



Tutorial 5: Computer Components - Memory Interface

Tutor Notes

1 Introduction

This tutorial gives you experience in interfacing memory devices to a CPU. It builds upon the LCD interface completed in Tutorial 4 by adding a number of memory devices to the CPU. The LCD display can be used for display of diagnostic information when testing the memory interface.

Before attempting this tutorial, Tutorial 4 must be completed using the solution provided on the ELEC4605 web site. Then, the Tutorial 4 solution can be modified into a hierarchy, in keeping with the principles of Digital Systems Designs. This includes moving the address decoding into a separate module. Finally, the memory devices will be added. This also requires the addition of a data-bus multiplexer to direct data back to the CPU core.

Read through the complete tutorial exercise first to understand what the final outcome should be. The final top-level design is shown in Figure 7. Don't be too concerned if you are unable to complete the tutorial as some of the material covered in this tutorial exercise will be repeated for the laboratory project.

Students will have to complete the first exercise from Tutorial 4 first. The solution is on the unit web site. This designed will be modified slightly as part of this tutorial to form a hierarchy as shown in Figure 7.

2 Complete Tutorial 4 Exercise

The solution to Tutorial 4 is on the ELEC4605 web site. If you have not already completed the first exercise from Tutorial 4, do this now as Tutorial 5 will build upon it.

Refer to the Tutorial 4 solutions on the ELEC4605 web site.

It may be possible for students to skip this step and go straight onto the next step if they have not already completed Tutorial 4. However, if the students have found Tutorial 4 difficult to complete last time, it would be best that they got Tutorial 4 working completely first.

3 Design Hierarchy

The solution from Tutorial 4 has the complete design on one schematic. It is good practice to divide a design into a number of functional modules in the form of a hierarchy as covered in the Digital Systems Design topic in lectures.

Copy your solution from Tutorial 4 into another directory to complete the Tutorial 5 exercise. Change the design by moving the LCD interface onto another schematic. This schematic should look like that shown in Figure 1. The VHDL code for the LCD interface also has to be changed to use a chip select input rather than decoding the address lines. The **BUSY** signal can also be connected directly to **DO0** without the need to use the **MEMRD** signal (done in the VHDL code).

Ensure that students copy their design from Tutorial 4 into a new directory. This way they have a copy of the previous working design and will hopefully avoid the problems that we had last time with the software.

The VHDL code for the LCD interface is on Page 6.

A separate VHDL symbol must be added to the top level that takes in the address lines and generates a chip select output for the LCD display. To make decoding easier, make the chip select line active for a 1K block starting at 0×0000 . The symbol for the decoder is shown in Figure 2. Note that it also shows chip select outputs used for the memory devices which will be added later.

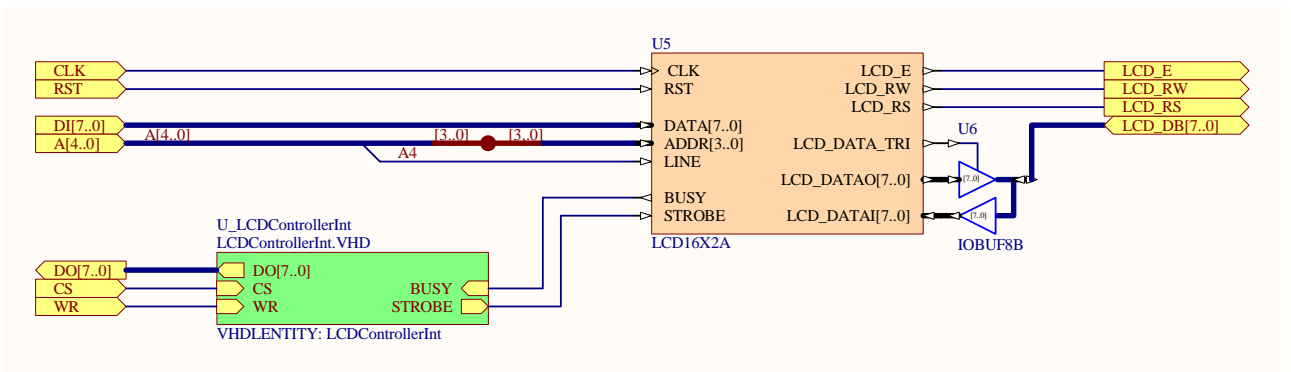


Figure 1: The LCD interface schematic.

