6) 8-bit logic processor on FPGA

a. Summary of all .SV modules and the changes you made to extend the

processor to 8-bits. Specifically, you need to describe even modules which

were provided, but you did not create.

1. Processor.sv: The Processor.sv is the top-level module which works to put all the other modules together. Within the file it creates instances of the compute\_unit module, reg\_unit module, and others to finalize the workings of an 8 bit logical processor.

Changes:

(1) input logic [3:0] Din 🡪input logic [7:0] Din;

(2) output logic [3:0] Aval 🡪 output logic [7:0] Aval;

output logic [3:0] Bval 🡪 output logic [7:0] Bval;

(3) logic [3:0] A 🡪 logic [7:0] A;

logic [3:0] B 🡪 logic [7:0] B;

logic [3:0] Din\_S 🡪 logic [7:0] Din\_S;

1. Register\_unit.sv: This module is responsible for creating the two 8-bit registers by instantiating them with the Reg\_4.sv file, respectively, Register A and Register B. This file will execute load and shift instruction.

Changes:

1. input logic [3:0] D 🡪 input logic [7:0] D;
2. output logic [3:0] A 🡪 output logic [7:0] A;
3. output logic [3:0] B 🡪 output logic [7:0] B;
4. Reg\_4.sv: For Reg\_4.sv, this module works to update or reset the bits held inside each register.

Changes:

1. input logic [3:0] D 🡪 input logic [7:0] D;
2. output logic [3:0] Data\_Out 🡪 output logic [7:0] Data\_Out;
3. Compute.sv: This file is used as the computation unit, responsible for providing all 8 functions to be executed.

No change.

1. Router.sv: The Router.sv file as the name suggests works to push the correct bits into the specified register.

No change.

1. Control.sv: This module is used for controlling the shift/halt/reset state transitions through the input Execute signal.

Changes:

1. We increase the number of counts from 4 to 8;

enum logic [3:0] {

s\_start,

s\_count0,

s\_count1,

s\_count2,

s\_count3,

s\_done

} curr\_state, next\_state;

🡪 enum logic [7:0] {

s\_start,

s\_count0,

s\_count1,

s\_count2,

s\_count3,

s\_count4,

s\_count5,

s\_count6,

s\_count7,

s\_done

} curr\_state, next\_state;

1. We increase the number of shifts of states from 4 to 8;

s\_count0 : next\_state = s\_count1;

s\_count1 : next\_state = s\_count2;

s\_count2 : next\_state = s\_count3;

s\_count3 : next\_state = s\_done;

s\_done :

🡪 s\_count0 : next\_state = s\_count1;

s\_count1 : next\_state = s\_count2;

s\_count2 : next\_state = s\_count3;

s\_count3 : next\_state = s\_count4;

s\_count4 : next\_state = s\_count5;

s\_count5 : next\_state = s\_count6;

s\_count6 : next\_state = s\_count7;

s\_count7 : next\_state = s\_done;

s\_done :

1. HexDriver.sv: The HexDriver.sv is the module that works to turn the LEDs on. More specifically, the nibble variable is used to hold the value that needs to be put on display on one of the LEDs and the hex variable is used to identify which of the 7 segments should be turned on or off to effectively display the character/number on the 7-segment display.

No change.

1. Synchronizers.sv: The Synchronizers.sv file is used to synchronize the inputs to the circuit with the rising clock edge. So, when the clock is at its rising edge, the output bit will be updated using flip flops. Since synchronization is the same for both the 4-bit logical processor and the 8-bit, no changes needed to be made for this file.

No change.

b. RTL block diagram - please only include the top-level design if using the RTL

viewer.

