MIST - A Design Aid for Programmable Pipelined Processors

Albert E. Casavant
C&C Research Laboratories, NEC USA, Inc.
4 Independence Way, Princeton, NJ 08540

Abstract — In this paper, a tool to aid pipelined processor instruction set implementation is described. The purpose of the tool is to choose from among design alternatives a design that minimizes overall processor cost. In the proposed cost model processor cost has two components, the cost of hardware necessary to realize the processor and the cost of degraded performance due to pipeline hazards as compared to an ideal pipelined processor. The tool user provides several alternate hardware implementations of each instruction, the cost of hardware operators used, a trade-off factor representing the relative importance of hardware cost versus degraded performance cost, and a straight line benchmark program which is used by the tool to determine frequency of occurrence of pairs of instructions. Using a linear programming approach, the tool selects an implementation for each instruction which gives an overall cost which is optimal.

1. INTRODUCTION

A major hurdle in pipeline design of programmable processors is pipeline hazards [1]. A hazard must be handled either by a compiler or the processor hardware to avoid incorrect processor operation. The compiler deals with hazards by inserting NOPs into the code which results in increased code size and compiler complexity. The hardware deals with hazards by using pipeline interlocks. A pipeline interlock detects a hazard and stalls the pipeline until the hazard producing condition is no longer present. An interlock produces the same effect as a NOP in the code and causes increased hardware complexity. Hazards increase *clock cycles per instruction* (CPI) resulting in reduced pipeline speedup. Generally speaking pipeline hazards can be avoided by a combination of careful design and increased hardware cost. There are some hazards which are unavoidable, such as the load hazard in RISC processors and control hazards due to branch instructions.

Structural hazards are caused by contention for resources by two or more instructions executing simultaneously in the pipeline. If resources are increased appropriately, contention can be eliminated. Data hazards are caused by data dependences between instructions. They fall into three categories RAW, WAR and WAW [1]. Of these the most troublesome in most designs are the RAW hazards. They can be avoided by proper register assignment by the compiler and/or data forwarding. Data forwarding involves the addition of multiplexing on the inputs of hardware operators which require a result generated by an earlier instruction that has not reached the execution stage that writes the result to memory. The added multiplexers select either the normal path from a register file or the memory data register or the data forwarding path.

Integer instructions in RISC processors have very few hazards, due in large part to equal length implementations of those instructions. The same cannot be said of floating–point instructions. In single chip microprocessors, inexpensive general purpose coprocessor designs, and special purpose designs these instructions have different lengths and generally there are several hazards present, as in the R4000 design [2]. Non-RISC designs, such as those typically used in supercomputers, where instructions lengths are variable and pipelines are deep also have many potential hazards.

This paper describes a tool to aid pipelined processor instruction set implementation. The tool should properly be viewed as a design aid because it does not synthesize the hardware, rather it selects from alternate designs that the tool user has provided. Alternatives can be generated automatically using shell scripts from a relatively small number of design templates. If there are n alternatives per instruction and i instructions to be implemented there are n^i combinations of implementations. Enumeration of combinations is practical only for small numbers of instructions having few implementation possibilities. Also it may be necessary to do the optimization several times. The tool described in this paper uses linear programming to avoid looking at the entire exponential solution search space. The purpose of the tool is to choose from among user-provided design alternatives a design that minimizes overall processor cost. Cost has two components: the cost of hardware necessary to realize the processor and any hazard prevention hardware, and the cost of degraded performance due to uncorrected hazards as compared to an ideal pipelined processor. In addition to alternate designs, the tool user provides a trade-off factor representing the relative importance of hardware cost versus degraded performance cost, and a benchmark program which is used to extract instruction frequencies.

Previous work in this general area has concentrated on synthesis [3][4][5][6] rather than analysis. In the “snapshot method” [6], snapshots of instructions from a benchmark are ordered by frequency of occurrence and hardware in a pipelined processor is incrementally added to satisfy the hardware needs of each snapshot. Difficult to schedule and/or unimportant snapshots are allowed to generate stalls in the pipeline. The most significant limitation of this tool is that all instructions are forced to be of equal length, which does not correspond to reality in the floating-point units of most commercial microprocessors. The underlying optimization algorithm is a greedy heuristic and despite its simplicity takes about 30 CPU days to synthesize the pipeline for the IBM ROMP processor and requires significant user interaction. Another drawback to a purely synthesis approach is that the number of design tricks in typical microprocessor datapaths seems to preclude using automatic datapath synthesis in a high volume general purpose application. In the recent past each succeeding microprocessor generation has had innovations in architecture and basic implementation. This designer innovation takes place in parallel with instruction implementation. Thus frequent modifications to the algorithms in the tool would be needed as the design progressed. Using the approach described in this paper, the tool user has complete control over the datapath design options, the tool helps him/her by efficiently searching the design space (independent of design methodology) for good combinations of implementations which minimize overall cost of the processor.

