THE DESIGN SPACE OF REGISTER RENAMING TECHNIQUES

TO BOOST PROCESSOR AND SYSTEM PERFORMANCE, VIRTUALLY ALL RECENT SUPERSCALARS RENAME REGISTERS.

Dezsö Sima
Budapest Polytechnic

Register renaming is a technique to remove false data dependencies—write after read (WAR) and write after write (WAW)—that occur in straight line code between register operands of subsequent instructions.[1-3] By eliminating related precedence requirements in the execution sequence of the instructions, renaming increases the average number of instructions that are available for parallel execution per cycle. This results in increased IPC (number of instructions executed per cycle).

The identification and exploration of the design space of register-renaming lead to a comprehensive understanding of this intricate technique. As this article shows, the design space of register renaming is spanned by four main dimensions: the scope of register renaming, the layout of the rename buffers, the method of register mapping, and the rename rate. Relevant aspects of the design space give rise to eight basic alternatives for register-renaming. In addition, the kind of operand fetch policy significantly affects how the processor carries out the rename process, which duplicates the eight basic alternatives to 16 possible implementation schemes. The article indicates which basic implementation scheme is used in relevant superscalar processors.

As register renaming is usually implemented in conjunction with shelving, the underlying microarchitecture is assumed to employ shelving. (See the "Instruction shelving principle" box for a discussion of this technique.)

Register renaming

The principle of register renaming is straightforward. If the processor encounters an instruction that addresses a destination register, it temporarily writes the instruction's result into a dynamically allocated rename buffer rather than into the specified destination register. For instance, in the case of the following WAR dependency:

```
i1:    add …, r2, …;  [… ← (r2) + (…)]
i2:    mul r2, …, …;  [r2 ← (…) * (…)]
```

the destination register of i2 (r2) is renamed, say to r33. Then, instruction i2 becomes

```
i2′ :   mul r33, …, …;  [r33 ← (…) * (…)]
```

Its result is written into r33 instead of into r2. This resolves the previous WAR dependency between i1 and i2. In subsequent instructions, however, references to source registers must be redirected to the rename buffers allocated to them as long as this renaming remains valid.[3]

A precursor to register renaming was introduced for floating-point instructions in 1967 by Tomasulo in the IBM 360/91,[4] a scalar supercomputer of that time that pioneered both pipelining and shelving (dynamic instruction issue). The 360/91 renamed floating-point registers to preserve the logical consistency of the program execution rather than to remove false data dependencies.

0272-1732/00/$10.00 © 2000 IEEE

Tjaden and Flynn[5] first suggested the use of register renaming for removing false data dependencies for a limited set of instructions that corresponds more or less to the load instructions. However, they didn't use the term "register renaming." Keller[6] introduced this designation in 1975 and extended renaming to cover all instructions including a destination register. He also described how to implement register renaming in processors. Even so, due to the complexity of this technique almost two decades passed after its conception before register renaming came into widespread use in superscalars at the beginning of the 1990s.

Early superscalars such as the HP PA 7100, Sun SuperSparc, DEC Alpha 21064, MIPS R8000, and Intel Pentium typically didn't use renaming. Renaming appeared gradually—first in a restricted form called partial renaming (to be discussed in the next section) —in the early 1990s in the IBM RS/6000 (Power1), Power2, PowerPC 601, and the NextGen Nx586 processors. See Figure 1. Full renaming emerged later, beginning in 1992, first in the high-end models of the IBM mainframe ES/9000, then in the PowerPC 603. Subsequently, renaming spread into virtually all superscalar processors with the notable exception of Sun's UltraSparc line. At present, register renaming is considered to be a standard feature of performance-oriented superscalar processors.

Design space of register-renaming techniques

The main dimensions of register renaming are as follows:

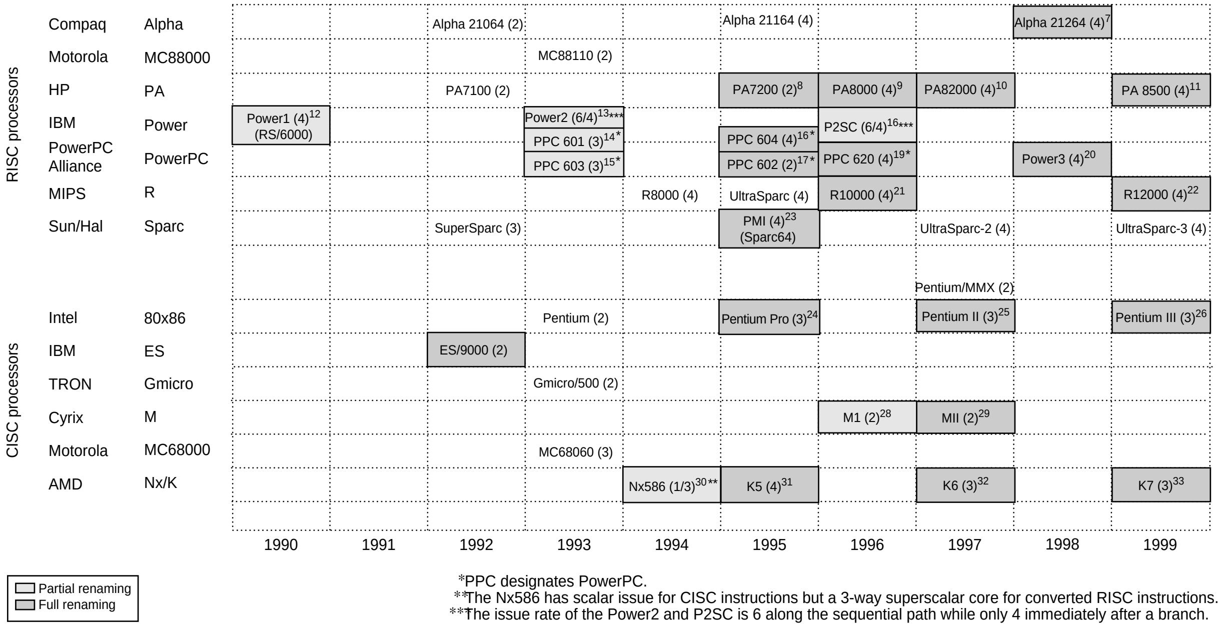
- scope of register renaming,
- layout of the renamed registers,

Instruction shelving principle

In early superscalars, decoded and executable instructions are issued immediately to the execution units. However, using this scheme control and data dependencies, and busy execution units, cause issue bottlenecks. The basic technique used to remove an issue bottleneck is instruction shelving, also known as dynamic instruction issue.[3,35,45]

