Digital System Design

Report 2

Michal Mysior (michal.mysior15@imperial.ac.uk)
& Jacob Kay (jacob.kay15@imperial.ac.uk)

In tasks 1 and 2, the on-chip ram was used as Nios II memory, which limited its capabilities and made it impossible to calculate test cases 2 and 3. This report describes the process of adding a controller for external SDRAM chip located on the DE0 board and evaluating system performance and resources utilisation when calculating more complicated mathematical expression, as well as with various hardware multipliers added.

Task 3: Storing the Program and Data on External Memories

Initially all the signals were connected and configured as instructed by the coursework booklet. For the first compilation, timing requirements were not met for 1200 mV 0C fast model; hold time was negative (i.e. a violation). However, the design is not used at those conditions (1200mV and 0^{O}C). Therefore, it was decided to ignore those warnings and proceed with the design. As for the PLL time shift, after a few trials, the system works with the value of -2.62 ns, which corresponds to the real value of -2.69 ns (as indicated by Quartus) due to finite precision of phase shift setting. Output 0 of PLL is used to drive Nios processor (with no phase shift), and output 1 to drive SDRAM memory controller (with negative phase shift, as mentioned earlier).

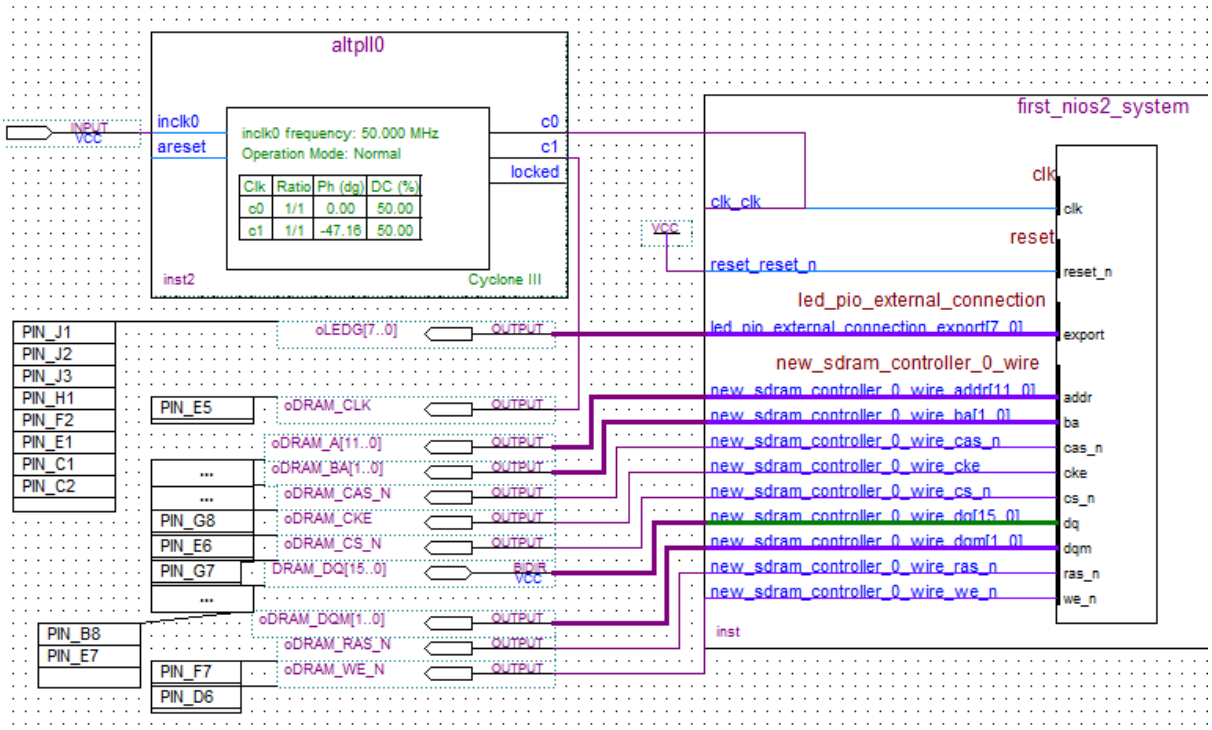


Figure 1: Block diagram for task 3

Having verified that the design is working, the performance of the system for the 3 test cases as was evaluated. Below are shown the results, all for 44212 bytes of RAM and 2KB of instruction cache. The error shown is compared to a reference MATLAB double implementation of the same code.

