Digital Systems Design Report 2

Justin Huang
jh4420@ic.ac.uk
Jubo Xu
jx1820@ic.ac.uk

February 17, 2023

0.1 Implementing SDRAM

In previous tasks, we instantiated only on-chip memory and stored our application in it. Although on-chip memory has its advantage of the fastest data access on FPGA, it has limited size and the most expensive cost. On the DE0 boards there are different types of memories such as SDRAM memory. The SDRAM has much more capacity, reduced cost, which results in its extensive application in digital system. However, the SDRAM requires memory controller which introduces complexity to the system and influences overall performance compared to on-chip memory.

0.1.1 System Design

The SDRAM is connected to the Nios II processor core using SDRAM controller via Avalon switch fabric. The clock for SDRAM controller and SDRAM chip need to be generated separately to meet the clock skew requirements. The clock skew depends on physical characteristics of the DE1-SOC board. For proper operation of the SDRAM chip, the DRAM-CLK, should lead the Nios II system clock iCLK-50 by 3 nanoseconds. This requires the use of a phase-locked loop (PLL) circuit. The phase difference of the PLL on our system is set to -2.55 ns. As shown in figure.

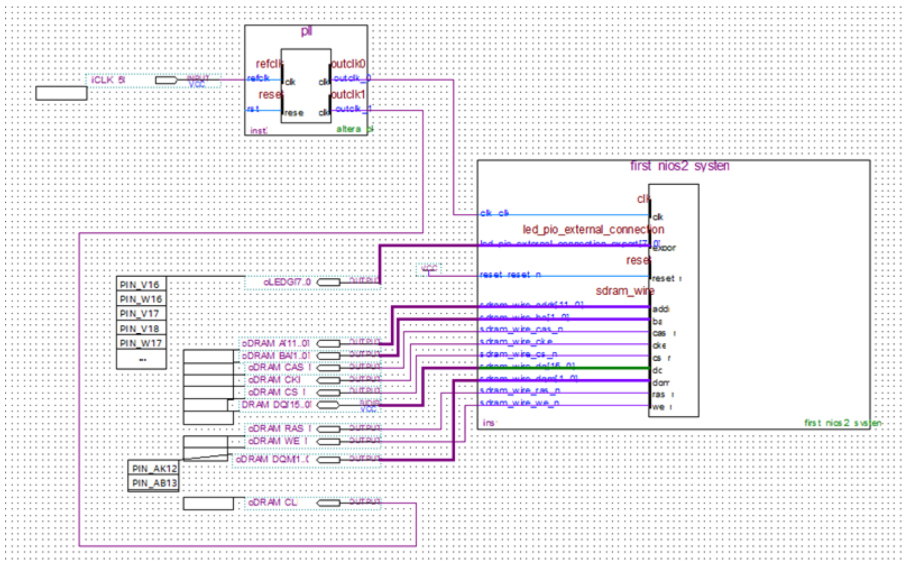
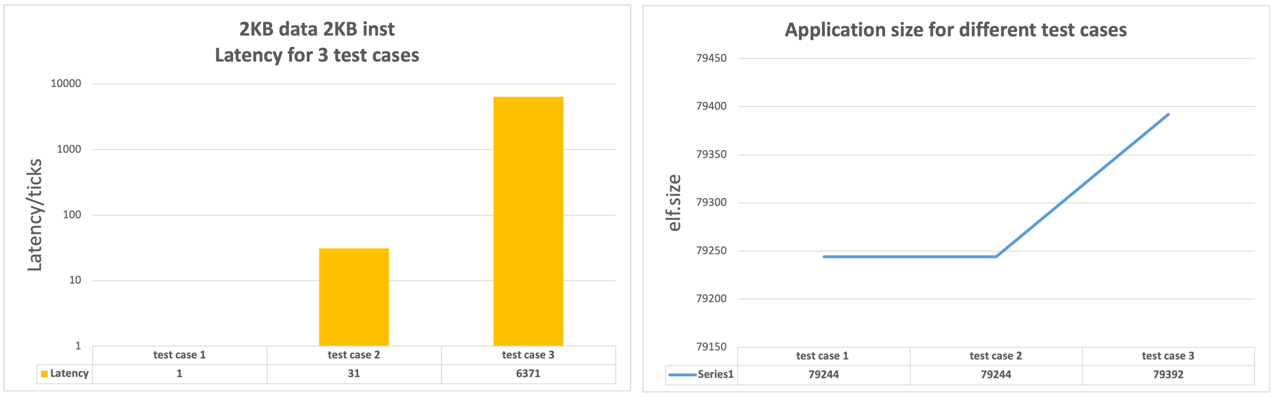


Figure 1: Overall system design

0.1.2 Size and The Evaluation Time of Function for each Test Cases

It takes 1 tick to compute test case 1, 31 ticks to compute test case 2 and 6371 ticks to compute test case 3. After adding SDRAM to the system, the system now has whole support for the C++ language as there is enough memory to store programs. The memory footprint of the program and the overall execution latency increase slightly, but it reduces the complexity of writing programs. The C printf can be used to print the floating point result. As shown in the Figure 2, the computation time using SDRAM for test case 2 (31 ticks) is longer than that using on-chip memory (28 ticks). Furthermore, test case 3 with step of 1/1024 and number 261121 can run on the system now due to the increased memory size.



