Implementation of Pipelined Double Precision Floating-point Adder.

Rohit M Kale **193079038**

Introduction:

The implementation of floating-point adders has become crucial due to increasing operating speed of processors. In order to cope up with fast operating processors, the arithmetic units are supposed to produce outputs at a faster rate. In this project, we try to implement one such algorithm for designing pipelined version of double precision floating-point adder excluding the subnormal conditions. The algorithm used is simple and straight forward and has 4 main stages namely exponent comparator, exponent adjuster, mantissa comparator, adder and normalizer. We have tried to combine as much operations as possible in these three component blocks so as to keep the design simple. All the component blocks are implemented on Verilog. We have tried to implement the code on EP4CE22F17C6 FPGA device. As this device doesn't input two 64-bit floating-point numbers we have created a self-checking testbench which is wrapper around the module and stores inputs output combinations. It inputs one pair of inputs at every clock at checks the output after the depth of the pipeline. If the output is in-correct it asserts a mistake signal otherwise keeps it low.

IEEE 754 Format:

The following sections briefly describes the IEEE 754 double precision floating-point number representation in detail. The double precision floating-point format is represented in 64-bit format as shown in Fig 1.

Fig 1. IEEE 754 double precision representation.

The 64-bit number is divided into 3 fields. First, the most significant sign bit, which indicates the sign of the number. When the bit is set, it represents a negative number. When the bit is cleared, it represents a positive number. Second, the exponent bits, these field is comprised of 11 bits, which represents the exponent of a normalized floating-point number. It has a bias of 1023 for signed as well as unsigned number representation. Therefor the exponent field having value from 0 to 1023 represents a negative exponent whereas a positive exponent has exponent field value from 1024 to 2047. The mantissa field represents the bits of the number after decimal for a normalized floating-point number. Hence, there is an implicit '1' at the MSB followed by the decimal value of mantissa. This MSB of the mantissa has to be included in the adder's input for computing the addition result. The result obtained after the addition is not in a normalized form. Hence to convert it into a normalized IEEE 754 format the MSB again has to be placed in its normal position by appropriate shifting and adjustments in the exponent field. This part is done by the normalizer component block in our design algorithm. The following table shows the representation of floating-point numbers some values of exponent and mantissa field. Few of this could be treated as extreme (sub-normal cases).

Double-Precision	Exponent (62:52) 11 bits	Mantissa (51:0) 52 bits	Value
Normalized number	1 to 2046	Anything	$\pm(1.F)_2 \times 2^{E-1023}$
Denormalized number	0	Non-zero	$\pm(0.F)_2 \times 2^{-1022}$
Zero	0	0	±0
Infinity	2047	0	±∞
NaN	2047	Non-zero	NaN

Tab 1. Overview of double point floating-point representation.

All valid double precision floating-point numbers have to be represented in the normalized form for appropriate comprehension of numbers.

Algorithm:

- Input the two floating-point numbers through the self-checking test bench.
- Sort the two numbers on the basis of their exponent.
- Shift the mantissa of the smaller exponent number right by the bit position equal to the difference of the two exponents.
- Include the implicit bit of the smaller exponent number while shifting the mantissa field.
- If the exponents are found equal, then do the following
    1) Generate a signal saying exponents are equal, the signal is de-asserted otherwise.
    2) Compare the mantissa fields of the two numbers.
    3) In this case, store the larger mantissa number in the variable where larger exponent number was supposed to be store.
    4) Store the smaller mantissa number in the other variable.
    5) If the mantissa fields are also equal, store the numbers in any of the variables. We design the next components in such a manner that this case is taken care of. If the sign bits of both the numbers are unequal, the output is ±0. If they are equal, they are added with sign of both numbers copied to the sign bit of the output.
- Store the larger exponent in one variable, and the smaller exponent number with shifted mantissa (denormalized number) in another variable.
- Copy the exponent field of the larger exponent number to that in smaller exponent field.
- If the sign bits of the numbers are unequal, subtract the lower exponent/mantissa number's mantissa field obtained from the above steps from that of the larger exponent/mantissa number. If they are equal add them. NOTE: include the implicit bit of the larger exponent number while adding/subtracting the mantissa. The inputs of mantissa adder/subtractor should be 53-bit wide.
- Copy the exponent field.
- If the sign bits are equal, set the sign bit of the output same as that of the inputs. If they are unequal, set it equal to the sign bit of the larger exponent/mantissa number.
- Store the addition result in 54-bit wide variable. The MSB holds the carry out from the mantissa addition/subtraction operation.
- Check if the entire output mantissa result has become zero or not. If it is zero, the final output result is zero.
- If the above case is not true, check if there is carry out from the addition operation of the mantissa fields. If there is copy the bits from 52 to 1 in to final output mantissa field and increment the exponent value by one. Copy the sign bit as it is.
- If the above two cases are not met, check if the 52^{nd} bit is set or not. If it is set copy the bits from 51 to 0 to final output mantissa field. Also copy the exponent field and the sign bit as it is.
- If the above three cases are not satisfied, left shift the output mantissa by one bit at a time and decrement the exponent field value. Go to above set and repeat.

