UNIT-2

- Pipelining Hazards
-Pipelining Performance Metrics
-Introduction to RISC and CISC
-Instruction Level Parallelism (ILP)
- Superscalar

Pipelining Hazards

- Pipelining is the process of collecting instructions from a processor using a pipeline.
- It allows you to save and execute instructions through a systematic process. It is also called pipeline processing.
- Whenever a pipeline must stop because of some reason it is called pipeline hazards.
- There are mainly three types of hazards:

1. Structural hazards: They arise from resource conflicts when the hardware cannot support all possible combinations of instructions in simultaneous overlapped execution

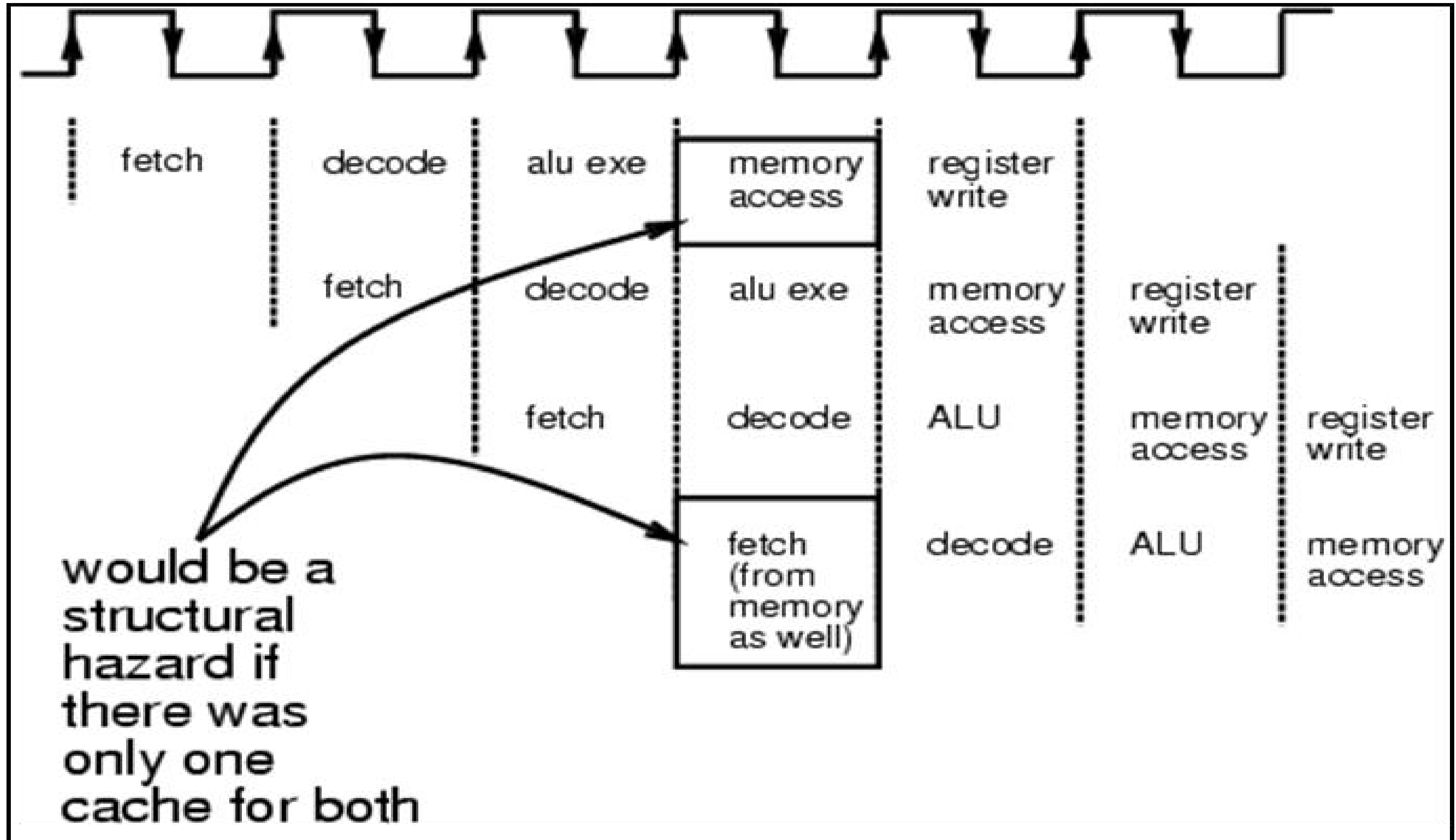
2. Data hazards: They arise when an instruction depends on the result of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline

