The Design Space of Shelving

Dezső Sima

Institute of Informatics, Kandó Polytechnic Budapest, 19 Nagyszombat utca, 1034 Budapest, Hungary

Abstract

While using the direct issue mode, dependent instructions cause issue blockages and thus an issue bottleneck. Shelving is a technique to avoid this and to increase the sustained issue rate. It takes advantage of two concepts: (a) the decoupling of dependency checking from instruction issue and (b) significantly widening the instruction window that is scanned in each clock cycle for executable instructions. In this paper we identify and explore the design space of shelving. We first outline its main dimensions, then we present and discuss feasible design alternatives along three of its crucial dimensions. Finally, we point out which design choices have been made in important superscalar processors. For a concise graphical representation of the design space we make use of DS-trees.

Keywords: Shelving, Dispatching, Instruction issue, Microarchitecture of superscalar processors, ILP-processing

1. Introduction

Although shelving was introduced more than thirty years ago in scalar supercomputers of that time, it has come into widespread use only recently in high end superscalars. While pioneering the basic approaches of parallel instruction execution, that is replication of execution units (EUs) [1] and pipelining [2], designers of Control Data's 6600 and IBM's 360/91 made use of shelving to *avoid instruction issue blockages* caused by data dependencies. In this way, the sustained issue rate and the overall performance of their machines could be increased. Nevertheless, for a number of reasons, including the complexity of its implementation, and the slight performance gain in scalar processors, the concept of shelving was itself shelved for a quarter of a century. In recent years, this advanced technique has been 'reinvented' in order to increase the sustained issue rate and consequently, the performance of superscalar processors [3]-[24]. In Section 2 we first introduce the principle of shelving. In Section 3 we outline the design space of shelving by indentifying its major dimensions and presenting the feasible implementation alternatives as regards three of its dimensions: the scope of shelving, the layout of shelving buffers and the operand fetch policy. The design space is represented through the use of DS-trees described in [25], [26], [29]. Throughout our discussion of the implementation alternatives we identify their actual use in recent superscalar processors and pinpoint trends. In Section 4, we detail how shelving operates. Finally, in Section 5, we recap how shelving has spread in recent superscalar processors and indicate the main features of the shelving schemes used.

2. The principle of shelving

The limitations of the *direct issue mode* initially applied in superscalar processors raised the need for a more sophisticated issue scheme early on. In the direct issue mode, executable instructions are issued from an issue window directly to the EUs. In an n-way superscalar processor the *issue window* comprises the last "n' entries of the instruction buffer (I-buffer), as shown in Fig. 1. In each clock cycle decoded instructions in this window are checked for dependencies on previous instructions still in execution. In the absence of dependencies all 'n' instructions are executable and will be issued directly to the EUs. However, each dependent instruction in the window gives rise to an *issue blockage*. Depending on how effectively issue blockages are handled in the processor, they decrease more or less severely the sustained issue rate and cause an issue bottleneck [26]. For instance the sustained issue rate of a 4-way superscalar processor is expected to be as low as about 2 while executing a general purpose program [27], [28]. The direct issue mode severely limits the performance of the processor. Because of this, high performance processors are forced to employ a more advanced issue mode such as shelving.

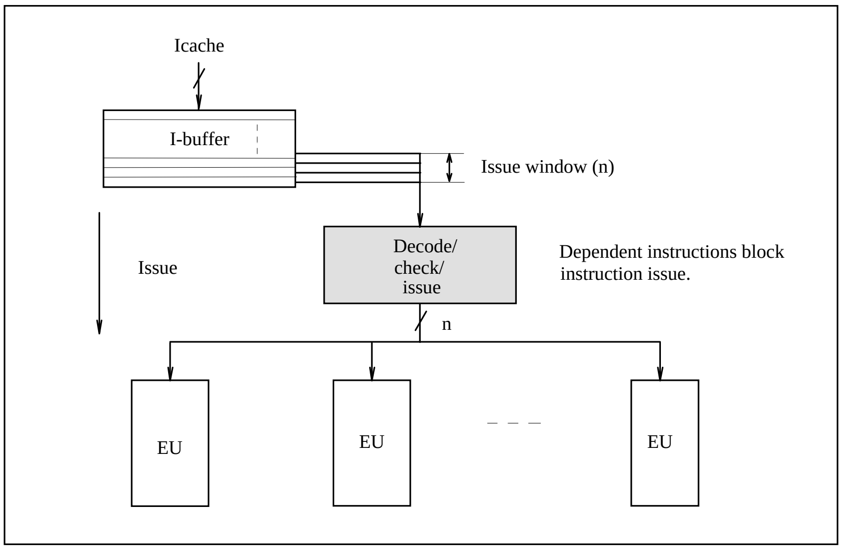


Figure 1: Principle of the direct issue mode.

(In the figure we assume a 4-way superscalar processor and individual reservation stations).

Shelving, also known as *shelved issue* or *indirect issue* avoids the limitations of the direct issue mode by utilising two ideas at the same time: (a) the decoupling of dependency checking from instruction issue, and (b) substantially widening the window which is scanned in the processor for executable instructions in each clock cycle. Shelving is implemented as follows: Instructions are first issued without checking for dependencies on previous instructions still in execution into special buffers called *shelving buffers*, that are provided in front of the EUs, as illustrated in Fig. 2. Thus, dependent instructions in the issue window do not cause issue blockages and an issue bottleneck is avoided. During a new subtask of the processing called *dispatching,* shelved instructions are checked for dependencies in each clock cycles and up to “m” independent instructions are forwarded to available EUs. Here “m” is the *dispatch rate,* which is the maximum number of instructions that can be forwarded in the same clock cycle to the EUs. Issued instructions held in the shelving buffers can be considered to be in the *dispatch window*. If the dispatch window is wide enough, say it comprises 20 to 40 instructions, and there is a wide enough execution bandwith available, it can be expected that the sustained dispatch rate will approach the issue rate. Thus, through shelving processor performance can be raised substantially. In Fig. 2, we outline how shelving is carried out, assuming a specific implementation alternative of the shelving buffers. Other possibilities for their implementation exist as well and are discussed later in Section 3.3.1.

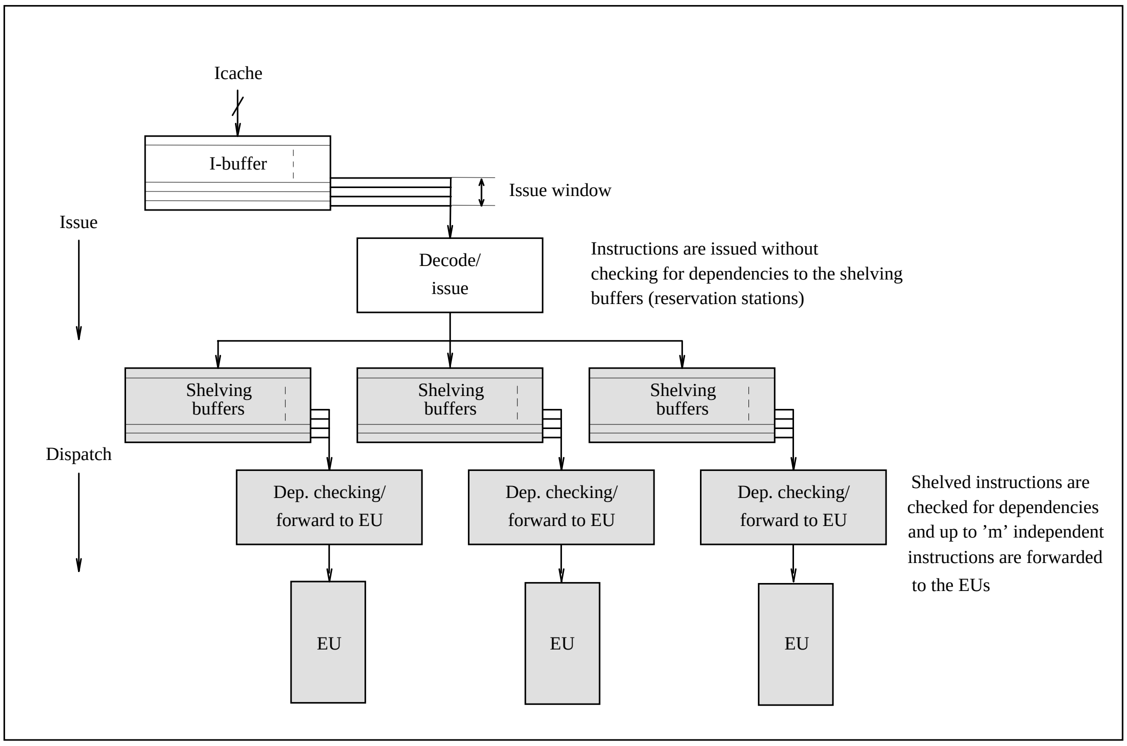

Icache
I-buffer
Issue window
Issue
Decode/ issue
Instructions are issued without checking for dependencies to the shelving buffers (reservation stations)
Shelving buffers
Shelving buffers
Shelving buffers
Dispatch
Dep. checking/ forward to EU
Dep. checking/ forward to EU
Dep. checking/ forward to EU
Shelved instructions are checked for dependencies and up to ’m’ independent instructions are forwarded to the EUs
EU
EU
EU

Figure 2.: The principle of shelving, assuming a 4-way superscalar proessor and individual reservation stations in front of each EU

Shelving, however has some peculiarities. First, as far as the instruction issue is concerned, even with shelving certain *resource constraints* can restrain the processor from issuing fewer instructions in a cycle than its issue rate is. There are two typical resource constraints: lack of free entries in buffers needed, such as in the reservation station, in the rename register file or in the reorder buffer and data path restrictions. However, issue blockages due to the above mentioned hardware constraints occur much less frequently than in the case of the direct issue mode.

Second, if *renaming* is employed along with shelving, instructions in the issue window must be checked for *inter-dependencies*, as detailed in [26]. Dependencies that do occur, however affect only the renaming mechanism, but do not cause issue blockages.

Third, shelving is predominantly used in connection with the *advanced superscalar issue policy*. This means that shelving is usually employed along *with speculative branch processing and register renaming* (as discussed in [29]). Speculative branch processing reduces a performance degradation due to control transfer instructions, while renaming removes issue blockages due to false register data dependencies, that is, due to WAR (Write After Read) and WAW (Write After Write) dependencies between register data [26]. Two implications follow. First, assuming a high enough execution and memory bandwidth, the hit rate of the speculative branch processing and the width of the dispatch window determines to what extent data and control dependencies confine performance. Thus, superscalar processors strive to increase the hit rate of branch prediction through introducing more and more intricate prediction schemes and to widen the width of the dispatch window by providing more and more shelving capacity. The second implication is that assuming speculative branch processing, register renaming and shelving, only true data dependencies, that is RAW (Read After Write) dependencies can prevent instructions held in the shelving buffers from being executed. In other words a shelved instruction becomes eligible for execution when all its operands are available. This criterion corresponds precisely to the *data flow principle of operation*.

A fourth remark concerns the *preservation of sequential consistency,* which calls for sustaining the logical integrity of the execution despite the fact that instructions are executed in parallel. In a sense, shelving makes processing more distributed, since the dispatch window is much wider than the issue window. To counteract this, shelving is typically used together with an advanced method of retaining sequential consistency called *instruction reordering.* For details see [26].

