High pin-count surface-mount packaging supports even wider registers, drivers, and transceivers. Most common are 16-bit devices, but there are also devices with 18 bits (for byte parity) and 32 bits. Also, the larger packages can offer more control functions, such as clear, clock enable, multiple output enables, and even a choice of latching vs. registered behavior all in one device.

8.2.6 Registers and Latches in ABEL and PLDs

As we showed in Section 7.11, registers are very easy to specify in ABEL. For example, Table 7-33 on page 541 showed an ABEL program for an 8-bit register with enable. Obviously, ABEL allows the functions performed at the D inputs of register to be customized in almost any way desired, limited only by the number of inputs and product terms in the targeted PLD. We describe sequential PLDs in Section 8.3.

With most sequential PLDs, few if any customizations can be applied to a register's clock input (e.g, polarity choice) or to the asynchronous inputs (e.g., different preset conditions for different bits). However, ABEL does provide appropriate syntax to apply these customizations in devices that support them, as described in Section 7.11.1.

Very few PLDs have latches built in; edge-triggered registers are much more common, and generally more useful. However, you can also synthesize a latch using combinational logic and feedback. For example, the excitation equation for an S-R latch is

$$Q* = S + R' \cdot Q$$

Thus, you could build an S-R latch using one combinational output of a PLD, using the ABEL equation "`Q = S # !R & Q`." Furthermore, the S and R signals above could be replaced with more complex logic functions of the PLD's inputs, limited only by the availability of product terms (seven per output in a 16V8C or 16L8) to realize the final excitation equation. The feedback loop can be created only when Q is assigned to a bidirectional pin (in a 16V8C or 16L8, pins IO2–IO7, not O1 or O8). Also, the output pin must be continuously output-enabled; otherwise, the feedback loop would be broken and the latch's state lost.

Probably the handiest latch to build out of a combinational PLD is a D latch. The basic excitation equation for a D latch is

$$Q* = C \cdot D + C' \cdot Q$$

However, we showed in Section 7.10.1 that this equation contains a static hazard, and the corresponding circuit does not latch data reliably. To build a reliable D latch, we must include a consensus term in the excitation equation:

$$Q* = C \cdot D + C' \cdot Q + D \cdot Q$$

The D input in this equation may be replaced with a more complicated expression, but the equation's structure remains the same:

$$Q* = C \cdot \textit{expression} + C' \cdot Q + \textit{expression} \cdot Q$$

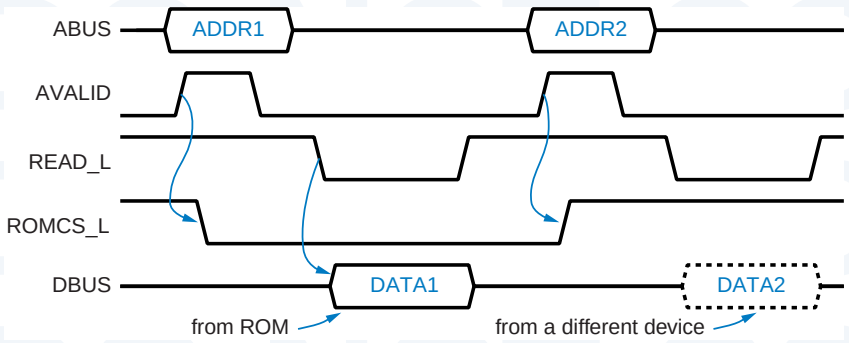

Figure 8-14
Timing diagram for a microprocessor read operation.

It is also possible to use a more complex expression for the C input, as we showed in Section 7.10.1. In any case, it is very important for the consensus term to be included in the PLD realization. The compiler can work against you in this case, since its minimization step will find that the consensus term is redundant and remove it.

`retain` property

Some versions of the ABEL compiler let you prevent elimination of consensus terms by including a keyword "`retain`" in the property list of the `istype` declaration for any output which is not to be minimized. In other versions, your only choice is to turn off minimization for the entire design.

Probably the most common use of a PLD-based latch is to simultaneously decode and latch addresses in order to select memory and I/O devices in microprocessor systems. Figure 8-14 is a timing diagram for this function in a typical system. The microprocessor selects a device and a location within the device by placing an address on its address bus (ABUS) and asserting an "address valid" signal (AVALID). A short time later, it asserts a read signal (READ_L), and the selected device responds by placing data on the data bus (DBUS).

Notice that the address does not stay valid on ABUS for the entire operation. The microprocessor bus protocol expects the address to be latched using AVALID as an enable, then decoded, as shown in Figure 8-15. The decoder selects different devices to be enabled or "chip-selected" according to the high-order bits of the address (the 12 high-order bits in this example). The low-order bits are used to address individual locations of a selected device.

