

# High Performance Architectures

Superscalar and VLIW

Part 1

# Basic computational models

- Turing
- von Neumann
- object based
- dataflow
- applicative
- predicate logic based

Computational model	Turing	von Neumann	Object-based	Dataflow	Applicative	Predicate logic-based
<b>Basic items of computation</b>	Elements of the tape symbol set Mappings defined on them	Data assigned to named entities (variables) Operations performed on data	Objects, which can be manipulated by a set of messages Messages sent to manipulate objects	Data assigned to named entities (variables) Operations performed on data	Arguments Applications of functions to their arguments	Elements of sets Predicates declared to them
<b>Style</b>	Procedural			Declarative		
<b>Problem description model</b>	<p>Input data</p>	<p>Input data</p>	<p>Initial states of objects</p>	<p>Input data</p>	<p>Arguments of the functions to be evaluated</p>	<p>Goal statement</p>
<b>Interpretation of how to perform the computation</b>	Starting with the initial state, a sequence of subsequent states will be performed until a final state is reached	The given sequence of instructions will be executed using the input data	The given sequence of messages will be executed	Input data flows through the dataflow graph and, as a consequence, output data is produced	The function with the given arguments is subsequently rewritten by substituting the function definitions referenced and performing the possible reductions until no more substitutions or reductions are feasible	The goal statement is subsequently rewritten using resolution until a logical inference is found or no more unifications are possible
<b>Assumed execution model</b>	<b>Execution semantics</b> State transition semantics	<b>Execution semantics</b> State transition semantics	<b>Execution semantics</b> State transition semantics	<b>Execution semantics</b> Dataflow semantics	<b>Execution semantics</b> Reduction semantics	<b>Execution semantics</b> SLD-resolution
<b>Control of the execution sequence</b>	Control driven	Control driven	Control driven	Data driven (eager evaluation)	Demand driven (lazy evaluation)	Defined by both the goal processing rule and search rule
<b>Concurrency</b>	Sequential			Parallel		

# The von Neumann computational model

- Basic items of computation are **data**
  - variables (named data entities)
  - memory or register locations whose addresses correspond to the names of the variables
  - data container
  - multiple assignments of data to variables are allowed
- Problem description model is **procedural** (sequence of instructions)
- Execution model is state transition semantics
  - Finite State Machine

# von Neumann model vs. finite state machine

– As far as execution is concerned the von Neumann model behaves like a finite state machine (FSM)

- $FSM = \{ I, G, \delta, G_0, G_f \}$
- $I$ : the input alphabet, given as the set of the instructions
- $G$ : the set of the state (global), data state space  $D$ , control state space  $C$ , flags state space  $F$ ,  $G = D \times C \times F$
- $\delta$ : the transition function:  $\delta: I \times G \rightarrow G$
- $G_0$ : the initial state
- $G_f$ : the final state

# Key characteristics of

## the von Neumann model

- Consequences of multiple assignments of data
  - history sensitive
  - side effects
- Consequences of control-driven execution
  - computation is basically a sequential one
- ++ easily be implemented
- Related language
  - allow declaration of variables with multiple assignments
  - provide a proper set of control statements to implement the control-driven mode of execution

# Extensions of the von Neumann computational model

- new abstraction of parallel execution
- communication mechanism allows the transfer of data between executable units
  - unprotected shared (global) variables
  - shared variables protected by modules or monitors
  - message passing, and
  - rendezvous
- synchronization mechanism
  - semaphores
  - signals
  - events
  - queues
  - barrier synchronization

# Key concepts relating to computational models

- Granularity
  - complexity of the items of computation
  - size
  - fine-grained
  - middle-grained
  - coarse-grained
- Typing
  - data based type ~ Tagged
  - object based type (object classes)



# Instruction-Level Parallel Processors

{**Objective**: executing two or more instructions in parallel}

- 4.1 Evolution and overview of ILP-processors
- 4.2 Dependencies between instructions
- 4.3 Instruction scheduling
- 4.4 Preserving sequential consistency
- 4.5 The speed-up potential of ILP-processing

# Improve CPU performance by

- increasing clock rates
  - (CPU running at 2GHz!)
- increasing the number of instructions to be executed in parallel
  - (executing 6 instructions at the same time)

# How do we increase the number of instructions to be executed?

Traditional  
von Neumann processors

Scalar  
ILP-processors

Superscalar  
ILP-processors

(Sequential issue,  
sequential execution)

(Sequential issue,  
parallel execution)

(Parallel issue,  
parallel execution)



Parallelism of instruction execution



Parallelism of instruction issue

*Typical  
implementation*

*Non-pipelined  
processors*

*Processors with multiple  
non-pipelined EUs and  
pipelined processors*

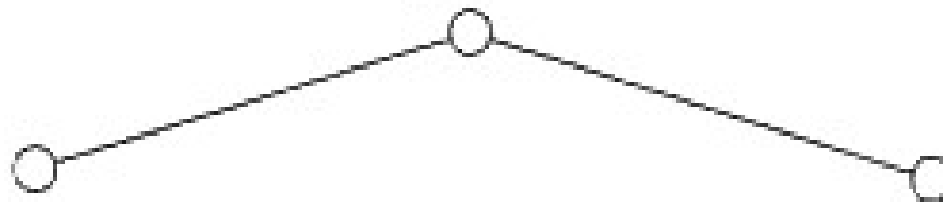
*VLIW and superscalar  
processors embodying  
multiple pipelined EUs*



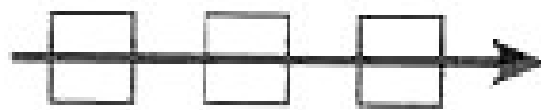
Processor performance

# Time and Space parallelism

Internal of operation  
principle  
of ILP-processors



**Pipelined operation**

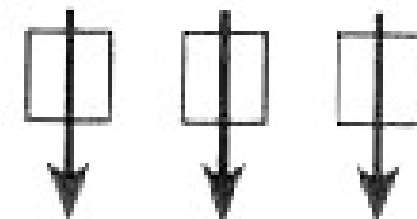


$EU_1$   $EU_2$   $EU_m$



**Pipelined processors**

**Parallel operation**

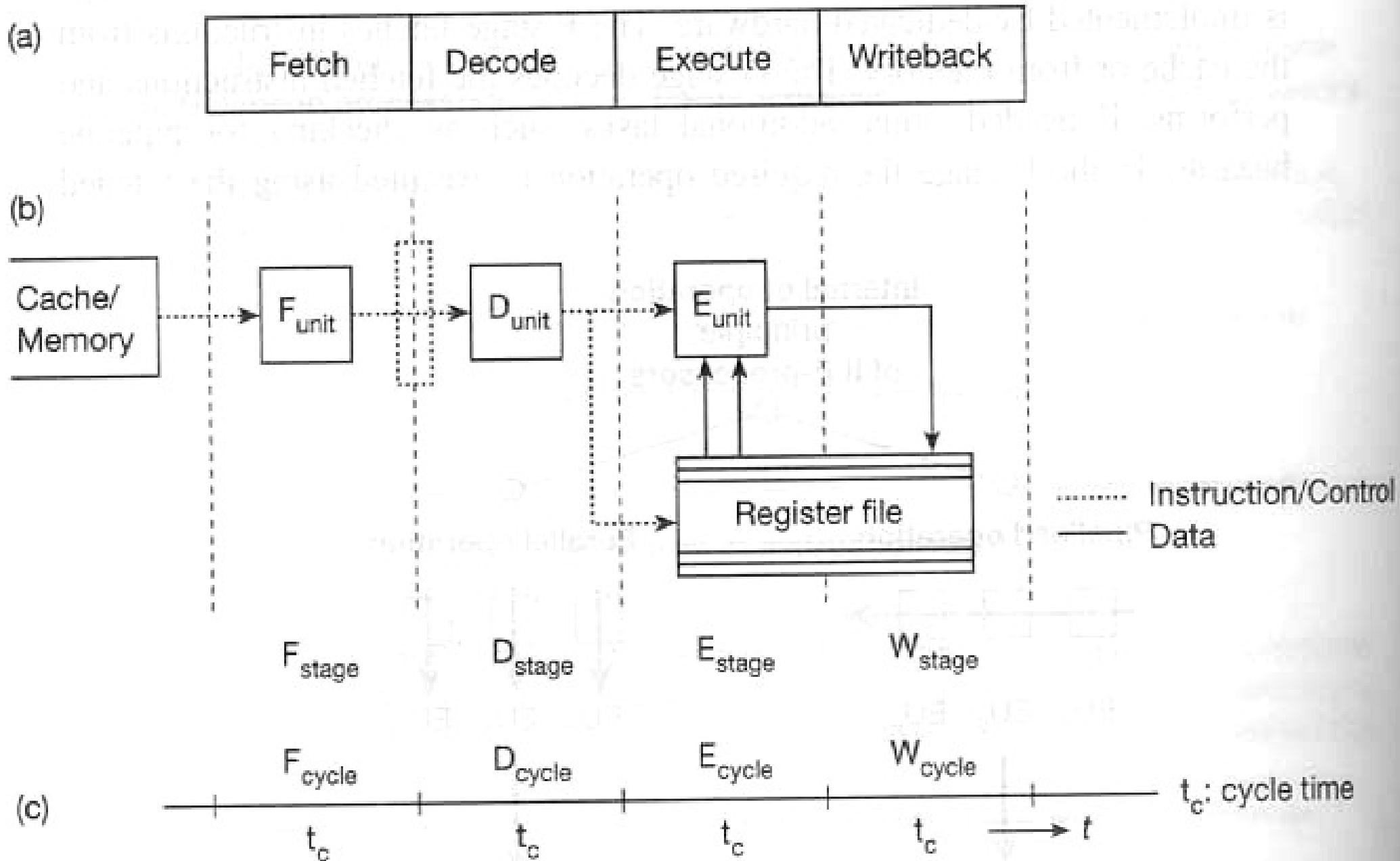


$EU_1$   $EU_2$   $EU_m$



**VLIW and superscalar processors**

# Pipeline (assembly line)

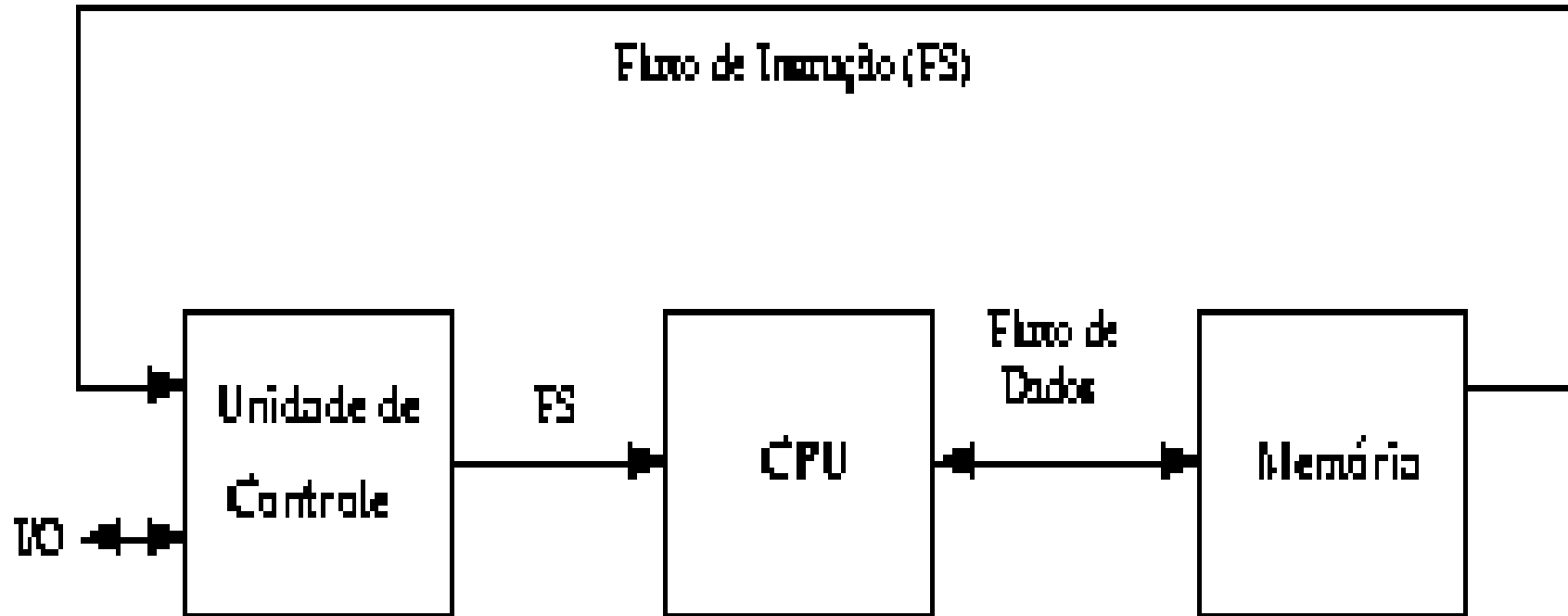




# Flynn's Taxonomy

- Organizes the processor architectures according to the data and instruction stream
  - SISD - *Single Instruction, Single Data stream*
  - SIMD - *Single Instruction, Multiple Data stream*
  - MISD - *Multiple instruction, Single Data stream*
  - MIMD - *Multiple Instruction, Multiple Data stream*