2. PROCESSOR MODEL

The most basic differentiation among pipeline architectures is order of instruction issue and completion. The possibilities are:

1. In-order issue and in-order completion.
2. In-order issue and out-of-order completion.
3. Out-of-order issue and out-of-order completion.

The first two categories are supported by the tool. The third category requires the use of dynamic scheduling, either using a "scoreboard" approach, the Tomosulo algorithm [7], or some variation/combination of the two. The optimization approach used by this tool is incompatible with any architecture where decisions concerning ordering of instructions issued is dependent on the particular hardware implementation of instructions and detailed hardware delays. An exception to "no dynamic scheduling" is a hybrid approach where only loads and stores are dynamically scheduled.

Another major differentiation concerns how the architecture handles instruction level parallelism. The tool is compatible with superscalar, VLIW and superpipelined designs. Currently, only single instruction issue per clock in a "standard" pipeline configuration is fully supported.

3. REQUIRED TOOL INPUTS

The required design information which must be provided to MIST can be summarized as:

1. The number of instructions.
2. The number of implementations of each instruction.
3. The number of execution stages in each implementation.
4. Functional units used and their cost. Functional units with different speeds are considered distinct.
5. *Stage residency* of the functional units, i.e. the stage of execution of the instruction where a functional unit is used.
6. Functional unit sharing options.

Hazards between instructions which occur frequently in programs intended to be run on the processor (as determined from a benchmark provided to the tool) are be given higher weight than hazards between infrequently occurring instructions. Hazards may have varying severity from a one clock cycle stall to several clock cycle stalls. The overall affect on CPI, i.e. the *cost of hazards* (CH), is thus directly proportional to frequency of stall (FOS) multiplied by stall severity (SS). FOS is determined for all possible spacings of instruction introductions into the pipeline.

The cost model used by the tool defines overall processor cost as the sum of hardware cost and the cost of degraded performance. The hardware cost (HC) is the sum of all functional unit costs. The cost of additional multiplexing required when hardware is shared is not considered.

The *trade-off factor* (TF) indicates how much silicon area (in units of area / stall) the designer is willing to trade-off to prevent one stall involving the most frequent instruction pair. Thus overall cost is: $HC + TF * FOS * SS$ in units of area. The designer may use the trade-off factor to balance hardware cost and acceptable performance degradation.

4. STALL STRATEGY

Stall strategy is the method of insertion of stalls in the pipeline. The selection of a strategy is influenced primarily by control complexity. In general, but not always, it is desirable to stall following instructions in as early a stage of execution as possible. It is also sometimes desirable to avoid stalls when a loop is in progress in one of the instructions involved to avoid complex control problems. It is possible that while avoiding one stall involving an instruction pair, other stalls will be caused in later stages of execution of the instruction pair. It may be best to insert a stall from the first occurrence of a hazard lasting until past the end of the last occurrence of a hazard to avoid complex control involved in stalling more than once on the same instruction pair.

Stall strategy is easily modified in the software implementation of MIST, and is independent of the core optimization. Two strategies were chosen for the examples in Section 7. The first of these is good from the point of view of control complexity, but tends to generate many hazards:

1. Stall on structural hazards.
2. When multiple hazards are involved, stall once to clear the last hazard.
3. When an instruction has a loop, stall the following instructions until after the loop.
4. When a following instruction has a loop, stall the following instruction until the loop is not executing simultaneously with previous instructions.

The second strategy is relatively bad from the control point of view, but tends to generate fewer hazards than the first strategy:

1. Stall on structural hazards.
2. Stall only when necessary, allowing multiple stalls on the same instruction pair.
3. Unwind all loops to look for structural hazards and stall only when there is a conflict involving structural resources.

Currently the tool handles only structural hazards. Extension to data hazards is straightforward given register assignments.

