Implementation of Complex Division Custom Instruction Hardware on Nios-II Processor
Addanki Purna Ramesh, B. Sandeep Kumar
Department of ECE, Sri Vasavi Engineering College, Pedatadepalli, Tadepalligudem, India

Abstract—As increasing the clock frequency leads to unmanageable heat and power dissipation the search is on for another way to get more power efficient and faster embedded systems. Given that chip area is also a constraint, we investigate the addition of a custom instruction to the processor instruction set which enables the execution of an efficient complex division. We have implemented software for complex floating point division using the custom instructions added.

Index Terms–Complex Numbers, Division, NiosII.

I. INTRODUCTION
Unlike real multiplication or real division, mathematical operations for complex numbers are usually provided by slow software. This is a common approach since the usage of complex operations is not widespread and its implementation is considered to be expensive [1]. However, fast complex dividers are necessary to drive an increasing number of applications, such as signal processing systems for image and audio manipulation, GPS, and multi-antenna systems. For example, Space Telescope Science Data Analysis System (STSDAS) is a product of the Space Telescope Science Institute, which is operated by AURA for NASA [2]. STSDAS offers math libraries for image analysis, including stdsas. analysis.fourier.carith, which is used to multiply or divide two complex images [3]. Enhancing the speed of such complex divisions can be realized using various techniques such as frequency scaling to speed up computation and co-processors to perform parallel computation. The clock frequency can be increased in order to speed up software. If the complex computation is performed by a co-processor then additional components must be added to the system bus and they are treated as slaves by the CPU. This concept requires additional commands to access the external divider co-processor, and hence this approach causes a significant communication overhead.

In our approach we present a small pipelined divider for complex numbers realized as an Instruction Set Extension (ISE). The benefits of appropriately selected ISEs are explained in [4], and include lower power consumption and faster application execution time. The hardware for our complex divider is implemented in a NIOS II configurable and extensible processor. Unlike the base instruction set, the complex divider has a MIMO (Multiple Input Multiple Output) data path, and so an advanced method for feeding and returning variables was adapted from [5] in order to fit the data path into the NIOS II ALU. The complex divider instruction contains pipelined multiplier, adder and divider components from the Altera LPM library. The C compiler in Alteras tool chain can access a generic instruction such as the complex divider using a template which looks like a function call. Including this template saves the user from inlining assembly code.

The division can be expressed by following equation (A...D are input values, E and F are output values).

\[
(E + jF) = \frac{(A + jB)}{(C + jD)} \quad \text{------------------------ (1)}
\]

In order to solve this equation we have to extend the denominator and numerator with the complex conjugate as follows

\[
(E + jF) = \frac{(A + jB)(C - jD)}{(C + jD)(C - jD)} = \frac{(AC + BD)(C^2 + D^2)}{(C^2 + D^2) + j(AD - BC)} \quad \text{------------------------ (2)}
\]

The last equation has to be translated in hardware. This can be achieved by disassembling the equation into the following parts.

- Six multiplications: A * C, B * D, A * D, B * C, C* C and D* D
- Two additions and one subtraction: AC+BD, AD–BC and CC + DD
- Two divisions: (AC+BD) / (CC+DD) and (AD–BC) / (CC+DD)

The computations listed above can be distributed to one multiplier, one adder and one divider. To use the given resources efficiently these three components get their data pipelined, that means the multiplier is fed with a stream of multiplications. The computed results are passed to the adder and to the divider. Therefore we get a total execution time of 5 cycles (multiplier) + 7 cycles (adder) and 6 cycles for the divider. Additionally we have to consider the 6 pipelined input values and a delay of 3 clock cycles to wait for the appropriate data (the data...
depends on the equations that are being considered). So we get a total execution time of 27 clock cycles. The state machine for the complex divider is controlled by the following signals.

- **Clk**: The state machine of the custom instruction is sensitive to each rising edge of this clock.
- **Clk en**: The clock enable signal controlled the activity of the multiplier, divider and adder so that they store their results.
- **Reset**: If an active reset signal occurs, the entire complex divider will be reset.
- **Data/Datab**: both signal vectors are inputs and have a width of 32 bits. In this system these signal vectors get the first data pair in the first and the second data pair in the second clock cycle.
- **Result**: After 26 clock cycles the first data (real part of the result) is available on this port. The next clock cycle can release the imaginary result.
- **Start**: The start signal triggers the state machine.
- **Done**: The done signal is set when the first result is ready. It stops being asserted during the first clock cycle of the state machine activation. The figure 1 shows the diagram of Adding a custom instruction to the NIOS II.

![Adding a custom instruction to the NIOS II](image.png)

**II. SOFTWARE INTERFACE**

The designed hardware for complex division can be used easily in assembly or C/C++ code. In assembly we can use a predefined instruction called custom. This instruction has the following form, custom N, rC, rA, rB C, A and B represent the register numbers for result, dataa, and datab, while N defines the type of custom instruction. Each value A, B and C represents one of the 32 registers in the register bank.

```
Float n = ALT_CI_COMPLEX_DIV_INST(0,in_A,in_C); //(1)
out_real = ALT_CI_COMPLEX_DIV_INST(1,in_B,in_D); //(2)
out_imag = ALT_CI_COMPLEX_DIV_INST(0,0.00,0.00); //3
```

In the first step the values A and C are loaded. Hence we have to signal that we want to use the custom instruction in mode 0 using the first parameter. Since this is the first execution of this instruction no result will be returned, although the result is sent to a dummy variable. In the second step the values B and D are forwarded to the macro. After 26 clock cycles the instruction returns the real value of the output. In the last step we execute
the custom instruction in mode 0 again to receive the imaginary part while a new value can be passed to the macro if this is necessary.

III. RESULTS

The simulation results of floating point complex division are shown on figure 2.

![Simulation Results of Floating Point Complex Division](image)

IV. CONCLUSION & FUTURE SCOPE

This paper presented a useful hardware divider for complex numbers that can be used in time critical applications such as signal processing. Pipelining was used to dramatically reduce the number of required hardware components in the ISE. An ISE can be simply implemented in existing NIOS II cores and may lead to a significant increase in the computational speed of the hardware divisions. For future work we may investigate further changes to the complex divider to support not only division but also multiplication and other types of multicycle computation.

REFERENCES


