3.4 Shelving

Recall that with a straightforward issue policy dependencies block instruction issue, as indicated in Figure 3.4.1.

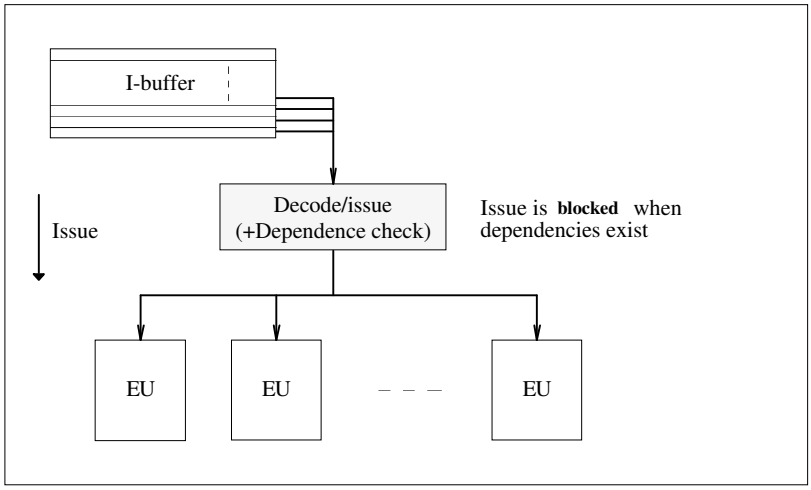

Figure 3.4.1: Principle of a straightforward issue policy

In Section 3.3.1 we introduced the concept of shelving. This is an advanced issue mode, which is employed to eliminate issue blockages due to dependencies. Shelving makes use of dedicated instruction buffers, called **shelving buffers**, in front of each EU. In Figure 3.4.2 and in the subsequent description of the operation, we assume that individual shelving buffers are associated with each EU. As we shall detail in Section 3.4.3, other possibilities for the layout of shelving buffers also exist. Nevertheless, this does not affect the principle of how shelving is carried out. Therefore, in the following we describe shelving, assuming individual reservation stations as seen in Figure 3.4.2.

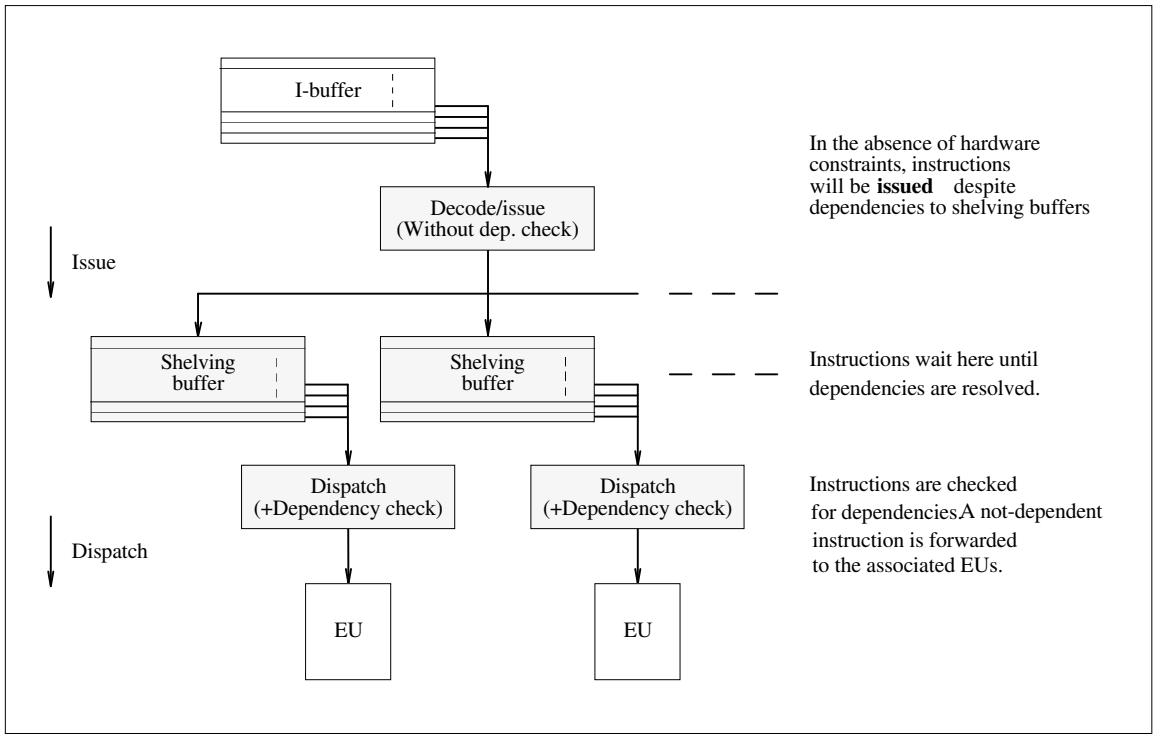

Figure 3.4.2: The principle of shelving

Practically, shelving decouples dependency checking from the instruction issue, and defers it to the dispatch phase. More precisely, with shelving decoded instructions are issued to the shelving buffers *without any checks for data- or control dependencies* or *for busy EUs*.

Nevertheless, certain *resource constraints*, can restrain the processor to issue fewer instructions in a cycle than its issue rate is. There are two typical resource constraints. On the one hand, *lack of free entries,* for instance, in the targeted reservation station, in the rename register file or in the ROB can prevent instruction issue. On the other hand, *data path restrictions* can also limit the number of instructions which can be transferred in one cycle to a particular reservation station. This point is discussed in more detail in Section 3.4.3. In the case of an *issue blockage*, processors with shelving usually employ *in order, aligned issue,* as stated before in Section 3.3.2.

In the shelving buffers, instructions are held until dependencies are resolved and instructions can be forwarded to the EUs. When an EU becomes free, the associated shelving buffers are checked for instructions eligible for execution. An instruction not bound to any dependency is eligible for execution.From the eligible instructions one is selected for execution according to the dispatch policy used, and will be forwarded to the connected EU.

As pointed out in Section 3.3.3 processors which use shelving predominantly employ the advanced superscalar issue policy. This means that shelving is employed usually, *in connection with speculative execution of unresolved conditional branches and register*

renaming. In this case, speculative branch processing eliminates execution blockages due to control dependencies whereas renaming removes those due to false register data dependencies, i.e. due to WAR and WAW dependencies. Then, only true data dependencies can prevent instructions from execution. Consequently, during instruction dispatch, dependency checking is reduced to checks for true data dependencies (RAW dependencies). This means that an instruction held in a shelving buffer becomes eligible for execution when all its operands are available. This is exactly the *data flow principle of operation*. Therefore, a superscalar processor with shelving, speculative branch processing, and register renaming executes instructions on the dataflow basis.

A last remark concerns the preservation of *sequential consistency.* This means that the logical integrity of the execution has to be sustained despite parallel execution of instructions. But, in a sense, shelving makes processing more distributed, since with shelving the number of instructions which are in processing is increased by the number of instructions held in shelving buffers. Therefore, shelving is used typically *in connection with* an advanced method of retaining sequential consistency called the *instruction reordering*, which is explained in Section 3.7.3.

Shelving is quite a complex topic. Recent superscalar processors implement shelving in a blurring variety. This section gives a framework for overviewing and understanding this challenging diversity.

3.4.1 The design space of shelving

Shelving has a fairly complex design space, which is depicted in Figure 3.4.3. It consists of the following four main components; the scope of shelving (Section 3.4.2), the layout of the shelving buffers used (Section 3.4.3), the operand fetch policy (Section 3.4.4) and the instruction dispatch scheme (Section 3.4.5). The *scope of shelving* declares whether shelving covers all data types or is restricted to a few of them. The *layout of the shelving buffer* component of the design space specifies the infrastructural background of instruction buffering after issue. The *operand fetch policy* 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. In the following sections we will discuss each of the main components mentioned.

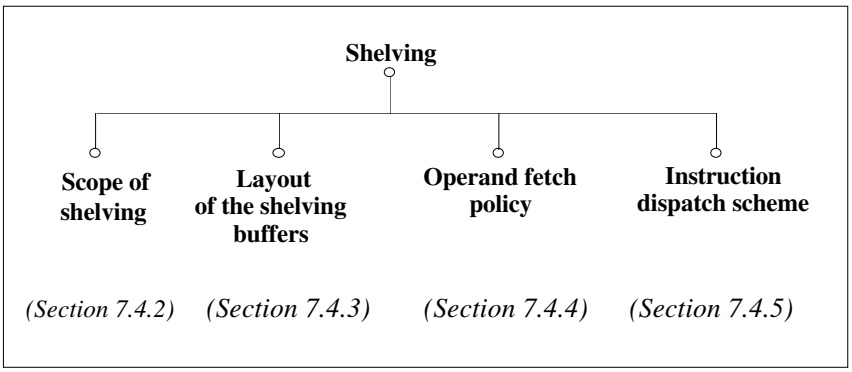

Figure 3.4.3: Design space of shelving

3.4.2 Scope of shelving

The **scope of shelving** specifies whether shelving is restricted to a few instruction types or is performed for all instructions, as indicated in Figure 3.4.4.

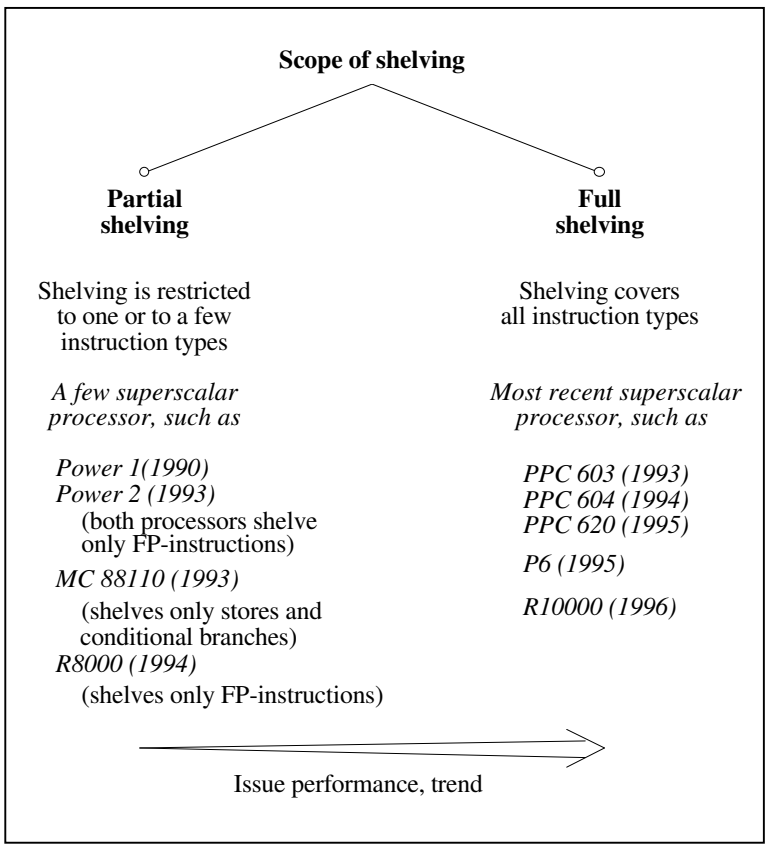

Figure 3.4.4: Scope of shelving

Partial shelving is only used in a few superscalar processors, such as the Power1, Power2, MC 88110 and the R8000. As pointed out in the figure, in three processors mentioned, shelving is restricted to FP-instructions, whereas the MC 88110 shelves stores and conditional branches.

Partial shelving is evidently an incomplete solution to eliminate issue blockages due to dependencies. Therefore, most recent superscalar processors employ **full shelving**. Examples are the PowerPC 603 - PowerPC 620, PentiumPro and the R10000.

3.4.3 Layout of the shelving buffers

Shelving buffers hold instructions from their issue until forwarding them to an EU. Shelving buffers are laid out by specifying three major aspects; their type, their capacity and the number of their read- and write ports, as depicted in Figure 3.4.5. These aspects shall be detailed in the following.

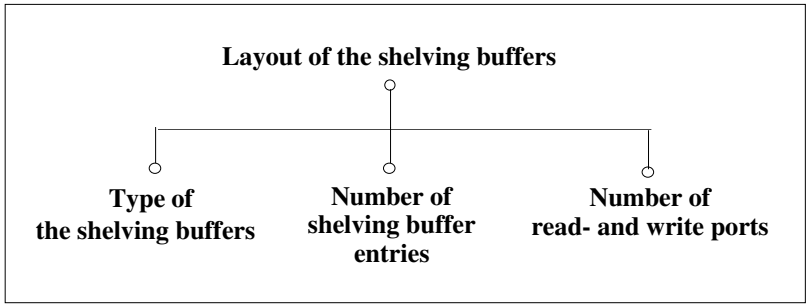

Figure 3.4.5: Layout of the shelving buffers

Types of shelving buffers

We distinguish between two basic types of shelving buffers, designated as *stand-alone shelving buffers* and *combined buffers* which serve at the same time for shelving, renaming and reordering, as shown in Figure 3.4.6.

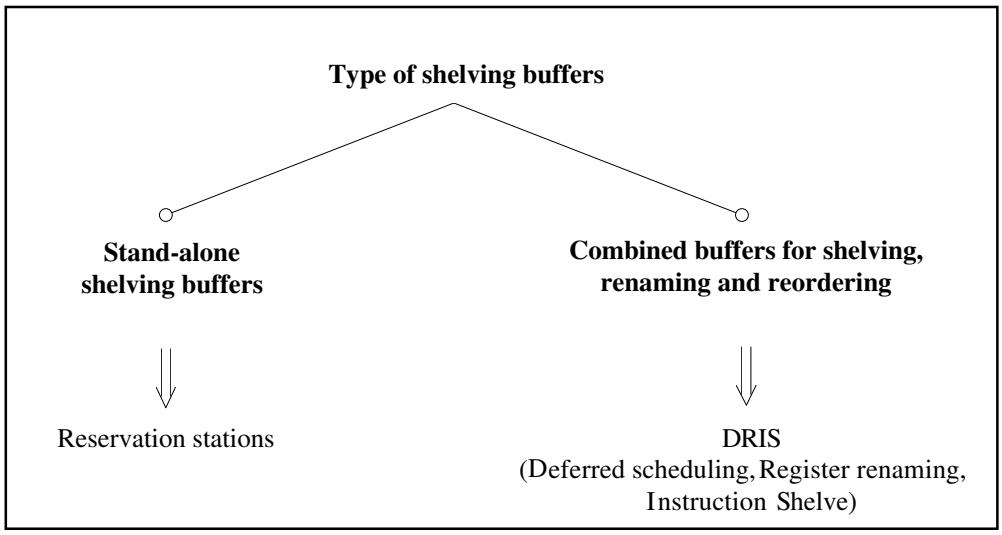

Figure 3.4.6: General layout of the implementation of shelving buffers

In most cases shelving buffers are implemented as *stand-alone shelving buffers*, that is buffers which are used exclusively for shelving. This type of shelving buffer is usually designated as a **reservation station**. In superscalar processors reservation stations are implemented using one of three basic schemes, as indicated in Figure 3.4.7.