3. Control hazards: They arise from the pipelining of branches and other instructions that change the PC.

1. Structural hazards

- Structural Hazards occurs when two instructions request the same hardware resource at the same time, causing the pipeline to halt for one of the instructions.
- The most common scenario is when two instructions access memory at the same time. While other instructions are being fetched, one instruction may need to access memory as part of the Execute or Write back phase.
- If both the instructions and the data are stored in the same memory, this is the case.
- Both instructions cannot run at the same time, so one must wait until the other completes the memory access portion. As a result, significant hardware resources are required to avoid structural dangers in general.

If there were no distinct data and instruction caches, a structural hazard would arise from memory access for instruction fetch and memory access for data:

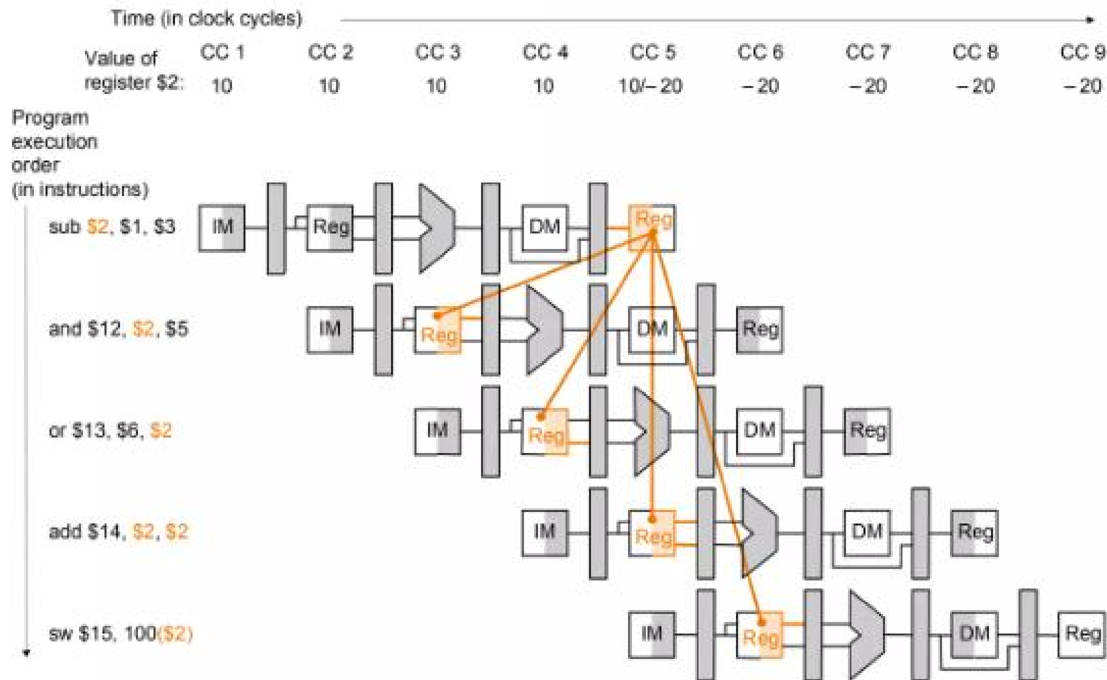


- In the above scenario, in cycle 4, instructions I1 and I4 are trying to access same resource (Memory) which introduces a resource conflict.
- To avoid this problem, we have to keep the instruction on wait until the required resource (memory in our case) becomes available. This wait will introduce stalls in the pipeline.
- STALLS : A stall is commonly called pipeline bubble, since it floats through the pipeline taking space but carry no useful work
- Another solution using the harvard architecture.

