The Codesign Challenge

Objectives
In the codesign challenge, your task is to accelerate a given software reference implementation as much as possible. You can use any of the previously discussed techniques to accelerate the implementation: optimize the software, build a coprocessor, optimize the hardware/software communication. The constraints of your implementation are

1. that it must be completed by 12/01/2008 at 5:00PM;
2. that it must run correctly on the Spartan 3E starter kit;
3. and that it follows the given testing procedure to demonstrate the performance of your implementation.

All solutions will be strictly ranked and graded accordingly. Thus, there will be a single 'best' solution, a single 'second' solution, and so on. There is no perfect solution; the only 'perfect' solution is the one that ranks higher than all the others. The quality of your solution will be evaluated using the following criteria:

1. The resulting clock cycle count of your implementation, with a ‘clock cycle’ corresponding to one tick of a PLB Timer module clocked at 50MHz.
2. The product of the Slice Usage (Spartan 3E slices) and the Clock Cycle Count.
3. The time when the solution was turned in (before the deadline, but earlier is better).

The clock cycle count is most important, the area-time product is second-most important, and the design time is third-most important. Faster (but correct) designs will always win. For solutions which have a cycle count within 5% of each other, the area-time product (item 2) will be used as a distinctive factor. In case two solutions are within 5% of each with respect to item 1 as well as item 2, the time of posting the solution will be used to resolve the ranking of the two designs.

Hence, it is in your interest to find the highest possible performance that can still be accommodated on a Spartan3 board, and to find that solution as quickly as possible.
Assignment: Counting Bits in an Array

The task is to implement a bit counting algorithm for a large array of pseudo-random data. The bit count of the array is simply the total number of bits that are '1'. The array contains 16K 32-bit integers. The reference implementation of the bit counting algorithm is shown below. The bit count of the array $tdata[]$ is obtained by calling $countbits$.

```c
#define MAXDATA 16384
unsigned tdata[MAXDATA];

static int n;      // index of array element being counted
static int idx;    // index of bit in array element
static int endofdata;
void initbit() {
    n = 0;
    idx = 0;
    endofdata = 0;
}

unsigned nextbit() {
    unsigned bit;
    bit = (tdata[n] >> (31 - idx)) & 1;
    idx++;
    if (idx == 32) {
        idx = 0;
        n++;
        if (n == MAXDATA)
            endofdata = 1;
    }
    return bit;
}

unsigned countbits() {
    unsigned len;
    len = 0;
    initbit();
    while (endofdata == 0)
        if (nextbit())
            len++;
    return len;
}
```
The reference architecture to execute this design is shown below. A Microblaze v7 is attached to a PLB bus, together with a timer and a DDR memory controller. Initially the 16K array is mapped in off-chip DDR memory.

To complete this assignment, you will start from a reference implementation that performs the bit counting process in software. The reference implementation performs the following steps:
1. Fill the \texttt{tdata[ ]} array with pseudorandom data
2. Start the timer
3. Call the \texttt{countbits()} function
4. Stop the timer
5. Display the resulting bitcount and the resulting cycle count.

You can modify the \texttt{countbits()} function in any desired way to improve the resulting cycle count. This includes for example the following optimizations.
- Adding hardware coprocessors on the PLB bus, or on the FSL interface of Microblaze.
- Optimizing the software using compiler flags or source code transformations.
- Re-allocating memory segments from off-chip to on-chip memory.
- Adding a cache to the Microblaze processor.
- Adding Microblaze processors.
- Adding scratch-pad memories to hold temporary results.
- Using Direct-Memory Access to overlap computations with memory-accesses.

However, you have to keep in mind there are limitations, including the following.
- The resulting design must fit on your Spartan 3E starter kit.
- The clock frequency of the design must remain at 50 MHz.
- You cannot 'tweak' the testvectors. Your implementation must really count all the bits of 16384 integers. Your design may be tested with a different testbench then the one you are given. The resulting bitcount should be exactly right.

If you violate the above limitations, your design will be disqualified from the ranking. In terms of ranking, this puts your design behind the lowest-ranked working solution. In
case of doubt if a certain shortcut is allowed or not, please send email or post a question on the bulletin board.

**How to start**

On Blackboard, download the baseline reference implementation. This design will run directly on your Spartan kit. Start by studying the reference implementation software. This reference implementation calls `countbits()`. Running the design on your board will show you the following result.

That is, counting the bits in `tdata[]` requires over one billion cycles. Eventually, you will need to decrease the cycle count as much as possible while maintaining the functionality.

- If you build a coprocessor, it is highly recommended to **construct a cosimulation model** of your design using GEZEL. While you can develop coprocessor hardware directly in VHDL, it will require you to take care of many details at once. Going through cosimulation first enables you to test your idea before taking it to the board.
- Also, when developing hardware, **first test your ideas on ‘small’ designs**, such as a `tdata[]` with 128 elements rather than 16K. When the low level components work fine, next verify how well it scales up.
- Also, carefully **consider tradeoffs**, as mentioned on the previous page. There are obviously many more implementation alternatives than you have time to try out. Thus, you will have to think before you implement, and to experiment to find the largest acceleration as quickly as possible.
- Always **focus on the bottleneck** in the overall system. Remember the earlier examples we discussed. Hardware parallelism is useless if you cannot provide data sufficiently fast.
- Also, **make use of your homework assignments/solutions** to see examples on how a memory-mapped interface or an FSL interface can be created.
What to turn in

By the deadline, post the following information on Blackboard.

- A short report (no more than 4 pages) that summarizes the main characteristics of your design. Your report must at least contain the following table.

<table>
<thead>
<tr>
<th>Area of the baseline design (slices)</th>
<th>Performance of the baseline design (cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Area of the optimized design (slices)</td>
<td>Performance of the optimized design (cycles)</td>
</tr>
</tbody>
</table>

In addition, you are encouraged to discuss the trade-offs you made, to provide a block-diagram of the resulting system, to describe the architectural features of the hardware coprocessor you made. Also include a screenshot of the design as it executes in XMD.

- If you developed a cosimulation model in GEZEL, also provide the cosimulation model (C driver and FDL file).

- The optimized implementation in XPS. Before posting the design on Blackboard, make sure you run Project->Clean All Generated Files. Then, zip the project directory and post it on Blackboard.

Grading

Your design will be graded as follows:

- Your ranked performance counts for 75% of the grade
- Your report counts for 25% of the grade.

Your cosimulation model and XPS project will be run to verify the correctness of the statements you make in the report.

The performance ranking criteria described above will be used. Having a working solution is not sufficient to obtain a good grade. Having a speed improvement of, for example, 3 times, is not sufficient to obtain a good grade. The best grade will go to the design with the highest performance. All other designs will be strictly ranked according in relation to the best one. This strict ranking rule is introduced based on the observation that, under free market conditions, better designs have a better chance to make it into a product.

However, don’t let this rule spoil the fun.

This is your chance to explore new ideas and to try out what you have learned in this class! We will discuss the design in detail in the class of 3 December.