The Microarchitecture of Superscalar Processors

James E. Smith
Department of Electrical and Computer Engineering
1415 Johnson Drive
Madison, WI 53706
ph: (608)-265-5737
fax:(608)-262-1267
email: jes@ece.wisc.edu

Gurindar S. Sohi
Computer Sciences Department
1210 W. Dayton
Madison, WI 53706
ph: (608)-262-7985
email: sohi@cs.wisc.edu

Aug. 20, 1995

Abstract

Superscalar processing is the latest in a long series of innovations aimed at producing ever-faster microprocessors. By exploiting instruction-level parallelism, superscalar processors are capable of executing more than one instruction in a clock cycle. This paper discusses the microarchitecture of superscalar processors. We begin with a discussion of the general problem solved by superscalar processors: converting an ostensibly sequential program into a more parallel one. The principles underlying this process, and the constraints that must be met, are discussed. The paper then provides a description of the specific implementation techniques used in the important phases of superscalar processing. The major phases include: i) instruction fetching and conditional branch processing, ii) the determination of data dependences involving register values, iii) the initiation, or issuing, of instructions for parallel execution, iv) the communication of data values through memory via loads and stores, and v) committing the process state in correct order so that precise interrupts can be supported. Examples of recent superscalar microprocessors, the MIPS R10000, the DEC 21164, and the AMD K5 are used to illustrate a variety of superscalar methods.

1. Introduction

Superscalar processing, the ability to initiate multiple instructions during the same clock cycle, is the latest in a long series of architectural innovations aimed at producing ever faster microprocessors. Introduced at the beginning of this decade, superscalar microprocessors are now being designed and produced by all the microprocessor vendors for high-end products. Although viewed by many as an extension of the Reduced Instruction Set Computer (RISC) movement of the 1980s, superscalar implementations are in fact heading toward increasing complexity. And superscalar methods have been applied to a spectrum of instruction sets, ranging from the DEC Alpha, the "newest" RISC instruction set, to the decidedly non-RISC Intel x86 instruction set.

A typical superscalar processor fetches and decodes the incoming instruction stream several instructions at a time. As part of the instruction fetching process, the outcomes of conditional branch instructions are usually predicted in advance to ensure an uninterrupted stream of instructions. The incoming instruction stream is then analyzed for data dependences, and instructions are distributed to functional units, often according to instruction type. Next, instructions are initiated for execution in parallel, based primarily on the availability of operand data, rather than their original program sequence. This important feature, present in many superscalar implementations, is referred to as *dynamic instruction scheduling*. Upon completion, instruction results are re-sequenced so that they can be used to update the process state in the correct (original) program order in the event that an interrupt condition occurs. Because individual instructions are the entities being executed in parallel, superscalar processors exploit what is referred to as *instruction level parallelism* (ILP).

1.1. Historical Perspective

Instruction level parallelism in the form of pipelining has been around for decades. A pipeline acts like an assembly line with instructions being processed in phases as they pass down the pipeline. With simple pipelining, only one instruction at a time is initiated into the pipeline, but multiple instructions may be in some phase of execution concurrently.

Pipelining was initially developed in the late 1950s [8] and became a mainstay of large scale computers during the 1960s. The CDC 6600 [61] used a degree of pipelining, but achieved most of its ILP through parallel functional units. Although it was capable of sustained execution of only a single instruction per cycle, the 6600's instruction set, parallel processing units, and dynamic instruction scheduling are similar to the superscalar microprocessors of today. Another remarkable processor of the 1960s was the IBM 360/91 [3]. The 360/91 was heavily pipelined, and provided a dynamic instruction issuing mechanism, known as *Tomasulo's algorithm* [63] after its inventor. As with the CDC 6600, the IBM 360/91 could sustain only one instruction per cycle and was not superscalar, but the strong influence of Tomasulo's algorithm is evident in many of today's superscalar processors.

The pipeline initiation rate remained at one instruction per cycle for many years and was often perceived to be a serious practical bottleneck. Meanwhile other avenues for improving performance via parallelism were developed, such as vector processing [28, 49] and multiprocessing [5, 6]. Although some processors capable of multiple instruction initiation were considered during the '60s and '70s [50, 62], none were delivered to the market. Then, in the mid-to-late 1980s, superscalar processors began to appear [21, 43, 54]. By initiating more than one instruction at a time into multiple pipelines, superscalar processors break the single-instruction-per-cycle bottleneck. In the years since its introduction, the superscalar approach has become the standard method for implementing high performance microprocessors.

1.2. The Instruction Processing Model

Because hardware and software evolve, it is rare for a processor architect to start with a clean slate; most processor designs inherit a legacy from their predecessors. Modern superscalar processors are no different. A major component of this legacy is *binary compatibility*, the ability to execute a machine program written for an earlier generation processor.

When the very first computers were developed, each had its own instruction set that reflected specific hardware constraints and design decisions at the time of the instruction set's development. Then, software was developed for each instruction set. It did not take long, however, until it became apparent that there were significant advantages to designing instruction sets that were compatible with previous generations and with different models of the same generation [2]. For a number of very practical reasons, the

instruction set architecture, or binary machine language level, was chosen as the level for maintaining software compatibility.

The sequencing model inherent in instruction sets and program binaries, the *sequential execution model*, closely resembles the way processors were implemented many years ago. In the sequential execution model, a program counter is used to fetch a single instruction from memory. The instruction is then executed -- in the process, it may load or store data to main memory and operate on registers held in the processor. Upon completing the execution of the instruction, the processor uses an incremented program counter to fetch the next instruction, with sequential instruction processing occasionally being redirected by a conditional branch or jump.

Should the execution of the program need to be interrupted and restarted later, for example in case of a page fault or other exception condition, the state of the machine needs to be captured. The sequential execution model has led naturally to the concept of a *precise state*. At the time of an interrupt, a precise state of the machine (architecturally-visible registers and memory) is the state that would be present if the sequential execution model was strictly followed and processing was stopped precisely at the interrupted instruction. Restart could then be implemented by simply resuming instruction processing with the interrupted instruction.

Today, a computer designer is usually faced with maintaining binary compatibility, i.e., maintaining instruction set compatibility *and* a sequential execution model (which typically implies precise interrupts[1]). For high performance, however, superscalar processor implementations deviate radically from sequential execution -- much has to be done in parallel. As a result, the program binary nowadays should viewed as a specification of *what* has to be done, not *how* it is done in reality. A modern superscalar microprocessor takes the sequential specification as embodied in the program binary and removes much of the non-essential sequentiality to turn the program into a parallel, higher-performance version, yet the processor retains the outward *appearance* of sequential execution.

1.3. Elements of High Performance Processing

Simply stated, achieving higher performance means processing a given program in a smaller amount of time. Each individual instruction takes some time to fetch and execute; this time is the instruction's *latency*. To reduce the time to execute a sequence of instructions (e.g. a program), one can: (i) reduce individual instruction latencies, or (ii) execute more instructions in parallel. Because superscalar processor implementations are distinguished by the latter (while adequate attention is also paid to the former), we will concentrate on the latter method in this paper. Nevertheless, a significant challenge in superscalar design is to not *increase* instruction latencies due to increased hardware complexity brought about by the drive for enhanced parallelism.

Parallel instruction processing requires: the determination of the dependence relationships between instructions, adequate hardware resources to execute multiple operations in parallel, strategies to determine when an operation is ready for execution, and techniques to pass values from one operation to another. When the effects of instructions are committed, and the visible state of the machine updated, the appearance of sequential execution must be maintained. More precisely, in hardware terms, this means a superscalar processor implements:

i) Instruction fetch strategies that simultaneously fetch multiple instructions, often by predicting the outcomes of, and fetching beyond, conditional branch instructions,

ii) Methods for determining true dependences involving register values, and mechanisms for communicating these values to where they are needed during execution,

iii) Methods for initiating, or *issuing*, multiple instructions in parallel,

iv) Resources for parallel execution of many instructions, including multiple pipelined functional units and memory hierarchies capable of simultaneously servicing multiple memory references,

---

[1] Some recently-developed instruction sets relax the strictly sequential execution model by allowing a few exception conditions to result in an "imprecise" saved state where the program counter is inconsistent with respect to the saved registers and memory values.

v) Methods for communicating data values through memory via load and store instructions, and memory interfaces that allow for the dynamic and often unpredictable performance behavior of memory hierarchies. These interfaces must be well-matched with the instruction execution strategies.

vi) Methods for committing the process state in correct order; these mechanisms maintain an outward appearance of sequential execution.

Although we will discuss the above items separately, in reality they cannot be completely separated -- nor should they be. In good superscalar designs they are often integrated in a cohesive, almost seamless, manner.

1.4. Paper Overview

In section 2, we discuss the general problem solved by superscalar processors: converting an ostensibly sequential program into a parallel one. This is followed in section 3 with a description of specific techniques used in typical superscalar microprocessors. Section 4 focuses on three recent superscalar processors that illustrate the spectrum of current techniques. Section 5 presents conclusions and discusses future directions for instruction level parallelism.

2. Program Representation, Dependences and Parallel Execution

An application begins as a high level language program; it is then compiled into the *static machine level program*, or the *program binary*. The static program in essence describes a set of executions, each corresponding to a particular set of data that is given to the program. Implicit in the static program is the sequencing model, the order in which the instructions are to be executed. Fig. 1 shows the *assembly code* for a high level language program fragment. (The assembly code is the human-readable version of the machine code.) We will use this code fragment as a working example.

As a static program executes with a specific set of input data, the sequence of executed instructions forms a *dynamic* instruction stream. As long as instructions to be executed are consecutive, static instructions can be entered into the dynamic sequence simply by *incrementing* the program counter, which points to the next instruction to be executed. When there is a conditional branch or jump, however, the program counter may be *updated* to a non-consecutive address. An instruction is said to be *control dependent* on its preceding dynamic instruction(s), because the flow of program control must pass through preceding instructions first. The two methods of modifying the program counter -- incrementing and updating -- result in two types of control dependences (though typically when people talk about control dependences, they tend to ignore the former).

The first step in increasing instruction level parallelism is to overcome control dependences. Control dependences due to an incrementing program counter are the simplest, and we deal with them first. One can view the static program as a collection of *basic blocks*, where a basic block is a contiguous block of instructions, with a single entry point and a single exit point [1]. In the assembly code of Fig. 1, there are three basic blocks. The first basic block consists of the 5 instructions between the label L2 and the *ble* instruction, inclusive, the second basic block consists the 5 instructions between the *ble* instruction, exclusive, and the label L3, and the third basic block consists of the 3 instructions between the label L3 and the *blt* instruction, inclusive. Once a basic block has been entered by the instruction fetcher, it is known that all the instructions in the basic block will be executed eventually. Therefore, any sequence of instructions in a basic block can be initiated into a conceptual *window of execution*, en masse. We consider the window of execution to be the full set of instructions that may be simultaneously considered for parallel execution. Once instructions have been initiated into this window of execution, they are free to execute in parallel, subject only to data dependence constraints (which we will discuss shortly).

