Digital Systems Design Coursework - Report 1

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 the first two tasks assigned in the coursework specification for Digital Systems Design. The following investigation was carried out using Intel Quartus Lite 21.0, on a Terasic DE-1 SoC board with a Cyclone V FPGA.

II. Task 1 - Nios II Setup

This task involves using the Platform Designer tool to create a system with a Nios II CPU, On-Chip Memory, a JTAG UART, an Interval Timer, a System ID Peripheral and Parallel I/O, using provided blocks within Quartus; and then using the Nios II Software Build Tools to run a simple "Hello World" C program on the system.

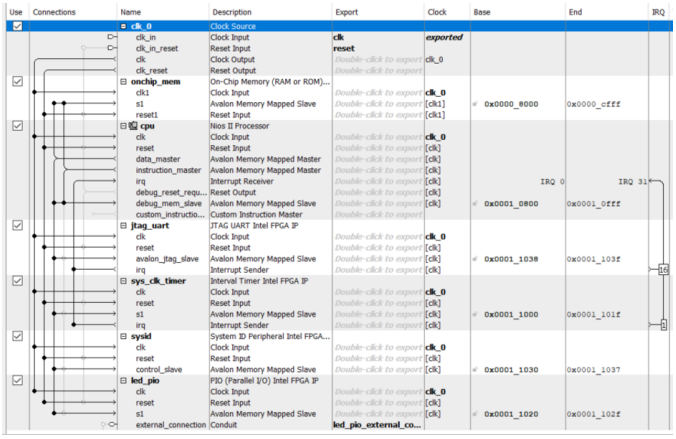

Fig. 1. System Components and Connections within the Platform Designer

A. System Design using Platform Designer

The complete setup is shown in Figure 1. Components that have hidden settings not shown in the figure are further detailed here:

1) Nios II: According to the instructions, a Nios II/f core was chosen as the instantiated soft core, with no hardware implementation of multiplication and division. A 2Kb instruction cache was added to the design, and tightly coupled memory was not used.

2) On-Chip Memory: The On-Chip Memory was specified as instructed - a writable 20 kilobyte sized RAM.

B. Instantiating the System on FPGA

By using the generated block description file, the system was added to the FPGA board and connected to the corresponding inputs and outputs for the clock, reset and LEDs. A screenshot of the complete block description is shown in Figure 2.

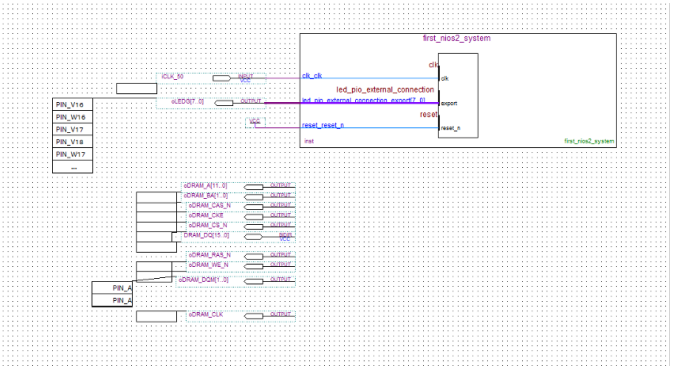

Fig. 2. Block Level Description of System

The unconnected IO in the bottom left of Figure 2 will be used in a later task.

C. Quartus Compilation Analysis

The design was successfully compiled and the reports for resource usage and timing analysis were obtained. It is expected that this will be one of the smallest designs as it contains the bare minimum, and thus future designs will not be better when compared on resource usage and timing analysis.

1) Resource Usage: The following table shows the resource usage obtained from the compilation of the current design.

Logic utilization (in ALMs)	1355 / 32070 (4%)
Total registers	2151
Total pins	47/457 (10%)
Total virtual pins	0
Total block memory bits	210048 / 4065280 (5%)
Total RAM Blocks	45 / 397 (11%)
Total DSP Blocks	0 / 87 (0%)
Total PLLs	0 / 6 (0%)
Total DLLs	0 / 4 (0%)

When broken down by entity, it is observed that most of the logic utilisation is by the Nios II processor, and the on-chip memory is the main consumer of block memory bits. As no

dedicated multiplication hardware is included, it is no surprise that the DSP utilisation sits at 0%.

2) Timing Analysis: The results from the timing analysis are included below:

	Setup	Hold	Recovery	Removal	Minimum pulse width
Worst-case Slack	10.458	0.017	14.518	0.339	8.441
altera_reserved _tck	23.300	0.017	48.418	0.339	48.746
sopc_clk	10.458	0.132	14.518	0.425	8.441

The timing analysis shows that at all simulated temperatures, there are no issues with meeting the necessary timing constraints at the current clock speed - 50MHz, and also means that there may be a bit of room for extra performance by increasing the clock speed.

No optimisations through Quartus or manual optimisations were carried out on the fitting of logic and timing, so these are baseline numbers that could possibly be improved slightly to increase performance and reduce resource usage.

III. Task 2 - Computing a Simple Function

The task at hand here is to use the Nios II processor initialised in the previous task to compute Equation 1 for three different inputs shown in Table I.