A final comment about terminology is also necessary. We have used two different terms, instruction *issue* and *dispatch,* to express different actions. The term ‘issue’ itself has a different meaning without shelving and with shelving. In the direct issue mode, ‘issue’ means forwarding of independent instructions directly to the EUs. When shelving is used it refers to the dissemination of both dependent and independent instructions to the shelving buffers. On the other hand, the term *‘dispatch’* is only applied in connection with shelving. It designates the dissemination of independent instructions from the shelving buffers to available EUs. While useful, this clear distinction is not common in the literature and both ‘issue’ and ‘dispatch’ are used in either interpretation.

Shelving is clearly a complex topic, and recent superscalar processors implement it in a variety of ways. The following sections offer a framework for this challenging diversity.

3. The design space of shelving

3.1 Overview

Shelving has a fairly complex design space. Its main dimensions are: the scope of shelving, the layout of the shelving buffers used, the operand fetch policy and the instruction dispatch scheme (Fig. 3). The *scope of shelving* declares whether shelving covers all data types or is restricted to a few of them. The *layout of the shelving buffers* specifies the infrastructural background of shelving. It is the *operand fetch policy* that decides whether operands are fetched in connection with issue or, in contrast, along with instruction dispatch. Finally, the *instruction dispatch scheme* specifies the details governing the selection and forwarding of instructions for execution. As instruction dispatch is a complex issue of its own, we restrict our discussion to the first three dimensions. Those interested can find more details on instruction dispatch in [24].

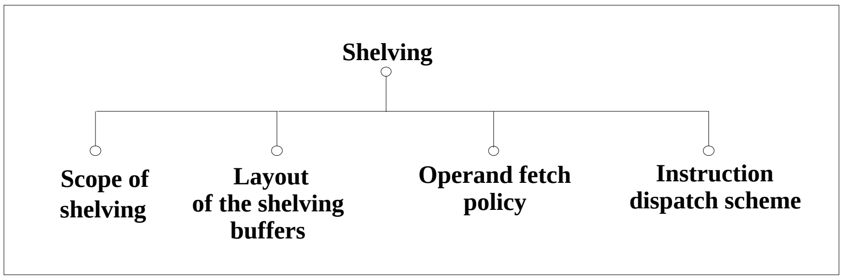


Figure 3: Design space of shelving

3.2 Scope of shelving

The scope of shelving indicates how comprehensively it is employed in a processor. *Partial shelving* is restricted to a few instruction types, whereas full shelving includes all instructions as shown in Fig. 4.

Partial shelving was employed in the introductory phase of shelving in a few superscalar processors, such as the Power1, Power2, MC 88110 and the R8000 (see Fig. 4). Three of these restrict shelving to FP-instructions, whereas the MC 88110 shelves stores and conditional branches. Partial shelving is clearly an incomplete solution to the problem of eliminating issue blockages caused by dependencies, thus most recent superscalar processors employ *full shelving*.

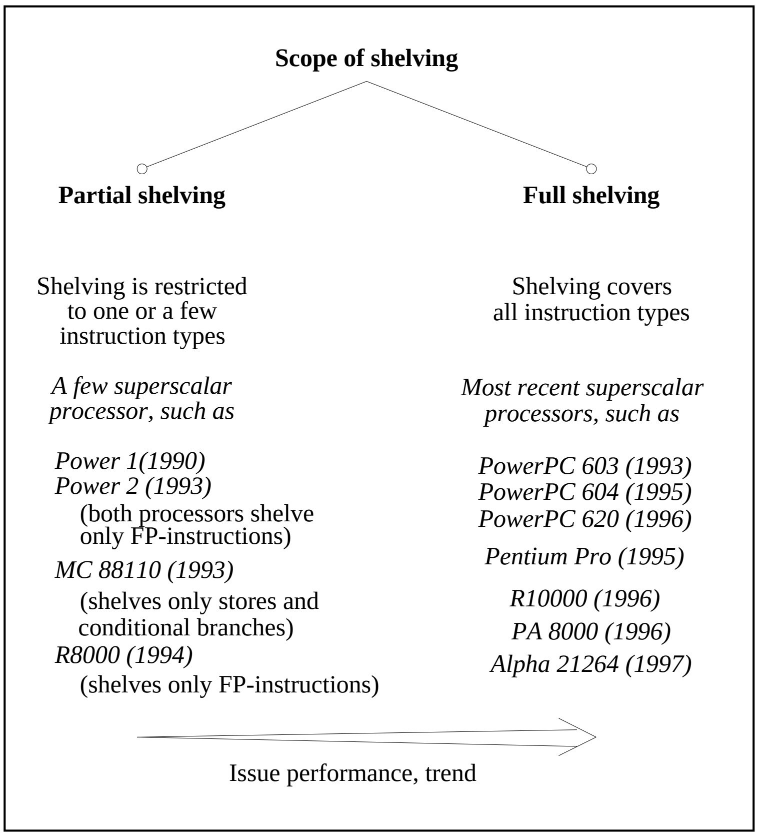

Scope of shelving
Partial shelving
Full shelving
Shelving is restricted to one or a few instruction types
Shelving covers all instruction types
A few superscalar processor, such as
Most recent superscalar processors, such as
Power 1(1990)
Power 2 (1993)
(both processors shelve only FP-instructions)
MC 88110 (1993)
(shelves only stores and conditional branches)
R8000 (1994)
(shelves only FP-instructions)
PowerPC 603 (1993)
PowerPC 604 (1995)
PowerPC 620 (1996)
Pentium Pro (1995)
R10000 (1996)
PA 8000 (1996)
Alpha 21264 (1997)
Issue performance, trend

Figure 4: Scope of shelving

3.3 Layout of the shelving buffers

Shelving buffers hold issued instructions until they can be forwarded for execution to an EU. Their layout can be distinguished by: the type and capacity of the buffers used and the number of their input- and output ports, as Fig. 5 shows.

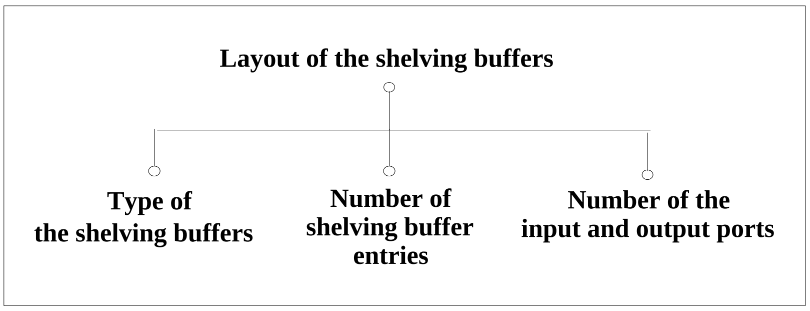

Layout of the shelving buffers
Type of the shelving buffers
Number of shelving buffer entries
Number of the input and output ports

Figure 5: Layout of the shelving buffers

3.3.1 Types of shelving buffers

We distinguish between two generic types of shelving buffers: *reservation stations* and *combined shelving buffers.* Reservation stations are used exclusively for shelving. In contrast, combined buffers are also used for reordering, and in some cases, for renaming as well, as shown in Fig. 6.

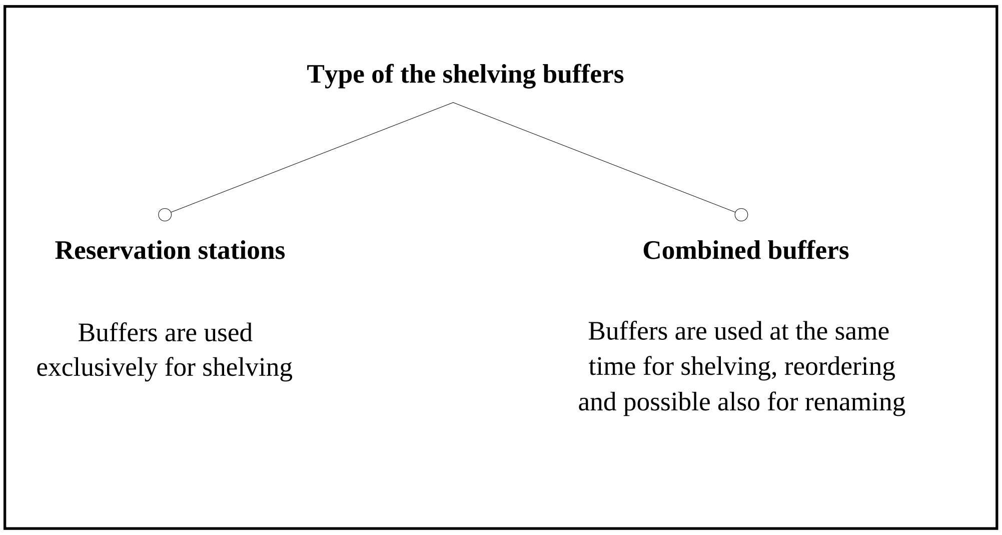

Type of the shelving buffers
Reservation stations
Buffers are used exclusively for shelving
Combined buffers
Buffers are used at the same time for shelving, reordering and possible also for renaming

Figure 6: Types of shelving buffers

Shelving buffers are in most cases implemented as *reservation stations*, a term which needs classification. When using the term *shelving buffer* we refer to all possible implementations, including reservation stations or combined buffer schemes. If we specifically use the term *reservation stations* we refer to the particular type of implementation.

In superscalar processors reservation stations are implemented according to one of three basic schemes, as indicated in Fig. 7.

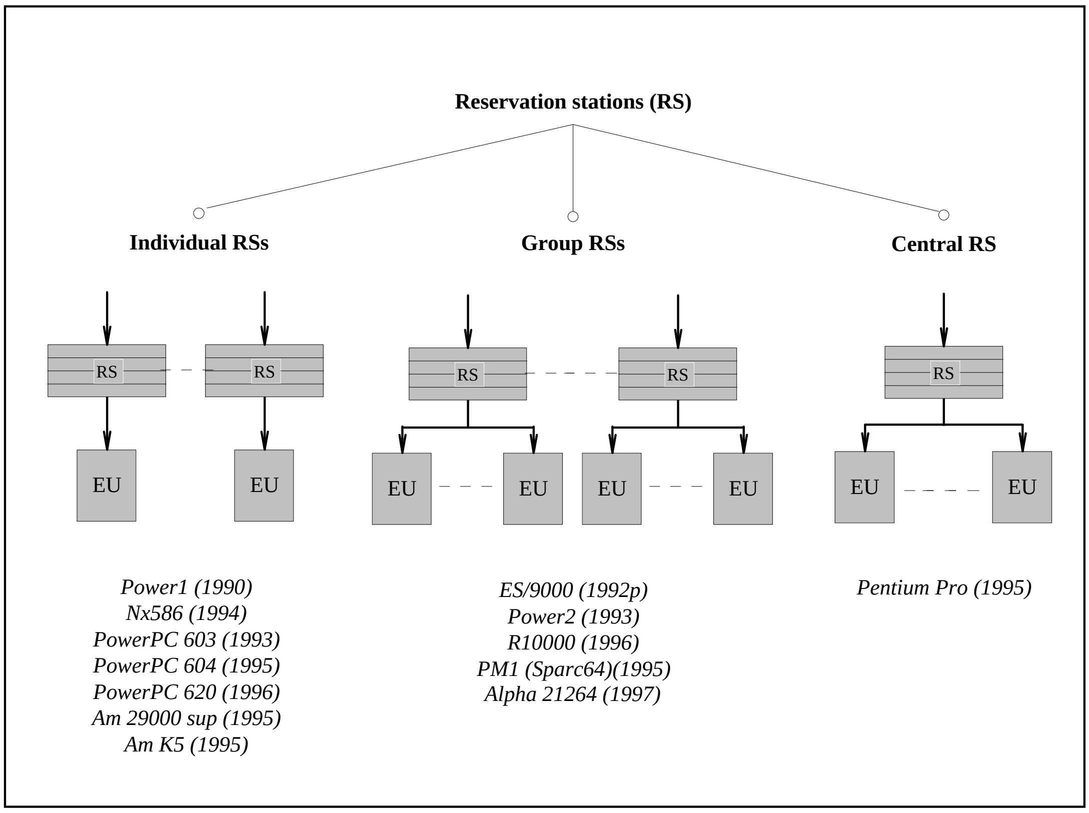


Figure 7.: Basic variants of reservation stations

In the simplest case, *individual reservation stations* are used in front of each EU. Here, instructions which are scheduled to be executed in a particular EU are first transferred into the associated reservation station preceding that EU. Usually, individual reservation stations provide enough space to hold only a small number of instructions, say 2 to 4. For example, in the PowerPC 620 the reservation stations in front of the three integer units and before the single FP-unit can hold two instructions each, whereas those associated with the load/store unit and the branch processing unit have places for three and four instructions, respectively.

