the register file occur in the next stage. Instructions then enter the execution pipelines in the functional units. The results are written to the register file in the write-back stage. Within a functional unit, the results are bypassed to the next instructions as soon as available. The results of unit FU0 are forwarded to units FU2 and FU3 with one clock cycle delay and are visible to the unit FU1 in the next clock cycle. Similarly, the results of unit FU1 are forwarded to unit FU0 without any delay.

5.4.8.4. Explicitly Parallel Instruction Computing

The *Explicitly Parallel Instruction Computing* (EPIC) architecture was developed at Hewlett-Packard Laboratories, starting in 1989, to enable higher levels of instruction-level parallelism without unacceptable hardware complexity. The EPIC architecture is an evolution of the VLIW architecture, which has also assimilated many concepts of the superscalar technique. EPIC provides a philosophy to build ILP processors, along with a set of architectural features that support this philosophy. The first commercial implementation of an EPIC architecture is Intel's IA-64 architecture.

EPIC Philosophy

One of the goals of the EPIC project was to retain the VLIW concept of statically constructing the plan of execution of the program by the compiler, but to enhance it with features specific to superscalar processing, that use the advantages of dynamic factors. To accomplish these goals, the EPIC philosophy has three main principles:

- The compiler should have the key role in constructing the plan of execution for the program, and the architecture should provide the required support for the compiler.
- The architecture should provide features that assist the compiler in using the advantages of ILP.
- The architecture should provide mechanisms to communicate the compiler's plan of execution to the processor.

The EPIC philosophy assigns the charge of constructing the plan of execution to the compiler. EPIC processors provide features that assist the compiler in successfully constructing the plan of execution. A basic requirement is that the behavior of EPIC processors be predictable and controllable from the compiler's viewpoint. The essence of constructing the plan of execution at compile time is to reorder the original sequential code to take advantage of the application's parallelism and to use in an optimal way the hardware resources, thereby minimizing the execution time. Without suitable architectural support, reordering can affect program correctness. For this reason, EPIC processors provide architectural features that allow extensive code reordering at compile time.

An EPIC compiler faces a major problem in constructing the plan of execution at compile time, because certain types of information can only be known at run-time. For example, the compiler cannot know the destination of each conditional branch. It is often impossible to construct a static schedule that optimizes all paths within the program. Ambiguity also results when the compiler is unable to resolve whether two memory references are to the same location. If they are, they need to be executed sequentially. If not, they can be scheduled in any order. In such circumstances, the compiler constructs and optimizes the plan of execution for the likely case. However, EPIC provides architectural features, such as control and data speculation, to ensure program correctness even when the guess is incorrect. Sometimes, a performance reduction results, but these situations should be infrequent.

Having constructed a plan of execution, the compiler needs to communicate this plan to the processor. In order to do so, the instruction set architecture must be able to express the compiler's decisions about the moment when to issue each operation and which resources to use. In particular, it should be possible to specify which operations are to be issued simultaneously. The alternative would be to create a sequential program which is presented to the processor and re-organized dynamically. When communicating the plan of execution to the processor, it is important to provide critical information at the appropriate time. An example is the branch operation which, if it is going to be taken, requires that instructions start being fetched from the branch target well in advance of the branch being issued. Rather than providing a branch target buffer to indicate what the target address is, the EPIC philosophy is to provide this information to the processor, explicitly and at the correct time, via the code.

There are other problems which must be solved by the microarchitecture that are not directly concerned with the execution of the code, but which do affect the execution time. One example is the management of the cache memory hierarchy and the associated decisions of what data to replace. The replacement policies are typically built into the cache memory hardware. EPIC allows the compiler to also manage these mechanisms of the microarchitecture. In conclusion, EPIC provides architectural features that permit programmatic control of the mechanisms which normally are controlled by the microarchitecture.

EPIC simplifies two major run-time responsibilities of the processor. First, the EPIC philosophy permits the elimination of run-time dependency checks for operations that the compiler has already demonstrated as independent. Second, EPIC permits the elimination of complex logic for issuing operations out of order by relying upon the issue order specified by the compiler. EPIC enhances the compiler's ability to statically generate a plan of execution by supporting various forms of compile-time code motion that would be illegal in a conventional architecture. Some of the architectural features of EPIC are described next.