Within individual basic blocks there is some parallelism, but to get more parallelism, control dependences due to updates of the program counter, especially due to conditional branches, have to be overcome. A method for doing this is to *predict* the outcome of a conditional branch and *speculatively* fetch and execute instructions from the predicted path. Instructions from the predicted path are entered into the window of execution. If the prediction is later found to be correct, then the speculative status of the instructions is removed, and their effect on the state is the same as any other instruction. If the prediction is later found to be incorrect, the speculative execution was incorrect, and recovery actions must be initiated so that the architectural process state is not incorrectly modified. We will discuss branch prediction and speculative execution in more detail in Section 3. In our example of Fig. 1, the branch instruction (*ble*) creates a

```
for (i=0; i<last; i++) {
  if (a[i] > a[i+1]) {
    temp = a[i];
    a[i] = a[i+1];
    a[i+1] = temp;
    change++;
  }
}
```

(a)

```
L2:
   move    r3,r7      #r3->a[i]
   lw      r8,(r3)    #load a[i]
   add     r3,r3,4    #r3->a[i+1]
   lw      r9,(r3)    #load a[i+1]
   ble     r8,r9,L3   #branch a[i]>a[i+1]

   move    r3,r7      #r3->a[i]
   sw      r9,(r3)    #store a[i]
   add     r3,r3,4    #r3->a[i+1]
   sw      r8,(r3)    #store a[i+1]
   add     r5,r5,1    #change++
L3:
   add     r6,r6,1    #i++
   add     r7,r7,4    #r4->a[i]
   blt     r6,r4,L2   #branch i<last
```

(b)

Fig. 1. a) A high level language program written in C; b) Its compilation into a static assembly language program (unoptimized). This code is a part of a sort routine; adjacent values in array a[] are compared and switched if a[i] > a[i+1]. The variable `change` keeps track of the number of switches (if `change` = 0 at the end of a pass through the array, then the array is sorted.)

control dependence. To overcome this dependence, the branch could be predicted as not taken, for example, with instructions between the branch and the label L3 being executed speculatively.

Instructions that have been placed in the window of execution may begin execution subject to *data dependence* constraints. Data dependences occur among instructions because the instructions may access (read or write) the same storage (a register or memory) location. When instructions reference the same storage location, a *hazard* is said to exist -- i.e. there is the possibility of incorrect operation unless some steps are taken to make sure the storage location is accessed in correct order. Ideally, instructions can be executed subject only to *true dependence* constraints. These true dependences appear as *read-after-write*

(RAW) hazards, because the consuming instruction can only *read* the value *after* the producing instruction has *written* it.

It is also possible to have artificial dependences, and in the process of executing a program, these artificial dependences have to be overcome to increase the available level of parallelism. These artificial dependences result from *write-after-read (WAR)*, and *write-after-write (WAW)* hazards. A WAR hazard occurs when an instruction needs to write a new value into storage location, but must wait until all preceding instructions needing to read the old value have done so. A WAW hazard occurs when multiple instructions update the same storage location; it must appear that these updates occur in proper sequence. Artificial dependences can be caused in a number of ways, for example: by unoptimized code, by limited register storage, by the desire to economize on main memory storage, and by loops where an instruction can cause a hazard with itself.

Fig. 2 shows some of the data hazards that are present in a segment of our working example. The *move* instruction produces a value in register r3 that is used both by the first *lw* and *add* instructions. This is a RAW hazard because a true dependence is present. The *add* instruction also creates a value which is bound to register r3. Accordingly, there is a WAW hazard involving the *move* and the *add* instructions. A dynamic execution must ensure that accesses to r3 made by instructions that occur after the *add* in the program access the value bound to r3 by the *add* instruction and not the *move* instruction. Likewise, there is a WAR hazard involving the first *lw* and the *add* instructions. Execution must ensure that the value of r3 used in the *lw* instruction is the value created by the *move* instruction, and not the value created by the *add* instruction.

After resolving control dependences and artificial dependences, instructions are issued and begin execution in parallel. In essence, the hardware forms a *parallel execution schedule*. This schedule takes into account necessary constraints, such as the true dependences and hardware resource constraints of the functional units and data paths.

A parallel execution schedule often means that instructions complete execution in an order different from that dictated by the sequential execution model. Moreover, speculative execution means that some instructions may complete execution when they would not have executed at all had the sequential model been followed (i.e., when speculated instructions follow an incorrectly predicted branch.) Consequently, the architectural storage, (the storage that is outwardly visible) cannot be updated immediately when instructions complete execution. Rather, the results of an instruction must be held in a temporary status until the architectural state can be updated. Meanwhile, to maintain high performance, these results must be usable by dependent instructions. Eventually, when it is determined that the sequential model would have executed an instruction, its temporary results are made permanent by updating the architectural state. This process is typically called *committing* or *retiring* the instruction.

To summarize and to set the stage for the next section, Fig. 3 illustrates the parallel execution method used in most superscalar processors. Instructions begin in a static program. The instruction fetch process, including branch prediction, is used to form a stream of dynamic instructions. These instructions are

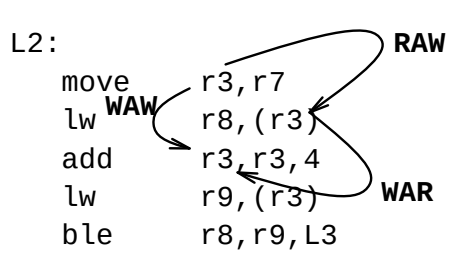


Fig. 2. Example of data hazards involving registers.

inspected for dependences with many artificial dependences being removed. The instructions are then dispatched into the window of execution. In Fig. 3, the instructions within the window of execution are no longer represented in sequential order, but are partially ordered by their true data dependences. Instructions issue from the window in an order determined by the true data dependences and hardware resource availability. Finally, after execution, instructions are conceptually put back into sequential program order as they are retired and their results update the architected process state.

3. The Microarchitecture of a Typical Superscalar Processor

Fig. 4 illustrates the microarchitecture, or hardware organization, of a typical superscalar processor. The major parts of the microarchitecture are: instruction fetch and branch prediction, decode and register dependence analysis, issue and execution, memory operation analysis and execution, and instruction reorder and commit. These phases are listed more-or-less in the same order as instructions flow through them; they will each be discussed in following subsections.

Keep in mind that underlying this organization there is a pipelined implementation where specific pipeline stages may or may not be aligned with the major phases of superscalar execution. The underlying pipeline stages are to some extent a lower-level logic design issue, depending on how much work is done in each pipeline stage. This will affect the clock period and causes some very important design tradeoffs regarding the degree of pipelining and the width of parallel instruction issue. However, we will not discuss these issues in detail here; discussions of these types of tradeoffs can be found in [13, 34, 38].

3.1. Instruction Fetching and Branch Prediction

The instruction fetch phase of superscalar processing supplies instructions to the rest of the processing pipeline. An *instruction cache*, which is a small memory containing recently-used instructions [52], is used in almost all current processors, superscalar or not, to reduce the latency and increase the bandwidth of the instruction fetch process. An instruction cache is organized into *blocks* or *lines* containing several consecutive instructions; the program counter is used to search the cache contents associatively to determine if the instruction being addressed is present in one of the cache lines. If so, there is a cache *hit*; if not, there is a *miss* and the line containing the instruction is brought in from main memory to be placed in the cache.

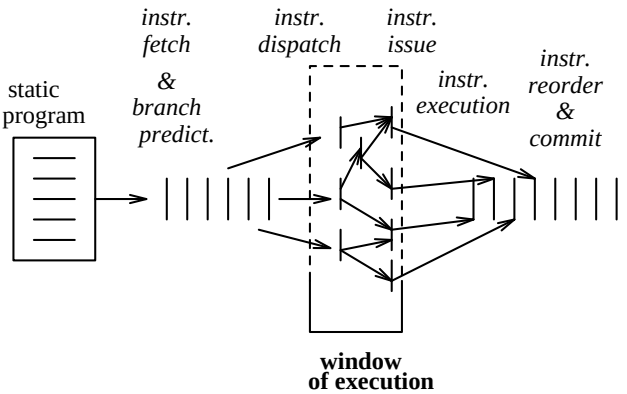


Fig. 3. A conceptual figure of superscalar execution. Processing phases are listed across the top of the figure.

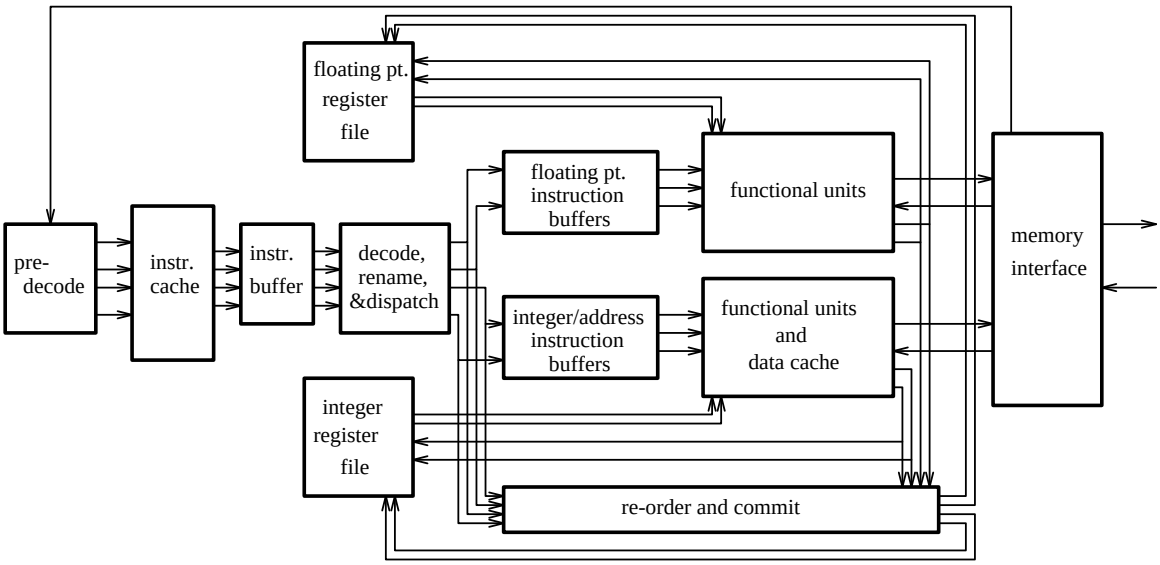


Fig. 4. Organization of a superscalar processor. Multiple paths connecting units are used to illustrate a typical level of parallelism. For example, four instructions can be fetched in parallel, two integer/address instructions can issue in parallel, two floating point instructions can complete in parallel, etc.