Both the early implementations of shelving employed individual reservation stations. The CDC 6600 provided one-entry stations before each EU, whereas in the IBM 360/91 the FP-Add unit was equipped with a two-entry station and the FP-Multiply unit with a three-entry station. Further examples of superscalar processors using individual reservation stations are shown in Fig. 7.

In an alternative approach, reservation stations are implemented as *group stations*. Here, the same reservation station holds instructions for a whole group of EUs, which execute instructions of the same type. For instance, the R10000 has three group stations, one of these serves two FX-ALUs, another a single address unit, while the third supports four FP-units.

Evidently, group stations need to have more available buffer space than individual stations. For instance, the R10000 has three group stations with 16 positions each, or the PM1 (Sparc 64) has four group stations with 8 or 16 entries each. Since group stations serve more than one EU, they should be capable of receiving and dispatching more than one instruction in each cycle in order to avoid performance bottlenecks. As an example, each group station of the R10000 can accept 4 instructions in each cycle. Two of these stations (FX-RS and FP-RS) can dispatch two instructions, whereas the third one (Address RS) can only dispatch one instruction per cycle.

Group stations have an edge over individual stations since they are *more flexible* in dispatching instructions to EUs than individual stations. Thus, well designed group stations are better utilised than individual stations, assuming the same number of entries provided in both cases. The price for this is an increased complexity caused by multiple input and output ports and by the associated logic.

The last alternative is when a *central reservation station* serves all EUs. Clearly, a central station needs to have a higher capacity than group stations. Furthermore, it should be able to accept and dispath a larger number of instructions per cycle than group stations. A central station does, however, have some implementation disadvantages. First, it must have a word length equal to the longest possible data word. Second, as central stations are expected to be able to accept and dispatch more instructions per cycle than group stations, they are more expensive to implement. Nevertheless, PentiumPro opted for a central station. In the PentiumPro a 20 entry central reservation station serves all available EUs (10 altogether).

A quite different approach to the implementation of shelving buffers is to use a combined buffer *for shelving and renaming*, or *for shelving and reordering*, or for *shelving, reordering and renaming*, as shown in Fig. 8.

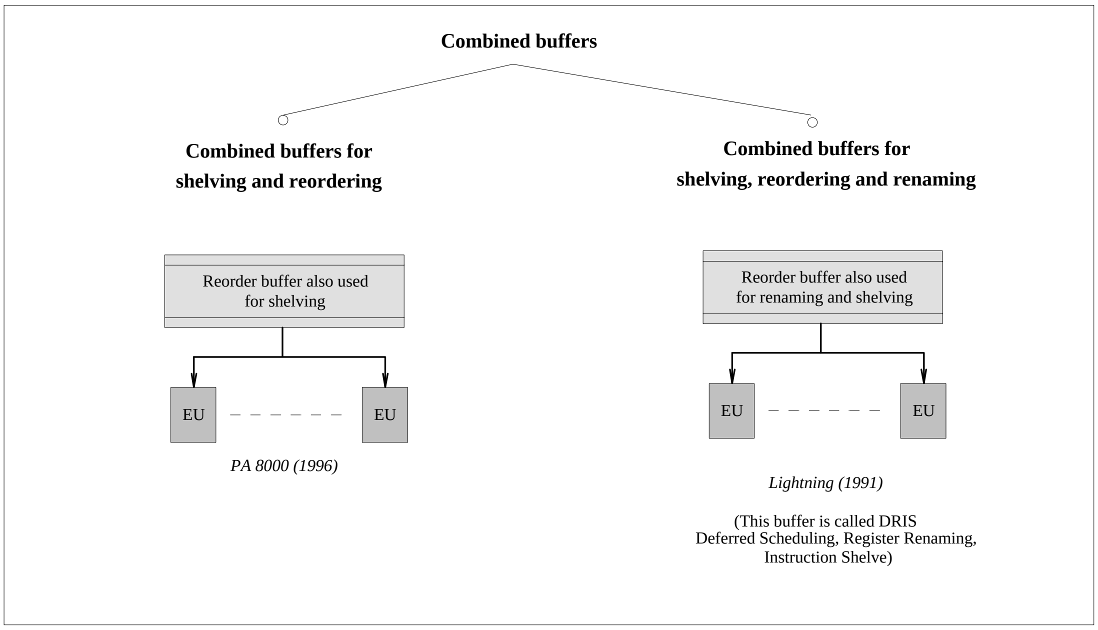

Combined buffers
Combined buffers for shelving and reordering
Combined buffers for shelving, reordering and renaming
Reorder buffer also used for shelving
Reorder buffer also used for renaming and shelving
EU
EU
EU
EU
PA 8000 (1996)
Lightning (1991)
(This buffer is called DRIS Deferred Scheduling, Register Renaming, Instruction Shelve)

Figure 8.: Use of combined buffers for shelving

In the first case, the reservation station is utilised also for renaming. For this reason, its entries are extended to hold the results of the instructions as well, until the instruction completes and the results will be written into the architectural registers, (see section 4.5). Clearly, all three types of reservation stations are suitable for a combined use with renaming. The IBM 360/91 employed this concept with individual reservation stations.

In both other cases, the *reorder buffer* (ROB), which assures the logical integrity of the program execution is extended to provide shelving, as in the PA8000 or even to provide both shelving and register renaming as well. A unique example of the latter is the Metaflow Lightning processor (1991), which was announced but never reached the market. In the Lightning, the combined structure was designated as *DRIS* (*D*eferred Scheduling, *R*egister Renaming, *I*nstruction *S*helve). Despite its complexity, the combined buffer approach is efficient and in the future we expect further processors to use this design option.

3.3.2 Number of shelving buffer entries

The progression from individual to group and then to central reservation stations brings with it the need to provide an increasing number of shelving places. Typically, individual reservation stations can shelve 2 - 4 instructions, group stations have 6, 12 or 16 entries, while the only implementation so far of a central reservation station, the PentiumPro, can hold 20 instructions. Clearly combined buffers have the same shelving capacity requirement as reservation stations do. The total number of entries in reservation stations provided account for the width of the dispatch window. Recent processors typically have a dispatch window of 15 and 40 entries, as Table 1 indicates.

Processor	**Width of the dispatch window**
PowerPC 603 (1993)	3
PowerPC 604 (1995)	12
PowerPC 620 (1996)	15
Nx586 (1994)	42
K5 (1995)	14
PM1 (Sparc64) (1995)	36
PentiumPro (1995)	20
R10000 (1996)	48
PA8000 (1996)	56
Alpha 21264 (1997)	35

Table 1: Width of the dispatch window in recent superscalar processors

3.3.3 Number of input and output ports

The last component of the layout of shelving buffers is the *number of input and output ports*. This element specifies how many instructions may be written into or read out from a particular shelving buffer in a single cycle.

First, let us consider the expected number of *output ports* (read ports). Obviously, individual reservation stations need only to forward a single instruction per cycle. A group or a central reservation station, on the other hand, needs to deliver multiple instructions per cycle, ideally as many as EUs are connected to it. It follows that individual reservation stations have a single output port, each group station is expected to provide multiple output ports, and central reservation stations are expected to have even more output ports.

Individual, group and central reservation stations require increasingly more *input ports* (write ports) as well. Processors with individual reservation stations often allow only one instruction per cycle to be issued into any one reservation station. Examples of this are the PowerPC 604 or the Nx 586. In contrast, recent superscalar processors with group reservation stations permit to transfer more than one or even all issued instructions into any of the reservation stations. For instance, the PM1 (Sparc64), which is a four-issue processor, can issue any combination of up to 2 FP-, 2 load/store-, 1 branch and 4 integer instructions into the corresponding group reservation stations. But no more than two integer instructions of these can may be complex ALU operations such as shifts, multiplications or divisions. In the four-issue R10000, even all four instructions issued can be forwarded into any of the three group stations. As far as combined shelving buffers are concerned they have similar input and output port requirements as corresponding reservation stations do.

3.4 Operand fetch policies

3.4.1 Overview

Closely connected with shelving is the policy governing how processors fetch operands. The fetch policy is either *issue bound* or *dispatch bound*, as indicated in Fig. 9.

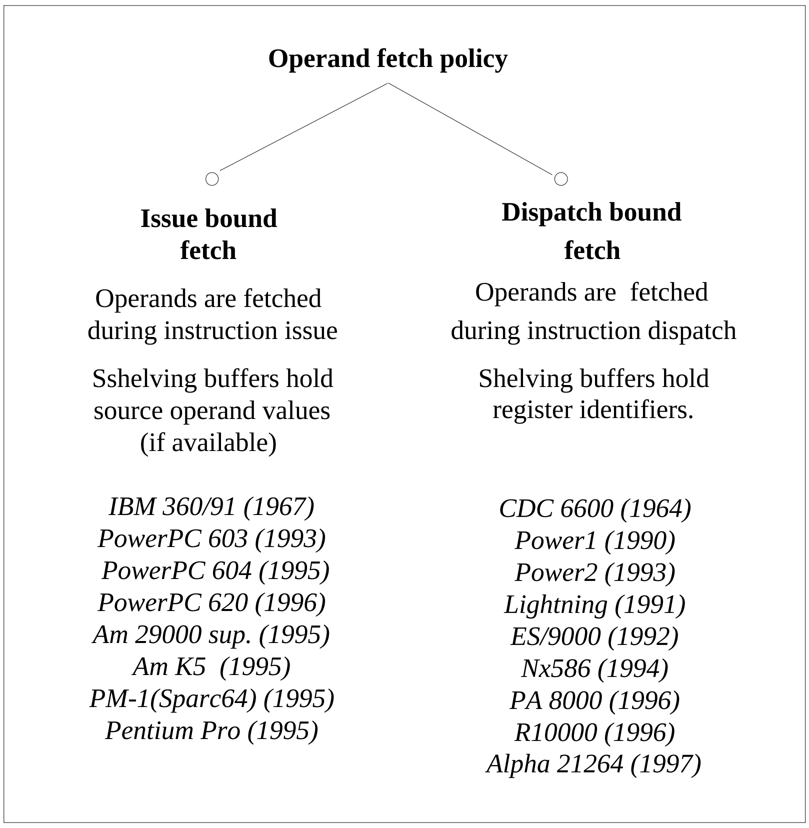


Figure. 9: Operand fetch policies

The *issue bound fetch policy* means that operands are fetched during instruction issue. In this case shelving buffers basically hold instructions with their operand values, requiring that the buffers be quite long to provide space for all the source operands. The other basic alternative is when operands are fetched during dispatching, called the *dispatch bound fetch policy***.** In this case, shelving buffers can be much shorter, since they contain instructions with their register identifiers.