5. NODE PACKING FORMULATION

The pipelined processor cost minimization problem described above can be formulated as a *node packing* problem. This problem has also been called the vertex packing or stable set problem and except for some special cases is NP-hard [8]. Among categories of integer programming problems, node packing and knapsack problems are those displaying the most computational success. Solutions of general unstructured integer programs remains very difficult. The relative success in node packing can be traced to theoretical work to find classes of facets of the convex polyhedron associated with the node packing integer program. One such category, maximal cliques [9] is extensively exploited in MIST. Further information on node packing and applications in high level synthesis can be found in [8][10][11]. The clique formulation of the node packing problem is: $MAX \sum_{v \in V} w_v x_v,\ \sum_{c \in C} x_v \leq 1 \quad for \quad all \quad cliques \quad C, \quad x_v \in \{0,1\} \quad for \quad all \quad v \in V.$

For the problem at hand, there are two categories of nodes, *hazard* nodes and *functional unit* nodes. Hazard node variables are one if the corresponding hazard is active and zero otherwise. Hazards are inactive when either or both of the implementations causing the hazard is not chosen in the final solution. Functional unit variables are zero if the functional unit is active (used by any implementation in the final solution) and one otherwise. The weight of a hazard node is the cost of lost performance due to the associated stall(s). The weight of a functional unit node is the cost of the functional unit. The objective function is algebraically manipulated to support 1–active (hazard) and 0–active (functional unit) costs. For each instruction pair, a *bank* of hazard nodes is formed. Each hazard between a pair of implementations of the pair of instructions associated with a bank is represented by only one node in a bank (although one node may represent many such hazards). Hence, exactly one node in a bank can be active in the final solution and the hazard nodes in a bank

form an equality clique constraint. Some banks are further specialized as *superbanks*; there is one superbank corresponding to each instruction. From all the instructions pairs containing a particular instruction, one of them is distinguished as a superbank. Superbanks are similar to normal banks except that there are restrictions on collapsing of nodes, as described below.

Edges fall into two categories, *hazard-to-hazard* (HTH) and *hazard-to-functional-unit* (HTF). For each hazard, define an *i* instruction and a *j* instruction which form the pair causing the hazard. A HTH edge is placed between two hazard nodes which correspond to different implementations in either the *i* or *j* dimension, thereby allowing only one of those hazard nodes to be active. In superbanks only, A HTF edge is placed between a hazard node and a functional unit node which is used in either the *i* or *j* instruction implementations associated with a hazard node. Thus, any use of a functional unit in any implementation associated with an active hazard node will cause the functional unit to be 0–active and its cost will be included in the cost of hardware. Note that this cost is charged only once, regardless of the number of times the functional unit is used.

The category of bank determines what type of node *collapsing* is permitted. Since each node becomes a linear programming variable, it is very beneficial to reduce the number of nodes as much as possible. In "normal" banks, collapsing is permitted when the costs of the nodes are the same (or close when there is a non-zero error tolerance, as discussed below) for all combinations of implementations which are represented by the collapsed node. HTH edges are adjusted accordingly. Since there are many possible sets of compatible combinations possible, a heuristic set covering algorithm is used to maximize collapsing. In superbanks, collapsing is more restrictive, since not only costs must be the same but functional unit use must be the same to permit collapsing. Also, in superbanks the *i* dimension must have only one implementation represented per node. Permitting multiple implementations in the *i* direction will allow infeasible choices of implementations in some cases.

6. CONSTRAINT GENERATION

Constraints are generated by inserting arcs among nodes in banks. For each constraint a *driving bank* is chosen. A particular node in that bank is chosen (the *driving node*) and following a clique generation strategy, a maximal clique constraint is generated. It is not guaranteed to be a maximum clique. Given a driving bank, there are three possible *driving modes*. Arcs are created in *ij* mode if nodes in the *i* dimension of the driving bank are connected by arcs to the *j* dimension of the driven bank. The *ji*, *ii*, and *jj* driving modes are defined similarly. In concert with the node-packing paradigm, a driving node connects to a driven node if they are *incompatible* i.e. they cannot be simultaneously chosen as part of the solution. A valid solution to the problem will have one node chosen (linear programming variable equal to 1) for each bank.