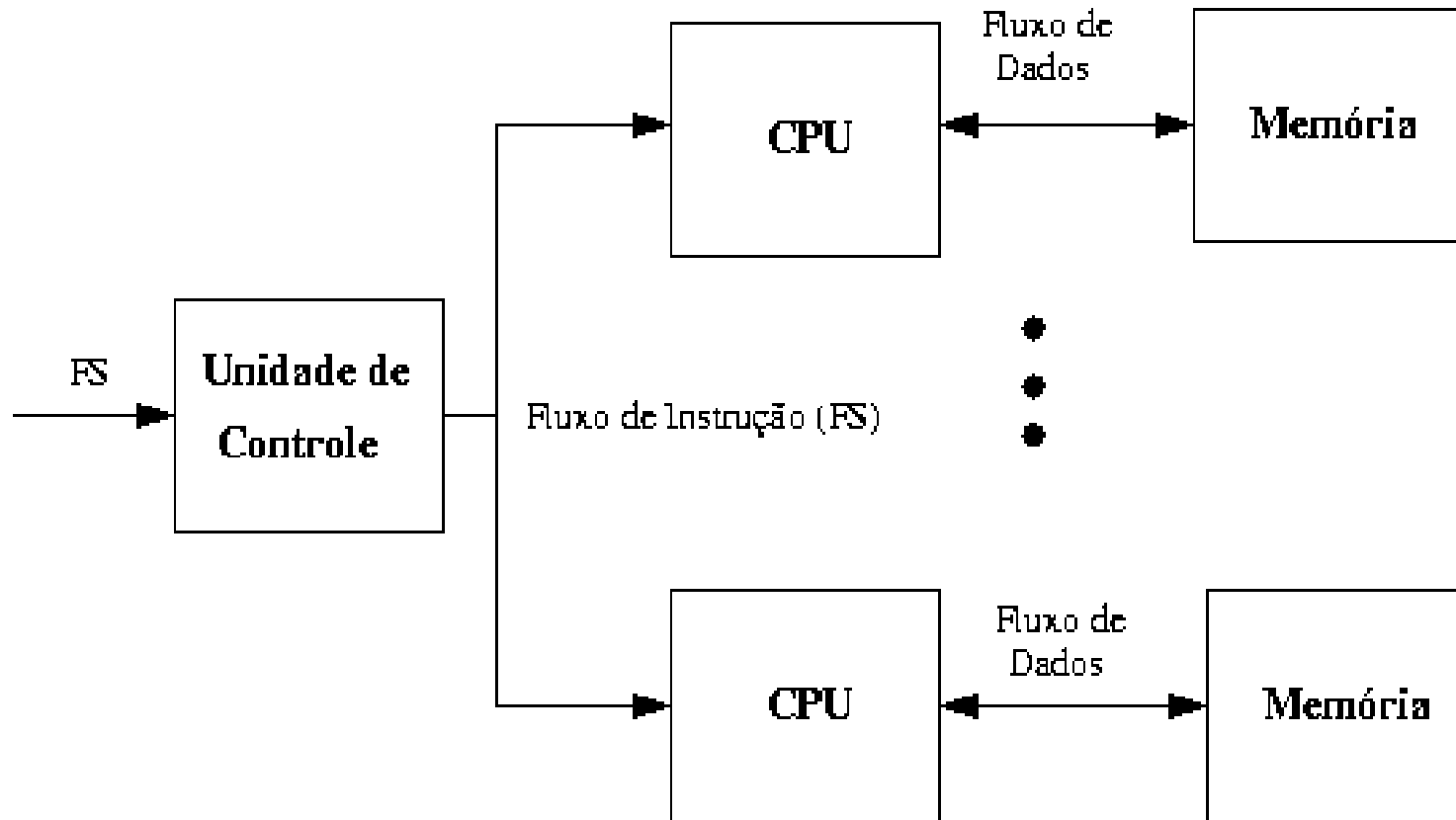
# SISD



**Exemplos: Estações de trabalho e Computadores pessoais com um único processador**



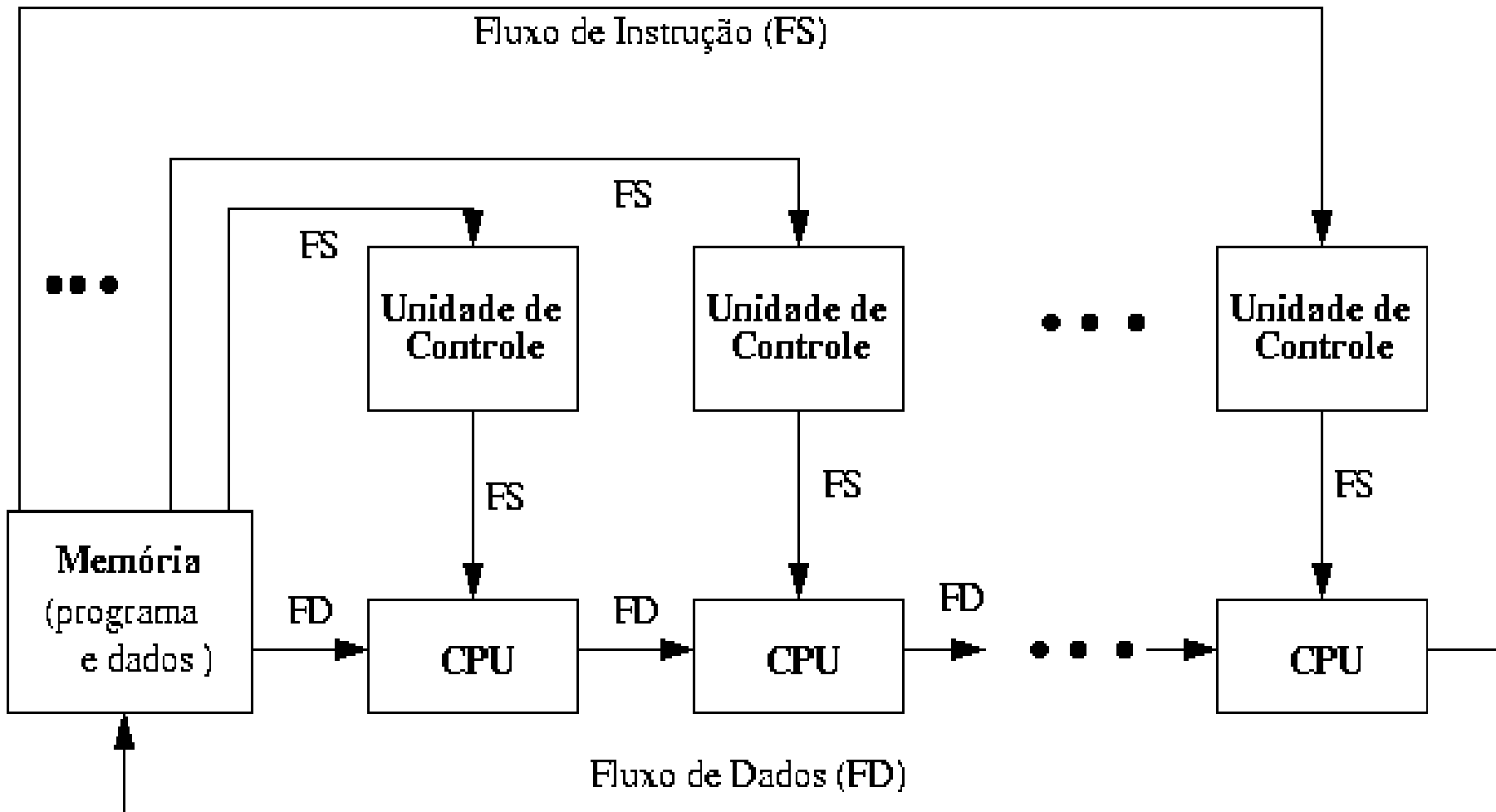
# SIMD



**Exemplos: ILIAC IV, MPP, DHP, MASP MP-2 e CPU Vetoriais**

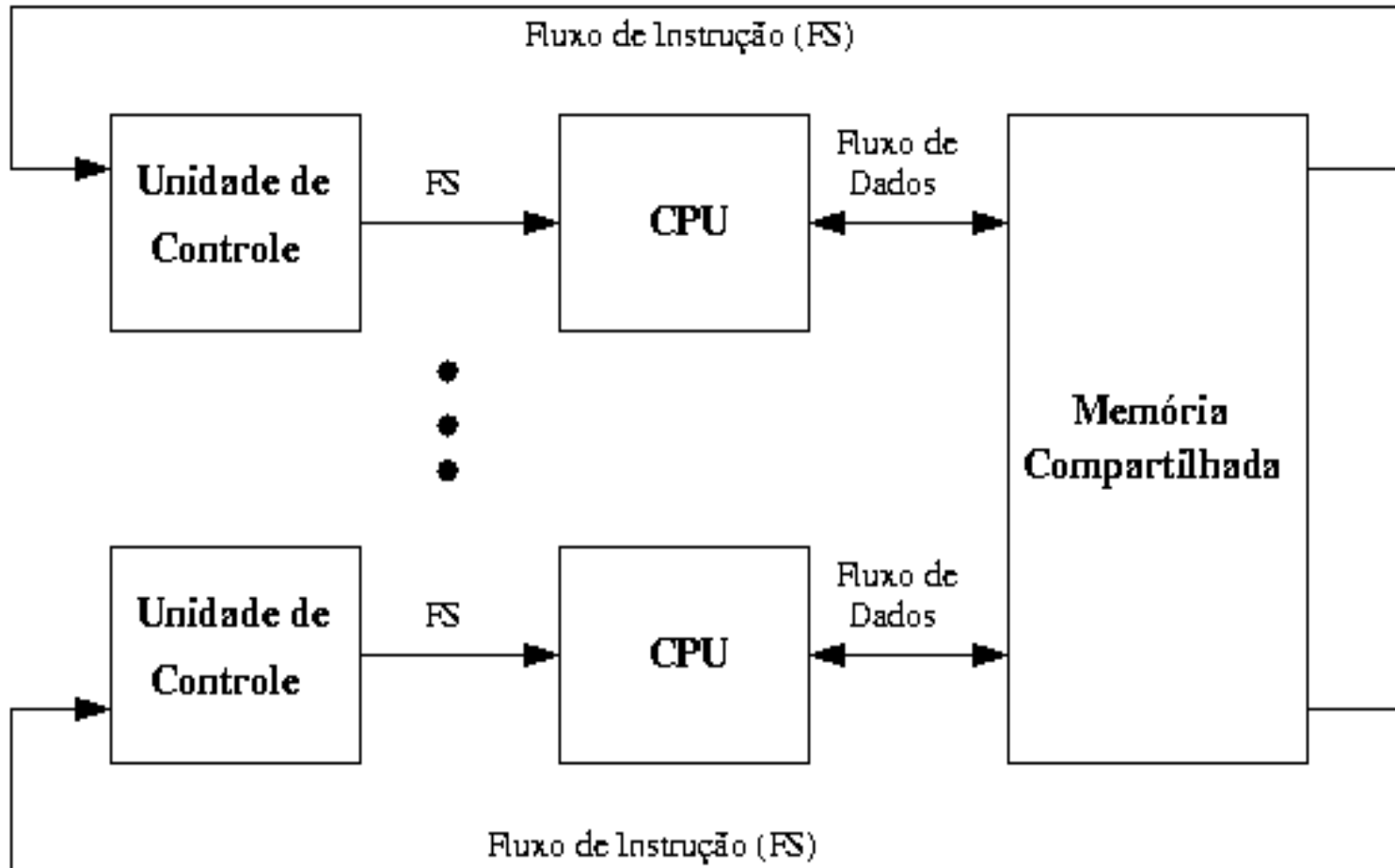
**Extensões multimídia são consideradas como uma forma de paralelismo SIMD**

# MISD



**Exemplos: Array Sistólicos**

# MIMD



**Exemplos: Cosmic Cube,  
nCube 2, iPSC, FX-2000,  
SGI Origin, Sun Enterprise  
5000 e Redes de  
Computadores**

**Mais difundida  
Memória Compartilhada  
Memória Distribuída**

**Flexível**

***Usa microprocessadores off-the-shelf***

# Resume

OBS.: The Flynn's taxonomy is a reference model.  
Actually, there exist some processors using features of more than one category

- **MIMD**

- Most of the parallel machines
- It seems adequate to the general purpose parallel computing
- 

- **SIMD**

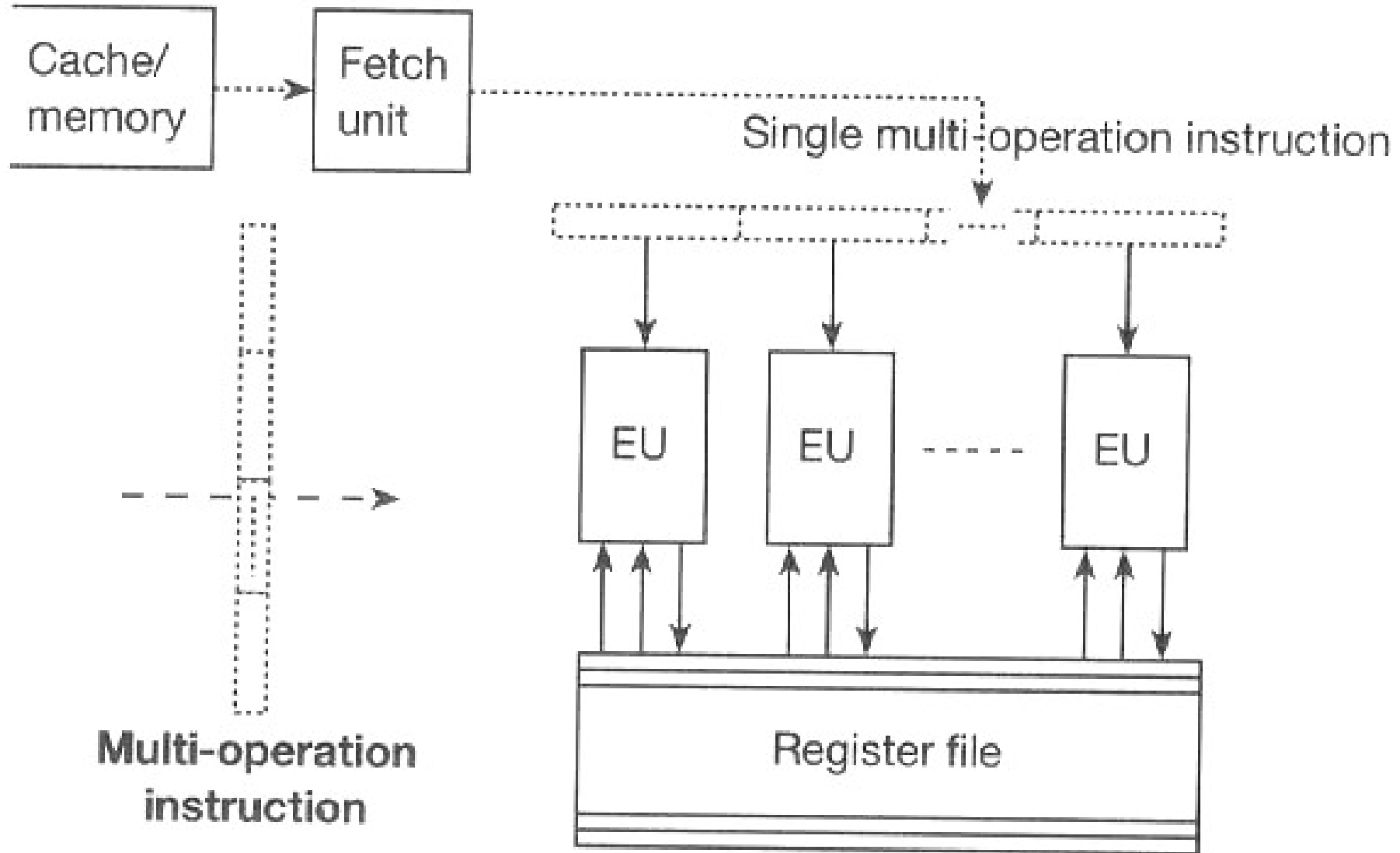
- **MISD**

- **SISD**

- Sequential computing

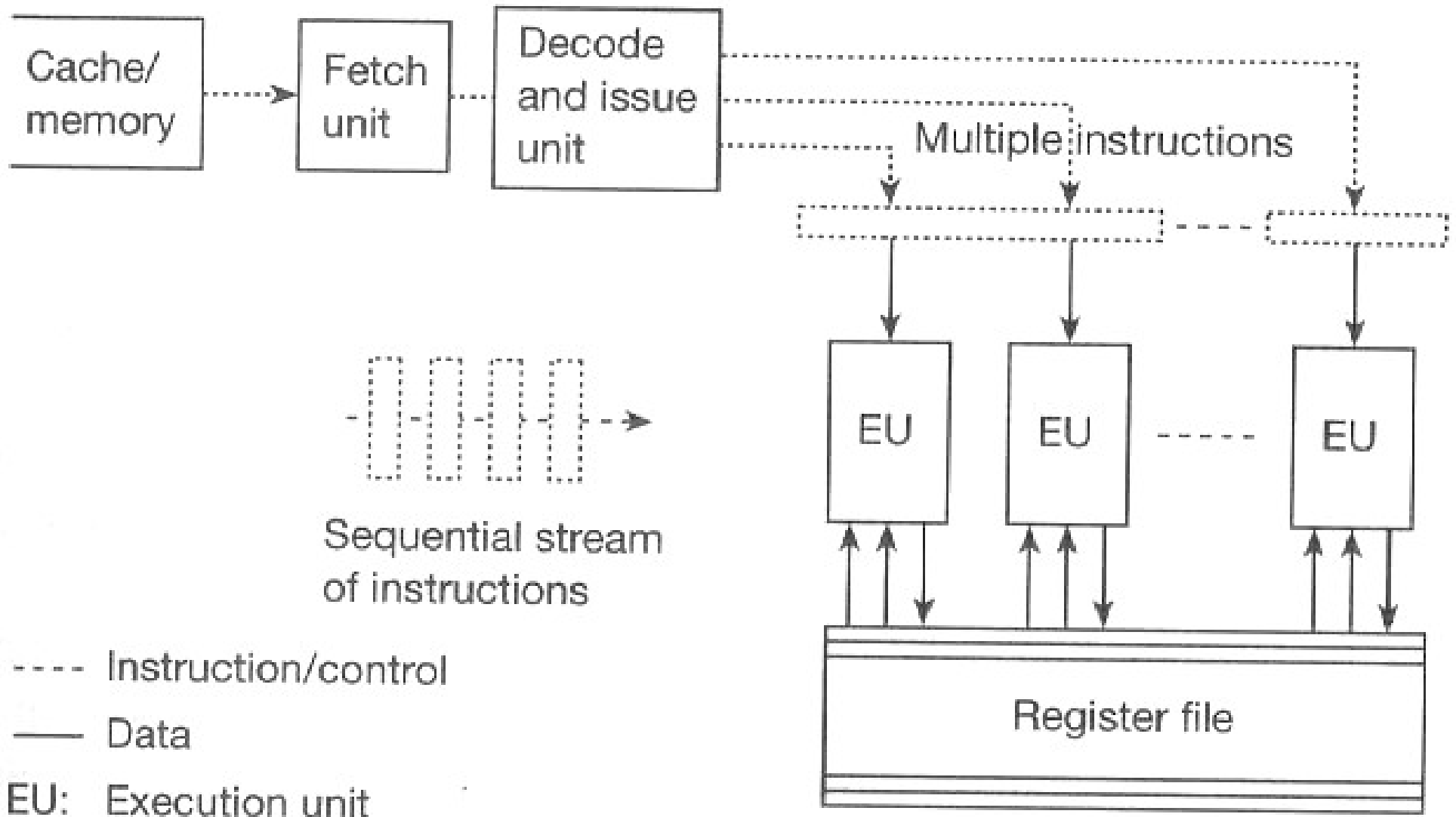
# VLIW

(very long instruction word, 1024 bits!)



