**Function Table**

|  |  |
| --- | --- |
| **Function** | **Meaning** |
| Addition\_V(‘c’, ‘a’, ‘b’) | c = a + b |
| Addition\_IF\_V(‘c’, ‘a’, ‘b’) | c = a + b after If\_V() |
| Addition\_Else\_V(‘c’, ‘a’, ‘b’) | c = a + b after Else\_V() |
| Subtraction\_V(‘c’, ‘a’, ‘b’) | c = a - b |
| Subtraction\_If\_V(‘c’, ‘a’, ‘b’) | c = a – b after If\_V() |
| Subtraction\_Else\_V(‘c’, ‘a’, ‘b’) | c = a – b after Else\_V() |
| Multiplication\_V(‘c’, ‘a’, ‘b’) | c = a \* b |
| Multiplication\_If\_V(‘c’, ‘a’, ‘b’) | c = a \* b after If\_V() |
| Multiplication\_Else\_V(‘c’, ‘a’, ‘b’) | c = a \* b after Else\_V() |
| Division\_V(‘c’, ‘a’, ‘b’) | c = a / b |
| Division\_If\_V(‘c’, ‘a’, ‘b’) | c = a / b after If\_V() |
| Division\_Else\_V(‘c’, ‘a’, ‘b’) | c = a / b after Else\_V() |
| Logarithm\_V(‘c’, ‘a’, ‘b’) | c = log a (b) |
| Logarithm\_If\_V(‘c’, ‘a’, ‘b’) | c = log a (b) after If\_V() |
| Logarithm\_Else\_V(‘c’, ‘a’, ‘b’) | c = log a (b) after Else\_V() |
| Power\_V(‘c’, ‘a’, ‘d’) | c = ad (d should be integer) |
| Power\_If\_V(‘c’, ‘a’, ‘d’) | c = ad (d should be integer) after If\_V() |
| Power\_Else\_V(‘c’, ‘a’, ‘d’) | c = ad (d should be integer) after Else\_V() |
| Sqrt\_V(‘c’, ‘a’) | c = sqrt(a) |
| Sqrt\_If\_V(‘c’, ‘a’) | c = sqrt(a) after If\_V() |
| Sqrt\_Else\_V(‘c’, ‘a’) | c = sqrt(a) after Else\_V() |
| SinCosTan\_V(sin\_r, cos\_r, tan\_r, a) | sin\_r = sin(a), cos\_r = cos(a), tan\_r = tan(a) |
| SinCosTan\_If\_V(sin\_r, cos\_r, tan\_r, a) | sin\_r = sin(a), cos\_r = cos(a), tan\_r = tan(a) after If\_V() |
| SinCosTan\_Else\_V(sin\_r, cos\_r, tan\_r, a) | sin\_r = sin(a), cos\_r = cos(a), tan\_r = tan(a) after Else\_V() |
| Value\_V(‘c’, ‘a’) | c = a (copy data in a to c) |
| Value\_IF\_V(‘c’, ‘a’) | c = a after If\_V() |
| Value\_Else\_V(‘c’, ‘a’) | c = a after Else\_V() |
| If\_V(‘c’, ‘a’, ‘condition’) | If(c condition a), condition could be >, >=, ==, <=, <, != |
| Else\_V(‘’) | Nothing need to be input |
| End\_IfElse\_V(‘’) | Nothing need to be input |
| number\_to\_hex(figure) | Transfer figure into 32-bit (16 bits for fractional) fixed point number |
| input\_define(‘a’) | Define a as input |
| output\_define(‘a’) | Define a as output |
| array\_define\_content(‘a’, a) | a is an array with figures inside will be stored in registers. If want to call a[i] in the future, please use array\_a\_wire\_i instead. |
|  |  |
| PyToVer("top") | After finish, generate unrolled Verilog file named “top.v” |

**What you can get:**

An unrolled Verilog code with automatic parallel/serial allocation transformed from the Python input from users following the rules and functions provided. The parallel or serial design will depend on the data dependency in a for loop. This tool could achieve at most 35 times faster than the Vivado HLS without requiring any hardware knowledge from users.

**What you should follow:**

1. If you want to put data as part of the equation eg. Division\_V(‘c’, ‘a’, 5), please use Division\_V(‘c’, ‘a’, number\_to\_hex (5)) since the datatype in this design is 32-bit fixed point number with 16 bits for the fractional part. Thus a transformation is necessarily needed.
2. All the inputs for the equations need the number\_to\_hex function if pure data is input except the Power\_V(‘c’, ‘a’, ‘d’), Power\_IF\_V(‘c’, ‘a’, ‘d’), and Power\_Else\_V(‘c’, ‘a’, ‘d’). In the power function, d should not use the number\_to\_hex function since this power does not support a fractional power factor.
3. If an array is needed and already declared using array\_define\_content(‘a’), then after that, please use array\_a\_wire\_i to call a[i].

Eg.

a = [number\_to\_hex(1), number\_to\_hex(2), number\_to\_hex(3),

number\_to\_hex(4) , number\_to\_hex(5)]

b = [number\_to\_hex(7), number\_to\_hex(8), number\_to\_hex(9),

number\_to\_hex(10) , number\_to\_hex(11)]

array\_define\_content("a", a)

array\_define\_content("b", b)

Multiplication\_V( 'temp', 'array\_a\_wire\_0', 'array\_b\_wire\_0' )

#for temp = a[0] + b[0]

1. In a for loop if you want to have a function like multiplication accumulation. Please use a new array to store the result in each step since the tool does not accept functions like result = result + temp.

Eg.

for i in range(1, 5):

wire\_a = "array\_a\_wire\_" + str(i)

wire\_b = "array\_b\_wire\_" + str(i)

wire\_x = "array\_x\_wire\_" + str(i)

wire\_x\_1 = "array\_x\_wire\_" + str(i-1)

Multiplication\_If\_V( 'temp', wire\_a, wire\_b) # temp = a[i] \* b[i]

Addition\_IF\_V( wire\_x, 'temp', wire\_x\_1) # x[i] = temp + x[i-1]

**What you cannot do at this version so far:**

1. A nested if/else:

Eg.

If(a>b):

If(c>d):

Addition\_V(‘c’, ‘a’, ‘b’)

Else:

Addition\_V(‘d’, ‘a’, ‘b’)

Instead, a serial if/else could be accepted.

1. Input name as same as output in a function:

Eg.

Power\_V(‘c’, ‘c’, ‘d’)

Addition\_V(‘c’, ‘c’, ‘b’)

Instead, a different output name should be used.