Lecture 9

Higher performance processor design

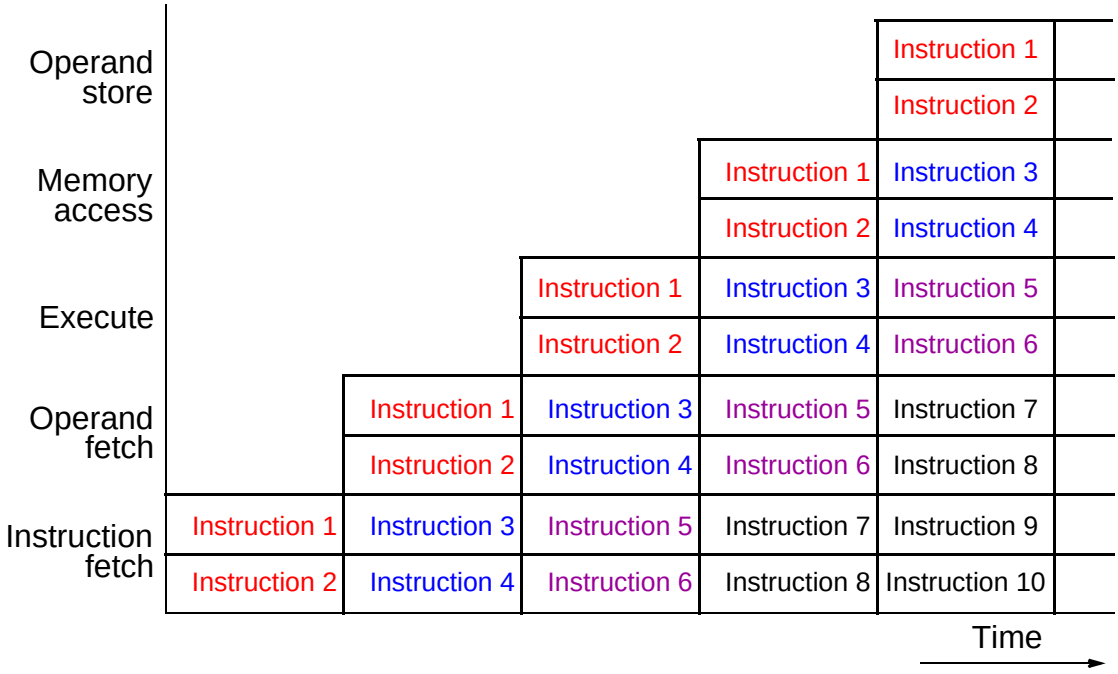
Superscalar processors

A conventional “scalar” processor executes scalar instructions, i.e., instructions operating upon single operands such as integers.

A *superscalar* processor is a processor which executes more than one (scalar) instruction concurrently.

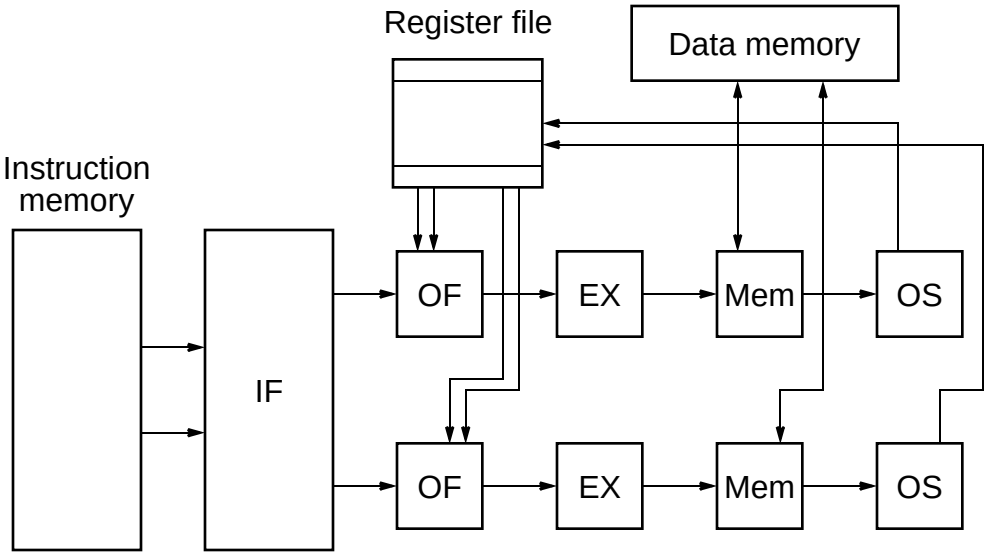
Achieved by fetching more than one instruction simultaneously, and then executing more than one instruction simultaneously.

Superscalar processor timing with two pipelines



Slide 245

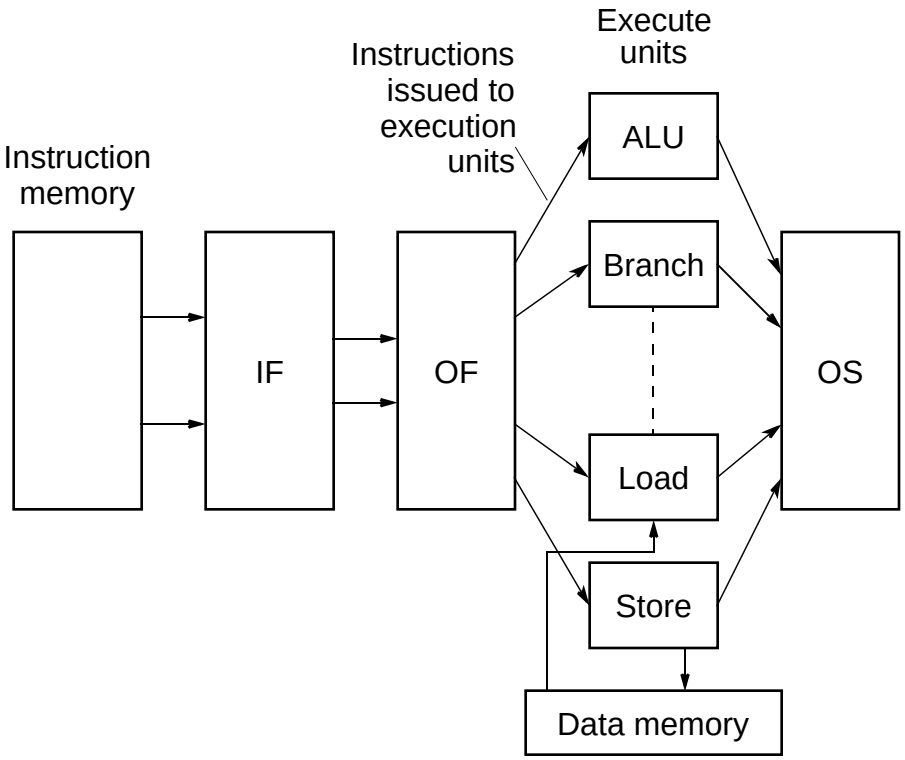
Dual pipeline processor



Example of this concept - Original Pentium processor

Slide 246

Superscalar design with specialized execution units

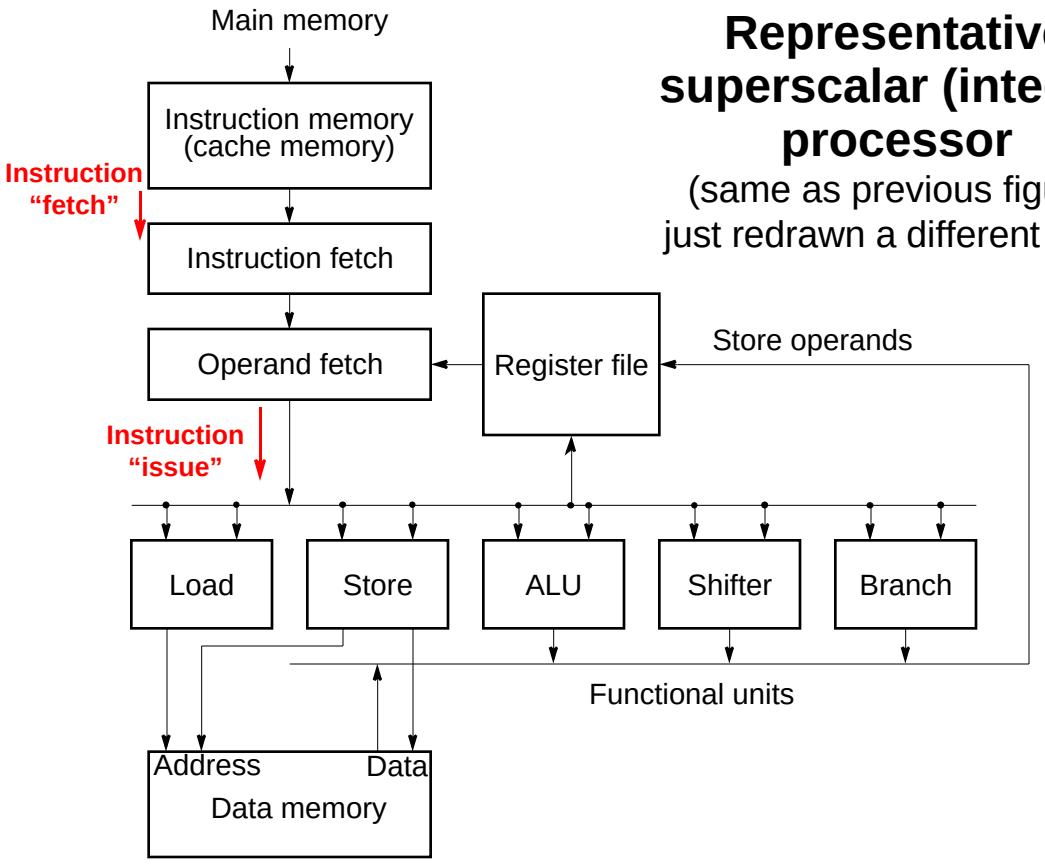


Example - Pentium Pro/II processors and most recent processors.

Slide 247

Representative superscalar (integer) processor

(same as previous figure, just redrawn a different way)



Slide 248

Instruction Fetch

Usually more than one instruction is fetched from the program memory (cache), a complete line in the cache, maybe 4 instructions.

Slide 249

In-order issue of instructions – Instructions sent for execution in program order

Out-of-order issue of instructions – Instructions sent for execution not in program order (instructions allowed to overtake stalled instructions).

Either way, usually instructions may finish execution out-of-order (*out-of-order completion*).

Slide 250

In-order completion rarely enforced. For example in the sequence:

```
MUL R1,R2,R3
ADD R4,R5,R6
```

even if we issue the MUL instruction before the ADD instruction, the MUL instruction is likely to require more cycles and will complete after the ADD instruction.

All dependencies are a definite problem in superscalar processors with their multiple pipelines and out-of-order issue/out-of-order completion.

Slide 251

Resource Conflicts

Factor which usually does not occur in scalar designs but appears in superscalar designs is a resource conflict for a functional unit.

Example

```
ADD R1,R2,R3
SUB R4,R5,R6
```

No instruction dependencies. However suppose only one ALU is provided, responsible for both addition and subtraction. Clearly both ADD and SUB instructions cannot be executed together in such a design. Number of functional units provided will be a compromise between cost and possible resource conflicts.

Slide 252

Out-of-order completion and sequential consistency

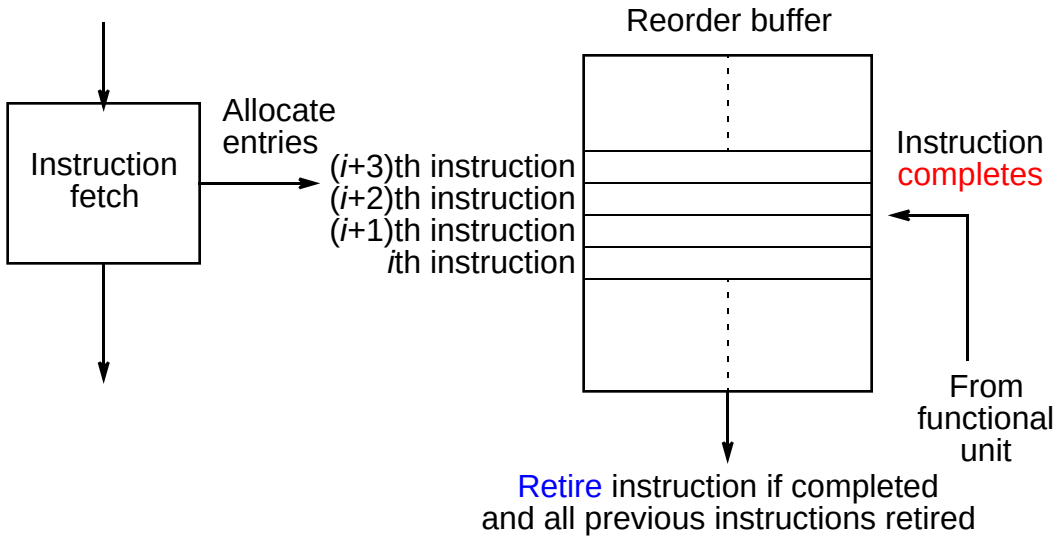
Clearly the final result of the execution of a program must be as though the instructions are executed in program order, i.e. so-called sequential consistency must be preserved.

(This concept also applies to multiprocessor systems, see much later).

