



Iterative Matrix Equation Solver for a Reconfigurable FPGA-Based Hypercomputer®

William S. Fithian
Star Bridge Systems
NASA Langley Research Center
Hampton, VA 23681

Samuel Brown
Star Bridge Systems
Midvale, UT 84047

Robert C. Singleterry, Ph.D.
NASA Langley Research Center
Hampton, VA 23681

Olaf O. Storaasli, Ph.D.
NASA Langley Research Center
Hampton, VA 23681

September 5, 2002

Iterative Matrix Equation Solver for a Reconfigurable FPGA-Based Hypercomputer®

William S. Fithian, Star Bridge Systems, Inc., at NASA Langley Research Center, Hampton, VA
Samuel S. Brown, Star Bridge Systems, Inc., Midvale, UT
Robert C. Singleterry, Ph.D., NASA Langley Research Center, Hampton, VA
Olaf O. Storaasli, Ph.D., NASA Langley Research Center, Hampton, VA

Abstract:

This paper details efforts to program a basic, Jacobi iterative matrix equation solver on a reconfigurable field-programmable-gate-array- (FPGA) based computer manufactured by Star Bridge Systems, Inc. An FPGA is an array of Silicon gates on a computer chip that can be reconfigured by programmers in the field for specific problems. In addition to the dynamic nature of FPGA hardware (which enables optimization to specific applications), FPGAs are also inherently parallel. This enables programmers to make as many operations run concurrently as they can physically fit on the FPGA, using its resources more efficiently than is possible on a CPU. Jacobi iteration, a relatively simple matrix equation solution algorithm, is well suited for implementation on an FPGA computer because of its inherent parallel nature and repetitive use of the same arithmetic operations. A Jacobi iterative matrix solver was created and tested successfully on a 3x3 matrix. The basic components were developed for a more parallel implementation of the solver, planned to be a useful tool for future engineering applications.

1. Introduction:

Solving systems of simultaneous linear equations is a common task for engineers to perform when modeling natural phenomena. Frequently, applications containing thousands or even millions of equations and variables require computer solutions because they are too complicated to be solved by hand. Current state-of-the-art matrix equation solvers for supercomputers utilize parallel processing (performing numerous steps in an algorithm simultaneously) to increase solution speed. To achieve this parallelism, they exploit multiple CPUs capable of communicating with each other. However, this type of parallelism is inefficient, using only a small fraction of CPU resources at any given time, while the rest of the silicon lies idle and wastes power. CPUs are designed to be general and capable of performing any function they will ever need to perform. Therefore, they contain many resources that are rarely used. In addition, the interprocessor communication time required by traditional matrix equation solvers seriously limits the number of processors that may operate efficiently in parallel (Reference [2]).

Field-programmable gate arrays (FPGAs), are a relatively new type of computer chip with various properties that set them apart from more traditional CPUs. They are basically large arrays of binary logical gates that can be programmed and connected to each other in any configuration. These gates can be configured and reconfigured an arbitrary number of times, allowing the creation of customized processors specifically programmed for an application.

Programmed FPGAs save chip space because they are specific rather than general. FPGAs also allow a great deal of inherent parallelism (Reference [3]). The number of processors that can exist and operate concurrently on one FPGA chip is limited only by the number of computational logic blocks (CLBs), or groups of gates, that exist on that FPGA to be programmed. This allows FPGAs to be significantly more efficient than CPUs in their use of processor space. However, the efficiency derived from optimizing hardware to a task is only useful if users have access to the hardware. Learning to design and optimize chips is normally a long and tedious process not available or feasible to most programmers. Fortunately, Star Bridge's breakthrough software, Viva, performs all the gate optimization and simplifies programming algorithms on FPGAs (Reference [3]).

1.1 Viva:

Viva, Star Bridge's revolutionary innovation, erases the traditional line between hardware and software, merging both into what is called "gateway." Viva is actually software that runs on a traditional CPU, allowing users to program algorithms using a click-and-drag graphical user interface. The user creates a graphical program called a "sheet" with his mouse, then directs Viva to "synthesize" the sheet. This synthesis is more than compiling the program into executable machine code; Viva creates routing instructions for reconfiguring the FPGA hardware into an optimized, inherently parallel, application-specific chip ready to run the Viva program. Viva is a major advance because

it is a high-level graphical language that gives chip designing capabilities to users who lack the previously required time or expertise to create their own application-specific integrated circuits (ASICs) (Reference [3]).

Viva's highly intuitive graphical user interface makes inherent parallelism a reality for users, and gives programs the look of flow charts. By clicking, dragging, and dropping library objects located in the right sidebar of the graphical user interface, the user can create input horns, output horns, functions with their own inputs and outputs, and pipes that carry data to and from all of these objects. Functions are represented as rectangular boxes with plug-in points located on the left and right for inputs and outputs, respectively. When sheets begin to get complicated, users can collapse any group of inputs, outputs, and functions into one of these small function boxes, enabling both easy creation of specialized libraries and easy management of screen clutter. Figure 1 is a screen shot of the Viva 2.0 interface, which calculates the dot product $(x1*y1)+(x2*y2)+(x3*y3)+(x4*y4)$. The "Go" input begins this vector computation.

1.2 Inherent Parallelism:

The four multiplications in Figure 1 are automatically concurrent, as are the first two additions, due to Viva's inherent parallelism. If the FPGA performed all the operations one by one, program execution would require about 103 clock cycles; by contrast, the parallel Viva solution requires only about 27.

Dot products and any other operations that exhibit a high degree of natural parallelism can reap great benefits from this parallel nature of FPGAs and Viva. Parallelism on FPGAs is limited only by two factors. The first is the number of operators that can fit on the available silicon. The other is the amount of parallelism in the process being programmed. In nearly all algorithms, some processes cannot operate simultaneously, due to one process's dependence on another process's results. These processes require separate sequential steps no matter how much silicon is available. Necessary sequentialism, however, is more prevalent in some algorithms than others.

Therefore, the best candidates for programming on FPGAs are those with minimal necessary sequentialism. For very parallel algorithms, most of the silicon may operate in parallel nearly all the time. Jacobi iteration, a solution method for systems of matrix equations, is one such algorithm.

1.3 Jacobi Iteration:

Jacobi Iteration (Reference [1]) is a mathematically simple iterative method to solve systems of simultaneous linear equations. Let us begin with the matrix equation

$$[\mathbf{A}]\{\mathbf{x}\} = \{\mathbf{b}\} \quad (1)$$

that represents the system of linear equations

$$\begin{aligned} \mathbf{A}_{11}\mathbf{x}_1 + \mathbf{A}_{12}\mathbf{x}_2 + \mathbf{A}_{13}\mathbf{x}_3 + \dots + \mathbf{A}_{1n}\mathbf{x}_n &= \mathbf{b}_1 \\ \mathbf{A}_{21}\mathbf{x}_1 + \mathbf{A}_{22}\mathbf{x}_2 + \mathbf{A}_{23}\mathbf{x}_3 + \dots + \mathbf{A}_{2n}\mathbf{x}_n &= \mathbf{b}_2 \\ \mathbf{A}_{31}\mathbf{x}_1 + \mathbf{A}_{32}\mathbf{x}_2 + \mathbf{A}_{33}\mathbf{x}_3 + \dots + \mathbf{A}_{3n}\mathbf{x}_n &= \mathbf{b}_3 \\ \dots & \\ \mathbf{A}_{n1}\mathbf{x}_1 + \mathbf{A}_{n2}\mathbf{x}_2 + \mathbf{A}_{n3}\mathbf{x}_3 + \dots + \mathbf{A}_{nn}\mathbf{x}_n &= \mathbf{b}_n \end{aligned} \quad (2)$$