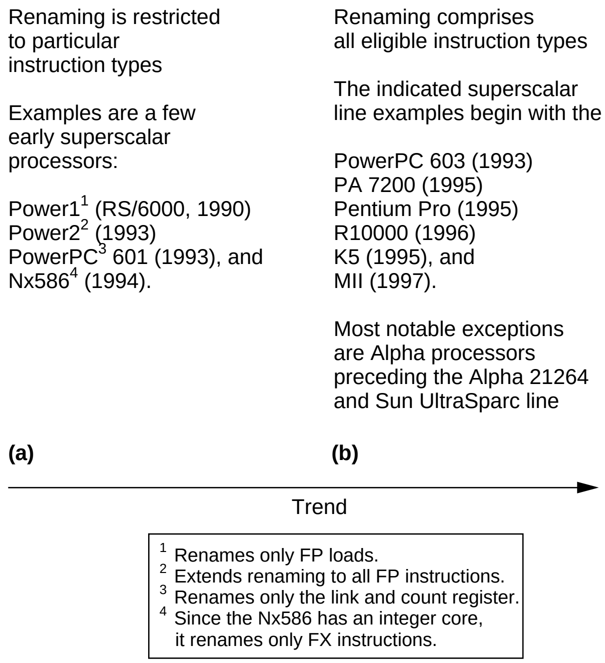
Shelving presumes the availability of dedicated buffers, called shelving buffers, in front of the execution units. The processor first issues instructions into available shelving buffers without checking for data or control dependencies, or for busy execution units. As data dependencies or busy execution units no longer restrict instruction issue, the issue bottleneck problem occurring in early superscalars is removed. In a second step, instructions held in the shelving buffers are dispatched for execution. During dispatching, instructions are checked for dependencies, and not-dependent instructions are forwarded to free execution units.

At the time being, there's no consensus on the use of terms instruction issue and instruction dispatch. Both terms are used in both possible interpretations.



*PPC designates PowerPC.
**The Nx586 has scalar issue for CISC instructions but a 3-way superscalar core for converted RISC instructions.
***The issue rate of the Power2 and P2SC is 6 along the sequential path while only 4 immediately after a branch.

Figure 1. Chronology of register renaming in commercial superscalar processors. The introduction date indicates the first year of volume production. Following the model designation is the issue rate of the processors (in parentheses). Note that for the issue rate of CISC processors, one x86 instruction is considered to be the equivalent of 1.3 to 1.9 RISC instructions.[34]



Renaming is restricted to particular instruction types

Examples are a few early superscalar processors:

Power1[1] (RS/6000, 1990)
Power2[2] (1993)
PowerPC[3] 601 (1993), and
Nx586[4] (1994).

Renaming comprises all eligible instruction types

The indicated superscalar line examples begin with the

PowerPC 603 (1993)
PA 7200 (1995)
Pentium Pro (1995)
R10000 (1996)
K5 (1995), and
MII (1997).

Most notable exceptions are Alpha processors preceding the Alpha 21264 and Sun UltraSparc line

[1] Renames only FP loads.
[2] Extends renaming to all FP instructions.
[3] Renames only the link and count register.
[4] Since the Nx586 has an integer core, it renames only FX instructions.

Figure 2. Register renaming scope: partial (a) and full (b).

- method of register mapping and,
- rename rate

as indicated in subsequent sections. Due to volume restrictions we ignore additional dimensions that are related to the renaming process such as recovery from a misprediction.

Scope of register renaming

To indicate how extensively the processor makes use of renaming, I distinguish between partial and full renaming. Partial renaming is restricted to one or only a few instruction types, for instance, only to floating-point instructions. Early processors typically employed this incomplete form of renaming; the Power1 (RS/6000), Power2, PowerPC 601, and the Nx586 are examples, as shown in Figure 2. Of these, the Power1 (RS/6000) renames only floating-point loads. As the Power1 has only a single floating-point unit, it executes floating-point instructions in sequence. Thus there's no need for renaming floating-point register instructions. The Power2 processor introduces multiple floating-point units and for this reason it extends renaming to all floating-point instructions. The Power PC 601 renames only the link and count register. The Nx586, which has an integer core, obviously restricted renaming to fixed-point instructions.

Full renaming covers all instructions including a destination register. As Figure 1 demonstrates, virtually all recent superscalar processors employ full renaming. Noteworthy exceptions are the Sun UltraSparc line and Alpha processors preceding the Alpha 21264.

Rename buffer layout

Rename buffers establish the actual framework for renaming. There are three essential design aspects in their layout:

- type of rename buffers,
- number of rename buffers, and
- number of read and write ports.

Rename buffer types

The choice of which type of rename buffers to use in a processor has far-reaching impact on the implementation of the rename process. Given its importance, designers must consider the various design options. To simplify this presentation, I initially assume a common architectural register file for all processed data types, and then extend the discussion to the split-register scenario that is commonly employed.

As Figure 3 illustrates, there are four fundamentally different ways to implement rename buffers. The range of choices includes a) using a merged architectural and rename register file, b) employing a stand-alone rename register file, c) keeping renamed values either in the reorder buffer (ROB), or d) in the shelving buffers.

In the first approach, rename buffers are implemented along with the architectural registers in the same physical register file called the merged architectural and rename register file or the merged register file for short. Here, both architectural and rename registers are dynamically allocated to particular registers of the same physical file.

Each physical register of the merged architectural and rename register file is at any time in one of four possible states.[27] These states reflect the actual use of a physical register as follows:

- uncommitted (available) state,
- used as an architectural register (archi-

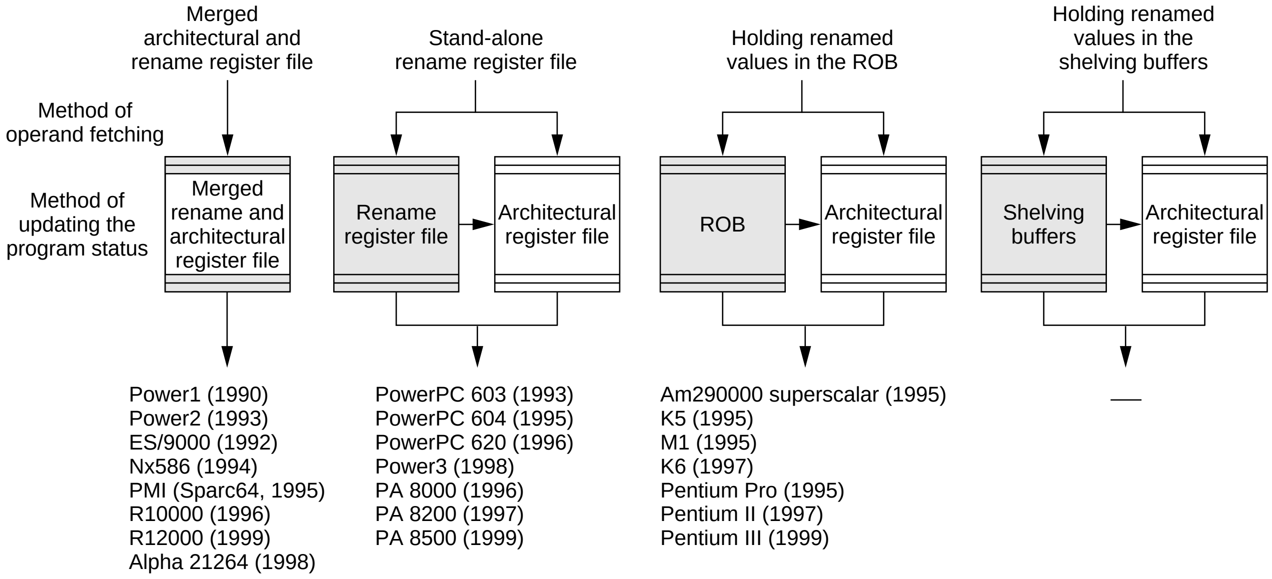

Merged architectural and rename register file
Stand-alone rename register file
Holding renamed values in the ROB
Holding renamed values in the shelving buffers
Method of operand fetching
Method of updating the program status
Merged rename and architectural register file
Rename register file
Architectural register file
ROB
Architectural register file
Shelving buffers
Architectural register file
Power1 (1990)
Power2 (1993)
ES/9000 (1992)
Nx586 (1994)
PMI (Sparc64, 1995)
R10000 (1996)
R12000 (1999)
Alpha 21264 (1998)
PowerPC 603 (1993)
PowerPC 604 (1995)
PowerPC 620 (1996)
Power3 (1998)
PA 8000 (1996)
PA 8200 (1997)
PA 8500 (1999)
Am290000 superscalar (1995)
K5 (1995)
M1 (1995)
K6 (1997)
Pentium Pro (1995)
Pentium II (1997)
Pentium III (1999)
—

Figure 3. Generic types of rename buffers. Shaded boxes indicate the rename buffers.

tectural register state),

- used as a rename buffer—but this register doesn't yet contain the result of the associated instruction (rename buffer, not-valid state), and
- used as a rename buffer—this register already contains the result of the associated instruction (rename buffer, valid state).

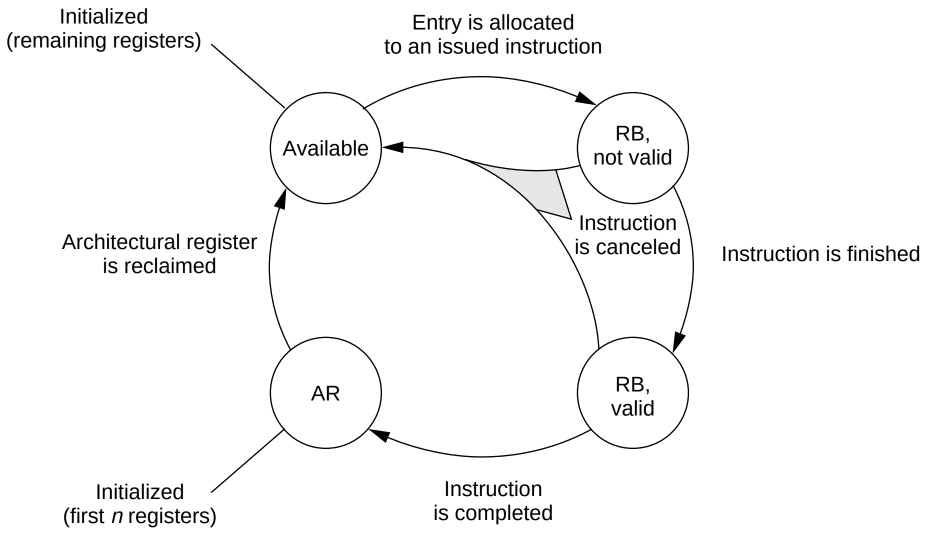

Initialized (remaining registers)
Entry is allocated to an issued instruction
Available
RB, not valid
Instruction is canceled
Architectural register is reclaimed
Instruction is finished
AR
RB, valid
Initialized (first n registers)
Instruction is completed

Figure 4. State transition diagram of a particular register of the merged architectural and rename register file.[27] AR: architectural register, RB: rename buffer.

During instruction processing, the states of the physical registers are changed as described in the following and indicated in the state transition diagram in Figure 4.

As part of the initialization the first *n* physical registers are assigned to the architectural registers, where *n* is the number of the registers declared by the instruction set architecture (ISA). These registers are set to be in the architectural register (AR) state; the remaining physical registers take on the available state. When an issued instruction includes a destination register, a new rename buffer is needed. For this reason, one physical register is selected from the pool of the available registers and allocated to the concerned destination register. Accordingly, its state is set to the rename buffer, not-valid state, and its valid bit is reset. After the associated instruction finishes execution, the produced result is written into the allocated rename buffer. Its valid bit is set, and its state changes to rename buffer, valid. Later, when the associated instruction completes, the allocated rename buffer will be declared to be the architectural register that implements the

ROB principle

The reorder buffer is implemented basically as a circular buffer whose entries are allocated and deallocated by means of two revolving pointers.[3,37] It operates as follows.

When instructions are issued, a ROB entry is allocated to each instruction, strictly in program order. Each ROB entry keeps track of the execution status of the associated instruction. The ROB allows instructions to complete (commit, retire) only in program order by permitting an instruction to complete only if it has finished its execution and all preceding instructions are already completed. In this way, instructions update the program state in exactly the same way as a sequential processor would have done. After an instruction has completed, the associated ROB entry is deallocated and becomes eligible for reuse.

destination register specified in the just-completed instruction. Its state then changes to the architectural register state to reflect this.

Note that in contrast to other rename buffer types, no data transfer is required for updating the architectural registers in merged architectural and rename register files. Instead, only the status of the related registers needs to be changed.

Finally, when an old architectural register is reclaimed, it is set to the available state. Assuming dispatch-bound operand fetching, a possibility for reclaiming such old architectural registers is to keep track of the physical registers that have been the previous instances of the same architectural register, and reclaim the previous instance when the instruction incorporating the new instance completes.

Also, not-yet-completed instructions must be canceled for exceptions or faulty executed speculative instructions. Then allocated rename buffers in a) the rename buffer, not-valid and b) the rename buffer, valid states are deallocated and changed to available states. In addition, the corresponding mappings—kept in either the mapping table or the rename buffer (as discussed later)—must be canceled.

Merged architectural and rename register files are employed, for instance, in the high-end models (520-based models) of the IBM ES/9000 mainframes, Power and R1x000 processors, and Alpha 21264s.

All other alternatives separate rename buffers from architectural registers.

In the first "separated" variant, a standalone rename register file (or rename register file for short) is used exclusively to implement rename buffers. The PowerPC 603/620 and PA8x00 processors are examples of using rename register files.

Alternatively, renaming can also be based on the reorder buffer (ROB); see "ROB principle" box. The ROB has recently been widely used to preserve the sequential consistency of instruction execution. When using a ROB, an entry is assigned to each issued instruction for the duration of its execution. It's quite natural to use this entry for renaming as well—basically by extending it with a new field that will hold the result of that instruction. Examples of processors using the ROB for renaming are the Am29000 superscalar, K5, K6, Pentium Pro, Pentium II, and Pentium III.