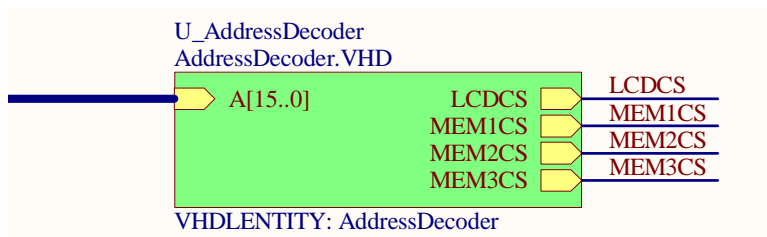


Figure 2: Address decoder used to select the devices connected to the CPU.

The VHDL code for the complete decoder is on Page 8. The simulation results are shown in the next section when the memory devices have been added.

The test program used in Tutorial 4 should be used to check that the design is working correctly. You should also simulate the VHDL code for the LCD interface and the address decoder to check for correct operation.

The VHDL code for the stimulus part of the test bench for the LCD interface is on Page 6. The simulation results are shown in Figure 3. Note that the simulation shows a practical situation. The **BUSY** flag is read first then a write operation

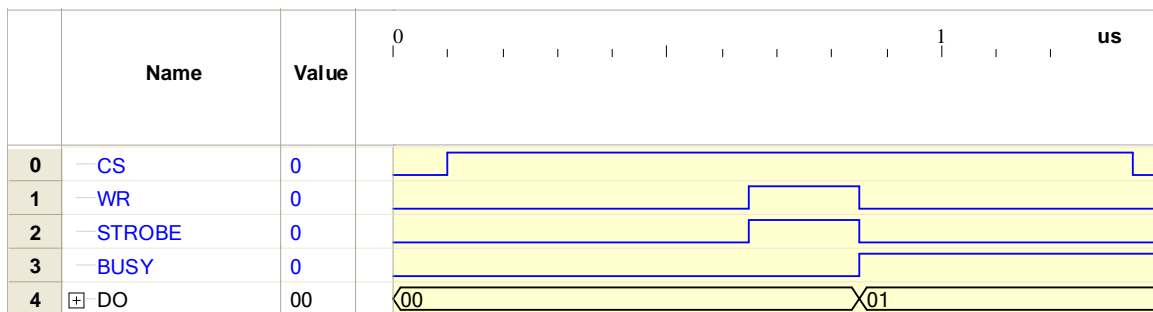


Figure 3: Simulation of the LCD controller interface.

occurs and then the **BUSY** flag is read again and shown to be high. However, in this case, no read signal is used.

4 Add Memory Devices

The final task is to add three 1K memory devices to the design starting at address 0x8000. Use the RAMSE_8x1K components. This is similar to the device used for the program memory but it also has an enable signal. To write to the memory, the enable signal must also be asserted otherwise the write-enable signal will have no effect. The enable signal uses positive logic (even though the documentation suggests otherwise).

First modify the address decoder to include the three chip-select signals used to select the memory devices for the address ranges mentioned above. Test that the chip-select signals for the memory devices work as expected.

As mentioned previously, the VHDL code for the complete decoder is on Page 8. The VHDL code for the stimulus part of the test bench for the address decoder is also on Page 8. The simulation results are shown in Figure 4.

