Digital Systems Design Report 1

Pranav Malhotra, CID:823 617

Dada Oluwatomisin, CID:846 141

29th January 2016

Contents

1 Aims

We have the following aims that we hope to achieve by completing Task 1 and Task 2 of the Digital System Design Coursework.

1. Get familiar with the design of a NIOS II processor and the design choices that impact the system's performance.
2. Use industry standard tools such as Quartus II and analyse compilation reports and assess feasibility of design.
3. Use industry standard tools such as Nios II Eclipse and analyse project size and resource allocations; conduct application profiling and understand linker files and .map files.
4. Analyse performance of program in terms of accuracy, speed and amount of resources used.

2 System on Programmable Chip Components

In this section, we will be discussing the components that our system consists of.

We will use a 50Mhz clock generated on the Cyclone III Chip to drive all parts of our Nios II processor. A processor clocked at 50Mhz is significantly slower than modern processors, such as Intel E5-1620V2, which are clocked at 3.7GHz[1]. However, the NIOS II processor provides the advantage of being a soft processor. We can add and remove complexity as we like. The NIOS II processor also allows us to add custom instructions that enable hardware acceleration. This is ideal for our task.

On-chip memory is the highest throughput, lowest latency memory possible in an FPGA-based embedded system. It typically has a latency of only one clock cycle[2]. For task 1 and task 2, we use on-chip memory to store our program, initialization data, stack and heap. However, On-Chip Memory is more occasionally used for:

- Separate exception stack for use only while handling interrupts
- Fast data buffers
- Fast sections of code
- Fast interrupt handler
- Critical loop
- Constant access time that is guaranteed not to have arbitration delays[3]

During the tutorial, we included an instruction cache in our system. Caches are added to streamline instruction fetches from external memory locations. Because memory accesses in most computer programs display some sort of temporal or spatial locality, fetching instructions from multiple memory locations and storing them in locations near the processor will significantly reduce latency. Because caches are also stored on on-chip memory, it does not make much sense to add an instruction cache to our system at this point. Instruction caches will come in handy in future tasks should we load our program and initialisation data onto the external SDRAM.

The interval timer is added to allow us to measure our program's performance using the times(NULL) command. We could keep implement a counter in software, however, this would cause a structural hazard as we will be using the Arithmetic Logic Unit (ALU) of the processor to maintain our counter instead of having dedicated hardware to do so. The interval timer core generates an IRQ whenever the internal counter reaches zero. This is an interrupt request (IRQ) that we have to handle. The Nios II HAL interprets low IRQ values as higher priority. The timer component must have the highest IRQ priority to maintain the accuracy of the system clock tick.

3 Memory Allocation on System

A major problem that we were facing while completing task 2 was that our program would not fit onto our NIOS II Processor. We then looked at the .map file that gave us the breakdown of our memory usage. Below is a short description of each data segment and what they are used for.

- The .data segment contains any global or static variables which have a pre-defined value and can be modified. That is any variables that are not defined within a function (and thus can be accessed from anywhere) or are defined in a function but are defined as static so they retain their value across subsequent calls.
- The .bss segment, also known as uninitialized data, is usually adjacent to the data segment. The BSS segment contains all global variables and static variables that are initialized to zero or do not have explicit initialization in source code.
- The .rodata segment is a read-only data segment that contains static constants rather than variables.
- The .text segment contains the program's executable instructions. This data segment is read-only on many architectures to prevent a program from accidentally modifying instructions.[4]

The gcvt instruction requires numerous libraries to be imported. This significantly increases the program size. Using the printf function, we were able to reduce our program size and evaluate the function for test case 2. Studying .map files gave us clearer understanding of the memory layout of a C program which is bound to come in handy in the upcoming tasks.

4 Implementations of Function

For task 2, we had to evaluate the expression:

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

We evaluated the expression over the provided vector with many different algorithms which are explained below.

