Power- and Complexity-Aware Issue Queue Designs

The improved performance of current microprocessors brings with it increasingly complex and power-dissipating issue logic. Recent proposals introduce a range of mechanisms for tackling this problem.

Jaume Abella and Ramon Canal
Universitat Politècnica de Catalunya

Antonio González
Universitat Politècnica de Catalunya and Intel Barcelona Research Center

Current microprocessors are designed to execute instructions in parallel and out of order. In general, superscalar processors fetch instructions in order. After the branch prediction logic determines whether a branch is taken (or not) and its target address, the processor decodes the instructions and renames the register operands, removing name dependences introduced by the compiler. Because processors generally have more physical than logical registers, multiple instructions with the same logical destination can be in flight simultaneously. The renamed instructions then go into the issue queue where they wait until their operands are ready and their required resources are available. At the same time, instructions go into the reorder buffer, where they remain until they commit their results. When an instruction executes, the wakeup logic notifies dependent instructions that the corresponding operand is available. Finally, instructions commit their results in program order.

This article focuses on the design of the logic that stores the instructions waiting for execution, as well as the logic associated with identifying whether operands are ready and selecting the instructions that start execution every cycle. All these components are part of the *issue logic*. Issue logic is one of the most complex parts of superscalar processors, one of the largest consumers of energy, and one of the main sites of power density. Its design is therefore critical for performance.

Researchers have used a variety of schemes to implement the issue queue. In particular, several recent proposals have attempted to reduce the issue logic's complexity and power. To the best of our knowledge, this article is the first attempt to perform a comprehensive and thorough survey of the issue logic design space.

Basic CAM-based approaches

One of the most common ways to implement the issue logic is based on content-addressable memory (CAM) and RAM array structures. These structures can store several instructions, but generally fewer than the total number of in-flight instructions. Each entry contains an instruction that has not been issued or has been issued speculatively but not yet validated and thus might need to be reexecuted.

In general, entries use RAM cells to store operations, destination operands, and flags indicating whether source operands are ready while CAM cells store source operand identifiers—referred to here as *tags*. After the issue logic selects an instruction for execution, it broadcasts the instruction's destination tag to all the instructions in the issue queue. The wakeup logic compares each source tag in the queue with the broadcast tag and, if there is a match, marks the operand as ready. This process is known as *wakeup*. A superscalar processor can broadcast and compare multiple tags in parallel. Figure 1 shows a block diagram of the issue logic associated to one entry of the issue queue. Whether the issue queue stores operand values or just operand tags affects the design, as Sima[1] and others discuss.

The *selection* process identifies instructions whose source operands are ready and whose required resources are available, and then issues them for execution. When more than one instruction competes for the same resource, the selection logic chooses one of them according to some heuristic.[2]

Overall, the issue logic's main source of complexity and power dissipation is the many tag comparisons it must perform every cycle. Researchers have proposed several approaches to improve the issue logic's power efficiency. We classify these approaches into two groups:

- *static* approaches, which use fixed structures, and
- *dynamic* approaches, which dynamically adapt some structures according to the properties of the executed code.

Orthogonally, researchers have proposed several more efficient circuit designs, but they don't reduce the inherent complexity.[3]

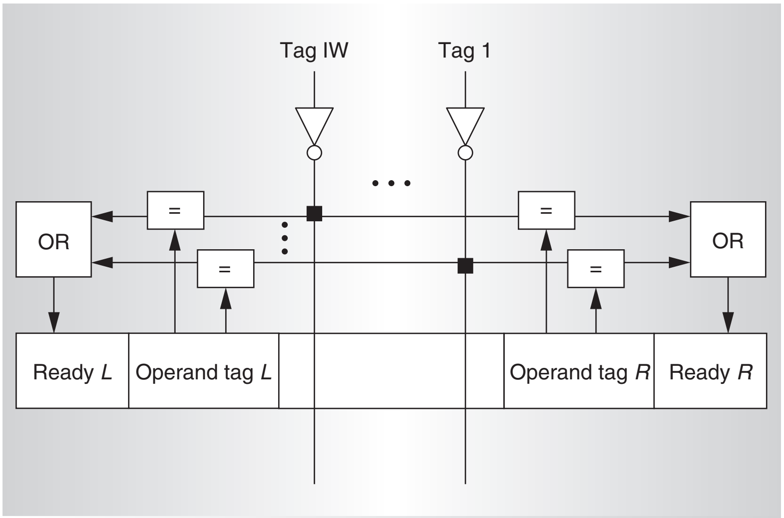


Figure 1. Issue logic for an entry in a CAM/RAM array. Issue width (IW) is the number of instructions issued per cycle.

