第 5 章

CHAPTER 5

基于 IP 的设计

本章主要介绍 Quartus Prime 中可重复利用的参数化模块库(LPM)设计资源,讲述如何配置和实例引用参数化模块等 IP 资源。希望读者通过这一章的内容,能够利用 Quartus Prime 软件工具提供的参数化模块资源对常用电路进行高效快速的 HDL 设计。

5.1 IP 核

IP(Intellectual Property)原指知识产权、著作权等,在 IC 设计领域通常被理解为实现某种功能的设计。IP 核则是完成某种常用但是比较复杂的算法或功能(如 FIR 滤波器、SDRAM 控制器、PCI 接口等),并且参数可修改的电路模块,又称为 IP 模块。随着 CPLD/FPGA 的规模越来越大,设计越来越复杂,越来越多的人开始认识到 IP 核以及 IP 复用技术的优越性,并努力推动 IP 复用设计技术的发展。

根据实现的不同,IP 核可以分为三类:完成行为域描述的软核(Soft Core),完成结构域描述的固核(Firm Core)和基于物理域描述并经过工艺验证的硬核(Hard Core)。三种 IP 核的特点比较见表 5-1。不同的用户可以根据自己的需要订购不同的 IP 产品。

表 5-1 三种 IP 核的特点比较

	软(soft)IP 核	固(firm)IP 核	硬(hard)IP 核
描述内容	模块功能	模块逻辑结构	物理结构
提供方式	HDL 文档	门电路级网表,对应具体工艺网表	电路物理结构掩模版图和全套工艺文件
优点	灵活,可移植	介于两者之间	后期开发时间短
缺点	后期开发时间长		灵活性差,不同工艺难移植

(1) 软核:用硬件描述语言(HDL)的形式描述功能的 IP 核,与具体的实现技术无关。软核是集成电路设计的高层描述,灵活性大。软核可以用于多种制作工艺,在新功能模块中重新配置,以实现重定目标电路。此类 IP 核只通过了功能和时序验证,其他的实现内容及相关测试等均需要使用者自己完成,因此软核 IP 用户的后继工作较大。

(2) 硬核:IP 硬核是基于半导体工艺的物理设计,已有固定的拓扑布局和具体工艺,并已经过工艺验证,具有可保证的性能。提供给用户的形式是电路物理结构掩膜版图和全套工艺文件,允许设计者将 IP 快速集成在衍生产品中。因为与工艺相关,硬核 IP 的灵活性

较差。

(3) 固核：在设计阶段介于软核和硬核之间的 IP 核。除了完成软核所有设计外，固核还完成了门级电路综合和时序仿真等环节，以 RTL 描述和可综合网表的形式提交。固核的用户使用灵活性介于软核和硬核之间。

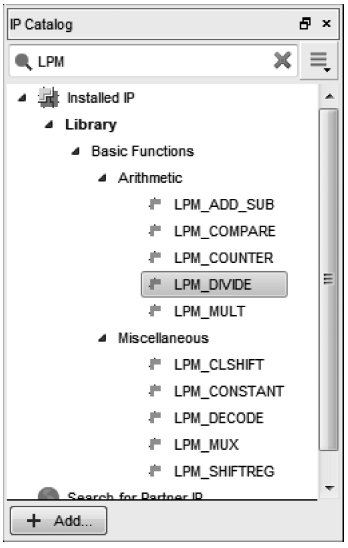

图 5-1　LPM 种类选择界面

Intel 公司以及第三方 IP 合作伙伴给用户提供了很多可用的功能模块，它们基本可以分为两类：免费的 LPM 宏功能模块(Megafunctions/LPM)和需要授权使用的 IP 知识产权(MEGACORE)。这两者只是从实现的功能上区分，使用方法上则基本相同。

LPM 宏功能模块是一些复杂或高级的构建模块，可以在 Quartus Prime 设计文件中和门、触发器等基本单元一起使用，这些模块的功能一般都是通用的，例如 Counter、FIFO、RAM 等。Altera 提供的可参数化 LPM 宏功能模块和 LPM 函数均为 Altera 器件结构做了优化，而且必须使用宏功能模块才可以使用一些 Altera 特定器件的功能，例如存储器、DSP 块、LVDS 驱动器、PLL 电路。

通过菜单 Tools→IP Catalog，并在 IP Catalog 中输入 LPM，会出现 Quartus Prime 软件已安装的 LPM 种类，如图 5-1 所示。通过选择需要的 LPM，单击并进行修改。

5.2　触发器 IP 核的 VHDL 设计应用

触发器(Flip-Flop)是数字电路设计中的基本单元，尤其是 D 触发器，通常被用来做延时和缓存处理。第 4 章给出了利用多个 D 触发器构造移位寄存器和 m 序列发生器的示例。将图 4-29 和图 4-30 给出的移位寄存器和 m 序列发生器结合在一起，可以形成串行输入初始状态的序列发生器，利用原理图方式进行设计，结果如图 5-2 所示。

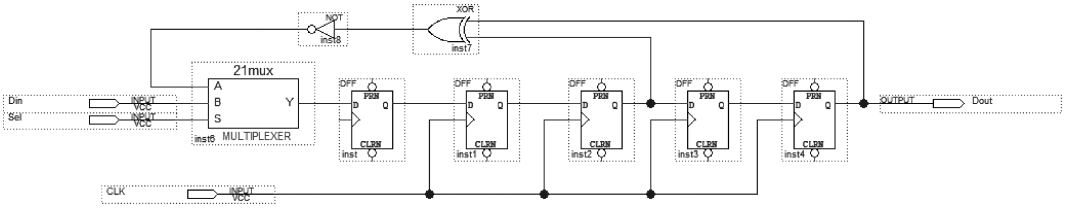

图 5-2　利用触发器构造序列发生器

触发器的延迟功能与移位寄存器功能类似，Altera LPM 宏功能模块中将两种功能结合在一起，用同一个模块实现。

如图 5-3 所示，在原理图输入模式下，可以在 Symbol 界面下，在 megafunctions → storage 下使用宏功能模块 LPM_DFF 完成功能更复杂的 D 触发器。

BDF(原理图)文件中插入 LPM_DFF 后，双击右上角参数列表或者选择右键菜单 Properties 后，可以进行 LPM_DFF 的端口和参数设置，如图 5-4 和图 5-5 所示。

