Lab #5 Design of a Register File and Datapath

Objectives:

- to gain insight into the internal logic for data movement between registers in a computer
- to introduce the concept of a control word for implementing an elementary set of micro-operations.
- to gain experience with the use of VHDL constructs
- to gain experience with incremental, modular design

Preparation:

Read the following experiment and study the circuits as shown. Study the ALU (example 6.24, p.360) and D flipflop (examples 7.2 and 7.3, p423) described in the textbook. Familiarize yourself with the VHDL code at the end of the lab that contains examples of the VHDL constructs: "SELECT", "PROCESS", "WHEN-ELSE" and "CASE". There are also several examples of the use of "STD_LOGIC_VECTOR" for defining a multi-bit input or output. Bring your textbook to the lab!

Devices used:

MAX7000 EPM7128SLC84-7 CPLD

Introduction:

The internal registers of a microprocessor characterize its architecture. For example, a 32-bit microprocessor has (mostly) 32-bit registers internally. Moving data among these registers is the single most frequent operation that takes place in a computer. In this session, we will construct a 4x4 "register file" comprising registers $\mathbf{R_0}$, $\mathbf{R_1}$, $\mathbf{R_2}$ **and** $\mathbf{R_3}$ (as shown in the figure of *Step 4*) to demonstrate the concept of register transfer logic. We will also implement a simple arithmetic and logic unit (ALU) and then combine the ALU and the register file to construct a simple computer datapath.

Moving data from one register to another may be more accurately described as a "copy" operation. The destination register takes on the value of the source register which itself remains unchanged after the operation. The source and destination registers may be the same register. These register transfers are designated using register transfer notation. For example, copying the contents of a register $\mathbf{R_s}$
into another register $\mathbf{R_d}$ would be written as:

$$\mathbf{R_d} \leftarrow \mathbf{R_s}$$

where $\mathbf{R_s}$ is the source register (and remains unchanged) and $\mathbf{R_d}$ is the destination register.

A register file comprises a decoder which chooses a destination register and a multiplexer to direct the outputs of any register through to the data output lines. The decoder select lines may then be viewed as the destination "address" and the multiplexer select lines as the source "address".

Experiment:

This datapath circuit to be built requires several components that we will design and implement and test individually. To facilitate successful implementation, verification and documentation of complex designs, one should proceed in an ***incremental, modular*** fashion whereby each component of a circuit is built and verified independently. The components are then put together and may form another, larger component at the next level of the design hierarchy. These combined components may then be combined to from even larger components and so on. This continues to the top level of the design. This practice applies to the design of hardware, software or any other system for that matter! Our final goal here is to design and implement the logic for a hardware datapath that contains a simple arithmetic and logic unit (ALU) that can perform low level processing.

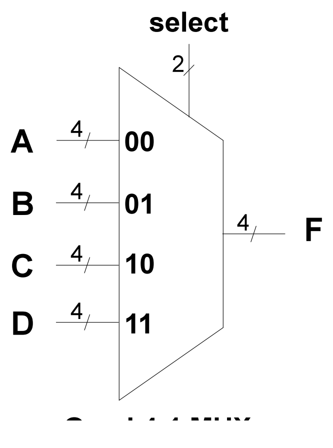
Proceed methodically and slowly and be mindful of bit ordering for the 4-bit bus connections. Use neatly arranged, colour-coded wiring to aid in debugging and masking tape to affix temporary labels for the toggle switches on the breadboards. We start by designing the register file.

Step 1 Decoder

The register file requires a 2-line to 4-line decoder with HI-true outputs and one HI-true enable input as shown in the circuit of *Step 4*. This is similar to the decoder you designed in a previous lab. Implement this component using the graphic design editor and test it in the MAX7000 device. Generate a symbol for this decoder which you will use later. See appendix B.5, page 851.

Step 2 Quad 4:1 MUX

The register file also requires a Quad 4:1 multiplexer. A Quad 4:1 MUX has four 4-bit data



inputs, a 4-bit data output and two select lines as shown below. Study the VHDL source code given at the end of this lab that implements a Quad 4:1 multiplexer. Be sure you understand the logic of the VHDL code. Compile this program, implement and test using the MAX7000 device. Generate a symbol for this MUX which you will use later.