Multiple Operations per Instruction (MultiOp)

A MultiOp instruction specifies a set of operations that are intended to be issued for execution simultaneously, each operation being the equivalent of an in-

struction of a conventional sequential processor. The compiler identifies operations scheduled to be issued for execution simultaneously and packages them into a MultiOp instruction. When issuing a MultiOp instruction, the processor does not need to perform dependence checking between its constituent operations. In addition, each MultiOp instruction has a notion of virtual time associated with it; a single MultiOp instruction is issued in each cycle of virtual time. Virtual time differs from actual time when the processor inserts run-time stalls, that the compiler did not anticipate.

In constructing a plan of execution, the compiler must take into account the number of resources of each type available in the processor, and it must perform resource allocation. The EPIC philosophy is to communicate these decisions to the processor via the code, so that the processor need not perform again the resource allocation at run-time. One way of achieving this is by using a positional instruction format, i.e., the position of an operation within the MultiOp instruction specifies the functional unit upon which the operation will execute. Alternatively, this information can be specified as part of each operation's opcode.

Architecturally Visible Latencies

Traditional sequential architectures define execution as a sequence of atomic operations; each operation completes before a subsequent operation begins. Such architectures do not allow the possibility of one operation's reads and writes being interleaved in time with those of other operations. With MultiOp instructions, operations are no longer atomic. When the operations within a single MultiOp instruction are executed, multiple operations may read their inputs before any operation writes its results. Thus, the non-atomicity and the latencies of operations are both architecturally visible.

The primary motivation for architecturally non-atomic operations is hardware simplicity for operations that, in reality, take more than one clock cycle to complete. If an operation will not attempt to use a result before it has been produced, the hardware does not need interlocks or a stall capability.

A non-atomic operation which produces at least one result with an architecturally assumed latency that is greater than one cycle is termed a *non-unit assumed latency* (NUAL) operation. A non-atomic operation which has architecturally assumed latencies of one clock cycle for all of its results is termed a *unit assumed latency* (UAL) operation. Assumed latencies can be specified as constants recognized by the EPIC compiler and the EPIC processor, or they may be specified dynamically by the program prior to or during execution. The processor then uses the assumed latency specification to ensure correct program interpretation. If, for any reason, the processor's actual latencies differ from the assumed latencies, the processor must ensure correct program execution, using specific techniques.

Resolving the Branch Problem

Many applications use a large number of branches. Branch latency as measured in processor cycles grows as clock frequency increases. Branch operations have a

hardware latency which extends from the time when the branch begins execution to the time when the instruction at the branch target begins execution. During this time, several actions occur: a branch condition is computed, a target address is formed, instructions are fetched from either the sequential or the target path, and the next instruction is decoded and issued for execution.

The fundamental problem is that although conventional instruction sets specify a branch as a single, atomic operation, its actions must actually be performed at different times, which increases the latency of the branch. When an insufficient number of operations are overlapped with branch execution, unsatisfactory performance results. EPIC's philosophy is to eliminate stall cycles by achieving a better overlap between branch processing and other computation. Rather than relying on hardware alone to solve the problem, EPIC provides architectural features which facilitate the following three capabilities:

- Separate branch component operations, by an explicit specification in the code when each of the actions of the branch must take place;
- Elimination of branches, especially those for which accurate prediction is difficult;
- Static motion of operations across multiple branches.

EPIC does not treat a branch as an atomic operation. Rather than viewing a branch as an single high-latency operation, EPIC unbundles branches into three distinct operations: a prepare-to-branch operation determines the target address and provides it to the branch unit; a compare computes the branch condition; and finally the actual branch marks the location in the instruction stream where control flow is conditionally transferred.