where the \mathbf{A} 's with subscripts are matrix coefficients, \mathbf{b} 's are the right-hand-side values, \mathbf{x} 's are the solution values, and n is the number of equations in the problem. First, each equation is solved for a different variable:

$$\begin{aligned} \mathbf{x}_1 &= (\mathbf{b}_1 - \mathbf{A}_{12}\mathbf{x}_2 - \mathbf{A}_{13}\mathbf{x}_3 - \dots - \mathbf{A}_{1n}\mathbf{x}_n) / \mathbf{A}_{11} \\ \mathbf{x}_2 &= (\mathbf{b}_2 - \mathbf{A}_{21}\mathbf{x}_1 - \mathbf{A}_{23}\mathbf{x}_3 - \dots - \mathbf{A}_{2n}\mathbf{x}_n) / \mathbf{A}_{22} \\ \mathbf{x}_3 &= (\mathbf{b}_3 - \mathbf{A}_{31}\mathbf{x}_1 - \mathbf{A}_{32}\mathbf{x}_2 - \dots - \mathbf{A}_{3n}\mathbf{x}_n) / \mathbf{A}_{33} \\ \dots & \\ \mathbf{x}_n &= (\mathbf{b}_n - \mathbf{A}_{n1}\mathbf{x}_1 - \mathbf{A}_{n2}\mathbf{x}_2 - \dots - \mathbf{A}_{n(n-1)}\mathbf{x}_{n-1}) / \mathbf{A}_{nn} \end{aligned} \quad (3)$$

Next, an initial guess is chosen for the solution vector, for example $\{\mathbf{x}\}^{(0)} = \{\mathbf{0}\}$. A second guess, $\{\mathbf{x}\}^{(1)}$, is generated by substituting the values of $\{\mathbf{x}\}^{(0)}$ into the right side of equations (3) and evaluating $\mathbf{x}_1^{(1)}$ through $\mathbf{x}_n^{(1)}$. The new solution vector $\{\mathbf{x}\}^{(1)}$ solved for in this manner is then substituted again into the right side of equations (3) to generate a third guess, $\{\mathbf{x}\}^{(2)}$. In general:

$$\begin{aligned} \mathbf{x}_1^{(k+1)} &= (\mathbf{b}_1 - \mathbf{A}_{12}\mathbf{x}_2^{(k)} - \mathbf{A}_{13}\mathbf{x}_3^{(k)} - \dots - \mathbf{A}_{1n}\mathbf{x}_n^{(k)}) / \mathbf{A}_{11} \\ \mathbf{x}_2^{(k+1)} &= (\mathbf{b}_2 - \mathbf{A}_{21}\mathbf{x}_1^{(k)} - \mathbf{A}_{23}\mathbf{x}_3^{(k)} - \dots - \mathbf{A}_{2n}\mathbf{x}_n^{(k)}) / \mathbf{A}_{22} \\ \mathbf{x}_3^{(k+1)} &= (\mathbf{b}_3 - \mathbf{A}_{31}\mathbf{x}_1^{(k)} - \mathbf{A}_{32}\mathbf{x}_2^{(k)} - \dots - \mathbf{A}_{3n}\mathbf{x}_n^{(k)}) / \mathbf{A}_{33} \\ \dots & \\ \mathbf{x}_n^{(k+1)} &= (\mathbf{b}_n - \mathbf{A}_{n1}\mathbf{x}_1^{(k)} - \mathbf{A}_{n2}\mathbf{x}_2^{(k)} - \dots - \mathbf{A}_{n(n-1)}\mathbf{x}_{n-1}^{(k)}) / \mathbf{A}_{nn} \end{aligned} \quad (4)$$

Under the right conditions, applying (4) repetitively yields a sequence of solutions that converges to the correct solution vector (Reference [1]).

This algorithm is very well suited to the unique advantages offered by FPGAs as it exhibits a great deal of inherent parallelism. Each iteration requires n^2 multiplications that could ideally all be executed simultaneously. Practically, there are limits to how

many operations can physically fit on an FPGA chip, which prevent such a staggering degree of parallelism from being realized. However, multiplications can take full advantage of all the parallelism the FPGAs have to offer. There are also n^2-n subtractions per iteration (not quite as inherently parallel as the multiplications) that like the multiplications are sufficiently parallel to fully exploit all the FPGA has to offer.

The Jacobi algorithm was also selected for its mathematical simplicity. Jacobi iteration has only one mathematical stage, which it repeats forever, distinguishing it from more complex direct matrix solvers, which typically have several stages, each requiring different mathematical operations (Reference [2]). The iterative method can also be programmed in a manner that minimizes the communication between different operations. In fact, we can treat the set of operations to evaluate the right side of each equation as a separate entity that only requires communication with the solution vector on two occasions. The equation sends a signal that says “I computed \mathbf{x}_i and this is its value,” and receives a signal that says “Begin the next iteration with the values $\{\mathbf{x}\}^{(k)}$.” This implementation of the Jacobi solver also fully exploits a matrix’s sparseness by only storing and operating on the nonzero elements of the matrix, as will be explained in the next section. All these reasons, taken together, make Jacobi iteration an excellent candidate for programming on an FPGA.

2. Solver Implementation:

The goal of this work was to create an iterative matrix equation solver to exploit the inherent parallelism of the FPGA hardware to as high a degree as possible. This section describes the programs that were written toward that end.

2.1 Vector Representation in RAM:

The matrix $[\mathbf{A}]$ is not stored all together as one entity, but rather as n independent vectors, each vector storing data for one row and referenced only by the process that created it. All of the computations this solver performs are vector operations on rows of the matrix and approximations to the solution vector. Though it might appear convenient to pass vectors around between functions the same way bits and numbers are passed, such an approach is not feasible for vectors that contain thousands or millions of numbers. Instead, a vector representation scheme that treats vectors as memory objects rather than data flowing through a pipe is necessary. For this reason, vectors in this Jacobi iterative solver are stored in *RAM*, and treated as objects into and out of which data must be passed one

number at a time. It takes one FPGA clock cycle to store or retrieve data in RAM.

There are two types of vectors used in this project, “dense” and “sparse.” The first type, dense, is a representation designed for vectors whose terms are all or nearly all nonzero. Approximations to the solution vector (hereafter referred to simply as solution vectors) are represented as dense vectors. A dense vector is stored as an array of numbers indexed by their address in the RAM

By contrast, rows of a matrix may contain only a small percentage of nonzero elements and therefore are represented and stored as sparse vectors. A sparse vector is an ordered array containing each nonzero element that occurs in the vector, paired with that element’s location in the array. By storing sparse vectors in this way, the solver can ignore any zero element of a matrix, allowing full exploitation of the matrix’s sparseness.

The number of nonzero elements (called the “size” of the vector) is also stored in a register associated with one sparse or dense vector’s RAM.

