Department of Electrical and Electronic Engineering
Imperial College London

EE3-05 Digital System Design

Report 2: Task 3-5

Kaiyue Sun
ks4016@ic.ac.uk
01197813

Yuwei Wang
yw6316@ic.ac.uk
01260801

TABLE OF CONTENT

I. Introduction

A. Aim

This report mainly focuses on presenting and analysing the results of exploring extra available supports (e.g. off-chip memory and hardware multipliers) to accelerate the function evaluation process on FPGA, as well as discussing and justifying any design choices made.

B. General Design Briefing

Since the on-chip memory could not meet the large memory requirements of evaluating test case 3, an SDRAM chip which can store 8 Mbytes of data was provided.

Due to the fact that there is a delay to reach the off-chip memory and get the data in time to be registered by the processor, the clock that drives the SDRAM should lead the clock that drives the CPU. Therefore, in **Figure-1**, the leading clock 'c1' which has a phase shift of -45.9 degrees from the PLL drives the SDRAM, and the clock 'c0' drives the CPU.

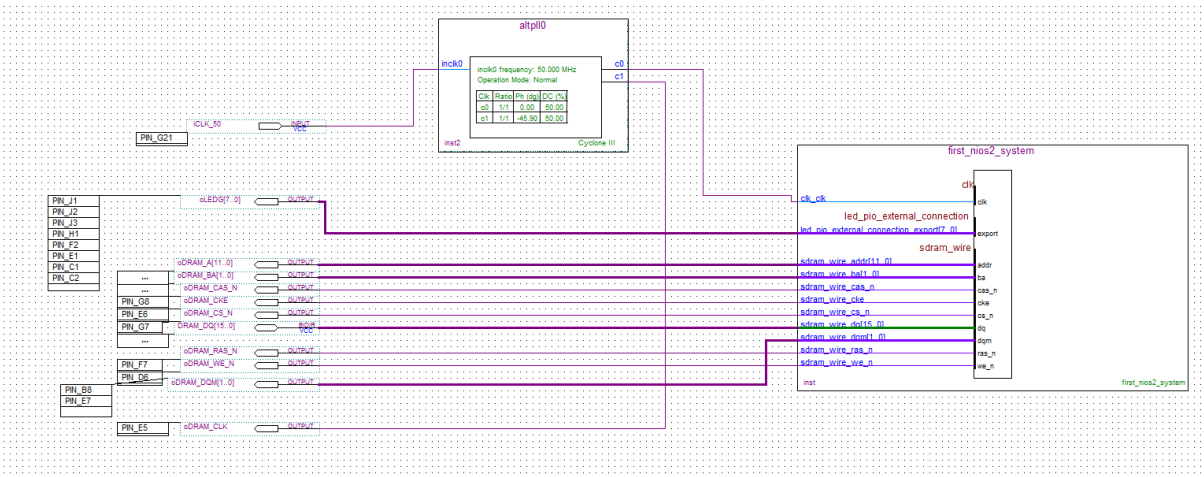

Figure-1 System Block Diagram

C. Notes

One thing deserves attention is that task 3 implements a different mathematical function from task 4 and 5.

Below is the metric for FPGA resources.

$$FPGA\ resources = \frac{1}{3}\left(\frac{LE}{Total\ LE\ in\ the\ device} + \frac{EM}{Total\ EM\ in\ the\ device} + \frac{MB}{Total\ MB\ in\ the\ device}\right)$$

II. Task 3: Storing the Program and Data on External Memories

The FPGA resources of the system are recorded in **Table-1** below, with default instruction cache size of 2 KB. Apparently, the memory bits occupied on the FPGA reduces after removing the on-chip memory, because data is stored in off-chip memory, which is not counted as part of the FPGA resources usage.

Resources	Usage	%
Logic Elements	3,023/15,408	20%
Combinational functions	2,722/15,408	18%
Dedicated logic registers	1,920/15,408	12%
Registers	1988	
Pins	47/347	14%
Virtual pins	0	
Memory bits	29,056/516,096	6%
Embedded Multiplier 9-bit elements	0/112	0%
PLLs	1/4	25%

Table-1 Resource Utilisation Summary

As mentioned in the last report, the lower bound of the memory required for test case 3 was 1058.01 KB which exceeded the size of on-chip memory but would only occupy around 1/8 of the SDRAM. Therefore, test case 3 would be able to run with the support of SDRAM.

However, it takes longer to access the off-chip memory than the on-chip one for the NIOS processor. This is verified by the latencies for different memory types shown in **Table-2** (both with default instruction cache size). SDRAM is about 1.5 times slower than on-chip for test case 2.

Memory	Test Case 1	Test Case 2	Test Case 3
On-chip	1	62	N/A
SDRAM	2	97	14692

Table-2 Latency (unit: ms) Comparison for Different Memory Types

The size of the program that needs to be downloaded to the system is 58 KB and is independent of the test cases because the vector x is declared as a local variable and uses dynamic memory allocation.

The impacts of increasing the instruction cache size to the required resources and to the achieved latency of the system are shown in **Figure-2, 3, 4**. Enlarging the instruction cache size occupies more memory in the device, so increases the FPGA resources used. On the bright side, it can reduce the latency.

