3.5 Register renaming

Register renaming, introduced earlier in Section 3.3.2, is a standard technique for removing false data dependencies, i.e. WAR and WAW dependencies among register data. It was first suggested by Tjaden and Flynn in 1970. They intended to use register renaming for a limited set of instructions that corresponds more or less to the class of load instructions, although they did not yet use the term "renaming". Keller (1975) introduced the designation *"register renaming"* and interpreted it for all suitable instructions. He also described one possible way to implement register renaming in processors, which we will discuss in Section 3.5.3.

Register renaming presumes the *three operand instruction format*. To illustrate this precondition, let us consider a two operand instruction, say

```
ad r1, r2,
```

with the interpretation

```
r1 ### (r1) + (r2)
```

As we know, in the two instruction format the result is written back in place of one of the source operands, in our example in place of r1. However, for renaming a different register than one of containing a source operand (r1) has to be used for the destination, say r11. Thus, after renaming we get

```
r11 ### (r1) + (r2).
```

This means that the renaming of two operand instructions always ends up as three operand instructions. As a consequence, two operand instructions can only be renamed using an internal two-to-three-operand conversion. For instance, the ES/9000 (Liptay 1992), which has a two operand architecture makes use of an internal conversion for renaming as described above.

Register renaming may be implemented either statically or dynamically, as Figure 3.5.1 depicts.

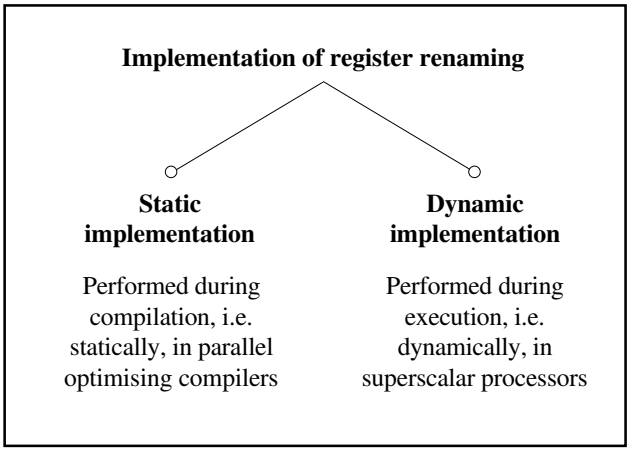
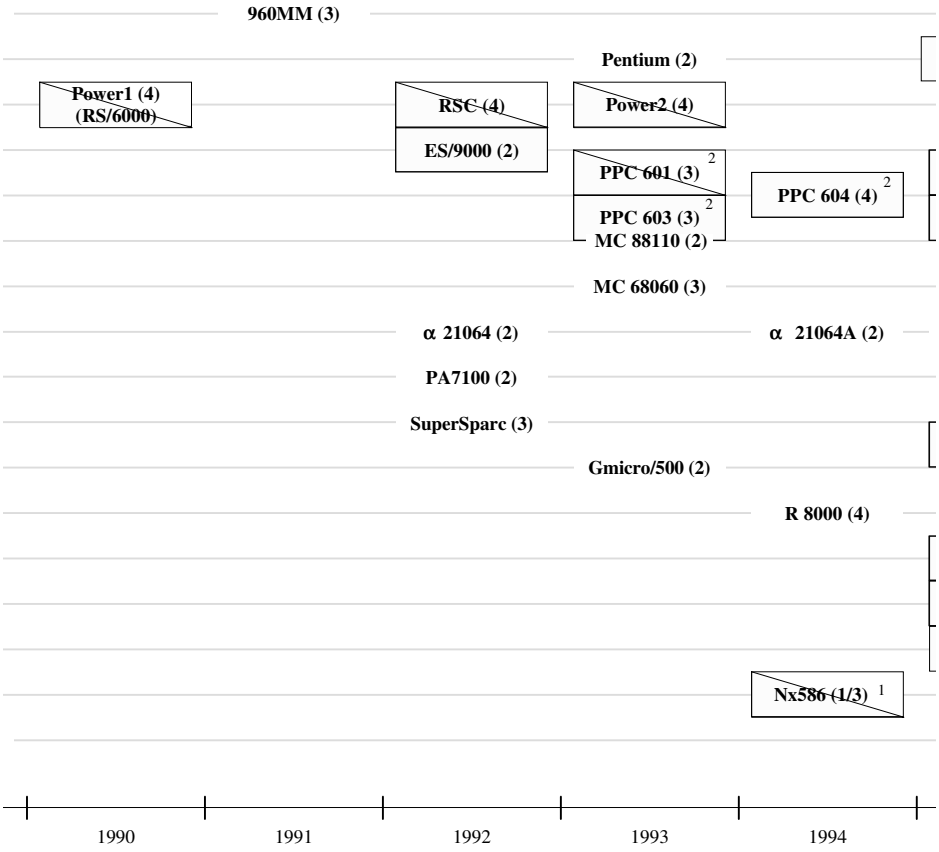


Figure 3.5.1: Implementation of register renaming

In the case of a *static implementation*, register renaming is carried out during compilation. This technique was first introduced in parallel optimising compilers of pipelined processors, and later on in superscalar processors.

When register renaming is implemented *dynamically,* renaming takes place during execution. Obviously, this requires extra circuitry in terms of supplementary register space, additional data paths and logic. Dynamic renaming has been used in advanced superscalar processors since the early '90s, as shown in Figure 3.5.2. Renaming was clearly introduced in two stages. In the first stage renaming was only partially implemented. Here, renaming is confined to one or a few particular data types, as discussed in Section 3.5.2. For instance, the processors Power1, Power2 and Nx586 employ partial renaming. Full renaming has emerged in the second stage, since about 1992.



naming

ming

[1] The Nx586 has scalar issue for CISC instructions but a 3-w superscalar core for converted RISC instructions.

[2] PowerPC is abbreviated here to PPC

3.5.2: Chronology of introduction of renaming in commercial superscalar processors

In both stages IBM was the first vendor to introduce renaming, initially in its Power-line with the Power1 (RS/6000) and later on in the high-end processors of its ES/9000 family. IBM continued to implement renaming in the Power-based PowerPC-line with the exception of the first model, the PowerPC 601. All vendors of significant superscalar processors have introduced renaming in their most recent models, except DEC in its **###**-line and Sun in the UltraSparc.Thus, register renaming is now almost a standard feature of superscalar processors.

As pointed out in Section 3.3.2, most superscalar processors use shelving and renaming at the same time. Only a few processors employ renaming but do not use shelving, such as the M1 and the PowerPC 602.

In the following section we focus on the technique of register renaming, emphasizing its design space and actual practice in superscalar processors.

3.5.1 The design space

The *design space of register renaming* resembles that of shelving. As Figure 3.5.3 shows, it consists of the following main components: the scope of register renaming, the layout of the rename buffers, the operand fetch policy, and the number of renames per cycle. These components are discussed in the following sections.

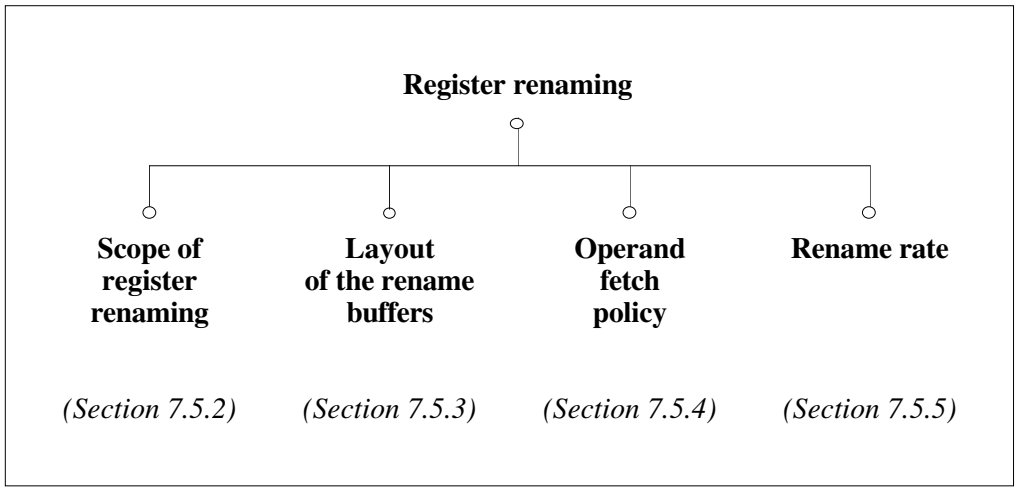


Figure 3.5.3: Design space of register renaming

3.5.2 Scope of register renaming

Most first generation superscalar processors did not employ renaming, like the PA 7100, SuperSparc, **###** 21064, PowerPC 601, Pentium and others. In the first stage of introduction, most early superscalar processors introduced **partial renaming**. In this case,

renaming is restricted to a particular instruction type or to a few types. Examples are the Power1 (RS6000), Power 2 and the Nx586. Of these, the Power1 (RS6000) renames only FP-loads, whereas Power2 extends renaming to all FP-instructions. We note that the Power1 has only one FP-unit, and it executes FP-instructions in sequence. FP-instructions are not renamed. On the other hand, the Nx586 is an FX-processor, thus it executes only FX-instructions.

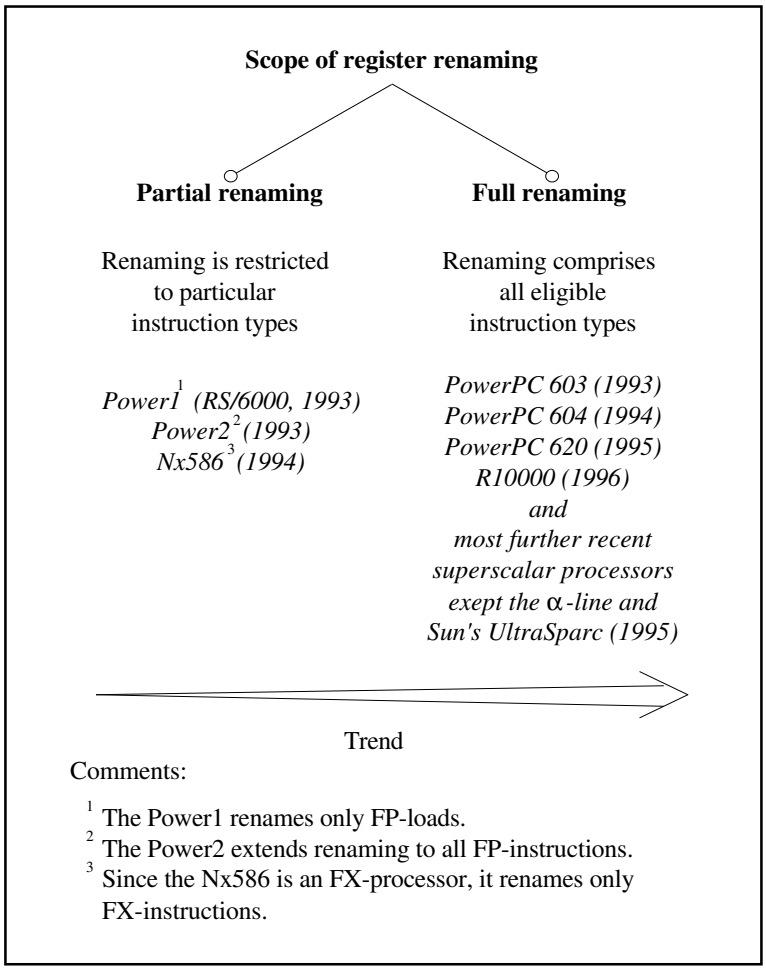


Figure 3.5.4: Scope of register renaming

The next stage in the introduction of register renaming is **full renaming**. In this case, renaming is carried out for all eligible instruction types. As Figure 3.5.4 demonstrates, virtually all recent superscalar processors employ full renaming. A few examples are the PowerPC 603-PowerPC 620 and the R10000. A noteworthy exception is DEC's entire α-line, including the recent ###21164 and Sun's UltraSparc. These processors do not use renaming at all.

3.5.3 Layout of the rename buffers

The layout of the rename buffers establishes the actual framework for renaming. Its three basic components are the type and the number of the rename buffers as well as the basic mechanism which is used for accessing rename buffers, as shown in Figure 3.5.5.

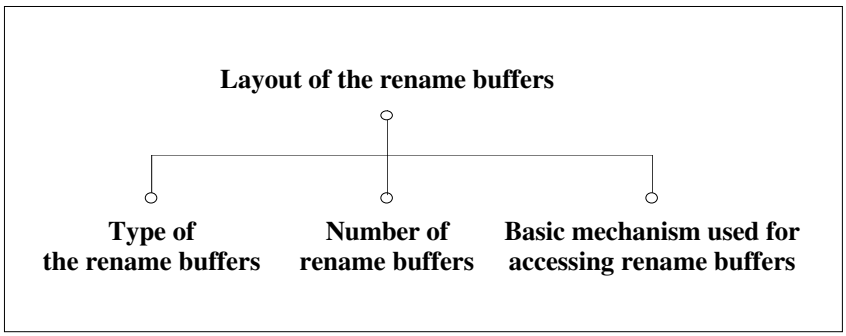


Figure 3.5.5: Layout of rename buffers

Types of rename buffers

Certainly, the chosen type of rename buffer has the largest impact on renaming. It is definitive for the basic approach of the implementation and thus it determines where the intermediate results of the instructions are to be written into or read from. Here, we designate as *intermediate* those *results* which have already been generated but are not yet qualified to modify the actual program state by writing them into the architectural registers. Intermediate results have to wait for permission to modify the actual program state until it is sure that the modification of the program state does not hurt the sequential consistency of the execution, as detailed in Section 3.7.

In the following, we give an overview of the basic types of rename buffers. For simplicity, initially we will suppose a single register file for all data types processed. Then, at the end of this section we will extend our discussion to the split register scenario.

As Figure 3.5.6 illustrates, there are four fundamentally different ways to implement rename buffers. These are: using a merged architectural and rename register file, employing a separate rename register file, or holding renamed values either in the ROB or in the DRIS. We note that in our discussion, we use the term **rename buffer** in a generic sense to denote any one of the implementation alternatives mentioned.

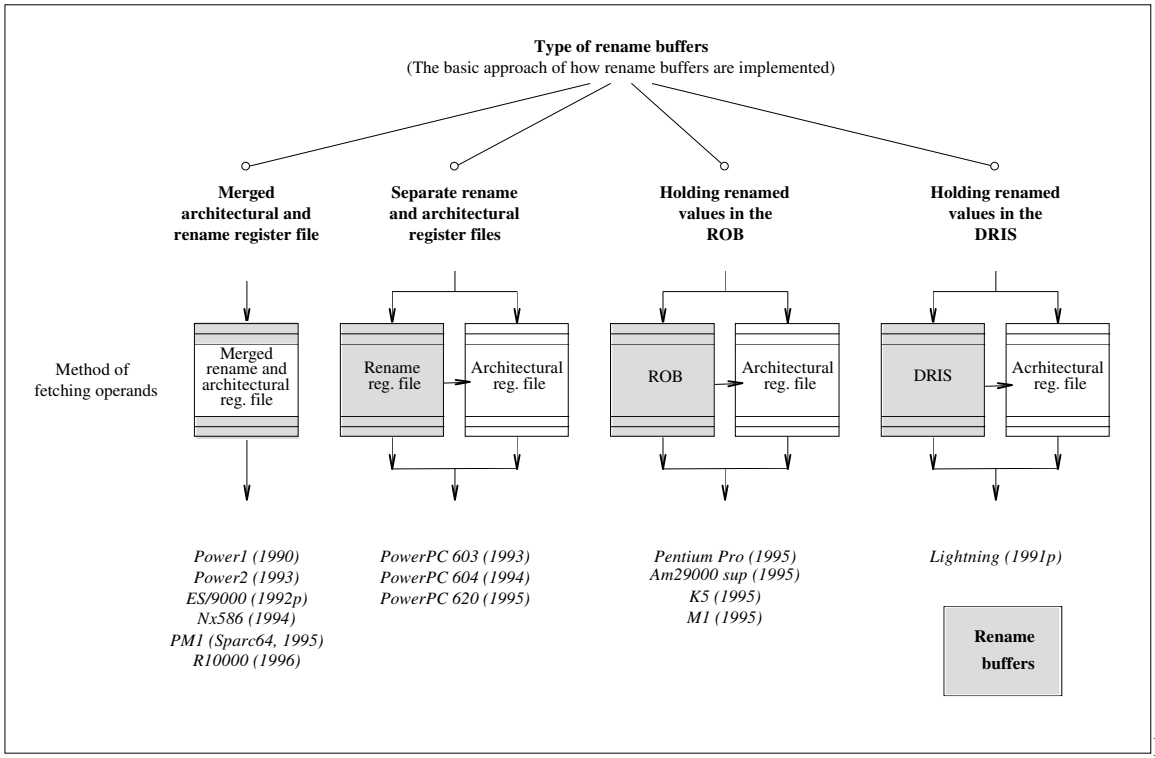


Figure 3.5.6: Type of rename buffers

According to the first approach, rename buffers are implemented along with the architectural registers in the same physical register file, called **merged architectural and rename register file**. Here, both architectural and rename registers are dynamically allocated to particular registers of the same physical register file. The merged register file obviously has to provide a large enough number of physical registers to implement both the architectural and rename registers. For instance, the Power1 provides 40 and the R10000 64 physical registers for renaming of 32 FP-architectural registers.