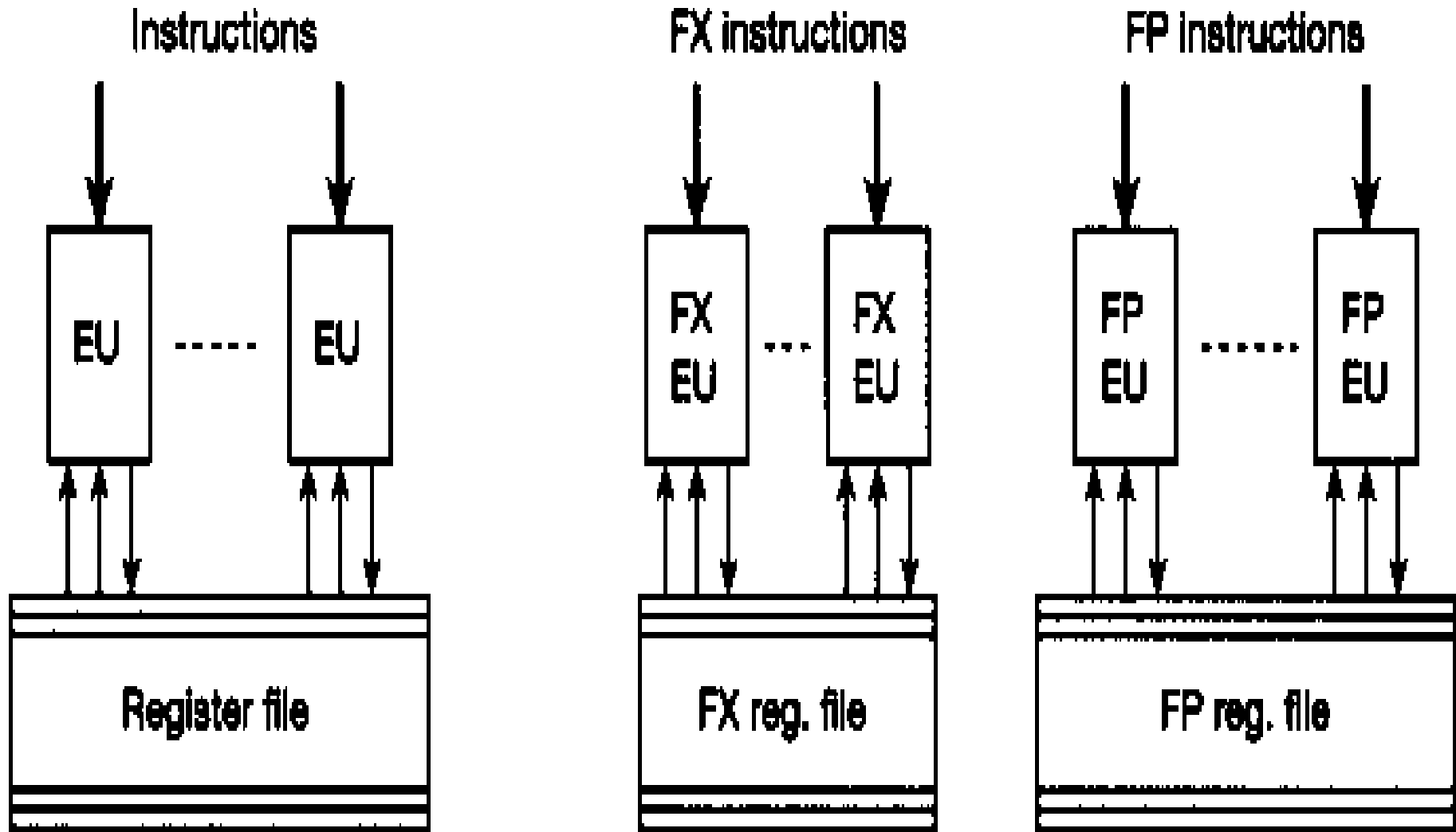
VLIW approach

# Superscalar (sequential stream of instructions)



**Superscalar approach**

# Basic Structure of VLIW Architecture



(a)

(b)

# Basic principle of VLIW

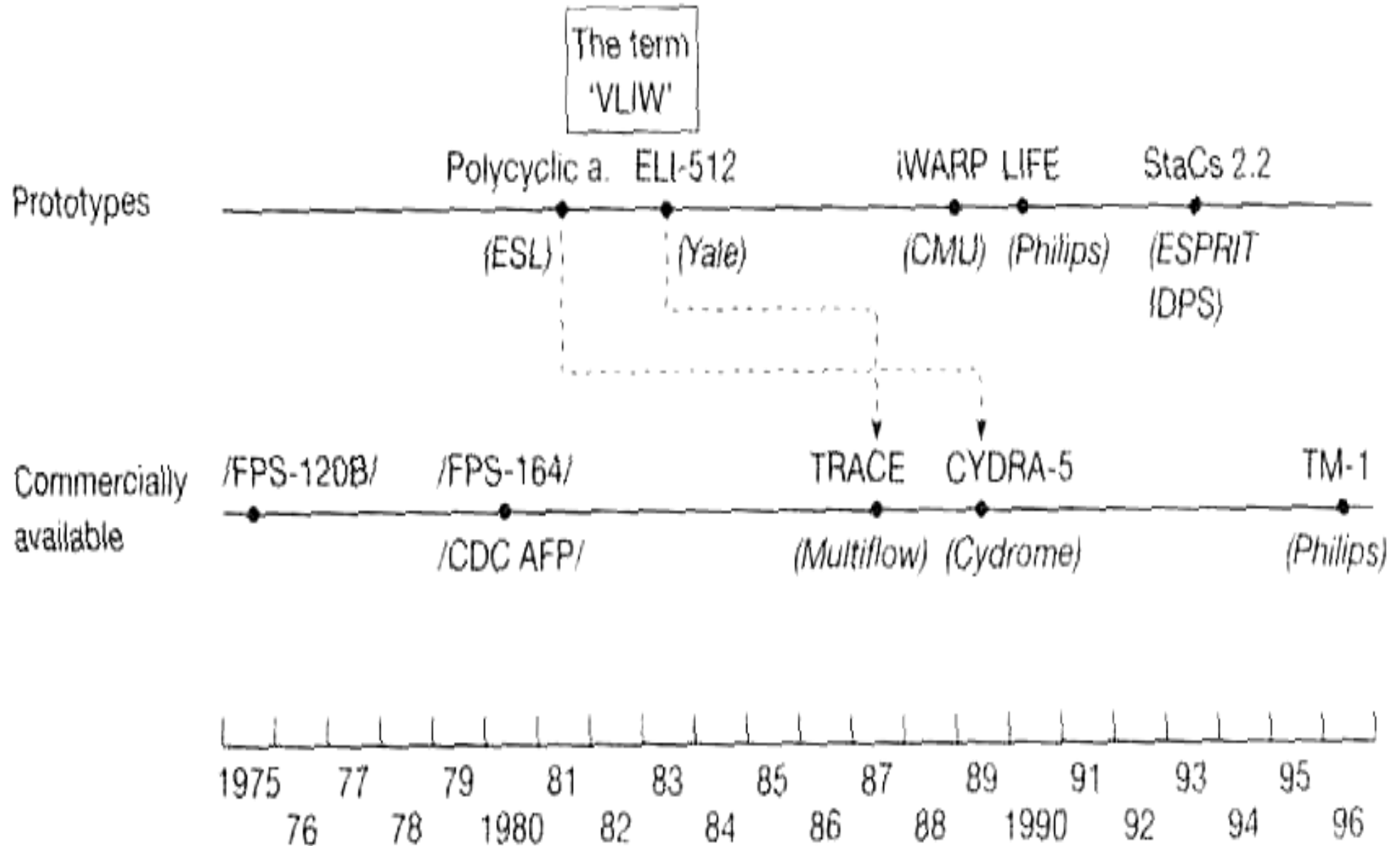
- Controlled by very long instruction words
  - comprising a control field for each of the execution units
- Length of instruction depends on
  - the number of execution units (5-30 EU)
  - the code lengths required for controlling each EU (16-32 bits)
  - 256 to 1024 bits
- Disadvantages: on average only some of the control fields will actually be used
  - waste memory space and memory bandwidth
  - e.g. Fortran code is 3 times larger for VLIW (Trace processor)



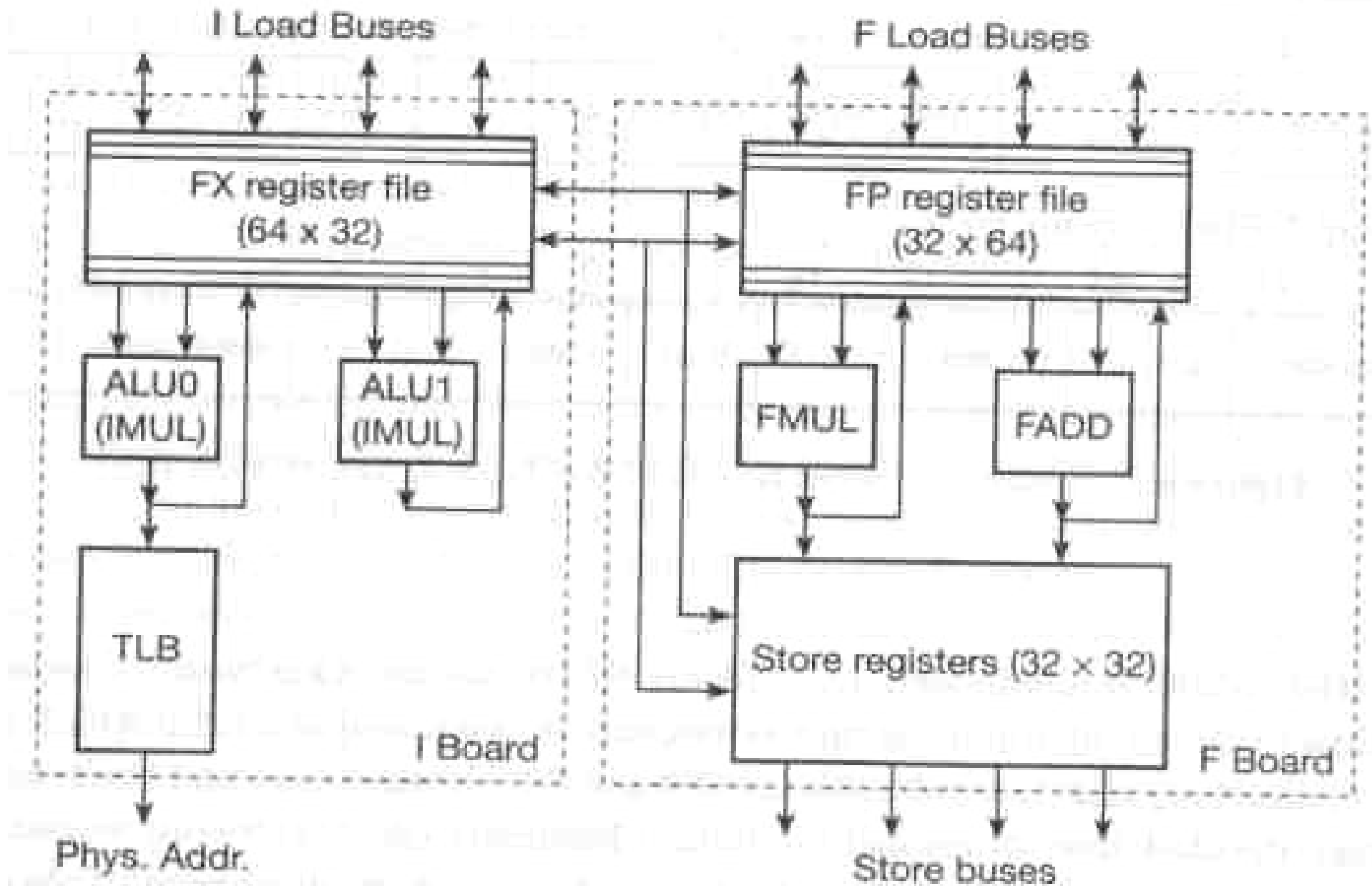
# VLIW: Static Scheduling of instructions/

- Instruction scheduling done entirely by [software] compiler
- Lesser Hardware complexity translate to
  - increase the clock rate
  - raise the degree of parallelism (more EU);  
{Can this be utilized?}
- Higher Software (compiler) complexity
  - compiler needs to aware hardware detail
    - number of EU, their latencies, repetition rates, memory load-use delay, and so on
    - cache misses: compiler has to take into account worst-case delay value
  - this hardware dependency restricts the use of the same compiler for a family of VLIW processors

## 6.2 overview of proposed and commercial VLIW architectures



# 6.3 Case study: Trace 200



# Trace 7/200

- 256-bit VLIW words
- Capable of executing 7 instructions/cycle
  - 4 integer operations
  - 2 FP
  - 1 Conditional branch
- Found that every 5<sup>th</sup> to 8<sup>th</sup> operation on average is a conditional branch
- Use sophisticated branching scheme: multi-way branching capability
  - executing multi-paths
  - assign priority code corresponds to its relative order

# Trace 28/200: storing long instructions

- 1024 bit per instruction
- a number of 32-bit fields maybe empty
- Storing scheme to save space
  - 32-bit mask indicating each sub-field is empty or not
  - followed by all sub-fields that are not empty
  - resulting still 3 time larger memory space required to store Fortran code (vs. VAX object code)
  - very complex hardware for cache fill and refill
- Performance data is Impressive indeed!

