

# Pipelined Datapath for an IEEE-754 64-Bit Floating-Point Jacobi Solver

Gerald R. Morris<sup>1</sup> and Viktor K. Prasanna<sup>2</sup>

Department of Electrical Engineering  
University of Southern California, Los Angeles, CA  
{grm,prasanna}@usc.edu

## Abstract

Solving linear equations is essential for certain embedded applications such as adaptive beam forming and synthetic aperture radar. When direct methods like Cholesky factorization are not viable, it becomes necessary to use an iterative approach. Even when the convergence of the basic iterative methods like Jacobi or Gauss-Seidel cannot be guaranteed, they are often used as preconditioners for more advanced methods like generalized minimum residual (GMRES) [1]. This paper presents a *binary tree datapath* for an IEEE-754 64-bit floating-point Jacobi iterative solver. The datapath component is written in VHDL; it is highly pipelined and parallelized for efficient execution on FPGAs. The Jacobi datapath circuit is implemented using a Xilinx Virtex-II Pro as the target device; size, clock speed, and peak GFLOPS statistics are presented.

## Introduction

Within the high performance embedded computing domain, FPGAs are no longer restricted to their traditional application space as substitutes for ASICs. Contemporary research spans a broad range of topics such as application design and synthesis [2], benchmarking [3], variable-precision floating-point operations [4], and higher-radix floating-point representations [5]. The mega gate counts, arithmetic capability, and other features of modern FPGAs have precipitated research into general-purpose linear algebra routines [6], as well as floating-point kernels from specific problem domains such as molecular dynamics [7]. FPGAs may soon have an order of magnitude peak floating-point performance over CPUs [8]. SRC Inc. offers a compact, high-end embedded computer that has user-programmable FPGA-based acceleration capability [9]. FPGA-based floating-point computational kernels are becoming a reality within the high performance embedded computing domain.

## The Jacobi Iterative Method

The Jacobi iterative method for solving  $A\mathbf{x} = \mathbf{b}$  involves splitting  $n \times n$  matrix,  $A$ , into lower triangular, upper triangular, and diagonal matrices and casting the resulting equation in the form of an iteration,  $\mathbf{x}^{(\delta+1)} \Leftarrow f(\mathbf{x}^{(\delta)})$ , where  $\delta$  is the iteration index. Plugging  $A = L + U + D$  into  $A\mathbf{x} = \mathbf{b}$  yields the vector form of the Jacobi iteration:

$$\mathbf{x}^{(\delta+1)} \Leftarrow D^{-1}(\mathbf{b} - (L + U)\mathbf{x}^{(\delta)}) \quad (1)$$

The Jacobi iteration can also be expressed in point form:

$$x_i^{(\delta+1)} \Leftarrow \frac{1}{a_{ii}} \left( b_i - \sum_{j=1, j \neq i}^n a_{ij} x_j^{(\delta)} \right) \quad (2)$$

Equation 2 can be trivially vectorized “since the updates could in principle be done simultaneously” [10]. As such, it is an ideal candidate for implementation within an FPGA.

## Design

To motivate the discussion, consider the simplified block diagram of the *complete* Jacobi iterative solver shown in Figure 1. This paper presents the design of the *binary tree datapath* shown in the center of the figure. For simplicity, a datapath width,  $k = 4$ , is shown.

