LABORATORY 4

ARITHMETIC LOGIC UNIT & REGISTER FILE

OBJECTIVES

No.	Objectives	Requirements
1	Implement a simple Arithmetic Logic Unit – ALU – performing eight functions.	▪ Write HDL code to implement the system. ▪ Re-used logic components from previous lab to build up this system.
2	Implement Register File – RF.	

EXPERIMENT 1

Objective: Known how to implement an ALU.

Requirements:

➢ Write Verilog/System Verilog HDL code to implement ALU block with 8-bit input data performing eight functions with 4-bit Flags output (N, Z, C, V).

➢ Propose a suitable circuit design for an ALU performing operations shown in **Table 1** and simulate it.

ALU_op_i[2:0]			Function
0	0	0	Add, A+B
0	0	1	Subtract, A-B
0	1	0	Logical shift left
0	1	1	Logical shift right
1	0	0	AND
1	0	1	OR
1	1	0	NOT
1	1	1	Forwarding input, A

Table 1 ALU operations.

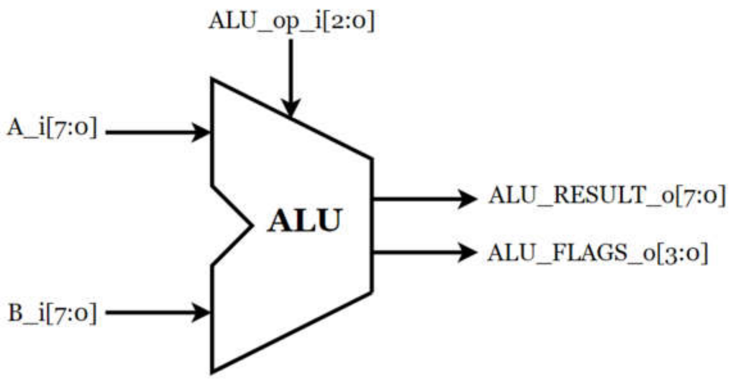


Figure 1 Block diagram of 8-bit ALU.

Signal	Description
A_i[7:0]	Input operand A
B_i[7:0]	Input operand B
ALU_op_i[2:0]	2-bit control signal which specifies which function to perform.
ALU_RESULT_o[7:0]	Output of ALU
ALU_FLAGS_o[3:0]	Extra outputs indicating information about the ALU output.

Table 2 ALU specification.

Instructions:

➢ An Arithmetic Logic Unit (ALU) integrates various mathematical and logical operations into a unified unit. In this part, students examine a sample of a basic ALU (block diagram) featuring four functions with output flags. Additionally, labs introduce the concept of shifters and rotators, covering not only their definitions but also several types of commonly utilized shifters.

➢ ***First***, a typical ALU might perform addition, subtraction, AND, and OR operations. ALU forms the heart of most computer systems. For instance, **figure 2(a)** shows a symbol for an N-bit ALU with N-bit inputs and outputs.

ALUControl[1:0]		Function
0	0	Add
0	1	Subtract
1	0	AND
1	1	OR

Table 3 Operations of simple ALU.

➢ ALU receives a 2-bit control signal ***ALUControl*** that specifies which function to perform. Control signals will generally be shown in ***blue*** to distinguish them from the data. **Table 3** lists typical functions that ALU can perform.