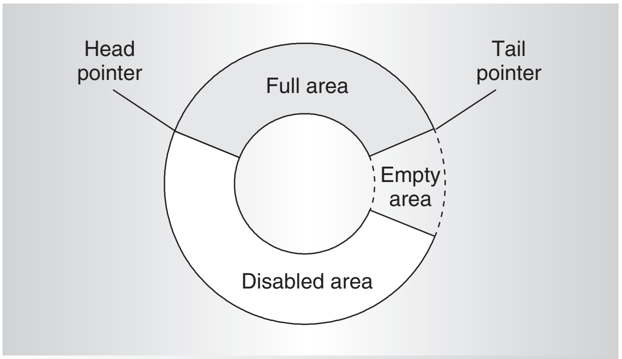


Figure 2. Adaptive-size issue queue. The head and tail pointers define the used area, which can expand through the empty area but not the disabled area.

Dynamic approaches

One approach to reducing the power dissipation is based on disabling the wakeup logic for CAM cells that are either empty or correspond to operands that are already ready (that is, have been woken up but the instruction hasn't been issued). This approach calls for gating off each cell's wakeup logic based on the value of the ready and empty bits,[4] but only saves dynamic power.

A multibanked implementation of the issue queue can also help reduce static power by turning off entire banks when they are empty. Figure 2 shows an adaptive-size issue queue with resizing capabilities.

Albonesi analyzes the relationships between issue queue size, latency, and performance[5] while Buyuktosunoglu et al.,[6] Ponomarev et al.,[7] and Dropsho et al.[8] adjust issue queue size to match the number of occupied entries. More aggressive techniques reduce the issue queue size even when the entries are occupied if doing so doesn't appear to significantly degrade performance. Folegnani and González propose a scheme that monitors the instructions-per-cycle rate (IPC) from the youngest part of the issue queue.[4] Every certain number of cycles, this technique calls for increasing the issue queue size by one bank. If the number of committed instructions issued from the youngest part of the queue is below a given threshold, however, issue queue size decreases by one bank.

Abella and González propose a mechanism that takes resizing decisions based on the time that instructions spend in both the issue queue

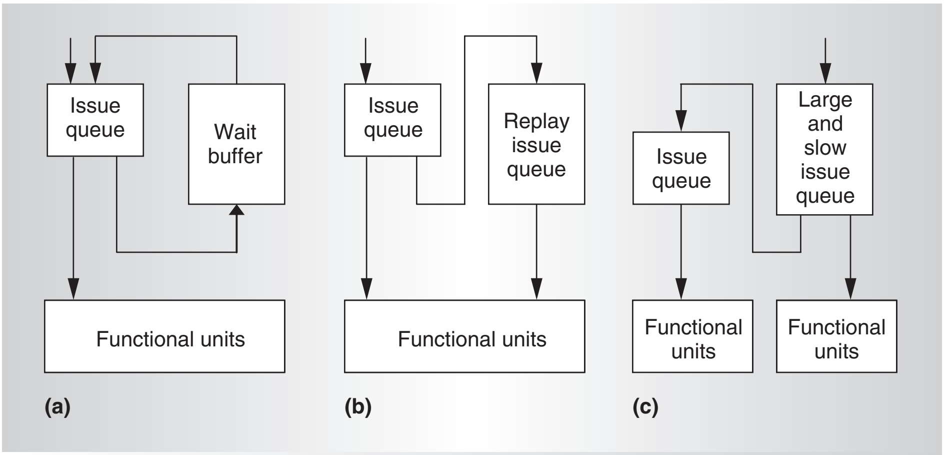


Figure 3. Schemes for static approaches using a CAM-based issue queue: conventional queue with waiting buffer (a), replay issue queue (b), and criticality-based queues (c).

and the reorder buffer.[9] If the ratio between the time in the issue queue and the time in the reorder buffer is below a given threshold, the mechanism increases the reorder buffer size. If it is above another threshold, the mechanism decreases it. The mechanism sets these thresholds dynamically, making the mechanism more aggressive when the reorder buffer size is large. On the other hand, this scheme resizes the issue queue based on the number of cycles the dispatch stage stalls because of unavailable issue queue entries.