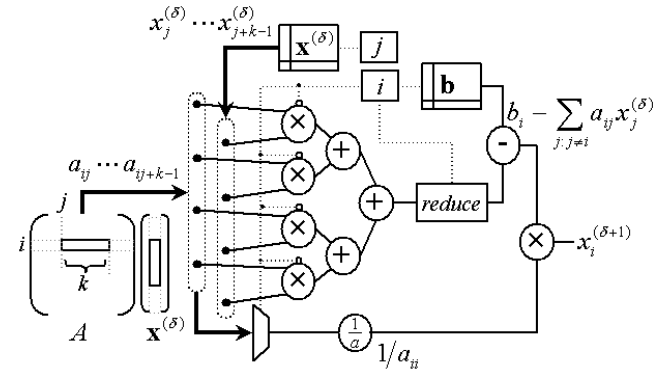


Figure 1: Jacobi Iteration Block Diagram

During a multi-cycle initialization, the  $\mathbf{b}$  and  $\mathbf{x}^{(\delta)}$  vectors are read in and stored internally. Then, the first  $k$  elements of row  $\mathbf{a}_i$  (a *sub-row*) are read in and held at the multiplier inputs along with the corresponding  $k$  elements of  $\mathbf{x}^{(\delta)}$  that were fetched from storage. On the next clock, the pipeline ingests the next sub-row, and so forth until the entire row has been dispatched. At this point, the next row begins to enter the datapath. The datapath reduces the inputs and emits one partial value per clock. The partial sums for row  $i$  are accumulated by the *reduce* circuit to generate the summation shown in Equation 2. This value is subtracted from  $b_i$  and the difference is fed into the output multiplier with  $1/a_{ii}$  to produce  $x_i^{(\delta+1)}$ . Counter,  $i$ , controls the datapath such that the  $a_{ii}x_i^{(\delta)}$  are set to 0;  $i$  also selects the  $a_{ii}$  values into the divider, the  $b_i$  values into the subtracter, and allows *reduce* to know when a new row has started. Counter  $j$  ensures the proper  $k$  values are read from the  $\mathbf{x}^{(\delta)}$  store.

<sup>1</sup> Supported in part by the DoD High Performance Computing Modernization Program.

<sup>2</sup> Supported by the United States National Science Foundation under award No. CCR-0311823 and in part by award No. ACI-0305763.

Two *reduce* circuits are presented in [11], the sub-row idea and control algorithms are given in [6], and the floating-point units (FPU) are described in [12]. This paper presents the design of the binary tree datapath, *not* a complete Jacobi iteration circuit. Due to both I/O and slice limitations, the widest datapath that fits on the target devices is  $k = 8$ . One of the challenges of this design was that the tools do not support IEEE-754 representation. Scrofano's *float2hex*, and *hex2float* utilities [7] were modified to generate VHDL test vectors.

## Experiments

Using the Xilinx ISE 6.3.03, Synplicity Synplify Pro 7.7.1, and Model Technology ModelSim XE II 5.8c development tools, with Xilinx Virtex-II Pro as target devices, the datapaths were implemented in the VHDL language. Extensive optimizations such as reentrant routing, and floor planning were not used. A 100 MHz timing constraint was placed at the top of the design hierarchy. The elided ModelSim diagram in Figure 2 shows a latency of 52 clock cycles, which is consistent with the analytical result: 10 cycles for the multiplier stage, and 14 cycles for each of the  $\lg(k)$  adder stages. Note that each  $k$ -vector in  $\mathbf{a}_i$  and the corresponding  $k$ -vector in  $\mathbf{x}^{(\delta)}$  (not shown) results in a single tree output value. If the  $k$  output values are added together, one gets the indicated summation.

