Lab 2: Profiling, Custom Instruction and Accelerator

Gao Xufeng, Sun Haoxin

June 3, 2022

1 Introduction

In this lab, we use three different ways to do bit manipulation for 32-bit data in a FPGA-based NIOS-II system. To be more specific, we deploy purely software C program, custom instructions, and a hardware accelerator with DMA in the system to cope with this task. Then to evaluate the performance, we use the GNU profiler and the performance counter to measure the time consumption in each scheme. Based on the measurements, we analyze these results and try to find some insights.

This report is organized as follows. Section 2 demonstrate our top-level scheme for this laboratory. Then Section 3 introduces how we use three different ways to implement the bit manipulation for 32-bit data. Section 4 shows the profiling tools we use in this lab to do timing measurements. In Section 5, the measurement results and corresponding analysis are provided. Finally in Section 6, we conclude our lab.

2 General System Design

In this section, we demonstrate our design scheme for the bit manipulation and profiling. The system block diagram of our design is shown as Fig. 1.

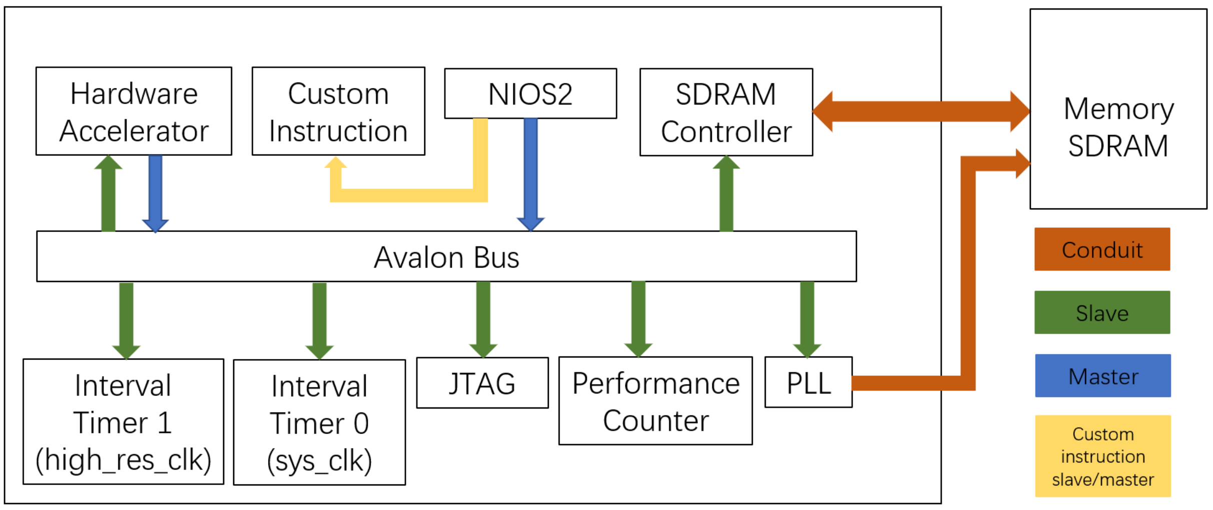

Figure 1: Block diagram of our designed system.

In our design, a NIOS-II processor controls the entire system. Because we need to perform more than $1,000,000$ times 32-bit data manipulation and print all floating points profile data in this lab, the off-chip SDRAM is enabled. In order to deal with the bit manipulation in hardware, a custom instruction module is connected to the processor, which can provide a custom operation for the given input. A DMA hardware accelerator is then combined with this custom logic to gain better performance on execution time. For the profiling measurement, we have an interval timer as the system clock, and a performance counter collaborated with another interval timer to do hardware profiling. Finally, A JTAG module is added to the system to debug and download the code.

3 Design for Bit Manipulation

In this section, we display how to use pure software C code, a custom instruction and a hardware accelerator to deal with the required bit manipulation shown below for a given 32-bit input number respectively.

