10. Using Flash Memory to Configure FPGAs

CF52010-2.3

Introduction

As Altera introduces higher-density FPGAs, the configuration bit stream size also increases. As a result, designs require more configuration devices to store the data and configure these devices. As an alternative, flash memory can be used to store configuration data. A flash memory controller is required to read and write to the flash memory and perform configuration. You can use a MAX® 3000A or MAX 7000 device to implement the flash memory controller.

Device Configuration Using Flash Memory & MAX 3000A Devices

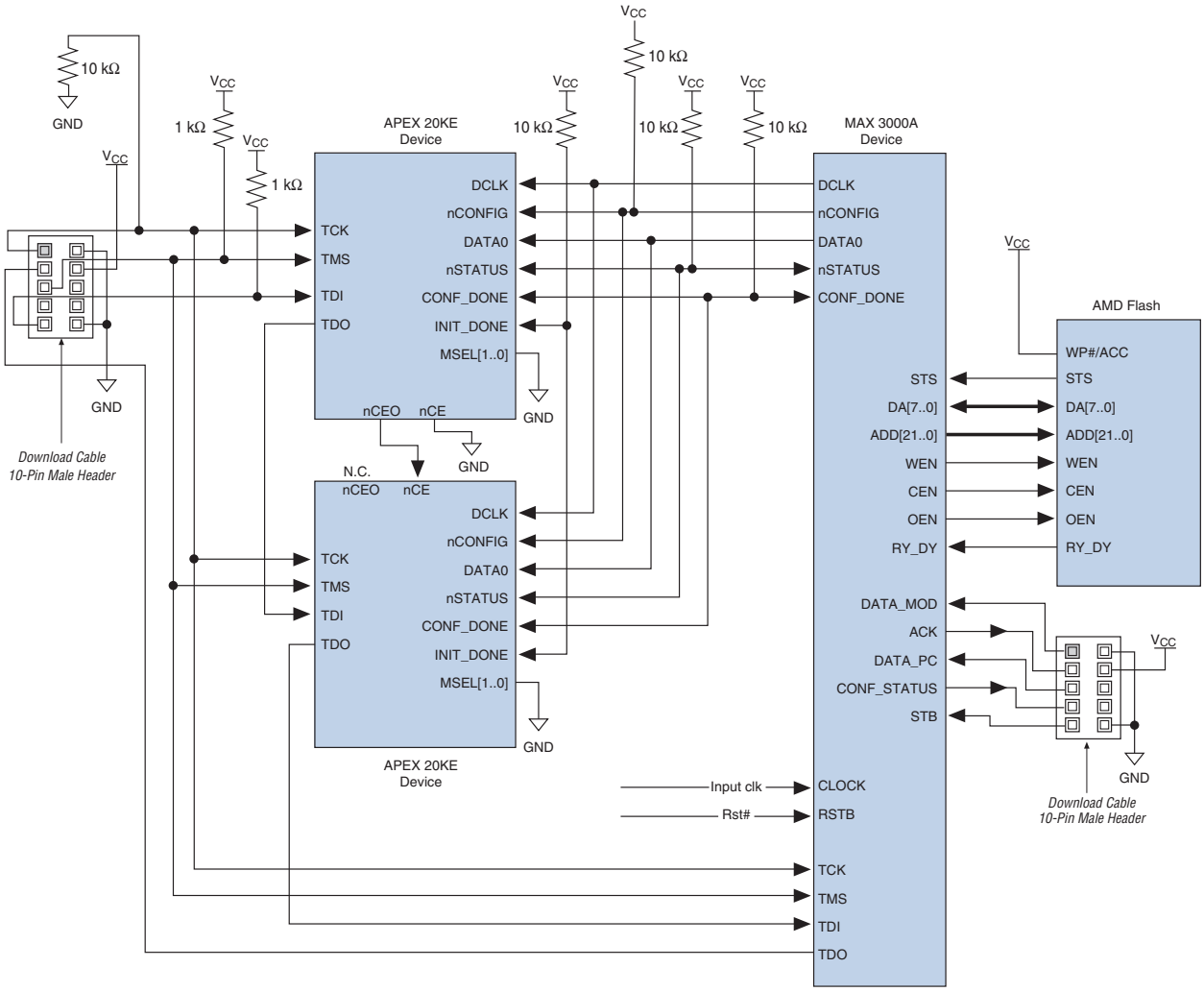
The flash memory controller can interface with a PC or microprocessor to receive configuration data via a parallel port (Figure 10–1). The controller generates a programming command sequence to program the flash memory and extract configuration data to configure FPGAs.