For a superscalar implementation to sustain the execution of multiple instructions per cycle, the fetch phase must be able to fetch multiple instructions per cycle from the cache memory. To support this high instruction fetch bandwidth, it has become almost mandatory to separate the instruction cache from the data cache (although the PowerPC 601[41] provides an interesting counter example).

The number of instructions fetched per cycle should at least match the peak instruction decode and execution rate and is usually somewhat higher. The extra margin of instruction fetch bandwidth allows for instruction cache misses and for situations where fewer than the maximum number of instructions can be fetched. For example, if a branch instruction transfers control to an instruction in the middle of a cache line, then only the remaining portion of the cache line contains useful instructions. Some of the superscalar processors have taken special steps to allow wider fetches in this case, for example by fetching from two adjacent cache lines simultaneously [20, 65]. A discussion of a number of alternatives for high bandwidth instruction fetching appears in [11].

To help smooth out the instruction fetch irregularities caused by cache misses and branches, there is often an instruction buffer (shown in Fig. 4) to hold a number of fetched instructions. Using this buffer, the fetch mechanism can build up a "stockpile" to carry the processor through periods when instruction fetching is stalled or restricted.

The default instruction fetching method is to increment the program counter by the number of instructions fetched, and use the incremented program counter to fetch the next block of instructions. In case of branch instructions which redirect the flow of control, however, the fetch mechanism must be redirected to fetch instructions from the branch target. Because of delays that can occur during the process of this redirection, the handling of branch instructions is critical to good performance in superscalar processors. Processing of conditional branch instructions can be broken down into the following parts:

1) recognizing that an instruction is a conditional branch,
2) determining the branch outcome (taken or not-taken),
3) computing the branch target,
4) transferring control by redirecting instruction fetch (in the case of a taken branch).

Specific techniques are useful for handling each of the above.

Recognizing conditional branches
This seems too obvious to mention, but it is the first step toward fast branch processing. Identifying all instruction types, not just branches, can be sped up if some instruction decode information is held in the instruction cache along with the instructions. These extra bits allow very simple logic to identify the basic instruction types[2]. Consequently, there is often pre-decode logic prior to the instruction cache (see Fig. 4) which generates pre-decode bits and stores them alongside the instructions as they are placed in the instruction cache.

Determining the branch outcome
Often, when a branch instruction is fetched, the data upon which the branch decision must be made is not yet available. That is, there is a dependence between the branch instruction and a preceding, uncompleted instruction. Rather than wait, the outcome of a conditional branch can be predicted using one of several types of branch prediction methods [35, 40, 44, 53, 66]. Some predictors use static information, i.e., information that can be determined from the static binary (either explicitly present or put there by the compiler.) For example, certain opcode types might more often result in taken branches than others, or a backward branch direction (e.g. when forming loops) might be more often taken, or the compiler might be able to set a flag in an instruction to indicate the most likely branch direction based on knowledge of the high level language program. Profiling information -- program execution statistics collected during a previous run of the program -- can also be used by the compiler as an aid for static branch prediction [18].

Other predictors use dynamic information, i.e., information that becomes available as the program executes. This is usually information regarding the past history of branch outcomes -- either the specific branch being predicted, other branches leading up to it, or both. This branch history is saved in a *branch history table* or *branch prediction table* [40, 53], or may be appended to the instruction cache line that contains the branch. The branch prediction table is typically organized in a cache-like manner and is accessed with the address of the branch instruction to be predicted. Within the table previous branch history is often recorded by using multiple bits, typically implemented as small counters, so that the results of several past branch executions can be summarized [53]. The counters are incremented on a taken branch (stopping at a maximum value) and are decremented on a not-taken branch (stopping at a minimum value). Consequently, a counter's value summarizes the dominant outcome of recent branch executions.

At some time after a prediction has been made, the actual branch outcome is evaluated. Dynamic branch history information can then be updated. (It could also be updated speculatively, at the time the prediction was made [26, 60]). If the prediction was incorrect, instruction fetching must be re-directed to the correct path. Furthermore, if instructions were processed speculatively based on the prediction, they must be purged and their results must be nullified. The process of speculatively executing instruction is described in more detail later.

Computing branch targets
To compute a branch target usually requires an integer addition. In most architectures, branch targets (at least for the commonly-used branches) are relative to the program counter and use an offset value held in the instruction. This eliminates the need to read the register(s). However, the addition of the offset and the program counter is still required; furthermore, there may still be some branches that do need register contents to compute branch targets. Finding the target address can be sped up by having a branch target buffer that holds the target address that was used the last time the branch was executed. An example is the Branch Target Address Cache used in the PowerPC 604 [22].

Transferring control
When there is a taken (or predicted taken) branch there is often at least a clock cycle delay in recognizing the branch, modifying the program counter, and fetching instructions from the target address. This delay can result in pipeline bubbles unless some steps are taken. Perhaps the most common solution is to use the instruction buffer with its stockpiled instructions to mask the delay; more complex buffers may contain instructions from both the taken and the not taken paths of the branch. Some of the earlier RISC instruction sets used *delayed branches* [27, 47]. That is, a branch did not take effect until the instruction after the branch. With this method, the fetch of target instructions could be overlapped with the execution of the

[2] Although this could be done through careful opcode selection, the need to maintain compatibility, dense assignments to reduce opcode bits, and cluttered opcode assignments caused by generations of architecture extensions often make this difficult in practice.

instruction following the branch. However, in more recent processor implementations, delayed branches are considered to be a complication because they make certain assumptions about the pipeline structure that may no longer hold in advanced superscalar implementations.

3.2. Instruction Decoding, Renaming, and Dispatch

During this phase, instructions are removed from the instruction fetch buffers, examined, and control and data dependence linkages are set up for the remaining pipeline phases. This phase includes detection of true data dependences (due to RAW hazards) and resolution of other register hazards, e.g. WAW and WAR hazards caused by register reuse. Typically, this phase also distributes, or *dispatches*, instructions to buffers associated with hardware functional units for later issuing and execution. This phase works closely with the following issue stage, as will become apparent in the next section.

The job of the decode phase is to set up one or more execution *tuples* for each instruction. An execution tuple is an ordered list containing: i) an operation to be executed, ii) the identities of storage elements where the input operands reside (or will eventually reside) and iii) locations where the instruction's result must be placed. In the static program, the storage elements are the architectural storage elements, namely the architected, or *logical*, registers and main memory locations. To overcome WAR and WAW hazards and increase parallelism during dynamic execution, however, there are often *physical* storage elements that may differ from the logical storage elements. Recall that the logical storage elements can be viewed simply as a device to help describe what must be done. During parallel execution, there may be multiple data values stored in multiple physical storage elements -- with all of the values associated with the same logical storage element. However, each of the values correspond to different points in time during a purely sequential execution process.

When an instruction creates a new value for a logical register, the physical location of the value is given a "name" known by the hardware. Any subsequent instructions which use the value as an input are provided with the name of its physical storage location. This is accomplished in the decode/rename/dispatch phase (see Fig. 4) by replacing the logical register name in an instruction's execution tuple (i.e., the register designator) with the new name of the physical storage location; the register is therefore said to be *renamed* [36].

There are two register renaming methods in common usage. In the first, there is a physical register file larger than the logical register file. A mapping table is used to associate a physical register with the current value of a logical register [21, 30, 45, 46]. Instructions are decoded and register renaming is performed in sequential program order. When an instruction is decoded, its logical result register (destination) is assigned a physical register from a *free list*, i.e., the physical registers that do not currently have a logical value assigned to them, and the mapping table is adjusted to reflect the new logical to physical mapping for that register. In the process, the physical register is removed from the free list. Also, as part of the rename operation, the instruction's source register designators are used to look-up their current physical register names in the mapping table. These are the locations from which the source operand values will be read. Instruction dispatch is temporarily halted if the free list is empty.

Fig. 5 is a simple example of physical register renaming. The instruction being considered is the first *add r3,r3,4* instruction in Fig. 1. The logical, architected registers are denoted with lower case letters, and the physical registers use upper case. In the *before* side of the figure, the mapping table shows that register r3 maps to physical register R1. The first available physical register in the free list is R2. During renaming, the *add* instruction's source register r3 is replaced with R1, the physical register where a value will be placed by a preceding instruction. The destination register r3 is renamed to the first free physical register, R2, and this register is placed in the mapping table. Any subsequent instruction that reads the value produced by the *add* will have its source register mapped to R2 so that it gets the correct value. The example shown in Fig. 8, to be discussed later, shows the renaming of all the values assigned to register r3.

The only remaining matter is the reclaiming of a physical register for the free list. After a physical register has been read for the last time, it is no longer needed and can be placed back into the free list for use by other instructions. Depending on the specific implementation, this can require some more-or-less complicated hardware bookkeeping. One method is to associate a counter with each physical register. The counter is incremented each time a source register is renamed to the physical register and is decremented each time an instruction issues and actually reads a value from the register. The physical register is free whenever the count is zero, *provided* the corresponding logical register has been subsequently renamed to

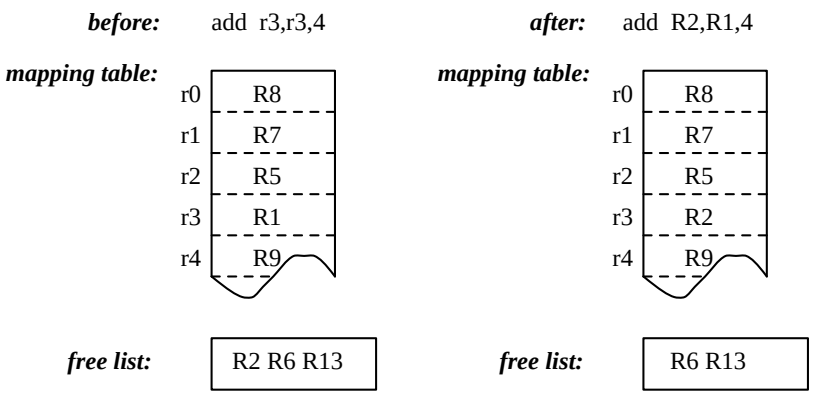


Fig. 5. Example of register renaming; logical registers are shown in lower case, and physical registers are in upper case.

another physical register.

In practice, a method simpler than the counter method is to wait until the corresponding logical register has not only been renamed by a later instruction, but the later instruction has received its value and has been committed (see section 3.5). At this point, the earlier version of the register is guaranteed to no longer be needed, including for precise state restoration in case of an interrupt.

