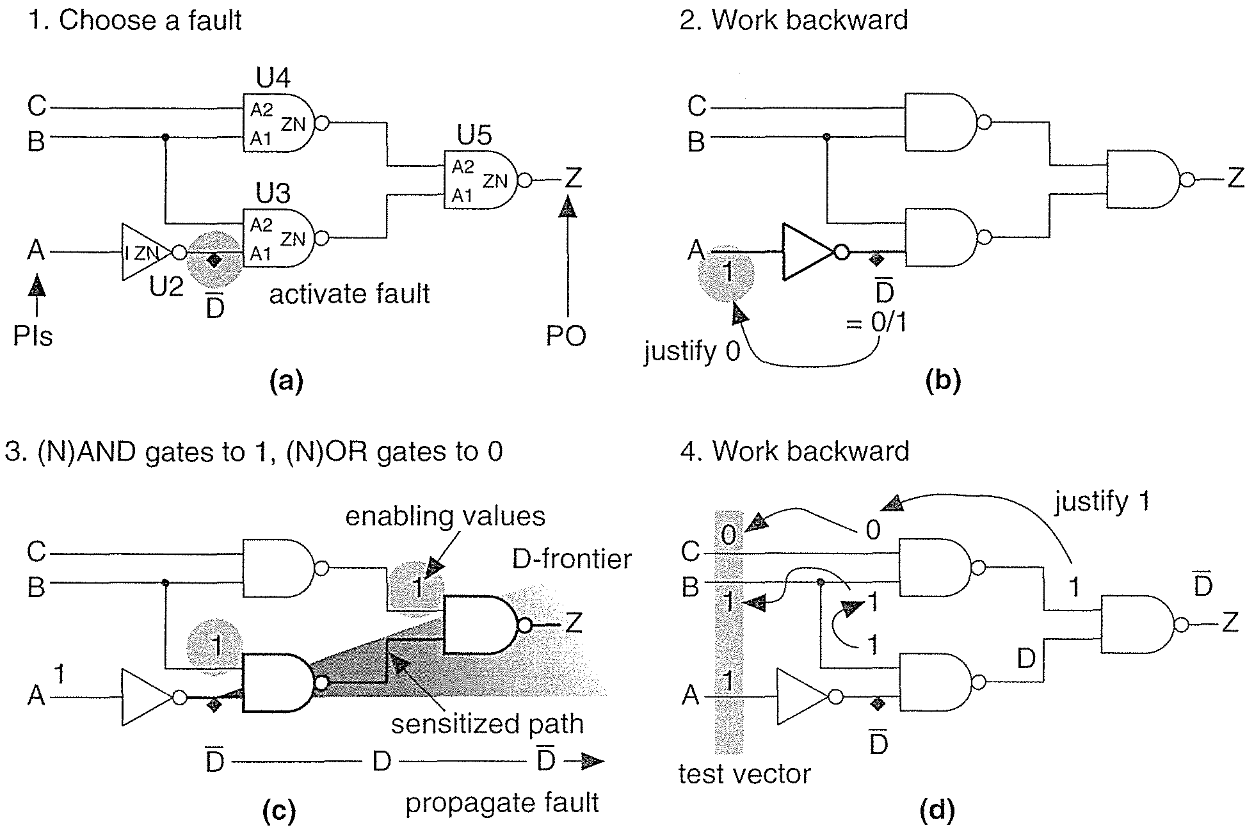


FIGURE 14.18 A basic ATPG (automatic test-pattern generation) algorithm for A'B + BC. (a) We activate a fault, U2.ZN stuck at 1, by setting the pin or node to '0', the opposite value of the fault. (b) We work backward from the fault origin to the PIs (primary inputs) by recursively justifying signals at the output of logic cells. (c) We then work forward from the fault origin to a PO (primary output), setting inputs to gates on a sensitized path to their enabling values. We propagate the fault until the D-frontier reaches a PO. (d) We then work backward from the PO to the PIs recursively justifying outputs to generate the sensitized path. This simple algorithm always works, providing signals do not branch out and then rejoin again.

14.5.2 A Basic ATPG Algorithm

A basic algorithm to generate test vectors automatically is shown in Figure 14.18. We detect a fault by first **activating** (or **exciting** the fault). To do this we must drive the faulty node to the opposite value of the fault. Figure 14.18(a) shows a stuck-at-1 fault at the output pin, ZN, of the inverter U2 (we call this fault U2.ZN.SA1). To create a test for U2.ZN.SA1 we have to find the values of the PIs that will justify node U2.ZN to '0'. We work backward from node U2.ZN justifying each logic gate output until we reach a PI. In this case we only have to justify U2.ZN to '0', and this is easily done by setting the PI A = '0'. Next we work forward from the fault origin and **sensitize** a path to a PO (there is only one PO in this example). This propagates the fault effect to the PO so that it may be **observed**. To propagate the fault effect to the PO Z, we set U3.A2 = '1' and then U5.A2 = '1'.

We can visualize fault propagation by supposing that we set all nodes in a circuit to unknown, 'X'. Then, as we successively propagate the fault effect toward the POs, we can imagine a wave of D's and $\overline{D}$'s, called the **D-frontier**, that propagates from the fault origin toward the POs. As a value of D or $\overline{D}$ reaches the inputs of a logic cell whose other inputs are 'X', we add that logic cell to the D-frontier. Then we find values for the other inputs to propagate the D-frontier through the logic cell to continue the process.

This basic algorithm of justifying and then propagating a fault works when we can justify nodes without interference from other nodes. This algorithm breaks down when we have **reconvergent fanout**. Figure 14.19(a) shows another example of justifying and propagating a fault in a circuit with reconvergent fanout. For direct comparison Figure 14.19(b) shows an irredundant circuit, similar to part (a), except the fault signal, B stuck at 1, branches and then reconverges at the inputs to gate U5. The reconvergent fanout in this new circuit breaks our basic algorithm. We now have two sensitized paths that propagate the fault effect to U5. These paths combine to produce a constant '1' at Z, the PO. We have a **multipath sensitization** problem.

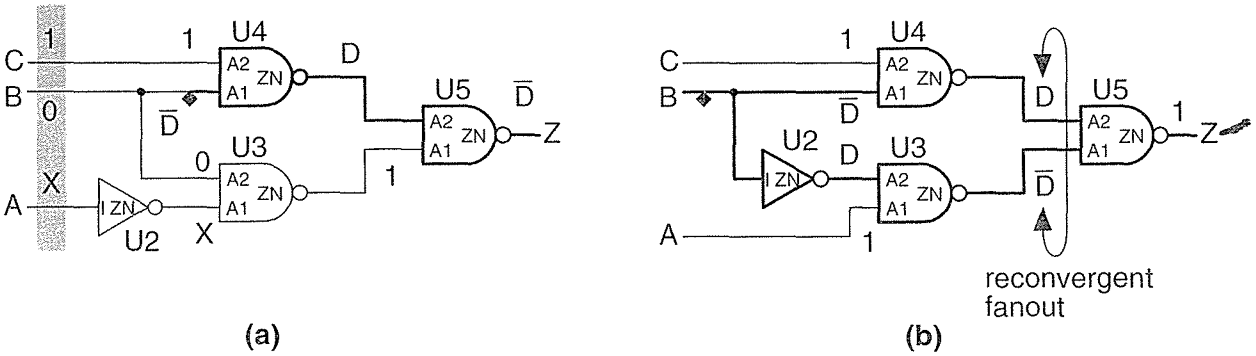


FIGURE 14.19 Reconvergent fanout. (a) Signal B branches and then reconverges at logic gate U5, but the fault U4.A1 stuck at 1 can still be excited and a path sensitized using the basic algorithm of Figure 14.18. (b) Fault B stuck at 1 branches and then reconverges at gate U5. When we enable the inputs to both gates U3 and U4 we create two sensitized paths that prevent the fault from propagating to the PO (primary output). We can solve this problem by changing A to '0', but this breaks the rules of the algorithm illustrated in Figure 14.18. The PODEM algorithm solves this problem.

14.5.3 The PODEM Algorithm

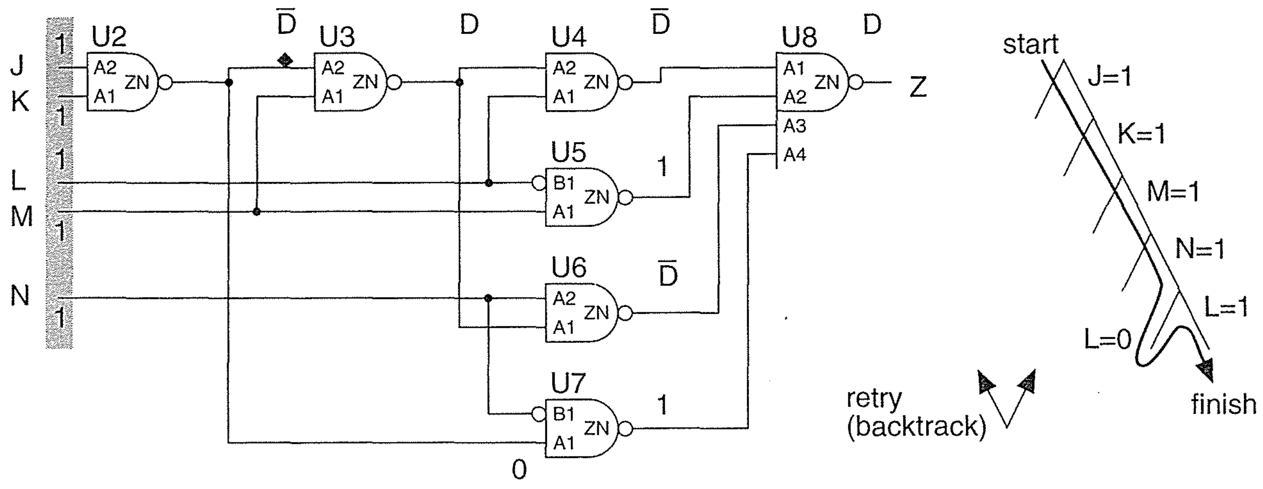
The **path-oriented decision making** (**PODEM**) algorithm solves the problem of reconvergent fanout and allows multipath sensitization [Goel, 1981]. The method is similar to the basic algorithm we have already described except PODEM will retry a step, reversing an incorrect decision. There are four basic steps that we label: objective, backtrace, implication, and D-frontier. These steps are as follows:

1. Pick an *objective* to set a node to a value. Start with the fault origin as an objective and all other nodes set to 'X'.
2. *Backtrace* to a PI and set it to a value that will help meet the objective.
3. Simulate the network to calculate the effect of fixing the value of the PI (this step is called *implication*). If there is no possibility of sensitizing a path to a PO, then retry by reversing the value of the PI that was set in step 2 and simulate again.
4. Update the *D-frontier* and return to step 1. Stop if the D-frontier reaches a PO.

Figure 14.20 shows an example that uses the following iterations of the four steps in the PODEM algorithm:

1. We start with activation of the fault as our objective, U3.A2 = '0'. We backtrace to J. We set J = '1'. Since K is still 'X', implication gives us no further information. We have no D-frontier to update.
2. The objective is unchanged, but this time we backtrace to K. We set K = '1'. Implication gives us U2.ZN = '1' (since now J = '1' and K = '1') and therefore U7.ZN = '1'. We still have no D-frontier to update.
3. We set U3.A1 = '1' as our objective in order to propagate the fault through U3. We backtrace to M. We set M = '1'. Implication gives us U2.ZN = '1' and U3.ZN = D. We update the D-frontier to reflect that U4.A2 = D and U6.A1 = D, so the D-frontier is U4 and U6.
4. We pick U6.A2 = '1' as an objective in order to propagate the fault through U6. We backtrace to N. We set N = '1'. Implication gives us U6.ZN = $\overline{\mathrm{D}}$. We update the D-frontier to reflect that U4.A2 = D and U8.A1 = $\overline{\mathrm{D}}$, so the D-frontier is U4 and U8.
5. We pick U8.A1 = '1' as an objective in order to propagate the fault through U8. We backtrace to L. We set L = '0'. Implication gives us U5.ZN = '0' and therefore U8.ZN = '0' (this node is Z, the PO). There is then no possible sensitized path to the PO Z. We must have made an incorrect decision, we retry and set L = '1'. Implication now gives us U8.ZN = D and we have propagated the D-frontier to a PO.

We can see that the PODEM algorithm proceeds in two phases. In the first phase, iterations 1 and 2 in Figure 14.20, the objective is fixed in order to activate the fault. In the second phase, iterations 3–5, the objective changes in order to prop-



Iteration	Objective	Backtrace[1]	Implication	D-frontier
1	U3.A2 = 0	J = 1		
2	U3.A2 = 0	K = 1	U7.ZN = 1	
3	U3.A1 = 1	M = 1	U3.ZN = D	U4, U6
4	U6.A2 = 1	N = 1	U6.ZN = $\overline{D}$	U4, U8
5a	U8.A1 = 1	L = 0	U8.ZN = 1	U4, U8
5b	Retry	L = 1	U8.ZN = D	A

[1] Backtrace is not the same as retry or backtrack.

FIGURE 14.20 The PODEM (path-oriented decision making) algorithm.

agate the fault. In step 3 of the PODEM algorithm there must be at least one path containing unknown values between the gates of the D-frontier and a PO in order to be able to complete a sensitized path to a PO. This is called the **X-path check**.

You may wonder why there has been no explanation of the backtrace mechanism or how to decide a value for a PI in step 2 of the PODEM algorithm. The decision tree shown in Figure 14.20 shows that it does not matter. PODEM conducts an implicit binary search over all the PIs. If we make an incorrect decision and assign the wrong value to a PI at some step, we will simply need to retry that step. Texts, programs, and articles use the term *backtrace* as we have described it, but then most use the term **backtrack** to describe what we have called a retry, which can be confusing. I also did not explain how to choose the objective in step 1 of the PODEM algorithm. The initial objective is to activate the fault. Subsequently we select a logic gate from the D-frontier and set one of its inputs to the enabling value in an attempt to propagate the fault.

We can use intelligent procedures, based on *controllability* and *observability*, to guide PODEM and reduce the number of incorrect decisions. PODEM is a development of the D-algorithm, and there are several other ATPG algorithms that are developments of PODEM. One of these is **FAN (fanout-oriented test generation)** that removes the need to backtrace all the way to a PI, reducing the search time [Fujiwara and Shimono, 1983; Schulz, Trischler, and Sarfert, 1988]. Algorithms based on the D-algorithm, PODEM, and FAN are the basis of many commercial ATPG systems.

14.5.4 Controllability and Observability

In order for an ATPG system to provide a test for a fault on a node it must be possible to both control and observe the behavior of the node. There are both theoretical and practical issues involved in making sure that a design does not contain buried circuits that are impossible to observe and control. A software program that measures the **controllability** (with three *l*'s) and **observability** of nodes in a circuit is useful in conjunction with ATPG software.

There are several different measures for controllability and observability [Butler and Mercer, 1992]. We shall describe one of the first such systems called **SCOAP (Sandia Controllability/Observability Analysis Program)** [Goldstein, 1979]. These measures are also used by ATPG algorithms.

Combinational controllability is defined separately from **sequential controllability**. We also separate **zero-controllability** and **one-controllability**. For example, the **combinational zero-controllability** for a two-input AND gate, $Y = \text{AND}(X_1, X_2)$, is recursively defined in terms of the input controllability values as follows:

$$\text{CC0}(Y) = \min\{\text{CC0}(X_1), \text{CC0}(X_2)\} + 1. \qquad (14.5)$$

We choose the minimum value of the two-input controllability values to reflect the fact that we can justify the output of an AND gate to '0' by setting any input to the control value of '0'. We then add one to this value to reflect the fact that we have passed through an additional level of logic. Incrementing the controllability measures for each level of logic represents a measure of the **logic distance** between two nodes.

We define the **combinational one-controllability** for a two-input AND gate as

$$\text{CC1}(Y) = \text{CC1}(X_1) + \text{CC1}(X_2) + 1. \qquad (14.6)$$

This equation reflects the fact that we need to set all inputs of an AND gate to the enabling value of '1' to justify a '1' at the output. Figure 14.21(a) illustrates these definitions.

An inverter, $Y = \text{NOT}(X)$, reverses the controllability values:

$$\text{CC1}(Y) = \text{CC0}(X) + 1 \quad \text{and} \quad \text{CC0}(Y) = \text{CC1}(X) + 1. \qquad (14.7)$$

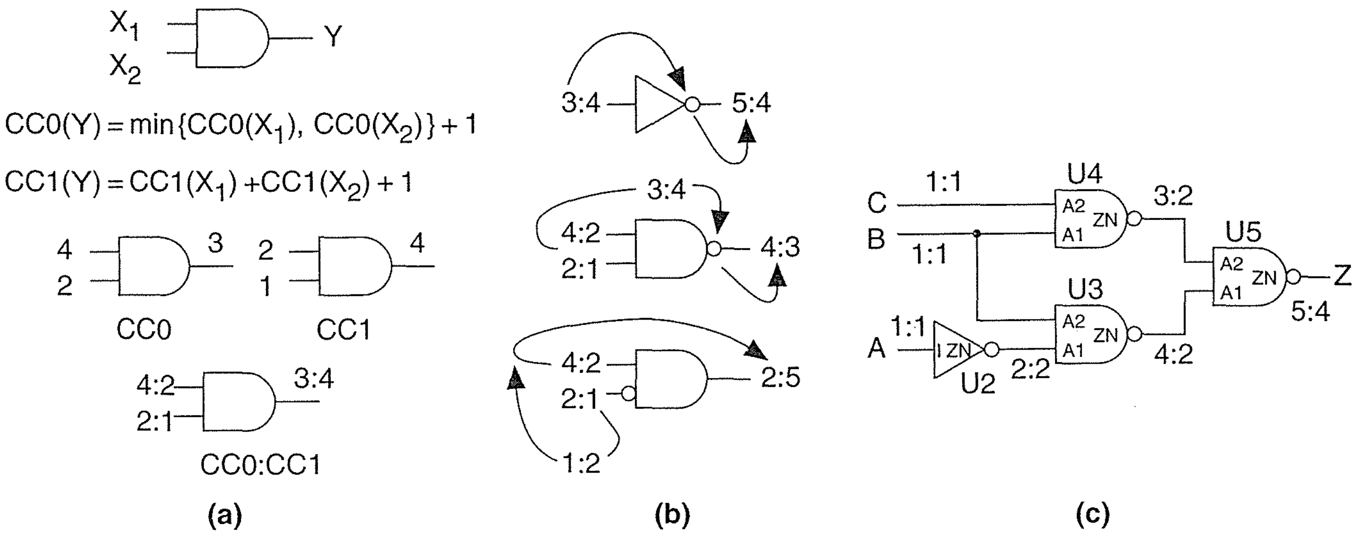


FIGURE 14.21 Controllability measures. (a) Definition of combinational zero-controllability, CC0, and combinational one-controllability, CC1, for a two-input AND gate. (b) Examples of controllability calculations for simple gates, showing intermediate steps. (c) Controllability in a combinational circuit.

Since we can construct all other logic cells from combinations of two-input AND gates and inverters we can use Eqs. 14.5–14.7 to derive their controllability equations. When we do this we only increment the controllability by one for each primitive gate. Thus for a three-input NAND with an inverting input, $Y = \mathrm{NAND}(X_1, X_2, \mathrm{NOT}(X_3))$:

$$\mathrm{CC0}(Y) = \mathrm{CC1}(X_1) + \mathrm{CC1}(X_2) + \mathrm{CC0}(X_3) + 1,$$

$$\mathrm{CC1}(Y) = \min\{\ \mathrm{CC0}(X_1), \mathrm{CC0}(X_2), \mathrm{CC1}(X_3)\ \} + 1. \qquad (14.8)$$

For a two-input NOR, $Y = \mathrm{NOR}(X_1, X_2) = \mathrm{NOT}(\mathrm{AND}(\mathrm{NOT}(X_1), \mathrm{NOT}(X_2))$:

$$\mathrm{CC1}(Y) = \min\{\ \mathrm{CC1}(X_1), \mathrm{CC1}(X_2)\ \} + 1,$$

$$\mathrm{CC0}(Y) = \mathrm{CC0}(X_1) + \mathrm{CC0}(X_2) + 1. \qquad (14.9)$$

Figure 14.21(b) shows examples of controllability calculations. A bubble on a logic gate at the input or output swaps the values of CC1 and CC0. Figure 14.21(c) shows how controllability values for a combinational circuit are calculated by working forward from each PI that is defined to have a controllability of one.

We define observability in terms of the controllability measures. The **combinational observability**, OC (X_1), of input X_1 of a two-input AND gate can be expressed in terms of the controllability of the other input CC1 (X_2) and the combinational observability of the output, OC (Y):

$$\mathrm{OC}\,(X_1) = \mathrm{CC1}\,(X_2) + \mathrm{OC}\,(Y) + 1. \qquad (14.10)$$

If a node X_1 branches (has fanout) to nodes X_2 and X_3 we choose the most observable of the branches:

$$\mathrm{OC}\,(X_1) = \min\,\{\ \mathrm{O}\,(X_2) + \mathrm{O}\,(X_3)\ \}. \qquad (14.11)$$

Figure 14.22(a) and (b) show the definitions of observability. Figure 14.22(c) illustrates calculation of observability at a three-input NAND; notice we sum the CC1 values for the other inputs (since the enabling value for a NAND gate is one, the same as for an AND gate). Figure 14.22(d) shows the calculation of observability working back from the PO which, by definition, has an observability of zero.

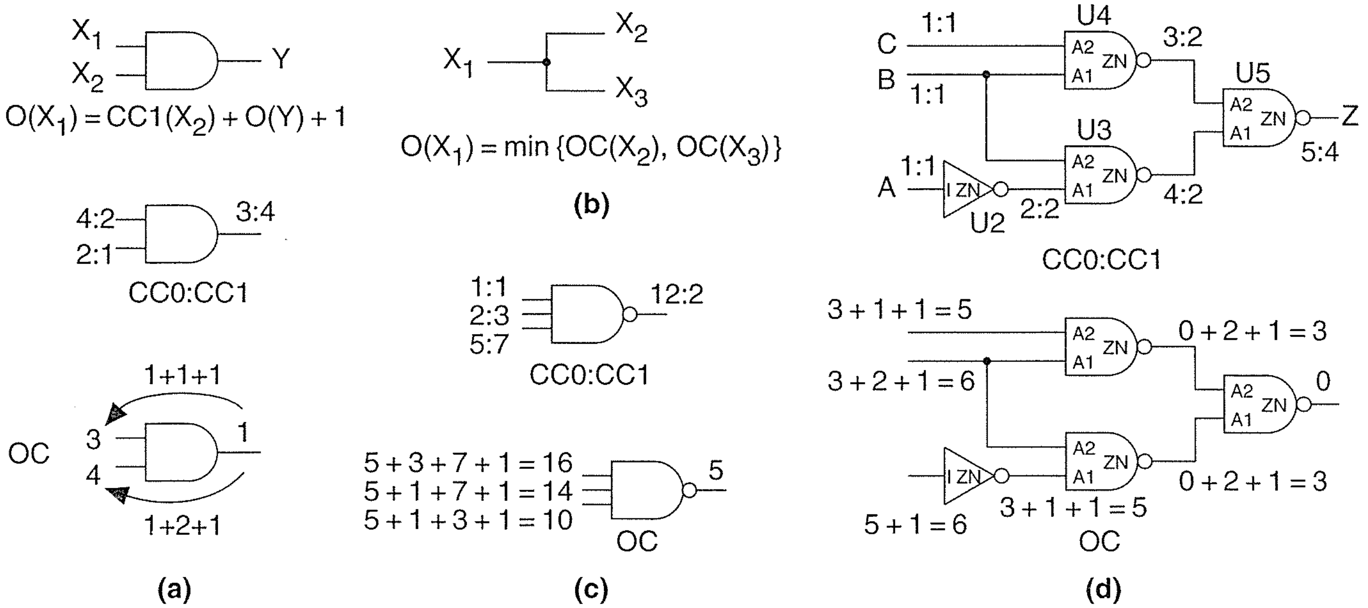


FIGURE 14.22 Observability measures. (a) The combinational observability, OC(X_1), of an input, X_1, to a two-input AND gate defined in terms of the controllability of the other input and the observability of the output. (b) The observability of a fanout node is equal to the observability of the most observable branch. (c) Example of an observability calculation at a three-input NAND gate. (d) The observability of a combinational network can be calculated from the controllability measures, CC0:CC1. The observability of a PO (primary output) is defined to be zero.

Sequential controllability and observability can be measured using similar equations to the combinational measures except that in the sequential measures (SC1, SC0, and OS) we measure logic distance in terms of the layers of sequential logic, not the layers of combinational logic.

14.6 Scan Test

Sequential logic poses a very difficult ATPG problem. Consider the example of a 32-bit counter with a final carry. If the designer included a reset, we have to clock the counter 2^{32} (approximately 4×10^9) times to check the carry logic. Using a 1 MHz tester clock this requires 4×10^3 seconds, 1 hour, or (at approximately $0.25 per second) $1,000 of tester time. Consider a 16-bit state machine implemented using a one-hot state register with 16 D flip-flops. If the designer did not include a reset we have a very complicated initialization problem. A sequential ATPG algorithm must consider over 2000 states when constructing sequential test vectors. In an ad hoc approach to testing we could construct special reset circuits or create manual test vectors to deal with these special situations, one at a time, as they arise. Instead we can take a **structured test** approach (also called **design for test**, though this term covers a wider field).

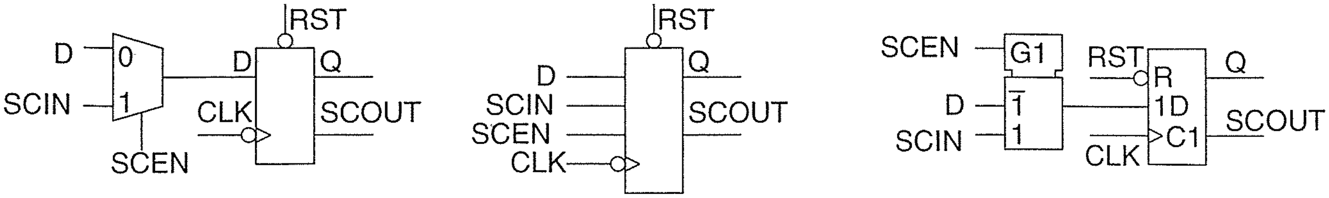
We can automatically generate test vectors for combinational logic, but ATPG is much harder for sequential logic. Therefore the most common sequential structured test approach converts sequential logic to combinational logic. In full-scan design we replace every sequential element with a scan flip-flop. The result is an internal form of boundary scan and, if we wish, we can use the IEEE 1149.1 TAP to access (and the boundary-scan controller to control) an internal-scan chain.

Table 14.9 shows a VHDL model and schematic symbols for a scan flip-flop. There is an area and performance penalty to pay for scan design. The scan MUX adds the delay of a 2:1 MUX to the setup time of the flip-flop; this will directly subtract from the critical path delay. The 2:1 MUX and any separate driver for the scan output also adds approximately 10 percent to the area of the flip-flop (depending on the features present in the original flip-flop). The scan chain must also be routed, and this complicates physical design and adds to the interconnect area. In ASIC design the benefits of eliminating complex sequential ATPG and the addition of observability and controllability usually outweigh these disadvantages.

The highly structured nature of full scan allows test software (usually called a **test compiler**) to perform automatic **scan insertion**. Using scan design we turn the output of each flip-flop into a **pseudoprimary input** and the input to each flip-flop into a **pseudoprimary output**. ATPG software can then generate test vectors for the combinational logic between scan flip-flops.

There are other approaches to scan design. In **partial scan** we replace a subset of the sequential elements with scan flip-flops. We can choose this subset using heuristic procedures to allow the remaining sequential logic to be tested using sequen-

TABLE 14.9 Scan flip-flop.



```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity  DFFSCAN is
generic (reset_value : STD_LOGIC := '0');
port (     Q : out STD_LOGIC ; D, CLK, RST : in STD_LOGIC;
           SCOUT : out STD_LOGIC; SCIN, SCEN : in  STD_LOGIC );
end DFFSCAN;

architecture behave of DFFSCAN is
signal RST_IN, CLK_IN , SCEN_IN , SCIN_IN, D_IN : STD_LOGIC ;
begin
RST_IN <= to_X01(RST); CLK_IN <= to_X01(CLK);
SCEN_IN <= to_X01(SCEN); SCIN_IN <= to_X01(SCIN); D_IN <= to_X01(D);
DFSCAN : process (CLK_IN, RST_IN) begin
  if RST_IN = '0' then Q <= reset_value; SCOUT <= reset_value;
  elsif RST_IN = '1' and rising_edge (CLK_IN) then
    if SCEN_IN = '1' then Q <= SCIN_IN; SCOUT <= SCIN_IN;
    end if;
    elsif SCEN_IN = '0' then Q <= D_IN; SCOUT <= D_IN;
    else Q <= 'X' ; SCOUT <= 'X';
    end if;
  elsif RST_IN = 'X' or CLK_IN = 'X' or SCEN_IN = 'X' then Q <= 'X'; SCOUT <= 'X';
  end if;
end process DFSCAN;
end behave;
```

tial ATPG techniques. In **destructive scan** we remove the values at the outputs of the flip-flops during the scan process (this is the usual form of scan design). In **nondestructive scan** we keep the flip-flop outputs intact so that we can shift out the scan chain and then resume where we left off. **Level-sensitive scan design (LSSD)** is a form of scan design developed at IBM that uses separate clock phases to drive scan elements.

We shall describe scan design, automated scan insertion, and test-program generation with several examples. First, though, we describe another important structured-test technique.

14.7 Built-in Self-test

The trend to include more test logic on an ASIC has already been mentioned. **Built-in self-test** (**BIST**) is a set of structured-test techniques for combinational and sequential logic, memories, multipliers, and other embedded logic blocks. In each case the principle is to generate test vectors, apply them to the **circuit under test** (**CUT**) or **device under test** (**DUT**), and then check the response.

14.7.1 LFSR

Figure 14.23 shows a **linear feedback shift register** (**LFSR**). The exclusive-OR gates and shift register act to produce a **pseudorandom binary sequence** (**PRBS**) at each of the flip-flop outputs. By correctly choosing the points at which we take the feedback from an n-bit shift register (see Section 14.7.5), we can produce a PRBS of length $2^n - 1$, a **maximal-length sequence** that includes all possible patterns (or vectors) of n bits, excluding the all-zeros pattern.

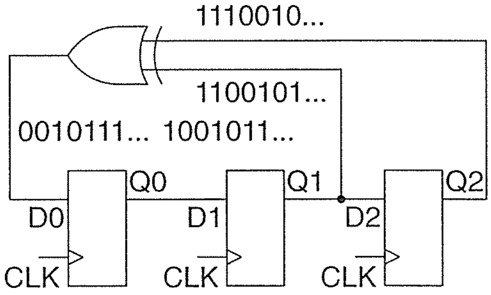


FIGURE 14.23 A linear feedback shift register (LFSR). A 3-bit maximal-length LFSR produces a repeating string of seven pseudorandom binary numbers: 7, 3, 1, 4, 2, 5, 6.

Table 14.10 shows the maximal-length sequence, with length $2^3 - 1 = 7$, for the 3-bit LFSR shown in Figure 14.23. Notice that the first (clock tick 1) and last rows (clock tick 8) are identical. Rows following the seventh row repeat rows 1–7, so that the length of this 3-bit LFSR sequence is $7 = 2^3 - 1$, the maximal length. The shaded regions show how bits are shifted from one clock cycle to the next. We assume the register is initialized to the all-ones state, but any initial state will work and produce the same PRBS, as long as the initial state is not all zeros (in which case the LFSR will stay stuck at all zeros).

14.7.2 Signature Analysis

Figure 14.24 shows the LFSR of Figure 14.23 with an additional XOR gate used in the first stage of the shift register. If we apply a binary input sequence to IN, the shift register will perform **data compaction** (or **compression**) on the input sequence. At the end of the input sequence the shift-register contents, Q0Q1Q2, will form a pattern that we call a **signature**. If the input sequence and the **serial-input**

TABLE 14.10 LFSR example of Figure 14.23.

Clock tick, t=	$Q0_{t+1} = Q1_t \oplus Q2_t$	$Q1_{t+1} = Q0_t$	$Q2_{t+1} = Q1_t$	Q0Q1Q2
1	1	1	1	7
2	0	1	1	3
3	0	0	1	1
4	1	0	0	4
5	0	1	0	2
6	1	0	1	5
7	1	1	0	6
8	1	1	1	7

signature register (**SISR**) are long enough, it is unlikely (though possible) that two different input sequences will produce the same signature. If the input sequence comes from logic that we wish to test, a fault in the logic will cause the input sequence to change. This causes the signature to change from a known good value and we shall then know that the circuit under test is bad. This technique, called **signature analysis**, was developed by Hewlett-Packard to test equipment in the field in the late 1970s.

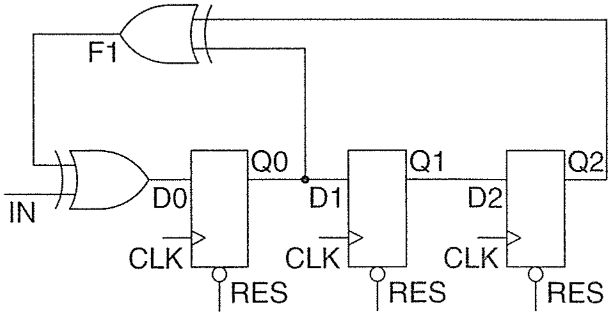


FIGURE 14.24 A 3-bit serial-input signature register (SISR) using an LFSR (linear feedback shift register). The LFSR is initialized to Q1Q2Q3 = '000' using the common RES (reset) signal. The signature, Q1Q2Q3, is formed from shift-and-add operations on the sequence of input bits (IN).

14.7.3 A Simple BIST Example

We can combine the PRBS generator of Figure 14.23 together with the signature register of Figure 14.24 to form the simple BIST structure shown in Figure 14.25(a). LFSR1 generates a maximal-length ($2^3 - 1 = 7$ cycles) PRBS. LFSR2 computes the signature ('011' for the good circuit) of the CUT. LFSR1 is initialized to '100' (Q0 = 1, Q1 = 0, Q2 = 0) and LFSR2 is initialized to '000'. The schematic in Figure 14.25(a) shows the bit sequences in the circuit, both for a good circuit and for a bad circuit with a stuck-at-1 fault, F1. Figure 14.25(b) shows how the bit sequences are calculated in the good circuit. The signature is formed as R0R1R2 seven clock edges (on the eighth clock cycle) after the active-low reset is taken high. Figure 14.26 shows the waveforms in the good and bad circuit. The bad circuit signature, '000', differs from the good circuit and the signature can either be compared with the known good signature on-chip or the signature may be shifted out and compared off-chip (both approaches are used in practice).

14.7.4 Aliasing

In Figure 14.26 the good and bad circuits produced different signatures. There is a small probability that the signature of a bad circuit will be the same as a good circuit. This problem is known as **aliasing** or **error masking**. For the example in Figure 14.25, the bit stream input to the signature analysis register is 7 bits long. There are 2^7 or 128 possible 7-bit-long bit-stream patterns. We assume that each of these 128 bit-stream patterns is equally likely to produce any of the eight (all-zeros is an allowed pattern in a signature register) possible 3-bit signatures. It turns out that this is a good assumption. Thus there are 128 / 8 or 16 bit-streams that produce the good signature, one of these belongs to the good circuit, the remaining 15 cause aliasing. Since there are a total of $128 - 1 = 127$ bit-streams due to bad circuits, the fraction of bad-circuit bit-streams that cause aliasing is 15 / 127, or 0.118. If all bad circuit bit-streams are equally likely (and this is a poor assumption) then 0.118 is also the probability of aliasing.

In general, if the length of the test sequence is L and the length of the signature register is R the probability p of aliasing (not detecting an error) is

$$p = \frac{2^{L-R} - 1}{2^L - 1} . \qquad (14.12)$$

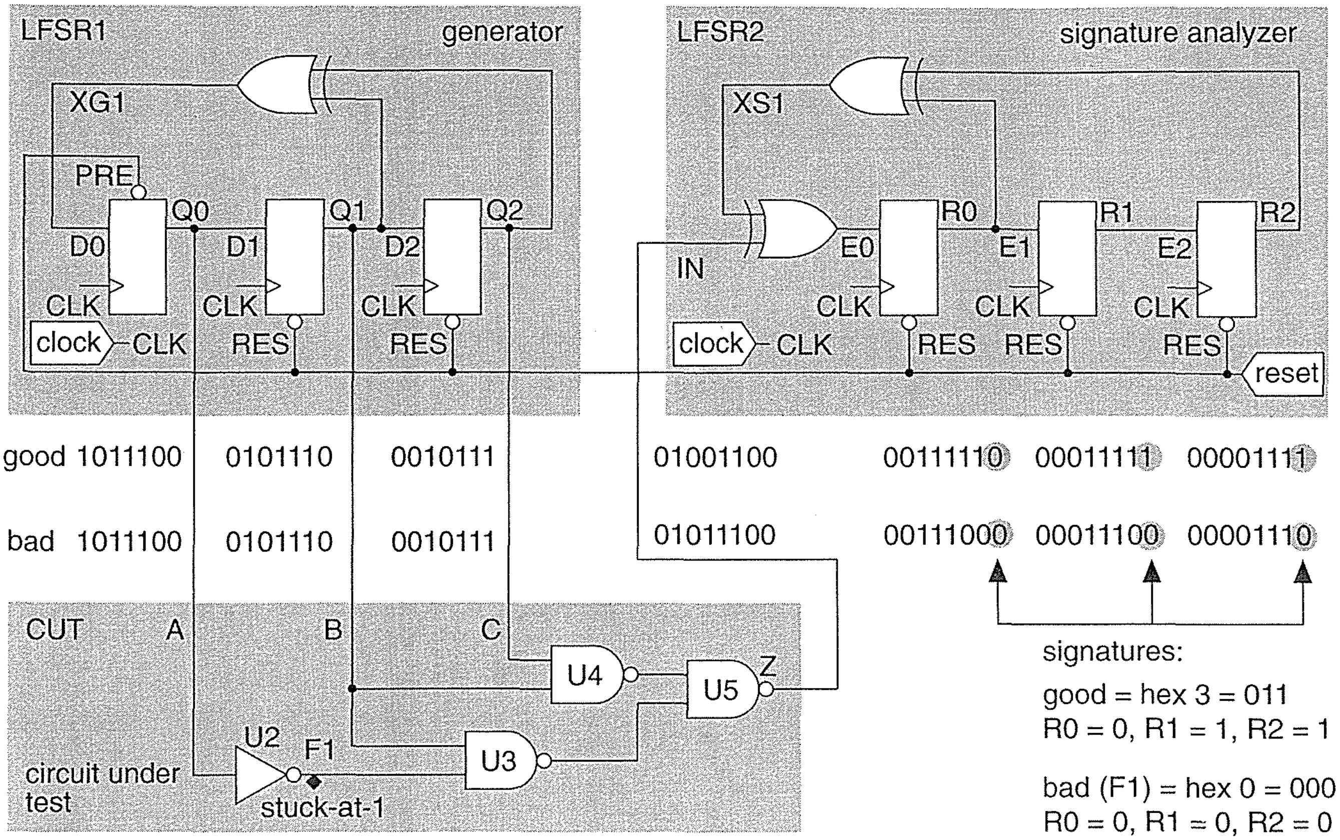
Thus, for the example in Figure 14.25, $L = 7$ and $R = 3$, and the probability of aliasing is $p = (2^{(7-3)} - 1) / (2^7 - 1) = 15/127 = 0.118$, as we have just calculated. This is a very high probability of error and we would not use such a short test sequence and such a short signature register in practice.

For $L >> R$ the error probability is

$$p \approx 2^{-R} . \qquad (14.13)$$

For example, if $R = 16$, $p \approx 0.0000152$ corresponding to an **error coverage** $(1 - p)$ of approximately 99.9984 percent. Unfortunately, these equations for error coverage are rather meaningless since there is no easy way to relate the error coverage to fault coverage. The problem lies in our assumption that all bad-circuit bit-streams are equally likely, and this is not true in practice (for example, bit-stream outputs of all ones or all zeros are more likely to occur as a result of faults). Nevertheless signature analysis with high error-coverage rates is found to produce high fault coverage.

(a)



(b)

$Q0_{t+1} =$ $Q1_t \oplus Q2_t$	$Q1_{t+1} = Q0_t$	$Q2_{t+1} = Q1_t$	$Z =$ $Q0'.Q1 + Q1.Q2$	$R0_{t+1} =$ $Z_t \oplus R0_t \oplus R2_t$	$R1_{t+1} = R0_t$	$R2_{t+1} = R1_t$
1	0	0	0	0	0	0
0	1	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	1	1	0
1	1	1	1	1	1	1
0	1	1	1	1	1	1
0	0	1	0	1	1	1
1	0	0	0	0	1	1

FIGURE 14.25 BIST example. (a) A simple BIST structure showing bit sequences for both good and bad circuits. (b) Bit sequence calculations for the good circuit. The signature appears on the eighth clock cycle (after seven positive clock edges) and is R0 = '0', R1 = '1', R2 = '1'; with R2 as the MSB this is '011' or hex 3.

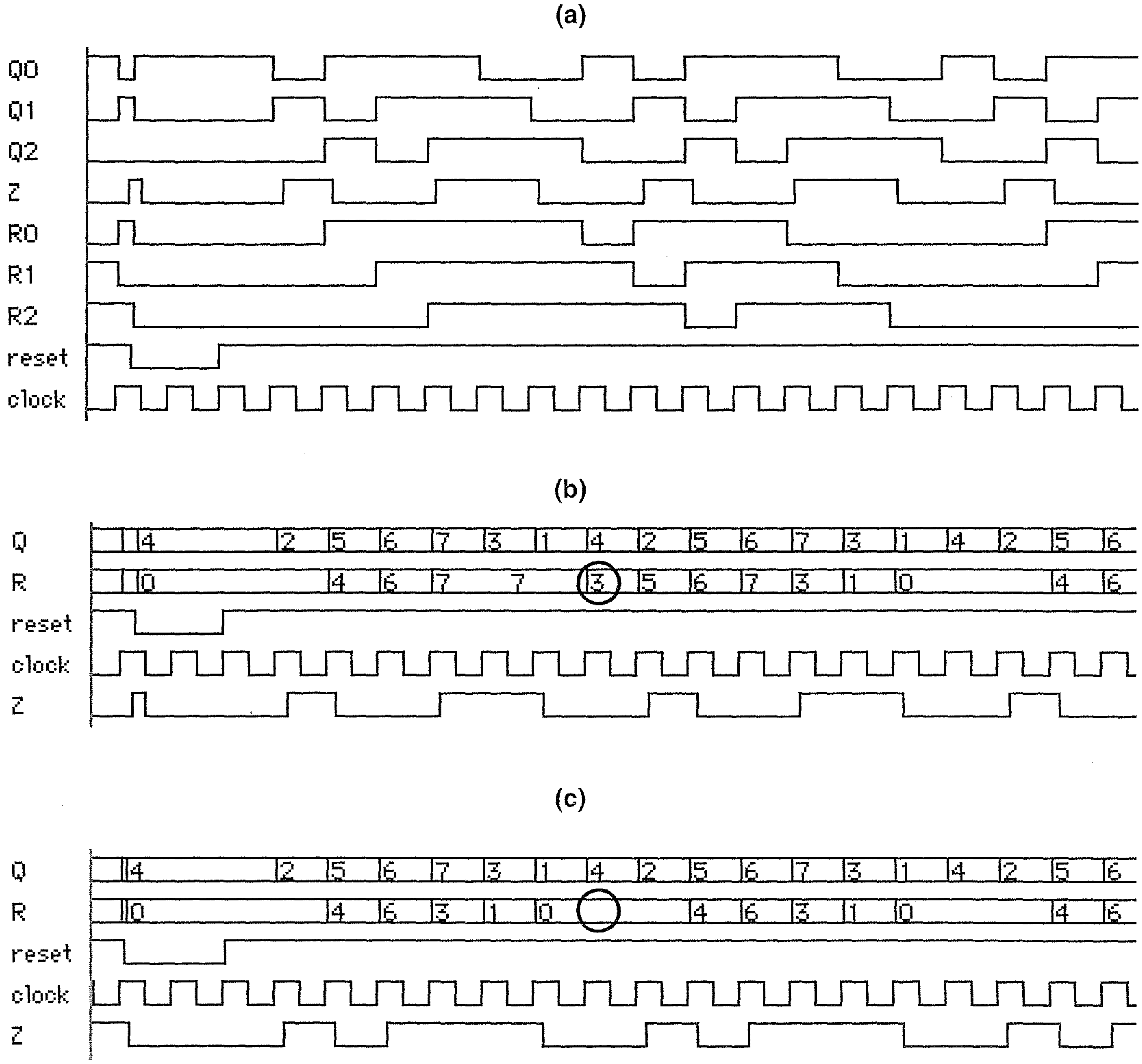


FIGURE 14.26 The waveforms of the BIST circuit of Figure 14.25. (a) The good-circuit response. The waveforms Q1 and Q2, as well as R1 and R2, are delayed by one clock cycle as they move through each stage of the shift registers. (b) The same good-circuit response with the register outputs Q0–Q2 and R0–R2 grouped and their values displayed in hexadecimal (Q0 and R0 are the MSBs). The signature hex 3 or '011' (R0 = 0, R1 = 1, R2 = 1) in R appears seven positive clock edges after the reset signal is taken high. This is one clock cycle after the generator completes its first sequence (hex pattern 4, 2, 5, 6, 7, 3, 1). (c) The response of the bad circuit with fault F1 and fault signature hex 0 (circled).

14.7.5 LFSR Theory

The operation of LFSRs is related to the mathematics of polynomials and Galois-field theory. The properties and behavior of these polynomials are well known and they are also used extensively in coding theory. Every LFSR has a **characteristic polynomial** that describes its behavior. The characteristic polynomials that cause an LFSR to generate a maximum-length PRBS are called **primitive polynomials.** Consider the primitive polynomial

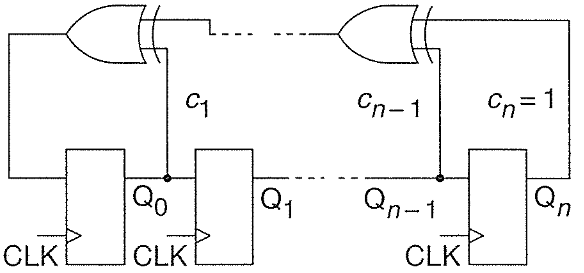
$$P(x) = 1 \oplus x^1 \oplus x^3, \qquad (14.14)$$

where $a \oplus b$ represents the exclusive-OR of a and b. The order of this polynomial is three, and the corresponding LFSR will generate a PRBS of length $2^3 - 1 = 7$. For a primitive polynomial of order n, the length of the PRBS is $2^n - 1$. Figure 14.27 shows the nonzero coefficients of some primitive polynomials [Golomb et al., 1982].

n	s	Octal	Binary
1	0, 1	3	11
2	0, 1, 2	7	111
3	0, 1, 3	13	1011
4	0, 1, 4	3	10011
5	0, 2, 5	45	100101
6	0, 1, 6	103	1000011
7	0, 1, 7	211	10001001
8	0, 1, 5, 6, 8	435	100011101
9	0, 4, 9	1021	1000010001
10	0, 3, 10	2011	10000001001

For $n=3$ and $s=0, 1, 3$: $c_0=1$, $c_1=1$, $c_2=0$, $c_3=1$

$P(x) = 1 \oplus c_1 x \oplus \ldots \oplus c_{n-1} x^{n-1} \oplus x^n$



or $P^*(x) = 1 \oplus c_{n-1} x \oplus \ldots \oplus c_1 x^{n-1} \oplus x^n$

FIGURE 14.27 Primitive polynomial coefficients for LFSRs (linear feedback shift registers) that generate a maximal-length PRBS (pseudorandom binary sequence). A schematic for a type 1 LFSR is shown.

Any primitive polynomial can be written as

$$P(x) = c_0 \oplus c_1 x^1 \oplus \ldots \oplus c_n x^n, \qquad (14.15)$$

where c_0 and c_n are always one. Thus for example, from Figure 14.27 for $n=3$, we see $s=0$, 1, 3; and thus the nonzero coefficients are c_0, c_1, and c_3. This corresponds to the primitive polynomial $P(x) = 1 \oplus x^1 \oplus x^3$. There is no easy way to determine the coefficients of primitive polynomials, especially for large n. There are many primitive polynomials for each n, but Figure 14.27 lists the one with the fewest nonzero coefficients.

The schematic in Figure 14.27 shows how the feedback taps on a LFSR correspond to the nonzero coefficients of the primitive polynomial. If the ith coefficient c_i is 1, then we include a feedback connection and an XOR gate in that position. If c_i is zero, there is no feedback connection and no XOR gate in that position.

The reciprocal of a primitive polynomial, $P^*(x)$, is also primitive, where

$$P^*(x) = x^n P(x^{-1}) . \tag{14.16}$$

For example, by taking the reciprocal of the primitive polynomial $P(x) = 1 \oplus x^1 \oplus x^3$ from Eq. 14.16, we can form

$$P^*(x) = 1 \oplus x^2 \oplus x^3 , \tag{14.17}$$

which is also a primitive polynomial.

This means that there are two possible LFSR implementations for every $P(x)$. Or, looked at another way, for every LFSR implementation, the characteristic polynomial can be written in terms of two primitive polynomials, $P(x)$ and $P^*(x)$, that are reciprocals of each other.

We may also implement an LFSR by using XOR gates in series with each flip-flop output rather than external to the shift register. The **external-XOR LFSR** is called a **type 1 LFSR** and the **internal-XOR LFSR** is called a **type 2 LFSR** (this is a nomenclature that most follow). Figure 14.28 shows the four different LFSRs that may be constructed for each primitive polynomial, $P(x)$.

There are differences between the four different LFSRs for each polynomial. Each gives a different output sequence. The outputs for the type 1 LFSRs, taken from the Q outputs of each flip-flop, are identical, but delayed by one clock cycle from the previous output. This is a problem when we use the parallel output from an LFSR to test logic because of the strong correlation between the test signals. The type 2 LFSRs do not have this problem. The type 2 LFSRs also are capable of higher-frequency operation since there are fewer series XOR gates in the signal path than in the corresponding type 1 LFSR. For these reasons, the type 2 LFSRs are usually used in BIST structures. The type 1 LFSR does have the advantage that it can be more easily constructed using register structures that already exist on an ASIC.

Table 14.11 shows primitive polynomial coefficients for higher values of n than Figure 14.27. Test length grows quickly with the size of the LFSR. For example, a 32-bit generator will produce a sequence with $2^{32} = 4{,}294{,}967{,}296 \approx 4.3 \times 10^9$ bits. With a 100 MHz clock (with 10 ns cycle time), the test time of 43 seconds would be impractical.

There is confusion over naming, labeling, and drawing of LFSRs in texts and test programs. Looking at the schematic in Figure 14.27, we can draw the LFSR with signals flowing from left to right or vice versa (two ways), we can name the leftmost flip-flop output Q_0 or Q_n (two more ways), and we can name the coefficient that goes with Q_0 either c_0 or c_{n-1} (two more ways). There are thus at least $2^3 \times 4$

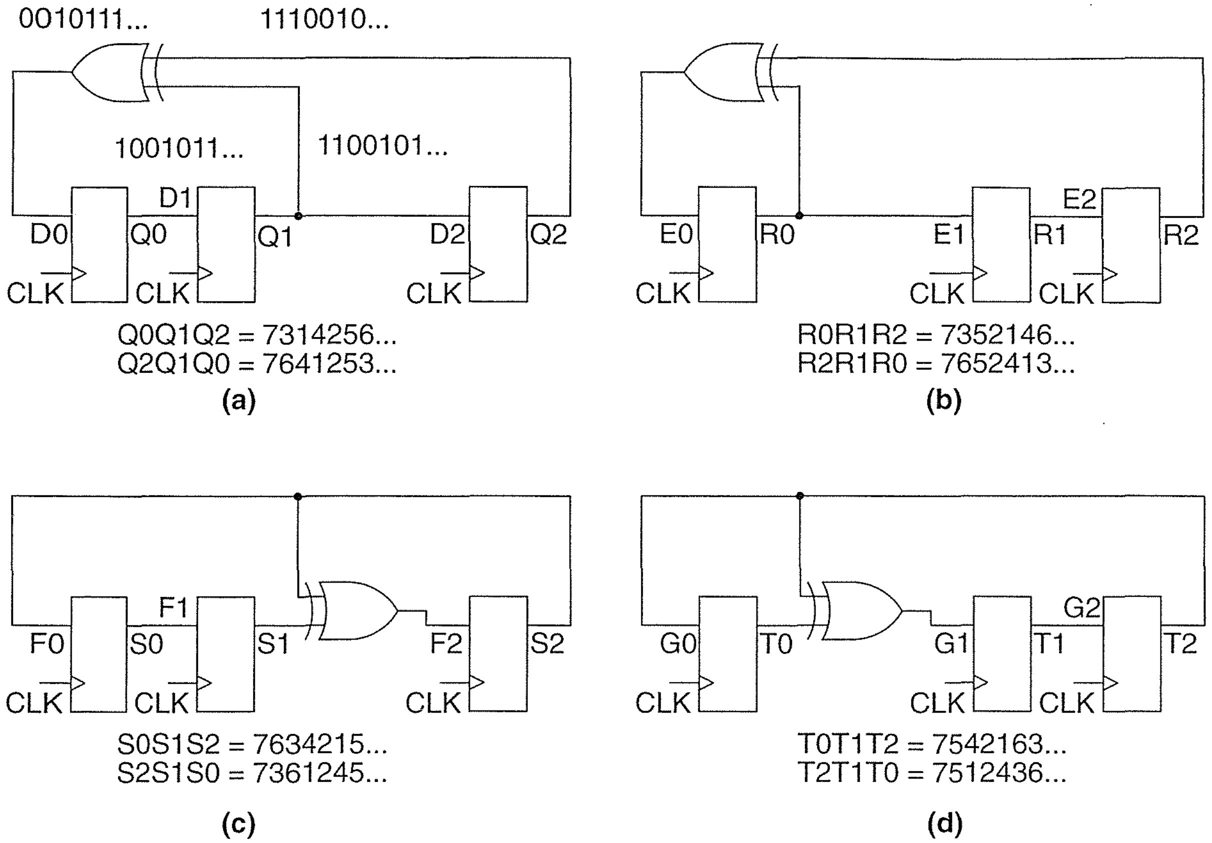


FIGURE 14.28 For every primitive polynomial there are four linear feedback shift registers (LFSRs). There are two types of LFSR; one type uses external XOR gates (type 1) and the other type uses internal XOR gates (type 2). For each type the feedback taps can be constructed either from the polynomial P(x) or from its reciprocal, P*(x). The LFSRs in this figure correspond to $P(x) = 1 \oplus x \oplus x^3$ and $P^*(x)= 1 \oplus x^2 \oplus x^3$. Each LFSR produces a different pseudorandom sequence, as shown. The binary values of the LFSR seen as a register, with the bit labeled as zero being the MSB, are shown in hexadecimal. The sequences shown are for each register initialized to '111', hex 7. (a) Type 1, P*(x). (b) Type 1, P(x). (c) Type 2, P(x). (d) Type 1, P*(x).

different ways to draw an LFSR for a given polynomial. Four of these are distinct. You can connect the LFSR feedback in the reverse order and the LFSR will still work—you will, however, get a different sequence. Usually this does not matter.

14.7.6 LFSR Example

We can use a cell compiler to produce LFSR and signature register BIST structures. For example, we might complete a property sheet as follows:

TABLE 14.11 Nonzero coefficients of primitive polynomials for LFSRs (linear feedback shift registers) that generate a maximal-length PRBS (pseudorandom binary sequence).

n	s	n	s	n	s	n	s
1	0, 1	11	0, 2, 11	21	0, 2, 21	31	0, 3, 31
2	0, 1, 2	12	0, 3, 4, 7, 12	22	0, 1, 22	32	0, 1, 27, 28, 32
3	0, 1, 3	13	0, 1, 3, 4, 13	23	0, 5, 23	40	0, 2, 19, 21, 40
4	0, 1, 4	14	0, 1, 11, 12, 14	24	0, 1, 3, 4, 24	50	0, 1, 26, 27, 50
5	0, 2, 5	15	0, 1, 15	25	0, 3, 25	60	0, 1, 60
6	0, 1, 6	16	0, 2, 3, 5, 16	26	0, 1, 7, 26	70	0, 1, 15, 16, 70
7	0, 1, 7	17	0, 3, 17	27	0, 1, 7, 27	80	0, 1, 37, 38, 80
8	0, 1, 5, 6, 8	18	0, 7, 18	28	0, 3, 28	90	0, 1, 18, 19, 90
9	0, 4, 9	19	0, 1, 5, 6, 19	29	0, 2, 29	100	0, 37, 100
10	0, 3, 10	20	0, 3, 20	30	0, 1, 15, 16, 30	256	0, 1, 3, 16, 256

```
property name          value    property name           value
-------------------    -----    -------------------     -----
LFSR_is_bilbo          false    LFSR_configuration      generator
LFSR_length            3        LFSR_init_hex_value     4
LFSR_scan              false    LFSR_mux_data           false
LFSR_mux_output        false    LFSR_xor_hex_function   max_length
LFSR_zero_state        false    LFSR_signature_inputs   1
```

The Verilog structural netlist for the compiled type 2 LFSR generator is shown in Table 14.12. According to our notation and the primitive polynomials in Figure 14.27, the corresponding primitive polynomial is $P^{*}(x) = 1 \oplus x^{2} \oplus x^{3}$. The LFSR has both serial and parallel outputs (taken from the inverted flip-flop outputs with inverting buffers, cell names in02d1). The clock and reset inputs are buffered (with noninverting buffers, cell names ni01d1) since these inputs would normally have to drive a load of more than 3 bits. Looking in the cell data book we find that the flip-flop corresponding to the MSB, instance FF0 with cell name dfptnb, has an active-low set input SDN. The remaining flip-flops, cell name dfctnb, have active-low clears, CDN. This gives us the initial value '100'.

Table 14.13 shows the serial-input signature register compiled using the reciprocal polynomial. Again the compiler has included buffers. All the flip-flops, cell names dfctnb, have active-low clear so that the initial content of the register is '000'.

TABLE 14.12 Compiled LFSR generator, using $P^*(x) = 1 \oplus x^2 \oplus x^3$.

```
module lfsr_generator (OUT, SERIAL_OUT, INITN, CP);
output [2:0] OUT; output SERIAL_OUT; input  INITN, CP;
  dfptnb FF2 (.D(FF0_Q), .CP(u4_Z), .SDN(u2_Z), .Q(FF2_Q), .QN(FF2_QN));
  dfctnb FF1 (.D(XOR0_Z), .CP(u4_Z), .CDN(u2_Z), .Q(FF1_Q), .QN(FF1_QN));
  dfctnb FF0 (.D(FF1_Q), .CP(u4_Z), .CDN(u2_Z), .Q(FF0_Q), .QN(FF0_QN));
  ni01d1 u2 (.I(u3_Z), .Z(u2_Z)); ni01d1 u3 (.I(INITN), .Z(u3_Z));
  ni01d1 u4 (.I(u5_Z), .Z(u4_Z)); ni01d1 u5 (.I(CP), .Z(u5_Z));
  xo02d1 XOR0 (.A1(FF2_Q), .A2(FF0_Q), .Z(XOR0_Z));
  in02d1 INV2X0 (.I(FF0_QN), .ZN(OUT[0]));
  in02d1 INV2X1 (.I(FF1_QN), .ZN(OUT[1]));
  in02d1 INV2X2 (.I(FF2_QN), .ZN(OUT[2]));
  in02d1 INV2X3 (.I(FF0_QN), .ZN(SERIAL_OUT));
endmodule
```

TABLE 14.13 Compiled serial-input signature register, using $P(x) = 1 \oplus x \oplus x^3$.

```
module lfsr_signature (OUT, SERIAL_OUT, INITN, CP, IN);
output [2:0] OUT; output SERIAL_OUT; input  INITN, CP; input  [0:0] IN;
  dfctnb FF2 (.D(XOR1_Z), .CP(u4_Z), .CDN(u2_Z), .Q(FF2_Q), .QN(FF2_QN));
  dfctnb FF1 (.D(FF2_Q), .CP(u4_Z), .CDN(u2_Z), .Q(FF1_Q), .QN(FF1_QN));
  dfctnb FF0 (.D(XOR0_Z), .CP(u4_Z), .CDN(u2_Z), .Q(FF0_Q), .QN(FF0_QN));
  ni01d1 u2 (.I(u3_Z), .Z(u2_Z)); ni01d1 u3 (.I(INITN), .Z(u3_Z));
  ni01d1 u4 (.I(u5_Z), .Z(u4_Z)); ni01d1 u5 (.I(CP), .Z(u5_Z));
  xo02d1 XOR1 (.A1(IN[0]), .A2(FF0_Q), .Z(XOR1_Z));
  xo02d1 XOR0 (.A1(FF1_Q), .A2(FF0_Q), .Z(XOR0_Z));
  in02d1 INV2X1 (.I(FF1_QN), .ZN(OUT[1]));
  in02d1 INV2X2 (.I(FF2_QN), .ZN(OUT[2]));
  in02d1 INV2X3 (.I(FF0_QN), .ZN(SERIAL_OUT));
  in02d1 INV2X0 (.I(FF0_QN), .ZN(OUT[0]));
endmodule
```

14.7.7 MISR

A serial-input signature register can only be used to test logic with a single output. We can extend the idea of a serial-input signature register to the **multiple-input signature register** (**MISR**) shown in Figure 14.29. There are several ways to connect the inputs to both types (type 1 and type 2) of LFSRs to form an MISR. Since the XOR operation is linear and associative, so that $(A \oplus B) \oplus C = A \oplus (B \oplus C)$, as long as the result of the additions are the same then the different representations are equivalent. If we have an *n*-bit long MISR we can accommodate up to *n* inputs to form the signature. If we use $m < n$ inputs we do not need the extra XOR gates in the last $n - m$ positions of the MISR.

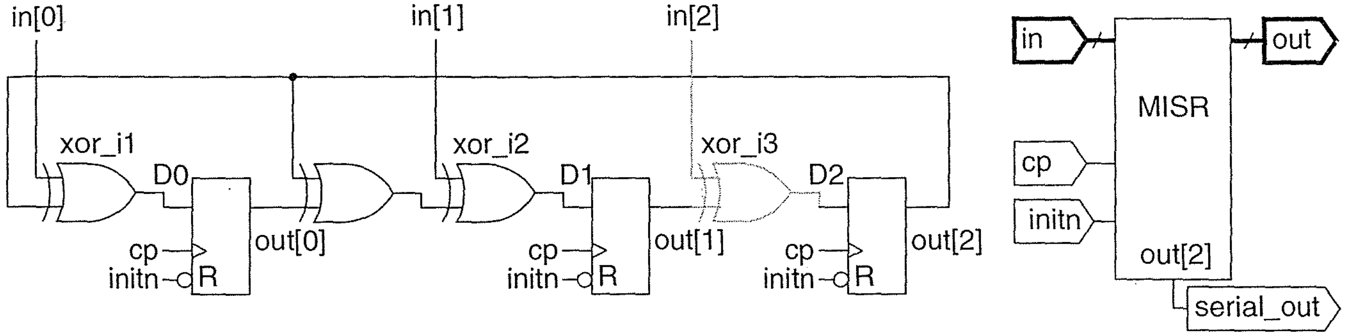


FIGURE 14.29 Multiple-input signature register (MISR). This MISR is formed from the type 2 LFSR (with $P^*(x) = 1 \oplus x^2 \oplus x^3$) shown in Figure 14.28(d) by adding XOR gates xor_i1, xor_i2, and xor_i3. This 3-bit MISR can form a signature from logic with three outputs. If we only need to test two outputs then we do not need XOR gate, xor_i3, corresponding to input in[2].

There are several types of BIST architecture based on the MISR. By including extra logic we can reconfigure an MISR to be an LFSR or a signature register; this is called a **built-in logic block observer** (**BILBO**). By including the logic that we wish to test in the feedback path of an MISR, we can construct circular BIST structures. One of these is known as the **circular self-test path** (**CSTP**).

We can test compiled blocks including RAM, ROM, and datapath elements using an LFSR generator and a MISR. To generate all 2^n address values for a RAM or ROM we can modify the LFSR feedback path to force entrance and exit from the all-zeros state. This is known as a **complete LFSR**. The pattern generator does not have to be an LFSR or exhaustive.

For example, if we were to apply an exhaustive test to a 4-bit by 4-bit multiplier this would require 2^8 or 256 vectors. An 8-bit by 8-bit multiplier requires 65,536 vectors and, if it were possible to test a 32-bit by 32-bit multiplier exhaustively, it would require 1.8×10^{19} vectors. Table 14.14 shows two sets of nonexhaustive test patterns, $\{SA\}$ and $\{SAE\}$, if A and B are both 4 bits wide. The test sequences $\{SA\}$ and $\{SAE\}$ consist of nested sequences of **walking 1's** and **walking 0's** (S1 and S1B), **walking pairs** (S2 and S2B), and triplets (S3, S3B). The sequences are extended for larger inputs, so that, for example, $\{S2\}$ is a sequence of seven vectors for an 8-bit input and so on. Intermediate sequences $\{SX\}$ and $\{SXB\}$ are concatenated from S1, S2, and S3; and from S1B, S2B, and S3B respectively. These sequences are chosen to exercise as many of the add-and-carry functions within the multiplier as possible.

TABLE 14.14 Multiplier test patterns.[1]

Sequence {SX}	Sequence {SXB}	Sequence {SA}	Sequence {SAE}
S1={1000 0100 0010 0001}	S1B={0111 1011 1101 0111}	{	{ { AB={S1, SX} }
S2={1100 0110 0011}	S2B={0011 1001 1100}	AB={S1, SX}	{ AB={S1B, SXB} }
S3={1110 0111}	S3B={0001 1000}	}	{ AB={S2, SX} }
			{ AB={S2B, SXB} }
SX={ {S1} {S2} {S3}}	SXB={ {S1B} {S2B} {S3B} }		{ AB={S3, SX} }
			{ AB={S3B, SXB} }
			}
Total = 3(X − 1) = 9, X = 4	Total = 3(X − 1) = 9, X = 4	Total = 4 × 9 = 3A(B − 1) = 36	Total = 3(2A − 1)(3B − 2) = 3 × 7 × 10 = 210

[1] {AB = {S1, SB} } means for each value of A in the sequence {S1} set B equal to all the values in {SB}.

The sequence length of {SA} is 3A (B − 1), and 3(2A − 1)(3B − 2) for {SAE}, where A and B are the sizes of the multiplier inputs. For example, {SA} is 168 vectors for A = B = 8 and 2976 vectors for A = B = 32; {SAE} is 990 vectors (A = B = 8) and 17,766 vectors (A = B = 32). From fault simulation, the stuck-at fault coverage is 93 percent for sequence {SA} and 97 percent for sequence {SAE}.

Figure 14.30 shows an MISR with a scan chain. We can now include the BIST logic as part of a boundary-scan chain, this approach is called **scanBIST**.

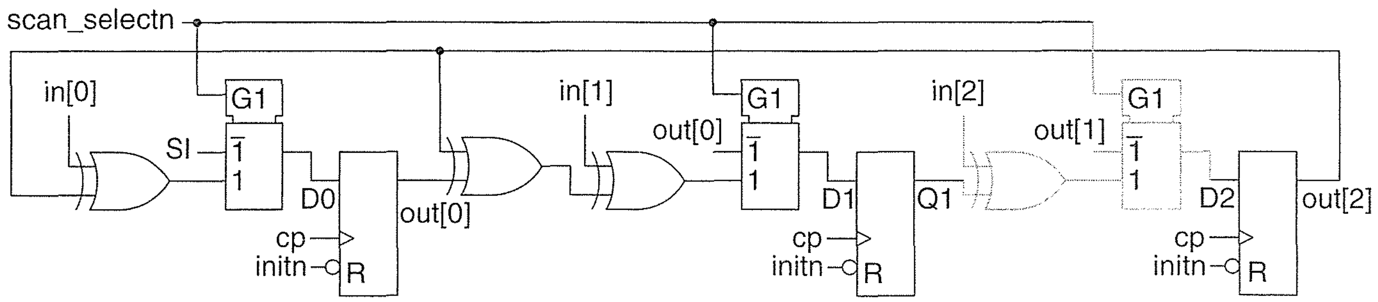


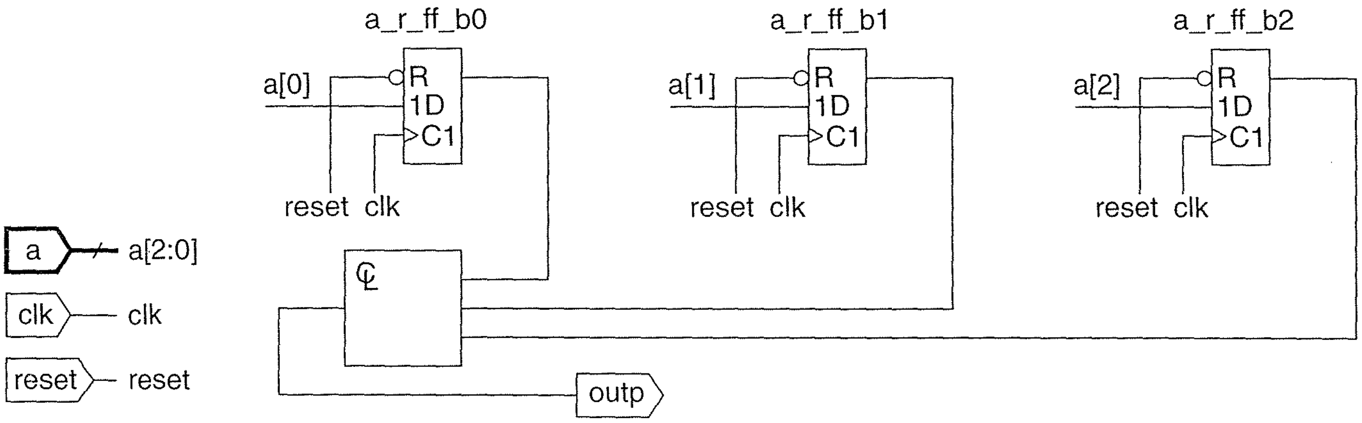
FIGURE 14.30 Multiple-input signature register (MISR) with scan generated from the MISR of Figure 14.29.

14.8 A Simple Test Example

As an example, we will describe automatic test generation using boundary scan together with internal scan. We shall use the function Z = A'B + BC for the core logic and register the three inputs using three flip-flops. We shall test the resulting sequential logic using a scan chain. The simple logic will allow us to see how the test vectors are generated.

14.8.1 Test-Logic Insertion

Figure 14.31 shows a structural Verilog model of the logic core. The three flip-flops (cell name `dfctnb`) implement the input register. The combinational logic implements the function, `outp = a_r[0]'.a_r[1] + a_r[1].a_r[2]`. This is the same function as Figure 14.14 and Figure 14.16.



```
module core_p (outp, reset, a, clk);
output outp; input  reset, clk; input  [2:0] a; wire  [2:0] a_r;
dfctnb a_r_ff_b0 (.D(a[0]), .CP(clk), .CDN(reset), .Q(a_r[0]), .QN(\a_r_ff_b0.QN ));
dfctnb a_r_ff_b1 (.D(a[1]), .CP(clk), .CDN(reset), .Q(a_r[1]), .QN(\a_r_ff_b1.QN ));
dfctnb a_r_ff_b2 (.D(a[2]), .CP(clk), .CDN(reset), .Q(a_r[2]), .QN(\a_r_ff_b2.QN ));
in01d0 u2 (.I(a_r[0]),   .ZN(u2_ZN));
nd02d0 u3 (.A1(u2_ZN),   .A2(a_r[1]),   .ZN(u3_ZN));
nd02d0 u4 (.A1(a_r[1]),  .A2(a_r[2]),   .ZN(u4_ZN));
nd02d0 u5 (.A1(u3_ZN),   .A2(u4_ZN),    .ZN(outp));
endmodule
```

FIGURE 14.31 Core of the Threegates ASIC.

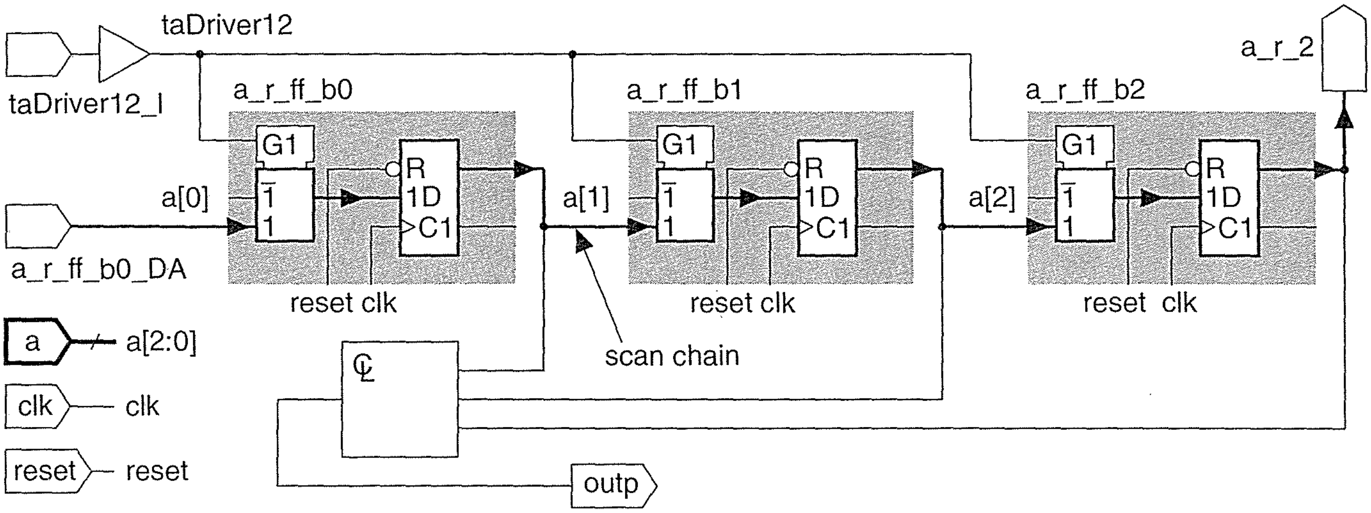
Table 14.15 shows the structural Verilog for the top-level logic of the Threegates ASIC including the I/O pads. There are nine pad cells. Three instances (up1_b0, up1_b1, and up1_b2) are the data-input pads, and one instance, up2_1, is the output pad. These were vectorized pads (even for the output that had a range of 1), so the synthesizer has added suffixes ('_1' and so on) to the pad instance names. Two pads are for power, one each for ground and the positive supply, instances up11 and up12. One pad, instance up3_1, is for the reset signal. There are two pad cells for the clock. Instance up4_1 is the clock input pad attached to the package pin and instance up6 is the clock input buffer.

The next step is to insert the boundary-scan logic and the internal-scan logic. Some synthesis tools can create test logic as they synthesize, but for most tools we need to perform **test-logic insertion** as a separate step. Normally we complete a parameter sheet specifying the type of test logic (boundary scan with internal scan in this case), as well as the ordering of the scan chain. In our example, we shall include all of the sequential cells in the boundary-scan register and order the boundary-scan cells using the pad numbers (in the original behavioral input). Figure 14.32 shows the modified core logic. The test software has changed all the flip-flops (cell names dfctnb) to scan flip-flops (with the same instance names, but the cell names are changed to mfctnb). The test software also adds a noninverting buffer to drive the scan-select signal to all the scan flip-flops.

The test software also adds logic to the top level. We do not need a detailed understanding of the automatically generated logic, but later in the design flow we will need to understand what has been done. Figure 14.33 shows a high-level view of the Threegates ASIC before and after test-logic insertion.

TABLE 14.15 The top level of the Threegates ASIC before test-logic insertion.

```
module asic_p (pad_outp, pad_a, pad_reset, pad_clk);
output [0:0] pad_outp; input  [2:0] pad_a; input [0:0] pad_reset, pad_clk;
wire [0:0] reset_sv, clk_sv, outp_sv; wire [2:0] a_sv; supply1 VDD; supply0 VSS;
core_p uc1 (.outp(outp_sv[0]), .reset(reset_sv[0]), .a(a_sv[2:0]), .clk(clk_bit));
pc3o07    up2_1    (.PAD(pad_outp[0]),    .I(outp_sv[0]));
pc3c01    up6      (.CCLK(clk_sv[0]),     .CP(clk_bit));
pc3d01r   up3_1    (.PAD(pad_reset[0]),   .CIN(reset_sv[0]));
pc3d01r   up4_1    (.PAD(pad_clk[0]),     .CIN(clk_sv[0]));
pc3d01r   up1_b0   (.PAD(pad_a[0]),       .CIN(a_sv[0]));
pc3d01r   up1_b1   (.PAD(pad_a[1]),       .CIN(a_sv[1]));
pc3d01r   up1_b2   (.PAD(pad_a[2]),       .CIN(a_sv[2]));
pv0f      up11     (.VSS(VSS));
pvdf      up12     (.VDD(VDD));
endmodule
```



```
module core_p_ta (a_r_2, outp, a_r_ff_b0_DA, taDriver12_I, a, clk, reset);
output a_r_2, outp; input a_r_ff_b0_DA, taDriver12_I;
input  [2:0] a; input  clk, reset; wire  [1:0] a_r; supply1 VDD; supply0 VSS;
ni01d5 taDriver12 (.I(taDriver12_I), .Z(taDriver12_Z));
mfctnb a_r_ff_b0 (.DA(a_r_ff_b0_DA), .DB(a[0]), .SA(taDriver12_Z), .CP(clk),
  .CDN(reset), .Q(a_r[0]), .QN(\a_r_ff_b0.QN ));
mfctnb a_r_ff_b1 (.DA(a_r[0]), .DB(a[1]), .SA(taDriver12_Z), .CP(clk), .CDN(reset),
  .Q(a_r[1]), .QN(\a_r_ff_b1.QN ));
mfctnb a_r_ff_b2 (.DA(a_r[1]), .DB(a[2]), .SA(taDriver12_Z), .CP(clk), .CDN(reset),
  .Q(a_r_2), .QN(\a_r_ff_b2.QN ));
in01d0 u2 (.I(a_r[0]),   .ZN(u2_ZN));
nd02d0 u3 (.A1(u2_ZN),   .A2(a_r[1]),   .ZN(u3_ZN));
nd02d0 u4 (.A1(a_r[1]),  .A2(a_r_2),    .ZN(u4_ZN));
nd02d0 u5 (.A1(u3_ZN),   .A2(u4_ZN),    .ZN(outp));
endmodule
```

FIGURE 14.32 The core of the Threegates ASIC after test-logic insertion.

14.8.2 How the Test Software Works

The structural Verilog for the Threegates ASIC is lengthy, so Figure 14.34 shows only the essential parts. The following main blocks are labeled in Figure 14.34:

A. This block is the logic core shown in Figure 14.32. The Verilog module header shows the "local" and "formal" port names. Arrows indicate whether each signal is an input or an output.

B. This is the main body of logic added by the test software. It includes the boundary-scan controller and clock control.

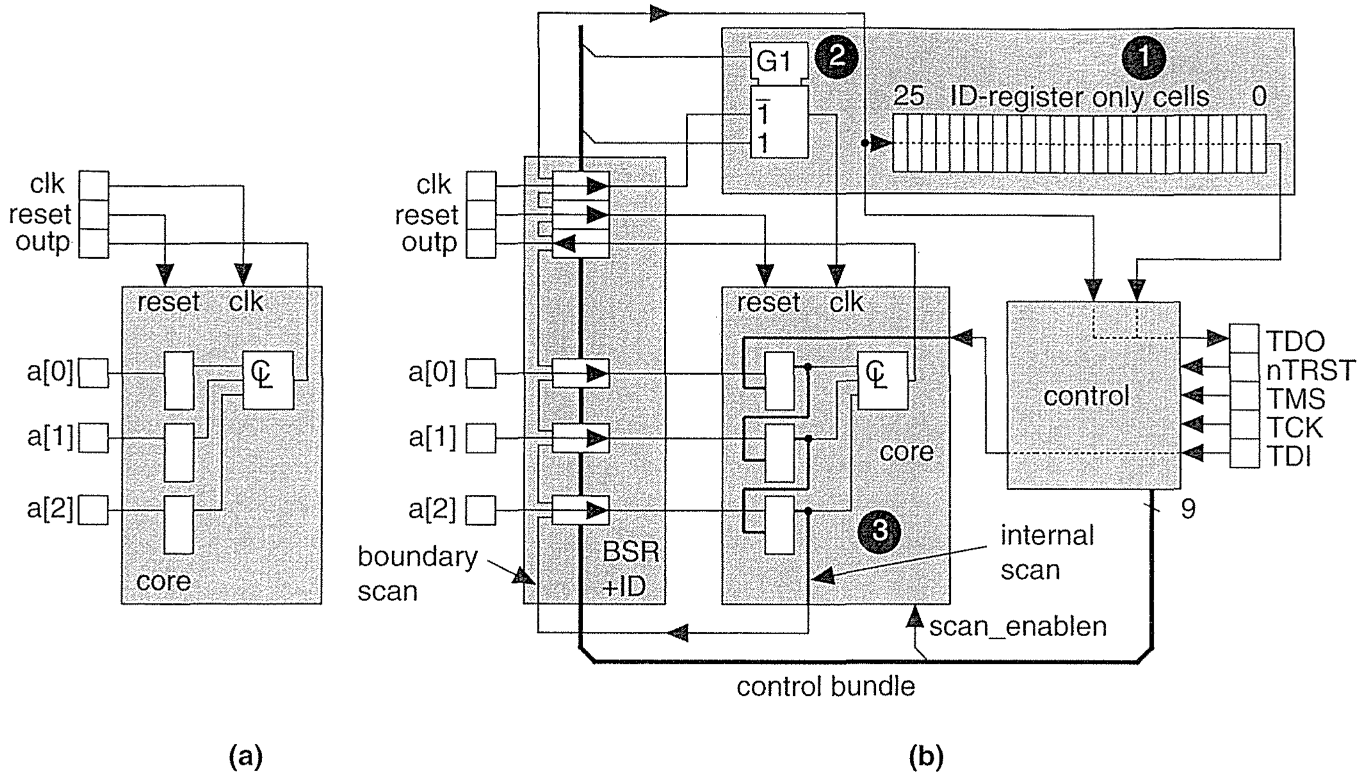


FIGURE 14.33 The Threegates ASIC. (a) Before test-logic insertion. (b) After test-logic insertion.

C. This block groups together the buffers that the test software has added at the top level to drive the control signals throughout the boundary-scan logic.

D. This block is the first boundary-scan cell in the BSR. There are six boundary-scan cells: three input cells for the data inputs, one output cell for the data output, one input cell for the reset, and one input cell for the clock. Only the first (the boundary-scan input cell for a[0]) and the last boundary-scan cells are shown. The others are similar.

E. This is the last boundary-scan cell in the BSR, the output cell for the clock.

F. This is the clock pad (with input connected to the ASIC package pin). The cell itself is unchanged by the test software, but the connections have been altered.

G. This is the clock-buffer cell that has not been changed.

H. The test software adds five I/O pads for the TAP. Four are input pad cells for TCK, TMS, TDO, and TRST. One is a three-state output pad cell for TDO.

I. The power pad cells remain unchanged.

J. The remaining I/O pad cells for the three data inputs, the data output, and reset remain unchanged, but the connections to the core logic are broken and the boundary-scan cells inserted.

The numbers in Figure 14.34 link the signals in each block to the following explanations:

1. The control signals for the input BSCs are `C_0,` `C_1,` `C_2,` and `C_4` and these are all buffered, together with the test clock `TCK`. The single output BSC also requires the control signal `C_3` and this is driven from the BST controller.
2. The clock enters the ASIC package through the clock pad as `.PAD(clk[0])` and exits the clock pad cell as `.CIN(up_4_1_CIN1)`. The test software routes this to the data input of the last boundary-scan cell as `.PI(up_4_1_CIN1)` and the clock exits as `.PO(up_4_1_cin)`. The clock then passes through the clock buffer, as before.
3. The serial input of the first boundary-scan cell comes from the controller as `.bst_control_BST_SI(test_logic_bst_control_BST_SI)`.
4. The serial output of the last boundary-scan cell goes to the controller as `.bst_control_BST(up4_1_bst_SO)`.
5. The beginning of the BSR is the first scan flip-flop in the core, which is connected to the TDI input as `.a_r_ff_b0_DA(ta_TDI_CIN)`.
6. The end of the scan chain leaves the core as `.a_r_2(uc1_a_r_2)` and enters the controller as `.bst_control_scan_SO(uc1_a_r_2)`.
7. The scan-enable signal `.bst_control_C_9(test_logic_bst_control_C_9)` is generated by the boundary-scan controller, and connects to the core as `.taDriver12_I(test_logic_bst_control_C_9)`.

The added test logic is shown in Figure 14.35. The blocks are as follows:

A. This is the module declaration for the test logic in the rest of the diagram, it corresponds to block B in Table 14.34.

B. This block contains buffers and clock control logic.

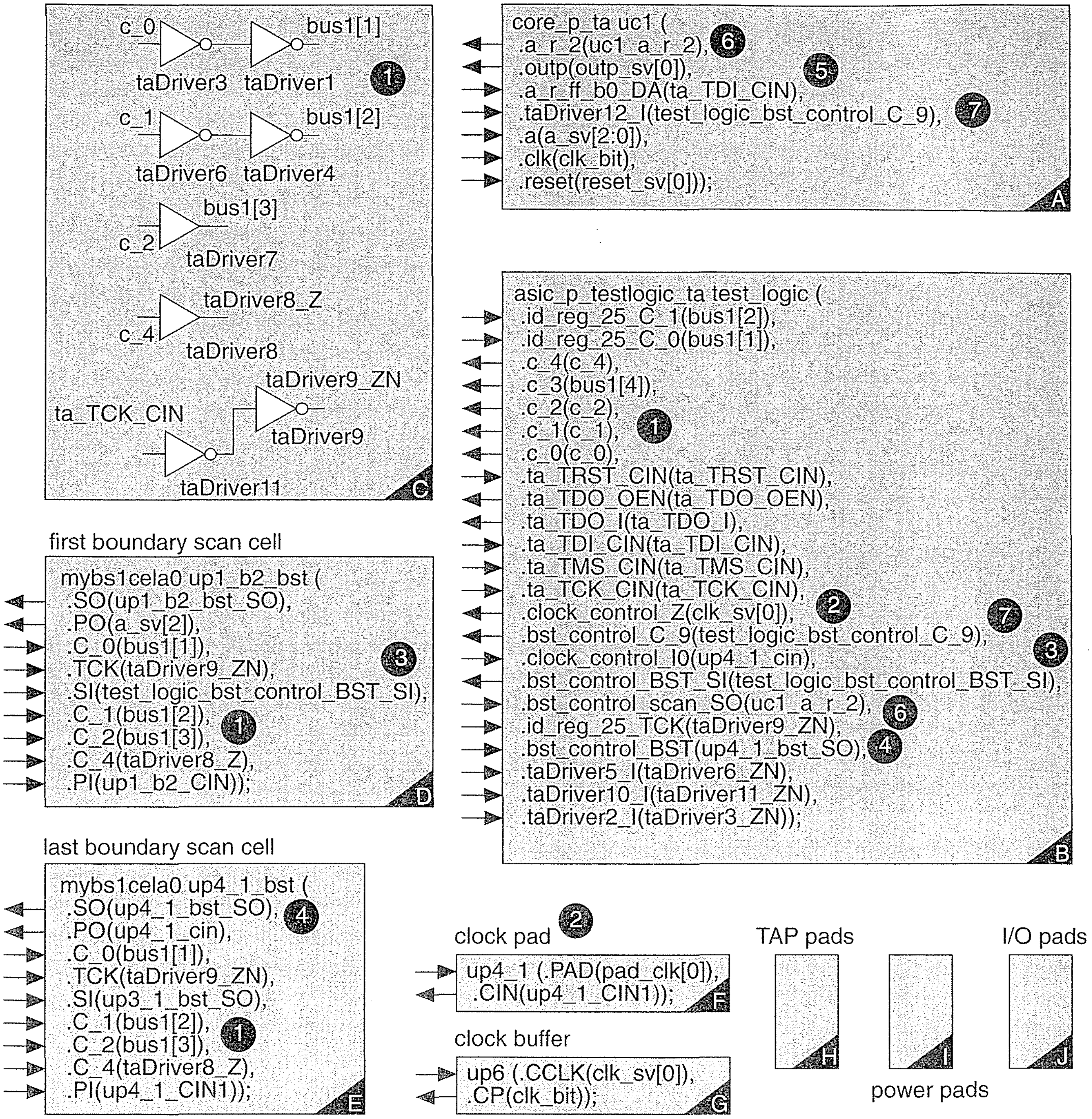
C. This is the boundary-scan controller.

D. This is the first of 26 IDR cells. In this implementation the IDCODE register is combined with the BSR. Since there are only six BSR cells we need (32 – 6) or 26 IDR cells to complete the 32-bit IDR.