The trend is not salient for test case 1 since the execution time is very close to the resolution of the interval timer.

For test case 2 and 3, the latency exhibits a sudden plunge when the cache size changes from 2 KB to 4 KB. However, it only shows a slight decrease or just stays at the same level as the cache is increased

further, implying that if the largest performance-critical instruction loop, which should be below 8KB in this program, can be fitted in the instruction cache, increasing the cache size furthermore does not help with the performance very much. Thus, in a more efficient way, the cache size should be restricted to 8KB for this program. It is clear that with more FPGA resources implemented, the on-chip memory achieves better latency performance than SDRAM as expected, since the on-chip memory is the fastest and the most expensive one on the FPGA.

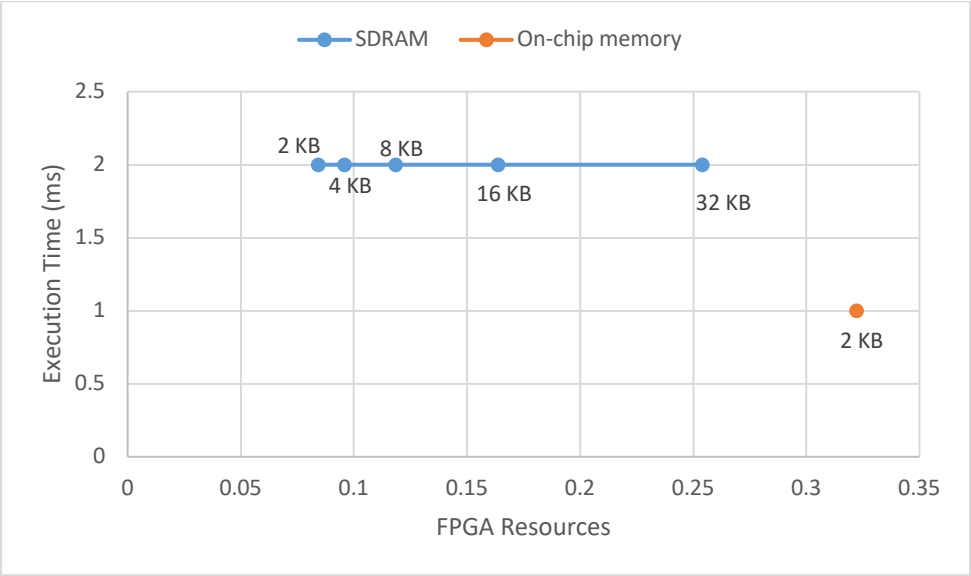

Figure-2 Execution Time vs. FPGA Resources (Test Case 1)

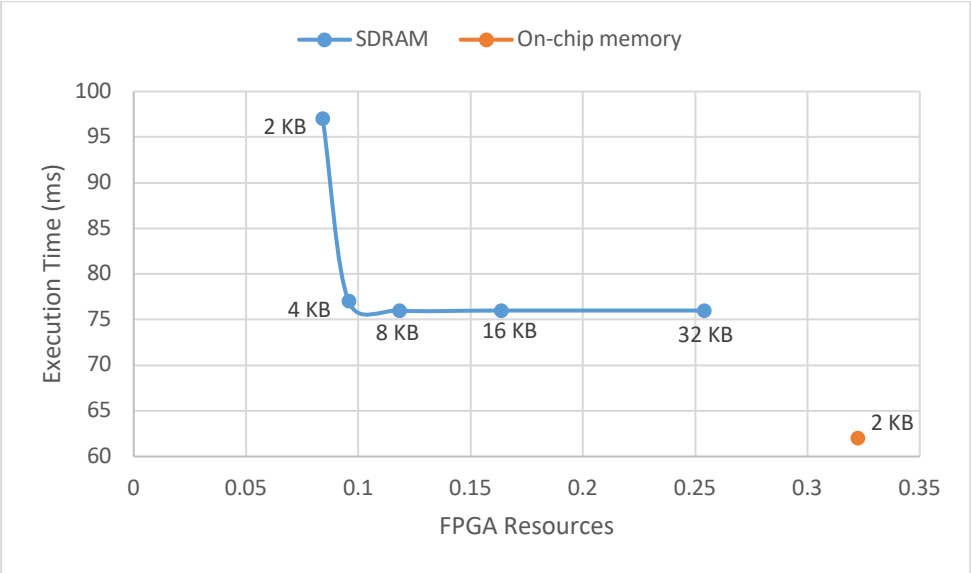

Figure-3 Execution Time vs. FPGA Resources (Test Case 2)

