The Impact of Pipelining on SIMD Architectures

James D. Allen and David E. Schimmel

School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, GA 30332-0250
{jallen, schimmel}@ece.gatech.edu

Abstract

This paper explores the fundamental impacts of pipeline technology on massively parallel SIMD architectures. The potential for performance improvement in the instruction delivery is explored, and stall penalties associated with reduction operations are derived. Scheduling mechanisms to mitigate stall cycles are also presented. In addition, the design of pipelined processing elements is considered, and formula for stall penalties, and area costs are constructed. These results indicate a 5–10 fold improvement in SIMD performance is well within technological limits.

Keywords: SIMD, Pipelined, Data Parallel, Scheduling

1.0 Introduction

A basic tenet of computer architecture, and indeed all of engineering is to allocate resources where they have the greatest impact. This paper addresses the results of applying pipeline processor techniques to SIMD architectures. We consider the impact of these methods on processor array control, and processing elements, and derive relationships between capabilities of processor memory and communication systems, and pipeline performance. At a time when many advocate the use of off the shelf parts as processing elements, we believe that it is appropriate to revisit these issues, because it is not particularly likely that a commercial microprocessor manufacturer will invest in such an endeavor. Nevertheless, there are strong arguments that SIMD architectures are ideal for truly massive systems from the perspectives of simplicity and regularity. However, we observe that processing element performance has been less than stellar for most SIMD implementations. Thus we are led to inquire whether the evolution of SIMD architecture might benefit from the application of similar principles that led the RISC revolution in the microprocessor arena.

Specifically, we will explore the potential to design a SIMD machine which operates at clock speeds comparable to the current generation of middle to high performance microprocessors; i.e. pipelined implementations which operate in the interval $50\ \text{MHz} \leq f \leq 100\ \text{MHz}$. This is certainly within the range of ca. 1994 process and packaging technologies.

1.1 A Brief History of SIMD

Single instruction multiple data architectures have been in existence for at least twenty years. Table 1 presents the clock period and year of introduction for a selection of these machines[6,19,10,7,5,4,14,13]. A cursory inspection of this data reveals that SIMD has not enjoyed the steady upward trend in clock frequency that has been

Table 1: Clock speeds for various SIMD machines.

Machine	Clock Cycle (ns)	Year
Illiac IV	66	1972
MPP	100	1983
CM-1	250	1985
CM-2	83.3	1987
Blitzen	50	1990
MP-1	80	1990
GF11	50	1990
MP-2	80	1992

1063-7133/95 $4.00 © 1995 IEEE

Proceedings of the 9th International Parallel Processing Symposium (IPPS '95)
1063-7133/95 $10.00 © 1995 IEEE

seen in uniprocessor implementation. Others have also observed this phenomenon [20]. The reason for this lack of improvement is not due to the use of old or low performance technologies. For example, the processor chip for the MasPar MP-2 is fabricated using a 1.0 micron CMOS process. While this is not state-of-the-art in 1995, it is certainly capable of delivering speeds greater than 12 MHz. In the remainder of this paper, we will explore various solutions to overcome this SIMD clock frequency barrier.

1.2 Background

The architectural organization of a typical SIMD machine is shown in Figure 1. An implicit premise of this

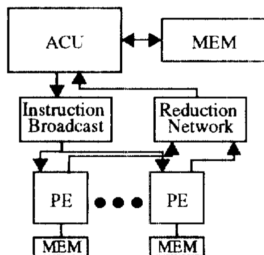

Figure 1. A typical SIMD organization.

organization is that the PEs are directly controlled every cycle. This implicit cycle by cycle synchronization may be one of the fundamental reasons that SIMD clock speeds have not risen over the last two decades. During each cycle, a microinstruction or nanoinstruction is issued from the Array Control Unit (ACU), and executed by the PEs. Since there may be many thousands or conceivably millions of PEs to be controlled, instruction broadcast requires a considerable amount of time to complete. This *instruction broadcast bottleneck* is a fundamental limit on the scalability of SIMD architectures.