E. This is the last IDR cell.

The numbers in Figure 14.35 refer to the following explanations:

1. The system clock (CLK, not the test clock TCK) from the top level (after passing through the boundary-scan cell) is fed through a MUX so that CLK may be controlled during scan.



```
core_p_ta uc1 (
 .a_r_2(uc1_a_r_2),
 .outp(outp_sv[0]),
 .a_r_ff_b0_DA(ta_TDI_CIN),
 .taDriver12_I(test_logic_bst_control_C_9),
 .a(a_sv[2:0]),
 .clk(clk_bit),
 .reset(reset_sv[0]));
```

```
asic_p_testlogic_ta test_logic (
 .id_reg_25_C_1(bus1[2]),
 .id_reg_25_C_0(bus1[1]),
 .c_4(c_4),
 .c_3(bus1[4]),
 .c_2(c_2),
 .c_1(c_1),
 .c_0(c_0),
 .ta_TRST_CIN(ta_TRST_CIN),
 .ta_TDO_OEN(ta_TDO_OEN),
 .ta_TDO_I(ta_TDO_I),
 .ta_TDI_CIN(ta_TDI_CIN),
 .ta_TMS_CIN(ta_TMS_CIN),
 .ta_TCK_CIN(ta_TCK_CIN),
 .clock_control_Z(clk_sv[0]),
 .bst_control_C_9(test_logic_bst_control_C_9),
 .clock_control_I0(up4_1_cin),
 .bst_control_BST_SI(test_logic_bst_control_BST_SI),
 .bst_control_scan_SO(uc1_a_r_2),
 .id_reg_25_TCK(taDriver9_ZN),
 .bst_control_BST(up4_1_bst_SO),
 .taDriver5_I(taDriver6_ZN),
 .taDriver10_I(taDriver11_ZN),
 .taDriver2_I(taDriver3_ZN));
```

first boundary scan cell

```
mybs1cela0 up1_b2_bst (
 .SO(up1_b2_bst_SO),
 .PO(a_sv[2]),
 .C_0(bus1[1]),
 .TCK(taDriver9_ZN),
 .SI(test_logic_bst_control_BST_SI),
 .C_1(bus1[2]),
 .C_2(bus1[3]),
 .C_4(taDriver8_Z),
 .PI(up1_b2_CIN));
```

last boundary scan cell

```
mybs1cela0 up4_1_bst (
 .SO(up4_1_bst_SO),
 .PO(up4_1_cin),
 .C_0(bus1[1]),
 .TCK(taDriver9_ZN),
 .SI(up3_1_bst_SO),
 .C_1(bus1[2]),
 .C_2(bus1[3]),
 .C_4(taDriver8_Z),
 .PI(up4_1_CIN1));
```

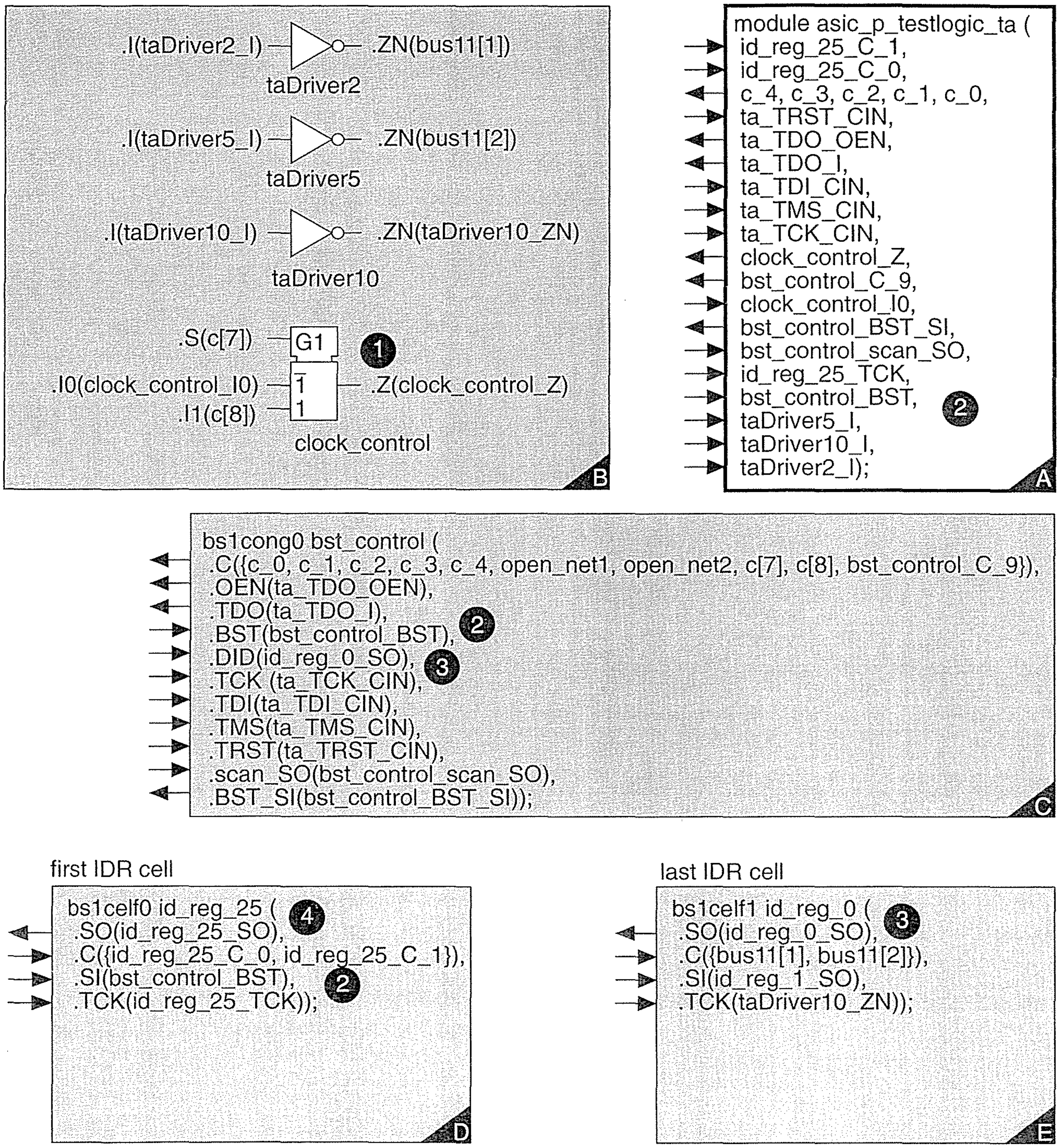
clock pad

```
up4_1 (.PAD(pad_clk[0]),
 .CIN(up4_1_CIN1));
```

clock buffer

```
up6 (.CCLK(clk_sv[0]),
 .CP(clk_bit));
```

FIGURE 14.34 The top level of the Threegates ASIC after test-logic insertion.



```
module asic_p_testlogic_ta (
 id_reg_25_C_1,
 id_reg_25_C_0,
 c_4, c_3, c_2, c_1, c_0,
 ta_TRST_CIN,
 ta_TDO_OEN,
 ta_TDO_I,
 ta_TDI_CIN,
 ta_TMS_CIN,
 ta_TCK_CIN,
 clock_control_Z,
 bst_control_C_9,
 clock_control_I0,
 bst_control_BST_SI,
 bst_control_scan_SO,
 id_reg_25_TCK,
 bst_control_BST,
 taDriver5_I,
 taDriver10_I,
 taDriver2_I);
```

```
bs1cong0 bst_control (
 .C({c_0, c_1, c_2, c_3, c_4, open_net1, open_net2, c[7], c[8], bst_control_C_9}),
 .OEN(ta_TDO_OEN),
 .TDO(ta_TDO_I),
 .BST(bst_control_BST),
 .DID(id_reg_0_SO),
 .TCK (ta_TCK_CIN),
 .TDI(ta_TDI_CIN),
 .TMS(ta_TMS_CIN),
 .TRST(ta_TRST_CIN),
 .scan_SO(bst_control_scan_SO),
 .BST_SI(bst_control_BST_SI));
```

```
bs1celf0 id_reg_25 (
 .SO(id_reg_25_SO),
 .C({id_reg_25_C_0, id_reg_25_C_1}),
 .SI(bst_control_BST),
 .TCK(id_reg_25_TCK));
```

```
bs1celf1 id_reg_0 (
 .SO(id_reg_0_SO),
 .C({bus11[1], bus11[2]}),
 .SI(id_reg_1_SO),
 .TCK(taDriver10_ZN));
```

FIGURE 14.35 Test logic inserted in the Threegate ASIC.

TABLE 14.16 The TAP (test-access port) control.[1]

TAP state	C_0	C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8[2]	C_9
Reset	x	x	xxxx0xx	xxxx0xx	xxxx0xx	xxxx0xx	xxxx1xx	xxxx0xx	xxxx0xx	xxxx0xx
Run_Idle	00x0xxx	11x1xxx	0	1001011	0001011	0000010	1	0000001	0000000	0000000
Select_DR	00x0xxx	11x1xxx	0	1001011	0001011	0000010	1	0000001	0000000	0000000
Capture_DR	00x01xx	00x00xx	0	1001011	0001011	0000010	1	0000001	000000T	0000000
Shift_DR	11x11xx	11x11xx	0	1001011	0001011	0000010	1111101	0000001	000000T	0000001
Exit1_DR	00x00xx	11x11xx	0	1001011	0001011	0000010	1	0000001	0000000	0000000
Pause_DR	00x00xx	11x11xx	0	1001011	0001011	0000010	1	0000001	0000000	0000000
Exit2_DR	00x00xx	11x11xx	0	1001011	0001011	0000010	1	0000001	0000000	0000000
Update_DR	00x0xxx	11x1xxx	110100	1001011	0001011	0	1111101	0000001	0000000	0000000
Select_IR	x	x	0	1001011	0001011	00000x0	11111x1	0000001	0000000	0000000
Capture_IR	x	x	0	1001011	0001011	00000x0	11111x1	0000001	0000000	0000000
Shift_IR	x	x	0	1001011	0001011	00000x0	11111x1	0000001	0000000	0000000
Exit1_IR	x	x	0	1001011	0001011	00000x0	11111x1	0000001	0000000	0000000
Pause_IR	x	x	0	1001011	0001011	00000x0	11111x1	0000001	0000000	0000000
Exit2_IR	x	x	0	1001011	0001011	00000x0	11111x1	0000001	0000000	0000000
Update_IR	x	x	0	1001011	0001011	00000x0	1111101	0000001	0000000	0000000

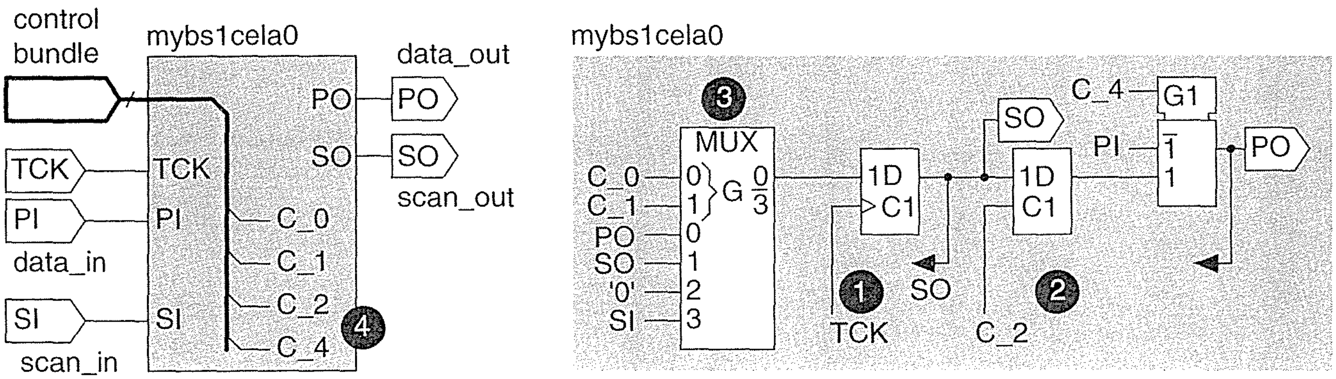
[1] Outputs are specified for each instruction as 0123456, where: 0 = EXTEST, 1 = SAMPLE, 2 = BYPASS, 3 = INTEST, 4 = IDCODE, 5 = RUNBIST, 6 = SCANM.

[2] T denotes gated clock TCK.

2. The signal `bst_control_BST` is the end (output) of the boundary-scan cells and the start (input) to the ID register only cells.
3. The signal `id_reg_0_SO` is the end (output) of the ID register.
4. The signal `bst_control_BST_SI` is the start of the boundary-scan chain.

The job of the boundary-scan controller is to produce the control signals (`C_1` through `C_9`) for each of the 16 TAP controller states (`reset` through `update_IR`) for each different instruction. In this BST implementation there are seven instructions: the required `EXTEST`, `SAMPLE`, and `BYPASS`; `IDCODE`; `INTEST` (which is the equivalent of `EXTEST`, but for internal test); `RUNBIST` (which allows on-chip test structures to operate); and `SCANM` (which controls the internal-scan chains). The boundary-scan controller outputs are shown in Table 14.16.

There are two very important differences between this controller and the one described in Table 14.5. The first, and most obvious, is that the control signals now depend on the instruction. This is primarily because INTEST requires the control signal at the output of the BSCs to be in different states for the input and output cells. The second difference is that the logic for the boundary-scan cell control signals is now purely combinational—we have removed the gated clocks. For example, Figure 14.36 shows the input boundary-scan cell. The clock for the shift flip-flop is now TCK and not a gated clock as it was in Table 14.5. We can do this because the output of the flip-flop, SO, the scan output, is added as input to the MUX that feeds the flip-flop data input. Thus, when we wish to hold the state of the flip-flop, the control signals select SO to be connected from the output to the input. This is called a **polarity-hold flip-flop**. Unfortunately, we have little choice but to gate the system clock if we make the scan chain part of the BSR. We cannot have one clock for part of the BSR and another for the rest. The costly alternative is to change every scan flip-flop to a scanned polarity-hold flip-flop.



```
module mybs1cela0 (SO, PO, C_0, TCK, SI, C_1, C_2, C_4, PI);
output SO, PO; input  C_0, C_1, C_2, C_4, TCK, SI, PI;
in01d1 inv_0      (.I(C_0), .ZN(iv0_ZN));
in01d1 inv_1      (.I(C_1), .ZN(iv1_ZN));
oa03d1 oai221_1   (.A1(C_0), .A2(SO), .B1(iv0_ZN), .B2(SI), .C(C_1), .ZN(oa1_ZN));
nd02d1 nand2_1    (.A1(na2_ZN), .A2(oa1_ZN), .ZN(na1_ZN));
nd03d1 nand3_1    (.A1(PO), .A2(iv0_ZN), .A3(iv1_ZN), .ZN(na2_ZN));
mx21d1 mux21_1    (.I0(PI), .I1(upo), .S(C_4), .Z(PO));
dfntnb dff_1      (.D(na1_ZN), .CP(TCK), .Q(SO), .QN(\so.QN ));
lantnb latch_1    (.E(C_2), .D(SO), .Q(upo), .QN(\upo.QN ));
endmodule
```

FIGURE 14.36 Input boundary-scan cell (BSC) for the Threegates ASIC. Compare this to the generic data-register (DR) cell (used as a BSC) shown in Figure 14.2.

14.8.3 ATVG and Fault Simulation

Table 14.17 shows the results of running the Compass ATVG software on the Threegates ASIC. We might ask: Why so many faults? and why is the fault coverage so poor? First we look at the details of the test software output. We notice the following:

- Line 2. The backtrace limit is 30. We do not have any deep complex combinational logic so that this should not cause a problem.
- Lines 4–6. An uncollapsed fault count of 184 indicates the test software has inserted faults on approximately 100 nodes, or at most 50 gates assuming a fanout of 1, less gates with any realistic fanout. Clearly this is less than all of the test logic that we have inserted.

To discover why the fault coverage is 68.25 percent we must examine each of the fault categories. First, Table 14.18 shows the undetected faults.

The ATVG program is generating tests for the core using internal scan. We cannot test the BST logic itself, for example. During the production test we shall test the BST logic first, separately from the core—this is often called a **flush test**. Thus we can ignore any faults from the BST logic for the purposes of internal-scan testing.

Next we find two redundant faults: `TA_TDO.1.I` `sa0` and `sa1`. Since `TDO` is three-stated during the test, it makes no difference to the function of the logic if this node is tied high or low—hence these faults are redundant. Again we should ensure these faults will be caught during the flush test. Finally, Table 14.19 shows the tied faults.

Now that we can explain all of the undetectable faults, we examine the detected faults. Table 14.20 shows only the detected faults in the core logic. Faults F1–F8 in the first part of Table 14.20 correspond to the faults in Figure 14.16. The fault list in the second part of Table 14.20 shows each fault in the core and whether it was detected (D) or collapsed and detected as an equivalent fault (CD). There are no undetected faults (U) in the logic core.

14.8.4 Test Vectors

Next we generate the test vectors for the Threegates ASIC. There are three types of vectors in scan testing. **Serial vectors** are the bit patterns that we shift into the scan chain. We have three flip-flops in the scan chain plus six boundary-scan cells, so each serial vector is 9 bits long. There are serial input vectors that we apply as a stimulus and serial output vectors that we expect as a response. **Parallel vectors** are applied to the pads before we shift the serial vectors into the scan chain. We have nine input pads (three core data, one core clock, one core reset, and four input TAP pads—`TMS`, `TCK`, `TRST`, and `TDI`) and two outputs (one core data output and `TDO`). Each parallel vector is thus 11 bits long and contains 9 bits of stimulus and 2 bits of response. A test program consists of applying the stimulus bits from one parallel vector to the nine input pins for one test cycle. In the next nine test cycles we shift a 9-bit stimulus from a serial vector into the scan chain (and receive a 9-bit

TABLE 14.17 ATVG (automatic test-vector generation) report for the Threegates ASIC.

```
CREATE: Output vector database cell defaulted to [svf]asic_p_ta
CREATE: Backtrack limit defaulted to 30
CREATE: Minimal compression effort: 10 (default)
Fault list generation/collapsing
Total number of faults: 184
Number of faults in collapsed fault list: 80
Vector generation
#
# VECTORS  FAULTS   FAULT COVER
#        processed
#
#     5      184     60.54%
#
# Total number of backtracks: 0
# Highest backtrack         : 0
# Total number of vectors   : 5
#
# STAR RESULTS summary
#                            Noncollapsed       Collapsed
# Fault counts:
#   Aborted                     0                 0
#   Detected                    89                43
#   Untested                    58                20
#                             ------            ------
#   Total of detectable         147               63
#
#   Redundant                   6                 2
#   Tied                        31                15
#
# FAULT COVERAGE              60.54 %           68.25 %
#
# Fault coverage = nb of detected faults / nb of detectable faults
Vector/fault list database [svf]asic_p_ta created.
```

response, the result of the previous tests, from the scan chain). We can generate the serial and parallel vectors separately, or we can merge the vectors to give a set of **broadside vectors**. Each broadside vector corresponds to one test cycle and can be used for simulation. Some testers require broadside vectors; others can generate them from the serial and parallel vectors.

Table 14.21 shows the serial test vectors for the Threegates ASIC. The third serial test vector is '110111010'. This test vector is shifted into the BSR, so that the first three bits in this vector end up in the first three bits of the BSR. The first three bits of the BSR, nearest TDI, are the scan flip-flops, the other six bits are

TABLE 14.18 Untested faults (not observable) for the Threegates ASIC.

Faults	Explanation
`TADRIVER4.ZN sa0`	Internal driver for BST control bundle (seven more faults like this).
`TA_TRST.1.CIN sa0`	BST reset TRST is active-low and tied high during test.
`TDI.O sa0 sa1`	TDI is BST serial input.
`UP1_B0.1.CIN sa0 sa1`	Data input pad (two more faults like this one).
`UP3_1.1.CIN sa0`	System reset is active-low and tied high during test.
`UP4_1.1.CIN sa0 sa1`	System clock input pad.
`# Total number: 20`	

TABLE 14.19 Tied faults.

Fault(s)	Explanation
`TADRIVER1.ZN sa0`	Internal BST buffer (seven more faults like this one).
`TA_TMS.1.CIN sa0`	TMS input tied low.
`TA_TRST.1.CIN sa1`	TRST input tied high.
`TEST_LOGIC.BST_CONTROL.U1.ZN sa1`	Internal BST logic.
`UP1_B0_BST.U1.A2 sa0`	Input pad (two more faults like this).
`UP3_1.1.CIN sa1`	Reset input pad tied high.
`# Total number: 15`	

boundary-scan cells). Since UC1.A_R_FF_B0.Q is a_r[0] and so on, the third test vector will set a_r = 011 where a_r[2] = 0. This is the vector we require to test the function a_r[0]'.a_r[1] + a_r[1].a_r[2] for fault UC1.U2.ZN sa1 in the Threegates ASIC. From Figure 14.31 we see that this is a stuck-at-1 fault on the output of the inverter whose input is connected to a_r[0]. This fault corresponds to fault F1 in the circuit of Figure 14.16. The fault simulation we performed earlier told us the vector ABC = 011 is a test for fault F1 for the function A'B + BC.

14.8.5 Production Tester Vector Formats

The final step in test-program generation is to format the test vectors for the production tester. As an example the following shows the Sentry tester file format for testing a D flip-flop. For an average ASIC there would be thousands of vectors in this file.

```
# Pin declaration: pin names are separated by semi-colons (all pins
# on a bus must be listed and separated by commas)
pre_; clr_; d; clk; q; q_;
```

TABLE 14.20 Detected core-logic faults in the Threegates ASIC.

Fault(s)	Explanation
`UC1.U2.ZN sa1`	F1
`UC1.U3.A2 sa1`	F2
`UC1.U3.ZN sa1`	F5
`UC1.U4.A1 sa1`	F3
`UC1.U4.ZN sa1`	F6
`UC1.U5.ZN sa0`	F8
`UC1.U5.ZN sa1`	F7
`UC1.A_R_FF_B2.Q.O sa1`	F4
`Fault list`	
`UC1.A_R_FF_B0.Q: (O) CD CD`	SA0 and SA1 collapsed to U3.A1
`UC1.A_R_FF_B1.Q: (O) D D`	SA0 and SA1 detected.
`UC1.A_R_FF_B2.Q: (O) CD D`	SA0 collapsed to U2. SA1 is F4.
`UC1.U2: (I) CD CD (ZN) CD D`	I.SA1/0 collapsed to O.SA1/0. O. SA1 is F1.
`UC1.U3: (A1) CD CD (A2) CD D (ZN) CD D`	A1.SA1 collapsed to U2.ZN.SA1.
`UC1.U4: (A1) CD D (A2) CD CD (ZN) CD D`	A2.SA1 collapsed to A_R_FF_B2.Q.SA1.
`UC1.U5: (A1) CD CD (A2) CD CD (ZN) D D`	A1.SA1 collapsed to U3.ZN.SA1

TABLE 14.21 Serial test vectors

	Serial-input scan data								
#1	1	1	1	0	1	0	1	1	0
#2	1	0	1	1	0	1	0	0	1
#3	1	1	0	1	1	1	0	1	0
#4	0	0	0	1	0	0	0	0	0
#5	0	1	0	0	1	1	1	0	1
	^UC1.A_R_FF_B0.Q	^UC1.A_R_FF_B1.Q	^UC1.A_R_FF_B2.Q	^UP1_B2_BST.SO.Q	^UP1_B1_BST.SO.Q	^UP1_B0_BST.SO.Q	^UP2_1_BST.SO.Q	^UP3_1_BST.SO.Q	^UP4_1_BST.SO.Q

Fault	Fault number	Vector number	Core input
`UC1.U2.ZN sa1`	**F1**	**3**	**011**
`UC1.U3.A2 sa1`	F2	4	000
`UC1.U3.ZN sa1`	F5	5	010
`UC1.U4.A1 sa1`	F3	2	101
`UC1.U4.ZN sa1`	F6	1	111
`UC1.U5.ZN sa0`	F8	1	111
`UC1.U5.ZN sa1`	F7	2	101
`UC1.A_R_FF_B2.Q.O sa1`	F4	2	101

```
# Pin declarations are separated from test vectors by $
$
# The first number on each line is the time since start in ns,
# followed by space or a tab.
# The symbols following the time are the test vectors
# (in the same order as the pin declaration)
# an "=" means don't do anything
# an "s" means sense the pin at the beginning of this time point
# (before the input changes at this time point have any effect)
#
#  pcdcqq
#  rlal _
#  ertk
#  __a
00 1010== # clear the flip-flop
10 1110ss # d=1, clock=0
20 1111ss # d=1, clock=1
30 1110ss # d=1, clock=0
40 1100ss # d=0, clock=0
50 1101ss # d=0, clock=1
60 1100ss # d=0, clock=0
70 ====ss
```

14.8.6 Test Flow

Normally we leave test-vector generation and the production-test program generation until the very last step in ASIC design after physical design is complete. All of the steps have been described before the discussion of physical design, because it is still important to consider test very early in the design flow. Next, as an example of considering test as part of logical design, we shall return to our Viterbi decoder example.

14.9 The Viterbi Decoder Example

Table 14.22 shows the timing analysis for the Viterbi decoder before and after test insertion. The Compass test software inserts internal scan and boundary scan exactly as in the Threegates example. The timing analysis is in the form of histograms showing the distributions of the timing delays for all paths. In this analysis we set an aggressive constraint of 20 ns (50 MHz) for the clock. The critical path before test insertion is 21.75 ns (the slack is thus negative at –1.75 ns). The path starts at `u1.subout6.Q_ff_b0` and ends at `u2.metric0.Q_ff_b4`, both flip-flops inside the flattened block, `v_1.u100`, that we created during synthesis in an attempt to improve speed. The first flip-flop in the path is a `dfctnb`; the last flip-flop is a `dfctnh`. The suffix `'b'` denotes 1X drive and suffix `'h'` denotes 2X drive.

TABLE 14.22 **Timing effects of test-logic insertion for the Viterbi decoder.**

Timing of critical paths before test-logic insertion

```
#   Slack(ns)  Num Paths
#    -3.3826     1       *
#    -1.7536     18      *******
#     -.1245     4       **
#     1.5045     1       *
#     3.1336     0       *
#     4.7626     0       *
#     6.3916     134     ******************************************
#     8.0207     6       ***
#     9.6497     3       **
#    11.2787     0       *
#    12.9078     24      ********
```

```
# instance name
# inPin --> outPin        incr     arrival  trs  rampDel  cap    cell
#                         (ns)      (ns)           (ns)   (pf)

# v_1.u100.u1.subout6.Q_ff_b0
# CP --> QN                 1.73      1.73    R    .20      .10    dfctnb
...
# v_1.u100.u2.metric0.Q_ff_b4
# setup: D --> CP            .16     21.75    F    .00      .00    dfctnh
```

After test-logic insertion

```
#    -4.0034     1       *
#    -1.9835     18      *****
#      .0365     4       **
#     2.0565     1       *
#     4.0764     0       *
#     6.0964     138     *******************************
#     8.1164     2       *
#    10.1363     3       **
#    12.1563     24      ******
#    14.1763     0       *
#    16.1963     187     ******************************************
```

```
# v_1.u100.u1.subout7.Q_ff_b1
# CP --> Q                  1.40      1.40    R    .28      .13    mfctnb
...
# v_1.u100.u2.metric0.Q_ff_b4
# setup: DB --> CP           .39     21.98    F    .00      .00    mfctnh
```

After test insertion the critical path is 21.98 ns. The end point is identical, but the start point is now subout7.Q_ff_b1. This is not too surprising. What is happening is that there are a set of paths of nearly equal length. Changing the flip-flops to their scan versions (mfctnb and mfctnh) increases the delay slightly. The exact delay depends on the capacitive load at the output, the path (clock-to-Q, clock-to-QN, or setup), and the input signal rise time.

Adding test logic has not increased the critical path delay substantially. Almost as important is that the distribution of delays has not changed substantially. Also very important is the fact that the distributions show that there are only approxi-

TABLE 14.23 Fault coverage for the Viterbi decoder.

```
Fault list generation/collapsing
Total number of faults: 8846
Number of faults in collapsed fault list: 3869
Vector generation
#
# VECTORS  FAULTS   FAULT COVER
#         processed
#
#     20     7515     82.92%
#     40     8087     89.39%
#     60     8313     91.74%
#     80     8632     95.29%
#     87     8846     96.06%

# Total number of backtracks: 3000
# Highest backtrack         : 30
# Total number of vectors   : 87

# STAR RESULTS summary
#                           Noncollapsed       Collapsed
# Fault counts:
#   Aborted                    178               85
#   Detected                   8427              3680
#   Untested                   168               60
#                            ------            ------
#   Total of detectable        8773              3825
#
#   Redundant                  10                6
#   Tied                       63                38
#
# FAULT COVERAGE              96.06 %           96.21 %
```

mately 20 paths with delays close to the critical path delay. This means that we should be able to constrain these paths during physical design and achieve a performance after routing that is close to our preroute predictions.

Next we check the logic for fault coverage. Table 14.23 shows that the ATPG software has inserted nearly 9000 faults, which is reasonable for the size of our design. Fault coverage is 96 percent. Most of the untested and tied faults arise from the BST logic exactly as we have already described in the Threegates example. If we had not completed this small test case first, we might not have noticed this. The aborted faults are almost all within the large flattened block, `v_1.u100`. If we assume the approximately 60 faults due to the BST logic are covered by a flush test, our fault coverage increases to 3740/3825 or 98 percent. To improve upon this figure, some, but not all, of the aborted faults can be detected by substantially increasing the backtrack limit from the default value of 30. To discover the reasons for the remaining aborted faults, we could use a controllability/observability program. If we wish to increase the fault coverage even further, we either need to change our test approach or change the design architecture. In our case we believe that we can probably obtain close to 99 percent stuck-at fault coverage with the existing architecture and thus we are ready to move on to physical design.

14.10 Summary

The primary reason to consider test early during ASIC design is that it can become very expensive if we do not. The important points we covered in this chapter are:

- Boundary scan
- Single stuck-at fault model
- Controllability and observability
- ATPG using test vectors
- BIST with no test vectors

14.11 Problems

* = Difficult, ** = Very difficult, *** = Extremely difficult

14.1 (Acronyms, 10 min.) Translate the following excerpt from a MOSIS report: “Chip description: DLX RISC ASIC with DFT, IEEE 1149 BST, and BIST using PRBS LFSR and MISR. Test results: compaction shorted words.”

14.2 (Economics of defect levels, 15 min.) You are the product manager for a new workstation. You use 10 similar ASICs as the key component in a computer that sells for $10,000 with a profit margin of 20 percent. You buy the ASICs for $10 each, and the shipping defect level is certified to be 0.1 percent by the ASIC vendor. You are having a problem with a large number of field failures, which you have traced to one of the ASICs. In the first nine months of shipment you have sold 49,500 computers, but 51 have failed in the field, 26 due to the ASIC. Finance estimates that all the field failures have cost at least $1 million in revenue and goodwill. You do not have the time, money, or capability to improve your incoming inspection or assembly tests. You estimate the product lifetime is another 18 months, in which time you will sell another 50,000 units at roughly the same price and profit margin. At an emergency meeting, the ASIC vendor's test engineer proposes to reduce the ASIC defect level to 0.01 percent immediately by improving the test program, but at a cost. You suggest a coffee break. With the information that you have, you have 15 minutes to estimate just how much extra you are prepared to pay for each ASIC.

14.3 (Defect level, 10 min.) In a series of experiments a customer of Zycad, which makes hardware fault-simulation accelerators, tested 10,000 parts from a lot with 30 percent yield. Each experiment used a different fraction of the test vector set. Fit the data in Table 14.24 to a model.

TABLE 14.24 Defect level as a function of fault coverage (Problem 14.3).

Fault coverage/%	Rejects	Defective parts	Defect level/%
50	6773	227	7
90	6877	133	3
99	6910	90	1
99.99	6997	3	0.01

14.4 (Test cost, 5 min.) Suppose, in the example of Section 14.1, reducing the bASIC defect level to 0.1 percent added an extra cost of $1 to each part. Now what is the best way to build the system?

14.5 (Defects, 5 min.) Finding defects in an ASIC is a hard problem. The average defect density for a submicron process is 1 cm^{-2} or less.

a. On average how many defects are there on a 1 cm chip?

b. If the average defect is 1 λ^2, and $\lambda = 0.25$ µm, what is defect area/chip area?

c. Estimate the ratio of needle volume to haystack volume and comment.

14.6 (Faults and nodes, 10 min.)

a. How many faults are there in a circuit with n nodes?

b. Considering fanout how many collapsed faults are there?

c. Estimate how many test cycles a fault simulator needs to find these faults.

d. With a 10 MHz clock, how long is a 100 k-gate test (with your estimates)?

e. Using a 100 MHz computer, how long does this fault simulation take? (Assume simulation time is four orders of magnitude slower than real time.)

14.7 (PRBS, 10 min.) What are the first three patterns for a 4-bit maximal-length LFSR, given a seed of '0001'? *Hint:* Is there more than one answer?

14.8 (Test time, 10 min.)

a. How long does a 16-bit shift-register test take at a clock speed of 1 MHz?

b. Estimate how long it takes to test a 64 k-bit static RAM using a walking 1's (or marching 1's) pattern.

14.9 (Test time, 10 min.) A modern production tester costs $5–10 million. This cost is depreciated over the life of the tester (usually five years in the United States due to Internal Revenue Service guidelines).

a. If the tester is in use 24 hours a day, 365 days a year, how much does 1 second of test time cost?

b. If, due to down time (maintenance, operator sick time and so on) a $10 million tester is actually in use 50 percent of the time for chip testing and test time is 2 seconds, how much does test add to the cost of an ASIC?

c. Suppose the ASIC die is 300 mils on a side, is fabricated on a 6-inch wafer whose fabrication cost is $1750, and the yield is 68 percent. What is the fraction of test cost to total die cost (fabrication plus test costs)? Assume that the number of die per wafer is equal to wafer area divided by chip area.

14.10 (Fault collapsing, 10 min.) Draw up tables to show how input and output faults collapse using gate collapsing for the following primitive logic gates: AND, OR, NAND, NOR, and EXOR (assume two-input logic cells in each case with inputs A, B and output F); a two-input MUX (inputs S0, S1, and SEL0; output F).

14.11 (Fault simulation, 15 min.) Mentor Graphic Corporation's QuickFault concurrent fault simulator uses a 12-state logic system with three logic values ('0', '1', 'X') and four strengths (strong = S, resistive = R, high impedance = Z, I = indeterminate). Complete Table 14.25 using D = detected fault, P = possibly detected fault, and '–' = undetected fault. Give two values, 1/2, for each cell: The first value is for the default fault model in which a tester cannot tell the difference between Z/S/R; the second value is for testers that can differentiate between Z and S/R. *Hint:* One line of the table has been completed as an example.

14.12 (Finding faults, 30 min.)

a. List all the possible stuck-at faults for the circuit in Figure 14.37 using node faults.

b. Find all of the equivalent fault classes using node collapsing.

c. List the prime faults.

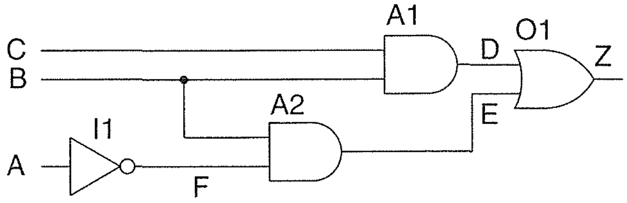
d. List all possible stuck-at faults using input and output faults (use A1.B and A2.B to distinguish between different inputs and outputs on the same net).

TABLE 14.25 The logic system used by Mentor Graphic Corporation's fault simulator, QuickFault (Problem 14.11).[1]

		Faulty circuit											
		0I	XI	1I	0Z	XZ	1Z	0R	XR	1R	0S	XS	1S
Good circuit	0I												
	XI												
	1I												
	0Z												
	XZ												
	1Z	–/P	–/P	–/P	–	–	–	–/D	–/D	–/D	–/D	–/D	–/D
	0R												
	XR												
	1R												
	0S												
	XS												
	1S												

[1] D=detected; P=possibly detected; '–'=undetected; x/y means x is the result of the default detection mechanism and y is the result when three-state detection is enabled (allowing the detection of the difference between Z and R/S strength).

FIGURE 14.37 An example circuit for fault collapsing (Problem 14.12).



e. List the fault-equivalence classes using gate collapsing.

f. List the prime faults.

14.13 (Blind faith, 10 min.) Consider the following code: `a = b && f(c)`. Verilog stops executing an expression as soon as it determines that the expression is false, whereas VeriFault does not. What effect does this have?

14.14 (Fault collapsing, 10 min.) Draw the Karnaugh maps including stuck-at faults for four-input NAND, AND, OR, and NOR gates.

14.15 (Fault dominance, 10 min.) If T_x is the set of test vectors that test for fault x and $T_b \subseteq T_a$, what can you say about faults a and b?

14.16 (*Fault dominance, 10 min.) Consider the network C = AND (A, B), D = NOT (B). List the PIs, POs, and faults under a pin-fault model. For each fault, state whether it is an equivalent fault, dominant fault, or dominated fault. Now consider this more formal definition of fault dominance: Fault a dominates b if and only if a and b are equivalent under the set of tests T for b. Two faults are equivalent under a test T if and only if the circuit response of the two faulty circuits is identical. *Hint:* Consider the fault at the input of the inverter very carefully.

14.17 (Japanese TVs, 20 min.) As an experiment a Japanese manufacturer decided not to perform any testing of its TVs before turning them on at the end of the production line. They achieved over a 90 percent turn-on rate. Build a cost model for this approach to testing. Make a one-page list of its advantages and disadvantages.

14.18 (Test costs, 20 min.) The CEO of an ASIC vendor called a meeting and asked the production manager to bring all wafers queued for rework. The CEO produced a hammer and smashed the several hundred wafers on the boardroom table. Construct a model around the following assumptions: 2 percent of wafers-in-process currently require rework after each of the 12 photo steps in the process, wafer cost is \$2,000, 30 percent of the wafer costs are in the photo steps; current process yield is 85 percent, 30 percent of the reworked wafers have to be scrapped. Explain why you were not as shocked by this episode as the production manager and how it helped you to explain to the CEO the need to add time to your ASIC design schedule to include design for test.

14.19 (ZyCAD RP, 10 min.) The ZyCAD Paradigm RP rapid prototyping system consists of a set of emulation boards. Each emulation board contains 18 Xilinx 3090 chips and 16 Xilinx 4010 chips. The Xilinx 4010 chips are mounted on eight daughterboards, and the 3090 chips are mounted directly on the motherboard. The Xilinx 4010 chips are used for logic block emulation and the Xilinx 3090 chips are used for crossbar routing. Each daughterboard has 288 I/O pins that are available to the crossbar chips for routing. Each Xilinx 4010 device has the capability to interface with any other 4010 device on the emulation board. The Xilinx 4010 devices have 400 Configurable Logic Blocks (CLBs) per device and 160 programmable I/O1s. Estimate the size of an ASIC that you could prototype with this system.

14.20 (IDDQ testing, 10 min.) In the **six-shorts-per-transistor fault model** for IDDQ testing we model six shorts per transistor. What are they?

14.21 (PRBS) Consider Table 14.26.

a. (15 min.) What is the autocorrelation function for a maximal-length pseudorandom binary sequence?

b. ** (30 min.) Suppose we apply a pseudorandom sequence to a linear system. What is its response?

c. *** (60 min.) Suppose we correlate this response with the original pseudorandom sequence delayed by *n* cycles. What is this correlation function?

TABLE 14.26 Autocorrelation of pseudorandom binary sequences (Problem 14.21).

	Delay (clock ticks)						
	0	1	2	3	4	5	6
	$Q2_t$	$Q2_{t-1}$	$Q2_{t-2}$	$Q2_{t-3}$	$Q2_{t-4}$	$Q2_{t-5}$	$Q2_{t-6}$
	1	0	1	0	0	1	1
	1	1	0	1	0	0	1
	1	1	1	0	1	0	0
	0	1	1	1	0	1	0
	0	0	1	1	1	0	1
	1	0	0	1	1	1	0
	0	1	0	0	1	1	1
Correlation with $Q2_t$	4	2	2	2	2	2	2

14.22 (Sentry, 20 min.) Write a Sentry test file to check the preset of a D flip-flop.

14.23 (Synthesis, 20 min.) Consider the following equations:

```
f1 = x1'x2' + y3; f2 = x1x2' + x1'x2 = y2; f3 = x1x2y2' + x1'x2' = y3
```

How many untestable stuck-at faults are there in this network? Suppose we simplify the logic to the following:

```
f1* = y2'; f2* = x1x2' + x1'x2
```

How many untestable stuck-at faults are there? *Hint:* If you are stuck, see [Bartlett et al., 1988; Brayton, Hachtel, and Sangiovanni-Vincentelli, 1990].

14.24 (Threegates, 30 min.) Recreate the Threegates example.

14.25 (LFSR) Determine the pattern sequence generated by the 4-bit LFSR shown in Figure 14.38. Use the same format as Table 14.10.

14.26 (BIST, 15 min.) Find the signature if the CUT of Figure 14.25 is $Z = A'B + AC$.

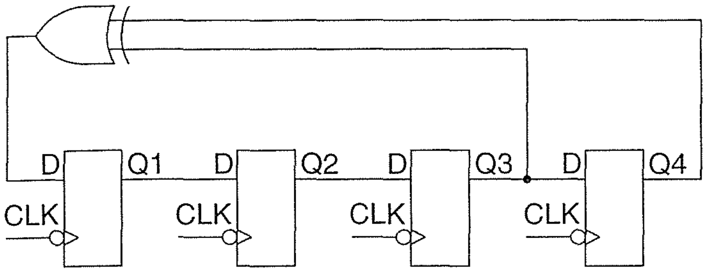


FIGURE 14.38 A 4-bit linear feedback shift register (LFSR) (Problem 14.25).

14.12 Bibliography

Books by Feugate and McIntyre [1988], Cheng and Agrawal [1989], and Fritzemeier, Nagle, and Hawkins [1989] contain explanations of basic testing terms and techniques. The book by Abramovici, Breuer, and Friedman [1990] is an advanced undergraduate and graduate-level review of test techniques. Needham's [1991] book reviews wafer and package testing. The text by Russell and Sayers [1989] is an undergraduate-level text with explanations of test algorithms. Turino's [1990] book covers a wide range of testing topics.

There are a number of books with collections of research papers on test, including works by Eichelberger, Lindblom, Waicukauski, and Williams [1991]; Lombardi and Sami [1987]; Williams [1986]; and Zobrist [1993]. Tsui's book contains a review of scan test and a large bibliography [1987]. The book by Ghosh, Devadas, and Newton [1992] describes test-generation algorithms for state machines at a level intended for CAD researchers. Bardell, McAnney, and Savir [1987] focus on pseudorandom BIST. A book by Yarmolik [1990] covers BIST and signature analysis; a second book by Yarmolik and Kachan [1993] concentrates on self-test. Books by Lavagno and Sangiovanni-Vincentelli [1993] and by Lee [1997] are advanced works on the integration of test synthesis and logic synthesis. The text by Jha and Kundu [1990] covers reliability in design. The book by Bhattacharya and Hayes [1990] covers modeling for testing (and includes a good description of the D and PODEM algorithms). There are alternative ASIC test techniques that we have not covered. For example, Chandra's paper describes the **CrossCheck** architecture for gate arrays [1993]. A book by Chakradhar, Agrawal, and Bushnell [1991] covers neural models for testing.

The major conferences in the area of test are the International Test Conference, known as the ITC (TK7874.I593, ISSN 0743-1686), the International Test Symposium (TK7874.I3274, ISBN depends on year), and the European Design and Test Conference (TK7888.4.E968, 1994: ISBN 0-8186-5410-4). The IEEE International Workshop on Memory Technology, Design, and Testing (TK7895.M4.I334) is a conference on memory testing. US DoD standard procedure

5012 of Mil-Std-883 sets requirements for simulation algorithms, fault collapsing, undetectable faults, potential detection, and detection strobing (see also *IEEE Design & Test Magazine*, Sept. 1993, pp. 68–79).

The IEEE has published a series of tutorials on test: *VLSI Support Technologies: Computer-Aided Design, Testing, and Packaging,* TK7874.T886, 1982; *VLSI Testing & Validation Techniques,* ISBN 0818606681, TK7874.T8855, 1985; *Test Generation for VLSI Chips,* ISBN 081868786X, TK7874.T8857, 1988.

The **Waveform and Vector Exchange Specification** (WAVES), IEEE Std 1029.1-1991 [IEEE 1029.1-1991], is a standard representation for digital stimulus and response for both design and test and allows digital stimulus and response information to be exchanged between different simulation and test tools. The syntax of WAVES is a subset of VHDL. WAVES was developed by the WAVES Analysis and Standardization Group (WASG). The WASG was jointly sponsored by the Automatic Test Program Generation (ATPG) subcommittee of the Standards Coordination Committee 20 (SCC20) and the Design Automation Standards Subcommittee (DASS) of the Computer Society.

14.13 References

Page numbers in brackets after the reference indicate the location in the chapter body.

Abramovici, M., M. A. Breuer, and A. D. Friedman. 1990. *Digital Systems Testing and Testable Design.* New York: W. H. Freeman, 653 p. ISBN 0-7167-8179-4. TK7874.A23. Introduction to testing and BIST. See also Breuer, M. A., and A. D. Friedman, 1976. *Diagnosis and Reliable Design of Digital Systems.* 2nd ed. Potomac, MD: Computer Science Press, ISBN 0-914894-57-9. TK7868.D5B73. [p. 800]

Agarwal, V. K., and A. S. F. Fung. 1981. "Multiple fault testing of large circuits by single fault test sets." *IEEE Transactions on Computing,* Vol. C-30, no. 11, pp. 855–865. [p. 740]

Bardell, P. H., W. H. McAnney, and J. Savir. 1987. *Built-In Test for VLSI: Pseudorandom Techniques.* New York: Wiley, 354 p. ISBN 0-471-62463-2. TK7874.B374. [p. 800]

Bartlett, K., et al. 1988. "Multilevel logic minimization using implicit don't cares," *IEEE Transactions on Computer-Aided Design,* Vol. CAD-7, no. 6, pp. 723–740. [p. 799]

Bhattacharya, D., and J. P. Hayes. 1990. *Hierarchical Modeling for VLSI Circuit Testing.* Boston: Kluwer, 159 p. ISBN 079239058X. TK7874.B484. Contains a good description of the D and PODEM algorithms. [p. 800]

Bleeker, H., P. v. d. Eijnden, and F. de Jong. 1993. *Boundary-Scan Test: A Practical Approach.* Boston: Kluwer, 225 p. ISBN 0-7923-9296-5. [p. 714]

Brayton, R. K., G. D. Hachtel, and A. L. Sangiovanni-Vincentelli. 1990. "Multilevel logic synthesis." *Proceedings of the IEEE,* Vol. 78, no. 2, pp. 264–300. [p. 799]

Butler, K. M., and M. R. Mercer. 1992. *Assessing Fault Model and Test Quality.* Norwell, MA: Kluwer, 125 p. ISBN 0-7923-9222-1. TK7874.B85. Introductory level discussion of test terminology, fault models and their limitations. Research-level discussion of the use of BDDs, ATPG, and controllability/observability. [p. 761]

Chandra, S., et al. 1993. "CrossCheck: an innovative testability solution." *IEEE Design & Test of Computers,* Vol. 10, no. 2, pp. 56–68. Describes a gate-array test architecture used by Sony, for example. [p. 800]

Chakradhar, S. T., V. D. Agrawal, and M. L. Bushnell. 1991. *Neural Models and Algorithms for Digital Testing.* Boston: Kluwer, 184 p. ISBN 0792391659. TK7868.L6.C44. [p. 800]

Cheng, K.-T., and V. D. Agrawal. 1989. *Unified Methods for VLSI Simulation and Test Generation.* Norwell, MA: Kluwer, 148 p. ISBN 0-7923-9025-3. TK7874.C525. 377 references. The first three chapters give a good introduction to fault simulation and test vector generation. [p. 800]

Eichelberger, E. B., E. Lindblom, J. A. Waicukauski, and T. W. Williams. 1991. *Structured Logic Testing.* Englewood Cliffs, NJ: Prentice-Hall, 183 p. ISBN 0-13-8536805. TK7868.L6S78. Includes material printed in 19 articles by the authors from 1987 to 1989. [p. 800]

Feugate Jr., R. J., and S. M. McIntyre. 1988. *Introduction to VLSI Testing.* Englewood Cliffs, NJ: Prentice-Hall, 226 p. ISBN 0134988663. TK7874 .F48. Chapters on: Automated Testing Overview; IC Fabrication and Device Specifications; Testing Integrated Circuits: Parametric Tests; Functional Tests; Example of a Functional Test Program; Characterization testing; Developing Test Patterns; Special Testing Problems: Memories; Special Testing Problems: Microcontrollers; Design for Testability; LSTL Language Summary; Example of a Production Test program; The D-Algorithm. [p. 800]

Fujiwara, H., and T. Shimono. 1983. "On the acceleration of test generation algorithms." *IEEE Transactions on Computers,* Vol. C-32, no. 12, pp. 1137–1144. Describes the FAN ATPG algorithm. [p. 761]

Fritzemeier, R. R., H. T. Nagle, and C. F. Hawkins. 1989. "Fundamentals of testability—a tutorial." *IEEE Transactions on Industrial Electronics,* Vol. 36, no. 2, pp. 117–128. 54 refs. A review of testing, failure mechanisms, fault models, fault simulation, testability analysis, and test-generation methods for CMOS VLSI circuits. [p. 800]

Ghosh, A., S. Devadas, and A. R. Newton. 1992. *Sequential Logic Testing and Verification.* Norwell, MA: Kluwer, 214 p. ISBN 0-7923-91888. TK7868.L6G47. Describes test generation algorithms for state machines at a level intended for CAD researchers. [p. 800]

Goel, P. 1981. "An implicit enumeration algorithm to generate tests for combinational logic circuits." *IEEE Transactions on Computers,* Vol. C-30, no. 3, pp. 215–222. [p. 759]

Goldstein, L. H. 1979. "Controllability/observability analysis of digital circuits." *IEEE Transactions on Circuits and Systems,* Vol. CAS-26, no. 9, pp. 685–693. Describes SCOAP measures. [p. 761]

Golomb, S. W., et al. 1982. *Shift Register Sequences.* 2nd ed. Laguna Hills, CA: Aegean Park Press, 247 p. ISBN 0-89412-048-4. QA267.5.S4 G6. See also: Golomb, S. W., *Shift Register Sequences* (with portions co-authored by L. R. Welch, R. M. Goldstein and A. W. Hales). San Francisco: Holden-Day (1967), 224 p. QA267.5.S4 G6. The second edition has a long bibliography. [p. 771]

Gulati, R. K., and C. F. Hawkins. (Ed.). 1993. *IDDQ Testing of VLSI Circuits.* Boston: Kluwer, 120 p. ISBN 0792393155. TK7874.I3223. [p. 743]

Hughes, J. L. A., and E. J. McCluskey. 1986. "Multiple stuck-at fault coverage of single stuck-at fault test sets." In *Proceedings of the IEEE International Test Conference,* pp. 368–374. [p. 740]

IEEE 1029.1. 1991. IEEE Standard for Waveform and Vector Exchange (WAVES) (ANSI). 96 p. IEEE reference numbers: [1-55937-195-1] [SH15032-NYF]. [p. 801]

IEEE 1149.1b. 1994. IEEE Std 1149.1-1990 Access Port and Boundary-Scan Architecture. 176 p. The first part of this updated standard includes supplement 1149.1a-1993. IEEE reference numbers: [1-55937-350-4] [SH16626-NYK] The second part of this standard includes 1149.1b-1994 Supplement to IEEE Std 1149.1-1990, IEEE Standard Test Access Port and Boundary-Scan Architecture (ANSI) (available separately). 80 p. IEEE reference numbers: [1-55937-497-7] [SH94256-NYK]. [p. 714, p. 716, p. 718, p. 718, p. 735, p. 735]

Jha, N. K., and S. Kundu. 1990. *Testing and Reliable Design of CMOS Circuits.* Boston: Kluwer, 231 p. ISBN 0792390563. TK7871.99.M44.J49. [p. 800]

Lavagno, L., and A. Sangiovanni-Vincentelli. 1993. *Algorithms for Synthesis and Testing of Asynchronous Circuits.* Boston: Kluwer, 339 p. ISBN 0792393643. TK7888.4.L38. [p. 800]

Lee, M. T.-C. 1997. *High-Level Test Synthesis of Digital VLSI Circuits.* Boston: Artech House, ISBN 0890069077. TK7874.75.L44. [p. 800]

Lombardi, F., and M. Sami (Ed.). 1987. *Testing and Diagnosis of VLSI and ULSI.* Norwell, MA: Kluwer, 533 p. ISBN 90-247-3794-X. TK7874.N345. A series of 20 research-level papers presented at a NATO advanced Study Institute. Contents: Trends in Design for Testability; Statistical Testing; Fault Models; Fault Detection and Design for Testability of CMOS Logic Circuits; Parallel Computer Systems Testing and Integration; Analog Fault Diagnosis; Spectral Techniques for Digital Testing; Logic Verification, Testing and Their Relationships to Logic Synthesis; Proving the Next Stage from Simulation; Petri Nets and Their Relation to Design Validation and Testing; Functional Test of ASICs and Boards; Fault Simulation Techniques — Theory and Practical Examples; Threshold-Value Simulation and Test Generation; Behavioral Testing of Programmable Systems; Testing of Processing Arrays; Old and New Approaches for the Repair of Redundant Memories; Reconfiguration of Orthogonal Arrays by Front Deletion; Device Testing and SEM Testing Tools; Advances in Electron Beam Testing. [p. 800]

Maunder, C. M., and R. E. Tulloss (Ed.). 1990. *The Test Access Port and Boundary-Scan Architecture.* Washington, DC: IEEE Computer Society Press. ISBN 0-8186-9070-4. TK867.T39. [p. 714]

Needham, W. M. 1991. *Designer's Guide to Testable ASIC Devices.* New York: Van Nostrand Reinhold, 284 p. ISBN 0-442-00221-1. TK7874.N385. Practical review of wafer and package testing. Includes summary of features and test file formats used by logic testers. [p. 800]

Parker, K. P. 1992. *The Boundary-Scan Handbook.* Norwell, MA: Kluwer, 262 p. ISBN 0-7923-9270-1. TK7868.P7 P3. Describes BSDL. [p. 714]

Rajsuman, R. 1994. *Iddq Testing for CMOS VLSI.* Boston: Artech House, 193 p. ISBN 0-89006-726-0. TK7871.99.M44R35. [p. 743]

Rao, G. K. 1993. *Multilevel Interconnect Technology.* New York: McGraw-Hill. ISBN 0-07-051224-8. Covers the design of a multilevel interconnect process, and manufacturing and reliability issues. [p. 736]

Roth, J. P. 1966. "Diagnosis of automata failures: A calculus and a method." *IBM Journal of Research and Development,* Vol. 10, no. 4, pp. 278–291. Describes the D-calculus and the D-algorithm. [p. 755]

Russell, G., and I. L. Sayers. 1989. *Advanced Simulation and Test Methodologies for VLSI Design.* London: Van Nostrand Reinhold (International), 378 p. ISBN 0-7476-0001-5. TK7874.R89. Good explanations with a simple example of the D-algorithm. [p. 800]

Sabnis, A. G. (Ed.). 1990. *VLSI Reliability.* San Diego: Academic Press. ISBN 0-12-234122-8. Covers ESD, electromigration, packaging issues, quality assurance, failure analysis, radiation damage. [p. 736]

Scheiber, S.F. 1995. *Building a Successful Board-Test Strategy.* Boston: Butterworth–Heineman, 286 p. ISBN 0-7506-9432-7. TK7868.P7S33. Practical description from a management point of view of board-level testing. [p. 714]

Schulz, M. H., E. Trischler, and T. M. Sarfert. 1988. "SOCRATES: a highly efficient automatic test pattern generation system." *IEEE Transactions on Computer-Aided Design,* Vol. 7, no. 1, pp. 126–137. [p. 761]

Turino, J. 1990. *Design to Test—A Definitive Guide for Electronic Design, Manufacture and Service.* 2nd ed. New York: Van Nostrand Reinhold, 368 p. ISBN 0-442-00170-3.. TK7874.T83. A small encyclopedia of testing. Includes a general introduction to testability, and guidelines for: system-level, analog, and general circuit testing; board-level guidelines, boundary scan, built-in test, testability buses, mechanical issues, surface-mount technology, test software, documentation, implementation, ad-hoc test techniques and strategies, testability checklists, and a testability rating system. [p. 800]

Tsui, F. F. 1987. *LSI/VLSI Testability Design.* New York: McGraw-Hill, 700 p. ISBN 0-07-065341-0. TK7874.T78. Extensive review of scan-test techniques. Approximately 100-page bibliography of papers published on test from 1962–1986. [p. 800]

Williams, T. W. (Ed.). 1986. *VLSI Testing.* Amsterdam: Elsevier Science, 275 p. ISBN 0-444-87895-5 (part of set 0-444-87890-4). TK7874.V5666. Seven papers on fault modeling, test generation, and fault simulation, testable PLA designs, design for testability, memory testing, semiconductor test equipment, and board level test equipment. [p. 800]

Yarmolik, V. N. 1990. *Fault Diagnosis of Digital Circuits.* New York: Wiley. Translated from Russian text. Covers D-algorithm, LSSD, random and pseudorandom testing and analysis, and signature analysis. [p. 800]

Yarmolik, V. N., and I. V. Kachan. 1993. *Self-Testing VLSI Design.* New York: Elsevier, 345 p. ISBN 0-444-89640-6. TK7874.I16. Extensive reference on pseudorandom testing techniques. Includes description of pseudorandom sequence generators and polynomials. [p. 800]

Zobrist, G.W. (Ed.). 1993. *VLSI Fault Modeling and Testing Techniques.* Norwood, NJ: Ablex, 199 p. ISBN 0-89391-781-8. TK7874.V5625. Includes six research-level papers on physical fault modeling, testing of CMOS open faults, testing bridging faults, BIST for PLAs, design for testability, and synthesis methods for testable circuits. [p. 800]

ASIC CONSTRUCTION 15

A town planner works out the number, types, and sizes of buildings in a development project. An architect designs each building, including the arrangement of the rooms in each building. Then a builder carries out the construction according to the architect's drawings. Electrical wiring is one of the last steps in the construction of each building. The physical design of ASICs is normally divided into *system partitioning*, *floorplanning*, *placement*, and *routing*. A microelectronic system is the town and the ASICs are the buildings. System partitioning corresponds to town planning, ASIC floorplanning is the architect's job, placement is done by the builder, and the routing is done by the electrician. We shall design most, but not all, ASICs using these design steps.

15.1 Physical Design

Figure 15.1 shows part of the design flow, the physical design steps, for an ASIC (omitting simulation, test, and other logical design steps that have already been covered). Some of the steps in Figure 15.1 might be performed in a different order from that shown. For example, we might, depending on the size of the system, perform system partitioning before we do any design entry or synthesis. There may be some iteration between the different steps too.

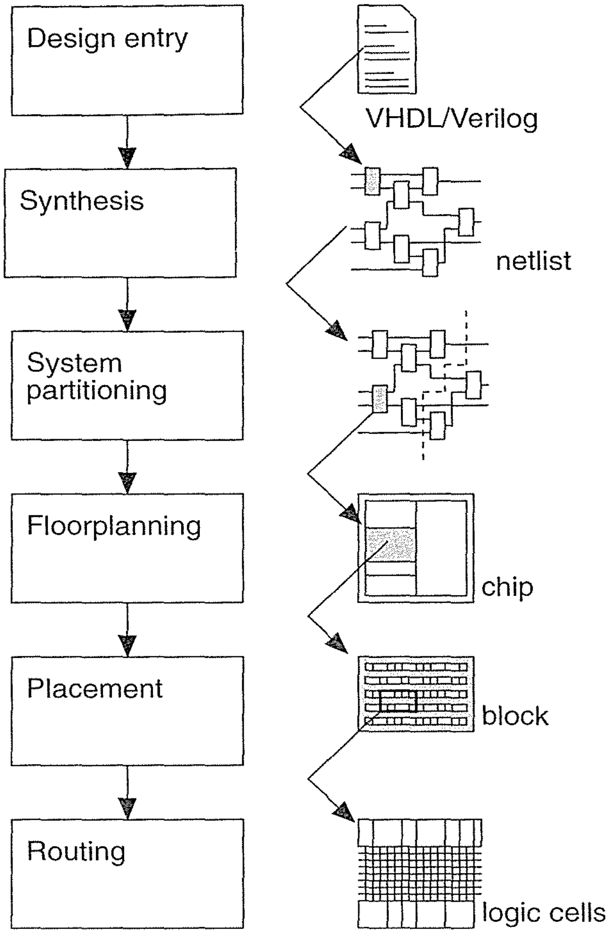


FIGURE 15.1 Part of an ASIC design flow showing the system partitioning, floorplanning, placement, and routing steps. These steps may be performed in a slightly different order, iterated or omitted depending on the type and size of the system and its ASICs. As the focus shifts from logic to interconnect, floorplanning assumes an increasingly important role. Each of the steps shown in the figure must be performed and each depends on the previous step. However, the trend is toward completing these steps in a parallel fashion and iterating, rather than in a sequential manner.

We must first apply **system partitioning** to divide a microelectronics system into separate ASICs. In **floorplanning** we estimate sizes and set the initial relative locations of the various blocks in our ASIC (sometimes we also call this **chip planning**). At the same time we allocate space for clock and power wiring and decide on the location of the I/O and power pads. **Placement** defines the location of the logic cells within the flexible blocks and sets aside space for the interconnect to each logic cell. Placement for a gate-array or standard-cell design assigns each logic cell to a position in a row. For an FPGA, placement chooses which of the fixed logic resources on the chip are used for which logic cells. Floorplanning and placement are closely related and are sometimes combined in a single CAD tool. **Routing** makes the connections between logic cells. Routing is a hard problem by itself and is normally split into two distinct steps, called global and local routing. **Global routing** determines where the interconnections between the placed logic cells and blocks will be situated. Only the routes to be used by the interconnections are decided in this step, not the actual locations of the interconnections within the wiring areas. Global routing is sometimes called **loose routing** for this reason. **Local routing** joins the logic cells with interconnections. Information on which interconnection areas to use comes from the global router. Only at this stage of layout do we finally decide on the width, mask layer, and exact location of the interconnections. Local routing is also known as **detailed routing**.

15.2 CAD Tools

In order to develop a CAD tool it is necessary to convert each of the physical design steps to a problem with well-defined goals and objectives. The **goals** for each physical design step are the things we must achieve. The **objectives** for each step are things we would like to meet on the way to achieving the goals. Some examples of goals and objectives for each of the ASIC physical design steps are as follows:

System partitioning:

- Goal. Partition a system into a number of ASICs.
- Objectives. Minimize the number of external connections between the ASICs. Keep each ASIC smaller than a maximum size.

Floorplanning:

- Goal. Calculate the sizes of all the blocks and assign them locations.
- Objective. Keep the highly connected blocks physically close to each other.

Placement:

- Goal. Assign the interconnect areas and the location of all the logic cells within the flexible blocks.
- Objectives. Minimize the ASIC area and the interconnect density.

Global routing:

- Goal. Determine the location of all the interconnect.
- Objective. Minimize the total interconnect area used.

Detailed routing:

- Goal. Completely route all the interconnect on the chip.
- Objective. Minimize the total interconnect length used.

There is no magic recipe involved in the choice of the ASIC physical design steps. These steps have been chosen simply because, as tools and techniques have developed historically, these steps proved to be the easiest way to split up the larger problem of ASIC physical design. The boundaries between the steps are not cast in stone. For example, floorplanning and placement are often thought of as one step and in some tools placement and routing are performed together.

15.2.1 Methods and Algorithms

A CAD tool needs **methods** or **algorithms** to generate a solution to each problem using a reasonable amount of computer time. Often there is no best solution possible to a particular problem, and the tools must use **heuristic algorithms**, or rules of thumb, to try and find a good solution. The term *algorithm* is usually reserved for a method that always gives a solution.

We need to know how practical any algorithm is. We say the **complexity** of an algorithm is $\mathbf{O}\,(\boldsymbol{f}\,(\boldsymbol{n}))$ (read as **order** $\boldsymbol{f}\,(\boldsymbol{n})$) if there are constants k and n_0 so that the running time of the algorithm $\mathrm{T}\,(n)$ is less than $\mathrm{k}f(n)$ for all $n > \mathrm{n}_0$ [Sedgewick, 1988]. Here n is a measure of the size of the problem (number of transistors, number of wires, and so on). In ASIC design n is usually very large. We have to be careful, though. The notation does not specify the units of time. An algorithm that is $\mathrm{O}\,(n^2)$ nanoseconds might be better than an algorithm that is $\mathrm{O}\,(n)$ seconds, for quite large values of n. The notation $\mathrm{O}\,(n)$ refers to an upper limit on the running time of the algorithm. A practical example may take less running time—it is just that we cannot prove it. We also have to be careful of the constants k and n_0. They can hide overhead present in the implementation and may be large enough to mask the dependence on n, up to large values of n. The function $f(n)$ is usually one of the following kinds:

- $f(n)$ = constant. The algorithm is **constant in time**. In this case, steps of the algorithm are repeated once or just a few times. It would be nice if our algorithms had this property, but it does not usually happen in ASIC design.
- $f(n) = \log n$. The algorithm is **logarithmic in time**. This usually happens when a big problem is (possibly recursively) transformed into a smaller one.
- $f(n) = n$. The algorithm is **linear in time**. This is a good situation for an ASIC algorithm that works with n objects.
- $f(n) = n \log n$. This type of algorithm arises when a large problem is split into a number of smaller problems, each solved independently.
- $f(n) = n^2$. The algorithm is **quadratic in time** and usually only practical for small ASIC problems.

If the time it takes to solve a problem increases with the size of the problem at a rate that is polynomial but faster than quadratic (or worse in an exponential fashion), it is usually not appropriate for ASIC design. Even after subdividing the ASIC physical design problem into smaller steps, each of the steps still results in problems that are hard to solve automatically. In fact, each of the ASIC physical design steps, in general, belongs to a class of mathematical problems known as **NP-complete** problems. This means that it is unlikely we can find an algorithm to solve the problem exactly in polynomial time.

Suppose we find a practical method to solve our problem, even if we can find a solution we now have a dilemma. How shall we know if we have a good solution if, because the problem is NP-complete, we cannot find the optimum or best solution to which to compare it? We need to know how close we are to the optimum solution to a problem, even if that optimum solution cannot be found exactly. We need to make a quantitative **measurement** of the quality of the solution that we are able to find. Often we combine several parameters or **metrics** that measure our goals and objectives into a **measurement function** or **objective function**. If we are minimizing the measurement function, it is a **cost function**. If we are maximizing the measurement function, we call the function a **gain function** (sometimes just **gain**).

Now we are ready to solve each of the ASIC physical design steps with the following items in hand: a set of goals and objectives, a way to measure the goals and objectives, and an algorithm or method to find a solution that meets the goals and objectives. As designers attempt to achieve a desired ASIC performance they make a continuous trade-off between speed, area, power, and several other factors. Presently CAD tools are not smart enough to be able to do this alone. In fact, current CAD tools are only capable of finding a solution subject to a few, very simple, objectives.

15.3 System Partitioning

Microelectronic systems typically consist of many functional blocks. If a functional block is too large to fit in one ASIC, we may have to split, or **partition**, the function into pieces using goals and objectives that we need to specify. For example, we might want to minimize the number of pins for each ASIC to minimize package cost. We can use CAD tools to help us with this type of system partitioning.

Figure 15.2 shows the system diagram of the Sun Microsystems SPARCstation 1. The system is partitioned as follows; the numbers refer to the labels in Figure 15.2. (See Section 1.3, "Case Study" for the sources of infomation in this section.)

- Nine custom ASICs (1–9)
- Memory subsystems (SIMMs, single-in-line memory modules): CPU cache (10), RAM (11), memory cache (12, 13)
- Six ASSPs (application-specific standard products) for I/O (14–19)
- An ASSP for time of day (20)
- An EPROM (21)
- Video memory subsystem (22)
- One analog/digital ASSP DAC (digital-to-analog converter) (23)

Table 15.1 shows the details of the nine custom ASICs used in the SPARCstation 1. Some of the partitioning of the system shown in Figure 15.2 is determined by whether to use ASSPs or custom ASICs. Some of these design decisions are based on intangible issues: time to market, previous experience with a technology, the ability to reuse part of a design from a previous product. No CAD tools can help with such decisions. The goals and objectives are too poorly defined and finding a way to measure these factors is very difficult. CAD tools cannot

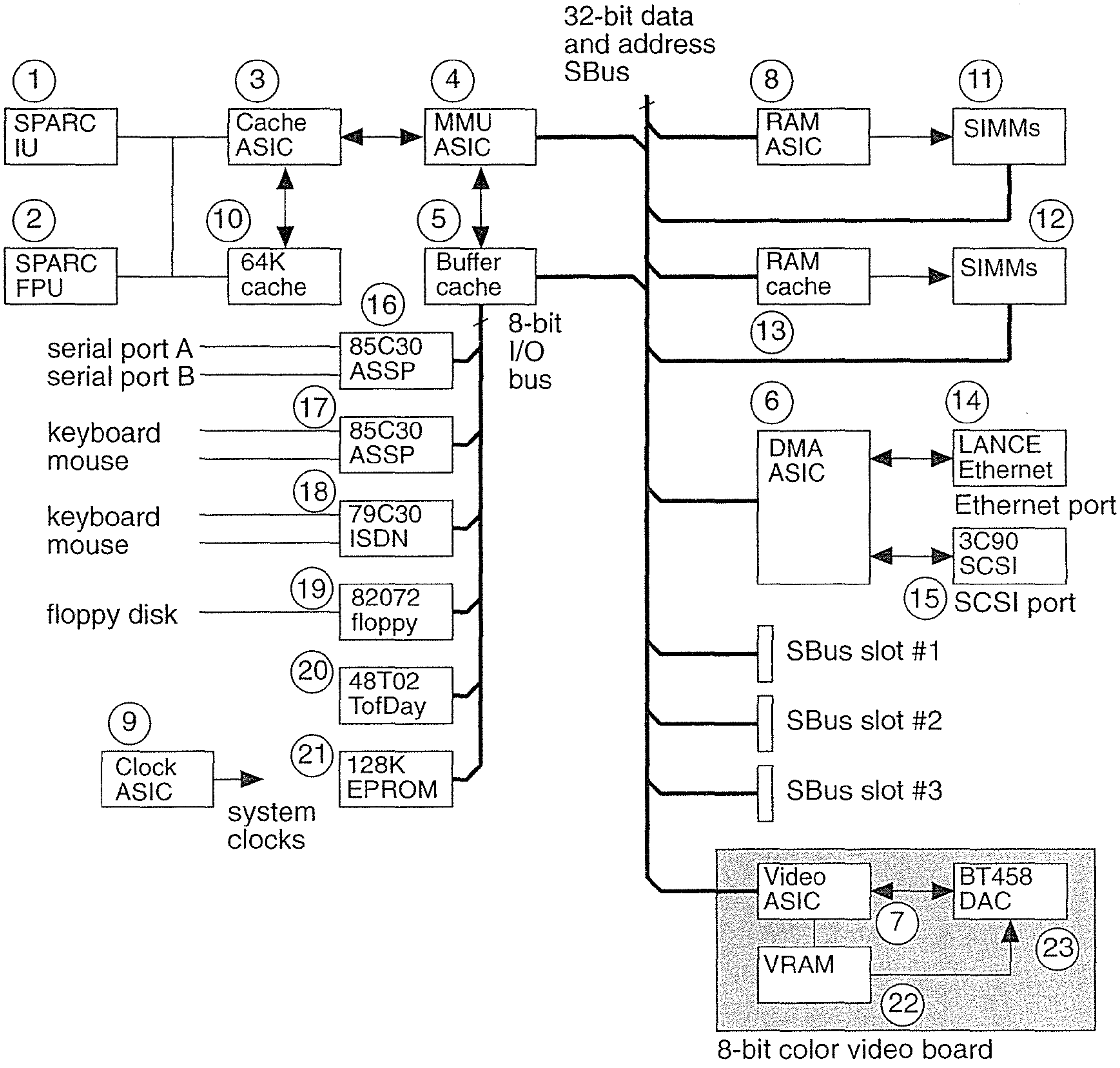


FIGURE 15.2 The Sun Microsystems SPARCstation 1 system block diagram. The acronyms for the various ASICs are listed in Table 15.1.

answer a question such as: "What is the cheapest way to build my system?" but can help the designer answer the question: "How do I split this circuit into pieces that will fit on a chip?" Table 15.2 shows the partitioning of the SPARCstation 10 so you can compare it to the SPARCstation 1. Notice that the gate counts of nearly all of the SPARCstation 10 ASICs have increased by a factor of 10, but the pin counts have increased by a smaller factor.

TABLE 15.1 System partitioning for the Sun Microsystems SPARCstation 1.

	SPARCstation 1 ASIC	Gates /k-gate	Pins	Package	Type
1	SPARC IU (integer unit)	20	179	PGA	CBIC
2	SPARC FPU (floating-point unit)	50	144	PGA	FC
3	Cache controller	9	160	PQFP	GA
4	MMU (memory-management unit)	5	120	PQFP	GA
5	Data buffer	3	120	PQFP	GA
6	DMA (direct memory access) controller	9	120	PQFP	GA
7	Video controller/data buffer	4	120	PQFP	GA
8	RAM controller	1	100	PQFP	GA
9	Clock generator	1	44	PLCC	GA

Abbreviations:

PGA = pin-grid array	CBIC = LSI Logic cell-based ASIC
PQFP = plastic quad flat pack	GA = LSI Logic channelless gate array
PLCC = plastic leaded chip carrier	FC = full custom

15.4 Estimating ASIC Size

Table 15.3 shows some useful numbers for estimating ASIC die size. Suppose we wish to estimate the die size of a 40 k-gate ASIC in a 0.35 μm gate array, three-level metal process with 166 I/O pads. For this ASIC the minimum feature size is 0.35 μm. Thus λ (one-half the minimum feature size) = 0.35 μm/2 = 0.175 μm. Using our data and Table 15.3, we can derive the following information. We know that 0.35 μm standard-cell density is roughly 5×10^{-4}gate/λ^2. From this we can calculate the gate density for a 0.35 μm gate array:

$$\text{gate density} = 0.35\ \mu\text{m standard-cell density} \times (0.8 \text{ to } 0.9)$$

$$= 4 \times 10^{-4} \text{ to } 4.5 \times 10^{-4}\ \text{gate}/\lambda^2. \qquad (15.1)$$

This gives the core size (logic and routing only) as

$$(4 \times 10^4 \text{ gates/gate density}) \times \text{routing factor} \times (1/\text{gate-array utilization})$$

$$= 4 \times 10^4/(4 \times 10^{-4} \text{ to } 4.5 \times 10^{-4}) \times (1 \text{ to } 2) \times 1/(0.8 \text{ to } 0.9) = 10^8 \text{ to } 2.5 \times 10^8\ \lambda^2$$

$$= 4840 \text{ to } 11{,}900\ \text{mil}^2. \qquad (15.2)$$

TABLE 15.2 System partitioning for the Sun Microsystems SPARCstation 10.

	SPARCstation 10 ASIC	Gates	Pins	Package	Type
1	SuperSPARC Superscalar SPARC	3 M-transistors	293	PGA	FC
2	SuperCache cache controller	2 M-transistors	369	PGA	FC
3	EMC memory control	40 k-gate	299	PGA	GA
4	MSI MBus–SBus interface	40 k-gate	223	PGA	GA
5	DMA2 Ethernet, SCSI, parallel port	30 k-gate	160	PQFP	GA
6	SEC SBus to 8-bit bus	20 k-gate	160	PQFP	GA
7	DBRI dual ISDN interface	72 k-gate	132	PQFP	GA
8	MMCodec stereo codec	32 k-gate	44	PLCC	FC

Abbreviations:

PGA = pin-grid array	GA = channelless gate array
PQFP = plastic quad flat pack	FC = full custom
PLCC = plastic leaded chip carrier	

We shall need to add $(0.175/0.5) \times 2 \times (15 \text{ to } 20) = 10.5$ to 21 mil (per side) for the pad heights (we included the effects of scaling in this calculation). With a pad pitch of 5 mil and roughly $166/4 = 42$ I/Os per side (not counting any power pads), we need a die at least $5 \times 42 = 210$ mil on a side for the I/Os. Thus the die size must be at least $210 \times 210 = 4.4 \times 10^4 \text{ mil}^2$ to fit 166 I/Os. Of this die area only $1.19 \times 10^4 / (4.4 \times 10^4) = 27\ \%$ (at most) is used by the core logic. This is a severely pad-limited design and we need to rethink the partitioning of this system.

Table 15.4 shows some typical areas for datapath elements. You would use many of these datapath elements in floating-point arithmetic (these elements are large—you should not use floating-point arithmetic unless you have to):

- A leading-one detector with barrel shifter normalizes a mantissa.
- A priority encoder corrects exponents due to mantissa normalization.
- A denormalizing barrel shifter aligns mantissas.
- A normalizing barrel shifter with a leading-one detector normalizes mantissa subtraction.

Most datapath elements have an area per bit that depends on the number of bits in the datapath (the datapath width). Sometimes this dependency is linear (for the multipliers and the barrel shifter, for example); in other elements it depends on the logarithm (to base 2) of the datapath width (the leading one, all ones, and zero detectors, for example). In some elements you might expect there to be a dependency on datapath width, but it is small (the comparators are an example).

TABLE 15.3 Some useful numbers for ASIC estimates, normalized to a 1 μm technology unless noted.

Parameter	Typical value	Comment[1]	Scaling
Lambda, λ	0.5 μm = 0.5 (minimum feature size)	In a 1 μm technology, $\lambda \approx 0.5$ μm.	NA
CAD pitch	1 micron = 10^{-6} m = 1 μm = minimum feature size	Not to be confused with minimum CAD grid size (which is usually less than 0.01 μm).	λ
Effective gate length	0.25 to 1.0 μm	Less than drawn gate length, usually by about 10 percent.	λ
I/O-pad width (pitch)	5 to 10 mil = 125 to 250 μm	For a 1 μm technology, 2LM ($\lambda = 0.5$ μm). Scales less than linearly with λ.	λ
I/O-pad height	15 to 20 mil = 375 to 500 μm	For a 1 μm technology, 2LM ($\lambda = 0.5$ μm). Scales approximately linearly with λ.	λ
Large die	1000 mil/side, 10^6 mil^2	Approximately constant	1
Small die	100 mil/side, 10^4 mil^2	Approximately constant	1
Standard-cell density	1.5×10^{-3} gate/μm^2 = 1.0 gate/mil^2	For 1μm, 2LM, library = 4×10^{-4} gate/λ^2 (independent of scaling).	$1/\lambda^2$
Standard-cell density	8×10^{-3} gate/μm^2 = 5.0 gate/mil^2	For 0.5 μm, 3LM, library = 5×10^{-4} gate/λ^2 (independent of scaling).	$1/\lambda^2$
Gate-array utilization	60 to 80 %	For 2LM, approximately constant	1
	80 to 90 %	For 3LM, approximately constant	1
Gate-array density	(0.8 to 0.9) × standard cell density	For the same process as standard cells	1
Standard-cell routing factor = (cell area + route area)/cell area	1.5 to 2.5 (2LM) 1.0 to 2.0 (3LM)	Approximately constant	1
Package cost	\$0.01/pin, "penny per pin"	Varies widely, figure is for low-cost plastic package, approximately constant	1
Wafer cost	\$1 k to \$5 k average \$2 k	Varies widely, figure is for a mature, 2LM CMOS process, approximately constant	1

[1] 2LM = two-level metal; 3LM = three-level metal.

The area estimates given in Table 15.4 can be misleading. The exact size of an adder, for example, depends on the architecture: carry-save, carry-select, carry-lookahead, or ripple-carry (which depends on the speed you require). These area figures also exclude the routing between datapath elements, which is difficult to predict—it will depend on the number and size of the datapath elements, their type, and how much logic is random and how much is datapath.

TABLE 15.4 Area estimates for datapath functions.[1]

Datapath function	**Area per bit/λ^2**	**Area/λ^2 (32-bit)**	**Area/λ^2 (64-bit)**
High-speed comparator (4–32 bit)	24,000	7.7E+05	1.5E+06
High-speed comparator (32–128 bit)	28,800	9.2E+05	1.8E+06
Leading-one detector (*n*-bit)	$7200 \log_2 n$	1.2E+06	2.8E+06
All-ones detector (*n*-bit)	$6000 + 800 \log_2 n$	3.2E+05	6.9E+05
Priority encoder (*n*-bit)	$19{,}000 + 1400 \log_2 (n-2)$	8.4E+05	1.8E+06
Zero detector (*n*-bit)	$5500 + 800 \log_2 n$	3.0E+05	6.6E+05
Barrel shifter/rotator (*n*- by *m*-bit)	$19{,}000 + 1000n + 1600m$	3.4E+06	1.2E+07
Carry-save adder	24,000	7.7E+05	1.5E+06
Digital delay line (*n* delay stages, *t* output taps)	$12{,}000 + 6000n + 8400t$	1.5E+07	6.0E+07
Synchronous FIFO (*n*-bit)	$34{,}000 + 9600n$	1.1E+07	4.1E+07
Multiplier-accumulator (*n*-bit)	$190{,}000 + 18{,}000n$	2.4E+07	8.5E+07
Unsigned multiplier (*n*- by *m*-bit)	$54{,}000 + 18{,}000(n-2)$	1.9E+07	7.4E+07
2:1 MUX	7200	2.3E+05	4.6E+05
8:1 MUX	29,000	9.2E+05	1.8E+06
Low-speed adder	28,000	8.8E+05	1.8E+06
2901 ALU	41,000	1.3E+06	2.6E+06
Low-speed adder/subtracter	30,000	9.6E+05	1.9E+06
Sync. up–down counter with sync. load and clear	43,000	1.4E+06	2.8E+06
Low-speed decrementer	14,000	4.6E+05	9.2E+05
Low-speed incrementer	14,000	4.6E+05	9.2E+05
Low-speed incrementer/decrementer	20,000	6.5E+05	1.3E+06

[1] Area estimates are for a two-level metal (2 LM) process. Areas for a three-level metal (3LM) process are approximately 0.75 to 1.0 times these figures.

Figure 15.3(a) shows the typical size of SRAM constructed on an ASIC. These figures are based on the use of a RAM compiler (as opposed to building memory from flip-flops or latches) using a standard CMOS ASIC process, typically using a six-transistor cell. The actual size of a memory will depend on (1) the required access time, (2) the use of synchronous or asynchronous read or write, (3) the number and type of ports (read–write), (4) the use of special design rules, (5) the number of interconnect layers available, (6) the RAM architecture (number of devices in RAM cell), and (7) the process technology (active pull-up devices or pull-up resistors).

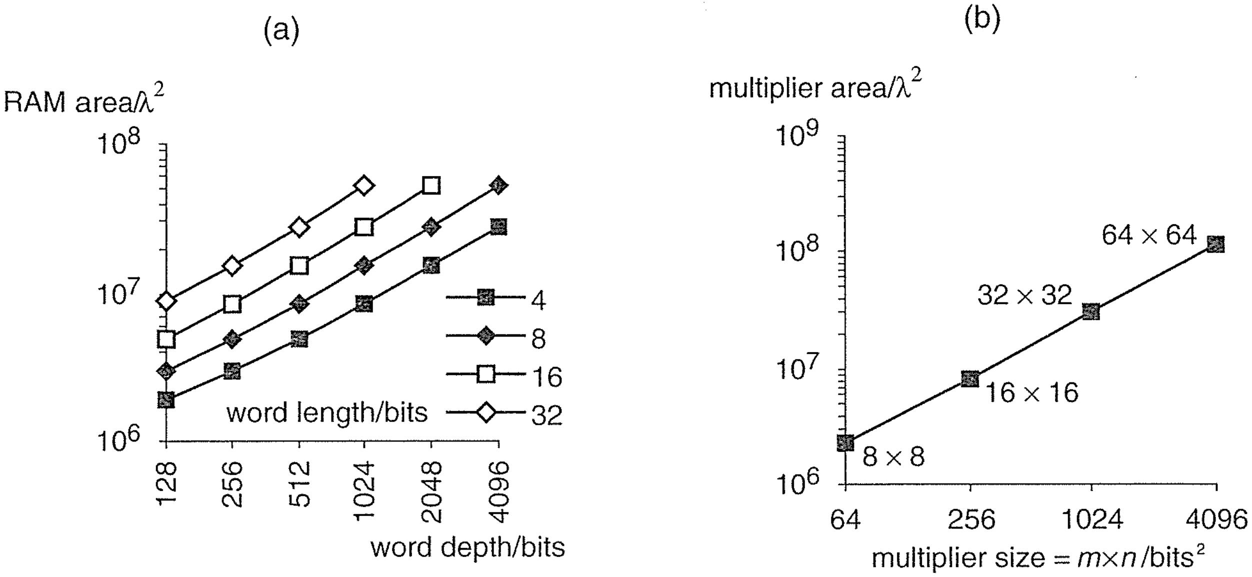


FIGURE 15.3 (a) ASIC memory size. These figures are for static RAM constructed using compilers in a 2LM ASIC process, but with no special memory design rules. The actual area of a RAM will depend on the speed and number of read–write ports. (b) Multiplier size for a 2LM process. The actual area will depend on the multiplier architecture and speed.

The maximum size of SRAM in Figure 15.3(a) is 32 k-bit, which occupies approximately $6.0 \times 10^7 \lambda^2$. In a 0.5 μm process (with $\lambda = 0.25\,\mu m$), the area of a 32 k-bit SRAM is $6.0 \times 10^7 \times 0.25 \times 0.25 = 3.75 \times 10^6\,\mu m^2$ (or about 2 mm on a side—a large piece of silicon). If you need an SRAM that is larger than this, you probably need to consult with your ASIC vendor to determine the best way to implement a large on-chip memory. Figure 15.3(b) shows the typical sizes for multipliers. Again the actual multiplier size will depend on the architecture (Booth encoding, Wallace tree, and so on), the process technology, and design rules. Table 15.5 shows some estimated gate counts for medium-size functions corresponding to some popular ASSP devices.

TABLE 15.5 Gate size estimates for popular ASSP functions.

ASSP device	Function	Gate estimate
8251A	Universal synchronous/asynchronous receiver/transmitter (USART)	2900
8253	Programmable interval timer	5680
8255A	Programmable peripheral interface	784–1403
8259	Programmable interrupt controller	2205
8237	Programmable DMA controller	5100
8284	Clock generator/driver	99
8288	Bus controller	250
8254	Programmable interval timer	3500
6845	CRT controller	2843
87030	SCSI controller	3600
87012	Ethernet controller	3900
2901	4 bit ALU	917
2902	Carry-lookahead ALU	33
2904	Status and shift control	500
2910	12-bit microprogram controller	1100

Source: Fujitsu channelless gate-array data book, AU and CG21 series.

15.5 Power Dissipation

Power dissipation in CMOS logic arises from the following sources:

- Dynamic power dissipation due to **switching current** from charging and discharging parasitic capacitance.
- Dynamic power dissipation due to **short-circuit current** when both *n*-channel and *p*-channel transistors are momentarily on at the same time.
- Static power dissipation due to **leakage current** and **subthreshold current**.

15.5.1 Switching Current

When the *p*-channel transistor in an inverter is charging a capacitance, C, at a frequency, f, the current through the transistor is $C(\mathrm{d}V/\mathrm{d}t)$. The power dissipation is thus $CV(\mathrm{d}V/\mathrm{d}t)$ during one-half the period of the input, $t = 1/(2f)$. The energy (in joules) dissipated in the *p*-channel transistor is thus

$$\int_0^{1/(2f)} CV\left(\frac{dV}{dt}\right)dt = \int_0^{V_{DD}} CVdV = \frac{1}{2}CV_{DD}^2 . \qquad (15.3)$$

When the *n*-channel transistor discharges the capacitor, the energy dissipated is the same. The average power dissipation over the whole cycle (in watts) is thus

$$P_1 = fCV_{DD}^2 . \qquad (15.4)$$

Most of the power dissipation in a CMOS ASIC arises from this source—the switching current. The best way to reduce power is to reduce V_{DD} (because it appears as a squared term in Eq. 15.4), and to reduce *C*, the amount of capacitance we have to switch. A rough estimate is that 20 percent of the nodes switch (or **toggle**) in a circuit per clock cycle. To determine more accurately the power dissipation due to switching, we need to find out how many nodes toggle during typical circuit operation using a dynamic logic simulator. This requires input vectors that correspond to typical operation, which can be difficult to produce. Using a digital simulator also will not take into account the effect of glitches, which can be significant. Power simulators are usually a hybrid between SPICE transistor-level simulators and digital event-driven simulators [Najm, 1994].

15.5.2 Short-Circuit Current

The short-circuit current or **crowbar current** can be particularly important for output drivers and large clock buffers. For a CMOS inverter (see Problem 15.17) the power dissipation due to the crowbar current is

$$P_2 = \frac{\beta f t_{rf}}{12}(V_{DD} - 2\mathrm{V}_{tn})^3 , \qquad (15.5)$$

where we assume the following: We ratio the *p*-channel and *n*-channel transistor sizes so that $\beta = (\mathrm{W/L})\mu\mathrm{C_{ox}}$ is the same for both *p*- and *n*-channel transistors, the magnitude of the threshold voltages $|\mathrm{V}_{tn}|$ are assumed equal for both transistor types, and t_{rf} is the rise and fall time (assumed equal) of the input signal [Veendrick, 1984]. For example, consider an output buffer that is capable of sinking 12 mA at an output voltage of 0.5 V. From Eq. 2.9 we can derive the transistor gain factor that we need as follows:

$$\begin{aligned}\beta &= \frac{I_{DS}}{\left[(V_{GS} - \mathrm{V}_{tn}) - \frac{1}{2}V_{DS}\right]V_{DS}} = \frac{12 \times 10^{-3}}{[(3.3 - 0.65) - (0.5)(0.5)](0.5)} \\ &= \frac{12 \times 10^{-3}}{[(3.3 - 0.65) - (0.5)(0.5)](0.5)} \qquad (15.6) \\ &= 0.01\,\mathrm{AV}^{-1}.\end{aligned}$$

If the output buffer is switching at 100 MHz and the input rise time to the buffer is 2 ns, we can calculate the power dissipation due to short-circuit current as

$$\begin{aligned} P_2 &= \frac{\beta f t_{rf}}{12}(V_{DD} - 2\mathrm{V}_{tn})^3 \\ &= \frac{(0.01)\ (100 \times 10^6)\ (2 \times 10^{-9})}{12}(3.3 - (2)\ (0.65))^3 \\ &= 0.00133333\,\mathrm{W} \quad \text{or about 1 mW.} \end{aligned} \tag{15.7}$$

If the output load is 10 pF, the dissipation due to switching current is

$$P_1 = fCV_{DD}{}^2 = (100 \times 10^6)\ (10 \times 10^{-12})\ (3.3)^2 = 0.01089\ \mathrm{W} \quad \text{or about 10mW.}$$

As a general rule, if we adjust the transistor sizes so that the rise times and fall times through a chain of logic are approximately equal (as they should be), the short-circuit current is typically less than 20 percent of the switching current.

For the example output buffer, we can make a rough estimate of the output-node switching time by assuming the buffer output drive current is constant at 12 mA. This current will cause the voltage on the output load capacitance to change between 3.3 V and 0 V at a constant slew rate dV/dt for a time

$$\Delta t = \frac{C\Delta V}{I} = \frac{(10 \times 10^{-12})\ (3.3)}{(12 \times 10^{-3})} = 2.75\ \text{ns.} \tag{15.8}$$

This is close to the input rise time of 2 ns. So our estimate of the short-circuit current being less than 20 percent of the switching current assuming equal input rise time and output rise time is valid in this case.

15.5.3 Subthreshold and Leakage Current

Despite the claim made in Section 2.1, a CMOS transistor is never completely *off*. For example, a typical specification for a 0.5 µm process for the **subthreshold current** (per micron of gate width for $V_{GS} = 0$ V) is less than 5 pAµm^{-1}, but not zero. With 10 million transistors on a large chip and with each transistor 10 µm wide, we will have a total subthreshold current of 0.1 mA; high, but reasonable. The problem is that the subthreshold current does not scale with process technology.

When the gate-to-source voltage, V_{GS}, of an MOS transistor is less than the threshold voltage, V$_t$, the transistor conducts a very small subthreshold current in the **subthreshold region**

$$I_{DS} = \mathrm{I}_0 \exp\left(\frac{\mathrm{q}V_{GS}}{\mathrm{nkT}} - 1\right), \tag{15.9}$$

where I$_0$ is a constant, and the constant, n, is normally between 1 and 2.