Design Flow:

There are 3 main components of the floating-point adder used in this project. These are described as follows:

1) Exponent comparator and adjuster:

This block compares the exponent of the two floating-point numbers. The one with smaller exponent is modified appropriately so that its exponent matches the exponent of the other number. Since addition between any two numbers can be performed if and only if they have equal exponents. This component makes sure that the two output floating point numbers generated have equal exponents. In order to make the exponents equal, it basically shifts the mantissa of the smaller number left by the number of positions equal to difference in the exponents of the number. If by default, both the numbers have same exponent, then it asserts an output signal called exponents equal. While shifting the mantissa bits of the smaller number towards left the implicit MSB '1' is also shifted along with the other bits of the mantissa. During every operation, it keeps the amended smaller number in one variable and the larger number in other number. This reduces the task of further sorting numbers on their original values. The numbers sorted on the basis of the exponents are passed to the adder block.

2) Mantissa comparator and Adder:

The adder block is responsible for adding the mantissas of the numbers sorted by the exponent block based on the sign bits of the two number. If the sign bits of both the numbers are equal then the mantissa bits are simply added and the sign bit of the result is kept same as the both the numbers. If the sign bits of both the numbers are different then the smaller exponent number which is in turn a smaller mantissa number is subtracted from the larger exponent number. The sign bit of the result is decided based on the sign bit of the larger number. While adding or subtracting the mantissa fields the MSB of the larger number is taken into consideration. However, the MSB of the smaller number has been already shifted in the mantissa field of the smaller number during sorting. Hence, simply a zero bit is appended at the most significant bit position the smaller number. Since the exponents are already made equal in the exponent component, this field is kept untouched. The output mantissa after addition or subtraction is stored in a 54-bit field. The MSB of this field is used to handle the carry out of the addition/subtraction result which is less likely to happen. The second MSB position represents the implicit MSB of mantissa which is set to '1' for larger exponent number and set to '0' for smaller exponent number. The outputs generated from this block are output exponent field, output sign bit, and 54-bit denormalized mantissa. These outputs are passed to the normalizer block for obtaining the result in appropriate IEEE 754 format as shown in Tab 1.

3) Normalizer:

This block normalizes the output mantissa generated from the adder block. There could be many combinations for the result. The important thing for normalizing a floating-point number is having a '1' at the most significant bit of the mantissa, which is the bit position 52. At first, the normalizer checks whether the entire output mantissa has become zero. If it is so, then the final output has to be zero, since the exponents of the two numbers are already matched and the output mantissa becomes zero including the implicit MSBs indicates that result has to be zero. If the bit at 53^{rd} bit location becomes '1' then output mantissa is shifted left by one position and the exponent field is incremented. The bit at 53^{rd} location is discarded and the bits from 52 to 1 bit location is put in normalized mantissa output. If the above two mentioned case do not happen the bit at 52^{nd} bit location is checked. If this bit is '1', then the mantissa bits from 51 to 0 are taken into final 52-bit mantissa and the exponent field is kept as it is. If all the above 3 condition are not satisfied than the normalizer looks for the closest bit '1' from the bit position 52. It shifts the mantissa left one bit at a time and decrements the exponent value, now it checks for the 52^{nd} bit position if it is '1' the procedure in the third case is followed otherwise the mantissa bits are again shifted left. Till the normalizer find a '1' at the 52^{nd} bit position it repeats the process.

RTL design of floating-point adder:

The following figure shows the RTL design obtained after simulating the code.

