Example 40

**Memory: Read-Only Memory**

In this example, we will introduce the concept of memory using a read-only memory (ROM). We will implement a ROM in VHDL and use it with the VGA controller to display your initials on the screen.

**Prerequisite knowledge:**

Example 39 – A VGA Controller

### 40.1 Read-Only Memory

Storage is an important part of designing digital systems. There are several different types of mass storage components. One such memory is called a read-only memory (ROM). As the name implies, it is a memory that has stored values that can only be read and cannot be altered. Common types of read only memories are ROM, EPROM, EEPROM, and Flash Memory. A ROM is typically an integrated circuit that is hardwired with information in the storage. An EPROM is an electronically programmable read only memory. These ICs were more common in the 80s and 90s and could be programmed with a chip programmer and erased using ultraviolet light. Similarly, an EEPROM is an electronically erasable programmable read only memory. An EEPROM can be electronically programmed like an EPROM, only it does not require ultraviolet light to be erased and reprogrammed, it can be reprogrammed electronically from a chip programmer or computer. Flash memory is electronically erasable in bulk achieving a faster reprogramming time.

These types of read-only memory are useful for defining predetermined data at design time that does not need to be changed during runtime. One such example is in the case of video sprites. Consider a game with a dragon as a character. Information about the dragon such as the size, shape, and color need to be stored and accessed when the controller draws the dragon on the screen, but it is never changed by the game; the dragon is always the dragon. The only thing that changes is where it is displayed on the screen.

Read-only memory generally has one input and one output. Since a ROM’s only function is to output stored data on command, only an address input and data output are required. That is, the ROM takes an address as input and the memory places the value stored at that address on the data output.
40.2 A VHDL ROM

Listing 40.1 shows a ROM implemented in VHDL. In a sense, it is a type of programmable read-only memory since we can change its contents in VHDL and reprogram our CPLD or FPGA. Therefore, the entity has been named Prom.

Listing 40.1 prom.vhd

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

entity Prom is
  port (
    addr: in STD_LOGIC_VECTOR (3 downto 0);
    M: out STD_LOGIC_VECTOR (0 to 31)
  );
end Prom;