The slope, S, of the transistor current in the subthreshold region is

$$S = \frac{-nkT}{q}\log_{10}e = 2.3\frac{nkT}{q}\ \text{V/decade}. \tag{15.10}$$

For example, at a junction temperature, T = 125 °C ($\approx$400 K) and assuming $n \approx 1.5$, S = 120 mV/decade ($q = 1.6 \times 10^{-19}\ \mathrm{Fm^{-1}}$, $k = 1.38 \times 10^{-23}\ \mathrm{JK^{-1}}$), which does not scale. The constant value of S = 120 mV/decade means it takes 120 mV to reduce the subthreshold current by a factor of 10 in any process. If we reduce the threshold voltages to 0.36 V in a deep-submicron process, for example, this means at $V_{GS} = 0$ V we can only reduce I_{DS} to 0.001 times its value at $V_{GS} = V_t$. This problem can lead to large static currents.

Transistor leakage is caused by the fact that a reverse-biased diode conducts a very small **leakage current**. The sources and drains of every transistor, as well as the junctions between the wells and substrate, form parasitic diodes. The parasitic-diode leakage currents are strongly dependent on the type and quality of the process as well as temperature. The parasitic diodes have two components in parallel: an area diode and a perimeter diode. The ideal parasitic diode currents are given by the following equation:

$$I = I_s \exp\left(\frac{qV_D}{nkT} - 1\right). \tag{15.11}$$

Table 15.6 shows specified maximum leakage currents of junction parasitic diodes as well as the leakage currents of the **field transistors** (the parasitic MOS transistors formed when poly crosses over the thick oxide, or field oxide) in a typical 0.5 µm process.

TABLE 15.6 Diffusion leakage currents (at 25 °C) for a typical 0.5 µm (λ = 0.25 µm) CMOS process.

Junction	Diode type	Leakage (max.)	Unit
n-diffusion/*p*-substrate	area	0.6	$\mathrm{fA\mu m^{-2}V^{-1}}$
n-diffusion/*p*-substrate	perimeter	2.0	$\mathrm{fA\mu m^{-1}V^{-1}}$
p-diffusion/*n*-well	area	0.6	$\mathrm{fA\mu m^{-2}V^{-1}}$
p-diff/*n*-well	perimeter	3.0	$\mathrm{fA\mu m^{-1}V^{-1}}$
n-well/*p*-substrate	area	1.0	$\mathrm{fA\mu m^{-2}V^{-1}}$
Field NMOS transistor		100	$\mathrm{fA\mu m^{-1}}$
Field PMOS transistor		30	$\mathrm{fA\mu m^{-1}}$

For example, if we have an *n*-diffusion region at a potential of 3.3 V that is 10 μm by 4 μm in size, the parasitic leakage current due to the area diode would be

$$40\ \mu\text{m}^2 \times 3.3\text{V} \times 0.6\text{fA}\ \mu\text{m}^{-2}\text{V}^{-1} = (40)\ (3.3)\ (0.6 \times 10^{-15}) = 7.92 \times 10^{-14}\text{A},$$

or approximately 80 fA.

The perimeter of this drain region is 28 μm, so that the leakage current due to the perimeter diode is

$$28\ \mu\text{m} \times 3.3\text{V} \times 2.0\ \text{fA}\ \mu\text{m}^{-1}\text{V}^{-1} = (28)\ (3.3)\ (2.0 \times 10^{-15}) = 1.848 \times 10^{-13}\text{A},$$

or approximately 0.2 pA, over twice as large as the area-diode leakage current.

As a very rough estimate, if we have 100,000 transistors each with a source and a drain 10 μm by 4 μm, and half of them are biased at 3.3 V, then the total leakage current would be

$$(100 \times 10^{5})\ (2)\ (0.5)\ (280 \times 10^{-15}) = 2.8 \times 10^{-6}\ \text{A}, \qquad (15.12)$$

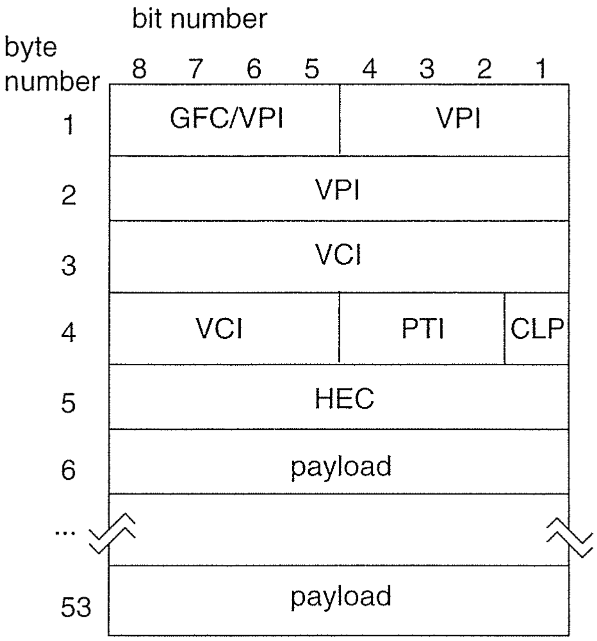
or approximately 3 μA. This is the same order of magnitude (a few microamperes) as the **quiescent leakage current**, I_{DDQ}, that we expect to measure when we test an ASIC with power applied, but with no signal activity. A measurement of more current than this in a nonactive CMOS ASIC indicates a problem with the chip manufacture or the design. We use this measurement to test an ASIC using an **IDDQ test**.

15.6 FPGA Partitioning

In Section 15.3 we saw how many different issues have to be considered when partitioning a complex system into custom ASICs. There are no commercial tools that can help us with all of these issues—a spreadsheet is the best tool in this case. Things are a little easier if we limit ourselves to partitioning a group of logic cells into FPGAs—and restrict the FPGAs to be all of the same type.

15.6.1 ATM Simulator

In this section we shall examine a hardware simulator for **Asynchronous Transfer Mode** (**ATM**). ATM is a signaling protocol for many different types of traffic including constant bit rates (voice signals) as well as variable bit rates (compressed video). The ATM Connection Simulator is a card that is connected to a computer. Under computer control the card monitors and corrupts the ATM signals to simulate the effects of real networks. An example would be to test different video compression algorithms. Compressed video is very bursty (brief periods of very high activity), has very strict delay constraints, and is susceptible to errors. ATM is based on ATM cells (packets). Each ATM cell has 53 bytes: a 5-byte header and a 48-byte payload; Figure 15.4 shows the format of the ATM packet. The ATM Connection Simulator looks at the entire header as an address.



GFC = generic flow control
VPI = virtual path identifier
VCI = virtual channel identifier
PTI = payload type identifier
CLP = cell loss priority
HEC = header error control

FIGURE 15.4 The asynchronous transfer mode (ATM) cell format. The ATM protocol uses 53-byte cells or packets of information with a data payload and header information for routing and error control.

Figure 15.5 shows the system block diagram of the ATM simulator designed by Craig Fujikami at the University of Hawaii. Now produced by AdTech, the simulator emulates the characteristics of a single connection in an ATM network and models ATM traffic policing, ATM cell delays, and ATM cell errors. The simulator is partitioned into the three major blocks, shown in Figure 15.5, and connected to an IBM-compatible PC through an Intel 80186 controller board together with an interface board. These three blocks are

- The traffic policer, which regulates the input to the simulator.
- The delay generator, which delays ATM cells, reorders ATM cells, and inserts ATM cells with valid ATM cell headers.
- The error generator, which produces bit errors and four random variables that are needed by the other two blocks.

The error generator performs the following operations on ATM cells:

1. Payload bit error ratio generation. The user specifies the Bernoulli probability, p_{BER}, of the payload bit error ratio.

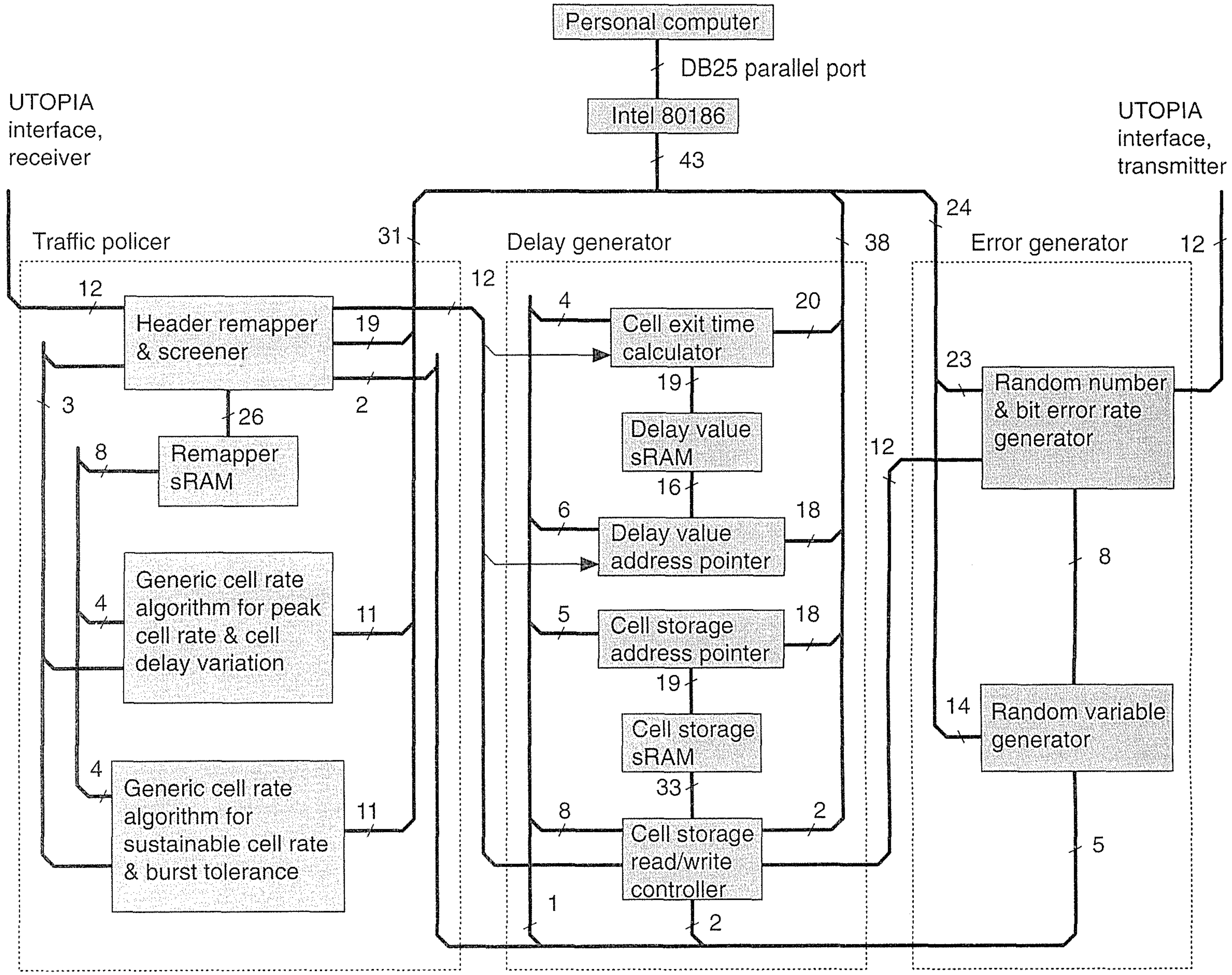

Personal computer
DB25 parallel port
Intel 80186
43
UTOPIA interface, receiver
UTOPIA interface, transmitter
Traffic policer
Delay generator
Error generator
31
38
24
12
12
Header remapper & screener
19
2
3
26
Remapper sRAM
8
Generic cell rate algorithm for peak cell rate & cell delay variation
4
11
Generic cell rate algorithm for sustainable cell rate & burst tolerance
4
11
12
4
Cell exit time calculator
20
19
Delay value sRAM
16
6
Delay value address pointer
18
5
Cell storage address pointer
18
19
Cell storage sRAM
33
8
Cell storage read/write controller
2
1
2
23
Random number & bit error rate generator
12
8
14
Random variable generator
5

FIGURE 15.5 An asynchronous transfer mode (ATM) connection simulator.

2. Random-variable generation for ATM cell loss, misinsertion, reordering, and deletion.

The delay generator delays, misinserts, and reorders the target ATM cells. Finally, the traffic policer performs the following operations:

3. Performs header screening and remapping.
4. Checks ATM cell conformance.
5. Deletes selected ATM cells.

Table 15.7 shows the partitioning of the ATM board into 12 Lattice Logic FPGAs (ispLSI 1048) corresponding to the 12 blocks shown in Figure 15.5. The Lattice Logic ispLSI 1048 has 48 GLBs (generic logic blocks) on each chip. This system was partitioned by hand—with difficulty. Tools for automatic partitioning of systems like this will become increasingly important. In Section 15.6.2 we shall briefly look at some examples of such tools, before examining the partitioning methods that are used in Section 15.7.

TABLE 15.7 Partitioning of the ATM board using Lattice Logic ispLSI 1048 FPGAs. Each FPGA contains 48 generic logic blocks (GLBs).

Chip #	Size	Chip #	Size
1	42 GLBs	7	36 GLBs
2	64k-bit × 8 SRAM	8	22 GLBs
3	38 GLBs	9	256k-bit × 16 SRAM
4	38 GLBs	10	43 GLBs
5	42 GLBs	11	40 GLBs
6	64k-bit × 16 SRAM	12	30 GLBs

15.6.2 Automatic Partitioning with FPGAs

Some vendors of programmable ASICs provide partitioning software. For example, Altera uses its own software system for design. You can perform design entry using an HDL, schematic entry, or using the Altera hardware design language (AHDL)—similar to PALASM or ABEL. In AHDL you can direct the partitioner to automatically partition logic into chips within the same family, using the AUTO keyword:

```
DEVICE top_level IS AUTO; % let the partitioner assign logic
```

You can use the `CLIQUE` keyword to keep logic together (this is not quite the same as a clique in a graph—more on this in Section 15.7.3):

```
CLIQUE fast_logic
BEGIN
```

```
  |shift_register: MACRO; % keep this in one device
END;
```

An additional option, to reserve space on a device, is very useful for making last minute additions or changes.

15.7 Partitioning Methods

System partitioning requires goals and objectives, methods and algorithms to find solutions, and ways to evaluate these solutions. We start with measuring connectivity, proceed to an example that illustrates the concepts of system partitioning and then to the algorithms for partitioning.

Assume that we have decided which parts of the system will use ASICs. The goal of partitioning is to divide this part of the system so that each partition is a single ASIC. To do this we may need to take into account any or all of the following objectives:

- A maximum size for each ASIC
- A maximum number of ASICs
- A maximum number of connections for each ASIC
- A maximum number of total connections between all ASICs

We know how to measure the first two objectives. Next we shall explain ways to measure the last two.

15.7.1 Measuring Connectivity

To measure connectivity we need some help from the mathematics of graph theory. It turns out that the terms, definitions, and ideas of graph theory are central to ASIC construction, and they are often used in manuals and books that describe the knobs and dials of ASIC design tools.

Figure 15.6(a) shows a circuit schematic, netlist, or **network**. The network consists of **circuit modules** A–F. Equivalent terms for a circuit module are a cell, logic cell, macro, or a block. A cell or logic cell usually refers to a small logic gate (NAND etc.), but can also be a collection of other cells; macro refers to gate-array cells; a block is usually a collection of gates or cells. We shall use the term *logic cell* in this chapter to cover all of these.

Each logic cell has electrical connections between the **terminals** (**connectors** or **pins**). The network can be represented as the mathematical **graph** shown in Figure 15.6(b). A graph is like a spider's web: it contains **vertexes** (or **vertices**) A–F (also known as graph **nodes** or **points**) that are connected by **edges**. A graph **vertex** corresponds to a logic cell. An electrical **connection** (a **net** or a **signal**) between two logic cells corresponds to a graph **edge**.

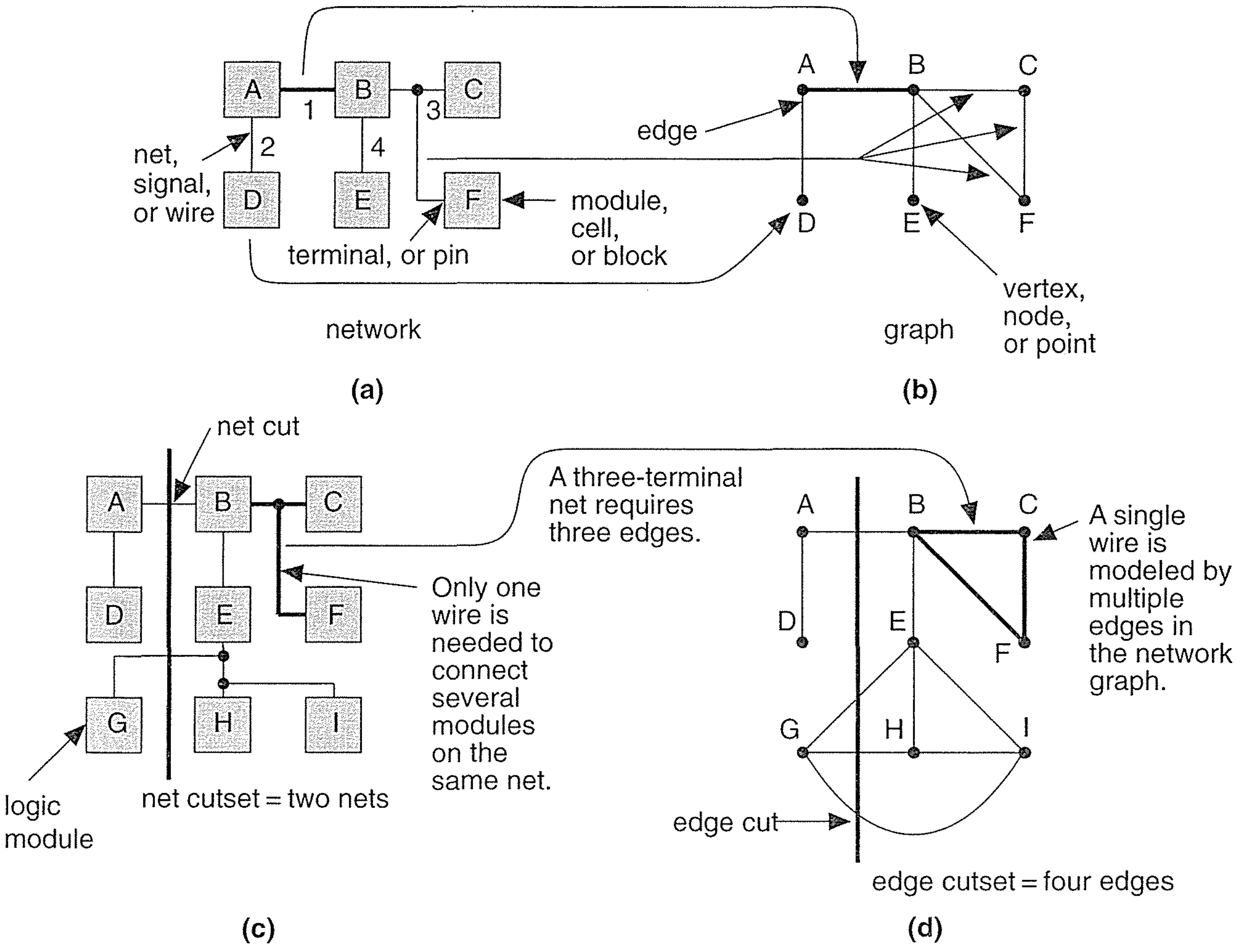


FIGURE 15.6 Networks, graphs, and partitioning. (a) A network containing circuit logic cells and nets. (b) The equivalent graph with vertexes and edges. For example: logic cell D maps to node D in the graph; net 1 maps to the edge (A, B) in the graph. Net 3 (with three connections) maps to three edges in the graph: (B, C), (B, F), and (C, F). (c) Partitioning a network and its graph. A network with a net cut that cuts two nets. (d) The network graph showing the corresponding edge cut. The net cutset in c contains two nets, but the corresponding edge cutset in d contains four edges. This means a graph is not an exact model of a network for partitioning purposes.

Figure 15.6(c) shows a network with nine logic cells A–I. A connection, for example between logic cells A and B in Figure 15.6(c), is written as net (A, B). Net (A, B) is represented by the single edge (A, B) in the network graph, shown in Figure 15.6(d). A net with three terminals, for example net (B, C, F), must be modeled with three edges in the network graph: edges (B, C), (B, F), and (C, F). A net with four terminals requires six edges and so on. Figure 15.6 illustrates the differences between the nets of a network and the edges in the network graphs. Notice that a net can have more than two terminals, but a terminal has only one net.

If we divide, or partition, the network shown in Figure 15.6(c) into two parts, corresponding to creating two ASICs, we can divide the network's graph in the same way. Figure 15.6(d) shows a possible division, called a **cutset**. We say that there is a **net cutset** (for the network) and an **edge cutset** (for the graph). The connections between the two ASICs are **external connections**, the connections inside each ASIC are **internal connections**.

Notice that the number of external connections is not modeled correctly by the network graph. When we divide the network into two by drawing a line across connections, we make **net cuts**. The resulting set of net cuts is the net cutset. The number of net cuts we make corresponds to the number of external connections between the two partitions. When we divide the network graph into the same partitions we make **edge cuts** and we create the edge cutset. We have already shown that nets and graph edges are not equivalent when a net has more than two terminals. Thus the number of edge cuts made when we partition a graph into two is not necessarily equal to the number of net cuts in the network. As we shall see presently the differences between nets and graph edges is important when we consider partitioning a network by partitioning its graph [Schweikert and Kernighan, 1979].

15.7.2 A Simple Partitioning Example

Figure 15.7(a) shows a simple network we need to partition [Goto and Matsud, 1986]. There are 12 logic cells, labeled A–L, connected by 12 nets (labeled 1–12). At this level, each logic cell is a large circuit block and might be RAM, ROM, an ALU, and so on. Each net might also be a bus, but, for the moment, we assume that each net is a single connection and all nets are weighted equally. The goal is to partition our simple network into ASICs. Our objectives are the following:

- Use no more than three ASICs.
- Each ASIC is to contain no more than four logic cells.
- Use the minimum number of external connections for each ASIC.
- Use the minimum total number of external connections.

Figure 15.7(b) shows a partitioning with five external connections; two of the ASICs have three pins; the third has four pins.We might be able to find this arrangement by hand, but for larger systems we need help.

Splitting a network into several pieces is a **network partitioning problem**. In the following sections we shall examine two types of algorithms to solve this problem and describe how they are used in system partitioning. Section 15.7.3 describes **constructive partitioning**, which uses a set of rules to find a solution. Section 15.7.4 describes **iterative partitioning improvement** (or **iterative partitioning refinement**), which takes an existing solution and tries to improve it.

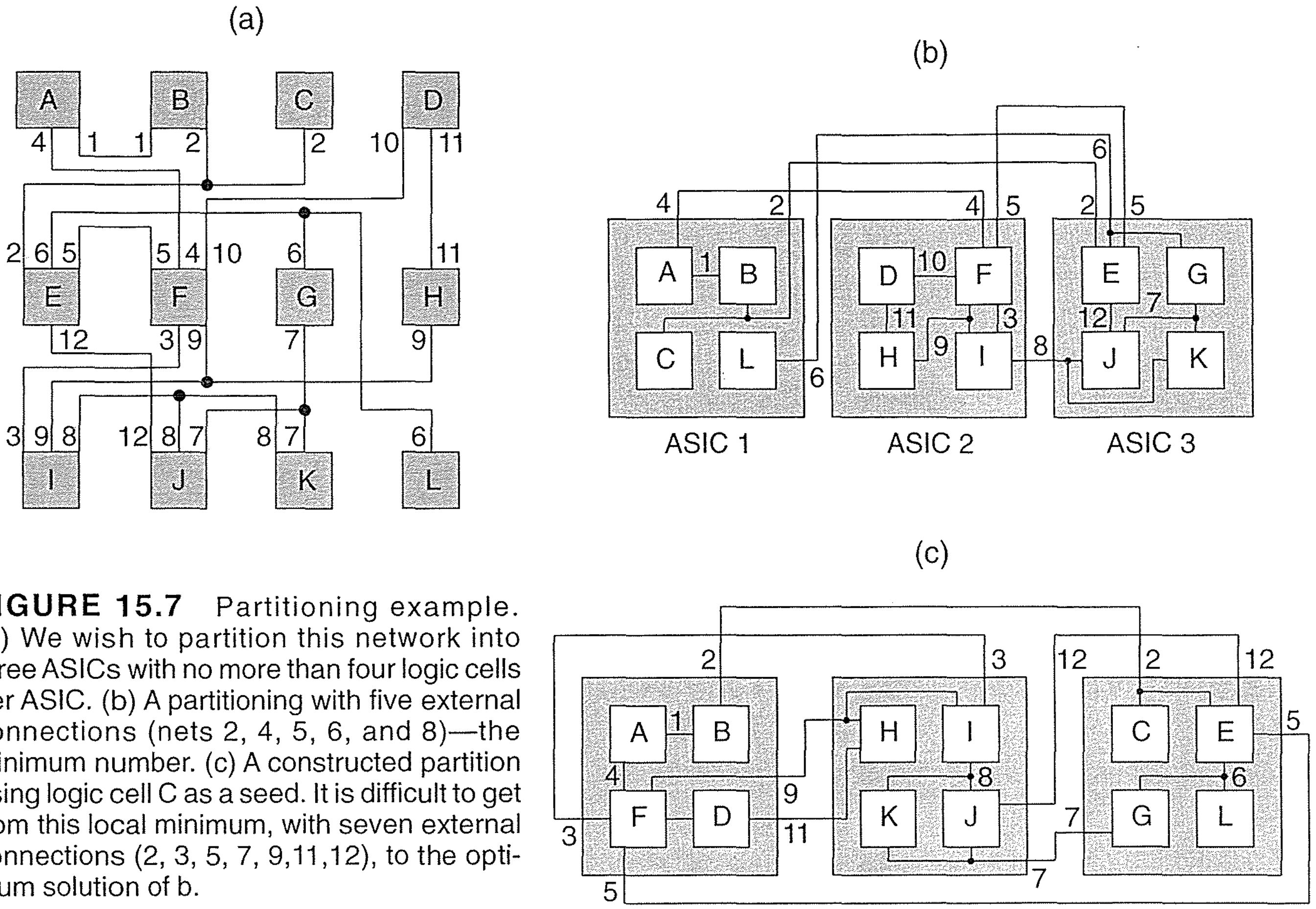


FIGURE 15.7 Partitioning example. (a) We wish to partition this network into three ASICs with no more than four logic cells per ASIC. (b) A partitioning with five external connections (nets 2, 4, 5, 6, and 8)—the minimum number. (c) A constructed partition using logic cell C as a seed. It is difficult to get from this local minimum, with seven external connections (2, 3, 5, 7, 9,11,12), to the optimum solution of b.

Often we apply iterative improvement to a constructive partitioning. We also use many of these partitioning algorithms in solving floorplanning and placement problems that we shall discuss in Chapter 16.

15.7.3 Constructive Partitioning

The most common constructive partitioning algorithms use **seed growth** or **cluster growth**. A simple seed-growth algorithm for constructive partitioning consists of the following steps:

1. Start a new partition with a seed logic cell.
2. Consider all the logic cells that are not yet in a partition. Select each of these logic cells in turn.
3. Calculate a gain function, *g(m)*, that measures the benefit of adding logic cell *m* to the current partition. One measure of gain is the number of connections between logic cell *m* and the current partition.
4. Add the logic cell with the highest gain *g(m)* to the current partition.

5. Repeat the process from step 2. If you reach the limit of logic cells in a partition, start again at step 1.

We may choose different gain functions according to our objectives (but we have to be careful to distinguish between connections and nets). The algorithm starts with the choice of a **seed logic cell** (**seed module**, or just **seed**). The logic cell with the most nets is a good choice as the seed logic cell. You can also use a set of seed logic cells known as a **cluster**. Some people also use the term *clique*—borrowed from graph theory. A **clique** of a graph is a subset of nodes where each pair of nodes is connected by an edge—like your group of friends at school where everyone knows everyone else in your clique. In some tools you can use schematic pages (at the leaf or lowest hierarchical level) as a starting point for partitioning. If you use a high-level design language, you can use a Verilog module (different from a circuit module) or VHDL entity/architecture as seeds (again at the leaf level).

15.7.4 Iterative Partitioning Improvement

The most common iterative improvement algorithms are based on **interchange** and **group migration**. The process of interchanging (swapping) logic cells in an effort to improve the partition is an **interchange method**. If the swap improves the partition, we accept the trial interchange; otherwise we select a new set of logic cells to swap.

There is a limit to what we can achieve with a partitioning algorithm based on simple interchange. For example, Figure 15.7(c) shows a partitioning of the network of part a using a constructed partitioning algorithm with logic cell C as the seed. To get from the solution shown in part c to the solution of part b, which has a minimum number of external connections, requires a complicated swap. The three pairs: D and F, J and K, C and L need to be swapped—all at the same time. It would take a very long time to consider all possible swaps of this complexity. A simple interchange algorithm considers only one change and rejects it immediately if it is not an improvement. Algorithms of this type are **greedy algorithms** in the sense that they will accept a move only if it provides immediate benefit. Such shortsightedness leads an algorithm to a **local minimum** from which it cannot escape. Stuck in a valley, a greedy algorithm is not prepared to walk over a hill to see if there is a better solution in the next valley. This type of problem occurs repeatedly in CAD algorithms.

Group migration consists of swapping groups of logic cells between partitions. The **group migration algorithms** are better than simple interchange methods at improving a solution but are more complex. Almost all group migration methods are based on the powerful and general **Kernighan–Lin algorithm** (**K–L algorithm**) that partitions a graph [Kernighan and Lin, 1970]. The problem of dividing a graph into two pieces, minimizing the nets that are cut, is the **min-cut problem**—a very important one in VLSI design. As the next section shows, the K–L algorithm can be applied to many different problems in ASIC design. We shall examine the algorithm next and then see how to apply it to system partitioning.

15.7.5 The Kernighan–Lin Algorithm

Figure 15.8 illustrates some of the terms and definitions needed to describe the K–L algorithm. External edges cross between partitions; internal edges are contained inside a partition. Consider a network with 2 m nodes (where m is an integer) each of equal size. If we assign a cost to each edge of the network graph, we can define a **cost matrix** $\mathrm{C} = c_{ij}$, where $c_{ij} = c_{ji}$ and $c_{ii} = 0$. If all connections are equal in importance, the elements of the cost matrix are 1 or 0, and in this special case we usually call the matrix the **connectivity matrix**. Costs higher than 1 could represent the number of wires in a bus, multiple connections to a single logic cell, or nets that we need to keep close for timing reasons.

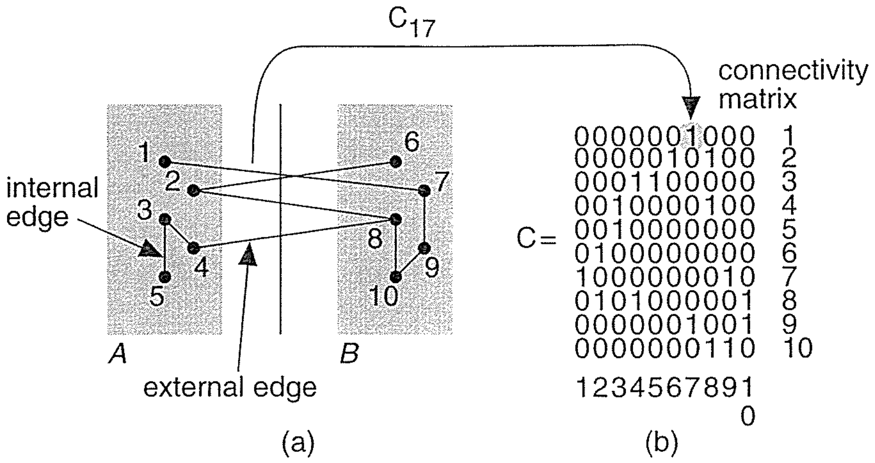


FIGURE 15.8 Terms used by the Kernighan–Lin partitioning algorithm. (a) An example network graph. (b) The connectivity matrix, C; the column and rows are labeled to help you see how the matrix entries correspond to the node numbers in the graph. For example, C_{17} (column 1, row 7) equals 1 because nodes 1 and 7 are connected. In this example all edges have an equal weight of 1, but in general the edges may have different weights.

Suppose we already have split a network into two partitions, A and B, each with m nodes (perhaps using a constructed partitioning). Our goal now is to swap nodes between A and B with the objective of minimizing the number of external edges connecting the two partitions. Each external edge may be weighted by a cost, and our objective corresponds to minimizing a cost function that we shall call the total external cost, **cut cost**, or **cut weight**, W:

$$W = \sum_{a \in A,\, b \in B} c_{ab} \,. \qquad (15.13)$$

In Figure 15.8(a) the cut weight is 4 (all the edges have weights of 1).

In order to simplify the measurement of the change in cut weight when we interchange nodes, we need some more definitions. First, for any node a in partition A, we define an **external edge cost,** which measures the connections from node a to B,

$$E_a = \sum_{y \in B} c_{ay} . \tag{15.14}$$

For example, in Figure 15.8(a) $E_1 = 1$, and $E_3 = 0$. Second, we define the **internal edge cost** to measure the internal connections to a,

$$I_a = \sum_{z \in A} c_{az} . \tag{15.15}$$

So, in Figure 15.8(a), $I_1 = 0$, and $I_3 = 2$. We define the edge costs for partition B in a similar way (so $E_8 = 2$, and $I_8 = 1$). The cost difference is the difference between external edge costs and internal edge costs,

$$D_x = E_x - I_x . \tag{15.16}$$

Thus, in Figure 15.8(a) $D_1 = 1$, $D_3 = -2$, and $D_8 = 1$. Now pick any node in A, and any node in B. If we swap these nodes, a and b, we need to measure the reduction in cut weight, which we call the gain, g. We can express g in terms of the edge costs as follows:

$$g = D_a + D_b - 2c_{ab} . \tag{15.17}$$

The last term accounts for the fact that a and b may be connected. So, in Figure 15.8(a), if we swap nodes 1 and 6, then $g = D_1 + D_6 - 2c_{16} = 1 + 1$. If we swap nodes 2 and 8, then $g = D_2 + D_8 - 2c_{28} = 1 + 2 - 2$.

The K–L algorithm finds a group of node pairs to swap that increases the gain even though swapping individual node pairs from that group might decrease the gain. First we pretend to swap all of the nodes a pair at a time. Pretend swaps are like studying chess games when you make a series of trial moves in your head.

This is the algorithm:

1. Find two nodes, a_i from A, and b_i from B, so that the gain from swapping them is a maximum. The gain is

$$g_i = D_{a_i} + D_{b_i} - 2c_{a_i b_i} . \tag{15.18}$$

2. Next pretend swap a_i and b_i even if the gain g_i is zero or negative, and do not consider a_{i} and b_i eligible for being swapped again.
3. Repeat steps 1 and 2 a total of m times until all the nodes of A and B have been pretend swapped. We are back where we started, but we have ordered pairs of nodes in A and B according to the gain from interchanging those pairs.

4. Now we can choose which nodes we shall actually swap. Suppose we only swap the first n pairs of nodes that we found in the preceding process. In other words we swap nodes $X = a_1, a_2, \ldots, a_n$ from A with nodes $Y = b_1, b_2, \ldots, b_n$ from B. The total gain would be

$$G_n = \sum_{i=1}^{n} g_i . \qquad (15.19)$$

5. We now choose n corresponding to the maximum value of G_n.

If the maximum value of $G_n > 0$, then we swap the sets of nodes X and Y and thus reduce the cut weight by G_n. We use this new partitioning to start the process again at the first step. If the maximum value of $G_n = 0$, then we cannot improve the current partitioning and we stop. We have found a locally optimum solution.

Figure 15.9 shows an example of partitioning a graph using the K–L algorithm. Each completion of steps 1 through 5 is a pass through the algorithm. Kernighan and Lin found that typically 2–4 passes were required to reach a solution. The most important feature of the K–L algorithm is that we are prepared to consider moves even though they seem to make things worse. This is like unraveling a tangled ball of string or solving a Rubik's cube puzzle. Sometimes you need to make things worse so they can get better later. The K–L algorithm works well for partitioning graphs. However, there are the following problems that we need to address before we can apply the algorithm to network partitioning:

- It minimizes the number of *edges* cut, not the number of *nets* cut.
- It does not allow logic cells to be different sizes.
- It is expensive in computation time.
- It does not allow partitions to be unequal or find the optimum partition size.
- It does not allow for selected logic cells to be fixed in place.
- The results are random.
- It does not directly allow for more than two partitions.

To implement a **net-cut partitioning** rather than an **edge-cut partitioning**, we can just keep track of the nets rather than the edges [Schweikert and Kernighan, 1979]. We can no longer use a connectivity or cost matrix to represent connections, though. Fortunately, several people have found efficient data structures to handle the bookkeeping tasks. One example is the Fiduccia–Mattheyses algorithm to be described shortly.

To represent nets with multiple terminals in a network accurately, we can extend the definition of a network graph. Figure 15.10 shows how a **hypergraph** with a special type of vertex, a **star**, and a **hyperedge**, represents a net with more than two terminals in a network.

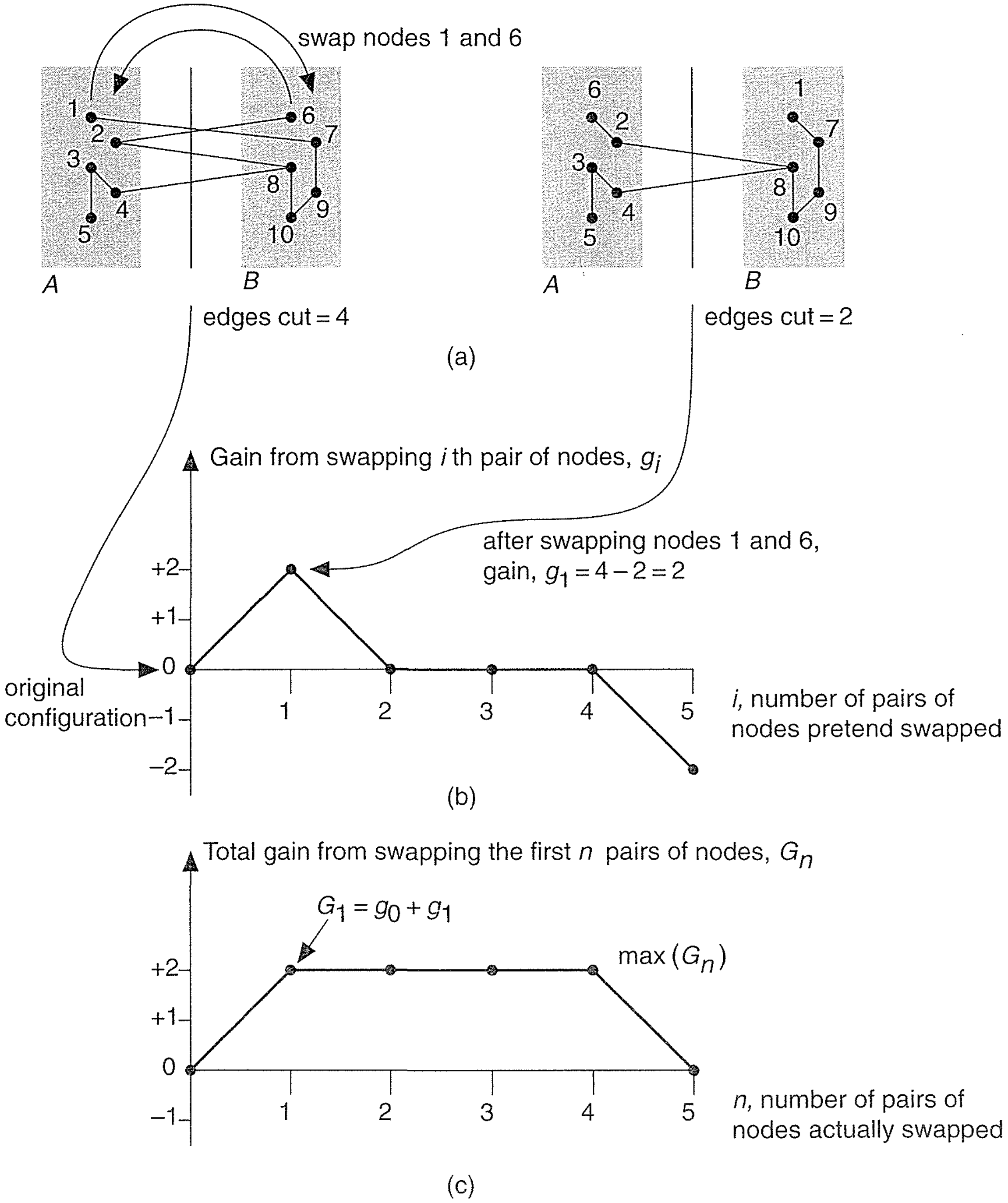

swap nodes 1 and 6
1
2
3
4
5
6
7
8
9
10
A
B
edges cut = 4
6
2
3
4
5
1
7
8
9
10
A
B
edges cut = 2
(a)
Gain from swapping i th pair of nodes, g_i
after swapping nodes 1 and 6,
gain, $g_1 = 4 - 2 = 2$
+2
+1
0
original configuration
−1
−2
1
2
3
4
5
i, number of pairs of nodes pretend swapped
(b)
Total gain from swapping the first n pairs of nodes, G_n
$G_1 = g_0 + g_1$
max (G_n)
+2
+1
0
−1
1
2
3
4
5
n, number of pairs of nodes actually swapped
(c)

FIGURE 15.9 Partitioning a graph using the Kernighan–Lin algorithm. (a) Shows how swapping node 1 of partition A with node 6 of partition B results in a gain of g = 1. (b) A graph of the gain resulting from swapping pairs of nodes. (c) The total gain is equal to the sum of the gains obtained at each step.

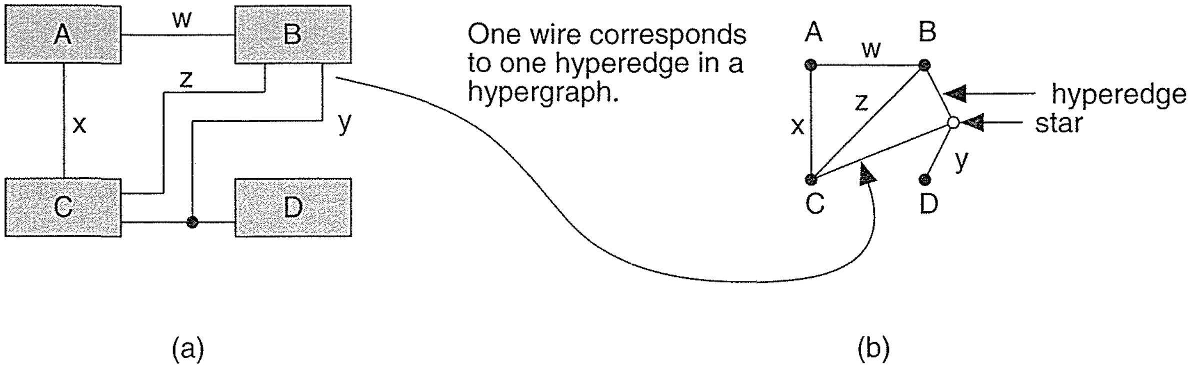


FIGURE 15.10 A hypergraph. (a) The network contains a net y with three terminals. (b) In the network hypergraph we can model net y by a single hyperedge (B, C, D) and a star node. Now there is a direct correspondence between wires or nets in the network and hyperedges in the graph.

In the K–L algorithm, the internal and external edge costs have to be calculated for all the nodes before we can select the nodes to be swapped. Then we have to find the pair of nodes that give the largest gain when swapped. This requires an amount of computer time that grows as $n^2 \log n$ for a graph with $2n$ nodes. This n^2 dependency is a major problem for partitioning large networks. The **Fiduccia–Mattheyses algorithm** (the **F–M algorithm**) is an extension to the K–L algorithm that addresses the differences between nets and edges and also reduces the computational effort [Fiduccia and Mattheyses, 1982]. The key features of this algorithm are the following:

- Only one logic cell, the **base logic cell**, moves at a time. In order to stop the algorithm from moving all the logic cells to one large partition, the base logic cell is chosen to maintain **balance** between partitions. The balance is the ratio of total logic cell size in one partition to the total logic cell size in the other. Altering the balance allows us to vary the sizes of the partitions.
- Critical nets are used to simplify the gain calculations. A net is a **critical net** if it has an attached logic cell that, when swapped, changes the number of nets cut. It is only necessary to recalculate the gains of logic cells on critical nets that are attached to the base logic cell.
- The logic cells that are free to move are stored in a doubly linked list. The lists are sorted according to gain. This allows the logic cells with maximum gain to be found quickly.

These techniques reduce the computation time so that it increases only slightly more than linearly with the number of logic cells in the network, a very important improvement [Fiduccia and Mattheyses, 1982].

Kernighan and Lin suggested simulating logic cells of different sizes by clumping s logic cells together with highly weighted nets to simulate a logic cell of size s. The F–M algorithm takes logic-cell size into account as it selects a logic cell to swap based on maintaining the balance between the total logic-cell size of each of the partitions. To generate unequal partitions using the K–L algorithm, we can introduce dummy logic cells with no connections into one of the partitions. The F–M algorithm adjusts the partition size according to the balance parameter.

Often we need to fix logic cells in place during partitioning. This may be because we need to keep logic cells together or apart for reasons other than connectivity, perhaps due to timing, power, or noise constraints. Another reason to fix logic cells would be to improve a partitioning that you have already partially completed. The F–M algorithm allows you to fix logic cells by removing them from consideration as the base logic cells you move. Methods based on the K–L algorithm find locally optimum solutions in a random fashion. There are two reasons for this. The first reason is the random starting partition. The second reason is that the choice of nodes to swap is based on the gain. The choice between moves that have equal gain is arbitrary. Extensions to the K–L algorithm address both of these problems. Finding nodes that are naturally grouped or clustered and assigning them to one of the initial partitions improves the results of the K–L algorithm. Although these are constructive partitioning methods, they are covered here because they are closely linked with the K–L iterative improvement algorithm.

15.7.6 The Ratio-Cut Algorithm

The **ratio-cut algorithm** removes the restriction of constant partition sizes. The cut weight W for a cut that divides a network into two partitions, A and B, is given by

$$W = \sum_{a \in A, b \in B} c_{ab} \, . \tag{15.20}$$

The K–L algorithm minimizes W while keeping partitions A and B the same size. The **ratio** of a cut is defined as

$$R = \frac{W}{|A||B|} \, . \tag{15.21}$$

In this equation $|A|$ and $|B|$ are the sizes of partitions A and B. The size of a partition is equal to the number of nodes it contains (also known as the **set cardinality**). The cut that minimizes R is called the **ratio cut**. The original description of the ratio-cut algorithm uses ratio cuts to partition a network into small, highly connected groups. Then you form a reduced network from these groups—each small group of logic cells forms a node in the reduced network. Finally, you use the F–M algorithm to improve the reduced network [Cheng and Wei, 1991].

15.7.7 The Look-ahead Algorithm

Both the K–L and F–M algorithms consider only the immediate gain to be made by moving a node. When there is a tie between nodes with equal gain (as often happens), there is no mechanism to make the best choice. This is like playing chess looking only one move ahead. Figure 15.11 shows an example of two nodes that have equal gains, but moving one of the nodes will allow a move that has a higher gain later.

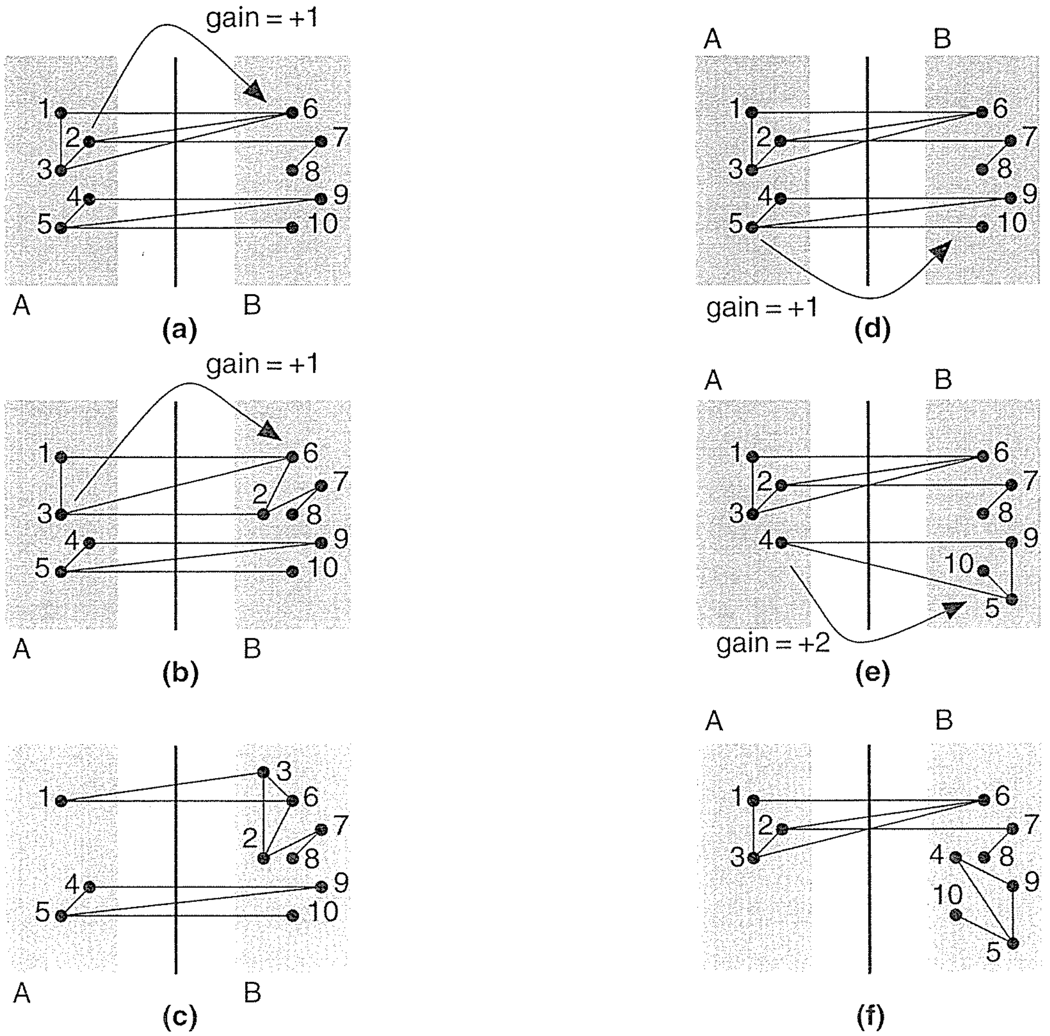


FIGURE 15.11 An example of network partitioning that shows the need to look ahead when selecting logic cells to be moved between partitions. Partitionings (a), (b), and (c) show one sequence of moves, partitionings (d), (e), and (f) show a second sequence. The partitioning in (a) can be improved by moving node 2 from A to B with a gain of 1. The result of this move is shown in (b). This partitioning can be improved by moving node 3 to B, again with a gain of 1. The partitioning shown in (d) is the same as (a). We can move node 5 to B with a gain of 1 as shown in (e), but now we can move node 4 to B with a gain of 2.

We call the gain for the initial move the first-level gain. Gains from subsequent moves are then second-level and higher gains. We can define a **gain vector** that contains these gains. Figure 15.11 shows how the first-level and second-level gains are calculated. Using the gain vector allows us to use a **look-ahead algorithm** in the choice of nodes to be swapped. This reduces both the mean and variation in the number of cuts in the resulting partitions.

We have described algorithms that are efficient at dividing a network into two pieces. Normally we wish to divide a system into more than two pieces. We can do this by recursively applying the algorithms. For example, if we wish to divide a system network into three pieces, we could apply the F–M algorithm first, using a balance of 2:1, to generate two partitions, with one twice as large as the other. Then we apply the algorithm again to the larger of the two partitions, with a balance of 1:1, which will give us three partitions of roughly the same size.

15.7.8 Simulated Annealing

A different approach to solving large graph problems (and other types of problems) that arise in VLSI layout, including system partitioning, uses the **simulated-annealing algorithm** [Kirkpatrick et al., 1983]. Simulated annealing takes an existing solution and then makes successive changes in a series of random moves. Each move is accepted or rejected based on an **energy function**, calculated for each new trial configuration. The minimums of the energy function correspond to possible solutions. The best solution is the **global minimum**.

So far the description of simulated annealing is similar to the interchange algorithms, but there is an important difference. In an interchange strategy we accept the new trial configuration only if the energy function decreases, which means the new configuration is an improvement. However, in the simulated-annealing algorithm, we accept the new configuration even if the energy function increases for the new configuration—which means things are getting worse. The probability of accepting a worse configuration is controlled by the exponential expression $\exp(-\Delta E / T)$, where ΔE is the resulting increase in the energy function. The parameter T is a variable that we control and corresponds to the temperature in the annealing of a metal cooling (this is why the process is called simulated annealing).

We accept moves that seemingly take us away from a desirable solution to allow the system to escape from a local minimum and find other, better, solutions. The name for this strategy is **hill climbing**. As the temperature is slowly decreased, we decrease the probability of making moves that increase the energy function. Finally, as the temperature approaches zero, we refuse to make any moves that increase the energy of the system and the system falls and comes to rest at the nearest local minimum. Hopefully, the solution that corresponds to the minimum we have found is a good one.

The critical parameter governing the behavior of the simulated-annealing algorithm is the rate at which the temperature T is reduced. This rate is known as the **cooling schedule**. Often we set a parameter α that relates the temperatures, T_i and T_{i+1}, at the ith and $i + 1$th iteration:

$$T_{i+1} = \alpha T_i . \qquad (15.22)$$

To find a good solution, a local minimum close to the global minimum, requires a high initial temperature and a slow cooling schedule. This results in many trial moves and very long computer run times [Rose, Klebsch, and Wolf, 1990]. If we are prepared to wait a long time (forever in the worst case), simulated annealing is useful because we can guarantee that we can find the optimum solution. Simulated annealing is useful in several of the ASIC construction steps and we shall return to it in Section 16.2.7.

15.7.9 Other Partitioning Objectives

In partitioning a real system we need to weight each logic cell according to its area in order to control the total areas of each ASIC. This can be done if the area of each logic cell can either be calculated or estimated. This is usually done as part of floorplanning, so we may need to return to partitioning after floorplanning.

There will be many objectives or constraints that we need to take into account during partitioning. For example, certain logic cells in a system may need to be located on the same ASIC in order to avoid adding the delay of any external interconnections. These **timing constraints** can be implemented by adding weights to nets to make them more important than others. Some logic cells may consume more power than others and you may need to add **power constraints** to avoid exceeding the power-handling capability of a single ASIC. It is difficult, though, to assign more than rough estimates of power consumption for each logic cell at the system planning stage, before any simulation has been completed. Certain logic cells may only be available in a certain technology—if you want to include memory on an ASIC, for example. In this case, **technology constraints** will keep together logic cells requiring similar technologies. We probably want to impose **cost constraints** to implement certain logic cells in the lowest cost technology available or to keep ASICs below a certain size in order to use a low-cost package. The type of test strategy you adopt will also affect the partitioning of logic. Large RAM blocks may require BIST circuitry; large amounts of sequential logic may require scan testing, possibly with a boundary-scan interface. One of the objects of testability is to maintain controllability and observability of logic inside each ASIC. In order to do this, **test constraints** may require that we force certain connections to be external. No automated partitioning tools can take into account all of these constraints. The best CAD tool to help you with these decisions is a spreadsheet.

15.8 Summary

The construction or physical design of ASICs in a microelectronics system is a very large and complex problem. To solve the problem we divide it into several steps: system partitioning, floorplanning, placement, and routing. To solve each of these smaller problems we need goals and objectives, measurement metrics, as well as algorithms and methods.

System partitioning is the first step in ASIC assembly. An example of the SPARCstation 1 illustrated the various issues involved in partitioning. Presently commercial CAD tools are able to automatically partition systems and chips only at a low level, at the level of a network or netlist. Partitioning for FPGAs is currently the most advanced. Next we discussed the methods to use for system partitioning. We saw how to represent networks as graphs, containing nets and edges, and how the mathematics of graph theory is useful in system partitioning and the other steps of ASIC assembly. We covered methods and algorithms for partitioning and explained that most are based on the Kernighan–Lin min-cut algorithm.

The important points in this chapter are

- The goals and objectives of partitioning
- Partitioning as an art not a science
- The simple nature of the algorithms necessary for VLSI-sized problems
- The random nature of the algorithms we use
- The controls for the algorithms used in ASIC design

15.9 Problems

*=Difficult, **=Very difficult, ***=Extremely difficult

15.1 (Complexity, 10 min.) Suppose the workstations we use to design ASICs increase in power (measured in MIPS—a million instructions per second) by a factor of 2 every year. If we want to keep the length of time to solve an ASIC design problem fixed, calculate how much larger chips can get each year if constrained by an algorithm with the following complexities:

a. O(k).

b. O(n).

c. O(log n).

d. O$(n$ log $n)$.

e. O(n^2).

15.2 (Complexity, 10 min.) In a film the main character looks 12 moves ahead to win a chess championship.

a. Estimate (stating your assumptions) the number of possible chess moves looking 12 moves ahead.

b. How long would it take to evaluate all these moves on a modern workstation?

15.3 (Chips and towns, 20 min.) This problem is adapted from an analogy credited to Chuck Seitz. Complete the entries in Table 15.8, which shows the progression of integrated circuit complexity using the analogy of town and city planning. If λ is half the minimum feature size, assume that a transistor is a square 2λ on a side and is equivalent to a city block (which we estimate at 200 m on a side).

TABLE 15.8 Complexity of ASICs (Problems 15.3 and 15.4).

Year	$\lambda/\mu m$	Chip size (mm on a side)	Transistor size (μm on a side)	Transistors = city blocks	City size (km on a side, 1 block = 200m)	Example
1970	50	5	200	$25 \times 25 = 625$	5	Palo Alto
1980	5	10	20	$500 \times 500 = 25 \times 10^3$		
1990	0.5	20	1	$1{,}000 \times 1{,}000 = 1 \times 10^6$		
2000	0.05	40	0.2	$20{,}000 \times 20{,}000 = 400 \times 10^6$		

15.4 (Polygons, 10 min.) Estimate (stating and explaining all your assumptions) how many polygons there are on the layouts for each of the chips in Table 15.8.

15.5 (Algorithm complexity, 10 min.) I think of a number between 1 and 100. You guess the number and I shall tell you whether you are high or low. We then repeat the process. If you were to write a computer program to play this game, what would be the complexity of your algorithm?

15.6 (Algorithms, 60 min.) For each of these problems write or find (stating your source) an algorithm to solve the problem:

a. An algorithm to sort n numbers.

b. An algorithm to discover whether a number n is prime.

c. An algorithm to generate a random number between 1 and n.

List the algorithm using a sequence of steps, pseudocode, or a flow chart. What is the complexity of each algorithm?

15.7 (Measurement, 30 min.) The traveling-salesman problem is a well-known example of an NP-complete problem (you have a list of cities and their locations and you have to find the shortest route between them, visiting each only once). Propose a simple measure to estimate the length of the solution. If I had to visit the 50 capitals of the United States, what is your estimate of my frequent-flyer mileage?

15.8 (Construction, 30 min.) Try and make a quantitative comparison (stating and explaining all your assumptions) of the difficulty and complexity of construction (for example, how many components in each?) for each of the following: a Boeing 747 jumbo jet, the space shuttle, and an Intel Pentium microprocessor. Which, in your estimation, is the most complex and why? Smailagic [1995] proposes measures of design and construction complexity in a description of the wearable computer project at Carnegie-Mellon University.

15.9 (Productivity, 20 min.). If I have six months to design an ASIC:

a. What is the productivity (in transistors/day) required for each of the chips in Table 15.8?

b. What does this translate to in terms of a productivity increase (measured in percent increase in productivity per month)?

c. Moore's Law says that chip sizes double every 18 months. What does this correspond to in terms of a percentage increase per month?

d. Comment on your answers.

15.10 (Graphs and edges, 30 min.) We know a net with two connections requires a single edge in the network graph, a net with three connections requires three edges, and a net with four connections requires six edges.

a. Can you guess a formula for the number of edges in the network graph corresponding to a net with n connections?

b. Can you prove the formula you guessed in part a? *Hint:* How many edges are there from one node to $n-1$ other nodes?

c. Large nets cause problems for partitioning algorithms based on a connectivity matrix (edges rather than wires). Suppose we have a 50-net connection that is no more critical for timing than any other net. Suggest a way to fool the partitioning algorithm so this net does not drag all its logic cells into one partition.

Most CAD programs treat large nets (like the clock, reset, or power nets) separately, but the nets are required to have special names and you only can have a limited number of them. The average net in an ASIC has between two and four connections and as a rule of thumb 80 percent of nets have a fanout of 4 or less (a fanout of 4 means a gate drives four others, making a total of five connections on the net).

15.11 (PC partitioning, 60 min.) Open an IBM-compatible PC, Apple Macintosh, or PowerPC that has a motherboard that you can see easily. Make a list of the chips (manufacturer and type), their packages, and pin counts. Make intelligent guesses as to the function of most of the chips. Obviously manufacturer's logos and chip identification markings help—perhaps they are in a data book. Identify the types of packages (pin-grid array, quad flat pack). Look for nearby components that may give a hint—crystals for clock generators or the video subsystem. Where are the chips located on the board—are they near the connectors for the floppy disk subsystem, the modem or serial port, or video output? To help you, Table 15.9 shows an example—a list of the first row of chips on an old H-P Vectra ES/12 motherboard. Use the same format for your list.

TABLE 15.9 A list of the chips on the first row of an HP Vectra PC (Problem 15.11).

Manufacturer	Chip	Package	Function	Comment
HP	87411AAE	24-pin DIP		
Intel	L7220048	40-pin DIP	EPROM (9/3/87)	Boot commands
Chips	7014-0093	80-pin quad flat pack	Custom ASIC	
Intel	80286-12	68-pin package	Microprocessor	CPU
TI	AS00	14-pin DIP	Quad 2-input NAND gate	Addressing
	S74F08D	14-pin DIP	Quad 2-input AND gate	Addressing
	F74F51	14-pin DIP	AOI gate	Addressing

15.12 (Estimates, 60 min.) System partitioning is not exact science. Estimate:

a. The power developed by a grasshopper, in watts (from a Cambridge University entrance exam).

b. The number of doors in New York City.

c. The number of grains of sand on Hawaii's beaches.

d. The total length of the roads in the continental United States in kilometers.

In each case: (i) Provide an equation that depends on parameters and symbols that you define. (ii) List the parameters in your equation, and the values that you assume with their uncertainty. (iii) Give the answer as a number (with units where necessary). (iv) Include a numerical estimate of the uncertainty in your answer.

15.13 (Pad-limited and core-limited die, 10 min.) As the number of I/O pads increases, an ASIC can become **pad-limited**. The spacing between I/O pads is determined by mechanical limitations of the equipment used for bonding—usually 2–5 mil (a mil is a thousandth of an inch). In a pad-limited design the number of pads around the outer edge of the die determines the die size, not the number of gates (see Figure 15.12). For the pad-limited design, shown in Figure 15.12(a), the price per I/O pad is more important than the price per gate. When we have a lot of logic but few I/O pads, we have a **core-limited** design—the opposite of a pad-limited ASIC—as shown in Figure 15.12(b). For a given number of I/O pads and a pad-limited design, all the different ASIC types will have the same die size, determined by a graph such as the one shown in Figure 15.12(c). If I/O pad spacing is 5 mil and gate density is 1.0 gate/mil^2, when does an ASIC becomes pad-limited? Express your answer as a function of the number of gates, G, and the number of I/Os, I.

15.18 (Connectivity matrix, 10 min.) Find the connectivity matrix for the ATM Connection Simulator shown in Figure 15.5. Use the following scheme to number the blocks and ordering of the matrix rows and columns: 1 = Personal Computer, 2 = Intel 80186, 3 = UTOPIA receiver, 4 = UTOPIA transmitter, 5 = Header remapper and screener, 6 = Remapper SRAM, . . . 15 = Random-number and bit error rate generator, 16 = Random-variable generator. All buses are labeled with their width except for two single connections (the arrows).

15.19 (K–L algorithm, 15 min.)

a. Draw the network graph for the following connectivity matrix:

$$\mathbf{C} = \begin{bmatrix} 0&0&0&0&0&0&1&0&0&0 \\ 0&0&0&0&0&1&0&1&0&0 \\ 0&0&0&1&0&0&0&1&0&0 \\ 0&0&1&0&1&0&0&0&1&0 \\ 0&0&0&1&0&0&0&0&0&0 \\ 0&1&0&0&0&0&0&0&1&0 \\ 1&0&0&0&0&0&0&0&1&0 \\ 0&1&1&0&0&0&0&0&1&0 \\ 0&0&0&1&0&0&1&1&0&1 \\ 0&0&0&0&0&0&0&0&1&0 \end{bmatrix} \qquad (15.26)$$

b. Draw the partitioned network graph for **C** with nodes 1–5 in partition A and nodes 6–10 in partition B. What is the cut weight?

c. Improve the initial partitioning using the K–L algorithm. Show the gains at each stage. What problems did you find in following the algorithm and how do you resolve them?

15.20 (The gain graph in the K–L algorithm, 20 min.). Continue with the K–L algorithm for the network that we started to partition in Figure 15.9(a).

a. Show that choices of logic cells to swap and the gains correspond to the graph of Figure 15.9(b).

b. Notice that $G_5 = 0$. In fact G_m (where there are $2m$ nodes to be partitioned) will always be zero. Can you explain why?

15.21 (Look-ahead gain in the K–L algorithm, 20 min.) In the K–L algorithm we have to compute the gain each time we consider swapping one pair of nodes:

$$g_I = D_a + D_b - 2c_{ab} . \qquad (15.27)$$

If we swap two pairs of nodes (a_1 and b_1 followed by a_2 and b_2), show that the gain is

$$g_{II} = D_{a_2} + D_{b_2} - 2c_{a_2 b_2} + 2c_{a_2 a_1} - 2c_{a_2 b_1} - 2c_{b_2 a_1} + 2c_{b_2 b_1} . \qquad (15.28)$$

15.22 (FPGA partitioning, 30 min.) Table 15.10 shows some data on FPGAs from company Z.

TABLE 15.10 FPGAs from company Z (Problem 15.22).

FPGA size	Die area/cm^2	Average gate count	Package pins	Cost
S	0.26	1500	68	\$26
M	0.36	2300	44	\$35
L	0.46	2800	84	\$50
XL	0.64	4700	84	\$90
XXL	0.84	6200	84	\$130

a. Notice that the FPGAs come in different package sizes. To eliminate the effect of package price, multiply the price for the S chip by 106 percent, and the M chip by 113 percent. Now all prices are normalized for an 84-pin plastic package. All the chips are the same speed grade; if they were not, we could normalize for this too (a little harder to justify though).

b. Plot the normalized chip prices vs. gate count. What is the cost per gate?

c. The part cost ought to be related to the yield, which is directly related to die area. If the cost of a 6-inch-diameter wafer is fixed (approximately \$1000), calculate the cost per die, assuming a yield Y (in percent), as a function of the die area, A (in cm^2). Assume you completely fill the wafer and you can have fractional die (i.e., do not worry about packing square die into a circular wafer).

d. There are many models for the yield of a process, Y. Two common models are

$$Y = \exp\left(-\sqrt{AD}\right) \tag{15.29}$$

and

$$Y = \left(\frac{1-\exp(-AD)}{AD}\right)^2 . \tag{15.30}$$

Parameter A is the die area in cm^2 and D is the spot defect density in defects/cm^2 and is usually around 1.0 defects/cm^2 for a good submicron CMOS process (above 5.0 defects/cm^2 is unusual). The most important thing is the yield; anything below about 50 percent good die per wafer is usually bad news for an ASIC foundry. Does the FPGA cost data fit either model?

e. Now disregard the current pricing strategy of company Z. If you had to bet that physics would determine the true price of the chip, how much worse or better off are you using two small FPGAs rather than one larger FPGA (assume the larger die is exactly twice the area of the smaller one) under these two yield models?

f. What assumptions are inherent in the calculation you made in part e? How much do you think they might affect your answer, and what else would affect your judgment?

g. Give some reasons why you might select two smaller FPGAs rather than a larger FPGA, even if the larger FPGA is a cheaper solution.

h. Give some reasons why you would select a larger FPGA rather than two smaller FPGAs, even if the smaller FPGAs were a cheaper solution.

15.23 (Constructive partitioning, 30 min.) We shall use the simple network with 12 blocks shown in Figure 15.7 to experiment with constructive partitioning. This example is topologically equivalent to that used in [Goto and Matsud, 1986].

a. We shall use a gain function, *g(m)*, calculated as follows: Sum the number of the *nets* (not *connections*) from the selected logic cell, *m*, that connect to the current partition—call this *P(m)*. Now calculate the number of *nets* that connect logic cell *m* to logic cells which are not yet in partitions—call this *N(m)*. Then *g(m)* = *P(m)* – *N(m)* is the gain of adding the logic cell *m* to the partition currently being filled.

b. Partition the network using the seed growth algorithm with logic cell C as the seed. Show how this choice of seed can lead to the partitioning shown in Figure 15.7(c). Use a table like Table 15.11 as a bookkeeping aid (a spreadsheet will help too). Each row corresponds to a pass through the algorithm. Fill in the measures, *P(m)* – *N(m)*, equal to the gain, *g(m)*. Once a logic cell is assigned to a partition, fill in the name of the partition (X, Y, or Z) in that column. The first row shows you how logic cell L is selected; proceed from there. What problems do you encounter while completing the algorithm, and how do you resolve them?

c. Now partition using logic cell F as the seed instead—the logic cell with the highest number of nets. When you have a tie between logic cells with the same gain, or you are starting a new partition, pick the logic cell with the largest *P(m)*. Use a copy of Table 15.12 as a bookkeeping aid. How does your partition compare with those we have already made (summarized in Table 15.13)?

d. Comment on your results.

Table 15.14 will help in constructing the gain function at each step of the algorithm.

15.24 (Simulated annealing, 15 min.) If you have a fixed amount of time to solve a partitioning problem, comment on the following alternatives and choose one:

i. Run a single simulated annealing cycle using a slow cooling schedule.

TABLE 15.11 Bookkeeping table for Problem 15.23 (b).

Pass	Gain	A	B	C	D	E	F	G	H	I	J	K	L
1	*P*–*N*	0–2	1–2	X	0–2	1–4	0–5	0–2	0–2	0–3	0–3	0–2	0–1
	=*g*	= –2	= –1		= –2	= –3	= –5	= –2	= –2	= –3	= –3	= –2	= –1
2				X							X		X

TABLE 15.12 Bookkeeping table for Problem 15.23 (c).

Pass	Gain	A	B	C	D	E	F	G	H	I	J	K	L
1	*P*–*N*	1–2	0–2	1–1	1–1	1–3	X	0–2	1–2	2–2	0–3	0–2	0–1
	=*g*	=–1	=–2	=0	=0	=–2		=–2	=–1	=0	=–3	=–2	=–1
2							X			X			

TABLE 15.13 Different partitions for the network shown in Figure 15.7 (Problem 15.23 c).

Partitioning	Total external connections	Partition contents X, Y, Z	Connections to each partition
Figure 15.7(b)	5	X=(A, B, C, L)	3
	(2, 4, 5, 6, 8)	Y=(D, F, H, I)	3
		Z=(E, G, J, K)	4
Figure 15.7(c)	7	X=(A, B, F, D)	5
	(2, 3, 5, 7, 9, 11, 12)	Y=(H, I, J, K)	5
		Z=(C, E, G, L)	4

ii. Run several (faster) min-cut based partitionings, using different seeds, and pick the best one.

iii. Run several simulated annealing cycles using a faster cooling schedule, and pick the best result.

15.25 (Net weights, 15 min.) Figure 15.13 shows a small part of a system and will help illustrate some potential problems when you weight nets for partitioning. Nets s1–s3 are critical, nets c1–c4 are not. Assume that all nets are weighted by a cost of one unless the special net weight symbol is attached.

a. Explain the problem with the net weights as shown in Figure 15.13(a).

b. Figure 15.13(b) shows a different way to assign weights. What problems might this cause in the rest of the system?

TABLE 15.14 An aid to calculating the gains for Problem 15.23.

Logic cell	Connects to:	Number of nets	Number of connections
A	B, F	2	2
B	A, (C, E)	2	3
C	(B, E)	1	2
D	F, H	2	2
E	(B, C), F, (G, L), J	4	6
F	A, D, E, (H, I_1), I_2	5	6
G	(E, L), (J, K)	2	4
H	D, (F, I)	2	3
I	F_1, (F_2, H), (J, K)	3	5
J	E, (G, K_1), (I, K_2)	3	5
K	(G, J_1), (I, J_2)	2	4
L	(E, G)	1	2

c. Figure 15.13(c) shows another possible solution. Discuss the advantages of this approach.

d. Can you think of another way to solve the problem?

This situation represents a very real problem with using net weights and tools that use min-cut algorithms. As soon as you get one critical net right, the tool makes several other nets too long and they become critical. The problem is worse during system partitioning when the blocks are big and there are many different nets with differing importance attached to each block—but it can happen during floorplanning and placement also.

15.26 (Cost, 60 min.) You have three chip sizes available for your part of project "DreamOn" (a new video game): S, M, and L. The L chip has twice the logic of the M chip. The M chip has twice the logic of the S chip. The L chip costs $16, which is 4 times as much as the M chip and 16 times as much as the S chip. There are two speed grades available: fast (F) and turbocharged (T). The T chip costs twice as much as the F version. Using a partitioning program, you find you need the equivalent of 1.8 of the L chips, but only a third of your logic needs a T chip.

a. What is the cheapest way to build "DreamOn"?

b. During prototyping you find you can use 90 percent of the S and M type chips, but for reliable routing you can only count on a maximum utilization of 85 percent for the L chip. You also find that, to maximize performance, you need to keep all of the logic that requires the turbo speed on one chip.

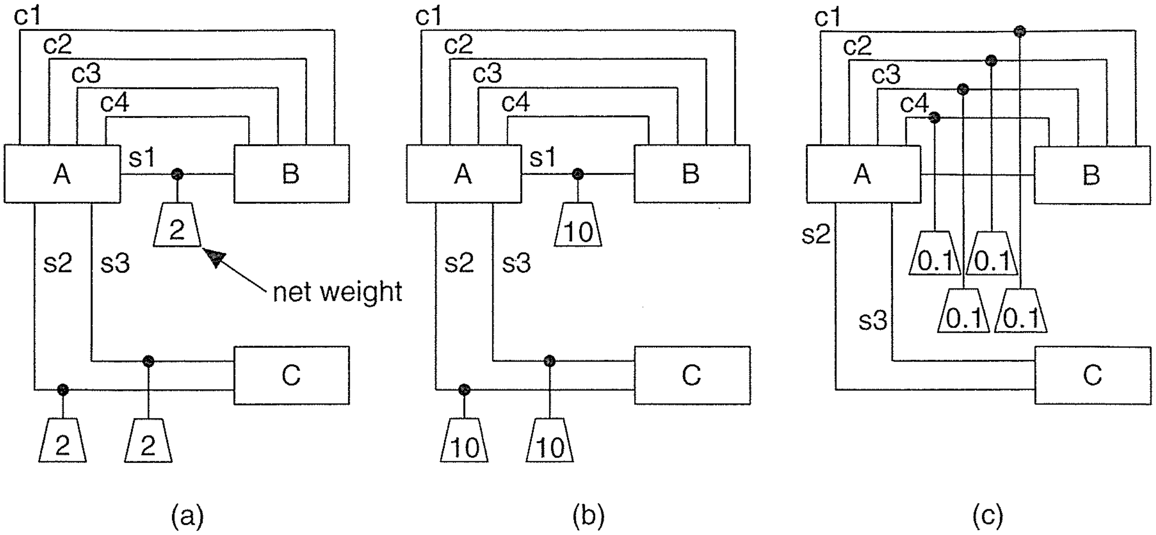


FIGURE 15.13 (For Problem 15.25.) An example of a problem in weighting nets. The symbols attached to the nets apply a weight or cost to that net during partitioning. Nets c1–c4 are control lines—they are not critical for timing purposes. Nets s1–s3 are signal lines that are critical—they must be kept short. The figure shows three different ways to handle this using net weights.

Our ASIC vendor, Xactera, promises us that the chip prices will fall by the time we go into production in one year. The estimates are that the prices will be almost proportional to chip size: The L chip will cost 2.2 times the M chip and 4.4 times whatever is the cost of the S chip by then (but Xactera will not commit to a future price for the S chip, only the present price). You predict the price of the S chip will fall 20 percent in one year (this is about average for the annual rate of price decrease for semiconductors). Xactera says the turbocharged speed grade will stay about twice the cost of the fast grade. How does this information affect your decision?

c. Some time later, as you are ready to go on vacation, the production department tells you that the board cost is about the same as the chip cost! The board area does not make much difference to the price, but there is an extra charge per package pin to reflow solder the surface-mount chips. We only need each chip to have the minimum size package—a 44-pin quad plastic package. Production has two price quotes: Boards-R-Us charges $5 per board plus $0.01 per pin, and PCB Inc. quotes at $0.05 per pin. What should we do? The CEO needs a recommendation today.

d. You come back from holiday and find out from your e-mail that we went with your recommendation on the board vendor but now we have other problems. The test company is charging per chip pin on the board since we are using an old style bed-of-nails tester. The cost is about $0.01 per chip pin. You can go back and add a test interface to all the chips, which is the equivalent of adding 10 percent of a small chip (type S) on each chip (S, M, or L). This would eliminate the bed-of-nails test, and reduce board test cost to $1 per board. Xactera also just lowered their prices: L chips are now $4, M chips are $2, and S chips are $0.95. There is also a new Xactera XL chip that has twice the capacity of the L chips and costs $8 (but you do not know what utilization to expect). These prices are for the fast speed grades, the turbo versions are now 2.5 times more expensive.

e. There are some serious consequences to making any design changes now (including schedule slips). We have an emergency meeting with production, finance, marketing, and the CEO this afternoon in the boardroom. I have to prepare a presentation outlining our past decisions and the advantages and disadvantages of each of our options (with quantitative estimates of their effect). Can you prepare four foils for me, and a one-page spreadsheet that will allow us to make some rapid "what-if" decisions in the meeting? Print the foils and the one-page spreadsheet.

f. A year later we are in full production and all is well. We are reviewing your performance on project "DreamOn." What did you learn from this project and how would you do things differently next time? (You only have room for 100 words on your review form.)

15.10 Bibliography

Many of the references in the bibliography in Chapter 1 are also sources for information on the physical design of ASICs. The European Conference on Design Automation is known as EuroDAC (TK7867.E93, ISBN and cataloging varies with year). Another European conference, EuroASIC, was absorbed by EuroDAC (TK7874.6.E88, ISSN 1066-1409 and ISSN 1064-5322, cataloging varies).

Preas and Lorenzetti's book [1988] contains an overview chapter on partitioning and placement. To dig a little deeper see the review article by Goto and Matsud [1986]. If you want to explore further the detailed workings of partitioning algorithms, Sherwani's book [1993] catalogs physical design algorithms, including those for partitioning. To learn more about simulated annealing see Sechen's book [1988]. Partitioning is an important part of high-level synthesis, and the book by Gajski et al. [1992] contains a chapter on partitioning for allocation and scheduling as well as system partitioning—including a description of clustering methods, which are not well covered elsewhere.This book describes SpecSyn, a tool that allows you to enter a design using a behavioral description with a graphical tool. SpecSyn can then partition the design given area, timing, and cost specifications. System partitioning at the behavioral level (**architectural partitioning**) is an area of current research (see [Lagnese and Thomas, 1991] for a description of the APARTY system). This means we partition a design based on a hardware design language rather than a schematic or other physical description. Papers published in the *Proceedings of the Design Automation Conference* (DAC) and articles in the *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* form a point at which to start working back through the recent research literature on system partitioning, an example is [Kucukcakar and Parker, 1991].

The *Proceedings of the 32nd Design Automation Conference* (1995) describe a special session on the design of the Sun Microsystems UltraSPARC-I (albeit from more of a systems perspective), which forms an interesting comparison to the SPARCstation 1 and SPARCstation 10 designs.

15.11 References

Page numbers in brackets after the reference indicate the location in the chapter body.

Cheng, C.-K., and Y.-C. A. Wei. 1991. "An improved two-way partitioning algorithm with stable performance." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* Vol. 10, no. 12, pp. 1502–1511. Describes the ratio-cut algorithm. [p. 834]

Fiduccia, C. M., and R. M. Mattheyses. 1982. "A linear-time heuristic for improving network partitions." In *Proceedings of the 19th Design Automation Conference,* pp. 175–181. Describes modification to Kernighan-Lin algorithm to reduce computation time. [p. 833]

Gajski, D. D., N. D. Dutt, A. C.-H. Wu, and S. Y.-L. Lin. 1992. *High-Level Synthesis: Introduction to Chip and System Design.* Norwell, MA: Kluwer. ISBN 0-7923-9194-2. TK7874.H52422. Chapter 6, Partitioning, is an introduction to system-level partitioning algorithms. It also includes a description of the system partitioning features of SpecSyn, a research tool developed at UC-Irvine. [p. 850]

Goto, S., and T. Matsud. 1986. "Partitioning, assignment and placement." In *Layout Design and Verification.* Vol. 4 of *Advances in CAD for VLSI* (T. Ohtsuki, Ed.) pp. 55–97, New York: Elsevier. [p. 826]

Kernighan, B. W., and S. Lin. 1970. "An efficient heuristic procedure for partitioning graphs." *Bell Systems Technical Journal,* Vol. 49, no. 2, February, pp. 291–307. The original description of the Kernighan–Lin partitioning algorithm. [p. 828]

Kirkpatrick, S., et al. 1983. "Optimization by simulated annealing." *Science,* Vol. 220, no. 4598, pp. 671–680. [p. 836]

Kucukcakar, K., and A. C. Parker, 1991. "CHOP: A constraint-driven system-level partitioner." In *Proceedings of the 28th Design Automation Conference,* pp. 514–519. [p. 851]

Lagnese, E., and D. Thomas. 1991. "Architectural partitioning for system level synthesis of integrated circuits." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* Vol. 10, no. 7, pp. 847–860. [p. 850]

Najm, F. N. 1994. "A survey of power estimation techniques in VLSI circuits." *IEEE Transactions on Very Large Scale Integration (VLSI) Systems,* Vol. 2, no. 4, pp. 446–455. 43 refs. [p. 817]

Preas, B. T., and P. G. Karger, 1988. "Placement, assignment and floorplanning." In *Physical Design Automation of VLSI Systems* (B. T. Preas and M. J. Lorenzetti, Eds.), pp. 87–155. Menlo Park, CA: Benjamin-Cummings. ISBN 0-8053-0412-9. TK7874.P47. [p. 850]

Rose, J., W. Klebsch, and J. Wolf, 1990. "Temperature measurement and equilibrium dynamics of simulated annealing placements." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* Vol. 9, no. 3, pp. 253–259. Discusses ways to speed up simulated annealing. [p. 837]

Schweikert, D. G., and B. W. Kernighan. 1979. "A proper model for the partitioning of electrical circuits." In *Proceedings of the 9th Design Automation Workshop.* Points out the difference between nets and edges. [pp. 831, 831]

Sechen, C. 1988. *VLSI Placement and Global Routing Using Simulated Annealing.* New York: Kluwer. Introduction; The Simulated Annealing Algorithm; Placement and Global Routing of Standard Cell Integrated Circuits; Macro/Custom Cell Chip-Planning, Placement, and Global Routing; Average Interconnection Length Estimation; Interconnect-Area Estimation for Macro Cell Placements; An Edge-Based Channel Definition Algorithm for Rectilinear Cells; A Graph-Based Global Router Algorithm; Conclusion; Island-Style Gate Array Placement. [p. 850]

Sedgewick, R. 1988. *Algorithms.* Reading, MA: Addison-Wesley. ISBN 0-201-06673-4. QA76.6.S435. Reference for basic sorting and graph-searching algorithms. [p. 808]

Sherwani, N. A. 1993. *Algorithms for VLSI Physical Design Automation.* Norwell, MA: Kluwer. ISBN 0-7923-9294-9. TK874.S455. [p. 850]

Smailagic, A., et al. 1995. "Benchmarking an interdisciplinary concurrent design methodology for electronic/mechanical systems." In *Proceedings of the 32nd Design Automation Conference.* San Francisco. Describes the evolution of the VuMan wearable computer. Includes some interesting measures of the complexity of system design. [p. 840]

Veendrick, H. J. M. 1984. "Short-circuit dissipation of static CMOS circuitry and its impact on the design of buffer circuits." *IEEE Journal of Solid-State Circuits,* Vol. SC-19, no. 4, pp. 468–473. [pp. 817, 843]

FLOORPLANNING AND PLACEMENT 16

The input to the floorplanning step is the output of system partitioning and design entry—a netlist. Floorplanning precedes placement, but we shall cover them together. The output of the placement step is a set of directions for the routing tools.

At the start of floorplanning we have a netlist describing circuit blocks, the logic cells within the blocks, and their connections. For example, Figure 16.1 shows the Viterbi decoder example as a collection of standard cells with no room set aside yet for routing. We can think of the standard cells as a hod of bricks to be made into a wall. What we have to do now is set aside spaces (we call these spaces the **channels**) for interconnect, the mortar, and arrange the cells. Figure 16.2 shows a finished wall—after floorplanning and placement steps are complete. We still have not completed any routing at this point—that comes later—all we have done is placed the logic cells in a fashion that we hope will minimize the total interconnect length, for example.

16.1 Floorplanning

Figure 16.3 shows that both interconnect delay and gate delay decrease as we scale down feature sizes—but at different rates. This is because interconnect capacitance tends to a limit of about 2 pFcm^{-1} for a minimum-width wire while gate delay continues to decrease (see Section 17.4, "Circuit Extraction and DRC"). Floorplanning allows us to predict this interconnect delay by estimating interconnect length.

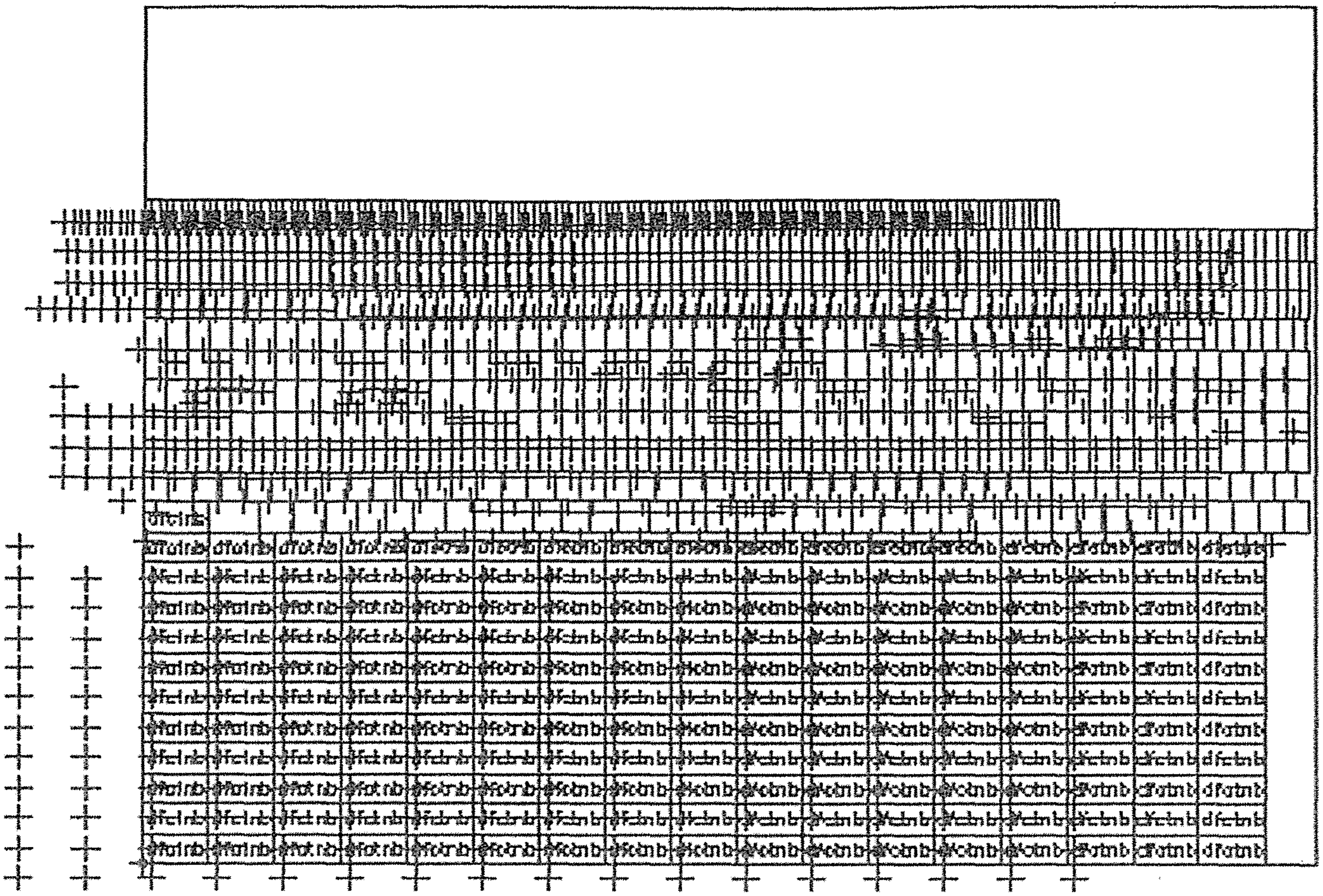


FIGURE 16.1 The starting point for the floorplanning and placement steps for the Viterbi decoder (containing only standard cells). This is the initial display of the floorplanning and placement tool. The small boxes that look like bricks are the outlines of the standard cells. The largest standard cells, at the bottom of the display (labeled dfctnb) are 188 D flip-flops. The '+' symbols represent the drawing origins of the standard cells—for the D flip-flops they are shifted to the left and below the logic cell bottom left-hand corner. The large box surrounding all the logic cells represents the estimated chip size. (This is a screen shot from Cadence Cell Ensemble.)

16.1.1 Floorplanning Goals and Objectives

The input to a floorplanning tool is a hierarchical netlist that describes the interconnection of the **blocks** (RAM, ROM, ALU, cache controller, and so on); the logic cells (NAND, NOR, D flip-flop, and so on) within the blocks; and the logic cell connectors (the terms *terminals*, *pins*, or *ports* mean the same thing as *connectors*). The netlist is a logical description of the ASIC; the floorplan is a physical description of an ASIC. Floorplanning is thus a mapping between the logical description (the netlist) and the physical description (the floorplan).