The second method of renaming uses a physical register file that is the same size as the logical register file, and the customary one-to-one relationship is maintained. In addition, there is a buffer with one entry per active instruction (i.e., an instruction that been dispatched for execution but which has not yet committed.) This buffer is typically referred to as a *reorder buffer* [32, 55, 57] because it is also used to maintain proper instruction ordering for precise interrupts.

Fig. 6 illustrates a reorder buffer. It is easiest to think of it as FIFO storage, implemented in hardware as a circular buffer with head and tail pointers. As instructions are dispatched according to sequential program order, they are assigned entries at the tail of the reorder buffer. As instructions complete execution, their result values are inserted into their previously assigned entry, wherever it may happen to be in the reorder buffer. At the time an instruction reaches the head of the reorder buffer, if it has completed execution, its entry is removed from the buffer and its result value is placed in the register file. An incomplete instruction blocks at the head of the reorder buffer until its value arrives. Of course, a superscalar processor must be capable of putting new entries into the buffer and taking them out more than one at a time; nevertheless, adding and removing new entries still follows the FIFO discipline.

A logical register value may reside either in the physical register file or may be in the reorder buffer. When an instruction is decoded, its result value is first assigned a physical entry in the reorder buffer and a mapping table entry is set up accordingly. That is, the table indicates that a result value can be found in the specific reorder buffer entry corresponding to the instruction that produces it. An instruction's source register designators are used to access the mapping table. The table either indicates that the corresponding register has the required value, or that it may be found in the reorder buffer. Finally, when the reorder buffer is full, instruction dispatch is temporarily halted.

Fig. 7 shows the re-naming process applied to the same *add r3,r3,4* instruction as in Fig. 5. At the time the instruction is ready for dispatch (the *before* half of the figure), the values for r1 through r2 reside in the register file. However, the value for r3 resides (or will reside) in reorder buffer entry 6 until that reorder buffer entry is committed and the value can be written into the register file. Consequently, as part of the renaming process, the source register r3 is replaced with the reorder buffer entry 6 (rob6). The *add* instruction is allocated the reorder buffer entry at the tail, entry number 8 (rob8). This reorder buffer number is then recorded in the mapping table for instructions that use the result of the *add*.

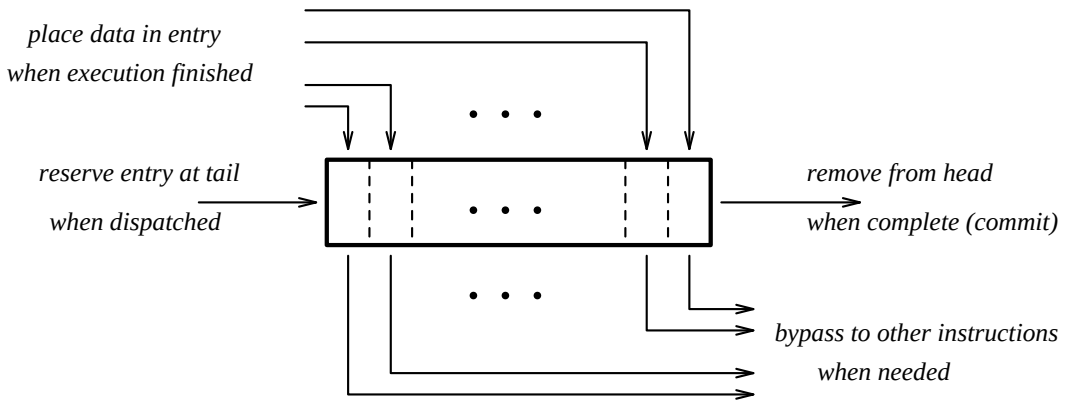


Fig. 6. A reorder buffer; entries are reserved and released in FIFO order. However, instruction results may placed in the entry at any time as they complete, and results may be read out at any time for use by dependent instructions.

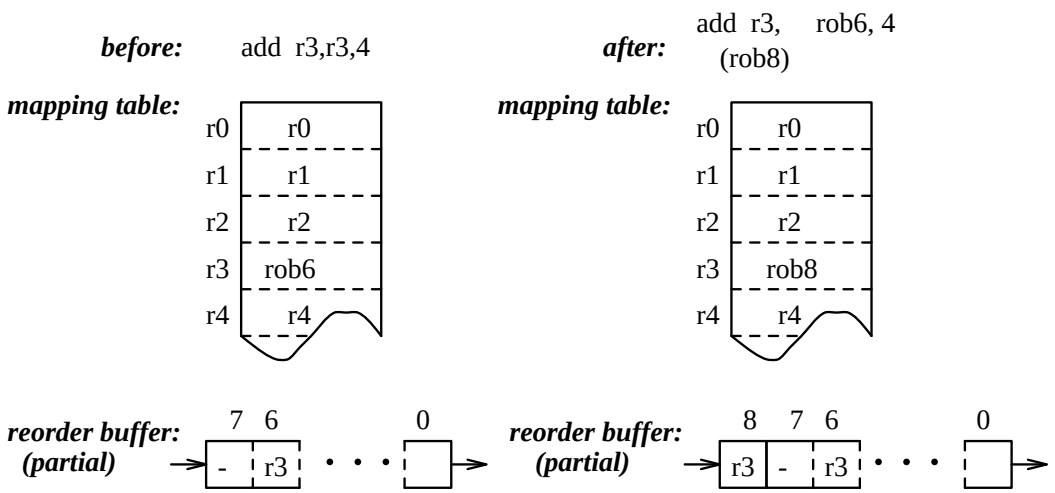


Fig. 7. Example of renaming with reorder buffer

In summary, regardless of the method used, register renaming eliminates artificial dependences due to WAW and WAR hazards, leaving only the true RAW register dependences.

3.3. Instruction Issuing and Parallel Execution

As we have just seen, it is in the decode/rename/dispatch phase that an execution tuple (opcode plus physical register storage locations) is formed. Once execution tuples are created and buffered, the next step is to determine which tuples can be *issued* for execution. Instruction issue is defined as the run-time checking for availability of data and resources. This is an area of the processing pipeline which is at the heart of many superscalar implementations -- it is the area that contains the window of execution.

Ideally an instruction is ready to execute as soon as its input operands are available. However, other constraints may be placed on the issue of instructions, most notably the availability of physical resources such as execution units, interconnect, and register file (or reorder buffer) ports. Other constraints are related to the organization of buffers holding the execution tuples.

Fig. 8 is an example of one possible parallel execution schedule for an iteration of our working example (Fig. 1). This schedule assumes hardware resources consisting of two integer units, one path to the memory, and one branch unit. The vertical direction corresponds to time steps, and the horizontal direction corresponds to the operations executed in the time step. In this schedule, we predicted that the branch *ble* was not going to be taken and are speculatively executing instructions from the predicted path. For illustrative purposes, we show only the renamed values for r3; in an actual implementation, the other registers would be renamed as well. Each of the different values assigned to r3 is bound to a different physical register (R1, R2, R3, R4).

Following paragraphs briefly describe a number of ways of organizing instruction issue buffers, in order of increasing complexity. Some of the basic organizations are illustrated in Fig. 9.

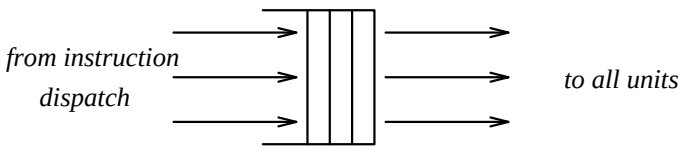
Single Queue Method
If there is a single queue, and no out-of-order issuing, then register renaming is not required, and operand availability can be managed via simple reservation bits assigned to each register. A register is *reserved* when an instruction modifying the register issues, and the reservation is *cleared* when the instruction completes. An instruction may issue (subject to physical resource availability) if there are no reservations on its operands.

Multiple Queue Method
With multiple queues, instructions issue from each queue in order, but the queues may issue out of order with respect to one another. The individual queues are organized according to instruction type. For example, there may be a floating point instruction queue, an integer instruction queue, and a load/store instruction queue. Here, renaming may be used in a restricted form. For example, only the registers loaded from memory may be renamed. This allows the load/store queue to "slip" ahead of the other instruction queues, fetching memory data in advance of when it is needed. Some earlier superscalar implementations used this method [21, 43, 54].

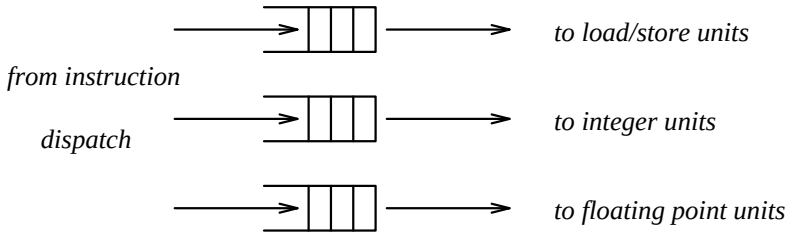
Reservation Stations
With reservation stations (see Fig. 10), which were first proposed as a part of Tomasulo's algorithm[63], instructions may issue out of order; there is no strict FIFO ordering. Consequently, all of the reservation stations simultaneously monitor their source operands for data availability. The traditional way of doing this is to hold operand data in the reservation station. When an instruction is dispatched to the reservation station, any already-available operand values are read from the register file and placed in the reservation

	INTEGER UNIT 1		INTEGER UNIT 2		MEMORY UNIT		BRANCH UNIT	
time								
↓	`move`	`R1,r7`						
	`add`	`R2,R1,4`			`lw`	`r8,(R1)`		
	`move`	`R3,r7`			`lw`	`r9,(R2)`		
	`add`	`R4,R3,4`						
	`add`	`r5,r5,1`	`add`	`r6,r6,1`	`sw`	`r9,(R3)`	`ble`	`r8,r9,L3`
	`add`	`r7,r7,4`			`sw`	`r8,(R4)`	`blt`	`r6,r4,L2`

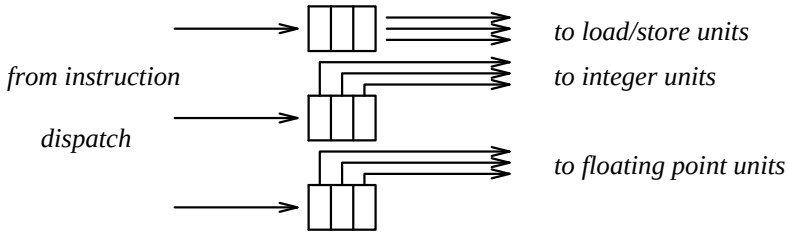
Fig. 8. Example of parallel execution schedule. In the example, we show only the renaming of logical register r3. Its physical register names are in upper case (R1, R2, R3, and R4.)



a) Single, shared queue



b) Multiple queues; one per instruction type



c) Multiple reservation stations; one per instruction type

Fig. 9. Methods of organizing instruction issue queues.

station. Then reservation station logic compares the operand designators of unavailable data with the result designators of completing instructions. When there is a match, the result value is pulled into the matching reservation station. When all the operands are ready in the reservation station, the instruction may issue (subject to hardware resource availability). Reservations may be partitioned according to instruction type (to reduce data paths)[45, 63] or may be pooled into a single large block [57]. Finally, some recent reservation station implementations do not hold the actual source data, but instead hold pointers to where the data can be found, e.g. in the register file or a reorder buffer entry.

