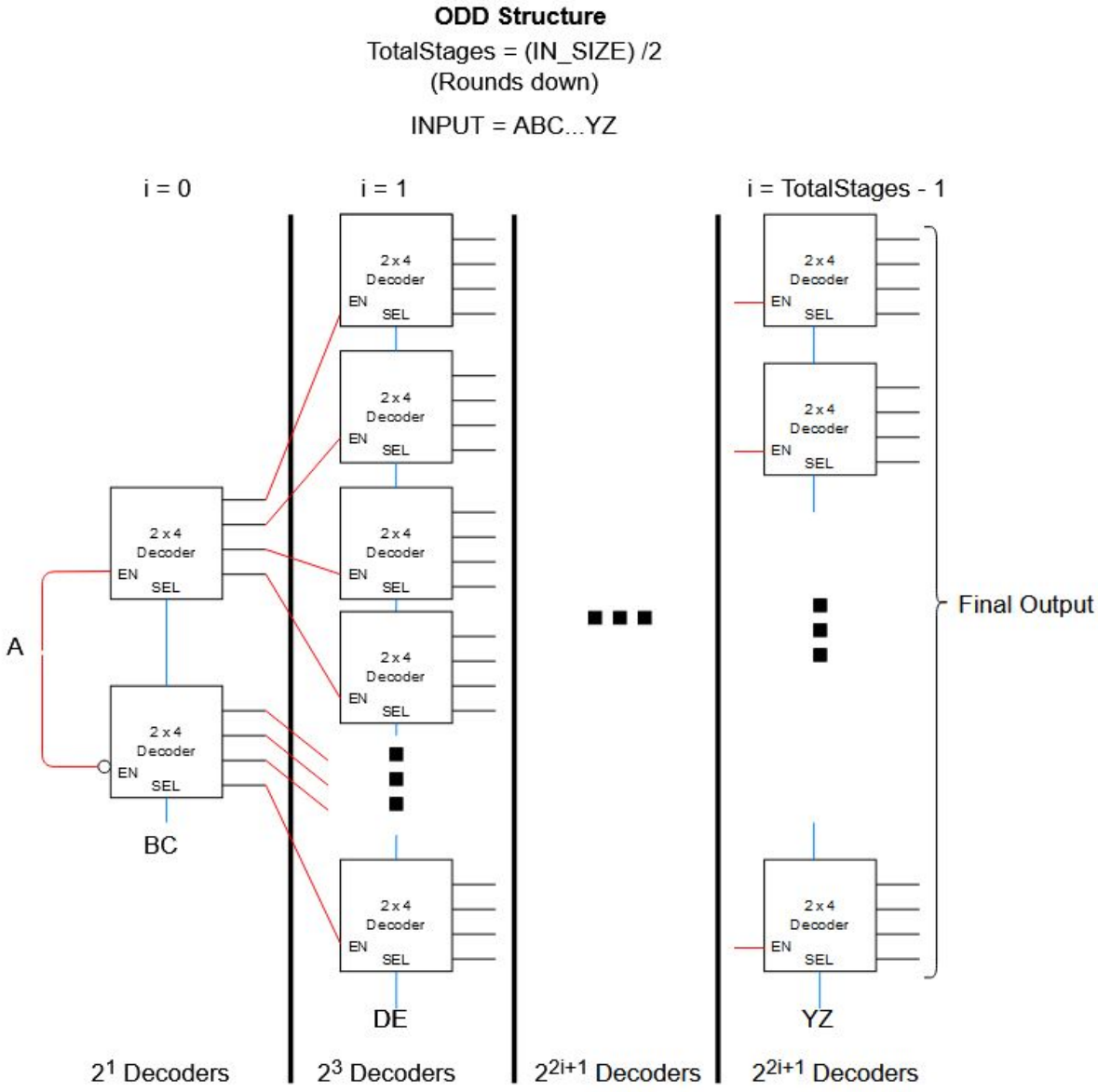
Lab 2 - Generic Decoder

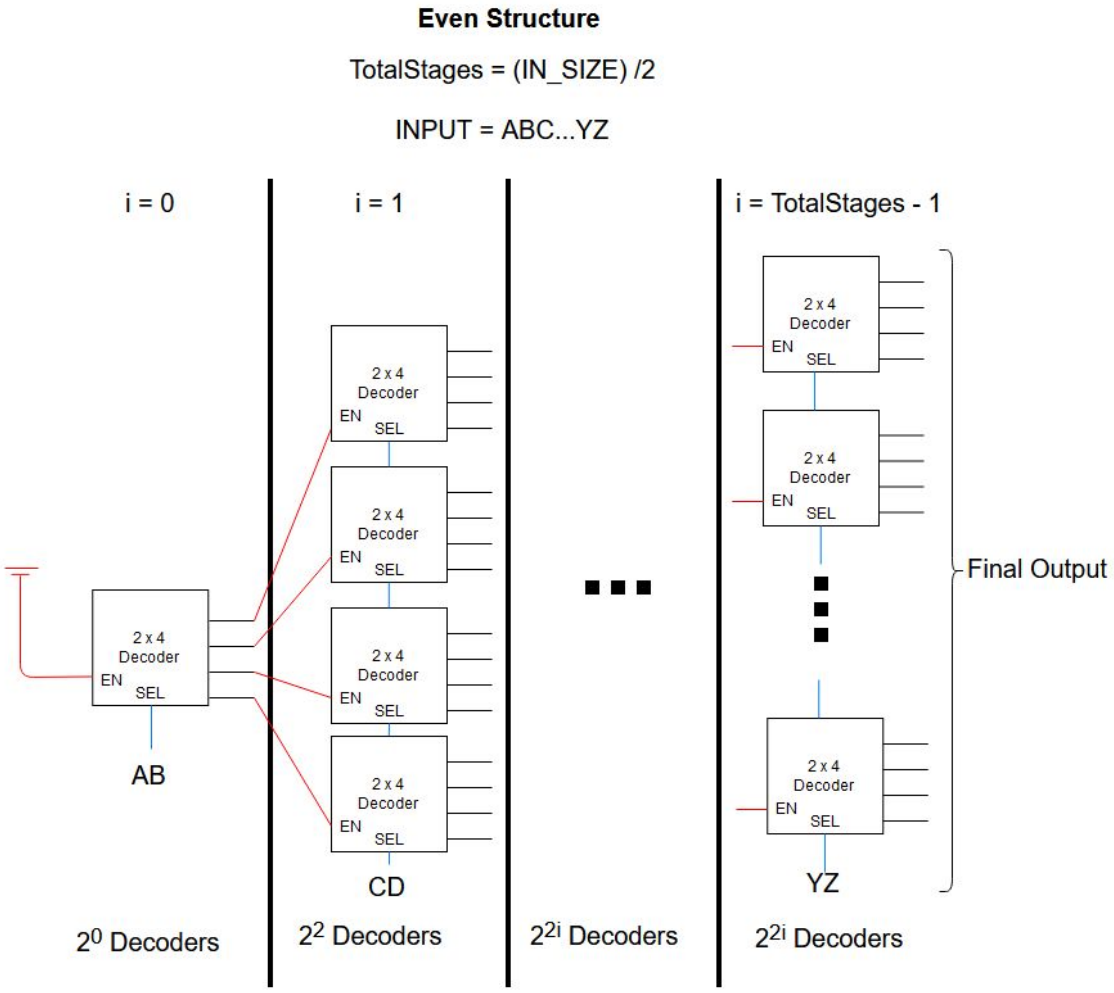
Group A - Yuta Akiya, Kyle Le, Megan Luong
Prof. Aly
ECE 4304

1) Report: Word/pdf report showing the architecture (circuit with details information before you start coding), the trick of the codes, and the list of possible corner cases you cover using testbench, pick the area/resources information from the tool.

