The Codesign Challenge

Objectives

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

1. that it must be completed by 11/26/2007 at 5:00PM.
2. that it must run correctly on the Spartan 3E starter kit.
3. that it follows the given testing procedure to demonstrate the performance of your implementation.

The quality of your design 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 an OPB Timer module clocked at 50MHz.
2. the area of your design, expressed in slices of the Spartan3E FPGA.
3. the time when you turned in the solution (before the deadline, but earlier is better).

The clock cycle count is a first-order criterium, the area is a second-order criterium, the design time is a third order criterium. Faster (but correct) designs will always win. For clock cycle counts that lie within 1% of each other, area will be used as a distinctive factor. For example, given four designs A, B, C, and D as shown below, the ranking would be as follows, from best to worst: D, B, C, A. In case the area as well as the cycle count are within 1% of each other, then the time of posting the solution will be used to resolve the ranking of the two designs.

Thus, all designs will be strictly ranked according to these criteria. It is in your interest to try and find the highest possible performance that can still be accommodated on a Spartan3 board, and to find that solution as quickly as possible.
Assignment: Coordinate Rotation Digital Computer (CORDIC)

The task is to implement a CORDIC algorithm as efficiently as possible. CORDIC is often used in digital hardware to implement trigonometric functions. The CORDIC kernel implements a vector rotation operation.

In a two-dimensional space, a vector rotation starts from a vector (X,Y) and rotates it over an angle phi as follows:

\[ x' = x \cos(\phi) - y \sin(\phi) \]
\[ y' = y \cos(\phi) + x \sin(\phi) \]

This can be rearranged to:

\[ x' = \cos(\phi)[x - y \tan(\phi)] \]
\[ y' = \cos(\phi)[y + x \tan(\phi)] \]

An efficient implementation of this formula is possible by restricting the rotation to amounts of angles for which \( \tan(\phi) = \pm 2^{-i} \). Thus, we should ensure that the tangent of the angle is a power of two. Under that condition, the above rotation formulas require only shift-operations to implement the multiplication with \( \tan(\phi) \). We call the rotation over such an angle an elementary rotation.

An arbitrary angle can now be approximated as a sequence of elementary rotations, much in the same way as the individual bits in a bitvector can express weights to approximate an integer number.

This idea is illustrated in the figure above. We need to implement a rotation over angle \( \beta \). We start with an initial vector \( v_0 \) at (1,0). The first elementary rotation is over an angle tan\(^{-1}(0.5)\). This rotates \( v_0 \) counter-clockwise to \( v_1 \), using the rotation formulas given
above. The next elementary rotation would be over an angle $\tan^{-1}(0.25)$. Again, this would be a counter-clockwise rotation, such that we decrease the error between the desired rotation angle $\beta$ and the approximations in terms of elementary rotations. $v1$ now moves to the position $v2$. The next rotation, over $\tan^{-1}(0.125)$, would be clockwise, since $v2$ has moved beyond the desired rotation $\beta$. By using increasingly smaller elementary rotations, we would obtain an increasingly better approximation.

Therefore, we can express the rotation formulas above using a set of difference equations.

\[
x_{i+1} = K_i [x_i - y_i d_i 2^{-i}]
\]
\[
y_{i+1} = K_i [y_i + x_i d_i 2^{-i}]
\]

with

\[
K_i = \cos(\tan^{-1} 2^{-i}) = \frac{1}{\sqrt{1 + 2^{-2i}}}
\]

\[
d_i = \pm 1
\]

At each iteration, a smaller rotation angle is selected, and a decision to rotate forward or backward is made ($d_i = \pm 1$) such that we obtain a better approximation of the actual rotation angle in terms of elementary rotations. Note that the constants in these formulas only depend on elementary rotations, and as such they can be evaluated upfront and stored as constants.

In CORDIC implementations, the $K_i$ factors are not applied at each rotation, but rather they are collected into a single scaling factor $A$. For a large number of (increasingly smaller) elementary rotations, $A$ converges to 1.647 and is given by

\[
A = \lim_{i \to \infty} \prod \sqrt{1 + 2^{-2i}}
\]

To find how well the target rotation angle is approximated by elementary rotations, we can also include an ‘angle-accumulator’ into the iterations, defined by

\[
z_{i+1} = z_i - d_i \tan^{-1}(2^{-i})
\]

This angle accumulator expresses the difference between the target angle and the series of elementary rotations.
CORDIC algorithms are used in two possible modes of operation.