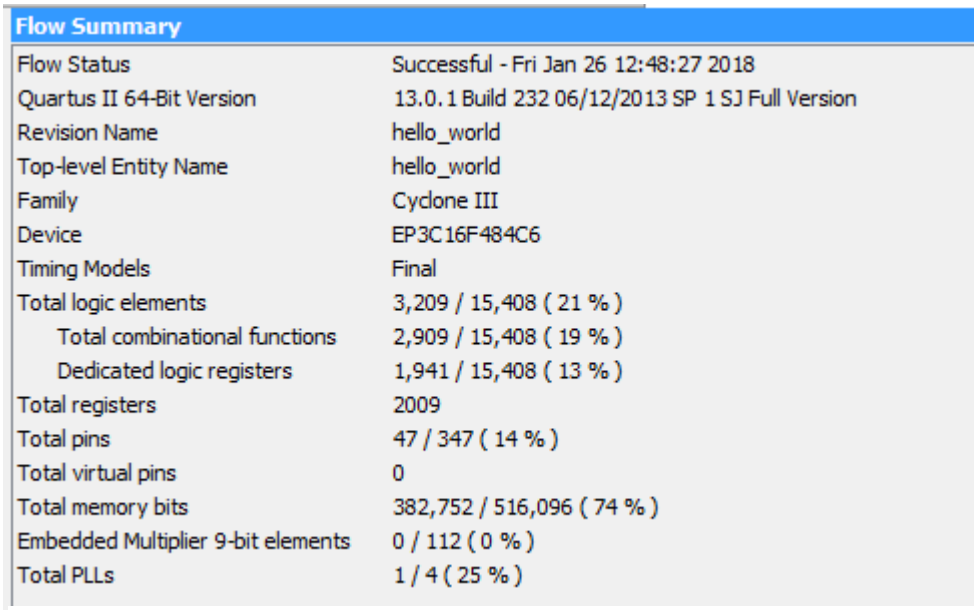
Test case	Latency (ms)	Output	MATLAB	Error	Program size	Memory free for stack + heap
1	4.814	1,144,780	1,144,780	0%	57KB	8131KB
2	256.51	55,630,384	55,629,019.249962	0.00245%	58KB	8131KB
3	26467	5,575,929,856	5,559,670,140.07228	0.292%	58KB	8131KB

Table 1: Performance measurement for 3 test cases after implementing SDRAM

Due to differences in time required to compute each test case, the latency was averaged over different numbers of runs. For test case 1 latency was averaged over 10,000 runs, for test case 2 over 1,000 runs and for test case 3 over just 1 run.

As can be seen, latency for test case 1 has increased by 54% compared to the use of on-chip RAM only (4.814 ms vs 3.1173 ms (task 2)), which is expected due to external memory being slower. Test cases 2 and 3 cannot be compared, as they could not be evaluated in previous exercises.
Moreover, it is clear that the Nios II application size does not change with different vector sizes; it is because larger integer value has little to no influence on the program size, and the vector itself is stored in RAM, not program memory.

The resources occupied by this system on FPGA can be seen below in figure 2.



Flow Summary	
Flow Status	Successful - Fri Jan 26 12:48:27 2018
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Full Version
Revision Name	hello_world
Top-level Entity Name	hello_world
Family	Cyclone III
Device	EP3C16F484C6
Timing Models	Final
Total logic elements	3,209 / 15,408 (21 %)
Total combinational functions	2,909 / 15,408 (19 %)
Dedicated logic registers	1,941 / 15,408 (13 %)
Total registers	2009
Total pins	47 / 347 (14 %)
Total virtual pins	0
Total memory bits	382,752 / 516,096 (74 %)
Embedded Multiplier 9-bit elements	0 / 112 (0 %)
Total PLLs	1 / 4 (25 %)

Figure 2: Resource usage for design with SDRAM

This results in an FPGA resource metric of 0.317, an increase of 0.019 (6.38%) compared to using just on-chip RAM. This increase, mostly in the number of logic elements used, is caused mostly by the use of PLL block.

The last thing to measure for this task was the impact of increasing the instruction cache size on the required resources and achieved latency. The results of these measurements are presented below in table 2.

RAM	Cache size	Resources usage	Latency (ms)		
			1	2	3
44212	2KB	0.317	4.8140	256.51	26467
40000	4KB	0.306	4.0028	224.83	23109
20480	16KB	0.273	4.0028	212.98	21462

Table 2: Latency for different on-chip RAM and cache memory sizes

As the cache memory was increased, the amount of on-chip RAM had to be decreased to accommodate all the blocks on the FPGA. Figure 3 below depicts latency of the system, for test case 3 and for different cache memory sizes (2, 4 and 16 KB):