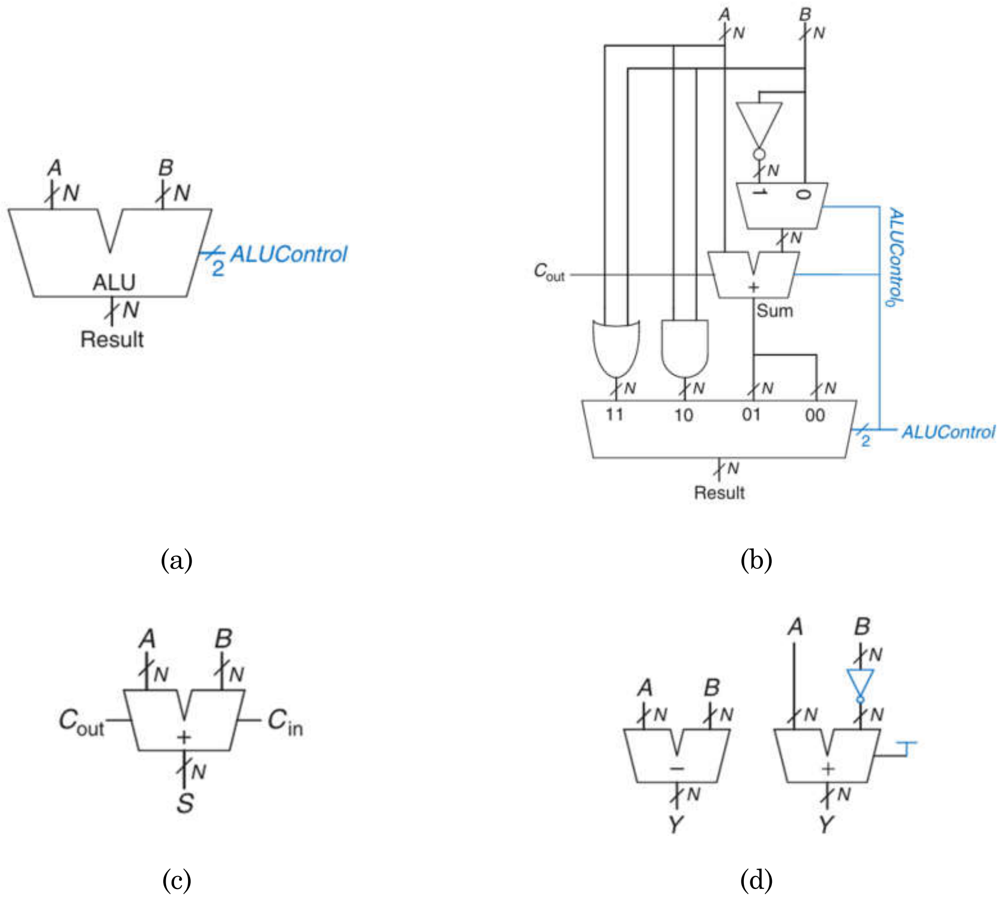
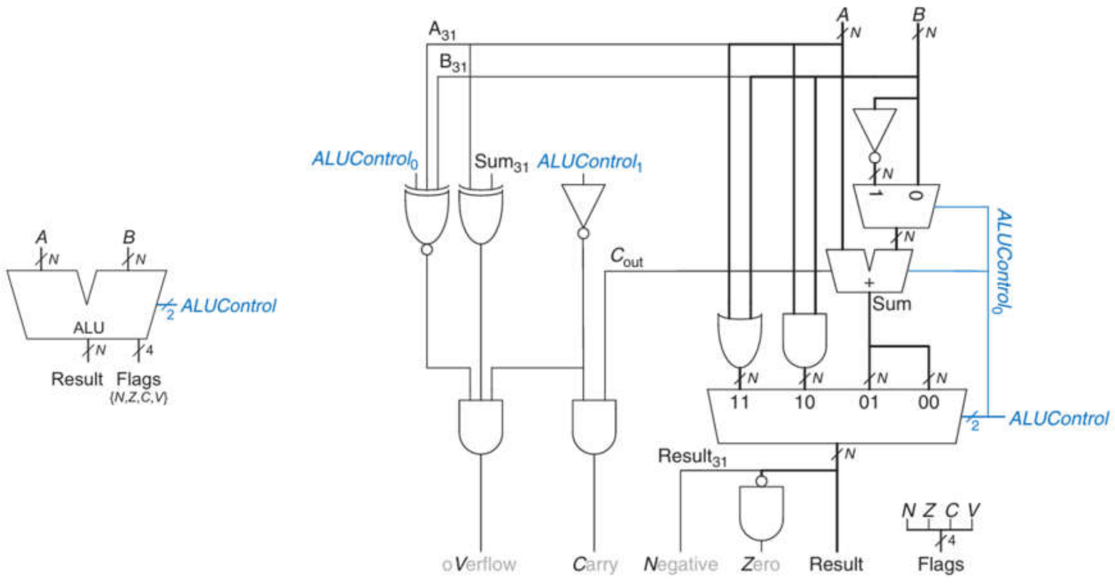


