Design Implementation

I first built a module 2 to 4 one-hot decoder but I ensured that it could be changed to decoders of different input and/or output bits (output always one hot) by using parameters in the module. I used a left shift operator in the module to shift 1 over to the position specified by the binary input to create a one-hot at the output.

I then built a 5 to 32 decoder module. I use a 2 to 4 decoder and a 3 to 8 decoder (parameterized instance of the 2 to 4 decoder) to decode the lower two bits of the input into a one-hot 4 bit signal and decode the upper three bits of the input signal into one-hot 8 bit signal respectively. I then AND each bit of the 8 bit signal (using replication) to all of the 4 bit signal to produce each four-field bit of the 32 bit output.

I then made the module for 4 to 1 multiplexer that takes 4 inputs of arbitrary widths (widths can be changed by using different values of the parameter k) and a 4 bit one-hot select input. Each bit of the select input is ANDed with one of the inputs (different input for each AND gate). The select bits are replicated according to the parameter value to ensure equal signals are ANDed together. The ANDed outputs are then ORed to get the final output (which is the same width as the input).

The next module constructed was a 32 to 1 multiplexer. It takes in 32 inputs of arbitrary widths along with a binary select signal of 5 bits. The binary select signal is converted to a one-hot select signal using the already constructed 5 to 32 decoder. Then 8 different intermediate signals are created using 8 4-1 multiplexers. Each multiplexer takes 4 inputs (4 consecutive inputs) from amongst the original inputs of the 32-1 multiplexer and it also takes 4 bits from the one-hot decoded select signal where the bits cover the indices of the 4 particular inputs. These 8 intermediate signals are then ORed together to get the final output of the multiplexer.

The next module was a D-F/F which takes in a 1 bit input signal, an asynchronous clear signal, a write enable signal, and a clock signal. The always block has if statements to checks whether the asynchronous clear signal is high, or whether the positive clock edge is high along with write enabled signal and gives an output based on these checks being true or not.

A register module creates a register using 32 D-F/Fs. The module has a 32 bit input, a 32 bit output, and the same control signals as those used in the D-F/F module. I use a generate statement where a D-F/F module is used in a for loop where each D-F/F's input corresponds to the i'th bit of the input and the D-F/F's output corresponds to i'th bit of the output where i is a genvar (a temporary variable used within a generate loop). The *for loop* covers all the 32 bits.

Finally, I create my main register files. I decode the 5 bit ctrl_writeReg signal to a 32 bit one-hot select signal using a 5 to 32 decoder. I then AND each bit of this one-hot signal with the ctrl_writeEnable signal. This gives me a 32 bit signal called 'enabled' which will select the register to write when ctrl_writeEnable is high. I created 32 32-bit wires (intermediate signals). I then created 32 registers using the register module where the input was data_writeReg and the output was one of the wires. The control signals for each clock were the ctrl_reset, clock, and particular bit of enabled signal that matches the index of the

register. Then I have two 32 to 1 multiplexers which have all the 32 wires as input. The first multiplexer has the 5 bit ctrl_readRegA as the binary control signal and data_readRegA as the 32 bit output while the second multiplexer has the 5 bit ctrl_readRegB as the binary control signal and data_readRegB as the 32 bit output.

Q) Include a description of how fast you estimate your register file can be clocked. Include timing waveforms to justify your estimate (PDF, or inline).

The clock can have a period of at least 12 ns. This is because it takes approximately 11.93 ns for the data to be written and read after the clock goes high i.e. the delay is around 12 ns. This can be confirmed by the snapshot of the timing simulation shown below. This implies that the next rising edge of the clock should occur 12 ns after the previous rising edge. As a result, the register file can be clocked at **83.33 MHz** (1/12ns).