Draft Chapter 3 from ***Fundamentals of Superscalar Processor Design***
by John P. Shen, to be published by McGraw-Hill, 2000.

CHAPTER 3 Superscalar Processors

The decade of the 1980's saw the popularization of instruction pipelining in microprocessor design. In the 1990's the dominant theme has become the design of "superscalar" microprocessors. Superscalar machines go beyond just a single instruction pipeline. They incorporate multiple functional units to achieve greater concurrent processing of instructions and higher instruction execution throughput. Similar to pipelining, the use of multiple functional units was introduced in the early 1960's in designing high-end mainframes. For example, the CDC 6600 incorporated both pipelining and the use of multiple functional units [Thornton 1964]. Another foundational attribute of superscalar processors is the ability to execute instructions in an order different from that specified by the original program. The sequential ordering of instructions in standard programs implies some unnecessary precedences between the instructions. The capability of executing instructions out of program order relieves this sequential imposition and allows more parallel processing of instructions without requiring modification of the original program. Such dynamic form of instruction execution was also introduced in the 1960's. The classic example is the Tomasulo's algorithm implemented in the floating-point unit of the IBM 360/91 [Tomasulo 1967]. While the foundations for superscalar processors were laid over 30 years ago, many new and much more aggressive techniques have emerged since then. This chapter attempts to codify the body of knowledge on superscalar processor design in a systematic fashion. We first focus on issues related to the pipeline organization of superscalar machines in Section 3.1. The techniques that address the dynamic interaction between the superscalar machine and the instructions being processed are presented next in Section 3.2. An in-depth microarchitecture description and performance analysis of a recently announced high-end superscalar processor is included in Section 3.3. The art of superscalar processor design has been advancing rapidly in recent years and there is no sign that this advancement is slowing.

3.1 SUPERSCALAR PIPELINE ORGANIZATION

Pipelining is an implementation technique for increasing the throughput of a processor. Since it is a technique implemented below the DSI (dynamic/static interface), it does not require special effort on the part of the user. Hence speedup can be obtained for existing sequential programs without any software modifications. This approach of providing performance enhancement while maintaining code compatibility is extremely attractive. In fact, this very approach is the primary reason for the current dominance of the microprocessor market by Intel since it introduced the pipelined i486 microprocessor, which is code compatible with previous generations of (non-pipelined) Intel microprocessors. While pipelining has proven to be an extremely effective microarchitecture technique, the type of *scalar pipelines* presented in the previous chapter have a number of shortcomings or limitations. Given the never-ending push for higher performance, these limitations must be overcome in order to continue to provide further speedup for existing programs. The solution is *superscalar pipelines* that are able to achieve performance levels beyond that which is possible with just scalar pipelines.

3.1.1 Limitations of Scalar Pipelines

Scalar pipelines are characterized by a single instruction pipeline of k stages. All instructions, regardless of instruction types, traverse through the same set or all of the pipeline stages. At most one instruction can be resident in each pipeline stage at any one time, and that instructions advance through the pipeline stages in a lock-step fashion. Except for the pipeline stages that are stalled, each instruction stays in each pipeline stage for exactly one cycle and advances to the next stage in the next cycle. Such rigid scalar pipelines have three fundamental limitations that are listed below and elaborated in the following three subsections.

1. The maximum throughput for a scalar pipeline is bounded by one instruction per cycle.
2. The unification of all instruction types into one pipeline can yield an inefficient design.
3. The stalling of a lock-step or rigid scalar pipeline induces unnecessary pipeline bubbles.

3.1.1.1 Upper Bound on Scalar Pipeline Throughput

As stated in the previous chapter, processor performance can be measured in terms of the time it takes to execute a program. The program execution time is a product of three components, namely instruction count, CPI, and cycle time, as shown in the following equation.

$$Performance = \frac{Time}{Program} = \frac{Instructions}{Program} \times \frac{Cycles}{Instruction} \times \frac{Time}{Cycle} \qquad \text{(EQ 1)}$$

A higher degree of pipelining can potentially increase the throughput of a scalar pipeline relative to a nonpipelined or less pipelined machine due primarily to the potential reduction of the cycle time of the machine. However, as shown in the previous chapter there is a point of diminishing return due to the hardware overhead of pipelining. Furthermore, a deeper pipeline can potentially incur higher penalties, in terms of the number of penalty cycles, for dealing with inter-instruction dependences. The additional CPI (average cycles per instruction) overhead due to this higher penalty can possibly eradicate the benefit due to the reduction of cycle time.

Regardless of the actual cycle time, a scalar pipeline can only initiate the processing of at most one instruction in every machine cycle. Essentially, the IPC (average instructions per cycle) for a scalar pipeline is fundamentally bounded by one. To get more instruction throughput, especially when deeper pipelining is no longer cost effective, the ability to initiate more than one instruction in every machine cycle is necessary. To achieve an IPC greater than one, a pipelined processor must be able to initiate the processing of more than one instruction in every machine cycle. This will require increasing the width of the pipeline to facilitate having more than one instruction resident in each pipeline stage at any one time. We identify such pipelines as *parallel pipelines*.

3.1.1.2 Inefficient Unification into Single Pipeline

Recall that the second idealized assumption of pipelining is that all the repeated computations to be processed by the pipeline are identical. For instruction pipelines, this is clearly not true. There are different instruction types that require different sets of subcomputations. In unifying these different requirements into one pipeline, difficulties and/or inefficiencies can result. Looking at the unification of different instruction types into the TYP pipeline in the previous chapter, it can be observed that in the earlier pipeline stages (such as IF, ID, and RD stages) there is significant uniformity. However, in the execution stages (such as ALU and MEM stages) there is substantial diversity. In fact, in the TYP example, we have ignored floating-point instructions on purpose due to the difficulty of unifying them with the other instruction types. It is for this reason that at one point in time during the "RISC revolution," that floating-point instructions were categorized as inherently CISC and considered to be violating RISC principles.

Certain instruction types make their unification into a single pipeline quite difficult. These include floating-point instructions and certain fixed-point instructions (such as multiply and divide instructions) that require multiple execution cycles. Instructions that require long and possibly variable latencies are difficult to unify with simple instructions that require only a single cycle latency. As the disparity between CPU and memory speeds continue to widen, the latency (in terms of number of machine cycles) of memory instructions will continue to increase. Other than latency differences, the hardware resources required to support the execution of these different instruction types are also quite different. With the continued push for faster hardware, more specialized execution units customized for specific instruction types will be required. This will also contribute towards the need for more diversity in the execution stages of the instruction pipeline.

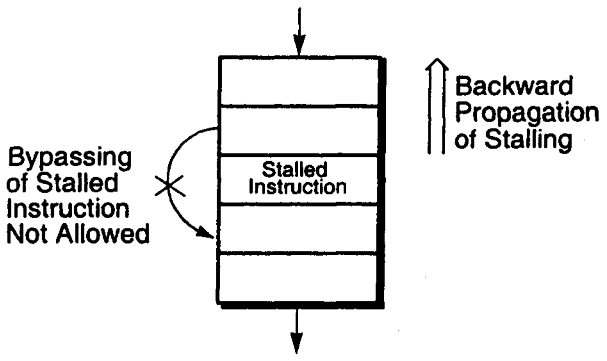
Consequently, the forced unification of all of the instruction types into a single pipeline becomes either impossible or extremely inefficient for future high-performance processors. For parallel pipelines there is strong motivation not to unify all of the execution hardware into one pipeline, but to implement multiple different execution units or sub-pipelines in the execution portion of parallel pipelines. We called such parallel pipelines *diversified pipelines*.

3.1.1.3 Performance Lost due to Rigid Pipeline

Scalar pipelines are rigid in the sense that instructions advance through the pipeline stages in a lock-step fashion. Instructions enter a scalar pipeline according to program order, i.e. in order.

When there is no stalls in the pipeline, all the instructions in the pipeline stages advance synchronously and the program order of instructions is maintained. When an instruction is stalled in a pipeline stage due to its dependence on a leading instruction, that instruction is held in the stalled pipeline stage while all leading instructions are allowed to proceed down the pipeline stages. Due to the rigid nature of a scalar pipeline, if a dependent instruction is stalled in pipeline stage i, then all earlier stages, i.e. stages 1,2,...,i-1, containing trailing instructions are also stalled. All i stages of the pipeline are stalled until the instruction in stage i is forwarded its dependent operand. After the inter-instruction dependence is satisfied, then all i stalled instructions can again advance synchronously down the pipeline. For a rigid scalar pipeline, a stalled stage in the middle of the pipeline affects all earlier stages of the pipeline; essentially the stalling of stage i is propagated backward through all the preceding stages of the pipeline.

FIGURE 1 Unnecessary stall cycles induced by backward propagation of stalling in a rigid pipeline.


Bypassing of Stalled Instruction Not Allowed
Stalled Instruction
Backward Propagation of Stalling

The backward propagation of stalling from a stalled stage in a scalar pipeline induces unnecessary pipeline bubbles or idling pipeline stages. While an instruction is stalled in stage i due to its dependence on a leading instruction, there may be another instruction trailing the stalled instruction which does not have a dependence on any leading instruction that would require its stalling. For example this independent trailing instruction could be in stage i-1 and would be unnecessarily stalled due to the stalling of the instruction in stage i. According to program semantics, it is not necessary for this instruction to wait in stage i-1. If this instruction is allowed to bypass the stalled instruction and continues down the pipeline stages, an idling cycle of the pipeline can be eliminated and effectively reducing the penalty due to the stalled instruction by one cycle; see Figure 1. If multiple instructions are able and allowed to bypass the stalled instruction, then multiple penalty cycles can be eliminated or "covered" in the sense that idling pipeline stages are given useful instructions to process. Potentially all of the penalty cycles due to the stalled instruction can be covered. Allowing the bypassing of a stalled leading instruction by trailing instructions is referred to as *out of order execution* of instructions. A rigid scalar pipeline does not allow out of order execution and hence can incur unnecessary penalty cycles in enforcing inter-instruction dependences. Parallel pipelines that support out of order execution are called *dynamic pipelines*.

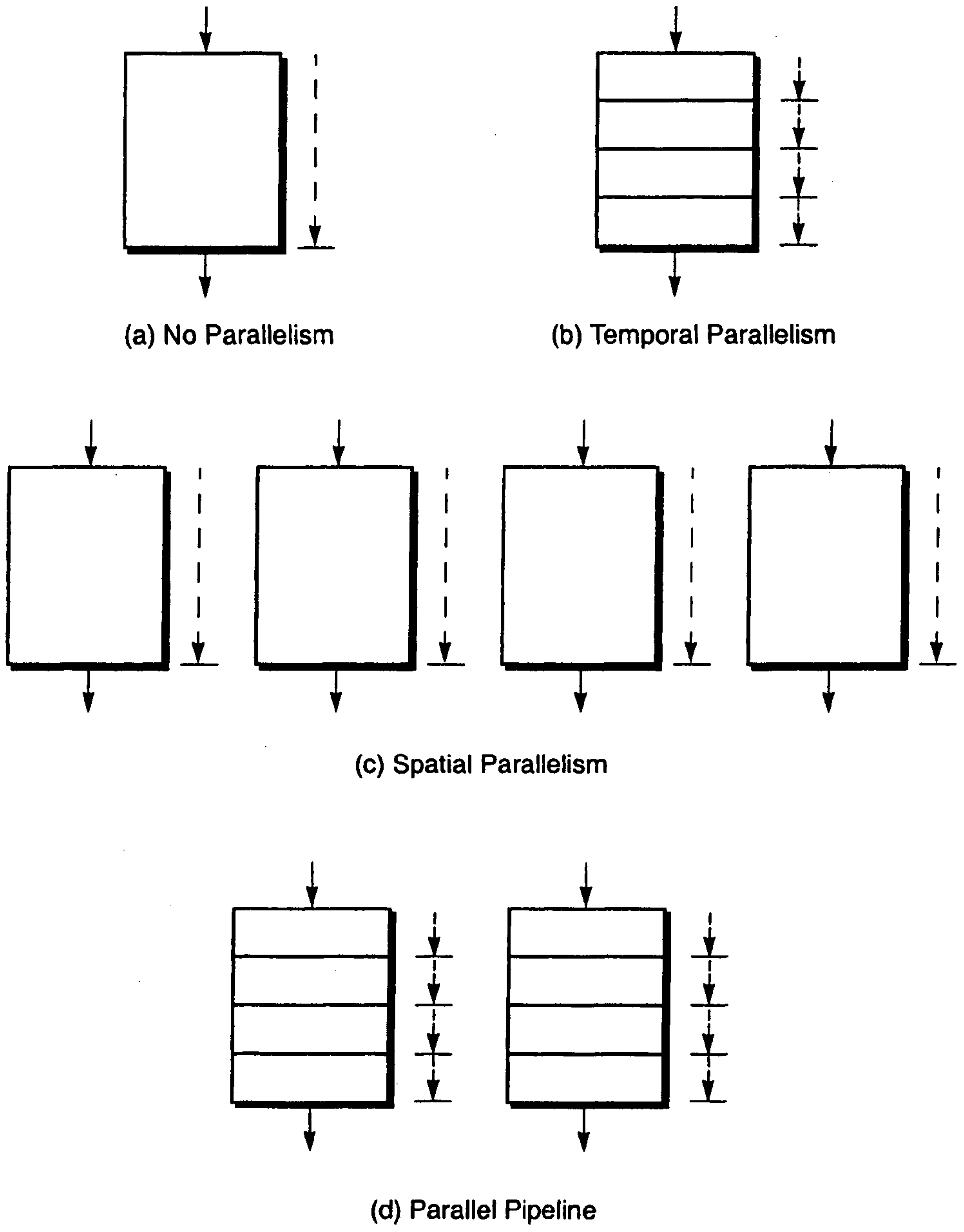
3.1.2 From Scalar to Superscalar Pipelines

Superscalar pipelines can be viewed as natural descendants of the scalar pipelines, and involve extensions to alleviate the three above-stated limitations with scalar pipelines. Superscalar pipelines are parallel pipelines, instead of scalar pipelines, in being able to initiate the processing of multiple instructions in every machine cycle. In addition, superscalar pipelines are diversified pipelines in employing multiple and heterogeneous functional units in their execution stage(s). Finally, superscalar pipelines can be implemented as dynamic pipelines in order to achieve the best possible performance without requiring reordering of instructions by the compiler. These three characterizing attributes of superscalar pipelines are further elaborated below.

