Unit III:

Principles of Pipelining and Vector Processing:

3.1 Basic concepts of Pipelining.

The speed of execution of programs is influenced by many factors. *One way to improve performance is to use faster circuit technology to build the processor and the main memory. Another possibility is to arrange the hardware so that more than one operation can be performed at the same time.* In this way, the number of operations performed per second is increased even though the elapsed time needed to perform any one operation is not changed.

Pipelining is a particularly effective w*ay of organizing concurrent activity in a computer system. It is a process of arrangement of hardware elements of the CPU such that its overall performance is increased. Simultaneous execution of more than one instruction takes place in a pipelined processor*.

Design of a basic pipeline

- *In a pipelined processor, a pipeline has two ends, the input end and the output end. Between these ends, there are multiple stages/segments such that output of one stage is connected to input of next stage and each stage performs a specific operation.*
- *Interface registers are used to hold the intermediate output between two stages. These interface registers are also called latch or buffer.*
- *All the stages in the pipeline along with the interface registers are controlled by a common clock.*

Consider how the idea of pipelining can be used in a computer. *The processor executes a program by fetching and executing instructions, one after the other*. Let F*i* and E*i* refer to the fetch and execute steps for instruction I*i*. Execution of a program consists of a sequence of fetch and execute steps, as shown in Figure 8.1*a*.

Lets consider a computer that has two separate hardware units, one for fetching instructions and another for executing them, as shown in Figure 8.1*b*. The instruction fetched by the fetch unit is deposited in an intermediate storage buffer, B1. This buffer is needed to enable the execution unit to execute the instruction while the fetch unit is fetching the next instruction. The results of execution are deposited in the destination location specified by the instruction.

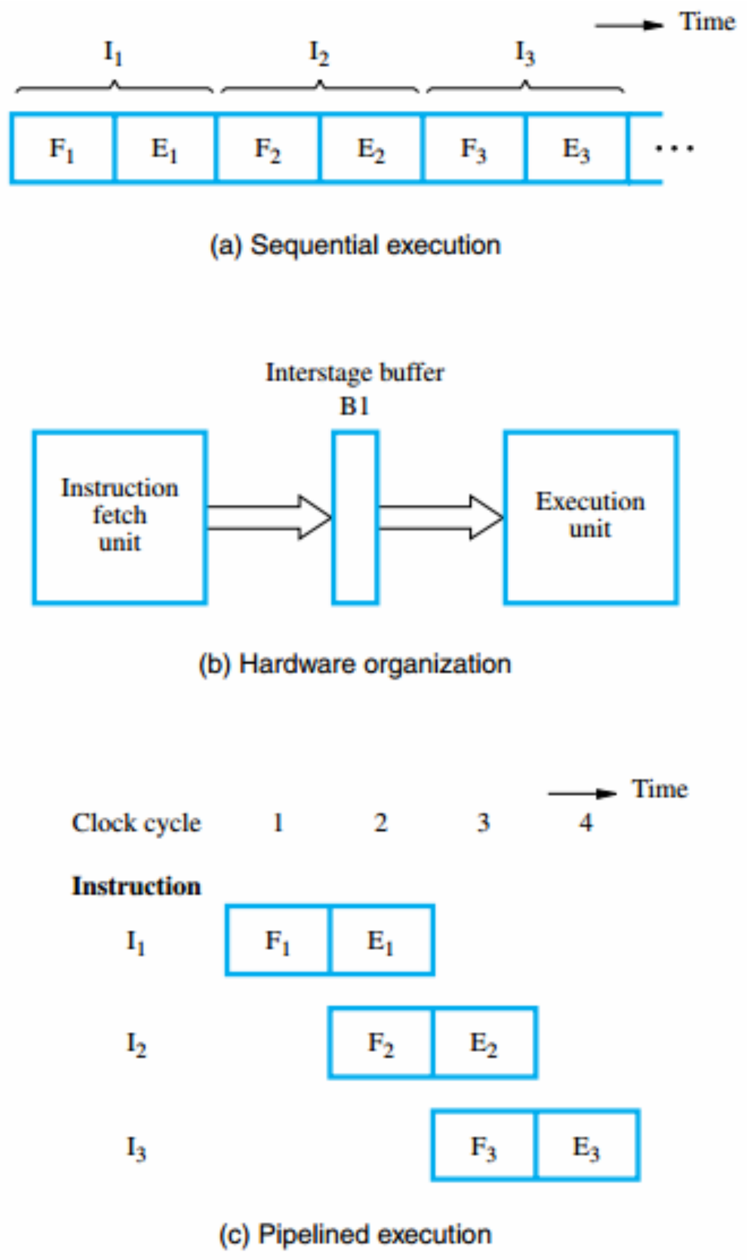


Figure 8.1 Basic idea of instruction pipelining.

The computer is controlled by a clock whose period is such that the fetch and execute steps of any instruction can each be completed in one clock cycle. Operation of the computer proceeds as in Figure 8.1*c*. In the first clock cycle, the fetch unit fetches an instruction I1 (step F1) and stores it in buffer B1 at the end of the clock cycle.

In the second clock cycle, the instruction fetch unit proceeds with the fetch operation for instruction I2 (step F2). Meanwhile, the execution unit performs the operation specified by instruction I1, which is available to it in buffer B1 (step E1). By the end of the second clock cycle, the execution of instruction I1 is completed and instruction I2 is available. Instruction I2 is stored in B1, replacing I1, which is no longer needed. Step E2 is performed by the execution unit during the third clock cycle, while instruction I3 is being fetched by the fetch unit.

In this manner, both the fetch and execute units are kept busy all the time. If the pattern in Figure 8.1*c* can be sustained for a long time, the completion rate of instruction execution will be twice that achievable by the sequential operation depicted in Figure 8.1*a*.

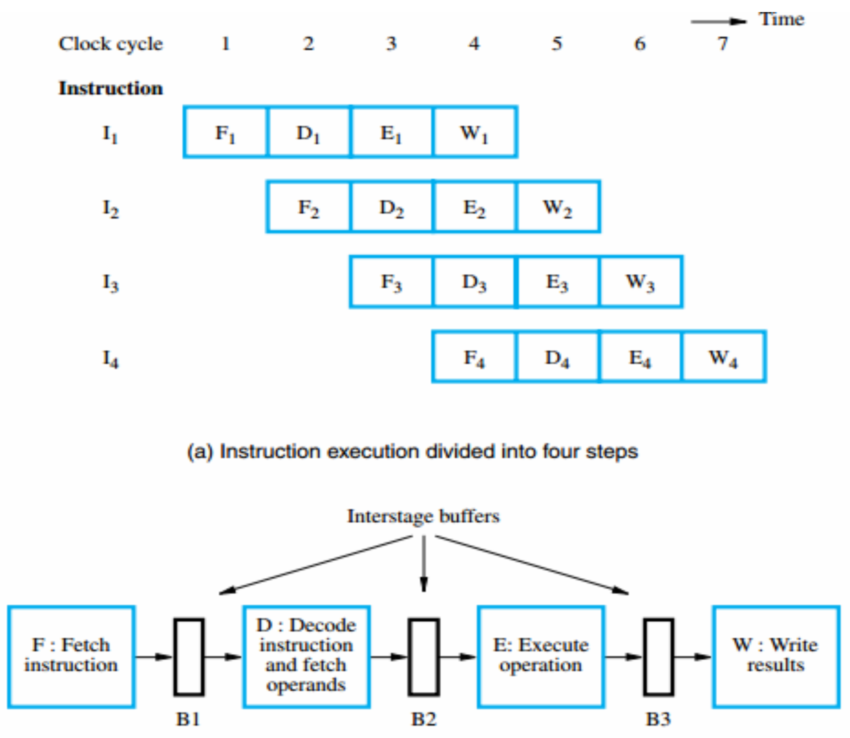
The processing of an instruction need not be divided into only two steps. For example, a pipelined processor may process each instruction in four steps, as follows:
F Fetch: read the instruction from the memory.
D Decode: decode the instruction and fetch the source operand(s).
E Execute: perform the operation specified by the instruction.
W Write: store the result in the destination location.
The sequence of events for this case is shown in Figure 8.2*a. Four instructions are in progress at any given time. This means that four distinct hardware units are needed, as shown in Figure 8.2b.* These units must be capable of performing their tasks simultaneously and without interfering with one another. Information is passed from one unit to the next through a storage buffer. As an instruction progresses through the pipeline, all the information needed by the stages downstream must be passed along.

