Lab2 Hardware-friendly Float Point Multiplier

贾梓越 22307130445

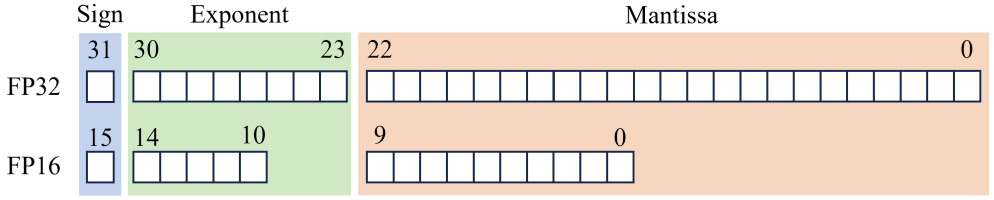
1 Overview

In this lab, I designed a float point multiplier that reuses a 8-bit integer multiplier to complete the multiply operation, saving much hardware overhead. The calculation precision is also configurable, supporting FP16 and FP32. Special cases are also taken into account, including infinite, zero, NaN and subnormal number for inputs, overflow and error mode for output. The result is displayed using two input pads and three 8-number seven-segment display modules. Each number represents a hexadecimal number, thus an 8-number module represents a 32 bits input or output. Considering the complexity of transferring float point number to decimal number, only the hexadecimal numbers are displayed.

2 Principle

2.1 Float Point Number Format

The module supports two precisions, FP16 and FP32. According to IEEE754, the formats of two data types are illustrated in the graph below.



The first bit is sign bit, 0 represents positive number and 1 represents negative. The next part is exponent number. For FP32, bit 23 to 30 is the exponent bit while for FP16 it's bit 14 to 10. The number of the exponent bits needs to subtract an offset number and then calculated as the exponent of 2. The last part is the mantissa part. The mantissa is considered as the fractional part of a binary fixed point number. In most cases, the integer part of the number is 1. For example, if the mantissa part of FP16 is `10'b10000_00000`, the corresponding binary fixed point number is `1.1000000000` in binary form, which is 1.5.

There are codes reserved for special numbers in the FP standard. They are listed in the chart below.

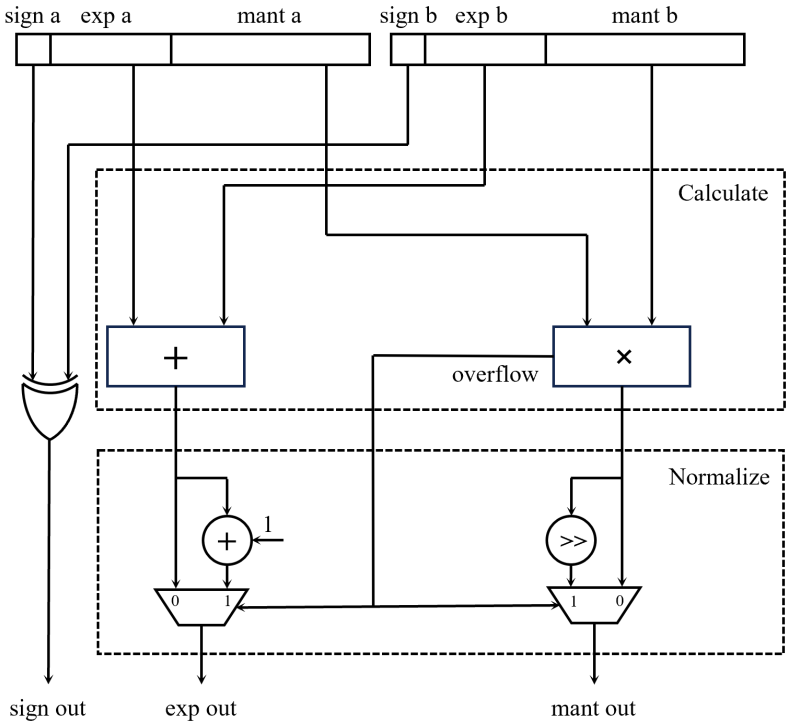
Data type	exponent	mantissa
zero	0	0
Inf	All 1	0
NaN	All 1	None zero
subnorm	0	None zero

For subnormal numbers, the integer part of mantissa is 0. The special standard enables FP32/16 to represent numbers smaller than $1 \times 2^{1-offset}$, which is the smallest positive number that can be represented by normal FP32/16 standard.

In summary, the float point is calculated with $(-1)^{Sign} \times 2^{Exp-offset} \times 1.Mantissa$ or $(-1)^{Sign} \times 2^{-offset} \times 0.Mantissa$ for subnormal number. For FP16, offset is 15, for FP32, offset is 127. With offset, the range of exponent number is $[-126, 127]$, allowing more precision for small numbers.

