Datapath Components

Overview: In a series of laboratory exercises you will incrementally implement a 32-bit processor that supports a basic set of operations. You will start with implementing the datapath components required by the processor. Later you will put together a datapath using these components and design a state machine that controls this datapath. Datapath will be configured by the controller to execute a specific operation in a single clock cycle. This way the datapath will be able to execute different operations in each cycle. You will eventually execute matrix multiplication code on your datapath to validate the functionality of your datapath. For now, let's start with the design and implementation of the basic datapath components described below.

Demo: You should have your ***post-synthesis*** waveform simulation for each component.

Requirement: You must turn in your source files.

References:

Lecture Notes

Objectives:

- Practice writing Datapath components and their testbench
- Learn to use the ***post-synthesis simulation*** for your design

32-bit processor architecture is provided on Canvas to design/write the following Datapath components:

1) (15 pts) **32-bit 2x1 Multiplexer (MUX)**: Write "Mux32Bit2to1.vhd" file to implement 32-bit 2x1 mux. Run Post-synthesis Functional simulation on your 32-bit mux.
   For simulation,
   a) Run Behavioral simulation. If working, continue to b)
   b) Under Synthesis, choose Run synthesis. Then under Simulation, choose Run Post-synthesis Functional simulation. It should provide the same output waveform as b).
   c) Run Post-synthesis Timing simulation. You will now see that the change in output waveform occurs with some delays after the change in input waveform.
   When you get the correct waveform, show your waveform simulation to your TA.

2) (5 pts) **PCAdder**: Write "PCAdder.vhd" file to implement an adder that always adds one input by 4. Complete the testbench and run Post-synthesis Functional simulation on your PCAdder. Use at least 4 cases in your testbench – one case should be 0xFFFFFFFC. When you get the correct waveform, show your waveform simulation to your TA.

3) (5 pts) **ProgramCounter**: Write "ProgramCounter.vhd" file to implement a 32-bit register with a synchronous reset. Complete the testbench and run Post-synthesis Functional simulation on your ProgramCounter. Use at least 3 cases in your testbench.

4) (5 pts) **SignExtension** module: Write "SignExtension.vhd" file to implement a sign extension module. Write the testbench and run Post-synthesis Functional simulation on your module.

Description - Sign extension module.
If the most significant bit of in (in[15]) = 0, create the 32-bit "out" output by making out[15:0] equal to "in" and make other bits (bits 16 to 31) to 0.
If the most significant bit of in (in[15]) = 1, create the 32-bit "out" output by making out[15:0] equal to "in" and make other bits to 1.

5) (45pts) **32-bit ALU**:

(10 pts) Use the comments in the given .vhd file in the folder "ALU32Bit" to implement (write the code for) the 32-bit ALU with 11 operations.

(30 pts) Write the testbench and run Post-synthesis Functional simulation on your 32-bit ALU. Use the following cases to complete the testbench.

(5 pts) Complete this calculation by hand (use your calculator if needed) and show your answers to the TAs – they should match with your output waveforms.