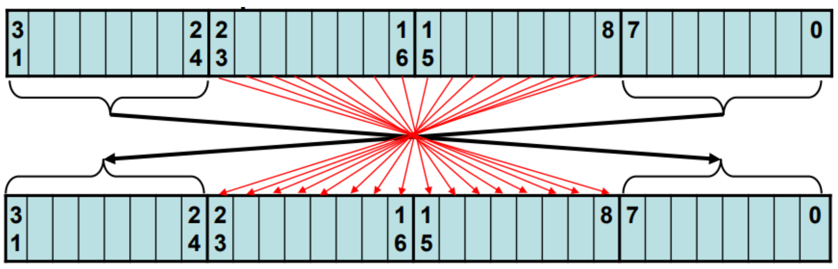

Figure 2: Block diagram of the hardware accelerator [1].

3.1 Pure Software

We first show the easiest way to do this task. From Fig. 2, we can see that the positions of the first 8 bits and the last 8 bits are swapped, and the 16 bits in the middle position are flipped. The bit-swapping can be easily implemented by using left shift and right shift operators in C. For bit-flipping operation, to improve the conversion speed of the code and avoid using a loop structure, a lookup table is created to give the bit-flipping output corresponding to an 8-bits input. The C code for this part is shown below[1]:

```c
// Imcomplete table
static uint32_t lookup_bitflip[256] = {
                0x00, 0x80, 0x40, 0xc0, 0x20, 0xa0, 0x60, 0xe0,
                0x10, 0x90, 0x50, 0xd0, 0x30, 0xb0, 0x70, 0xf0,
                0x08, 0x88, 0x48, 0xc8, 0x28, 0xa8, 0x68, 0xe8,
                0x18, 0x98, 0x58, 0xd8, 0x38, 0xb8, 0x78, 0xf8,
                ...., ...., ...., ...., ...., ...., ...., ....
};

uint32_t software_Bit_Manipulator(uint32_t input) {
    // most 8 bits
        uint32_t high = input & 0xFF000000;
    // least 8 bits
        uint32_t low = input & 0x000000FF;
    // middle 16 bits, right shift 8 bits
        uint32_t middle = (input & 0x00FFFF00) >> 8;
    // get the reverse order 16 bits
        uint32_t flipped_middle = lookup_bitflip[middle & 0x00FF] << 8 |
         ↪  lookup_bitflip[middle >> 8];
    // concatenate all bits
        return low << 24 | flipped_middle << 8 | high >> 24;
}
```

[1]A complete lookup_bitflip table can be found from submitted profile.c file

3.2 Custom Instruction

In parallel to the normal ALUs, a custom logic can also be added to achieve this specific function. Here, we write a VHDL file to create a combinational component that can do the bit manipulation for a 32-bit input value. As Fig. 3 shows, it takes two 32-bit data as inputs and the corresponding 32-bit output will be resolved in 1 clock cycle. One thing should be noted here is that the DataB is always fed with zero in our case, because the bit manipulation only takes one operand (i.e., DataA).

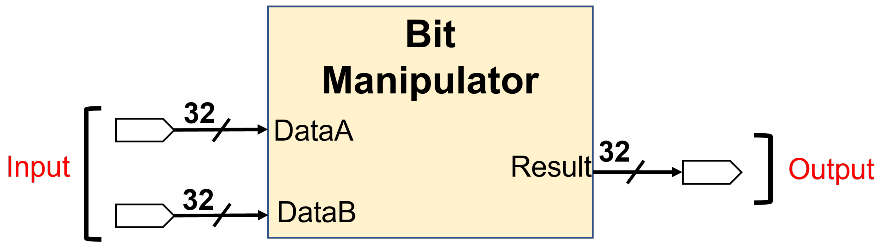

Figure 3: Block diagram of the bit manipulator.

The VHDL logic architecture for the bit manipulator is based on the assignment operator and a generate for-loop, which is the following.

```
architecture archi of BitManipulate is
begin
    -- BitSwap
    result(7 downto 0) <= dataA(31 downto 24);
    result(31 downto 24) <= dataA(7 downto 0);

    -- BitFlip
    BitFlip: for i in 0 to 15 generate
        result (8+i) <= dataA(23-i);
    end generate BitFlip;
end archi;
```

Then this component is registered as a custom instruction slave connected to the NIOS-II processor, which can be accessed in C program with macro defined functions.

```
// Macro
#define ALT_CI_BIT_MANIPULATE_0(A,B)
↪   __builtin_custom_inii(ALT_CI_BIT_MANIPULATE_0_N,(A),(B))
#define ALT_CI_BIT_MANIPULATE_0_N 0x0

uint32_t Custom_Instruction_Bit_Manipulator(uint32_t input) {
        return ALT_CI_BIT_MANIPULATE_0(input, 0);
}
```