For example, during clock cycle 4, the information in the buffers is as follows:

• Buffer B1 holds instruction I3, which was fetched in cycle 3 and is being decoded by the instruction-decoding unit.

• Buffer B2 holds both the source operands for instruction I2 and the specification of the operation to be performed. This is the information produced by the decoding hardware in cycle 3. The buffer also holds the information needed for the write step of instruction I2 (step W2). Even though it is not needed by stage E, this information must be passed on to stage W in the following clock cycle to enable that stage to perform the required Write operation.

• Buffer B3 holds the results produced by the execution unit and the destination information for instruction I1.


Time
Clock cycle 1 2 3 4 5 6 7
Instruction
I_1 F_1 D_1 E_1 W_1
I_2 F_2 D_2 E_2 W_2
I_3 F_3 D_3 E_3 W_3
I_4 F_4 D_4 E_4 W_4
Interstage buffers
F : Fetch instruction
B1
D : Decode instruction and fetch operands
B2
E: Execute operation
B3
W : Write results

(a) Instruction execution divided into four steps

(b) Hardware organization

Figure 8.2 A 4-stage pipeline.

Execution in a pipelined processor

Execution sequence of instructions in a pipelined processor can be visualized using a space-time diagram. For example, consider a processor having 4 stages and let there be 2 instructions to be executed. We can visualize the execution sequence through the following space-time diagrams:

Non overlapped execution:

Stage\Cycle	1	2	3	4	5	6	7	8
S1	I_1				I_2			
S2		I_1				I_2		
S3			I_1				I_2	
S4				I_1				I_2

Total time = 8 cycles

Overlapped execution:

Stage\Cycle	1	2	3	4	5
S1	I_1	I_2			
S2		I_1	I_2		
S3			I_1	I_2	
S4				I_1	I_2

Total time = 5 cycles

speedup due to pipelining

Consider a ‘k’ segment pipeline with clock cycle time as ‘Tp’. Let there be ‘n’ tasks to be completed in the pipelined processor. Now, the first instruction is going to take ‘k’ cycles to come out of the pipeline but the other ‘n – 1’ instructions will take only ‘1’ cycle each, i.e, a total of ‘n – 1’ cycles. So, time taken to execute ‘n’ instructions in a pipelined processor:

$ET_{pipeline} = k + n - 1$ cycles

$= (k + n - 1)\ Tp$

In the same case, for a non-pipelined processor, execution time of ‘n’ instructions will be:

$ET_{non\text{-}pipeline} = n\ \ k\ \ Tp$

So, speedup (S) of the pipelined processor over non-pipelined processor, when ‘n’ tasks are executed on the same processor is:

S = Performance of pipelined processor / Performance of Non-pipelined processor

As the performance of a processor is inversely proportional to the execution time, we have,

$S = ET_{non\text{-}pipeline} / ET_{pipeline}$

$\Rightarrow S = [n\ \ k\ \ Tp] / [(k + n - 1)\ \ Tp]$

$S = [n\ \ k] / [k + n - 1]$

When the number of tasks ‘n’ are significantly larger than k, that is, n >> k

$S \approx n\ \ k / n$

$S \approx k$

where ‘k’ are the number of stages in the pipeline.

Throughput = Number of instructions / Total time to complete the instructions

So, Throughput = $n / (k + n - 1)\ \ Tp$

3.2 Pipeline Processor Classification

Pipeline processors can be classified based on several criteria, including the nature of the pipeline, the level of parallelism, and the control mechanisms used. Here are some common classifications:

1. Based on Pipeline Nature:

- Linear Pipeline: Instructions flow sequentially from one stage to the next without any feedback or branching.

- Non-linear Pipeline: Instructions can take multiple paths through the pipeline, including feedback loops and branching.

2. Based on Level of Parallelism:

- Instruction-Level Parallelism (ILP): Multiple instructions are executed simultaneously within the same pipeline stage.

- Data-Level Parallelism (DLP): Multiple data elements are processed simultaneously in parallel pipelines.

- Task-Level Parallelism (TLP): Different tasks or threads are executed in parallel across multiple pipelines.

3. Based on Control Mechanisms:

- Static Pipeline: The order of instruction execution is determined at compile time and does not change during runtime.

- Dynamic Pipeline: The order of instruction execution can be adjusted dynamically at runtime based on dependencies and resource availability.

4. Based on Pipeline Depth:

- Shallow Pipeline: A pipeline with fewer stages, typically used in simpler processors.

- Deep Pipeline: A pipeline with many stages, allowing for higher clock speeds and greater throughput but with increased complexity and potential for pipeline stalls.

5. Based on Instruction Set Architecture (ISA):

- RISC Pipeline: Typically used in Reduced Instruction Set Computing (RISC) architectures, where the pipeline is designed to execute simple instructions quickly.

- CISC Pipeline: Used in Complex Instruction Set Computing (CISC) architectures, where the pipeline is designed to handle more complex instructions.

6. Based on Application Domain:

- General-Purpose Pipeline: Designed for a wide range of applications and instruction sets.

- Specialized Pipeline: Optimized for specific types of computations, such as graphics processing units (GPUs) or digital signal processors (DSPs).

Non-linear Pipeline Processor

A non-linear pipeline processor is a type of pipeline architecture where the stages of the pipeline are not strictly linear. Unlike linear pipelines, where instructions flow sequentially from one stage to the next, non-linear pipelines allow for more complex interactions between stages, including feedback loops, branching, and parallel processing.

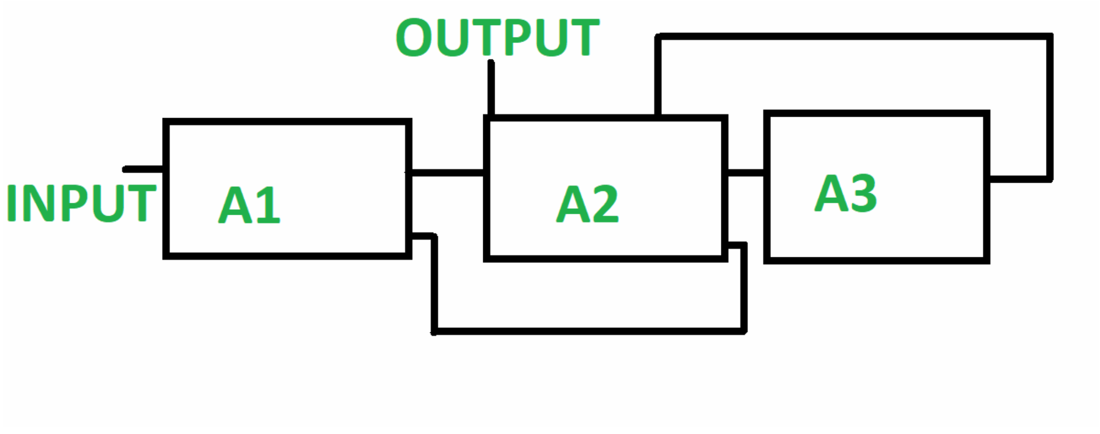


Fig: non linear pipeline

Non-linear pipeline processors are characterized by their ability to handle complex instruction flows that deviate from the strict sequential progression seen in linear pipelines.