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

A. Function

To compute Equation 1, function `sumVector()` is written as shown in Listing 1. The function takes in vector `x` with length `M` and returns `y` as a `float`. It is assumed that the elements are single-precision and are in the range between 0 and 255.

```
float sumVector(float x[], int M){
  int i;
  float rtn = 0;
  for (i=0; i<N; i++) {
    rtn += x[i] + x[i]*x[i];
  }
  return rtn;
}
```

Listing 1: sumVector

B. Memory

It was observed that the system runs out of memory and is not able to evaluate Case 3 in Table I. Running using debug tools shows that the array generation function runs up to a certain number and then stops and this number also approximately equal to the memory allocated for stack/heap divided by 4 bytes (size of float). The vector in case 3 requires approximately 1Mb of memory which exceeds the maximum available on chip memory size (256kB) on this particular FPGA board. Hence, case 3 can only be evaluated with external memory which will be further discussed in the next report.

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

Program size of the `elf` file is around 12kB, obtained during compilation, and is consistent across all test cases. Although each case has different vector size, it is declared during run-time and stored in the stack. The total memory required by the application would be approximately the sum of the program size and vector size as stated in Table I.

As the total memory for Case 2 would be around 32kB, the size of the on-chip memory is specified to be 34kB to reduce resource usage.

C. Correctness

As in Nios II, the `printf()` function does not support floating point, the result obtained on the console is scaled down. A separate Python script is written to recover the value calculated. However, during the scale down process, there would be data loss leading to inaccuracy. Hence the C program was also compiled and run on a desktop computer to test the program for functional correctness and ensure that the Nios II is carrying out the correct instructions.

In addition, a Python script is also written to compare the result. Results are recorded in table II.

TABLE II
Results Obtained by Different Methods

	Nios II	C	Python
Case 1	1143808	1144780	1144780
Case 2	44509184	44509760	44509745.3125
Case 3	N/A	5693058048	5693101440.041504

The `float` type in C is defined as single-precision (4 bytes) while they are double-precision (8 bytes) in Python. This is further confirmed as the C script provides the same results with Python when all floats are replaced by doubles. The impact of precision in floating points becomes more obvious as there are more data with more significant figures, which is shown progressively in the 3 cases.

However further increasing floating point precisions (e.g. 128 bits) would not improve accuracy. As elements are single-precision with 7 significant decimal digits and the highest order in the function is 2, a double-precision floating point with 15 significant decimal digits would be more than enough to store the precise value.

D. Measuring Performance

In order to determine the performance of our system, we timed runs of the `sumVector` function to use as a benchmark.

The specification suggested using the system clock ticks to measure performance, but as the timestamp driver is more accurate with a higher time frequency (in sync with clock rate), it was chosen. It uses the same hardware components as the system clock driver, and an example of how it was used is shown in Listing 2.

```
#include <sys/alt_timestamp.h>
#include <alt_types.h>
...
int main(){
  ...
  if (alt_timestamp_start() < 0){
    printf("No timer available");
  }
  alt_u32 time0 = alt_timestamp();
  sumVector(x,N);
  alt_u32 time1 = alt_timestamp();
  ...
}
```

Listing 2: Timestamp Timing

The `alt_timestamp()` calls return unsigned 32 bit integers, corresponding to the number of ticks that have passed since `alt_timestamp_start()` is called. This can be converted to time by using the known timestamp frequency (50 MHz).

The benchmark used 10 runs in repeat within the same program without re-downloading the ELF, and the time for each run was noted. Based on the individual times, the average time and standard deviation is also obtained.

TABLE III
BASELINE PERFORMANCE

	Average Tick	Average Time (us)	Standard Deviation (us)
Case 1	38710.3	774.206	0.1204
Case 2	1548065.2	30961.304	0.1165

E. Effect of Cache Size on Performance

The initial Nios II was setup with a 2KB instruction cache and a 2KB data cache. As the on-chip memory is not directly connected to the CPU and communicates through the Avalon Master-Slave protocol, there is a certain latency attached to it despite it being implemented on "fast" memory. Therefore, the size of the instruction cache and the size of the data cache could have significant impacts on the performance of the program. It was hypothesised that an instruction cache that could fit the whole program into memory would be able to bring about one of the largest speedups.

For results shown in table IV, cache size refers to the total size of both the instruction cache and data cache[1] and resources is the total logic utilisation in percentage[2]. Each benchmark run includes one untimed function call to warm up the cache and provide accurate results. No software optimisations were applied both within code or through compiler options, only the cache size was altered. The data from Table IV is also shown in Figure 3

TABLE IV
BENCHMARK PERFORMANCES WITH DIFFERENT CACHE SIZE

Cache Size (kB)	Resources	Case 1 (us)	Case 2 (us)	Avg.% Change
0 (0I + 0D)	4.3009%	2681.052	107341.1440	-267.40%
2 (1I + 1D)	4.6297%	931.186	36839.834	-100.00%
4 (2I + 2D)	4.7713%	774.206	30961.304	0.00%
8 (4I + 4D)	5.0485%	653.738	25776.214	11.10%
16 (8I + 8D)	6.734%	653.102	25666.828	11.33%
48 (16I+32D)	7.8433%	653.086	25660.692	11.34%
128 (64I + 64D)	13.4809%	653.02	25660.37	11.35%

It was observed that performance increases significantly with cache size for small values, showing the impact of latency when having to fetch each instruction and data from on-chip memory, but plateauing when the cache size reaches 4kB. An increase in cache size allows more instructions and data to be stored, reducing cache miss rate and reducing the average memory access time. However, reaching 4kB, the cache is large enough to taker in sufficient data to avoid stalls due to memory access. It is suspected that the bottleneck sits in other factors such as the current software implementation of multiplication and will be further discussed in following reports.

