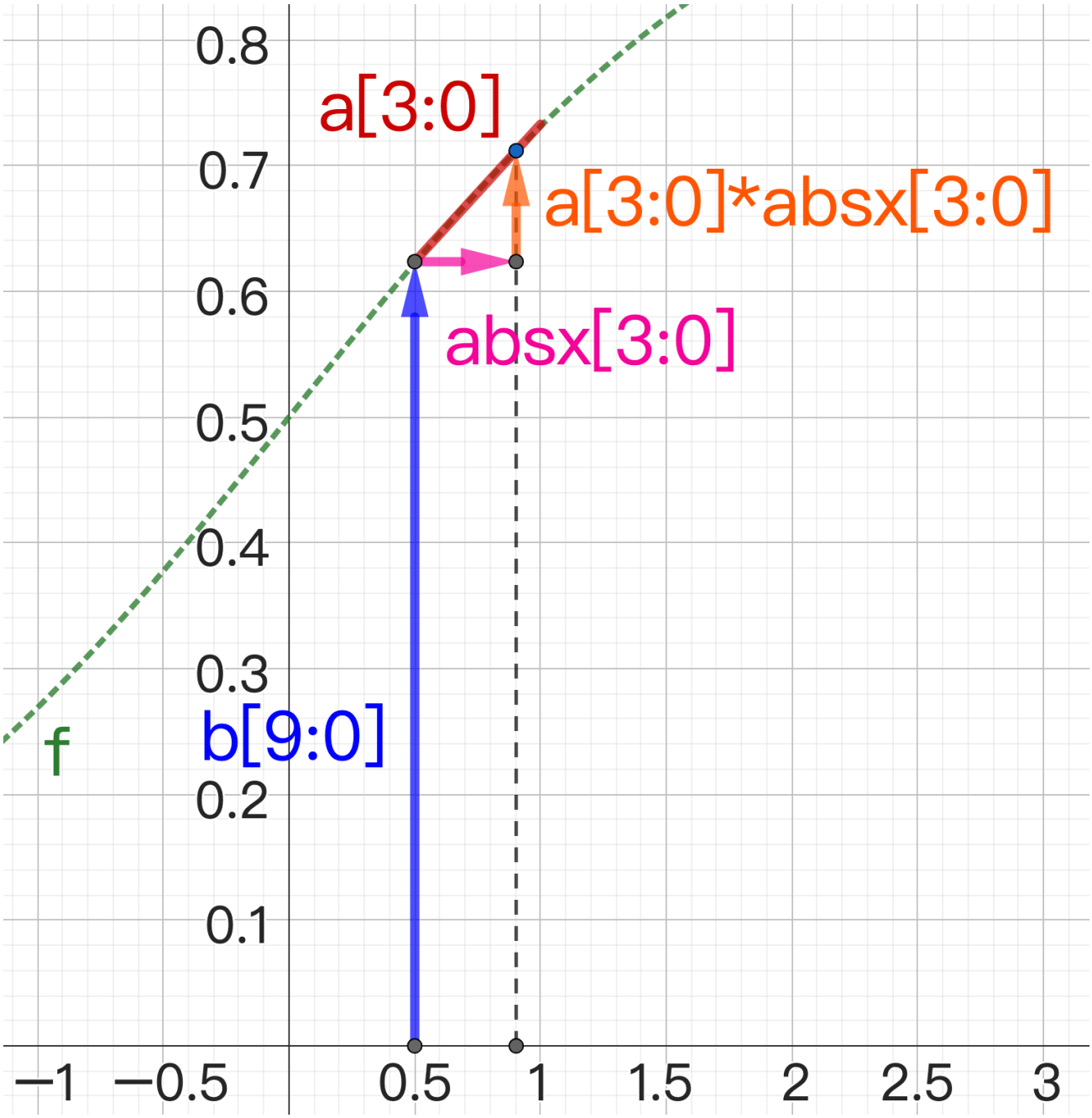
IC Design HW4

B10901176 蔡弘祥

1. Solution Idea

Since sigmoid function has a special property, which is f(x)+f(-x) =1, we only need to handle the non-negative part of the function.

1. Getting the 8-bit absolute value abs_x of the input x.
    i. The absolute value of most of the negative cases can be expressed in 8 bits.
    ii. The only case that cannot be expressed is 1000_0000, which is -4 in decimal. In this case, we will directly output the golden value. This will consume lots of resources in the circuit.
2. Deriving the 4-bit slope and the 10-bit intercept by range of abs_x
    i. Splitting [0,4] into 8 equivalent intervals, which are [0,0.5), [0.5,1), …, [3.5,4). In this case, we only need the three bits, abs_x[6:4], to determine the interval that abs_x lies on.
    ii. Each interval corresponds to a specific slope and intercept, by an 8 to 1 mux bus, we can determine the value of the 4-bit slope, a[3:0], and the 10-bit intercept, b[9:0].
    iii. We can observe that the first 4 bits, abs_x[7:4], are the same in every interval no matter abs_x is, so when doing multiplication, we only need to multiply the last 4 bits, abs_x[3:0], which are 2^{-2} to 2^{-5}.
    iv. The 4-bit slope are 2^{-3} to 2^{-6}. This is enough precise. More bits will only increase little precision, but cost much more time on calculating.
    v. The 10-bit intercept are 2^{-2} to 2^{-11}. Since 2^{0} will always be 0 and 2^{-1} will always be 1 when considering the non-negative input of the sigmoid function, so we don't have to consider it when deciding the intercept.
    vi. Checkout the graph below
3. Calculating the f(abs_x)
    i. The first step is to calculate a[3:0] * abs_x[3:0]. The result is mul[7:0] with 8 bits, which are 2^{-4} to 2^{-11}.
    ii. The next step is to calculate mul[7:0] + b[9:0]. The result is funcOut[9:0] with 10 bits, which are 2^{-2} to 2^{-11}. You might wonder why the result is 10 bits (2^{-2} to 2^{-11}) while not 11 bits (2^{-1} to 2^{-11}). The general case of an 8-bit number plus a 10-bit number is a 11-bit number. But in our case, we don't have to consider about 2^{-1} since 2^{-1} is always 1 in every case. Thus, we can use less computational resources to get the same result.
4. If x>=0, y=f(abs_x). If x<0, y=1- f(abs_x).
    i. We will first check the sign of the input, and then determine the output.
    ii. The positive cases are easy to handle, just set y[13:4] to the value we just calculate. Then set y[15:14] to 01, y[3:0] to 0011 (set y[3:0] to 0011 will be the least error case).
    iii. In general, calculating 1- f(abs_x) is hard. But if we just invert every bit of f(abs_x), we will also get the similar result since the error can be ignored (the error is 2^{-11}). After inverting, just set y[13:4] to the value. Then set y[15:14] to 00, y[3:0] to 1011 (set y[3:0] to 1011 will be the least error case).
    iv. The last consideration is the case of 1000_000, we will directly output the golden value.