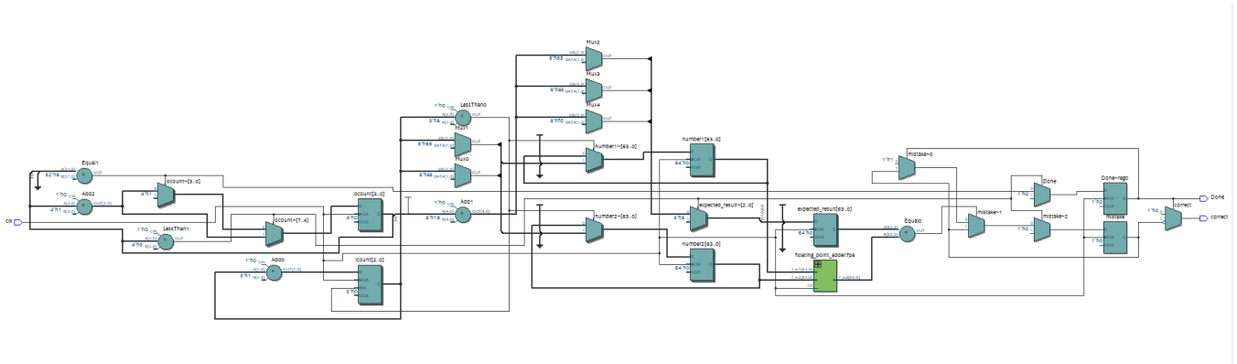

Fig 1. RTL design with the wrapper.

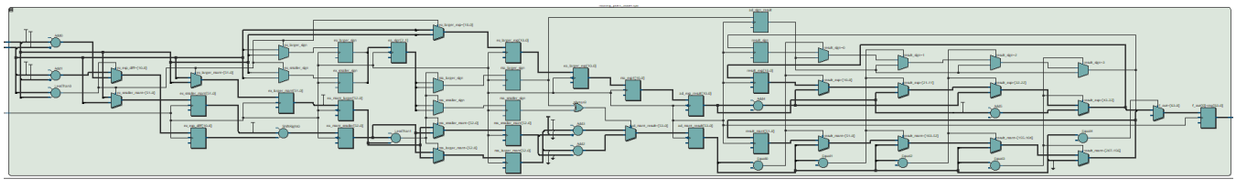

Fig 2. RTL design of Floating-point adder.

From the above figure, the signal flow from block to block is depicted.

Result:

The Verilog code is simulated on the Quartus using a test bench. The test bench inputs few inputs to the design. The RTL was run on EP4CE22F17C6 FPGA device. The following figure shows the output observed on the Modelsim.

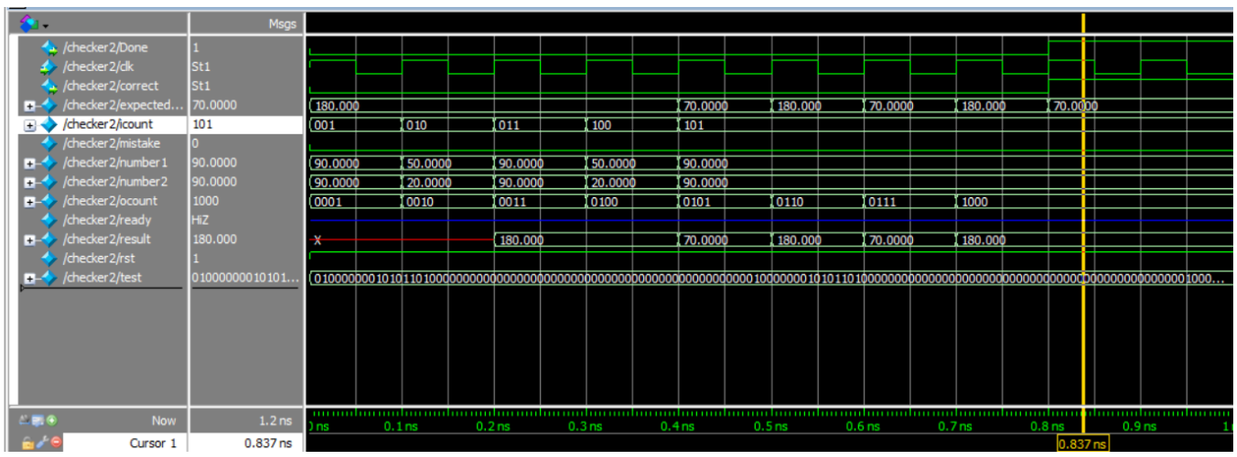

Fig 3. Result observed on Modelsim.