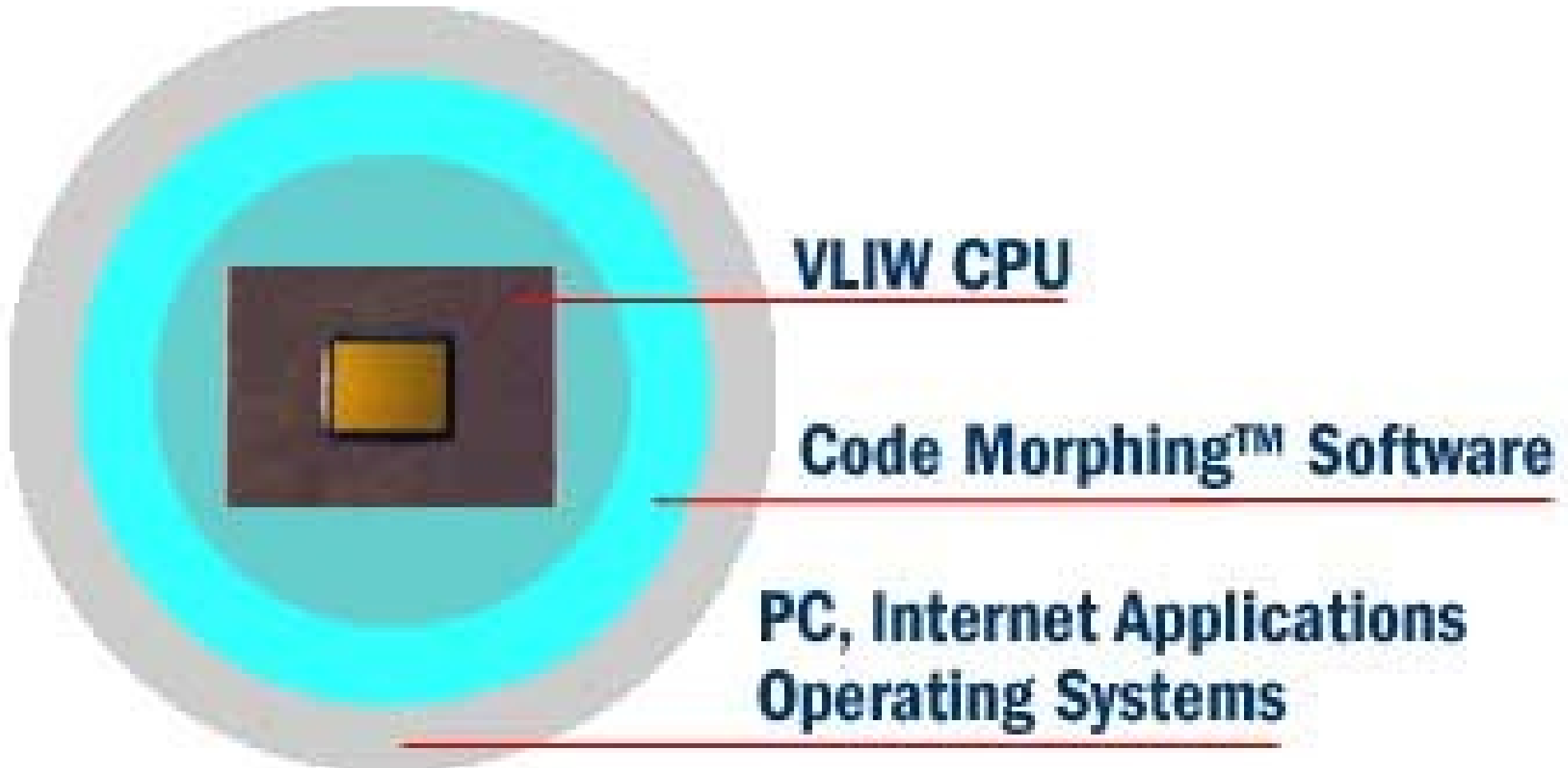
# Trace: Performance data

**Table 6.1** Performance data for Trace processors compared with that of the DEC 8700 and Cray XMP.

	<i>Instruction issue rate (ns)</i>	<i>Compiled Linpack (Full precision) (MFLOPS)</i>
Trace 7/200	130	6
Trace 14/200	130	10
DEC 8700	45	0.97
Cray XMP	8	24

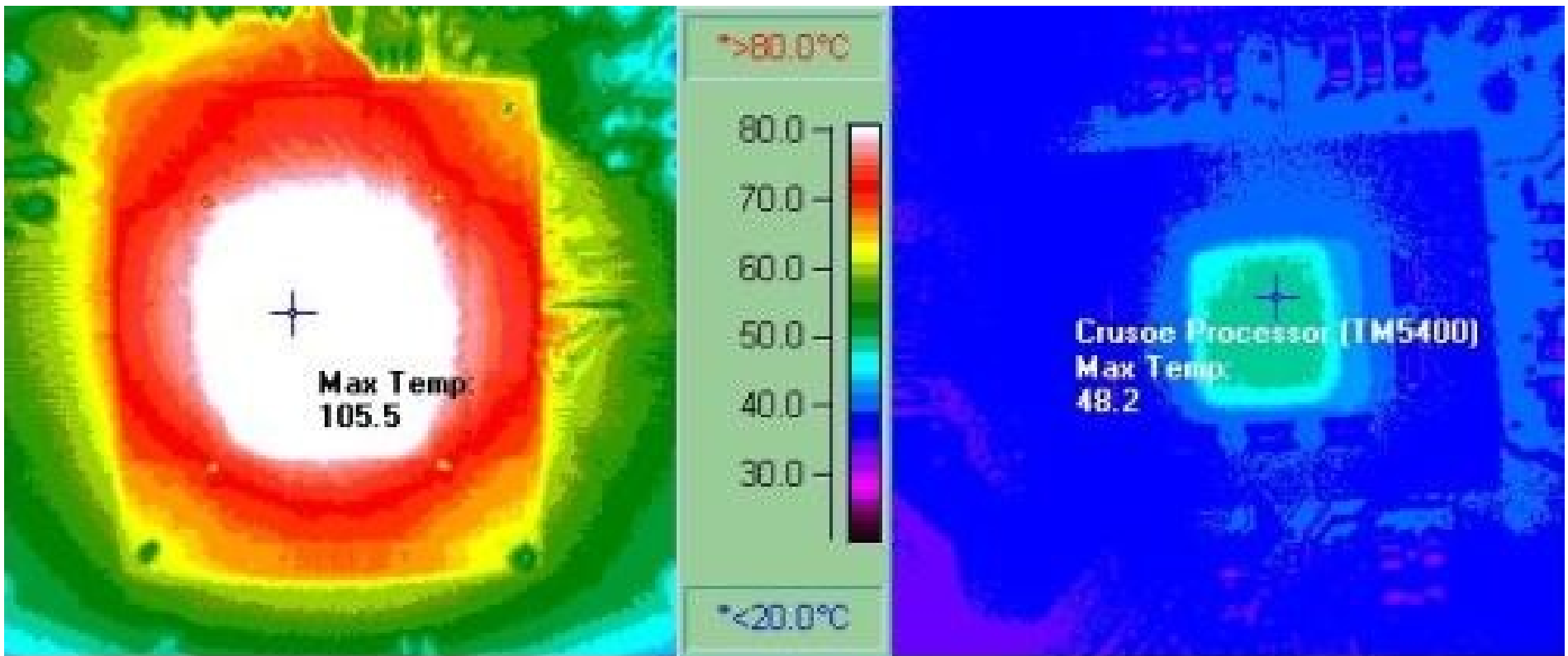
# Transmeta: Crusoe

- Full x86-compatible: by dynamic code translation
- High performance: 700MHz in mobile platforms



# Crusoe

- “Remarkably low power consumption”
  - “A full day of web browsing on a single battery charge”
- Playing DVD: Conventional? 105 C vs. Crusoe: 48 C

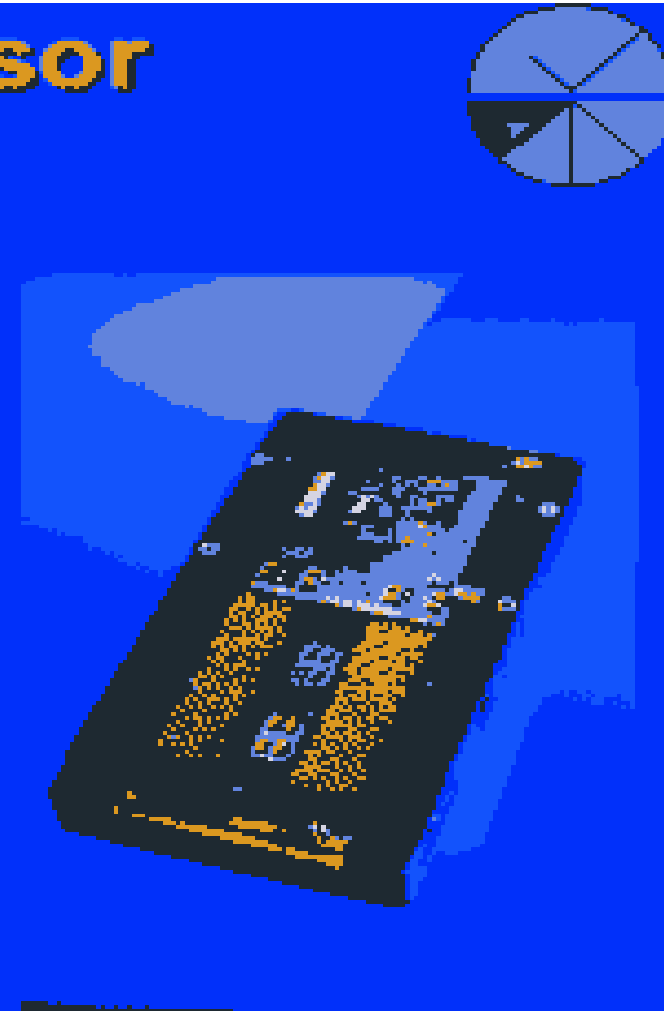




# Intel: Itanium (VLIW) {EPIC} Processor

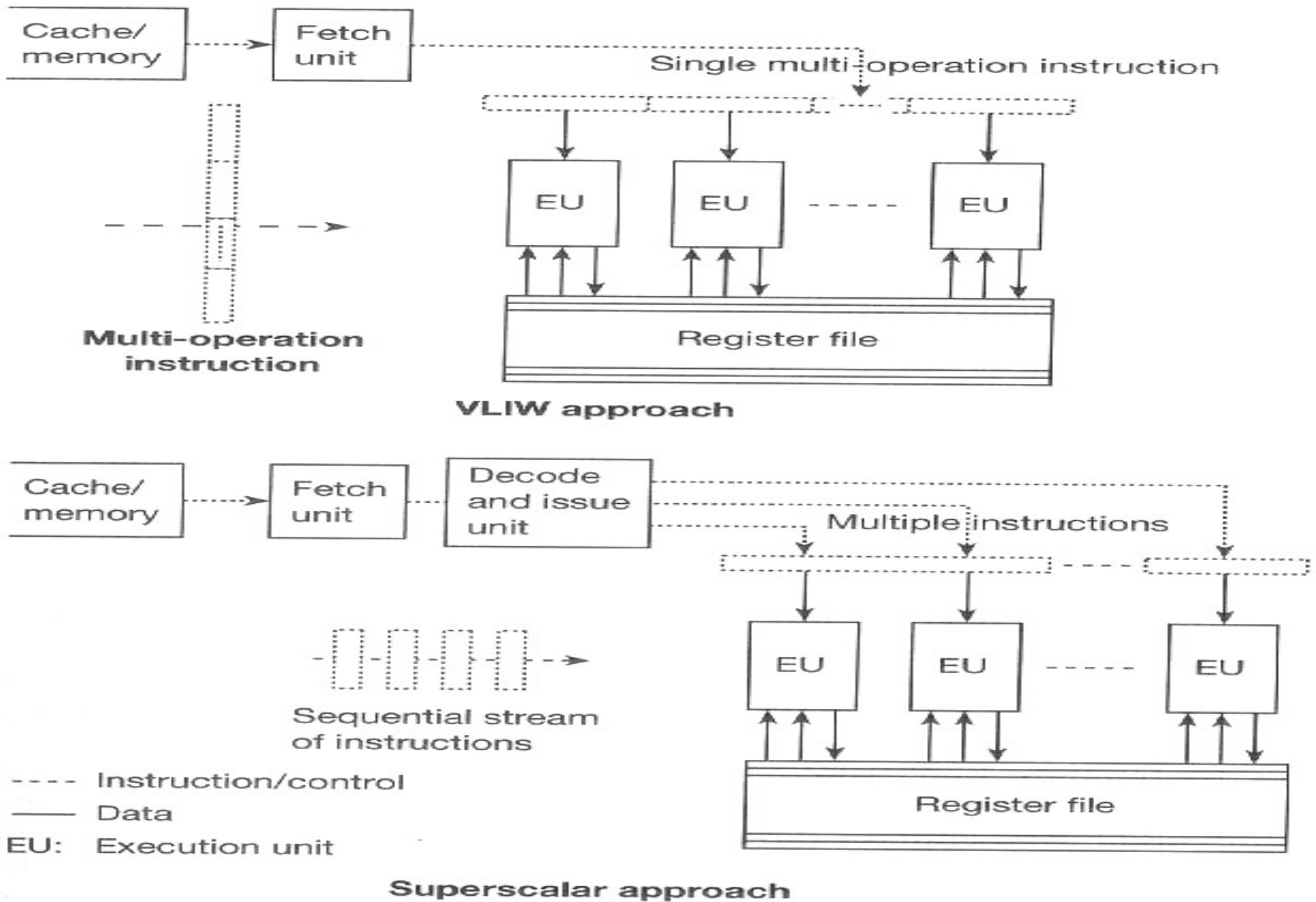
## Itanium™ Processor

- 800 MHz production frequency
  - EPIC enables up to 20 operations per clock
- 4 MB high speed on-cartridge L3 cache
- Over 320M transistors
  - 25M in CPU, 295M in L3 cache
- 2.1 GB/s multi-drop system bus
  - Enhanced Defer Mechanism enables high scalability through improved bus efficiency
- Extensive reliability and availability
  - ECC, parity protection, enhanced MCA
- Excellent functionality on initial silicon
  - No architectural or ISA changes required
  - MP functionality on track to plan