This scheme operates in the following way. Let us assume that at a given moment the architectural registers are associated with a particular set of physical registers. Now, for each *destination register* specified in any instruction, a free physical register is allocated, if available. In this way, the architectural registers are represented by a dynamically changing set of physical registers. The actual allocation of the architectural registers to physical registers is tracked in a *mapping table*. For instance, let us assume that the rename process allocates to the destination register of the instruction

```
ad   r2, ..., ...,
```

i.e. to r2 to the physical register p3. An entry is then set up in the mapping table, as shown in Figure 3.5.7. The first part of the entry is the entry valid bit. If this bit indicates a valid entry the second part is interpreted as the rename buffer index (RB-index), which points to the physical register used as a rename register. This is in our case p3.

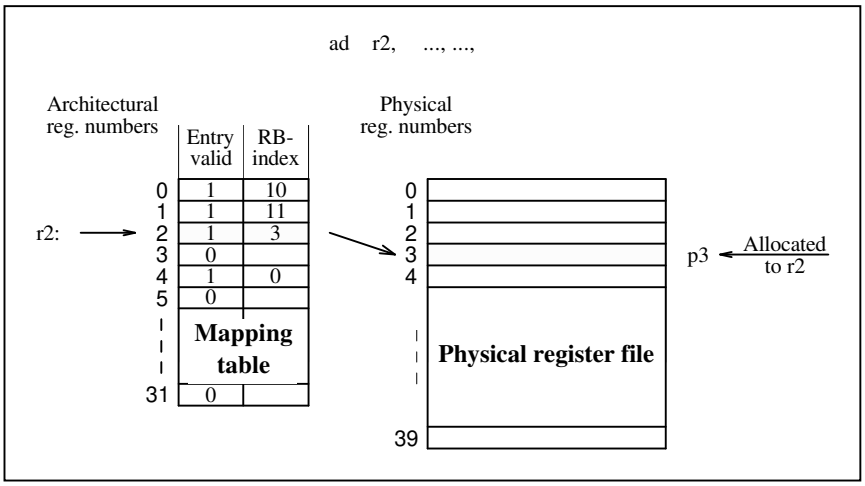


Figure 3.5.7: Renaming architectural register r2 to physical register p3

After renaming we get:

```
ad   p3, ..., ...
```

As described before, *source operands* are renamed simply by substituting the architectural register numbers with the physical register numbers found in the corresponding RB-index fields. Figure 3.5.7.b illustrates this for the instruction:

```
ad  ..., r0, r1
```

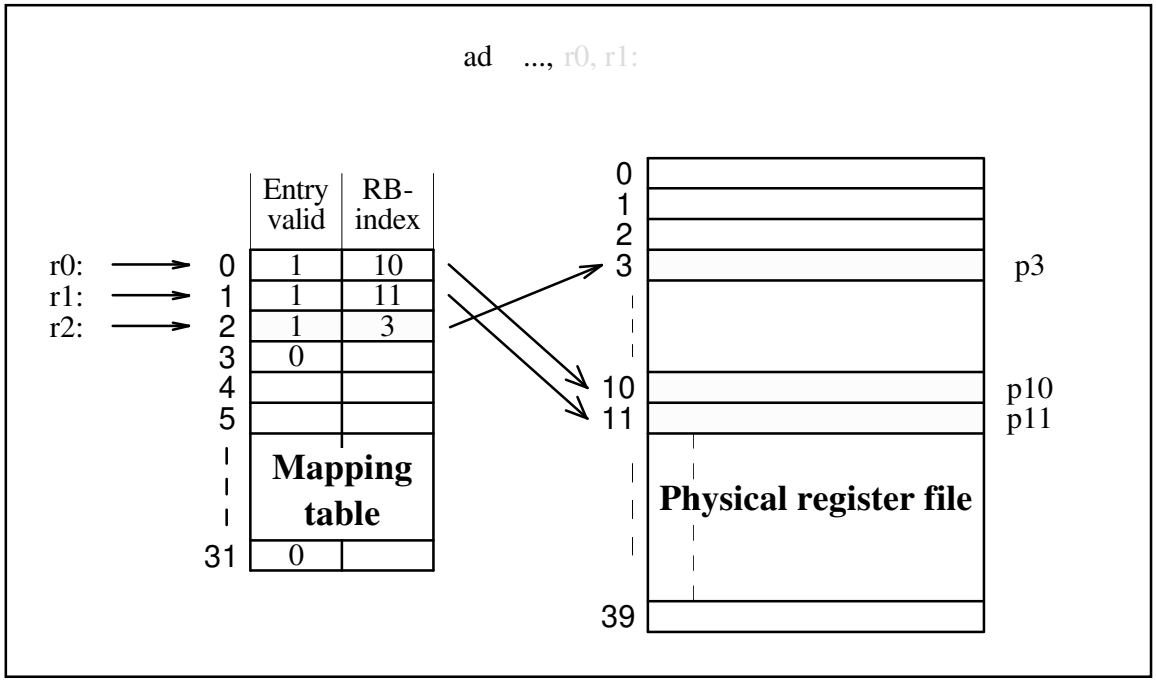


Figure 3.5.8: Renaming the ad ..., r0, r1 instruction to ad p3, p10, p11

In this way, renaming of the instruction ad r2,r0,r1 results in

```
ad   p3, p10, p11.
```

An additional scheme is required in order to *reclaim physical registers* no longer in use. In the case of a merged register file, reclaiming is a rather complex task, which we do not are going to discuss. For a possible solution we refer to

Renaming in this manner was first purposed by Keller (1975). Most early superscalar processors which introduced register renaming employed this scheme, including the Power1 (RS 6000), Power2, ES 9000 and Nx586. Further examples of using merged register files are the recent PM1 (Sparc64) and the R10000.

In a*ll other types* of renaming rename buffers are implemented separately from the architectural registers. In these cases rename buffers are realized either as separate rename register, as file on extension of the reorder buffer (ROB) (see Section 3.7.3) or as part of the DRIS (see Section 3.7). All three alternatives operate basically in the same manner, the only difference is the location of their implementation.

When renamed register values are held in a separate register file, this file is called the **rename register file**. Here, each time a *destination register* is referred to, a new rename register is allocated to it. This allocation remains valid until either a subsequent instruction refers to the same destination register and the architectural register is reallocated to a different rename register or until the instruction which uses that particular destination register completes (retires) and the allocation becomes invalid. The PowerPC processors PowerPC 603 - PowerPC 620 are examples that employ a rename register file.

A further possibility to implement rename buffers is to use the **reorder buffer** (ROB) therefore. As described in Section 3.7.3, while using a ROB each issued instruction will be allocated a separate ROB-entry. Thus, it is quite natural to store the generated (intermediate) result of that instruction in the same entry as well. All that is needed is to extend each ROB entry with appropriate fields. Examples of processors which use the ROB for storing intermediate results are the Am 29000 superscalar, K5 and the PentiumPro.

In the final method of implementation, the **DRIS** is used in the same manner as the ROB. The Lightning made use of this alternative.

In all three cases of using a distinct register file for renaming, intermediate results are held in the distinct file until their retirement. During retirement the content of the related rename buffer (rename register entry, ROB- or DRIS entry) is written back into the architectural register file. In this way, the architectural register file is updated and the rename buffer is reclaimed for further use.

Seen from another point of view, the type of rename buffer also specifies the *location from where operands can be fetched.*

As far as operand fetching is concerned, let us take into account that a required source operand is either an intermediate result or is a sane result. Intermediate results are held in a rename buffer, whereas sane results are stored in the architectural register file. Thus, while looking for an operand, both possible places have to be accessed at the same time and the access mechanism has to provide the right selection. This requires a priority access. If both the architectural and the intermediate register contents exist, then the intermediate content shall be delivered as the operand value.

For simplicity's sake in our discussion so far we assumed that all data types are stored in the same register file. However, processor architectures *usually* provide *distinct register files* for FX- and FP-data. Then there are also distinct merged or architectural and rename register files for FX- and FP-data, as depicted for instance in Figures 3.10.1 and 3.10.3.

The situation is quite different when renaming takes place *within the ROB or the DRIS*. Since in both cases a *single mechanism* is maintained for preserving the sequential consistency of the execution, renamed FX- and FP-data must be hold in the same place. Because of this, the ROB or the DRIS ordinarily provides fields that are long enough to hold either FX- or FP-data.

Number of rename buffers

Table 3.5.1 gives an overview of how many rename buffers are provided by recent processors. Each of the three sections of the table covers a different implementation type. In the first section we list processors that use a *merged register file* for architectural and rename registers. Of course, in these processors we find quite a large number of additional registers which can be used for renaming. The number of rename buffers ranges from 8 in the Power1, up to 32 in the R10000 and even 38 in the 64-bit Sparc-processor PM1 (Sparc64).

Implementation of renaming		**Number of rename buffers**	
Processor type		**FX**	**FP**
Merged rename and arch. register file			
Power1	*(1990)*	-	8 (32 arch. +8 rename)
Power2	*(1993)*	-	22 (32 arch. +22 rename)
ES/9000	*(1992p)*	16 (16 arch. +16 ren.)	12 (4 arch. +12 rename)
PM1	*(1995)*	38 (78 arch. +38 ren.)	24 (32 arch. +24 rename)
R10000	*(1996)*	32 (32 arch. +32 ren.)	32 (32 arch. +32 rename)

Separate rename register file			
PowerPC 603	*(1993)*	n.a.	4
PowerPC 604	*(1994)*	12	8
PowerPC 620	*(1995)*	8	8
Renaming within the ROB			
Am29000 sup	*(1995)*	10	
K5	*(1995)*	16	
PentiumPro	*(1995)*	40	

Table 3.5.1: Number of rename buffers provided

Processors in the next section employ *separate rename registers*. They use between 4 and 12 dedicated registers for renaming. Examples of this type are the 603 - 620 models of the PowerPC family.

And finally, processors that implement *renaming within the ROB*, such as the Am 29000 superscalar, K5 or the PentiumPro, use between 16 and 40 rename buffers, the number of ROB-entries provided.

We stress again that in the first two alternatives, distinct register files are typically used for the renaming of FX- and FP-data. In contrast, when renaming is accomplished by means of an ROB, available ROB-entries are used for all types of data processed.

Basic mechanisms used for accessing rename buffers

Rename buffers need to be accessed for several purposes, such as to fetch operands, to update them or to deallocate them. At this time, we deal with how operands can be accessed from rename buffers. There are two entirely different *access mechanisms* available therefore, an *associative* and an *indexed access mechanism*, as Figure 3.5.9 shows.

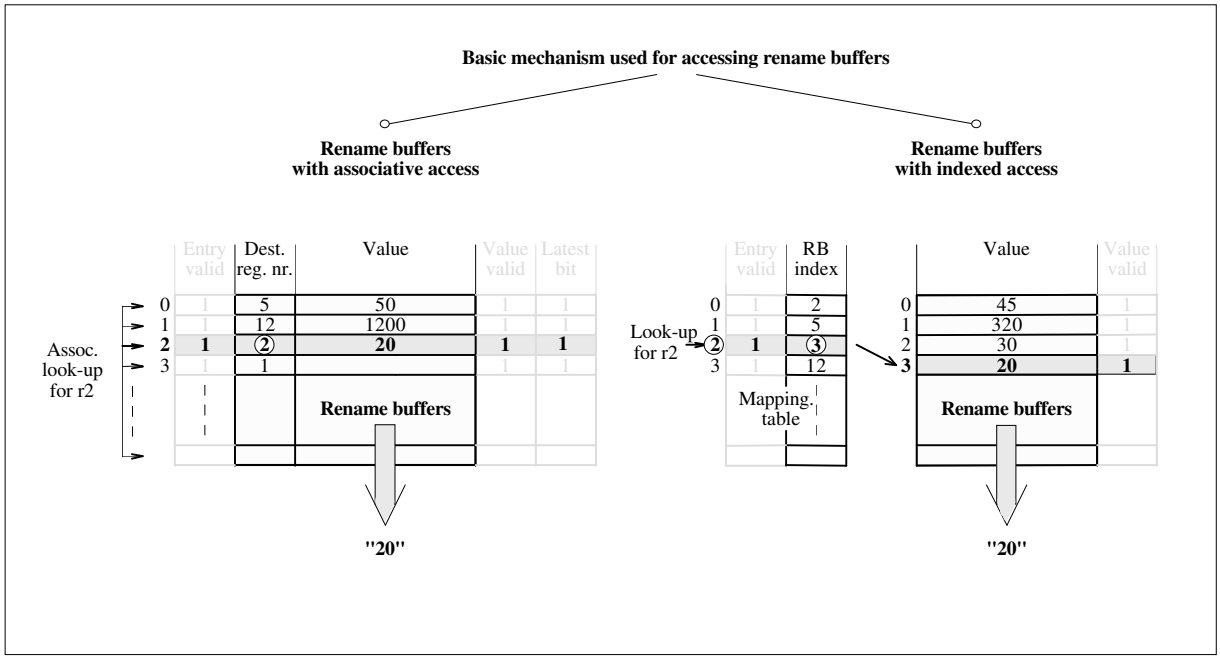


Figure 3.5.9: Basic mechanisms to access rename buffers

A typical layout of the **associative access mechanism** is depicted on the left of Figure 3.5.9. Rename buffers with associative access hold typically three kind of information; the destination register numbers, their values and necessary status information. In this scheme, when a register content is to be fetched, say that of r2, all entries are looked-up *associatively* to find the particular entry whose destination register field matches r2. In the figure, this is entry nr. 2. Thus, the corresponding value of "20" will be fetched.

In addition, *multiple instances of the same register* may exist; that is, two or more entries can belong to the same architectural register. Evidently, this complicates the operation considerably. How this situation can be handled is explained in the detailed demonstration of renaming in Section 3.5.8.

The other, fundamentally different access mechanism is the indexed access mechanism. In this case a **mapping table** is used to obtain the actual *index* of the rename buffer file, where the referenced register is. On the right of Figure 3.5.9, the mapping table contains for register r2 the index "3". Thus, the register value of "20" will be accessed from the corresponding entry in the rename buffer. We note that the mapping mechanism provides a unique index into the rename buffer file which always corresponds to the most recent instance (renaming) of the destination register concerned. Thus, multiple instances of the same register do not occur.

3.5.4 Operand fetch policies

In our introduction of shelving we already discussed possible operand fetch policies. As explained there, processors employ two possible policies. One policy is to fetch operands *during instruction issue* and store instructions along *with their operands* in the shelving buffers. The other policy is to shelve instructions *without their operands* and to delay *fetching* of the operands until the instructions are *dispatched* from the shelving buffers, as discussed in Section 3.4.4. In the case of renaming we have exactly the same possibilities. One possibility is to fetch referenced operands during renaming, the other one is to defer operand fetch until dispatching.

Here we stress again that shelving and renaming (as well as reordering) of the instructions are usually common features of recent superscalar processors. However, when both shelving and renaming are used, and in addition, operands are fetched during renaming, at first sight it seems that shelving and renaming are intermingled. We can still discuss shelving and renaming as separate concepts since renaming with operand fetch can conceptionally be subdivided into two steps: first, renaming without operand fetch, and second, usual operand fetch. Thus, we can describe shelving without taking into account renaming, as we did in Section 3.4.

3.5.5 Rename rate

The **rename rate** is the maximum number of renames per cycle that a processor is able to perform. In order to avoid bottlenecks the rename rate usually equals the issue rate. Recent processors can often issue up to four FX- or FP-instructions or any mix of the two. We expect then these processors to rename up to four instructions in each cycle. This may require a large number of ports at register files and mapping tables, if they are used. For instance, the R10000 performs renaming by using a merged file, which holds both the architectural and renamed register values. This processor fetches the operands during dispatching and accesses the merged register file via a mapping table. In order to be able to map four FX- or FP-instructions, the FX-mapping table has 12 read ports and 4 write ports, whereas the FP-table is equipped with 16 read and 4 write ports. This many ports are needed since FX-instructions can refer up to three and FP-instructions up to four operands.

Another example, worth looking at is the PM1, also called Sparc64. This 4-way superscalar processor implements renaming in the same way as the R10000 with one difference. That is, PM1 fetches operands during renaming. In this case, both the FX-mapping table and the merged register file have 10 read and 4 write ports while its FP counterpart has 6 read and 3 write ports. According to Asato & al. (1995), this 14-port 116 word 64-bit merged register file has a remarkably high complexity of 371K transistors. In

comparison, the whole Intel 8086 processor has a complexity of about 30K transistors, and even the i386 has no more than about 275 K transistors (Crawford 1990).

3.5.6 Further issues

As we know, shelving makes it unnecessary to check for data- and control dependencies as well as for busy EUs during issue. Nevertheless, during renaming we do need to *check for data dependencies among the instructions issued in the same group*. For instance, the PowerPC 620 or the R10000 check whether there are data dependencies among the four instructions issued in the same group, and if dependencies are encountered, the rename logic has to be modified accordingly. For instance, let us consider true data dependencies (RAW dependencies) between two instructions, as in the following example:

```
mul   r2, ..., ...
ad    ..., r2, ...
```

Here, the ad instruction uses the result of the mul instruction as a source operand. Furthermore, let us assume that the mul instruction is renamed to

```
mul   p12, ..., ...
```

Then, because of the true data dependency, r2 in the ad instruction also has to be renamed to p12 rather than to the register number corresponding to r2 before renaming of the "mul" instruction.

Similarly, if there are WAW dependencies among the instructions issued in the same group, as for instance, between the instructions

```
mul r2, ..., ...
ad        r2, ..., ...,
```

special care has to be taken during renaming. This means that despite simultaneous renaming of all instructions in the same issue group in this case, different rename buffers have to be allocated to the destination registers of the "mul" and the "ad" instruction.