Figure 2 shows a dense vector and a sparse vector, programmed in graphical Viva code. The parameters passed into the dense RAM object are “Data,” “i” (index), “Go” (initiates a read or a write), “Read” and “Write” (true/false bits which tell the RAM how to interpret a “Go” signal), “Size” (the input to the size-storing register), and “Resize” (updates the size-storing register). The parameters passed out are “Data,” “Size,” and “Done” (indicates a read or a write is complete). “ClkG” is the global clock.

The sparse RAM object has mostly nearly identical inputs and outputs, except that the location of data in the mathematical vector is not necessarily the same as its location in the memory. “i” still represents the mathematical location of data, but now “Addr” represents the memory location where data is stored.

In addition, vectors’ input and output parameters, for the sake of easier programming and less cluttered sheets, are “packed” into one pipe and passed together. This feature does not actually alter the functionality of any program or function and is not necessary – it is merely a mechanism for packaging seven or eight distinct pieces of information into one graphical pipe rather than in seven or eight pipes.

Because the RAM vector representation is not like other data in Viva, which can be passed intact through pipes, into and out of boxes, functions that operate on vectors must be implemented differently than those that operate on more conventional data. In many cases, vector functions require for loops and other control structures to cycle through the vector. In addition, these functions may not only require input from the vectors on which they operate, but also provide output parameters back to those vectors, in order to tell the

vectors which data to send and when. It is easiest to place the vector object inside a function that uses it, but often it is necessary to keep vectors outside of functions, especially if more than one function uses them.

Typically more than one function operates on a vector during the execution of a program (e.g. one function loads the vector, and another dots it with another vector). Both functions must be able to send the vector a set of input parameters, but the vector must only respond to the correct set of parameters for the specific stage of execution that is underway. The best way to achieve this goal is to pass both sets of parameters into a Viva Multiplexer object (abbreviated “Mux” in Viva). The Mux also requires a “state input” that signals to the Mux which stage of execution is currently underway, and therefore which set of parameters applies and should be passed on to the vector. This state input can be a bit, double bit, triple bit, or any other data type, depending on how many different stages it must distinguish among.

Combining a Mux and a vector into one object was used to form five-input and six-input vectors, with two and three different sets of parameters to choose from, respectively. This maintains the same functionality but saves space on complex sheets. Figure 3 contains one such five-input vector. The “Loader_S” object sends one set of parameters to the sparse vector, to the input point on Sparse labeled “S_I_0” (sparse input zero). The “Reader_S” object sends another set to the “S_I_1” (sparse input one) input point. “Load/Read,” the input which leads into the “S” input point, contains the state bit, selecting which set of parameters to use. When “Load/Read” is 0, the vector uses the “S_I_0” or loading parameters, and when “Load/Read” is 1, the vector uses the “S_I_1” or reading parameters.

2.2 Dot Product Operation:

The only mathematical operation required during the iterative stage of execution is a modified dot product between a sparse matrix row and a dense solution vector. Equations (3) look very much like dot products, but are not exactly dot products. Consider the similarities and differences between

$$\mathbf{x}_i^{(k+1)} = (\mathbf{b}_i - \mathbf{A}_{i1}\mathbf{x}_1^{(k)} - \mathbf{A}_{i2}\mathbf{x}_2^{(k)} - \dots - \mathbf{A}_{i(i-1)}\mathbf{x}_{(i-1)}^{(k)} - \mathbf{A}_{i(i+1)}\mathbf{x}_{(i+1)}^{(k)} - \dots - \mathbf{A}_{in}\mathbf{x}_n^{(k)}) / \mathbf{A}_{ii} \quad (5)$$

and the simpler dot product,

$$\mathbf{x}_i^{(k+1)} = \langle \mathbf{A}_i, \mathbf{x}_i^{(k)} \rangle \quad (6)$$

where \mathbf{A}_i is the i^{th} row of \mathbf{A} .

The first difference is that all the elements of the row in (5) must be negated and then divided by the diagonal element of the row, \mathbf{A}_{ii} . Second, the diagonal element term $\mathbf{A}_{ii}\mathbf{x}_i^{(k)}$ is included in the dot product (6) but not in (5). Third, the right hand side \mathbf{b}_i , divided by \mathbf{A}_{ii} , must be added into the result in (5). The dot product operation implemented in the iterative solver must be modified to account for these differences in procedure. Though the operation in (5) is not exactly a dot product, it will hereafter sometimes be referred to as a dot product or modified dot product as there is no other term that could describe it.

To divide all the elements of \mathbf{A}_i by \mathbf{A}_{ii} and negate them during each iteration is quite wasteful. To divide and negate at the very beginning, before the first iteration, saves time and resources. The data generated by this process will be much more efficient to use than the raw matrix data.

To complete the data stored for the equation, $\mathbf{b}_i/\mathbf{A}_{ii}$ is stored in place of the diagonal element, indexed as the diagonal element would be. The “dot product” function is programmed to recognize when the index of the sparse vector has reached i . When it has, the function skips the multiplication step and simply adds the data from the sparse vector into the accumulated dot product.

2.3 Intuitive Description of Iterative Solver:

The matrix equation solver consists of one communication and control “hub” and numerous dot product “factories” that perform mathematical computations and communicate only with the hub.

The first of the hub’s two functions is communication. The hub stores the current approximation to the solution vector, receives updates on this approximation from each dot product factory during iterations, and broadcasts copies of itself to every factory between iterations. The second function is synchronization. The hub keeps track of which equations have reported solution vector updates during the current iteration. When all equations have reported, and the “Dot Product” stage of execution is complete for one iteration, the hub sends the factories a signal to proceed to the “Copy” stage of execution. When copying is finished, the hub sends another signal to the factories, initiating another “Dot Product” stage.

The factories perform the actual mathematical operations that solve the system of equations. Each factory contains sparse row data vectors, copies of the dense solution vector, adds, and multiplies. During the “Dot Product” stage, the factory computes the modified dot products on all of this data with a user-defined degree of parallelism, and reports the results back to the hub until it is finished with all its rows. Each factory

then waits for the other factories to finish their own computations. During the “Copy” stage, each factory receives the new solution vector, one value at a time. Signals from the hub tell each factory when to begin and end each stage.

2.4 Parallelization of Solver:

To use FPGA computers to their full potential, the programmer must endeavor to take the best possible advantage of the inherently parallel nature of the FPGA. In the case of this iterative solver, both the algorithm and the FPGA have a great deal of potential parallelism, and exploiting this parallelism to the maximum is essential to achieving high solution speed.

Since the number of CLBs on the FPGA (and therefore the number of specialized processors that can be created) limit parallelism, the best results are achieved when most of the processors can operate simultaneously. For large problems, complete parallelization is impossible, but a good combination of sequentialism and parallelism can use most of the FPGA’s resources most of the time. Minimizing chip space and operation time governs many decisions regarding how to structure Viva programs.

Often, it is advantageous to let several different processes share the same adder or multiplier. Unfortunately, such optimization can complicate matters, requiring complex timing and routing structures to tell the adder or multiplier where to obtain inputs and where to send outputs at any given time. These control structures are explained in more detail in section 2.5.