In the following, we describe both operand fetch policies mentioned. In our discussion we assume individual reservation stations and a common register file for both FX- and FP-data. Furthermore, we assume that no register renaming is used. These assumptions do not affect the principles discussed, but allow us to focus on the vital points. Subsequently, we extend our discussion to the split register scenario and to the case when renaming is used as well.

3.4.2. The issue bound fetch policy

Fig. 10 shows the principle of the issue bound operand fetch policy. In this case, while issuing the instructions, the referenced source register numbers are forwarded to the register file in order to fetch the source operands. In addition, the operation codes (OC), the destination register numbers of the issued instructions (Rd), and the fetched operand values (Op1 and Op2) are written into the allocated reservation stations.

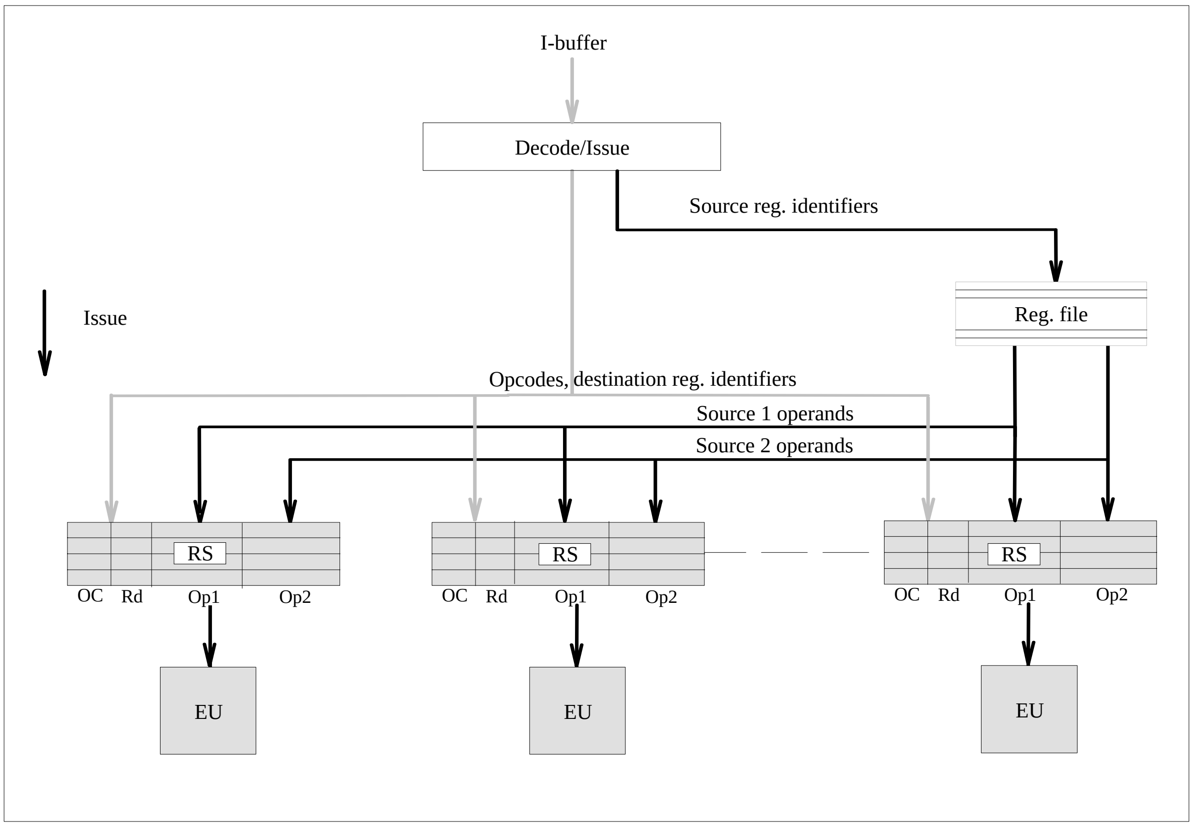


Figure 10.: The principle of issue bound operand fetching. In the figure we assume a common register file for both FX- and FP-data and individual reservation stations

3.4.3. The dispatch bound operand fetch policy

An alternative to the issue bound fetch policy we have discussed is the dispatch bound operand fetch policy. In this case operands are fetched in connection with instruction dispatch rather than with instruction issue.

As Fig. 11 shows each entry of the reservation station holds the operation code (OC), the destination register (Rd) and the source register numbers (Rs1, Rs2). During dispatch, the operation codes and destination register identifiers of the dispatched instructions are forwarded from the reservation stations to the associated EUs, and the source register identifiers are passed to the register file. After fetching, the source operands are gated into the inputs of the corresponding EUs.

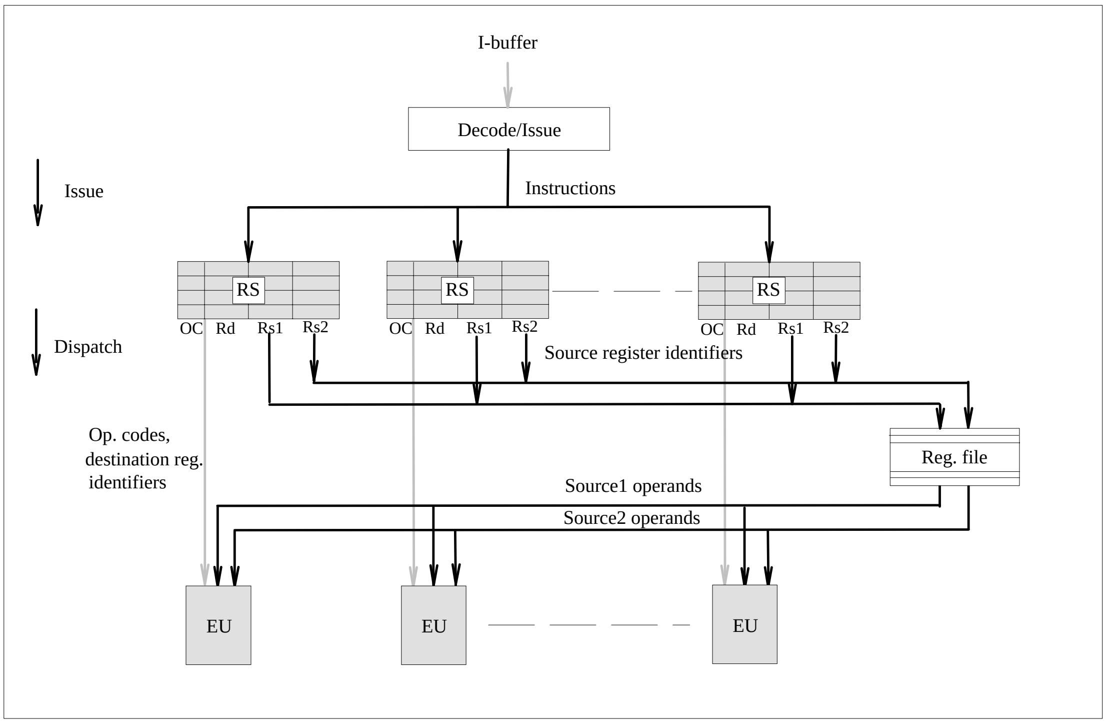


Figure 11: The principle of dispatch bound operand fetching. In the figure we assume a common register file for both FX- and FP-data and individual reservation stations

3.4.4. Operand fetching assuming split register files for FX- and FP-data

In our introduction of operand fetch policies we assumed a *common* register file for both FX- and FP-data. However, most recent architectures such as the x86, R, PA, Alpha and PowerPC architectures, use *distinct* register files for FX- and FP-data. Accordingly, the corresponding lines of processors implement distinct FX- and FP-register files. There are only a few recent processors which have a single FX register file, such as the Am 29000 architecture, which supports only FX-data.

Another exception is the Nx586 which implements only the FX-part of the architecture in the main processor, while the FP-part is realized by an FP-coprocessor chip, called the Nx587.

In the following we revisit both operand fetch policies assuming distinct register files for FX- and FP-data.

As Fig. 12 and 13 illustrate, when there are distinct FX- and FP-register files the microarchitecture has a symmetrical internal structure. FX-instructions and FX-data on the one hand, and FP-instructions and FP-data on the other, are processed now by two distinct and more or less symmetrical processor parts.