architecture Prom_arch of Prom is
  type rom_array is array (NATURAL range <>) of STD_LOGIC_VECTOR (31 downto 0);
  constant rom: rom_array := (
    "0111111000001100000111000000010",   --0
    "01000001000011000001101000000010",   --1
    "01000001000010000001101000000010",   --2
    "01000000010010000001101000000010",   --3
    "01000000010010000001101000000010",   --4
    "01000000010010000001101000000010",   --5
    "01000000010010000001101000000010",   --6
    "01000000010010000001101000000010",   --7
    "01000000010010000001101000000010",   --8
    "01000000010010000001101000000010",   --9
    "01000000010010000001101000000010",   --10
    "01000000010010000001101000000010",   --11
    "01000000010010000001101000000010",   --12
    "01000000010010000001101000000010",   --13
    "01000000010010000001101000000010",   --14
    "0111111000001000000011000000010"    --15
  );
begin
  process(addr)
  variable j: integer;
  begin
    j := conv_integer(addr);
    M <= rom(j);
  end process;
end Prom_arch;
```

The entity contains one address input, `addr`, and one data output, `M`. This ROM contains a data bus that is 32 bits wide, `M(0 to 31)`, and contains 16 of these words. In the architecture, a new type is defined, a `rom_array` type. The `rom_array` is declared as an array of 32-bit standard logic vectors. By declaring `(NATURAL range <>),` the compiler will determine the number of words based on the individual instance of that type.
After declaring the new type, an instance, `rom`, of that type is declared as a constant followed by the data in the array. The data can be entered using binary, hex, or other standard VHDL notation. In this case, we entered the data in binary. Since we have declared 16 data values, `rom`’s natural range is from 0 to 15.

Finally, the process that outputs the data based on the input address is given. Using the `conv_integer` function, the input address is converted to an integer, `j`. Next, `j` is used as an index into the constant `rom` array and set to the data output, `M`. The entire process is combinational.

### 40.3 Using the PROM and VGA Controller to Display Your Initials on the Screen

The PROM shown in Listing 40.1 contains sixteen data entries, each a 32-bit word. One use of this PROM would be to indicate whether or not a pixel should be drawn on the screen. A zero would indicate that the pixel should be turned off, while a one indicates that a pixel should be appropriately placed on the screen. Notice that the ones in this PROM spell the initials DMH among zeros that make up the background. This 16 by 32 image contained in the PROM can be used in conjunction with the VGA controller from Example 39 to display those initials on the screen. This 2-dimensional smaller image integrated into a larger scene is called a sprite. Listings 40.2 through 40.4 show a VGA controller that uses the PROM in this fashion. Listing 40.2 shows the entity for the VGA controller and component declaration from the PROM. The arrows point out the difference between this VGA controller and the VGA controller used to draw stripes in Example 39, Listings 39.1 through 39.4. In this instance, the switches have been added as input for a user to control the screen position and color of the initials displayed.

Listing 40.2 _vga_initials.vhd_ (entity and component declarations)

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity vga_prom is

Port ( clk, clr : in std_logic;
      sw: in std_logic_vector(7 downto 0);
      hsync : out std_logic;
      vsync : out std_logic;
      red : out std_logic;
      green : out std_logic;
      blue : out std_logic);
end vga_prom;

architecture vga_prom of vga_prom is

-- Component declaration of the PROM unit where the bitmap is stored
component prom

port(
   addr : in std_logic_vector(3 downto 0);
   M : out std_logic_vector(31 downto 0));
end component;
```
Listing 40.3 gives the constants for the standard VGA controller at a 640x480 resolution. In addition, constants $w$ and $h$ have been added that refer to the width and height of the sprite containing the initials, respectively. In this case, the initials have been created using 32 bits across and 16 lines high according to the data in the PROM. Next, signals to keep track of the current column and row, $C1$ and $R1$, have been declared. Signals $rom\_addr$ and $M$ will be used when wiring the PROM component to the VGA controller. While the $rom\_addr$ signal will refer to the current address for the PROM, the $rom\_pix$ signal will refer to a particular bit in the word that is being addressed. In this fashion, $rom\_addr$ is related to the row and $rom\_pix$ is related to the column. Together, they refer to a single bit in the bitmap which will indicate whether or not a pixel should be placed at the corresponding location on the screen. $Spriteon$ is used to signal whether or not the current pixel given by $(vc, hc)$ is in the region where the sprite is to be drawn.

Listing 40.3  vga_initials.vhd  (continued, constant and signal declarations)

```vhdl
constant hpixels: std_logic_vector(10 downto 0) := "01100100000";
--Value of pixels in a horizontal line = 800
constant vlines: std_logic_vector(10 downto 0) := "01000001001";
--Number of horizontal lines in the display = 521
constant hbp: std_logic_vector(10 downto 0) := "00010010000";
--Horizontal back porch = 144 (128+16)
constant hfp: std_logic_vector(10 downto 0) := "01100010000";
--Horizontal front porch = 784 (128+16+640)
constant vbp: std_logic_vector(10 downto 0) := "00000011111";
--Vertical back porch = 31 (2+29)
constant vfp: std_logic_vector(10 downto 0) := "00111111111";
--Vertical front porch = 511 (2+29+480)
signal hc, vc: std_logic_vector(10 downto 0);
--These are the Horizontal and Vertical counters
signal vidon : std_logic;
--Tells whether or not it's ok to display data
signal vsenable: std_logic;
--Enable for the Vertical counter
constant w: integer := 32;
constant h: integer := 16;
signal C1, R1: std_logic_vector(10 downto 0);
signal rom_addr, rom_pix: std_logic_vector(10 downto 0);
signal M: std_logic_vector(0 to 31);
signal spriteon: std_logic;
```

Listing 39.2 and 39.3, from the previous example, show the beginning of the architecture including the horizontal counter process, vertical counter process, the horizontal sync signal, and the vertical sync signal. All of these processes and signals remain unchanged in this example. Following these processes and signals, Listing 40.4 picks up from the $vidon$ signal used to determine the visible video regions. The new $spriteon$ signal is similar to the $vidon$ signal in that it is high if the current pixel coordinate designated by $(vc, hc)$ is in the $16 \times 32$ region designated for the sprite. To accomplish this, it is defined to be high as a function of both $hc$ and $vc$. For it to be high,
$hc$ must be between $hbp + C1$ and $hbp + C1 + w$. That is, between the beginning of the visible horizontal range plus the column offset, $C1$, and that plus the width of the sprite, $w$, which is 32 in this example. Similarly, $vc$ must be between $vbp + R1$ and that plus the sprite height, $h$, which is 16 in this example. These conditions define the visible sprite area.