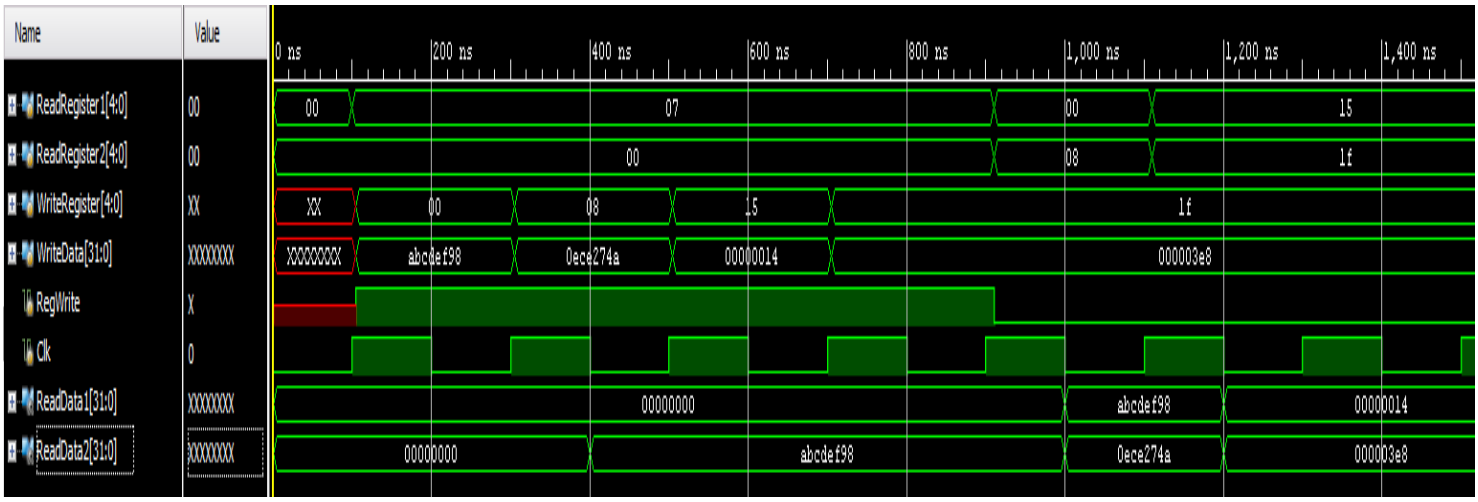
```
// Module - ALU32Bit.vhd
// Description - 32-Bit wide arithmetic logic unit (ALU).
//
// INPUTS:-
// ALUControl: 4-Bit input control bits to select an ALU operation.
// A: 32-Bit input port A.
// B: 32-Bit input port B.
//
// OUTPUTS:-
// ALUResult: 32-Bit ALU result output.
// ZERO: 1-Bit output flag.
//
// FUNCTIONALITY:-
// Design a 32-Bit ALU, so that it supports a set of arithmetic and
// logical operaitons. The 'ALUResult' will output the corresponding
// result of the operation based on the 32-Bit inputs, 'A', and 'B'.
// The 'Zero' flag is high when 'ALUResult' is '0'.
// The 'ALUControl' signal determines the function of the ALU based
// on the table below.

// Op | ALUControl  value  | Description | Notes
// ========================
// ADDITION                 | 0000 | ALUResult = A + B
// SUBRACTION               | 0001 | ALUResult = A - B
// MULTIPLICATION           | 0010 | ALUResult = A * B       (see notes)
// AND                      | 0011 | ALUResult = A and B
// OR                       | 0100 | ALUResult = A or B
// SET LESS THAN            | 0101 | ALUResult =(A < B)? 1:0  (see notes )
// SET EQUAL                | 0110 | ALUResult =(A=B)  ? 1:0
// SET NOT EQUAL            | 0111 | ALUResult =(A!=B) ? 1:0
// LEFT SHIFT               | 1000 | ALUResult = A << B      (see notes )
// RIGHT SHIFT              | 1001 | ALUResult = A >> B       (see notes )
// ROTATE RIGHT             | 1010 | ALUResult = A ROTR B     (see notes )
```

ALU Control	A	B	ALUResult	Zero
0000 (add)	0x00000064	0x00000056	_____	_____
0001 (sub)	0x00000064	0x00000056	_____	_____
0010 (mul)	0x00000064	0x00000056	_____	_____
0011 (AND)	0x00000064	0x00000056	_____	_____
0100 (OR)	0x00000064	0x00000056	_____	_____
0101 (A<B)	0x00000064	0x00000056	_____	_____
0110 (A==B)	0x00000064	0x00000056	_____	_____
0111 (A!=B)	0x00000064	0x00000056	_____	_____
0000 (add)	0xFFFF0000	0x0000000F	_____	_____
0001 (sub)	0xFFFF0000	0x0000000F	_____	_____
0010 (mul)	0xFFFF0000	0x0000000F	_____	_____
0011 (AND)	0xFFFF0000	0x0000000F	_____	_____
0100 (OR)	0xFFFF0000	0x0000000F	_____	_____
0000 (add)	0xFFFFFFFF	0x00000001	_____	_____
0001 (sub)	0xFFFFFFFF	0x00000001	_____	_____
0000 (add)	0xFFFFFFFF	0xFFFFFFFF	_____	_____
0001 (sub)	0xFFFFFFFF	0xFFFFFFFF	_____	_____
0010 (mul)	0xFFFFFFFF	0xFFFFFFFF	_____	_____
0100 (A<B)	0xFFFFFFFF	0xFFFFFFFF	_____	_____
0110 (A==B)	0xFFFFFFFF	0xFFFFFFFF	_____	_____
0111 (A!=B)	0xFFFFFFFF	0xFFFFFFFF	_____	_____
1000 (A<<B)	0x00000FED	0x00000005	_____	_____
1001 (A>>B)	0x00000FED	0x00000005	_____	_____
1010 (A ROTRB)	0x00000FED	0x00000005	_____	_____

6) (25 pts) **32x32 Register File**: Use the comments in the given .vhd file in the folder "RegisterFile" to implement the 32x32 register file. Write a testbench (Read carefully to understand all the test cases) and run Post-synthesis Functional simulation on your 32x32 Register File.

See below for the waveform after the post-synthesis functional simulation of 32x32 Register File



Grading Criteria

- Submit all the vhd source files (.vhd only) including testbenches on Canvas to the designated dropbox.