Processor Implementations Using Queues

Michael K. Milligan

Harvey G. Cragon
University of Texas at Austin

For several decades, designers have used queues to resolve two processor-memory interface problems—long latency and low bandwidth. Here, we discuss the evolution of instruction and branch target queues. We also explore their use to support variable-length instructions and reduce misalignment problems. From the 1956 IBM 7030 (Stretch) to today's PowerPC, we present queue configurations and prefetch strategies along with the design decisions that led to their final architectures.

Attempting to improve overall computing performance, designers often focus their efforts on increasing the rate of information transfer between memory and the processor. The primary problem they face is that the processor's maximum transfer rate, or maximum bandwidth, is as much as 10 times higher than the memory's maximum rate. As a result, the slower memory subsystem limits the overall transfer rate. To make matters worse, this rate mismatch problem intensifies as we introduce more powerful processors, which require more and more information per unit time. Memory devices have not been able to keep up with the increasing requirements of processors, and the gap between them continues to widen.

To help solve this mismatch problem, designers introduce interface devices that allow the processor and memory to run at their respective maximum transfer rates. In effect, these devices allow the processor to operate as if the memory matched its transfer rate at all times. The two basic parameters associated with the interface are latency and bandwidth. We describe latency as the total time the memory requires to satisfy a request from the processor (see Figure 1). Since the processor is much faster than the main memory, without an interface device, latency often causes the processor to go into one or more wait states. This results in wasted clock cycles.

We define bandwidth as the rate of information transfer between the processor and memory that supports the required processing rate. As we noted, the slower memory usually limits the bandwidth. Ideally, designers would like a memory system that provides a latency of one clock cycle and a bandwidth high enough to support the processor's maximum data rate. Designers introduced cache memory as part of the memory hierarchy to help solve the latency problem, but it alone is not a solution.

The interface techniques and devices that support low latency and high bandwidth vary; we classify them as either buffers or queues. Much has been written about these devices, and as a result, definitions and descriptions of the two overlap considerably. In this article, we consider a buffer to be a small memory subsystem that holds previously used code or data for reuse, providing "look-behind" memory. A buffer is useful, for example, for storing code for a repeatedly executed loop or for holding data waiting to be written to memory. A queue, on the other hand, holds code or data that has been prefetched, thereby providing "look-ahead" memory. It anticipates the processor's needs from memory and stores that information in advance, reducing the memory's effective access time. This article examines the use of queues as interface devices—their operation, properties, design considerations, and historical use.

Originally published in *IEEE Micro*, Aug. 1995, pp. 58–66.

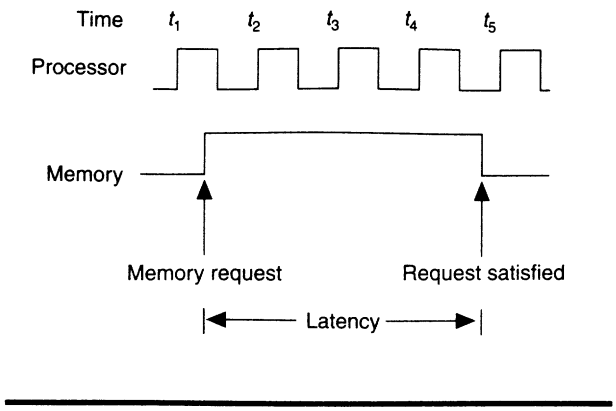

Figure 1. Memory latency.

Queues

Webster's Dictionary of Computer Terms defines a queue as "a linear list for which all insertions are made at one end of the list; all deletions (and usually all accesses) are made at the other end." Thus, we can describe the queue as a first-in, first-out (FIFO) device: The first data entered into the queue is also the first data retrieved.

Designers use queues as matching devices to interface two subsystems with different duty cycles. Queue applications include processor-memory, processor-processor (each running at a different clock speed), and controller-peripheral interfaces. This article addresses their use in processor to memory interfaces.

To describe a queue and its state, we use six parameters: width, depth, size, fill rate, empty rate, and percent full (or empty). Queue width w is the number of bits of each entry, and is typically one or more bytes. Queue depth d is the number of individual entries in the queue, normally one to several bytes. Together, width and depth represent the capacity of the queue, which we refer to as queue size. We express fill rate λ_f and empty rate λ_e in bytes per second. They represent the rates at which information is written to and read from the queue. These rates are in effect the bandwidths into and out of the queue. Percent full, α, represents the percentage of queue entries that are valid and can be described using the following relationship:

$$\alpha = (\text{number of valid entries}/d)100$$

For a queue to be helpful in supporting processor operations, α must be greater than zero. This implies the queue contains useful information, that is, code and data are available to the processor when needed. Many times during program execution, the fill rate exceeds the rate at which the queue empties ($\lambda_f > \lambda_e$). This may occur following periods when the queue becomes empty, such as at initial program load and immediately after taken branches. It may also occur after the queue's contents are rapidly depleted (instances when instructions are executed more rapidly than they can be fetched). In these instances, λ_f is greater than λ_e, so the queue builds up a reserve of instructions. However, on average in the steady state, λ_f is equal to λ_e.