2.2 Multiplication Process

The multiplication can be divided into three parts: first add the two exponents, then multiply the two mantissas, finally decide whether to adjust exponent and mantissa if overflow happens in the multiplication. The overall structure chart is as follows:

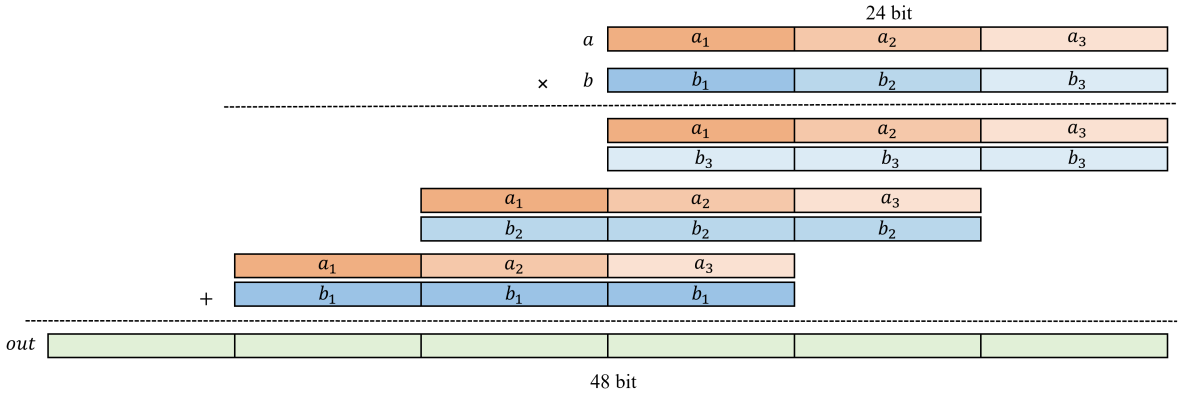


2.2.1 Exponent Sum

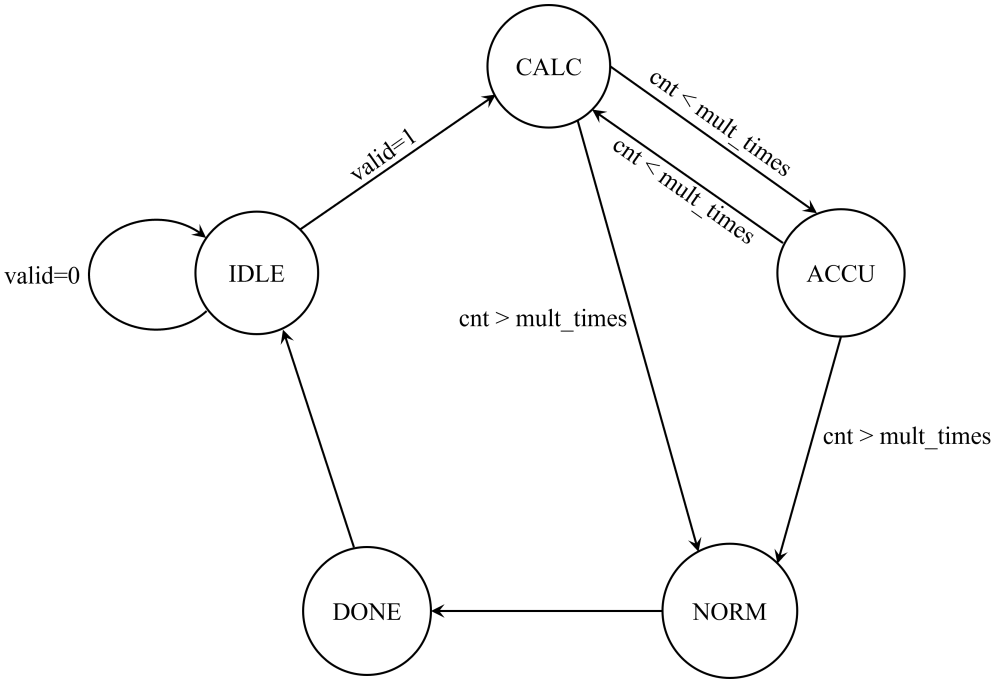
For the multiplication of two float point numbers, their exponent parts need to be added. If the result is out of the range represented by exponent part, then an overflow or underflow signal is generated.

2.2.2 Segmented Mantissa Multiplication

The two mantissa parts with the implicit 1 at the highest bit are multiplied. Then we'll take the higher 23 or 10 bits as result. But a 24 bit multiply module is complex and will cause long delay. So in this design an 8 bit multiplier is implemented. The 24 bit input is segmented into three parts, multiplied and accumulated to generate the final result. Calculating a 24 bit multiplication needs 9 times of 8 bit multiplication, as illustrated in the following graph.