4.1 Analytic Algorithm

To evaluate the equation, we formed a closed form expression that takes advantage of the regular nature of the test cases. Our algorithm only uses the step size and the size of the vector to compute the answer. We expect the analytic algorithm to have the best performance as it does not access the vector x. This algorithm, however is not representative of the computational complexity that is required for the future tasks for two main reasons. Firstly, our algorithm will not work on any arbitrary vector x. Secondly, the equation in the following tasks does not have a closed form expression.

```
float sumVectorAnalytic(float s, float M){
        return (s*(M/2))*(M-1)+(s*(M-1))*(s*M)*(2*(M-1)+1)/6;
}
```

4.2 Iterative Algorithm

The iterative algorithm is more representative of the approach we will use to complete the future tasks. A for loop runs through the vector x and evaluates the equation, accumulating the result in the variable sum. While storing our vector x, we can either store the values as single-precision floating points (float) or as double-precision floating points (double). A double will require twice as much space as a float, however, it will give us greater precision. The discrepancy in precision comes from the fact that when we conduct floating point additions, we require the exponents of the two numbers to be the same. To illustrate the cause of the error, we shall add 10 000 000 to 0.5.

We first write the numbers in their scientific notations:

$$10,000,000 \xrightarrow{inscientificnotation} 1.0011000100101101 * 2^{23} \tag{2}$$

$$0.5 \xrightarrow{inscientificnotation} 1.0 * 2^{-1} \tag{3}$$

Next, we ensure that radix of the smaller number is shifted to ensure that both the numbers have the same exponent. Then, we perform the addition.

	0	0	1	1	0	0	0	1	0	0	1	0	1	1	0	1	0	0	0	0	0	0	0	0
+	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
	0	0	1	1	0	0	0	1	0	0	1	0	1	1	0	1	0	0	0	0	0	0	0	1

This addition returns a mantissa that is 24 bits long. However the single-precision floating point representation only allows us to have a 23 bit mantissa. This means our result will be truncated and the last bit is ignored; the result is 10 000 000. We notice that, adding two numbers that have an exponent that differ by more than 23 will result in effectively adding zero to our larger number. Floating point multiplications will cause similar rounding errors. Thus, we have to carefully plan the trade off between precision and complexity of our program, both in terms of computational time and memory usage. It is important to note that MATLAB, by default, stores all values as uses double-precision floating-points.

Below are two algorithms that implement the addition using floating points and doubles respectively. Their efficiency is discussed in the next section of this report.

4.2.1 sumVectorFloat()

```
float sumVectorFloat()
{
        int i;
        float sum = 0;
        for(i=0; i<N; i++)
                sum += numbers[i]*(1+numbers[i]);
        return sum;
}
```

4.2.2 sumVectorDouble()

```
double sumVectorDouble()
{
        int i;
        double sum = 0;
        for(i=0; i<N; i++)
                sum += numbers[i]*(1+numbers[i]);
        return sum;
}
```