Designers use queues in conjunction with interleaved memory to address the bandwidth problem in the processor-memory interface. An interleaved memory organizes memory into a series of banks, each accessed on a different clock cycle. When addresses are distributed sequentially over the memory banks, the queue can access information on each clock cycle, thus increasing bandwidth, as shown in Figure 2.

Queues have a number of interesting properties that make them useful in supporting processor functions. One is their ability to "hide" cache misses. Cache misses occur whenever the processor does not find the requested information in the cache. This results in delays, since the processor must access the memory's slower, lower levels. However, look-ahead memory allows queues and their associated prefetch logic to hide cache misses by prefetching information from memory that would otherwise not be in the cache. In effect, the queue's prefetch logic anticipates the cache miss and retrieves the needed information in advance, so the processor never "sees" the cache miss.

Originally published in *IEEE Micro*, Aug. 1995, pp. 58–66.

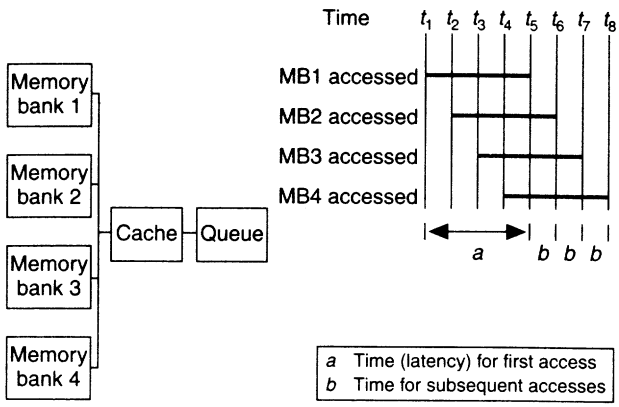

Figure 2. Four-way, interleaved memory.

Another useful property is the queue's ability to supply, in one clock cycle, the proper number of bytes required for variable-length instructions. With a queue in its architecture, the processor can read as many bytes as required in one clock cycle rather than initiating successive reads from memory. The processor requires only two fetches—the first to decode the opcode and the second to fetch the remaining required specifiers/operands.

Variable-length instructions exist in several instruction sets, including those in the Intel 80x86, the IBM 360/91, and VAX products. Figure 3 depicts a fetch of a variable-length instruction. In Figure 3a, the processor's next instruction is only one-byte long, so the queue forwards its first entry to the processor. In Figure 3b, the queue forwards four bytes of the instruction simultaneously. When there are several prefetched instructions stored in the queue, the processor can read one instruction per clock cycle. The processor simply retrieves as many bytes as needed, usually without regard to the instruction's length. Thus, the processor avoids additional memory access cycles when fetching multiple bytes of a long instruction.

Finally, queues eliminate the misalignment problem that might exist if the processor were to fetch directly from the cache or memory. Misalignment occurs when the stored code or data does not begin on the same byte boundaries the processor accesses. For example, the cache may split a long, multiple-byte instruction, storing it in more than one addressable block (Figure 4a). As a result, the processor would require more than one access to read this instruction. However, once in the queue, the instruction is aligned, and the processor requires only one access (Figure 4b).

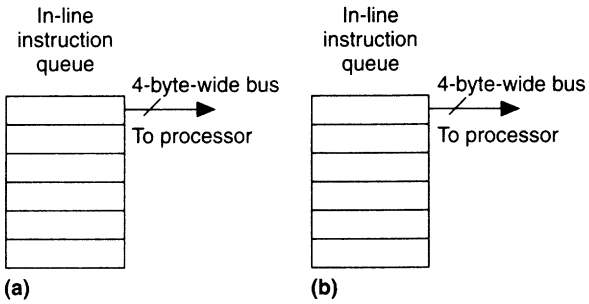

Figure 3. Fetch of a variable-length instruction: one-byte instruction (a) and four-byte instruction (b).

Originally published in *IEEE Micro*, Aug. 1995, pp. 58–66.