3.1.2.1 Parallel Pipelines

The degree of parallelism of a machine can be measured by the maximum number of instructions that can be concurrently in progress at any one time. A k-stage scalar pipeline can have k instructions concurrently resident in the machine, and can potentially achieve a factor of k speedup over a non-pipelined machine. Alternatively, the same speedup can be achieved by employing k copies of the non-pipelined machine to process k instructions in parallel. These two forms of machine parallelism are illustrated in Figure 2 (b) and (c), and can be denoted *temporal machine parallelism* and *spatial machine parallelism*, respectively. Temporal and spatial parallelisms of the same degree can yield about the same factor of potential speedup. Clearly, temporal parallelism via pipelining requires less hardware than spatial parallelism, which requires replication of the entire processing unit. Parallel pipelines can be viewed as employing both temporal and spatial machine parallelisms, as illustrated in Figure 2 (d), to achieve higher instruction processing throughput in an efficient manner.

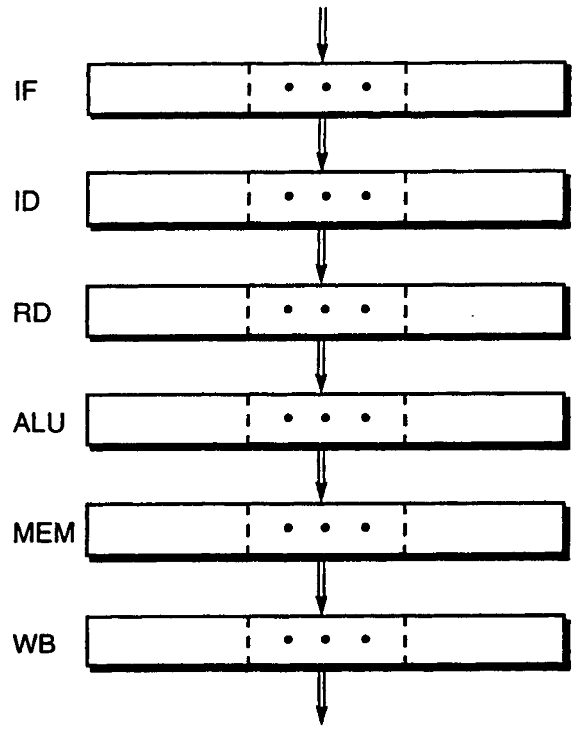
FIGURE 2 Machine parallelism: (a) no parallelism (non-pipelined); (b) temporal parallelism (pipelined); (c) spatial parallelism (multiple units); (d) combined temporal and spatial parallelisms.



The speedup of a scalar pipeline is measured with respect to a non-pipelined design and is primarily determined by the depth of the scalar pipeline. For parallel pipelines, or superscalar pipelines, the speedup now is usually measured with respect to a scalar pipeline and is primarily determined by the "width" of the parallel pipeline. A parallel pipeline with width s can concurrently process up to s instructions in each of its pipeline stages, which can lead to a potential speedup of s over a scalar pipeline. Figure 3 illustrates a parallel pipeline of width s=4.

Significant additional hardware resources are required for implementing parallel pipelines. Each pipeline stage can potentially process and advance up to s instructions in every machine cycle. Hence, the logic complexity of each pipeline stage can increase by a factor of s. In the worst case, the circuitry for inter-stage interconnection can increase by a factor of s^2 if an sxs crossbar is used to connect all s instruction buffers from one stage to all s instruction buffers of the next stage. In order to support concurrent register file accesses by s instructions, the number of read and write ports of the register file must be increased by a factor of s. Similarly, additional I-cache and D-cache access ports must be provided.

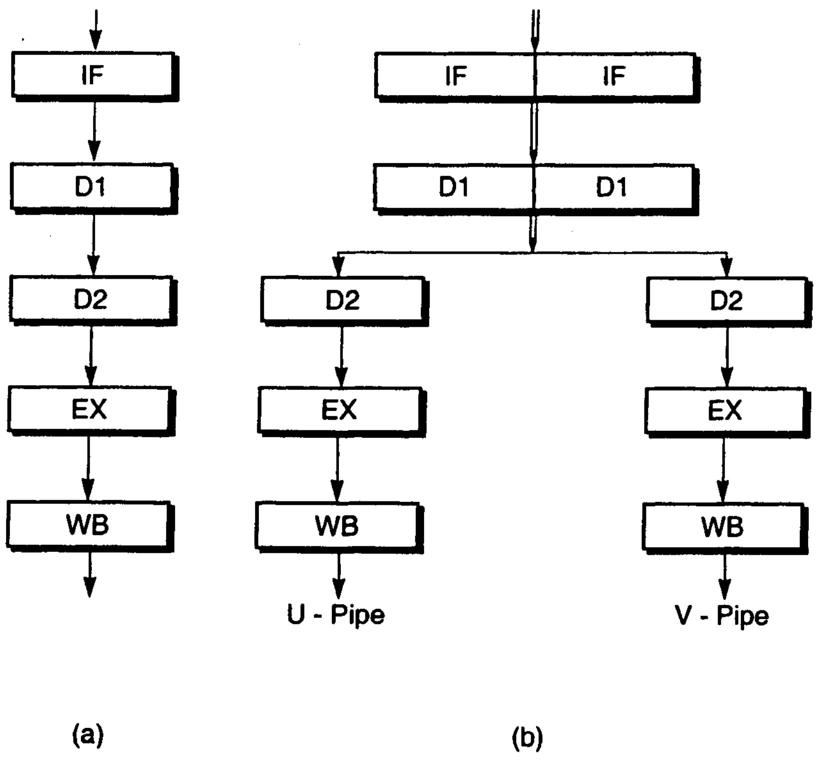
FIGURE 3 A parallel pipeline of width s=4.



As shown in the previous chapter, the Intel i486 is a 5-stage scalar pipeline. The sequel to the i486 is the recently introduced Pentium microprocessor from Intel. The Pentium microprocessor is a superscalar machine implementing a parallel pipeline of width s=2. It essentially implements two i486 pipelines; see Figure 4. Multiple instructions can be fetched and decoded by the first two stages of the parallel pipeline in every machine cycle. In each cycle, potentially two instructions can be issued into the two execution pipelines, i.e. the U-pipe and the V-pipe. The goal is to maximize the number of dual-issue cycles. The superscalar Pentium microprocessor can achieve a peak execution rate of two instructions per machine cycle.

As compared to the scalar pipeline of i486, the Pentium parallel pipeline requires significant additional hardware resources. First of all, the five pipeline stages have been doubled in width. The two execution pipes can accommodate up to two instructions in each of the last three stages of the pipeline. The execute stage can perform an ALU operation or access the D-cache. Hence, additional ports to the register file must be provided to support the concurrent execution of two ALU operations in every cycle. If the two instructions in the execute stage are both load/store instructions, then the D-cache must provide dual access. A true dual-ported D-cache is complex and expensive to implement. Instead, the Pentium D-cache is implemented as a single-ported D-cache with 8-way interleaving. Simultaneous accesses to two different banks by the two load/store instructions in the U and V pipes can be supported. If there is a bank conflict, i.e. both load/store instructions must access the same bank, then the two D-cache accesses are serialized.

FIGURE 4 (a) The 5-stage i486 scalar pipeline; (b) The 5-stage Pentium parallel pipeline of width s=2.



3.1.2.2 Diversified Pipelines

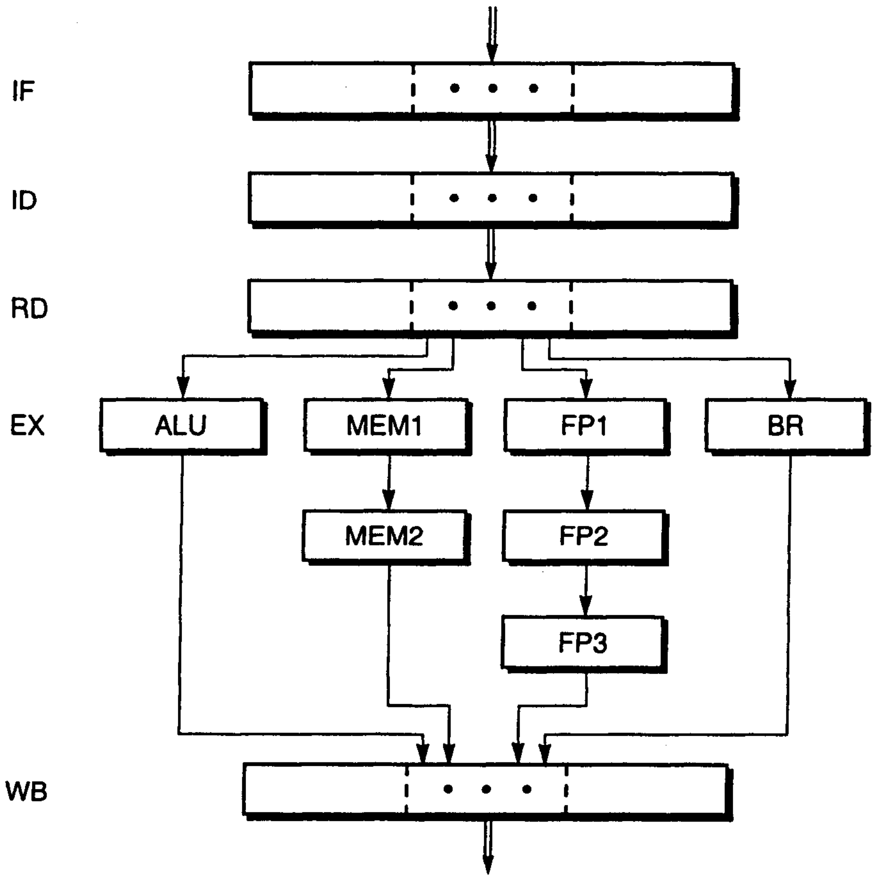
The fetching and initial decoding of instructions are subcomputations that are common to the processing of all instructions regardless of their types. Most instructions also require the reading of registers to obtain operands and the writing of registers to store results. However, the actual execution of instructions can differ significantly depending on the instruction types. Consequently, the hardware resources required to support the execution of different instruction types can vary significantly. For a scalar pipeline, all the diverse requirements for the execution of all instruction types must be unified into a single pipeline. The resultant pipeline can be highly inefficient. Each instruction type will only require a subset of the execution stages, but must traverse all the execution stages. Every instruction will be idling as it traverses the unnecessary stages and incur significant dynamic external fragmentation. The execution latency for all instruction types will be equal to the total number of execution stages. This can result in unnecessary stalling of trailing instructions and/or requiring additional forwarding paths.

This inefficiency due to unification into one single pipeline is naturally addressed in parallel pipelines by employing multiple different functional units in the execution stages. Instead of implementing s identical pipes in an s-wide parallel pipeline, in the execution portion of the parallel pipeline diversified execution pipes can be implemented; see Figure 5. In this example, four execution pipes, or functional units, of differing latencies are implemented. The RD stage dispatches instructions to the four execution pipes based on the instruction types.

There are a number of advantages in implementing diversified execution pipes. Each pipe can be customized for a particular instruction type resulting in efficient hardware design. Each instruction type incurs only the necessary latency and makes use of all the stages of an execution pipe. This is certainly more efficient than implementing *s* identical copies of an universal execution pipe each of which can execute all instruction types. If all inter-instruction dependences between different instruction types are resolved prior to dispatching, then once instructions are issued into the individual execution pipes no further stalling can occur due to instructions in other pipes. This allows the distributed and independent control of each execution pipe.

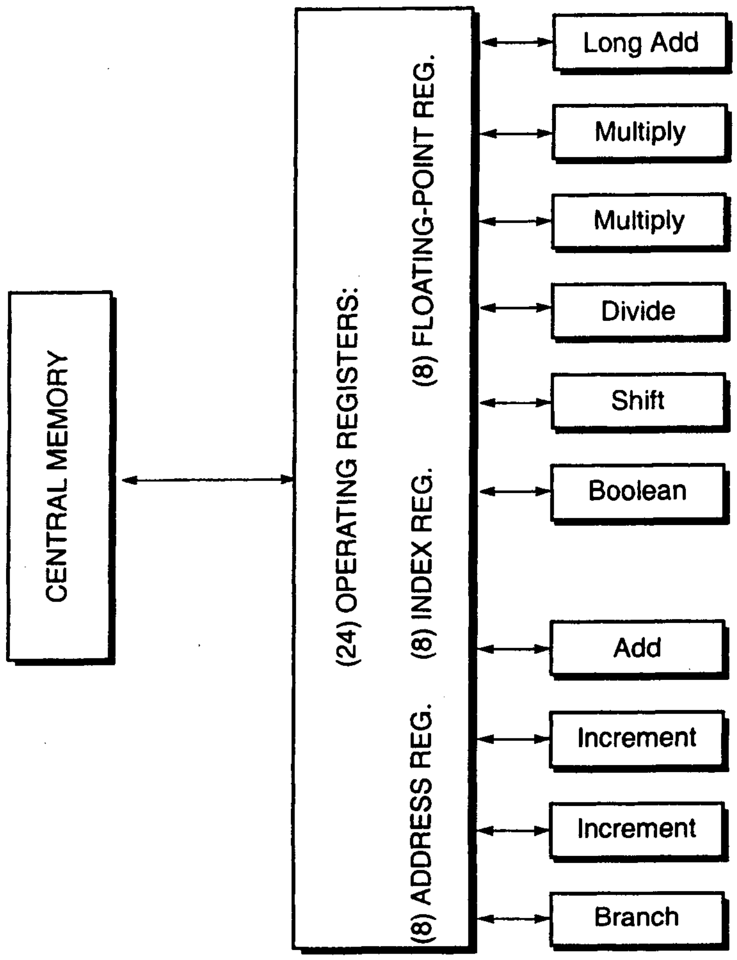
The design of a diversified parallel pipeline does require special considerations. One important consideration is the number and mix of functional units. Ideally the number of functional units should match the available instruction-level parallelism of the program and the mix of functional units should match the dynamic mix of instruction types of the program. Most first generation superscalar processors simply integrated a second execution pipe for processing floating-point instructions with the existing scalar pipe for processing non-floating-point instructions. As superscalar designs evolved from two-issue machines to four-issue machines, typically four functional units are implemented for executing integer, floating-point, load/store and branch instructions. Some recent designs incorporate multiple integer units, some of which are dedicated to long-latency integer operations such as multiply and divide, and others are dedicated to the processing of special operations for image and signal processing applications.