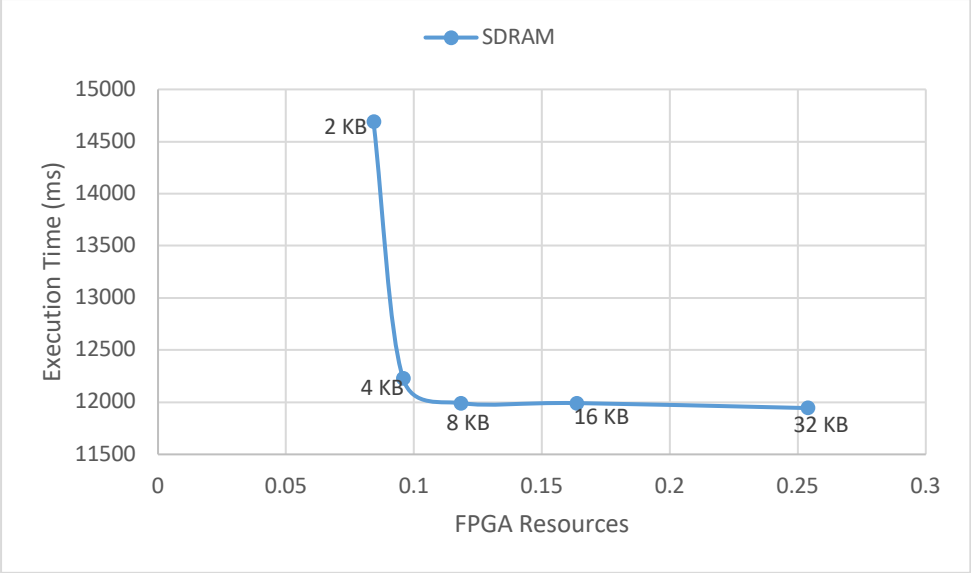

Figure-4 Execution Time vs. FPGA Resources (Test Case 3)

III. Task 4: Evaluate a More Complex Mathematical Expression

Since speed is quite important for a successful digital system, several attempts were made to improve the system's latency performance on computing mathematical expression.

$$f(x) = \sum 0.5x + x^2 \cos\left(\frac{x - 128}{128}\right)$$

Firstly, the expression inside cos function (x-128)/128 was simplified to x/128-1. Also, x was factorized out.

$$f(x) = \sum x(0.5 + x \cos\left(\frac{x}{128} - 1\right))$$

Moreover, *cosf*( ) was substituted for *cos*( ) because both *cos*( ) and *cosf*( ) compute the cosine of the argument in radian. *cosf*( ) is just a single-precision version of *cos*( ). Computing with less precision will improve the speed significantly without losing too much accuracy, since both *x* and *f*(*x*) are float.

$$f(x) = \sum x(0.5 + x \operatorname{cosf}\left(\frac{x}{128} - 1\right))$$

Another thing popped in mind was utilising shift to implement division, due to the fact that 128 is a power of 2. However, *x* is float, not allowing easy implementation of shift. One possible way to achieve the shift is to cast *x* into an integer first, realise the shift and then cast it back to float. This approach will decrease the latency by considerable amount but at the expense of sacrificing a lot of precision.

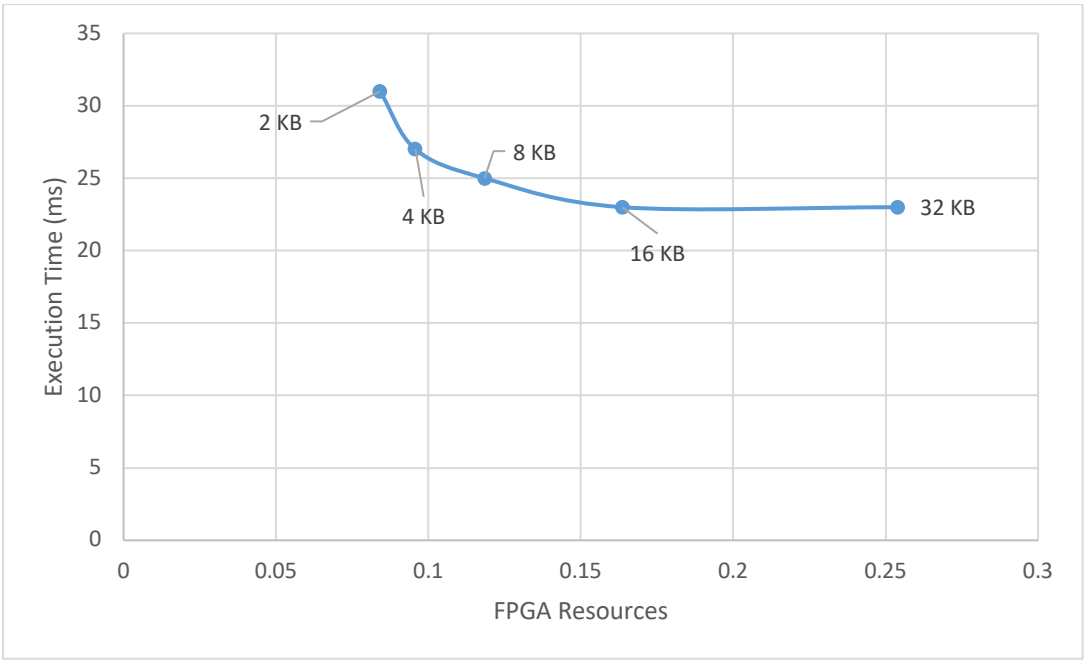

Figure-5 Execution Time vs. FPGA Resources with SDRAM (Test Case 1)