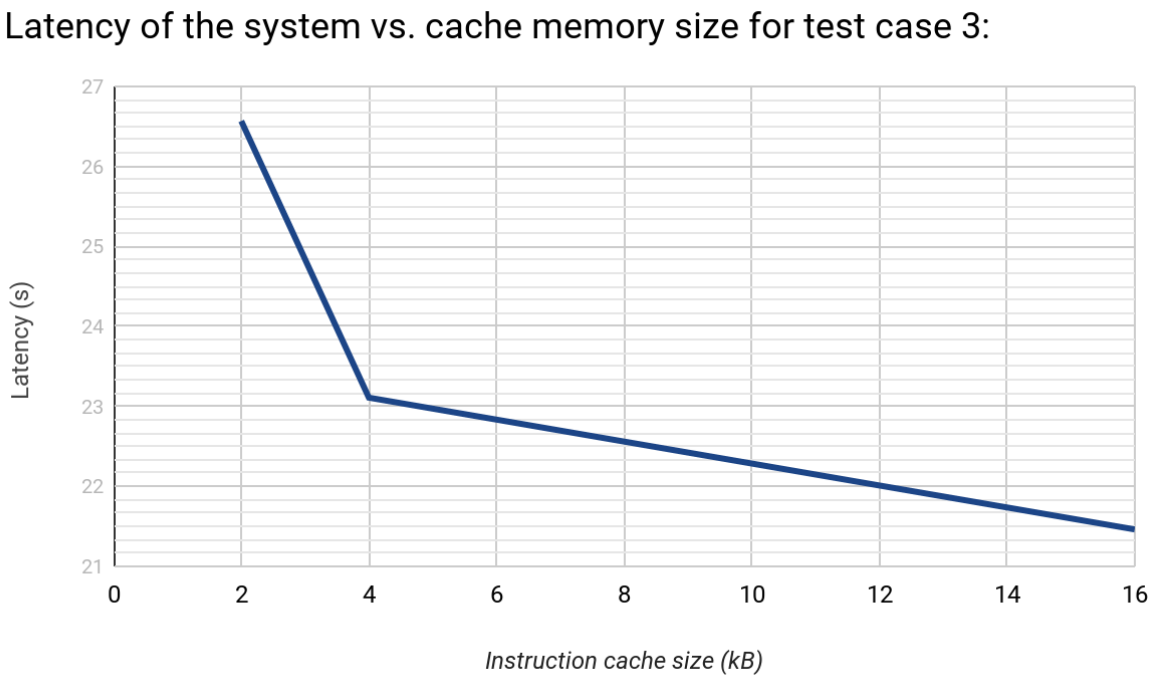


Figure 3: Latency vs. instruction cache size for test case 3

It is clear that the latency decreases greatly (by 3.358s or 12.6%) when cache is increased from 2KB to 4KB, and only slightly (by 0.647s or 2.8%) when cache is increased from 4KB to 16KB. Similar behaviour can be observed for test cases 1 and 2 by looking at Table 2. This is because initial increase in cache size (2KB to 4KB) allows for a lot more of the loop to fit in and reduces the amount of reads from SDRAM, thus greatly improving performance. A further increase, from 4KB to 16KB, allows for more instructions to be stored but now the limiting factor is the speed of getting data from the SDRAM to the Nios II. For test case 1, the full loop fits into 4KB so there are no gains from increasing to 16KB. A possible speedup would be to use data cache.

The next aspect worth looking at is the use of resources as the instruction cache size is increased. Figures 4 and 5 show this.

Flow Summary	
Flow Status	Successful - Mon Jan 29 12:29:03 2018
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Full Version
Revision Name	hello_world
Top-level Entity Name	hello_world
Family	Cyclone III
Device	EP3C16F484C6
Timing Models	Final
Total logic elements	3,208 / 15,408 (21 %)
Total combinational functions	2,914 / 15,408 (19 %)
Dedicated logic registers	1,942 / 15,408 (13 %)
Total registers	2010
Total pins	47 / 347 (14 %)
Total virtual pins	0
Total memory bits	366,720 / 516,096 (71 %)
Embedded Multiplier 9-bit elements	0 / 112 (0 %)
Total PLLs	1 / 4 (25 %)

Figure 4: Resource usage for 4KB instruction cache and 40,000 byes on-chip RAM

Flow Summary	
Flow Status	Successful - Mon Jan 29 13:28:18 2018
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Full Version
Revision Name	hello_world
Top-level Entity Name	hello_world
Family	Cyclone III
Device	EP3C16F484C6
Timing Models	Final
Total logic elements	3,179 / 15,408 (21 %)
Total combinational functions	2,880 / 15,408 (19 %)
Dedicated logic registers	1,943 / 15,408 (13 %)
Total registers	2011
Total pins	47 / 347 (14 %)
Total virtual pins	0
Total memory bits	315,904 / 516,096 (61 %)
Embedded Multiplier 9-bit elements	0 / 112 (0 %)
Total PLLs	1 / 4 (25 %)

Figure 5: Resource usage for 16KB instruction cache and 20,480 bytes on-chip RAM

Due to decreasing the RAM size when cache size was increased, the resources used actually shrank for higher cache sizes.

At this point it was decided that the use of on-chip RAM was not necessary; therefore, it was removed, and the design was compiled and tested for 3 different cache memory sizes. Table 3 below presents latencies for 3 test cases when no on-chip RAM is used.

Cache size	Resources usage	Latency (ms)		
		1	2	3
2KB	0.084	4.0528	226.53	23159
4KB	0.096	4.0529	226.23	23160
16KB	0.164	4.0027	214.17	21529

Table 3: Latency with no on-chip RAM and various instruction cache sizes

Comparing this data with Table 2., it can be observed that the latency is only slightly different - lower for 2KB of cache, and marginally higher for 4KB and 16KB. At the same time, resources usage is much lower. For this reason, it can be concluded that not using an on-chip RAM is a good design choice.

Figure 6 below shows latency for different resource usages as % of latency for 2KB cache memory and 0.084 resource usage (according to suggested metric).