2.Data hazards

- When an instruction is dependent on the outcome of a preceding instruction, and the previous instruction's result has not yet been computed, data hazards occur.
- whenever the same storage is used by two separate instructions. The location must appear to be executed in the correct sequence.

- **Data hazards occur when data is used before it is ready**



The use of the result of the SUB instruction in the next three instructions causes a data hazard, since the register $2 is not written until after those instructions read it.

- There are three types of Data Hazards:

1) **Read after Write (RAW):** True dependency or Flow dependency are other terms for RAW. It happens when a value generated by one instruction is required by another instruction.

For example, These hazards need the use of stalls.

ADD R1, — , — ;

SUB — , R1, — ;

$$IO\{NI\} \cap OO\{1stIN\} \neq \Phi$$

Input operand

Next Instruction

Output operand

1st Instruction

2. **Write after Read (WAR)**: Anti-dependency is often known as WAR. These hazards arise when an instruction's output register is used after it has been read by a previous instruction.

For example,

ADD — , R1, — ;

SUB R1, — , — ;

$$IO\{1stI\} \cap OO \neq \phi$$

Input operand

1st Instruction

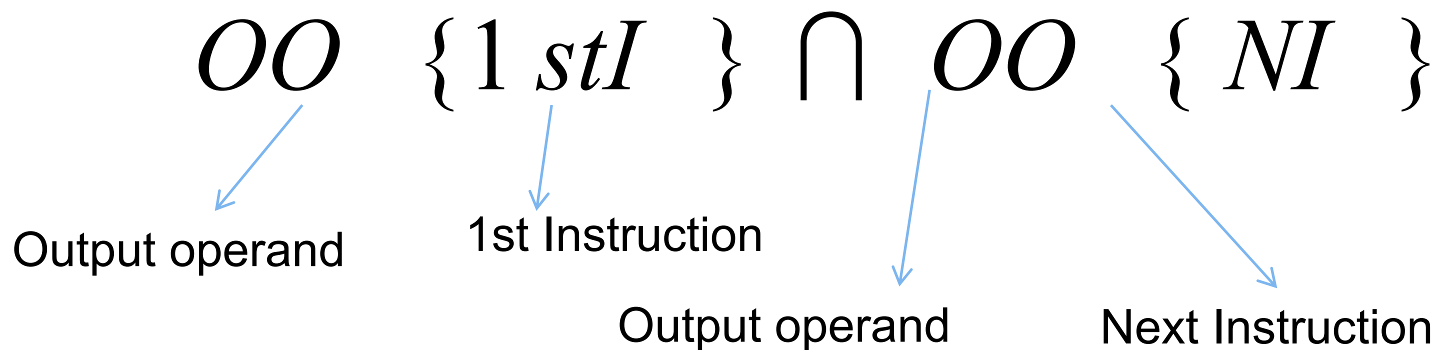
Output operand

3. Write after Write (WAW): Output dependency is sometimes known as WAW. These hazards arise when an instruction's output register is used for writing after it has been written by a previous instruction.