3.4. Handling Memory Operations

Memory operations need special treatment in superscalar processors. In most modern RISC instruction sets, only explicit load and store instructions access memory, and we shall use these instructions in our discussion of memory operations (although the concepts are easily applied to register-storage and storage-storage instruction sets).

To reduce the latency of memory operations, memory hierarchies are used. The expectation is that most data requests will be serviced by data cache memories residing at the lower levels of the hierarchy.

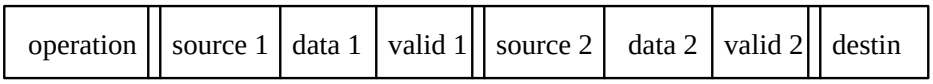


Fig. 10. A typical reservation station. There are entries for the *operation* to be performed and places for each *source* designator, its *data* value, and a *valid* bit indicating that the data value is present in the reservation station. As an instruction completes and produces result data, a comparison of the instruction's destination designator is made with the source designators in the reservation station to see if instruction in the station is waiting for the data; if so, the data is written into the value field, and the corresponding valid bit is set. When all the source operands are valid, the instruction in the reservation station is ready to issue and begin execution. At the time it begins execution, the instruction takes its destination designator along, to be used later for its reservation station comparisons.

Virtually all processors today contain a data cache, and it is rapidly becoming commonplace to have multiple levels of data caching, e.g. a small fast cache at the *primary* level and a somewhat slower, but larger, *secondary* cache. Most microprocessors integrate the primary cache on the same chip as the processor; notable exceptions are some of processors developed by HP[4] and the high-end IBM POWER series [65].

Unlike ALU instructions, for which it is possible to identify during the decode phase the register operands that will be accessed, it is not possible to identify the memory locations that will be accessed by load and store instructions until after the issue phase. The determination of the memory location that will be accessed requires an *address calculation*, usually an integer addition. Accordingly, load and store instructions are issued to an execute phase where address calculation is performed. After address calculation, an *address translation* may be required to generate a physical address. A cache of translation descriptors of recently accessed pages, the *translation lookaside buffer (TLB)*, is used to speed up this address translation process [52]. Once a valid memory address has been obtained, the load or store operation can be submitted to the memory. It should be noted that although this suggests address translation and memory access are done in series, in many superscalar implementations these two operations are overlapped -- the initial cache access is done in parallel with address translation and the translated address is then used to compare with cache tag(s) to determine if there was hit.

As with operations carried out on registers, it is desirable to have a collection of memory operations execute in as little time as possible. This means reducing the latency of memory operations, executing multiple memory operations at the same time (i.e., overlapping the execution of memory operations), overlapping the execution of memory operations with non-memory operations, and possibly allowing memory operations to execute out-of-order. However, because there are many more memory locations than registers, and because memory is indirectly addressed through registers, it is not practical to use the solutions described in the previous sections to allow memory operations to be overlapped and proceed out of order. Rather than have rename tables with information for each memory location, as we did in the case of registers, the general approach is to keep information only for a *currently active* subset of the memory locations, i.e., memory locations with currently pending memory operations, and search this subset associatively when memory needs to be accessed [3, 7, 19, 46].

Some superscalar processors only allow single memory operations per cycle, but this is rapidly becoming a performance bottleneck. To allow multiple memory requests to be serviced simultaneously, the memory hierarchy has to be multiported. It is usually sufficient to multiport only the lowest level of the memory hierarchy, namely the primary caches since many requests do not proceed to the upper levels in the hierarchy. Furthermore, transfers from the higher levels to the primary cache are in the form of lines, containing multiple consecutive words of data. Multiporting can be achieved by having multiported storage cells, by having multiple banks of memory [9, 29, 58], or by making multiple serial requests during the same cycle [65]. To allow memory operations to be overlapped with other operations (both memory and non-memory), the memory hierarchy must be *non-blocking* [37, 58]. That is, if a memory request misses in the data cache, other processing operations, including further memory requests, should be allowed to proceed.

The key to allowing memory operations to be overlapped, or to proceed out of order, is to ensure that hazards are properly resolved, and that sequential execution semantics are preserved[3]. *Store address buffers*, or simply store buffers, are used to make sure that operations submitted to the memory hierarchy do not violate hazard conditions. A store buffer contains addresses of all pending store operations. Before an operation (either a load or store) is issued to the memory, the store buffer is checked to see if there is a pending store to the same address. Fig. 11 illustrates a simple method for implementing hazard detection logic.

Once the operation has been submitted to the memory hierarchy, the operation may hit or miss in the data cache. In the case of a miss, the line containing the accessed location has to be fetched into the cache. Further accesses to the missing line must be made to wait, but other accesses may proceed. *Miss Handling Status Registers (MHSRs)* are used to track the status of outstanding cache misses, and allow multiple requests to the memory hierarchy to be overlapped [16, 37, 58].

3.5. Committing State

The final phase of the lifetime of an instruction is the *commit* or the *retire* phase, where the effects of the instruction are allowed to modify the logical process state. The purpose of this phase is to implement the appearance of a sequential execution model even though the actual execution is very likely non-sequential, due to speculative execution and out-of-order completion of instructions. The actions necessary in this phase depend upon the technique used to recover a precise state. There are two main techniques used to recover a precise state, both of which require the maintenance of two types of state: the state that is being updated as the operations execute and other state that is required for recovery.

In the first technique, the state of the machine at certain points is saved, or *checkpointed*, in either a *history buffer* or a checkpoint [31, 55]. Instructions update the state of the machine as they execute and when a precise state is needed, it is recovered from the history buffer. In this case, all that has to be done in the commit phase is to get rid of history state that is no longer required.

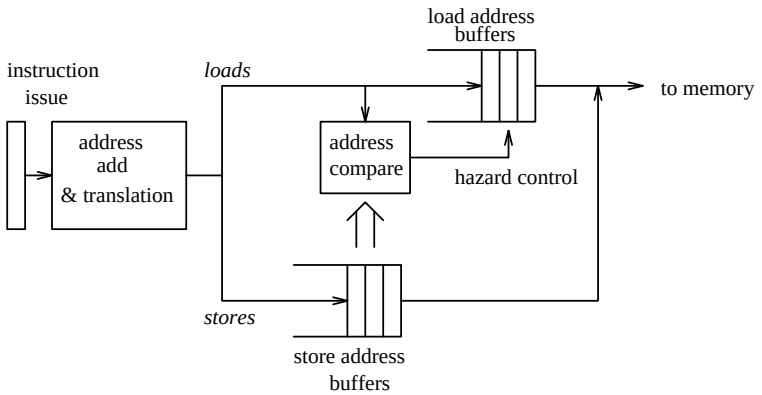


Fig. 11. Memory hazard detection logic. Store addresses are buffered in a queue (FIFO). Store addresses must remain buffered until 1)their data is available and 2) the store instruction is ready to be committed (to assure precise interrupts). New load addresses are checked with waiting store addresses. If there is a match, the load must wait for the store it matches. Store data may be bypassed to the matching load in some implementations.

[3] Some complex memory ordering issues can arise in shared memory multiprocessor implementations [15], but these are beyond the scope of this paper.

The second technique is to separate the state of the machine into two: the implemented physical state and a logical (architectural) state. The physical state is updated immediately as the operations complete. The architectural state of the machine is updated in sequential program order, as the speculative status of operations is cleared. With this technique, the speculative state is maintained in a reorder buffer; to commit an instruction, its result has to be moved from the reorder buffer into the architectural register file (and to memory in the case of a store), and space is freed up in the reorder buffer. Also, during the commit phase of a store instruction, a signal is sent to the store buffer and the store data is written to memory.

Only part of each reorder buffer entry is shown in Fig. 6. Because we were illustrating the renaming process, only the designator of the result register is shown. Also in a reorder buffer entry is a place for the data value after it has been computed, but before it is committed. And, there is typically a place to hold the program counter value of the instruction and any interrupt conditions that might occur as a result of the instruction's execution. These are used to inform the commit logic that when an interrupt should be initiated and the program counter value that should be entered as part of the precise interrupt state.

Thus far, we have discussed the reorder buffer method as it is used in implementations which use the reorder buffer method of register renaming (as illustrated in Fig. 6). However, the reorder buffer is also useful in the other style of renaming (where physical registers contain all the renamed values as in Fig. 5). With the physical register method, the reorder buffer does not have to hold values prior to being committed, rather they can be written directly into the physical register file. However, the reorder buffer does hold control and interrupt information. When an instruction is committed, the control information is used to move physical registers to the free list. If there is an interrupt, the control information used to adjust the logical-to-physical mapping table so that the mapping reflects the correct precise state.

Although the checkpoint/history buffer method has been used in some superscalar processors [12], the reorder buffer technique is by far the more popular technique because, in addition to providing a precise state, the reorder buffer method also helps implement the register renaming function [57].

3.6. The Role of Software

Though a major selling point of superscalar processors is to speed up the execution of existing program binaries, their performance can be enhanced if new, optimized binaries can be created[4]. Software can assist by creating a binary so that the instruction fetching process, and the instruction issuing and execution process, can be made more efficient. This can be done in the following two ways: (i) increasing the likelihood that a group of instructions that are being considered for issue can be issued simultaneously, and (ii) decreasing the likelihood that an instruction has to wait for a result of a previous instruction when it is being considered for execution. Both these techniques can be accomplished by *scheduling* instructions statically. By arranging the instructions so that a group of instructions in the static program matches the parallel execution constraints of the underlying superscalar processor, the first objective can be achieved. The parallel execution constraints that need to be considered include the dependence relationships between the instructions, and the resources available in the microarchitecture. (For example, to achieve the parallel issue of four consecutive instructions, the microarchitecture might require all four of them to be independent.) To achieve the second objective, an instruction which produces a value for another instruction can be placed in the static code so that it is fetched and executed far in advance of the consumer instruction.

