

# Issue Logic for a 600-MHz Out-of-Order Execution Microprocessor

James A. Farrell and Timothy C. Fischer

**Abstract**—The logic and circuits are presented for a 20-entry instruction queue which scoreboards 80 registers and issues four instructions per cycle in a 600-MHz microprocessor. The request logic and arbiter circuits that control integer execution are described in addition to a novel compaction scheme that maintains temporal order in the queue. The issue logic data path is implemented in 141 000 transistors, occupying 10 mm<sup>2</sup> in a 0.35- $\mu$ m CMOS process.

**Index Terms**—CMOS digital integrated circuit, issue, micro-processor, out-of-order, queue.

## I. INTRODUCTION

THE microprocessor contains four integer execution units arranged as two clusters, with each cluster containing a complete copy of the register file, as shown in Fig. 1 [1]. The physical distance between the clusters requires that one cycle of latency exists between the conclusion of an operation in one cluster and the availability of the destination register in the other cluster. The register result data cross between the clusters on the intercluster register bypass buses.

The integer issue logic schedules instructions to minimize the intercluster latency by trying to issue instructions on the same cluster that produces the dependent register. The instructions are statically assigned to request either the upper or lower pairs of execution units, allowing an instruction the opportunity to issue on either cluster. Static assignment is a determination of which execution unit, or pair of execution units, an instruction may issue to before the instruction enters the queue.

The main integer issue operation for the out-of-order microprocessor is performed by a 20-entry instruction queue. The queue allows any of the 20 instructions to issue to either cluster, which is a 10% performance improvement on SPECINT95 benchmarks over a queue that statically assigns instructions to a single cluster. A previous out-of-order instruction queue allowed instructions to issue to only one execution unit, as was physically determined by the instruction's location in the queue [2]. As will be described later in the paper, maintaining the oldest instruction in the bottom of the queue and adding new instructions to the top of the queue simplifies the arbitration. The microprocessor fetches four instructions per cycle, so the integer issue queue must be able to both

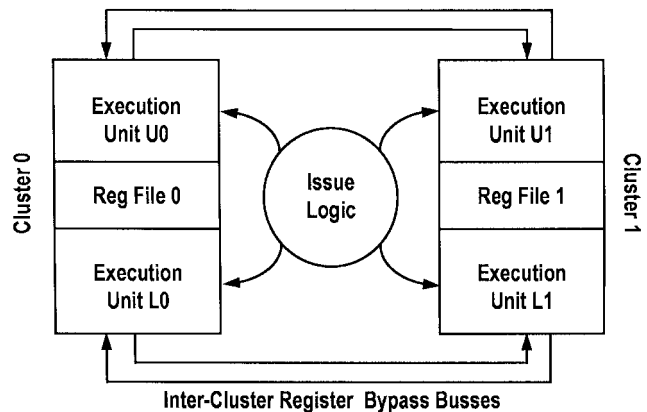


Fig. 1. Location of execution units with respect to register files.

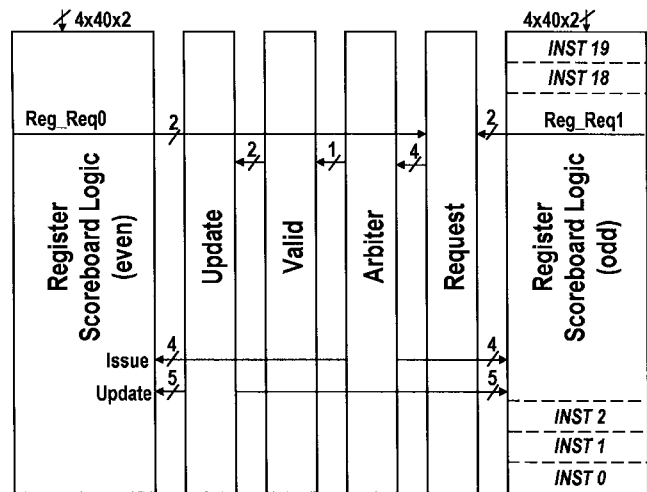


Fig. 2. Block diagram of issue logic, with horizontal wiring for one of 20 rows shown.

enqueue and issue up to four instructions each cycle to maintain peak bandwidth.