The ROB can even be extended further to serve as a central shelving buffer as well. In this case, the ROB is also occasionally designated as the DRIS (deferred scheduling register renaming instruction shelf). The Lightning processor proposal[36] and the K6 made use of this solution. Because the Lightning proposal—which dates back to the beginning of the 1990s—was too ambitious in the light of the technology available at that time, it couldn't be economically implemented and never reached the market.

The last conceivable implementation alternative of rename buffers is to use the shelving buffers for renaming (see the "Instruction shelving principle" box again). In this case, each shelving buffer must be extended functionally to also perform the task of a rename buffer. But this alternative has a drawback resulting from the different deallocation mechanisms of the shelving and rename buffers. While shelving buffers can be reclaimed as soon as the instruction has been dispatched, rename buffers can be deallocated only at a later time, not earlier than the instruction has been completed. Thus, a deeper analysis is needed to reveal the appropriateness of using shelving buffers for renaming. To date, no processor has chosen this alternative.

Split rename register files

For simplicity's sake, I've so far assumed that all data types are stored in a common architectural register file. But usually, processors provide distinct architectural register files for fixed-point and floating-point data. Consequently, they typically employ distinct rename register files, as shown in Figure 5.

As depicted in this figure, when the processor employs the split-register principle, dis-

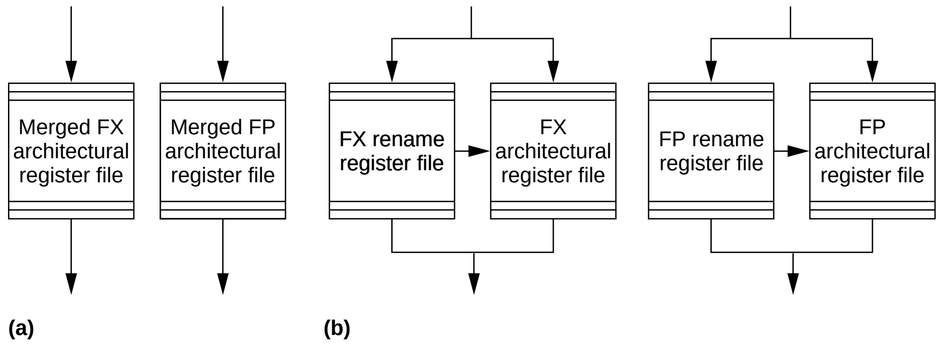

Merged FX architectural register file
Merged FP architectural register file
FX rename register file
FX architectural register file
FP rename register file
FP architectural register file
(a)
(b)

Figure 5. Using split registers in the case of (a) merged register files, and (b) a stand-alone rename register file. FX indicates fixed point; FP indicates floating point.

tinct fixed-point and floating-point register files are needed for merged files and stand-alone rename register files. In this case separate data paths are also needed to access the fixed-point and the floating-point registers.

Recent processors typically incorporate split rename registers. When renaming takes place within the ROB, usually a single mechanism is maintained for the preservation of the sequential consistency of instruction execution. Then all renamed instructions are kept in the same ROB queue despite using split architectural register files for fixed-point and floating-point data. In this case, clearly, each ROB entry is expected to be long enough to hold either fixed-point or floating-point data.

Number of rename buffers

Rename buffers keep register results temporarily until instructions complete. By taking into account that not every instruction produces a register result, we can state that in a processor up to as many rename buffers are needed as the maximum number of instructions that are in execution; that is, issued but not yet completed. Issued but not yet completed instructions are either

- held in shelving buffers waiting for execution (if shelving is employed),
- just been processed in execution units,
- in the load queue waiting for cache access (if there is a load queue), or
- in the store queue waiting for completion, then forwarded to the cache to execute the required store operation (if there is a store queue).

Thus, the maximal number of instructions that may have been issued but not yet completed in the processor (*npmax*) is

$$npmax = wdw + nEU + nLq + nSq \qquad (1)$$

where

wdw is the width of the dispatch window (total number of shelving buffers),
nEU is the number of the execution units that may operate in parallel,
nLq is the number of the entries in the load queue, and
nSq is the number of the entries in the store queue.

Assuming a worst-case design approach, from this formula we can determine that the total number of rename buffers required (*nrmax*) is

$$nrmax = wdw + nEU + nLq, \qquad (2)$$

since instructions held in the store queue don't require rename buffers.

Furthermore, if the processor includes a ROB, based on Equation 1 we can say that the total number of ROB entries required (*nROBmax*) is

$$nROBmax = npwax. \qquad (3)$$

Nevertheless, if the processor has fewer rename buffers or fewer ROB entries than expected, according to the worst-case approach (as given by Equations 2 and 3) issue blockages can occur due to missing free

Table 1. Type and available number of rename buffers in recent superscalars as well as four related parameters of the enlisted processors.

Processor type (year of volume shipment)	Type of rename buffer	No. of rename buffers FX	FP	Issue rate	Width of dispatch window (*wdw*)	Total no. of rename buffers (*nr*)	Reorder width (*nROB*)
RISC processors							
PowerPC 603 (1993)	Ren. reg. file	N/A	4	3	3	N/A	5
PowerPC 604 (1995)	Ren. reg. file	12	8	4	12	20	16
PowerPC 620 (1996)	Ren. reg. file	8	8	4	15	16	16
Power3 (1998)	Ren. reg. file	16	24	4	20 (?)	40	32
R10000 (1996)	Merged	32	32	4	48	64	32
R12000 (1998)	Merged	32	32	4	48	64	48
Alpha 21264 (1998)	Merged	48	41	4	35	89	80
PA 8000 (1986)	Ren. reg. file	56	56	4	56	112	56
PM1 (1996)	Merged	38	24	4	36	62	62
x86 (CISC) processors							
Pentium Pro (1995)	In the ROB	40		3[2]	20[1]	40	40[1]
Pentium II (1997)	In the ROB	40		3[2]	20[1]	40	40[1]
K5 (1995)	In the ROB	16		4[2]	11[1] (?)	16	16[1]
K6 (1996)	In the ROB	24		3[2]	24[1]	24	24[1]
M3 (2000 expected)	Merged	32	N/A	3[2]	56[1]	N/A	32[2]

[1] RISC operations

[2] x86 instructions (on average, produce 1.3 to 1.9 RISC operations[34])

? Questionable data

N/A Not available

rename buffers or ROB entries. With a decreasing number of entries provided, we expect a smooth, slight performance degradation. Hence, a stochastic design approach is also feasible. There, the required number of entries is derived from the tolerated level of performance degradation.

Based on Equations 1 to 3, the following relations are typically valid concerning the width of the processor's dispatch window (*wdw*), the total number of the rename buffers (*nr*), and the reorder width (*nROB*), which equals the total number of ROB entries available:

$$wdw < nr \leq nROB \qquad (4)$$

Table 1 summarizes the type and the number of rename buffers provided in recent RISC and x86 superscalar processors. In addition, Table 1 shows four key parameters of the enlisted processors: the issue rate, width of the dispatch window (*wdw*), total number of rename buffers provided (*nr*), and the reorder width (*nROB*).