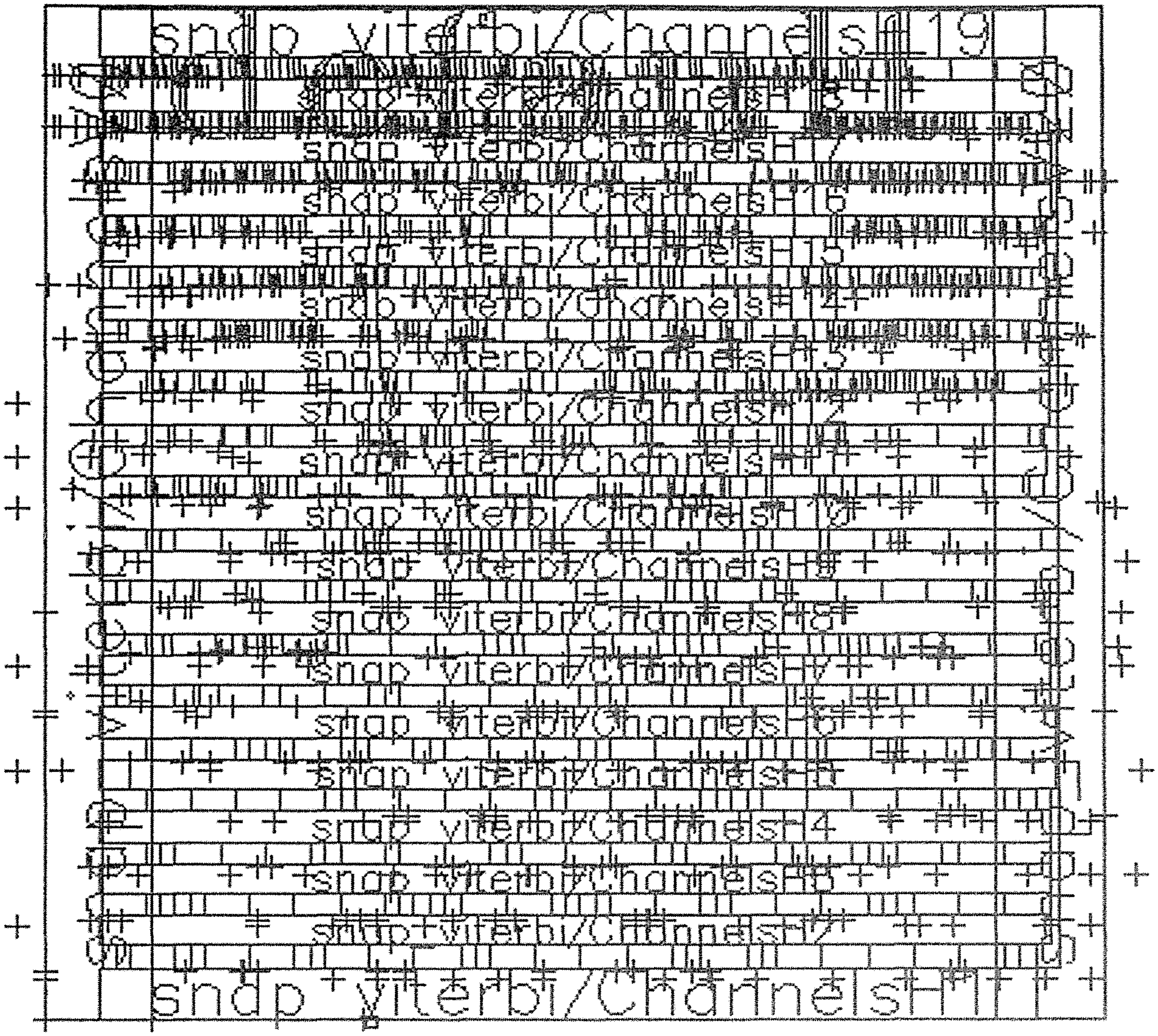


FIGURE 16.2 The Viterbi Decoder (from Figure 16.1) after floorplanning and placement. There are 18 rows of standard cells separated by 17 horizontal channels (labeled 2–18). The channels are routed as numbered. In this example, the I/O pads are omitted to show the cell placement more clearly. Figure 17.1 shows the same placement without the channel labels. (A screen shot from Cadence Cell Ensemble.)

The goals of floorplanning are to:

- arrange the blocks on a chip,
- decide the location of the I/O pads,
- decide the location and number of the power pads,

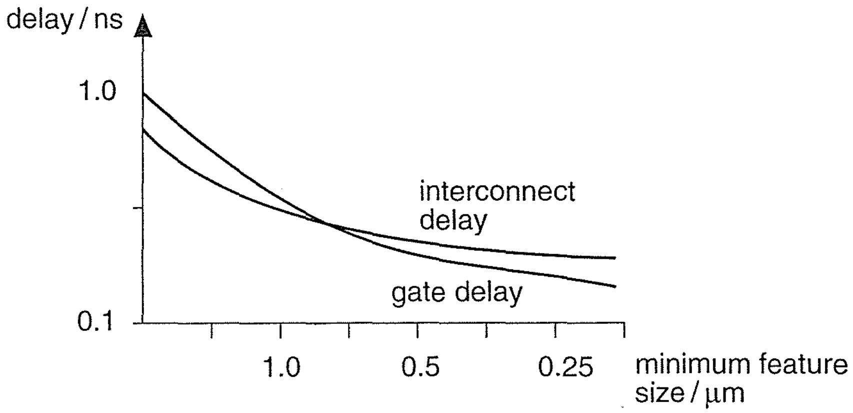


FIGURE 16.3 Interconnect and gate delays. As feature sizes decrease, both average interconnect delay and average gate delay decrease—but at different rates. This is because interconnect capacitance tends to a limit that is independent of scaling. Interconnect delay now dominates gate delay.

- decide the type of power distribution, and
- decide the location and type of clock distribution.

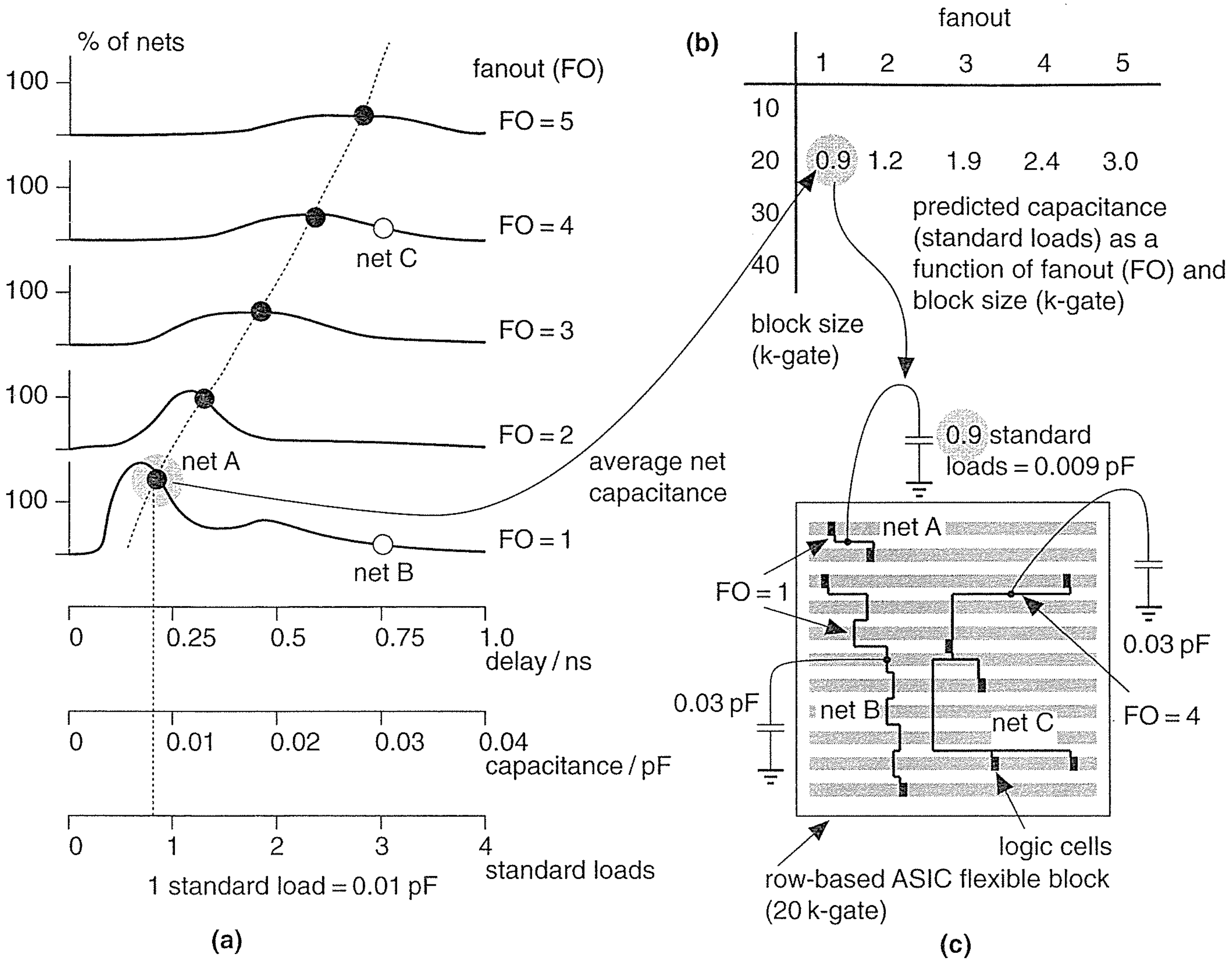
The objectives of floorplanning are to minimize the chip area and minimize delay. Measuring area is straightforward, but measuring delay is more difficult and we shall explore this next.

16.1.2 Measurement of Delay in Floorplanning

Throughout the ASIC design process we need to predict the performance of the final layout. In floorplanning we wish to predict the interconnect delay before we complete any routing. Imagine trying to predict how long it takes to get from Russia to China without knowing where in Russia we are or where our destination is in China. Actually it is worse, because in floorplanning we may move Russia or China.

To predict delay we need to know the **parasitics** associated with interconnect: the **interconnect capacitance** (**wiring capacitance** or **routing capacitance**) as well as the interconnect resistance. At the floorplanning stage we know only the **fanout** (**FO**) of a net (the number of gates driven by a net) and the size of the block that the net belongs to. We cannot predict the resistance of the various pieces of the interconnect path since we do not yet know the shape of the interconnect for a net. However, we can estimate the total length of the interconnect and thus estimate the total capacitance. We estimate interconnect length by collecting statistics from previously routed chips and analyzing the results. From these statistics we create tables that predict the interconnect capacitance as a function of net fanout and block size. A floorplanning tool can then use these **predicted-capacitance tables** (also known as **interconnect-load tables** or **wire-load tables**). Figure 16.4 shows how we derive and use wire-load tables and illustrates the following facts:

- Typically between 60 and 70 percent of nets have a FO = 1.
- The distribution for a FO = 1 has a very long tail, stretching to interconnects that run from corner to corner of the chip.



	1	2	3	4	5
10					
20	0.9	1.2	1.9	2.4	3.0
30					
40					

FIGURE 16.4 Predicted capacitance. (a) Interconnect lengths as a function of fanout (FO) and circuit-block size. (b) Wire-load table. There is only one capacitance value for each fanout (typically the average value). (c) The wire-load table predicts the capacitance and delay of a net (with a considerable error). Net A and net B both have a fanout of 1, both have the same predicted net delay, but net B in fact has a much greater delay than net A in the actual layout (of course we shall not know what the actual layout is until much later in the design process).

- The distribution for a FO = 1 often has two peaks, corresponding to a distribution for close neighbors in subgroups within a block, superimposed on a distribution corresponding to routing between subgroups.
- We often see a twin-peaked distribution at the chip level also, corresponding to separate distributions for **interblock routing** (inside blocks) and **intrablock routing** (between blocks).

- The distributions for FO > 1 are more symmetrical and flatter than for FO = 1.
- The wire-load tables can only contain one number, for example the average net capacitance, for any one distribution. Many tools take a worst-case approach and use the 80- or 90-percentile point instead of the average. Thus a tool may use a predicted capacitance for which we know 90 percent of the nets will have less than the estimated capacitance.
- We need to repeat the statistical analysis for blocks with different sizes. For example, a net with a FO = 1 in a 25 k-gate block will have a different (larger) average length than if the net were in a 5 k-gate block.
- The statistics depend on the shape (aspect ratio) of the block (usually the statistics are only calculated for square blocks).
- The statistics will also depend on the type of netlist. For example, the distributions will be different for a netlist generated by setting a constraint for minimum logic delay during synthesis—which tends to generate large numbers of two-input NAND gates—than for netlists generated using minimum-area constraints.

There are no standards for the wire-load tables themselves, but there are some standards for their use and for presenting the extracted loads (see Section 16.4). Wire-load tables often present loads in terms of a **standard load** that is usually the input capacitance of a two-input NAND gate with a 1X (default) drive strength.

TABLE 16.1 A wire-load table showing average interconnect lengths (mm).[1]

		Fanout		
Array (available gates)	Chip size (mm)	1	2	4
3 k	3.45	0.56	0.85	1.46
11 k	5.11	0.84	1.34	2.25
105 k	12.50	1.75	2.70	4.92

[1] Interconnect lengths are derived from interconnect capacitance data. Interconnect capacitance is 2 pFcm^{-1}.

Table 16.1 shows the estimated metal interconnect lengths, as a function of die size and fanout, for a series of three-level metal gate arrays. In this case the interconnect capacitance is about 2 pFcm^{-1}, a typical figure.

Figure 16.5 shows that, because we do not decrease chip size as we scale down feature size, the worst-case interconnect delay increases. One way to measure the worst-case delay uses an interconnect that completely crosses the chip, a **coast-to-coast interconnect**. In certain cases the worst-case delay of a 0.25 μm process may be worse than a 0.35 μm process, for example.

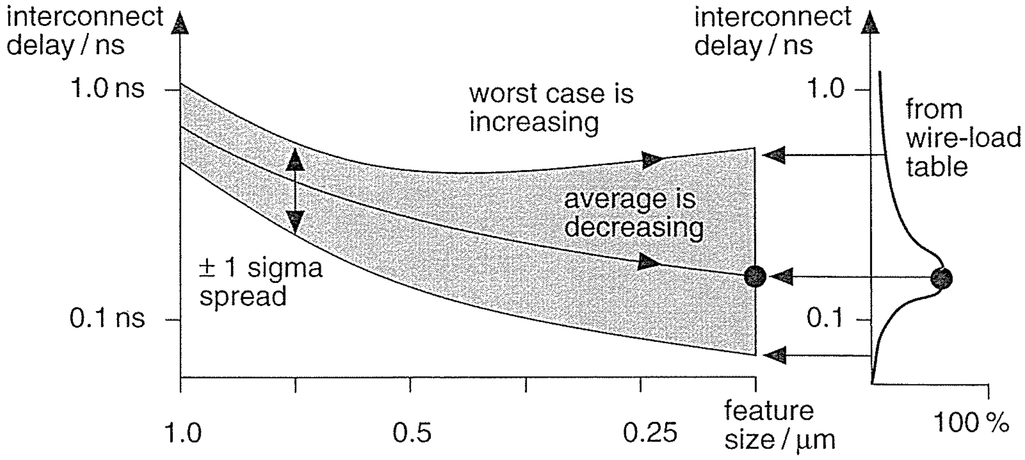


FIGURE 16.5 Worst-case interconnect delay. As we scale circuits, but avoid scaling the chip size, the worst-case interconnect delay increases.

16.1.3 Floorplanning Tools

Figure 16.6(a) shows an initial **random floorplan** generated by a floorplanning tool. Two of the blocks, A and C in this example, are standard-cell areas (the chip shown in Figure 16.1 is one large standard-cell area). These are **flexible blocks** (or **variable blocks**) because, although their total area is fixed, their shape (aspect ratio) and connector locations may be adjusted during the placement step. The dimensions and connector locations of the other **fixed blocks** (perhaps RAM, ROM, compiled cells, or megacells) can only be modified when they are created. We may force logic cells to be in selected flexible blocks by **seeding**. We choose **seed cells** by name. For example, `ram_control*` would select all logic cells whose names started with `ram_control` to be placed in one flexible block. The special symbol, usually '*', is a **wildcard symbol**. Seeding may be hard or soft. A **hard seed** is fixed and not allowed to move during the remaining floorplanning and placement steps. A **soft seed** is an initial suggestion only and can be altered if necessary by the floorplanner. We may also use **seed connectors** within flexible blocks—forcing certain nets to appear in a specified order, or location at the boundary of a flexible block.

The floorplanner can complete an estimated placement to determine the positions of connectors at the boundaries of the flexible blocks. Figure 16.6(b) illustrates a **rat's nest** display of the connections between blocks. Connections are shown as **bundles** between the centers of blocks or as **flight lines** between connectors. Figure 16.6(c) and (d) show how we can move the blocks in a floorplanning tool to minimize routing **congestion**.

We need to control the **aspect ratio** of our floorplan because we have to fit our chip into the **die cavity** (a fixed-size hole, usually square) inside a package. Figure 16.7(a)–(c) show how we can rearrange our chip to achieve a square aspect ratio. Figure 16.7(c) also shows a **congestion map**, another form of **routability** display. There is no standard measure of routability. Generally the **interconnect channels**, (or wiring channels—I shall call them channels from now on) have a cer-

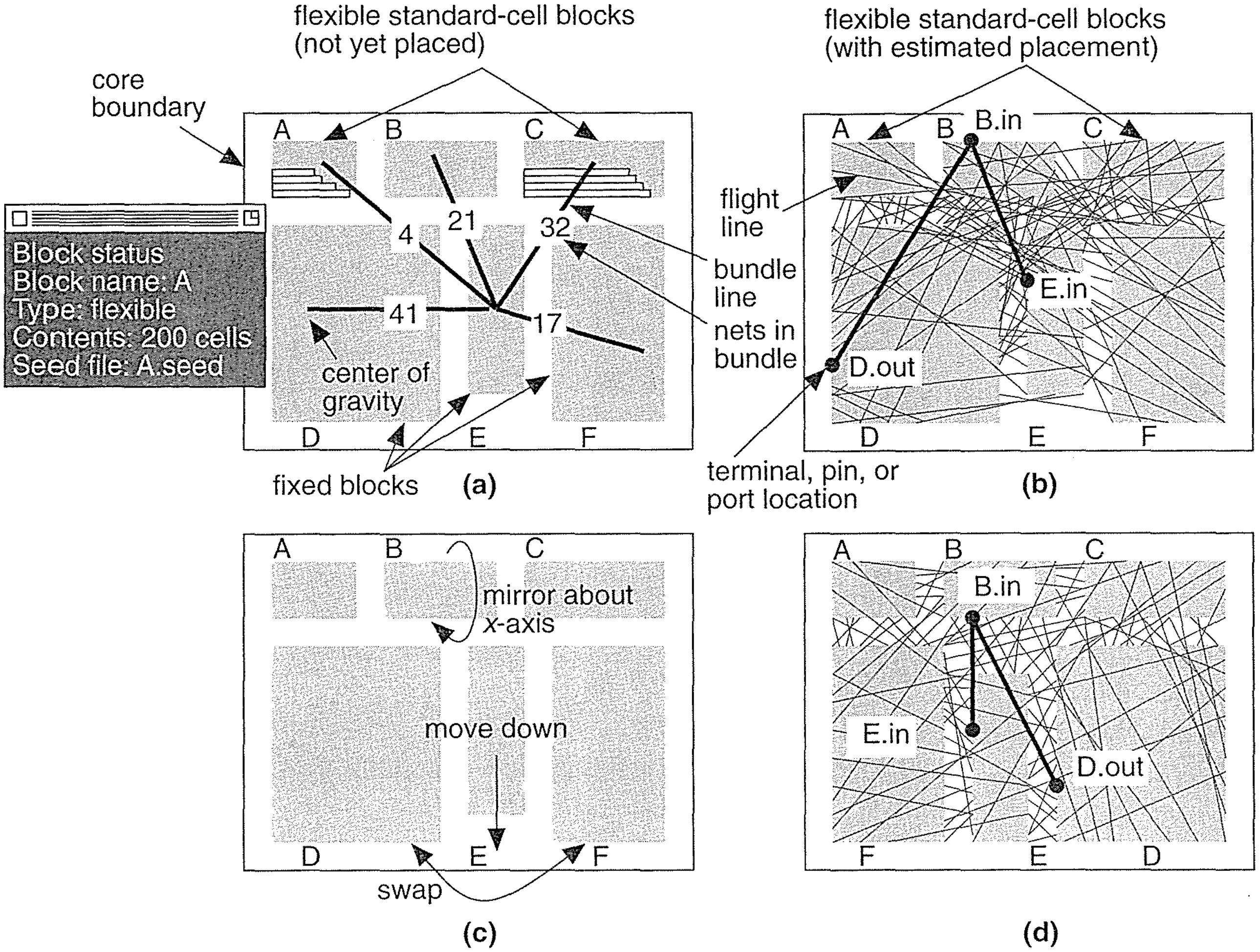


FIGURE 16.6 Floorplanning a cell-based ASIC. (a) Initial floorplan generated by the floorplanning tool. Two of the blocks are flexible (A and C) and contain rows of standard cells (unplaced). A pop-up window shows the status of block A. (b) An estimated placement for flexible blocks A and C. The connector positions are known and a rat's nest display shows the heavy congestion below block B. (c) Moving blocks to improve the floorplan. (d) The updated display shows the reduced congestion after the changes.

tain **channel capacity**; that is, they can handle only a fixed number of interconnects. One measure of congestion is the difference between the number of interconnects that we actually need, called the **channel density**, and the channel capacity. Another measure, shown in Figure 16.7(c), uses the ratio of channel density to the channel capacity. With practice, we can create a good initial placement by floorplanning and a pictorial display. This is one area where the human ability to recognize patterns and spatial relations is currently superior to a computer program's ability.

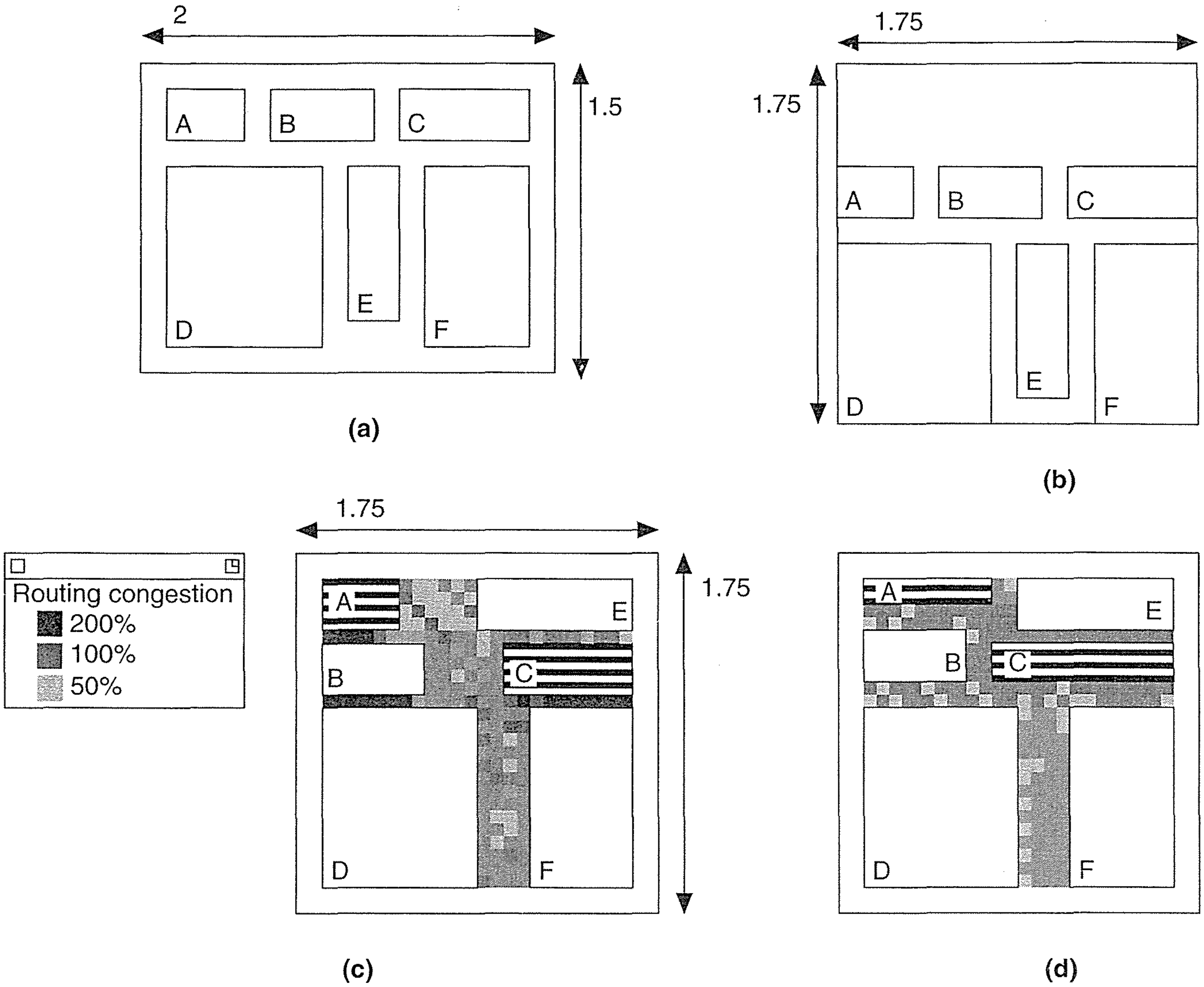


FIGURE 16.7 Congestion analysis. (a) The initial floorplan with a 2:1.5 die aspect ratio. (b) Altering the floorplan to give a 1:1 chip aspect ratio. (c) A trial floorplan with a congestion map. Blocks A and C have been placed so that we know the terminal positions in the channels. Shading indicates the ratio of channel density to the channel capacity. Dark areas show regions that cannot be routed because the channel congestion exceeds the estimated capacity. (d) Resizing flexible blocks A and C alleviates congestion.

16.1.4 Channel Definition

During the floorplanning step we assign the areas between blocks that are to be used for interconnect. This process is known as **channel definition** or **channel allocation**. Figure 16.8 shows a T-shaped junction between two rectangular channels

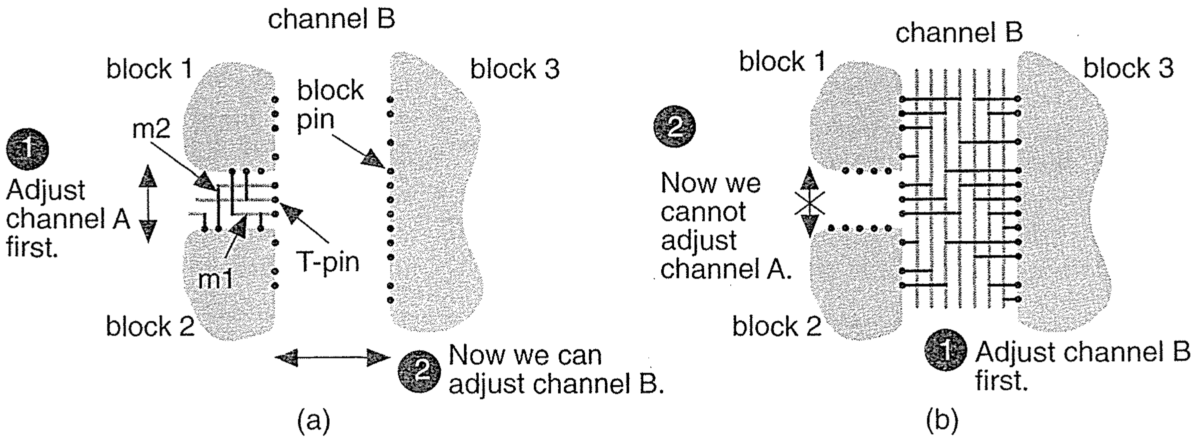


FIGURE 16.8 Routing a T-junction between two channels in two-level metal. The dots represent logic cell pins. (a) Routing channel A (the stem of the T) first allows us to adjust the width of channel B. (b) If we route channel B first (the top of the T), this fixes the width of channel A. We have to route the stem of a T-junction before we route the top.

and illustrates why we must route the stem (vertical) of the T before the bar. The general problem of choosing the order of rectangular channels to route is **channel ordering**.

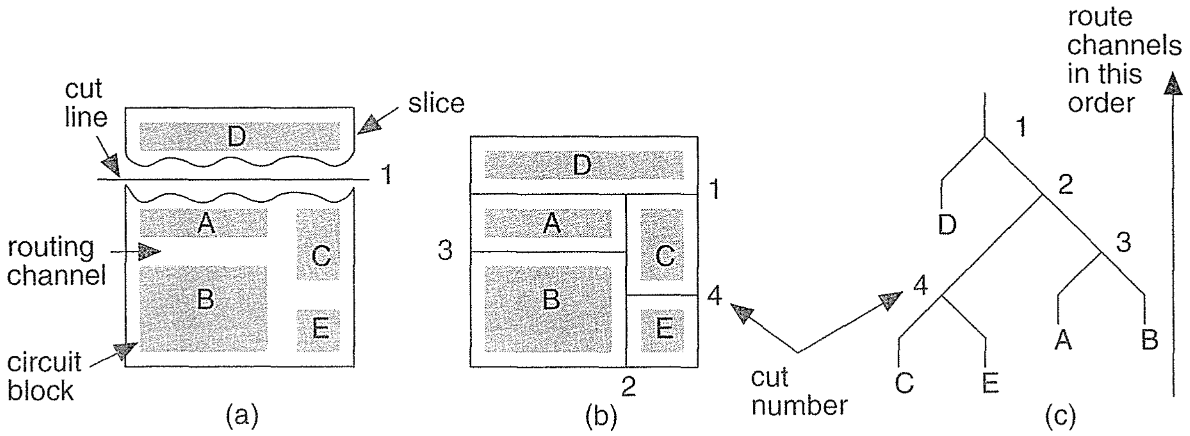


FIGURE 16.9 Defining the channel routing order for a slicing floorplan using a slicing tree. (a) Make a cut all the way across the chip between circuit blocks. Continue slicing until each piece contains just one circuit block. Each cut divides a piece into two without cutting through a circuit block. (b) A sequence of cuts: 1, 2, 3, and 4 that successively slices the chip until only circuit blocks are left. (c) The slicing tree corresponding to the sequence of cuts gives the order in which to route the channels: 4, 3, 2, and finally 1.

Figure 16.9 shows a floorplan of a chip containing several blocks. Suppose we cut along the block boundaries slicing the chip into two pieces (Figure 16.9a). Then suppose we can slice each of these pieces into two. If we can continue in this fashion until all the blocks are separated, then we have a **slicing floorplan** (Figure 16.9b). Figure 16.9(c) shows how the sequence we use to slice the chip defines a hierarchy of the blocks. Reversing the slicing order ensures that we route the stems of all the channel T-junctions first.

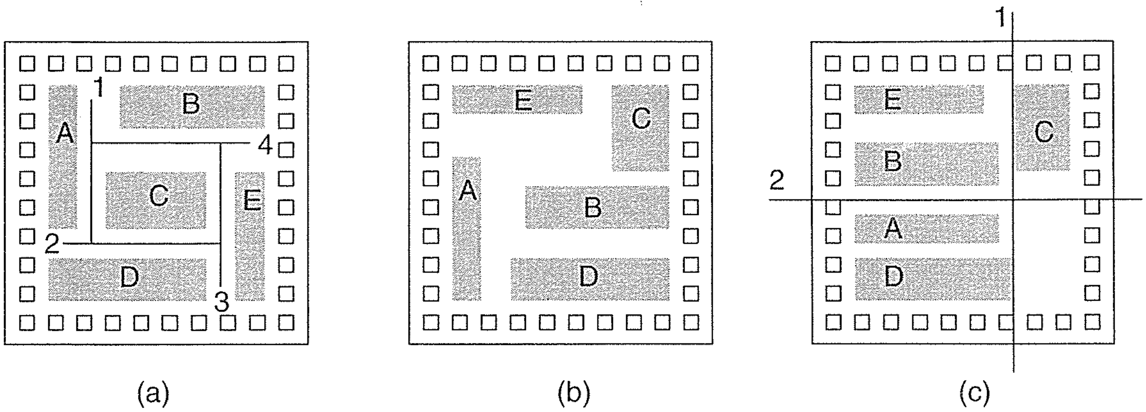


FIGURE 16.10 Cyclic constraints. (a) A nonslicing floorplan with a cyclic constraint that prevents channel routing. (b) In this case it is difficult to find a slicing floorplan without increasing the chip area. (c) This floorplan may be sliced (with initial cuts 1 or 2) and has no cyclic constraints, but it is inefficient in area use and will be very difficult to route.

Figure 16.10 shows a floorplan that is not a slicing structure. We cannot cut the chip all the way across with a knife without chopping a circuit block in two. This means we cannot route any of the channels in this floorplan without routing all of the other channels first. We say there is a **cyclic constraint** in this floorplan. There are two solutions to this problem. One solution is to move the blocks until we obtain a slicing floorplan. The other solution is to allow the use of L-shaped, rather than rectangular, channels (or areas with fixed connectors on all sides—a **switch box**). We need an area-based router rather than a channel router to route L-shaped regions or switch boxes (see Section 17.2.6, "Area-Routing Algorithms").

Figure 16.11(a) displays the floorplan of the ASIC shown in Figure 16.7. We can remove the cyclic constraint by moving the blocks again, but this increases the chip size. Figure 16.11(b) shows an alternative solution. We **merge** the flexible standard cell areas A and C. We can do this by **selective flattening** of the netlist. Sometimes flattening can reduce the routing area because routing between blocks is usually less efficient than routing inside the row-based blocks. Figure 16.11(b) shows the channel definition and **routing order** for our chip.

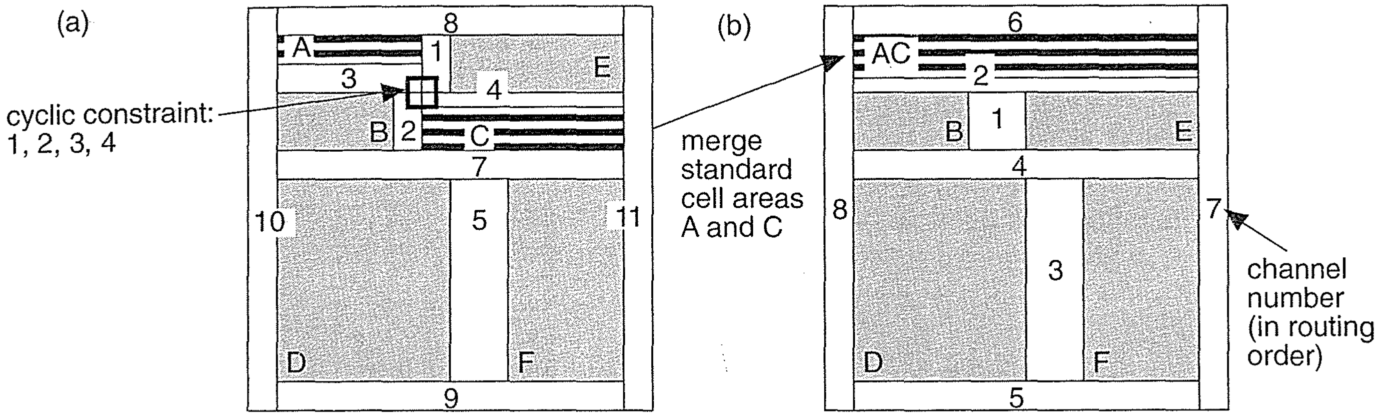


FIGURE 16.11 Channel definition and ordering. (a) We can eliminate the cyclic constraint by merging the blocks A and C. (b) A slicing structure.

16.1.5 I/O and Power Planning

Every chip communicates with the outside world. Signals flow onto and off the chip and we need to supply power. We need to consider the I/O and power constraints early in the floorplanning process. A silicon chip or **die** (plural die, dies, or dice) is mounted on a **chip carrier** inside a chip **package**. Connections are made by **bonding** the chip **pads** to fingers on a metal **lead frame** that is part of the package. The metal lead-frame fingers connect to the **package pins**. A die consists of a logic **core** inside a **pad ring**. Figure 16.12(a) shows a **pad-limited die** and Figure 16.12(b) shows a **core-limited die**. On a pad-limited die we use tall, thin **pad-limited pads**, which maximize the number of pads we can fit around the outside of the chip. On a core-limited die we use short, wide **core-limited pads**. Figure 16.12(c) shows how we can use both types of pad to change the aspect ratio of a die to be different from that of the core.

Special **power pads** are used for the positive supply, or VDD, **power buses** (or **power rails**) and the ground or negative supply, VSS or GND. Usually one set of VDD/VSS pads supplies one **power ring** that runs around the pad ring and supplies power to the I/O pads only. Another set of VDD/VSS pads connects to a second power ring that supplies the logic core. We sometimes call the I/O power **dirty power** since it has to supply large transient currents to the output transistors. We keep dirty power separate to avoid injecting noise into the internal-logic power (the **clean power**). I/O pads also contain special circuits to protect against **electrostatic discharge (ESD)**. These circuits can withstand very short high-voltage (several kilovolt) pulses that can be generated during human or machine handling.

Depending on the type of package and how the foundry attaches the silicon die to the **chip cavity** in the chip carrier, there may be an electrical connection between the chip carrier and the die substrate. Usually the die is cemented in the chip cavity with a conductive epoxy, making an electrical connection between substrate and the

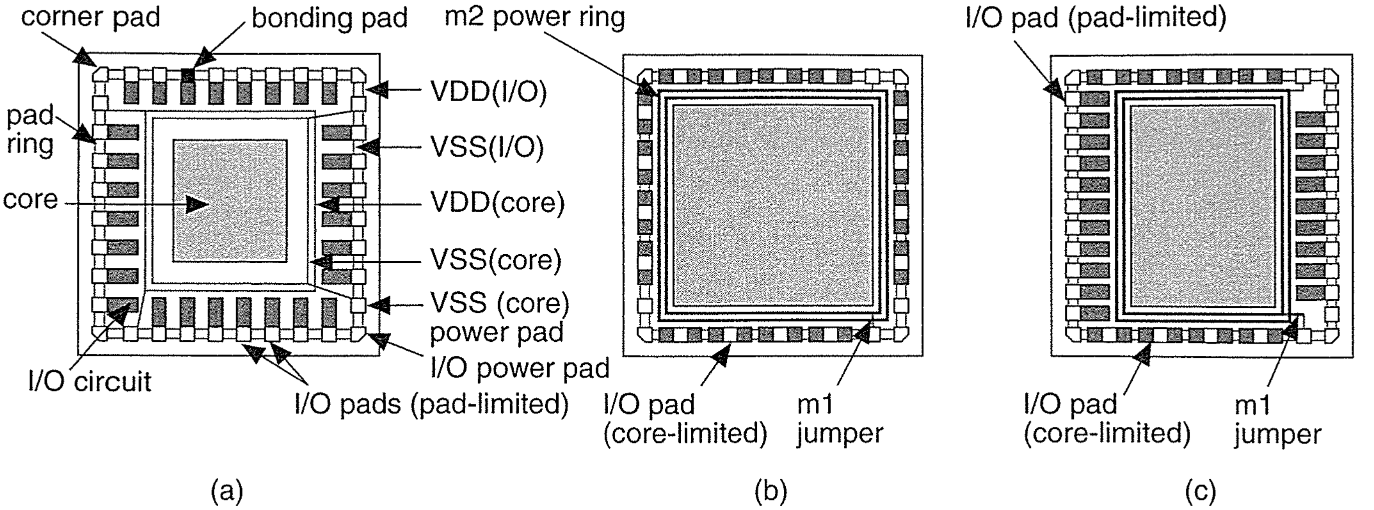


FIGURE 16.12 Pad-limited and core-limited die. (a) A pad-limited die. The number of pads determines the die size. (b) A core-limited die: The core logic determines the die size. (c) Using both pad-limited pads and core-limited pads for a square die.

package cavity in the chip carrier. If we make an electrical connection between the substrate and a chip pad, or to a package pin, it must be to VDD (*n*-type substrate) or VSS (*p*-type substrate). This **substrate connection** (for the whole chip) employs a **down bond** (or drop bond) to the carrier. We have several options:

- We can dedicate one (or more) chip pad(s) to down bond to the chip carrier.
- We can make a connection from a chip pad to the lead frame and down bond from the chip pad to the chip carrier.
- We can make a connection from a chip pad to the lead frame and down bond from the lead frame.
- We can down bond from the lead frame without using a chip pad.
- We can leave the substrate and/or chip carrier unconnected.

Depending on the package design, the type and positioning of down bonds may be fixed. This means we need to fix the position of the chip pad for down bonding using a **pad seed**.

A **double bond** connects two pads to one chip-carrier finger and one package pin. We can do this to save package pins or reduce the series inductance of bond wires (typically a few nanohenries) by parallel connection of the pads. A **multiple-signal pad** or pad group is a set of pads. For example, an **oscillator pad** usually comprises a set of two adjacent pads that we connect to an external crystal. The oscillator circuit and the two signal pads form a single logic cell. Another common example is a **clock pad**. Some foundries allow a special form of **corner pad** (normal pads are **edge pads**) that squeezes two pads into the area at the corners of a chip using a special **two-pad corner cell**, to help meet **bond-wire angle design rules** (see also Figure 16.13b and c).

To reduce the series resistive and inductive impedance of power supply networks, it is normal to use multiple VDD and VSS pads. This is particularly important with the **simultaneously switching outputs** (**SSOs**) that occur when driving buses off-chip [Wada, Eino, and Anami, 1990]. The output pads can easily consume most of the power on a CMOS ASIC, because the load on a pad (usually tens of picofarads) is much larger than typical on-chip capacitive loads. Depending on the technology it may be necessary to provide dedicated VDD and VSS pads for every few SSOs. Design rules set how many SSOs can be used per VDD/VSS pad pair. These dedicated VDD/VSS pads must "follow" groups of output pads as they are seeded or planned on the floorplan. With some chip packages this can become difficult because design rules limit the location of package pins that may be used for supplies (due to the differing series inductance of each pin).

Using a **pad mapping** we translate the **logical pad** in a netlist to a **physical pad** from a **pad library**. We might control pad seeding and mapping in the floorplanner. The handling of I/O pads can become quite complex; there are several nonobvious factors that must be considered when generating a pad ring:

- Ideally we would only need to design library pad cells for one orientation. For example, an edge pad for the south side of the chip, and a corner pad for the southeast corner. We could then generate other orientations by rotation and flipping (mirroring). Some ASIC vendors will not allow rotation or mirroring of logic cells in the mask file. To avoid these problems we may need to have separate horizontal, vertical, left-handed, and right-handed pad cells in the library with appropriate logical to physical pad mappings.
- If we mix pad-limited and core-limited edge pads in the same pad ring, this complicates the design of corner pads. Usually the two types of edge pad cannot abut. In this case a corner pad also becomes a **pad-format changer**, or **hybrid corner pad**.
- In single-supply chips we have one VDD net and one VSS net, both **global power nets**. It is also possible to use **mixed power supplies** (for example, 3.3 V and 5 V) or **multiple power supplies** (digital VDD, analog VDD).

Figure 16.13(a) and (b) are magnified views of the southeast corner of our example chip and show the different types of I/O cells. Figure 16.13(c) shows a **stagger-bond** arrangement using two rows of I/O pads. In this case the design rules for bond wires (the spacing and the angle at which the bond wires leave the pads) become very important.

Figure 16.13(d) shows an **area-bump** bonding arrangement (also known as flip-chip, solder-bump or C4, terms coined by IBM who developed this technology [Masleid, 1991]) used, for example, with **ball-grid array** (**BGA**) packages. Even

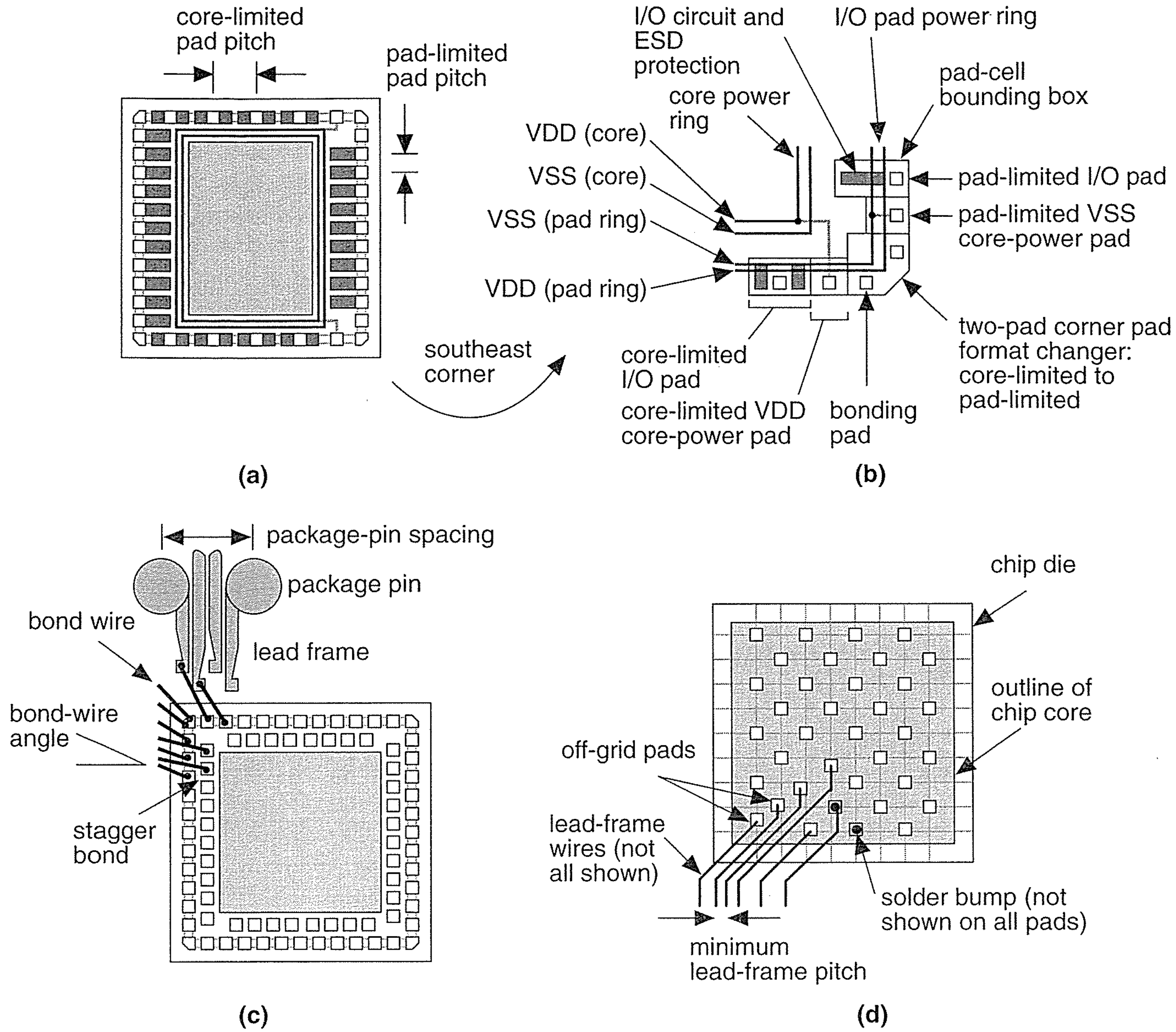


FIGURE 16.13 Bonding pads. (a) This chip uses both pad-limited and core-limited pads. (b) A hybrid corner pad. (c) A chip with stagger-bonded pads. (d) An area-bump bonded chip (or flip-chip). The chip is turned upside down and solder bumps connect the pads to the lead frame.

though the bonding pads are located in the center of the chip, the I/O circuits are still often located at the edges of the chip because of difficulties in power supply distribution and integrating I/O circuits together with logic in the center of the die.

In an MGA the pad spacing and I/O-cell spacing is fixed—each pad occupies a fixed **pad slot** (or **pad site**). This means that the properties of the pad I/O are also fixed but, if we need to, we can parallel adjacent output cells to increase the drive. To increase flexibility further the I/O cells can use a separation, the **I/O-cell pitch**, that is smaller than the **pad pitch**. For example, three 4 mA driver cells can occupy two pad slots. Then we can use two 4 mA output cells in parallel to drive one pad, forming an 8 mA output pad as shown in Figure 16.14. This arrangement also means the I/O pad cells can be changed without changing the base array. This is useful as bonding techniques improve and the pads can be moved closer together.

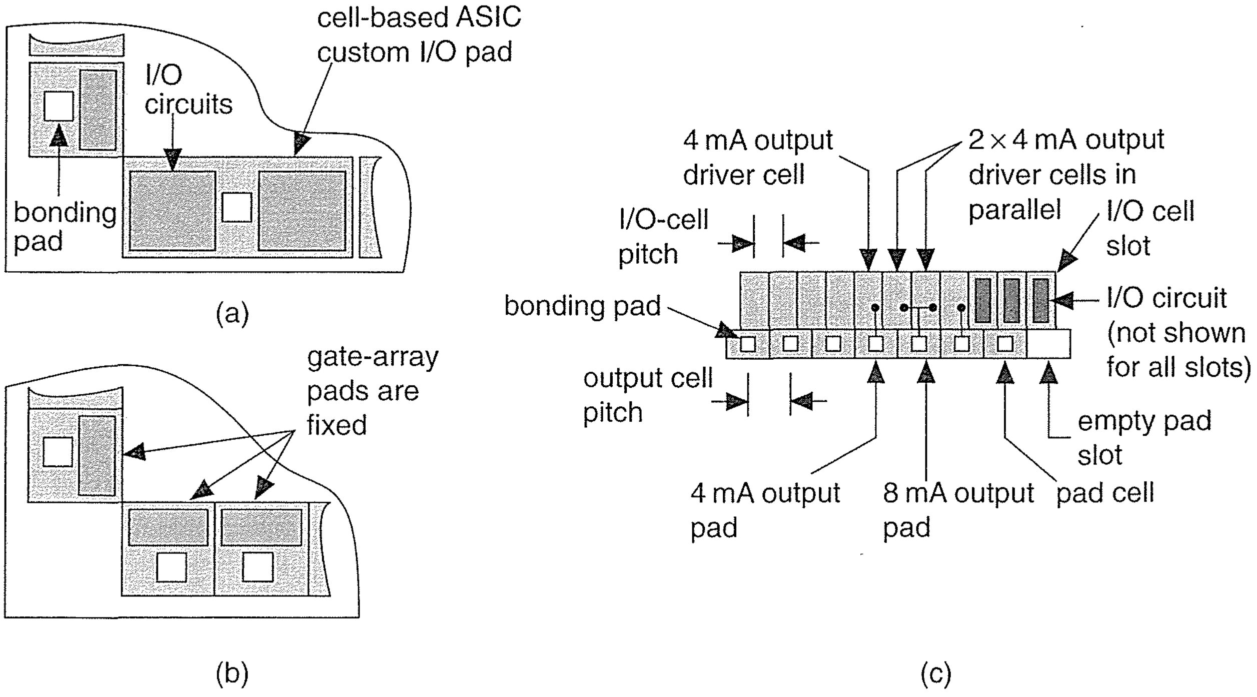


FIGURE 16.14 Gate-array I/O pads. (a) Cell-based ASICs may contain pad cells of different sizes and widths. (b) A corner of a gate-array base. (c) A gate-array base with different I/O cell and pad pitches.

Figure 16.15 shows two possible power distribution schemes. The long direction of a rectangular channel is the **channel spine**. Some automatic routers may require that metal lines parallel to a channel spine use a **preferred layer** (either m1, m2, or m3). Alternatively we say that a particular metal layer runs in a **preferred direction**. Since we can have both horizontal and vertical channels, we may have the situation shown in Figure 16.15, where we have to decide whether to use a preferred layer or the preferred direction for some channels. This may or may not be handled automatically by the routing software.

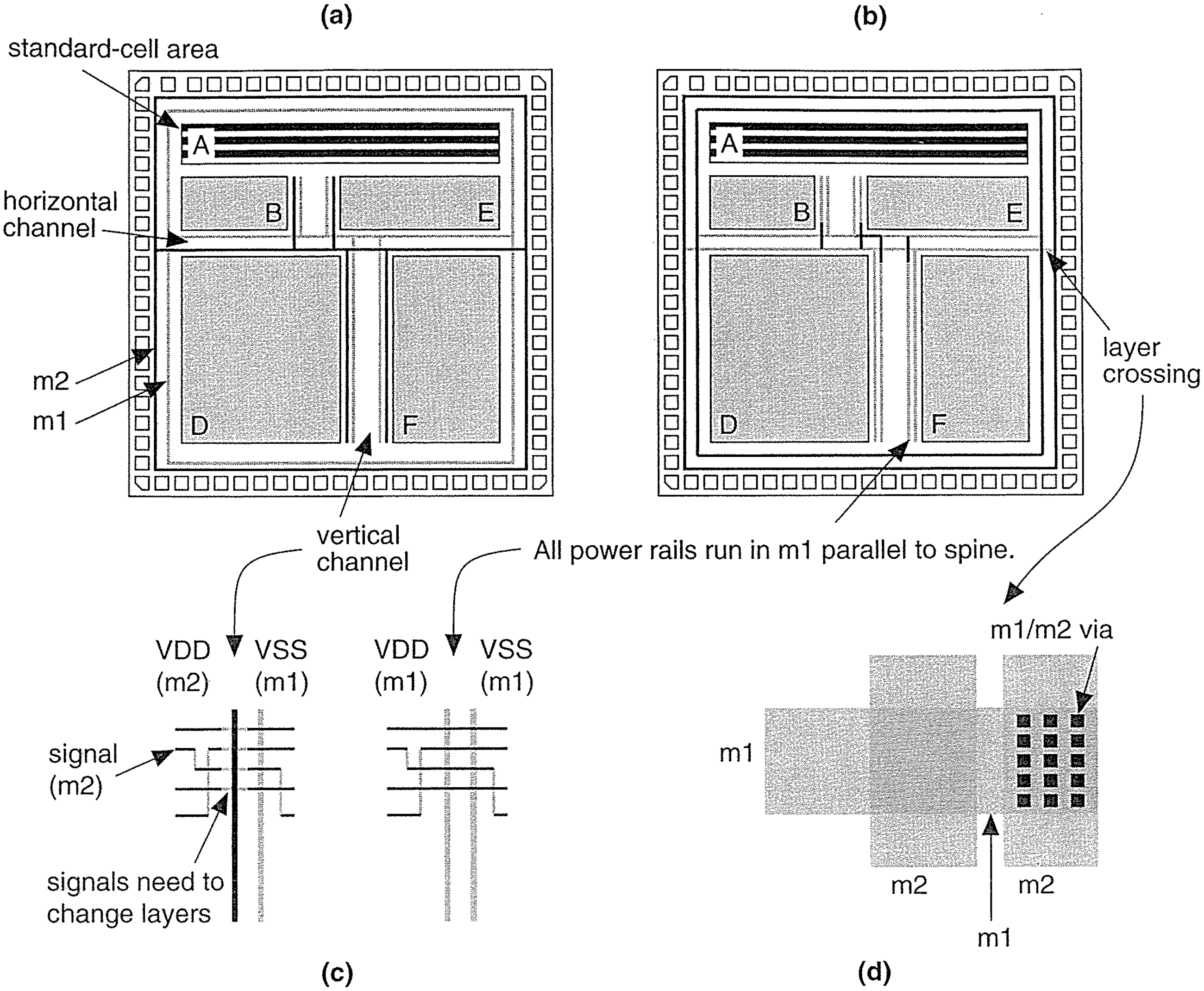


FIGURE 16.15 Power distribution. (a) Power distributed using m1 for VSS and m2 for VDD. This helps minimize the number of vias and layer crossings needed but causes problems in the routing channels. (b) In this floorplan m1 is run parallel to the longest side of all channels, the channel spine. This can make automatic routing easier but may increase the number of vias and layer crossings. (c) An expanded view of part of a channel (interconnect is shown as lines). If power runs on different layers along the spine of a channel, this forces signals to change layers. (d) A closeup of VDD and VSS buses as they cross. Changing layers requires a large number of via contacts to reduce resistance.

16.1.6 Clock Planning

Figure 16.16(a) shows a **clock spine** (not to be confused with a channel spine) routing scheme with all clock pins driven directly from the clock driver. MGAs and FPGAs often use this fish bone type of clock distribution scheme. Figure 16.16(b)

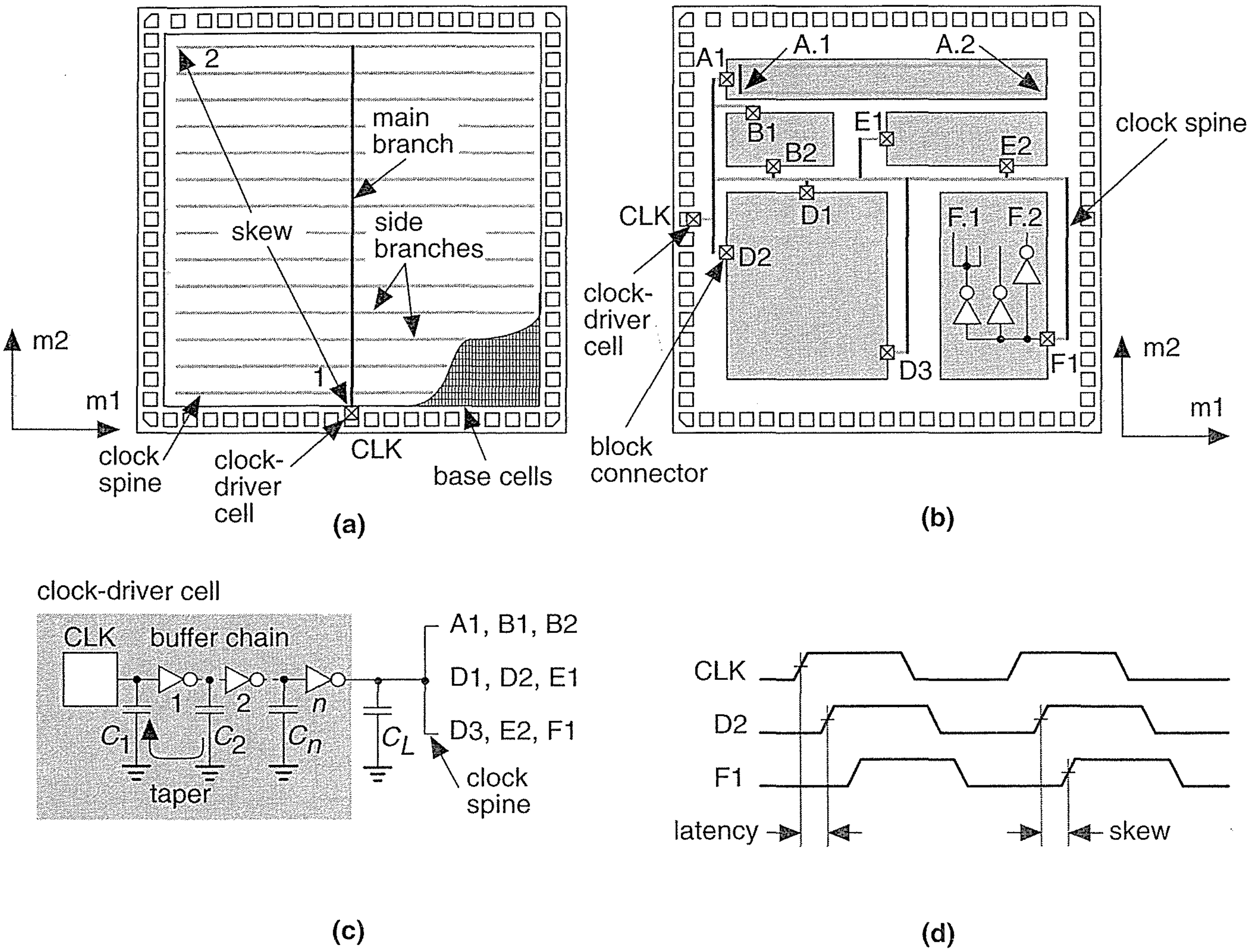


FIGURE 16.16 Clock distribution. (a) A clock spine for a gate array. (b) A clock spine for a cell-based ASIC (typical chips have thousands of clock nets). (c) A clock spine is usually driven from one or more clock-driver cells. Delay in the driver cell is a function of the number of stages and the ratio of output to input capacitance for each stage (taper). (d) Clock latency and clock skew. We would like to minimize both latency and skew.

shows a clock spine for a cell-based ASIC. Figure 16.16(c) shows the clock-driver cell, often part of a special clock-pad cell. Figure 16.16(d) illustrates **clock skew** and **clock latency**. Since all clocked elements are driven from one net with a clock spine, skew is caused by differing interconnect lengths and loads. If the clock-driver delay is much larger than the interconnect delays, a clock spine achieves minimum skew but with long latency.

Clock skew represents a fraction of the clock period that we cannot use for computation. A clock skew of 500 ps with a 200 MHz clock means that we waste 500 ps of every 5 ns clock cycle, or 10 percent of performance. Latency can cause a similar loss of performance at the system level when we need to resynchronize our output signals with a master system clock.

Figure 16.16(c) illustrates the construction of a clock-driver cell. The delay through a chain of CMOS gates is minimized when the ratio between the input capacitance C_1 and the output (load) capacitance C_2 is about 3 (exactly $e \approx 2.7$, an exponential ratio, if we neglect the effect of parasitics). This means that the fastest way to drive a large load is to use a chain of buffers with their input and output loads chosen to maintain this ratio, or **taper** (we use this as a noun and a verb). This is not necessarily the smallest or lowest-power method, though.

Suppose we have an ASIC with the following specifications:

- 40,000 flip-flops
- Input capacitance of the clock input to each flip-flop is 0.025 pF
- Clock frequency is 200 MHz
- $V_{DD} = 3.3$ V
- Chip size is 20 mm on a side
- Clock spine consists of 200 lines across the chip
- Interconnect capacitance is 2 pFcm^{-1}

In this case the clock-spine capacitance $C_L = 200 \times 2\,\text{cm} \times 2\,\text{pFcm}^{-1} = 800\,\text{pF}$. If we drive the clock spine with a chain of buffers with taper equal to $e \approx 2.7$, and with a first-stage input capacitance of 0.025 pF (a reasonable value for a 0.5 µm process), we will need

$$\log \frac{800 \times 10^{-12}}{0.025 \times 10^{-12}} = 10.4\,, \quad \text{or 11 stages.} \tag{16.1}$$

The power dissipated charging the input capacitance of the flip-flop clock is $fC V^2$ or

$$P_1^1 = (4 \times 10^4)\ (200\ \text{MHz})\ (0.025\ \text{pF})\ (3.3\ \text{V})^2 = 2.178\ \text{W}\,, \tag{16.2}$$

or approximately 2 W. This is only a little larger than the power dissipated driving the 800 pF clock-spine interconnect that we can calculate as follows:

$$P_1^2 = (200)\ (200\ \text{MHz})\ (20\ \text{mm})\ (2\ \text{pFcm}^{-1})\ (3.3\ \text{V})^2 = 1.7424\ \text{W}\,. \tag{16.3}$$

All of this power is dissipated in the clock-driver cell. The worst problem, however, is the enormous peak current in the final inverter stage. If we assume the needed rise time is 0.1 ns (with a 200 MHz clock whose period is 5 ns), the peak current would have to approach

$$I = \frac{(800\ \text{pF})\ (\ 3.3\text{V})}{0.1\ \text{ns}} = 25\ \text{A}\,. \tag{16.4}$$

Clearly such a current is not possible without extraordinary design techniques. Clock spines are used to drive loads of 100–200 pF but, as is apparent from the power dissipation problems of this example, it would be better to find a way to spread the power dissipation more evenly across the chip.

We can design a tree of clock buffers so that the taper of each stage is e ≈ 2.7 by using a fanout of three at each node, as shown in Figure 16.17(a) and (b). The **clock tree**, shown in Figure 16.17(c), uses the same number of stages as a clock spine, but with a lower peak current for the inverter buffers. Figure 16.17(c) illustrates that we now have another problem—we need to balance the delays through the tree carefully to minimize clock skew (see Section 17.3.1, "Clock Routing").

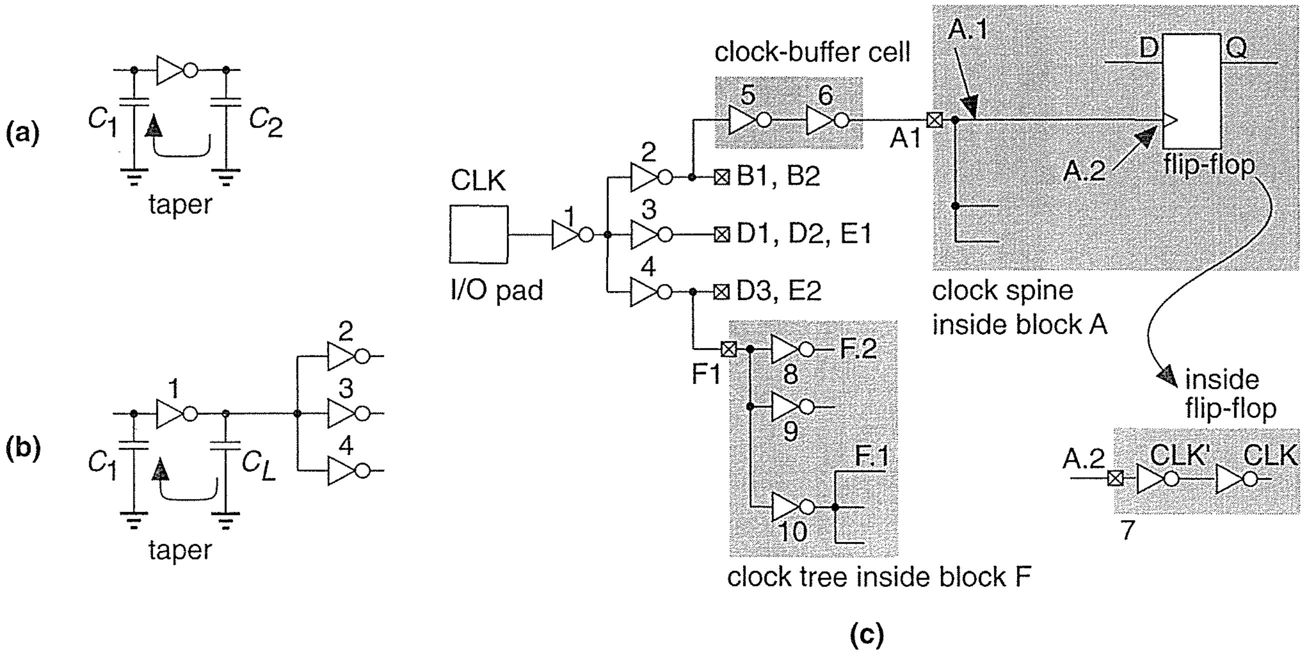


FIGURE 16.17 A clock tree. (a) Minimum delay is achieved when the taper of successive stages is about 3. (b) Using a fanout of three at successive nodes. (c) A clock tree for the cell-based ASIC of Figure 16.16b. We have to balance the clock arrival times at all of the leaf nodes to minimize clock skew.

Designing a clock tree that balances the rise and fall times at the leaf nodes has the beneficial side-effect of minimizing the effect of **hot-electron wearout**. This problem occurs when an electron gains enough energy to become "hot" and jump out of the channel into the gate oxide (the problem is worse for electrons in *n*-channel devices because electrons are more mobile than holes). The trapped electrons change the threshold voltage of the device and this alters the delay of the buffers. As the buffer delays change with time, this introduces unpredictable skew. The problem is worst when the *n*-channel device is carrying maximum current with a high voltage across the channel—this occurs during the rise-and fall-time transitions. Balancing the rise and fall times in each buffer means that they all wear out at the same rate, minimizing any additional skew.

A **phase-locked loop** (PLL) is an electronic flywheel that locks in frequency to an input clock signal. The input and output frequencies may differ in phase, however. This means that we can, for example, drive a clock network with a PLL in such a way that the output of the clock network is locked in phase to the incoming clock, thus eliminating the latency of the clock network. A PLL can also help to reduce random variation of the input clock frequency, known as **jitter**, which, since it is unpredictable, must also be discounted from the time available for computation in each clock cycle. Actel was one of the first FPGA vendors to incorporate PLLs, and Actel's online product literature explains their use in ASIC design.

16.2 Placement

After completing a floorplan we can begin placement of the logic cells within the flexible blocks. Placement is much more suited to automation than floorplanning. Thus we shall need measurement techniques and algorithms. After we complete floorplanning and placement, we can predict both intrablock and interblock capacitances. This allows us to return to logic synthesis with more accurate estimates of the capacitive loads that each logic cell must drive.

16.2.1 Placement Terms and Definitions

CBIC, MGA, and FPGA architectures all have rows of logic cells separated by the interconnect—these are **row-based ASICs**. Figure 16.18 shows an example of the interconnect structure for a CBIC. Interconnect runs in horizontal and vertical directions in the channels and in the vertical direction by crossing through the logic cells. Figure 16.18(c) illustrates the fact that it is possible to use **over-the-cell routing** (**OTC** routing) in areas that are not blocked. However, OTC routing is complicated by the fact that the logic cells themselves may contain metal on the routing layers. We shall return to this topic in Section 17.2.7, "Multilevel Routing." Figure 16.19 shows the interconnect structure of a two-level metal MGA.

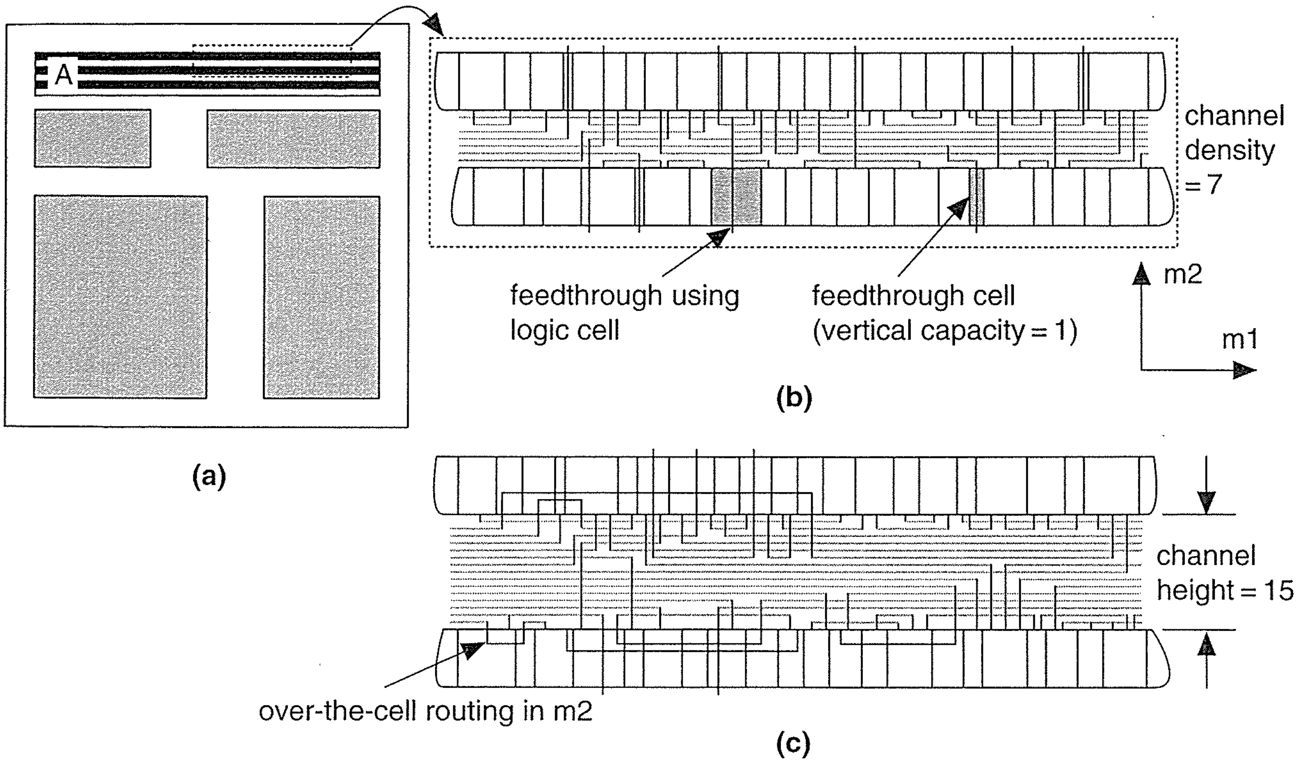


FIGURE 16.18 Interconnect structure. (a) The two-level metal CBIC floorplan shown in Figure 16.11b. (b) A channel from the flexible block A. This channel has a channel height equal to the maximum channel density of 7 (there is room for seven interconnects to run horizontally in m1). (c) A channel that uses OTC (over-the-cell) routing in m2.

Most ASICs currently use two or three levels of metal for signal routing. With two layers of metal, we route within the rectangular channels using the first metal layer for horizontal routing, parallel to the channel spine, and the second metal layer for the vertical direction (if there is a third metal layer it will normally run in the horizontal direction again). The maximum number of horizontal interconnects that can be placed side by side, parallel to the channel spine, is the **channel capacity**.

Vertical interconnect uses **feedthroughs** (or **feedthrus** in the United States) to cross the logic cells. Here are some commonly used terms with explanations (there are no generally accepted definitions):

- An unused **vertical track** (or just **track**) in a logic cell is called an **uncommitted feedthrough** (also **built-in feedthrough**, **implicit feedthrough**, or **jumper**).
- A vertical strip of metal that runs from the top to bottom of a cell (for **double-entry cells**), but has no connections inside the cell, is also called a feedthrough or jumper.

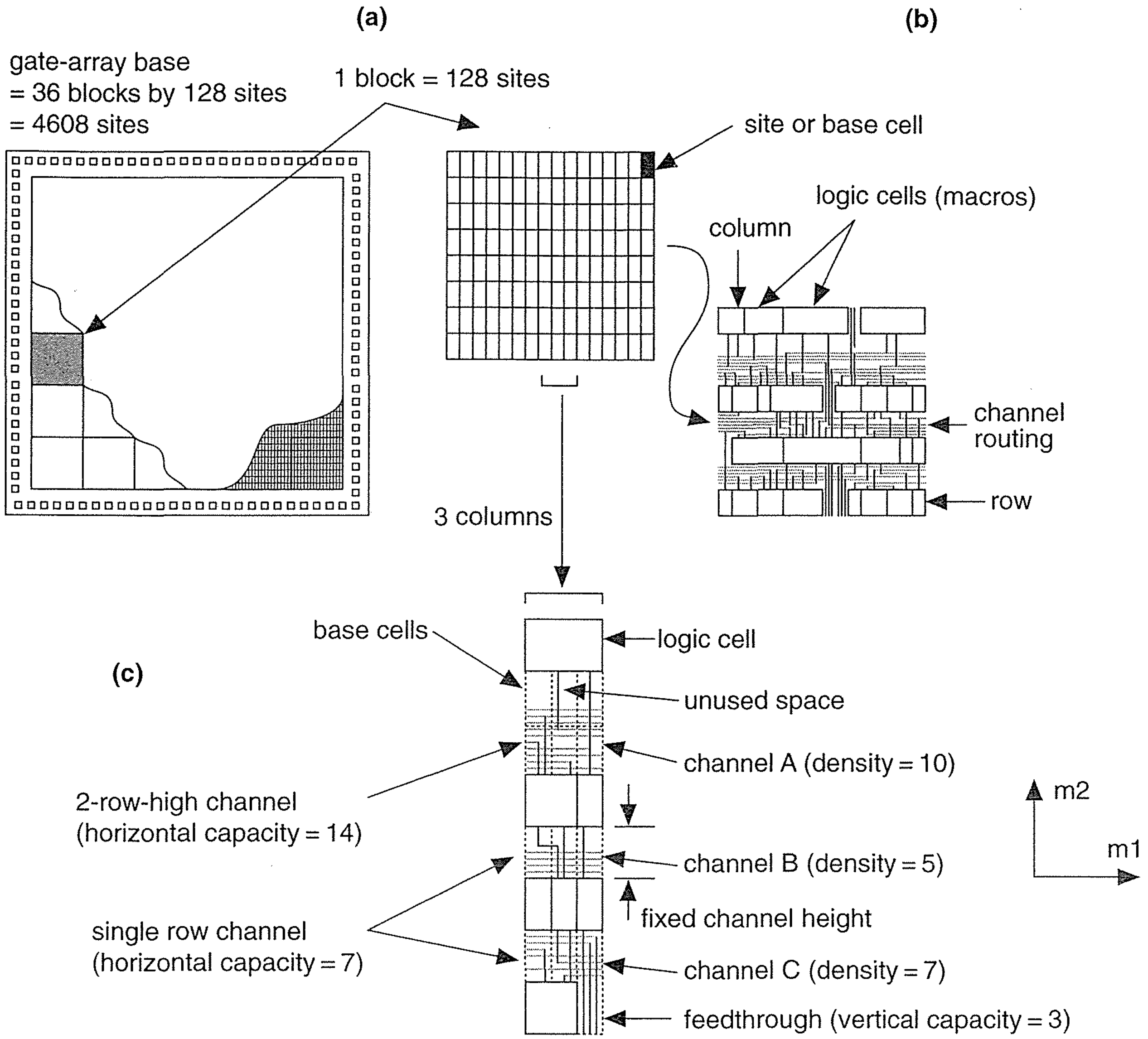


FIGURE 16.19 Gate-array interconnect. (a) A small two-level metal gate array (about 4.6 k-gate). (b) Routing in a block. (c) Channel routing showing channel density and channel capacity. The channel height on a gate array may only be increased in increments of a row. If the interconnect does not use up all of the channel, the rest of the space is wasted. The interconnect in the channel runs in m1 in the horizontal direction with m2 in the vertical direction.

- Two connectors for the same physical net are **electrically equivalent connectors** (or **equipotential connectors**). For double-entry cells these are usually at the top and bottom of the logic cell.
- A dedicated **feedthrough cell** (or **crosser cell**) is an empty cell (with no logic) that can hold one or more vertical interconnects. These are used if there are no other feedthroughs available.

- A **feedthrough pin** or **feedthrough terminal** is an input or output that has connections at both the top and bottom of the standard cell.
- A **spacer cell** (usually the same as a feedthrough cell) is used to fill space in rows so that the ends of all rows in a flexible block may be aligned to connect to power buses, for example.

There is no standard terminology for connectors and the terms can be very confusing. There is a difference between connectors that are joined inside the logic cell using a high-resistance material such as polysilicon and connectors that are joined by low-resistance metal. The high-resistance kind are really two separate **alternative connectors** (that cannot be used as a feedthrough), whereas the low-resistance kind are electrically equivalent connectors. There may be two or more connectors to a logic cell, which are not joined inside the cell, and which must be joined by the router (**must-join connectors**).

There are also **logically equivalent connectors** (or functionally equivalent connectors, sometimes also called just equivalent connectors—which is very confusing). The two inputs of a two-input NAND gate may be logically equivalent connectors. The placement tool can swap these without altering the logic (but the two inputs may have different delay properties, so it is not always a good idea to swap them). There can also be **logically equivalent connector groups**. For example, in an OAI22 (OR-AND-INVERT) gate there are four inputs: A1, A2 are inputs to one OR gate (gate A), and B1, B2 are inputs to the second OR gate (gate B). Then group A = (A1, A2) is logically equivalent to group B = (B1, B2)—if we swap one input (A1 or A2) from gate A to gate B, we must swap the other input in the group (A2 or A1).

In the case of channeled gate arrays and FPGAs, the horizontal interconnect areas—the channels, usually on m1—have a fixed capacity (sometimes they are called **fixed-resource ASICs** for this reason). The channel capacity of CBICs and channelless MGAs can be expanded to hold as many interconnects as are needed. Normally we choose, as an objective, to minimize the number of interconnects that use each channel. In the vertical interconnect direction, usually m2, FPGAs still have fixed resources. In contrast the placement tool can always add vertical feedthroughs to a channeled MGA, channelless MGA, or CBIC. These problems become less important as we move to three and more levels of interconnect.

16.2.2 Placement Goals and Objectives

The goal of a placement tool is to arrange all the logic cells within the flexible blocks on a chip. Ideally, the objectives of the placement step are to

- Guarantee the router can complete the routing step
- Minimize all the critical net delays
- Make the chip as dense as possible

We may also have the following additional objectives:

- Minimize power dissipation
- Minimize cross talk between signals

Objectives such as these are difficult to define in a way that can be solved with an algorithm and even harder to actually meet. Current placement tools use more specific and achievable criteria. The most commonly used placement objectives are one or more of the following:

- Minimize the total estimated interconnect length
- Meet the timing requirements for critical nets
- Minimize the interconnect congestion

Each of these objectives in some way represents a compromise.

16.2.3 Measurement of Placement Goals and Objectives

In order to determine the quality of a placement, we need to be able to measure it. We need an approximate measure of interconnect length, closely correlated with the final interconnect length, that is easy to calculate.

The graph structures that correspond to making all the connections for a net are known as **trees on graphs** (or just **trees**). Special classes of trees—**Steiner trees**—minimize the total length of interconnect and they are central to ASIC routing algorithms. Figure 16.20 shows a minimum Steiner tree. This type of tree uses diagonal connections—we want to solve a restricted version of this problem, using interconnects on a rectangular grid. This is called **rectilinear routing** or **Manhattan routing** (because of the east–west and north–south grid of streets in Manhattan). We say that the **Euclidean distance** between two points is the straight-line distance ("as the crow flies"). The **Manhattan distance** (or rectangular distance) between two points is the distance we would have to walk in New York.

The **minimum rectilinear Steiner tree** (MRST) is the shortest interconnect using a rectangular grid. The determination of the MRST is in general an NP-complete problem—which means it is hard to solve. For small numbers of terminals heuristic algorithms do exist, but they are expensive to compute. Fortunately we only need to estimate the length of the interconnect. Two approximations to the MRST are shown in Figure 16.21.

The **complete graph** has connections from each terminal to every other terminal [Hanan, Wolff, and Agule, 1973]. The **complete-graph measure** adds all the interconnect lengths of the complete-graph connection together and then divides by $n/2$, where n is the number of terminals. We can justify this since, in a graph with n terminals, $(n-1)$ interconnects will emanate from each terminal to join the other $(n-1)$ terminals in a complete graph connection. That makes $n(n-1)$ interconnects in total. However, we have then made each connection twice. So there are one-half

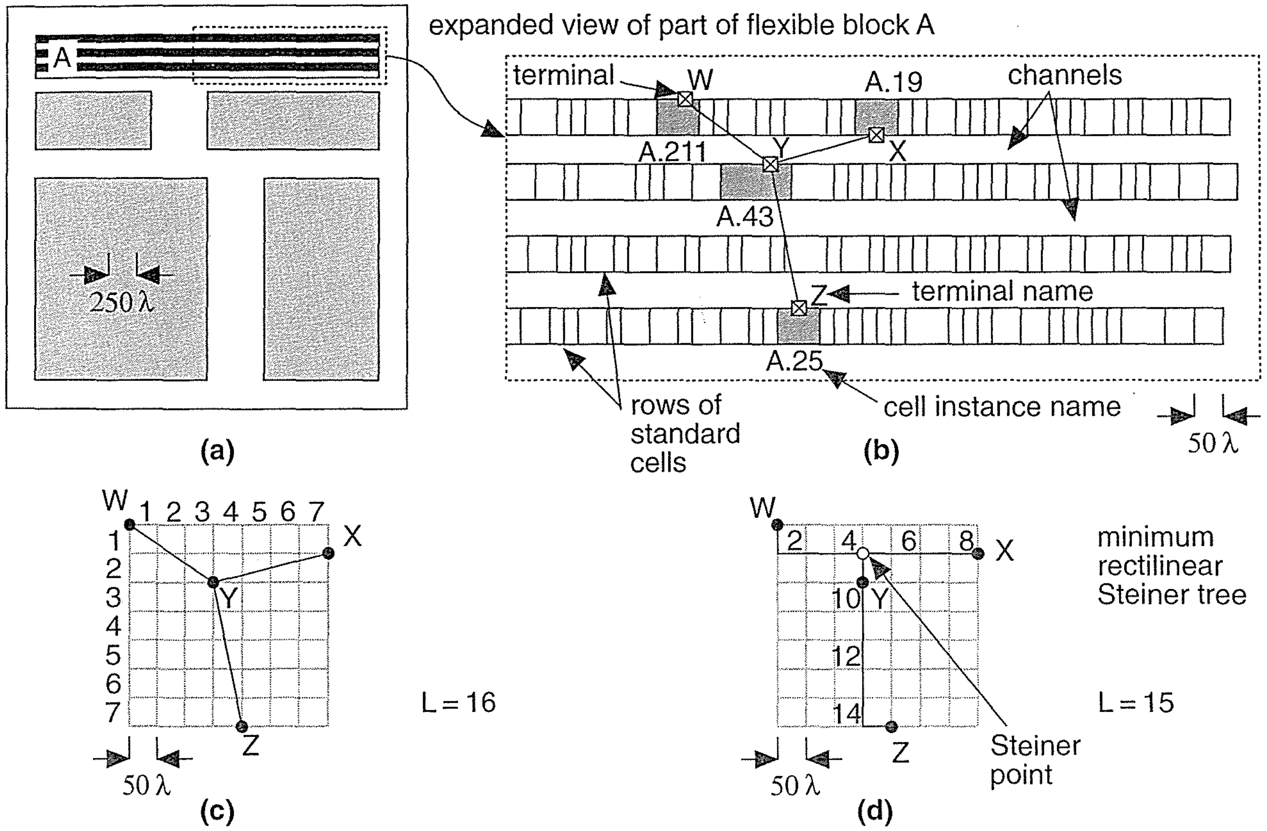


FIGURE 16.20 Placement using trees on graphs. (a) The floorplan from Figure 16.11b. (b) An expanded view of the flexible block A showing four rows of standard cells for placement (typical blocks may contain thousands or tens of thousands of logic cells). We want to find the length of the net shown with four terminals, W through Z, given the placement of four logic cells (labeled: A.211, A.19, A.43, A.25). (c) The problem for net (W, X, Y, Z) drawn as a graph. The shortest connection is the minimum Steiner tree. (d) The minimum rectilinear Steiner tree using Manhattan routing. The rectangular (Manhattan) interconnect-length measures are shown for each tree.

this many, or $n(n-1)/2$, interconnects needed for a complete graph connection. Now we actually only need $(n-1)$ interconnects to join n terminals, so we have $n/2$ times as many interconnects as we really need. Hence we divide the total net length of the complete graph connection by $n/2$ to obtain a more reasonable estimate of minimum interconnect length. Figure 16.21(a) shows an example of the complete-graph measure.

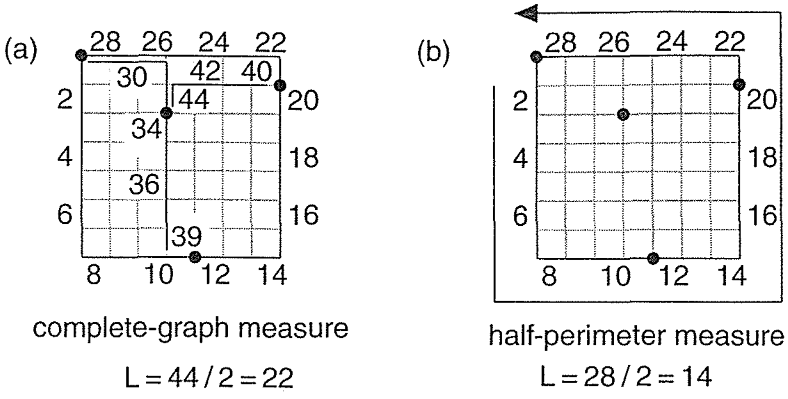


FIGURE 16.21 Interconnect-length measures. (a) Complete-graph measure. (b) Half-perimeter measure.

The **bounding box** is the smallest rectangle that encloses all the terminals (not to be confused with a logic cell bounding box, which encloses all the layout in a logic cell). The **half-perimeter measure** (or bounding-box measure) is one-half the perimeter of the bounding box (Figure 16.21b) [Schweikert, 1976]. For nets with two or three terminals (corresponding to a fanout of one or two, which usually includes over 50 percent of all nets on a chip), the half-perimeter measure is the same as the minimum Steiner tree. For nets with four or five terminals, the minimum Steiner tree is between one and two times the half-perimeter measure [Hanan, 1966]. For a circuit with m nets, using the half-perimeter measure corresponds to minimizing the cost function,

$$f = \frac{1}{2}\sum_{i=1}^{m} h_i \, , \qquad (16.5)$$

where h_i is the half-perimeter measure for net i.

It does not really matter if our approximations are inaccurate if there is a good correlation between actual interconnect lengths (after routing) and our approximations. Figure 16.22 shows that we can adjust the complete-graph and half-perimeter measures using correction factors [Goto and Matsuda, 1986]. Now our wiring length approximations are functions, not just of the terminal positions, but also of the number of terminals, and the size of the bounding box. One practical example adjusts a Steiner-tree approximation using the number of terminals [Chao, Nequist, and Vuong, 1990]. This technique is used in the Cadence Gate Ensemble placement tool, for example.

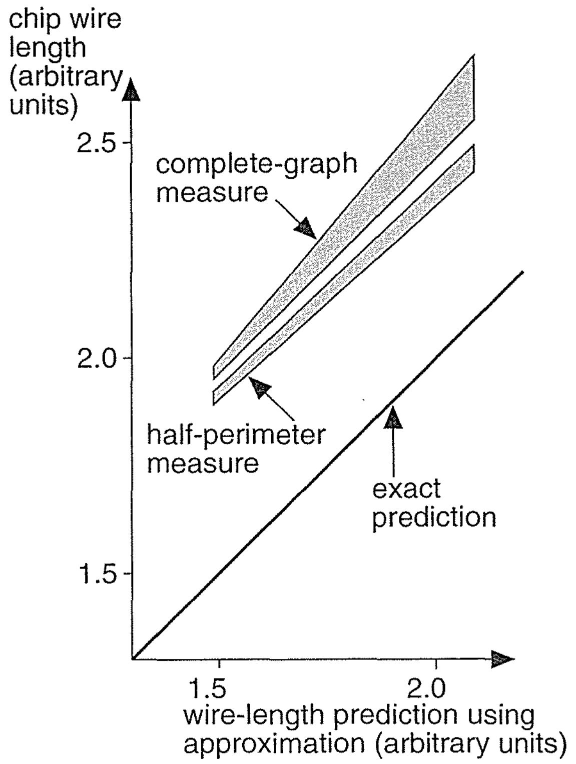


FIGURE 16.22 Correlation between total length of chip interconnect and the half-perimeter and complete-graph measures.

One problem with the measurements we have described is that the MRST may only approximate the interconnect that will be completed by the detailed router. Some programs have a **meander factor** that specifies, on average, the ratio of the interconnect created by the routing tool to the interconnect-length estimate used by the placement tool. Another problem is that we have concentrated on finding estimates to the MRST, but the MRST that minimizes total net length may not minimize net delay (see Section 16.2.8).

There is no point in minimizing the interconnect length if we create a placement that is too congested to route. If we use minimum **interconnect congestion** as an additional placement objective, we need some way of measuring it. What we are trying to measure is interconnect density. Unfortunately we always use the term *density* to mean channel density (which we shall discuss in Section 17.2.2, "Measurement of Channel Density"). In this chapter, while we are discussing placement, we shall try to use the term *congestion*, instead of density, to avoid any confusion.

One measure of interconnect congestion uses the **maximum cut line**. Imagine a horizontal or vertical line drawn anywhere across a chip or block, as shown in Figure 16.23. The number of interconnects that must cross this line is the **cut size** (the number of interconnects we cut). The maximum cut line has the highest cut size.