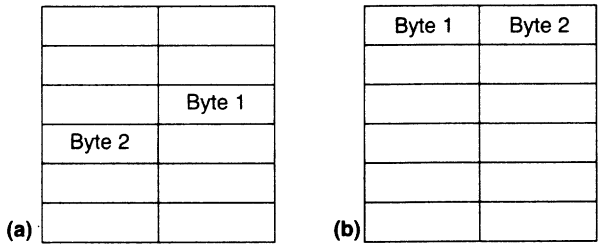

Figure 4. Instruction alignment problem: misaligned instruction in cache (a) and aligned instruction in queue (b).

Designers have used queues in numerous architectures to support various processor functions and to hold instructions (both decoded and nondecoded), branch target instructions, and data during reads and writes (Figure 5). We typically classify queues by the type of information they contain—instructions (program code) or data (operands). We can further classify them as shown Figure 6.

Instruction queues hold instructions before their execution, while data queues hold data before execution (during a load) and data buffers after (during a store). We can further categorize instruction queues as in-line instruction queues and branch target instruction queues. In-line queues load instructions from the sequencing (in-line) instruction stream directly from memory or the cache. These exist in two types as well: queues that hold decoded instructions and queues that hold nondecoded instructions. Branch target queues also store instructions, but we classify them separately since the processor executes these instructions conditionally.

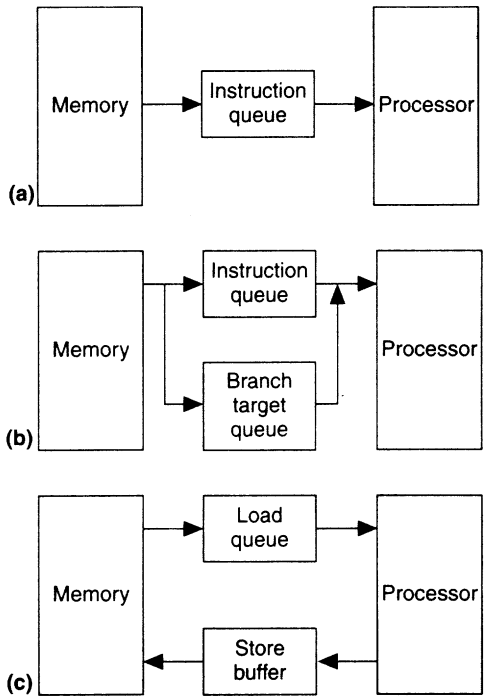

Figure 5. Queue configurations: in-line instruction (a), branch target instruction (b), and data (c).

Originally published in *IEEE Micro*, Aug. 1995, pp. 58–66.

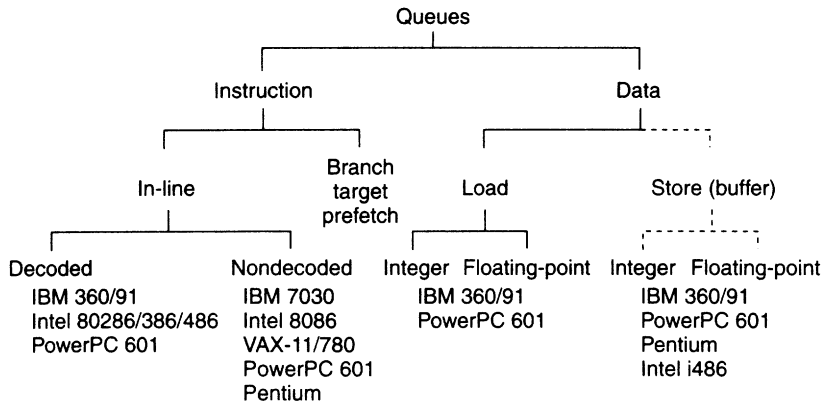

Figure 6. Queue classification.

In-line instruction queues

A decoded, in-line queue is actually a series of two queues—one that holds instructions loaded directly from memory (or cache), and a second that holds the output of the instruction decoder (a decoded instruction). This two-level queue, shown in Figure 7, provides the control unit access to decoded instructions awaiting execution by the processor. This allows limited preprocessing, which is useful for predicting conditional branches. Branch prediction is extremely important in improving overall processor performance, and all of the newest microprocessor designs incorporate this function. Conversely, nondecoded queues have no second level, and controller preprocessing is not possible.

Before examining specific architectural examples, we must address two design issues involving previously mentioned terms—queue size and prefetch logic. Queue size is not a straightforward design choice, since performance deteriorates if the queue depth is too short or too long.[1] For example, one may be tempted to make the depth of an in-line instruction queue as large as possible to ensure it won't ever run out of instructions. However, several issues limit its practical size. Keeping the queue full (or nearly full) requires memory accesses. As the depth of the queue increases, the resulting address and data bus traffic also increases. This could force the processor to wait for bus accesses that might otherwise be available if the queue weren't reading from memory.