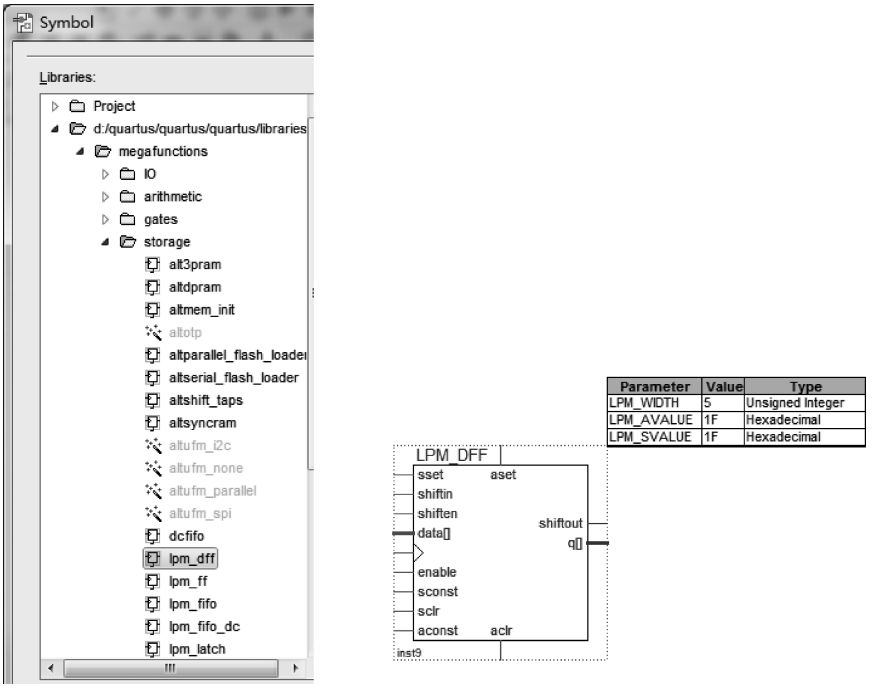

图 5-3　原理图输入方式下的 LPM_DFF

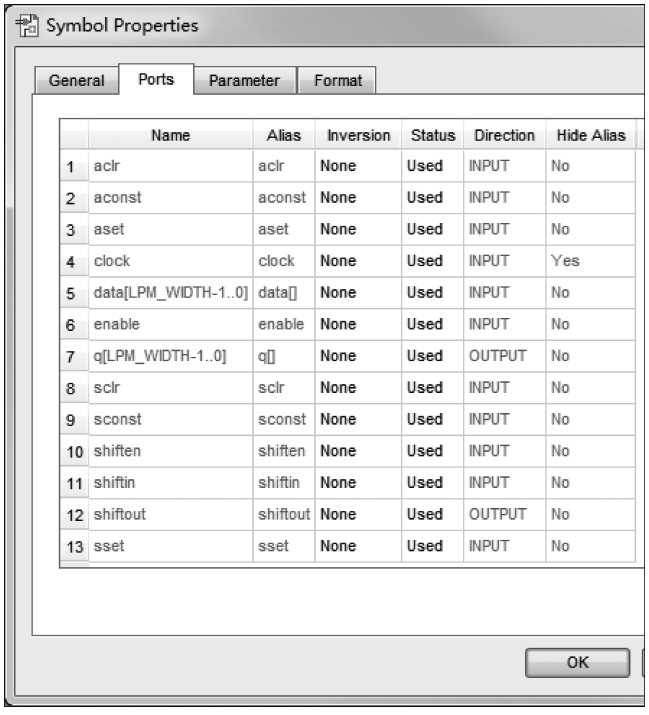

图 5-4　LPM_DFF 端口设置

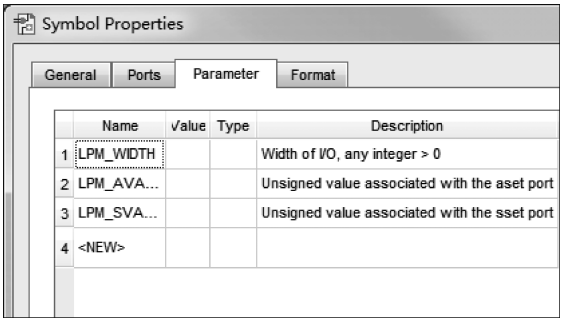

图 5-5　LPM_DFF 参数设置

利用图 5-4 所示页面，对端口的状态设置为使用或不使用，可以改变 LPM_DFF 的端口，从而使相应的功能有效或无效。图 5-5 所示的参数设置，可以确定 D 触发器的级数和初始值等。

通过 MegaWizard Plug-In Manager 同样可以进行 D 触发器设计。在 MegaWizard Plug-In Manager 中，没有 LPM_DFF，而是命名为 LPM_SHIFTREG。在 IP Catalog 栏中输入 LPM_SHIFTREG，并单击，弹出 Save IP Variation 对话框，如图 5-6 所示；选择文件类型为 VHDL，并命名为 LPM_SHIFTREG1，然后单击 OK，打开 MegaWizard Plug-In Manager 界面。

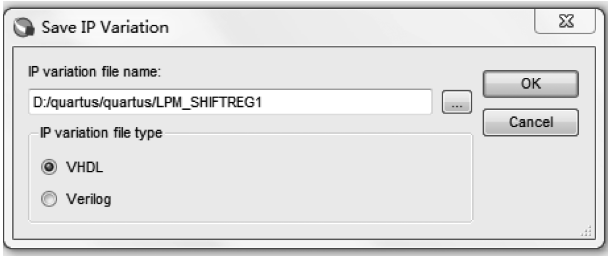

图 5-6　触发器应用设计——LPM_SHIFTREG

在图 5-7 和图 5-8 所示参数设置页面，可以对 LPM_SHIFTREG 进行各种属性设置，这里设置了并行输出 q 的宽度为 5bit，表示内部有 5 级 D 触发器，形成 5 位的移位寄存器。另外还有输出端口的选择和输入端口的配置，如并行输入、输出端口以及同步、异步端口设置等。

比较 LPM_DFF 和 LPM_SHIFTREG 可以看到，二者实现的功能相同，对比分析可以更好地理解各个端口的功能和使用方法。LPM_SHIFTREG 的其他设置可采取默认值，最终可以实现定制的 LPM_SHIFTREG 功能。