The first level of parallelism applied to the matrix equation solver program was obtained by separating different rows’ operations from one another. Computations for row 4 have nothing to do with computations for row 1, so the two sets of computations are inherently parallel. If we had enough FPGA space, we could simply create a factory for each row, but given current technology it is only feasible to have a limited number of factories, and assign each factory to perform the computations for numerous rows, one row at a time. This was the only level of parallelism programmed into Figure 4, the 3x3 solver that was tested successfully. Each factory includes one multiplier and one adder to utilize for dot product computation. Further analysis, however, shows that dramatically increased parallelism is possible by adding a second level of parallelism.

Some Viva objects are much larger than others; On the Xilinx 4062, the FPGA used for this solver, 32-bit-floating-point adders are quite large (14% of the CLBs on one of the FPGAs), 32-bit-floating-point multipliers are smaller (7%), and most other operations necessary to the solver are very small (<1%). The CLBs necessary to access different RAM addresses can also

grow to a significant size, given larger-sized or denser matrices. (Given the small nature of the matrices tested on this solver, RAM did not take up a significant amount of FPGA space. However, when the solver is tested on a larger scale, RAM accessing may become a consideration).

There also are differences in the amount of time a given operation on an FPGA takes to complete. Many operations, such as bit inversion, logical gates, and Muxes are asynchronous (effectively “instantaneous;” they require much less than one clock cycle), and some control or memory processes such as accessing or storing RAM are synchronous, with one nominal clock cycle. An addition requires only one cycle, while a multiplication requires around 25.

Programming strategies must take these differences into account. Each iteration requires about as many multiplications as it does additions, but since a multiplication takes much longer than an addition, and adders take up more space on the chip than multipliers, it makes sense to create more multipliers than adders in execution of the algorithm. One adder can effectively service almost 25 times as many dot product processes simultaneously as can one multiplier. For this reason, the number of multipliers is the determining factor for the actual degree of parallelization in the solver, and it is advantageous to create extra multipliers that share fewer adders. The following parallelization scheme for factories, though not completely implemented and tested in a working matrix equation solver, was programmed. This scheme, when implemented, will greatly increase solution speed by increasing the number of multipliers that operate concurrently.

2.4.1 NRows and NRows1*:

An object, called “NRows,” stores N sparse vectors of row data, and what may be thought of as a “pointer.” These vectors are accessed one at a time. Inputs into NRows are only received by the row that the pointer points to, and outputs from NRows also come from this row. When the “First_Row” input initializes NRows, the pointer points to the first row. When the program finishes with each row vector, another input, “Next_Row” advances the pointer. When the pointer reaches the end of the list, it ceases pointing to any of the rows.

Another object, called “NRows1*,” consists of an NRows object, one multiply object (the asterisk represents multiplication), and a few other functions, including a copy of the solution vector. This object evaluates the (modified) dot product for one row at a time. NRows1* contains both the control structures for the dot product and the multiplier necessary to compute it. The control structure selects each nonzero element in the row data one by one, matches it with the appropriate coefficient in the solution vector, and sends

it to the multiplier to be multiplied. The result of the multiplication is then sent to the adder along with the current accumulated dot product. Most of these components can be seen in Figure 5, the graphical code for `NRows1*`. The rest are included in the `Rowdot` function, expanded and displayed Figure 6.

The adder, however, is *not* included in an `NRows1*` object. This is because one adder is shared among several `NRows1*` objects. Therefore, the adder must remain an outside resource for `NRows1*` to access only when required, and when another `NRows1*` object is not using it simultaneously.

2.4.2 M(NRows1*) and M(NRows1*)1+:

The `M(NRows1*)` object contains a group of `M NRows1*` objects, which work simultaneously, sharing an adder. The `M(NRows1*)1+` object consists of an `M(NRows1*)` object and one “TDM_Add” object. `TDM_Add` is an addition object that both performs addition and mediates between the multipliers requesting the adder’s resources. `TDM_Add` is explained more completely in section 2.5.

2.4.3 Recursion for Flexibility:

One important feature of an FPGA program is the user’s freedom to choose the parameters that will affect its operation. In this solver, a number of parameters are passed out to the top level to give the user options concerning parallelism, including two in particular: `(MSB)M`, and `(MSB)N`, which define respectively the number of multipliers sharing each adder, and the number of rows sharing each multiplier. The value obtained from each of these variables is the bit length of its type, not the actual data it holds. They must be defined before compile time because they affect the actual creation of the factory part of the iterator on the chip. The way this recursion works (which is the clever standard mode of recursion in Viva developed by Samuel Brown), is that a function is created with different definitions according to the data types of its input. The end leaf of recursion is the function definition where the recursion input is a bit, and the programmer strips one bit off the data type at each non-end-leaf level of recursion. In this way, the number of bits in the recursion input is the number of recursion levels. `$M(NRows1*)` (the lower-level function inside `M(NRows1*)` that actually implements the recursion) uses this recursion method and is shown in Figure 7. `(MSB)M` is the recursion variable.

2.5 Control Structures:

Since Viva programming is inherently parallel, and functions do not necessarily progress in any particular order, the Viva programmer must make use of Viva control structures in order to synchronize operation. In other words, all operations automatically execute in

parallel; *any* sequential operation must be explicitly programmed and defined precisely. Since most algorithms require a combination of sequential and parallel operations, and since the best Viva programs reuse many of their operators, control structures often are more complex than a simple one-by-one progression from function to function. These control structures afford users great power in defining how programs work, and thus can achieve large speedups from parallelism. However, they can become complex and more difficult than any other parts of programs.

The most basic control structure available in Viva is *Go-Done* circuitry. Synchronous objects read in their input and begin operation as soon as they receive a *Go* “pulse” (a bit which goes to one for a single clock cycle, then returns to zero). When finished with operation, they output a *Done* pulse to indicate the fact.

For loops are another easy-to-use control structure built into the Viva language, and are similar to *for* loops in text based languages in their purposes, though their functionality is adapted to the different type of programming; they are objects, with input pulses and output pulses. Inputs include *Next* (indicates to *For* object that an iteration is complete) and *Go* (initiates first iteration), while outputs include *i* (the looping index), *Pulse* (sets in motion the next iteration after a *Next* is received), and *Done* (indicates all iterations are complete).

Other control structures needed to be more complex. Programming of control structures turned out to be the single most challenging part of creating the iterator. It would take up too much space to explain the workings of every control machine created for the iterator, but brief outlines of what the more important structures do are necessary in an explanation of the overall program, and therefore are provided.

The previously discussed hub, a function called *Update/Copy*, is the control structure which drives the highest level of timing, the alternation between copy and dot product stages. In the copy stage, the current solution vector guess, stored in the hub, is broadcast to all the factories, along with a signal letting the factories know what stage is currently “on.” During the dot product stage, the hub signals the factories to begin computation, awaits reports from each row that a dot product has been completed, and keeps track of which rows have and have not reported. A string of bits as long as the number of rows is required to keep track of this; as a row reports, its associated bit goes from zero to one. When the bit string is full of all ones, the dot product stage is ended.

Recursive structures that are accessed one by one also require recursive control structures that maintain a “pointer” to whichever level of recursion is currently being accessed. This functionality usually consists mostly of a register that stores one bit on each level of

recursion, signaling if each level is “on” or “off.” Beginning with the bottom level, as each level of recursion is used, that level is turned off. In the meantime, signals are passed down through the “on” levels until they come to a level under which is an “off” level. In this way, only the bottommost “on” level receives and sends data, while the others are bypassed. In Figure 7, the variable “Pass2Me” is the level’s on/off switch.