Software support for high performance processors is a vast subject -- one that cannot be covered in any detail in this paper. A good discussion of software topics appears in the article in this issue written by Wen-Mei Hwu.

4. Three Superscalar Microprocessors

In this section we discuss three current superscalar microprocessors in light of the above framework. The three were chosen to cover the spectrum of superscalar implementations as much as possible. They are the MIPS R10000 [23], which fits the "typical" framework described above, the DEC Alpha 21164 [24] which strives for greater simplicity, and the AMD K5 [51] which implements a more complex and older

[4]The new binary could use the same instruction set and the same sequential execution model as the old binary; the only difference might be the order in which the instructions are arranged.

instruction set than the other two -- the Intel x86.

4.1. MIPS R10000

The MIPS R10000 does extensive dynamic scheduling and is very similar to our "typical" superscalar processor described above. In fact, Fig. 4 is modeled after the R10000 design. The R10000 fetches four instructions at a time from the instruction cache. These instructions have been pre-decoded when they were put into the cache. The pre-decode generates four additional bits per instruction which help determine the instruction type immediately after the instructions are fetched from the cache. After being fetched, branch instructions are predicted. The prediction table is contained within the instruction cache mechanism; the instruction cache holds 512 lines and there are 512 entries in the prediction table. Each entry in the prediction table holds a two-bit counter value that encodes history information used to make the prediction.

When a branch is predicted to be taken, it takes a cycle to redirect instruction fetching. During that cycle, sequential instructions (i.e., those on the not-taken path) are fetched and placed in a *resume cache* as a hedge against an incorrect prediction. The resume cache has space for four blocks of instructions, so four branch predictions can be handled at any given time.

Following instruction fetch, instructions are decoded, their register operands are renamed, and they are dispatched to the appropriate instruction queue. The predecoded bits, mentioned earlier, simplify this process. All register designators are renamed, using physical register files that are twice the size of the logical files (64 physical versus 32 logical). The destination register is assigned an unused physical register from the free list, and the map is updated to reflect the new logical-to-physical mapping. Operand registers are given the correct designators by reading the map.

Then, up to four instructions at a time are dispatched into one of three instruction queues: memory, integer, and floating point, i.e., the R10000 uses a form of queued instruction issue as in Fig. 9c. Other than the total of at most four, there are no restrictions on the number of instructions that may be placed in any of the queues. The queues are each 16 entries deep, and only a full queue can block dispatch. At the time of dispatch, a reservation bit for each physical result register is set busy. Instructions in the integer and floating point instruction queues do not issue in a true FIFO discipline (hence, the term "queue" is a little misleading). The queue entries act more like reservation stations; however, they do not have "value" fields or "valid" bits as in Fig. 10. Instead of value fields, the reservation stations hold the physical register designators which serve as pointers to the data. Each instruction in the queue monitors the global register reservation bits for its source operands. When the reservation bits all become "not busy", the instruction's source operands are ready, and subject to availability of its functional unit, the instruction is free to issue.

There are five functional units: an address adder, two integer ALUs, a floating point multiplier/divider/square-rooter and a floating point adder. The two integer ALUs are not identical -- both are capable of the basic add/subtract/logical operations, but only one can do shifts and the other can do integer multiplications and additions.

As a general policy, an effort is made to process the older instruction first when more than one instruction has ready operands and is destined for the same functional unit. However, this policy does not result in a strict priority system in order to simplify the arbitration logic. The memory queue, however, does prioritize instructions according to age, which makes address hazard detection simpler.

The R10000 supports an on-chip primary data cache (32Kb, 2-way set associative, 32-byte lines), and an off-chip secondary cache. The primary cache blocks only on the second miss.

The R10000 uses a reorder buffer mechanism for maintaining a precise state at the time of an exception condition. Instructions are committed in the original program sequence, up to four at a time. When an instruction is committed, it frees up the old physical copy of the logical register it modifies. The new physical copy (which may have been written many cycles before) becomes the new architecturally precise version. Exception conditions for non-committed instructions are held in the reorder buffer. When an instruction with exceptions is ready to be committed, an interrupt will occur. Information in the reorder buffer for instructions following the interrupting one are used to re-adjust the logical-to-physical register mapping so that the logical state is consistent with the interrupting instruction.

When a branch is predicted, the processor takes a snapshot of the register mapping table. Then, if the branch is subsequently found to be mispredicted, the register mapping can be quickly restored. There is

space to allow snapshots of up to four predicted branches at any given time. Furthermore, by using the resume cache, instruction processing can often begin immediately.

4.2. Alpha 21164

The Alpha 21164 (Fig. 12) is an example of a simple superscalar processor that forgoes the advantages of dynamic instruction scheduling in favor of a high clock rate. Instructions are fetched from an 8 Kbytes instruction cache four at a time. These instructions are placed in one of two instruction buffers, each capable of holding four instructions. Instructions are issued from the buffers in program order, and a buffer must be completely emptied before the next buffer can be used. This restricts the instruction issue rate somewhat, but it greatly simplifies the necessary control logic.

Branches are predicted using a table that is associated with the instruction cache. There is a branch history entry for each instruction in the cache; each entry is a two bit counter. There can be only one predicted, and yet unresolved, branch in the machine at a time; instruction issue is stalled if another branch is encountered before a previously-predicted branch has been resolved.

Following instruction fetch and decode, instructions are inspected and arranged according to type, i.e., the functional unit they will use. Then, provided operand data is ready (in registers or available for bypassing) instructions are issued to the units and begin execution. During this entire process, instructions are not allowed to pass one another. The 21164 is therefore an example of the single queue method shown in Fig. 9a.

There are four functional units: two integer ALUs, a floating point adder, and a floating point multiplier. The integer ALUs are not identical -- only one can perform shifts and integer multiplies, the other is the only one that can evaluate branches.

The 21164 has two levels of cache memory on the chip. There are a pair of small, fast primary caches of 8K bytes each -- one for instructions and one for data. The secondary cache, shared by instructions and data, is 96K bytes and is three-way set associative. The small direct mapped primary cache is to allow single-cycle cache accesses at a very high clock rate. The primary cache can sustain a number of outstanding misses. There is a six entry *miss address file (MAF)* that contains the address and target register for a load that misses. If the address is to the same line as is another address in the MAF, the two are merged into the same entry. Hence the MAF can hold many more than 6 outstanding cache misses (in

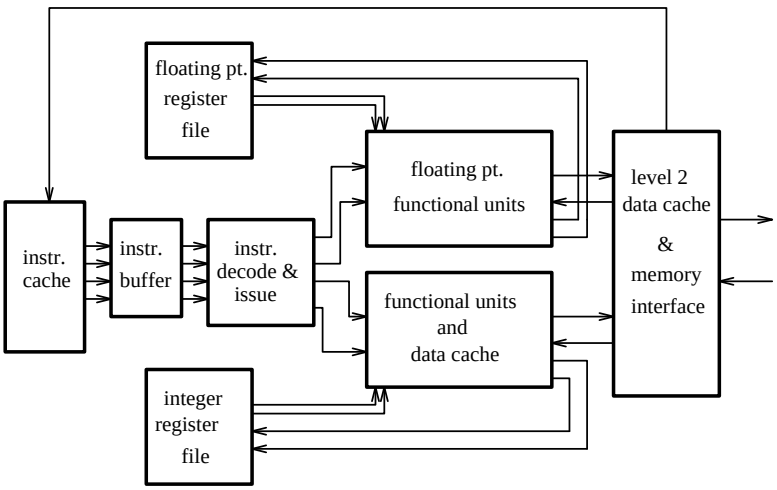


Fig. 12. DEC Alpha 21164 superscalar organization.

theory, up to 21).

To provide a sequential state at the time of a interrupt, the 21164 does not issue out of order, and keeps instructions in sequence as they flow down the pipeline. The final pipeline stage updates the register file in original program sequence. There are bypasses along the pipeline so that data can be used before it is committed to the register file. This is a simple form of a reorder buffer with bypasses. The pipeline itself forms the reorder buffer (and order is maintained so "*re*order" overstates what is actually done). Floating point instructions are allowed to update their files out of order, and, as a result, not all floating point exceptions result in a precise architectural state.

4.3. AMD K5

In contrast to the other two microprocessors discussed in this section, the AMD K5 (Fig. 13) implements an instruction set that was not originally designed for fast, pipelined implementation. It is an example of a complex instruction set -- the Intel x86.

The Intel x86 instruction set uses variable length instructions. This means, among other things, one instruction must be decoded (and its length established) before the beginning of the next can be found. Consequently, instructions are sequentially pre-decoded as they are placed into the instruction cache. There are five pre-decode bits per instruction byte. These bits indicate information such as whether the byte is the beginning or end of an instruction and identifies bytes holding opcodes and operands. Instructions are then fetched from the instruction cache at a rate of up to 16 bytes per cycle. These instruction bytes are placed into a 16 element byte queue where they wait for dispatch.

As in the other microprocessors described in this section, the K5 integrates its branch prediction logic with the instruction cache. There is one prediction entry per cache line. A single bit reflects the direction taken by the previous execution of the branch. The prediction entry also contains a pointer to the target instruction -- this information indicates where in the instruction cache the target can be found. This reduces the delay for fetching a predicted target instruction.

Due to the complexity of the instruction set, two cycles are consumed decoding. In the first stage, instruction bytes are read from the byte queue and are converted to simple, RISC-like OPerations (called ROPs by AMD). Up to four ROPs are formed at a time. Often, the conversion requires a small number of ROPs per x86 instruction. In this case, the hardware does a single-cycle conversion. More complex x86 instructions are converted into a sequence of ROPs via sequential lookup from a ROM. In this case, the

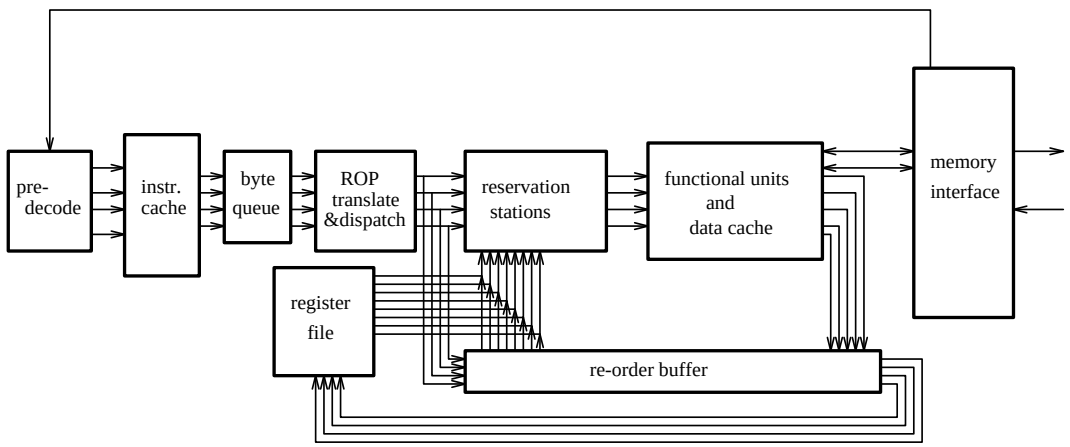


Fig. 13. AMD K5 superscalar implementation of the Intel x86 instruction set.