One of the key features is the presence of **feedback paths**, which allow data to be sent back to previous stages for further processing, enabling iterative operations. Additionally, non-linear pipelines support **branching**, where instructions can take different paths through the pipeline based on conditions, accommodating complex control flows like loops and conditional statements.

They also leverage **parallel processing**, allowing multiple instructions to be processed simultaneously across different stages, which enhances throughput. Furthermore, non-linear pipelines often employ **dynamic scheduling**, where the order of instruction execution is adjusted in real-time based on dependencies and resource availability, optimizing performance.

These characteristics make non-linear pipelines highly flexible and capable of handling intricate computational tasks, but they also introduce challenges such as increased complexity, higher latency due to feedback and branching, and greater hardware overhead.

Despite these challenges, non-linear pipelines are widely used in high-performance computing, real-time systems, and multimedia processing, where their advanced capabilities are essential.

Following are the differences between linear and non linear parallelism:

S.NO.	**Linear Pipeline**	**Non-Linear Pipeline**
1	In linear pipeline a series of processors are connected together in a serial manner.	In Non-Linear pipeline different pipelines are present at different stages.
2	Linear pipeline is also called as static pipeline as it performs fixed functions.	Non-Linear pipelines is also called as dynamic pipeline as it performs different functions.
3	The output is always produced from the last block.	The output is not necessarily produced from the last block.
4	Linear pipeline has linear connections.	Non-Linear pipeline has feedback and feed-forward connections.
5	It generates a single reservation table.	It can generate more than one reservation table.
6	It allows easy functional partitioning.	Functional partitioning is difficult in non-linear pipeline.

Advantages of non-linear pipelines

- Increased Flexibility: Non-linear pipelines can handle more complex instruction sets and control flows.

- Higher Throughput: Parallel processing and dynamic scheduling can lead to higher overall performance.

- Better Resource Utilization: Resources can be allocated more efficiently based on the current state of the pipeline.

Disadvantages of non-linear pipelines:

- Complexity: Non-linear pipelines are more complex to design and implement compared to linear pipelines.

- Increased Latency: Feedback paths and branching can introduce additional latency.

- Hardware Overhead: Additional hardware is required to manage the non-linear flow of instructions.

Applications:

- High-Performance Computing: Non-linear pipelines are often used in supercomputers and high-performance processors where complex computations are required.

- Real-Time Systems: Systems that require real-time processing and complex control flows can benefit from non-linear pipelines.

- Multimedia Processing: Applications that involve parallel processing of multimedia data, such as video and audio, can leverage non-linear pipelines.

3.4 Instruction pipeline design

DLX is a simple pipeline architecture for CPU. It is mostly used in universities as a model to study pipelining technique.

The architecture of DLX was chosen based on observations about most frequently used primitives in programs.

Every DLX instruction can be implemented in at most five clock cycles. The five clock cycles are

a. Instruction fetch cycle (IF)
b. Instruction decode/register fetch (ID)
c. Execution/Effective address cycle (EX)
d. Memory access/branch completion cycle (MEM)
e. Write-back cycle (WB)

DLX datapath with almost no changes by starting a new instruction on each clock cycle. Each of the clock cycles of the DLX datapath now becomes a pipe stage: a cycle in the pipeline.
While each instruction takes five clock cycles to complete, during each clock cycle the hardware will initiate a new instruction and will execute some part of the five different instructions. The typical way to show what is going on is:

Instr Num	1	2	3	4	5	6	7	8	9
instr i	IF	ID	EX	MEM	WB				
instr i+1		IF	ID	EX	MEM	WB			

instr i+2	IF	ID	EX	MEM	WB		
instr i+3		IF	ID	EX	MEM	WB	
instr i+4			IF	ID	EX	MEM	WB

3.5 Pipeline Hazards

There are mainly three types of dependencies possible in a pipelined processor. These are:

1) Structural Dependency

2) Control Dependency

3) Data Dependency

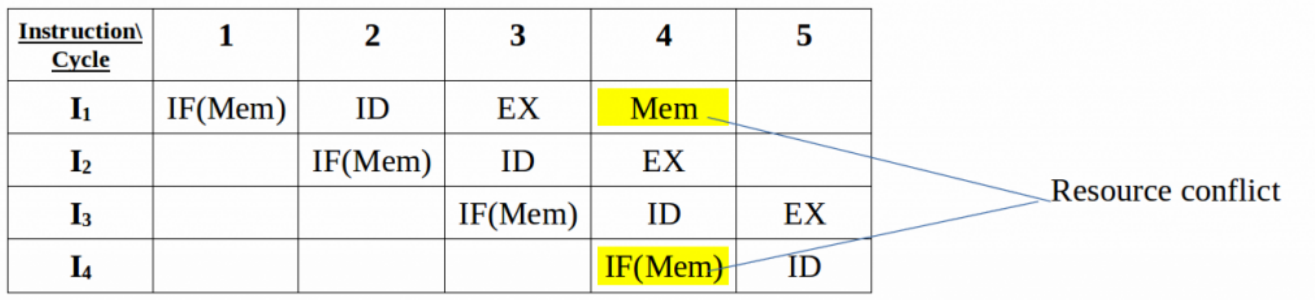
These dependencies may introduce stalls in the pipeline.

Stall: A stall is a cycle in the pipeline without new input.

Structural dependency

This dependency arises due to the resource conflict in the pipeline. A resource conflict is a situation when more than one instruction tries to access the same resource in the same cycle. A resource can be a register, memory, or ALU.

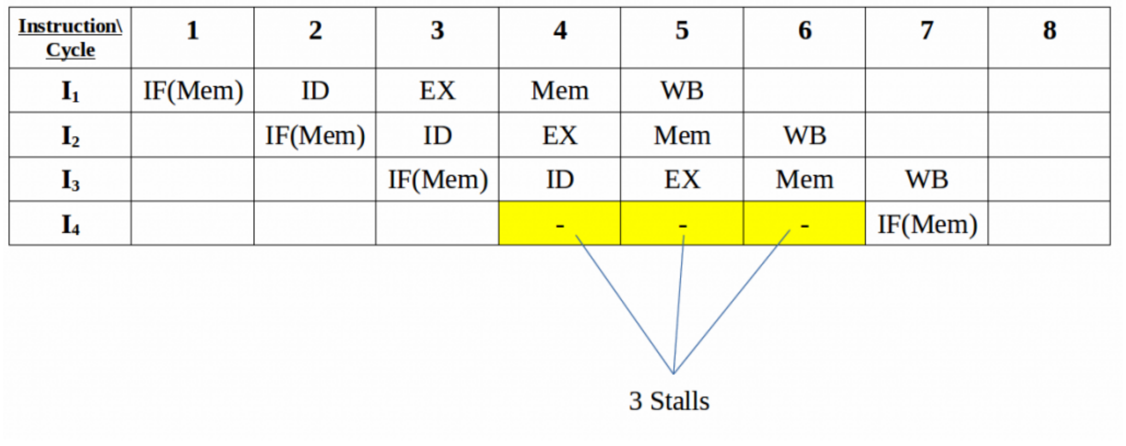
Example:



Instruction\ Cycle	1	2	3	4	5
I_1	IF(Mem)	ID	EX	Mem	
I_2		IF(Mem)	ID	EX	
I_3			IF(Mem)	ID	EX
I_4				IF(Mem)	ID

Resource conflict

In the above scenario, in cycle 4, instructions I_1 and I_4 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 as shown below:



Instruction\ Cycle	1	2	3	4	5	6	7	8
I_1	IF(Mem)	ID	EX	Mem	WB			
I_2		IF(Mem)	ID	EX	Mem	WB		
I_3			IF(Mem)	ID	EX	Mem	WB	
I_4				-	-	-	IF(Mem)	

Solution for structural dependency
To minimize structural dependency stalls in the pipeline, we use a hardware mechanism called Renaming.
Renaming : According to renaming, we divide the memory into two independent modules used to store the instruction and data separately called Code memory(CM) and Data memory(DM) respectively. CM will contain all the instructions and DM will contain all the operands that are required for the instructions.

