Design of a Massively Parallel Computer Using Bit Serial Processing Elements

Maurice F. Aburdene
Kamal S. Khouri
Jason E. Piatt
Jianqing Zheng
Department of Electrical Engineering
Bucknell University
Lewisburg, PA 17837

January 24, 1995

Abstract

A 1-bit serial processor designed for a parallel computer architecture is described. This processor is used to develop a massively parallel computational engine, with a single instruction-multiple data (SIMD) architecture. The computer is simulated and tested to verify its operation and to measure its performance for further development.

1 Introduction

Recent trends for the design of massively parallel computers have shifted from emphasis on the design of single instruction-multiple data (SIMD) machines to multiple instruction-multiple data (MIMD) machines. However, it is our contention that the trend is based on inappropriately drawn conclusions. Because of the early rush to build SIMD computers, inadequate attention was devoted to exploring the full potential of these architectures. Current MIMD machines that are based on commercially available processors lack the scalability originally intended for massively parallel computers. These computers employ processors on the order of a hundred or a thousand rather than a million, and yet consume large amounts of power and space. One of the main criticisms of these systems is the use of processors designed for workstation rather than supercomputer applications. Placing such processors in a supercomputer environment has reduced the individual performance of each processor to a fraction of their capabilities and has required the use of co-processors to aid them adapt to the multi-node environment.

The objective of our project is to design a massively parallel SIMD architecture with greater than a million processors. A low power, 1-bit serial processor specifically designed for a SIMD architecture will be utilized.

2 1-Bit Serial Processing Element

As shown in Figure 1, the processing element consists of a bit serial arithmetic and logic unit (ALU), a distributed bit serial RAM, and a 2-dimensional router. The ALU is further divided into:

- Computational logic
- Three registers or D-flip-flops - an accumulator register (A), a carry register (C), and a mask register (M)
- Transmission gates

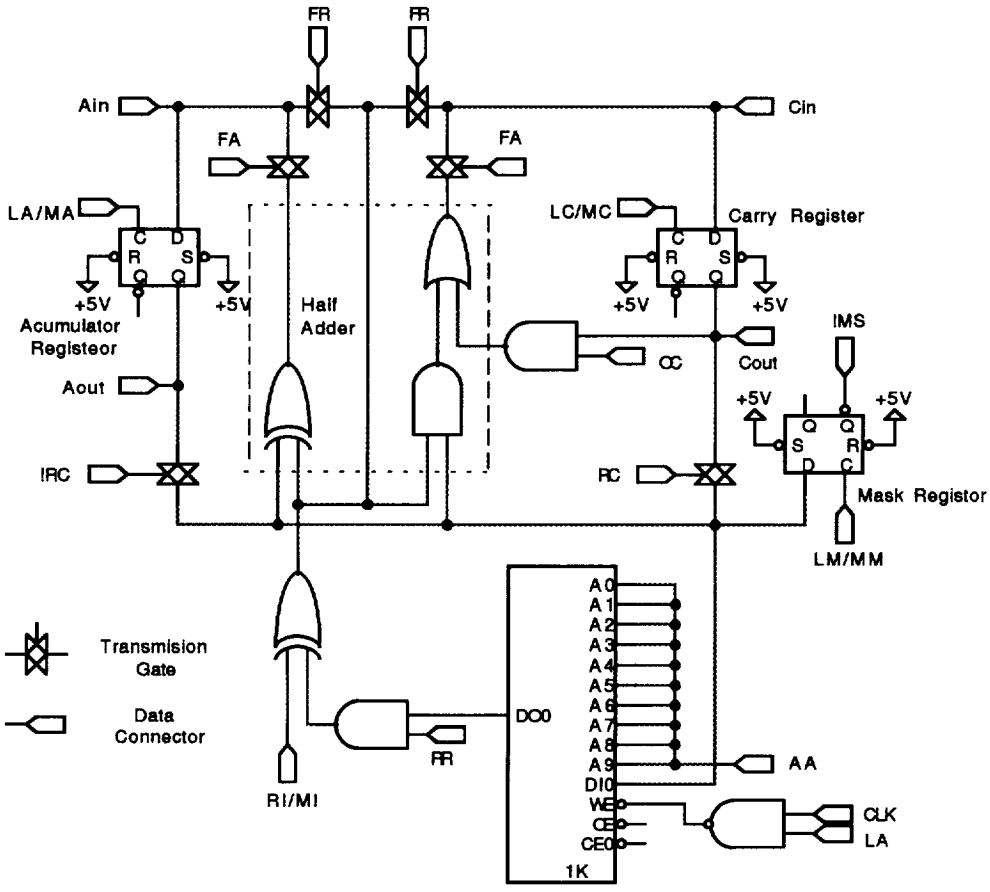

Figure 1: Registers and ALU of PE

In this design, a half adder is the fundamental component of the ALU. The results of such operations may be stored in either the accumulator or carry register. The use of a shift register may considerably reduce the execution time for functions such as multiplication. However, to reduce the size of the processing element by at least a factor of five, a shift register is not used. The mask register M enables certain processors to perform a given operation while the others are "masked" and therefore do not execute that operation. There are sixteen control signals from which functions may be constructed :

