National Institute of Technology Karnataka Surathkal
Department of Information Technology

IT 301 Parallel Computing

Introduction

Dr. Geetha V

Assistant Professor

Dept of Information Technology

NITK Surathkal

Index

- 1: Moore's Law and Need for Parallel Processing
- 2. Parallel Processing in Uniprocessor System
  - Bit Level , Instruction level, Task level and Data level parallelism
- 3: Multiprocessor and Multicores System

1: Introduction

Course Plan: Theory:

Part A: Parallel Computer Architectures

Week 1,2,3: ***Introduction to Parallel Computer Architecture:***

Parallel Computing,

Parallel architecture,

bit level, instruction level , data level and task level parallelism.

Instruction level parallelisms: pipelining(Data and control instructions),

scalar and superscalar processors,

vector processors.

Parallel computers and computation.

1: Moore's Law and Need for Parallel Processing

- Chip performance doubles every 18-24 months
- Power consumption is proportional to frequency of the system
- Limitations of serial Computing
  - Heating issues
  - Limit to transmissions seeds
  - Leakage currents
  - Limit to miniaturization

- Smaller transistor = faster processors
- Faster processor = increased power consumption
- Increased power consumption = increased heat
- Increased heat = unreliable processors

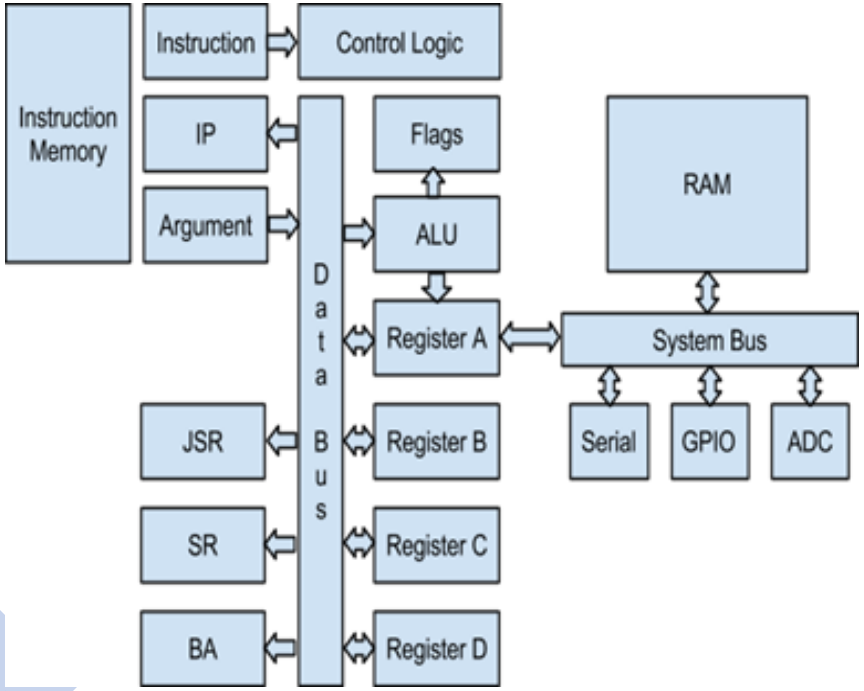
- Parallel processing in Uniprocessor systems
  - Bit level parallelism, Instruction level parallelism (pipelining, Superscalar processors), Data parallelism (vector processors) and task level parallelism (threads)
- Parallel processing in Multiprocessor and Multicore systems
  - Multi core processor, clusters, grids

2. Parallel Processing in Uniprocessor System

- Bit level parallelism
- Instruction level parallelism (pipelining, Superscalar processors)
- Data parallelism (vector processors)
- Task level parallelism (threads)

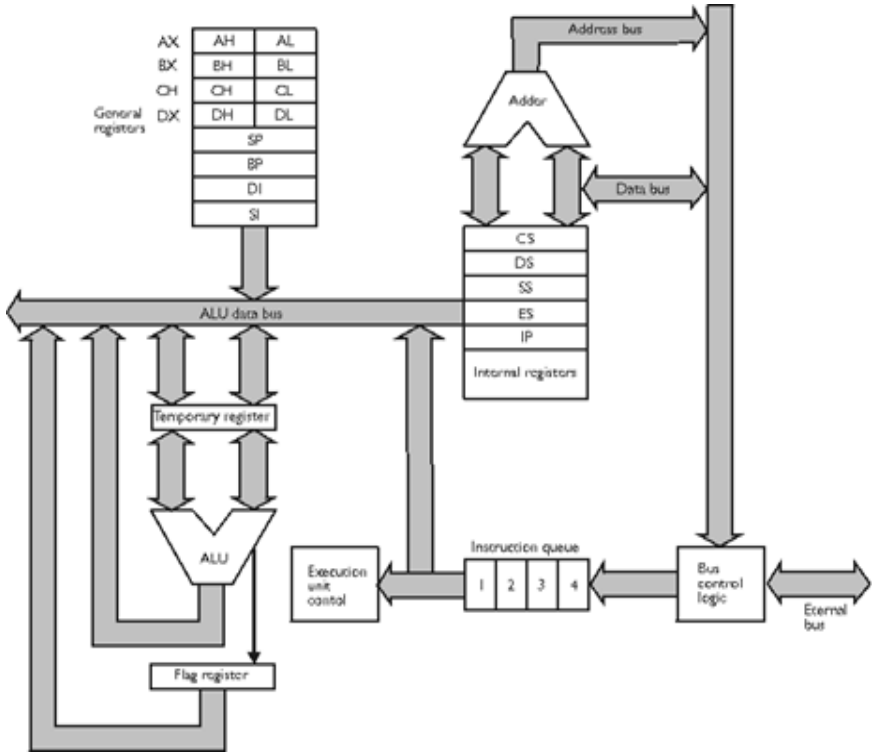
2. Parallel Processing in Uniprocessor System

- Bit level parallelism
- Adding 16-bit number in 8-bit processor



http://ryanohs.com/2015/09/high-level-architecture/

- Adding 16-bit number in 16-bit processor



http://www.c-jump.com/CIS77/CPU/VonNeumann/lecture.html#V77_0010_von_neumann

2. Parallel Processing in Uniprocessor System

- Bit Level Parallelism (eg. 1516H +2829H)
  - Adding 16-bit number in 8-bit processor

```
MVI A, #16H
MVI B, #29H
ADD B
MVI A, #15H
MVI B, #28H
ADC B
MOV D, A
```

  - Adding 16-bit number in 16-bit processor

```
MVI AX, #1516H
MVI BX, #2829H
ADD AX, BX
MOV CX, AX
```

2. Parallel Processing in Uniprocessor System

- Bit Level Parallelism
  - 8-bit processor (1972-1974)

Intel 8008, Intel 8080

  - 16-bit processor (1979 – 1982)

Intel 8086, Intel 286

  - 32-bit processor (1985 -2006)

Intel 386, Intel 486, P5 (Pentium)

Pentium Pro, Pentium II, Pentium III, Pentium M, Intel Core, Dual-core Xeon LV

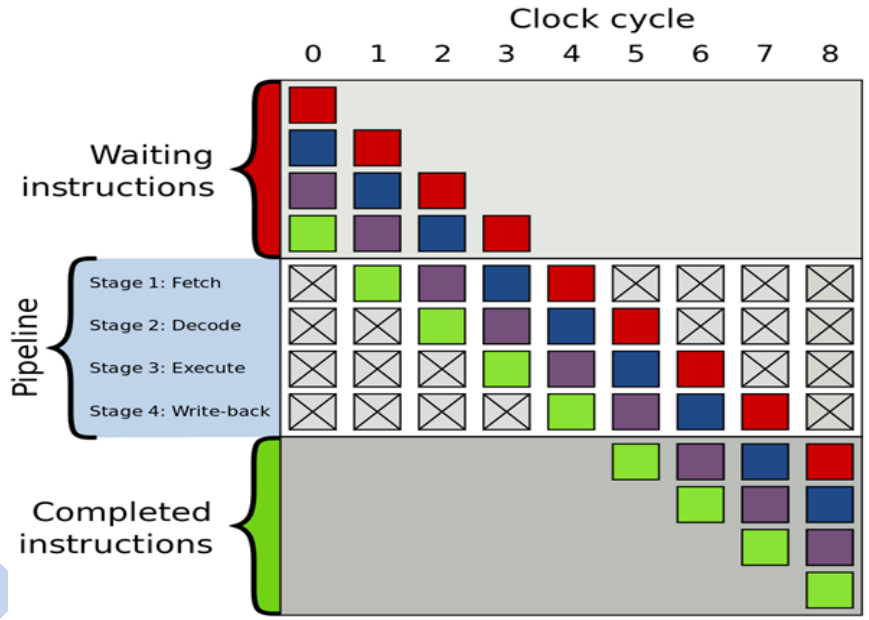
  - 64-bit processor (2004 onwards)

Itanium , Itanium 2 (IA-64), Pentium 4F, Pentium D, Intel 6: Xeon (NetBurst), Intel Core 2, Intel Pentium Dual Core, Celeron, Celeron M; Nehalem: Intel Pentium, Core i3, i5, i7, Xeon…..etc

2. Parallel Processing in Uniprocessor System

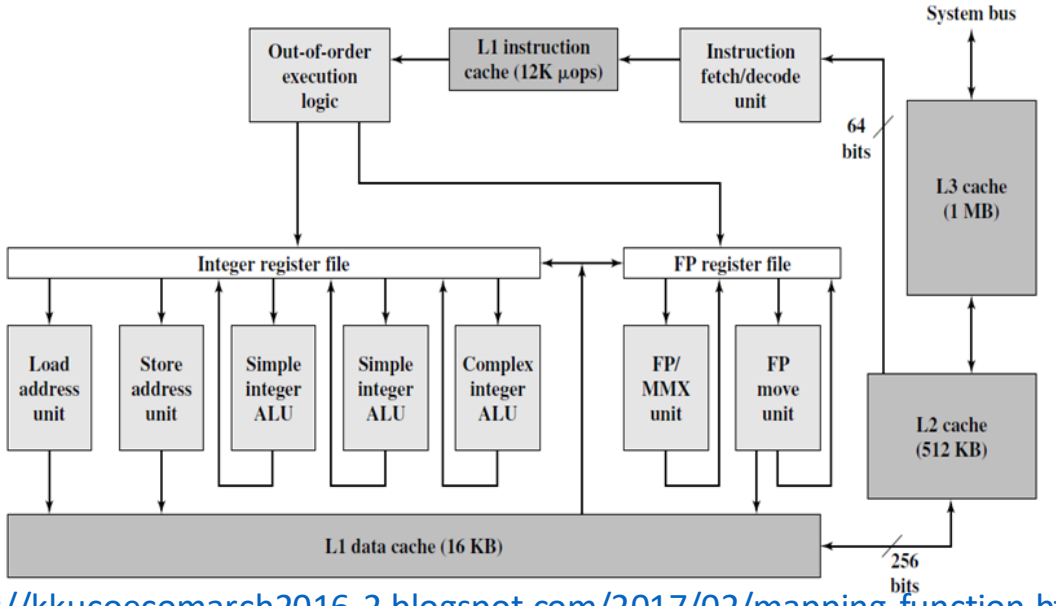
- Instruction level parallelism (pipelining, Superscalar processors)

- Scalar Processors
  - Goal towards one instruction execution per cycle



https://en.wikipedia.org/wiki/Instruction_pipelining

- Superscalar Processors
  - Multiple Instruction per cycle



http://kkucoecomarch2016-2.blogspot.com/2017/02/mapping-function.html

2. Parallel Processing in Uniprocessor System

- Instruction level parallelism (pipelining, Superscalar processors)
  - Scalar Processors
    - •Goal towards one instruction execution per cycle
    - •Reduced Instruction Set Computer (RISC) Scalar processors
      - Intel i860, Motorola MC8810, SUN's SPARC CY7C601 etc
    - •Complex Instruction Set Computer (CISC) scalar Processors
      - Intel 386, 486; Motorola's 68030, 68040; etc
  - Superscalar Processors
    - •Multiple Instruction per cycle
      - Pentium , Pentium Pro, Pentium II, Pentium III Motorola 88110 etc.

2. Parallel Processing in Uniprocessor System

- Task level parallelism

Task parallelism or function level parallelism is a form of parallelization of computer code across multiple processors in parallel computing environment

- Models
  - •Task dependency graph model
  - •Master Slave Model
  - •Pipeline/Producer Consumer model

- Data level parallelism

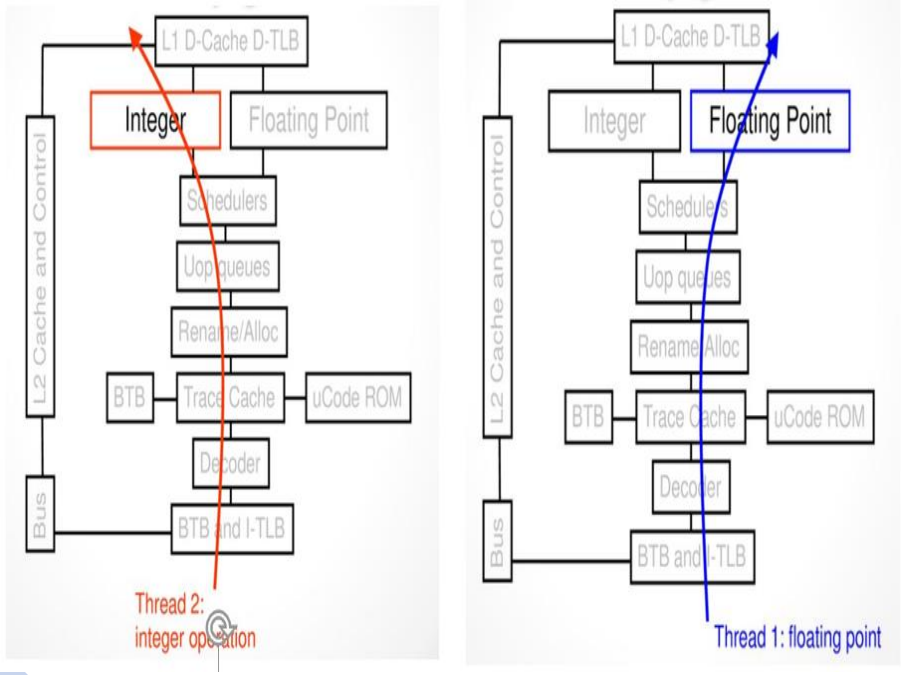
Data parallelism is a form of parallelization of computing across multiple processors in parallel computing environment.

- Architectures
  - •Vector Processors
  - •Single Instruction Multiple Data (SIMD)

2. Parallel Processing in Uniprocessor System

- Task level parallelism

Intel Xeon hyperthreading

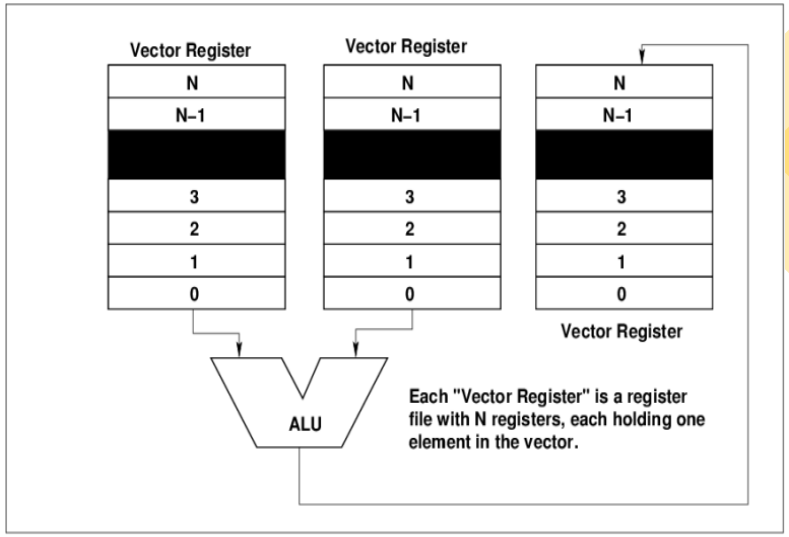


https://slideplayer.com/slide/14851286/

- Data level parallelism

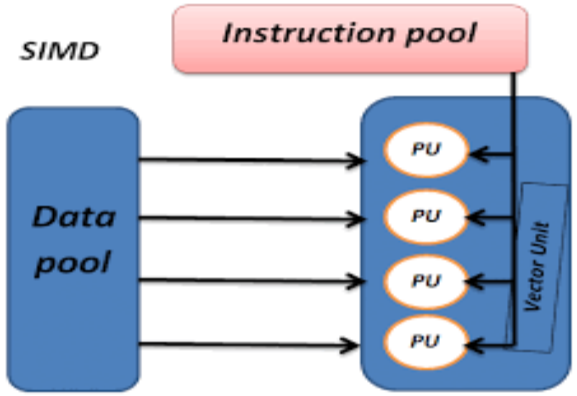
- Vector Processor

Eg. Cray 1



- SIMD

Eg. ILLIAC IV

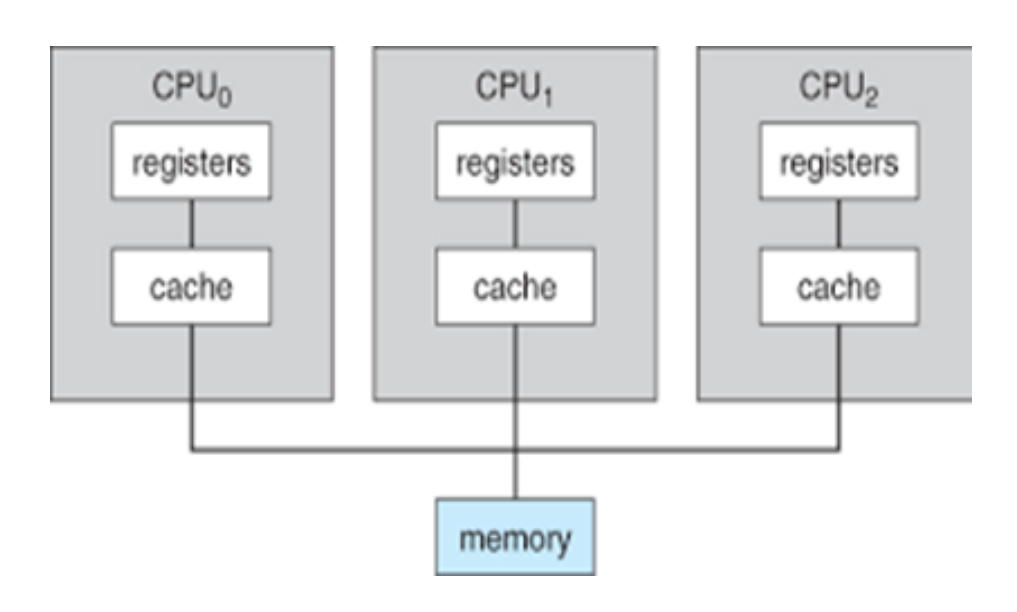


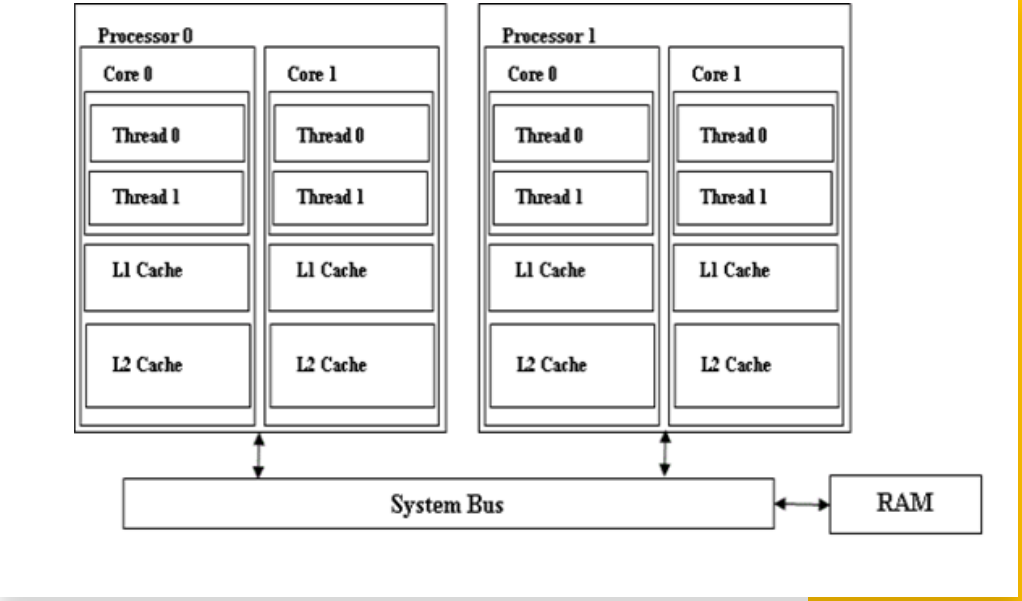
https://www.researchgate.net/profile/Ahmed_Al-Araji/publication/342107319/figure/fig1/AS:901279303864320@1591893139023/Figure-1-Big-picture-of-SIMD-architecture.ppm
https://www.ausairpower.net/XIMG/vector-1.png

3: Multiprocessor and Muticore system

- Multiprocessor system: Two or more CPU within single computer system

- Multicore processor: Multiple Execution units (cores) o the same chip.

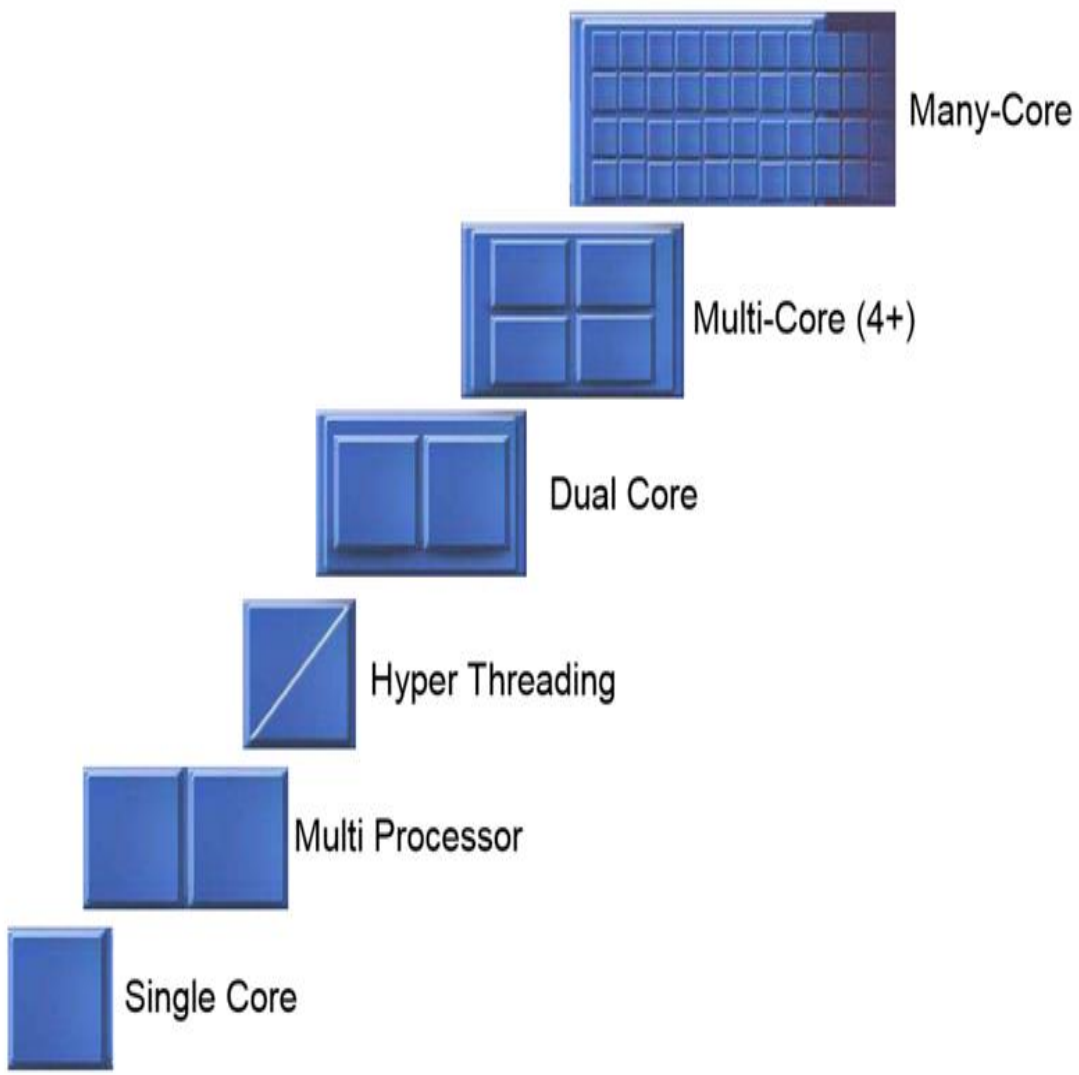




3: Multiprocessor and Muticore system

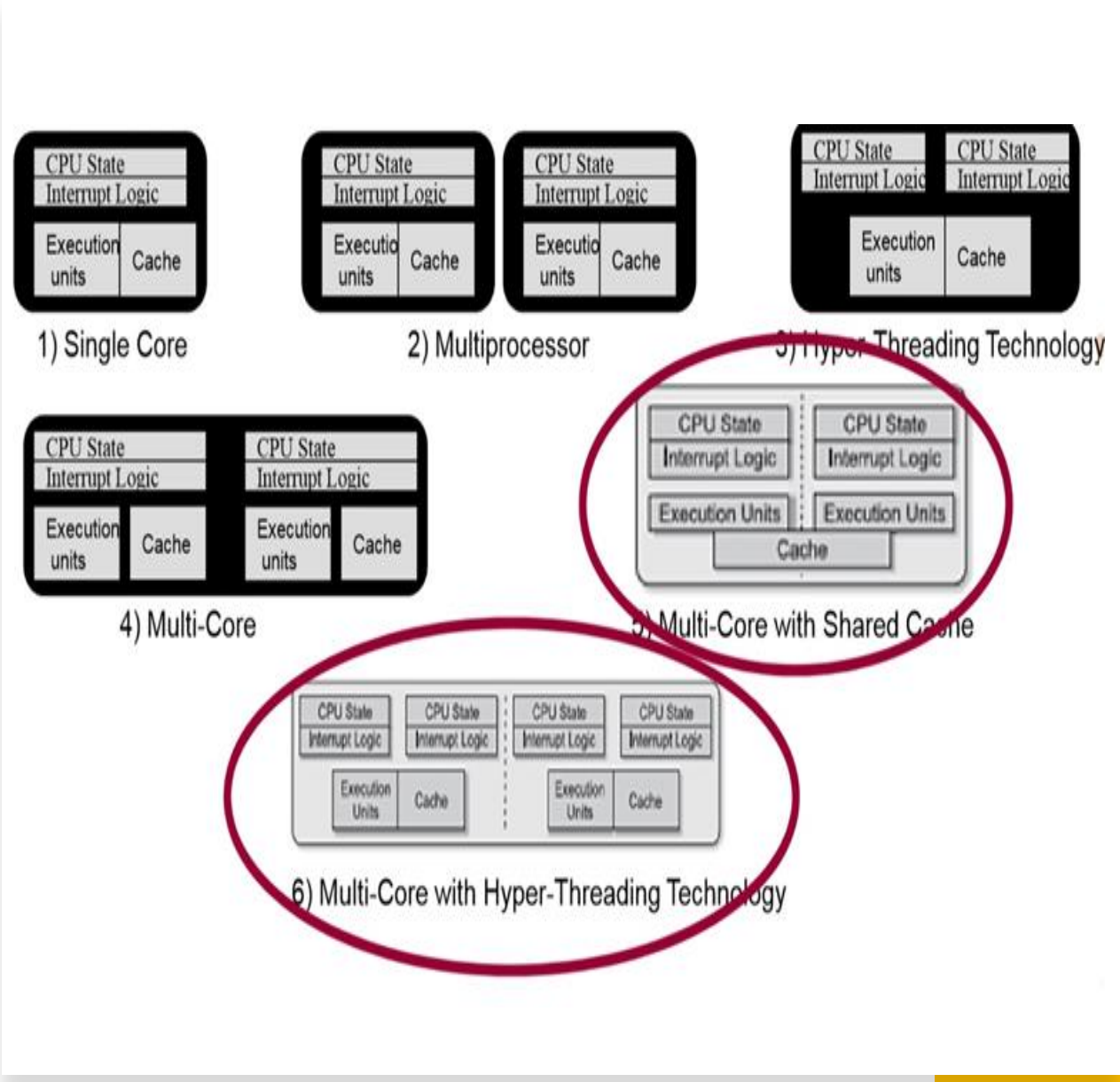
- Multiprocessor system: Two or more CPU within single computer system

- Multicore processor: Multiple Execution units (cores) o the same chip.



3: Multiprocessor and Muticore system

- Single Core: One execution unit in one chip
- Multiprocessor : Two or more CPU within single computer system
- Hyperthreading : Task level parallelism
- Multicore processor: Multiple Execution units (cores) on the same chip.
- Multicore with shared cache: All core shares same cache
- Multicore with Hyperthreading: task level parallelism



Reference

Textbooks and/or Reference Books:

1. Professional CUDA C Programming – John Cheng, Max Grossman, Ty McKercher, 2014
2. Wilkinson, M. Allen, ”Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers”, Pearson Education, 1999
3. I. Foster, ”Designing and building parallel programs”, 2003
4. Parallel Programming in C using OpenMP and MPI – Micheal J Quinn, 2004
5. Introduction to Parallel Programming – Peter S Pacheco, Morgan Kaufmann Publishers, 2011
6. Advanced Computer Architectures: A design approach, Dezso Sima, Terence Fountain, Peter Kacsuk, 2002
7. Parallel Computer Architecture : A hardware/Software Approach, David E Culler, Jaswinder Pal Singh Anoop Gupta, 2011 8. Introduction to Parallel Computing, Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar, Pearson, 2011

Thank You

National Institute of Technology Karnataka Surathkal
Department of Information Technology

IT 301 Parallel Computing

Scalar Processors – Instruction Pipeline

Dr. Geetha V

Assistant Professor

Dept of Information Technology

NITK Surathkal

Index

- 1: Instruction Pipeline - Introduction
- 2. Five Stage pipeline design
- 3: Issues of pipeline and Design Solutions
- 4: Performance evaluation of pipelining

1: Introduction

Course Plan: Theory:

Part A: Parallel Computer Architectures

Week 1,2,3: ***Introduction to Parallel Computer Architecture:***

Parallel Computing,

Parallel architecture,

bit level, instruction level , data level and task level parallelism.

Instruction level parallelisms: pipelining (Data and control instructions),

scalar processors and superscalar processors,

vector processors.

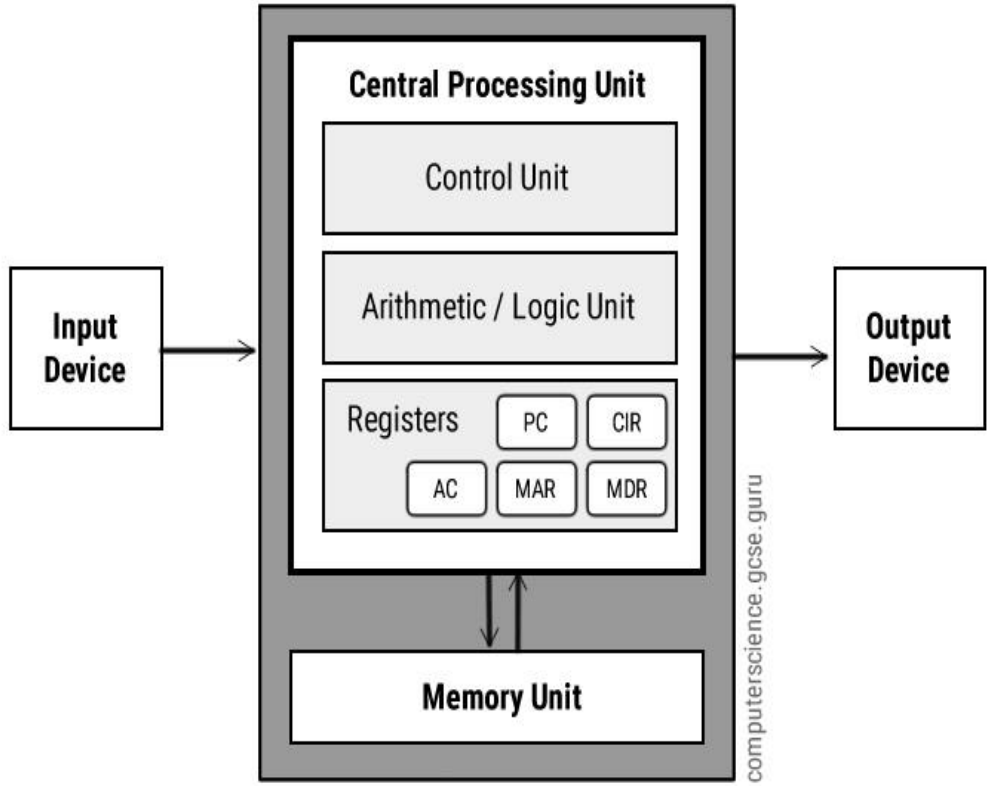
Parallel computers and computation.

1: Instruction Pipeline - Introduction

Features

- Established by John von Neumann in 1945.
- Stored Program Concept
- **PC :** Program Counter
- **CIR:** Current Instruction Register
- **AC:** Accumulator,
- **MAR:** Memory Address Register,
- **MDR:** Memory Data Register
- **Buses:** Address, Data and Control
- **Execution :** One Instruction at a Time

Von Neumann Architecture



https://www.computerscience.gcse.guru/theory/von-neumann-architecture

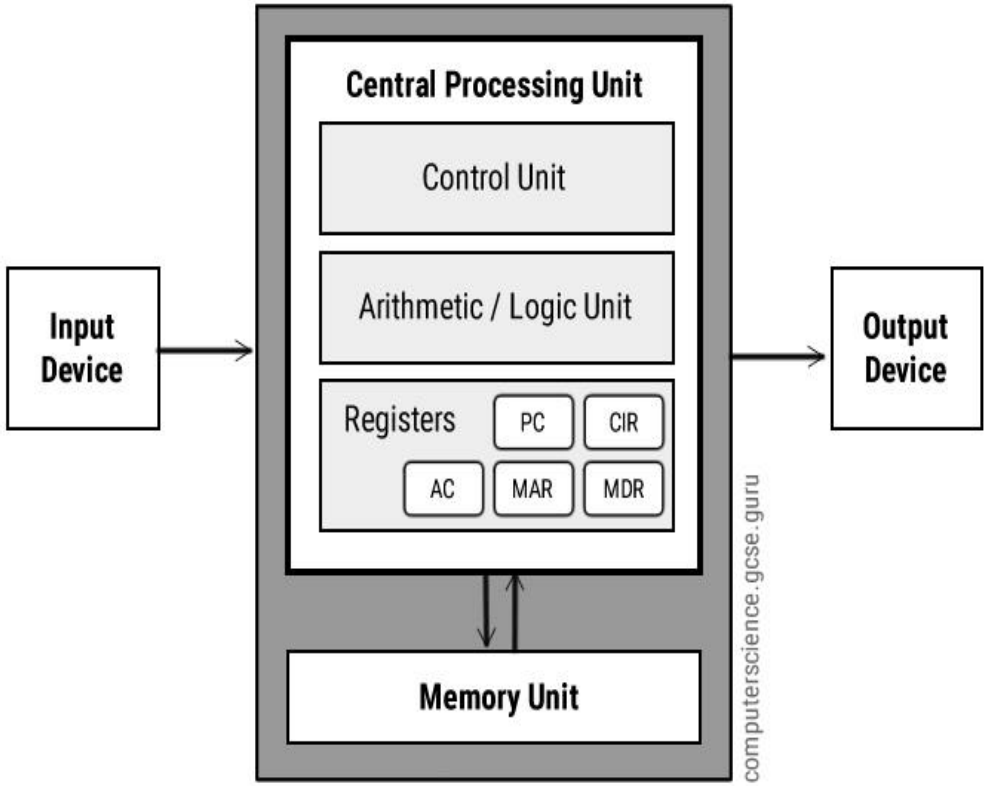
1: Instruction Pipeline - Introduction

Features of RISC

- **RISC:** Reduced Instruction Set Computer
- Fixed **instruction size**
- **Load/Store** Memory Operation
- Few **Addressing Modes**
- **Register to Register** operation

- These features leads towards **pipelined execution**

Von Neumann Architecture



https://www.computerscience.gcse.guru/theory/von-neumann-architecture

2. Five Stage Pipeline Design

- Fetch Stage: The address in program counter is sent to Instruction register and instruction is fetched from memory. PC is incremented.
- Decode Stage: The operands are fetched from register file; Sign extension done if necessary.
- Execution Stage: The instruction is executed.
- Memory Stage: The Load and Store instructions execution.
- Write Back Stage: The results are written back to register file.

Fetch (F)	Decode (D)	Execute (E)	Memory Access (M)	Write Back (W)

2. Five Stage Pipeline Design

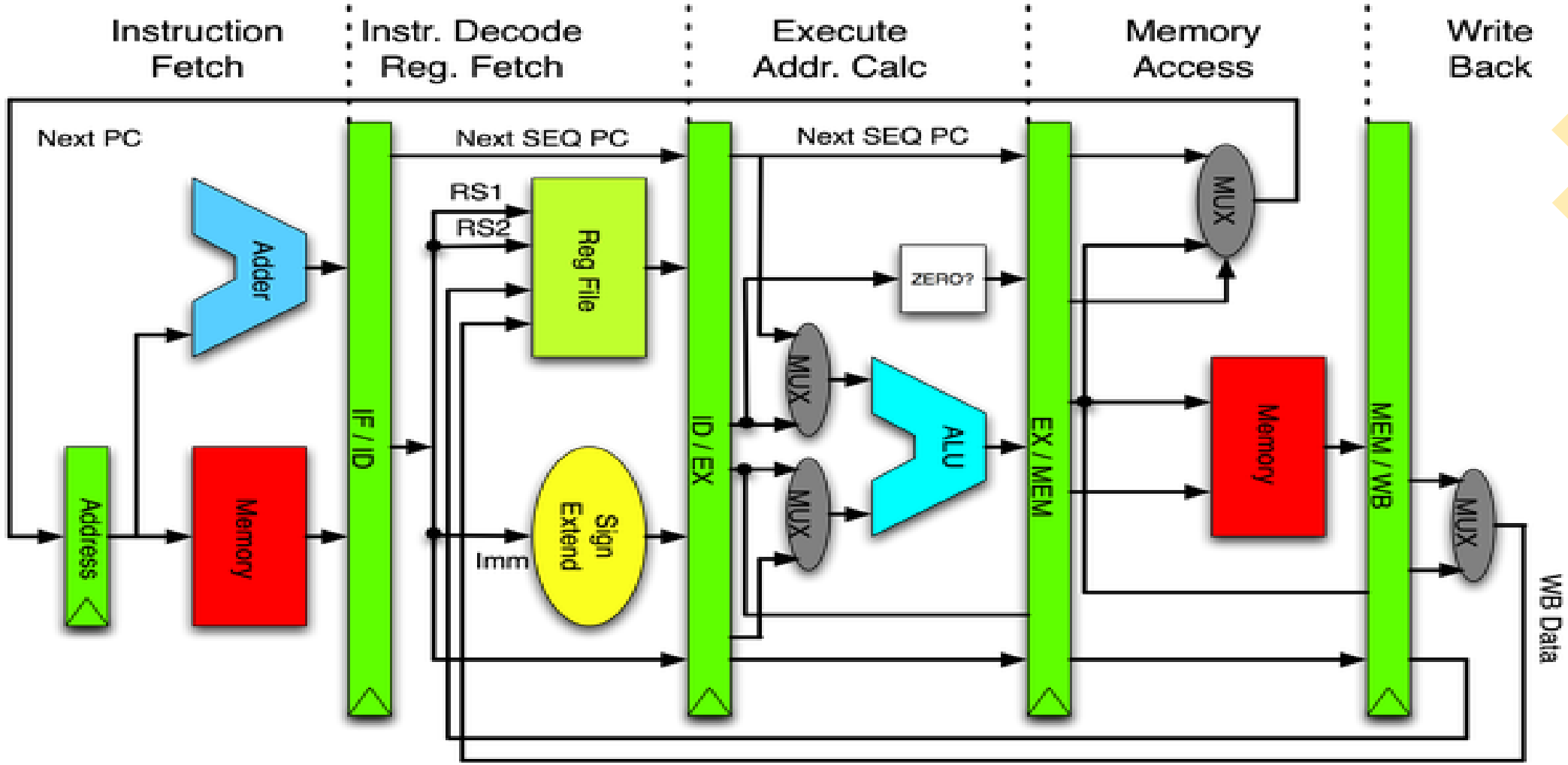


Image from Wikimedia

2. Five Stage Pipeline Design

	Instruction	(F)	(D)	(E)	(M)	(W)	
I_1	Move R5, #2000h	I_1					
I_2	Move R1, #15	I_2	I_1				
I_3	Move R2, #16	I_3	I_2	I_1			
I_4	Move R3, R1	I_4	I_3	I_2	I_1		
I_5	Add R1, R3	I_5	I_4	I_3	I_2	I_1	I1: Complete
I_6	Move R4, R2	I_6	I_5	I_4	I_3	I_2	I2: Complete
I_7	Add R2, R4	I_7	I_6	I_5	I_4	I_3	I3: Complete
I_8	Add R1, R2	I_8	I_7	I_6	I_5	I_4	I4: Complete
I_9	Store [R5], R1	I_9	I_8	I_7	I_6	I_5	I5: Complete
I_{10}			I_9	I_8	I_7	I_6	I6: Complete
I_{11}				I_9	I_8	I_7	I7: Complete
I_{12}					I_9	I_8	I8: Complete
I_{13}						I_9	I9: Complete

- Adding (2x+2y)
- x=15 y=16
- If no pipeline, then this program takes

9 x 5 = 45 cycles

- If 5 stage pipeline is used, then it takes

5 + 8 = 13 cycles

(first instruction takes 'n' stages/cycle) + (n-1) instructions

3: Issues of pipeline and Design Solutions

	Instruction	(F)	(D)	(E)	(M)	(W)	
I_1	Move R5, #2000h	I_1					
I_2	Move R1, #15	I_2	I_1				
I_3	Move R2, #16	I_3	I_2	I_1			
I_4	Move R3, R1	I_4	I_3	I_2	I_1		
I_5	Add R1, R3	I_5	I_4	I_3	I_2	I_1	I1: Complete
I_6	Move R4, R2	I_6	I_5	I_4	I_3	I_2	I2: Complete
I_7	Add R2, R4	I_7	I_6	I_5	I_4	I_3	I3: Complete
I_8	Add R1, R2	I_8	I_7	I_6	I_5	I_4	I4: Complete
I_9	Store [R5], R1	I_9	I_8	I_7	I_6	I_5	I5: Complete
I_{10}			I_9	I_8	I_7	I_6	I6: Complete
I_{11}				I_9	I_8	I_7	I7: Complete
I_{12}					I_9	I_8	I8: Complete
I_{13}						I_9	I9: Complete

- **Data Dependency**

Dependency of instructions due to operand values unavailability

- **Control Dependency**

Dependency due to unresolved decision on control instructions.

3: Issues of pipeline and Design Solutions

Cycle 1: I1 in fetch stage

I_1	Move R5, #2000h
I_2	Move R1, #15
I_3	Move R2, #16
I_4	Move R3, R1
I_5	Add R1, R3
I_6	Move R4, R2
I_7	Add R2, R4
I_8	Add R1, R2
I_9	Store [R5], R1

I1:Move R5, #2000h

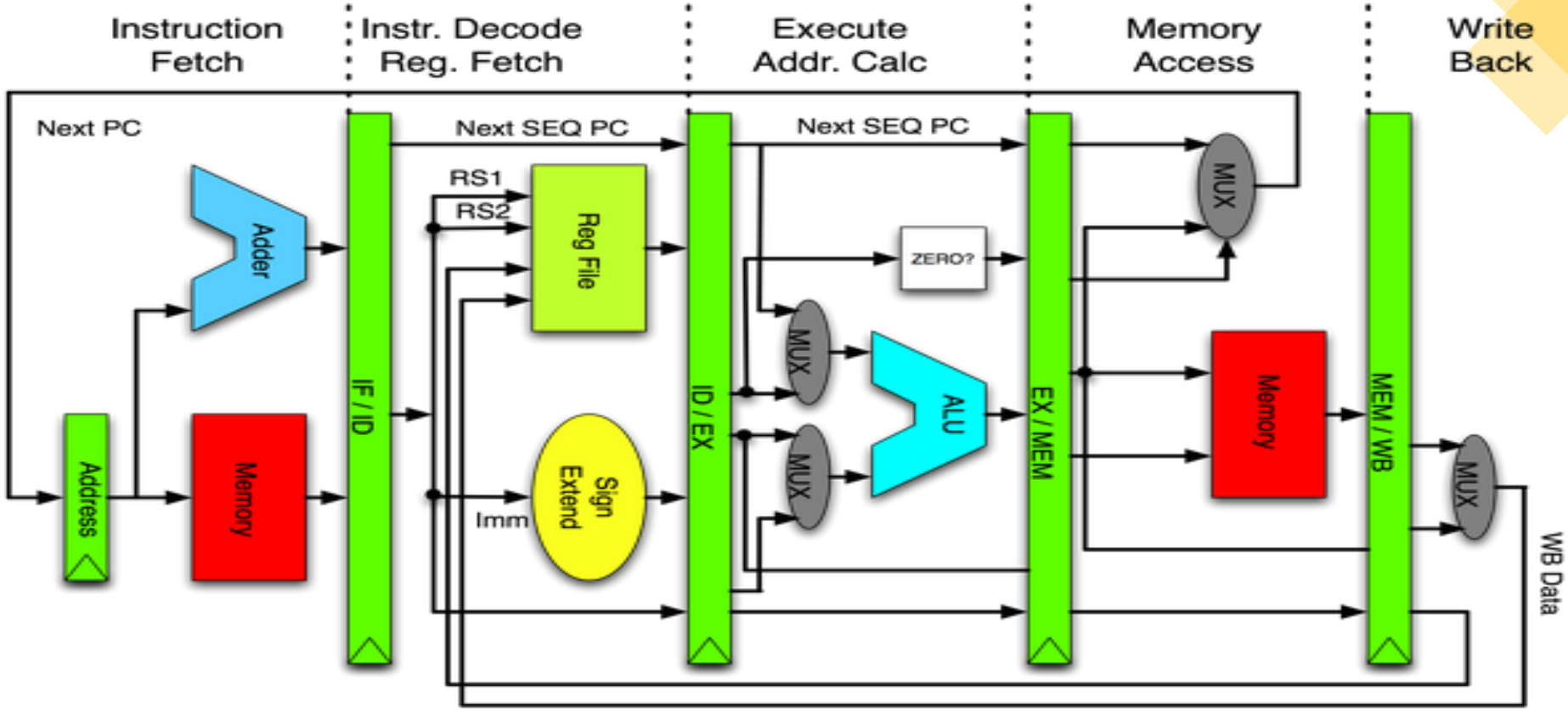


Image from Wikimedia

3: Issues of pipeline and Design Solutions

Cycle 2: I1 in Decode stage, I2 in fetch stage

I_1	Move R5, #2000h
I_2	Move R1, #15
I_3	Move R2, #16
I_4	Move R3, R1
I_5	Add R1, R3
I_6	Move R4, R2
I_7	Add R2, R4
I_8	Add R1, R2
I_9	Store [R5], R1

I2: Move R1, #15

I1: Move R5, #2000h

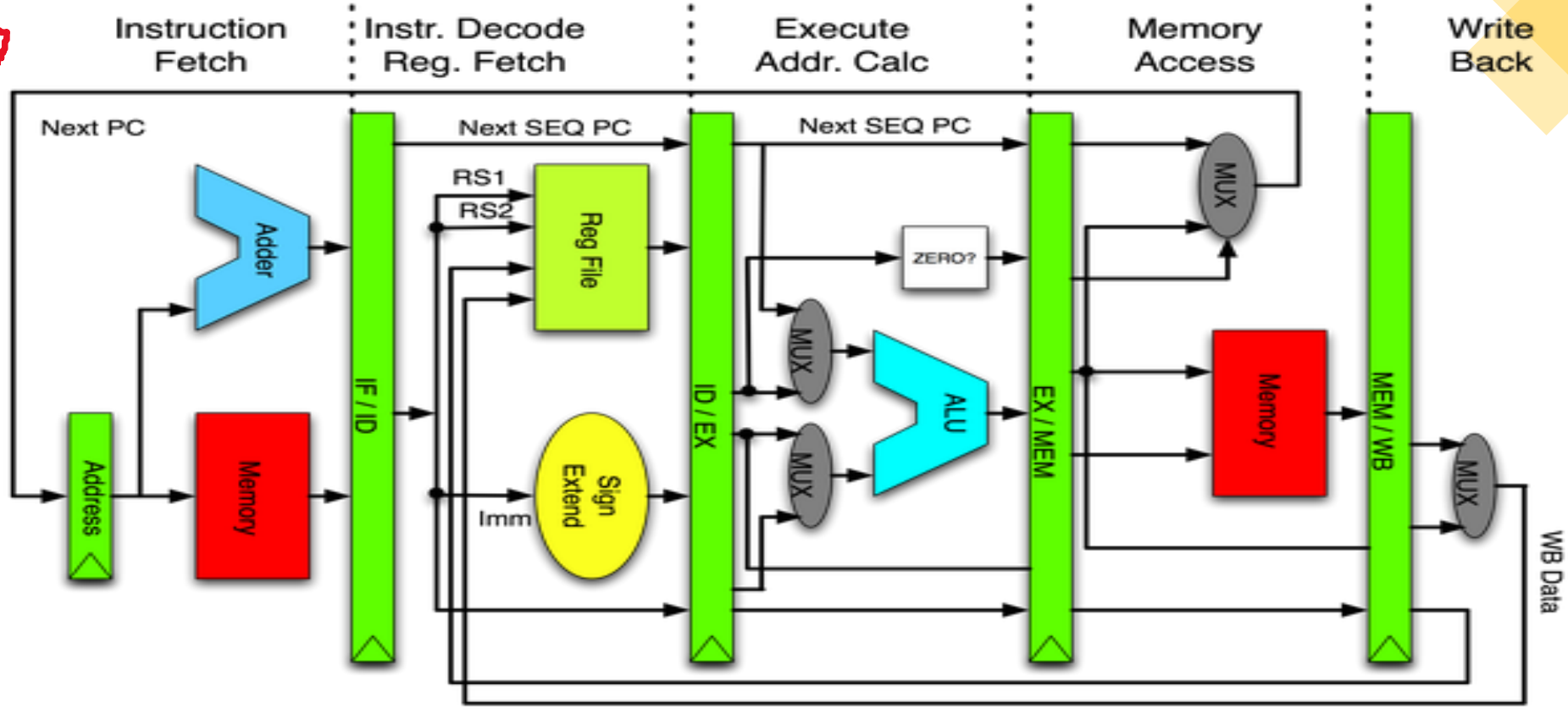


Image from Wikimedia

3: Issues of pipeline and Design Solutions

Cycle 3: I1 in Execute stage, I2 in Decode stage, I3 in fetch stage

I_1	Move R5, #2000h
I_2	Move R1, #15
I_3	Move R2, #16
I_4	Move R3, R1
I_5	Add R1, R3
I_6	Move R4, R2
I_7	Add R2, R4
I_8	Add R1, R2
I_9	Store [R5], R1

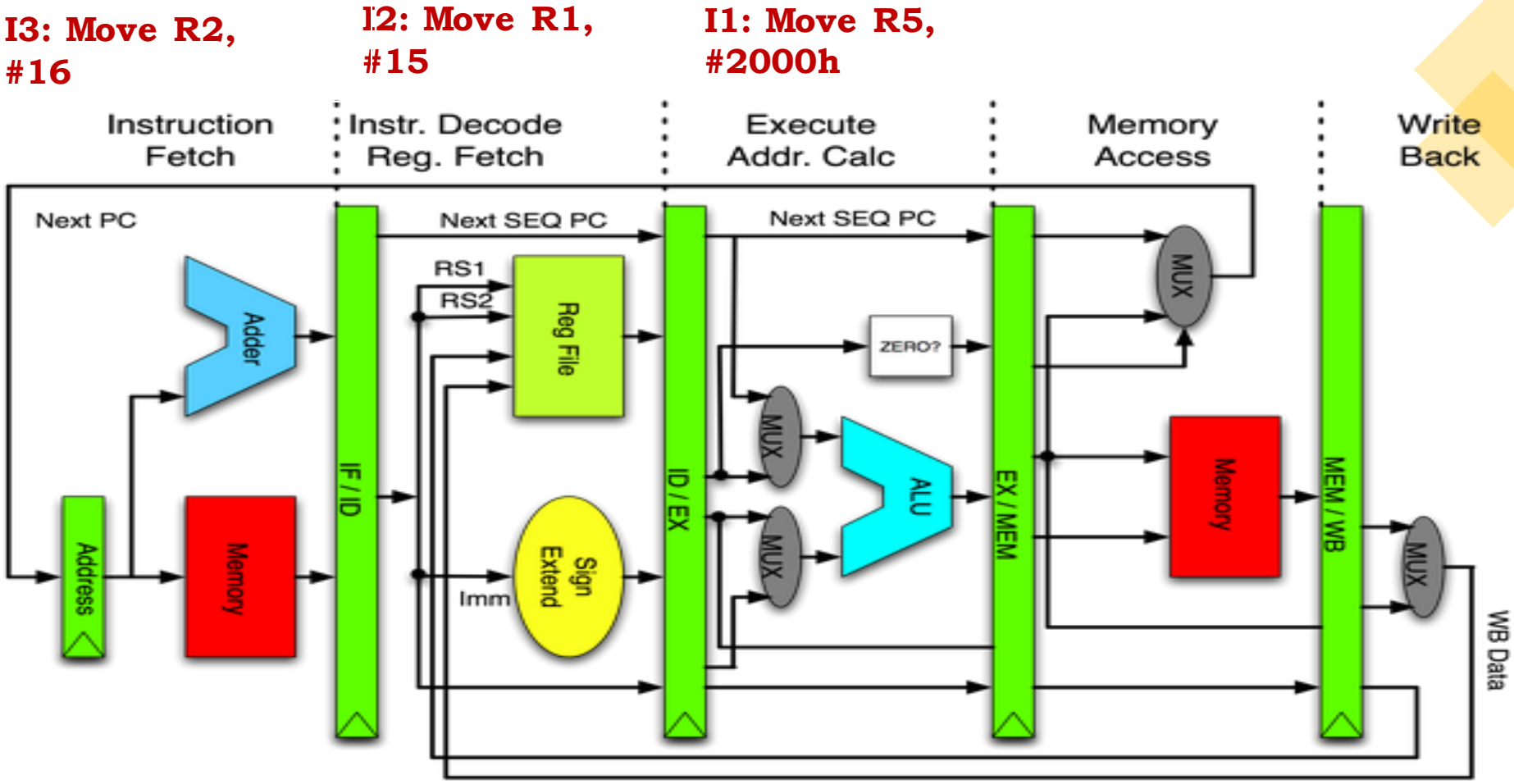


Image from Wikimedia

3: Issues of pipeline and Design Solutions

Cycle 4: I1 in memory access stage, I2 in execute , I3 in Decode and I4 in fetch stage

I_1	Move R5, #2000h
I_2	Move R1, #15
I_3	Move R2, #16
I_4	Move R3, R1
I_5	Add R1, R3
I_6	Move R4, R2
I_7	Add R2, R4
I_8	Add R1, R2
I_9	Store [R5], R1

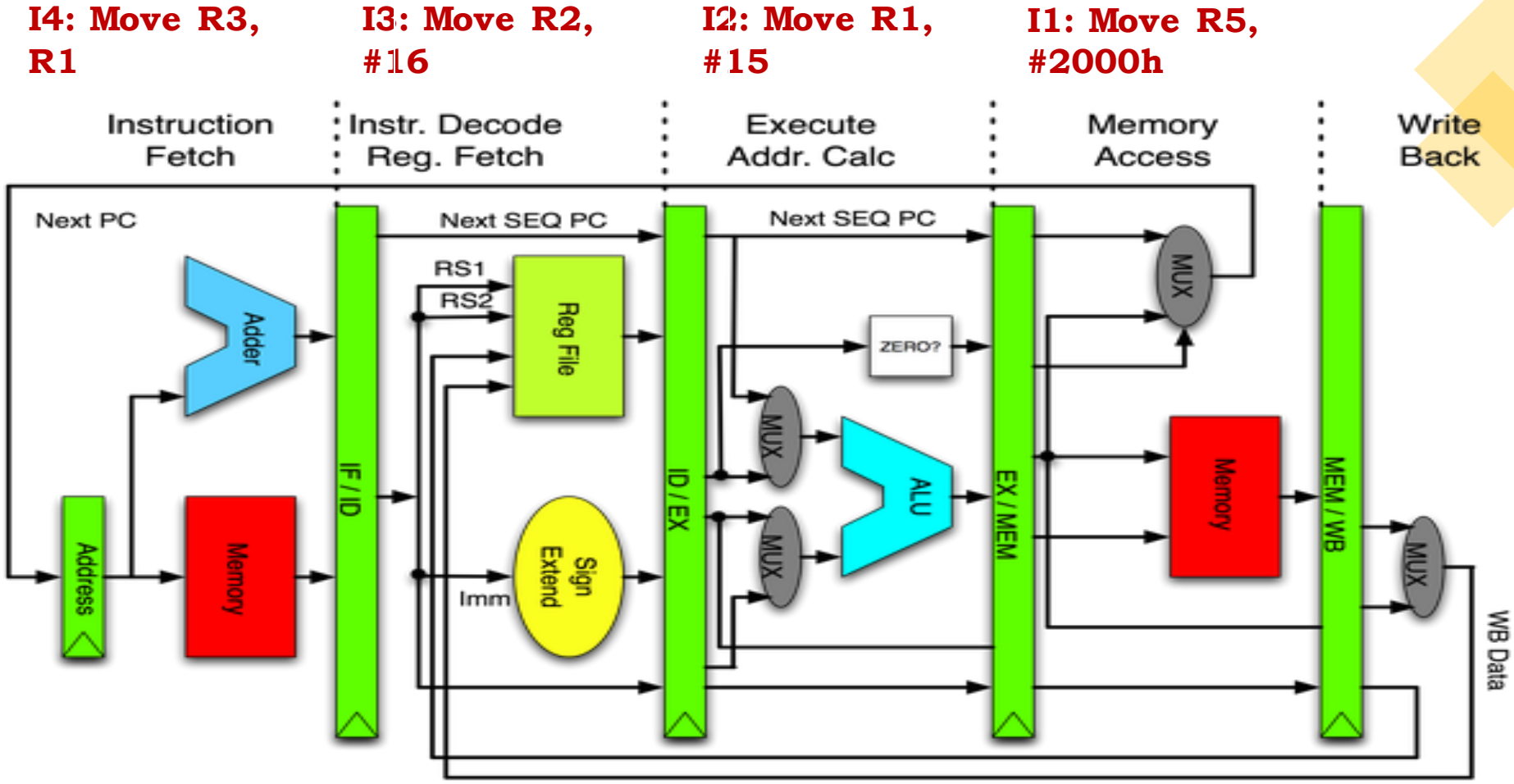


Image from Wikimedia

3: Issues of pipeline and Design Solutions

Cycle 5: R1 data must be Written from I1 to Register file ; R1 Data must be read by Instruction 4.

I_1	Move R5, #2000h
I_2	Move R1, #15
I_3	Move R2, #16
I_4	Move R3, R1
I_5	Add R1, R3
I_6	Move R4, R2
I_7	Add R2, R4
I_8	Add R1, R2
I_9	Store [R5], R1

I5: Add R1, R3 | **I4: Move R3, R1** | **I3: Move R2, #16** | **I2: Move R1, #15** | **I5: Move R5, #2000h**

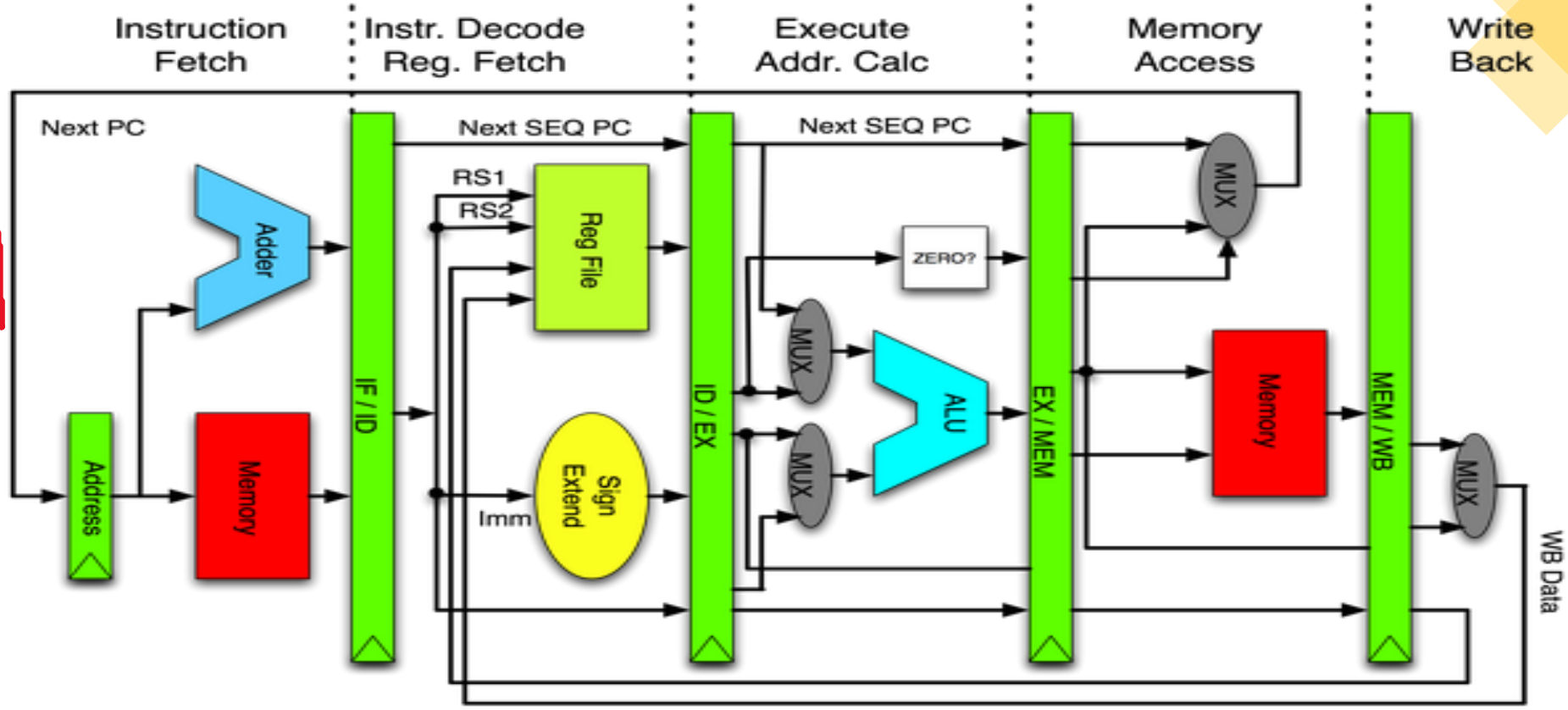


Image from Wikimedia

3: Issues of pipeline and Design Solutions

Cycle 5: R1 data must be Written from I1 to Register file ; R1 Data must be read by Instruction 4.

Data Dependency Issue at Decode Stage: Solve by Read register content after write operation.