For example,

ADD R1, — , — ;

SUB R1, — , — ;



Solution for Data Hazards Handling:

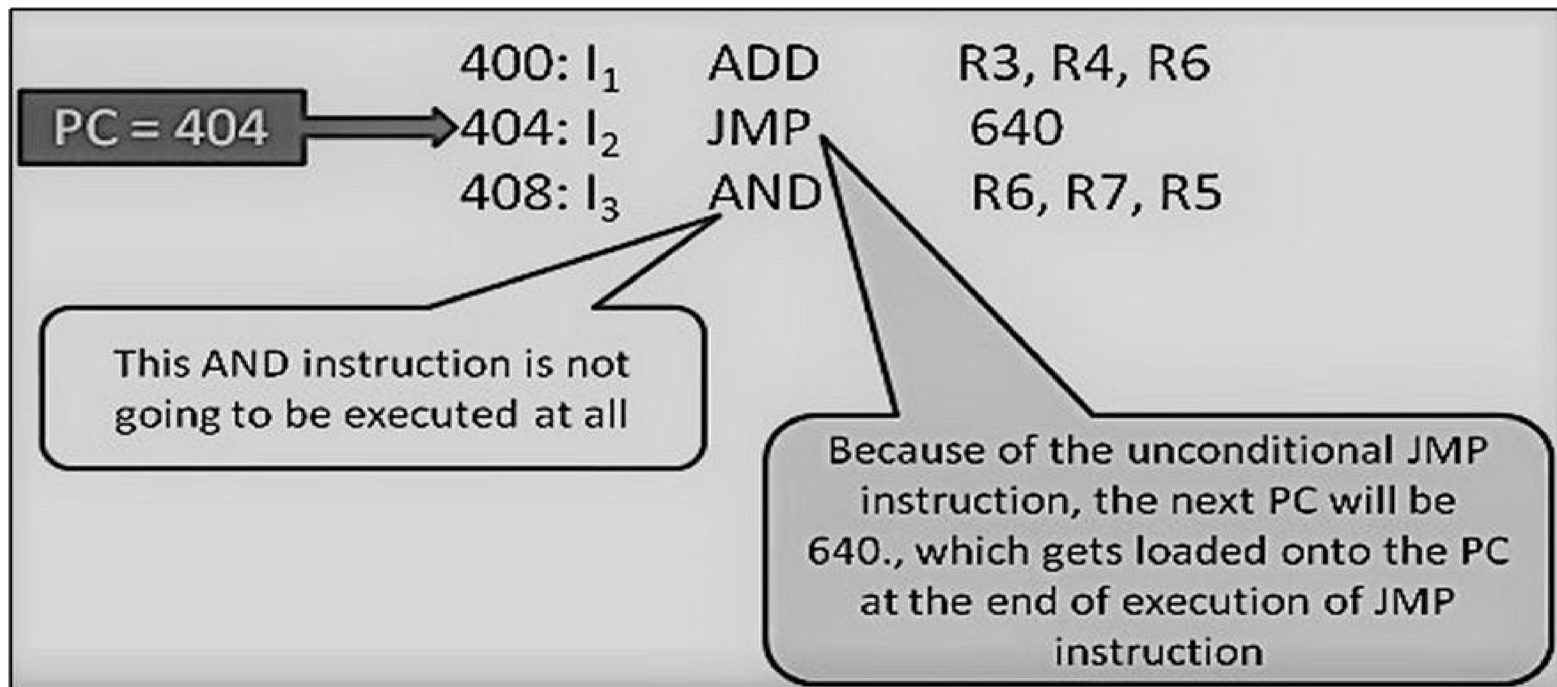
1) Forwarding: It modifies the pipeline by adding special circuitry. This method works because the required values take less time to go via a wire than a pipeline segment does to compute its result.

2) Code Reordering: Reordering code necessitates the use of specialized software. A hardware-dependent compiler is the name for this type of software.

3) Stall Insertion: This method inserts one or more installs (no-op instructions) into the pipeline, delaying execution of the current instruction until the required operand is written to the register file. However, this method reduces pipeline efficiency and throughput.

3. Control hazards:

- Control hazard occurs when the pipeline generates incorrect branch prediction decisions, resulting in instructions entering the pipeline that must be discarded.