Data Hazards

Data hazards occur when two instructions in a pipeline refer to the same register and at least one of them writes to the register. Compiler writers use the phrase "data dependences" to cover the same kind of problem, but their terminology refers to what you can see in an instruction stream without considering the pipeline.

Also, execution circuitry is usually broken up into multiple functional units, each performing different types of operations. These functional units can be performing operations in parallel. More complex operations may take several cycles to complete. To illustrate the difficulties that result, consider the following MIPS code snippet.

```
div.d   $f0, $f2, $f4

mul.d   $f6, $f8, $f0

add.d   $f0, $f10, $f12
```

The use of register $f0 can give rise to three different kinds of problems in this code.

- Read after Write (RAW) hazards, also known as true dependences
- Write after Write (WAW) hazards, also known as output dependences

- Write after Read (WAR) hazards, also known as antidependences

The naming of these hazards is based on what is supposed to happen. That is, a WAR hazard occurs when an instruction that writes to a register follows soon after an instruction that reads from the same register. If the write precedes the read then the first instruction (the read) is working with the wrong data value.

Read after Write Hazards (True Dependences)

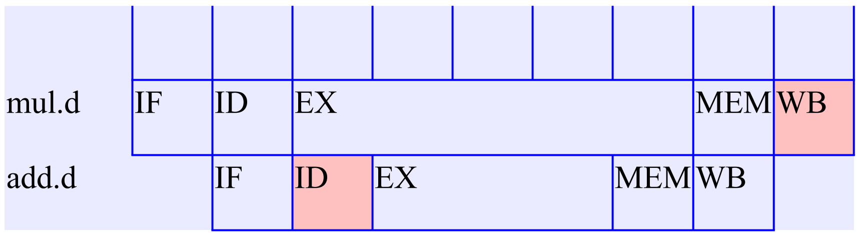
A *true dependence* arises when one instruction computes a value that is used by a later instruction. More precisely, the same operand register is a destination operand in the earlier instruction and a source operand in the later instruction, and there are no instructions between them that write a different value to the register.

If the two instructions can be in the pipeline at the same time and there is a possibility that the value will not be ready when the second instruction reads its source operands the condition is called a *read after write* (RAW) hazard.

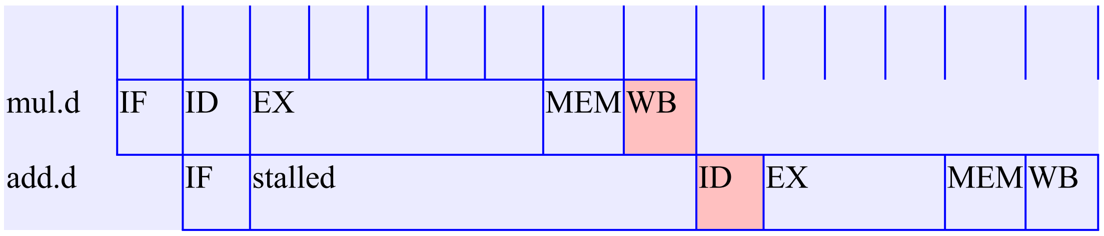
For example, consider the following code.

```
mul.d   $f0, $f2, $f4
add.d   $f6, $f8, $f0
```

Here, the value produced by the mul.d instruction in $f0 is used as a source operand by the add.d instruction. Even if the execution phase of the mul.d instruction takes a single cycle, there is a hazard because the value is produced late but it is needed early as a source operand. Suppose the mul.d instruction takes 5 cycles for its EX stage and the add.d takes 3 cycles. The following chart indicates the timing of the two instructions with the RAW hazard ignored.



In order to handle the RAW hazard correctly, the ID stage of the add.d instruction should be stalled until $f0 has the value written by the mul.d instruction.



Register forwarding is a technique for faster handling of RAW hazards. It involves adding and controling direct data paths from functional unit outputs to functional unit inputs. With register forwarding, the ID stage of the add.d instruction can be started during the last cycle of the EX stage of the mul.d instruction, as shown below.

mul.d	IF	ID	EX					MEM	WB			
add.d		IF	stalled				ID	EX			MEM	WB

Write after Write Hazards (Output Dependences)

An *output dependence* arises when an earlier instruction writes a result to the same place that a later instruction writes to. More precisely, the same operand register is a destination operand in both the earlier instruction and the later instruction, and there are no instructions between them that write a different value to the register.

If the two instructions can be in the pipeline at the same time and there is a possibility that the second instruction will write its result before the first instruction writes its result then the condition is called a *write after write* (WAW) hazard.

For example, consider the following code.

```
mul.d   $f0, $f2, $f4
add.d   $f0, $f10, $f12
```

Here, the mul.d instruction and the add.d instructions will both try to write to $f0. Suppose the mul.d instruction takes 5 cycles for its EX stage and the add.d takes 3 cycles. The following chart indicates the timing of the two instructions.

mul.d	IF	ID	EX					MEM	WB
add.d		IF	ID	EX			MEM	WB	

This will result in later instructions seeing the wrong value in $f0 - the result from the mul.d instruction rather than the result of the add.d instruction. To remedy this problem, the WB stage of the add.d instruction should be stalled until after the WB stage of the mul.d instruction, as shown below.

mul.d	IF	ID	EX					MEM	WB	
add.d		IF	ID	EX			MEM	stalled		WB

Write after Read Hazards (Antidependences)

An *antidependence* arises when an earlier instruction reads a value from the same place that will be written by a later instruction. That is, the same operand register is a source operand in the earlier instruction and a destination operand in the later instruction, and there are no instructions between them that write a different value to the register.

If the two instructions can be in the pipeline at the same time and there is a possibility that the later instruction will write its result before the earlier instruction has read its source operand value then the condition is called a *write after read* (WAR) hazard.

For example, consider the following code.

```
div.d   $f2, $f4, $f6
add.d   $f8, $f2, $f0
sub.d   $f0, $f10, $f12
```

Here, the value in $f0 is a source operand for the add.d instruction and a destination operand for the sub.d instruction. The add.d instruction executes incorrectly if it reads $f0 after the sub.d instruction has written to $f0. The following chart seems to indicate that this cannot happen. It assumes that the EX stage for a div.d instruction takes 8 cycles and the EX stage for a add.d or a sub.d instruction takes 3 cycles.

div.d	IF	ID	EX								MEM	WB
add.d		IF	ID	EX			MEM	WB				
sub.d			IF	ID	EX			MEM	WB			

However, the WAR hazard arises when the register read for the add.d instruction is delayed due to its RAW hazard with the div.d instruction regarding $f2. The following chart shows the real situation, assuming that register forwarding is used.

div.d	IF	ID	EX								MEM	WB			

add.d	IF	stalled				ID	EX	MEM	WB
sub.d		IF	ID	EX	MEM	WB			

Now the write from the sub.d instruction changes the value in $f0 before the add.d has had a chance to read the original value. The problem arises here because the add.d has a source operand, $f2, that must be read late due to a RAW hazard, while another source operand, $f0, should be read early to avoid a WAR hazard. If the two reads can be done at different times then the WAR hazard disappears.

Control Hazard.