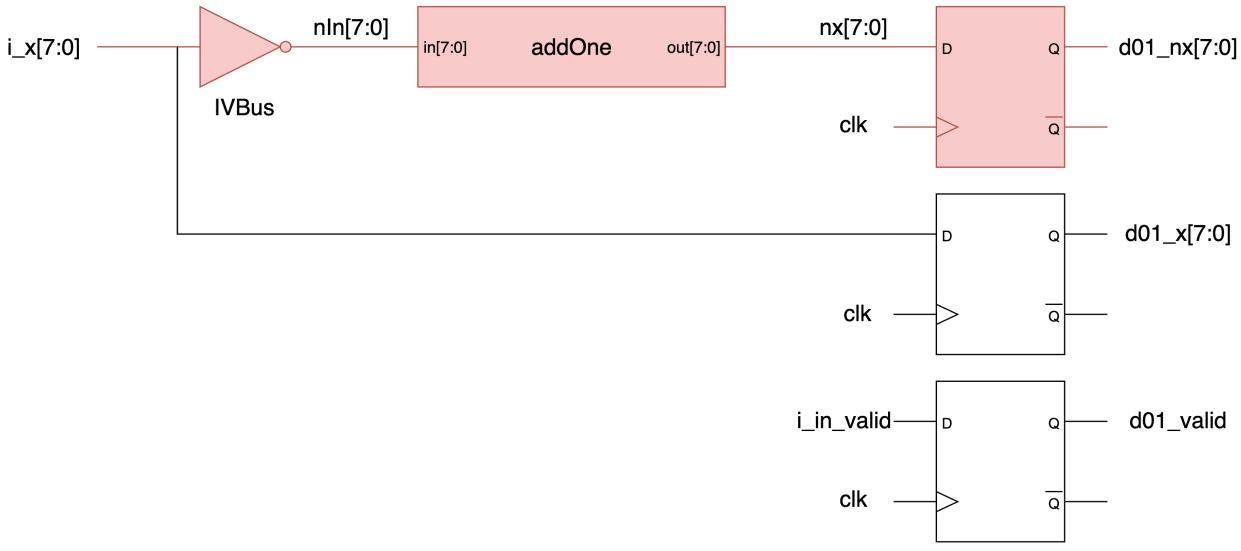


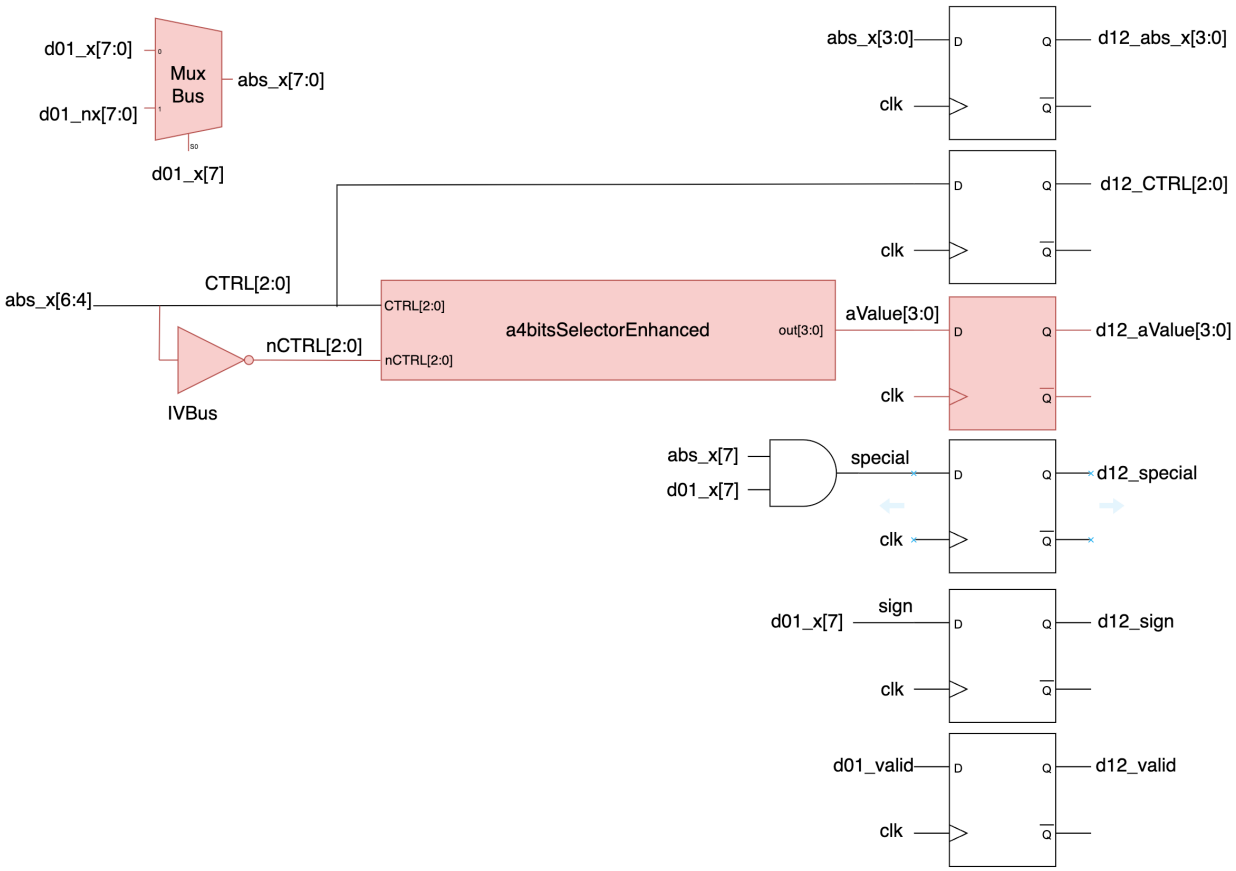
2. Pipeline Stages Overview

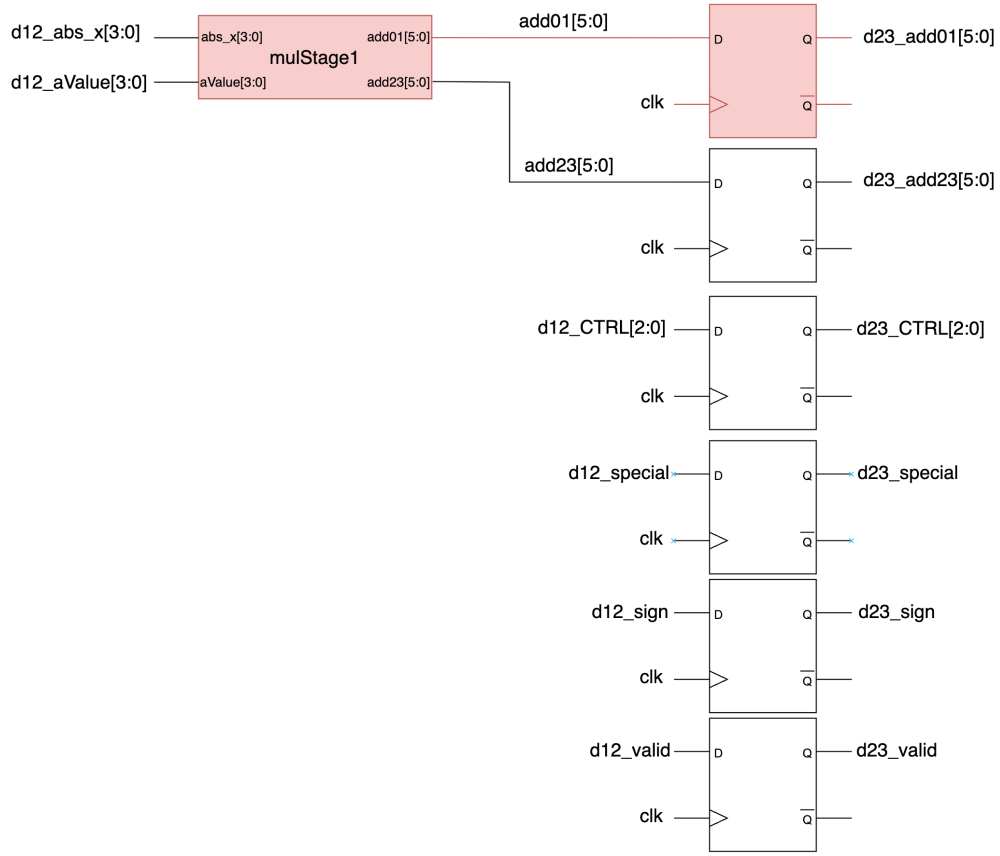
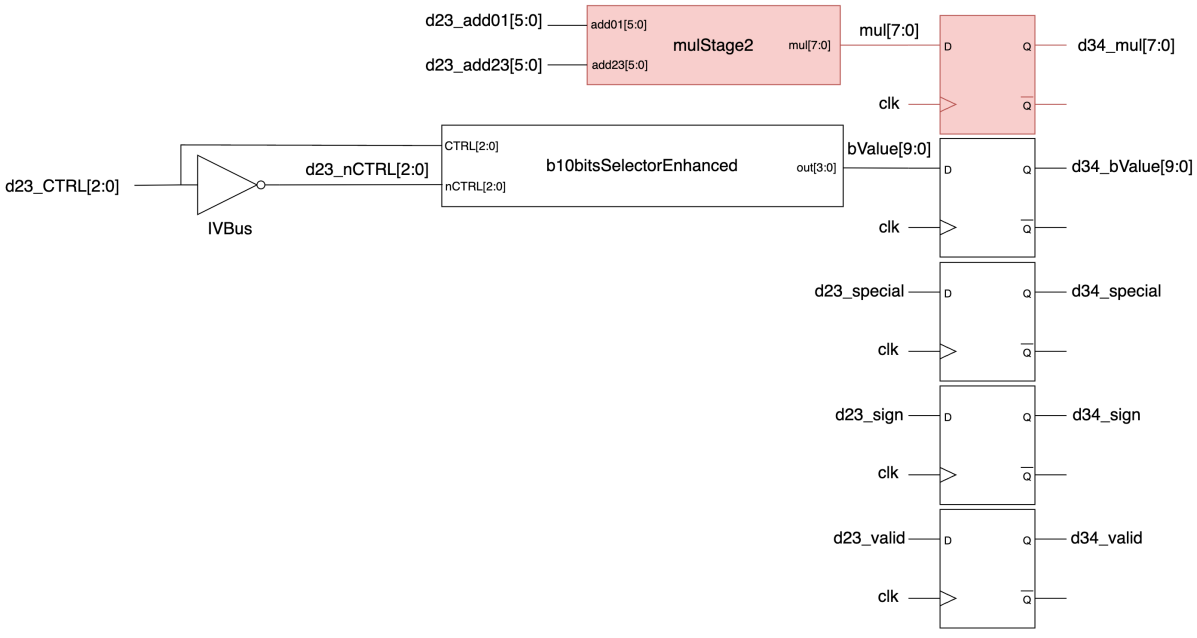
Stage	Input Signals	Output Signals	Explanation
0	i_x[7:0] i_in_valid	d01_x[7:0] d01_nx[7:0] d01_valid	2's complement handling
1	d01_x[7:0] d01_nx[7:0] d01_valid	d12_abs_x[3:0] d12_CTRL[2:0] d12_aValue[3:0] d12_special d12_sign d12_valid	Absolute value of x deriving Slope deriving

