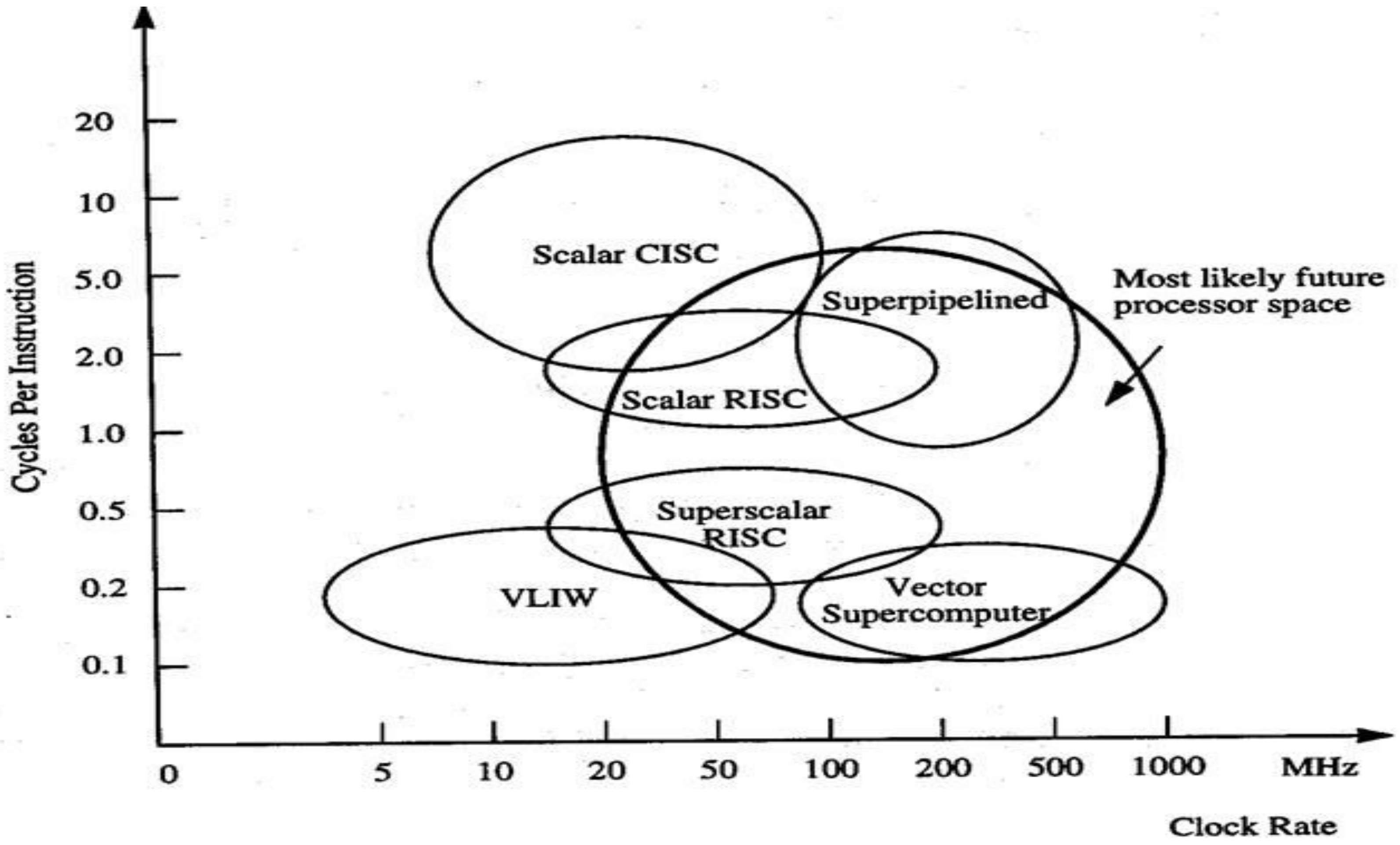
Superscalar & Superpipelined Processor

P. K. Roy
Asst. Professor
Siliguri Institute of Technology

Modern Processor Families



Design space of modern processor families.

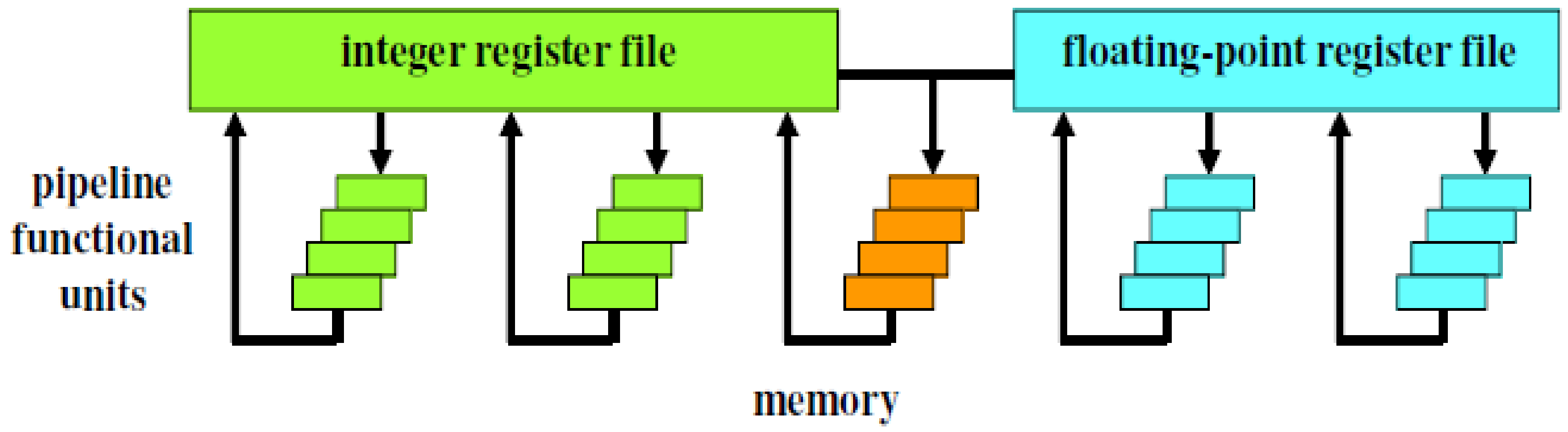
Scalar Processor

- Single Instruction Stream Single Data stream.
- A CPU that performs computations on one number or set of data at a time, typically integers or floating point numbers.
- Can take up some time.
- Only one instruction is issued per cycle, and only one completion of instruction is expected from the pipeline per cycle.
- Has a one-cycle latency for a simple operation.
- Has a one-cycle Latency between instruction issues.

