CHAPTER 11

XILINX SPARTAN-3 SPECIFIC MEMORY

11.1 INTRODUCTION

A digital system frequently requires memory for storage. To facilitate this need, most FPGA devices contain dedicated embedded memory modules. While these modules cannot replace the massive external memory devices, they are useful for applications that require small or intermediate-sized memory.

Although the basic internal structure of memory modules is similar, there are many subtle differences in their I/O interfaces. It is usually difficult for synthesis software to extract the desired features from the code and to infer a matching memory module from the underlying device library. In Xilinx ISE, we can use *HDL instantiation*, the Core Generator program, or the *behavioral HDL inference template* to incorporate an embedded memory module into a design. The third one is semi-device independent and we use this method in this book. In this chapter, we briefly examine Spartan-3 memory modules and the first two methods and provide detailed descriptions of several key behavioral HDL templates.

11.2 EMBEDDED MEMORY OF SPARTAN-3 DEVICE

11.2.1 Overview

There are two types of embedded memory in a Spartan-3 device: distributed RAM and block RAM. A *distributed RAM* is constructed from the logic cell's look-up table (LUT). The LUT can be configured as a 16-by-1 synchronous RAM, and multiple LUTs can be cascaded to form a wider and deeper memory module. The Spartan-3 XC3S200 device of the S3 board can provide up to 30K bits of distributed memory, which is small compared to a block RAM or external memory. Furthermore, since the distributed RAM uses the logic cells, it competes for resources with the normal logic. Thus, it is feasible only for applications that require relatively small storage.

A *block RAM* is a special memory module embedded in an FPGA device and is separated from the regular logic cells. It can be thought of as a fast SRAM wrapped by a synchronous, configurable interface. Each block RAM consists of 16K (2^{14}) data bits plus optional 2K parity bits. It can be organized in different widths, from 16K by 1 (i.e., 2^{14} by 2^{0}) to 512 by 32 (i.e., 2^{9} by 2^{5}). The Spartan-3 XC3S200 device has 12 block RAMs, totaling 172K data bits. These block RAMs can be used for intermediate-sized applications, such as a FIFO, a large look-up table, or an intermediate-sized local memory. In comparison, the external SRAM chips of the S3 board have a capacity of 8M bits.

Both the distributed RAM and block RAM are already "wrapped" with a synchronous interface, and thus no additional memory controller circuit is needed. They are very flexible and can be configured to perform single- and dual-port access and to support various types of buffering and clocking schemes. Detailed discussion is beyond the scope of this book. We only examine several commonly used configurations, including a synchronous single-port RAM, a synchronous dual-port RAM, and a ROM in Section 11.4.

11.2.2 Comparison

The Spartan-3 device and the S3 board provide several options for storage elements. It is a good idea to keep in mind the relative capacities of these options:

- *XC3S200's FFs* (for registers): about 4.5K bits, embedded in logic cells and I/O buffers
- *XC3S200's distributed RAM*: 30K bits, constructed from the logic cells
- *XC3S200's block RAM*: 172K bits, configured as twelve 16K-bit modules
- *External SRAM*: 8M bits, configured as two 256K-by-16 SRAM chips

This helps us to decide which option is most suitable for an application at hand.

11.3 METHOD TO INCORPORATE MEMORY MODULES

Although memory modules have similar internal structure, there are many subtle differences in their interfaces, such as the numbers of read and write ports, clocking scheme, data and address buffering, enable and reset signals, and initial values. Although it is possible to describe the desired module behaviors in HDL code, the synthesis software may or may not recognize the designer's intention. Therefore, the HDL code cannot always infer the proper memory module and is normally not portable. In Xilinx ISE, there are three methods to incorporate an embedded memory module into a design:

- HDL instantiation
- The Core Generator program
- The behavioral HDL inference template

The first two are specific for Xilinx devices and the third is a semi-device-independent behavioral description. Because of the clarity of the behavioral description, we use the third method in this book. We provide a brief overview of the three methods in this section.