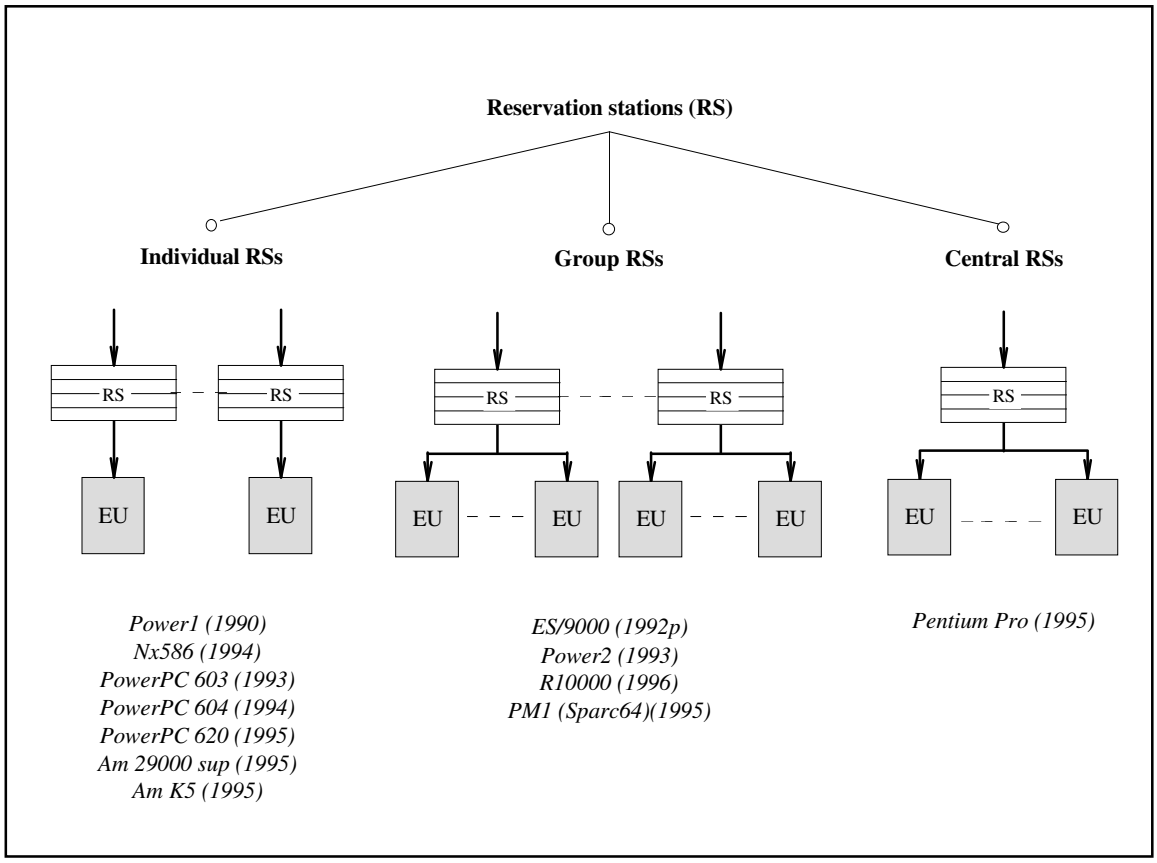

Figure 3.4.7: Basic variants of shelving buffers

In the simplest case, **individual reservation stations** are used in front of each EU. Instructions which are issued for execution in a particular EU are transferred into the individual reservation station of that EU. Usually, individual reservation stations provide space to hold only a small number of instructions, say 2 to 4. For instance, in the PowerPC 620 the reservation stations in front of the three integer units and 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. (See Figure 3.10.3 for details of the microarchitecture of the PowerPC 620.)

Both 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 and the FP-Multiply unit with a three-entry station. Further examples of superscalar processors using individual reservation stations are the

Power1 and high-performance models of the PowerPC line (PowerPC 603 - PowerPC 620).

In an alternative approach, reservation stations are implemented as **group stations** (see Figure 3.4.7). 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. As illustrated in Figure 3.10.1, one of these serves two FX-ALUs, another one a single address unit, and the third group station four FP-units. In addition to the R10000 we can find group reservation stations in the ES/9000, Power2 and in the PM1 (Sparc 64) as well.

Evidently, group stations must have more available buffer space than individual stations. For instance, the R10000 has three group stations with 16 positions each; the PM1 has four group stations with 8 or 16 entries each. Furthermore, since group stations serve more than one EU, they should be capable of shelving or dispatching more than one instruction in each cycle to avoid performance bottlenecks. As an example, each group station of the R10000 can accept 4 instructions in each cycle. Two of these stations are able to dispatch two instructions whereas the third one only one instruction per cycle.

Group stations have an edge over individual stations since they are *more flexible* in dispatching instructions to EUs. Thus, well designed group stations are better utilised than individual stations, assuming the same number of locations provided in both cases.

The last alternative is when a single reservation station serves all ALUs. This is called a **central reservation station**. Evidently, a central station should have even more capacity then group stations have. Furthermore, it should be able to accept or deliver even more instructions per cycle then group stations do. A central station has however, from the point of implementation disadvantages as well. First, a central station must have a word length which is 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 do, central stations are more expensive to implement than group stations. Nevertheless, the PentiumPro opted for a central station. In the PentiumPro a 20 entry central reservation station serves all available EUs (10 altogether). For more details consult Figure 3.10.6.

A quite different approach to the implementation of shelving buffers is to use the same buffer *for shelving, renaming and reordering* as shown in Figure 3.4.8. Then we are concerned with a **combined buffer**.

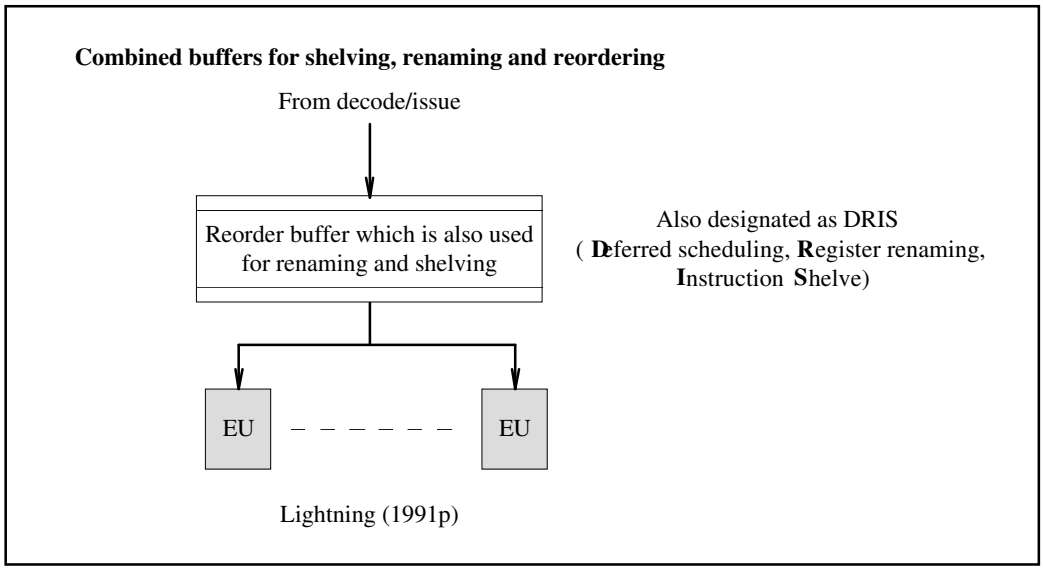

Figure 3.4.8: Using a combined buffer for shelving, renaming and reordering

In this case, the *reorder buffer* (ROB), which assures the logical integrity of the program execution, is extended such that it is used for shelving and register renaming as well. (See Section 3.7.3 for a discussion of the reorder buffer.) An example of the combined implementation is the Metaflow Lightning processor (1991p). This processor 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 efficiency of this approach makes it attractive. In the future, we expect further processors to use this design option.

Finally, a comment on our terminology. When we use the term *shelving buffer* we refer to all possible implementations, including reservation stations or DRIS. By contrast, with the terms *reservation stations* or *DRIS* we refer to the particular type of implementation only.

Number of shelving buffer entries

As a rule of thumb, individual, group and central reservation stations each have to provide an increasing number of shelving places. Typically, individual reservation stations shelve 2 - 4 instructions, group stations have 6, 7, 12 or 16 entries, whereas the only implementation so far of a central reservation station (the PentiumPro) has room to hold 20 instructions. The total number of entries in the reservation stations provided restricts the number of the instructions waiting in shelving buffers to be dispatched. Recent processors have typically between 15 and 40 shelves, as Table 3.4.1 indicates.

Processor	Total number of shelves
PowerPC 603 (1993)	3
PowerPC 604 (1994)	12
PowerPC 620 (1995)	15
Nx586 (1994)	42
K5 (1995)	14
PM1 (Sparc64) (1995)	36
PentiumPro (1995)	20
R10000 (1996)	48

Table 3.4.1: Comparison of available shelves in recent superscalar processors

Number of read- and write-ports

The last component of the layout of shelving buffers is the **number of read- and write-ports**. It specifies how many instructions may be written into or read out from a particular shelving buffer in a cycle. First, let us consider the expected number of read ports (output ports). Clearly, individual reservation stations only have to forward a single instruction per cycle. A group- or central reservation station, on the other hand, should deliver multiple instructions per cycle, ideally, as many as EUs are connected to it. Thus, individual reservation stations have a single read port, each group station is expected to provide multiple read ports, and central reservation stations should have even more read ports than group stations have.

As far write ports (input ports), individual, group and central reservation stations require, increasingly more write ports. So processors with individual reservation stations often allow only one instruction per cycle to be issued into any one reservation station. Examples are the PowerPC 604 or the Nx 586. In contrast, recent superscalar processors with group reservation stations permit the transfer of 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 up to 2 FP-, 2 load/store-, 1 branch and 4 integer instructions into the corresponding group reservation stations. Here we mention that from the integer instructions not more than two instructions 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.

3.4.4 Operand fetch policies

Basic fetch policies

Closely connected with shelving is the policy according to processors fetch operands. Basically, operands are fetched either *issue bound* or *dispatch bound*, as indicated in Figure 3.4.9.

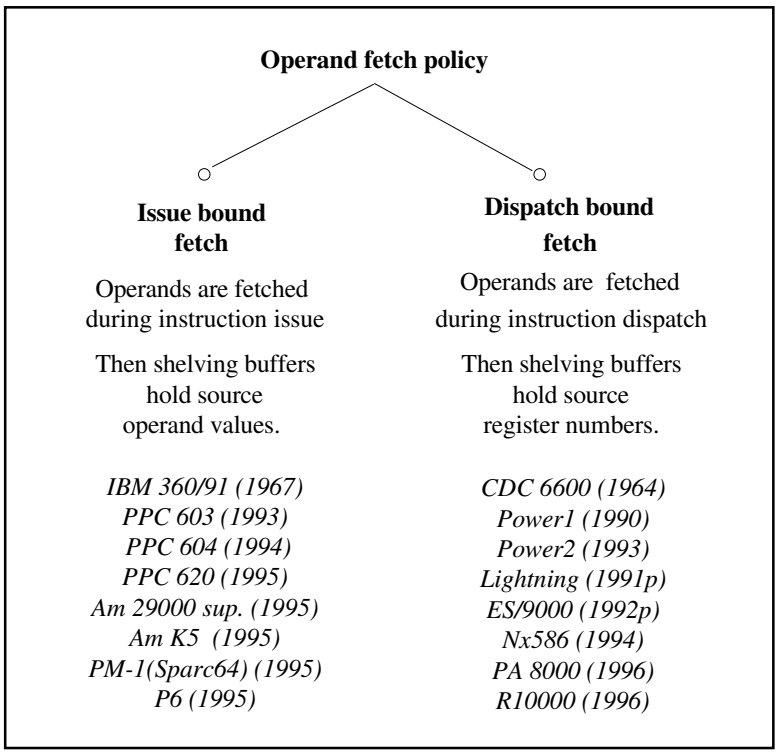

Figure 3.4.9: Operand fetch policies

The **issue bound fetch policy** means that operands are fetched during instruction issue. In this case shelving buffers have to provide entries long enough to hold source operands. The other basic operand fetch policy is the **dispatch bound fetch** *policy*. According to this policy, operands are fetched during dispatching. In this case shelving buffers contain much shorter register identifiers instead of long operands.

In the following, we describe both operand fetch policies more detailed. In our discussion we assume individual reservation stations. This assumption does not affect the principles discussed here and allows to focus better on the vital points. Furthermore, first we describe both policies assuming the simplest case, i.e. when a single register file is employed for both FX-and FP-data and no renaming is used. Subsequently, we shall extend our discussion for the case when split register files or renaming is used.

Figure 3.4.10 shows the principle of the *issue bound operand fetch policy*. During instruction issue, the source register numbers of the issued instructions are forwarded to the register file for fetching the referenced operands. In addition, the operation codes, the

destination register numbers of the issued instructions, as well as the fetched operand values (Source 1 and Source 2 operands in the Figure) are written into the allocated reservation stations.

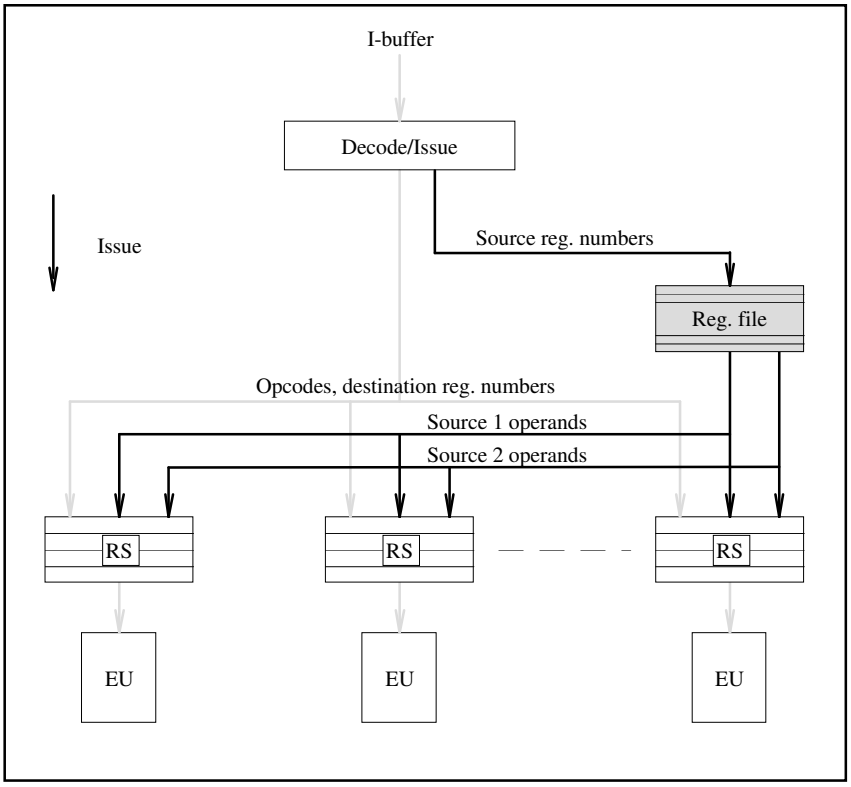

Figure 3.4.10: Operand fetch during instruction issue, assuming a single register file for both FX- and FP-data

The other policy is the *dispatch bound* operand fetch policy. In this case *operands* are fetched *in connection with instruction dispatch* as Figure 3.4.11 shows.