Fig. 2 shows the issue logic datapath for the instruction queue. Shown is the horizontal wiring for 1 of 20 rows, with one instruction occupying each row. Instructions are ordered with the oldest in row 0 at the bottom and the newest in row 19 at the top. Up to four instructions enter the instruction queue per cycle, requiring compaction of the instructions remaining in the queue toward the bottom. There are five major components to the datapath.

- 1) The Register Scoreboard logic tracks data dependencies for 80 registers and is split in half to reduce circuit

Manuscript received September 1997; revised December 1, 1997. The authors are with Digital Equipment Corporation, Hudson, MA 01749 USA.

Publisher Item Identifier S 0018-9200(98)02229-X.

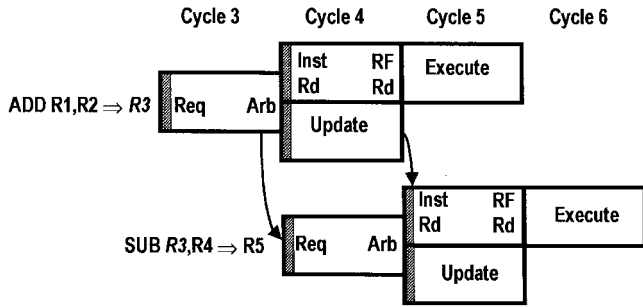


Fig. 3. Timing diagram showing interaction between two instructions in the pipeline.

delays. The logic holds two decoded fields: a decoded destination register number and a logical OR of the instruction's two decoded source register numbers. In each cycle, the source field is compared to outstanding destination register numbers and a match is signaled on one of two register request wires running across the datapath. This is the checking of the read-after-write register conflict.

- 2) The Request logic stores execution unit assignment information and combines it with the two register request signals. The output is a 4-b field that indicates the execution units requested to the arbiter by each row.
- 3) The Arbiter contains two pick-oldest-two arbiters operating in parallel to choose four instructions each cycle for execution.
- 4) The Valid logic maintains a valid bit for each instruction and deasserts one of the request lines when invalid. It calculates a new valid bit each cycle based on issue, invalidate, and reset information.
- 5) Finally, the Update logic accepts new valid bits each cycle and produces MUX selects which compact the instructions in the queue.

Fig. 3 is a timing diagram illustrating how the instruction issue information is communicated to subsequent instructions. The ADD enters the queue in cycle 3, requests an execution unit, and wins the issue arbitration by the end of the cycle. The issue information is used several ways. First, the issue signal indexes into an array (separate from those shown in Fig. 2) containing instruction data (Inst Rd) that is passed to the corresponding execution unit for the register file read (RF Rd) and execution stages. Next, the issue of the ADD is communicated to the SUB, which is dependent on R3. The SUB can request to issue in cycle 4 if R4 is also available. Finally, the ADD participates in the update calculation, which overwrites its row in cycle 5 since the ADD has exited the queue.

The queue is compacted at the beginning of each cycle, as indicated by the shaded areas in the timing diagram. The update logic calculates MUX selects from valid state of instructions in the queue and moves enough instructions to the bottom of the queue to free up to four rows at the top for the newly fetched instructions. If four rows cannot be freed, then stall is asserted for earlier stages in the pipeline [3].

The next three sections detail each of the major components of the issue logic: Request, Arbiter, and Update. Two full cycles are necessary to complete the logical path from an instruction entering the queue to when the instruction is overwritten by the Update logic after issuing.

## II. REQUEST LOGIC

The register scoreboard logic in Fig. 4 shows how the update movement is performed. One vertical register slice for two rows is shown. Source (Src) and destination (Dst) field data enter from the busses at the top and are MUXed with recirculated data coming from the storage latches. The update wires indicate which set of data will be written into this row.

The incoming source data are immediately compared to the vertical register clean wires, which indicate if that register is available for use by the instruction. If the register is unavailable, the register request wire is deasserted. The destination data are pipelined for one cycle in the latch, and then compared to the issue signal for that instruction. If the instruction does not issue, then the destination register will not be available for dependent instructions in the following cycle.