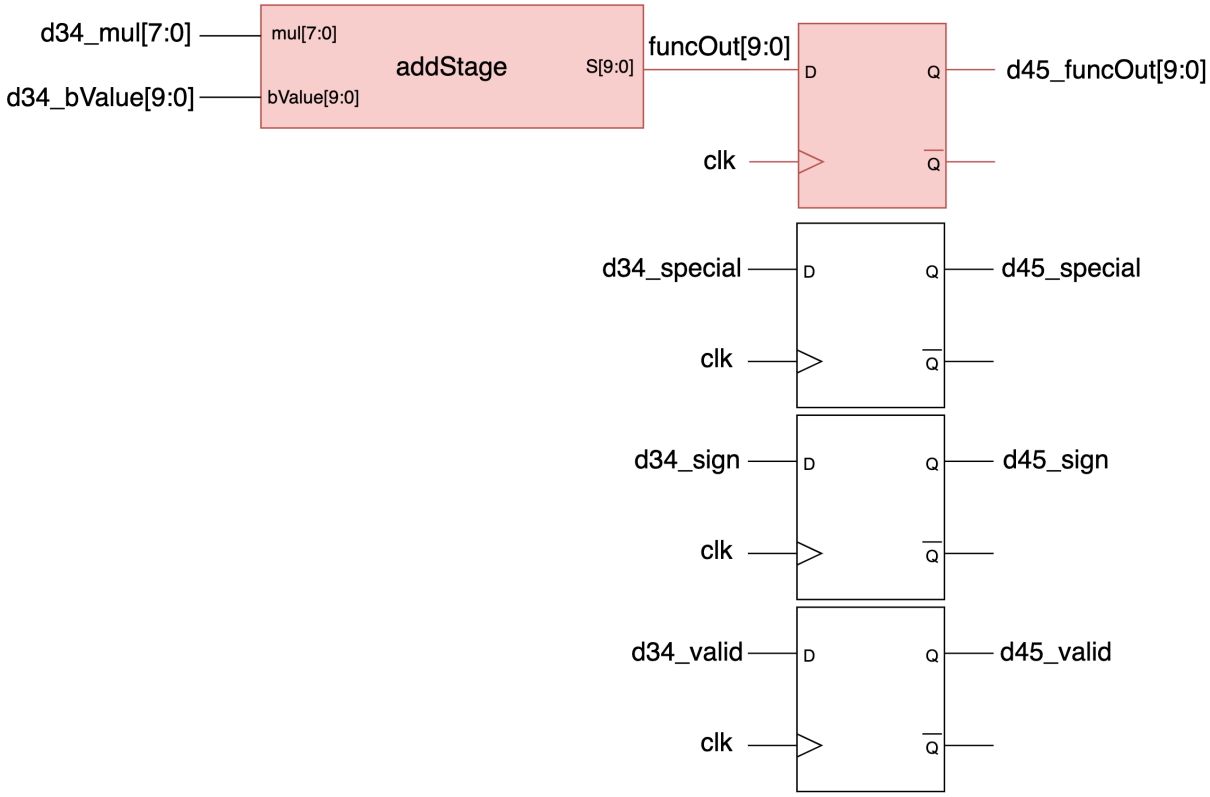
2	d12_abs_x[3:0] d12_CTRL[2:0] d12_aValue[3:0] d12_special d12_sign d12_valid	d23_add01[5:0] d23_add23[5:0] d23_CTRL[2:0] d23_special d23_sign d23_valid	Multiplication Stage 1
3	d23_add01[5:0] d23_add23[5:0] d23_CTRL[2:0] d23_special d23_sign d23_valid	d34_mul[7:0] d34_bValue[9:0] d34_special d34_sign d34_valid	Multiplication Stage 2 Intercept deriving
4	d34_mul[7:0] d34_bValue[9:0] d34_special d34_sign d34_valid	d45_funcOut[9:0] d34_special d34_sign d34_valid	Addition Stage
5	d45_funcOut[9:0] d34_special d34_sign d34_valid	o_y[15:0] o_out_valid	Output handling

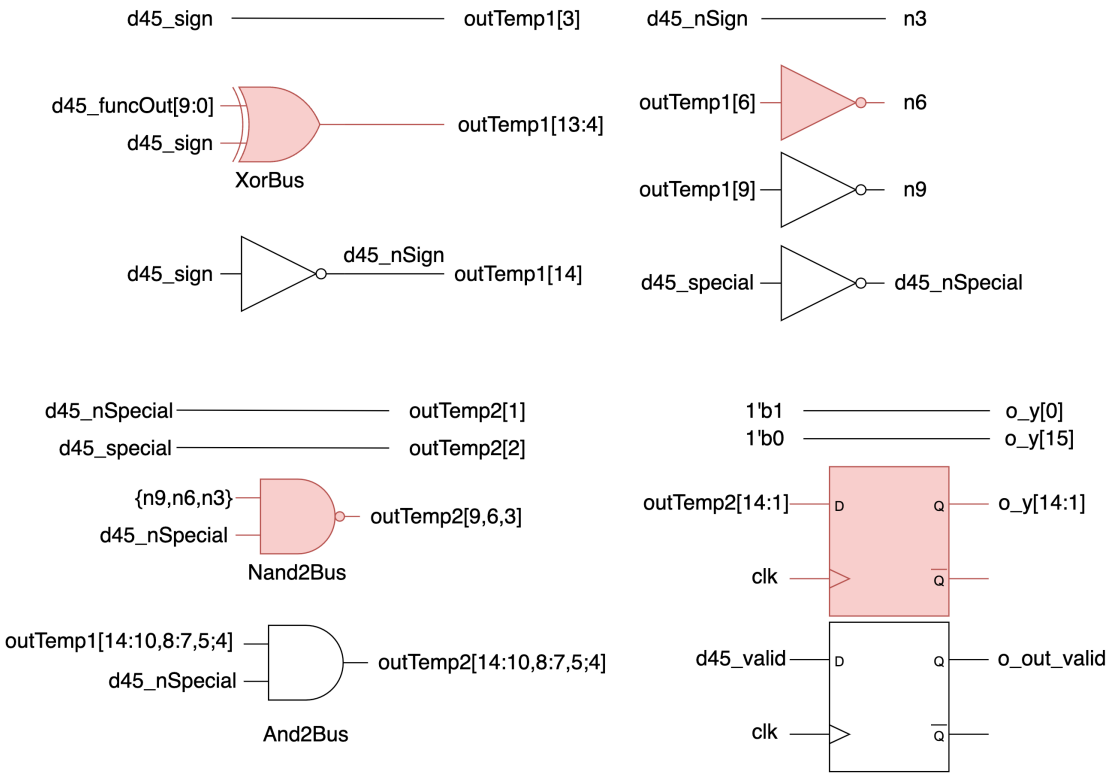
3. Each Stage Circuit Diagram and Combinational Part Explanation