Solutions for Control Hazards:

- Stall – stop loading instructions until result is available
- Predict – assume an outcome and continue fetching (undo if prediction is wrong) – lose cycles only on mis-prediction
- Delayed branch – specify in architecture that the instruction immediately following branch is always executed

Pipelining Performance Metrics

1. SpeedUp
2. Throughput
3. Efficiency

This metrics help us to understand how effecient the pipelne in operating compared to a non-pipelined design.

1. SpeedUP

- It gives an idea of “how much faster” the pipelined execution is as compared to non-pipelined execution.
- It is calculated as-

$$\text{Speed Up (S)} = \frac{\text{Non-pipelined execution time}}{\text{Pipelined execution time}}$$

2. Throughput

- Throughput is defined as number of instructions executed per unit time.
- It is calculated as-

$$\text{Throughput} = \frac{\text{Number of instructions executed}}{\text{Total time taken}}$$

3. Efficiency

- The efficiency of pipelined execution is calculated as-

$$\text{Efficiency } (\eta) = \frac{\text{Speed Up}}{\text{Number of stages in Pipelined Architecture}}$$

OR

$$\text{Efficiency } (\eta) = \frac{\text{Number of boxes utilized in phase-time diagram}}{\text{Total number of boxes in phase-time diagram}}$$

Important point to remember:

In pipelined architecture:

1. Calculating Pipelined Execution Time-

- Multiple instructions execute parallelly.
- Number of clock cycles taken by the first instruction = k clock cycles
- After first instruction has completely executed, one instruction comes out per clock cycle.
- So, number of clock cycles taken by each remaining instruction = 1 clock cycle
- Thus,
- Pipelined execution time

= Time taken to execute first instruction + Time taken to execute remaining instructions

= 1 x k clock cycles + (n-1) x 1 clock cycle

= (k + n – 1) clock cycles

2. Calculating Speed Up-

Speed up= Non-pipelined execution time / Pipelined execution time

$= n \times k$ clock cycles / $(k + n – 1)$ clock cycles

$= n \times k / (k + n – 1)$

$= n \times k / n + (k – 1)$

$= k / \{ 1 + (k – 1)/n \}$

- For very large number of instructions, $n \to \infty$. Thus, speed up = k.
- Practically, total number of instructions never tend to infinity.
- Therefore, speed up is always less than number of stages in pipeline.

Problem-01: Consider a pipeline having 4 phases with duration 60, 50, 90 and 80 ns. Given latch delay is 10 ns. Calculate-

1. Pipeline cycle time
2. Non-pipeline execution time
3. Speed up ratio
4. Pipeline time for 1000 tasks
5. Throughput

Solution:

Given-

Four stage pipeline is used

Delay of stages = 60, 50, 90 and 80 ns

Latch delay or delay due to each register = 10 ns

1. **Pipeline Cycle Time-**

= Maximum delay due to any stage + Delay due to its register

= Max { 60, 50, 90, 80 } + 10 ns

= 90 ns + 10 ns = 100 ns

2: **Non-Pipeline Execution Time-**

Non-pipeline execution time for one instruction

= 60 ns + 50 ns + 90 ns + 80 ns = 280 ns

3: **Speed Up Ratio-**

= Non-pipeline execution time / Pipeline execution time

= 280 ns / Cycle time

= 280 ns / 100 ns = 2.8

4. Pipeline time for 1000 tasks

= Time taken for 1st task + Time taken for remaining 999 tasks

= 1 x 4 clock cycles + 999 x 1 clock cycle

= 4 x cycle time + 999 x cycle time

= 4 x 100 ns + 999 x 100 ns = 400 ns + 99900 ns

= 100300 ns

6: Throughput-

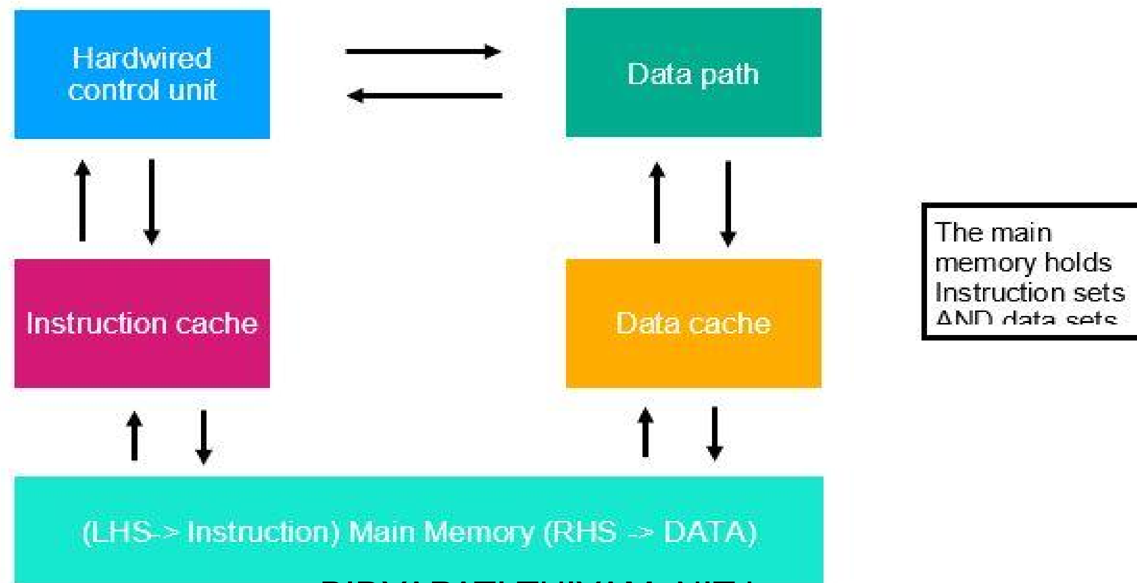
Throughput for pipelined execution

= Number of instructions executed per unit time

= 1000 tasks / 100300 ns

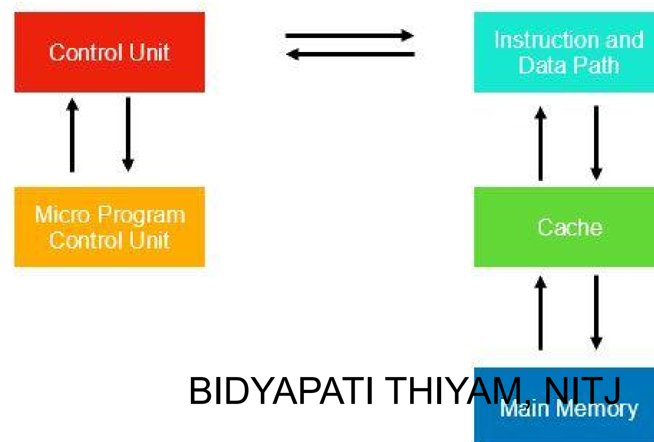
RISC

- Reduced Instruction Set Computer (RISC), is a type of computer architecture that operates on a small, highly optimised set of instructions, instead of a more specialised set of instructions, which can be found in other types of architectures.
- Reduced Instruction Set Computer is a microprocessor that is designed to carry out few instructions at a similar time. Based on small commands, these chips need fewer transistors, which makes the transistors inexpensive to design and produce.



CISC

- The term CISC stands for ‘’Complex Instruction Set Computer’’.
- It is a CPU design plan based on single commands, which are skilled in executing multi-step operations.
- CISC computers have small programs.
- It has a huge number of compound instructions, which take a long time to perform.
- Here, a single set of instructions is protected in several steps; each instruction set has an additional than 300 separate instructions. Maximum instructions are finished in two to ten machine cycles.
- In CISC, instruction pipelining is not easily implemented.