Conditional branches also limit queue size. Conditional branches disrupt the smooth flow of instructions by executing portions of code not within the current range of the queue. When the processor encounters a conditional branch, it flushes the in-line queue and reloads it with instructions from the taken branch instruction stream. If conditional branches occur frequently, the resulting bus traffic (due to reloading of the queue) may also be heavy.

When considered together, large queue sizes and conditional branches have an even more pronounced effect. The increase in memory bus traffic caused by the deeper queue will likely contain many instructions the processor will never execute (due to taken branches).

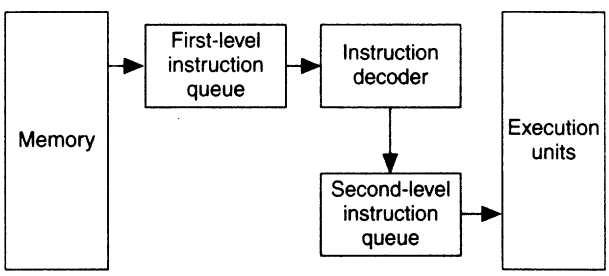

Figure 7. Decoded instruction queue used in the Intel 80386.

Originally published in *IEEE Micro*, Aug. 1995, pp. 58–66.

An equally important design issue is choosing a prefetch strategy. The prefetch strategy is the logic that determines when and how much information to prefetch from the cache or memory. The designer chooses a strategy based on the type and size of the queue. In-line instruction queue prefetches are fairly straightforward; the queue initiates them depending on its current state. For example, when a given fraction of the instruction queue is empty (for example, 1/8, 1/4, 1/2) the prefetch logic initiates a prefetch, bringing one or more bytes into the queue. Branch target instruction and data queue prefetch strategies are not as simple, and we will discuss them later in some detail.

Example architectures. Next, we examine examples of instruction queues supporting several processor implementations—from the IBM 7030 to the PowerPC and the Pentium.

IBM 7030 (Stretch). The first known implementation of an instruction queue was in the IBM 7030 computer, introduced in 1956. IBM started Project Stretch to achieve a hundred-fold increase in speed over the IBM 704.[2] The 7030 used a nondecoded in-line instruction queue to take advantage of idle bus cycles and fetch instructions from its four-way interleaved memory. During development, designers termed the queue and associated logic the "look-ahead unit," since the device was designed to look ahead in the code stream.

Developers determined the queue size and measured its impact on processor performance by varying the queue depth (levels of look ahead) and then running various programs. The designers achieved the most significant gain in performance with a queue depth of two. Thereafter, increasing the queue depth resulted in only marginal performance increases compared to queue size. Thus, the designers chose two levels of look ahead as the queue depth. The prefetch strategy required the processor to initiate a fetch when one instruction (4 bytes) was vacant in the queue.[3]

IBM 360/91. Debuted in 1964, this machine introduced two new concepts: cache memory and decoded (two-level) instruction queues. Although the cache memory improved average execution times, cache misses still limited overall performance, so designers used instruction queues to hide many of these misses. As shown in Figure 8, the first level of the two-level, in-line instruction queue held instructions going to the instruction unit (decoder). The second level contained two individual queues—each one holding decoded instructions for the fixed- and floating-point units. The first-level queue held six instructions, while the second level held six floating-point and eight fixed-point instructions. Designers decided on these figures after analyzing the machine on a typical program, which examined anticipated delays, instruction mix, and relative time and frequency of execution bottlenecks (that is, branches).[4] The 360/91 fetch strategy required the processor to initiate a fetch whenever space for two instructions was available in the queue.

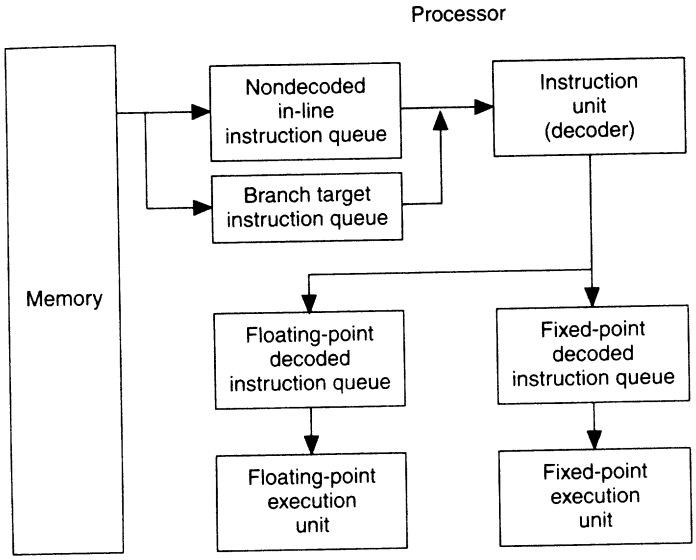

Figure 8. IBM 360/91 instruction queue organization.