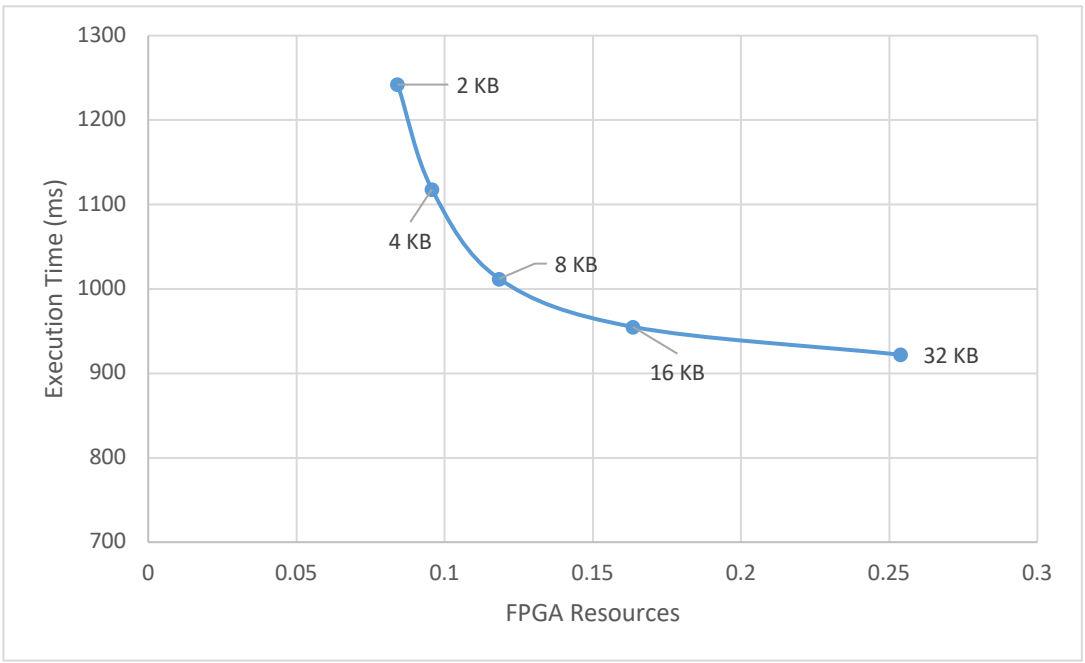

Figure-6 Execution Time vs. FPGA Resources with SDRAM (Test Case 2)

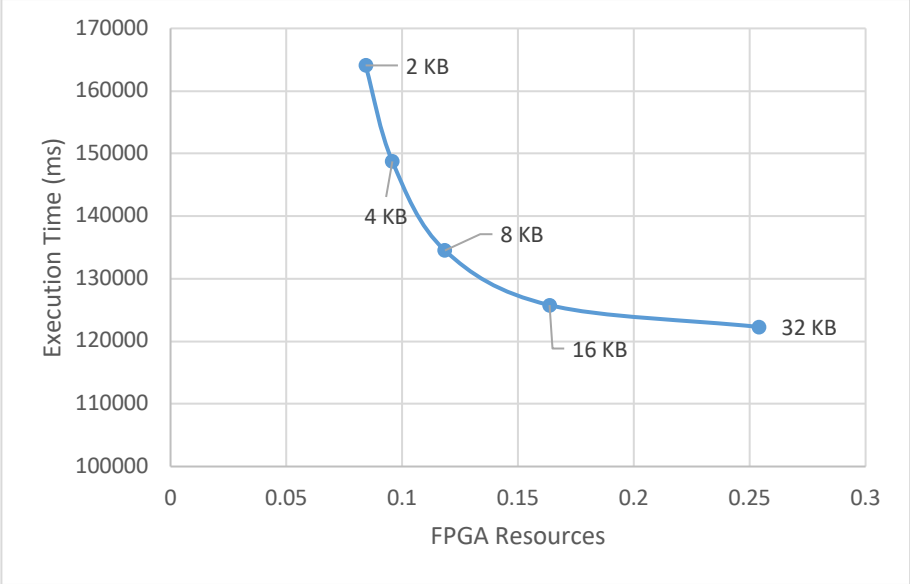

Figure-7 Execution Time vs. FPGA Resources with SDRAM (Test Case 3)

As the **Figure-5, 6, 7** shown, there clearly exists a trade-off between hardware resources and execution time. The more resources devoted, the less execution time needed. At the beginning, increasing cache size from 2 KB leads to significant improvement on latency performance without occupying too much additional resources, but on the contrary, when the cache grows furthermore, especially from 16 KB to 32 KB, the improvement is trivial despite the fact that it already occupies almost 10% more FPGA resources. Nevertheless, there is indeed a few seconds improve on latency.

	Result	MATLAB Result	Relative Error
Case 1	920413.5	920413.626649442	-1.37601E-05
Case 2	36123104	36123085.5519791	5.10699E-05
Case 3	4621531136	4621489017.88862	0.000911354

Table-3 MATLAB Result Comparison

The final result of each test case remains the same for different sizes of instruction caches. Although the result produced is not as accurate as MATLAB double-precision one, the relative error is quite small as shown in **Table-3** and it still meets the required accuracy for single-precision floating point calculation, therefore considered to be acceptable.