Yet another important control necessity arises from shared adders and a shared hub. Though multipliers are shared, it is not difficult to control their use, as only one row will ever use that multiplier at any one time. Adders, on the other hand, may receive several different sets of data from several different multipliers simultaneously. Since an adder can only perform one addition at a time, control structures must be present to store the other sets of data while the adder adds one set, and to send another waiting step as soon as the adder is finished. The TDM_Add (TDM stands for Time Division Multiplexing) object performs both of these functions, containing control structures and an adder to handle simultaneous input from numerous different multipliers. The Hold object performs these same functions, except for the fact that instead of managing several multipliers sharing one adder, it manages several factories, all with inputs to one hub.

2.6 Input:

Before iteration actually starts, there are some operations that must be done to prepare the raw matrix data – see the explanation of the algorithm in section 1.3 for a description of these initial data processing operations. For the purpose of not wasting FPGA space on functions that are only performed once, at the beginning of execution, these preparatory steps are left outside of the iterative solver. When file input is implemented, a program to perform these preparatory steps should be created for the CPU, to get the data into the form it must take for the FPGA solver.

Inputs were sometimes read from the screen to test component functions and the 3x3 iterator, which did not require a large volume of input. However, for larger matrices, screen input would certainly be far too tedious and lengthy, and automatically loaded file input would be preferred. Automatic loading procedures were developed for input that was defined in the actual code, but as the solver is still at a level of small test cases, it has not yet been necessary to create file input procedures. Such procedures should be somewhat complicated to create, but not prohibitively complicated since Viva does contain viable File input/output procedures. Future extensions of this solver will undoubtedly require automated file loading procedures.

Such automatic loading procedures are complicated by the fact that the Viva objects that reference the RAM

are removed by several steps from the top level. File data input into a factory must be routed to the correct M(NRows1*)1+ object. Once passed into that M(NRows1*), it must be routed to the correct NRows1* object, where it must then be routed to the correct Row in NRows. Each of these routing steps requires a large number of registers, Muxes, Demuxes, and other cooperative control structures to keep track of where every piece of incoming data should be stored. The file input all enters the program at the same place, and rather than directly sending it to somewhere in the program to be stored, the program itself must simply go through all its rows one by one in an orderly fashion, opening and then closing each to incoming data. A different scheme for loading, storing and accessing data might be advantageous in future versions of the equation solver if it allowed for more simplicity of data loading and accessing.

With an automatically loading iterator, the user will need to make several inputs before compiling, including (MSB)M and (MSB)N. Other variables that must be defined at compile time are DType (data type), which tells Viva what type of specialized multipliers, adders, registers, and other functions to build for data processing; iType (index type), which tells Viva what type (must be an MSB type) to use to represent indices and row numbers; and NBits, the bit length of which must be at least the total number of rows in the matrix. Some constants, such as the size of the matrix, also must be defined either before or after compilation. Finally, the user must choose the number of factories, and each factory individually created and connected. This is a weakness that should be redressed in future matrix equation solvers.

An ideally user-friendly solver should only take a few inputs defining the amount of parallelism, a file name containing data, and a single signal to set the iterations in motion.

3. Results:

At this stage, the iterative solver has had successful initial results. A 3x3 application of the iterator has been tested successfully. Large versions with hundreds or thousands of equations and variables are at this point not possible, due to limitations of current hardware and code and incomplete loading procedures.

3.1 Small-Scale Success:

The aforementioned test of the solver on the following 3x3 matrix equation:

$$\begin{bmatrix} 1.0 & 0.5 & 0.25 \\ 0.5 & 1.0 & 0.125 \\ 0.25 & 0.125 & 1.0 \end{bmatrix} \{x\} = \begin{bmatrix} 2.75 \\ 2.875 \\ 3.5 \end{bmatrix}$$

was successful. This test correctly found the solution $x_1 = 1$, $x_2 = 2$, $x_3 = 3$ to a high degree of precision.

A few troublesome idiosyncrasies emerged from the results, however. After converging to within one millionth from the exact solution, if the iterator continued iterating, it would briefly stray from the correct solution, as much as a hundredth off the exact answer. Another finding was that, viewing the results after each iteration, during millions of clock cycles per second, an occasional aberration would appear that was far away from the correct answer, (e.g. 4.7×10^{37}). It is unknown what caused these errors, but round-off error is not likely to be the culprit; the numbers were stored as 32-bit float numbers, so round-off error should be very small. Small errors do not have a chance to compound in an iterative method as they do in direct methods, because though error might offset the current guess at the solution vector by a small amount, the vector automatically converges back toward the correct answer, nullifying the error. Therefore, it is more likely that something sporadically goes awry within a multiplier or adder, or that a small error in the solver's programmed control structures manifests itself infrequently.

3.2 Large-Scale Limitations:

Numerous factors have frustrated attempts to create the more parallel matrix equation solver described in section 2.4.1-2.4.3. First and foremost have been difficulties using Viva adders and multipliers for floating-point data types. For unknown reasons, some functions programmed and tested successfully on the "byte" data type failed when applied to the "float" data type. Examples of errors from adders and multipliers usually took the form of errors such as $1.0+2.0=4.7 \times 10^{37}$, which suggest misplacement of bits resulting from multipliers and adders being confused about whether they are operating on unsigned integers or floating-point numbers. In some cases, programmer error was the cause of these problems, in the form of misapplication of data type casting rules and procedures in Viva. These rules can be complex and were sometimes confusing to the programmer. For instance, data in feedback loops must be recast before being fed back into the functions, and data read in from files must be cast to its own data type before being used. Data type management is simply another example of a tool Viva programmers possess which increases their power, but can also confuse.

In other cases, however, casting rules such as the ones above seemed to have been followed but nonsensical results still came out of floating-point adders and multipliers. Such bugs limited the speed of developing functions and ultimately were the main obstacle that prevented a solver using the complex

recursion scheme outlined in this paper from ever being successfully tested.

Another limitation that affected the successfully tested but not-as-parallel solver was the number of bits that could be passed between FPGAs, currently 32 bits. The implementation of this iterative solver requires 37 bits of communication. The upshot was that only one FPGA could be used. However, this problem should be solved soon enough since the new FPGA Hypercomputer set to be obtained by NASA in September will allow 128 bits to be passed between FPGAs.

In the end, most of the necessary functions, such as $NRows$ through $M(NRows1^*)1+$, have been programmed and tested successfully for integer data. Automated file input, fewer user inputs, and a few small functions to connect things on the top level still have yet to be programmed and are planned for the future to make the solver complete.

3.3 Future Promise:

Viva and Star Bridge Hypercomputers are both advancing dramatically at the present time. Before Viva 2.0, this solver would not have been possible, due to the inaccessibility of RAM and File input/output. Viva software also became much more reliable with Viva 2.0. Future versions of Viva are expected to make even larger strides, making control structures and data types much easier to create and manage. In addition, a new, state-of-the-art hypercomputer is coming soon to NASA Langley Research Center. This new hypercomputer, which includes new Xilinx FPGAs, allows 128 bits of communication between FPGAs, more space on each FPGA, and faster floating-point multiplications. With this more powerful board and advanced version of Viva, and other new boards and software to come, it is to be expected that applications such as large-scale matrix equation solution will take better advantage of all that FPGA computing has to offer.