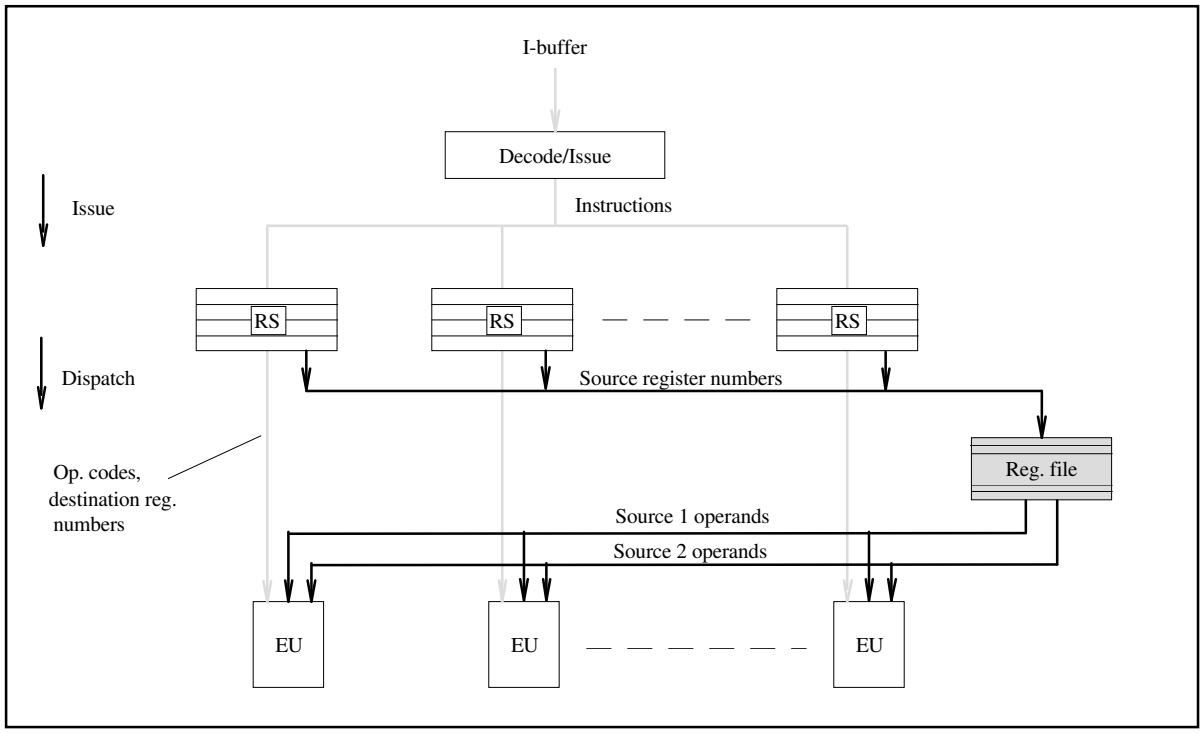

Figure 3.4.11: Operand fetch during instruction dispatch assuming a single register file for both FX- and FP-data

In this case, the reservation stations hold the instructions including the *source register numbers*. During dispatch, the operation codes and destination register numbers of the dispatched instructions are forwarded from the reservation stations to the associated EUs. Furthermore, the source register numbers are passed to the register file. Subsequently, after a delay, the fetched source operand values appear at the inputs of the EUs and are gated into the requesting EUs.

Now, let us *compare* the operand fetch policies described. If operands are fetched during instruction issue, the register file must be able to supply operands for all issued instructions at the same time. Assuming two referenced operands per instruction, this requires twice as many read ports in the register file as the maximum issue rate. For instance, under the assumptions made, a four-issue superscalar processor requires a register file with 8 read ports.

If operands are fetched during instruction dispatch, ideally as many read ports must be available as the dispatch rate. Since the maximum dispatch rate is usually higher than the maximum issue rate (see Section 3.4.5 and Table 3.4.2), the dispatch bound operand fetch policy usually requires more read ports on the register file than the issue bound policy. The same relationship is also true for the number of the required operand buses. This fact is a drawback of the dispatch bound operand fetch policy.

On the other hand, the dispatch bound operand fetch policy has two advantages over the issue bound policy. First, in this case, the critical decode/issue path is shorter. Second,

shelving buffers are less complex, since register numbers require considerably less buffer space than operand values.

As the benefits and drawbacks of the basic fetch policies are roughly in balance, there is no clear trend in their use at the moment. A number of processors use the issue bound operand fetch policy, such as the early IBM 360/91, the PowerPC 603, PowerPC 620 etc. A similar number of processors employ the dispatch bound policy. Examples are the early CDC 6600, the Power1, Power2, R10000 etc. Figure 3.4.9 shows further examples for both fetching policies.

Operand fetch assuming split register files for FX- and FP-data

Our simplified introduction of operand fetch policies in the preceding section assumed a common register file for FX- and FP-data. However, in practice, most processors use *distinct register files* for *FX- and FP-data*. Figure 3.4.12 revisits the *issue bound operand fetch policy assuming distinct register files for FX- and FP-data.*

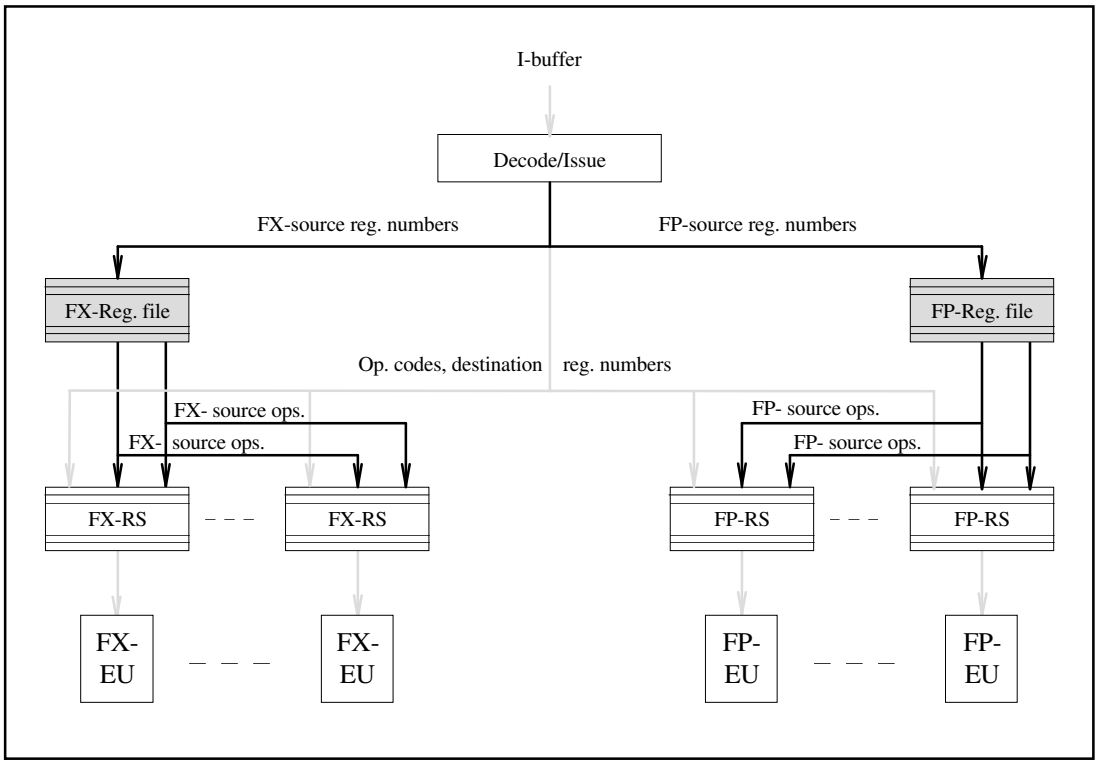

Figure 3.4.12: Issue bound operand fetch assuming distinct register files for FX- and FP-data

In fact, the microarchitecture now 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 by two distinct and more or less symmetrical parts of the processor.

As far as the other fetching policy is concerned the situation is much the same. Figure 3.4.13 shows the principle of the dispatch bound operand fetch policy assuming split register files.

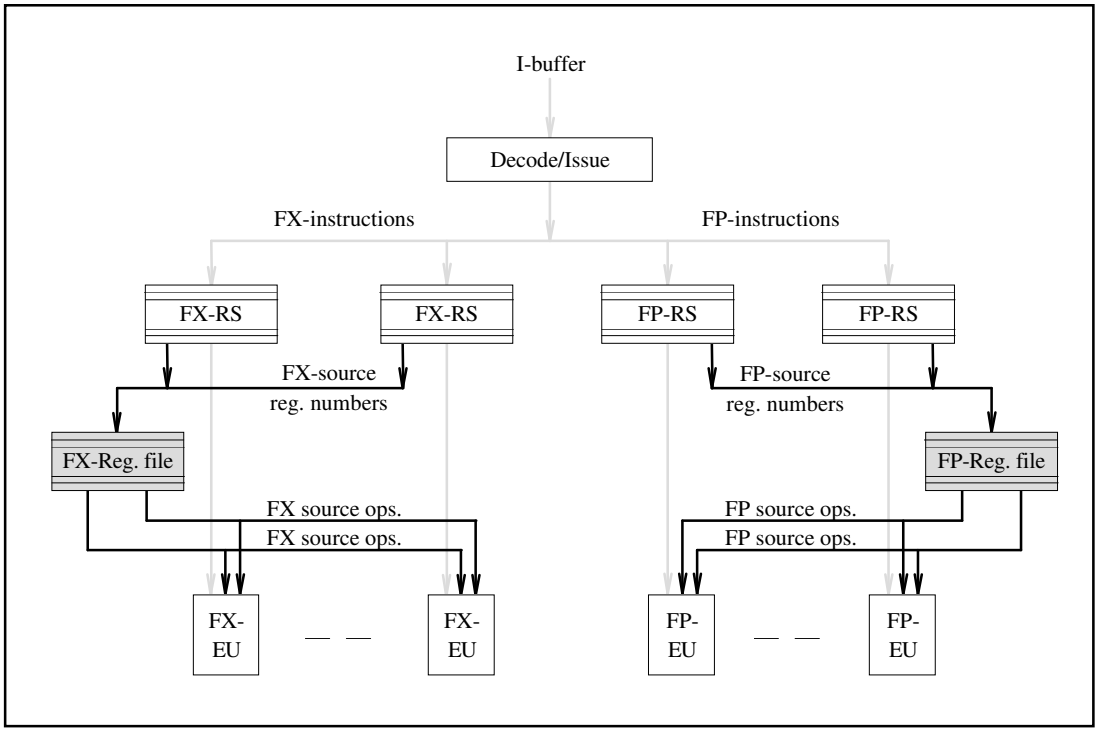

Figure 3.4.13: Dispatch bound operand fetch assuming distinct register files for FX- and FP-data

Again we see a symmetrical internal structure with distinct FX- and FP-parts. In this case, the FX- and FP-register files supply the operands directly to the EUs rather than to the reservation stations.

Actually, most recent architectures declare *separate register files* for FX- and FP-data, like the PowerPC, the x86, the R and the PA architectures. Accordingly, superscalar implmentations of these architectures which make use of shelving, such as the PowerPC line of processors, the PentiumPro, R10000 and the PA-8000 fetch operands in principle as indicated in Figures 3.4.13 and 3.4.14. We note that some embedded processors, like the Am 29000 superscalar, support only FX-data, and consequently have only a single FX-register file. Other processors, such as the Nx586, implement only the FX-part of the architecture in the main processor, while the FP-part is realised by an FP-coprocessor chip, called Nx587.

Operand fetch using renaming

Without renaming all required operands are supplied by the architectural register file or files. When register renaming is used we encounter a quite different picture since additional register space is needed for storing renamed values. Then, required *operands* may be *either in the architectured register file(s)* or *in the register space used for renaming*. In this case as well, the same two basic operand fetch policies are used, i.e. either the issue bound or the dispatch bound operand fetch policy. (See Section 3.5.1 for a discussion of further issues related to register renaming.)

Most frequently used combinations of shelving buffer types and operand fetch policies

In conclusion, we will give an overview of the most frequently used combinations of shelving buffer types and operand fetch policies. All possible combinations are depicted in Figure 3.4.14 along with the choices made in recent processors.

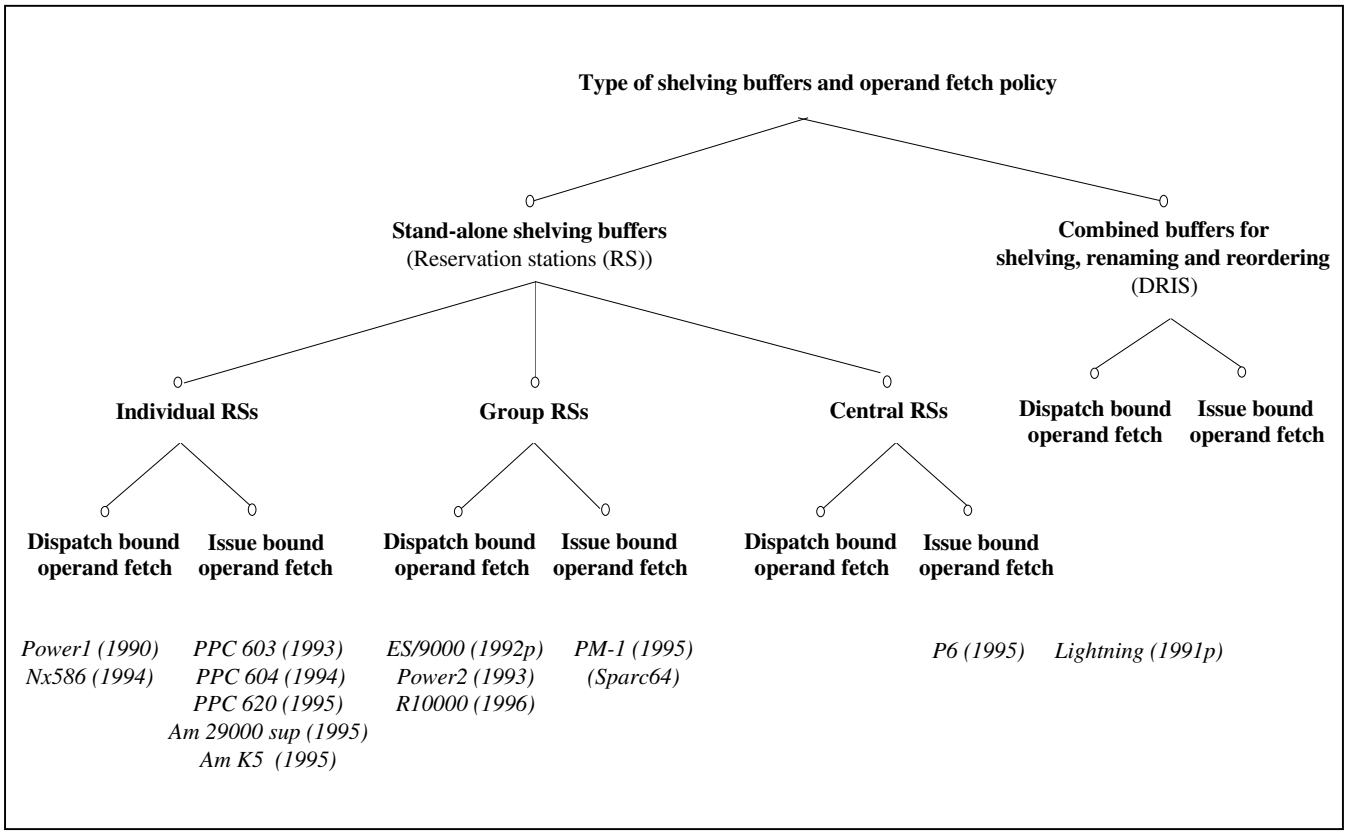

Figure 3.4.14: Most frequently used combinations of shelving buffer types and operand fetch policies

As shown in the figure *individual reservation stations and dispatch bound operand fetch* is used in the Power1 and in the Nx586. By contrast, *individual stations with issue bound operand fetch* are incorporated into the PowerPC processors and the Am superscalar processors (K5, Am 29000 sup.). *Group reservation stations* are paired *with dispatch bound operand fetch* in the ES/9000, Power2 and R10000. The PM1 (Sparc64) employs *group stations while fetching operands already during instruction issue*. So far the single implementation of a *central reservation station* is the PentiumPro, where again *operands are fetched during issue*. Finally, *combined buffers for renaming, shelving and reordering* by means of a DRIS were chosen only once by the Lightning processor (1991p), where operands are fetched during dispatch. At the moment, no clear trend is visible in the used combinations of shelving buffers and operand fetch policies.

3.4.5 Instruction dispatch schemes

Instruction dispatch can be broken down into two basic tasks: *scheduling the instructions* held in a particular reservation station *for execution*, and *disseminating the scheduled instruction* or *instructions* for the allocated EU(s). These basic tasks involve a number of subtasks which are detailed in the following sections.

The design space

The design space of instruction dispatch is quite complex. It resembles in some aspects, that of the instruction issue but it also comprises two additional aspects, as depicted in Figure 3.4.15.