	test case 1	test case 2	test case 3
🟨 Latency	1	31	6371

	test case 1	test case 2	test case 3
Series1	79244	79244	79392

Figure 2: 2KB data 2KB inst cache latency and size for 3 test cases

0.2 Impact of Cache

0.2.1 Influence of both Data Cache and Instruction Cache

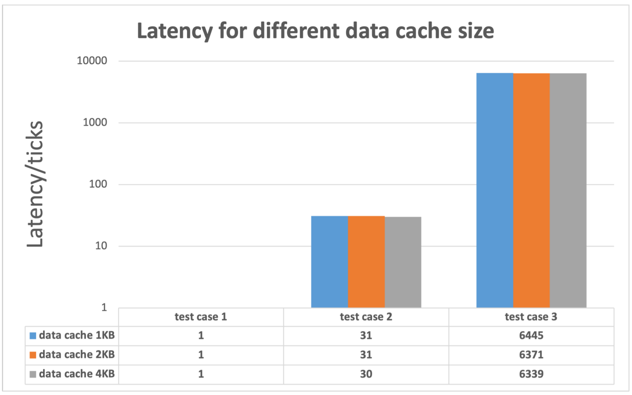
Caches are used in processor to hide memory access latency and improve the overall system performance for the given application. The targeted pro-gram performs a sum of vector product operation. This program involves the use of for loop structures, and the instructions present good temporal locality. The amount of instruction reuse depends on the number of iterations N in the application. The boost in performance is not evident for small number of iterations in test case 1 ($N = 52$) because the amount of instruction reuse is limited. The boost in performance is significant for a large number of iterations in test case 2 or case 3 ($N = 2041/N = 261121$). Given the same benchmark ($N = 261121, step = 1/1024$) and the same data cache size ($2KB$), the execution time for the $2KB$ instruction cache is 6371 clock ticks, and the execution time for the 1KB instruction cache is 10059 clock ticks. The reusable instructions can be stored in a larger instruction cache without being evicted to accommodate for incoming instructions, which improves the system performance significantly. How-ever, the improvement in performance saturates between $4KB$ and $8KB$ data cache. The computation time is reduced by 9 ticks as the data cache in-creased from $4KB$ to $8KB$.

According to our measurement, increase the size of data cache has negligible impact on the system performance for the given application. This is because in each iteration, each vector element $x[i]$ is used only one time and is not being used later in the program. The data used by the program present poor temporal locality. Given the same benchmark ($N = 261121, step = 1/1024$) and the same instruction cache size ($2KB$), the execution time for the $2KB$ and $4KB$ data caches are 6371 and 6339 clock ticks respectively. The vector element can be evicted immediately from data cache after its square is added to the sum. If the same element is accessed repetitively in the program, the effect of increasing data cache may be more evident.

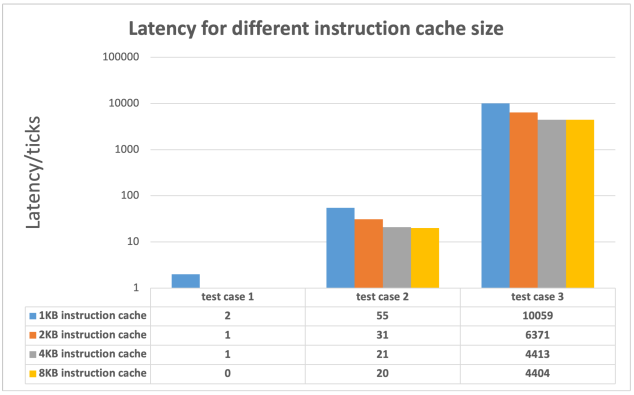
0.2.2 he allocation of on-chip memory to data cache/instruction

The size of the data cache and instruction cache can vary depending on the specific applications. For example, an application that is heavily focused on numerical computation should allocate a larger data cache to the system, as numerical data are frequently accessed. On the other hand, an application that is focused on executing a large number of instructions should allocate a larger instruction cache to store the frequently executed instructions.

Our application involves a large number of numerical computations and instruction executions. However, the data used in the numerical computation present poor temporal locality whereas the instruction used presents high temporal locality. The optimal allocation of on-chip memory therefore should allocate more on-chip memory to instruction cache than data cache. A larger instruction cache size will result in improved performance, but this will come at the cost of increased power consumption and reduced on-chip area.



	test case 1	test case 2	test case 3
🟦 data cache 1KB	1	31	6445
🟧 data cache 2KB	1	31	6371
⬜ data cache 4KB	1	30	6339



