Digital Systems Design Coursework - Report 2

Group 3

1st Kwok Tsz Wing
01708991
twk119@ic.ac.uk

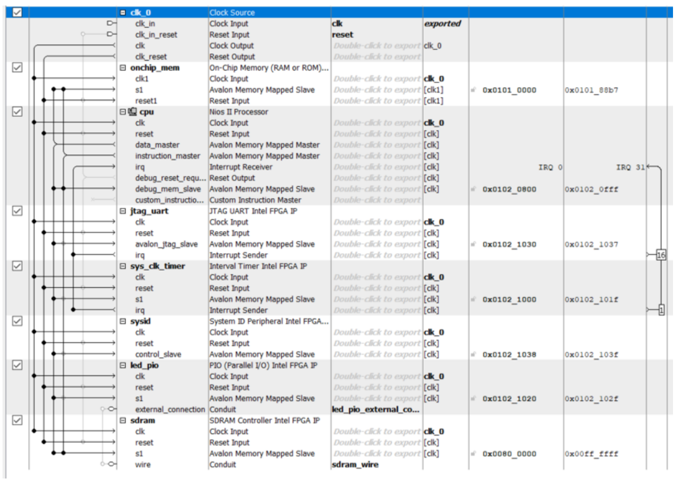
2nd Josiah Mendes
01760165
jam419@ic.ac.uk

I. Introduction

The following report discusses Task 3 to Task 5 assigned in the coursework specification for Digital Systems Design. The following investigation was carried out using Intel Quartus Lite 20.1, on a Terasic DE-1 SoC board with a Cyclone V FPGA.

II. Task 3 - Using SDRAM

As mentioned in the previous report, only using on-chip memory composed of FPGA logic elements results in a small amount of usable memory. It is insufficient to compute the simple mathematical function $y = \sum_{i=1}^{N} x_i + x_i^2$ with $N = 26121$. This particular case would require around 1MB of memory, and therefore this task introduces the use of off-chip memory and SDRAM.



clk_0	Clock Source					
clk_in	Clock Input	**clk**	***exported***			
clk_in_reset	Reset Input	**reset**				
clk	Clock Output	*Double-click to export*	clk_0			
clk_reset	Reset Output	*Double-click to export*				
onchip_mem	On-Chip Memory (RAM or ROM)...					
clk1	Clock Input	*Double-click to export*	**clk_0**			
s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk1]	0x0101_0000	0x0101_88b7	
reset1	Reset Input	*Double-click to export*	[clk1]			
cpu	Nios II Processor					
clk	Clock Input	*Double-click to export*	**clk_0**			
reset	Reset Input	*Double-click to export*	[clk]			
data_master	Avalon Memory Mapped Master	*Double-click to export*	[clk]			
instruction_master	Avalon Memory Mapped Master	*Double-click to export*	[clk]			
irq	Interrupt Receiver	*Double-click to export*	[clk]		IRQ 0	IRQ 31
debug_reset_requ...	Reset Output	*Double-click to export*	[clk]			
debug_mem_slave	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0102_0800	0x0102_0fff	
custom_instructio...	Custom Instruction Master	*Double-click to export*				
jtag_uart	JTAG UART Intel FPGA IP					
clk	Clock Input	*Double-click to export*	**clk_0**			
reset	Reset Input	*Double-click to export*	[clk]			
avalon_jtag_slave	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0102_1030	0x0102_1037	
irq	Interrupt Sender	*Double-click to export*	[clk]			16
sys_clk_timer	Interval Timer Intel FPGA IP					
clk	Clock Input	*Double-click to export*	**clk_0**			
reset	Reset Input	*Double-click to export*	[clk]			
s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0102_1000	0x0102_101f	
irq	Interrupt Sender	*Double-click to export*	[clk]			1
sysid	System ID Peripheral Intel FPGA...					
clk	Clock Input	*Double-click to export*	**clk_0**			
reset	Reset Input	*Double-click to export*	[clk]			
control_slave	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0102_1038	0x0102_103f	
led_pio	PIO (Parallel I/O) Intel FPGA IP					
clk	Clock Input	*Double-click to export*	**clk_0**			
reset	Reset Input	*Double-click to export*	[clk]			
s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0102_1020	0x0102_102f	
external_connection	Conduit	**led_pio_external_co...**				
sdram	SDRAM Controller Intel FPGA IP					
clk	Clock Input	*Double-click to export*	**clk_0**			
reset	Reset Input	*Double-click to export*	[clk]			
s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0080_0000	0x00ff_ffff	
wire	Conduit	**sdram_wire**				

Fig. 1. System Components and Connections within the Platform Designer

The DE1 SoC Board comes with a 64Mb SDRAM chip. In order to utilise it, a SDRAM controller is necessary to control transactions between the processor and the SDRAM, this is due to the large amount of protocols to be followed in order to address each memory location on off-chip memory. Intel provides a SDRAM Controller Core with an Avalon Memory-Mapped interface within the Platform Designer in Quartus up to v20.1, and so this was chosen and added to the system (shown in Figure 1). This SDRAM Controller acts as an interface between the main system on FPGA fabric and the off-chip SDRAM.