I_1	Move R5, #2000h
I_2	Move R1, #15
I_3	Move R2, #16
I_4	Move R3, R1
I_5	Add R1, R3
I_6	Move R4, R2
I_7	Add R2, R4
I_8	Add R1, R2
I_9	Store [R5], R1

I5: Add R1, R3 | **I4: Move R3, R1** | **I3: Move R2, #16** | **I2: Move R1, #15** | **I1: Move R5, #2000h**

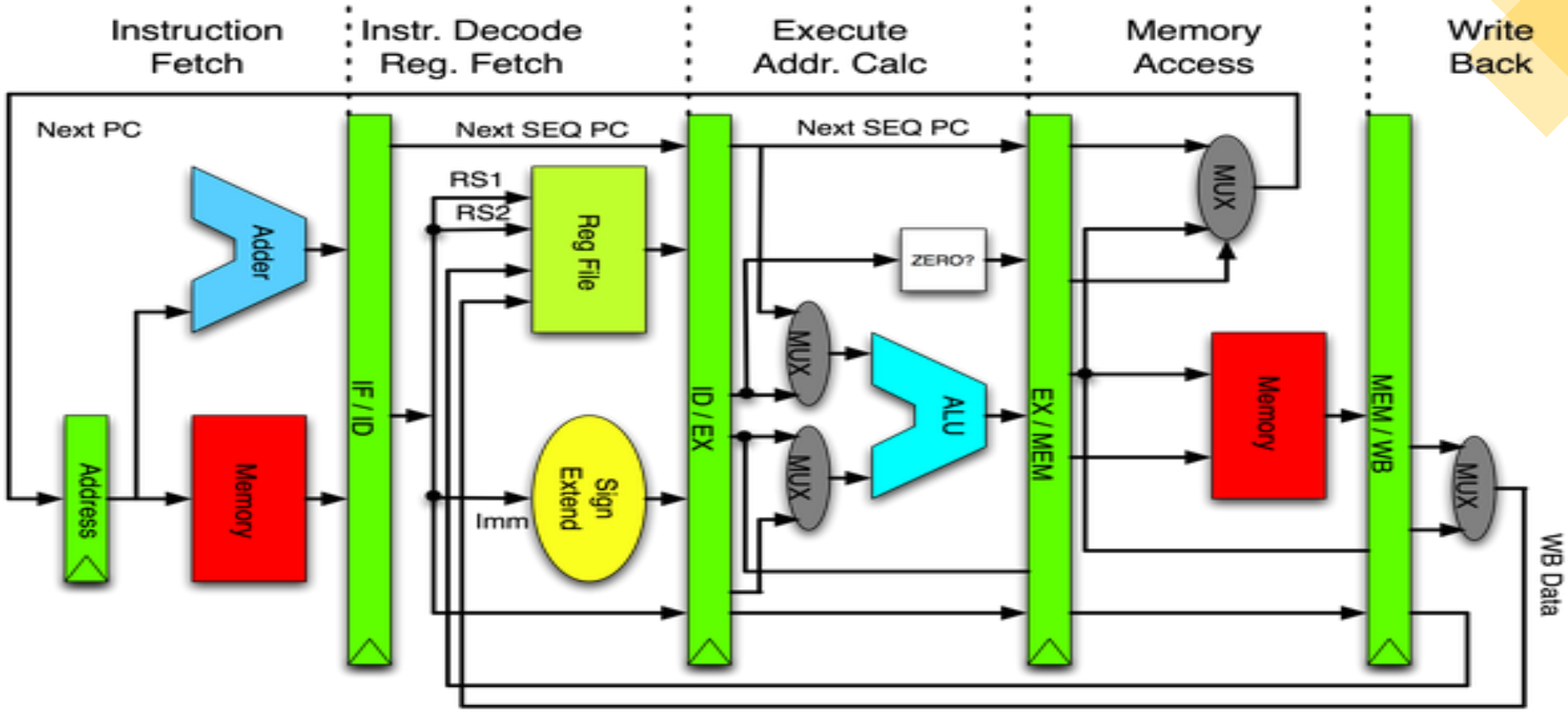


Image from Wikimedia

3: Issues of pipeline and Design Solutions

Cycle 6: R3 data must be Written from I4 to Register file ; R3 Data must be read by Instruction 5.

Data Dependency Issue at Decode Stage: Since Data is not ready, R3 old value is read

I_1	Move R5, #2000h
I_2	Move R1, #15
I_3	Move R2, #16
I_4	Move R3, R1
I_5	Add R1, R3
I_6	Move R4, R2
I_7	Add R2, R4
I_8	Add R1, R2
I_9	Store [R5], R1

I6:Move R4, R2 | **I5: Add R1, R3** | **I4: Move R3, R1** | **I3: Move R2, #16** | **I2: Move R1, #15**

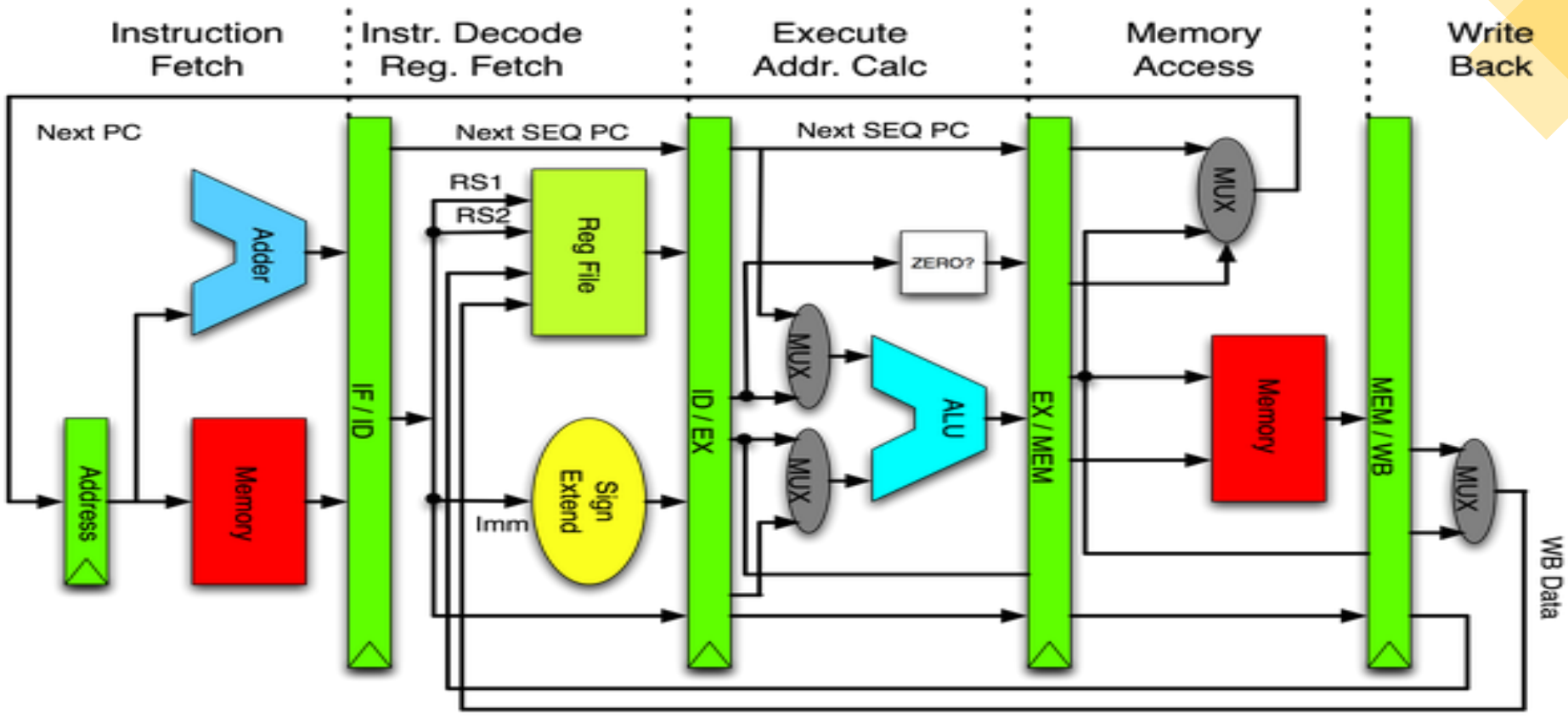


Image from Wikimedia

3: Issues of pipeline and Design Solutions

Cycle 7: R3 data not ready due to instruction dependency

Data Dependency Issue at Execution stage : Stall the pipeline

I_1	Move R5, #2000h
I_2	Move R1, #15
I_3	Move R2, #16
I_4	Move R3, R1
I_5	Add R1, R3
I_6	Move R4, R2
I_7	Add R2, R4
I_8	Add R1, R2
I_9	Store [R5], R1

I7: Add R2, R4 | **I6:Move R4, R2** | **I5: Add R1, R3** | **I4: Move R3, R1** | **I3: Move R2, #16**

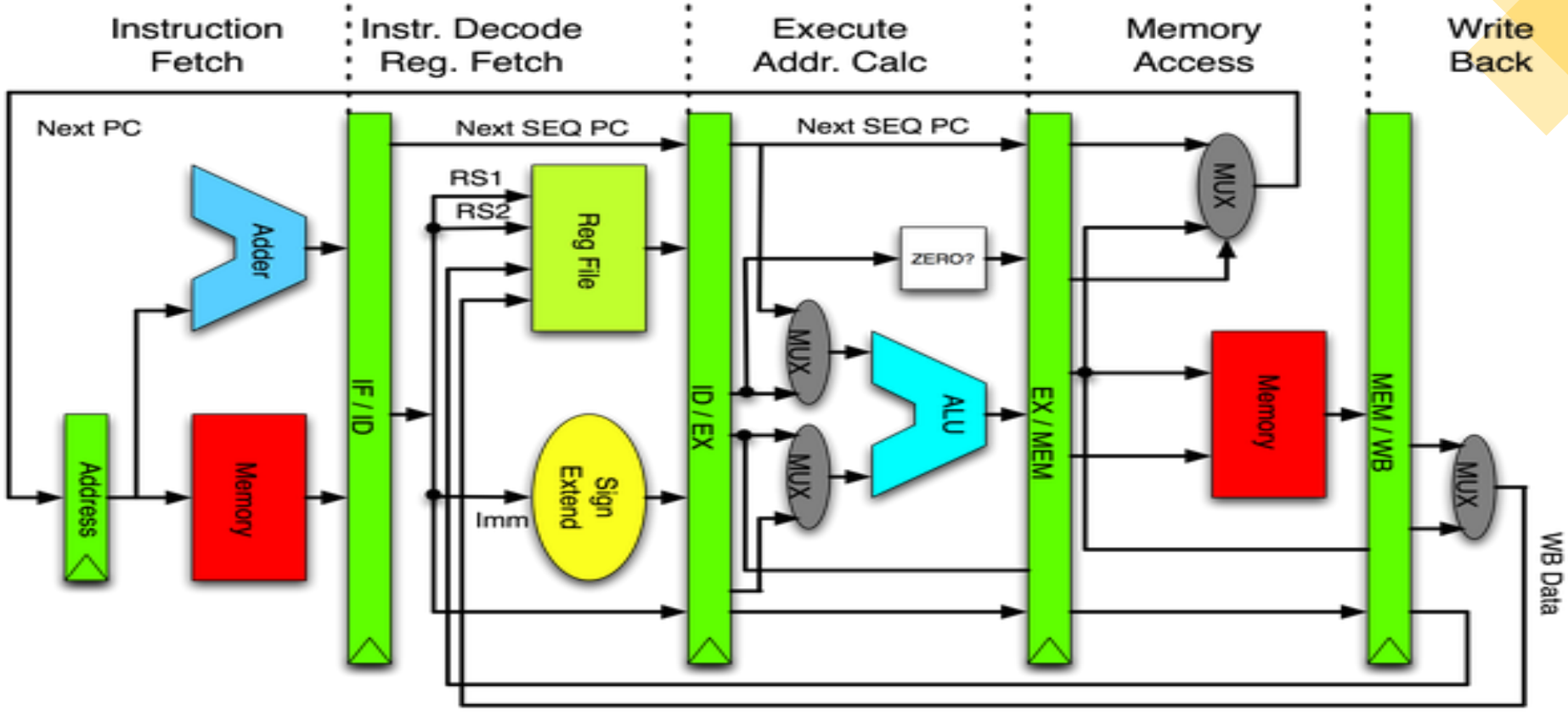


Image from Wikimedia

3: Issues of pipeline and Design Solutions

Cycle 8: Pipeline stall: no new fetch instruction. Only write back happens

Data Dependency Issue at Execution stage : Stall the pipeline

I_1	Move R5, #2000h
I_2	Move R1, #15
I_3	Move R2, #16
I_4	Move R3, R1
I_5	Add R1, R3
I_6	Move R4, R2
I_7	Add R2, R4
I_8	Add R1, R2
I_9	Store [R5], R1

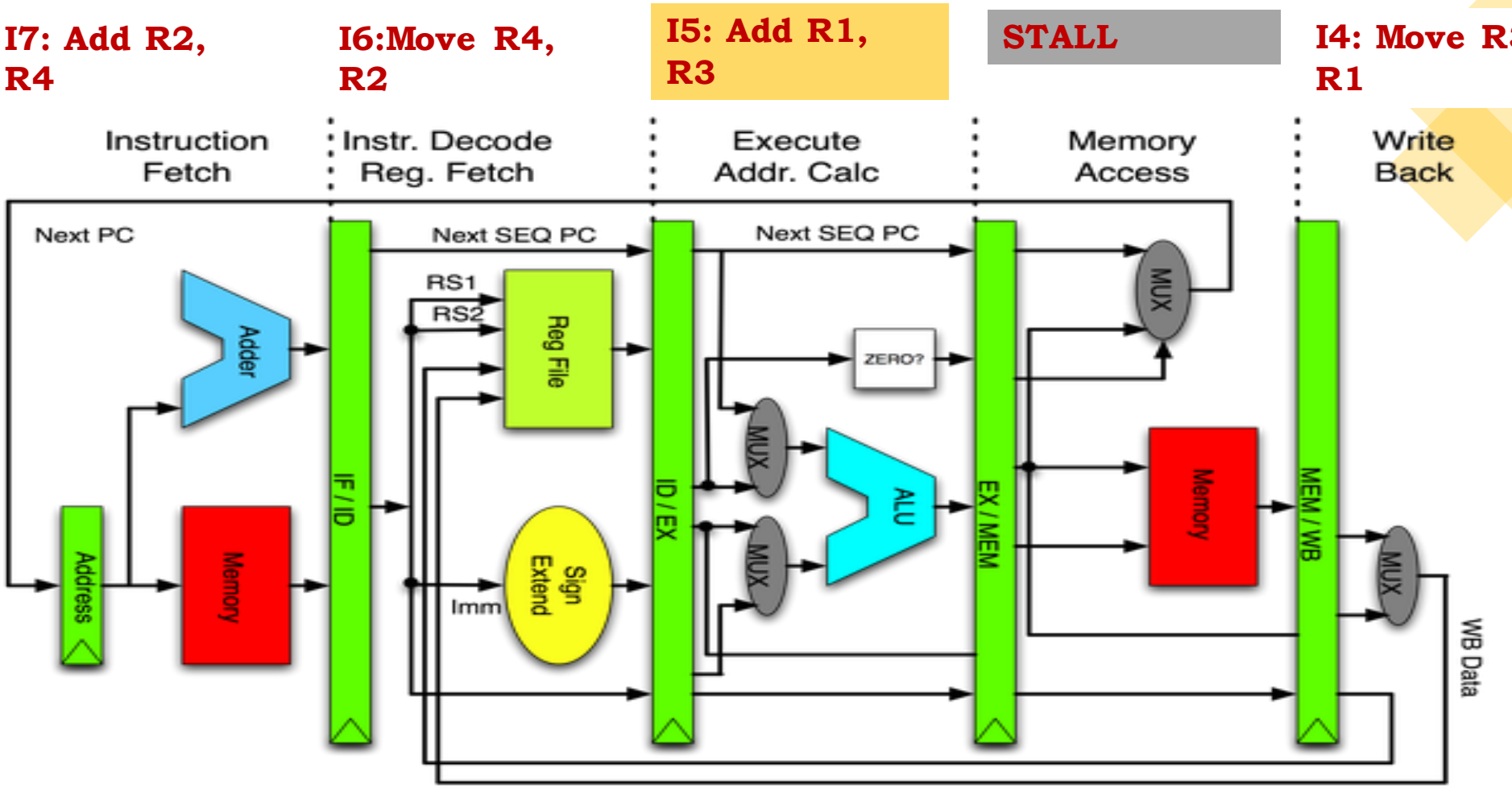


Image from Wikimedia

3: Issues of pipeline and Design Solutions

Cycle 8: We need data of R3 at execution stage , Bu I4 is performing write back to register now.
Use the concept of Data Forwarding

I_1	Move R5, #2000h
I_2	Move R1, #15
I_3	Move R2, #16
I_4	Move R3, R1
I_5	Add R1, R3
I_6	Move R4, R2
I_7	Add R2, R4
I_8	Add R1, R2
I_9	Store [R5], R1

I7: Add R2, R4 | **I6:Move R4, R2** | **I5: Add R1, R3** | **STALL** | **I4: Move R3, R1**

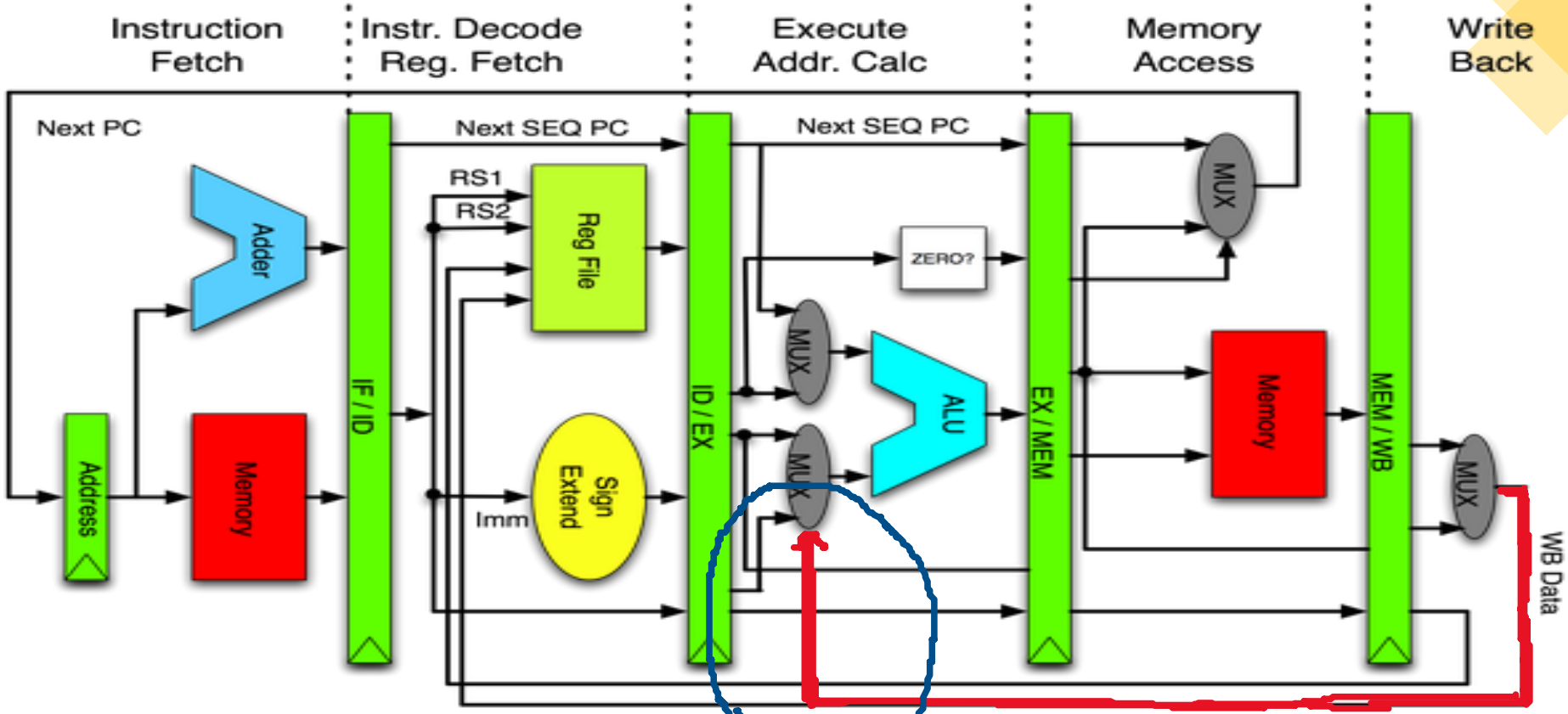


Image from Wikimedia

3: Issues of pipeline and Design Solutions

Cycle 9: Nothing in Writeback stage

I_1	Move R5, #2000h
I_2	Move R1, #15
I_3	Move R2, #16
I_4	Move R3, R1
I_5	Add R1, R3
I_6	Move R4, R2
I_7	Add R2, R4
I_8	Add R1, R2
I_9	Store [R5], R1

I8: Add R1, R2

I7: Add R2, R4

I6:Move R4, R2

I5: Add R1, R3

STALL

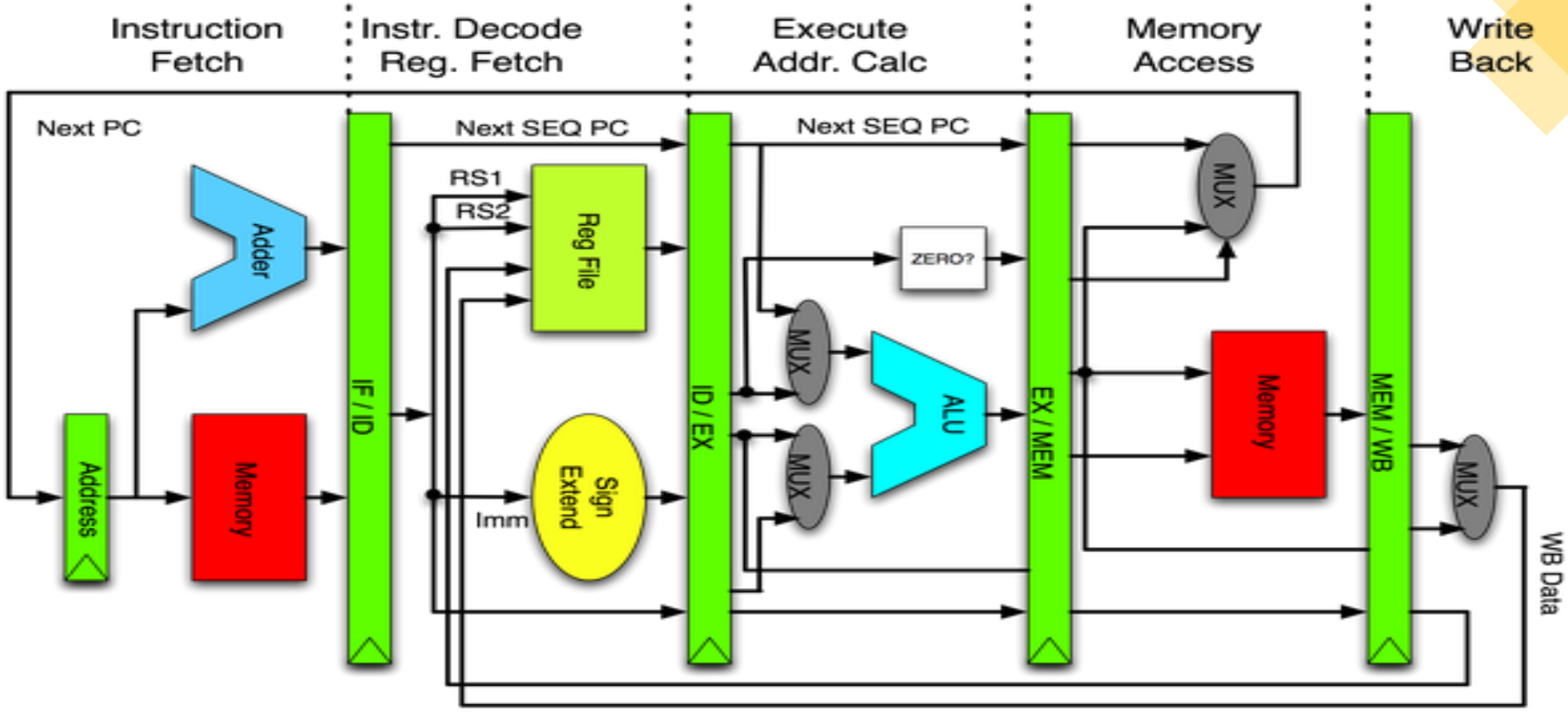


Image from Wikimedia

3: Issues of pipeline and Design Solutions

Cycle 10: Data dependency at Execution stage; Wait

I_1	Move R5, #2000h
I_2	Move R1, #15
I_3	Move R2, #16
I_4	Move R3, R1
I_5	Add R1, R3
I_6	Move R4, R2
I_7	Add R2, R4
I_8	Add R1, R2
I_9	Store [R5], R1

I9: Store [R5],R1 **I8: Add R1, R2** **I7: Add R2, R4** **I6:Move R4, R2** **I5: Add R1, R3**

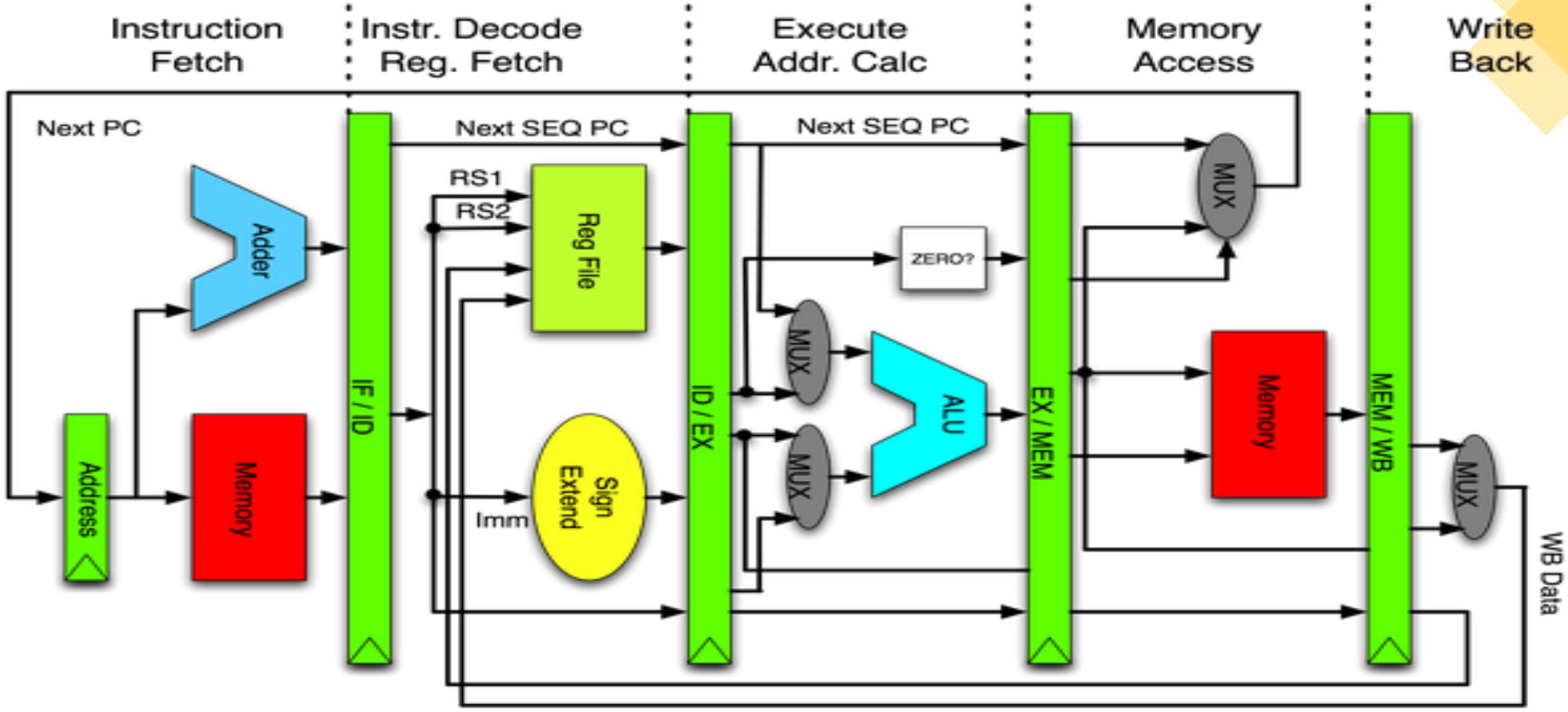


Image from Wikimedia

3: Issues of pipeline and Design Solutions

Cycle 11: Data dependency at Execution stage; Data forwarding from writeback stage

I_1	Move R5, #2000h
I_2	Move R1, #15
I_3	Move R2, #16
I_4	Move R3, R1
I_5	Add R1, R3
I_6	Move R4, R2
I_7	Add R2, R4
I_8	Add R1, R2
I_9	Store [R5], R1

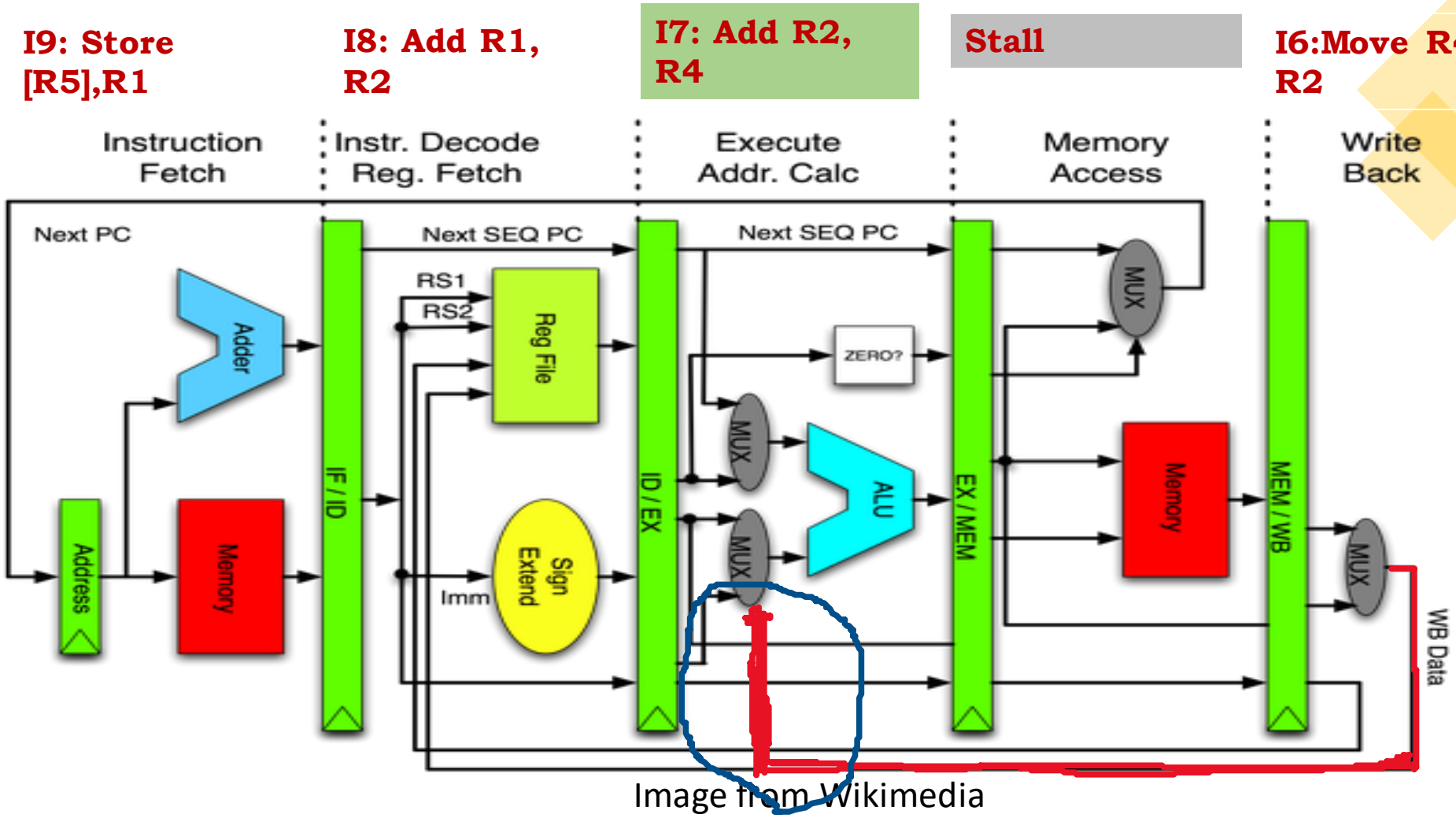


Image from Wikimedia

3: Issues of pipeline and Design Solutions

Cycle 12: Data dependency at Execution stage; Wait

I_1	Move R5, #2000h
I_2	Move R1, #15
I_3	Move R2, #16
I_4	Move R3, R1
I_5	Add R1, R3
I_6	Move R4, R2
I_7	Add R2, R4
I_8	Add R1, R2
I_9	Store [R5], R1

I9: Store [R5], R1. | **I8: Add R1, R2** | **I7: Add R2, R4** | **Stall**

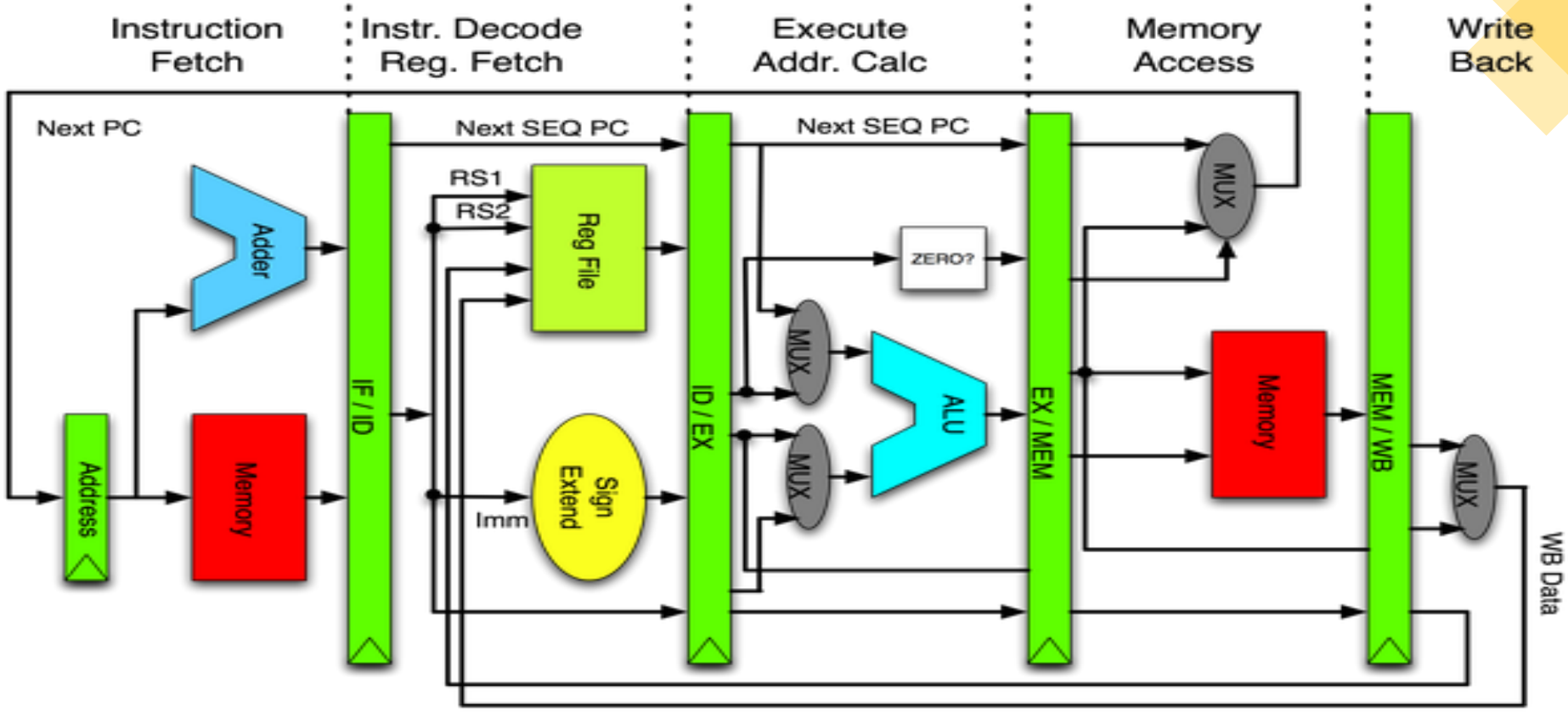


Image from Wikimedia

3: Issues of pipeline and Design Solutions

Cycle 13: Data dependency at Execution stage; Data Forwarding

I_1	Move R5, #2000h
I_2	Move R1, #15
I_3	Move R2, #16
I_4	Move R3, R1
I_5	Add R1, R3
I_6	Move R4, R2
I_7	Add R2, R4
I_8	Add R1, R2
I_9	Store [R5], R1

I9: Store [R5], R1. | **I8: Add R1, R2** | **STALL** | **I7: Add R2, R4**

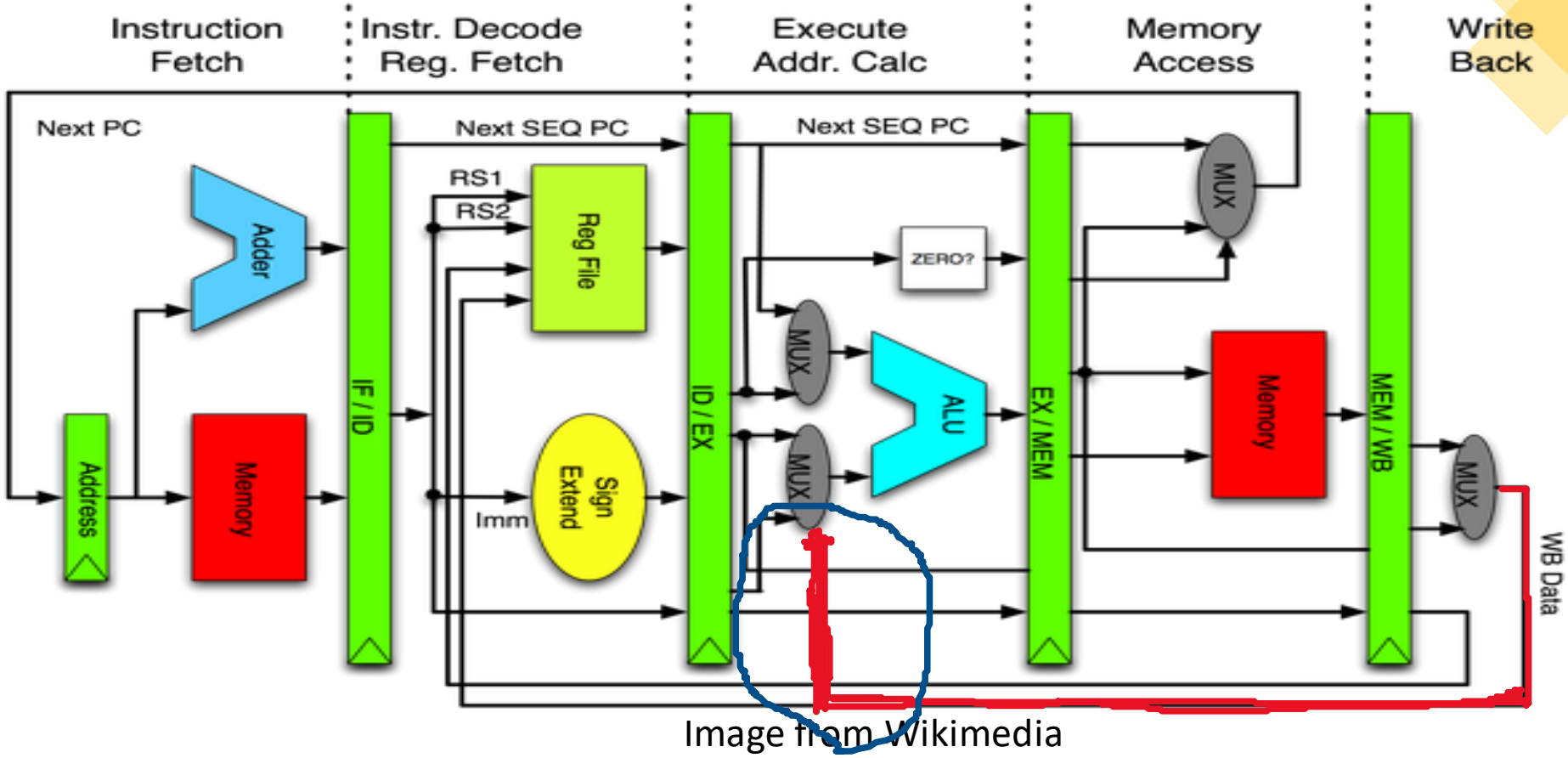


Image from Wikimedia

3: Issues of pipeline and Design Solutions

Cycle 14: Data dependency at Execution stage; Stall Pipeline

I_1	Move R5, #2000h
I_2	Move R1, #15
I_3	Move R2, #16
I_4	Move R3, R1
I_5	Add R1, R3
I_6	Move R4, R2
I_7	Add R2, R4
I_8	Add R1, R2
I_9	Store [R5], R1

I9: Store [R5], R1.

I8: Add R1, R2

STALL

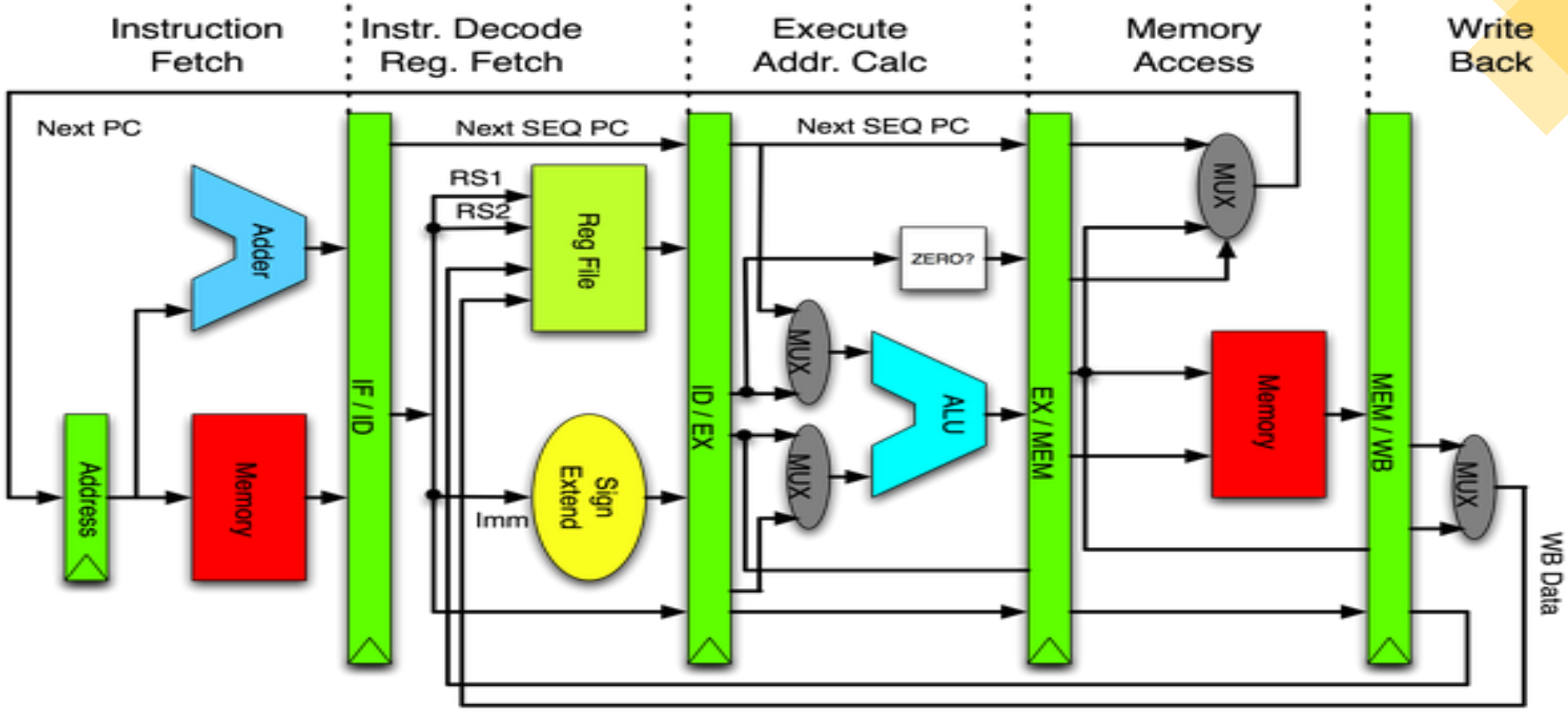


Image from Wikimedia

3: Issues of pipeline and Design Solutions

Cycle 15: Data Dependency at Memory Write Stage : Use Data Forwarding ·

I_1	Move R5, #2000h
I_2	Move R1, #15
I_3	Move R2, #16
I_4	Move R3, R1
I_5	Add R1, R3
I_6	Move R4, R2
I_7	Add R2, R4
I_8	Add R1, R2
I_9	Store [R5], R1

I9: Store [R5], R1.

I8: Add R1, R2

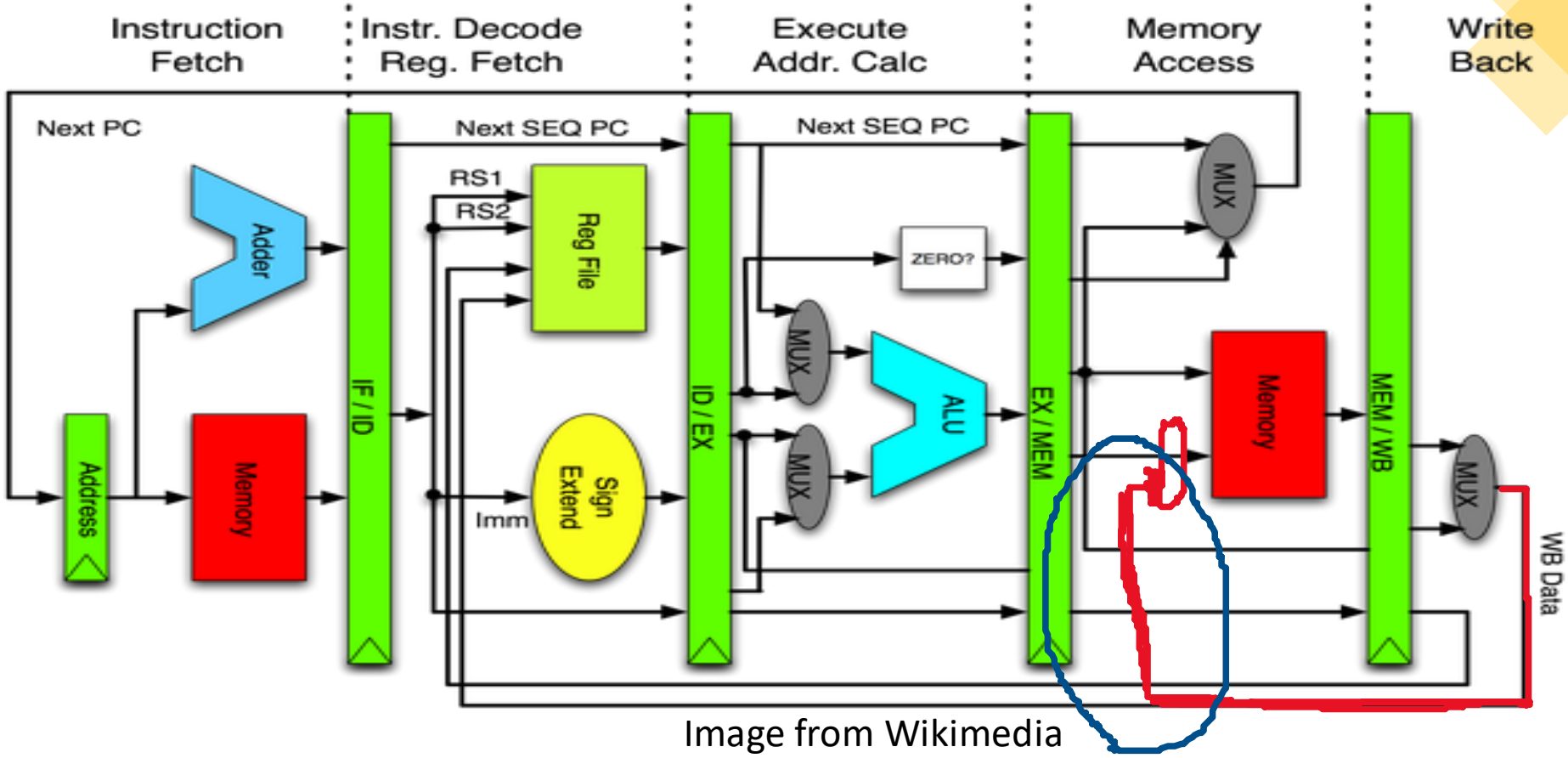


Image from Wikimedia

3: Issues of pipeline and Design Solutions

Cycle 16: Execution of Program Complete

Total Number of cycles taken = 16

I9: Store [R5], R1.

I_1	Move R5, #2000h
I_2	Move R1, #15
I_3	Move R2, #16
I_4	Move R3, R1
I_5	Add R1, R3
I_6	Move R4, R2
I_7	Add R2, R4
I_8	Add R1, R2
I_9	Store [R5], R1

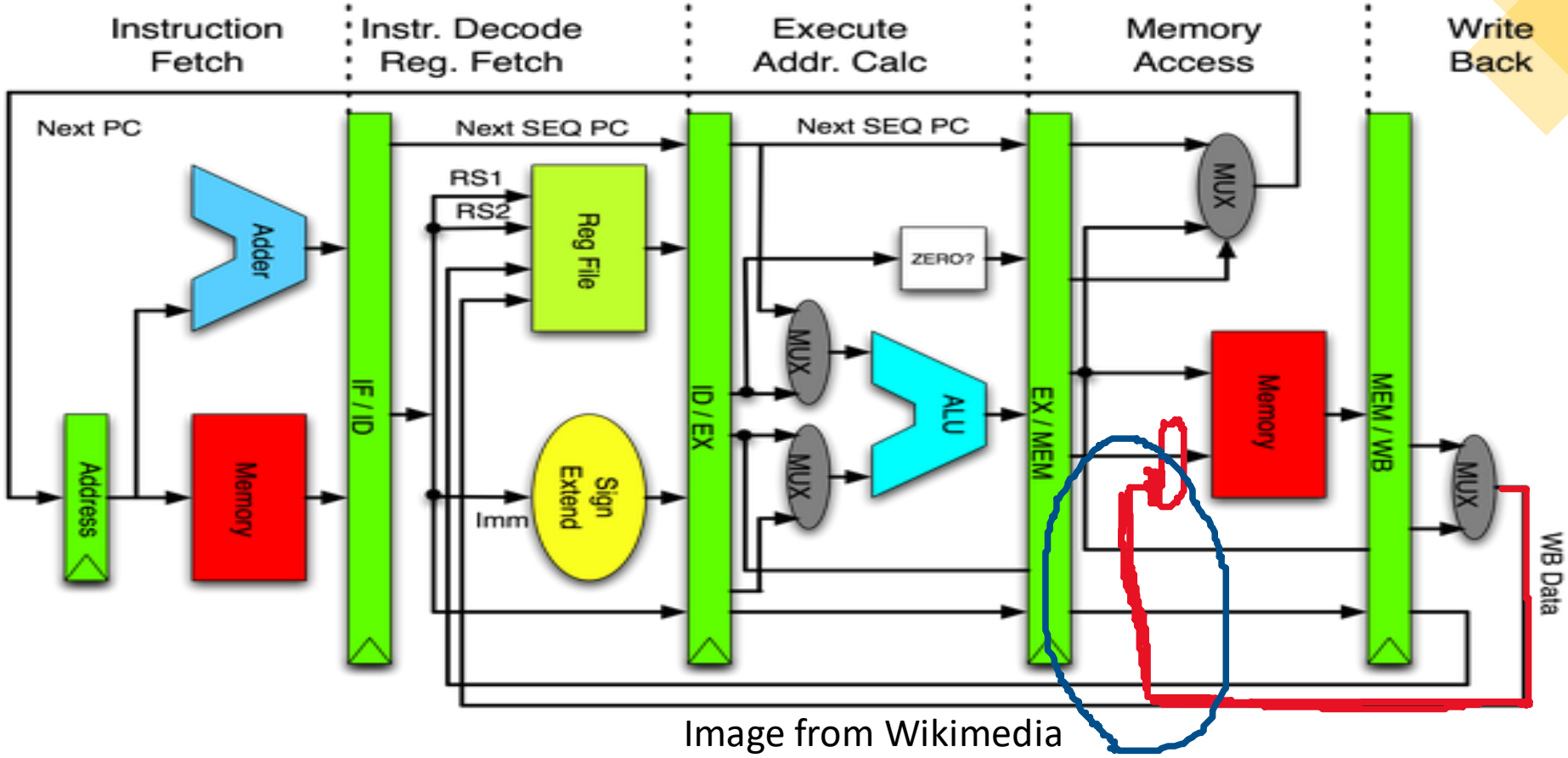


Image from Wikimedia

2. Five Stage Pipeline Design

	Instruction	(F)	(D)	(E)	(M)	(W)	
I_1	Move R5, #2000h	I_1					
I_2	Move R1, #15	I_2	I_1				
I_3	Move R2, #16	I_3	I_2	I_1			
I_4	Move R3, R1	I_4	I_3	I_2	I_1		
I_5	Add R1, R3	I_5	I_4	I_3	I_2	I_1	I1: Complete
I_6	Move R4, R2	I_6	I_5	I_4	I_3	I_2	I2: Complete
I_7	Add R2, R4	I_7	I_6	I_5	I_4	I_3	I3: Complete
I_8	Add R1, R2	I_8	I_7	I_6	I_5	I_4	I4: Complete
I_9	Store [R5], R1	I_9	I_8	I_7	I_6	I_5	I5: Complete
I_{10}			I_9	I_8	I_7	I_6	I6: Complete
I_{11}				I_9	I_8	I_7	I7: Complete
I_{12}					I_9	I_8	I8: Complete
I_{13}						I_9	I9: Complete

- If no pipeline, then this program takes

  9 x 5 = 45 cycles

- If 5 stage pipeline is used, then it takes

  5 + 8 = 13 cycles

(first instruction takes 'n' stages/cycle) + (n-1) instructions

- Actual Cycles taken due to **data dependency = 16 Cycles.**

Reference

Textbooks and/or Reference Books:

1. Professional CUDA C Programming – John Cheng, Max Grossman, Ty McKercher, 2014
2. Wilkinson, M. Allen, ”Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers”, Pearson Education, 1999
3. I. Foster, ”Designing and building parallel programs”, 2003
4. Parallel Programming in C using OpenMP and MPI – Micheal J Quinn, 2004
5. Introduction to Parallel Programming – Peter S Pacheco, Morgan Kaufmann Publishers, 2011
6. Advanced Computer Architectures: A design approach, Dezso Sima, Terence Fountain, Peter Kacsuk, 2002
7. Parallel Computer Architecture : A hardware/Software Approach, David E Culler, Jaswinder Pal Singh Anoop Gupta, 2011 8. Introduction to Parallel Computing, Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar, Pearson, 2011

Thank You

National Institute of Technology Karnataka Surathkal
Department of Information Technology

IT 301 Parallel Computing

Memory Models, Flynn's Classification

Dr. Geetha V

Assistant Professor

Dept of Information Technology

NITK Surathkal

Course Outline

Course Plan: Theory:

Part A: Parallel Computer Architectures

Week 1,2,3: ***Introduction to Parallel Computer Architecture:*** Parallel Computing, Parallel architecture, bit level, instruction level , data level and task level parallelism. Instruction level parelllelism: pipelining(Data and control instructions), scalar and superscalar processors, vector processors. Parallel computers and computation.

Week 4,5: Memory Models: UMA, NUMA and COMA. Flynns classification, Cache coherence,

Week 6,7: Amdahl's Law. Performance evaluation, Designing parallel algorithms : Divide and conquer, Load balancing, Pipelining.

Week 8 -11: ***Parallel Programming techniques like Task Parallelism using TBB, TL2, Cilk++ etc. and software transactional memory techniques.***

Course Outline

Part B: OpenMP/MPI/CUDA

Week 1,2,3 : ***Shared Memory Programing Techniques:*** Introduction to OpenMP : Directives: parallel, for, sections, task, single, critical, barrier, taskwait, atomic. Clauses: private, shared, firstprivate, lastprivate, reduction, nowait, ordered, schedule, collapse, num_threads, shared, if().

Week 4,5: ***Distributed Memory programming Techniques:*** MPI: Blocking, Non-blocking.

Week 6,7 : CUDA : OpenCL, Execution models, GPU memory, GPU libraries.

Week 10,11,: ***Introduction to accelerator programming using CUDA/OpenCL and Xeon-phi. Concepts of Heterogeneous programming techniques.***

Practical:

Implementation of parallel programs using OpenMP/MPI/CUDA.

Assignment: Performance evaluation of parallel algorithms (in group of 2 or 3 members)

Index

1. Flynn's Classification

- In 1966, M.J. Flynn proposed a classification for the organization of a computer system by the number of instructions and data items that are manipulated simultaneously.
- Flynn uses the stream concept for describing a machine's structure.
- The sequence of instructions read from memory constitutes an **instruction stream**.
- The operations performed on the data in the processor constitute a **data stream**.

Flynn's Classification

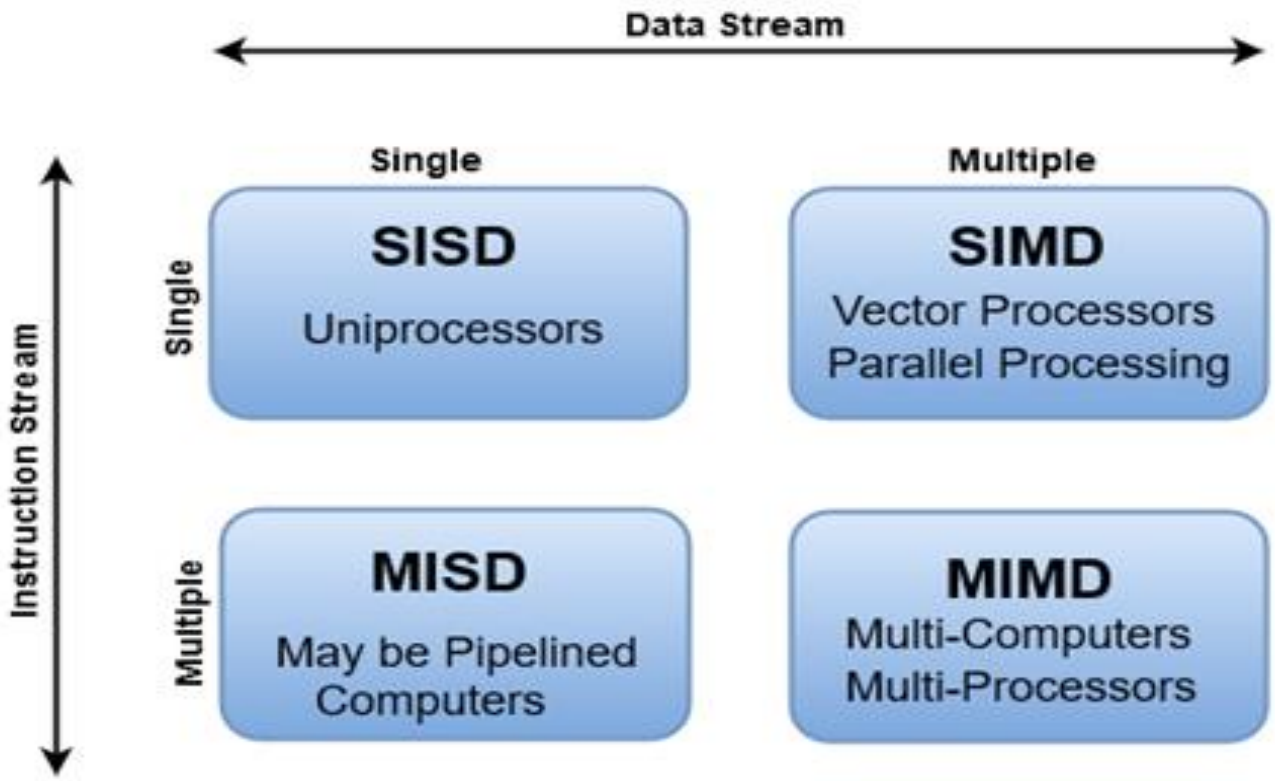
The classification of computer architectures based on the number of instruction streams and data streams (Flynn's Taxonomy)

Flynn's Classification divides computers into four major groups .

- Single instruction stream, single data stream (SISD)
- Single instruction stream, multiple data stream (SIMD)
- Multiple instruction stream, single data stream (MISD)
- Multiple instruction stream, multiple data stream (MIMD)

Flynn's Classification

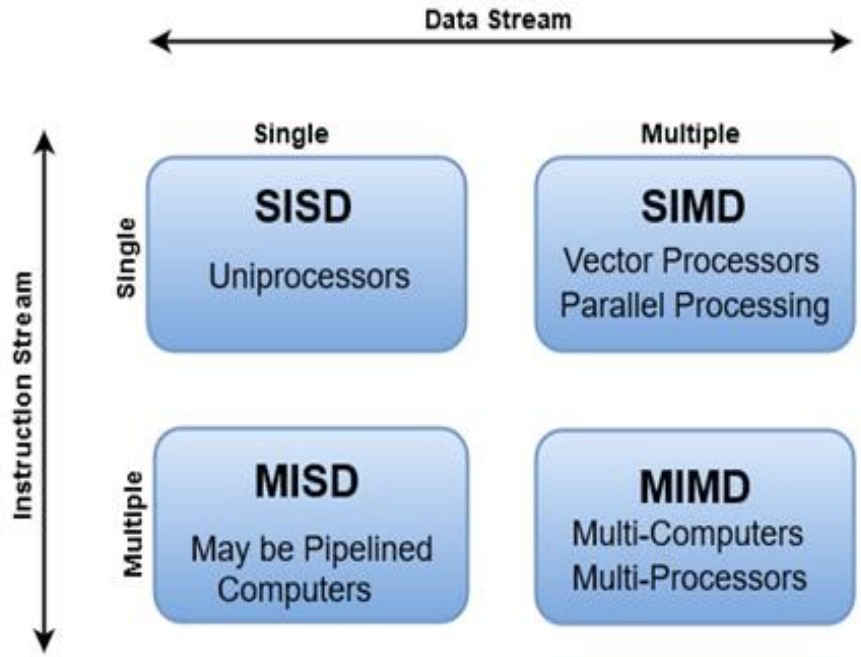
Flynn's Classification divides computers into four major groups .