4. Conclusions:

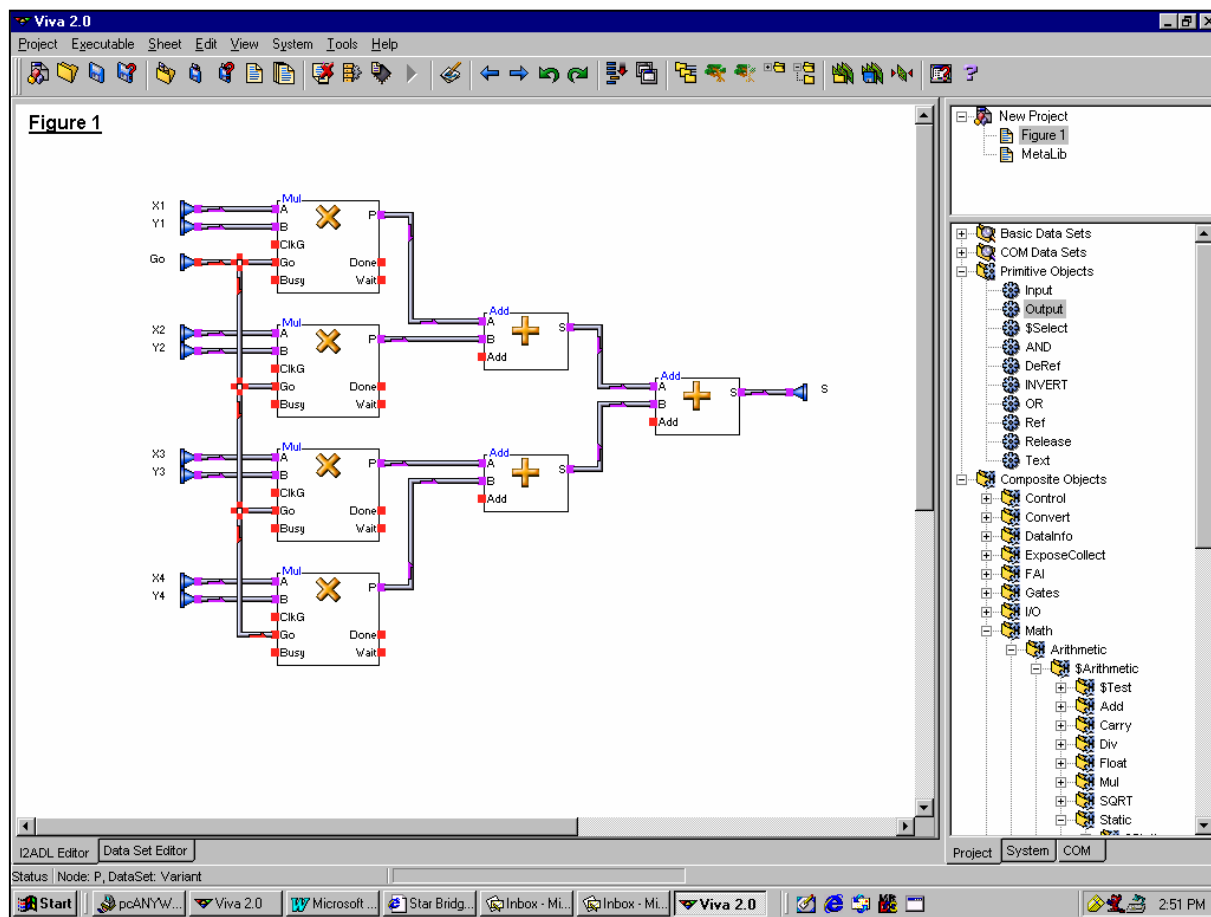
FPGA technology, paired with Viva, shows serious promise for application to matrix equation solution. Solving matrix equations was found to be both possible and feasible given current FPGA technology, and if the current rate of FPGA and Viva advancement continues, it is very likely that FPGAs will achieve parallelism unequalled in current state-of-the-art CPU-based computers. This parallelism should yield much faster solutions for iterative matrix equation solvers and other applications that can exploit parallelism.

5. Acknowledgements:

First and foremost, I would like to thank Dr. Olaf O. Storaasli, who has volunteered so much of his time in the past two years to open up wonderful opportunities for me. I also would like to thank him for helping to edit this report. I would also like to acknowledge Dr. Robert Singleterry and Samuel Brown for teaching me a great deal about the Viva language, explaining various concepts to me, providing me with ideas, and helping me to debug code. In particular, many of the recursion and time-division multiplexing ideas used in the solver are inspired by the clever ways Samuel Brown has solved other problems. Finally, I would like to thank NASA Langley Research Center and Star Bridge Systems for making this work possible.

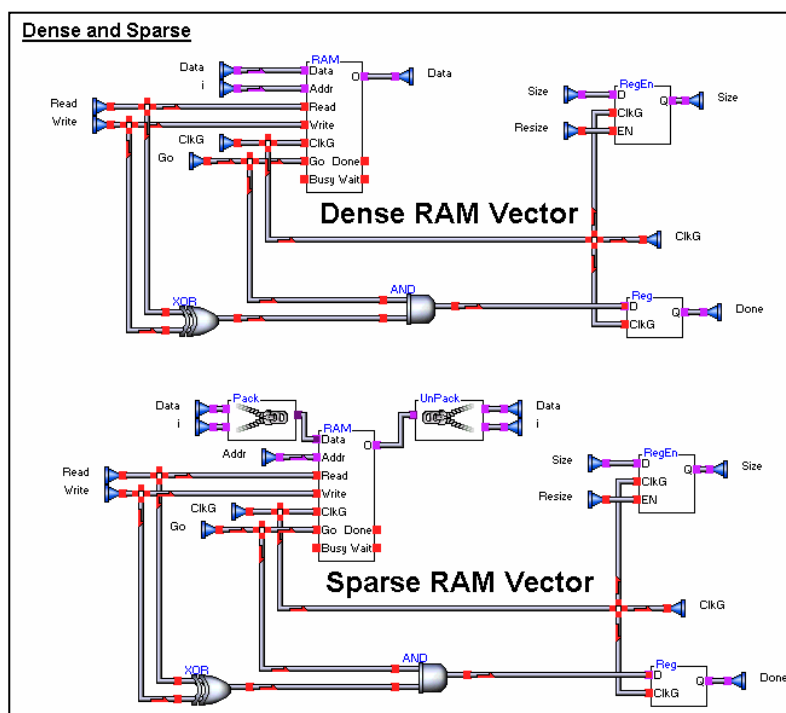
6. References:

1. Golub, Gene H. and Charles F. Van Loan. 1989. Matrix Computations, second ed. The Johns Hopkins University Press, Baltimore.
2. Singleterry, Robert C., Jaroslaw Sobieszczanski-Sobieski, and Samuel Brown. "Field-Programmable Gate Array Computer in Structural Analysis: An Initial Exploration." 43rd American Institute of Aeronautics and Astronautics (AIAA) Structures, Structural Dynamics, and Materials Conference. April 22-25, 2002.
3. Star Bridge Systems, Inc. 2002. Star Bridge Systems Web Site. <<http://www.starbridgesystems.com>>