- RR - Read RAM

  - 0 - 0 is fed to PE
  - 1 - load value from memory address
- LR - Load RAM
- RI - Read RAM Inverted
- MI - Mask Invert signal
- LA - Load A Register (Accumulator)
- MA - Mask load of A
- LC - Load C Register (Carry)
- MC - Mask load of C
- LM - Load M Register (Mask)
- MM - Mask load of M
- RC - Read from Carry
  - 0 - Read from A
  - 1 - Read from C
- OC - OR with Carry

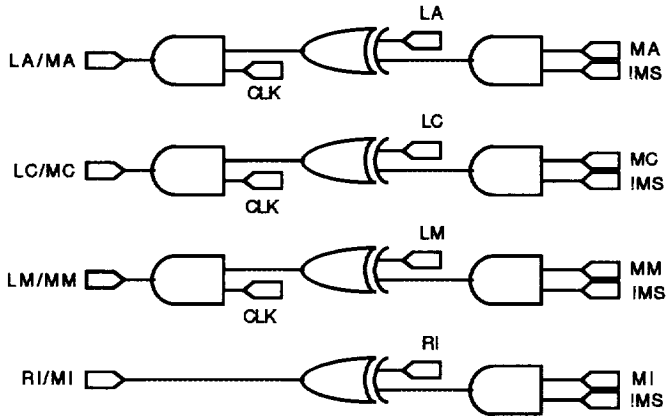

Figure 2: Logic for PE Local Control Signals

- AC - Load A/C input buses from:
  - 00 - From RAM (FR)
  - 01 - From Adder (FA)
  - 10 - Activate Router (RT)
  - 11 - Activate Global operations network (GB)

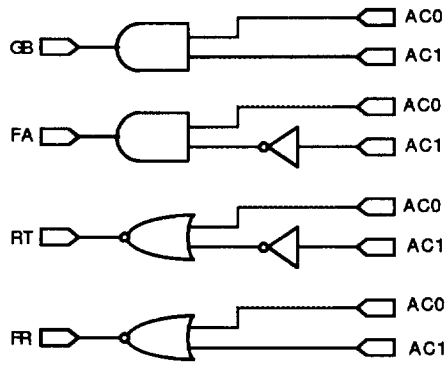

Figure 3: Logic for PE Network Level Control Signals

- DR - Router Direction
  - 0 - Send Data To North and To West
  - 1 - Send Data To South and To East
- IO - Controls external I/O via router

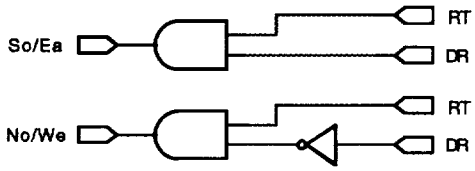

Figure 4: Logic for PE Router Control Signals

Figures 2, 3, and 4 show the logic used to implement the different control signals listed. Control signals for the PE may originate from a controller chip, a compiler or control memory. In addition to the 16 control signals, there are 10 AA bits that represent the address space of the distributed RAM. Communication with RAM is controlled by RR, RC and LR. When RR=1, data in a selected memory address appears on the data bus. RC controls where the data on the RAM write bus comes from, and when LR=1, the selected memory address is written to.

The signals LA, LC and LM cause the A,C, and M registers to load respectively. While the signals MA, MC and MM cause these same registers to be masked. Hence, if LA=0 and MA=0 then the A register will not be loaded from its input bus. If LA=1 and MA=0, then the A register will be unconditionally loaded. On the other hand, if LA=0 and MA=1 then the A register will be conditionally loaded from its input bus if and only if the corresponding M register contains the value of 0. If LA=1 and MA=1

then the A register will only be loaded if the corresponding M register contains the value 1. This set of conditions also hold true for the signal pairs LC/MC, LM/MM and RI/MI. Unlike the other signals, RI/MI do not control the loading of a register. Instead, they control the inversion of values coming from RAM. If RI=1, then the output of RAM is inverted, and MI causes a conditional inversion depending on the value of M.

Designed for CMOS implementation, the processor uses transmission gates to guarantee that there are no conflicting signals, and that no combination of control signals will produce a short from the power supply to the ground of any device.

There are three levels from which the processing element may receive and transmit data. A model is of this configuration is shown in Figure 5. These levels are:

