Bradley Barrett (btb8db)

ECE 4435
4/8/2016

Lab 10 Report

The purpose of the memory stage is to receive information from the execute stage and interface with the system memory accordingly. Following the Memory stage, the Writeback stage then writes any necessary values back to the system Register File. The Register File itself has is capable of 2 simultaneous reads and 1 write. Provided the system memory may not be able to perform the requested read or write in one clock cycle, it may send a memory delay to the Memory Stage. When this happens, the memory stage must send a stall back to the earlier stages in the pipeline if it is not currently performing a NOP.

Testing:

Testing was conducted in 3 stages: 1) test Memory stage, 2) test Writeback stage, and 3) test Register File.

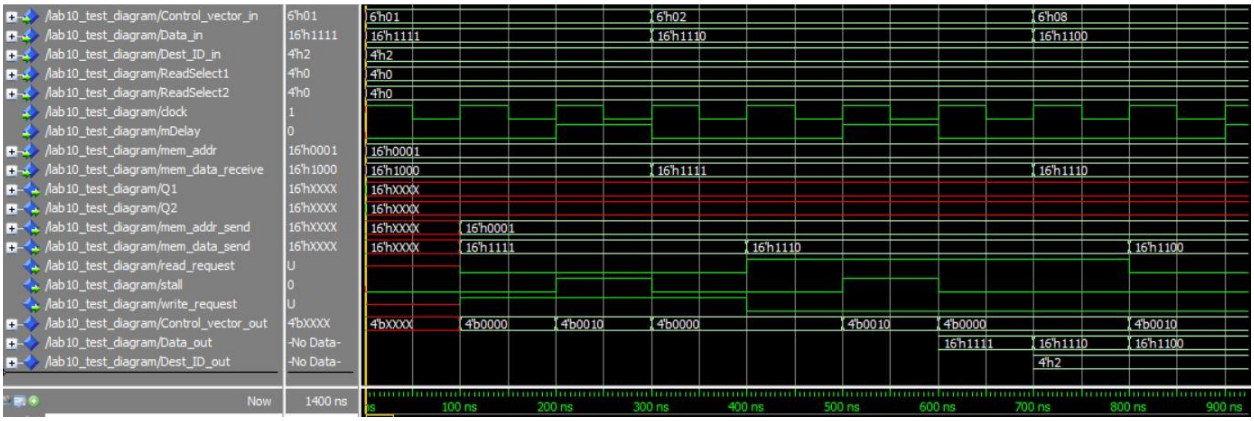
Test Memory Stage:

The memory stage can receive 4 different types of instructions: 1) ST – write a value to memory, 2) LD – retrieve a value from memory and send on that value, 3) NOP – send on a NOP and ignore memory delay, and 4) any other instruction not requiring memory usage – ignore memory delay and send on data received from execute. These four cases were all tested in the presence and absence of a memory delay. The results and expected inputs and outputs are shown in the table and Modelsim simulation below. All tests passed.

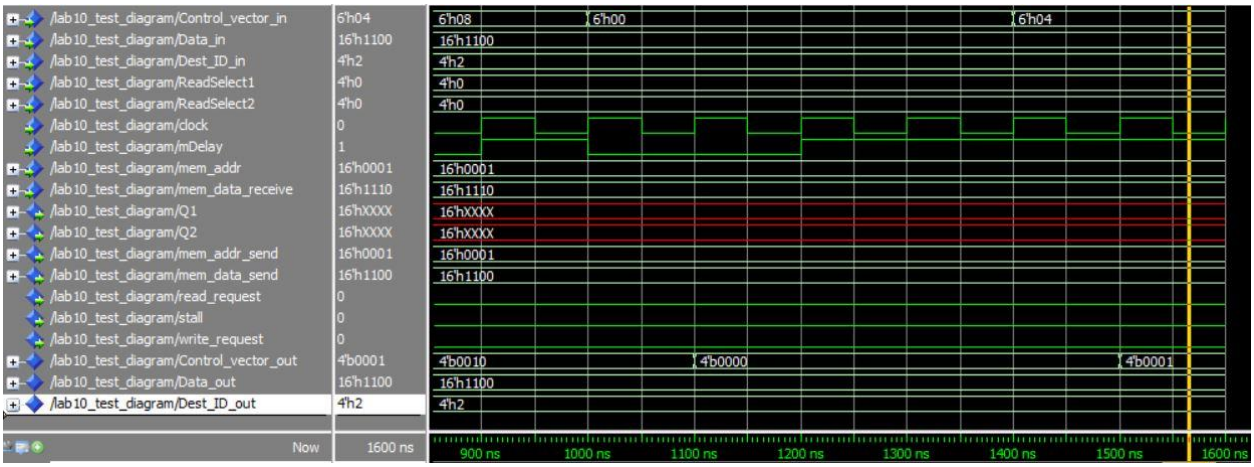
Simulation Inputs and Expected Outputs:

Start time (ns)	End time (ns)	Instr. Type	INPUT								OUTPUT									
			CV_in	Data_in	Dest_ID_in	mem_addr	mem_data_received	mDelay	ReadSelect1	ReadSelect2	mem_addr_send	mem_data_send	read_request	write_request	stall	Q1	Q2	CV_out	Data_out	Dest_ID_out
0	200	ST	000001	0x1111	0x2	0x0001	0x1000	0	0x0000	0x0000	0x0001	0x1111	0	1	0	0xXXXX	0xXXXX	0000	0x1111	0x2
200	300	ST	000001	0x1111	0x2	0x0001	0x1000	1	0x0000	0x0000	0x0001	0x1111	0	1	1	0xXXXX	0xXXXX	0000	0x1111	0x2
300	500	LD	000010	0x1110	0x2	0x0001	0x1111	0	0x0000	0x0000	0x0001	0x1110	1	0	0	0xXXXX	0xXXXX	0000	0x1111	0x2
500	600	LD	000010	0x1110	0x2	0x0001	0x1111	1	0x0000	0x0000	0x0001	0x1110	1	0	1	0xXXXX	0xXXXX	0000	0x1111	0x2
600	700	ST	000001	0x1111	0x2	0x0001	0x1000	0	0x0000	0x0000	0x0001	0x1111	0	1	0	0xXXXX	0xXXXX	0000	0x1111	0x2
700	900	NOP	001000	0x1100	0x2	0x0001	0x1110	0	0x0000	0x0000	don't care	don't care	0	0	0	0xXXXX	0xXXXX	0010	0x1110	0x2
900	1000	NOP	001000	0x1100	0x2	0x0001	0x1110	1	0x0000	0x0000	don't care	don't care	0	0	0	0xXXXX	0xXXXX	0010	0x1110	0x2
1000	1200	Other non-mem	000000	0x1100	0x2	0x0001	0x1110	0	0x0000	0x0000	don't care	don't care	0	0	0	0xXXXX	0xXXXX	0001	0x1100	0x2
1200	1600	Other non-mem	000000	0x1100	0x2	0x0001	0x1110	1	0x0000	0x0000	don't care	don't care	0	0	0	0xXXXX	0xXXXX	0001	0x1100	0x2

Simulation Results:



Simulation Results: (continued)



Test Writeback Stage:

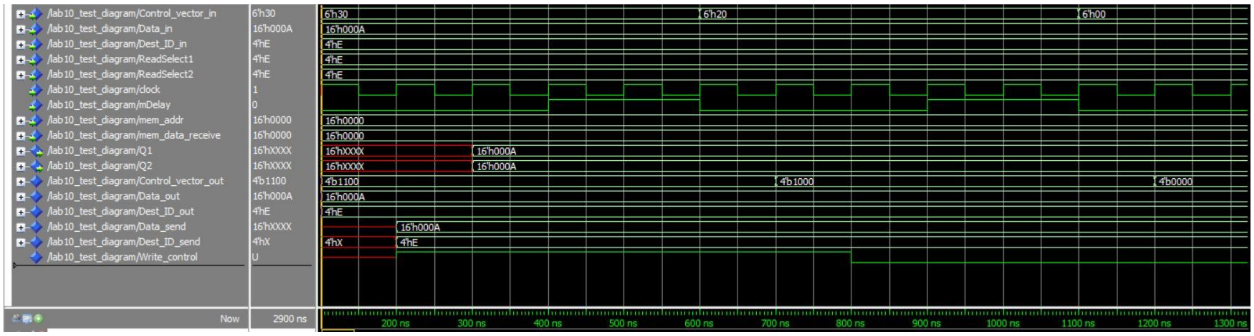
Provided that the Writeback stage passes the Dest_ID and Data values received from the Memory directly on to the Register File for all instructions, the only output that must be closely tested is the Write_control output. The Write_control output is determined directly from the bits of the input control vector for the stage. Thus, by testing all possible input control vector, all output cases can be evaluated. Given my selection for the control vector, the following cases were tested (all tests passed):

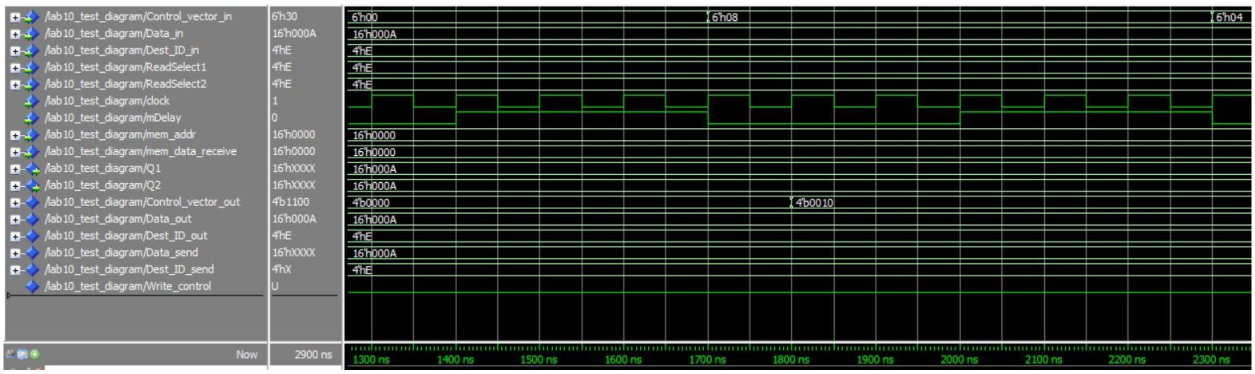
Instr. Type	Control Vector Bits				Write Control
	Jump	IR[8]	nop	RegWrite	
JAL	1	1	0	0	1
JMP	1	0	0	0	0
NOP	0	0	1	0	0
other no Reg Write	0	0	0	0	0
other requires Reg Write	0	0	0	1	1

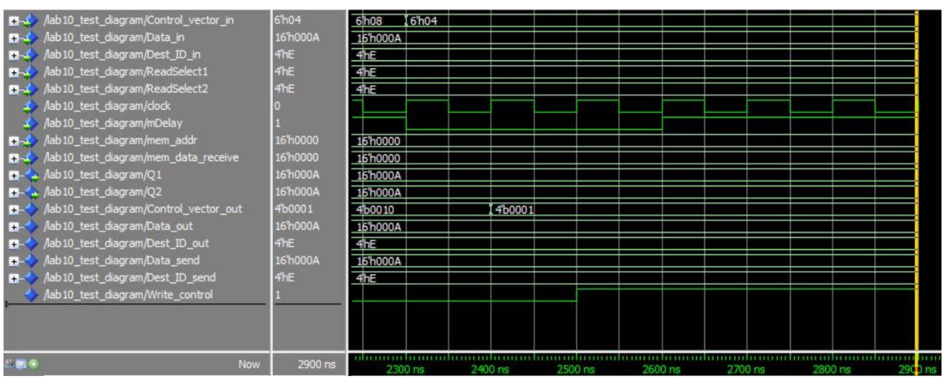
Simulation Inputs and Expected Outputs:

Start time (ns)	End time (ns)	Instr. Type	INPUT						Output					
			CV_in	Data_in	Dest_ID_in	mem_addr	mem_data_received	mDelay	Q1	Q2	CV_out	Data_send	Dest_ID_send	Write_control
0	400	JAL	110000	0x000A	0xE	0x0001	0x1110	0	0x000A	0x000A	1100	0x000A	0xE	1
400	600	JAL	110000	0x000A	0xE	0x0001	0x1110	1	0x000A	0x000A	1100	0x000A	0xE	1
600	900	JMP	100000	0x000A	0xE	0x0001	0x1110	0	0x000A	0x000A	0000	0x000A	0xE	1
900	1100	JMP	100000	0x000A	0xE	0x0001	0x1110	1	0x000A	0x000A	0000	0x000A	0xE	1
1100	1400	other no Reg Write	000000	0x000A	0xE	0x0001	0x1110	0	0x000A	0x000A	1000	0x000A	0xE	0
1400	1700	other no Reg Write	000000	0x000A	0xE	0x0001	0x1110	1	0x000A	0x000A	1000	0x000A	0xE	0
1700	2000	NOP	001000	0x000A	0xE	0x0001	0x1110	0	0x000A	0x000A	0010	0x000A	0xE	0
2000	2300	NOP	001000	0x000A	0xE	0x0001	0x1110	1	0x000A	0x000A	0010	0x000A	0xE	0
2300	2600	other requires Reg Write	000100	0x000A	0xE	0x0001	0x1110	0	0x000A	0x000A	0001	0x000A	0xE	1
2600	2900	other requires Reg Write	000100	0x000A	0xE	0x0001	0x1110	1	0x000A	0x000A	0001	0x000A	0xE	1

Simulation Results:







Test Register File:

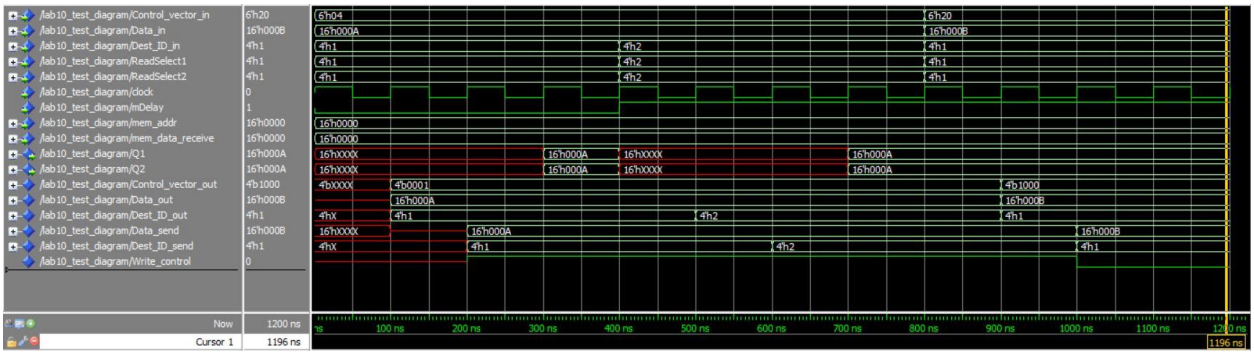
The Register File should be able to perform 2 simultaneous reads and 1 write when enabled of selected registers. When not enabled, any attempt to write a register should not change the value stored. As a result, a read on the register before after the write attempt when disabled should produce the same value. The following tests were performed to verify this behavior (all tests passed):

1) Write value to register 1 and read that register on read line 1 and 2.
2) Write value to register 2 and read that register on read line 1 and 2.
3) Try to write to register 1 with the load disabled and read that register on read line 1 and 2.