Originally published in *IEEE Micro*, Aug. 1995, pp. 58–66.

VAX-11/780. In 1977, Digital introduced the VAX-11/780, incorporating into its design a nondecoded, in-line instruction queue 1-byte wide and 8 deep. The VAX instruction set contains instructions of various lengths, each instruction 1- to 50-bytes long.[5] However, the weighted average instruction length is 3.8 bytes.[6] Since the instruction queue holds 8 bytes, it contains approximately two instructions when full. The processor removes instructions from the queue 1 byte at a time, and when there is space available for a 4-byte block load from the cache, it initiates a prefetch.

Intel 80x86. The Intel family of 80x86 microprocessors uses both decoded and nondecoded, in-line instruction queues in their microprocessor models (see Figure 7). In the 80x86 instruction set, some instructions can execute faster than they can be fetched (such as ADD IMMEDIATE), and some more slowly (such as MULTIPLY). During the time needed for slow instructions, the instruction queue stockpiles enough bytes for intervening fast instructions to run at full speed without putting the processor into wait states.

During 8086 development, designers determined the queue depth through a series of simulator programs. Designers loaded the programs with typical instruction sequences and executed them. Results showed a diminished return at a queue size of approximately 6 bytes—about two instructions (Figure 9).[1] The queue's instruction prefetch strategy is to fetch 2 bytes when space for them becomes available in the queue. If the queue is completely empty (following a branch, for example), the first byte into the queue becomes immediately available to the processor and is not stored.

The 80286 and 80386 microprocessors use a similar strategy, although queue sizes differ (see Table 1). The i486 microprocessor has a similar design to earlier members of its family, but includes an on-chip cache in conjunction with the decoded, in-line instruction queue. The successor to the i486, the Pentium microprocessor, is unique in the Intel family in that it incorporates two independent, line-size (32-byte), nondecoded, in-line instruction queues (Figure 10). Only one instruction queue actively prefetches from the instruction cache at any given time, while the other is reserved for use during branches (discussed later).

PowerPC 601. Like the Pentium, the IBM/Apple/Motorola PowerPC 601 is a superscalar processor, but its queue structure is entirely different (Figure 11). Its nondecoded, in-line instruction queue holds as many as eight instructions (equivalent to a cache block) and fills from the cache during a single cycle. The 601's instruction queue is unique in that it does not operate on a strict FIFO principle. During each cycle, the dispatch logic decodes the bottom four entries of the queue and dispatches up to three instructions, one each to the fixed-point, floating-point, and branch processing units. The positions from which the queue can dispatch instructions vary for each functional unit. The queue can dispatch branch instructions and most floating-point instructions from any of the four entries, while it always dispatches fixed-point instructions from the bottom queue location. A tagging and counting mechanism preserves the program order for the out-of-order completion of instructions.[7]

Table 1 lists characteristics of several instruction queues used in past and current architectures.

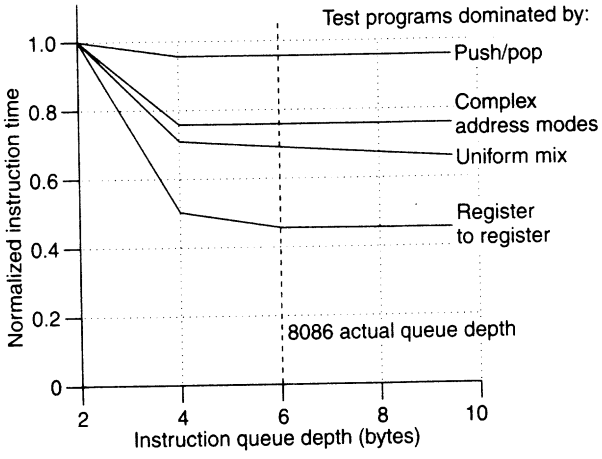

Figure 9. 8086 instruction queue depth versus performance.[1] (© IEEE, 1979.)

Table 1. In-line instruction queue characteristics.