<table>
<tr><th>Stage</th><th colspan="2">Circuit Diagram and Combinational Part Explanation (Red Parts Are Critical Paths)</th></tr>
<tr><td rowspan="2">0</td><td colspan="2"> 
</td></tr>
<tr><td>2’s complement handling</td><td>d01_nx[7:0] is the 2’s complement of i_x[7:0].
We will regard 1000_0000 as a special case since we cannot express -128’s 2’s complement in 8 bits.
In this special case, we will directly output its golden value.</td></tr>
</table>

1		
	Absolute value of x deriving	We can determine whether the input value is negative or not by the most significant bit of the input. To get the absolute value of the input, we only need to use a 2 to 1 mux bus.
	Slope deriving	a4bitsSelectorEnhanced is a module that can determine the value of a[3:0]. It used to be an 8-1 mux named a4bitsSelector. Since the inputs are all constants, we can simplify it to lessen the number of transistors and lower the latency.
	CTRL	CTRL signal needs to propagate until the intercept is determined.
	Special	The special case will be determined here. If it is true, then it means the input is 1000_0000.
	Sign	Sign signal needs to propagate to the last stage.

2		
	Multiplication Stage 1	Handle the first stage of multiplication. A 4-bit number times a 4-bit number can be regarded as adding 4 numbers, which is three additions. We will handle two additions in advanced and handle the last addition in the next stage.
3		
	Multiplication Stage 2	Handle the second stage of multiplication.
	Intercept deriving	b10bitsSelectorEnhanced is a module that can determine the value of b[9:0]. It used to be an 8-1 mux named b10bitsSelector. Since the inputs are all constants, we can simplify it to lessen the number of transistors and lower the latency.

<table>
<tr><td>4</td><td>


<table>
<tr><td>Addition Stage</td><td>Handle the addition of mul[7:0] and b[9:0].</td></tr>
</table>
</td></tr>
<tr><td>5</td><td>


<table>
<tr><td>outTemp1</td><td>outTemp1 is the output of the input case except (1000_0000).</td></tr>
<tr><td>outTemp2</td><td>outTemp2 is the output considering the output of 1000_0000, which is 0000_0010_0100_1101. It used to be a 2 to 1 mux, but we can simplify into some simple logic gates.</td></tr>
<tr><td>Output handling</td><td>Send outTemp2 and d45_valid into flip-flops, it will output the result in the next cycle.</td></tr>
</table>
</td></tr>
</table>

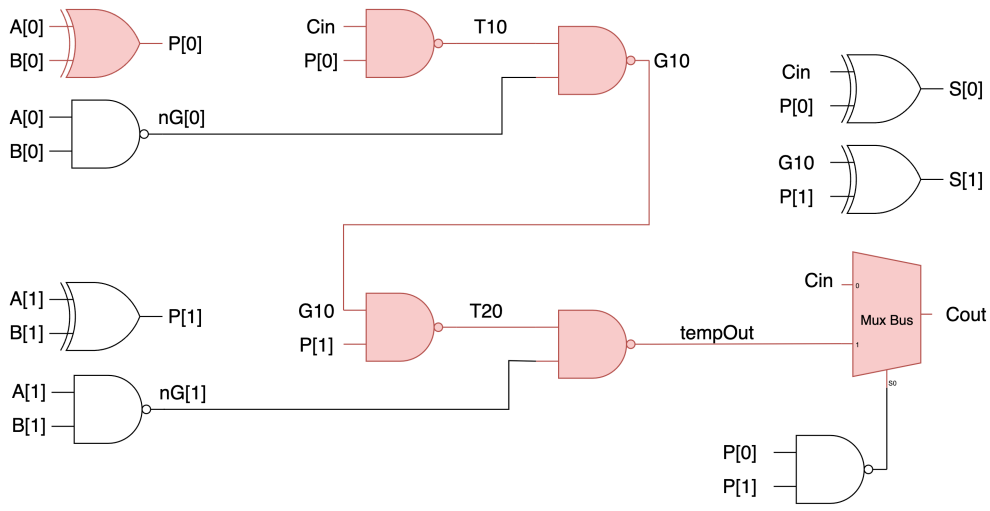
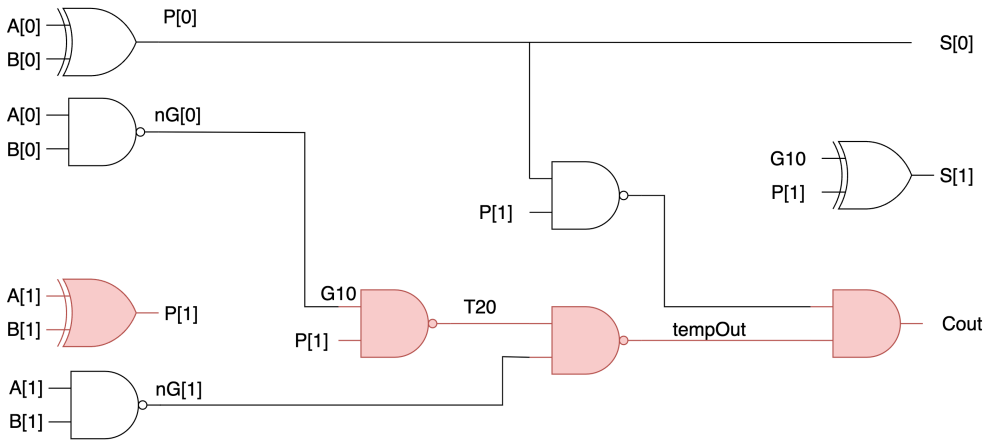
4. Self-defined Sub Modules Overview

Sub Module Name	Input signals	Output signals	Explanation
carrySkip2	A[1:0] B[1:0] Cin	S[1:0] Cout	Carry skip adder with 2-bit inputs
carrySkip2NoCin	A[1:0] B[1:0]	S[1:0] Cout	Carry skip adder with 2-bit inputs No Cin signal
carrySkip2NoB	A[1:0] Cin	S[1:0] Cout	Carry skip adder with 2-bit inputs No B[1:0] signals
carrySkip2NoBCin1	A[1:0]	S[1:0] Cout	Carry skip adder with 2-bit inputs Cin=1 No B[1:0] signals
carrySkip2NoBNoCout	A[1:0] Cin	S[1:0]	Carry skip adder with 2-bit inputs No B[1:0] signals No Cout signals
carrySkip4	A[3:0] B[3:0] Cin	S[3:0] Cout	Cascading 2 carrySkip2s
carrySkip4NoCin	A[3:0] B[3:0]	S[3:0] Cout	Cascading a carrySkip2NoCin and a carrySkip2
carrySkip8NoCin	A[7:0] B[7:0]	S[4:0] Cout	Cascading a carrySkip2NoCin and 3 carrySkip2s
addOne	in[7:0]	out[7:0]	Input a 8-bit number and add 1. Do not need to consider the case of overflowing since it won't happen.
a4bitsSelectorEnhanced	CTRL[2:0] nCTRL[2:0]	out[3:0]	Simplify the original 8 to 1 mux bus (a4bitsSelector) into simple logic gates
b10bitsSelectorEnhanced	CTRL[2:0] nCTRL[2:0]	out[9:0]	Simplify the original 8 to 1 mux bus (b10bitsSelector) into simple logic gates
mulStage1	aValue[5:0] abs_x[3:0]	add01[5:0] add23[5:0]	The first stage of multiplication, which is two additions. add01[5:0] : 2^{-6} to 2^{-11} add01[5:0] : 2^{-4} to 2^{-9}
mulStage2	add01[5:0] add23[5:0]	mul[7:0]	The second stage of multiplication, which is a addition. mul[7:0] : 2^{-4} to 2^{-11}
addStage	mul[7:0] bValue[9:0]	S[9:0]	The stage of addition S[9:0] : 2^{-2} to 2^{-11}