Reorder Buffer

General method of achieving sequential consistency in a superscalar processor is through the use of a reorder buffer, which allows instructions to complete out-of-order but their results are caused to be committed to the destination registers in program order, i.e. the results are "reordered" into program order.

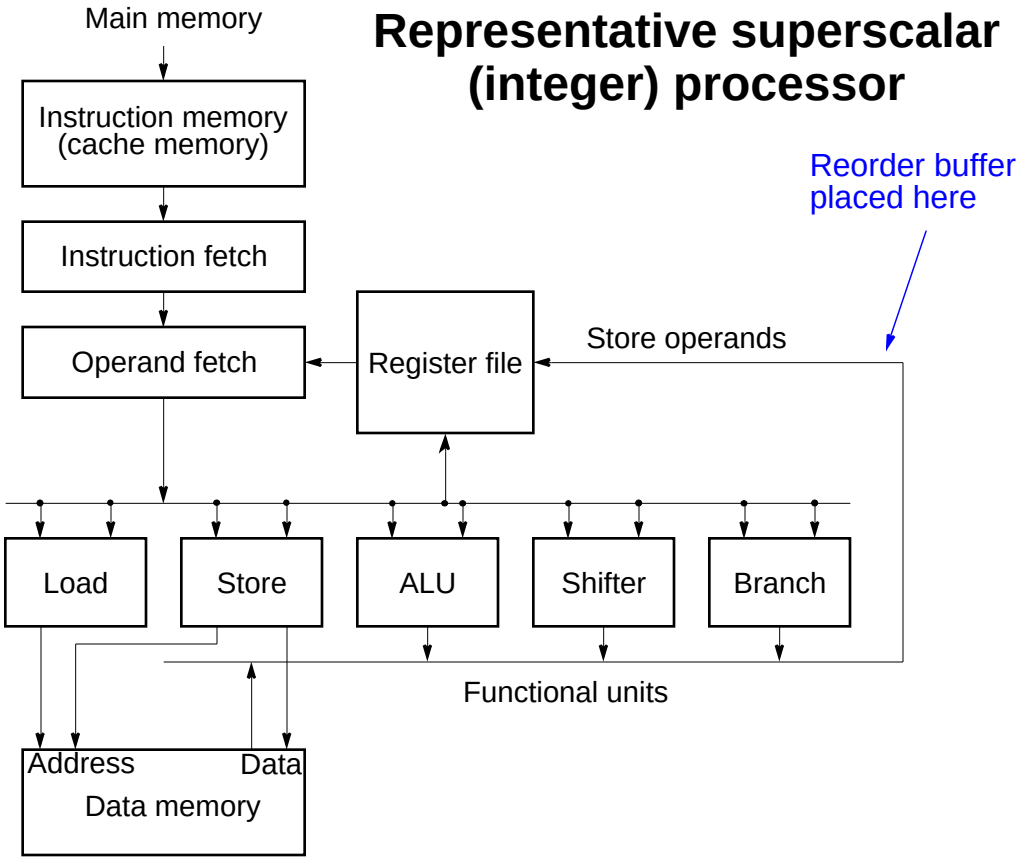
Reorder Buffer - basic idea



Reorder buffer usually implemented as circular buffer with pointer to last entry and pointer to first entry. Contents of reorder buffer depends upon implementation (typically hold results of instructions).

Slide 255

Representative superscalar (integer) processor



Slide 256

Instruction Window

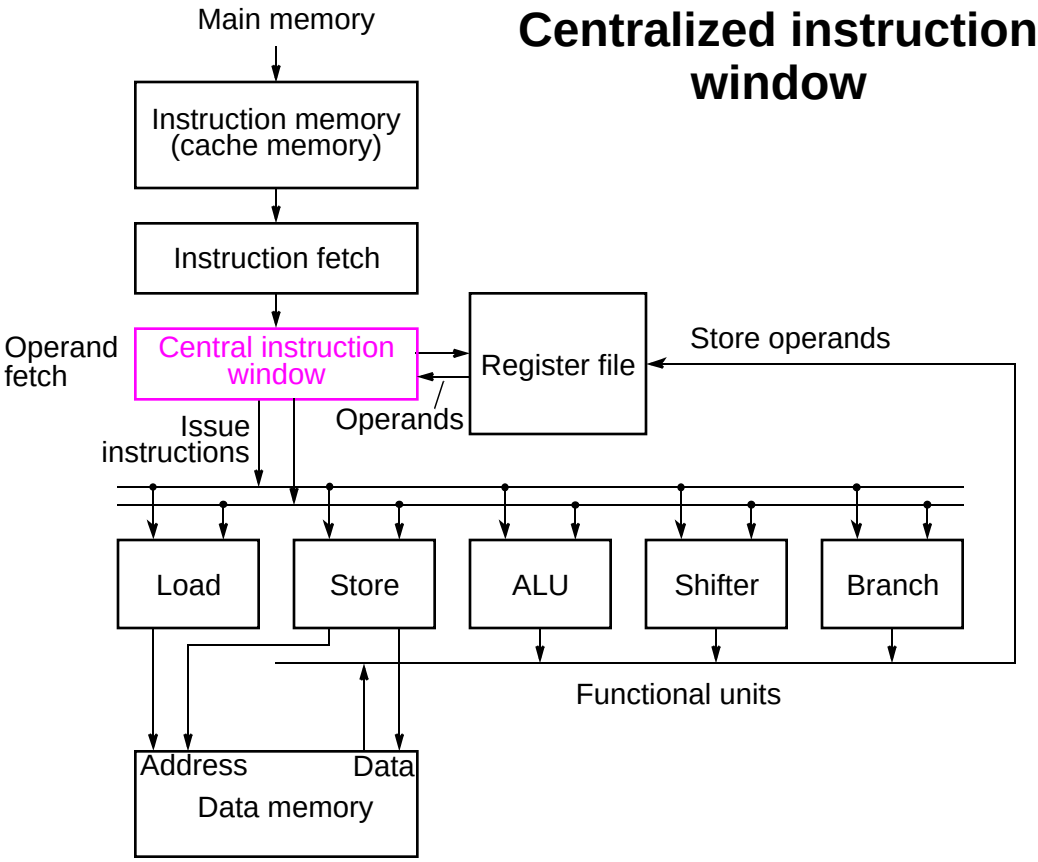
To achieve out-of-order issue, an instruction buffer called an *instruction window* is used. Placed between fetch and execute stages to hold instructions waiting to be executed. Instructions issued from the window whenever it is possible to execute the instructions, which occurs when the operands the instruction needs are available and the functional unit required for the operation is free.

Instruction Window Implementation

The instruction window can be implemented in two ways:

1. Centralized or
2. Distributed.

Centralized instruction window



Instruction window contents

Instr.	Opcode	Destination register	Operand 1	Operand 1 register	Operand 2	Operand 2 register
1	Operation	ID	Operand value	ID		ID
2	Operation	ID	Operand value	ID		ID
3	Operation	ID		ID	Operand value	ID
4	Operation	ID	Operand value	ID	Operand value	ID
5	Operation	ID		ID	Operand value	ID

Example

Suppose the following instruction sequence is fetched:

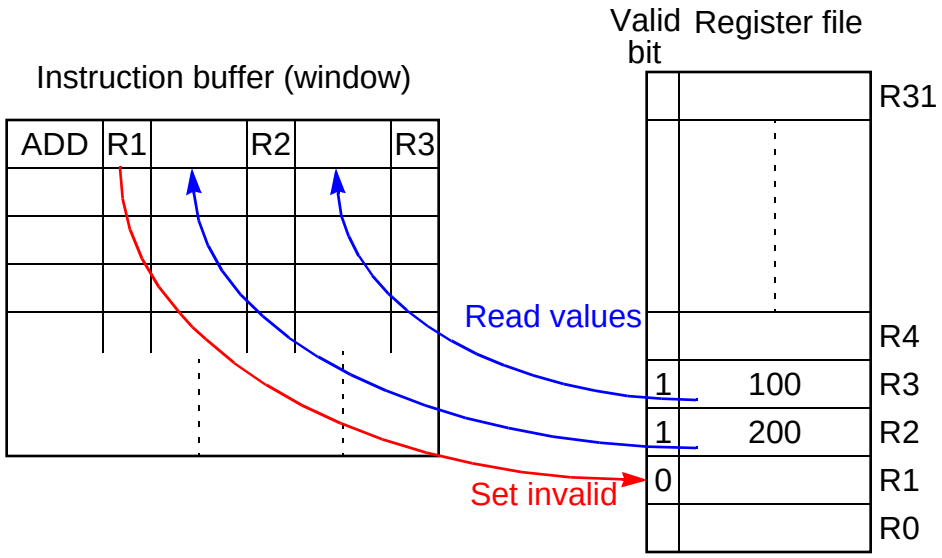
```
1.        ADD R1,R2,R3
2.        SUB R4,R1,R3
3.        MUL R1,R4,R1
```

and instructions are fetched one at a time.

(In superscalar processors, multiple instructions would usually be fetched simultaneously.)

Slide 261

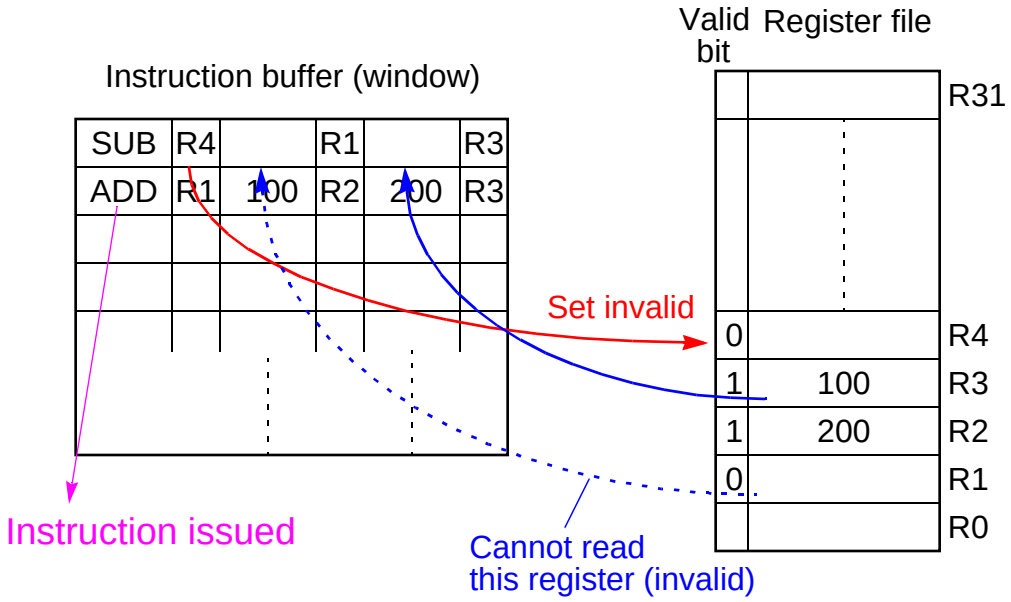
After first instruction loaded:



Values in registers just to give a concrete example.

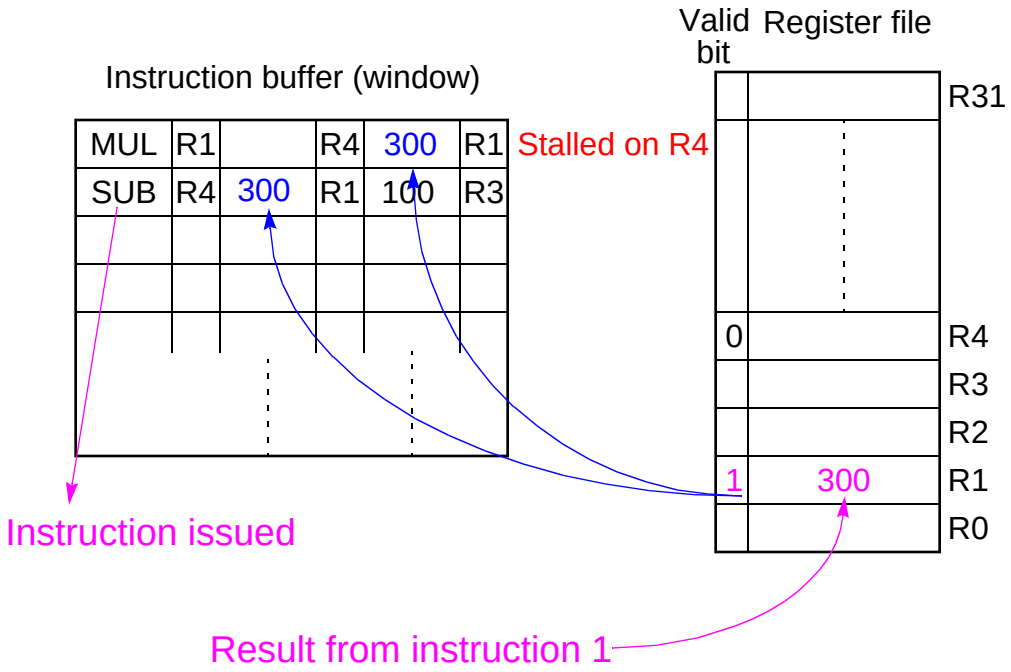
Slide 262

After second instruction loaded:



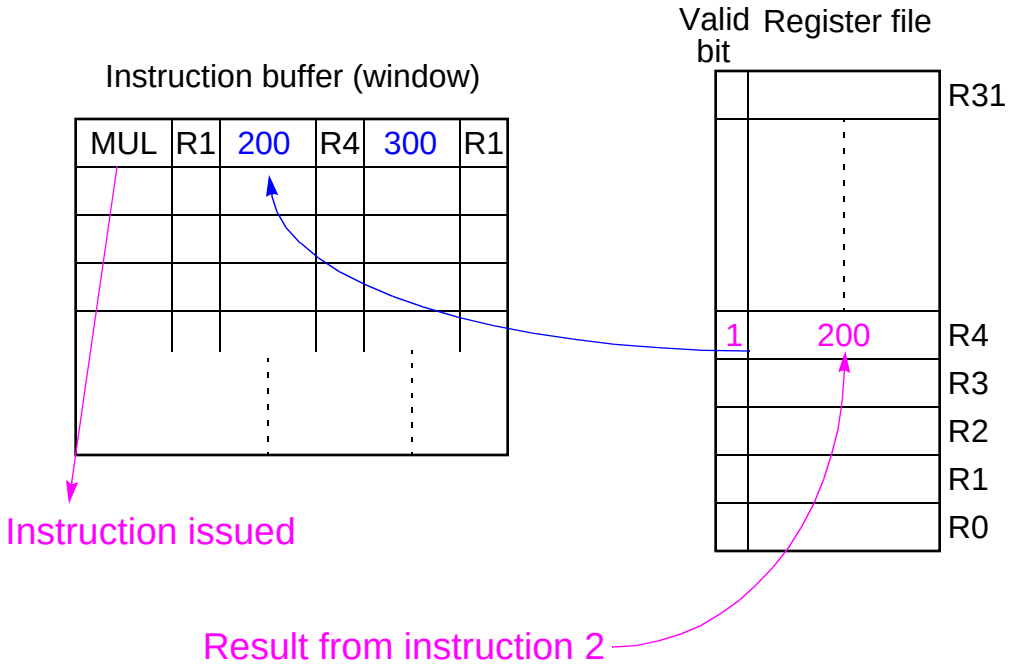
Slide 263

After third instruction loaded and completion of instruction 1:



Slide 264

After completion of instruction 2:



Slide 265

CDC 6600 Scoreboard

CDC 6600 computer system (1964) introduced concept of a *scoreboard* which holds information centrally to control when operands could be fetched and instruction execution can start and when results could be stored. Key points:

- If a structural hazard exists, for example a suitable functional unit is not available, the instruction is stalled.
- The instruction is also stalled if a write-after-write hazard exists, (which is a form of structural hazard, the destination register being reused).
- Otherwise, the instruction is issued to a suitable function unit.

Operands provided when available under control of scoreboard. Similarly, the scoreboard controls when results can be written (when write-after-read hazards are not present)

Slide 266

Original CDC6600 scoreboard only of historical significance and devilishly difficult! Here, we very briefly review the method.

More details can be found in:

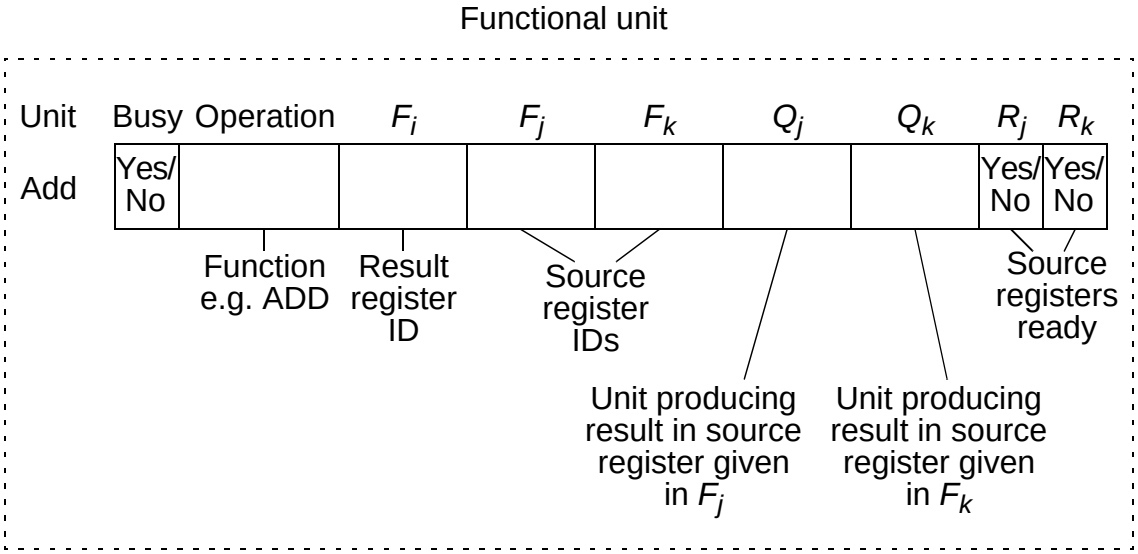
Patterson D. A., and J. L Hennessy, *Computer Architecture A Quantitative Approach 2nd edition*, Morgan Kaufmann, 1996, pp. 240-261.

or:

Thornton, J. E., *Design of a Computer, the Control Data 6600*, Scott, Foresman, Glenview Ill, 1970.

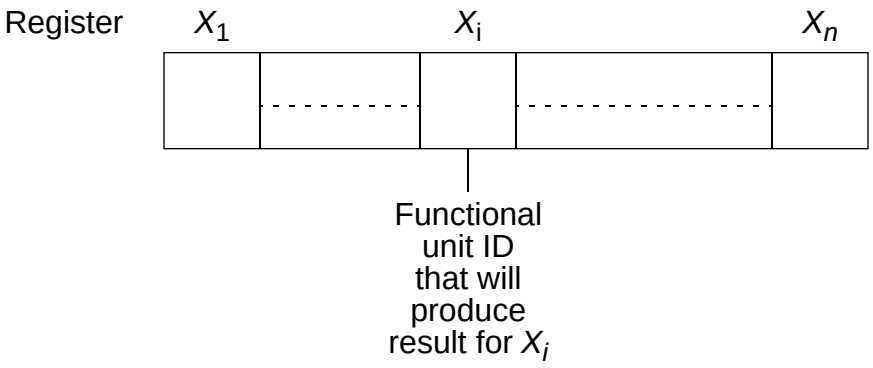
Slide 267

CDC 6600 Scoreboard information on functional unit

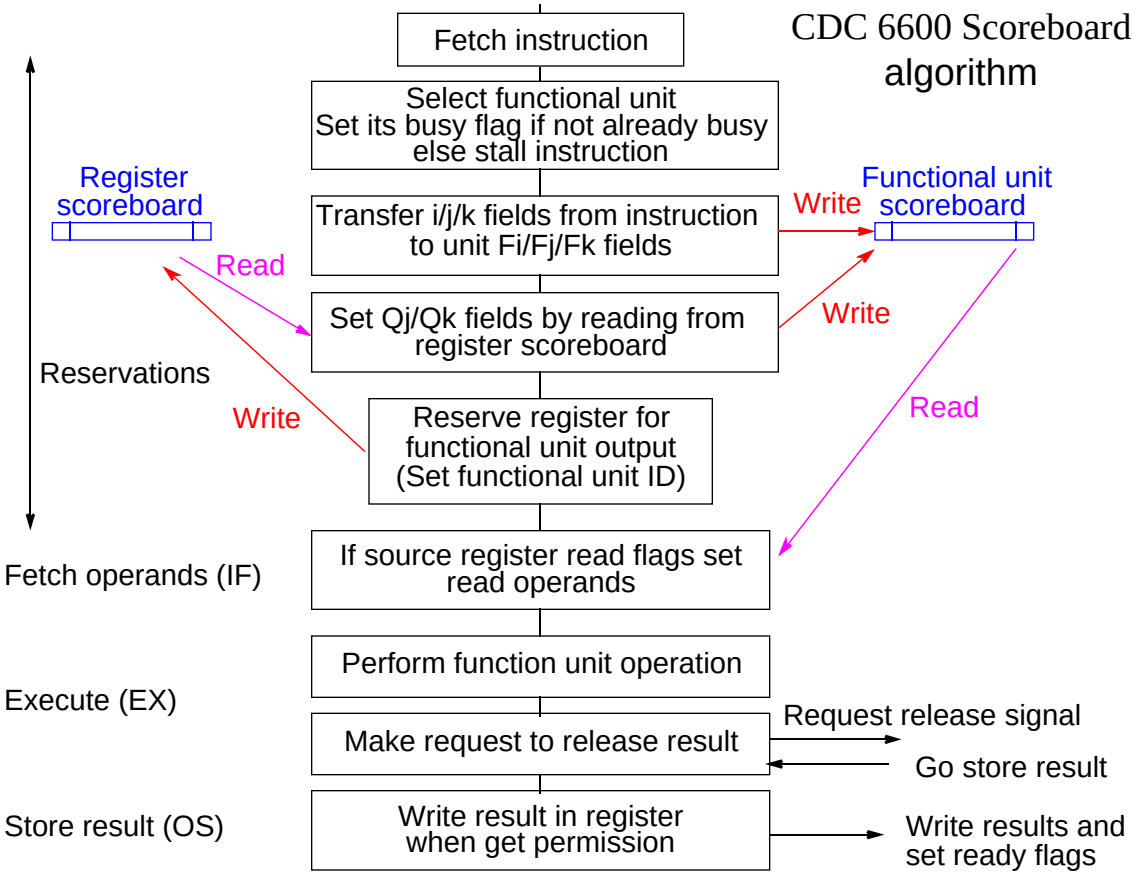


Slide 268

CDC 6600 Scoreboard information regarding source of register results



Slide 269



Slide 270

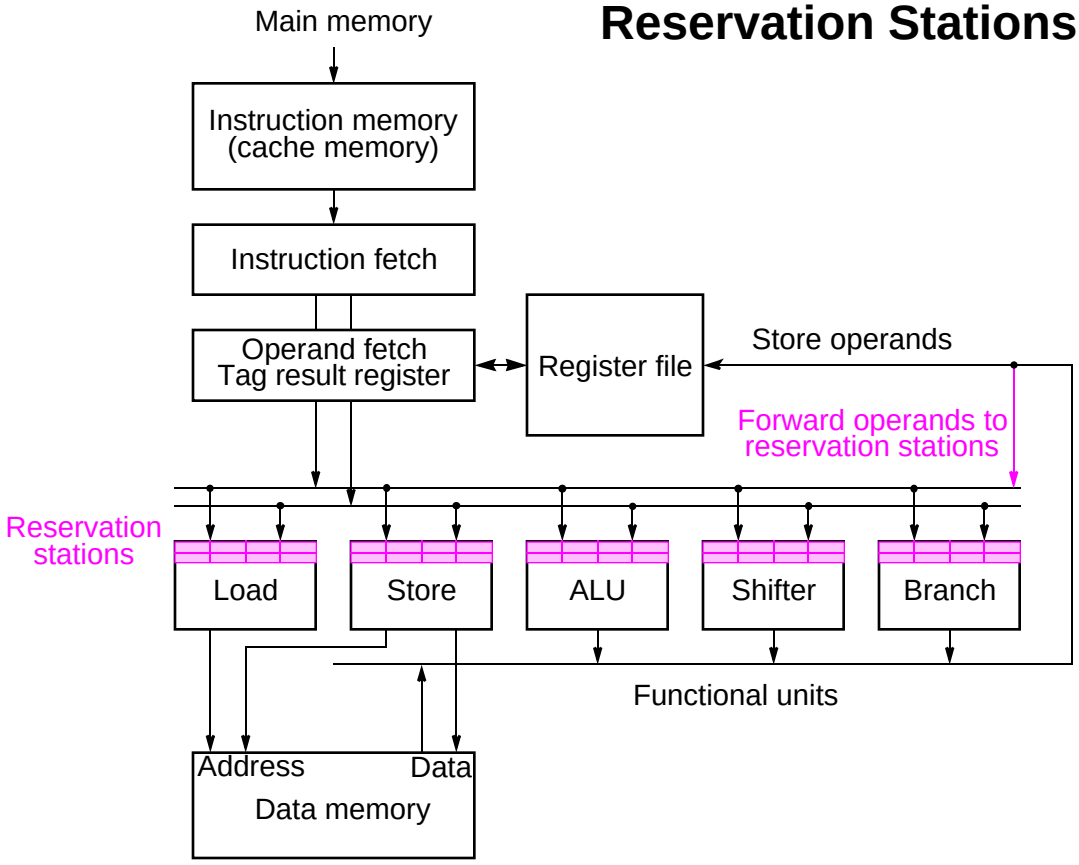
Lecture 10

Distributed Instruction Window Approach

Instruction buffers called reservation stations placed at front of each functional unit.