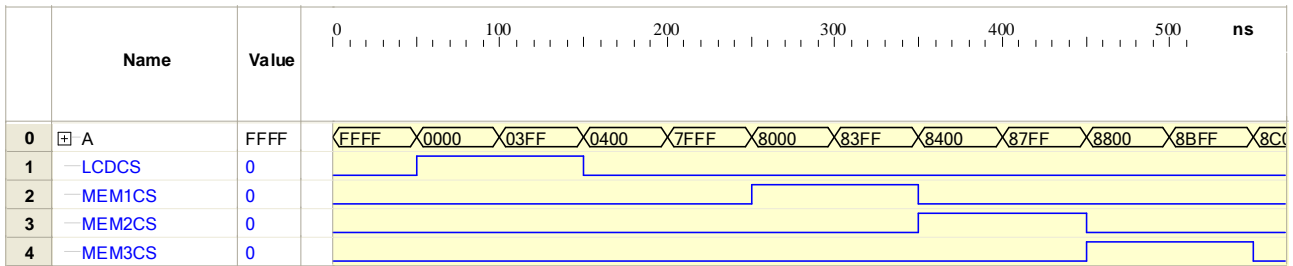


Figure 4: Simulation of the address decoder.

As there are now four devices connected to the CPU (the LCD display and the three memory devices) it is not possible to connect all the data output signals back to the CPU. Instead a data-bus multiplexer must be used to select one data source which will be connected back to the CPU. Implement this data-bus multiplexer in VHDL. The symbol is shown in Figure 5. You should also simulate the data-bus multiplexer to ensure correct operation.

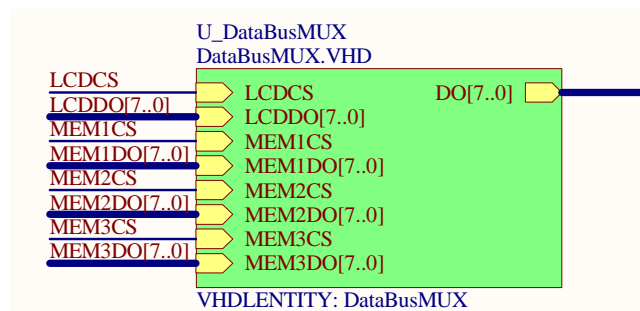


Figure 5: Data-bus multiplexer to select which data lines are connected to the CPU.

The VHDL code for the data-bus multiplexer is shown on Page 9. The VHDL code for the stimulus part of the test bench for the data-bus multiplexer is also on Page 9. The simulation results are shown in Figure 6.

Another method for connecting data signals back to the CPU will be used for the laboratory project. This will highlight that there are a number of ways in which a design can be decomposed.

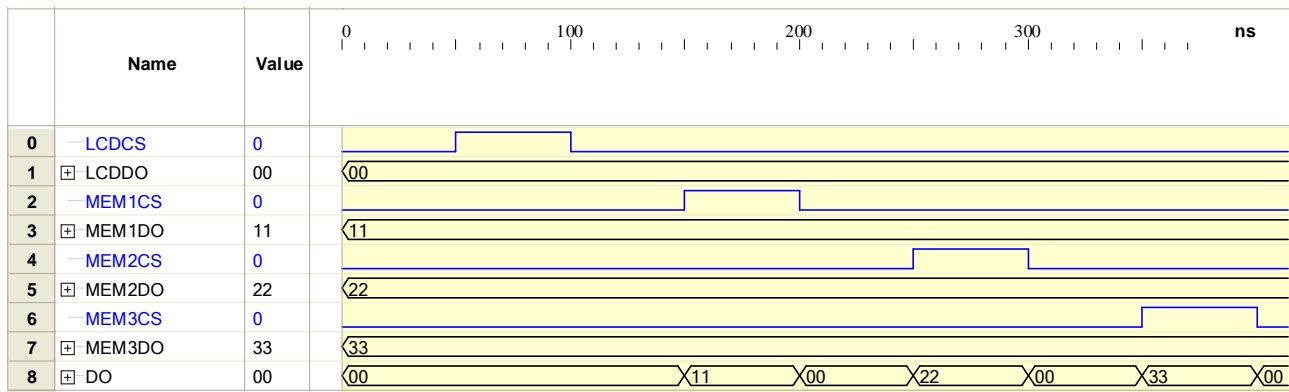


Figure 6: Simulation of the data-bus multiplexer.