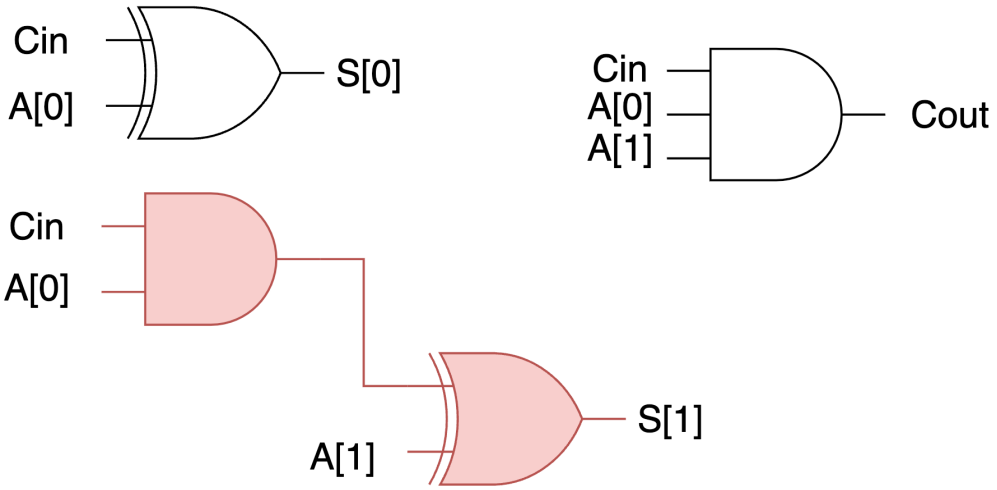
5. Slope and Intercept

x	a (2^{-3} to 2^{-6})	b (2^{-2} to 2^{-11})
[0,0.5)	1111	00_0000_0011
[0.5,1)	1110	00_1111_1101
[1,1.5)	1011	01_1101_1110
[1.5,2)	1000	10_1000_1111
[2,2.5)	0110	11_0000_1011
[2.5,3)	0100	11_0110_0100
[3,3.5)	0010	11_1010_0010
[3.5,4)	0001	11_1100_1000

6. Self-defined Sub Module Circuit Diagrams

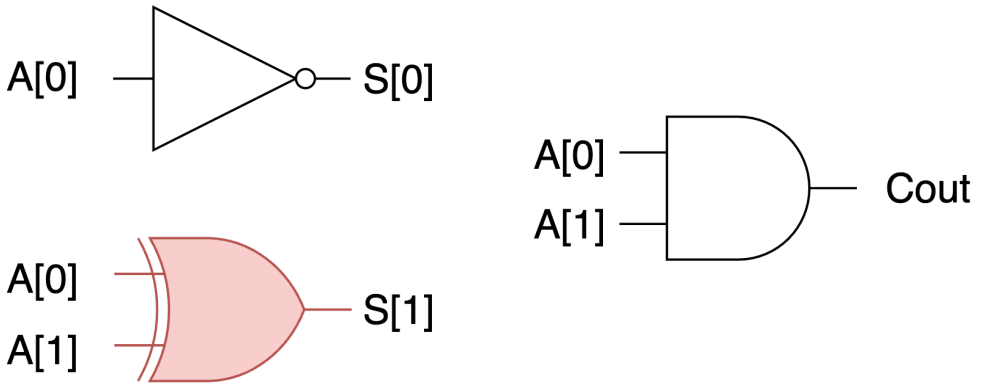
<table>
<tr><th>Sub Module Name</th><th>Circuit Diagram (Red Parts Are Critical Paths)</th></tr>
<tr><td rowspan="2">carrySkip2</td><td> 
</td></tr>
<tr><td>Simplified from CHAP8 slide</td></tr>
<tr><td rowspan="2">carrySkip2NoCin</td><td> 
</td></tr>
<tr><td>Simplified from carrySkip2</td></tr>
</table>

carrySkip2NoB



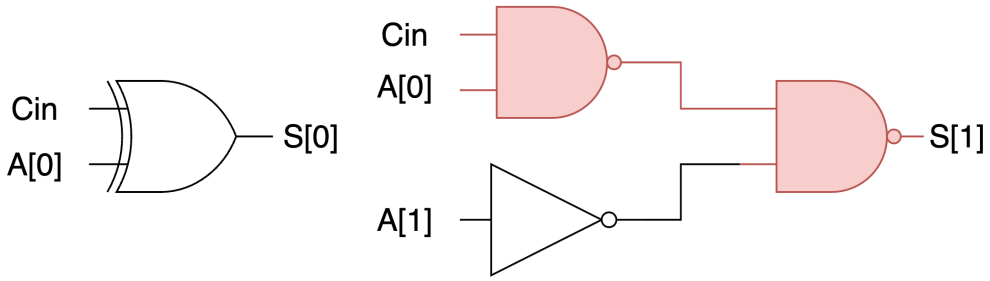
{Cout,S[1:0]}	Cin=0	Cin=1	S[0]	Cin^A[0]
A[1:0]=00	000	001		
A[1:0]=01	001	010	S[1]	(CinA[0])^A[1]
A[1:0]=11	011	100		
A[1:0]=10	010	011	Cout	CinA[0]A[1]

carrySkip2NoBCin1

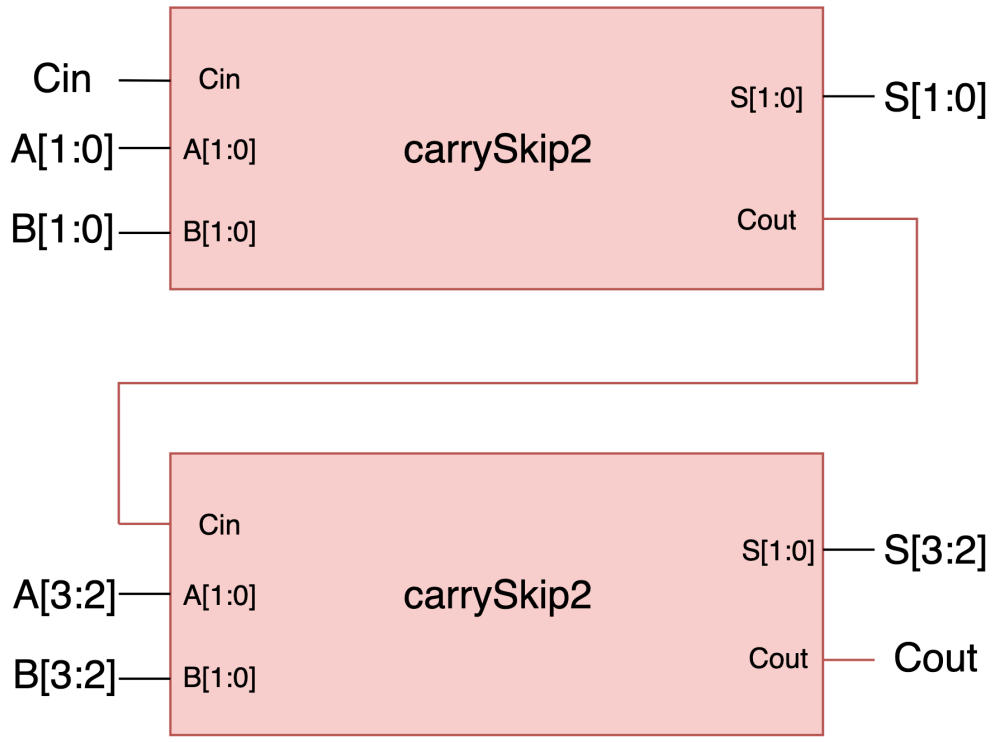
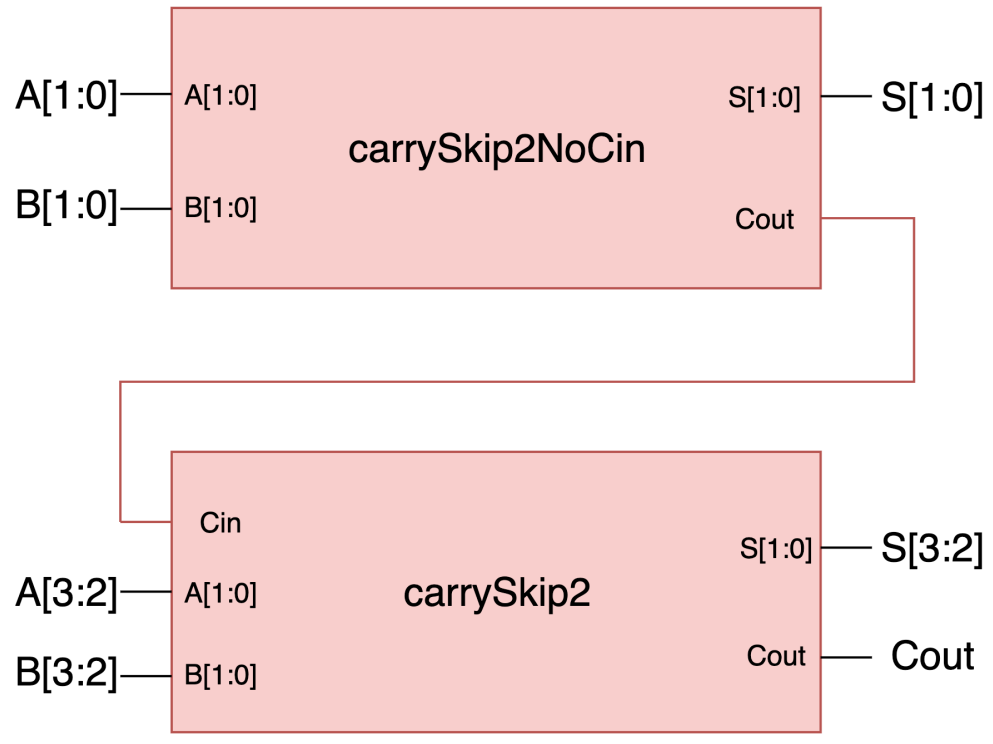