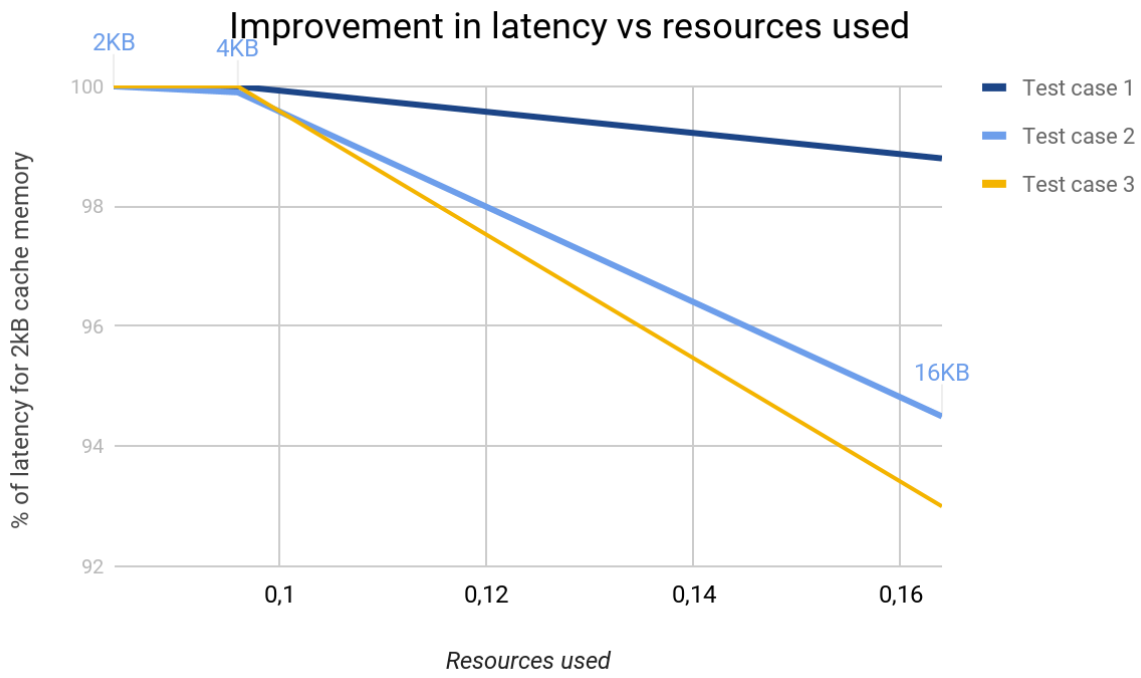


Figure 6: Comparison of latency improvements and resource usage (via instruction cache changes)

Similarly, the improvement is most significant for test case 3. As such most graphs focus on the latency of test case 3.

Task 4: Evaluating a more complex mathematical expression

The aim of task 4 was to compute a more complicated mathematical expression, now that the Nios II had enough memory to use the math.h library and compute all test cases. The equation given for implementation is:

$$\sum_{i=1}^{N} 0.5x_i + x_i^2 cos(floor(\frac{x_i}{4}) - 32)$$

The evaluation of this equation for the test vectors requires the SDRAM added in task 3. This is because of the extra stack requirements from computing cosine.

Initial implementations of task 4 used the **cos** and **floor** functions defined in math.h. This resulted in abnormally high execution times. It was discovered that the C99 math.h header file that was being used, did not overload its functions. This meant when **cos** and **floor** were used, they expected a double input and gave a double as an output. This caused a performance penalty as the Nios II had to convert the input to the cosine function from a float to a double, and then the output had to be cast to a float before the sum could be computed.

```
float sumVector(float x[], int M)
{
    float sum = 0.00;
    int i;
    for(i = 0; i<M; i++){
        sum = sum + ((0.5*x[i]) +(x[i]*x[i]*cosf(floorf(x[i]/4)-32)));
    }

    return sum;
}
```

Figure 7: Final sumVector function implemented on Nios II for task 4

Utilising the float specific **cosf** and **floorf** removed this performance penalty (without needing more resources) as these functions take a float as an input and returns a float as an output. This increased the performance of the system by a factor of ~2.3x. Table 4 below summarises the latency improvements from using the float specific functions, for each test case. Latencies are in ms.

Instruction Cache	Instructions Used	1	2	3
4KB	cos/floor	177.60	8780.2	883364
4KB	cosf/floorf	75.16	3730.0	376092
% of double-type functions		42.3%	42.5%	42.6%

Table 4: Improvement in latency from changing double functions to float functions

Task 4 performance was computed for a variety of instruction cache sizes, to compare latency and resource usage.

