Csc343 LAB

Register File

Objective
Design a 4*4 register file and test it on VSIM, QUARTUS, and UP3 board.

What is a register file
A register file is the central storage of a microprocessor. Most operations involve using or modifying data stored in the register file. The register file that will be designed has 4 locations(such as 00,01,10,11). Figure 1 is the suggested block diagram.

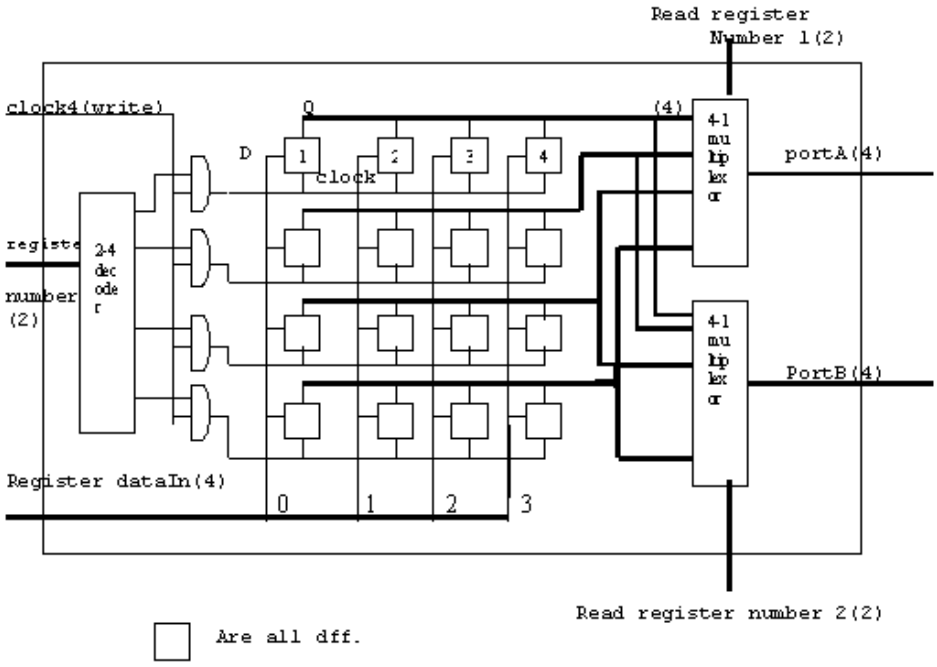


figure 1

Our Register File Description

If a register file is 4*4, it means we have 4 registers and each register is 4 bits wide. Each cell of the register file, in this case, is constructed by a Dff. Therefore, a register is constructed by 4 DFFs, for example, D1,D2,D3,and D4. Each of the four registers has a clock input (positive edge triggered), a data input, and a data output. This collection of registers is managed by one '2:4 decoder' and two**'**4:1 multiplexors'.
There are two output ports('PortA' and 'PortB') for the register file. Therefore, two registers can be read simultaneously. Each output of the registers is connected to two multiplexors. By changing the 'Read register number 1' and **'**Read register number 2',

we can choose which two registers we want to extract data into the ‘PortA’ bus and ‘PortB’ bus.
We use clock4(write) combining with ‘and’ gates to control the register writing. When a clock comes in, a candidate register will be selected to be updated according to which ‘and’ gates is ON (on of the input = ‘1’). This is determined by the ‘2:4 decoder’.

Compiling and Simulating your register file using VSIM

Get the ‘registerfile.vhd’ file from the Appendix B. You need to input your design of add, dff, and multiplexor into your project and make them as the corresponding names that the registerfile.vhd calls.
Design your own test_registerfile.vhd.
In your test file designing, first step, generate four clocks and combine the output of the ‘2:4 decoder’(00,01,10,11) to input the data(0000, 1010,0011,1111) into register1, register2, register3, and register4. Then, extract these data from the corresponding registers by giving different values(for example, 00,01,10,11) to ‘read register number 1’ and ‘read register number 2’.
After completing your test file design, according your pervious knowledge of how to run VISM, you should get the wave form as figure 2.