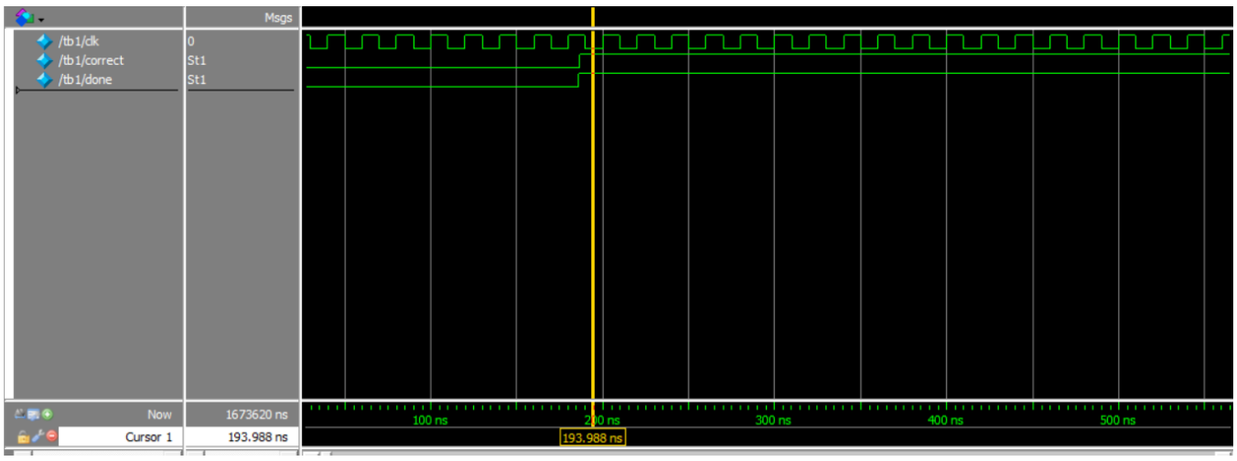

Fig 4. Gatelevel simulation for correct results

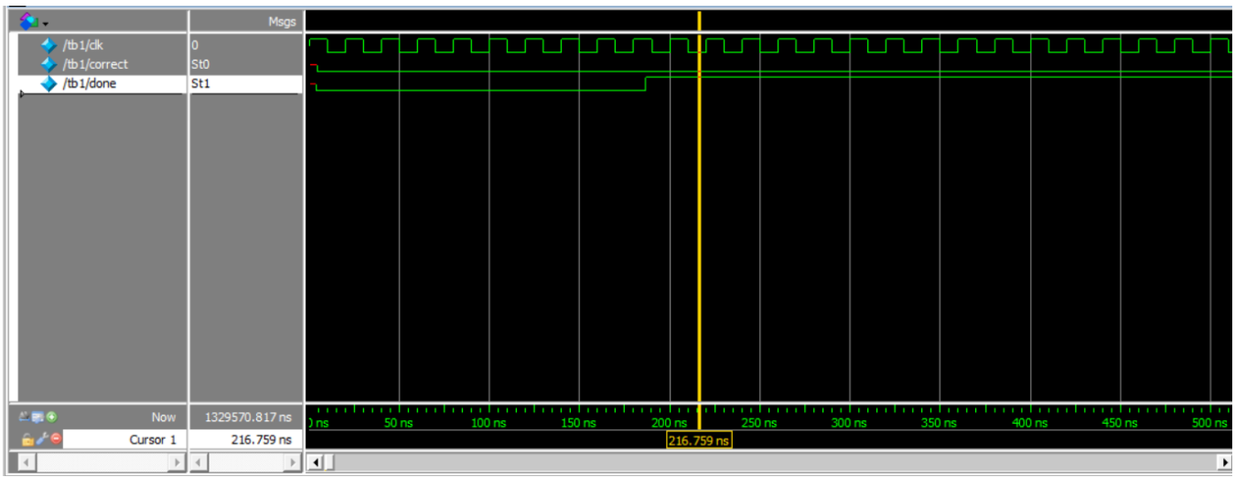

Fig 5. Gatelevel simulation for incorrect results.