FIGURE 5 A diversified parallel pipeline of width s=4.



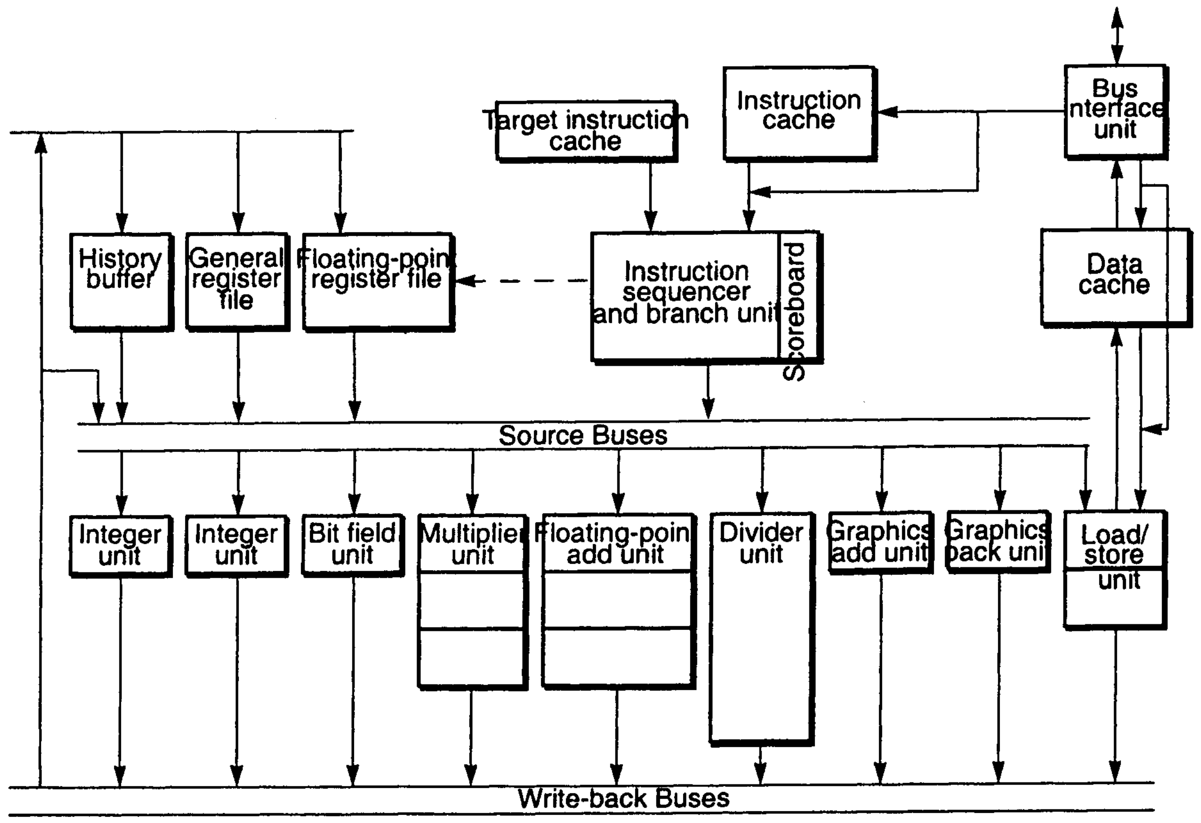
Similar to pipelining, the employment of a multiplicity of diversified functional units in the design of a high-performance CPU is not a recent invention. The CDC 6600 incorporated both pipelining and the use of multiple functional units [Thornton 1964]. The CPU of the CDC 6600 employs 10 diversified functional units as shown in Figure 6. The 10 functional units operate on data stored in 24 operating registers, which consist of 8 address registers (18 bits), 8 index registers (18 bits) and 8 floating-point registers (60 bits). The 10 functional units operate independently and consist of a fixed-point adder (18 bits), a floating-point adder (60 bits), two multiply units (60 bits), a divide unit (60 bits), a shift unit (60 bits), a Boolean unit (60 bits), two increment units, and a branch unit. The CDC 6600 CPU is a pipelined processor with two decoding stages preceding the execution portion, however the 10 functional units are not pipelined and have variable execution latencies. For example, a fixed-point add requires 3 cycles, and a floating-point multiply (divide) requires 10 (29) cycles. The goal of the CDC 6600 CPU is to sustain an issue rate of one instruction per machine cycle.

FIGURE 6 The CDC 6600 with ten functional units in its CPU.



A recent superscalar microprocessor employed similar mix of functional units as the CDC 6600. Just prior to the formation of the PowerPC alliance with IBM and Apple, Motorola had developed a very clean design of a wide superscalar microprocessor called the 88110 [Diefendorff & Allen 1992]. Interestingly, the 88110 also employs 10 functional units; see Figure 7. The 10 functional units consist of two integer units, a bit-field unit, a floating-point add unit, a multiply unit, a divide unit, two graphic units, a load/store unit, and an instruction sequencing/branch unit. Most of the units have single cycle latency. With the exception of the divide unit, the other units with multi-cycle latencies are all pipelined. In terms of the total number of functional units, the 88110 still represents one of the widest superscalar designs todate.

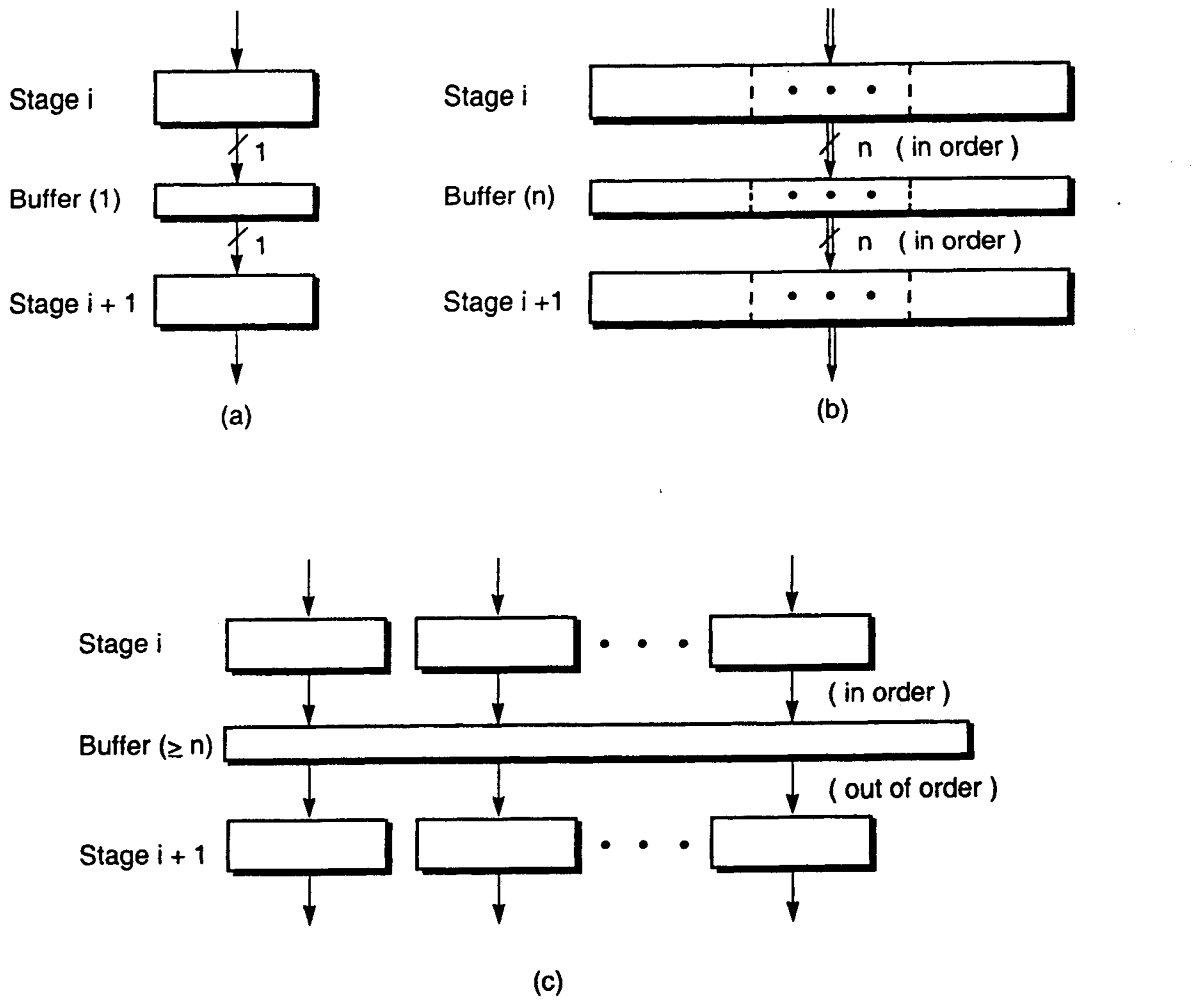
FIGURE 7 The Motorola 88110 superscalar microprocessor.



3.1.2.3 Dynamic Pipelines

In any pipelined design, buffers are required between pipeline stages. In a scalar rigid pipeline, a single-entry buffer is placed between two consecutive pipeline stages (stages i and j) as shown in Figure 8 (a). The buffer holds all essential control and data bits for the instruction that has just traversed stage i of the pipeline and is ready to traverse stage j in the next machine cycle. Single entry buffers are quite easy to control. In every machine cycle, the buffer's current content is used as input to stage j; and at the end of the cycle, the buffer latches in the result produced by stage i. Essentially the buffer is "clocked" in every machine cycle. The exception occurs when the instruction in the buffer must be held back and prevented from traversing stage j. In that case, the clocking of the buffer is disabled and the instruction is stalled in the buffer. Clearly if this buffer is stalled in a scalar rigid pipeline, all stages preceding stage i must also be stalled. Hence, in a scalar rigid pipeline, if there is no stalling, then every instruction remains in each buffer for exactly one machine cycle and then advances to the next buffer. All the instructions enter and leave each buffer in exactly the same order as specified in the original sequential code.

FIGURE 8 Inter-pipeline-stage buffers: (a) single entry buffer; (b) multi-entry buffer; (c) multi-entry buffer with reordering.



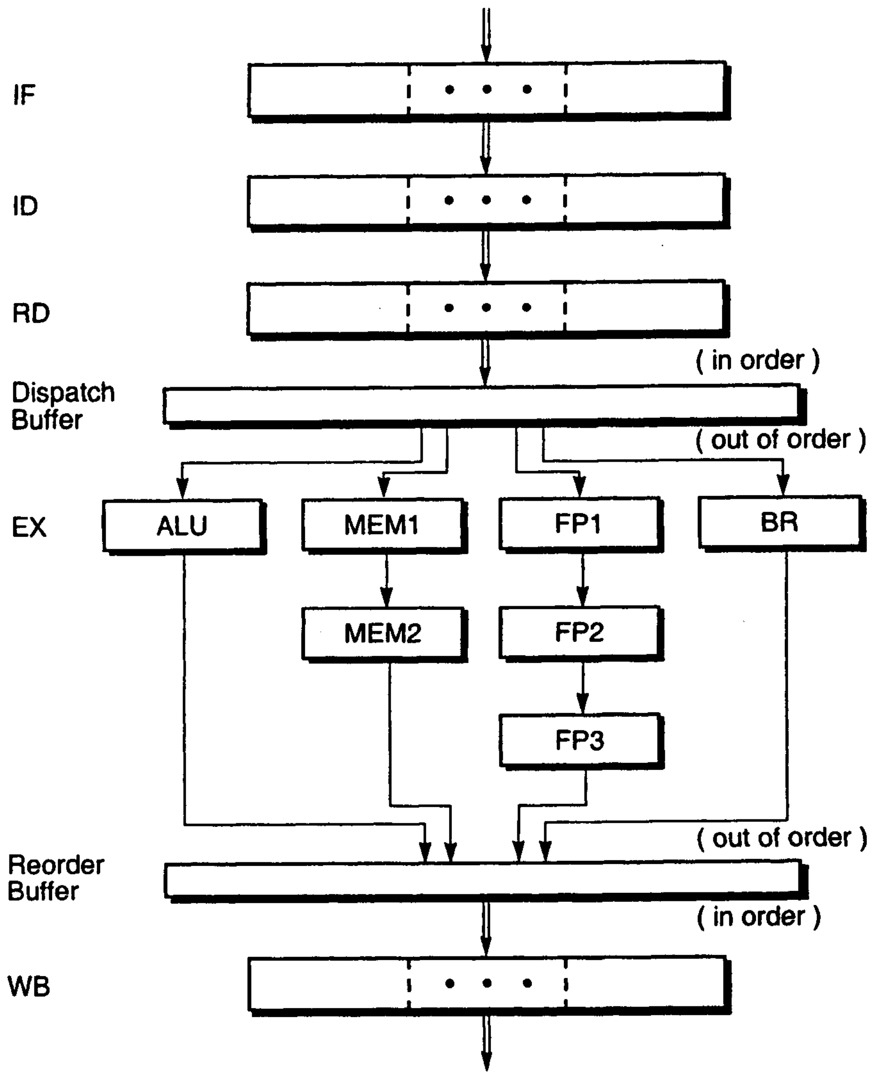
In a parallel pipeline, multi-entry buffers are needed between two consecutive pipeline stages as shown in Figure 8 (b). Multi-entry buffers can be viewed as simple extension of the single-entry buffers. Multiple instructions can be latched into each multi-entry buffer in every machine cycle. In the next cycle, these instructions can then traverse the next pipeline stage. If all of the instructions in a multi-entry buffer are required to advance simultaneously in a lock-step fashion, then the control of the multi-entry buffer is similar to that of the single-entry buffer. The entire multi-entry buffer is either clocked or stalled in each machine cycle. However such operation of the parallel pipeline may induce unnecessary stalling of some of the instructions in a multi-entry buffer. For more efficient operation of a parallel pipeline, much more sophisticated multi-entry buffers are needed.