Figure 2 (a) ALU symbol (b) N-bit ALU (c) Carry Propagate Adder (d) Subtractor: symbol, and implementation.

➢ **Figure 2(b)** shows an implementation of the ALU which contains:
  - An N-bit adder and N 2-input AND, and OR gates.
  - Inverters and a multiplexer to invert input B when ***ALUControl[0]*** is asserted.
  - A 4:1 multiplexer chooses the desired function based on ***ALUControl***.

➢ According to table 2 presenting operations ALU N-bit in **Figure 2(b)**:
  - If *ALUControl[1:0] = 00*, the output multiplexer chooses A + B.
  - If *ALUControl[1:0] = 01*, the ALU computes A – B.
    - $\bar{B} + 1 = -B$ in two's complement arithmetic. Because *ALUControl[0]* is 1, the adder receives inputs A and asserted carry in, causing it to perform subtraction: $A + \bar{B} + 1 = A - B$.
  - If *ALUControl[1:0] = 10*, the ALU computes A AND B.
  - If *ALUControl[1:0] = 11*, the ALU performs A OR B.



(a) (b)

Figure 3 (a) ALU symbol with output flags (b) N-bit ALU with output flags.

➢ Some ALUs produce extra outputs, called flags, that indicate information about the ALU output. **Figure 3(a)** shows the ALU symbol with a 4-bit Flags output. As shown in the schematic of this ALU in **Figure 3(b)**, the Flags output is composed of the N, Z, C, and V flags that indicate, respectively, that the ALU output, **Result**, is negative or zero or that the adder produced a carry out or overflowed. Recall that the most significant bit of a two's complement number is 1 if it is negative and 0 otherwise.

- The *N (Negative) flag* is connected to the most significant bit of the ALU output.
- The *Z (Zero) flag* is asserted when all of the bits of **Result** are 0, as detected by the N-bit NOR gate in **Figure 3(b)**.
- The *C (Carry out)* flag is asserted when the adder produces a carry out and the ALU is performing addition or subtraction (indicated by *ALUControl[1] = 0*).
- Overflow detection, as shown on the left side of **Figure 2(b)**, occurs when the addition of two same signed numbers produces a result with the opposite sign.
  - *V (oVerflow)* is asserted when all three of the following conditions are true:
    - (1) the ALU is performing addition or subtraction (*ALUControl[1] = 0*).
    - (2) A and Sum have opposite signs, as detected by the XOR gate.
    - (3) overflow is possible.
  - That is, as detected by the XNOR gate,

- Either A and B have the same sign and the adder is performing addition (*ALUControl[0] = 0*) or
- A and B have opposite signs and the adder is per forming subtraction (*ALUControl[0] = 1*).

▪ The 3-input AND gate detects when all three conditions are true and asserts V.

➢ ***Shifters and rotators*** move bits and multiply or divide by powers of 2. As the name implies, *a shifter shifts a binary number left or right by a specified number of positions*. Several kinds of commonly used shifters exist:

○ **Logical shifter** – shifts the number to the left or right and fills empty spots with 0's.

Example: 11001 >> 2 = 00110; 11001 << 2 = 00100

○ **Arithmetic shifter** – is the same as a logical shifter, but on right shifts fills the most significant bits with a copy of the old most significant bit. This is useful for multiplying and dividing signed numbers. Arithmetic shift left is the same as logical shift left.