- In the rotation mode, we start with a desired rotation angle and rotate a given vector over that angle. At each iteration, the decision to rotate counter-clockwise or clockwise is made based on the sign of the angle accumulator. The objective is to drive the angle accumulator to zero. The result of the rotation mode is a given vector rotated over a given angle.

- In the vector mode, we start with a given vector and rotate that vector until the vector is aligned with the X axis. At each iteration, the decision to rotate counter-clockwise or clockwise is made by the sign of the Y component of the vector. The objective is to drive the Y component to zero. The result of the vector mode is the angle of a given vector.

**CORDIC implementation on Spartan 3E Starter Kit**

The codesign challenge is described by the following initial architecture.

In a DDR Ram, three 64 KWord arrays are stored. The objective is to rotate a unit vector (1,0) over all the angles expressed in `target_angle`[], and store the result of each rotation in `result_X`[] and `result_Y`[]. The performance of your design is measured as the time it takes to complete this set of rotations (including reading from/writing to DDR). To accelerate the design, you can modify the hardware as needed (add coprocessors, develop efficient data transfer techniques, etc).
You design will be tested using a test program (running on Microblaze) as described above. Initially, the microblaze will generate 64K random target angles. Next, it will collect the execution timing for 64K rotations on two cordic functions. The first is a reference implementation in software (reference_cordic). The ratio of the two cycle counts determines the relative speedup obtained by your implementation. Note that this method of speedup measurement is relatively independent of the compiler optimization level, since the -O2 flag will benefit the reference implementation as well.

Finally, your design results are verified against the golden reference. For a valid solution, zero errors are required (i.e. if your solution shows a single error, it is automatically moved to ‘lowest rank’ of all designs returned by the class).

The CORDIC reference algorithm is implemented using fixed-point arithmetic and is expressed using integers. A fixed-point data type <32,28> is used. In this data type, the value 1 is expressed as (1 << 28). The scaling factor allows expression of fractional values. For example, 0.75 is expressed as:

\[ 0.75 = 0.5 + 0.25 = (1 \ll 27)_{<32,28>} + (1 \ll 26)_{<32,28>} = 671,088,640_{<32,28>} \]

For the verification process described above to succeed, your accelerated CORDIC implementation must have the same bit-accuracy as the reference CORDIC implementation.
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 uses calls to golden_cordic in order to implement the your_cordic function. Eventually, you need to accelerate your_cordic as fast as possible.

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, initially test your ideas on ‘small’ designs, such as 100 rotations (rather then 64K). When the low level components work fine, next verify how well it scales up to 64K rotations.

Also, carefully consider tradeoffs.
- You can move part of the golden_cordic function to hardware, or move the complete golden_cordic to hardware.
- You can use a memory-mapped interface, or use an FSL interface.
- You can write VHDL or GEZEL code (If HDL are unfamiliar to you, please stick to GEZEL).
- You can implement the golden_cordic in hardware as a completely unrolled function, or design it in hardware as an FSMD, using multiple control steps.
- You can send arguments serially or in parallel.
- You can provide arguments with a processor (Microblaze) or through DMA.

There are obviously more implementation alternatives than the allocated design time. Thus, you will have to think before you implement, and 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 unless the datapipes into that hardware has sufficient bandwidth.

Also, make use of your homework assignments/solutions to see examples 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 trade-offs you made, to provide a block-diagram of the resulting system, to describe the architectural features of the hardware coprocessor you made, and so on. Also include a screenshot of the design as it executes, such as shown below.

- 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 based on the numbers you report, in combination with the cosimulation model and the XPS project you will turn in. The cosimulation model, and the XPS project may be run to verify the correctness of the statements you make in the report.

The ranking criteria described above will be used. Having a working solution is not sufficient to obtain a full grade. Having a speed improvement of, for example, 3 times, is not sufficient to obtain a full grade. The full 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 November 12, and partly in the class of November 14.