Each entry of the simple multi-entry buffer of Figure 8 (b) is hardwired to one write port and one read port and there is no interaction between the multiple entries. One enhancement to this simple buffer is to add connectivity between the entries to facilitate movement of data between entries. For example, the entries can be connected into a linear chain like a shift register and function as a FIFO queue. Another enhancement is to provide mechanism for independent accessing of each entry in the buffer. This will require the ability to explicitly address each individual entry in the buffer and independently control the reading/writing of each entry. If each input/output port of the buffer is given the ability to access any entry in the buffer, then such a multi-entry buffer will effectively resemble a small multi-ported RAM. With such a buffer an instruction can remain in an entry of the buffer for many machine cycles and can be updated or modified while resident in that buffer. A further enhancement can incorporate associative accessing of the entries in the buffer. Instead of using conventional addressing to index into an entry in the buffer, the content of an entry can be used as an associative tag to index into that entry. With such accessing mechanism, the multi-entry buffer becomes a small associative cache memory.

Superscalar pipelines differ from (rigid) scalar pipelines in one key aspect, that is the use of complex multi-entry buffers for buffering instructions in flight. In order to minimize unnecessary stalling of instructions in a parallel pipeline, trailing instructions must be allowed to bypass a stalled leading instruction. Such bypassing can change the order of execution of instructions from the original sequential order of the static code. With out of order execution of instructions, there is the potential of approaching the data-flow limit of instruction execution, i.e. instructions are executed as soon as their operands are available. A parallel pipeline that supports out of order execution of instructions is called a dynamic pipeline. A dynamic pipeline achieves out of order execution via the use of complex multi-entry buffers that allow instructions to enter and leave the buffers in different orders. Such complex multi-entry buffers, as shown in Figure 8 (c), can be called *reorder buffers*.

Figure 9 illustrates a parallel diversified pipeline of width s=4 that is a dynamic pipeline. The execution portion of the pipeline consisting of the four pipelined functional units are bracketed by two reorder buffers. The first reorder buffer, called the *dispatch buffer*, is loaded with decoded instructions according to program order and then dispatches instructions to the functional units potentially in an order different from the program order. Hence instructions can leave the dispatch buffer in different order than the order in which they enter the dispatch buffer. This pipeline also implements a set of diverse functional units with different latencies. With potential out of order issuing into the functional units and/or the variable latencies of the functional units, instructions can clearly finish execution out of order. In order to ensure that exceptions can be handled according to original program order, the instructions must be completed, i.e. update machine state, in program order. With instructions finishing execution out of order, to ensure in order completion another reorder buffer is needed at the back end of the execution portion of the pipeline. This reorder buffer, called the *completion buffer*, buffers the instructions that may have finished execution out of order and retires the instructions in order by outputting instructions to the final write-back stage in program order. Such a dynamic pipeline facilitates the out of order execution of instructions to achieve the shortest possible execution time, and yet can provide precise exception by retiring the instructions and updating the machine state according to program order. One primary form of added complexity is the implementation of the complex multi-entry buffers and the logic required to control the accessing of these reorder buffers.

FIGURE 9 A dynamic pipeline of width *s*=4.



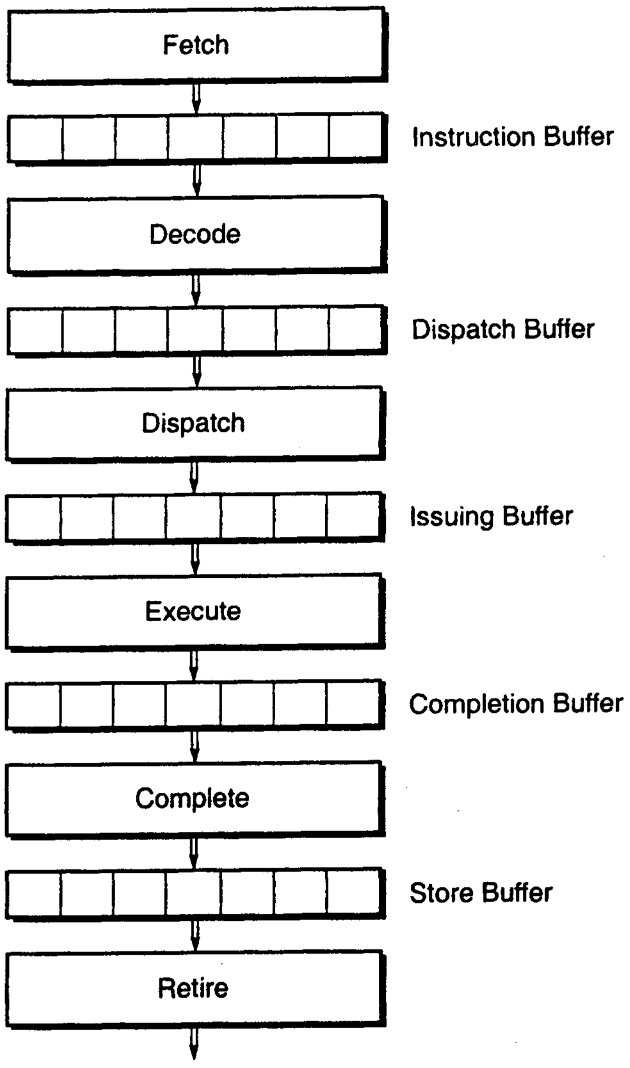
3.1.3 Superscalar Pipeline Design

Section 3.1 focuses on issues related to the organization of the pipeline. Limitations associated with scalar pipelines are presented, and the three key attributes needed in the organization of superscalar pipelines, namely they must be parallel, diversified and dynamic pipelines, are outlined. This subsection presents the critical issues involved in the design of superscalar pipelines. The focus is on the organization, or structural design, of superscalar pipelines. Issues and techniques related to the dynamic interaction of machine organization and instruction semantics and the optimization of the resultant machine performance are covered in the next section. Essentially this section focuses on the design of the machine, while the next section takes into account the interaction between the machine and the program.

Similar to the use of the 6-stage TYP pipeline in Chapter 2 for scalar pipeline design, we will use the 6-stage TEM superscalar pipeline shown in Figure 10 as a "template" for discussion on the organization of superscalar pipelines. Compared to scalar pipelines, there is far more variety and greater diversity in the implementation of superscalar pipelines. The TEM superscalar pipeline should not be viewed as an actual implementation of a typical or representative superscalar pipeline. The six stages of the TEM superscalar pipeline should be viewed as "logical" pipeline stages which may or may not correspond to six physical pipeline stages. The six stages of the TEM superscalar pipeline provide a nice framework or outline for discussing the six major portions of, or six major tasks performed by, most superscalar pipeline organizations.

The six stages of the TEM superscalar pipeline are: Fetch, Decode, Dispatch, Execute, Complete, and Retire. The Execute stage can include multiple (pipelined) functional units of different types with different execution latencies. This necessitates the Dispatch stage to distribute instructions of different types to their corresponding functional units. With out-of-order execution of instructions in the Execute stage, the Complete stage is needed to reorder the instructions and ensure the in-order updating of the machine state. Note also that there are multi-entry buffers separating these six stages. The complexity of these buffers can vary depending on their functionality and location in the superscalar pipeline. These six stages and design issues related to them are now addressed in turn.

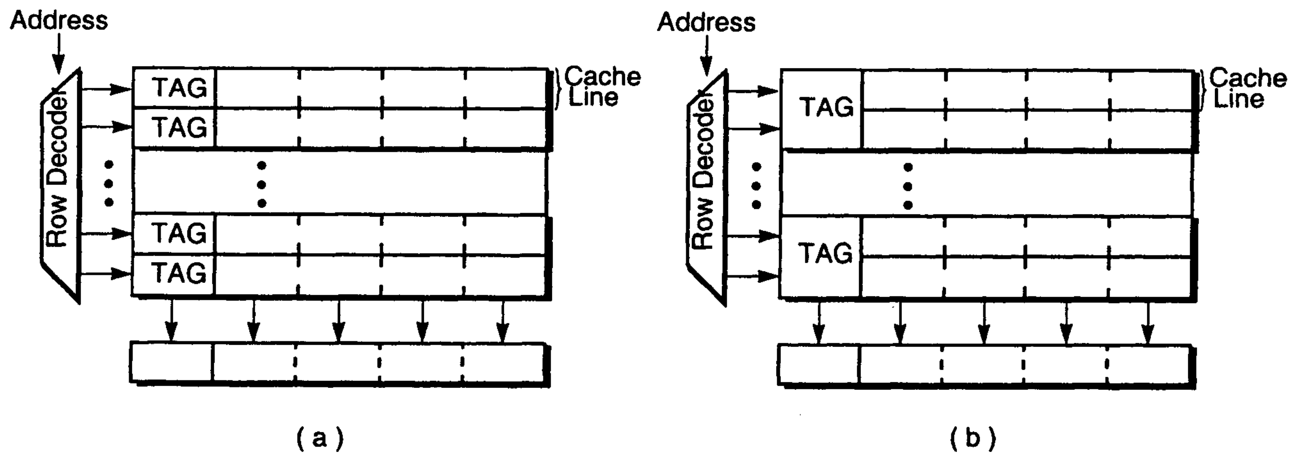
FIGURE 10 The 6-stage TEMPLATE (TEM) superscalar pipeline.



3.1.3.1 Instruction Fetching

Unlike a scalar pipeline, a superscalar pipeline being a parallel pipeline, is capable of fetching more than one instruction from the I-cache in every machine cycle. Given a superscalar pipeline of width s, its Fetch stage should be able to fetch s instructions from the I-cache in every machine cycle. This implies that the physical organization of the I-cache must be wide enough, so that each row of the I-cache array can store s instructions and that an entire row can be accessed at one time. In our current discussion, we assume that the access latency of the I-cache is one cycle. Typically in such a wide cache organization, a cache line corresponds to a physical row in the cache array; it is also possible that a cache line can span several physical rows of the cache array as illustrated in Figure 11.

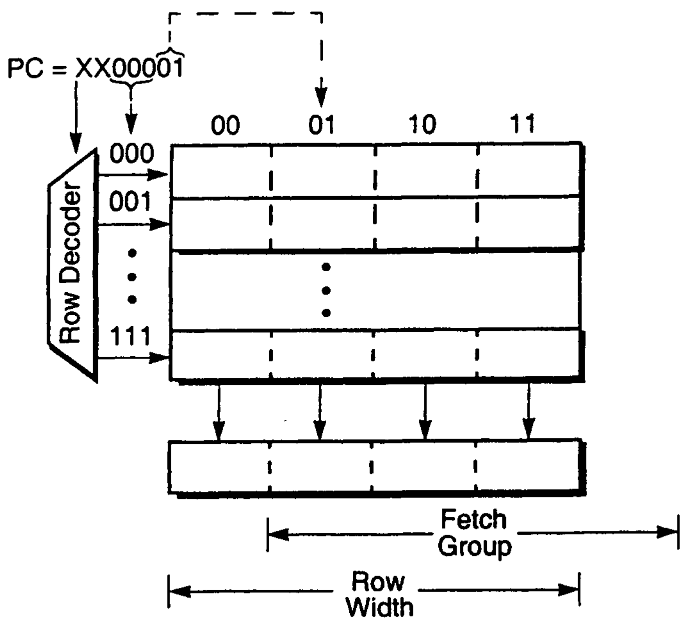
FIGURE 11 Organization of a wide I-cache: (a) one cache line is equal to one physical row; (b) one cache line is equal to two physical rows.



The primary objective of the Fetch stage is to maximize the instruction-fetching bandwidth. The sustained throughput achieved by the Fetch stage will impact the overall throughput of the superscalar pipeline, because the throughput of all subsequent stages depends on and cannot possibly exceed the throughput of the Fetch stage. Two primary impediments to achieving the maximum throughput of s instructions fetched per cycle are: 1) the misalignment of the s instructions being fetched, called the *fetch group*, with respect to the row organization of the I-cache array; and 2) the presence of control-flow changing instructions in the fetch group.

In every machine cycle, the Fetch stage uses the program counter (PC) to index into the I-cache to fetch the instruction pointed to by the PC along with the next s-1 instructions, i.e. the s instructions of the fetch group. If the entire fetch group is stored in the same row of the cache array, then all s instructions can be fetched. On the other hand, if the fetch group crosses a row boundary, then not all s instructions can be fetched in that cycle (assuming that only one row of the I-cache can be accessed in each cycle). Hence, only those instructions in the first row can be fetched; the remaining instructions will require another cycle for their fetching. Consequently, the fetch bandwidth is effectively reduced by half, for it now requires two cycles to fetch s instructions. This is due to the *misalignment* of the fetch group with respect to the row boundaries of the I-cache array, as illustrated in Figure 12. Such misalignments reduce the effective fetch bandwidth. In case that each cache line corresponds to a physical row, as is the case shown in Figure 11 (a), then the crossing of a row boundary also corresponds to the crossing of a cache line boundary, which can incur additional problems. If a fetch group spans two cache lines, then it can induce an I-cache miss involving the second line even though the first line is resident. Even if both lines are resident in the I-cache, the physical accessing of multiple cache lines in one cycle is problematic and is not supported by most I-cache designs.