Superscalar Processor

- 1st invented in 1987.
- Superscalar processing is the ability to initiate multiple instructions during the same clock cycle.
- A superscalar machine is able to execute multiple instructions independently and concurrently in multiple pipelines.
- Superscalar architecture exploit the potential of ILP (Instruction Level Parallelism).
- Applicable to both RISC & CISC, but usually in RISC.
- A superscalar processor typically fetches multiple instructions at a time and then attempts to find nearby instructions that are independent of one another and can therefore be executed in parallel. If the input to one instruction depends on the output of a preceding instruction, then the latter instruction cannot complete execution at the same time or before the former instruction.
- Superscalar performs only one pipeline stage per clock cycle in each parallel pipeline.
- The effective CPI of a superscalar processor should be less than that of a generic scalar RISC processor.
- Clock rates of scalar RISC and superscalar RISC machines are similar.

Superscalar Processor

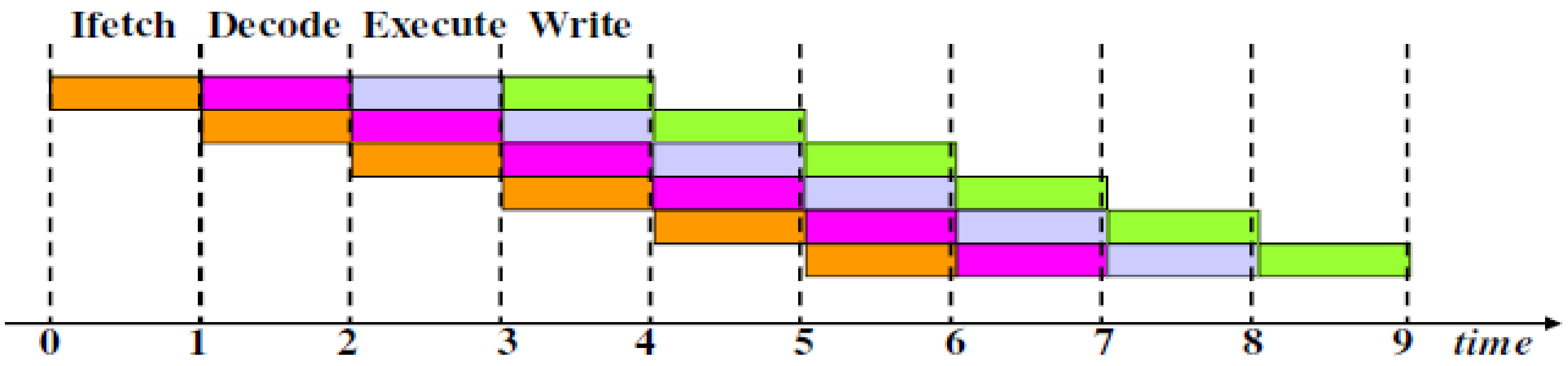


General Superscalar Organization

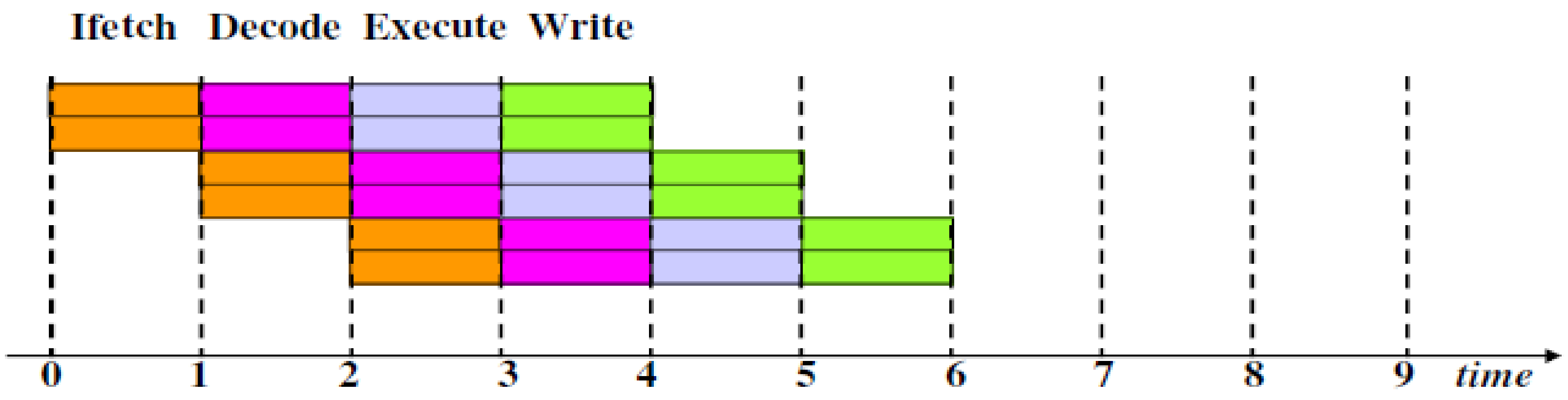
Supports the parallel execution of two integer operations, two floating point operations and one memory operation.

