Register Renaming

Register renaming is a standard technique for removing false data dependencies among register data. It was first suggested by Tjaden and Flynn in 1970, although they did not use the term "renaming". Keller introduced the term in 1975 and described one possible implementation.

Register renaming may be implemented either *statically* or *dynamically*. Partial dynamic renaming has been used in superscalar processor since 1990, with full renaming emerging around 1992.

[Register renaming presumes the three-operand instruction format.]

Implementation of register renaming

Static implementation

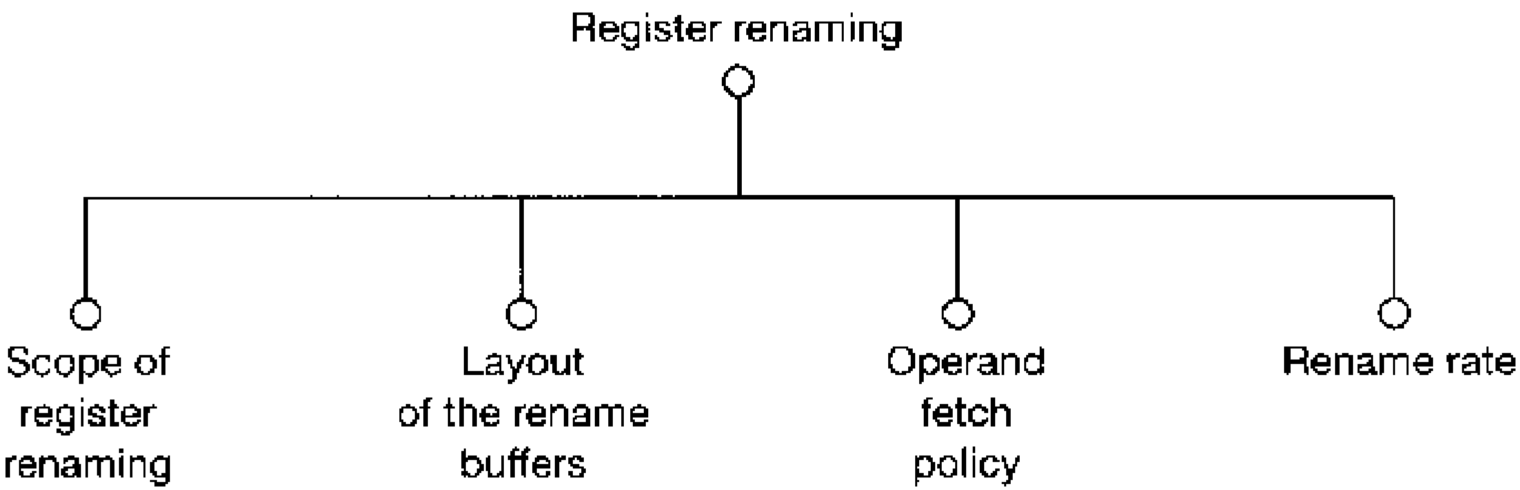
Performed during compilation, i.e. statically, in parallel optimizing compilers