11.3.1 Memory module via HDL component instantiation

We have used HDL component instantiation in many earlier design examples to include predesigned modules or to create a hierarchy. Instantiating a Xilinx memory module is similar except that there is no HDL description for the architecture body. We must check the manual to find the exact entity name and the associated generics and I/O port definitions. This is a tedious process and is particularly error-prone for memory modules because of the large number of configurations and options.

The instantiation code for many Xilinx components can be obtained directly from ISE by selecting Edit ≻ Language Templates. The following are segments of a 16K-by-1 dual-port RAM:

```
-- RAMB16_S1_S1 : Virtex-II/II-Pro,
-- Spartan-3/3E 16k x 1 Dual-Port RAM
-- Xilinx HDL Language Template version 8.1i
RAMB16_S1_S1_inst : RAMB16_S1_S1
generic map(
   init_a => "0",
   init_b => "0",
   srval_a => "0",
   srval_b => "0",
   write_mode_a => "WRITE_FIRST",
   write_mode_b => "WRITE_FIRST",
   sim_collision_check => "ALL",
   init_00 => x"0 ... 0",
   ...
   init_3f => x"0 ... 0"
),
port map(
   doa => doa,         -- port a 1-bit data output
   dob => dob,         -- port b 1-bit data output
   addra => addra,     -- port a 14-bit address input
   addrb => addrb,     -- port b 14-bit address input
   clka => clka,       -- port a clock
   clkb => clkb,       -- port b clock
   dia => dia,         -- port a 1-bit data input
   dib => dib,         -- port b 1-bit data input
   ena => ena,         -- port a ram enable input
   enb => enb,         -- port b ram enable input
   ssra => ssra,       -- port a synchronous set/reset input
   ssrb => ssrb,       -- port b synchronous set/reset input
   wea => wea,         -- port a write enable input
   web => web          -- port b write enable input
);
```

Although the code is readily available, we must study the manual carefully to find the right component and proper configuration parameters.

11.3.2 Memory module via Core Generator

To simplify the instantiation process, Xilinx provides a utility program, known as Core Generator (Coregen), to generate Xilinx-specific components. This utility can be invoked from the ISE environment by selecting Project ≻ New Source. After the New Source

Wizard dialog appears, we select IP (Coregen & Architecture Wizard) to invoke the Coregen program. The program guides the users through a series of questions and then generates several files. The file with the .xco extension is a text file that contains the information necessary to construct the desired memory component. The file with the .vhd extension contains the "wrapper" code for simulation purpose. This file cannot be used to instantiate the desired component and is ignored during the synthesis process.

Although using the Coregen program is more convenient than direct HDL instantiation, it is not within the HDL framework and can lead to a compatibility problem when a design is not done in the Xilinx ISE environment.

11.3.3 Memory module via HDL inference

Although it is not possible to develop a device-independent HDL description, the synthesis program of ISE, known as XST, provides a collection of behavioral HDL templates to infer memory modules from Xilinx FPGA devices. These templates are done by behavioral descriptions and contain no device-specific component instantiation. They are easy to understand and can be simulated without an additional HDL library. However, while the description does not explicitly refer to any Xilinx component, the code may not be recognized by other third-party synthesis software, and the desired memory module cannot always be inferred. Thus, these templates can best be described as "semi-portable" and "semi-device-independent" behavioral descriptions. Templates for commonly used memory modules are discussed in Section 11.4.

On the downside, the template approach is based on the ability of the XST software to recognize the template and infer the proper memory module accordingly. The software may change during upgrade or misinterpret some code. It is a good idea to check the XST synthesis report to ensure that the desired memory module is inferred correctly.

11.4 HDL TEMPLATES FOR MEMORY INFERENCE

To use behavioral HDL description to infer the Xilinx memory module, the XST's templates should be followed closely. To avoid misinterpretation, we should refrain from creating our own "innovative" code. The codes in the following subsections are all based on templates of the *XST v8.1i Manual*. They are the same as the original templates except that generics are used for the width of address bits and the width of data bits, and the `numeric_std` package is used to replace the proprietary `std_logic_unsigned` package. It is a good practice to confine the memory description in a separate HDL module so that the module can easily be identified and replaced when needed. In this section, we discuss the behavioral HDL templates for six configurations, including two for single-port RAMs, two for dual-port RAMs, and two for ROMs.