Superscalar Execution

Conventional Pipeline Time diagram



Superscalar Time diagram



Degree = number of pipelines (e.g., degree 2 superscalar pipeline=> two pipelines)

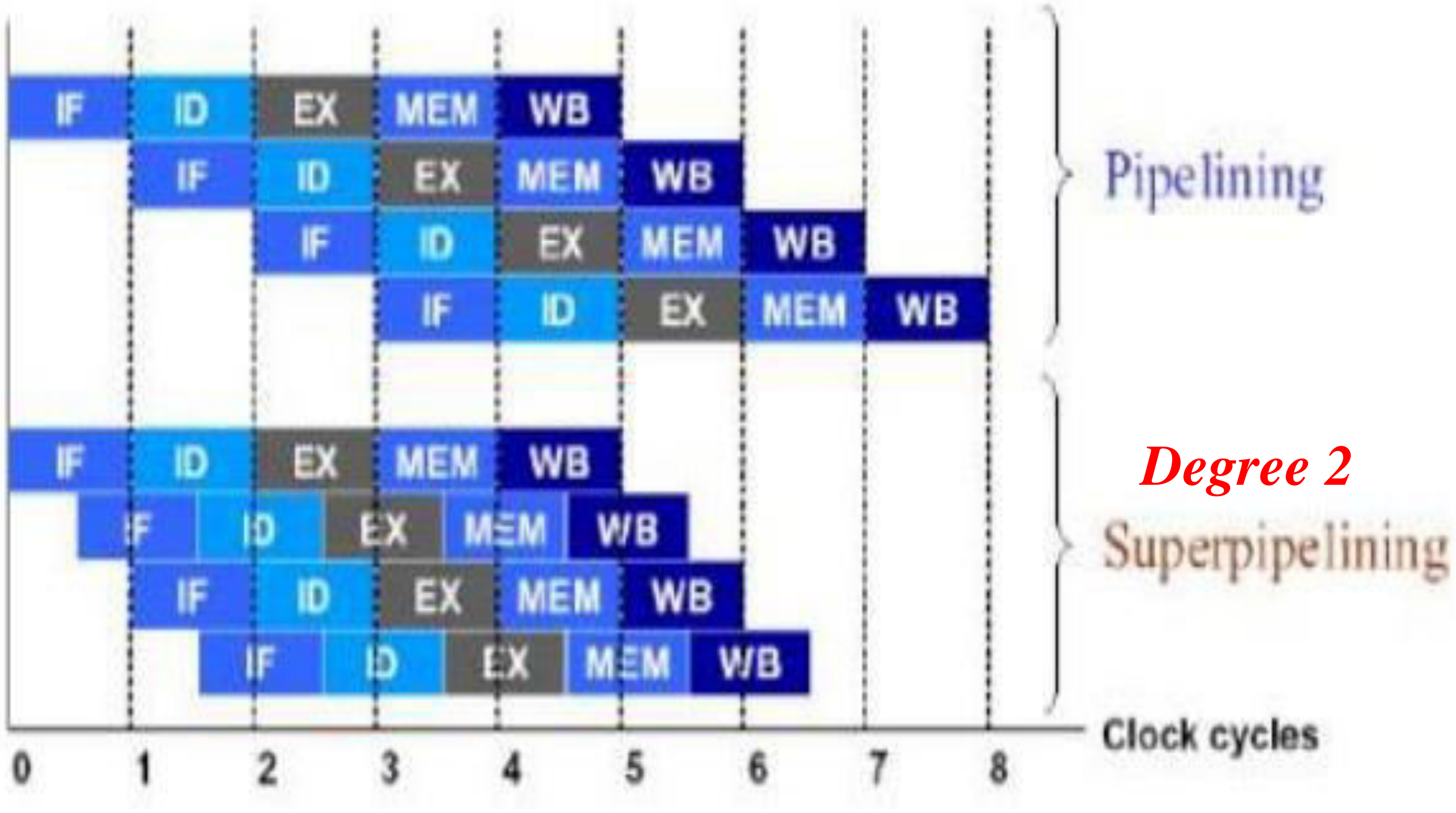
Limitations of superscalar

- The superscalar approach depends on the ability to execute multiple instructions in parallel. The term Instruction-level parallelism refers to the degree to which the instructions of a program can be executed in parallel. A combination of compiler-based optimization and hardware techniques can be used to maximize instruction-level parallelism.