The register request is created from the odd and even array request wires into a global request signal for the four execution units, as shown in Fig. 5. Four request signals are sent to the arbiter for each row, of which only one signal is shown. Execution unit assignment information, L0.H, is logically combined with other instruction state information, labeled as OTHER.L, in a precharged gate. The gate output is latched as the global request signal, Req.L0.L, in this example.

On the right side of Fig. 5, note that of the four L0.H wires that enter into the MUX from rows above, only three continue on to rows below in the array. The MSB of the bus is replaced by shifting the bus by one bit and merging in the latch output. This method allows an instruction to shift down by four rows using a maximum of four metal tracks.

## III. ARBITER

The arbiter accepts the four request signals from each row and decides which four instructions will issue. One Pick-2 arbiter decides for the lower execution units and one decides for the upper execution units. This Pick-2 arrangement enables the performance increase from reducing the cross-cluster latency impact.

There is only one phase allocated for arbitration, so only one clock edge is available to initiate this operation. A standard priority encoder is a precharged NMOS array with one winning grant line left precharged. Unfortunately, that circuit will not allow the dominoing of one array into another. For example, if the highest priority row requests both execution units, it will win in both priority encoders. This would allocate two execution units to the same instruction.

Our solution is to use the output of the first array as the set for an RS latch, and the grant from the second array as the reset. If an instruction loses in the first array, it enables, or sets, the output latch for the second array to issue.

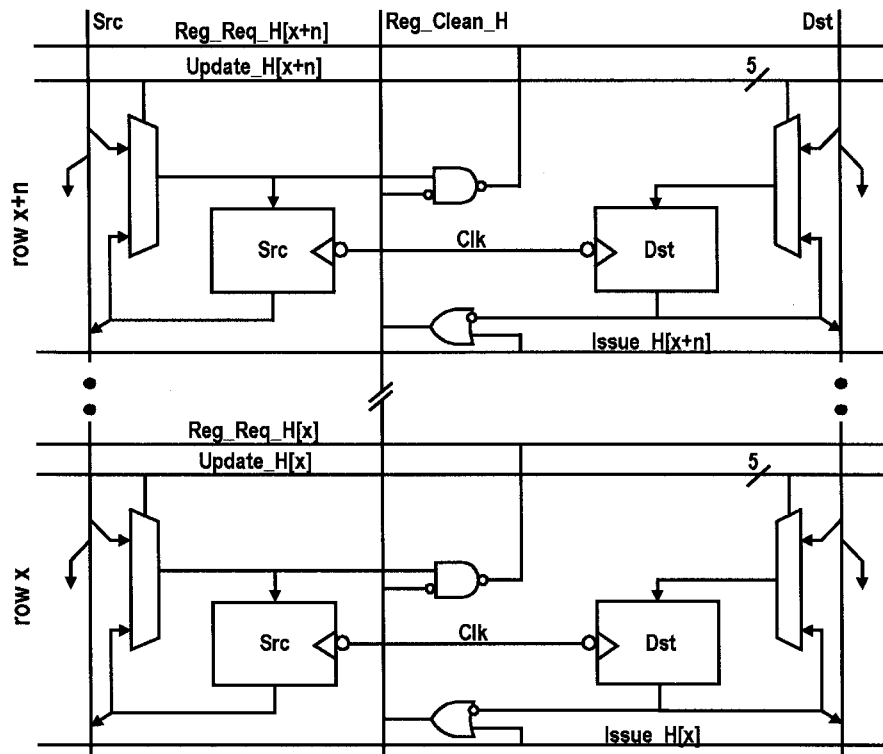


Fig. 4. One register slice of the register scoreboard array, with two rows illustrated.