Static approaches

Because implementing large out-of-order issue queues at high clock rates is difficult, some schemes combine small issue queues with simpler structures. The issue logic dispatches only a subset of the in-flight instructions to the small and complex issue queues, sending the rest to the simpler structures. These simpler structures do not allow a full out-of-order issue, however. In this type of scheme, deciding which instructions go to each structure is critical for performance.

For instance, because instructions that depend on a load that misses in cache will not issue until the miss is serviced, Lebeck et al.[10] propose a mechanism that places instructions in a conventional issue queue, but, when a load misses in cache, it moves all the instructions that depend directly or indirectly on the load to a waiting buffer. After the cache services the miss, the mechanism moves the instructions back to the issue queue as the proposed waiting buffer has no issue capabilities. Figure 3a depicts this scheme.

Current superscalar processors speculatively issue load-dependent instructions assuming that the load will hit in cache; otherwise, they pay a significant performance penalty. Such processors usually keep these instructions in the issue queue until there is confirmation that the load hit in cache. If the load misses, the instructions must be reissued. This speculative technique obviously increases the required number of issue queue entries.

Moreshet and Bahar apply smart techniques to move speculatively issued instructions to another buffer, in response to load-hit predictions.[11] Their mechanism moves such instructions to the replay issue queue (see Figure 3b) once they are issued. If the load hits, the mechanism removes the instructions; if it misses, they are reissued through the replay issue queue. The mechanism prioritizes the replay issue queue for reissued instructions. To simplify the selection logic, the mechanism issues instructions from only one queue in a given cycle. A load that misses in the cache causes some cycles to elapse between miss detection and servicing. Thus, you can implement the replay issue queue so that it dissipates less power and is less complex than a conventional queue, because its latency has a reduced impact on performance.

Another way to decide which instructions go into the fast issue queue relies on estimating the instructions' *criticality*.[12,13] Criticality is the number of cycles that the data path can delay an instruction without affecting the program's execution time. No one has devised a feasible method to compute instruction criticality in a real program, so the proposed schemes use heuristics to estimate the criticality.

If we can classify instructions according to their criticality, we can implement the issue logic with a small and fast CAM-based issue queue for critical instructions, and a larger and simpler slow issue queue that dissipates less power for the rest. Brekelbaum et al.[14] propose classifying instructions as dynamically critical or noncritical. The instruction dispatch logic sends all the instructions to the slow issue queue, as Figure 3c shows, and every cycle the issue logic moves the oldest nonready instructions in the slow issue queue to the fast issue queue. Brekelbaum et al. base their proposal on a two-cluster microarchitecture with an issue queue in each cluster, making the selection logic simpler.

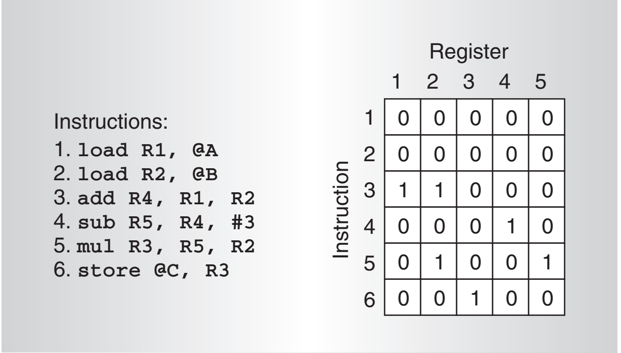
Another way to reduce the issue queue complexity is based on the observation that most instructions have one or none nonready operands at dispatch time. Ernst and Austin propose three issue queues: one without CAM logic for instructions ready at dispatch, one with CAM logic for instructions with only one nonready operand at dispatch, and a third with CAM logic for instructions with both operands nonready at dispatch.[15]

Matrix-based approaches

Bit matrixes are an alternative way to implement the issue logic. The bit matrix has as many rows as entries in the issue queue, and as many columns as physical registers. Figure 4 shows a matrix-based issue queue scheme. In the example, it is assumed that the instructions go into the issue queue in the same order as they appear, and that there are only five registers.

When the issue logic receives an instruction from the dispatch logic, this scheme calls for clearing all the bits in that instruction's row except for those corresponding to its nonready input physical registers. The wakeup process clears the column corresponding to the newly generated output physical register. An instruction is ready when all the bits in its row are zero.