To realize this function, a FSM is designed. There are five states: IDLE, CALC, ACCU, NORM, DONE. The state transition diagram is as follows.



When no data comes, the FSM stays at IDLE state. When data comes, `valid=1`, FSM turns to CALC state, a combinational circuit is implemented to generate the 8 bit multiplication result, then the counter add 1. While counter is less than the calculate times (9 times for 24bit, 4 times for 11 bit), FSM goes to ACCU state, where the result is shifted and accumulated. Then FSM goes back to CALC state to calculate the next segment. After all segments are calculated, FSM goes to NORM state to normalize the result if there are carry in the result. Finally FSM turns to DONE state. A finish signal is given and a result is given. Then FSM returns to IDLE state, waiting for the next data to come.

2.2.3 Exponent Update & mantissa normalization

For normal numbers, the range of mantissa is [1,2). The product of two mantissas range from [1,4). So it is necessary to normalize the product when it's mantissa is supposed to represent a number larger than 2. The method is to add 1 to exponent part, and right shift the mantissa.

2.3 Special cases

2.3.1 Denorm problems at input & output

Denorm * Denorm

The output exponent is $-126 - 126 = -252$, which is less than the smallest exponent, causing underflow.

Denorm * Norm

One exponent is 8'b0, which is seen as -126 when calculating the float point number in decimal, but when calculating the sum of two exponents, it is actually -127 ($0 - offset$), another exponent ranges from -126 to 127. So the calculated exponent sum ranges between $[-252, 0]$, actual value ranges between $[1, 254]$, the mantissa product ranges between $[0, 2)$.

The result can be underflow, denorm or norm.

1. Underflow: Happens when exponent sum of two exponents is less than -126. This circumstance can be detected by referring to the carry bit, which is sum[8] of the exponent sum, if sum[8] is 1, it indicates that either overflow or underflow happened during the exponent adding. Both of the circumstances are called overflow(`output overflow`) in this module.
2. Denorm: Happens when exponent sum is -126 and the product of two mantissas is in between $[0, 1)$. In this circumstance, the -126 is calculated by adding 0 and 127 then minus 127, so the output is 0, satisfying the denorm standard.
3. Norm: Happens when exponent sum is -126 and the mantissa is in between $[1, 2)$, or exponent sum is bigger than -126. For the second case no special operation is needed. For the first case, the calculated -126 is the sum of 8'b0000_0000 and 8'b0111_1111 minus 8'b0111_1111 (127), resulting 8'b0000_0000, which is the -126 for denorm number. It needs to be transferred to -126 for normal number. When one input is denorm, we need to check the second highest bit of the mantissa product, if it is 1, meaning mantissa product is bigger than 1, then a `denorm_normalize` signal is set to 1, and the exponent part is set to 1 (-126 for normal number).

Norm * Norm

The exponent sum ranges between $[-252, 254]$, the mantissa product ranges between $[1, 4)$.

The result can be underflow, denorm, norm or overflow.

1. Underflow: Happens hen exponent sum is less than -128. This can be achieved by checking the carry bit of exponent sum.
2. Denorm: Happens when exponent sum is -127 (8'b0000_0000) and mantissa product is in between $[1, 2)$. We need to right shift the mantissa product but do not add 1 to exponent sum as the sum is already 0, automatically satisfying the denorm standard.
3. Norm: This is the most common case, happening when exponent sum is in between $[-126, 126]$ or exponent sum is -127 and mantissa product is in between $[2, 4)$. The later case also doesn't need special operation.
4. Overflow: Happens when exponent sum is more than 126 or is 126 while mantissa product is in between $[2, 4)$. This can be achieved by checking the carry bit of exponent sum.

Now, for different circumstances, the way to determine the correct case is by checking the exponent sum and the mantissa product. In most cases no special operation is needed.

2.3.2 Error, Overflow and Underflow

In this design, overflow and underflow is not distinguished. Both are generated when the exponent carry bit is 1, meaning it's overflow caused by adding two positive exponents or underflow caused by adding two negative exponents.

There are some special rules. If one input is Inf, then the output is also Inf. If one input is NaN, then the output is also NaN, and the error signal is active.

3 Module Description

3.1 top

The main function of top module is to transfer the output signals from the FP multiply module to display signals for Rabbit components. To save IO resources, the design uses 7-segment display and a select signal to display eight numbers. The total IO is 69. The ports and descriptions are listed below.