FIGURE 12 Misalignment of the fetch group relative to the row boundaries of the I-cache array.

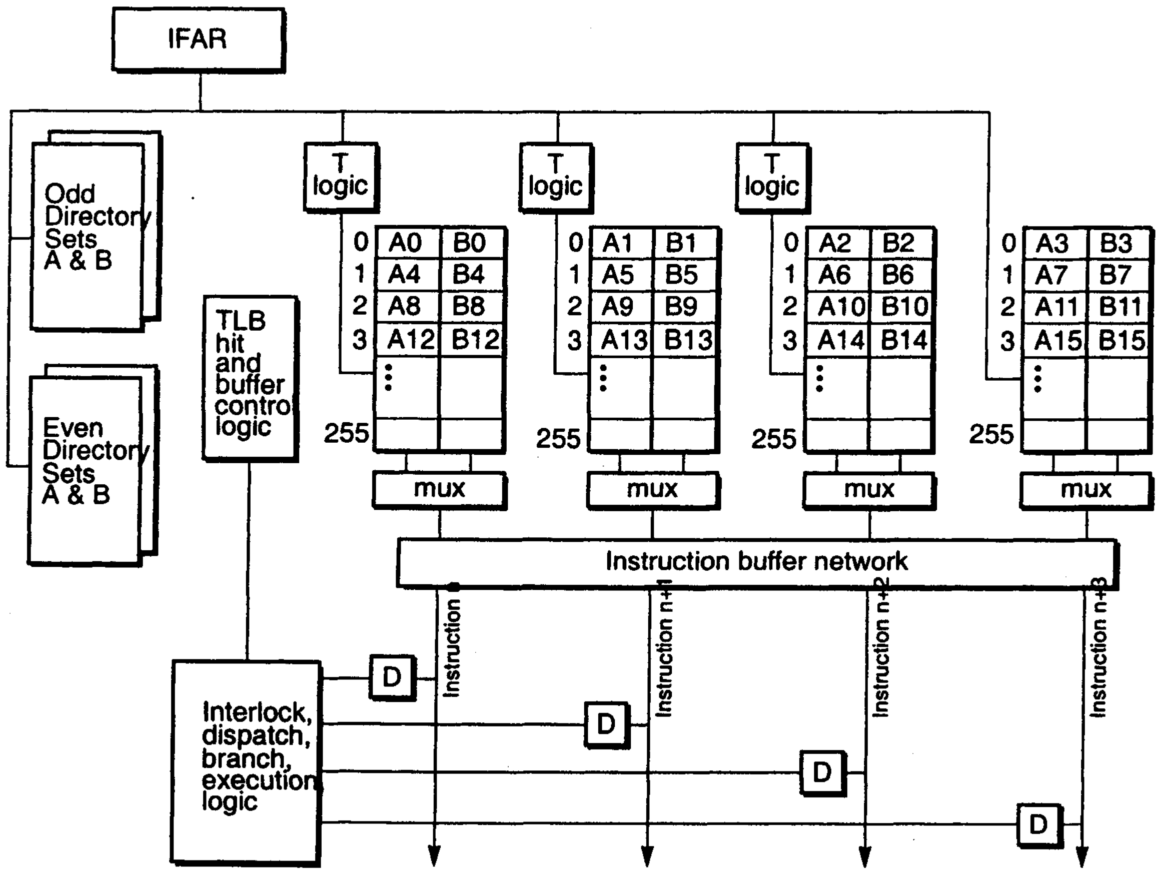


There are two possible solutions to the misalignment problem. The first solution is a static technique employed at compile time. The compiler or assembler can be given information on the organization of the I-cache, e.g. its indexing scheme and row size. Based on this information instructions can be appropriately placed in memory locations so as to ensure the aligning of fetch groups with physical rows. For example, every instruction that is the target of a branch can be placed in a memory location that is mapped to the first instruction of a row. This will increase the probability of fetching *s* instructions from the beginning of a row. Such techniques have been implemented and are reasonably effective [ ]. A problem with this solution is that the object code is tuned to a particular I-cache organization and may not be properly aligned for other I-cache organizations. Another problem is that the static code now spans a larger address range, which can potentially lead to a higher I-cache miss rate.

The second solution to the misalignment problem involves using hardware at run time. Alignment hardware can be incorporated to ensure that *s* instructions are fetched in every cycle even if the fetch group crosses a row boundary (but not a cache line boundary). Such alignment hardware is incorporated in the IBM RS/6000 design [IBM-JRD, Jan.90]; we now briefly describe this design.

The RS/6000 employs a two-way set-associative I-cache with a line size of 16 instructions (64 bytes). Each row of the I-cache array stores four associative sets (two per set) of instructions. Hence, each line of the I-cache spans four physical rows, as shown in Figure 13. The physical I-cache array is actually composed of four independent subarrays (denoted 0,1,2, and 3), which can be accessed in parallel. One instruction can be fetched from each subarray in every I-cache access. Which of the two instructions (either A or B) in the associative set is accessed depends on which of the two has a tag match with the address. The instruction addresses are allocated in an interleaved fashion across the four subarrays.

FIGURE 13 Organization of the RS/6000 two-way set-associative I-cache with auto-realignment.



If the PC happens to point to the first subarray, i.e. subarray 0, then four consecutive instructions can be simultaneously fetched from the four subarrays. All four of these instructions reside in the same physical row of the I-cache, and all four subarrays are accessed using the same row address. On the other hand, if the PC indexes into the middle of the row, i.e. the first instruction of the fetch group resides in subarray 2, then the four consecutive instructions in the fetch group will span across two rows. The RS/6000 deals with this problem by detecting when the starting address points to a subarray other than subarray 0, and automatically incrementing the row address of the non-consecutive subarrays. This is done by the "T-logic" hardware associated with each subarray. For example, if the PC indexes into subarray 2, then subarrays 2 and 3 will be accessed with the same row address presented to them. However the T-logic of subarrays 0 and 1 will detect this condition and automatically increment the row address presented to subarrays 0 and 1. Consequently the two instructions fetched from subarrays 0 and 1 will actually be from the next physical row of the I-cache. Therefore, regardless of the starting address and where that address points to in an I-cache row, four consecutive instructions can always be fetched in every cycle as long as the fetch group does not cross a cache line boundary. When a fetch group crosses a cache line boundary, only instructions in the first cache line can be fetched in that cycle. Given the fact that the cache line of the RS/6000 consists of 16 instructions, and that there are 16 possible starting addresses of a word in a cache line, on the average the fetch bandwidth of this I-cache organization is $(13/16)\text{x}4 + (1/16)\text{x}3 + (1/16)\text{x}2 + (1/16)\text{x}1 = 3.625$ instructions per cycle.

Although the fetch group can begin in any one of the four subarrays, only subarrays 0, 1, and 2 require the T-logic hardware. The row address of subarray 3 never needs to be incremented regardless of the starting subarray of a fetch group. The "instruction buffer network" in the RS/6000 contains a rotating network which can rotate the four fetched instructions so as to present the four instructions, at its output, in original program order. This design of the I-cache is quite sophisticated and can ensure high fetch bandwidth even if the fetch group is misaligned with respect to the row organization of the I-cache. However it is also quite hardware intensive and may not be feasible if the RS/6000 were implemented on a single chip with on-chip caches.

Other than the misalignment problem, the second impediment to sustaining the maximum fetch bandwidth of *s* instructions per cycle is the presence of control-flow changing instructions within the fetch group. If one of the instructions in the middle of the fetch group is a conditional branch, then the subsequent instructions in the fetch group will be discarded if the branch is taken. Consequently, when this happens the fetch bandwidth is effectively reduced. This problem is fundamentally due to the presence of control dependences between instructions and is related to the handling of conditional branches. This topic, viewed as more related to the dynamic interaction between the machine and the program, is addressed in Subsection 3.2.1 which covers techniques for dealing with control dependences and branch instructions.

3.1.3.2 Instruction Decoding

Instruction decoding involves the identification of the individual instructions, determination of the instruction types, and the detection of inter-instruction dependences, among the group of instructions that have been fetched but not yet dispatched. The complexity of the instruction decoding task is strongly influenced by two factors, namely the ISA and the width of the parallel pipeline. For a typical RISC instruction set with fixed-length instructions and simple instruction formats the decoding task is quite straight forward. No explicit effort is needed to determine the beginning and ending of each instruction. The relatively few different instruction formats and addressing modes makes the distinguishing of instruction types reasonably easy. By simply decoding a small portion, e.g. one opcode byte, of an instruction the instruction type and the format used can be determined, and the remaining fields of the instruction and their interpretation can be quickly identified. A RISC instruction set simplifies the instruction decoding task.

For a RISC scalar pipeline, instruction decoding is quite trivial. Frequently the decode stage is used for accessing the register operands and is merged with the register read stage. However for a RISC parallel pipeline with multiple instructions being simultaneously decoded, the decode stage must identify dependences between these instructions and determine the independent instructions that can be dispatched in parallel. Furthermore, in order to support efficient instruction fetching, the decode stage must quickly identify control-flow changing branch instructions among the instructions being decoded in order to provide quick feedback to the fetch stage. These two tasks in conjunction with accessing many register operands can make the logic for the decode stage of a RISC parallel pipeline quite complex. Large number of comparators are needed for determining register dependences between instructions. The register files must be multi-ported and able to support many simultaneous accesses. Multiple buses are also needed to route the accessed operands to their appropriate destination buffers. It is possible that the decode stage can become the critical stage in the overall superscalar pipeline.

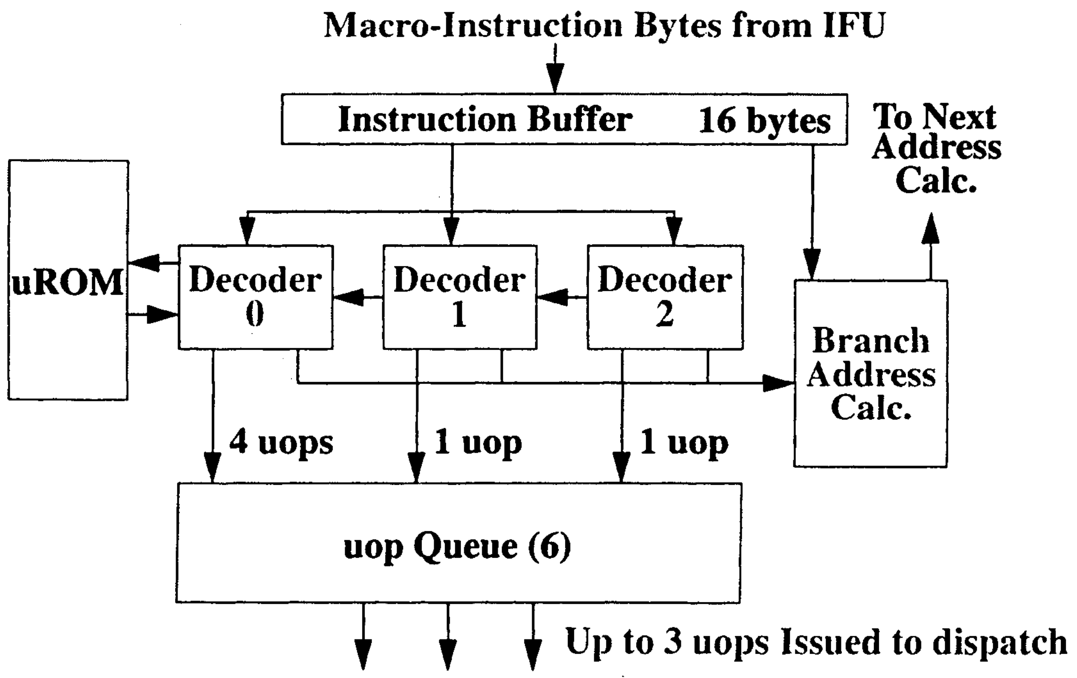
For a CISC parallel pipeline, the instruction decoding task can become even more complex and usually requires multiple pipeline stages. For such a parallel pipeline, the identification of individual instructions and their types is no longer trivial. Both the Intel Pentium [ ] and the AMD K5 [ ] employ two pipeline stages for decoding x86 instructions. On the more deeply pipelined Intel Pentium Pro, a total of five machine cycles are required to access the I-cache and decode the x86 instructions [ ]. The use of variable instruction lengths imposes an undesirable sequentiality to the instruction decoding task; the leading instruction must be decoded and have its length determined, before the beginning of the next instruction can be identified. Consequently, the simultaneous parallel decoding of multiple instructions can become quite challenging. In the worst case, it must be assumed that a new instruction can begin anywhere within the fetch group and a large number of decoders are used to simultaneously and "speculatively" decode instructions starting at every byte boundary. This is extremely complex and can be quite inefficient.

There is an additional burden on the instruction decoder of a CISC parallel pipeline. The decoder must translate the architected instructions into internal low-level operations that can be directly executed by the hardware. These internal operations resemble RISC instructions and can be viewed as vertical microinstructions. In the AMD K5 these operations are called "RISC operations" or "ROPs" (pronounced "ar-ops"). In the Intel P6 these internal operations are identified as "micro-operations" or "uops" (pronounced "you-ops"). Each x86 instruction is translated into one or more ROPs or uops. According to Intel, on the average one x86 instruction is translated into 1.5-2.0 uops [ ]. In these CISC parallel pipelines, between the instruction decoding and instruction completion stages all instructions in flight within the machine are these internal operations. In this book, for convenience we will adopt the Intel terminology and refer to these internal operations as uops.