2.0 Pipelined Broadcast

An approach to overcome the broadcast bottleneck is to hide the average broadcast latency by pipelining the broadcast. We refer to this as instruction distribution. Figure 1 shows both instruction broadcast and distribution. The instructions are delivered to the PEs via a k-ary tree composed of instruction latches (ILs) in the body of the tree, and PE chips at the leaves of the tree. In fact, depending on the number of PEs per chip, the pipeline tree may extend onto the PE chip as well. As a historical note, this is a generalization of the approach used by the Blitzen project [5]. In that system, PE instructions are passed through a three stage pipeline: decode, broadcast, execute.

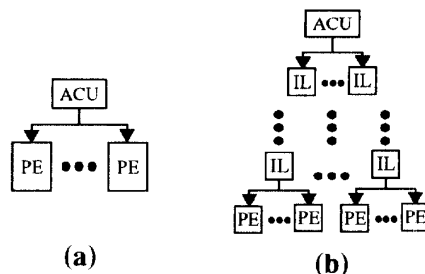

Figure 2. Two methods for delivering instructions to a PE (a) broadcast (b) distribution.

In the decode stage, the instruction is expanded from a 23-bit instruction into a horizontal 59-bit micro instruction. During the second stage, the micro instruction is broadcast to all the PEs. In the final stage all the PEs execute the micro instruction. Since both the decode and execute stages will complete well within the 50ns cycle time, there is a clear opportunity to expand the broadcast stage.

We now consider several issues involved in the design of such an instruction delivery mechanism. In general, for n leaf nodes, and d stages in the network then

$$n = \prod_{i=1}^{d} k_i \qquad (1)$$

where k_i is a positive integer representing the fan out at stage i. However (1) does not directly aid us in the design of the network because n is fixed and d, and k_i are the parameters which are required. We could arbitrarily decide to make the fan out of all stages equal such that $\forall i;\ k_i = k$ (this is a restriction on packaging and board layout). Using this simplification reduces (1) to $n = k^d$. Thus we have $d = \log_k n$. The choice of k is determined by the time required to move an instruction through an individual stage. If we assume a simple capacitive load model for all stages then we can estimate this time as

$$t_i = \tau_i k_i \qquad (2)$$

where t_i is the time required to drive the load, and τ_i is the RC time constant of stage i. Now let t_{MIN} be the minimum clock period possible to execute every microinstruction. This is a constant based on VLSI technology and complexity of the microinstructions. Since the PEs are the drain of the instruction broadcast pipe, we set the cycle time to $T = t_i = t_{MIN}$. Using this value in (2), we can determine the fan out as a function of broadcast technology. Now by formulating how the instruction will be distributed to the PEs, we can determine the minimum number of pipeline stages needed to reach all of the PEs.

Proceedings of the 9th International Parallel Processing Symposium (IPPS '95)
1063-7133/95 $10.00 © 1995 IEEE

If the system is constructed from a number of printed circuit boards inter-connected on a backplane, then T is the maximum of the time required to send one instruction over this bus and t_{MIN}. However, from an architectural stand point, it is immaterial how the instructions are distributed. We are only interested in cycle time T, and the number of pipeline stages, d.

Rockoff has recently proposed a *SIMD instruction cache* as another alternative to overcome the instruction broadcast bottleneck [20]. The notion is to download blocks of instructions to the PE array, and then sequence through them locally. Hence, these instructions avoid the bottleneck. As such, the cached instructions can be processed at a higher speed. This method is able to get substantial speedup on a variety of algorithms.

3.0 Reduction Hazards

Pipelining the instruction broadcast trades latency for throughput. For most instructions the latency in execution is unimportant. However, there is one class of operations which have an interaction from the PEs back to the ACU. We refer to these as *global reduction* instructions. All SIMD architectures have some means of logically combining a value from every PE to produce a single result. For the remainder of this paper, and without loss of generality, we will assume that the operation is a global OR (GOR) of some value on the PEs.

Most GOR occurrences are due to parallel if-then-else constructs. Figure 3 shows the way a compiler might generate code for a branch which depends on parallel data.

```
PR0 = (PR1==PR2)              /* if(PR1=PR2) PR0 = 1 else PR0 = 0 */
A-flag = PR0                  /* set the Active Flag */
R0 = Reduce-Or (A-flag)       /* Make sure at least one PE is active */
if(R0==0) goto else_code      /* if no PEs active skip to else section */
then_code:  ....
```

Figure 3. Pseudo code for a branch on a parallel variable

The Active Flag (A-flag) is a single bit in each PE which indicates that the processor is currently executing. The reduce instruction is placed into the code so that if the set of active processors is empty, time is not wasted in executing those instructions. As a result, a data dependency is created between the *reduce* and the *branch*. In a pipelined system, this dependency may necessitate the insertion of stall cycles after the *reduce*, until the result of the *reduce* is known.