3.5.7 Most frequently used basic implementation alternatives of renaming

In the preceding section we acquainted the reader with the design space of renaming. Here, we present the most frequently used implementation alternatives. By taking into account all four essential aspects (scope of register renaming, the layout of shelving buffers, the operand fetch policy as well as the rename rate) and possible alternatives for each of them, we arrive at a large number of feasible *implementation alternatives.*

In our discussion of the basic implementation alternatives we will take only two aspects, the *layout of rename buffers* and the *operand fetch policy*, into account. We ignore the scope of shelving, since recently most processors implement full shelving, and also the issue rate component, since it is only of a quantitative character. As far as the layout of rename buffers is concerned, here we will take into account the type of rename buffers used and their access mechanism. We ignore also the number of rename buffers for obvious reasons. Thus, for the discussion of the basic implementation alternatives, we consider altogether three aspects as vital: the type and access mechanism of rename buffers as well as the operand fetch policy.

Figure 3.5.10 depicts the corresponding *reduced design space* of renaming. Each of the leaves represent a basic implementation alternative. In this figure we also show which basic alternative is employed in relevant superscalar processors, as well as some references to when certain basic alternatives were proposed and by when.

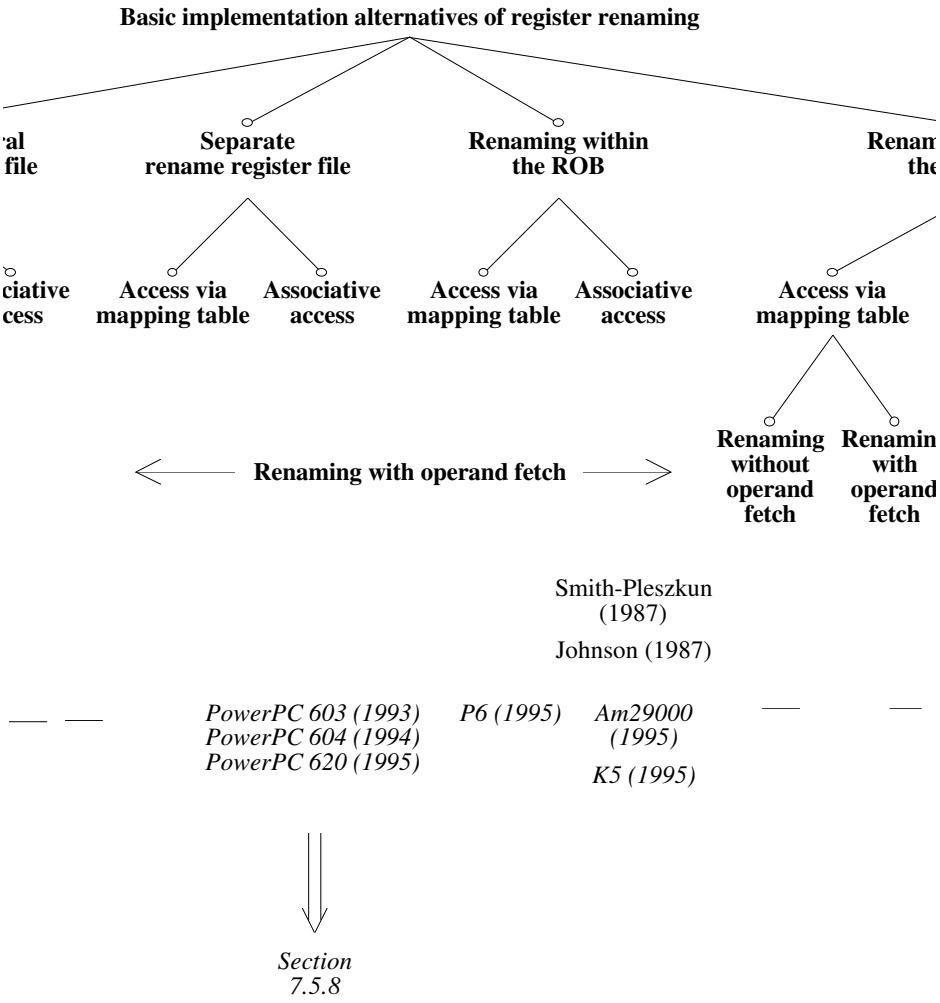


Figure 3.5.10: Basic implementation alternatives of register renaming

As Figure 3.5.10 shows, there are *six basic alternatives* of renaming which are currently used in processors. In the following we overview these. A number of processors employ a *merged register file* for both architectural and rename registers, as was proposed by Keller a long time ago (1975). Since a merged register file requires a large number of registers, all processors which have opted for this implementation alternative use a *mapping table* for accessing the merged file. Most of them postpone operand *fetching* until *dispatching,* such as the Power1, Power2, ES 9000 and the PentiumPro. In contrast, the PM1 (Sparc64) *fetches operands during renaming.*

The next important alternative employs a *separate rename register file*, accesses it *associatively* and *fetches operands during renaming*. The PowerPC models 603, 604 and 620 follow this method of implementation.

The *use of the ROB for renaming* seems to be an appealing alternative, since for most issued instructions both a new ROB-entry and a new rename buffer have to be allocated. However, while a new ROB entry has to be allocated to each issued instruction, not every instruction has to be renamed. Renaming is needed only for those instructions which write into architectural registers. Thus, when buffer space is provided along with each and every ROB entry, a part of the ROB-space is wasted. This is a slight drawback of this alternative.

The principle of renaming within the ROB was first described by Smith and Pleszkun as early as 1985 and later also by Johnson (1991). Examples of processors utilizing this alternative are the Am 29000 superscalar and the K5. These processors access the ROB *associatively*. It is worth noting here that the cited author was the principal architect of both Am processors. Another recent processor which has chosen this implementation alternative, while accessing the ROB via a *mapping table* and *fetching operands during issue*, is the PentiumPro.

The final basic alternative to be mentioned implements *renaming within the DRIS*, accesses it in an *associative way* and *fetches operands along with dispatching,* as proposed by Sohi and Vajapeyam (1987). This variation was selected for renaming by the architects of the Lightning processor, which never reached the market. Probably the high complexity requirement of its implementation turned out to be prohibitive for manufacturing at that time.

3.5.8 Detailed example of renaming

Next we describe in depth how renaming is accomplished in one of the basic alternatives, which features a *separate rename register file,* with an *associative access* and *operand fetching during renaming.* Processors employing this basic alternative are the PowerPC 603 - PowerPC 620.

In this demonstration we show how the following short instruction sequence

```
mul     r2, r0, r1
ad      r3, r1, r2
sub     r2, r0, r1
```

is renamed.

In our example, the "mul" instruction is WAW-dependent on the "sub", since r2 is used repeatedly as a destination register, first in the "mul" and then in the "sub" instruction. We also note that the "ad" instruction is true data dependent, that is RAW-dependent on the preceding "mul" instruction, since it refers to r2 as a source operand.

For our discussion let us assume that the rename registers have five characteristic entries, as illustrated in Figure 3.5.11.a. These entries are the Entry valid bit, the Destination register number, the Value of the renamed destination register, the Value valid bit and the Latest entry bit. With the exception of the last, these fields are self-explanatory.

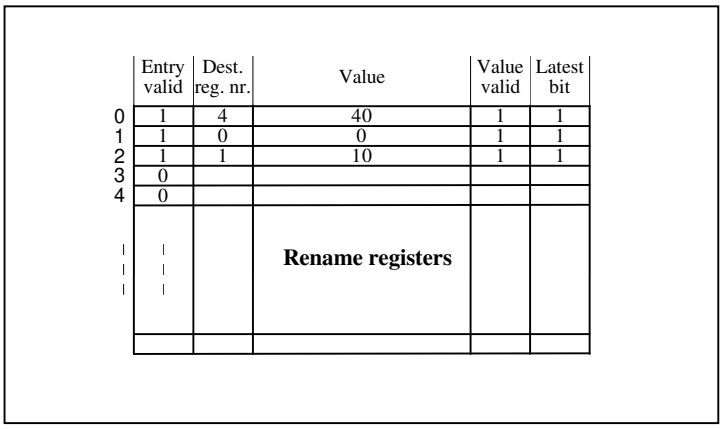


Figure 3.5.11.a: Structure of the rename buffers and its supposed initial contents

The Latest entry bit is necessary since any particular destination register (for instance r2 in our example) may have been renamed repeately and thus, multiple rename registers may belong to the same architectural register, as illustrated in Figure 3.5.11.b.

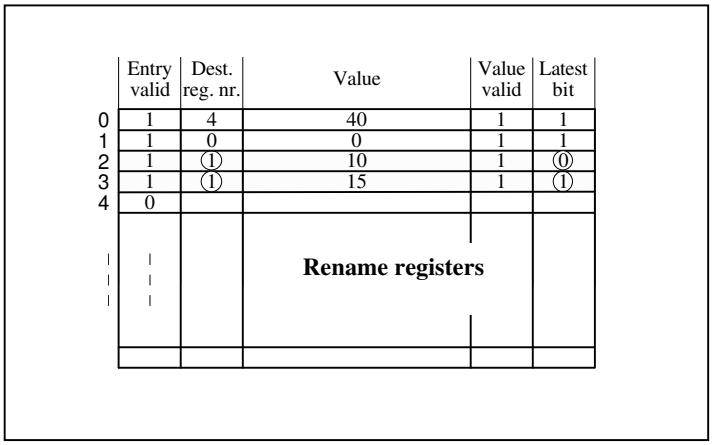


Figure 3.5.11.b: Multiple subsequent renames belonging to the same architectural register (r1)

In this case, architectural register r1 has two renames. Based on the information provided in the Latest entry bit, subsequent instructions which ask for register r1 as a source operand, will access only its latest value, which is 15, rather than the previous value of 10. Thus, in general, when source operands are fetched during renaming, the most recent (latest) renames of the registers are to be accessed. The Latest bit is needed therefore, to mark the most recent rename of an architectural register.

Figure 3.5.11.a also shows the supposed initial content of the rename registers. In the figure there are three valid entries, each containing a valid latest value of the architectural registers 4, 0 and 1, respectively. These registers have the content of "40", "0" and "10", respectively. Entries 3, 4 and so on are free.

Renaming consist of the following subtasks:

a. Allocation of a free rename register to a destination register.
b. Accessing valid register values.
c. Accessing a register value that is not yet available
d. Reallocation of a destination register
e. Updating a particular rename buffer with a computed result
f. Deallocation of a rename buffer that is no longer needed.

In our demonstration example we assume that the given instructions are renamed one after the other. Obviously, superscalar processors perform multiple renames per cycle, as discussed in Section 3.5.3. Nevertheless, for better understanding we describe renaming of the example instructions as if it were performed in sequence.

a. Allocation of a free rename buffer to a destination register

Each time when an instruction is issued which references a destination register, a new rename register needs to be allocated to the destination register in concern. This allocation is performed by looking for and reserving a free entry, and properly initialising its fields. Usually, rename registers are handled as circular buffers. Then a head pointer designates the first free entry whereas a tail pointer indicates say, the last one (see Figure 3.5.11.c).

When a new entry is required, the entry indicated by the Head pointer (in our case entry 3) is selected for use and the pointer is stepped. On the other hand, if an entry is no longer needed, this entry will be deallocated, by updating the Tail pointer.

During allocation of a new rename register, the following updates have to be accomplished: setting the Entry valid and Latest entry bits, copying the destination register number into the corresponding field and resetting the Value valid bit. Figure 3.5.11.c demonstrates this in connection with the renaming of the destination register of the "mul"

instruction (r2). The upper part of this figure displays the initial content of the rename registers in accordance with Figure 3.5.11.a.

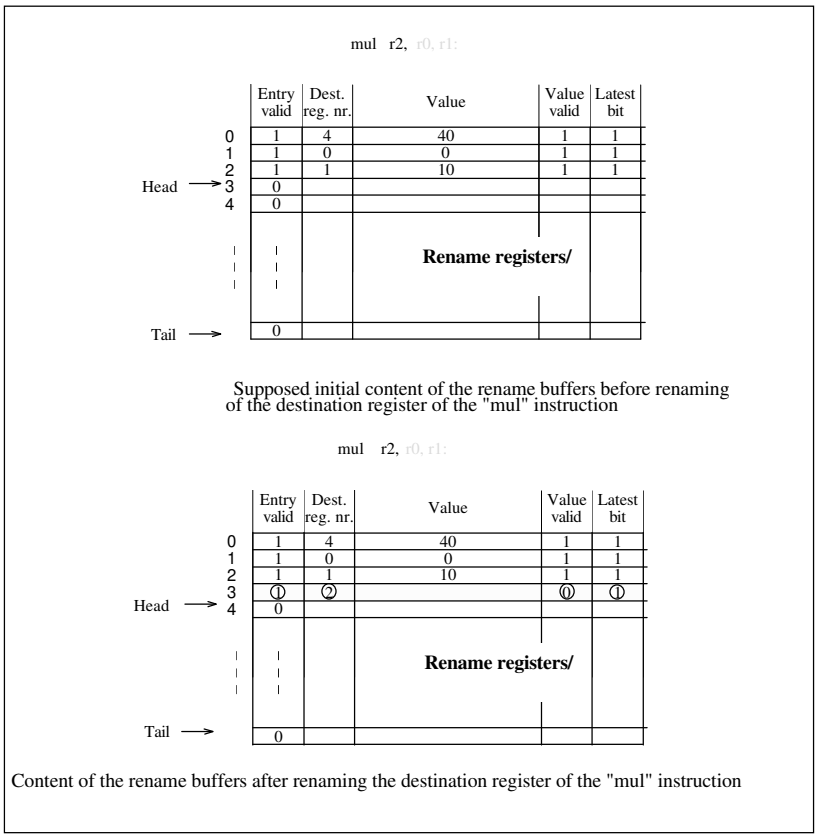


Figure 3.5.11.c: Allocation of a new rename buffer to destination register r2

The lower part of the figure shows what happens after allocation. As depicted, the next free entry, that is entry nr. 3, will be allocated to register 2 and the status fields of the allocated entry are initialised as discussed before. Here, we note that for simplification in the subsequent figures, we do not indicate the Head and Tail pointers.

b. Accessing valid register values

Let us now consider, how valid operand values are fetched from the rename buffers. This is the case when in our example the source operands of the "mul" instruction, r0 and r1, are fetched. Remember here that in the basic variant considered rename registers are accessed associatively. Then for accessing a referenced operand value, all rename registers have to be looked up associatively for the latest value of the specified register number, i.e. for r0 and r1 (see Figure 3.5.11.d). If there is a matching entry and in this entry both the Value valid and the Latest bits are set, the latest valid register value has been found. It is then fetched and appropriately marked as "valid", as illustrated below, for the referenced register r0 and r1.

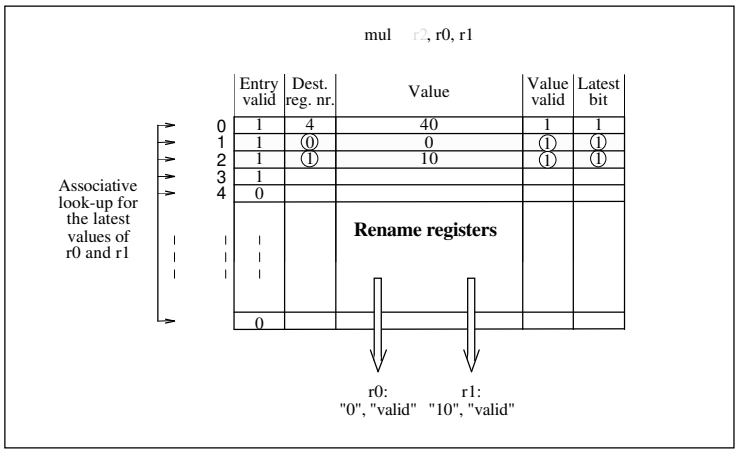


Figure 3.5.11.d: Accessing available register values

Of course, both renaming of the destination register and fetching the specified source operands are accomplished in the same cycle.

c. Accessing a register value that is not yet available

When an accessed register value is not valid, since it has not yet been computed, the register value can not yet be fetched. Then instead, the entry identifier is fetched with an appropriate marking.

Figure 3.5.11.c demonstrates this by showing what happens when the second instruction

```
ad    r3, r1, r2
```

is renamed with simultaneous fetching of the source operands.

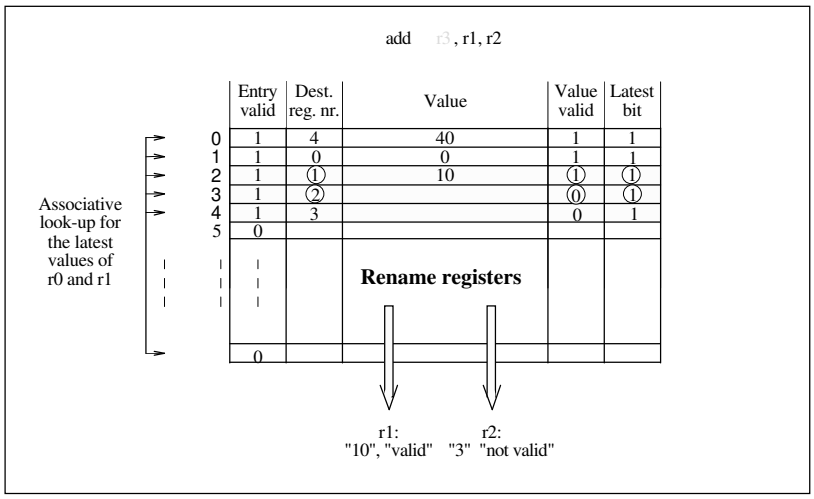


Figure 3.5.11.e: Accessing a register value that is not yet available

The first subtask is to allocate a new rename register to r3, this is entry nr. 4. This allocation is done in the same way as described earlier. In addition, the contents of r1 and r2 are to be fetched. Since, the latest rename of r1 holds a valid value, it can be fetched, as discussed before. However, r2 has not yet been computed. Therefore, in place of the requested value, the index of the associated rename buffer (which in our example is 3) is forwarded, as illustrated.

d. Reallocation of a destination register

Obviously, during program execution the same architectural register may be used repeatedly to hold results. Therefore, it can happen that multiple valid rename entries belong to the same register. For instance, in Figure 3.5.11.f we show what happens when the next instruction , that is the "sub" instruction is issued before the "mul" instruction has been completed.