*Tuning / testing for production this year*

# Superscalar Processors vs. VLIW

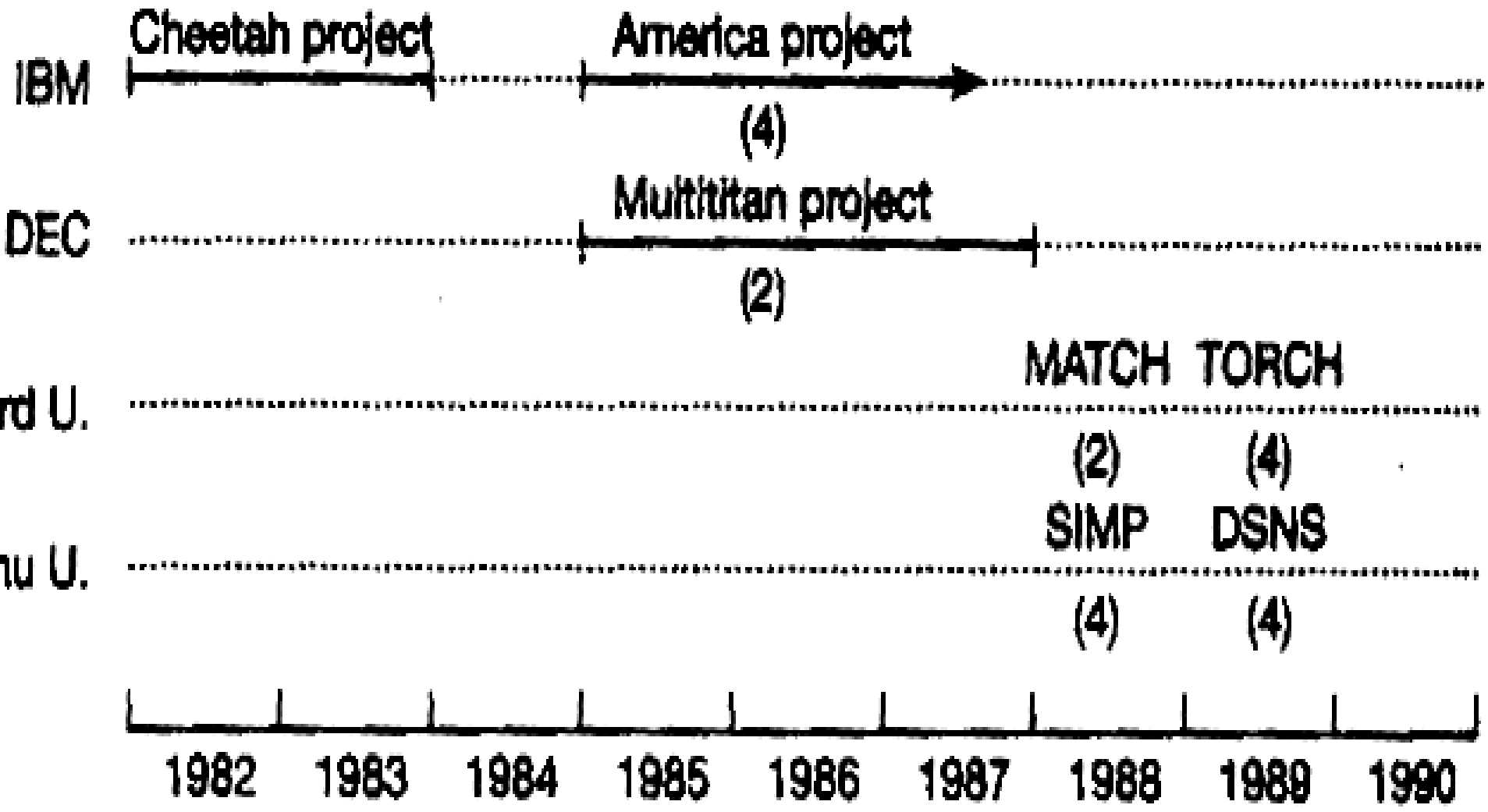


# Superscalar Processor: Intro

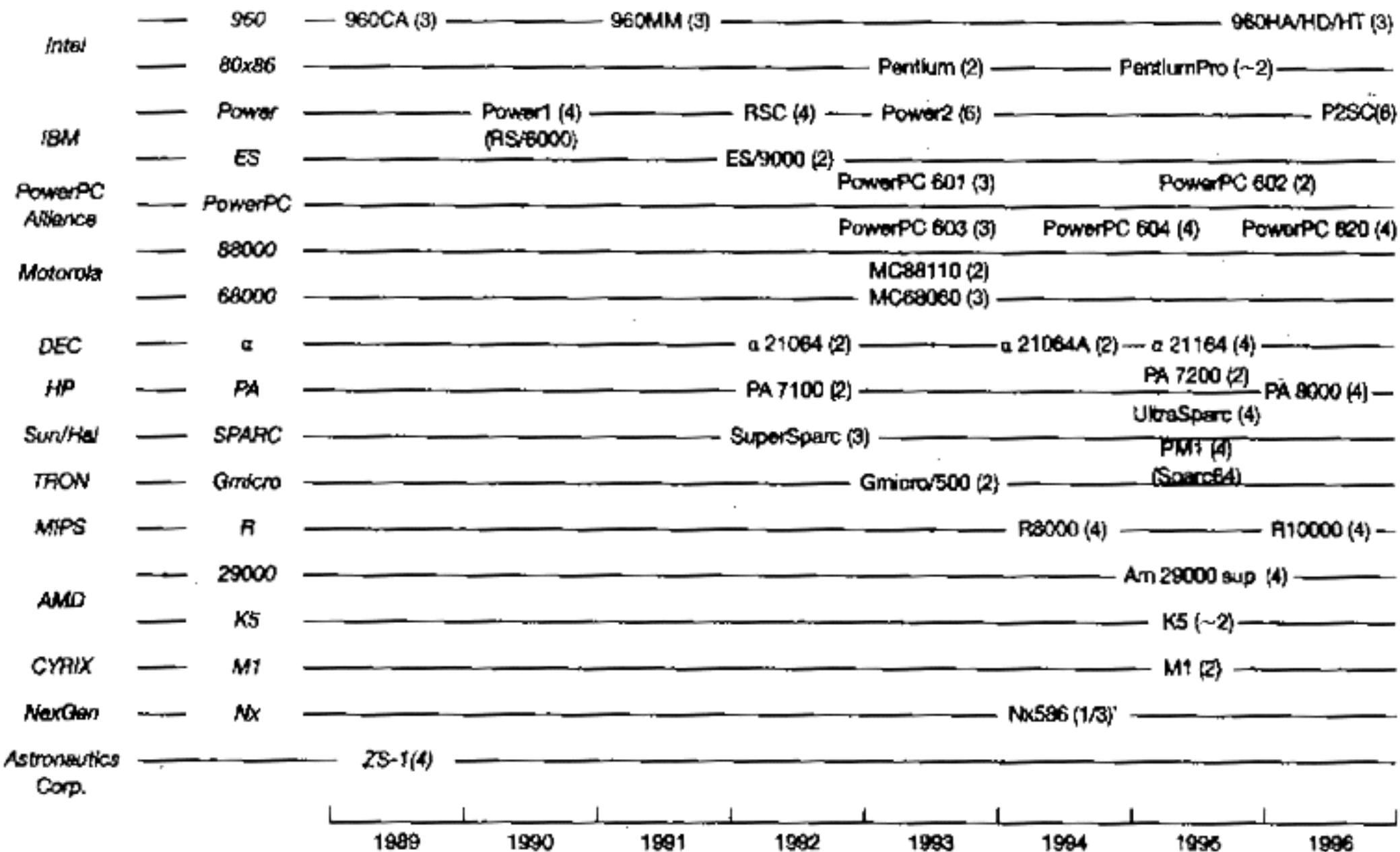
- Parallel Issue
- Parallel Execution
- {Hardware} Dynamic Instruction Scheduling
- Currently the predominant class of processors
  - Pentium
  - PowerPC
  - UltraSparc
  - AMD K5-
  - HP PA7100-
  - DEC  $\alpha$

# Emergence and spread of superscalar processors

The term  
'Superscalar'

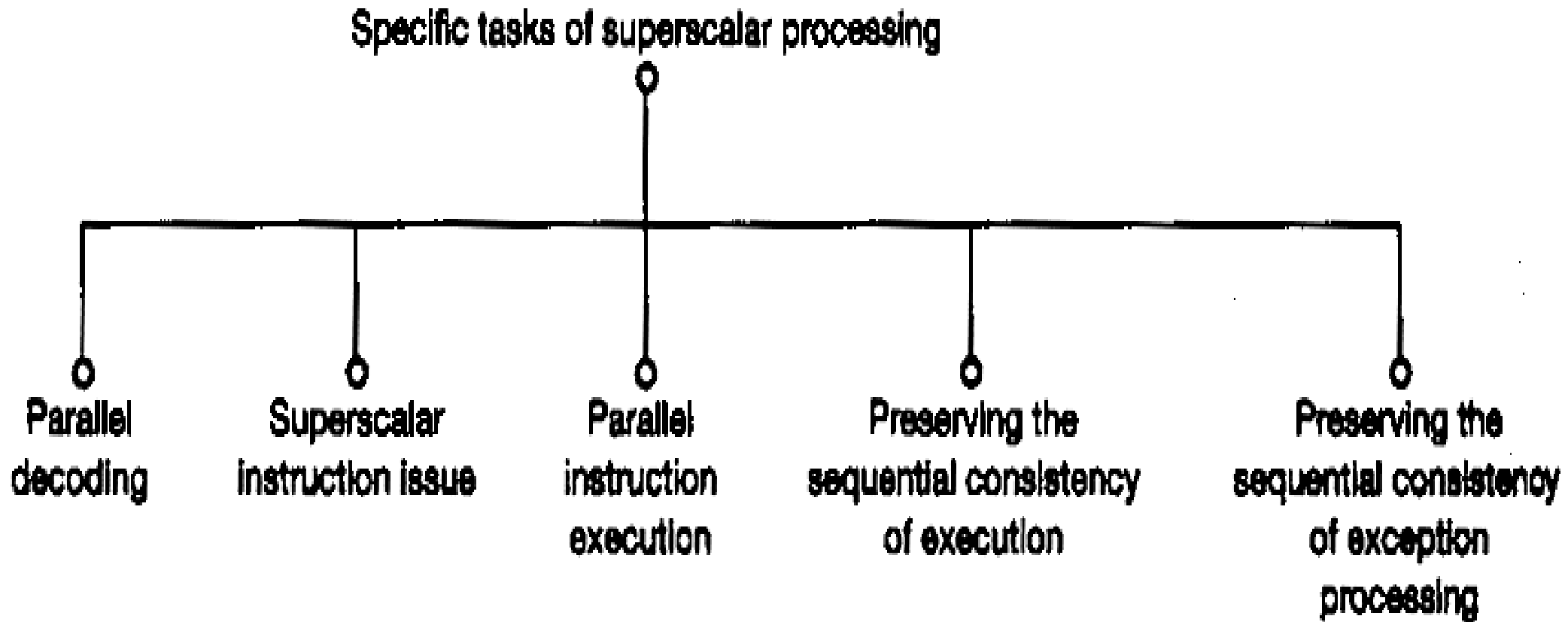


# Evolution of superscalar processor



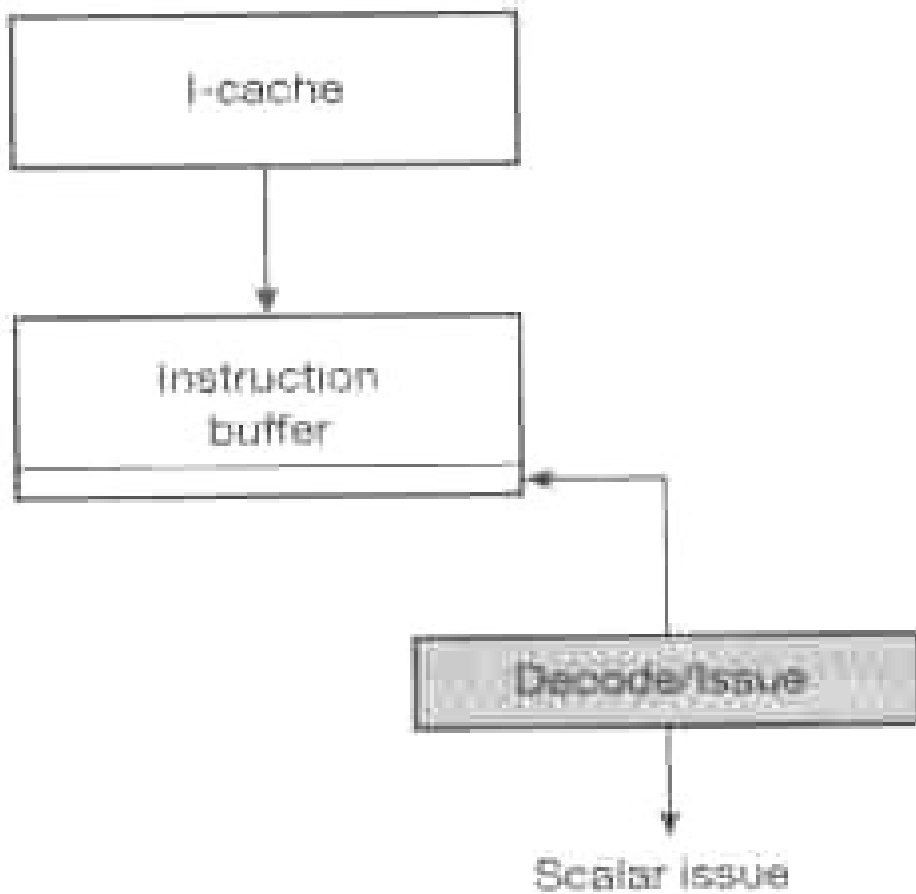
<sup>1</sup>The Nx586 has scalar issue for CISC instructions but a 3-way superscalar core for converted RISC instructions