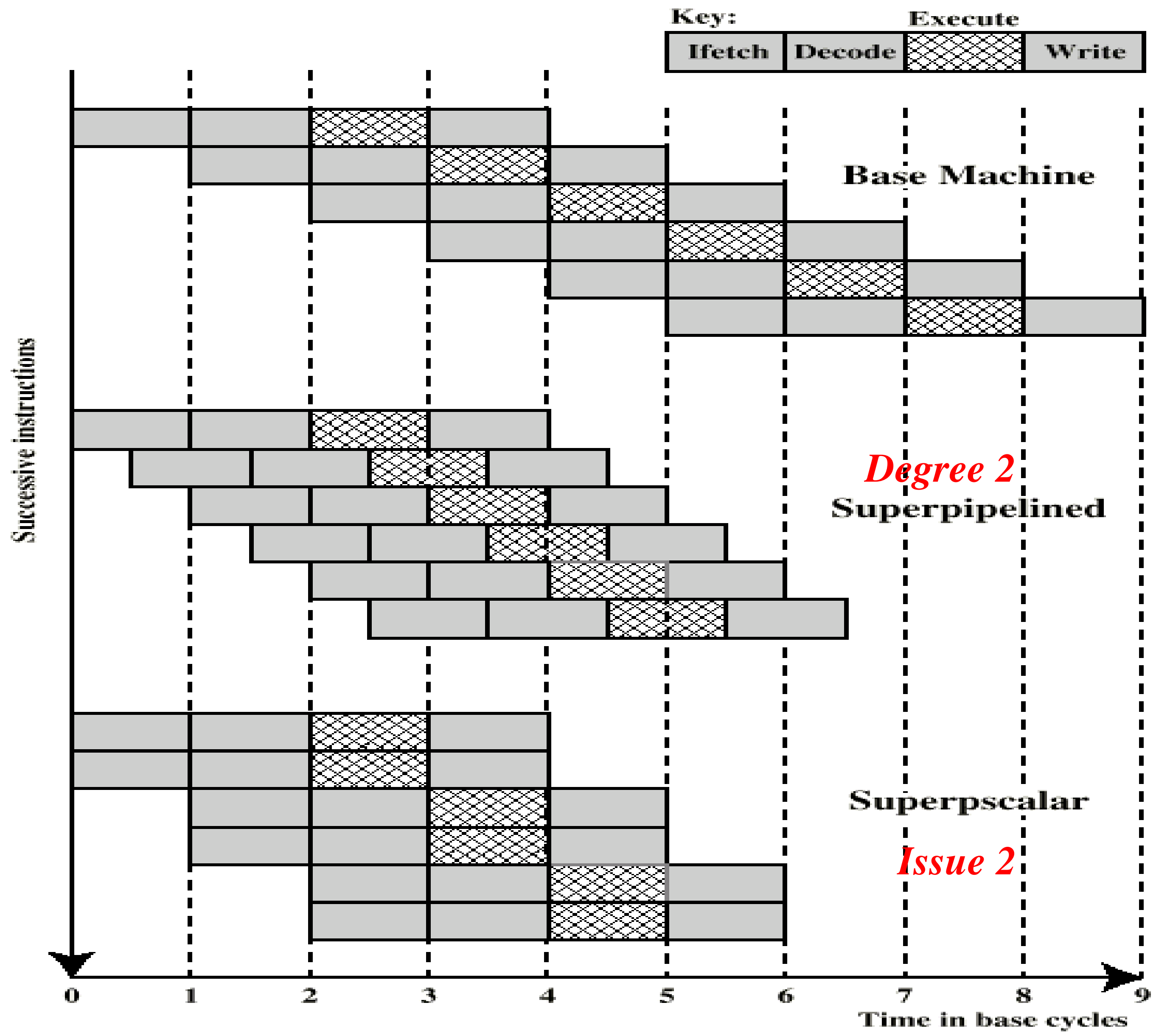
Superpipelined Processor

- 1st invented in 1988.
- Result from the observation that a large number of pipeline operations do not require a full clock cycle to complete.
- Many pipeline stages need less than half a clock cycle.
- Super-pipelining is the breaking of stages of a given pipeline into smaller stages (thus making the pipeline deeper) in an attempt to shorten the clock period and thus enhancing the instruction throughput by keeping more and more instructions in flight at a time.
- Super-pipeline system is capable of performing two pipeline stages per clock cycle.

Superpipelined Processor



Superscalar v Superpipeline



Super pipeline benefit & drawback

- **Benefits:** The major benefit of super-pipelining is the increase in the number of instructions which can be in the pipeline at one time and hence the level of parallelism.

- **Drawbacks:** The larger number of instructions "in flight" (i.e., in some part of the pipeline) at any time, increases the potential for data dependencies to introduce stalls.

Superscalar Performance

Let ,

No. of independent instructions through the pipeline = N

No. of stages in a pipeline = K

Therefore,

Time required by the scalar base m/c = K + N -1

The ideal time required by an *m-issue* superscalar m/c = K + $\dfrac{N-m}{m}$

Where,

K = Time to execute 1st m instruction through the m pipelines simultaneously

$\dfrac{N-m}{m}$ = Time to execute remaining (N – m) instructions, m per cycle, through m pipelines.

$$\textit{Speedup} = \frac{m(N+K-1)}{N+m(K-1)}$$

As N→ ∞ , speedup → m

Superpipeline Performance

In a *superpipelined processor of degree n*, the pipeline cycle time is *1/n* of the base cycle. As a comparison, while a fixed point add takes *1 cycle in the base scalar processor*, the same operation takes *n short cycles in a superpipelined processor* implemented with the same technology.

Therefore,

The min. time required to execute N instructions for a superpipelined m/c of degree n with K stages in the pipeline is

$$K + \frac{1}{n}(N-1)$$

Speedup $= \dfrac{n(K + N - 1)}{nK + N - 1}$ *As* $N \to \infty$ *, speedup* $\to n$

Superscalar-Superpipelined Processor

- Combination of both approaches.