Instruction Cache	1	2	3	Resources
2KB	79.38	3954.9	397406	0.08414
4KB	75.16	3730.0	376093	0.09575
8KB	65.56	3271.7	327182	0.11842
16KB	62.54	3107.3	310757	0.16374
32KB	61.77	3068.1	306776	0.25315

Table 5: Latency and resources for various instruction cache sizes for task 4

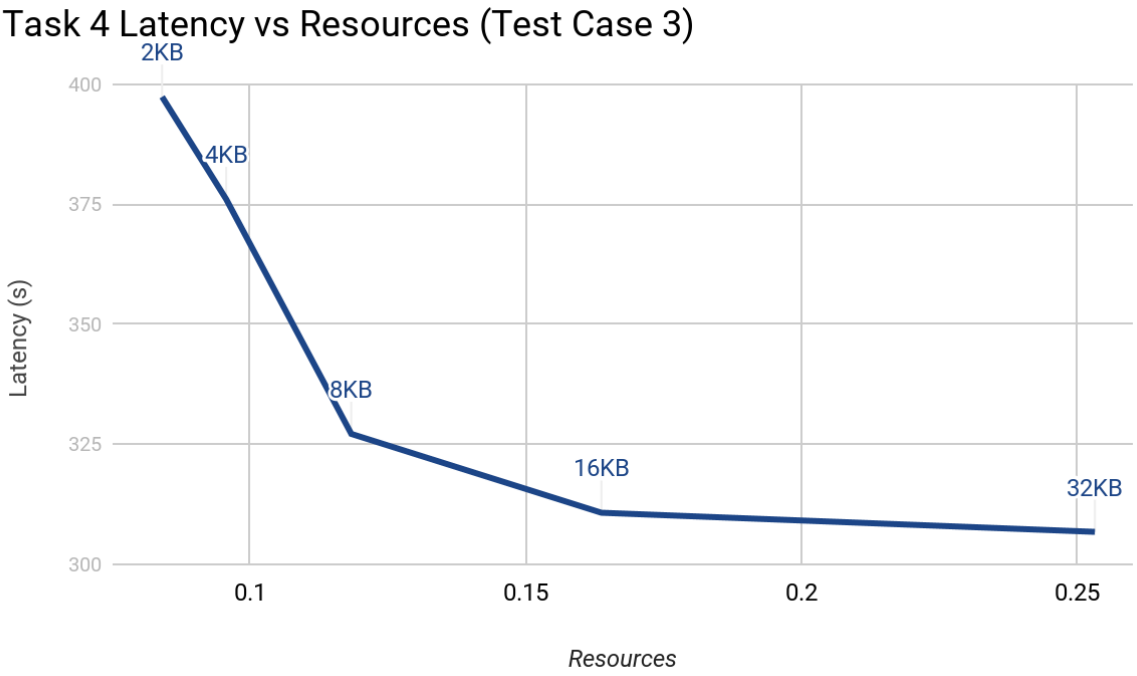


Figure 8: Comparison of latency and resource usage for various instruction cache sizes

This was then plotted in figure 8. This compares the latency of the system and the resources used. Since the aim of the project is to minimise latency and resources used, an optimal choice will be the one closest to the origin. So for task 4, the optimal configuration is to utilise 8KB of instruction cache. This is down to the fact that even though going beyond 8KB of instruction cache speeds up the computation; the improvements in latency are diminishing. The improvements in latency are down to more of the loop fitting into the fast instruction cache, meaning fewer requests for instructions from the SDRAM. The diminishing returns are down to a lack of data cache, as the data required for the computation is only stored on the slow to access SDRAM.

A note on numerical accuracy of floating point numbers in task 4

```
%% Test case 2
step = 0.1;
N = 2551;
input = zeros(1,N);
%Generate
for i = 2:N
    input(i) = input(i-1) + step;
end
%Task4+ function
result = 0;
for i = 1:N
    result = result + ( (0.5*input(i)) + (input(i)*input(i)*cos(floor(input(i)/4)-32)));
end
```

Figure 9: Implementation of task 4 maths in MATLAB

Utilising single precision floating point numbers to implement this equation limited the accuracy of the final answer. Table 6 shows the values for the equation as computing by the Nios II and MATLAB. The MATLAB values are from a double precision implementation (figure 9) of the same code used on the Nios II.

Test Case	MATLAB	Nios II	% Error
1	57879.8729731121	57879.867188	9.995E-06%
2	-126818.141927063	-76973.640625	-39.30%
3	-12774366.2941997	37022468	-389.82%

Table 6: Comparison of numerical accuracy in task 4

The error for test case 1 is minimal, and this is due to the fact each element of the input vector can be represented perfectly as a float. This is because the step is an integer. So the error mainly comes down to evaluating the cosine function and the result it outputs.

For test cases 2 and 3, the error is massive. This is down to the accuracy limitations of floating point numbers, but mainly a by-product of how the input vector is generated. Firstly, not all numbers in the input vectors can be represent perfectly as floating point numbers, this is just a limitation present in floating point numbers; especially single precision floats. They can only store 23 bits of significand.

The cascaded errors caused by the way the input vector is generated are responsible for most of the difference between Nios II and MATLAB. These errors are caused by the way the **generateVector** function is defined. It generates the current value of the array by adding the fixed step to the previous value. This would be perfectly fine if floating point numbers were infinitely precise. However, if the previous value could not be perfectly represented as a floating point number, adding the step to it means the current value contains that inaccuracy. Furthermore, if this

new number can't be stored exactly then the error grows. Table 7 shows this. Test case 2 has 2551 elements, and test case 3 255001 elements.