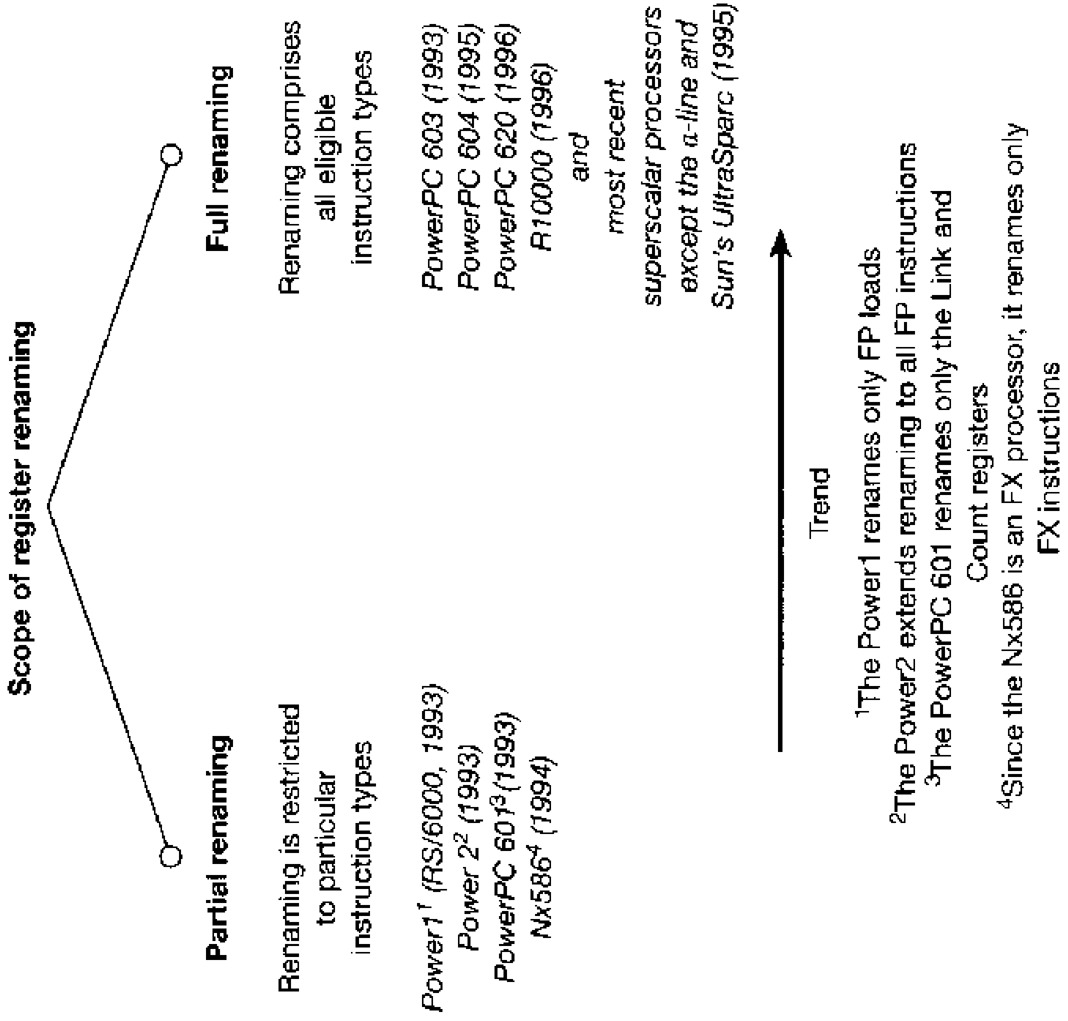
Dynamic implementation

Performed during execution, i.e. dynamically, in superscalar processors

Design Space of Register Renaming

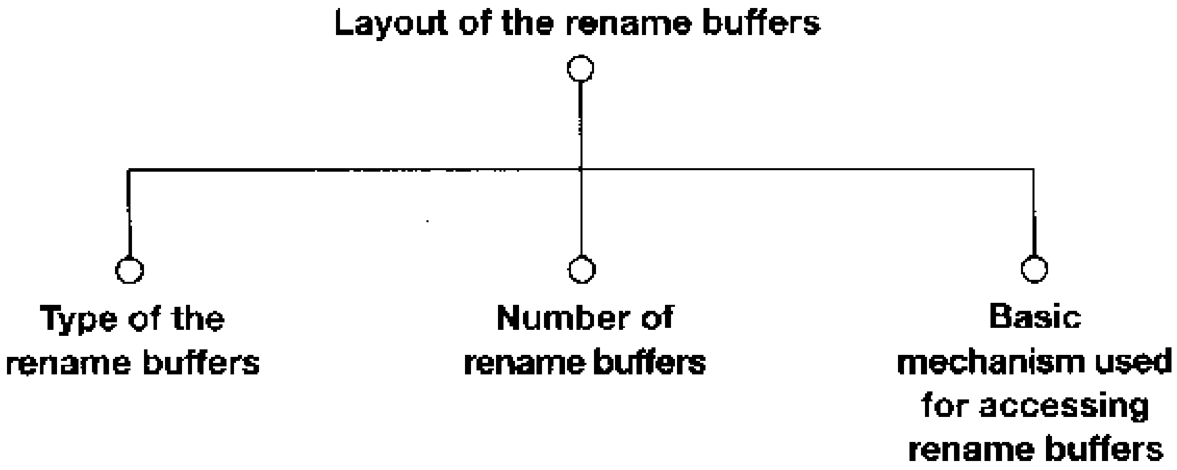
The design space of register renaming resembles that of shelving.