Simulation Inputs and Expected Outputs:

Start time (ns)	End time (ns)	Test Number	INPUT							Output	
			CV_in	Data_in	Dest_ID_in	mem_addr	mem_data_received	mDelay	ReadSelect1	ReadSelect2	Q1	Q2
0	400	1	000100	0x000A	0x1	0x0000	0x0000	0	0x1	0x1	0x000A	0x000A
400	800	2	000100	0x000A	0x2	0x0000	0x0000	1	0x2	0x2	0x000A	0x000A
800	1200	3	100000	0x000A	0x1	0x0000	0x0000	1	0x1	0x1	0x000A	0x000A

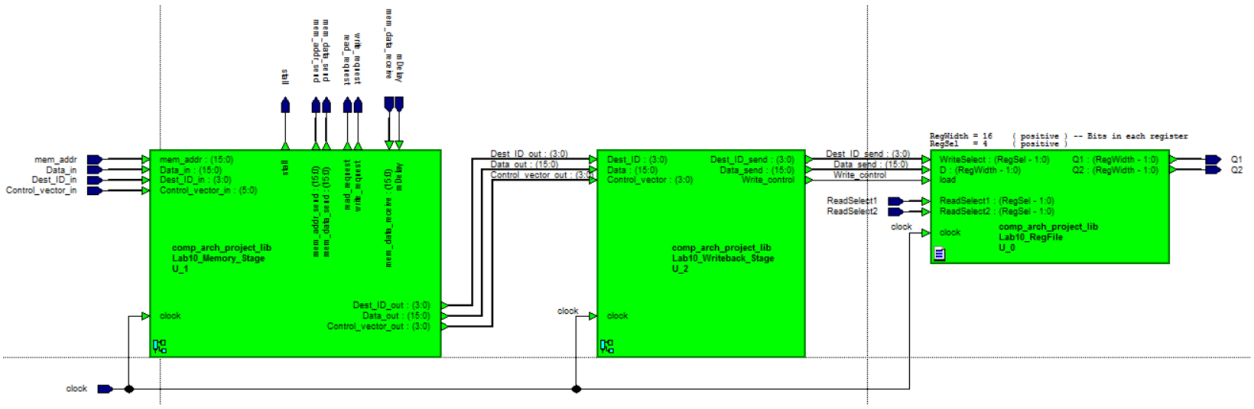
Simulation Results:



Block Diagrams and Code:

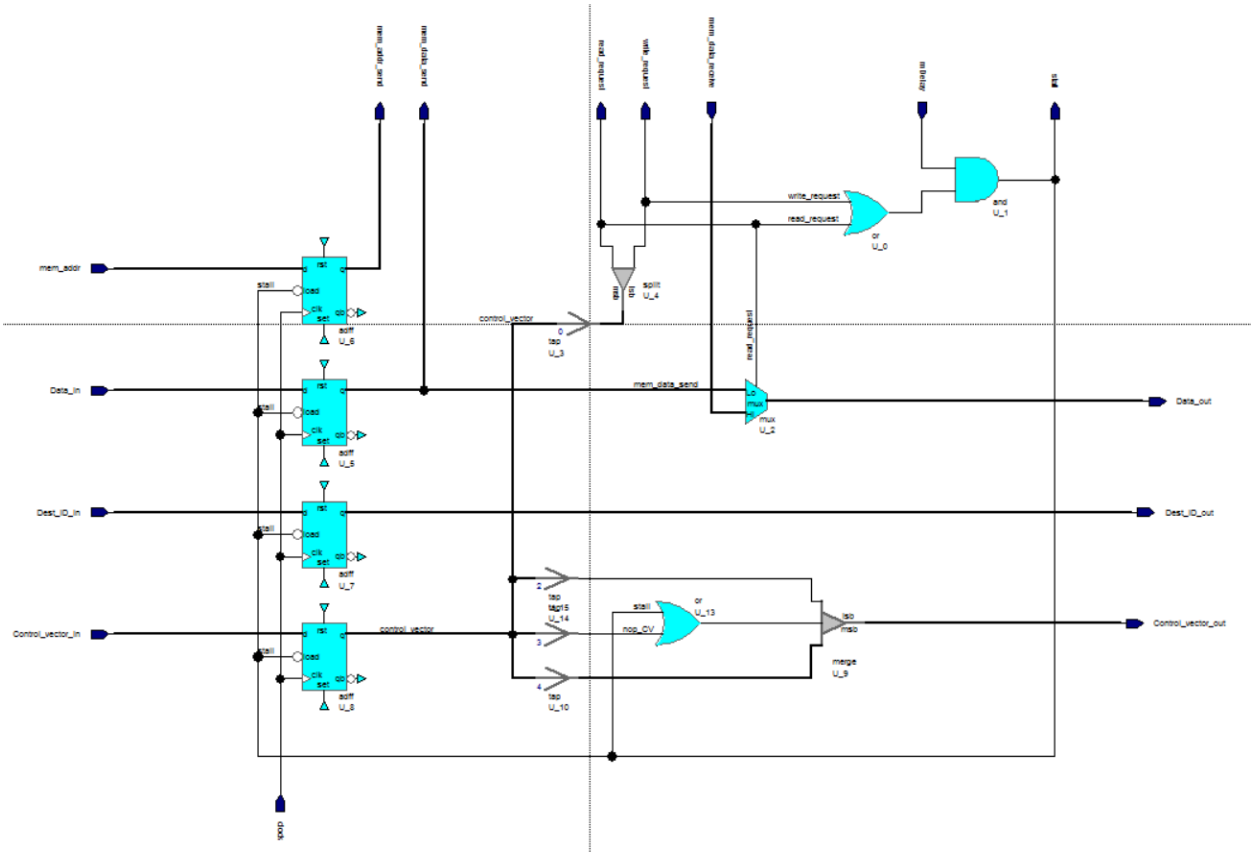
Lab10_Test_Diagram

Block Diagram:



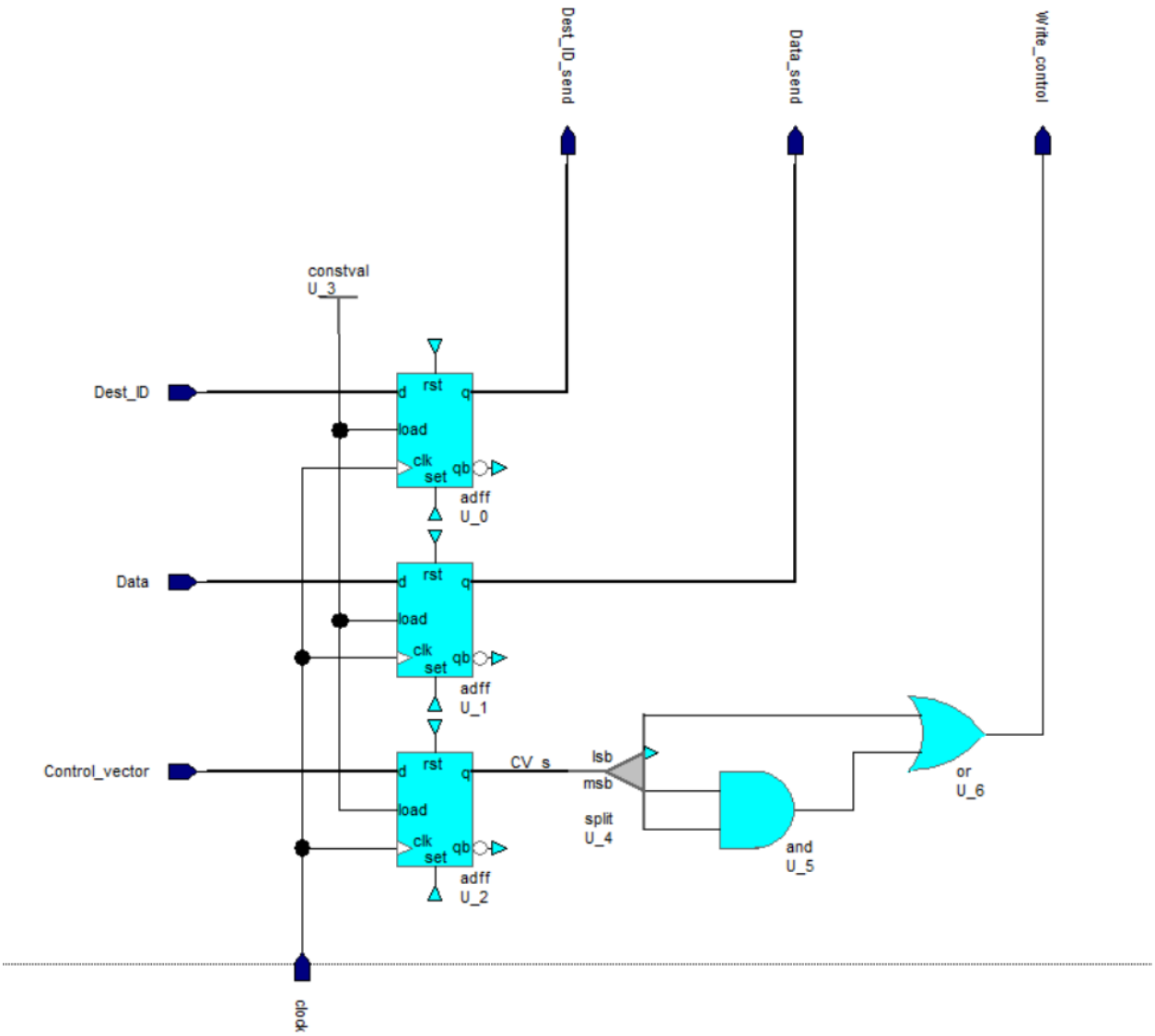
Lab10_Memory_Stage

Block Diagram:



Lab10_Writeback_Stage

Block Diagram:



Lab10_RegFile

Code:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

ENTITY Lab10_RegFile IS
  GENERIC(  RegWidth : positive := 16; -- Bits in each register
            RegSel : positive := 4); -- The bits required to select a
register
  PORT (  D : IN std_logic_vector(RegWidth - 1 DOWNTO 0); -- Data Input
          Q1, Q2 : OUT std_logic_vector(RegWidth - 1 DOWNTO 0); -- Data
Output: Q1 = Read 1, Q2 = Read 2
          ReadSelect1, ReadSelect2, WriteSelect : IN std_logic_vector(RegSel
- 1 DOWNTO 0);
          clock, load : IN std_logic);
END ENTITY Lab10_RegFile;

--
ARCHITECTURE Struct OF Lab10_RegFile IS
  -- wires
  -- connect Read1, Read2, and Write decoders to ER1, ER2, and EW enables of
RegRead2Write1
  SIGNAL ReadDecode1, ReadDecode2, WriteDecode : std_logic_vector(2**RegSel-1
DOWNTO 0);
BEGIN
  ReadDecoder1 :  ENTITY work.Lab10_Decoder(Behavior)
                  GENERIC MAP( controls => RegSel )
                  PORT MAP( sel=>ReadSelect1, onehot=>ReadDecode1,
enable=>'1' ); --read is always enabled

  ReadDecoder2 :  ENTITY work.Lab10_Decoder(Behavior)
                  GENERIC MAP( controls => RegSel )
                  PORT MAP( sel=>ReadSelect2, onehot=>ReadDecode2,
enable=>'1' ); --read is always enabled

  WriteDecoder :  ENTITY work.Lab10_Decoder(Behavior)
                  GENERIC MAP( controls => RegSel )
                  PORT MAP( sel=>WriteSelect, onehot=>WriteDecode,
enable=>load );