Element	Test Case 2		
	Expected	MATLAB	Nios II
0	0	0	0
18	1.8	1.8	1.8
1372	137.2	137.199999999996	137.199173
2550	255	254.99999999999	255.006363
	Test Case 3		
	Expected	MATLAB	Nios II
0	0	0	0
18	0.018	0.018	0.018
1372	1.372	1.37199999999996	1.372008
2550	2.550	2.54999999999983	2.54998
19905	19.905	19.9050000000013	19.905689
201697	201.697	201.697000000599	202.159653
255000	255	255.000000000854	255.840042

Table 7: Comparison of values generated by Nios II and MATLAB for task 4

As can be seen, the errors accumulate. What's important to note is that even MATLAB has some errors when using doubles using this **generateVector** implementation. However, MATLAB's numbers are much closer to the desired values. This is problematic in the function given to implement, as the **floor(x/4)** part is sensitive to small variations. For example, if x is meant to be 3.999, **floor(3.999/4)** should evaluate to 0. But let's say it's stored as 4.00001, **floor(4.00001/4)** will evaluate to 1. Since this is an input to a trigonometric function, this will cause an error in the answer. Small errors in the input to the floor function will result in a bigger error at its output, and since this goes into a cosine the error at its output will be large (large in respect to the initial error). For example **cosf(1)** = 0.540302 and **cosf(2)** = -0.416147. So a small shift in input to the cosine function will massively effect the overall sum.

```
void generateVector(float x[N] )
{
    int i;
    x[0] = 0;
    for (i=1; i<N; i++){
        //x[i] = x[i-1] + step;
        x[i] = i*step;
    }
}
```

Figure 10: Alternative generateVector definition

```
format longG;
format compact;
step = 0.1;
N = 2551;
input = zeros(1,N);
%Generate
for i = 2:N
    input(i) = (i-1)*step;
end
result = 0;
for i = 1:N
    result = result + ( (0.5*input(i)) + (input(i)*input(i)*cos(floor(input(i)/4)-32)));
end
result
```

Figure 11: MATLAB implementation of the alternative definition

A way around this accumulation of errors, is to define **generateVector** as shown in figures 10 and 11 above. This forces the only errors to be due to the limited precision of single precision floats, i.e. some elements can't be represented perfectly in a float. Using this definition, the following outputs are generated from the Nios II and MATLAB:

Test Case	MATLAB	Nios II	% Error
1	57879.8729731121	57879.867188	9.995E-06 %
2	-68594.2079648701	-68595.390625	-0.001724%
3	-12767828.5041377	-12767618	-0.001649%

Table 8: Comparison of numerical answers with alternative definition of generateVector

What's interesting to note is that the MATLAB answer for test case 2 and 3 are different in this alternative implementation. This implies even a double has issues storing required elements accurately for task 2. However, for the purposes of this report, the original generateVector method is used for further tasks. This is to allow for consistency between reports. It is important to keep in mind that single precision floating point numbers can be used accurately if care is taken with the generation of their inputs.

Cosine Implementation

As noted previously, the built in math.h library has multiple functions for calculating cosine. These are dependent on the type of input; float, double, long double – **cosf**, **cos**, **cosl**. These are most likely implemented using a CORDIC (Coordinate Rotation Digital Computer) algorithm. This is an efficient algorithm that converges one bit per iteration. This algorithm is what is usually used in microcontrollers and FPGA's. An alternative is to use Taylor series to evaluate the cosine function. Another option is a simple look up table, but this comes. Implementation of cosine will be looked at in more detail in the next report, as task 7 involves creating a hardware implementation for cosine.

Task 5: Adding Multiplier Support

Task 5 involved comparing the performance (latency vs resources) when hardware multiplication was enabled. Until this point, all multiplication had been done using fixed point adders and shifting. Hardware multipliers allow the Nios II's arithmetic logic unit to use dedicated hardware to speed up fixed point (integer) multiplication. The two options available are Embedded Multipliers, and Logic Elements. Embedded multipliers utilise the 9 by 9 bit hardware multipliers present on the FPGA. Logic element based hardware multiplying uses logic elements built up to emulate a hardware multiplier, and hence are slower.

It is important to note that these hardware multipliers only do integer multiplication, and not floating point. Floating point multiplication is done via software mapping the floating point multiplication to a sum of exponents and a product of the mantissa with appropriate shifting (and adding/subtracting to the exponent) and rounding. This results in a 23 by 23 bit integer multiplication. Shifting without hardware multipliers is done one bit per clock cycle. This is sped up via the hardware multipliers.

In task 4 it was decided 8KB of instruction cache is optimal. However, task 5 was thought to run faster, and as such there might be a different optimal instruction cache size.