Port name	Type	Width	Description
clk	input	1	Clock
rst_n	input	1	Reset, 0 active
data_type	input	1	Input data type, 1 means FP32, 0 means FP16
enter	input	1	Start calculation
ce	input	1	Clear all inputs
in1_row	input	4	Row signal of the first input, connected to the 4*4 input pad in Rabbit
in1_col	input	4	Collum signal of the first input, connected to the 4*4 input pad in Rabbit
in2_row	input	4	Row signal of the second input, connected to the 4*4 input pad in Rabbit
in2_col	input	4	Collum signal of the second input, connected to the 4*4 input pad in Rabbit
disp1_seg	output	7	Segment signal of first input, connected to 7-segment display component
disp1_sel	output	8	Segment select signal of first input
disp2_seg	output	7	Segment signal of second input
disp2_sel	output	8	Segment select signal of second input
out_seg	output	7	Segment signal of output
out_sel	output	8	Segment select signal of output
ready	output	1	Set 1 when the calculation is done
overflow	output	1	Overflow happens
error	output	1	Error happens

3.2 float_mult

The main module of float point multiplication. Incorporates add8 and mult24. The ports and descriptions are listed below.

Port name	Type	Width	Description
clk	input	1	Clock
rst_n	input	1	Reset, 0 active
data_type	input	1	Input data type, 1 means FP32, 0 means FP16
valid	input	1	New data comes, next clock calculation starts
in1_32	input	32	Input1 when mode is FP32
in2_32	input	32	Input2 when mode is FP32
in1_16	input	16	Input1 when mode is FP16
in2_16	input	16	Input2 when mode is FP16
out_32	input	32	FP32 output, 0 when mode is FP16
out_16	output	16	FP16 output, 0 when mode is FP32
ready	output	1	Set 1 when the calculation is done
overflow	output	1	Overflow happens
error	output	1	Error happens
in1_disp	output	32	Signal for input display
in2_disp	output	32	Signal for input display

In this module, inputs are first preprocessed to avoid unnecessary calculation. It first check whether the inputs are zero, Inf, NaN. If true, the multiply module will not be activated, the results can be generated after one clock, saving time and power.

After the preprocessing, the implicit 1 or 0 (subnormal case) is added to the mantissa is sent into mult24 module. For FP16, mantissa length is 11(with the implicit 1). So it takes 4 8-bit multiplications. For FP32, the mantissa length is 24 (with the implicit 1). So it takes 9 8-bit multiplications.

For FP32, the exponents are directly sent into add8. For FP16, the exponent is 5-bit, so the high 3-bit is filled with zeros. Because the exponent result is:

$$(exp_1 - offset) + (exp_2 - offset) + offset = exp_1 + exp_2 - offset$$

So the sum of exponents needs to minus offset to get the final result.

The sign bit is the exclusive OR of two input signs.

After mut24 finished calculating, the results are put together as the final result, and the ready signal is set to 1, the calculation is finished.

3.3 add8

The longest exponent is 8 bit in FP32, the exponent sum employs a 8 bit adder. For FP16, the high 3 bit of exponent part is filled with zero before sent into the adder. The adder also outputs a carry bit, indicating overflow of the exponent part. The whole logic is combinational. The ports are listed below.

Port name	Type	Width	Description
In1	input	8	Input1
In2	input	8	Input2
sum	output	9	Output with carry bit

3.4 mult_24

This module inputs two 24-bit numbers, and output the 23-bit product without the implicit highest bit. The `normalize` and `more_than_1` signals are introduced in 2.3.1. The `more_than_1` is used to generate the `denorm_normalize` signal in 2.3.1. The ports are listed below.

Port name	Type	Width	Description
clk	input	1	Clock
rst_n	input	1	Reset, 0 active
valid	input	1	New data comes, next clock calculation starts
mode	input	1	Input data type, deciding which bit indicates normalization
in1	input	24	Input1
in2	input	24	Input2
product	output	23	Product of two inputs, without the implicit 1 or 0
normalize	output	1	Product is larger than 2, exponent needs to add 1
more_than_1	output	1	Product is more than 1 while exp is 0, transfer subnormal to normal number by setting exp as 9'b1(-126)
ready	output	1	1 when calculation is finished

3.5 mult_8

This is a combinational logic module, realizing the multiplication of two 8-bit numbers. The ports are listed below.