  -- Array of RegRead2Write1
  RegArray : FOR i IN 0 TO (2**RegSel-1) GENERATE
  BEGIN
    Regi : ENTITY work.Lab10_RegRead2Write1(Struct)
           GENERIC MAP( size => RegWidth )
           PORT MAP( D=>D, Q1=>Q1, Q2=>Q2, EW=>WriteDecode(i),
ER1=>ReadDecode1(i), ER2=>ReadDecode2(i), CLK=>clock );
  END GENERATE RegArray;
END ARCHITECTURE Struct;
```

Lab10_Decoder

Code:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

ENTITY Lab10_Decoder IS
  GENERIC( controls : positive := 4);
  PORT( sel : IN std_logic_vector(controls-1 DOWNTO 0);
        onehot : OUT std_logic_vector((2**controls)-1 DOWNTO 0);
        enable : IN std_logic);
END ENTITY Lab10_Decoder;

--
ARCHITECTURE Behavior OF Lab10_Decoder IS
BEGIN
  PROCESS(sel, enable)
    VARIABLE selection : natural;
    VARIABLE result : std_logic_vector((2**controls)-1 DOWNTO 0);
    CONSTANT zero : std_logic_vector((2**controls)-1 DOWNTO 0) := (others =>
'0');
  BEGIN
    result := zero;
    IF(enable = '1') THEN
      selection := To_Integer(unsigned(sel));
      result(selection) := '1';
    END IF;
    onehot <= result;
  END PROCESS;
END ARCHITECTURE Behavior;
```

Lab10_ RegRead2Write1

Code:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

ENTITY Lab10_RegRead2Write1 IS
  GENERIC( size : positive := 16 );
  PORT( D : IN std_logic_vector(size-1 DOWNTO 0); -- Data outputs for Read 1
and Read 2
        Q1, Q2 : OUT std_logic_vector(size-1 DOWNTO 0);
        EW, ER1, ER2, CLK : IN std_logic); --  EW = Enable Write, ER1 =
Enable Read 1, ER2 = Enable Read 2
END ENTITY Lab10_RegRead2Write1;

--
ARCHITECTURE Struct OF Lab10_RegRead2Write1 IS
  -- wires
  SIGNAL reg_out_buff_in : std_logic_vector(size-1 DOWNTO 0); -- conection of
Reg output to TriStateBuffer input

BEGIN
  Reg : ENTITY work.Lab10_Reg(Behavior)
        PORT MAP( D=>D, Q=>reg_out_buff_in, CLK=>CLK, E=>EW );

  TriStateBuff1 : ENTITY work.Lab10_TriStateBuffer(Behavior)
                  PORT MAP( A=>reg_out_buff_in, Y=>Q1, TC=>ER1 );

  TriStateBuff2 : ENTITY work.Lab10_TriStateBuffer(Behavior)
                  PORT MAP( A=>reg_out_buff_in, Y=>Q2, TC=>ER2 );

END ARCHITECTURE Struct;
```

Lab10_ Reg

Code:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

ENTITY Lab10_Reg IS
  GENERIC( size : positive := 16);
  PORT( D : IN std_logic_vector(size-1 DOWNTO 0);
        Q : OUT std_logic_vector(size-1 DOWNTO 0);
        CLK, E: IN std_logic);
END ENTITY Lab10_Reg;

--
ARCHITECTURE Behavior OF Lab10_Reg IS
BEGIN
  PROCESS( CLK )
  BEGIN
    IF( E = '1' and rising_edge(CLK) ) THEN
      Q <= D;
    END IF;
  END PROCESS;
END ARCHITECTURE Behavior;
```

Lab10_ TriStateBuffer

Code:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

ENTITY Lab10_TriStateBuffer IS
  GENERIC( size : positive := 16 );
  PORT( A : IN std_logic_vector(size-1 DOWNTO 0);
        Y : OUT std_logic_vector(size-1 DOWNTO 0);
        TC : IN std_logic); -- tri-state control, active high
END ENTITY Lab10_TriStateBuffer;

--
ARCHITECTURE Behavior OF Lab10_TriStateBuffer IS
BEGIN
  PROCESS(A, TC)
  BEGIN
    IF( TC = '1' ) THEN
      -- assign output, Y, to input, A
      Y <= A;
    ELSE
      -- assign output, Y, to high impedance
      FOR i IN 0 TO size-1 LOOP
        Y(i) <= 'Z';
      END LOOP;
    END IF;
  END PROCESS;
END ARCHITECTURE Behavior;
```