5 Write Test Program

Write a test program to check that the memory devices are working correctly. You can use the LCD display for displaying diagnostic messages.

An example of some test code is shown on Page 10. It simply writes to the memory devices and then reads back the values and checks to see if it is the expected value. Note that an odd-length sequence has been used (255 rather than 256). You may want to try more-complex methods for testing that the memory interface is working correctly.

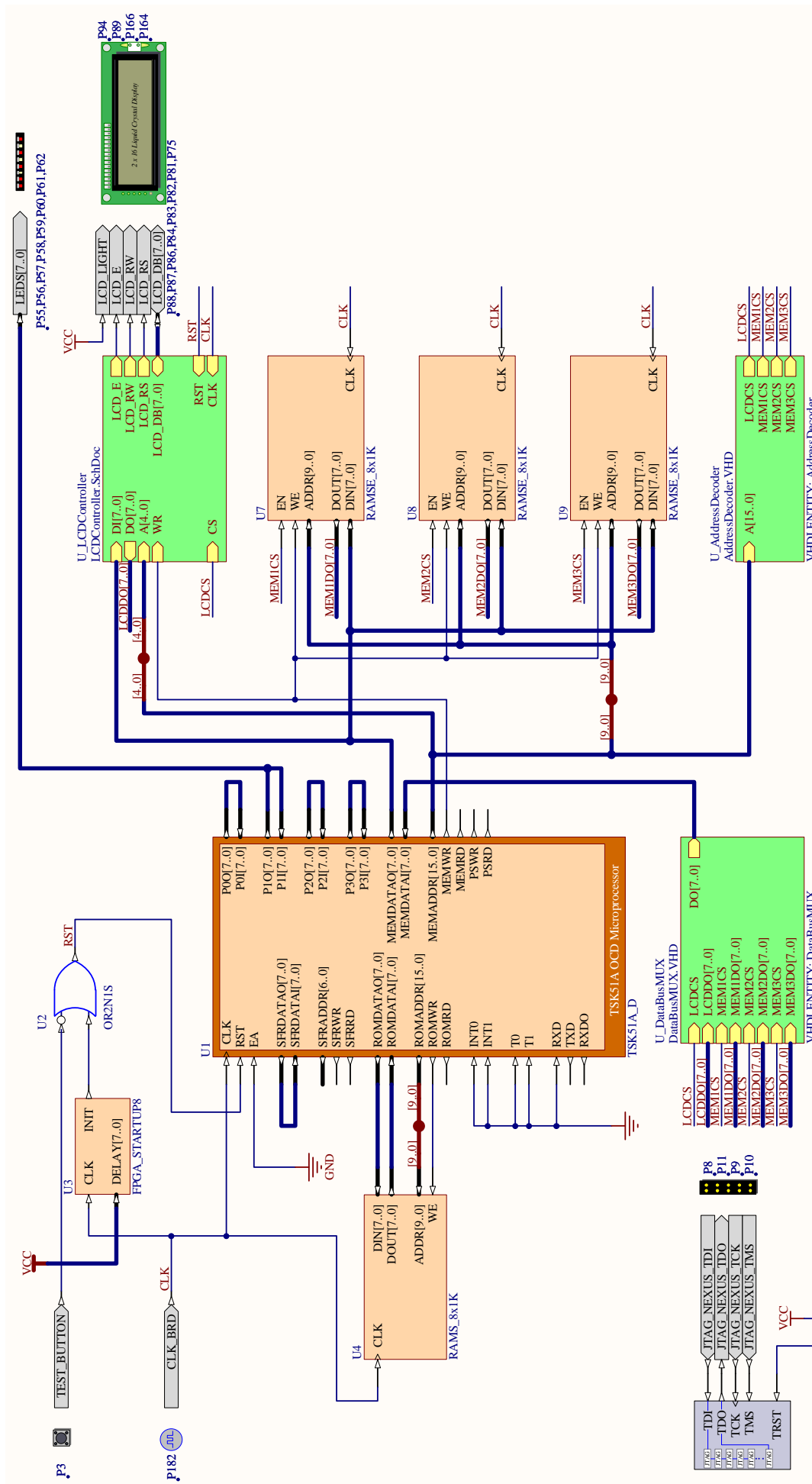


Figure 7: Complete design containing LCD display and three extra memory devices.