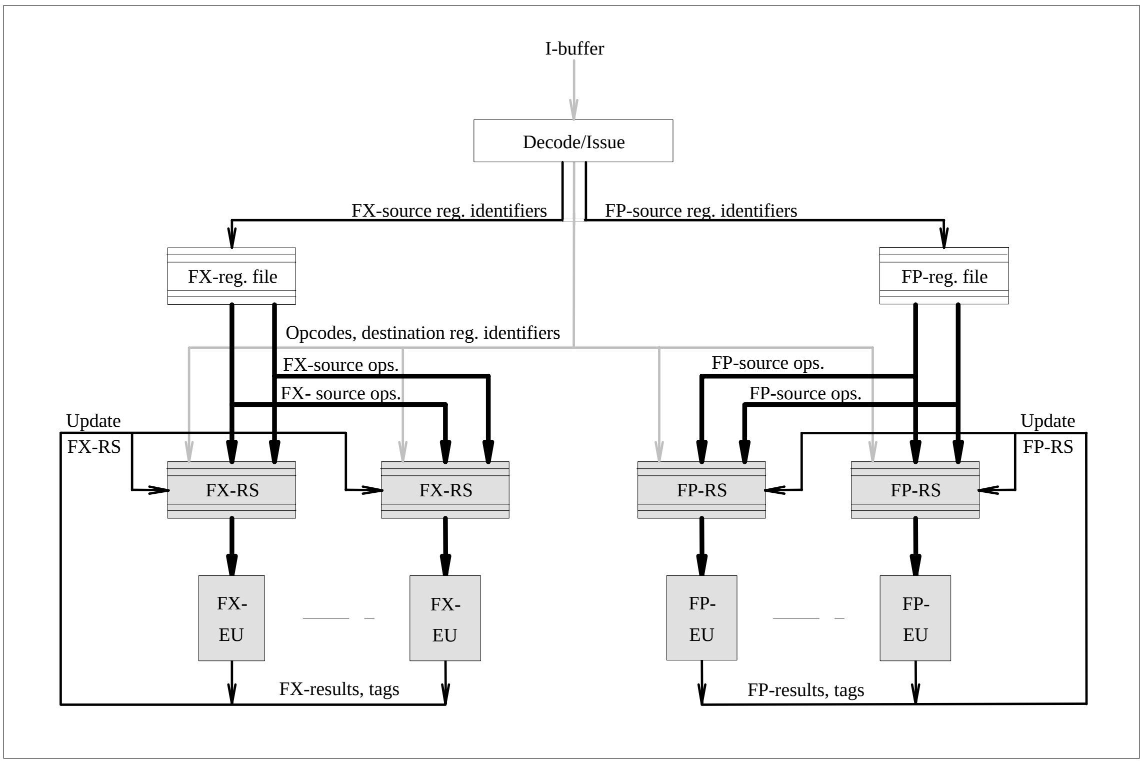


Figure 12: Issue bound operand fetch assuming split FX- and FP-register files

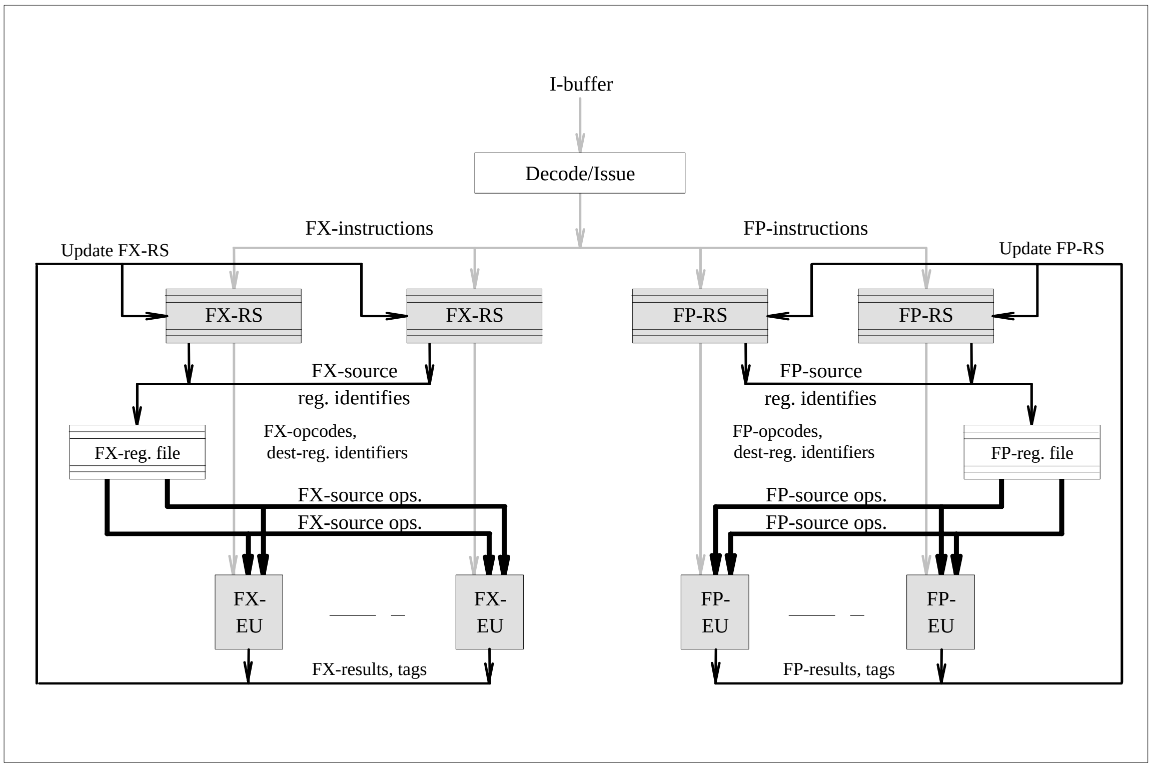


Figure 13: Dispatch bound operand fetch assuming split FX- and FP- register files

3.4.5. Operand fetching using renaming

In the absence of register renaming, all required register operands are supplied by the architectural register files. When register renaming is used, however, a quite different situation arises since in this case *intermediate results* are held in an additional register space, called the *rename buffer file*. Intermediate results are generated by instructions with renamed destination registers. They become *final results* and part of the program state when instructions *complete* in program order. Here, we note that clearly more than one intermediate result may belong to a particular architectural register since multiple valid renamings of the same architectural register may happen. Now if operands are fetched during instruction issue, obviously, for each referenced source operand its latest renamed value needs to be accessed. This is achieved by a parallel prioritised access to both the appropriate architectural and the rename buffer file. If the rename buffer file does not hold an intermediate result belonging to the addressed architectural register, the required (latest) operand value is held in the architectural register file. Otherwise, the latest value of the required source register is the youngest renamed value. In accordance with the renaming scheme employed, this additional register space may be implemented either as rename register files, as a common register pool for architectural and rename registers or as an extension of the reorder buffer (ROB), as detailed in [26].

3.4.6. Comparison of the issue bound and the dispatch bound operand fetch policies

In the absence of register renaming, operand fetch policies can be assessed based on three aspects: (a) the impact on the time-critical decode-issue path, (b) the complexity of the shelving buffers and (c) the number of output ports in the register files.

In two of these the dispatch bound operand fetch policy is more advantageous than the issue bound policy. The critical *decode/issue path* is shorter, and *shelving buffers* are *less complex,* since register identifiers require considerably less buffer space than operand values. However, the comparison is far more complex when we look at the number of output ports.

The *number of the required output ports* (read ports), is a major influence on the floor space required for the implementation of the registers. In the case of the *issue bound* operand fetch policy, register files have to supply the source operands during the issue process. This requires as many output ports as there may be operands in the issued instruction group. If there are no restrictions on the issuable instruction mix, then the FX- and the FP-register files each must be able to deliver all source operands for the maximum number of issuable instructions. For example, in a 4-way superscalar processor, the FX-register file typically has to offer 8 and the FP-register file 12 output ports, assuming up to two FX- and three FP-source operands per instruction. If instruction mix restrictions are in place, for instance if 4-way superscalar processor can’t issue more than two FP-instructions in the same cycle, the requested number of output ports in the FX- and FP-register files will be reduced accordingly.

The situation is different when we consider the output port requirements of the *dispatch bound* operand fetch policy. In this case each of the FX- and FP-register files has to have as many output ports as required for the maximum number of dispatched instructions. If, for example, up to four FX-instructions and up to two FP-instructions may be dispatched in the same cycle, the FX- and the FP-register files must have 8 and 6 output ports, respectively, assuming again up to two FX- and three FP-source operands per instruction.

This basic information allows us to compare the output port requirements of the issue bound and dispatch bound operand fetch policies. Assuming that there are no issue mix restrictions, for high end superscalar processors the following estimation can be made. Using the issue bound fetch policy, the FX- and FP-register files should each be able to deliver operands for as many instructions as the issue rate. In other words, the FX- and FP-register files together should be able to serve twice as many instructions as the issue rate. By contrast, in the case of the dispatch bound fetch policy the FX- and FP-register files together should serve as many instructions as the dispatch rate. Since the dispatch rate is usually higher than the issue rate, but less then twice the issue rate (see Table 2), in the absence of issue mix restrictions we expect the dispatch bound fetch policy to require altogether fewer output ports than the issue bound fetch policy. In a typical situation where the dispatch rate is six and the issue rate is four, the FX- and FP register files need to serve six instructions using the dispatch bound fetch policy and eight using the issue bound fetch policy.

Processors/Year of volume shipment	Issue rate instr./cycle	Dispatch rate[1] instr./cycle
PowerPC 603 (1993)	3	3
PowerPC 604 (1995)	4	6
PowerPC 620 (1996)	4	6
Power2 (1993)	4/6[2]	10
Nx586 (1994)	3/4[3.4]	3/4[3.4]
K5 (1995)	4[4]	5[4]
PentiumPro (1995)	4	5[4]
PM1 (Sparc 64) (1995)	4	8
PA8000 (1996)	4	4
R10000 (1996)	4	5
Alpha 21264 (1997)	4	6

Comments:

1 Because of address calculations performed separately, the given numbers are usually to be interpreted as operations/cycle. For instance, the Power2 performs maximum 10 operations/cycle, which corresponds to 8 instructions/cycle.

2 The issue rate is 4 for sequential mode and 6 for target mode.

3 Both rates are 3 without an optional FP-unit (labelled Nx587) and 4 with it.

4 Both rates are related to RISC operations (rather than to the native CISC operations) performed by the superscalar RISC core.

Table 2. Comparison of issue and dispatch rates of recent superscalar processors

If *renaming is employed* there are two opposite implications. On the one hand, as far as the output port requirements are concerned, the issue bound operand fetch policy turns out to be even more disadvantageous. The reason is that the issue bound policy requires more output ports than the dispatch bound policy not only in the architectural register files but in the rename buffers as well. On the other hand, if renaming is implemented by a merged architectural and rename file, or by separate FX- and FP-rename register files, the issue bound operand fetch policy has an edge over the dispatch bound policy since in this case the reclaiming of rename buffers is more straightforward than when using the dispatch bound fetch policy.

The output port requirements of the issue bound fetch policy can be reduced if some issue restrictions are instituted, which of course impede performance. Then in a typical situation we expect no significant difference concerning the output port requirements of the issue bound and the dispatch bound operand fetch policies. Obviously, in a given case a more precise statement can easily be made by inspecting the specifications involved.

In summary, based on our foregoing assessment, the dispatch bound fetch policy seems to be more benefitial than the issue bound policy.

4. Detailed operation of shelving

4.1. Overview

In this section we describe in detail how shelving is carried out. For simplification however, our description is based on an example and assumes a straightforward scenario, as indicated in Fig. 14a-14f. Our scenario consists of a common register file, a single individual reservation station and issue bound operand fetching. Furthermore, we presume an ROB for preserving the sequential consistency of instruction execution and register renaming within the ROB (as done in the Am29000 superscalar and in the K5).

The sequential consistency of the program execution is preserved by the ROB in the following way: (a) All issued instructions are written into the ROB in program order. They are stored into subsequent free entries pointed to by the Head pointer. (b) Instructions may complete, that is write their results into the architectural register file, or into the memory again only in program order. The instruction which comes next in the program order, as indicated by the Tail pointer.

In our example we assume that register renaming is performed within the ROB. This means that the results of the instructions are first written into the ROB-entry which is allocated to the instruction generating the result, rather than into the specified architectural register. Accordingly, each ROB-entry must have three fields, as indicated in Fig. 14a-14f. These fields are as follows: (a) the ‘Rd’ field, which holds the number of the destination register of the associated instruction, (b) the ‘Value’ field, which is provided to store the generated result, and (c) a status bit (‘E’) indicating whether the associated instruction has already been executed. Clearly, the ‘Value’ field is valid only if the E-bit is set. In addition, we extend the ROB-entries with a further field (‘OC’). Although not required for the operation of the ROB, this field, which indicates the operation codes, is a helpful reference to the instructions held in the ROB.