Listing 40.4  vga_initials.vhd  (continued, visibility, sprite signals and PROM port map)

```vhdl
vidon <= '1' when (((hc < hfp) and (hc > hbp)) and (vc < vfp) and (vc > vbp))) else '0';
--Enable video out when within the porches
spriteon <= '1' when (((hc >= C1 + hbp) and (hc < C1 + hbp + w)) and (vc >= R1 + vbp) and (vc < R1 + vbp + h))) else '0';
--Enable sprite video out when within the sprite region
--set C1 and R1 using switches
C1 <= "00" & SW(3 downto 0) & "00001";
R1 <= "00" & SW(7 downto 4) & "00001";
rom_addr <= vc - vbp - R1;
rom_pix <= hc - hbp - C1;

process (spriteon, vidon, rom_pix, M)
variable j: integer;
begin
red <= '0';
green <= '0';
blue <= '0';
if spriteon = '1' and vidon = '1' then
j := conv_integer(rom_pix);
red <= M(j);
green <= M(j);
blue <= M(j);
end if;
end process;

P1: prom port map
(addr => rom_addr(3 downto 0), M => M);
end vga_prom;
```

$C1$ and $R1$ are set by the switches as shown. At a minimum, the sprite can start at pixel coordinate $(1,1)$ and at a maximum $(481, 481)$ which is off the screen. The first line of the sprite region corresponds with the first row in the sprite bitmap which is at address zero. Likewise, the first column of the sprite region corresponds to the first pixel in the row of data, pixel zero. Therefore, $rom\_addr$ is set to $vc - vbp - R1$ and $rom\_pix$ is set to $hc - hbp - C1$. Together, the coordinate $(rom\_addr, rom\_pix)$ identifies the pixel in the sprite bitmap.

Next, a process for controlling the pixel colors is shown where pixels are only displayed in the visible video region ($vidon='1'$) and visible sprite region ($spriteon='1'$). Whether they are on or off in that region depends on the $j^{th}$ bit of $M$ where $j$ is the integer value of $rom\_pix$ (the column) and $M$ is the data from the PROM at address $rom\_addr$. 
Finally, the PROM is port mapped in to the design using \textit{rom_addr} and \textit{M}. Fig. 40.1 illustrates how the \textit{rom_addr} and \textit{rom_pix} are calculated as the \textit{vc} and \textit{hc} proceed to increment relative to the \textit{vbp}, \textit{hbp}, and selected location for the sprite, \textit{R1} and \textit{C1}. The top of Fig. 40.1 shows the display screen. The bitmap for the initials in the sprite PROM is shown at the bottom of the figure.

The example is Fig. 40.1 uses a selected sprite position of \textit{R1} = 1 and \textit{C1} = 1. The \textit{rom_addr} and \textit{rom_pix} will therefore be calculated as shown in the legend labeled ‘Given’ in Fig 40.1. As the vertical counter, \textit{vc}, and horizontal counter, \textit{hc}, increment past the horizontal and vertical blanking periods, to the first pixel where video is visible (32,145), \textit{spriteon} is high. As shown in Sample 1, the \textit{rom_addr} = 0 yielding the first row of data in the sprite bitmap from the PROM while \textit{rom_pix} = 0 selecting the first bit of that data word. As the horizontal counter continues across the screen, \textit{rom_pix} also moves across the data word. Once the horizontal counter increments past the visible sprite region to \textit{hc} = 177, \textit{spriteon} becomes low and data from the PROM is not relevant. Finally, as the vertical counter moves down the screen, \textit{rom_addr} moves to successive
data words. Similarly, once the vertical counter has incremented to $vc = 48$, spriteon becomes low and data from the PROM is no longer relevant.