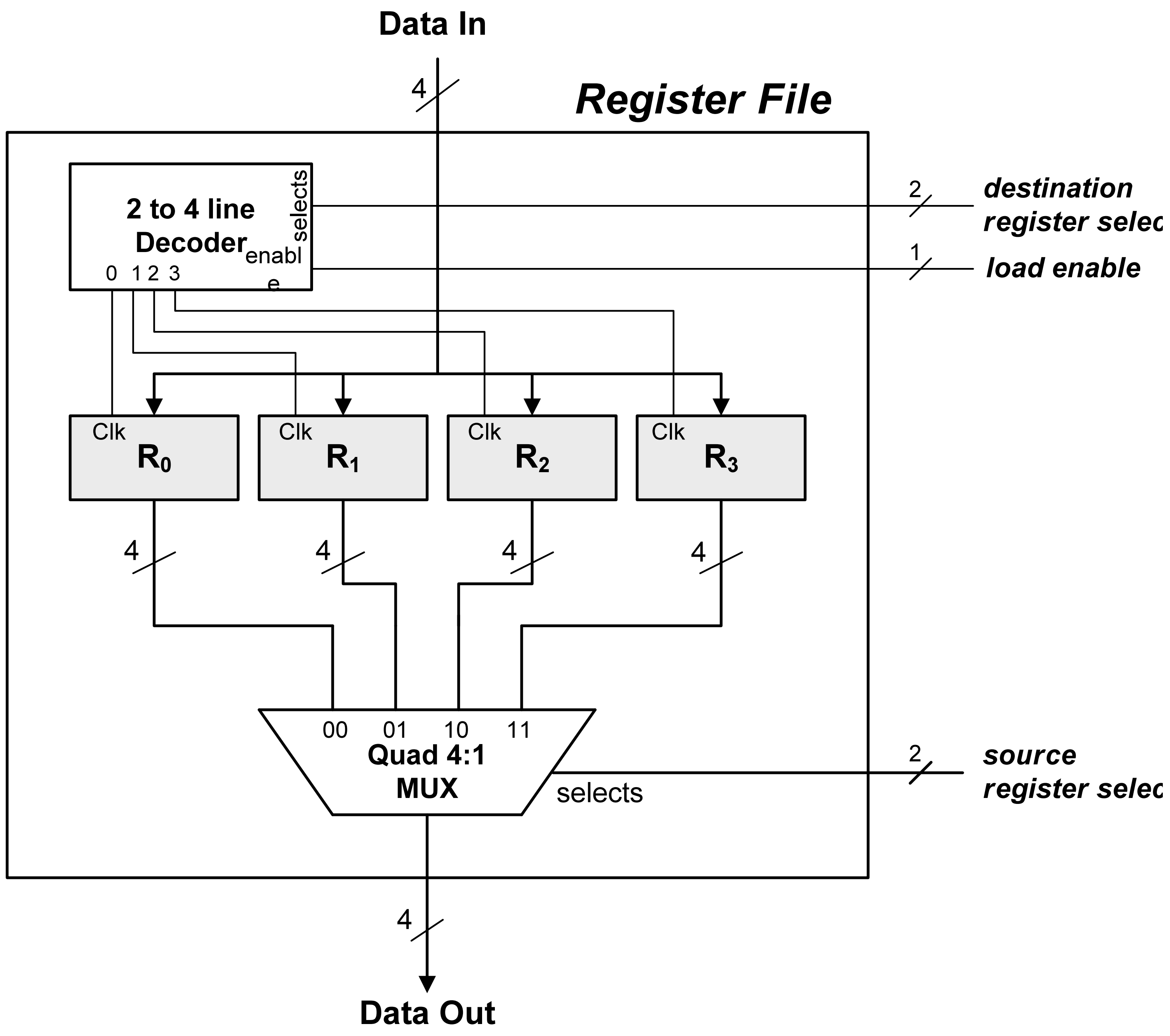
What would an Octal 4:1 MUX look like? a Quad 2:1 MUX ? Draw a block diagram.

Step 3 Registers

The four registers **R_0, R_1, R_2 and R_3** in the diagram below are to be implemented using the VHDL code at the end of this lab (similar to Figure 7.46, p. 429 of the textbook). Each register comprises 4 positive edge-triggered D flipflops. Each register has a 4-bit input data and a 4-bit output data. The clock input to all flipflops in the register is defined as **Clk**. Compile this code and make a symbol for the register.