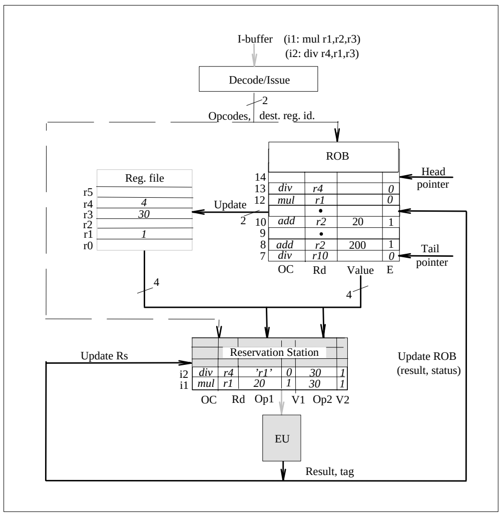


Figure 14a: Issue of the instructions i1 and i2

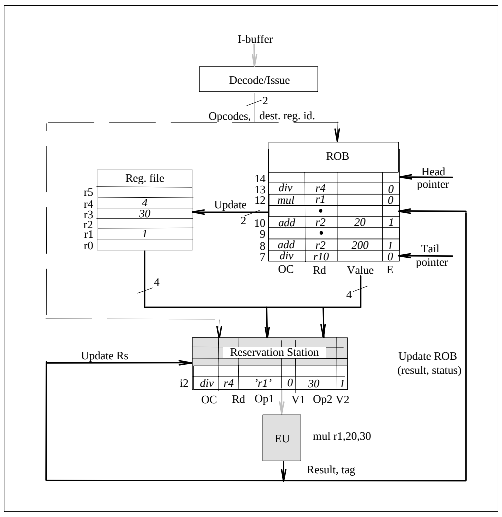


Figure 14b: Dispatching of instruction i1

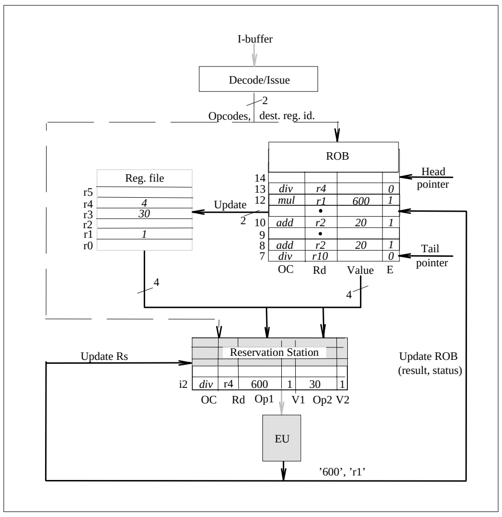


Figure 14c: Generating the result of i1 and updating of both the reservation station and of the ROB

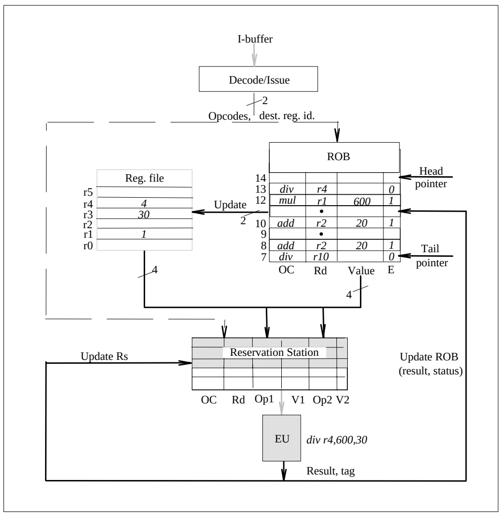


Figure 14d: Dispatching of the instruction i2

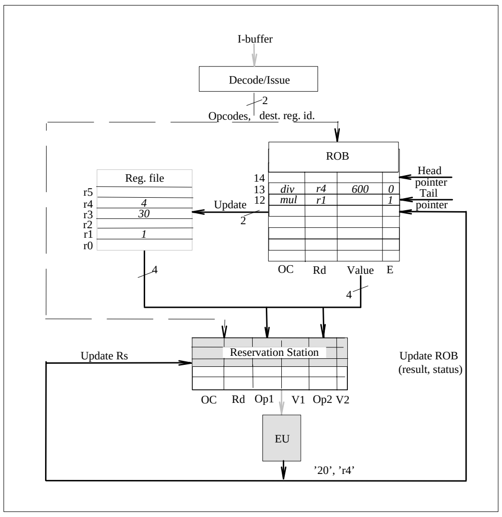


Figure 14e: Execution status before completing of instruction i1

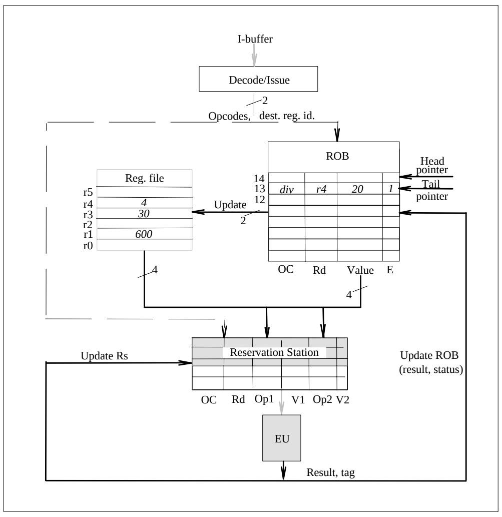


Figure 14f: Completing of instruction i1, and updating of the ROB

The overall operation is as follows. First, instructions are issued into the reservation station and into the ROB. While instructions are issued into the reservation station referenced source operands are fetched and their availability is marked in the allocated entries of the reservation station. A shelved instruction is launched for execution if all of its source operands are available. Execution results are used to update both the reservation station and the ROB. Finally, when an instruction comes next in the logical order of program execution (that is it completes), its result is written to the architectural register file to update the program state.

In our following description of the operation we distinguish among the following subtasks:

a, instruction issue,

b, instruction dispatch,

c, updating of the reservation station and of the ROB, and

d, completing of the instructions.

In our example we assume that the following two instructions are issued and processed:

*i1: mul r1, r2, r3; //r1 ⇐ (r2) * (r3)*

i2: div r4, r1, r3; //r4 ⇐ (r1) - (r3)

4.2. Instruction issue

Assuming at least a two-way superscalar processor, instructions are issued at the same cycle both into the ROB and into the reservation station. While issuing i1 and i2 into the ROB, their destination register numbers (r1 and r5) are written into the appropriate fields of the next two free entries, pointed to by the Head pointer, and the associated E-bits (Executed bits) are reset, as illustrated in Fig. 14a.

Instructions i1 and i2 are also written into the reservation station and their operands are fetched. As shown in Fig. 14a, each entry of the reservation station holds the operation code (OC), the number of the destination register (Rd), the operand values (Op1 and Op2) and two status bits indicating the availability of the associated operands (V1 and V2). Accordingly, the operation codes (‘mul’ and ‘div’) and the destination register numbers (r1 and r5) of the issued instructions are stored in the associated fields. However, fetching of the operands is a more complicated task, since for each source operand its latest value needs to be fetched, as described in Section 3.4.5. For this reason both the architectural register file and the rename buffer file, in our case this is the ROB, need to be accessed at the same time. Let us assume that the ROB looks for the latest values of the operands in an associative way. (We note that other possibilities for accessing the operands exist as well and are discussed in [26]). During accessing of both files three different cases may occur: (a) a referenced operand is not found in the ROB, (b) a referenced operand is held in the ROB and is available, and (c) a referenced operand is held in the ROB but not available.

(a) A referenced operand does not exist in the ROB

If the ROB does not hold an entry for the referenced source register, the latest value of that source register is stored in the corresponding architectural register. In our example we assume that no ROB entry contains the destination register r3. Than it’s latest value has to be fetched from the corresponding architectural register, which is ‘30’.

(b) A referenced operand is held in the ROB and is available

If the referenced register number is found in the ‘Rd’ field of one or more ROB entries, it is always the latest value that needs to be fetched. For instance, in our example r2 occurs twice in the ROB. From these, the value of ‘20’ is the latest. This value is available, as indicated by the associated E-bit, thus the value of ‘20’ will be fetched for r2.

(c) A referenced operand is found in the ROB but is not available

Clearly, it can happen that the latest value of a referenced operand needs to be accessed from the ROB but is not yet available, since its calculation is still in progress. In this case, instead of the requested value, a unique identifier is forwarded to the reservation station, which is usually the index of the rename buffer. In our case this is the same as the ROB index of the instruction which generates the requested value. In our example r1 is held in entry no12 of the ROB, but its value is not yet available. Thus, instead of its value the ROB index (‘12’) needs to be written into the reservation station. However, to allow a better distinction between data values and tags, in the figures we identify this tag symbolically by its register number (‘r1’).

4.3. Instruction dispatch

Instructions shelved in the reservation station are scanned in each clock cycle to check for executable ones. If an instruction has all of its operands available, it is executable and can be forwarded to an available EU. In Fig. 14b. the ‘mul’ instruction has both of its operands available and will be forwarded to the associated EU.

4.4. Updating of the reservation station and of the ROB

If an EU produces a result both the reservation station and the ROB need to be updated. *To update the reservation station* the generated result value and its tag (in our example ‘600’ and ‘r1’), are forwarded to it, as indicated in Fig. 14c. Updating requires an associative search in all entries of the reservation station to see whether any of their source operand fields holds the result tag received. All matching tags are then replaced with the actual result value and the associated V-bits are set. In our case the only hit is the first source operand of the ‘div’ instruction. As a consequence the corresponding value of ‘600’ will be written into the Op1 field and its V-bit set. Subsequently, in the next cycle when the reservation station is checked for executable instructions, this operand will appear as already available and instruction i2 can be dispatched in the same way as discussed before for i1 (see Fig. 14d.).

We point out that reservation stations must be updated globally since an operand value not available in a reservation station is not necessarily produced by the associated EU. Thus, for updating, all results must be forwarded along with their tags to any reservation station which may hold instructions of the same type (e.g. FX-, FP- or L/S-instructions), as shown in Fig. 15.