1. Local RAM and ALU
2. Router Network
3. Global Operation Network.

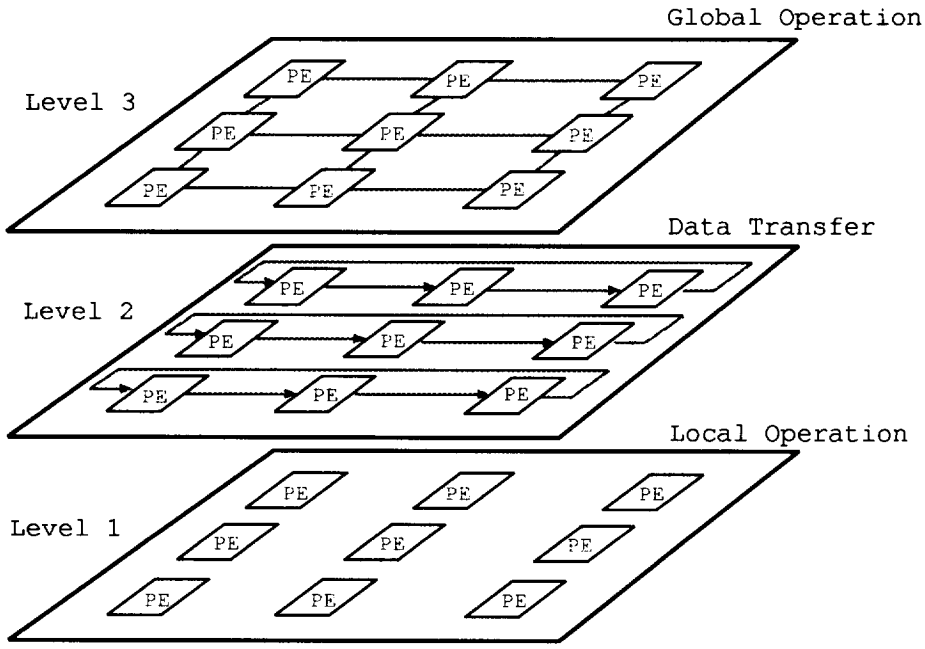

Figure 5: Three Levels of PE Network Operation

Each processing element may only communicate with one of levels 2 and 3 at a time. The first level is used to perform operations on data stored in the bit serial RAM or any one of the registers. The second level is for inter-processor data transfer and external I/O, and the third level is for global functions and operations. The signals AC0 and AC1 control which level data is loaded from and sent to. If AC1=0, then data transfer is local (ALU or RAM). When AC1=1, data is transferred from levels 2 and 3. When AC0=0 the data is sent and received from the router, and when AC0=1 the data is sent through the global operations network.

3 Communication Network and Router

The router network allows processors to transmit and receive information from each other. Any data received may be stored locally and operated on. As mentioned above, the router is a two-dimensional mesh shown in Figure 6. This means that each element may communicate directly with any of its four closest neighbors - north, south, east or west.

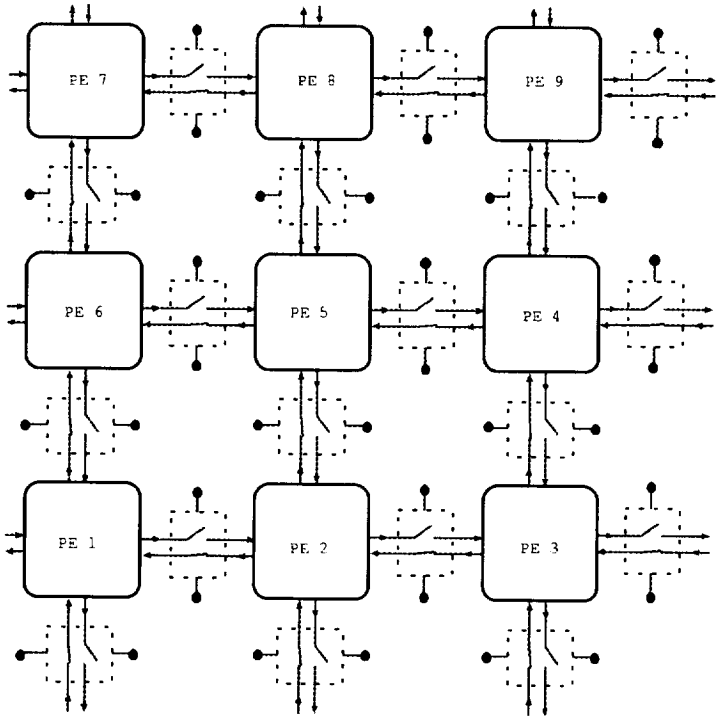

Figure 6: Block Representation of a 3x3 Toroidal 2-D Mesh of Bit Serial PE's

Since there are two registers available to store data, we may use both for communication purposes. The accumulator register is chosen to accommodate east-west data transfer, while the carry register is used for north-south data transfer. This scheme optimizes the use of a two-dimensional router, and if need be, can allow for two sets of data to be transmitted at a time. For example one command may be "send carry register contents north" or it may be "send carry register contents north and accumulator register contents west."

Each processing element is connected to two routing switches - one switch for north-south communication and another for east-west communication. As shown in Figure 7, each switch consists of four transmission gates in a square formation, where opposing gates are controlled by a common signal - So/Ea and No/We. The corners of each switch are connected to four data transmission lines as shown in Figure 8 - input of a register, output of a register, and the switches from the two adjacent processors. The control signals dictate which transmission gates are on and hence which direction the data is transmitted. When signal RT=1, then the router is activated. The DR signal controls the direction of data transfer as seen in Figure 4. When DR=0, values in the C register move south, and values in the A register move east. When DR=1, values

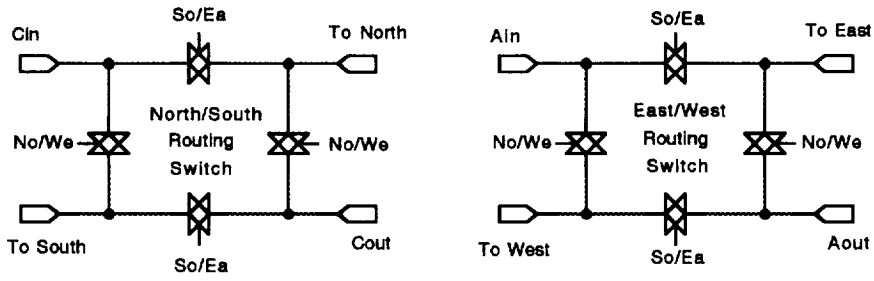

Figure 7: E-W Switch and N-S Switch for Router

in the C register move north, and values in the A register move west. Figure 8 is an example of data travelling north and west.