Regarding the improvement of accuracy, the algorithm to realise summation can be optimised. So far, the summation was calculated through accumulation.

$$f(x) = f(x) + x[i](0.5 + x[i]\,\mathrm{cosf}\left(\frac{x[i]}{128} - 1\right))$$

However, as $f(x)$ becomes larger gradually, the update part for $f(x)$ remains relatively small. At a certain point, the result of adding a small number to a big number is not that accurate anymore because of the floating-point representation in the calculation.

Completing summation in groups is able to address this problem. For example, the first ten elements in *x* can be evaluated and sum together then the next ten. After the computations of all groups in ten are finished, adding all results will give the final values.

Batch Size	Result	Execution Time (ms)	Absolute Error
100	4621490688	124290	1670.11
400	4621489664	123973	646.11
500	4621488640	123907	377.89
600	4621488128	123878	889.89
800	4621488128	123821	889.89
1000	4621490688	123949	1670.11

Table-4 Summary of Implementing Multiple Streams Method

It is found out that increasing the batch size will not only improve the precision but also decrease the execution time. However, after a certain stage, the performance starts to degrade. Base on the exploration done so far, a group size of 500 seems to be optimal.

This approach will largely increase the accuracy since the numbers added together at last stage are almost comparable, but the latency performance is not as good as before. Whether it is worth or not trading the latency for precision depends on where this is applied to. For the purpose of this project, latency is the foremost factor to consider as long as the result meets expected accuracy. Therefore, the method of multiple streams will not be adopted.

The actual implementation of cos function is quite complicated. The library will choose different algorithms depending on the input value given, since there is no single algorithm that is able to compute as fast as possible and at the same time accurate over the whole range of input values. The algorithm that would be selected also depends on the hardware resources available (e.g. adder, multiplier, divider, etc.).

One of the possible choices is Taylor Series.

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} \dots$$

Another commonly used one is Chebyshev Polynomials and however the polynomials actually used vary with circumstances in order to achieve fastest operations. For example, with a fast divider, the fastest way to compute may be $P_1(x)/P_2(x)$ where P_1 and P_2 are Chebyshev polynomials, while without it, P(x), which contains much more term than P_1 and P_2, may become the solution. Then, what left is to combine Chebyshev polynomials in a proper way (e.g. cos function usually in the form of cos(ax) = a*P(x)) and find out the corresponding order of polynomials as well as the coefficients in the appropriate look-up tables according to the required precision. If 7-digit precision is needed, the corresponding order will be 4, implying a polynomial in the form of $p_4*x^4 + p_3*x^3 + p_2*x^2 + p_1*x + p_0$

(p_i is the coefficient). At last, implementing $(((p_4*x + p_3)*x + p_2)*x + p_1)*x + p_0$ will generate answers with desirable precision.

IV. Task 5: Add Multiplier Support

Logic Elements multiplier (LE) or Embedded Multiplier (EM) is added on FPGA in order to provide hardware support for multiplication. The results are the same as in task 4 and the code size is unchanged as well.

The comparisons between the three cases (LE, EM and without multiplier) are quite similar across all three test cases as shown in **Figure-8, 9, 10**.

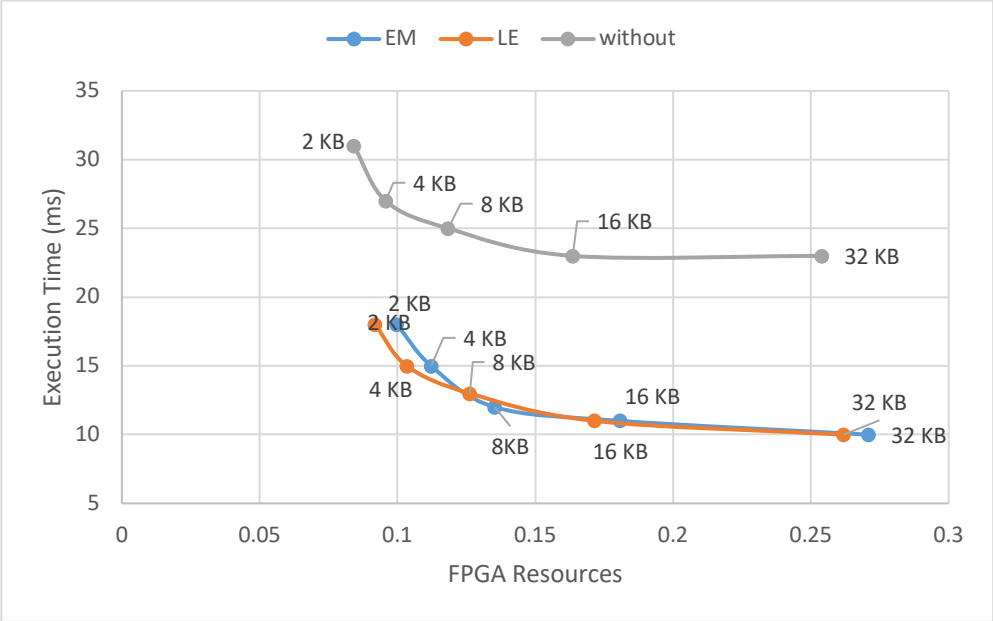

Figure-8 Execution Time vs. FPGA Resources with Multiplier (Test Case 1)