As the data in Table 1 indicates, the designs of most processors have taken into account the four interrelations. There are, however, two obvious exceptions. First, the PowerPC 604 provides 20 rename buffers, more than the processor's reorder width of 16. In the subsequent PowerPC 620, Intel decreased the number to 16. Second, the R10000 provides only 32 ROB entries. This number is far too low compared to the dispatch width (48) and to the number of available rename buffers (64). MIPS addressed this disproportion in the R12000 by increasing the reorder width of the processor to 48.

Number of read and write ports

By taking into account current practice, in the following discussion I assume split register files.

Clearly, as many read ports are required in the rename buffers as there are data items that the rename buffers may need to supply in any one cycle. Note that rename buffers supply required operands for the instructions to be executed and also forward the results of the completed instructions to the addressed architectural registers.

The number of operands that need to be delivered in the same cycle depends first of all on whether the processor fetches operands during instruction issue or during instruction dispatch. (See the "Operand fetch policies" box.)

If operands are fetched issue bound, the rename buffers need to supply the operands for all instructions that are issued into the shelving buffers in the same cycle. Thus, both the fixed-point and floating-point rename buffers are expected to deliver in each cycle all required operands for up to as many instructions as the issue rate. This means that in recent four-way superscalar processors the fixed-point and the floating-point rename buffers typically need to supply 8 and 12 operands respectively, assuming up to two fixed-point and three floating-point operands in each fixed-point and floating-point instruction, respectively. If, however, there are some issue restrictions, the required number of read ports is decreased accordingly.

In contrast, if the processor uses the dispatch-bound fetch policy, the rename buffers should provide the operands for all instructions that are forwarded from the dispatch window (instruction window) for execution in the same cycle. In this case, the fixed-point rename buffers need to supply the required fixed-point operands for the integer and load-store units (including register operands for the specified address calculations and fixed-point data for the fixed-point store instructions).

The floating-point rename buffers need to deliver operands for the floating-point units (floating-point register data) and also for the load-store units (floating-point operands of the floating-point store instructions). In the Power3, for instance, this implies the following read port requirements. The fixed-point rename buffers need to have 12 read ports (up to 3 × 2 operands for the 3 integer-units as well as 2 × 2 address operands and 2 × 1 data operands for the 2 load-store units). On the other hand, the floating-point rename registers need to have 8 read ports (up to 2 × 3 operands for the 2 floating-point units and 2 × 1 operands for the 2 load-store units).

In addition, if rename buffers are implemented separately from the architectural registers, the rename buffers must forward in each cycle as many result values to the architectural registers as the completion rate (retire rate) of the processor. Since recent processors usually complete up to four instructions per cycle, this task typically increases the required number of read ports in the rename buffers by four.

Too many read ports in a register file may unduly increase the physical size of the data path and consequently the cycle time. To avoid this problem, a few high-performance processors (such as the Power2, Power3, and Alpha 21264) implement two copies of particular register files. The Power2 duplicates the fixed-point architectural register file, the Power3 doubles both the fixed-point rename and the architectural file, and the Alpha 21264 provides two copies of the fixed-point merged architectural and rename register file.

As a result, fewer read ports are needed in each of the copies. For example, with two copies of the fixed-point merged register file, the Power3 needs only 10 read ports in each file, instead of 16 read ports in a fixed-point register file. A drawback of this approach is, however, that a scheme is also required to keep both copies coherent.

Now let's turn to the required number of write ports (input ports). Since in each cycle rename buffers need to accept all results produced by the execution units, these buffers must provide as many write ports as results the execution units may produce per cycle. The fixed-point rename buffers receive results from the available integer-execution units and from the load-store units (fetched fixed-point data). In contrast, the floating-point rename buffers hold the results of the floating-point execution units and the load-store units (fetched float-

Operand fetch policies

If a processor uses the issue-bound fetch policy, it fetches referenced register operands during instruction issue—that is, while it forwards decoded instructions into the shelving buffers.[3,11] In contrast, the dispatch-bound fetch policy defers operand fetching until executable instructions are forwarded from the shelving buffers to the execution units. When a processor fetches issue-bound operands, shelving buffers hold the source operand values. In contrast, in dispatch-bound operand fetching, shelving buffers have much shorter entries as they contain only the register identifiers.

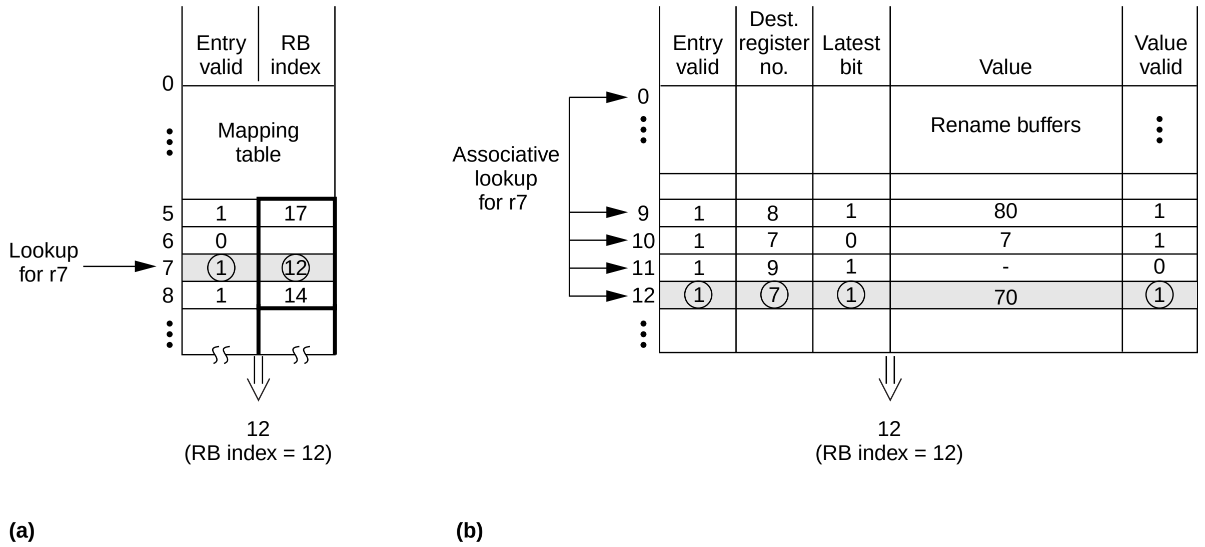


Figure 6. Methods for keeping track of the actual mapping of architectural registers to rename buffers: using a mapping table (a) and mapping within rename buffers (b).

ing-point data). Most results are single data items requiring one write port. However, there are a few exceptions. When execution units generate two data items, they require two write ports as well, similar to the PowerPC processor load-store units. After execution of the LOAD-WITH-UPDATE instruction, these units return both the fetched data value and the updated address value.

Register mapping methods

During renaming, the processor needs to allocate rename buffers to the destination registers of the instructions (or usually to every instruction to simplify logic). It also must keep track of the mappings actually used and deallocate rename buffers no longer used. Accordingly, the related aspect of the design space has three components:

- allocation scheme of the rename buffers,
- method of keeping track of actual mappings, and
- deallocation scheme of rename buffers.