Architecture

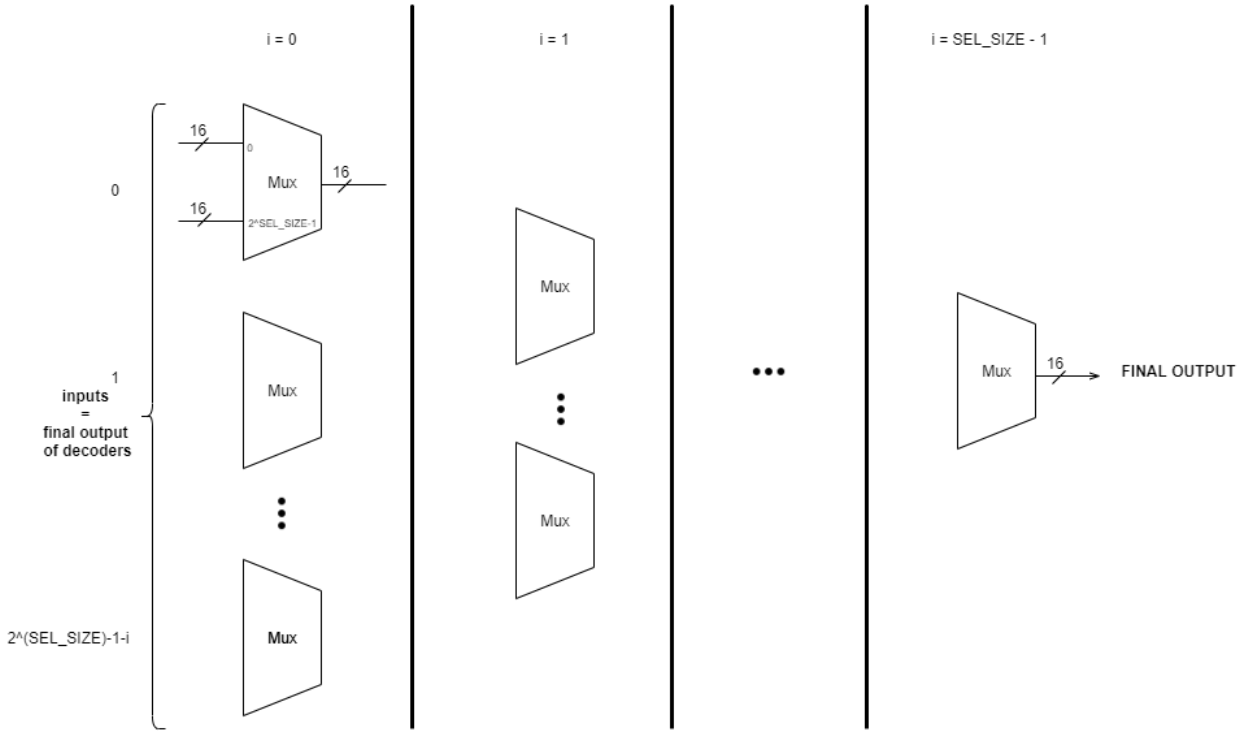
The architecture of the generic decoder is similar to that of the generic mux from lab 1. We are also limited to using 2x4 decoders as the architecture's building block, use parameterized values for all variables to make the design generic, and have a memory mechanism to store values of stages to use as inputs for the next stage.





To deal with the limited I/O, a simple component is made which takes a portion size out of a std_logic_vector array according to a SEL input. By specifying the size of that portion and the width of SEL, the values that want to be displayed can be sectioned out as needed.

Similarly to Lab 1, we created a generic Nx1 mux that could be used for different input bit sizes using layering of 2x1 muxes that also could have different input bit sizes. In this lab's case, 16-bit widths were needed to accommodate the number of leds on the board for testing. The input for the mux is the final output of the decoders which are grouped in 16-bit groups. The select size is determined by the size of the decoder output.



Code Detail

Generic Decoder:

The main difference in design comes from the fact that the generic decoder requires “if generate” statements because a decoder with an odd input size is slightly different from a decoder with an even input size. To accommodate for this difference, two “if generate” blocks are included in the architecture, one for odd and one for even decoder input size.