# Specific tasks of superscalar processing

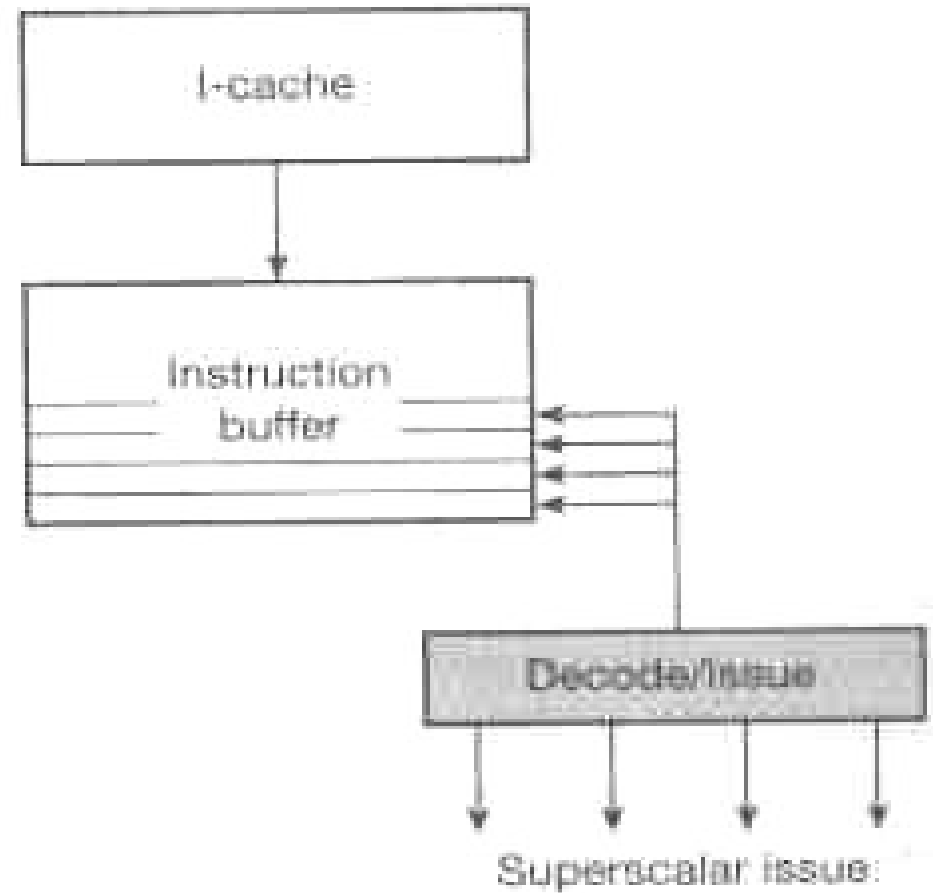


# Parallel decoding {and Dependencies check}

- What need to be done



Typical FX pipeline layout

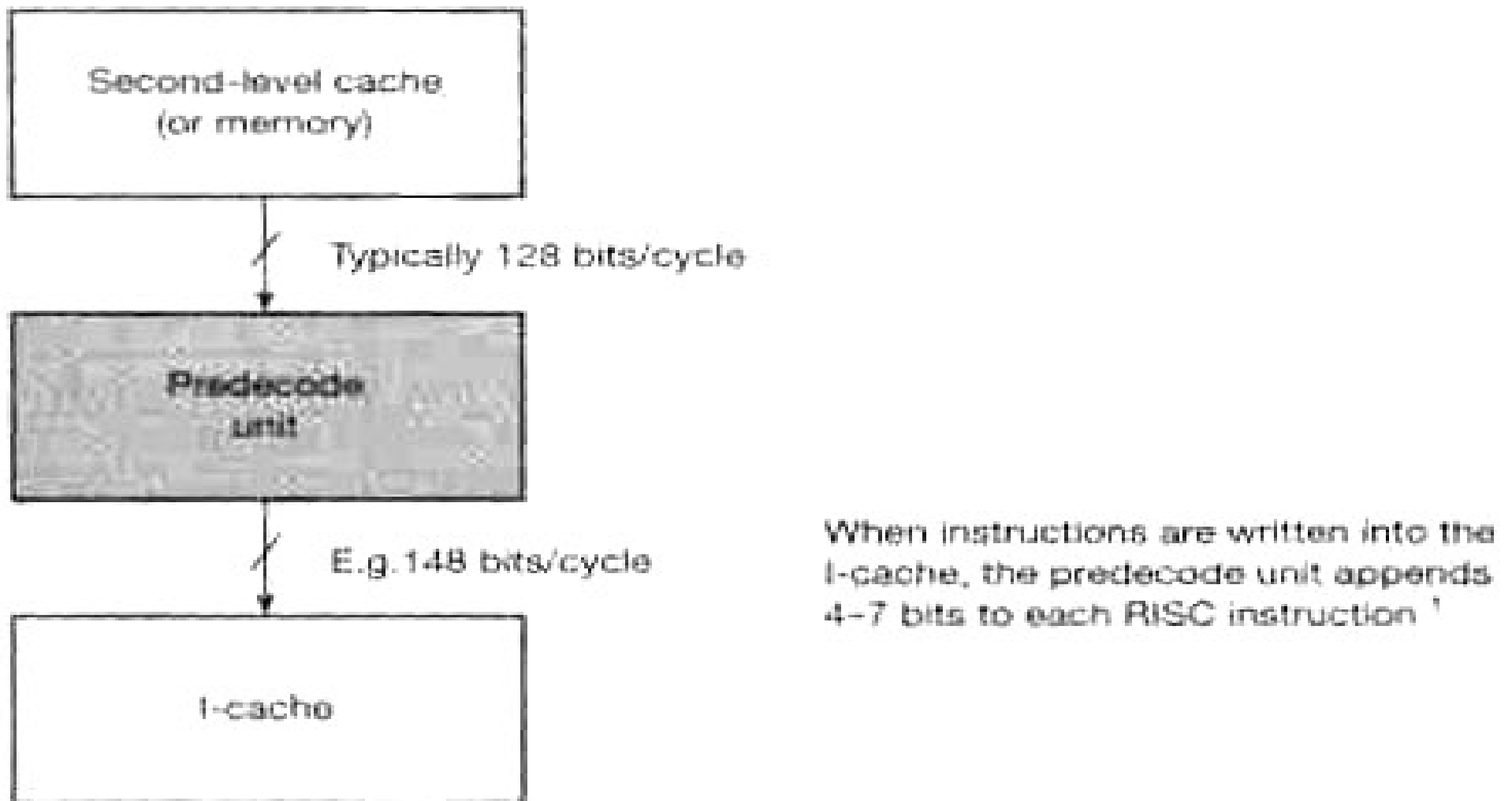


# Decoding and Pre-decoding

- Superscalar processors tend to use 2 and sometimes even 3 or more pipeline cycles for decoding and issuing instructions
- >> Pre-decoding:
  - shifts a part of the decode task up into loading phase
  - resulting of pre-decoding
    - the instruction class
    - the type of resources required for the execution
    - in some processor (e.g. UltraSparc), branch target addresses calculation as well
  - the results are stored by attaching 4-7 bits
- + shortens the overall cycle time or reduces the number of cycles needed



# The principle of predecoding



<sup>1</sup> In the AMD K5, which is an x86-compatible CISC processor, the predecode unit appends 5 bits to each byte

# Number of predecode bits used

**Table 7.1** Number of predecode bits used.

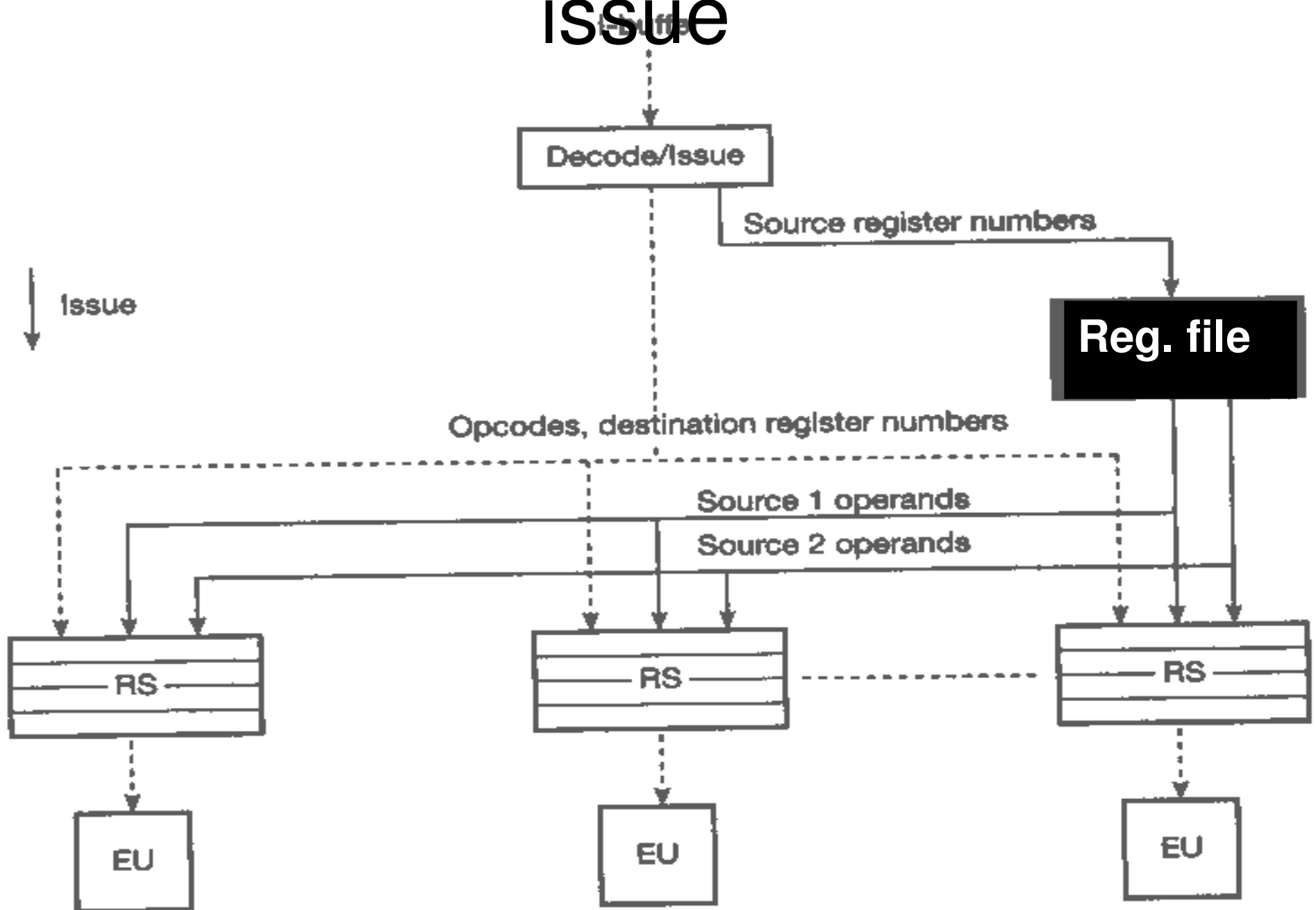
<i>Type/year of first volume shipment</i>	<i>Number of predecode bits appended to each instruction</i>
PA 7200 (1995)	5
PA 8000 (1996)	5
PowerPC 620 (1996)	7
UltraSparc (1995)	4
HAL PM1 (1995)	4
AMD K5 (1995)	5 <sup>†</sup>
R10000 (1996)	4

<sup>†</sup>In the K5, 5 predecode bits are added to each byte

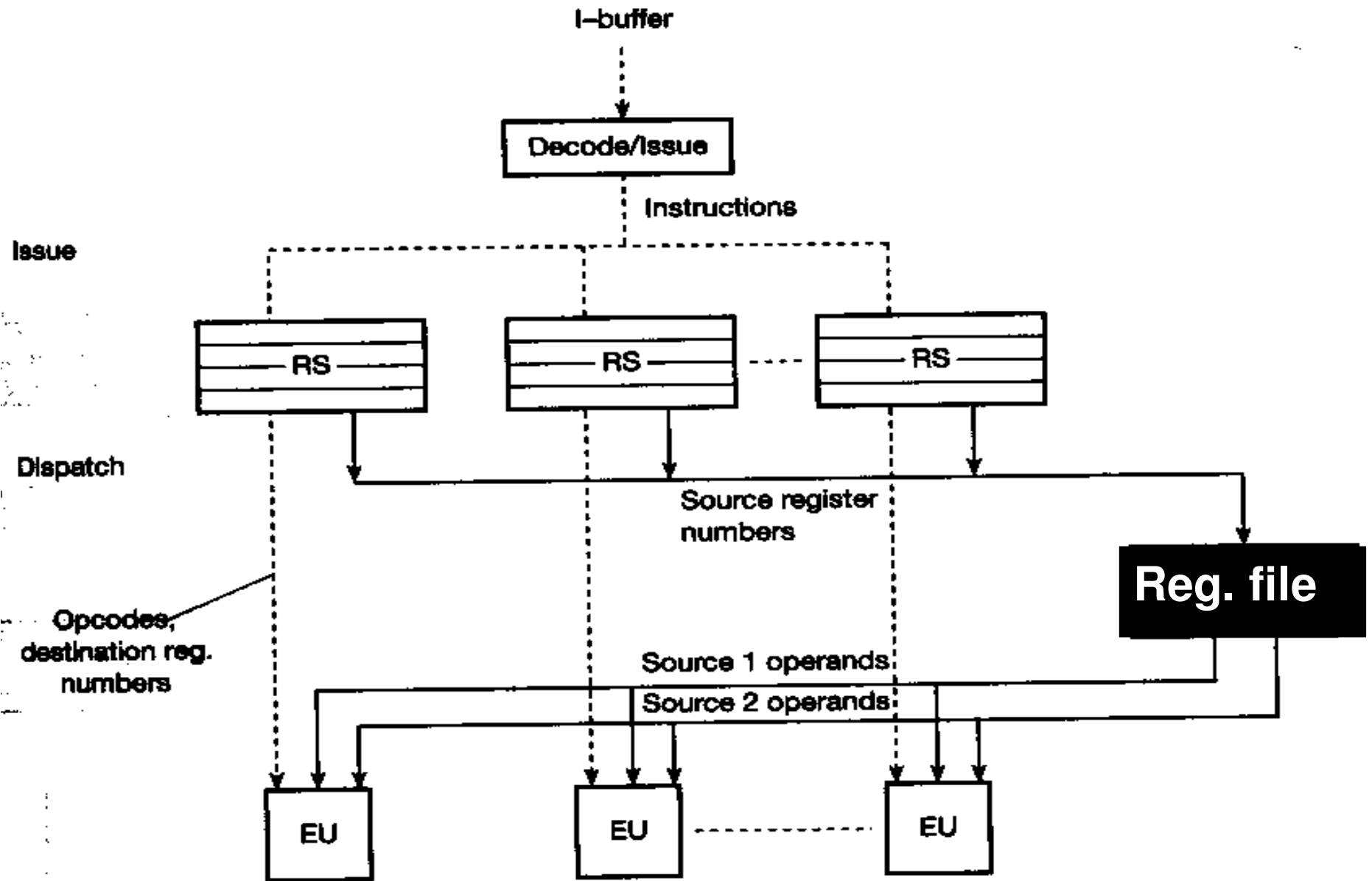
# Number of read and write ports

- how many instructions may be written into (input ports) or
- read out from (output ports) a particular shelving buffer in a cycle
- depend on individual, group, or central reservation stations

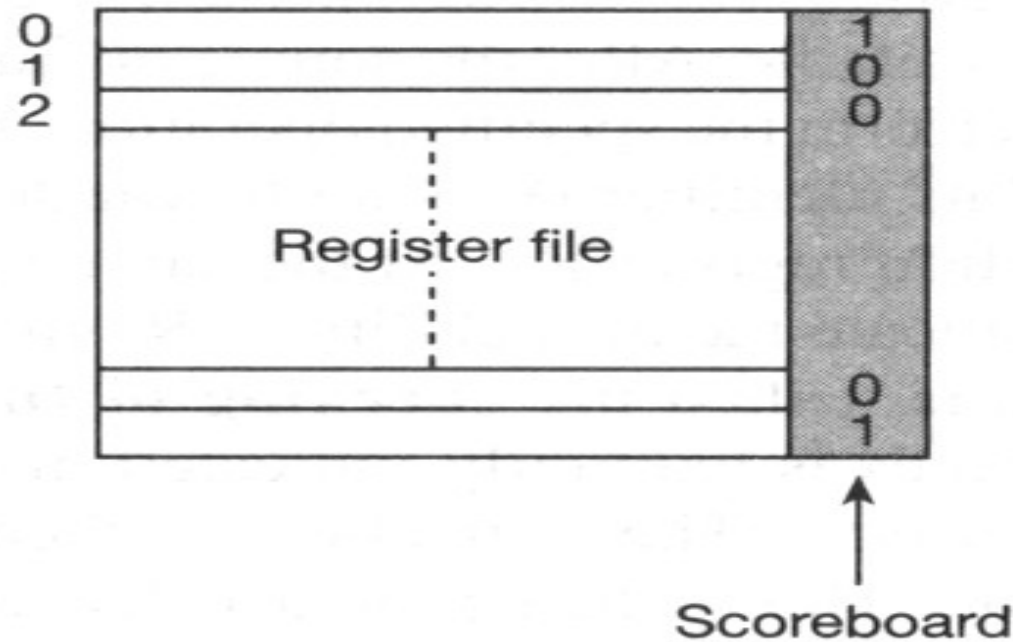
# Operand fetch during instruction issue



# Operand fetch during instruction dispatch



- Scheme for checking the availability of operands: The principle of scoreboarding



Interpretation:

- 0 ← When an instruction is issued the scoreboard entry corresponding to the destination register is reset to '0'
- 1 ← When the execution of an instruction is completed, and the result is written into the destination register, the corresponding scoreboard entry is set to '1'

# Schemes for checking the availability of operand

## Schemes for checking the availability of operands



### **Direct check of the scoreboard bits**

The availability of source operands is not explicitly indicated in the RS.

