Real Time Embedded System Lab2 Report
Profiling, custom instruction & hardware accelerator

Olivér Facklam, Ju Wu

April 2021

Introduction

The goal of this lab is to design several methods for a specific bit manipulation operation and evaluate their respective performances. Three methods are designed: a software bit manipulation through a C function, a custom instruction for the NIOS-II processor, as well as a hardware accelerator using DMA. The various methods are then evaluated through both software and hardware profiling.

Bit manipulation methods

Software Design From C

In this part, we write a C function to transform the 32-bit input data as required. To facilitate conversion speed of the code and avoid using a loop structure, we create a lookup table for the bit-flipped output corresponding to an 8-bit input.

Hardware Custom Instruction

In this part, we write a *.vhd* file and create a corresponding new combinational component that can do the required conversion for a 32-bit input value. This component is registered as a custom instruction connected to the NIOS-II processor.

Hardware Accelerator

In this part, we use the same combinational logic as previously to carry out the bit manipulation. However, we combine this with a DMA with burst capability in order to efficiently and directly interact with the on-chip memory, without the need to go through the processor. The DMA has a single master interface, and uses 2 FIFOs (one for input and one for output) to efficiently handle the transfers. The register map of the accelerator component allows to set the source and destination addresses, as well as the buffer length, and allows to poll a "done" flag. The conversion is started as soon as the buffer length is non-zero.

System architecture

Once all three methods are developed, we integrate them into a system containing – in addition to the processor, on-chip memory, custom instruction and hardware accelerator – also a system timer and a performance counter necessary for the profiling we will do in the next section. The system is represented in figure 1.

Profiling For the Above Methods

In order to do the profiling of our bit manipulation methods, we execute them on a given test buffer. First the cache is flushed and the buffer is filled with a specific bit pattern. Then one of the bit manipulation methods is called to do the