Slide 271

Reservation Stations



Slide 272

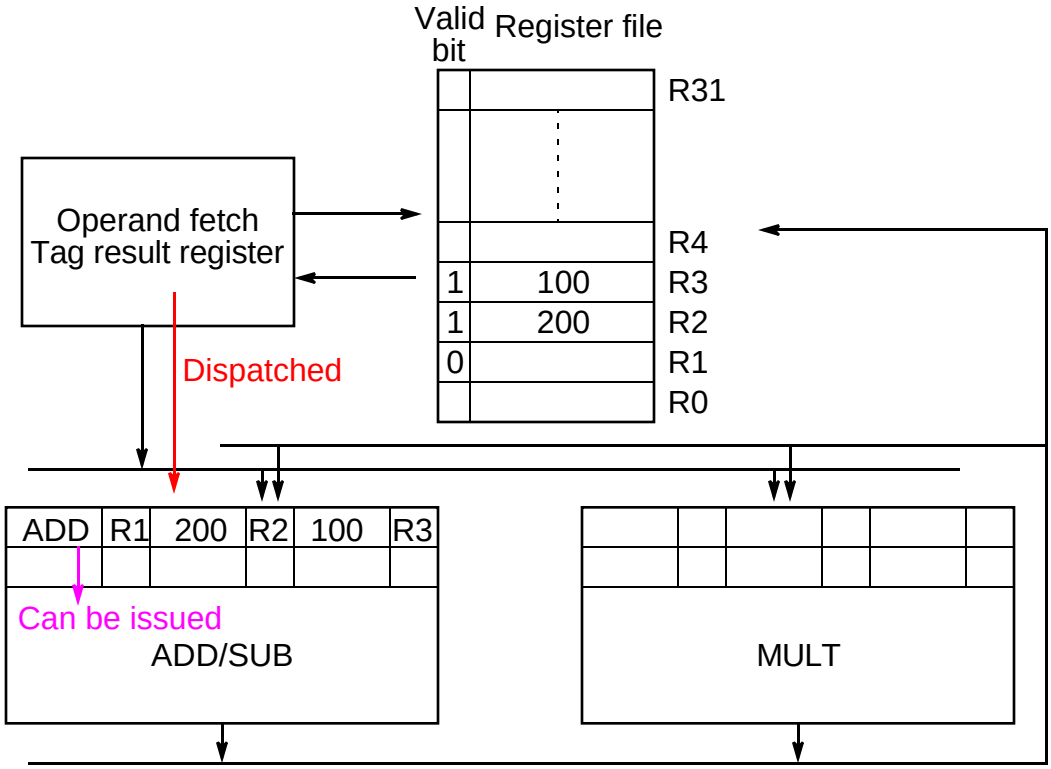
Example

Suppose the same instruction sequence is fetched:

```
1.      ADD R1,R2,R3
2.      SUB R4,R1,R3
3.      MUL R1,R4,R1
```

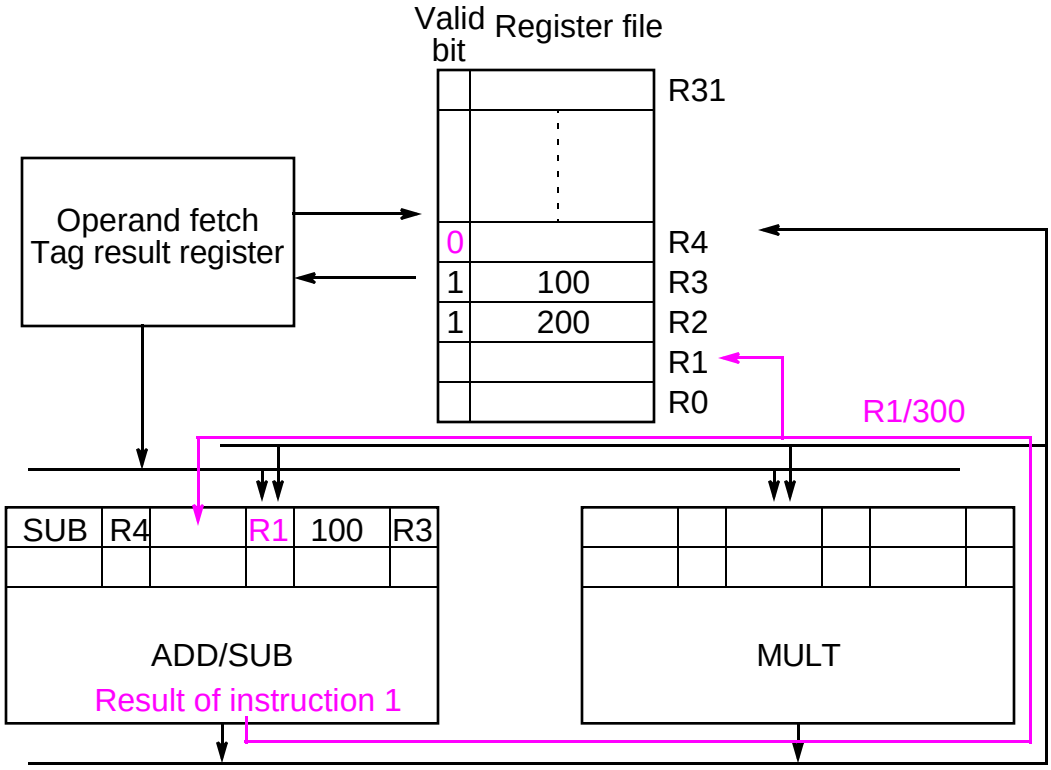
Slide 273

After first instruction fetched and dispatched into a reservation station:



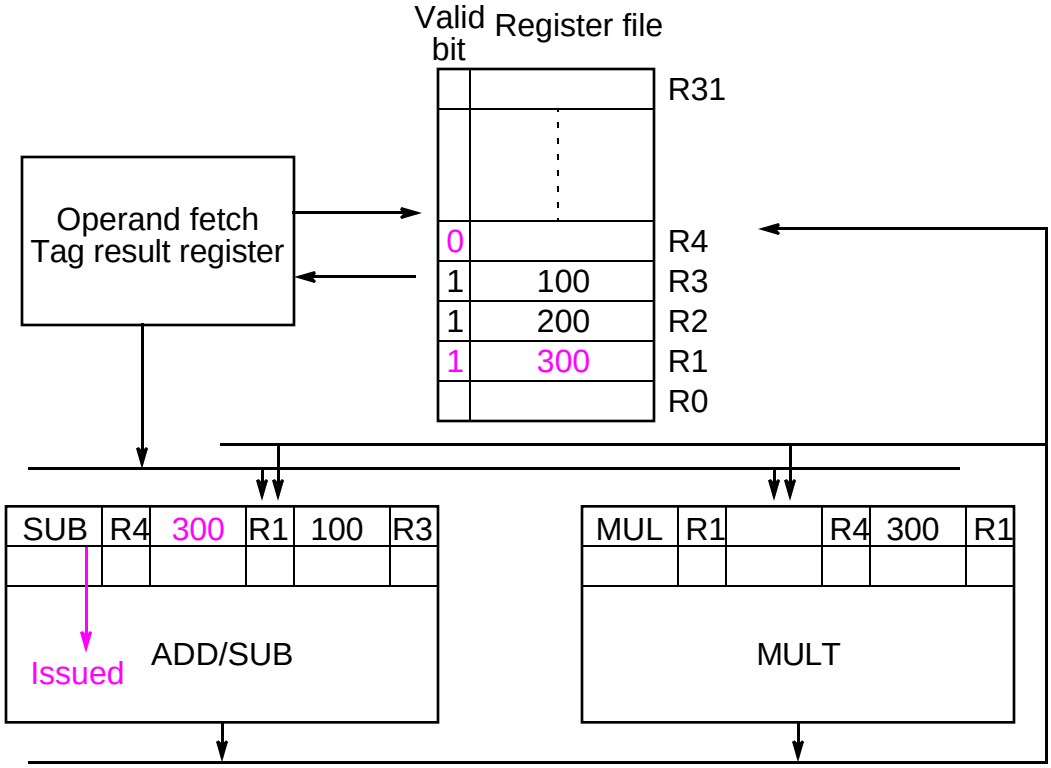
Slide 274

After first instruction completed and second instruction fetched and dispatched into a reservation station:



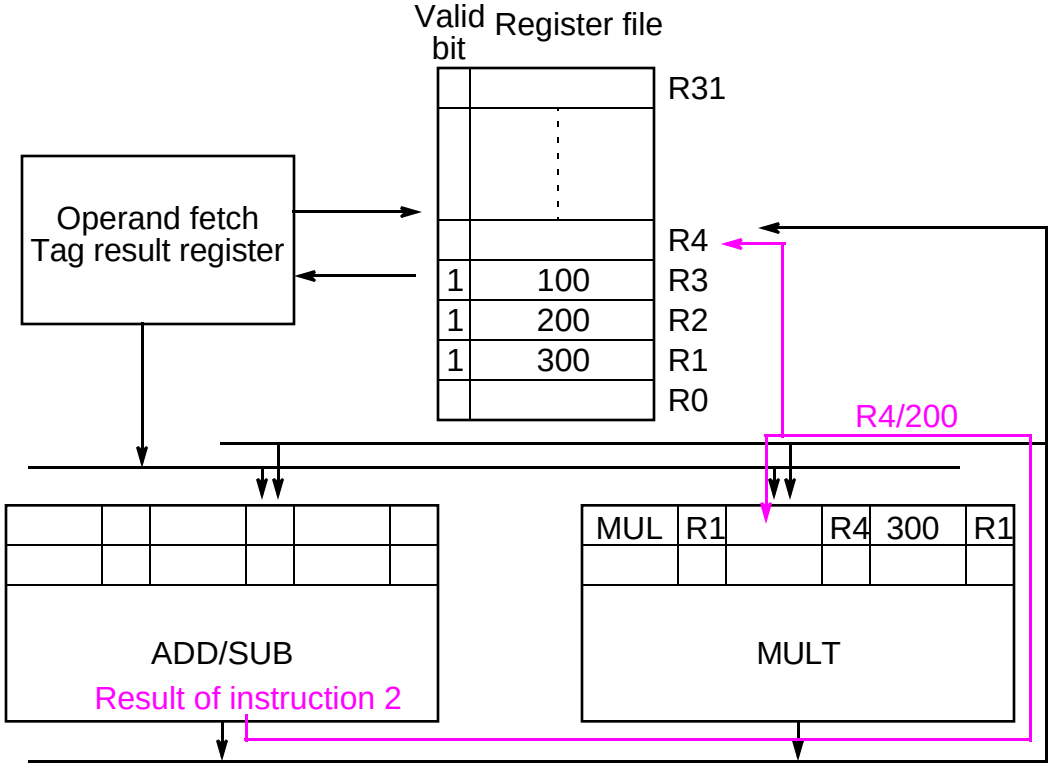
Slide 275

Second instruction issued and third instruction dispatched:



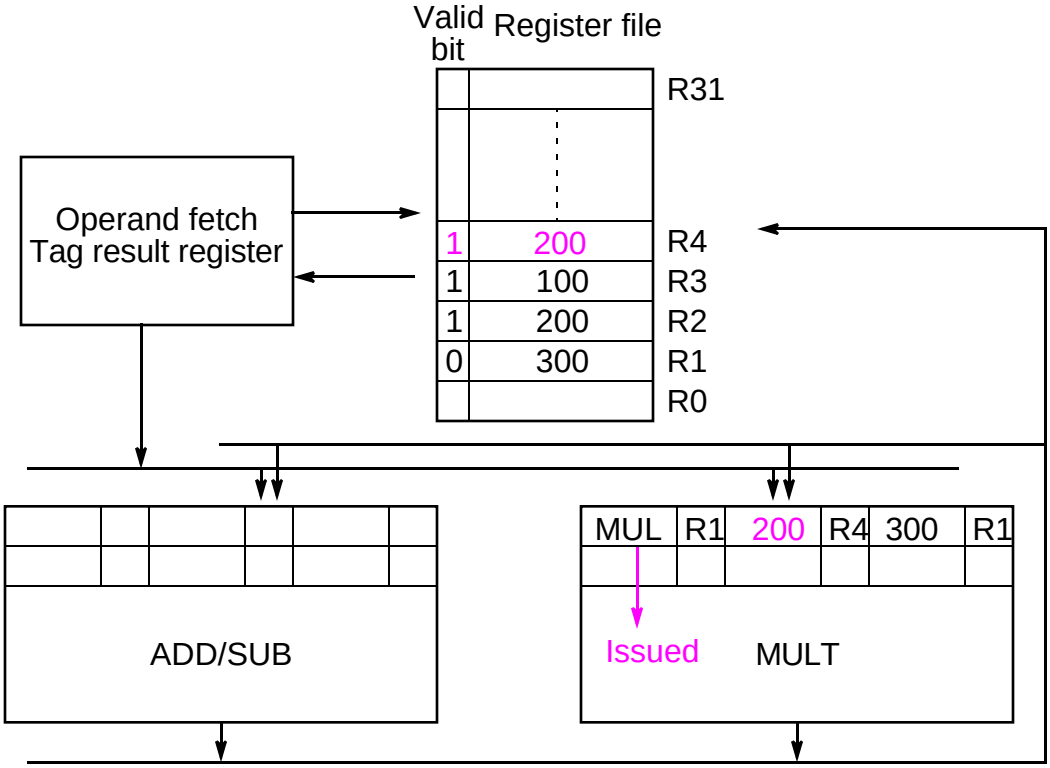
Slide 276

Second instruction completed:



Slide 277

Third instruction issued:



Slide 278

Given sufficient paths, possible for more than one instruction to be dispatched simultaneously - one instruction could be issued within each functional unit simultaneously.

Limitation of valid bit approach for handling dependencies

Only one pending update to each register allowed - i.e. not allowed to dispatch/issue multiple instructions that write to the same register.

Example

```
1. ADD R4, R2, R1
2. MUL R3, R4, R5
3. SUB R4, R6, R7
```

Instruction 2 dependent upon instruction 1 and cannot be issued until instruction 1 writes its result to R4.