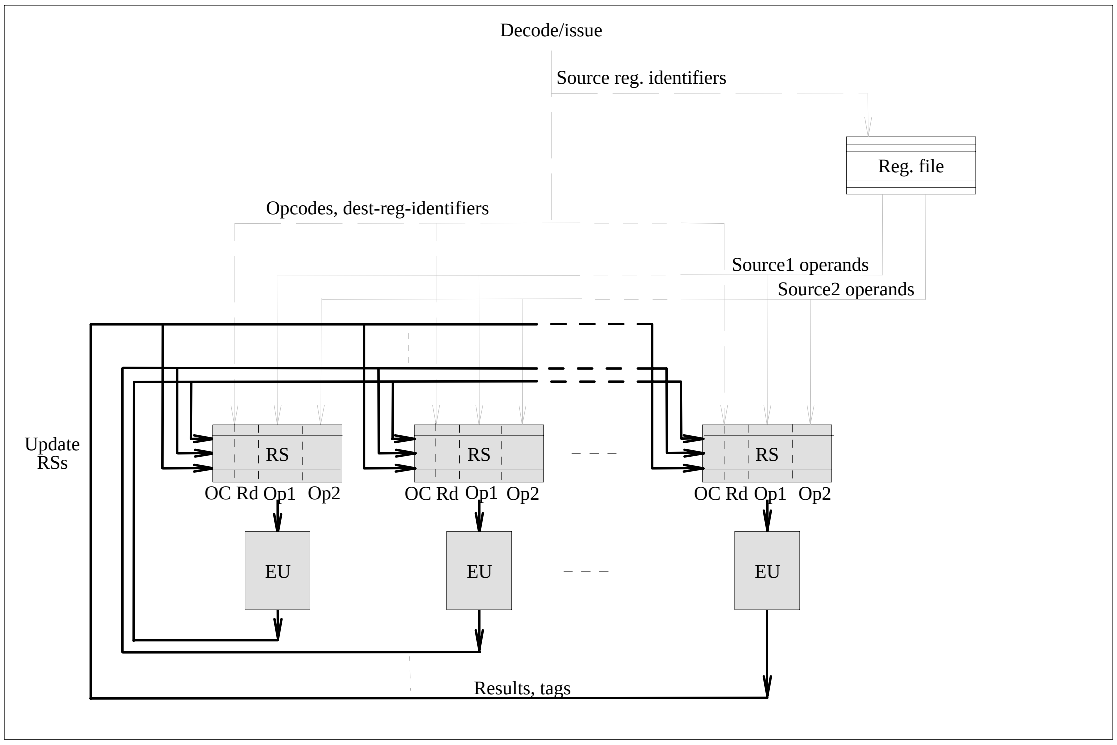


Figure 15.: Updating multiple individual reservation stations

It follows that in the case of multiple reservation stations as many result buses are required as there are EUs executing the same type of instructions, and that in each reservation station multiple associative searches must be carried out, one for each tag supplied by one of the result buses.

Updating of the ROB is more straightforward, since the tags which accompany the generated results are the ROB indices. Thus, to update the ROB a simple indexed access is needed for each finished instruction. This means in our example, (see Fig. 14c) that ROB entry no. 12 is updated by writing the result ‘600’ into the ‘Value’ field and setting the associated E-bit to indicate that the instruction is executed already and its result is available.

4.5. Completing of instructions

The ROB allows only the instruction which comes next in program order to complete. During completion an instruction updates the program state by writing its result into the architectural register file or into the memory. For this reason, the ROB maintains a Tail Pointer that points to the instruction, which is next in the program order. This instruction is allowed to complete, if it has been finished (E=1). When this instruction has been completed, the associated ROB-entry is released and the Tail Pointer is stepped to check the next instruction.

In our example let us assume that in a particular clock cycle instruction i1 is next to be completed, since the Tail pointer points to it as shown in Fig. 14e. It follows that during completion the generated result of r1 (‘600’) is written into the register file as illustrated in Fig. 14f and the Tail pointer of the ROB is increased by one.

So far, we have discussed the operation of shelving assuming the use of the issue bound fetch policy. When the *dispatch bound* fetch policy is employed shelving is carried out in essentially the same way. The main difference is that in this case the reservation station holds the source register identifiers rather than the operand values.

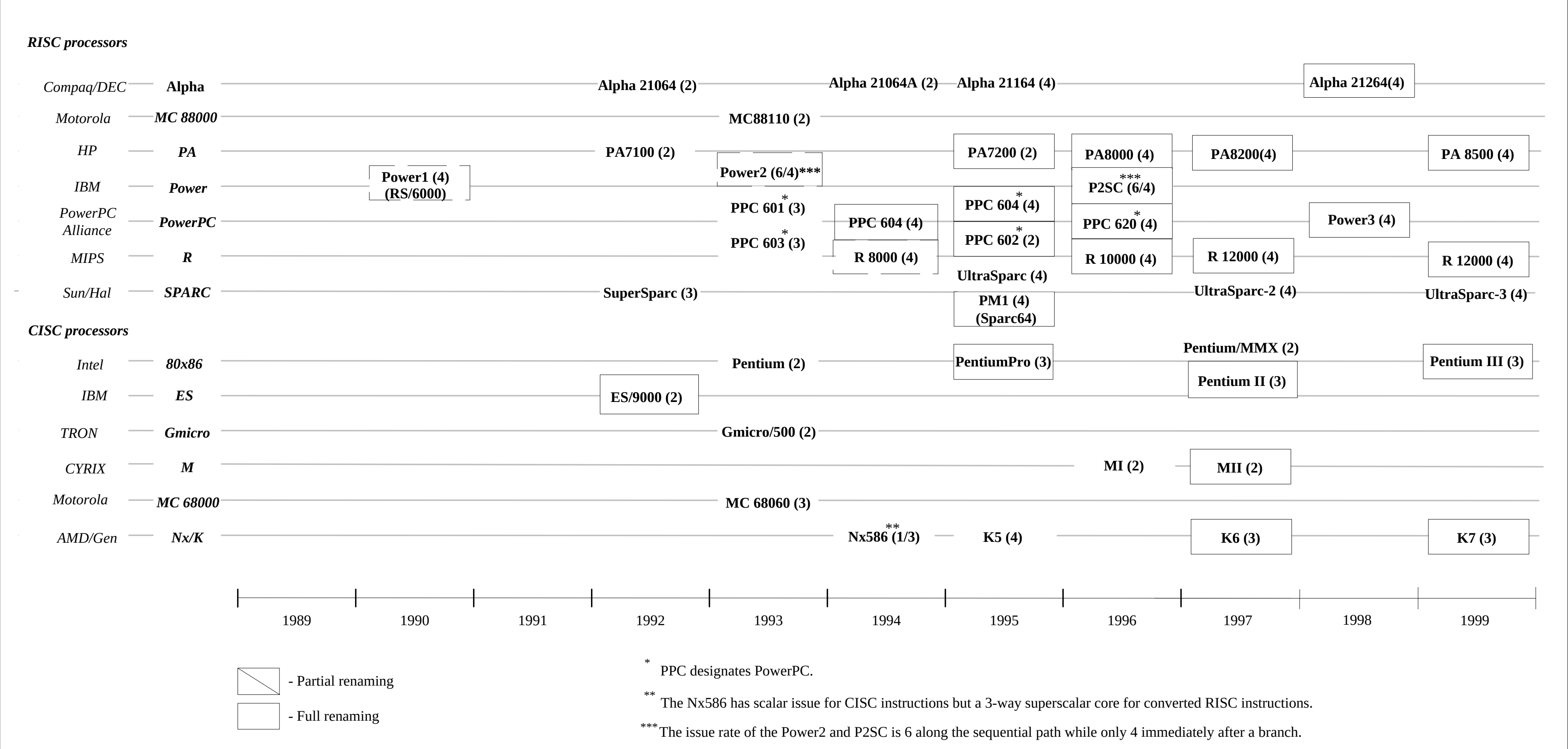
5. The use of shelving in superscalar processors

5.1. Spread of shelving

As we alluded to earlier, the concept of shelving has had a somewhat checkered past. After its invention in the late 60s, shelving slept like Sleeping Beauty for more than 25 years. It was only in 1990 when shelving appeared again in the RS/6000, later renamed to Power1, as shown in Fig. 16. Not surprisingly Power1 implemented shelving only partially, since it shelved only FP-intructions. In 1993-95 three further processors followed which also employed partial shelving; the MC88110, the Power2 and the R8000. Full shelving was first introduced in 1992 by IBM in its high end models of the ES/9000 line of mainframe processors, and subsequently in the PowerPC 603. Finally, shelving came to widespread use between 1995 and 1997 in all major lines ultimately also in the Alpha family.

5.2 Basic shelving schemes and their use in recent superscalar processors

In the design space discussed each feasible combination of design choices results in a possible shelving scheme. This yields a large number of possible schemes. However, we can simplify our discussion by taking into account that (a) most recent superscalar processors employ full shelving, and (b) in a qualitative discussion we can omit quantitative aspects of the design space. Thus in our subsequent discussion of shelving schemes we consider a reduced design space which consists only of two crucial design aspects; which are the type of shelving buffers used and the operand fetch policy employed. The associated shelving schemes will be designated as *basic shelving schemes* and are depicted in Fig. 17. In the figure, we also indicate which basic shelving scheme is used in recent superscalar processors.

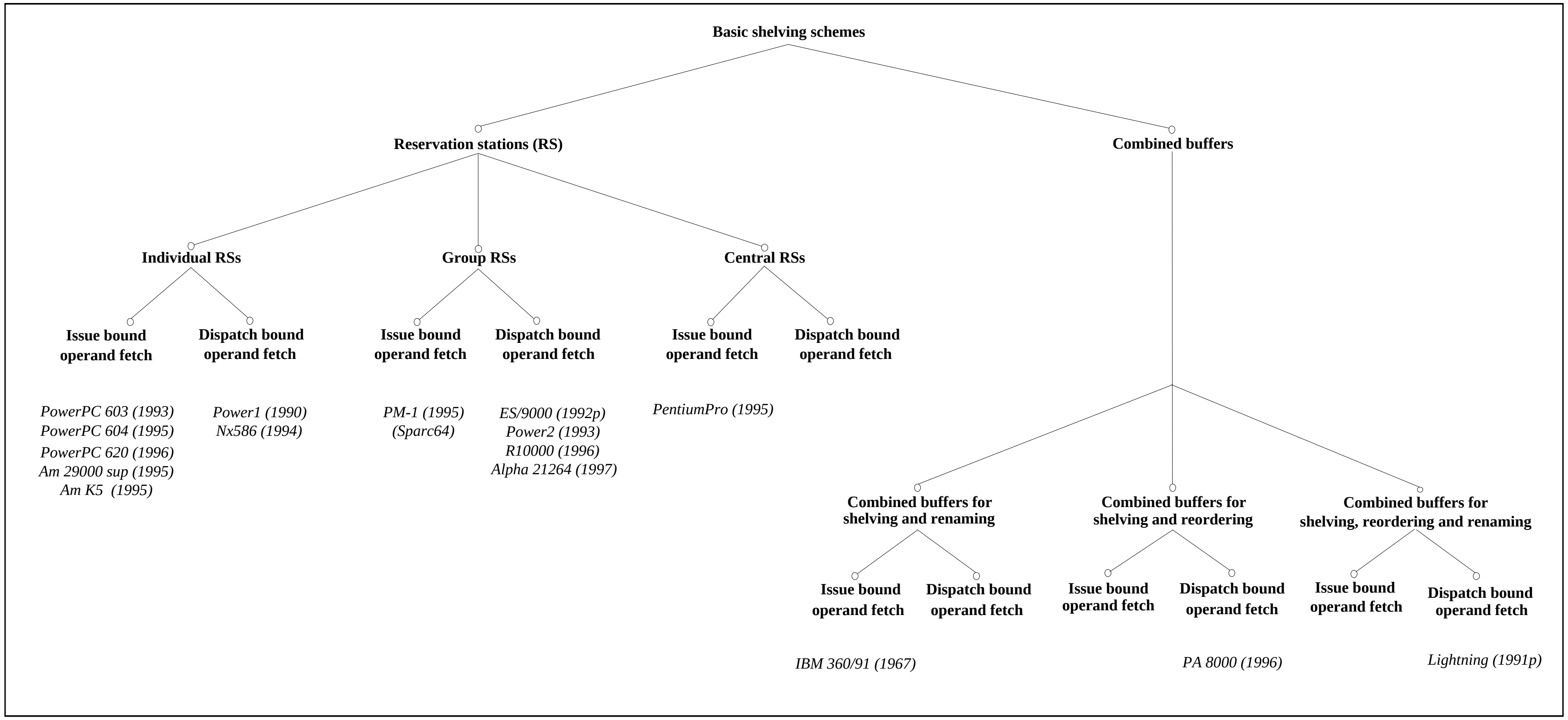


* PPC designates PowerPC.