The flash memory controller supports various commands such as:

- Programming the flash memory
- Configuring FPGAs

A reference design that uses the MAX 3000 device is available on the Altera web site. The reference design can be used with an AMD or Fujitsu flash.

Figure 10–1. Configuring an FPGA through Flash Memory & MAX 3000A Controller



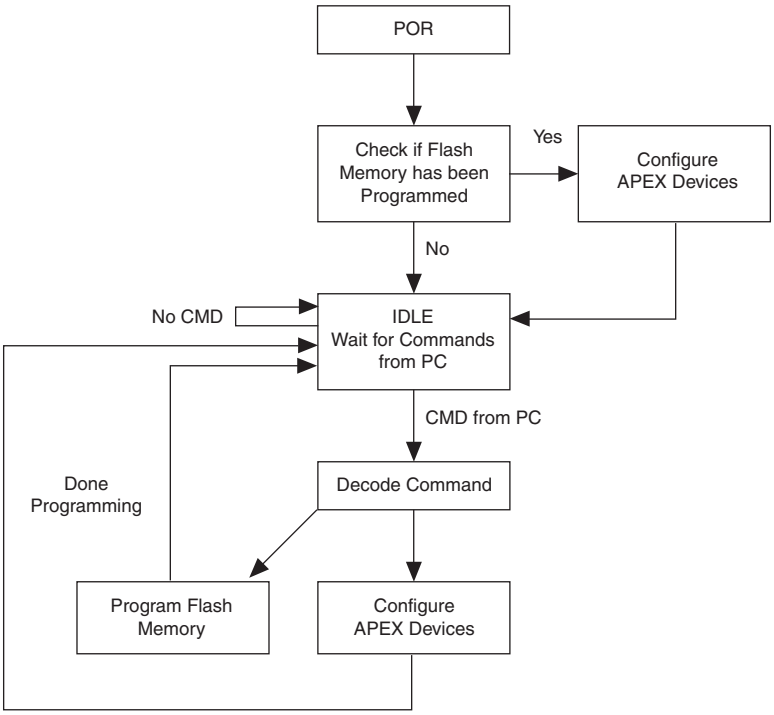
Flash Memory Controller Design Specification

The controller will check to see if the flash memory is programmed successfully after the board powers up. If the flash memory is programmed successfully, then the controller configures the FPGAs. If flash memory is not programmed successfully, then the controller waits for commands from the PC or microprocessor. The receiver decodes the commands it receives from the PC or microprocessor as one of the following:

- Program flash memory
- Configure FPGA

After a command is executed, the controller returns to idle mode and waits for the next command. Figure 10–2 shows the controller state machine.

Figure 10–2. Flash Memory Controller State Machine



Flash Memory Controller Functionality

The controller writes a byte to a special location in the flash memory when it programs the memory. After POR, the controller checks this special location in the flash memory to see if the byte is written there or not.

If the byte is written, then the flash memory has been programmed and the controller can proceed to configuring the FPGAs by reading data from the flash memory. If this byte is not there or the value is not as expected, the controller will go idle and wait to be programmed by the PC or microprocessor.