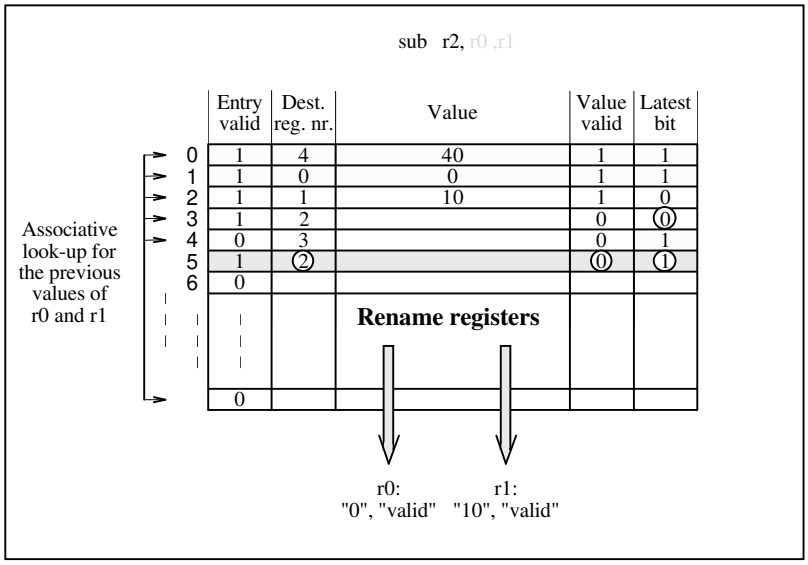


Figure 3.5.11.f: Reallocation of r2

As the figure demonstrates, in this case a second entry has to be allocated to register r2. For this reason a new free entry is selected (entry nr. 5) and the the corresponding Value valid and Latest bits are set appropriately. At the same time, the latest bit of the previous r2 entry (entry nr. 3) has to be reset. Thus, from now on all subsequently issued instructions will access the latest r2 value from entry nr. 5 instead of the previous one. This remains true until a subsequent instruction is issued whose destination register is again r2. Then a new rename becomes the latest one and so on. We note that, in our example, resetting of the Latest bit of the previously allocated rename register requires an associative search for the latest r2 allocation.

e. Updating a particular rename buffer with a computed result

If an instruction is finished, the destination register in concern has to be updated. So subsequently issued instructions will be able to access the generated result. Updating is based on the result identifier, which is supposedly the index of the rename register which is allocated to the destination register in question.

In our example, let us assume that the "mul" instruction has been finished. Then the result ("0") is governed by the identifier "3", which means that the result has to be written in rename register 3.

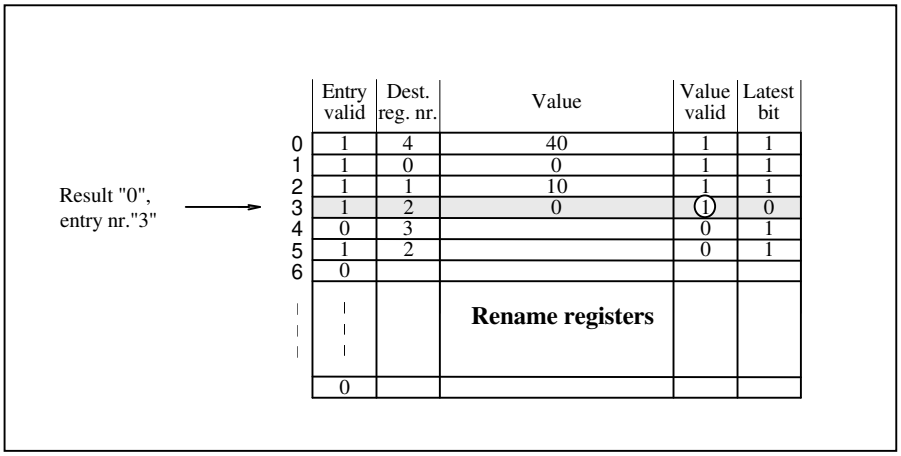


Figure 3.5.11.g: Updating of the rename buffers of register 2 with the result, "0"

Entry nr. 3 can be directly accessed and updated. Then the Value valid bit is set. This indicates that the value of register 2 became available. Here we note that, this value is not the latest one and it is no longer needed in the rename register file. Nevertheless, it can be needed to update failing operand values in the reservation station.

f. Deallocation of a rename buffer that is no longer needed

The final logical step in the renaming process is the deallocation of an entry that is no longer needed, as demonstrated in the following figure. Deallocation is usually performed in conjunction with the ROB, as detailed in Section 3.7.3. The ROB assures sequentially consistent program execution through allowing instructions to retire only in strict program sequence. When an instruction retires, the corresponding rename buffer entry can be deallocated. This is carried out by assuming that the index of a not more needed rename buffer entry is provided by the ROB.

In our example, let us assume that the instruction, whose result is in entry 0, retires. Thus, the ROB requests the deallocation of the associated rename register. Deallocation is

accomplished simply by resetting the corresponding Entry valid bit, as Figure 3.5.11.h. demonstrates. Thus, this entry again becomes available for allocation.

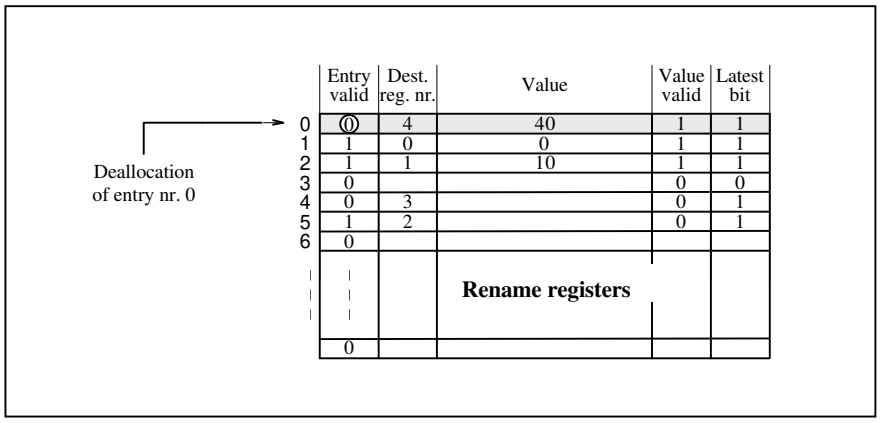


Figure 3.5.11.h: Deallocation of the rename buffer nr.3

Here we note that the ROB retires instructions in strict program order, so deallocation will be accomplished in the sequence of increasing indices.

3.6 Parallel execution

When instructions are executed in parallel, they will generally be *finished in out-of-program-order*. Here, it does not matter whether instructions are issued or dispatched in-order or out-of-order, or whether shelving is used or not. The point is that unequal execution times force instructions to finish out-of-order, even if they are issued (and dispatched) in-order. Then short, "younger" instructions may be finished earlier than long, "older" ones. Thus, superscalar execution gives rise to an out-of-order finishing of instructions.

Here, we make a distinction between the terms *"to finish"*, *"to complete"* and *"to retire"* an instruction. We use the term **"to finish"** an instruction if we want to indicate that the required operation of the instruction is accomplished, except writing back the result into the architectural register or memory location specified and/or updating the status bits. In contrast, we employ the term **"to complete"** an instruction if we want to refer to the last action of instruction execution, which is the writing back of the result into the referenced architectural register. Finally, in connection with the ROB "instead of the term "to complete" we say "**to retire**" since in this case two tasks have to be performed, to write back the result and to delete the completed instruction from the last ROB-entry. Nevertheless, we use both terms perfectly in the same sense. We note furthermore that there is no generally accepted use of these terms and the terminology differs from processor to processor. Thus, care must be taken in the interpretation of the publications.

Under special conditions *out-of-order finishing* of the instructions can be *avoided* in spite of multiple EUs. The conditions therefore are as follows: instructions must be issued in-order and all parallel operating EUs must have equal execution times. Practically, these conditions may be fullfilled by using a *dual pipeline* and *lock-stepping* them, that is lengthening the shorter pipeline by introducing unused extra cycles ("bubbles") into it, as described in Section 5.3.2. Evidently, these prerequisites are overly restrictive and impede performance. Therefore, there are only a few superscalar processors which avoid out-of-order completion in this way. Examples are the MC 68060 and the Pentium, both of which employ lock-stepped dual pipelines.

3.7 Preserving the sequential consistency of instruction execution

3.7.1 Sequential consistency models

As discussed above, in superscalar processors, or in a more general sense in processors with multiple parallel operating EUs, instructions *finish* in general in an *out-of-order fashion.* Nevertheless, overall instruction execution should mimic sequential execution, that is, it should preserve *sequential consistency*. Although, as stated before, the problem of preserving the sequential consistency relates to a broader class of processors than superscalar ones, for sake of consistency with the recent chapter, in the following we confine our discussion only to superscalar processors.

Sequential consistency of instruction execution relates to two aspects, first, to the *order of how instructions are completed;* and second, to *the order of how memory is accessed* due to load and store instructions or memory references of other instructions, as indicated in Figure 3.7.1.

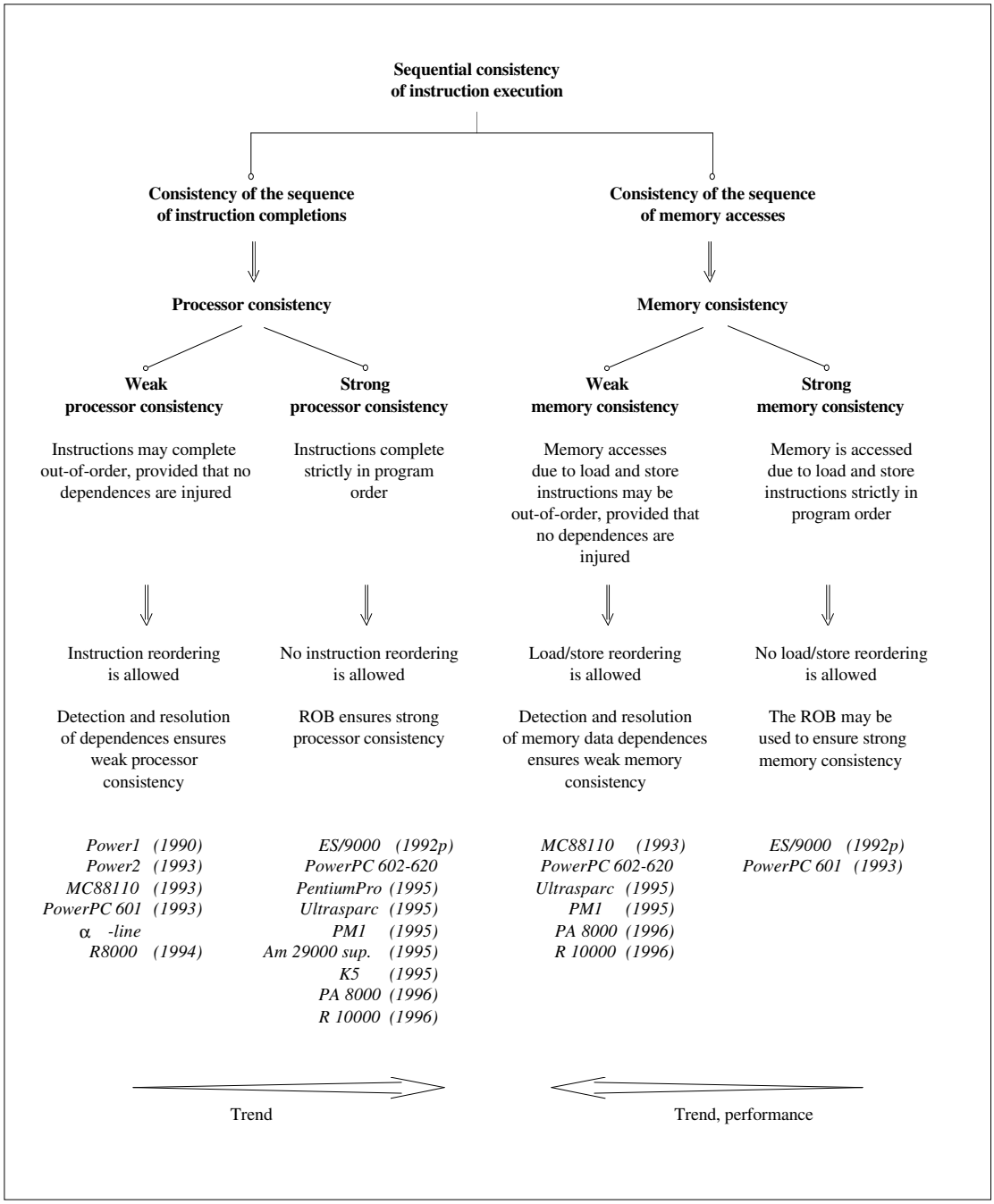


Figure 3.7.1: Interpretation of the concept of sequential consistency of instruction execution

Concerning the first aspect we are interested in whether instructions during a superscalar execution complete in the same order as in a sequential processor. Here, we use the term *complete* as explained in the previous section. We refer to the *consistency of completions with the sequential instruction execution* briefly as **processor consistency.**

As far as processor consistency is concerned, superscalar processors preserve either a *weak* or a *strong consistency*. A **weak processor consistency** means that instructions *may complete out-of-order*, provided that *no data dependencies are sacrified*. In this case, instructions may be reordered by the processor, only if no dependencies are injured. In

order to achieve this, data dependencies have to be detected and appropriately resolved during superscalar execution. Early superscalar processors usually provided a weak processor consistency. Examples are the Power1, Power2, PowerPC 601 and many others, as indicated in Figure 3.7.1.

In the case of **strong processor consistency**, instructions are forced by a superscalar processor to *complete in strict program order*. Usually, this is achieved by employing a reorder buffer (ROB). The ROB is a very practical tool as it can also be used to implement renaming or shelving as well, as emphasized in the related sections. It is little wonder that ROBs are now widely used in superscalar processors. Most recent processors guarantee strong processor consistency, since it is easy to implement, like the ES 9000, all models of the PowerPC family except the PowerPC 601 and a number of further recent processors.

The other aspect of superscalar instruction execution is whether memory accesses are performed in the same order as in a sequential processor or not. This aspect is termed **memory consistency**. Here, again we can distinguish between *weak* and *strong* memory consistency.

We say that **memory consistency is weak** if *memory accesses may be out-of-order* compared to a strict sequential program execution.

However, regardless of out-of-order memory accesses, *data dependencies must still not be injured*. In other words, weak consistency allows *load/store reordering* provided only that dependencies and in particular memory data dependencies are detected and resolved.

As we shall discuss in the following section, weak memory consistency is a means to increase processor performance. Therefore most up-to-date superscalar processors rely on it. Some examples are the MC 88110, the PowerPC line of processors except the PowerPC 601, the UltraSparc, PM1, PA 8000 and the R10000.

The other alternative is **strong memory consistency**. In this case *memory accesses* remain during superscalar execution *strictly in program order*. Strong memory consistency forbids any load/store reordering. An example of a processor which maintains strong memory consistency is the PowerPC 601.

Until now we have discussed processor and memory consistency separately. Evidently, superscalar processors maintain either a weak or a strong processor consistency and at the same time either a weak or a strong memory consistency. **The sequential consistency model of a processor** integrates both aspects. It specifies which kind of consistency is maintained *by the processor* and *at the same time by the memory*. Thus, by taking into account both aspects of processor and memory consistency, we arrive at four possible sequential consistency models of instruction execution (se Figure 3.7.2). These are the **WW**, **WS**, **SW** and **SS** consistency models. Here, the first character of the model

designation refers to the type of the processor consistency (**W**eak/**S**trong) and the second character indicates the sort of the memory consistency maintained (**W**eak/**S**trong).