Thus, the scoreboard bits are tested for availability

*Usually employed if operands are fetched during instruction dispatch, as assumed below*

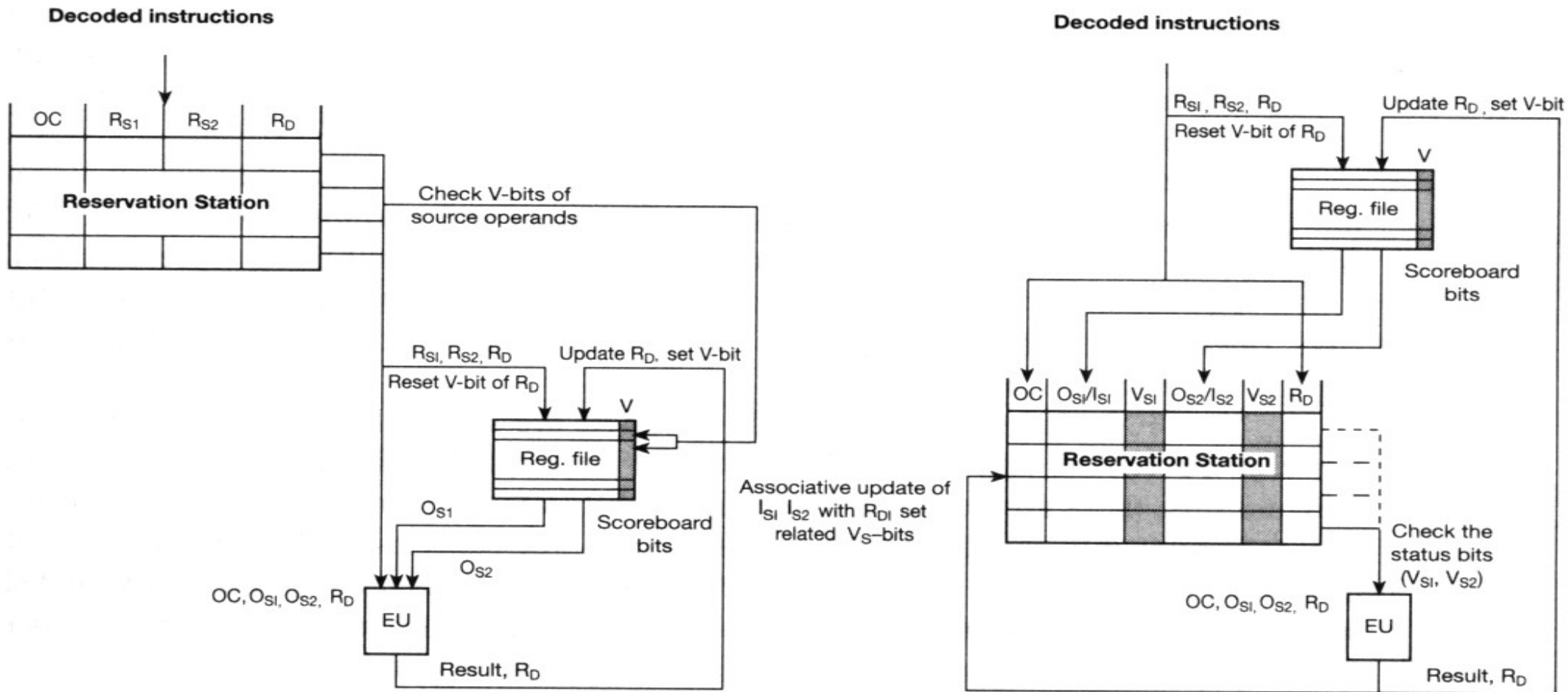
### **Check of the explicit status bits**

The availability of source operands is explicitly indicated in the RS.

These explicit status bits are tested for availability

*Usually employed if operands are fetched during instruction issue, as assumed below*

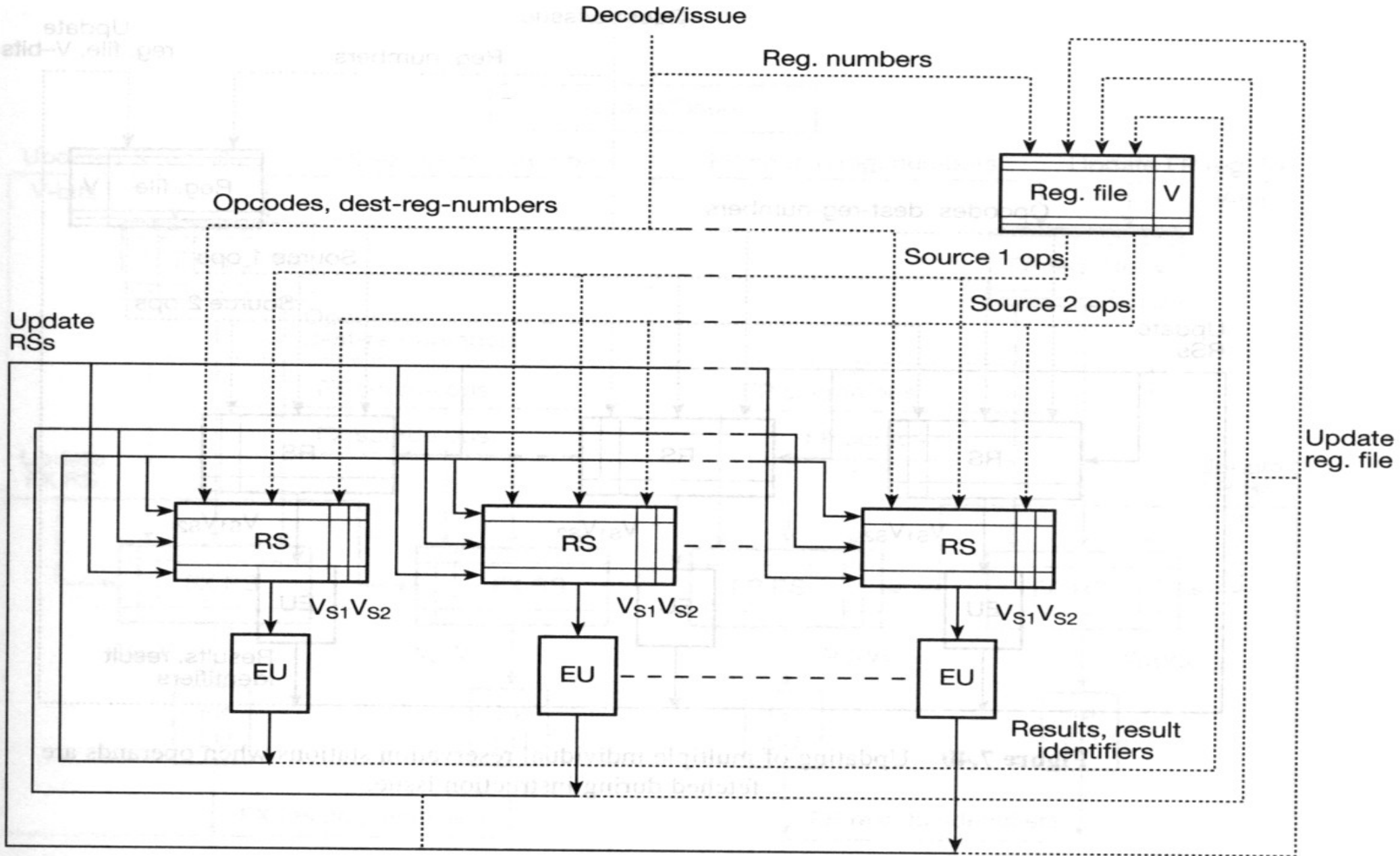
# Operands fetched during dispatch or during issue



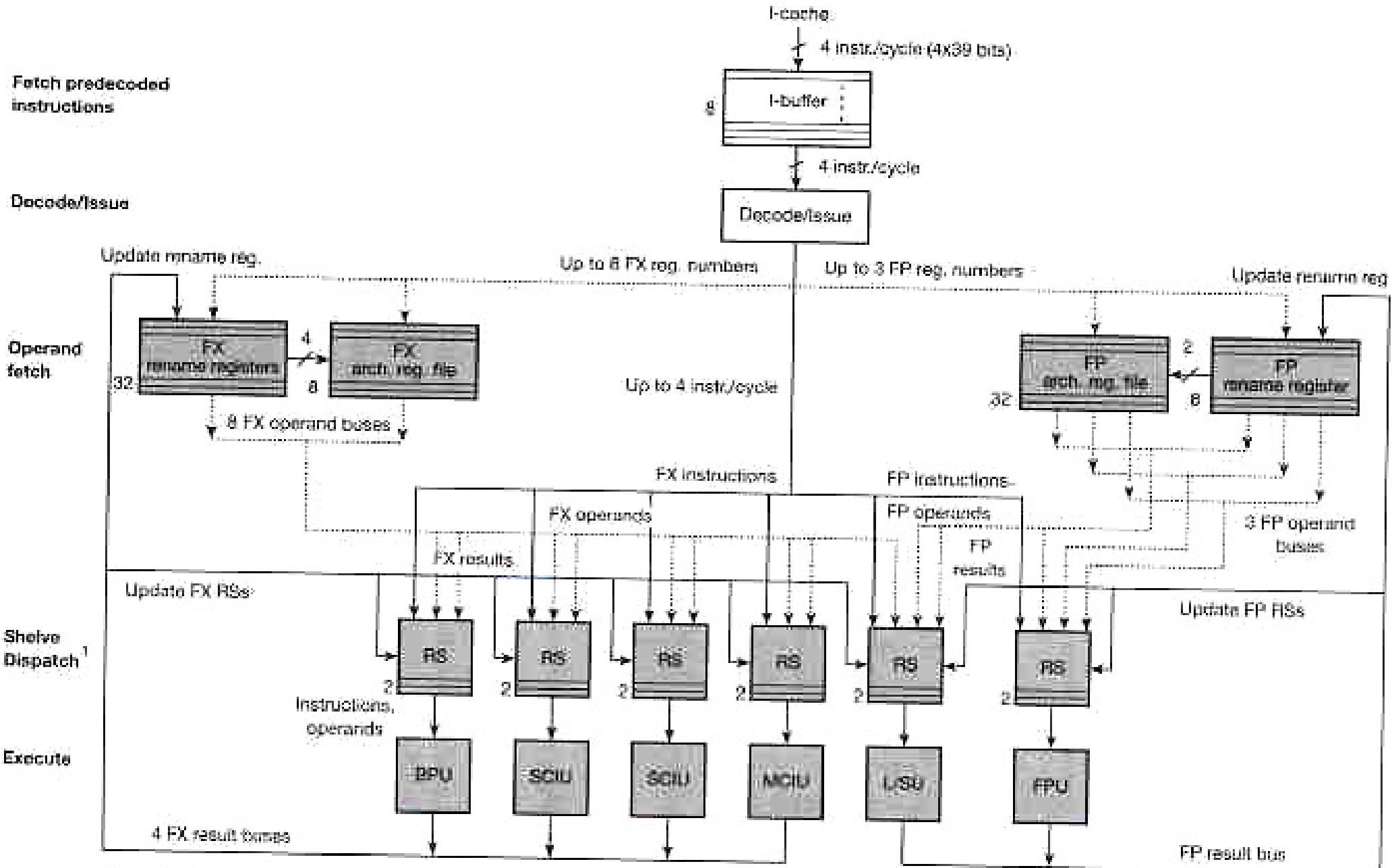
- OC: Operation code
- RS1, RS2: Source register numbers
- RD: Destination register number
- OS1, OS2: Source operand values
- IS1, IS2: Source operand identifiers (tags)
- VS1, VS2: Source operand valid bits
- RS: Reservation station



# Use of multiple buses for updating multiple individual reservation stations



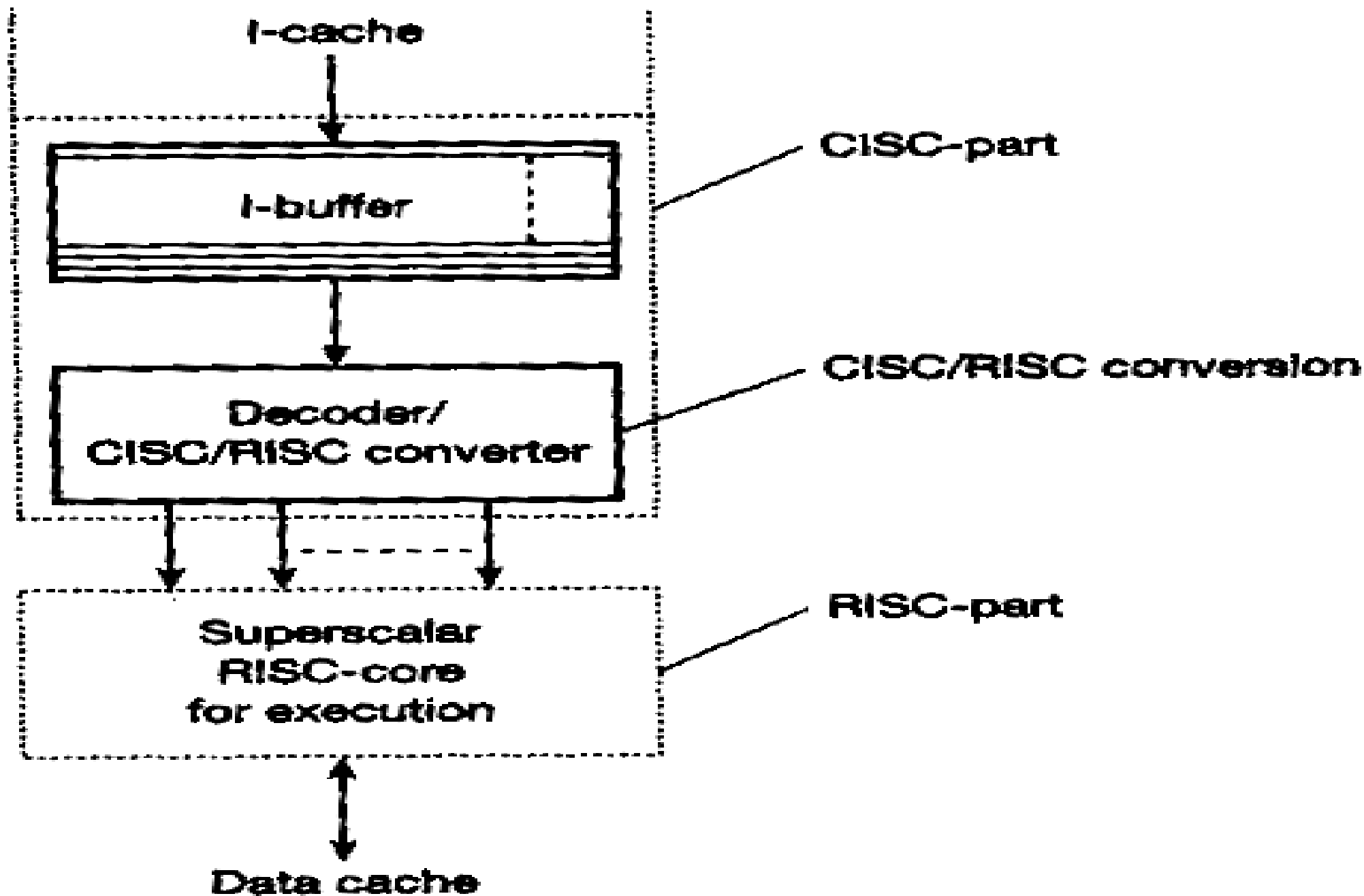
# Internal data paths of the powerpc 604