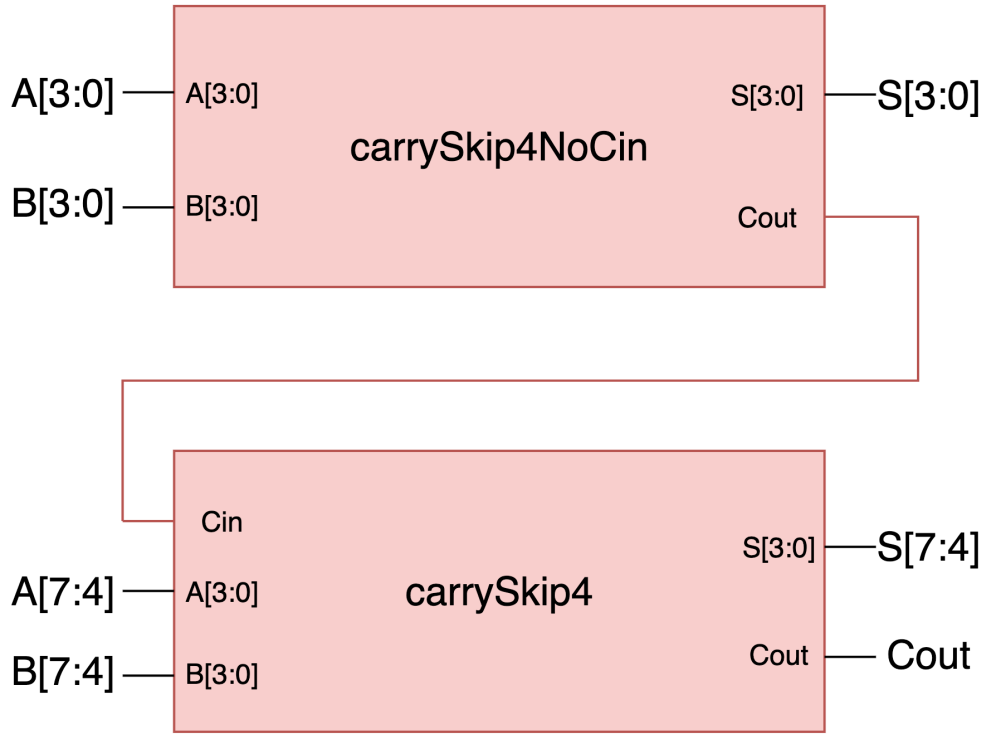
{Cout,S[1:0]}	A[1]=0	A[1]=1	S[0]	A[0]’
A[0]=0	001	011	S[1]	A[0]^A[1]
A[0]=1	010	100	Cout	A[0]A[1]

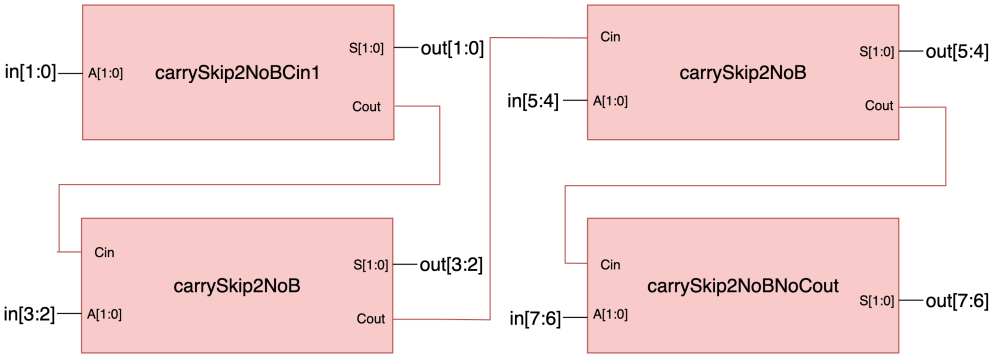
carrySkip2NoBNoCout

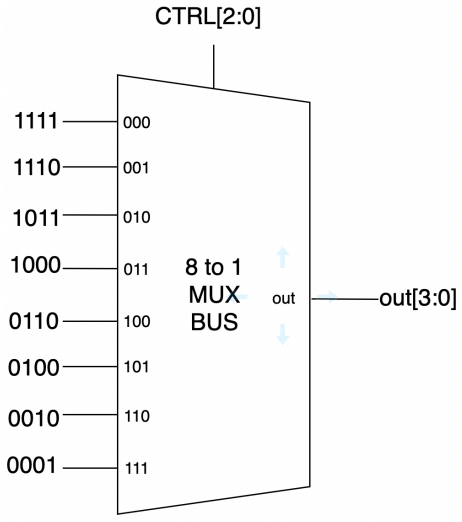


S[1:0]	Cin=0	Cin=1	S[0]	Cin^A[0]
A[1:0]=00	00	01		
A[1:0]=01	01	10	S[1]	CinA[0] + A[1]
A[1:0]=11	11	xx		
A[1:0]=10	10	11		

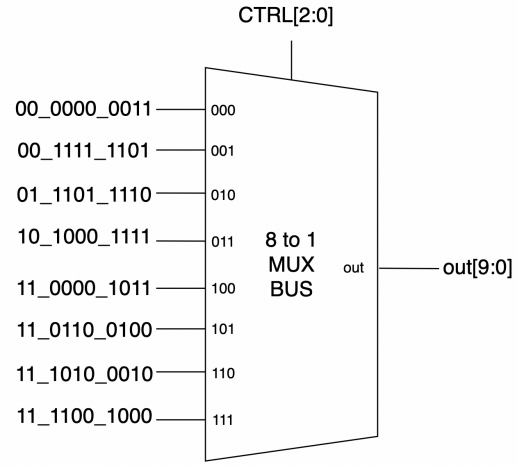
carrySkip4	 Cascading 2 carrySkip2s
carrySkip4NoCin	 Cascading a carrySkip2NoCin and a carrySkip2