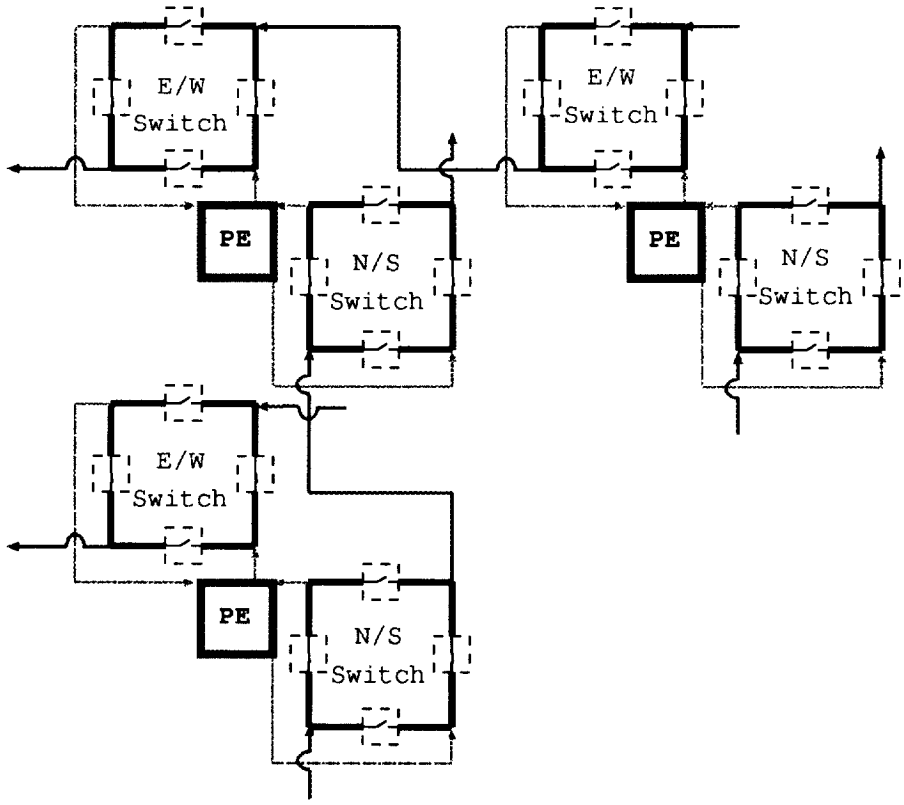

Figure 8: Three PE's Connected with a 2-D Router Switches

4 Global Operations

Often it is necessary to inquire about the status of all the processors. These types of operations are described as global operations, and require the use of an independent network. This is the third level network for data transmission that consists of a linear mesh of OR gates. Figure 9 shows one row of OR gates for N processors. The areas surrounded by a dashed square, are blocks repeated for each processor in the row. A

similar set up is used for columns of processors, and connect the C register and the north/south routing switches. The register contents of each PE are ORed and the result of the operation placed in each processor.

It may be seen in Figure 9, that on the edges of the mesh, a series of external data lines have been added to allow for external I/O. I/O may be accessed from either the east or south ends of the mesh. When the external I/O is activated, the router network remains on, but any toroidal connect must be removed to allow data to enter from only one side of the mesh.

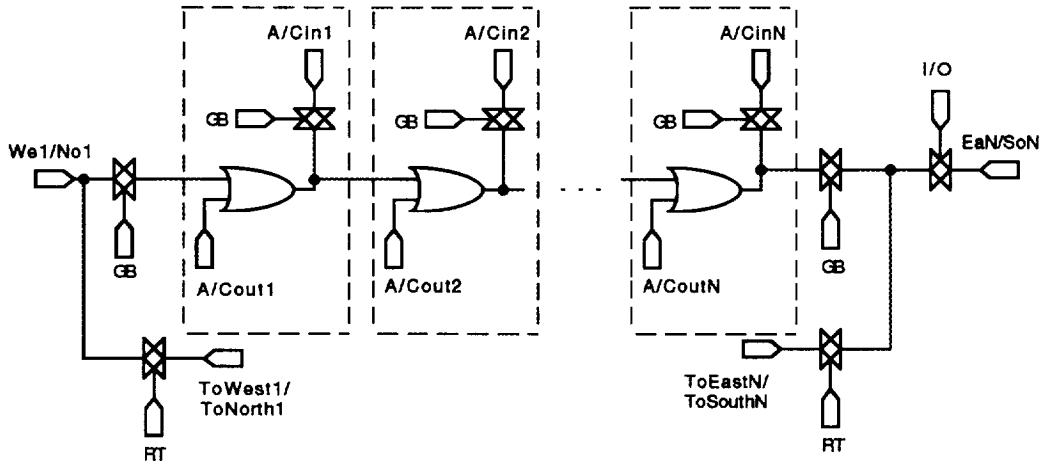

Figure 9: Global OR Function Network

The global AND function may also be implemented using the global OR network. This is done by inverting the inputs to the OR gates, and inverting the output. The inversion process may be done locally by each processor. This eliminates the need for a separate network.