编写例 5.1 所示 VHDL 代码，对产生的 5 位 LPM_SHIFTREG 进行调用，可以产生第 4 章 m 序列产生器设计的相同功能。

【例 5.1】　调用 LPM_SHIFTREG 模块形成 m 序列。

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY myShift5Reg IS
  port(clk : in std_logic;
```

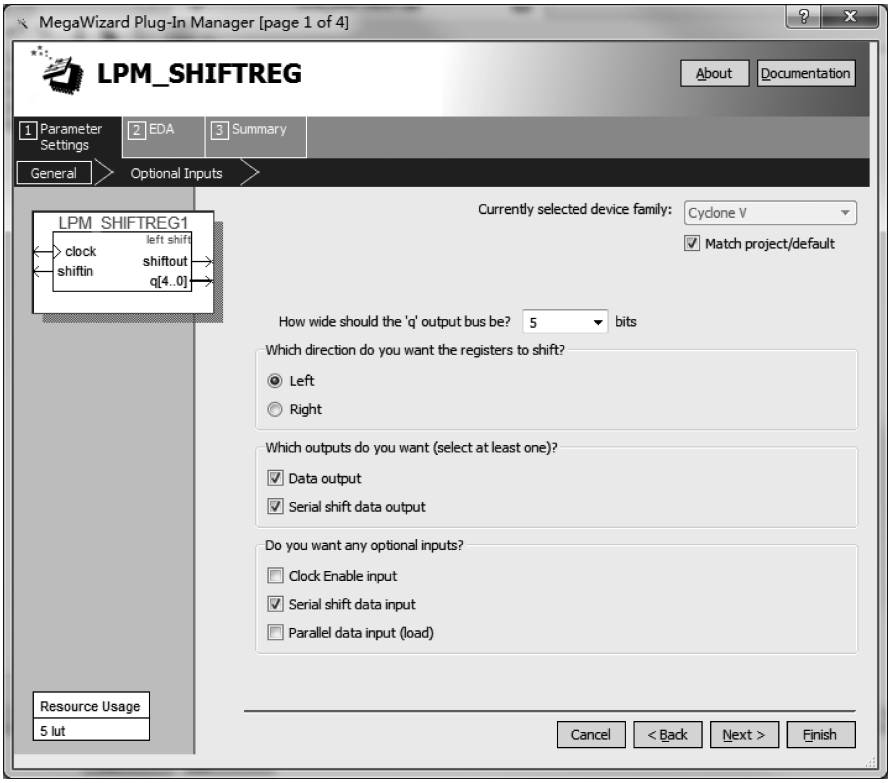

图 5-7　LPM_SHIFTREG 参数设置(1)

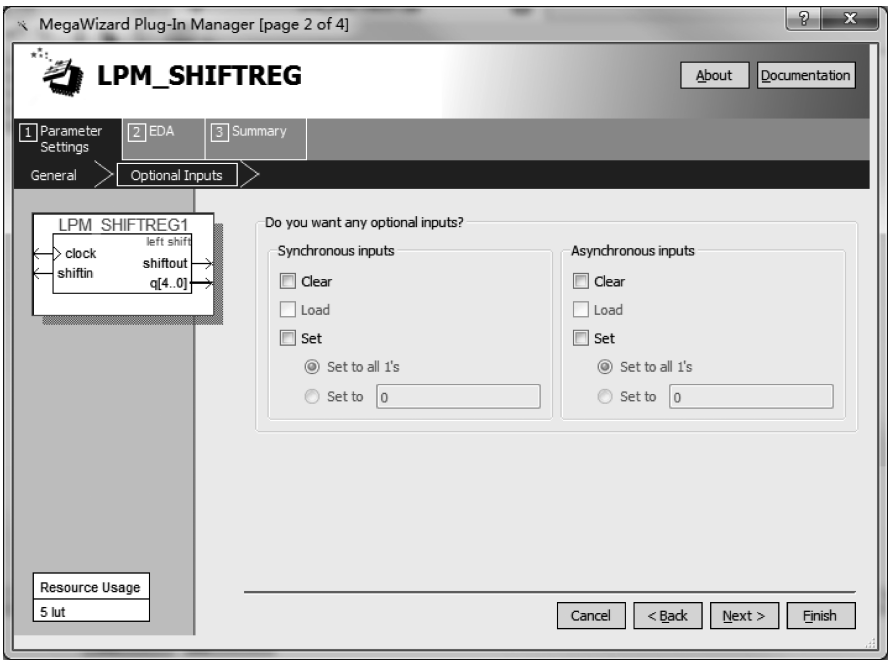

图 5-8　LPM_SHIFTREG 参数设置(2)

```
      qout : buffer STD_LOGIC_VECTOR(0 to 4);
      dout: out std_logic );
END myShift5Reg;
ARCHITECTURE rtl OF myShift5Reg IS

COMPONENT LPM_SHIFTREG1 IS
    PORT
    (
        clock         : IN STD_LOGIC ;
        shiftin       : IN STD_LOGIC ;
        q             : OUT STD_LOGIC_VECTOR (0 to 4);
        shiftout      : OUT STD_LOGIC
    );
END COMPONENT;
begin

mySReg_inst : LPM_SHIFTREG1 PORT MAP (
        clock         => clk,
        shiftin       => not (qout(0) xor qout(2)),
        q             => qout,
        shiftout      => dout
    );
end rtl;
```

5.3　存储器 IP 核的 VHDL 设计应用

存储器是 FPGA 设计中常用的模块之一，包括 RAM、ROM 等。可以通过模板(Template)很快给出完整代码，如例 5.2 给出的最基本单口 RAM 代码就是通过菜单 Edit→Insert Template 获得的。

【例 5.2】 单口 RAM 模板。

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

entity single_port_ram is
    generic
    (   DATA_WIDTH : natural := 8;
        ADDR_WIDTH : natural := 6
    );

    port
    (   clk           : in std_logic;
        addr          : in natural range 0 to 2 ** ADDR_WIDTH - 1;
        data          : in std_logic_vector((DATA_WIDTH-1) downto 0);
        we            : in std_logic := '1';
        q             : out std_logic_vector((DATA_WIDTH -1) downto 0)
    );
end entity;

architecture rtl of single_port_ram is
    -- Build a 2-D array type for the RAM
    subtype word_t is std_logic_vector((DATA_WIDTH-1) downto 0);
```

```
    type memory_t is array(2 ** ADDR_WIDTH - 1 downto 0) of word_t;
    -- Declare the RAM signal.
    signal ram : memory_t;
    -- Register to hold the address
    signal addr_reg : natural range 0 to 2 ** ADDR_WIDTH - 1;

begin
    process(clk)
    begin
    if(rising_edge(clk)) then
        if(we = '1') then
            ram(addr) <= data;
        end if;
        -- Register the address for reading
        addr_reg <= addr;
    end if;
    end process;

    q <= ram(addr_reg);
end rtl;
```

下面利用 DES 数据加密算法中的 S 盒设计，给出通过 MegaWizard Plug-In Manager 进行单口 RAM 设计的过程。

通过菜单 Tools→IPCatalog，在 IPCatalog 栏中输入 RAM，然后会列出相关的 IP 核，在这里选择 RAM:1-PORT，并双击进入如图 5-6 所示的界面。设计语言选择 VHDL，输入输出文件名，然后单击 OK 按钮，依次进入图 5-9 到图 5-11 所示的参数设置界面。