Scope of register renaming
Partial renaming
Full renaming
Renaming is restricted to particular instruction types
Power1[1] (RS/6000, 1993)
Power2[2] (1993)
PowerPC 601[3] (1993)
Nx586[4] (1994)
Renaming comprises all eligible instruction types
PowerPC 603 (1993)
PowerPC 604 (1995)
PowerPC 620 (1996)
R10000 (1996)
and
most recent superscalar processors except the α-line and Sun's UltraSparc (1995)
Trend
[1]The Power1 renames only FP loads
[2]The Power2 extends renaming to all FP instructions
[3]The PowerPC 601 renames only the Link and Count registers
[4]Since the Nx586 is an FX processor, it renames only FX instructions

The Layout of Rename Buffers

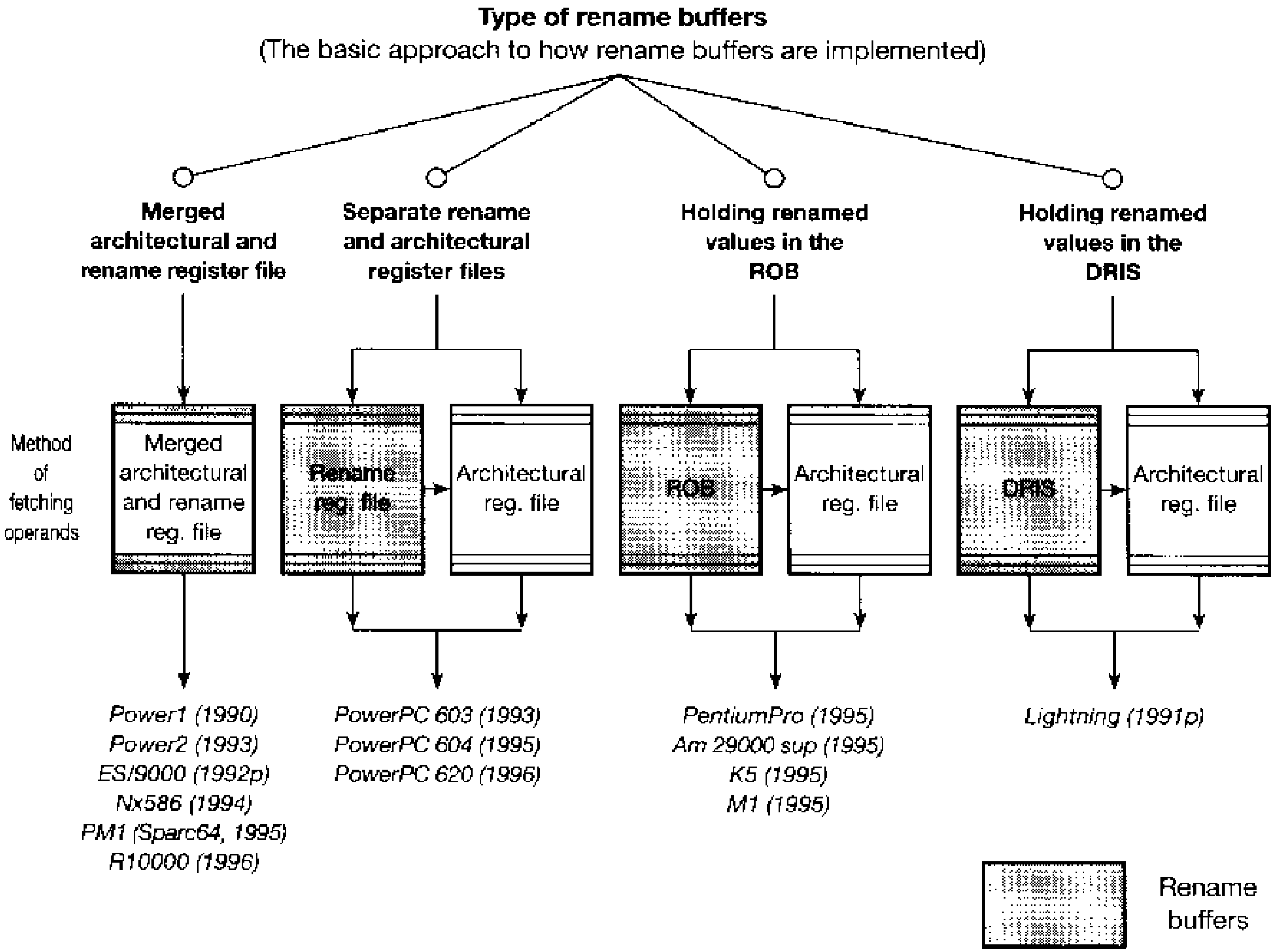
The layout of the rename buffers establishes the actual framework for renaming.