A CAM-based implementation holds the tags, and the structure's resulting complexity is logarithmic with respect to the number of physical registers. In contrast, a matrix-based


Instructions:

```
1. load R1, @A
2. load R2, @B
3. add R4, R1, R2
4. sub R5, R4, #3
5. mul R3, R5, R2
6. store @C, R3
```

	1	2	3	4	5
1	0	0	0	0	0
2	0	0	0	0	0
3	1	1	0	0	0
4	0	0	0	1	0
5	0	1	0	0	1
6	0	0	1	0	0

Figure 4. Matrix-based issue queue. Rows correspond to instruction entries in the issue queue, and columns correspond to physical registers in the system.

implementation holds bit vectors that result in a linear complexity with respect to the number of physical registers. Weaknesses of a CAM-based issue queue are the high number of CAM cell ports required by the wakeup process (because many results can be generated in a given cycle), and high power dissipation. The weaknesses of a matrix-based issue queue also relate to size: The matrix can contain many rows and columns, it can require numerous decoders to select the column for clearing during the wakeup process, and the logic to detect when an instruction is ready can be complex because it must check many bits.

Researchers have suggested improvements to the matrix-based approach that would reduce matrix complexity and size. Goshima et al.[16] distribute the matrix for integer, floating-point, and load/store instructions and narrow the matrix based on their observation that most instructions depend on instructions that are close to each other in the reorder buffer.

Other researchers have used the matrix-based approach to implement the selection logic. Brown, Stark, and Patt propose using a matrix to detect which operands are ready, which functional units an instruction requires, and which functional units are available.[17] Their approach allows pipelining the issue logic.

Dynamic code prescheduling

Issue logic schemes based on dynamic code prescheduling offer another alternative to conventional issue queues. Prescheduling-based schemes attempt to schedule instructions in an in-order buffer, ordering the instructions according to their expected issue time. This

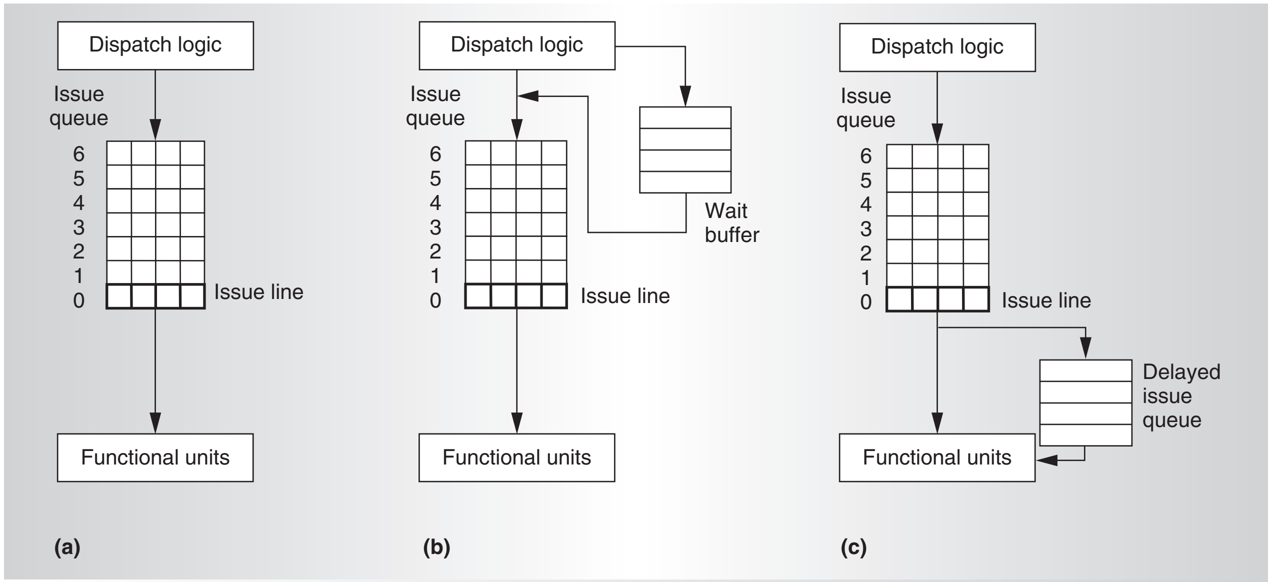


Figure 5. Structure of code-layout issue queues: basic (a), distance (b), and deterministic latency (c) schemes.

scheme calculates instruction issue time based on the latencies and expected issue times of producer instructions (those instructions producing results used by other instructions, consumer instructions). These techniques are an approximation of a runtime VLIW organization. The scheme's main advantage is elimination of the associative search needed for wakeup.