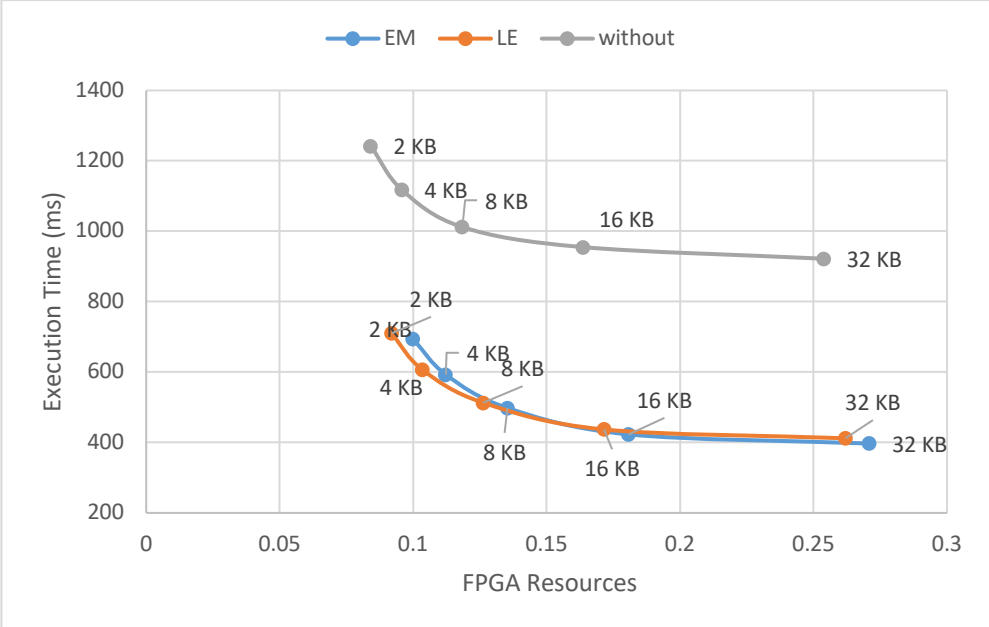

Figure-9 Execution Time vs. FPGA Resources with Multiplier (Test Case 2)

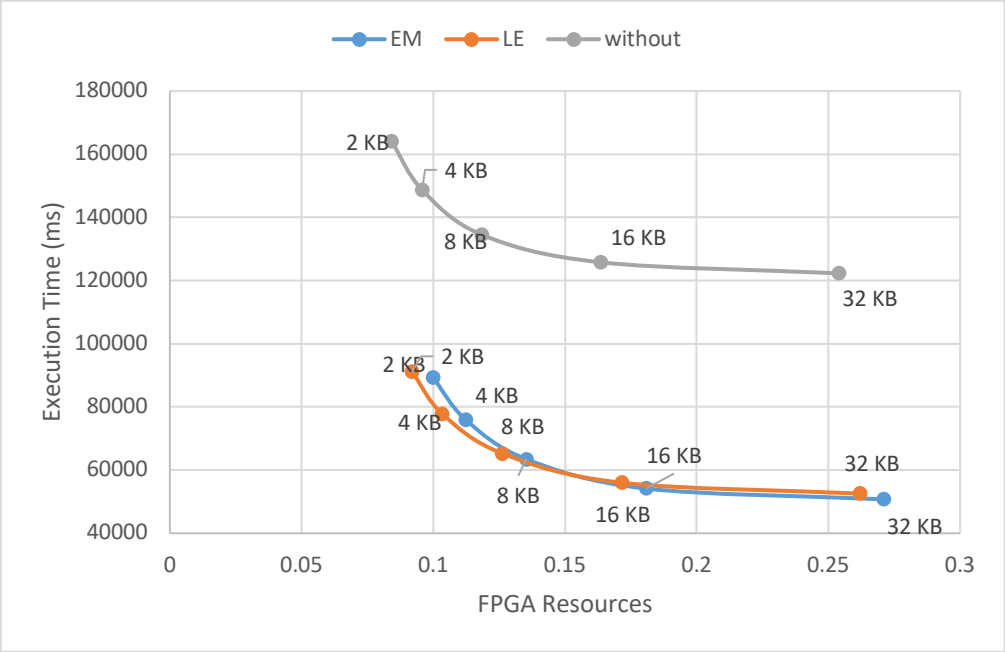

Figure-10 Execution Time vs. FPGA Resources with Multiplier (Test Case 3)

With the information available, the following points can be observed.

1) Apparently, the execution time is shorter with multiplier than without, at the expense of more resources used.
2) EM uses approximately 1% more resources than LE for every cache size.
3) The latencies of LE and EM are very close to each other across different test cases at each cache size, but EM is a bit faster.

   Since floating-point arithmetic is used in this case, the significand is too long to take advantage of LUT-based multiplier. Therefore, the dedicated embedded multiplier, which is efficient in computing fixed word-length, is expected to perform better.
4) The system with LE or EM performs better and better than that without as the instruction cache size increases. This speed-up effect can be revealed from the two sets of numbers: $\frac{latency\ of\ no\ multiplier}{latency\ of\ LE}$ (Ratio 1) and $\frac{latency\ of\ no\ multiplier}{latency\ of\ EM}$ (Ratio 2) for every test case.