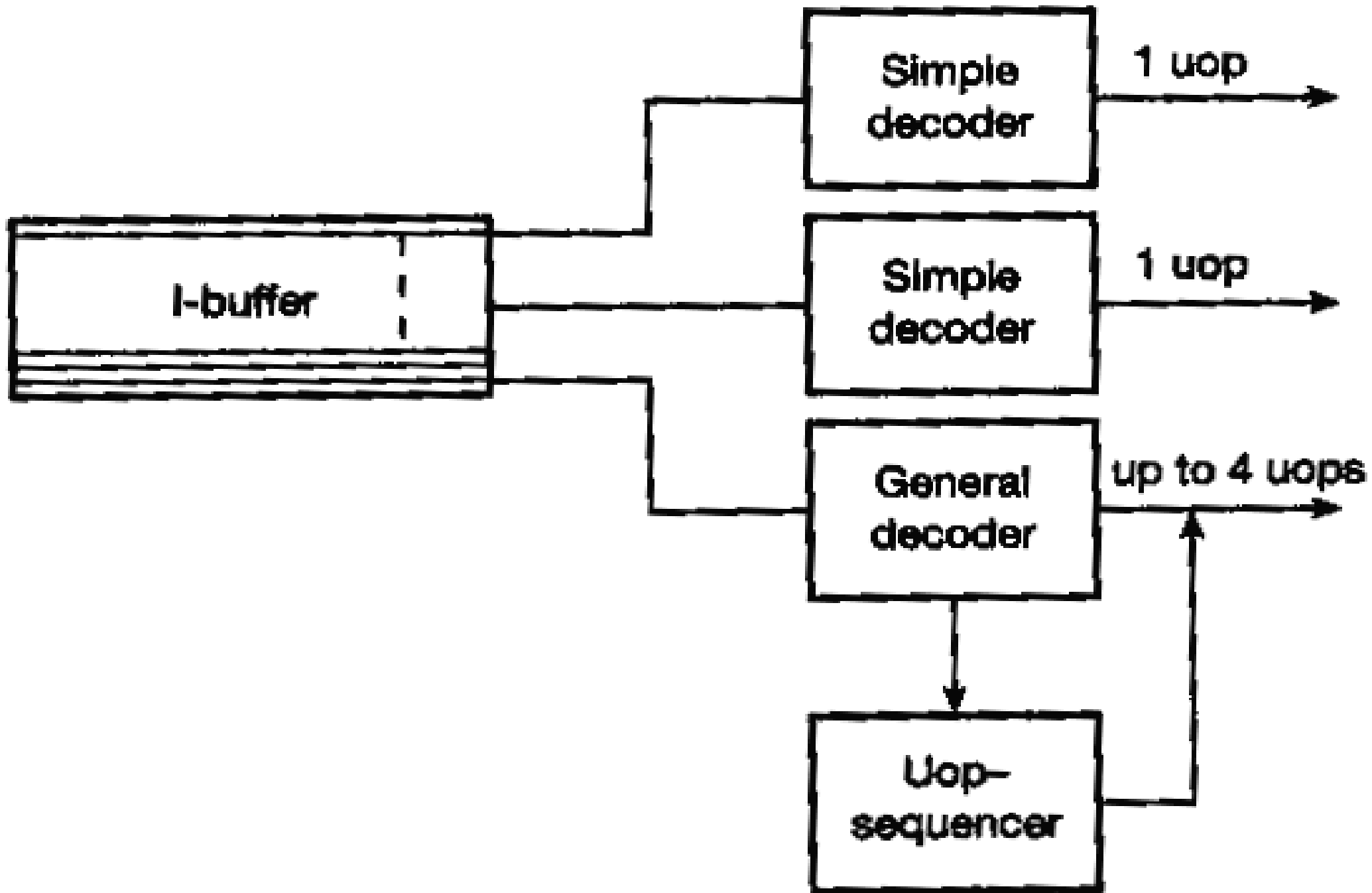
# Implementation of superscalar CISC processors using superscalar RISC core

- CISC instructions are first converted into RISC-like instructions <during decoding>.
  - Simple CISC register-to-register instructions are converted to single RISC operation (1-to-1)
  - CISC ALU instructions referring to memory are converted to two or more RISC operations (1-to-(2-4))
    - SUB EAX, [EDI]
      - converted to e.g.
    - MOV EBX, [EDI]
    - SUB EAX, EBX
  - More complex CISC instructions are converted to long sequences of RISC operations (1-to-(more than 4))
- On average one CISC instruction is converted to 1.5-2 RISC operations

# The principle of superscalar CISC execution using a superscalar RISC core

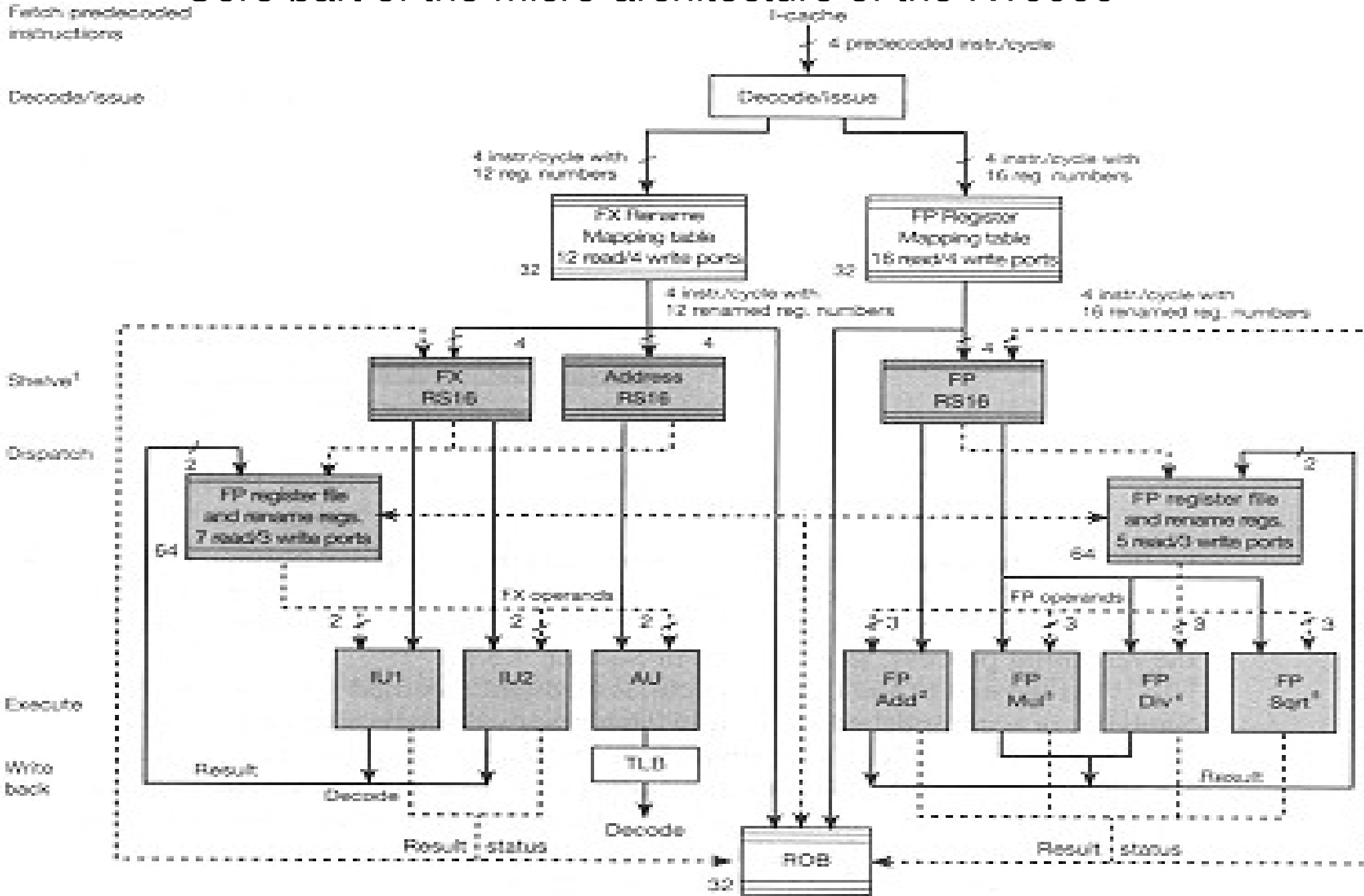


PentiumPro: Decoding/converting CISC instructions to RISC operations (are done in program order)

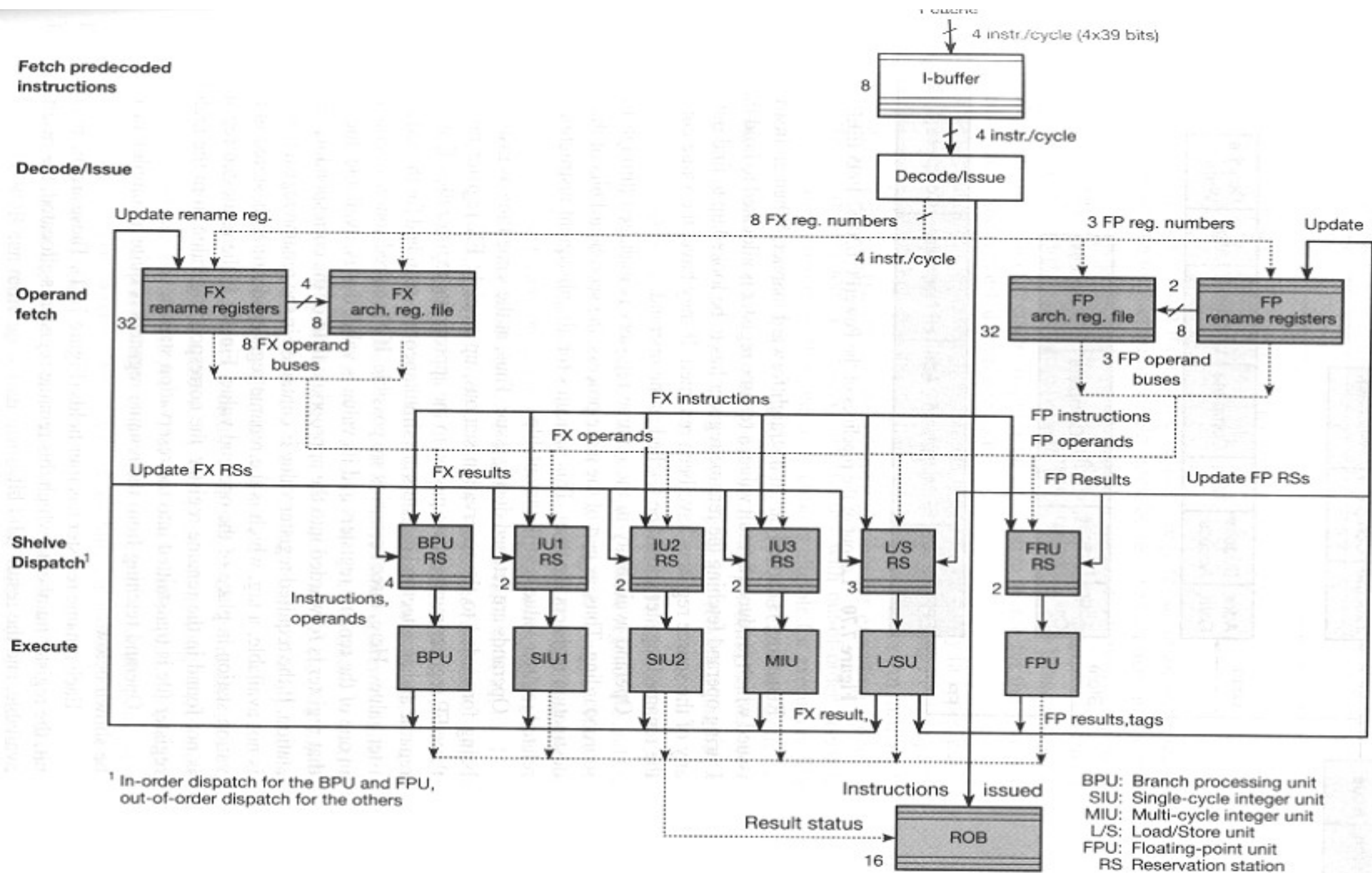


# Case Studies: R10000

## Core part of the micro-architecture of the R10000

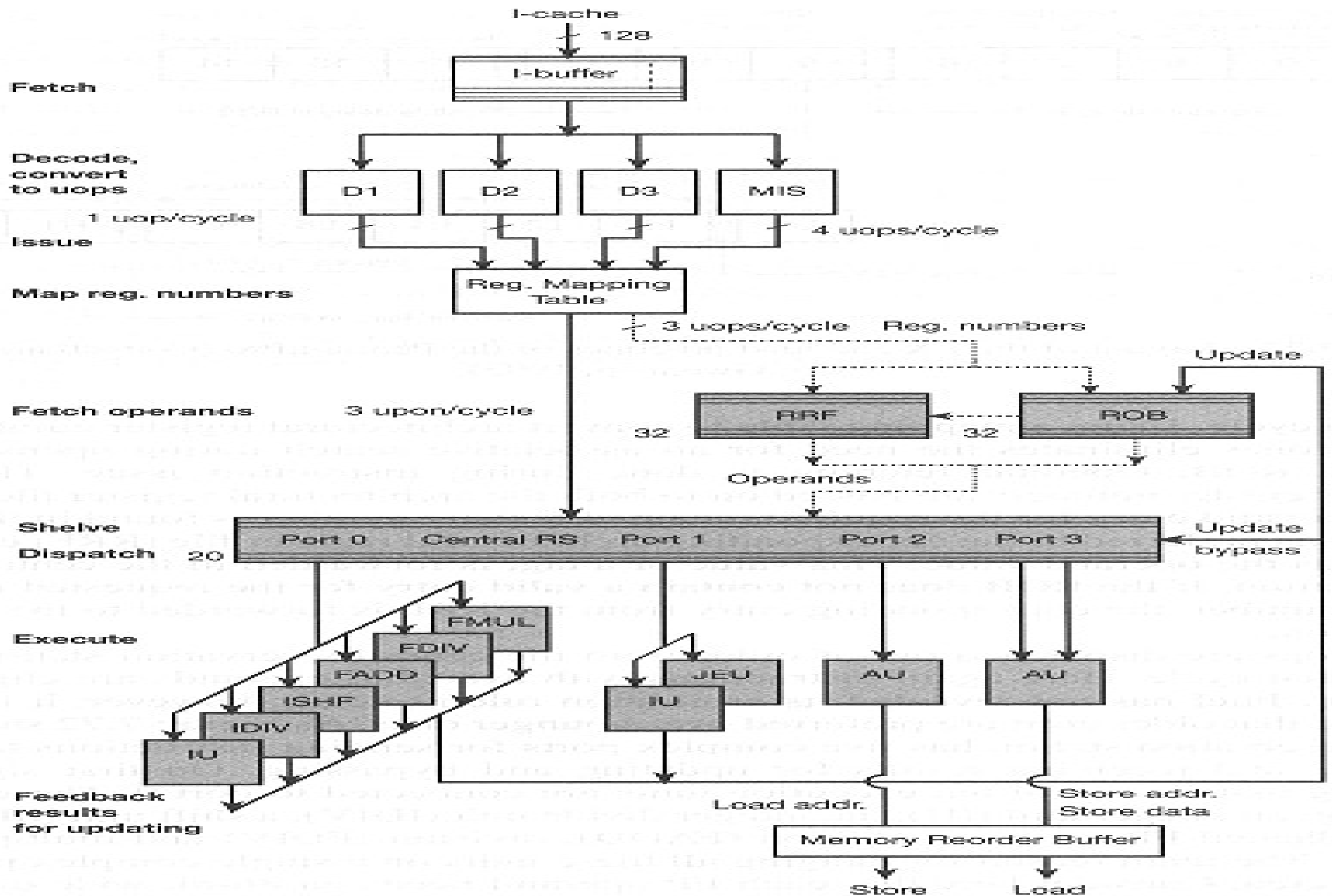


# Case Studies: PowerPC 620



# Case Studies: PentiumPro

## Core part of the micro-architecture





# PentiumPro Long pipeline: Layout of the FX and load pipelines