As can be seen from the above figure, for various inputs for the adder, the design produces the correct output for each of the combination.

The following figures show the implementation of the double precision floating point adder on De0-Nano FPGA board. The done signal gets asserted when all the input combination stored in the checker.v (self-checking) module are applied. The correct signal is asserted when the module generated correct output for all the input combinations stored in the checker,v module. If any of the input combinations could not generate a correct ouyput, the correct signal is deasserted. For checking purpose, we introduce a digit error in one of the input combinations to check whether the checker2.v module works correctly or not. The case 2 shown in figure 9, 8 show that the module deasserts the correct signal as one of the input combinations failed to produce the desired result (which was done intentionally). Figure 6 and 7 that the signal done and correct remain ON as their design generated the correct output for all the cases.

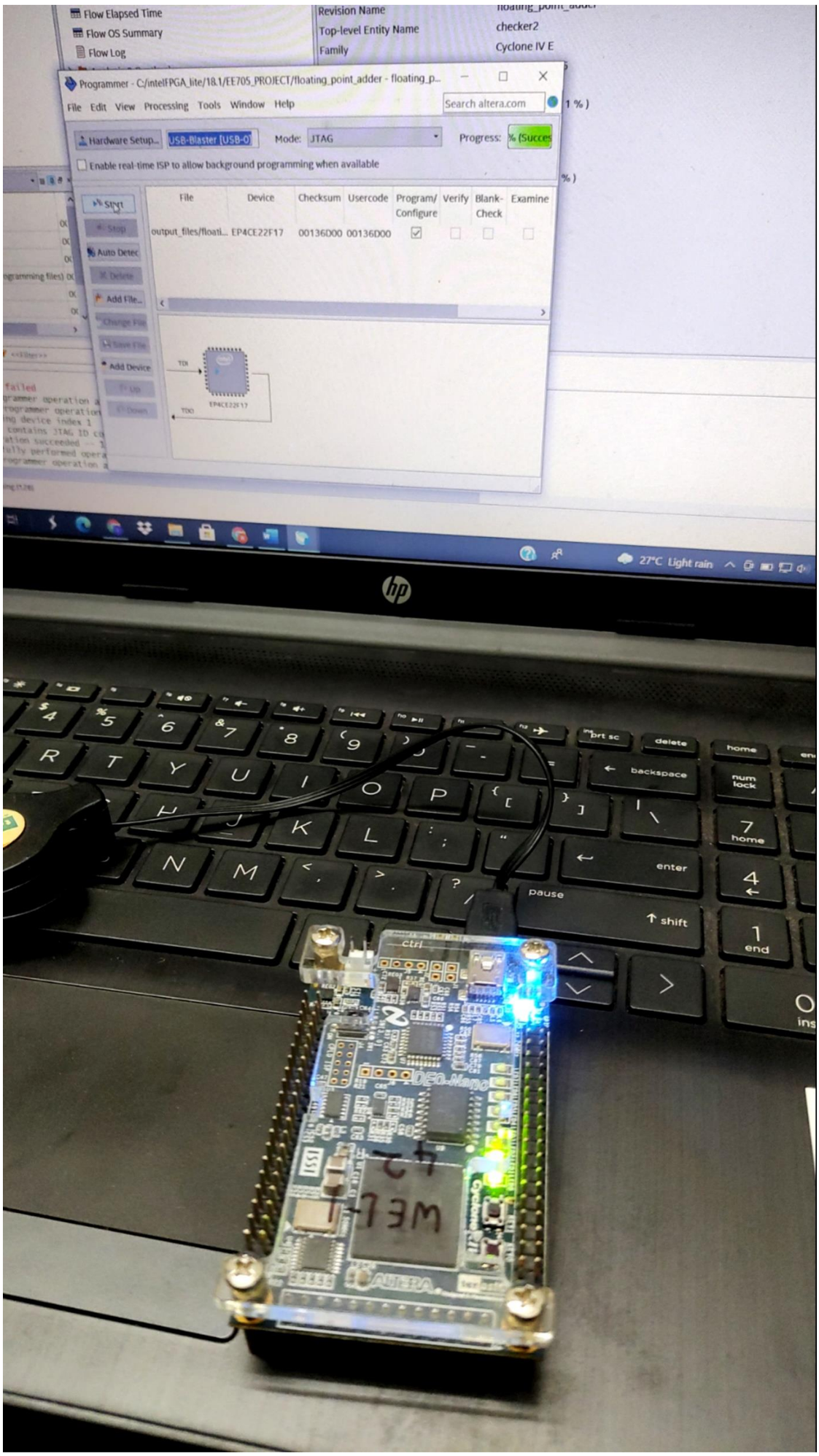

Fig 6. Loading the code on De0-nano board.