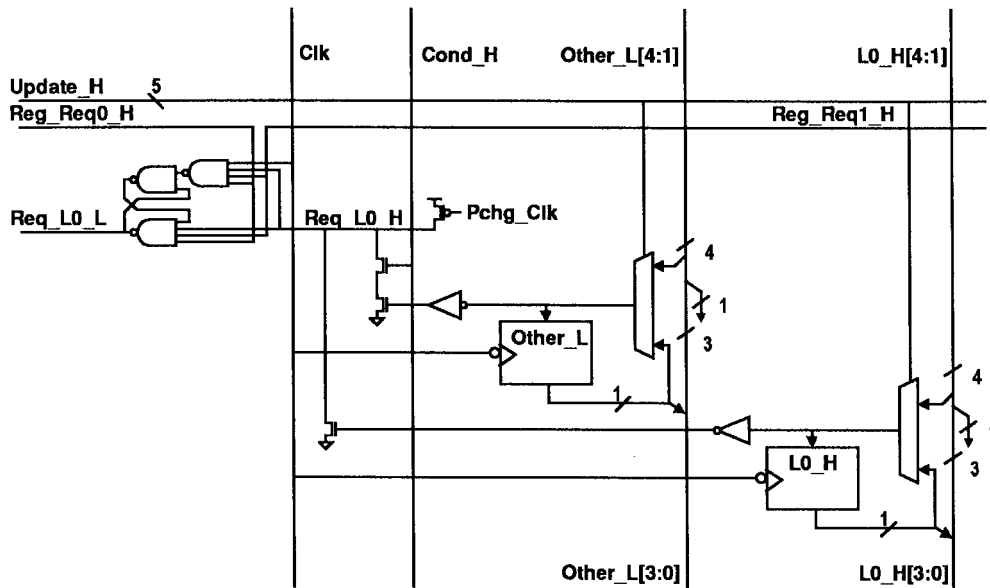


Fig. 5. Request logic for one row, showing the movement of instruction information through the MUXes.

The advantage of compacting the queue to the bottom each cycle is critical for this circuit. If the instructions are not physically ordered, the priority encoders would be significantly more complicated and would not be able to meet the cycle time.

Fig. 6 is a simplified circuit showing three rows from the arbiter. The stage 0 priority encoder on the right is an array of kill signals and precharged grant signals. Row 0 is the highest priority and discharges all upper row grants if it requests the first execution unit.

Stage 1 combines the request signals for the second execution unit with the grant signal from the previous array into a kill signal. Thus, the stage 1 kill signal is only active if the row did not win in stage 0. The grant signal in the second stage is like the first stage: the highest priority active kill signal discharges all of the lower priority grant signals. As stated previously, if row 0 requests both stages 0 and 1, both grant signals for row 0 will still be precharged at this point. This difficulty is overcome by using the grant signal from the first array to set the output latch if it does not issue, thus

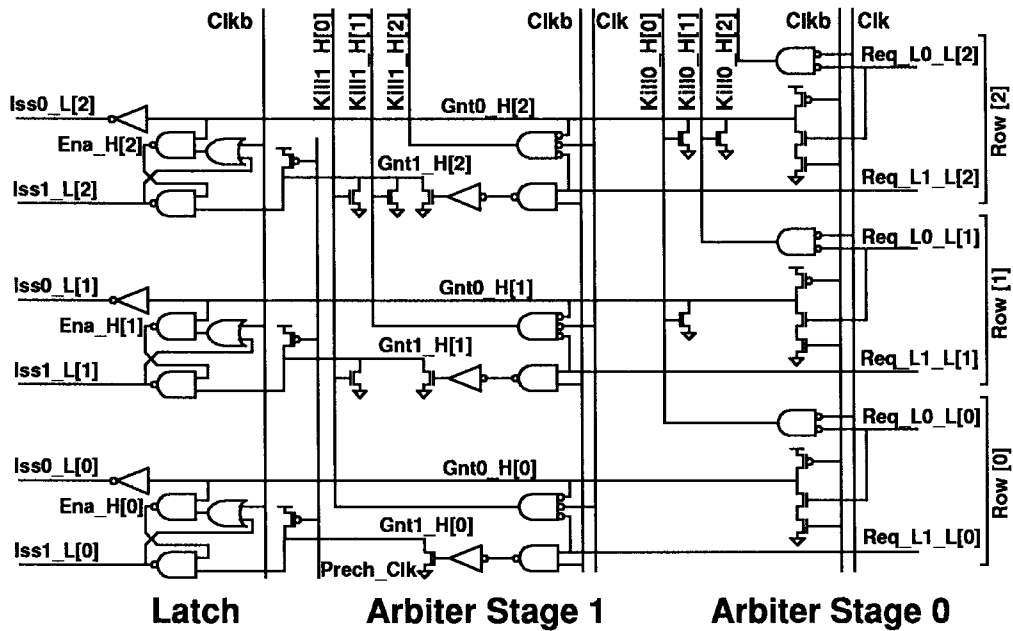


Fig. 6. Simplified circuit showing three bottom (highest priority) rows of the arbiter.