11.4.1 Single-port RAM

The embedded memory of a Spartan-3 device is already wrapped with a synchronous interface similar to that in Section 10.3. Its write operation is always synchronous. At the rising edge of the clock, the address, input data, and relevant control signals, such as `we` (i.e., write enable), are sampled. If `we` is asserted, a write operation is performed (i.e., the input data is stored into the memory location designated by the address signal).

The read operation can be asynchronous or synchronous. For *asynchronous read*, the address signal is used directly to access the RAM array. After the address signal changes, the data becomes available after a short delay. For *synchronous read*, the address signal is sampled at the rising edge of the clock and stored in a register. The registered address is then used to access the RAM array. Because of the register, the availability of data is delayed and is synchronized by the clock signal. Due to the internal structure, asynchronous read operation can only be realized by the distributed RAM.

Single-port RAM with asynchronous read The template for the single-port RAM with asynchronous read is shown in Listing 11.1. It is modified after the `rams_04` entity of the *XST Manual.*

Listing 11.1 Template for a single-port RAM with asynchronous read

```
-- single-port RAM with asynchronous read
-- modified from XST 8.1i rams_04
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity xilinx_one_port_ram_async is
   generic(
      ADDR_WIDTH: integer:=8;
      DATA_WIDTH: integer:=1
   );
   port(
      clk: in std_logic;
      we: in std_logic;
      addr: in std_logic_vector(ADDR_WIDTH-1 downto 0);
      din: in std_logic_vector(DATA_WIDTH-1 downto 0);
      dout: out std_logic_vector(DATA_WIDTH-1 downto 0)
   );
end xilinx_one_port_ram_async;

architecture beh_arch of xilinx_one_port_ram_async is
   type ram_type is array (2**ADDR_WIDTH-1 downto 0)
        of std_logic_vector (DATA_WIDTH-1 downto 0);
   signal ram: ram_type;
begin
   process (clk)
   begin
      if (clk'event and clk = '1') then
         if (we='1') then
            ram(to_integer(unsigned(addr))) <= din;
         end if;
      end if;
   end process;
   dout <= ram(to_integer(unsigned(addr)));
end beh_arch;
```

The code is very similar to the register file discussed in Section 4.2.3 except that the read and write operations use the same address. It contains a user-defined two-dimensional array data type for storage and uses dynamic indexing to access the element in the array. The code shows that the write operation is controlled by the clock signal and the read operation depends only on the address. Since asynchronous read can be realized only by the distributed RAM, this configuration is only recommended for applications that require small storage.

Single-port RAM with synchronous read The template for the single-port RAM with synchronous read is shown in Listing 11.2. It is modified after the rams_07 entity of the *XST Manual.*

Listing 11.2 Template for a single-port RAM with synchronous read

```
-- single-port RAM with synchronous read
-- modified from XST 8.1i rams_07
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity xilinx_one_port_ram_sync is
   generic(
      ADDR_WIDTH: integer:=12;
      DATA_WIDTH: integer:=8
   );
   port(
      clk: in std_logic;
      we: in std_logic;
      addr: in std_logic_vector(ADDR_WIDTH-1 downto 0);
      din: in std_logic_vector(DATA_WIDTH-1 downto 0);
      dout: out std_logic_vector(DATA_WIDTH-1 downto 0)
    );
end xilinx_one_port_ram_sync;

architecture beh_arch of xilinx_one_port_ram_sync is
   type ram_type is array (2**ADDR_WIDTH-1 downto 0)
        of std_logic_vector (DATA_WIDTH-1 downto 0);
   signal ram: ram_type;
   signal addr_reg: std_logic_vector(ADDR_WIDTH-1 downto 0);
begin
   process (clk)
   begin
      if (clk'event and clk = '1') then
         if (we='1') then
            ram(to_integer(unsigned(addr))) <= din;
         end if;
         addr_reg <= addr;
      end if;
   end process;
   dout <= ram(to_integer(unsigned(addr_reg)));
end beh_arch;
```

Note that the addr signal is now sampled and stored to the addr_reg register at the rising edge of the clock, and the memory array (the ram signal) is accessed via the addr_reg signal. The data is available only after the addr_reg is updated and thus implicitly synchronized to the clk signal.