Note that, the operators <<, >>, and >>> typically indicate shift left, logical shift right, and arithmetic shift right, respectively.

Example: 11001 >>> 2 = 11110; 11001 << 2 = 00100

○ **Rotator** – rotates a number in a circle such that empty spots are filled with bits shifted off the other end. ROR and ROL indicate rotate right, and rotate left, respectively.

Example: 11001 ROR 2 = 01110; 11001 ROL 2 = 00111

➢ An *N*-bit shifter can be built from N N:1 multiplexers. The input is shifted by 0 to $(N - 1)$ bits, depending on the value of the $\log_2 N\_bit$ select lines. **Figure 3** shows the symbol and hardware of 4-bit shifters. Depending on the value of the 2-bit shift amount ***shamt[1:0]***, the output Y receives the input A shifted by 0 to 3 bits. For all shifters, when *shamt[1:0] = 00*, Y = A.

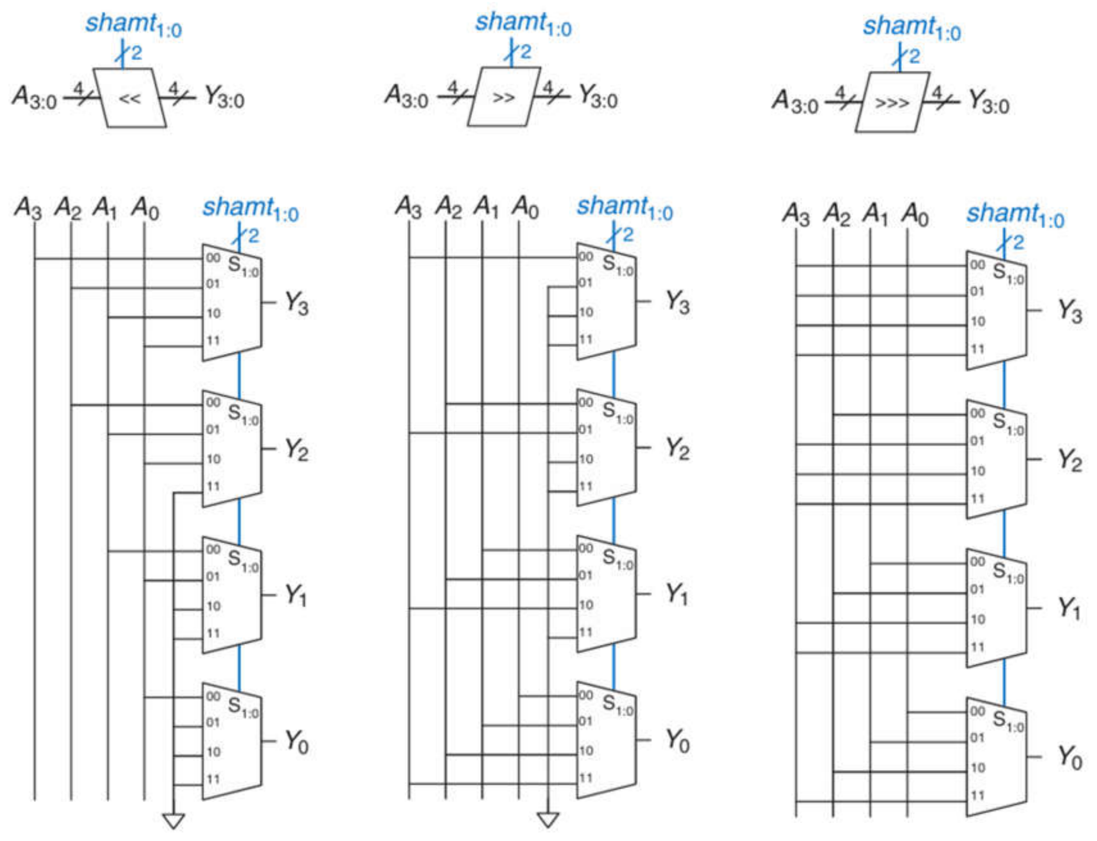
➢ In addition, there are two features of left shift and arithmetic right shift that you may pay attention to:

○ A left shift is a special case of multiplication. A left shift by N bits multiplies the number by 2^N.