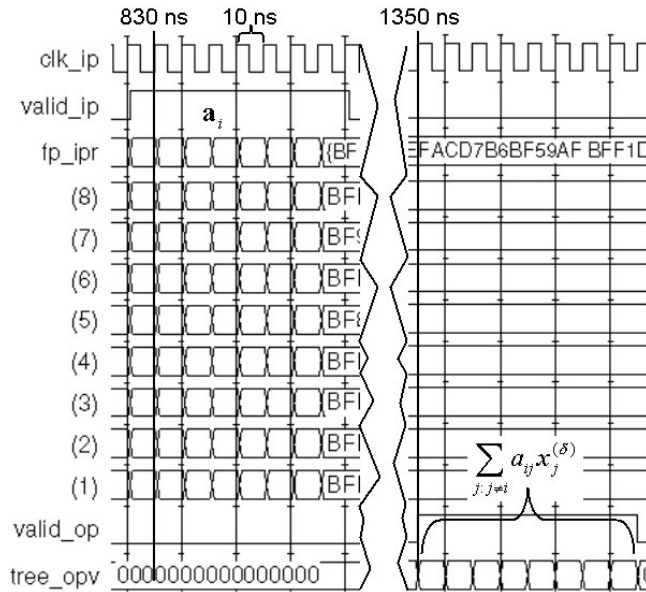


Figure 2: Latency

Xilinx place and route reports show the datapath has the size and clock speed characteristics listed in Table 1.

Table 1: Place & Route Statistics

device	$n$	I/O	slices	clk
xc2vp100-6	8	581	20527	9.351 ns
xc2vp70-6	8	581	20188	9.696 ns

If the vector length,  $n \gg k$ , then pipeline latency can be ignored when calculating peak GFLOPS. There are  $k$  multipliers, and  $k - 1$  adders. This corresponds to 15 IEEE

64-bit floating-point operations per clock cycle. The datapath can run at 10 ns, so there is a peak rating of 1.5 GFLOPS.

## Future Work

Implement complete Jacobi iteration using sub-row design as in [6] coupled with the serial reduction techniques described in [12]. Implement sparse matrix version.

## Acknowledgements

Thanks to ERDC MSRC computational scientists, Tom Oppe, Bill Ward, and Alvaro Fernandez for clarifying some theoretical issues. Thanks to Ron Scrofano for his *float2hex* and *hex2float* utilities.

## References

- [1] Y. Saad, *Iterative Methods for Sparse Linear Systems*. Boston: PWS Publishing Company, 1985.
- [2] J. Frigo, M. Gokhale, and D. Lavenier, "Evaluation of the streams-C C-to-FPGA compiler: an applications perspective," *2001 ACM/SIGDA 9th International Symposium on Field Programmable Gate Arrays*, Feb 2001.
- [3] S. Kumar, L. Pires, S. Ponnuswamy, C. Nanavati, J. Golusky, M. Vojta, S. Wadi, D. Pandalai, and H. Spaanenberg, "A benchmark suite for evaluating configurable computing systems—status, reflections, and future directions," *2005 ACM/SIGDA 13th International Symposium on Field-Programmable Gate Arrays*, Feb 2005.
- [4] M. Leeser and X. Wang, "Variable precision floating-point division and square root," *8th Annual High Performance Embedded Computing Workshop*, Sep 2004.
- [5] B. Catanzaro and B. Nelson, "Higher radix floating-point representations for FPGA-based arithmetic," *13th IEEE Symposium on Field-Programmable Custom Computing Machines*, Apr 2005.
- [6] L. Zhuo and V. K. Prasanna, "Sparse matrix-vector multiplication on FPGAs," *2005 ACM/SIGDA 13th International Symposium on Field-Programmable Gate Arrays*, Feb 2005.
- [7] R. Scrofano and V. K. Prasanna, "Computing Lennard-Jones potentials and forces with reconfigurable hardware," *International Conference on Engineering Reconfigurable Systems and Algorithms*, Jun 2004.
- [8] K. Underwood, "FPGAs vs. CPUs: Trends in peak floating-point performance," *2004 ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays*, Feb 2004.
- [9] SRC Computers, Inc., <http://www.srccomp.com>
- [10] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, 2nd Edition, SIAM, 1994.
- [11] G. R. Morris, L. Zhuo, and V. K. Prasanna, "High-performance FPGA-based general reduction methods," *13th IEEE Symposium on Field-Programmable Custom Computing Machines*, Apr 2005.
- [12] G. Govindu, L. Zhuo, S. Choi, and V. K. Prasanna, "Analysis of high-performance floating-point arithmetic on FPGAs," *11th Reconfigurable Architectures Workshop*, Apr 2004.