Report 1

(Task 1-2)

Digital System Design

Group 10

Sun, Kaiyue (CID: 01197813, email: ks4016@ic.ac.uk)
Wang, Yuwei (CID: 01260801, email: yw6316@ic.ac.uk)

I. Introduction

The objective of this report is to present and justify the result of designing and programming digital system that implements function evaluation based on available FPGA, as well as to discuss and analyze the problems encountered during the whole process.

II. NIOS II Setup

The FPGA resources of the baseline system was recorded below.

Resources	*Usage*	*%*
Logic Elements	2,313/15,408	15%
Combinational functions	2,125/15,408	14%
Dedicated logic registers	1,368/15,408	9%
Registers	1368	
Pins	47/347	14%
Virtual pins	0	
Memory bits	192,384/516,096	37%
Embedded Multiplier 9-bit elements	0/112	0%
PLLs	0/4	0%

Table-1 Resource Utilization Summary

As the flow summary of the compilation report shown, when the size of the on-chip memory were set to be 20 KB, the utilized *Memory Bits* accounted for 37% of the total.

III. Computing a Simple Function

However, when the original *hello_world* C code was modified to be the evaluation of given sum-vector function, the currently available resources, especially the on-chip memory, were far from enough for any test case defined in task 2.

When the project finished building, a reminder of the total size of the program size (code and initialized data) would show in the console and remain unchanged regardless the different vector sizes in the three test cases.

Due to the fact that vector *x* was declared as a local variable inside the function, dynamic memory (free stack and heap) allocation was performed during execution to store *x*. Additionally, the evaluation of function value *y* also needed dynamic memory but the amount was ambiguous and therefore it was omitted in the calculation and later an estimation would be found out by experimental testing.

The memory required could be estimated by summing up the program size and the dynamic memory allocated to the array *x*: the memory size of *float* (32 bits) times the number of elements in *x* giving the memory size occupied by *x*.

$$Memory\ required\ (KB) = 38 + \frac{(32 \times N)}{8} \div 1024$$

Test Case	*N (size of array x[ ])*	*Program size (KB)*	*Memory requirement (KB)*
1	52	38	38.21
2	2041	38	45.98
3	261121	38	1058.01

Table-2 Memory Required (float)

To experimentally find out the dynamic memory needed besides the *x[N]*, various sizes of memory were added on top of memory calculated previously and then the program was run to observe the result. Unsurprisingly, not all the tests were able to run given available resources. For both test case 1 and 2, an overhead memory of 1000 bytes was sufficient to run the program. For the test case 3, since the vector instantiated during execution (vector size of 261121) was too large to be stored in the on-chip memory, no matter how the memory size was manipulated, even to the maximum capacity of the FPGA board, running the program still resulted in a failure, indicating that additional memory was needed in order to run test case 3.

After the memory size was raised to 47.85 KB (49000 bytes), considering that the program still needed some memory in addition to the number computed before, the usage of *Memory Bits* became 81%, almost equivalent to the maximum percentage that could be fitted in the device. A further increment of the memory size would cause *Quartus* to be incapable of allocating all the RAM cells in design because some RAM cells served for other functions (e.g. Logic Element).

Test Case	*NIOS II SBT for Eclipse*		*Matlab*
	y	int(y)	
1	1117.949219	1117	1.117949218750000e+03
2	43466.54816	43466	4.346654815673828e+04

Table-3 Result Comparison (NIOS II vs. Matlab)

The results obtained by evaluating the function in *Matlab* were slightly different. Before *y* was cast to integer inside the *gcvt()* function in the code, the result was a *float*, while *Matlab* was by default using *double* precision to represent floating point numbers, offering more accurate results. After casting *y* into an integer, the result was rounded down, leading to a different representation.

If the data type was changed into *double*, which takes 64-bit memory, occupying twice the space than *float*, so only test case 1 can be implemented by the system. It also took longer to run the program. The time required was given in clock ticks corresponding to the timeout period (1ms) in the interval timer instantiated previously.

$$Memory\ required\ (KB) = 34 + \frac{(64 \times N)}{8} \div 1024$$

Test Case	*N (size of array x[ ])*	*Program size (KB)*	*Memory requirement (KB)*
1	52	34	34.41
2	2041	34	49.95
3	261121	34	2074.01

Table-4 Memory Required (double)

Test Case	*N (size of array x[ ])*	*Tick (1ms)*	*Tick (1ms)*
		float	double
1	52	1	3
2	2041	62	N/A
3	261121	N/A	N/A

Table-5 Result Comparison (float vs. double)

Thus, the time required to evaluate the function depended on both the precision used as well as the vector size.

It seems that there is a relationship between the hardware resources and performance resulted. However, the only hardware resources that could be manipulated at this stage is the size of memory, either on-chip memory or instruction cache. Varying the size of on-chip memory did not bring any influence on performance in term of latency. On the other hand, enlarging the instruction cache should theoretically lead to significant improvement on performance, but given the available resources, the cache could not be increased any more. For example, 51.85 KB (47.85+4) already exceeded the maximum percentage allowed to use in memory (mentioned above).

Because test case 1 had smaller vector size than task case 2, the on-chip memory could be reduced to provide extra memory for instruction cache. The relationship of cache against performance was verified below.

Test Case	*Cache: 512Bytes*	*Cache: 1KB*	*Cache: 2KB*	*Cache: 4KB*
1	2	2	1	1
2	70	67	62	N/A
3	N/A	N/A	N/A	N/A

Table-6 Latency (unit: ms) Comparison for Different Cache Sizes

IV. Conclusion

In conclusion, the size of memory should be carefully estimated prior to implemented in the actual design. It should be large enough to guarantee that all data and instructions can be fitted into the system and should not exceed the maximal memory bits available to on-chip memory.

The instruction cache is designed for instructions that are more likely to be used to realize fast instruction access. Larger cache means faster execution time but enlarging the size of cache stops benefiting when it is more than the size of the program.

Float is better than double in this case because it takes less memory space without sacrificing too much precision of the outcome. Thus, given the limited amount of memory, there exists a trade-off between the accuracy and latency.