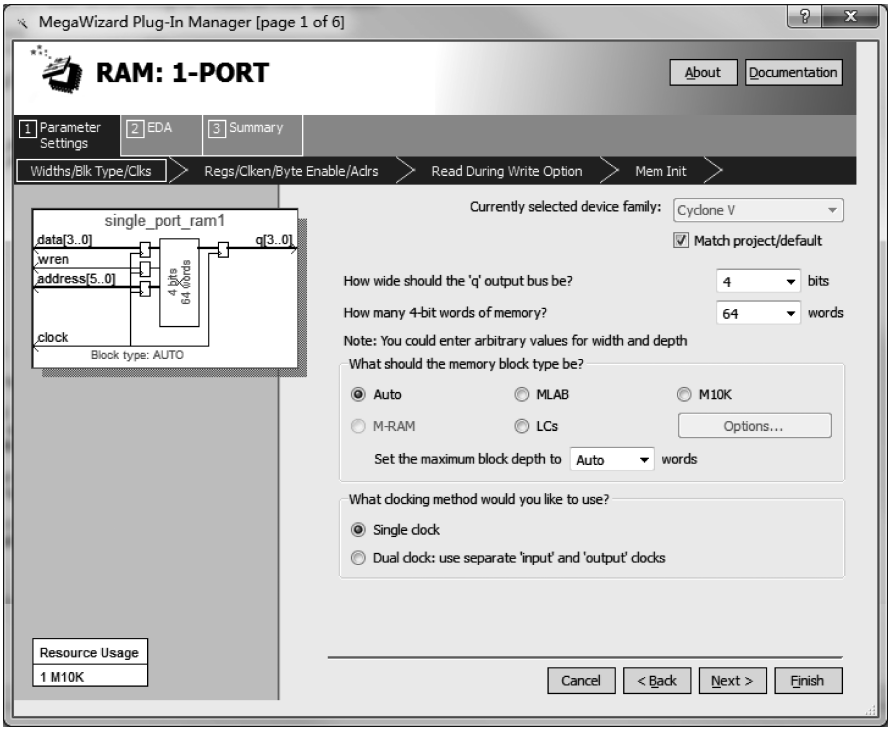

图 5-9　单端口 RAM 模块的参数设置(1)

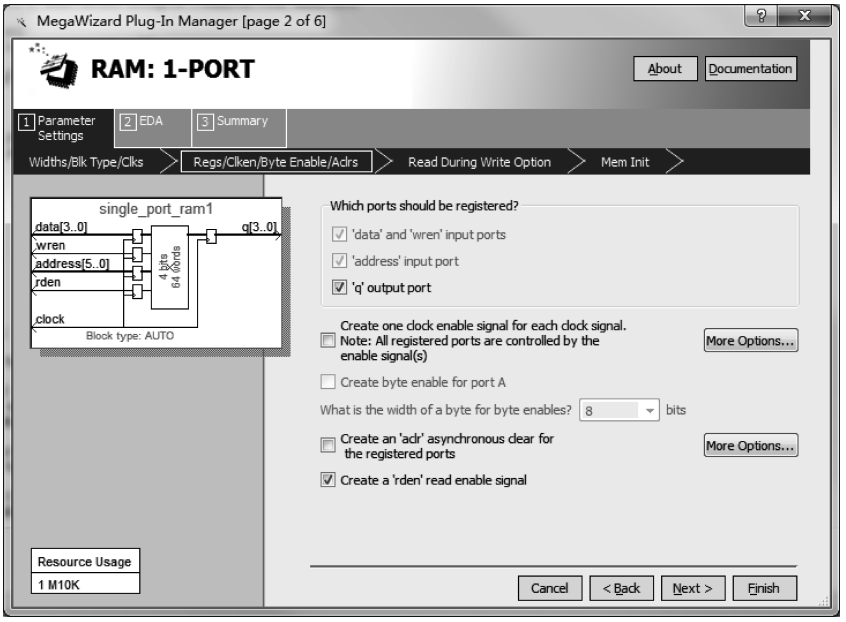

图 5-10　单端口 RAM 模块的参数设置(2)

因为 DES 算法的 S 盒是 6 进 4 出即包含 64 个 4bit 数据,因此,在图 5-9 页面中设置存储容量 64words、数据宽度 4bit,输入输出使用相同时钟；图 5-10 页面中设置字节使能、寄存器存储、独立读使能等相关属性；在图 5-11 中可以指定 RAM 的初始内容,这里使用的是内存初始化文件 sbox.mif,该初始化文件的生成过程参照图 5-12 和图 5-13 所示。

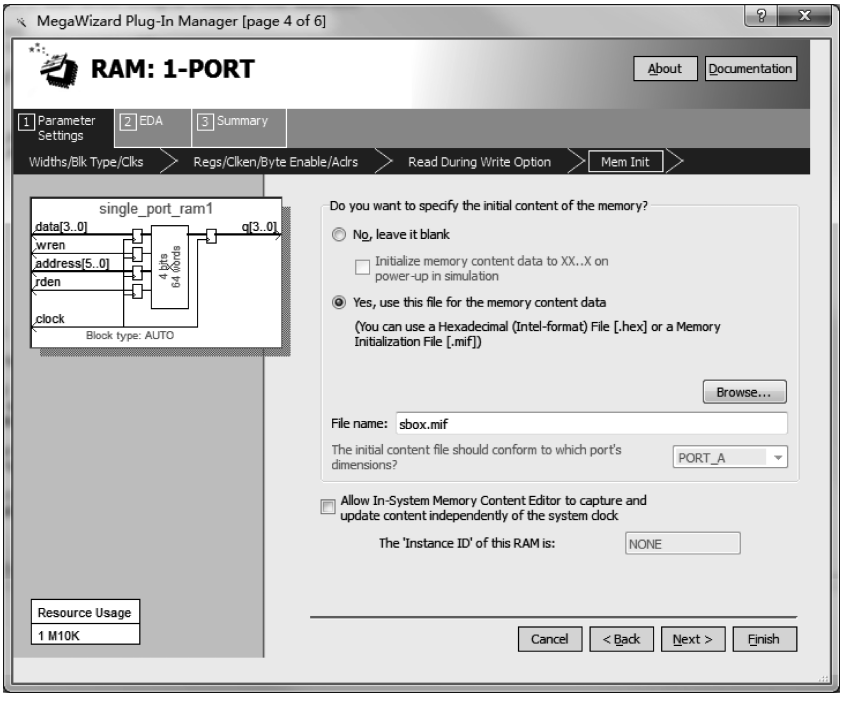

图 5-11　单端口 RAM 模块的参数设置(3)

首先选择文件菜单下的新建文件，选择 Memory Initialization File，在如图 5-12 所示弹出对话框中设置存储字数和字大小，单击 OK 按钮后，可以生成如图 5-13 所示 mif 文件编辑界面，将 DES 的 S 盒数据输入，然后保存即可。

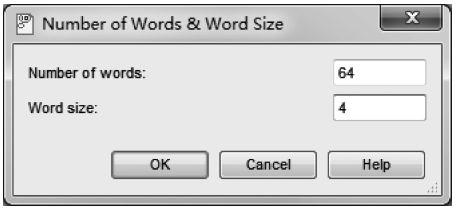

图 5-12　内存初始化文件容量设置

Addr	+0	+1	+2	+3	+4	+5	+6	+7	ASCII
0	15	1	8	14	6	11	3	4	.....
8	9	7	2	13	12	0	5	10	.....
16	3	13	4	7	15	2	8	14	.....
24	12	0	1	10	6	9	11	5	.....
32	0	14	7	11	10	4	13	1	.....
40	5	8	12	6	9	3	2	15	.....
48	13	8	10	1	3	15	4	2	.....
56	11	6	7	12	0	5	14	9	.....