Ref:https://www.javatpoint.com/flynns-classification-of-computers

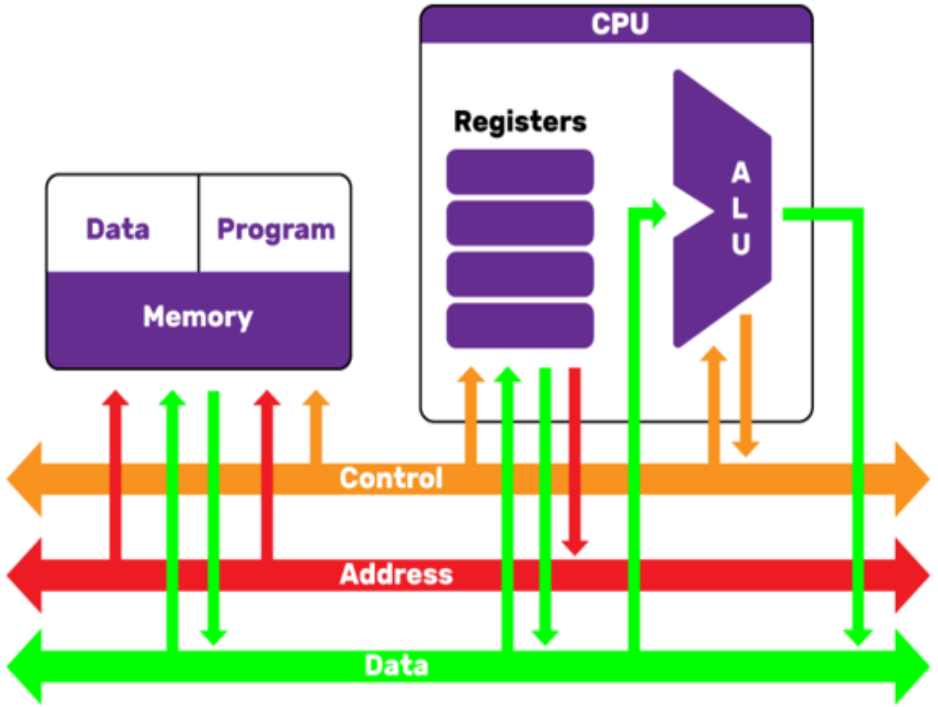
Flynn's Classification: SISD

Flynn's Classification



Ref:https://www.javatpoint.com/flynns-classification-of-computers

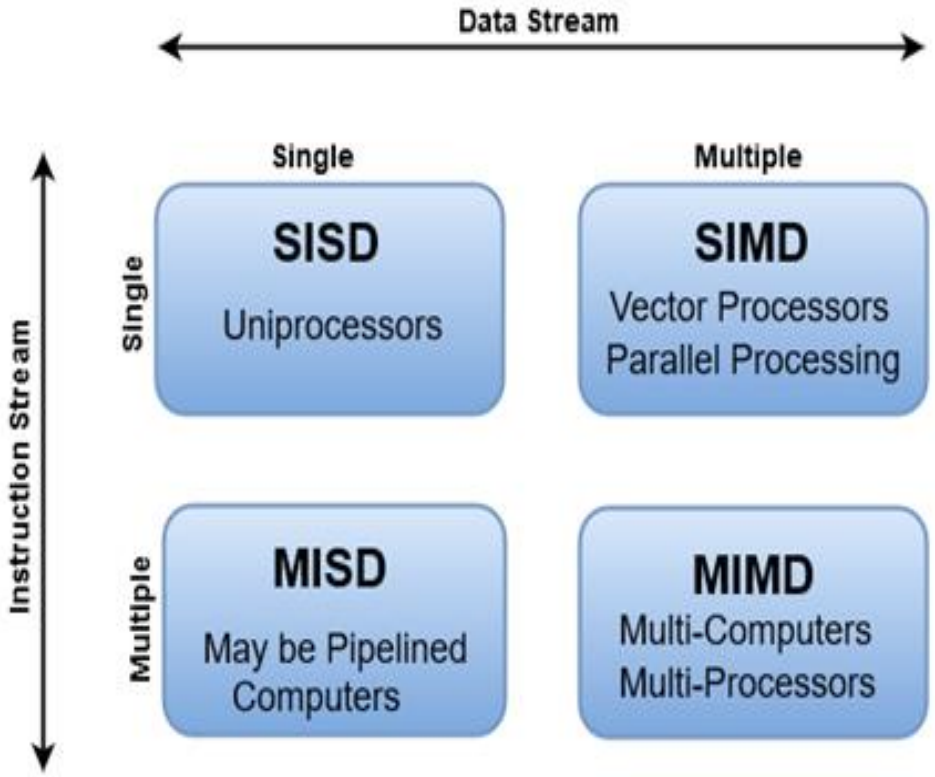
Example: Traditional Computers



Ref:https://www.futurelearn.com/courses/how-computers-work/0/steps/49283

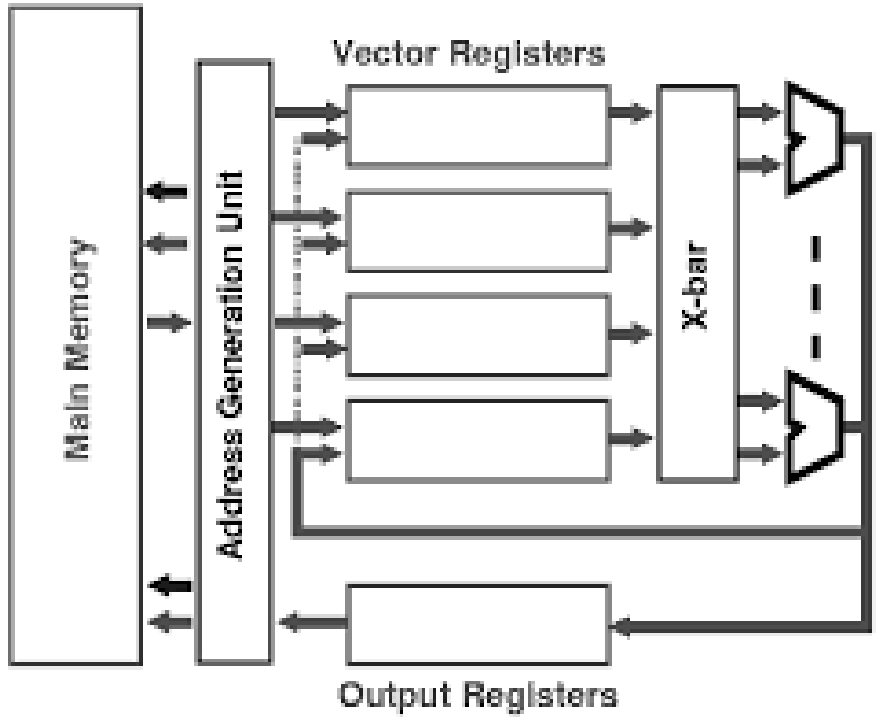
Flynn's Classification: SIMD

Flynn's Classification



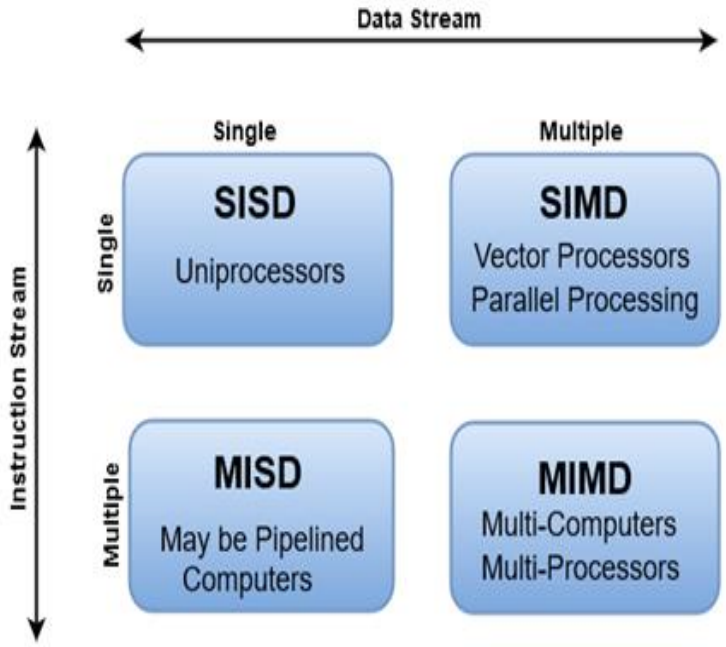
Ref:https://www.javatpoint.com/flynns-classification-of-computers

Example: Vector processor CRAY -I



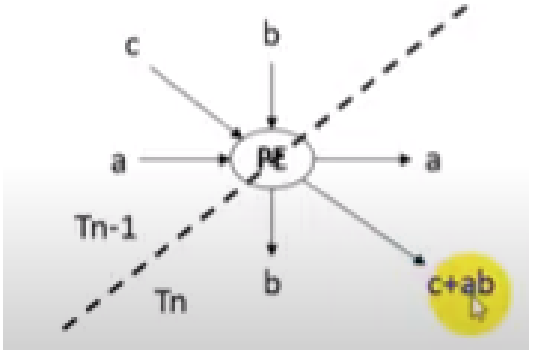
Flynn's Classification: MISD

Flynn's Classification

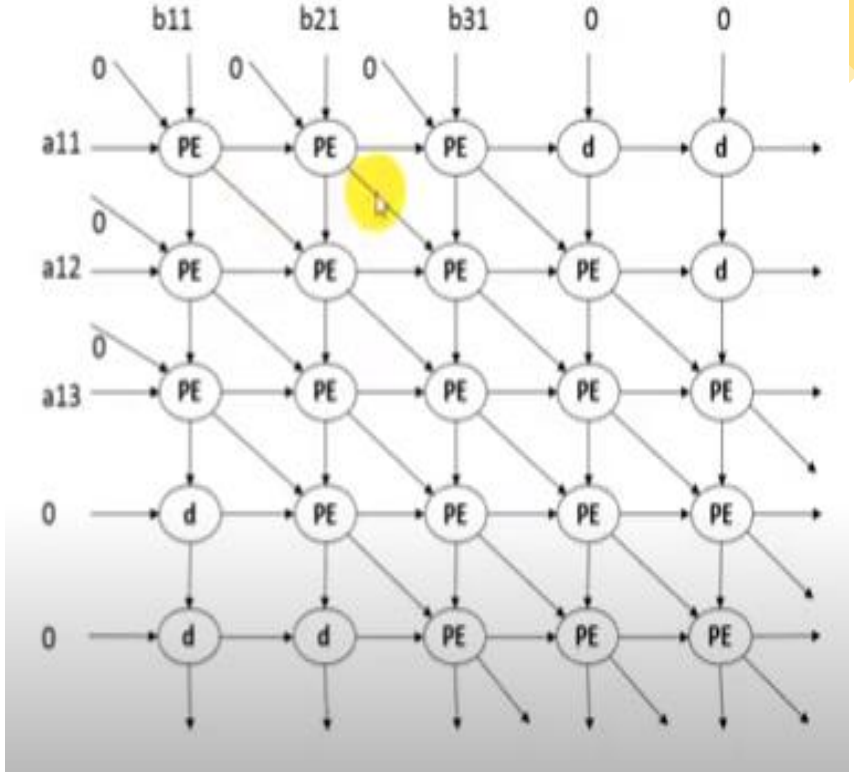


Example: Systolic arrays : eg: 3 x 3 matrix multiplication

T1 Cycle



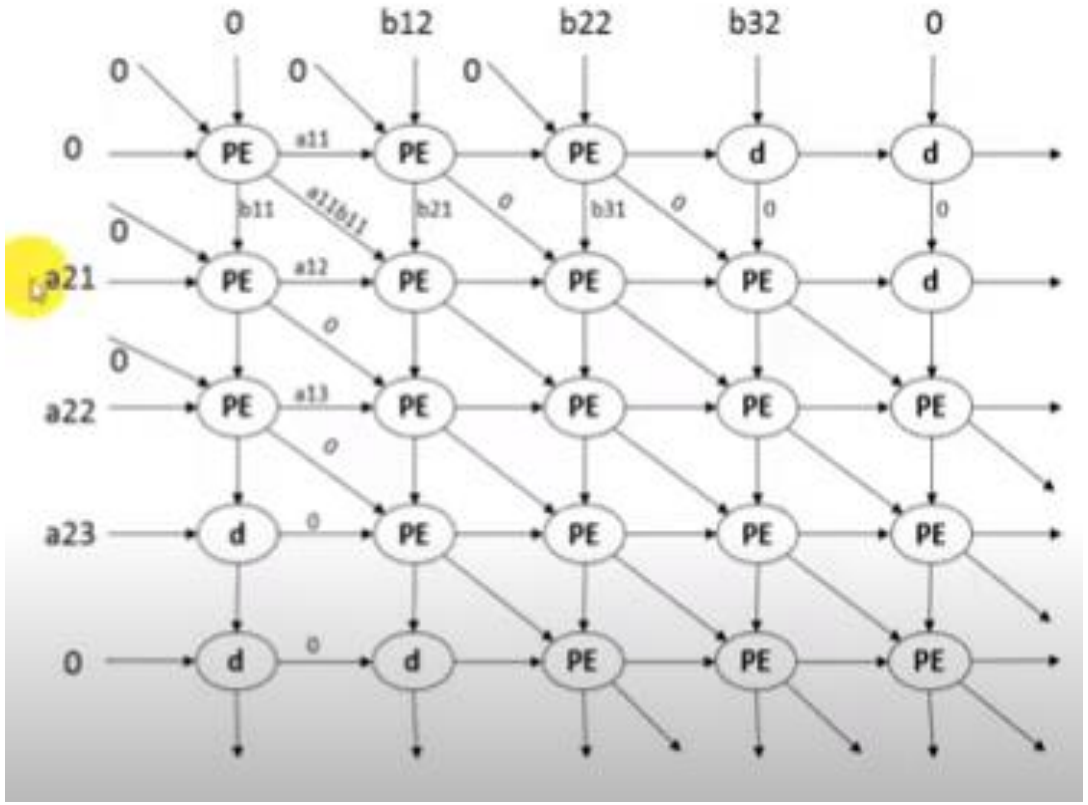
c11 = c11 + a11b11 + a12b21 + a13b31
c12 = c12 + a11b12 + a12b22 + a13b32
c13 = c13 + a11b13 + a12b23 + a13b33
c21 = c21 + a21b11 + a22b21 + a23b31
c22 = c22 + a21b12 + a22b22 + a23b32
c23 = c23 + a21b13 + a22b23 + a23b33
c31 = c31 + a31b11 + a32b21 + a33b31
c32 = c31 + a31b12 + a32b22 + a33b32
c33 = c33 + a31b13 + a32b23 + a33b33



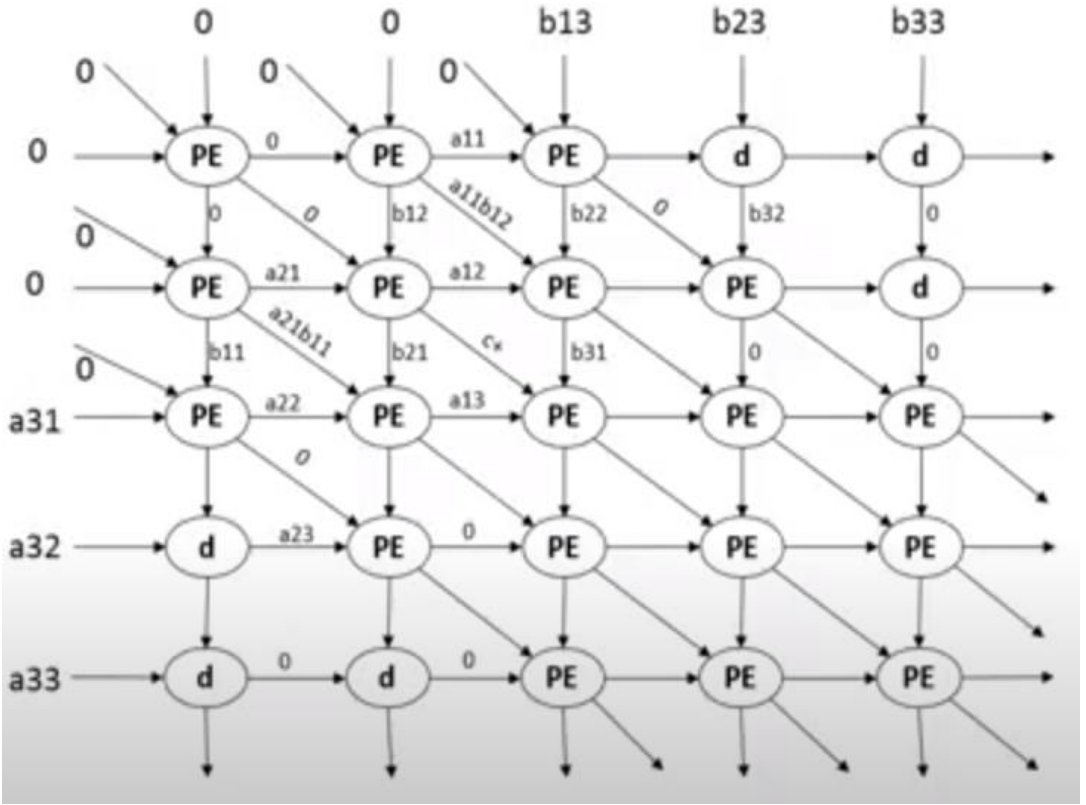
Ref:https://www.javatpoint.com/flynns-classification-of-computers

https://www.youtube.com/watch?v=vADVh1ogNo0

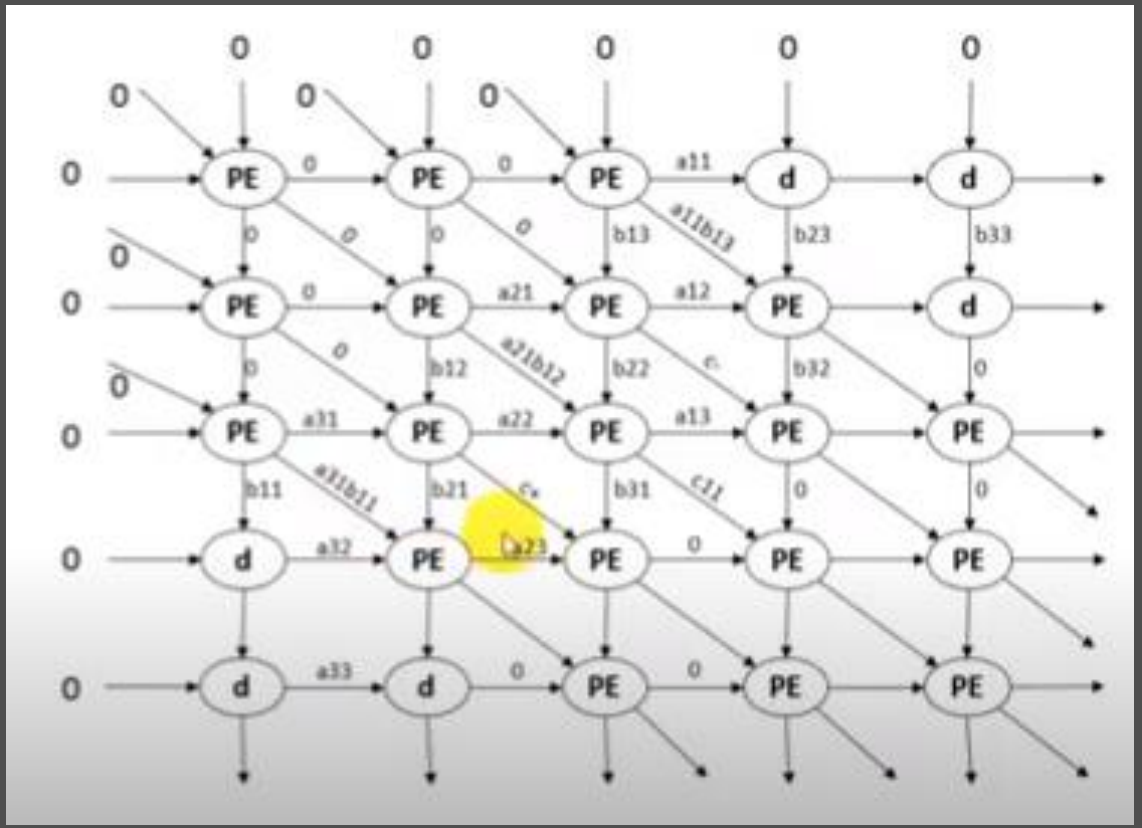
T2 Cycle



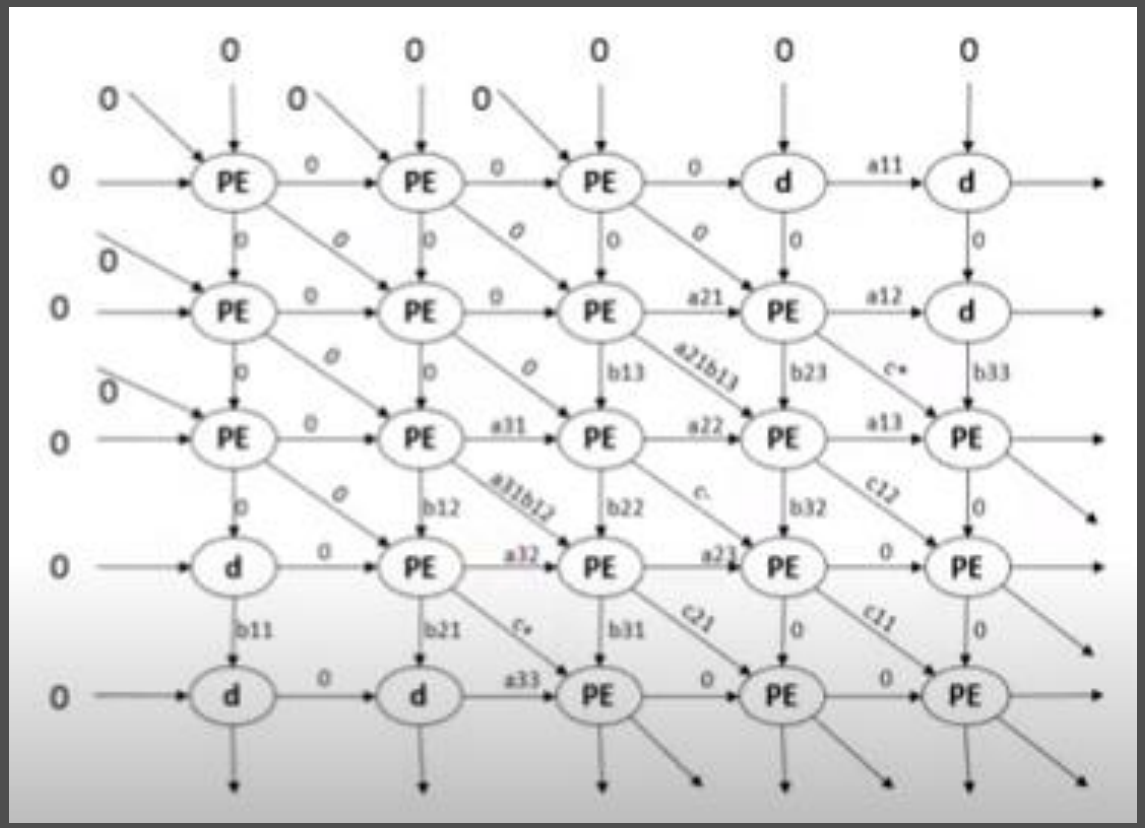
T3 Cycle



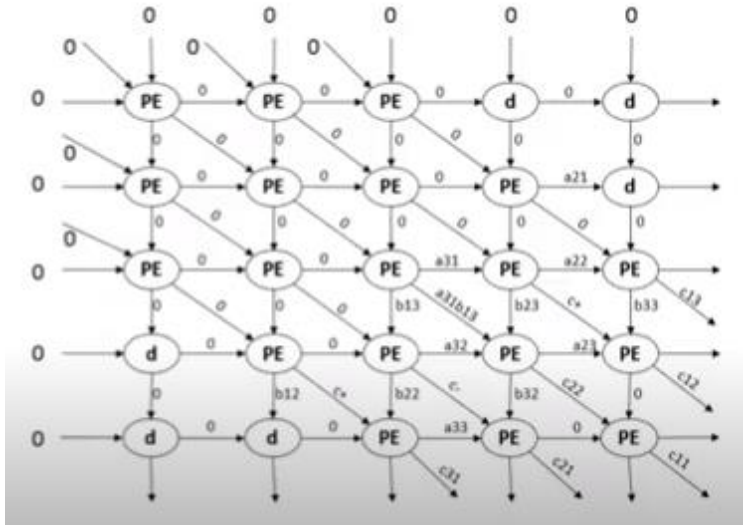
T4 Cycle



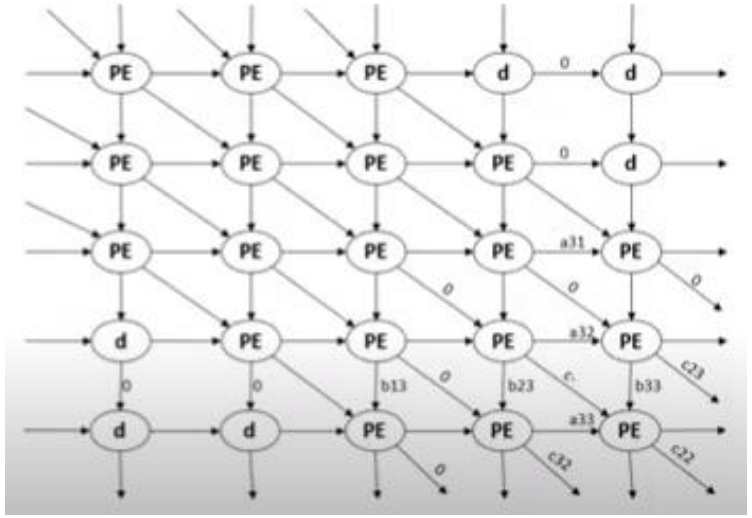
T5 Cycle



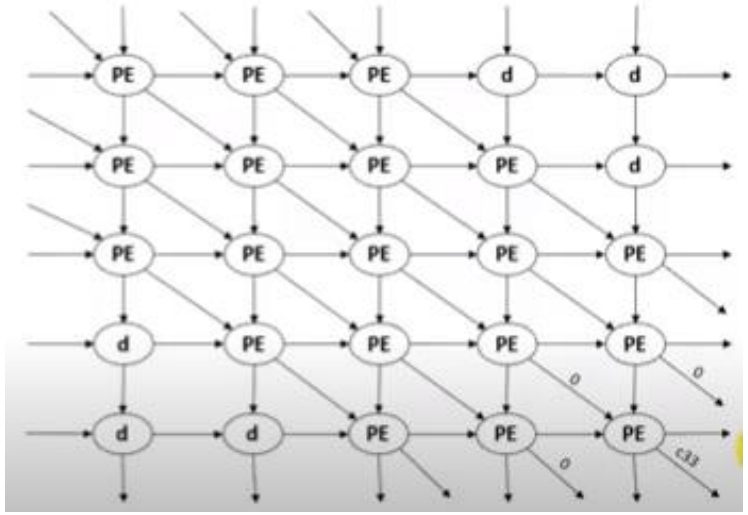
T6 Cycle



T7 Cycle

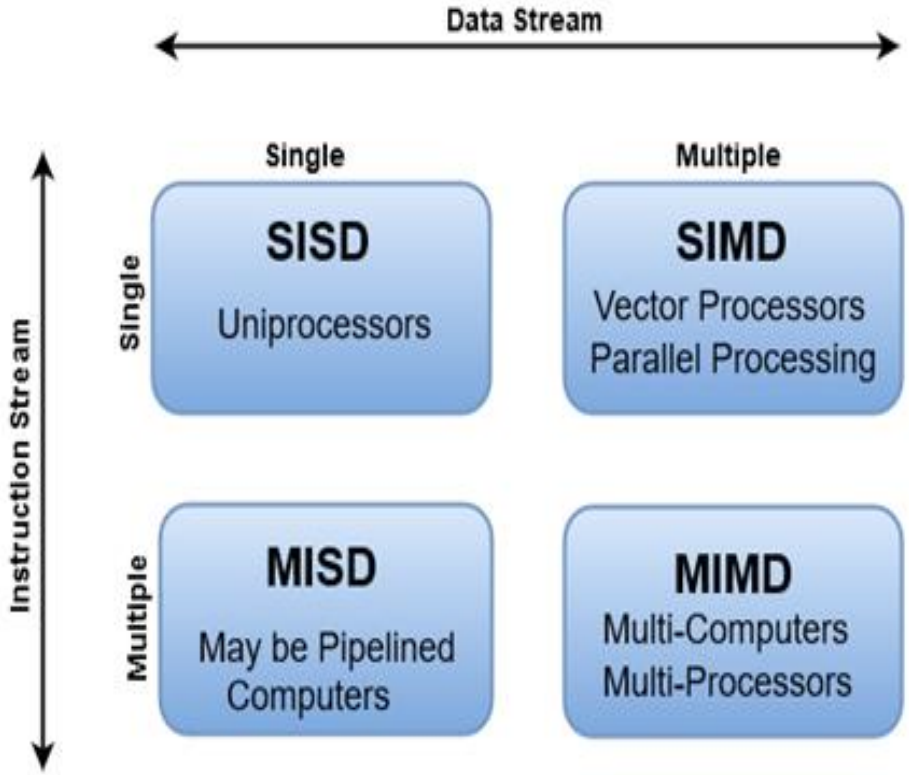


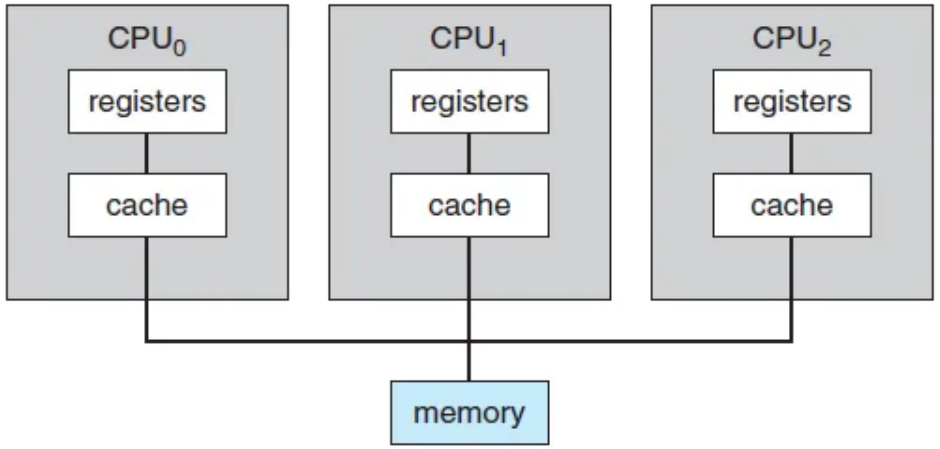
T8 Cycle

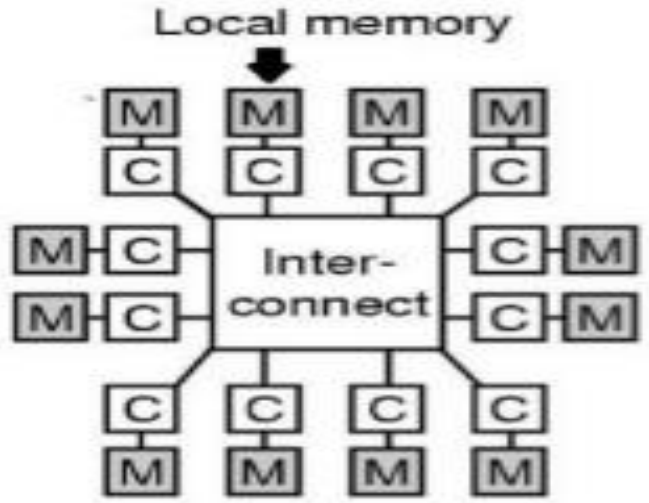


Flynn's Classification MIMD

Flynn's Classification







Ref:https://www.javatpoint.com/flynns-classification-of-computers

2.Memory Models

Parallel Computers Architectural Model/ Physical Model

Distinguished by having-

1. Shared Common Memory:

Three Shared-Memory Multiprocessor Models are:

i. UMA (Uniform-Memory Access)

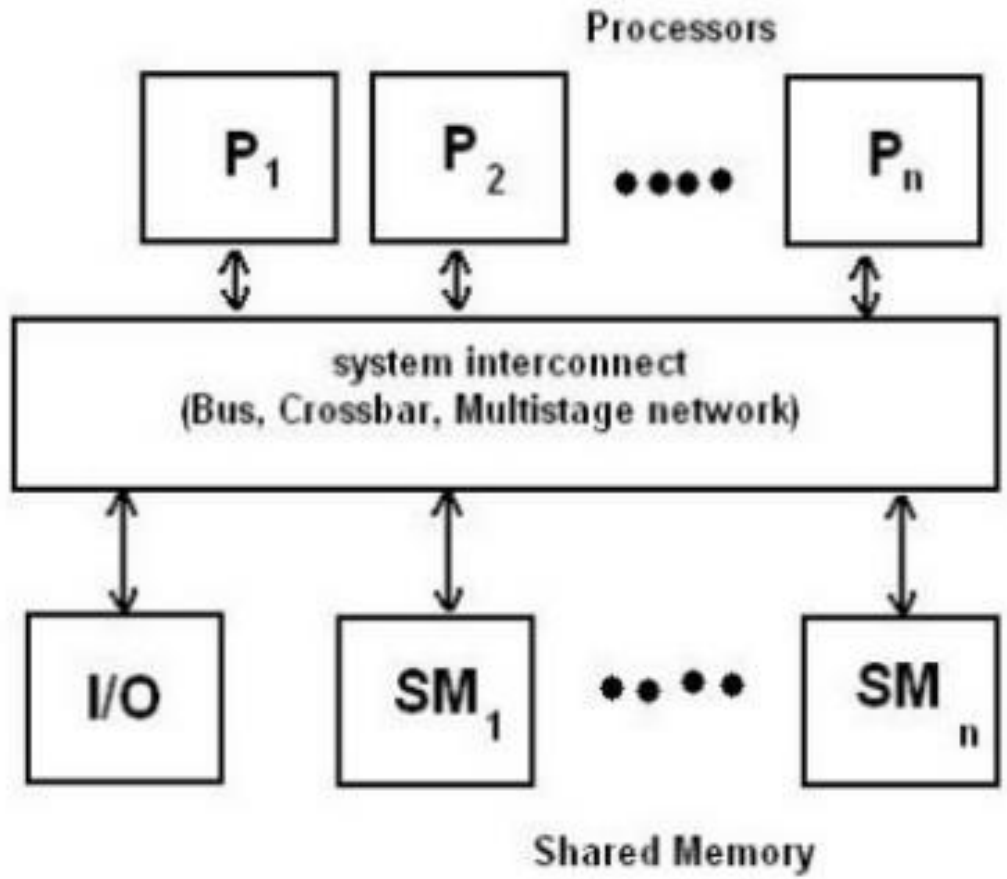
ii. NUMA (Non-Uniform-Memory Access)

iii. COMA (Cache-Only Memory Architecture)

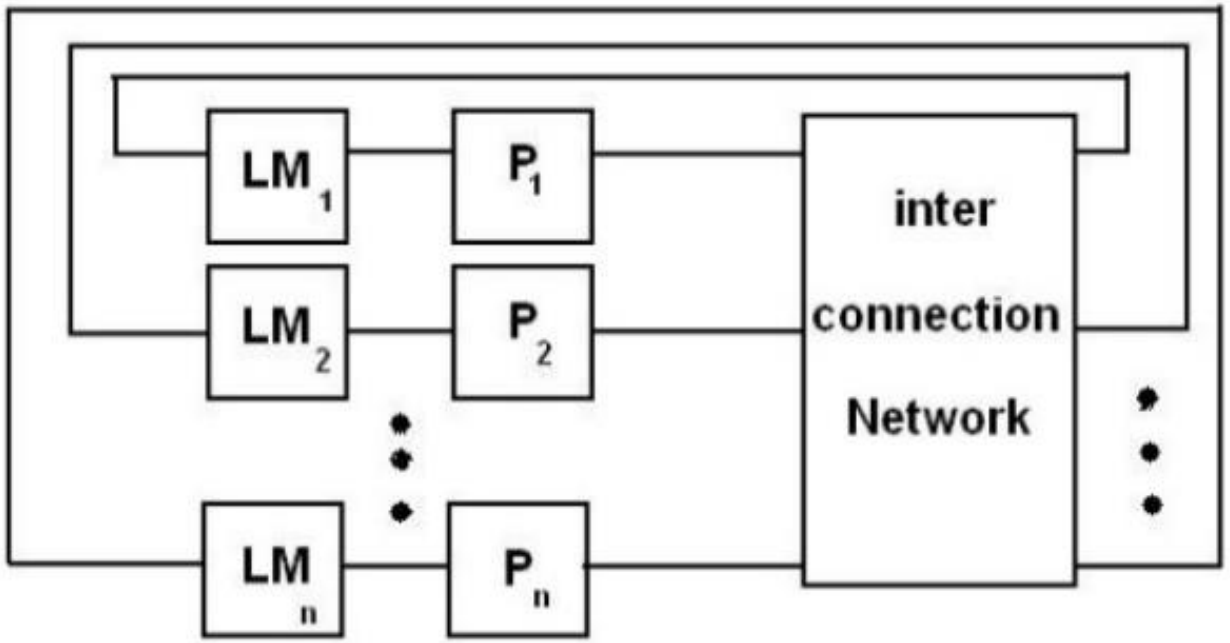
2. Unshared Distributed Memory

i. CC-NUMA (Cache-Coherent -NUMA)

UMA Multiprocessor Model



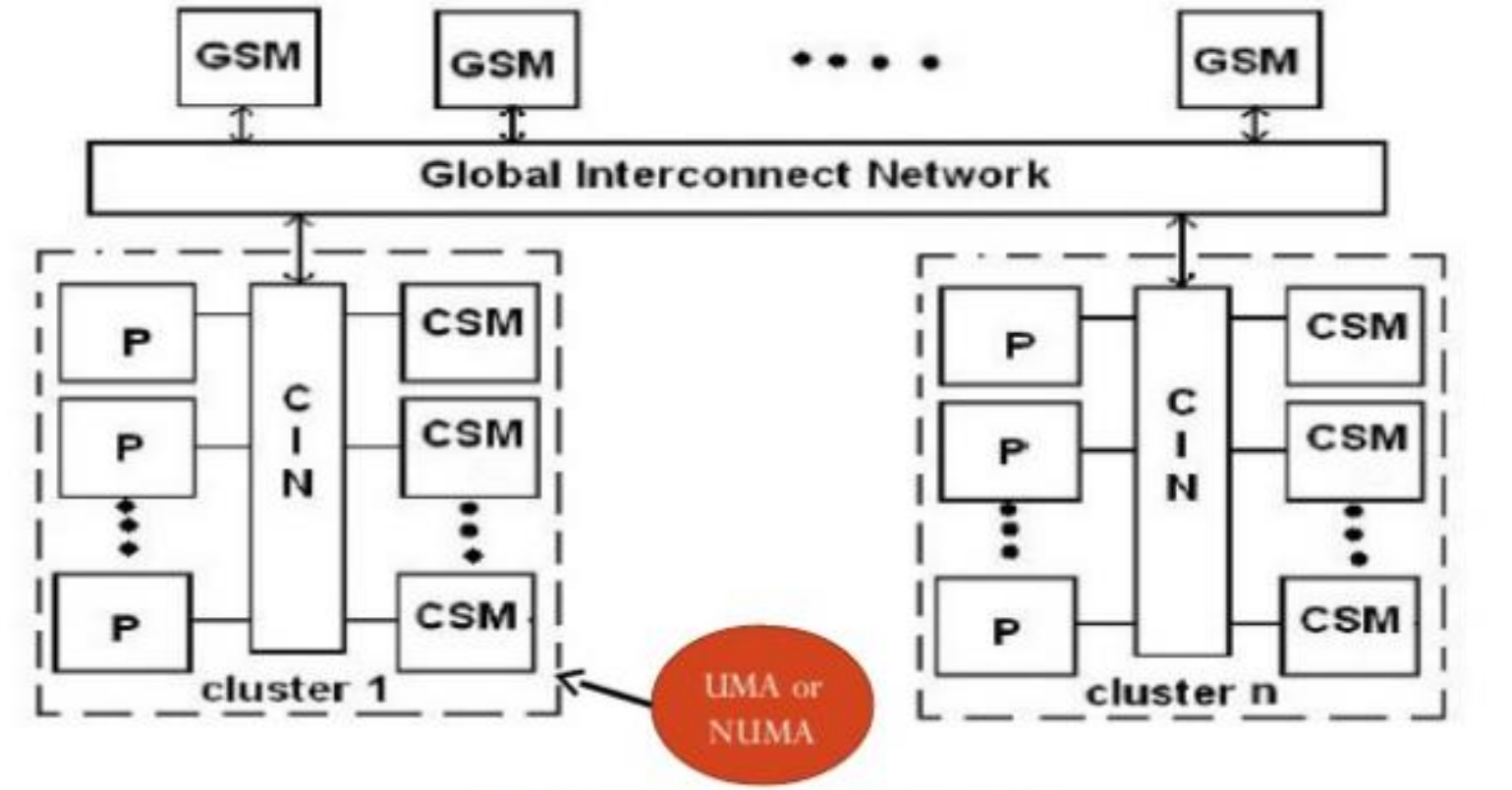
NUMA - Memory Models



(a) Shared local memories

LM – Local Memory
P - Local Processor

NUMA - Memory Models

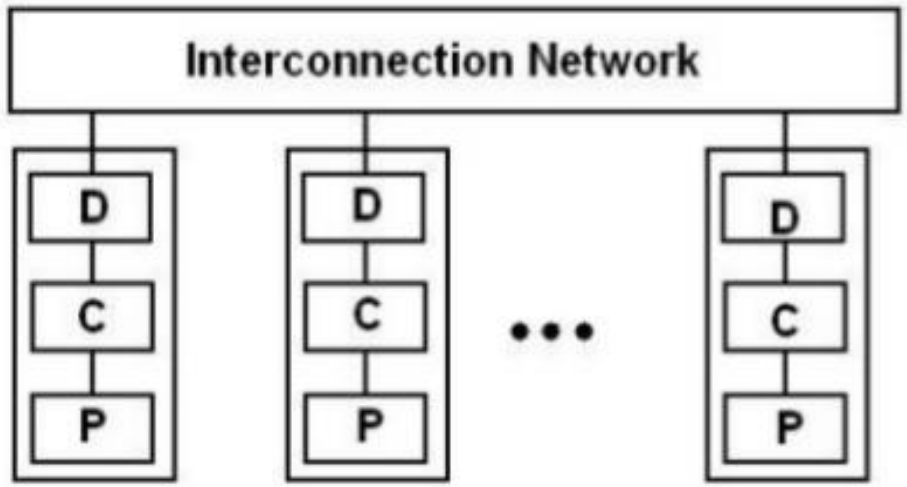


(b) A hierarchical cluster model

(Access of Remote Memory)

P – Processor
CSM – Cluster Shared Memory
CIN – Cluster Interconnection Network
GSM – Global Shared Memory

COMA Multiprocessor Model



C – Cache
P – Processor
D - Directories

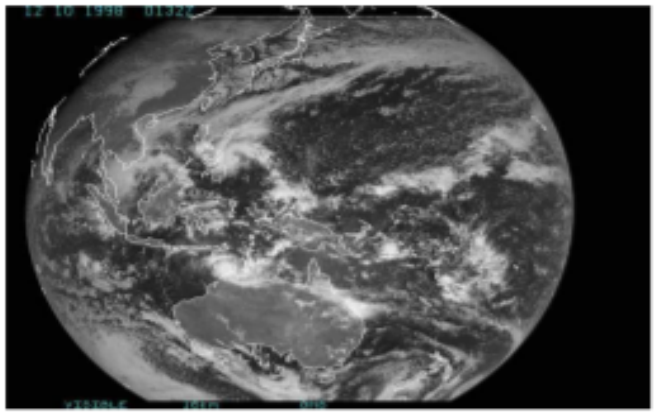
- Distributed Main Memory converted to Cache
- Cache form Global Address Space
- Remote Cache access by – Distributed cache Directories

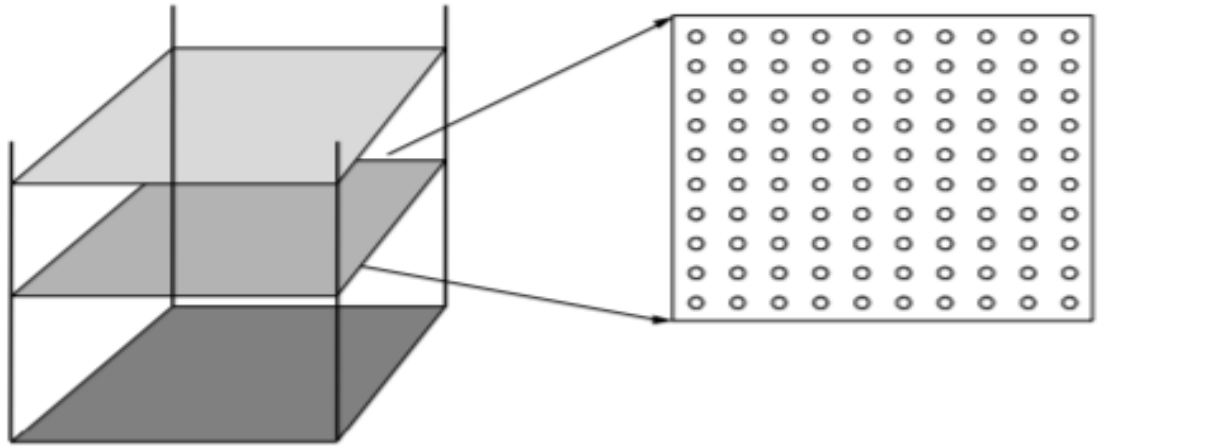
3. Perspective on parallel programming

- Motivating Problems
- Process of creating a parallel program

3. Perspective on parallel programming

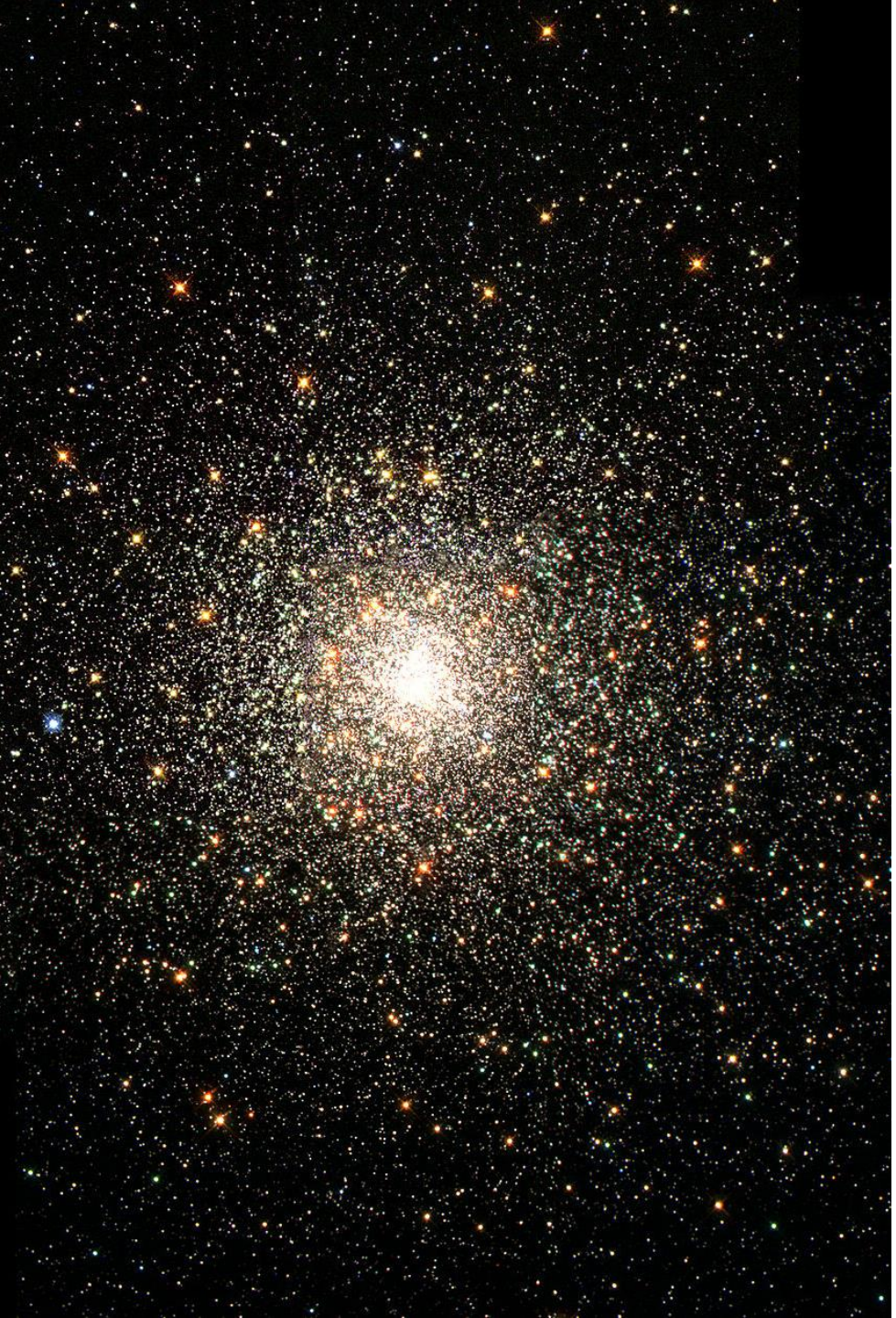
- Motivating Problems : Simulating Ocean Currents
  - Model as two dimensional grid :
    - Discretize in space and time
    - Finer and temporal resolution => greater accuracy
  - Many different computations per time step





(a) Cross sections

(b) Spatial discretization of a cross section



3. Perspective on parallel programming

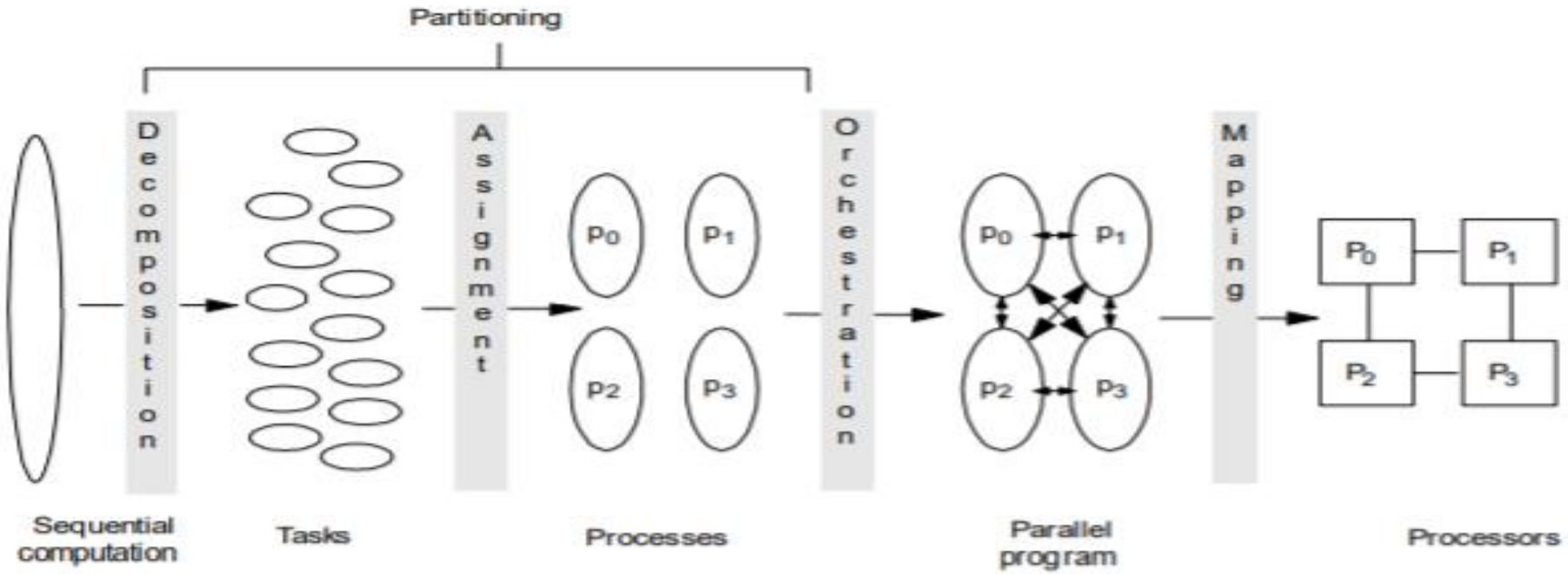
- Motivating Problems : Simulating interactions of many stars evolving over time.
  - Computing forces is expensive

  Eg. Stars on which forces of other elements need to be computed
  - Many time steps, plenty of concurrency across stars within each step.

3. Perspective on parallel programming

- Creating a parallel program
  - Identify the work that can be done in parallel : computation, data access and I/O
  - Partition of work and perhaps data among processes
  - Manage data access, communication and synchronization

3. Perspective on parallel programming



- **Decomposition** of computation in tasks
- **Assignment** of tasks to processes
- **Orchestration** of data access, comm, synch.
- **Mapping** processes to processors

Reference

Text Books and/or Reference Books:

1. Professional CUDA C Programming – John Cheng, Max Grossman, Ty McKercher, 2014
2. B.Wilkinson, M.Allen, ”Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers”, Pearson Education, 1999
3. I.Foster, ”Designing and building parallel programs”, 2003
4. Parallel Programming in C using OpenMP and MPI – Micheal J Quinn, 2004
5. Introduction to Parallel Programming – Peter S Pacheco, Morgan Kaufmann Publishers, 2011
6. Advanced Computer Architectures: A design approach, Dezso Sima, Terence Fountain, Peter Kacsuk, 2002
7. Parallel Computer Architecture : A hardware/Software Approach, David E Culler, Jaswinder Pal Singh Anoop Gupta, 2011 8. Introduction to Parallel Computing, Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar, Pearson, 2011

Thank You

National Institute of Technology Karnataka Surathkal
Department of Information Technology

IT 301 Parallel Computing

Memory Models and Cache Coherence

Dr. Geetha V

Assistant Professor

Dept of Information Technology

NITK Surathkal

Course Outline

Course Plan: Theory:

Part A: Parallel Computer Architectures

Week 1,2,3: ***Introduction to Parallel Computer Architecture:*** Parallel Computing, Parallel architecture, bit level, instruction level , data level and task level parallelism. Instruction level parelllelism: pipelining(Data and control instructions), scalar and superscalar processors, vector processors. Parallel computers and computation.

Week 4,5: Memory Models: UMA, NUMA and COMA. Flynns classification, Cache coherence,

Week 6,7: Amdahl's Law. Performance evaluation, Designing parallel algorithms : Divide and conquer, Load balancing, Pipelining.

Week 8 -11: ***Parallel Programming techniques like Task Parallelism using TBB, TL2, Cilk++ etc. and software transactional memory techniques.***

Course Outline

Part B: OpenMP/MPI/CUDA

Week 1,2,3 : ***Shared Memory Programing Techniques:*** Introduction to OpenMP : Directives: parallel, for, sections, task, single, critical, barrier, taskwait, atomic. Clauses: private, shared, firstprivate, lastprivate, reduction, nowait, ordered, schedule, collapse, num_threads, shared, if().

Week 4,5: ***Distributed Memory programming Techniques:*** MPI: Blocking, Non-blocking.

Week 6,7 : CUDA : OpenCL, Execution models, GPU memory, GPU libraries.

Week 10,11,: ***Introduction to accelerator programming using CUDA/OpenCL and Xeon-phi. Concepts of Heterogeneous programming techniques.***

Practical:

Implementation of parallel programs using OpenMP/MPI/CUDA.

Assignment: Performance evaluation of parallel algorithms (in group of 2 or 3 members)

Index

1. Memory models
2. Cache Coherence
    1. Cache coherence problem
    2. Snooping Bus Protocols: MSI.MESI

2.Memory Models

Parallel Computers Architectural Model/ Physical Model

Distinguished by having-

1. Shared Common Memory:

Three Shared-Memory Multiprocessor Models are:

i. UMA (Uniform-Memory Access)

ii. NUMA (Non-Uniform-Memory Access)

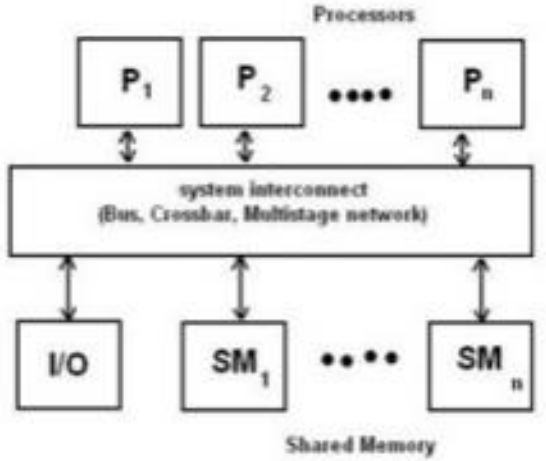
iii. COMA (Cache-Only Memory Architecture)

2. Unshared Distributed Memory

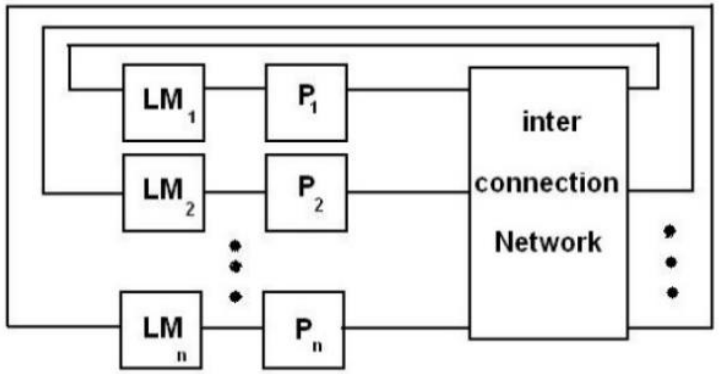
i. CC-NUMA (Cache-Coherent -NUMA)

Memory Models: UMA, NUMA and COMA

Uniform Memory Access



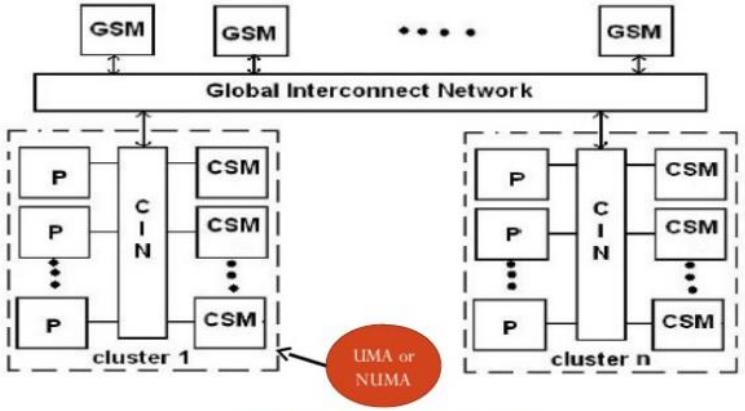
Non Uniform Memory Access



(a) Shared local memories

LM – Local Memory
P - Local Processor

Non Uniform Memory Access

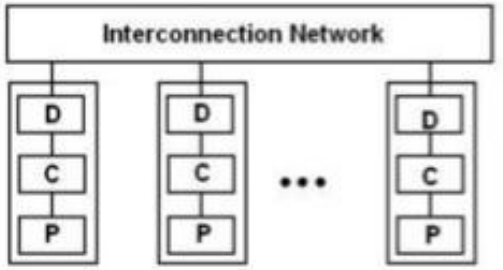


(b) A hierarchical cluster model

(Access of Remote Memory)

P – Processor
CSM – Cluster Shared Memory
CIN – Cluster Interconnection Network
GSM – Global Shared Memory

Cache Only Memory Access



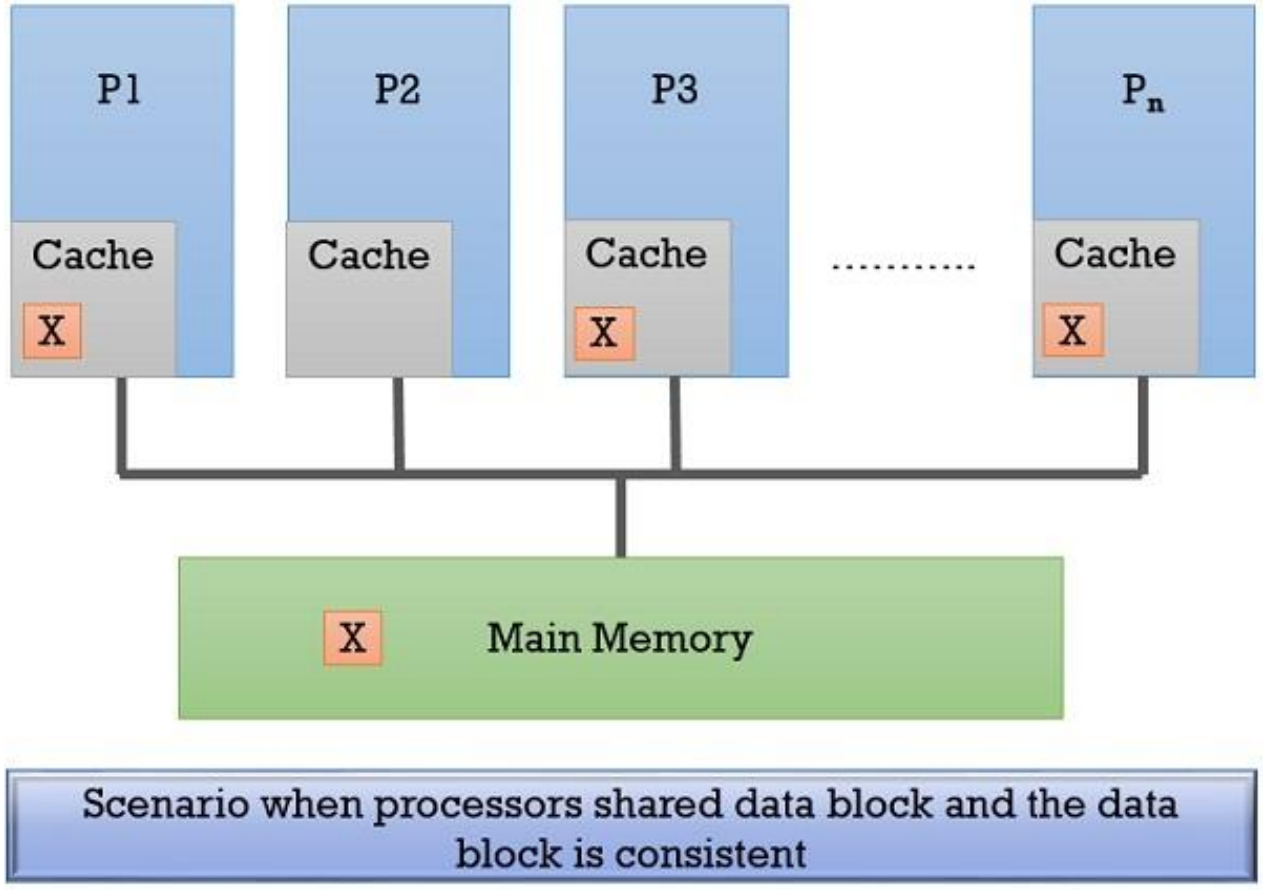
C – Cache
P – Processor
D - Directories

• Distributed Main Memory converted to Cache

•Cache form Global Address Space

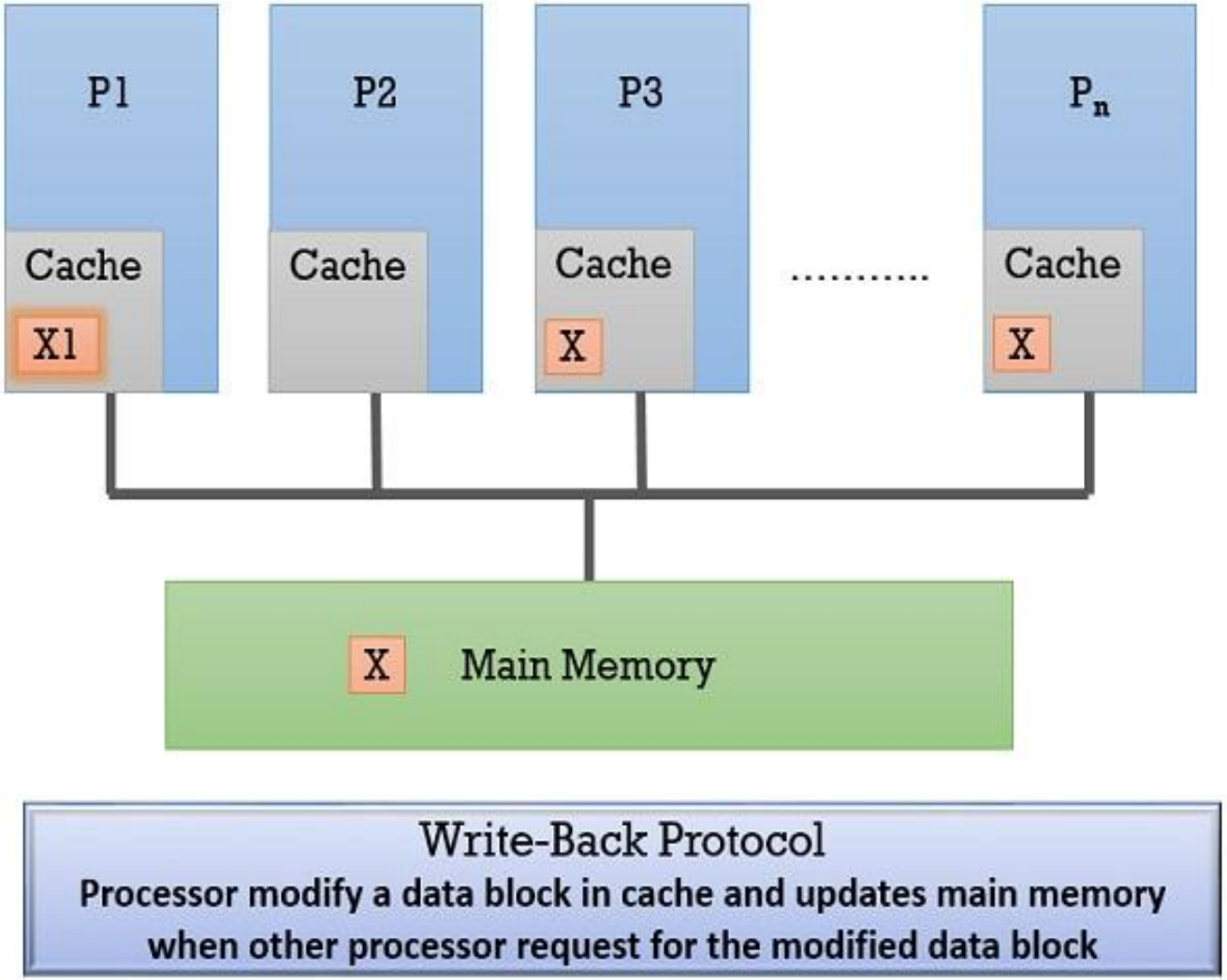
•Remote Cache access by – Distributed cache Directories

Shared Memory Model



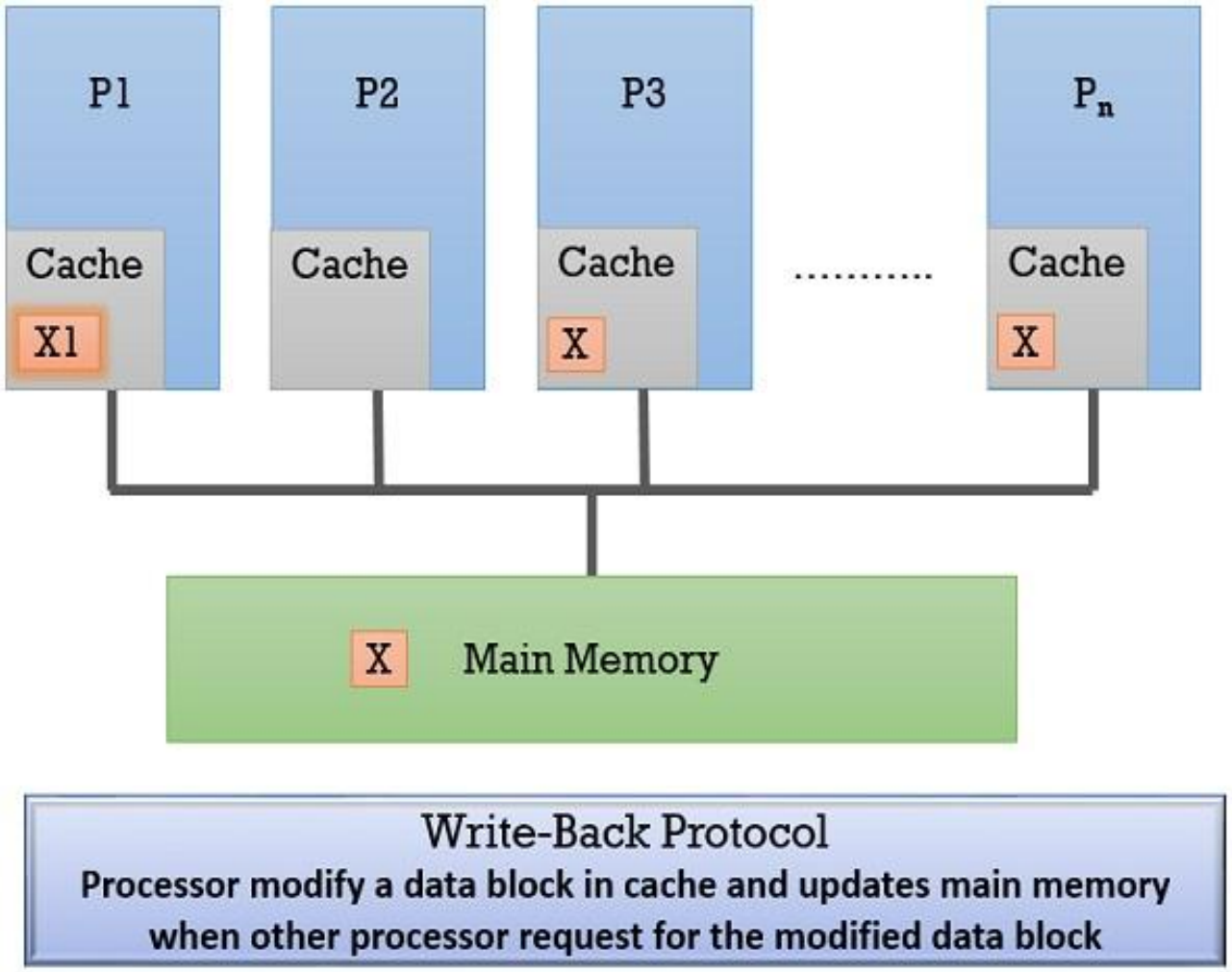
- Main memory is shared among all the processors.
- Accessing main memory every time is time consuming
- Better to have private cache for all the processors
- Shared memory consistency must be maintained.
- When the data block is shared among all other processors, the change in one cache must be reflected in all other processor cache. Then, the system can be considered as memory consistency is maintained.