The memory mechanism used in the code is a 2-D array, INTERNAL_CARRY, to store enables for the decoders in each stage and a 1-D array, SEL_CARRY, to hold select values for each stage.

When the input size is even, the design is slightly simpler because it can divide evenly into the 2x4 decoders. Thus, the input, A, is directly loaded into SEL_CARRY and a fixed “1” in order to enable the first stage of decoders. The total number of stages that will be generated is calculated by rounding down the quotient of the input size and two. The number of decoders that will be generated is `2**(i * 2)` for the i th layer. For the port mapping, SEL_CARRY is mapped to the next two bits of inputs from A, the enable bit E and the output, X, is mapped to the INTERNAL_CARRY. The output will be used as the enable bits for the next stage of decoders.

When the input size is odd, we cannot divide the inputs evenly onto 2x4 decoders. Therefore, we add one additional stage consisting of 1 decoder. The inputs for this additional decoder will be a fixed ‘0’ input and the most significant bit of the input A. The output from this single decoder becomes the enable for the next stage of decoders. Therefore, INTERNAL_CARRY is loaded with the most significant bit of A and its complement. Similar to the even case generate, input size/2 rounded up will be the amount of stages generated. However, the amount of decoders in

each stage is `(2**(i * 2 + 1))` to account for the difference in design between even input and odd input decoder.

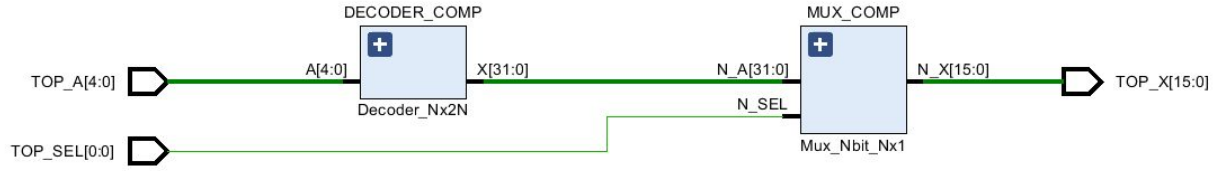
After the even or odd generate is over, the output X is loaded with the final output of the generic decoder which is stored in the last row of INTERNAL_CARRY.

Generic Mux:
The main challenge of this generic mux was to be able to accommodate different sized decoders which was solved by adding and adjusting the mux's parameters. To create this mux, we had to tweak the generic Nx1 mux and 2x1 mux created in the last lab, adding the parameter for the inputs' width and changing the input parameter. The input parameter was changed to match the input parameter of the generic decoder, so the input and select had to be calculated from that value. The output of the mux was generated by using a 2-D array to create a cascading algorithm, like the generic mux with inputs of a single bit width.