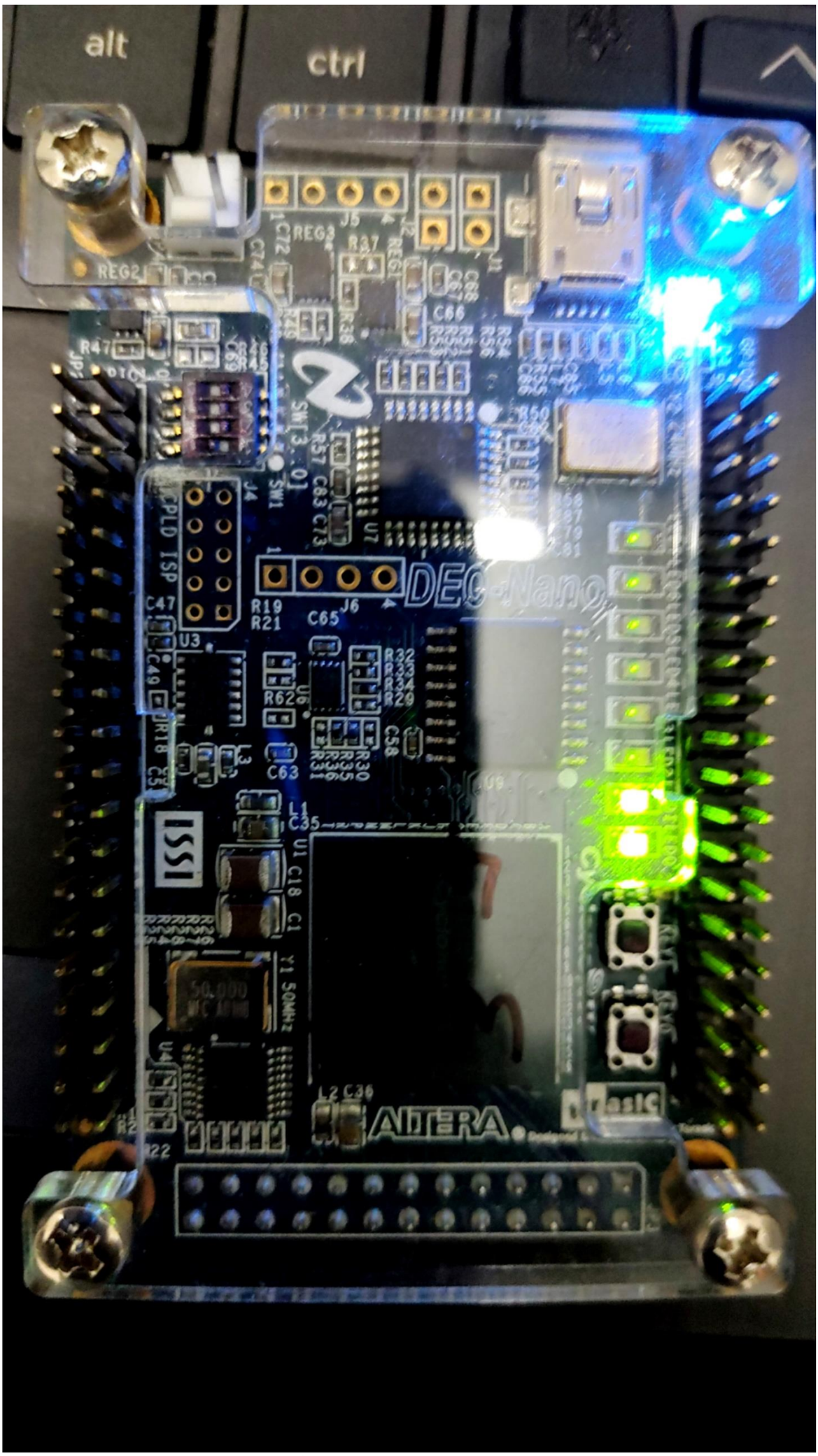

Fig 7. Two LEDs flashing representing Done and correct signals.