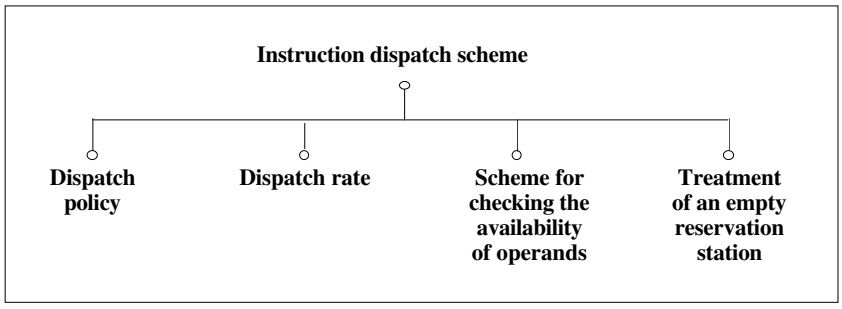

Figure 3.4.15: Design space of instruction dispatch

The *dispatch* policy specifies how instructions are selected for execution and how dispatch blockages are handled. A further aspect of the design space details how many instructions can be dispatched from the shelving buffers in one cycle, termed as *dispatch rate*. The method for checking *the availability of operands* must also be specified. And finally, we must decide whether an *empty reservation station* may be *bypassed* and the

instructions may be forwarded without additional delay for execution or not. The following sections discuss these aspects in more detail.

Dispatch policy

As mentioned above, the dispatch policy decides how instructions are selected for execution and how dispatch blockages are handled. From one point of view, the dispatch policy may also be considered a scheduling policy consisting of the components specified in Figure 3.4.16.

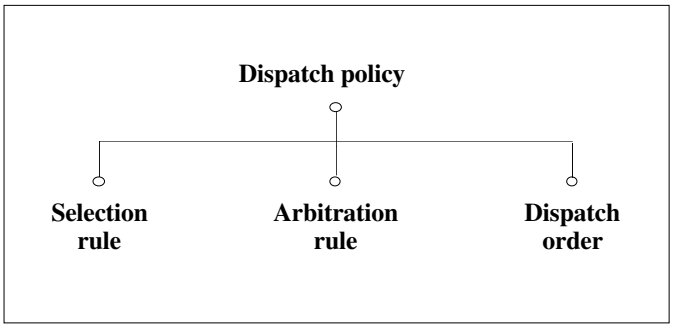

Figure 3.4.16: Design-space of instruction dispatch

Selection rule

Basically, the *selection rule* specifies when instructions are considered executable. Let us take it for granted that renaming is employed and unresolved conditional transfer instructions are met by speculative branch processing. Then assuming the availability of the needed EU, those instructions are executable whose operands are available. In this case the selection rule is the same as the data *flow principle of operation* introduced in Section 1.1.2. In other cases, the selection rule must include further necessary dependency checks as well.

Arbitration rule

In addition to the selection rule, we also need an *arbitration rule* for the case when more instructions are eligible for execution than can be disseminated in the next cycle. Most processors use a straightforward rule of arbitration that prefers "older" instructions over "younger" ones. The Power1, Power2, PowerPC 620, PM1 (Sparc64) and the PentiumPro are some examples that use this rule for arbitration.

Dispatch order

And finally, the dispatch order determines whether a non-executable instruction prevents all subsequent instructions from being dispatched or not, as summarised in Figure 3.4.17.

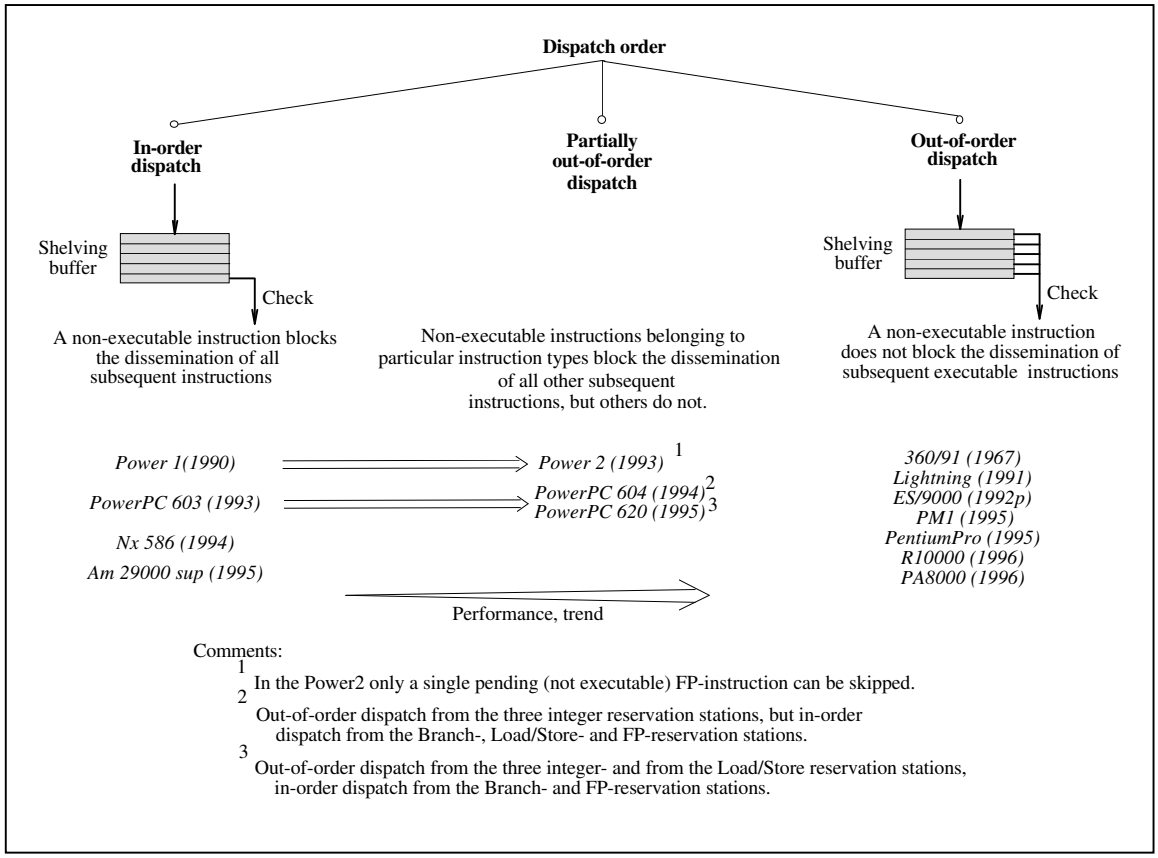

Figure 3.4.17: Dispatch order

When an **in-order dispatch scheme** is employed, a non-executable instruction blocks any further dispatch. Thus, even if subsequent instructions were available for execution, they could not be dispatched in this scheme. Obviously, in-order dispatching impedes performance. Processors using in-order dispatch are for instance the Power1 and the PowerPC 603.

A slight performance increase may be achieved using a **partially out-of-order dispatch**. The Power2, PowerPC 604 and PowerPC 620 are processors using this approach. The Power2 shelves only FP-instructions, and allows merely a single non-executable FP-instruction to be skipped by an executable FP-instruction during dispatching. The PowerPC 604 and PowerPC 620 dispatch instructions from particular reservation stations in-order and from others out-of-order. Thus, the PowerPC 604 performs in-order dispatch from the Branch-, Load/store- and FP-reservation stations, whereas it dispatches instructions out-of-order from the three FX-reservation stations (see also Figure 3.4.24).

The dispatch scheme of the PowerPC 620 is slightly improved over that of the PowerPC 604, since the Load/Store reservation station is converted to an out-of-order dispatch fashion (see 3.10.2). All other reservation stations that are not mentioned dispatch in the same manner as the PowerPC 604, as indicated in Figure 3.4.17.

The most advanced scheme is clearly the **out-of-order dispatch**. Here, a not executable instruction does not block the dissemination of subsequent executable instructions. Instead, any executable instructions held in the shelving buffer are eligible for dispatch. From the implementation point of view then all or at least a large number of shelving buffer entries have to be inspected for executable instructions. By contrast, in case of an in-order-dispatch scheme it suffices to check only the last buffer entry therefore. A very early forerunner of out-of-order dispatch was the processor IBM 360/91. Further examples followed more than two decades later, such as the Lighting and the ES/9000. The majority of the most recent processors, such as the PM1 (Sparc64), PentiumPro, R10000 and the PA 8000, employ out-of-order dispatch.

As we know, group and central reservation stations have to supply executable instructions for multiple EUs. If in these cases a not executable instruction would block the dispatch of subsequent executable instructions, performance could be impeded to a considerable degree. Therefore, we expect processors which incorporate group-reservation stations or a central station to use the out-of-order dispatch scheme.

Dispatch rate

Unlike individual reservation stations, a group- or central reservation station, or even a DRIS, must be capable of dispatching more than one instruction in each cycle. In these cases, the design space has to have an additional component that determines how many instructions can be dispatched from each of the reservation stations, or from the DRIS, per cycle. This component is called "**dispatch rate**" (Figure 3.4.18).

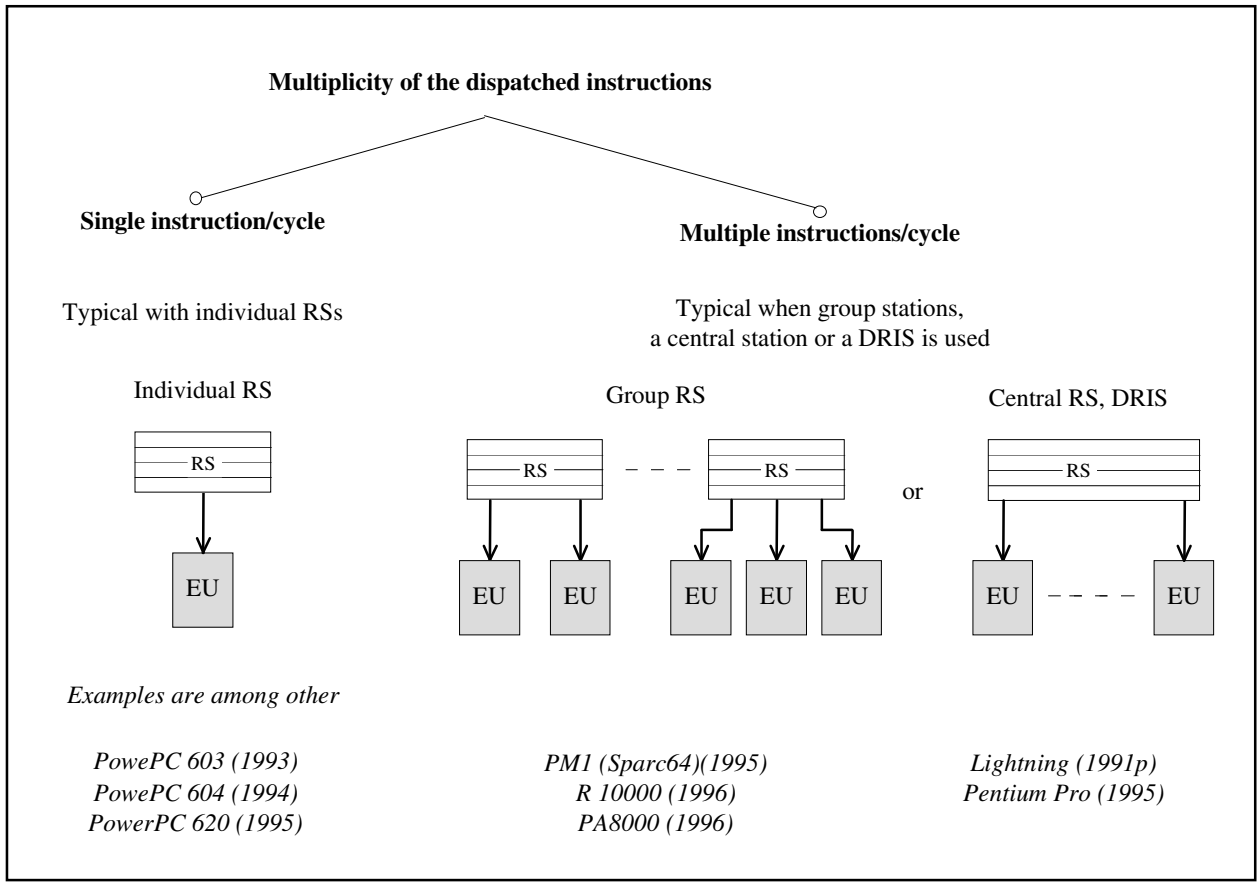

Figure 3.4.18: Multiplicity of dispatched instructions

Ideally, a shelving buffer must be capable of dispatching one instruction to any EU connected to it in each cycle. Obviously, this is easier to achieve for group stations with two to three EUs than for a central station or a DRIS with a considerable number of EUs connected to it. The R10000, for instance, employs *group reservation stations*. Its FX-reservation station can dispatch two instructions per cycle, one each to the EUs served. In contrast, its FP-reservation station serves four FP-EUs, but can only dispatch up to two instructions per cycle. One instruction can be forwarded to the FP-Adder, and one to either of the FP-Multiplier, the FP-Divider or the FP-Square root unit. These dispatch rate limitations are mainly due to data path or register port limitations aimed at reducing complexity. (See Figure 3.1.1.)

In the case of a *central reservation station*, or *DRIS*, higher dispatch rates are clearly required than for group reservation stations. For instance, the PentiumPro can dispatch 5 RISC instructions (called nop-s) per cycle, as indicated in Figure 3.10.6. Here, it is remarkable that port 0 is shared by 6 EUs. This is a design trade off. In the PentiumPro FP-data require 86 bits internally. Considering that each FP-unit needs at least two operands and delivers at least one result, by sharing one complex input/output port, a considerable savings can be achieved in the die area.

In conclusion, we note that in future processors with higher available transistor counts, we expect fewer hardware restrictions imposed on data path and port limitations, and thus higher dispatch rates than in recent ones.

For an overview, we have compiled the maximum issue and dispatch rates of superscalar processors with shelving, as shown in Table 3.4.2.

Processor/Year of volume shipment	Maximum issue rate instr./cycle	Maximum dispatch rate instr./cycle
PowerPC 603 (1993)	3	3
PowerPC 604 (1994)	4	6
PowerPC 620 (1995)	4	6
Power2 (1993)	4/6 [2]	10
PM1 (Sparc 64) (1995)	4	8
PA8000 (1996)	4	4
R10000 (1996)	4	5
Nx586 (1994)	3/4 [3,4]	3/4 [3,4]
K5 (1995)	4 [4]	5 [4]
Pentium Pro (1995)	4	5 [4]

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 3.4.2: Maximum issue and dispatch rates of superscalar processors with shelving

As Table 3.3.2 shows, in some cases both rates are the same, for instance in the processors PowerPC 603 and PA 8000. However, in most cases superscalar processors with shelving are capable of dispatching more instructions for execution than of issuing them for shelving. For example, the PowrPC 604 and PowerPC 620 issue up to 4 instructions but are able to start the execution of up to 6 operations in each cycle. As other examples Hal's PM1 (also termed as Sparc64,) and the Power2 issue up to 4 and 6 instructions respectively, whereas they are in a position to start the execution of up to 8 and 10 operations per cycle, respectively.

The motivation for setting *higher dispatch rates than issue rates* is the following. Shelving allows to issue the maximum number of instructions in most cycles. By contrast, complex instructions such as division, square root calculation, etc. cannot be started immediately one after the other. Thus, for a balance of instruction issue and dispatch the processor should be capable of dispatching more instructions to available EUs than of issuing to the shelving buffers per cycle.

Scheme for checking the availability of operands

The availability of operands has to be checked in two scenarios. When operands are fetched from the register file either during instruction issue or dispatch a scheme is needed to check whether requested operands are available in the register file or not. A similar scheme is needed during instruction dispatch to check whether all operands of the instructions held in the shelving buffers are available or not. In the discussion of possible solutions to this, we make use of a technique which is generally employed to indicate the availability of register operands, called scoreboarding.

The term **scoreboard** has been introduced in connection with the CDC 6600 (1964) to denote the complex circuitry used in this processor for controlling parallel operation. Recently, this term has been generalised and reinterpreted to designate a technique for checking the availability of register operands. Figure 3.4.19 demonstrates its principle.