As the SDRAM is not located on the FPGA chip itself, there is a signal propagation delay that is also influenced by the timing characteristics of the SDRAM chip. These delays both for access time and output hold times affects the timing characteristics for the overall system. Therefore, the SDRAM clock is phase shifted with respect to the system clock so that it leads the system clock, ensuring that the output signals of the SDRAM arrive in time to meet timing constraints. This phase difference is generated using a PLL block from Intel's IP library.

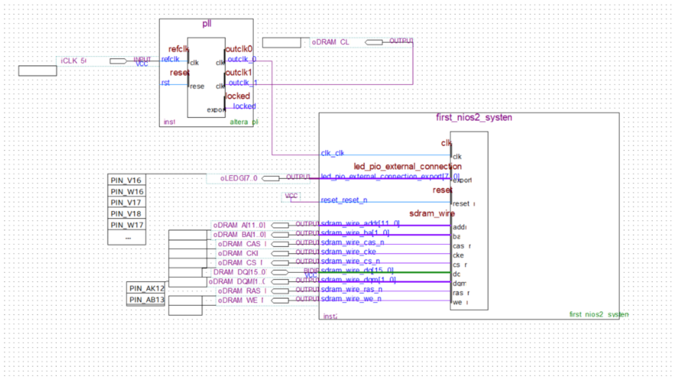


Fig. 2. Block Level Description of System on FPGA Fabric

The PLL block has two outputs (`outclk0` and `outclk1`), where the first output has no phase difference and is connected to the main system (`first_nios2_system`) and `outclk1`, with a –46 degrees phase shift, is connected to the SDRAM clock input.

A. Function Performance

With the inclusion of SDRAM, the available memory space has increased to 8 megabytes and it is now possible to use the Nios II processor initialised in the previous task to compute Equation 1 for all three different cases shown in Table I.

$$y = \sum_{i=1}^{N} x_i + x_i^2 \tag{1}$$

As there are no longer memory constraints, a reduced size C library is no longer used. The program size of the

TABLE I
VECTOR SIZE IN EACH CASE

		Number of Elements	Size (Bytes)
Case 1	[0:5:255]	52	208
Case 2	[0:1/8:255]	2041	20164
Case 3	[0:1/1024:255]	261121	1044484

elf files is increased to around 75kB from 12kB, obtained during compilation. This also opens up possibilities to import large libraries such as `math.h` and also utilising the full `printf` function. Although each case has different vector sizes, the program size is consistent across all test cases as the vectors are declared during run-time and stored in the stack as observed in the previous report.

The performance for each case where the system has a 4kB instruction cache, 2kB data cache and no compiler optimisations is shown in Table II. The timestamp function was used to collect timing over 10 runs, with a resolution of 50MHz.

TABLE II
FUNCTION 1 PERFORMANCE FOR EACH CASE

	Average Tick	Average Time (us)	Standard Deviation (us)
Case 1	32686.8	653.736	0.669
Case 2	1308830.8	26176.616	1.504
Case 3	194212239	3884244.78	22.192

It was observed that the execution time increases compared to previous tasks. It is suspected that with the introduction of off-chip memory, the average memory access time increases. SDRAM has a much higher CAS latency, 3 clock cycles compared to 1 clock cycle for on-chip memories. Caching can be used to hide memory access latency by “preloading” the data. However, with Nios II not supporting out-of-order execution, the processor has to stall when there is a cache miss, leading to an increase in run-time. This becomes more apparent for case 3 where majority of the elements are stored off-chip.

B. Cache Size

With a larger problem size and changes to memory access, it was thought necessary to re-evaluate the cache size choices. A larger cache size would lead to a higher cache hit rate, it would improve average memory access time and in turn performance.

Average time in Figure 3 refers to the average time across the three test cases. It was observed that performance increases significantly when the instruction cache size increases from 0kB to 2kB, but plateaus after that. Further increases in performance are obtained by increasing the data cache size, but this also has limited effect, with limited gains once reaching 2kB in size.

The function of our interest does not have a lot of branch and jump instructions, but traverses the data elements in the one-dimensional array. Having no cache would require the system to keep fetching from external memory which has