Constraint generation proceeds by connecting all possible nodes in the driven bank to the original node in the driving bank. In this sense, constraint generation is greedy. Other constraints may be possible by adding fewer than the maximum nodes in the driven bank to the constraint, so constraint generation is not exhaustive. Note that for each driving bank, there are two possible driven banks using different driving modes. For example, if one driven bank is driven in *ij* mode by the driving bank, then another driven bank can be driven in *jj* mode. The two driven banks can then interact in *ii* mode. Figure 1 shows this interaction. Additional arcs are shown for cliques generated among the nodes in a bank. Note that for a given driving bank, constraints are generated independently for each driven bank. A different constraint *may* be generated, because interaction between driven banks favors a different dominant driven bank on each of the two constraint generations involving the same three banks. Each node of the driving bank becomes a driving node for forming constraints. Conflicts between the two driven banks, i.e. inconsistent node choices with respect to the common instruction in both driven banks are resolved in favor of the dominant driven bank for that constraint. After conflict resolution, a *reflection* operation on the driving bank is performed. This operation adds more nodes in the driving bank in an attempt to maximize nodes in the constraint. Of course, all nodes added must be consistent with nodes already in the constraint. In the example of Figure 1, there are no nodes added. In the event that a driven bank is not active, i.e. there are no arcs connecting it to the driving bank, there is a possibility of an alternate third bank. A fourth instruction which does not appear in the active driven bank or driving bank may be either an *i* or *j* dimension of a alternate third bank.

The constraint generation algorithm given above is repeated for constraints involving functional units. For each functional unit, nodes are added to a constraint when they DO use the function unit, in keeping with the notion that functional units are active when their corresponding node variables are zero. Thus constraints will include the functional unit node and hazard nodes compatible with each other and the functional unit. Unfortunately, constraint generation of this type does not guarantee that functional unit use is properly constrained. If all the uses of each functional unit in each superbank do not appear in other constraints, then additional constraints are added.

The number of banks is approximately i^2, where i is the number of instructions. For each bank, each node in the bank can be a driving node for a constraint. Checking for the presence of arcs between driving and driven bank and checking compatibility between driven banks are On^2 operations where n is the maximum number of implementations of any instruction and n^2 is the maximum number of nodes in a bank. When an alternate third bank is called for, there are $i - 1$ possibilities for this bank. Constraint generation is done, in the worst case, for each functional unit, f. Hence, the number of constraint generation cycles is Oi^2n^2f and each constraint generation cycle takes On^2i, giving an overall complexity of On^4i^3f. There is

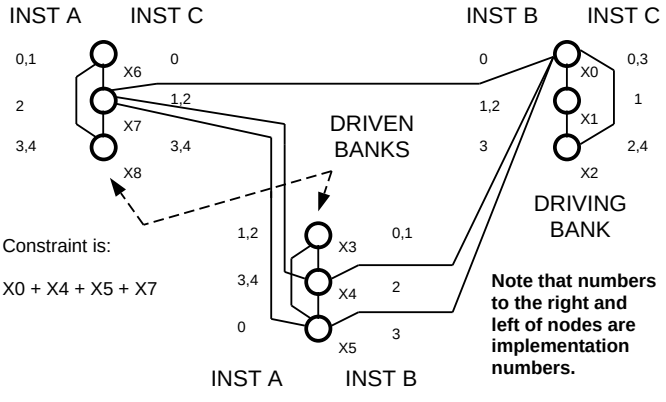

Figure 1: Constraint Generation.

a significant constant multiplier needed for the overhead associated with performing these operations.

7. EXAMPLE GENERATION AND COMPUTATIONAL EXPERIENCE

The following types of implementation variability are present in the examples described below:

1. Algorithmic.
2. Hardware sharing, both intra-instruction and inter-instruction.
3. Hardware speed.
4. Hardware stage residency.

Algorithmic variation refers to the implementation algorithms for instructions e.g. for a multiplier, booth, radix 4 and radix 8, one pass to three passes, or the odd/even method [12] [13], radix 4 and radix 8, one pass to two passes, or modified Wallace tree [12] [14], radix 4, one pass. Inter-instruction hardware sharing is used to save total hardware, but possibly at the cost of increased stalls due to structural hazards e.g. the *fp-add* and *fp-mult* instructions use/don't use the same carry propagate adder. The *fp-add* uses the adder for adding, rounding or both. *Fp-mult* uses the adder as a final carry propagate add after partial products have been computed by carry save adders. Intra-instruction hardware sharing is also used to save hardware, but effectively breaks the pipeline at the point where sharing occurs e.g. the partial product adder and rounding adder (required for compliance with the IEEE floating-point standard) use/don't use the same adder. Hardware speed variation and hardware stage residency often go hand-in-hand. Moving hardware between stages may allow the use of slower (less expensive) components e.g. final exponent addition may be performed in the same pipeline stage as the rounding addition (requiring fast, expensive adders because both adds are on the critical path) or in different stages (requiring slower, less expensive adders).