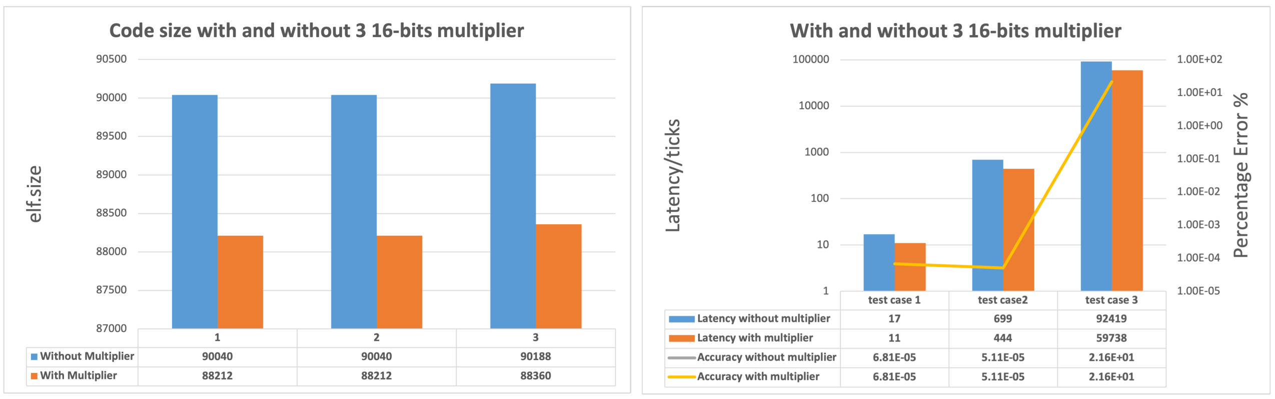
	test case 1	test case 2	test case 3
🟦 1KB instruction cache	2	55	10059
🟧 2KB instruction cache	1	31	6371
⬜ 4KB instruction cache	1	21	4413
🟨 8KB instruction cache	0	20	4404

Figure 3: Latency for different data and instruction cache size

0.3 Implementation of Multiplier

As shown in the graph, adding multiplier reduces the latency of execution, especially for larger number of iterations. For test case 3, the latency reduces from 92419 without the multiplier to 59738 with the multiplier. The code size of the application is also reduced after adding hardware multiplier support. For test case 3, the code size reduced from 90188 without the multiplier to 88360 with the multiplier. This is because without hardware multiplier, the multiplication is emulated with additions. Fewer instructions are needed to perform the same floating point multiplication with the help of multiplier support. The accuracy of the computation remains unchanged because the floating-point arithmetic is still emulated using fix-point arithmetic.

In order to achieve the maximum performance, multiplier support should be included as it reduces the latency of computation, reduces the code size of the application and does not influence the accuracy of the computation. These benefits outweigh the additional system resources used by the multiplier.



	1	2	3
Without Multiplier	90040	90040	90188
With Multiplier	88212	88212	88360

	test case 1	test case2	test case 3
Latency without multiplier	17	699	92419
Latency with multiplier	11	444	59738
Accuracy without multiplier	6.81E-05	5.11E-05	2.16E+01
Accuracy with multiplier	6.81E-05	5.11E-05	2.16E+01

Figure 4: Code size, Latency, and Accuracy with and without multiplier

0.4 Code Optimization for COS function

0.4.1 The implementation of $cos()$ in $< math.h >$

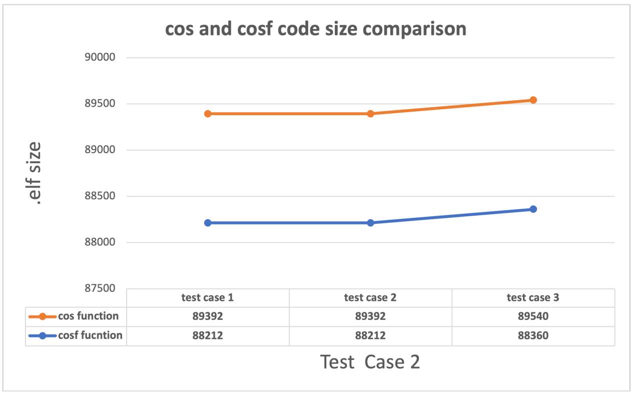
The implementation of the cosine function in *math.h* in GCC is not exact because it can vary for different hardware platform and configuration. But there are several methods that can be used. First, some processors have specific optimized hardware instructions that are part of the *FPU* for calculating the trigonometric functions, for example, most x86 processors have *FCOS* instructions for cosine. Second, the cos function can be implemented in assembly to take advantage of platform-specific features such as *SIMD* that can increase performance. Besides, if the hardware acceleration is not available or the precision of the hardware calculation is not sufficient, then the cos function will be achieved in software based on mathematical algorithms, like Taylor Series Expansion. In general, the *cos* function in *math.h* may not only based on method, usually it's a combination of several methods, and there will be some decision mechanisms to determine which method is used in different situations to optimize the performance.

0.4.2 The difference between $cos()$ and $cosf()$

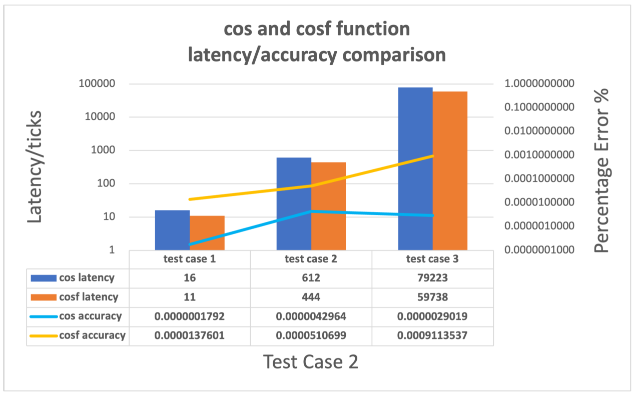
cosf is defined for single-precision floating point number, and *cos* is defined for double-precision floating point number, we are using 32 bit system, so we need two 32 bit numbers to use *cos*, and that's why the size of the application using *cos* is larger than that using *cosf*, as shown in figure. In another view, since '*double*' is represented using 64 bits, then the double precision version $cos()$ will have more significant digits than the single precision version $cosf()$, so $cos()$ will have a more precise result than $cosf()$ which means that the $cos()$ will have a lower error percentage than $cos()$, as shown in figure. The relative error percentage is calculated by comparing with the value calculated from *MATLAB* and *Python*, which is

$$\|\frac{Result_{matlab} - Result_{nios}}{Result_{matlab}}\|$$

Besides, because $cosf()$ uses less memory than $cos()$, it can be faster in some cases, as shown in figure.