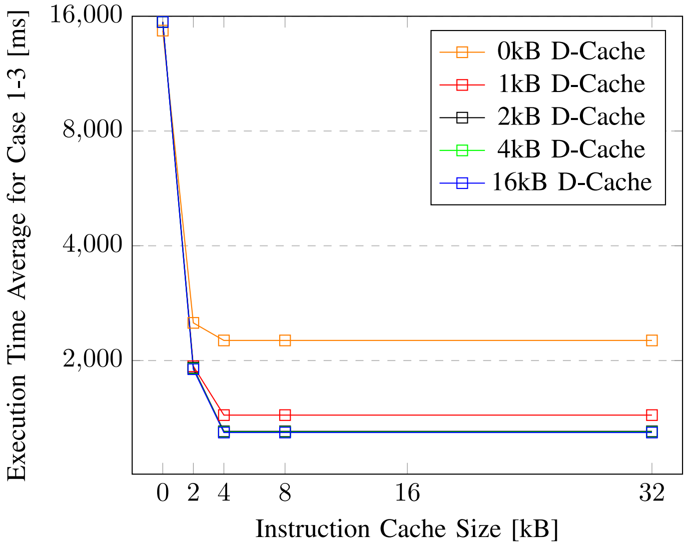


Fig. 3. Effect of Cache Size on Performance

high access time due to data being stored off-chip. Once the instruction cache is large enough to fit the instructions for each loop, further increases have little to no effect. It is also observed that the performance is not strongly affected by the data cache size. As only one `float` from memory is necessary in each loop iteration, a small data cache is more than capable of pre-fetching and fitting multiple elements to exploit spatial locality. The aggressiveness of the pre-fetching is also fixed and cannot be customised, so larger data cache sizes do not show a corresponding reduction in latency.

It is suspected that the bottleneck does not sit in instruction nor data fetch but in other factors such as current software implementation and this will be further discussed in the next section.

To minimize resources usage, the optimal cache size of the system remains as 2kB data cache + 4kB instruction cache with a resource score of 4.6064%[1]. The full resource utilisation is shown in Table III and the performance of all 3 cases for this system is recorded in Table II.

TABLE III
TASK 3 OPTIMAL SYSTEM FPGA[a] RESOURCE UTILISATION

Logic utilization (in ALMs)	1710 / 32070 (5%)
Total registers	2732
Total pins	47/457 (10%)
Total virtual pins	0
Total block memory bits	345024 / 4065280 (5%)
Total RAM Blocks	52 / 397 (11%)
Total DSP Blocks	0 / 87 (0%)
Total PLLs	1 / 6 (0%)
Total DLLs	0 / 4 (0%)

[a] The resource utilisation does not include the SDRAM chip as this is part of the board and cannot be removed.

[1]The Resource Percentage is obtained using the formula given in the specification: $\frac{1}{3}\left(\frac{\text{Multipliers Used}}{\text{Total Multipliers}} + \frac{\text{Memory Bits Used}}{\text{Total Memory Bits}} + \frac{\text{Logic Elements Used}}{\text{Total Logic Elements}}\right)$

III. Task 4 - A More Complicated Function

This task introduces the function (Equation 2) that is to be accelerated, and also provides a baseline performance insight into how fast the function can perform under software with invariant hardware across the three cases described in Table I.

$$f(x) = \sum_{i=1}^{N} 0.5x_i + x_i^2 \times \cos(\frac{x-128}{128}) \qquad (2)$$

Initially, a straightforward implementation using the `<math.h>` library implementation of `cos` was used and this is shown in Listing 1.

```c
float math_expr(float x[], int M){
  int i;
  float rtn = 0;
  for (i=0; i<M; i++){
    rtn += 0.5*x[i] + pow(x[i],2)
           *cos((x[i]-128)/128);
  }
  return rtn;
}
```

Listing 1: Unoptimised Implementation

A. Baseline System

The function is run on the default system as described in the previous section with a 4kB instruction cache and a 2kB data cache. Table III shows the resource usage obtained from the compilation of this design with a resource score of 4.6064%.

The performance of the 3 cases with the above code and system are recorded in table IV. The program size is 91kB and was consistent among all 3 cases.

TABLE IV
Function 2 Performance for Each Case

	Average Time (us)	Standard Deviation (us)
Case 1	27151.30	0.938
Case 2	1079754.32	0.453
Case 3	159718000.00	0.011

With careful look in the generated assembly code, it was observed that floating points are expanded to double-precision before arithmetic operations. Hence types are explicitly declared to force the expression be evaluated in single precision as shown in Listing 2.

The result obtained from this function will be the baseline system of this project, and accuracy of any future design should not be lower than Table V. A Python script that calculates in double-precision is written to compare the results.

B. C Optimisations

As suggested in previous section, the bottleneck of the system sits in the software implementation of the function. Optimisations will accumulate in the section, hence all comparisons would be relative to the previous sub section.

```c
float math_expr(float x[], int M) {
 int i;
 float rtn = 0;
 for (i=0; i<M; i++){
  float tmp = x[i];
  float tmp_val = tmp-128.0f;
  float tmp_pow = pow(tmp,2);
  tmp_val /= 128.0f;
  rtn += 0.5f*tmp;
  float tmp_cos = cos(tmp_val);
  rtn += tmp_pow * tmp_cos;
 }
 return rtn;
}
```

Listing 2: Single Precision Implementation

TABLE V
Function 2 Accuracy for Each Case

	Result	Error
Case 1	920413.50	1.3760e-7
Case 2	36123052.00	9.2882e-7
Case 3	4616063488.00	1.1740e-3

1) Power and Multiplication: It was suggested that it is more efficient to multiply the variable by itself rather than calling the `pow` function, especially since the equation of our interest has order of 2 only. This would mean substituting all occurrences of `pow(x[i], 2)` with `x[i] * x[i]`. As the current system does not have hardware multipliers, software routines are used to multiply both integer and floating point numbers. It is observed that `float_mult` function has less instructions, especially branch instructions, compared to `pow` function. This can be taken advantage of and hence leading to a better performance.