As far as the allocation scheme of rename buffers is concerned, rename buffers are usually allocated to instructions during instruction issue. If rename buffers are assigned to the instructions as early as during instruction issue, rename buffer space is wasted, since rename buffers are not needed until the results become available in the last execution cycle. Delaying the allocation of rename buffers to the instructions[37] saves rename buffer space. Various schemes have been proposed for this, such as virtual renaming[37-40] and others.[41] In fact, a virtual allocation scheme has already been introduced into the Power3.[37]

Established mappings must be maintained until their invalidation. There are two possibilities for keeping track of the actual mapping of particular architectural registers to allocated rename buffers. The processor can use a mapping table for this or can track the actual register mapping within the rename buffers themselves. See Figure 6.

A mapping table has as many entries as there are architectural registers provided by the instruction set architecture (usually 32). Each entry holds a status bit (called the entry valid bit in Figure 6a), which indicates whether the associated architectural register is renamed. Each valid entry supplies the index of the rename buffer, which is allocated to the architectural register belonging to that entry (called the RB index). For instance, Figure 6a shows that the mapping table holds a valid entry for architectural register r7, which contains the RB index of 12. This indicates that architectural register r7 is actually renamed to rename buffer 12.

Each entry is set up during instruction issue, while new rename buffers are allocated to the issued instructions. A valid mapping is updated when the architectural register belonging to that entry is renamed again. It will be invalidated when the related mapping is no longer needed and the allocated rename buffer is reclaimed. In this way, the mapping table continuously provides the latest allocations. Source registers are renamed by accessing the mapping table with the register numbers as indices and fetching the associated rename buffer identifiers (RB indices), as Figure 6a shows.

Obviously, for split architectural register files, separate fixed-point and floating-point mapping tables are needed.

As discussed earlier, mapping tables should provide one read port for each source operand that may be fetched in any one cycle and one write port for each rename buffer that may be allocated in any one cycle.

The other fundamentally different alternative for keeping track of actual register mappings relies on an associative mechanism (see Figure 6b). In this case no mapping table exists, but each rename buffer holds the identifier of the associated architectural register (usually the register number of the renamed destination register) and additional status bits. These entries are set up during instruction issue when a particular rename buffer is allocated to a specified destination register. As Figure 6b shows, in this case each rename buffer holds the following five pieces of information:

- a status bit, which indicates that this rename buffer is actually allocated (called the entry valid bit in the figure),
- the identifier of the associated architectural register (Dest. Reg. No.),
- a further status bit, called the latest bit, whose role will be explained later,
- another status bit, called the value valid bit, which shows whether the actual value of the associated architectural register has already been generated, and
- the value itself, provided that the value valid bit signifies an already produced result.

The latest bit marks the last allocation of a given architectural register if it has more than one valid allocation due to repeated renaming. For instance, in our example architectural register r7 has two subsequent allocations. From these, entry 12 is the latest one as its latest bit has been set. Thus, in Figure 6b, renaming source register r7 would yield the RB index of 12. This method of renaming source registers requires an associative lookup in all entries searching for the latest allocation of the given source register.

With issue-bound operand fetching, source registers are both renamed and accessed during the issue process. For this reason, in this case, processors usually integrate renaming and operand accessing, and therefore keep track of the register mapping within the rename buffers. For dispatch-bound operand fetching, however, these tasks are separated. Source registers are renamed during instruction issue, whereas the source operands are accessed while the processor dispatches the instructions to the execution units. Therefore, in this case, processors typically use mapping tables.

If rename buffers are no longer needed, they should be reclaimed (deallocated). The scheme of deallocation depends on key aspects of the overall renaming process. In particular, they depend on the allocation scheme of the rename buffers, the type of rename buffers used, the method of keeping track of actual allocations, and even whether issue-bound or dispatch-bound operands are fetched. However, lack of space restricts discussion of this aspect of the design space in detail here.

Rename rate

As its name suggests, the rename rate stands for the maximum number of renames that a processor can perform in a cycle. Basically, the processor should rename all instructions issued in the same cycle to avoid performance degradation. Thus, the rename rate should equal the issue rate. This is easier said than done, since it is not at all an easy task to implement a high rename rate (four or higher). Two reasons make it difficult.

First, for higher rename rates the detection and handling of interinstruction dependencies during renaming becomes a more complex task. Second, higher rename rates require a larger number of read and write ports on register files and mapping tables. For instance, the four-way superscalar R10000 can issue any combination of four fixed-point and floating-point instructions. Accordingly, its fixed-point

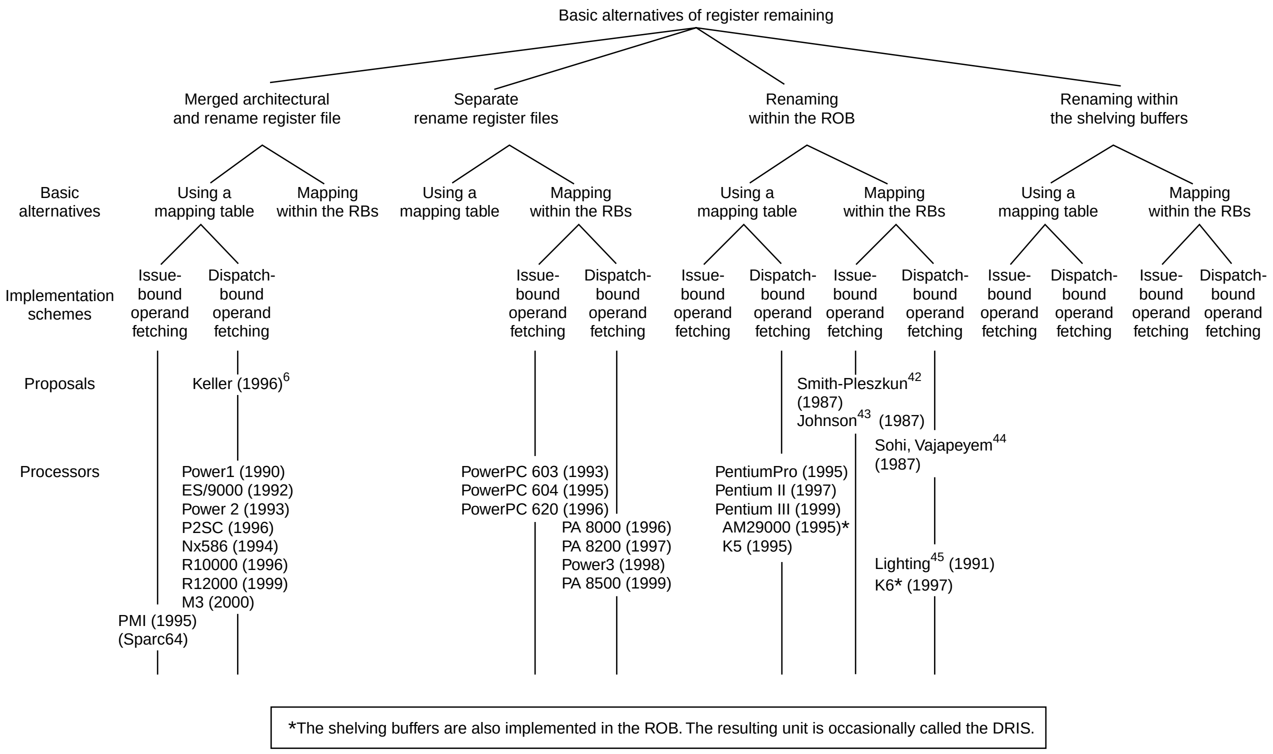


Figure 7. Basic implementation alternatives of register renaming. RB designates the rename buffer.

mapping table needs 12 read ports and 4 write ports, and its floating-point table requires 16 read and 4 write ports. This many ports are needed since fixed-point instructions can refer up to 3, and floating-point instructions up to 4 source operands in this processor.