Step 4 Register File

Now we will design the register file using the *graphic design editor* by connecting the multiplexer, decoder and four registers as shown below. Compile and test the register file circuit in the MAX7000 chip to ensure that all four registers can be loaded using toggle switches on the **Data In** lines, and read using LEDs connected at the **Data Out** lines. Be sure that you understand the timing of the "*load enable*" input relative to all the other inputs and outputs.



Make a symbol for this circuit for use later.

Show this working circuit to a Lab Demonstrator before proceeding.

Step 5 Datapath

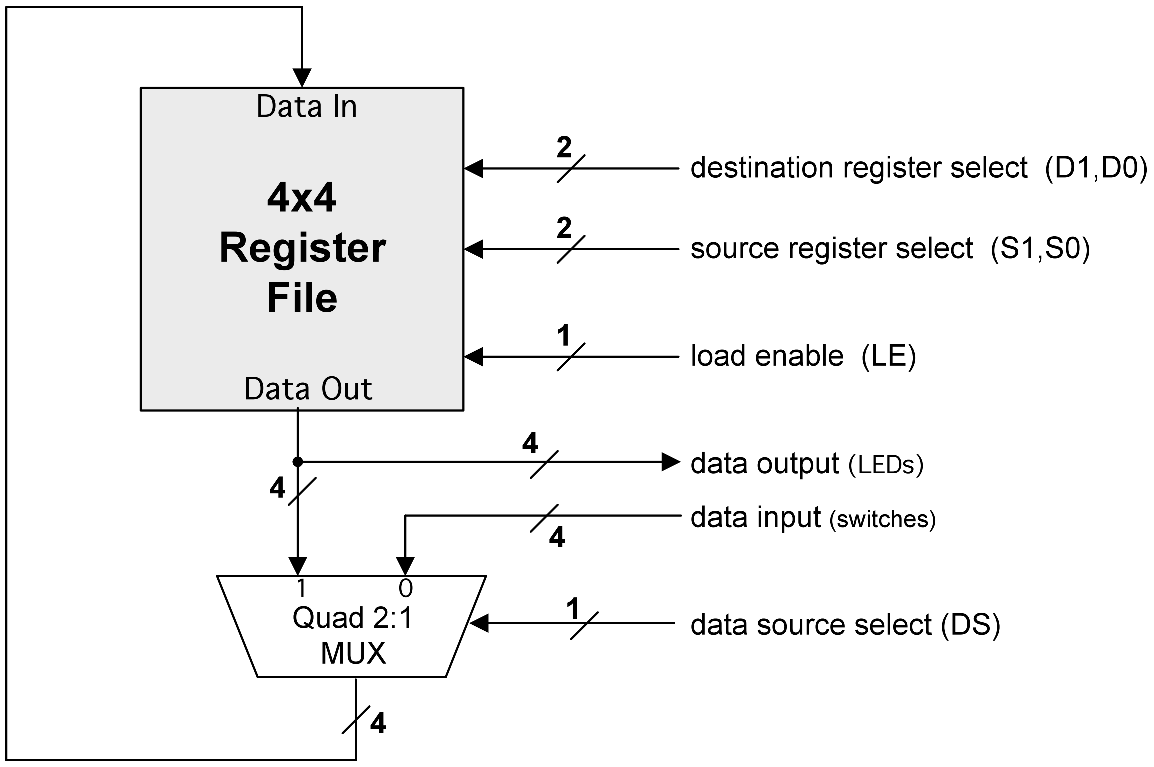
The register file forms the basis of a "datapath" which is a fundamental building block of a computer. See the diagram below. Data is selected from any register then stored back into any other register in the register file, all in a single clock cycle (a lo-hi-lo pulse applied to the load enable LE input). A Quad 2:1 MUX included as shown below allows external data to be inserted into the datapath. Data can thus be transferred between any two registers of our register file *or* any register can be loaded with external data. This datapath can execute the following operations:

(a) any register can be loaded with external data from switches **R_d ← data** (4-bits) (where d=0,1,2 or 3)

(b) any register can be loaded with the data contained in any one of the other registers, including itself (register-to-register transfer) **R_d ← R_s** (where d, s = 0, 1, 2 or 3)

The implementation is shown below. The inputs [D1, D0, S1, S0, DS] form a 5-bit "control" word which specifies the source (S1, S0) and destination (D1, D0) registers of the register file and an operation (DS) that is to take place. For DS=0, external data from switches is loaded into the destination register; for DS=1, data is transferred from the source register to the destination register. Once the control word and data input (if appropriate) are set on the level switches, execution is achieved by applying a load enable (LE) input to the register file. This LE input may be considered as the clock to the entire system. You can view the results of each operation using four LEDs connected to the output of the register file as shown.