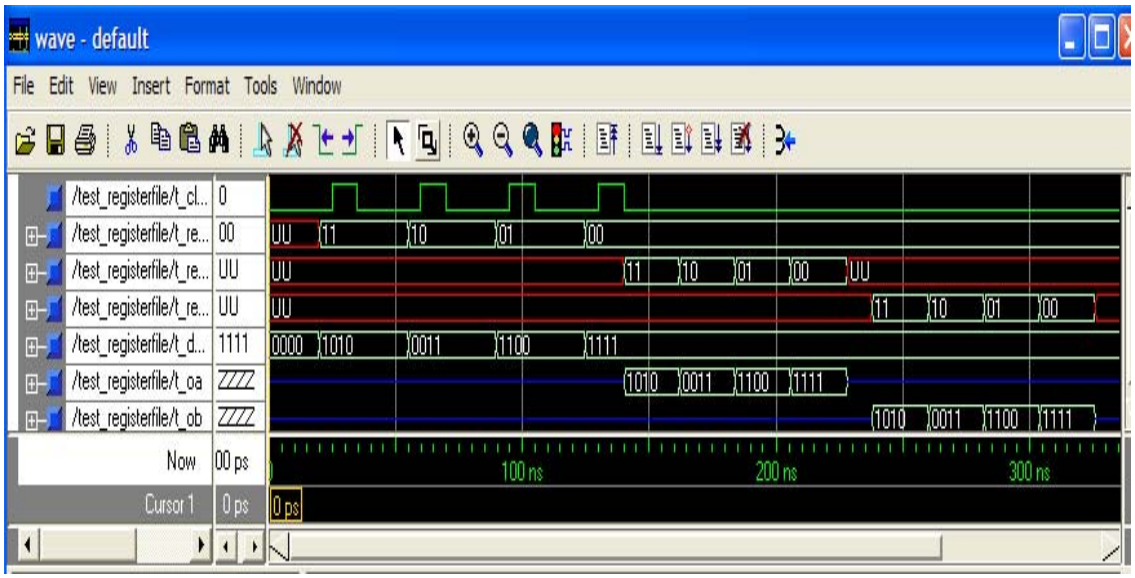


figure 2

Compiling and Simulating your register file using QUARTUSII

Open QuartusII and create a new object named LABN. Then do the same procedures except in **new project wizard of page 3** select the corresponding parameters for UP3.

In QuartusII, create symbol for **registerfile.vhd** called **registerfile.bdf**. Create LABN.bdf file by adding pins to registerfile.bdf. Then do compilation and simulation procedures.

Putting your design into UP3 board

In order to put your design into the corresponding board, usually, we need to do input preparation and output preparation.

Input preparation

1.Using push-button to generate clocks and count the number of clocks using the figure 3's suggestion.

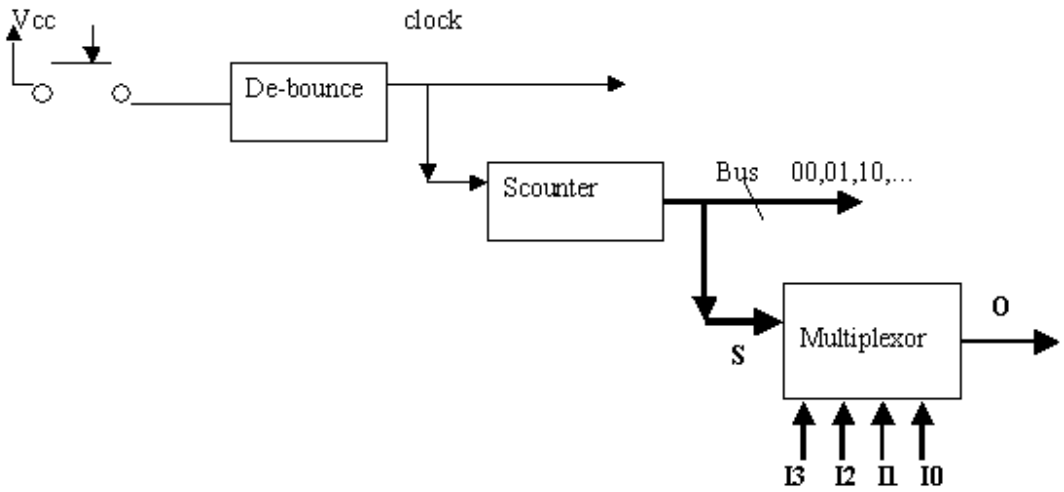


figure 3

The function of de-bounce is to filter the bounces which generated by the push-button and output a clock with a good shape, you can find this function in the disk of the book or you can design it by yourself.

The Souncter function counts the number of clocks. You can use the module 4 counter that we already have, or you can use megafunctions by specify the inputs and outputs. The purpose of using Souncter function is to simultaneously select address for the register file.