   There is a clear trend that both indicators increase with the cache size as shown in **Figure-11, 12, 13** (data from **Table-8** in Appendix). After the multiplier completes one operation, it needs to wait for the cache to fetch the next part of instructions but does nothing in this period of time. As the cache size increases, the total delay that the multiplier has to wait becomes shorter making it work more efficiently, which can be seen as a bonus in addition to the ordinary acceleration due to the increase of cache size, but the computation speed of multiplier itself remains unchanged.

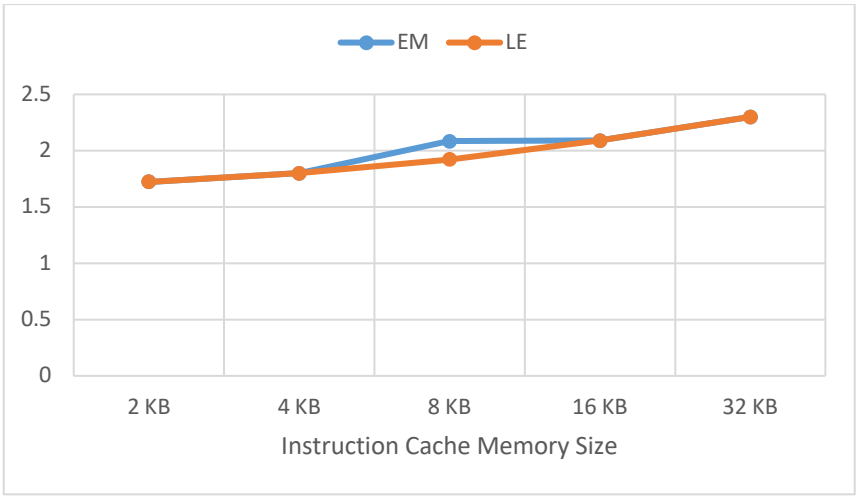

Figure-11 Ratio 1 & 2 (Test Case 1)

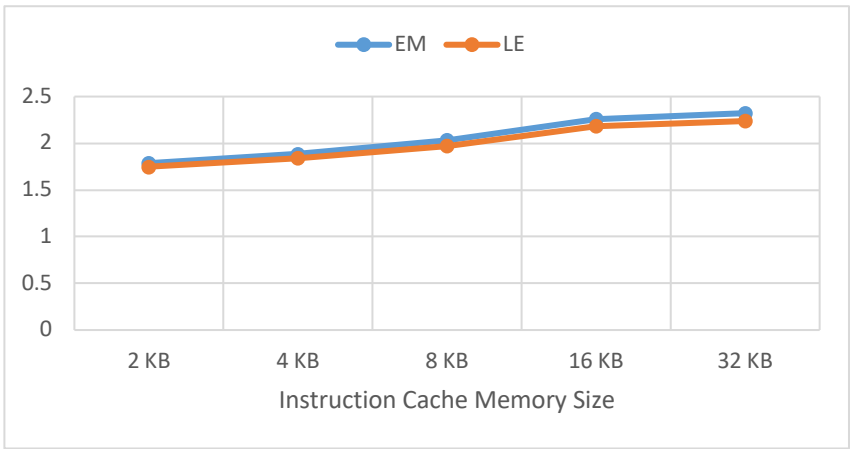

Figure-12 Ratio 1 & 2 (Test Case 2)

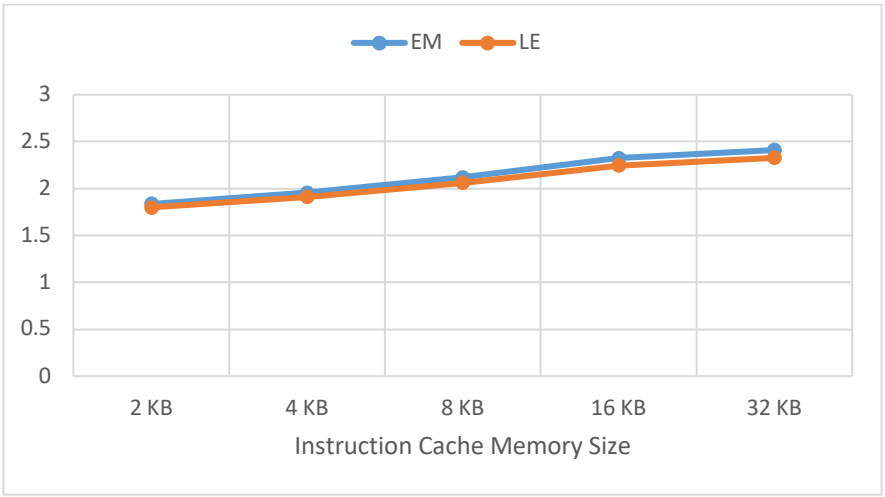

Figure-13 Ratio 1 & 2 (Test Case 3)

5) From the analysis above, it is worth noting that the difference between EM and LE also becomes wider along with the increase of cache, which seems like EM becomes faster and faster compared with LUT-based multiplier, but the operating speed is actually fixed. Since EM is intrinsically faster than LE, one of the reasons could be that there is still space for EM to improve because the delay of EM is longer, but the slower LE is approaching the limit as cache grows larger. To sum up, the

improvement of EM is greater than that of LE with increasing cache size. That can also be represented by the increase of another indicator $\frac{latency\ of\ LE - latency\ of\ EM}{latencyc\ of\ EM} \times 100$ (Rate 1) especially for test case 2 & 3 as shown in **Figure-14** (data from **Table-9** in Appendix).