Design this data path using the graphic design editor. VHDL code for the Quad 2:1 MUX design is given at the end of this lab. Test the circuit for various combinations of the register transfers summarized in the following table.



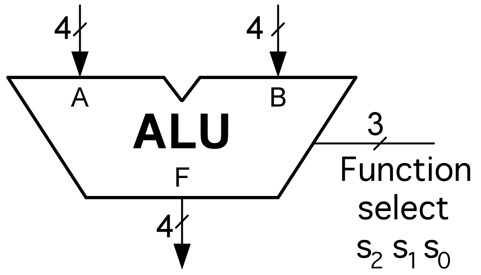
Summary of register transfer operations

Data Sourc e DS	Source Register S1 S0	Destination Register D1 D0	data input	Register Transfer Operation
0	X X	0 0	a b c d	R0 ←abcd
0	X X	0 1	a b c d	R1 ← abcd
0	X X	1 0	a b c d	R2 ← abcd
0	X X	1 1	a b c d	R3 ← abcd
1	0 0	0 0	XXXX	R0 ← R0
1	0 0	0 1	XXXX	R1 ← R0
1	0 0	1 0	XXXX	R2 ← R0
1	0 0	1 1	XXXX	R3 ← R0
1	0 1	0 0	XXXX	R0 ← R1
1	0 1	0 1	XXXX	R1 ← R1
1	0 1	1 0	XXXX	R2 ← R1
1	0 1	1 1	XXXX	R3 ← R1
1	1 0	0 0	XXXX	R0 ← R2
1	1 0	0 1	XXXX	R1 ← R2
1	1 0	1 0	XXXX	R2 ← R2
1	1 0	1 1	XXXX	R3 ← R2
1	1 1	0 0	XXXX	R0 ← R3
1	1 1	0 1	XXXX	R1 ← R3
1	1 1	1 0	XXXX	R2 ← R3
1	1 1	1 1	XXXX	R3 ← R3

Step 6 ALU

An ALU is a combinational logic circuit that performs various arithmetic and logic operations on *n*-bit data (operands). A simple 8-function ALU that operates on 4-bit inputs A and B is specified in the following table. The block symbol for an ALU is also given below. The number of bits on the "function select" input determines how many operations may be performed on the operands (in this example there are $2^3 = 8$ functions). Its definition in VHDL code is given at the end of this lab (see Figure 6.48, p.360 of the textbook). Compile this code, then implement and test this function in the MAX7000 device using several values of A and B. Verify the function table:

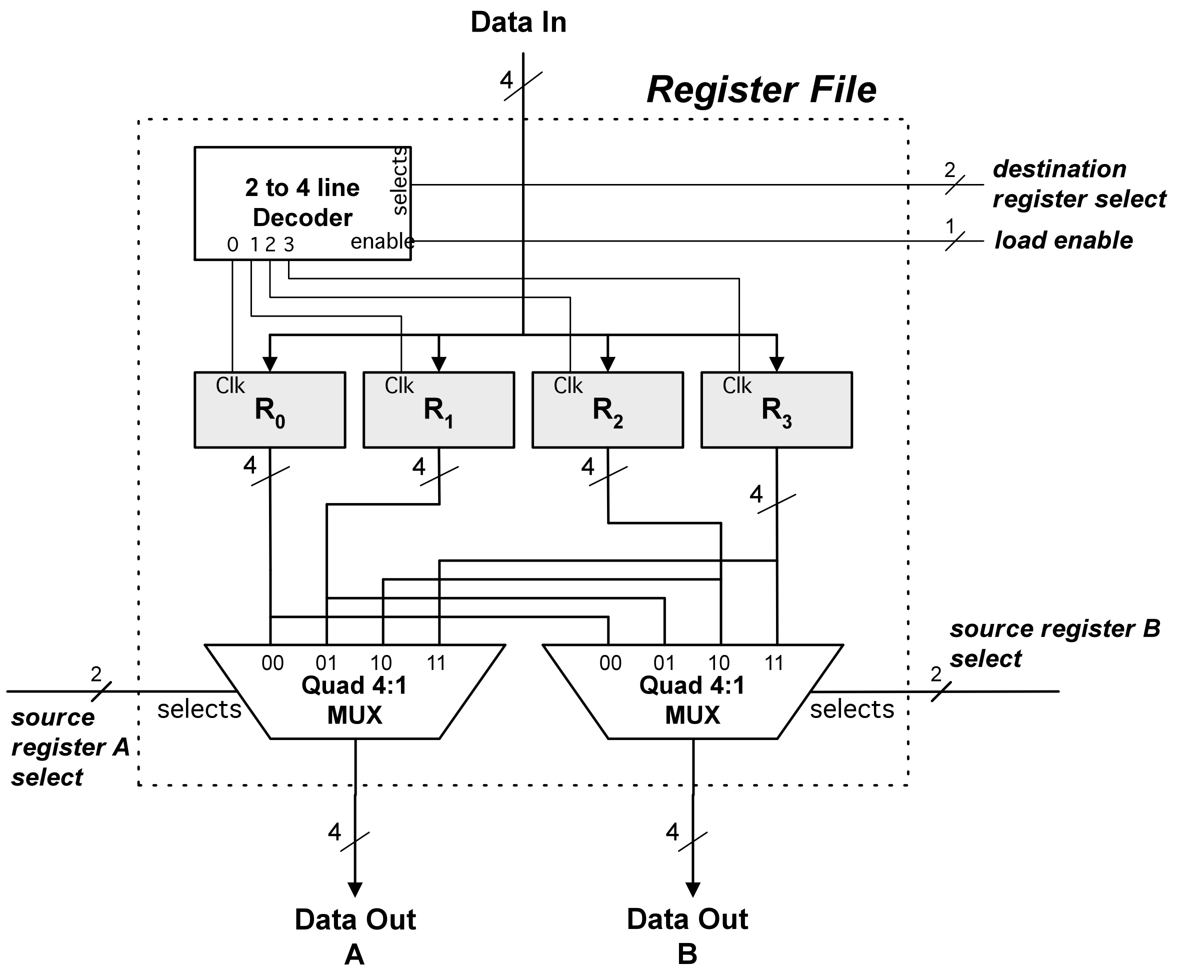
Function	Outpu	Operatio
Select	t	n



S2 S1 S0	F	
0 0 0	0 0 0 0	clear
0 0 1	B - A	subtract
0 1 0	A - B	subtract
0 1 1	A + B	add
1 0 0	A XOR B	exclusive OR
1 0 1	A OR B	logical OR
1 1 0	A AND B	logical AND
1 1 1	1 1 1 1	preset

Step 7 Modify the Register File

To be able to include an ALU in our datapath, we must first modify our register file design so that it has the capability to select *two* registers as outputs (Source Register A and Source Register B). This will allow the contents of any two registers to be applied to the **A** and **B** inputs of the ALU. This is easily achieved by adding a second Quad 4:1 MUX to the design of the register file as shown below:



The inputs to this register file are now:

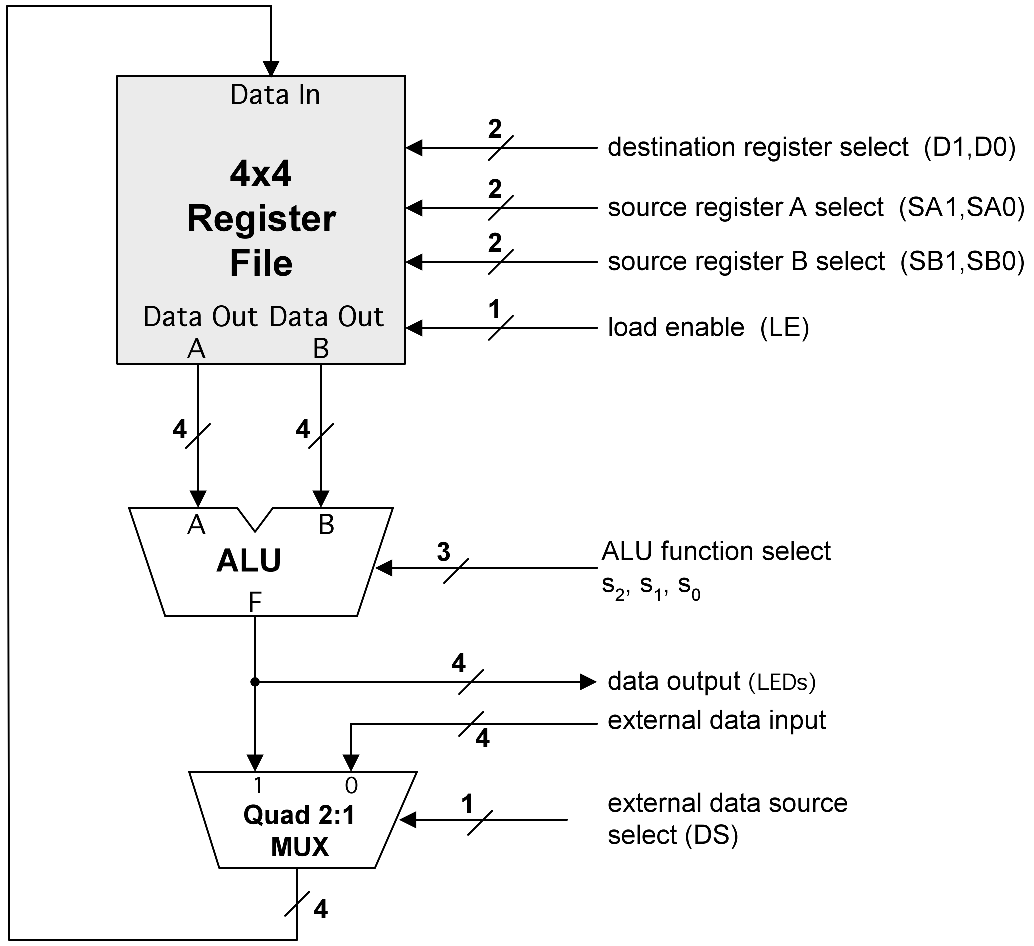
destination register select (2 bits), ***source register A select*** (2 bits), ***source register B select*** (2 bits) and ***load enable*** (1 bit).

The outputs are ***Data Out A*** and ***Data Out B*** (each 4 bits) as shown.

Compile this design and create a symbol for this register file.

Step 8 Datapath with ALU

Now, the datapath design can be extended by including the ALU from *Step 6* and the register file from *Step 7* as shown in the next figure:



Using the graphic design editor, build this circuit using the previous components and compile, implement and test. This datapath has *much* more capability than our first design in *Step 5*:

(a) as before, any register can be loaded with external data from switches:

$$\mathbf{R_d} \leftarrow \mathbf{data} \text{ (4 bits) (where d = 0, 1,2 or 3)}$$

(b) any register can be loaded with the result of any of the eight functions supported by our ALU whose input is the data contained in any two of the registers. For example:

$$\mathbf{R_d} \leftarrow \mathbf{R_A} + \mathbf{R_B} \quad \text{(where d, A, B = 0, 1, 2 or 3)}$$

would allow us to load any register with the sum of any two registers.

The inputs [D1, D0, SA1, SA0, SB1, SB0, s_2, s_1, s_0, DS] comprise a 10-bit control word which specifies a destination register (D1, D0) , the two source registers (SA1, SA0) and (SB1, SB0), and the ALU function (s_2, s_1, s_0) that operates on the source registers. The DS input allows loading of the registers with external data via the Quad 2:1 MUX. For DS=0, external data from switches is loaded into the destination register; for DS=1, data is transferred from the ALU output to the destination register. As in *Step 5*, once the control word and data input (if appropriate) are set on the level switches, execution is achieved by applying a load enable (LE) input (pulse lo-hi-lo) to the register file. This LE input may be considered as the clock to the entire system. You can view the results of each operation using four LEDs connected to the output of the ALU as shown. The function that is executed in response to a control word and a LE clock input is called a ***microoperation***. A series of microoperations applied to a datapath is called a ***microprogram***.