图 5-13　内存初始化文件内容编辑

下面是定制的 RAM 模块实现代码，可以看到，RamTest_SBox 是通过例化 Altera 内部模块 altsyncram，然后进行端口配置实现的。

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

LIBRARY altera_mf;
USE altera_mf.all;

ENTITY RamTest_SBox IS
    PORT
    (
        address       : IN STD_LOGIC_VECTOR (5 DOWNTO 0);
        clock         : IN STD_LOGIC := '1';
        data          : IN STD_LOGIC_VECTOR (3 DOWNTO 0);
        rden          : IN STD_LOGIC := '1';
        wren          : IN STD_LOGIC ;
        q             : OUT STD_LOGIC_VECTOR (3 DOWNTO 0)
    );
END RamTest_SBox;


ARCHITECTURE SYN OF ramtest_sbox IS

    SIGNAL sub_wire0: STD_LOGIC_VECTOR (3 DOWNTO 0);
    COMPONENT altsyncram
    GENERIC (
        clock_enable_input_a           : STRING;
        clock_enable_output_a          : STRING;
        init_file                      : STRING;
        intended_device_family         : STRING;
        lpm_hint                       : STRING;
        lpm_type                       : STRING;
```

```
        numwords_a                      : NATURAL;
        operation_mode                  : STRING;
        outdata_aclr_a                  : STRING;
        outdata_reg_a                   : STRING;
        power_up_uninitialized          : STRING;
        read_during_write_mode_port_a   : STRING;
        widthad_a                       : NATURAL;
        width_a                         : NATURAL;
        width_byteena_a                 : NATURAL
    );
    PORT (
            address_a: IN STD_LOGIC_VECTOR (5 DOWNTO 0);
            clock0: IN STD_LOGIC ;
            data_a: IN STD_LOGIC_VECTOR (3 DOWNTO 0);
            wren_a: IN STD_LOGIC ;
            q_a: OUT STD_LOGIC_VECTOR (3 DOWNTO 0);
            rden_a: IN STD_LOGIC
    );
    END COMPONENT;

BEGIN
    q  <=  sub_wire0(3 DOWNTO 0);

    altsyncram_component : altsyncram
    GENERIC MAP (
        clock_enable_input_a => "BYPASS",
        clock_enable_output_a => "BYPASS",
        init_file => "sbox.mif",
        intended_device_family => "Cyclone IV GX",
        lpm_hint => "ENABLE_RUNTIME_MOD=NO",
        lpm_type => "altsyncram",
        numwords_a => 64,
        operation_mode => "SINGLE_PORT",
        outdata_aclr_a => "NONE",
        outdata_reg_a => "CLOCK0",
        power_up_uninitialized => "FALSE",
        read_during_write_mode_port_a => "NEW_DATA_NO_NBE_READ",
        widthad_a => 6,
        width_a => 4,
        width_byteena_a => 1
    )
    PORT MAP (
        address_a => address,
        clock0 => clock,
        data_a => data,
        wren_a => wren,
        rden_a => rden,
        q_a => sub_wire0
    );

END SYN;
```

对生成程序进行综合，然后编写测试用例进行仿真，例 5.3 给出的测试用例向测试对象提供时钟激励，读信号 rden 始终有效，并且在该时钟上升沿顺序给出地址数据，从而完成 S 盒内容读取的仿真。

【例 5.3】 基于存储器 IP 的 DES 算法 S 盒实现的仿真测试用例。

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;

ENTITY RamTest_SBox_vhd_tst IS
END RamTest_SBox_vhd_tst;
ARCHITECTURE RamTest_SBox_arch OF RamTest_SBox_vhd_tst IS
SIGNAL address : STD_LOGIC_VECTOR(5 DOWNTO 0) := "000000";
SIGNAL clock : STD_LOGIC := '0';
SIGNAL data : STD_LOGIC_VECTOR(3 DOWNTO 0);
SIGNAL q : STD_LOGIC_VECTOR(3 DOWNTO 0);
SIGNAL rden : STD_LOGIC := '1';          -- 读信号始终有效
SIGNAL wren : STD_LOGIC := '0';          -- 不允许写
COMPONENT RamTest_SBox
    PORT (
    address : IN STD_LOGIC_VECTOR(5 DOWNTO 0);
    clock : IN STD_LOGIC;
    data : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
    q : OUT STD_LOGIC_VECTOR(3 DOWNTO 0);
    rden : IN STD_LOGIC;
    wren : IN STD_LOGIC
    );
END COMPONENT;
BEGIN
    i1 : RamTest_SBox
    PORT MAP (
    address => address,
    clock => clock,
    data => data,
    q => q,
    rden => rden,
    wren => wren
    );
always : PROCESS (clock)                 -- 时钟下降沿给出地址激励信号
BEGIN
    if falling_edge(clock) then
     if (address = "11111") then address <= "00000";
        else address <= address + "00001";
        end if;
    end if;
END PROCESS always;

CLOCK_Pro : PROCESS                      -- 提供时钟激励信号，周期 20ns
BEGIN
    wait for 10 ns; clock <= not clock;
END PROCESS CLOCK_Pro;
END RamTest_SBox_arch;
```

基于上述测试用例，通过 ModelSim 进行仿真，得到图 5-14 所示仿真波形，如图中所示，地址输入在时钟下降沿发生变化，读信号始终有效，在时钟上升沿数据正常输出。图中 data 始终显示红色，是因为测试用例中没有给出该信号的具体值。该信号是向 RAM 写入的数据，由于写信号 wren 始终为 0，因此没有真正写入 RAM 中。

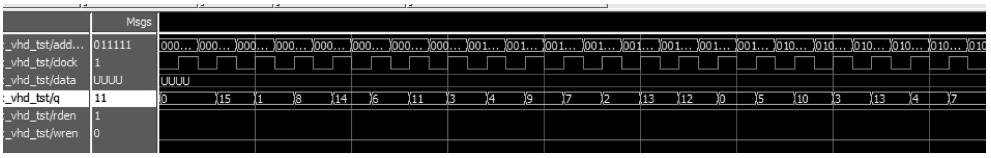

图 5-14　基于 RAM 存储器 IP 的 DES 算法 S 盒的仿真波形

密码算法的 S 盒变换通常是固定数据的，读者可以将上述设计修改为 ROM 实现，也可以通过 wren 和 data 端口对 S 盒内容进行更改，从而可以用在不同的算法中。

5.4　锁相环 IP 核的 VHDL 设计应用

Intel 在很多型号的 FPGA 芯片中都提供有专用锁相环电路，用来实现设计所需多种时钟频率。通过 Quartus Ⅱ或者 Quartus Prime 软件的参数化模块库中的 PLL 模块可以很好地利用 FPGA 芯片中的锁相环资源。下面利用 ALTPLL 模块的参数配置和实例化对锁相环电路 IP 核的 VHDL 设计进行简单介绍。