The purpose of using multiplexor here is to simultaneously select the input data to be written into the selected register according to the push-button.

Output preparation

LED display (It is not a case in UP3 but I prefer include it)

For the register file, we have two output buses, ‘PortA’ and ‘PortB’. These ports all output binary numbers. In order to show the corresponding number in LED display, we need to design our own CONNECTOR to convert binaries into the corresponding LED displayer codes. Table 1 shows the relationship.

Table 1

Hexadecimal	Binary code	LED display code
1	0001	1100111
2	0010	0010010
3	0011	0000110
4	0100	1001100
5	0101	0100100
6	0110	0100000
7	0111	0001111
8	1000	0000000
9	1001	0001100
A	1010	0001000
B	1011	1100000
C	1100	0110000
D	1101	1000010
E	1110	0110000
F	1111	0111000
0	0000	0000001

CONNECTOR can be designed according to Table 1.

Figure 4 is the output preparation design. This design will extract data from these 4 registers according to clock serially. For example, when you first time push the button, it will show the data has the address 01, second, the data of address 10, third, the data of address 11, fourth, the data of address 00.

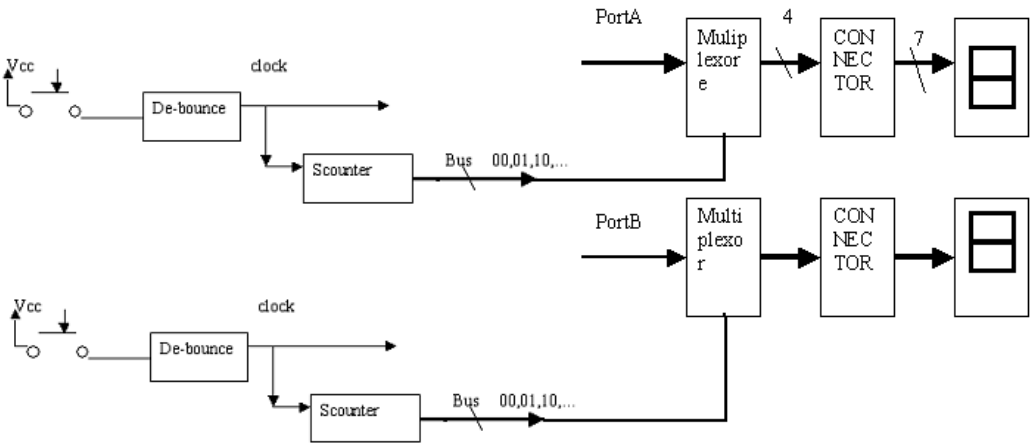


figure 4.

Appendix A shows the result of VISM according these design ideas. The clocks of the first three lines are generated by three push-buttons. The first clock is for generating addresses(00,01,10,11) and inputting data (0001, 0010, 0011,0100) into registers with these addresses in the register file. The second clock is for reading data from the register file into 'PortA' at the same time converting binary code into LED display code, and the third clock is doing the same thing for 'PortB'.

LCD display

Because we have two output ports 'PortA' and 'PortB' and only one LCD display in UPS3 board, you need to add one 2:1 multiplexor for the selection.

The LCD display is totally different from the LED. Either you can design your own LCD CONNECTOR or use the book provided **Lcd_display.vhd.** In both ways, to make LCD working, we need to have an enable, a data bus, a read and write signal, and a read and write memory address. And these signals have strict requirements(details see appendix C) . Figure 5 shows the basic concept design for LCD.