The Types of Rename Buffers

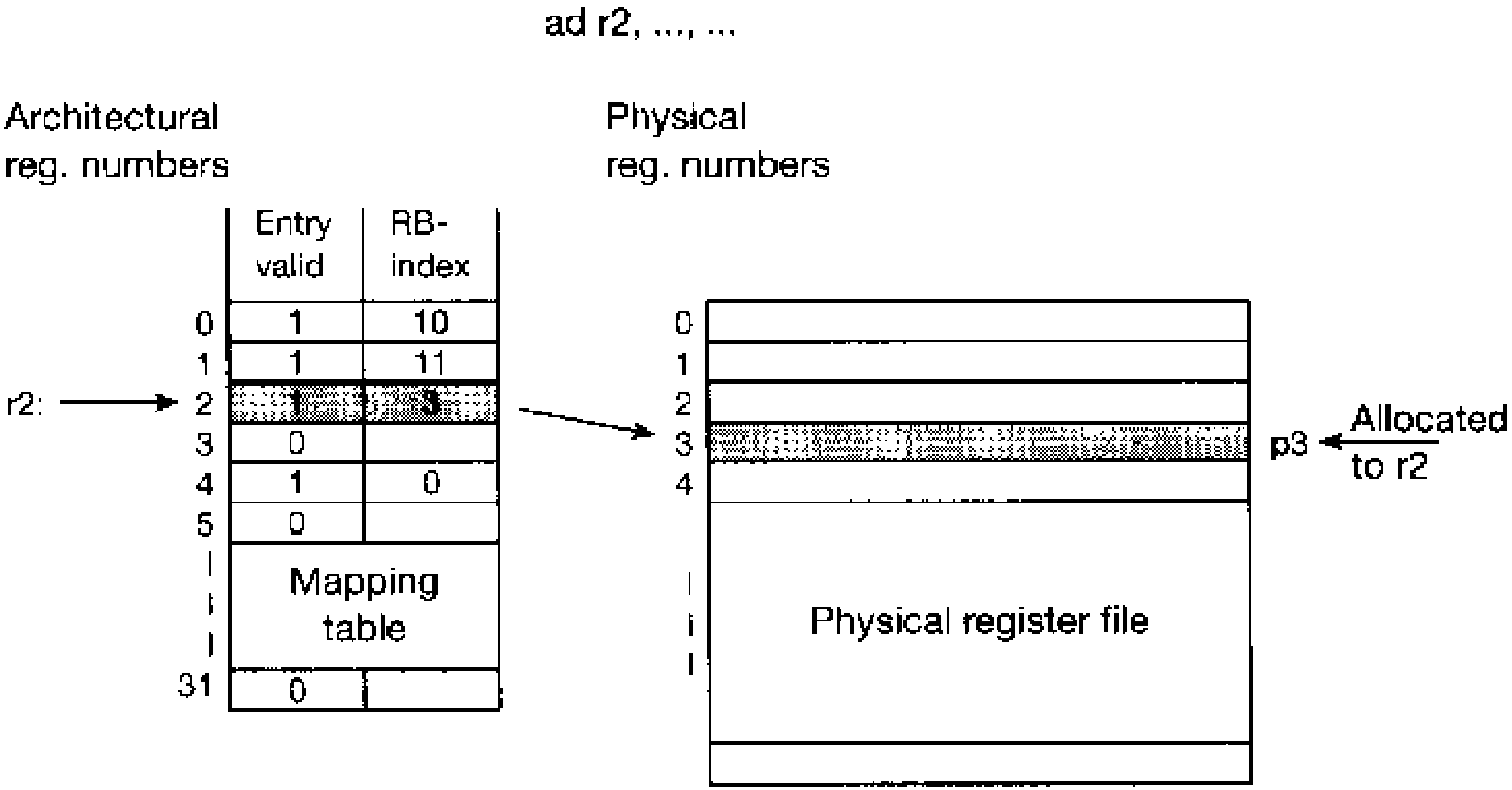
The chosen type of rename buffer has the largest impact on renaming since it determines where the *intermediate* results of instructions are written into or read from.