5 Suggested Operating Parameters

Parameters such as the number of floating point operations per second (FLOPS), instructions per second and execution speed are difficult to describe without using a working prototype. The following is an approximation to these parameters assuming we have 1 million processors in a 1000x1000 2-dimensional mesh and a clock running at 100MHz. It is also assumed that the delay associated with a command reaching a processor from the control unit is the same for all processors.

The commands multiply and processor-to-processor crossover exchange are the two longest commands to execute, each requiring 11 clock cycles for one bit. In other words, one bit requires a total of 0.11μ seconds to be processed. For a million processors, this produces 9.09 TeraFLOPS. To achieve PetaFLOPS operation, at least 110 million processors would be required.

The band-width of such a mesh would be 1000 bits in any one direction. For a bit to travel to its closest neighbor would require 3 clock cycles to complete. The worst case condition in a torus connected mesh, would require a data bit to travel 1000 steps

(500 for each dimension). Hence it would take 30μ seconds to transfer data, producing a slowest data transfer rate of 33 million bits per second.

6 Progress to Date

The group has successfully simulated a 3x3 toroidal mesh of processing elements using circuit design software. The simulation included all local operations. In addition, the router and global networks have been designed, and we are currently in the process of simulating them. Plans are to:

- Simulate a larger network to measure propagation delay for a millions processors - this requires the use of a more powerful software package and/or computer.
- Begin to develop a VLSI prototype - test processor performance and reliability under real conditions.
- Replace the global OR with a tree structure - speed global functions
- Add more global functions such as XOR, ADD
- Develop a fault detection and avoidance system - detect faulty processors and bypass them on the network
- Develop a multiple controller SIMD structure - this may be a solution to propagation delay.

Appendix

A Primitive Function Descriptions

The following section contains functions and the lines of code needed to describe their execution. Each line of code consists of a list of signals, a memory address location (AA), and a comment line (comment lines start with "//") or the loop statements "repeat" and "while (i)." It is assumed that "repeat" does not take any time, and "while (i)" is an end of loop comparison combined with the last instruction of the loop. All other lines of code take one clock cycle to execute. If a signal is represented in the line of code, then it takes on a value of 1, otherwise it is 0 for that instruction. Names written in lower case, such as op1, op2 and sz, are counter and register values generated by the array control unit, when an instruction is being executed by the array. The notation AA(op1++) means address "op1" will be used will be used in the current instruction to access array memory address AA, while it is being incremented in the control unit.

```
Addition:
        LC       // clear C
        repeat
             RR ACO RC LC LA       AA(op1++) in-- // 1st half add
             RR ACO OC LC LA       AA(op2++)      // 2nd half add
             LR                    AA(op3++)
        while (in)

Subtraction:
        RI LC   // set C
        repeat
             RR ACO RC LC LA       AA(op1++) in-- // 1st half add
             RR ACO OC LC LA RI    AA(op2++)      // 2nd half sub
             LR                    AA(op3++)
        while (in)
```

Multiplication:

```
        // op1 -- LSB of multiplier
        // op2 -- LSB of multiplicand
        // op3 -- LSB of product
        // sz1 -- size of multiplier
        // sz2 -- size of multiplicand

        // load multiplicand into product

        // load A with 1st bit of multiplier
        RR LA    AA(op1++) (sz->in) (sz1->inn) (op3->tp3) (op2->tp2)
        // Load M from A, clear A and C
        LM LR LC           (op3++) (inn--)
        repeat   // load product masked
             RR LA MA      AA(tp2++) (in--)
             LR            AA(tp3++)
        while (in)
        LR RC LA LC        AA(tp3)

        // perform multiply

        repeat
             RR LR  AA(op1++) (sz->in) (op3->tp3) (op2->tp2)
             LM               (op3++) (inn--)
             repeat       // add in multiplicand masked
                  RR ACO RC LC LA         AA(tp3++) in--
                  RR ACO OC LC MC LA MA   AA(tp2++)
                  LR                      AA(tp3++)
             while (in)
             LR RC LA LC                  AA(tp3)
        while (inn)
```

```
Divide:

        // op1 -- LSB of divisor
        // op2 -- MSB of dividend
        // sz1 -- size of divisor
        // sz2 -- size of dividend

        // clear (sz1) bits more significant than op2
        LA ((op2+1)->tp2 ) (sz1->in1)//clear A
        repeat
              LR   AA(op2++) (in1--)
        while (in1)

        LA RR      AA(op2) (op2->tp2) (sz2->in2) (op1->tp1)
        repeat
              LC LM LR AA(tp2) (sz1->in1) (op2->tp2) in2--
              repeat
                    RR ACO RC LC LA           AA(tp2++) in1--
                    RR ACO RC LC LA RI MI     AA(tp1++)
                    LR                        AA(tp2++)
              while (in1)
              RI ACO LA           (op2--) (op1->tp1)
        while (in2)
        LR        AA(tp2)

        // op2     -- LSB of remainder (size=sz1)
        // tp2     -- LSB of quotient (size=sz2)
```

```
Logic Functions

And:
        repeat
              RR LC           AA(op1++) in--
              RR RC ACO LC    AA(op2++)
              RC LR           AA(op1++)
        while (in)

Or:
        RI LC
        repeat
              RR LC           AA(op1++) in--
              RR OC ACO LC    AA(op2++)
              RC LR           AA(op1++)
        while (in)

Xor:
        repeat
              RR LA           AA(op1++) in--
              RR ACO LA       AA(op2++)
              LR              AA(op1++)
        while (in)

Not:
        repeat
              RR RI LA        AA(op1++) in--
              LR              AA(op2++)
        while (in)
```

```
Comparison
    // op1 -- MSB of 1st operand
    // op2 -- MSB of 2nd operand
    // C   -- greater than flag
    // M   -- equal flag

    LA RI
    repeat
          LC MC LA MA RR LM     AA(op1--) sz--
          RR RI LA MA           AA(op2--)
    while(sz)

Equal:
     // A -- result flag
     perform Comparison
     RI MI LA

GreaterThanOrEqual:
     // A -- result flag
     perform Comparison
     perform Equal
     RC ACO MA

LessThanOrEqual:
     // A -- result flag
     perform Comparison
     perform Equal
     RI RC ACO MA

NotEqual:
     // A -- result flag
     perform Comparison
     MI LA

GreaterThan:
     // A -- result flag
     perform Comparison
     perform NotEqual
     RC ACO LA MA

LessThan:
     // A -- result flag
     perform Comparison
     perform NotEqual
     RI RC ACO LA MA
```

```
Memory to Memory Moves:

Move:
    // op1 -- LSB of source
    // op2 -- LSB of destination
    repeat
         LA RR          AA(op1++) sz--
         LR             AA(op2++)
    while (sz)

Move (masked):
    // op1 -- LSB of source
    // op2 -- LSB of destination
    // ms  -- exchange mask
    LA RR              AA(ms)
    LM
    repeat
         LA RR             AA(op2) sz--
         LA MA RR          AA(op1++)
         LR                AA(op2++)
    while (sz)

Memory to Memory Exchange:

Exchange:
     // op1 -- location 1
     // op2 -- location 2
     repeat
          RR LA          AA(op1) sz--
          RR LC          AA(op2)
          LC RC          AA(op1++)
          LR             AA(op2++)
     while (sz)

Exchange (masked):
     // op1 -- location 1
     // op2 -- location 2
     // ms  -- exchange mask
     LA RR              AA(ms)
     LM
     repeat
          RR LA MA MC       AA(op1) sz--
          RR MA LC MC       AA(op2)
          LR RC             AA(op1++)
          LR                AA(op2++)
     while (sz)
```