Computer	Queue width (bytes)	Queue depth (bytes)	Cache/memory data path width (bytes)
IBM 7030	4	2	4
IBM 360/91	9*	8	8
IBM 370/195	8	3	8
IBM RT	4	4	4
VAX-11/780	1	8	1-4
Intel 8086	2	3	2
Intel 80286	2	3	2
Intel 80386	2	8	4
Intel i486	2	16	4
AMD 29000	4	4	4
Pentium **	32	2	32
Power PC 601	4	8	32
Power PC 603	4	6	32

* Includes parity

** Double queues, each 32-bytes wide, 2-bytes deep

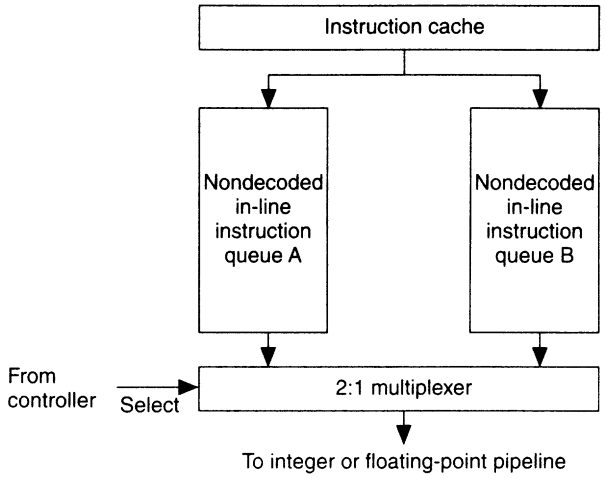

Figure 10. Pentium instruction queue configuration.

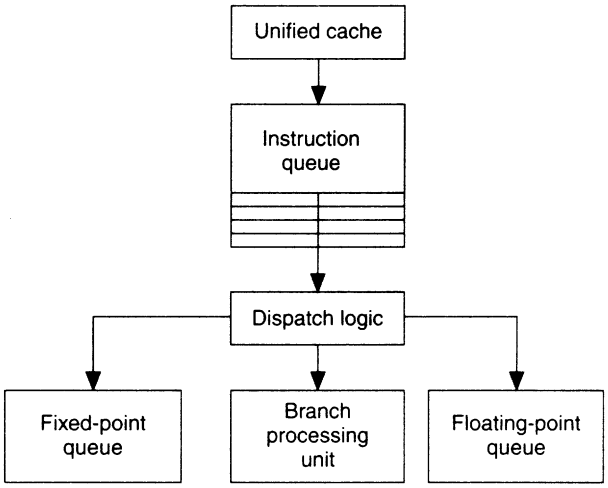

Figure 11. PowerPC 601 instruction queue organization.

Branch target instruction queues

Like in-line instruction queues, branch target instruction queues hold prefetched instructions (Figure 5b). However, they differ from the queues described earlier in that they store prefetched instructions that the processor executes conditionally. We consider them secondary instruction queues.

A conditional branch means that instructions may continue to come from the in-line instruction stream, or they may come from elsewhere in memory, depending on the outcome of a condition (test). If the branch is taken and no branch target queue exists, the processor must flush the in-line instruction queue and reload it with branch target code instructions. This results in several cycles of delay. The branch target instruction queue's prediction logic helps avoid this problem by predicting the outcome of conditional branches before the execution unit of the processor tests the condition. Once the logic makes a prediction, the branch target queue prefetches the appropriate stream of code.

In effect, this system employs two parallel instruction queues—a primary queue for in-line instruction prefetches and a secondary queue for branch target instruction prefetches. This technique saves clock cycles since both the branch taken instructions (stored in the branch target queue) and branch not-taken instructions (stored in the in-line instruction queue) are readily available to the processor.[8]

Associated with the branch target instruction queue is logic that predicts whether or not branches are likely to be taken. If the logic indicates the branch will be taken, the branch target queue loads instructions from the branch target code stream; otherwise it takes no action. The branch prediction logic can be either static or dynamic. Static prediction logic is hardwired into the system; the logic makes the same prediction under the same circumstances. For example, the PowerPC 601 always predicts that backward branches will be taken and forward branches will not be taken.

Dynamic branch prediction resolves the "to branch or not to branch" question by using information available to the processor at that time. Examples of such information include the magnitude or direction of the branch and whether or not the branch was taken the last time it was executed. The Pentium uses a form of dynamic prediction logic based on recent branch history.

Example architectures. Here, we show examples of processor architectures that employ branch target queues.