Difference between RISC and CISC processor:

CISC	RISC
A large number of instructions are present in the architecture.	Very fewer instructions are present. The number of instructions are generally less than 100.
Some instructions with long execution times. These include instructions that copy an entire block from one part of memory to another and others that copy multiple registers to and from memory.	No instruction with a long execution time due to very simple instruction set. Some early RISC machines did not even have an integer multiply instruction, requiring compilers to implement multiplication as a sequence of additions.
Variable-length encodings of the instructions. Example: IA32 instruction size can range from 1 to 15 bytes.	Fixed-length encodings of the instructions are used. Example: In IA32, generally all instructions are encoded as 4 bytes.
Multiple formats are supported for specifying operands. A memory operand specifier can have many different combinations of displacement, base and index registers.	Simple addressing formats are supported. Only base and displacement addressing is allowed.
CISC supports array.	RISC does not supports array.
Arithmetic and logical operations can be applied to both memory and register operands.	Arithmetic and logical operations only use register operands. Memory referencing is only allowed by load and store instructions, i.e. reading from memory into a register and writing from a register to memory respectively.
Implementation programs are hidden from machine level programs. The ISA provides a clean abstraction between programs and how they get executed.	Implementation programs exposed to machine level programs. Few RISC machines do not allow specific instruction sequences.
Condition codes are used.	No condition codes are used.
The stack is being used for procedure arguments and return addresses.	Registers are being used for procedure arguments and return addresses. Memory references can be avoided by some procedures.

Parallel Instruction (PI)

- PI are a set of instructions that do not depend on each other to be executed.
- Its a technique that involves executing multipleinstruction simultaneously in a computer program.
- Different Categories of PI:

1. Bit-level parallelism:

- It is based on increasing the processor word size.
- if the word size increases the number of instruction must be reducing.

2. Instruction Level Parallelism (ILP)

- It is a measure of how many operation in a computer program can be performed.
- ILP allows the compiler and the processor to overlap the execution of multiple instruction or even to change the order in which the instruction is executed.

3. Loop level parallelism:

loop will determine to execute simultaneously.

eg.
```
for(i=1;i<=1000;i=i+1)
      x[i]= x[i]+y[i]
```

4. Thread level parallelism:

- Multi-core computers
- Which program can be executed by the processor at the same time.

Instruction Level Parallelism (ILP) Techniques:

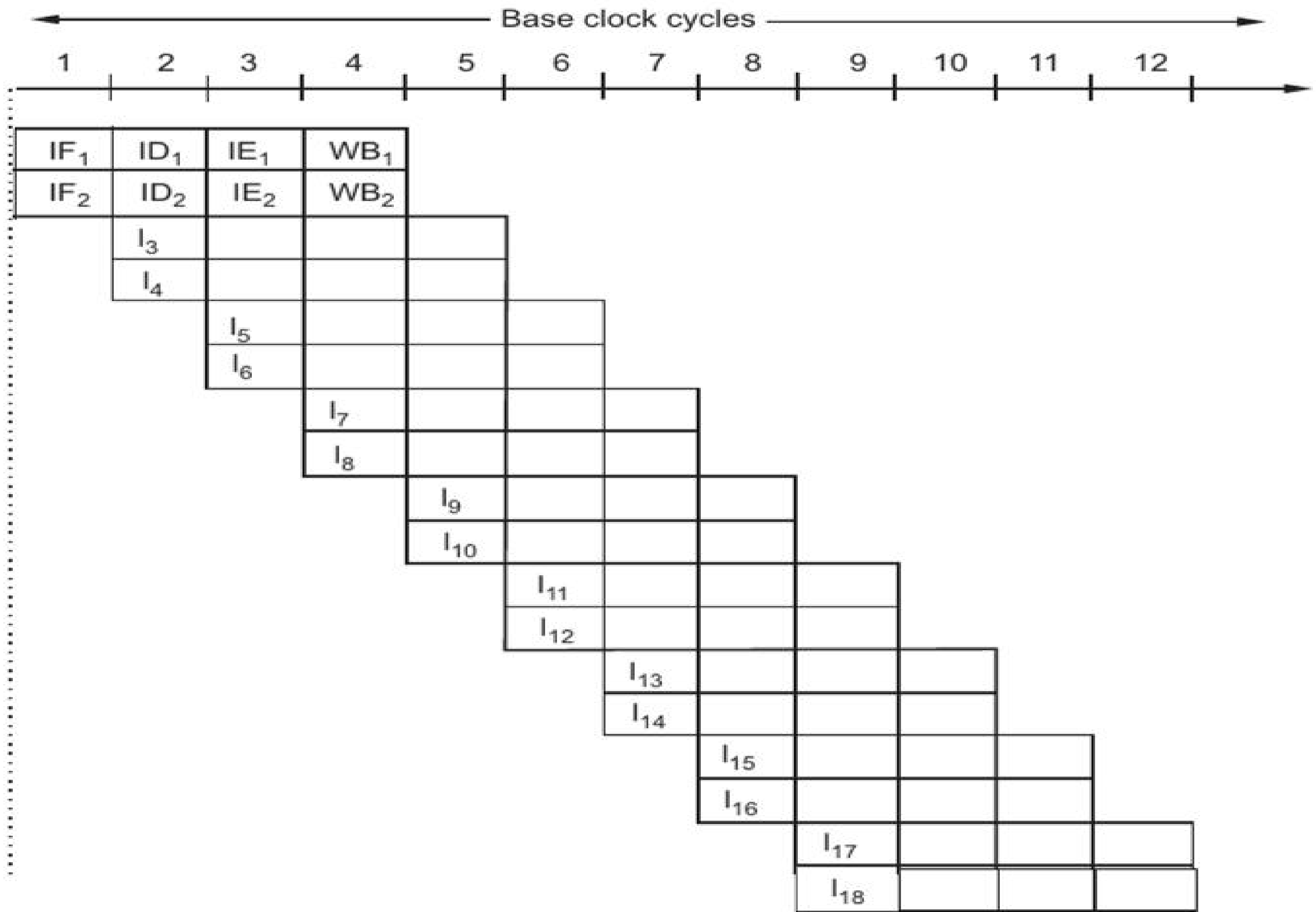
1. Instruction Pipelining
2. Suprescalar ***
3. Out-of-Order execution
4. Register Renaming
5. Speculative execution
6. Branch Prediction

Superscalar

- The term superscalar first appeared sometime in 1987 and simply refers to a machine being designed to increase the performance of the execution of scalar instructions.
- A scalar processor executes single instruction for each clock cycle; a superscalar processor can execute more than one instruction during a clock cycle.
- The design techniques of superscalar normally comprise parallel register renaming, parallel instruction decoding, out-of-order executions & speculative execution.
- These methods are normally used with complementing design methods like pipelining, branch prediction, caching & multi-core within current designs of microprocessors.

- The superscalar approach essentially exploits spatial parallelism, which means the concurrent execution of multiple operations on separate pieces of hardware (i.e. multiple execution units, where each unit again is usually pipelined with a number of different functional stages, giving rise to a scalar-based pipeline).
- A superscalar processor thus contains one or more instruction pipelines, each consisting of a set of functional units such as an integer addition unit, floating-point addition unit, multiplication unit, division unit, and graphic unit in a single CPU.
- These different functional units execute multiple instructions per clock cycle (ILP) while several instructions are simultaneously issued or dispatched to these different functional units (machine parallelism, as there is a number of parallel pipelines)
- Exmple:

A superscalar CPU with two four-stage instruction pipelines ($m = 2$).


Base clock cycles
1
2
3
4
5
6
7
8
9
10
11
12
IF1
ID1
IE1
WB1
IF2
ID2
IE2
WB2
I3
I4
I5
I6
I7
I8
I9
I10
I11
I12
I13
I14
I15
I16
I17
I18