The instruction decoder for the Intel Pentium Pro is presented as an illustrative example of instruction decoding for a CISC parallel pipeline. A diagram of the Fetch/Decode unit of the P6 is shown in Figure 14. In each machine cycle, the I-cache can deliver 16 aligned bytes to the instruction queue. Three parallel decoders simultaneously decode instruction bytes from the instruction queue. The first decoder at the front of the queue is capable of decoding all x86 instructions, while the other two decoders have more limited capability and can only decode simple x86 instructions such as register-to-register instructions. The decoders translate x86 instructions into the internal 3-address uops. The uops employ the load/store model. Each x86 instruction with complex addressing modes is translated into multiple uops. The first (generalized) decoder can generate up to four uops per cycle in response to the decoding of an x86 instruction. Each of the other two (restricted) decoders can generate only one uop per cycle in response to the decoding of a simple x86 instruction. In each machine cycle at least one x86 instruction will be decoded by the generalized decoder, leading to the generation of one or more uops. The goal is to go beyond this and have the other two restricted decoders also decode two simple x86 instructions that trail the leading x86 instruction in the same machine cycle. In the most ideal case the three parallel decoders can generate a total of 6 uops in one machine cycle. For those complex x86 instructions that require more than four uops to translate, when they reach the front of the instruction queue, the generalized decoder will invoke a uops sequencer to emit microcode, which is simply a preprogrammed sequence of normal uops. These uops will require two or more machine cycles to generate. All the uops generated by the three parallel decoders are loaded into the reorder buffer (ROB), which has 40 entries to hold up to 40 uops, to await dispatching to the functional units.

For many superscalar processors, especially those that implement wide and/or CISC parallel pipelines, the instruction decoding hardware can be extremely complex and require partitioning into multiple pipeline stages. When the number of decoding stages is increased the branch penalty, in terms of number of machine cycles, is also increased. Hence, it is not desirable to just keep increasing the depth of the decoding portion of the parallel pipeline. To help alleviate this complexity, in recent years a technique call *predecoding* has been proposed and implemented.

FIGURE 14 The Fetch/Decode unit of the Intel P6 superscalar pipeline.

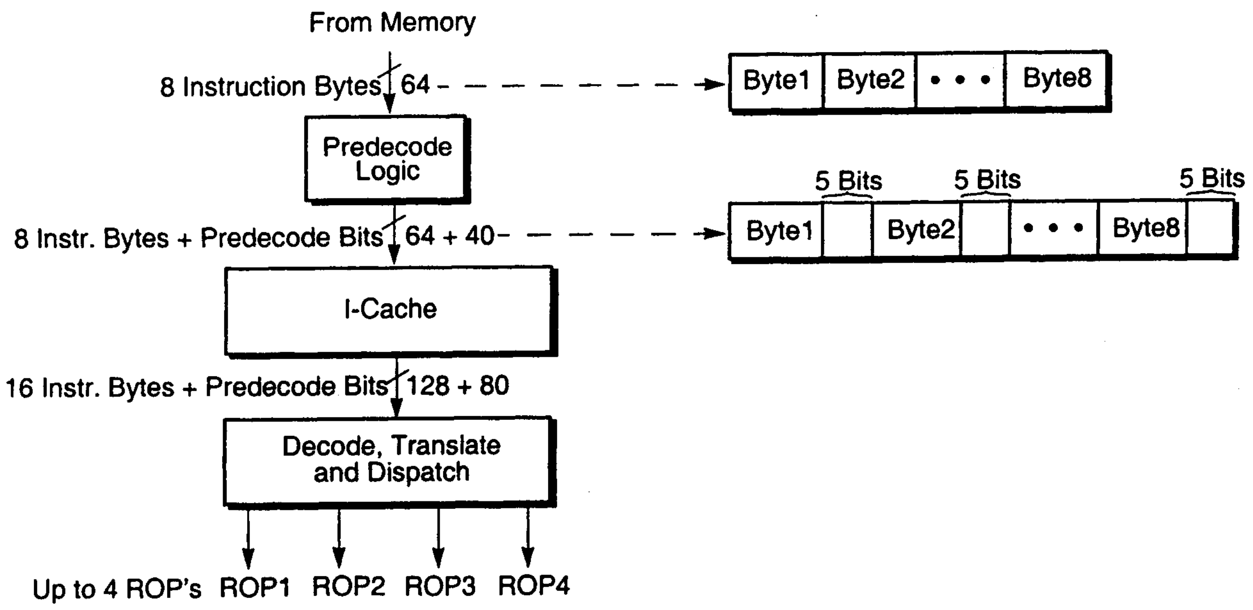


Predecoding moves a part of the decoding task to the other side, i.e. the input side, of the I-cache. When an I-cache miss occurs and a new cache line is being brought in from the memory, the instructions in that cache line are partially decoded by decoding hardware that is placed between the memory and the I-cache. The instructions and some additional decoded information are stored in the I-cache. The decoded information, in the form of predecode bits, simplifies the instruction decoding task when the instructions are fetched from the I-cache. Hence, part of the decoding is performed only once when instructions are loaded into the I-cache, instead of every time when these instructions are fetched from the I-cache. With some of the decoding hardware having been moved to the input side of the I-cache, the actual instruction decoding stage(s) of the parallel pipeline can be significantly simplified.

The AMD K5 is an example of a CISC superscalar pipeline that employs aggressive predecoding of x86 instructions as they are fetched from memory and prior to their being loaded into the I-cache. In a single bus transactions a total of eight instruction bytes are fetched from memory. These bytes are predecoded, and five additional bits are generated by the predecoder for each of the instruction bytes. These five predecode bits contain information about the location of the start and end of an x86 instruction, the number of uops (or ROPs) needed to translate that x86 instruction, and the location of opcodes and prefixes. These additional predecode bits are stored in the I-cache along with the original instructions bytes. Consequently, the original I-cache line size of

128 bits (16 bytes) is increased by an additional 80 bits; see Figure 15. In each I-cache access, the 16 instruction bytes are fetched along with the 80 predecode bits. The predecode bits significantly simplify instruction decoding and allow the simultaneous decoding of multiple x86 instructions by four identical decoders/translators that can generate up to four uops in each cycle.

FIGURE 15 The predecoding mechanism of the AMD K5.


From Memory
8 Instruction Bytes 64
Byte1 Byte2 • • • Byte8
Predecode Logic
5 Bits 5 Bits 5 Bits
8 Instr. Bytes + Predecode Bits 64 + 40
Byte1 Byte2 • • • Byte8
I-Cache
16 Instr. Bytes + Predecode Bits 128 + 80
Decode, Translate and Dispatch
Up to 4 ROP's ROP1 ROP2 ROP3 ROP4

There are two forms of overhead associated with predecoding. The I-cache miss penalty can be increased due to the necessity of predecoding the instruction bytes fetched from memory. This is not a serious problem if the I-cache miss rate is very low. The other overhead involves the storing of the predecode bits in the I-cache and the consequent increase of the I-cache size. For the K5 the size of the I-cache is increased by about 50%. There is clearly a trade-off between the aggressiveness of predecoding and the I-cache size increase.

Predecoding is not just limited to alleviating the sequential bottleneck in parallel decoding of multiple CISC instructions in a CISC parallel pipeline. It can also be used to support RISC parallel pipelines. RISC instructions can be predecoded when they are being loaded into the I-cache. The predecode bits can be used to identify control-flow changing branch instructions within the fetch group and to explicitly identify subgroups of independent instructions within the fetch group. For example, the PowerPC 620 employs 7 predecode bits for each instruction word in the

I-cache. The UltraSPARC, MIPS R10000, and HP PA-8000 also employ either 4 or 5 predecode bits for each instruction [up rep. Dec. 26, 1994].

As superscalar pipelines become wider and the number of instructions that must be simultaneously decoded increases, the instruction decoding task will become more of a bottleneck and more aggressive use of predecoding can be expected. The predecoder partially decodes the instructions, and effectively transforms the original undecoded instructions into a format that makes the final decoding task easier. One can view the predecoder as translating the instructions fetched from memory into different instructions that are then loaded into the I-cache. Extending on this view, the possibility of enhancing the predecoder to do run-time object code translation between ISAs could be an interesting idea.

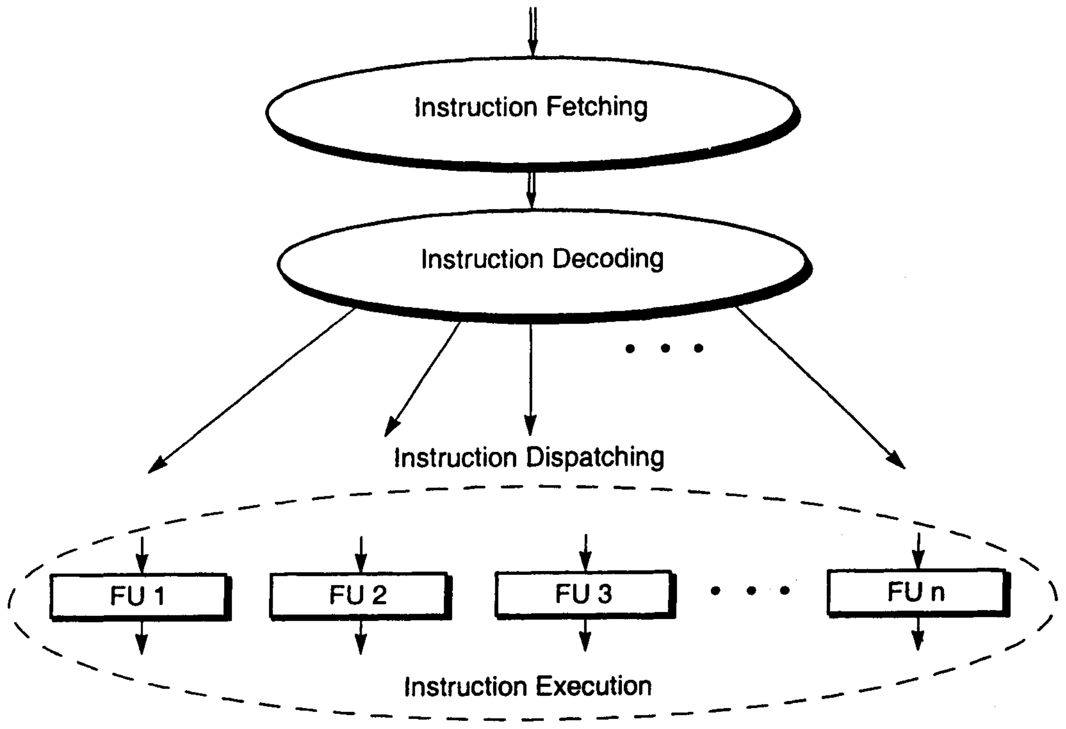
3.1.3.3 Instruction Dispatching

Instruction dispatching is necessary for superscalar pipelines. In a scalar pipeline, all instructions regardless of their types flow through the same single pipeline. Superscalar pipelines are diversified pipelines that employ a multiplicity of heterogeneous functional units in their execution portion. Different types of instructions are executed by different functional units. Once the type of an instruction is identified in the decode stage, it must be routed to the appropriate functional unit for execution; this is the task of instruction dispatching.

Although superscalar pipelines are parallel pipelines, both the instruction fetching and instruction decoding tasks are usually carried out in a centralized fashion, i.e. all the instructions are managed by the same controller. Although multiple instructions are fetch in a cycle, all instructions must be fetched from the same I-cache. Hence all the instructions in the fetch group are accessed from the I-cache at the same time and they are all deposited into the same buffer. Instruction decoding is done in a centralized fashion because in the case of CISC instructions, all the bytes in the fetch group, must be decoded collectively by a centralized decoder in order to identify the individual instructions. Even with RISC instructions, the decoder must identify inter-instruction dependences, which also requires centralized instruction decoding.

On the other hand, in a diversified pipeline all the functional units can operate independently in a distributed fashion in executing their own types of instructions once the inter-instruction dependences are resolved. Consequently, going from instruction decoding to instruction execution, there is a change from centralized processing of instructions to distributed processing of instructions. This change is carried out by and is the reason for the instruction dispatching stage in a superscalar pipeline. This is illustrated in Figure 16.