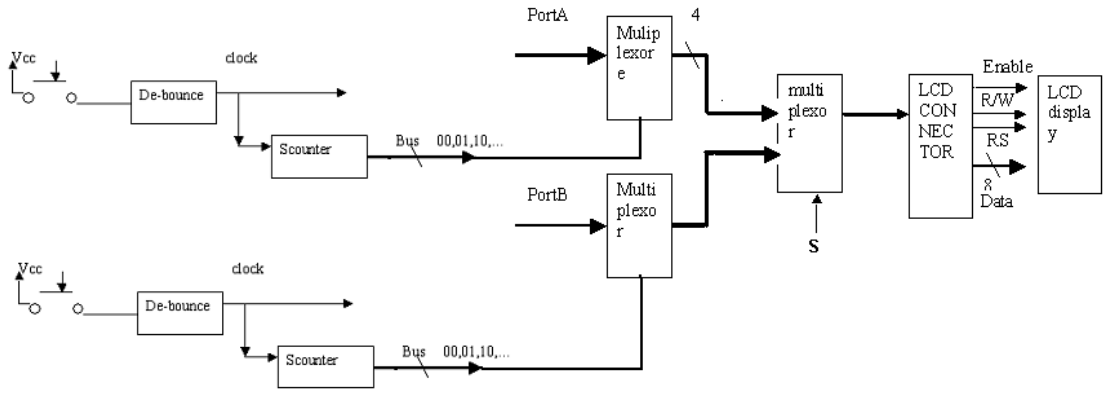
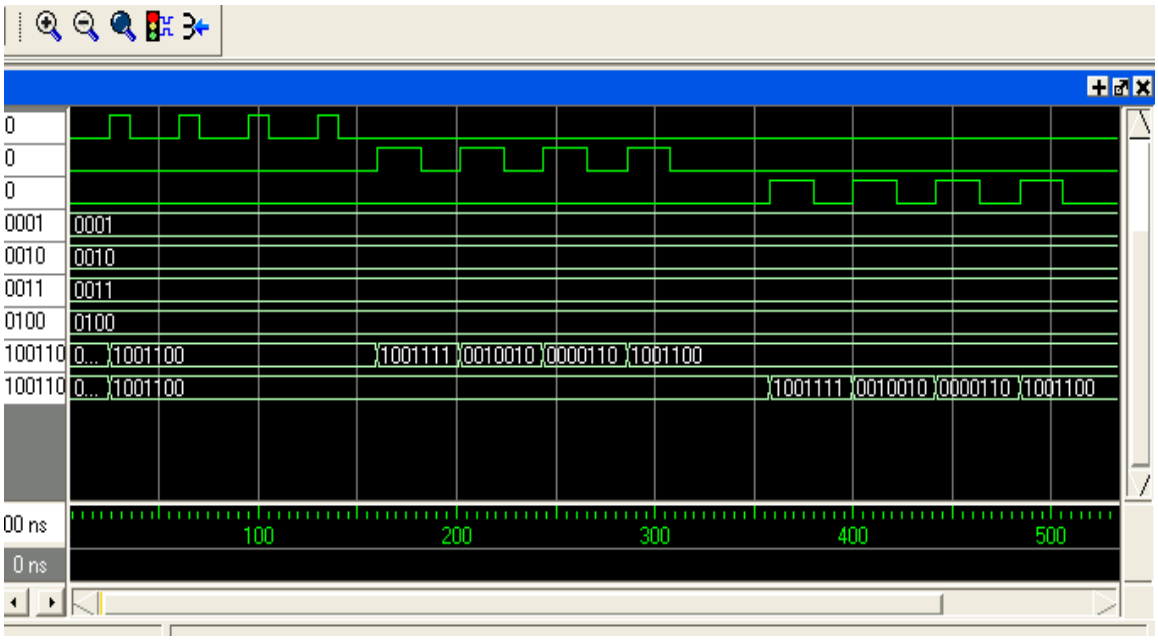


figure 5.

Appendix A:

The output after input preparation and output preparation



Appendix B: register file vhdl code

```
==================registerfile.vhd=================================

library ieee;
use ieee.std_logic_1164.all;
use work.all;

entity registerfile is
 port ( clock4:   in std_logic;
        registerNumber: in std_logic_vector(1 downto 0);
        Read_registerNumber1: in std_logic_vector(1 downto 0);
        Read_registerNumber2: in std_logic_vector(1 downto 0);
        DataIn:   in std_logic_vector(3 downto 0);
        DataOut_A:  out std_logic_vector(3 downto 0);
        DataOut_B:  out std_logic_vector(3 downto 0)
       );
 end registerfile;

architecture struct of registerfile is
    component dff is
     port   ( clock: in std_logic;
              D:     in std_logic;
              Q:     out std_logic
             );
    end component;
    component decoder is
     port ( I: in std_logic_vector(1 downto 0);
            O: out std_logic_vector(3 downto 0)
           );
    end component;
    component AND1 is
     port ( x: in std_logic;
            y: in std_logic;
            F: out std_logic
           );
     end component;

    component multiplexor4 is
     port ( I3: in  std_logic_vector(3 downto 0);
          I2: in  std_logic_vector(3 downto 0);
          I1: in  std_logic_vector(3 downto 0);
          I0: in  std_logic_vector(3 downto 0);
          S:      in  std_logic_vector(1 downto 0);
          O:      out std_logic_vector(3 downto 0)
          );
    end component;


  signal ss1: std_logic;
  signal ss2: std_logic;
  signal ss3: std_logic;
  signal ss4: std_logic;

  signal ss: std_logic_vector(3 downto 0);
```