An example based on a 4 instruction floating point unit was constructed. The *fp-add-sub* instruction exhibits 4 different add algorithms, taking advantage of techniques given in [13] and [15]. Also shown in this instruction is both inter and intra-instruction sharing of the exponent add and mantissa add hardware and different stage residencies, giving a total of 36 implementations. The *fp-mult* uses 5 basic algorithms in both radix 4 and radix 8. The amount of sharing of carry-save adders is varied yielding different numbers of iterations through the carry-save stage of multiplication. There is also inter and intra-instruction sharing of the exponent add and final carry propagate add of the partial products, exploiting techniques given in [16] and different stage residencies, giving a total of 300 implementations. *Fp-div* and *fp-sqrt* are implemented quite similarly and follow closely techniques given in [17] and [18]. Radix 2 and radix 4 implementations of SRT division, with intra-instruction sharing of hardware in the basic loop give variation from 3 passes for one of the radix 8 implementations to 24 passes for one of the radix 2 implementations. The final carry propagate add is possibly shared with *fp-add-sub* and/or *fp-mult*. The number of implementations for both *fp-div* and *fp-sqrt* is 80.

Overall operation of the tool is as follows:

1. All potential hazards among all possible implementations of all possible instruction pairs are generated. Nodes representing the hazards are placed in banks and collapsing is performed.
2. Constraints are generated.
3. Optimization is performed using CPLEX™, a commercial linear programming code from CPLEX Optimization, Inc.

Tables 1 and 2 give the computational results associated with both the example discussed above (indicated by "lg") and an abbreviated example having 22, 36, 40 and 40 implementations respectively for *fp-add-sub*, *fp-mult*, *fp-div*, and *fp-sqrt* (indicated by "sm"). Two stall strategies were used as described in Section 4 and indicated by "ld" for low or "hd" for high density of hazards. Two trade-off factors are represented, "lw" for low weight, indicating that one hazard between the two most frequently occurring instruction pairs can be traded off for one tenth of an adder, and "hw" for high weight, indicating the trade-off is for ten adders. Two quantities of constraints were generated, "f" for full constraint generation i.e. all constraints which constraint generation can produce, and "a" for abbreviated, a minimum set guaranteed to give correct results. The last number gives the error tolerance as 0, 5, or 10 percent. Error tolerances greater than zero have the effect of increasing the amount of collapsing possible by allowing nodes to collapse into one node when their costs fall within the error tolerance. The overall effect is to intentionally cause errors in the input data, with a benefit of fewer variables in the linear programming formulation of the problem. The *worst case number of hazards* is calculated assuming that every possible pair of implementations of each pair of instructions will generate a hazard. The *number of raw hazards* is the actual number of hazards which were detected using the specified stall strategy. The *number of variables* is the number of variables in the resulting linear program and reflects the results of node collapsing and the error tolerance on the original number of hazards.

Table 2 show the results of constraint generation and linear programming optimization. The *number of matrix entries* is the total number of non-zeros in the constraint matrix and is a good indication of overall size of the problem. The optimization time is given as two entries, the first is the time used by CPLEX™ to solve the linear programming program with all data loaded into its data structures. The other entry is the time to read data from files, perform other miscellaneous reporting, etc. during optimization and solve the linear program. The time shown for constraint generation includes that for hazard generation. The *measured error* is the error of the result compared with the optimal choice as determined by enumeration. When there is a non-zero error tolerance, costs of nodes are determined as the center of a band of costs.

It should be re-emphasized that MIST does not guarantee finding an integer solution and indeed, of 100 randomly generated problems tested, 7 yielded fractional answers.

8. EXTENSIONS AND CONCLUSION

An instruction pair is *enclosed* by another instruction pair if the first instruction of the enclosing pair is introduced prior to the first instruction of the enclosed pair and the last instruction of the enclosing pair is introduced after the last instruction of the enclosed pair. Enclosed pairs which generate hazards can cause errors in the analysis using the current formulation. Extension of MIST to handle this enclosed pairs is currently under investigation.

References

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

[2] G. Kane and J. Heinrich, *MIPS RISC Architecture*. Prentice Hall, Englewood Cliffs, NJ, 1992.