VHDL Code

LCD Controller Interface

```

— ELEC4605 Computer Engineering
— LCDControllerInt.vhd
— Peter Stepien
— 2006

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity LCDControllerInt is
  port(DO      : out std_logic_vector(7 downto 0);
        CS     : in  std_logic;
        WR     : in  std_logic;
        BUSY   : in  std_logic;
        STROBE : out std_logic);
end LCDControllerInt;

architecture behaviour of LCDControllerInt is
begin
  — Write to the LCD controller
  STROBE <= WR when CS='1' else '0';

  — Read from LCD controller (only the busy signal)
  DO(7 downto 1) <= b"0000000";
  DO(0) <= BUSY;
end behaviour;

```

LCD Controller Interface Test Bench Stimulus

```

STIMULUS0: process
begin
  — insert stimulus here
  — Set initial values
  CS <= '0';
  WR <= '0';
  BUSY <= '0';
  wait for 100ns;
  — Read cycle to LCD controller
  — Only reading the BUSY flag
  CS <= '1';
  wait for 150ns;
  wait for 200ns; — Read signal asserted
  wait for 100ns;
  — Write cycle to LCD controller
  — Note data connected directly to the controller
  CS <= '1';
  wait for 100ns;
  WR <= '1'; wait for 200ns;
  WR <= '0';
  BUSY <= '1'; — Assume BUSY flag set since LCD processing commant
  wait for 50ns;
  — Read cycle to LCD controller
  — Only reading the BUSY flag
  CS <= '1';
  wait for 150ns;

```

```
wait for 200ns; — Read signal asserted  
wait for 100ns;  
— Finish by not selecting LCD  
CS <= '0';  
wait for 100ns;  
wait;  
end process;
```

Address Decoder

```

— ELEC4605 Computer Engineering
— AddressDecoder.vhd
— Peter Stepien
— 2006

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity AddressDecoder is
  port(A      : in std_logic_vector(15 downto 0);
        LCDCS : out std_logic;
        MEM1CS : out std_logic;
        MEM2CS : out std_logic;
        MEM3CS : out std_logic);
end AddressDecoder;

architecture behaviour of AddressDecoder is
begin
  — LCD controller (1k address starting at 0x0000)
  LCDCS <= '1' when A(15 downto 10)=b"000000" else '0';
  — MEM1 (1k address starting at 0x8000)
  MEM1CS <= '1' when A(15 downto 10)=b"100000" else '0';
  — MEM2 (1k address starting at 0x8400)
  MEM2CS <= '1' when A(15 downto 10)=b"100001" else '0';
  — MEM3 (1k address starting at 0x8800)
  MEM3CS <= '1' when A(15 downto 10)=b"100010" else '0';
end behaviour;

```

Address Decoder Test Bench Stimulus

```

STIMULUS0: process
begin
  — insert stimulus here
  — Initialise
  A <= x"ffff"; wait for 50ns;
  — Go through various addresses
  — LCD
  A <= x"0000"; wait for 50ns; — Start LCD
  A <= x"03ff"; wait for 50ns; — End LCD
  A <= x"0400"; wait for 50ns; — After LCD
  — MEM1
  A <= x"7fff"; wait for 50ns; — Before MEM1
  A <= x"8000"; wait for 50ns; — Start MEM1
  A <= x"83ff"; wait for 50ns; — End MEM1
  — MEM2
  A <= x"8400"; wait for 50ns; — Start MEM2
  A <= x"87ff"; wait for 50ns; — End MEM2
  — MEM3
  A <= x"8800"; wait for 50ns; — Start MEM3
  A <= x"8bff"; wait for 50ns; — End MEM3
  A <= x"8c00"; wait for 50ns; — After MEM3
  wait;
end process;