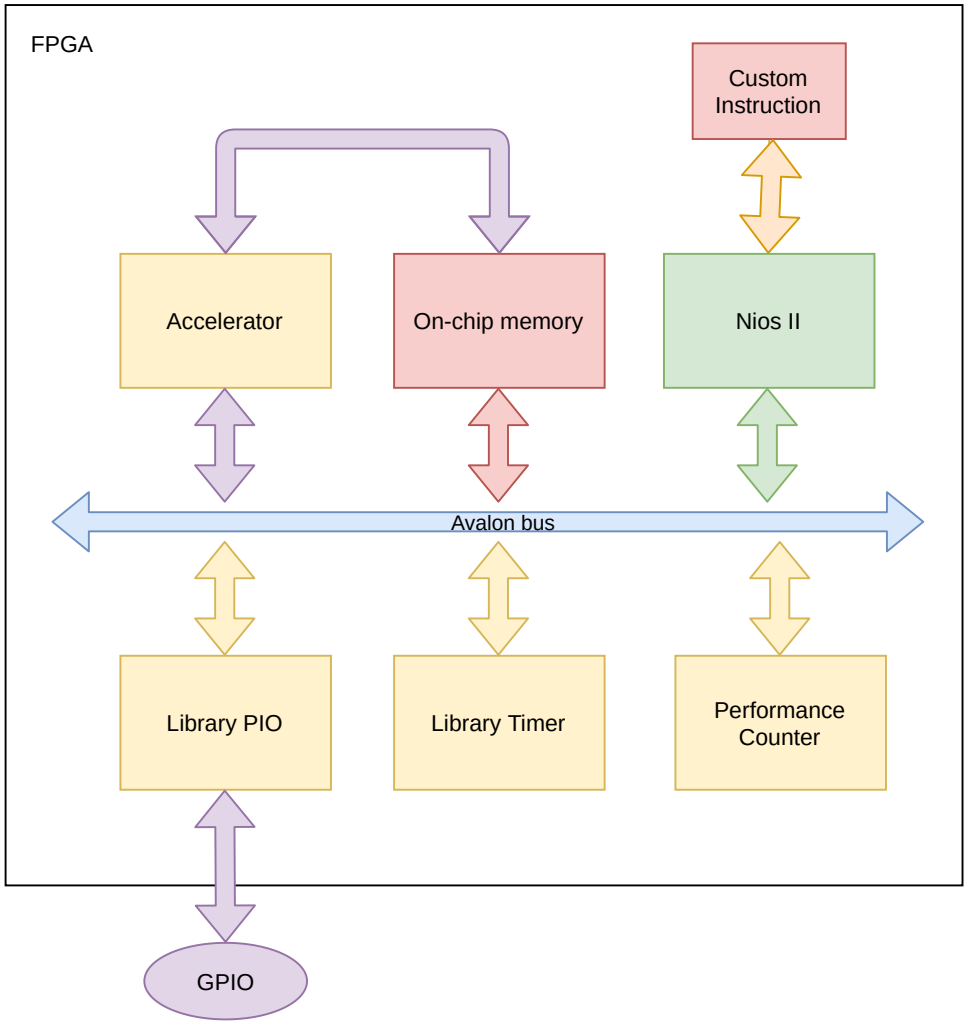

Figure 1: System architecture implementing profiling techniques & all three bit manipulation methods

operation in-place. This function call's duration is measured through the two different profiling methods described below. After the conversion is done, the buffer is checked for correctness by comparing to the expected resulting bit pattern.

Software profiling

In this section we enable the system timer and the `gprof` library and use the GNU profiler to measure the performance of all 3 methods. Unfortunately, even with a timer resolution of 100 μs, the profiling didn't give any exploitable results for the bit manipulation of a single 32-bit input word.

However, for a table of 1000 input words, we obtain the following software profiling result:

- Software: 3.40 ms
- Custom instruction: 1.40 ms
- Hardware accelerator: 0.10 ms

Hardware profiling

In this section we use the performance counter and its library functions to profile the relevant sections of code (the body of each method). We obtain a tick-level resolution, allowing us to do the profiling for 1 input word as well as a vector of

1000 input words. The results are as follows.

Hardware profiling with 1 input word:

- Software: 269 ticks = 5.38 μs
- Custom instruction: 133 ticks = 2.66 μs
- Hardware: 73 ticks = 1.46 μs

Hardware profiling with 1000 input words:

- Software: 129993 ticks = 2.60 ms
- Custom instruction: 64589 ticks = 1.29 ms
- Hardware: 2438 ticks = 0.049 ms

Observation

The first thing to note is that although there is a slight discrepancy between software and hardware profiling for 1000 input words, the obtained results are in the same order of magnitude, and confirm the same trend.

In all cases, the hardware accelerator is the fastest method, followed by the custom instruction ($\approx 25\times$ slower for 1000 operations), and finally the software function ($\approx 50\times$ slower). This is to be expected since, contrary to the software method, the custom instruction can execute the manipulation in a single clock cycle, and the accelerator can do the same while also bypassing the cache.

Finally, we can note that all operations increase in speed when the size of the input is scaled up. Indeed, the software method and the custom instruction almost double their speed when scaling from 1 operation to 1000 operations:

- the software function goes from 5.38 μs/op to 2.60 μs/op
- the custom instruction goes from 2.66 μs/op to 1.29 μs/op

However, it is the hardware accelerator that scales the best, thanks to the burst functionality. It goes from 1.46 μs/op to 0.05 μs/op, corresponding to a speed up of almost a factor 30.

Conclusion

In this lab, we have successfully developed 3 methods to perform the required bit manipulation: a software function, a custom instruction, and a hardware accelerator. Through both software and hardware profiling, we could evaluate the performance of each method. We observe that the hardware accelerator is the fastest method, followed by the custom instruction, leaving the software implementation the slowest – and this holds for all input sizes.