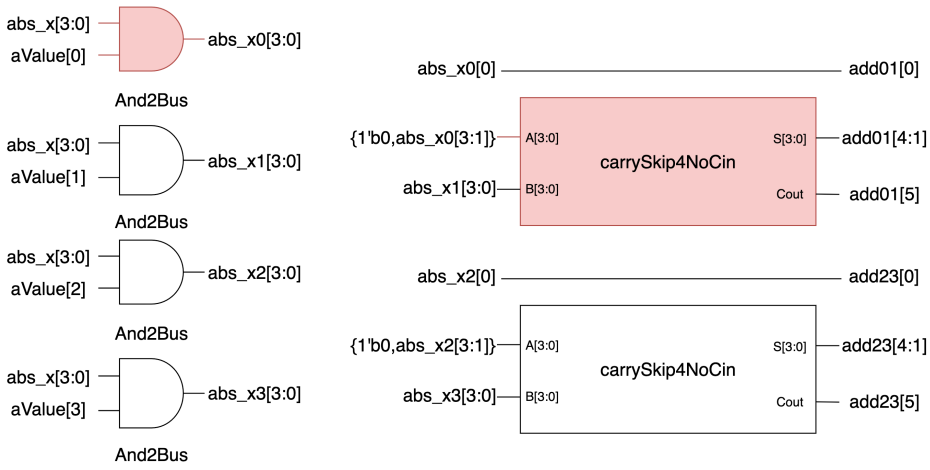
<table>
<tr><td>carrySkip8NoCin</td><td> 

Cascading a carrySkip4NoCin and a carrySkip4</td></tr>
<tr><td>addOne</td><td> 

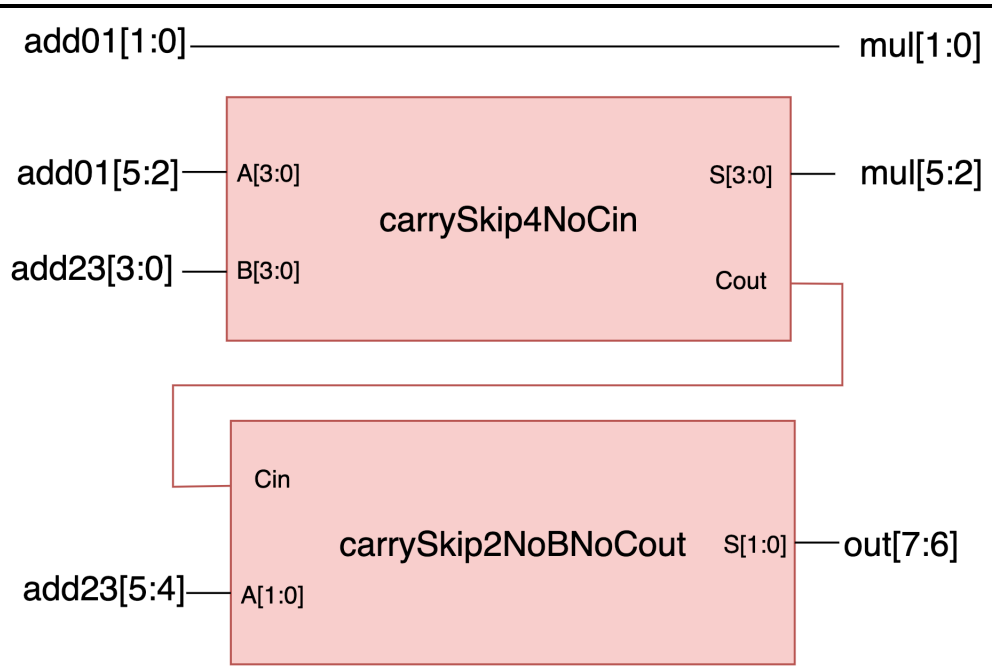
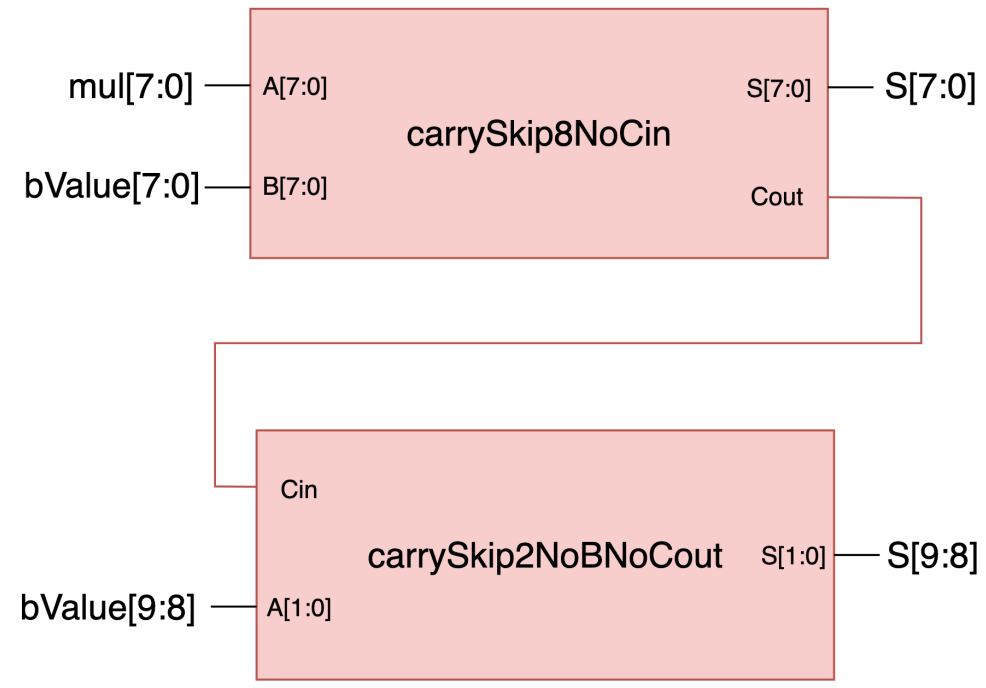
Cascading a carrySkip2NoBCin1, two carrySkip2NoBs,
and a carrySkip2NoBNoCout</td></tr>
<tr><td>a4bitsSelectorEnhanced</td><td> 

The left diagram is the original 8 to 1 mux bus.
The right table is the simplified logics. Use nand gates to realize.</td><td><table>
<tr><td>out[0]</td><td>CTRL[2]'CTRL[0]'
+ CTRL[2]CTRL[1]CTRL[0]
This is the critical path.</td></tr>
<tr><td>out[1]</td><td>CTRL[2]'CTRL[1]'
+ CTRL[0]'</td></tr>
<tr><td>out[2]</td><td>CTRL[1]'</td></tr>
<tr><td>out[3]</td><td>CTRL[2]'</td></tr>
</table></td></tr>
</table>

<table>
<tr><td rowspan="2">b10bitsSelectorEnhanced</td><td>


</td><td>
<table>
<tr><td>out[0]</td><td>CTRL[1]'CTRL[0]'
+ CTRL[2]'CTRL[0]</td></tr>
<tr><td>out[1]</td><td>CTRL[0]'
+ CTRL[2]'CTRL[1]</td></tr>
<tr><td>out[2]</td><td>CTRL[1]'CTRL[0]
+ CTRL[2]'CTRL[1]</td></tr>
<tr><td>out[3]</td><td>CTRL[2]'CTRL[1]
+ CTRL[2]'CTRL[0]
+ CTRL[1]CTRL[0]
+ CTRL[2]CTRL[1]'CTRL[0]'
This is the critical path.</td></tr>
<tr><td>out[4]</td><td>CTRL[2]'CTRL[1]'CTRL[0]
+ CTRL[2]'CTRL[1]CTRL[0]'</td></tr>
<tr><td>out[5]</td><td>CTRL[1]'CTRL[0]
+ CTRL[2]CTRL[1]CTRL[0]'</td></tr>
<tr><td>out[6]</td><td>CTRL[1]'CTRL[0]
+ CTRL[2]CTRL[0]
+ CTRL[2]'CTRL[1]CTRL[0]'</td></tr>
<tr><td>out[7]</td><td>CTRL[1]
+ CTRL[2]'CTRL[0]</td></tr>
<tr><td>out[8]</td><td>CTRL[2]
+ CTRL[1]CTRL[0]'</td></tr>
<tr><td>out[9]</td><td>CTRL[2]
+ CTRL[1]CTRL[0]</td></tr>
</table>
</td></tr>
<tr><td>The left diagram is the original 8 to 1 mux bus.
The right table is the simplified logics. Use nand gates to realize.</td></tr>
<tr><td>mulStage1</td><td>