Instruction 3 has valid input operands and could be issued before instruction 2 (or even before instruction 1). However, it must not write its result to R4 before instruction 1 writes its results to R4 and instruction 2 has read R4.

Hence does not eliminate output dependencies (write-after-write hazards) - processor simply stalls on these.

Slide 281

To enable more than one instruction dispatched/issued which updates the same register simultaneously would require incredibly complex control! Even the CDC 6600 scoreboard was not able to do that.

Fortunately there is a viable alternative, called register renaming, which is used on all recent processors.

Slide 282

Register renaming

Antidependencies and output dependencies caused by reusing storage locations, i.e., they are resource conflicts.

Consequently can eliminate these dependencies by providing additional storage locations (duplicating resources).

Example

```
ADD R1,R2,R4      ;R1 = R2 + R4
ADD R2,R3,1       ;R2 = R3 + 1
ADD R1,R2,R5      ;R1 = R2 + R5
```

has an antidependency and an output dependency. (It also has a resource conflict if there is only one adder and a true dependency.)

By introducing different registers, R8 and R9:

```
ADD R1,R2,R4;R1 = R2 + R4
ADD R8,R3,1 ;R8 = R3 + 1
ADD R9,R8,R5;R9 = R8 + R5
```

eliminating the antidependency and output dependency.

Clearly we cannot keep creating new registers throughout program.

A solution is to rename registers temporarily when there are first specified as a destination register. R1 might be temporarily be called R1a, R1b, R1c … as R1 is being reused in the program, and similarly for other registers, i.e.

```
ADD R1a,R2,R4
ADD R2a,R3,1
ADD R1b,R2a,R5
```

Assumed the very first instruction in program otherwise R2 and R4 would have new names.

This is known as *register renaming*.

Slide 285

Register Renaming

Register renaming can remove both antidependencies and output dependencies, and is implemented in hardware.

New register instances are created and destroyed when there are no outstanding references to the stored values.

Most effective with small number of main registers as in the Pentium but also widely used on RISCs. An alternative is to provide a very large number of main registers initially as in the Itanium (128 registers) but then factors introduced such needing to specify registers in the instruction and having to save them on interrupts.

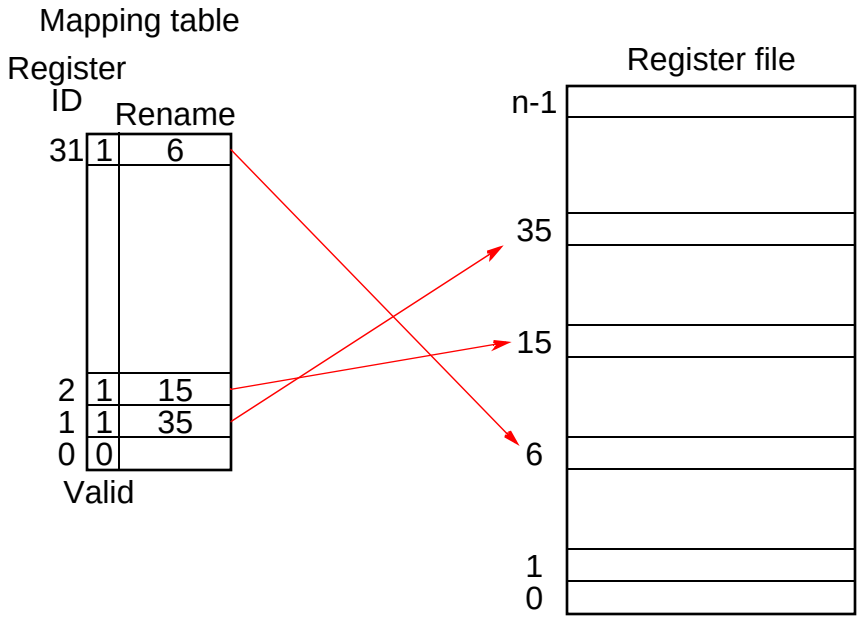
Slide 286

How to implement register renaming

Several methods, including:

- Single register file - usually accessed via a mapping table pointing to dynamically allocated rename registers
- Using a separate register file for renamed registers
- Using a reorder buffer

Single register file with a mapping table



R1 renamed as R35, R2 renamed as R15, R31 renamed as R6.

After destination register renamed, every reference to it as a source register found from mapping table.

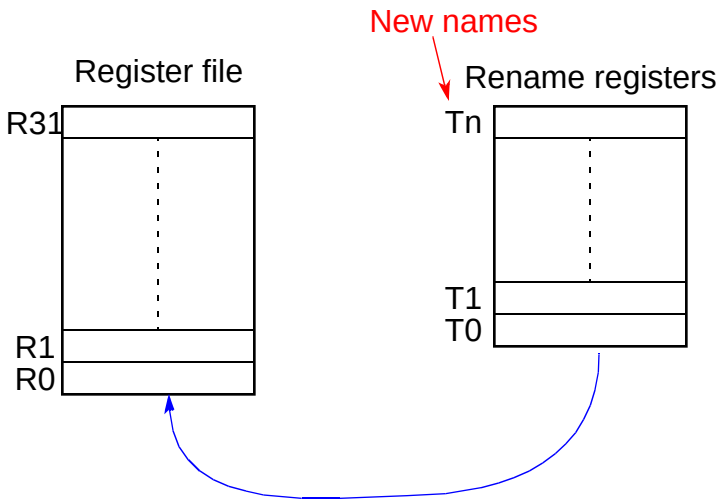
Notice mapping table only gives most recent name. Previous names must be retained as needed, e.g. with pending instructions in instruction buffer. Registers must be released for reuse when not needed. (complex)

Method first suggested by Keller, R. M., “Look-ahead processors,” *Computing Surveys*, Vol 7, Dec., 1975, pp. 177-196.

Examples of processors using mapping table method with a single register file: Power1 (RS 6000), Power2, Nx586, PM1 (Sparc64), R10000

Using separate register file for renamed registers

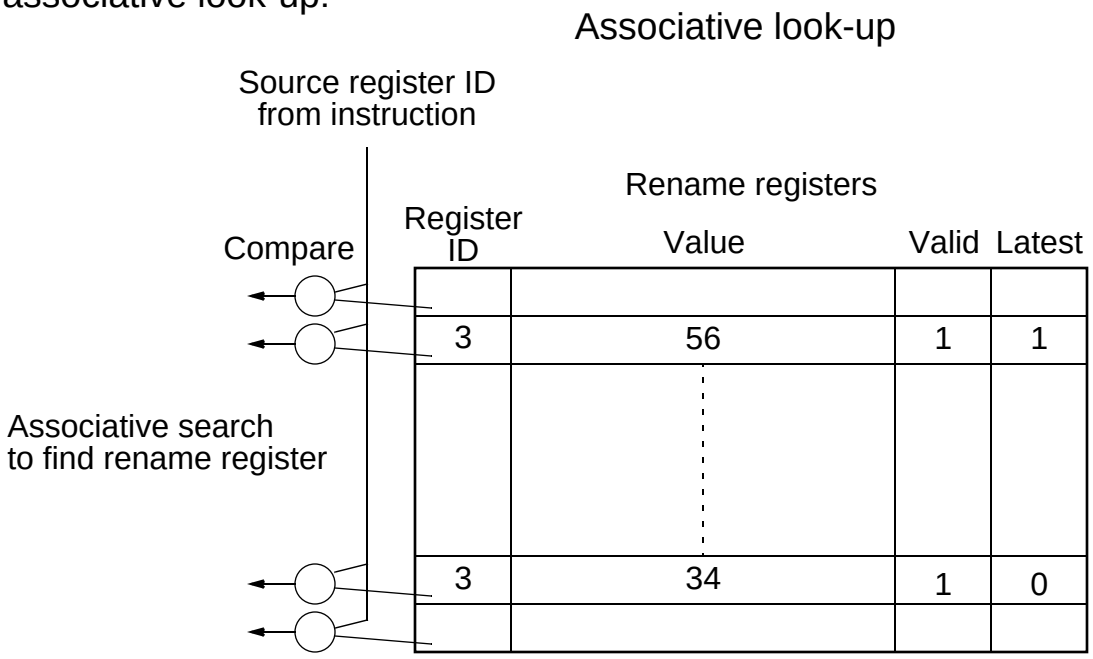
Here have the main register file (say R0 to R31), plus a separate register file for renamed registers, say T0 to Tn. After renaming, rename register file accessed instead of main register file.



When no longer needed, contents copied to main register file and rename register deallocated.

Renaming

Can be done using a mapping table as described previously or by associative look-up:



Slide 291

With the associative look-up method, there may be more than one instance of a register (R3 previously has two entries) The latest bit is set to show the most recent entry.

Examples using separate rename register file

PowerPC 603, 604, 620 using associative look-up

Pentium Pro but with a mapping table

Slide 292

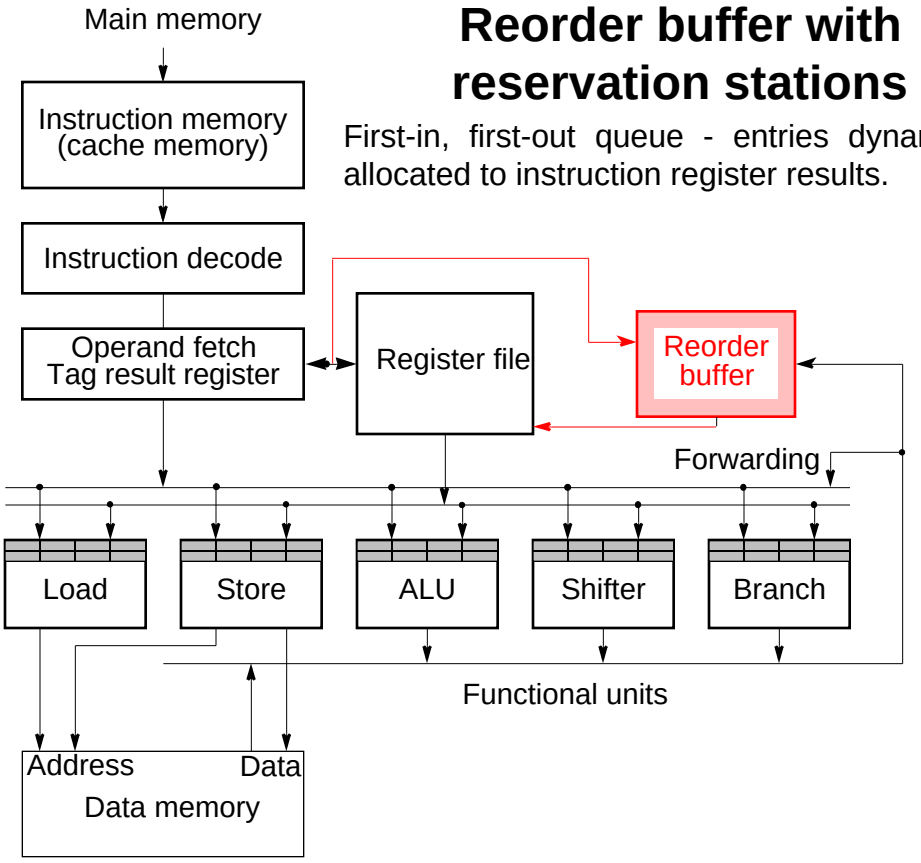
Using Reorder Buffer

Here the task of maintaining sequential consistency and rename registers containing results not yet retired are combined - an attractive solution.

Slide 293

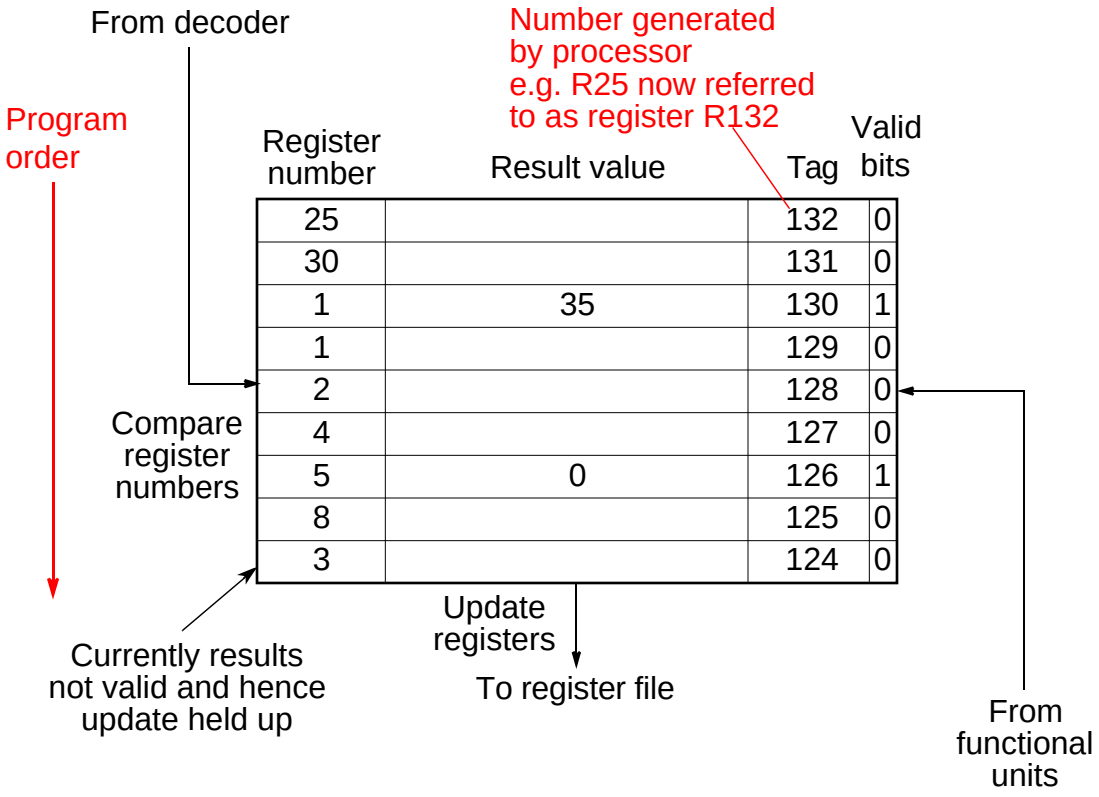
Reorder buffer with reservation stations

First-in, first-out queue - entries dynamically allocated to instruction register results.