Shared Memory Model



- Cache Coherent Problem:
- Caching of shared data, however, introduces **Cache Coherence problem.**
- **Shared data can have different data in different cache.**
- This must be handled properly

Shared Memory Model



- Cache Coherent: A memory system is coherent if any read of a data items return the most recently written value of that data item .

Shared Memory Model

- A memory system is consistent if the following hold good:

1. A read by a processor P to a location X that follows a write by P to X, with no writes of X by another processor occurring between the write and the read by P, always returns the value written by P. **: Maintains program order**

Assume X=5;

N=Read X; Write X, 10; N=Read X : Here, P must get value 10 , written by P

2. A read by a processor to location X that follows a write by another processor to X returns the written value if the read and write are sufficiently separated in time and no other writes to X occur between the two accesses. **: Coherent rule of memory**

Assume X=5;

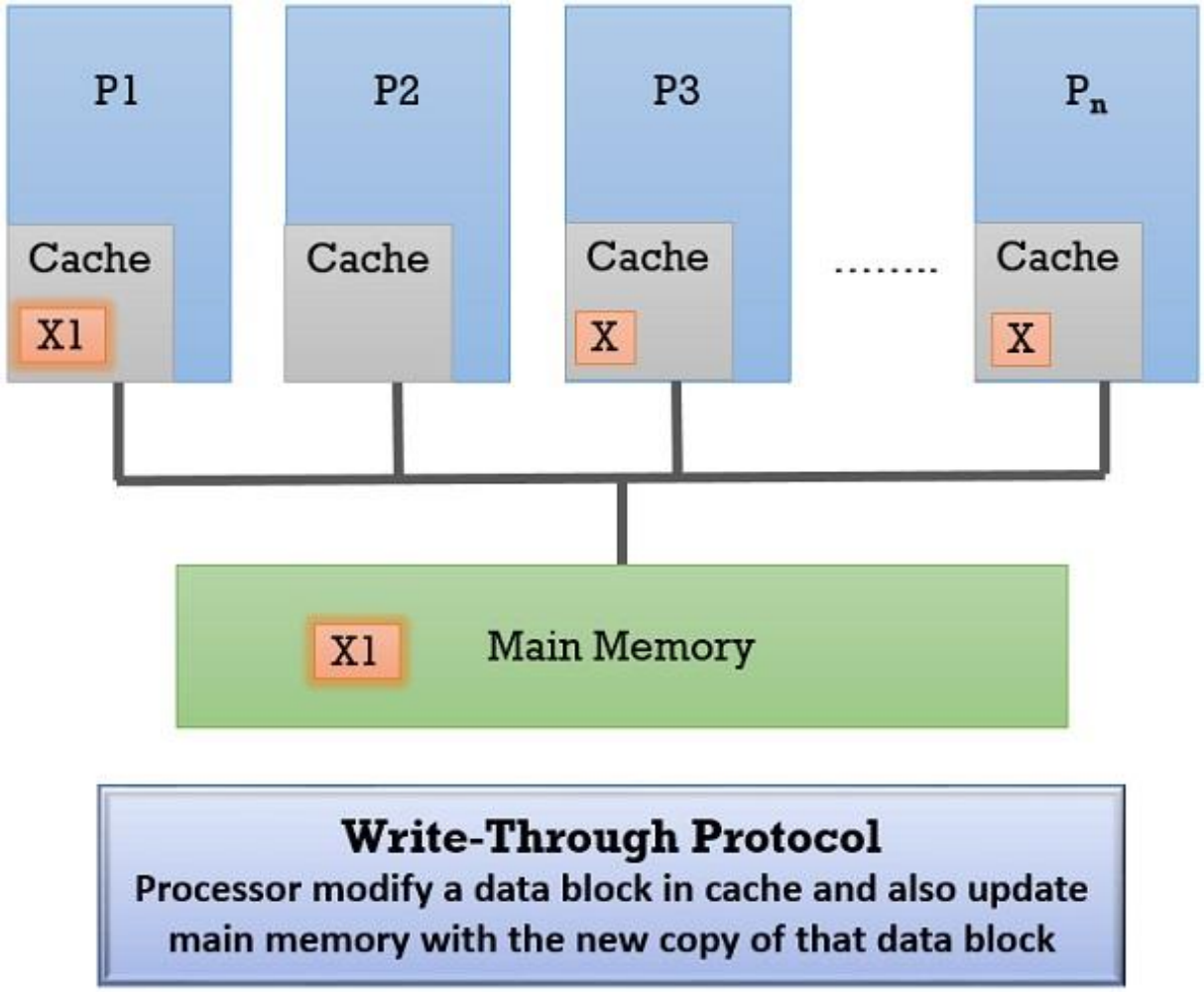
N=Read X by P1; Write X, 10 by P2; N=Read X by P1 : Here, must get value 10 , written by P2

3. Writes to the same location are *serialized;* that is, two writes to the same location by any two processors are seen in the same order by all processors. This ensures that we do not see the older value after the newer value. **: Serialization of memory operations**

Assume X=5;

Write X, 10 by P1; Write X, 15 by P2; N=Read X by P1 : Here, must get value 15 , written by P2

Cache Coherence Protocols

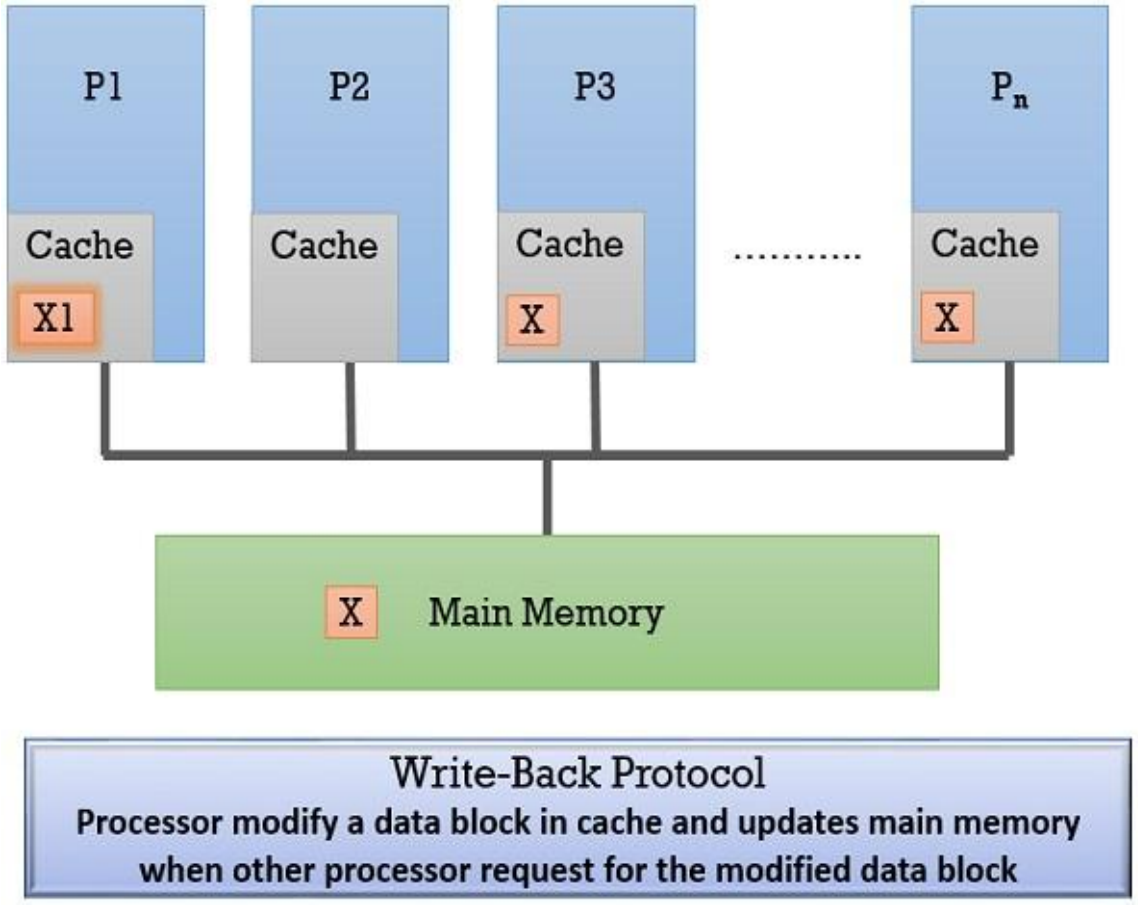


- **Memory Write Operations**

1. Write Through:

- When a processor modifies a data block in its cache, it immediately updates the main memory with the new copy of the same data block.
- Main memory has consistent data.
- Cache coherence : Either through update or invalidate

Cache Coherence Protocols

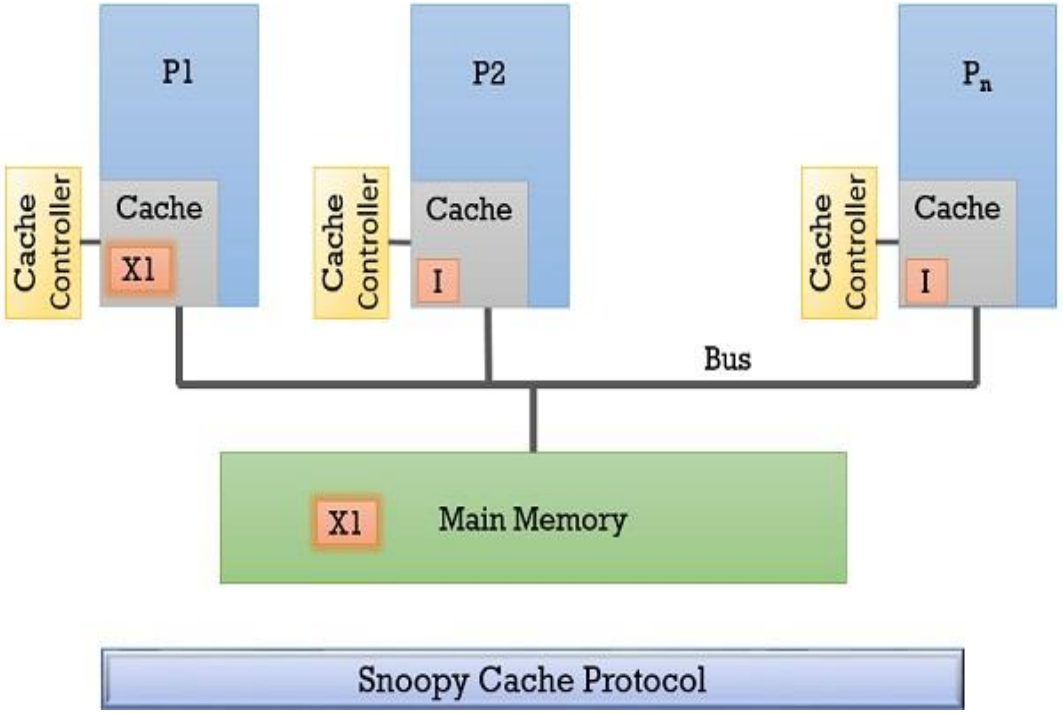


- **Memory Write Operations**

1. Write Back:

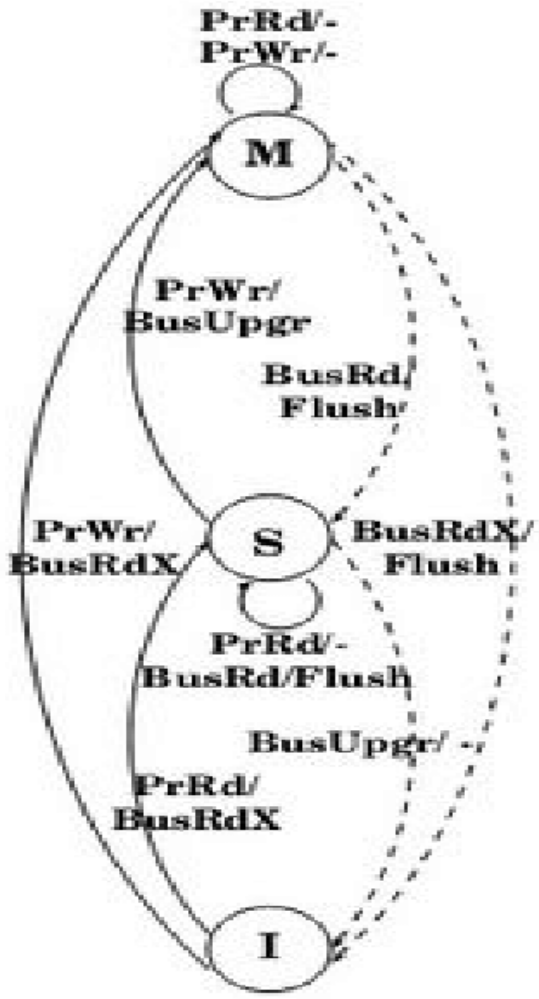
- When a processor modifies a data block in its cache, it **does not** immediately update the main memory with the new copy of the same data block. Write back durig cache replacement or request from other processors.
- That processor has the data exclusively
- **Cache coherence : invalidate all other cache entries and update when there is request.**

Cache Coherence Protocols : Snoopy Bus Protocol

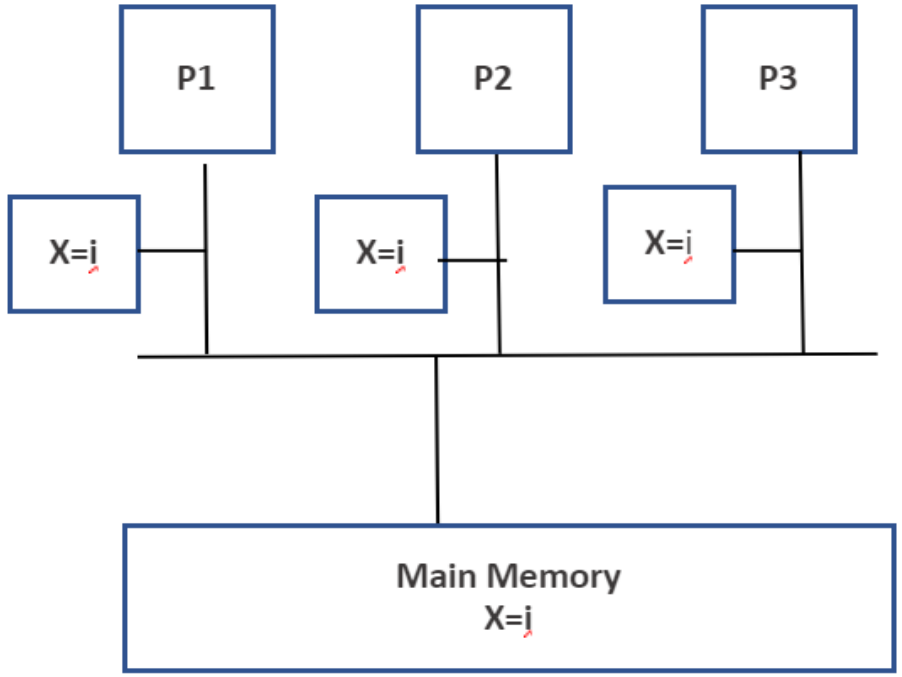


- **Snoopy Bus Protocol**
- Main memory is shared among processors via single bus.
- Cache controller performs snoop operation on all transaction over bus.
- Snoop bus protocol is hardware solution.
- It is used for small multiprocessor environment

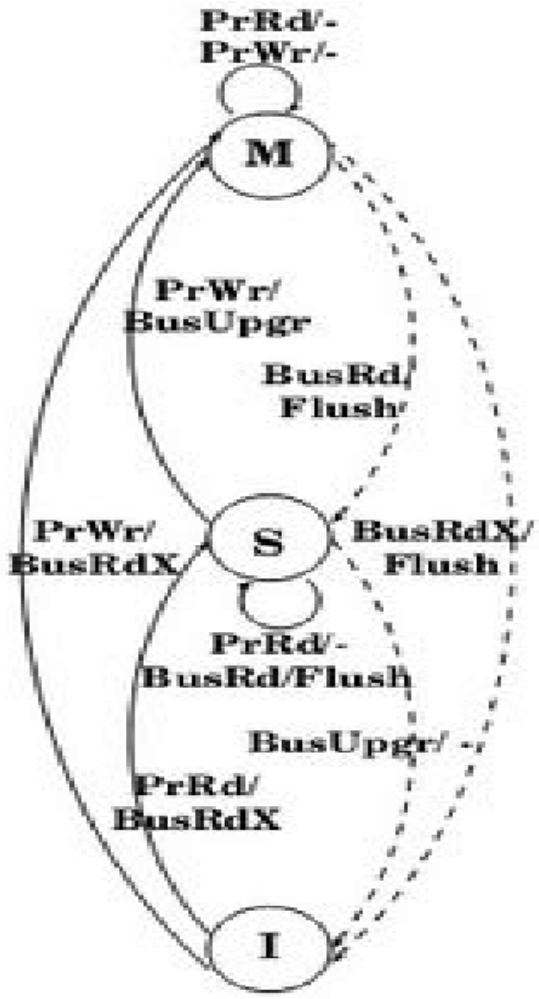
Snoopy Bus Protocol MSI:Modified-Shared-Invalid



- **Snoopy Bus Protocol**
- Invalid State: Initially all the processor cache elements are invalid.

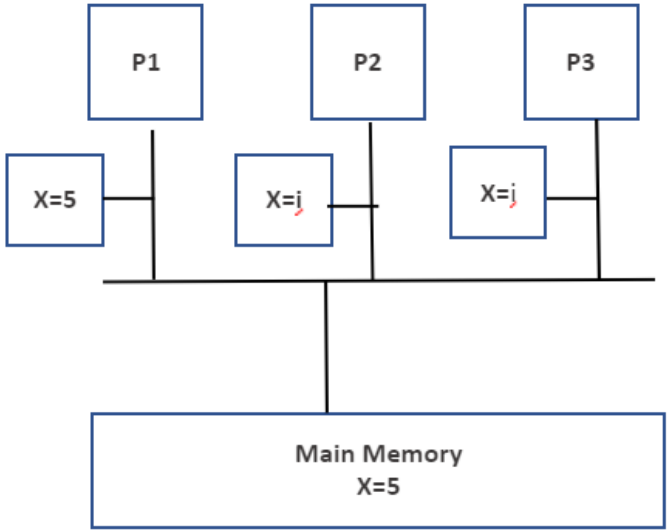


Snoopy Bus Protocol MSI:Modified-Shared-Invalid

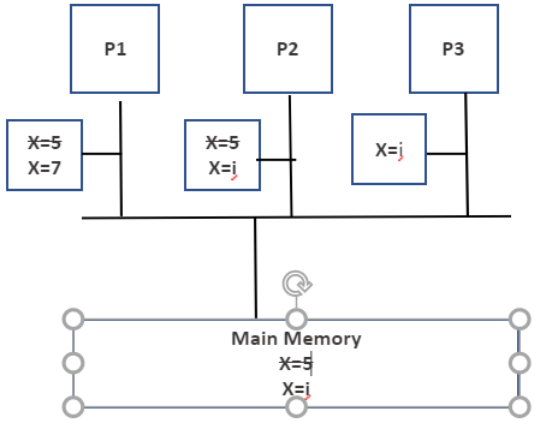


- **Snoopy Bus Protocol**
- Shared State: Initially all the processor cache elements are invalid.

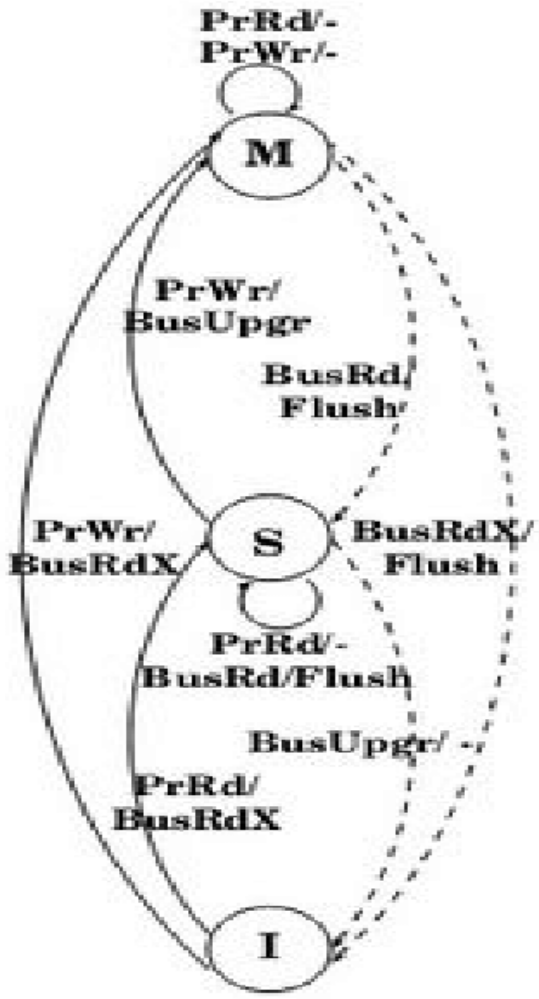
I-S: PrRd/BusRdx: P1 sending read



S-M: PrWr/BusUpgr : P1 in shared state, send PrWr

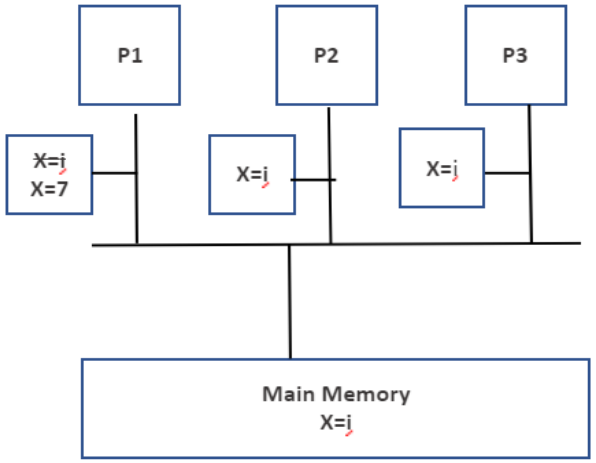


Snoopy Bus Protocol MSI:Modified-Shared-Invalid

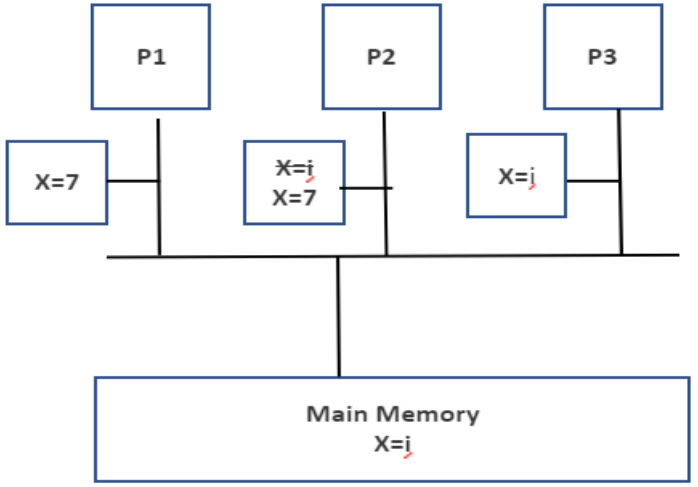


- **Snoopy Bus Protocol**
- Modified State: Initially all the processor cache elements are invalid.

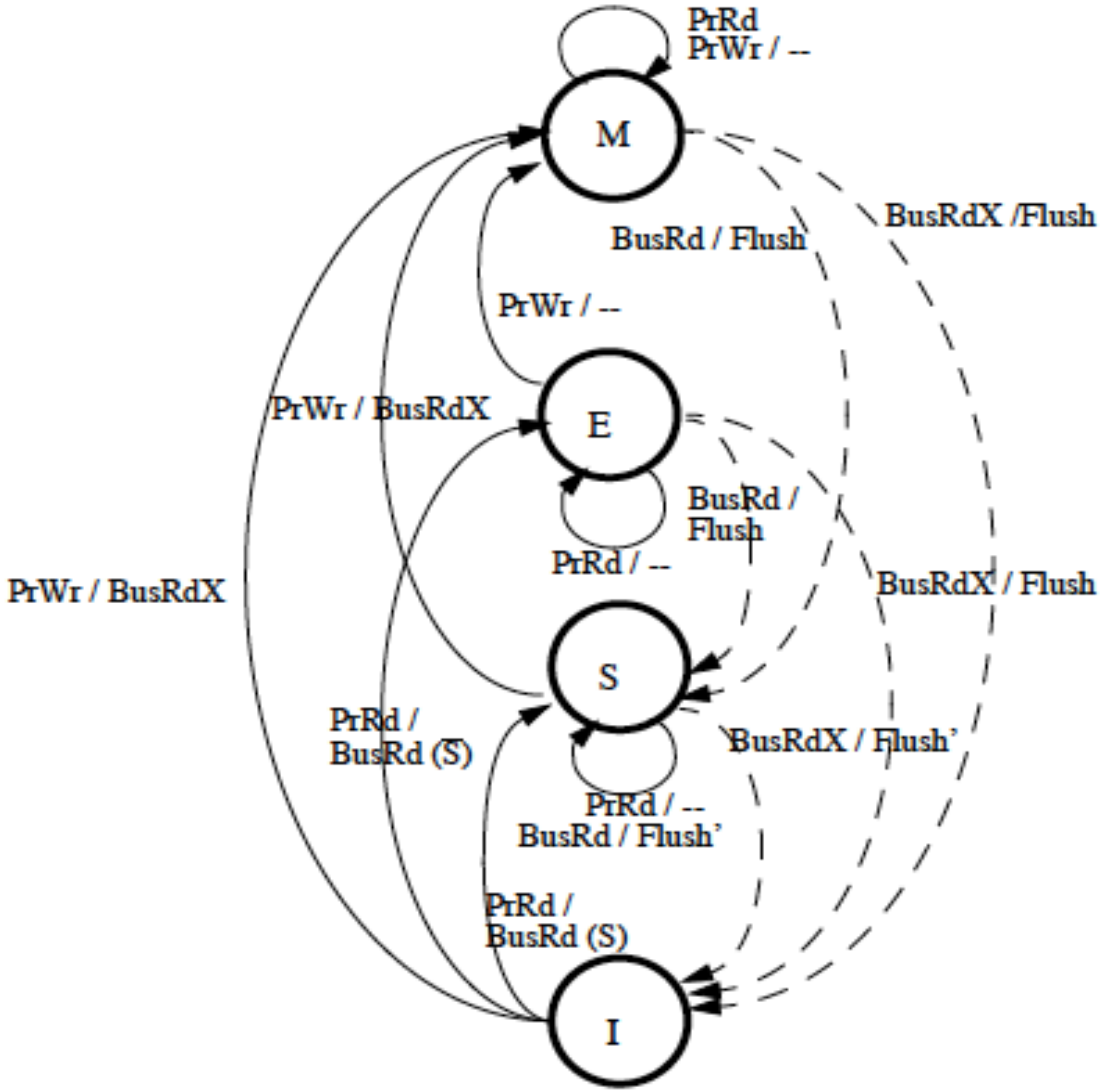
I-M: PrWr/BusRdx: P1 sending Write



M-S: BusRd: P2 sending PrRd



Snoopy Bus Protocol MESI:Modified-Exclusive-Shared-Invalid



- **Snoopy Bus Protocol: MESI**
- Invalid State: The data items are invalid initially.
- Shared state: Valid copy of data in many cache
- Exclusive: The processor wants to perform write operation, and it has exclusive copy. **Invalid to Excusive**
- Modified State: Data block is modified, and the processor has modified data. Shared to Modified.

Summary

1. Memory models
2. Cache Coherence
    1. Cache coherence problem
    2. Snooping Bus Protocols: MSI.MESI

Reference

Text Books and/or Reference Books:

1. Professional CUDA C Programming – John Cheng, Max Grossman, Ty McKercher, 2014
2. B.Wilkinson, M.Allen, ”Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers”, Pearson Education, 1999
3. I.Foster, ”Designing and building parallel programs”, 2003
4. Parallel Programming in C using OpenMP and MPI – Micheal J Quinn, 2004
5. Introduction to Parallel Programming – Peter S Pacheco, Morgan Kaufmann Publishers, 2011
6. Advanced Computer Architectures: A design approach, Dezso Sima, Terence Fountain, Peter Kacsuk, 2002
7. Parallel Computer Architecture : A hardware/Software Approach, David E Culler, Jaswinder Pal Singh Anoop Gupta, 2011
8. Introduction to Parallel Computing, Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar, Pearson, 2011

Thank You

National Institute of Technology Karnataka Surathkal
Department of Information Technology

IT 301 Parallel Computing

Scalar Processors – Control Instructions

Dr. Geetha V

Assistant Professor

Dept of Information Technology

NITK Surathkal

Index

- 1. Five Stage pipeline design
- 2: Control Hazards

1: Introduction

Course Plan: Theory:

Part A: Parallel Computer Architectures

Week 1,2,3: ***Introduction to Parallel Computer Architecture:***

Parallel Computing,

Parallel architecture,

bit level, instruction level , data level and task level parallelism.

Instruction level parallelisms: pipelining (Data and control instructions),

scalar processors and superscalar processors,

vector processors.

Parallel computers and computation.

1. Five Stage Pipeline Design

- Fetch Stage: The address in program counter is sent to Instruction register and instruction is fetched from memory. PC is incremented.
- Decode Stage: The operands are fetched from register file; Sign extension done if necessary.
- Execution Stage: The instruction is executed.
- Memory Stage: The Load and Store instructions execution.
- Write Back Stage: The results are written back to register file.

Fetch (F)	Decode (D)	Execute (E)	Memory Access (M)	Write Back (W)

1. Five Stage Pipeline Design

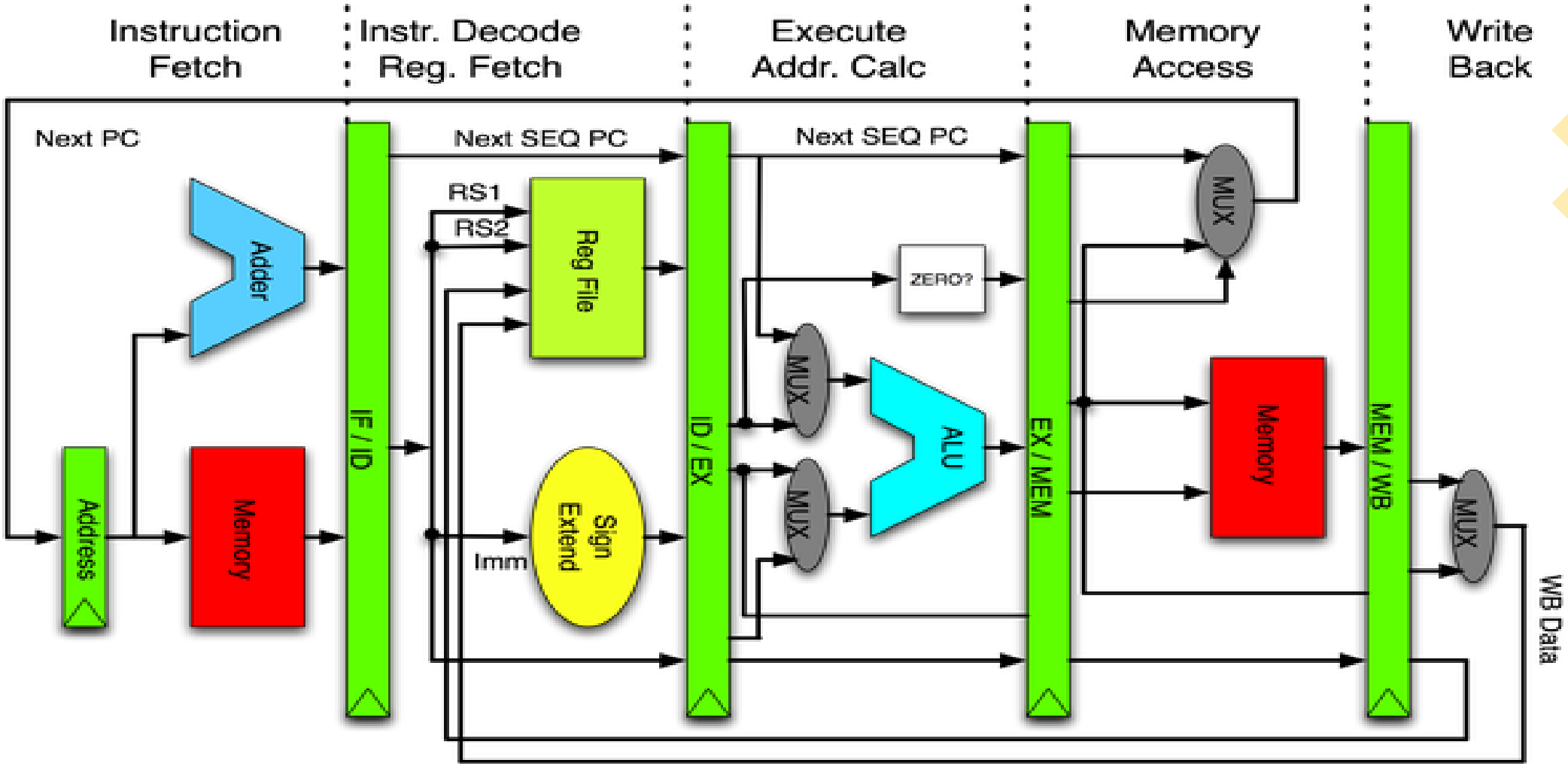


Image from Wikimedia

1: Five stage pipeline design

- **Data Dependency (Data Hazard)**

Dependency of instructions due to operand values unavailability

Solution :

(a) Data forwarding

(b) Stall the pipeline untill the data is available.

- **Control Dependency (Control Hazard)**

Dependency due to unresolved decision on control instructions.

2. Control hazards

- The branches – conditional branches – cause pipeline hazard
- The outcome of a conditional branch is not known until the end of the EX stage, but is required at IF to load another instruction and keep the pipeline full.

```
        JGE Next
        Add CX, 02
        Dec BL
        JMP NEXT
NEXT:
```

2. Control Hazards

- **Data Dependency (Data Hazard)**

Dependency of instructions due to operand values unavailability

Solution :

(a) Data forwarding

(b) Stall the pipeline untill the data is available.

- **Control Dependency (Control Hazard)**

Dependency due to unresolved decision on control instructions.

(a) Insert NOP (No Operation)

(b) NOP after execution of instruction

(c) Branch Prediction

2. Control hazards

Memory	Instruction	Operation
400	MOV SI, 500	SI <- 500
403	MOV DI, 600	DI <- 600
406	MOV AX, 0000	AX = 0000
409	MOV CL, [SI]	CL <- [SI]
40B	MOV BL, CL	BL <- CL
40D	INC SI	SI = SI + 1
40E	NXT: ADD AL, [SI]	AL = AL + [SI]
410	ADC AH, 00	AH = AH + 00 + cy
412	INC SI	SI = SI + 1
413	DEC CL	CL = CL – 1
415	JNZ NXT (40E)	JUMP if ZF = 0
417	DIV BL	AX = AX / BL
419	MOV [DI], AX	[DI] <- AX
41B	HLT	Stop

- Program to find average of array elements

(a) Inserting NOP

1. Insert NOP (No operation) when Branch instruction

F	D	E	M	W
NOP	NOP	JNZ 40E	DEC CL	INC SI

F	D	E	M	W
ADD AL, [SI]	NOP	NOP	JNZ 40E	DEC CL

2. Control hazards

Memory	Instruction	Operation
400	MOV SI, 500	SI <- 500
403	MOV DI, 600	DI <- 600
406	MOV AX, 0000	AX = 0000
409	MOV CL, [SI]	CL <- [SI]
40B	MOV BL, CL	BL <- CL
40D	INC SI	SI = SI + 1
40E	NXT: ADD AL, [SI]	AL = AL + [SI]
410	ADC AH, 00	AH = AH + 00 + cy
412	INC SI	SI = SI + 1
413	DEC CL	CL = CL – 1
415	JNZ NXT (40E)	JUMP if ZF = 0
417	DIV BL	AX = AX / BL
419	MOV [DI], AX	[DI] <- AX
41B	HLT	Stop

- Program to find average of array elements

(b) NOP after execution of instruction

2.Insert NOP (No operation) when Branch is taken (after execution of branch)

F	D	E	M	W
MOV [DI], AX	DIV BL	JNZ 40E	DEC CL	INC SI

F	D	E	M	W
~~MOV [DI], AX~~ NOP	~~DIV BL~~ NOP	JNZ 40E	DEC CL	INC SI

F	D	E	M	W
ADD AL, [SI]	~~MOV [DI], AX~~ NOP	~~DIV BL~~ NOP	JNZ 40E	DEC CL

2. Control hazards

Memory	Instruction	Operation
400	MOV SI, 500	SI <- 500
403	MOV DI, 600	DI <- 600
406	MOV AX, 0000	AX = 0000
409	MOV CL, [SI]	CL <- [SI]
40B	MOV BL, CL	BL <- CL
40D	INC SI	SI = SI + 1
40E	NXT: ADD AL, [SI]	AL = AL + [SI]
410	ADC AH, 00	AH = AH + 00 + cy
412	INC SI	SI = SI + 1
413	DEC CL	CL = CL – 1
415	JNZ NXT (40E)	JUMP if ZF = 0
417	DIV BL	AX = AX / BL
419	MOV [DI], AX	[DI] <- AX
41B	HLT	Stop

- Program to find average of array elements

(c) Branch Prediction : Branch Taken

3. Branch Prediction: Branch Taken

F	D	E	M	W
ADC AH, 00	ADD AL, [SI]	JNZ 40E	DEC CL	INC SI

Branch not taken

F	D	E	M	W
~~ADC AH, 00~~ NOP	~~ADD AL, [SI]~~ NOP	JNZ 40E	DEC CL	INC SI

F	D	E	M	W
DIV BL	~~ADC AH, 00~~ NOP	~~ADD AL, [SI]~~ NOP	JNZ 40E	DEC CL

Branch Taken

F	D	E	M	W
INC SI	ADC AH, 00	ADD AL, [SI]	JNZ 40E	DEC CL

Thank You

Reference

Textbooks and/or Reference Books:

1. Professional CUDA C Programming – John Cheng, Max Grossman, Ty McKercher, 2014
2. Wilkinson, M. Allen, ”Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers”, Pearson Education, 1999
3. I. Foster, ”Designing and building parallel programs”, 2003
4. Parallel Programming in C using OpenMP and MPI – Micheal J Quinn, 2004
5. Introduction to Parallel Programming – Peter S Pacheco, Morgan Kaufmann Publishers, 2011
6. Advanced Computer Architectures: A design approach, Dezso Sima, Terence Fountain, Peter Kacsuk, 2002
7. Parallel Computer Architecture : A hardware/Software Approach, David E Culler, Jaswinder Pal Singh Anoop Gupta, 2011 8. Introduction to Parallel Computing, Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar, Pearson, 2011

National Institute of Technology Karnataka Surathkal
Department of Information Technology

IT 301 Parallel Computing

Superscalar Processors 1

Dr. Geetha V

Assistant Professor

Dept of Information Technology

NITK Surathkal

Index

1: Introduction

Course Plan: Theory:

Part A: Parallel Computer Architectures

Week 1,2,3: ***Introduction to Parallel Computer Architecture:***

Parallel Computing,

Parallel architecture,

bit level, instruction level , data level and task level parallelism.

Instruction level parallelisms: pipelining (Data and control instructions),

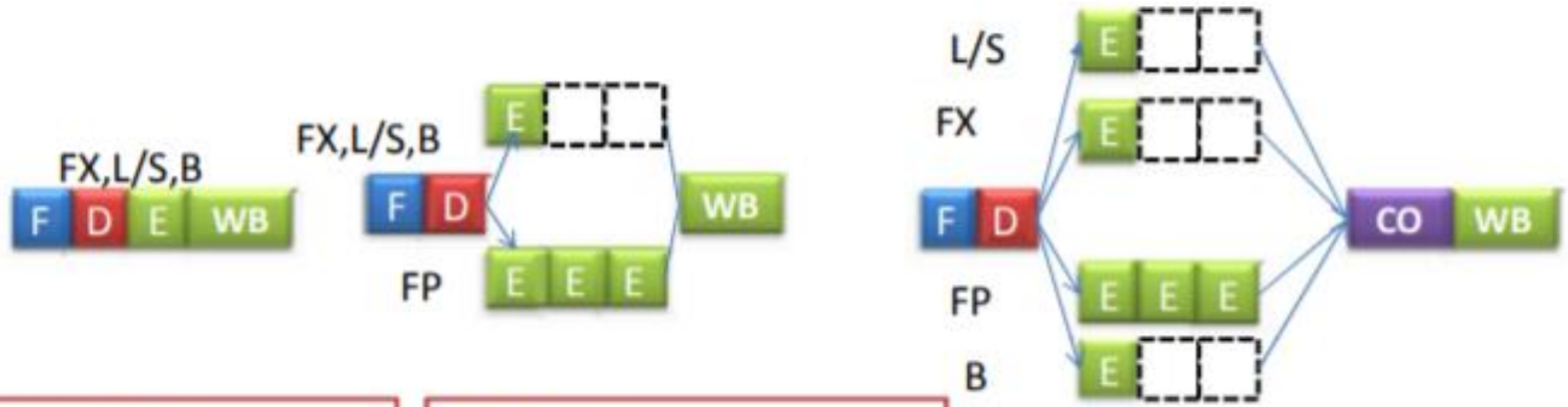
scalar processors and **superscalar processors**,

vector processors.

Parallel computers and computation.

1. Instruction level parallelism

Scheduling and Preserving Sequential Consistency



Early RISC Processors
Without an FP unit

RISC I, II (1983)
MIPS (1981)
IBM 801 (1978)

Scalar Processors
With an FP unit

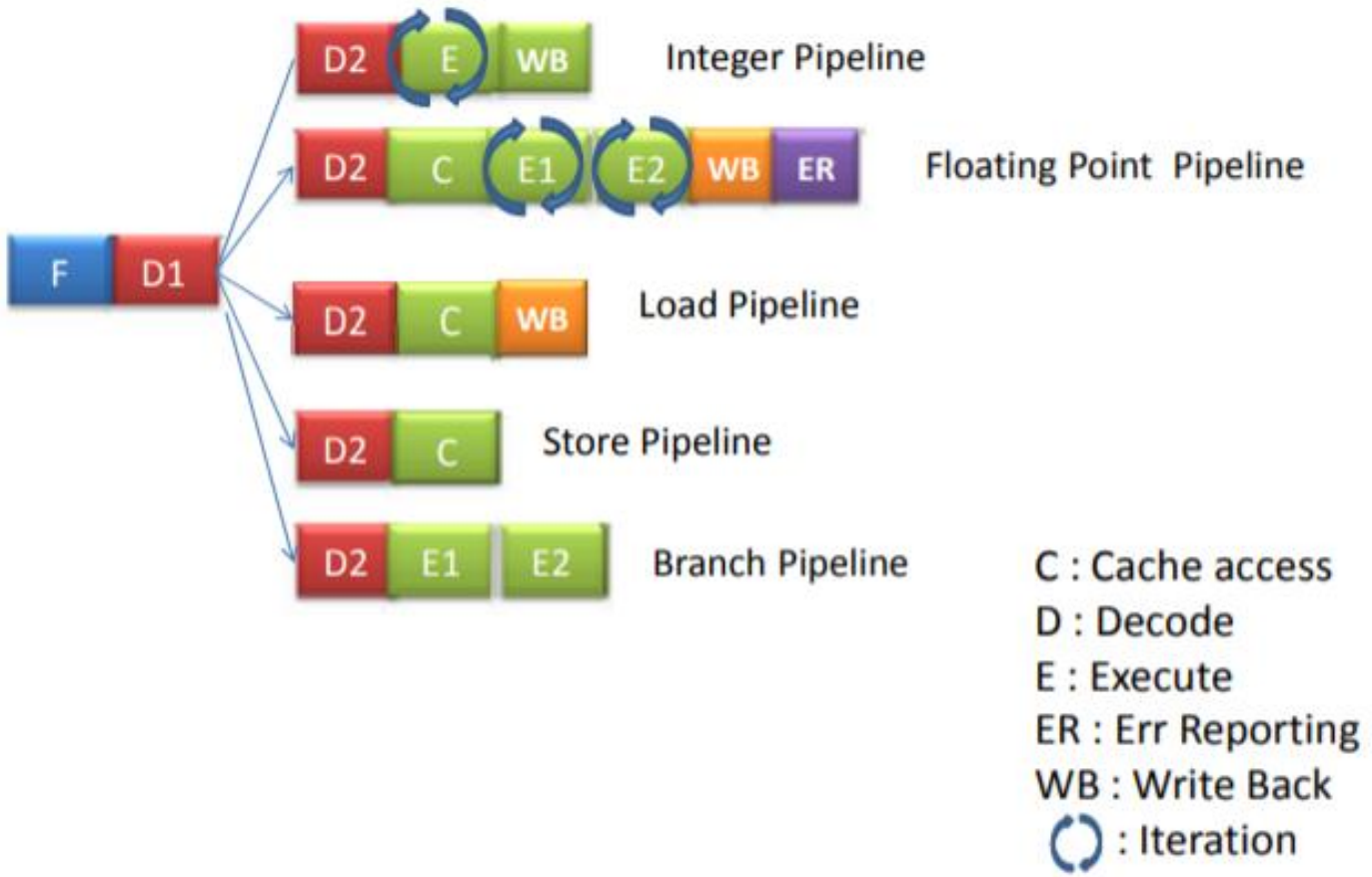
Pentium (1993)
I486 (1988)
MIPS R 2/3000 (1988)

Supercalar Processors

Pentium Pro(1993)
PowerPC 603 (1993)

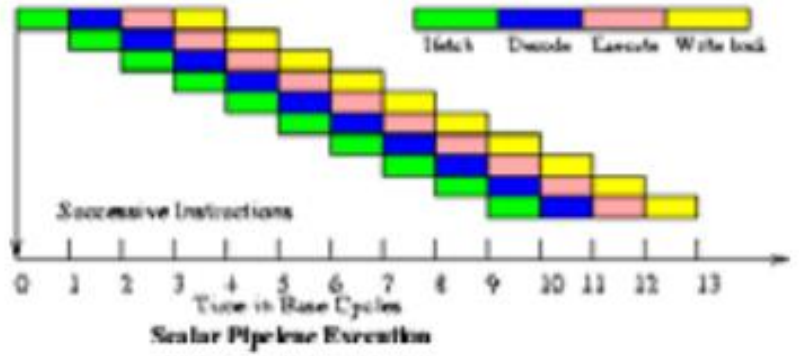
1. Instruction level parallelism

Case Study : Logical Layout of Pentium's Pipeline

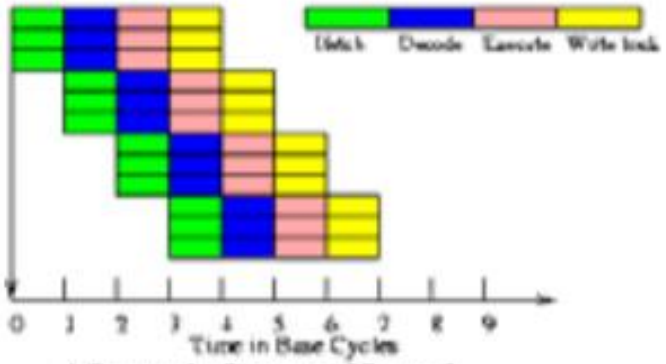


1. Instruction level parallelism

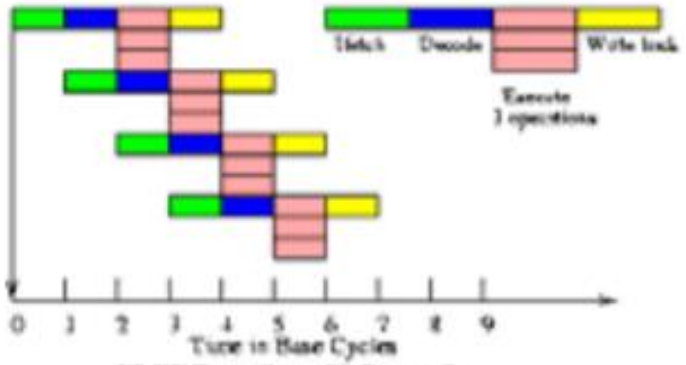
Superscalar vs VLIW



Scalar Pipeline Execution



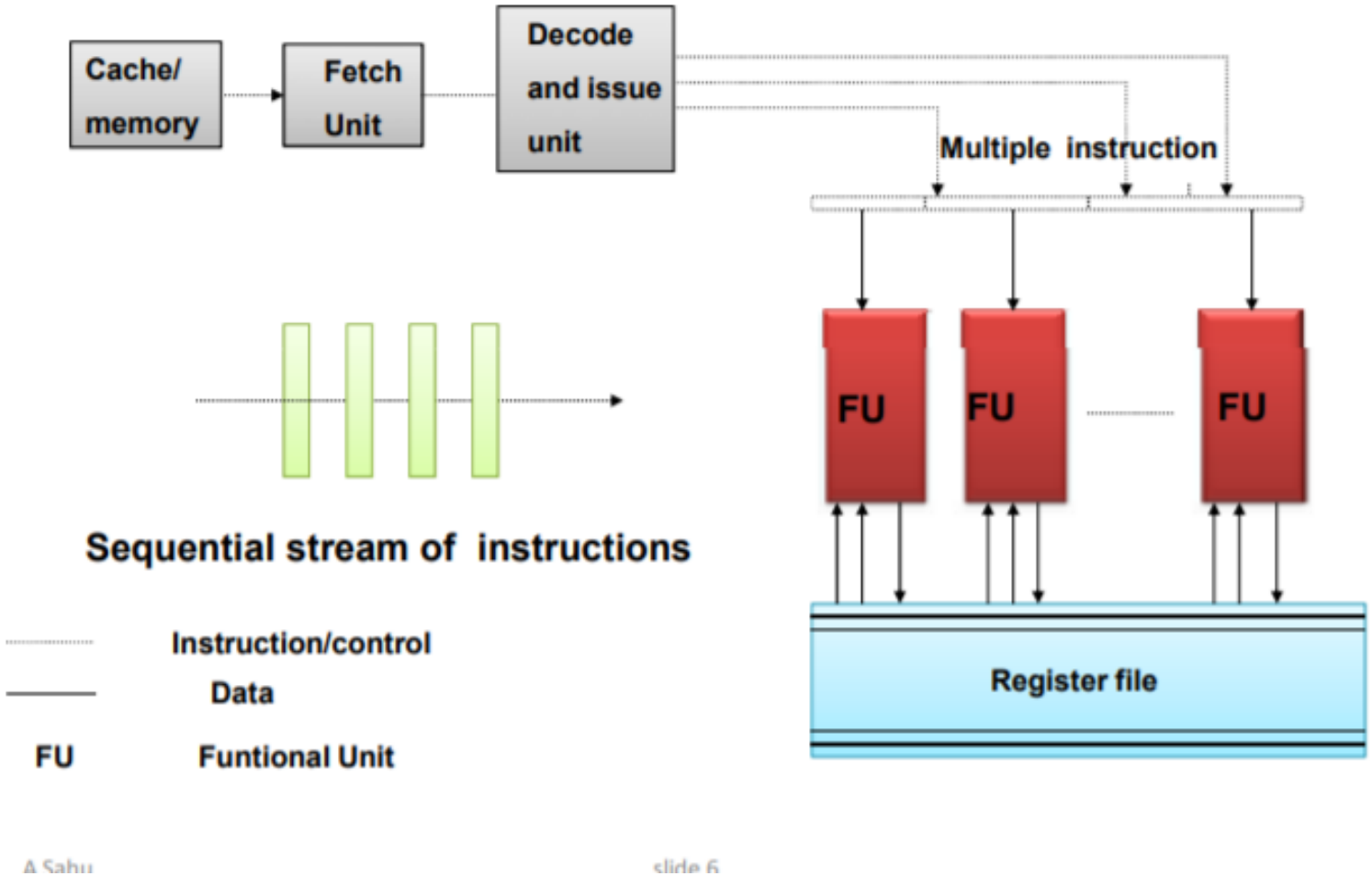
A Superscalar Processor of Degree 3



VLIW Execution with Degree 3

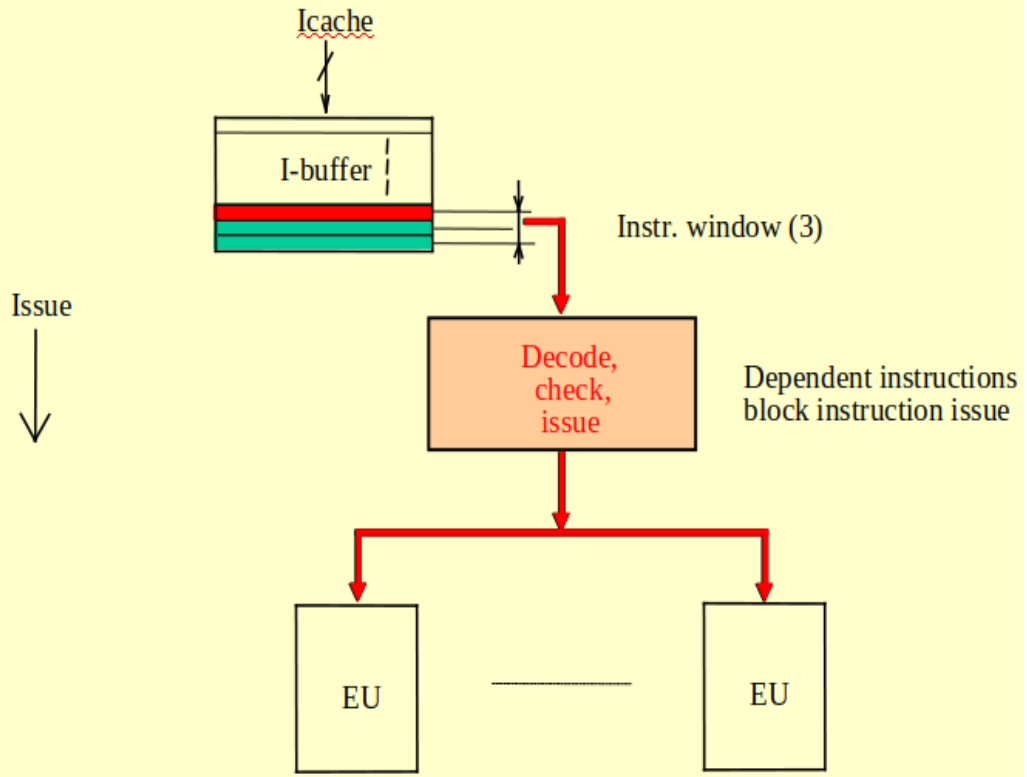
1. Instruction level parallelism

ILP in Superscalar processor

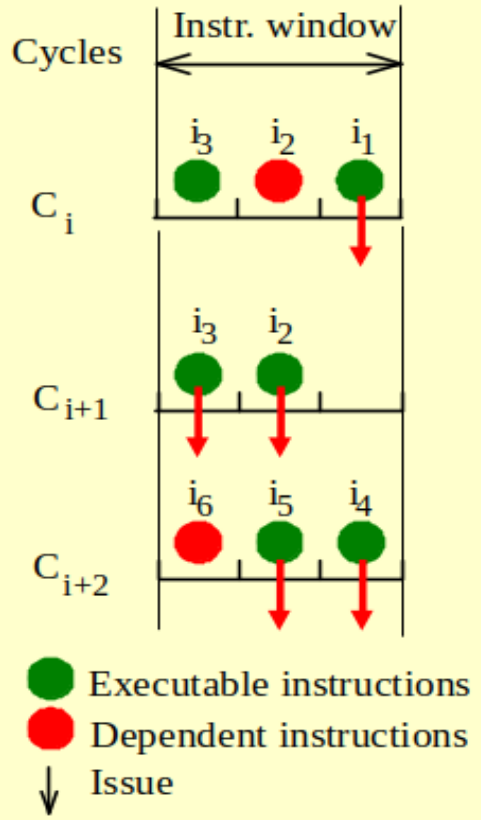


2: Data Hazard and Instruction Issue

The issue bottleneck



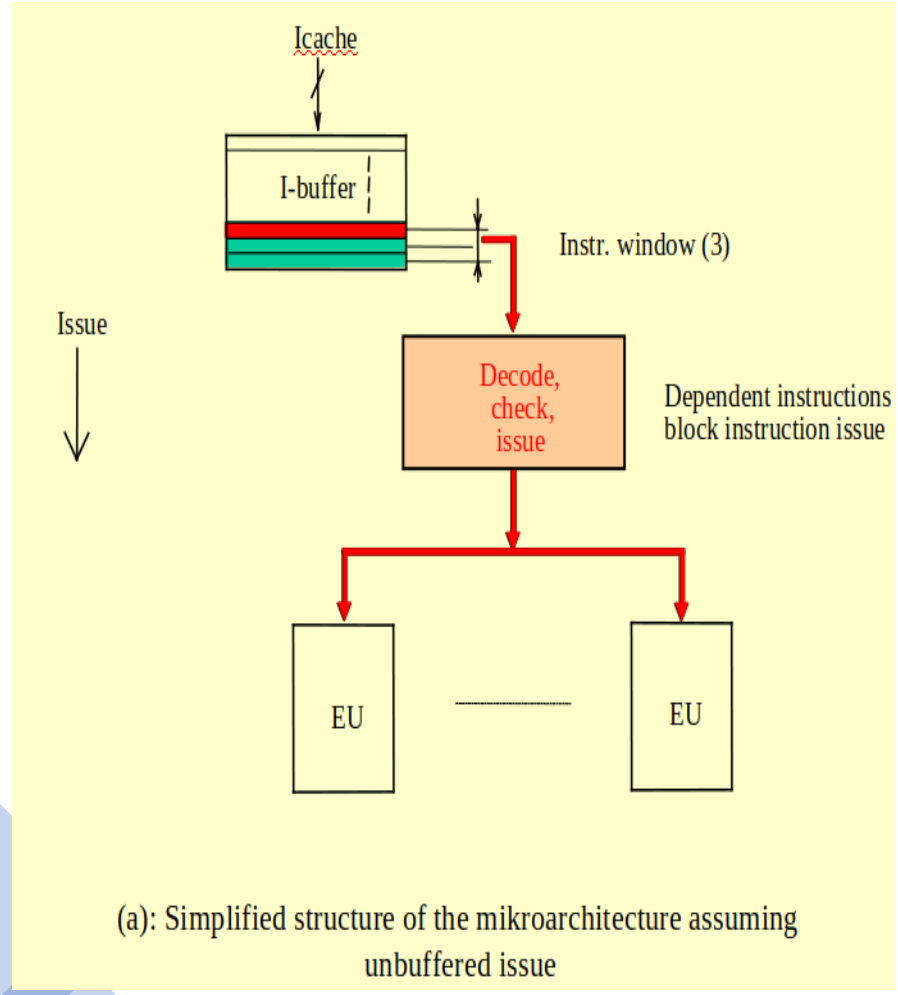
(a): Simplified structure of the mikroarchitecture assuming unbuffered issue



(b): The issue process

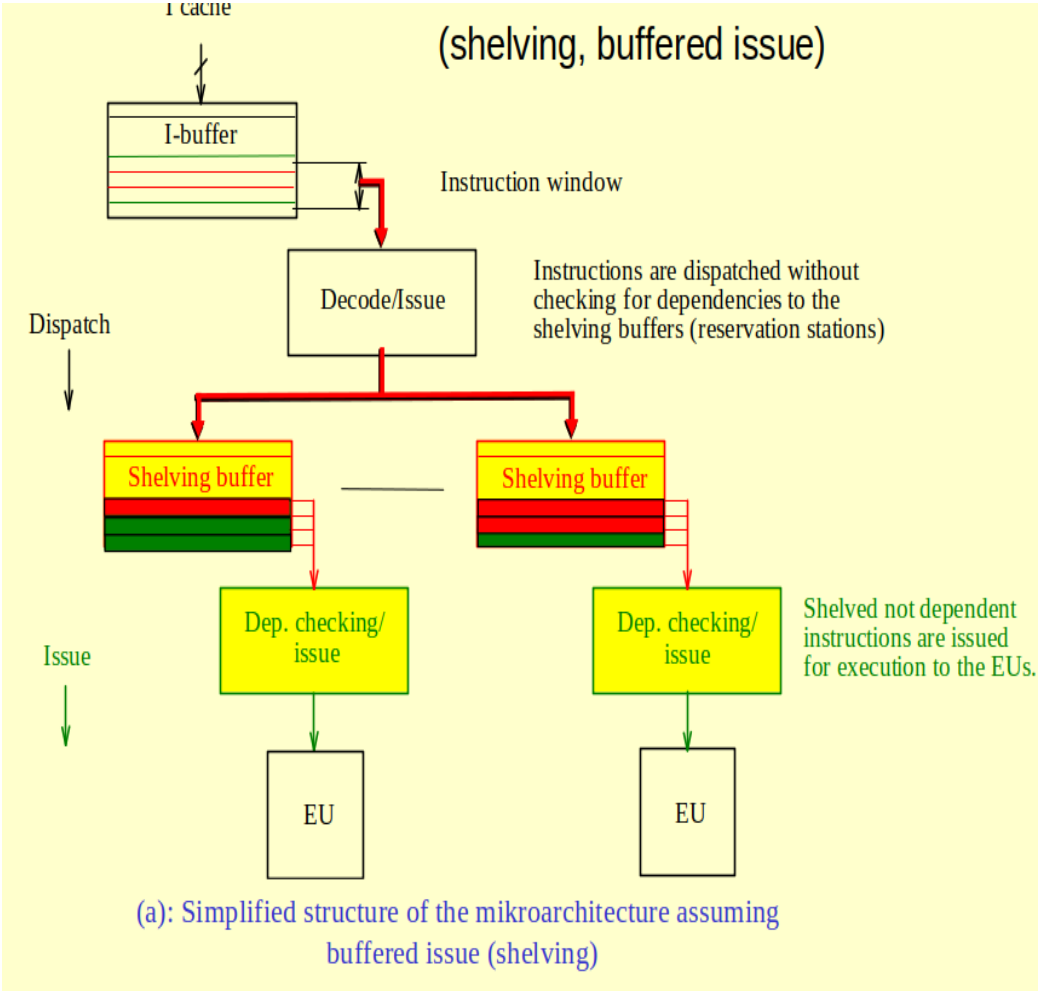
2: Data Hazard and Instruction Issue

The issue bottleneck



(a): Simplified structure of the mikroarchitecture assuming unbuffered issue

Dynamic instruction issue



(a): Simplified structure of the mikroarchitecture assuming buffered issue (shelving)

2: Data Hazard and Instruction Issue

- **Data Dependency (Data Hazard)**

Dependency of instructions due to operand values unavailability

- **Control Dependency (Control Hazard)**

Dependency due to unresolved decision on control instructions.