在 IPCatalog 栏中输入 PLL 或者 ALTPLL，然后在 Library 中单击 ALTPLL，在弹出的如图 5-15 所示的 Save IP Variation 界面，选择 VHDL 作为创建的设计文件语言，将输出文件命名为 mypll。单击 OK 按钮后进入图 5-16 所示的对话框，在这里对输入时钟 inclk0 的频率和 PLL 的工作模式进行设置，假设输入频率为 100MHz，工作模式采用 normal 模式。输入频率用于输出频率设置的参考，不与实际工作频率相关。Altera 器件共有四种工作模式：normal 模式，PLL 的输入引脚与 I/O 单元的输入时钟寄存器相关联；zero delay buffer 模式，PLL 的输入引脚和 PLL 的输出引脚的延时相关联，通过 PLL 的调整，到达两者“零”延时；External feedback 模式，PLL 的输入引脚和 PLL 的反馈引脚延时相关联；no compensation 模式，不对 PLL 的输入引脚进行延时补偿。

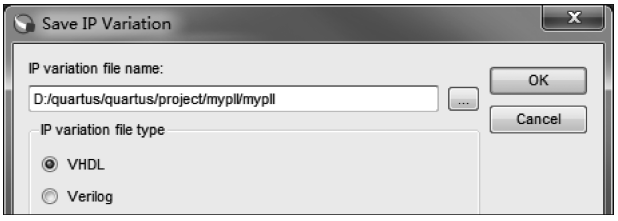

图 5-15　创建新的参数化模块——锁相环 PLL

参数化模块 ALTPLL 可以设置 9 个输出时钟，这里仅使用两个输出时钟：c0 和 c1，分别设置为 300MHz 和 75MHz，如图 5-17 和图 5-18 所示。这里的时钟输出频率都是以设定乘因子和除因子的方式给出，也可以直接输入预期时钟频率(Requested Setting)。