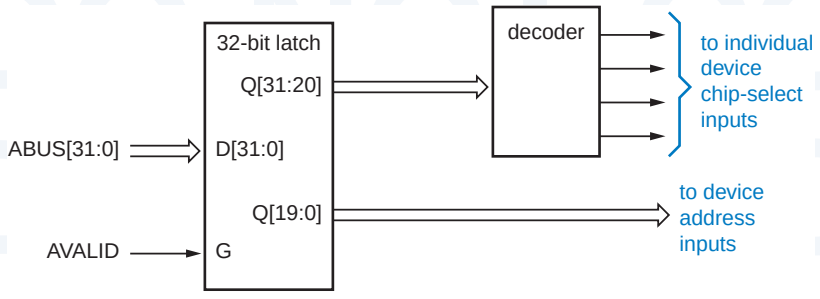

Figure 8-15
Microprocessor address latching and decoding circuit.

WHY A LATCH? The microprocessor bus protocol in Figure 8-14 raises several questions:

- Why not keep the address valid on ABUS for the entire operation? In a real system using this protocol, the functions of ABUS and DBUS are combined (multiplexed) onto one three-state bus to save pins and wires.
- Why not use AVALID as the clock input to a positive-edge-triggered register to capture the address? There isn't enough setup time; in a real system, the address may first be valid at or slightly after the rising edge of AVALID.
- OK, so why not use AVALID to clock a negative-edge-triggered register? This works, but the latched outputs are available sooner; valid values on ABUS flow through a latch immediately, without waiting for the falling clock edge. This relaxes the access-time requirements of memories and other devices drvien by the latched outputs.

Using a PLD, the latching and decoding functions for the high-order bits can be combined into a single device, yielding the block diagram in Figure 8-16. Compared with Figure 8-15, the "latching decoder" saves devices and pins, and may produce a valid chip-select output more quickly (see Exercise 8.1).

Table 8-2 is an ABEL program for the latching decoder. Since it operates on only the high-order bits `ABUS[31..20]`, it can decode addresses only in 1-Mbyte or larger chunks (2^{20} = 1M). A read-only memory (ROM) is located in the highest 1-Mbyte chunk, addresses `0xfff00000`–`0xffffffff`, and is selected by `ROMCS`. Three 16-Mbyte banks of read/write memory (RAM) are located at lower addresses, starting at addresses `0x00000000`, `0x00100000`, and `0x00200000`, respectively. Notice how don't-cares are used in the definitions of the RAM bank address ranges to decode a chunk larger than 1 Mbyte. Other approaches to these definitions are also possible (e.g., see Exercise 8.2).

The equations in Table 8-2 for the chip-select outputs follow the D-latch template that we gave on page 574. The expressions that select a device, such as "`ABUS==ROM`," each generate a single product term, and each equation generates three product terms. Notice the use of the "`retain`" property in the pin declarations to prevent the compiler from optimizing away the consensus terms.

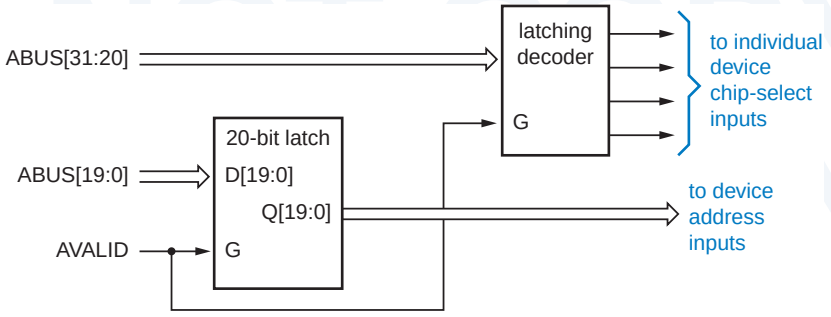

Figure 8-16 Using a combined address latching and decoding circuit.

Table 8-2 ABEL program for a latching address decoder.

```
module latchdec
title 'Latching Microprocessor Address Decoder'

" Inputs
AVALID, ABUS31..ABUS20           pin;
" Latched and decoded outputs
ROMCS, RAMCS0, RAMCS1, RAMCS2    pin istype 'com,retain';

ABUS = [ABUS31..ABUS20];
ROM = ^hFFF;
RAMBANK0 = [0,0,0,0,0,0,0,0,.X.,.X.,.X.,.X.];
RAMBANK1 = [0,0,0,0,0,0,0,1,.X.,.X.,.X.,.X.];
RAMBANK2 = [0,0,0,0,0,0,1,0,.X.,.X.,.X.,.X.];

equations

ROMCS  = AVALID & (ABUS==ROM) # !AVALID & ROMCS
       # (ABUS==ROM) & ROMCS;
RAMCS0 = AVALID & (ABUS==RAMBANK0) # !AVALID & RAMCS0
       # (ABUS==RAMBANK0) & RAMCS0;
RAMCS1 = AVALID & (ABUS==RAMBANK1) # !AVALID & RAMCS1
       # (ABUS==RAMBANK1) & RAMCS1;
RAMCS2 = AVALID & (ABUS==RAMBANK2) # !AVALID & RAMCS2
       # (ABUS==RAMBANK2) & RAMCS2;

end latchdec
```