2: Data Hazard and Instruction Issue

- **Data Dependency (Data Hazard)**

Dependency of instructions due to operand values unavailability

- Read After Read (RAR) : No dependency
- Read After Write (RAW) : Common Hazard
- Write After Read (WAR) : Hazard in Out of order execution
- Write After Write (WAW) : Hazard in Out of order execution

2: Data Hazard and Instruction Issue

- **Data Dependency (Data Hazard)**

Dependency of instructions due to operand values unavailability

- **Read After Read (RAR) :** No dependency

  ADD R1, **R2**, R6 ; R1 <- **R2**+R6

  SUB R4, **R2**, R7; R4 <- **R2**+R7

2: Data Hazard and Instruction Issue

- **Data Dependency (Data Hazard)**

Dependency of instructions due to operand values unavailability

- **Read After Write (RAW) :** Common Hazard

ADD **R1**, R2, R6 ; R1 <- R2+R6

SUB R4, **R1**, R7; R4 <- R1+R7

Solve Using Data Forwarding

2: Data Hazard and Instruction Issue

- **Data Dependency (Data Hazard)**

Dependency of instructions due to operand values unavailability

- **Write After Read (WAR) :** Not Common in in-order pipeline

  ADD R1, **R2**, R3 ; R1 <- R2+R3

  SUB **R2**, R4, R5; R2 <- R4+R5

For Out of Order execution, Use dependency check for execution

2: Data Hazard and Instruction Issue

- **Data Dependency (Data Hazard)**

Dependency of instructions due to operand values unavailability

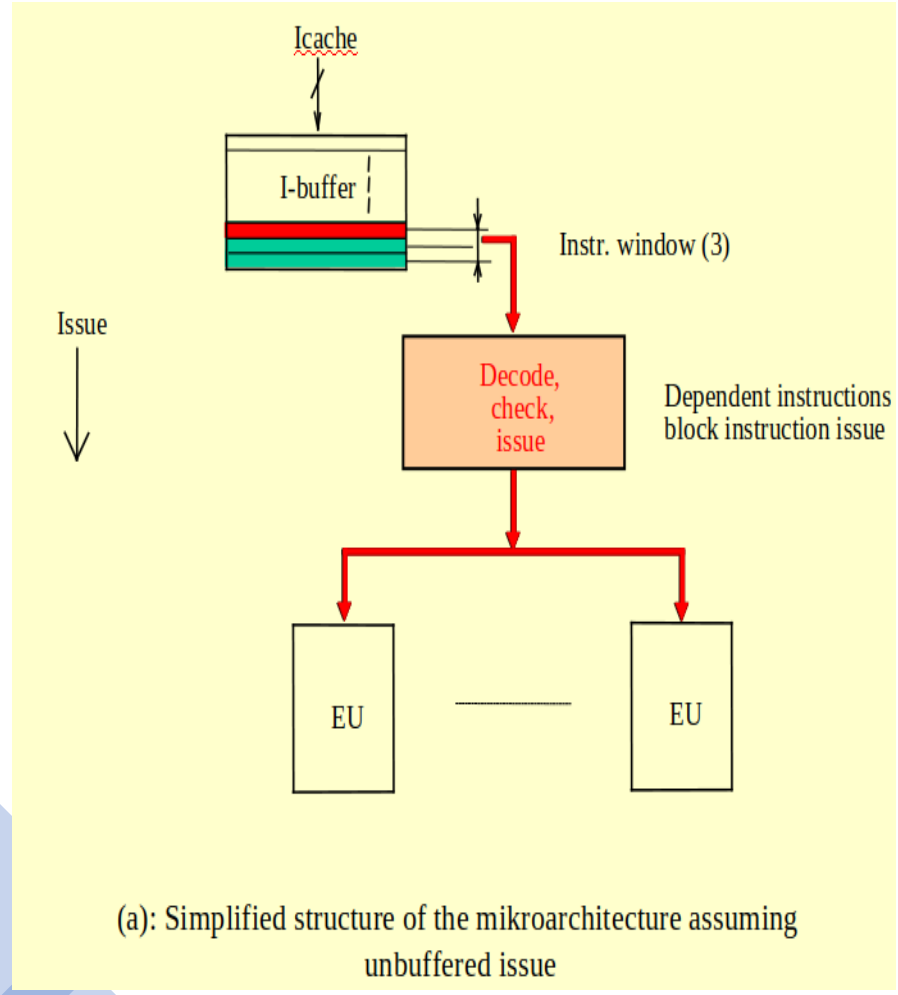
- **Write After Write (WAW) :** Not hazard in in-order pipeline

ADD **R1**, R2, R3 ; R1 <- R2+R3

SUB **R1**, R4, R5; R1 <- R4+R5

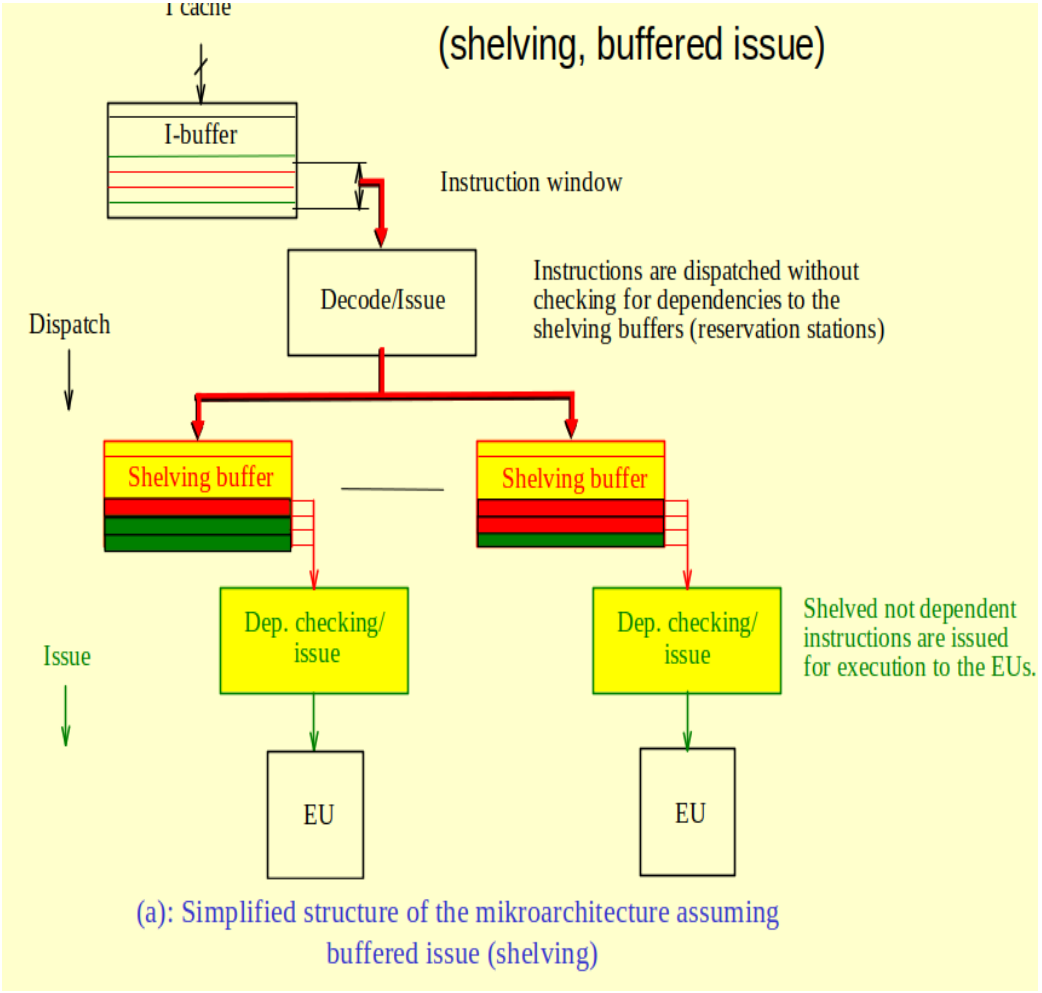
2: Data Hazard and Instruction Issue

The issue bottleneck



(a): Simplified structure of the mikroarchitecture assuming unbuffered issue

Dynamic instruction issue



(a): Simplified structure of the mikroarchitecture assuming buffered issue (shelving)

3. Inorder Vs Out of order execution

- **In-order instruction execution**
  - Instructions are fetched, executed and completed in compiler generated order
  - One instruction stall, stalls all other instructions
  - Instructions are Statistically Scheduled

- **Out-of-order instruction execution**
  - Instructions are fetched in compiler-generated order
  - Instruction completion may be in-order or out-of-order
  - Independent instruction behind a stalled instruction can pass it
  - Instructions are Dynamically executed

https://courses.cs.washington.edu/courses/csep548/06au/lectures/introOOO.pdf

3. Inorder Vs Out of order execution

- **Out-of-order procesors**
  - After instruction decode
    - Check for structural hazards
      - An instruction can be issued when a functional unit is available
      - An instruction stalls if no appropriate functional unit
    - Check for Data Hazards
      - An instruction can execute when its operands have been calculated or loaded from memory
      - An instruction stalls if operands are not available.

3. Inorder Vs Out of order execution

- **Out-of-order procesors**
  - After instruction decode
    - Independent ready instructions can execute before earlier instructions that are stalled

in-order processors

```
lw $3, 100($4)      in execution, cache miss
add $2, $3, $4      waits until the miss is satisfied
sub $5, $6, $7      waits for the add
```

out-of-order processors

```
lw $3, 100($4)      in execution, cache miss
sub $5, $6, $7      can execute during the cache miss
add $2, $3, $4      waits until the miss is satisfied
```

3. Inorder Vs Out of order execution

- **Out-of-order procesors**
  - After instruction decode
    - Independent ready instructions can execute before earlier instructions that are stalled
      - When path instructions are waiting for a branch condition to be computed
        - The instructions that are issued from the predicted path are issued speculatively, called speculative execution.
        - Speculative isntructions are execute (but not commit) before the branch is resolved
        - If the prediction was wrong, speculative instructions are flushed from the pipeline
        - If prediction is right, instructions are no longer speculative.

3. Inorder Vs Out of order execution

- **Speculative Execution**
  - Executing an instruction before it is known that it should be executed
    - All instructions that are fetched because of a prediction are speculative
    - Inorder pipeline : branch is executed before the path
    - Out of order pipeline
      - Path can be executred before the branch
      - Speculative instructions can execute but not committed
      - Getting rid of wrong path instructions is not just a matter of flushing them from the pipeline.

Thank You

Reference

Textbooks and/or Reference Books:

1. Professional CUDA C Programming – John Cheng, Max Grossman, Ty McKercher, 2014
2. Wilkinson, M. Allen, ”Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers”, Pearson Education, 1999
3. I. Foster, ”Designing and building parallel programs”, 2003
4. Parallel Programming in C using OpenMP and MPI – Micheal J Quinn, 2004
5. Introduction to Parallel Programming – Peter S Pacheco, Morgan Kaufmann Publishers, 2011
6. Advanced Computer Architectures: A design approach, Dezso Sima, Terence Fountain, Peter Kacsuk, 2002
7. Parallel Computer Architecture : A hardware/Software Approach, David E Culler, Jaswinder Pal Singh Anoop Gupta, 2011 8. Introduction to Parallel Computing, Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar, Pearson, 2011

National Institute of Technology Karnataka Surathkal
Department of Information Technology

IT 301 Parallel Computing

Superscalar Processor 2

Dr. Geetha V

Assistant Professor

Dept of Information Technology

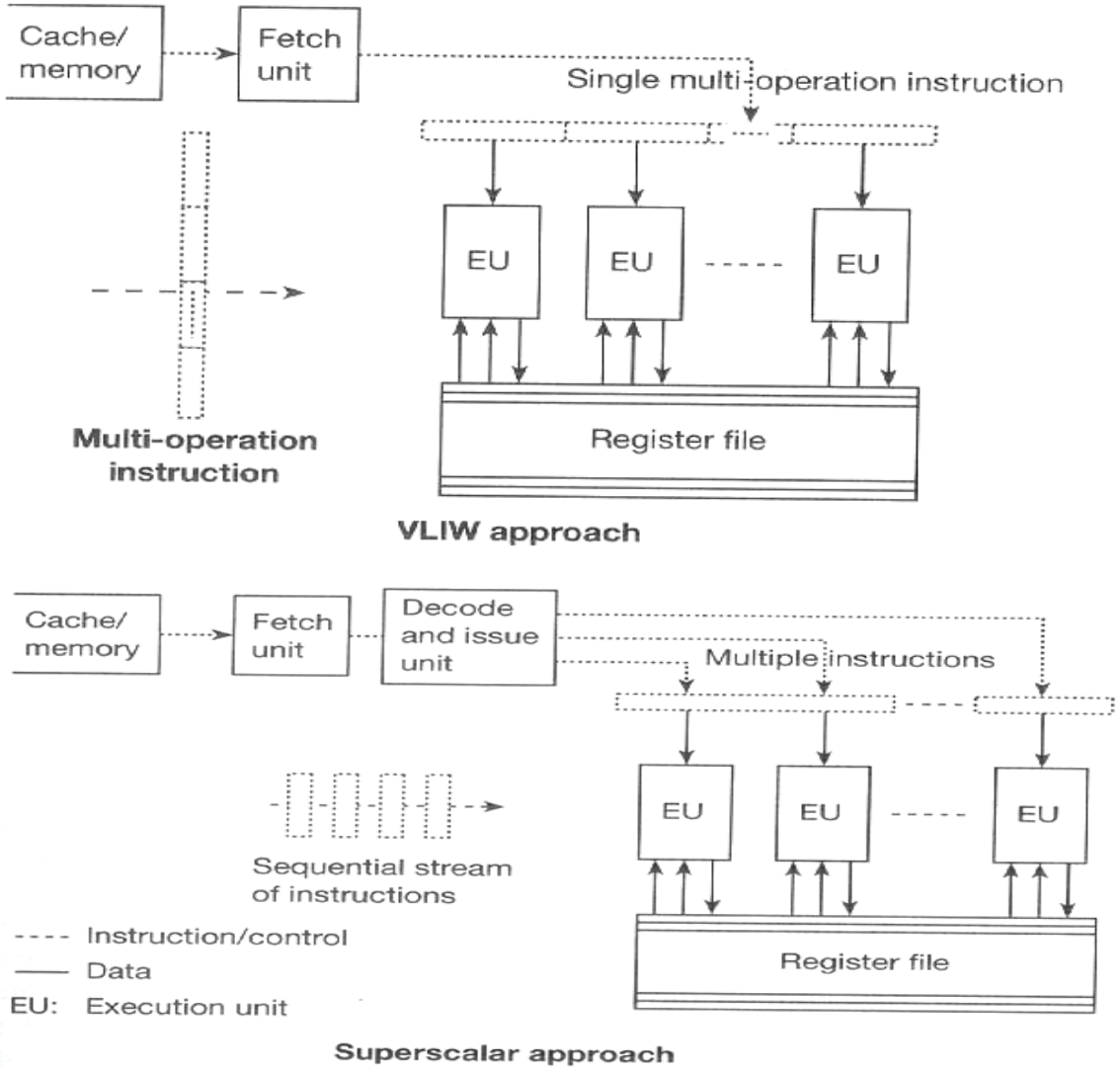
NITK Surathkal

Acknowledgment to slides :
www2.latech.edu/~choi/Bens/Teaching/Csc521/

Superscalar Processors

- 7.1 Introduction
- 7.2 Parallel decoding
- 7.3 Superscalar instruction issue
- 7.4 Shelving -Example
- 7.5 Register renaming
- 7.6 Parallel execution
- 7.7 Preserving the sequential consistency of instruction execution
- 7.8 Preserving the sequential consistency of exception processing
- 7.9 Implementation of superscalar CISC processors using a superscalar RISC core
- 7.10 Case studies of superscalar processors

7.1 Superscalar Processors vs. VLIW

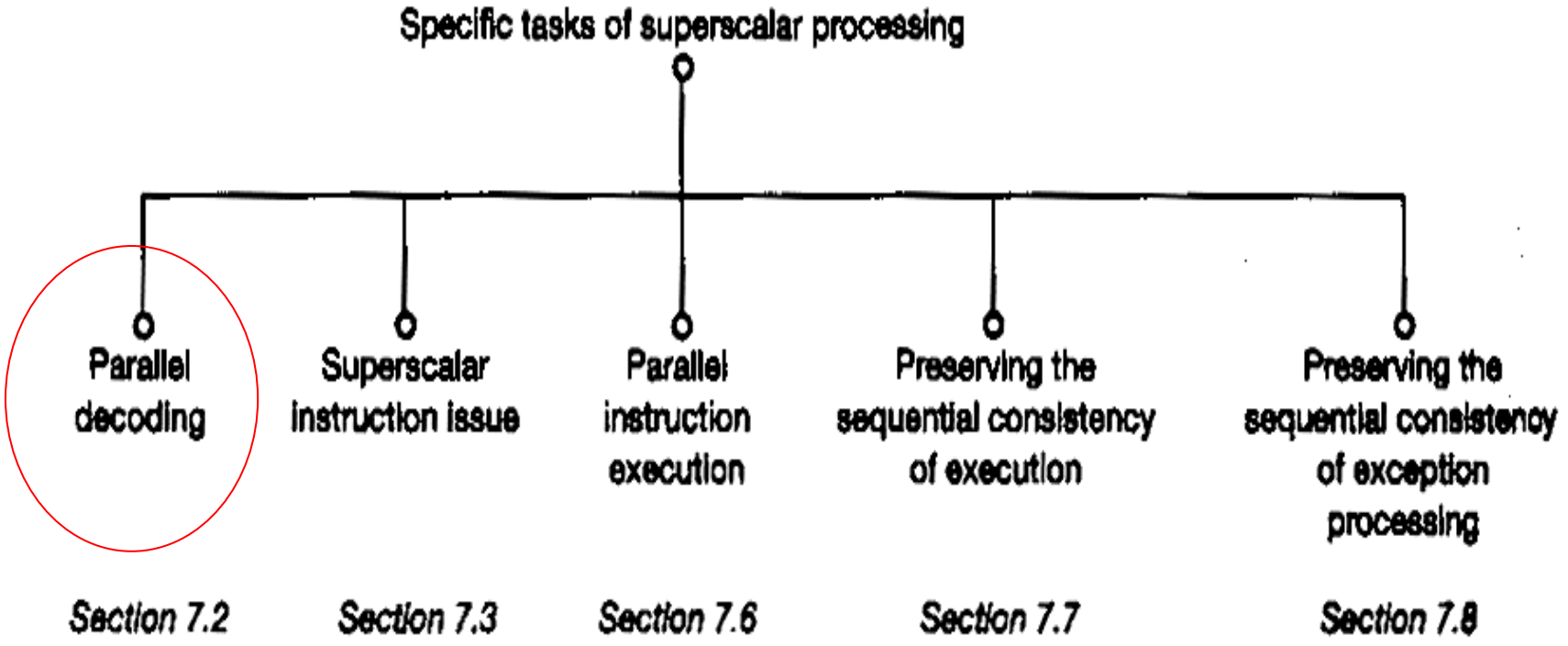


- **VLIW architecture** : (Very large Instruction Word)
- One single word contains many instructions, without any data dependency
- **Superscalar approach** uses dynamic instruction execution with dynamic instruction scheduling
- Multiple instructions are fetched and dependencies are verified dynamically before issuing instruction for execution.

7.1 Superscalar Processor: Intro

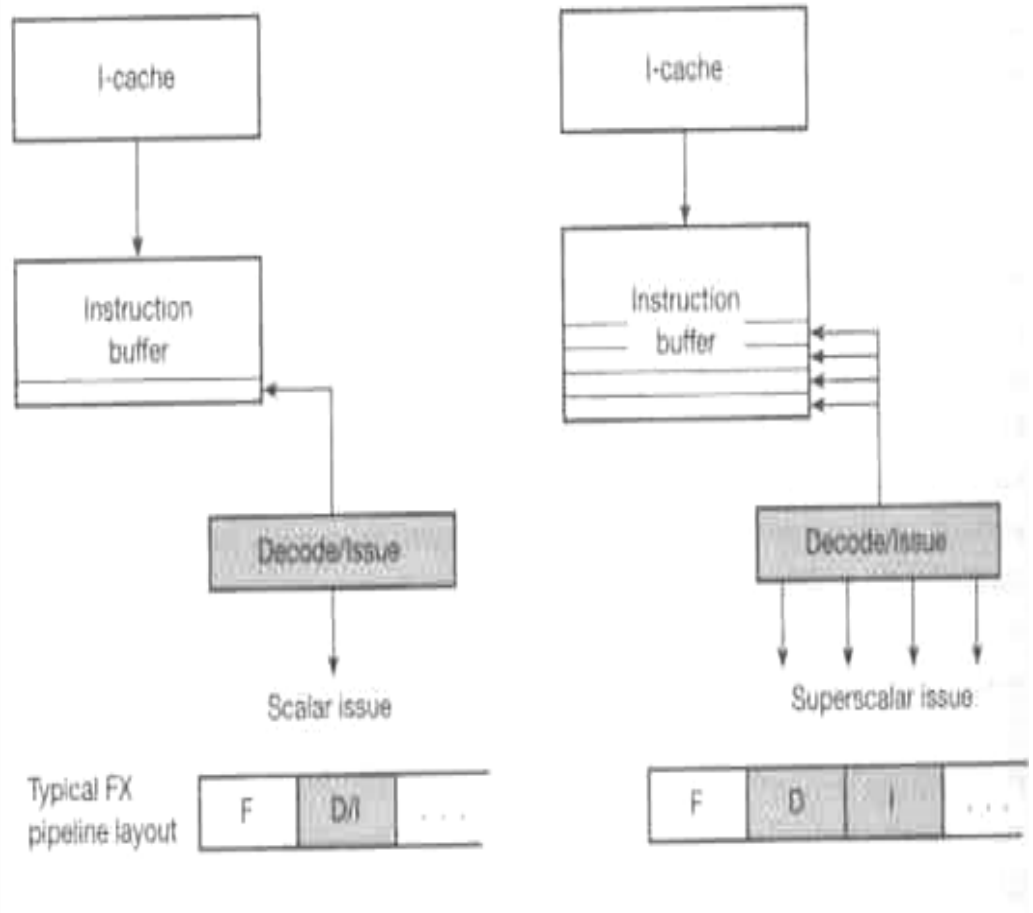
- Parallel Issue
- Parallel Execution
- {Hardware} Dynamic Instruction Scheduling
- Currently the predominant class of processors
  - **Pentium**
  - **PowerPC**
  - **UltraSparc**
  - **AMD K5-**
  - **HP PA7100-**
  - **DEC α**

7.1 Specific tasks of superscalar processing



7.2 Parallel decoding {and Dependencies check}

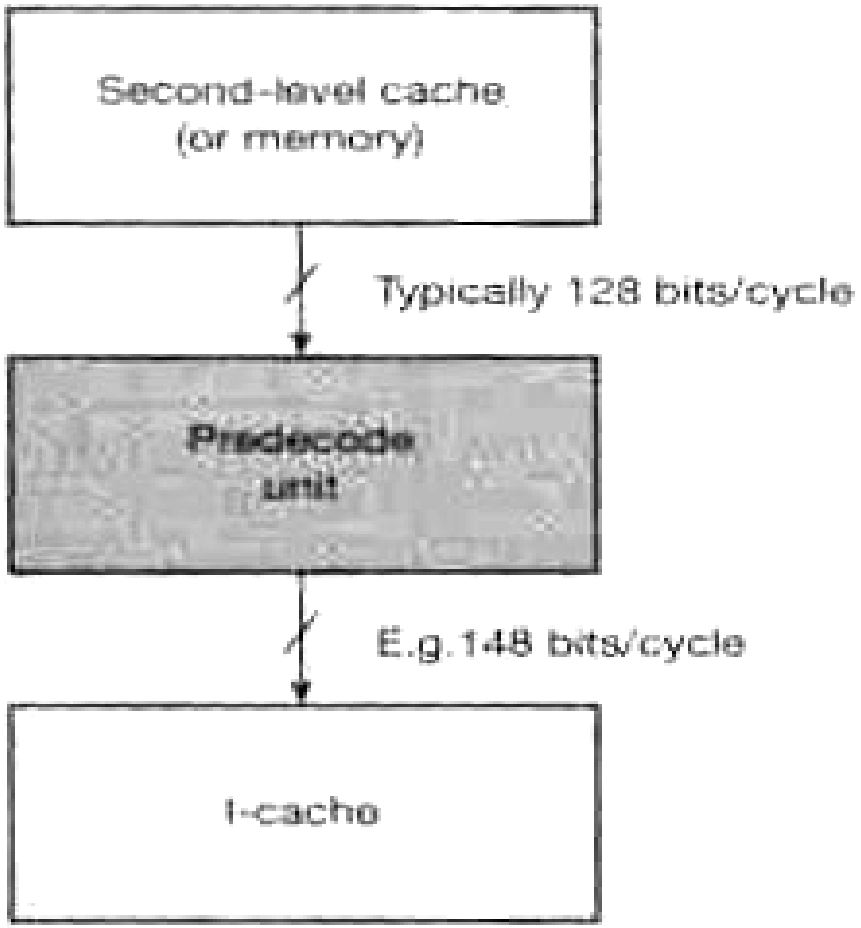
- In scalar processor the decode and instruction issue takes place in same stage.
- In superscalar processor, more than one instruction will be fetched and kept in Instruction Buffer for dependency check.
- More than one instruction is decoded and issued in superscalar processor at the same time.



7.2 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 (data transmission, logical, arithmetic, control, memory operations etc)
    - the type of resources required for the execution (FX /FP load/store etc)
    - in some processor (e.g. UltraSparc), branch target addresses calculation as well (advanced branch address calculation for branch taken and not taken path)
  - **the results are stored by attaching 4-7 bits**
- + shortens the overall cycle time or reduces the number of cycles needed

7.2 The principle of perdecoding

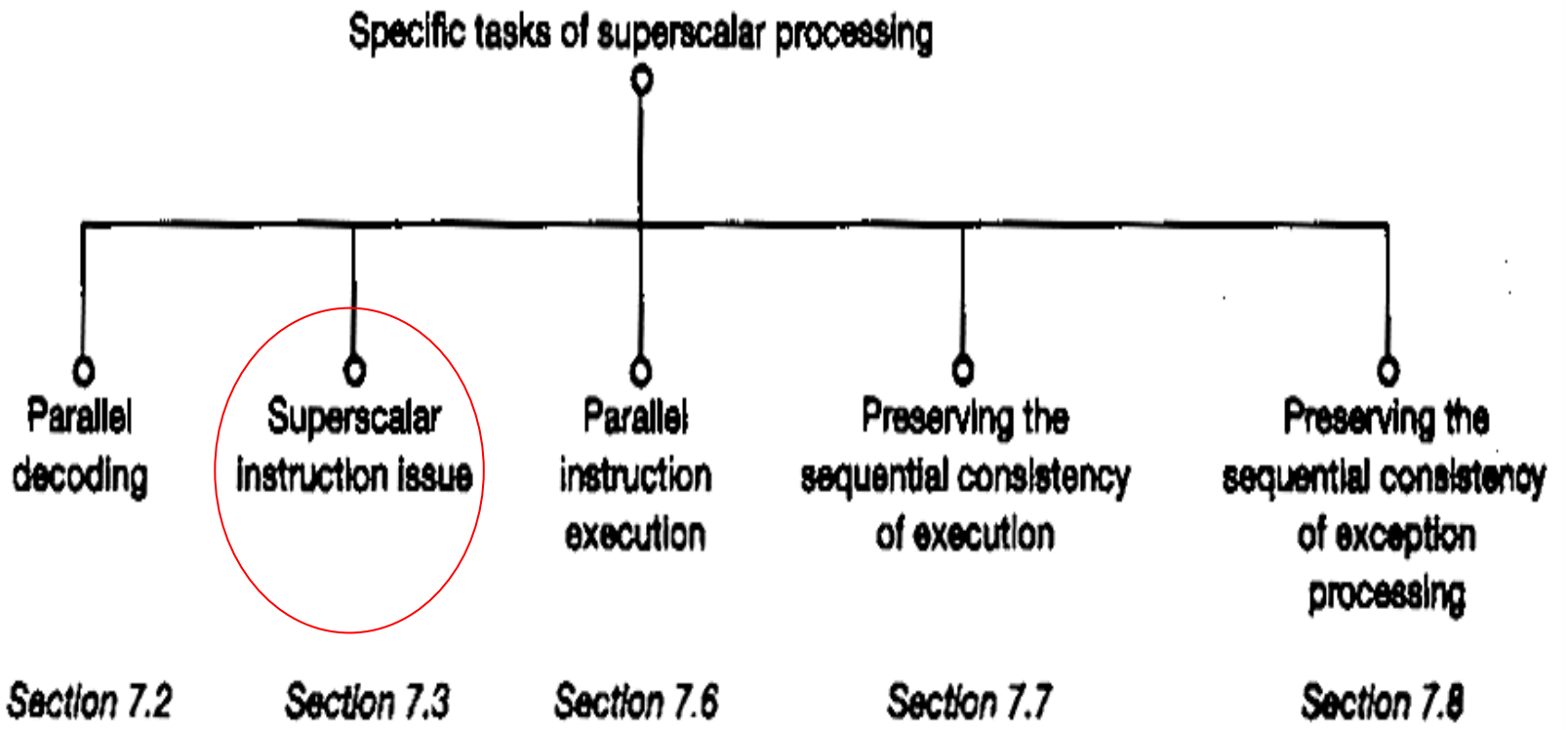


- In precode stage identify class of instruction, calculate branch target address, resources required for execution etc.
- 4/5/7 bits are attached in precode stage

When instructions are written into the I-cache, the predecode unit appends 4–7 bits to each RISC instruction [1]

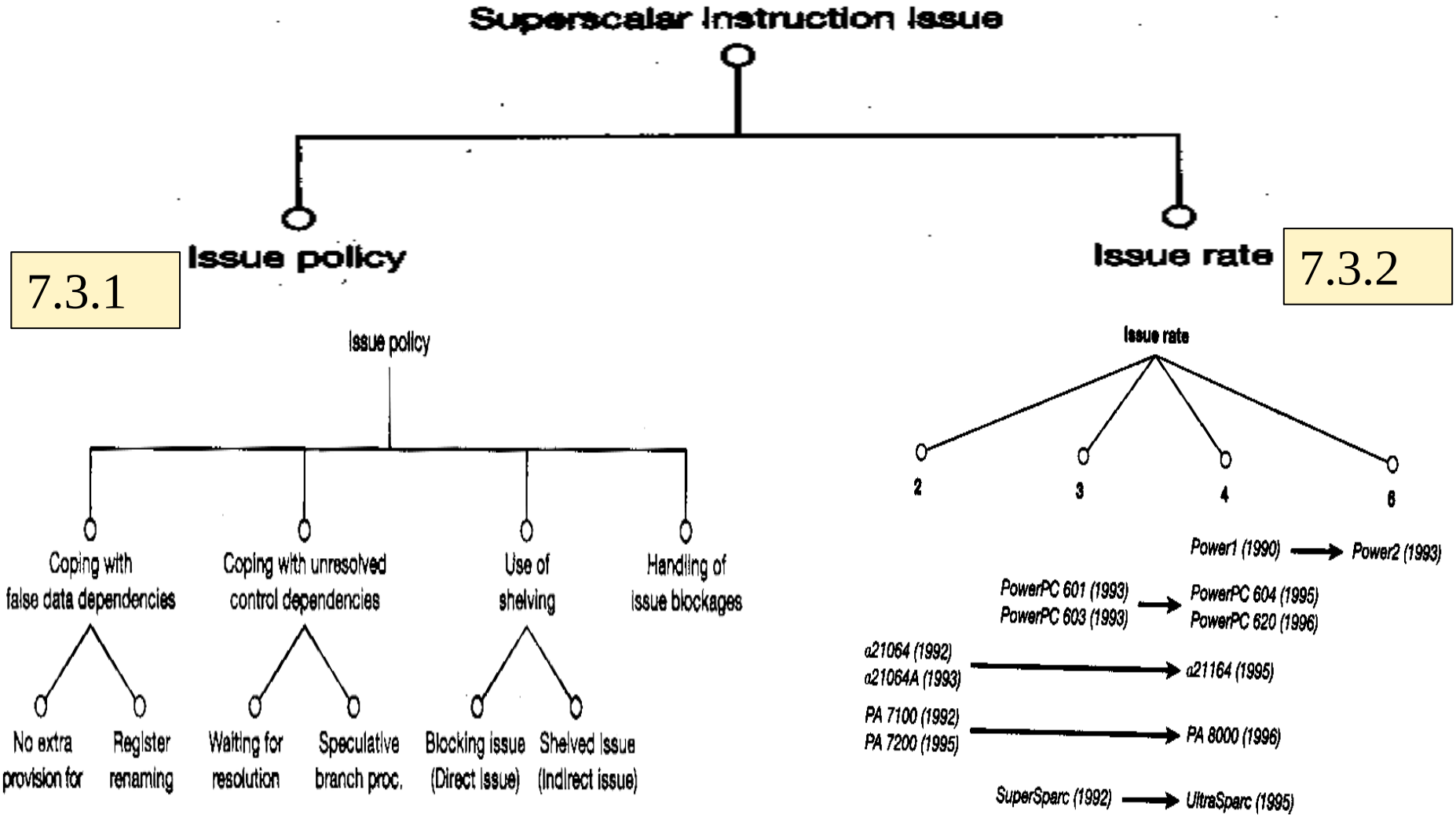
[1] In the AMD K5, which is an x86-compatible CISC processor, the predecode unit appends 5 bits to each byte

7.3 Specific tasks of superscalar processing: Issue

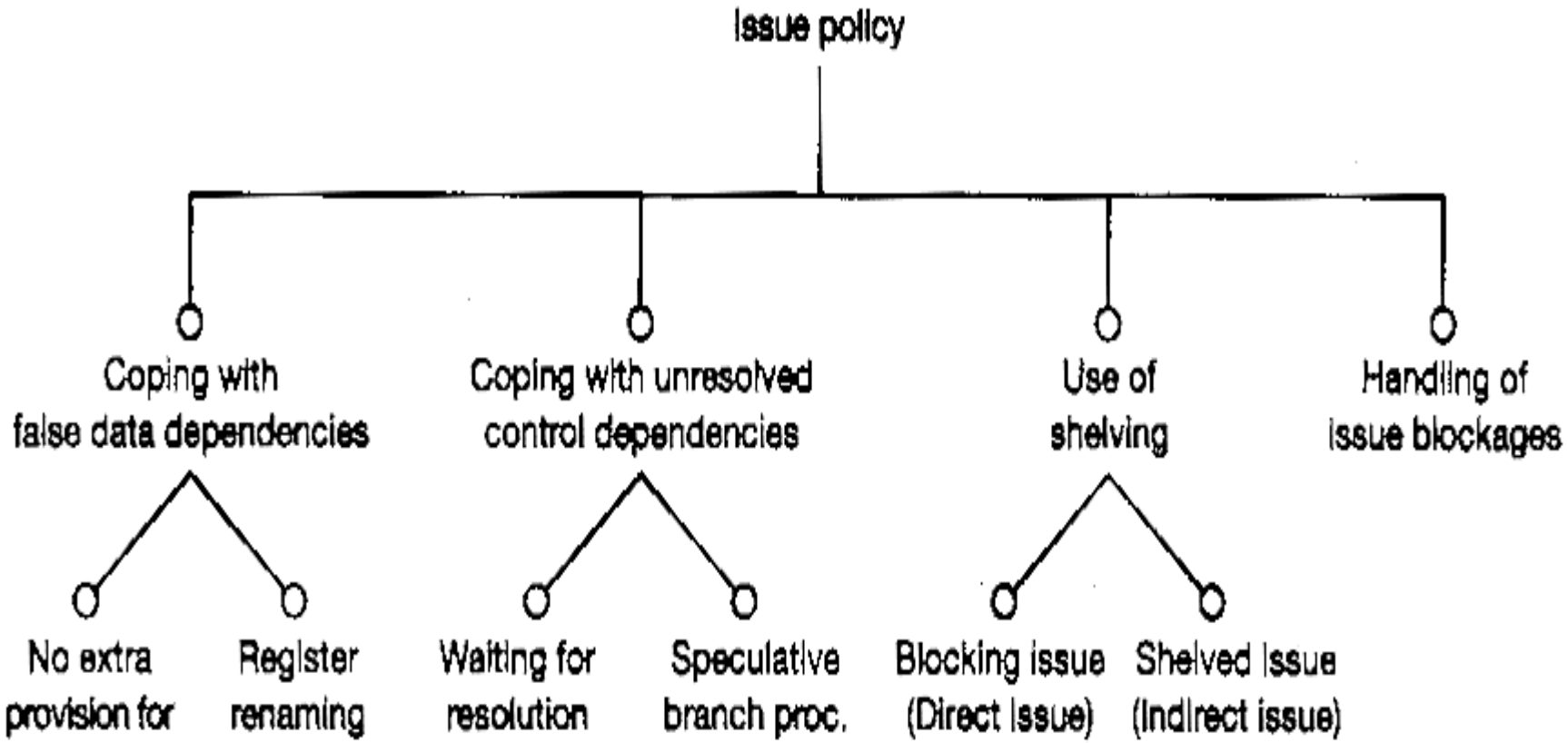


7.3 Superscalar instruction issue

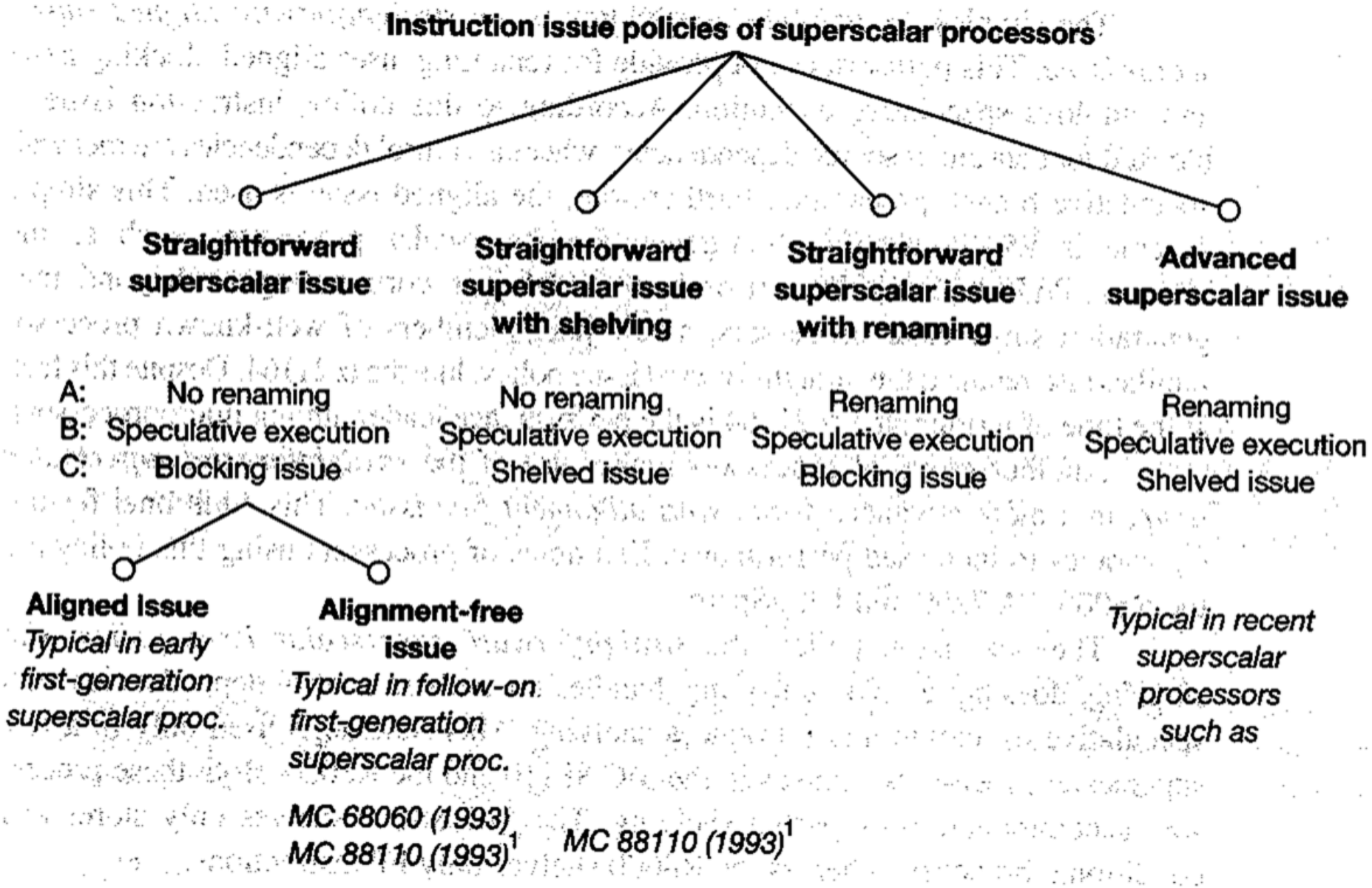
- How and when to send the instruction(s) to EU(s)



7.3.1 Issue policies

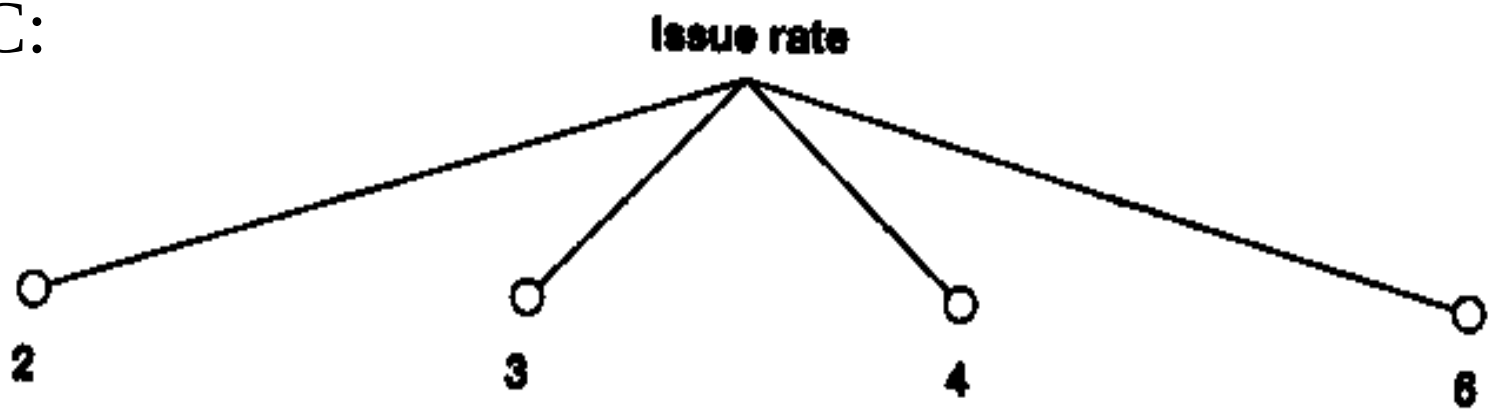


7.3.1 Instruction issue policies of superscalar processors: ---Performance, tread----->



7.3.2 Issue rate {How many instructions/cycle}

- CISC about 2
- RISC:



Power1 (1990) → Power2 (1993)

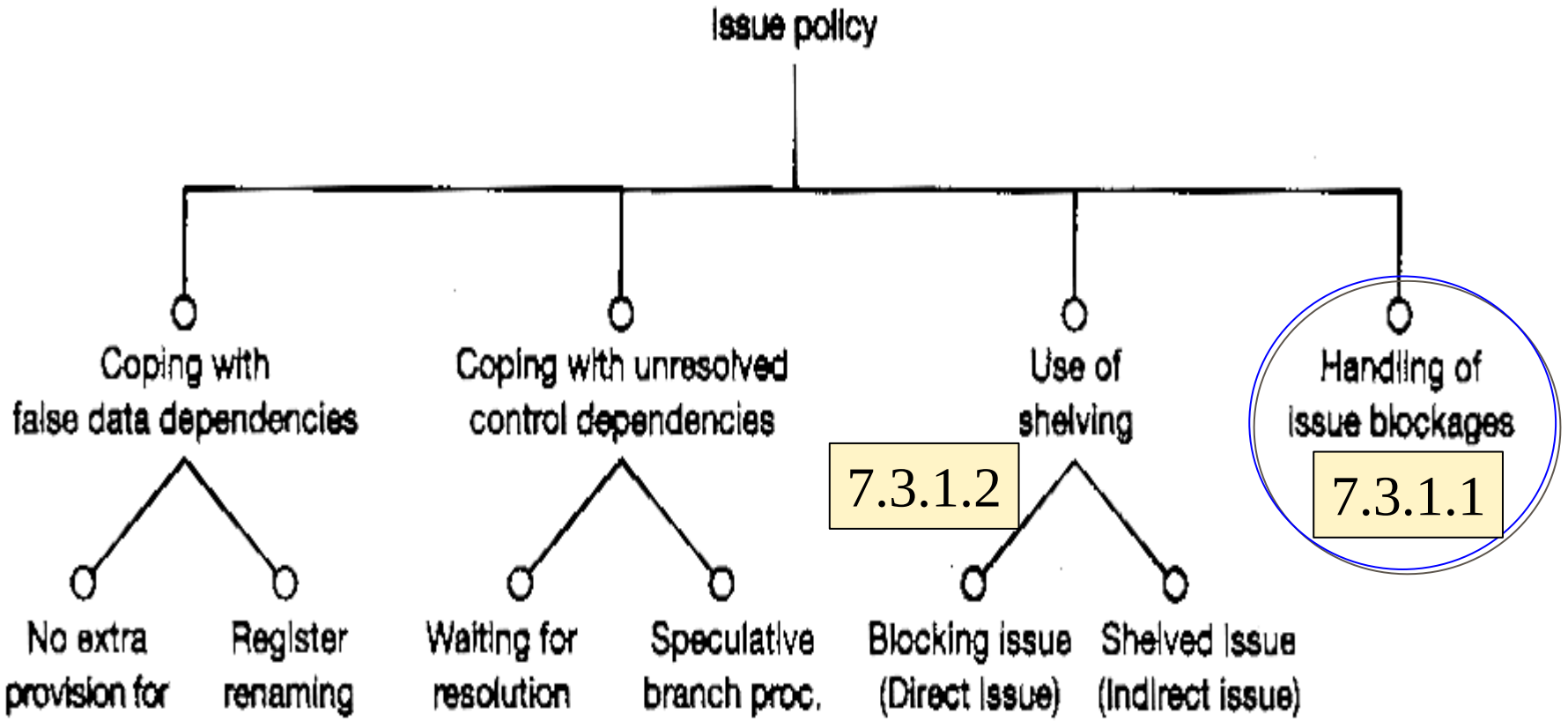
PowerPC 601 (1993)
PowerPC 603 (1993) → PowerPC 604 (1995)
PowerPC 620 (1996)

a21064 (1992)
a21064A (1993) → a21164 (1995)

PA 7100 (1992)
PA 7200 (1995) → PA 8000 (1996)

SuperSparc (1992) → UltraSparc (1995)

7.3.1 Issue policies: Handing Issue Blockages

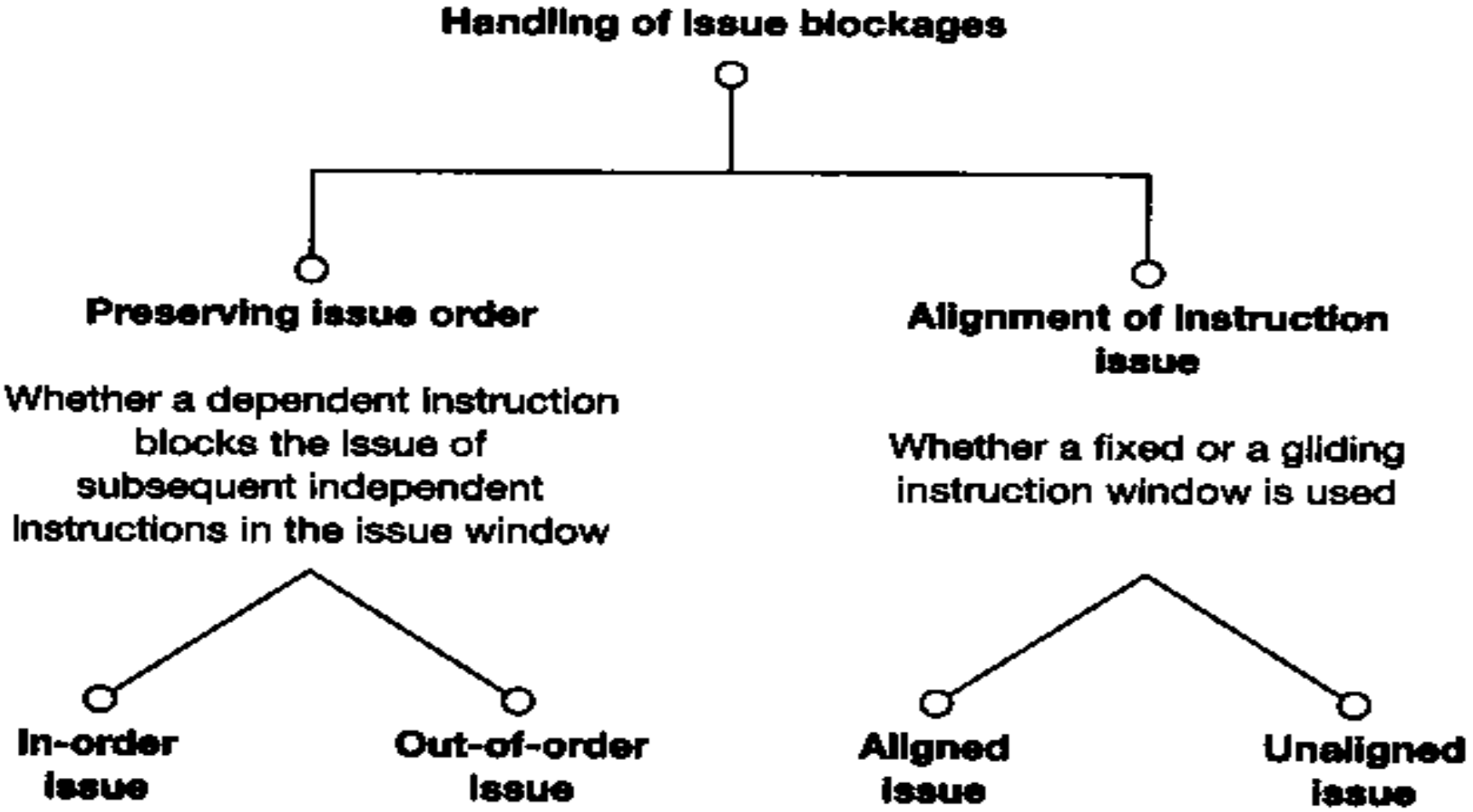


Resolving RAW and WAR hazards using register renaming

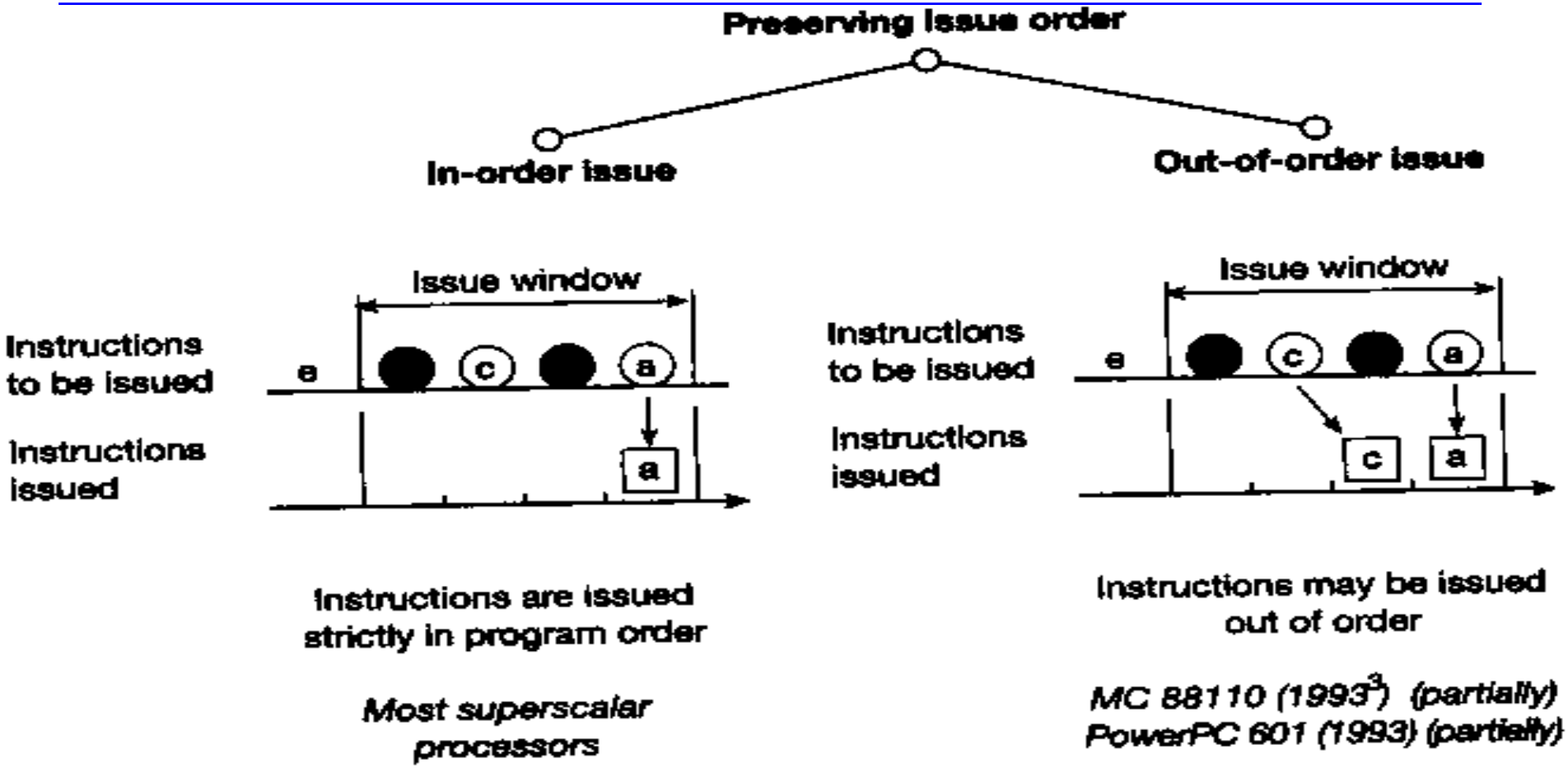
Speculating as branch taken or not taken based on history bits

7.3.1.1 Issue stopped by True dependency

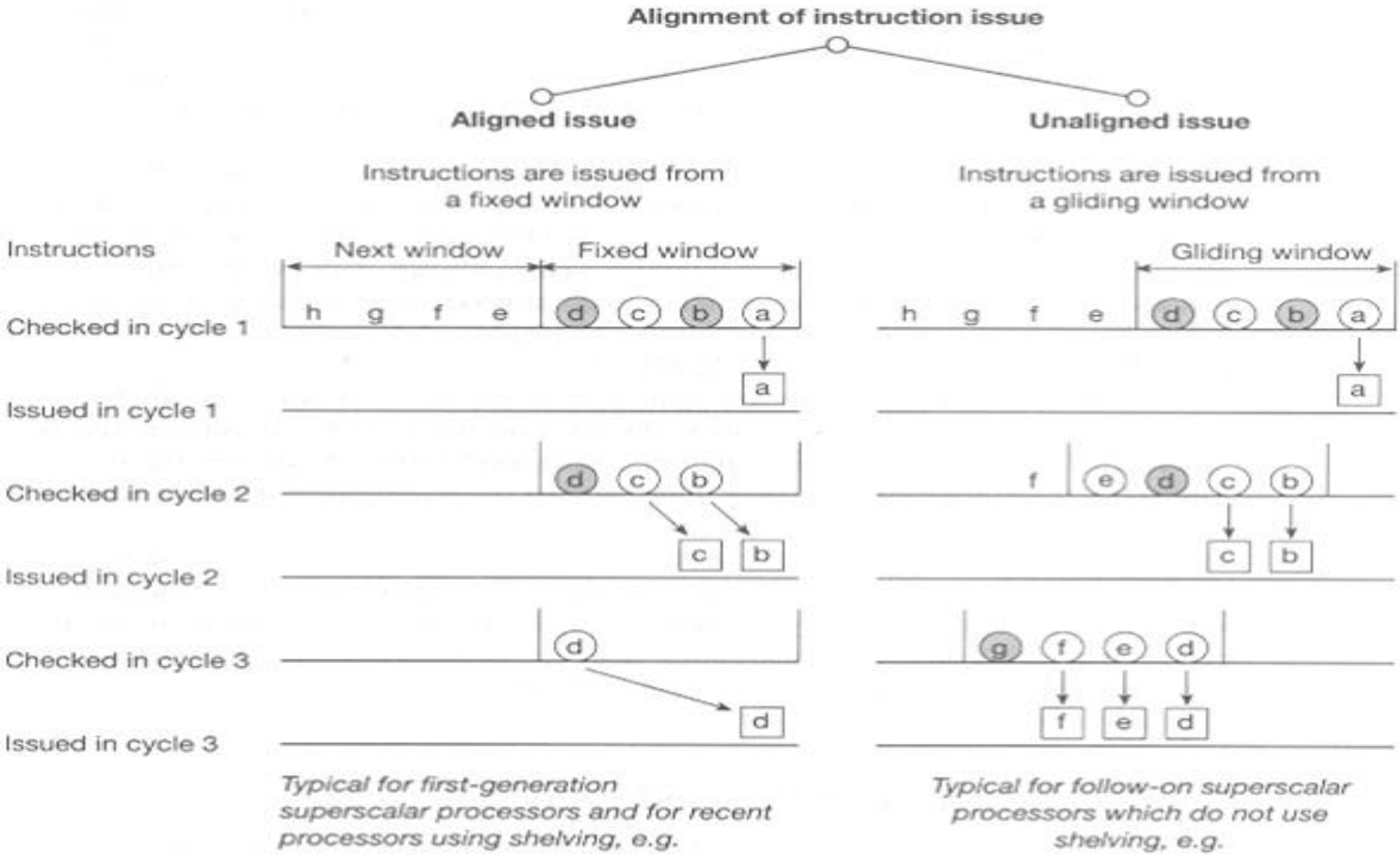
- True dependency → (Blocked: need to wait)



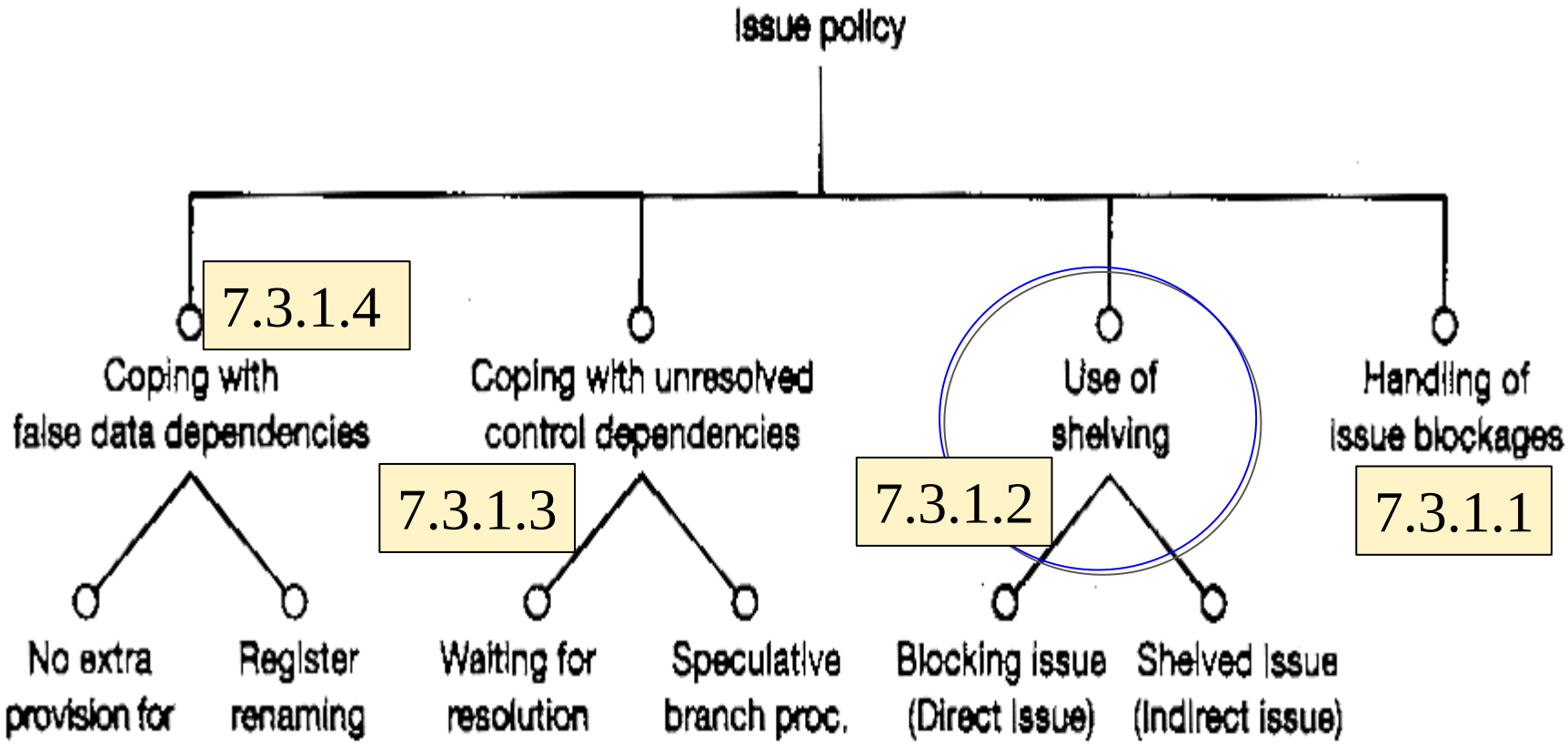
7.3.1.1. Issue order of instructions



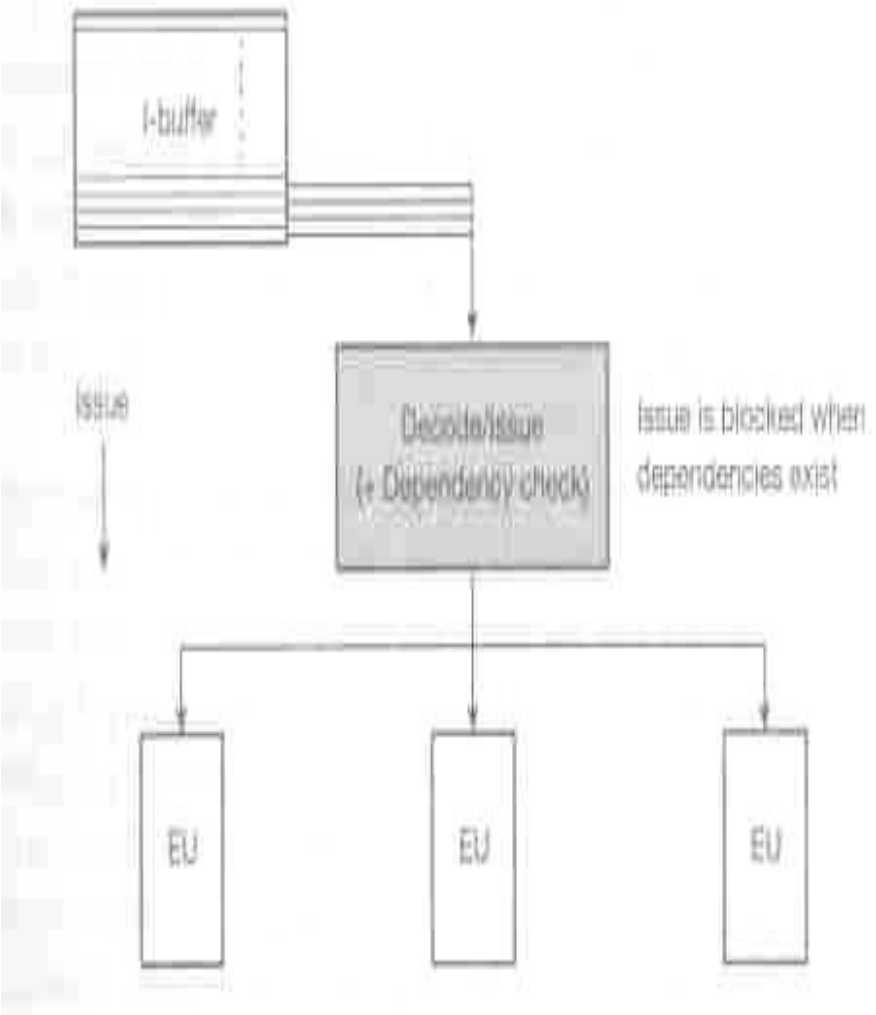
7.3.1.1 Aligned vs. unaligned issue



7.3.1 Issue policies: Handing Issue Blockages

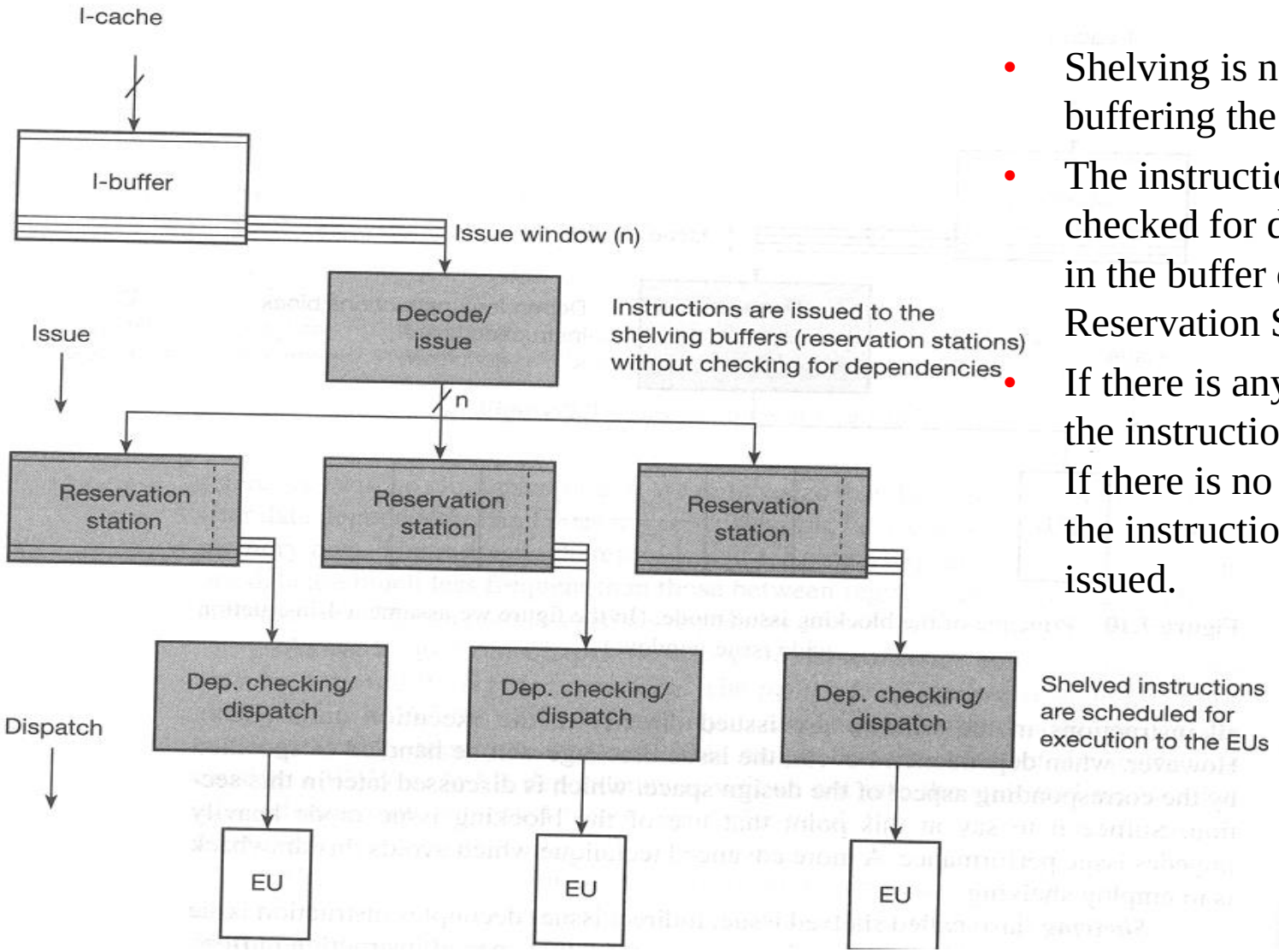


7.3.1.2 Blocking Issue (Direct Issue)



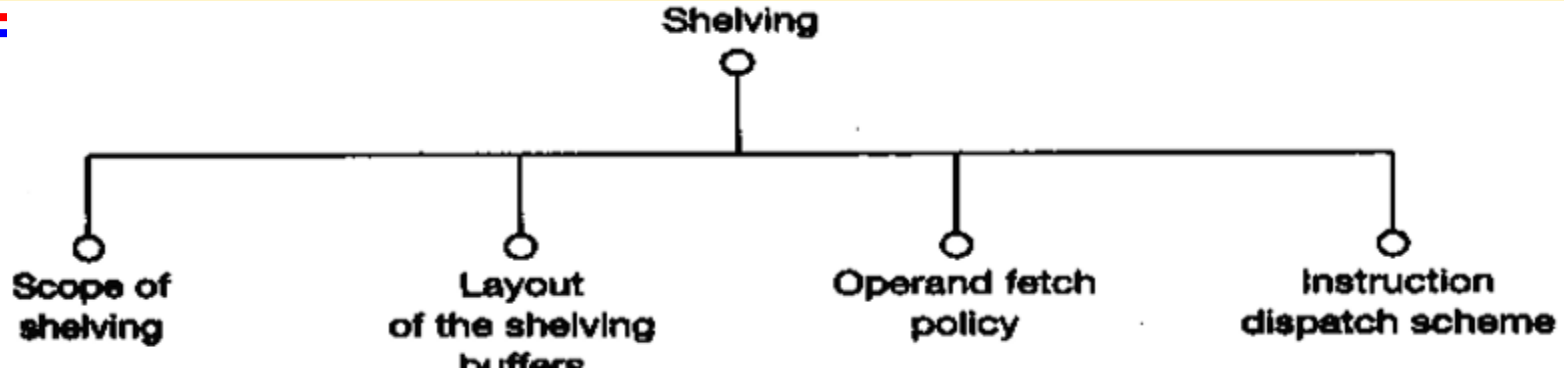
- The instructions are checked for dependency in decode stage.
- If there is any dependency the instruction is blocked. If there is no dependency the instruction is directly issued.