#	PROBLEM NAME	NUM VARI-ABLES	NUM RAW HAZARDS	WORST CASE NUM HAZARDS	NUM COMBS	ENUM-ERATION TIME
1	rand_20	385	428	10,115	$3.2 * 10^{16}$	NA
2	fp_sm_ld_lw_f_0	644				
3	fp_sm_ld_lw_a_0	644				
4	fp_sm_ld_lw_f_10	621	785			
5	fp_sm_hd_hw_f_0	711				
6	fp_sm_hd_hw_a_0	711		7170	1,267,200	352
7	fp_sm_hd_hw_f_5	578	6873			
8	fp_sm_hd_hw_f_10	565				
9	fp_lg_ld_lw_a_10	3846	6906	71,456	69,120,000	19,371

Table 1: Floating point example data — part 1.

#	PROBLEM	NUM CON-STRAINTS	NUM MATRIX ENTRIES	GEN TIME	OPTIM TIME	MEAS'D ERROR
1	rand_20	1307	21,277	6	.53 / 4	NA
2	fp_sm_ld_lw_f_0	14,502	1,664,958	788	13.0 / 174	0%
3	fp_sm_ld_lw_a_0	5,085	628,612	383	3.9 / 63	0%
4	fp_sm_ld_lw_f_10	13,600	1,574,798	712	12.75 / 165	6.53%
5	fp_sm_hd_hw_f_0	9,794	1,183,138	638	9.1 / 153	0%
6	fp_sm_hd_hw_a_0	3,740	422,988	198	2.7 / 56	0%
7	fp_sm_hd_hw_f_5	7,642	823,640	486	7.0 / 104	2.50%
8	fp_sm_hd_hw_f_10	7,413	787,652	464	6.9/ 103	5.01%
9	fp_lg_ld_lw_a_10	15,382	3,901,420	36,702	25.6 / 1584	6.51%

Table 2: Floating point example data — part 2.

[3] P. Paulin and J. Knight, "Force-Directed Scheduling in Automatic Data Path Synthesis," in *24th Design Automation Conference*, pp. 195–202, 1987.

[4] C.-T. Hwang, Y.-C. Hsu, and Y.-L. Lin, "Scheduling for Functional Pipelining and Loop Winding," in *28th Design Automation Conference*, pp. 764–769, 1991.

[5] M. Nourani and C. Papachristou, "Moving Frame Scheduling and Mixed Scheduling-Allocation for Automated Synthesis of Digital Systems," in *29th Design Automation Conference*, pp. 99–105, 1992.

[6] R. J. Cloutier and D. E. Thomas, "Synthesis of Pipelined Instruction Set Processors," in *30th Design Automation Conference*, pp. 583–588, 1993.

[7] R. M. Tomosulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM Journal of Research and Development*, vol. 11, no. 1, pp. 25–33, 1967.

[8] G. L. Nemhauser and L. A. Wolsey, *Integer and Combinatorial Optimization*. Wiley Interscience, 1988.

[9] M. W. Padberg, "On the Facial Structure of Set Packing Polyhedra," *Mathematical Programming*, vol. 5, pp. 199–215, 1973.

[10] G. L. Nemhauser and G. Sigismondi, "A Strong Cutting Plane/Branch-and-Bound Algorithm for Node Packing," tech. rep., School of Industrial and Systems Engineering, Georgia Institute of Technology, 1989.

[11] C. H. Gebotys and M. I. Elmasry, *Optimal VLSI Architectural Synthesis: Area, Performance, Testability*. Kluwer Academic Publishers, 1992.

[12] V. Peng, S. S., and G. M., "On the Implementation of Shifters, Multipliers, and Dividers in VLSI Floating Point Units," in *8th Symposium on Computer Arithmetic*, pp. 95–102, 1987.

[13] D. Goldberg, *Computer Architecture A Quantitative Approach*, ch. Computer Arithmetic, pp. A1–A66. Morgan Kaufmann Publishers, Inc., 1990.

[14] W. M. McAllister and D. Zuras, "An NMOS 64b Floating-Point Chip Set," in *ISSCC 86*, pp. 34–35, 1986.

[15] J. Gosling, "Some Tricks of the (Floating-Point) Trade," in *6th Symposium on Computer Arithmetic*, pp. 218–220, 1983.

[16] M. R. Santoro, G. Bewick, and M. Horowitz, "Rounding Algorithms for IEEE Multipliers," in *9th Symposium on Computer Arithmetic*, pp. 176–183, 1989.

[17] J. Fandrianto, "Algorithm for High Speed Shared Radix 4 Division and Radix 4 Square Root," in *8th Symposium on Computer Arithmetic*, pp. 73–79, 1987.

[18] J. Fandrianto, "Algorithm for High Speed Shared Radix 8 Division and Radix 8 Square Root," in *9th Symposium on Computer Arithmetic*, pp. 68–75, 1989.