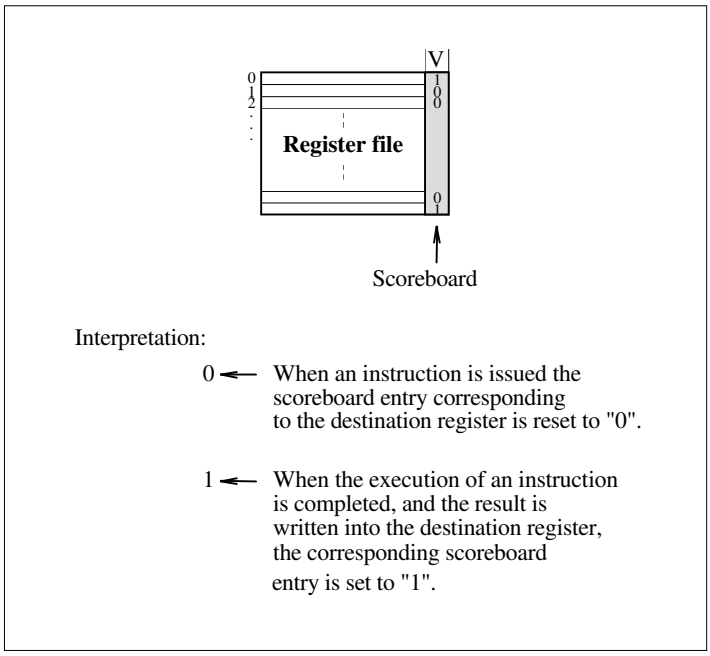

Figure 3.4.19: The principle of scoreboarding

The scoreboard is actually a status register consisting of one-bit entries. Each of the entries may be envisaged as a one-bit extension of the corresponding register. The scoreboard bits indicate whether data in the associated register is valid, that is available, or not. Usually, a "1" designates available data. When an instruction is issued, the scoreboard bit of the corresponding destination register is reset to 0. This indicates to all subsequent instructions requesting the register content in question that its value is not yet available. Later, when the instruction in question has been executed and the result is written into the destination register, the corresponding scoreboard bit is set to 1. From now on, the appropriate register content is available to all instructions requesting this register content.

As Figure 3.4.20 shows, there are two basic schemes for checking the availability of operands.

In the first scheme, the *scoreboard bits are directly checked* for the availability of operands, whereas in the second scheme *explicit status bits are maintained* and checked.

Figure 3.4.20 demonstrates both schemes. For the sake of simplicity, we assume an individual reservation station with out-of-order dispatch of one instruction per cycle as well as a single register file for all data types. Also for simplicity, we do not take renaming into account.

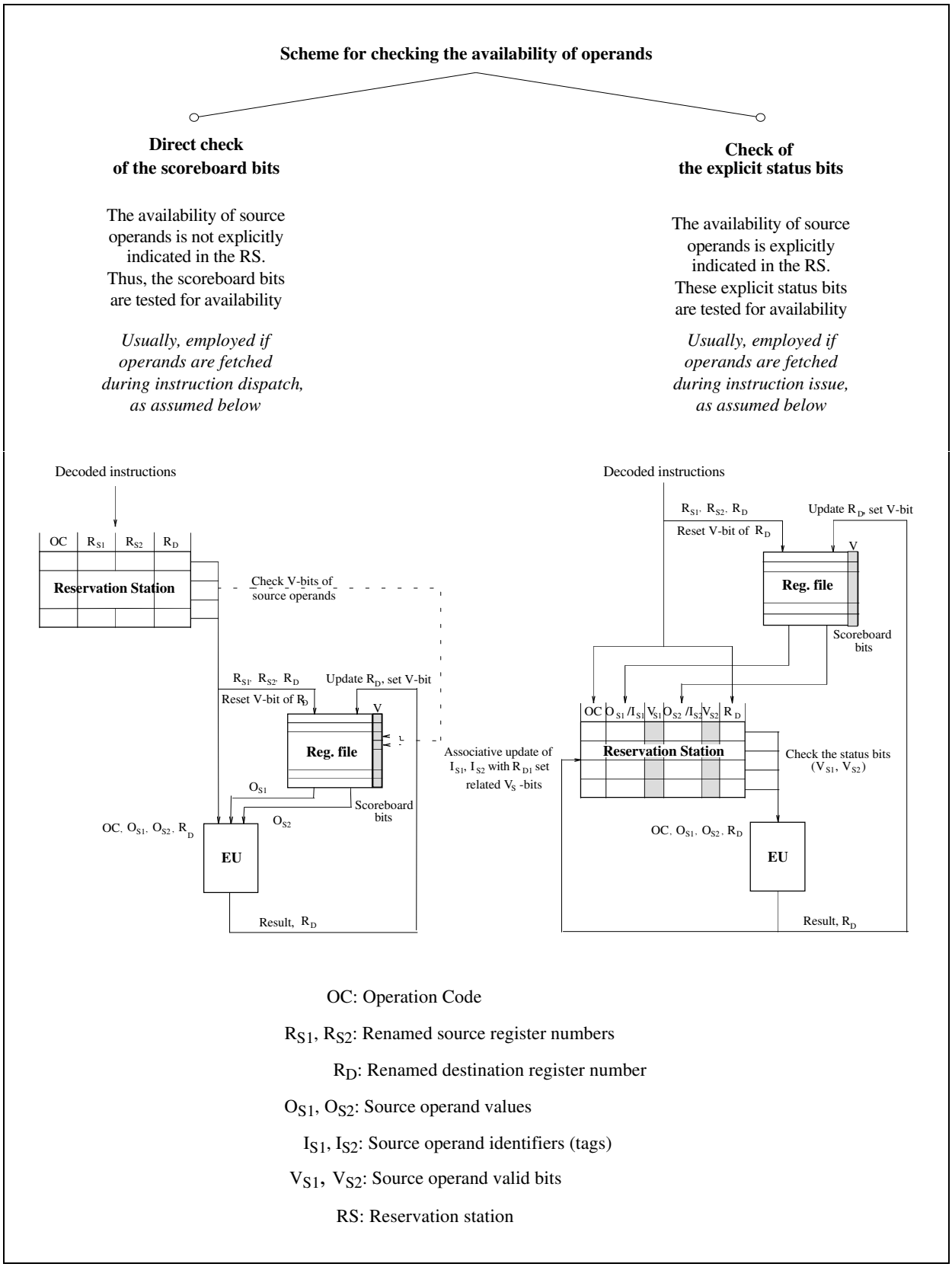

Figure 3.4.20: Schemes for checking the availability of operands

In the first scheme, the reservation station does not hold any explicit status information indicating whether source operands are available or not. Thus, the availability of source

operands is examined by a **direct check of the** associated **scoreboard bits**. This scheme is usually employed *when operands are fetched during instruction dispatch*, as is illustrated in the left part of the figure.

In this scheme, decoded instructions are written into the associated reservation station. The reservation station holds the operation code (OC), the source register numbers (R_{S1}, R_{S2}) and the destination register number (R_D) of the instructions issued.

During instruction issue, the scoreboard bits associated with the destination registers of the issued instructions are reset. Thus, all subsequent instructions are prevented from accessing register values that are not yet available.

In each cycle, instructions in the reservation station are checked to see whether any are eligible for execution. As already mentioned, this check is performed by a direct inspection of the scoreboard bits (V-bits) pertaining to the source operands of the instructions in the reservation station. If a single executable instruction is present, it will be selected for execution. When more than one executable instructions are available, one will be chosen according to the dispatch policy employed, and forwarded to the EU. During forwarding of an instruction, the operation code (OC) and the destination register number (R_D) of the selected instruction are transfered to the EU, and the source register numbers (R_{S1} and R_{S2}) are passed to the register file. After the source operands (O_{S1} and O_{S2}) have been fetched, the EU starts to perform the specified operation. When the operation has been completed, the result and its identifier (R_D) are forwarded to the register file to update the value of the destination register R_D. At the same time, the scoreboard bit of R_D is set to "1" in order to indicate the availability of R_D in subsequent checks.

The checking of the scoreboard bits for availability is performed by dedicated hardware which operates in accordance with the dispatch policy chosen. Usually, instructions are dispatched in an out-of-order fashion. Then all instructions in the reservation stations are checked in parallel, as discussed in Section 3.4.5.

In the other basic scheme, the ***availability*** *of source operands* ***is explicitly indicated*** in the reservation station **by** introducing ***status bits*** *(valid bits)*. The status bits are then tested to see whether operands are available or not. This scheme is usually employed, *if operands are fetched during instruction issue*, as demonstrated in the right part of Figure 3.4.20. In this case the reservation station holds the operation code (OC), the fetched source operands (O_{S1}, O_{S2}), the associated valid bits (V_{S1}, V_{S2}) and the renamed destination register number (R_D). However, if during issue a requested operand is not yet available, the register file delivers an identifier (I_{S1} or I_{S2}) instead of the lacking operand value. As in the previously discussed scheme, the *availability of the register contents* is managed by *scoreboarding*. During instruction issue the scoreboard bits of the destination registers of the issued instructions are reset to "0". Again, when the EU has produced the result, the

destination register (R_D) is updated and the corresponding scoreboard bit is set to "1". In this way, subsequently issued instructions can access the produced result value.

In this scheme, additionally the status of the reservation station must be updated in each clock cycle. For updating, generated results and their identifiers (R_D) are forwarded to the reservation station, as indicated in the figure. However, *updating* of the reservation stations requires an *associative search*, which finds all source register identifiers equal to the identifier of the generated result. All matching source register identifiers are substituted with the result value. At the same time the corresponding valid bits (V_{S1}, V_{S2}) are also set to "1" to indicate the availability of the corresponding operands. Thus, checking during the next cycle meets already updated status bits (V_{S1}, V_{S2}).

Two extensions of this basic scenario must also be examined: multiple reservation stations and split registers.

If *multiple reservation stations* exist, their updating must be done *globally*. We should take into account here that a not yet available operand value in a reservation station is not necessarily produced by the EU that is connected to it. Thus, for updating, all results must be forwarded along with the result identifiers to any reservation station, as indicated in Figure 3.4.21.

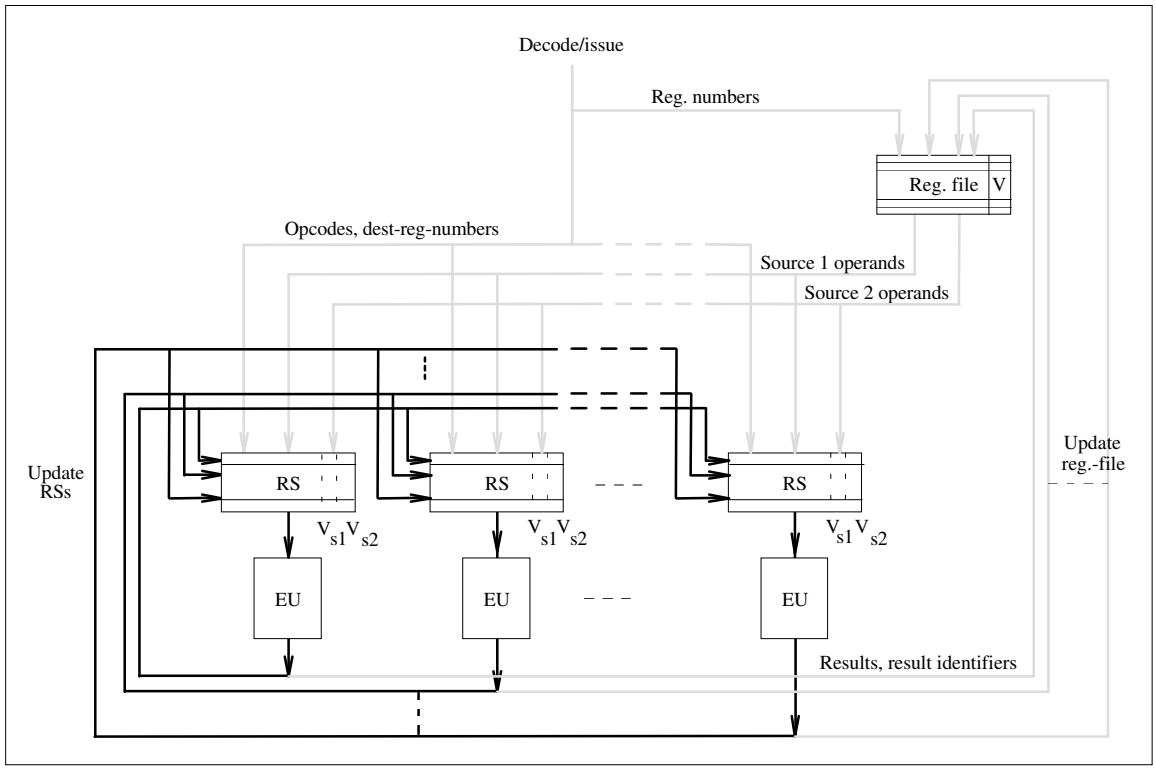

Figure 3.4.21: Use of multiple buses for updating multiple individual reservation stations

Thus, in the case of multiple individual reservation stations as many result buses are required as there are EUs that can operate in parallel. As a consequence, in each reservation

station multiple associative searches must be carried out, one for each result identifier supplied by one of the result buses.

Concluding this point we redraw Figure 3.4.21 in such a way that we represent multiple buses by a single line with multiple inputs and outputs (see Figure 3.4.22). Note that in this chapter usually we are using this kind of simplified representation.

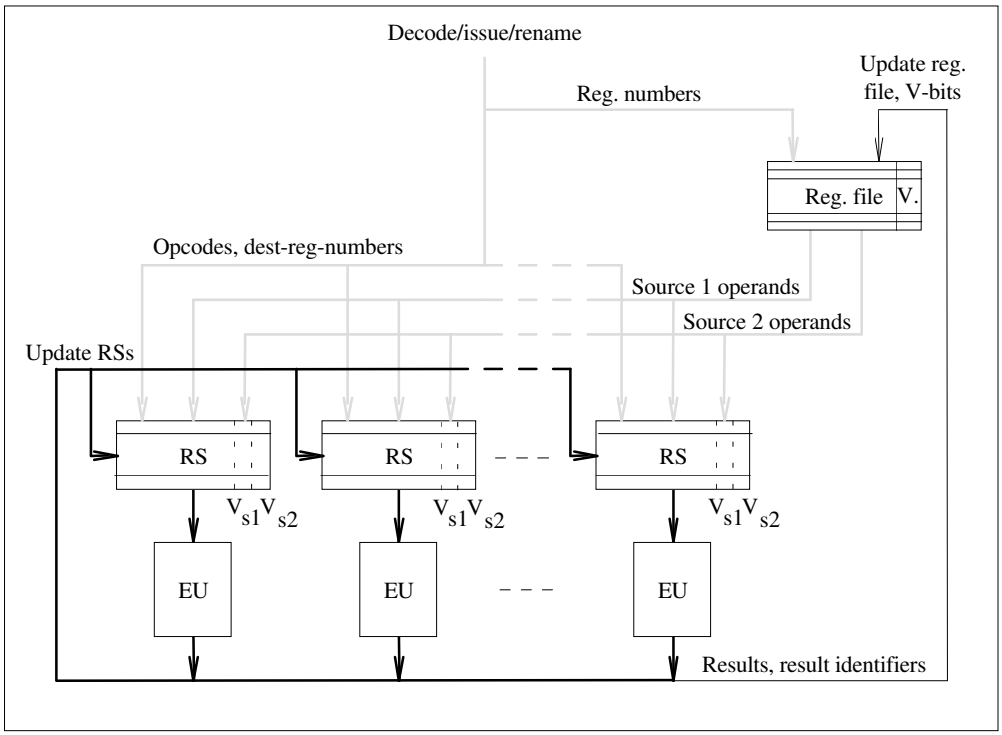

Figure 3.4.22: Updating of multiple individual reservation stations when operands are fetched during instruction issue

The second, scenario is when split registers are used for FX- and FP-data. In this case updating is separated into a distinct FX- and FP-part as depicted in Figure 3.4.23.