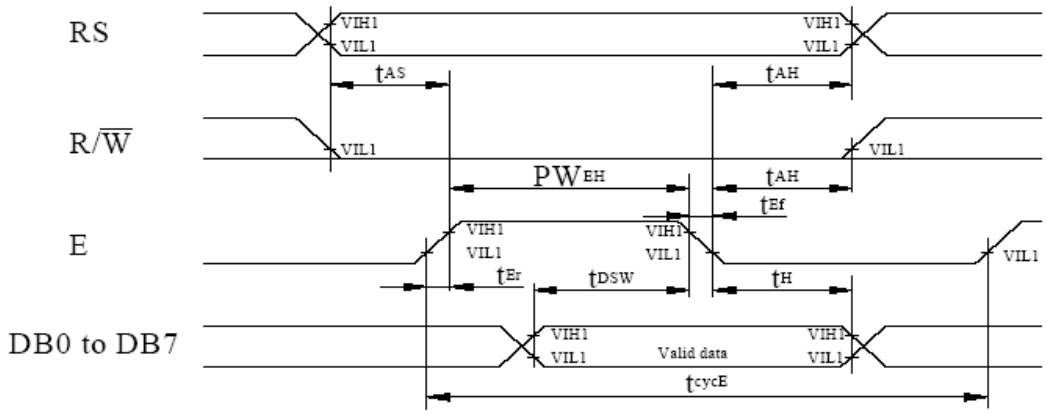
```
  signal DataOut1: std_logic_vector(3 downto 0);
  signal DataOut2: std_logic_vector(3 downto 0);
  signal DataOut3: std_logic_vector(3 downto 0);
  signal DataOut4: std_logic_vector(3 downto 0);


begin
  decoder1: decoder port map ( I(0) => registerNumber(0),
                               I(1) => registerNumber(1),
                               O(0)=>ss1, O(1)=>ss2,O(2)=>ss3,
                               O(3)=>ss4);

  And_1: AND1 port map ( X=>clock4,Y=>ss1,F=>ss(0));
  And_2: AND1 port map ( X=>clock4,Y=>ss2,F=>ss(1));
  And_3: AND1 port map ( X=>clock4,Y=>ss3,F=>ss(2));
  And_4: AND1 port map ( X=>clock4,Y=>ss4,F=>ss(3));


  D1: dff port map
  (clock => ss(0), D =>DataIn(0), Q=>DataOut1(0));
  D2: dff port map
  (clock => ss(0), D=>DataIn(1), Q=>DataOut1(1));
  D3: dff port map
  (clock=>ss(0), D=>DataIn(2), Q=>DataOut1(2));
  D4: dff port map
  (clock=>ss(0), D=>DataIn(3), Q=>DataOut1(3));
  D5: dff port map
  (clock => ss(1), D =>DataIn(0), Q=>DataOut2(0));
  D6: dff port map
  (clock => ss(1), D=>DataIn(1), Q=>DataOut2(1));
  D7: dff port map
  (clock=>ss(1), D=>DataIn(2), Q=>DataOut2(2));
  D8: dff port map
  (clock=>ss(1), D=>DataIn(3), Q=>DataOut2(3));
  D9: dff port map
  (clock => ss(2), D =>DataIn(0), Q=>DataOut3(0));
  D10: dff port map
  (clock => ss(2), D=>DataIn(1), Q=>DataOut3(1));
  D11: dff port map
  (clock=>ss(2), D=>DataIn(2), Q=>DataOut3(2));
  D12: dff port map
  (clock=>ss(2), D=>DataIn(3), Q=>DataOut3(3));
  D13: dff port map
  (clock => ss(3), D =>DataIn(0), Q=>DataOut4(0));
  D14: dff port map
  (clock => ss(3), D=>DataIn(1), Q=>DataOut4(1));
  D15: dff port map
  (clock=>ss(3), D=>DataIn(2), Q=>DataOut4(2));
  D16: dff port map
  (clock=>ss(3), D=>DataIn(3), Q=>DataOut4(3));

 mux1: multiplexor4 port map
     (S(0)=>Read_registerNumber1(0), S(1)=>Read_registerNumber1(1),
     I0(0)=>
DataOut1(0),I0(1)=>DataOut1(1),I0(2)=>DataOut1(2),I0(3)=>DataOut1(3),
     I1(0)=>DataOut2(0),I1(1)=>DataOut2(1),I1(2)=>DataOut2(2),
```

```
     I1(3)=>DataOut2(3),
I2(0)=>DataOut3(0),I2(1)=>DataOut3(1),I2(2)=>DataOut3(2),
     I2(3)=>DataOut3(3),
     I3(0)=>DataOut4(0),I3(1)=>DataOut4(1),I3(2)=>DataOut4(2),
       I3(3)=>DataOut4(3),
     O(0)=>DataOut_A(0),O(1)=>DataOut_A(1), O(2)=>DataOut_A(2),
      O(3)=>DataOut_A(3));

  mux2: multiplexor4 port map
     (S(0)=>Read_registerNumber2(0), S(1)=>Read_registerNumber2(1),
     I0(0)=>
DataOut1(0),I0(1)=>DataOut1(1),I0(2)=>DataOut1(2),I0(3)=>DataOut1(3),
     I1(0)=>DataOut2(0),I1(1)=>DataOut2(1),I1(2)=>DataOut2(2),
       I1(3)=>DataOut2(3),
     I2(0)=>DataOut3(0),I2(1)=>DataOut3(1),I2(2)=>DataOut3(2),
      I2(3)=>DataOut3(3),
     I3(0)=>DataOut4(0),I3(1)=>DataOut4(1),I3(2)=>DataOut4(2),
        I3(3)=>DataOut4(3),
O(0)=>DataOut_B(0),O(1)=>DataOut_B(1), O(2)=>DataOut_B(2),
O(3)=>DataOut_B(3));


end struct;
```