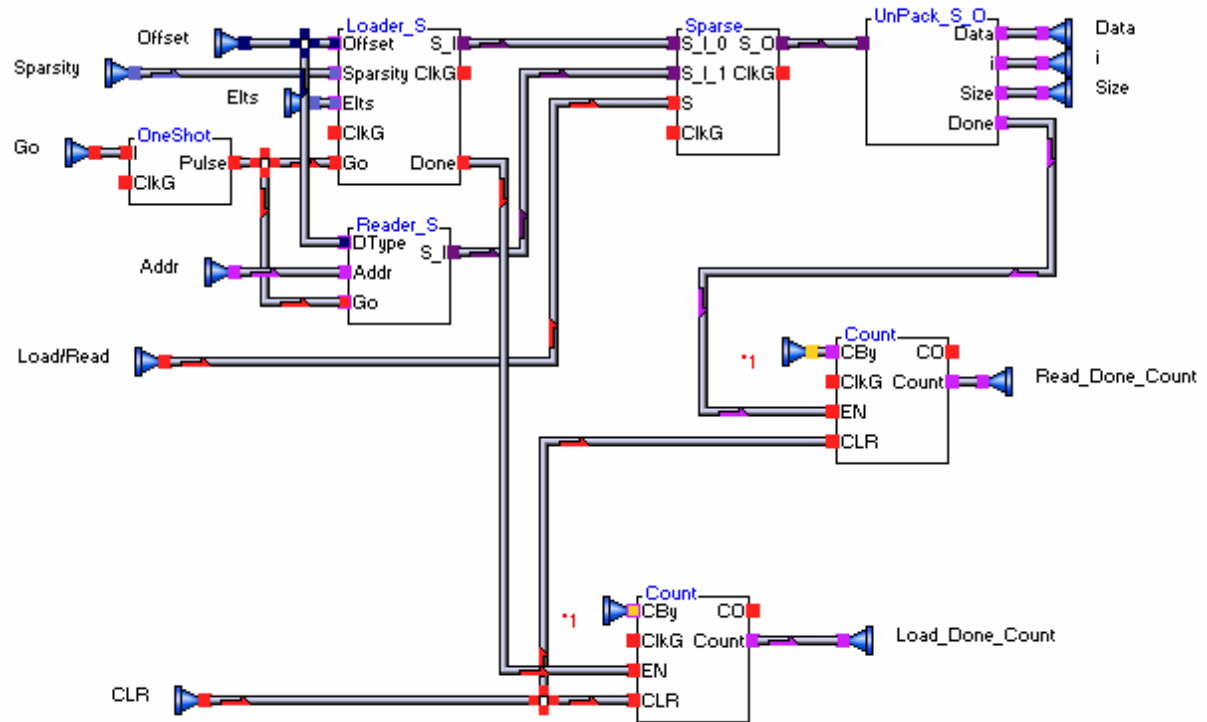


Above, Figure 1: Screen shot of Viva program to compute $(x1*y1)+(x2*y2)+(x3*y3)+(x4*y4)$.

Below, Figure 2: Code for Dense RAM Vector and Sparse RAM Vector.