Although estimating operation latency is trivial for fixed-latency nonmemory instructions, it is not so easy for memory accesses. The schemes we describe differ mainly in their solutions for variable-latency instructions, and in the way they deal with direct or indirect consumers.

Figure 5a depicts the basic structure of the prescheduling proposals we discuss in this section. The dispatch logic places the instructions in the issue queue, according to their estimated issue time. Only the instructions at the bottom of the queue (the *issue line*) are candidates for execution. These schemes use simplified wakeup and selection logic because they consider the availability of both the operands and the execution resources when scheduling.

Researchers have proposed several techniques for dealing with variable-latency operations. In the distance scheme,[18] illustrated in Figure 5b, an instruction buffer (the *wait buffer*) holds all instructions dependent on an unknown-latency load until its latency is known. The instruction buffer then preschedules them in the issue queue with the rest of the instructions.

Another alternative is to schedule the instructions assuming a fixed latency (that is, the latency of an instruction that hits in the L1 cache) and move these instructions to an alternative buffer if they arrive at the issue line and are not ready for issue. The alternative buffer can be an in-order or out-of-order issue queue. Figure 5c depicts a block diagram of this scheme, called the deterministic latency scheme.[19]

Michaud and Seznec propose using the associative buffer (the *issue buffer*) for all instructions.[20] The instruction buffer preschedules instructions and then sends them to the issue buffer before execution. Instructions remain in the issue buffer until they are ready to execute. Figure 6a illustrates this scheme.

Raasch et al. propose a prescheduling-based scheme in which all instructions except those that depend on a load that has missed in cache advance toward the bottom of the queue every cycle.[21] The scheme adds a field in each entry of the issue queue to identify dependence chains. All instructions that depend on a load that misses in cache can continue advancing so they reach the bottom of the queue just in time to consume the loaded data. Because of