In any set of instructions, there is normally a need for some kind of statement that allows the flow of control to be something other than sequential. Instructions that do this are included in every programming language and are called *branches*. In general, about 30% of all instructions in a program are branches. This means that branch instructions in the pipeline can reduce the throughput tremendously if not handled properly.

Whenever a branch is taken, *the performance of the pipeline is seriously affected. Each such branch requires a new address to be loaded into the program counter, which may invalidate all the instructions that are either already in the pipeline or prefetched i*n the buffer. This draining and refilling of the pipeline for each branch degrade the throughput of the pipeline to that of a sequential processor.

It is to be noted *that the presence of a branch statement does not automatically cause the pipeline to drain and begin refilling. A branch not taken allows the continued sequential flow of uninterrupted instructions to the pipeline*. Only when a branch is taken does the problem arise.

In general, branch instructions can be classified into three groups: *(1) unconditional branch, (2) conditional branch.*

An unconditional branch always alters the sequential program flow. It sets a new target address in the program counter, rather than incrementing it by 1 to point to the next sequential instruction address, as is normally the case.

A conditional branch sets a new target address in the program counter only when a certain condition, usually based on a condition code, is satisfied. Otherwise, the program counter is incremented by 1 as usual. In other words, a conditional branch selects a path of instructions based on a certain condition. If the condition is satisfied, the path starts from the target address and is called a *target path*. If it is not, the path starts from the next sequential instruction and is called a *sequential path*.

Conditional branches are the hardest to handle. As an example, consider the following conditional branch instruction sequence:

*i*1
*i*2 (conditional branch to *i*k)
*i*3
.
.
.
*i*k (target)
*i*k+1

Figure 3.11 shows the execution of this sequence in our instruction pipeline when the target path is selected. In this figure, *c* denotes the branch penalty, that is, the number of cycles wasted whenever the
target path is chosen.

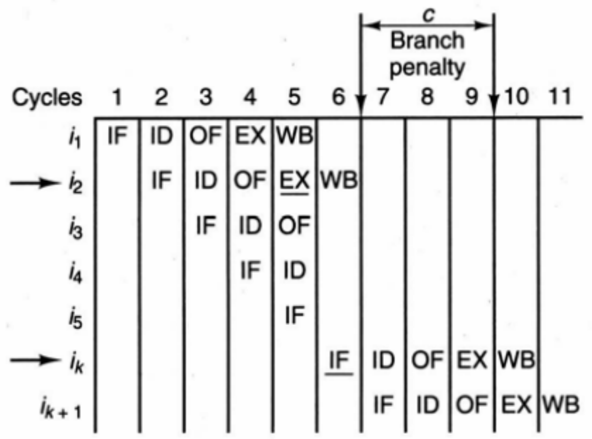


Figure 3.11 Branch Penalty.

Some of the better known techniques are branch prediction, delayed branching, and multiple prefetching. Each of these techniques is explained next.

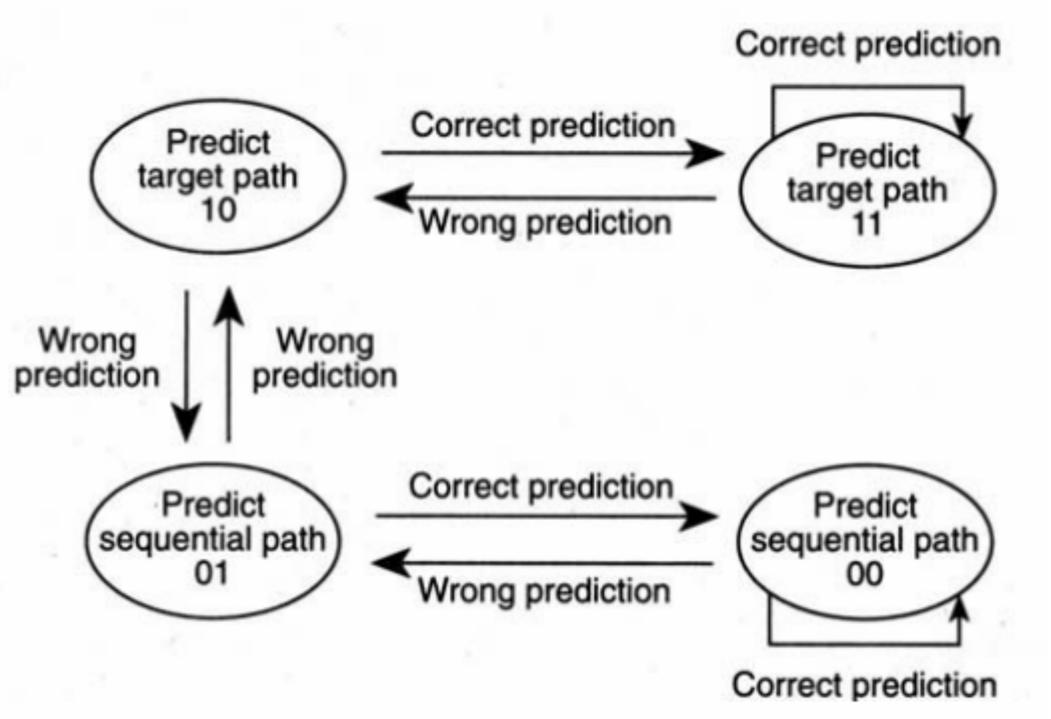
Branch Prediction.

In this type of design, the outcome of a *branch decision is predicted before the branch is actually executed*. Therefore, based on a particular prediction, the sequential path or the target path is chosen for execution. Although the chosen path often reduces the branch penalty, it may increase the penalty in case of incorrect prediction.
There are two types of predictions, static and dynamic.

In static prediction, a fixed decision for *prefetching one of the two paths* is made before the program runs. For example, a simple technique would be to always assume that the branch is taken. This technique simply loads the program counter with the target address
when a branch is encountered. Another such technique is to automatically choose one path (sequential or target) for some branch types and another for the rest of the branch types. If the chosen path is wrong, the pipeline is drained and instructions corresponding to the correct path are fetched; the penalty is paid.

In dynamic prediction, during the execution of the program the processor makes a *decision based on the past information* of the previously executed branches. For example, a simple technique would be to record the history of the last two paths taken by each branch instruction. If the last two executions of a branch instruction have chosen the same path, that path will be chosen for the current execution of the branch instruction. If the two paths do not match, one of the paths will be chosen randomly.

A better approach is to associate an n-bit counter with each branch instruction. This is known as the counter-based branch prediction approach. In this method, after executing a branch instruction for the first time, its counter, C, is set to a threshold, T, if the target path was taken, or to T-1 if the sequential path was taken. From then on, whenever the branch instruction is about to be executed, if $C \geq T$, then the target path is taken; otherwise, the sequential path is taken. The counter value C is updated after the branch is resolved. If the correct path is the target path, the counter is incremented by 1; if
not, C is decremented by 1. If C ever reaches 2n-1 (an upper bound), C is no longer incremented, even if the target path was correctly predicted and chosen. Likewise, C is never decremented to a value less than 0.
In practice, often n and T are chosen to be 2. Studies have shown that 2-bit predictors perform almost as well as predictors with more number of bits. The following diagram represents the possible states in a 2-bit predictor.



An alternative scheme to the preceding 2-bit predictor is to change the prediction only when the predicted path has been wrong for two consecutive times.

Most processors employ a small size cache memory called *branch target buffer* (BTB); sometimes referred to as *target instruction cache* (TIC). Often, each entry of this cache keeps a branch instruction's address with its target address and the history used by the prediction scheme.

When a branch instruction is first executed, the processor allocates an entry in the BTB for this instruction. When a branch instruction is
fetched, the processor searches the BTB to determine whether it holds an entry for the corresponding branch instruction. If there is a hit, the recorded history is used to determine whether the sequential or target path should be taken.