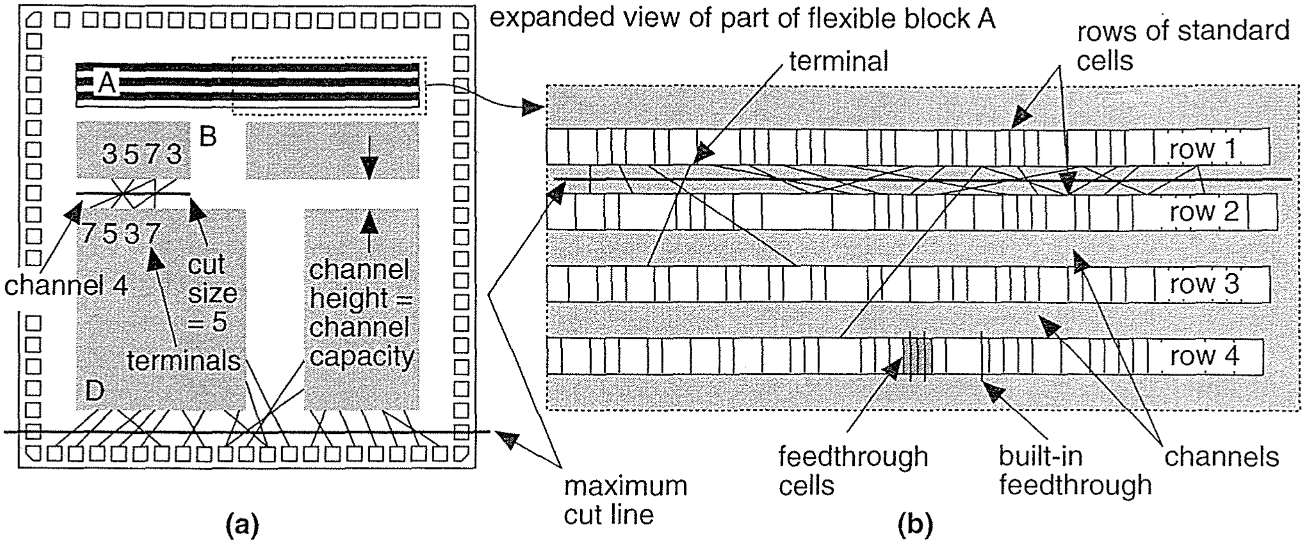


FIGURE 16.23 Interconnect congestion for the cell-based ASIC from Figure 16.11(b). (a) Measurement of congestion. (b) An expanded view of flexible block A shows a maximum cut line.

Many placement tools minimize estimated interconnect length or interconnect congestion as objectives. The problem with this approach is that a logic cell may be placed a long way from another logic cell to which it has just one connection. This logic cell with one connection is less important as far as the total wire length is concerned than other logic cells, to which there are many connections. However, the one long connection may be critical as far as timing delay is concerned. As technology is scaled, interconnection delays become larger relative to circuit delays and this problem gets worse.

In **timing-driven placement** we must estimate delay for every net for every trial placement, possibly for hundreds of thousands of gates. We cannot afford to use anything other than the very simplest estimates of net delay. Unfortunately, the minimum-length Steiner tree does not necessarily correspond to the interconnect path that minimizes delay. To construct a minimum-delay path we may have to route with non-Steiner trees. In the placement phase typically we take a simple interconnect-length approximation to this minimum-delay path (typically the half-perimeter measure). Even when we can estimate the length of the interconnect, we do not yet have information on which layers and how many vias the interconnect will use or how wide it will be. Some tools allow us to include estimates for these parameters. Often we can specify **metal usage**, the percentage of routing on the different layers to expect from the router. This allows the placement tool to estimate RC values and delays—and thus minimize delay.

16.2.4 Placement Algorithms

There are two classes of placement algorithms commonly used in commercial CAD tools: constructive placement and iterative placement improvement. A **constructive placement method** uses a set of rules to arrive at a constructed placement. The most commonly used methods are variations on the **min-cut algorithm.** The other commonly used constructive placement algorithm is the **eigenvalue method.** As in system partitioning, placement usually starts with a constructed solution and then improves it using an iterative algorithm. In most tools we can specify the locations and relative placements of certain critical logic cells as **seed placements.**

The **min-cut placement** method uses successive application of partitioning [Breuer, 1977]. The following steps are shown in Figure 16.24:

1. Cut the placement area into two pieces.
2. Swap the logic cells to minimize the cut cost.
3. Repeat the process from step 1, cutting smaller pieces until all the logic cells are placed.

Usually we divide the placement area into **bins.** The size of a bin can vary, from a bin size equal to the base cell (for a gate array) to a bin size that would hold several logic cells. We can start with a large bin size, to get a rough placement, and then reduce the bin size to get a final placement.

The **eigenvalue placement algorithm** uses the cost matrix or weighted **connectivity matrix** (eigenvalue methods are also known as **spectral methods**) [Hall, 1970]. The measure we use is a cost function *f* that we shall minimize, given by

$$f = \frac{1}{2} \sum_{i,j=1}^{n} c_{ij} d_{ij}^{2} , \tag{16.6}$$

where $\mathbf{C} = [c_{ij}]$ is the (possibly weighted) connectivity matrix, and d_{ij} is the Euclidean distance between the centers of logic cell *i* and logic cell *j*. Since we are going to minimize a cost function that is the square of the distance between logic cells, these methods are also known as **quadratic placement** methods. This type of cost function leads to a simple mathematical solution. We can rewrite the cost function *f* in matrix form:

$$f = \frac{1}{2} \sum_{i,j=1}^{n} c_{ij} (x_i - x_j)^2 + (y_i - y_j)^2 = \mathbf{x}^T \mathbf{B} \mathbf{x} + \mathbf{y}^T \mathbf{B} \mathbf{y} , \tag{16.7}$$

In Eq. 16.7, **B** is a symmetric matrix, the **disconnection matrix** (also called the Laplacian).

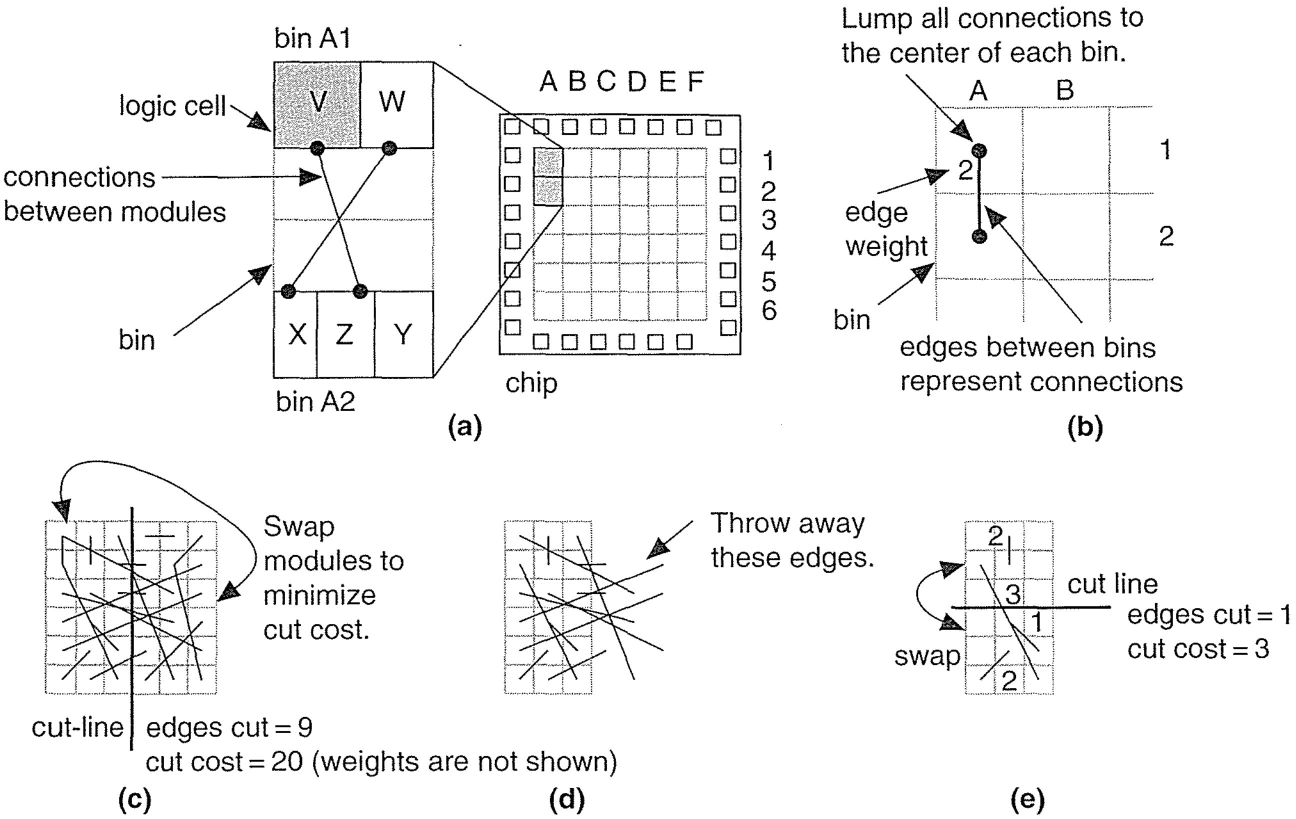

bin A1
logic cell
V
W
connections between modules
bin
X
Z
Y
bin A2
A B C D E F
1
2
3
4
5
6
chip
(a)
Lump all connections to the center of each bin.
A
B
1
2
2
edge weight
bin
edges between bins represent connections
(b)
Swap modules to minimize cut cost.
cut-line
edges cut = 9
cut cost = 20 (weights are not shown)
(c)
Throw away these edges.
(d)
2
3
1
2
cut line
edges cut = 1
cut cost = 3
swap
(e)

FIGURE 16.24 Min-cut placement. (a) Divide the chip into bins using a grid. (b) Merge all connections to the center of each bin. (c) Make a cut and swap logic cells between bins to minimize the cost of the cut. (d) Take the cut pieces and throw out all the edges that are not inside the piece. (e) Repeat the process with a new cut and continue until we reach the individual bins.

We may express the Laplacian **B** in terms of the connectivity matrix **C**; and **D**, a diagonal matrix (known as the degree matrix), defined as follows:

$$\mathbf{B} = \mathbf{D} - \mathbf{C};\ d_{ii} = \sum_{j=1}^{n} c_{ij}\,,\ i = 1, \ldots, n;\ d_{ij} = 0,\quad i \neq j. \qquad (16.8)$$

We can simplify the problem by noticing that it is symmetric in the x- and y-coordinates. Let us solve the simpler problem of minimizing the cost function for the placement of logic cells along just the x-axis first. We can then apply this solution to the more general two-dimensional placement problem. Before we solve this simpler problem, we introduce a constraint that the coordinates of the logic cells must correspond to valid positions (the cells do not overlap and they are placed on-grid). We make another simplifying assumption that all logic cells are the same size and we must place them in fixed positions. We can define a vector **p** consisting of the valid positions:

$$\mathbf{p} = [p_1, \ldots, p_n] \ . \qquad (16.9)$$

For a valid placement the x-coordinates of the logic cells,

$$\mathbf{x} = [x_1, \ldots, x_n] \ , \qquad (16.10)$$

must be a permutation of the fixed positions, **p**. We can show that requiring the logic cells to be in fixed positions in this way leads to a series of n equations restricting the values of the logic cell coordinates [Cheng and Kuh, 1984]. If we impose all of these constraint equations the problem becomes very complex. Instead we choose just one of the equations:

$$\sum_{i=1}^{n} x_i^{\ 2} = \sum_{i=1}^{n} p_i^{\ 2} \ . \qquad (16.11)$$

Simplifying the problem in this way will lead to an approximate solution to the placement problem. We can write this single constraint on the x-coordinates in matrix form:

$$\mathbf{x}^T\mathbf{x} = \mathrm{P}; \qquad \mathrm{P} = \sum_{i=1}^{n} p_i^2 \ , \qquad (16.12)$$

where P is a constant. We can now summarize the formulation of the problem, with the simplifications that we have made, for a one-dimensional solution. We must minimize a cost function, g (analogous to the cost function f that we defined for the two-dimensional problem in Eq. 16.7), where

$$g = \mathbf{x}^T\mathbf{B}\mathbf{x} \qquad (16.13)$$

subject to the constraint:

$$\mathbf{x}^T\mathbf{x} = \mathrm{P} \ . \qquad (16.14)$$

This is a standard problem that we can solve using a Lagrangian multiplier:

$$\Lambda = \mathbf{x}^T\mathbf{B}\mathbf{x} - \lambda[\mathbf{x}^T\mathbf{x} - \mathrm{P}] \ . \qquad (16.15)$$

To find the value of **x** that minimizes g we differentiate Λ partially with respect to **x** and set the result equal to zero. We get the following equation:

$$[\mathbf{B} - \lambda\mathbf{I}]\,\mathbf{x} = \mathbf{0} \ . \qquad (16.16)$$

This last equation is called the **characteristic equation** for the disconnection matrix **B** and occurs frequently in matrix algebra (this λ has nothing to do with scaling). The solutions to this equation are the **eigenvectors** and **eigenvalues** of **B**. Multiplying Eq. 16.16 by $\mathbf{x}^T$ we get:

$$\lambda \mathbf{x}^T \mathbf{x} = \mathbf{x}^T \mathbf{B}\mathbf{x}. \qquad (16.17)$$

However, since we imposed the constraint $\mathbf{x}^T\mathbf{x} = \mathrm{P}$ and $\mathbf{x}^T\mathbf{B}\mathbf{x} = g$, then

$$\lambda = \frac{g}{\mathrm{P}}. \qquad (16.18)$$

The eigenvectors of the disconnection matrix **B** are the solutions to our placement problem. It turns out that (because something called the rank of matrix **B** is $n-1$) there is a degenerate solution with all x-coordinates equal ($\lambda = 0$)—this makes some sense because putting all the logic cells on top of one another certainly minimizes the interconnect. The smallest, nonzero, eigenvalue and the corresponding eigenvector provides the solution that we want. In the two-dimensional placement problem, the x- and y-coordinates are given by the eigenvectors corresponding to the two smallest, nonzero, eigenvalues. (In the next section a simple example illustrates this mathematical derivation.)

16.2.5 Eigenvalue Placement Example

Consider the following connectivity matrix **C** and its disconnection matrix **B**, calculated from Eq. 16.8 [Hall, 1970]:

$$\mathbf{C} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{bmatrix} ; \ \mathbf{B} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 2 \end{bmatrix} - \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & -1 \\ 0 & 2 & -1 & -1 \\ 0 & -1 & 1 & 0 \\ -1 & -1 & 0 & 2 \end{bmatrix} \qquad (16.19)$$

Figure 16.25(a) shows the corresponding network with four logic cells (1–4) and three nets (A-C). Here is a MatLab script to find the eigenvalues and eigenvectors of **B**:

```
C=[0 0 0 1; 0 0 1 1; 0 1 0 0; 1 1 0 0]
D=[1 0 0 0; 0 2 0 0; 0 0 1 0; 0 0 0 2]
B=D-C
[X,D] = eig(B)
```

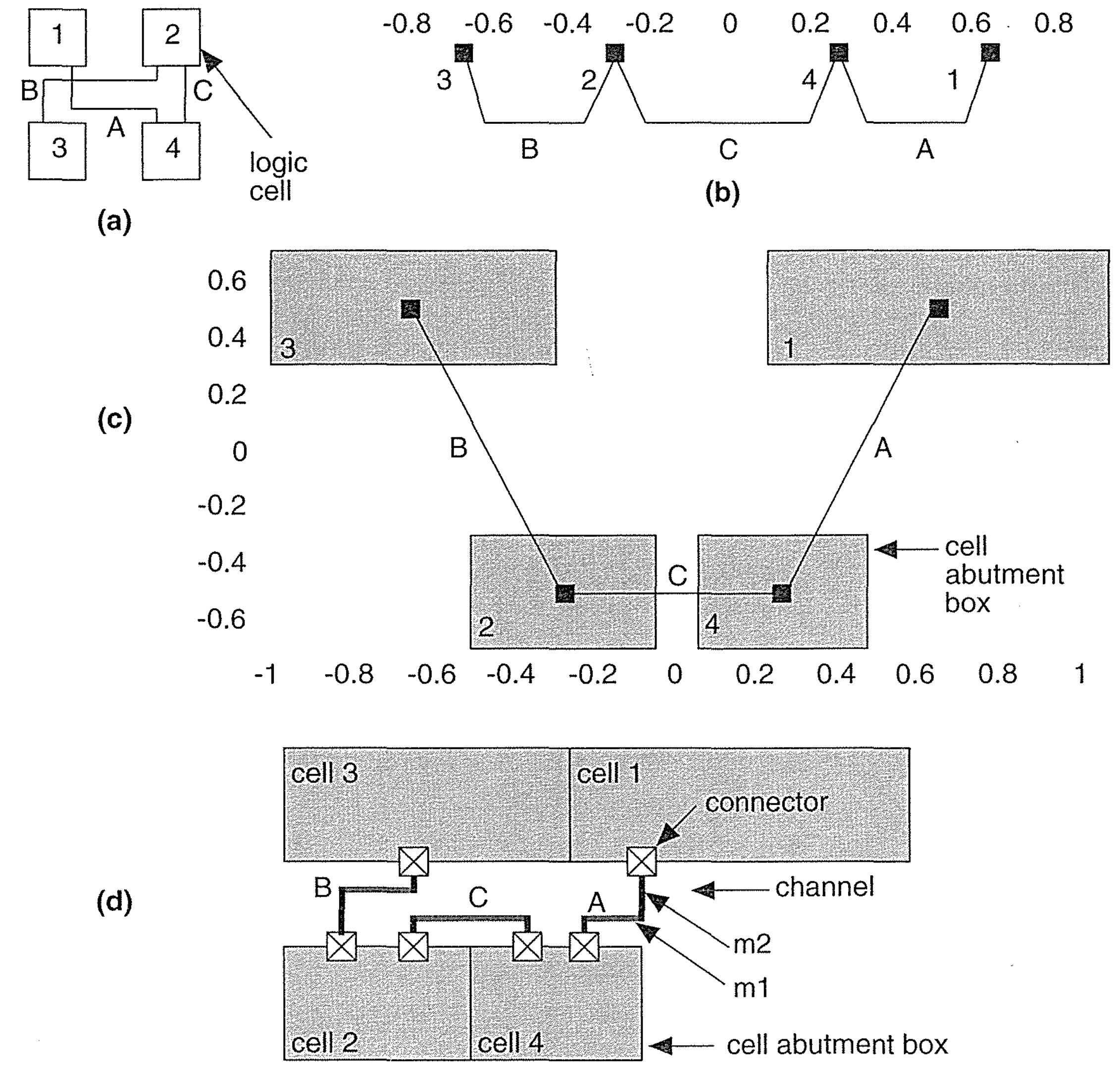


FIGURE 16.25 Eigenvalue placement. (a) An example network. (b) The one-dimensional placement.The small black squares represent the centers of the logic cells. (c) The two-dimensional placement. The eigenvalue method takes no account of the logic cell sizes or actual location of logic cell connectors. (d) A complete layout. We snap the logic cells to valid locations, leaving room for the routing in the channel.

Running this script, we find the eigenvalues of **B** are 0.5858, 0.0, 2.0, and 3.4142. The corresponding eigenvectors of **B** are

$$\begin{bmatrix} 0.6533 & 0.5000 & 0.5000 & -0.2706 \\ -0.2706 & 0.5000 & -0.5000 & -0.6533 \\ -0.6533 & 0.5000 & 0.5000 & 0.2706 \\ 0.2706 & 0.5000 & -0.5000 & 0.6533 \end{bmatrix} . \qquad (16.20)$$

For a one-dimensional placement (Figure 16.25b), we use the eigenvector (0.6533, –0.2706, –0.6533, –0.2706) corresponding to the smallest nonzero eigenvalue (which is 0.5858) to place the logic cells along the x-axis. The two-dimensional placement (Figure 16.25c) uses these same values for the x-coordinates and the eigenvector (0.5, –0.5, 0.5, –0.5) that corresponds to the next largest eigenvalue (which is 2.0) for the y-coordinates. Notice that the placement shown in Figure 16.25(c), which shows logic-cell outlines (the logic-cell abutment boxes), takes no account of the cell sizes, and cells may even overlap at this stage. This is because, in Eq. 16.11, we discarded all but one of the constraints necessary to ensure valid solutions. Often we use the approximate eigenvalue solution as an initial placement for one of the iterative improvement algorithms that we shall discuss in Section 16.2.6.

16.2.6 Iterative Placement Improvement

An **iterative placement improvement** algorithm takes an existing placement and tries to improve it by moving the logic cells. There are two parts to the algorithm:

- The selection criteria that decides which logic cells to try moving.
- The measurement criteria that decides whether to move the selected cells.

There are several **interchange** or **iterative exchange** methods that differ in their selection and measurement criteria:

- pairwise interchange,
- force-directed interchange,
- force-directed relaxation, and
- force-directed pairwise relaxation.

All of these methods usually consider only pairs of logic cells to be exchanged. A source logic cell is picked for trial exchange with a destination logic cell. We have already discussed the use of interchange methods applied to the system partitioning step. The most widely used methods use group migration, especially the Kernighan–Lin algorithm. The **pairwise-interchange algorithm** is similar to the interchange algorithm used for iterative improvement in the system partitioning step:

1. Select the source logic cell at random.
2. Try all the other logic cells in turn as the destination logic cell.
3. Use any of the measurement methods we have discussed to decide on whether to accept the interchange.
4. The process repeats from step 1, selecting each logic cell in turn as a source logic cell.

Figure 16.26(a) and (b) show how we can extend pairwise interchange to swap more than two logic cells at a time. If we swap λ logic cells at a time and find a locally optimum solution, we say that solution is **λ-optimum**. The **neighborhood exchange algorithm** is a modification to pairwise interchange that considers only

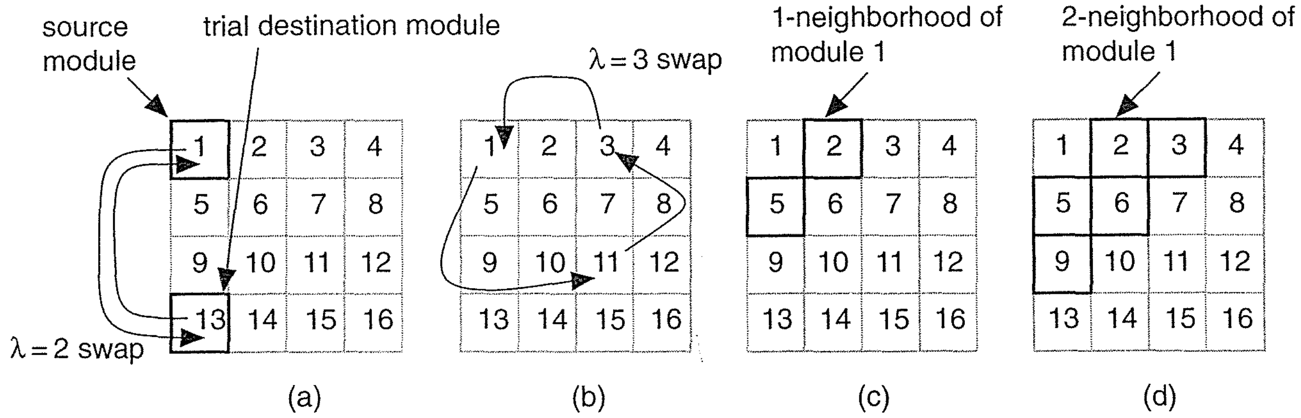


FIGURE 16.26 Interchange. (a) Swapping the source logic cell with a destination logic cell in pairwise interchange. (b) Sometimes we have to swap more than two logic cells at a time to reach an optimum placement, but this is expensive in computation time. Limiting the search to neighborhoods reduces the search time. Logic cells within a distance ε of a logic cell form an ε-neighborhood. (c) A one-neighborhood. (d) A two-neighborhood.

destination logic cells in a **neighborhood**—cells within a certain distance, ε, of the source logic cell. Limiting the search area for the destination logic cell to the **ε-neighborhood** reduces the search time. Figure 16.26(c) and (d) show the one- and two-neighborhoods (based on Manhattan distance) for a logic cell.

Neighborhoods are also used in some of the **force-directed placement methods**. Imagine identical springs connecting all the logic cells we wish to place. The number of springs is equal to the number of connections between logic cells. The effect of the springs is to pull connected logic cells together. The more highly connected the logic cells, the stronger the pull of the springs. The force on a logic cell *i* due to logic cell *j* is given by **Hooke's law**, which says the force of a spring is proportional to its extension:

$$F_{ij} = -c_{ij}x_{ij} \tag{16.21}$$

The vector component x_{ij} is directed from the center of logic cell *i* to the center of logic cell *j*. The vector magnitude is calculated as either the Euclidean or Manhattan distance between the logic cell centers. The c_{ij} form the connectivity or cost matrix (the matrix element c_{ij} is the number of connections between logic cell *i* and logic cell *j*). If we want, we can also weight the c_{ij} to denote critical connections. Figure 16.27 illustrates the force-directed placement algorithm.

In the definition of connectivity (Section 15.7.1, "Measuring Connectivity") it was pointed out that the network graph does not accurately model connections for nets with more than two terminals. Nets such as clock nets, power nets, and global reset lines have a huge number of terminals. The force-directed placement algorithms usually make special allowances for these situations to prevent the largest

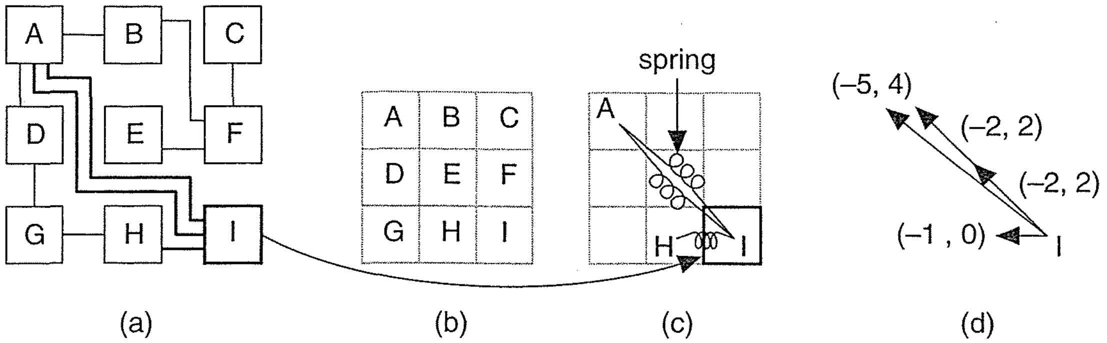


FIGURE 16.27 Force-directed placement. (a) A network with nine logic cells. (b) We make a grid (one logic cell per bin). (c) Forces are calculated as if springs were attached to the centers of each logic cell for each connection. The two nets connecting logic cells A and I correspond to two springs. (d) The forces are proportional to the spring extensions.

nets from snapping all the logic cells together. In fact, without external forces to counteract the pull of the springs between logic cells, the network will collapse to a single point as it settles. An important part of force-directed placement is fixing some of the logic cells in position. Normally ASIC designers use the I/O pads or other external connections to act as anchor points or fixed seeds.

Figure 16.28 illustrates the different kinds of force-directed placement algorithms. The **force-directed interchange** algorithm uses the force vector to select a

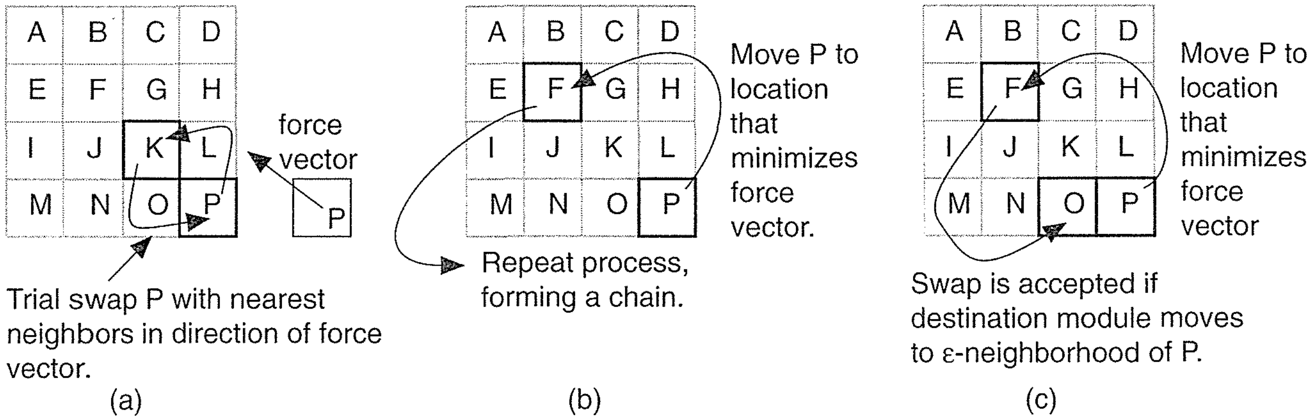

A B C D
E F G H
I J K L
M N O P
force vector
P
Trial swap P with nearest neighbors in direction of force vector.
(a)
Move P to location that minimizes force vector.
Repeat process, forming a chain.
(b)
Move P to location that minimizes force vector
Swap is accepted if destination module moves to ε-neighborhood of P.
(c)

FIGURE 16.28 Force-directed iterative placement improvement. (a) Force-directed interchange. (b) Force-directed relaxation. (c) Force-directed pairwise relaxation.

pair of logic cells to swap. In **force-directed relaxation** a chain of logic cells is moved. The **force-directed pairwise relaxation** algorithm swaps one pair of logic cells at a time.

We reach a force-directed solution when we minimize the energy of the system, corresponding to minimizing the sum of the squares of the distances separating logic cells. Force-directed placement algorithms thus also use a quadratic cost function.

16.2.7 Placement Using Simulated Annealing

The principles of simulated annealing were explained in Section 15.7.8, "Simulated Annealing." Because simulated annealing requires so many iterations, it is critical that the placement objectives be easy and fast to calculate. The optimum connection pattern, the MRST, is difficult to calculate. Using the half-perimeter measure (Section 16.2.3) corresponds to minimizing the total interconnect length. Applying simulated annealing to placement, the algorithm is as follows:

1. Select logic cells for a trial interchange, usually at random.
2. Evaluate the objective function E for the new placement.
3. If ΔE is negative or zero, then exchange the logic cells. If ΔE is positive, then exchange the logic cells with a probability of $\exp(-\Delta E/T)$.
4. Go back to step 1 for a fixed number of times, and then lower the temperature T according to a cooling schedule: $T_{n+1} = 0.9\, T_n$, for example.

Kirkpatrick, Gerlatt, and Vecchi first described the use of simulated annealing applied to VLSI problems [1983]. Experience since that time has shown that simulated annealing normally requires the use of a slow cooling schedule and this means long CPU run times [Sechen, 1988; Wong, Leong, and Liu, 1988]. As a general rule, experiments show that simple min-cut based constructive placement is faster than simulated annealing but that simulated annealing is capable of giving better results at the expense of long computer run times. The iterative improvement methods that we described earlier are capable of giving results as good as simulated annealing, but they use more complex algorithms.

While I am making wild generalizations, I will digress to discuss **benchmarks** of placement algorithms (or any CAD algorithm that is random). It is important to remember that the results of random methods are themselves random. Suppose the results from two random algorithms, A and B, can each vary by ±10 percent for any chip placement, but both algorithms have the same average performance. If we compare single chip placements by both algorithms, they could falsely show algorithm A to be better than B by up to 20 percent or vice versa. Put another way, if we run enough test cases we will eventually find some for which A is better than B by 20 percent—a trick that Ph.D. students and marketing managers both know well. Even single-run evaluations over multiple chips is hardly a fair comparison. The only way to obtain meaningful results is to compare a statistically meaningful number of runs for a statistically meaningful number of chips for each algorithm. This same caution applies to any VLSI algorithm that is random. There was a Design Automation Conference panel session whose theme was "Enough of algorithms claiming improvements of 5 %."

16.2.8 Timing-Driven Placement Methods

Minimizing delay is becoming more and more important as a placement objective. There are two main approaches: net based and path based. We know that we can use net weights in our algorithms. The problem is to calculate the weights. One method finds the *n* most critical paths (using a timing-analysis engine, possibly in the synthesis tool). The net weights might then be the number of times each net appears in this list. The problem with this approach is that as soon as we fix (for example) the first 100 critical nets, suddenly another 200 become critical. This is rather like trying to put worms in a can—as soon as we open the lid to put one in, two more pop out.

Another method to find the net weights uses the **zero-slack algorithm** [Hauge et al., 1987]. Figure 16.29 shows how this works (all times are in nanoseconds). Figure 16.29(a) shows a circuit with **primary inputs** at which we know the **arrival times** (this is the original definition, some people use the term **actual times**) of each signal. We also know the **required times** for the **primary outputs**—the points in time at which we want the signals to be valid. We can work forward from the primary inputs and backward from the primary outputs to determine arrival and required times at each input pin for each net. The difference between the required and arrival times at each input pin is the **slack time** (the time we have to spare). The zero-slack algorithm adds delay to each net until the slacks are zero, as shown in Figure 16.29(b). The net delays can then be converted to weights or constraints in the placement. Notice that we have assumed that all the gates on a net switch at the same time so that the net delay can be placed at the output of the gate driving the net—a rather poor timing model but the best we can use without any routing information.

An important point to remember is that adjusting the net weight, even for every net on a chip, does not theoretically make the placement algorithms any more complex—we have to deal with the numbers anyway. It does not matter whether the net weight is 1 or 6.6, for example. The practical problem, however, is getting the weight information for each net (usually in the form of timing constraints) from a synthesis tool or timing verifier. These files can easily be hundreds of megabytes in size (see Section 16.4).

With the zero-slack algorithm we simplify but overconstrain the problem. For example, we might be able to do a better job by making some nets a little longer than the slack indicates if we can tighten up other nets. What we would really like to do is deal with *paths* such as the critical path shown in Figure 16.29(a) and not just *nets*. Path-based algorithms have been proposed to do this, but they are complex and not all commercial tools have this capability (see, for example, [Youssef, Lin, and Shragowitz, 1992]).

There is still the question of how to predict path delays between gates with only placement information. Usually we still do not compute a routing tree but use simple approximations to the total net length (such as the half-perimeter measure) and then use this to estimate a net delay (the same to each pin on a net). It is not until the routing step that we can make accurate estimates of the actual interconnect delays.

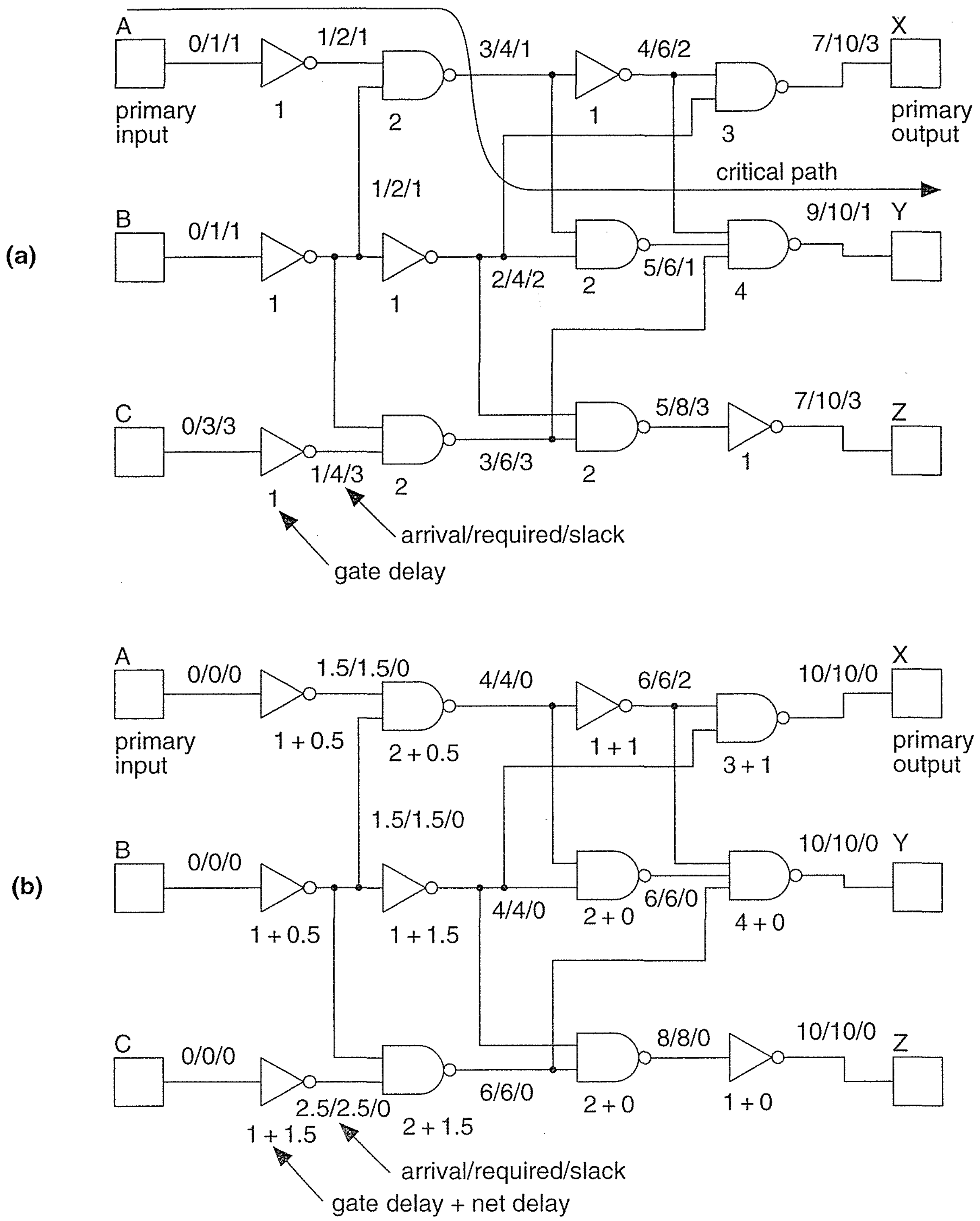
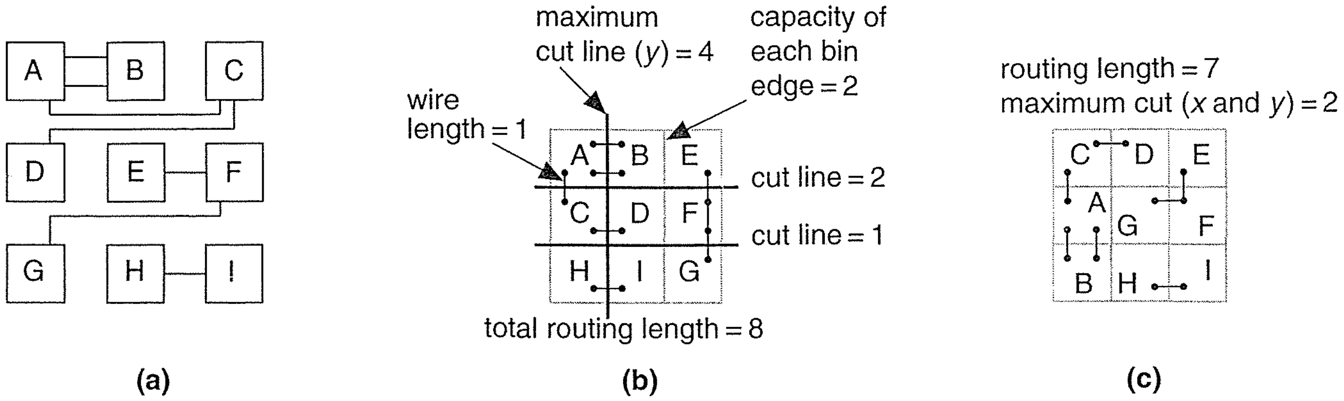


FIGURE 16.29 The zero-slack algorithm. (a) The circuit with no net delays. (b) The zero-slack algorithm adds net delays (at the outputs of each gate, equivalent to increasing the gate delay) to reduce the slack times to zero.

16.2.9 A Simple Placement Example

Figure 16.30 shows an example network and placements to illustrate the measures for interconnect length and interconnect congestion. Figure 16.30(b) and (c) illustrate the meaning of total routing length, the maximum cut line in the x-direction, the maximum cut line in the y-direction, and the maximum density. In this example we have assumed that the logic cells are all the same size, connections can be made to terminals on any side, and the routing channels between each adjacent logic cell have a capacity of 2. Figure 16.30(d) shows what the completed layout might look like.



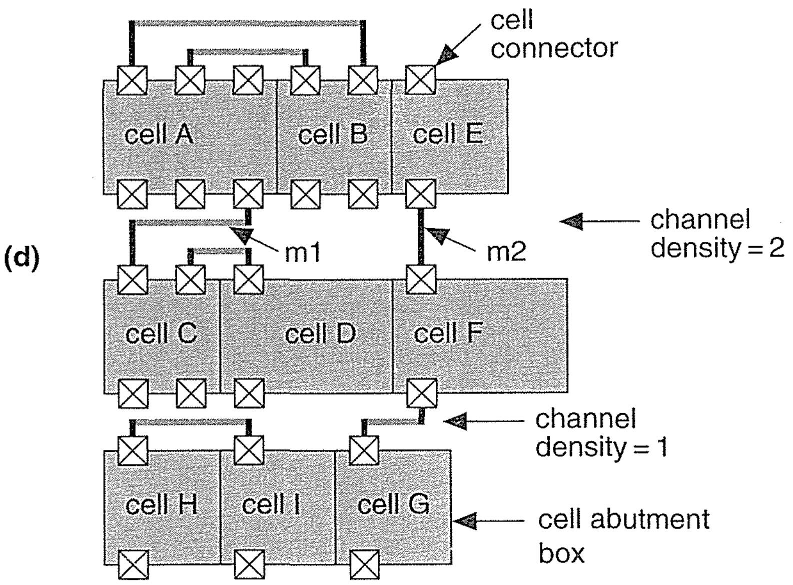


FIGURE 16.30 Placement example. (a) An example network. (b) In this placement, the bin size is equal to the logic cell size and all the logic cells are assumed equal size. (c) An alternative placement with a lower total routing length. (d) A layout that might result from the placement shown in b. The channel densities correspond to the cut-line sizes. Notice that the logic cells are not all the same size (which means there are errors in the interconnect-length estimates we made during placement).

16.3 Physical Design Flow

Historically placement was included with routing as a single tool (the term P&R is often used for place and route). Because interconnect delay now dominates gate delay, the trend is to include placement within a floorplanning tool and use a separate router. Figure 16.31 shows a design flow using synthesis and a floorplanning tool that includes placement. This flow consists of the following steps:

1. *Design entry.* The input is a logical description with no physical information.
2. *Synthesis.* The initial synthesis contains little or no information on any interconnect loading. The output of the synthesis tool (typically an EDIF netlist) is the input to the floorplanner.
3. *Initial floorplan.* From the initial floorplan interblock capacitances are input to the synthesis tool as load constraints and intrablock capacitances are input as wire-load tables.
4. *Synthesis with load constraints.* At this point the synthesis tool is able to resynthesize the logic based on estimates of the interconnect capacitance each

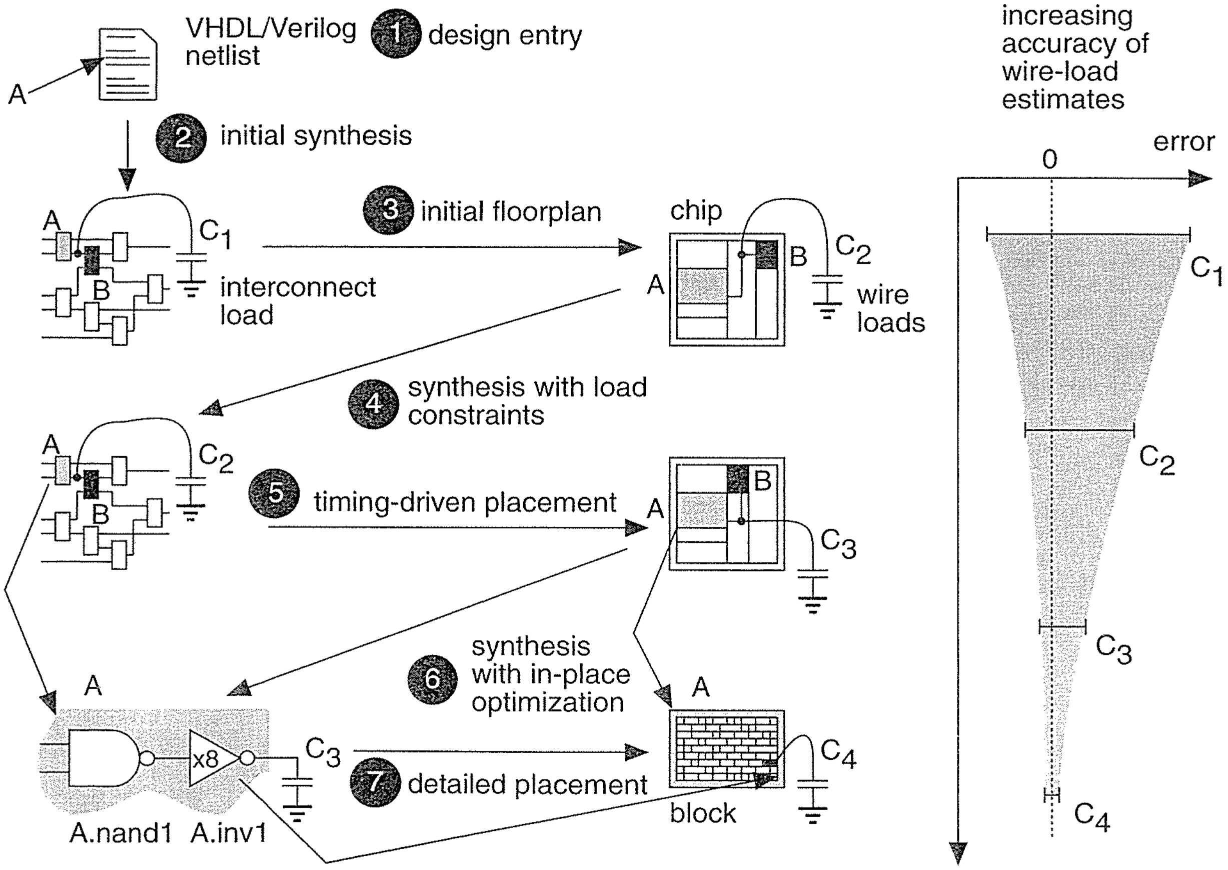


FIGURE 16.31 Timing-driven floorplanning and placement design flow. Compare with Figure 15.1 on p. 806.

gate is driving. The synthesis tool produces a forward annotation file to constrain path delays in the placement step.

5. *Timing-driven placement.* After placement using constraints from the synthesis tool, the location of every logic cell on the chip is fixed and accurate estimates of interconnect delay can be passed back to the synthesis tool.
6. *Synthesis* with **in-place optimization** (**IPO**). The synthesis tool changes the drive strength of gates based on the accurate interconnect delay estimates from the floorplanner without altering the netlist structure.
7. *Detailed placement.* The placement information is ready to be input to the routing step.

In Figure 16.31 we iterate between floorplanning and synthesis, continuously improving our estimate for the interconnect delay as we do so.

16.4 Information Formats

With the increasing importance of interconnect a great deal of information needs to flow between design tools. There are some de facto standards that we shall look at next. Some of the companies involved are working toward releasing these formats as IEEE standards.

16.4.1 SDF for Floorplanning and Placement

In Section 13.5.6, "SDF in Simulation," we discussed the structure and use of the standard delay format (SDF) to describe gate delay and interconnect delay. We may also use SDF with floorplanning and synthesis tools to **back-annotate** an interconnect delay. A synthesis tool can use this information to improve the logic structure. Here is a fragment of SDF:

```
(INSTANCE B) (DELAY (ABSOLUTE
  (INTERCONNECT A.INV8.OUT B.DFF1.Q (:0.6:) (:0.6:))))
```

In this example the rising and falling delay is 60 ps (equal to 0.6 units multiplied by the time scale of 100 ps per unit specified in a TIMESCALE construct that is not shown). The delay is specified between the output port of an inverter with instance name A.INV8 in block A and the Q input port of a D flip-flop (instance name B.DFF1) in block B. A '.' (period or fullstop) is set to be the hierarchy divider in another construct that is not shown.

There is another way of specifying interconnect delay using NETDELAY (a short form of the INTERCONNECT construct) as follows:

```
(TIMESCALE 100ps) (INSTANCE B) (DELAY (ABSOLUTE
(NETDELAY net1 (0.6)))
```

In this case all delays from an output port to, possibly multiple, input ports have the same value (we can also specify the output port name instead of the net name to identify the net). Alternatively we can lump interconnect delay at an input port:

```
  (TIMESCALE 100ps) (INSTANCE B.DFF1) (DELAY (ABSOLUTE
    (PORT CLR (16:18:22) (17:20:25))))
```

This PORT construct specifies an interconnect delay placed at the input port of a logic cell (in this case the CLR pin of a flip-flop). We do not need to specify the start of a path (as we do for INTERCONNECT).

We can also use SDF to **forward-annotate** path delays using **timing constraints** (there may be hundreds or thousands of these in a file). A synthesis tool can pass this information to the floorplanning and placement steps to allow them to create better layout. SDF describes timing checks using a range of TIMINGCHECK constructs. Here is an example of a single path constraint:

```
(TIMESCALE 100ps) (INSTANCE B) (TIMINGCHECK
  (PATHCONSTRAINT A.AOI22_1.O B.ND02_34.O (0.8) (0.8)))
```

This describes a constraint (keyword PATHCONSTRAINT) for the rising and falling delays between two ports at each end of a path (which may consist of several nets) to be less than 80 ps. Using the SUM construct we can constrain the sum of path delays to be less than a specific value as follows:

```
(TIMESCALE 100ps) (INSTANCE B) (TIMINGCHECK
  (SUM (AOI22_1.O ND02_34.I1) (ND02_34.O ND02_35.I1) (0.8)))
```

We can also constrain skew between two paths (in this case to be less than 10 ps) using the DIFF construct:

```
(TIMESCALE 100ps) (INSTANCE B) (TIMINGCHECK
  (DIFF (A.I_1.O B.ND02_1.I1) (A.I_1.O.O B.ND02_2.I1) (0.1)))
```

In addition we can constrain the skew between a reference signal (normally the clock) and all other ports in an instance (again in this case to be less than 10 ps) using the SKEWCONSTRAINT construct:

```
(TIMESCALE 100ps) (INSTANCE B) (TIMINGCHECK
  (SKEWCONSTRAINT (posedge clk) (0.1)))
```

At present there is no easy way in SDF to constrain the skew between a reference signal and other signals to be greater than a specified amount.

16.4.2 PDEF

The **physical design exchange format** (**PDEF**) is a proprietary file format used by Synopsys to describe placement information and the clustering of logic cells. Here is a simple, but complete PDEF file:

```
(CLUSTERFILE
  (PDEFVERSION "1.0")
  (DESIGN "myDesign")
  (DATE "THU AUG 6 12:00 1995")
  (VENDOR "ASICS_R_US")
  (PROGRAM "PDEF_GEN")
  (VERSION "V2.2")
  (DIVIDER .)
  (CLUSTER (NAME "ROOT")
    (WIRE_LOAD "10mm x 10mm")
    (UTILIZATION 50.0)
    (MAX_UTILIZATION 60.0)
    (X_BOUNDS 100 1000)
    (Y_BOUNDS 100 1000)
      (CLUSTER (NAME "LEAF_1")
        (WIRE_LOAD "50k gates")
        (UTILIZATION 50.0)
        (MAX_UTILIZATION 60.0)
        (X_BOUNDS 100 500)
        (Y_BOUNDS 100 200)
        (CELL (NAME L1.RAM01)
        (CELL (NAME L1.ALU01)
        )
    )
)
```

This file describes two clusters:

- ROOT, which is the top-level (the whole chip). The file describes the size (*x*- and *y*-bounds), current and maximum area utilization (i.e., leaving space for interconnect), and the name of the wire-load table, '10mm x 10mm', to use for this block, chosen because the chip is expected to be about 10 mm on a side.
- LEAF_1, a block below the top level in the hierarchy. This block is to use predicted capacitances from a wire-load table named '50k gates' (chosen because we know there are roughly 50 k-gate in this block). The LEAF_1 block contains two logic cells: L1.RAM01 and L1.ALU01.

16.4.3 LEF and DEF

The **library exchange format** (**LEF**) and **design exchange format** (**DEF**) are both proprietary formats originated by Tangent in the TanCell and TanGate place-and-route tools which were bought by Cadence and now known as Cell3 Ensemble and Gate Ensemble respectively. These tools, and their derivatives, are so widely used that these formats have become a de facto standard. LEF is used to define an IC process and a logic cell library. For example, you would use LEF to describe a gate array: the base cells, the legal sites for base cells, the logic macros with their size and connectivity information, the interconnect layers and other information to set up the database that the physical design tools need. You would use DEF to describe all the physical aspects of a particular chip design including the netlist and physical location of cells on the chip. For example, if you had a complete placement from a floorplanning tool and wanted to exchange this information with Cadence Gate Ensemble or Cell3 Ensemble, you would use DEF.

16.5 Summary

Floorplanning follows the system partitioning step and is the first step in arranging circuit blocks on an ASIC. There are many factors to be considered during floorplanning: minimizing connection length and signal delay between blocks; arranging fixed blocks and reshaping flexible blocks to occupy the minimum die area; organizing the interconnect areas between blocks; planning the power, clock, and I/O distribution. The handling of some of these factors may be automated using CAD tools, but many still need to be dealt with by hand. Placement follows the floorplanning step and is more automated. It consists of organizing an array of logic cells within a flexible block. The criterion for optimization may be minimum interconnect area, minimum total interconnect length, or performance. There are two main types of placement algorithms: based on min-cut or eigenvector methods. Because interconnect delay in a submicron CMOS process dominates logic-cell delay, planning of interconnect will become more and more important. Instead of completing synthesis before starting floorplanning and placement, we will have to use synthesis and floorplanning/placement tools together to achieve an accurate estimate of timing.

The key points of this chapter are:

- Interconnect delay now dominates gate delay.
- Floorplanning is a mapping between logical and physical design.
- Floorplanning is the center of ASIC design operations for all types of ASIC.
- Timing-driven floorplanning is becoming an essential ASIC design tool.
- Placement is now an automated function.

16.6 Problems

* = Difficult, ** = Very difficult, *** = Extremely difficult

16.1 (Wire loads, 30 min.) Table 16.2 shows a wire-load table. Since you might expect the interconnect load to be a monotonic increasing function of fanout and block area, it seems as though some of the data in Table 16.2 may be in error; these figures are shown preceded by an asterisk, '*' (this table is from an ASIC vendor data book). Using a spreadsheet, analyze the data in Table 16.2.

a. By graphing the data, indicate any figures in Table 16.2 that you think might be in error. If you think that there is an error, predict the correct values—either by interpolation (for values in error in the body of the table), or by fitting the linear model parameters, the slope and the intercept (for any values in error in the last two columns of the table).

b. Including any corrections, how accurate is the model that predicts load as a linear function of fanout for a given block size? (Use the maximum error of the linear model expressed as a percentage of the table value.)

c. Can you fit a simple function to the (possibly corrected) figures in the last column of the table and explain its form?

d. What did you learn about wire-load tables from this problem?

TABLE 16.2 Wire-load table. Predicted interconnect loads (measured in standard loads) as a function of block size and fanout (Problem 16.1).

Size ($/mm^2$)	Fanout											Slope	Intercept
	1	2	3	4	5	6	7	8	16	32	64		
0.5×0.5	0.65	0.95	1.25	1.54	1.84	2.14	2.44	2.74	5.13	9.91	19.47	0.299	0.349
1×1	0.80	1.20	1.59	1.99	2.39	2.79	3.19	3.59	6.77	13.15	25.9	0.398	0.398
2×2	0.96	1.48	1.99	2.51	3.02	3.54	4.05	4.57	8.68	16.92	33.38	0.515	0.448
3×3	1.20	1.83	2.46	3.09	3.72	4.35	4.98	5.61	10.66	20.75	40.94	0.631	0.564
4×4	1.41	2.11	2.81	3.50	4.20	4.90	5.59	6.29	11.87	23.02	45.33	0.697	0.714
5×5	1.51	2.24	2.97	3.70	4.43	5.16	5.89	6.62	12.47	24.15	47.53	0.730	0.780
6×6	1.56	2.31	3.05	3.80	4.55	5.30	6.04	6.79	12.77	24.72	48.62	0.747	0.813
7×7	1.83	2.62	3.42	4.22	5.01	5.81	6.61	7.40	13.78	26.53	52.02	0.797	*1.029
8×8	1.88	2.74	3.6	4.47	5.33	6.19	7.06	7.92	14.82	26.64	56.26	0.863	1.013
9×9	2.01	2.94	3.87	4.80	5.73	6.66	7.59	8.52	15.95	30.83	60.57	0.930	1.079
10×10	2.01	2.98	3.94	4.90	5.86	6.83	7.79	8.75	16.45	31.86	62.67	0.963	*1.050
11×11	2.46	3.46	4.45	5.45	6.44	7.44	8.44	9.43	17.4	33.33	65.20	0.996	1.465
12×12	3.04	4.1	5.17	6.23	7.3	8.35	9.42	10.48	18.8	36.03	70.00	1.063	1.964

16.2 (Trees, 20 min.) For the network graph shown in Figure 16.32(f), draw the following trees and calculate their Manhattan lengths:

a. The minimum Steiner tree.

b. The **chain connection**.

c. The minimum rectilinear Steiner tree.

d. The **minimum rectilinear spanning tree** [Hwang, 1976].

e. The minimum single-trunk rectilinear Steiner tree (with a horizontal or vertical trunk).

f. The **minimum rectilinear chain connection** (easy to compute).

g. The **minimum source-to-sink connection.**

Calculate:

h. The complete-graph measure and the half-perimeter measure.

Figure 16.32 parts (a–e) illustrate the definitions of these trees. There is no known solution to the minimum Steiner-tree problem for nets with more than five terminals.

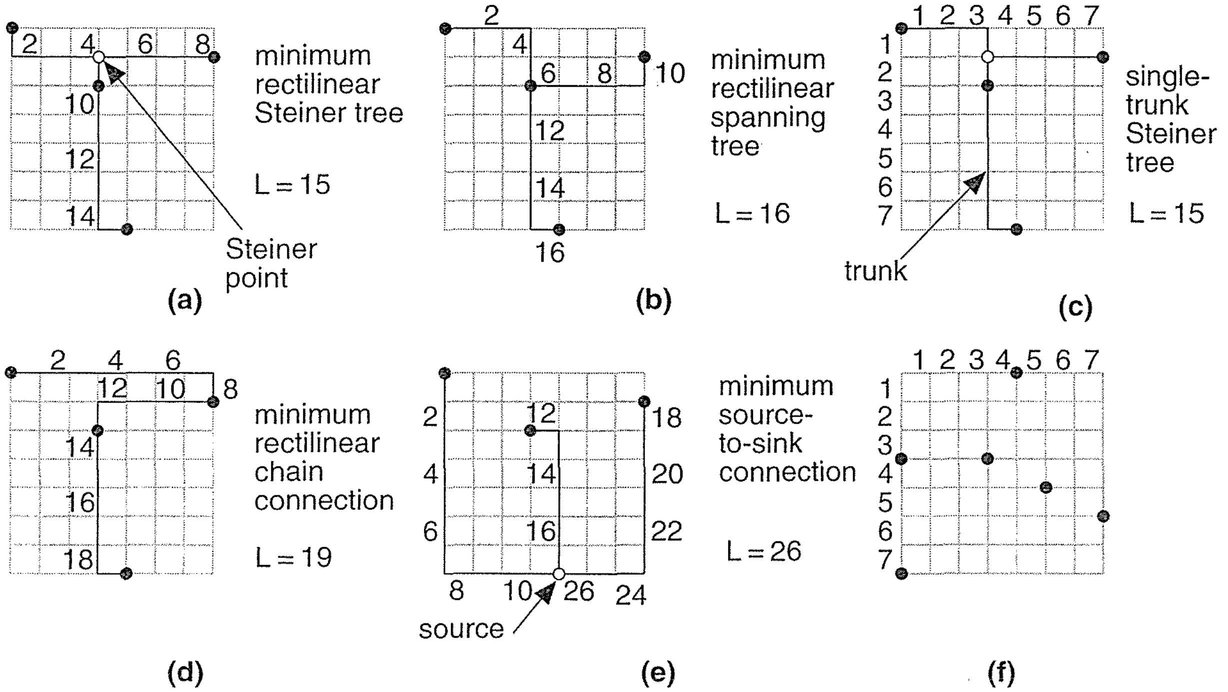


FIGURE 16.32 Tree routing. (a) The minimum rectilinear Steiner tree (MRST). (b) The minimum rectilinear spanning tree. (c) The minimum single-trunk rectilinear Steiner tree (1-MRST). (d) The minimum rectilinear chain connection. (e) The minimum source-to-sink connection. (f) Example net for Problem 16.2.

16.3 (Eigenvalue placement constraints, 10 min. [Cheng and Kuh, 1984]) Consider the one-dimensional placement problem with a vector list of valid positions for the logic cells $\mathbf{p} = [p_i]$ and a vector list of x-coordinates for the logic cells $\mathbf{x} = [x_i]$.

Show that for a valid placement $\mathbf{x}$ (where the vector elements x_i are some permutation of the vector elements p_i), the following equations hold:

$$\sum_{i=1}^{n} x_i = \sum_{i=1}^{n} p_i \qquad \sum_{i=1}^{n} x_i^2 = \sum_{i=1}^{n} p_i^2 \qquad \ldots \qquad \sum_{i=1}^{n} x_i^n = \sum_{i=1}^{n} p_i^n \tag{16.22}$$

(*Hint:* Consider the polynomial $(x + x_i)^n$. In our simplification to the problem, we chose to impose only the second equation of these constraints.)

16.4 (*Eigenvalue placement, 30 min.) You will need MatLab, Mathematica, or a similar mathematical calculus program for this problem.

a. Find the eigenvalues and eigenvectors for the disconnection matrix corresponding to the following connection matrix:

```
C=
[ 0 1 1 0 0 0 1 0 0;
  1 0 0 0 0 0 0 0 0;
  1 0 0 1 0 0 0 1 0;
  0 0 1 0 0 1 0 0 0;
  0 0 0 0 0 1 0 0 1;
  0 0 0 1 1 0 1 0 0;
  1 0 0 0 0 1 0 0 0;
  0 0 1 0 0 0 0 0 1;
  0 0 0 0 1 0 0 1 0;]
```

(*Hint:* Check your answer. The smallest, nonzero, eigenvalue should be 0.5045.)

b. Use your results to place the logic cells. Plot the placement and show the connections between logic cells (this is easy to do using an X-Y plot in an Excel spreadsheet).

c. Check that the following equation holds:

$$\lambda = \frac{g}{\mathrm{P}}.$$

16.5 (Die size, 10 min.) Suppose the minimum spacing between pad centers is W mil (1 mil = 10^{-3} inch), there are N I/O pads on a chip, and the die area (assume a square die) is A mil^2:

a. Derive a relationship between W, N, and A that corresponds to the point at which the die changes from being pad-limited to core-limited.

b. Plot this relationship with N (ranging from 50 to 500 pads) on the x-axis, A on the y-axis (for dies ranging in size from 1 mm to 20 mm on a side), and W as a parameter (for $W = 1, 2, 3,$ and 4 mil).

16.6 (Power buses, 20 min.) Assume aluminum metal interconnect has a resistance of about 30 mΩ/square (a low value). Consider a power ring for the I/O pads. Suppose you have a high-power chip that dissipates 5 W at $V_{DD} = 5$ V, and assume that half of the supply current (0.5 A) is due to I/O. Suppose the square die is L mil on a side, and that the I/O current is equally distributed among the N VDD pads that are on the chip. In the worst case, you want no more than 100 mV drop between any VDD pad and the I/O circuits drawing power (notice that there will be an equal drop on the VSS side; just consider the VDD drop).

a. Model the power distribution as a ring of N equally spaced pads. Each pad is connected by a resistor equal to the aluminum VDD power-bus resistance between two pads. Assume the I/O circuits associated with each pad can be considered to connect to just one point on the resistors between each pad. If the resistance between each pad is R, what is the worst-case resistance between the I/O circuits and the supply?

b. Plot a graph showing L (in mil) on the x-axis, W (the required power-bus width in microns) on the y-axis, with N as a parameter (with $N = 1, 2, 5, 10$).

c. Comment on your results.

d. An upper limit on current density for aluminum metallization is about $50\,\text{kAcm}^{-2}$; at current densities higher than this, failure due to electromigration (which we shall cover in Section 17.3.2, "Power Routing") is a problem. Assume the metallization is $0.5\,\mu\text{m}$ thick. Calculate the current density in the VDD power bus for this chip in terms of the power-bus width and the number of pads. Comment on your answer.

16.7 (Interconnect-length approximation, 10 min.) Figure 16.22 shows the correlation between actual interconnect length and two approximations. Use this graph to derive a correction function (together with an estimation of the error) for the complete-graph measure and the half-perimeter measure.

16.8 (Half-perimeter measure, 10 min.) Draw a tree on a rectangular grid for which the MRST is equal to the half-perimeter measure. Draw a tree on a rectangular grid for which the MRST is twice the half-perimeter measure.

16.9 (***Min-cut, 120 min.) Many floorplanning and placement tools use min-cut methods and allow you to alter the type and sequence of bisection cuts. Research and describe the difference between: quadrature min-cut placement, bisection min-cut placement, and slice/bisection min-cut placement.

16.10 (***Terminal propagation, 120 min.) There is a problem with the min-cut algorithm in the way connectivity is measured. Figure 16.33 shows a situation in which logic cells G and H are connected to other logic cells (A and F) outside the area R_1 that is currently being partitioned. The min-cut algorithm ignores connections outside the area to be divided. Thus logic cells G and H may be placed in partition R_3 rather than partition R_2. Suggest solutions to this problem. *Hint:* See Dunlop [1983]; Hartoog [1986]; or the Barnes–Hut galaxy model.

16.11 (Benchmarks and statistics, 30 min.) Your boss asks you to compare two placement programs from companies ABC and XYZ. You run five test cases for both on a single netlist, P1. You get results (measured in arbitrary units) of 9, 8, 9, 7, 11 for ABC; 6, 9, 10, 13, 8 for XYZ.

a. Calculate the mean and standard deviations for these results.

b. What confidence (in the statistical sense) do you have in these figures?

c. What can you say about the relative performance of ABC and XYZ?

On average each test case takes about 0.5 hours (wall clock) for both ABC and XYZ. Next you run six test cases on another netlist, P2 with the following results: 4, 6, 7, 8, 5, 7 for ABC, and 4, 5, 3, 6, 4, 3 for XYZ. These test cases take about 0.75 hours (wall clock) each.

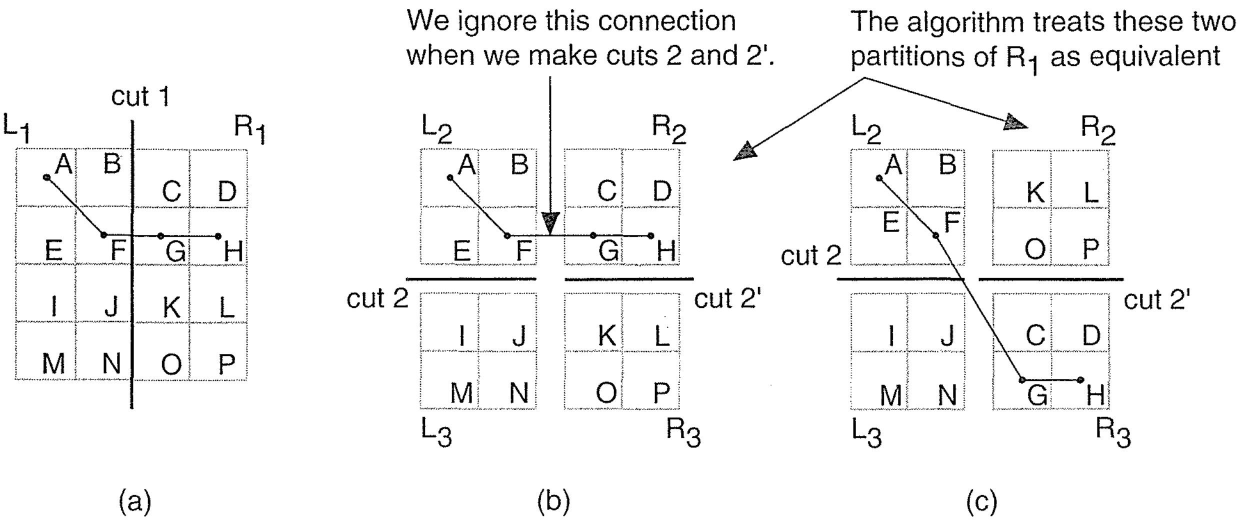

We ignore this connection when we make cuts 2 and 2'.
The algorithm treats these two partitions of R1 as equivalent
cut 1
L1
R1
A B C D
E F G H
I J K L
M N O P
L2
R2
cut 2
cut 2'
L3
R3
(a)
(b)
(c)

FIGURE 16.33 (For Problem 16.10.) A problem with the min-cut algorithm is that it ignores connections to logic cells outside the area being partitioned. (a) We perform a vertical cut 1 producing the areas R_1 and R_2. (b) Next we make a horizontal cut 2, producing L_2 and L_3, and a cut 2', producing R_2 and R_3. (c) The min-cut algorithm ignores the connection from L_2 and is equally likely to produce the arrangement shown here when we make cut 2'.

d. What can you say about the P2 results?

e. Given the P1 and P2 results together, what can you say about ABC and XYZ?

f. How many P1 test cases should you run to get a result so that you can say ABC is better or worse than XYZ with 90 percent confidence (i.e., you make the right decision 9 out of 10 times)? How long would this take?

g. Find the same figures for the P2 netlist. Comment on your answers.

h. Suppose you had more netlists and information about the variation of results from each netlist, together with the average time to run each netlist. How would you use this information to get the most meaningful result in the shortest time?

16.12 (Linear and quadratic placement, 20 min.) [Sigl, Doll, and Johannes, 1991] Figure 16.34(a) shows a simple network that we will place. Figure 16.34(b) shows the problem. The logic cells are all the same size: 1 grid unit wide by 1 grid unit high. Logic cells 1 and 3 are fixed at the locations shown. Logic cell 2 is movable and placed at coordinates (for the lower-left corner) of (x_2, y_2). The lower-left corners of logic cells should be placed at grid locations and should not overlap.

a. What is the connection matrix c_{ij} for this network?

b. Calculate and draw (showing the logic-cell coordinates) the placement that minimizes the linear cost function (or objective function) f_L,

$$f_L = \frac{1}{2} \sum_{i,j=1}^{n} c_{ij} d_{ij} \qquad (16.23)$$

where d_{ij} is the distance between logic cells i and j.

c. Calculate and draw (showing coordinates) the placement that minimizes the quadratic cost function f_Q,

$$f_Q = \frac{1}{2} \sum_{i,j=1}^{n} c_{ij} d_{ij}^2 . \qquad (16.24)$$

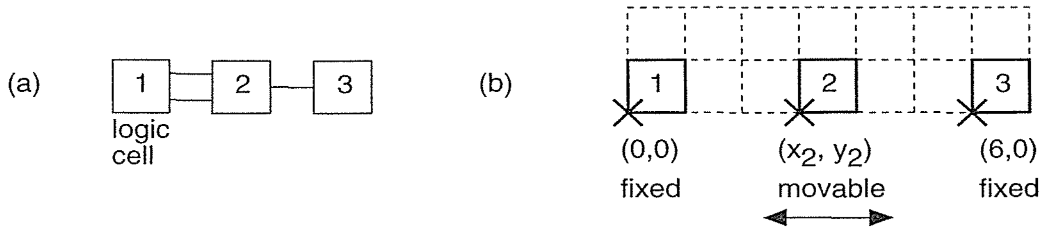


FIGURE 16.34 Problem 16.12 illustrates placement objectives. (a) An example network for placement. (b) The placement restrictions. Logic cells 1 and 3 are fixed in position, the placement problem is to optimize the position of logic cell 2 under different placement objectives.

16.13 (Placement interconnect lengths, 45 min.) Figure 16.30(d) shows the actual routing corresponding to a placement with an estimated routing length of 8 units (Figure 16.30b).

a. Draw the layout (with routing) corresponding to the placement of Figure 16.30(c), which has a lower estimated total routing length of 7 units.

b. Compare the actual total routing length for both layouts and explain why they are different from the estimated lengths and describe the sources of the errors.

c. Consider flipping both logic cells A and B about the y-axis in the layout shown in Figure 16.30(d). How much does this shorten the total interconnect length? Some placement algorithms consider such moves.

16.14 (Zero-slack algorithm, 60 min.) For the circuit of Figure 16.35:

a. Find all of the arrival, required, and slack times (all delays are in nanoseconds).

b. What is the critical path?

c. If the gate delay of A2 is increased to 5 ns, what is the new critical path?

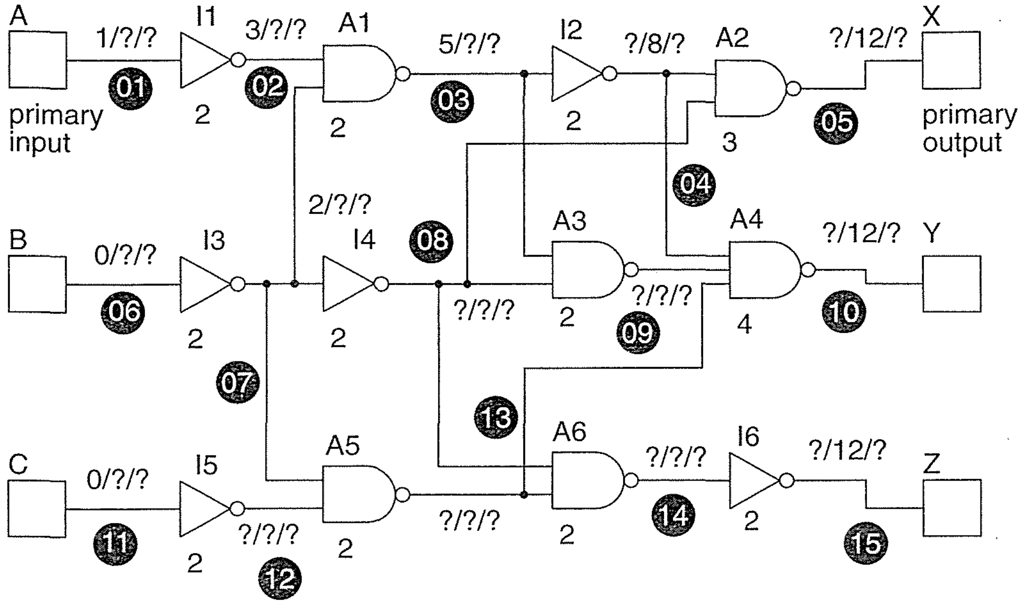


FIGURE 16.35 A circuit to illustrate the zero-slack algorithm (Problem 16.14).

d. ** Using your answer to part a find the upper bounds on net delays by means of the zero-slack algorithm as follows:

i. Find arrival, required, and slack times on all nets.

ii. Find an input pin p with the least nonzero slack S_p on a net which has not already been selected. If there are none go to step 6.

ii. Find the path through p (may include several gates) on which all pins have slack S_p.

iv. Distribute a delay equal to the slack S_p along the path assigning a fraction to each net at the output pins of the gates on the path.

v. Work backward from p updating all the required times as necessary and forward from p updating all the arrival times.

vi. Convert net delays to net lengths.

Hint: You can consult the original description of the zero-slack algorithm if this is not clear [Hauge et al., 1987].

16.15 (World planning, 60 min.) The seven continents are (with areas in millions of square miles): Europe—strictly a peninsula of Asia (4.1), Asia (17.2), North America (9.4), South America (6.9), Australia (3.0), Africa (11.7), and Antarctica (5.1). Assume the continents are flexible blocks whose aspect ratio may be adjusted.

a. Create a slicing floorplan of the world with a square aspect ratio.

b. Draw a world connectivity graph with seven nodes and whose edges are labeled with the distances between Moscow, Beijing, Chicago, Rio de Janeiro, Sydney, Nairobi, and the South Pole.

c. Suppose you want to floorplan the world so that the difference in distances between the centers of the continental blocks and the corresponding edges in the world connectivity graph is minimized. How would you measure the differences in distance? Suggest a method to minimize your measure.

d. Use an eigenvalue method to floorplan the world. Draw the result with coordinates for each block and explain your approach.

16.7 Bibliography

There are no recent monographs or review articles on floorplanning modern ASICs with interconnect delay dominating gate delay. Placement is a much more developed topic. Perhaps the simplest place to dig deeper is the book by Preas and Lorenzetti that contains a chapter titled "Placement, assignment, and floorplanning" [Preas and Karger, 1988]. The collection edited by Ohtsuki [1986] contains a review paper by Yoshida titled "Partitioning, assignment, and placement." Sangiovanni-Vincentelli's review article [1986] complements Ohtsuki's edited book, but both are now dated. Sechen's book [1988] describes simulated annealing and its application to placement and chip-planning for standard cell and gate array ASICs. Part III of the IEEE Press book edited by Hu and Kuh [1983] is a collection of papers on wireability, partitioning, and placement covering some of the earlier and fundamental work in this area. For a more recent and detailed look at the inner workings of floorplanning and placement tools, Lengauer's [1990] book on algorithms contains a chapter on graph algorithms and a chapter on placement, assignment, and floorplanning. Most of these earlier book references deal with placement before the use of timing as an additional objective. The tutorial paper by Benkoski and Strojwas [1991] contains a number of references on performance-driven placement. Luk's book [1991] describes methods for estimating net delay during placement.

Papers and tutorials on all aspects of floorplanning and placement (with an emphasis on algorithms) are published in *IEEE Transactions on Computer-Aided Design.* The newest developments in floorplanning and placement appear every year in the *Proceedings of the ACM/IEEE Design Automation Conference* (DAC) and *Proceedings of the IEEE International Conference on Computer-Aided Design* (ICCAD).

16.8 References

Page numbers in brackets after a reference indicate its location in the chapter body.

Benkoski, J., and A. J. Strojwas. 1991. "The role of timing verification in layout synthesis." In *Proceedings of the 28th ACM/IEEE Design Automation Conference,* San Francisco, pp. 612–619. Tutorial paper with 60 references. This was an introduction to a session on Placement for Performance Optimization containing five other papers on this topic. [p. 906]

Breuer, M. A. 1977. “Min-cut placement.” *Journal of Design Automation and Fault Tolerant Computing,* Vol. 1, no. 4, pp. 343–362. [p. 882]

Chao, A. H., E. M. Nequist, and T. D. Vuong. 1990. “Direct solution of performance constraints during placement.” In *Proceedings of the IEEE Custom Integrated Circuits Conference.* Describes algorithms used in Cadence Gate Ensemble for performance-driven placement. Wiring estimate is based on single trunk Steiner tree with corrections for bounding rectangle aspect ratio and pin count. [p. 879]

Cheng, C.-K., and E. S. Kuh. 1984. “Module placement based on resistive network optimization.” *IEEE Transactions on Computer-Aided Design for Integrated-Circuits and Systems,* Vol. CAD-3, pp. 218–225. [pp. 884, 900]

Dunlop, A. E., and B. W. Kernighan. 1983. “A placement procedure for polycell VLSI circuits.” In *Proceedings of the IEEE International Conference on Computer Aided Design,* Santa Clara, CA, September 13–15. Describes the terminal propagation algorithm. [p. 902]

Goto, S. and T. Matsuda. 1986. “Partitioning, assignment and placement.” In *Layout Design and Verification,* T. Ohtsuki (Ed.), Vol. 4, pp. 55–97. New York: Elsevier. ISBN 0444878947. TK 7874. L318. [p. 879]

Hall, K. M. 1970. “An r-dimensional quadratic placement algorithm.” *Management Science,* Vol. 17, no. 3, pp. 219–229. [p. 885]

Hanan, M. 1966. “On Steiner's problem with rectilinear distance.” *Journal SIAM Applied Mathematics,* Vol. 14, no. 2, pp. 255–265. [p. 879]

Hanan, M., P. K. Wolff Sr., and B. J. Agule. 1973. “Some experimental results on placement techniques.” In *Proceedings of the 13th Design Automation Conference.* Reference to complete graph wire measure. [p. 877]

Hartoog, M. R., 1986. “Analysis of placement procedures for VLSI standard cell layout.” In *Proceedings of the 23rd Design Automation Conference.* [p. 902]

Hauge, P. S., et al. 1987. “Circuit placement for predictable performance.” In *Proceedings of the IEEE International Conference on Computer Aided Design,* pp. 88–91. Describes the zero-slack algorithm. *See also:* Nair, R., C. L. Berman, P. S. Hauge, and E. J. Yoffa, “Generation of performance constraints for layout,” *IEEE Transactions on Computer Aided Design,* Vol. 8, no. 8, pp. 860–874, August 1989; and Burstein, M. and M. N. Housewife, “Timing influenced layout design,” in *Proceedings of the 22nd Design Automation Conference,* 1985. Defines required, actual, and slack times. Describes application of timing-driven restrictions to placement using F–M algorithm and hierarchical global routing. [p. 905]

Hu, T. C., and E. S. Kuh (Eds.). 1983. *VLSI Circuit Layout: Theory and Design.* New York: IEEE Press. Contains 26 papers divided into six parts; Part 1: Overview; Part II: General; Part III: Wireability, Partitioning and Placement; Part IV: Routing; Part V: Layout Systems; Part VI: Module Generation. ISBN 0879421932. TK7874. V5573. [p. 906]

Hwang, F. K. 1976. “On Steiner minimal trees with rectilinear distance.” *SIAM Journal of Applied Mathematics,* Vol. 30, pp. 104–114. *See also:* Hwang, F. K., “An O(n log n) Algorithm for Suboptimal Rectilinear Steiner Trees,” *IEEE Transactions on Circuits and Systems,* Vol. CAS-26, no. 1, pp. 75–77, January 1979. Describes an algorithm to improve the rectilinear minimum spanning tree (RMST) approximation to the minimal rectilinear Steiner tree (minimal RST). The approximation is at most 1.5 times longer than the minimal RST, since the RMST is at worst 1.5 times the length of the minimal RST. [p. 899]

Kirkpatrick, S., C. D. Gerlatt Jr., and M. P. Vecchi. 1983. “Optimization by simulated annealing,” *Science,* Vol. 220, no. 4598, pp. 671–680. [p. 890]

Lengauer, T. 1990. *Combinatorial Algorithms for Integrated Circuit Layout.* Chichester, England: Wiley. ISBN 0-471-92838-0. TK7874.L36. Contains chapters on circuit layout; optimization problems; graph algorithms; operations research and statistics; combinatorial layout problems; circuit partitioning; placement; assignment; floorplanning; global routing and area routing; detailed routing; and compaction. 484 references. [p. 906]

Luk, W. K. 1991. "A fast physical constraint generator for timing driven layout." In *Proceedings of the 28th ACM/IEEE Design Automation Conference*. Introduction to timing-driven placement and net- and path-based approaches. Describes some different methods to estimate interconnect delay during placement. ISBN 0-89791-395-7. [p. 906].

Masleid, R. P. 1991. "High-density central I/O circuits for CMOS." *IEEE Journal of Solid-State Circuits,* Vol. 26, no. 3, pp. 431–435. An I/O circuit design that reduces the percentage of chip area occupied by I/O circuits from roughly 22 percent to under 3 percent for a 256 I/O chip. Uses IBM C4 technology that allows package connections to be located over chip circuitry. 10 references. [p. 866]

Ohtsuki, T. (Ed.). 1986. *Layout Design and Verification.* New York: Elsevier. Includes nine papers on CAD tools and algorithms: "Layout strategy, standardisation, and CAD tools," Ueda, Kasai and Sudo; "Layout compaction," Mylynski and Sung; "Layout verification," Yoshida; "Partitioning, assignment and placement," Goto and Matsuda; "Computational complexity of layout problems," Shing and Hu; "Computational and geometry algorithms," Asano, Sato and Ohtsuki; an excellent survey and tutorial paper by M. Burstein: "Channel routing"; "Maze-running and line-search algorithms" an easily-readable paper on detailed routing by Ohtsuki; and a mathematical paper, "Global routing," by Kuh and Marek-Sadowska. ISBN 0444878947. TK7874. L318. [p. 906]

Preas, B. T., and P. G. Karger. 1988. "Placement, assignment and floorplanning." In *Physical Design Automation of VLSI Systems,* B. T. Preas and M. J. Lorenzetti (Eds.), pp. 87–155. Menlo Park, CA: Benjamin-Cummings. ISBN 0-8053-0412-9. TK7874.P47. [p. 906]

Sangiovanni-Vincentelli, A. 1986. "Automatic layout of integrated circuits." In *Nato Advanced Study on "Logic Synthesis and Silicon Compilers for VLSI Design"*, G. De Micheli, A. Sangiovanni-Vincentelli, and A. Paolo (Eds.). Norwell, MA: Kluwer. ISBN 90-247-2689-1, 90-247-3561-0. TK7874.N338. [p. 906]

Schweikert, D. G., 1976. "A 2-dimensional placement algorithm for the layout of electrical circuits." In *Proceedings of the 9th Design Automation Conference*. Description of half-perimeter wire measure. [p. 879]

Sechen, C. 1988. *VLSI Placement and Global Routing Using Simulated Annealing*. Norwell, MA: Kluwer. Contains chapters on the simulated annealing algorithm; placement and global routing; floorplanning; average interconnection length estimation; interconnect-area estimation; a channel definition algorithm; and a global router algorithm. ISBN 0898382815. TK7874. S38. [p. 890]

Sigl, G., K. Doll, and F. M. Johannes. 1991. "Analytical placement: a linear or quadratic objective function?" In *Proceedings of the 28th ACM/IEEE Design Automation Conference*. Compares quadratic and linear cost function for placement algorithms. Explains the Gordian place-and-route system from the Technical University of Munich. ISBN 0-89791-395-7. [p. 903].

Wada, T., M. Eino, and K. Anami. 1990. "Simple noise model and low-noise data-output buffer for ultrahigh-speed memories." *IEEE Journal of Solid-State Circuits,* Vol. 25, no. 6, pp. 1586–1588. An analytic noise model for voltage bounce on internal VDD/VSS lines. [p. 866]

Wong, D. F., H. W. Leong, and C. L. Liu. 1988. *Simulated Annealing for VLSI Design*. Norwell, MA: Kluwer. Introduction; Placement; Floorplan Design; Channel Routing; Permutation Channel Routing; PLA Folding; Gate Matrix Layout; Array Optimization. ISBN 0898382564. TK7874. W65. [p. 890]

Youssef, H., R.-B. Lin, and E. Shragowitz. 1992. "Bounds on net delays for VLSI circuits." *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing,* Vol. 39, no. 11, pp. 315–324. An alternative to the weight-based approach is development of delay bounds on all nets. 21 references. [p. 891].

ROUTING 17

Once the designer has floorplanned a chip and the logic cells within the flexible blocks have been placed, it is time to make the connections by routing the chip. This is still a hard problem that is made easier by dividing it into smaller problems. Routing is usually split into **global routing** followed by **detailed routing**.

Suppose the ASIC is North America and some travelers in California need advice on how to drive from Stanford (near San Francisco) to Caltech (near Los Angeles). The floorplanner has decided that California is on the left (west) side of the ASIC and the placement tool has put Stanford in Northern California and Caltech in Southern California. Floorplanning and placement have defined the roads and freeways. There are two ways to go: the coastal route (using Highway 101) or the inland route (using Interstate I5, which is usually faster). The global router specifies the coastal route because the travelers are not in a hurry and I5 is congested (the global router knows this because it has already routed onto I5 many other travelers that are in a hurry today). Next, the detailed router looks at a map and gives indications from Stanford onto Highway 101 south through San Jose, Monterey, and Santa Barbara to Los Angeles and then off the freeway to Caltech in Pasadena.

Figure 17.1 shows the core of the Viterbi decoder after the placement step. This implementation consists entirely of standard cells (18 rows). The I/O pads are not included in this example—we can route the I/O pads after we route the core (though this is not always a good idea). Figure 17.2 shows the Viterbi decoder chip after global and detailed routing. The routing runs in the channels between the rows of logic cells, but the individual interconnections are too small to see.

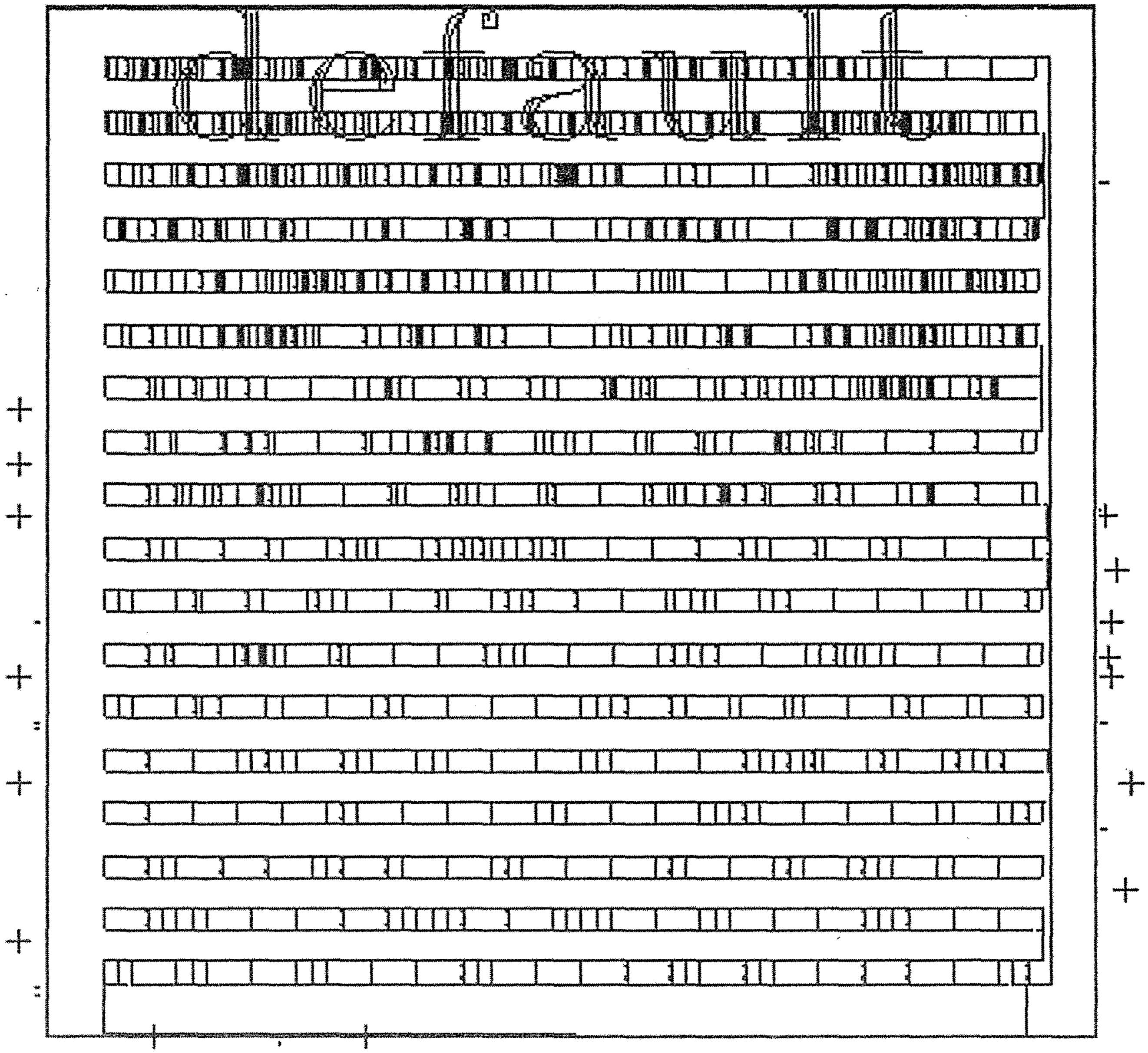


FIGURE 17.1 The core of the Viterbi decoder chip after placement (a screen shot from Cadence Cell Ensemble). This is the same placement as shown in Figure 16.2, but without the channel labels. You can see the rows of standard cells; the widest cells are the D flip-flops.