The hypothesis is further confirmed with an improvement in performance when executing the function, as shown in Table VI.

TABLE VI
Performance with Multiplication Optimisation

	Average Time (us)	Uplift
Case 1	26684.84	+1.71%
Case 2	1056951.21	+2.11%
Case 3	153946000.00	+3.61%

2) Use of Loop Constant: In the function, the length of array is passed in as a parameter. However, length is defined in a macro and is known at compile time, hence the variable can be replaced with the macro-defined constant. This could improve performance by having one less parameter passing into the function, reducing function call overhead.

In addition, the use of a constant for the loop encourages loop unrolling, which is a common software optimisation to improve performance. Loop unrolling decreases the average loop overhead per iteration by allowing more than one loop body to be executed before checking the conditions.

`-funroll-loops` compiler flag was used together with `pragmas` to ensure loop unrolling was performed, but the improvement was not significant as recorded in table VII despite the increase in program file size by 5kB.

TABLE VII
PERFORMANCE WITH CONSTANT ARRAY LENGTH

	Average Time (us)	Uplift
Case 1	26518.71	+0.62%
Case 2	1048755.72	+0.77%
Case 3	152614000.00	+0.87%

It is suspected that the bottleneck of the performance sits in the loop body, and execution time in the body is much larger than the loop overhead. Hence the impact of reducing overhead is not significant. However, it is anticipated that in future design when the loop body takes less instructions, the importance of loop unrolling can come into play.

C. *Cosine Optimisations*

As the cos function in the expression takes the largest proportion of the loop body in the generated assembly code, it was thought that it would be the main bottleneck in performance improvements. Hence different implementation of the cos function are explored. Performances recorded in this section are obtained with the two software optimisations and compared with the performance as stated in the previous section.

1) Use of cosf: `math.h` library in C provides both cos and cosf function where cos returns a double and cosf returns a float. As the accumulator (`rtn`) in the function is a float, using cosf would not decrease accuracy. With the function taking in and returning a float, it does not require additional float to double extension and double to float truncation. Each conversion takes 68 instructions, hence changing from cos function to cosf function improves the performance as shown in Table VIII.

TABLE VIII
PERFORMANCE AND ACCURACY OF cosf

	Average Time (us)	Uplift	Result	Error
Case 1	10345.722	+60.99%	920413.5	1.3760e-7
Case 2	409943.04	+60.91%	36123108	6.2143e-7
Case 3	56899010.71	+62.72%	4621532160	9.3351e-6

The cos function keeps in useful when a higher accuracy would like to be achieved. By having the accumulator (`rtn`) to be a `double`, and only type cast it before returning from the function as shown in Listing 3 would improve accuracy significantly as shown in Table IX.

TABLE IX
ACCURACY FOR EACH CASE WITH DOUBLE IMPLEMENTATION

	Result	Error
Case 1	920413.625	1.792e-9
Case 2	36123084.00	4.29636e-8
Case 3	4621489152.00	2.90196e-8

```c
float math_expr(float x[], int M)
{
 int i;
 double rtn = 0;
 for (i=0; i<M; i++){
  rtn += 0.5*x[i] + pow(x[i],2)*
    cos(((x[i]-128)/128));
 }
 return (float) rtn;
}
```

Listing 3: Double Precision Implementation

Although accuracy is not of main concern in current stage, in future development if accuracy needs to be improved, such optimisation can be implemented with a performance cost.

2) Taylor Series: Without any floating point arithmetic hardware in the system, the cosine function in the `math.h` C library is suspected to be implemented by a 6-term Taylor series which is described as equation 3 with n = 6, as observed through the generated assembly.

$$cos(x) = \sum_{k=0}^{n} \frac{(-1)^k}{(2k)!} x^{2k} \tag{3}$$

Instead of calling the library function, the cosine value can be calculated by explicitly implementing a Taylor series approximation of the cos function as shown in Listing 4. This would reduce function call overhead and also allow us to perform our own optimisations on the implementation.

```c
int factorial(int x) {
 float rtn = 1;
 for (int i=1; i<x+1; i++){
  rtn *= i;
 }
 return rtn;
}
float cos_taylor_6terms(float x) {
 float rtn = 0;
 for (int i=0; i<7; i++){
  rtn += pow(-1,i)*pow(x,2*i) /
    factorial(2*i);
 }
 return rtn;
}
```