	test case 1	test case 2	test case 3
cos function	89392	89392	89540
cosf fucntion	88212	88212	88360



	test case 1	test case 2	test case 3
cos latency	16	612	79223
cosf latency	11	444	59738
cos accuracy	0.0000001792	0.0000042964	0.0000029019
cosf accuracy	0.0000137601	0.0000510699	0.0009113537

Figure 5: Code size, Latency, and Accuracy for $cos()$ and $cosf()$

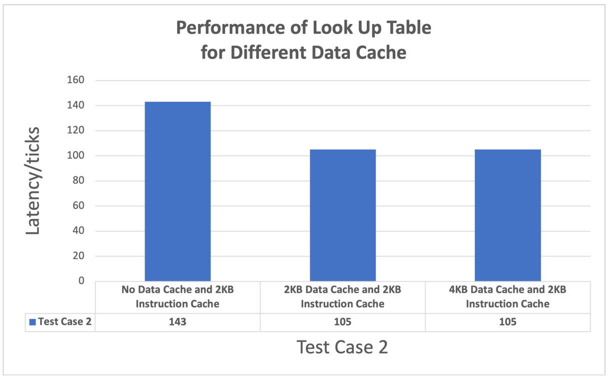
0.4.3 Using Constant Look Up Table for Optimization

Look Up Table is the most straightforward method that can be used to optimize the time complexity by introducing much more space complexity. The purpose of Look Up Table is to store some constants so that we could directly access them without any arithmetic calculation. If we are manipulating hardware we could store these values in a memory unit, like RAM or FIFO, and we could design specific control unit to access them just like we access data from memory. Now we are mainly focus on the software way, which we use an array to store all the constants we need and let the compiler and linker to assign this continuous area into physical memory in our system. There are two kinds of implementations we used for our system, the first one is to calculate all the values specificly for one case and store it for the further use, another way is to store the value of cos for the whole period based on some discrete level, this would introduce error and we could implement linear interpolation, which calculated the weighted average between two nearest value for an input that isn't in the table.

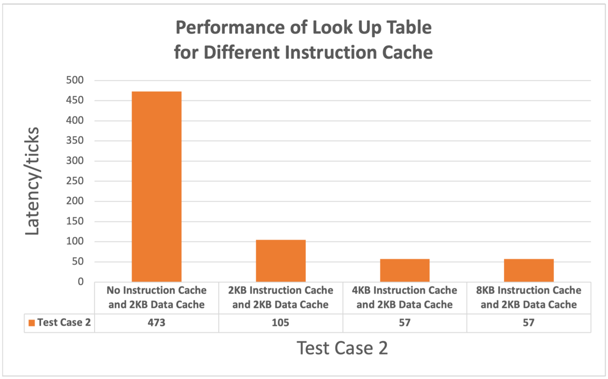
The graphs show the performance of a simple one level look up table for test case 2 with various cache sizes. Adding a data cache has less performance improvement compared to adding an instruction cache. Adding a 2KB data cache reduces the number of clock ticks by 30 whereas adding a 2KB instruction reduces the number of clock ticks by 360. This is due to the poor temporal locality of data and the high temporal locality of instructions. Therefore, allocating a larger instruction cache will results in a better system performance for the look up table implementation.

The conclusions for increasing data cache size and increasing instruction cache size are similar to previous tasks. The initial introduction of a small data cache slightly boost the performance by providing faster data access for the processor core. Afterward, increasing the size of data cache has negligible impact on the system performance. The benefit of increasing the instruction cache size is more evident. However, as there are only 2041 iterations for test case 2, the improvement saturates quickly.

The implementation of look up table for test case 3 triggers an error in Eclipse. The look up table is too large for the given system. The linker uses relative addressing but the bss, text and data section in the memory are too far away in the address space. On potential solution would be editing the addresses so that they are contiguous in the memory space. Another solution would be to write a custom link script that declares a single space including all the memories and that maps all the sections (except heap and stack) to that space.