Getting Data from the PC or Microprocessor

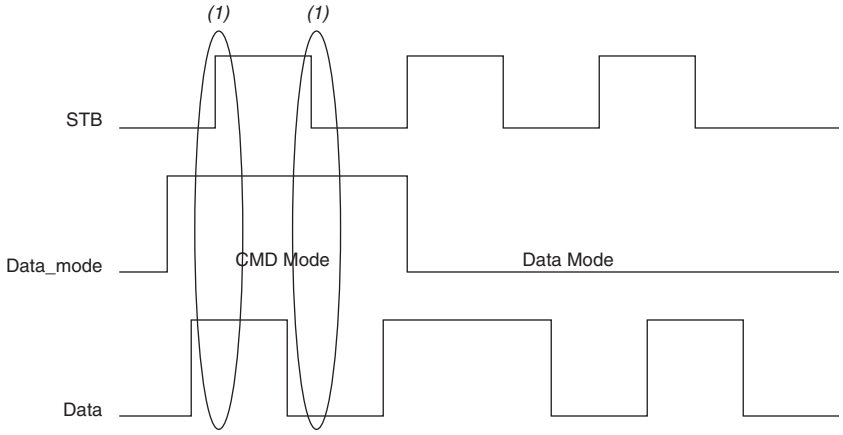
The PC or microprocessor uses the parallel port to interface with the controller. There are two types of signals involved in this connection (see Figure 10–3), a 3-bit input signal from the PC or microprocessor to the controller, and a 2-bit output signal from the controller to the PC or microprocessor. The input signal includes the following three signals:

- `STB`: Strobe signal from the PC or microprocessor to indicate that the PC or microprocessor's data is valid.
- `data_mode`: Indicates whether the controller is in command mode or data mode. When `data_mode` is high, the controller is in command mode; when `data_mode` is low, the controller is in data mode.
- `data`: Content of this signal depends on `data_mode`. It can be data for command mode or data mode.

The output signal contains the following two signals:

- ACK: Acknowledge signal is a handshaking signal from the controller to the PC or microprocessor.
- conf_status: Indicates configuration status.

Figure 10–3. Getting Data from PC or Microprocessor

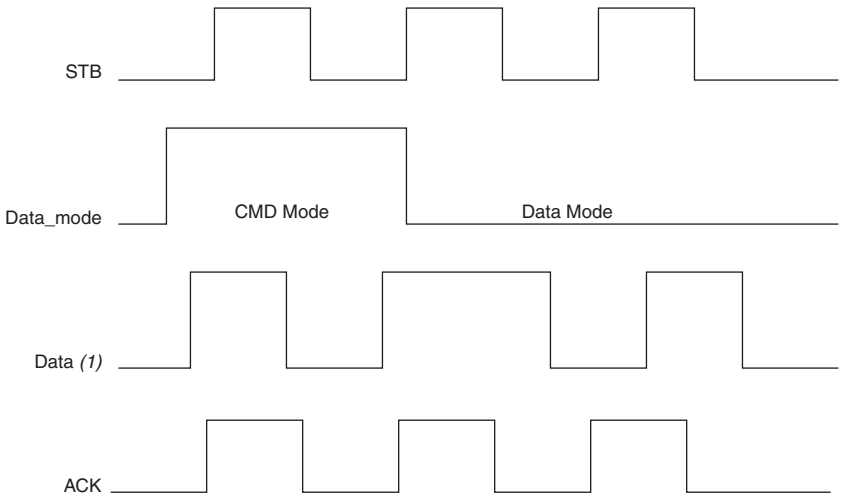


Note to Figure 10–3:

(1) Data is sent on both positive and negative edges of the STB signal.

The controller receives a bit of data or a command from the PC or microprocessor on the rising and falling edges of the STB signal. After receiving this data, the controller will send an acknowledgement signal to the PC or microprocessor to initiate sending of the next bit of data. The acknowledge signal (ACK) should be the same logic level as the last received STB signal. By de-asserting ACK, the controller can stop the PC or microprocessor from sending data. Figure 10–4 shows the STB and ACK relationship.

Figure 10–4. Sending Acknowledge Signal (ACK) to PC or Microprocessor



Note to Figure 10–4:

(1) One bit of data is received at each STB signal edge (both positive and negative).