Now consider the number of cycles between when the reduce instruction is executed and when the corresponding branch may be executed. On a base machine with no pipelining, i.e. $d = 0$, let the reduction operation take l cycles. Now define s to be the improvement in clock speed available from pipelining the instruction broadcast, i.e. $T_{old} = sT_{new}$. Assuming that the time to complete a reduction is constant, *reduce* now requires $\lceil sl \rceil$ cycles to execute. In addition, an instruction no longer begins execution on the cycle it is issued from the ACU. Rather, the instructions is executed $d + 1$ cycles after it is issued. So the total number of stall cycles between the *reduce* instruction and the *branch* is $d + \lceil sl \rceil + 1$.

If we assume that none of these stall cycles can be filled with useful work, then the time spent executing *reduce* instructions places an upper bound on the speedup available. It also serves to degrade the performance improvement obtained by increasing the clock speed through pipelined instruction distribution. We find that

$$speedup = \frac{1}{\frac{1 - f_{GOR}}{s} + f_{GOR} + \frac{d}{s} f_{GORI}} \quad (3)$$

where f_{GOR} is the fraction of time spent performing global reduction operations in the original execution, and f_{GORI} is the fraction of instructions that are global reduction instructions. The first two terms in the denominator are the standard terms from Amdahl's Law. The third term represents the additional time spent waiting for the GOR instructions to reach the PEs. Table 2 shows these quantities for several sample programs. These data were obtained from dynamic instruction traces executing on a MasPar MP-2. The program Nn is an artificial neural net program. Cholesky computes the Cholesky factorization of a symmetric, positive definite matrix. Life is Conway's game of life. M/M/1 is a single server single queue simula-

Table 2: Reduction operations as a fraction of both time and instruction count for several programs on the MP-2

Program	f_{GOR}	f_{GORI}
Nn	.0057	.0058
M/M/1	.025	.032
Life	.11	.075
Cholesky	.014	.011
mean	.039	.031

Proceedings of the 9th International Parallel Processing Symposium (IPPS '95)
1063-7133/95 $10.00 © 1995 IEEE

tion program. The last row is the average of the four programs. Placing the values for Nn, Life, and the average of all four programs into (3), we can examine the effects of reduction operations on speed up for various values of s. These are plotted in Figure 4 through Figure 6. The maximum achievable speedup is also plotted. Setting $d = 0$ demonstrates the best case analysis for obtainable speedup. This case might arise from a fast optical broadcast of PE instruction or possibly where an I-cache is very effective (and cold start effects are ignored). Alternatively, it may be viewed as the situation where all stall cycles can be filled with useful work. In general, we expect a higher value of s to correspond to a higher value of d.

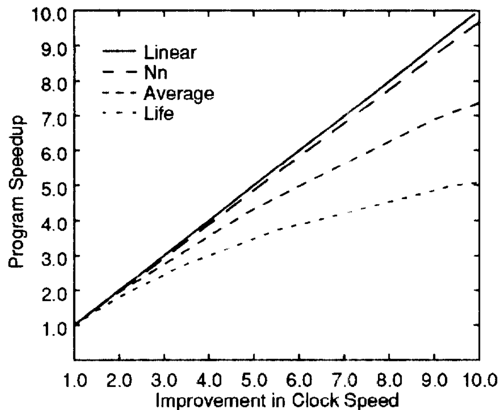

Figure 4. Speedup of sample programs for $d = 0$.

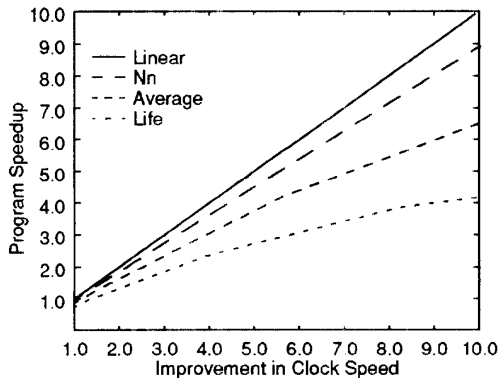

Figure 5. Speedup of Programs for $d = 5$.