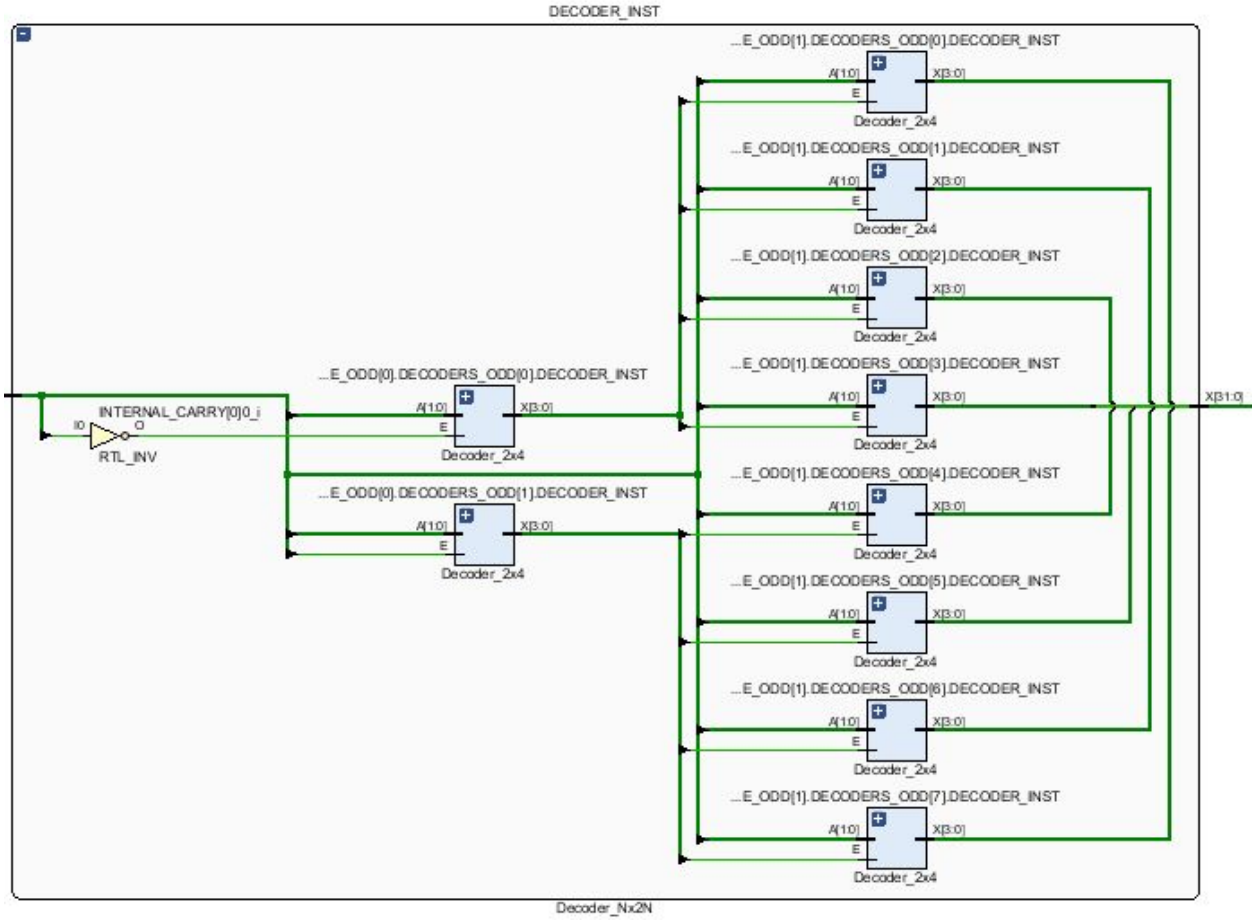
Corner Cases
Since our generic decoder design consists of two if generates as mentioned earlier, two test benches (one odd input one even input) were conducted using text I/O.
For a 5x32 decoder the input file consisted of 5 bit binary numbers starting from 00000 to 11111, where the number was incremented by one on every new line. The output file consisted of 32 bit binary numbers which one hot encoded the decoder input values.
For a 6x64 decoder, a simple Python script was created to make the input file. The inputs cover every possible input for the 6 bit input decoder (000000 to 111111). The output file, again, consisted of 64 bit one hot encoded outputs of the 6 bit input.
Since both odd and even input size cases are covered as well as every input within those two cases, all corner cases are covered.