17.1 Global Routing

The details of global routing differ slightly between cell-based ASICs, gate arrays, and FPGAs, but the principles are the same in each case. A global router does not make any connections, it just plans them. We typically global route the whole chip

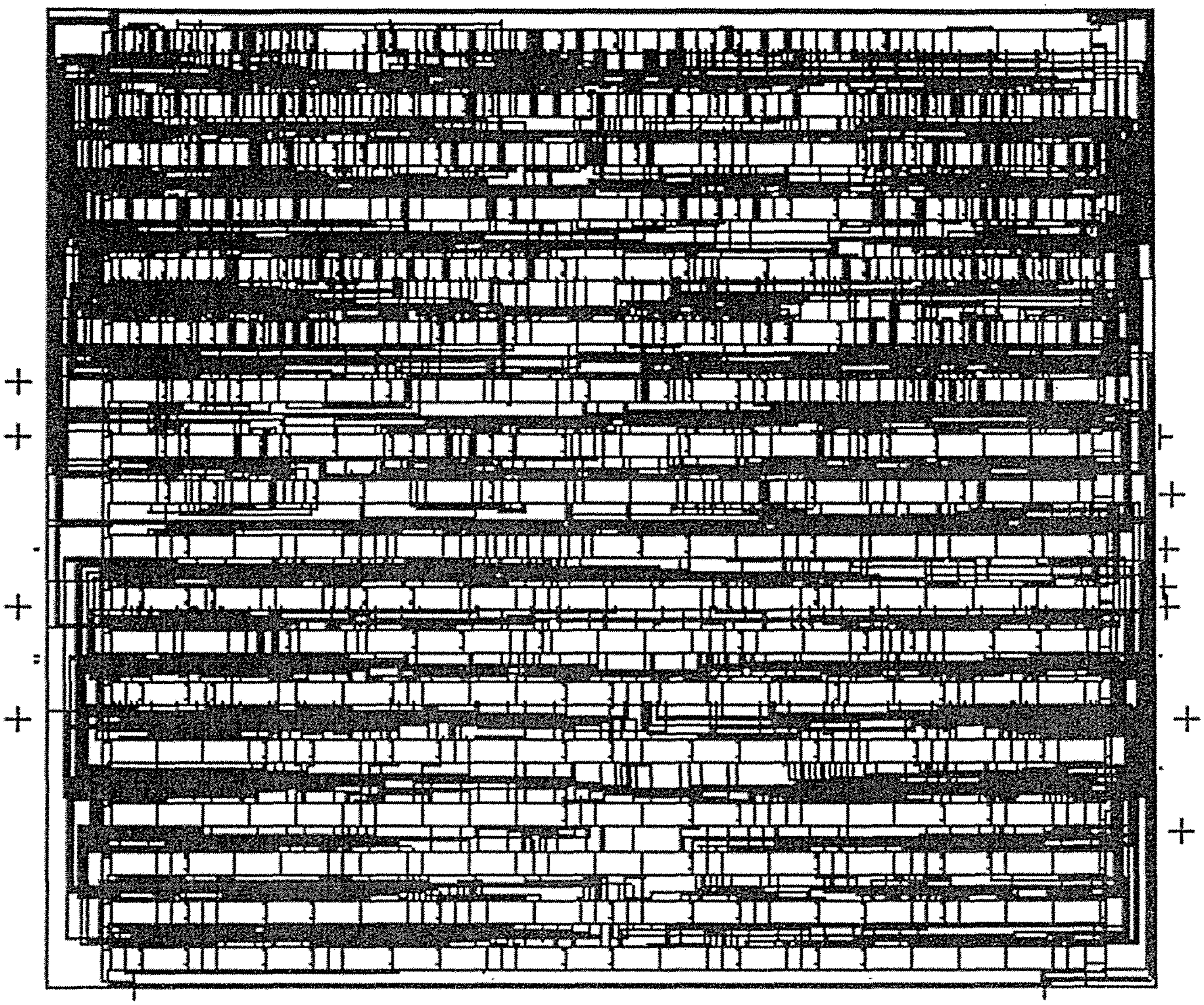


FIGURE 17.2 The core of the Viterbi decoder chip after the completion of global and detailed routing (a screen shot from Cadence Cell Ensemble). This chip uses two-level metal. Although you cannot see the difference, m1 runs in the horizontal direction and m2 in the vertical direction.

(or large pieces if it is a large chip) before detail routing the whole chip (or the pieces). There are two types of areas to global route: inside the flexible blocks and between blocks (the Viterbi decoder, although a cell-based ASIC, only involved the global routing of one large flexible block).

17.1.1 Goals and Objectives

The input to the global router is a floorplan that includes the locations of all the fixed and flexible blocks; the placement information for flexible blocks; and the locations of all the logic cells. The goal of global routing is to provide complete instructions to the detailed router on where to route every net. The objectives of global routing are one or more of the following:

- Minimize the total interconnect length.
- Maximize the probability that the detailed router can complete the routing.
- Minimize the critical path delay.

In both floorplanning and placement, with minimum interconnect length as an objective, it is necessary to find the shortest total path length connecting a set of *terminals*. This path is the MRST, which is hard to find. The alternative, for both floorplanning and placement, is to use simple approximations to the length of the MRST (usually the half-perimeter measure). Floorplanning and placement both assume that interconnect may be put anywhere on a rectangular grid, since at this point nets have not been assigned to the channels, but the global router must use the wiring channels and find the actual path. Often the global router needs to find a path that minimizes the delay between two terminals—this is not necessarily the same as finding the shortest total path length for a set of terminals.

17.1.2 Measurement of Interconnect Delay

Floorplanning and placement need a fast and easy way to estimate the interconnect delay in order to evaluate each trial placement; often this is a predefined look-up table. After placement, the logic cell positions are fixed and the global router can afford to use better estimates of the interconnect delay. To illustrate one method, we shall use the Elmore constant to estimate the interconnect delay for the circuit shown in Figure 17.3.

The problem is to find the voltages at the inputs to logic cells B and C taking into account the parasitic resistance and capacitance of the metal interconnect. Figure 17.3(c) models logic cell A as an ideal switch with a pull-down resistance equal to R_{pd} and models the metal interconnect using resistors and capacitors for each segment of the interconnect.

The Elmore constant for node 4 (labeled V_4) in the network shown in Figure 17.3(c) is

$$\tau_{D4} = \sum_{k=1}^{4} R_{k4}C_k = R_{14}C_1 + R_{24}C_2 + R_{34}C_3 + R_{44}C_4 , \qquad (17.1)$$

where,

$$\begin{aligned} R_{14} &= R_{pd} + R_1 & R_{24} &= R_{pd} + R_1 \\ R_{34} &= R_{pd} + R_1 + R_3 & R_{44} &= R_{pd} + R_1 + R_3 + R_4 \end{aligned} \qquad (17.2)$$

In Eq. 17.2 notice that $R_{24} = R_{pd} + R_1$ (and not $R_{pd} + R_1 + R_2$) because R_1 is the resistance to V_0 (ground) shared by node 2 and node 4.

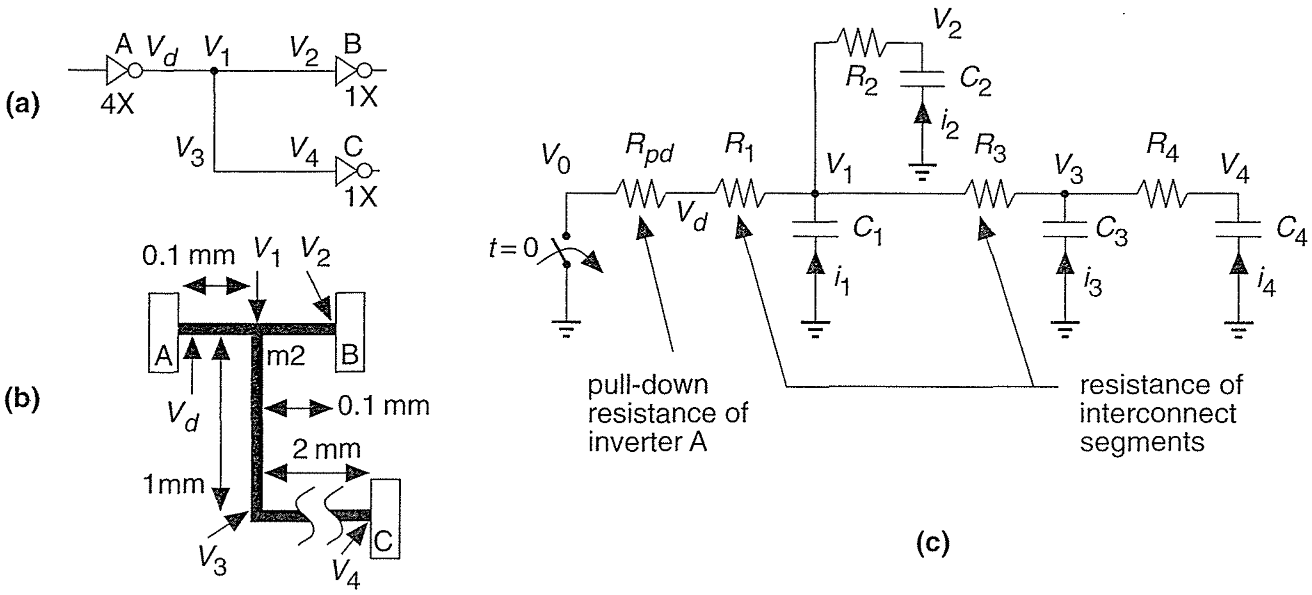


FIGURE 17.3 Measuring the delay of a net. (a) A simple circuit with an inverter A driving a net with a fanout of two. Voltages V_1, V_2, V_3, and V_4 are the voltages at intermediate points along the net. (b) The layout showing the net segments (pieces of interconnect). (c) The RC model with each segment replaced by a capacitance and resistance. The ideal switch and pull-down resistance R_{pd} model the inverter A.

Suppose we have the following parameters (from the generic 0.5 µm CMOS process, G5) for the layout shown in Figure 17.3(b):

- m2 resistance is 50 mΩ/square.
- m2 capacitance (for a minimum-width line) is 0.2 pFmm^{-1}.
- 4X inverter delay is 0.02 ns + 0.5C_L ns (C_L is in picofarads).
- Delay is measured using 0.35/0.65 output trip points.
- m2 minimum width is 3 λ = 0.9 µm.
- 1X inverter input capacitance is 0.02 pF (a standard load).

First we need to find the pull-down resistance, R_{pd}, of the 4X inverter. If we model the gate with a linear pull-down resistor, R_{pd}, driving a load C_L, the output waveform is exp $-t/(C_L R_{pd})$ (normalized to 1V). The output reaches 63 percent of its final value when $t = C_L R_{pd}$, because exp(–1) = 0.63. Then, because the delay is measured with a 0.65 trip point, the constant 0.5 nspF^{-1} = 0.5 kΩ is very close to the equivalent pull-down resistance. Thus, $R_{pd} \approx 500\,\Omega$.

From the given data, we can calculate the *R*'s and *C*'s:

$$R_1 = R_2 = \frac{(0.1\ \text{mm})\ (50 \times 10^{-3}\ \Omega)}{0.9\ \mu\text{m}} = 6\ \Omega$$
$$R_3 = \frac{(1\ \text{mm})\ (50 \times 10^{-3}\ \Omega)}{0.9\ \mu\text{m}} = 56\ \Omega \tag{17.3}$$
$$R_4 = \frac{(2\ \text{mm})\ (50 \times 10^{-3}\ \Omega)}{0.9\ \mu\text{m}} = 112\ \Omega$$

$$\begin{aligned} C_1 &= (0.1\text{mm})\ (0.2\ \text{pFmm}^{-1}) = 0.02\ \text{pF} \\ C_2 &= (0.1\ \text{mm})\ (0.2\ \text{pFmm}^{-1}) + 0.02\ \text{pF} = 0.04\ \text{pF} \\ C_3 &= (1\ \text{mm})\ (0.2\ \text{pFmm}^{-1}) = 0.2\ \text{pF} \\ C_4 &= (2\ \text{mm})\ (0.2\ \text{pFmm}^{-1}) + 0.02\ \text{pF} = 0.42\ \text{pF} \end{aligned} \tag{17.4}$$

Now we can calculate the path resistance, R_{ki}, values (notice that $R_{ki} = R_{ik}$):

$$\begin{aligned} R_{14} &= 500\ \Omega + 6\ \Omega = 506\ \Omega \\ R_{24} &= 500\ \Omega + 6\ \Omega = 506\ \Omega \\ R_{34} &= 500\ \Omega + 6\ \Omega + 56\ \Omega = 562\ \Omega \\ R_{44} &= 500\ \Omega + 6\ \Omega + 56\ \Omega + 112\ \Omega = 674\ \Omega \end{aligned} \tag{17.5}$$

Finally, we can calculate Elmore's constants for node 4 and node 2 as follows:

$$\begin{aligned} \tau_{D4} &= R_{14}C_1 + R_{24}C_2 + R_{34}C_3 + R_{44}C_4 \\ &= (506)\ (0.02) + (506)\ (0.04) + (562)\ (0.2) + (674)\ (0.42) \\ &= 425\ \text{ps}, \end{aligned} \tag{17.6}$$

$$\begin{aligned} \tau_{D2} &= R_{12}C_1 + R_{22}C_2 + R_{32}C_3 + R_{42}C_4 \\ &= (R_{pd} + R_1 + R_2)\ C_2 + (R_{pd} + R_1)\ (C_1 + C_3 + C_4) \\ &= (500 + 6 + 6)\ (0.04) + (500 + 6)\ (0.02 + 0.2 + 0.42) \\ &= 344\ \text{ps}, \end{aligned} \tag{17.7}$$

and $\tau_{D4} - \tau_{D2} = (425 - 344) = 81$ ps.

A **lumped-delay model** neglects the effects of interconnect resistance and simply sums all the node capacitances (the **lumped capacitance**) as follows:

$$\begin{aligned} t_D &= R_{pd}(C_1 + C_2 + C_3 + C_4) \\ &= (500)\ (0.02 + 0.04 + 0.2 + 0.42) \\ &= 340 \text{ ps}. \end{aligned} \tag{17.8}$$

Comparing Eqs. 17.6–17.8, we can see that the delay of the inverter can be assigned as follows: 20 ps (the intrinsic delay, 0.2 ns, due to the cell output capacitance), 340 ps (due to the pull-down resistance and the output capacitance), 4 ps (due to the interconnect from A to B), and 65 ps (due to the interconnect from A to C). We can see that the error from neglecting interconnect resistance can be important.

Even using the Elmore constant we still made the following assumptions in estimating the path delays:

- A step-function waveform drives the net.
- The delay is measured from when the gate input changes.
- The delay is equal to the time constant of an exponential waveform that approximates the actual output waveform.
- The interconnect is modeled by discrete resistance and capacitance elements.

The global router could use more sophisticated estimates that remove some of these assumptions, but there is a limit to the accuracy with which delay can be estimated during global routing. For example, the global router does not know how much of the routing is on which of the layers, or how many vias will be used and of which type, or how wide the metal lines will be. It may be possible to estimate how much interconnect will be horizontal and how much is vertical. Unfortunately, this knowledge does not help much if horizontal interconnect may be completed in either m1 or m3 and there is a large difference in parasitic capacitance between m1 and m3, for example.

When the global router attempts to minimize interconnect delay, there is an important difference between a path and a net. The path that minimizes the delay between two terminals on a net is not necessarily the same as the path that minimizes the total path length of the net. For example, to minimize the path delay (using the Elmore constant as a measure) from the output of inverter A in Figure 17.3(a) to the input of inverter B requires a rather complicated algorithm to construct the best path. We shall return to this problem in Section 17.1.6.

17.1.3 Global Routing Methods

Global routing cannot use the interconnect-length approximations, such as the half-perimeter measure, that were used in placement. What is needed now is the actual path and not an approximation to the path length. However, many of the methods used in global routing are still based on the solutions to the tree on a graph problem.

One approach to global routing takes each net in turn and calculates the shortest path using tree on graph algorithms—with the added restriction of using the available channels. This process is known as **sequential routing**. As a sequential routing algorithm proceeds, some channels will become more congested since they hold more interconnects than others. In the case of FPGAs and channeled gate arrays, the channels have a fixed channel capacity and can only hold a certain number of interconnects. There are two different ways that a global router normally handles this problem. Using **order-independent routing**, a global router proceeds by routing each net, ignoring how crowded the channels are. Whether a particular net is processed first or last does not matter, the channel assignment will be the same. In order-independent routing, after all the interconnects are assigned to channels, the global router returns to those channels that are the most crowded and reassigns some interconnects to other, less crowded, channels. Alternatively, a global router can consider the number of interconnects already placed in various channels as it proceeds. In this case the global routing is **order dependent**—the routing is still sequential, but now the order of processing the nets will affect the results. Iterative improvement or simulated annealing may be applied to the solutions found from both order-dependent and order-independent algorithms. This is implemented in the same way as for system partitioning and placement: A constructed solution is successively changed, one interconnect path at a time, in a series of random moves.

In contrast to sequential global-routing methods, which handle nets one at a time, **hierarchical routing** handles all nets at a particular level at once. Rather than handling all of the nets on the chip at the same time, the global-routing problem is made more tractable by dividing the chip area into levels of hierarchy. By considering only one level of hierarchy at a time the size of the problem is reduced at each level. There are two ways to traverse the levels of hierarchy. Starting at the whole chip, or highest level, and proceeding down to the logic cells is the top-down approach. The bottom-up approach starts at the lowest level of hierarchy and globally routes the smallest areas first.

17.1.4 Global Routing Between Blocks

Figure 17.4 illustrates the global-routing problem for a cell-based ASIC. Each edge in the **channel-intersection graph** in Figure 17.4(c) represents a channel. The global router is restricted to using these channels. The weight of each edge in the graph corresponds to the length of the channel. The global router plans a path for each interconnect using this graph.

Figure 17.5 shows an example of global routing for a net with five terminals, labeled A1 through F1, for the cell-based ASIC shown in Figure 17.4. If a designer wishes to use minimum total interconnect path length as an objective, the global router finds the minimum-length tree shown in Figure 17.5(b). This tree determines the channels the interconnects will use. For example, the shortest connection from A1 to B1 uses channels 2, 1, and 5 (in that order). This is the information the global router passes to the detailed router. Figure 17.5(c) shows that minimizing the total path length may not correspond to minimizing the path delay between two points.

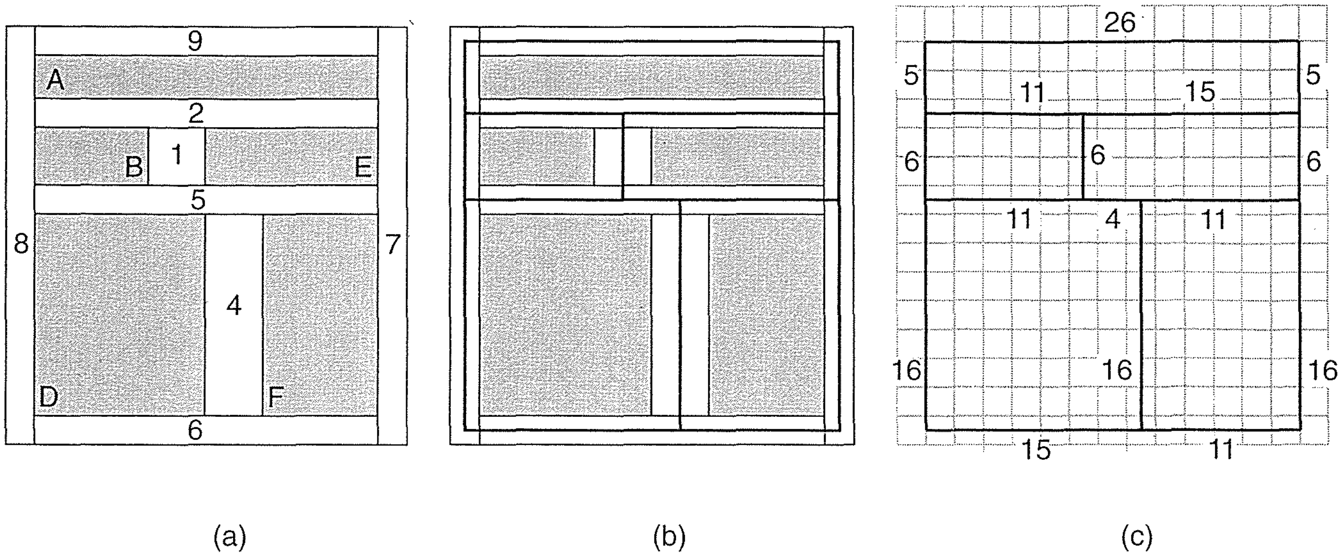


FIGURE 17.4 Global routing for a cell-based ASIC formulated as a graph problem. (a) A cell-based ASIC with numbered channels. (b) The channels form the edges of a graph. (c) The channel-intersection graph. Each channel corresponds to an edge on a graph whose weight corresponds to the channel length.

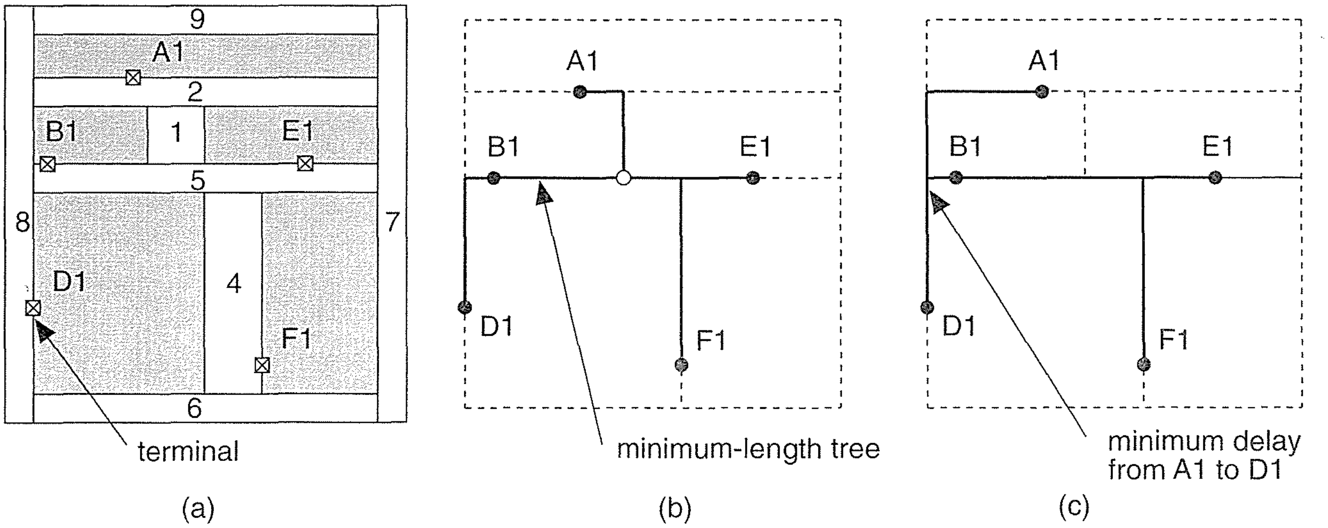


FIGURE 17.5 Finding paths in global routing. (a) A cell-based ASIC (from Figure 17.4) showing a single net with a fanout of four (five terminals). We have to order the numbered channels to complete the interconnect path for terminals A1 through F1. (b) The terminals are projected to the center of the nearest channel, forming a graph. A minimum-length tree for the net that uses the channels and takes into account the channel capacities. (c) The minimum-length tree does not necessarily correspond to minimum delay. If we wish to minimize the delay from terminal A1 to D1, a different tree might be better.

Global routing is very similar for cell-based ASICs and gate arrays, but there is a very important difference between the types of channels in these ASICs. The size of the channels in sea-of-gates arrays, channelless gate arrays, and cell-based ASICs can be varied to make sure there is enough space to complete the wiring. In channeled gate-arrays and FPGAs the size, number, and location of channels are fixed. The good news is that the global router can allocate as many interconnects to each channel as it likes, since that space is committed anyway. The bad news is that there is a maximum number of interconnects that each channel can hold. If the global router needs more room, even in just one channel on the whole chip, the designer has to repeat the placement-and-routing steps and try again (or use a bigger chip).

17.1.5 Global Routing Inside Flexible Blocks

We shall illustrate global routing using a gate array. Figure 17.6(a) shows the routing resources on a sea-of-gates or channelless gate array. The gate array base cells are arranged in 36 blocks, each block containing an array of 8-by-16 gate-array base cells, making a total of 4068 base cells.

The horizontal interconnect resources are the routing channels that are formed from unused rows of the gate-array base cells, as shown in Figure 17.6(b) and (c). The vertical resources are feedthroughs. For example, the logic cell shown in Figure 17.6(d) is an inverter that contains two types of feedthrough. The inverter logic cell uses a single gate-array base cell with terminals (or *connectors*) located at the top and bottom of the logic cell. The inverter input pin has two electrically equivalent terminals that the global router can use as a feedthrough. The output of the inverter is connected to only one terminal. The remaining vertical **track** is unused by the inverter logic cell, so this track forms an uncommitted feedthrough.

You may see any of the terms **landing pad** (because we say that we "drop" a via to a landing pad), **pick-up point**, **connector**, **terminal**, **pin**, or **port** used for the connection to a logic cell. The term *pick-up point* refers to the physical pieces of metal (or sometimes polysilicon) in the logic cell to which the router connects. In a three-level metal process, the global router may be able to connect to anywhere in an area—an **area pick-up point**. In this book we use the term *connector* to refer to the physical pick-up point. The term *pin* more often refers to the connection on a logic schematic icon (a dot, square box, or whatever symbol is used), rather than layout. Thus the difference between a pin and a connector is that we can have multiple connectors for one pin. *Terminal* is often used when we talk about routing. The term *port* is used when we are using text (EDIF netlists or HDLs, for example) to describe circuits.

In a gate array the channel capacity must be a multiple of the number of **horizontal tracks** in the gate-array base cell. Figure 17.6(e) shows a gate-array base cell with seven horizontal tracks (see Section 17.2 for the factors that determine the track width and track spacing). Thus, in this gate array, we can have a channel with a capacity of 7, 14, 21, ... horizontal tracks—but not between these values.

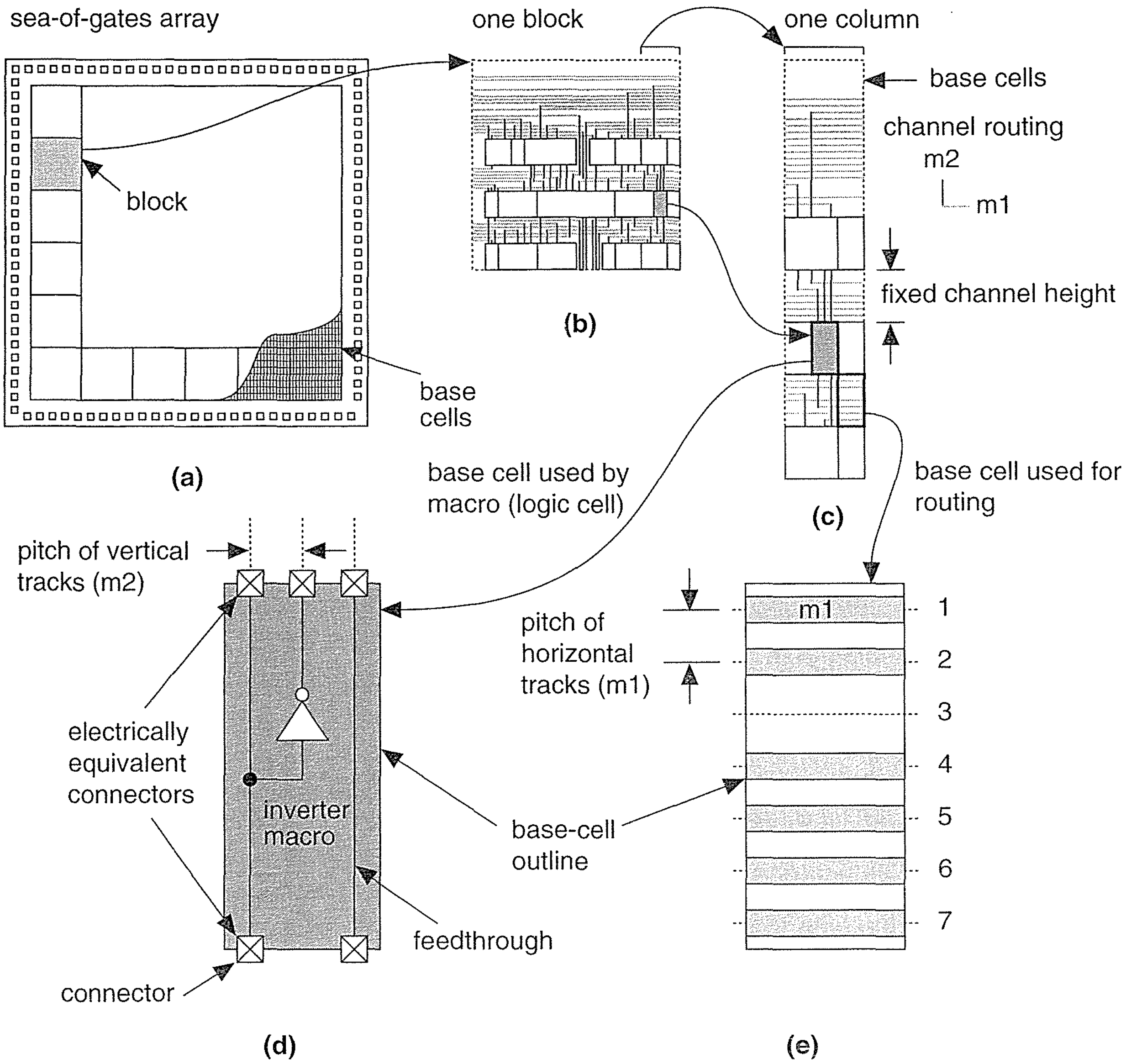


FIGURE 17.6 Gate-array global routing. (a) A small gate array. (b) An enlarged view of the routing. The top channel uses three rows of gate-array base cells; the other channels use only one. (c) A further enlarged view showing how the routing in the channels connects to the logic cells. (d) One of the logic cells, an inverter. (e) There are seven horizontal wiring tracks available in one row of gate-array base cells—the channel capacity is thus 7.

Figure 17.7 shows the inverter macro for the sea-of-gates array shown in Figure 17.6. Figure 17.7(a) shows the base cell. Figure 17.7(b) shows how the internal inverter wiring on m1 leaves one vertical track free as a feedthrough in a two-level metal process (connectors placed at the top and bottom of the cell). In a three-level metal process the connectors may be placed inside the cell abutment box (Figure 17.7c). Figure 17.8 shows the global routing for the sea-of-gates array. We

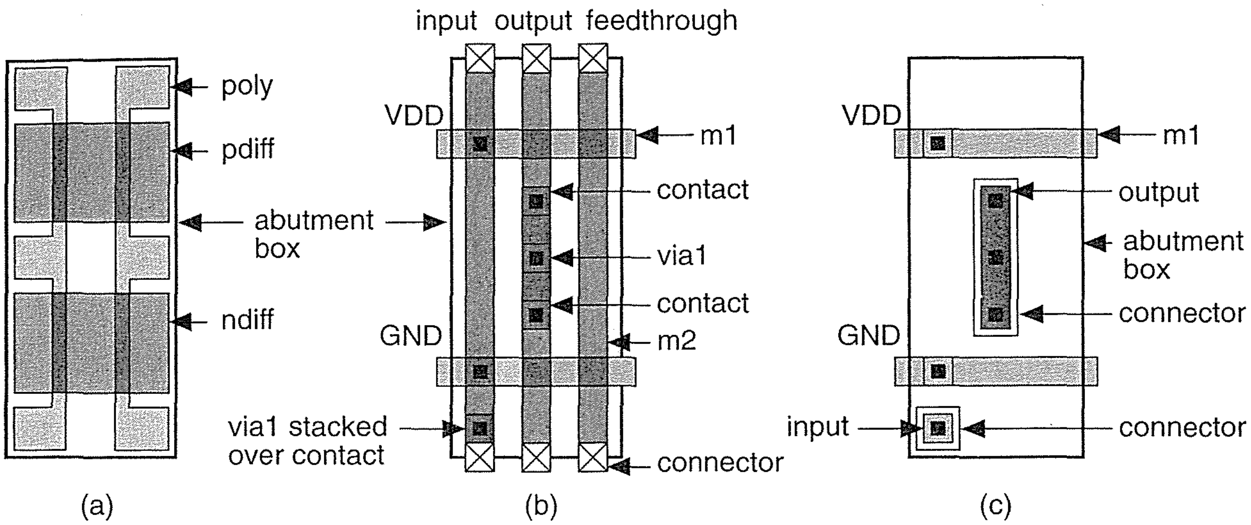


FIGURE 17.7 The gate-array inverter from Figure 17.6d. (a) An oxide-isolated gate-array base cell, showing the diffusion and polysilicon layers. (b) The metal and contact layers for the inverter in a 2LM (two-level metal) process. (c) The router's view of the cell in a 3LM process.

divide the array into nonoverlapping **routing bins** (or just **bins**, also called **global routing cells** or **GRCs**), each containing a number of gate-array base cells.

We need an aside to discuss our use of the term *cell*. Be careful not to confuse the global routing cells with gate-array base cells (the smallest element of a gate array, consisting of a small number of *n*-type and *p*-type transistors), or with logic cells (which are NAND gates, NOR gates, and so on).

A large routing bin reduces the size of the routing problem, and a small routing bin allows the router to calculate the wiring capacities more accurately. Some tools permit routing bins of different size in different areas of the chip (with smaller routing bins helping in areas of dense routing). Figure 17.8(a) shows a routing bin that is 2-by-4 gate-array base cells. The logic cells occupy the lower half of the routing bin. The upper half of the routing bin is the channel area, reserved for wiring. The global router calculates the edge capacities for this routing bin, including the vertical feedthroughs. The global router then determines the shortest path for each net considering these edge capacities. An example of a global-routing calculation is shown in Figure 17.8(b). The path, described by a series of adjacent routing bins, is passed to the detailed router.

17.1.6 Timing-Driven Methods

Minimizing the total pathlength using a Steiner tree does not necessarily minimize the interconnect delay of a path. Alternative tree algorithms apply in this situation, most using the Elmore constant as a method to estimate the delay of a path (Section 17.1.2). As in timing-driven placement, there are two main approaches to

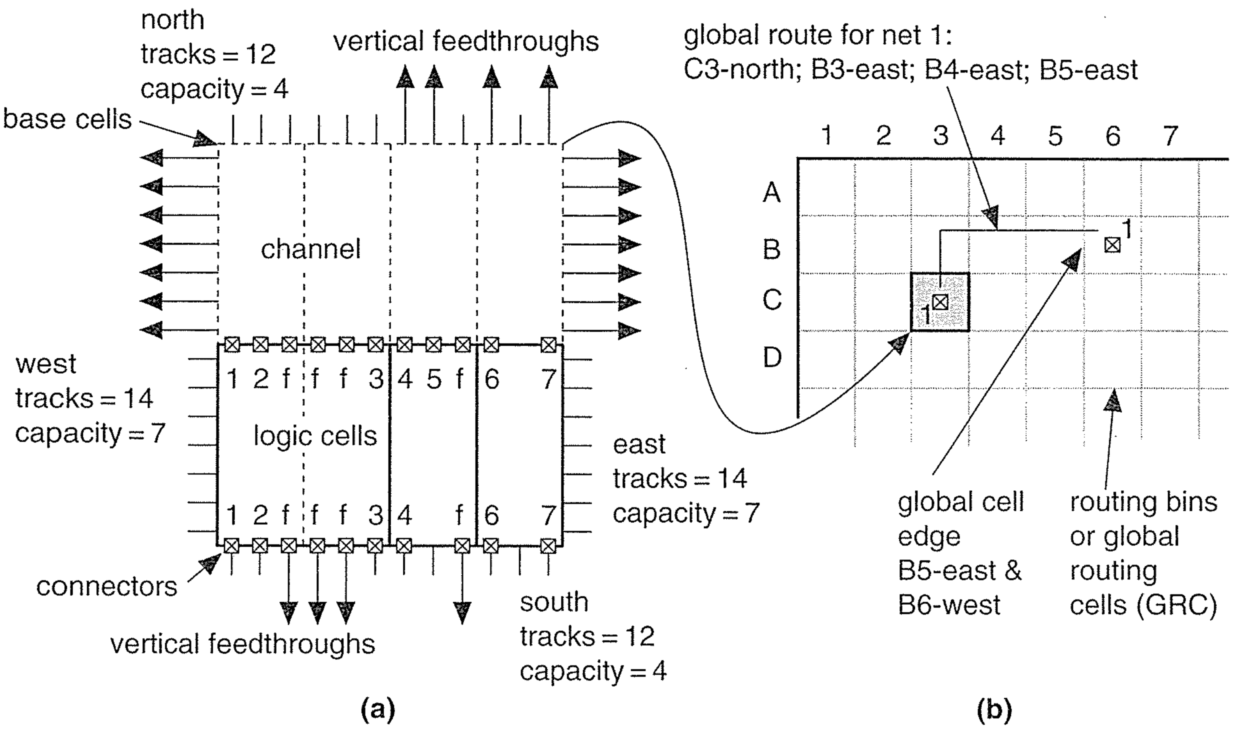


FIGURE 17.8 Global routing a gate array. (a) A single global-routing cell (GRC or routing bin) containing 2-by-4 gate-array base cells. For this choice of routing bin the maximum horizontal track capacity is 14, the maximum vertical track capacity is 12. The routing bin labeled C3 contains three logic cells, two of which have feedthroughs marked 'f'. This results in the edge capacities shown. (b) A view of the top left-hand corner of the gate array showing 28 routing bins. The global router uses the edge capacities to find a sequence of routing bins to connect the nets.

timing-driven routing: net-based and path-based. Path-based methods are more sophisticated. For example, if there is a critical path from logic cell A to B to C, the global router may increase the delay due to the interconnect between logic cells A and B if it can reduce the delay between logic cells B and C. Placement and global routing tools may or may not use the same algorithm to estimate net delay. If these tools are from different companies, the algorithms are probably different. The algorithms must be compatible, however. There is no use performing placement to minimize predicted delay if the global router uses completely different measurement methods. Companies that produce floorplanning and placement tools make sure that the output is compatible with different routing tools—often to the extent of using different algorithms to target different routers.

17.1.7 Back-annotation

After global routing is complete it is possible to accurately predict what the length of each interconnect in every net will be after detailed routing, probably to within 5 percent. The global router can give us not just an estimate of the total net length (which was all we knew at the placement stage), but the resistance and capacitance of each path in each net. This **RC information** is used to calculate net delays. We can back-annotate this net delay information to the synthesis tool for in-place optimization or to a timing verifier to make sure there are no timing surprises. Differences in timing predictions at this point arise due to the different ways in which the placement algorithms estimate the paths and the way the global router actually builds the paths.

17.2 Detailed Routing

The global routing step determines the channels to be used for each interconnect. Using this information the detailed router decides the exact location and layers for each interconnect. Figure 17.9(a) shows typical metal rules. These rules determine the m1 **routing pitch** (**track pitch**, **track spacing**, or just **pitch**). We can set the m1 pitch to one of three values:

1. **via-to-via** (**VTV**) pitch (or spacing),
2. **via-to-line** (**VTL** or **line-to-via**) pitch, or
3. **line-to-line** (**LTL**) pitch.

The same choices apply to the m2 and other metal layers if they are present. Via-to-via spacing allows the router to place vias adjacent to each other. Via-to-line spacing is hard to use in practice because it restricts the router to nonadjacent vias. Using line-to-line spacing prevents the router from placing a via at all without using jogs and is rarely used. Via-to-via spacing is the easiest for a router to use and the most common. Using either via-to-line or via-to-via spacing means that the routing pitch is larger than the minimum metal pitch.

Sometimes people draw a distinction between a cut and a via when they talk about large connections such as shown in Figure 17.10(a). We split or **stitch** a large via into identically sized cuts (sometimes called a **waffle via**). Because of the profile of the metal in a contact and the way current flows into a contact, often the total resistance of several small cuts is less than that of one large cut. Using identically sized cuts also means the processing conditions during contact etching, which may vary with the area and perimeter of a contact, are the same for every cut on the chip.

In a **stacked via** the contact cuts all overlap in a layout plot and it is impossible to tell just how many vias on which layers are present. Figure 17.10(b–f) show an alternative way to draw contacts and vias. Though this is not a standard, using the diagonal box convention makes it possible to recognize stacked vias and contacts on a layout (in any orientation). I shall use these conventions when it is necessary.

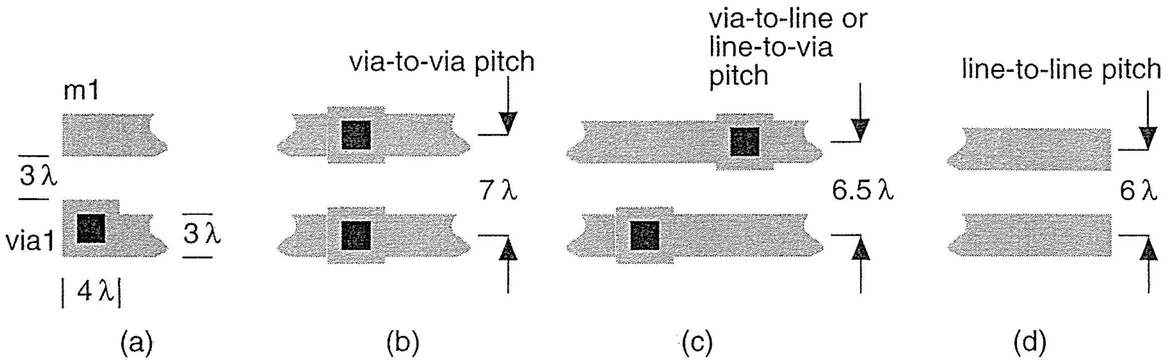


FIGURE 17.9 The metal routing pitch. (a) An example of λ-based metal design rules for m1 and via1 (m1/m2 via). (b) Via-to-via pitch for adjacent vias. (c) Via-to-line (or line-to-via) pitch for nonadjacent vias. (d) Line-to-line pitch with no vias.

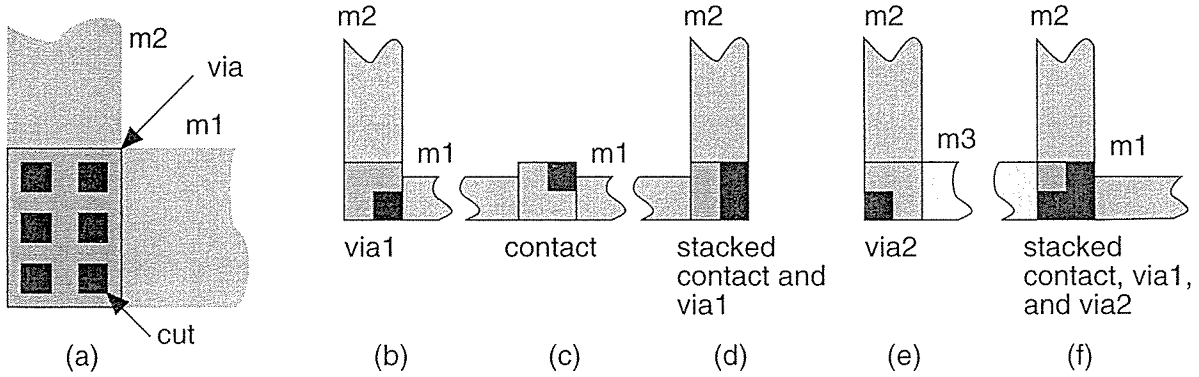


FIGURE 17.10 (a) A large m1 to m2 via. The black squares represent the holes (or cuts) that are etched in the insulating material between the m1 and 2 layers. (b) A m1 to m2 via (a via1). (c) A contact from m1 to diffusion or polysilicon (a contact). (d) A via1 placed over (or stacked over) a contact. (e) A m2 to m3 via (a via2) (f) A via2 stacked over a via1 stacked over a contact. Notice that the black square in parts b–c do *not* represent the actual location of the cuts. The black squares are offset so you can recognize stacked vias and contacts.

In a two-level metal CMOS ASIC technology we complete the wiring using the two different metal layers for the horizontal and vertical directions, one layer for each direction. This is **Manhattan routing**, because the results look similar to the rectangular north–south and east–west layout of streets in New York City. Thus, for example, if terminals are on the m2 layer, then we route the horizontal branches in a channel using m2 and the vertical trunks using m1. Figure 17.11 shows that, although we may choose a **preferred direction** for each metal layer (for example, m1 for horizontal routing and m2 for vertical routing), this may lead to problems in cases that have both horizontal and vertical channels. In these cases we define a **preferred metal layer** in the direction of the channel spine. In Figure 17.11, because the logic cell connectors are on m2, any vertical channel has to use vias at every logic cell location. By changing the orientation of the metal directions in vertical channels, we can avoid this, and instead we only need to place vias at the intersection of horizontal and vertical channels.

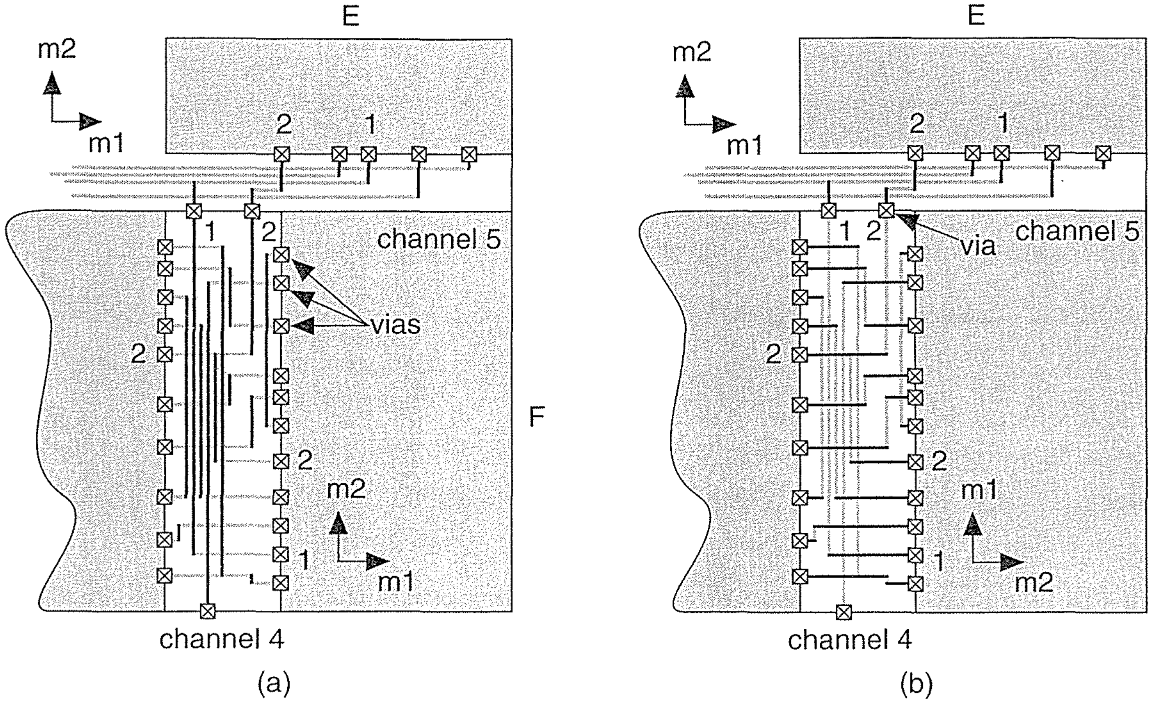


FIGURE 17.11 An expanded view of part of a cell-based ASIC. (a) Both channel 4 and channel 5 use m1 in the horizontal direction and m2 in the vertical direction. If the logic cell connectors are on m2 this requires vias to be placed at every logic cell connector in channel 4. (b) Channel 4 and 5 are routed with m1 along the direction of the channel spine (the long direction of the channel). Now vias are required only for nets 1 and 2, at the intersection of the channels.

Figure 17.12 shows an imaginary logic cell with connectors. Double-entry logic cells intended for two-level metal routing have connectors at the top and bottom of the logic cell, usually in m2. Logic cells intended for processes with three or more levels of metal have connectors in the center of the cell, again usually on m2. Logic cells may use both m1 and m2 internally, but the use of m2 is usually minimized. The router normally uses a simplified view of the logic cell called a **phantom**. The phantom contains only the logic cell information that the router needs: the connector locations, types, and names; the abutment and bounding boxes; enough layer information to be able to place cells without violating design rules; and a **blockage map**—the locations of any metal inside the cell that blocks routing.

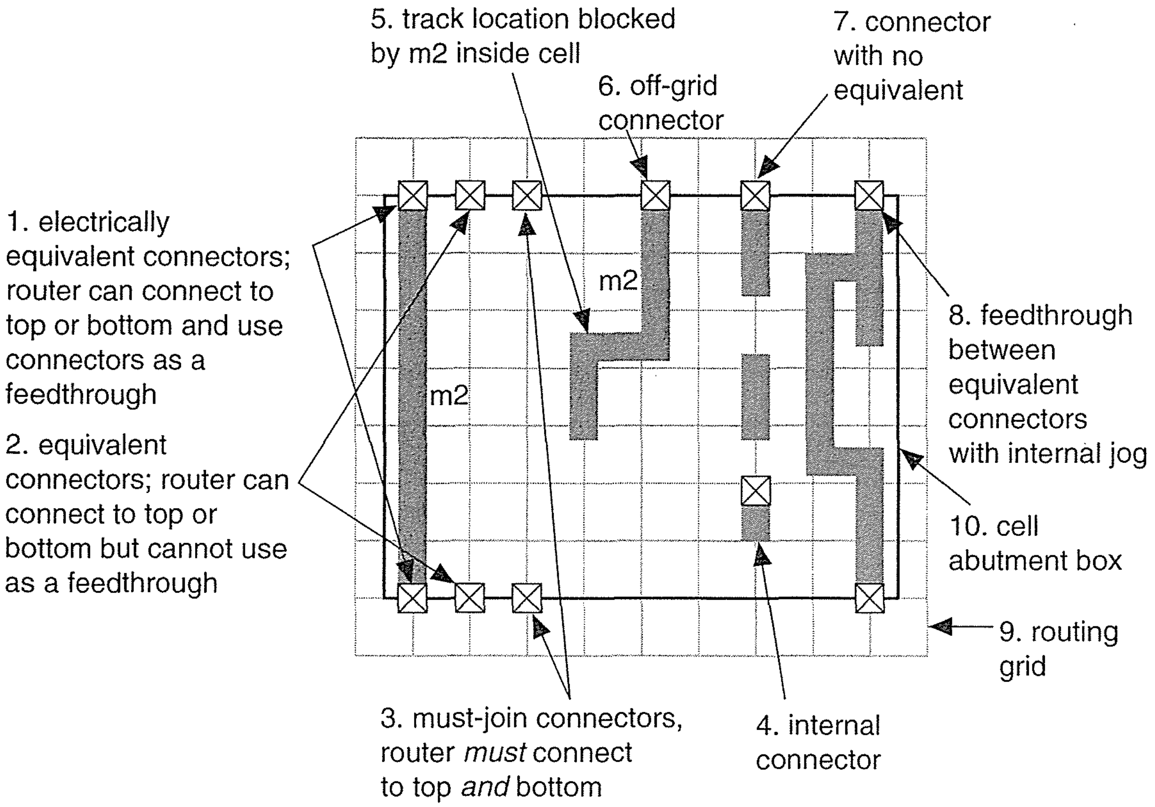

5. track location blocked by m2 inside cell
6. off-grid connector
7. connector with no equivalent
1. electrically equivalent connectors; router can connect to top or bottom and use connectors as a feedthrough
2. equivalent connectors; router can connect to top or bottom but cannot use as a feedthrough
m2
m2
8. feedthrough between equivalent connectors with internal jog
10. cell abutment box
9. routing grid
3. must-join connectors, router *must* connect to top *and* bottom
4. internal connector

FIGURE 17.12 The different types of connections that can be made to a cell. This cell has connectors at the top and bottom of the cell (normal for cells intended for use with a two-level metal process) and internal connectors (normal for logic cells intended for use with a three-level metal process). The interconnect and connections are drawn to scale.

Figure 17.13 illustrates some terms used in the detailed routing of a channel. The channel spine in Figure 17.13 is horizontal with terminals at the top and the bottom, but a channel can also be vertical. In either case terminals are spaced along the longest edges of the channel at given, fixed locations. Terminals are usually located on a grid defined by the routing pitch on that layer (we say terminals are either **on-grid** or **off-grid**). We make connections between terminals using interconnects that consist of one or more **trunks** running parallel to the length of the channel and **branches** that connect the trunk to the terminals. If more than one trunk is used, the trunks are connected by **doglegs**. Connections exit the channel at **pseudoterminals.**

The trunk and branch connections run in **tracks** (equispaced, like railway tracks). If the trunk connections use m1, the **horizontal track spacing** (usually just called the **track spacing** for channel routing) is equal to the m1 routing pitch. The maximum number of interconnects we need in a channel multiplied by the horizontal track spacing gives the minimum height of a channel (see Section 17.2.2 on how to determine

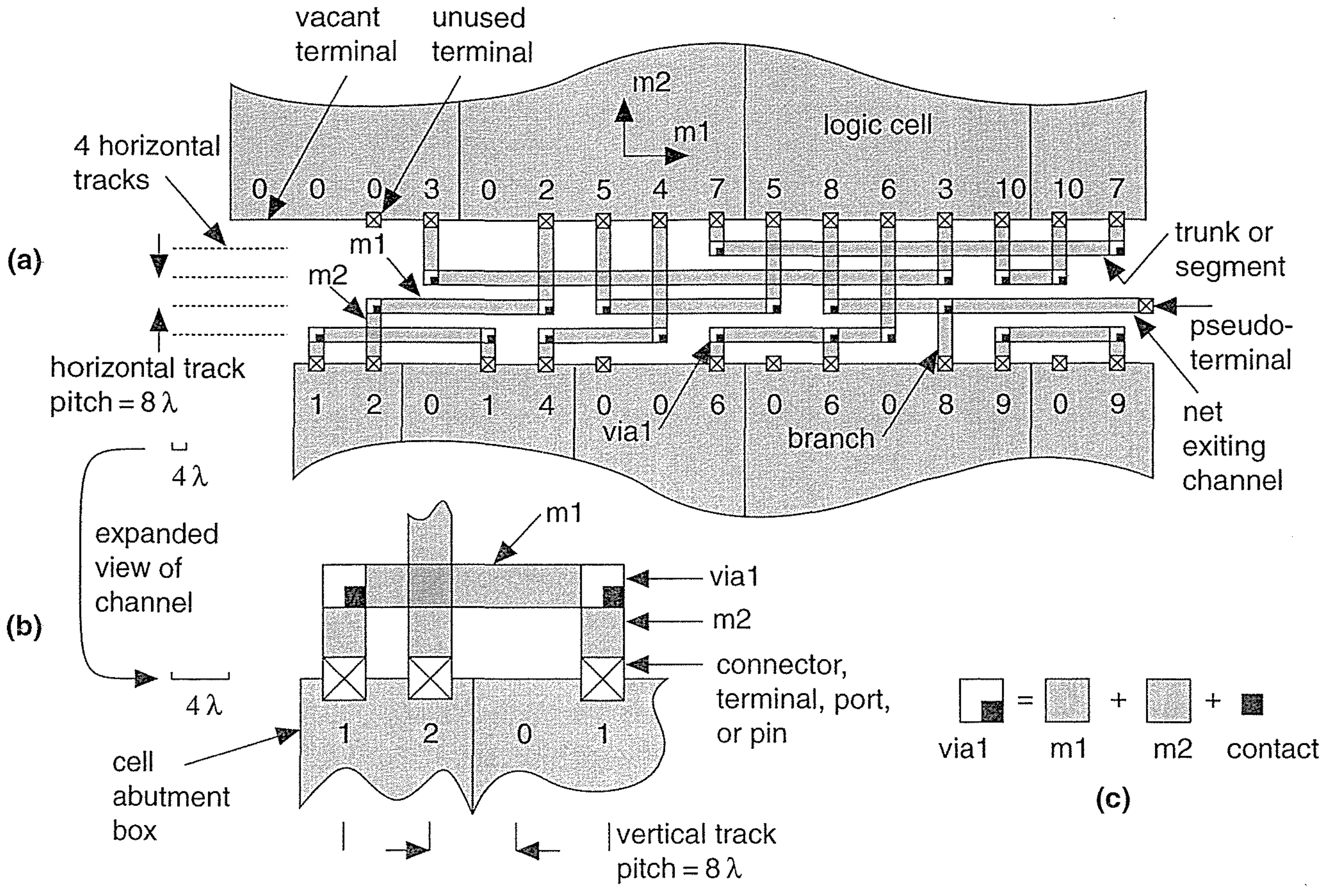


FIGURE 17.13 Terms used in channel routing. (a) A channel with four horizontal tracks. (b) An expanded view of the left-hand portion of the channel showing (approximately to scale) how the m1 and m2 layers connect to the logic cells on either side of the channel. (c) The construction of a via1 (m1/m2 via).

the maximum number of interconnects needed). Each terminal occupies a **column**. If the branches use m2, the **column spacing** (or **vertical track spacing**) is equal to the m2 routing pitch.

17.2.1 Goals and Objectives

The goal of detailed routing is to complete all the connections between logic cells. The most common objective is to minimize one or more of the following:

- The total interconnect length and area
- The number of layer changes that the connections have to make
- The delay of critical paths

Minimizing the number of layer changes corresponds to minimizing the number of vias that add parasitic resistance and capacitance to a connection.

In some cases the detailed router may not be able to complete the routing in the area provided. In the case of a cell-based ASIC or sea-of-gates array, it is possible to increase the channel size and try the routing steps again. A channeled gate array or FPGA has fixed routing resources and in these cases we must start all over again with floorplanning and placement, or use a larger chip.

17.2.2 Measurement of Channel Density

We can describe a channel-routing problem by specifying two lists of nets: one for the top edge of the channel and one for the bottom edge. The position of the net number in the list gives the column position. The net number zero represents a vacant or unused terminal. Figure 17.14 shows a channel with the numbered terminals to be connected along the top and the bottom of the channel.

We call the number of nets that cross a line drawn vertically anywhere in a channel the **local density**. We call the maximum local density of the channel the **global density** or sometimes just **channel density**. Figure 17.14 has a channel density of 4. Channel density is an important measure in routing—it tells a router the absolute fewest number of horizontal interconnects that it needs at the point where the local density is highest. In two-level routing (all the horizontal interconnects run on one routing layer) the channel density determines the minimum height of the channel. The channel capacity is the maximum number of interconnects that a channel can hold. If the channel density is greater than the channel capacity, that channel definitely cannot be routed (to learn how channel density is calculated, see Section 17.2.5).

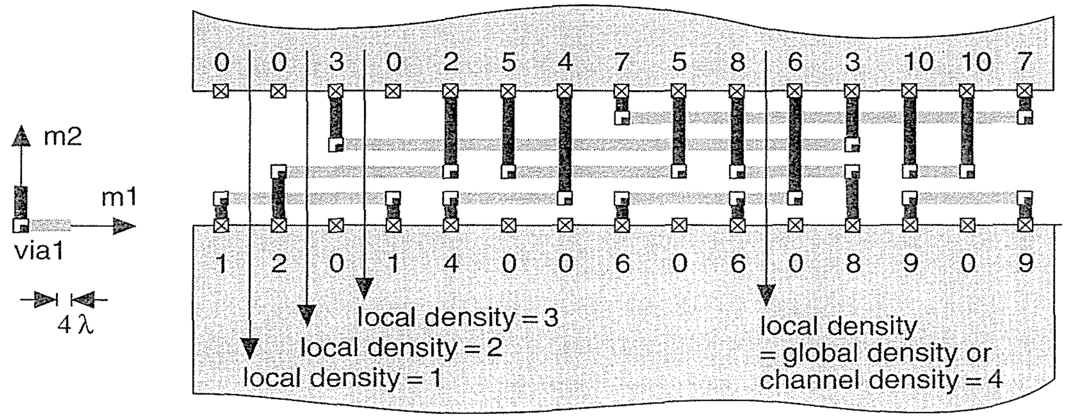


FIGURE 17.14 The definitions of local channel density and global channel density. Lines represent the m1 and m2 interconnect in the channel to simplify the drawing.

17.2.3 Algorithms

We start discussion of routing methods by simplifying the general channel-routing problem. The **restricted channel-routing problem** limits each net in a channel to use only one horizontal segment. In other words the channel router uses only one trunk for each net. This restriction has the effect of minimizing the number of connections between the routing layers. This is equivalent to minimizing the number of vias used by the channel router in a two-layer metal technology. Minimizing the number of vias is an important objective in routing a channel, but it is not always practical. Sometimes constraints will force a channel router to use jogs or other methods to complete the routing (see Section 17.2.5). Next, though, we shall study an algorithm that solves the restricted channel-routing problem.

17.2.4 Left-Edge Algorithm

The **left-edge algorithm** (**LEA**) is the basis for several routing algorithms [Hashimoto and Stevens, 1971]. The LEA applies to two-layer channel routing, using one layer for the trunks and the other layer for the branches. For example, m1 may be used in the horizontal direction and m2 in the vertical direction. The LEA proceeds as follows:

1. Sort the nets according to the leftmost edges of the net's horizontal segment.
2. Assign the first net on the list to the first free track.
3. Assign the next net on the list, which will fit, to the track.
4. Repeat this process from step 3 until no more nets will fit in the current track.
5. Repeat steps 2–4 until all nets have been assigned to tracks.
6. Connect the net segments to the top and bottom of the channel.

Figure 17.15 illustrates the LEA. The algorithm works as long as none of the branches touch—which may occur if there are terminals in the same column belonging to different nets. In this situation we have to make sure that the trunk that connects to the top of the channel is placed above the lower trunk. Otherwise two branches will overlap and short the nets together. In the next section we shall examine this situation more closely.

17.2.5 Constraints and Routing Graphs

Two terminals that are in the same column in a channel create a **vertical constraint**. We say that the terminal at the top of the column imposes a vertical constraint on the lower terminal. We can draw a graph showing the vertical constraints imposed by terminals. The nodes in a **vertical-constraint graph** represent terminals. A vertical constraint between two terminals is shown by an edge of the graph connecting the two terminals. A graph that contains information in the direction of an edge is a **directed graph**. The arrow on the graph edge shows the direction of the con-

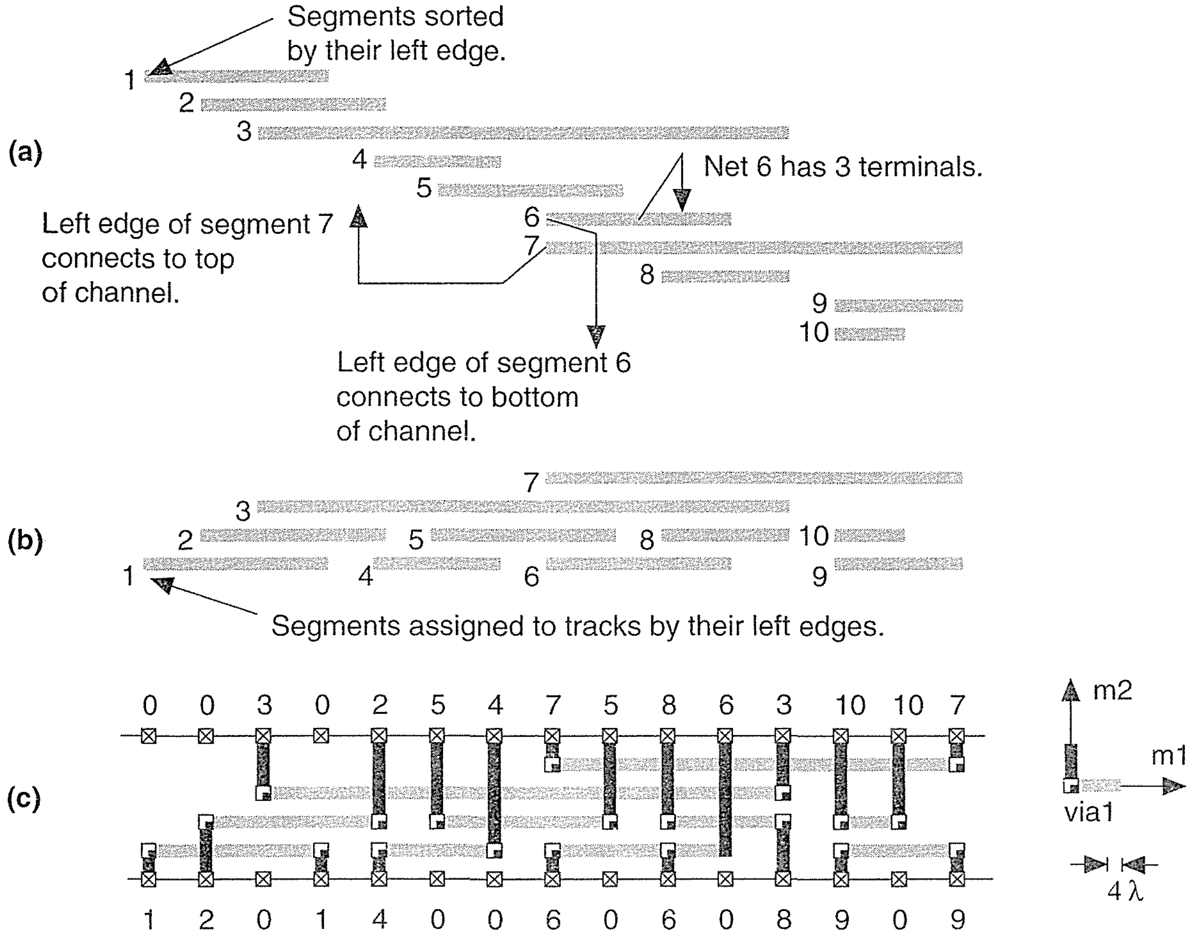

(a)
Segments sorted by their left edge.
1
2
3
4
5
6
7
8
9
10
Net 6 has 3 terminals.
Left edge of segment 7 connects to top of channel.
Left edge of segment 6 connects to bottom of channel.
(b)
7
3
2
5
8
10
1
4
6
9
Segments assigned to tracks by their left edges.
(c)
0 0 3 0 2 5 4 7 5 8 6 3 10 10 7
1 2 0 1 4 0 0 6 0 6 0 8 9 0 9
m2
m1
via1
4 λ

FIGURE 17.15 Left-edge algorithm. (a) Sorted list of segments. (b) Assignment to tracks. (c) Completed channel route (with m1 and m2 interconnect represented by lines).

straint—pointing to the lower terminal, which is constrained. Figure 17.16(a) shows an example of a channel, and Figure 17.16(b) shows its vertical constraint graph.

We can also define a **horizontal constraint** and a corresponding **horizontal-constraint graph**. If the trunk for net 1 overlaps the trunk of net 2, then we say there is a horizontal constraint between net 1 and net 2. Unlike a vertical constraint, a horizontal constraint has no direction. Figure 17.16(c) shows an example of a horizontal constraint graph and shows a group of 4 terminals (numbered 3, 5, 6, and 7) that must all overlap. Since this is the largest such group, the global channel density is 4.

If there are no vertical constraints at all in a channel, we can guarantee that the LEA will find the minimum number of routing tracks. The addition of vertical constraints transforms the restricted routing problem into an NP-complete problem. There is also an arrangement of vertical constraints that none of the algorithms based on the LEA can cope with. In Figure 17.17(a) net 1 is above net 2 in the first

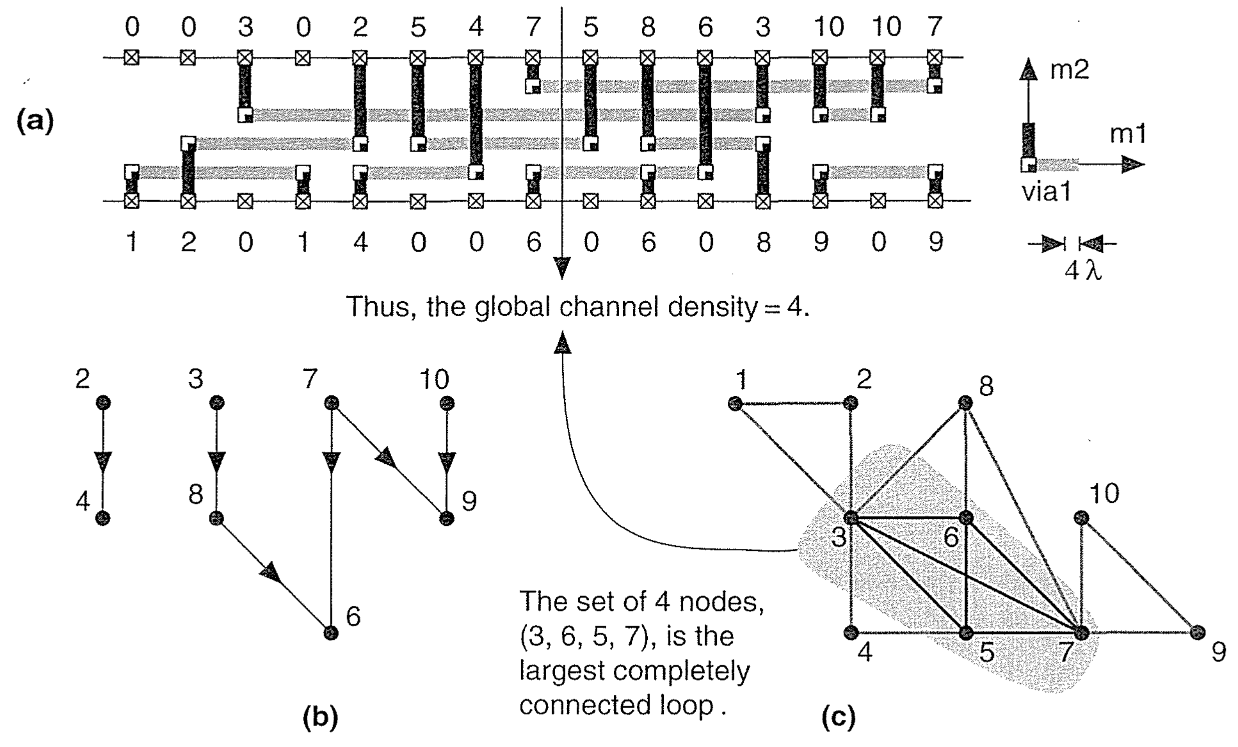

(a)
0 0 3 0 2 5 4 7 5 8 6 3 10 10 7
1 2 0 1 4 0 0 6 0 6 0 8 9 0 9
m2
m1
via1
4λ
Thus, the global channel density = 4.
(b)
2 3 7 10
4 8 9
6
The set of 4 nodes, (3, 6, 5, 7), is the largest completely connected loop .
(c)
1 2 8
10
3 6
4 5 7 9

FIGURE 17.16 Routing graphs. (a) Channel with a global density of 4. (b) The vertical constraint graph. If two nets occupy the same column, the net at the top of the channel imposes a vertical constraint on the net at the bottom. For example, net 2 imposes a vertical constraint on net 4. Thus the interconnect for net 4 must use a track above net 2. (c) Horizontal-constraint graph. If the segments of two nets overlap, they are connected in the horizontal-constraint graph. This graph determines the global channel density.

column of the channel. Thus net 1 imposes a vertical constraint on net 2. Net 2 is above net 1 in the last column of the channel. Then net 2 also imposes a vertical constraint on net 1. It is impossible to route this arrangement using two routing layers with the restriction of using only one trunk for each net. If we construct the vertical-constraint graph for this situation, shown in Figure 17.17(b), there is a loop or cycle between nets 1 and 2. If there is any such **vertical-constraint cycle** (or **cyclic constraint**) between two or more nets, the LEA will fail. A **dogleg router** removes the restriction that each net can use only one track or trunk. Figure 17.17(c) shows how adding a dogleg permits a channel with a cyclic constraint to be routed.

The channel-routing algorithms we have described so far do not allow interconnects on one layer to run on top of other interconnects on a different layer. These algorithms allow interconnects to cross at right angles to each other on different layers, but not to **overlap**. When we remove the restriction that horizontal and vertical routing must use different layers, the density of a channel is no longer the lower bound for the number of tracks required. For two routing layers the ultimate lower

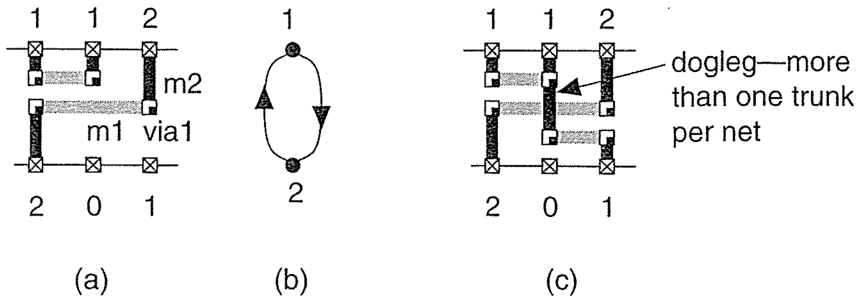


FIGURE 17.17 The addition of a dogleg, an extra trunk, in the wiring of a net can resolve cyclic vertical constraints.

bound becomes half of the channel density. The practical reasoning for restricting overlap is the parasitic **overlap capacitance** between signal interconnects. As the dimensions of the metal interconnect are reduced, the capacitance between adjacent interconnects on the same layer (**coupling capacitance**) is comparable to the capacitance of interconnects that overlap on different layers (**overlap capacitance**). Thus, allowing a short overlap between interconnects on different layers may not be as bad as allowing two interconnects to run adjacent to each other for a long distance on the same layer. Some routers allow you to specify that two interconnects must not run adjacent to each other for more than a specified length.

The channel height is fixed for channeled gate arrays; it is variable in discrete steps for channelless gate arrays; it is continuously variable for cell-based ASICs. However, for all these types of ASICs, the channel wiring is fully customized and so may be compacted or compressed after a channel router has completed the interconnect. The use of **channel-routing compaction** for a two-layer channel can reduce the channel height by 15 percent to 20 percent [Cheng et al., 1992].

Modern channel routers are capable of routing a channel at or near the theoretical minimum density. We can thus consider channel routing a solved problem. Most of the difficulty in detailed routing now comes from the need to route more than two layers and to route arbitrary shaped regions. These problems are best handled by area routers.

17.2.6 Area-Routing Algorithms

There are many algorithms used for the detailed routing of general-shaped areas (see the paper by Ohtsuki in [Ohtsuki, 1986]). Many of these were originally developed for PCB wiring. The first group we shall cover and the earliest to be used historically are the **grid-expansion** or **maze-running** algorithms. A second group of methods, which are more efficient, are the **line-search** algorithms.

Figure 17.18 illustrates the **Lee maze-running algorithm**. The goal is to find a path from X to Y—i.e., from the start (or source) to the finish (or target)—avoiding any obstacles. The algorithm is often called **wave propagation** because it sends out waves, which spread out like those created by dropping a stone into a pond.

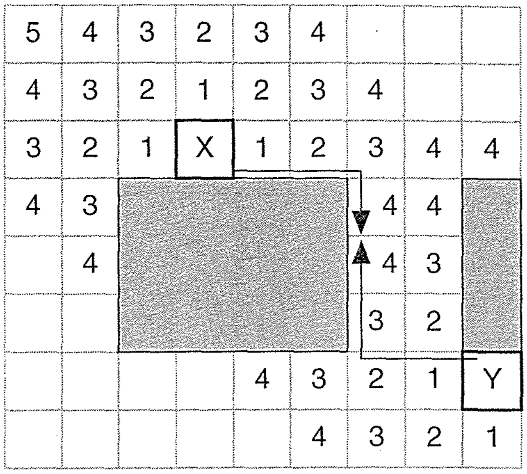


FIGURE 17.18 The Lee maze-running algorithm. The algorithm finds a path from source (X) to target (Y) by emitting a wave from both the source and the target at the same time. Successive outward moves are marked in each bin. Once the target is reached, the path is found by backtracking (if there is a choice of bins with equal labeled values, we choose the bin that avoids changing direction). (The original form of the Lee algorithm uses a single wave.)

Algorithms that use lines rather than waves to search for connections are more efficient than algorithms based on the Lee algorithm. Figure 17.19 illustrates the **Hightower algorithm**—a **line-search algorithm** (or **line-probe algorithm**):

1. Extend lines from both the source and target toward each other.
2. When an extended line, known as an **escape line**, meets an obstacle, choose a point on the escape line from which to project another escape line at right angles to the old one. This point is the **escape point**.
3. Place an escape point on the line so that the next escape line just misses the edge of the obstacle. Escape lines emanating from the source and target intersect to form the path.

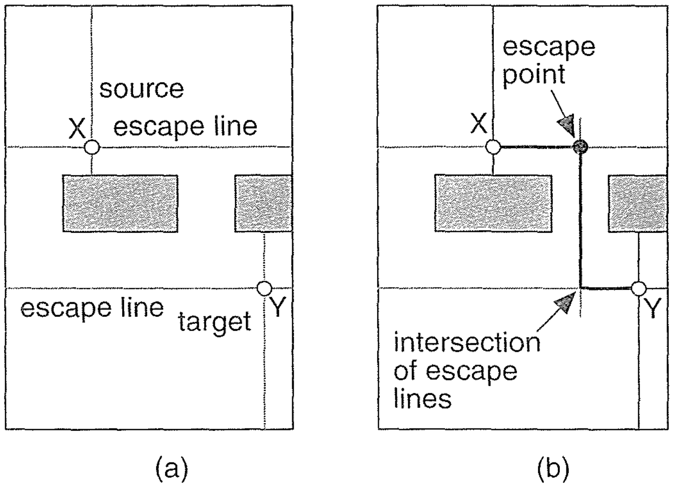


FIGURE 17.19 Hightower area-routing algorithm. (a) Escape lines are constructed from source (X) and target (Y) toward each other until they hit obstacles. (b) An escape point is found on the escape line so that the next escape line perpendicular to the original misses the next obstacle. The path is complete when escape lines from source and target meet.

The Hightower algorithm is faster and requires less memory than methods based on the Lee algorithm.

17.2.7 Multilevel Routing

Using **two-layer routing**, if the logic cells do not contain any m2, it is possible to complete some routing in m2 using over-the-cell (OTC) routing. Sometimes poly is used for short connections in the channel in a two-level metal technology; this is known as **2.5-layer routing**. Using a third level of metal in **three-layer routing**, there is a choice of approaches. **Reserved-layer routing** restricts all the interconnect on each layer to flow in one direction in a given routing area (for example, in a channel, either parallel or perpendicular to the channel spine). **Unreserved-layer routing** moves in both horizontal and vertical directions on a given layer. Most routers use reserved routing. Reserved three-level metal routing offers another choice: Either use m1 and m3 for horizontal routing (parallel to the channel spine), with m2 for vertical routing (**HVH routing**) or use **VHV routing**. Since the logic cell interconnect usually blocks most of the area on the m1 layer, HVH routing is normally used. It is also important to consider the pitch of the layers when routing in the same direction on two different layers. Using HVH routing it is preferable for the m3 pitch to be a simple multiple of the m1 pitch (ideally they are the same). Some processes have more than three levels of metal. Sometimes the upper one or two metal layers have a coarser pitch than the lower layers and are used in **multilevel routing** for power and clock lines rather than for signal interconnect.

Figure 17.20 shows an example of three-layer channel routing. The logic cells are 64 λ high, the m1 routing pitch is 8 λ, and the m2 and m3 routing pitch is 16 λ. The channel in Figure 17.20 is the same as the channel using two-layer metal shown in Figure 17.13, but using three-level metal reduces the channel height from 40 λ (=5 × 8 λ) to 16 λ. Submicron processes try to use the same metal pitch on all metal layers. This makes routing easier but processing more difficult.

With three or more levels of metal routing it is possible to reduce the channel height in a row-based ASIC to zero. All of the interconnect is then completed over the cell. If all of the channels are eliminated, the core area (logic cells plus routing) is determined solely by the logic-cell area. The point at which this happens depends on not only the number of metal layers and channel density, but also the routing resources (the blockages and feedthroughs) in the logic cell. This the **cell porosity**. Designing porous cells that help to minimize routing area is an art. For example, it is quite common to be able to produce a smaller chip using larger logic cells if the larger cells have more routing resources.

17.2.8 Timing-Driven Detailed Routing

In detailed routing the global router has already set the path the interconnect will follow. At this point little can be done to improve timing except to reduce the number of vias, alter the interconnect width to optimize delay, and minimize overlap

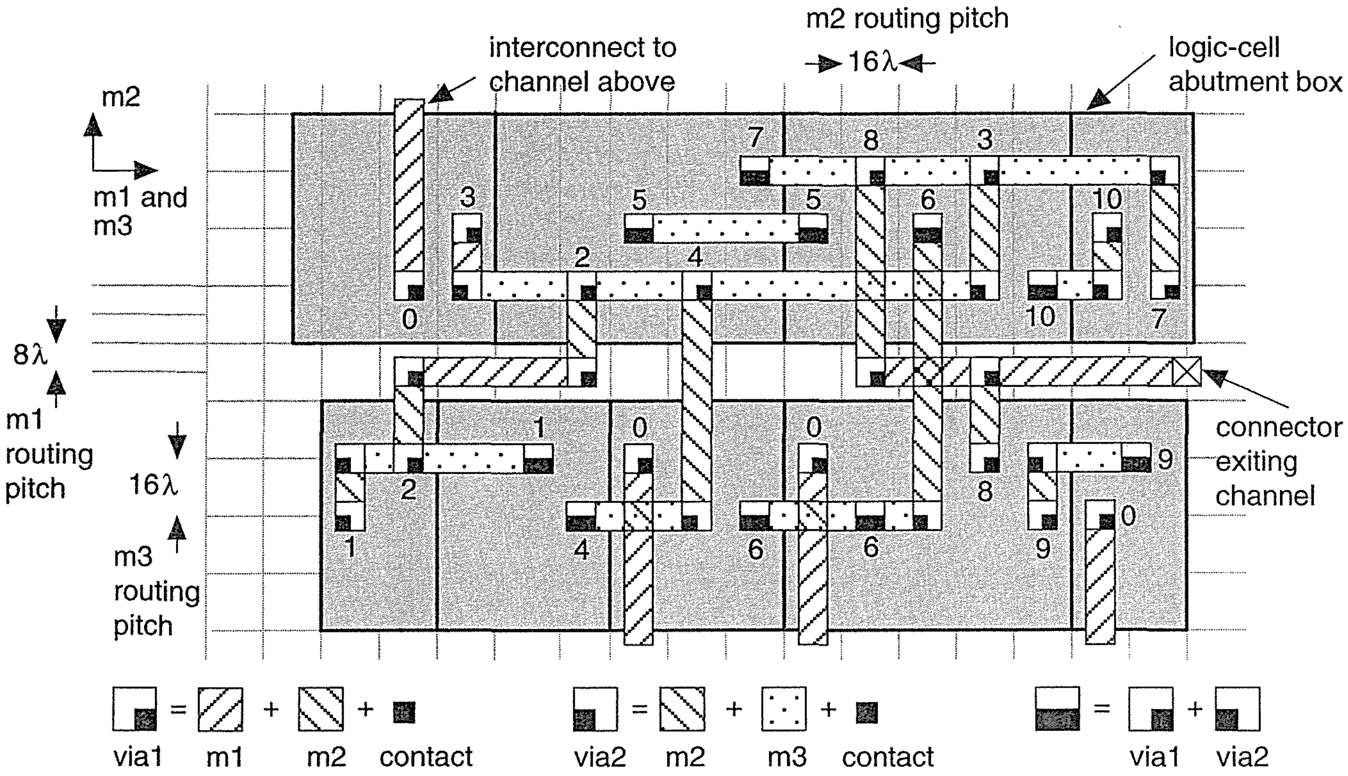


FIGURE 17.20 Three-level channel routing. In this diagram the m2 and m3 routing pitch is set to twice the m1 routing pitch. Routing density can be increased further if all the routing pitches can be made equal—a difficult process challenge.

capacitance. The gains here are relatively small, but for very long branching nets even small gains may be important. For high-frequency clock nets it may be important to shape and **chamfer** (round) the interconnect to match impedances at branches and control reflections at corners.

17.2.9 Final Routing Steps

If the algorithms to estimate congestion in the floorplanning tool accurately perfectly reflected the algorithms used by the global router and detailed router, routing completion should be guaranteed. Often, however, the detailed router will not be able to completely route all the nets. These problematical nets are known as **unroutes**. Routers handle this situation in one of two ways. The first method leaves the problematical nets unconnected. The second method completes all interconnects anyway but with some design-rule violations (the problematical nets may be shorted to other nets, for example). Some tools flag these problems as a warning (in fact there can be no more serious error).

If there are many unroutes the designer needs to discover the reason and return to the floorplanner and change channel sizes (for a cell-based ASIC) or increase the base-array size (for a gate array). Returning to the global router and changing bin sizes or adjusting the algorithms may also help. In drastic cases it may be necessary to change the floorplan. If just a handful of difficult nets remain to be routed, some tools allow the designer to perform hand edits using a **rip-up and reroute** router (sometimes this is done automatically by the detailed router as a last phase in the routing procedure anyway). This capability also permits **engineering change orders (ECO)**—corresponding to the little yellow wires on a PCB. One of the last steps in routing is **via removal**—the detailed router looks to see if it can eliminate any vias (which can contribute a significant amount to the interconnect resistance) by changing layers or making other modifications to the completed routing. **Routing compaction** can then be performed as the final step.

17.3 Special Routing

The routing of nets that require special attention, clock and power nets for example, is normally done before detailed routing of signal nets. The architecture and structure of these nets is performed as part of floorplanning, but the sizing and topology of these nets is finalized as part of the routing step.

17.3.1 Clock Routing

Gate arrays normally use a clock spine (a regular grid), eliminating the need for special routing (see Section 16.1.6, "Clock Planning"). The clock distribution grid is designed at the same time as the gate-array base to ensure a minimum clock skew and minimum clock latency—given power dissipation and clock buffer area limitations. Cell-based ASICs may use either a clock spine, a clock tree, or a hybrid approach. Figure 17.21 shows how a clock router may minimize clock skew in a clock spine by making the path lengths, and thus net delays, to every leaf node equal—using jogs in the interconnect paths if necessary. More sophisticated clock routers perform **clock-tree synthesis** (automatically choosing the depth and structure of the clock tree) and **clock-buffer insertion** (equalizing the delay to the leaf nodes by balancing interconnect delays and buffer delays).

The clock tree may contain multiply-driven nodes (more than one active element driving a net). The net delay models that we have used break down in this case and we may have to extract the clock network and perform circuit simulation, followed by back-annotation of the clock delays to the netlist (for circuit extraction, see Section 17.4) and the bus currents to the clock router. The sizes of the clock buses depend on the current they must carry. The limits are set by reliability issues to be discussed next.

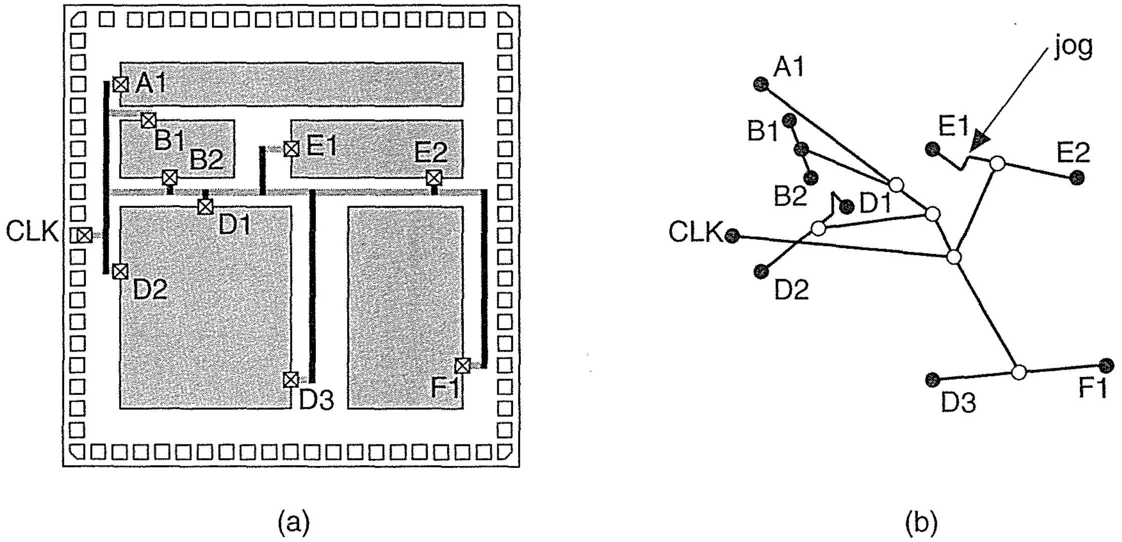


FIGURE 17.21 Clock routing. (a) A clock network for the cell-based ASIC from Figure 16.11. (b) Equalizing the interconnect segments between CLK and all destinations (by including jogs if necessary) minimizes clock skew.

Clock skew induced by hot-electron wearout was mentioned in Section 16.1.6, "Clock Planning." Another factor contributing to unpredictable clock skew is changes in clock-buffer delays with variations in power-supply voltage due to data-dependent activity. This **activity-induced clock skew** can easily be larger than the skew achievable using a clock router. For example, there is little point in using software capable of reducing clock skew to less than 100 ps if, due to fluctuations in power-supply voltage when part of the chip becomes active, the clock-network delays change by 200 ps.

The power buses supplying the buffers driving the clock spine carry direct current (unidirectional current or DC), but the clock spine itself carries alternating current (bidirectional current or AC). The difference between *electromigration* failure rates due to AC and DC leads to different rules for sizing clock buses. As we explained in Section 16.1.6, "Clock Planning," the fastest way to drive a large load in CMOS is to taper successive stages by approximately $e \approx 3$. This is not necessarily the smallest-area or lowest-power approach, however [Veendrick, 1984].