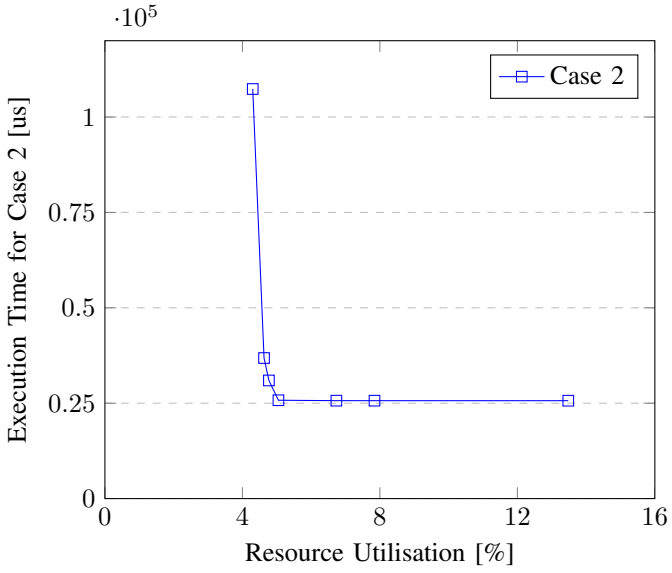

Fig. 3. Effect of Cache Size on Performance and Resources

Hence, future designs for this probelm would have an instruction cache of 4kB and a data cache of 2kB.

[1]When unspecified, the size of each cache is equal

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

F. Use of Compiler Flags to Increase Performance

With the baseline cache configuration (2KB I-cache + 2KB D-cache), a couple different compiler flags were also tested and their effects measured.

1) Size: The -Os compiler flag allows the compiler to optimise for program size, generating both a smaller ELF file and reducing execution time as shown in table V. This is probably an optimisation that should be used to reduce the size taken up by the program in memory.

2) O3: Using the -O3 compiler flag enables a significant uptick in performance, reducing execution time for both cases, coming close to providing a similar performance benefit to the increased cache size without the increased resource utilisation.

TABLE V
PERFORMANCE OF DIFFERENT OPTIMISATION LEVELS

Compiler Flag	Program Size (kB)	Case 1 Time (us)	Case 2 Time (us)
Nil	12	774.206	30961.304
-Os	10	729.236	28610.012
-O3	10	700.386	27498.434

However, this better performance comes in an expense of compilation and possibly the ability to debug the problem. Hence changing the compiler flag should only be done at the final stage of design where most bugs are found and fixed. Using O3 could be an option to increase performance without needing extra resources to get to that point.

IV. CONCLUSION

This paper presented a series of barebones designs to compute a simple function that operated on a vector of floats. This implementation works well for small and medium cases, but does not function for large cases as it runs out of memory due to the limited amount of on-chip memory. The cache size was also tested and optimised for this function, to the point where increases in cache size only led to increase resource usage without corresponding increases in performance.

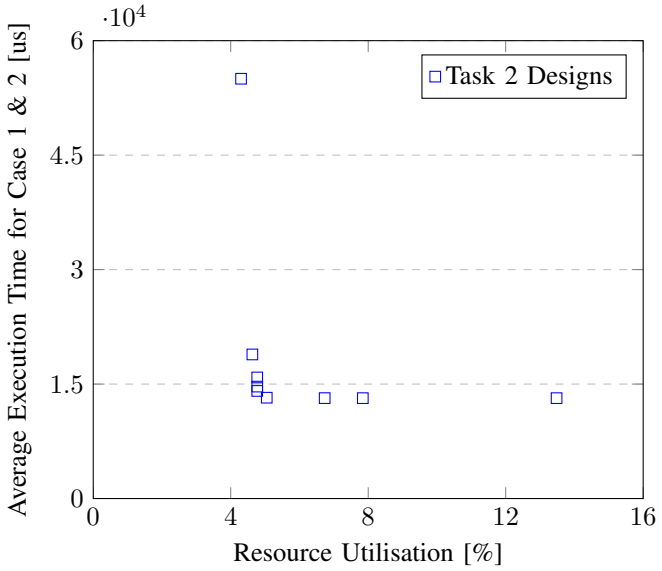

Fig. 4. Performance vs Resources for Generated Designs in Task 2

This suggests that the design's bottleneck when computing this function is now 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.

The use of compiler flags to gain a performance improvement in software with no resource cost was also explored, and may be useful in future designs.