Area/Resources Information

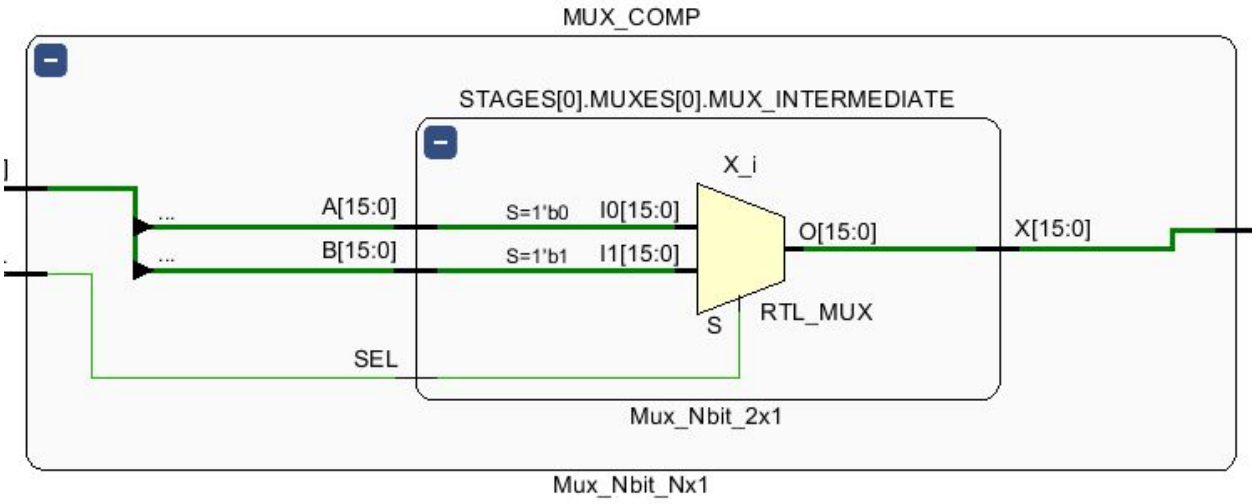
Elaborated Design of Entire Design :



Elaborated Design of Generic Decoder set to 5x32:



Elaborated Design of Generic N-bit Nx1 Mux set to 16-bit 2x1:



Resource Usage of Entire System

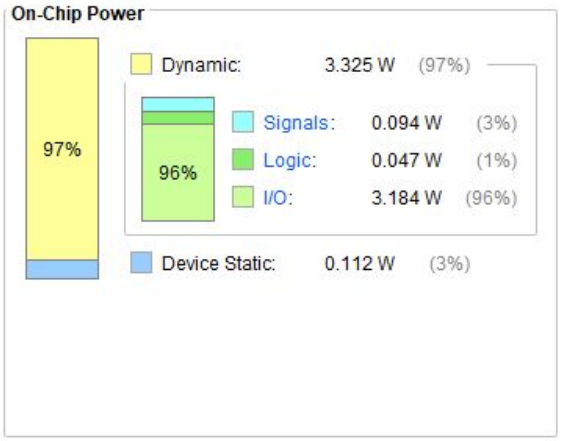
Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Total Power	Failed Routes	LUT	FF	BRAMs	URAM	DSP
synth_1	constrs_1	Synthesis Out-of-date								16	0	0.0	0	0
impl_1	constrs_1	Implementation Out-of-date	NA	NA	NA	NA	NA	3.437	0	16	0	0.0	0	0

Power Usage Details

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**3.437 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**40.7°C**
Thermal Margin:	44.3°C (9.6 W)
Effective ϑJA:	4.6°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

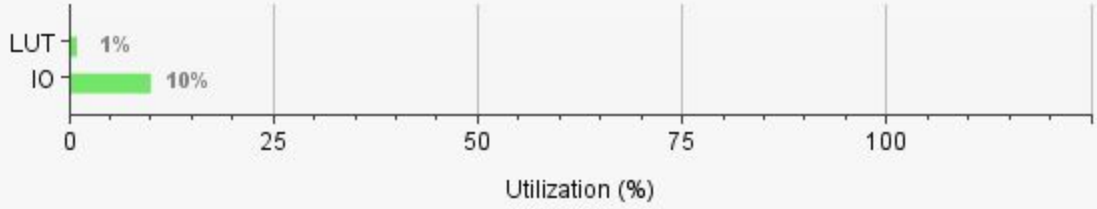
Launch Power Constraint Advisor to find and fix invalid switching activity



Post-Implementation Resource Utilization

Utilization Post-Synthesis | **Post-Implementation**

Graph | Table



Utilization Post-Synthesis | **Post-Implementation**

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	16	63400	0.03
IO	22	210	10.48