ROPs are essentially a form of microcode; up to 4 ROPs per cycle are generated from the ROM. After conversion to ROPs, most of the processor is focussed on their execution as individual operations, largely ignoring the original x86 instructions from which they were generated.

Following the first decode cycle, instructions read operand data (if available) and are dispatched to functional unit reservation stations, up to 4 ROPS per cycle (corresponding to up to 4 x86 instructions). Data can be in the register file, or can be held in the reorder buffer. When available this data is read and sent with the instruction to functional unit reservation stations. If operand data has not yet been computed, the instruction will wait for it in the reservation station.

There are six functional units: two integer ALUs, one floating point unit, two load/store units, and a branch unit. One of the integer ALUs capable of shifts, and the other can perform integer divides. Although they are shown as a single block in Fig. 13, the reservation stations are partitioned among the functional units. Except the FPU, each of these units has two reservation stations; the FPU has only one. Based on availability of data, ROPs are issued from the reservation stations to their associated functional units. There are enough register ports and data paths to support up to 4 such ROP issues per cycle.

There is an 8K byte data cache which has 4 banks. Dual load/stores are allowed, provided they are to different banks. (One exception is if both references are to the same line, in which case they can both be serviced together.)

A 16 entry reorder buffer is used to maintain a precise process state when there is an exception. The K5 reorder buffer holds result data until it is ready to be placed in the register file. There are bypasses from the buffer and an associative lookup is required to find the data if it is complete, but pending the register write. This is a classic reorder buffer method of renaming we described earlier. The reorder buffer is also used to recover from incorrect branch predictions.

5. Conclusions: What Comes Next?

Both performance and compatibility have driven the development of superscalar processors. They implement a sequential execution model although the actual execution of a program is far from sequential. After being fetched, the sequential instruction stream is torn apart with only true dependences holding the instructions together. Instructions are executed in parallel with a minimum of constraints. Meanwhile enough information concerning the original sequential order is retained so that the instruction stream can conceptually be squeezed back together again should there be need for a precise interrupt.

A number of studies have been done to determine the performance of superscalar methods, for example [56, 64]. Because the hardware and software assumptions and benchmarks vary widely, so do the potential speedups -- ranging from close to 1 for some program/hardware combinations to many 100s for others. In any event, it is clear that there is something to be gained, and every microprocessor manufacturer has embraced superscalar methods for high-end products.

While superscalar implementations have become a staple just like pipelining, there is some consensus that returns are likely to diminish as more parallel hardware resources are introduced. There are a number of reasons for thinking this, we give two of the more important. First, there may be limits to the instruction level parallelism that can be exposed and exploited by currently-known superscalar methods, even if very aggressive methods are used [64]. Perhaps the most important of these limits results from conditional branches. Studies that compare performance using real branch prediction with theoretically perfect branch prediction note a significant performance decline when real prediction is used. Second, superscalar implementations grow in complexity as the number of simultaneously issued instructions increases. Data path complexity (numbers of functional units, register and cache ports, for example) grows for obvious reasons. However, control complexity also grows because of the "cross-checks" that must be done between all the instructions being simultaneously dispatched and/or issued. This represents quadratic growth in control complexity which, sooner or later, will affect the clock period that can be maintained.

In addition, there are some important system level issues that might ultimately limit performance. One of the more important is the widening gap between processor and main memory performance. While memories have gotten larger with each generation, they have not gotten much faster. Processor speeds, on the other hand have improved markedly. A number of solutions are being considered, ranging from more sophisticated data cache designs, better cache prefetch methods, more developed memory hierarchies, and memory chip designs that are more effective at providing data. This issue affects all the proposed methods for increasing instruction level parallelism, not just the superscalar implementations.

In the absence of radically new superscalar methods, many believe that 8-way superscalar (more or less) is a practical limit -- this is a point that will be reached within the next couple of generations of processors. Where do we go from that point? One school of thought believes that the Very Large Instruction Word (VLIW) model [10, 17, 48] will become the paradigm of choice. The VLIW model is derived from the sequential execution model: control sequences from instruction to instruction, just like in the sequential model discussed in this paper. However, each instruction in a VLIW is a collection of several independent operations, which can all be executed in parallel. The compiler is responsible for generating a parallel execution schedule, and representing the execution schedule as a collection of VLIW instructions. If the underlying hardware resources necessitate a different execution schedule, a different program binary must be generated for optimal performance.

VLIW proponents claim several advantages for their approach, and we will summarize two of the more important. First, with software being responsible for analyzing dependences and creating the execution schedule, the size of the instruction window that can be examined for parallelism is much larger than what a superscalar processor can do in hardware. Accordingly, a VLIW processor can expose more parallelism. Second, since the control logic in a VLIW processor does not have to do any dependence checking (the entire parallel execution schedule is orchestrated by the software), VLIW hardware can be much simpler to implement (and therefore may allow a faster clock) than superscalar hardware.

VLIW proponents have other arguments that favor their approach, and VLIW opponents have counter arguments in each case. The arguments typically center around the actual effect that superscalar control logic complexity will have on performance and the ability of a VLIW's compile time analysis to match dynamic scheduling using run-time information. With announced plans from Intel and Hewlett-Packard and with other companies seriously looking at VLIW architectures, the battle lines have been drawn and the outcome may be known in a very few years.

Other schools of thought strive to get away from the single flow of control mindset prevalent in the sequential execution model (and the superscalar implementations of the model). Rather than sequence through the program instruction-by-instruction, in an attempt to create the dynamic instruction sequence from the static representation, more parallel forms of sequencing may be used.

In [39] it is shown that allowing multiple control flows provides the potential for substantial performance gains. That is, if multiple program counters can be used for fetching instructions, then the effective window of execution from which independent instructions can be chosen for parallel execution can be much wider.

One such method is a variation of the time-tested *multiprocessing* method. Some supercomputer and mini-supercomputer vendors have already implemented approaches for automatic parallelization of high level language programs [5, 33]. With chip resources being sufficient to allow multiple processors to be integrated on a chip, multiprocessing, once the domain of supercomputers and mainframes, is being considered for high-end microprocessor design [25, 42]. If multiprocessing is to exploit instruction level parallelism at the very small grain size that is implied by "instruction level", suitable support has to be provided to keep the inter-processor synchronization and communication overhead extremely low. As with traditional multiprocessing, single-chip multiprocessors will require enormous amounts compiler support to create a parallel version of a program statically.

Looking even farther ahead, more radical approaches that support multiple flows of control are being studied [14, 59]. In these approaches, individual processing units are much more highly integrated than in a traditional multiprocessor, and the flow control mechanism is built into the hardware, as well. By providing mechanisms to support speculative execution, such methods allow the parallel execution of programs that can't be analyzed and parallelized statically.

Microprocessor designs have successfully passed from simple pipelines, to simple superscalar implementations, to fully-developed superscalar processors -- all in little more than a decade. With the benefit of hindsight, each step seems to be a logical progression from the previous one. Each step tapped new sources of instruction level parallelism. Today, researchers and architects are facing new challenges. The next logical step is not clear, and the search for new, innovative approaches to instruction level parallelism promises to make the next decade as exciting as the previous one.

Acknowledgments

This work was supported in part by NSF Grants CCR-9303030 and MIP-9505853, ONR Grant N00014-93-1-0465, the University of Wisconsin Graduate School, and Intel Corp. The authors would like to thank Mark Hill for comments and Scott Breach and Dionisios Pnevmatikatos for their help with some of the figures.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, 1986.

[2] G. Amdahl, et al, ‘‘Architecture of the IBM System/360,’’ *IBM J. Res. Devel.*, vol. 8, pp. 87-101, April 1964.

[3] D. W. Anderson, F. J. Sparacio, and R. M. Tomasulo, ‘‘The IBM System/360 Model 91: Machine Philosophy and Instruction-Handling,’’ *IBM Journal of Research and Development*, vol. 11, pp. 8-24, January 1967.

[4] T. Asprey, et al, ‘‘Performance Features of the PA7100 Microprocessor,’’ *IEEE Micro*, vol. 13, pp. 22-35, June 1993.

[5] M. C. August, G. M. Brost, C. C. Hsiung, and A. J. Schiffleger, ‘‘Cray X-MP: The Birth of a Supercomputer,’’ *IEEE Computer*, vol. 22, pp. 45-54, January 1989.

[6] C. G. Bell, ‘‘Multis: a New Class of Multiprocessor Computers,’’ *Science*, vol. 228, pp. 462-467, April 1985.

[7] L. J. Boland, G. D. Granito, A. U. Marcotte, B. U. Messina, and J. W. Smith, ‘‘The IBM System/360 Model 91: Storage System,’’ *IBM Journal*, vol. 11, pp. 54-68, January 1967.

[8] Werner Bucholz, ed., *Planning a Computer System*. New York: McGraw-Hill, 1962.

[9] B. Case, ‘‘Intel Reveals Pentium Implementation Details,’’ *Microprocessor Report*, pp. 9-13, Mar. 29, 1993.

[10] R. P. Colwell, R. P. Nix, J. J. O’Donnell, D. B. Papworth, and P. K. Rodman, ‘‘A VLIW Architecture for a Trace Scheduling Compiler,’’ *IEEE Transactions on Computers*, vol. 37, pp. 967-979, August 1988.

[11] T. M. Conte, P. M. Mills, K. N. Menezes, and B. A. Patel, ‘‘Optimization of Instruction Fetch Mechanisms for High Issue Rates,’’ *Proc. 22nd Annual International Symposium on Computer Architecture*, pp. 333-344, June 1995.

[12] K. Diefendorff and M. Allen, ‘‘Organization of the Motorola 88110 Superscalar RISC Microprocessor,’’ *IEEE Micro*, vol. 12, April 1992.

[13] P. K. Dubey and M. J. Flynn, ‘‘Optimal Pipelining,’’ *Journal of Parallel and Distributed Computing*, vol. 8, pp. 10-19, 1990.

[14] P. K. Dubey, K. O’brien, K. O’brien, and C. Barton, ‘‘Single-Program Speculative Multithreading (SPSM) Architecture: Compiler-assisted Fine-Grained Multithreading,’’ *Proc. Int. Conf. on Parallel Architectures and Compilation Techniques (PACT ’95)*, pp. 109-121, June 1995.