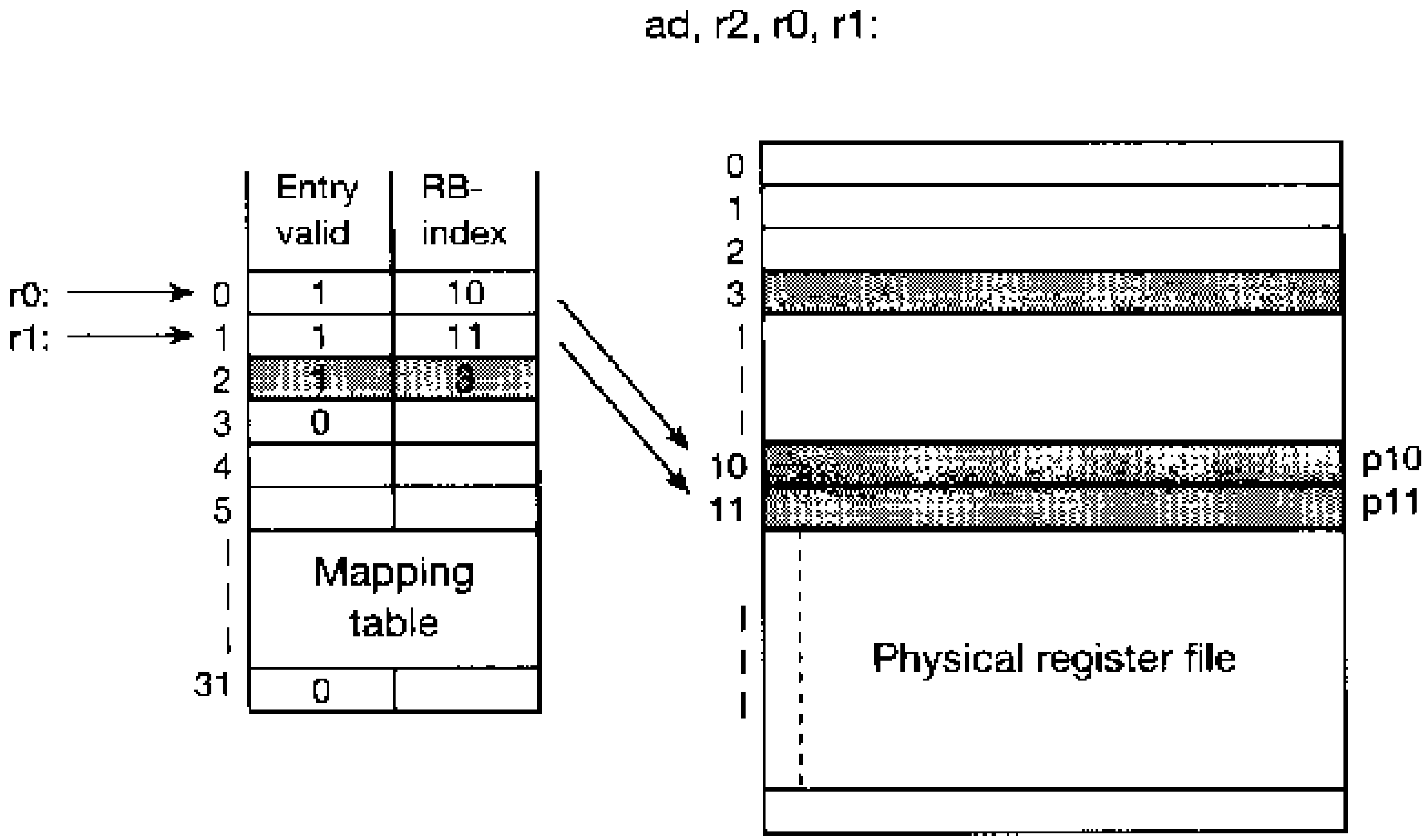
Intermediate results are those which have already been generated but are not yet qualified to modify the actual program state by writing them into the architectural registers. They have to wait until it is sure that the modification of the program state does not violate the sequential consistency of the execution.