Fig 8. Loading modified code on the board.

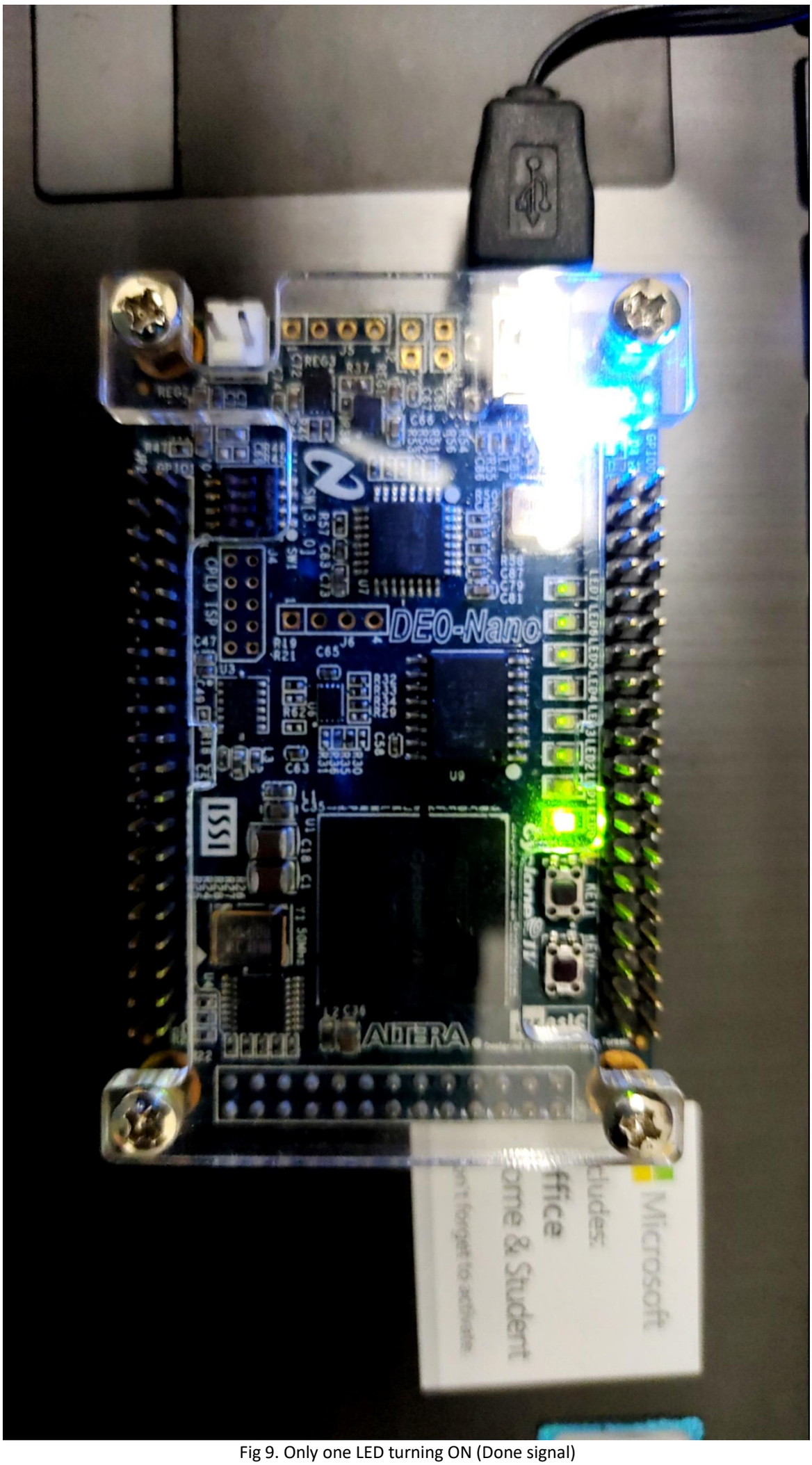

Fig 9. Only one LED turning ON (Done signal)