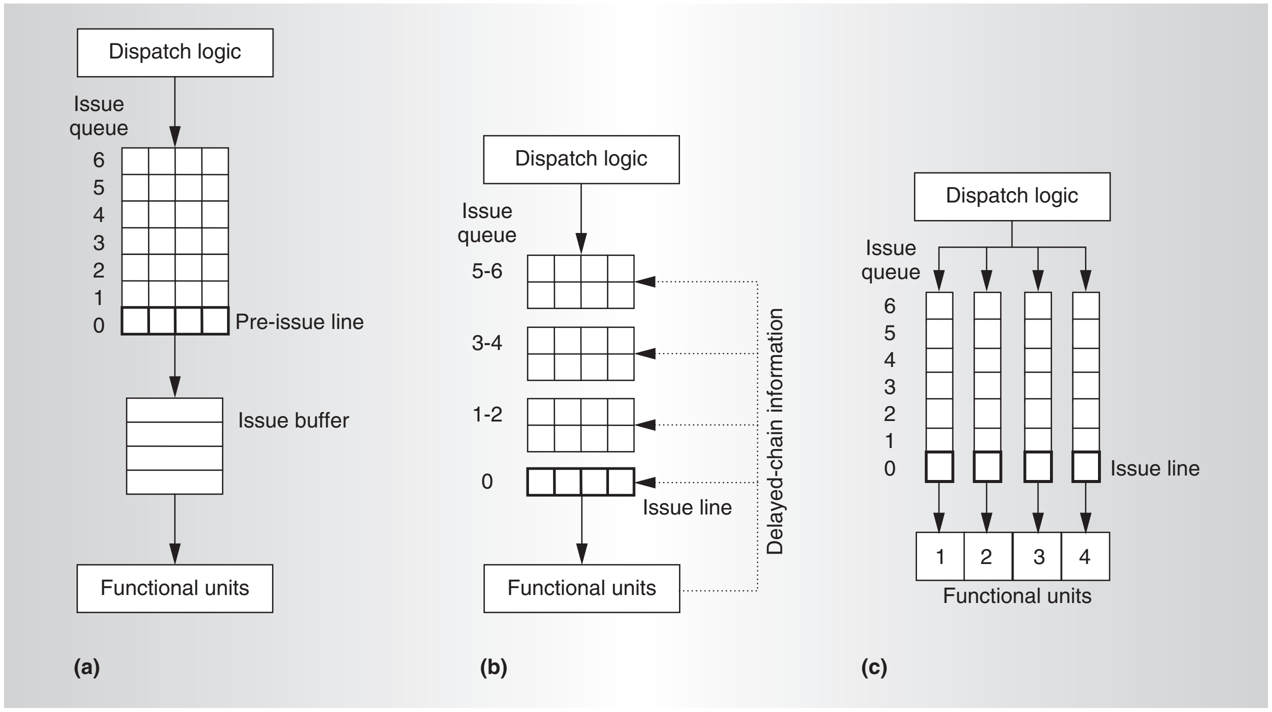


Figure 6. Code layout issue queues: prescheduling (a), segmented (b), and delayed (c) issue queues.

implementation constraints, this scheme partitions the issue queue into several segments. Each segment stores instructions that the scheme estimates can issue after a given range of cycles. Figure 6b shows this scheme.

Grossman proposes tagging each instruction at compile time with the delay of its source operands according to its producers' latencies.[22] At runtime, the instructions go into a functional-unit-specific issue queue; the dispatch logic assigns an issue slot according to each instruction's latency tag, as Figure 6c shows.

Dependence tracking

The last category of schemes reduces issue logic complexity through mechanisms that track dependences among instructions; such schemes also link producer and consumer instructions. By tracking this explicit relationship, these schemes avoid (or reduce) the associative lookup inherent in conventional issue logic. Most of these schemes exploit the fact that results generated by an instruction typically have just one consumer.[19] Thus, propagating the results only to the consumers is much more efficient than broadcasting the results to all instructions in the queue.

In these mechanisms, a direct-access RAM structure replaces the associative logic required by conventional schemes. All these mechanisms use a table to track dependences, keeping instructions in a separate structure (the dependence structure). When the instructions are ready to execute, the dependence structure forwards them to the issue logic. Figure 7 gives a block diagram of a queue for dependence-tracking schemes.

Two classes of alternatives differ as to where they keep instructions before issue: in the dependence structure[18,19,23] or in a separate structure.[24,25]

A physical-register identifier indexes the dependence structure. For the simplest schemes, each entry keeps just one consumer instruction for the corresponding register, reducing the amount of parallelism the issue logic can exploit. Several approaches attempt to relax this constraint.

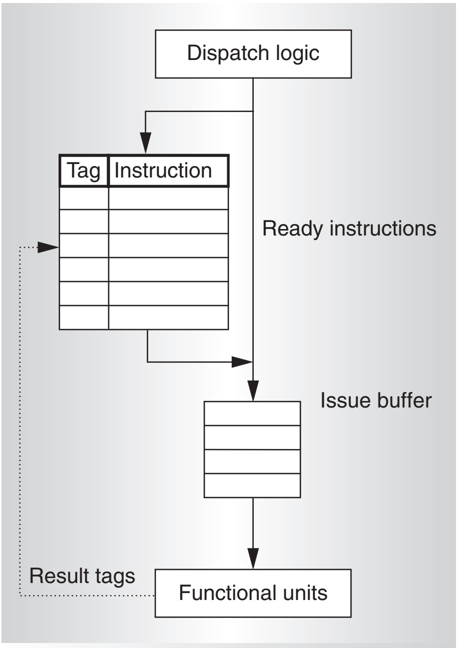


Figure 7. Basic dependence-based issue queue.

Huang, Renau, and Torrellas introduce extra bits in the instruction window that let producers have more than one consumer.[24] Because basic dependence-based schemes cannot implement direct wakeup when an instruction has more than one consumer, this proposed approach allows a limited associative search to wake up the consumers.

Önder and Gupta[23] extend the links between producer and consumer by allowing a link from consumer to consumer (assuming they have the same producer). This extra link implements the relationship between one producer and more than one consumer, preventing the issue logic from stalling when one instruction is the second consumer and limitations in the basic structure prevent its dispatch.

Canal and González introduce associativity in the dependence structure, permitting the connection of producers with more than one consumer.[18] In later work, the authors extend the mechanism with an extra associative buffer that holds instructions untrackable through a dependence-based structure.[19]

Sato, Nakamura, and Arita introduce a scoreboarding mechanism that allows the issue logic to consider for issue consumers instructions that the direct-search mechanism cannot track.[25] These instructions continuously monitor the register file for operand availability. In other words, this approach handles instructions conventionally, as though they were part of an associative buffer.