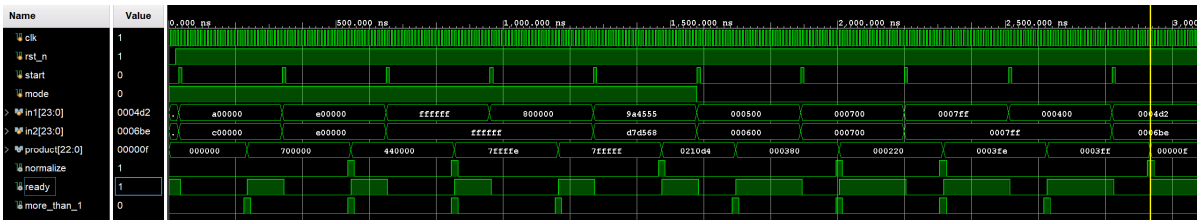
Port name	Type	Width	Description
In1	input	8	Input1
In2	input	8	Input2
product	output	16	Complete product

4 Results

The project is tested on Vivado, the testbench files are in folder `lab2_testbench`. The tests are run after synthesis to guarantee the correctness on FPGA implementation.

4.1 mult24 Test Results

Two input modes are tested. The test cases are 1.25*1.5 (without normalization), 1.75*1.75 (with normalization), max*max, zero case and a random case. According to the waveform, the results are correct.



4.2 float_mult Test Results

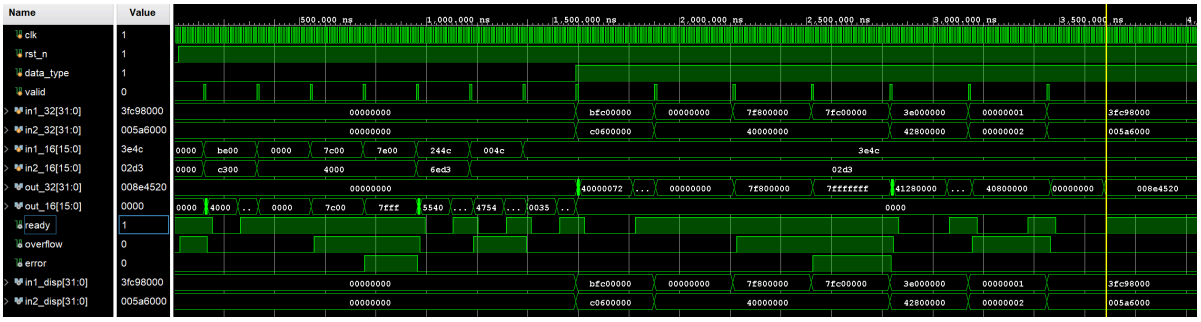
Two types of input are tested. The test cases include:

```
Test case 1: Normal multiplication (negative)
Test case 2: Zero multiplication
Test case 3: Infinity multiplication
Test case 4: NaN multiplication
Test case 5: Random multiplication
Test case 6: Random multiplication (denorm number underflow)
Test case 7: Random multiplication (denorm number normalize)
```

The TCL output is as follows:

```
Test case 1):    in1 = 1011111000000000, in2 = 1100001100000000, out_16 = 0100010101000000(ref: 0100010101000000), overflow = 0, error = 0 (FP16 Normal multiplication (negative)
Test case 2):    in1 = 0000000000000000, in2 = 0100000000000000, out_16 = 0000000000000000(ref: 0000000000000000), overflow = 0, error = 0 (FP16 Zero multiplication)
Test case 3:     in1 = 0111110000000000, in2 = 0100000000000000, out_16 = 0111110000000000(ref: 0111110000000000), overflow = 1, error = 0 (FP16 Infinity multiplication)
Test case 4:     in1 = 0111111000000000, in2 = 0100000000000000, out_16 = 0111111111111111(ref: 0111111111111111), overflow = 1, error = 1 (FP16 NaN multiplication)
Test case 5:     in1 = 0010010001001100, in2 = 0110111011010011, out_16 = 0101011101010100(ref: 0101011101010100), overflow = 0, error = 0 (FP16 Random multiplication)
Test case 6:     in1 = 0000000001001100, in2 = 0000001011010011, out_16 = 0100010000110101(ref: ***underflow****), overflow = 1, error = 0 (FP16 Random multiplication (denorm number underflow))
Test case 7:     in1 = 0011111001001100, in2 = 0000001011010011, out_16 = 0000010001110010(ref: 0000010001110010), overflow = 0, error = 0 (FP16 Random multiplication (denorm number normalize))
Test case 8:     in1 = 10111111111000000000000000000000, in2 = 11000000110000000000000000000000, out_32 = 01000000101010000000000000000000(ref: 01000000010100000000000000000000), overflow = 0, error = 0 (FP32 Normal multiplicati
Test case 9:     in1 = 00000000000000000000000000000000, in2 = 01000000000000000000000000000000, out_32 = 00000000000000000000000000000000(ref: 00000000000000000000000000000000), overflow = 0, error = 0 (FP32 Zero multiplication
Test case 10:    in1 = 01111111100000000000000000000000, in2 = 01000000000000000000000000000000, out_32 = 01111111100000000000000000000000(ref: 01111111100000000000000000000000), overflow = 1, error = 0 (FP32 Infinity multiplica
Test case 11:    in1 = 01111111111000000000000000000000, in2 = 01000000000000000000000000000000, out_32 = 01111111111111111111111111111111(ref: 01111111111111111111111111111111), overflow = 1, error = 1 (FP32 NaN multiplication)
Test case 12:    in1 = 00111110000000000000000000000000, in2 = 01000010100000000000000000000000, out_32 = 01000001000000000000000000000000(ref: 01000001000000000000000000000000), overflow = 0, error = 0 (FP32 Random multiplicati
Test case 13:    in1 = 00000000000000000000000000000001, in2 = 00000000000000000000000000000010, out_32 = 01000000100000000000000000000000(ref: **********underflow**********), overflow = 1, error = 0 (FP32 Random multiplicati
Test case 14:    in1 = 00111111111001001100000000000000, in2 = 00000000010110100110000000000000, out_32 = 00000000100011100100010100100000(ref: 00000000100011100100010100100000), overflow = 0, error = 0 (FP32 Random multiplicati
$stop called at time : 3760 ns : File "D:/Vivado Workplace/float_mult/float_mult.srcs/sim_1/new/tb_float_mult.v" Line 241
relaunch_sim: Time (s): cpu = 00:00:01 ; elapsed = 00:00:07 . Memory (MB): peak = 3237.156 ; gain = 0.000
```

