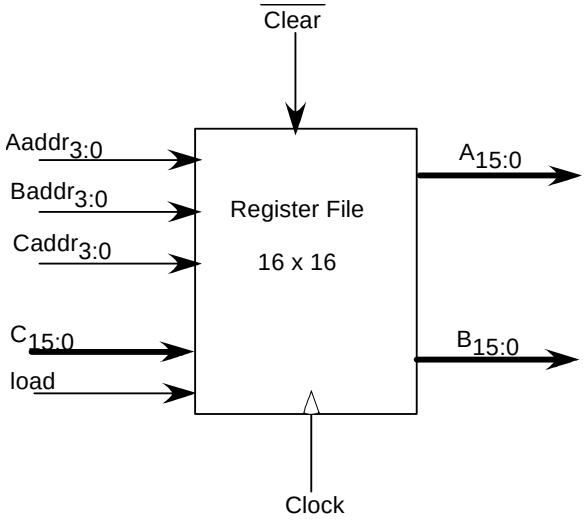
Laboratory 3

Objective

The objective of this lab is to develop structural and behavioral models for a register file and an ALU. These components are basic building blocks for your CPU. The problems *must* be solved using the components from the KURM RTL library provided with these labs.

Problem 1

Using components from the KRM RTL library, develop a behavioral and a structural model of a single input, dual output register file. A block diagram of the register file is shown below.



The following entity definition defines the template interface for your register file:

```
entity reg_file is
  port (
    A : in  word;       -- input data port
    Aaddr : in  nibble; -- register select for input A
    B,C : out word;     -- output data ports
    Bddr,Caddr : in nibble; -- register select for outputs
                            -- B and C respectively
    load : in  bit;     -- synchronous load enable
    clear : in  bit;    -- asynchronous, negative logic
                        -- clear signal
    clk : in  bit);     -- rising edge triggered clock
end entity reg_file;
```

Signals `A`, `B` and `C` represent the single data input and dual data outputs respectively. `Aaddr`, `Baddr`,and `Caddr` are the register numbers for storing input `A` data and for outputting `B` and `C` data, respectively. When the `load` signal is asserted, a rising edge on `clk` causes the data input on `A` to be stored in the register identified by `Aaddr`. If the `load` signal is not asserted, a rising edge has no effect. When the `clear` signal is asserted, all register values should be set to 0. Note that `clear` is asynchronous and is negative logic.

Implement two architectures for the register file entity. The first architecture should use behavioral techniques similar to those used in Laboratory 2. Represent your registers as an array of words and use a process to update register contents and register file outputs. The second architecture should implement your design using structural techniques. Specifically, your architecture should include only instantiated entities and no behavioral code. *You may use only devices defined in the KURM library to develop the structural register file.*

Implement a test bench for your register file to demonstrate correctness. Your test bench should include both a behavioral and structural model for your register file. A process should drive the two models with the same inputs to show that they model the same system. The following entity/architecture pair defines a template for your test system:

```
entity reg_file_test is
end reg_file_test;

architecture mixed of reg_file_test is
  Define signals to drive and observe models
begin  -- mixed
   dut1: entity work.reg_file(beh)
    port map ( instantiate behavioral model );
   dut2: entity work.reg_file(struct)
    port map ( instantiate structural model );
   test: process
    begin  -- process test
      Drive behavioral and structural models with the same
      input signals and compare outputs.
    end process test;
end architecture mixed;
```

Note that the same signals can be used to drive both models, i.e., both models can share input signals. However, different signals must be used to observe the outputs to avoid multiple driver errors.

Please note the use of the types `word` and `nybble` that are defined in the memory utilities package. You must include a use clause prior to your entity definition to include these definitions.

Problem 2

Using components from the KURM RTL library, develop structural and behavioral models of a sequential ALU to support the KURM instruction set. Your ALU should perform operations that support the mathematical and logical instructions as well as offset calculation for memory access and branching. To support these requirements, your ALU should minimally implement:

1. Unsigned addition of 16-bit numbers
2. 2s-complement addition/subtraction of 16-bit numbers
3. Logical “and”
4. Logical “or”
5. Set on less than
6. Branch if not equal

In addition to a result, your ALU should generate control outputs for word equivalence and word less than. Your ALU should provide for carry in, carry out and a 2s-complement overflow indicator. The following entity defines a template for your ALU:

```
entity alu is
  port (
    x, y      : in  word;     -- dual inputs
    z         : out word;     -- single word result
    c_in      : out bit;      -- carry in
    c_out     : out bit;      -- carry out
    lt, eq, gt: out bit;      -- comparison indicator bits
    overflow  : out bit;      -- overflow indicator
    c         : in  bit_vector(2 downto 0)); -- operation select
end entity alu;
```

The x and y inputs and the z output represent the two data inputs and one data output of the ALU. The c_in and c_out bits represent carry in and carry out, respectively. The three comparison indicator bits, lt, eq, and gt, are asserted when x < y, x = y and x > y are true, respectively. The overflow bit is asserted when the add or subtract operations generate an overflow, as defined in standard 2s-compliment mathematics. Finally, c is a vector of three control bits that select the operation the ALU will perform.

Use the encoding for control inputs shown in the table below. You may also add other operations that you feel might be helpful in designing your CPU. Further note that the comparison bits are always output without regard to the operation being performed.

ALU Control Lines	Function
0 0 0	And
0 0 1	Or
0 1 0	Add
0 1 1	Subtract (beq)
1 1 1	Set-on-less-than

Implement two architectures for the ALU entity. The first architecture should use behavioral techniques similar to those used in Laboratory 2. Use a case statement to model the various operations performed by the ALU. The second architecture should implement your design using structural techniques. Specifically, your architecture should include only instantiated entities and no behavioral code. *You may use only devices defined in the KURM library to develop the ALU.*

Implement a test bench for your ALU models to demonstrate correctness. Your test bench should include both a behavioral and structural model for your ALU. A process should drive the two models with the same inputs to show that they model the same system. Use a template similar to that given in Problem 1 to implement the test.