IBM 360/91. This machine was the first known to implement a branch target instruction queue in its design (Figure 8). When the processor decoded an instruction and detected a branch, it fetched the first four instructions of the branch target path and placed them into the branch target queue. Simultaneously, the processor determined the outcome of the branch decision. Depending on the outcome, the processor would fetch instructions from either the in-line instruction or branch target queue. If the processor reached the decision to take the branch before the branch target queue loaded all four instructions, additional optimizing hardware routed the target fetch instructions around the queue and directly into the instruction unit for decoding and execution. This procedure saved precious clock cycles.[4]

Intel 80x86. Intel did not use branch target instruction queues or branch prediction logic in their micro-

processor designs until the introduction of the Pentium in 1993. In previous members of the 80x86 family, the processor continued to fetch in-line instructions until the execution unit determined the outcome of a branch. If the branch was to be taken, the processor flushed the in-line instruction queue and reloaded it with branch target instructions, resulting in several cycles of delay. Over a period of time, it was determined that 15 to 20 percent of instructions in typical programs were branches, representing an obvious area of improvement by the Pentium processor.[9]

The Pentium incorporates logic that predicts whether a branch will be taken based on the individual branch's previous history. Instead of a dedicated branch target instruction queue, the Pentium employs two devices—an alternate in-line instruction queue and a branch target buffer (actually an associative memory). When the processor first takes a branch instruction, it allocates an entry in the buffer to associate the branch instruction's address with its destination address and to initialize the history used in the prediction algorithm. As the processor decodes instructions, it searches the buffer to see if it contains an entry for a corresponding branch instruction. When there is a match, the processor uses the history to determine if the branch should be taken. If so, it uses the target address previously stored in the branch target buffer to begin fetching and decoding instructions from the target path.[9] The processor enables the in-line instruction queue not being used (for example, Queue B in Figure 10), which begins to prefetch. If the branch is mispredicted, the processor flushes the instruction queue, and instruction prefetching starts over.

PowerPC 601. Unlike the Pentium, the PowerPC 601 employs a branch processing unit to assist in resolving branch predictions (Figure 11). This unit scans the bottom half of the instruction queue for conditional branch instructions and evaluates them. If the displacement of the target address is negative (a backward branch), the unit predicts the branch will be taken. If the displacement is positive (a forward branch), it predicts the branch won't be taken. (As an aid for compiler-directed prediction, a bit in the branch instruction opcode allows the programmer to reverse prediction direction.[7]) When the branch processing unit predicts a conditional branch instruction will be taken, the 601 fetches instructions from the branch target stream until the conditional branch is resolved. At that time, either instructions continue to come from the branch target code, or, if the branch was mispredicted, prefetching resumes from the in-line instruction sequence. Table 2 lists characteristics of selected branch target queues.

Data queues and buffers

Designers use data queues to smooth the flow of data between the slower memory subsystem and the faster processor (Figure 5c). Load queues hold data (operands) coming from memory during read operations, while the processor uses buffers to perform stores (writes). These devices provide locations where the main memory can dump data for the processor, and to which the processor can write. They isolate the processor from the memory and provide an effective access time of one clock cycle for both reads and writes.[4] As with instruction queues, designers must consider size and prefetch logic. It is fairly straightforward to determine queue size based on the number of expected data transactions with memory over a period of time. Since only approximately 22 percent of all memory references involve data reads, and 7 percent involve data writes,[10] data queue sizes tend to be smaller than their instruction counterparts. They range from a few to several bytes (see Table 3). However, data prefetch strategies are not as straightforward as queue size selection.

Table 2. Branch target instruction queue characteristics.

Computer	Queue width (bytes)	Queue depth (bytes)
IBM 360/91	9	2
Pentium*	32	2
PowerPC 601/603**	N.A.	N.A.

* Employs both branch target buffer and an alternate in-line target instruction queue

** Incorporates branch prediction unit

Originally published in *IEEE Micro*, Aug. 1995, pp. 58–66.

Table 3. Data queue/buffer characteristics.

	Load queue (bytes)		Store buffer (bytes)	
Computer	Width	Depth	Width	Depth
IBM 7030	8	2	8	4
IBM 360/91	8	6	8	3
IBM 370/195	8	6	8	3
VAX-11/780	1	8	4	1
Intel i486	None	None	4	4
Pentium	None	None	8	1
PowerPC				
601/603	8	1	16	3

Prefetching data from memory is the consequence of a load instruction decoded early in the instruction decode cycle. The processor reads data from main memory and places it in a load queue. While this provides look-ahead information and reduces access time, it also presents potential problems. For example, the address of the prefetched data could change after the processor initiates the prefetch. This can occur when an instruction modifies a register that another instruction will use later to form the data address. This address generation interlock (AGI) problem can result in the processor's using erroneous data during program execution.

In addition to the AGI problem associated with load queues, store buffers encounter problems of their own. They perform no prefetching, but are susceptible to true (or flow) dependencies. This problem occurs when one instruction is reading data from memory, while the result (and logically correct) value of a previous instruction is in the store buffer waiting to be written to the same memory location. As a result, the subsequent instruction could use an incorrect value. In this case, the processor should search the store buffer for any address matches to the memory location currently being accessed.