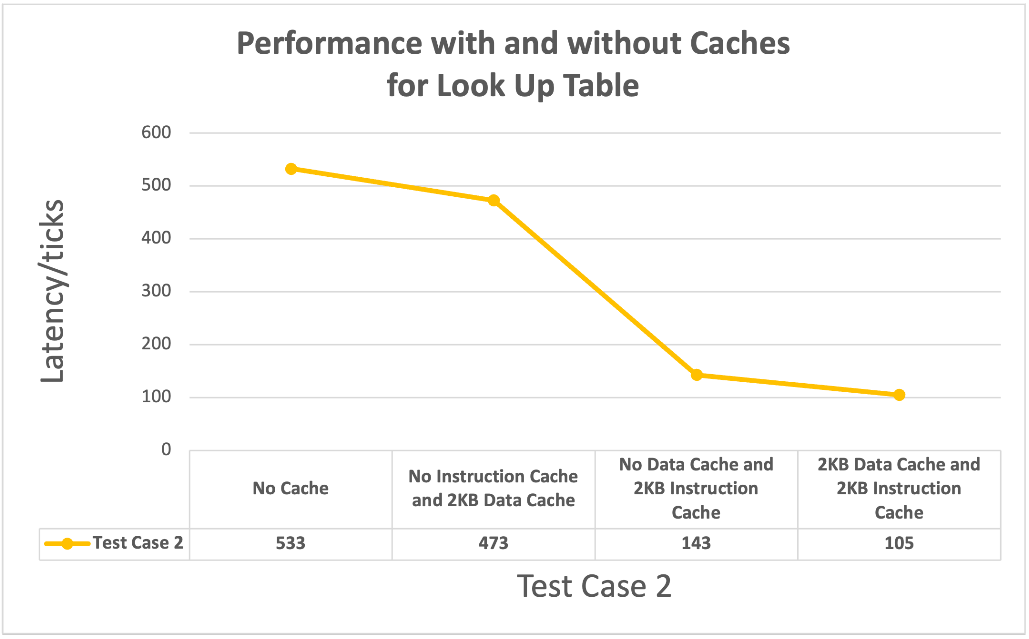


	No Data Cache and 2KB Instruction Cache	2KB Data Cache and 2KB Instruction Cache	4KB Data Cache and 2KB Instruction Cache
Test Case 2	143	105	105



	No Instruction Cache and 2KB Data Cache	2KB Instruction Cache and 2KB Data Cache	4KB Instruction Cache and 2KB Data Cache	8KB Instruction Cache and 2KB Data Cache
Test Case 2	473	105	57	57

Figure 6: Performance of look up table for different data and inst cache



	No Cache	No Instruction Cache and 2KB Data Cache	No Data Cache and 2KB Instruction Cache	2KB Data Cache and 2KB Instruction Cache
Test Case 2	533	473	143	105

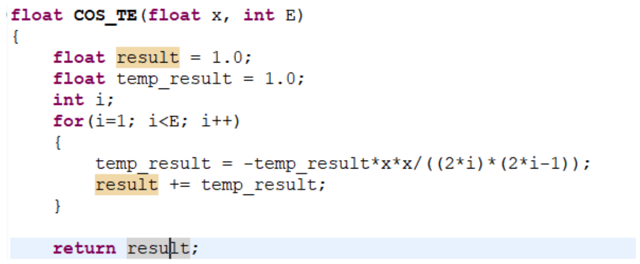
Figure 7: Performance with and without cache for LUT

0.4.4 Optimization based on Factorization

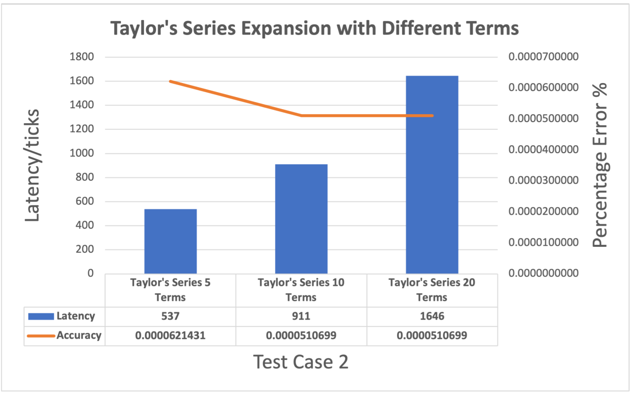
Factorisation of x in the function slightly reduces the latency of the execution because factoring out the variable x reduces the number of multiplications needed for each iteration. However, its effect is not significant for our application.

0.4.5 Optimization based on Taylor Series Expansion

To evaluate the cos function using other methods, the Taylor Series expansion is usually used. Two different ways are implemented for Taylor series for float type, we first used a naïve version, which means we directly put the Taylor series formula as our function, and the second version is based on iterative method, which I could tune the number of order for testing, as shown in figure. The naïve version for order of 10 has application size of 83000 bytes and latency of 1201 ticks and the iterative version with the same order has application size of 81272 bytes and latency of 911 ticks. This is probably because that in the naïve version there are lots of terms and multiplications, after compilation, there would be lots of instructions to operate, which would increase the text size and operation time. We mainly used the iterative version to check the influence of the number of terms on the performance of cos. Theoretically, if we include more terms, than the result would be more precise, but since there are more terms to operate, the operation time will increase. As shown in figure, when the number of terms increases, the latency increases a lot, but for accuracy, it can be seen if there are only 5 terms, the value is not that precise, but if we exceed a certain level, for example, the value for 10 terms and 20 terms is the same, because the higher terms have lower values, and maybe the value is too small to be represented by a infinite number of bits.



```
float COS_TE(float x, int E)
{
    float result = 1.0;
    float temp_result = 1.0;
    int i;
    for(i=1; i<E; i++)
    {
        temp_result = -temp_result*x*x/((2*i)*(2*i-1));
        result += temp_result;
    }

    return result;
```



	Taylor's Series 5 Terms	Taylor's Series 10 Terms	Taylor's Series 20 Terms
Latency	537	911	1646
Accuracy	0.0000621431	0.0000510699	0.0000510699