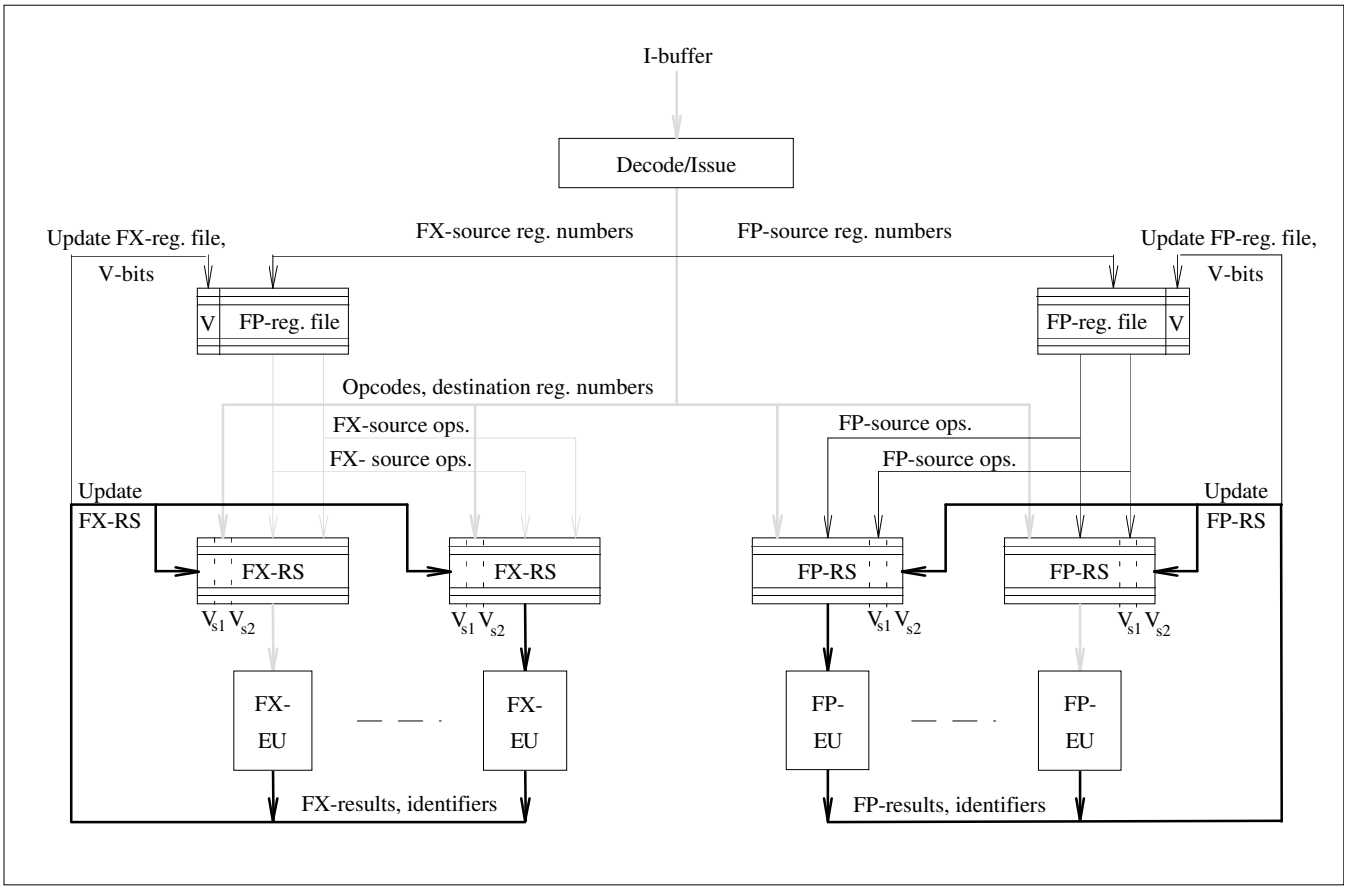

Figure 3.4.23: Updating of reservation stations in case of split FX- and FP-register files

Now FX-results and associated identifiers are sent to FX-reservation stations, whereas FP-results and result identifiers are forwarded to FP-reservation stations. Beyond this, there are no further significant differences to the original scenario.

While introducing the basic schemes for checking the availability of operands, for simplicity, we made a number of assumptions that need to be expanded upon. We remind that, we assumed individual reservation stations with out-of-order dispatch of one dispatched instruction per cycle. Furthermore, we did not took renaming into account.

First, let us consider using group or central stations. Then the dispatch scheme should be extended to provide instead of a single instruction multiple executable instructions to available EUs. In this case multiple availability checks are required, maximum as many in each cycle as the dispatch rate of the considered RS is.

On the other hand, when renaming is used, operands are fetched either from the rename buffers or from the architectural registers as explained in Section 3.5.1. Another difference is that in the case of renaming the rename buffers are to be updated with the generated results rather then the architectural registers.

Finally, we show a case example to demonstrate how operand availability checks are integrated into processor operation. Our example is the PowerPC 604 which internal data paths is depicted in Figure 3.4.24, based on Intel 1994.

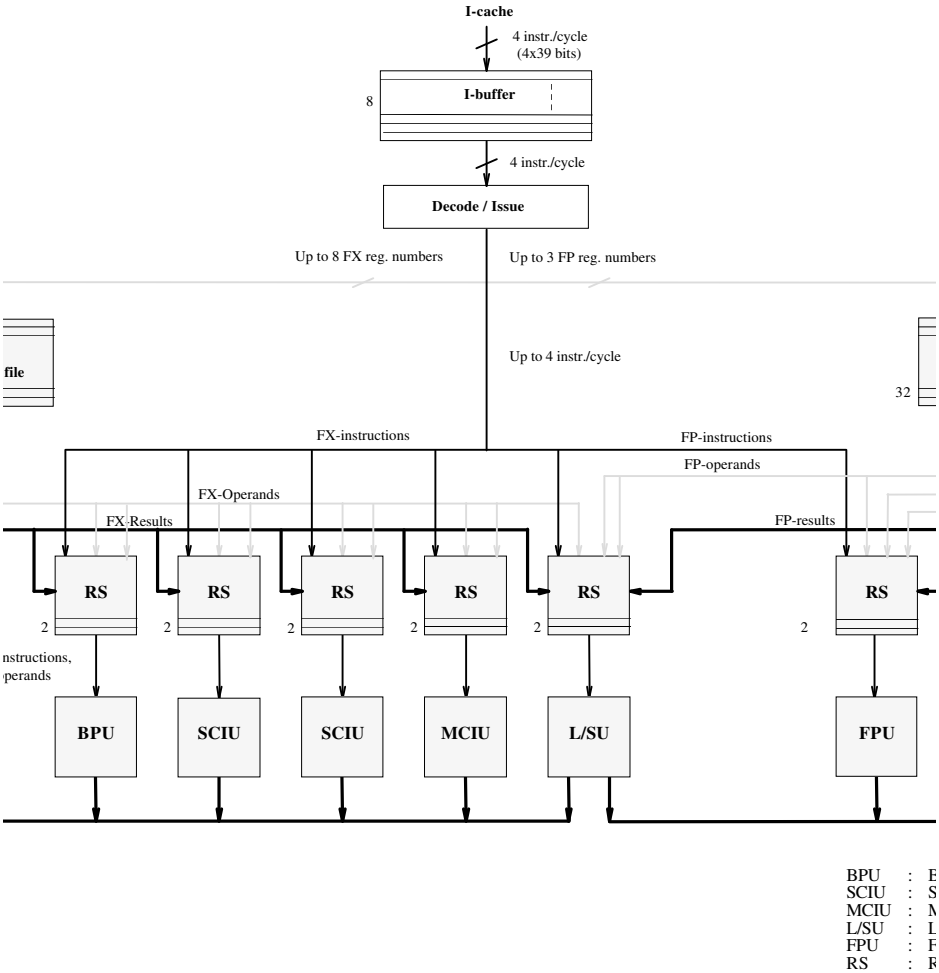

Figure 3.4.24: Internal data paths of the PowerPC 604

D. Sima, T. J. Fountain, P. Kacsuk Advanced Computer Architectures

The PowerPC 604 uses split registers for FX- and FP-data, and as do most recent processors, employs renaming. As discussed in Section 3.5.3, there are separate register files for rename- and architected registers. This processor has altogether six EUs. In addition to the branch processing unit (BPU), there are three FX-units, (2x SCIU, MCIU) , a load/store unit (SU) and an FP-unit (FPU) available. Two of the FX-units, called single cycle integer units (SCIU), are straightforward FX-ALUs executing instructions in one cycle. A third FX-unit executes multi-cycle FX-operations (MCIU). The load/store unit (L/SU) computes addresses and processes loads and stores. Thus, as indicated in the figure altogether four FX-result buses are available. In front of each FX-unit and of the load-store unit there is a reservation station with two entries.

The other part of the microarchitecture covers FP-processing. Here, there is a single FP-unit available (FPU) with a two-entry reservation station in front of it. Note that the L/SU also delivers FP-results. Thus, two FP result buses are needed to pass the FP-results for updating both the FP-reservation station and the FP-rename register file.

In this architecture up to 4 predecoded instructions are decoded and issued into the appropriate reservation stations in each cycle. As indicated in the figure, during issue up to 8 FX-operands and 3 FP-operands are fetched from the register files, if available. Here, rename- and architectural registers are accessed in parallel. (See Section 3.5.1 for further details.)

Operand availability is explicitly indicated in the reservation stations. If a referenced source operand is available, the operand itself is read into the reservation station. If the operand is not available an identifier is latched instead of the operand and the availability bit is set accordingly. Instructions held in the reservation stations are inspected in each cycle to see whether they are executable or not by checking the corresponding availability bits of the source operands. Instructions are dispatched from the reservation stations partly in order (BPU, FPU), partly out-of-order (remaining units), up to one from each station in each cycle.

The generated results are used to update the reservation stations and the rename registers. Four FX- and two FP-result buses transfer result values and their identifiers to the reservation-stations and rename registers. In the reservation stations an associative search is conducted to see whether generated results are waiting in any reservation station. If there is a match, the appropriate result is latched and the associated availability bit is set. Section 3.5.1 details the update process of the rename registers.

Treatment of an empty reservation station

When instructions arrive at an empty reservation station superscalar processors follow two different aproaches (see Figure 3.4.25). The **straightforward approach** is to process the instructions in the same way as with a partially filled reservation station. However, then entering instructions have to stay at least one cycle in the empty station until they are dispatched. The Nx586 in an example of a processor that treats an empty reservation station in this way.

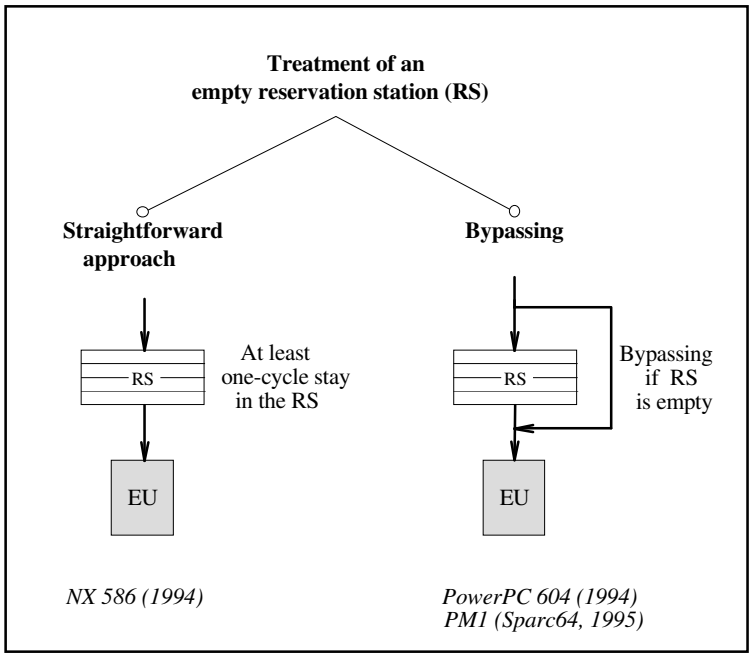

Figure 3.4.25: Treatment an empty reservation station

A more advanced approach for treating empty reservation stations is the use of **bypasses**. Here, some additional circuitry permits instructions to bypass an empty station and, immediately be forwarded to the EUs without any additional delay. The PowerPC 604, the PM1 (Sparc64) employ this more efficient approach.

Typical dispatch schemes

Thus far, we have discussed the individual components of dispatch schemes. In the following we point out what dispatch schemes are used most frequently in superscalar processors. For simplicity in our discussion, we focus on only two aspects, the *dispatch order* and the *dispatch rate*. These are the aspects which are used differently in various superscalar processors. Regarding other aspects of the design space we stress the following typical solutions:

Hiba! A könyvjelzõ nem létezik. Most recent processors employ register renaming and speculative execution. Thus, as selection rule, the

dataflow selection rule is used most often with the oldest instruction preferred if an arbitration is needed,

Hiba! A könyvjelzõ nem létezik. When operands are fetched during instruction dispatch most processors use the direct check method. Alternatively, when operands are fetched during instruction issue, explicit status bits are typically maintained and checked for operand availability.

Hiba! A könyvjelzõ nem létezik. An empty reservation station is usually bypassed.

If we focus only on the dispatch order and the dispatch rate, three typical approaches can be identified in recent superscalar processors, which are summarised in Figure 3.4.26.

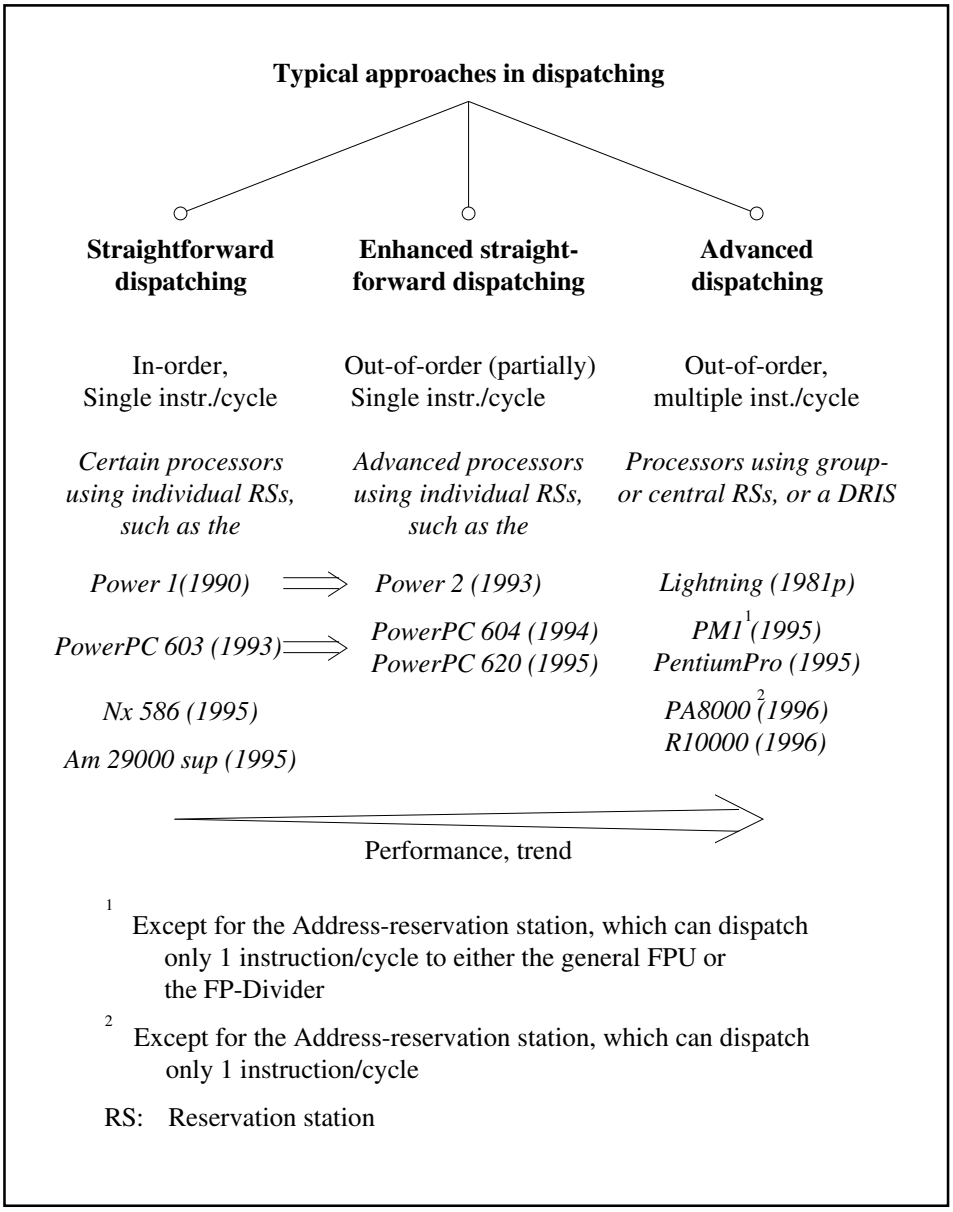

Figure 3.4.26.: Typical approaches in dispatching

In *straightforward dispatching,* instructions are dispatched one at a time. In this simple approach an instruction that is not yet executable *blocks* further dispatching until this instruction becomes executable. Some processors, using *individual reservation stations*, such as the Power1, PowerPC 603, Nx586 and the Am 29000 superscalar employ this straightforward but inefficient approach.