Example architectures. The IBM 360/91 uses both load queues and store buffers. It has one store buffer and two load queues—one for the fixed-point execution unit and one for the floating-point execution unit. Early members of the Intel 80x86 family had neither load queues nor store buffers for data. Later processors, including the i486 and Pentium, include store buffers in their designs, but no load queues. The PowerPC 601/603 contains both load queues and store buffers that isolate operations between main memory and the cache. Table 3 lists characteristics of several data queues.

Conclusion

The use of instruction and data queues to help overcome the memory interface bottleneck has a long history in the computer architecture community. Their low latency and high bandwidth features make them very attractive to designers trying to improve overall system performance. The queue's many advantages make it flexible for a number of applications. With processing rates increasing much faster than memory access times, queues will continue to be an integral part of high-performance computer systems. We plan to use queues and prefetching techniques to support current research on real-time architectures and memory systems.

References

1. J. McKevitt and J. Bayliss, "New Options from Big Chips," *IEEE Spectrum*, Vol. 16, No. 3, Mar. 1979, pp. 28-34.
2. W. Bucholz, *Planning a Computer System, Project Stretch*, McGraw-Hill, New York, 1959.
3. E. Bloch, "The Engineering Design of the Stretch Computer," *Proc. Eastern Joint Computer Conf.* National Joint Computer Committee, 1959, pp. 48-59.
4. D.W. Anderson, F.K. Sparacio and R.M. Tomasulo, "The IBM System/360 Model 91: Machine Philosophy and Instruction-Handling," *IBM J.*, Vol. 11, No. 1, Jan.1967, pp. 8-24.
5. R.H. Eckhouse, Jr. and H.M. Levy, *Computer Programming and Architecture, the VAX-11*, Digital Press, Bedford, Mass., 1980.
6. J.S. Emer and D.W. Clark, "A Characterization of Processor Performance in the VAX-11/780," *Proc. Int'l Symp. Computer Architecture*, IEEE, Piscataway, N.J. 1984, pp. 301-310.
7. M.C. Becker et al., "The PowerPC 601 Microprocessor," *IEEE Micro*, Vol. 13, No. 7, Oct. 1993, pp. 54-67.

Originally published in *IEEE Micro*, Aug. 1995, pp. 58–66.

8. I. Flores, ";Lookahead Control in the IBM 370 Model 165," *Computer*, Vol. 7, No. 11, Nov. 1974, pp. 24-38.
9. D. Alpert and D. Avnon, "Architecture of the Pentium Microprocessor," *IEEE Micro*, Vol. 13, No. 3, June 1993, pp. 11-21.
10. D.B. Fite, T. Fossum, and D. Manley, "Design Strategy for the VAX 9000 System," *Digital Technical J.*, Vol. 2, No. 4, Fall 1990, pp. 13-24.

Michael K. Milligan is a captain in the US Air Force and a PhD student in electrical engineering at the University of Texas at Austin. He formerly served as an instructor in the Electrical Engineering Department at the US Air Force Academy and as a communications engineer at Air Force Space Command, both in Colorado Springs, Colorado. In addition, he served as a project engineer at the Air Force Electronic Systems Division in Bedford, Massachusetts. His current research interests include computer architecture design methodology and computer architectures to support real-time computing.

Milligan holds a BSEE from Michigan State University, an MSEE from the University of Massachusetts, Lowell, and an MBA degree from Western New England College. He is a member of the IEEE Computer Society and the ACM.

Harvey G. Cragon has held the Ernest Cockrell, Jr., Centennial Chair in Engineering at the University of Texas at Austin since 1984. Previously, he was employed at Texas Instruments, Inc., for 25 years, where he designed and constructed the first integrated circuit computer, the first TTL computer, and a number of other computers and microprocessors. His current interests are in computer architecture design methodology and high-speed computers. He is a member of the National Academy of Engineering, a Fellow of the IEEE and ACM, and recipient of the IEEE Emanuel R. Piore Award in 1984 and the ACM/IEEE Eckert-Mauchly Award in 1986. He is also a member of the IEEE Computer Society and a former trustee of the Computer Museum in Boston.

Direct questions concerning this article to Michael K. Milligan, University of Texas at Austin, Dept. of Electrical and Computer Engineering, Engineering Sciences Building, Austin, TX 78712-1084; milligan@uts.cc.utexas.edu.

Originally published in *IEEE Micro*, Aug. 1995, pp. 58–66.