Politecnico di Torino

Electronic Engineering
Embedded Systems

Microelectronic Systems Read-only and Read-write Memories

Authors:
Andrea Paternò 206526
Giuseppe Luca Macaluso 206009

November 27, 2014

1 Introduction

One of the main features in Harvard architectures is the net separation of the Instruction Ram and the Data Ram, along with their buses. The Instruction Ram is usually a static Read-Only Memory, while the Data Ram a true Random Access Memory. Therefore the following implementation works in this direction, providing two different components, one to be used for the intruction management, and one for the data.

However, their management is quite similar.

2 Read-only RAM

The read-only ram works in two different ways, depending on the value of the **RST** signal.

The ROM entity is listed below, and is to be used as a reference of this chapter's signal names.

Listing 1: Read Only Memory entity

```
entity ROMEM is
    generic (
        FILE_PATH    : string;                 -- ROM data file
        ENTRIES      : integer := 128;         -- Number of lines in the ROM
        WORD_SIZE    : integer := 32;          -- Number of bits per word
        DATA_DELAY   : natural := 2            -- Delay ( in # of clock cycles )
    );
    port (
        CLK             : in std_logic;
        RST             : in std_logic;
        ADDRESS         : in std_logic_vector(WORD_SIZE - 1 downto 0);
        ENABLE          : in std_logic;
        DATA_READY      : out std_logic;
        DATA            : out std_logic_vector(2*WORD_SIZE - 1 downto 0)
    );
end ROMEM;
```

2.1 Reset high

If **RST** is high, than the ROM refreshes its internal buffer contents with the ones in the dump file, whose path is to be set in the **FILE_PATH** variable. The data to be provided must be in hexadecimal form, just as the compilation dump of the compiler provided. The **ENTRIES** variable holds the number of instructions (data words) that the ROM can hold.

The **WORD_SIZE** variable is already set to 32, so as to be compliant with the compiler itself, which encodes the ASM instruction is a classical MIPS 32-bit fixed-length encoding.

2.2 Reset low

If **RST** is low, than depending on the value of the **ENABLE** signal, the ROM is either ready to accept incoming requests (**ENABLE** high) or simply remains in its idle state (**ENABLE** low).

The **DATA_DELAY** variable is used to simulate the delay in real-world memories, and is expressed in numbers of clock cycles.

Usage In order to send a request to the memory, we must drive the signals as follows:

- CLK -> Clock of the memory. Can be equal to the one used by the uP.
- RST -> Low
- ENABLE -> High
- ADDRESS -> The address is a vector of **WORD_SIZE** bits, representing the index of the word to be read. It has to be stable until the **DATA_READY** signal becomes high.

After **DATA_DELAY** clock cycles, the ROM will drive **DATA_READY** high and place the data in the bus named **DATA**.

3 Read-write RAM

Many of the signals in the RAM entity maintain the same name and functionality as the ones in the ROM. Also its management works in the same way, but for a few changes detailed below.

Listing 2: Read Write Memory entity

```
entity RWMEM is
    generic(
            FILE_PATH: string;                  -- RAM output data file
            FILE_PATH_INIT: string;             -- RAM initialization data file
            WORD_SIZE: natural := 32;           -- Number of bits per word
            ENTRIES:      natural := 128;       -- Number of lines in the ROM
            DATA_DELAY: natural := 2            -- Delay ( in # of clock cycles )
        );
    port (
            CLK                 : in std_logic;
            RST                 : in std_logic;
            ADDRESS             : in std_logic_vector(WORD_SIZE - 1 downto 0);
            ENABLE              : in std_logic;
            READNOTWRITE        : in std_logic;
            DATA_READY          : out std_logic;
            INOUT_DATA              : inout std_logic_vector(2*WORD_SIZE-1 downto 0)
        );
end RWMEM;
```

The **DATA** bus in the ROM component was replaced by the **INOUT_DATA** bus, which (guess?) this time is driven by tri-state buffers, as, depending on the mode of operation, holds either the data to be written in the RAM, or read from it.

The type of operation to be performed in controlled by the **READ-NOTWRITE** signal, which, if high, tells the RAM that we're interested in reading the memory cell (and its adjacent one, in this current implementation) located at **ÆDDRESS**; while if low, that we're interested in writing to those cells with the contents of the bus.

In order to reduce the number of pins of the component, in fact, we chose to use a shared buffer for both the operations, excluding the case in which there is a concurrent writing and reading.

4 The testbench

The testbench only instantiates the DLX, RAM and ROM components and connects them together. It also generates the global clock and reset signals.

Listing 3: Component Instantiation

```
-- IRAM
IRAM : ROMEM
    generic map ("/home/gandalf/Desktop/dlx/rocache/hex.txt")
    port map (CLK, RST, IRAM_ADDRESS, IRAM_ENABLE, IRAM_READY, IRAM_DATA);

-- DRAM
DRAM : RWMEM
    generic map (
        "/home/gandalf/Desktop/dlx/rwcache/hex_init.txt",
        "/home/gandalf/Desktop/dlx/rwcache/hex.txt"
    )
    port map ( CLK, RST, DRAM_ADDRESS, DRAM_ENABLE,
        DRAM_READNOTWRITE, DRAM_READY, DRAM_DATA
    );

-- DLX
GIANLUCA : DLX
    port map ( CLK, RST, IRAM_ADDRESS, IRAM_ENABLE, IRAM_READY,
        IRAM_DATA, DRAM_ADDRESS, DRAM_ENABLE, DRAM_READNOTWRITE,
        DRAM_READY, DRAM_DATA
    );
```