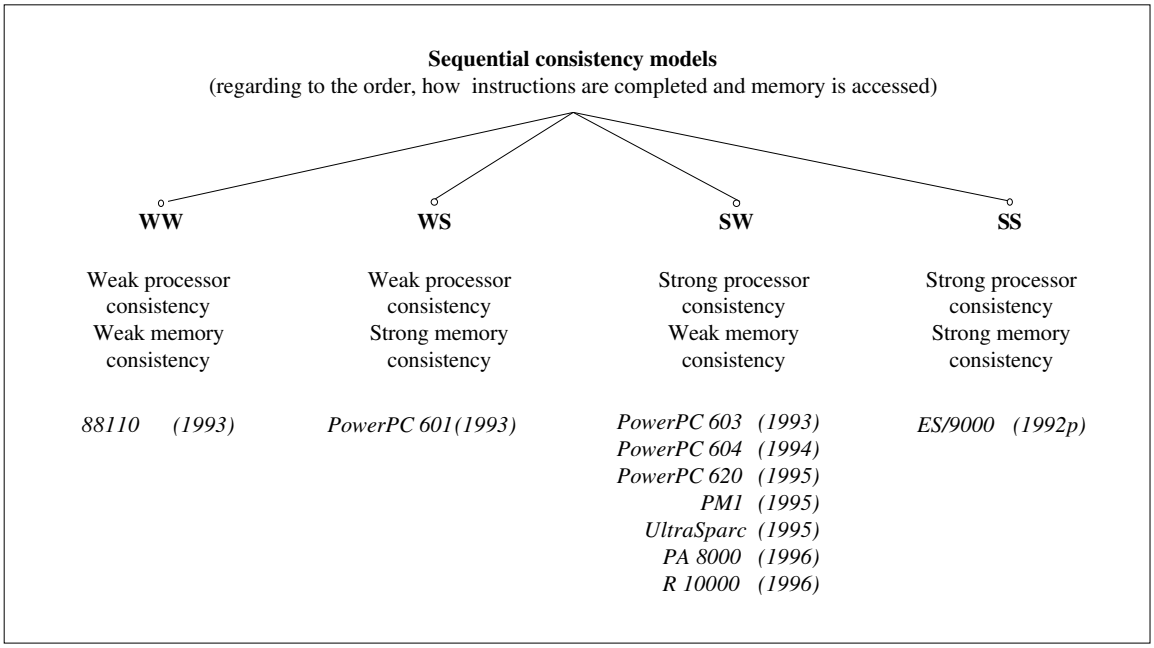


Figure 3.7.2: Sequential consistency models of instruction execution

As indicated before, *strong processor consistency* and *weak memory consistency* have advantages. Consequently, recent processors tend to maintain the **SW consistency model**. Examples of processors supporting this model are the PowerPC 603-PowerPC 620, PM1, UltraSparc, PA 8000 and the R10000. The other models are typically maintained only in earlier superscalar processors. For instance, the WW model is used in the MC 88110, the WS model in the PowerPC 601 and the SS model in the ES/9000. This concludes our overview of the consistency of instruction execution in superscalar processors. In the following section we will discuss some aspects of the weak memory consistency model.

3.7.2 Load/store reordering

Load and store instructions are specific in that they involve actions affecting both the processor and the memory. While executing, both loads and stores must first wait for their addresses to be computed by an ALU or address unit. Then, loads can access the data cache to fetch the requested memory data. Retrieved data are then made available in a register. This is when a load instruction is finished. The load is then completed usually by writing the fetched data into the specified architectural register.

Stores have quite a different execution pattern. After having received their generated addresses, stores have to wait for their operands to be available. Unlike other instructions, when operands become available, a store is considered to be finished. Now, let us assume

having an ROB. Then, when the ROB indicates that the store comes next in the sequential execution, the memory address and data to be stored are forwarded to the cache and a cache store operation is initiated.

A processor which supports *weak memory consistency* allows the reordering of memory accesses to each other. This is advantageous for at least three reasons,

- it permits load/store bypassing,
- it makes feasible speculative loads or stores, and
- it allows cache misses to be hidden.

In the following section we discuss these points on the basis of Figure 3.7.3.

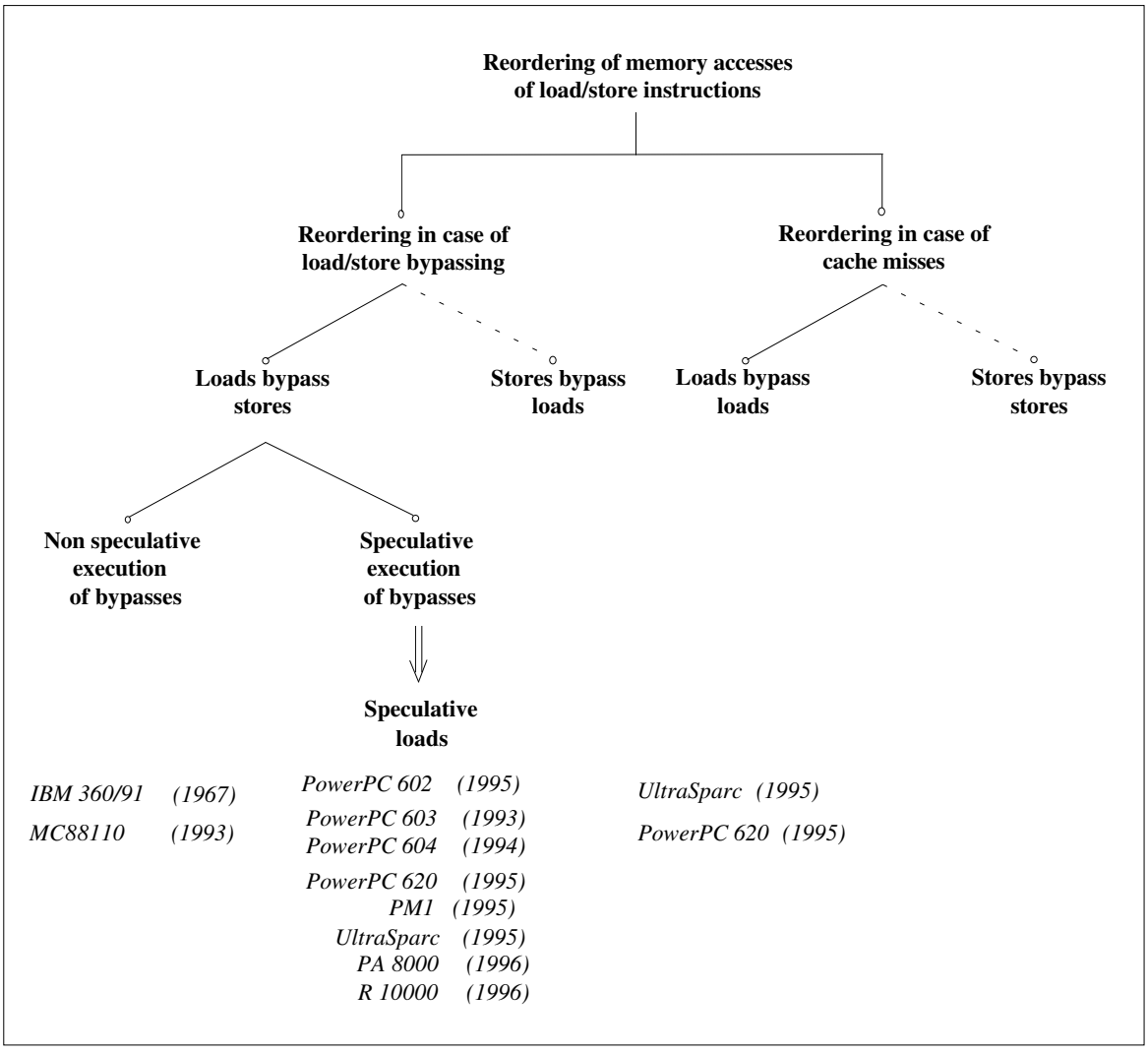


Figure 3.7.3: Reordering of memory accesses of load/store instructions

Load/store bypassing means that either loads can bypass pending stores or vice versa, provided that no memory data dependencies are injured. As Figure 3.7.3 indicates, a

number of recent processors allow loads to bypass stores (either non speculatively or speculatively, as will be explained later) but not vice versa.

Permitting loads to bypass stores has the advantage among others of allowing the runtime overlapping of tight loops. The overlapping is achieved by allowing loads at the beginning of an iteration to access memory without having to wait until stores at the end of the previous iteration are completed. Notice that this runtime overlapping of cycles is comparable with software pipelining (see Section 9.3.3).

Evidently, in order to avoid fetching a false data value, a load can bypass pending stores only if *none of the preceding stores has the same target address* as the load has. In order to check this requirement, the address of the load has to be compared against the addresses of all pending stores. However, when the load address is compared with all addresses of pending stores, it may happen that certain addresses of pending stores are not yet available. Then no decision can yet be made as to whether the load is dependent on the pending stores or not. There are two possible ways to handle this situation. The simpler scheme is to delay the permission for a bypass of the load until all pending store addresses are computed and a decision can be reached. We term this the **non-speculative execution of bypasses**. Earlier processors that permit loads to bypass stores do so in a non-speculative fashion, like the pioneering IBM 360/91 (as early as 1967) and the MC 88110.

The more advanced handling of this situation is to let loads bypass stores speculatively. This is what is called **speculative loads**. Speculative loads avoid delaying memory accesses until all required addresses have been computed and an address equivalence can be ruled out. In this case a memory access will be started *in spite of unresolved address checks.* For a vast majority of bypassed loads the addresses of subsequently computed preceding stores do not coincide with the load address and this speculative manner is justified. The correctness of speculative loads has to be checked in any case and if needed the speculative load must be undone. This is performed as follows. When store addresses have been computed, they are compared against all addresses of younger loads. If a hit is found, the corresponding speculative load has fetched a data value that is wrong. At this point the load instruction and all subsequent instructions are cancelled and the load is reexecuted.

We note that speculative loads are quite similar to speculative branches. Speculative branches (see Section 9.4.4) react in exactly the same way to the situation of unresolved condition checks as do speculative loads to unresolved dependency checks. A number of recent processors allow speculative execution of loads. Examples are the PowerPC 602 - PowerPC 620, the UltraSparc, PM1, PA 8000 and the R10000.

It should be noted that the address checks are usually carried out by writing the computed target addresses of loads and stores into the ROB (or DRIS) and performing the address comparisons there. To reduce the complexity of the required circuitry, the address check is often restricted to a part of the full effective address. For instance, the PowerPC

604 or the PowerPC 620 store and use only the low order 12 bits of the effective address for address checks.

Cache misses are another source of performance impediment which can be reduced by load/store reordering. Usually, a cache miss causes a blockage of all subsequent operations of the same type. In other words, a load miss blocks subsequent loads and a store miss impedes subsequent stores. The performance degradation of load misses can be reduced if loads are allowed to bypass pending loads, as has been implemented in the UltraSparc, the PowerPC 620 and the R10000. For instance, the PowerPC 620 is able to service loads in spite of up to three pending loads. Here, the pending loads are stored in a Load miss register (one entry) and in the three load/store reservation stations (three entries).

3.7.3 The reorder buffer (ROB)

The ROB was first described by Smith and Pleszkun in 1988. Originally, they conceived the ROB to solve the precise interrupt problem. Today, a **ROB** is understood as a tool which assures sequential consistency of execution in case of multiple parallel operating EUs. Basically, the ROB is a *circular buffer* with a head and a tail pointer, as depicted in Figure 3.7.4. The *head pointer* indicates the location of the next free entry. Instructions are written into the ROB in strict program order. As instructions are issued, a new entry is allocated to each in sequence. Each entry indicates the status of the corresponding instruction, say whether the instruction is issued (i), is in execution (x) or is already finished (f). The *tail pointer* marks the instruction which will retire, that is leave the ROB next. The ROB operates such that an instruction is allowed to retire only if it has already been finished and all previous instructions are already retired. This mechanism ensures a strict in-order retirement of the instructions. Sequential consistency is preserved in that only instructions which are retiring are permitted to complete, i.e. to update the program state by writing this result into the referenced architectural register(s) or memory.

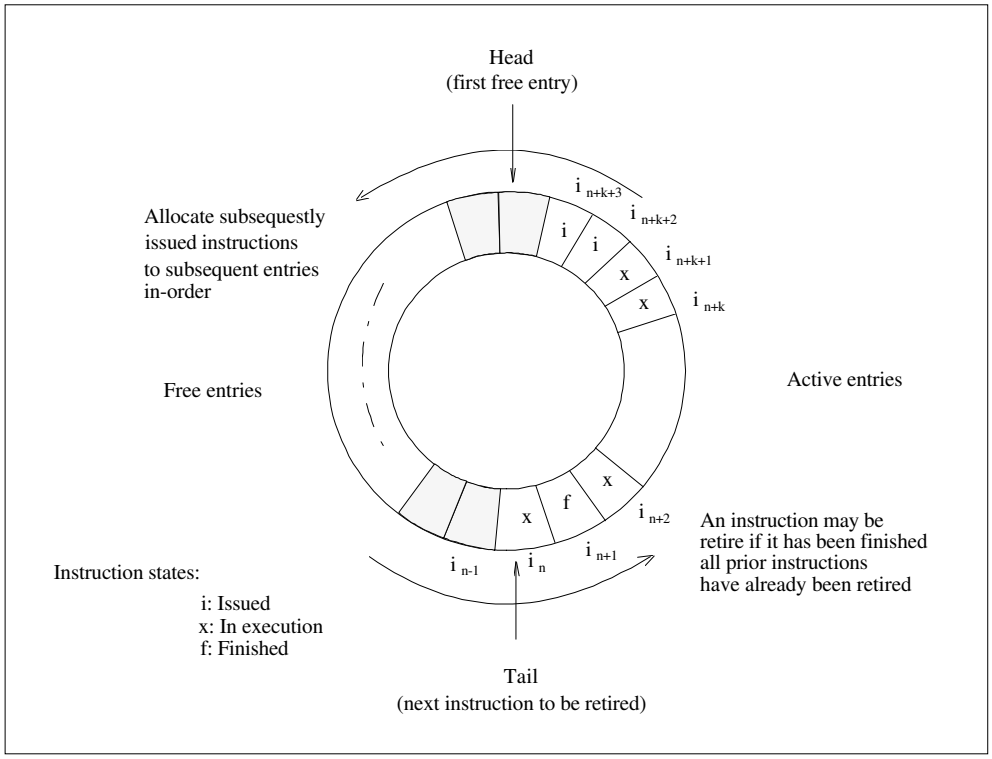


Figure 3.7.4: Principle of the ROB

Here, we note that an ROB can effectively *support both speculative execution* and *interrupt handling*.

As we know, in *speculative execution* the processor keeps on executing instructions in spite of an unresolved condition, such as an unresolved conditional branch or memory address check. Later on, when the condition is resolved, it becomes clear whether the speculatively executed instructions can be acknowledged as part of the regular execution or whether they have to be ignored and re-executed along the right path. An ROB easily supports speculative execution. All we have to assume is that each ROB entry is extended with a speculative status field, indicating whether the corresponding instruction is executed speculatively or not. In addition, finished instructions are not authorised to retire when they are in a speculative state. The whole operation is then as follows. Speculatively executed instructions are marked as such and are not eligible for retirement. Later when the related condition is resolved, it turns out whether speculative execution can be recognised as ordinary execution or whether it has to be purged. When speculative execution is acknowledged, the corresponding markings must be modified. For correctly speculated instructions, in-order completion is maintained as described before. If, in contrast, the speculative execution happens to be incorrect, all corresponding instructions which are marked as being speculative have to be cancelled from the ROB and instruction execution has to be continued with the correct instructions.

An ROB also *assists interrupt handling* in a natural way. Consider here that interrupts generated in connection with instruction execution can easily be made precise, by accepting

interrupt requests only when the related instruction becomes the next to retire, as discussed in the next section.

ROBs were introduced primarily in connection with shelving and register renaming. As visual demonstration of this, Figure 3.7.5 gives an overview of the introduction of ROBs compared with the introduction of renaming. It can easily be checked that, with a few exceptions, renaming and ROBs (as well as shelving) appeared at the same time in superscalar processors.

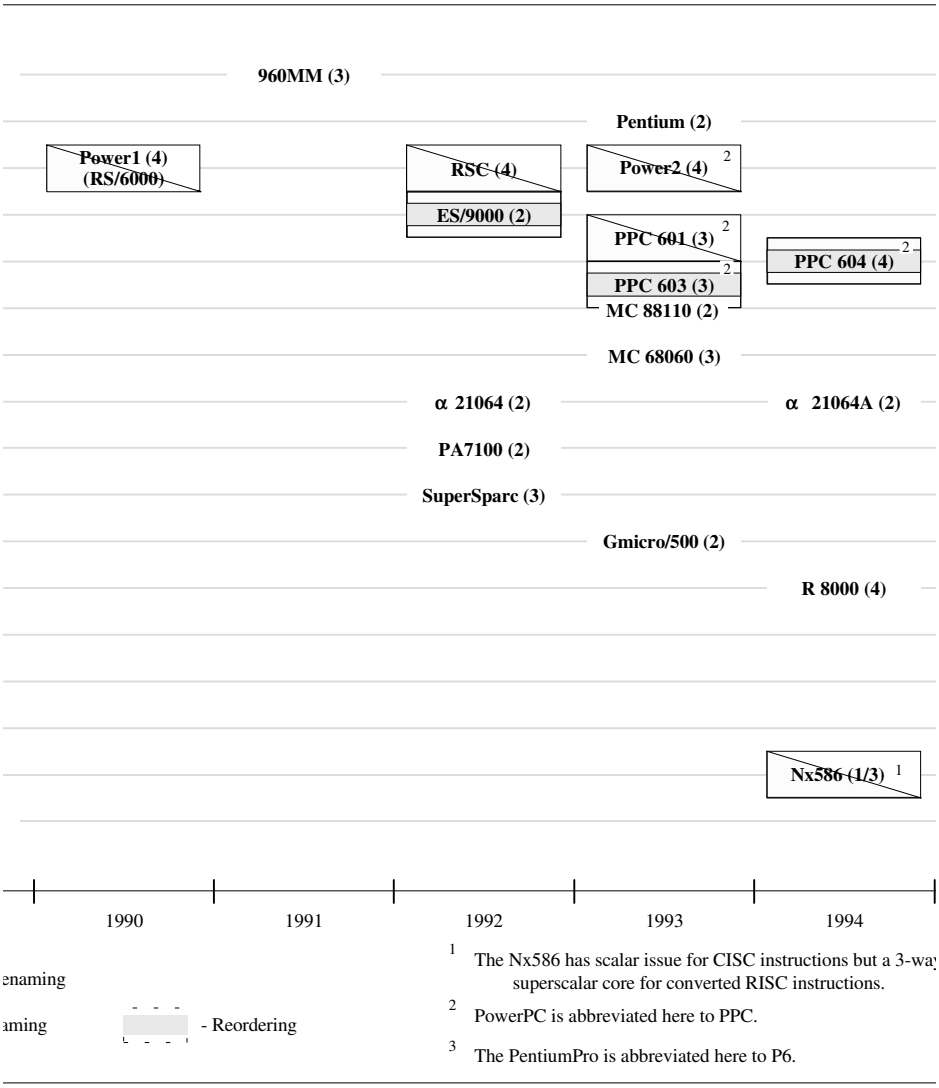


Figure 3.7.5: Introduction of ROBs in commercial superscalar processors

Next, we will discuss the *design space of ROBs*. Here, we are concerned with their *basic layout* and with some *implementation details,* such as ROB-size and retire rate, as designated in Figure 3.7.6.