Try the following examples of microoperations in your implementation:

Destination Register D1 D0	**Source Register A SA1 SA0**	**Source Register B SB1 SB0**	**ALU function select s_2 s_1 s_0**	**Data Source DS**	**data input**	**Operation**
0 0	X X	X X	X X X	0	a b c d	R0 ←abcd
0 1	X X	X X	X X X	0	a b c d	R1 ← abcd
1 0	X X	X X	X X X	0	a b c d	R2 ← abcd
1 1	X X	X X	X X X	0	a b c d	R3 ← abcd
0 0	0 0	0 1	0 1 1	1	XXXX	R0 ← R0 + R1
0 1	0 0	1 0	0 1 0	1	XXXX	R1 ← R0 - R2
1 0	X X	X X	0 0 0	1	XXXX	R2 ← 0
1 1	X X	X X	1 1 1	1	XXXX	R3 ← -1
0 0	1 0	1 1	1 1 0	1	XXXX	R0 ← R2 AND R3
0 1	0 1	1 0	1 0 1	1	XXXX	R1 ← R1 OR R2
1 0	0 1	1 1	1 0 0	1	XXXX	R2 ← R1 XOR R3
1 1	1 1	1 1	1 0 0	1	XXXX	R3 ← R3 XOR R3

Note: *(1) the first four lines of this table allow for initializing the register contents* (DS = 0).
(2) this is not a complete table of all possible microoperations that can execute

What microoperation is implemented with the control word: **10 01 11 011 1** *?*

What is the effect of the last microoperation in this table ?

What microoperation is implemented with the control word: **11 11 11 010 1** *?*

Can you think of yet another way to implement this same operation ?

If you want to experiment further, try changing the ALU functions. For example, replace the "preset" function with an "increment" (A+1) operation. Add status bits for Zero (Z) and Sign (S) as outputs from the ALU. Changes can be done very quickly in our modular design !

The VHDL source code for Quad 4:1 MUX

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

ENTITY quad4to1mux IS
      PORT (      a, b, c, d        : IN  STD_LOGIC_VECTOR(3 DOWNTO 0) ;
                  s                 : IN  STD_LOGIC_VECTOR(1 DOWNTO 0) ;
                  f                 : OUT STD_LOGIC_VECTOR(3 DOWNTO 0) );
END quad4to1mux ;

ARCHITECTURE Behavior OF quad4to1mux IS
BEGIN
      WITH s SELECT
            f <=  a WHEN "00",
                  b WHEN "01",
                  c WHEN "10",
                  d WHEN OTHERS ;
END Behavior ;
```

The VHDL source code for 4-bit Register

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

ENTITY reg4 IS
      PORT (      D      : IN       STD_LOGIC_VECTOR(3 DOWNTO 0) ;
                  Clk    : IN       STD_LOGIC ;
                  Q      : OUT      STD_LOGIC_VECTOR(3 DOWNTO 0) );
END reg4 ;

ARCHITECTURE Behavior OF reg4 IS
BEGIN
PROCESS (Clk)
  BEGIN
    IF Clk'EVENT AND Clk = '1' THEN Q <= D;
    END IF;
END PROCESS ;

END Behavior ;
```

The VHDL source code for Quad 2:1 MUX

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

ENTITY quad2to1mux IS
      PORT (      a, b        : IN  STD_LOGIC_VECTOR(3 DOWNTO 0) ;
                  s           : IN  STD_LOGIC ;
                  f           : OUT       STD_LOGIC_VECTOR(3 DOWNTO 0) );
END quad2to1mux ;

ARCHITECTURE Behavior OF quad2to1mux IS
BEGIN

f <= a WHEN s='0' ELSE b ;

END Behavior ;
```

The VHDL source code for 8-function, 4-bit ALU

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

ENTITY alu IS
      PORT (      s           : IN  STD_LOGIC_VECTOR(2 DOWNTO 0) ;
                  A, B        : IN  STD_LOGIC_VECTOR(3 DOWNTO 0) ;
                  F           : OUT STD_LOGIC_VECTOR(3 DOWNTO 0) ) ;
END alu ;

ARCHITECTURE Behavior OF alu IS
BEGIN
      PROCESS ( s, A, B )
      BEGIN
            CASE s IS
                  WHEN "000" =>
                        F <= "0000" ;
                  WHEN "001" =>
                        F <= B - A ;
                  WHEN "010" =>
                        F <= A - B ;
                  WHEN "011" =>
                        F <= A + B ;
                  WHEN "100" =>
                        F <= A XOR B ;
                  WHEN "101" =>
                        F <= A OR B ;
                  WHEN "110" =>
                        F <= A AND B ;
                  WHEN OTHERS =>
                        F <= "1111" ;
            END CASE ;
      END PROCESS ;
END Behavior ;
```