After seeing how easy it is to build S-R and D latches using combinational PLDs, you might be tempted to go further and try to build an edge-triggered D flip-flop. Although this is possible, it is expensive because an edge-triggered flip-flop has four internal states and thus two feedback loops, consuming two PLD outputs. Furthermore, the setup and hold times and propagation delays of such a flip-flop would be quite poor compared to those of a discrete flip-flop in the same technology. Finally, as we discussed in Section 7.10.6, the flow tables of all edge-triggered flip-flops contain essential hazards, which can be masked only by controlling path delays, difficult in a PLD-based design.

8.2.7 Registers and Latches in VHDL

Register and latch circuits can be specified using structural VHDL. For example, Table 8-3 is a structural VHDL program corresponding to the D latch circuit of Figure 7-12 on page 441. However, writing structural programs is not really our motivation for using VHDL; our goal is to use behavioral programs to model the operation of circuits more intuitively.

Table 8-4 is a process-based behavioral architecture for the D latch that requires, in effect, just one line of code to describe the latch's behavior. Note that the VHDL compiler "infers" a latch from this description—since the code

Table 8-3 VHDL structural program for the D latch in Figure 7-12.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity Vdlatch is
  port (D, C: in STD_LOGIC;
        Q, QN: buffer STD_LOGIC );
end Vdlatch;

architecture Vdlatch_s of Vdlatch is
  signal DN, SN, RN: STD_LOGIC;
  component inv port (I: in STD_LOGIC; O: out STD_LOGIC ); end component;
  component nand2b port (I0, I1: in STD_LOGIC; O: buffer STD_LOGIC ); end component;
begin
  U1: inv port map (D,DN);
  U2: nand2b port map (D,C,SN);
  U3: nand2b port map (C,DN,RN);
  U4: nand2b port map (SN,QN,Q);
  U5: nand2b port map (Q,RN,QN);
end Vdlatch_s;
```

Table 8-4 VHDL behavioral architecture for a D latch.

```
architecture Vdlatch_b of Vdlatch is
begin
process(C, D, Q)
  begin
    if (C='1') then Q <= D; end if;
    QN <= not Q;
  end process;
end Vdlatch_b;
```

inferred latch

doesn't say what to do if C is not 1, the compiler creates an *inferred latch* to retain the value of Q between process invocations. In general, a VHDL compiler infers a latch for a signal that is assigned a value in an `if` or `case` statement if not all input combinations are accounted for.

`event` attribute

In order to describe edge-triggered behavior of flip-flops, we need to use one of VHDL's predefined signal attributes, the *`event`* attribute. If "SIG" is a signal name, then the construction "`SIG'event`" returns the value `true` at any delta time when SIG changes from one value to another, and `false` otherwise.

Using the `event` attribute, we can model a positive-edge triggered flip-flop as shown in Table 8-6. In the `IF` statement, "`CLK'event`" returns true on any clock edge, and "`CLK='1'`" ensures that D is assigned to Q only on a *rising* edge. Note that the process sensitivity list includes only CLK; changes on D at other times are not relevant in this functional model.

BUFFS 'N' STUFF Note that in Table 8-3 we defined the type of `Q` and `QN` to be `buffer` rather than `out`, since these signals are used as inputs as well as outputs in the architecture definition. Then we had to define a special 2-input NAND gate `nand2b` with output type `buffer`, to avoid having a type mismatch (`out` vs. `buffer`) in the component instantiations for `U4` and `U5`. Alternatively, we could have used internal signals to get around the problem as shown in Table 8-5. As you know by now, VHDL has many different ways to express the same thing.

Table 8-5 Alternative VHDL structural program for the D latch in Figure 7-12.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity Vdlatch is
  port (D, C: in STD_LOGIC;
        Q, QN: out STD_LOGIC );
end Vdlatch;

architecture Vdlatch_s2 of Vdlatch is
  signal DN, SN, RN, IQ, IQN: STD_LOGIC;
  component inv port (I: in STD_LOGIC; O: out STD_LOGIC ); end component;
  component nand2 port (I0, I1: in STD_LOGIC; O: out STD_LOGIC ); end component;
begin
  U1: inv port map (D,DN);
  U2: nand2 port map (D,C,SN);
  U3: nand2 port map (C,DN,RN);
  U4: nand2 port map (SN,IQN,IQ);
  U5: nand2 port map (IQ,RN,IQN);
  Q <= IQ; QN <= IQN;
end Vdlatch_s2;
```