Programming Flash Memory

After receiving a command from the PC or microprocessor, the controller first erases and then starts programming the flash memory. A separate state machine is required to generate a programming command sequence and programming pulse width.

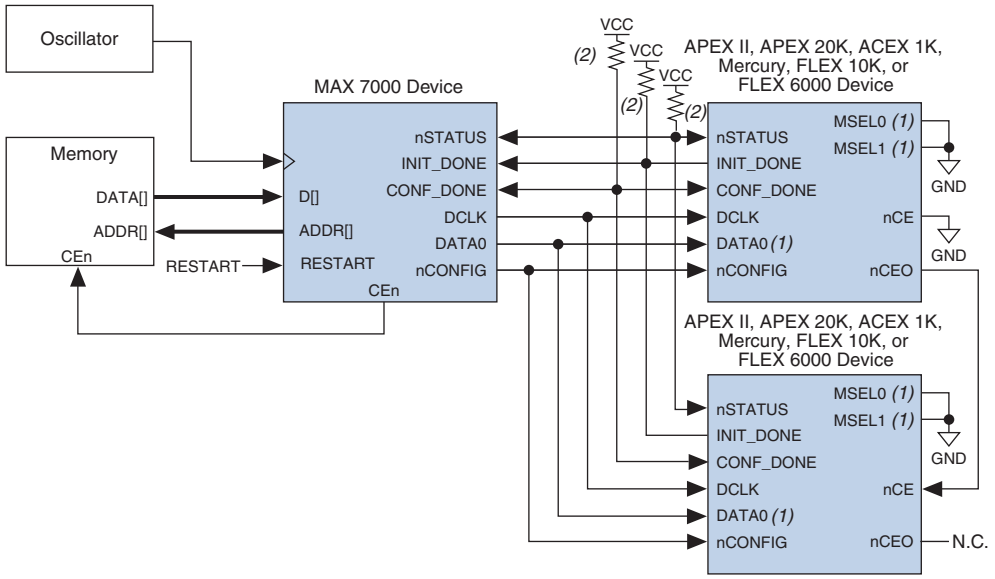
While programming the flash memory, the controller must check if a command (`data_mode=1`) has been received or not. A command indicates the end of data from the PC or microprocessor, and the controller will exit the Program_Flash_memory state and go into idle mode.

Another state machine is required to read and serialize byte data from the flash memory and generate DCLK and DATA0. The controller needs to monitor CONF_DONE signals from the FPGAs to determine if configuration is complete. When configuration is done, the controller exits the configure state and goes back to idle mode.

Device Configuration Using Flash Memory & MAX 7000 Devices

Figure 10–5 shows the schematic for this configuration scheme with a MAX 7000 device. Two sample design files for the MAX 7000 device (Design File for Configuring APEX™ 20K Devices and Design File for Configuring FLEX® 10K and FLEX 6000 Devices) are available on the Altera web site.

Figure 10–5. Device Configuration Using External Memory & a MAX 7000 Device



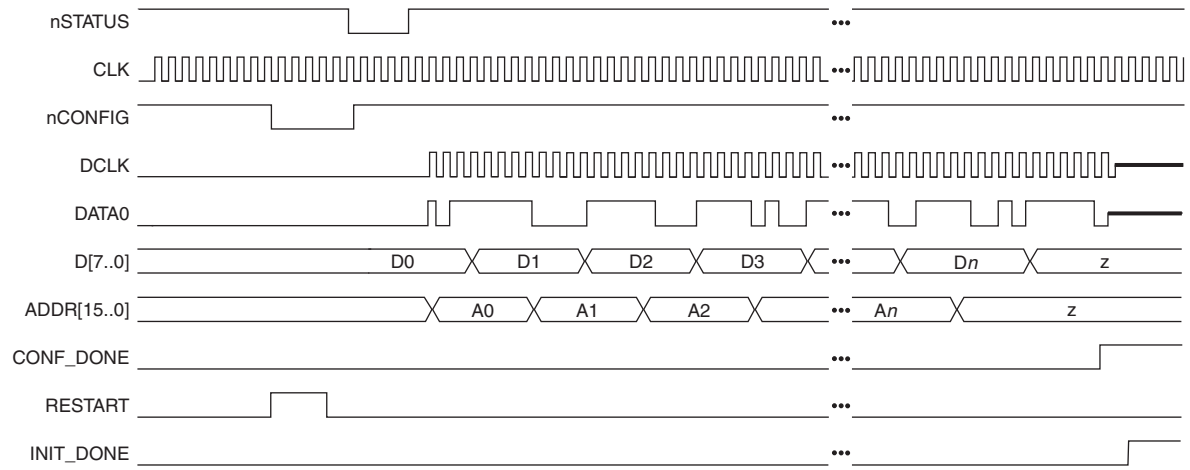
Notes to Figure 10–5:

(1) FLEX 6000 devices have a single MSEL pin, which is tied to ground, and its DATA0 pin is renamed DATA.

(2) All pull-up resistors are 1 kΩ. On APEX 20KE and APEX 20KC devices, pull-up resistors for nSTATUS, CONF_DONE, and INIT_DONE pins should be 10 kΩ.

Figure 10–6 shows the timing waveform for configuring an APEX™ II, APEX 20K, Mercury™, ACEX® 1K, FLEX 10K, or FLEX 6000 device using external memory and a MAX 7000 device.

Figure 10–6. Timing Waveform for Configuration Using External Memory & a MAX 7000 Device



Design Example Using MAX 3000 Devices

A MAX® 3000 device can be used to stream the data from the flash memory into a large FPGA. This configuration technique allows faster configuration times. Since a fixed-frequency oscillator (or any available clock on the system) is used to generate the clock for the configuration, the clock frequency can be as high as 57 MHz (the maximum for an APEX 20KE device).

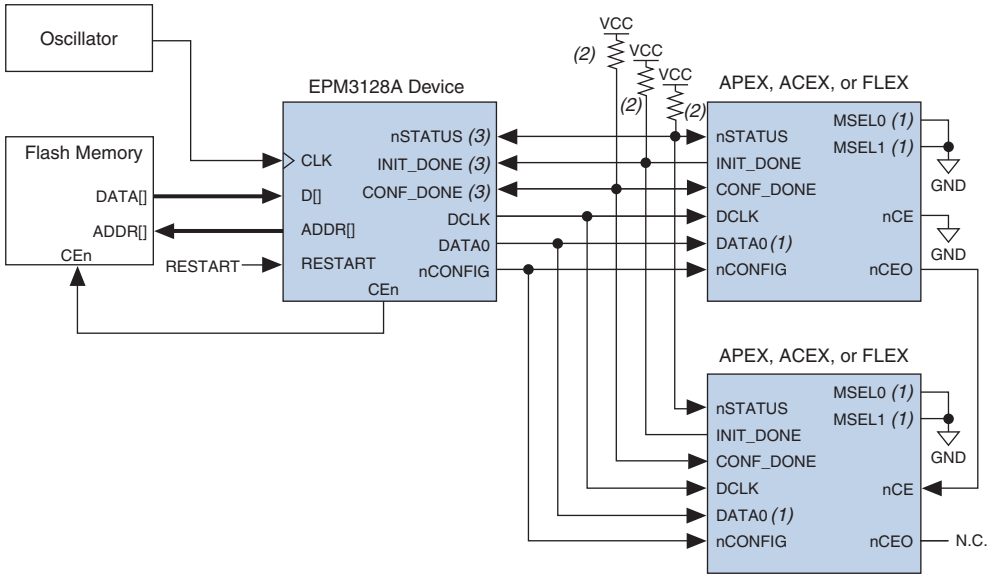
Flash memory is a type of nonvolatile memory that can be used as a data storage device. Flash memory can be erased and reprogrammed in units of memory called blocks.