Static prediction methods usually require little hardware, but they may increase the complexity of the compiler. In contrast, dynamic prediction methods increase the hardware complexity, but they require less work at compile time.

In general, dynamic prediction obtains better results than static prediction and also provides a greater degree of object code compatibility, since decisions are made after compile time.

Delayed Branching.

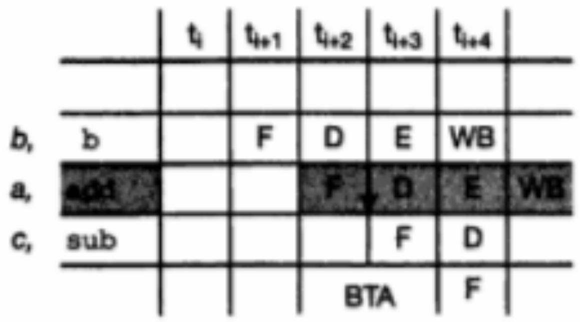
The delayed branching scheme eliminates or significantly reduces the effect of the branch penalty. In this type of design, a certain number of instructions after the branch instruction is fetched and executed regardless of which path will be chosen for the branch.

For example, a processor with a branch delay of *k* executes a path containing the next *k* sequential instructions and then either continues on the same path or starts a new path from a new target address.

As often as possible, the compiler tries to fill the next *k* instruction slots after the branch with instructions that are independent from the branch instruction. NOP (no operation) instructions are placed in any remaining empty slots.

As an example, consider the following case:

Principle of delayed branching



		t_i	t_{i+1}	t_{i+2}	t_{i+3}	t_{i+4}	
b,	b		F	D	E	WB	
a,	add			F	D	E	WB
c,	sub				F	D	
				BTA	BTA	F	

Branching to the target instruction (sub) is executed with one pipeline cycle of delay. This cycle is utilized to execute the instruction in the delay slot (add). Thus, delayed branching results in the following execution sequence:

a,	add
b,	b
c,	sub

This scheme is applicable to both conditional and unconditional branches.

Multiple Prefetching.

In this type of design, the processor fetches both possible paths. Once the branch decision is made, the unwanted path is thrown away.

By prefetching both possible paths, the fetch penalty is avoided in the case of an incorrect prediction. *To fetch both paths, two buffers are employed to service the pipeline.* In normal execution, the first buffer is loaded with instructions from the next sequential address of the branch instruction. *If a branch occurs, the contents of the first buffer are invalidated, and the secondary buffer, which has been loaded with instructions from the target address of the branch instruction, is used as the primary buffer.*

This double buffering scheme ensures a constant flow of instructions and data to the pipeline and reduces the time delays caused by the draining and refilling of the pipeline.

List the challenges of implementing pipelining?

Challenges of implementing pipelining can be listed as follows:

a. *Because the divide unit is not fully pipelined, structural hazards can occur*
b. *Because the instructions have varying running times, the number of register writes required in a cycle can be larger than 1*
c. *WAW hazards are possible, since instructions no longer reach WB in order WAR hazards are NOT possible, since register reads are still taking place during the ID stage*
d. *Instructions can complete in a different order than they were issued, causing problems with exceptions*
e. *Longer latency of operations makes stalls for RAW hazards more frequent*

3. **What is impact of pipelining in Instruction Set Design?**

RISC machines like MIPS are well suited for pipelining. Instruction format is simple and execution relatively uniform. Pipelining is more complex for CISC machines, because the instructions may take different lengths of time to execute. However, RISC-style pipelining is now incorporated into high-performance CISC processors (such as the Pentium and Core 2) by translating most instructions into a series of RISC-like operations.

Execution of an instruction in a pipelined processor differs from execution in a non pipelined processor. Most of the instructions execute smoothly in pipeline without reducing the pipeline performance. But, some instructions either reduce pipeline performance or cause side effects. Since all modern processors are pipelined, impact of various instructioins in pipeline execution should be taken into consideration while finalizing the instruction set of processors.

Addressing modes.

Complex addressing modes are avoided in pipelined processors. Instructions using complex addressing modes take long execution time and likely to cause pipeline stall. A processor designer prefers to have following features in addressing modes:

1. Operand fetch should not require more than one memory access
2. No side effects should be caused
3. Instructions other than load and store should not access memory for operand fetch or result store.

These feature are present in three basic addressing modes: register addressing, register indirect addressing and index addressing. They do not cause side effects. The index addressing mode involves operand address calculation in one clock cycle, and the operand fetch in next clock cycle. The other two addressing modes do not involve operand address calculation.

Condition codes:

While compiler tries to reorder the instructions to avoid pipeline stall, it should not change the program logic due to instruction reordering. The condition codes set (or reset) by certain instruction should be taken into account by the compiler so as not to disturb the program logic due to instruction reordering. To simplify the compiler’s task, the instruction set designer should choose only few instructions that affect condition codes. In addition, the compiler should have provision to indicate the instruction which have affected the condition codes.

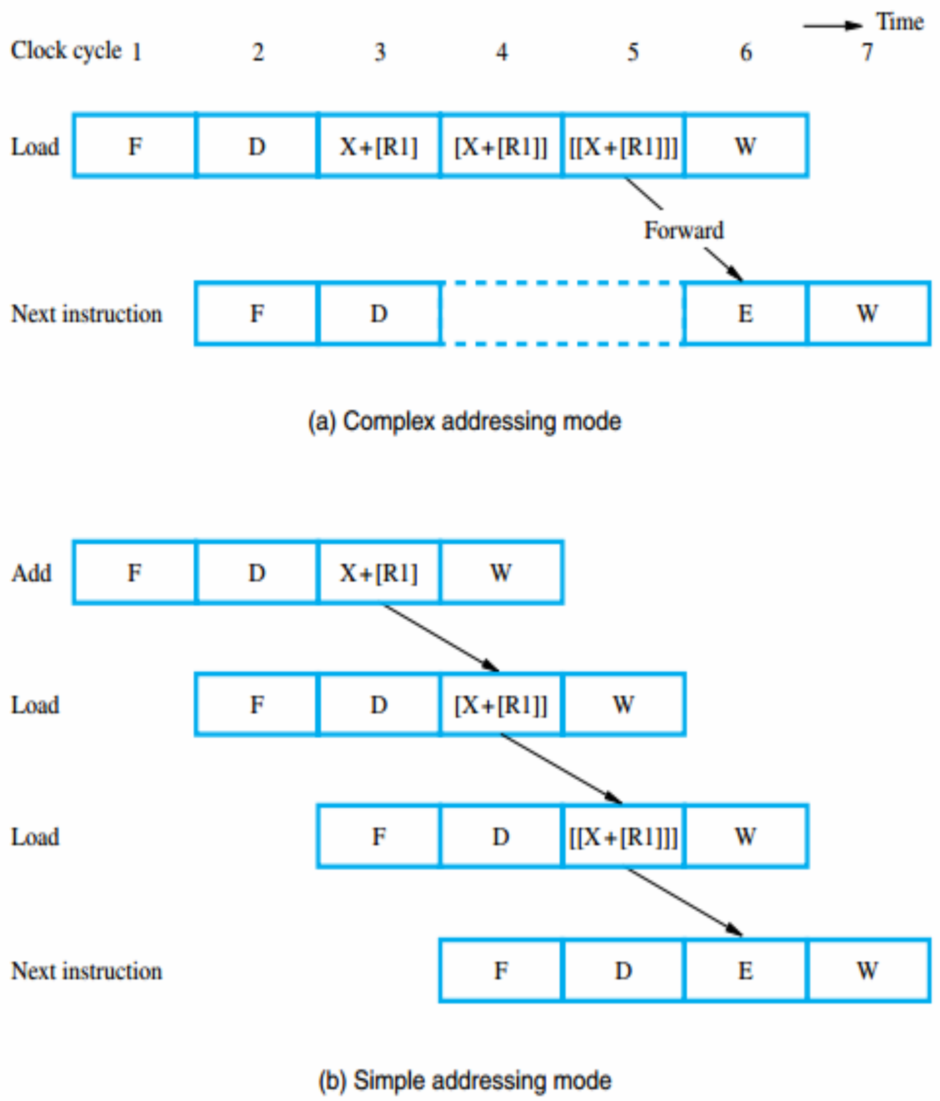

Time
Clock cycle 1 2 3 4 5 6 7
Load
F
D
X+[R1]
[X+[R1]]
[[X+[R1]]]
W
Forward
Next instruction
E
(a) Complex addressing mode
Add
(b) Simple addressing mode

Figure 8.16 Equivalent operations using complex and simple addressing modes.