Listing 4: Naive Implementation of Taylor Series

It was observed that with known `n`, the terms `pow(-1,i)` and `factorial(2*i)` are known at compile time. This can be simplified and expanded to compile-time constants instead of variables calculated in run-time as shown in Listing 5. This significantly reduces amount of computation needed and the number of branch instructions, hence improving performance significantly.

```
float cos_taylor_6terms(float x) {
 return 1 - ((x*x)/(2)) +
  ((x*x*x*x)/(24)) -
  ((x*x*x*x*x*x)/(720)) +
  ((x*x*x*x*x*x*x*x)/(40320)) -
  ((x*x*x*x*x*x*x*x*x*x)/(3628800)) +
  ((x*x*x*x*x*x*x*x*x*x*x*x)/
   (479001600));
}
```

Listing 5: Implementation of Taylor Series with Constants

The accuracy and performance of Taylor series approximation with 4 terms is also evaluated as it requires less computation and would be able to provide the result with lower latency. Although the accuracy performance of 4-term Taylor series is considerably poorer as magnitude of x increases, the error is small for $|x| < 3$ as depicted in Figure 4. For a 6-term approximation, the range where error is small increases to $|x| < 4.5$, while providing a slightly better approximation around $|x| < 1$. The performance and results for all three cases with both alternative $\cos$ implementations are recorded in Table X.

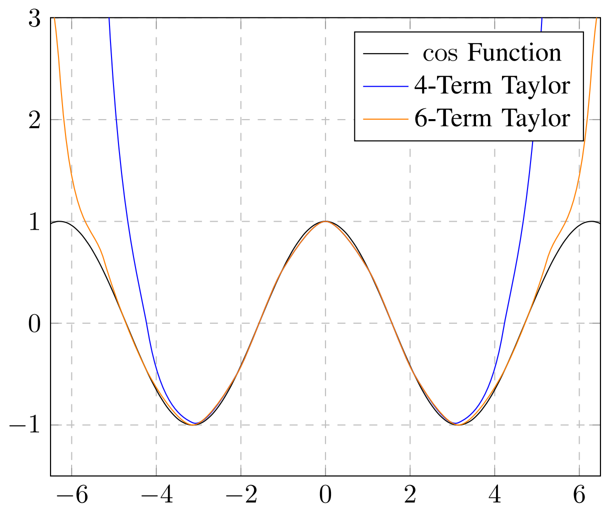


Fig. 4. Approximation of Cos Function using Taylor Series with 4 & 6 terms

TABLE X
PERFORMANCE AND ACCURACY OF TAYLOR SERIES COSINE APPROXIMATION

	Average Time (us)	Uplift	Result	Error
Case 1	17494.90	34.03%	920413.5	1.3760e-7
Case 2	700552.12	33.20%	36123108	6.2143e-7
Case 3	96107000.00	37.03%	4621532160	9.3351e-6

(a). 4 Term Taylor Series

	Average Time (us)	Uplift	Result	Error
Case 1	29268.332	-10.37%	920413.5	1.3760e-7
Case 2	1265346.36	-20.65%	36123104	5.1070e-7
Case 3	172628000	-13.11%	4621531136	9.1135e-6

(b). 6 Term Taylor Series

In the mathematical expression of our interest, the parameter passing into cos function is limited to [-1,1]. Hence, evaluating the cosine term with a Taylor series of 4 terms could produce a comparably accurate result, fulfilling the baseline accuracy stated in Section III-A with a massive improvement in performance. The heavy use of multiplication could also be exploited in the next section by the addition of integer multipliers.

It was observed that all term in the series has an order of a multiple of 2. Hence to further improve performance, x^2 is calculated and stored separately to reduce repeated multiplication as shown in listing 6. This gives an improvement of 47.7% for case 3.

```
float cos_taylor_4terms(float x)
{
 float xx = x*x;
 return 1 - ((xx)/(2)) +
  ((xx*xx)/(24)) -
  ((xx*xx*xx)/(720)) +
  ((xx*xx*xx*xx)/(40320));
}
```

Listing 6: Implementation of Taylor Series with Precalculated x^2

In C, floating-point constants have double-precision unless suffixed, causing extra instructions during floating-point arithmetic. In previous section, it was discussed that such optimisation would lead to a decrease in accuracy. However, by trial and error, it was observed that after replacing `math.h` *cos* function with 4-term Taylor series, the impact of single-precision on accuracy is insignificant. Hence the suffix `f` is added to each constant, forcing the compiler to treat the constants as floats and use single precision arithmetic.

This gives a remarkable improvement as shown in table XI.

TABLE XI
PERFORMANCE OF 4-TERM TAYLOR SERIES + TYPED CONSTANTS

	Average Time (us)	Uplift	Result	Error
Case 1	8696.1	67.21%	920413.5	1.3760e-7
Case 2	353670.6911	66.23%	36123108	6.2143e-7
Case 3	48173894.68	68.43%	4621532160	9.3351e-6

3) Lookup Table: Besides Taylor series, another approach to obtain the value of cosine function would be a lookup table. Values are pre-calculated and stored in an array table with a predefined precision. Instead of calculating the values during run-time, they are obtained from the lookup table. A Python script is written to generate a header file containing the lookup table array, and the cosine function uses the known index to obtain the correct index in the table during run-time as shown in listing 7.