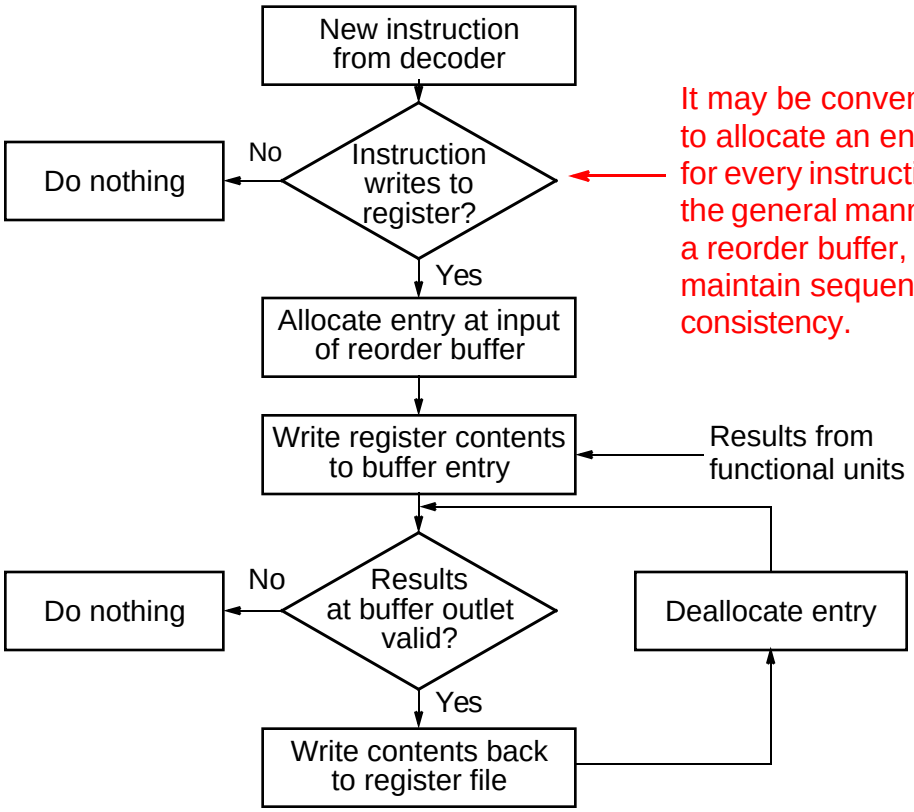
Slide 294

Reorder buffer organization



Slide 295

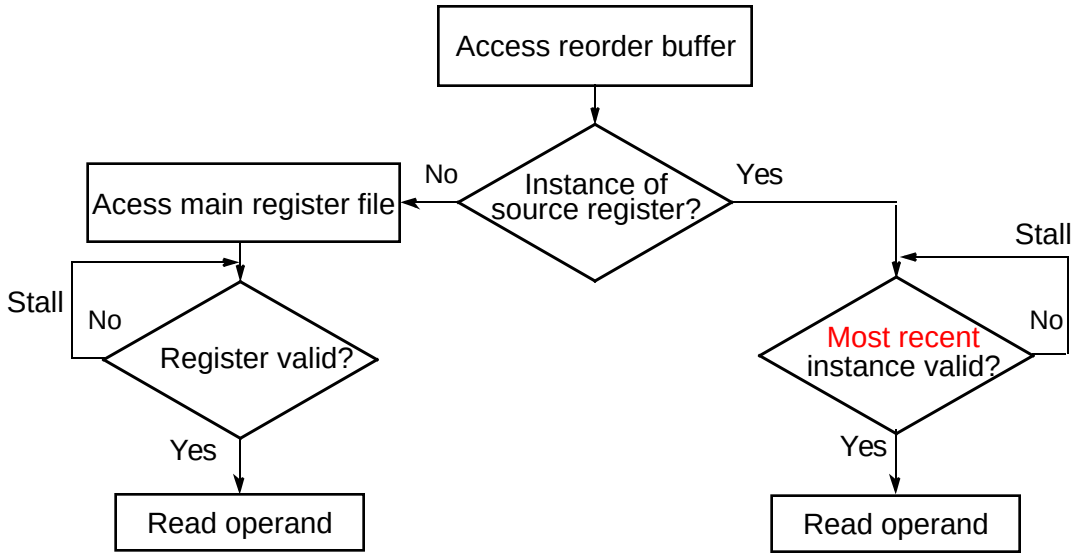
Reorder buffer update algorithm



It may be convenient to allocate an entry for every instruction in the general manner of a reorder buffer, to maintain sequential consistency.

Slide 296

Reorder buffer - reading operands



Quiz

Explain the following phrases:

- Instruction fetch
- Instruction issue
- Instruction dispatch
- Instruction completes
- Instruction retires

Lecture 11

Data Memory

The data memory is usually cache memory which we will discuss later in the course. Cache memory accesses will take more time (cycles) than register-register instructions. The location may not even be in the cache (cache miss), incurring significant extra delay.

Memory Data Hazards

Data dependencies can exist between load and store instructions operating on the same memory location, e.g.:

```
ST [R2],R4
LD R1,[R2]
ST [R2],R4
```

Need to check load addresses with store addresses. (“Dynamic disambiguation of addresses”!) More difficult to handle that register dependencies as addresses are larger. Sometimes only lower significant bits of the addresses are examined.

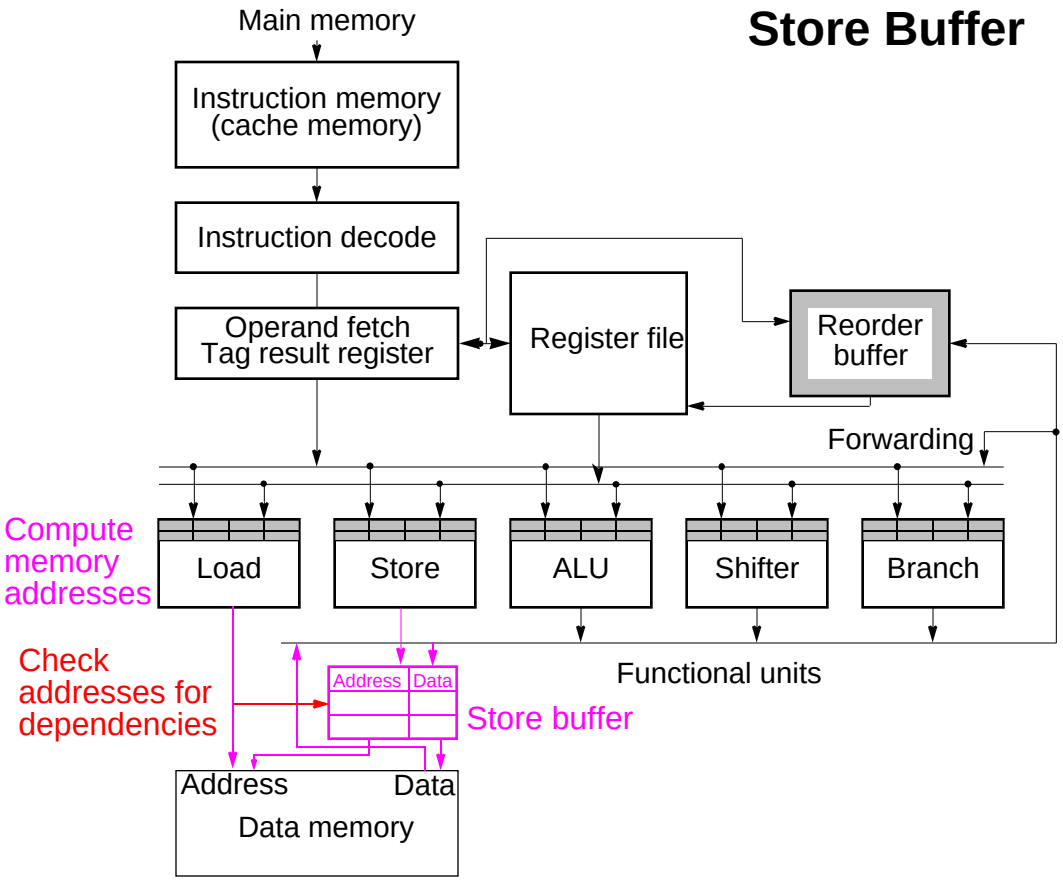
Load/Store Ordering

Generally, stores only performed after all previous instructions have completed (successfully). Maintains data memory (cache) in an in-order state and correct, and to allows error recovery.

Loads may be allowed out-of-order and to overtake subsequent stores if no dependencies with stores. With one data memory, use store buffer to hold pending stores allowing loads to overtake them. Usually more important to read memory (load) than to store result in memory, i.e. loads have priority over stores.

Slide 301

Store Buffer



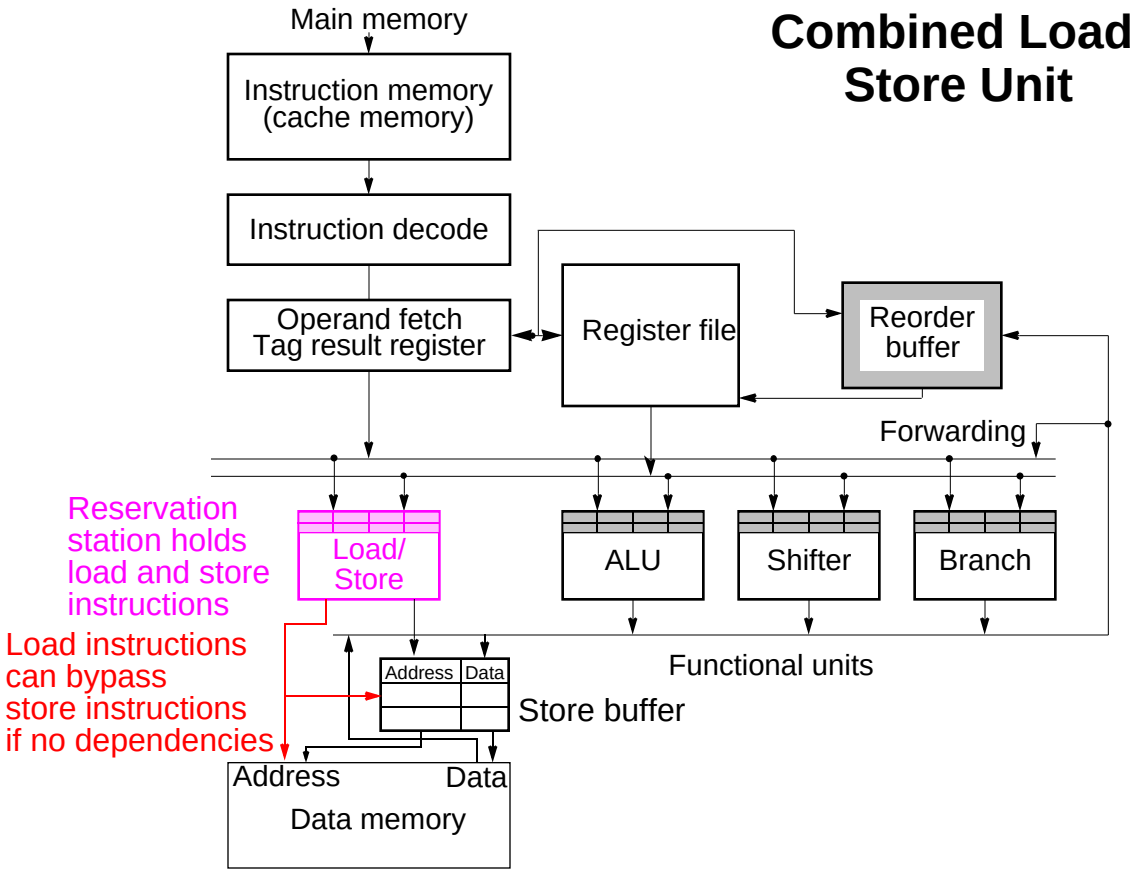
Slide 302

Combined Load/Store Unit

May be better to have combined load/store unit with a single reservation station (or load/store instructions held in a central window). Then can more easily maintain sequential consistency and can share address logic.

Slide 303

Combined Load/ Store Unit



Slide 304

Interrupts

The term *interrupt* is the name given to a mechanism whereby the processor can stop executing its current program and respond to an event. This event could be within the processor or external to the processor.

Typically requires that the program being executed is stopped to execute an interrupt service routine. After this routine executed, original program must be restarted (unless no recover).