7.3.1.2 The principle of shelving: Indirect Issue



- Shelving is nothing but buffering the instructions
- The instructions are checked for dependency in the buffer called Reservation Station.
- If there is any dependency the instruction is blocked. If there is no dependency the instruction is directly issued.

7.3.1.2 Design space of shelving



a)Partial shelving: Shelving is restricted to one or to a few instruction types (Power1, Power2 – only FP instructions)

b)Full Shelving: Shelving covers all instruction types (PowerPC 620)

a) Type of the shelving buffer: S:22
- standalone shelving Buffer (Reservation station)
- Combined buffer for shelving, renaming and reordering (DRIS: Deferred scheduling, register renaming, Instruction shelve) S:23

b) Number of Shelving buffers
PowerPC 620 (15 number)
PentiumPro (20 number)

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

a)Issue Bound Fetch : Operands are fetched during instruction issue. Then shelving buffer hold source operand value S:24
Eg: PowerPC 620
PentiumPro

b) Dispatch bound fetch Operands are fetched during Instruction dispatch. Then shelving buffers hold source register numbers S:25
Eg: Power1, Power2

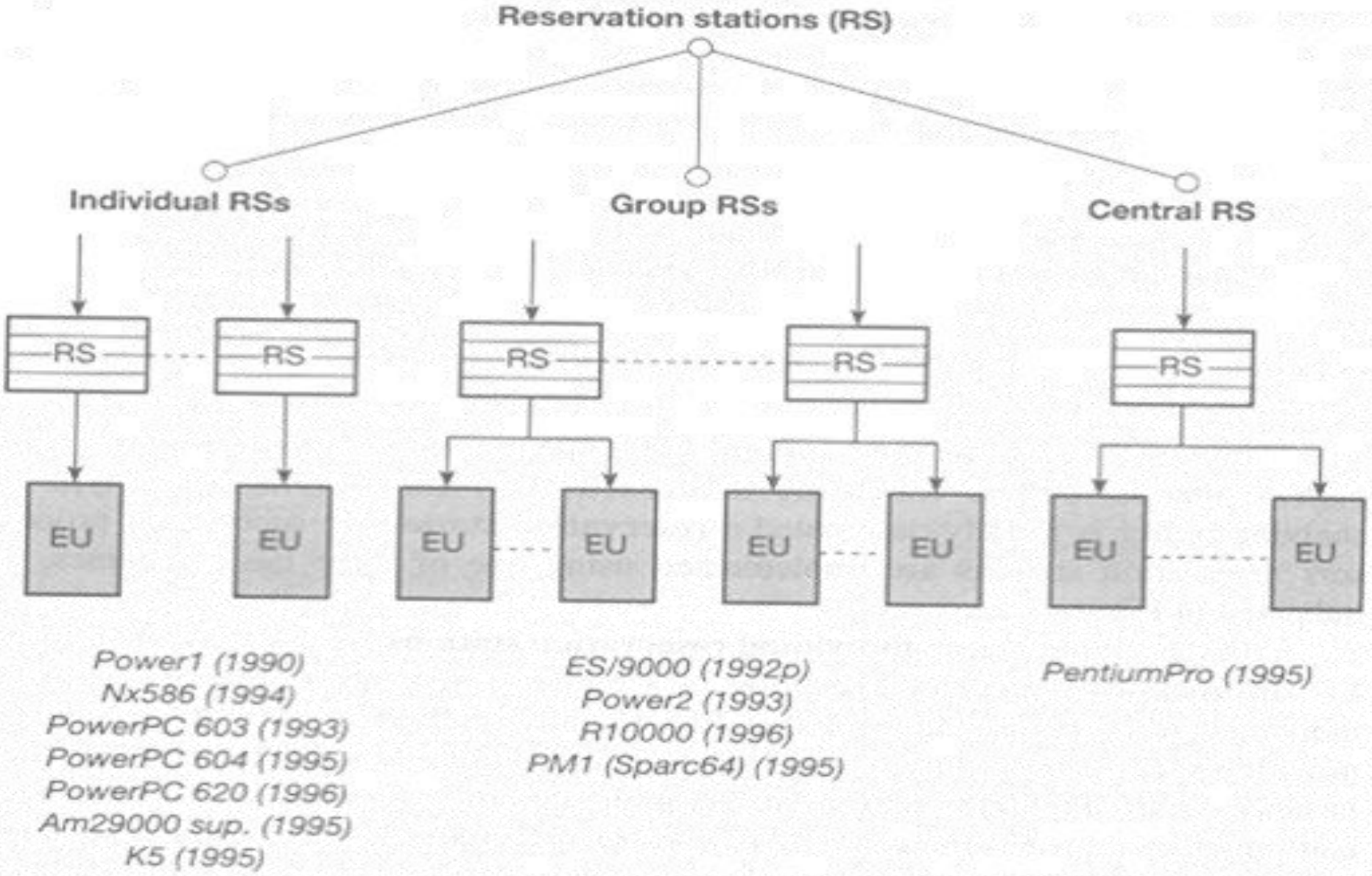
a)Dispatch policy S:26 -
selection rule
-arbitrary rule
-dispatch order S:27

b) Dispatch rate S:28 S:29

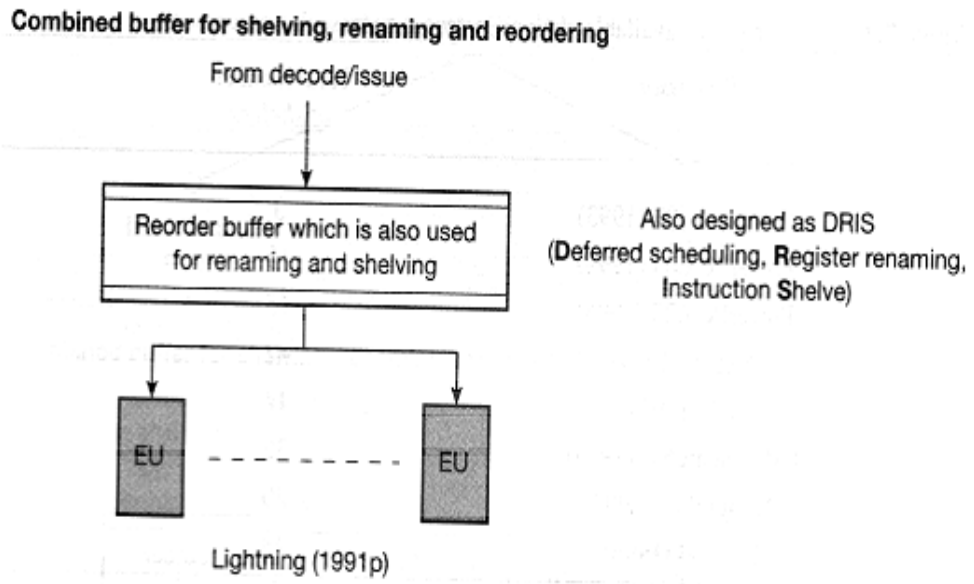
c)Scheme for checking the availability of operands.
Score boarding
S: 30 S:31 S:32 S:33 S:34

d) Treatment of an empty reservation station
--Straightforward approach
--Bypassing
S:35

7.3.1.2 Basic variants of shelving buffers



7.3.1.2 Using a combined buffer for shelving, renaming, and reordering

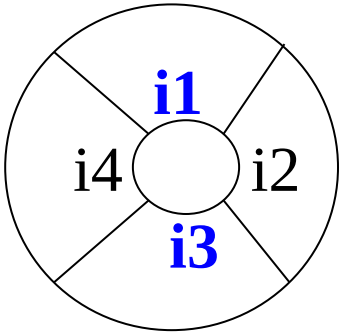


- **The buffer is used for**
  -shelve the instructions
  -Register renaming
  -Reordering Instruction

- **Shelving:** Buffer for saving instructions

- **Reordering** : The execution of instruction may be out-of-order. But final writeback must be in order.
- Here i1 and i3 may complete out of order. But writeback of i3 happens only after completion of i2

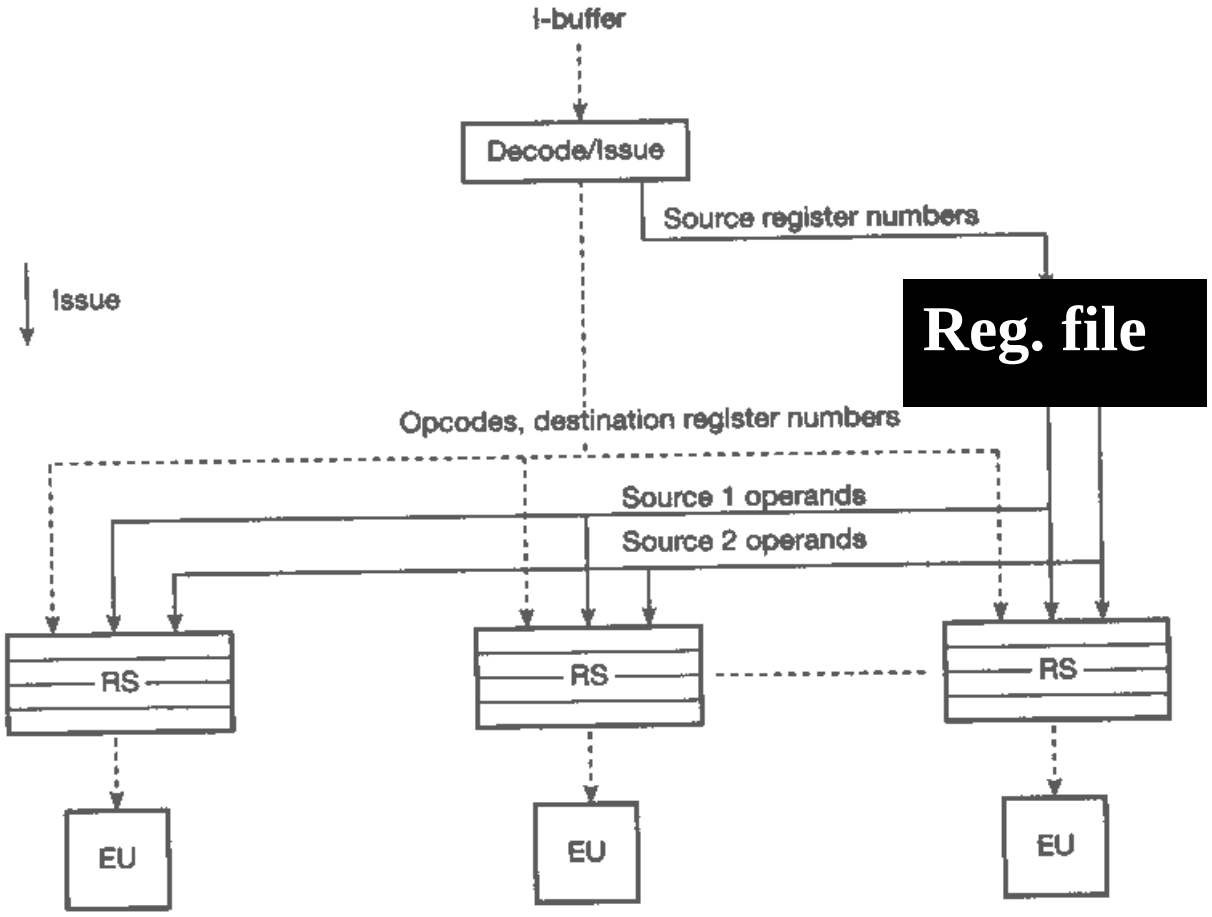
Reorder Buffer



- **Register Renaming:**

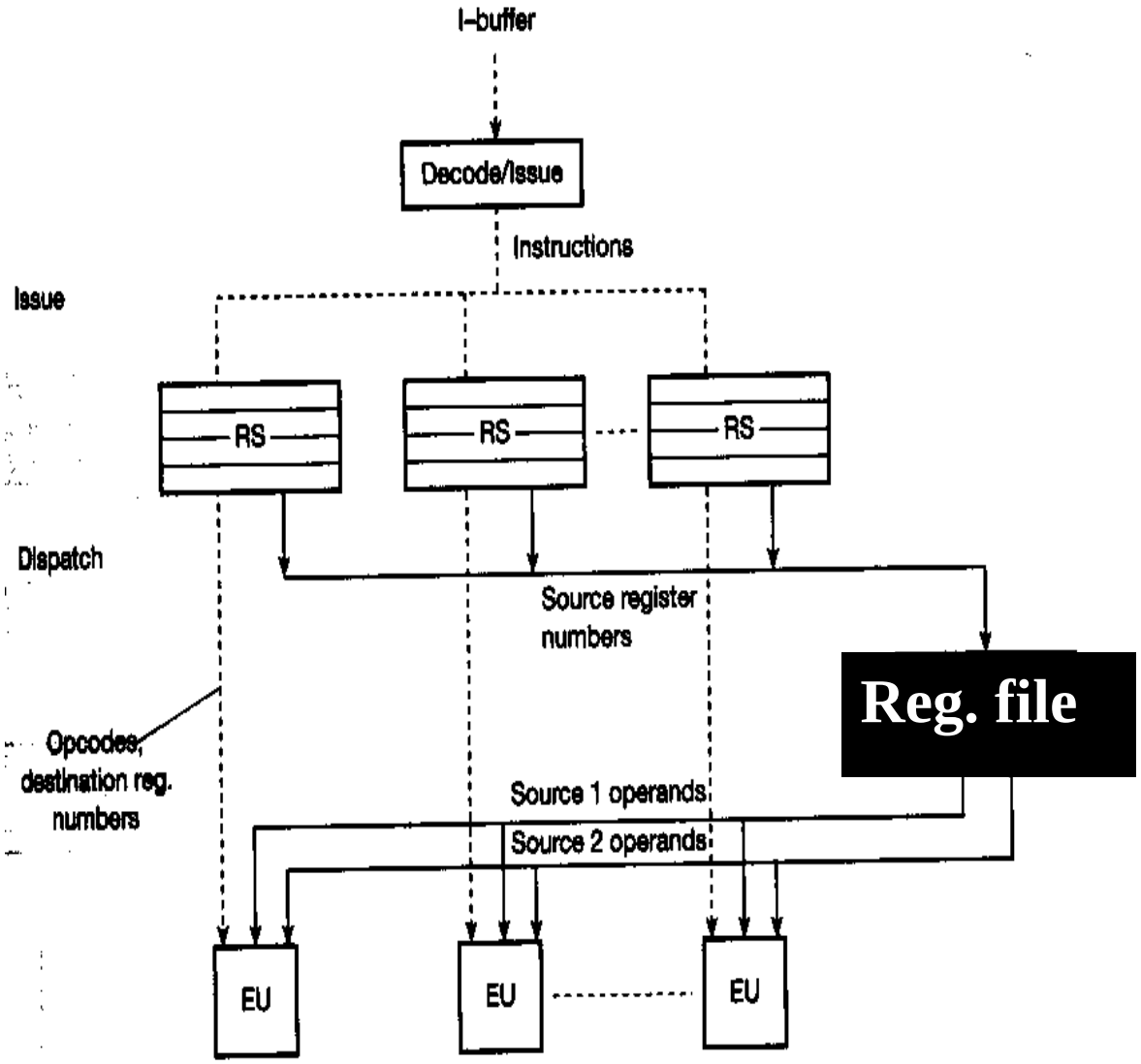
i1: Add **R1,** R2, R3
i2: Store **R1**, Loc1
i3: Add **R1,** R4, R5
i4: Store **R1,** Loc2
Rename as follows
i1: Add **R11,** R2, R3
i2: Store **R11**, Loc1
i3: Add **R12,** R4, R5
i4: Store **R12,** Loc2

7.3.1.2 Operand fetch during instruction issue



- Operands are fetched from register file during instruction issue

7.3.1.2 Operand fetch during instruction dispatch

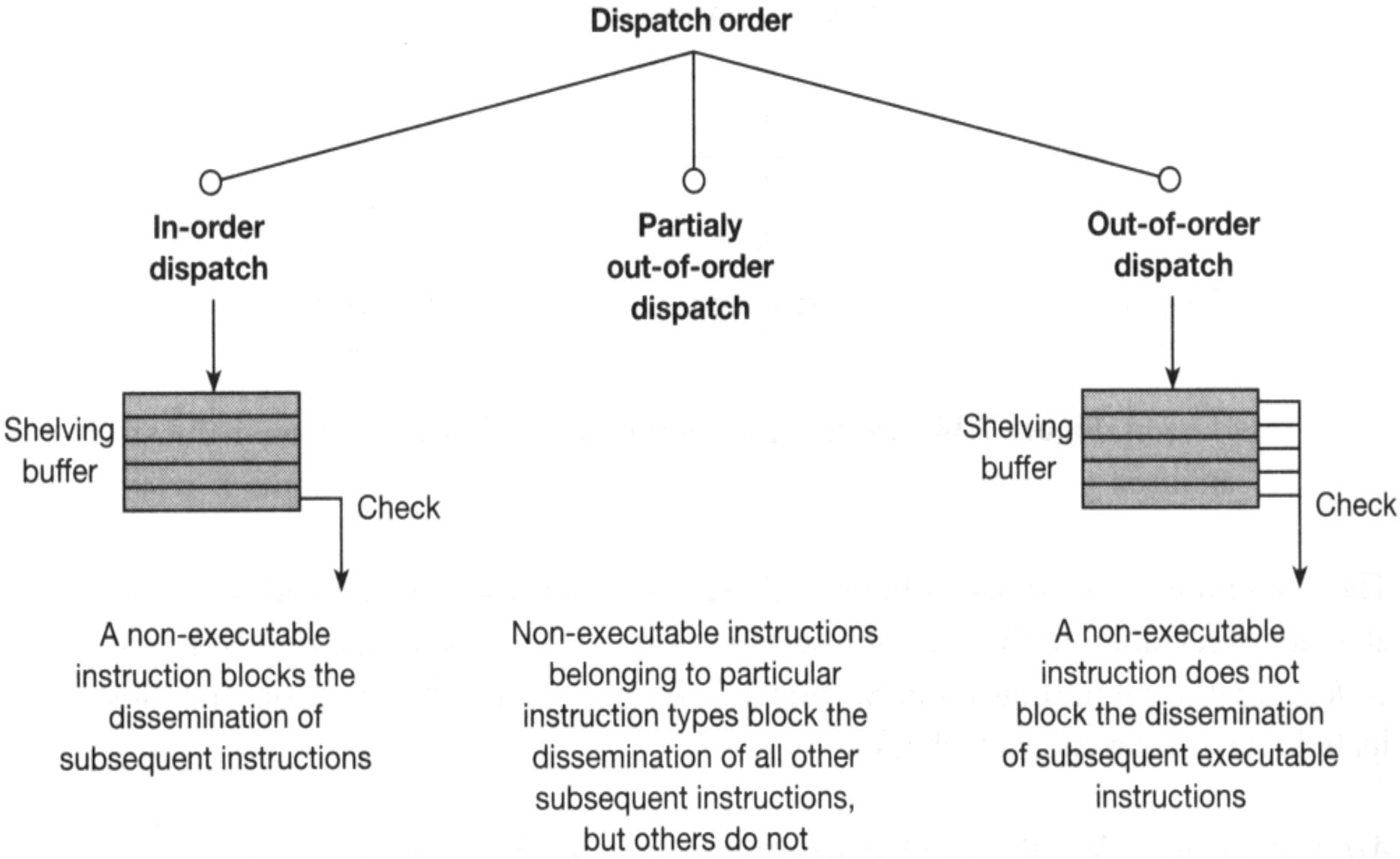


- Operands are fetched from register file during instruction dispatch to Execution unit

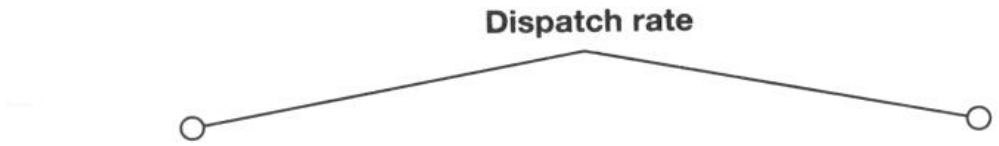
7.3.1.2- Dispatch policy

- Selection Rule
  - **Specifies when instructions are considered executable**
  - **e.g. Dataflow principle of operation**
    - Those instructions whose **operands** are available are executable.
- Arbitration Rule
  - **Needed when more instructions are eligible for execution than can be disseminated.**
  - **e.g. choose the ‘oldest’ instruction.**
- Dispatch order
  - **Determines whether a non-executable instruction prevents all subsequent instructions from being dispatched.**

7.3.1.2 Dispatch policy: Dispatch order



7.3.1.2 -Dispatch rate (instructions/cycle)

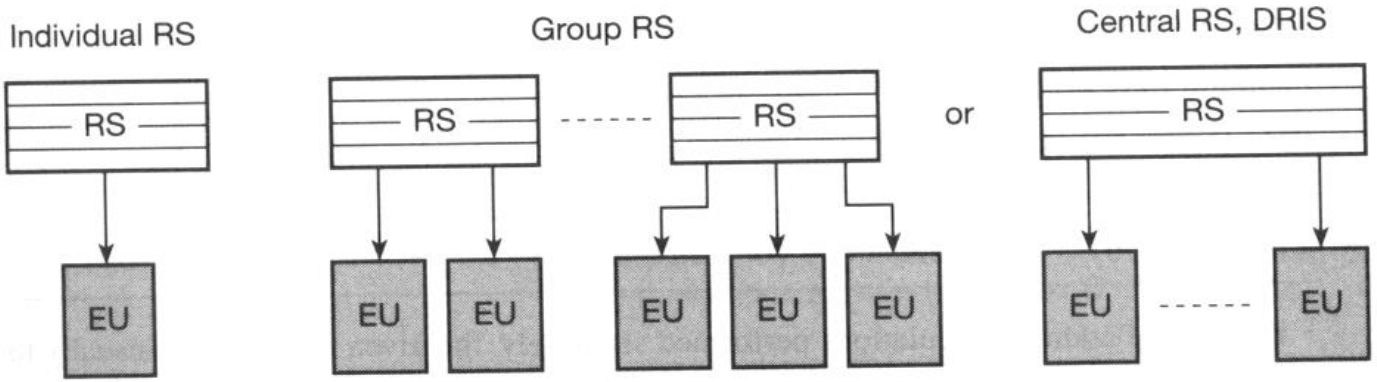


Single instruction/cycle

Typical with individual RSs

Multiple instructions/cycle

Typical when group stations,
a central station or a DRIS is used



Examples are, among others:

PowerPC 603 (1993)
PowerPC 604 (1994)
PowerPC 620 (1995)

PM1 (Sparc64) (1995)
R10000 (1996)
PA 8000 (1996)

Lightning (1991p)
PentiumPro (1995)

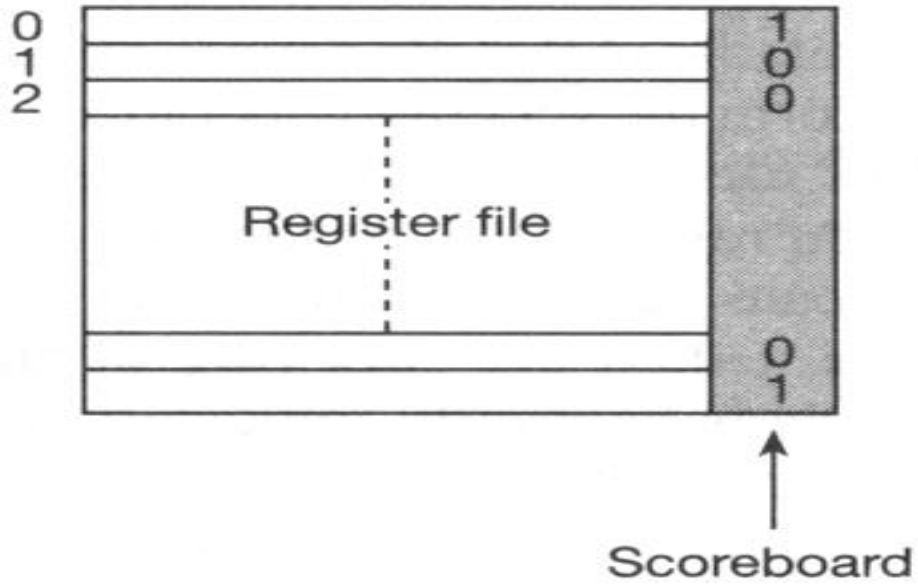
Maximum issue rate <= Maximum dispatch rates
>> issue rate reaches max more often than dispatch rates

Table 7.3 Maximum issue and dispatch rates of superscalar processors with shelving.

Processor/Year of volume shipment	*Maximum issue rate instr./cycle*	*Maximum dispatch rate[1] instr./cycle*
PowerPC 603 (1993)	3	3
PowerPC 604 (1995)	4	6
PowerPC 620 (1996)	4	6
Power2 (1993)	4/6[2]	10
Nx586 (1994)	3/4[3,4]	3/4[3,4]
K5 (1995)	4[4]	5[4]
PentiumPro (1995)	4	5[4]
PM1 (Sparc 64) (1995)	4	8
PA8000 (1996)	4	4
R10000 (1996)	4	5

[1] Because of address calculations performed separately, the given numbers are usually to be interpreted as operations/cycle. For instance, the Power2 performs maximum 10 operations/cycle, which corresponds to 8 instructions/cycle.
[2] The issue rate is 6 for sequential mode and 4 for target mode.
[3] Both rates are 3 without an optional FP unit (labelled Nx587) and 4 with it.
[4] Both rates are related to RISC operations (rather than to the native CISC operations) performed by the superscalar RISC core.

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

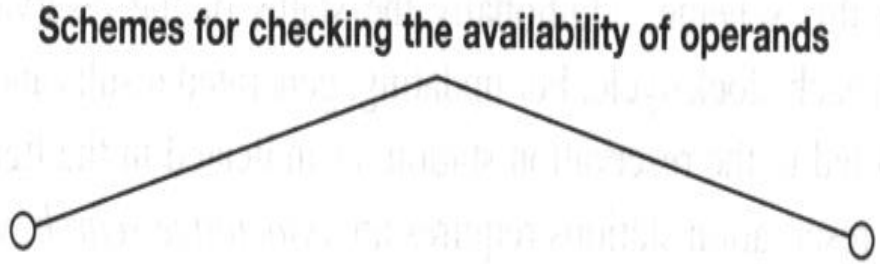


Interpretation:

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'

7.3.1.2 Schemes for checking the availability of operand



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

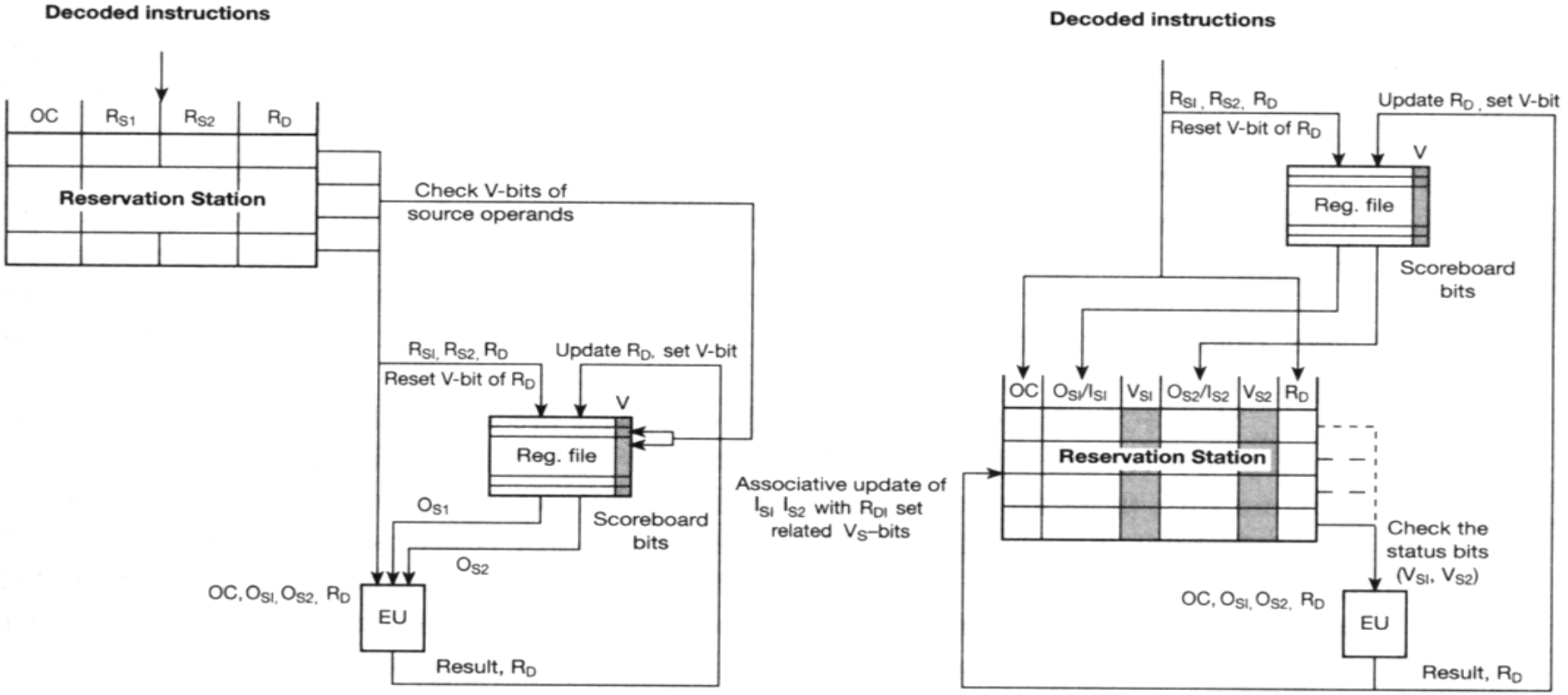
Usually employed if operands are fetched during instruction dispatch, as assumed below

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

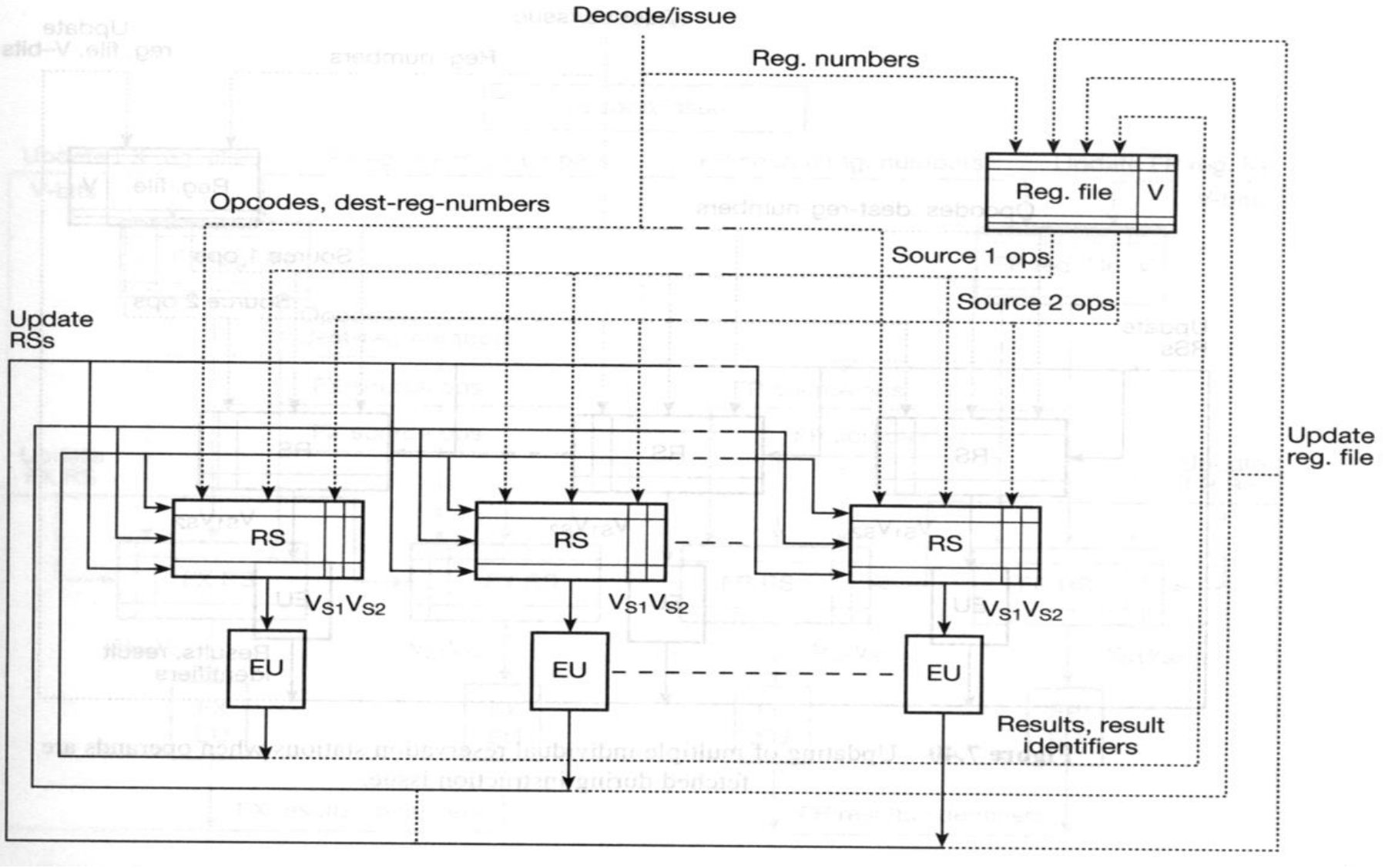
Usually employed if operands are fetched during instruction issue, as assumed below

7.3.1.2 Operands fetched during dispatch or during issue

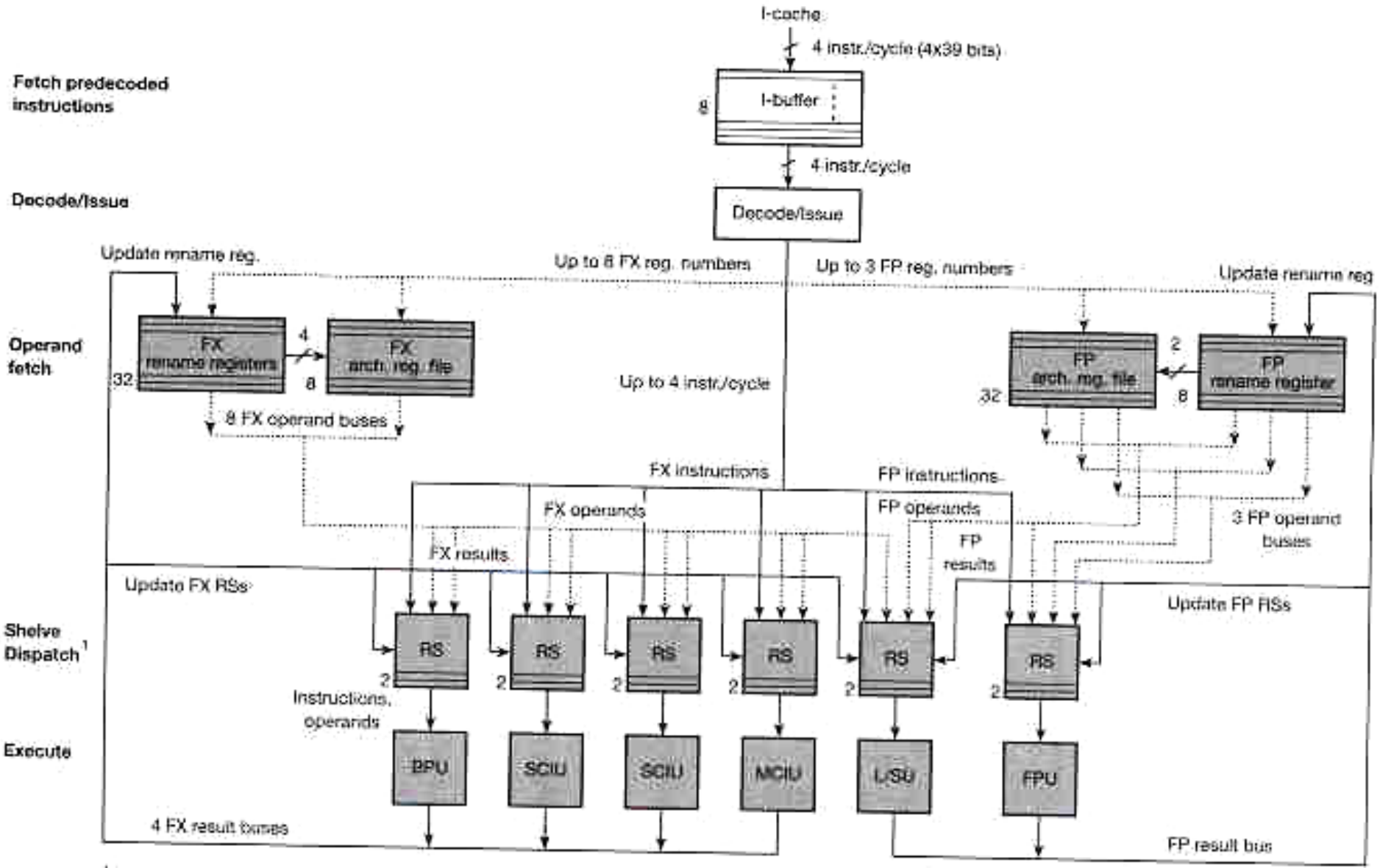


OC:	Operation code
R_{SI}, R_{S2}:	Source register numbers
R_D:	Destination register number
O_{SI}, O_{S2}:	Source operand values
I_{SI}, I_{S2}:	Source operand identifiers (tags)
V_{SI}, V_{S2}:	Source operand valid bits
RS:	Reservation station

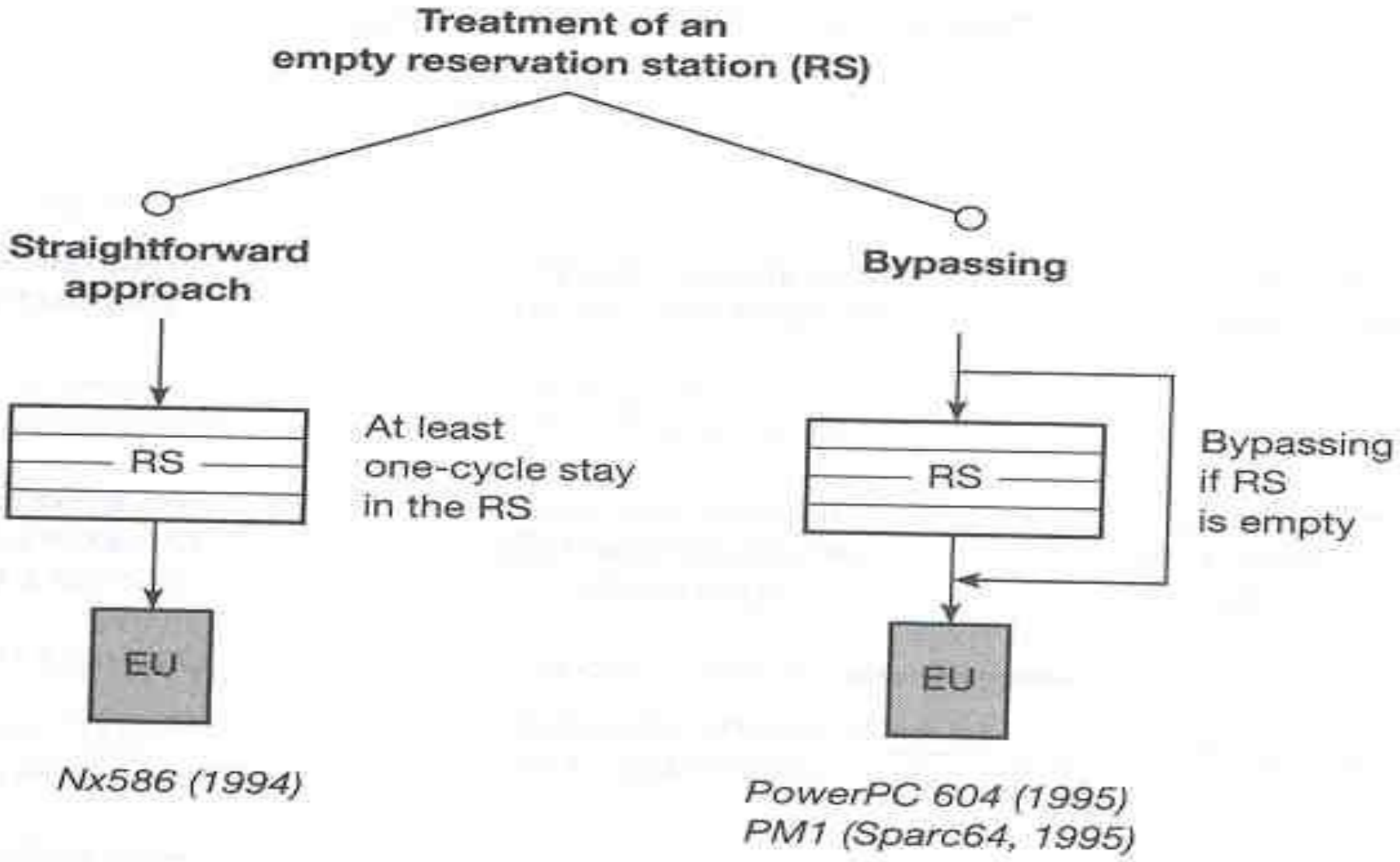
7.3.1.2 Use of multiple buses for updaing multiple individual reservation strations



7.3.1.2 Interal data paths of the powerpc 604



7.3.1.2 -Treatment of an empty reservation station



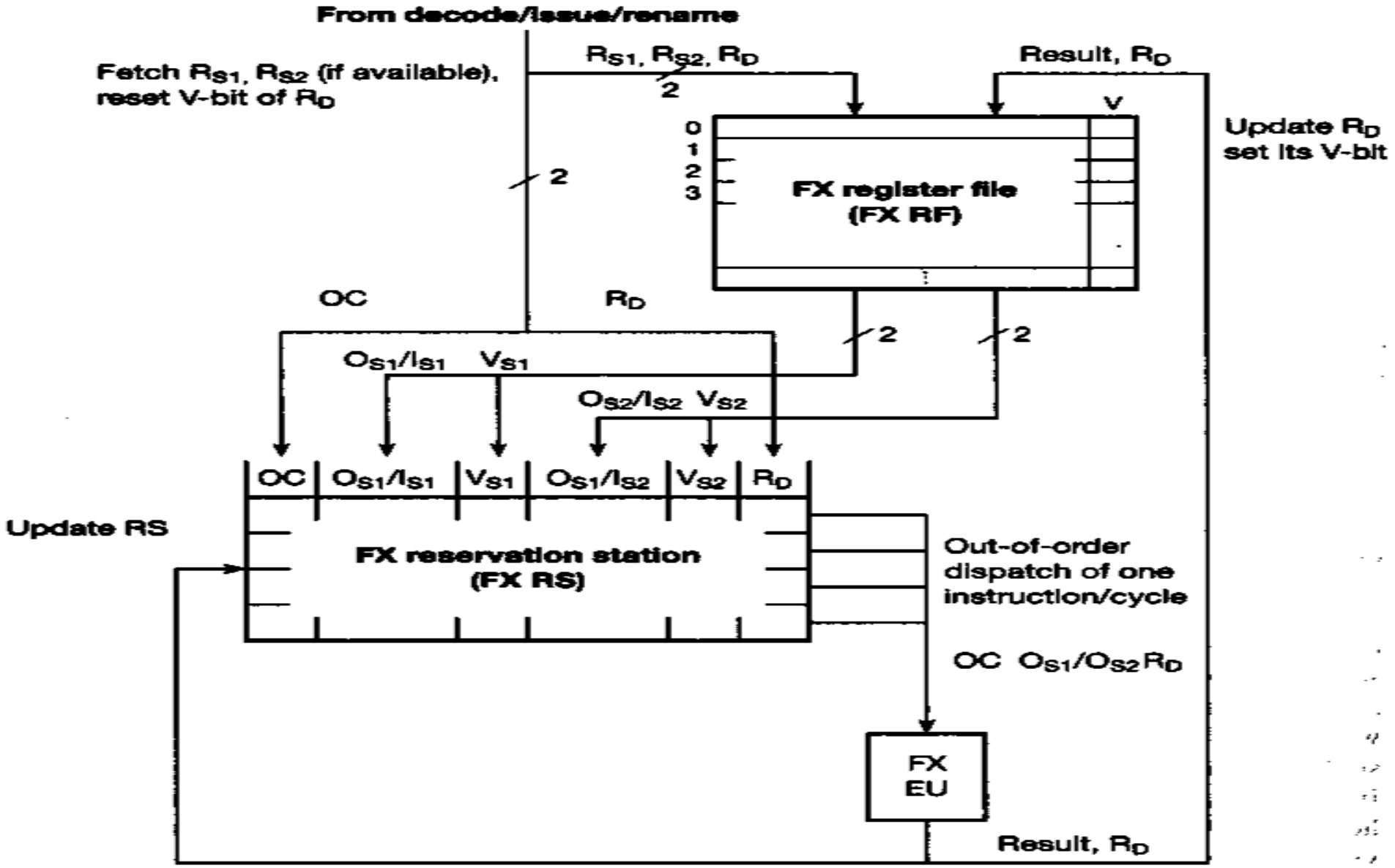
7.4 Detail Example of Shelving

- Issuing the following instruction
  - **cycle i: mul r1, r2, r3**
  - **cycle i+1: ad r2, r3, r5**
  - **ad r3, r4, r6**

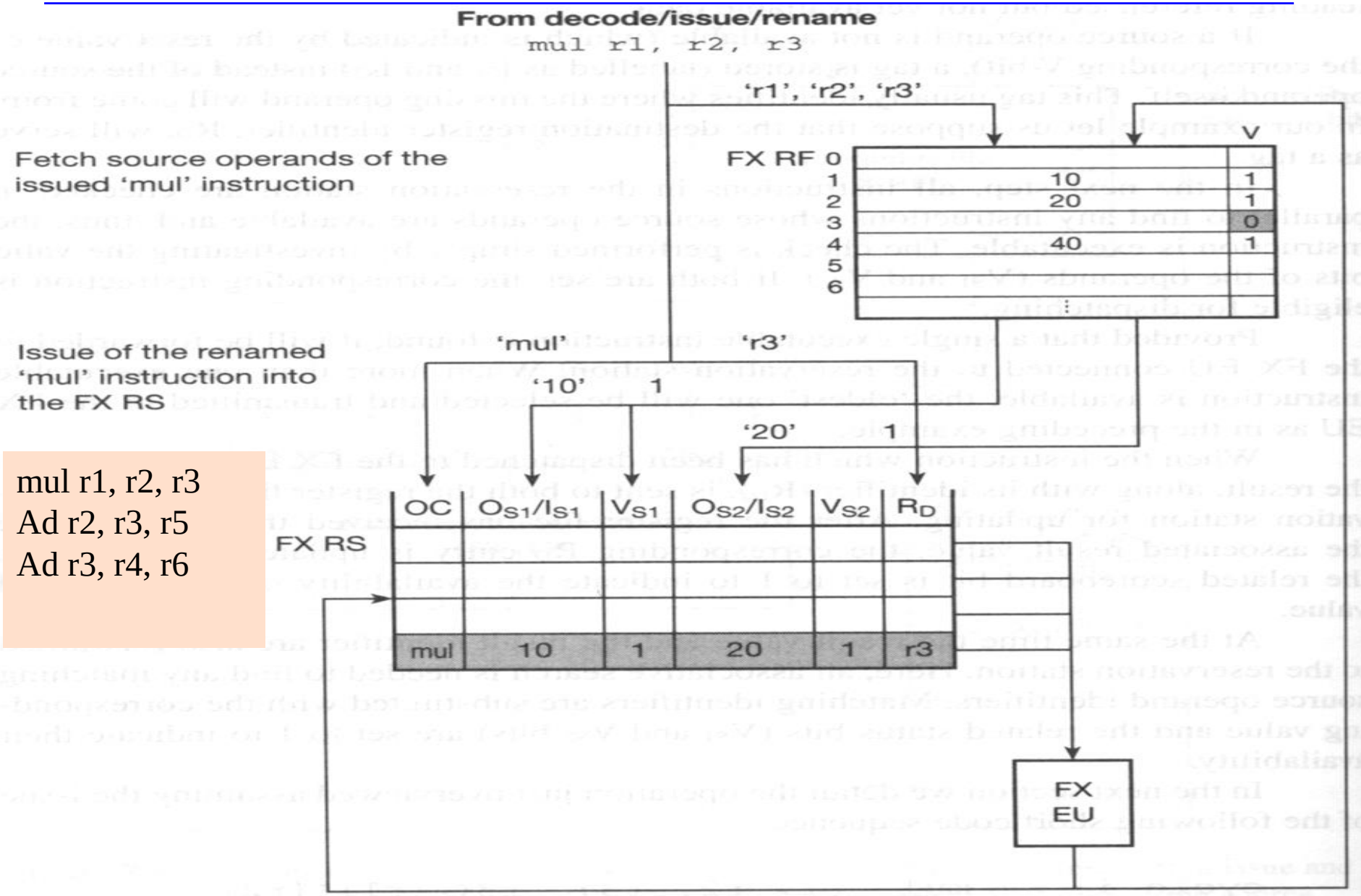
    - format: Rs1, Rs2, Rd

mul r1, r2, r3
Ad r2, r3, r5
Ad r3, r4, r6

Example overview



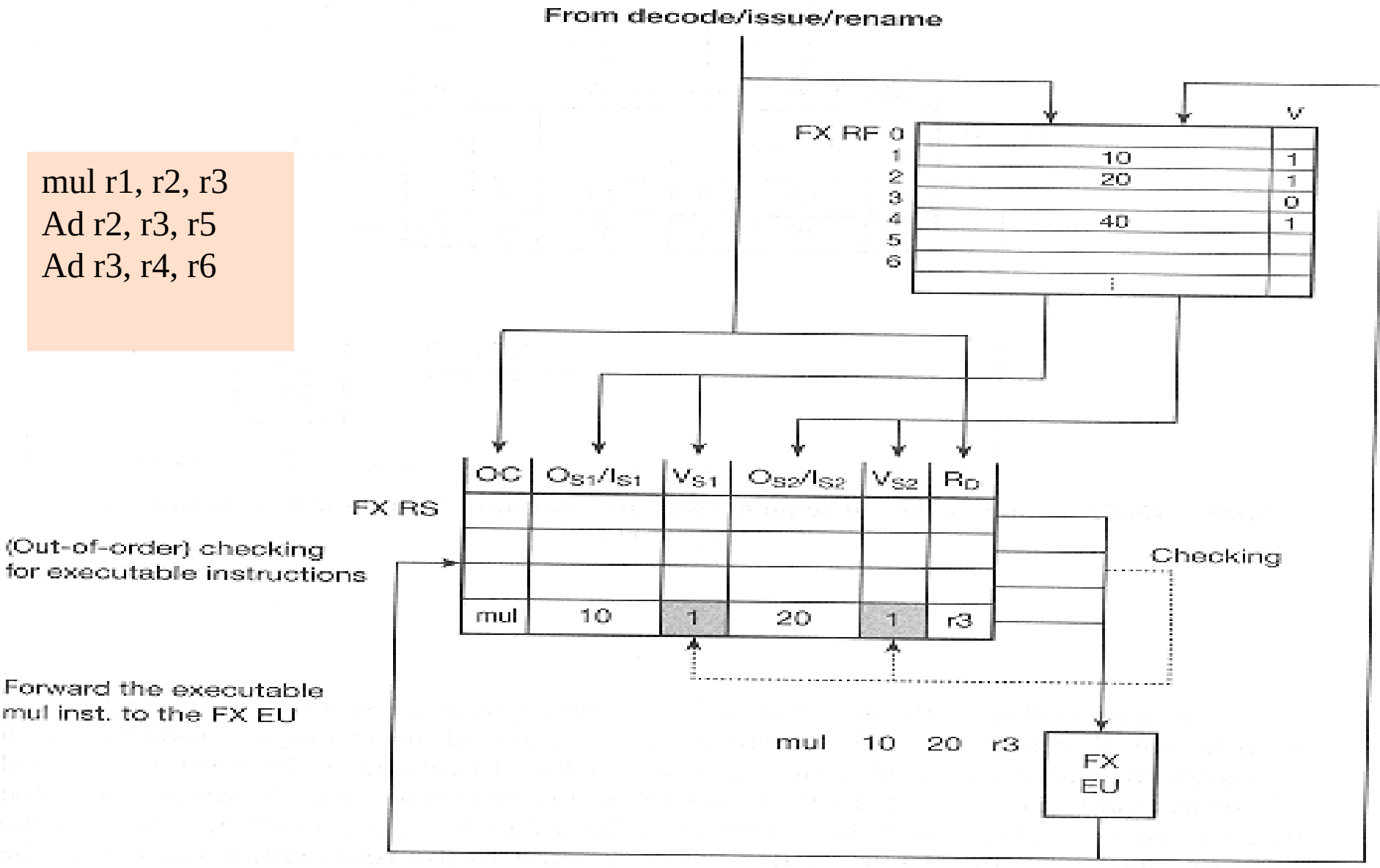
Cycle i: Issue of the 'mul' instruction into the reservation station and fetching of the corresponding operands



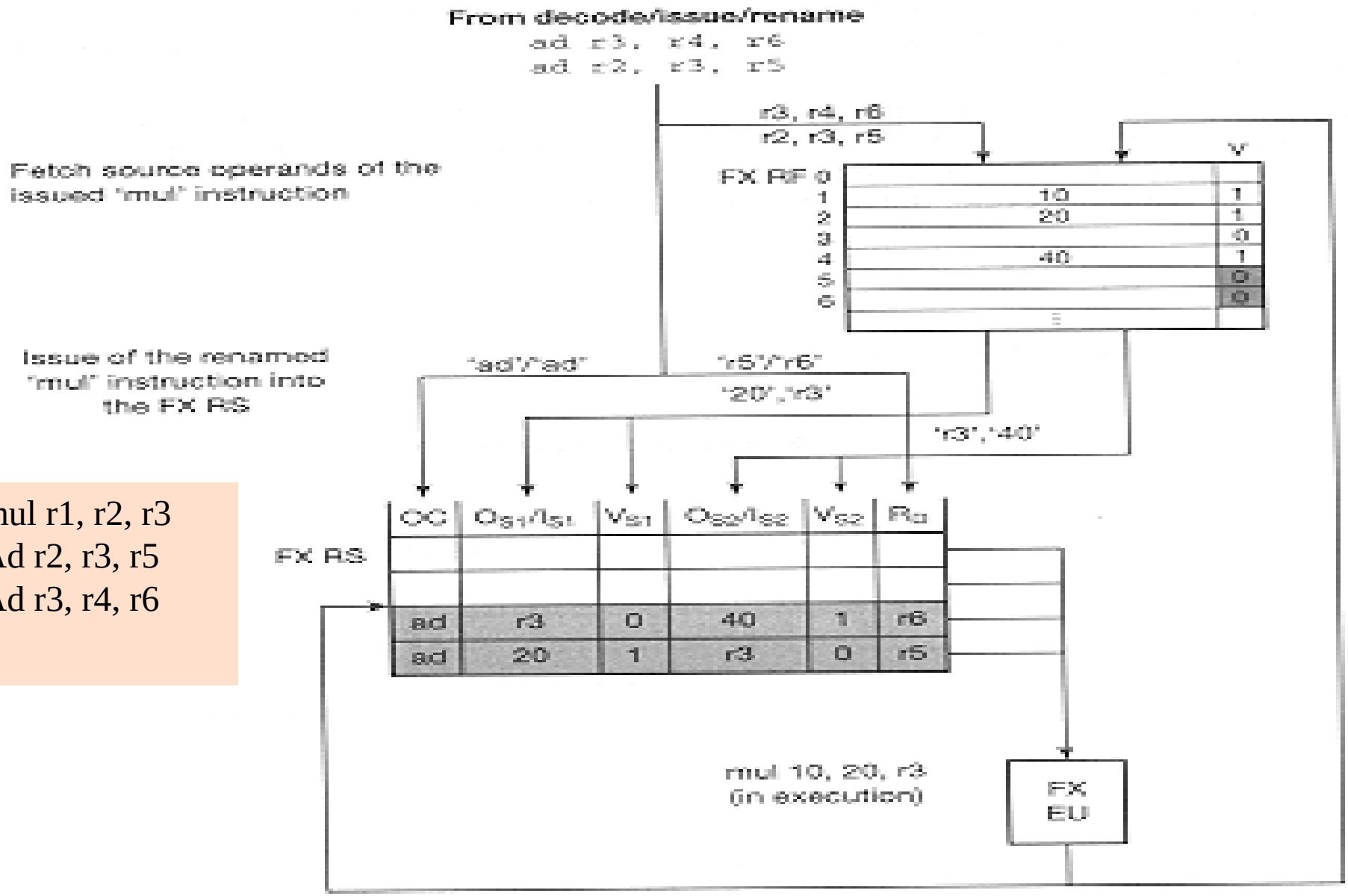
mul r1, r2, r3
Ad r2, r3, r5
Ad r3, r4, r6