Table XII shows the performance and error with different precisions of the lookup table approach.

```c
float cos_lut(float x) {
 // POWER is 1/precision
 int idx = (x+1) * POWER;
 return COS_LUT[idx];
}
```

Listing 7: Implementation of Lookup Table

TABLE XII
PERFORMANCE AND ACCURACY OF LOOKUP TABLE IMPLEMENTATION

		Average Time (us)		
Precision	File Size	Case 1	Case 2	Case 3
1e-04	156kB	6798.52	278329.44	3493622.90
1e-.5	859kB	6707.61	268444.987	34001558.73
1.25e-06	7891kB	6894.02	271429.469	34142747.14
1e-06	7891kB	6706.43	268179.1067	-

(a). Performance

Precision	Case 1	Case 2	Case 3
1e-04	2.593769e-05	2.686503e-05	0.9099417
1e-.5	1.303582e-05	1.269127e-05	1.16024118e-03
1.25e-06	1.42474e-06	7.32163947e-07	1.17231694e-03
1e-06	6.61121873e-08	2.64428659e-07	-

(b). Accuracy

It is observed that only until the precision is increased to 1e-06 that the accuracy meets the baseline. Although this optimisation gives a massive improvement in performance, it comes in the cost of a large file size, 100 times larger than the baseline system. Additionally, it is not possible to evaluate case 3 due to the lack of available memory on the system caused by the large program file. Hence it was decided that this is not an ideal optimisation to be implemented in future design.

D. Final Optimised Implementation

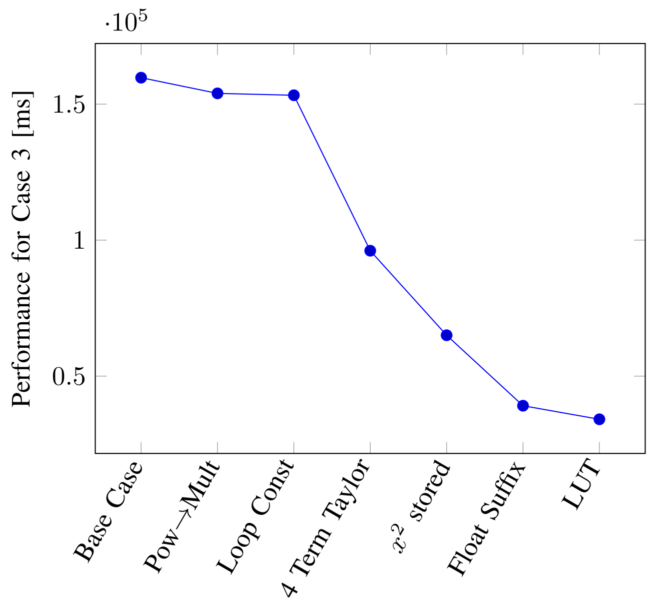


Fig. 5. Performance Optimisations Effect on Case 3

Figure 5 shows a summary of the performance as different optimisations are implemented along the section. Hence it can be concluded that the optimum software implementation of Equation 2 with the current system would be as shown in Listing 8.

```c
float cos_4_taylor (float x)
{
 float xx = x*x;
 return 1 - ((xx)/(2)) +
 ((xx*xx)/(24)) -
 ((xx*xx*xx)/(720)) +
 ((xx*xx*xx*xx)/(40320));
}

float math_expr(float x[])
{
 int i;
 float rtn = 0;
 for (i=0; i<N; i++){
  rtn += 0.5f*x[i]+ x[i]* x[i]*
  cos_4_taylor((tmp-128.0f)/128.0f);
 }
 return rtn;
}
```

Listing 8: Optimum Software Implementation

As this section is focusing on software optimisation of the function, the resource usage remains 4.6064% throughout. Table XIII shows a summary of the implementation.

TABLE XIII
SUMMARY OF CURRENT IMPLEMENTATION

	Average Time (us)	Error	File Size
Case 1	8696.1	1.3760e-7	78kB
Case 2	353670.6911	6.2143e-7	78kB
Case 3	48173894.68	9.3351e-6	91kB

E. Other Potential Improvements

Due to time constraints, other potential optimisations were not analysed. It is expected that the next main performance increase would come from the optimisation of division as the current hardware is using a software routine to emulate floating point division. As this division is used both in the main loop, but also in the calculation of the 4-term Taylor series, optimisations that take advantage of the constant denominators could have a big performance impact.

IV. Task 5 - Multiplier Hardware Support

In order to speed up execution, 3 16-bit multipliers were added to support hardware based multiply instructions for both 16-bit multiplication and multiply extended instructions. As mentioned in the previous section, the 4-term Taylor Series implementation of cos is chosen initially as it was the fastest and is also thought to be able to take advantage of the additional multipliers.