17.3.2 Power Routing

Each of the power buses has to be **sized** according to the current it will carry. Too much current in a power bus can lead to a failure through a mechanism known as **electromigration** [Young and Christou, 1994]. The required power-bus widths can be estimated automatically from library information, from a separate **power simulation** tool, or by entering the power-bus widths to the routing software by hand.

Many routers use a default power-bus width so that it is quite easy to complete routing of an ASIC without even knowing about this problem.

For a direct current (DC) the **mean time to failure** (MTTF) due to electromigration is experimentally found to obey the following equation:

$$\mathrm{MTTF} = \mathrm{A}J^{-2}\exp\frac{-E}{\mathrm{k}T}, \tag{17.9}$$

where J is the current density; E is approximately 0.5 eV; k, Boltzmann's constant, is $8.62 \times 10^{-5}\ \mathrm{eVK}^{-1}$; and T is absolute temperature in kelvins.

There are a number of different approaches to model the effect of an AC component. A typical expression is

$$\mathrm{MTTF} = \frac{\mathrm{A}\exp\frac{-E}{\mathrm{k}T}}{\bar{J}\overline{|J|} + \mathrm{k}_{\mathrm{AC/DC}}\overline{|J|}^{2}}, \tag{17.10}$$

where $\bar{J}$ is the average of $J(t)$, and $\overline{|J|}$ is the average of $|J|$. The constant $\mathrm{k}_{\mathrm{AC/DC}}$ relates the relative effects of AC and DC and is typically between 0.01 and 0.0001. Electromigration problems become serious with a MTTF of less than 10^5 hours (approximately 10 years) for current densities (DC) greater than $0.5\ \mathrm{GAm}^{-2}$ at temperatures above 150 °C.

Table 17.1 lists example **metallization reliability rules**—limits for the current you can pass through a metal layer, contact, or via—for the typical 0.5 µm three-level metal CMOS process, G5. The limit of 1 mA of current per square micron of metal cross section is a good rule-of-thumb to follow for current density in aluminum-based interconnect.

Some CMOS processes also have **maximum metal-width rules** (or **fat-metal rules**). This is because stress (especially at the corners of the die, which occurs during **die attach**—mounting the die on the chip carrier) can cause large metal areas to lift. A solution to this problem is to place slots in the wide metal lines. These rules are dependent on the ASIC vendor's level of experience.

To determine the power-bus widths we need to determine the bus currents. The largest problem is emulating the system's operating conditions. Input vectors to test the system are not necessarily representative of actual system operation. Clock-bus sizing depends strongly on the parameter $\mathrm{k}_{\mathrm{AC/DC}}$ in Eq. 17.10, since the clock spine carries alternating current. (For the sources of power dissipation in CMOS, see Section 15.5, "Power Dissipation.")

Gate arrays normally use a regular **power grid** as part of the gate-array base. The gate-array logic cells contain two fixed-width power buses inside the cell, running horizontally on m1. The horizontal m1 power buses are then strapped in a vertical direction by m2 buses, which run vertically across the chip. The resistance of the power grid is extracted and simulated with SPICE during the base-array design to model the effects of IR drops under worst-case conditions.

TABLE 17.1 Metallization reliability rules for a typical 0.5 micron (λ = 0.25μm) CMOS process.

Layer/contact/via	Current limit[1]	Metal thickness[2]	Resistance[3]
m1	$1\ \text{mA}\ \mu\text{m}^{-1}$	7000 Å	95 mΩ/square
m2	$1\ \text{mA}\ \mu\text{m}^{-1}$	7000 Å	95 mΩ/square
m3	$2\ \text{mA}\ \mu\text{m}^{-1}$	12,000 Å	48 mΩ/square
0.8μm square m1 contact to diffusion	0.7 mA		11 Ω
0.8 μm square m1 contact to poly	0.7 mA		16 Ω
0.8 μm square m1/m2 via (via1)	0.7 mA		3.6 Ω
0.8 μm square m2/m3 via (via2)	0.7 mA		3.6 Ω

[1] At 125 °C for unidirectional current. Limits for 110 °C are × 1.5 higher. Limits for 85 °C are × 3 higher. Current limits for bidirectional current are × 1.5 higher than the unidirectional limits.

[2] 10,000 Å (ten thousand angstroms) = 1 μm.

[3] Worst case at 110 °C.

Standard cells are constructed in a similar fashion to gate-array cells, with power buses running horizontally in m1 at the top and bottom of each cell. A row of standard cells uses **end-cap cells** that connect to the VDD and VSS power buses placed by the power router. Power routing of cell-based ASICs may include the option to include vertical m2 straps at a specified intervals. Alternatively the number of standard cells that can be placed in a row may be limited during placement. The power router forms an interdigitated comb structure, minimizing the number of times a VDD or VSS power bus needs to change layers. This is achieved by routing with a **routing bias** on preferred layers. For example, VDD may be routed with a left-and-down bias on m1, with VSS routed using right-and-up bias on m2.

Three-level metal processes either use a m3 with a thickness and pitch that is comparable to m1 and m2 (which usually have approximately the same thickness and pitch) or they use metal that is much thicker (up to twice as thick as m1 and m2) with a coarser pitch (up to twice as wide as m1 and m2). The factor that determines the m3/4/5 properties is normally the sophistication of the fabrication process.

In a three-level metal process, power routing is similar to two-level metal ASICs. Power buses inside the logic cells are still normally run on m1. Using HVH routing it would be possible to run the power buses on m3 and drop vias all the way down to m1 when power is required in the cells. The problem with this approach is that it creates pillars of blockage across all three layers.

Using three or more layers of metal for routing, it is possible to eliminate some of the channels completely. In these cases we complete all the routing in m2 and m3 on top of the logic cells using connectors placed in the center of the cells on m1. If we can eliminate the channels between cell rows, we can flip rows about a horizontal axis and abut adjacent rows together (a technique known as **flip and abut**). If the power buses are at the top (VDD) and bottom (VSS) of the cells in m1 we can abut or overlap the power buses (joining VDD to VDD and VSS to VSS in alternate rows).

Power distribution schemes are also a function of process and packaging technology. Recall that flip-chip technology allows pads to be placed anywhere on a chip (see Section 16.1.5, "I/O and Power Planning," especially Figure 16.13d). Four-level metal and aggressive stacked-via rules allow I/O pad circuits to be placed in the core. The problems with this approach include placing the ESD and latch-up protection circuits required in the I/O pads (normally kept widely separated from core logic) adjacent to the logic cells in the core.

17.4 Circuit Extraction and DRC

After detailed routing is complete, the exact length and position of each interconnect for every net is known. Now the parasitic capacitance and resistance associated with each interconnect, via, and contact can be calculated. This data is generated by a **circuit-extraction** tool in one of the formats described next. It is important to extract the parasitic values that will be on the silicon wafer. The mask data or CIF widths and dimensions that are drawn in the logic cells are not necessarily the same as the final silicon dimensions. Normally mask dimensions are altered from drawn values to allow for process bias or other effects that occur during the transfer of the pattern from mask to silicon. Since this is a problem that is dealt with by the ASIC vendor and not the design software vendor, ASIC designers normally have to ask very carefully about the details of this problem.

Table 17.2 shows values for the parasitic capacitances for a typical 1 μm CMOS process. Notice that the fringing capacitance is greater than the parallel-plate (area) capacitance for all layers except poly. Next, we shall describe how the parasitic information is passed between tools.

17.4.1 SPF, RSPF, and DSPF

The **standard parasitic format** (**SPF**) (developed by Cadence [1990], now in the hands of OVI) describes interconnect delay and loading due to parasitic resistance and capacitance. There are three different forms of SPF: two of them (**regular SPF** and **reduced SPF**) contain the same information, but in different formats, and model the behavior of interconnect; the third form of SPF (**detailed SPF**) describes the actual parasitic resistance and capacitance components of a net. Figure 17.22 shows the different types of simplified models that regular and reduced SPF support. The load at the output of gate A is represented by one of three models: lumped-C, lumped-RC, or PI segment. The pin-to-pin delays are modeled by RC delays. You can represent the pin-to-pin interconnect delay by an ideal voltage source, V(A_1) in this case, driving an RC network attached to each input pin. The actual pin-to-pin delays may not be calculated this way, however.

TABLE 17.2 Parasitic capacitances for a typical 1 μm ($\lambda = 0.5\,\mu$m) three-level metal CMOS process.[1]

Element	Area/fFμm^{-2}	Fringing/fFμm^{-1}
poly (over gate oxide) to substrate	1.73	NA[2]
poly (over field oxide) to substrate	0.058	0.043
m1 to diffusion or poly	0.055	0.049
m1 to substrate	0.031	0.044
m2 to diffusion	0.019	0.038
m2 to substrate	0.015	0.035
m2 to poly	0.022	0.040
m2 to m1	0.035	0.046
m3 to diffusion	0.011	0.034
m3 to substrate	0.010	0.033
m3 to poly	0.012	0.034
m3 to m1	0.016	0.039
m3 to m2	0.035	0.049
n+ junction (at 0V bias)	0.36	NA
p+ junction (at 0V bias)	0.46	NA

[1] Fringing capacitances are per isolated line. Closely spaced lines will have reduced fringing capacitance and increased interline capacitance, with increased total capacitance.

[2] NA = not applicable.

The key features of regular and reduced SPF are as follows:

- The loading effect of a net as seen by the driving gate is represented by choosing one of three different RC networks: lumped-C, lumped-RC, or PI segment (selected when generating the SPF) [O'Brien and Savarino, 1989].
- The pin-to-pin delays of each path in the net are modeled by a simple RC delay (one for each path). This can be the Elmore constant for each path (see Section 17.1.2), but it need not be.

Here is an example regular SPF file for just one net that uses the PI segment model shown in Figure 17.22(e):

```
#Design Name : EXAMPLE1
#Date : 6 August 1995
#Time : 12:00:00
#Resistance Units : 1 ohms
#Capacitance Units : 1 pico farads
```

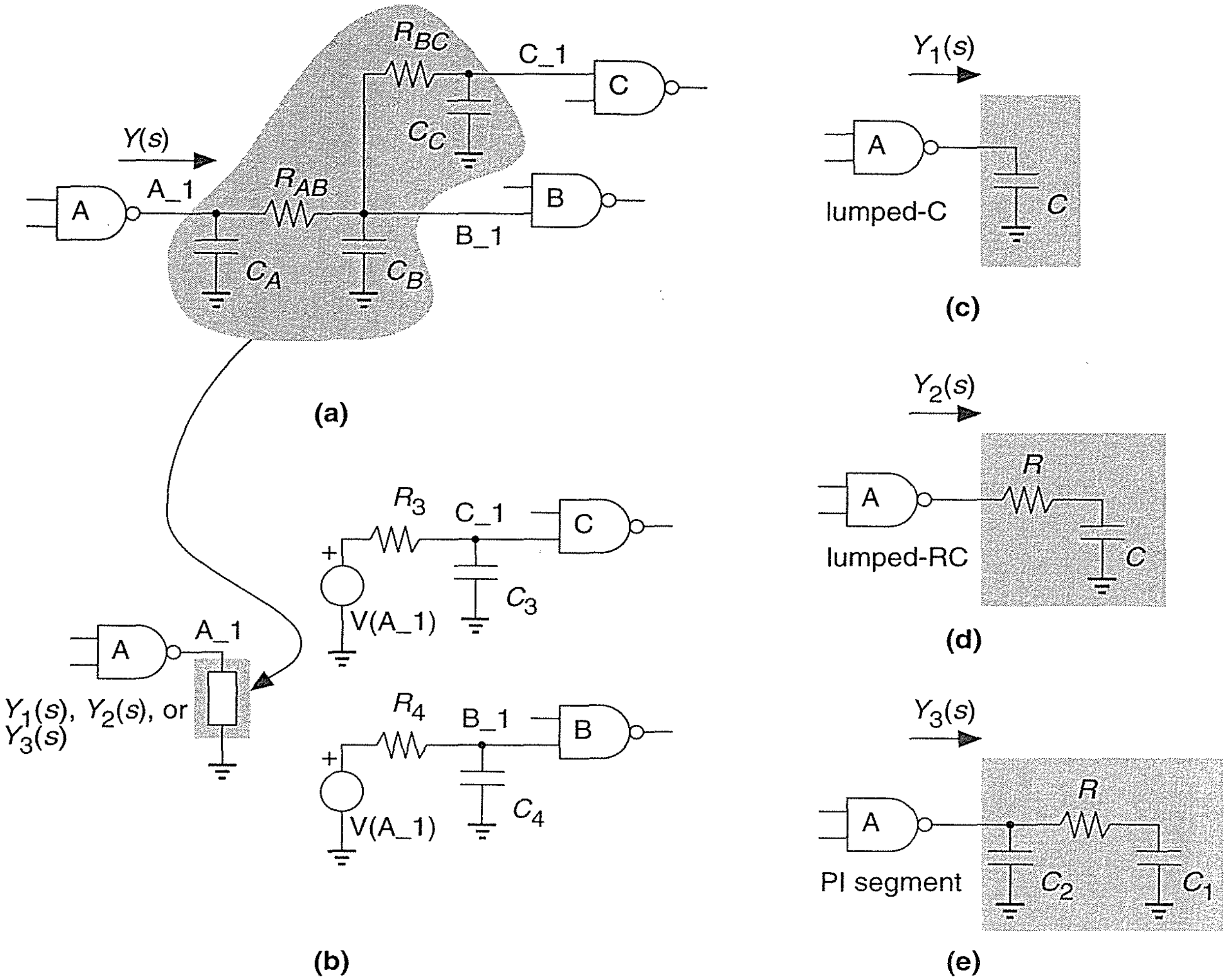


FIGURE 17.22 The regular and reduced standard parasitic format (SPF) models for interconnect. (a) An example of an interconnect network with fanout. The driving-point admittance of the interconnect network is $Y(s)$. (b) The SPF model of the interconnect. (c) The lumped-capacitance interconnect model. (d) The lumped-RC interconnect model. (e) The PI segment interconnect model (notice the capacitor nearest the output node is labeled C_2 rather than C_1). The values of C, R, C_1, and C_2 are calculated so that $Y_1(s)$, $Y_2(s)$, and $Y_3(s)$ are the first-, second-, and third-order Taylor-series approximations to $Y(s)$.

```
#Syntax :
#N <netName>
#C <capVal>
# F <from CompName> <fromPinName>
# GC <conductance>
# |
# REQ <res>
# GRC <conductance>
# T <toCompName> <toPinName> RC <rcConstant> A <value>
# |
# RPI <res>
# C1 <cap>
# C2 <cap>
# GPI <conductance>
# T <toCompName> <toPinName> RC <rcConstant> A <value>
# TIMING.ADMITTANCE.MODEL = PI
# TIMING.CAPACITANCE.MODEL = PP
N CLOCK
C 3.66
  F ROOT Z
  RPI 8.85
  C1 2.49
  C2 1.17
  GPI = 0.0
  T DF1 G RC 22.20
  T DF2 G RC 13.05
```

This file describes the following:

- The preamble contains the file format.
- This representation uses the PI segment model (Figure 17.22e).
- This net uses pin-to-pin timing.
- The driving gate of this net is ROOT and the output pin name is Z.
- The PI segment elements have values: C1 = 2.49 pF, C2 = 1.17 pF, RPI = 8.85 Ω. Notice the order of C1 and C2 in Figure 17.22(e). The element GPI is not normally used in SPF files.
- The delay from output pin Z of ROOT to input pin G of DF1 is 22.20 ns.
- The delay from pin Z of ROOT to pin G of DF2 is 13.05 ns.

The **reduced SPF** (RSPF) contains the same information as regular SPF, but uses the SPICE format. Here is an example RSPF file that corresponds to the previous regular SPF example:

```
* Design Name : EXAMPLE1
* Date : 6 August 1995
* Time : 12:00:00
* Resistance Units : 1 ohms
* Capacitance Units : 1 pico farads
*| RSPF 1.0
*| DELIMITER "_"
.SUBCKT EXAMPLE1 OUT IN
*| GROUND_NET VSS
* TIMING.CAPACITANCE.MODEL = PP
*|NET CLOCK 3.66PF
*|DRIVER ROOT_Z ROOT Z
```

```
*|S (ROOT_Z_OUTP1 0.0 0.0)
R2 ROOT_Z ROOT_Z_OUTP1 8.85
C1 ROOT_Z_OUTP1 VSS 2.49PF
C2 ROOT_Z VSS 1.17PF
*|LOAD DF2_G DF1 G
*|S (DF1_G_INP1 0.0 0.0)
E1 DF1_G_INP1 VSS ROOT_Z VSS 1.0
R3 DF1_G_INP1 DF1_G 22.20
C3 DF1_G VSS 1.0PF
*|LOAD DF2_G DF2 G
*|S (DF2_G_INP1 0.0 0.0)
E2 DF2_G_INP1 VSS ROOT_Z VSS 1.0
R4 DF2_G_INP1 DF2_G 13.05
C4 DF2_G VSS 1.0PF
*Instance Section
XDF1 DF1_Q DF1_QN DF1_D DF1_G DF1_CD DF1_VDD DF1_VSS DFF3
XDF2 DF2_Q DF2_QN DF2_D DF2_G DF2_CD DF2_VDD DF2_VSS DFF3
XROOT ROOT_Z ROOT_A ROOT_VDD ROOT_VSS BUF
.ENDS
.END
```

This file has the following features:

- The PI segment elements (C1, C2, and R2) have the same values as the previous example.
- The pin-to-pin delays are modeled at each of the gate inputs with a capacitor of value 1 pF (C3 and C4 here) and a resistor (R3 and R4) adjusted to give the correct RC delay. Since the load on the output gate is modeled by the PI segment it does not matter what value of capacitance is chosen here.
- The RC elements at the gate inputs are driven by ideal voltage sources (E1 and E2) that are equal to the voltage at the output of the driving gate.

The **detailed SPF** (DSPF) shows the resistance and capacitance of each segment in a net, again in a SPICE format. There are no models or assumptions on calculating the net delays in this format. Here is an example DSPF file that describes the interconnect shown in Figure 17.23(a):

```
.SUBCKT BUFFER OUT IN
* Net Section
*|GROUND_NET VSS
*|NET IN 3.8E-01PF
*|P (IN I 0.0 0.0 5.0)
*|I (INV1:A INV A I 0.0 10.0 5.0)
C1 IN VSS 1.1E-01PF
C2 INV1:A VSS 2.7E-01PF
R1 IN INV1:A 1.7E00
*|NET OUT 1.54E-01PF
*|S (OUT:1 30.0 10.0)
```

```
*|P (OUT O 0.0 30.0 0.0)
*|I (INV:OUT INV1 OUT O 0.0 20.0 10.0)
C3 INV1:OUT VSS 1.4E-01PF
C4 OUT:1 VSS 6.3E-03PF
C5 OUT VSS 7.7E-03PF
R2 INV1:OUT OUT:1 3.11E00
R3 OUT:1 OUT 3.03E00
*Instance Section
XINV1 INV:A INV1:OUT INV
.ENDS
```

The nonstandard SPICE statements in DSPF are comments that start with '*|' and have the following formats:

```
*|I(InstancePinName InstanceName PinName PinType PinCap X Y)
*|P(PinName PinType PinCap X Y)
*|NET NetName NetCap
*|S(SubNodeName X Y)
*|GROUND_NET NetName
```

Figure 17.23(b) illustrates the meanings of the DSPF terms: InstancePinName, InstanceName, PinName, NetName, and SubNodeName. The PinType is I (for IN) or O (the letter 'O', not zero, for OUT). The NetCap is the total capacitance on each net. Thus for net IN, the net capacitance is

$$0.38 \text{ pF} = C1 + C2 = 0.11 \text{ pF} + 0.27 \text{ pF}.$$

This particular file does not use the pin capacitances, PinCap. Since the DSPF represents every interconnect segment, DSPF files can be very large in size (hundreds of megabytes).

17.4.2 Design Checks

ASIC designers perform two major checks before fabrication. The first check is a **design-rule check** (**DRC**) to ensure that nothing has gone wrong in the process of assembling the logic cells and routing. The DRC may be performed at two levels. Since the detailed router normally works with logic-cell phantoms, the first level of DRC is a **phantom-level DRC**, which checks for shorts, spacing violations, or other design-rule problems between logic cells. This is principally a check of the detailed router. If we have access to the real library-cell layouts (sometimes called **hard layout**), we can instantiate the phantom cells and perform a second-level DRC at the transistor level. This is principally a check of the correctness of the library cells. Normally the ASIC vendor will perform this check using its own software as a type of incoming inspection. The Cadence Dracula software is one de facto standard in this area, and you will often hear reference to a **Dracula deck** that consists of the Dracula code describing an ASIC vendor's design rules. Sometimes ASIC vendors will give their Dracula decks to customers so that the customers can perform the DRCs themselves.

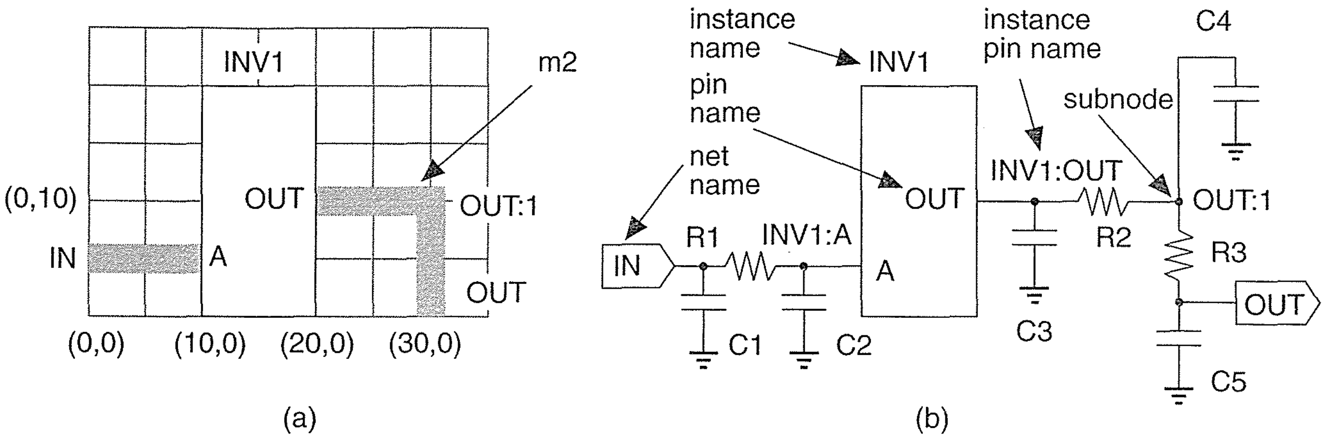


FIGURE 17.23 The detailed standard parasitic format (DSPF) for interconnect representation. (a) An example network with two m2 paths connected to a logic cell, INV1. The grid shows the coordinates. (b) The equivalent DSPF circuit corresponding to the DSPF file in the text.

The other check is a **layout versus schematic** (**LVS**) check to ensure that what is about to be committed to silicon is what is really wanted. An electrical schematic is extracted from the physical layout and compared to the netlist. This closes a loop between the logical and physical design processes and ensures that both are the same. The LVS check is not as straightforward as it may sound, however.

The first problem with an LVS check is that the transistor-level netlist for a large ASIC forms an enormous graph. LVS software essentially has to match this graph against a reference graph that describes the design. Ensuring that every node corresponds exactly to a corresponding element in the schematic (or HDL code) is a very difficult task. The first step is normally to match certain key nodes (such as the power supplies, inputs, and outputs), but the process can very quickly become bogged down in the thousands of mismatch errors that are inevitably generated initially.

The second problem with an LVS check is creating a true reference. The starting point may be HDL code or a schematic. However, logic synthesis, test insertion, clock-tree synthesis, logical-to-physical pad mapping, and several other design steps each modify the netlist. The reference netlist may not be what we wish to fabricate. In this case designers increasingly resort to formal verification that extracts a Boolean description of the function of the layout and compare that to a known good HDL description.

17.4.3 Mask Preparation

Final preparation for the ASIC artwork includes the addition of a **maskwork symbol** (M inside a circle), copyright symbol (C inside a circle), and company logos on each mask layer. A bonding editor creates a bonding diagram that will show the connection of pads to the lead carrier as well as checking that there are no design-rule violations (bond wires that are too close to each other or that leave the chip at extreme angles). We also add the **kerf** (which contains alignment marks, mask identification, and other artifacts required in fabrication), the **scribe lines** (the area where the die will be separated from each other by a diamond saw), and any special hermetic **edge-seal structures** (usually metal).

The final output of the design process is normally a magnetic tape written in **Caltech Intermediate Format** (**CIF**, a public domain text format) or **GDSII Stream** (formerly also called Calma Stream, now Cadence Stream), which is a proprietary binary format. The tape is processed by the ASIC vendor or foundry (the **fab**) before being transferred to the **mask shop**.

If the layout contains drawn *n*-diffusion and *p*-diffusion regions, then the fab generates the active (thin-oxide), *p*-type implant, and *n*-type implant layers. The fab then runs another polygon-level DRC to check polygon spacing and overlap for all mask levels. A **grace value** (typically 0.01 µm) is included to prevent false errors stemming from rounding problems and so on. The fab will then adjust the mask dimensions for fabrication either by bloating (expanding), shrinking, and merging shapes in a procedure called **sizing** or **mask tooling**. The exact procedures are described in a **tooling specification**. A **mask bias** is an amount added to a drawn polygon to allow for a difference between the mask size and the feature as it will eventually appear in silicon. The most common adjustment is to the active mask to allow for the **bird's beak effect**, which causes an active area to be several tenths of a micron smaller on silicon than on the mask.

The mask shop will use e-beam mask equipment to generate metal (usually chromium) on glass masks or **reticles**. The e-beam **spot size** determines the resolution of the mask-making equipment and is usually 0.05 µm or 0.025 µm (the smaller the spot size, the more expensive is the mask). The spot size is significant when we break the integer-lambda scaling rules in a deep-submicron process. For example, for a 0.35 µm process ($\lambda = 0.175$ µm), a 1.5 λ separation is 0.525 µm, which requires more expensive mask-making equipment with a 0.025 µm spot size. For **critical layers** (usually the polysilicon mask) the mask shop may use **optical proximity correction** (**OPC**), which adjusts the position of the mask edges to allow for light diffraction and reflection (the deep-UV light used for printing mask images on the wafer has a wavelength comparable to the minimum feature sizes).

17.5 Summary

The completion of routing finishes the ASIC physical design process. Routing is a complicated problem best divided into two steps: global and detailed routing. Global routing plans the wiring by finding the channels to be used for each path. There are differences between global routing for different types of ASICs, but the algorithms to find the shortest path are similar. Two main approaches to global routing are: one net at a time, or all nets at once. With the inclusion of timing-driven routing objectives, the routing problem becomes much harder and requires understanding the differences between finding the shortest net and finding the net with the shortest delay. Different types of detail routing include channel routing and area-based or maze routing. Detailed routing with two layers of metal is a fairly well understood problem.

The most important points in this chapter are:

- Routing is divided into global and detailed routing.
- Routing algorithms should match the placement algorithms.
- Routing is not complete if there are unroutes.
- Clock and power nets are handled as special cases.
- Clock-net widths and power-bus widths must usually be set by hand.
- DRC and LVS checks are needed before a design is complete.

17.6 Problems

* = Difficult, ** = Very difficult, *** = Extremely difficult

17.1 (Routing measures, 20 min.). Channel density is a useful measure, but with the availability of more than two layers of metal, area-based maze routers are becoming more common. Lyle Smith, in his 1983 Stanford Ph.D. thesis, defines the **Manhattan area measure** (MAM) as:

$$\text{MAM} = \text{area needed} / \text{area available}, \qquad (17.11)$$

where you calculate the area needed by assuming routing on a single layer and ignore any interconnect overlaps. Calculate the MAM for Figure 17.14. Once the MAM reaches 0.5, most two-layer routers have difficulty.

17.2 (*Benchmarking routers, 30 min.) Your design team needs a new router to complete your ASIC project. Your boss puts you in charge of benchmarking. She wants a list of the items you will test, and a description of how you will test them.

17.3 (Timing-driven routing) **(a)** Calculate the delay from A to C in Figure 17.3(b) if the wire between V_3 and V_4 is increased to 5 mm. **(b)** If you want to measure the delay to the 90 percent point, what is the skew in signal arrival time between inverters B and C? **(c)** If you use the Elmore constant to characterize the delay between inverter A and inverter C as an RC element, what is the delay (measured to the 50 percent trip point) if you replace the step function at the output of inverter A with a linear ramp with a fall time of 0.1 ns?

17.4 (Elmore delay, 30 min.) Recalculate τ_{D4}, τ_{D2}, and $\tau_{D4} - \tau_{D2}$ for the example in Section 17.1.2 neglecting the pull-down resistance R_{pd} and comment on your answers.

17.5 (Clock routing, 30 min.) Design a clock distribution system with minimum latency given the following specifications: The clocked elements are distributed randomly, but uniformly across the chip. The chip is 400 mil per side. There are 16,000 flip-flops to clock; each flip-flop clock input presents a load of 0.02 pF (one standard load). There are four different types of inverting buffer available (typical for a 0.5 μm process):

1X buffer: $T_D = 0.1 + 1.5\,C_L$ ns; 4X buffer: $T_D = 0.3 + 0.55\,C_L$ ns;

8X buffer: $T_D = 0.5 + 0.25\,C_L$ ns; 32X buffer: $T_D = 2 + 0.004\,C_L$ ns.

In these equations T_D is the buffer delay (assume rise and fall times are approximately equal) and C_L is the buffer load expressed in standard loads. Electromigration limits require a limit of 1 mA (DC) per micron metal width or 10 mA per micron for AC signals with no DC component. No metal bus may be wider than 100 μm. The m2 line capacitance is 0.015 fFμm^{-2} (area) and 0.035 fFμm^{-1} (fringing).

17.6 (Power and ground routing, 10 min.) Calculate the parallel-plate capacitance between a VDD power ring routed on m2 and an identical VSS ring routed on m1 directly underneath. The chip is 500 mil on a side; assume the power ring runs around the edge of the chip. The VDD and VSS bus are capable of carrying 0.5 A and are both 500 μm wide. Assume that m1 and m2 are separated by a SiO_2 dielectric 10,000 Å thick. This capacitance can actually be used for decoupling supplies.

17.7 (Overlap capacitance, 10 min.) Consider two interconnects, both of width W, separated by a layer of SiO_2 of thickness T, and that overlap for a distance L.

a. What is the overlap capacitance, assuming there are no fringing effects?

b. Calculate the overlap capacitance if $W = 1$ μm, $T = 0.5$ μm, for $L = 1$, 10, and 100 μm.

c. Calculate the gate capacitance of an n-channel transistor with transistor size $W/L = 2/1$ (that is, $W = 2$ μm, $L = 1$ μm), with a gate oxide thickness of 200 Å (again assuming no fringing effects).

d. Comment on your answers.

17.8 (Standard load, 10 min.) Calculate the size of a standard load for the 1 μm process with the parasitic capacitance values shown in Table 17.2. Assume the n-channel and p-channel devices in a two-input NAND gate are all 10/1 with minimum length.

17.9 (Fringing capacitance, 45 min) You can calculate the capacitance per unit length (including fringing capacitance) of an interconnect with rectangular cross section (width W, thickness T, and a distance H above a ground plane) from the approximate formula (from [Barke, 1988]—the equation was originally proposed by van der Meijs and Fokkema):

$$C = \varepsilon\left(\frac{W}{H} + 1.064\sqrt{\frac{W}{H}} + 1.06\sqrt{\frac{T}{H}} + 0.77\right), \qquad (17.12)$$

where $\varepsilon = \varepsilon_r \varepsilon_0$ is the dielectric constant of the insulator surrounding the interconnect. The relative permittivity of a SiO_2 dielectric ε_r=3.9, and the permittivity of free space $\varepsilon_0 = 3.45 \times 10^{-11}\ Fm^{-1}$.

a. Calculate C for $W = T = H = 1\ \mu m$.

b. Compare this value with the parallel-plate value (assuming no fringing capacitance).

c. Assume that the interconnect cross-sectional area (i.e., *WH*) is kept constant as technology scales, in order to keep the resistance per unit length of the interconnect constant. Assume that the width scales as *sW*, the height as *sH*, and the thickness as *T/s*, where *s* is a scaling factor from one to 0.1. Use a spreadsheet to calculate values for different scaling factors, assuming that for $s = 1$: $W = T = H = 1\ \mu m$.

d. Plot your results (with *C* on the *y*-axis vs. *s* on the *x*-axis).

17.10 (Coupling capacitance, 30 min.) One of the reasons to follow quasi-ideal scaling for the physical dimensions of the interconnect is to try and reduce the parasitic area capacitance as we scale. (The other reason is to try and keep interconnect resistance constant.) Area capacitance scales as $1/s$ by following ideal scaling rules, but scales as $1/s^{1.5}$ by using quasi-ideal scaling. Using quasi-ideal scaling means reducing the widths and horizontal spacing of the interconnect by $1/s$ and the height of the lines and their vertical separation from other layers by only $1/s^{0.5}$. The effect is rather like turning the interconnects on their sides. As a result we must consider parasitic capacitances other than just the parallel-plate capacitance between two layers. The parasitic capacitance between neighboring interconnects is called **coupling capacitance**. **Fringing capacitance** results from the fact that the electric field lines spill out from the edges of a conductor. This means the total parasitic capacitance is greater than if we just considered the capacitance to be formed by two parallel plates.

The following equation is an approximate expression for the capacitance per unit length of an isolated conductor of width *W* and thickness *T*, separated by a distance *H* from a conducting plane, and surrounded by a medium of permittivity ε [Sakurai and Tamaru, 1983]:

$$\frac{C_1}{\varepsilon} = 1.15\left(\frac{W}{H}\right) + 2.80\left(\frac{T}{H}\right)^{0.222} . \tag{17.13}$$

This equation is of the form,

$$C_1 = C_a + C_b , \tag{17.14}$$

where C_a represents the contribution from two parallel plates and C_b is the fringing capacitance (for both edges). The following equation then takes into account the coupling capacitance to a neighbor conductor separated horizontally by a gap G between the edges of the conductors:

$$\frac{C_2}{\varepsilon} = \frac{C_1}{\varepsilon} + \left[0.03\left(\frac{W}{H}\right) + 0.83\left(\frac{T}{H}\right) - 0.07\left(\frac{T}{H}\right)^{0.222}\right]\left(\frac{G}{H}\right)^{-1.34} . \tag{17.15}$$

This equation is of the form,

$$C_2 = C_1 + C_c , \tag{17.16}$$

where C_c is the coupling capacitance from the conductor to one neighbor. For a conductor having two neighbors (one on each side), the total capacitance will be

$$C_2 = C_1 + 2C_c . \tag{17.17}$$

Table 17.3 shows the result of evaluating these equations for different values of T/H, W/H, and S/H for $\lambda = 0.5\ \mu\text{m}$.

a. Calculate the corresponding values for $\lambda = 0.125\ \mu\text{m}$ assuming quasi-ideal scaling.

Table 17.4 shows the predicted fringing and coupling capacitance for a $\lambda = 0.5\ \mu\text{m}$ process expressed in pFcm^{-1}.

b. Complete the corresponding values for $\lambda = 0.125\ \mu\text{m}$, again assuming quasi-ideal scaling.

c. Comment on the difference between $\lambda = 0.5\ \mu\text{m}$ and $\lambda = 0.125\ \mu\text{m}$.

17.11 (**Routing algorithms, 60 min.) "The Lee algorithm is guaranteed to find a path if it exists, but not necessarily the shortest path." Do you agree with this statement? Can you prove or disprove it?

"The Hightower algorithm is not guaranteed to find a path, even if one exists." Do you agree with this statement? Can you prove or disprove it? *Hint:* The problems occur not with routing any one net but with routing a sequence of nets.

17.12 (Constraint graphs, 10 min.) Draw the horizontal and vertical constraint graphs for the channel shown in Figure 17.13(a). Explain how to handle the net that exits the channel and its pseudoterminal.

17.13 (**Electromigration, 60 min.) You just received the first prototype of your new ASIC. The first thing you do is measure the resistance between VDD and VSS and find they are shorted. Horrified, you find that you added your initials on m1 instead of m2 and shorted the supplies, next to the power pads. Your initials are only 10 µm wide, but about 200 µm high! Fortunately only the first capital "I" is actually shorting the supplies. The power-supply rails are approximately 100 µm wide at that

TABLE 17.3 Calculated fringing capacitance (per unit length and normalized by permittivity) using quasi-ideal scaling and the Sakurai–Tamaru equations. Problem 17.10 completes this table.

Parameter	$\lambda = 0.5\,\mu m$	$\lambda = 0.125\,\mu m$
$T\,(\mu m)$	0.5	
$W\,(\mu m)$	1.5	
$S\,(\mu m)$	1.5	
$H\,(\mu m)$	0.5	
T/H	1	
W/H, S/H	3	
$C_1 = C_a + C_b$	6.25	
$C_2 = C_1 + C_c$	6.44	
$C_3 = C_1 + 2\,C_c$	6.63	
C_a = parallel plate	3.45	
C_b = fringe (two edges)	2.80	
C_c = coupling (one neighbor)	0.19	
C_c/C_a	6%	
C_a/C_3	52%	
C_b/C_3	42%	
C_c/C_3	3%	

TABLE 17.4 Predicted line capacitance including fringing and coupling capacitance ($pFcm^{-1}$) for $\lambda = 0.125\,\mu m$ and using quasi-ideal scaling and the Sakurai equations. Problem 17.10 completes this table.

Parameter	$\lambda = 0.5\,\mu m$	$\lambda = 0.125\,\mu m$	Comment
$C_1 = C_a + C$	2.16		C_1 is capacitance of line to ground.
$C_2 = C_1 + C_c$	2.22		C_2 is capacitance including one neighbor.
$C_3 = C_1 + 2C_c$	2.29		C_3 is capacitance including two neighbors.
C_a = plate	1.19		C_a is parallel-plate capacitance.
C_b = fringe	0.97		C_b is fringe for both edges.
C_c = coupling	0.07		C_c is coupling to one neighbor only.

point. A thought occurs to you—maybe you can electromigrate your initial away. You remember that electromigration obeys an equation of the form:

$$\mathrm{MTTF} = \frac{\mathrm{A}\exp\frac{-E}{\mathrm{k}T}}{J^2} \qquad (17.18)$$

where MTTF is the mean time to failure, A is a constant, J is the current density, E is an activation energy, k is Boltzmann's constant, and T is absolute temperature. You also remember the rule that you can have about 1 mA of current for every λ of metal width for a reasonable time to failure of more than 10 years. Since this chip is in 0.5 μm CMOS ($\lambda = 0.25\ \mu$m), you guess that the metal is about 0.5 μm thick, and the resistance is at least 50 mΩ/square.

a. How much current do you estimate you need to make your initials fail so that you can test the chip before your boss gets back in a week's time?

b. What else could you do to speed things up?

c. How are you going to do this? (P.S. This sometimes actually works.)

17.14 (**Routing problems, 20 min.) We have finished the third iteration on the new game chip and are having yield problems in production. This is what we know:

1. We changed the routing on v3 by using an ECO mechanism in the detailed router from Shortem. We just ripped up a few nets and rerouted them without changing anything else.

2. The ASIC vendor, Meltem, is having yield problems due to long metal lines shorting—but only in one place. It looks as though they are the metal lines we changed in v3. Meltem blames the mask vendor—Smokem.

3. To save money we changed mask vendors after completing the prototype version v1, so that v2 and v3 uses the new mask vendor (Smokem). Smokem confirms there is a problem with the v3 mask—the lines we changed are shifted very slightly toward others and have a design rule violation. However, the v2 mask was virtually identical to v3 and there are no problems with that one, so Smokem blames the router from Shortem.

4. Shortem checks the CIF files for us, claims the mask data is correct, and they suggest we blame Meltem.

We do not care (yet) who is to blame, we just need the problem fixed. We need suggestions for the source of the problem (however crazy), some possible fixes, and some ideas to test them. Can you help?

17.15 (*Coupling capacitance, 30 min.) Suppose we have three interconnect lines running parallel to each other on a bus. Consider the following situations (VDD = 5 V, VSS = 0 V):

a. The center line switches from VSS to VDD. The neighbor lines are at VSS.

b. The center line switches from VSS to VDD. At the same time the neighbor lines switch from VDD to VSS.

- **c.** The center line switches from VSS to VDD. At the same time the neighbor lines also switch from VDD to VSS.

How do you define capacitance in these cases? In each case what is the effective capacitance from the center to the neighboring lines using your definition?

17.16 (**2LM and 3LM routing, 10 min.) How would you attempt to measure the difference in die area obtained by using the same standard-cell library with two-level and three-level routing?

17.17 (***SPF, 60 min)

- **a.** Write a regular SPF file for the circuit shown in Figure 17.3(b), using the lumped-C model and the Elmore constant for the pin-to-pin timings.
- **b.** Write the equivalent RSPF file.
- **c.** Write a DSPF file for the same circuit.
- **d.** Calculate the PI segment parameters for the circuit shown in Figure 17.3(b). *Hint:* You may need to consult [O'Brien and Savarino, 1989] if you need help.

17.18 (***Standard-cell aspect ratio, 30 min.) How would you decide the optimum value for the logic cell height of a standard-cell library?

17.19 (Electromigration, 20 min.)

- **a.** What is the current density in a 1 μm wide wire that is 1 μm thick and carries a current of 1 mA?
- **b.** Using Eq. 17.9, can you explain the temperature behavior of the parameters in Table 17.1?
- **c.** Using Eq. 17.10, can you explain the dependence on current direction?

17.20 (***SPF parameters, 120 min.). *Hint:* You may need help from [O'Brien and Savarino, 1989] for this question.

- **a.** Find an expression for $Y(s)$, where $s = j\omega$, the driving-point admittance (the reciprocal of the driving-point impedance), for the interconnect network shown in Figure 17.22(a), in terms of C_A, C_B, C_C, R_{AB}, and R_{BC}.
- **b.** Find the first three terms of the Taylor-series expansion for $Y(s)$.
- **c.** Derive expressions for $Y_1(s)$, $Y_2(s)$, and $Y_3(s)$ for the lumped-C, the lumped-RC, and the PI segment network models (Figure 17.22b–d).
- **d.** Comparing your answers to parts b and c, derive the values of the parameters of the lumped-C, the lumped-RC, and the PI segment network models in terms of C_A, C_B, C_C, R_{AB}, and R_{BC}.

17.21 (**Distributed-delay routing, 120 min. [Kahng and Robins, 1995]) The Elmore constant is one measure of net delay,

$$\tau_{Di} = \sum_k R_{ki} C_k \,. \qquad (17.19)$$

The **distributed delay**, defined as follows, is another measure of delay in a network:

$$\tau_P = \sum_k R_{kk} C_k \ . \qquad (17.20)$$

We can write this equation in terms of network components as follows:

$$\tau_P = \sum_{\text{nodes } k} (R_0 L_{kn} + R_d)\,(C_0 + C_n) \ . \qquad (17.21)$$

In this equation there are two types of capacitors: those due to the interconnect, C_0, and those due to the gate loads at each sink, C_n. R_d is the driving resistance of the driving gate (the pull-up or pull-down resistance); R_0 is the resistance of a one-grid-long piece of interconnect; and C_0 is the capacitance of a one-grid-long piece of interconnect. Thus,

$$C_k = C_0 + C_n \quad \text{and} \quad R_{kn} = R_0 L_{kn} + R_d , \qquad (17.22)$$

since every path to ground must pass through R_d. L_{kn} is the path length (in routing-grid units) between a node k and one of the n sink nodes.

With these definitions we can expand Eq. 17.21 to the following:

$$\tau_P = \sum_{\text{nodes } k} C_0 R_0 L_{kn} + \sum_{\text{nodes } k} C_n R_0 L_{kn} + R_d C_0 + R_d C_n \ . \qquad (17.23)$$

Figure 17.24 shows examples of three different types of trees. The MRST minimizes the rectilinear path length. The **shortest-path tree** (**SPT**) minimizes the sum of path lengths to all sinks. The **quadratic minimum Steiner tree** (**QMST**) minimizes the sum of path lengths to all nodes (every grid-point on the tree).

a. Find the measures for the MRST, SPT, and QMST for each of the three different tree types shown in Figure 17.24.

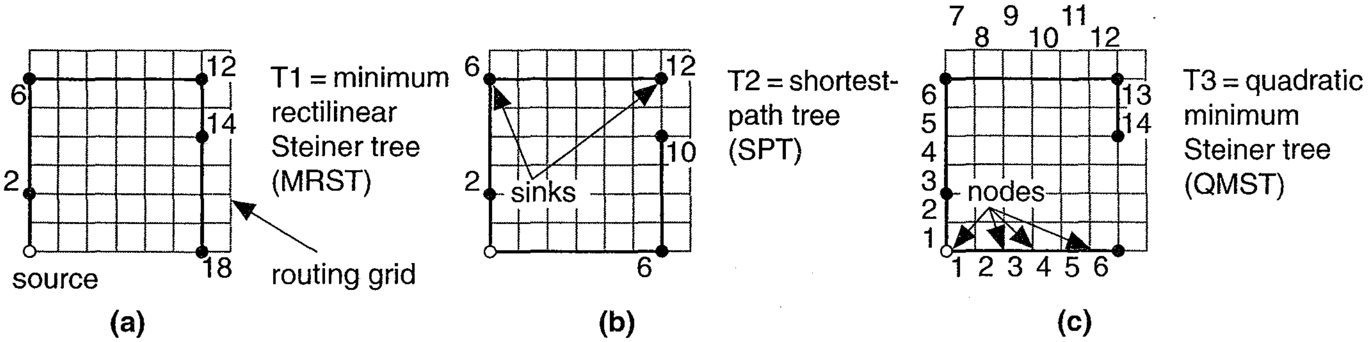
b. Explain how to apply these trees to Eq. 17.23.

c. Compare Eqs.17.19 and 17.20 for the purposes of timing-driven routing.

17.22 (**Elmore delay, 120 min.) Figure 17.25 shows an RC tree. The mth moment of the impulse response for node i in an RC tree network with n nodes is

$$\mu_1(i) = \sum_{k=1}^{n} R_{ki} C_k$$

$$\mu_{m+1}(i) = (m+1) \sum_{k=1}^{n} R_{ki} C_k \mu_m(k) \ . \qquad (17.24)$$



6
2
12
14
18
source
T1 = minimum rectilinear Steiner tree (MRST)
routing grid
(a)
6
2
12
10
6
sinks
T2 = shortest-path tree (SPT)
(b)
7 8 9 10 11 12
6 5 4 3 2 1
1 2 3 4 5 6
13
14
nodes
T3 = quadratic minimum Steiner tree (QMST)
(c)

FIGURE 17.24 Examples of trees for timing-driven layout. (a) The MRST. (b) The shortest-path tree (SPT). (c) The quadratic minimum Steiner tree (QMST). (Problem 17.21)

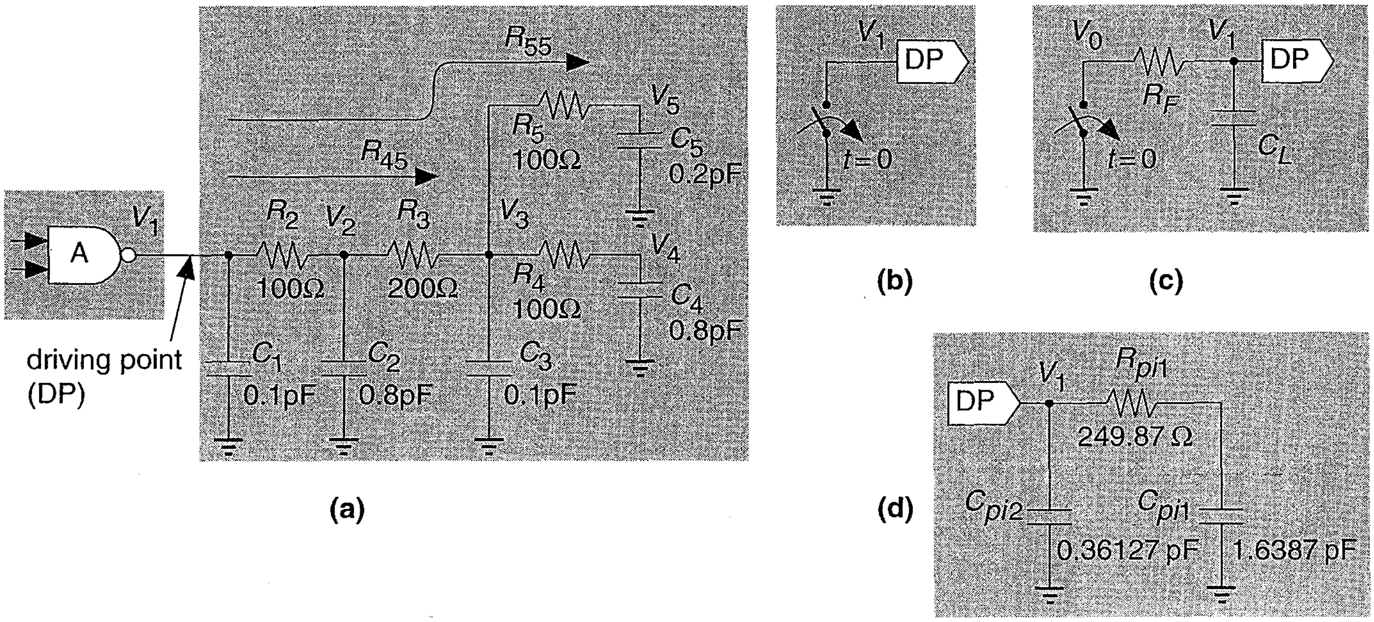


FIGURE 17.25 Standard parasitic format (SPF) (Problem 17.22). (a) An RC interconnect tree driven by a NAND gate. (b) The NAND gate modeled by an ideal switch. (c) The NAND gate modeled with a pull-down resistance, R_F, and output capacitance, C_L. (d) The PI segment model for the RC tree (the order of C_{pi1}—last—and C_{pi2} is correct).

The Elmore constant is the first moment of the impulse response. We calculate the weighted-capacitance values in Eq. 17.24 as follows:

$$k_0 = \sum_{k=1}^{n} C_k, \qquad (17.25)$$

$$k_m = \frac{1}{m!}\sum_{k=1}^{n} C_k \mu_m(k).$$

We derive the PI segment parameters used in SPF from the k_i as follows:

$$C_{pi1} = k_1^2/k_2; \quad R_{pi1} = k_2^2/k_1^3; \quad C_{pi2} = k_0 - C_{pi1}.$$

a. Calculate Elmore's constant for the RC tree in Figure 17.25(a).

b. Derive the PI segment model shown in Figure 17.25(d).

c. What is the difference between using the model of Figure 17.25(b) and the model of Figure 17.25(c) for the NAND gate?

17.7 Bibliography

The *IEEE Transactions on Computer-Aided Design* (TK7874.I327, ISSN 0278-0070) contains papers and tutorials on routing (with an emphasis on algorithms). The *Proceedings of the ACM/IEEE Design Automation Conference* (DAC, TA174.D46a, ISSN 0146-7123, catalogued under various titles) and the *Proceedings of the IEEE International Conference on Computer-Aided Design* (ICCAD, TK7874.I3235a, ISSN 1063-6757 and 1092-3152) document the two conferences at which new ideas on routing are often presented.

The edited book by Preas and Lorenzetti [1988] is the best place to learn more about routing. Books by Sarrafzadeh and Wong [1996] and Sait and Youssef [1995] are more recent introductions to physical design including routing. The book Hu and Kuh [1983] edited for IEEE Press contains early papers on routing, including an introductory paper with many references. Ohtsuki's [1986] edited book on layout contains tutorials on routing, including reviews of channel routing by Burstein and area routing by Ohtsuki; a more recent edited book by Zobrist [1994] also contains papers on routing. A good introduction to routing is Joobanni's thesis published as a book [1986]. New routing techniques are becoming important for FPGAs, a recent paper describes some of these [Roy, 1993]. Books by Lengauer [1990] and Sherwani [1993] describe algorithms for both the global and detailed routing problems. A book by Sherwani et al. [1995] covers two-level and three-level routing. Kahng and Robins [1995] cover timing-driven detailed routing in their book; Sapatnekar and Kang [1993] cover timing-driven physical design in general. The book by Pillage et al. [1994] includes a chapter on bounding and asymptotic approximations that are related to the models used in SPF. Nakhla and Zhang [1994] and Goel [1994] cover modeling of interconnect. The IEEE Press book edited by Friedman [1995] covers clock distribution. Routing is often performed in parallel on several machines; books by Banerjee [1994] and Ravikumar [1996] describe parallel algorithms for physical design. Taylor and Russell [1992] review knowledge-based physical design in an edited book.

Najm's review paper covers power estimation [1994]. Books by Shenai [1991] and Murarka [1993] cover all aspects of metallization. To learn more about the causes of electromigration in particular, see D'Heurle's [1971] classic paper and a paper by Black [1969]. The edited book by Gildenblat and Schwartz [1991] covers metallization reliability. A tutorial paper by Young and Christou [1994] reviews current theories of the causes of electromigration. To learn more about masks and microlithography in VLSI, see the handbook by Glendinning and Helbert [1991].

17.8 References

Page numbers in brackets after a reference indicate its location in the chapter body.

Banerjee, P. 1994. *Parallel Algorithms for VLSI Computer-Aided Design Applications.* Englewood Cliffs, NJ: Prentice-Hall, 699 p. ISBN 0130158356. TK7874.75.B36. [p. 956]

Barke, E. 1988. "Line-to-ground capacitance calculation for VLSI: A comparison." *IEEE Transactions on Computer-Aided Design,* Vol. 7, no. 2, pp. 295–298. Compares various equations for line to ground capacitance and finds the van der Meijs and Fokkema equation the most accurate. [p. 948]

Black, J. R. 1969. "Electromigration failure modes in aluminum metallization for semiconductor devices." *Proceedings of the IEEE,* Vol. 57, no. 9, pp. 1587–1594. Describes mechanism and theory of electromigration. Two failure modes are discussed: dissolution of silicon into aluminum, and condensation of aluminum vacancies to form voids. Electromigration failures in aluminum become important (less than 10 year lifetime) at current densities greater than 50 kA/sq.cm and temperatures greater than 150 °C. [p. 956]

Cadence. 1990. "Gate Ensemble User Guide." Product Release 2.0. Describes gate-array place-and-route software. The algorithms for timing-driven placement are described in A. H. Chao, E. M. Nequist, and T. D. Vuong, "Direct solution of performance constraints during placement," in *Proceedings of the IEEE Custom Integrated Circuits Conference,* 1990. The delay models for timing analysis are described in "Modeling the driving-point characteristic of resistive interconnect for accurate delay estimation," in P. R. O'Brien and T. L. Savarino, in *Proceedings of the International Conference on Computer-Aided Design,* 1989. [p. 939]

Cheng, C.-K., et al. 1992. "Geometric compaction on channel routing." *IEEE Transactions on Computer-Aided Design,* Vol. 11, no. 1, pp. 115–127. [p. 931]

Chowdhury, S., and J. S. Barkatullah. 1988. "Current estimation in MOS IC logic circuits." In *Proceedings of the International Conference on Computer-Aided Design.* Compares estimates for transient current flow for CMOS logic gates. Algebraic models give results close to SPICE simulations. The rest of the paper discusses the calculation of static current flow for nMOS logic gates. A model for static current for CMOS gates is developed in terms of the nMOS models.

D'Heurle, F. M. 1971. "Electromigration and failure in electronics: an introduction." *Proceedings of the IEEE,* Vol. 59, no. 10, pp. 1409–1417. Describes the theory behind electromigration in bulk and thin-film metals. Includes some experimental results and reviews work by others. Describes the beneficial effects of adding copper to aluminum metallization. [p. 956]

Friedman, E. G. (Ed.). 1995. *Clock Distribution Networks in VLSI Circuits and Systems.* New York: IEEE Press, ISBN 0780310586. TK7874.75.C58. [p. 956]

Gildenblat, G. S., and G. P. Schwartz (Eds.). 1991. *Metallization: Performance and Reliability Issues for VLSI and ULSI.* Bellingham, WA: SPIE, the International Society for Optical Engineering, 159 p. ISBN 0819407275. TK7874.M437. [p. 956]

Glendinning, W. B., and J. N. Helbert, (Eds.). 1991. *Handbook of VLSI Microlithography : Principles, Technology, and Applications.* Park Ridge, NJ: Noyes Publications, 649 p. ISBN 0815512813. TK7874.H3494. [p. 957]

Goel, A. K. 1994. *High Speed VLSI Interconnections: Modeling, Analysis, and Simulation.* New York: Wiley-Interscience, 622 p. ISBN 0471571229. TK7874.7.G63. 21 pages of references. [p. 956]

Hashimoto, A., and J. Stevens. 1971. "Wire routing by optimal channel assignment within large apertures." In *Proceedings of the 8th Design Automation Workshop,* pp. 155–169. [p. 928]

Hu, T. C., and E. S. Kuh (Eds.). 1983. *VLSI Circuit Layout: Theory and Design.* New York: IEEE Press. ISBN 0879421932. TK7874 .V5573. Contains 26 papers divided into six chapters; Part 1: Overview (a paper written for this book with 167 references on layout and routing); Part II: General; Part III: Wireability, Partitioning and Placement; Part IV: Routing; Part V: Layout Systems; Part VI: Module Generation. [p. 956]

Joobbani, R. 1986. *An Artificial Intelligence Approach to VLSI Routing.* Hingham, MA: Kluwer. ISBN 0-89838-205-X. TK7874.J663. Ph.D thesis on the development and testing of an intelligent router including an overview of the detailed routing problem and the Lee and "greedy" algorithms. [p. 956]

Kahng, A. B., and G. Robins. 1995. *On Optimal Interconnections for VLSI.* Norwell, MA: Kluwer. ISBN 0-7923-9483-6. TK7874.75.K34. Extensive reference work on timing-driven detailed routing. [pp. 953, 956]

Lengauer, T. 1990. *Combinatorial Algorithms for Integrated Circuit Layout.* Chichester, England: Wiley. ISBN 0-471-92838-0. TK7874.L36. Background: Introduction to circuit layout; Optimization problems; Graph algorithms; Operations research and statistics. Combinatorial layout problems: The layout problem; Circuit partitioning; Placement, assignment, and floorplanning; Global routing and area routing; Detailed routing; Compaction. 484 references. [p. 956]

Nakhla, M. S., and Q. J. Zhang (Eds.). 1994. *Modeling and Simulation of High Speed VLSI Interconnects.* Boston: Kluwer, 106 p. ISBN 0792394410. TK7874.75.M64. [p. 956]

Murarka, S. P. 1993. *Metallization: Theory and Practice for VLSI and ULSI.* Stoneham, MA: Butterworth-Heinemann, 250 p. ISBN 0-7506-9001-1. TK7874.M868. Includes chapters on metal properties; crystal structure; electrical and mechanical properties; diffusion and reaction in thin metallic films; deposition method and techniques; pattern definition; packaging applications; reliability. [p. 956]

Najm, F. N. 1994. "A survey of power estimation techniques in VLSI circuits." *IEEE Transactions on Very Large Scale Integration (VLSI) Systems,* Vol. 2, no. 4, pp. 446–55. 43 references. [p. 956]

O'Brien, P. R., and T. L. Savarino. 1989. "Modeling the driving-point characteristic of resistive interconnect for accurate delay estimation." In *Proceedings of the International Conference on Computer-Aided Design,* pp. 512–515. Describes SPF PI segment model. [pp. 940, 953].

Ohtsuki, T. (Ed.). 1986. *Layout Design and Verification.* New York: Elsevier Science, ISBN 0444878947. TK7874.L318. Includes nine papers on CAD tools and algorithms: "Layout strategy, standardisation, and CAD tools," Ueda, Kasai, and Sudo; "Layout compaction," Mylynski and Sung; "Layout verification," Yoshida; "Partitioning, assignment and placement," Goto and Matsuda; "Computational complexity of layout problems," Shing and Hu; "Computational and geometry algorithms," Asano, Sato, and Ohtsuki; an excellent survey and tutorial paper by M. Burstein — "Channel routing;" "Maze-running and line-search algorithms," a good, easily readable paper on detailed routing by Ohtsuki; and a more mathematical paper, "Global routing," by Kuh and Marek-Sadowska. [pp. 932, 957]

Pillage, L., et al. 1994. *Electronic Circuit and System Simulation Methods.* New York: McGraw-Hill, 392 p. ISBN 0-07-050169-6. TK7874.P52. [p. 956]

Preas, B. T., and M. J. Lorenzetti. 1988. *Physical Design Automation of VLSI Systems.* Menlo Park, CA: Benjamin-Cummings, 510 p. ISBN 0805304129. TK7874.P47. Chapters on: physical design automation; interconnection analysis, logic partitioning; placement, assignment and floorplanning; routing; symbolic layout and compaction; module generation and silicon compilation; layout analysis and verification; knowledge-based physical design automation; combinatatorial complexity of layout problems. [p. 956]

Ravikumar, C. P. 1996. *Parallel Methods for VLSI Layout Design.* Norwood, NJ: Ablex, 195 p. ISBN 0893918288. TK7874.R39. [p. 956]

Roy, K. 1993. "A bounded search algorithm for segmented channel routing for FPGA's and associated channel architecture issues." *IEEE Transactions on Computer-Aided Design,* Vol. 12, no. 11, pp. 1695–1704. [p. 956].

Sait, S. M., and H. Youssef. 1995. *VLSI Physical Design Automation, Theory and Practice.* New York: IEEE Press/McGraw-Hill copublication, 426 p. ISBN 0-07-707742-3. TK7874.75.S24. Covers floorplanning, placement, and routing. [p. 956]

Sakurai, T., and K. Tamaru. 1983. "Simple formulas for two- and three-dimensional capacitances." *IEEE Transactions on Electron Devices.* Vol. 30, no. 2. [p. 949]

Sapatnekar, S. S., and S.-M. Kang. 1993. *Design Automation for Timing-Driven Layout Synthesis.* Boston: Kluwer, 269 p. ISBN 0792392817. TK7871.99.M44.S37. 19 pages of references. [p. 956]

Sarrafzadeh, M., and C. K. Wong. 1996. *An Introduction to VLSI Physical Design.* New York: McGraw-Hill, 334 p. ISBN 0070571945. TK7874.75.S27. 17 pages of references. [p. 956]

Shenai, K. (Ed.). 1991. *VLSI Metallization: Physics and Technologies.* Boston: Artech House, 529 p. ISBN 0890065012. TK7872.C68.V58. [p. 956]

Sherwani, N. A. 1993. *Algorithms for VLSI Physical Design Automation.* 2nd ed. Norwell, MA: Kluwer, 538 p. ISBN 0-7923-9294-9. TK874.S455. See also the first edition. [p. 956]

Sherwani, N. A., et al. 1995. *Routing in the Third Dimension: From VLSI Chips to MCMs.* New York: IEEE Press. ISBN 0-7803-1089-6. TK7874.75.R68. Reviews two-layer and multilayer routing algorithms. Contains chapters on: graphs and basic algorithms; channel routing; routing models; routing algorithms for two- and three-layer processes and MCMs. [p. 956]

Taylor, G., and G. Russell. (Eds.). 1992. *Algorithmic and Knowledge Based CAD for VLSI.* London: P. Peregrinus, 273 p. ISBN 086341267X. TK7874.A416. [p. 956]

Veendrick, H. J. M. 1984. "Short-circuit dissipation of static CMOS circuitry and its impact on the design of buffer circuits." *IEEE Journal of Solid-State Circuits,* Vol. 19, no. 4, pp. 468–473. [p. 936]

Young, D., and A. Christou. 1994. "Failure mechanism models for electromigration." *IEEE Transactions on Reliability,* Vol. 43, no. 2, pp. 186–192. A tutorial on electromigration and its relation to microstructure. [pp. 936, 957]

Zobrist, G. W. (Ed.). 1994. *Routing, Placement, and Partitioning.* Norwood, NJ: Ablex, 293 p. ISBN 0893917842. TK7874.R677. [p. 956]

VHDL RESOURCES

A

The definitive reference for VHDL is the VHDL **language reference manual** (**LRM**), currently IEEE Std 1076-1993.[1] References here such as [93LRM 1.1], for example, refer to Section 1.1 of the VHDL-93 LRM [IEEE 1076-1993]. According to IEEE bylaws all standards are updated (reaffirmed, reballoted, or dropped) every five years, and thus VHDL-87 (the original standard) is superceded by VHDL-93. However, some software systems (and some IEEE standards, notably VITAL) are based on VHDL-87 [IEEE 1076-1987]. Both VHDL-93 and VHDL-87 are covered in this Appendix.

A.1 BNF

Appendix A of the LRM describes the syntax of VHDL using **keywords** (or **reserved words**) and characters in a shorthand notation called the **BNF** (**Backus–Naur form**). As an example, the BNF definition given in Appendix A of the LRM for the syntax of the `wait` statement is

```
wait_statement ::=
  [label : ] wait [ sensitivity_clause ] [ condition_clause ]
    [ timeout_clause ] ;
```

This definition means: "The `wait` statement consists of the keyword, `wait`, followed by three optional parts: a sensitivity clause, a condition clause, and a timeout clause."

You treat the BNF as a series of equations. The left-hand side is called a **production** or **construct**, the symbol **::=** (two colons and an equal sign) represents **equivalence**, the right-hand side contains the **parts** that comprise the production. Parts may be keywords (in bold here). Parts may be other productions contained in square brackets `[]`. This signifies that the part is optional. Parts may also have curly brackets or

[1] IEEE Std 1076-1993, © Copyright 1995 IEEE. All rights reserved. The BNF in this appendix is derived from the IEEE copyright material with permission.

braces {}. This indicates that the part is optional and may be repeated. The BNF is hierarchical; for example, the wait statement is defined in terms of other constructs. We can expand the wait statement definition, by substituting the BNF for sensitivity_clause, condition_clause, and timeout_clause:

```
wait_statement ::=
   [label : ] wait
      [ on signal_name { , signal_name } ]
      [ until boolean_expression ]
      [ for time_expression ] ;
```

Expanding the BNF makes it easier to see the structure of the wait statement. The expanded BNF shows that the following are valid wait statements (as far as syntax is concerned):

```
wait;
wait on a;
wait on a, b, c until count = 0 for 1 + 1 ns;
```

A disadvantage of expanding the BNF is that we lose the names and the definitions of the intermediate constructs (sensitivity_clause, condition_clause, and timeout_clause). The VHDL-93 LRM uses 238 production rules; the following section contains the same definitions in BNF, but in expanded form (using 94 rules).

There is one other disadvantage of expanding the BNF syntax definitions. Expanding the definition of a loop statement illustrates this problem:

```
loop_statement ::=
      [ loop_label : ]
   [ iteration_scheme ] loop
   sequence_of_statements
   end loop [ loop_label ] ;
```

The definition of sequence_of_statements is

```
sequence_of_statements ::= {sequential_statement}
```

The definition of iteration_scheme is

```
iteration_scheme ::= while condition | for loop_parameter_specification
```

The definitions of condition and parameter_specification are

```
condition ::=  boolean_expression
parameter_specification ::= identifier in discrete_range
```

The definition of discrete_range is

```
discrete_range ::=  discrete_subtype_indication | range
```

If we stop expanding at this level, we can write out what we have so far in our expanded definition of a loop statement:

```
loop_statement ::=
[ loop_label : ]
[ while boolean_expression
```

```
  | for identifier in discrete_range ]
  loop
    {sequential_statement}
  end loop [ loop_label ] ;
```

There is (theoretically) some ambiguity in this definition as far as the choices either side of the | symbol are concerned. Does this definition mean that we choose between **while** *boolean*_expression and **for** identifier **in** discrete_range? If we were in a contrary mood, we could also interpret the BNF as indicating a choice between *boolean*_expression and **for**. Notice that this ambiguity is also present in the definition of iteration_scheme.

Adding angle brackets around the clauses, < **while** ... > | < **for** ... >, makes the grouping of choices clear:

```
loop_statement ::=
[ loop_label : ]
[ < while boolean_expression >
  | < for identifier in discrete_range > ]
loop
  { sequential_statement }
end loop [ loop_label ] ;
```

Unfortunately the symbols < and > are already valid lexical elements in VHDL. In fact, since {} and [] are already used, and () are part of the language too, there are no brackets left to use. We live with this inconvenience. The BNF (here or in the LRM) does not define VHDL, but helps us understand it.

A.2 VHDL Syntax

In the rules that follow an underline (like this) indicates syntax that is present in VHDL-93, but not in VHDL-87. A strikethrough (~~like this~~) indicates syntax that is present in VHDL-87, but not in VHDL-93; this occurs only in the rule for file_declaration (rule 38 on p. 968). This means that any VHDL-87 code that contains keywords **in** or **out** in a file declaration will not compile in a VHDL-93 environment. Apart for this one exception, VHDL-93 is a superset of VHDL-87.

The VHDL productions are in alphabetical order. The highest-level production is the definition for design_file; this is where you start to traverse the tree starting at the top level. The following parts (indicated by the use of uppercase in the BNF) are the lowest-level constructions: UPPER_CASE_LETTER (A–Z plus accented uppercase letters), LOWER_CASE_LETTER (a–z and accented lowercase letters, é, and so on), LETTER (either uppercase or lowercase letters, a–z, and all accented letters), DIGIT (0–9), SPACE_CHARACTER (' ' and nonbreaking space), UNDERLINE ('_'), SPECIAL_CHARACTER (" # & ' ( ) * + , - . / : ; < = > [ ] _ |), and OTHER_SPECIAL_CHARACTER (all remaining characters such as ! $ % @ ? and so on, but not including format effectors). Format effectors are the ISO (and ASCII) characters called horizontal tabulation, vertical tabulation, carriage return, line feed, and form feed.

Keywords are shown in bold. Notice that the terms *label, literal,* and *range* are keywords (**label**, **literal**, **range**) and are also used as the name of a part (label, literal, range), as they are in the LRM. Construct names that commence with italics, such as *time*_expression, are intended to make the syntax definitions easier to read. The italic part of the construct is treated as a comment. You look up the definition for *time*_expression under 'e' for expression, not 't' for time. There are no formal definitions of the italic modifiers; if you are not sure exactly what is meant, you must look up the semantics in the body of the LRM.

```
actual_part ::= [93LRM 4.3.2.2]                                                  [1]
   expression
   | signal_name | variable_name | file_name | open
   | function_name (expression | signal_name | variable_name |file_name | open)
   | type_mark ( expression | signal_name | variable_name | file_name | open)

aggregate ::= [93LRM 7.3.2]                                                      [2]
   ( [ choice { | choice } => ] expression {, [ choice { | choice } => ] expression } )

alias_declaration ::= [93LRM 4.3.3]                                              [3]
   alias identifier | ' graphic_character ' | " { graphic_character } "
   [ : subtype_indication ] is name [ signature ] ;

architecture_body ::= [93LRM 1.2]                                                [4]
   architecture identifier of entity_name is
     { block_declarative_item }
   begin
     { concurrent_statement }
   end [ architecture ] [architecture_identifier ];

assertion ::= [93LRM 8.2]                                                        [5]
   assert boolean_expression [ report expression ] [ severity expression ]

association_list ::= [93LRM 4.3.2.2]                                             [6]
   [ formal_part => ] actual_part {, [ formal_part => ] actual_part }

attribute_declaration ::= [93LRM 4.4] attribute identifier : type_mark ;         [7]

attribute_name ::= [93LRM 6.6] prefix [ signature ] ' attribute_identifier       [8]
     [ ( expression ) ]

attribute_specification ::= [93LRM 5.1]                                          [9]
   attribute attribute_identifier of entity_name_list : entity_class is expression ;

based_literal ::= [93LRM 13.4.2]                                                [10]
   integer # DIGIT | LETTER { [ UNDERLINE ] DIGIT | LETTER }
   [ . DIGIT | LETTER { [ UNDERLINE ] DIGIT | LETTER } ]
   # [ E [ + ] integer | E – integer ]

basic_graphic_character ::= [93LRM 13.1]                                        [11]
   UPPER_CASE_LETTER | DIGIT | SPECIAL_CHARACTER | SPACE_CHARACTER

bit_string_literal ::= [93LRM 13.7]                                             [12]
   B | O | X " [DIGIT | LETTER { [ UNDERLINE ] DIGIT | LETTER } ]"

block_configuration ::= [93LRM 1.3.1]                                           [13]
   for architecture_name
   | block_statement_label
```

```
  | generate_statement_label
    [ ( discrete_subtype_indication | range | static_expression ) ]
  { use prefix.suffix {, prefix.suffix } ; }
  { block_configuration | component_configuration }
  end for ;

block_declarative_item ::= [93LRM 1.2.1]                                    [14]
    subprogram_specification; | subprogram_body | type_declaration
  | subtype_declaration | constant_declaration | signal_declaration
  | shared variable_declaration
  | file_declaration | alias_declaration
  | component_declaration | attribute_declaration
  | attribute_specification | configuration_specification
  | disconnection_specification | use_clause
  | group_template_declaration | group_declaration

block_statement ::= [93LRM 9.1]                                             [15]
  block_label :
  block [ ( guard_expression ) ] [ is ]
    [ generic ( generic_interface_list );
    [ generic map (generic_association_list ) ; ] ]
    [ port (port_interface_list );
    [ port map (port_association_list ) ; ] ]
      { block_declarative_item }
      begin
      { concurrent_statement }
  end block [ block_label ] ;

case_statement ::= [93LRM 8.8]                                              [16]
  [ case_label : ] case expression is
    when choice { | choice } => { sequential_statement }
    { when choice { | choice } => { sequential_statement } }
  end case [ case_label ] ;

choice ::= [93LRM 7.3.2]                                                    [17]
  simple_expression | discrete_range | element_identifier | others

component_configuration ::= [93LRM 1.3.2]                                   [18]
  for
    instantiation_label { , instantiation_label } : component_name
    | others : component_name
    | all : component_name
      [ [ use
        entity entity_name [ ( architecture_identifier ) ]
        | configuration configuration_name
        | open ]
      [ generic map (generic_association_list ) ]
      [ port map ( port_association_list ) ] ; ]
    [ block_configuration ]
  end for ;
```

```
component_declaration ::= [93LRM 4.5]                                         [19]
  component identifier [ is ]
    [ generic (local_generic_interface_list ) ; ]
    [ port (local_port_interface_list ) ; ]
  end component [ component_identifier ] ;

component_instantiation_statement ::= [93LRM 9.6]                             [20]
  instantiation_label : [ component ] component_name
    | entity entity_name [ ( architecture_identifier ) ]
    | configuration configuration_name
  [ generic map ( generic_association_list )]
  [ port map ( port_association_list ) ] ;

concurrent_statement ::= [93LRM 9]                                            [21]
  block_statement
  | process_statement
  | [ label : ] [ postponed ] procedure_call ;
  | [ label : ] [ postponed ] assertion ;
  | [ label : ] [ postponed ] conditional_signal_assignment
  | [ label : ] [ postponed ] selected_signal_assignment
  | component_instantiation_statement
  | generate_statement

conditional_signal_assignment ::= [93LRM 9.5.1]                               [22]
  name | aggregate <= [ guarded ] [ transport | [ reject time_expression ] inertial ]
  { waveform when boolean_expression else }
  waveform [ when boolean_expression ] ;

configuration_declaration ::= [93LRM 1.3]                                     [23]
  configuration identifier of entity_name is
    { use prefix.suffix { , prefix.suffix } ;
    | attribute_specification
    | group_declaration }
    block_configuration
  end [ configuration ] [ configuration_identifier ] ;

configuration_specification ::= [93LRM 5.2]                                   [24]
  for
    instantiation_label { ,instantiation_label } : component_name
    | others : component_name
    | all : component_name
  [ use
    entity entity_name [ ( architecture_identifier ) ]
    | configuration configuration_name
    | open ]
  [ generic map ( generic_association_list ) ]
  [ port map ( port_association_list ) ] ;

constant_declaration ::= [93LRM 4.3.1.1]                                      [25]
  constant identifier { , identifier } : subtype_indication [ := expression ] ;
```

```
constraint ::= range_constraint | index_constraint [93LRM 4.2]                  [26]

decimal_literal ::= [93LRM 13.4.1]                                               [27]
  integer [ . integer ] [ E [ + ] integer | E – integer ]

design_file ::= [93LRM 11.1]                                                     [28]
  { library_clause | use_clause } library_unit
  { { library_clause | use_clause } library_unit }

disconnection_specification ::= [93LRM 5.3]                                      [29]
  disconnect guarded_signal_list : type_mark after time_expression ;

discrete_range ::= [93LRM 3.2.1] discrete_subtype_indication | range             [30]

entity_class ::= [93LRM 5.1]                                                     [31]
  entity
  | architecture     | configuration    | procedure      | function
  | package          | type             | subtype        | constant
  | signal           | variable         | component      | label
  | literal          | units            | group          | file

entity_declaration ::= [93LRM 1.1]                                               [32]
  entity identifier is
    [ generic ( formal_generic_interface_list ) ; ]
    [ port ( formal_port_interface_list ) ; ]
    { subprogram_specification ;       | subprogram_body
    | subtype_declaration              | constant_declaration
    | signal_declaration               | file_declaration
    | alias_declaration                | attribute_declaration
    | attribute_specification          | type_declaration
    | disconnection_specification      | use_clause
    | shared variable_declaration
    | group_template_declaration       | group_declaration }
  [ begin
    { [ label : ] [ postponed ] assertion ;
    | [ label : ] [ postponed ] passive_procedure_call ;
    | passive_process_statement } ]
  end [ entity ] [entity_identifier ] ;

entity_name_list ::= [93LRM 5.1]                                                 [33]
   identifier | " { graphic_character } " | ' graphic_character ' [ signature ]
  { , identifier | " { graphic_character } " | ' graphic_character ' [ signature ] }
  | others
  | all

enumeration_literal ::= [93LRM 3.1.1] identifier | ' graphic_character '         [34]

exit_statement ::= [93LRM 8.11]                                                  [35]
[label:] exit [ loop_label ] [ when boolean_expression ] ;

expression ::= [93LRM 7.1]                                                       [36]
  relation { and relation }
```

| relation { **or** relation }
| relation { **xor** relation }
| relation [**nand** relation]
| relation [**nor** relation]
| relation { **xnor** relation }

factor ::= [93LRM 7.1] primary [** primary] | **abs** primary | **not** primary [37]

file_declaration ::= [93LRM 4.3.1.4] [38]
file identifier { , identifier } : subtype_indication
[[**open** *file_open_kind*_expression] **is** ~~{ **in** | **out** }~~ *string*_expression] ;

formal_part ::= [93LRM 4.3.2.2] [39]
*generic*_name | *port*_name | *parameter*_name
| *function*_name (*generic*_name | *port*_name | *parameter*_name)
| type_mark (*generic*_name | *port*_name | *parameter*_name)

function_call ::= [93LRM 7.3.3] *function*_name [(*parameter*_association_list)] [40]

generate_statement ::= [93LRM 9.7] [41]
*generate*_label:
for identifier **in**
discrete_subtype_indication | range
| **if** *boolean*_expression
generate
[{ block_declarative_item } **begin**]
{ concurrent_statement }
end generate [*generate*_label] ;

graphic_character ::= [93LRM 13.1] [42]
basic_graphic_character | LOWER_CASE_LETTER | OTHER_SPECIAL_CHARACTER

group_declaration ::= [93LRM 4.7] [43]
group identifier : *group_template*_name
(name | ' graphic_character '
{ , name | ' graphic_character ' }) ;

group_template_declaration ::= [93LRM 4.6] [44]
group identifier **is** (entity_class [<>] { , entity_class [<>] }) ;

identifier ::= [93LRM 13.3] [45]
LETTER { [UNDERLINE] LETTER | DIGIT }
| \ graphic_character { graphic_character } \

if_statement ::= [93LRM 8.7] [46]
[*if*_label :] **if** *boolean*_expression **then** { sequential_statement }
{ **elsif** *boolean*_expression **then** { sequential_statement } }
[**else** { sequential_statement }]
end if [*if*_label] ;

index_constraint ::= [93LRM 3.2.1] (discrete_range { , discrete_range }) [47]

integer ::= [§ 13.4.1] DIGIT { [UNDERLINE] DIGIT } [48]

```
interface_declaration ::= [93LRM 4.3.2]                                    [49]
  [constant] identifier { , identifier }
    : [ in ] subtype_indication [ := static_expression ]
  | [ signal ] identifier { , identifier }
    : [ in | out | inout | buffer | linkage ]
    subtype_indication [ bus ] [ := static_expression ]
  | [ variable ] identifier { , identifier}
    : [in | out | inout | buffer | linkage ] subtype_indication [ := static_expression]
  | file identifier { , identifier } : subtype_indication

interface_list ::= [93LRM 4.3.2.1] interface_declaration {; interface_declaration}      [50]

label ::=  identifier [93LRM 9.7]                                           [51]

library_clause ::= [93LRM 11.2] library identifier {, identifier} ;         [52]

library_unit ::= [93LRM 11.1]                                               [53]
  entity_declaration | configuration_declaration | package_declaration
  | architecture_body | package_body

literal ::= [93LRM 7.3.1]                                                   [54]
  decimal_literal          | based_literal      | physical_literal
  | enumeration_literal    | string_literal     | bit_string_literal | null

loop_statement ::= [93LRM 8.9]                                              [55]
  [ loop_label : ]
  [ while boolean_expression | for identifier in discrete_range ]
    loop
      { sequential_statement }
    end loop [ loop_label ] ;

name ::= [93LRM 6.1]                                                        [56]
  identifier
  | " { graphic_character } "
  | prefix.suffix
  | prefix ( expression { , expression } )
  | prefix ( discrete_range )
  | attribute_name

next_statement ::= [93LRM 8.10]                                             [57]
  [ label : ] next [ loop_label ] [ when boolean_expression ] ;

null_statement ::= [93LRM 8.13] [ label : ] null ;                          [58]

package_body ::= [93LRM 2.6]                                                [59]
  package body package_identifier is
  { subprogram_specification ; | subprogram_body | type_declaration
  | subtype_declaration | constant_declaration | file_declaration
  | alias_declaration | use_clause
  | shared_variable_declaration
  | group_template_declaration | group_declaration }
  end [ package body ] [ package_identifier ] ;
```

```
package_declaration ::= [93LRM 2.5]                                              [60]
  package identifier is
  { subprogram_specification ; | type_declaration | subtype_declaration
  | constant_declaration | signal_declaration | file_declaration
  | alias_declaration | component_declaration
  | attribute_declaration | attribute_specification
  | disconnection_specification | use_clause
  | shared variable declaration
  | group_template_declaration | group_declaration }
  end [ package ] [ package_identifier ] ;

physical_literal ::= [93LRM 3.1.3] [ decimal_literal | based_literal ] unit_name  [61]

physical_type_definition ::= [93LRM 3.1.3]                                       [62]
  range_constraint
    units identifier ;
    { identifier = physical_literal ; }
    end units [ physical_type_identifier ]

prefix ::= [93LRM 6.1] name | function_call                                      [63]

primary ::= [93LRM 7.1]                                                          [64]
  name | literal | aggregate | function_call
  | type_mark ' ( expression ) | type_mark ' aggregate | type_mark ( expression )
  | ( expression )
  | new subtype_indication | new type_mark ' (expression) | new type_mark ' aggregate

procedure_call ::= [93LRM 8.6] procedure_name [ ( parameter_association_list ) ]  [65]

process_statement ::= [93LRM 9.2 ]                                               [66]
  [ process_label : ]
  [ postponed ] process [ ( signal_name { , signal_name } ) ] [ is ]
    { subprogram_specification; | subprogram_body
    | type_declaration
    | subtype_declaration | constant_declaration
    | variable_declaration
    | file_declaration | alias_declaration | attribute_declaration
    | attribute_specification | use_clause
    | group_template_declaration | group_declaration }
  begin
    { sequential_statement }
  end [ postponed ] process [ process_label ] ;

range ::= [93LRM 3.1]                                                            [67]
  range_attribute_name
  | simple_expression to | downto simple_expression

range_constraint ::= [93LRM 3.1] range range                                     [68]

record_type_definition ::= [93LRM 3.2.2]                                         [69]
  record
    identifier {, identifier} : subtype_indication ;
```

```
      { identifier {, identifier} : subtype_indication ; }
   end record [ record_type_identifier ]

relation ::= [93LRM 7.1]                                                    [70]
   simple_expression [ sll | srl | sla | sra | rol | ror simple_expression ]
   [ = | /= | < | <= | > | >=
   simple_expression [ sll | srl | sla | sra | rol | ror simple_expression ] ]

report_statement ::= [93LRM 8.3]                                            [71]
   [ label : ] report expression [ severity expression ] ;

return_statement ::= [93LRM 8.12] [ label : ] return [ expression ] ;       [72]

selected_signal_assignment ::= [93LRM 9.5.2]                                [73]
   with expression select
      name | aggregate <= [ guarded ]
   [ transport | [ reject time_expression ] inertial ]
      waveform when choice { | choice }
         { , waveform when choice { | choice } } ;

sequential_statement ::= [93LRM 8]                                          [74]
   wait_statement
   | [ label : ] assertion ;
   | report_statement
   | signal_assignment_statement
   | variable_assignment_statement
   | [ label : ] procedure_call ;
   | if_statement
   | case_statement
   | loop_statement
   | next_statement
   | exit_statement
   | return_statement
   | null_statement

signal_assignment_statement ::= [93LRM 8.4]                                 [75]
   [ label : ] name | aggregate <=
   [ transport | [ reject time_expression ] inertial ] waveform ;

signal_declaration ::= [93LRM 4.3.1.2]                                      [76]
   signal identifier {, identifier } : subtype_indication
      [register | bus] [ := expression] ;

signal_list ::= [93LRM 5.3] signal_name { , signal_name } | others | all    [77]

signature ::= [93LRM 2.3.2]                                                 [78]
   [ [ type_mark { , type_mark } ] [ return type_mark ] ]

simple_expression ::= [93LRM 7.1] [ + | - ] term { + | - | & term }         [79]

string_literal ::= [93LRM 13.6] " { graphic_character } "                   [80]
```

```
subprogram_body ::= [93LRM 2.2]                                              [81]
  subprogram_specification is
  { subprogram_specification ;
  | subprogram_body
  | type_declaration
  | subtype_declaration
  | constant_declaration
  | variable_declaration
  | file_declaration
  | alias_declaration
  | attribute_declaration
  | attribute_specification
  | use_clause
  | group_template_declaration
  | group_declaration }
  begin
    { sequential_statement }
  end [ procedure | function ]
    [ identifier | " { graphic_character } " ] ;

subprogram_specification ::= [93LRM 2.1]                                     [82]
  procedure identifier | " { graphic_character } "
    [ ( parameter_interface_list ) ]
  | [ pure | impure ] function identifier | " { graphic_character } "
    [ ( parameter_interface_list ) ]
  return type_mark

subtype_declaration ::= [93LRM 4.2]                                          [83]
  subtype identifier is
  [ resolution_function_name ] type_mark [ constraint ] ;

subtype_indication ::= [93LRM 4.2]                                           [84]
  [ resolution_function_name ] type_mark [ constraint ]

suffix ::= [93LRM 6.3]                                                       [85]
  identifier
  | ' graphic_character '
  | " { graphic_character } "
  | all

term ::= [93LRM 7.1] factor { * | / | mod | rem factor }                     [86]

type_declaration ::= [93LRM 4.1]                                             [87]
  type identifier ;
  | type identifier is
    ( identifier | ' graphic_character '
      { , identifier | ' graphic_character ' } ) ;
    | range_constraint ;
    | physical_type_definition ;
    | record_type_definition ;
    | access subtype_indication ;
    | file of type_mark ;
    | array index_constraint of element_subtype_indication ;
    | array ( type_mark range <> { , type_mark range <> } ) of
      element_subtype_indication ;

type_mark ::= [93LRM 4.2] type_name | subtype_name                          [88]

use_clause ::= [93LRM 10.4] use prefix.suffix {, prefix.suffix} ;           [89]

variable_assignment_statement ::= [93LRM 8.5]                               [90]
  [ label : ] name | aggregate := expression ;

variable_declaration ::= [93LRM 4.3.1.3]                                    [91]
  [ shared ] variable identifier {, identifier} : subtype_indication
  [ := expression ] ;

wait_statement ::= [93LRM 8.1]                                              [92]
  [ label : ] wait
    [ on signal_name { , signal_name } ]
    [ until boolean_expression ]
    [ for time_expression ] ;

waveform ::= [93LRM 8.4] waveform_element { , waveform_element } | unaffected   [93]

waveform_element ::= [93LRM 8.4.1]                                          [94]
  value_expression [ after time_expression ] | null [ after time_expression ]
```

A.3 BNF Index

Table A.1 is an index to the VHDL BNF productions. For example, to find the legal positions for a process statement you would locate production rules 21 and 32 opposite `process_statement` in Table A.1. These rule numbers correspond to the productions for `concurrent_statement` (21) and `entity_declaration` (32). Next, turning to rule 32 for `entity_declaration` on page 967, you will find that only a *passive*`_process_statement` is allowed in an entity declaration. Table A.2 is a list of VHDL keywords and an index to rules that reference a keyword.

A.4 Bibliography

The book by Ashenden [1995] covers VHDL-93 in detail. Other books on VHDL include: Coelho [1989]; Lipsett, Schaefer, and Ussery [1989]; Armstrong [1989]; Augustin et al. [1991]; Perry [1991]; Mazor and Langstraat [1992]; three books by Bhasker [1992, 1995, 1996]; Armstrong and Gray [1993]; Baker [1993]; Navabi [1993]; Ott and Wilderotter [1994]; Airiau, Bergé, and Olive [1994]; two books by Cohen [1995, 1997]; Pick [1996]; Jerraya et al. [1997]; Pellerin and Taylor [1997]; Sjoholm and Lindh [1997]; and Chang [1997]. Of these, the book by Armstrong and Gray and Perry's books (two editions) are easy-to-read

TABLE A.1 Index to VHDL BNF rules (list of rules that reference a rule).