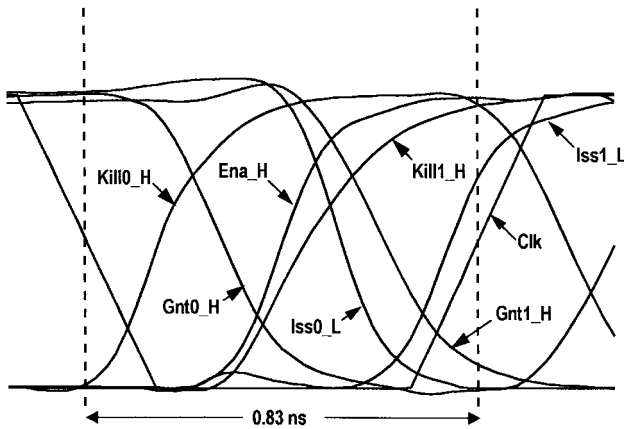


Fig. 7. SPICE simulation of the arbiter for example circuit shown in Fig. 6.

allowing the issue signal to be enabled for the grants from the second array. For the case where row 0 requests in both arrays, Gnt0\_H[0] and Clkb are both high, so the Iss1\_L would remain deasserted.

The arbitration sequence can be further explained with the timing diagram in Fig. 7. Assume that all three lower rows have requested both execution units. The kill signals from row 0 and row 1, Kill0\_H and Kill1\_H, discharge the grants in stage 0 and stage 1, Gnt0\_H and Gnt1\_H, respectively. As Gnt0\_H falls, the enable signal for that row is asserted, which in turn asserts its corresponding Iss0\_L signal. The deassertion of Gnt1\_H, however, results in Iss1\_L being deasserted, which is the critical path for the arbiter.

The issue signals are now forwarded to the instruction storage array, the register scoreboard logic, and the valid logic.

Row	Cycle [N]	Valid	Invalid Count				Cycle [N+1]
			4	3	2	1	
7	Instr H	0	0	0	0	0	•
6	Instr G	1	0	0	0	0	•
5	Instr F	1	0	0	0	0	•
4	Instr E	0	0	0	0	0	•
3	Instr D	0	0	0	0	0	•
2	Instr C	1	0	0	0	0	→ Instr G
1	Instr B	0	0	0	0	0	→ Instr F
0	Instr A	1	0	0	0	0	→ Instr C
			0	0	0	0	→ Instr A

Fig. 8. Algorithm for update logic.

The valid logic combines the issue signal with other state, such as reset, to determine which queue rows will be removed by the update calculation.

#### IV. UPDATE LOGIC

The update logic accepts a new valid signal from each instruction every cycle. The instruction is either invalid and removed from the queue next cycle, or is shifted between zero and four rows lower in the queue to free room at the top for new instructions to enter.

Fig. 8 illustrates the algorithm employed for compaction. The update algorithm counts valid bits from the bottom up as shown: in cycle *N*, instructions *B*, *D*, *E*, and *H* are invalid. Each invalid bit increments a 4-b saturating counter, implemented by shifting a token in a 5-b vector left one position. Thus, row 0 does not shift, leaving the count at 0, but row 1 shifts for a count of 1. Row 2 is valid and leaves the count at 1.