This section describes how to configure an FPGA with flash memory. By using a MAX 3000 device to configure higher density FPGAs, the flash memory can store configuration data and the MAX 3000 device can serialize and transmit the data to the FPGA. This configuration technique can be used with APEX, ACEX, or FLEX devices.

Configuring FPGAs

Figure 10–7 shows a device that uses an EPM3128A device and flash memory to configure the FPGAs.

Figure 10–7. Device Configuration Using Flash Memory & EPM3128A Device



Notes to Figure 10–7

(1) FLEX 6000 devices have a single `MSEL` pin, which is tied to ground. Additionally, its `DATA0` pin is renamed `DATA`.

(2) Pull-up resistors are 1 kΩ except for APEX 20KE devices. For APEX 20KE devices, pull up resistors are 10 kΩ.

(3) The `nSTATUS`, `CONF_DONE`, and `INIT_DONE` pins are open-drain on the APEX, ACEX, and FLEX devices. The corresponding pins on the EPM3128A should also be open_drain.

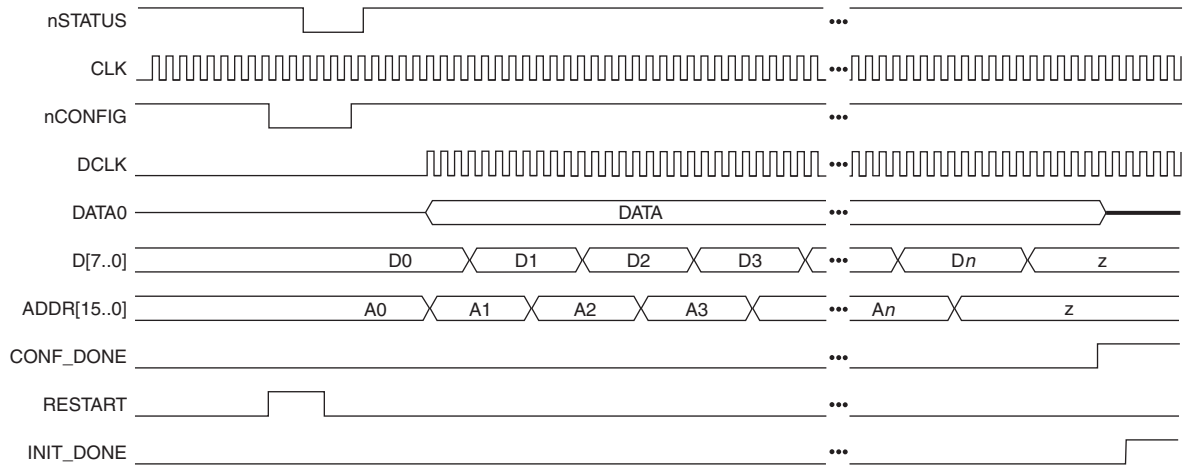
A VHDL design file called **MAXconfig**, shown in the "Configuration Design File" section, allows an EPM3128A device to control the configuration process. The **MAXconfig** design configures the FPGA using the configuration data stored in the attached flash memory. The **MAXconfig** design contains a sequencer and an address generator, which drives the correct data to the FPGA's programming pins. The **MAXconfig** design file is available on the Altera web site at *http://www.altera.com/literature/wp/maxconfig.txt*.

When the **MAXconfig** design is reset, the **MAXconfig** design reads the data from the flash memory, one byte at a time. The **MAXconfig** design then serializes and sends the data to the APEX, ACEX, or FLEX device. The serialized data is sent to the FPGA using the passive serial interface pins such as DCLK, DATA, nSTATUS, INIT_DONE, and nCONFIG. Since the passive serial mode is used, the flash pins are not directly connected to the APEX, ACEX, or FLEX device.

Flash memory can be programmed prior to being put onto a board with standard programming equipment or it can be programmed in-system by a processor or test equipment. Since different flash memories have different algorithms, consult the flash memory data sheet for programming information.

Figure 10–8 shows a configuration timing waveform of an EPM3128A device downloading data to an APEX, ACEX, or FLEX device.