** 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 16: Chronology of introduction of shelving in commercial superscalar processors



Basic shelving schemes
Reservation stations (RS)
Combined buffers
Individual RSs
Group RSs
Central RSs
Issue bound operand fetch
Dispatch bound operand fetch
Issue bound operand fetch
Dispatch bound operand fetch
Issue bound operand fetch
Dispatch bound operand fetch
PowerPC 603 (1993)
PowerPC 604 (1995)
PowerPC 620 (1996)
Am 29000 sup (1995)
Am K5 (1995)
Power1 (1990)
Nx586 (1994)
PM-1 (1995)
(Sparc64)
ES/9000 (1992p)
Power2 (1993)
R10000 (1996)
Alpha 21264 (1997)
PentiumPro (1995)
Combined buffers for shelving and renaming
Combined buffers for shelving and reordering
Combined buffers for shelving, reordering and renaming
Issue bound operand fetch
Dispatch bound operand fetch
Issue bound operand fetch
Dispatch bound operand fetch
Issue bound operand fetch
Dispatch bound operand fetch
IBM 360/91 (1967)
PA 8000 (1996)
Lightning (1991p)

Figure 17: Basic shelving schemes and choices made in recent superscalars

At first glance no clear trend is visible concerning the design choices made in recent superscalar processors. Nevertheless, for reasons discussed previously we expect future processors to employ predominantly either group reservation stations or combined buffers used at the same time for shelving and for reordering and possible for renaming as well. Furthermore, we expect the dispath bound operand fetch policy to become predominant over the issue bound fetch policy.

5.3 The efficiency of microarchitectures

Over the years the enhancements introduced including shelving, raised the *efficiency of superscalar microarchitectures.* In Fig. 18 we indicate this by showing the *relative integer performance* of processors. We calculated the given performance figures by subdividing the SPECInt95 performance values of a particular processor by its clock frequency, given as a multiple of 100 MHz. In this way we are able to compare the performance of microarchitectures as if they were operating at the same clock frequency. As Fig. 18 shows, superscalar processors without shelving (and renaming) remain basically in the 1.5 to 3 relative performance region. It is clear that partial shelving contributes to medium range performance figures whereas full shelving and further enhancements are a prerequisite to achieve a higher relative performance.

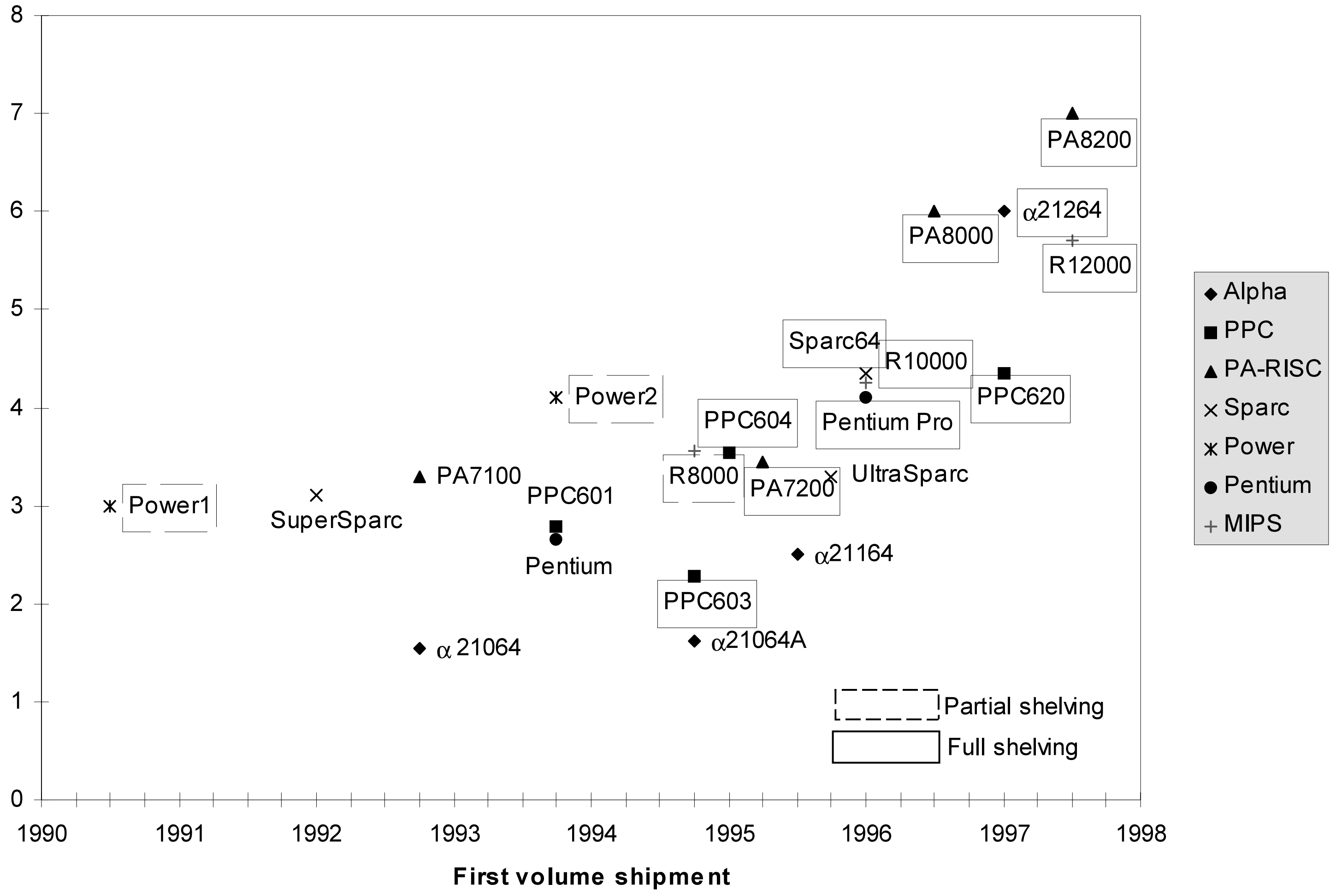


Figure 18: Efficiency of microarchitectures

(In the figure PowerPC is abbreviated to PPC)

6. Conclusions

Shelving has been recognized as an inevitable concept for increasing the performance of superscalar processors. In this paper, we identified the design space of shelving, which is spanned by the following four dimensions: the scope of shelving, the layout of the shelving buffers, the operand fetch policy used and the instruction dispatch scheme. In the first three dimensions feasible implementation alternatives have been outlined and assessed. It has been shown that full shelving, both group reservation stations and combined shelving buffers as well as the dispatch bound operand fetch policy are the most favorable design options. Furthermore, through appropriate reduction of the design space the basic shelving schemes could be identified.

Acknowledgement

A number of figures appear from Chapter 7 of [26], written by the same author courtesy of the Publisher.

References

[1] J.E. Thornton, *Design of a computer: The Control Data 6600* (Scott, Foresmann & Company, Glenview, 1970).

[2] R.M. Tomasulo, An efficient algorithm for exploiting multiple arithmetic units, *IBM J. Res. and Dev.* 11(1) (1967) 25-33.

[3] G.F. Grohoski, Machine organization of the IBM RISC System/6000 processor, *IBM J. Res. Development* 34(1) (1990) 37-58.

[4] V. Popescu, M. Schultz, J. Spracklen, G. Gibson, B. Lightner and D. Isaman, The Metaflow architecture, *IEEE Micro* 7(June) (1991) 10-13, 63-71.

[5] T. N. Hicks, R.E. Frey and P.E. Harvey, POWER2 floating-point unit: architecture and implementation, In *PowerPC and Power2: Technical aspects of the new IMB RISC System/6000* (IBM Corp. 1994) 29-44.

[6] J.S. Liptay, Design of the IBM Enterprise Sytem/9000 high-end processor, *IBM J. Res. Develop.* 36(4) (1992) 713-731.

[7] B. Burgess, et al., The PowerPC 603 Microprocessor, *Comm. ACM* 37 34-42.

[8] S.P. Song, et al., The PowerPC 604 RISC Microprocessor, *IEEE Micro* 12(Oct.) (1994) 8-17.

[9] D. Levitan, et al., The PowerPC 620 microprocessor: a high performance superscalar RISC microprocessor, *Proc. COMPCOM (*1995) 285-291.

[10] K. Diefendorff and M. Allen, Organization of the Motorola 88110 superscalar RISC microprocessor, IEEE *Micro* 10(April) (1992) 40-62.

[11] L. Gwennap, NexGen enters market with 66-MHz Nx586, *Microprocessor Report* 8(4) (1994) 12-17.

[12] M. Slater, AMD’s K5 designed to outrun Pentium,*Microprocessor Report* 8(14) (1994) 1-11.

[13] B. Case, AMD unveils first superscalar 29K core, *Microprocessor Report* 8(14) (1994) 23-26.

[14] R.P. Colwell and R.L. Steck, A 0.6 µm BiCMOS processor with dynamic execution, Intel Corp. (1995).

[15] L. Gwennap, Intel’s P6 uses decoupled superscalar design, *Microprocessor Report* 9(2) (1995) 9-15.

[16] P. Y-T. Hsu, Designing the FPT microprocessor, *IEEE Micro* 12(April) (1994) 23-33.

[17] N. Patkar, et al., Microarchitecture of HaL's CPU, *Proc. COMPCON* (1995) 259-266.

[18] G. Kurpanek, et al., PA-7200: A PA-RISC processor with integrated high preformance MP bus interface, *Proc. COMPCON* (1994) 375-382.

[19] L. Gwennap, PA-8000 combines complexity and speed, *Microprocessor Report* 8(15) (1994) 1-8.

[20] A. Kummar, The HP PA-8000 RISC CPU, *IEEE Micro* 17(March,April) (1997) 27-32.

[21] J. Heinrich, *MIPS R10000 Microprocessor User’s Manual, version 2.0,* (MIPS Technologies Inc. Mountain View, CA, Sept. 1996).

[22] L. Gwennap, Digital 21264 sets new standard, *Microprocessors Report* 10(14) (1996) 11-16.

[23] D. P. Bhandarkar, *Alpha implementations and architecture*, (Digital Press, Newton MA, 1996)

[24] D. Tabak, *RISC systems and applications*, (RSP, UK and Wiley, NY, 1996)

[25] D. Sima, *A novel approach for the description of computer architectures*, D.Sc thesis, Hungarian Academy of Sciences, Budapest, 1993

[26] D. Sima, T. Fountain and P. Kacsuk, *Advanced Computer Architectures*, (Addison Wesley Longman, Harlow etc., 1997).

[27] N.P. Jouppi, D. W. Wall, Available instruction-level parallelism for superscalar and superpipelined machines, *In Proc. ASPLOS III*, 1989, 272-282

[28] D. W. Wall, Limits of instruction level parallelism, *In Proc ASPLOS IV*, 1991, 176-188

[29] D. Sima, The design space of superscalar instruction issue, *IEEE Micro* 17 (September, October) (1997), 29-39