Palacharla, Jouppi, and Smith propose a somewhat different, but related, approach. This scheme distributes the issue logic into several first-in first-out (FIFO) queues.[26] The dispatch logic forwards each instruction to the FIFO queue in which the last instruction is the producer of a source operand; if no FIFO queue meets this condition, the instruction goes to an empty queue. If no empty queue is available, the dispatch stage stalls. Placing instructions in this way guarantees that the instructions in a given FIFO queue execute sequentially; thus, this scheme monitors only the youngest instruction in each queue for potential issue.

Each of the schemes discussed here has advantages and drawbacks, and thus the best solution will depend on the particular application scenario. Because future processors might require even larger instruction windows and faster clock rates, issue logic design will continue to be an interesting area of research.

Finally, leakage currents will become a significant source of energy consumption and power dissipation for the forthcoming generations of processors. Most of the proposed techniques can help reduce leakage because they shut down the unused portions of hardware. Still, techniques specifically targeted at reducing leakage could be more appropriate. MICRO

Acknowledgments

The work reported here was supported by the Spanish Ministry of Education under grant CICYT TIC2001-0995-C02-01, the Generalitat de Catalunya under grant 2001TDOC00049, and Intel Corporation. Jaume Abella and Ramon Canal would like to thank their fellow Projectistes Becaris Col·laboradors (PBCs) for their patience and precious help.

References

1. D. Sima, "Superscalar Instruction Issue," *IEEE Micro*, vol 17, no. 5, Sept.-Oct. 1997, pp. 28-39.
2. M. Butler and Y. Patt, "An Investigation of the Performance of Various Dynamic Scheduling Techniques," *Proc. 25th Int'l Symp. Microarchitecture* (Micro-25), IEEE CS Press, 1992, pp. 1-9.
3. A. Buyuktosunoglu et al., "Tradeoffs in Power-Efficient Issue Queue Design," *Proc. Int'l Conf. Low-Power Electronics and Design* (ISLPED 02), ACM Press, 2002, pp. 184-189.
4. D. Folegnani and A. González, "Energy-Effective Issue Logic," *Proc. 28th Int'l Symp. Computer Architecture* (ISCA 01), IEEE CS Press, 2001, pp. 230-239.
5. D. Albonesi, "Dynamic IPC/Clock Rate Optimization," *Proc. 25th Int'l Symp. Computer Architecture* (ISCA 98), IEEE CS Press, 1998, pp. 282-292.
6. A. Buyuktosunoglu et al., "A Circuit Level Implementation of an Adaptive Issue Queue for Power-Aware Microprocessors," *Proc. 11th Great Lakes Symp. VLSI* (GLSVLSI 01), ACM Press, 2001, pp. 73-78.
7. D. Ponomarev, G. Kucuk, and K. Ghose, "Reducing Power Requirements of Instruction Scheduling Through Dynamic