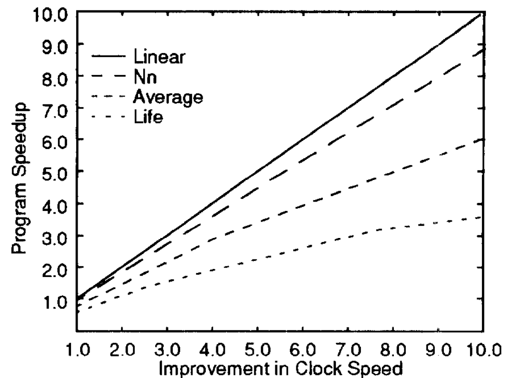

Figure 6. Speedup of Programs for $d = 10$.

3.1 Reduction of Global Reduce Hazards

As the speedup figures indicate, a critical requirement for high speed pipelined distribution of microcode to be effective is the ability to fill the *reduce* stall cycles with useful work. It is tempting to assume that the time for a *reduce* operation scales at the same rate as the clock period. However, since the reduction network is simply a network of logic gates this assumption does not hold. So we will now describe some of the possible methods to fill the stall slots created by GOR instructions. The discussion will center around the pseudo code found in Figure 3.

The most obvious method is to move the GOR instruction earlier in the instruction stream. Unfortunately, in our example each of the four prior instructions has a data dependency with its own prior instruction. This means that the instructions can not be moved up unless the entire block is moved up. Also setting the Active Flag can not be moved ahead of any PE instruction. An alternative is to perform a Reduce-OR on PR0. This has two advantages. First, it allows us to move the setting of the Active Flag into one of the stall slots. Secondly, and of greater importance, it allows us to move instructions from above the first instruction to between the Reduce-OR and Branch as data dependencies allow.

A third possibility is to remove the *reduce* instruction and the corresponding *branch*. The CM-2 does this optimization when running the code fragments that take less time than producing the result of the reduction[7]. It may also be possible to do an analysis of the program at compile time and determine that the active partition will never be empty. In such cases there is no reason to perform the reduction.

Proceedings of the 9th International Parallel Processing Symposium (IPPS '95)
1063-7133/95 $10.00 © 1995 IEEE

Another solution is to assume *branch not taken* and move code from beyond the *then_code* label branch and place that code above the ACU branch. PE instructions from after the branch will execute conditionally based upon the value of the A-flag. This means they can be issued, execute, and even complete before the Reduce-OR is resolved. The reason that this is possible is that (local to each PE) the results of an instruction will be stored conditionally based on the value of the A-flag. If no PEs are active and the branch is to be taken then all of the PEs will squash a broadcast instruction locally. Therefore, there is nothing to undo and the branch can be taken when it is resolved.

There are two classes of instructions that cannot be moved ahead of a branch. The first consists of PE instructions that operate unconditionally. These instructions are normally used in setting and restoring the A-flag. Since they are unconditional instructions, they are not ignored when the A-flag is not set. Unconditional instructions may only be moved ahead of the branch if they do not need to be undone. For example, instructions that operate on a temporary location that will not be read again before it is written could be moved ahead of the branch. The second class of instructions that can not be moved are ACU instructions. Since these will not be squashed, they can not be allowed to complete. However, these instructions could be executed speculatively if that capability is present in the ACU.

4.0 Pipelining PE Instructions

We now consider the possibility of pipelining the execution of PE instructions. This differs from the previous discussion in a number of ways. Before, we were only concerned with hiding the latency involved in broadcasting an instruction to a large number of PEs. We were not changing the instruction executed by the PEs. Other than the possibility of performing the full instruction decode one cycle ahead of execution we were not trying to find parallelism inside the PE instruction. In addition, we did not consider optimization of the PE architecture. In this section, we will investigate all three objectives.