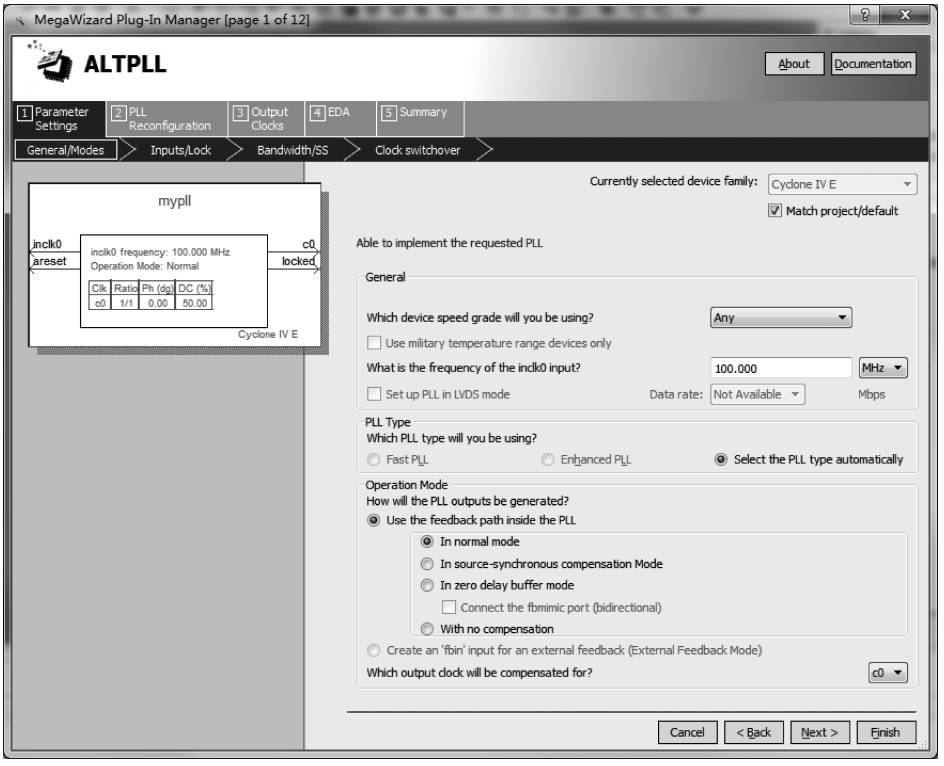

图 5-16　参数化模块 ALTPLL 的参数设置

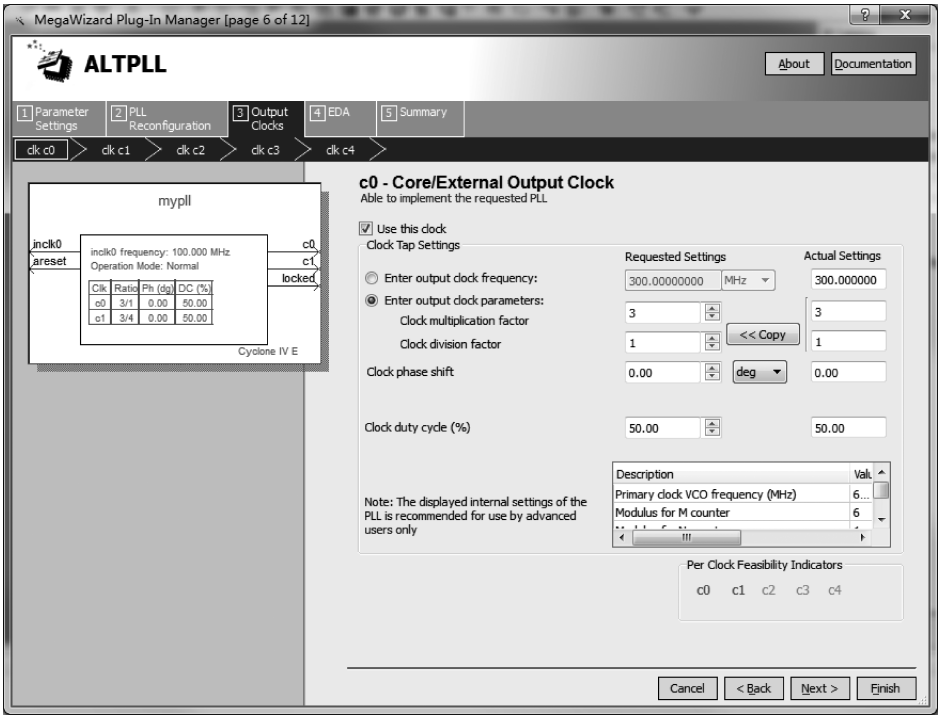

图 5-17　参数化模块 ALTPLL 的参数设置-c0

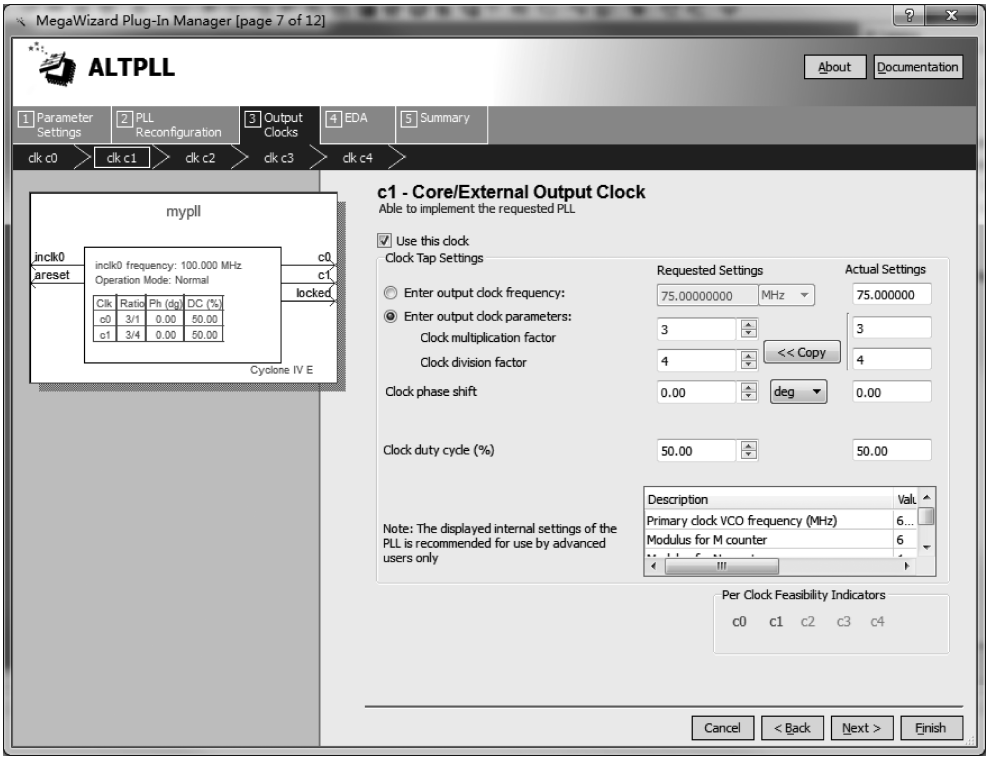

图 5-18　参数化模块 ALTPLL 的参数设置-c1

时钟模块的其他设置均采用默认设置。通过给定输入时钟频率进行仿真，可以得到如图 5-19 所示的仿真图。

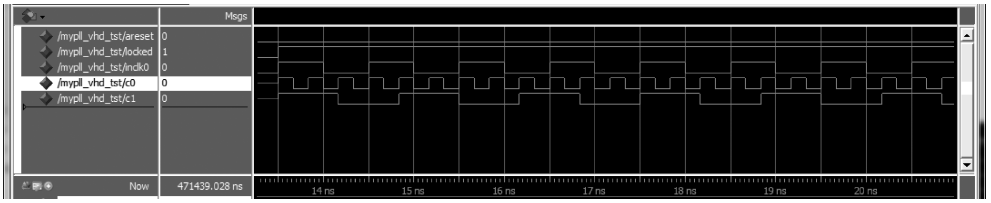

图 5-19　锁相环 PLL 的仿真结果

分析图 5-19 所示仿真波形，其中 inclk0 是输入时钟信号，时钟周期为 10000ps，时钟频率为 100MHz；c0 和 c1 是输出信号，三个时钟信号都是占空比 1∶1 的时钟信号。如图所示，inclk0 经过 1 个时钟周期后，c0 恰好经过了 3 个时钟周期，即 c0 的频率是 inclk0 的 3 倍，即 300MHz。再分析 c1 和 c0 的周期特性，可以发现，c1 的频率是 c0 的 1/4，即 75MHz。所以，通过仿真波形可知，仿真结果与 ALTPLL 的设置一致，PLL 设计正确。

5.5　运算电路 IP 核的 VHDL 设计应用

Quartus Prime 软件的参数化模块库对运算单元的 IP 模块有很好的支持，常用的数学运算都可以在这里完成。下面利用 LPM_ADD_SUB 模块设计一个简单的 8 位加法器，对

运算电路 IP 核的 VHDL 设计进行简单介绍。

在 IP Catalog 栏中输入 LPM_ADD_SUB，在 Library 中选择 LPM_ADD_SUB 并双击，弹出 Save IP Variation 界面，如图 5-20 所示，选择 VHDL 作为创建的设计文件语言，将输出文件命名为 adder。单击 OK 按钮后进入如图 5-21 所示的对话框，指定输入数据的位宽为 8bit，只选择加法功能；单击 Next 按钮，进入图 5-22 所示对话框确认输入数据的类型，按默认值设置两个操作数均为可变无符号数。再次单击 Next 按钮，在图 5-23 所示对话框，指定加入进位输入端和进位输出端。然后，在图 5-24 页面下，设置流水线设计参数，其他内容可按默认设置使用。

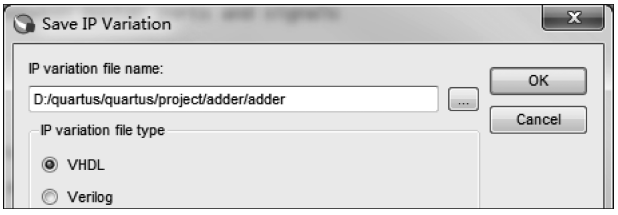

图 5-20　创建新的参数化模块——运算电路 LPM_ADD_SUB

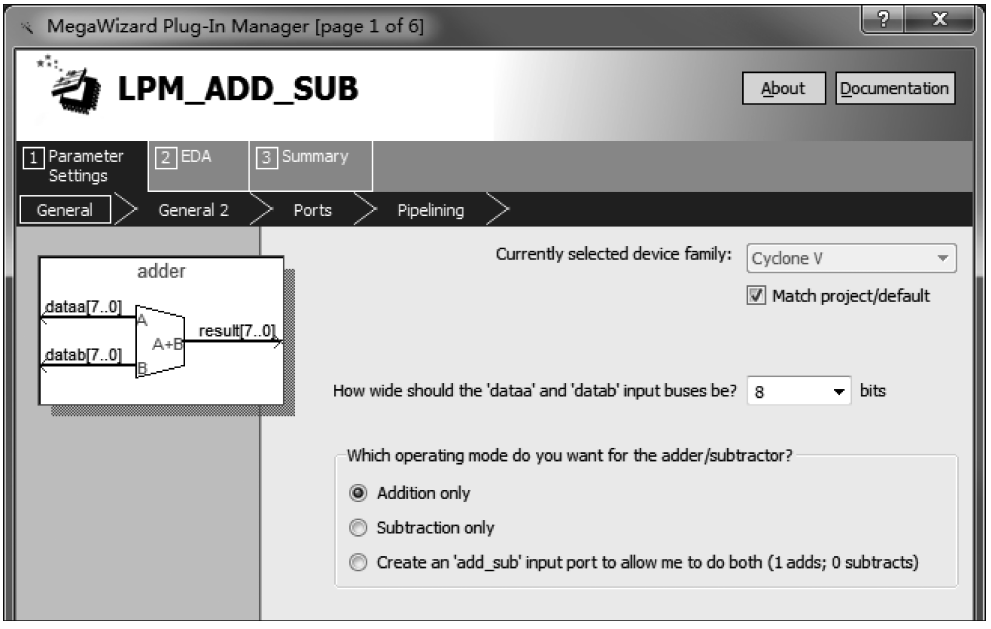

图 5-21　参数化模块 LPM_ADD_SUB 的参数设置(1)