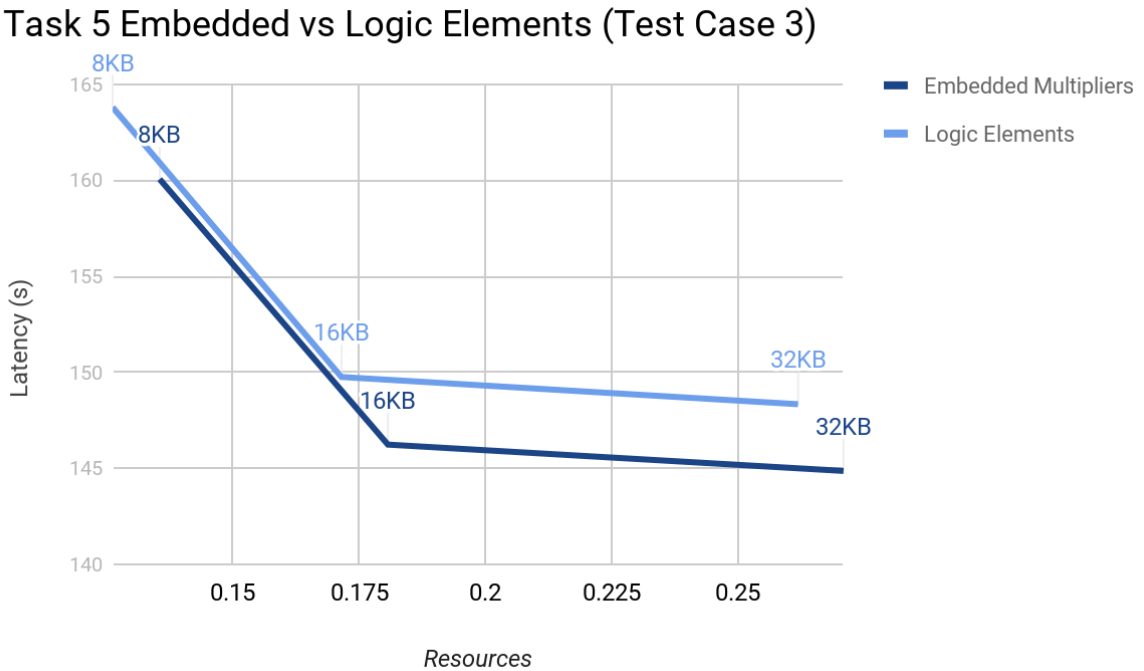


Figure 12: Latency vs. resources for various instruction cache sizes and EM and LE

Therefore, a quick comparison was run for 8KB,16KB, and 32KB of instruction cache. From figure 12, 16KB of instruction cache is the optimal choice for task 5. As it is closer to the origin than either 8KB or 16KB. This might be down to the CPU taking less time to run the instructions, and as such benefitting from more

instructions being present in the fast instruction cache. This means the CPU stalls less while waiting for instructions. Once again 32KB of cache lead to diminishing gains in latency.

For detailed comparison, 16KB of instruction cache was used as it was noticed to be optimal. Table 9 below presents the results for 3 test cases and 3 different hardware multiplication methods. The multiplication method had no influence on the numerical accuracy of the results, so the errors are identical as in task 4 and therefore not mentioned in table 9.

H/W Multiplication	Latency (ms), for test case:			Program size	Free for heap + stack	Resource utilisation
	1	2	3			
Embedded Mult.	29.51	1462.5	146238	71KB	8113KB	0.180
Logic Elements	30.22	1497.7	149759	72KB	8113KB	0.172
None	62.55	3107.3	310756	72KB	8112KB	0.164

Table 9: Latency, program size, and resource usage for different test cases and resource utilisation

It is clear that using hardware multiplication improves latency greatly; to better visualise this, figure 13 below presents % change in latency for all test cases, where no hardware multiplication latency is 100%.