Cycle i+1: Checking for executable instructions and dispatching of the 'mul' instruction

mul r1, r2, r3
Ad r2, r3, r5
Ad r3, r4, r6



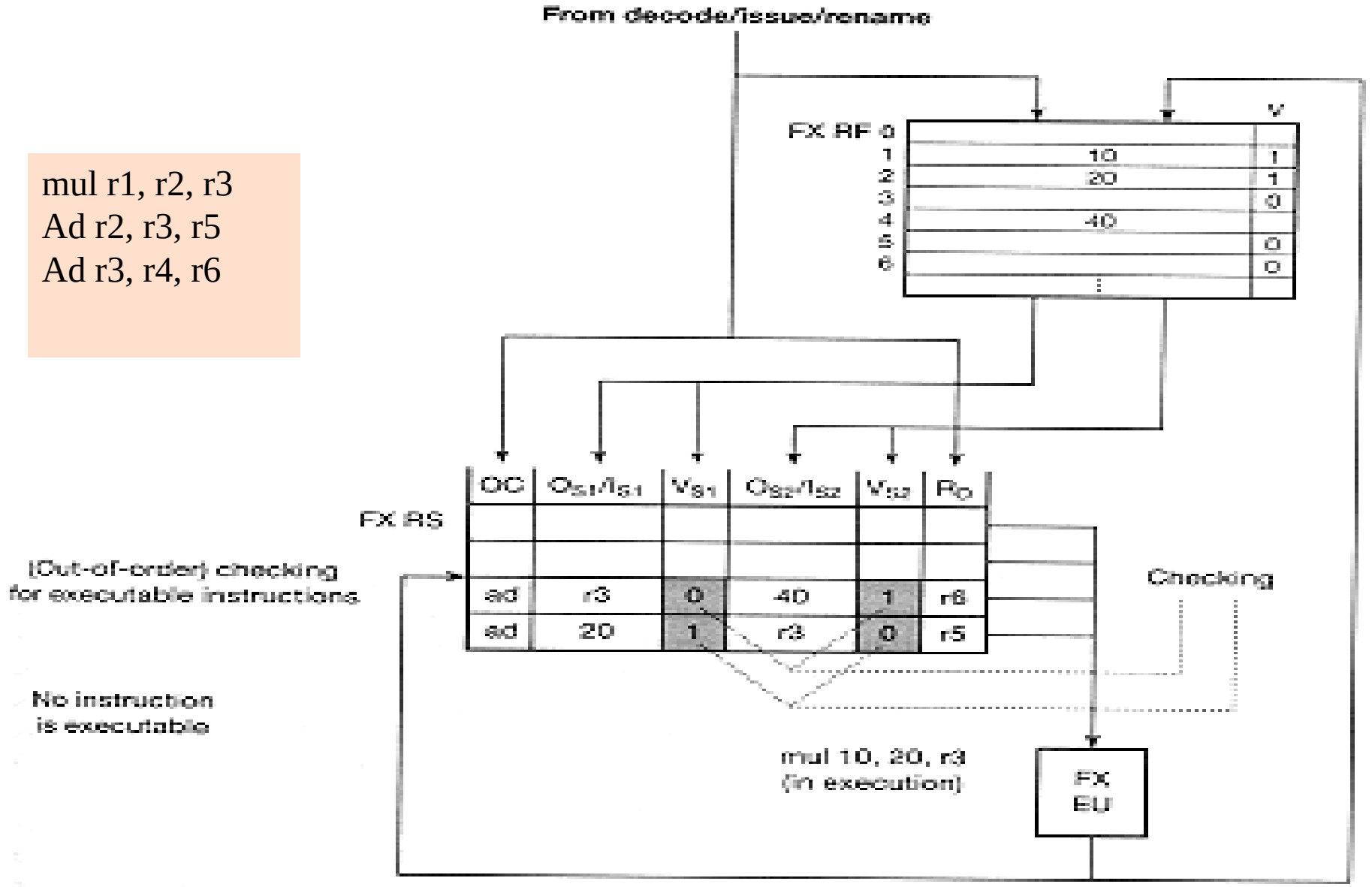
Cycle i+1 (2nd phase): Issue of the subsequent two 'ad' instructions into the reservation station



mul r1, r2, r3
Ad r2, r3, r5
Ad r3, r4, r6

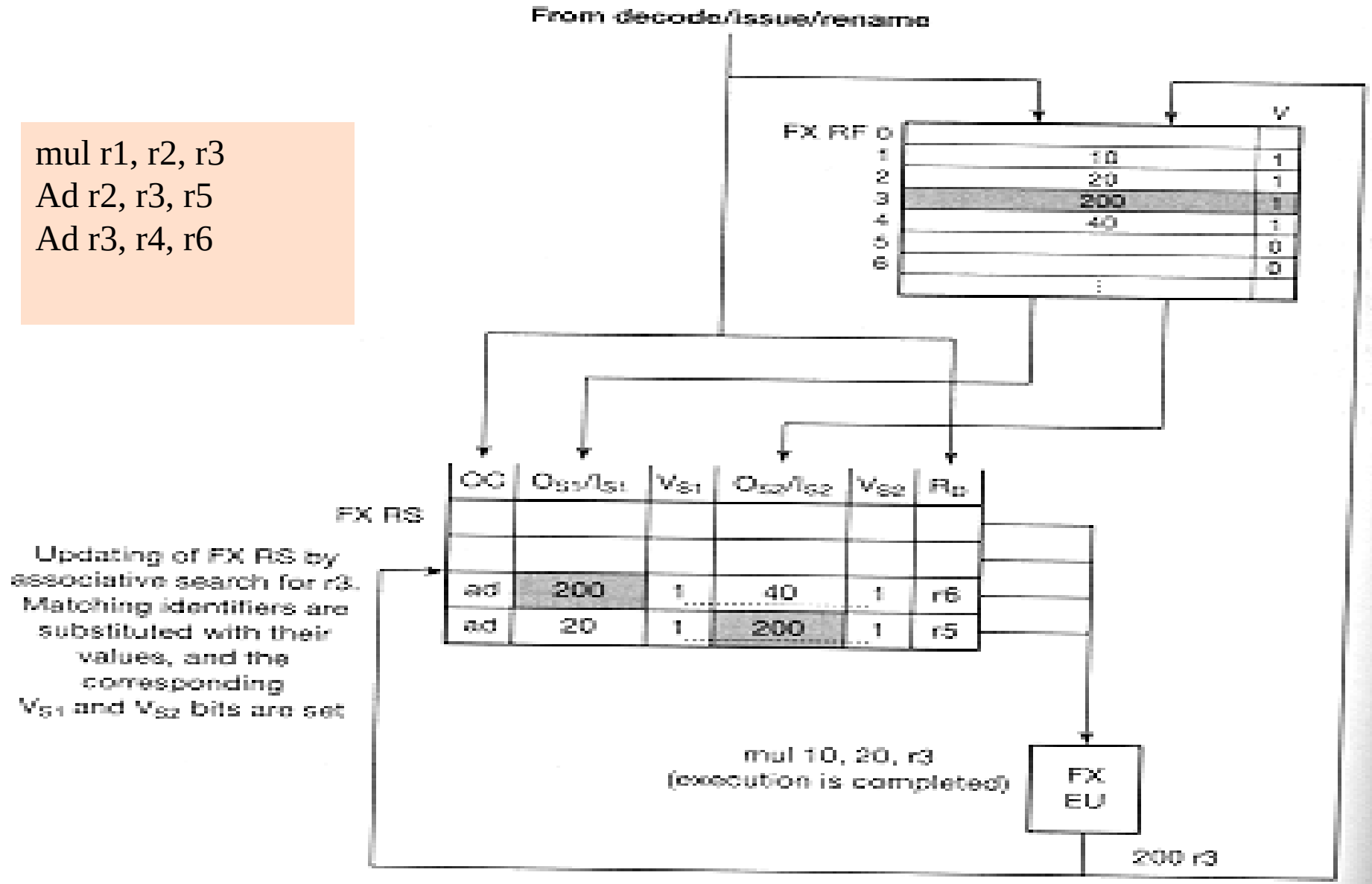
Cycle i+2: Checking for executable instruction (mul not yet completed)

mul r1, r2, r3
Ad r2, r3, r5
Ad r3, r4, r6



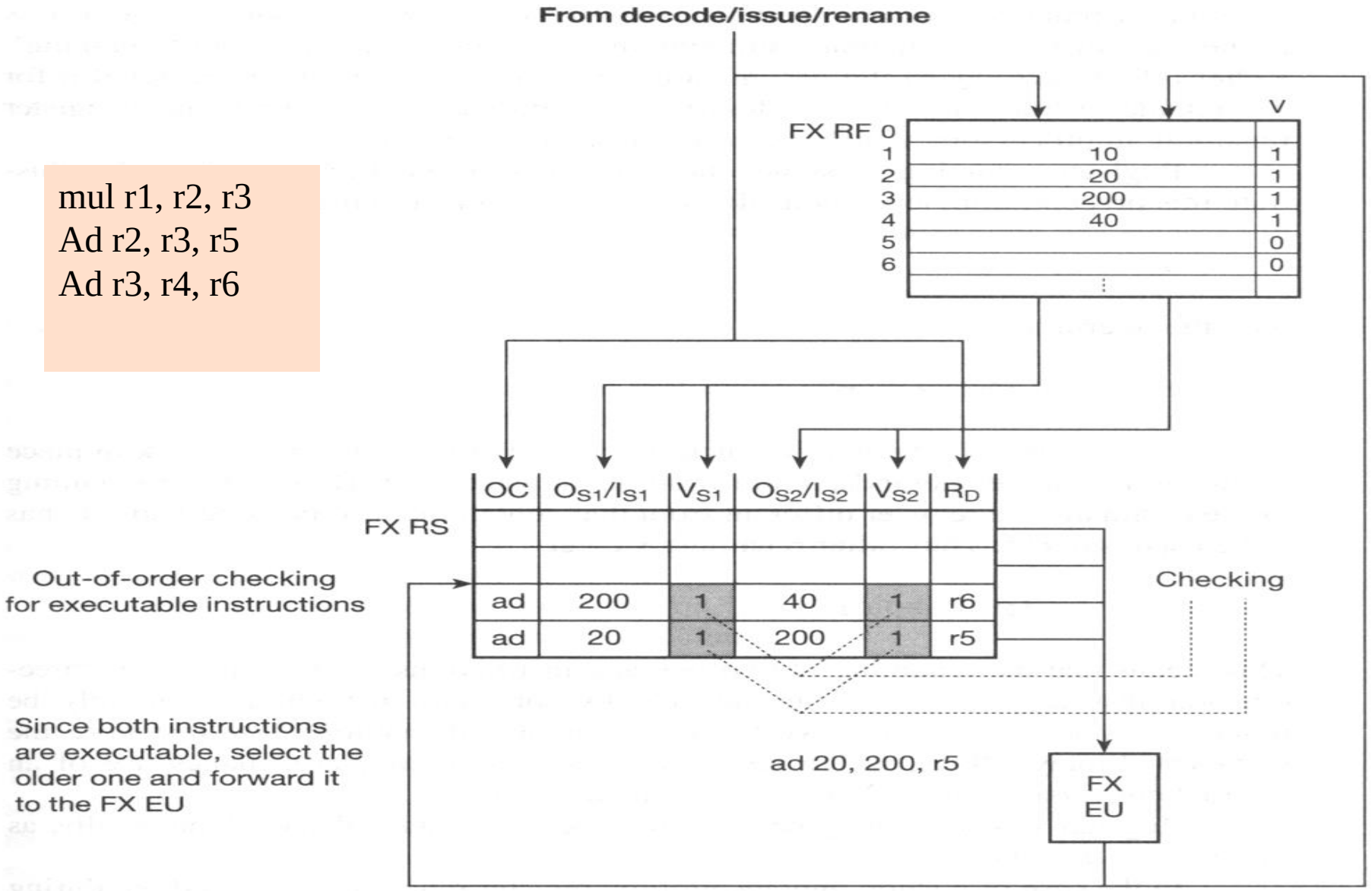
Cycle i+3: Updating the FX register file with the result of the 'mul' instruction

mul r1, r2, r3
Ad r2, r3, r5
Ad r3, r4, r6

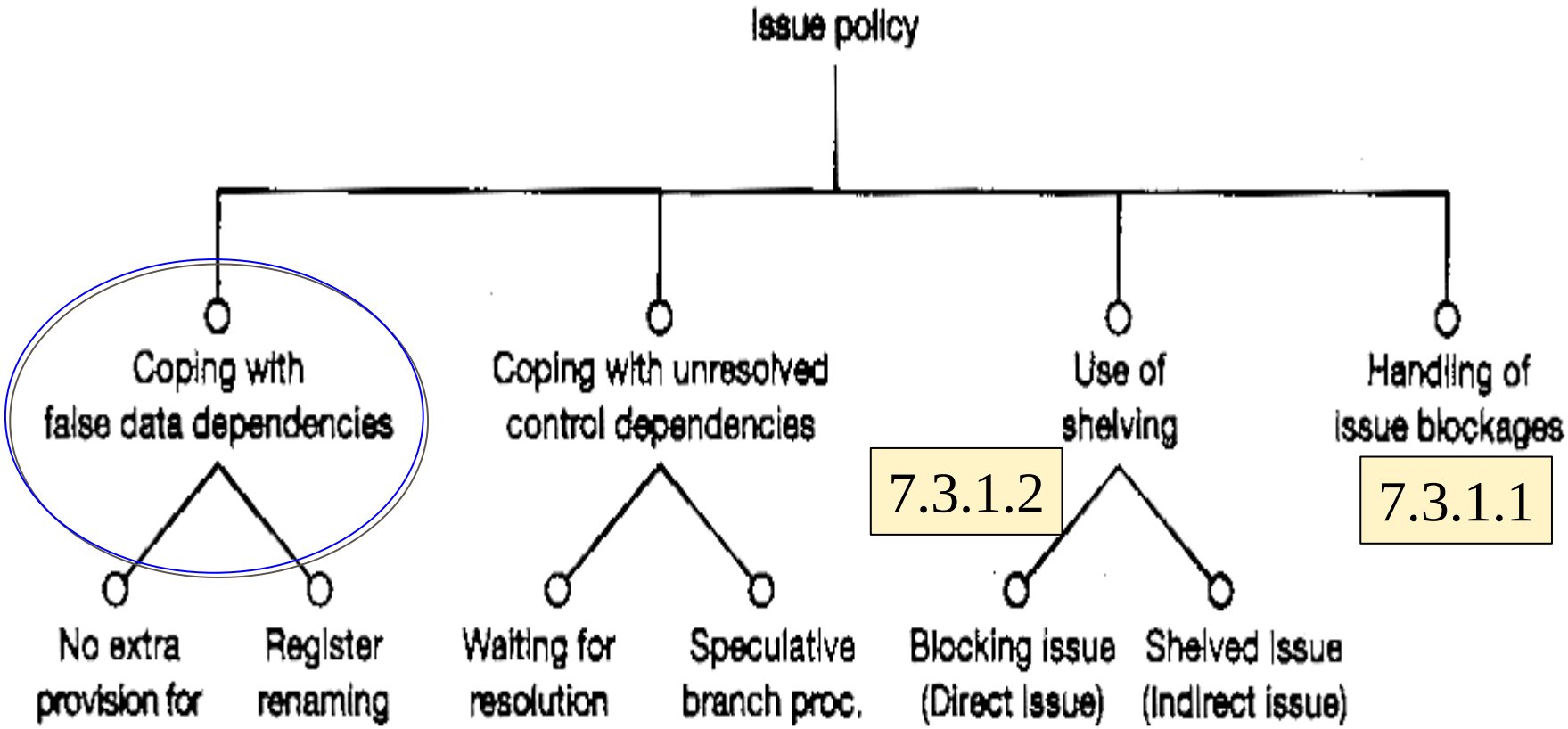


Cycle i+3 (2nd phase): Checking for executable instructions and dispatching the 'older' 'ad' instruction

mul r1, r2, r3
Ad r2, r3, r5
Ad r3, r4, r6



7.3.1 Issue policies: Handing Issue Blockages



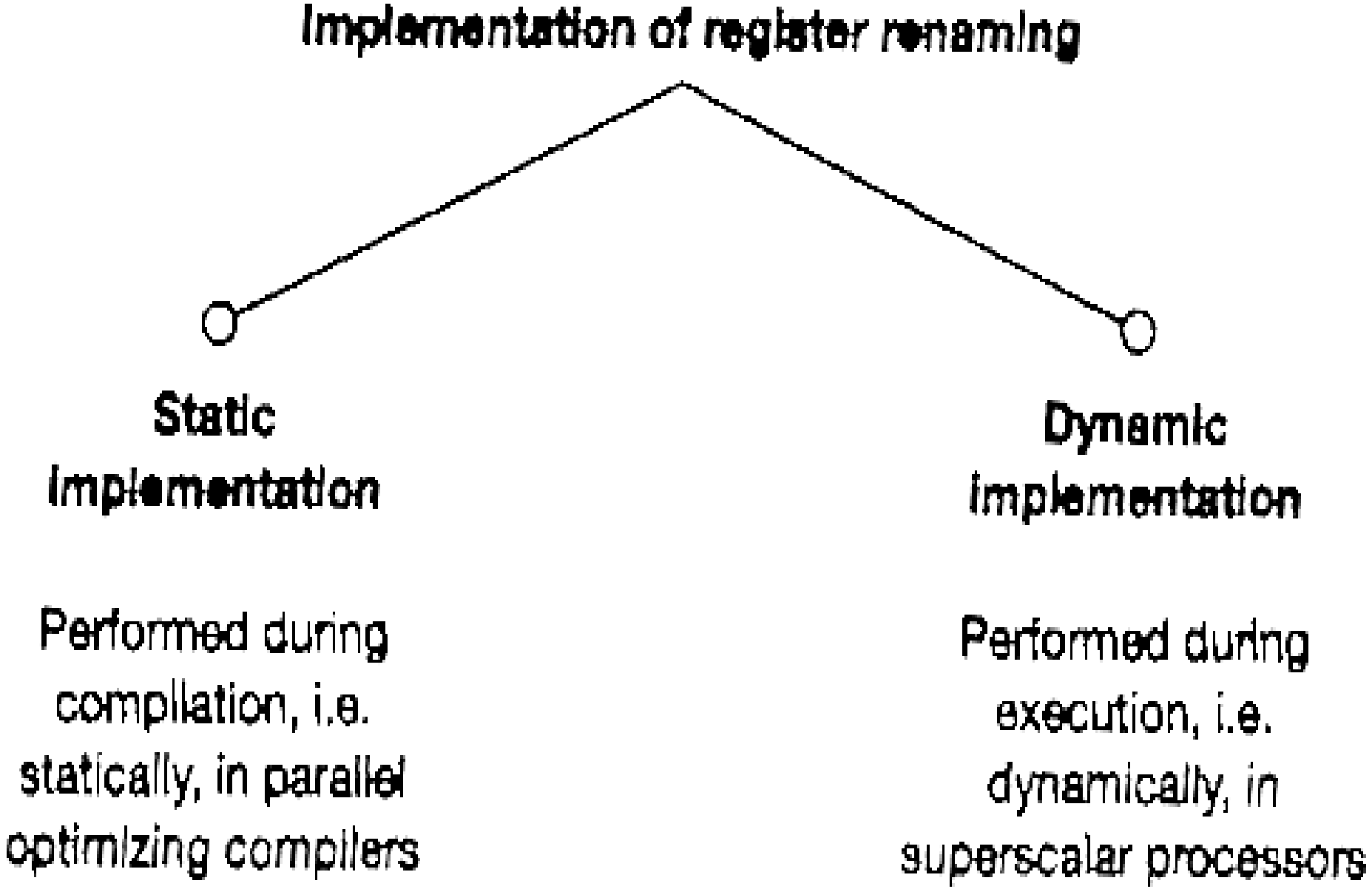
Resolving WAW hazards using register renaming

Speculating as branch taken or not taken based on history bits

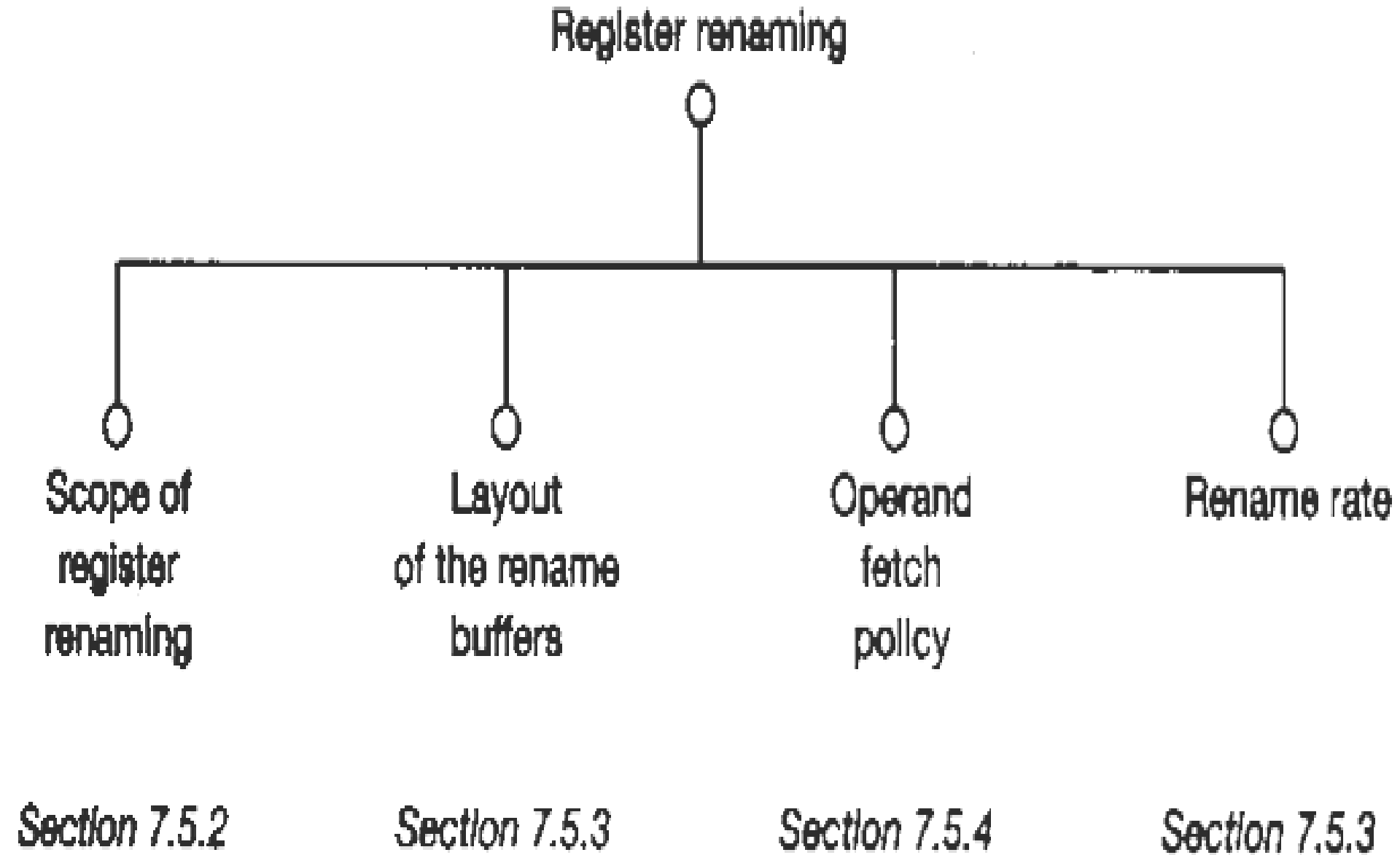
Register Remaining and dependency

- three-operand instruction format
- e.g. Rd, Rs1, Rs2
- False dependency (WAW)
  - **mul r2, …, …**
  - **add r2, …, …**
    - two different rename buffer have to allocated
- True data dependency (RAW)
  - **mul r2, …, …**
  - **ad …, r2, …**
    - rename to e.g.
    - mul p12, …, …
    - ad …, p12, ….

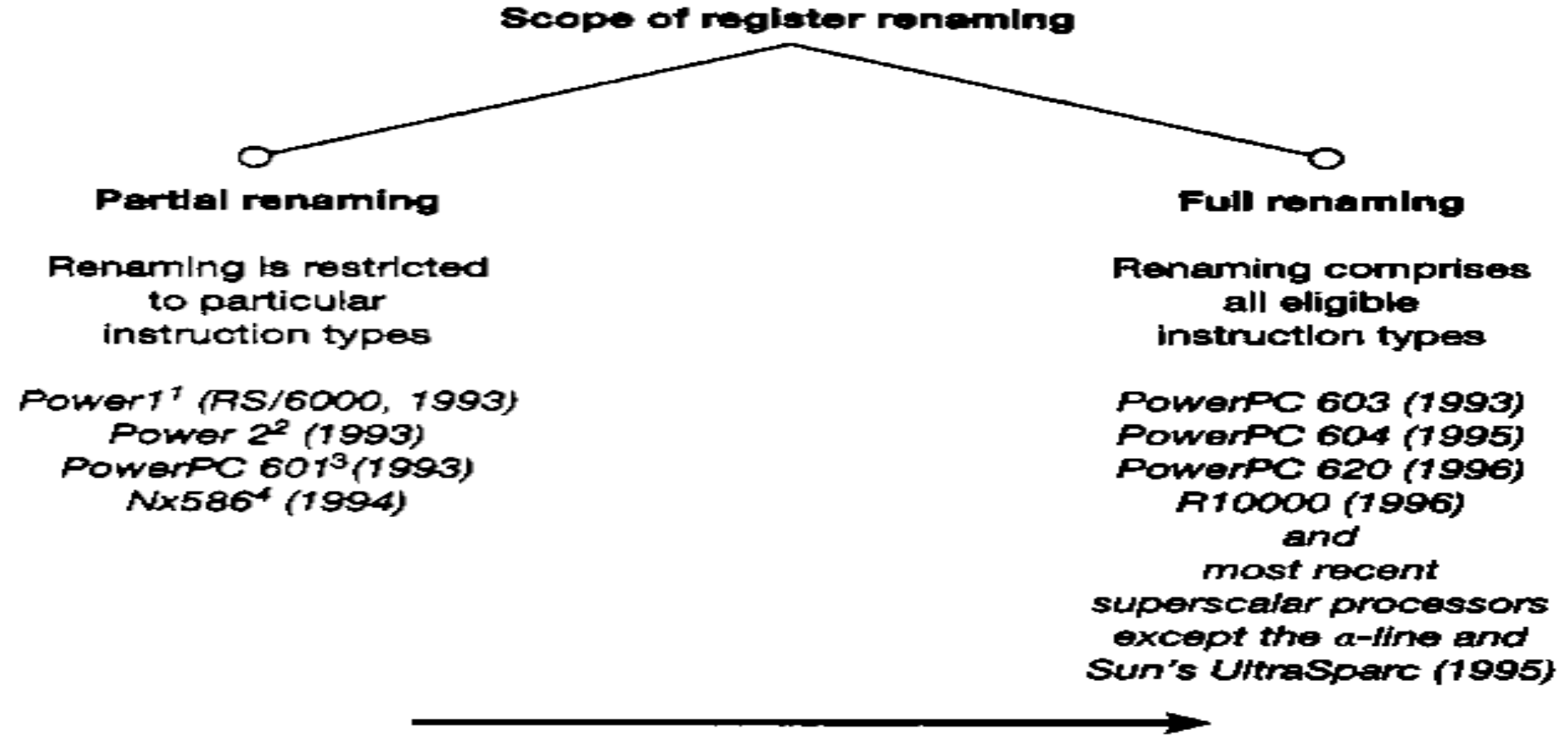
Static or Dynamic Renaming



>Design space of register renaming

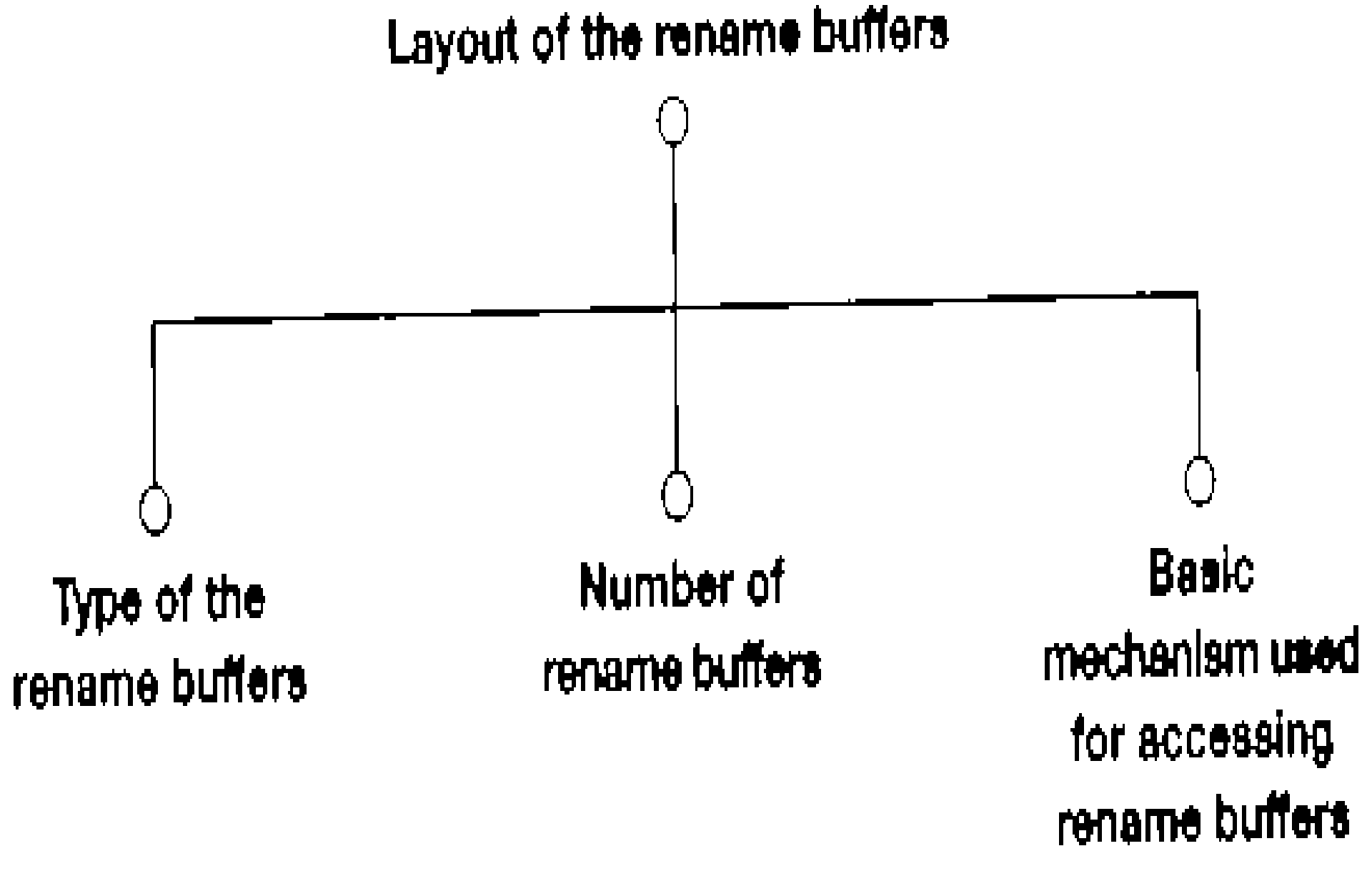


-Scope of register renaming

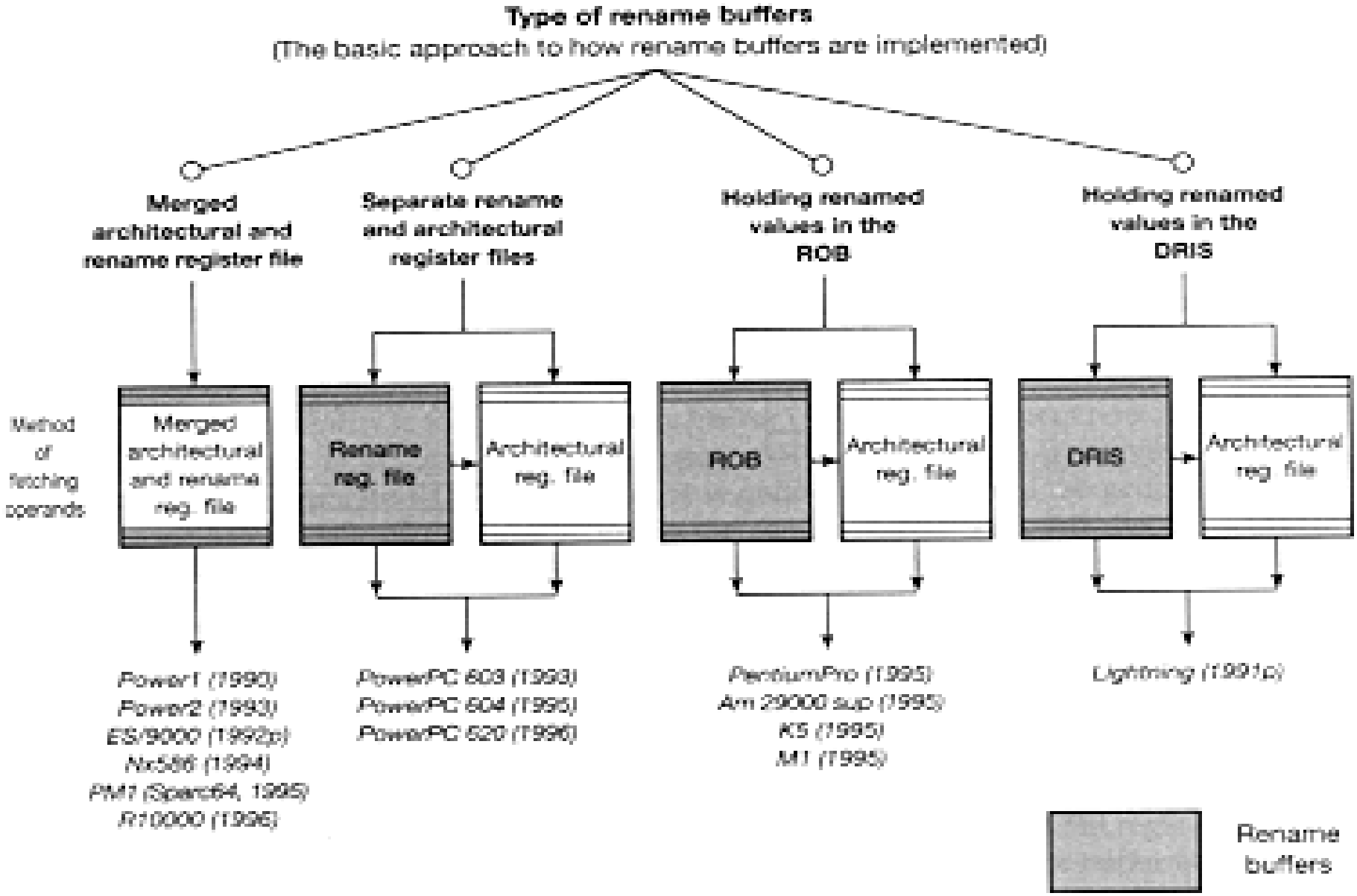


[1]The Power1 renames only FP loads
[2]The Power2 extends renaming to all FP instructions
[3]The PowerPC 601 renames only the Link and Count registers
[4]Since the Nx586 is an FX processor, it renames only FX instructions

-Layout of rename buffers



-Type of rename buffers



Rename buffers hold intermediate results

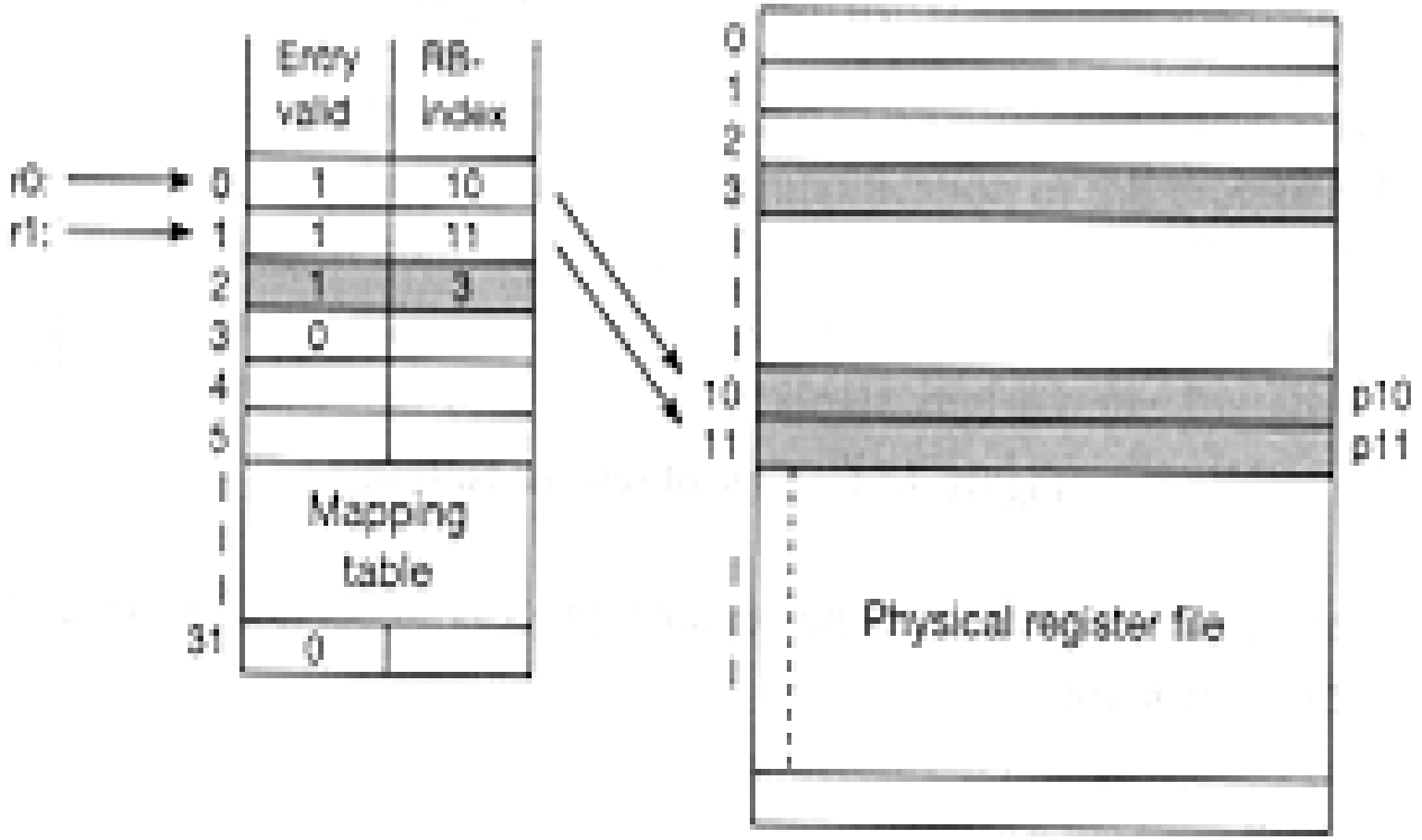
- Each time a Destination register is referred to, a new rename register is allocated to it.
- Final results are stored in the Architectural Register file
- Access both rename buffer and architectural register file to find the latest data,
  - **if found in both, the data content in rename buffer (the intermediate result) is chosen.**
- When an instruction completed (retired),
  - **(ROB) {retire only in strict program sequence}**
  - **the correspond rename buffer entry is writing into the architectural register file (as a result modifying the actual program state)**
  - **the correspond rename buffer entry can be de-allocated**

-Number of rename buffers

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	(1995)	12	8
PowerPC 620	(1996)	8	8
Renaming within the ROB			
Am29000 sup	(1995)	10	
K5	(1995)	16	
PentiumPro	(1995)	40	

-Basic mechanisms used for accessing rename buffers

- **Rename buffers with associative access (latter e.g.)**
- **Rename buffers with indexed access**
  - ✈ (always corresponds to the most recent instance of renaming)



-Operand fetch policies and Rename Rate

- rename bound: fetch operands during renaming (during instruction issue)
- dispatch bound: fetch operand during dispatching

- Rename Rate
  - **the maximum number of renames per cycle**
  - **equals the issue rate: to avoid bottlenecks.**

7.5.8 Detailed example of renaming

- renaming:
  - **mul r2, r0, r1**
  - **ad r3, r1, r2**
  - **sub r2, r0, r1**

- format:
  - **op Rd, Rs1, Rs2**

- Assume:
  - **separate rename register file,**
  - **associative access, and**
  - **operand fetching during renaming**

Structure of the rename buffers and their supposed initial contents

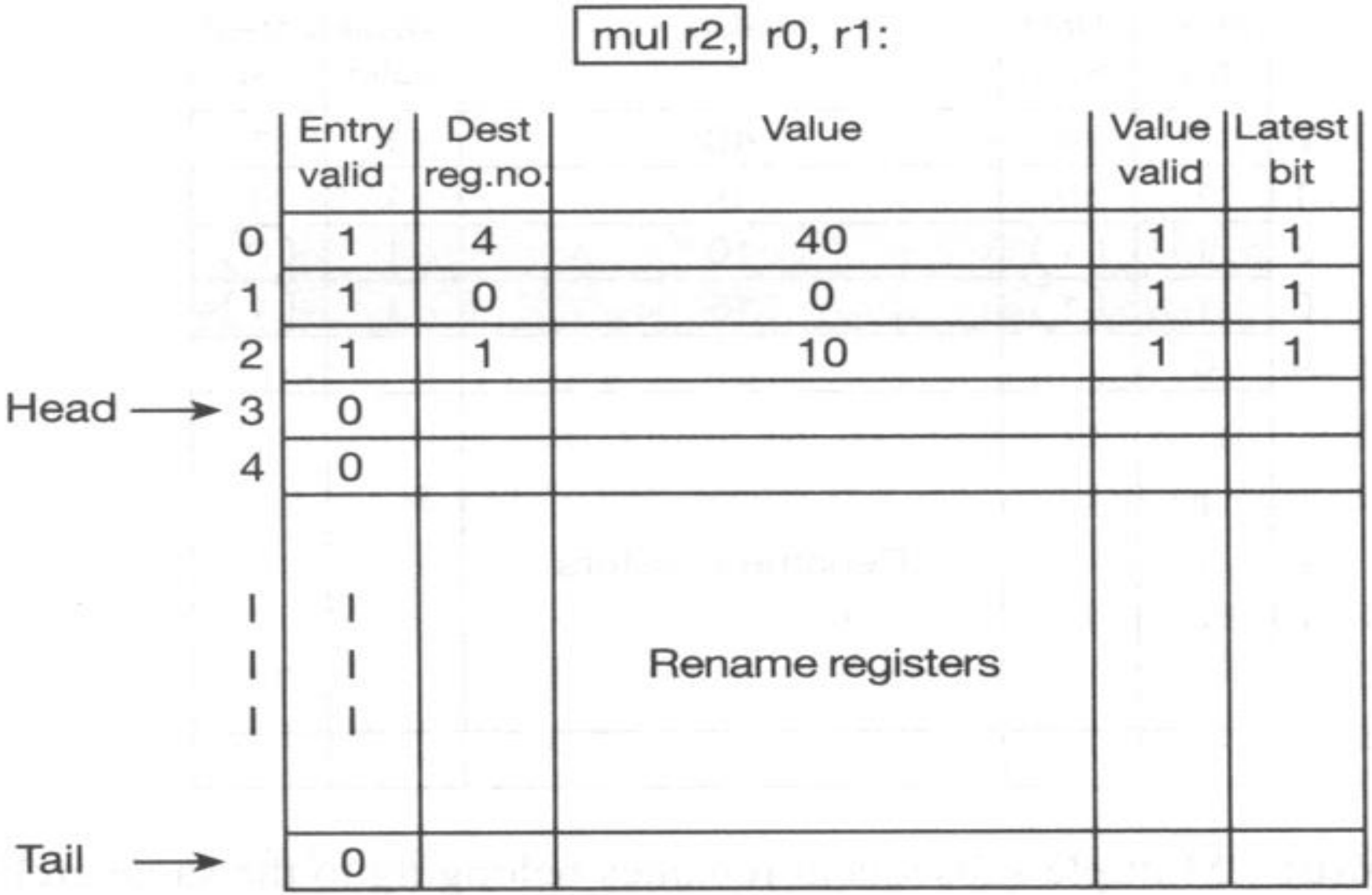
- Latest bit: the most recent rename 1, previous 0

	Entry valid	Dest reg.no.	Value	Value valid	Latest bit
0	1	4	40	1	1
1	1	0	0	1	1
2	1	1	10	1	1
3	0				
4	0				
!	!		Rename registers		
!	!				
!	!				

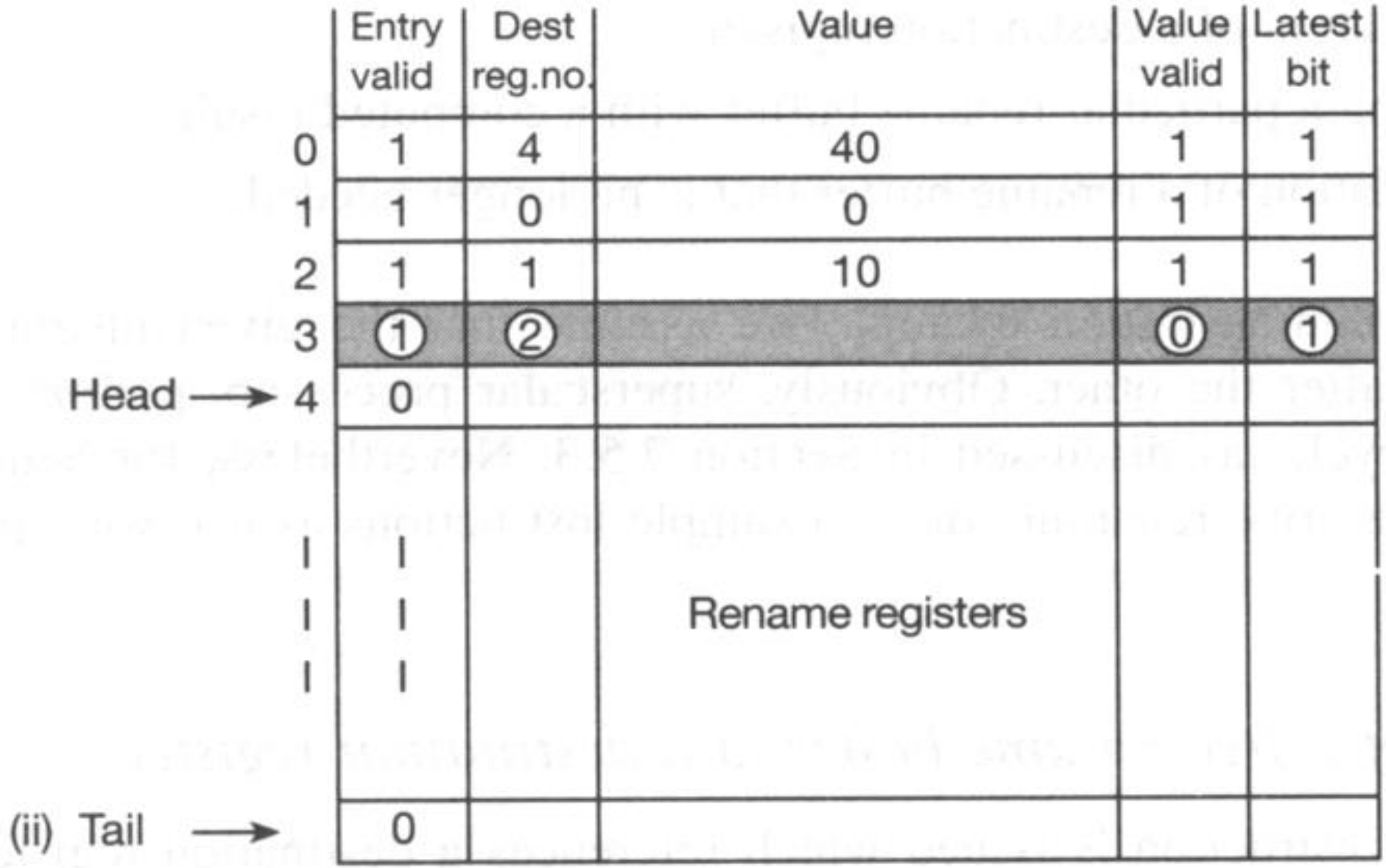
Renaming steps

- Allocation of a free rename register to a destination register
- Accessing valid source register value or a register value that is not yet available
- Re-allocation of destination register
- Updating a particular rename buffer with a computed result
- De-allocation of a rename buffer that is no longer needed.

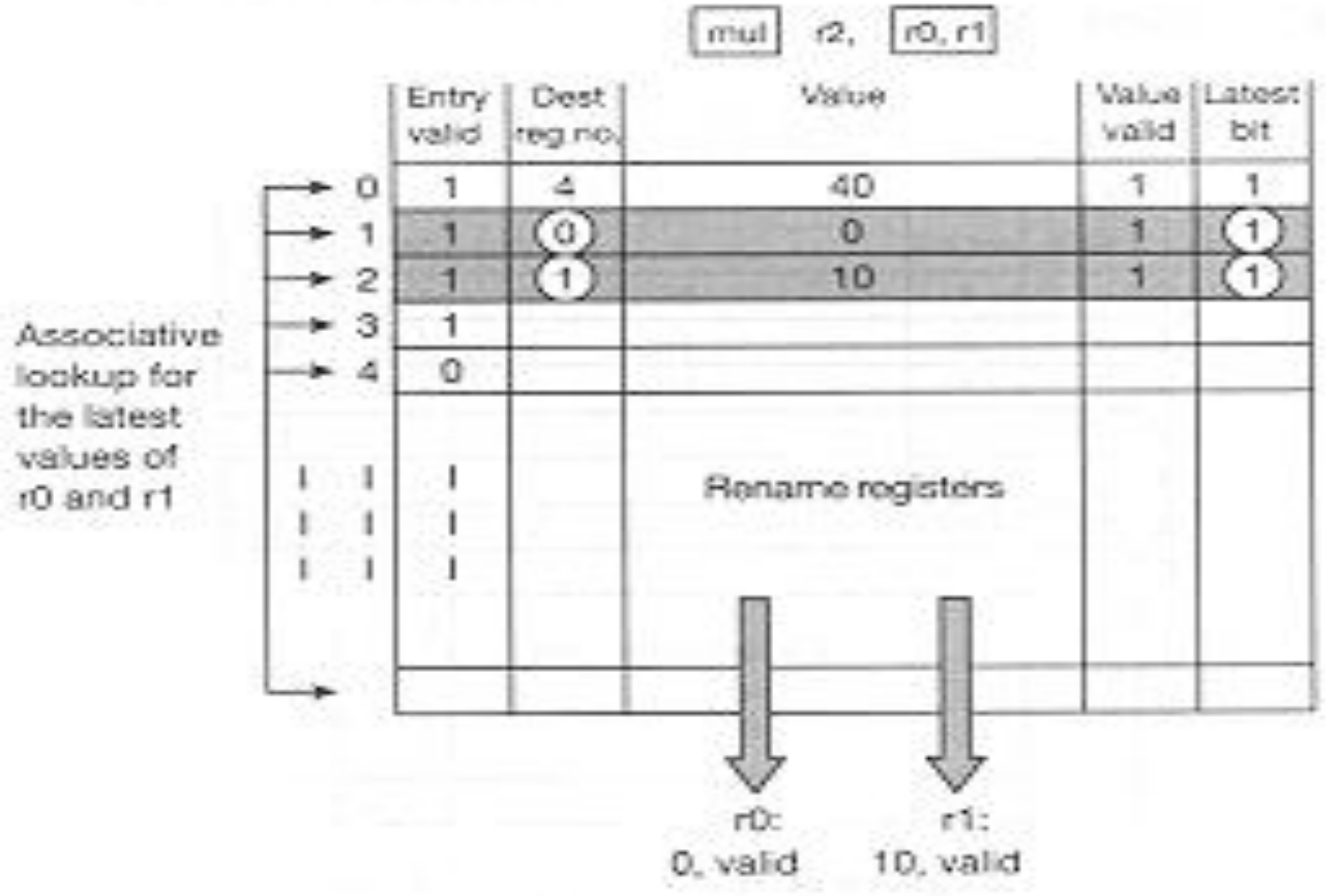
Allocation of a new rename buffer to destination register (circular buffer: Head and Tail) (before allocation)



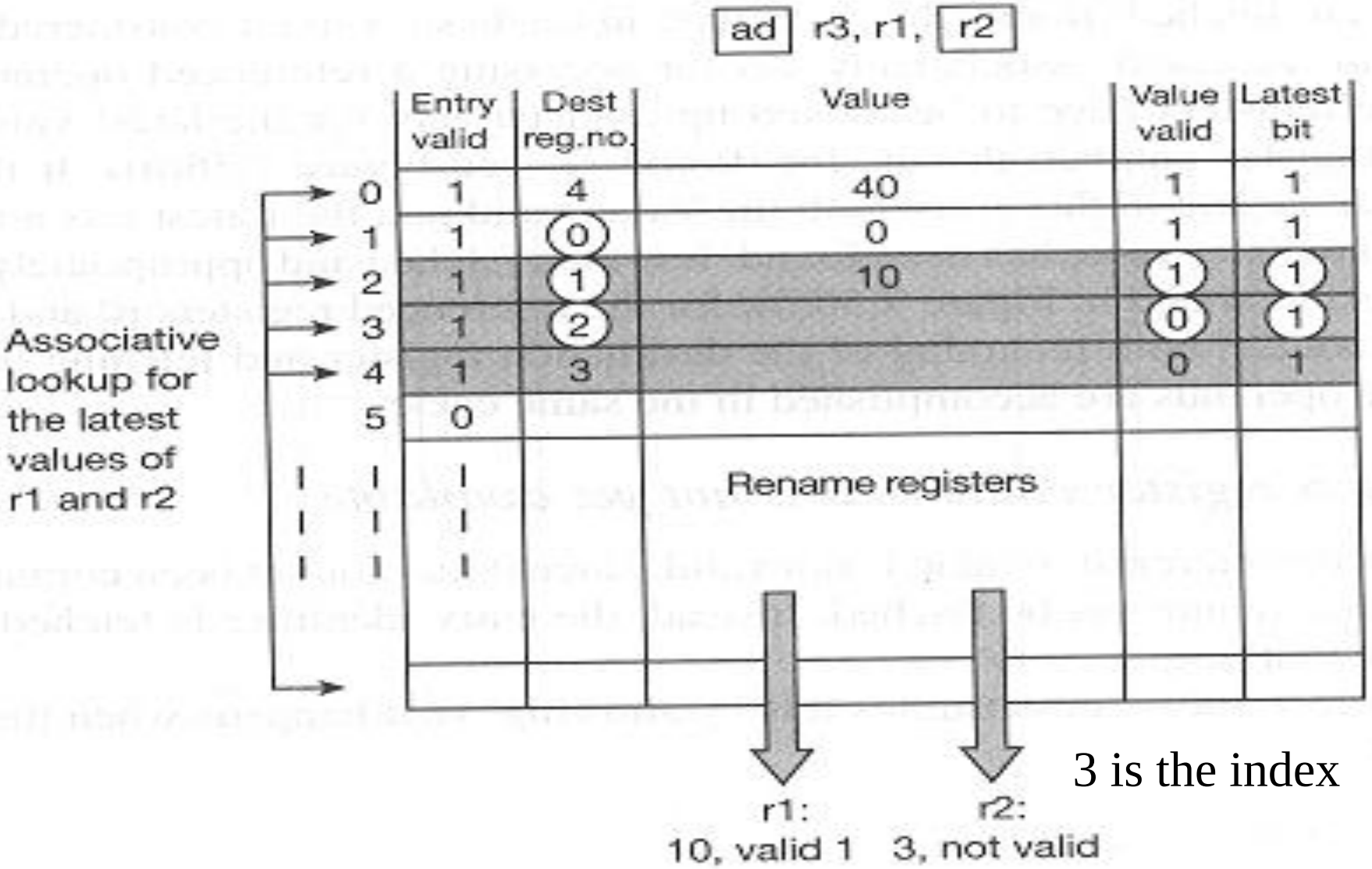
(After allocation) of a destination register



Accessing abailable register values



Accessing a register value that is not yet available



Re-allocate of r2 (a destination register)

sub r2, r0, r1

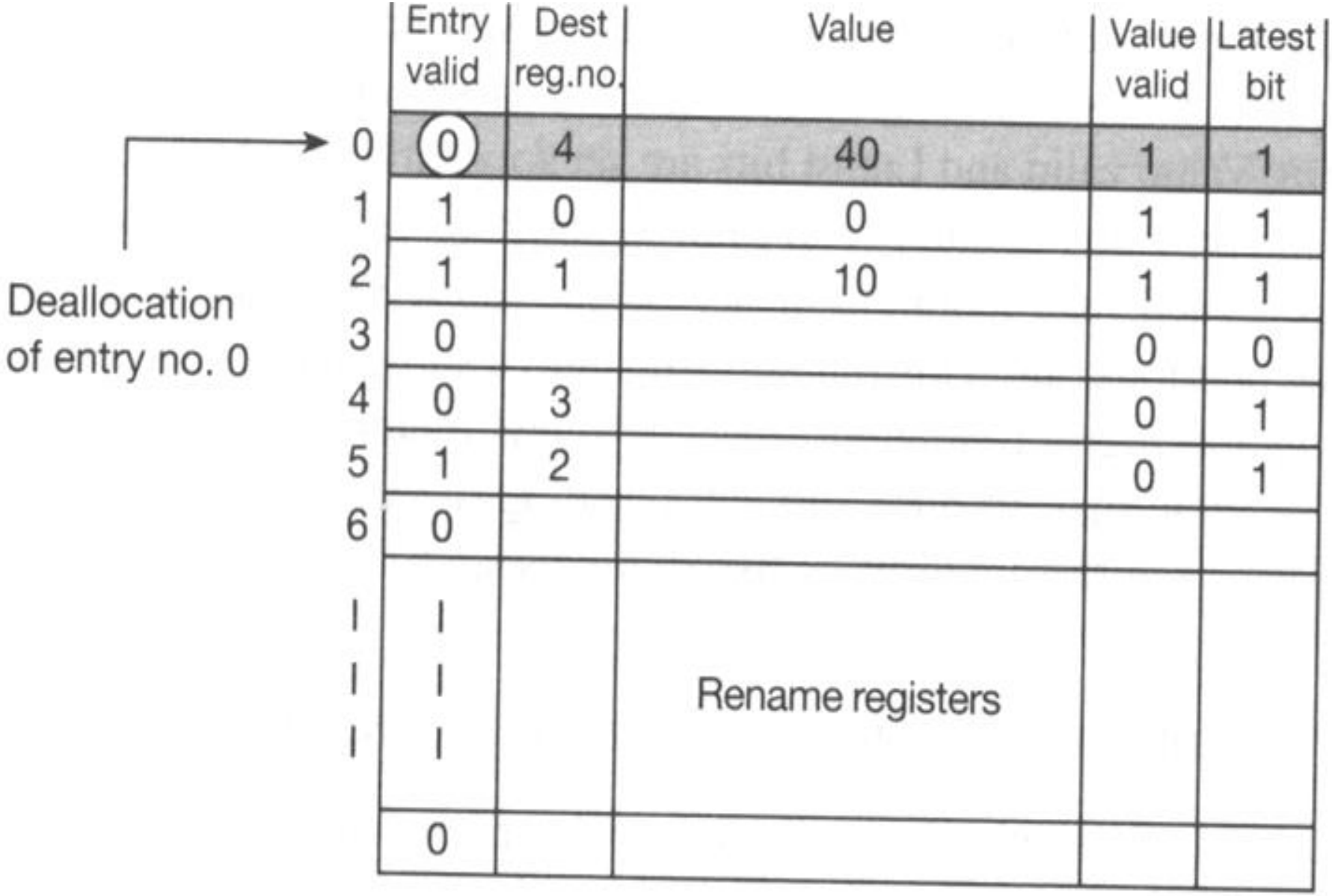
	Entry valid	Dest reg.no.	Value	Value valid	Latest bit
0	1	4	40	1	1
1	1	0	0	1	1
2	1	1	10	1	0
3	1	2		0	0
4	0	3		0	1
5	1	(2)		(0)	(1)
6	0				
I	I		Rename registers		
I	I				
I	I				
	0				

Updating the rename buffers with computed result of {mul r2, r0, r1} (register 2 with the result 0)

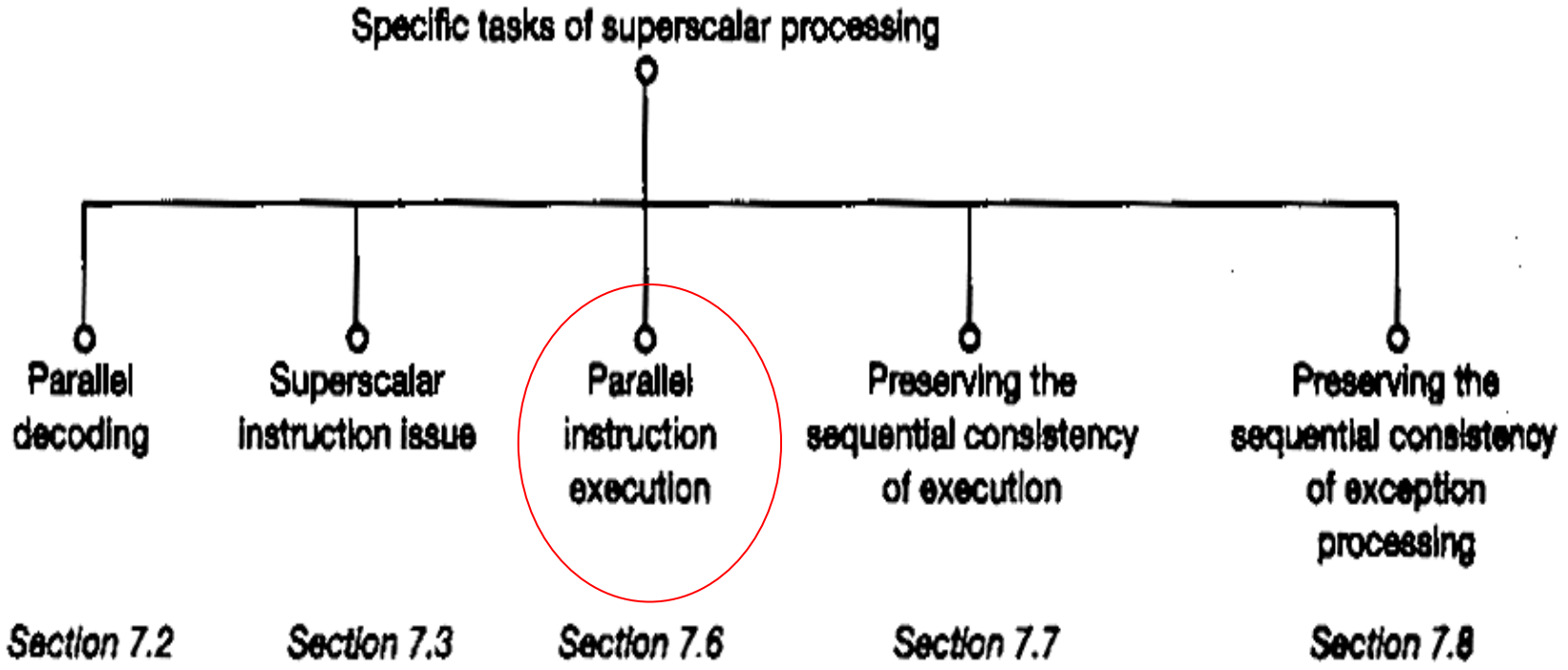
Result 0 entry no 3 →

	Entry valid	Dest reg.no.	Value	Value valid	Latest bit
0	1	4	40	1	1
1	1	0	0	1	1
2	1	1	10	1	1
3	1	2	0	(1)	0
4	0	3		0	1
5	1	2		0	1
6	0				
I	I		Rename registers		
I	I				
I	I				
	0				