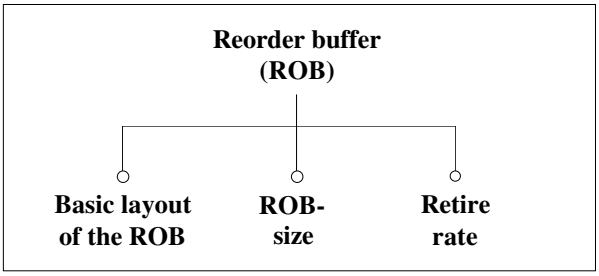


Figure 3.7.6: Design space of ROBs

Before discussing possible basic layouts of ROBs we introduce the term **"reordering of instructions"**. We use this term to designate the basic function of an ROB, that is to provide sequential consistency by means of ensuring strict in-order instruction completion. Thus, the basic function of ROBs is reordering.

However, as mentioned before, ROBs may be used for more tasks than reordering alone. As Figure 3.7.7 shows, ROBs can be employed in three different ways. The simplest application of an ROB is to *provide reordering alone*, as discussed before. For instance, the processors ES/9000, PowerPC 603, PowerPC 604, PowerPC 620, PM1 (Sparc64) and R10000 employ ROBs for reordering alone.

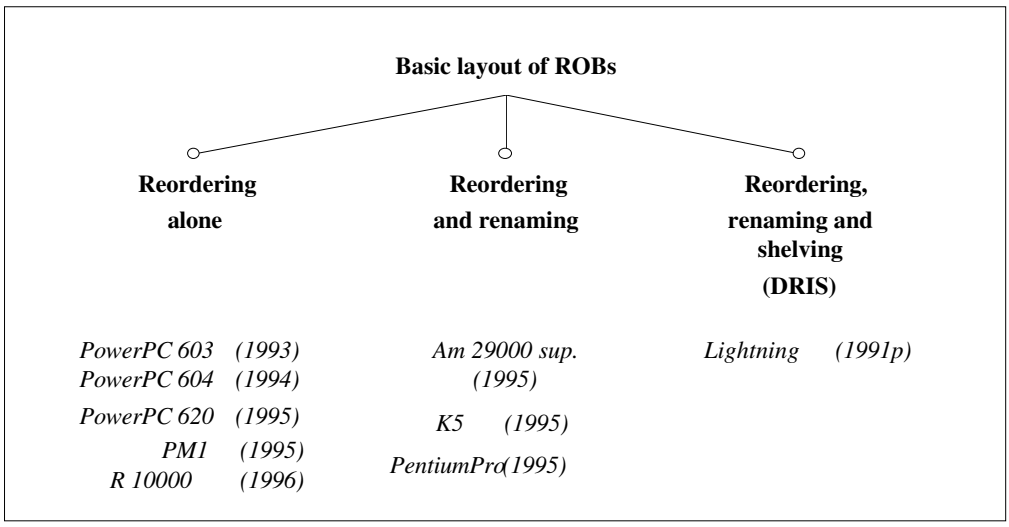


Figure 3.7.7: Basic layout of ROBs

In the next layout, an ROB is *utilised for both reordering and renaming,* as was discussed in Section 3.5. In this case, each ROB entry has to provide space to hold the

result of the corresponding instruction as well. Examples of this alternative are the Am 29000 superscalar, K5 and the PentiumPro. Here, we note that in the original proposal of Smith and Pleszkun (1988) the ROB also contained the instruction results.

In the third alternative, beyond *reordering and renaming the ROB is used for shelving as well.* In this case, the ROB is frequently designated as the DRIS (Deferred-scheduling, Register renaming Instruction Shelf). It follows that in this case, the ROB also has to provide space for shelving, which means either space for the source register numbers or for source operands, depending on the operand fetch policy employed. Although the DRIS is an attractive construct, it is a fairly complex one as well. So far,the DRIS was intended to be used only in a single processor, proposed, which has been the Lightning. We note that this ambitious processor never reached the market, as already mentioned.

In addition to the basic layout alternatives we will briefly discuss two further aspects of the design space. In particular, we are concerned with two implementation details, with the *ROB-size* and the *retire rate*. The **ROB-size** determines the number of entries in an ROB. This parameter limits the number of active, that is issued but not yet completed, instructions in a processor. The **retire rate** specifies the maximum number of instructions that can be completed by the ROB in each cycle. This value indicates the maximum throughput of the processor.

Here we note that as far as the maximum throughput is concerned, typically, the processor may be considered as consisting of a number of sequentially linked subsystems, which are decoupled from each other by some sort of buffers. These subsystems are; the fetch, issue, dispatch (if applicable), execute and retire (or completion) subsystem. The maximum throughput of the processor as a whole is determined by the "weakest" subsystem. In a good design, the throughput of all sequentially linked subsystems is balanced. This means that the fetch width, the maximum issue rate, the maximum dispatch rate (if applicable) and the maximum retire rate should be more or less the same. For fine-tuning of these parameters, extensive simulations are needed using benchmark programs.

In Table 3.7.1 we have compiled the ROB-size and retire rate for recent superscalar processors. We have also indicated whether intermediate results are stored into the ROB or not, and how the ROB is designated in the associated processor literature.

	ROB size	**Retire rate**	**Intermediate results stored**	**Designation**
ES/9000 *(1992p)*	32	2	no	Completion control logic
PowerPC 602 *(1995)*	4	1	n.a.	Completion unit
PowerPC 603 *(1993)*	5	2	no	Completion buffer
PowerPC 604 *(1994)*	16	4	no	ROB

PowerPC 620 *(1995)*	16	4	no	ROB
PentiumPro (1995)	40	3	yes	ROB
Am 29000 sup. *(1995)*	10	2	yes	ROB
K5 *(1995)*	16	4	yes	ROB
PM1 *(Sparc64, 1995)*	64	4	no	Precise stack unit
UltraSparc *(1995)*				Instruction reorder buffer
PA 8000 *(1996)*	56	4	yes	Active list
R 10000 *(1996)*	32	4	no	Completion unit

Table 3.7.1: Implementation details of ROBs

As the data in the above figure show, recent powerful superscalar processors provide as many as 16 - 64 entries in the ROB. This means that maximum 16 - 64 instructions may be active in these processors.

We note that the total number of shelves and the number of reorder buffer entries must be in balance. The reorder buffer holds all pending, i.e. not yet completed instructions. Some of the pending instructions are waiting in shelving buffers for their operands and/or for dispatch. Others are just in the process of execution. Therefore, we expect *more reorder buffer entries than shelves* in recent superscalar processors. As the data in Table 3.7.2 demonstrate, this is true for most processors listed in the Table. Exceptions are the Nx 586 and the R10000, where more shelves than reorder buffer entries are available.

Processor	**Total number of shelves**	**Number of ROB entries**
PowerPC 603 (1993)	3	5
PowerPC 604 (1994)	12	16
PowerPC 620 (1995)	15	16
Nx586 (1994)	42	14
K5 (1995)	14	16
PM1 (Sparc64) (1995)	36	64
PentiumPro (1995)	20	40
R10000 (1996)	48	32

Table 3.7.2: Comparison of available shelves and reorder buffer (ROB) entries in recent superscalar processors

With reference to Table 3.7.2 let us compare the retire rates and issue rates of the processors. What we see is that with a few exceptions (Am 29000 sup., PentiumPro) the retire rate equals the issue rate, which is quite natural.
Finally, we discuss the point whether intermediate results are stored in the ROB or not. In this respect it is definitive whether the ROB supports renaming or not. If the ROB also serves the renaming process, intermediate results have to be stored there, as in the processors PentiumPro, K5 and Am 29000 superscalar. If not, there is no need to hold intermediate results in the ROB.

We note yet that the reorder unit is named differently in connection with various processors. The most frequently used designation is the reorder buffer. Other terms used are the completion unit (PowerPC 602, UltraSparc), completion buffer (PowerPC 603), completion control logic (ES/9000), active list (R10000) or precise state unit (PM1).

3.8 Preserving the sequential consistency of exception processing

When instructions are executed in parallel, *interrupt requests*, which are caused by exceptions in instruction execution, are also generated *out-of-order.* If these requests are acted upon immediately, interrupts occur out-of-order, that is in a different order than in a sequentially operating processor. In this situation we say that the *sequential consistency of the interrupts is weak,* or in other words that we have to deal with **imprecise interrupts**, as depicted in Figure 3.8.1.

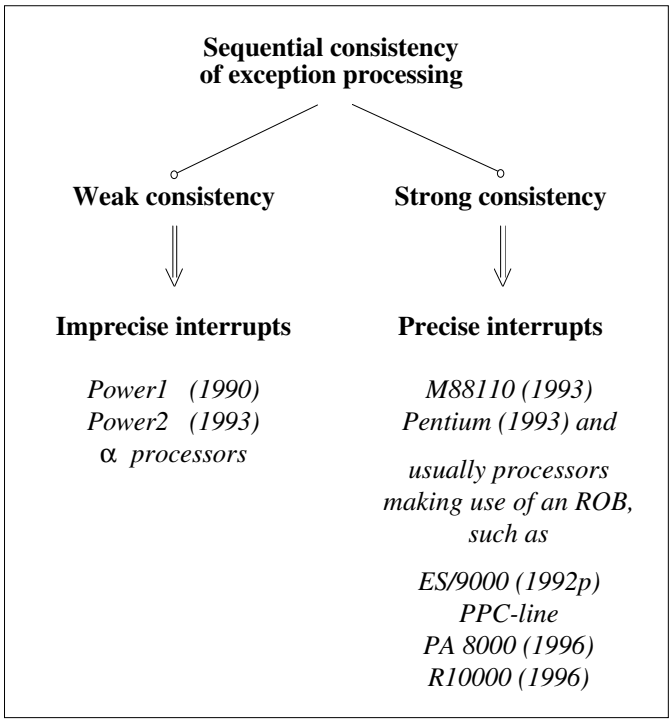


Figure 3.8.1: Sequential consistency of exception processing

When an imprecise interrupt occurs, the processor is unable to reconstruct the correct state unless appropriate additional mechanisms are employed. For instance, let us assume that the processor executes two subsequent instructions in parallel, an "older" division (div) and a "younger" addition (ad). It can happen that the "younger" "ad" finishes first and updates the processor state. If subsequently, the "older" "div" instruction causes an interrupt, for example because of an overflow, at the time when the interrupt request of the "div" is accepted, the processor state is already "corrupted" by the result of a later instruction. Thus, without any additional measure, it becomes impossible to reconstruct the correct state of the processor at the time when it accepted the interrupt request due to the "div" instruction. In a number of ILP-processors, including early superscalar processors, interrupts can be imprecise, such as in the Power2, or R8000. In both processors mentioned, the FP-interrupts are the ones which are imprecise. A further example is the α architecture, where all arithmetic exceptions are imprecise.

Most advanced superscalar processors maintain *strong sequential consistency with respect to interrupts.* This means that after interrupts the state of the processor remains consistent with the state a sequential processor would have. An obvious way to achieve **precise interrupts** is to maintain in-order instruction completion, for instance by using an ROB, and to accept interrupts caused by an instruction only when the related instruction retires. A few earlier and most recent superscalar processors offer precise interrupts, such as the MC88110, Pentium, the PowerPC line of processors, the PA 8000 and the R10000.

3.9 Implementation of superscalar CISC-processors using a superscalar RISC core

As far as the implementation of superscalar CISC processors is concerned, a trend has become evident. An increasing number of recent CISC processors utilize a *superscalar RISC core.* Examples are the Nx586, K5 and the PentiumPro. In these x86 compatible processors, the CISC instructions are first converted into RISC-like instructions during decoding. The RISC-like instructions are then executed using a superscalar RISC core, as outlined in Figure 3.9.1.

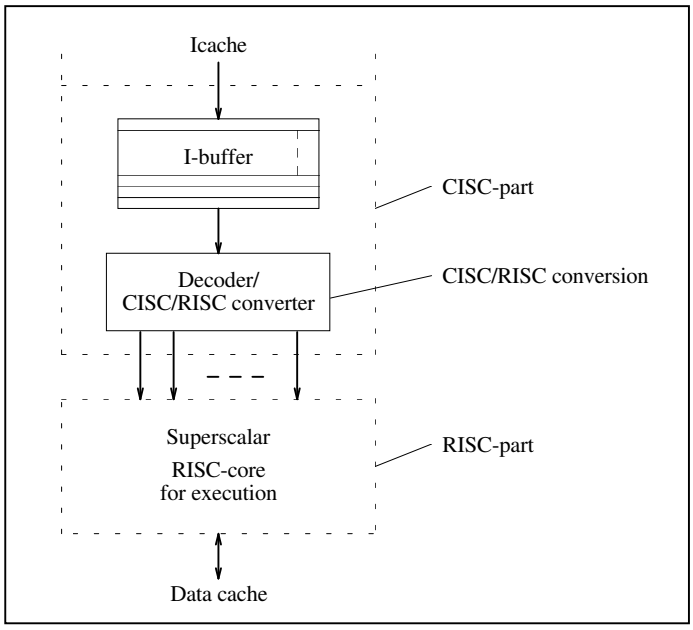


Figure 3.9.1: The principle of superscalar CISC execution using a superscalar RISC core, as employed in the processors Nx586 (1994), K5 (1995) and PentiumPro (1995).

The RISC-like instructions, called subsequently *RISC-operations,* are of the load-store type. Thus, many CISC instructions are converted directly to a single RISC operation, such as all register to register ALU instructions, loads and stores. In contrast, ALU instructions referring to memory operands generate two or more RISC operations. For instance, the x86 (CISC) instruction

```
SUB   EAX, [EDI];
```

which subtracts the content of the memory location [EDI] from the content of the register EAX, would be translated into the following two RISC-like operations:

```
MOV   EBX, [EDI];

SUB   EAX, EBX;
```

Here,. the first RISC-like instruction loads `EBX` with the referenced memory operand. Then the subsequent register-register instruction performs the required substraction.

More complex CISC instructions, such as string or transcendental ones, will be converted into long sequences of RISC operations. However, the frequency of such complex instructions is quite small. On the average, one CISC instruction generates 1.5 - 2 RISC operations (Gwennap 1995).

Some characteristic data on superscalar CISC-processors using a superscalar RISC core for the execution, are summarised in Table 3.9.1.

Processor	Decoding/ conversion speed	Designation of RISC-operations	Length of RISC-operation encoding	Maximum issue-rate of RISC-operations

Nx586 [1] (1994)	1 CISC instr./cycle	RISC86 instructions	~ 100 bits	3 - 4
K5 (1995)	n.a.	R-ops.	59 bits (on average)	4
PentiumPro (1996)	1 - 3 CISC instr./cycle	uops	118 bits	3

Comment:
[1] The Nx586 is actually a scalar processor from the point of view of CISC-instructions. However, it is a superscalar processor in respect to the RISC-operations.

Table 3.9.1: Characteristic data on superscalar CISC processors using a superscalar RISC core for the execution

The indicated processors convert 1-3 CISC instructions per cycle. The Nx 586 translates only a single CISC instruction in each cycle while the PentiumPro can handle up to three. Actually, the PentiumPro has three instruction decoders (see Figure 3.9.2). The general decoder can translate any CISC instruction which will be converted into not more than four RISC operations, called *uops*. For more complex CISC instructions, which will be converted into more than four uops, such as string- or transcendental instructions, an uop-sequencer takes over the task of uop-generation. This sequencer, which is in essence a ROM, is capable of generating up to four uops in each cycle.