  e.g. Digital Equipments Alpha Processor & MIPS R4000 Processor

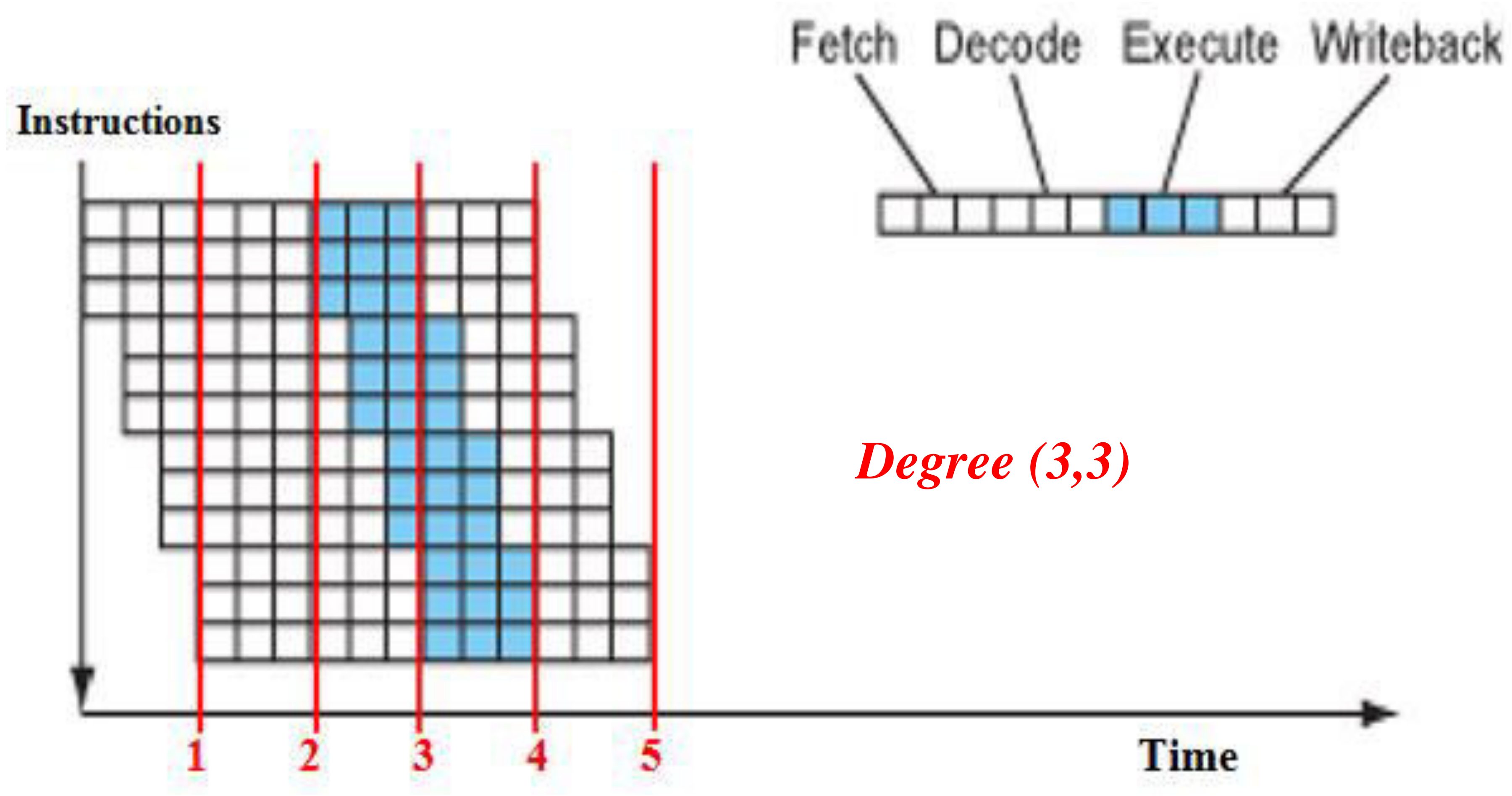
- The m/c executes *m instructions every cycle with a pipeline cycle 1/n of the base cycle*. Thus the *degree of (m,n)*.

- Simple operation *Latency is n* pipeline cycles.

- The level of parallelism to fully utilize this m/c is *mn instructions*.

******Self Review: DEC Alpha Processor & MIPS R4000 From Kai Hwang***

Superscalar-Superpipelined Processor



Degree (3,3)

Superscalar-Superpipeline Performance

Min. time required to execute N independent instructions n on a superscalar-superpipelined m/c of *degtree (m,n)* is –

$$K + \frac{(N-m)}{mn}$$

Therefore,

$$\text{Speedup} = \frac{mn(K+N-1)}{mnK+N-m}$$

As $N \to \infty$, speedup $\to mn$

Hurdles in Superscalar

- Limited by
  - True data dependency
  - Procedural dependency
  - Resource conflicts
  - Output dependency
  - Antidependency

Data Dependency

1. **True Data Dependency**
   or *Read-after-Write* (RAW)

```
. . .
ADD     EAX, ECX
MOV     EBX, EAX
. . .
```

2. **Output Dependency**
   or *Write-after-Write* (WAW)

```
. . .
ADD     EAX, ECX
MOV     EAX, EBX
. . .
```

3. **Antidependency**
   or *Write-after-Read* (WAR)

```
. . .
ADD     EBX, EAX
MOV     EAX, ECX
. . .
```