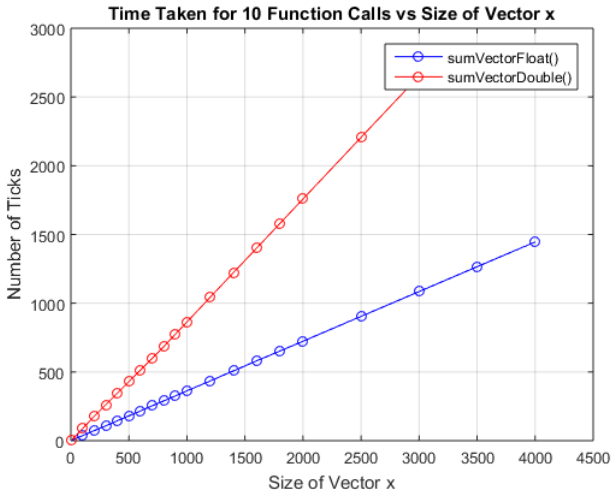
5 Performance of Computation

5.1 Analytic Algorithm

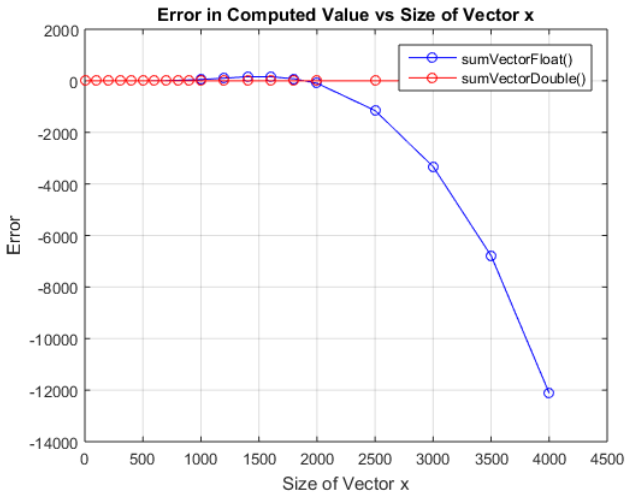
The analytic algorithm gives us the best performance. It has a time complexity of $O(1)$, a space complexity of $O(1)$ and the returns the same result as MATLAB. However, as stated above, the applications of an analytic algorithm are highly limited.

5.2 Iterative Algorithm

We tested the two iterative functions and obtained the following data.



It is clear that the time taken for both functions increases linearly with the size of vector x. This is expected as the algorithm runs through the array once and thus has a complexity of $O(n)$. There is however a difference between the gradient of the two graphs. The difference arises from the fact that operations on double-precision floating points take significantly longer than operations on single-precision floating points.



Comparing the two functions based on their precision, we see that as size of the vector increases, the function sumVectorFloat() returns a value with an error that grows exponentially. The error is introduced because of the truncation errors when adding floating-point numbers that have exponents that differ by more than 23. There are also errors that occur in sumVectorDouble() however these are due to the fact that we are using truncated libraries and thus cannot print floating-points. Thus, rounding off errors occur when we cast a double into an integer while printing. Tables of data collect are provided in the appendix.

6 Questions asked in Booklet

1. Record the FPGA resources used by your baseline system (i.e. Total logic elements, Total memory bits, Embedded Multiplier 9-bit elements, and Total PLLs).

   - For our baseline system we used, 2313 logic elements, 192 384 memory bits and 0 embedded multiplier 9-bit elements or PLLs.

2. How many resources does your system occupy on the FPGA? (for Task 2)

   - For our system to be able to compute test case 2, we had to increase the on-chip memory. We provided 49 152 bytes of on-chip memory. As such, our system used 2335 logic elements, 421 824 memory bits and 0 embedded multiplier 9-bit elements or PLLs.

3. Is your system able to evaluate the given function for all test cases? Why not?

   - We were not able to evaluate the function for the third test case using the iterative algorithm. For the iterative algorithm, we have to first generate the vector and store it in memory. The Cyclone III FPGA chip has limited memory bits and thus, without using an external source of memory such as the SDRAM (available on the DEO board), we cannot generate a vector of size 255001.
   - To generate a vector of floats of size 255001, we would need 102 004 bytes of memory. We are limited to 49 152 bytes of on-chip memory, some of which has to be used to store our program.

4. What is the size of your NIOS II application for different vector sizes and what is the maximum memory required?

   - The size of our application does not change for different vector sizes. The reason for this was discussed in detail in the data segment section of the report. The vector is declared in the main function and thus is generated on the stack.
   - Should we declare the vector x as a global variable, then the program size + initialization data size will change. Studying the .map files, it is clear that single-precision floating points occupy 4 bytes of memory, whereas double-precision floating points occury 8 bytes of memory.