This count is turned into MUX selects as shown by the diagonal grouping: selects for a row are produced from the

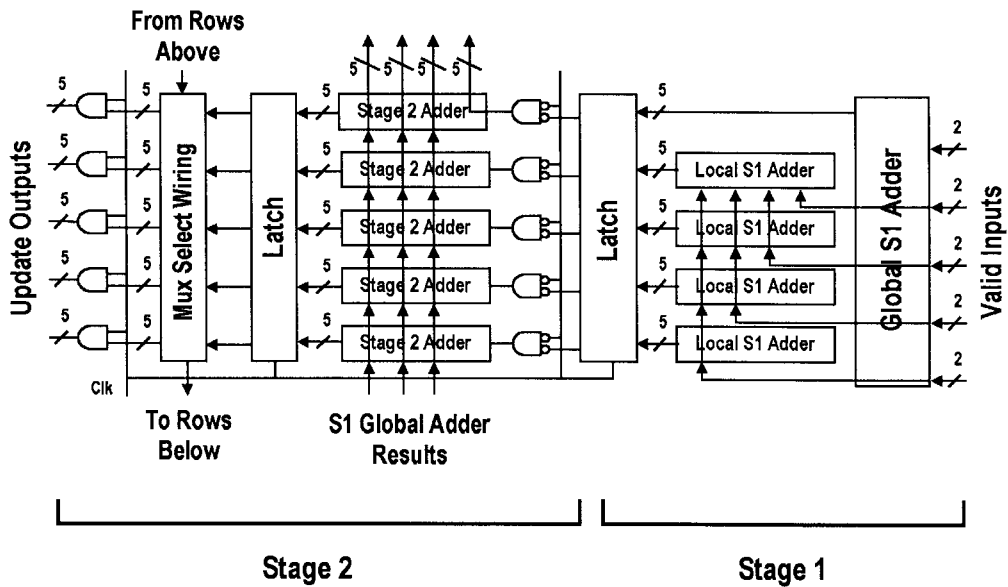


Fig. 9. Block diagram for the update logic.

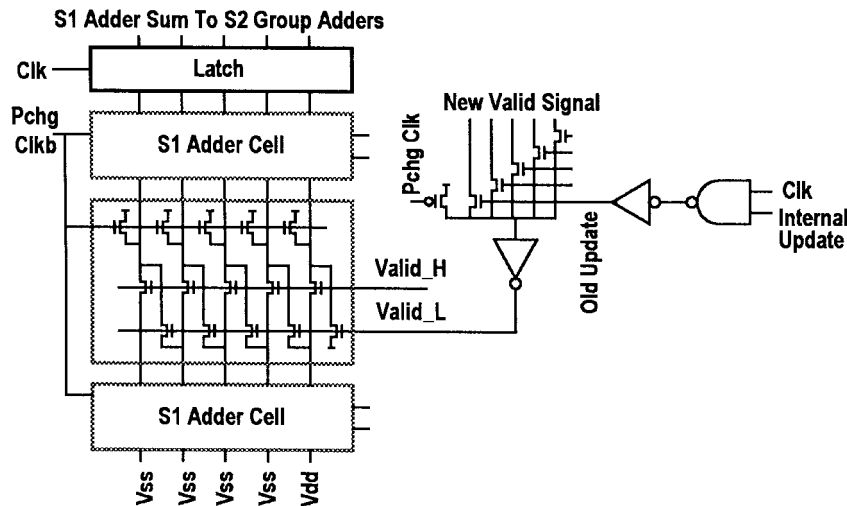


Fig. 10. Simplified circuit showing first stage adder of the update logic.

invalid counts from that row to four rows above it. The count bit in each row is logically ANDed with the valid bit in each row, as shown by the  $X$  over certain tokens in the Invalid Count. For example, rows 3 and 4 are invalid, so their count bit is masked and only one valid MUX select remains on the diagonal for instruction *C*. In cycle  $N + 1$ , instructions *B*, *D*, *E*, and *H* have been removed from the queue and the remaining instructions have been compacted down, maintaining their original order. Instructions in rows 4 and higher shift down four, freeing up four rows at the top for new instructions.

The Update Logic has one cycle of latency and consists of two stages, each taking one phase to complete. Shown in Fig. 9 is a five-row slice of the 20-row update datapath. The structure of the datapath is similar to a carry lookahead adder, with the global adders performing the lookahead function.

Stage 1 adds instruction invalid bits up to and including each row and saturates at a sum of 4. Stage 1 adders are broken into groups of five rows, as shown, to reduce their delay. Local adders in four of the five rows sum the invalid bits within the group up to and including the designated row. Global adders sum all five invalid bits in each group, similar to a lookahead adder. Local and global adder sums are then driven into stage 2.