Basic alternatives, possible implementation schemes

Theoretically in the design space of register renaming, each possible combination of the available design choices yields one possible implementation alternative. However, instead of considering all possible implementation alternatives, it makes sense to focus only on those that differ in relevant qualitative aspects from each other—the basic alternatives. Possible basic alternatives can be derived from the design space in two steps: by identifying the relevant qualitative design aspects involved and then by composing their possible combinations.

When selecting the relevant qualitative design aspects, we should recall the design space of renaming mentioned earlier. First, we ignore two main aspects: the scope of register renaming because recent processors typically implement full renaming, and the rename rate because of its quantitative character. Two main design aspects remain: the layout of the rename buffers and the method of register mapping. Furthermore, the layout of the rename buffers itself covers three design aspects: the type of rename buffers, their number, and the number of the read and write ports. Of these only the type of the rename buffers is of qualitative character. It follows that the design space of register renaming includes only two relevant qualitative aspects: the type of the rename buffers and the method of register mapping.

The design choices available for these two relevant design aspects result in eight possible combinations, called the basic alternatives for register renaming, as shown in Figure 7. In addition, this figure also takes into account that the processor's operand fetch policy—which is a design aspect of shelving—significantly affects how the renaming process is carried out. This splits the eight basic renaming alternatives into 16 feasible implementation schemes. Figure 7 also indicates which implementation schemes are used in relevant superscalar processors and some hints about their origins.

As Figure 7 indicates, relevant superscalar processors make use of only four of the eight possible basic alternatives of renaming. Moreover, most of the latest processors employ the following basic alternatives of renaming when fetching dispatch-bound operands:

- use of merged architectural and rename register files and mapping tables (R10000, R12000, M3).
- use of separate rename register files and mapping registers within the rename registers (PA8x00 line, Power3), and
- renaming within the ROB and using mapping tables (Pentium Pro, Pentium II, Pentium III).

It's also conceivable to use different basic alternatives for renaming fixed-point and floating-point instructions, as is done in the K7. This processor uses the ROB for renaming fixed-point instructions and a merged architectural and rename register file for renaming floating-point ones. However, as AMD didn't disclose the method of register mapping, this processor isn't included in Figure 7.

As this figure shows, the latest processors fetch predominantly dispatch-bound operands due to the comparative advantage of this fetch policy.[40] The move away from the issue-bound operands to the dispatch-bound fetch policy is manifested in AMD's subsequent K5 and K6, and by the fact that the PowerPC 620-based Power3 has also made this transition.

Register renaming and shelving mark the first major step in the evolution of superscalar processors. The introduction of these techniques became necessary to resolve the issue bottleneck of early superscalar designs. As revealed in this article, register renaming is a complex technique whose understanding requires the identification and exploration of its design space. MICRO

References

1. B.R. Rau and J.A. Fisher, "Instruction Level Parallel Processing: History, Overview and Perspective," *J. Supercomputing,* Vol. 7. No. 1, 1993, pp. 9-50.
2. P.E. Smith and G.S. Sohi, "The Microarchitecture of Superscalar Processors," *Proc. IEEE,* IEEE Press, Piscataway, N.J., Vol. 83, No. 12, Dec. 1995, pp. 1609-1624.
3. D. Sima, T. Fountain, and P. Kacsuk, *Advanced Computer Architectures,* Addison Wesley Longman, Harlow, England, 1997.
4. R.M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM J. Research and Development,* Vol. 11, No. 1, 1967, pp. 25-33.
5. G.S. Tjaden and M.J. Flynn, "Detection and Parallel Execution of Independent Instructions," *IEEE Trans. Computers,* Vol. C-19, No. 10, 1970, pp. 889-895.
6. R.M. Keller, "Look-Ahead Processors," *Computing Surveys,* Vol. 7, No. 4, 1975, pp. 177-195.
7. D. Leibholz and R. Razdan, "The Alpha 21264: A 500 MIPS Out-of-Order Execution Microprocessor," *Proc. Compcon,* IEEE Computer Society Press, Los Alamitos, Calif., 1997, pp. 28-36.
8. G. Kurpanek et al., "PA-7200: A PA-RISC Processor with Integrated High Performance MP Bus Interface," *Proc. Compcon,* IEEE CS Press, 1994, pp. 375-382.
9. D. Hunt, "Advanced Performance Features of the 64-Bit PA-8000," *Proc. Compcon,* IEEE CS Press, 1995, pp. 123-128.
10. A.P. Scott et al., "Four-Way Superscalar PA-RISC Processors," *Hewlett-Packard J.,* Aug. 1997, pp. 1-9.
11. G. Lesartre and D. Hunt, *PA-8500: The Continuing Evolution of the PA-8000 Family,* Hewlett-Packard Co., Palo Alto, Calif., 1998, pp. 1-11.
12. G.F. Grohoski, "Machine Organization of the IBM RISC System/6000 Processor," *IBM J. Research and Development,* Vol. 34, No. 1, 1990, pp. 37-58.
13. S. White and J. Reysa, *PowerPC and POWER2: Technical Aspects of the New IBM RISC System/6000,* IBM Corp., Austin, Texas,1994.
14. M. Becker et al., "The PowerPC 601 Microprocessor," *IEEE Micro,* Oct. 1993, pp. 54-68.
15. B. Burgess et al., "The PowerPC 603 Microprocessor," *Comm. ACM,* ACM Press, New York, Vol. 37, No. 6, 1994, pp. 34-42.
16. S.P. Song et al., "The PowerPC 604 RISC

Microprocessor," *IEEE Micro,* Oct. 1994, pp. 8-17.