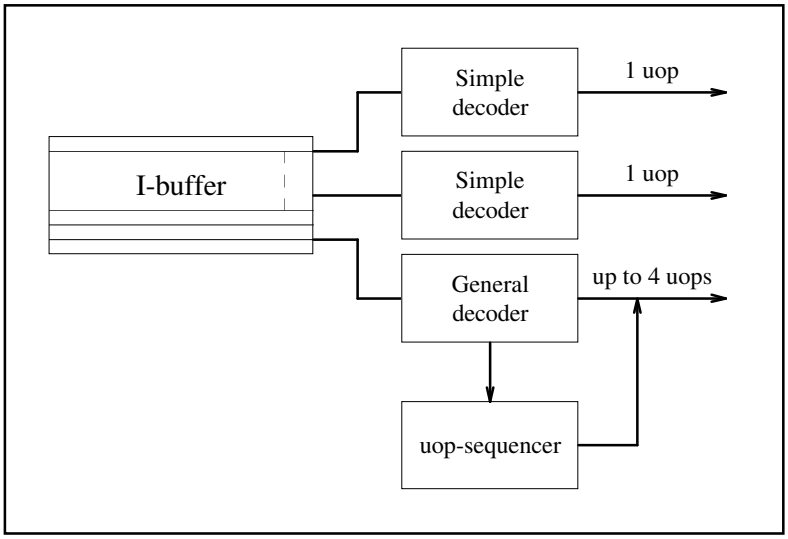


Figure 3.9.2: Decoding/converting of CISC instructions to RISC operations, called uops, in the PentiumPro (Gwennap 1995)

In the PentiumPro simple decoders are restricted to decoding straightforward CISC instructions like register to register ALU instructions. Such instructions which can be converted into a single uop. Decoding/conversion is done in program order. Thus, if a simple decoder cannot manage to convert the subsequent instruction, decoding will be blocked. As a result, the PentiumPro decodes and converts more than one CISC instruction in a cycle only when the second (and the third) CISC instruction is a simple one.

In the superscalar CISC processors mentioned above, the RISC operations are designated differently. In the PentiumPro they are called uops, whereas in the Nx586 terminology they are labelled RISC86 instructions; the K5 literature refers to them as R-ops.

We point out that the internal encoding of the RISC operations requires much longer words than traditional RISC instructions are. This is due to the microinstruction-like format which directly contains the referenced data. As Table 3.9.1 indicates, the Nx586 and the PentiumPro use a fixed format requiring about 100 bits for each RISC operation. On the other hand, the K5 employs a variable-length encoding with 59 bits requested on the average for a single RISC operation.

We conclude our description of superscalar CISC processors with a RISC core by discussing the *maximum issue rate* for the RISC operations (see Table 3.9.1). The Nx586 and the K5 are capable of issuing four RISC operations per cycle, while the PentiumPro can handle only 3. We note, however, that the Nx586 is restricted to converting at most one single CISC instruction/cycle. Thus, the Nx586 is a scalar processor from the point of view of CISC execution, and a superscalar one if we consider the RISC core. In confront, the K5 and the PentiumPro are roughly dual-issue superscalar CISC processors.

3.10 Case studies of superscalar processors

In this chapter we describe a number of the latest superscalar processors which use shelving and renaming. We present the microarchitecture and describe the operation of the following processors:

- the *R10000*,
- the *PowerPC 620*, and
- the *PentiumPro*.

In order to facilitate understanding and cross-comparisons, we have followed essentially the same format in our discussion of each.

Our description focuses on the crucial points of up-to-date superscalar operation, that is on *instruction* issue, *renaming*, *shelving*, *dispatching and reordering*. Further points, like, cache- and memory implementation, load/store and branch operations or exception handling are omitted. In regards to branch processing, we refer to chapter 8. Concerning other points we refer to the standard literature as well as to the listed publications of the particular processors.

For further clarity, we have simplified our block diagrams in places. For example, we picture multiple buses (like instruction, operand or result buses) as single buses and indicate their multiplicity by an appropriate marking.

Finally, we note that certain important details may be missing from the descriptions due to the lack of the pertaining relevant proprietary information.

3.10.1 R10000

The MIPS R10000 (1996) is the latest member of the R line of processors. It implements the MIPS IV ISA (Instruction Set Architecture), which is a superset of the MIPS III, supported for instance by the R8000. At a planned clock rate of 200 MHz, this four-way superscalar processor has a peak performance of 800 MIPS.

The R10000 has the following main features:

- it is a four way superscalar processor with a maximum dispatch rate of five,
- it uses predecoding,
- it has three group reservation stations,
- operands are fetched in connection with instruction dispatch,
- renaming is implemented using a merged architectural and rename register file, and
- sequential consistency is preserved by means of an ROB.

We describe the core part of the microarchitecture of the R10000 with reference to Figures 3.10.1 and 3.10.2.

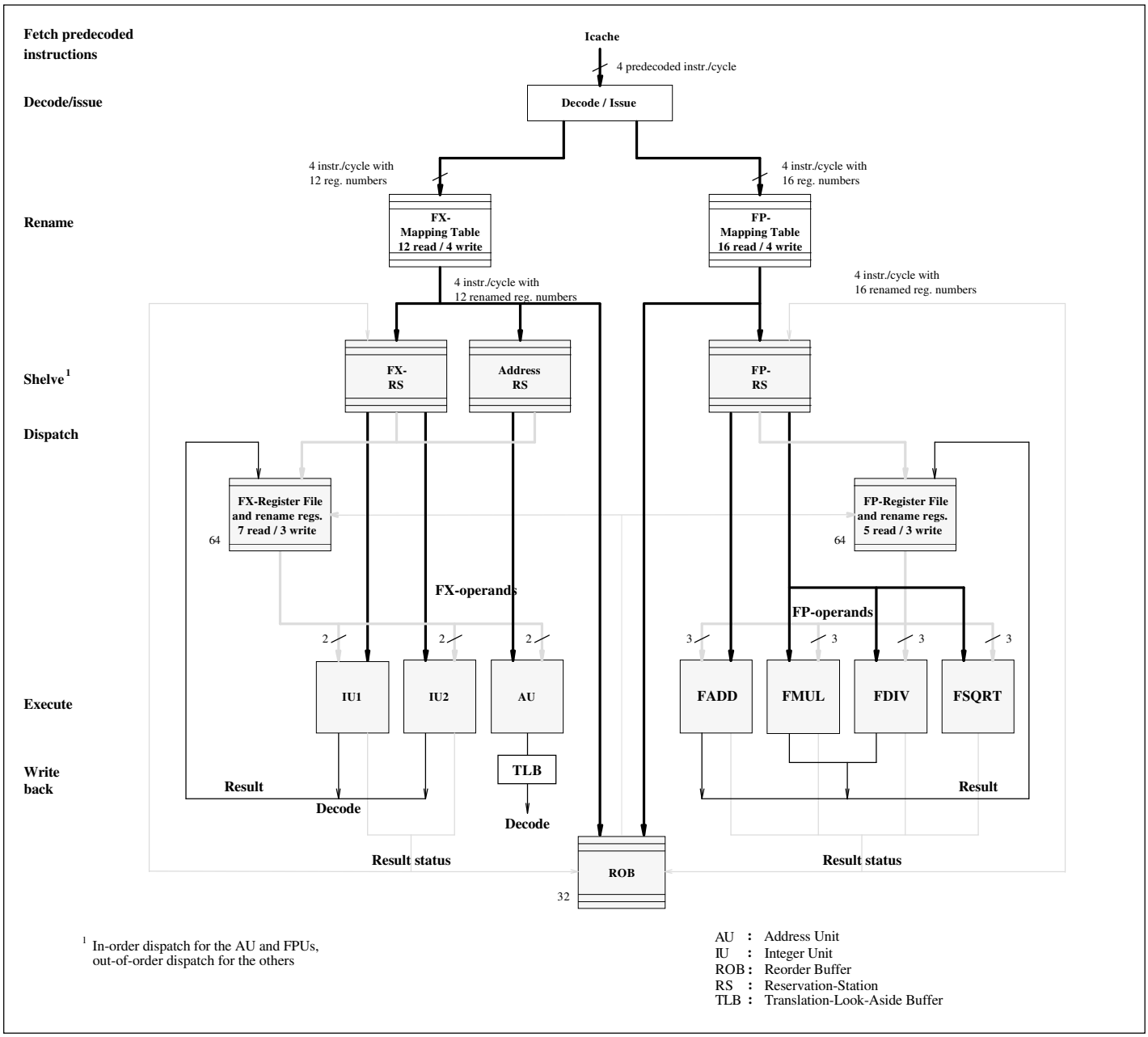


Figure 3.10.1: Core part of the microarchitecture of the R10000

As shown in Figure 3.10.1, the R10000 execution core consists of the following main units: decode/issue unit, FX- and FP-mapping tables, three group reservation stations, merged FX- and FP-register files, seven FUs and the ROB. The EUs are as follows: three FX-units and four FP-units. The FX-units are the IU1, IU2 and the AU. IU1 and IU2 can perform a wide range of simple arithmetic and logical instructions. In addition, IU2 can carry out integer multiply and divide operations. The address unit (AU) is an address adder for generating addresses. The available FP-units are used for FP-addition (FADD), multiplication (FMUL), division (FDIV) and square root calculation (FSQRT).

The R10000 has a short five-stage pipeline for FX-operations, consisting of the Fetch, Decode, Dispatch, Execute and Write back cycles, as depicted in Figure 3.10.2. This figure also indicates the layout of the load/store- and FP-pipelines, which are not discussed here.

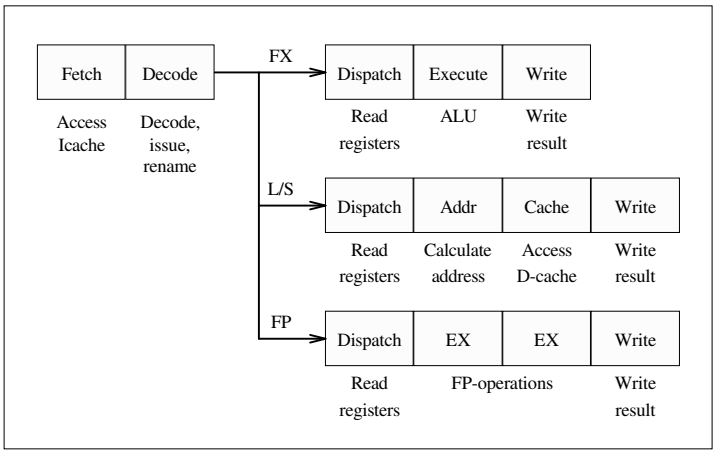


Figure 3.10.2: Layout of the R10000 pipelines

The R10000 employs predecoding in order to shorten the critical decode/issue/rename path. In each cycle four predecoded instructions are fetched from the Icache (Fetch cycle). Subsequently, all four instructions are decoded, issued and renamed in the next cycle (Decode cycle). Both decoding and issue are carried out in program order. Decoded register numbers of issued instructions are renamed. Renaming is carried out using separate merged register files and mapping tables for FX- and FP-instructions. Each mapping table has 32 entries and enough read and write ports to rename up to four instructions of the same type in each cycle. Actually, the FX-mapping table provides 12 read and 4 write ports while the FP-table has 16 read and 4 write ports. Thus, FX-instructions may have up to three and FP-instructions at most four operands.

Destination and source register numbers are renamed quite differently. During renaming, each destination register is allocated a new physical register out of the 64 physical registers available in each of the merged FX- and FP register files, provided that a free one is at hand. In addition, each table is updated with the newly established mappings of up to four destination registers. Source registers are renamed simply by reading the actual physical register numbers, which are allocated to the architectural registers in concern from the related mapping table. During renaming a check is made for possible data dependencies among the instructions to be renamed in the same cycle. Recognised dependencies are taken into account in the rename process, as was explained in Section 3.5.6.

At the end of the Decode cycle, issued and renamed instructions are shelved. Shelving is accomplished by writing each renamed instruction into one of the reservation stations. The R10000 provides three group reservation stations, one each for FX-instructions, FP-

instructions and addresses of load/store instructions. Each group station has 16 entries. Up to four instructions can be written into any of the group stations in each cycle. Thus, if necessary, all four instructions renamed in the same cycle can be written into the same group station.

Next, we describe how shelved instructions are dispatched from the group reservation stations. Since the R10000 employs both register renaming and speculativebranch processing, only true data dependencies can restrict instructions from becoming eligible for dispatching. In other words, dispatching is dataflow-based. Later in this section we return to the question of how availability of the operands is checked in the reservation stations.

Instructions are dispatched from each reservation station in an out-of-order fashion. In this way, older instructions that are not yet ready do not block the dispatching of younger instructions that are ready for execution. In most cases EUs can accept a new instruction in each cycle. Exceptions are the IU2 while performing an FX-multiplication or division and the FP-units Fmul, FDIV and FSQRT. For illustration, we summarise in Table 3.10.1 the latency and repeat rate values of the R10000 instructions (MIPS 1995).

Instruction		**Latency cycles**	**Repeat rate cycles**
Simple integer and logical instructions		1	1
Load/Store integer data		2	1
Load/Store FP-data		3	1
Integer multiply	32 bit 64 bit	6 10	6 10
FP-multiply		2	1
Integer divide	32 bit 64 bit	35 67	35 67
FP-divide	32 bit 64 bit	9 17	8 16
FP-square root	32 bit 64 bit	16 30	15 29

Table 3.10.1: Instruction latencies and throughput values of the R10000

When dispatching instructions out-of-order, priority rules are needed to break the tie if multiple instructions are ready for execution in the same cycle. As usual, the R10000 gives priority to non-speculative instructions over speculative ones and to older instructions over younger ones. The FX- and FP-reservation stations are able to select up to two instructions per cycle, whereas the address reservation station is confined to choosing only one instruction per cycle, as indicated in Figure 3.10.1. Thus, at most five instructions can be dispatched in every cycle.

Dispatched instructions are forwarded to the corresponding EUs while the operands needed are picked up from the register files. The merged FX-register file has 7 read ports for simultaneously delivery of two operands for each of three FX-instructions and one data value for the cache. The merged FP-register file has five read ports to provide two FP-instructions with two operands each and to supply one FP-data to the cache. We note that Figure 3.10.1 shows only the execution engine and does not indicate data paths to and from the cache.

The results are forwarded to the related register files. Two write ports are available for updating each of the merged files with the computed destination register values. Results are accompanied by tags identifying the physical register where the result is to be written into. We note that each physical register contains a "register value valid" bit (usually called scoreboard bit). When a new physical register is allocated to an architectural destination register, the "register value valid" bit of that physical register is reset. Later, when the result has been forwarded, this bit is set to "1", indicating the availability of the corresponding register value. The reservation stations keep track of the availability of the source operands needed based on the "register value valid" bits.

The R10000 preserves a strong sequential consistency by using an ROB. The ROB holds 32 consecutive active instructions. During issue, up to four instructions are written into the ROB in each cycle in program order. After finishing, they will retire in program order again, up to four instructions in each cycle. However, not more than one store can be retired per cycle due to data path constraints. In order to keep track of instruction execution the ROB gets "finished" status reports from the EUs. Furthermore, the ROB is responsible for indicating which FX- and FP-physical registers can be reclaimed following the retirement of instructions.

3.10.2 The PowerPC 620

The PowerPC 620 is the highest performance model of the PowerPC line. Its microarchitecture is based on the PowerPC 604 which is shown in Figure 3.4.24, but incorporates certain extensions for achieving greater performance. With its initial clock rate of 133 MHz, the PowerPC 620 offers a peak FX-performance rate of 530 MIPS.

The main features of the PowerPC 620 are:

- it is a four-way superscalar processor,
- it employs individual reservation stations,
- renaming is implemented by separate architectural and rename register files,
- sequential consistency is maintained using a reorder buffer.

As shown in Figure 3.10.3, the PowerPC 620 has six EUs capable of parallel operation. These units are as follows: a branch processing unit (BPU), two simple integer units (IU1 and IU2) performing one-cycle integer and logical operations, a single multi-cycle integer unit (MIU), performing mainly integer multiplications and divisions, a load/store unit (L/SU) and an FP-unit (FPU).

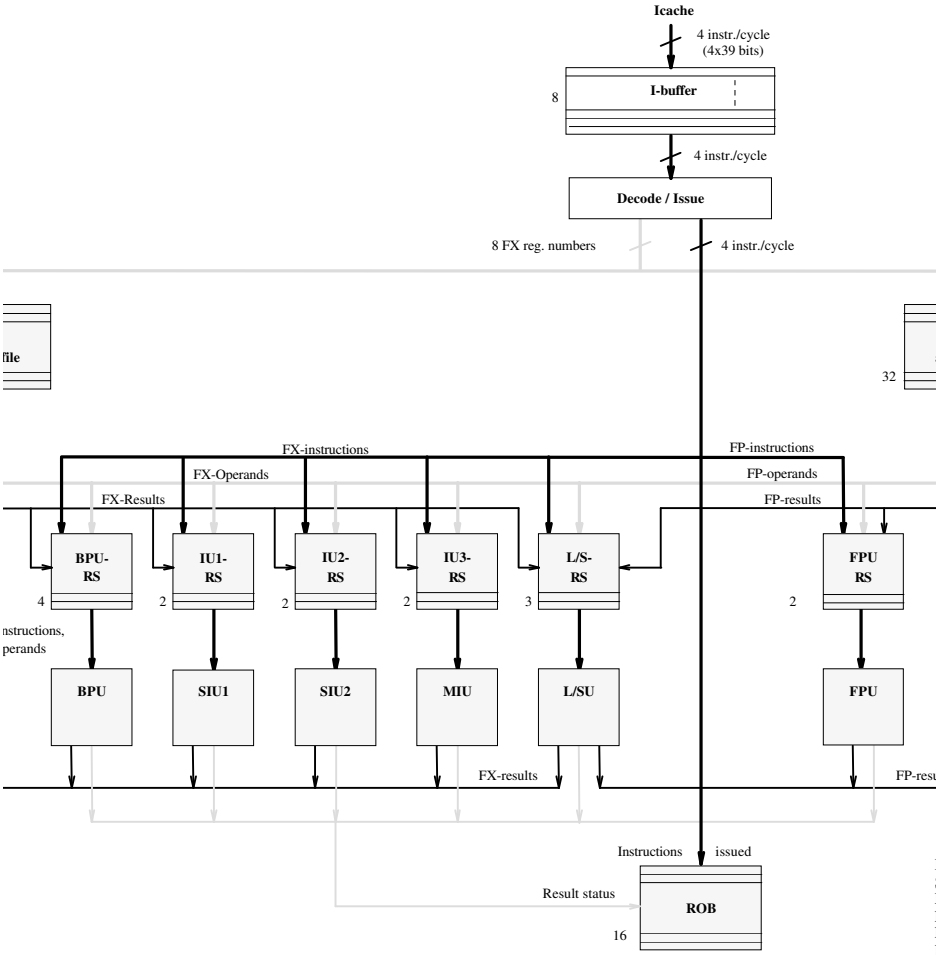


Figure 3.10.3: Core part of the microarchitecture of the PowerPC 620

The PowerPC 620 employs individual reservation stations. Two buffers are available in the integer- and FP-units, whereas the L/SU has three entries and the BPU four. Here we note that the existence of four entries in the BPU allows speculative execution of up to four unresolved conditional branch instructions.