Figure 10–8. Configuration Timing Waveform



Configuration Design File

This section shows the MAXconfig design file that controls the configuration process on APEX, ACEX, or FLEX devices:

Example 10–1. MAXconfig design file (Part 1 of 4)

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity MAXconfig is
port
    (
    clock  : in   std_logic;
    init_done: in std_logic;
    nStatus: in   std_logic;
    D      : in   std_logic_vector(7 downto 0);
    restart: in   std_logic;
    Conf_Done: in std_logic;

    Data0  : out  std_logic;
    Dclk   : out  std_logic;
    nConfig: bufferstd_logic;

--To increase the size of the memory, change the size of std_logic_vector for ADDR output
and --std_logic_vector signal inc:

    ADDR   : out  std_logic_vector(15 downto 0);
    CEn    : out  std_logic);
-- The polarity of the CEn signal is determined by the type of Flash device
end;

architecture rtl of MAXconfig is

--The following encoding is done in such way that the LSB represents the nConfig signal:

constant start      :std_logic_vector(2 downto 0) := "000";
constant wait_nCfg_8us:std_logic_vector(2 downto 0) := "100";
constant status     :std_logic_vector(2 downto 0) := "001";
constant wait_40us  :std_logic_vector(2 downto 0) := "101";
constant config     :std_logic_vector(2 downto 0) := "011";
constant init       :std_logic_vector(2 downto 0) := "111";


signal pp           :std_logic_vector(2 downto 0);
signal count        :std_logic_vector(2 downto 0);
signal data0_int, dclk_int:std_logic;
signal inc          :std_logic_vector(15 downto 0);
signal div          :std_logic_vector(2 downto 0);
signal waitd        :std_logic_vector(11 downto 0);
--The width of signal 'waitd' is determined by the frequency. For 57 MHz (APEX 20KE
devices), --'waitd' is 12 bits. For 33 MHz (FLEX 10KE and ACEX devices) 'waitd' is 11
bits. To calculate --the width of the 'waitd' signal fordifferent frequencies, calculate
the following:
--(multiply tcf2ck * clock frequency)+ 40
--Then convert this value to binary to obtain the width.
--For example, for 33 MHz (FLEX 10KE & ACEX devices), converting 1360 ((40us *
33MHz)+40=1360)
--to binary code, the 'waitd' is an 11-bit signal. So signal 'waitd' will be:
--signal waitd      :std_logic_vector(10 downto 0);

begin
--The following process is used to divide the CLOCK:
    PROCESS (clock,restart)
    begin
```

Example 10–1. MAXconfig design file (Part 2 of 4)

```
      if restart = '0' then
         div <= (others => '0');
      else
         IF (clock'EVENT AND clock = '1') THEN
            div <= div + 1;
         end if;
      end if;
   END PROCESS;

   PROCESS (clock,restart)
   begin
      if restart = '0' then
         pp<=start;
         count <= (others => '0');
         inc   <= (others => '0');
         waitd <= (others => '0');
      else
      if clock'event and clock='1' then

--The following test is used to divide the CLOCK. The value compared to must be such that
the
--condition is true at a maximum rate of 57 MHz (tclk = 17.5 ns min) for APEX 20KE devices
--and at a maximum rate of 33 MHz (tclk=30ns min) for FLEX 10KE or ACEX devices.

          if (div = 7) then
             case pp is
             when start =>
                       count <= (others => '0');
                       inc   <= (others => '0');
                       waitd <= (others => '0');
                       pp <= wait_nCfg_8us;
--This state is used in order to verify the tcfg timing (nCONFIG low pulse width).
--Tcfg = 8µs => min= 456 clock cycle of a 57 MHz clock (APEX 20KE devices). For different
--clocks, multiply 8µs to clock frequency. For example, for 33MHz (FLEX 10KE or ACEX
devices) this --value is 8*33=264. This clock is CLOCK divided by the divider -div-.
             when wait_nCfg_8us =>
                       count <= (others => '0');
                        inc   <= (others => '0');
                        waitd <= waitd + 1;
                        if waitd = 456 then
--For 33 MHz FLEX 10KE or ACEX devices this line is: if waitd = 264 then

                    pp <= status;
                       end if;