</td></tr>
<tr><td>Go through an And2Bus to determine whether to add or not.
There are two additions.
{1’b0,abs_x0[3:1]} + abs_x1[3:0]
{1’b0,abs_x2[3:1]} + abs_x3[3:0]
Thus we need 2 carrySkip4NoCins.</td></tr>
</table>

mulStage2	 There is only one addition {2'b00,add01[5:0]} + {add23[5:0],2'b00} But we can split into two add01[5:2] + add23[3:0] add23[5:4] + carry
addStage	 There is only one addition. {2'b00,mul[7:0]} + bValue[9:0] But we can split into two. mul[7:0] + bValue[7:0] bValue[9:8] + carry

7. Discussion

1. Result

```
==========================================================
                    Simulation finished
==========================================================

==========================================================
                        Summary
==========================================================
  Clock cycle:                    3.1 ns
  Number of transistors:          3916
  Total excution cycle:           261
  Approximation Error Score:      1500.0
  Performance Score:              3168435.6
==========================================================
```

2. How to determine the splitting points ?

For convenience, splitting the non-negative sides into 2,4,8,16… equivalent intervals are easier. Since the error of 4 intervals is too large and splitting into 16 intervals costs too many transistors, splitting into 8 intervals is the most efficient and economical.

3. How to determine the number of bits of the slope and the intercept of every interval ?

Since we need to do multiplication on the value of the slope, it is obviously to choose less bits. After plotting the sigmoid function on GeoGebra, I found out that the slope of every interval is less than 0.25. Then I started testing the number of bits and it turned out that choosing 4 bits, 2^{-3} to 2^{-6} ,is the most economical solution. Thus, we can also determine the intercept should be 2^{0} to 2^{-11} . While the first two bits of the intercept will always be 01, so choosing 10 bits is enough. More bits will only cost more resources but increase little precision.

4. How to determine the value of the slope and the intercept of every interval ?

After deciding the manner of splitting and the number of bits, we can now determine the value of the slope and the intercept of every interval. I first used GeoGebra to approximate some values. The second step is to write some codes to determine them. I've wrote JavaScript code to calculate the error.

Below are some parts of my code.

```
const aComponent = [
    [3, 4, 5, 6],   // 0    0.5
    [3, 4, 5],      // 0.5  1
    [3, 5, 6],      // 1    1.5
    [3],            // 1.5  2
    [4, 5],         // 2    2.5
    [4],            // 2.5  3
    [5],            // 3    3.5
    [6]             // 3.5  4
]

const bComponent = [
    [1, 9, 10],               // 0    0.5
    [1, 7, 8, 9, 11],         // 0.5  1
    [1, 5, 6, 7, 8, 9, 10],   // 1    1.5
    [1, 3, 8, 9, 10, 11],     // 1.5  2
    [1, 3, 4, 8, 10, 11],     // 2    2.5
    [1, 2, 6, 9],             // 2.5  3
    [1, 2, 4, 5, 6, 10],      // 3    3.5
    [1, 2, 3, 5, 7, 8]        // 3.5  4
]

const a = aComponent.map(each => {
    // combine the components to get the value
    return each.reduce((acc, current) => acc + Math.pow(2, -current), 0)
})

const b = bComponent.map(each => {
    // combine the components to get the value
    return each.reduce((acc, current) => acc + Math.pow(2, -current), 0)
})
```

```
// Convert binary to decimal and write to output file
const outputFilePath = '../jsResults/OutPractice5.dat';
var mse = 0;
const outputData = inputData.map((line, index) => {
    const decimalX = binaryToDecimal(line.trim()) / 32;
    const absX = Math.abs(decimalX)
    const decimalY = golDataDec[index]
    let approximateY = a[Math.floor(absX * 2)] * absX + b[Math.floor(absX * 2)]
    if (absX == 4) approximateY = a[7] * absX + b[7]
    if (line.trim()[0] === '1') approximateY = 1 - Math.pow(2, -11) - approximateY + Math.pow(2, -12) + Math.pow(2, -14) + Math.pow(2, -15)
    else approximateY += Math.pow(2, -14) + Math.pow(2, -15)
    const error = approximateY - decimalY
    mse += error * error
    // console.log(a[Math.floor(decimalX * 2) + 8], b[Math.floor(decimalX * 2) + 8])
    return ("InputDec : " + decimalX.toString()
        + "\nInputBin : " + line
        + "\nGoldenDec : " + golDataBin[index]
        + "\nGoldenBin : " + decimalY.toString()
        + "\nApproximateDec : " + approximateY.toString()
        + "\nApproximateBin : " + padBinaryTo16Bits((approximateY * Math.pow(2, 15)).toString(2))
        + "\nApproximateHex : " + (approximateY * Math.pow(2, 15)).toString(16)
        + "\nError : " + error.toString()
        + "\n")
});
```

5. How to choose adders ?

There are many kinds of adder we can choose. Carry-skip adder performs well while not costing too many transistors. We've learned a 4-bit based carry-skip adder in the class. After googling, I found out that using 2-bit based carry-skip-adder is better though it cost a bit more transistors. Since I at most need a 10-bit adder and most of them are 8-bit or 4-bit adders, using 2-bit based adders is wise. By wiki, when n=8, m=sqrt (8/2) =2.

I used the 4-bit based carry-skip adder in my circuit. Although it saved some transistors, but it cost 0.4 ns more than the 2-bit based carry-skip adder.

Fixed size block–carry–skip adders [edit]

Fixed size block–carry–skip adders split the n bits of the input bits into blocks of m bits each, resulting in $k = \frac{n}{m}$ blocks. The critical path consists of the ripple path and the skip element of the first block, the skip paths that are enclosed between the first and the last block, and finally the ripple–path of the last block.

$$T_{FCSA}(n) = T_{CRA_{[0:c_{out}]}}(m) + T_{CSK} + (k-2)\cdot T_{CSK} + T_{CRA}(m) = 3D + m\cdot 2D + (k-1)\cdot 2D + (m+2)2D = (2m+k)\cdot 2D + 5D$$

The optimal block size for a given adder width *n* is derived by equating to 0

$$\frac{dT_{FCSA}(n)}{dm} = 0$$
$$2D\cdot\left(2 - n\cdot\frac{1}{m^2}\right) = 0$$
$$\Rightarrow m_{1,2} = \pm\sqrt{\frac{n}{2}}$$

Only positive block sizes are realizable

$$\Rightarrow m = \sqrt{\frac{n}{2}}$$

Reference: https://en.wikipedia.org/wiki/Carry-skip_adder

6. How to lower transistors between different cycles ?

Encoding is important when sending signals between cycles. By encoding, we can save a lot of DFFs (DFFs cost a lot of transistors). Meanwhile, we only need to send useful signals. If the signal is not useful, just don't send it.

For example, I determined the intercept in stage 3 while not stage 1. Since the bandwidth of the intercept is 10 while the CTRL signals is only 3, we can instead send CTRL to stage 3 to conserve some DFFs. Lastly, just decode CTRL into intercept in stage 3.

7. How to accelerate multiplication ?

In our case, multiplication is 4-bit * 4-bit. We can divide the task into three additions : [(A+B)+(C+D)], where A,B,C,D are the 4-bit numbers that multiplicand times each bit of the multiplier and shifts. First, do two additions parallelly : (A+B) and (C+D). Second, add two results together, which is handling the middle addition of [(A+B)+(C+D)]. Although it takes more transistors to realize, we can save much time on it and thus decrease the cycle time. Note that

A : 2^{-8} to 2^{-11}, B : 2^{-7} to 2^{-10}, C : 2^{-6} to 2^{-9}, D : 2^{-5} to 2^{-8}

A+B : 2^{-6} to 2^{-11}, C+D : 2^{-4} to 2^{-9}, final result : 2^{-4} to 2^{-11}

8. How to determine pipeline stages ?

There is no easy way to determine pipelining. I first did the work without pipeline, and then found out that I could not lower the cycle time much (9.5 ns). Then I started splitting tasks into different stages. After testing different number of stages, I found out that 6-stage case performs well. More stages only cost more transistors, but decrease little cycle time. Although the number of transistors increased, but I could further lower the cycle time into (3.1 ns).