19.7 Read-Only Memories (ROM)

The memories discussed thus far in this chapter have all been Random Access Memories (RAM), meaning they can be both read and written. It is, however, not uncommon to have a need to include memories in a design which can only be read from.

Why might you want to use a ROM in a design? Imagine you are designing a circuit which needs to use $sin(x)$ values as a part of its computation. You have two options for producing $sin(x)$ where x is a multi-bit binary value.

The first option would be to (1) study up on how to compute $sin(x)$ values from some mathematical formulation such as a Taylor Series and then (2) design a circuit to implement that Taylor Series computation. The resulting circuit likely would be very large (require a large silicon area) and/or run slowly (possibly requiring many clock cycles to compute the result).

The second option would be to create a lookup table of $sin(x)$ values using a ROM and, when needed, simply look up $sin(x)$ by providing x to the address lines of the ROM. Along the way you might perform a variety of enhancements to how you organize your table so that it could be used for both $sin(x)$ and $cos(x)$. Many other uses for ROM blocks exist — think of essentially anywhere you might use a table of read-only values (a lookup table) in a software computation.

In reality, a ROM is not a memory at all and there is no *storage* involved. Rather, it is a collection of wires and gates (or simply transistors) which return an output value in response to an input address value. Since the ROM contents will never be changed, there is no reason to waste silicon creating flip flops to hold the memory contents!

Section 10.5 discussed the use of lookup tables (LUTs) for logic. In that context it used the term ROM to describe those lookup tables. That is precisely what a ROM is — it is a hardware implementation of a lookup table. You provide an address and it returns the value associated with that address in the table.

Program 19.7.2 shows the SystemVerilog code for a ROM design. It consists of a case statement that outputs data values in response to address values (note the default assignment for unspecified locations). The synthesis tools know how to convert this description into the circuitry needed to implement the lookup table.

Program 19.7.1 SystemVerilog Code for a ROM

```
module myROM(
       input logic[2:0] Addr,
       output logic[3:0]  DataOut);

  always_comb
  begin
    DataOut = 0;  // Default output assignment
    case (Addr)
      0: DataOut = 5;
      2: DataOut = 9;
      3: DataOut = 10;
      4: DataOut = 2;
      5: DataOut = 11;
    endcase
  end
endmodule
```

You may be thinking "this is all fine and good for tiny lookup tables, but for larger ones this is horribly verbose." Yes, it is. And also, it may not generate the most optimal circuit design.

Many vendors' tools have mechanisms which allow you to specify ROM contents as the contents of a text file which can then be read using the functions *$readmemh()* or *$readmemb()* during simulation and synthesis. For example, here is one such example:

Program 19.7.2 SystemVerilog Code for a ROM

```
module dual_port_rom (
       input logic clk,
       input logic[7:0] addr,
       output logic[15:0] q);

   // Declare the ROM as an array
   logic [15:0] rom[256];

   initial // Read the ROM contents from a file
   begin
      $readmemb("rom_init.txt", rom);
   end

   // Access the ROM
   assign q = rom[addr];
endmodule
```

Further, here is a sample memory initialization file for use with the above program. Consult the web for examples using hex values for the data values.

Program 19.7.3 Contents of File "rom_init.txt"

```
// This is a memory initialization file
//   for use in Verilog.
// Addresses of entries must be specified in HEX
//   and are prefixed with the @ symbol.
// Data values are either in binary (for readmemb)
//   or in hex (for readmemh)
// Unitialized locations will contain 'X'
// You can place underscore symbols in the middle of
//   data values to enhance readability if desired.
// Blank lines are ignored.
// C-style comments are allowed.

// This memory has 16 locations of 8 words each
//   and this file is in binary.

   0101_1111  // Location 0
@2 0010_1001  // Location 2
   0100_1001  // Location 3
   0111_1111  // Location 4
   0011_0011  // Location 5
   0101_1100  // Location 6
@A 1111_1111  // Location 10
   0000_1111  // Location 11
```