3.6 Dynamic instruction scheduling

Data hazards in a program cause a processor to stall. With static scheduling the compiler tries to reorder these instructions during compile time to reduce pipeline stalls. It uses less hardware and can use more powerful algorithms. With dynamic scheduling the hardware tries to rearrange the instructions during run-time to reduce pipeline stalls. It uses simpler compiler and handles dependencies not known at compile time. It allows code compiled for a different machine to run efficiently.

Dynamic scheduling offers several advantages.

1. First, it allows code that was compiled with one pipeline to run efficiently on a different pipeline, eliminating the need to recompile for a different microarchitecture.

2. Second, it enables handling of code when dependences are unknown at time of compilation.
3. Third, and most important, it allows the processor to tolerate unpredictable delays.

Instruction issue: The process of letting an instruction move from the instruction decode stage (ID) into the execution stage (EX)of this pipeline.

•An instruction that has made this step is said to have *issued*.

•A pipeline fetches an instruction and issues it,

–if there is no *data dependence* between an instruction already in the pipeline and the fetched instruction that cannot be hidden with forwarding.

•If there is a data dependence that cannot be hidden, then the hazard detection hardware stalls the pipeline (starting with the instruction that uses the result).

•*Forwarding* logic reduces the effective pipeline latency so that the certain dependences do not result in hazards. *No* new instructions are fetched or issued until the dependencies cleared.

In case of Static scheduling: Compiler techniques can be used for scheduling the instructions to separate dependent instructions.

While in dynamic scheduling: the hardware rearranges the instruction execution to reduce the stalls.

Its advantages can be listed as follows:

–It simplifies the compiler

–It enables handling some cases when dependences are unknown at compile time.

–It allows code that was compiled with one pipeline in mind to run efficiently on a different pipeline.

The disadvantage is increase in hardware complexity.

Dynamically scheduled processor tries to avoid stalling when dependences are present.

In our previous model, all instructions executed in the order that they appear This can lead to unnecessary stalls. Consider following sets of instructions:

DIVD FO, F2, F4

ADDD F10, F0, F8

SUBD F12, F8, F14

Here, SUBD stalls waiting for the ADDD to go first, even though SUBD does not have a data dependency.

Scoreboarding

The scoreboard implements a centralized control scheme that detects all resource and data hazards which allows instructions to execute out-of-order when no resource hazards or data dependencies. With out-of-order execution, the SUBD is allowed to executed before the add. This can lead to out of order completion, which can cause WAW and WAR hazards.

The solution for WAW is to detect WAW hazard before reading operands and stall write until other instruction completes.

Similarly, the Solutions for WAR is to detect WAR hazards before writing back to the register files and stall the write back

It was first implemented in 1964 by the CDC 6600. This scoreboard does not take advantage of forwarding (i.e. bypasses), since it waits until both results are written back to the register file. It's pipelining has four stages as follows:

• Decode+Issue (Issue)

  – decode instructions

  – check for structural and WAW hazards

  – stall until structural and WAW hazards are resolved

• Read operands (Read)

  – wait until no RAW hazards

  – then read operands

• Execution (EX)

  – operate on operands

  – may be multiple cycles

  - notify scoreboard when done

• Write result (WB)

– finish execution

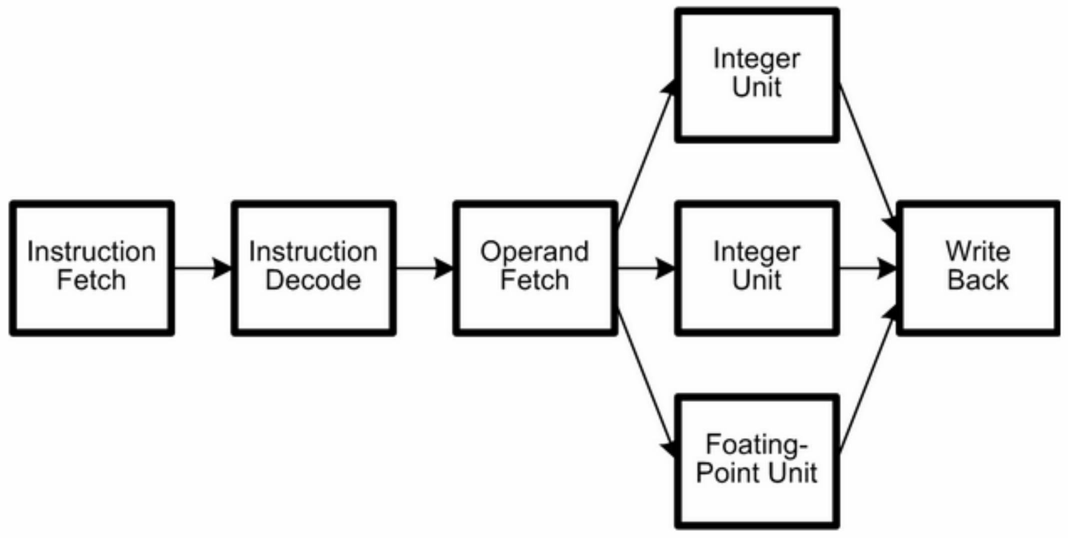
– stall if WAR hazard

4. **Discuss superscalar and super pipelining architecture.**

Superscalar Architecture (SSA) describes a microprocessor design that execute more than one instruction at a time during a single clock cycle. In a SSA design, the processor or the instruction compiler is able to determine whether an instruction can be carried out independently of other sequential instructions, or whether it has a dependency on another instruction and must be executed sequentially. The design is sometimes called "Second Generation RISC". Another term used to describe superscalar processors is multiple instruction issue processors.

In a SSA, several scalar instructions can be initiated simultaneously and executed independently. A long series of innovations aimed at producing ever-faster microprocessors. It includes all features of pipelining but, in addition, there can be several instructions executing simultaneously in the same pipeline stage.

SSA introduces a new level of parallelism, called instruction-level parallelism. In Superscalar CPU Architecture implementation of Instruction Level Parallelism (ILP) within a single processor allows faster CPU at a given clock rate. A superscalar processor executes more than one instruction during a clock cycle by simultaneously dispatching multiple instructions to functional units. Each functional unit is not a separate CPU core but an execution resource within a single CPU such as an arithmetic logic unit, a bit shifter, or a multiplier.