Example: 000011B << 4 = 110000B is equivalent to $3D \times 2^4 = 48D$.

○ An arithmetic right shift is a special case of division. An arithmetic right shift by N bits divides the number by 2^N.

<u>Example:</u> 11100D >>> 2 = 11111D is equivalent to $\frac{-4D}{2^2} = -1D$.



(a) (b) (c)

Figure 4 4-bit shifters: (a) shift left, (b) logical shift right, (c) arithmetic shift right.

➢ In this experiment, students should create two bit wide buses for input, internal nodes and output. To add a bus, click the <u>*Wire (wide)* icon</u> or <u>Schematic Editor L > Create > Wire (wide)</u> or <u>Shift + W</u>. Draw the bus as you would draw any normal (narrow) wire.

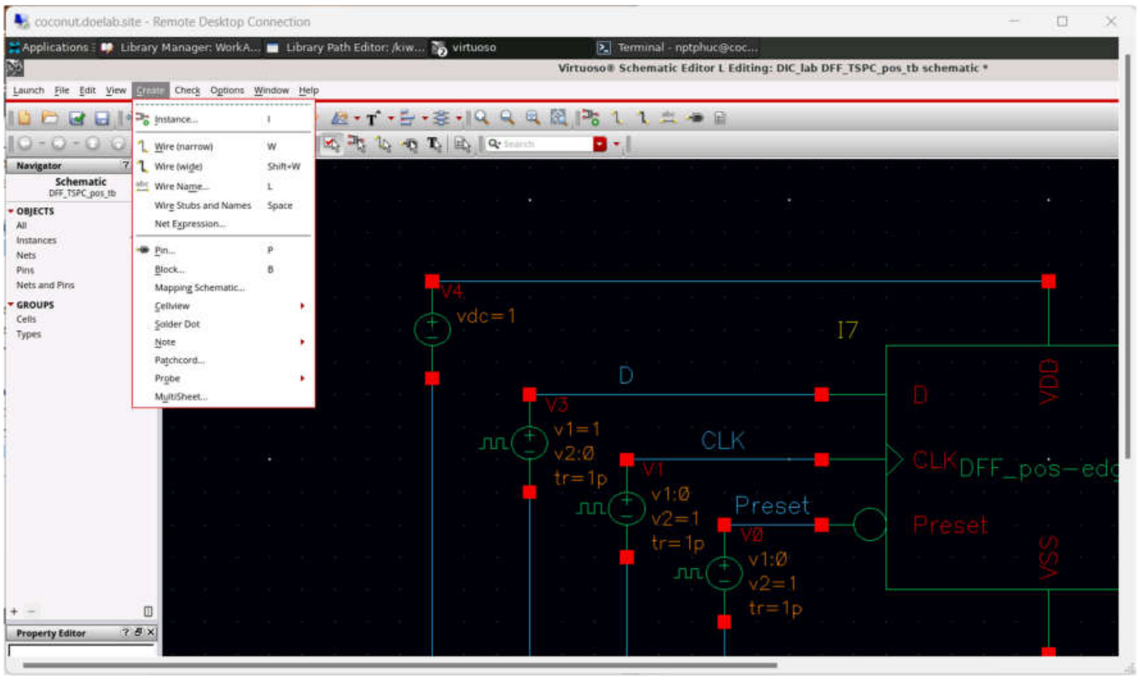
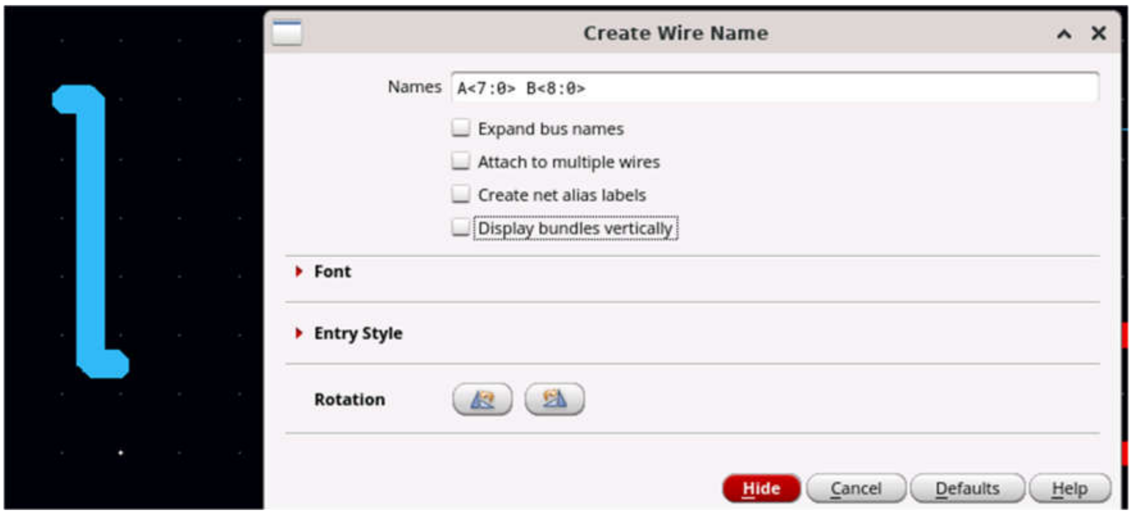