The waveform is as follows.

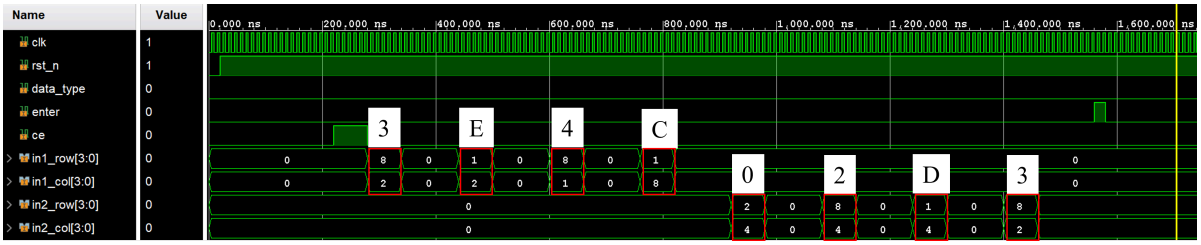


4.3 top Test Results

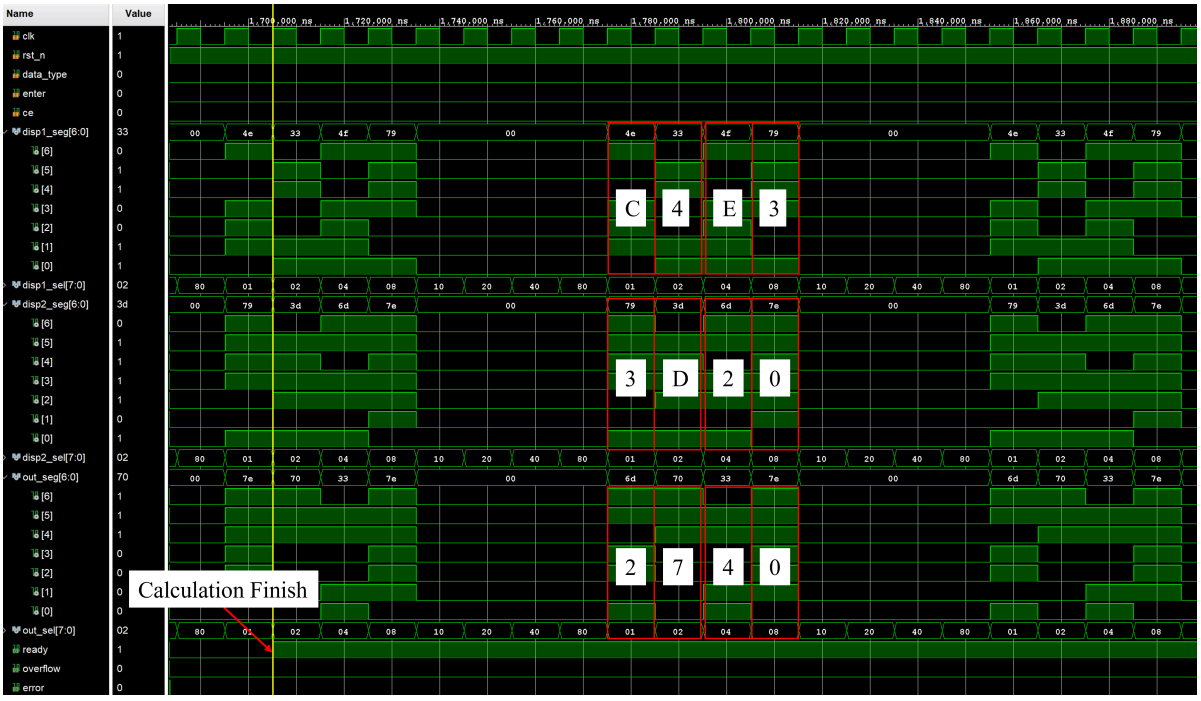
As the input process is long, the test of top module mainly test the correctness of display. Only two cases are tested:

```
Test FP16 mode:
16'b0_01111_1001001100 * 16'b0_00000_1011010011 = 3E4C * 02D3 =
0472
Test FP32 mode:
32'b0011_1111_1100_1001_1000_0000_0000_0000 *
32'b0000_0000_0101_1010_0110_0000_0000_0000 = 3FC98000 * 005A6000
= 008E4520
```