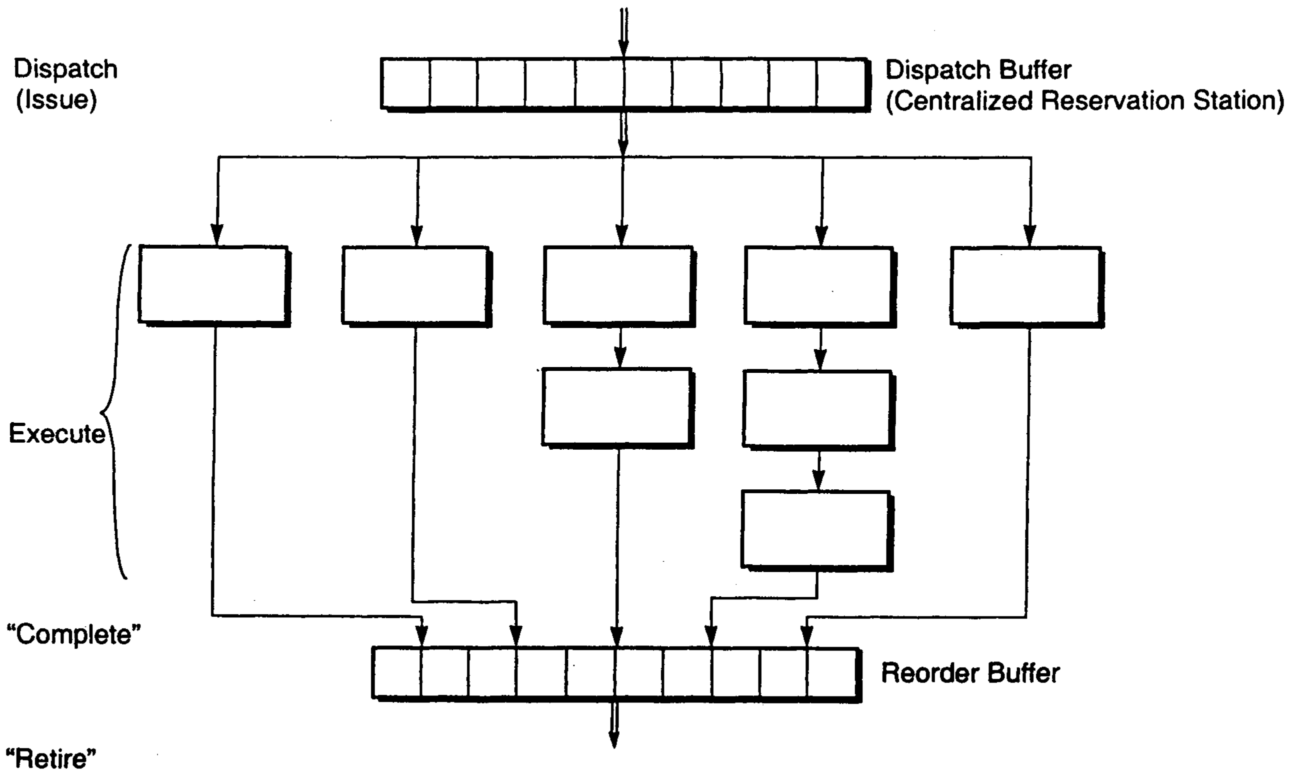
FIGURE 16 The necessity of instruction dispatching in a superscalar pipeline.



Another mechanism that is necessary between instruction decoding and instruction execution is the temporary buffering of instructions. Prior to its execution, an instruction must have all of its operands. During decoding, register operands are fetched from the register files. In a superscalar pipeline it is possible that some of these operands are not yet ready due to earlier instructions, which update these registers, being still in flight. When this situation occurs, an obvious solution is to stall the decoding stage until all register operands are ready. This solution seriously restricts the decoding throughput and is not desirable. A better solution is to fetch those register operands that are ready and go ahead and advance these instructions into a separate buffer to await those register operands that are not ready. When all register operands are ready, those instructions can then exit this buffer and be issued into the functional units for execution. Borrowing the term from Tomasulo's algorithm [Tomasulo 1967] we will denote such a temporary instruction buffer as a *reservation station*. The use of reservation station decouples instruction decoding and instruction execution, and provides a buffer to take up the slack between decoding and execution stages due to the temporal variation of throughput rates in the two stages. This eliminates unnecessary stalling of the decoding stage and prevents unnecessary starvation of the execution stage.

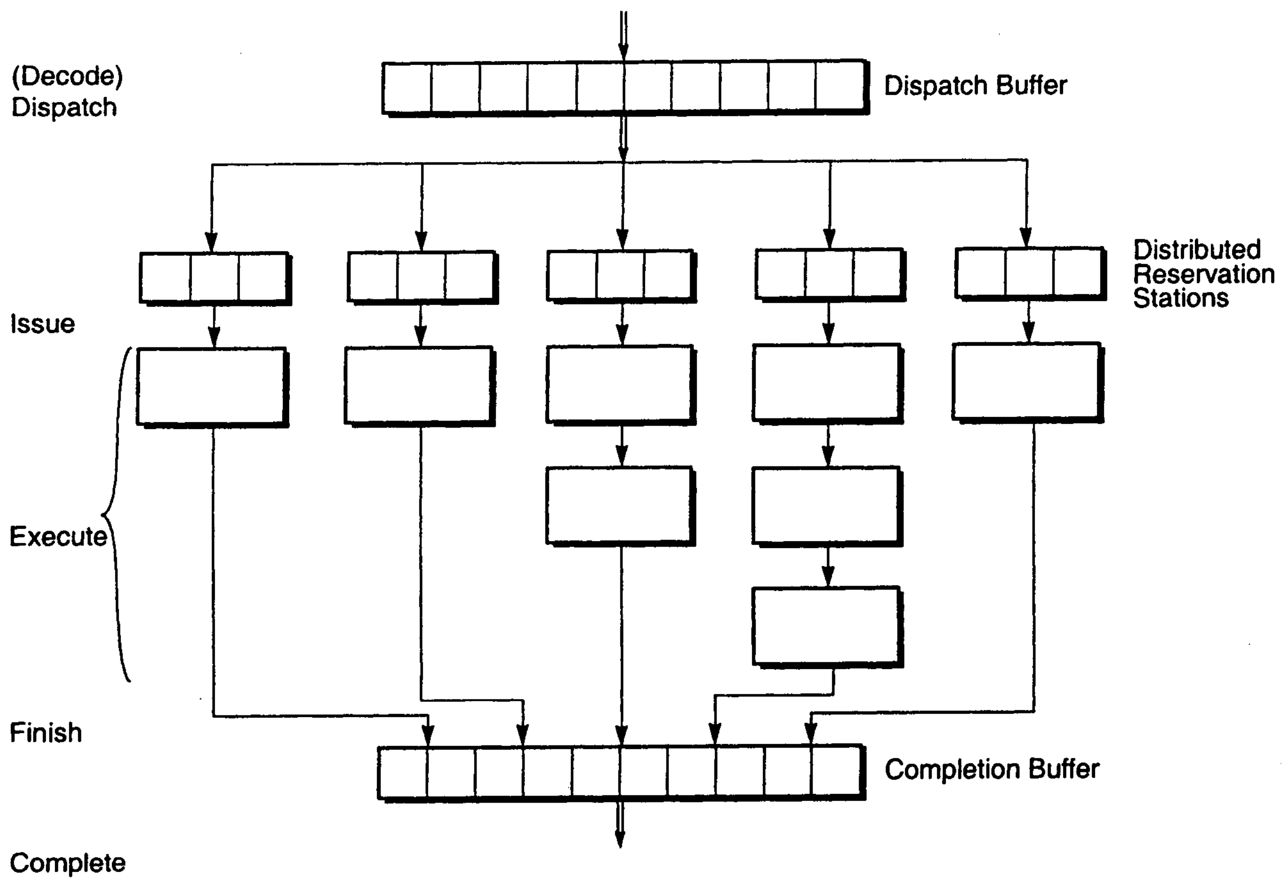
Based on the placement of the reservation station relative to instruction dispatching, two types of reservation station implementations are possible. If a single buffer is used at the source side of dispatching, we identified this as a *centralized reservation station.* On the other hand if multiple buffers are placed at the destination side of dispatching, they are identified as *distributed reservation stations*. Figure 17 and Figure 18 illustrate the two ways of implementing reservation stations.

FIGURE 17 Centralized reservation station.



The Intel Pentium Pro implements a centralized reservation station [ ]. In such an implementation, one reservation station with many entries feeds all the functional units. Instructions are dispatched from this centralized reservation station directly to all the functional units to begin execution. On the other hand the PowerPC 620 employs distributed reservation stations [ ]. In this implementation, each functional unit has its own reservation station on the input side of the unit. Instructions are dispatched to the individual reservation stations based on instruction type. These instructions remain in these reservation stations until they are ready to be issued into the functional units for execution. Of course, these two implementations of reservation stations represent only the two extreme alternatives. Hybrids or compromises of these two approaches are also possible. For example, the MIPS R10000 employs one such hybrid implementation [ ]. We identified such hybrid implementations as *clustered reservation stations*. With clustered reservation stations, instructions are dispatched to multiple reservation stations, and each reservation station can feed or be shared by more than one functional unit. Typically the reservation stations and functional units are clustered based on instruction or data types.

FIGURE 18 Distributed reservation stations.



Reservation station design involves certain trade-offs. A centralize reservation station allows all instruction types to share the same reservation station and will likely achieve the best overall utilization of all the reservation station entries. However a centralized implementation can incur the most complexity in its hardware design. It requires centralized control and a buffer that is highly multi-ported to allow multiple concurrent accesses. Distributed reservation stations can be single-ported buffers, each with only a small number of entries. However, each reservation station's idling entries cannot be used by instructions destined for execution in other functional units. The overall utilization of all the reservation station entries will lower. It is also likely that one reservation station can saturate when all of its entries are occupied and hence induce stalls in instruction dispatching.

With the different alternatives for implementing reservation stations, we need to clarify our use of certain terms. In this book the term "dispatching" implies the associating of instruction types with functional unit types after instructions have been decoded. On the other hand, the term "issuing" always means the initiation of execution in functional units. In a distributed reservation station design, these two events occur separately. Instructions are *dispatched* from the centralized decode/dispatch buffer to the individual reservation stations first, and when all their operands are available, they are then *issued* into the individual functional units for execution. With a centralized reservation station, the dispatching of instructions from the centralized reservation station does not occur until all their operands are ready. All instructions, regardless of types, are held in the centralized reservation station until they are ready to execute, at which time instructions are *dispatched* directly into the individual functional units to begin execution. Hence, in a machine with a centralized reservation station, the associating of instructions to individual functional units occurs at the same time as when their execution is initiated. Therefore, with a centralized reservation station, instruction *dispatching* and instruction *issuing* occur at the same time, and these two terms become interchangeable. This is also illustrated in Figure 17.

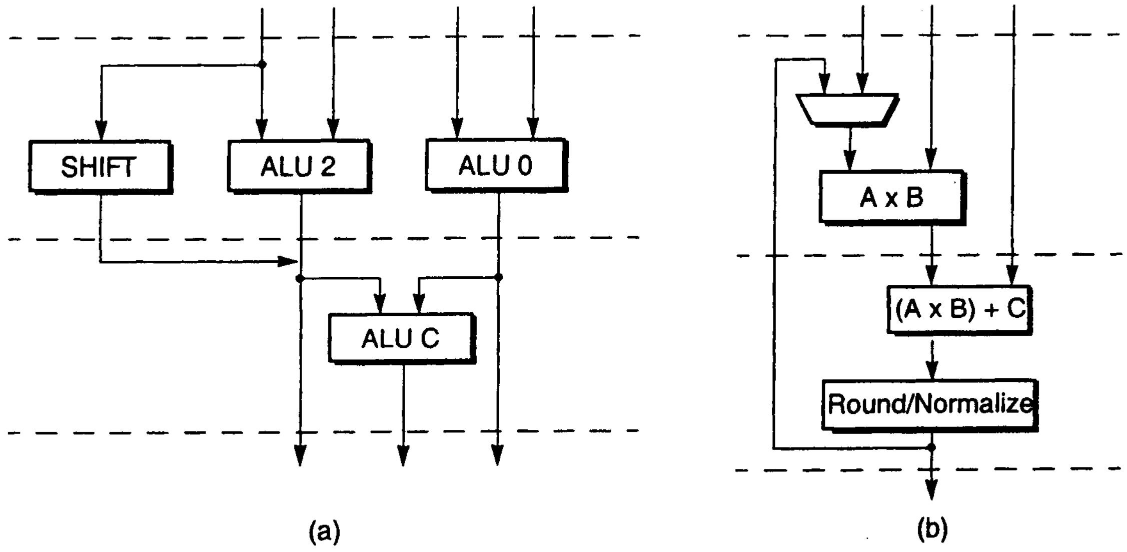
3.1.3.4 Instruction Execution

Instruction execution stage is the heart of a superscalar machine. The current trend in superscalar pipeline design is towards more parallel and more diversified pipelines. This translates into having more functional units and having these functional units be more specialized. By specializing them for executing specific instruction types, these functional units can be more performance efficient. Early scalar pipelined processors have essentially one functional unit. All instruction types (excluding floating-point instructions that are executed by a separate floating-point coprocessor chip) are executed by the same functional unit. In the TYP pipeline example, this functional unit is a 2-stage pipelined unit consisting of the ALU and MEM stages of the TYP pipeline. Most first-generation superscalar processors are parallel pipelines with two diversified functional units, one executing integer instructions and the other floating-point instructions. These early superscalar processors simply integrated floating-point execution in the same instruction pipeline instead of employing a separate coprocessor unit.

Current superscalar processors usually employ multiple integer units and some have multiple floating-point unit. These are the two most fundamental functional unit types. Some of these units are becoming quite sophisticated and capable of executing more than one operation involving more than two source operands in each cycle. Figure 19 (a) illustrates the integer execution unit of the TI SuperSPARC [up.rep.12/4/91] which contains a cascaded ALU configuration.

Three ALU's are included in this 2-stage pipelined unit, and up to two integer operations can be issued into this unit in one cycle. If they are independent, then both operations are executed in the first stage using ALU0 and ALU2. If the second operation depends on the first, then the first one is executed in ALU2 during the first stage with the second one executed in ALUC in the second stage. Implementing such a functional unit allows more cycles in which two instructions are simultaneously issued.

FIGURE 19 (a) Integer functional unit in the TI SuperSPARC; (b) Floating-point unit in the IBM RS/6000.



The floating-point unit in the IBM RS/6000 [JRD,Jan'90pg.60] is implemented as a 2-stage pipelined MAF (multiply-add-fused) unit that takes three inputs (A,B,C) and performs (AxB)+C. This is illustrated in Figure 19 (b). The MAF unit is motivated by the most common use of floating-point multiplication to carry out the dot-product operation, i.e. D=(AxB)+C. If the compiler is able to merge many multiply-add pairs of instructions into single MAF instructions, and the MAF unit can sustain the issuing of one MAF instruction in every cycle, then an effective throughput of two floating-point instructions per cycle can be achieved using only one MAF unit. The normal floating-point multiply instruction is actually executed by the MAF unit as (AxB)+0, while the floating-point add instruction is performed by the MAF unit as (Ax1)+C. Since the MAF unit is pipelined, even without executing MAF instructions, it can still sustain an execution rate of one floating-point instruction per cycle.