For the processing element (PE) architecture, we propose the use of a 'vanilla' reduced instruction set computer (RISC) approach, such as used by the MIPS R2000, and many others [1, 2,8,9]. In particular, we believe the following features characteristic of that class of architectures will be important in achieving our pipeline and cycle time objectives. First, we posit a simple pipeline either four or five stages deep. Next, all simple integer instructions must execute in one cycle. Third, all instructions must complete in order. We assume that floating point instructions will take multiple cycles and the floating point units may not be pipelined (although FP primitives such as *align* may be). Similarly, integer multiply and divide operations will take more than one cycle and are not explicitly pipelined. Finally, branch conditions are computed using a general purpose register target. That is, our architecture will not utilize condition codes.

What are the effects of data hazards in a pipelined SIMD PE? Hazards between ALU operations can be resolved by forwarding in the usual manner. In addition, if an instruction causes a hazard on one PE it will cause a hazard on all currently active PEs. Therefore it is possible to move the hazard detection circuitry into the ACU. An additional hazard we must consider is introduced by communication operations. Since a result from a communication operation may not be available for several cycles, there is the possibility that communication operations will create long stalls in the instruction pipeline. Generally, we can treat communication hazards as we would load/store hazards.

Control hazards are handled much differently on SIMD PEs compared with uniprocessors. Control hazards result from instructions changing the PC, e.g. branches. On a SIMD machine, there is no (parallel) PC and therefore no branch penalties. Branches on a parallel condition set a register which determines whether received instructions are executed. Therefore we may use this register to control writeback of results from the ALU, and to control the memory interface (this is similar to Section 3.0, where the instruction after the branch was automatically squashed). Assuming all PE state information is explicitly set by the instructions and the active flag is forwarded to the memory interface (in case the active flag is modified followed by a memory instruction), then the instruction immediately following the parallel branch will be executed correctly on the next cycle.

Architecturally, this design has a number of advantages over previous microcoded PEs. First, most instructions are designed to be executed in one cycle. By reducing the ideal CPI of most integer instructions from three or four to one, we gain a corresponding speed up for those instructions. In addition, separating register access and ALU execution into different cycles enables a further reduction of cycle time t_{MIN}.

Proceedings of the 9th International Parallel Processing Symposium (IPPS '95)
1063-7133/95 $10.00 © 1995 IEEE

4.1 Floating Point Performance

Generally, floating point operations take considerable time on SIMD machines. For example, the MP-2 can perform an integer add in three cycles while a floating point add takes 25 cycles [15]. Depending on the methodology chosen for floating point operations, pipelining the integer unit may bring a corresponding improvement in floating point performance. For example, if floating point operations are constructed from primitive operations (supported at the hardware level) similar to simple integer instructions, then floating point performance will improve at nearly the rate of integer performance. Depending on the particular sequences of microinstructions, the speed up for floating point operations should be between one and three. Floating point speedup was bound at three since performance improvement should never be better than that of simple integer operations. Table 3 shows the predicted results for our test examples. For these results, we are

Table 3: Speedup due to pipeline execution of instructions

Program	No speedup of FP	Speedup of 2 on FP	Speedup of 3 on FP
Nn	1.01	1.23	1.56
MM/1	1.06	1.41	2.09
Life	1.17	1.17	1.17
Cholesky	1.02	1.14	1.28
mean	1.06	1.23	1.52

ignoring any improvement in t_{MIN} due to simplification of the architecture. These numbers only reflect the reduction in the total number of cycles. We are also not including a possible reduction in multiply and divide cycles. From Table 3, we see that the only significant speedup is due to reducing the number of cycles needed for floating point operations. This is precisely because these benchmarks are floating point intensive.

5.0 Cost of Pipelined Execution of SIMD Instructions

5.1 Three ported register file

The first necessity for pipelined execution is a multiported register file. The major cost of the multiple register ports is silicon area. We consider two basic cost models that may be adopted. The first model proposed by Snyder and Holman is based on equal area analysis [11]. Under this model, the area used to enhance a PE could have been used to construct additional PEs. A further assumption is that the machine will achieve linear speed up with respect to the additional PEs. Consequently, equivalent speed up is given as

$$su = \frac{c_{BA}}{c_{IPE}}\left(1 - f + \frac{f}{SU}\right)^{-1} \quad (4)$$

where c_{BA} is the area of a PE in the original architecture, c_{IPE} is the area of an enhanced PE, f is the fraction of the time the enhancement is effective, SU is the speed up gained from the enhancement, and su is the speed up gained after considering the cost of the enhancement. Note the second term is simply Amdahl's law. So (4) states that for an enhancement to be worthwhile it has to increase performance by factor greater than the increase in PE size to implement the enhancement.