Deallocation of the rename buffer no. 0 (ROB retires instructions) (update tail pointer)



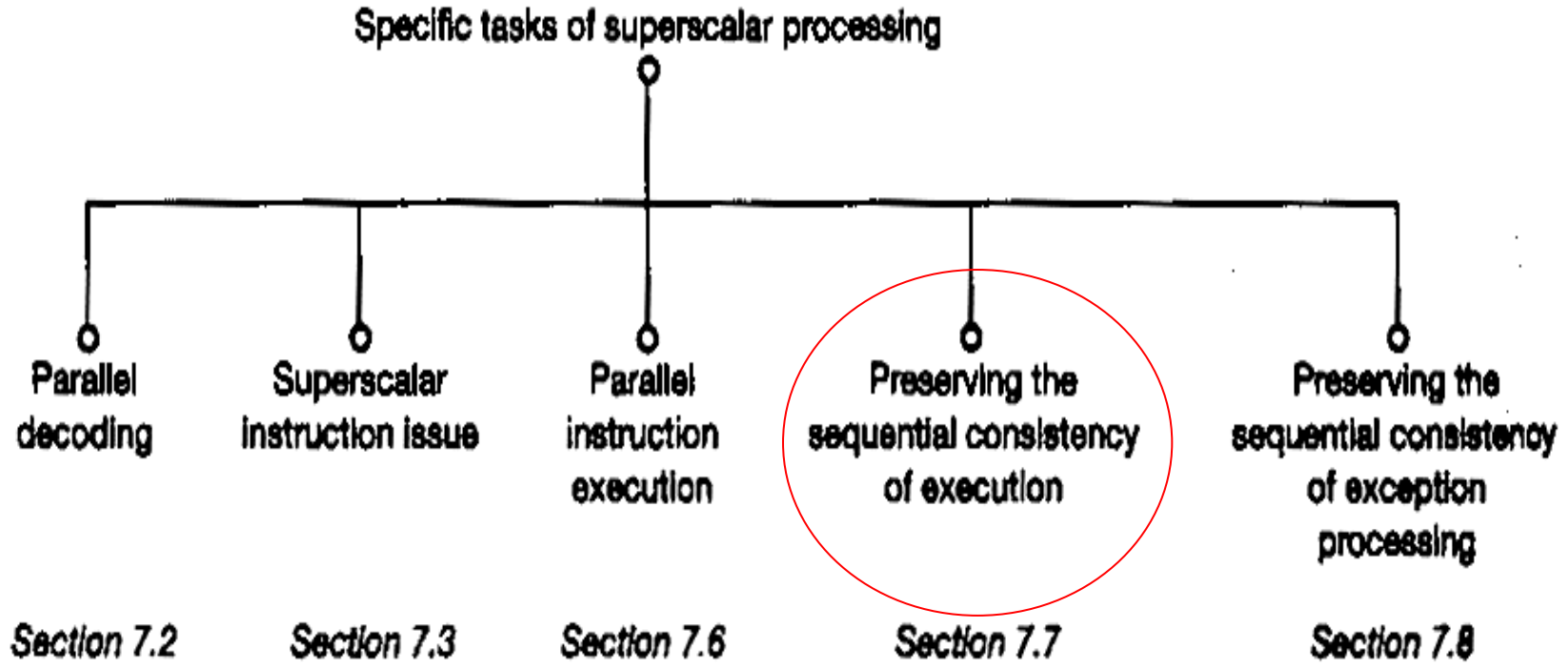
7.1 Specific tasks of superscalar processing



7.6 Parallel Execution

- Executing several instruction in parallel
  - **instructions will generally be finished in out-of-program order**
- to finish
  - **operation of the instruction is accomplished,**
  - **except for writing back the result into**
    - the architectural register or
    - memory location specified, and/or
    - updating the status bits
- to complete
  - **writing back the results**
- to retire (ROB)
  - **write back the results, and**
  - **delete the completed instruction from the last ROB entry**

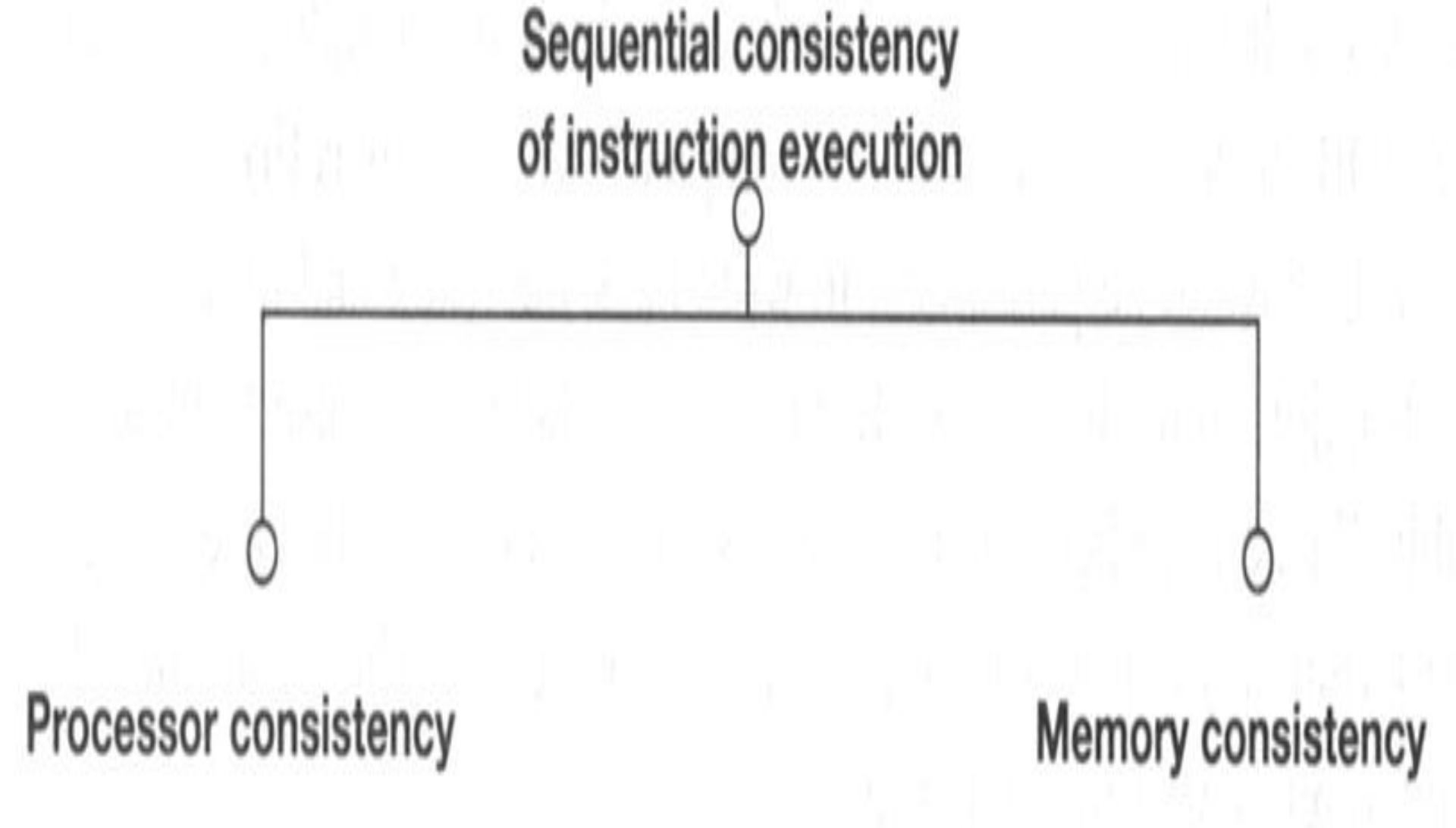
7.1 Specific tasks of superscalar processing



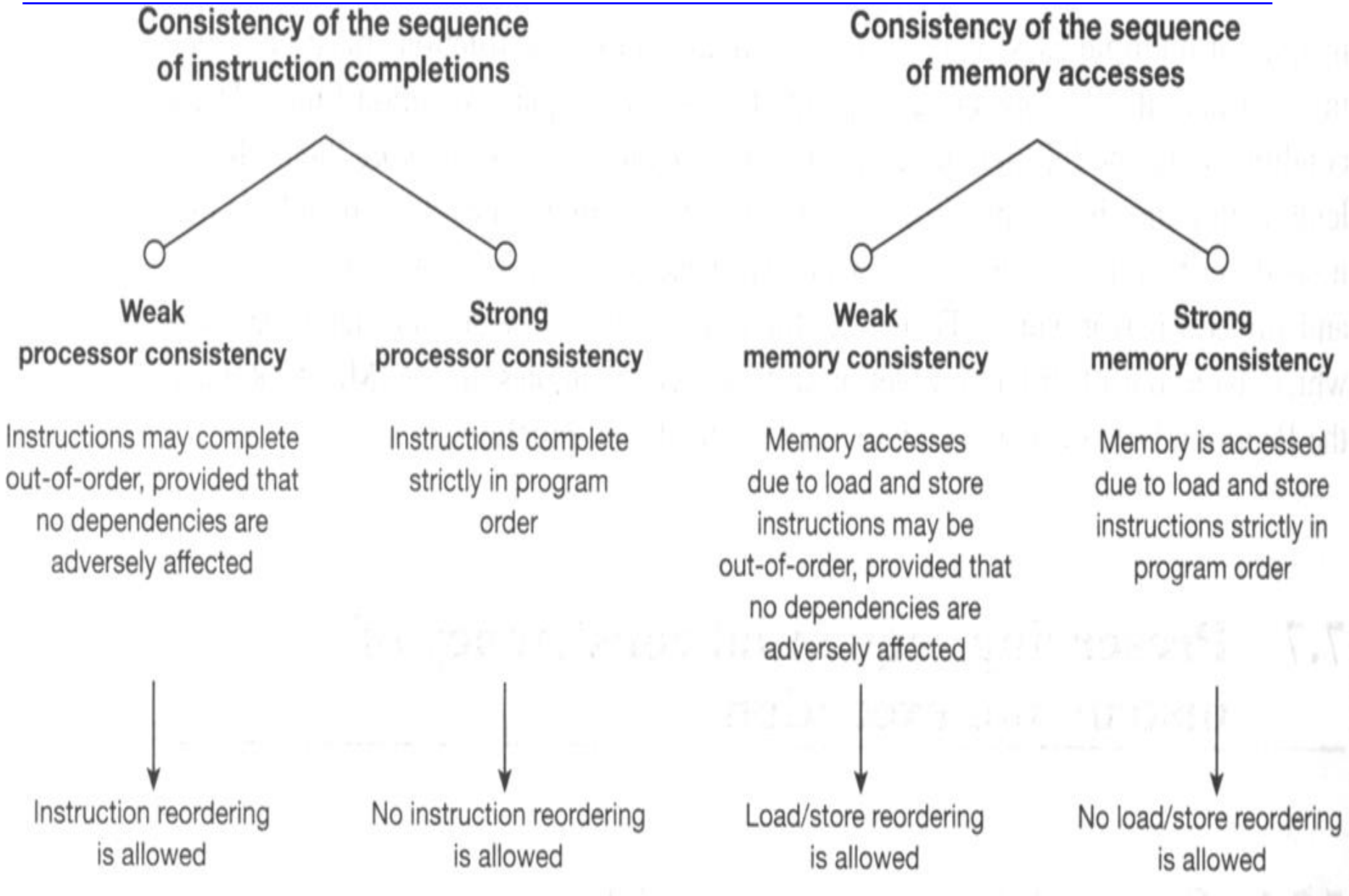
7.7 Preserving Sequential Consistency of instruction execution //

- Multiple EUs operating in parallel, the overall instruction execution should
  >> mimic sequential execution
  - **the order in which instruction are completed**
  - **the order in which memory is accessed**

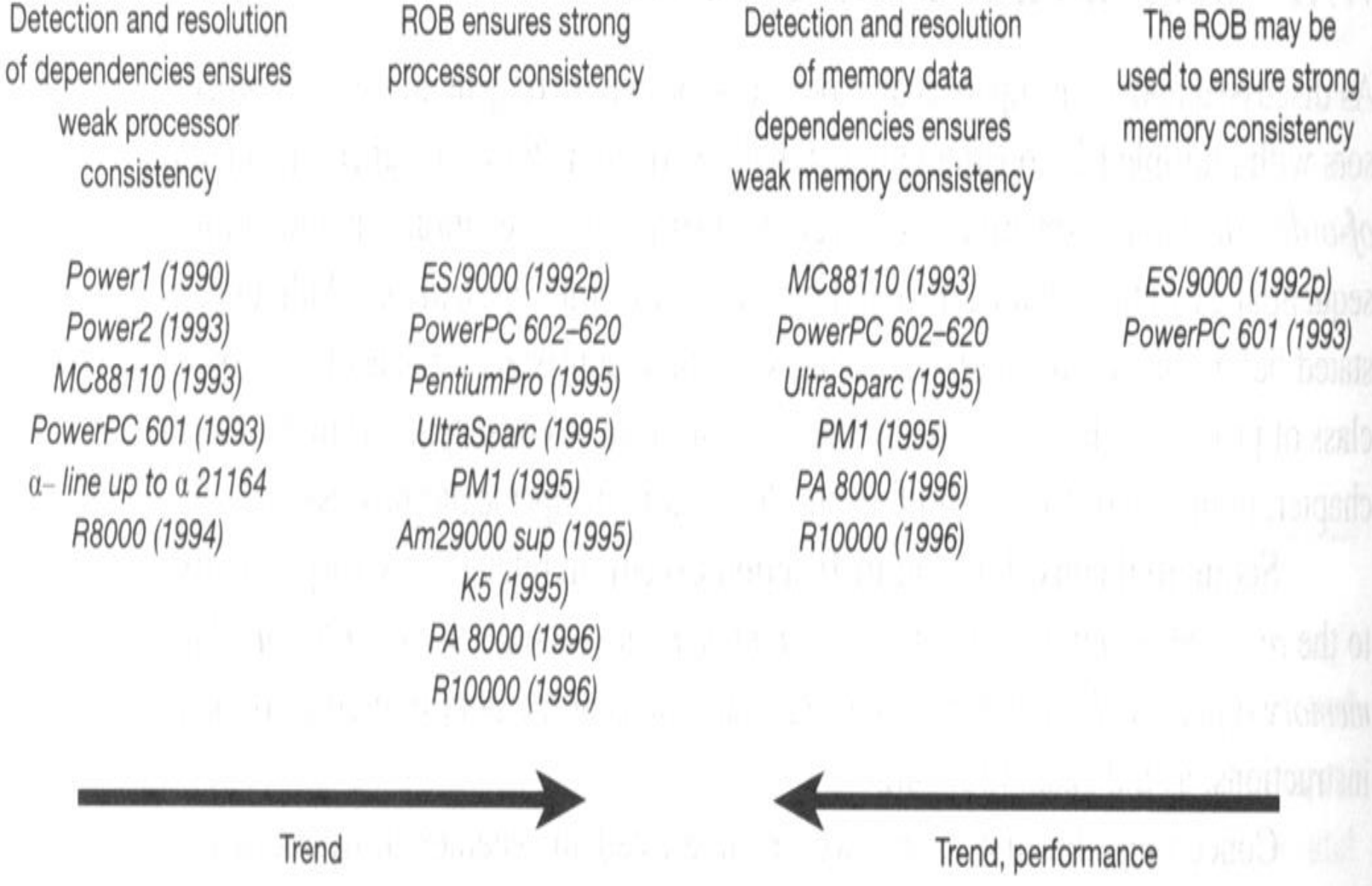
Sequential consistency models



Consistency relate to instruction completions or memory access



Trend and performance



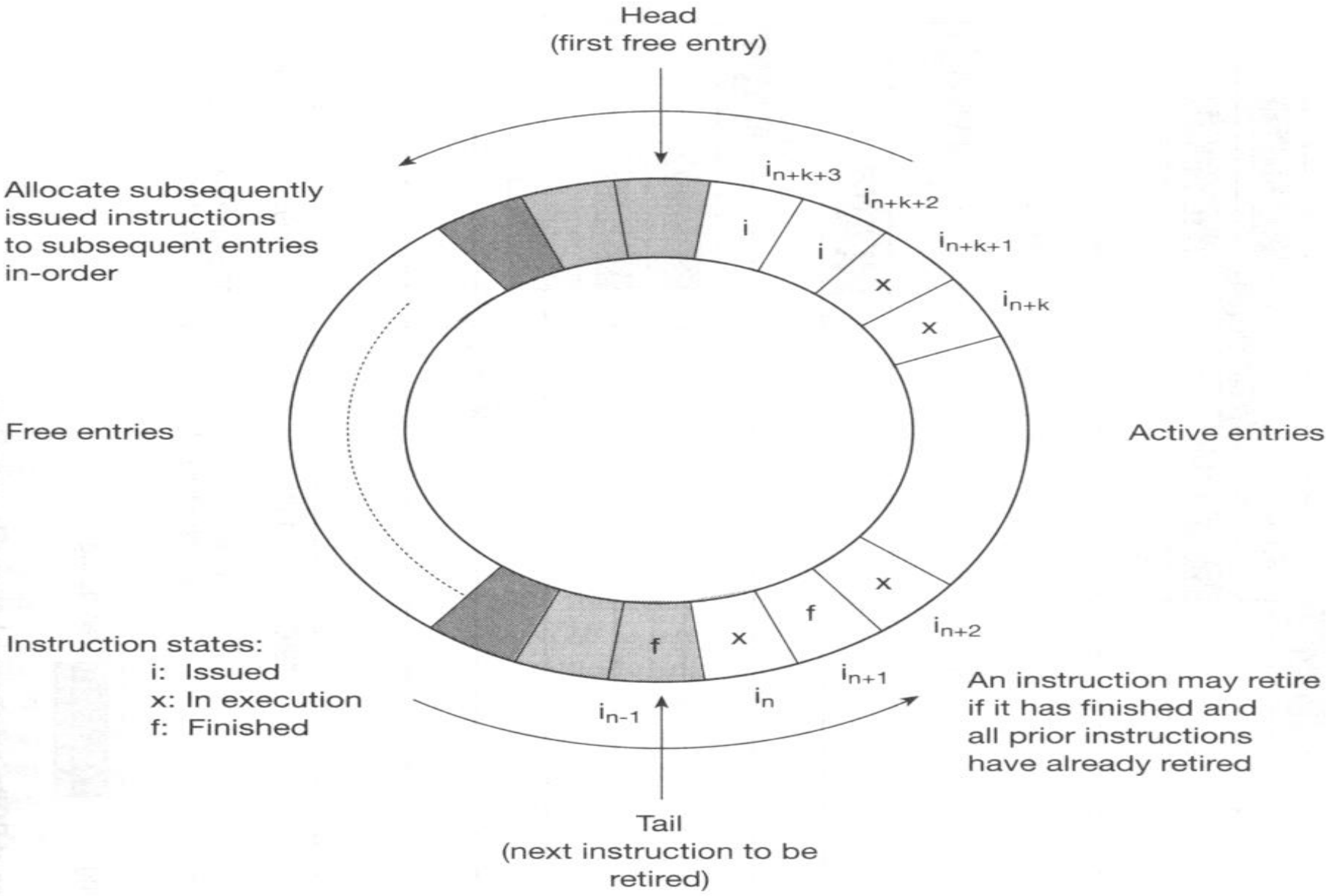
Allows the reordering of memory access

- it permits load/store reordering
  - **either loads can be performed before pending stores, or vice versa**
  - **a load can be performed before pending stores only IF**
  - **none of the preceding stores has the same target address as the load**
- it makes Speculative loads or stores feasible
  - **When addresses of pending stores are not yet available,**
  - **speculative loads avoid delaying memory accesses, perform the load anywhere.**
  - **When store addresses have been computed, they are compared against the addresses of all younger loads.**
  - **Re-load is needed if any hit is found.**
- it allows cache misses to be hidden
  - **if a cache miss, it allows loads to be performed before the missed load; or it allows stores to be performed before the missed store.**

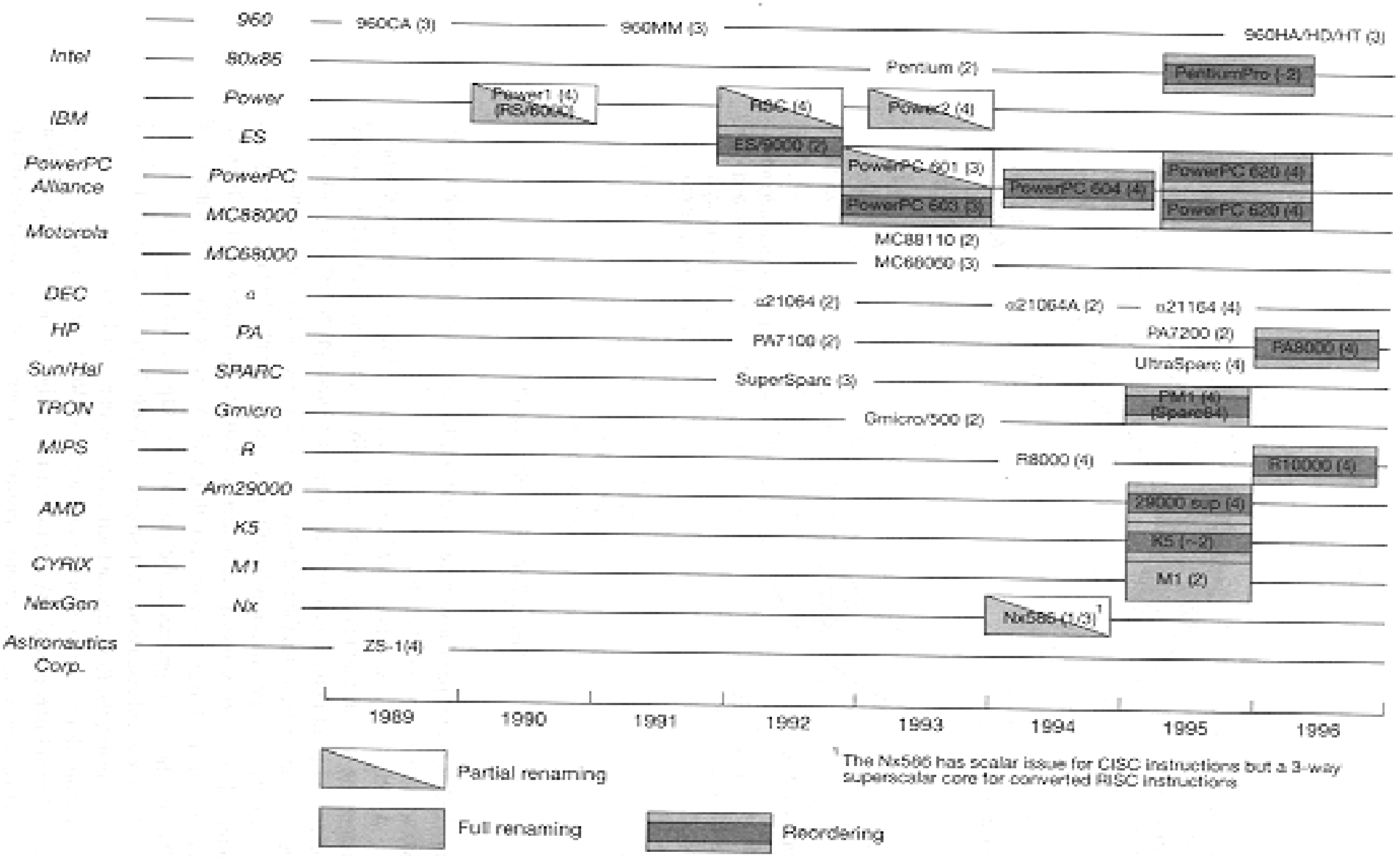
Using Re-Order Buffer (ROB) for Preserving: The order in which instruction are <completed>

- 1. Instruction are written into the ROB in strict program order:
  - **One new entry is allocated for each active instruction**
- 2. Each entry indicates the status of the corresponding instruction
  - **issued (i), in execution (x), already finished (f)**
- 3. An instruction is allowed to retire only if it has finished and all previous instruction are already retired.
  - **retiring in strict program order**
  - **only retiring instructions are permitted to complete, that is, to update the program state:**
    - by writing their result into the referenced architectural register or memory

Principle of the ROB {Circular Buffer}



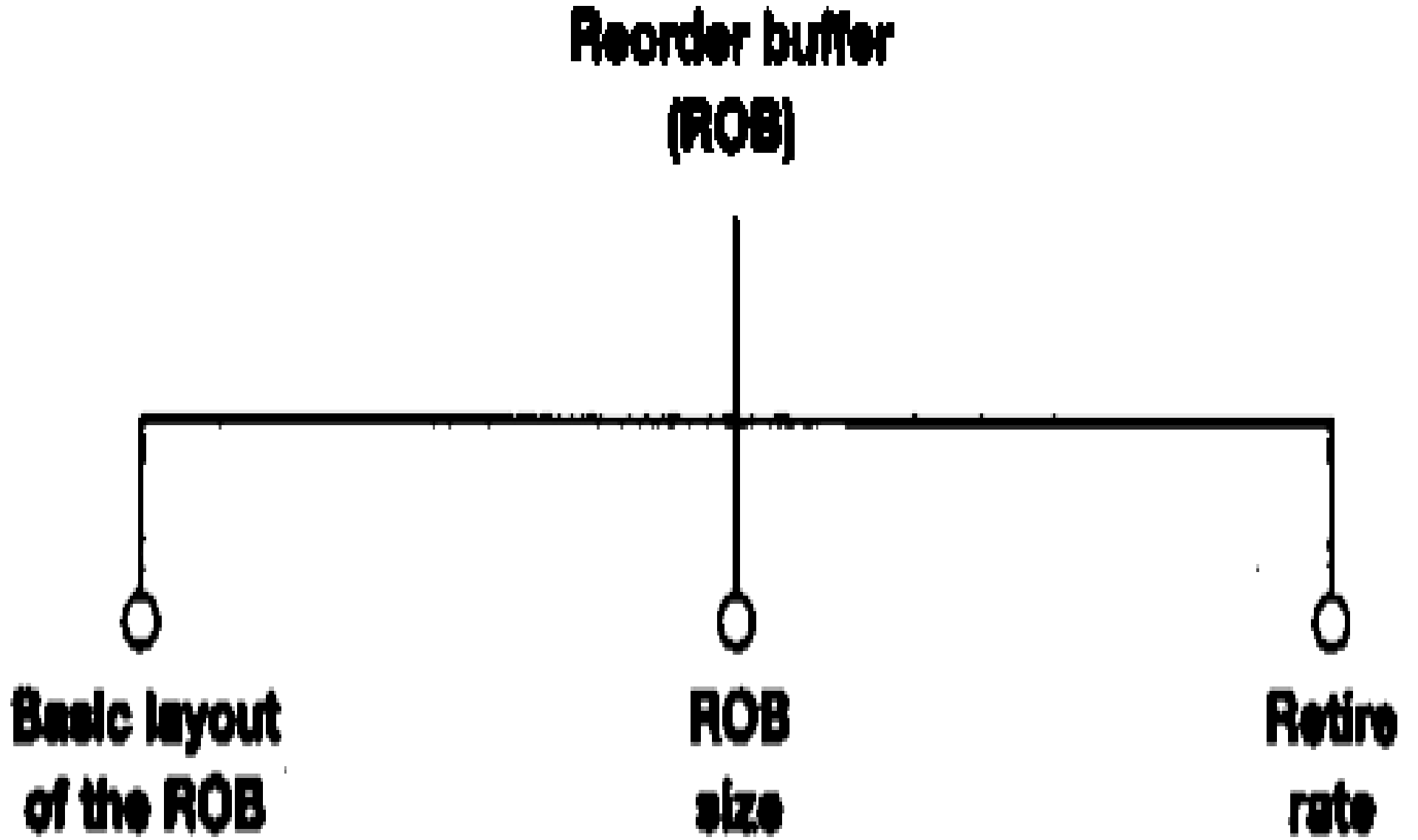
Introduction of ROBs in commercial superscalar processors



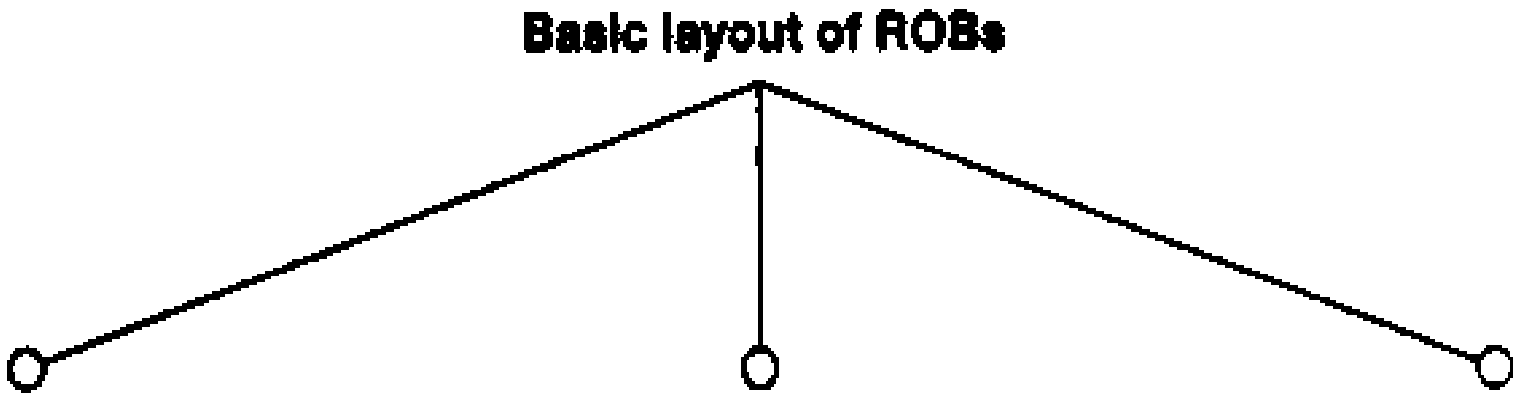
Use ROB for speculative execution

- Guess the outcome of a branch and execution the path
  - **before the condition is ready**
- 1. Each entry is extended to include a speculative status field
  - **indicating whether the corresponding instruction has been executed speculatively**
- 2. speculatively executed instruction are not allow to retire
  - **before the related condition is resolved**
- 3. After the related condition is resolved,
  - **if the guess turn out to be right, the instruction can retire in order.**
  - **if the guess is wrong, the speculative instructions are marked to be cancelled.**
    Then, instruction execution continue with the correct instructions.

Design space of ROBs



Basic layout of ROBs



ES/9000
PowerPC 603 (1993)
PowerPC 604 (1995)
PowerPC 620 (1996)
PM1 (1995)
R10000 (1996)

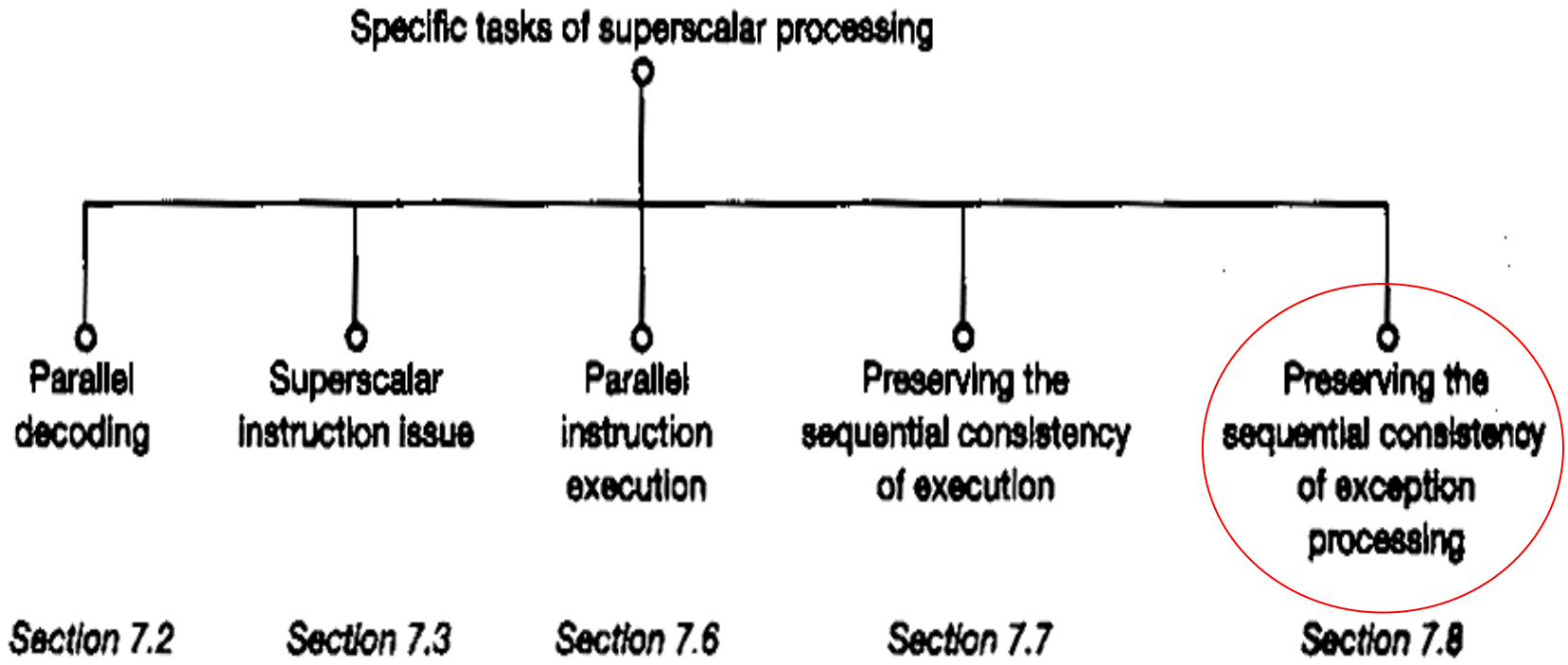
Am29000 sup (1995)
K5 (1995)
PentiumPro (1995)

Lightning (1991p)

ROB implementation details

	ROB size	*Issue rate*	*Retire rate*	*Intermediate results stored*	*Designation*
ES/9000 (1992p)	32	2	2	No	Completion control logic
PowerPC 602 (1995)	4	2	1	n.a.	Completion unit
PowerPC 603 (1993)	5	3	2	No	Completion buffer
PowerPC 604 (1995)	16	4	4	No	ROB
PowerPC 620 (1996)	16	4	4	No	ROB
PentiumPro (1995)	40	3	3	Yes	ROB
Am29000 sup (1995)	10	4	2	Yes	ROB
K5 (1995)	16	4	4	Yes	ROB
PM1 (Sparc64, 1995)	64	4	4	No	Precise state unit
UltraSparc (1995)	n.a.	4	n.a.	n.a.	Completion unit
PA 8000 (1996)	56	4	4	Yes	Instruction reorder buffer
R10000 (1996)	32	4	4	No	Active list

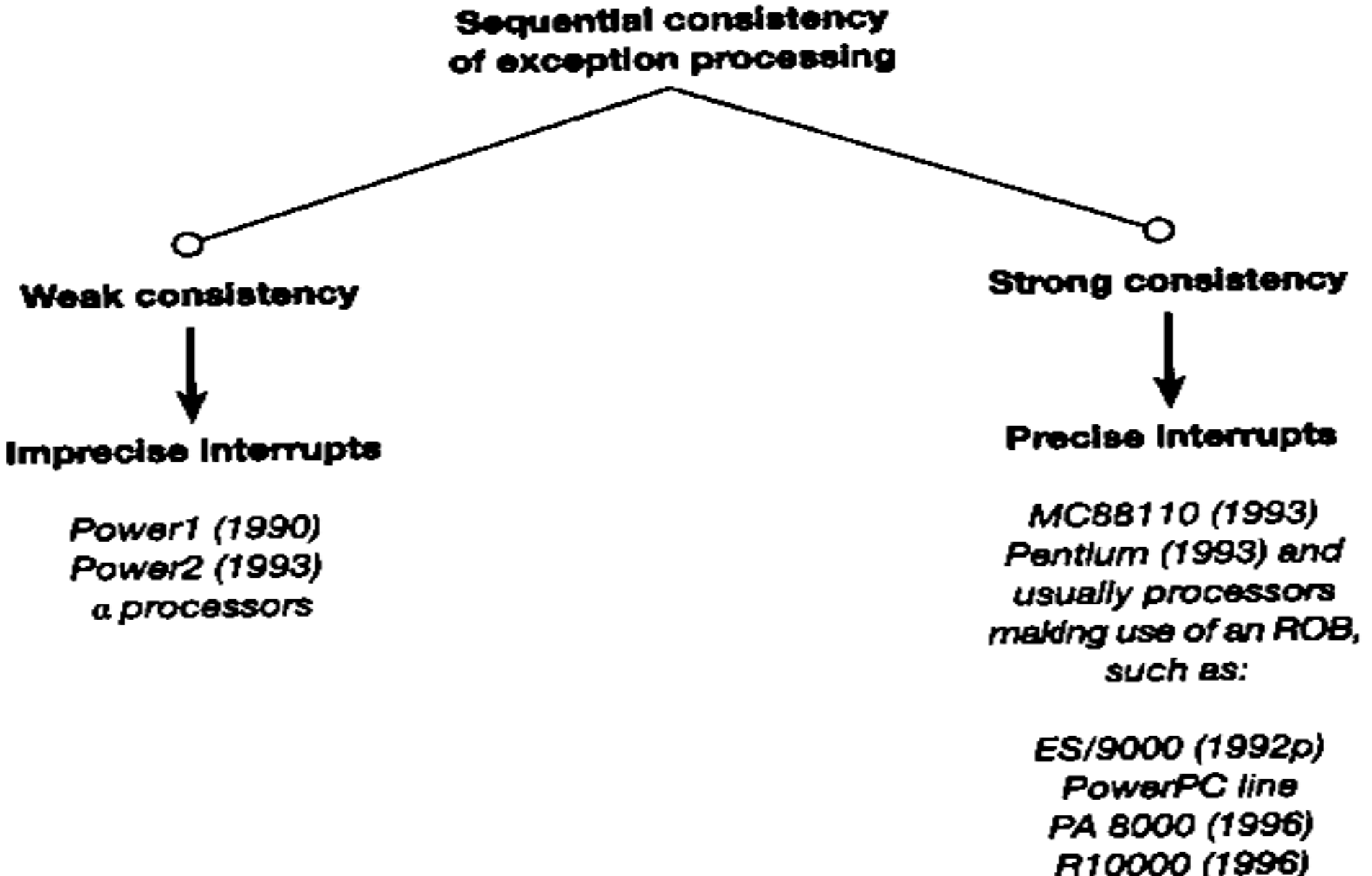
7.1 Specific tasks of superscalar processing



7.8 Preserving the Sequential consistency of exception processing

- When instructions are executed in parallel,
  - **interrupt request, which are caused by exceptions arising in instruction <execution>,**
  - **are also generated out of order.**
- If the requests are acted upon immediately,
  - **the requests are handled in different order than in a sequential operation processor**
  - **called imprecise interrupts**
- Precise interrupts: handling the interrupts in consistent with the state of a sequential processor

Sequential consistency of exception processing



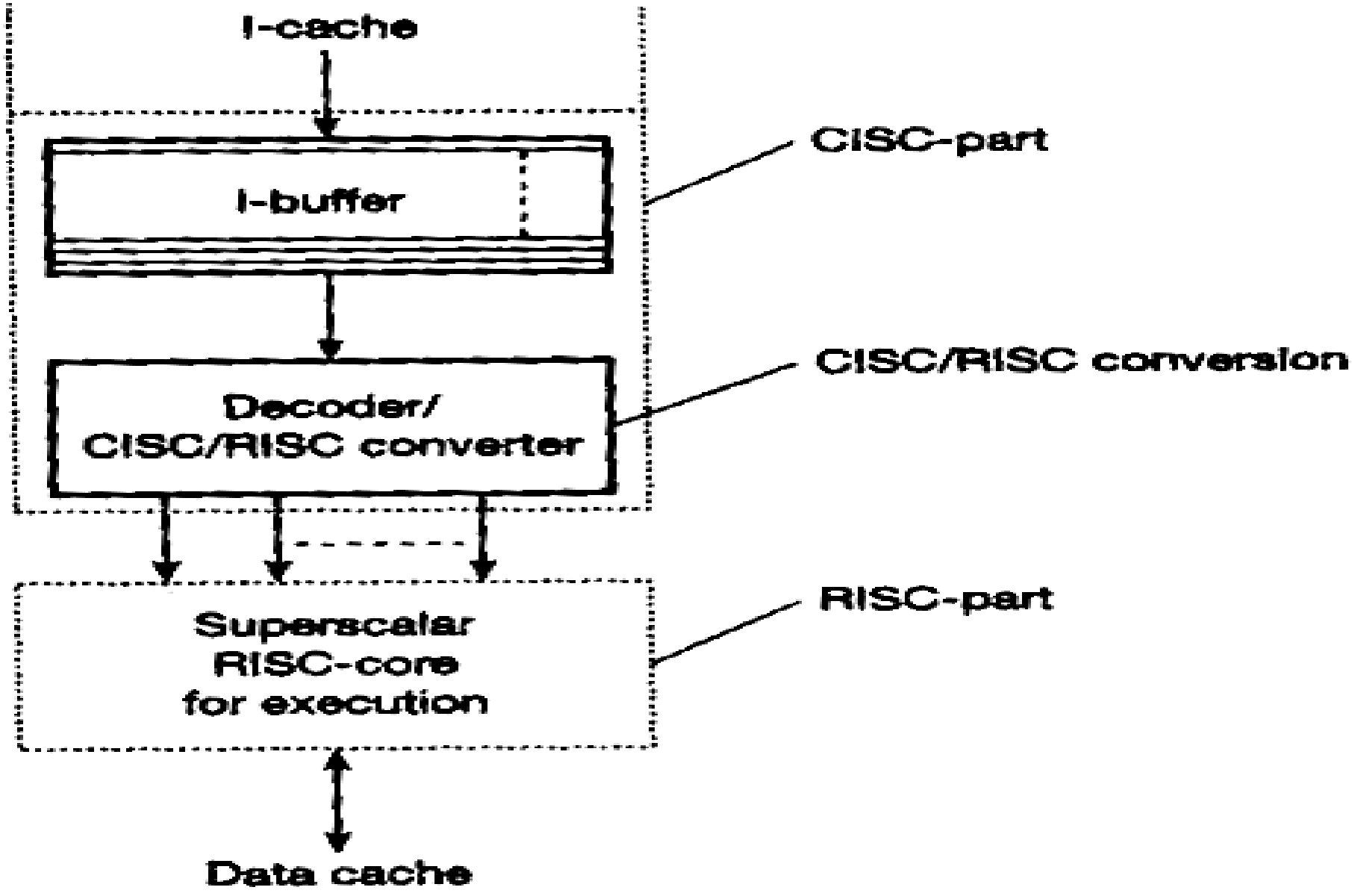
Use ROB for preserving sequential order of interrupt requests

- Interrupts generated in connection with instruction execution
  - **can handled at the correct point in the execution,**
  - **by accepting interrupt requests only when the related instruction becomes the next to retire.**

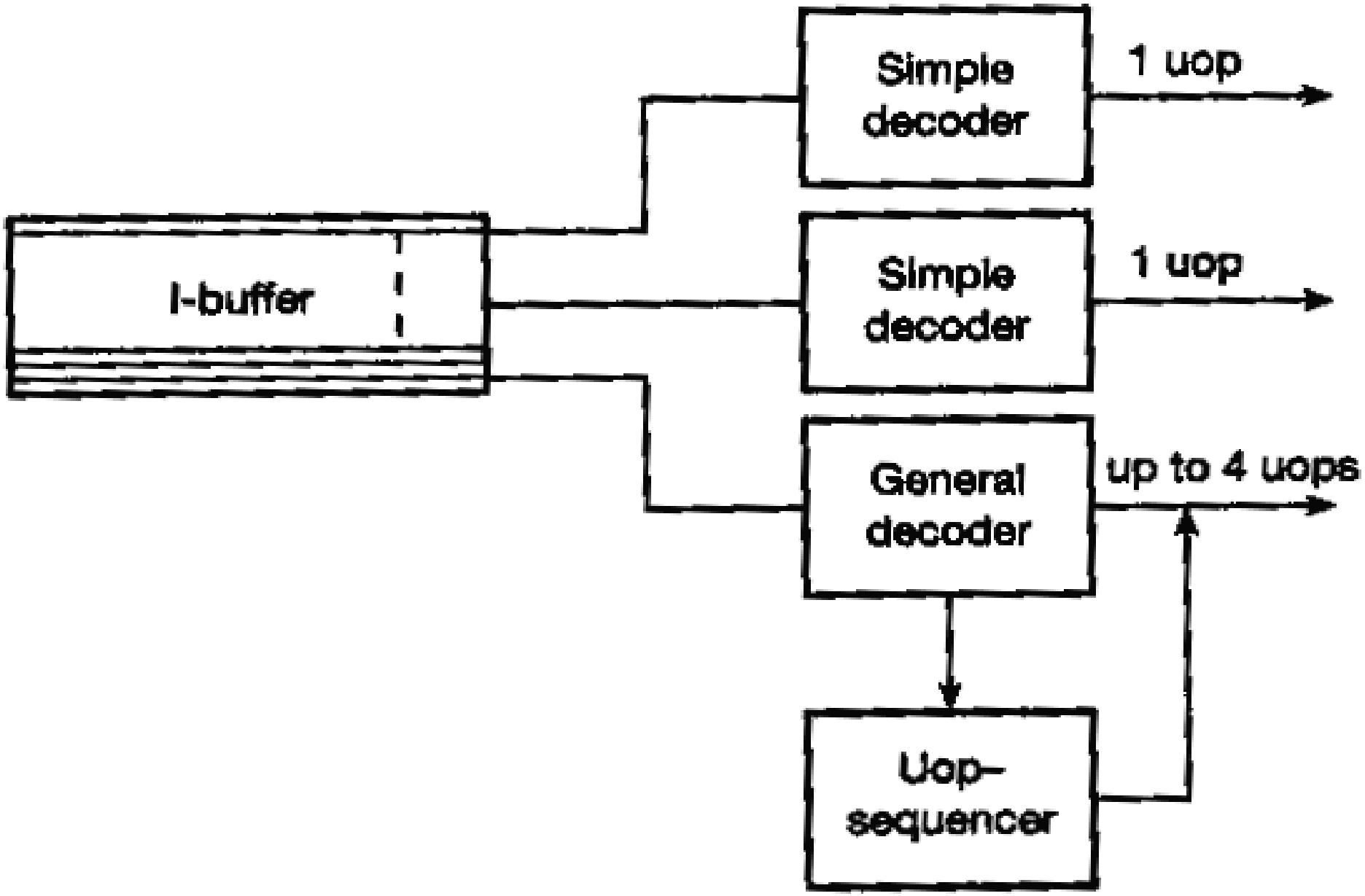
7.9 Implementation of superscalar CISC processors using superscalar RISC core

- 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

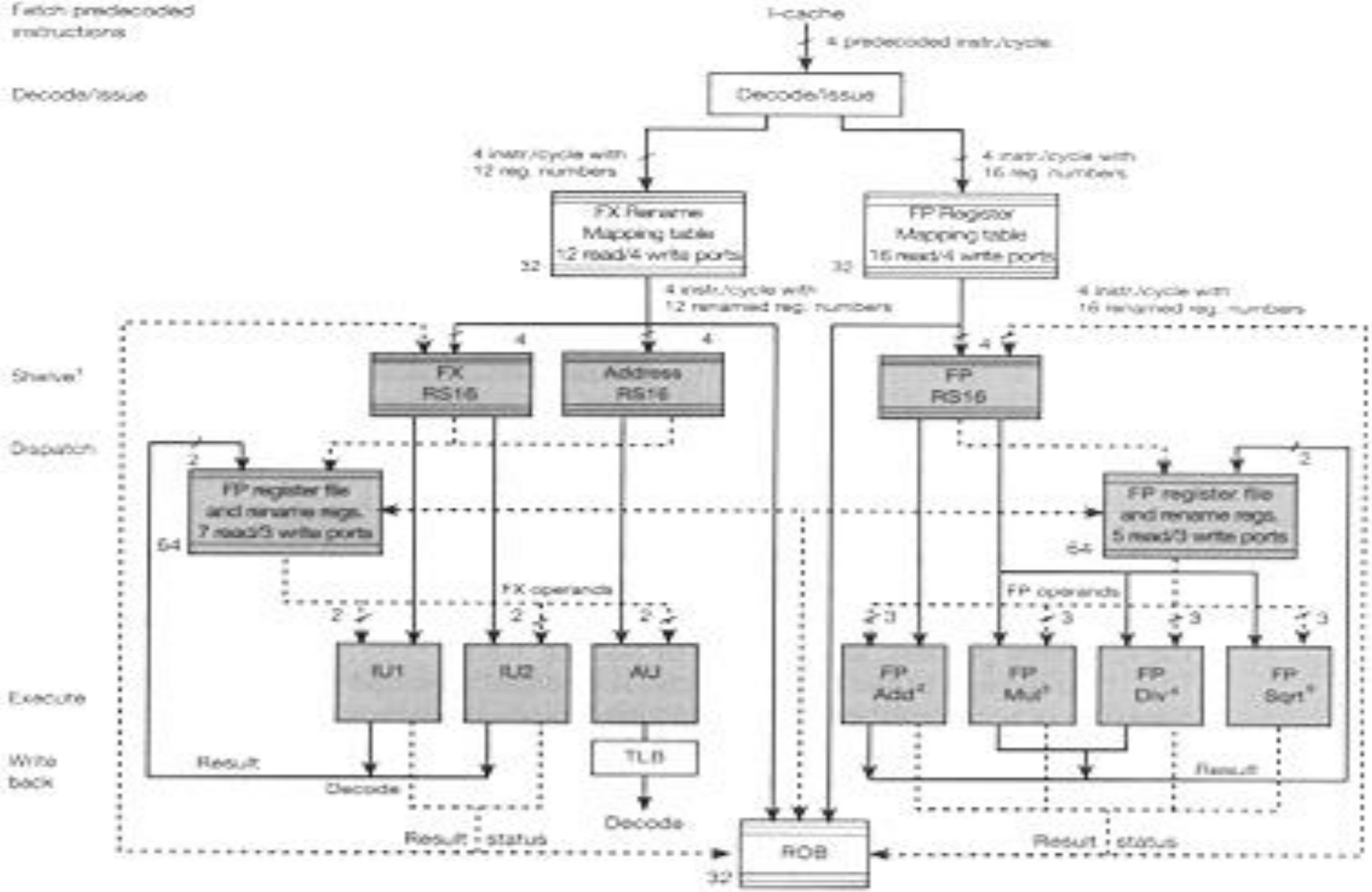
The peinciple of superscalar CISC execution using a superscalar RISC core



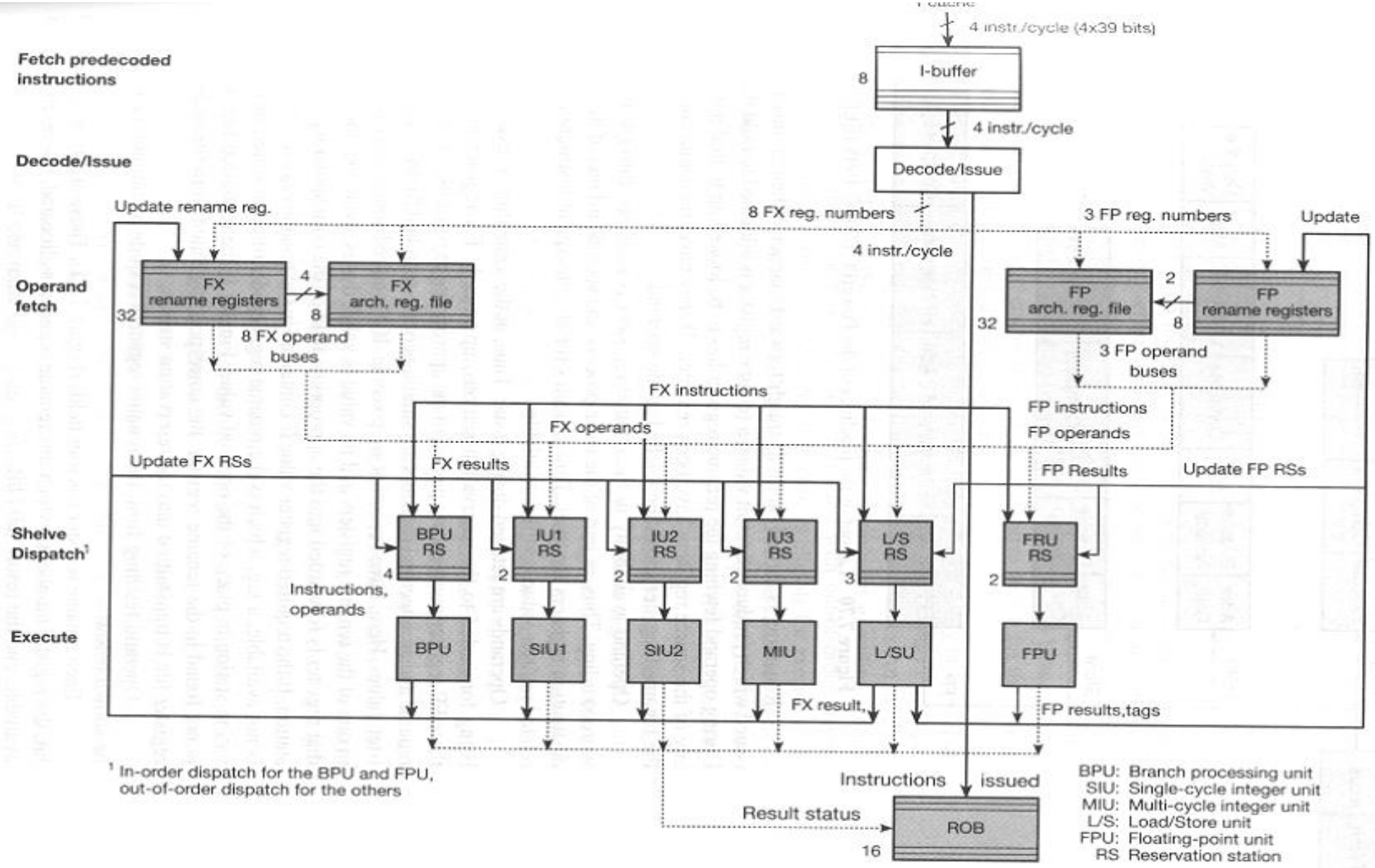
PentiumPro: Decoding/converting CISC instructions to RISC operations (are done in program order)



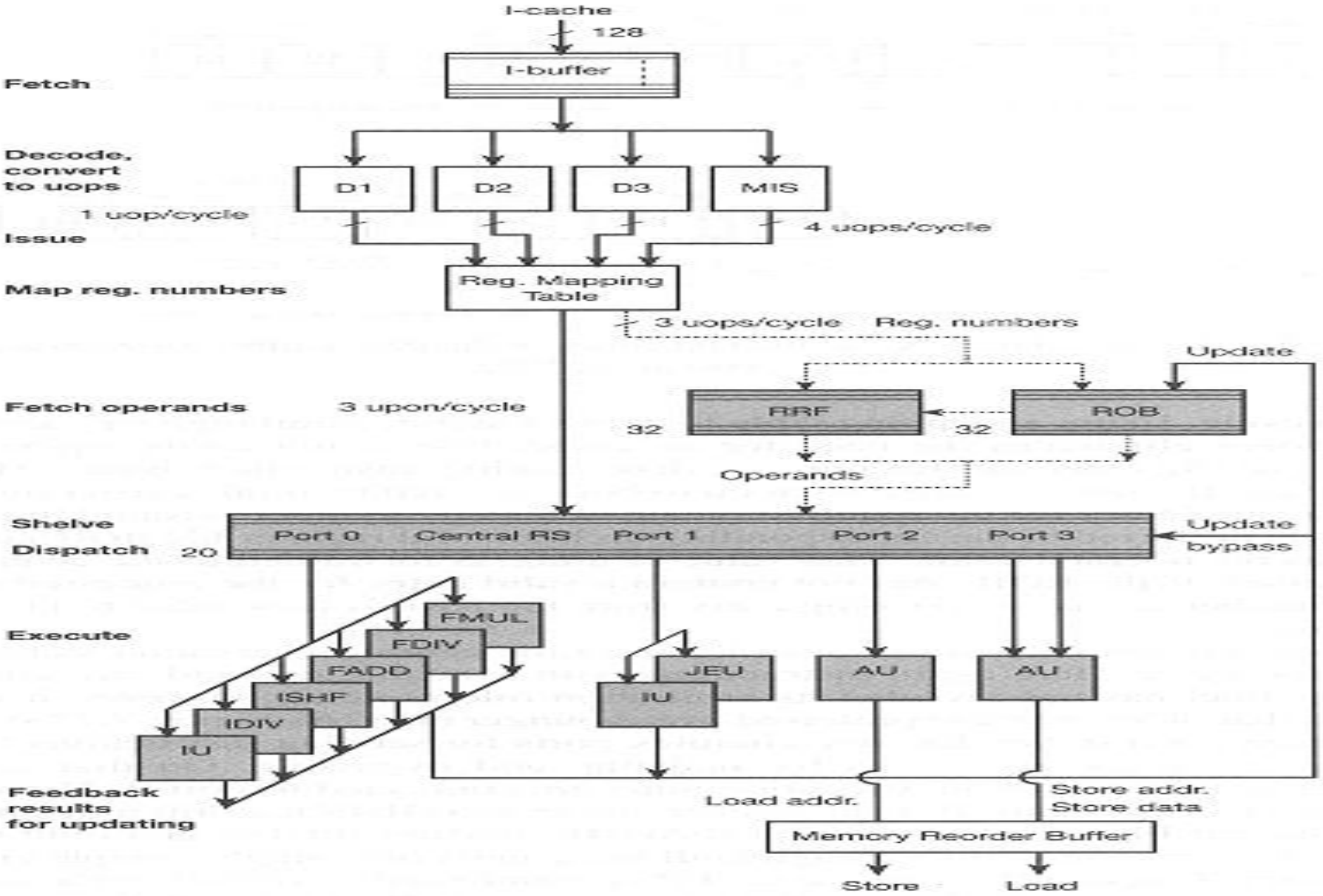
Case Studies: R10000
Core part of the micro-architecture of the R10000



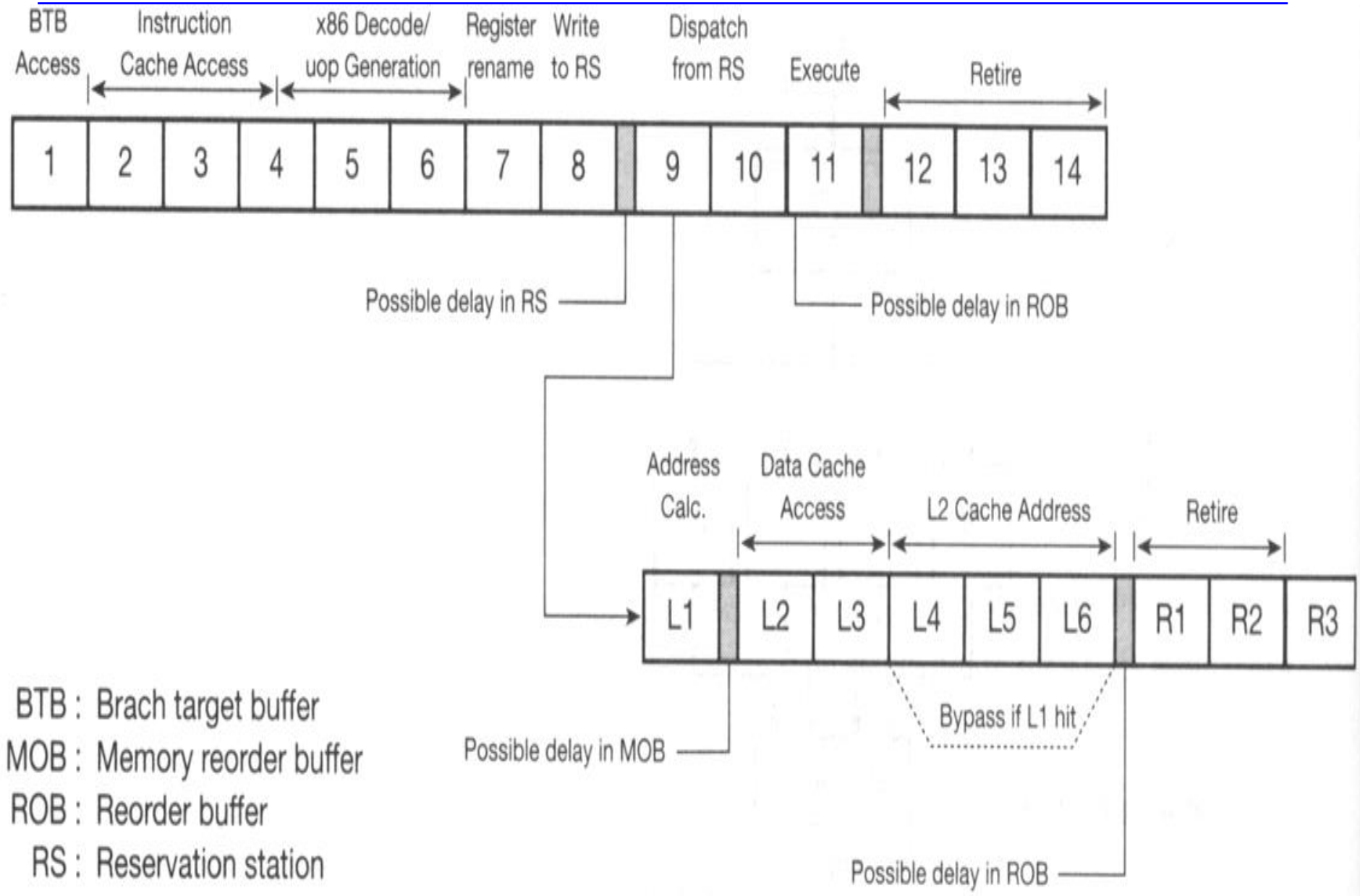
Case Studies: PowerPC 620



Case Studies: PentiumPro
Core part of the micro-architecture



PentiumPro Long pipeline: Layout of the FX and load pipelines



Department of Information Technology
National Institute of Technology Karnataka, Surathkal

IT301 : PARALLEL COMPUTING

Topic: AMDAHL's LAW

By,
Thanmayee,
Adhoc Faculty,
Department of IT,
NITK, Surathkal

Parallel Processing → Speed Up

- Speed up is the factor by which the time is reduced compared to a single processor.

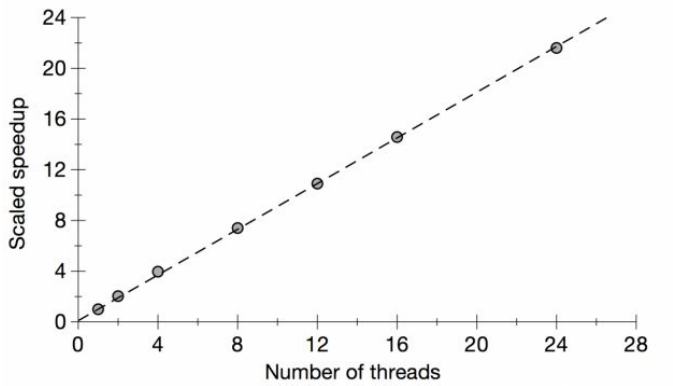
$$\textbf{Speedup} = \frac{\textbf{Execution time using one processor}}{\textbf{Execution time using multiple processors}}$$

Example: It time taken to perform a task (T_1) using **one processor is 1 unit** then time taken to perform the same task (T_N) on **N processors is 1/N unit.**

Then Speed up is = $T_1/T_N = \frac{1}{1/N}$

Parallel Processing → Speed Up

- Speedup is linear !
- That means as the number of processors increase, the speed up should also increase.
- Ideally this is what is to be achieved.
- But it is not practically achieved.
- Thus Amdahl's law gives us a better insight.



Amdahl's Law

- Speed up depends on the portion of program that is serial and portion of program that is parallel or enhanced.
- So the PARALLEL program (T) will have SERIAL part (S) + PARALLEL part (P)
- In single processor Time taken by Parallel program $T_1 = S+P$
- In multiple processor (say n processors) Time taken by $T_n = S+P/n$
- If time required for parallel program on single processor is 1 unit. Then time taken to execute Serial portion is S=1-P

$$\text{Speed up} = \frac{T_1}{T_n} \qquad \text{FINALLY} \qquad \text{Speed up} = \frac{1}{(1-P) + P/n}$$

THANK YOU