3.3 Hardware Accelerator

Apart from the these two methods, a hardware accelerator with DMA is also designed in our system. In order to accelerate the data reading and writing processes, this module should be able to take over the Avalon Bus when needed, i.e., has the function of DMA. Therefore, it should include a master interface on the Bus to read and write the memory. Since the CPU needs to write control information to this

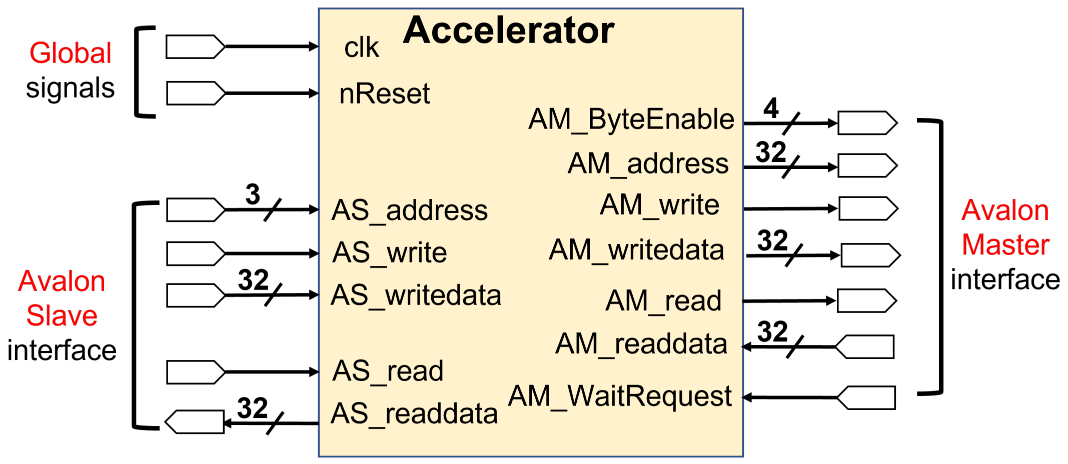

Figure 4: Block diagram of the hardware accelerator.

module, as well as read its status, it also has a Avalon Bus slave part. Thus, according to this analysis, the block diagram is shown like Fig. 4.

Then the register map can be derived like Table 1. Since this module is supposed to access the memory to read input and write back the output, write and read addresses should be provided by the processor. Then to deal with batch data, a register is used to store the total number of computations. Lastly, two registers are included for the processor to control the behavior of this module, i.e. to start computation and to check if the computation is finished.

Table 1: Register map of the hardware accelerator

Address [2:0]	Register [31:0]	Function
000	RegRDaddr	Start address for read master
001	RegWRaddr	Start address for write master
010	RegNoComp	Number of computation need to be done
011	RegStart	Computation start flag
100	RegDone	Computation done flag
others	-	Don't care

Because the module need a series of operations to do the computation, i.e., reads the input data from the memory, does bit/byte swapping, and then writes back the data to the memory, we implement a finite state machine to assist our design, which is shown as Fig. 5. When the CPU sends a start flag to the module, it will start the process, loading all parameters needed, i.e. initial addresses and number of computation in the state LoadPara. Then the DMA part will initialize a read request on the Avalon Bus in the SendReadReq state, and go to the WaitForRead state. When the master can take over the bus, i.e., the signal waitrequest is 0, meaning that it has read the input data from the memory, it can do the bit/byte swapping computation, which is exactly the same process as in the custom instruction's implementation. After computing, the module again tries to take over the bus and write the output to the memory in the SendWriteReq state. After the writing process, it either goes back to Idle, waiting for a new round of computation, or goes to SendReadReq to read a new input, depending on whether all data in this batch is finished, i.e. the register RegCount is 0.

Unlike the previous two methods, our hardware accelerator accesses the memory directly. When the data cache is enabled, data values assigned by the software program may not be written into the memory immediately (as they are hit in the cache). Thus, the data cache may lead to some inconsistency during the computing. To avoid this problem, we use the *alt_dcache_flush_all()* function to flush the entire cache before starting the computation of the hardware accelerator. An alternative way is to use the *IOWR* and *IORD* marco directly, as they will bypass the data cache and access the target memory directly.