The unbundling of branches allows the compiler to move the prepare-to-branch and the compare operations sufficiently in advance of the actual branch so that the processor can finish computing the branch condition and prefetching the appropriate instructions by the time that the actual branch is reached. On executing the prepare-to-branch operation, the processor can speculatively – before the branch condition is known – prefetch instructions at the branch target. After executing the compare, the processor can determine whether the branch will be taken, dismiss unnecessary instructions prefetched speculatively, and also initiate non-speculative prefetches of instructions. These techniques permit overlapped processing of branch components, while using only the static motion of branch components.

EPIC reduces the branch penalty by eliminating branches using *predicated execution.* Predicated execution refers to the conditional execution of operations based on a Boolean-valued input, called a predicate, associated with the basic block containing the operation. Compare operations compute the predicates such that they are true if the program would reach the corresponding basic block in the control flow graph; predicates are false otherwise. The semantics of the following generic operation guarded by predicate *p*,

$$r_1 = \text{op}\ (r_2, r_3)\ \text{if}\ p$$

are that the operation executes normally if *p* is true and the operation is nullified (i.e., has no effect on the architectural state) if *p* is false. In particular, the nullified operation does not modify any destination register or memory location, it does not signal any exceptions, and it does not branch. Predicated execution is often a more efficient method for controlling execution than branching and it provides additional freedom for static code motion.

Predicated execution is used by the compiler for a technique known as *if-conversion*. A simple example of *if*-conversion is shown in Figure 5.23. Figure 5.23(a) shows the control flow graph for an *if-then-else* construct, while Figure 5.23(b) shows the resulting converted code. A single compare computes complementary predicates, each of which guards operations in one of the conditional clauses. The code for which *if*-conversion was applied contains no branches and is easily scheduled in parallel with other code, often substantially enhancing the instruction-level parallelism.

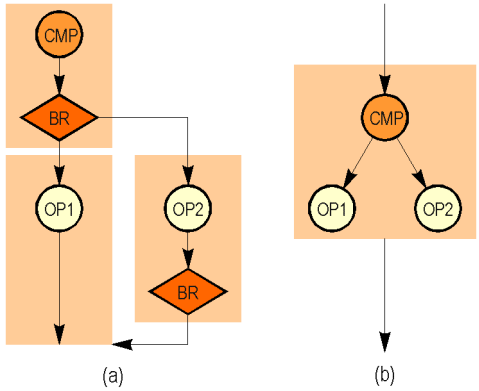

Figure 5.23. Use of predicated execution to perform if-conversion.

Branches represent obstacles to the static reordering of operations needed to perform efficient schedules. In addition to predicated execution, EPIC provides another technique that increases operation mobility across branches: *control speculation*. Consider the program fragment shown in Figure 5.24(a), which consists of two basic blocks. Control speculation is shown in Figure 5.24(b); OP1 has been moved from the second basic block into the first to reduce the length of the path which is dependent upon the branch. The label OP1* is attached to the operation to indicate that it needs a speculative operation code.

While static speculation enhances instruction-level parallelism, it also requires hardware assistance to handle exceptions. If an operation reports a speculative exception immediately, the exception may be false. This would occur if OP1* reports an exception, and the subsequent branch is taken. The exception is false because it is reported even though OP1 would not have been executed in the original program of Figure 5.24(a).

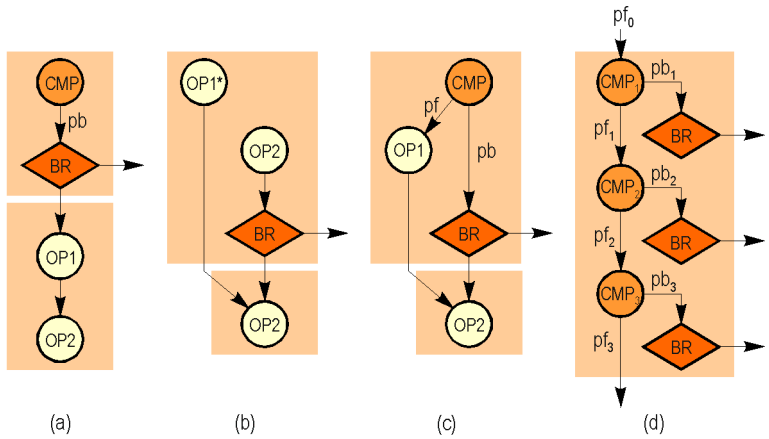

Figure 5.24. Example of code motion above one or more branches.