Type of rename buffers
(The basic approach to how rename buffers are implemented)
Merged architectural and rename register file
Separate rename and architectural register files
Holding renamed values in the ROB
Holding renamed values in the DRIS
Method of fetching operands
Merged architectural and rename reg. file
Rename reg. file
Architectural reg. file
ROB
Architectural reg. file
DRIS
Architectural reg. file
Power1 (1990)
Power2 (1993)
ES/9000 (1992p)
Nx586 (1994)
PM1 (Sparc64, 1995)
R10000 (1996)
PowerPC 603 (1993)
PowerPC 604 (1995)
PowerPC 620 (1996)
PentiumPro (1995)
Am 29000 sup (1995)
K5 (1995)
M1 (1995)
Lightning (1991p)
Rename buffers

Merged Register File Approach

- Rename buffers and the architectural registers are implemented within the same physical register file.
- The register file obviously needs to provide a large enough number of physical registers to implement both the architectural and rename registers.
- A free *physical* register is allocated for the destination *architectural* register of each instruction.
- The actual allocation of registers is tracked in a *mapping table*.
- Need a scheme to reclaim the physical registers no longer in use.


ad r2, ..., ...
Architectural
reg. numbers
Physical
reg. numbers
Entry
valid
RB-
index
0
1
2
3
4
5
31
1
10
11
3
0
Mapping
table
r2:
Physical register file
p3
Allocated
to r2


ad, r2, r0, r1:
Entry valid
RB- index
r0:
r1:
0
1
2
3
4
5
31
1
1
1
0
0
10
11
3
Mapping table
0
1
2
3
10
11
p10
p11
Physical register file

Separate Rename Buffers

In the other types of renaming, the rename buffers are implemented separately from the architectural registers either as a *rename register file*, as an extension of the reorder buffer (ROB), or as part of the DRIS.

- Each time a destination register is referred to, a new rename register is allocated to it. This allocation remains valid until either:
  1. a subsequent instruction refers/writes to the same destination register, when the architectural register will then be reallocated; or until
  2. the instruction which uses that particular destination register completes (*retires*) and the allocation becomes invalid.

Separate Rename Buffers$_{(cont)}$

- In machines which use a ROB (or the DRIS) for renaming, each instruction is allocated a separate entry. Thus it is quite natural to store the generated result of that instruction in the entry as well.
- In all three cases, results are held in the respective rename buffer until their retirement. During retirement the content of the rename buffer (rename register entry, ROB entry or DRIS entry) is written back into the architecture register file, and the buffer is freed for further use.
- If distinct register files are used for FX and FP data then two separate renaming schemes are used unless renaming takes place within the ROB or DRIS.