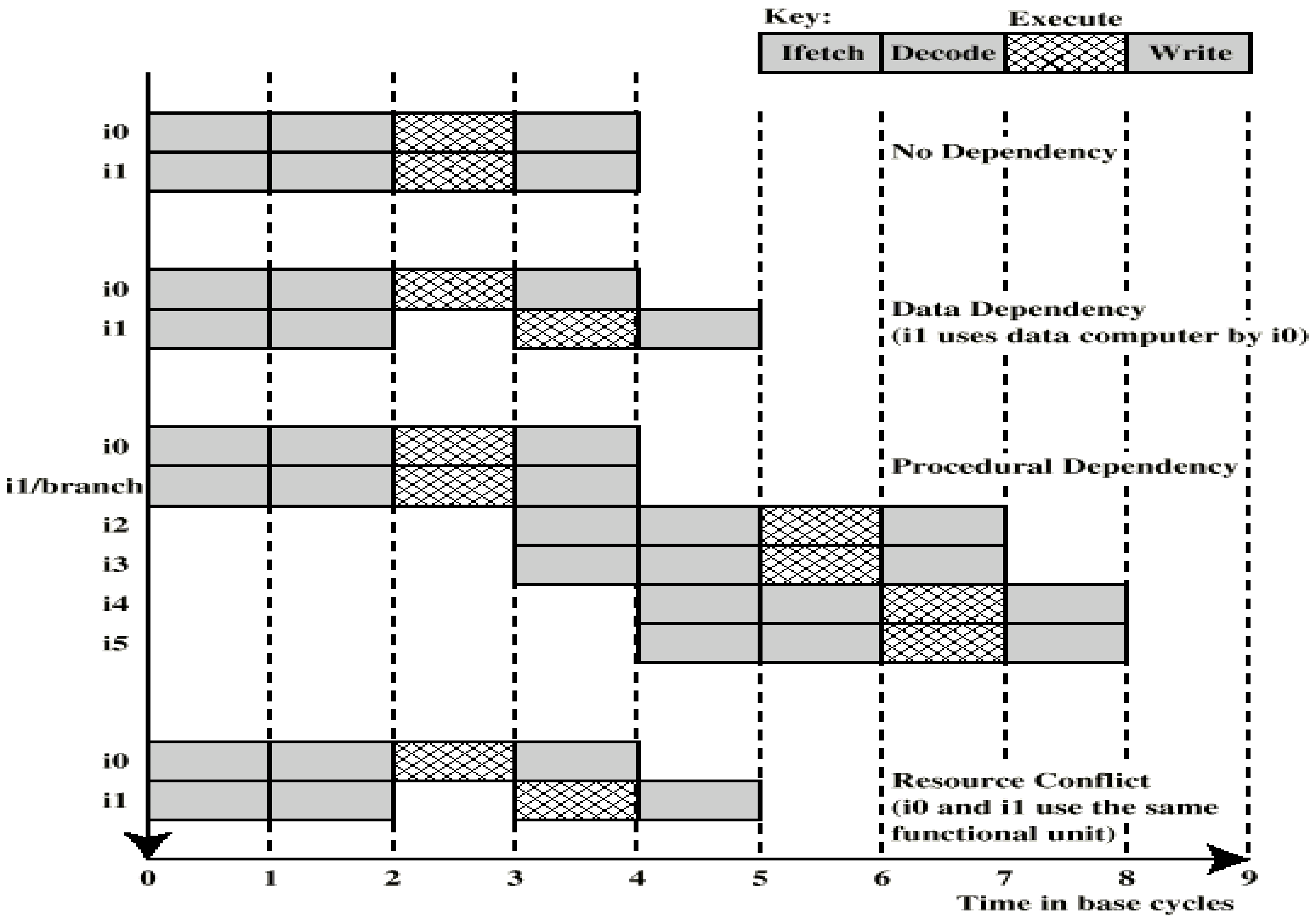
Procedural Dependency

- Can not execute instructions after a branch in parallel with instructions before a branch
- Also, if instruction length is not fixed, instructions have to be decoded to find out how many fetches are needed
- This prevents simultaneous fetches

Resource Conflict

- Two or more instructions requiring access to the same resource at the same time
  - e.g. two arithmetic instructions
- Can duplicate resources
  - e.g. have two arithmetic units

Effect of Dependencies



Design Issues

- A typical superscalar will have
  - multiple instruction pipelines
  - an instruction cache that can provide multiple instructions per fetch
  - multiple buses among the functional units

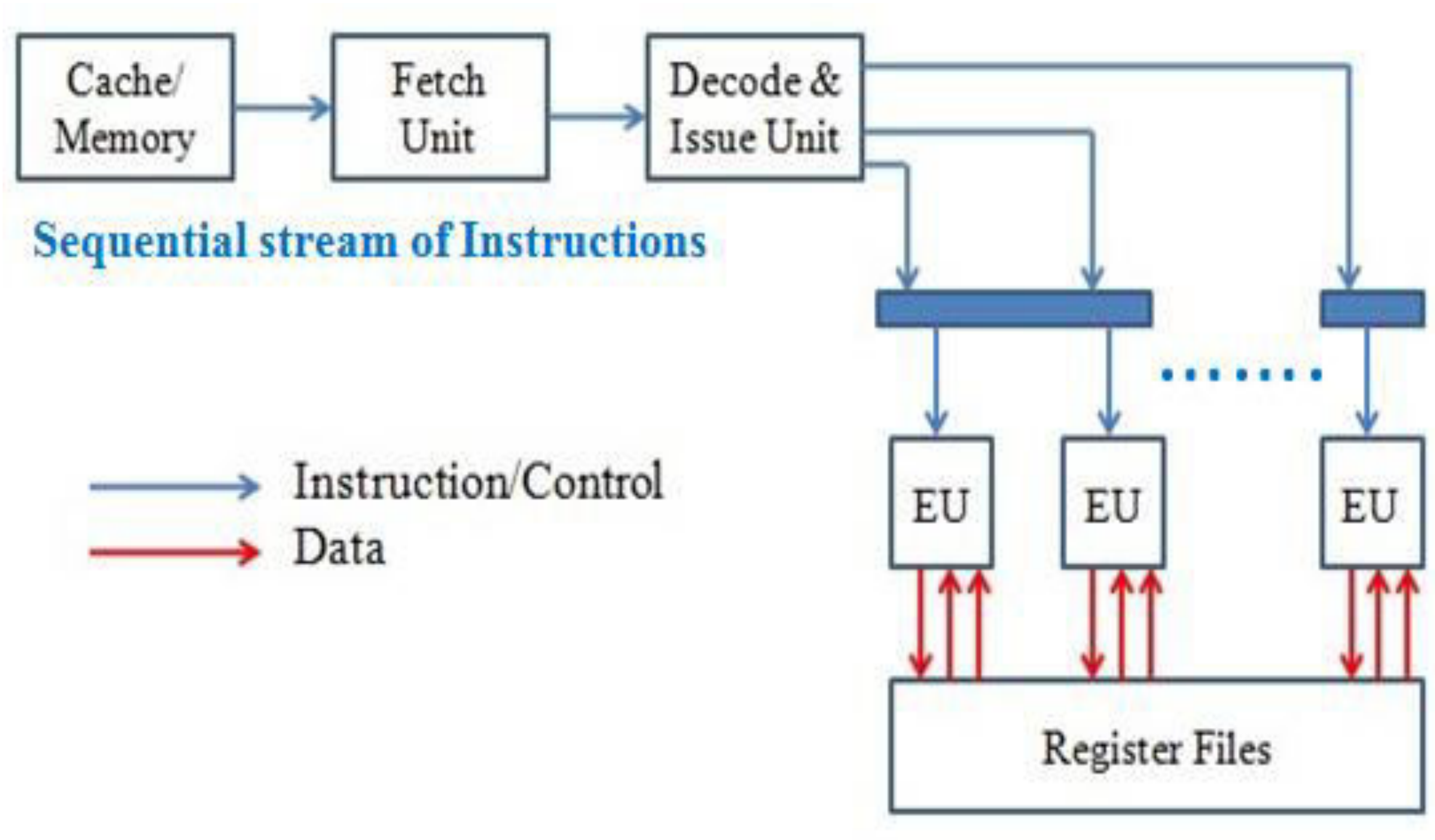
- Instruction level parallelism
  - Instructions in a sequence are independent
  - Execution can be overlapped
  - Governed by data and procedural dependency