EPIC avoids false exceptions using speculative opcodes and tagged operands. When a speculative operation such as OP1* in Figure 5.24(b) causes an exception, the operation does not report the exception immediately, but it generates an operand that is tagged as erroneous. The exception is reported later, when a non-speculative operation uses the erroneous operand. If the branch is not taken, OP2 correctly reports the exception generated by OP1*. If the branch is taken, the erroneous operand is ignored and the exception is not reported.

An EPIC processor does not execute operations like branches or stores to memory speculatively, because these can cause side effects that are not easily eliminated. Instead, EPIC uses predicated execution to allow operations to move across branches non-speculatively. Figure 5.24(c) shows again the motion of OP1 across a branch. However, in this case, OP1 remains non-speculative because it is guarded using a predicate corresponding to the complement of the branch exit condition ($pf = \overline{pb}$). As in the original program, OP1 executes only if the branch is not taken.

EPIC allows non-speculative code motion across multiple branches. The compiler can cascade compare operations that compute predicates across multiple branches, as shown in Figure 5.24(d). Each compare evaluates an exit condition and computes predicates for a branch (pb_i) and for the basic block reached when the branch is not taken (pf_i). A branch predicate (pb_i) is true when its basic block predicate is true (pf_{i-1}) and its exit condition is true. A subsequent basic block predicate (pf_i) is true when the previous basic block's predicate is true (pf_{i-1}) and the branch exit condition is false. Such a predicate, which is computed over a multiblock region, is called a *fully resolved predicate* (FRP). Using FRPs, branches and other non-speculative operations are easily reordered to optimize the schedule.

Resolving the Memory Access Problem

Memory accesses can also reduce performance. Since the processor's clock period is decreasing faster than the memory access time, the memory access time (measured in processor cycles) is increasing. Data cache memories can reduce performance degradation due to increasing main memory latency. However, hardware-managed cache memories sometimes degrades performance even below that of a system without a cache memory. EPIC provides architectural mechanisms that allow compilers to explicitly control the motion of data through the cache memory hierarchy.

Unlike other operations, a load operation can introduce several different latencies, depending on the cache memory level at which the referenced datum is found. For NUAL loads, the compiler must communicate to the processor the specific latency that is assumed for each load. For this purpose, EPIC architectures provide load operations with a source *cache memory specifier* that the compiler uses to inform the processor of where within the cache memory hierarchy it can expect to find the referenced datum and, implicitly, the assumed latency. In order to generate a high quality schedule, the compiler must perform a correct prediction of the latency of each load operation and then communicate this to the processor using the cache memory specifier.

EPIC architectures also provide the load and store operations with a target cache memory specifier that the compiler uses to indicate the cache memory level to which the load or store operation should promote or demote the referenced data for use by subsequent memory operations. The target cache memory specifier reduces misses in the cache memories by controlling their contents. The compiler can exclude data with insufficient temporal locality from a certain level of cache memory, and can remove data from this cache memory on last use.

5.4.8.5. Comparison of Throughput Improvement Methods

A comparison of the throughput improvement methods presented earlier shows a few interesting differences. For example, the superscalar and VLIW approaches are more sensitive to resource conflicts than the superpipelined approach. In a superscalar or VLIW processor, a resource must be duplicated to reduce the chance of conflicts, while the superpipelined technique avoids any resource conflicts. An EPIC architecture tries to avoid resource conflicts using the compiler.

To prevent a superpipelined processor from being slower than a superscalar processor, the technology used in the superpipelined processor must reduce the delay of the lengthy instruction pipeline. Therefore, in general, superpipelined processors require faster transistor technology such as GaAs (gallium arsenate), while superscalar processors require a higher number of transistors to allow the duplication of hardware resources. The superscalar approach often uses CMOS technology, since this technology provides high circuit density. Existing technology generally favors an increasing circuit density over an increasing circuit speed. Historically, circuit density has in-