The introduction of hardware integer multipliers allows for multiplication to be implemented using hardware instead of software routines. Although floating point multipliers are not integrated into hardware, the software routines to emulate floating point multiplication can make use of the new instructions to achieve a speedup in performance.

A. Hardware Multiplier Implementation

The performance and the implementation of the hardware multipliers for the Nios II core is largely dependent on the available hardware on the board which the soft core is being instantiated on.

Each Digital Signal Processing (DSP) block on a Cyclone V device supports one of three operating modes for multiplication: one 27 x 27 multiplier or two 28 x 19 multipliers or three 9 x 9 multipliers. [1]. The Nios II also has four different options for the implementation of hardware multiplication which are summarised below[b]:

Implementation	Cycles Till Result	Supported Instructions
Logic Elements	13	mul, muli
32-bit multiplier	3	mul, muli, mulxss, mulxsu, mulxuu
3 16-bit multipliers	3	mul, muli
4 16-bit multipliers	4	mul, muli, mulxss, mulxsu, mulxuu

[b] Data obtained from Nios® II Processor Reference Guide. [2]

As the current board does not support 32-bit multipliers, that option is not available. The board does have embedded multipliers though, and it is desirable to minimise latency and logic element usage on our board, so the options are between the third and fourth rows on that table. Both options were tested and compared. Although the extra cycle of latency between result did not appear to have a significant impact, it was noted that as the compiled code did not make use of the extra instructions for extended multiplication, there was no point in having the extra multiplier for extended support.

B. Resource Utilisation

The use of hardware multipliers does increase resource utilisation on the FPGA, with a few more logic elements and also the use of DSP blocks as described in Table XIV. This results in an increase in resource utilisation score from 4.6064% to 5.8431%.

C. Performance

The performance and the relative improvement of the previous system with a 4 Kbytes instruction cache and a 2Kbyte data cache with the addition of multipliers is described in Table XV.

TABLE XIV
Task 5 System FPGA Resource Utilisation

Logic utilization (in ALMs)	1710 / 32070 (5%)
Total registers	2738
Total pins	47/457 (10%)
Total virtual pins	0
Total block memory bits	345024 / 4065280 (5%)
Total RAM Blocks	52 / 397 (11%)
Total DSP Blocks	3 / 87 (0%)
Total PLLs	1 / 6 (0%)
Total DLLs	0 / 4 (0%)

TABLE XV
Performance with Hardware Multipliers

	Average Time (us)	Uplift
Case 1	4673.61	+46.26%
Case 2	184086.29	+47.95%
Case 3	23524843.54	+51.17%

It is observed that the use of hardware multipliers brings a huge speedup to the performance of the system, due to the reduced number of instructions and cycles needed to perform integer multiplication.

D. Cache Size Re-tuning

The previous experiments used the best cache size for a different problem which was adequate, but as the memory access pattern differs, in order to get optimal performance, the sizes of the instruction and data caches should be re-tuned to see if there is room for improvements in performance, or if there are resource savings that can be made.

It is likely that a larger instruction cache would be greatly beneficial as the each loop iteration has significantly more instructions than the previous problem. Each iteration does not access multiple elements in the array, so it is likely that the data cache does not need to be extremely big. Based on these two factors and in order to reduce compilation and simulation time, only combinations of caches with size between 4kB and 32kB were tested.

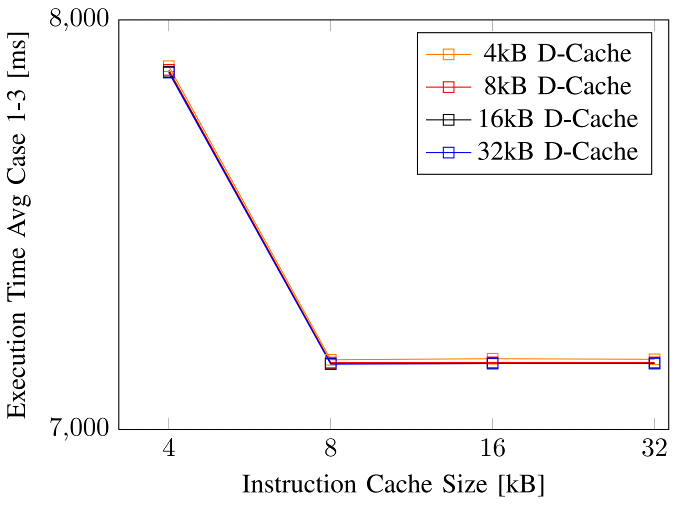


Fig. 6. Effect of Cache Size on Performance

As the results show in Figure 6, the size of the instruction cache has a more significant effect on the execution time, similar to the case in task 3. As the main program is largely compromised of a loop operation, a cache that is able to fit the critical loop is best optimised for this program, this matches our earlier findings in Task 3. Branch mis-prediction is not a significant factor in the program due to the use of a dynamic 2-bit branch history table and hence would only likely mispredict two or three times due to the repetition.