Slide 305

Types of Interrupts

Many types and sources:

- Timer interrupts - hardware timers causes program to be interrupted to update time and time sharing.
- Input/Output interrupt - An input/output device requests action through interrupt request signal to the processor.
- Hardware faults - may be detected and generate an interrupt
- Power failure - may be sensed and generate an interrupt
- Virtual memory interrupt - memory not indicated in translation look-aside buffer (see later)
- Instruction error condition -, such as divide by zero. Usually called an exception
- Unimplemented instruction - an instruction in the instruction set not implemented by the particular processor and must be emulated in software
- Software interrupts/traps - a form of procedural call, intended to cause a context switch in a similar fashion as other interrupts.

Slide 306

General classification

Interrupts can be classified as:

- Internal or
- External

and in each case as:

- Error or
- Time critical

Each type may be handled differently.

Question: Classify each of the previous interrupts.

Slide 307

Interrupt handling

When an interrupt occurs in a pipelined processor, instructions in the pipeline will be at various stages of completion.

Highly complicated in a superscalar processor!

Interrupts can be handled as *precise interrupts* or *imprecise interrupts*.

Slide 308

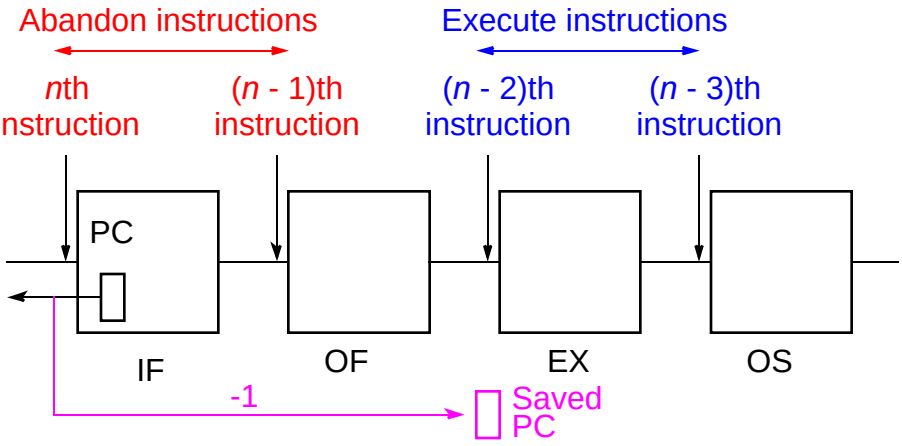
Precise Interrupts

In *precise interrupts*, interrupt takes effect at an exact point within program. The following three conditions must be satisfied:

- All instructions issued prior to instruction being interrupted are completely executed.
- All instructions issued after instruction being interrupted abandoned. Process state must not have been modified by these instructions.
- The interrupted instruction either abandoned (without modify the process state) or allowed to be completely executed.

Sufficient information must to stored to enable the processor to restart at the exact point where it was interrupted.

Interrupt mechanism in a simple pipeline



Need to save program counter to be able to return from the interrupt.

Imprecise interrupt

Precise interrupts can be very difficult and expensive to implement in a superscalar processor.

For an imprecise interrupt, not all information is stored to enable the processor to restart exactly where it was interrupted. Maybe ok if one doesn't expect to return to the program after the interrupt.

Unfortunately this approach is not satisfactory for many sources of interrupt.

Slide 311

Precise interrupt handing in a superscalar processor

Possibilities:

1. Force in-order completion - then service interrupt at the end of current instruction - incurs significant performance overhead, and generally not acceptable

2. Out-of-order completion - store copies of contents of registers before they are overwritten.

Slide 312

Precise Interrupts with out-of-order completion

Three ways:

- History file - contains previous register contents. Then can backtrack to these values on an interrupt.

- Future file - contains updated values that will be placed into registers when all previous instructions have completed.

- Reorder buffer - a form of future file that maintains sequential consistency. (Reorder buffer originally proposed for this purpose.)
-

Provide a roll-back to a certain place in code.

Slide 313

Superscalar Processor Case Studies

The following has been extracted from:

Advanced Computer Architecture A Design Space Approach by D. Sima, T. Fountain, and P. Kacsuk, Addison-Wesley, 1997, pages 280-293.

for educational use only.

Slide 314

Processors

- R10000
- PowerPC 620
- Pentium Pro

MIPS R10000

- 4-way superscalar processor with maximum dispatch rate of five
- Instruction predecoding
- Reservation stations - three groups
- Renaming by merged architectural and rename register file
- Sequential consistency preserved with a reorder buffer

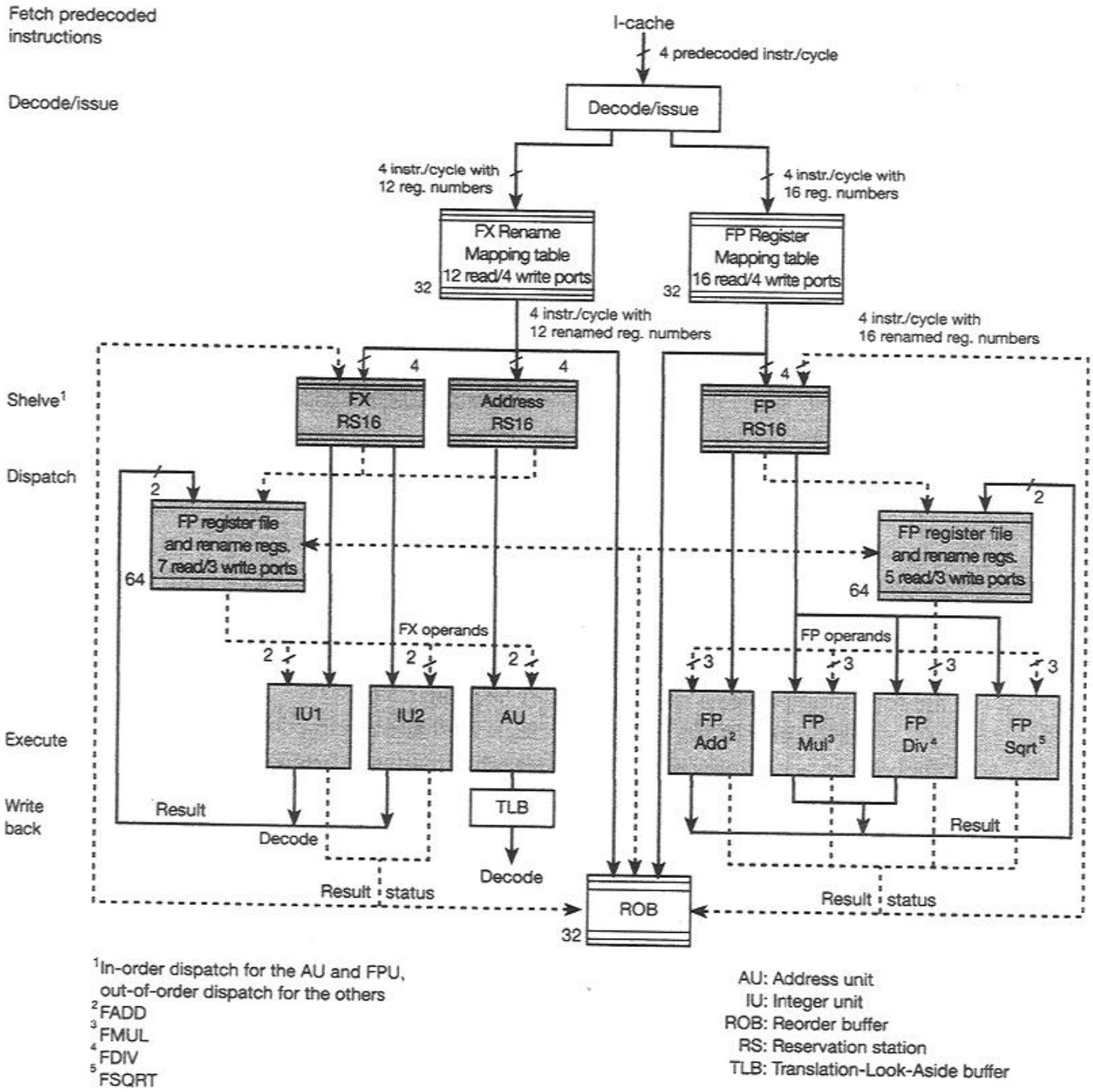


Figure 7.67 Core part of the microarchitecture of the R10000.

PowerPC 620

- Four-way superscalar processor
- Individual reservation stations
- Renaming by separate architectural and rename registers
- Sequential consistency preserved with a reorder buffer

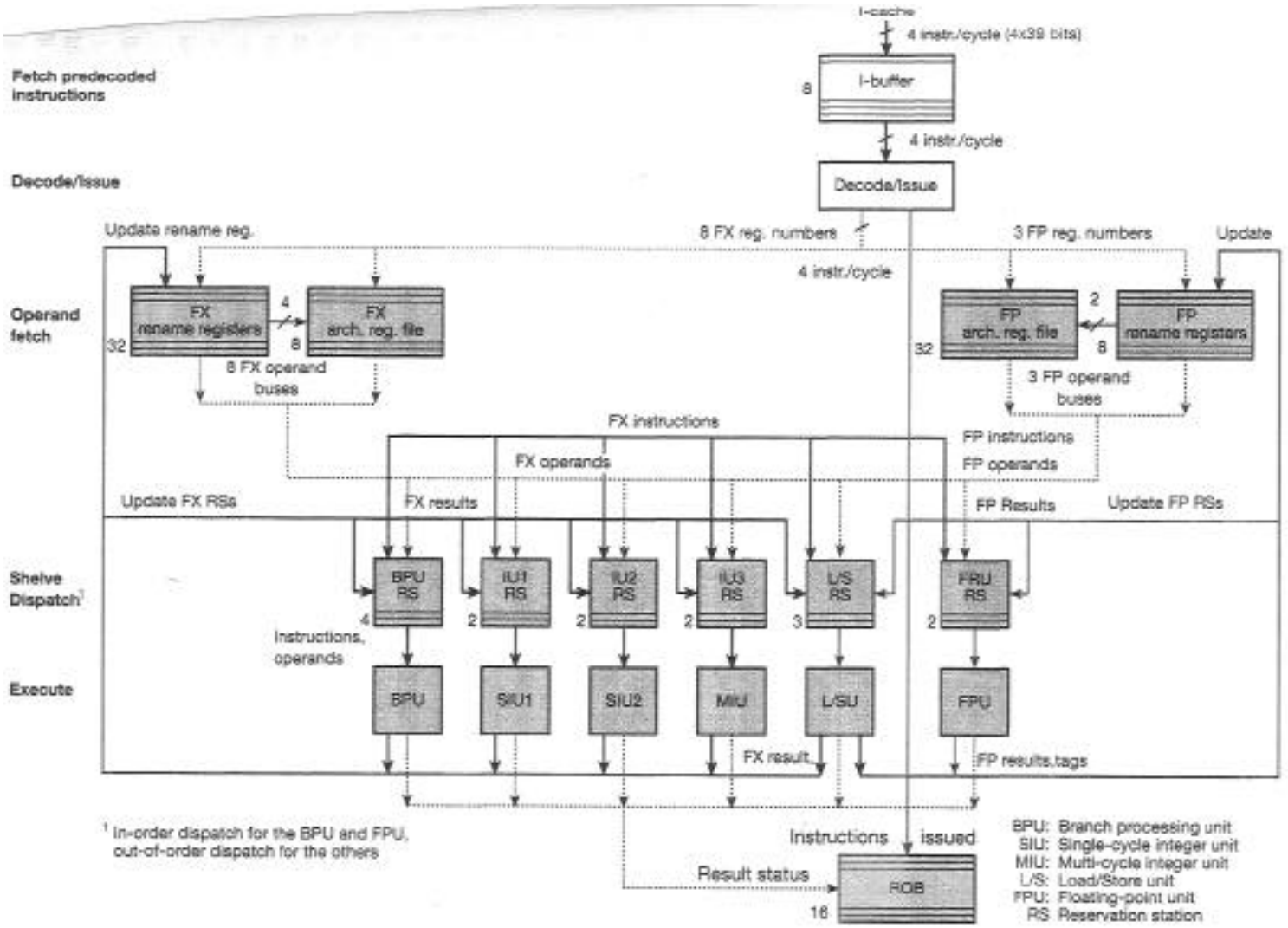


Figure 7.69 Core part of the microarchitecture of the PowerPC 620.