Synthesis report During synthesis, a proper RAM module should be inferred from the code template. We can check the synthesis report to confirm the inference of the RAM module. For example, consider the instantiation of a 4K-by-8 RAM (2^{12}-by-2^3) with synchronous read:

```
unit_4K_by_8: entity work.xilinx_one_port_sram_sync
   generic map(ADDR_WIDTH=>12, DATA_WIDTH=>8)
   port map(clk=>clk, we=>we, addr=>addr,
            din=>din,dout=>dout);
```

The inference of RAM should be indicated in the HDL Synthesis section of the synthesis report:

```
==========================================================
*                      HDL Synthesis                     *
==========================================================
...
Found 4096x8-bit single-port block RAM for signal <ram>.
---------------------------------------------------------
| mode          | write-first                  |      |
| aspect ratio  | 4096-word x 8-bit            |      |
| clock         | connected to signal <clk>    | rise |
| write enable  | connected to signal <we>     | high |
| address       | connected to signal <addr>   |      |
| data in       | connected to signal <din>    |      |
| data out      | connected to signal <dout>   |      |
| ram_style     | Auto                         |      |
---------------------------------------------------------
Summary:
inferred   1 RAM(s).
```

The number of block RAMs used should be reported in the Final Report section of the synthesis report:

```
Device utilization summary:
Selected Device : 3s200ft256-5
 ...
 Number of BRAMs:    2  out of     12    16%
 ...
```

As we expected, a 4K-by-8 single-port block RAM is inferred and two block RAMs are used to realize the circuit.

11.4.2 Dual-port RAM

A dual-port RAM includes a second port for memory access. Ideally, the second port should be able to conduct read or write operation independently and have its own set of address, data input and output, and control signals. To be compatible with older versions of XST, we consider a configuration with the second port that can conduct a read operation only. In this book, the main application of the dual-port configuration is for video memory, which requires one write port and one read port. Thus, this configuration does not impose a serious limitation for our purposes. As in a single-port RAM, the read operation of a dual-port RAM can be asynchronous or synchronous.

Dual-port RAM with asynchronous read The template for the dual-port RAM with asynchronous read is shown in Listing 11.3. It is modified after the rams_09 entity of the *XST Manual.*

Listing 11.3 Template for a dual-port RAM with asynchronous read

```
-- dual-port RAM with asynchronous read
-- modified from XST 8.1i rams_09
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity xilinx_dual_port_ram_async is
   generic(
      ADDR_WIDTH: integer:=6;
      DATA_WIDTH:integer:=8
   );
   port(
      clk: in std_logic;
      we: in std_logic;
      addr_a: in std_logic_vector(ADDR_WIDTH-1 downto 0);
      addr_b: in std_logic_vector(ADDR_WIDTH-1 downto 0);
      din_a: in std_logic_vector(DATA_WIDTH-1 downto 0);
      dout_a: out std_logic_vector(DATA_WIDTH-1 downto 0);
      dout_b: out std_logic_vector(DATA_WIDTH-1 downto 0)
   );
end xilinx_dual_port_ram_async;

architecture beh_arch of xilinx_dual_port_ram_async is
   type ram_type is array (0 to 2**ADDR_WIDTH-1)
        of std_logic_vector (DATA_WIDTH-1 downto 0);
   signal ram: ram_type;
begin
   process(clk)
   begin
     if (clk'event and clk = '1') then
        if (we = '1') then
           ram(to_integer(unsigned(addr_a))) <= din_a;
        end if;
     end if;
   end process;
   dout_a <= ram(to_integer(unsigned(addr_a)));
   dout_b <= ram(to_integer(unsigned(addr_b)));
end beh_arch;
```

The write operation is similar to that of the single-port RAM, but the code includes a second output port, dout_b, which retrieves data from the second address, addr_b. As in a single-port RAM with asynchronous read, the dual-port version can be realized only by distributed RAM, and thus its size is limited. Note that if we ignore the dout_a port, it is the same as the single-read-port register file of Listing 4.6.