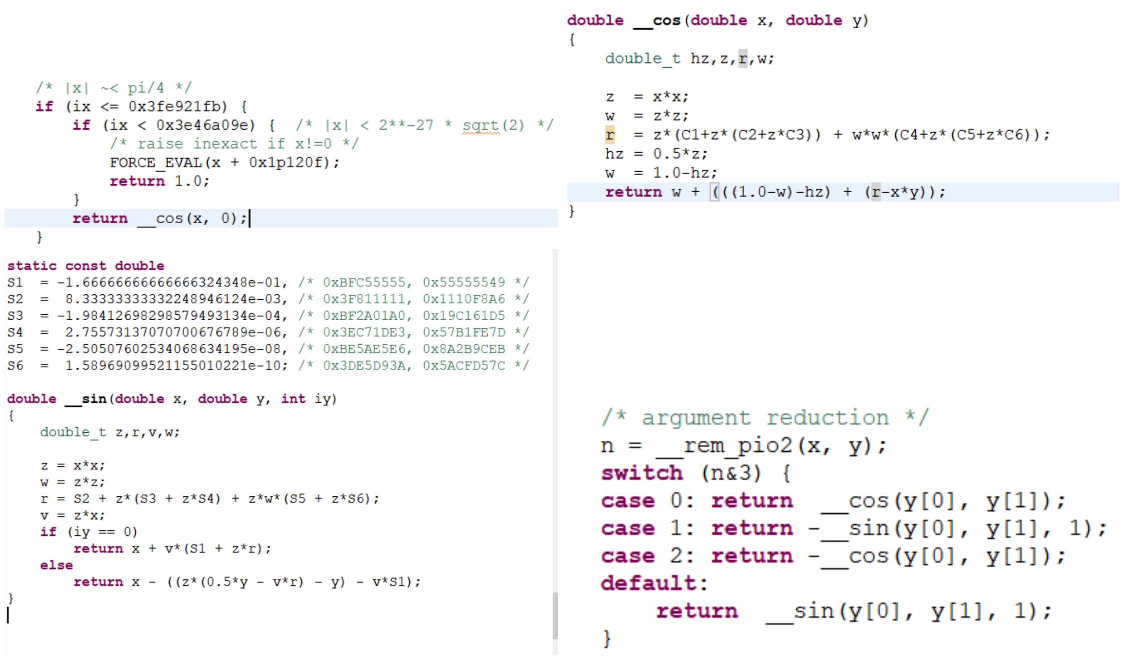
Figure 8: Taylor Series code and comparison

0.4.6 Advanced Optimization

To have a general consideration about code optimization for cosine implementation, there are lots of tricks that could be used to optimize the performance based on Taylor Series method. For example, because cos is periodic, we could first make the x into the form of $x^{'} + 2k\pi$, and use Taylor Series to calculate the $cos(x^{'})$, otherwise the result will not be very precise because Taylor Series doesn't work very well for large values. The method is this: reduce the input x to $x^{'} = x - k\pi/2$ such that $x^{'}$ is in $[-\frac{\pi}{4}, \frac{\pi}{4}]$, and also let $n = kmod4$. Based on trigonometry, if n = 0, then we could calculate $cosx$ directly,; if n =1, then $cosx = -sinx^{'}$; if n=2, $cosx = -cosx^{'}$; and if n=3, $cosx = sinx^{'}$. Therefore, the basic procedure is: after getting the input x, first use masking operation & =0x7fffffff to clear the sign bit of the float, which takes the absolute value of x. For single-precision floating point, ix now contains the exponent term and a part of mantissa of x, although it's not exactly equal to the absolute value of x, it's in the same magnitude. Since cos(-x) = cos(x), we only need to consider about positive x, we could then use this approximate value to determine whether the magnitude of x is smaller than $\frac{\pi}{4}$, which is 0x3f490fdb for single precision floating point representation, and also check if it's smaller than a certain threshold, because for very small x, we don't need to do any calculation, cosx is approximately equal to 1, otherwise, we use function $cos()$ to calculate the cosine of x. The basic principle of $cos()$ is this: Because Taylor Series expansion is more precise for smaller value, we could further divide input x into two small values x1 and x2, and use Taylor Series(assume the highest order is 14), then $cos(a) \approx 1 - 0.5a^2 + C_1a^4 + \ldots + C_6a^{14}$, and we could store the value of these constants. Therefore $cos(x) = cos(x_1 + x_2) = cosx_1cosx_2 \breve{\ } sinx_1sinx_2$, because x1 and x2 are relatively small, so $cosx_2 \approx 1, sinx_2 \approx x_2, sinx_1 \approx x_1$, so $cos(x) \approx cosx_1 - x_1x_2$, let $r = C_1{x_1}^4 + .. + C_{14}{x_1}^{14}$, then $cos(x) = 1 - (0.5 \times {x_1}^2 - (r - x_1x_2))$, we could also add a tiny fixed correction term to compensate for the error, as shown in code. $sin(x)$ uses the same principle.

If the absolute value of x is greater than $\frac{\pi}{4}$, then we could first use a function to find the quotient and remainder of x divided by $\frac{\pi}{4}$, k is the quotient and array y[2] stores the remainder, where y[0]+y[1] = remainder. The implementation of this function varies for platforms, but usually there's a built in function $rempio2(x, y)$ that can be used. Then following our methodology, there are four cases of n = k mod 4, this can be achieved as n = k&3 because k can be divided into two parts: the most significant bits and last two least significant bits, the first part can always be divided by 4 because it can simply shift right by two bits, the remainder is just the second part, therefore if we and this number with 0b11, the result would be the remainder. Then, if n is 0, $cos(x) = cos(y[0], y[1])$; if n is 1, $cos(x) = -sin(y[0], y[1])$; if n is 2, $cos(x) = -cos(y[0], y[1])$; otherwise, $cos(x) = sin(y[0], y[1])$. Last but not least, there should be a consideration about Inf and NaN. As shown in code.