Figure 5 Instruction to create wide wire.

➢ After creating the bus, the bus must be named. The same command to add wire names for narrow wires is used, however the syntax for naming the bus is slightly different.

➢ Bus name is in the form **`Name<a:b>`**, where **`a`** and **`b`** denotes the range of bits of the bus. In addition, it is possible to add several names to different buses at one time. To do this, make sure the *Bus Expansion* button is off and simply type the names of the buses separated by a space in the *Names* field. The names will be added in order when you click on the various nets.


Create Wire Name

Names: A<7:0> B<8:0>

- [ ] Expand bus names
- [ ] Attach to multiple wires
- [ ] Create net alias labels
- [ ] Display bundles vertically

Figure 6 Create Wire Name window.

➢ To draw or tap wires to and from the buses, draw wires from the buses and name them correspondingly to their bits. To name the individual bit lines, type in the bus name (e.g. **A<7:0>** or **<7:0>**) and turn on *Bus Expansion* in the Add Wire Name window. The *Bus Expansion* button is used to extract individual bit names from the bus. When you start placing the names on the wires, you will notice that the first name will be **A<7>** (or **<7>**), the next will be **A<6>** (or **<6>**), and so on.

Check: Students must show in your report these results.

➢ The waveform to prove your design in HDL correctly, and students should show all the cases of flags. Besides, the result of RTL viewer is obligatory to present.

➢ Students show the waveforms of simulation results performed on your ALU by verifying the outputs of the final ALU with three different patterns. For each pattern, the results are verified for all eight components.
(Note: students can use any addition algorithm such as: Ripple Carry Adder, Carry Look-Ahead…)

A[7:0]	B[7:0]	ALU_op_i	ALU_RESULT_o[7:0]	Function
0000_0000	0101_0101	000		ADD
0000_0000	0101_0101	001		SUB
0000_0000	0101_0101	010		SLL
0000_0000	0101_0101	011		SRL
0000_0000	0101_0101	100		AND
0000_0000	0101_0101	101		OR
0000_0000	0101_0101	110		NOT
0000_0000	0101_0101	111		Forward A

Table 4 First Test Vector Truth Table.