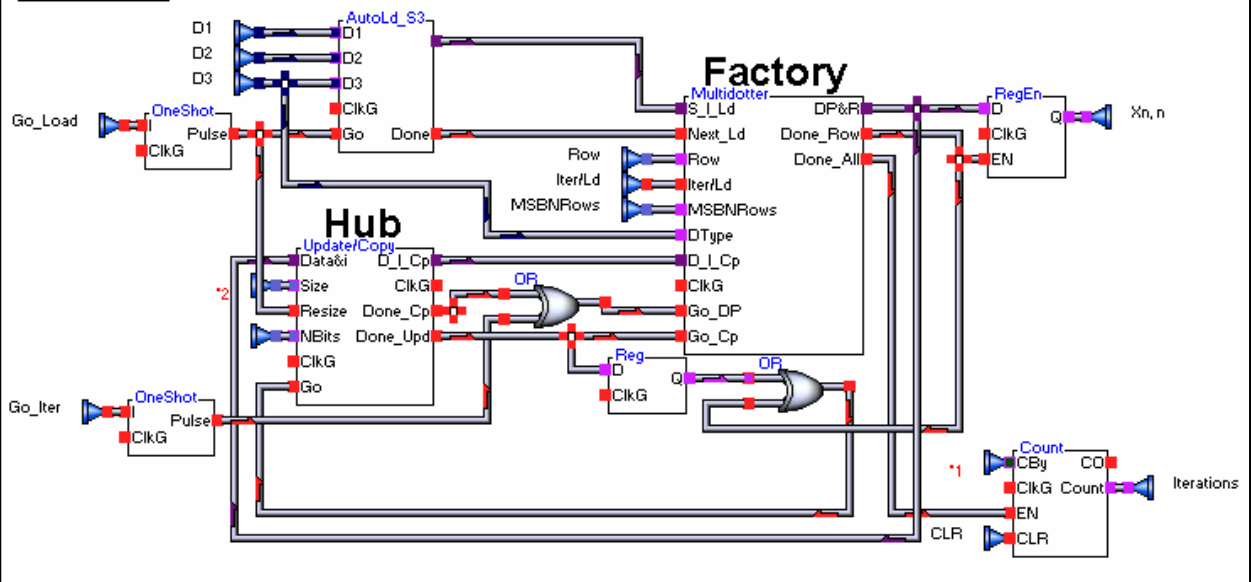


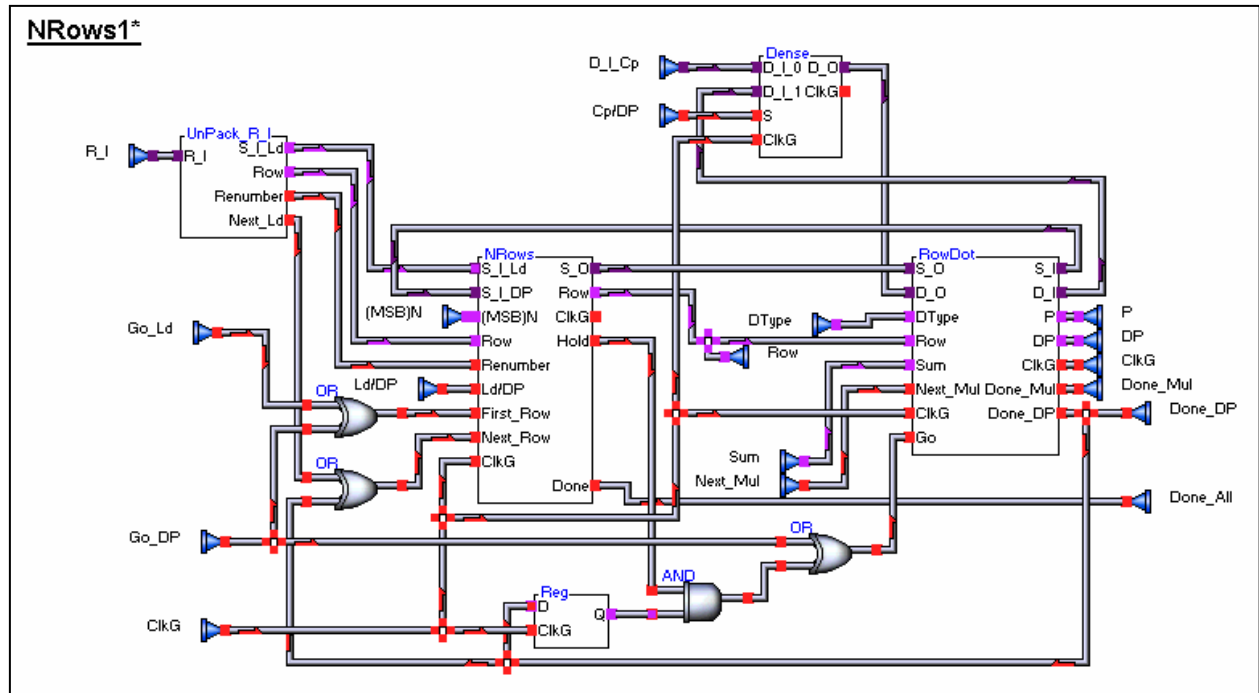
Loader S Test



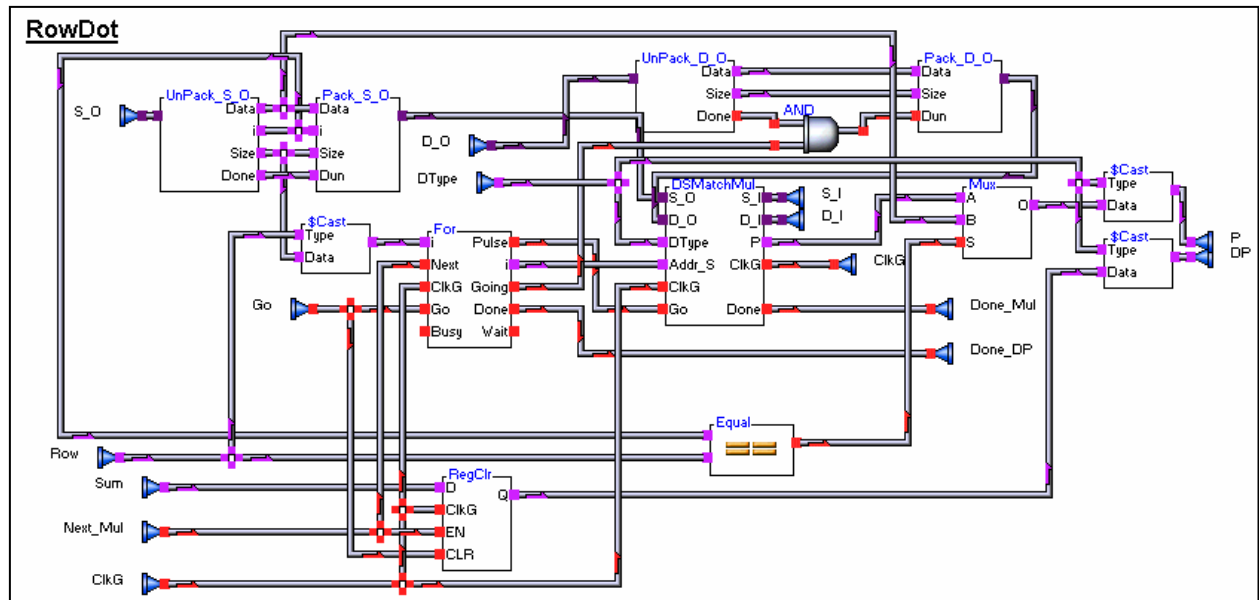
Above, Figure 3: Example of five-input Sparse
 Below, Figure 4: Complete 3x3 iterator

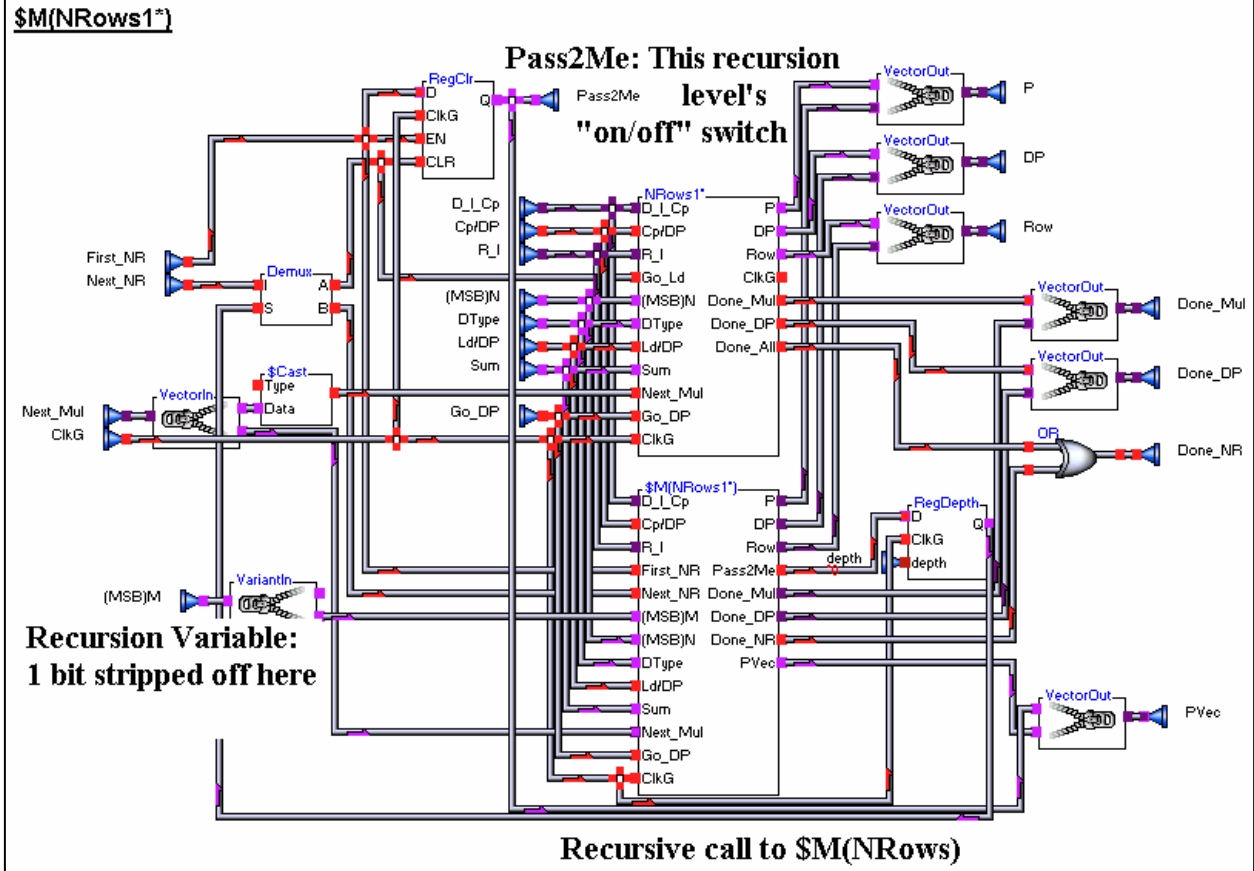
3x3 Iterator





Above, Figure 5: NRows1*
 Below, Figure 6: Rowdot





Above, Figure 7: \$M(NRows)