In general, such kind of code optimization is a hybrid structure, it uses the Taylor Series as the basis and solving the problem of imprecision due to large input value by mapping it into a small range based on the periodicity of the cos function. It also calculates the cos function for different cases.



```c
	/* |x| ~< pi/4 */
	if (ix <= 0x3fe921fb) {
		if (ix < 0x3e46a09e) {  /* |x| < 2**-27 * sqrt(2) */
			/* raise inexact if x!=0 */
			FORCE_EVAL(x + 0x1p120f);
			return 1.0;
		}
		return __cos(x, 0);
	}
```

```c
static const double
S1  = -1.66666666666666324348e-01, /* 0xBFC55555, 0x55555549 */
S2  =  8.33333333332248946124e-03, /* 0x3F811111, 0x1110F8A6 */
S3  = -1.98412698298579493134e-04, /* 0xBF2A01A0, 0x19C161D5 */
S4  =  2.75573137070700676789e-06, /* 0x3EC71DE3, 0x57B1FE7D */
S5  = -2.50507602534068634195e-08, /* 0xBE5AE5E6, 0x8A2B9CEB */
S6  =  1.58969099521155010221e-10; /* 0x3DE5D93A, 0x5ACFD57C */

double __sin(double x, double y, int iy)
{
	double_t z,r,v,w;

	z = x*x;
	w = z*z;
	r = S2 + z*(S3 + z*S4) + z*w*(S5 + z*S6);
	v = z*x;
	if (iy == 0)
		return x + v*(S1 + z*r);
	else
		return x - ((z*(0.5*y - v*r) - y) - v*S1);
}
```

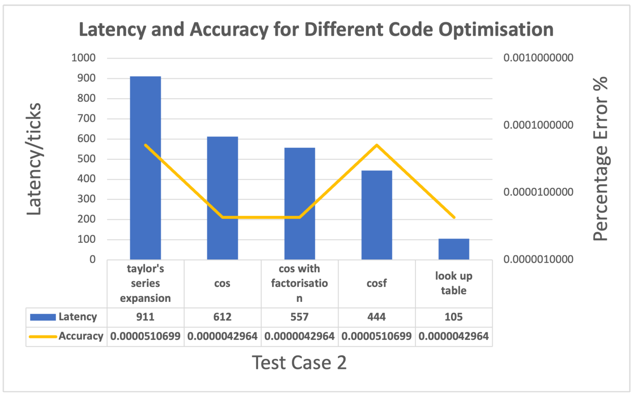
```c
double __cos(double x, double y)
{
	double_t hz,z,r,w;

	z  = x*x;
	w  = z*z;
	r  = z*(C1+z*(C2+z*C3)) + w*w*(C4+z*(C5+z*C6));
	hz = 0.5*z;
	w  = 1.0-hz;
	return w + (((1.0-w)-hz) + (r-x*y));
}
```

```c
	/* argument reduction */
	n = __rem_pio2(x, y);
	switch (n&3) {
	case 0: return  __cos(y[0], y[1]);
	case 1: return -__sin(y[0], y[1], 1);
	case 2: return -__cos(y[0], y[1]);
	default:
		return  __sin(y[0], y[1], 1);
	}
```

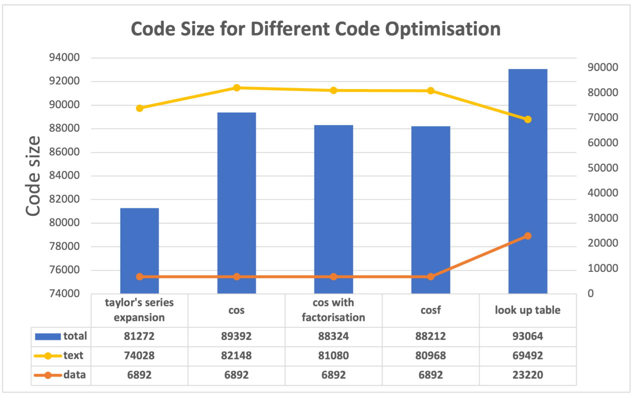
Figure 9: Optimization Code

0.4.7 General Consideration

It can be seen from the figures that Look up table is the fastest one but requires the most resources. The pure Taylor Series Expansion is the slowest one but requires the least resources. This is reasonable because Loop Up table needs to store all the values we need but we could directly access it. The Taylor Series Expansion needs to operate iteratively therefore requires more instruction cycles to get the result, but it doesn't need to store anything except for the temp result for each cycle. It can also be seen that the factorization has some improvement on the speed but doesn't influence the application size a lot, therefore it's better to use factorization if we could.The percentage error in the figures are calculated by comparing with the double-precision value calculated by MATLAB.



	taylor's series expansion	cos	cos with factorisation	cosf	look up table
▬ Latency	911	612	557	444	105
▬ Accuracy	0.0000510699	0.0000042964	0.0000042964	0.0000510699	0.0000042964



	taylor's series expansion	cos	cos with factorisation	cosf	look up table
▬ total	81272	89392	88324	88212	93064
● text	74028	82148	81080	80968	69492
● data	6892	6892	6892	6892	23220

Figure 10: Code size, performance, and accuracy for different code optimisation method

0.5 Resources and Performance

As shown in figure, this is the resources and performance for 2 kB Data cache and different Instruction cache, and performance 1, 2, 3 is for case 1, 2, 3. The resources is calculated based on the formula given in the coursework instruction. It can be seen that generally when resources increases, the performance increases, but when the resource reach a limit, the performance will be almost the same.