Since we have already calculated the speedup available from pipelining the execution, we need only to estimate its area cost. For this we use the results of Mulder et. al, who find that single ported registers require approximately 0.6 times the area of triple ported registers [16]. Equivalently, a three ported register file is 1.66 times the size of a one ported register file. Thus

$$c_{IPE} = (1 + 0.66 p_{REG}) c_{BA} \quad (5)$$

where p_{REG} is the fraction of PE area occupied by the register file. On the MP-2 this was found to be 0.3[21]. So for the MasPar, pipelining is advantageous only if it can deliver at least a 20% improvement in performance. This statement ignores any possible reduction in clock cycle time that might be enabled by pipelining the execution of PE instructions. From Table 3, we see that under this constraint a 20% speedup is possible if there is a factor of two or better improvement in floating point performance.

The second model for the cost of silicon area is much simpler. It states that for a machine in at least its second generation, the silicon used to enhance a PE is free so long as it does not change the number of PEs on a chip. The model is based on the observation that most software cannot automatically take advantage of additional PEs. Also some software may count on a certain PE configuration. The addition of PEs may disrupt this configuration and therefore cause the program not to run correctly. Under these assumptions, it is generally not possible to add PEs to the machine. If this is the case then the additional silicon area cannot be used for such purposes. So additional area that becomes available can only be used to enhance

Proceedings of the 9th International Parallel Processing Symposium (IPPS '95)
1063-7133/95 $10.00 © 1995 IEEE

the PEs. This does not eliminate the possible conflict in deciding between two improvements, but such trade-offs are always present.

5.2 I/O Costs

There is an additional cost involved in pipelining the execution of PE instructions. This additional cost is the increase in off-chip I/O bandwidth. The increase in bandwidth is composed of two factors. The first is an increase in the number of bits per cycle that are needed to support one PE. For example, suppose a program runs in c cycles on a non-pipelined machine. On a pipeline machine, the program now runs in c/s cycles, where s is speedup in cycles gained from pipelining. Now consider the I/O requirements of the program on both machines. Let the program generate m word of I/O. The non-pipelined machines requires at least m/c words of traffic per cycle per PE. The pipelined machine requires at least $(sm)/c$ words of I/O per cycle per PE. So the I/O per cycle must grow at the same rate as processor speedup. If the clock rate doesn't change then this factor also represents the increase in bandwidth per second. However, we stated that pipelining should reduce t_{MIN}. This reduction in cycle time produces a second factor in determining required bandwidth. The two factors are *multiplied* together to obtain the total increase in physical bandwidth required per PE. Supporting this high bandwidth requirement may be the hardest part in designing pipelined SIMD machines.

6.0 Conclusions

The clock frequencies of SIMD machines have not increased at the same rate as those of uniprocessors. The principal reason for this is the *instruction broadcast bottleneck*. By transforming instruction broadcast to a k-ary distribution tree, we trade instruction latency for instruction throughput. This enables higher clock rates for the PE array. The speed improvement naturally translates into high system performance.

If the global reduction network is not similarly enhanced then the time to perform *reduce* operations is a limiting factor for the speedup gained from increasing the clock rate. If the instruction broadcast is pipelined, then there are additional penalties due to the latency between issue and execution. By scheduling code, it is possible to fill the slots after a reduction with useful work. It is also possible to mitigate the penalty for a reduction following a parallel branch by moving the GOR instruction beyond the branch. This ability is due to the semantics of SIMD machines.

Additionally, it is possible to pipeline the execution of PE instructions. This allows simple integer operations to execute in one cycle. It also may allow floating point operations to finish in fewer cycles. The resulting speed up is highly dependant on the amount of improvement achieved on floating point operations.

Among issues remaining to be explored are methods of constructing high performance memory and communication systems that are required to support the new higher level of processing. Code scheduling is also important, and new algorithms to effectively schedule slots after a GOR need to be derived. In addition, the set of benchmark programs must be expanded to include a wider variety of algorithms.