The efficiency of this simple approach can be increased by introducing out-of-order dispatching for particular reservation stations. We call this approach the *enhanced straightforward dispatching*. In this case, the reservation stations use an out-of-order dispatch, and subsequent executable instructions can *bypass* an instruction that is not yet executable. Obviously, this results in a performance increase over the basic approach. This approach is typically employed in advanced processors using *individual reservation stations*. Examples include the Power2, PowerPC 604 and the PowerPC 620.

The *advanced dispatching* is the most powerful. It allows *out-of-order dispatch* and is capable of forwarding *multiple instructions* per cycle to available EUs. This advanced dispatch policy is employed in processors *with group-stations* or with a *central reservation station* or with a DRIS. For instance, the Lightning, PM1 (Sparc64), PentiumPro, PA 8000 and the R10000 offer this efficient dispatching. Usually, subsequent processors of the same line tend to use more powerful approaches for dispatching. Thus, the trend in dispatching corresponds to the sequence of our discussion of the most frequently used approaches.

3.4.6 Detailed example of shelving

As indicated in in Figures 3.4.3-3.4.9, 3.4.15-3.4.18 and 3.4.20 shelving has an extremely complex design space. Consequently, there is a puzzling variety of possible ways how it can operate. From the large number of possible layouts, in the following, we demonstrate how shelving is carried out in one of the most frequently used scenarios while executing a short piece of code. Our scenario assumes individual reservation stations, issue bound operand fetching and explicit indication of operand availability. This model is used basically in the PowerPC 603-620, and in some sense also in the K5 and Am 29000 superscalar processors. Shelving also operates quite similarly in the PM1 (Sparc 64), the main difference being, the PM1 has group reservation stations.

Before getting door to details, let us first describe the assumptions we have made concerning the microarchitecture, instruction issue, shelving and dispatching scheme. As far as the architecture is concerned, for the sake of simplicity, we consider only its FX-part. Here, we assume split register files. Concerning instruction issue, we take for granted that the advanced superscalar issue policy is used. In our example no issue blockages can occur, therefore it does not matter how issue blockages are handled. As regards shelving buffers, we assume an individual reservation station for FX-instructions which has four entries and is capable of accepting at most two instructions per cycle. Furthermore, we assume that

operands are fetched during instruction issue. Finally, concerning the dispatch policy we assume a selection rule which considers instructions to be executable when source operands are all available. Furthermore, we take it for granted that the oldest instruction is preferred when more than one instruction is executable.

From the FX-reservation station instructions should be dispatched out-of-order, at most one instruction per cycle, to the connected FX-EU. We assume that the availability of operands is explicitly indicated by valid bits in the reservation station. Lastly, we suppose that an empty reservation station is not bypassed. To summarize the salient points, we assume individual reservation stations, issue bound operand fetching and explicit indication of operand availability.

a. Overview of the operation

As indicated in Figure 3.4.27.a, in our example decoded, issued (and renamed) FX-instructions are written into the FX-reservation station, up to two per cycle. According to our assumption, the source operands of the instructions will be fetched during issue, if available. Therefore, the source operand identifiers (R_{S1}, R_{S2}) of up to two instructions are sent during issue to the FX-register file. The register file maintains scoreboard bits (V-bits) associated with each of its entries in the same way as described above. Thus, during instruction issue the availability of the referred register operands can be checked by inspecting the corresponding V-bits. Available source operands are written into the corresponding fields of the reservation station, which are designated by O_{S1} and O_{S2} in Figure 3.4.27.a. At the same time the associated valid bits (V_{S1} and V_{S2}) are set and the scoreboard bits of the destination registers R_D are reset in order to prevent all subsequent instructions from reading referenced but not yet available data.

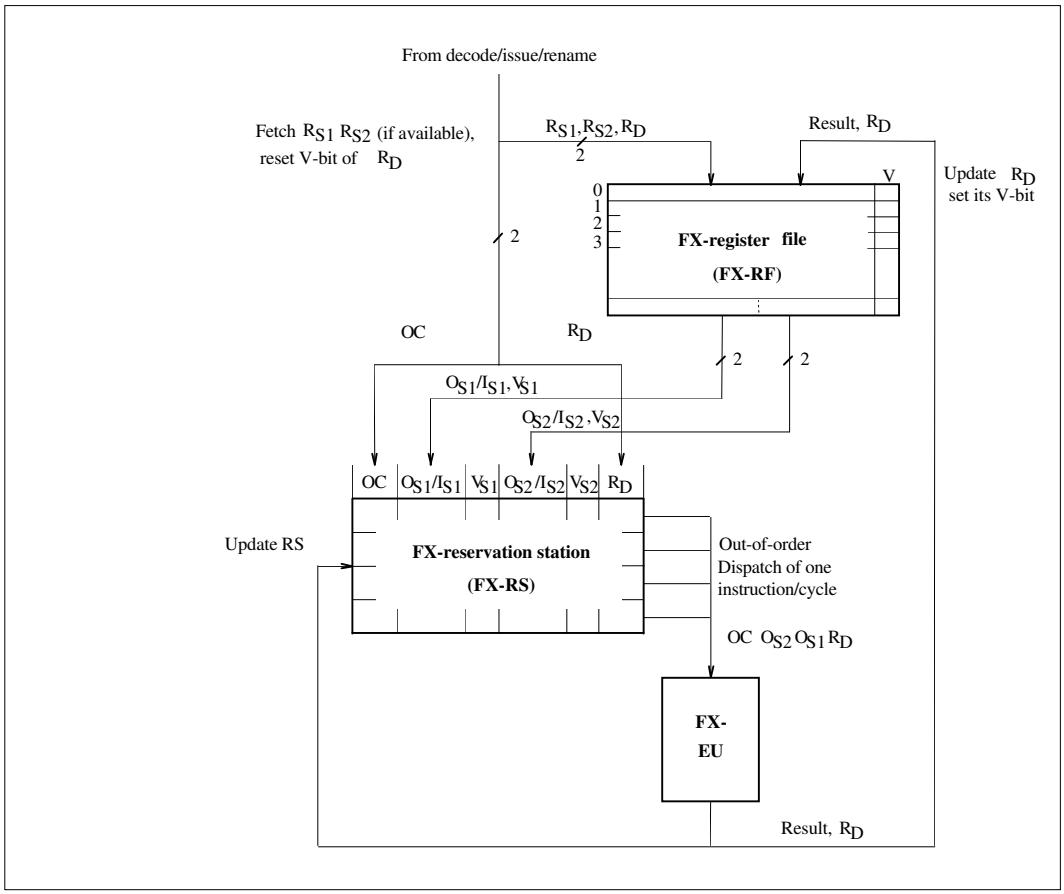

Figure 3.4.27.a: Operation principle of shelving when operands are fetched by instruction issue and the reservation station contains status information about the availability of the source operands

If a source operand is not available (which is indicated by the reset value of the corresponding V-bit:), a tag is stored (labelled as I_{S1} and I_{S2}) instead of the source operand itself. This tag usually identifies where the lacking operand will come from. In our example let us suppose that the destination register identifier, that is R_D, will serve as a tag.

In the next step, all instructions in the reservation station are checked in parallel to find any instructions whose source operands are available and, thus, the instruction is executable. The check is performed simply by investigating the valid bits of the operands (V_{S1} and V_{S2}). If both are set, the corresponding instruction is eligible for dispatching.

Provided that a single executable instruction is found, it will be forwarded to the FX-EU (FX-EU) connected to the reservation station. When more than one executable instruction is available, the "oldest" one will be selected and transmitted to the FX-EU as in the preceding example.

When the instruction which has been dispatched to the FX-EU is completed, the result along with its identifier (R_D) is sent to both the register file and the reservation station for updating. After the register file has received the identifier and the associated result value, the corresponding R_D entry is updated. Furthermore, the related scoreboard bit is set to 1 to indicate the availability of the associated value.

At the same time the result value and the result identifier are also forwarded to the reservation station. Here, an associative search is needed to find any matching source operand identifiers. Fitting source operand identifiers are substituted with the corresponding value and the related status bits (V_{S1} and V_{S2} bits) are set to "1" to indicate their availability.

In the next section we detail the operation just overviewed using the following short code sequence:

```
cycle i:      mul  r1, r2, r3;
cycle i+1:    ad   r2, r3, r5;
              ad   r3, r4, r6;
```

with the interpretation

r3 **Hiba! A könyvjelzõ nem létezik.** r1 * r2, etc.

In our demonstration we assume that the "mul" instruction is issued in cycle i, whereas the two "ad" instructions in cycle i+1.

b. Issue of the "mul" instruction into the reservation station in cycle i

After decoding and renaming, the "mul" instruction

```
mul      r1, r2, r3
```

is issued into the reservation station (Figure 3.4.27.b).

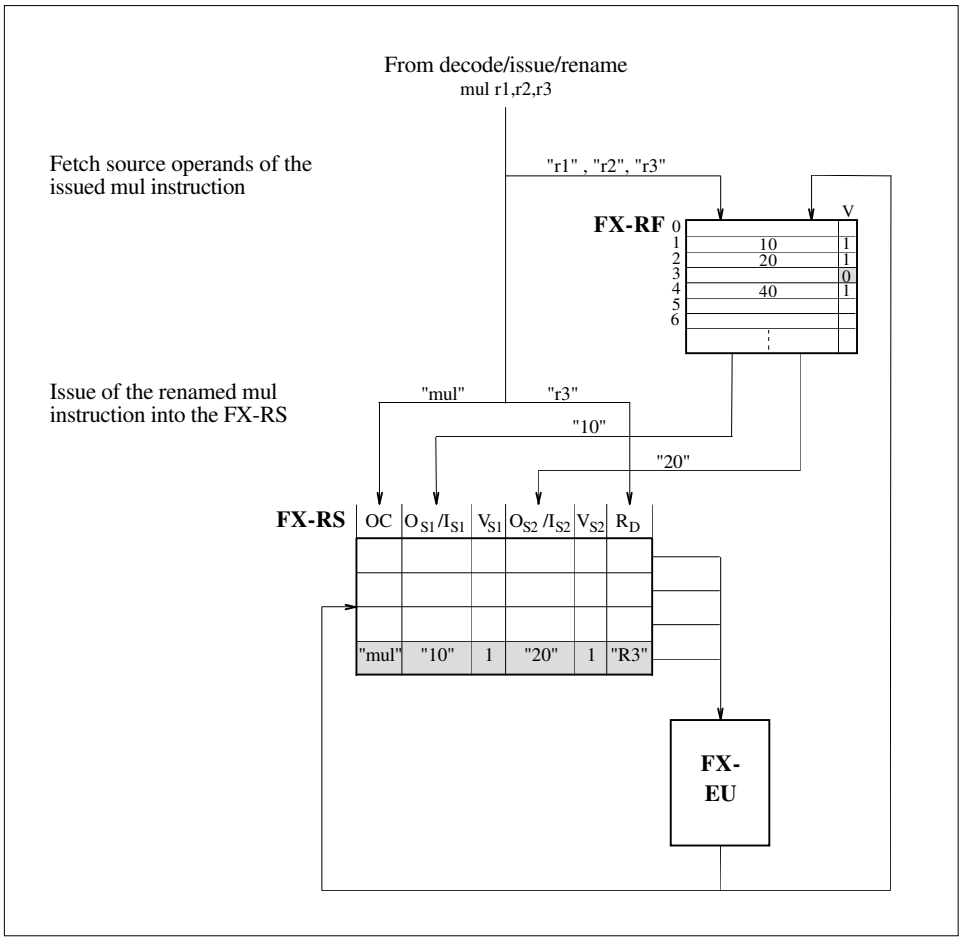

Figure 3.4.27.b: Issue of the mul instruction into the reservation station in cycle i with fetching the corresponding operands

The operands of the "mul" instruction (r1 and r2) will already be fetched during instruction issue from the register file. As the associated V-bits indicate, both values are available and thus, the fetched values (r1 = "10", r2 = "20") are sent to the matching fields of the FX-RS. In addition, the scoreboard bit of r3 is reset to prevent any further access to it.

c. Dispatching of the "mul" instruction from the reservation station and issue of the subsequent two "ad" instructions into the reservation station, in cycle i+1

In the next cycle (cycle i+1), the valid bits (V_{S1} and V_{S2}) of all instructions in the reservation station are checked, (see Figure 3.4.27.c.) Since both source operands are available, the "mul" instruction is passed for execution to the FX-EU.

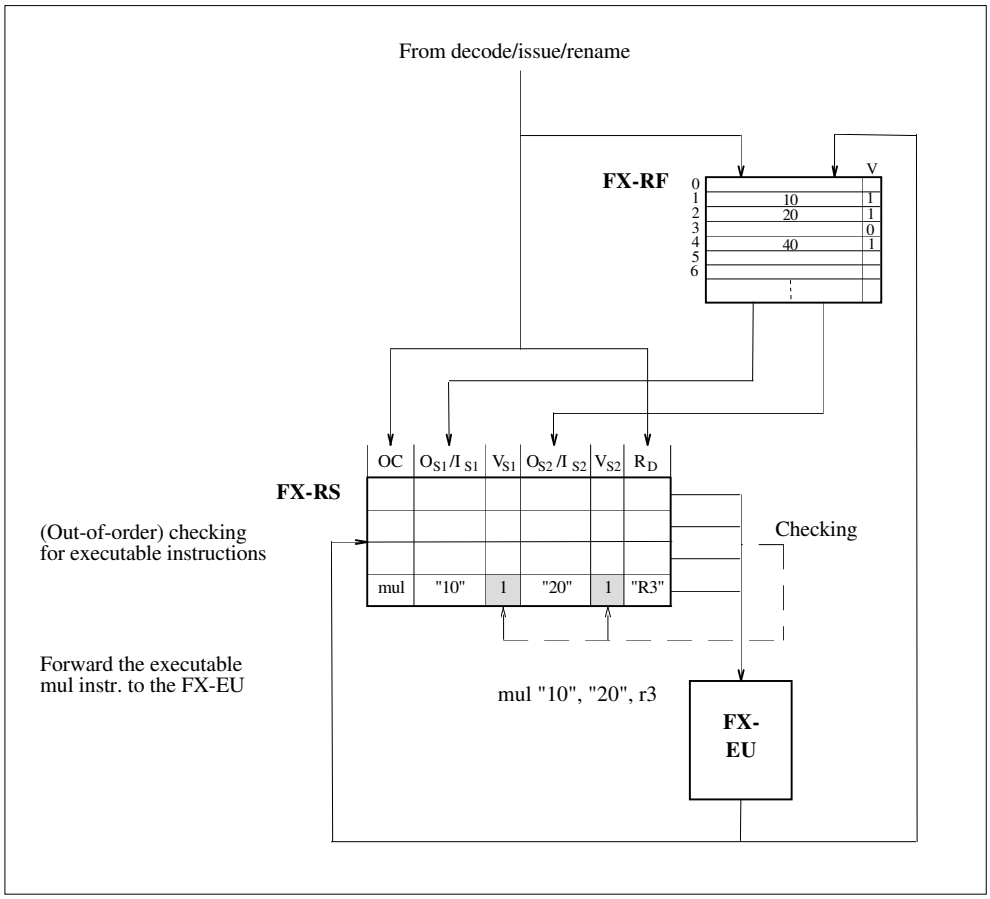

Figure 3.4.27.c: Checking for executable instructions and dispatching of the "mul" instruction in cycle i+1

In the same cycle, the next two "ad" instructions

```
ad  r2, r3, r6;
ad  r3, r4, r5;
```

arrive, as Figure 3.4.27.d indicates.