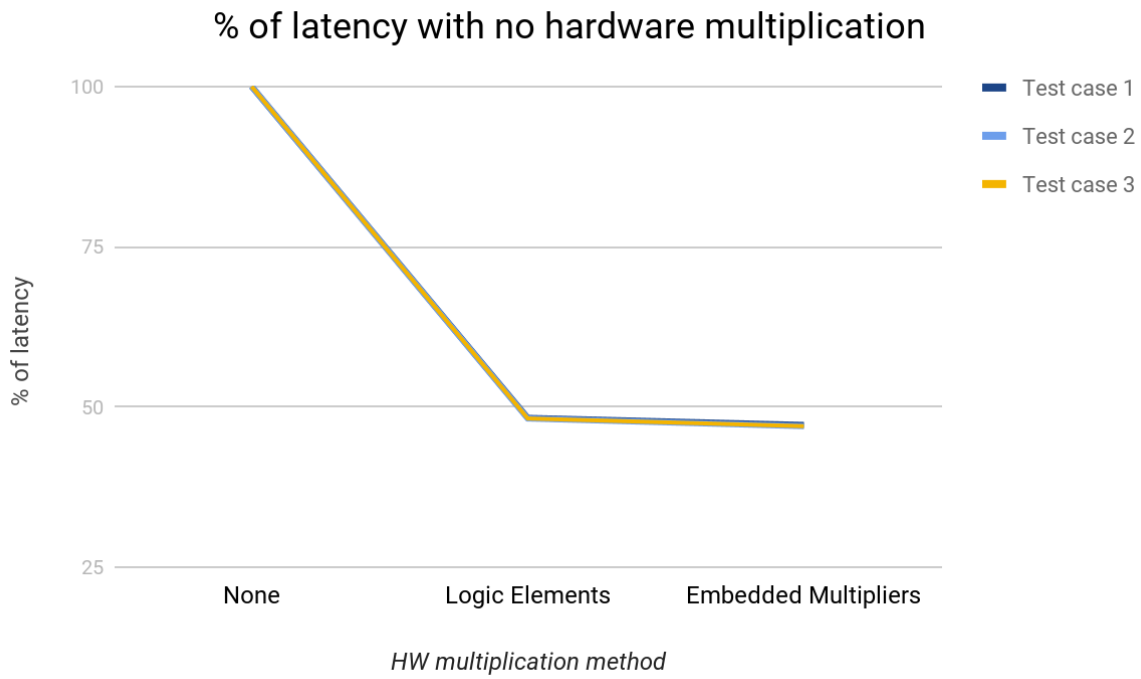


Figure 13: Change in latency for different hardware multiplication methods

As can be observed, the change is virtually identical for all 3 test cases, and most visible between no hardware multiplication and logic elements/embedded multipliers

(over 50% improvement); the difference between the use of logic elements and embedded multipliers is almost marginal.

Finally, the relation between resource usage and performance should be considered. Those are depicted in figure 14 below.

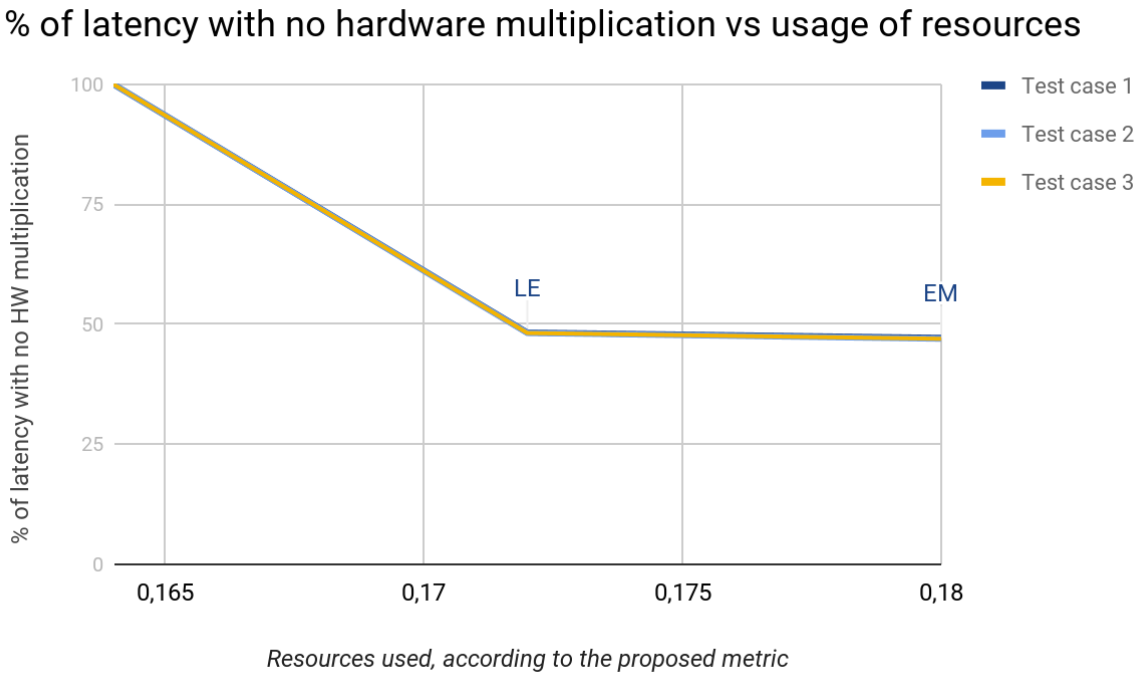


Figure 14: Comparison of latency vs. resource usage (for No Mult, LE, EM)

It can be seen that for task 5, the optimal solution is utilising logic elements. The latency improvements from no hardware multipliers vastly outweighs the resource cost. Embedded multipliers are marginally faster (~2.4%) than logic elements, the resource penalty (using the defined metric) outweighs the latency gain.

Assuming that maximum performance is required, the best design choice is to use embedded multipliers as they allow the design to achieve lowest latency. However, the difference marginal compared to the use of logic elements as a hardware multiplication method. If maximum performance is desired, with no consideration for resource usage, maximum instruction cache should be used (32KB)

Conclusion

Removal of on-chip memory in task 3 reduced resource usage, while maintaining performance. Implementation of the desired equation in task 4 brought up interesting issues about the accuracy and precision of floating point numbers. Implementing a more complicated equation led to a slow down. Utilising hardware multipliers in task 5 increased performance by a factor of 2 with minimal extra resources. Numerical accuracy was not affected. The system derived in task 5 is a good starting point for the later tasks. With more time, implementing either data cache (through use of the Nios II/f), tightly coupled memory, or on-chip RAM (with code optimisation to ensure critical instructions and data is on it), would have most likely improved performance. The next steps are to speed up the computation through building dedicated hardware blocks (with custom instructions).