17. D. Ogden et al., "A New PowerPC Microprocessor for Low Power Computing Systems," *Proc. Compcon,* IEEE CS Press, 1995, pp. 281-284.
18. L. Gwennap, "IBM Crams Power2 Onto Single Chip," *Microprocessor Report,* Micro Design Resources, Sunnyvale, Calif., Vol. 10, No. 11, 1996, pp. 14-16.
19. D. Levitan et al., "The PowerPC 620 Microprocessor: A High Performance Superscalar RISC Microprocessor," *Proc. Compcon,* IEEE CS Press, 1995, pp. 285-291.
20. S.P. Song, "IBM's Power3 to Replace P2SC," *Microprocessor Report,* Micro Design Resources, Vol. 11, No. 15, 1997, pp. 23-27.
21. L. Gwennap, "MIPS R10000 Uses Decoupled Architecture," *Microprocessor Report,* Micro Design Resources, Vol. 8, No. 14, 1994, pp. 18-22.
22. L. Gwennap, "MIPS R12000 to Hit 300 MHz," *Microprocessor Report,* Micro Design Resources, Vol. 11, No. 13, 1997, pp. 1,6-7,17.
23. N. Patkar et al., "Microarchitecture of HaL's CPU," *Proc. Compcon,* IEEE CS Press, 1995, pp. 259-266.
24. L. Gwennap, "Intel's P6 Uses Decoupled Superscalar Design," *Microprocessor Report,* Micro Design Resources, Vol. 9, No. 2, 1995, pp. 9-15.
25. L. Gwennap, "Klamath Extends P6 Family," *Microprocessor Report,* Micro Design Resources, Vol. 11, No. 2, 1997, pp. 1, 6-8.
26. *Pentium III Processor, Product Overview,* Intel Corp, Santa Clara, Calif., 1999.
27. J.S. Liptay, "Design of the IBM Enterprise Sytem/9000 High-End Processor," *IBM J. Research and Development,* Vol. 36, No. 4, July 1992, pp. 713-731.
28. B. Burkhardt, "Delivering Next-Generation Performance on Today's Installed Computer Base," *Proc. Compcon,* IEEE CS Press, 1994, pp. 11-16.
29. *Cyrix 686MX,* Cyrix Corporation, Richardson, Texas, Order No. 94329-00, July 1997.
30. L. Gwennap, "NexGen Enters Market with 66-MHz Nx586," *Microprocessor Report,* Micro Design Resources, Vol. 8, No. 4, 1994, pp. 12-17.
31. M. Slater, "AMD's K5 Designed to Outrun Pentium," *Microprocessor Report,* Micro Design Resources, Vol. 8, No. 14, 1994, pp. 1-11.
32. B. Shriver and B. Smith, *The Anatomy of a High-Performance Microprocessor,* IEEE CS Press, 1998.
33. K. Diefendorff, "K7 Challenges Intel," *Microprocessor Report,* Micro Design Resources, Vol. 12, No. 14, 1998, pp. 1, 6-11.
34. L. Gwennap, "Nx686 Goes Toe-to-Toe with Pentium Pro," *Microprocessor Report,* Micro Design Resources, Vol. 9, No. 14, 1995, pp. 1, 6-10.
35. D. Sima, "Superscalar Instruction Issue," *IEEE Micro,* Sept.-Oct. 1997, pp. 29-39.
36. V. Popescu et al., "The Metaflow Architecture," *IEEE Micro,* June 1991, pp. 10-13, 63-71.
37. T. Monreal et al., "Delaying Physical Register Allocation Through Virtual-Physical Registers," *Proc. MICRO-32,* IEEE CS Press, 1999, pp. 186-192.
38. S. Wallace and N. Bagheryadeh, "A Scalable Register File Architecture for Dynamically Scheduled Processors," *Proc. 1996 Conf. Parallel Architectures and Compilation Techniques,* 1996, pp. 179-184.
39. A. González et al., "Virtual Registers," *Proc. Third Int'l Symp. High-Performance Computer Architecture,* IEEE CS Press, 1997, pp. 364-369.
40. A. González, J. González, and M. Valero, "Virtual-Physical Register," *Proc. Fourth Int'l Symp. High-Performance Computer Architecture,* IEEE CS Press, 1998, pp. 175-184.
41. S. Jourdan et al., "A Novel Renaming Scheme to Exploit Value Temporal Locality Through Physical Register Reuse and Unification," *Proc. MICRO-31,* IEEE CS Press, 1998, pp. 216-225.
42. J.E. Smith and A.R. Pleszkun, "Implementing Precise Interrupts in Pipelined Processors," *IEEE Trans. Computers,* IEEE CS Press, Vol. C-37, No. 5, 1988, pp. 562-573.
43. M. Johnson, *Superscalar Microprocessor Design,* Prentice Hall, Englewood Cliffs, N.J., 1991.
44. G.S. Sohi and S. Vajapayem, "Instruction Issue Logic for High Performance, Interruptible Pipelined Processors," *Proc. 14th ISCA,* IEEE CS Press, 1987, pp. 27-36.
45. D. Sima, "The Design Space of Shelving," *J. Systems Architecture,* Vol. 45, No. 11, 1999, pp. 863-885.

Dezsö Sima is the dean of the John von Neumann Faculty of Informatics at Budapest Polytechnic, Hungary. He has taught computer architecture at the Technical University of Dresden; at Kandó Polytechnic, Budapest; and at South Bank University, London; and been a guest lecturer on computer architectures at several European universities. He was the first professor to hold the Barkhausen Chair at the Technical University of Dresden. His research interests include computer architectures and computer-assisted teaching and learning. Sima holds an MSc degree in electrical engineering and a PhD degree in telecommunications, both from the Technical University of Dresden, Germany. He has authored more than 40 papers and a book used in advanced architecture courses. He served as president of the John von Neumann Computer Society in Hungary and is an IEE Fellow and a member of the IEEE.

Direct comments about this article to the author at Budapest Polytechnic, John von Neumann Faculty of Informatics, PO Box 112, H-1431 Budapest 8, Hungary; sima@bmf.hu.

IEEE Micro 2001 Editorial Calendar

January-February

Hot Interconnects

This issue focuses on the hardware and software architecture and implementation of high-performance interconnections on chips. Topics include network-attached storage; voice and video transport over packet networks; network interfaces, novel switching and routing technologies that can provide differentiated services, and active network architecture.

Ad close date: 2 January

March-April

Hot Chips

An extremely popular annual issue, Hot Chips presents the latest developments in microprocessor chip and system technology used to construct high-performance workstations and systems.

Ad close date: 1 March

May-June

Mobile/Wearable computing

The new generation of cell phones and powerful PDAs has made mobile computing practical. Wearable computing will soon be moving into the deployment stage.

Ad close date: 1 May

July-August

General Interest

IEEE Micro gathers together the latest details on new developments in chips, systems, and applications.

Ad close date: 1 July

September-October

Embedded Fault-Tolerant Systems

To avoid loss of life, certain computer systems—such as those in automobiles, railways, satellites, and other vital systems—cannot fail. Look for articles that focus on the verification and validation of complex computers, embedded computing system design, and chip-level fault-tolerant designs.

Ad close date: 1 September

November-December

RF-ID and noncontact smart card applications

Equipped with radio-frequency signals, small electronic tags can locate and recognize people, animals, furniture, and other items.

Ad close date: 1 November

IEEE Micro is a bimonthly publication of the IEEE Computer Society. Authors should submit paper proposals to micro-ma@computer.org, include author name(s) and full contact information.