A diagram of a computer

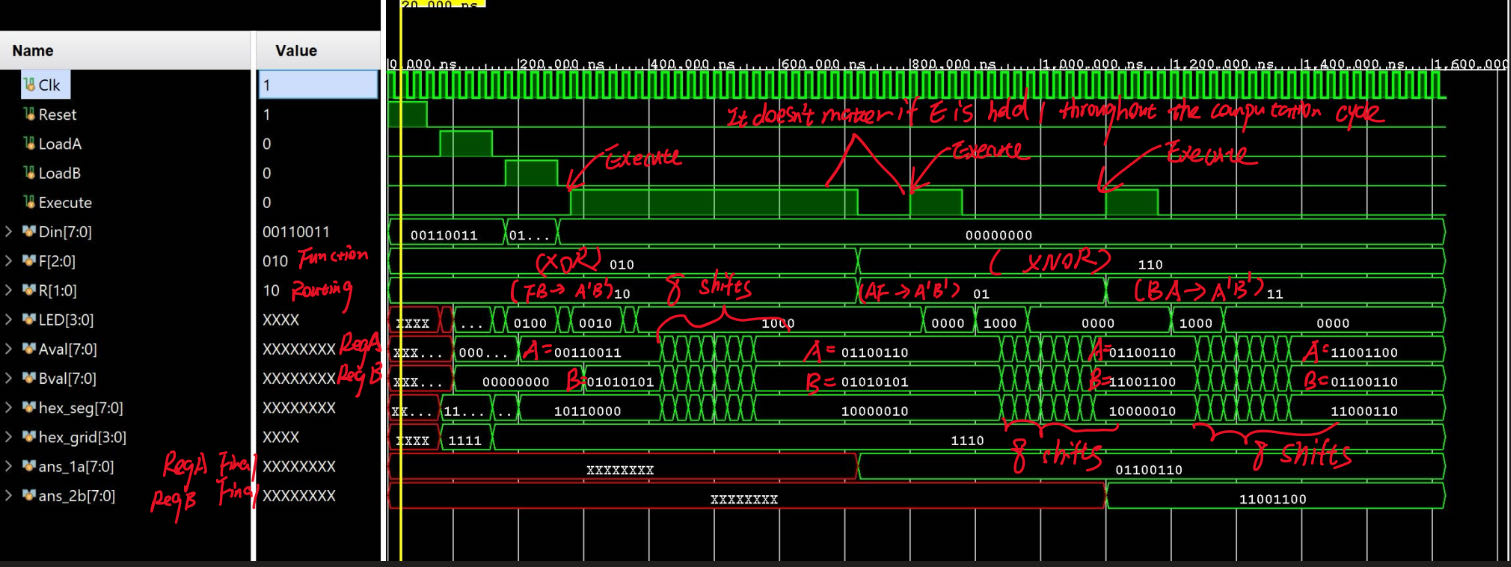
Description automatically generated

3.12

c. Include a simulation of the processor that has notes (annotations) that give

information such as what operation is being performed, where the result was

stored, etc.



d. Include procedure used to generate Vivado Debug Core trace, as well as the

result of such trace executing an example operation. E.g.: “Step 1, Set the

trigger on signal <signal here>…”

**Procedure used to generate Vivado Debug Core trace:**

1. First, go to “Set up Debug” under the Synthesis tab.
2. On the “Nets to Debug” tab, find Register A named \*Aval\_OBUF\*, Register B named \*Bval\_OBUF\*, and the Execute bit labeled \*Execute\_IBUF\*.
3. Set both the probe type of \*Aval\_OBUF\* and \*Bval\_OBUF\* to “Data.” This is because we only want the data of both register A and B to display on the waveform.
4. Set \*Execute\_IBUF\*’s probe type to “Data and Trigger.” This way, one will not only use the “Execute” bit as the trigger but also see the value of the “Execute” bit.
5. Once these values have been inputted for the Debug Core, one must Run Synthesis/Implementation and Generate Bitstream again. These processes may take longer than usual to compile since adding a debug core adds more logic to facilitate the debugging process.
6. Once these processes have finished compiling, program the FPGA with both the bitstream file and the debug probes file that we just created.
7. Then, a waveform tab should appear. Inside the waveform tab, locate trigger setup. Set the operator to == (equal) and the value to B (both transitions).
8. After setting Register A, Register B, the function, and routing, press “Run trigger for this ILA core” and hit the Execute button on the FPGA. In the example shown below Register A = 0x33, Register B = 0x55, F2-F0 = 000 for the AND operation, and R1-R0 = 10 to make Register A hold F (solution) and Register B hold B.
9. After hitting the Execute bit, one should be able to see the waveform visual. By zooming into the portion where the Execute bit changed, one should be able to see the shifts, bit by bit. For instance, in my example below with RegA = 0x33 RegB = 0x55, one can see RegA = 0x11 since the bits that were shifted out (1 and 1) ANDed to 1 and were shifted into the MSB of RegA due to the routing specified. As for RegB, its LSB is circling around to its MSB because of the routing protocol.
10. One can see after 8 shifts, the registers have been updated based on its original values and the specific function/routing protocol. As for the example shown above, RegA now holds 0x11 and RegB holds its original value of 0x55.e.

Include the output of the debug core trace performing an operation on the 8-bit logic processor (this doesn’t need to be the same operation as the one your

**Example Output of the Debug Core Trace Performing an AND Operation with AB -> FB Routing on the 8-bit Logic Processor:**

**A screenshot of a computer

Description automatically generated**

TA asked to demonstrate, but it needs to be non-trivial).

7) Description of all bugs encountered, and corrective measures taken.

**Description of all bugs encountered:**

We first used the Execute signal itself (denoted as Execute\_IBUF) as the trigger signal but we couldn’t see any changes of bits for signal A and B. Later, Prof. Chen posted that we should use the signal LED\_OBUF[3] instead. That is because the FPGA board uses debouncing switch to control the real Execute signal that actually triggers the state machine, it had to go through a sync\_debouncer unit, which results some time delay and further makes it invisible for us to observe the change of signal A and B.