7.0 Acknowledgments

The authors would like to thank the reviewers for their comments and a significant correction. We also like thank Dr. Russ Tuck of the MasPar Corporation for his assistance and insights into the MP-2 PE architecture.

8.0 References

[1] Alpert, Donald, and Avnon, Don, "Architecture of the Pentium Microprocessor," IEEE Micro, Vol. 13, June 1993, pp. 11 - 21.

[2] Alsup, Mitch, "Motorola's 8800 Family Architecture," *IEEE Micro*, Vol 10, June 1990, pp. 48 - 66.

[3] Batcher, K.E., "Design of a Massively Parallel Processor," *IEEE Transactions on Computers*, Vol. C-29, No 9, September 1980, pp. 836-840.

[4] Blank, T., "MasPar MP-1 architecture," *Proceedings of COMPCON Spring '90 - The Thirty-Fifth IEEE Computer Society International Conference*, San Francisco, CA, pp. 20-24.

[5] Blevins, D.W., Davis, E.W., Heaton, R.A., and Reif, J.H., "BLITZEN: A Highly Integrated Massively Parallel Machine," *Journal of Parallel and Distributed Computing*, Vol. 8, No. 2, February 1990, pp. 150-160.

[6] Bouknight W.J., *et al.*, "The Illiac IV System," *Proceeding of the IEEE*, Vol 60, No 4, April 1972, pp. 369 - 388.

[7] *Connection Machine Model CM-2 Technical Summary*, Thinking Machines Corporation, Version 51., May 1989

Proceedings of the 9th International Parallel Processing Symposium (IPPS '95)
1063-7133/95 $10.00 © 1995 IEEE

[8] Diefendorff, Keith, and Allen, Michael, "Organization of the Motorola 88110 Superscalar RISC Microprocessor," *IEEE Micro*, Vol 12, April 1992, pp. 40 - 63.

[9] Hennessey, J.L., and Patterson, D.A., *Computer Architecture A Quantitative Approach*, Morgan Kaufmann Publishers, Inc. 1990.

[10] Hillis, W.D., The Connection Machine, The MIT Press, 1985.

[11] Holman, T.J., and Snyder, L., "Architectural Tradeoffs in Parallel Computer Design," *Advanced Research in VLSI: Proceedings of the Decennial Caltech Conference on VLSI*, Cambridge, MA, 1989, pp. 317-334.

[12] Hwang, K and Briggs, F.A., *Computer Architecture and Parallel Processing*, McGraw-Hill. Inc., 1984.

[13] Kim, Won and Tuck, Russ, "MasPar MP-2 PE Chip: A Totally Cool Hot Chip," *IEEE 1993 Hot Chips Symposium*, March, 1993.

[14] Kumar, M., "Unique design concepts in GF11 and their impact on performance," *IBM Journal of Research and Development*, Vol 36, No 6, Nov. 1992, pp. 990-1000.

[15] *MasPar Programming Language (ANSI C compatible MPL) User Guide*, MasPar Computer Corporation, Software Version 3.2, Revision: A5, July 1993.

[16] Mulder, Johannes M., Quach, Nhon T., and Flynn, Michael J. "An Area Model for On-Chip Memories and its Application," *IEEE Journal of Solid-State Circuits*, Vol 26, No. 2, February 1991, pp. 98 - 106.

[17] Nickolls J.R., "The Design of the MasPar MP-1: A cost effective massively parallel computer" *Proceedings of COMPCON Spring '90 - The Thirty-Fifth IEEE Computer Society International Conference*, San Francisco, CA, pp. 25-28.

[18] *Paris Release Notes*, The Thinking Machines Corporation, Version 5.0, February 1989.

[19] Potter, J.L., ed., *The Massively Parallel Processor*, The MIT Press, 1985.

[20] Rockoff, T.E., An Analysis of Instruction-Cached SIMD Computer Architecture, CMU-CS-93-218, Computer Science Division, Carnegie Mellon University, Pittsburgh, PA.

[21] Tuck, Russ., MasPar Corporation, Personal Correspondence

Proceedings of the 9th International Parallel Processing Symposium (IPPS '95)
1063-7133/95 $10.00 © 1995 IEEE