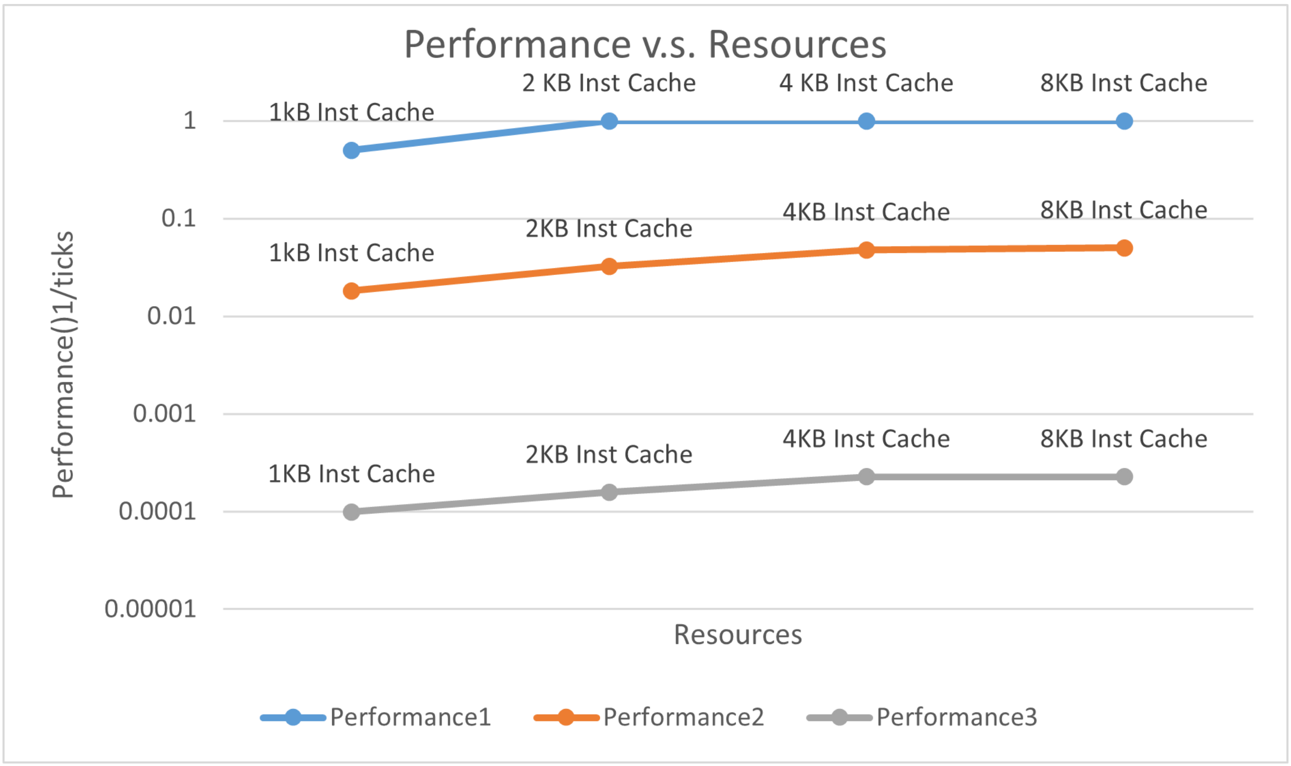


Figure 11: Code size, performance, and accuracy for different code optimisation method

0.6 Conclusion

In general, after implementing the SDRAM, we have sufficient memory storage to do more complex operations. When we try to do a more complex calculation in this case, when including the nonlinear operation cos, the effect of data and instruction cache is still remarkable, especially for the instruction in our case because we have lots of repeated instructions in our case. Therefore changing the instruction cache could optimize our operation. Besides, after adding the multipliers, the operation speed is much faster, this is because hardware is usually faster than software simulation because for software simulation, they system needs to use the existed operation to do some non exist operation, which requires much more cycles. Further more, different implementations of cos function is tried in our case, the fastest one is look up table but it requires too much resources which makes it not quite suitable for the practical use especially if the resources is limited and highly demanded. There are some mathematical algorithms can be used to implement cos and Taylor Series Expansion is the most common one, it's very easy to be implemented but the accuracy is relatively good. But in the real life applications, usually the cos is not implemented only by one method, because each algorithm is valid only for a certain range of input value, therefore, the cos algorithm is always a combination of several algorithms, based on the value of input data we could determine which method to use, and the implementation is also different for different hardware platforms. The cos implementation needs to be optimized for the specific hardware and software in a specific way. Last but not least, when comparing the result calculated by our system with the result calculated by Matlab and python, if it is case 1, which means we use integer for calculation, then result is basically the same, but if the value is float, then the result of our system is generally not as accurate as that of matlab and python. This is mainly because we are using single-precision here, but matlab and python are based on our laptop which uses double-precision, therefore more bits for exponent term and mantissa, and therefore has a more precise representation.