- Machine Parallelism
  - Ability to take advantage of instruction level parallelism
  - Governed by number of parallel pipelines & by ability to find independent instructions

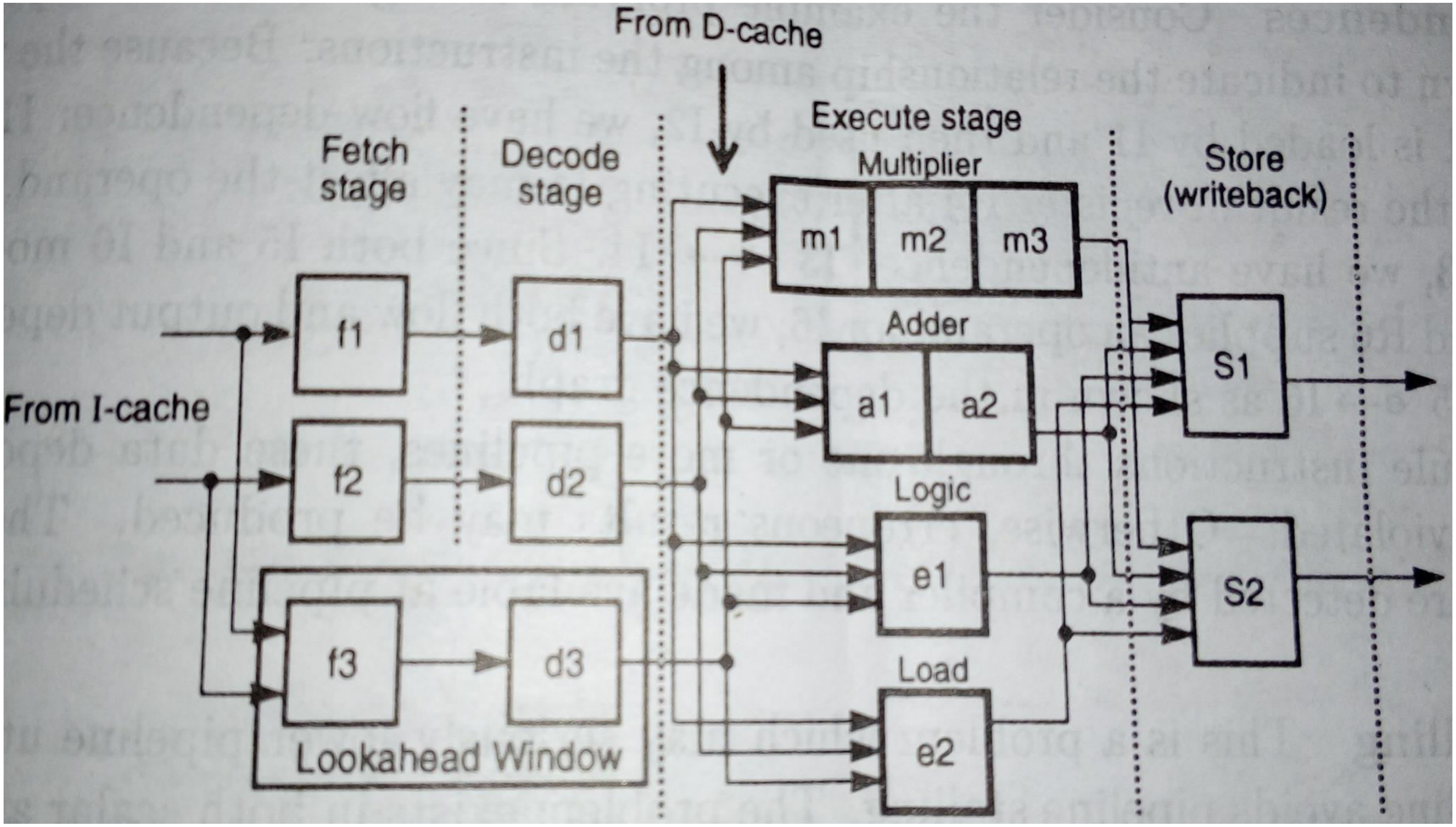
Superscalar Hardware Support

- Facilities to simultaneously fetch multiple instructions
- Logic to determine true dependencies involving register values and Mechanisms to communicate these values
- Mechanisms to initiate multiple instructions in parallel
- Resources for parallel execution of multiple instructions
- Mechanisms for committing process state in correct order

Superscalar Architecture (1)



Superscalar Architecture (2)



A dual-pipeline superscalar processor with 4 functional units in the execution stage & a lookahead window producing out-of-order issues

Instruction Issue Policy

- **Instruction fetch:** order in which instructions are fetched
- **Instruction execution:** order in which instructions are delivered to a functional unit to execute the operation
- **Instruction commit:** order in which instruction results are stored in registers and memory

- **Three categories of issue policies -**

  - In-order issue with in-order completion

  - In-order issue with out-of-order completion

  - Out-of-order issue with out-of-order completion

In-Order Issue In-Order Completion

- Issue instructions in the order they occur and write results in same order
- May fetch more than1 instructions
- Not very efficient – Instructions may stall if:
  - "Partnered" instruction requires more time
  - "Partnered" instruction requires same resource
- Instructions must stall if necessary due to resource conflicts, procedural or any data dependencies.
- Forced order of output
- Easy to implement but rarely used in scalar processors due to some unnecessary delays to maintain instruction order.
- Still used in multiprocessor environment.