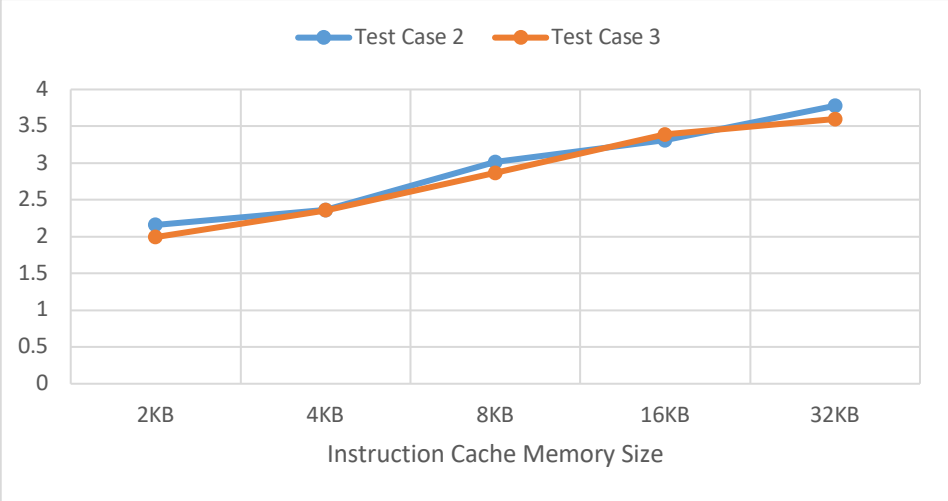

Figure-14 Rate 1

6) At some point beyond the maximum capacity of instruction cache that can be achieved at present, the acceleration due to the cache will saturate and stop. Based on the above analysis, it is justified to deduce that this point could be different for different cases: running with different multipliers and no multiplier.

V. Conclusion

Through the experiments done so far with all sorts of combinations of resources, the optimal design is the one with the support of SDRAM and embedded multiplier. Implementing off-chip memory generally reduces resource usage while the embedded multiplier largely improves the system's latency performance. At the same time, utilising 32 KB instruction cache gives minimal execution time.

VI. Reference

Donald Murray. (Feb 2013). *How does C compute sin() and other math functions?* [Online]. Available: https://stackoverflow.com/questions/2284860/how-does-c-compute-sin-and-other-math-functions

VII. Appendix

Instruction Cache Size	FPGA Resources (%)	Execution Time (ms)		
		Test Case 1	Test Case 2	Test Case 3
32 KB	25.39	23	922	122300
16 KB	16.36	23	955	125787
8 KB	11.84	25	1012	134518
4 KB	9.57	27	1118	148751
2 KB	8.42	31	1242	164085

Table-5 Summary of System without Multiplier (Task 4)

Instruction Cache Size	FPGA Resources (%)	Execution Time (ms)		
		Test Case 1	Test Case 2	Test Case 3
32 KB	27.09	10	397	50741
16 KB	18.08	11	423	54164
8 KB	13.53	12	498	63450
4 KB	11.22	15	593	76020
2 KB	9.99	18	695	89402

Table-6 Summary of System with Embedded Multiplier

Instruction Cache Size	FPGA Resources (%)	Execution Time (ms)		
		Test Case 1	Test Case 2	Test Case 3
32 KB	26.19	10	412	52567
16 KB	17.15	11	437	56000
8 KB	12.62	13	513	65268
4 KB	10.34	15	607	77813
2 KB	9.20	18	710	91184

Table-7 Summary of System with Logic Element

Cache	Latency of no Multiplier/Latency of EM			Latency of no Multiplier/Latency of LE		
	Test Case 1	Test Case 2	Test Case 3	Test Case 1	Test Case 2	Test Case 3
32 KB	2.3	2.322418	2.41028	2.3	2.237864	2.326555
16 KB	2.090909	2.257683	2.322336	2.090909	2.185355	2.246196
8 KB	2.083333	2.032129	2.120063	1.923077	1.97271	2.06101
4 KB	1.8	1.885329	1.956735	1.8	1.841845	1.911647
2 KB	1.722222	1.78705	1.835362	1.722222	1.749296	1.799493

Table-8 Ratio 1 & 2

Cache Size	(Latency of LE – Latency of EM)/Latency of EM		
	Test Case 1	Test Case 2	Test Case 3
32 KB	0	3.778338	3.598668
16 KB	0	3.309693	3.389705
8 KB	8.333333	3.012048	2.865248
4 KB	0	2.360877	2.35859
2 KB	0	2.158273	1.993244

Table-9 Rate 1