Renaming is carried out by means of separate architectural registers and rename registers. The PowerPC 620 utilizes eight FX- and eight FP-rename registers. Sequential consistency of the execution is preserved by a 16-entry reorder buffer (ROB).

The PowerPC 620 has basically a five-stage pipeline, as indicated in Figure 3.10.4.

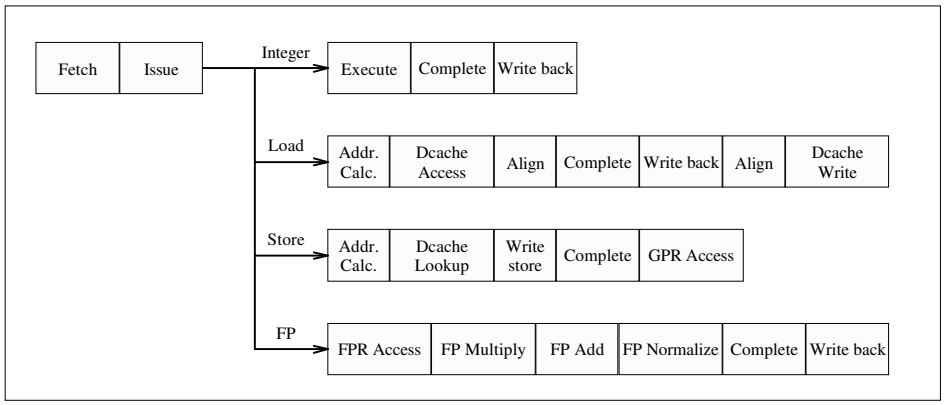


Figure 3.10.4: Layout of the pipelines of the PowerPC 620

The processor operates in the following way. Predecoded instructions are fetched in the Fetch-cycle from the Icache into an 8-entry I-buffer, four at a time. In the Issue-cycle the processor decodes and issues up to four instructions per cycle into the six individual reservation stations present. Instructions are issued in order and aligned. However, there are four issue constraints that need to be mentioned:

a, not more than one instruction can be written into a particular reservation station per issue;

b, instructions requiring two FX-source operands can only be issued from the first half of the lower four entries of the I-buffer.

c, each instruction to be issued requires a free entry in the ROB, and

d, if the instruction produces a result value, register renaming must be performed, which requires a free rename register entry.

Renaming is carried out in a straightforward manner. When an instruction is issued which produces a result value, a rename register is allocated to hold the result. During operand fetching the rename register has to be looked up to find out whether any of the

source registers have been renamed or not. If they have, the value contained in the renamed register should be fetched as the operand.

Operand availability in the rename registers is managed through the use of scoreboarding. Thus, as part of the issue process, the scoreboard bits of the renamed destination registers are reset. This indicates for all subsequent instructions that the related register value is still unavailable.

Operands are fetched during issue. Thus, at the same time as instructions are being forwarded to the reservation stations, up-to 8 FX-register numbers and 3 FP-register numbers are passed to the appropriate register files. Corresponding rename and architectural registers are simultaneously searched for the requested register values. Here, three scenarios are possible. If a required source operand is found in one of the rename registers and its value is valid, i.e. available, the contents of that register is forwarded into the appropriate field of the corresponding reservation station. If the required register value is contained in the rename register, but its value is not available, a tag, which is the rename register identifier, is stored in the reservation station in place of the operand value. Finally, if the requested register value is not found in the rename register, the corresponding value from the architectural register file is transmitted into the reservation station.

Operand fetching from the rename registers is quite a complex task, as will be shown below.

Each rename register has four fields (see Figure 3.10.5). These are the rename valid bit, the register number to which this rename register is allocated, the result value, if available, and the result valid bit.

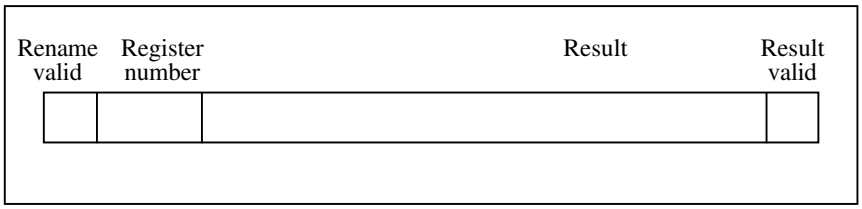


Figure 3.10.5: Format of the rename register entries

Fetching operands from the rename registers requires an associative search since the contents of the register number fields must be searched for matching source register numbers. Furthermore, since subsequent instructions may use the same destination register, the architectural register can have multiple renames. During an associative search, clearly, the latest rename is to be accessed. Thus, accessing operands from the rename registers requires an associative search for the youngest renamed value of the requested source register.

Dispatching from the reservation stations is performed partly in-order, partly out-of-order. The reservation stations associated with the BPU and FPU dispatch instructions in-

order, whereas the other stations operate out-of-order. An instruction which is held in the reservation station becomes eligible for execution when all of its operands are available. For reservation stations with out-of-order dispatch, arbitration is needed when more than one instruction is eligible for execution. The method used is to select the oldest eligible instruction. Now, the execution cycle can begin. Some operations, such as simple FX-operations are performed in one cycle. More complex operations such as division or multiplication require a number of execution cycles.

If an EU finishes its operation, it forwards its result and the associated tag to the result bus. Results are used to update both the related rename register file and the reservation stations. Updating the corresponding rename register is simple. Since the result is governed by a tag, which is the rename register identifier, the related rename register can be accessed using the tag as an index. Thus, the corresponding rename register is updated with the produced value and the result valid bit is set to "1" to allow subsequently issued instructions to access that value. The reservation stations also have to be updated. This update, however, requires an associative search for the tag (rename register identifier) for each result in all source operand fields of each reservation station concerned. If there is a match, the rename register identifier is replaced with the result value, and is marked as available. Thus, instructions waiting for these operands become eligible for execution during the next cycle. The finishing of an instruction changes its processing state. The information that a particular instruction has finished execution is called the result status.

The ROB ensures the correct ordering of writing back the results into the architectural register files. This is achieved by appropriate controlling of the retirement of the instructions. Retirement is the final step of instruction processing which updates the program state. The ROB lets instructions retire in strict program order by allowing an instruction to retire only if has been completed and all prior instructions have already been retired.

Once it receives the result status, that is the identifiers of a completed instructions, the ROB is able to retire up to four instructions in each cycle. As part of the retirement process, results provisionally stored in the rename registers are written back into the architectural registers and the related rename valid bits are reset to become reusable for subsequent renaming. Four FX-results and two-FP-results can be written back into the associated architectural file. When an instruction is retired, the allocated ROB entry is also deallocated for further use. This completes our description of the operation of the PowerPC 620.

3.10.3 The PentiumPro

The PentiumPro is the flagship of Intel's x86 line of processors. It features a 133 MHz initial clock rate, which is quite an impressive figure for a CISC processor.

This processor has the following main features:

- it is a superscalar CISC processor with a RISC core,
- it issues up to three RISC operations per cycle, and dispatches up to five RISC operations per cycle,
- it has a unified central reservation station with 20 entries, used at the same time for all types of instructions, such as for FX-, FP-instructions etc.,
- strict sequential consistency is retained using a reorder buffer,
- renaming is implemented in the reorder buffer.

The execution engine of the PentiumPro is shown in Figure 3.10.6.

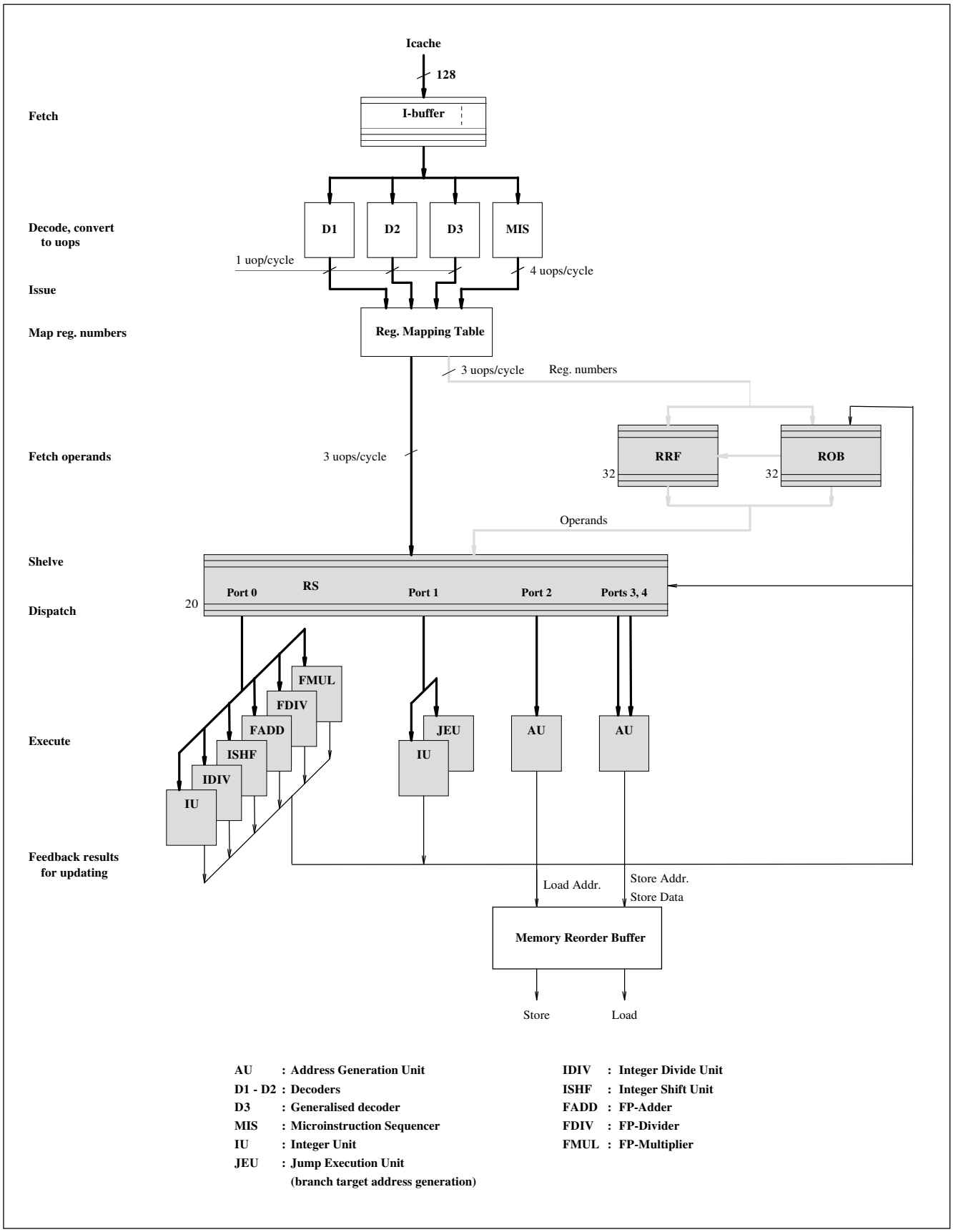


Figure 3.10.6: Core part of the microarchitecture of the PentiumPro

The PentiumPro has an extremely long pipeline of at least 12 stages for FX instructions (see Figure 3.10.7).

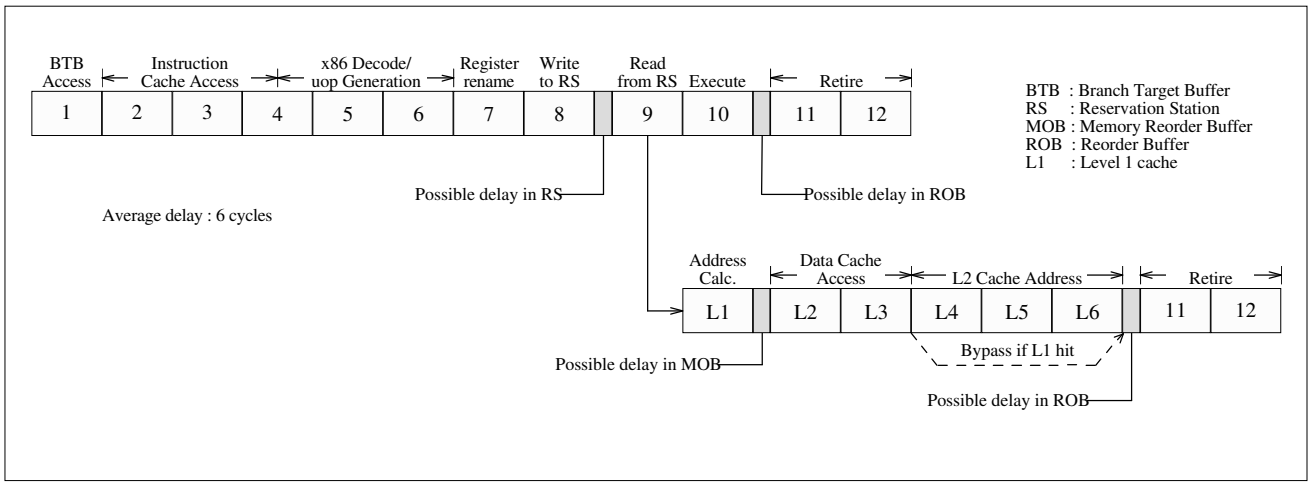


Figure 3.10.7: Layout of the pipelines of the PentiumPro

Instructions are fetched in 128 bit chunks from the Icache into the I-buffer. Like other superscalar CISC processors (see section 3.9) the PentiumPro first converts the fetched CISC instructions internally into RISC ones, known as uops. Each uop has a regular format and a fixed length of 118 bits. Subsequently, a superscalar RISC core executes the uops. Finally, the back-end of the processor ensures for the logical consistency of the execution.

Instructions taken from the I-buffer must first be aligned because of the variable instruction length character of the CISC instructions. Then, up to three CISC instructions are decoded and converted to RISC instructions in each cycle. The conversion is carried out by two simple decoders (D1 and D2), a general decoder (D3) and a microinstruction sequencer (MIS). Both simple decoders can only accept instructions which are converted into a single uop. More complex instructions which translate into maximum four uops are translated by the generalised decoder (D3), and instructions resulting in more than four uops are converted by the MIS. We emphasise that decoding and conversion is carried out in-order.

The next steps of the operation are register renaming, operand fetch and issuing of the renamed uops into the central reservation station.

Register renaming is implemented using a register mapping table while intermediate results of renamed registers are stored in the ROB. The register mapping table (called the register alias table, RAT) is able to map register numbers of three uops in a cycle. Using a mapping table to convert architectural register numbers into renamed ones eliminates the need for an associative search during operand fetch from the ROB. Operand fetching is done during issue. Therefore, renamed register numbers are passed on to both the architectural register file and to the ROB. If a valid entry for the required renamed register number is found in the ROB, this entry is taken as the operand value.

Uops are dispatched out-of-order from the central reservation station, up to five in each cycle. Here again instructions with available operands are eligible for execution. Intel has not revealed its arbitration rule, in detail; however it has been published that older uops are preferred over younger ones. As Figure 3.10.6 shows, the central reservation station has five complex ports for sending instructions to execution units and receiving results for updating and bypassing. On first sight it is surprising that six out of ten execution units are connected to port 0. These are the following: an integer unit (IU), the integer divide unit (IDIV), a shift unit (SHF), and three dedicated FP-units for addition (FADD), division (FDIV) and multiplication (Fmul). The main reason for hanging all these units on a single complex port is to save die area. Consider here that each FP-operand needs an 86 bit wide data path. Thus, the sharing of a single port by six units is a design trade off between performance and resulting complexity.

There are two units connected to port 1, a second integer unit (IU) and a jump execution unit (JEU), which calculates branch addresses. Port 2 serves the address generation unit (AU) whose task is calculate load addresses. Finally, ports 3 and 4 are devoted to the calculation of store addresses by a further address generation unit (AU).

Results and their tags are used to update both the central reservation station and the ROB. Updating of the central reservation station must be done associatively. If shelved instructions contains the same tag as broadcasted by one of the execution units these tags must be replaced with the corresponding result values. In addition, the availability status of the related operands must be updated.

Updating of the ROB serves two purposes. First, the destination register whose value has just been produced needs to be updated for subsequent instructions to be able to fetch this value. The other purpose of updating concerns the need to retain sequential consistency of execution. Thorough the use of a controlled retirement of instructions the ROB maintains logical consistency in the following way. When an instruction is finished, the ROB let it retire only after all prior instructions have been retired. An additional requirement for its retirement is that uops belonging to the same CISC instruction are allowed to retire only if all uops belonging to the same CISC instruction are finished.

Retirement of an uop makes the floor free for writing back the result of the corresponding ROB entry into the related architectural register and for deallocation of the corresponding ROB entry.