Accessing Renaming Buffers

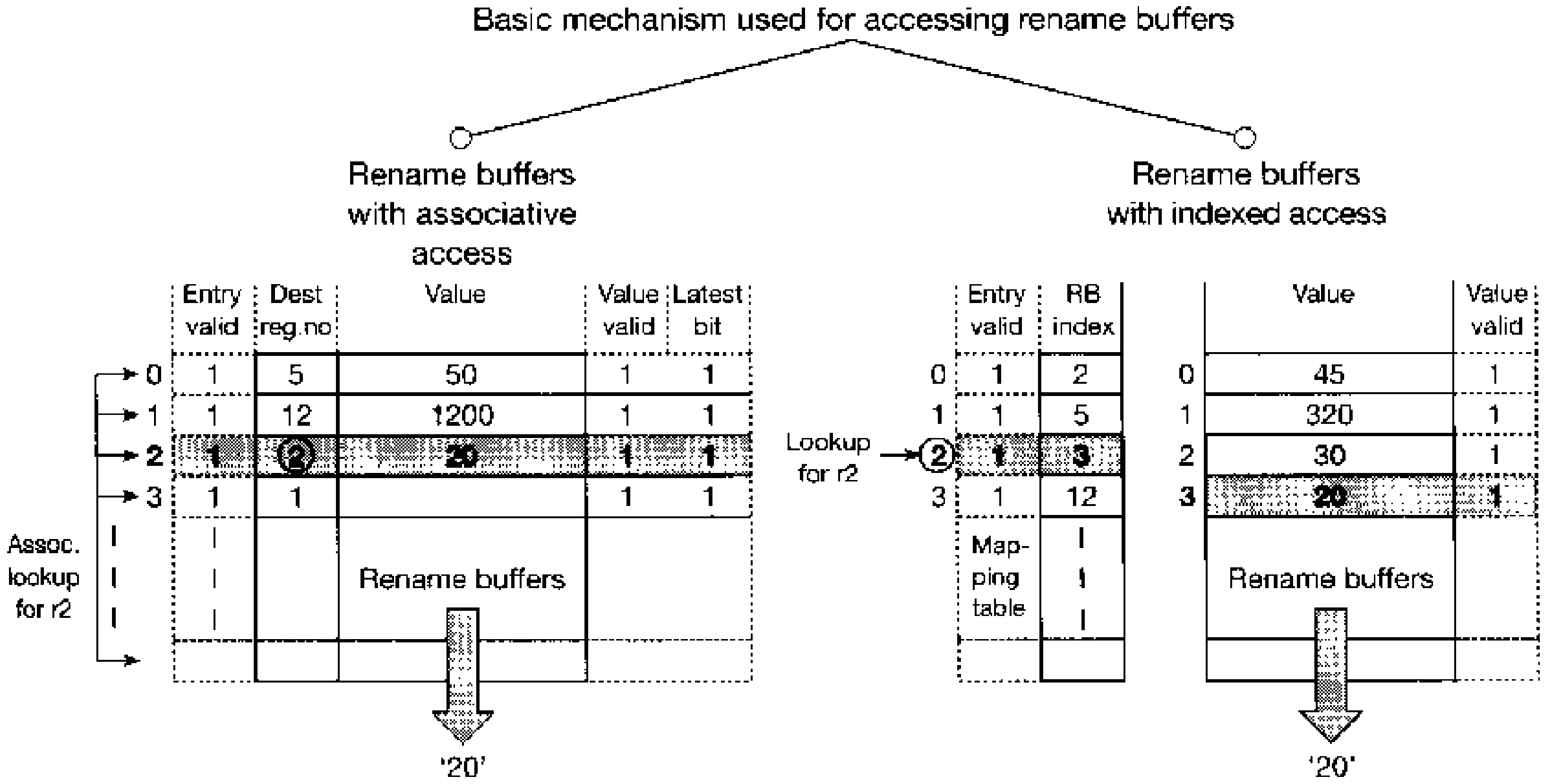
Rename buffers need to be accessed for several purposes, such as to fetch operands, to update them or to deallocate them. Operands are accessed using one of two different access mechanisms:

- Rename buffers with *associative access* typically hold three kinds of information: the destination register number; their values and necessary status information. When a register value is to be fetched, all entries are looked up associatively to find the particular entry whose destination field matches the required register.

Accessing Renaming Buffers$_{(cont)}$

- With the *indexed access mechanism*, a mapping table is used to obtain the actual index into the rename buffer file. The mapping mechanism provides a unique index to the rename buffer file which always corresponds to the most recent instance of the destination register concerned.

Note that a source operand may exist in both the architectural and the rename register file, if so priority is given to the rename register during operand access.