Allocation of Multiple Datapath Resources," *Proc. 33rd Int'l Symp. Microarchitecture* (Micro-33), IEEE CS Press, 2001, pp. 90-101.
8. S. Dropsho et al., "Integrating Adaptive On-Chip Storage Structures for Reduced Dynamic Power," *Proc. 11th Parallel Architectures and Compilation Techniques*, IEEE CS Press, 2002, pp. 141-152.
9. J. Abella and A. González, "Power-Aware Adaptive Issue Queue and Register File," *Proc. Int'l Conf. High-Performance Computing* (HiPC), 2003, to appear.
10. A.R. Lebeck et al., "A Large, Fast Instruction Window for Tolerating Cache Misses," *Proc. 29th Int'l Symp. Computer Architecture* (ISCA 02), IEEE CS Press, 2002, pp. 59-70.
11. T. Moreshet and R.I. Bahar, "Complexity-Effective Issue Queue Design under Load-Hit Speculation," *Proc. Workshop Complexity-Effective Design*, 2002, http://www.ece.rochester.edu/~albonesi/wced02/slides/bahar.pdf.
12. B. Fields, S. Rubin, and R. Bodík, "Focusing Processor Policies via Critical-Path Prediction," *Proc. 28th Int'l Symp. Computer Architecture* (ISCA 01), IEEE CS Press, 2001, pp. 74-85.
13. S. Srinivasan et al., "Locality vs Criticality," *Proc. 28th Int'l Symp. Computer Architecture* (ISCA 01), IEEE CS Press, 2001, pp. 132-143.
14. E. Brekelbaum et al., "Hierarchical Scheduling Windows," *Proc. 35th Int'l Symp. Microarchitecture* (Micro-35), IEEE CS Press, 2002, pp. 27-36.
15. D. Ernst and T. Austin, "Efficient Dynamic Scheduling through Tag Elimination," *Proc. 29th Int'l Symp. Computer Architecture* (ISCA 02), IEEE CS Press, 2002, pp. 37-46.
16. M. Goshima et al., "A High-Speed Dynamic Instruction Scheduling Scheme for Superscalar Processors," *Proc. 33rd Int'l Symp. Microarchitecture* (Micro-33), IEEE CS Press, 2001, pp. 225-236.
17. M.D. Brown, J. Stark, and Y.N. Patt, "Select-Free Instruction Scheduling Logic," *Proc. 34th Int'l Symp. Microarchitecture* (Micro-34), IEEE CS Press, 2001, pp. 204-213.
18. R. Canal and A. González, "A Low-Complexity Issue Logic," *Proc. ACM Int'l Conf. Supercomputing* (ICS 00), ACM Press, 2000, pp. 327-335
19. R. Canal and A. González, "Reducing the Complexity of the Issue Logic," *Proc. ACM Int'l Conf. Supercomputing* (ICS 01), ACM Press, 2001, pp. 312-320.
20. P. Michaud and A. Seznec, "Data-Flow Prescheduling for Large Instruction Windows in Out-of-Order Processors," *Proc. Int'l Symp. High-Performance Computer Architecture* (HPCA 01), IEEE CS Press, 2001, pp. 27-36.
21. S.E. Raasch, N.L. Binkert, and S.K. Reinhardt, "A Scalable Instruction Queue Design Using Dependence Chains," *Proc. 29th Int'l Symp. Computer Architecture* (ISCA 02), IEEE CS Press, 2002, pp. 318-329.
22. J.P. Grossman, "Cheap Out-of-Order Execution Using Delayed Issue," *Proc. Int'l Conf. Computer Design 2000* (ICCD 00), IEEE CS Press, 2000, pp. 549-551.
23. S. Önder and R. Gupta, "Superscalar Execution with Dynamic Data Forwarding," *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques*, IEEE CS Press, 1998, pp. 130-135.
24. M. Huang, J. Renau, and J. Torrellas, "Energy-Efficient Hybrid Wakeup Logic," *Proc. Int'l Symp. Low-Power Electronics and Design*, (ISLPED 02), ACM Press, 2002, pp. 196-201.

25. T. Sato, Y. Nakamura, and I. Arita, "Revisiting Direct Tag Search Algorithm on Superscalar Processors," *Proc. Workshop Complexity-Effective Design*, 2001, http://www.ece.rochester.edu/~albonesi/wced01/papers/tsato.ps.
26. S. Palacharla, N.P. Jouppi, and J.E. Smith, "Complexity-Effective Superscalar Processors," *Proc. 24th Int'l Symp. Computer Architecture* (ISCA 97), IEEE CS Press, 1997, pp. 206-218.

Jaume Abella is a research assistant and a PhD candidate in the Computer Architecture Department at Universitat Politècnica de Catalunya (UPC). His research interests include power-efficient issue logic and data-cache designs. Abella has a BSc and MSc in computer science from UPC.

Ramon Canal is an assistant professor and a PhD candidate in the Computer Architecture Department at UPC. His research interests include low-power and energy-efficient microprocessors. Canal has a BSc and MSc in computer science from UPC. He is a member of the Collegi Oficial d'Enginyeria en Informàtica de Catalunya.

Antonio González is a professor in the Computer Architecture Department at UPC. He also leads the Intel-UPC Barcelona Research Center, where research focuses on new microarchitecture paradigms and code-generation techniques for future microprocessors. González has a BSc, an MSc, and a PhD in computer science from UPC. He is a member of the IEEE Computer Society.

Direct questions and comments about this article to Antonio González, Dept of Computer Architecture, Universitat Politècnica de Catalunya, Cr. Jordi Girona, 1-3, Mòdul D6, 08034 Barcelona, Spain; antonio@ac.upc.es.