A[7:0]	B[7:0]	ALU_op_i	ALU_RESULT_o[7:0]	Function
1111_1111	0110_0110	000		ADD
1111_1111	0110_0110	001		SUB
1111_1111	0110_0110	010		SLL
1111_1111	0110_0110	011		SRL
1111_1111	0110_0110	100		AND
1111_1111	0110_0110	101		OR
1111_1111	0110_0110	110		NOT
1111_1111	0110_0110	111		Forward A

Table 5 Second Test Vector Truth Table.

A[7:0]	B[7:0]	ALU_op_i	ALU_RESULT_o[7:0]	Function
0010_1010	1001_1111	000		ADD
0010_1010	1001_1111	001		SUB
0010_1010	1001_1111	010		SLL
0010_1010	1001_1111	011		SRL
0010_1010	1001_1111	100		AND
0010_1010	1001_1111	101		OR
0010_1010	1001_1111	110		NOT
0010_1010	1001_1111	111		Forward A

Table 6 Second Test Vector Truth Table.

EXPERIMENT 2

Objective: Known how to implement Register File – RF.

Requirements:

- Write VHDL/Verilog/System Verilog HDL code to implement Register File including eight 8-bit registers.
- Propose a suitable circuit for your design and simulate it.

Instructions:

- A typical datapath has multiple registers. The construction of a bus system with a large number of registers requires different techniques. A set of registers having common operations performed on them may be organized into a register file. The ***register file*** (RF) contains all the registers and provides two read ports and one write port. The RF always provides the contents of the registers corresponding to the read register inputs on the outputs.
- In general, a register file is denoted as $2^m \times n$, where m is the number of register address bits and n is the number of bits per register. In this experiment, students will implement $2^3 \times 3$ register file. The block diagram of register file, Register File specifications, and register file architecture are introduced in **Figure 7**, **Table 7**, and **Figure 8**, respectively.

Besides, the register file architecture shown in **Figure 8** is not obligatory to follow, students can feel free to modify or propose your design, however, please present it similar to **Figure 7**, **Table 7**, and **Figure 8**.

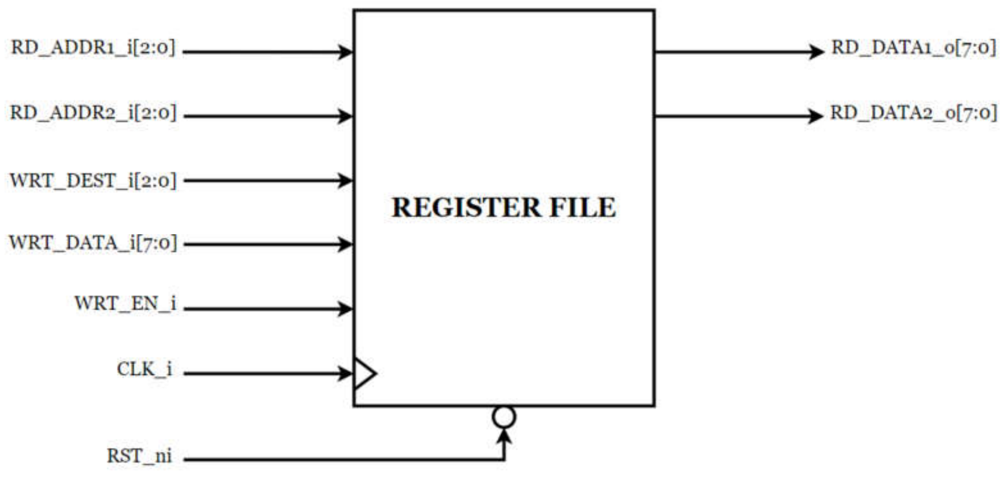


Figure 7 Block diagram of Register File.

Signal	Description
CLK_i	Positive clock
RST_ni	Low negative reset
RD_ADDR1_i[2:0]	The first address to be read from.
RD_ADDR2_i[2:0]	The second address to be read from.
WRT_DEST_i[2:0]	Address supposed to be written into the register file.
WRT_DATA_i[7:0]	Data supposed to be written into the register file.
WRT_EN_i	1: write to the Register File
	0: read the Register File
RD_DATA1_o	The read data from first address.
RD_DATA2_o	The read data from second address.