Processor to Processor Moves:

```
MoveN:
    // op1 -- LSB of source
    // op2 -- LSB of destination
    // sz  -- size of operands
    // ds  -- distance
    repeat
          LC,RR          AA(op1++) sz--
          repeat
                LC,AC1        ds--         //if moving north
                // LC,AC1,DR                //if moving south
          while (ds)
          LR,RC           AA(op2++)
    while (sz)

MoveW:
    // op1 -- LSB of source
    // op2 -- LSB of destination
    // sz  -- size of operands
    // ds  -- distance
    repeat
          LA,RR                 AA(op1++) sz--
          repeat
                LA,AC1          ds--       // if moving west
                // LA,AC1,DR                // if moving east
          while (ds)
          LR            AA(op2++)
    while(sz)
```

```
Processor to Processor Moves (masked):

MoveN (masked):
    // op1 -- LSB of source
    // op2 -- LSB of destination
    // sz  -- size of operands
    // ds  -- distance
    // ms -- exchange mask
    LA RR       AA(ms)
    LM
    repeat
          LC RR         AA(op1++) sz--
          repeat
                LC AC1       ds--        //if moving north
                // LC AC1 DR               //if moving south
          while (ds)
          MA RR       AA(op2)
          LR          AA(op2++)
    while (sz)

MoveW (masked):
    // op1 -- LSB of source
    // op2 -- LSB of destination
    // sz  -- size of operands
    // ds  -- distance
    // ms  -- exchange mask
    LA RR      AA(ms)
    LM
    repeat
          LA RR               AA(op1++) sz--
          repeat
                LA AC1        ds--      // if moving west
                // LA AC1 DR              // if moving east
          while (ds)
          MC RR           AA(op2)
          LR RC           AA(op2++)
    while(sz)
```

```
Processor to Processor Exchange (masked):

ExchangeNS (masked):
     // op1 -- LSB of source
     // sz  -- size of operands
     // ds  -- distance
     // nm  -- northern mask
     // em  -- exchange mask
     LC,RR            AA(ms)
     LM
     repeat
          LC,RR       AA(op1) sz-- (ds->dss) // load C
          repeat
               LC,AC1           dss--        // north
          while (dss)
          RI,AC0,LC                         // move C to A
          LC,RR    AA(op1)     (ds->dss)    // load C again
          repeat
               LC,AC1,DR       ds--         // south
          while (ds)
          LC,RR         AA(nm)  // load western mask
          LM
          LC,MC,AC0      // load C from A in northern PEs
          LC,RR        AA(em)    // load exchange mask
          LM
          MC,RR,RC             // load C masked if not exchanging
          LR           AA(op1++)        // store A
     while (sz)
```