1	actual_part	6
2	aggregate	22, 64, 73, 75, 90
3	alias_declaration	14, 32, 59, 60, 66, 81
4	architecture_body	53
5	assertion	21, 32, 74
6	association_list	15, 18, 20, 24, 40, 65
7	attribute_declaration	14, 32, 60, 66, 81
8	attribute_name	56, 67
9	attribute_specification	14, 23, 32, 60, 66, 81
10	based_literal	54, 61
11	basic_graphic_character	42
12	bit_string_literal	54
13	block_configuration	13, 18, 23
14	block_declarative_item	4, 15, 41
15	block_statement	21
16	case_statement	74
17	choice	2, 16, 74
18	component_configuration	13
19	component_declaration	14, 60
20	component_instantiation_statement	21
21	concurrent_statement	4, 15, 21, 41
22	conditional_signal_assignment	21
23	configuration_declaration	53
24	configuration_specification	14
25	constant_declaration	14, 32, 59, 60, 66, 81
26	constraint	83, 84
27	decimal_literal	54, 61
28	design_file	0
29	disconnection_specification	14, 32, 60
30	discrete_range	17, 47, 55, 56, 61
31	entity_class	9, 44
32	entity_declaration	53
33	entity_name_list	9
34	enumeration_literal	54
35	exit_statement	74
36	expression[3]	
37	factor	86
38	file_declaration	14, 32, 59, 60, 66, 81
39	formal_part	6
40	function_call	63, 64
41	generate_statement	21
42	graphic_character	3, 33, 34, 42, 43, 45, 56, 80, 81, 82, 85, 87
43	group_declaration	14, 23, 32, 59, 60, 66, 81
44	group_template_declaration	14, 32, 59, 60, 66, 81
45	identifier[5]	
46	if_statement	74
47	index_constraint	26, 87
48	integer	10, 27
49	interface_declaration	50
50	interface_list	15, 19, 32, 82
51	label[1]	
52	library_clause	28
53	library_unit	28
54	literal	64
55	loop_statement	74
56	name[2]	
57	next_statement	74
58	null_statement	74
59	package_body	53
60	package_declaration	53
61	physical_literal	54, 62
62	physical_type_definition	87
63	prefix	8, 13, 23, 56, 89
64	primary	37
65	procedure_call	21, 32, 74
66	process_statement	21, 32
67	range	13, 30, 41, 68, 87
68	range_constraint	26, 62, 87
69	record_type_definition	87
70	relation	36
71	report_statement	74
72	return_statement	74
73	selected_signal_assignment	21
74	sequential_statement	16, 46, 55, 66, 81
75	signal_assignment_statement	74
76	signal_declaration	14, 32, 60
77	signal_list	29
78	signature	3, 8, 33
79	simple_expression	17, 67, 70
80	string_literal	54
81	subprogram_body	14, 32, 59, 66, 81
82	subprogram_specification	14, 32, 59, 60, 66, 81
83	subtype_declaration	14, 32, 59, 60, 66, 81
84	subtype_indication[4]	
85	suffix	13, 23, 56, 89
86	term	79
87	type_declaration	14, 32, 59, 60, 66, 81
88	type_mark	1, 7, 29, 39, 64, 78, 82, 83, 84, 87
89	use_clause	14, 28, 32, 59, 60, 66, 81
90	variable_assignment_statement	74
91	variable_declaration	14, 32, 59, 60, 66, 81
92	wait_statement	74
93	waveform	22, 73, 75
94	waveform_element	93

[1] 13, 15, 16, 18, 20, 21, 24, 31, 32, 35, 41, 46, 55, 57, 66, 71, 72, 74, 75, 90, 92

[2] 1, 3, 4, 13, 18, 20, 22, 23, 24, 39, 40, 43, 61, 63, 64, 65, 66, 73, 75, 77, 83, 84, 88, 90, 92

[3] 1,2, 5, 8, 9, 13, 15, 16, 17, 22, 25, 29, 35, 39, 41, 46, 49, 55, 56, 57, 64, 67, 70, 71, 72, 73, 75, 76, 79, 90, 91, 92, 94

[4] 3, 13, 25, 30, 38, 41, 49, 64, 69, 76, 87, 91

[5] 3,4,7,8,9,17,18, 18, 20, 23, 24, 25, 32, 33, 34, 38, 41, 43, 44, 49, 51, 52, 55, 56, 59, 60, 61, 62, 69, 76, 81, 82, 83, 85, 87, 91

TABLE A.2 VHDL keywords and index (list of rules that reference a keyword).[1]

abs	37
access	87
after	29, 94
alias	3
all	18, 24, 77, 85
and	36
architecture	4, 31
array	87
assert	5
attribute	7, 9
begin	4, 15, 32, 41, 66, 81
block	15
body	59
buffer	49
bus	49, 76
case	16
component	19, 20, 31
configuration	18, 20, 23, 24, 31
constant	25, 31, 49
disconnect	29
downto	67
else	22, 46
elsif	46
end	4, 13, 15, 16, 18, 19, 23, 32, 41, 46, 55, 59, 60, 62, 66, 69, 81
entity	18, 20, 24, 31, 32
exit	35
file	31, 38, 87
for	13, 18, 24, 41, 55, 92
function	31, 81, 82
generate	41
generic	15, 18, 19, 20, 24, 32
group	31, 43, 44
guarded	22, 73
if	41, 46
impure	82
in[2]	41, 49, 55
inertial	22, 73, 75
inout	49
is	3, 4, 9, 15, 16, 19, 23, 32, 38, 44, 59, 60, 66, 81, 83, 87
label	31
library	52
linkage	49
literal	31
loop	55
map	15, 18, 20, 24
mod	86
nand	36
new	64
next	57
nor	36
not	37
null	54, 58, 94
of	4, 9, 23, 87
on	92
open	1, 18, 24, 38
or	36
others	17, 18, 24, 33, 77
out[3]	49
package	31, 59, 60
port	15, 18, 19, 20, 24, 32
postponed	21, 32, 66
procedure	31, 81, 82
process	66
pure	82
range	68
record	69
register	76
reject	22, 73, 75
rem	86
report	5, 71
return	72, 78, 82
rol	70
ror	70
select	73
severity	5, 71
signal	31, 49, 76
shared	91
sla	70
sll	70
sra	70
srl	70
subtype	31, 83
then	46
to	67
transport	22, 73, 75
type	31, 87
unaffected	93
units	31, 62
until	92
use	13, 18, 23, 24, 89
variable	31, 49, 91
wait	92
when	16, 22, 35, 57, 73
while	55
with	73
xnor	36
xor	36

[1] Underlines denote VHDL-93 keywords that are not VHDL-87 keywords.

[2] Excluding VHDL-87 file_declaration.

[3] Excluding VHDL-87 file_declaration.

introductions. The following books describe example VHDL models for ASICs: Leung and Shanblatt [1989], Skahill [1996], Smith [1996]. Edited books by Bergé et al. [1992, 1993]; Harr and Stanculescu [1991]; Mermet [1992]; and Schoen et al. [1991] contain papers on more advanced aspects of VHDL.

There are some issues and interpretations of VHDL that are covered in a separate IEEE document [IEEE 1076-1991]; also relevant are the IEEE standard logic system for VHDL [IEEE 1164-1993], the WAVES standard [IEEE 1029.1-1991], and the VITAL standard [IEEE 1076.4-1995]. The IEEE has produced a VHDL interactive tutorial on CD-ROM [IEEE 1164-1997]. Hanna et al. [1997] cover the IEEE WAVES standard.

Updates and extensions to VHDL are controlled by the IEEE working groups (WG). These include study and WGs on: Object Oriented VHDL, Open Modeling Forum, Simulation Control Language (SimCL), System Design & Description Language, VHDL Analog Extensions (PAR 1076.1), VHDL Math Package (PAR 1076.2), VHDL Synthesis Package (PAR 1076.3), Utility (PAR 1076.5), VHDL Shared Variables (PAR 1076; mod a), VHDL Analysis and Standards Group (VASG) Issues Screening and Analysis Committee (ISAC), VHDL Library, VHDL Parallel Simulation, and VHDL Test. Links to the activities of these groups, as well as an explanation of a Project Authorization Request (PAR) and the standards process, may be found at `http://ieee.org` and `http://stdsbbs.ieee.org`.

A.5 References

Page numbers in brackets after a reference indicate its location in the chapter body.

The current IEEE standards and material listed here are published by The Institute of Electrical and Electronics Engineers, Inc., 345 East 47th Street, New York, NY 10017 USA. Inside the United States, IEEE standards may be ordered at 1-800-678-4333. See also `http://ieee.org` and `http://stdsbbs.ieee.org`.

Airiau, R., J.-M. Bergé, and V. Olive. 1994. *Circuit Synthesis with VHDL.* Boston: Kluwer, 221 p. ISBN 0792394291. TK7885.7.A37. Introduction to VHDL aimed at ASIC designers.

Armstrong, J. R. 1989. *Chip-Level Modeling with VHDL.* Englewood Cliffs, NJ: Prentice-Hall, 148 p. ISBN 0131331906. TK7874.A75 1989.

Armstrong, J. R., and F. G. Gray. 1993. *Structured Logic Design with VHDL.* Englewood Cliffs, NJ: Prentice-Hall, 482 p. ISBN 0138552061. TK7885.7.A76. 20 pages of references.

Ashenden, P. J. 1995. *The Designer's Guide to VHDL.* San Francisco: Morgan Kaufmann, 688 p. ISBN 1-55860-270-4. TK7888.3.A863. A complete reference to VHDL from a system design perspective.

Augustin, L. M., et al. 1991. *Hardware Design and Simulation in VAL/VHDL.* Boston: Kluwer, 322 p. ISBN 0792390873. TK7885.7.H38. Two pages of references.

Baker, L. 1993. *VHDL Programming with Advanced Topics.* New York: Wiley, 365 p. ISBN 0471574643. TK7885.7.B35. Basic to intermediate level coverage of VHDL.

Bergé, J.-M., et al. (Eds.). 1992. *VHDL Designer's Reference.* Boston: Kluwer, 455 p. ISBN 0792317564. TK7885.7.V47. Two pages of references.

Bergé, J.-M., et al. (Eds.). 1993. *VHDL '92.* Boston: Kluwer, 214 p. ISBN 0792393562. TK7885.7.V46. Covers new constructs in VHDL-93.

Bhasker, J. 1992. *A VHDL Primer.* Englewood Cliffs, NJ: Prentice-Hall, 253 p. ISBN 013952987X. TK7885.7.B53. See also the revised edition, ISBN 0131814478, 1995. A basic introduction to VHDL.

Bhasker, J. 1995. *A Guide to VHDL Syntax: Based on the New IEEE Std 1076-1993.* Englewood Cliffs, NJ: Prentice-Hall, 268 p. ISBN 0133243516. TK7885.7.B52. Uses graphics to illustrate BNF syntax.

Bhasker, J. 1996. *A VHDL Synthesis Primer.* Allentown, PA: Star Galaxy, 238 p. ISBN 0965039102. TK7885.7.B534.

Chang, K. C. 1997. *Digital Design and Modeling with VHDL and Synthesis.* Los Alamitos, CA: IEEE Computer Society Press. ISBN 0818677163. TK7874.7.C47.

Coelho, D. R. 1989. *The VHDL Handbook.* Boston: Kluwer, 389 p. ISBN 0792390318. TK7874.C6. A description of VHDL models, including details of models for simple combinational and sequential logic devices and memory. Two pages of references.

Cohen, B. 1995. *VHDL Coding Styles and Methodologies.* Boston: Kluwer, 365 p. ISBN 0792395980. TK7885.7.C65.

Cohen, B. 1997. *VHDL Answers to Frequently Asked Questions.* Boston: Kluwer, 291 p. ISBN 0792397916. TK7885.7.C64.

Hanna, J. P., et al. 1997. *Using WAVES and VHDL for Effective Design and Testing.* Boston: Kluwer, 304 p. ISBN 0792397991. TK7874.7.U87.

Harr, R. E., and A. G. Stanculescu (Eds.). 1991. *Applications of VHDL to Circuit Design.* Boston: Kluwer, 232 p ISBN 0792391535. TK7867.A64.

IEEE 1076-1987. *IEEE Standard VHDL Language Reference Manual.* This version of the VHDL LRM is replaced by VHDL-93; however, some systems (and some IEEE Standards, notably VITAL) are based on VHDL-87. For instructions on how to obtain obsolete IEEE standards (known as archive standards), see `http://stdsbbs.ieee.org`. [cited on p. 961]

IEEE 1076-1991. *1076 Interpretations, 1991 IEEE Standards Interpretations: IEEE Std 1076-1987, IEEE Standard VHDL Language Reference Manual.* 208 p. ISBN 1-55937-181-1. TK7885.7.I58. IEEE Ref. SH14894-NYF.

IEEE 1029.1-1991. *IEEE Standard for Waveform and Vector Exchange (WAVES) (ANSI).* 96 p. ISBN 1-55937-195-1. IEEE Ref. SH15032-NYF.

IEEE 1164-1993. *IEEE Standard Multivalue Logic System for VHDL Model Interoperability (Std_logic_1164).* 24 p. ISBN 1-55937-299-0. IEEE Ref. SH16097-NYF.

IEEE 1076-1993. *IEEE Standard VHDL Language Reference Manual (ANSI).* 288 p. ISBN 1-55937-376-8. IEEE Ref. SH16840-NYF. [cited on p. 961]

IEEE 1076.4-1995. *IEEE Standard VITAL Application-Specific Integrated Circuit (ASIC) Modeling Specification.* 96 p. ISBN 1-55937-691-0. IEEE Ref. SH94382-NYF. Includes an MS-DOS diskette containing the ASCII code for the `VITAL_Timing` and `VITAL_Primitives` packages.

IEEE 1076-1997. *VHDL Interactive Tutorial.* This CD-ROM tutorial is available from IEEE by itself or with a printed version of the VHDL LRM. The CD-ROM is available in the following formats: IBM Windows (Windows 3.1 and Windows 95), Macintosh, Sun OS, and Sun Solaris.

Jerraya, A. A., et al. 1997. *Behavioral Synthesis and Component Reuse with VHDL.* Boston: Kluwer, 263 p. ISBN 0792398270. TK7874.75.B45. Eight pages of references.

Leung, S. S., and M. A. Shanblatt. 1989. *ASIC System Design with VHDL: A Paradigm.* Boston: Kluwer, 206 p. ISBN 0-7923-90932-6. TK7874.L396. Describes VHDL models for an ASIC intended for IKS (inverse kinematic solution) which converts cartesian space to the robot-joint space. Eight pages of references.

Lipsett, R., C. Schaefer, and C. Ussery. 1989. *VHDL: Hardware Description and Design.* Boston: Kluwer, 299 p. ISBN 079239030X. TK7887.5.L57. Intermediate guide to VHDL.

Mazor, S., and P. Langstraat. 1992. *A Guide to VHDL.* Boston: Kluwer. ISBN 0792392558. TK7885.7.M39. See also 2nd ed., ISBN 0792393872, 1993. Basic introduction to VHDL.

Mermet, J. (Ed.). 1992. *VHDL for Simulation, Synthesis, and Formal Proofs of Hardware.* Boston: Kluwer. ISBN 0792392531. TK7885.7.V48.

Navabi, Z. 1993. *VHDL: Analysis and Modeling of Digital Systems.* New York: McGraw-Hill, 375 p. ISBN 0070464723. TK7874.N36. Introduction to VHDL.

Ott, D. E., and T. J. Wilderotter. 1994. *A Designer's Guide to VHDL Synthesis.* Boston: Kluwer. ISBN 0792394720. TK7885.7.O89.

Pellerin, D., and D. Taylor. 1997. *VHDL Made Easy!* Upper Saddle River, NJ: Prentice-Hall, 419 p. ISBN 0136507638. TK7885.7.P46.

Perry, D. L. 1991. *VHDL.* New York: McGraw-Hill, 458 p. ISBN 0070494339. TK7885.7.P47. See also 2nd ed., ISBN 0070494347, 1994. Good introduction to VHDL.

Pick, J. 1996. *VHDL Techniques, Experiments, and Caveats.* New York: McGraw-Hill, 382 p. ISBN 0070499063. TK7885.7.P53. A series of intermediate to advanced examples that illustrate mistakes in VHDL.

Schoen, J. M., et al. (Eds.). 1991. *Performance and Fault Modeling with VHDL.* Englewood Cliffs, NJ: Prentice-Hall, 406 p ISBN 0136588166. TK7888.4.P47.

Sjoholm, S., and L. Lindh. 1997. *VHDL for Designers.* Englewood Cliffs, NJ: Prentice-Hall. ISBN 0134734149. TK7885.7.S54.

Skahill, K. 1996. *VHDL for Programmable Logic.* Menlo Park, CA: Addison-Wesley, 593 p. ISBN 0-201-89573-0. TK7885.7.S55. Covers VHDL design for PLDs using Cypress Warp.

Smith, D. J. 1996. *HDL Chip Design: A Practical Guide for Designing, Synthesizing, and Simulating ASICs and FPGAs using VHDL or Verilog.* Madison, AL: Doone Publications, 448 p. ISBN 0965193438. TK7874.6.S62.

VERILOG HDL RESOURCES

B

The definitive reference for the Verilog HDL is IEEE Std 1364-1995. This standard is known as the **IEEE Verilog® HDL language reference manual** (**LRM**) and the 1995 version is referred to here as the 95 LRM [IEEE 1364-1995].[1] Verilog is a registered trademark of Cadence Design Systems and Verilog-XL is a commercial simulator.

B.1 Explanation of the Verilog HDL BNF

Annex A of the Verilog HDL LRM describes syntax using the BNF (Backus–Naur form). The Verilog HDL BNF is slightly different from that employed in the VHDL LRM (see Appendix A, Section A.1, "BNF"). The BNF syntax in the Verilog LRM is **normative**, which means that the syntax is part of the definition of the language (and the complete BNF description is contained in an **annex**). The BNF syntax in the VHDL LRM is **informative**, which means the BNF is not part of the standard defining the language (and the complete BNF description is contained in an **appendix**). The following items summarize the Verilog HDL BNF syntax:

- `name` (in lowercase) is a **syntax construct item** (**term**, or **syntactic category**) defined by other syntax construct token items (**items**, **parts**, or **tokens**) or by lexical token items.
- `NAME` (in uppercase) is a **lexical token item**, the leaves in a tree of definitions.
- `[ name ]` is an **optional item**.
- `{ name }` is one or more items.
- The symbol **::=** gives a syntax definition (**definition**, **rule**, **construct**, or **production**) for an item (i.e., the symbol **::=** means *is equivalent to*).
- | introduces an alternative syntax definition (i.e., the symbol | means *or*).
- Braces and brackets, { } and [ ], that are required by the syntax are set in bold, **{ } [ ]**, in the 95 LRM, but are difficult to distinguish from the plain versions. Here they are set in outline, like this: { } [ ].

[1] IEEE Std 1364-1995, Copyright © 1995. IEEE. All rights reserved. The Verilog HDL syntax section in this appendix is reprinted from the IEEE copyright material with permission.

- The vertical bar, |, that represents an alternative definition is set in bold, **|**, in the 95 LRM, but is difficult to distinguish from the plain version. Here it is set in outline, like this: |.
- All other characters that are set in bold (as they are in Annex A of the 95 LRM) are literals required by the syntax (for example, a plus sign '**+**').
- Italic prefixes, for example, *msb*_constant_expression, are comments.
- Keywords are printed in **bold**, as they are in Annex A of the 95 LRM.
- Definitions here are in alphabetical order (Annex A of the 95 LRM groups definitions by function). The highest-level definition is source_text; this is where you start. The lowest-level items are in uppercase; these are where you end.
- The BNF is reproduced *exactly* as it appears in Annex A of the 95 LRM. Footnotes explain a number of typographical issues.
- References in brackets immediately following the ::= symbol form backward-pointing links to the constructs that reference a particular item. Thus, for example, always_construct ::= [94] indicates that construct number 94 (module_item) references the item always_construct (see also Table B.1 on p. 995, which collects all these links together, and Table B.2 on p. 996, which is a keyword index).
- References in brackets following the construct links refer to the 95 LRM. Thus, for example, always_construct ::= [94] [95LRM 9.9.2], indicates that section 9.9.2 of the 95 LRM contains the definition for always_construct.

B.2 Verilog HDL Syntax

```
always_construct ::= [94] [95LRM 9.9.2] always statement                      [1]

binary_base ::= [4] [95LRM 2.5.1] 'b | 'B                                     [2]

binary_digit ::= [4] [95LRM 2.5.1] x | X | z | Z | 0 | 1                      [3]

binary_number ::= [114] [95LRM 2.5.1]                                         [4]
  [ size ] binary_base binary_digit { _ | binary_digit }

binary_operator ::= [19, 52] [95LRM 4.1.2]                                    [5]
  +  |  -  |  *  |  /  |  %  |  ==  |  !=  |  ===  |  !==  |  &&  |  ||
  |  <  |  <=  |  >  |  >=  |  &  |  |  |  ^  |  ^~  |  ~^  |  >>  |  <<

blocking assignment² ::= [181] [95LRM 9.2.1]                                  [6]
  reg_lvalue = [ delay_or_event_control ] expression

block_item_declaration ::= [57, 133, 166, 190] [95LRM 9.8.1]                  [7]
   parameter_declaration | reg_declaration
```

[2] The term blocking(space)assignment is referenced as blocking(underscore)assignment.

The BNF syntax on this page is from IEEE Std 1364-1995, Copyright © 1995. IEEE. All rights reserved.

```
  | integer_declaration | real_declaration | time_declaration
  | realtime_declaration | event_declaration

case_item ::= [9] [95LRM 9.5] expression { , expression } : statement_or_null       [8]
  | default [ : ] statement_or_null

case_statement ::= [181] [95LRM 9.5]                                                [9]
  | case ( expression ) case_item { case_item } endcase
  | casez ( expression ) case_item { case_item } endcase
  | casex ( expression ) case_item { case_item } endcase

charge_strength ::= [107] [95LRM 3.4.1] ( small ) | ( medium ) | ( large )         [10]

cmos_switchtype ::= [58] [95LRM 7.7] cmos | rcmos                                  [11]

cmos_switch_instance ::= [58] [95LRM 7.1] [ name_of_gate_instance ]                [12]
  ( output_terminal , input_terminal ,
    ncontrol_terminal , pcontrol_terminal )

combinational_body ::= [198] [95LRM 8.1.4]                                         [13]
  table combinational_entry { combinational_entry } endtable

combinational_entry ::= [13] [95LRM 8.1.4] level_input_list : output_symbol ;      [14]

comment ::= [—] [95LRM 2.3] short_comment | long_comment                           [15]

comment_text ::= [88, 168] [95LRM 2.3] { Any_ASCII_character }                     [16]

concatenation ::= [21, 109, 146, 161] [95LRM 4.1.14] { expression { , expression } }   [17]

conditional_statement ::= [181] [95LRM 9.4]                                        [18]
  | if ( expression ) statement_or_null [ else statement_or_null ]

constant_expression ::= [see Table B.1] [95LRM 4.1] constant_primary               [19]
  | unary_operator constant_primary
  | constant_expression binary_operator constant_expression
  | constant_expression ? constant_expression : constant_expression
  | string

constant_mintypmax_expression ::= [34, 74, 139] [95LRM 4.3] constant_expression    [20]
  | constant_expression : constant_expression : constant_expression

constant_primary ::= [19] [95LRM 4.1] number | parameter_identifier                [21]
  | constant_concatenation | constant_multiple_concatenation

continuous_assign ::= [94] [95LRM 6.1]                                             [22]
   assign [ drive_strength ] [ delay3 ] list_of_net_assignments ;

controlled_timing_check_event ::= [189] [95LRM 14.5.11]                            [23]
  timing_check_event_control
    specify_terminal_descriptor [ &&& timing_check_condition ]

current_state ::= [165] [95LRM 8.1] level_symbol                                   [24]
```

data_source_expression ::= [53, 127] [95LRM 13.3.3] expression [25]

decimal_base ::= [28] [95LRM 2.5.1] **'d** | **'D** [26]

decimal_digit ::= [206] [95LRM 2.5.1] **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **0** [27]

decimal_number ::= [114] [95LRM 2.5.1] [sign] unsigned_number [28]
| [size] decimal_base unsigned_number

delay2 ::= [58, 202] [95LRM 7.1, 7.15] [29]
delay_value | **#** **(** delay_value [**,** delay_value] **)**

delay3 ::= [22, 58, 107] [95LRM 7.1, 7.15] [30]
delay_value | **#** **(** delay_value [**,** delay_value [**,** delay_value]] **)**

delay_control ::= [32] [95LRM 9.7, 9.7.1] [31]
delay_value | **#** **(** mintypmax_expression **)**

delay_or_event_control ::= [6, 112, 148] [95LRM 9.7] [32]
delay_control | event_control | **repeat** **(** expression **)** event_control

delay_value ::= [29, 30, 31] [95LRM 7.1.3, 7.15] [33]
unsigned_number | parameter_identifier | constant_mintypmax_expression

description ::= [173] [95LRM 8.1, 12.1] module_declaration | udp_declaration [34]

disable_statement[3] ::= [181] [95LRM 11] [35]
| **disable** *task*_identifier **;** | **disable** *block*_identifier **;**

drive_strength ::= [22, 58, 107, 202] [95LRM 3.2.1, 3.4.2, 6.1.4] [36]
(strength0 **,** strength1 **)** | **(** strength1 **,** strength0 **)**
| **(** strength0 **,** **highz1** **)** | **(** strength1 **,** **highz0** **)**
| **(** **highz1** **,** strength0 **)** | **(** **highz0** **,** strength1 **)**

edge_control_specifier[4] ::= [196] [95LRM 14.5.9] **edge** [37]
[edge_descriptor [**,** edge_descriptor] **]**

edge_descriptor ::= [37] [95LRM 14.5.9] **01** | **10** | **0x** | **x1** | **1x** | **x0** [38]

edge_identifier ::= [53, 127] [95LRM 14.5.9] **posedge** | **negedge** [39]

edge_indicator ::= [41] [95LRM 8.1, 8.1.6, 8.4] [40]
(level_symbol level_symbol **)** | edge_symbol

edge_input_list ::= [167] [95LRM 8.1, 8.1.6, 8.4] [41]
{ level_symbol } edge_indicator { level_symbol }

edge_sensitive_path_declaration ::= [138, 183] [95LRM 13.3.3] [42]
parallel_edge_sensitive_path_description = path_delay_value
| full_edge_sensitive_path_description = path_delay_value

[3] The construct for disable_statement has a leading vertical bar, |, in Annex A of the 95 LRM.

[4] The outer brackets in term edge_control_specifier are lexical elements; the inner brackets are an optional item.

```
edge_symbol ::= [40] [95LRM 8.1.6] r | R | f | F | p | P | n | N | *            [43]

enable_gatetype ::= [58] [95LRM 7.1] bufif0 | bufif1 | notif0 | notif1          [44]

enable_gate_instance ::= [58] [95LRM 7.1] [ name_of_gate_instance ]              [45]
  ( output_terminal , input_terminal , enable_terminal )

enable_terminal ::= [ 45, 98, 134] [95LRM 7.1] scalar_expression                [46]

escaped_identifier ::= [63] [95LRM 2.7.1]                                        [47]
  \ {Any_ASCII_character_except_white_space} white_space

event_control ::= [32] [95LRM 9.7] @ event_identifier | @ ( event_expression )   [48]

event_declaration ::= [7, 95] [95LRM 9.7.3]                                      [49]
  event event_identifier { , event_identifier } ;

event_expression ::= [48, 50] [95LRM 9.7] expression | event_identifier          [50]
  | posedge expression | negedge expression
  | event_expression or event_expression

event_trigger5 ::= [181] [95LRM 9.7.3] | -> event_identifier ;                   [51]

expression ::= [see Table B.1] [95LRM 4] primary | unary_operator primary       [52]
  | expression binary_operator expression | expression ? expression : expression
  | string

full_edge_sensitive_path_description6 ::= [42] [95LRM 13.3.2]                    [53]
   ( [ edge_identifier ] list_of_path_inputs *>
     list_of_path_outputs
      [ polarity_operator ] : data_source_expression ) )

full_path_description ::= [171] [95LRM 13.3.2, 13.3.5]                           [54]
   ( list_of_path_inputs [ polarity_operator ] *> list_of_path_outputs )

function_call ::= [146] [95LRM 10.3.3]                                           [55]
  function_identifier ( expression { , expression} )
  | name_of_system_function [ ( expression { , expression} ) ]

function_declaration ::= [95] [95LRM 10.3.1]                                     [56]
   function [ range_or_type ] function_identifier ;
   function_item_declaration { function_item_declaration }
   statement
   endfunction

function_item_declaration ::= [56] [95LRM 10.3.1]                                [57]
  block_item_declaration | input_declaration
```

[5] The construct for event_trigger has a leading vertical bar, |, in Annex A of the 95 LRM.

[6] The term full_edge_sensitive_path_description contains an unmatched right parenthesis in Annex A and Section 13.3.3 of the 95 LRM. The examples in Section 13.3.3 of the 95 LRM do not include the final trailing right parenthesis.

```
gate_instantiation[7] ::= [94] [95LRM 7.1] n_input_gatetype [ drive_strength ]         [58]
  [delay2] n_input_gate_instance { , n_input_gate_instance } ;
  | n_output_gatetype [ drive_strength ] [ delay2 ]
    n_output_gate_instance { , n_output_gate_instance } ;
  | enable_gatetype [ drive_strength ] [ delay3 ]
    enable_gate_instance { , enable_gate_instance} ;
  | mos_switchtype [ delay3 ]
    mos_switch_instance { , mos_switch_instance } ;
  | pass_switchtype pass_switch_instance { , pass_switch_instance } ;
  | pass_en_switchtype [ delay3 ]
    pass_en_switch_instance { , pass_en_switch_instance } ;
  | cmos_switchtype [ delay3 ]
    cmos_switch_instance { , cmos_switch_instance } ;
  | pullup [ pullup_strength ]
    pull_gate_instance { , pull_gate_instance } ;
  | pulldown [ pulldown_strength ]
    pull_gate_instance { , pull_gate_instance } ;

hex_base ::= [61] [95LRM 2.5.1] 'h | 'H                                                 [59]

hex_digit ::= [61] [95LRM 2.5.1]                                                        [60]
  x | X | z | Z | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
  | a | b | c | d | e | f | A | B | C | D | E | F

hex_number ::= [114] [95LRM 2.5.1] [ size ] hex_base hex_digit { _ | hex_digit }        [61]

identifier ::= [see Table B.1] [95LRM 2.7] IDENTIFIER [ { . IDENTIFIER } ]              [62]
  The period in identifier may not be preceded or followed by a space.

IDENTIFIER ::= [62] [95LRM 2.7] simple_identifier | escaped_identifier                  [63]

initial_construct ::= [94] [95LRM 9.9.1] initial statement                             [64]

init_val ::= [200] [95LRM 8.1, 8.5]                                                     [65]
  1'b0 | 1'b1 | 1'bx | 1'bX | 1'B0 | 1'B1 | 1'Bx | 1'BX | 1 | 0

inout_declaration ::= [95, 190] [95LRM 12.3.2]                                          [66]
  inout [ range ] list_of_port_identifiers ;

inout_terminal ::= [134, 137] [95LRM 7.1]                                               [67]
  terminal_identifier | terminal_identifier ⟦ constant_expression ⟧

input_declaration ::= [57, 95, 203] [95LRM 12.3.2]                                      [68]
  input [ range ] list_of_port_identifiers ;

input_identifier ::= [175] [95LRM 13.3.2]                                               [69]
  input_port_identifier | inout_port_identifier

input_terminal ::= [12, 45, 98, 116, 118] [95LRM 7.1] scalar_expression                 [70]
```

[7] The term pass_en_switch_instance is defined as pass_enable_switch_instance.

integer_declaration ::= [7, 95] [95LRM 3.9] **integer** list_of_register_identifiers ; [71]

level_input_list ::= [14, 167] [95LRM 8.1, 8.1.6] level_symbol { level_symbol } [72]

level_symbol ::= [24, 40, 41, 72] [95LRM 8.1, 8.1.6] **0** | **1** | **x** | **X** | **?** | **b** | **B** [73]

limit_value ::= [152] [95LRM 13.7] constant_mintypmax_expression [74]

list_of_module_connections ::= [92] [95LRM 12.1.2, 12.3.3, 12.3.4] [75]
ordered_port_connection { , ordered_port_connection }
| named_port_connection { , named_port_connection }

list_of_net_assignments ::= [22] [95LRM 3.10] net_assignment { , net_assignment } [76]

list_of_net_decl_assignments ::= [107] [95LRM 3.2.1] [77]
net_decl_assignment { , net_decl_assignment }

list_of_net_identifiers ::= [107] [95LRM 2.7] *net*_identifier { , *net*_identifier } [78]

list_of_param_assignments ::= [129, 130] [95LRM 3.10] [79]
param_assignment { , param_assignment }

list_of_path_delay_expressions ::= [140] [95LRM 13.4] [80]
*t*_path_delay_expression
| *trise*_path_delay_expression, *tfall*_path_delay_expression
| *trise*_path_delay_expression, *tfall*_path_delay_expression, *tz*_path_delay_expression
| *t01*_path_delay_expression, *t10*_path_delay_expression, *t0z*_path_delay_expression,
*tz1*_path_delay_expression, *t1z*_path_delay_expression, *tz0*_path_delay_expression
| *t01*_path_delay_expression, *t10*_path_delay_expression, *t0z*_path_delay_expression,
*tz1*_path_delay_expression, *t1z*_path_delay_expression, *tz0*_path_delay_expression,
*t0x*_path_delay_expression, *tx1*_path_delay_expression, *t1x*_path_delay_expression,
*tx0*_path_delay_expression, *txz*_path_delay_expression, *tzx*_path_delay_expression

list_of_path_inputs ::= [53, 54] [95LRM 13.3.2] [81]
specify_input_terminal_descriptor { , specify_input_terminal_descriptor }

list_of_path_outputs ::= [53, 54] [95LRM 13.3.2] [82]
specify_output_terminal_descriptor { , specify_output_terminal_descriptor }

list_of_ports ::= [91] [95LRM 12] **(** port { , port } **)** [83]

list_of_port_identifiers ::= [66, 68, 123] [95LRM 12.3.2] [84]
*port*_identifier { , *port*_identifier }

list_of_real_identifiers ::= [155, 156] [95LRM 2.7] [85]
*real*_identifier { , *real*_identifier }

list_of_register_identifiers ::= [71, 160, 193] [95LRM 3.2.2] [86]
register_name { , register_name }

list_of_specparam_assignments ::= [180] [95LRM 13.2] [87]
specparam_assignment { , specparam_assignment }

long_comment ::= [15] [95LRM 2.3] /* comment_text */ [88]

```
loop_statement ::= [181] [95LRM 9.6]                                                   [89]
  | forever statement
  | repeat ( expression ) statement
  | while ( expression ) statement
  | for ( reg_assignment ; expression ; reg_assignment ) statement

mintypmax_expression ::= [31, 146] [95LRM 4.3] expression                              [90]
  | expression : expression : expression

module_declaration ::= [34] [95LRM 12.1] module_keyword module_identifier               [91]
  [ list_of_ports ] ; {module_item } endmodule

module_instance ::= [93] [95LRM 12.1, 12.1.2]                                          [92]
  name_of_instance ( [ list_of_module_connections ] )

module_instantiation ::= [94] [95LRM 12.1.2]                                           [93]
  module_identifier [ parameter_value_assignment ]
  module_instance { , module_instance } ;

module_item ::= [91] [95LRM 12.1]                                                      [94]
  module_item_declaration | parameter_override
  | continuous_assign | gate_instantiation | udp_instantiation
  | module_instantiation | specify_block | initial_construct
  | always_construct

module_item_declaration ::= [94] [95LRM 12.1]                                          [95]
  parameter_declaration | input_declaration
  | output_declaration | inout_declaration | net_declaration
  | reg_declaration | integer_declaration | real_declaration
  | time_declaration | realtime_declaration | event_declaration
  | task_declaration | function_declaration

module_keyword ::= [91] [95LRM 12.1] module | macromodule                              [96]

mos_switchtype ::= [58] [95LRM 7.1, 7.5] nmos | pmos | rnmos | rpmos                   [97]

mos_switch_instance ::= [58] [95LRM 7.1]                                               [98]
  [ name_of_gate_instance ] ( output_terminal , input_terminal , enable_terminal )

multiple_concatenation[8] ::= [21, 146] [95LRM 4.1.14]                                 [99]
  { expression { expression { , expression } } }

named_port_connection ::= [75] [95LRM 12.1.2, 12.3.4]                                 [100]
  . port_identifier ( [ expression ] )

name_of_gate_instance ::= [12, 45, 98, 116, 118, 134, 137, 151] [95LRM 7.1]           [101]
  gate_instance_identifier [ range ]
```

[8] The two sets of outer braces (four) in the term multiple_concatenation are lexical elements; the inner braces (two) indicate an optional item.

```
name_of_instance ::= [92] [95LRM 12.1.2] module_instance_identifier [ range ]            [102]

name_of_system_function ::= [55] [95LRM 14] $identifier                                   [103]
  Note: the $ in name_of_system_function may not be followed by a space.

name_of_udp_instance ::= [201] [95LRM 8.6] udp_instance_identifier [ range ]             [104]

ncontrol_terminal ::= [12] [95LRM 7.1] scalar_expression                                  [105]

net_assignment ::= [76, 147] [95LRM 6.1, 9.3] net_lvalue = expression                     [106]

net_declaration ::= [95] [95LRM 3.2.1]                                                    [107]
     net_type [ vectored | scalared ] [ range ] [ delay3 ] list_of_net_identifiers ;
   | trireg [ vectored | scalared ]
     [ charge_strength ] [ range ] [ delay3 ] list_of_net_identifiers ;
   | net_type [ vectored | scalared ]
     [drive_strength] [range] [delay3] list_of_net_decl_assignments ;

net_decl_assignment ::= [77] [95LRM 3.2.1] net_identifier = expression                    [108]

net_lvalue ::= [106, 147] [95LRM 6.1]                                                     [109]
  net_identifier | net_identifier [ expression ]
  | net_identifier [ msb_constant_expression : lsb_constant_expression ]
  | net_concatenation

net_type ::= [107] [95LRM 3.2.1]                                                          [110]
  wire | tri | tri1 | supply0 | wand | triand | tri0 | supply1 | wor | trior

next_state ::= [165] [95LRM 8.1, 8.1.6] output_symbol | -                                 [111]

non-blocking assignment[9] ::= [181] [95LRM 9.2.2]                                        [112]
  reg_lvalue <= [ delay_or_event_control ] expression

notify_register ::= [189] [95LRM 14.5.10] register_identifier                             [113]

number ::= [21, 146] [95LRM 2.5]                                                          [114]
  decimal_number | octal_number | binary_number | hex_number | real_number

n_input_gatetype ::= [58] [95LRM 7.1] and | nand | or | nor | xor | xnor                  [115]

n_input_gate_instance ::= [58] [95LRM 7.1]                                                [116]
    [ name_of_gate_instance ] ( output_terminal , input_terminal { , input_terminal } )

n_output_gatetype ::= [58] [95LRM 7.1] buf | not                                          [117]

n_output_gate_instance ::= [58] [95LRM 7.1]                                               [118]
  [ name_of_gate_instance ] ( output_terminal { , output_terminal } , input_terminal )

octal_base ::= [121] [95LRM 2.5.1] 'o | 'O                                                [119]

octal_digit ::= [121] [95LRM 2.5.1]                                                       [120]
  x | X | z | Z | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
```

[9] The term, non(hyphen)blocking(space)assignment, is referenced as non(underscore)blocking(space)assignment.

The BNF syntax on this page is from IEEE Std 1364-1995, Copyright © 1995. IEEE. All rights reserved.

```
octal_number ::= [114] [95LRM 2.5.1]                                                [121]
  [ size ] octal_base octal_digit { _ | octal_digit}

ordered_port_connection ::= [75] [95LRM 12.1.2, 12.3.3] [ expression ]              [122]

output_declaration ::= [95, 190, 203] [95LRM 12.3.2]                                [123]
  output [ range ] list_of_port_identifiers ;

output_identifier ::= [177] [95LRM 13.3.2]                                          [124]
  output_port_identifier | inout_port_identifier

output_symbol ::= [14, 111] [95LRM 8.1, 8.1.6] 0 | 1 | x | X                        [125]

output_terminal ::= [12, 45, 98, 116, 118, 151] [95LRM 7.1]                         [126]
  terminal_identifier | terminal_identifier [ constant_expression ]

parallel_edge_sensitive_path_description10 ::= [42] [95LRM 13.3.2]                  [127]
   ( [ edge_identifier ] specify_input_terminal_descriptor =>
   specify_output_terminal_descriptor
    [ polarity_operator ] : data_source_expression ) )

parallel_path_description ::= [171] [95LRM 13.3.2]                                  [128]
  ( specify_input_terminal_descriptor
    [ polarity_operator ] => specify_output_terminal_descriptor )

parameter_declaration ::= [7, 95] [95LRM 3.10]                                      [129]
  parameter list_of_param_assignments ;

parameter_override ::= [94] [95LRM 12.2] defparam list_of_param_assignments ;       [130]

parameter_value_assignment ::= [93] [95LRM 12.1.2]                                  [131]
  # ( expression { , expression } )

param_assignment ::= [79] [95LRM 3.10] parameter_identifier = constant_expression   [132]

par_block ::= [181] [95LRM 9.8.2]                                                   [133]
   fork [ : block_identifier { block_item_declaration } ] { statement } join

pass_enable_switch_instance11 ::= [58] [95LRM 7.1]                                   [134]
   [ name_of_gate_instance ] ( inout_terminal , inout_terminal , enable_terminal )

pass_en_switchtype ::= [58] [95LRM 7.1]                                             [135]
  tranif0 | tranif1 | rtranif1 | rtranif0

pass_switchtype ::= [58] [95LRM 7.1] tran | rtran                                   [136]

pass_switch_instance ::= [58] [95LRM 7.1]                                           [137]
  [ name_of_gate_instance ] ( inout_terminal , inout_terminal )
```

[10] The term `parallel_edge_sensitive_path_description` has an unmatched right parenthesis in Annex A and Section 13.3.3 of the 95 LRM. The examples in Section 13.3.3 do not include the final trailing right parenthesis.

[11] The term `pass_enable_switch_instance` is referenced as `pass_en_switch_instance`.

The BNF syntax on this page is from IEEE Std 1364-1995, Copyright © 1995. IEEE. All rights reserved.

path_declaration[12] ::= [176] [95LRM 13.3] simple_path_declaration ; [138]
| edge_sensitive_path_declaration ; | state-dependent_path_declaration ;

path_delay_expression ::= [80] [95LRM 13.4] constant_mintypmax_expression [139]

path_delay_value ::= [42, 171] [95LRM 13.4] [140]
list_of_path_delay_expressions | (list_of_path_delay_expressions)

pcontrol_terminal ::= [12] [95LRM 7.1] *scalar*_expression [141]

polarity_operator ::= [53, 54, 127, 128] [95LRM 13.3.2] + | - [142]

port ::= [83] [95LRM 12.3.1] [143]
[port_expression] | . *port*_identifier ([port_expression])

port_expression ::= [143] [95LRM 12.3.1] [144]
port_reference | ⦃ port_reference { , port_reference } ⦄

port_reference ::= [144] [95LRM 12.3.1] *port*_identifier [145]
| *port*_identifier ⟦ constant_expression ⟧
| *port*_identifier ⟦ *msb*_constant_expression : *lsb*_constant_expression ⟧

primary ::= [52] [95LRM 4] number | identifier | identifier ⟦ expression ⟧ [146]
| identifier ⟦ *msb*_constant_expression : *lsb*_constant_expression ⟧
| concatenation | multiple_concatenation | function_call
| (mintypmax_expression)

procedural_continuous_assignment[13] ::= [181] [95LRM 9.3] [147]
| **assign** reg_assignment ;
| **deassign** reg_lvalue ; | **force** reg_assignment ;
| **force** net_assignment ; | **release** reg_lvalue ;
| **release** net_lvalue ;

procedural_timing_control_statement ::= [181] [95LRM 9.7] [148]
delay_or_event_control statement_or_null

pulldown_strength ::= [58] [95LRM 7.1] (strength0 , strength1) [149]
| (strength1 , strength0) | (strength0)

pullup_strength ::= [58] [95LRM 7.1] (strength0 , strength1) [150]
| (strength1 , strength0) | (strength1)

pull_gate_instance ::= [58] [95LRM 7.1] [name_of_gate_instance] (output_terminal) [151]

pulse_control_specparam[14] ::= [179] [95LRM 13.7] [152]
PATHPULSE$ = (*reject*_limit_value [, *error*_limit_value]) ;

[12] The term state-dependent_path_declaration is defined as state_dependent_path_declaration.

[13] The construct for procedural_continuous_assignment has a leading vertical bar, |, in Annex A of the 95 LRM.

[14] The specparam PATHPULSE$ is shown in bold in the 95 LRM but is not a keyword.

| **PATHPULSE$**specify_input_terminal_descriptor$specify_output_terminal_descriptor
= (*reject*_limit_value [, *error*_limit_value]) ;

range ::= [66, 68, 101, 102, 104, 107, 123, 154, 160] [95LRM 7.1.5] [153]
⟦ *msb*_constant_expression : *lsb*_constant_expression ⟧

range_or_type ::= [56] [95LRM 10.3.1] [154]
range | **integer** | **real** | **realtime** | **time**

realtime_declaration ::= [7, 95] [95LRM 3.9] **realtime** list_of_real_identifiers ; [155]

real_declaration ::= [7, 95] [95LRM 3.9] **real** list_of_real_identifiers ; [156]

real_number[15] ::= [114] [95LRM 2.5.1] [157]
[sign] unsigned_number **.** unsigned_number
| [sign] unsigned_number [**.** unsigned_number] **e** [sign] unsigned_number
| [sign] unsigned_number [**.** unsigned_number] **e** [sign] unsigned_number

register_name ::= [86] [95LRM 3.2.2] *register*_identifier [158]
| *memory*_identifier ⟦ *upper_limit*_constant_expression :
*lower_limit*_constant_expression ⟧

reg_assignment ::= [89, 147] [95LRM 9.3] reg_lvalue = expression [159]

reg_declaration ::= [7, 95, 203] [95LRM 3.2.2] [160]
reg [range] list_of_register_identifiers ;

reg_lvalue ::= [6, 112, 147, 159] [95LRM 9.2.1] [161]
*reg*_identifier | *reg*_identifier ⟦ expression ⟧
| *reg*_identifier ⟦ *msb*_constant_expression : *lsb*_constant_expression ⟧
| *reg*_concatenation

scalar_constant ::= [163] [95LRM 2.5.1] [162]
1'b0 | **1'b1** | **1'B0** | **1'B1** | **'b0** | **'b1** | **'B0** | **'B1** | **1** | **0**

scalar_timing_check_condition ::= [194] [95LRM 14.5.11] expression [163]
| ~ expression | expression == scalar_constant
| expression === scalar_constant
| expression != scalar_constant
| expression !== scalar_constant

sequential_body ::= [198] [95LRM 8.1, 8.1.4] [udp_initial_statement] [164]
table sequential_entry { sequential_entry } **endtable**

sequential_entry ::= [164] [95LRM 8.1, 8.3, 8.4] [165]
seq_input_list : current_state : next_state ;

seq_block ::= [181] [95LRM 9.8.1] **begin** [: *block*_identifier [166]
{ block_item_declaration }] { statement } **end**

[15] The term real_number has identical entries for the two forms of scientific notation in Annex A of the 95 LRM. In Section 2.5 of the 95 LRM the last alternative uses E (uppercase) instead of e (lowercase).

The BNF syntax on this page is from IEEE Std 1364-1995, Copyright © 1995. IEEE. All rights reserved.

seq_input_list ::= [165] [95LRM 8.1] level_input_list | edge_input_list [167]

short_comment ::= [15] [95LRM 2.3] // comment_text **\n** [168]

sign ::= [28, 157] [95LRM 2.5.1] + | - [169]

simple_identifier[16] ::= [63] [95LRM 2.7] [a-zA-Z_][a-zA-Z_$] [170]

simple_path_declaration ::= [138, 183] [95LRM 13.3.2] [171]
parallel_path_description = path_delay_value
| full_path_description = path_delay_value

size ::= [4, 28, 61, 121] [95LRM 2.5.1] unsigned_number [172]

source_text ::= [—] [95LRM 2.1] { description } [173]

specify_block ::= [94] [95LRM 13.1] **specify** [specify_item] **endspecify** [174]

specify_input_terminal_descriptor ::= [81, 127, 128, 152, 178] [95LRM 13.3.2] [175]
input_identifier
| input_identifier **[** constant_expression **]**
| input_identifier **[** *msb*_constant_expression **:** *lsb*_constant_expression **]**

specify_item ::= [174] [95LRM 13.1] [176]
specparam_declaration | path_declaration | system_timing_check

specify_output_terminal_descriptor ::= [82, 127, 128, 152, 178] [95LRM 13.3.2] [177]
output_identifier
| output_identifier **[** constant_expression **]**
| output_identifier **[** *msb*_constant_expression **:** *lsb*_constant_expression **]**

specify_terminal_descriptor ::= [23, 195] [95LRM 13.3.2] [178]
specify_input_terminal_descriptor
| specify_output_terminal_descriptor

specparam_assignment ::= [87] [95LRM 13.2] [179]
*specparam*_identifier = constant_expression | pulse_control_specparam

specparam_declaration ::= [176] [95LRM 13.2] [180]
specparam list_of_specparam_assignments **;**

statement[17] ::= [1, 56, 64, 89, 133, 166, 182] [95LRM 9.1] [181]
blocking_assignment **;** | non_blocking assignment **;**
| procedural_continuous_assignments **;**
| procedural_timing_control_statement | conditional_statement
| case_statement | loop_statement | wait_statement
| disable_statement | event_trigger | seq_block | par_block
| task_enable | system_task_enable

[16] The underscore in the first bracket is missing in the 95 LRM.

[17] The term blocking(underscore)assignment is defined as blocking(space)assignment. The term non(underscore)blocking(space)assignment is a single term defined as non(hyphen)blocking(space)assignment. The term procedural_continuous_assignments is defined as procedural_continuous_assignment (singular).

The BNF syntax on this page is from IEEE Std 1364-1995, Copyright © 1995. IEEE. All rights reserved.

```
statement_or_null17 ::= [8, 18, 148, 191, 207] [95LRM 9.1]                 [182]
   statement | ;

state_dependent_path_declaration18 ::= [138] [95LRM 13.3.4]                [183]
    if ( conditional_expression ) simple_path_declaration
   | if ( conditional_expression ) edge_sensitive_path_declaration
   | ifnone simple_path_declaration

strength0 ::= [36, 149, 150] [95LRM 7.10]                                   [184]
   supply0 | strong0 | pull0 | weak0

strength1 ::= [36, 149, 150] [95LRM 7.10]                                   [185]
   supply1 | strong1 | pull1 | weak1

string ::= [19] [95LRM 2.6] " { Any_ASCII_Characters_except_new_line } "    [186]

system_task_enable ::= [181] [95LRM 2.7.3]                                  [187]
   system_task_name [ ( expression { , expression } ) ] ;

system_task_name ::= [187] [95LRM 2.7.3] $identifier                        [188]
   Note: The $ may not be followed by a space.

system_timing_check19 ::= [176] [95LRM 14.5]                                [189]
   $setup ( timing_check_event , timing_check_event ,
     timing_check_limit [ , notify_register ] ) ;
   | $hold ( timing_check_event , timing_check_event ,
     timing_check_limit [ , notify_register ] ) ;
   | $period ( controlled_timing_check_event , timing_check_limit
     [ , notify_register ] ) ;
   | $width ( controlled_timing_check_event , timing_check_limit ,
     constant_expression [ , notify_register ] ) ;
   | $skew ( timing_check_event , timing_check_event ,
     timing_check_limit [ , notify_register ] ) ;
   | $recovery ( controlled_timing_check_event , timing_check_event ,
     timing_check_limit [ , notify_register ] ) ;
   | $setuphold ( timing_check_event , timing_check_event ,
     timing_check_limit , timing_check_limit [ , notify_register ] ) ;

task_argument_declaration20 ::= [191?] [95LRM 10.2.1]                       [190]
   block_item_declaration | output_declaration | inout_declaration
```

[17] The term statement_or_null is equivalent to a statement term (which, when expanded, will be terminated by a semicolon) or the combination of nothing (null) followed by a semicolon.

[18] The term state_dependent_path_declaration is referenced as state-dependent_path_declaration.

[19] The names of the system timing check tasks are shown in bold in the 95 LRM but are not keywords.

[20] Annex A of the 95 LRM defines task_argument_declaration, which is not referenced in Annex A; see the footnote for the term task_declaration.

The BNF syntax on this page is from IEEE Std 1364-1995, Copyright © 1995. IEEE. All rights reserved.

task_declaration[21] ::= [95] [95LRM 10.2.1] [191]
task *task*_identifier ; {task_item_declaration} statement_or_null **endtask**

task_enable ::= [181] [95LRM 10.2.2] [192]
*task*_identifier [(expression { , expression })] ;

time_declaration ::= [7, 95] [95LRM 3.9] **time** list_of_register_identifiers ; [193]

timing_check_condition ::= [23, 195] [95LRM 14.5.11] [194]
scalar_timing_check_condition | (scalar_timing_check_condition)

timing_check_event ::= [189] [95LRM 14.5] [195]
[timing_check_event_control]
specify_terminal_descriptor [**&&&** timing_check_condition]

timing_check_event_control ::= [23, 195] [95LRM 14.5] [196]
posedge | **negedge** | edge_control_specifier

timing_check_limit ::= [189] [95LRM 14.5] expression [197]

udp_body ::= [199] [95LRM 8.1] combinational_body | sequential_body [198]

udp_declaration ::= [34] [95LRM 8.1, 8.1.1] [199]
primitive *udp*_identifier (udp_port_list) ;
udp_port_declaration { udp_port_declaration } udp_body
endprimitive

udp_initial_statement ::= [164] [95LRM 8.1, 8.5] [200]
initial *udp_output_port*_identifier = init_val ;

udp_instance ::= [202] [95LRM 8.6] [201]
[name_of_udp_instance]
(output_port_connection , input_port_connection { , input_port_connection })

udp_instantiation ::= [94] [95LRM 8.6] [202]
*udp*_identifier [drive_strength] [delay2] udp_instance { , udp_instance } ;

udp_port_declaration ::= [199] [95LRM 8.1] [203]
output_declaration | input_declaration | reg_declaration

udp_port_list ::= [199] [95LRM 8.1, 8.1.2] [204]
*output_port*_identifier , *input_port*_identifier { , *input_port*_identifier }

unary_operator ::= [19, 52] [95LRM 4.1] [205]
+ | **-** | **!** | **~** | **&** | **~&** | **|** | **~|** | **^** | **~^** | **^~**

unsigned_number ::= [28, 33, 157, 172] [95LRM 2.5.1] [206]
decimal_digit { _ | decimal_digit }

[21] Annex A and Section 10.2.1 of the 95 LRM define `task_declaration` using the term `task_item_declaration`, which is not defined in Annex A. Section 10.2.1 defines `task_item_declaration` similarly to the Annex A definition of `task_argument_declaration`, but with the addition of the alternative term `input_declaration`.

```
wait_statement[22] ::= [181] [95LRM 9.7.5]                                  [207]
  | wait ( expression ) statement_or_null

white_space ::= [47] [95LRM 2.2] space | tab | newline                     [208]
```

B.3 BNF Index

Table B.1 is an index to the 208 Verilog HDL BNF productions, as defined in Annex A of the 95 LRM. For example, to find the legal positions of `wait_statement` (rule 207) we look up 207 in Table B.1 and find rule 181 (`statement`), which is in turn referenced by rules 1, 56, 64, 89, 133, 166, and 182. Thus we know a `wait` statement is legal in the following places: `always_construct` (1), `function_declaration` (56), `initial_construct` (64), `loop_statement` (89), `par_block` (a parallel block, 133), `seq_block` (a sequential block, 166), and anywhere `statement_or_null` (182) is legal. Turning again to Table B.1 (or using the backward-pointing links in rule 182), we find `statement_or_null` (rule 182) is legal in the following places: 8 (`case_item`), 18 (`conditional_statement`), 148 (`procedural_timing_control_statement`), 191 (`task_declaration`), and 207 (`wait_statement`).

Table B.2 is a list of the 102 Verilog HDL keywords in the 95 LRM and an index to the rules that reference these keywords. Note the spelling of the keyword scalared (not scalered). For example, to find out how to use the keyword `parameter` to define a constant, we look up *parameter* in Table B.2 to find rule 129 (`parameter_declaration`), which includes a reference to section 3.10 of the 95 LRM. The index in this book will also help (the entry for *parameter* points you to examples in Section 11.2.4, "Numbers," in this case).

There are many Verilog tools currently available that use many versions of the Verilog language. Most tool vendors explain which of the Verilog constructs are supported; many use the 95 LRM BNF syntax in this explanation.

B.4 Verilog HDL LRM

An important feature of Verilog is the ability to extend tools by writing your own code and integrating it with a Verilog-based tool. For example, the following code calls a user-written system task, `$hello`:

```
initial $hello(a_reg);
```

Here is the C program, `hello.c`, that prints the full hierarchical name of the instance in which the Verilog code containing the call to `$hello` is located:

```
#include "veriuser.h"
#include "acc_user.h"
int hello()
```

[22] The construct for `wait_statement` has a leading vertical bar, |, in Annex A of the 95 LRM.

TABLE B.1 Index to Verilog HDL BNF rules (list of rules that reference a rule).

1	94	43	40	85	155, 156	127	42	169	28, 157
2	4	44	58	86	71, 160, 193	128	171	170	63
3	4	45	58	87	180	129	7, 95	171	138, 183
4	114	46	45, 98, 134	88	15	130	94	172	4, 28, 61, 121
5	19, 52	47	63	89	181	131	93	173	Highest-level
6	181	48	32	90	31, 146	132	79	174	94
7	57, 133, 166, 190	49	7, 95	91	34	133	181	175	81, 127, 128, 152, 178
8	9	50	48, 50	92	93	134	58	176	174
9	181	51	181	93	94	135	58	177	82, 127, 128, 152, 178
10	107	52	See below	94	91	136	58	178	23, 195
11	58	53	42	95	94	137	58	179	87
12	58	54	171	96	91	138	176	180	176
13	198	55	146	97	58	139	80	181	See below
14	13	56	95	98	58	140	42, 171	182	8, 18, 148, 191, 207
15	Not referenced	57	56	99	21, 146	141	12	183	138
16	88, 168	58	94	100	75	142	53, 54, 127, 128	184	36, 149, 150
17	21, 109, 146, 161	59	61	101	See below	143	83	185	36, 149, 150
18	181	60	61	102	92	144	143	186	19
19	See below	61	114	103	55	145	144	187	181
20	33, 74, 139	62	See below	104	201	146	52	188	187
21	19	63	62	105	12	147	181	189	176
22	94	64	94	106	76, 147	148	181	190	191 (See BNF footnote)
23	189	65	200	107	95	149	58	191	95
24	165	66	95, 190	108	77	150	58	192	181
25	53, 127	67	134, 137	109	106, 147	151	58	193	7, 95
26	28	68	57, 95, 203	110	107	152	179	194	23, 195
27	206	69	175	111	165	153	See below	195	189
28	114	70	See below	112	181	154	56	196	23, 195
29	58, 202	71	7, 95	113	189	155	7, 95	197	189
30	22, 58, 107	72	14, 167	114	21, 146	156	7, 95	198	199
31	32	73	24, 40, 41, 72	115	58	157	114	199	34
32	6, 112, 148	74	152	116	58	158	86	200	164
33	29, 30, 31	75	92	117	58	159	89, 147	201	202
34	173	76	22	118	58	160	7, 95, 203	202	94
35	181	77	107	119	121	161	6, 112, 147, 159	203	199
36	22, 58, 107, 202	78	107	120	121	162	163	204	199
37	196	79	129, 130	121	114	163	194	205	19, 52
38	37	80	140	122	75	164	198	206	28, 33, 157, 172
39	53, 127	81	53, 54	123	95, 190, 203	165	164	207	181
40	41	82	53, 54	124	177	166	181	208	47
41	167	83	91	125	14, 111	167	165		
42	138, 183	84	66, 68, 123	126	See below	168	15		

19	19, 20, 67, 109, 126, 132, 145, 146, 153, 158, 161, 175, 177, 179, 189
52	6, 8, 9, 17, 18, 25, 32, 50, 52, 55, 89, 90, 99, 100, 106, 108, 109, 112, 122, 131, 146, 159, 161, 163, 187, 192, 197, 207
62	21, 35, 48, 49, 50, 51, 55, 56, 67, 69, 78, 84, 85, 91, 93, 100, 101, 102, 103, 104, 108, 109, 113, 124, 126, 132, 133, 143, 145, 146, 158, 161, 166, 179, 188, 191, 192, 199, 200, 202, 204
70	12, 45, 98, 116, 118
101	12, 45, 98, 116, 118, 134, 137, 151
126	12, 45, 98, 116, 118, 151
153	66, 68, 101, 102, 104, 107, 123, 154, 160
181	1, 56, 64, 89, 133, 166, 182

TABLE B.2 Verilog HDL keywords and index (list of rules that reference a keyword).

always 1	endprimitive 199	medium 10	realtime 154, 155	tranif0 135
and 115	endspecify 174	module 96	reg 160	tranif1 135
assign 22, 147	endtable 13, 164	nand 115	release 147	tri 110
begin 166	endtask 191	negedge 39, 50, 196	repeat 32, 89	tri0 110
buf 117	event 49	nmos 97	rnmos 97	tri1 110
bufif0 44	for 89	nor 115	rpmos 97	triand 110
bufif1 44	force 147	not 117	rtran 136	trior 110
case 9	forever 89	notif0 44	rtranif0 135	trireg 107
casex 9	fork 133	notif1 44	rtranif1 135	vectored 107
casez 9	function 56	or 50, 115	scalared 107	wait 207
cmos 11	highz0 36	output 123	small 10	wand 110
deassign 147	highz1 36	parameter 129	specify 174	weak0 184
default 8	if 18, 183	pmos 97	specparam 180	weak1 185
defparam 130	ifnone 183	posedge 39, 50, 196	strong0 184	while 89
disable 35	initial 64, 200	primitive 199	strong1 185	wire 110
edge 37	inout 66	pull0 184	supply0 110, 184	wor 110
else 18	input 68	pull1 185	supply1 110, 185	xnor 1115
end 166	integer 71, 154	pulldown 58	table 13, 164	xor 115
endcase 9	join 133	pullup 58	task 191	
endfunction 56	large 10	rcmos 11	time 154, 193	
endmodule 91	macromodule 96	real 154, 156	tran 136	

```
{ handle mod_handle; char *full_name; acc_initialize();
mod_handle = acc_handle_tfarg(1);
io_printf("Hello from: %s\n", acc_fetch_fullname(mod_handle));
acc_close(); }
```

The details of how to compile and link your program with the Verilog executable depend on the particular tool; the names, functions, and parameters of ACC routines, the header files, veriuser.h and acc_user.h (most companies include these with their Verilog products), as well as older TF routines and the newer VPI routines are described in detail in Sections 17–23 of the 95 LRM.

Annex F of the 95 LRM describes widely used Verilog system tasks and functions that are not required to be supported as part of IEEE Std 1364-1995. Table B.3 summarizes these tasks and functions. Annex G of the 95 LRM describes additional compiler directives that are not part of IEEE Std 1364-1995 and are not often used by ASIC designers. Two directives, `default_decay_time and `default_trireg_strength, are used to model charge decay and the strength of high-impedance trireg nets. Four more compiler directives: `delay_mode_distributed, `delay_mode_path, `delay_mode_unit, and `delay_mode_zero are used to specify the delay mode for modules.

TABLE B.3 System tasks and functions (not required in IEEE Std 1364-1995).

`$countdrivers ( net, [ net_is_forced, number_of_01x_drivers, number_of_0_drivers, number_of_1_drivers, number_of_x_drivers ] ) ;` Returns a 0 if there is no more than one driver on the net and returns a 1 otherwise (indicating contention).

`$getpattern ( mem_element ) ; // Drive a pattern from an indexed memory.`
Example: `assign {i1, i2, i3, i4} = $getpattern ( mem [ index ] )`

`$input ("filename"); // Allows input from file rather than terminal.`
`$key [ ( "filename" ) ] ; $nokey ; // Enable/disable key file in interactive mode.`
`$list [ ( hierarchical_name ) ] ; // List current or specified object.`
`$log [ ( "filename" ) ] ; $nolog ; // Enable/disable log file for standard output.`

`$reset [ ( stop_value [ , reset_value , [ diagnostics_value ] ] ) ] ; // Reset time.`
`$reset_count ; // Count the number of resets.`
`$reset_value ; // Pass information prior to reset to simulation after reset.`

`$save ( "file_name" ) ; // Save simulation for later restart.`
`$restart ( "file_name" ) ; // Restart simulation from saved file.`
`$incsave ( "incremental_file_name" ) ; // Save only changes since last $save`

`$scale ( hierarchical_name ) ; // Convert to time units of invoking module.`

`$scope ( hierarchical_name ) ; // Sets the specified level of hierarchy as current scope.`
`$showscopes [ ( n ) ]; // Show scope (n = none or zero) else show all items below scope.`

`$showvars [ ( list_of_variables ) ] ; // Show status of scope or specified variables.`

`$sreadmemb ( mem_name , start_address , finish_address , string { , string } ) ;`
`$sreadmemh ( mem_name , start_address , finish_address , string { , string } ) ;`
Load data into `mem_name` from character string (same format as `$readmemb/h`).

B.5 Bibliography

There are fewer books available on Verilog than on VHDL. The best reference book is the IEEE Verilog HDL LRM [IEEE 1364-1995]; it is detailed as well as containing many examples. In addition to the references given in Chapter 11, the following books concentrate on Verilog: Sternheim, Singh, and Trivedi [1990] (Yatin Trivedi was the technical editor for the 95 LRM); Thomas and Moorby [1991]; Smith [1996]; and Golze and Blinzer [1996]. Capilano Computing Systems has produced a book to accompany its Verilog Modeler product [Capilano, 1997].

Sandstrom compiled an interesting cross-reference between Verilog and VHDL (a 2.5 page table listing the correspondence between major constructs in both languages) in a pull-out supplement to *Integrated System Design Magazine*. An electronic version of this article is at `http://www.isdmag.com` (the article is labeled January 1996, but filed under October 1995). Other online articles related to Verilog at `www.isdmag.com`, include case studies of Sun Microsystems' ULTRASparc-1 (June 1996) and Hewlett–Packard's PA-8000 (January, February, and March 1997); both CPUs were designed with Verilog behavioral models. The March 1997 issue also contains an article on the recent history and the future plans of Open Verilog International (OVI). OVI helped create IEEE Std 1364-1995 and sponsored the annual *International Verilog HDL Conference* (IVC). In 1997 the IVC merged with the VHDL International Users' Forum (VIUF) to form the *IVC/VIUF Conference* (see `http://www.hdlcon.org`).

In January of 1995 OVI reactivated the Technical Coordinating Committee (TCC) to recommend updates and changes to Verilog HDL. The TCC comprises technical subcommittees (TSC), which are developing a delay calculator standard (LM-TSC), analog extensions to Verilog HDL (VA-TSC), an ASIC library modeling standard (PS-TSC), cycle-based simulation standard (VC-TSC), timing-constraint formats (VS-TSC), as well as Verilog language enhancements and extensions (VD-TSC). Links and information about OVI are available at `http://www.avanticorp.com` and `http://www.chronologic.com`. The OVI web site is `http://www.verilog.org/ovi`. Information on the activities of the OVI committees is available at the Meta-Software site, `ftp://ftp.metasw.com/pub`.

The work of the OVI and IEEE groups is related. For example, the IEEE Design Automation Standards Committee (DASC) contains the Verilog Working Group (PAR 1364), the Circuit Delay and Power Calculation (DPC) System Study Group (P1481), as well as the VHDL and other WGs. Thus, the OVI DC-TSC directory contains the Standard Delay Calculation System (DCS) Specification (v1.0) approved by OVI/CFI and currently being studied by the IEEE DPC Study Group. DCS provides a standard system for designers to calculate chip delay and power using the following methods: Delay Calculation Language (DCL) from IBM and CFI, Detailed Standard Parasitic Format (DSPF) and Reduced Standard Parasitic Format (RSPF) from Cadence Design Systems (combined into a new Standard Parasitics Exchange Format, SPEF), and Physical Design Exchange Format (PDEF) from Synopsys. The current IEEE standardization work is expanding the scope to add power calculation. Thus, useful information relating to Verilog may be found at the VHDL site, VIUF Internet Services (VIIS at `http://www.vhdl.org`), as well as the OVI site.

Two `usenet` newsgroups are related to Verilog: `comp.lang.verilog` and `comp.cad.synthesis`. In January of 1997 the Verilog news archive was lost due to a disk problem. While attempts are made to restore the archive, the Verilog Frequently Asked Questions (FAQ) list is still available at `http://www.lib.ox.ac.uk/internet/news/faq/archive/verilog-faq.html`. A list of CAD-related newsgroups (including `comp.lang.verilog`) is maintained at Sun Microsystems' DACafe (`http://www.ibsystems.com/DACafe/TECHNICAL/Resources/NewsGps.html`. Sun (`~/DACafe/US ERSGROUPS`) also maintains the following user groups that often discuss Verilog: Cadence, Mentor Graphics, Synopsys, VeriBest, and Viewlogic. A number of tools and resources are available on the World Wide Web, including Verilog modes for the emacs editor; Verilog preprocessors in Perl and C (which allow the use of `` `define `` and `` `ifdef `` with logic synthesis tools, for example); and demonstration versions of the following simulators: Viper from InterHDL (`http://www.interhdl.com`) and VeriWell from Wellspring Solutions (`http://www.wellspring.com`). VeriWell now supports the Verilog PLI, including the `acc` and `tf` routines in IEEE Std 1364-1995 (requiring Visual C++ 4.0 or newer for the Windows version, Code Warrior 9 or newer for the Macintosh, and GNU C 2.7.0 or newer for the Linux and Sparc versions).

Several personal Web pages focus on Verilog HDL; these change frequently but can be found by searching. Actel has placed a number of Verilog examples (including synthesizable code for a FIFO and a RAM) at its site: `http://wwwtest.actel.com/HLD/verimain.html`. Many universities maintain Web pages for Verilog-related classes. Examples are the Web site for the ee282 class at Stanford (`http://lummi.Stanford.EDU/class/ee282`), which contains Verilog models for the DLX processor in the second edition of Hennessy and Patterson's "Computer Architecture: A Quantitative Approach"; and course material for 18-360, "Introduction to Computer-Aided Digital Design," by Prof. Don Thomas at `http://www.ece.cmu.edu`.

B.6 References

Page numbers in brackets after a reference indicate its location in the chapter body.

Capilano. 1997. *LogicWorks Verilog Modeler: Interactive Circuit Simulation Software for Windows and Macintosh.* Menlo Park, CA: Capilano Computing, 102 p. ISBN 0201895854. TK7888.4.L64 (as cataloged by the LOC). Addison-Wesley also gives the following additional ISBN numbers for this work: ISBN 0-201-49885-5 (Windows book and software), ISBN 0-201-49884-7 (Macintosh book and software); also available bundled with LogicWorks 3: ISBN 0-201-87436-9 (Macintosh), ISBN 0-201-87437-7 (Windows).

Golze, U., and P. Blinzer. 1996. *VLSI Chip Design with the Hardware Description Language VERILOG: An Introduction Based on a Large RISC Processor Design.* New York: Springer, 358 p. ISBN 3540600329. TK7874.75.G65. Four pages of references. Includes a version of VeriWell from Wellsprings Solutions.

IEEE 1364-1995. *IEEE Standard Description Language Based on the Verilog® Hardware Description Language.* 688 p. ISBN 1-55937-727-5. IEEE Ref. SH94418-NYF. Published by The IEEE, Inc., 345 East 47th Street, New York, NY 10017, USA. Inside the United States, IEEE standards may be ordered at 1-800-678-4333. See also `http://www.ieee.org` and `http://stdsbbs.ieee.org`. This standard was approved by the IEEE on 12 December, 1995; and approved by ANSI on 1 August, 1996 (and thus these two organizations have different publication dates). Contents: overview (4 pages); lexical conventions (8 pages); data types (13 pages); expressions (18 pages); scheduling semantics (5 pages); assignments (4 pages); gate and switch level modeling (31 pages); user-defined primitives (11 pages); behavioral modeling (26 pages); tasks and functions (6 pages); disabling of named blocks and tasks (1 page); hierarchical structures (16 pages); specify blocks (18 pages); system tasks and functions (35 pages); value change dump file (11 pages); compiler directives (8 pages); PLI TF and ACC interface mechanism (6 pages); using ACC routines (36 pages); ACC routine definitions (178 pages); using TF routines (5 pages); TF routine definitions (76 pages); using VPI routines (6 pages); VPI routine definitions (25 pages); formal syntax definition; list of keywords; system tasks and functions; compiler directives; `acc_user.h`; `veriuser.h`; `vpi_user.h`. [p. 979]

Smith, D. J. 1996. *HDL Chip Design: A Practical Guide for Designing, Synthesizing, and Simulating ASICs and FPGAs using VHDL or Verilog.* Madison, AL: Doone Publications, 448 p. ISBN 0965193438. TK7874.6.S62.

Sternheim, E., R. Singh, and Y. Trivedi. 1990. *Digital Design with Verilog HDL.* Cupertino, CA: Automata Publishing, 215 p. ISBN 0962748803. TK7885.7.S74.

Thomas, D. E., and P. Moorby. 1991. *The Verilog Hardware Description Language.* Boston, MA: Kluwer, 223 p. ISBN 0-7923-9126-8, TK7885.7.T48 (1st ed.). ISBN 0-7923-9523-9 (2nd ed.). ISBN 0792397231 (3rd ed.).

GLOSSARY OF SYMBOLS AND ACRONYMS

Symbols

A_0, parameter in input-slope delay model 673

A_1, parameter in input-slope delay model 673

$\mathrm{A_D}$, transistor drain area 124

$\mathrm{A_S}$, transistor source area 124

β_n, transistor gain factor 44

C, transistor gate capacitance 43

C_{BD}, bulk-to-drain capacitance 123

C_{BDJ}, bulk-to-drain junction area capacitance 123

C_{BDSW}, bulk-to-drain junction sidewall capacitance 123

C_{BS}, bulk-to-source capacitance 123

C_{BSJ}, bulk-to-source junction area capacitance 123

$C_{BDJ\mathrm{GATE}}$, bulk-to-drain channel-edge capacitance 123

$C_{BSJ\mathrm{GATE}}$, bulk-to-drain channel-edge capacitance 123

C_{BSSW}, bulk-to-source junction sidewall capacitance 123

C_{GB}, gate-to-bulk capacitance 123

$\mathrm{C_{GBOV}}$, gate-to-bulk overlap capacitance 123

C_{GD}, gate-to-drain variable capacitance 123

$\mathrm{C_{GDOV}}$, gate-to-drain overlap capacitance 123

C_{GS}, gate-to-source capacitance 123

$\mathrm{C_{GSOV}}$, gate-to-source overlap capacitance 123

C_{inv}, input capacitance of minimum-size inverter 131

$C_{J\mathrm{GATE}}$, channel edge capacitance 124

C_L, output load capacitance 138

$\mathrm{C_O}$, transistor gate capacitance (calculated using effective gate width and effective gate length) 126

C_{out}, extrinsic output capacitance 118

$\mathrm{C_{ox}}$, gate capacitance per unit area 43

C_p, intrinsic output capacitance 118

C_R, critical ramp delay in input-slope model 673

C_S, transistor channel–bulk depletion capacitance 126

D, delay in input-slope model 673

D, path delay in logical effort model 136

D_0, experimentally determined factor in input-slope delay model 673

D_0, parameter in input-slope delay model 673

D_1, parameter in input-slope delay model 673

d_A, parameter in input-slope delay model 673

d_D, parameter in input-slope delay model 673

D_{t0}, time from the beginning of the input to the beginning of the output (input-slope delay model) 672

D_{t1}, time from the beginning of the input to the end of the output (input-slope delay model) 672

E, electric field (vector) 42

ε_0, vacuum permittivity 104

ε_r, relative permittivity of silicon 104

ε_Si, permittivity of silicon 104

ε_ox, permittivity of silicon dioxide 43

E_x, horizontal component of electric field in a transistor 42

F, path effort 139

f, effort delay 131

ϕ_0, surface potential 104

G, path logical effort 138

γ, back-gate bias coefficient 104

g, logical effort 131

H, path electrical effort 139

h, electrical effort 131

h_i, stage electrical effort 139

$I_{DS(\mathrm{sat})}$, transistor drain–source saturation current 44

I_{DSn}, transistor drain–source current 42

I_R, time from the beginning to the end of the input ramp (input-slope delay model) 672

k_n' , process transconductance parameter 44

L, transistor length 43

$\mathrm{L_D}$, lateral diffusion 123

$\mathrm{L_{eff}}$, transistor effective gate length 45

μ_n, electron mobility 42

μ_p, hole mobility 42

N, number of inverters in an inverter chain 140

n, number of inputs to a logic cell 133

O_R, output ramp delay in input-slope model 673

P, path parasitic delay 139

p, parasitic delay 132

$\mathrm{P_D}$, transistor drain perimeter (excluding channel edge) 124

$\mathrm{P_S}$, transistor source perimeter (excluding channel edge) 124

Q, path nonideal delay 139

Q, transistor channel charge 42
q, nonideal delay 133
R, pull resistance 120
r, logic ratio 132
R_{inv}, pull resistance of minimum-size inverter 131
R_{pd}, pull-down resistance 117
R_{pu}, pull-up resistance 117
τ, time constant for delay measurement 131
t', time after input crosses trip-point 120
t_f, transistor time-of-flight 42
$\mathrm{T_{ox}}$, transistor gate-oxide thickness 43
t_{PD}, inverter propagation delay 117
t_{PDf}, inverter fall-time propagation delay 117
t_{PDr}, inverter rise-time propagation delay 117
t_q, nonideal delay 130
v, electron velocity (vector) 42
V_{DD}, positive supply voltage 40
V_{DS}, transistor drain–source voltage 41
$V_{DS(\mathrm{sat})}$, transistor drain–source saturation voltage 44
V_{GC}, transistor gate–channel potential 43
V_{GS}, transistor gate–source voltage 41
V_{SS}, negative supply voltage 40
$\mathrm{V}_{\mathrm{t0}n}$, threshold voltage at zero substrate-bias 104
$\mathrm{V}_{\mathrm{t}n}$, n-channel transistor threshold voltage 41
$\mathrm{V}_{\mathrm{t}p}$, p-channel transistor threshold voltage 45
v_x, horizontal component of carrier velocity in a transistor 43
W, transistor width 43
$\mathrm{W_{EFF}}$, effective transistor width 124
x_d, transistor channel–bulk depletion width 126
Z, experimentally determined factor in input-slope delay model 673

A

A/D, analog-to-digital converter 248
AB, cell abutment box 150
ACC, (Verilog) access routines 541
ACM, area-calculation method, SPICE model parameter 124
ACM, Association of Computing Machinery 114
ALU, arithmetic and logical unit 10
AOI, AND-OR-INVERT 60
AQL, average quality level 712
ASIC, application-specific integrated circuit 1
ASSP, application-specific standard product 4
ATM, Asynchronous Transfer Mode 820
ATPG, automatic test-pattern generation 755
ATVG, automatic test-vector generation 755
AWE, asymptotic waveform evaluation 708

B

BB, cell bounding box 150
BEOL, back end of the line 60
BGA, ball-grid array 866
BiCMOS, bipolar/CMOS 3
BIDI, bidirectional 284
BILBO, built-in logic block observer 776
BIST, built-in self-test 766
BNF, Backus-Naur form 390
BOM, bill of materials 398
BPSG, boron-doped phosphosilicate glass 58
BR, bypass register 716
BSC, boundary-scan cell 716
BSDL, boundary-scan description language 732
BSIM, Berkeley short-channel IGFET model 692
BSR, boundary-scan register 716

C

C5, industrial 0.5 µm process 130
CAE, computer-aided engineering 176
CBA, carry-bypass adder 81
CBA, cell-based array 149
CBIC, cell-based integrated circuit 6
CDM, charge-device model 101
CGBO, SPICE parameter 693
CGDO, SPICE parameter 693
CGSO, SPICE parameter 693
CICC, Custom Integrated Circuits Conference 3
CIF, Caltech Intermediate Format 946
CIN, carry in 75
CJ, SPICE parameter 693
CJGATE, channel-edge capacitance, SPICE model parameter 124
CJSW, SPICE parameter 693
CLA, carry-lookahead adder 83
CMOS, complementary MOS 2
CMP, chemical mechanical polishing 57
COT, customer-owned tooling 28
COUT, carry out 75
CPA, carry-propagate adder 81
CPU, central processing unit 4
CR, carriage return 391
CSA, carry-save adder 80
CSD, canonical signed digit 87
CSTP, circular self-test path 776
CUT, circuit under test 766
CVD, chemical vapor deposition 51

D

DAC, Design Automation Conference 851
DASC, Design Automation Standards Committee 998
DASS, Design Automation Standards Subcommittee 801
DC, direct current 937
DCL, Delay Calculation Language 998
DCS, Delay Calculation System 998
DEF, design-exchange format 897
DEL, delete 400
DELTA, SPICE parameter 693
DES, data encryption standard 639
DIP, dual-in-line package 714
DMA, direct memory access 18
DoD, (U.S.) Department of Defense 379
DP, datapath 9
DPC, Delay and Power Calculation 998
DRAM, dynamic random-access memory 3
DRC, design-rule check 944
DS, SPICE parameter 706
DSPF, detailed SPF (standard parasitic format) 943
DUM1, SPICE parameter 706
DUT, device under test 766

E

E2W3, Electrical Engineering on the World Wide Web 37
ECL, emitter-coupled logic 2
EDA, electronic design automation 21
EDAC, Electronic Design Automation Companies 37
EEPROM, electrically erasable PROM 15
EIA, Electronic Industries Association 37
emf, electromotive force 691
EOS, electrical overstress 101
EPLD, erasable PLD 15
EPROM, electrically programmable read-only memory 15
ESD, electrostatic discharge 100, 864
ETA, SPICE parameter 693
EXOR, exclusive-OR 69

F

FA, full adder 75
FAMOS, floating-gate avalanche MOS 175
FAN, fanout-oriented test generation 761
FAQ, frequently asked questions 321
FEOL, front end of the line 60
FF, form feed 391, 400
FIFO, first-in first-out register 98
FIT, failures in time 737
FO, fanout 856
FOX, field oxide 52
FPU, floating-point unit 18
FS, ASCII control character (FSP in VHDL) 400
FSB, functional system block 7
FSM, finite-state machine 605

G

GA, gate array 11
GaAs, gallium arsenide 36
GAMMA, SPICE parameter 693
GB, gain–bandwidth product 251
GDS, small-signal drain–source conductance, SPICE output parameter 122
GM, small-signal transconductance, SPICE output parameter 122
GMB, small-signal back-gate transconductance, SPICE output parameter 122
GND, negative supply voltage 40
GRC, global-routing cell 920
GS, ASCII control character (GSP in VHDL) 400
GTL, Gunning transistor logic 242

H

HBM, human-body model 101
HDL, hardware description language 300
HT, horizontal tabulation 391
HTTP, HyperText Transfer Protocol 37
HVH, horizontal-vertical-horizontal 933

I

ICCAD, International Conference on Computer-Aided Design 956
ICCD, International Conference on Computer Design 114
IDCODE, device identification register 716
IDD, supply current 743
IDDQ, quiescent supply current (IDD) 743
IEC, International Electrotechnical Committee 101
IEEE, Institute of Electrical and Electronics Engineers 3
ILD, inter-level dielectric 58
IMO, inter-metal oxide 58
IOB, input/output block 258
IOC, I/O Control Block 261
IOE, I/O Element 261
IR, instruction register 716
IRE, Institute of Radio Engineers 114
ISAC, VHDL Issues Screening and Analysis Committee 976
ISBN, International Standard Book Number vii
ISI, Information Sciences Institute 37
ISP, in-system programming 172

ISSN, International Standard Serials Number vii
ITC, International Test Conference 800
IU, integer unit 18

J

JETAG, Joint European Test Action Group 714
JIT, just-in-time 177
JTAG, Joint Test Action Group 716

K

KAPPA, SPICE parameter 693
KP, SPICE parameter 693

L

λ, minimum feature size 3
LC, Logic Cell (Xilinx) 207
LCA, Logic Cell Array 301
LD, SPICE parameter 693
LDD, lightly doped drain 105
LE, Logic Element 209
LEA, left-edge algorithm 928
LEF, library-exchange format 897
LEVEL, SPICE parameter 693
LF, line feed 391, 400
LFSR, linear feedback shift register 766
LHS, left-hand side 424
LI, local interconnect 57
LIFO, last-in first-out register 98
LOC, Library of Congress vii
LOCIS, LOC information system vii
LPCVD, low-pressure chemical vapor deposition 105
LPM, Library of Parameterized Modules 355
LRM, Language Reference Manual 380
LSB, least-significant bit 77
LSI, large-scale integration 2
LSSD, level-sensitive scan design 765
LTL, line to line 922
LUT, look-up table 204
LVS, layout versus schematic 30
LVT, long vertical track 277

M

MAC, multiplier-accumulator 98
MAJ, majority function 75
ME, matrix element 228
MGA, masked gate array 11
MIPS, million instructions per second 20
MISR, multiple-input signature register 775
MIT, Massachussetts Institute of Technology 37
MJ, SPICE parameter 693
MJSW, SPICE parameter 693
MM,.machine model 101
MMU, memory management unit 18
modem, modulator-demodulator 4
MOS, metal-oxide-silicon 2
MOSIS, MOS Implementation Service 37
MRST, minimum rectilinear Steiner tree 877
MSB, most-significant bit 77
MSI, medium-scale integration 2
MTBF, mean time between failures 737
MTBU, mean time between upsets 250
MTTF, mean time to failure 737, 937
MUG, MOSIS user's group 37
MUX, multiplexer 67

N

NAND, not an acronym—a type of logic gate 2
NBSP, nonbreaking space 391
NFS, SPICE parameter 693
NMOS, n-channel MOS 2
NOR, not an acronym, a type of logic gate 2
NSUB, SPICE parameter 693

O

OAI, OR-AND-INVERT 60
OE, output enable 99
OEM, original equipment manufacturer 299
ONO, oxide–nitride–oxide 170
OPC, optical proximity correction 946
OTC, over the cell 873
OTP, one-time programmable 170
OVI, Open Verilog International 479

P

P&R, place and route 894
PAL, programmable array logic 15
PARITY, parity function 75
PB, SPICE parameter 693
PCB, printed-circuit board 177
PCM, process control monitor 30
PDEF, physical design exchange format 896
PGA, pin-grid array 1
PHI, SPICE parameter 693

PI, parallel in (boundary scan) 716
PI, primary input 745
PIA, Programmable Interconnect Array 289
PLA, programmable logic array 15
PLD, programmable logic device 14
PLI, Programming Language Interface 541
PLL, phase-locked loop 873
PO, parallel out (boundary scan) 717
PO, primary output 745
PODEM, path-oriented decision-making 759
PRBS, pseudorandom binary sequence 766
PREP, Programmable Electronics Performance Company 179
PROM, programmable read-only memory 14
PSK, phase-shift keying 517
PS-TSC, ASIC library modeling standard technical subcommittee 998
PTO, (U.S.) Patent and Trademark Office 188

Q

QMST, quadratic minimum Steiner tree 954

R

RAC, Rambus access cell 243
RCA, ripple-carry adder 75
ROM, read-only memory 14
RPM, Relationally Placed Modules 302
RS, ASCII control character (RSP in VHDL) 400
RSH, SPICE parameter 693
RSPF, reduced SPF (standard parasitic format) 939
RTL, register-transfer level 468

S

s/t/s, sustained three-state 238
SCOAP, Sandia controllability/observability analysis program 761
SCR, silicon-controlled rectifier 101
SDF, Standard Delay Format 512
SEMATECH, SEmiconductor MAnufacturing TECHnology 37
SI, serial in (boundary scan) 716
SISR, serial-input signature register 767
SLM, system-level macro 7
SO, serial out (boundary scan) 716
SOG, sea-of-gates 12
SOG, silicon overglass 105
SP, space 391
SPEF, Standard Parasitics Exchange Format 998
SPF, Standard Parasitic Format 939
SPT, shortest-path tree 954
SRAM, static random-access memory 3
SSF, single stuck-at fault 740
SSO, simultaneously switching output 99
STD_ULOGIC, VHDL type 402

T

TAP, test-acess port 715
TCC, OVI Technical Coordinating Committee 998
TCK, test clock 716
TDI, test data input 716
TDO, test data output 716
TDR, test data register 716
TF, (Verilog) task/function routine 541
TG, transmission gate 66
THETA, SPICE parameter 693
TMS, test-mode select 716
TOX, SPICE parameter 693
TPG, SPICE parameter 693
TRST*, test-reset input signal 716
TSC, OVI technical subcommittee 998
TSPC, true single-phase clocking 688
TTL, transistor–transistor logic 2
TX, transmission (as in TX gate) 66

U

UCB, University of California at Berkeley 638
ULSI, ultralarge scale integration 2
US, ASCII control character (USP in VHDL) 400
USC, University of Southern California 37
UVPROM, UV-erasable programmable read-only memory 15

V

VASG, VHDL Analysis and Standards Group 975
VA-TSC, analog extensions to Verilog HDL technical subcommittee 998
VCD, value change dump (file) 705
VDD, positive supply voltage 40
VHDL, VHSIC Hardware Description Language 379
VHSIC, very high-speed IC 379
VHV, vertical-horizontal-vertical 933
VIIS, VIUF Internet Services 998
VITAL, VHDL initiative towards ASIC libraries 405
VIUF, VHDL International Users' Forum 477
VLSI, very large-scale integration 2
VMAX, SPICE parameter 693
VPI, Verilog Procedural Interface 541
VQFP, very thin quad flatpack 257

VSS, negative supply voltage 40
VS-TSC, timing-constraint formats technical subcommittee 998
VT, vertical tabulation 391
VTL, via to line 922
VTO, SPICE parameter 693
VTV, via to via 922

W

w-c, worst-case 203

X

XJ, SPICE parameter 693
XNOR, exclusive-NOR 70
XOR, exclusive-OR 69

INDEX

Page references in bold refer to principal entries. For acronyms see the glossary.

tegrated Circuits/VLSI/Electrical Engineering

Application-Specific Integrated Circuits

Michael John Sebastian Smith

This comprehensive book on application-specific integrated circuits (ASICs) describes the latest methods in VLSI-systems design. ASIC design, using commercial tools and predesigned cell libraries, is the fastest, most cost-effective, and least error-prone method of IC design. As a consequence, ASICs and ASIC-design methods have become increasingly popular in industry for a wide range of applications.

The book covers both semicustom and programmable ASIC types. After describing the fundamentals of digital logic design and the physical features of each ASIC type, the book turns to ASIC logic design—design entry, logic synthesis, simulation, and test—and then to physical design—partitioning, floorplanning, placement, and routing. You will find here, in practical, well-explained detail, everything you need to know to understand the design of an ASIC, and everything you must do to begin and to complete your own design.

Features

- Broad coverage includes, in one information-packed volume, cell-based ICs, gate arrays, field-programmable gate arrays (FPGAs), and complex programmable logic devices (PLDs).
- Examples throughout the book have been checked with a wide range of commercial tools to ensure their accuracy and utility.
- Separate chapters and appendixes on *both* Verilog and VHDL, including material from IEEE standards, serve as a complete reference for high-level, ASIC-design entry.

As in other landmark VLSI books published by Addison-Wesley—from Mead and Conway to Weste and Eshraghian—the author's teaching expertise and industry experience illuminate the presentation of useful design methods. Any engineer, manager, or student who is working with ASICs in a design project, or who is simply interested in knowing more about the different ASIC types and design styles, will find this book to be an invaluable resource, reference, and guide.

Michael John Sebastian Smith is an ASIC researcher, designer, and educator. He teaches at the University of Hawaii and is a consultant in ASIC design. Previously, he worked at the IBM T. J. Watson Research Center and was a member of the team that founded Compass Design Automation, which is now part of Avant! Corporation. Smith received B.A. and M.A. degrees from Queens' College, Cambridge University, and M.S. and Ph.D. degrees from Stanford University. In 1989, he was named a U.S. National Science Foundation Presidential Young Investigator.

♲ Text printed on recycled paper

ADDISON-WESLEY
Pearson Education

56495
9 780201 500226
ISBN 0-201-50022-1
$64.95 US $97.50 CANADA