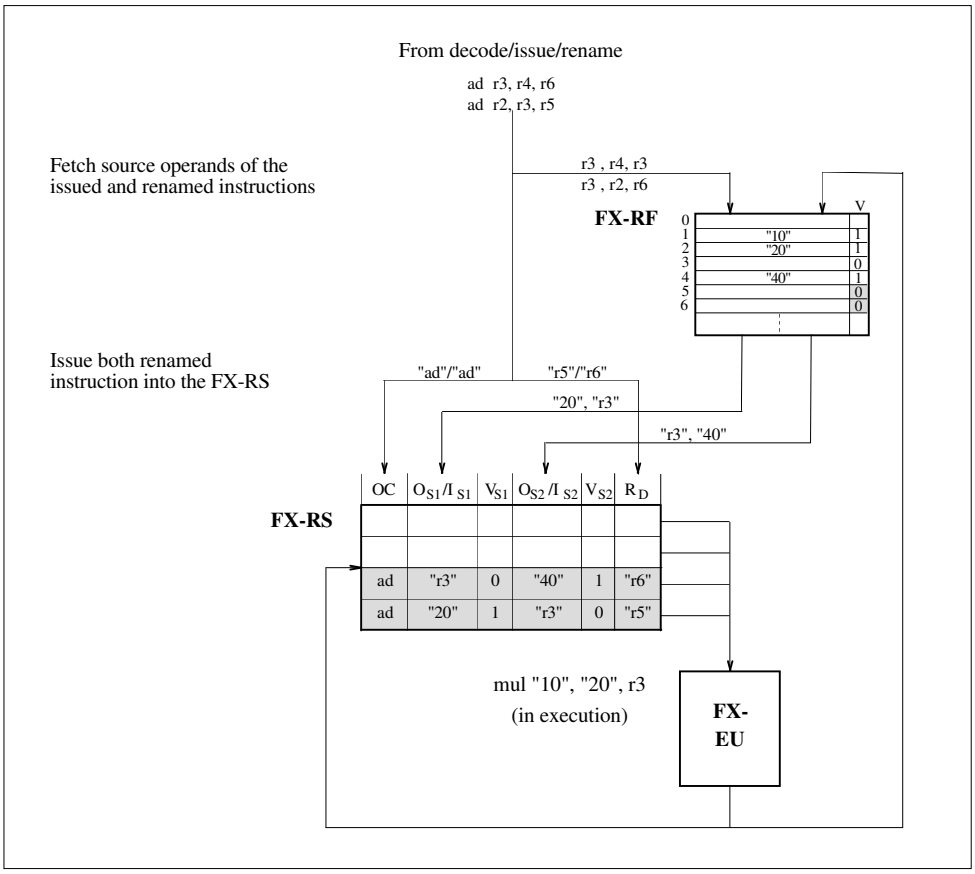

Figure 3.4.27.d: Issue of the subsequent two ad instructions into the reservation station in cycle i+1

In connection with this, two activities take place. First, the scoreboard bits of both destination registers (r5 and r6) are reset. Second, all required source operands are attempted to be fetched from the register file. In this case, the values of r2 and r4 are available, as indicated by the associated scoreboard bits. However, the scoreboard bit of r3 is reset. Thus, in place of the missing operand value its identifier (r3) is passed to the reservation station. In the reservation station the status bits of the source operands (V_{S1} and V_{S2}) are set accordingly, ("0" for a not available source operand and "1" for an available one).

d. Checking for executable instructions in cycle i+2

In the next cycle (cycle i+2), again, all instructions in the reservation station are checked for eligibility for dispatching. Since one of the operand values is not yet available (r3), no new instruction can be dispatched in this cycle, as indicated in Figure 3.4.27.e.

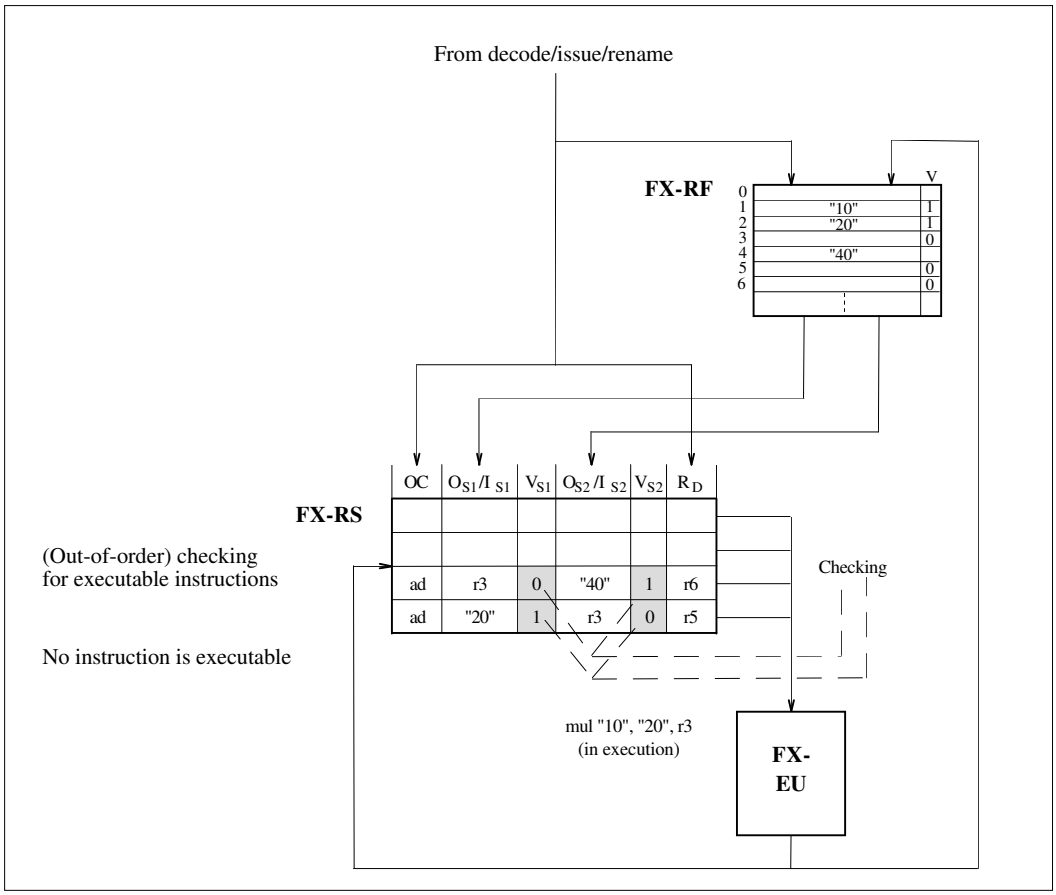

Figure 3.4.27.e: Checking for executable instructions in cycle i+2

e. Completing the computation of the "mul" instruction, checking the instructions held in the reservation station and forwarding the "older" "ad" instruction for execution, in cycle i+3

Let us now assume that the execution of the "mul" instruction in this cycle has been finished. Then the result of the operation ("200") and the associated identifier ("r3") appears on the result bus, as seen in Figure 3.4.27.f.

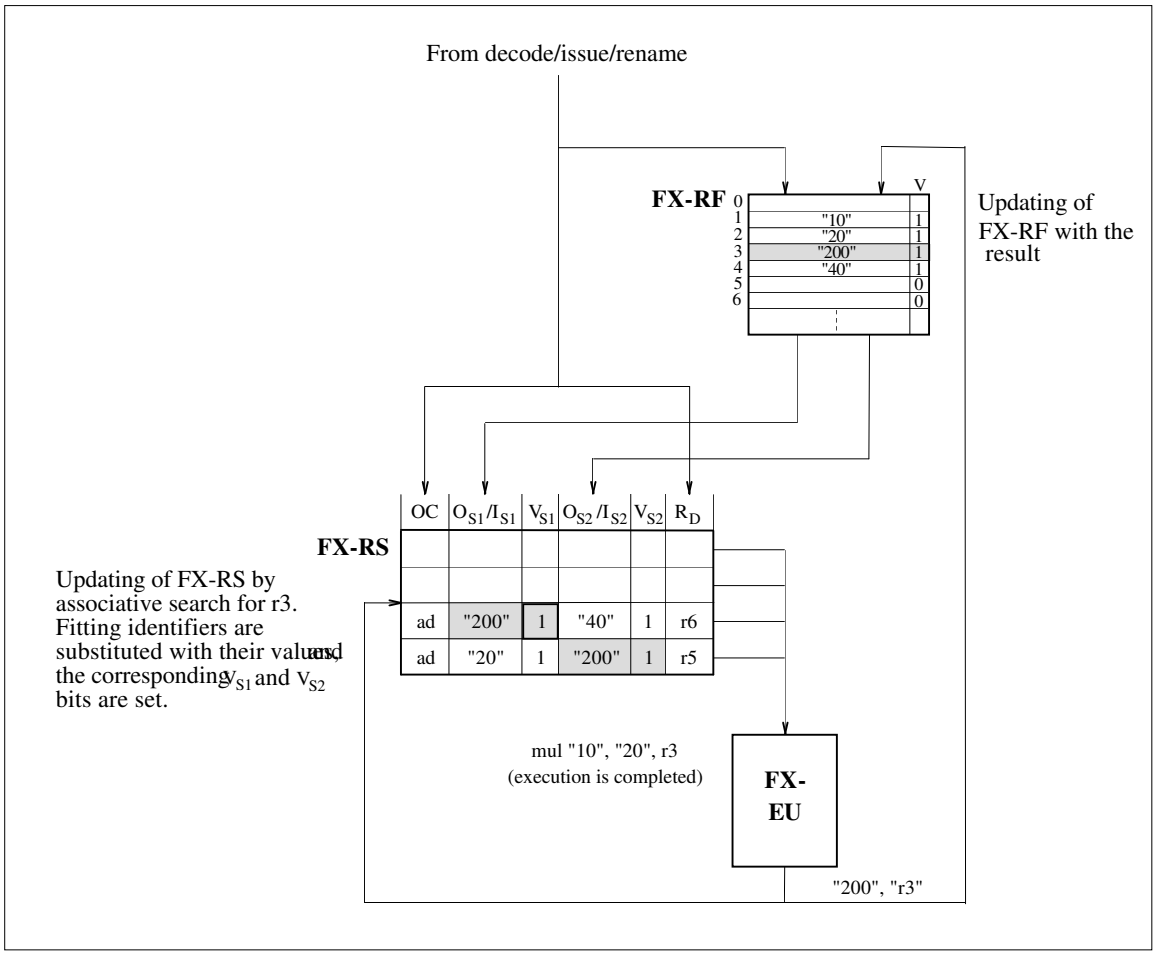

Figure 3.4.27.f: Updating the FX-register file with the result of the mul instruction in cycle i+3

Now both the register file (FX-RF) and the reservation station (FX-RS) have to be updated. The content of r3 in the register file is changed to the value of "200" and the associated scoreboard bit will be set to 1. From now on, the value of r3 may be accessed by all subsequently issued instructions. On the other hand, the reservation station is updated by performing an associative search for the identifier r3 in all source operand fields (O_{S1}/I_{S1}, O_{S2}/I_{S2}). In our case, there are two hits. In these fields the identifier of R3 is replaced with the operand value of "200". In addition, the related status bits (V_{S1}, V_{S2}) are also set to indicate operand availability.

Subsequently, in this cycle, a check is again performed for executable instructions, as shown in Figure 3.4.27.g. At this time all status bits of the required source operands are set, and, both "ad" instructions in the reservation station are eligible for execution. Since more than one executable instruction was found, the arbitration rule is used for selection. According to this rule the "older" instruction is preferred and sent for execution to the FX-EU.

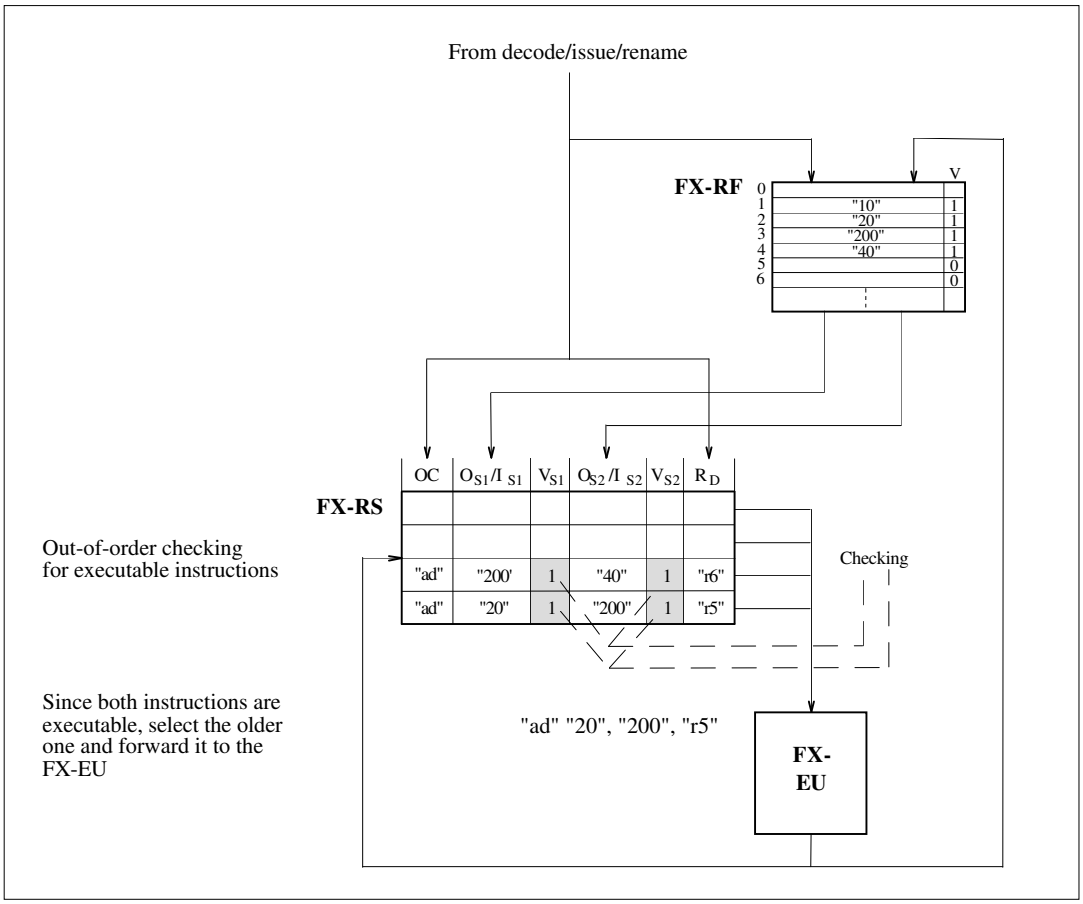

Figure 3.4.27.g: Checking for executable instructions and dispatching the "older" "ad" instruction in cycle i+3

This concludes the description of our demonstration example, since all subsequent processing steps are more or less trivial.