Stage 2 combines local and global sums from stage 1 and forms MUX selects for each instruction queue row as shown in the diagonal in the previous slide. Stage 2 outputs are clocked MUX selects, driven across the entire instruction queue which control precharged NMOS pass transistor MUXes to shift instructions within the queue.

Fig. 10 is the stage 1 adder, shown in more detail to illustrate how the update MUX selects from the previous



Fig. 11. Microphotograph of issue logic.

cycle initiates the new update calculation. Each instructions new valid bit is calculated from its issue signal during low assertion time of Clk, phase *B*. In phase *A*, or the high assertion time of Clk, the valid bit is shifted into its new row through a precharged NMOS pass MUX and dominoes into the stage 1 adder using the old update MUX outputs. Note that complementary versions of each row's valid bit are needed for the pass/shift function. The example shown would have each of three rows drive into the S1 Adder cells.

The adder is implemented as a 5-b precharged NMOS shifter array for fast addition, as illustrated by the center box. The center box adds by shifting a token left one bit for each valid entry. The five bits represent an invalid count that saturates at four. Shown is a typical local adder that sums the valid bits for three rows.

The stage 1 sum is latched at the top and driven to stage 2 in phase *B*. The update logic stage 2 output is turned into MUX selects as previously described and is driven across the issue logic datapath during the next phase *A*, which closes the loop.

## V. CONCLUSION

A microphotograph of the issue logic is shown in Fig. 11. The datapath at the bottom of the photo contains 141 000 transistors and occupies 10 mm<sup>2</sup> of the 314-mm<sup>2</sup> die. The part is fabricated in a drawn 0.35- $\mu$ m process, featuring six layers of metal. Two of the metal layers are coarse pitch, two are fine pitch, and two are planes supplying power and ground. The part is estimated to operate at 600 MHz. First-pass silicon has demonstrated queue operation on multiple operating systems with no known bugs or electrical sensitivities at over 550 MHz at a VDD of 2.2 V. Planned edits to the microprocessor will increase the overall frequency to over the target of 600 MHz.

The key to high-speed microprocessor design is deciding which problems are worth solving and which are not. Choosing a small number of problems to solve allows careful analysis of architecture, circuits, layout, and process to meet the high-speed goals. This instruction queue minimizes the cross-cluster latency problem and maintains very high frequency.

The Arbiter described picks the oldest instructions that can issue each cycle with a novel cascaded priority encoder. Compaction of the oldest instructions to the bottom of the queue each cycle simplified the arbitration circuits at an acceptable complexity and area cost.

## ACKNOWLEDGMENT

The authors acknowledge the contributions of the Instruction Box Design Team (M. Bhaiwala, S. Britton, C. Chafin, H. Fair, B. Fields, D. Leibholz, S. Meier, N. Raughley, and M. Tareila) and B. Gieseke, who made significant contributions to the circuits described.

## REFERENCES

- [1] B. Gieseke *et al.*, "A 600 MHz superscalar RISC microprocessor with out-of-order execution," in *ISSCC Dig. Tech. Papers*, Feb. 1997, pp. 176–177.
- [2] N. B. Gaddis, J. R. Butler, A. Kumar, and W. J. Queen, "A 56-entry instruction reorder buffer," in *ISSCC Dig. Tech. Papers*, Feb. 1996, pp. 212–213.
- [3] T. Fischer and D. Leibholz, "Design tradeoffs in stall control circuits for 600 MHz instruction queues," in *ISSCC Dig. Tech. Papers*, Feb. 1998.



**James A. Farrell** received the B.S.E.E. (magna cum laude) and M.S.E.E. degrees from Rensselaer Polytechnic Institute, Troy, NY, in 1985.

He is with Digital Equipment Corporation, Hudson, MA. He has participated in the design of three microprocessors, including a low-temperature project and the Alpha 21264. He is currently responsible for the implementation of the Alpha 21264A microprocessor.



**Timothy C. Fischer** is a Principal Engineer at Digital Equipment Corporation, Hudson, MA. He is currently leading the instruction fetcher implementation for a next-generation Alpha CPU. He has contributed to the design of several generations of Alpha and VAX microprocessors and coauthored many papers on CPU design.