The size of the data cache has a less significant effect due to the multiple access to elements being one at a time, exhibiting more temporal locality than spatial locality. Spatial locality is also present as the elements of the input array are accessed in-order, which is also exploited by the cache. This is likely the reason for the small increases in performance as data cache increases the number of misses is reduced.

Although small, there is a local minima when the instruction cache size is equal to 8Kbytes. This is likely due to the larger caches having higher access times and causing a slight slowdown. As the entire critical loop is able to fit in the smaller instruction cache, the larger instruction cache is not made use of due to it's fixed associativity and block size. Therefore the optimum configuration is a 8Kbyte instruction cache and a 32Kbyte data cache. Increasing the data cache size to 64Kbytes does not provide a significant increase in performance.

E. Revaluation of Cosine Implementation

With a new cache configuration, it was thought prudent to revaluate the chosen implementations of cosine in software and to ensure that changes in the software were not unevenly affected by the hardware bottleneck. The software is run using the code described in Section III-D, with the variable being the implementation of cos.

The `-O3` compiler flag is also enabled to allow further uptick in performance, as it enables a performance uptick without any changes in resources as discussed in the previous report. Raising the optimisation level comes in at the expense of a possible increase in compilation time and possibly more debug issues due to micro-code optimisations. Hence it is only enabled at this stage where most bugs are found and fixed.

The performance, accuracy and performance uplift of case 3 relative to the `math.h` library implementation is listed in Table XVI.

TABLE XVI
PERFORMANCE AND ACCURACY OF CASE 3 FOR COS IMPLEMENTATIONS

Implementation	Performance[us]	Error	Perf. Uplift
math.h cos	36561997.064	9.11400e-06	0%
6-term Taylor	32395194.480	9.11354e-06	+11.40%
4-term Taylor	21308881.070	9.33511e-06	+41.72%
math.h cosf	20671745.942	9.11354e-06	+43.48%
math.h cosf (-O3)	17993500.440	9.11354e-06	+50.79%
4-term Taylor (-O3)	15829388.416	9.33511e-06	+56.71%

It is observed that although `cosf` has a best performance in `O0` optimisation level, the explicitly-implemented 4-term Taylor Series has the best performance when the `-O3` flag is enabled.

Hence it is still safe to conclude that Taylor Series approximation with 4 terms is the most optimal software implementation within the current system.

V. Conclusion

This paper presented a series of system designs designed with enough memory to compute large cases of functions that operated on a vector of floats, by integrating off-chip memory into the design. It also presented a series of software optimisations in C for optimising loops, single-precision calculations and cosine evaluations. These optimisations were then enhanced with the hardware addition of multipliers to accelerate software emulation of floating point multiplication. The cache size was also tested and optimised at each stage, to the point where increases in cache size only led to increase resource usage without corresponding increases in performance. Figure 7 shows the average performance across the three cases against the system resource utilisation with the accuracy of Task 3 for each system illustrated by the colour bar.

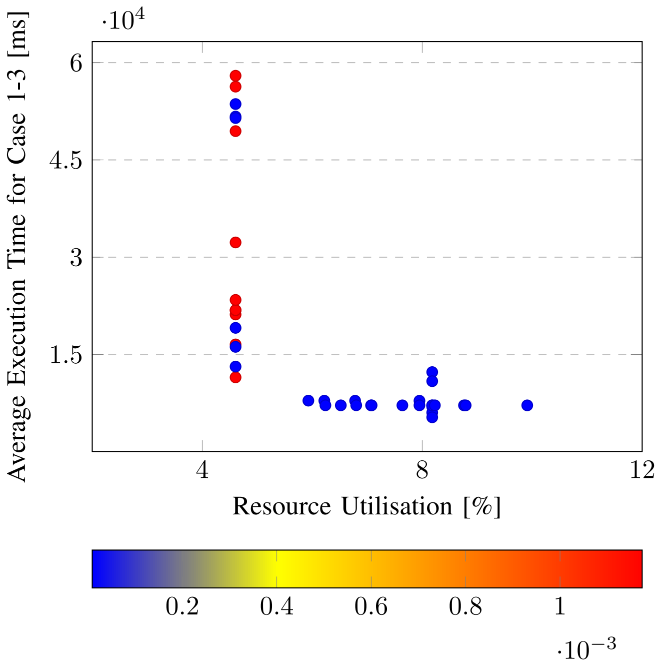


Fig. 7. Performance vs Resources for Generated Designs in Task 4-5

This suggests that the design's bottleneck when computing this function remains at computation rather than memory. Therefore, to reduce execution time and increase performance, the next steps should be moving computation from software into hardware, which may lead to a resource usage increase, but should bring about a relatively greater performance increase.

References

[1] Intel Cyclone V Device Handbook Volume 1: Device Interfaces and Integration. *Intel.* (2018,11)
[2] Nios® II Processor Reference Guide. *Intel.* (2020,10)