Dual-port RAM with synchronous read The template for the dual-port RAM with synchronous read is shown in Listing 11.4. It is modified after the rams_11 entity of the *XST Manual.*

Listing 11.4 Template for a dual-port RAM with synchronous read

```
-- dual-port RAM with synchronous read
-- modified from XST 8.1i rams_11
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity xilinx_dual_port_ram_sync is
   generic(
      ADDR_WIDTH: integer:=6;
      DATA_WIDTH:integer:=8
   );
   port(
      clk: in std_logic;
      we: in std_logic;
      addr_a: in std_logic_vector(ADDR_WIDTH-1 downto 0);
      addr_b: in std_logic_vector(ADDR_WIDTH-1 downto 0);
      din_a: in std_logic_vector(DATA_WIDTH-1 downto 0);
      dout_a: out std_logic_vector(DATA_WIDTH-1 downto 0);
      dout_b: out std_logic_vector(DATA_WIDTH-1 downto 0)
   );
end xilinx_dual_port_ram_sync;

architecture beh_arch of xilinx_dual_port_ram_sync is
   type ram_type is array (0 to 2**ADDR_WIDTH-1)
        of std_logic_vector (DATA_WIDTH-1 downto 0);
   signal ram: ram_type;
   signal addr_a_reg, addr_b_reg:
          std_logic_vector(ADDR_WIDTH-1 downto 0);
begin
   process(clk)
   begin
     if (clk'event and clk = '1') then
        if (we = '1') then
           ram(to_integer(unsigned(addr_a))) <= din_a;
        end if;
        addr_a_reg <= addr_a;
        addr_b_reg <= addr_b;
     end if;
   end process;
   dout_a <= ram(to_integer(unsigned(addr_a_reg)));
   dout_b <= ram(to_integer(unsigned(addr_b_reg)));
end beh_arch;
```

The code is similar to Listing 11.3 except that the two addresses are first stored in two registers and the registered outputs are used to access memory.

11.4.3 ROM

Despite its name, a ROM (read-only memory) is a combinational circuit and has no internal state. Its output depends only on its input (i.e., address). There is no real embedded ROM in a Spartan-3 device, but it can be emulated by a combinational circuit or a single-port RAM with the write operation disabled. The content of the ROM can be expressed as a constant in the HDL code and the values are loaded to the RAM when the device is programmed. Since the ROM is based in a RAM, the read operation can be asynchronous or synchronous.

ROM with asynchronous read A real ROM is a combinational circuit and thus should not have a buffer or a clock signal. To be consistent with the terms used in this section, we call it a *ROM with asynchronous read.* The template of this type of ROM is shown by an example in Listing 11.5. The code is to implement the hex-to-seven segment LED encoder, similar to that in Listing 3.12. The address of the ROM functions as the 4-bit hexadecimal input and its content is the corresponding LED patterns. The content of the ROM is defined by the HEX2LED_ROM constant and is essentially the truth table of this circuit.

Listing 11.5 Template for a ROM with asynchronous read

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity rom_template is
   port(
      addr: in std_logic_vector(3 downto 0);
      data: out std_logic_vector(6 downto 0)
   );
end rom_template;

architecture arch of rom_template is
   constant ADDR_WIDTH: integer:=4;
   constant DATA_WIDTH: integer:=7;
   type rom_type is array (0 to 2**ADDR_WIDTH-1)
        of std_logic_vector(DATA_WIDTH-1 downto 0);
   -- ROM definition
   constant HEX2LED_ROM: rom_type:=(  -- 2^4-by-7
      "0000001",  -- addr 00
      "1001111",  -- addr 01
      "0010010",  -- addr 02
      "0000110",  -- addr 03
      "1001100",  -- addr 04
      "0100100",  -- addr 05
      "0100000",  -- addr 06
      "0001111",  -- addr 07
      "0000000",  -- addr 08
      "0000100",  -- addr 09
      "0001000",  -- addr 10
      "1100000",  -- addr 11
      "0110001",  -- addr 12
      "1000010",  -- addr 13
      "0110000",  -- addr 14
      "0111000"   -- addr 15
   );
begin
   data <= HEX2LED_ROM(to_integer(unsigned(addr)));
end arch;
```

Note that the memory row is defined in ascending order:

```
... array (0 to 2**ADDR_WIDTH-1) of ...
```

and the first row of the HEX2LED_ROM constant corresponds to the address 00 of the ROM. The rows defined in the HEX2LED_ROM table must be reversed if the rom_type data type is defined in descending order:

```
... array (2**ADDR_WIDTH-1 downto 0) of ...
```

Since there is no address or data buffer in this circuit, the ROM cannot be realized by a block RAM. It is actually synthesized as a combinational circuit with the logic cells. The code can be considered as another form of a selected signal assignment or case statement. This type of ROM is feasible only for a small table. This code template is very general and is not specific to Xilinx devices.

ROM with synchronous read For a large table, it is better to utilize a block RAM to realize the ROM. Since the read operation of a block RAM is controlled and synchronized by a clock signal, the ROM requires a clock signal as well. The template for the ROM with synchronous read is shown in Listing 11.6. It is modified after the rams_21c entity of the *XST Manual*, and the hex-to-seven segment LED encoder is used for demonstration.

Listing 11.6 Template for a ROM with synchronous read

```
-- ROM with synchronous read
-- modified from XST 8.1i rams_21c
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity xilinx_rom_sync_template is
   port(
      clk: in std_logic;
      addr: in std_logic_vector(3 downto 0);
      data: out std_logic_vector(6 downto 0)
   );
end xilinx_rom_sync_template;

architecture arch of xilinx_rom_sync_template is
   constant ADDR_WIDTH: integer:=4;
   constant DATA_WIDTH: integer:=7;
   type rom_type is array (0 to 2**ADDR_WIDTH-1)
        of std_logic_vector(DATA_WIDTH-1 downto 0);
   -- ROM definition
   constant HEX2LED_ROM: rom_type:=(  -- 2^4-by-7
      "0000001",  -- addr 00
      "1001111",  -- addr 01
      "0010010",  -- addr 02
      "0000110",  -- addr 03
      "1001100",  -- addr 04
      "0100100",  -- addr 05
      "0100000",  -- addr 06
      "0001111",  -- addr 07
      "0000000",  -- addr 08
      "0000100",  -- addr 09
      "0001000",  -- addr 10
      "1100000",  -- addr 11
      "0110001",  -- addr 12
      "1000010",  -- addr 13
```