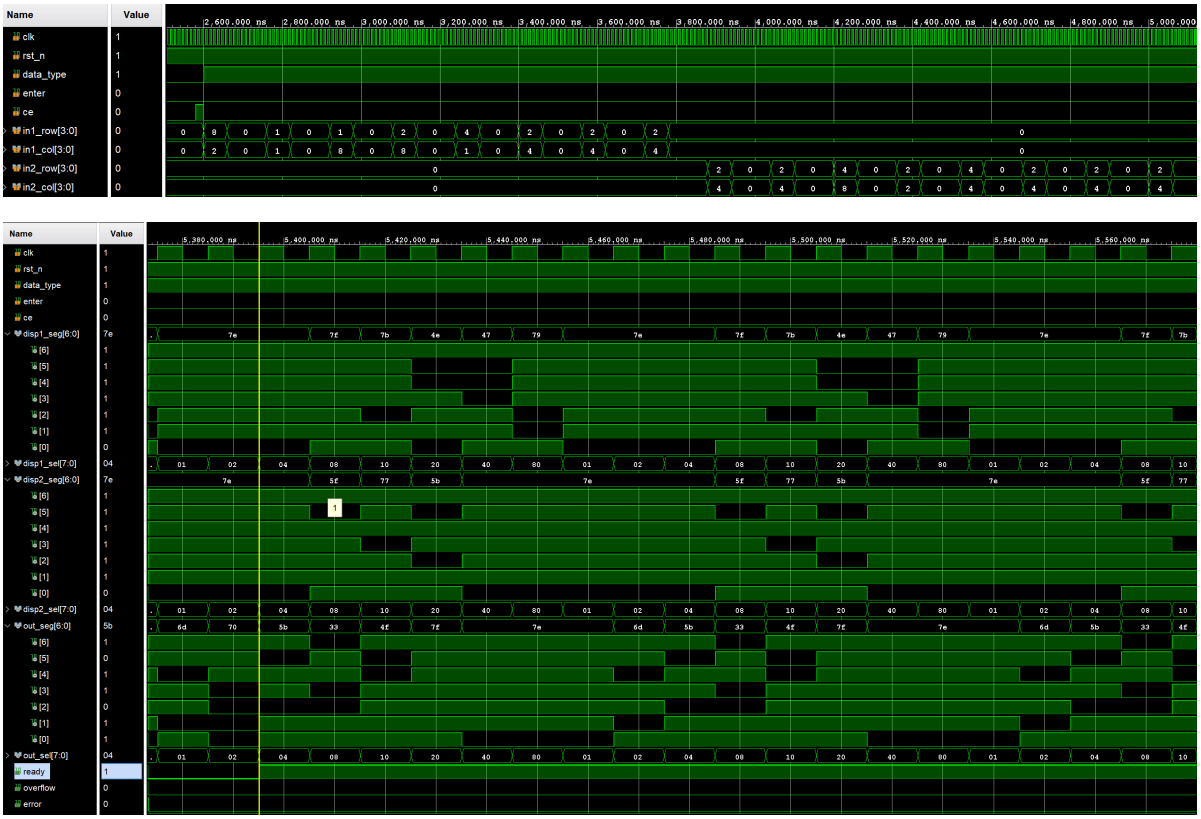
The input of first case is as follows:



The output of the first case is as follows:

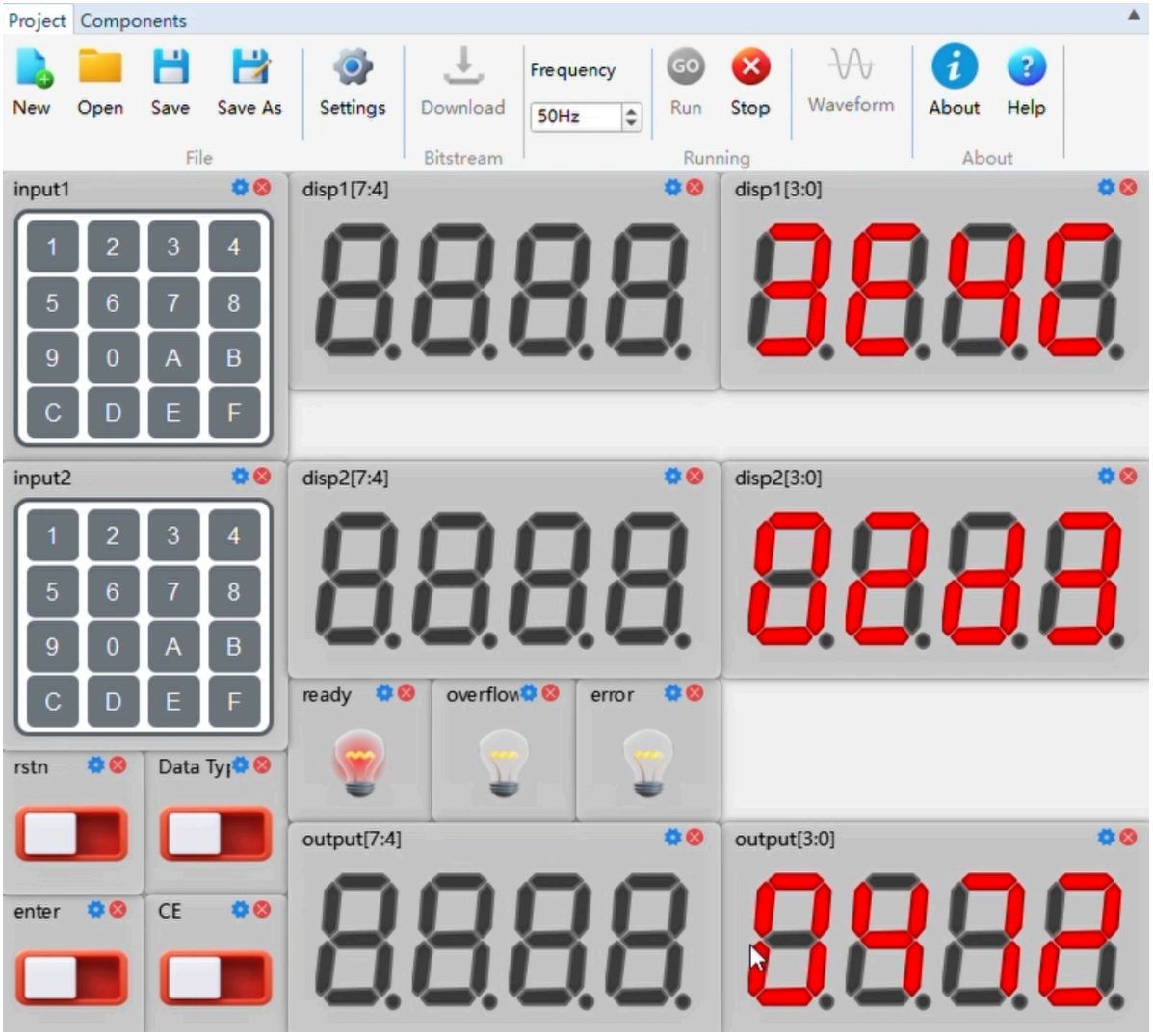


The waveform of the second test case is as follows, the display function is correct.



4.4 Rabbit Testing

The component layout is as follows. There are two 4*4 pads for input and three 8-number 7-segment display unit. The display effect is as follows.



5 Problems & Outlook

When using Design Compiler, it's worth noticing that the `source` command may not work, the specific file name should be added like `source compile.tcl`. Also, `Fatal:Design compileris notenabled.(DCSH-1)` may easily occur if the virtual machine time is synchronized. This may be solved by retrieving a previous snapshot or reinstalling the virtual machine.

The gate level file generated by DC may have false port connections, and the output segment display is mismatched. This is solved after running DC many times using the same RTL code. The possible cause is currently unknown.

Due to limited IO, eight segment numbers have to share one select signal, but the four number component in Rabbit will be off if there are no select signal, which will happen every four clocks. So the displayed numbers will flicker no matter how much visual persistence is set. So the numbers may not be so easy to read in the demo video.

For future works, the booth code will be implemented in the multiply unit to simplify the procedure. Also, other data types can be supported with the existing hardware, such as INT4, INT8 and BF16 to realize mix-precision calculation.