Table 7 Register File specifications.

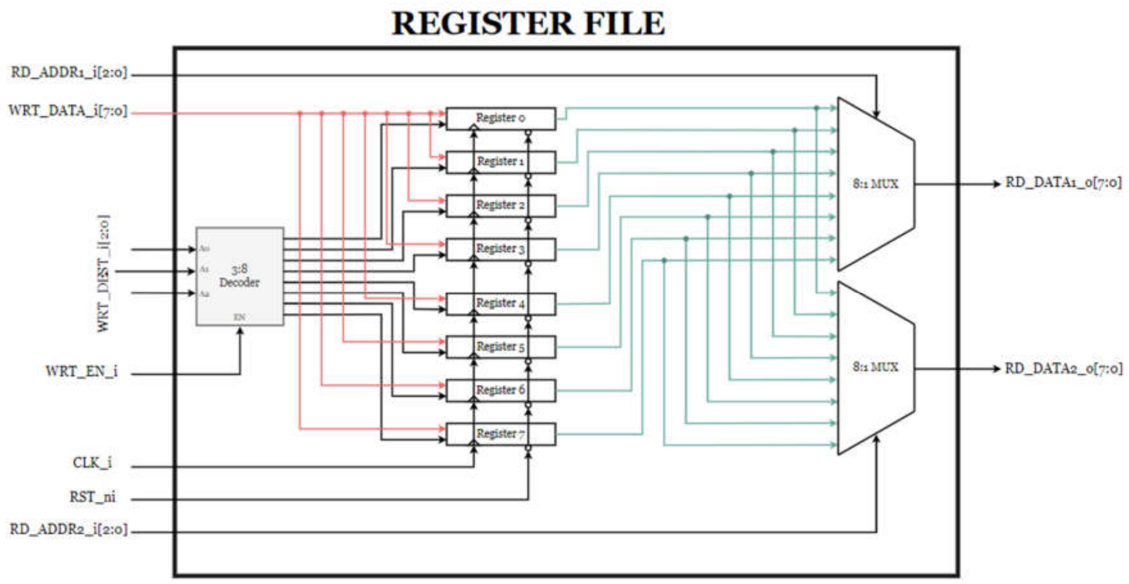


Figure 8 Implementation of 8 × 8 Register File.

- There are some points that students need to take care of:
  - The structure of registers in the register file is **PIPO**.

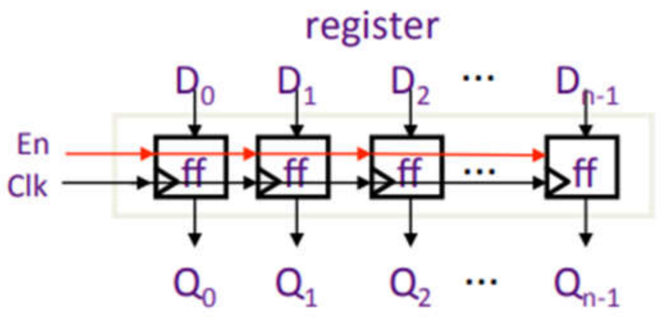


Figure 9 The registers structure.

  - Reads and writes are always completed in one cycle.
    - A read can be done any time (i.e. combinational).
    - A write is performed at the rising clock edge.

If *WRT_EN_i = 1* → the write address and data must be stable at the clock edge.

Check:

- The waveform to prove your design in HDL correctly, and the result of RTL viewer is obligatory to present.
- Students will show the waveforms of simulation results performed on your RF in circuit design.

❖ ONE MORE THING...

After completing this lab, please answer the following questions.

- How much time did you spend completing this lab?
- Is there any problem when you do? Please let me know.