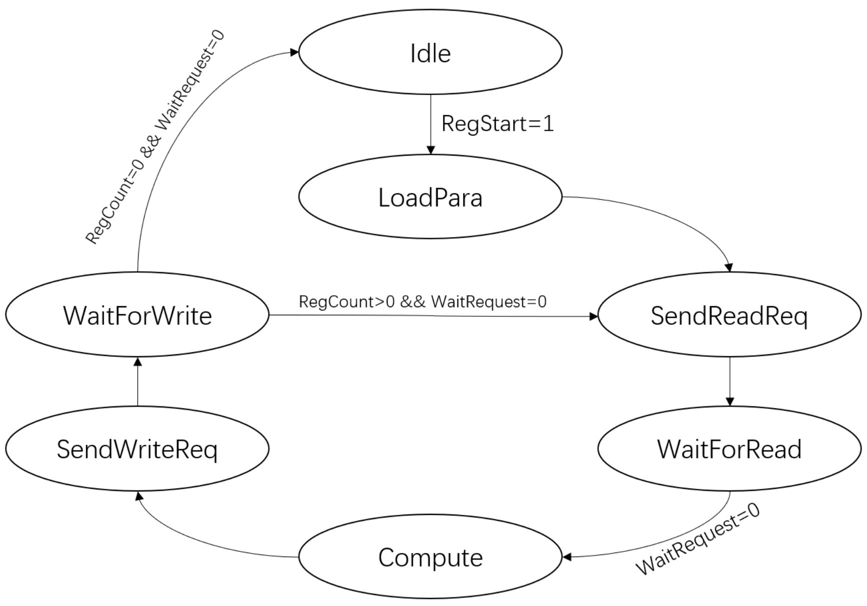

Figure 5: Finite state machine of the hardware accelerator.

4 Profiling the System

In order to evaluate the timing performance of our designed methods for bit manipulation, we deploy software and hardware profiling tools to measure the execution time of these three functions. This section will illustrate how we utilize these two tools separately.

4.1 Compiler Optimization Level

Before moving to the profiling part, we realized that the C/C++ compiler speed and size optimizations affect functions in different ways. These differences affect cache usage and resource contention, which may then change the relative start times and thus increase the required overheads of the function. For these reasons, we first set the optimization level to -O3 and then profile the code to gain the most insight on how to improve the final form of our application.

4.2 Software profiling - GNU Profiler

Firstly, we use the GNU Profiler tool provided by the compiler to do profiling. This profiler can measure the number of calls and the time consumption of each function in the call graph, which provides an overview of the entire system, although some additional overhead is added.

To make use of it, we add an interval timer in the hardware design, which is then configured as the system clock timer in the BSP settings, and set the period as 1 ms. Then by enabling the gprof itself, we generate the profiling report for the entire system, which summarizes the execution time of each function in the program. We demonstrate the measurement results by this GNU Profiler in the Section 5.

Despite its simplicity and convenience, GNU Profiler indeed has some disadvantages. For example, using the GNU Profiler will add instructions to each function call, which actually affects the behavior of the program. Besides, it may also affect the instruction cache and data cache behavior since its introduction of additional instructions and data for profiling. Another key disadvantage about it is that it can only be done with the entire system, not single functions.

4.3 Hardware profiling - Performance Counter

To solve the above-mentioned disadvantages of the GNU Profiler, a performance counter is used to evaluate the execution time. Unlike the GNU Profiler, which adds some instructions at every function in the system, profiling with a performance counter only requires one or two instructions for the *BEGIN* and *END* marco, leading to less intrusion to the original program. Besides this, hardware profiling with performance counter will be an order of magnitude faster than the software one, which is important in large-scale system design.

To utilize this module, we add a performance counter peripheral, and an interval timer as a high resolution timer whose period is 1 us in the Qsys design. In the BSP settings, the system clock timer is remained the same as the software profiler, but the timestamp timer is configured as the new-added high resolution timer. Then in the software C code, we add *BEGIN* and *END* marco around each bit manipulation function. By doing this, we can measure the execution time of each single function simultaneously, rather than have to check with the entire system. Therefore, compared with using pure software profiling, we can get more direct measurement results, which are shown in Section 5.