In-Order Issue In-Order Completion

Time →

	1	2	3	4	5	6	7	8	9
Pipe 1 , I1	f1	d1	e2	s1					
Pipe 2 , I2	f2	d2		a1	a2	s2			
Pipe 1 , I3		f1	d1		a1	a2	s1		
Pipe 2 , I4		f2	d2	m1	m2	m3	s2		
Pipe 1 , I5			f1	d1	e1			s1	
Pipe 2 , I6			f2	d2		m1	m2	m3	s2

In-Order Issue with In-Order completion in 9 cycles

I1:	Ld	R1, A
I2:	Add	R2, R1
I3:	Add	R3, R4
I4:	Mul	R4, R5
I5:	Comp	R6
I6:	Mul	R6, R7

In-Order Issue Out-of-Order Completion

- Independent instructions are allowed to complete out-of-order.
- Improved pipeline utilization rate.
- Output dependency
  - R3:= R3 + R5; (I1)
  - R4:= R3 + 1; (I2)
  - R3:= R5 + 1; (I3)
  - I2 depends on result of I1 - data dependency
  - If I3 completes before I1, the result from I1 will be wrong - output (read-write) dependency
- Output dependency makes instruction issue logic more complex.
- Forced order of input.
- Found in both scalar & superscalar processors.

In-Order Issue Out-of-Order Completion

Time →

	1	2	3	4	5	6	7	8	9
Pipe 1 , I1	f1	d1	e2	s1					
Pipe 2 , I2	f2	d2		a1	a2	s2			
Pipe 1 , I3		f1	d1		a1	a2	s1		
Pipe 2 , I4		f2	d2	m1	m2	m3	s2		
Pipe 1 , I5			f1	d1	e1	s1			
Pipe 2 , I6			f2	d2		m1	m2	m3	s2

	4	5	6	7	8	9
Pipe 1	I1		I5	I3		
Pipe 2			I2	I4		I6

Completion order

I1:	Ld	R1, A
I2:	Add	R2, R1
I3:	Add	R3, R4
I4:	Mul	R4, R5
I5:	Comp	R6
I6:	Mul	R6, R7

Out-of-Order Issue Out-of-Order Completion

- Use of lookahead window (sometimes called instruction window or buffer) with its own fetch & decode logic.
- Can continue to fetch and decode until this buffer is full
- When a functional unit becomes available, an instruction is assigned to that pipe to be executed provided:
  - it needs that particular functional unit
  - no conflicts or dependencies are currently blocking its execution
- Since instructions have been decoded, processor can lookahead in hopes of identifying independent instructions (reordering).
- Provides optimized pipeline execution.
- Extra cost for lookahead window .

Out-of-Order Issue Out-of-Order Completion

Time →

	1	2	3	4	5	6	7
Pipe 1 , I3	f1	d1	a1	a2	s1		
Pipe 2 , I4	f2	d2	m1	m2	m3	s2	
Lookahead, I5	f3	d3	e1	s1	(Pipe 1)		
Pipe 1 , I6		f1	d1	m1	m2	m3	s2
Pipe 2 , I1		f2	d2	e2	s2		
Pipe 2 , I2			f1	d1	a1	a2	s1

	1	2	3
Pipe 1	I3	I6	
Pipe 2	I4	I1	I2
Lookahead	I5		

Issue Order

	4	5	6	7
Pipe 1	I5	I3		I6
Pipe 2		I1	I4	I2

Completion Order

I1:	Ld	R1, A
I2:	Add	R2, R1
I3:	Add	R3, R4
I4:	Mul	R4, R5
I5:	Comp	R6
I6:	Mul	R6, R7

Exercise

- Consider the following code segment

1. R3=R0*R1
2. R4=R0+R2
3. R5=R0/R1
4. R6=R1+R4
5. R7=R1*R2
6. R1=R0-R2
7. R3=R3*R1
8. R1=R4+R4

Perform: 1. In-order issue with in-order completion

2. In-order issue with out-of-order completion

3. Out-of-order issue with out-of-order completion

Antidependency

- Allowing for rearranged entrance to execution unit may result in Antidependency.
- Called Anitdependency because it is the exact opposite of data dependency
- Data dependency: I2 depends on data from I1.
- Antidependency: I1 depends on data that could be destroyed by I2.
- **Example:**

```
I1:    R3 = R3 + R5
I2:    R4 = R3 + 1
I3:    R3 = R5 + 1
I4:    R7 = R3 + R4
```

- I3 can not complete before I2 starts as I2 needs a value in R3 and I3 changes R3.

Register Renaming

- Output and antidependencies occur because register contents may not reflect the correct ordering from the program
- May result in a pipeline stall
- Registers allocated dynamically
  - i.e. registers are not specifically named
- These problems are storage conflicts – multiple instructions competing for use of same register.
- **Solution:** duplicate resources
- Assigning a value to a register dynamically creates new register
- Subsequent reads to that register must go through renaming process

Register Renaming example

I1:	**R3b = R3a + R5a**
I2:	**R4b = R3b + 1**
I3:	**R3c = R5a + 1**
I4:	**R7b = R3c + R4b**

- Without subscript refers to logical register in instruction
- With subscript is hardware register allocated

Exercise

I1:	**R7 = R3 + R4**
I2:	**R3 = R7**
I3:	**R7 = R7 + 1**
I4:	**R4 = R5**
I5:	**R3 = R7 + R3**
I6:	**R5 = R4 + R3**

Solve the antidependency using Register Renaming.

References

1. PPT on Pipelining from CS303 (3rd Semester)
2. Advanced Computer Architecture – Kai Hwang
3. Advanced Computer Architectures – Dezso Sima, Peter Karsuk
4. Computer Organization – Carl Hamacher
5. Computer Architecture & Organization – John P. Hayes
6. Computer System Architecture – M. Morris Mano
7. Computer Organization & Architecture – T. K. Ghosh
8. Computer Organization & Architecture – Xpress Learning

Thank You