通过向导定制完加法器后，可以创建相应的 VHDL 文件，然后编写如例 5.4 所示的测试用例，利用 ModelSim 进行仿真，可以得到如图 5-25 所示的仿真结果。

【例 5.4】 加法器测试用例。

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY adder_vhd_tst IS
```

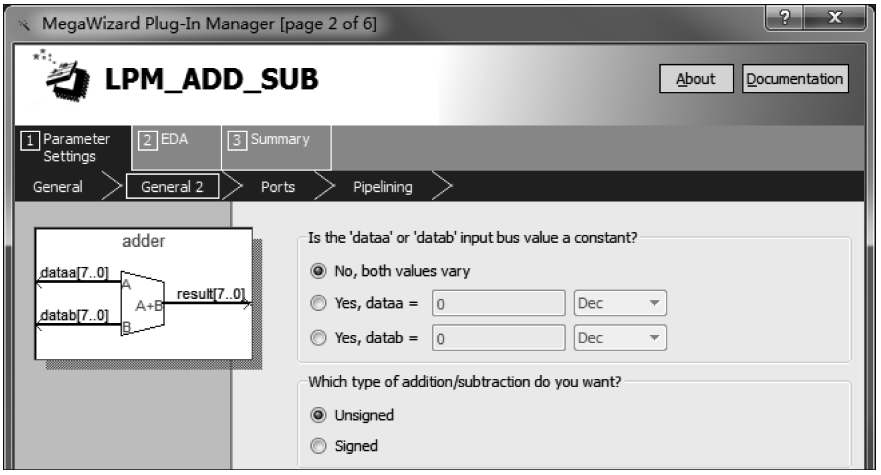

图 5-22　参数化模块 LPM_ADD_SUB 的参数设置(1)

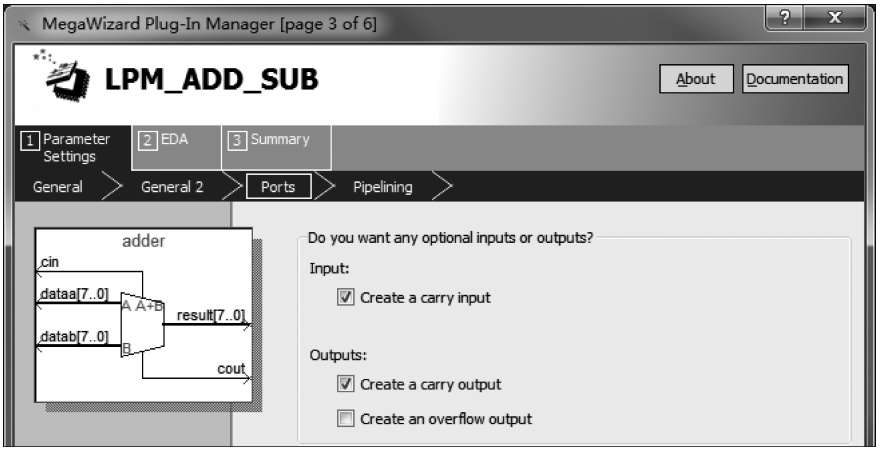

图 5-23　参数化模块 LPM_ADD_SUB 的参数设置(2)

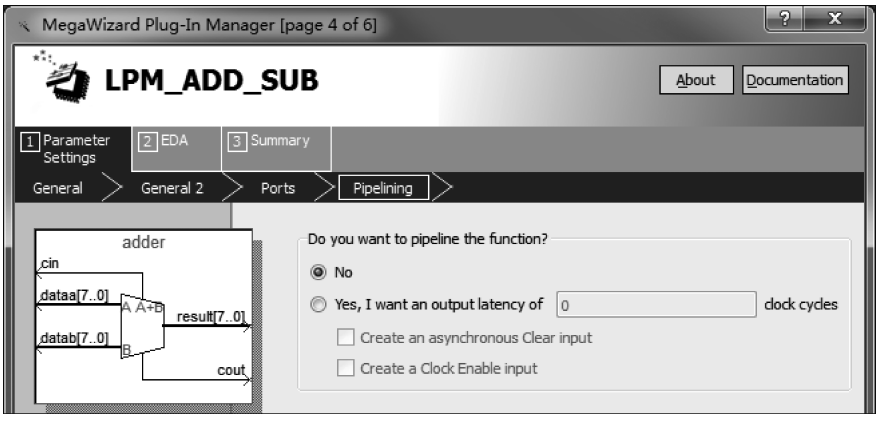

图 5-24　参数化模块 LPM_ADD_SUB 的参数设置(3)

```
END adder_vhd_tst;
ARCHITECTURE adder_arch OF adder_vhd_tst IS

SIGNAL cin : STD_LOGIC;
SIGNAL cout : STD_LOGIC;
SIGNAL dataa : STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL datab : STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL result : STD_LOGIC_VECTOR(7 DOWNTO 0);
COMPONENT adder
    PORT (
    cin : IN STD_LOGIC;
    cout : OUT STD_LOGIC;
    dataa : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
    datab : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
    result : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
    );
END COMPONENT;
BEGIN
    i1 : adder
    PORT MAP (
-- list connections between master ports and signals
    cin => cin,
    cout => cout,
    dataa => dataa,
    datab => datab,
    result => result
    );
init : PROCESS
-- variable declarations
BEGIN
        -- code that executes only once
    wait for 1 ns; cin <= '0'; dataa <= x"3f"; datab <= x"57"; -- 等待 1ns 后数据有效
    wait for 1 ns; cin <= '1'; dataa <= x"7f"; datab <= x"57";
    wait for 1 ns; cin <= '0'; dataa <= x"9a"; datab <= x"85";
    wait for 1 ns; cin <= '1'; dataa <= x"97"; datab <= x"68";
    WAIT;
END PROCESS init;
END adder_arch;
```

图 5-25 给出了上述测试用例仿真后波形，图中数据开始的红色部分是由于测试用例中首先等待了 1ns 后才给出有效数据，这一段时间输入没有初始值，所以数据均为 X。这里给出的运算电路是一个带进位的 8 位全加器，数据显示均为十六进制。从图中可以看出，在 1000ps 处，输入数据为：进位位 0，加数 dataa 和 datab 分别是 0x3F 和 0x57，结果 result 为 0x86，进位输出 0，之后在 2000ps、3000ps、4000ps 处，随着输入数据的变化，输出结果做出相应变化，分析可知，输出符合设计要求，设计结果正确。

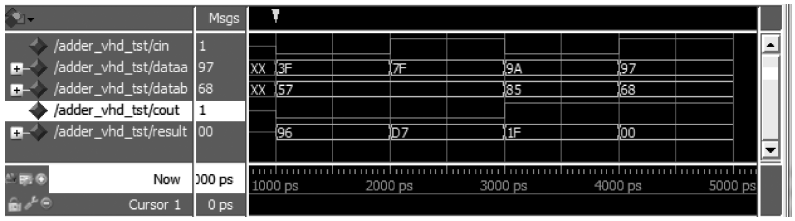

图 5-25　加法器仿真结果