5. Compare your results with MATLAB. Do your results agree? If not, why not?

   - As discussed in section 5, when we used double-precision floating points to evaluate the function, we obtained the same results as MATLAB. There was a discrepancy in results when we used single-precision floating points. This was due to the truncation and rounding errors.

6. What is the time required to evaluate the sumVector() function for each test case?

   - A graph showing the amount of time it take to evaluate both iterative functions is presented in section 5. Detailed data can be found in the form of a table in Appendix A.

7. The best design is the one that provides the least latency for the least possible resources. Where is your current design in that space?

   - As stated in the coursework booklet, we need vector x to be stored as a single-precision floating point. Our tests have conclusively shown that a single-precision floating point representation will use half as many resources and will be faster, however the error grows exponentially with the size of the vector x. Thus, limited by the coursework stipulations, our design trades off precision for an increase in speed. In addition, a single-precision floating point representation allows us to minimise resources on the FPGA.

7 Conclusion

We achieved the aims that were set out at the start of the report. Firstly, we gained a deeper understanding of the basic building blocks of the NIOS II system, made smart design choices to minimise the resources required while maximising efficiency and are now able to understand compilation reports and assess feasibility of our designs.

Secondly, we noticed some of limitations that the NIOS II system has with regards to printing floating-point numbers and also studied how fundamental operations such as single-precision and double-precision floating point additions and multiplications affect our final result. We also learnt the way that memory is allocated on the NIOS II processor by looking at .map files.

Lastly, we designed two algorithms to evaluate the result of the expression. We studied some of the limitations of an analytic algorithm and analysed the performance of our iterative algorithm with respect to latency, precision and memory.

References

[1] Intel® Xeon® Processor E5-1620 v2 (10M Cache, 3.70 GHz) Specifications. (n.d.). Retrieved January 29, 2016, from http://ark.intel.com/products/75779/Intel-Xeon-Processor-E5-1620-v2-10M-Cache-$3_70 - GHz$.

[2] Corporation, A. (2010). Memory System Design. Retrieved January 26, 2016, from https://www.altera.com/content/dam/altera-www/global/$en_US/pdfs/literature/hb/nios2/edh_ed51008.pdf$.

[3] Corporation, A. (2011). . Using Tightly Coupled Memory with the Nios II Processor. Retrieved January 26, 2016, from https://www.altera.com/content/dam/altera-www/global/$en_US/pdfs/literature/tt/tt_nios2_tightly_coupled_memory_tutorial.pdf$.

[4] Data Segment. (n.d.). Retrieved January 26, 2016, from https://en.wikipedia.org/wiki/$Data_segment$.

A Data Collected

	sumVectorFloat()		
Size of Vector x, N	Time Taken, ticks	Result of Computation, y	Error in Computation
10	4	7	0
101	37	3888	0
201	73	28877	0
301	108	94965	0
401	145	222154	0
501	181	430442	0
601	217	739827	4
701	254	1170310	9
801	290	1741889	19
901	326	2474563	33
1001	363	3388333	52
1201	435	5839160	102
1401	508	9254379	160
1601	581	13794058	158
1801	652	19618214	79
2001	724	26886866	-96
2501	905	52428368	-1156
3001	1085	90498488	-3333
3501	1266	143597392	-6795
4001	1446	214225664	-12124

Table 1: Data for sumVectorFloat()

	sumVectorDouble()		
Size of Vector x, N	Time Taken, ticks	Result of Computation, y	Error in Computation
10	8	7	0
101	88	3888	0
201	177	28876	1
301	261	94965	0
401	345	222154	0
501	430	430442	0
601	515	739831	0
701	599	1170319	0
801	685	1741908	0
901	775	2474596	0
1001	864	3388384	1
1201	1044	5839261	1
1401	1223	9254538	1
1601	1401	13794215	1
1801	1580	19618292	1
2001	1759	26886769	1
2501	2207	52427212	0
3001	2654	90495155	0

Table 2: Data for sumVectorDouble()