In addition to executing integer ALU instructions, an integer unit can also be used for generating memory addresses and executing branch and load/store instructions. However in most recent designs separate branch and load/store units have been incorporated. The branch unit is responsible for updating the PC, while the load/store unit is directly connected to the D-cache. Other specialized functional units are emerging for supporting graphics and image processing applications. For example in the Motorola 88110 there is a dedicated functional unit for bit manipulation and two functional units for supporting pixel processing [???]. For many of the signal processing and multimedia applications, the common data type is a byte. Frequently four bytes are packed into a 32-bit word for simultaneous processing by specialized 32-bit functional units for increased throughput. In the TriMedia VLIW processor [???] intended for such applications, such functional units are employed. For example, the TriMedia-1 processor can execute the *quadavg* instruction in one cycle. The *quadavg* instruction sums four rounded averages and is quite useful in MPEG decoding for decompressing compressed video images; it carries out the following computation.

$$quadavg = \frac{(a+e+1)}{2} + \frac{(b+f+1)}{2} + \frac{(c+g+1)}{2} + \frac{(d+h+1)}{2} \qquad \text{(EQ 2)}$$

The eight variables denote eight byte operands with a, b, c, and d stored as one 32-bit quantity and e, f, g, and h stored as another 32-bit quantity. The functional unit takes as input these two 32-bit operands and produces the *quadavg* result in one cycle. This single cycle operation replaces numerous add and divide instructions that would have been required if the eight single byte operands were manipulated individually. With the expected widespread deployment of multimedia applications, more of such specialized functional units that operate on packed-pixel data types will emerge.

What is the best mix of functional units for a superscalar pipeline is an interesting question. Clearly the answer is application domain dependent. If we use the statistics from the previous chapter of typical programs having 40% ALU instructions, 20% branches, and 40% load/store instructions, then we can have a 4-2-4 rule of thumb. For every 4 ALU units, we should have 2 branch units and 4 load/store units. Many of the current leading superscalar processors have four or more ALU type functional units (including both integer and floating-point units). Most of them have only one branch unit, but are able to speculate beyond one conditional branch instruction. However most of these processors have only one load/store unit; some are able to process two load/store instructions in every cycle with some constraints. Clearly there seems be an imbalance in having too few load/store units. The reason is that implementing multiple load/store units that operate in parallel in accessing the same D-cache is a difficult task. It requires the D-cache being multi-ported. Multi-ported memory modules involve very complex circuit design and can significantly slow down the memory speed. In many designs multiple memory banks are used to simulate a truly multi-ported memory. A memory is partitioned into multiple banks. Each bank can perform a read/write operation is a machine cycle. If the effective addresses of two load/store instructions happen to reside on different banks, then both instructions can be carried out by the two different banks at the same time. However, if there is a bank conflict, then the two instructions must be serialized. Multi-banked D-caches have been used to simulate multi-ported D-caches. For example the Intel Pentium processor uses an 8-banked D-cache to simulate a 2-ported D-cache [ ]. A truly multi-ported memory can guarantee conflict-free simultaneous accesses. Typically more read ports than write ports are needed. Multiple read ports can be implemented by having multiple copies of the memory. All memory writes are broadcast to all the copies, with all the copies containing identical content. Each copy can provide small number of read ports with the total number of read ports being the sum of all the read ports on all the copies. For example, a memory with four read ports and two write ports can be implemented as two copies of simpler memory modules each with only two write ports and two read ports. Implementing multiple, especially more than two, load/store units to operate in parallel is currently a challenge in designing wide superscalar pipelines.

The amount of resource parallelism in the instruction execution portion is determined by the combination of *spatial* and *temporal* parallelisms. Having multiple functional units is a form of spatial parallelism. Alternatively, parallelism can be obtained via pipelining of these functional units, which is a form of temporal parallelism. For example, instead of implementing a dual-ported D-cache, in some current designs D-cache access is pipelined into two pipeline stages so that two load/store instructions can be concurrently serviced by the D-cache. Currently, there is a general trend towards implementing deeper pipelines in order to reduce the cycle time and increase the clock speed. Spatial parallelism also tends to require more hardware complexity and silicon real estate. Temporal parallelism makes more efficient use of hardware but does increase the overall instruction processing latency and potentially pipeline stall penalties due to inter-instruction dependences.

In real superscalar pipeline designs we often see that the total number of functional units exceeds the actual width of the parallel pipeline. Typically the width of a superscalar pipeline is determined by the number of instructions that can be fetched, decoded or completed in every machine cycle. However, due to the dynamic variation of instruction mix and the resultant nonuniform distribution of instruction mix during program execution, on a cycle by cycle basis there is a potential dynamic mismatch of instruction mix and functional unit mix. The former varies in time and the later stays fixed. Due to the specialization and heterogeneity of the functional units the total number of functional units must exceed the width of the superscalar pipeline to avoid having the instruction execution portion become the bottleneck due to excessive structural dependences related to the unavailability of certain functional unit types. Some of the aggressive compiler back-ends actually try to smooth out this dynamic variation of instruction mix to ensure better sustained match with the functional unit mix. Of course, different application programs can exhibit different inherent overall mix of instruction types. The compiler can only make localized adjustments to achieve some performance gain. Studies have been done in assessing the best number and mix of functional units based on SPEC benchmarks [ISCA-95,pg.117].

With large number of functional units, there is additional hardware complexity other than the functional units themselves. Results from the outputs of functional units need to be forwarded to inputs of the functional units. A multiplicity of busses are required, and potentially logic for bus control and arbitration is needed. Usually a full crossbar interconnection network is too costly and not absolutely necessary. The mechanism for routing operands between functional units introduces another form of structural dependence. The interconnect mechanism also contributes to the latency of the execution stage(s) of the pipeline. In order to support data forwarding the reservation station(s) must monitor the buses for tag matches, indicating the availability of needed operands, and latch in the operands when they are broadcasted on the buses. Potentially the complexity of the instruction execution stage can grow at the rate of n^2 where n is the total number of functional units.

3.1.3.5 Instruction Completion and Retiring

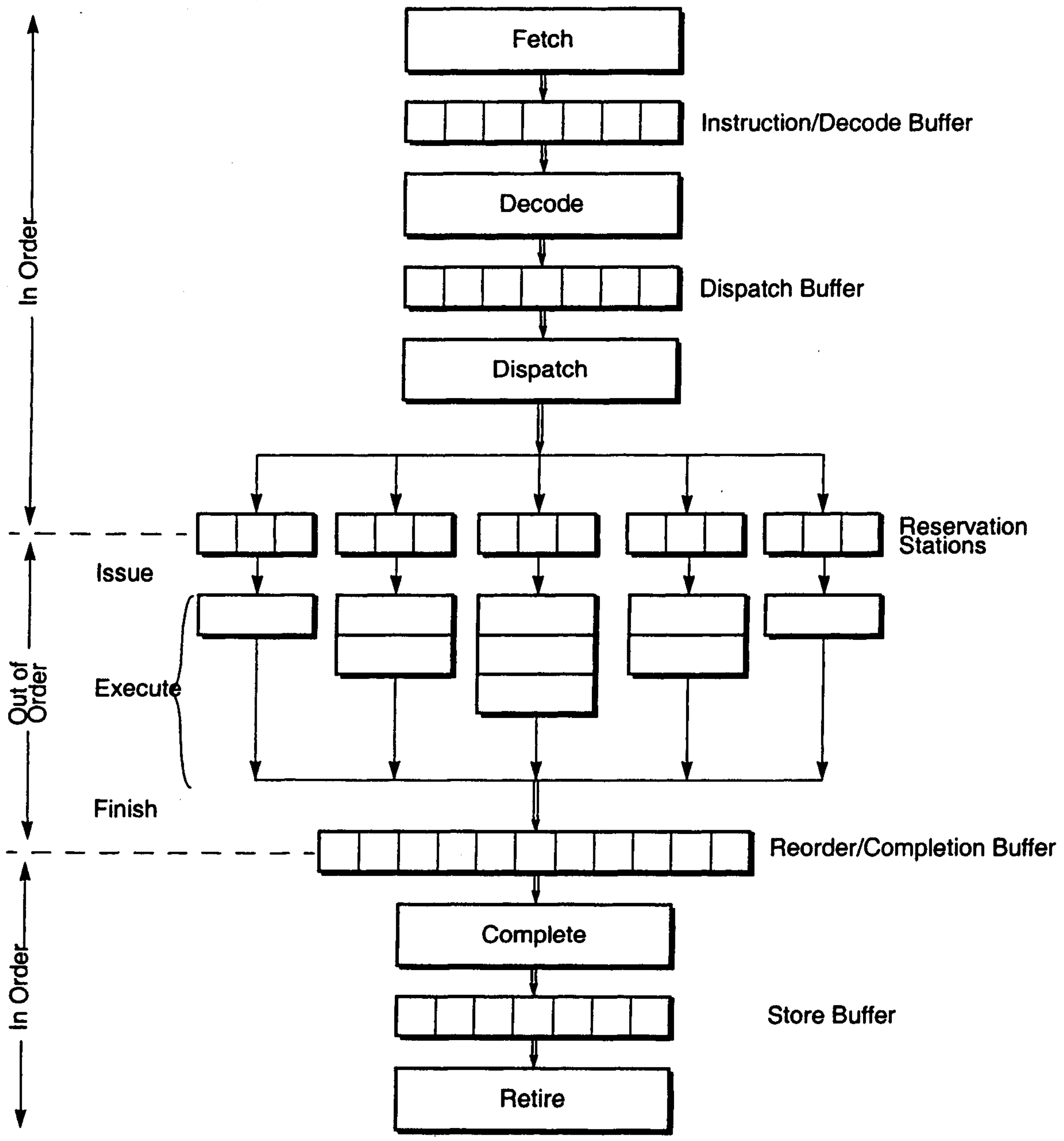
An instruction is considered *completed* when it finishes execution and updates the machine state. An instruction finishes execution when it exits the functional unit and enters the completion buffer. Subsequently it exits the completion buffer and becomes completed. When an instruction finishes execution its result may only reside in nonarchitected buffers. However, when it is completed its result is written into an architecture register. With instructions that actually update memory locations, there can be a time period between when they are architecturally completed and the memory locations being updated. For example a store instruction can be architecturally completed when it exits the completion buffer and enters the store buffer to wait for the availability of a bus cycle in order to write to the D-cache. This store instruction is considered *retired* when it exits the store buffer and updates the D-cache. Hence, in this book instruction *completion* involves the updating of the machine state, whereas instruction *retiring* involves the updating of the memory state. For instructions that do not update the memory, retiring occurs at the same time as completion. So, in a distributed reservation station machine, an instruction can go through the following phases: *fetch*, *decode*, *dispatch*, *issue*, *execute*, *finish*, *complete*, and *retire*. Issuing and finishing simply refer to starting execution and ending execution, respectively. Some of the superscalar processor vendors use these terms in slightly different ways. Frequently, dispatching and issuing are used almost interchangeably, similarly with completion and retiring. Sometimes completion is used to mean finishing execution and retiring is used to mean updating the machine's architectural state. There is yet no standardization on the use of these terms.

In order to achieve better instruction throughput, superscalar processors employ dynamic pipelines that facilitate out-of-order execution of instructions. However, out-of-order execution introduces a potential problem. During the execution of a program, interrupts and exceptions can occur. *Interrupts* are usually induced by the external environment such as I/O devices. These occur in an asynchronous fashion with respect to the program execution. *Exceptions* are induced by the execution of the instructions of the program. An instruction can induce an exception due to arithmetic operations, such as dividing by zero, and floating-point overflow or underflow. When such exceptions occur the results of the computation may no longer be valid. Exceptions can also occur due to the occurrence of page faults in a paging based virtual memory system. Such exceptions can occur when instructions reference the memory. When these exceptions occur a new page must be brought in from secondary storage which can require on the order of thousands of machine cycles. Consequently, the execution of the program that induced the page fault is usually suspended and the execution of a new program is initiated. This involves the operating system in performing a context swap. When such exceptions occur it is important that the architectural state of the machine present at the time the excepting instruction is executed is saved so that the program can resume execution from that state after the exception is serviced.

Machines that are capable of supporting this suspension and resumption of execution of a program is said to have *precise exception.* In order to support precise exception the superscalar processor must maintain its architectural state and evolve this machine state in such a way as if the instructions in the program are executed one at a time according to the original program order. The reason being that when an exception occurs, the state the machine is in at that time must reflect the condition that all instructions preceding the excepting instructions have completed while no instructions following the excepting instruction has completed. For a dynamic pipeline to have precise exception this sequential evolving of the architectural state of the machine must be maintained even though instructions are actually executed out of program order.

In a dynamic pipeline instructions are fetched and decoded in program order, but are executed out of program order. Essentially, instructions can enter the reservation station(s) in order but exit the reservation station(s) out of order. They also finish execution out of order. To support precise exceptions, instruction completion must occur in program order so as to update the architectural state of the machine in program order. In order to accommodate out of order finishing of execution and in order completion of instructions, a *reorder buffer* is needed in the instruction completion stage of the parallel pipeline. As instructions finish execution they enter the reorder buffer out of order, but they exit the reorder buffer in program order. As they exit the reorder buffer they are considered completed. This is illustrated in Figure 20 with the reservation station and the reorder buffer bounding the out of order region of the pipeline or essentially the instruction execution portion of the pipeline. The terms adopted in this book, referring to the various phases of instruction processing, are illustrated in Figure 20.

FIGURE 20 A dynamic pipeline with reservation station and reorder buffer.



Fetch
Instruction/Decode Buffer
Decode
Dispatch Buffer
Dispatch
Reservation Stations
Issue
Execute
Finish
Reorder/Completion Buffer
Complete
Store Buffer
Retire
In Order
Out of Order
In Order