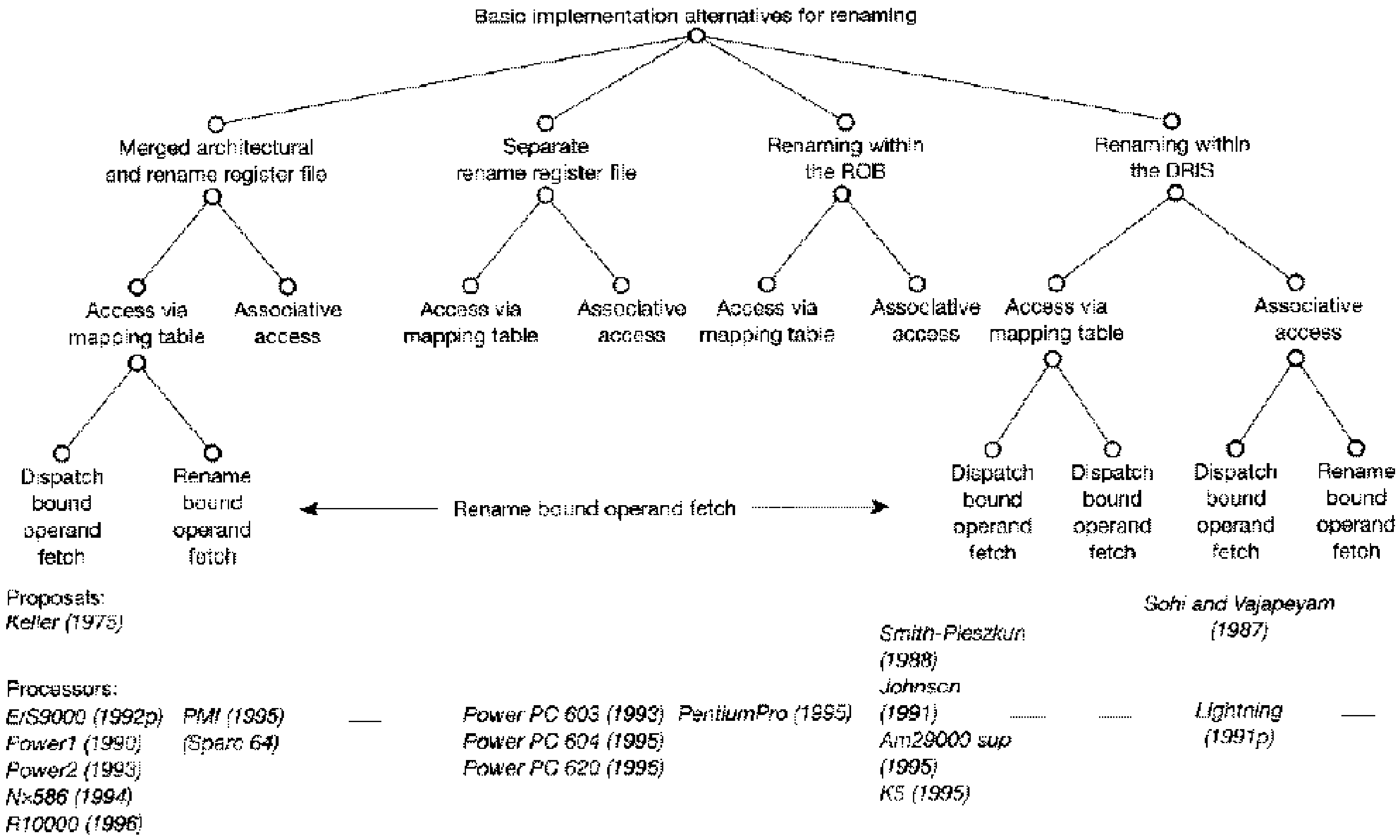

Basic mechanism used for accessing rename buffers
Rename buffers with associative access
Rename buffers with indexed access
Entry valid
Dest reg.no
Value
Value valid
Latest bit
0 1 5 50 1 1
1 1 12 1200 1 1
2 1 2 20 1 1
3 1 1 1 1
Assoc. lookup for r2
Rename buffers
'20'
Lookup for r2
Entry valid
RB index
0 1 2
1 1 5
2 1 3
3 1 12
Map-ping table
Value
Value valid
0 45 1
1 320 1
2 30 1
3 20 1
Rename buffers
'20'

Further Issues

- In terms of operand fetch policy, the same options exist as those for shelving – a *rename (or issue) bound* or a *dispatch bound operand fetch policy*.

- 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.

- A check needs to be carried out for data dependencies among the instructions issued in the same cycle.


Basic implementation alternatives for renaming
Merged architectural and rename register file
Separate rename register file
Renaming within the ROB
Renaming within the DRIS
Access via mapping table
Associative access
Access via mapping table
Associative access
Access via mapping table
Associative access
Access via mapping table
Associative access
Dispatch bound operand fetch
Rename bound operand fetch
Rename bound operand fetch
Dispatch bound operand fetch
Dispatch bound operand fetch
Dispatch bound operand fetch
Rename bound operand fetch
Proposals:
Keller (1975)
Smith-Pleszkun (1988)
Johnson (1991)
Am29000 sup (1995)
K5 (1995)
Sohi and Vajapeyam (1987)
Processors:
E/S9000 (1992p)
Power1 (1990)
Power2 (1993)
Nx586 (1994)
R10000 (1996)
PM1 (1995)
(Sparc 64)
Power PC 603 (1993)
Power PC 604 (1995)
Power PC 620 (1995)
PentiumPro (1995)
Lightning (1991p)

Parallel Execution

- When instructions are executed in parallel, they will generally finish out of order.
- The term "finish" indicates that the required operation of the instruction has been accomplished. An instruction "completes" when the result is written to ROB and is "retired" or "committed" when the result is written back to the architectural register and the entry removed from the ROB.
- Some processors avoid out-of-order execution by enforcing in-order issue and forcing all EUs to have equal execution times, i.e. effectively operating lock-stepped pipelines.