Appendix C: time requirement for writing and reading LCD

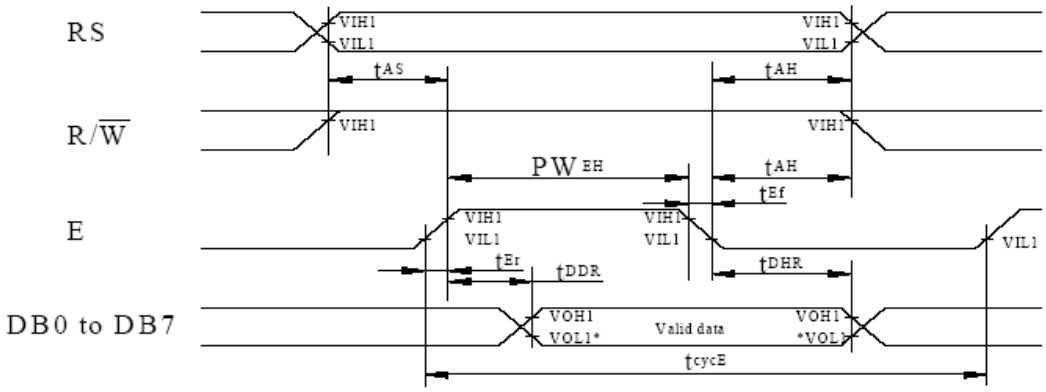
13. Timing Characteristics

13.1 Write Operation



Item	Symbol	Min	Typ	Max	Unit
Enable cycle time	t_{cycE}	500	–	–	ns
Enable pulse width (high level)	PW_{EH}	230	–	–	ns
Enable rise/fall time	t_{Er},t_{Ef}	–	–	20	ns
Address set-up time (RS, R/W to E)	t_{AS}	40	–	–	ns
Address hold time	t_{AH}	10	–	–	ns
Data set-up time	t_{DSW}	80	–	–	ns
Data hold time	t_H	10	–	–	ns

13.2 Read Operation



NOTE: *VOL1 is assumed to be 0.8V at 2 MHZ operation.

ITEM	Symbol	Min	Typ	Max	Unit
Enable cycle time	t_{cycE}	500	–	–	ns
Enable pulse width (high level)	PW_{EH}	230	–	–	ns
Enable rise/fall time	t_{Er},t_{Ef}	–	–	20	ns
Address set-up time (RS, R/W to E)	t_{AS}	40	–	–	ns
Address hold time	t_{AH}	10	–	–	ns
Data delay time	t_{DDR}	–	–	160	ns
Data hold time	t_{DHR}	5	–	–	ns