Processor to Processor Crossover Exchange:

```
CrossOverNS:
     // op1 -- LSB of source
     // op2 -- LSB of source
     // sz  -- size of operands
     // ds  -- distance
     // ms  -- northern mask
     LC,RR             AA(ms)
     LM
     repeat
          LC,RR        AA(op1) sz-- (ds->dss) // load A op1
          repeat
               LC,AC1           dss--        // north
          while (dss)
          RI,AC0,LA                          // move C to A
          LC,RR     AA(op2)   (ds->dss)      // load C op2
          repeat
               LC,AC1,DR     ds--            // south
          while (ds)
          LC,MC,RC     AA(op1)             // load C from AA(op1)
          LR        AA(op1++)           // store C
          LA,MA     AA(op2)             // load A from AA(op2)
          LR        AA(op2++)           // store A
     while (sz)
```

```
Global Operations:

Global OR:
    // op1 -- parallel operand
    // op2 -- pointer into scalar operand register (SS)
    repeat
         LC RR    AA(op1++)sz--
         AC0 AC1  SS(op2++)
    while(sz)
    repeat
         RC FA    AA(op1++)sz--
         AC0 AC1  SS(op2++)
    while(sz)
```

B Simulation Results and Timing

The following is an example of a simulation for the ADD function. Figure 10 is the timing diagram. Two binary numbers, 1 and 0, are added in that sequence, and the result is stored in memory. The ToRAM line shows the result of this addition 01.

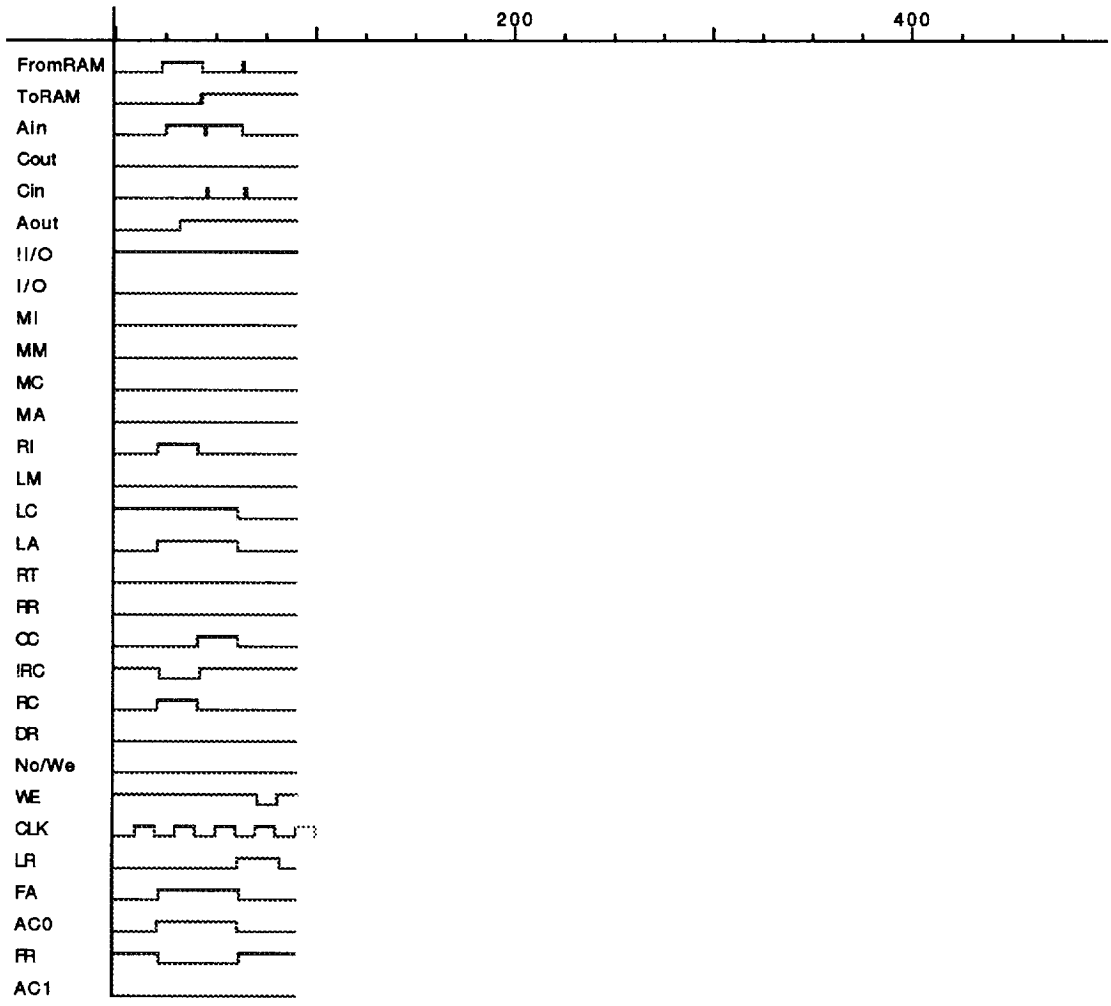

Figure 10: Simulation Timing Diagram for ADD function

The process takes 4 clock cycles to complete - compare this to the primitive command description, which is composed of 4 command lines each requiring a clock cycle to complete.

Figure 11, represents a continuous addition process, where 1 is repeatedly added to itself. The result may be seen as a 10101010... sequence.