Table 8-6 VHDL behavioral model of an edge-triggered D flip-flop.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity Vdff is
  port (D, CLK: in STD_LOGIC;
        Q: out STD_LOGIC );
end Vdff;

architecture Vdff_b of Vdff is
begin
process(CLK)
  begin
    if (CLK'event and CLK='1') then Q <= D; end if;
  end process;
end Vdff_b;
```

Table 8-7 VHDL model of a 74x74-like D flip-flop with preset and clear.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity Vdff74 is
  port (D, CLK, PR_L, CLR_L: in STD_LOGIC;
        Q, QN: out STD_LOGIC );
end Vdff74;

architecture Vdff74_b of Vdff74 is
signal PR, CLR: STD_LOGIC;
begin
process(CLR_L, CLR, PR_L, PR, CLK)
  begin
    PR <= not PR_L; CLR <= not CLR_L;
    if (CLR and PR) = '1' then Q <= '0'; QN <= '0';
    elsif CLR = '1' then Q <= '0'; QN <= '1';
    elsif PR = '1' then Q <= '1'; QN <= '0';
    elsif (CLK'event and CLK='1') then Q <= D; QN <= not D;
    end if;
  end process;
end Vdff74_b;
```

Table 8-8 VHDL model of a 16-bit register with many features.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity Vreg16 is
  port (CLK, CLKEN, OE_L, CLR_L: in STD_LOGIC;
        D: in STD_LOGIC_VECTOR(1 to 16);      -- Input bus
        Q: out STD_ULOGIC_VECTOR (1 to 16) ); -- Output bus (three-state)
end Vreg16;

architecture Vreg16 of Vreg16 is
signal CLR, OE: STD_LOGIC;  -- active-high versions of signals
signal IQ: STD_LOGIC_VECTOR(1 to 16); -- internal Q signals
begin
process(CLK, CLR_L, CLR, OE_L, OE, IQ)
  begin
    CLR <= not CLR_L;  OE <= not OE_L;
    if (CLR = '1') then IQ <= (others => '0');
    elsif (CLK'event and CLK='1') then
      if (CLKEN='1') then IQ <= D; end if;
    end if;
    if OE = '1' then Q <= To_StdULogicVector(IQ);
    else Q <= (others => 'Z'); end if;
  end process;
end Vreg16;
```

The D-flip-flop model can be augmented to include asynchronous inputs and a complemented output as in the 74x74 discrete flip-flop, as shown in Table 8-7. This more detailed functional model shows the non-complementary behavior of the `Q` and `QN` outputs when preset and clear are asserted simultaneously. However, it does not include timing behavior such as propagation delay and setup and hold times, which are beyond the scope of the VHDL coverage in this book.

Larger registers can of course be modeled by defining the data inputs and outputs to be vectors, and additional functions can also be included. For example, Table 8-8 models a 16-bit register with three-state outputs and clock-enable, output-enable, and clear inputs. An internal signal vector `IQ` is used to hold the flip-flop outputs, and three-state outputs are defined and enabled as in Section 5.6.4.

SYNTHESIS RESTRICTIONS

In Table 8-8, the first `elsif` statement theoretically could have included all of the conditions needed to assign `D` to `IQ`. That is, it could have read "`elsif (CLK'event) and (CLK='1') and (CLKEN='1') then ...`" instead of using a nested `if` statement to check `CLKEN`. However, it was written as shown for a very pragmatic reason.

Only a subset of the VHDL language can be synthesized by the VHDL compiler that was used to prepare this chapter; this is true of any VHDL compiler today. In particular, use of the "`event`" attribute is limited to the form shown in the example, and a few others, for detecting simple edge-triggered behavior. This gets mapped into edge-triggered D flip-flops during synthesis. The nested `IF` statement that checks `CLKEN` in the example leads to the synthesis of multiplexer logic on the D inputs of these flip-flops.

8.3 Sequential PLDs

8.3.1 Bipolar Sequential PLDs

PAL16R8

The *PAL16R8*, shown in Figure 8-17, is representative of the first generation of sequential PLDs, which used bipolar (TTL) technology. This device has eight primary inputs, eight outputs, and common clock and output-enable inputs, and fits in a 20-pin package.

The PAL16R8's AND-OR array is exactly the same as the one found in the PAL16L8 combinational PLD. However, the PAL16R8 has edge-triggered D flip-flops between the AND-OR array and its eight outputs, O1–O8. All of the flip-flops are connected to a common clock input, CLK, and change state on the rising edge of the clock. Each flip-flop drives an output pin through a 3-state buffer; the buffers have a common output-enable signal, OE_L. Notice that, like