--This state is used to have nCONFIG high.
             when status =>
                count <= (others => '0');
                inc   <= (others => '0');
                waitd <= (others => '0');
                pp <= wait_40us;
```

Example 10–1. MAXconfig design file (Part 3 of 4)

```
--This state is used to generate the tcf2ck timing (nCONFIG high to first rising edge on
DCLK).
--Tcf2ck = 40µs min => 2280 clock cycles of a 57MHz (APEX 20KE) clock. This clock is CLOCK
--divide by the divider -div-
--Tcf2ck = 40µs min => 1320 clock cycles of a 33MHz (FLEX 10KE/ACEX) clock. This clock
is CLOCK --divided by the divider -div-)
--For any other clock frequency, multiply tcf2ck * clock frequency.

             when wait_40us =>
                         count <= (others => '0');
                         inc   <= (others => '0');
                         waitd <= waitd + 1;
                        if waitd = 2280 then
--For 33 MHz (FLEX 10KE or ACEX devices), this line is:    if waitd = 1320 then

                        pp <= config;
                end if;

--This state is used to increment the memory address. In the same state when
--the Conf_Done is high clock cycles are added in order to have the initialization
completed.

             when config =>
                count <= count + 1;
                if Conf_Done='1' then
                   waitd <= waitd + 1;
                end if;
                if count=7 then
                   inc <= inc + 1;
                end if;
                if waitd = 2320 then
--Modification: Add 40 clock cycles. For APEX 20KE devices, it is 2280+40=2320
--For FLEX 10KE and ACEX devices, it is 1320+40=1360. This line becomes:    if waitd=
1360 then

                pp<= init;
                end if;

             when init =>
                count <= (others => '0');
                inc   <= (others => '0');
                waitd <= (others => '0');
                if nStatus = '0' then
                   pp <= start;
                else
                   pp <= init;
                end if;

             when others =>
                pp <= start;
             end case;
          else
             pp <= pp;
             inc <= inc;
             count <= count;
          end if;
       end if;
   end if;
   end PROCESS;

   dclk_int <= div(2) when pp=config else '0';
```

Example 10–1. MAXconfig design file (Part 4 of 4)

```
--The following process is used to serialize the data byte :
   PROCESS (count,D,pp)
   begin
      if pp=config then
             case count is
             when "000" => data0_int <= D(0);
             when "001" => data0_int <= D(1);
             when "010" => data0_int <= D(2);
             when "011" => data0_int <= D(3);
             when "100" => data0_int <= D(4);
             when "101" => data0_int <= D(5);
             when "110" => data0_int <= D(6);
             when "111" => data0_int <= D(7);
             when others => null;
             end case;
      else
             data0_int <= '0';
      end if;
   end PROCESS;

   nConfig <= pp(0);
   CEn <= not nconfig;
   Dclk  <= '0' when pp(1)='0' else dclk_int;
   Data0 <= '0' when pp(1)='0' else data0_int;
   ADDR <= inc;
   end;
```

Conclusion

Altera provides high-density FPGAs that require larger configuration files. By using a flash memory device and an EPM3128A device in a design, a FPGA can be quickly configured.

Document Revision History

Table 10–1 shows the revision history for this document.

Table 10–1. Document Revision History

Date and Revision	Changes Made	Summary of Changes
October 2008, version 2.3	■ Updated "Configuring FPGAs" section. ■ Updated new document format.	—
April 2007, version 2.2	■ Added document revision history.	—
August 2005, version 2.1	■ Removed active cross references referring to document outside Chapter 10.	—
July 2004, version 2.0	■ Removed Intel flash reference design. ■ Updated Figure 10–1. ■ Removed Flash Memory Content Verification section.	—
September 2003, version 1.0	■ Initial Release.	—