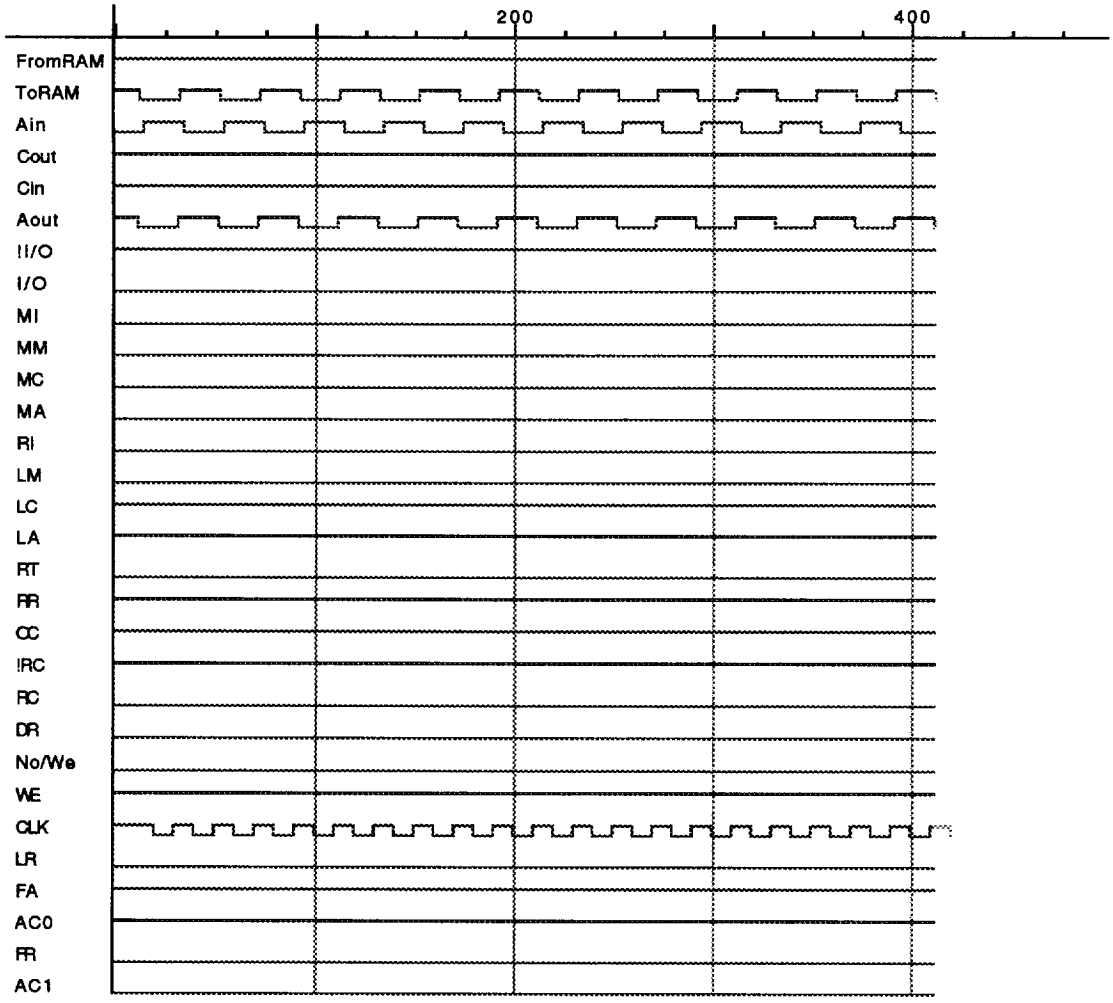

Figure 11: Simulation Timing Diagram for continuous ADD function

References

[1] Bouaziz, S. Pissaloux, E.E., Merigot, A., Devos, F., "Some Hardware and Software Considerations for Multi-SIMD Control Strategy of Massively Parallel Machines", Proceedings. Advanced Computer Technology, Reliable Systems and Applications. 5th Annual European Computer Conference CompEuro '91, pp. 180-183.

[2] Chiang, C., Sarrafzadeh, M., Wong, C.K., "A Powerful Global Router: Based on Steiner Min-max Trees", 1989 International Conference on Computer-Aided Design. Digest of Technical Papers, pp. 2-5.

[3] Dorband, John E., "Processing Element Description", NASA Goddard Space Flight Center, 1989.

[4] Fischer, James R., Schaefer, David H., Wang, Pearl Y., Dorband, John E., "Massively Parallel Computation", *Encyclopedia of Computer Science and Technology*, Ed. Allen Kent, James G. Williams, Carolyn M. Hall, Rosalind Kent, Volume 26, Supplement 11, 1992, pp. 271-332.

[5] Greenberg, R.I., Ishii, A.T., Sangiovanni-Vincentelli, A.L., "MulCh: A Multi-layer Channel Router Using One, Two, and Three Layer Partitions", IEEE International Conference on Computer-Aided Design, ICCAD-89. Digest of Technical Papers, pp. 88-91.

[6] Gross, T., Hinrichs, S., O'Hallaron D.R., Stricker T., Hasegawa A., "Communication Styles for Parallel Systems", IEEE Computer Journal, December 1994, pp. 34-44.

[7] Morley, R.E., Jr., Christensen, G.E., Sullivan, T.J., Kamin, O., "The Design of a Bit-serial Coprocessor to Perform Multiplication and Division on a Massively Parallel Architecture", Proceedings. The 2nd Symposium on the Frontiers of Massively Parallel Computation, pp. 419-422.

[8] Rose, J., "LocusRoute: A Parallel Global Router for Standard Cells", 25th ACM/IEEE Design Automation Conference. Proceedings 1988, pp. 189-195.