[15] M. Dubois, C. Scheurich, and F. A. Briggs, ‘‘Synchronization, Coherence, and Event Ordering in Multiprocessors,’’ *IEEE Computer*, vol. 21, pp. 9-21, February 1988.

[16] K. I. Farkas and N. P. Jouppi, ‘‘Complexity/Performance Tradeoffs with Non-Blocking Loads,’’ *Proc. 21st Annual International Symposium on Computer Architecture*, pp. 211-222, April 1994.

[17] J. A. Fisher and B. R. Rau, ‘‘Instruction-Level Parallel Processing,’’ *Science*, pp. 1233-1241, September 1991.

[18] J. A. Fisher and S. M. Freudenberger, ‘‘Predicting Conditional Branch Directions from Previous Runs of a Program,’’ *Proc. Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, 1992.

[19] M. Franklin and G. S. Sohi, ‘‘ARB: A Hardware Mechanism for Dynamic Memory Disambiguation,’’ *to appear in IEEE Transactions on Computers*.

[20] G. F. Grohoski, J. A. Kahle, L. E. Thatcher, and C. R. Moore, ‘‘Branch and Fixed Point Instruction Execution Units,’’ in *IBM RISC System/6000 Technology*, Austin, TX.

[21] G. F. Grohoski, ‘‘Machine Organization of the IBM RISC System/6000 processor,’’ *IBM Journal of Research and Development*, vol. 34, pp. 37-58, January 1990.

[22] L. Gwennap, ‘‘PPC 604 Powers Past Pentium,’’ *Microprocessor Report*, pp. 5-8, Apr. 18, 1994.

[23] L. Gwennap, ‘‘MIPS R10000 Uses Decoupled Architecture,’’ *Microprocessor Report*, pp. 18-22, Oct. 24, 1994.

[24] L. Gwennap, ‘‘Digital Leads the Pack with the 21164,’’ *Microprocessor Report*, pp. 1,6-10, Sept. 12, 1994.

[25] L. Gwennap, ‘‘Architects Debate VLIW, Single-Chip MP,’’ *Microprocessor Report*, pp. 20-21, December 5, 1994.

[26] E. Hao, P.-Y. Chang, and Y. N. Patt, ‘‘The Effect of Speculatively Updating Branch History on Branch Prediction Accuracy, Revisited,’’ *Proc. 27th Int. Symp. on Microarchitecture*, pp. 228-232, December 1994.

[27] J. Hennessy, N. Jouppi, F. Baskett, T. Gross, and J. Gill, ‘‘Hardware/Software Tradeoffs for Increased Performance,’’ *Proc. Int. Symp. on Arch. Support for Prog. Lang. and Operating Sys.*, pp. 2-11, March 1982.

[28] R. G. Hintz and B. P. Tate, ‘‘Control Data STAR-100 Processor Design,’’ *COMPCON*, p. 396, Sept. 1972.

[29] P. Y. T. Hsu, ‘‘Design of the R8000 Microprocessor,’’ *IEEE Micro*, pp. 23-33, April 1994.

[30] W. W. Hwu and Y. N Patt, ‘‘HPSm, a High Performance Restricted Data Flow Architecture Having Minimal Functionality,’’ *Proc. 13th Annual International Symposium on Computer Architecture*, pp. 297-307, June 1986.

[31] W. W. Hwu and Y. N. Patt, ‘‘Checkpoint Repair for High-Performance Out-of-Order Execution Machines,’’ *IEEE Transactions on Computers*, vol. C-36, pp. 1496-1514, December 1987.

[32] M. Johnson, *Superscalar Design*. Englewood Cliffs, New Jersey: Prentice Hall, 1990.

[33] T. Jones, ‘‘Engineering Design of the Convex C2,’’ *IEEE Computer*, vol. 22, pp. 36-44, January 1989.

[34] N. P. Jouppi and D. W. Wall, ‘‘Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines,’’ in *Proc. Architectural Support for Programming Languages and Operating Systems (ASPLOS-III)*, Boston, MA, April 1989.

[35] D. R. Kaeli and P. G. Emma, ‘‘Branch History Table Prediction of Moving Target Branches Due to Subroutine Returns,’’ *Proc. 18th Annual International Symposium on Computer Architecture*, pp. 34-42, May, 1991.

[36] R. M. Keller, ‘‘Look-Ahead Processors,’’ *ACM Computing Surveys*, vol. 7, pp. 66-72, December 1975.

[37] D. Kroft, ‘‘Lockup-Free Instruction Fetch/Prefetch Cache Organization,’’ *Proc. 8th Annual Symposium on Computer Architecture*, pp. 81-87, May 1981.

[38] S. R. Kunkel and J. E. Smith, ‘‘Optimal Pipelining in Supercomputers,’’ *Proc. 13th Annual International Symposium on Computer Architecture*, pp. 404-413, June 1986.

[39] M. S. Lam and R. Wilson, ‘‘Limits of Control Flow on Parallelism,’’ *Proc. 19th Annual International Symposium on Computer Architecture*, pp. 46-57, May 1992.

[40] J. K. F. Lee and A. J. Smith, ‘‘Branch Prediction Strategies and Branch Target Buffer Design,’’ *IEEE Computer*, vol. 17, pp. 6-22, January 1984.

[41] C. R. Moore, ‘‘The PowerPC 601 Microprocessor,’’ *Proc. Compcon 1993*, pp. 109-116, Februrary 1993.

[42] B. A. Nayfeh and K. Olukotun, ‘‘Exploring the Design Space for a Shared-Cache Multiprocessor,’’ *Proc. 21st Annual International Symposium on Computer Architecture*, pp. 166-175, April, 1994.

[43] R. R. Oehler and R. D. Groves, ‘‘IBM RISC System/6000 Processor Architecture,’’ *IBM Journal of Research and Development*, vol. 34, pp. 23-36, January 1990.

[44] S.-T. Pan, K. So, and J. T. Rahmeh, ‘‘Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation,’’ *Proc. Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pp. 76-84, October 1992.

[45] Y. N. Patt, W. W. Hwu, and M. Shebanow, ‘‘HPS, A New Microarchitecture: Rationale and Introduction,’’ *Proc. 18th Annual Workshop on Microprogramming*, pp. 103-108, December 1985.

[46] Y. N. Patt, S. W. Melvin, W. W. Hwu, and M. Shebanow, ‘‘Critical Issues Regarding HPS, A High Performance Microarchitecture,’’ *Proc. 18th Annual Workshop on Microprogramming*, pp. 109-116, December 1985.

[47] D. A. Patterson and C. H. Sequin, ‘‘RISC 1: A Reduced Instruction Set VLSI Computer,’’ *Proc. 8th Annual Symposium on Computer Architecture*, pp. 443-459, May 1981.

[48] B. R. Rau, D. W. L. Yen, W. Yen, and R. Towle, ‘‘The Cydra 5 Departmental Supercomputer: Design Philosophies, Decisions, and Trade-offs,’’ *IEEE Computer*, vol. 22, pp. 12-35, January, 1989.

[49] R. M. Russel, ‘‘The CRAY-1 Computer System,’’ *Communications of the ACM*, vol. 21, pp. 63-72, January 1978.

[50] H. Schorr, ‘‘Design Principles for a High Performance System,’’ in *Proc. of the Symposium on Computers and Automata*, New York, NY, pp. 165-192, April 1971.

[51] M. Slater, ‘‘AMD’s K5 Designed to Outrun Pentium,’’ *Microprocessor Report*, pp. 1,6-11, Oct. 24, 1994.

[52] A. J. Smith, ‘‘Cache Memories,’’ *ACM Computing Surveys*, vol. 14, pp. 473-530, Sept. 1982.

[53] J. E. Smith, ‘‘A Study of Branch Prediction Strategies,’’ *Proc. 8th Annual Symposium on Computer Architecture*, pp. 135-148, May 1981.

[54] J. E. Smith, et al, ‘‘The ZS-1 Central Processor,’’ *Proc. Architectural Support for Programming Languages and Operating Systems (ASPLOS-II)*, pp. 199-204, October 1987.

[55] J. E. Smith and A. R. Pleszkun, ‘‘Implementing Precise Interrupts in Pipelined Processors,’’ *IEEE Transactions on Computers*, vol. 37, pp. 562-573, May 1988.

[56] M. D. Smith, M. Johnson, and M. A. Horowitz, ‘‘Limits on Multiple Instruction Issue,’’ *Proc. Architectural Support for Programming Languages and Operating Systems (ASPLOS-III)*, pp. 290-302, 1989.

[57] G. S. Sohi, ‘‘Instruction Issue Logic for High-Performance, Interruptible, Multiple Functional Unit, Pipelined Computers,’’ *IEEE Trans. on Computers*, vol. 39, pp. 349-359, March 1990.

[58] G. S. Sohi and M. Franklin, ‘‘High-Bandwidth Data Memory Systems for Superscalar Processors,’’ *Proc. Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pp. 53-62, April, 1991.

[59] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar, ‘‘Multiscalar Processors,’’ *Proc. 22nd Annual International Symposium on Computer Architecture*, pp. 414-425, June 1995.

[60] A. R. Talcott, W. Yamamoto, M. J. Serrano, R. C. Wood, and M. Nemirovsky, ‘‘The Impact of Unresolved Branches on Branch Prediction Performance,’’ in *Proc. 21st Annual International Symposium on Computer Architecture*, Chicago, Illinois, pp. 12-21, April 1994.

[61] J. E. Thornton, ‘‘Parallel Operation in the Control Data 6600,’’ *Fall Joint Computers Conference*, vol. 26, pp. 33-40, 1961.

[62] G. S. Tjaden and M. J. Flynn, ‘‘Detection and Parallel Execution of Independent Instructions,’’ *IEEE Transactions on Computers*, vol. C-19, pp. 889-895, October 1970.

[63] R. M. Tomasulo, ‘‘An Efficient Algorithm for Exploiting Multiple Arithmetic Units,’’ *IBM Journal of Research and Development*, pp. 25-33, January 1967.

[64] D. W. Wall, ‘‘Limits of Instruction-Level Parallelism,’’ *Proc. Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pp. 176-188, April 1991.

[65] S. Weiss and J. E. Smith, *Power and PowerPC: Principles, Architecture, Implementation*. San Francisco, CA: Morgan Kaufmann, 1994.

[66] T. Y. Yeh and Y. N. Patt, ‘‘Alternative Implementations of Two-Level Adaptive Training Branch Prediction,’’ *Proc. 19th Annual International Symposium on Computer Architecture*, pp. 124-134, May, 1992.