```
        "0110000",   -- addr 14
        "0111000"    -- addr 15
     );
     signal addr_reg: std_logic_vector(ADDR_WIDTH-1 downto 0);
  begin
     -- addr register to infer block RAM
     process (clk)
     begin
        if (clk'event and clk = '1') then
          addr_reg <= addr;
        end if;
     end process;
     data <= HEX2LED_ROM(to_integer(unsigned(addr_reg)));
  end arch;
```

The code is similar to that of the single-port RAM with synchronous read but with a predefined constant. Note that operation of this ROM depends on the clock signal, and its timing is different from that of a normal ROM. Artificial inclusion of the clock signal is necessary to infer a block RAM for the ROM implementation. During synthesis, the software automatically determines whether to use regular logic cells or block RAMs to realize this circuit.

11.5 BIBLIOGRAPHIC NOTES

Two Xilinx application notes, *XAPP464 Using Look-Up Tables as Distributed RAM in Spartan-3 Generation FPGAs* and *XAPP463 Using Block RAM in Spartan-3 Generation FPGAs*, provide detailed information on the distributed RAM and block RAM. Chapter 2 of the *XST User Guide v8.1i*, titled *HDL Coding Techniques*, includes about two dozen HDL code templates to infer various memory configurations.

The comprehensive ISE tutorial, *ISE In-Depth Tutorial*, includes a section on the Core Generator program. Although the program is simple, we need to know the module's basic functionalities and its relevant parameters to create a proper instance.

11.6 SUGGESTED EXPERIMENTS

11.6.1 Block-RAM-based FIFO

In Section 4.5.3, we design a FIFO buffer that uses a register file for storage. To increase its capacity, we can replace the register file with a block RAM-based dual-port RAM module. Derive the HDL code for the new design. Synthesize the verification circuit discussed in Section 4.5.3 with the new FIFO buffer and verify its operation. Note that due to the synchronous read, the behavior of the new FIFO is not completely identical to that of the original FIFO.

11.6.2 Block-RAM-based stack

We discuss the function of a stack in Experiment 4.7.7. To increase its capacity, we can replace the register file with a block RAM-based dual-port RAM module. Repeat the experiment.

11.6.3 ROM-based sign-magnitude adder

We can implement any n-input, m-output function with a 2^n-by-m ROM. Consider the sign-magnitude adder discussed in Section 3.7.2 and assume that a and b are 4-bit input signals. Design this circuit as follows:

1. Write a program in a conventional programming language, such as C or Java, to generate a 2^8-by-4 truth table for this circuit.
2. Follow the ROM template in Listing 11.5 to derive the HDL code. Cut and paste the table to the code.
3. Synthesize the circuit and verify its operation.
4. Check the synthesis report and compare the sizes (in terms of the number of logic cells) of the original implementation and the ROM-based implementation.
5. Expand a and b to 8-bit input signals and repeat steps 1 to 4.

11.6.4 ROM based $\sin(x)$ function

One way to implement a sinusoidal function, $\sin(x)$, is to use a look-up table. Assume that the desired implementation requires 10-bit input resolution [i.e., there are 1024 (2^{10}) points between the input range of 0 and 2π] and 8-bit output resolution [i.e., there are 256 (2^8) points between the output range of −1 and +1]. Let the input and output be the 10-bit x signal and the 8-bit y signal. The relationship between x and y is

$$\frac{y}{2^7} = \sin\left(2\pi\frac{x}{2^{10}}\right)$$

Because of the symmetry of the sin function, we only need to construct a 2^8-by-7 table for the first quadrant (i.e., between 0 and $\frac{\pi}{2}$) and use simple pre- and postprocessing circuits to obtain the values in other quadrants. Design this circuit as follows:

1. Write a program in a conventional programming language to generate the 2^8-by-7 table for the first quadrant.
2. Follow the ROM template in Listing 11.6 to derive the HDL code for the look-up table. Cut and paste the table to the code.
3. Derive the complete HDL code.
4. Derive a testbench to generate the sinusoidal output for three complete periods. This can be done by using a 10-bit counter to generate the 10-bit ROM address for $3 * 2^{10}$ clock cycles. In ModelSim, we can display the y signal in Analog format to emulate the effect of a digital-to-analog converter.

11.6.5 ROM-based $\sin(x)$ and $\cos(x)$ functions

In many communication modulation schemes, the $\sin(x)$ and $\cos(x)$ functions are needed at the same time. Assume that the format of the input and output is similar to that in Experiment 11.6.4. The new circuit has two outputs, y_s and y_c:

$$\frac{y_s}{2^7} = \sin\left(2\pi\frac{x}{2^{10}}\right)$$
$$\frac{y_c}{2^7} = \cos\left(2\pi\frac{x}{2^{10}}\right)$$

Although we can follow the previous procedure and create a new ROM for the $\cos(x)$ function, a better alternative is to share the same ROM for both $\sin(x)$ and $\cos(x)$ functions.

This is based on the observations that $\cos(x)$ is only a phase shift of $\sin(x)$ and that the FPGA's block RAM can provide dual-port access.

Note that this circuit requires essentially a "dual-port ROM." No HDL behaviorial template is given for this type of memory. We need to experiment with HDL codes and to check the synthesis report to ensure that only one block RAM is inferred. It may be necessary to use the Core Generator program or direct HDL component instantiation to achieve this goal.

Construct this special ROM and derive the HDL code for the pre- and postprocessing circuits. Use a testbench similar to that in Experiment 11.6.4 to verify the circuit's operation.