5 Measurement Results

We show the measurement results in this section, and try to make some analysis and insights based on these results. In order to evaluate each method completely, the amount of bit manipulations is varied from 1, 10^3, to 10^6.

5.1 Software profiling results

In this part, the relevant parts of codes used to execute bit manipulations are profiled by using the GNU profiler with a system timer. The corresponding results are shown in Table 2. However, we can see from the first column in the table that the software profiling didn't provide any useful results for the bit manipulation of a single 32-bit data, even with a high resolution timer (in 100 μs level). Fortunately, as the amount of bit manipulations increases, some exploitable results are obtained.

Table 2: Time overheads of 1, 10^3 and 10^6 manipulations by GNU Profiler

Type	Total Overheads (ms)		
	1 data	1,000 data	1,000,000 data
Pure Software	NA	3.87	3240
Custom Instruction	NA	2.37	1660
Hardware Accelerator	NA	0.50	120

5.2 Hardware profiling results

In this part, the bit manipulation is profiled by using a performance counter and a high resolution timer (in 1 μs level). All measurement results are recorded in the below Table 3.

Table 3: Time overheads of 1, 10^3 and 10^6 manipulations by the performance counter

Type	Total Overheads (ms)		
	1 data	1,000 data	1,000,000 data
Pure Software	0.03450	3.57	3020
Custom Instruction	0.01576	2.28	1589
Hardware Accelerator	0.00348	0.45	101

5.3 Observation

In general, because the GNU profiler itself adds some overheads, e.g., additional function calls and higher cache misses, the overheads measured by software profiling are slightly larger than the ones measured by hardware profiling. However, both profiling methods give the same trend, although there is a slight discrepancy between them. We can see that the hardware accelerator is the fastest method in all cases, which is followed by the the custom instruction ($\approx 20\times$ slower for $1,000$ manipulations) and pure software ($\approx 35\times$ slower for $1,000$ manipulations). This is to be expected, because the custom instruction is able to complete the bit manipulation in a single clock cycle, and the hardware accelerator can provide the same benefit while also taking the advantage of direct memory access (e.g., cache bypass). For pure software, it might require a complex sequence of standard instructions to execute a single bit manipulation, which takes more overheads. Finally, as the amount of manipulations is scaled up, we can note that the execution performance of each method improves, as summarized below.

Table 4: Comparison of improved performance of each method as the scale of manipulations increases

Type	Overhead per manipulation (μs)		
	1 data	$1,000$ data	$1,000,000$ data
Pure Software	34.50	3.57	3.02
Custom Instruction	15.76	2.28	1.59
Hardware Accelerator	3.48	0.45	0.10

For the pure software and custom instruction methods, the improved speed is almost 10 times than before when the manipulation is scaling from 1 data to $1,000,000$ data. Moreover, it is the hardware accelerator again that gives the best scaling performance, which goes from 3.48 μs per manipulation to 0.10 μs per manipulation, corresponding to a speed up of almost a factor 30.

6 Conclusions

In this lab, we design three different methods to do bit manipulation for a given 32-bit input. In the first method, we deploy a look-up table in pure C program to deal with the task. As this is a specific computation task, then we implement a custom instruction to directly compute the output result from the given input. Finally, a hardware accelerator with DMA unit is designed to gain a better performance on time consumption.

In order to evaluate the execution time of each method, we utilize software profiling - GNU Profiler with a system timer, and hardware profiling - a performance counter with a high resolution timer to do the time measurement. The results show that using pure C program is the slowest way, while the hardware accelerator is the fastest. It can be observed that with DMA unit, the hardware accelerator spends less time on data transfer, i.e., reading and writing data. In real applications where many computations are performed in real time, the timing gain obtained from hardware accelerator is quite essential which guarantees the system to response within the defined time constraints.

References

[1] René Beuchat. *"System On Programmable Chip" - NIOS2 Custom Instruction.* `https://moodle.epfl.ch/pluginfile.php/1090141/mod_resource/content/3/NIOSII_CustomInstruction_01d.pdf`. Accessed: 2022–04-06.