```

Data Bus Multiplexer

```

— ELEC4605 Computer Engineering
— DataBusMUX.vhd
— Peter Stepien
— 2006

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity DataBusMUX is
  port(DO      : out std_logic_vector(7 downto 0);
        LCDDO  : in  std_logic_vector(7 downto 0);
        LCDCS  : in  std_logic;
        MEMIDO : in  std_logic_vector(7 downto 0);
        MEM1CS : in  std_logic;
        MEM2DO : in  std_logic_vector(7 downto 0);
        MEM2CS : in  std_logic;
        MEM3DO : in  std_logic_vector(7 downto 0);
        MEM3CS : in  std_logic);
end DataBusMUX;

architecture behaviour of DataBusMUX is
begin
  — Select data output from LCD, MEM1, MEM2 or MEM3
  — based on MEM1CS, MEM2CS and MEM3CS
  DO <= MEMIDO when MEM1CS='1' else
        MEM2DO when MEM2CS='1' else
        MEM3DO when MEM3CS='1' else
        LCDDO;
end behaviour;

```

Data Bus Multiplexer Test Bench Stimulus

```

STIMULUS0: process
begin
  — insert stimulus here
  — Initialise
  LCDCS <= '0'; LCDDO <= x"00";
  MEM1CS <= '0'; MEMIDO <= x"11";
  MEM2CS <= '0'; MEM2DO <= x"22";
  MEM3CS <= '0'; MEM3DO <= x"33";
  wait for 50ns;
  — Test each select input
  LCDCS <= '1'; wait for 50ns; LCDCS <= '0'; wait for 50ns;
  MEM1CS <= '1'; wait for 50ns; MEM1CS <= '0'; wait for 50ns;
  MEM2CS <= '1'; wait for 50ns; MEM2CS <= '0'; wait for 50ns;
  MEM3CS <= '1'; wait for 50ns; MEM3CS <= '0'; wait for 50ns;
  wait;
end process;

```

Test Code

```

// ELEC4605 Computer Engineering
// Tutorial #5
// Program to test memory interface using the LCD
// display for diagnostic output.
// Peter Stepien
// 2006

__xdata volatile char *LCD = (__xdata volatile char *)0x0000;
__xdata volatile unsigned char *MEM = (__xdata volatile unsigned char *)0x8000;
__xdata volatile unsigned char *MEM1 = (__xdata volatile unsigned char *)0x8000;
__xdata volatile unsigned char *MEM2 = (__xdata volatile unsigned char *)0x8400;
__xdata volatile unsigned char *MEM3 = (__xdata volatile unsigned char *)0x8800;

void write_string(__rom char *string)
{
    unsigned int i;

    i=0;
    while (string[i]!=0)
    {
        while ((LCD[0] & 0x01) == 1) {}
        LCD[i]=string[i];
        i++;
    }
}

void pause ()
{
    unsigned long i;
    for (i=0;i<0xffff;i++)
    {
        __asm("nop");
    }
}

void main(void)
{
    unsigned int i;
    unsigned char j;
    unsigned char count=0;

    write_string(".....");
    P1=0;
    for (;;)
    {
        write_string("Test_memory_..."); pause();
        // Write an odd-length byte sequence
        j=0;
        for (i=0;i<0xbff;i++)
        {
            MEM[i]=j;
            j++;
            if (j==255) j=0;
        }
        // Check memory
        j=0;
        for (i=0;i<0xbff;i++)
        {
            if (MEM[i]!=j)

```

```
    {  
        // Data not read back correctly  
        for (;;)   
        {  
            write_string("MEMORY_ERROR....."); pause();  
            write_string("....."); pause();  
        }  
    }  
    j++;  
    if (j==255) j=0;  
}  
write_string("Memory test OK. "); pause();  
P1=count++;  
}  
}
```