Slide 319

Pentium Pro

- Superscalar CISC processor with RISC core
- Issues three RISC operations per cycle and dispatches up to five RISC operations per cycle
- Unified central reservation station with 20 entries for all types of instructions
- Strict sequential consistency preserved with a reorder buffer
- Renaming done in reorder buffer

Slide 320

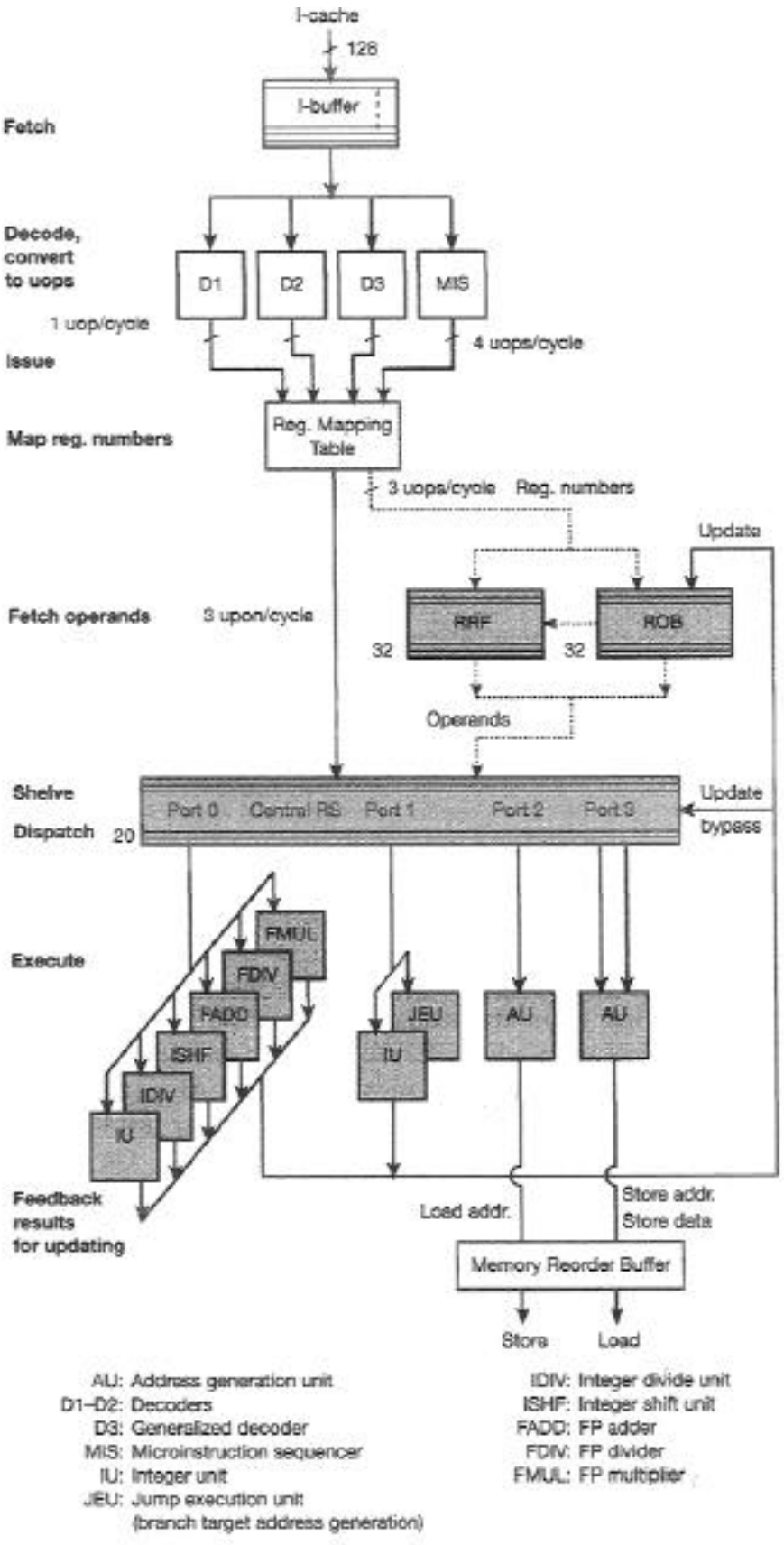


Figure 7.72 Core part of the microarchitecture of the PentiumPro.

Lecture 12

Processor design concepts of recent interest

Conditional Move Instructions

For constructs such as:

```
if (x == 0) a = b;
```

provide a single conditional move instruction:

```
CMOVZ R1, R2, R3 ;if R1 = 0, R2 = R3
```

Eliminates condition code register. Instruction found in some RISCs

Conditional Load Instructions

Could also have condition load instruction:

```
LDC R1, R2 [R3]  ;if R1=0, copy contents of memory
                 ;location into R2
```

although this version not as common.

Slide 323

Branch Predication

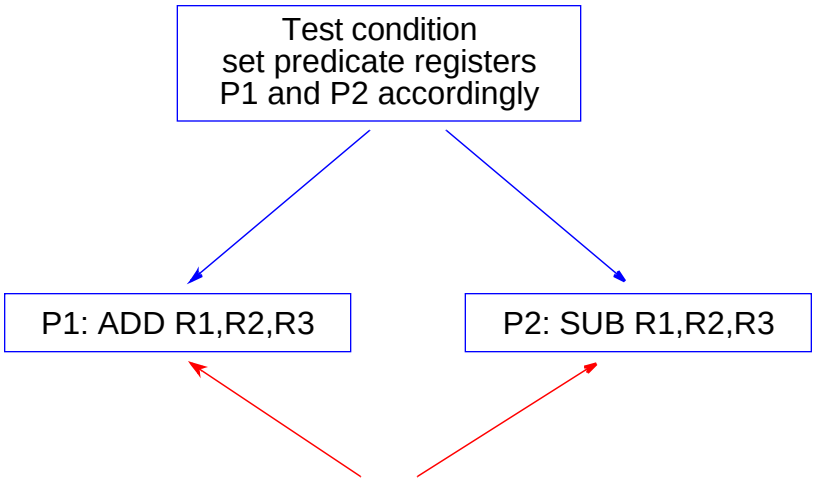
Used in Intel IA-64/Itanium processor (Intel's new processor). Branch predication proposed/developed earlier by others (University of Illinois).

Replace branch instruction with an instruction which sets a "predicate" register to TRUE and another "predicate" register to FALSE. Then have "predicated" instructions - regular machine instructions but with add field which specifies which predicate register must be TRUE for the instruction to complete. Can start execution of the predicated instruction before that but it must not retire its results unless predicate is TRUE.

Slide 324

Example

```
if (R1==R2) R1 = R2 + R3; else R1 = R2 - R3;
```



Start executing both instructions even before predicates set. Only allow one to write to R1, the one in which the predicate is TRUE

Slide 325

Predicated code

```
CMPE R1,R2,P1,P2 ;if R1=R2, set P1=TRUE, P2=FALSE
                 ;else set P1 = FALSE, P2 = TRUE
P1: ADD R1,R2,R3
P2: SUB R1,R2,R3
```

P1/P2 are single bits which turn instruction on/off
- part of instruction not labels

Could have the predicate generators (CMPZ above) predicated itself. (Actual notation for predicate generator may be different.)

Slide 326

Advantages of predicated code

- Allows instructions to be executed simultaneously and "speculatively"
- Reduces branch misprediction penalties and hence can produced significantly faster code - Intel/HP quote 50% fewer branches and 37% faster code
- Most useful when branch prediction is hard to do accurately, e.g. in sorting, data compression, non-deterministic applications.

Instructions can be fetched/grouped together.

Slide 327

Disadvantages

- Requires a completely new instruction set - cannot be fully grafted onto existing machines - hence Intel's completely new design.
- Speculatively executing instructions is wasteful of resources within the processor, if there is a high probability that the instructions will have to be abandoned.

Slide 328

Speculative Load

Loading data from memory before needed to reduce effects of memory latency. Done by moving load instruction to earlier in program than where it would normally be needed - "hoisted" to an earlier point.

Compilers can do this to some extent anyway.

Problem occurs when hoisting is across a branch instruction and a memory exception occurs, e.g. an invalid address, segmentation fault. This would generate an exception even if load was not needed finally, i.e. the branch was down the path not needing the load.

Slide 329

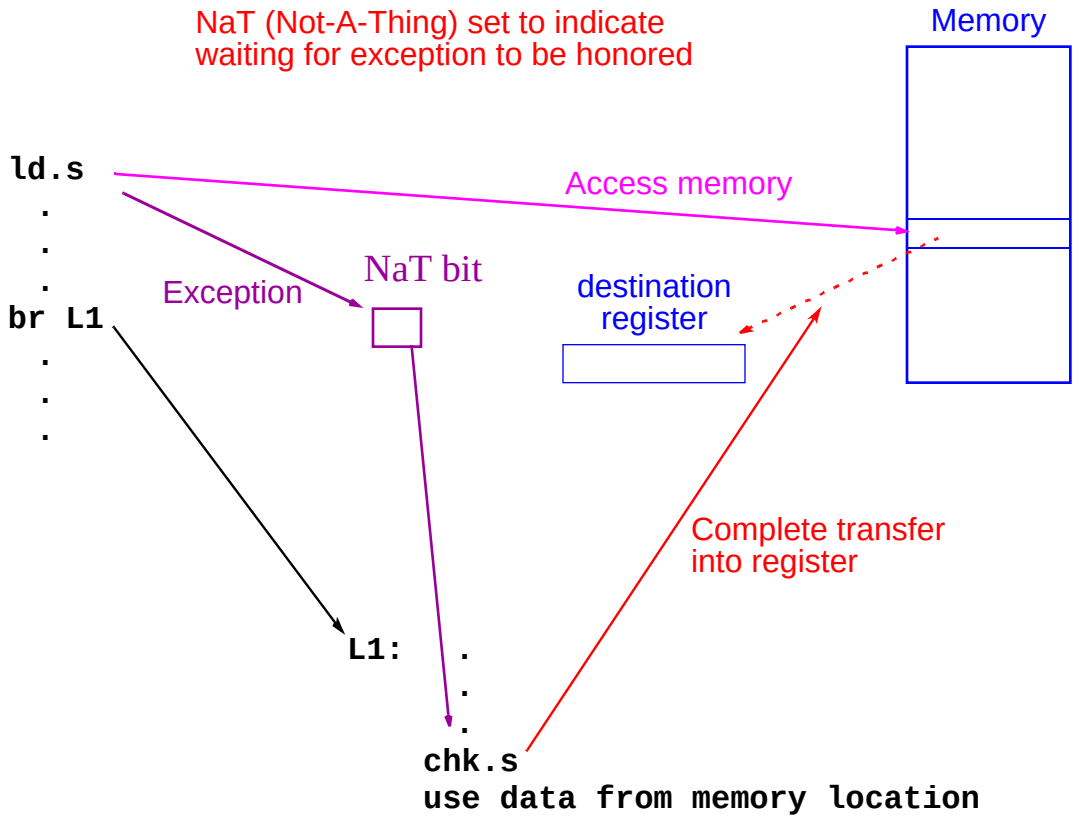
IA-64 solution

Have two instructions:

- A speculative load instruction, ld.s, which performs the load operation without loading the destination register. If an exception occurs, a flag is set.
- A check instruction, chk.s, which checks whether an exception occurred. If it has, an exception handler is called, otherwise the destination is loaded.

The speculative load is placed as early as possible in the code. The check is placed where the result is needed. Check can be predicated.

Slide 330



Slide 331

Advantages of Speculative Load Instructions

- Hides the memory latency, a significant factor in obtaining improved performance. Intel quotes a 79% improvement when combined with predication (August et al, 1998)
- Particularly effective with many memory (cache) accesses such as in large databases, operating systems.
- Scheduling flexibility to obtain parallelism

Slide 332

Intel/Hewlett-Packard IA-64 Architecture

Based upon VLIW (very long instruction word) concept proposed in the 1980's.

Independent instructions packaged into groups and sent to processor. Processor executed the group of instructions simultaneously (if sufficient internal resources available).

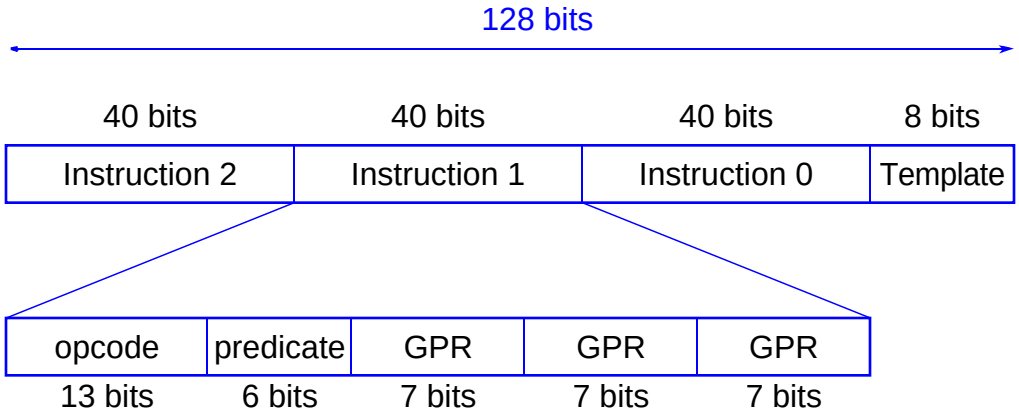
Instruction level parallelism where the compiler made the decision on which instructions were to be executed together.

Simple processor - does not detect parallelism itself during execution.

Intel/HP call their version as "EPIC - Explicit Parallel Instruction Computing."

Slide 333

IA-64 instruction format



GPR = specifies one of 128 general-purpose registers

Slide 334

Sources of further information

W.-M. Hwu, “Introduction to Predicated Execution,” *IEEE Computer*, January 1998, pp. 49-50.

M. S. Schlansker and R.R. Rau, “EPIC: Explicitly Parallel Instruction Computing,” *IEEE Computer*, February 2000, pp. 37-45.
C. Dulong, “The IA-64 Architecture at Work,” *IEEE Computer*, July 1998, pp. 24-32.

C. Zheng and C. Thompson, “PA-RISC to IA-64: Transparent Execution, No Recompilation,” *IEEE Computer*, March 2000, pp. 47-52. (see also other articles in this issue.)

“Inside Intel’s Mersed A Strategic Planning Discussion An Executive White Paper,” Aberdeen Group, Inc. Boston MA, July 1999. (See www.aberdeen.com)