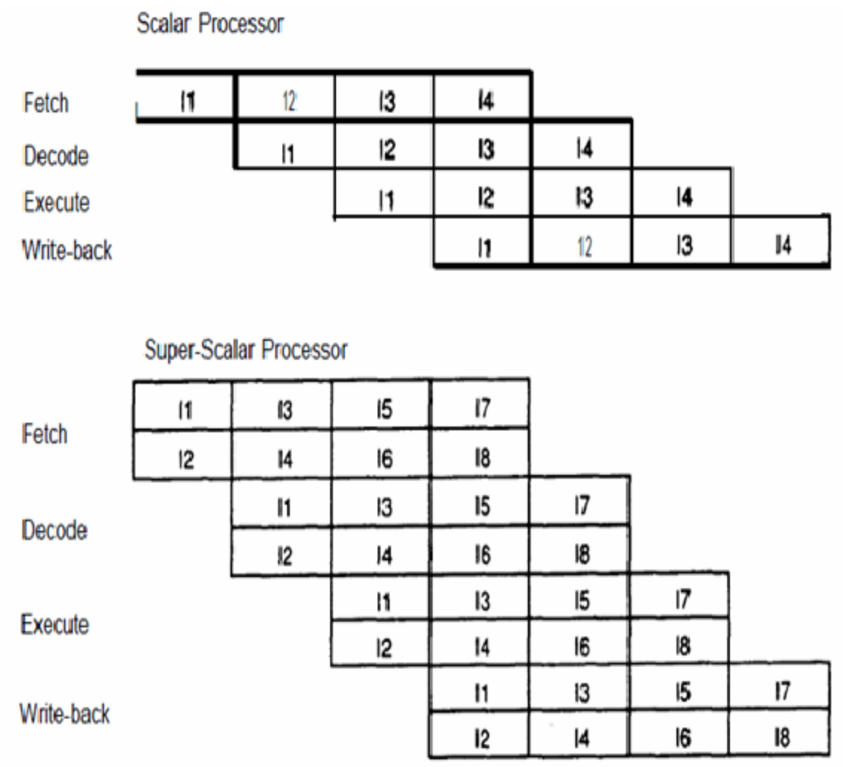
A Superscalar Processor with 3 Functional Units

The simplest processors are scalar processors. Each instruction executed by a scalar processor typically manipulates one or two data items at a time. In a superscalar CPU the dispatcher reads instructions from memory and decides which one can be run in parallel. Therefore, a superscalar processor can be proposed having multiple parallel pipelines, each of which is processing instructions simultaneously from a single instruction thread.

Pipelining in Superscalar Architecture

Pipelining is the process of breaking down task into sub steps and executing them in different parts of processor. In order to fully utilise a superscalar processor of degree *m* with pipelining, *m* instructions must be executable parallelly. This situation may not be true in all clock cycles.

In a superscalar processor, the simple operation latency should require only one cycle, as in the base scalar processor. Pipelining is the process of breaking down task into sub steps and executing them in different parts of processor. Following diagram shows the pipelined execution in scalar and super scalar processer:



Effect of Dependencies

The situations which prevent instructions to be executed in parallel by SSA are very similar to those which prevent efficient execution on a pipelined architecture (pipeline hazards):

Resource conflicts.

Control (procedural) dependency.

Data dependencies.

Their consequences on SSA are more severe than those on simple pipelines, because the potential of parallelism in SSA is greater and, thus, a larger amount of performance will be lost.

Resource conflicts.

Several instructions compete for the same hardware resource at the same time.

e.g., two arithmetic instructions need the same floating-point unit for execution.

It is similar to structural hazards in pipeline. They can be solved partly by introducing several hardware units for the same functions. e.g., have two floating-point units. The hardware units can also be pipelined to support several operations at the same time.

Procedural conflicts

The presence of branches creates major problems in assuring the optimal parallelism. It cannot execute instructions after a branch in parallel with instructions before a branch. It is similar to control hazards in pipeline. If instructions are of variable length, they cannot be fetched and issued in parallel, since an instruction has to be decoded in order to identify the following one. Therefore, superscalar techniques are more efficiently applicable to RISCs, with fixed instruction length and format.

Data conflicts

It is caused by data dependencies between instructions in the program.

It is similar to date hazards in pipeline. To address the problem and to increase the degree of parallel execution, SSA provides a great liberty in the order in which instructions are issued and executed. Therefore, data dependencies have to be considered and dealt with much more carefully.

Due to data dependencies, only some part of the instructions are potential subjects for parallel execution. In order to find instructions to be issued in parallel, the processor has to select from a sufficiently large instruction sequence.

There are usually a lot of data dependencies in a short instruction sequence.

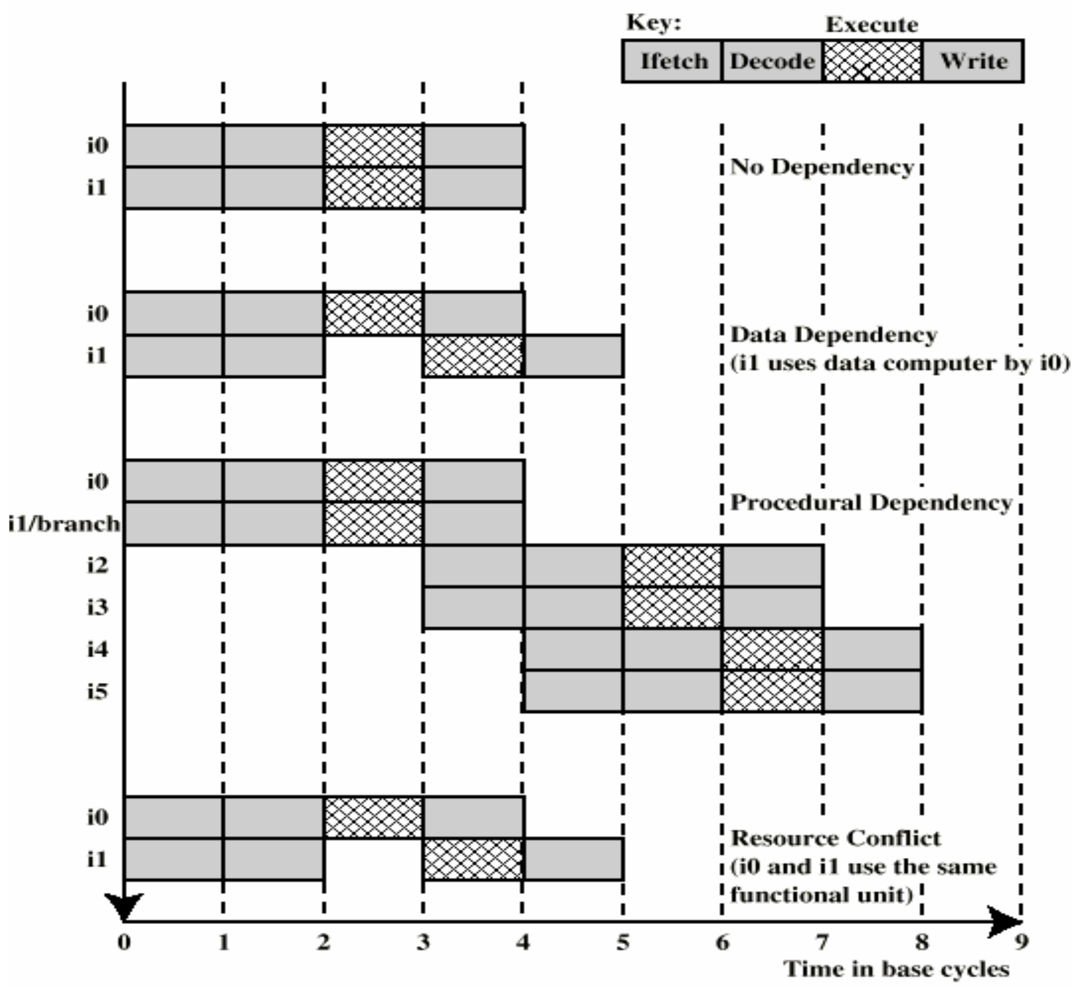
Window of execution is defined as the set of instructions that is considered for execution at a certain moment. The number of instructions in the window should be as large as possible. However, this is limited by:

Capacity to fetch instructions at a high rate.

The problem of branches.

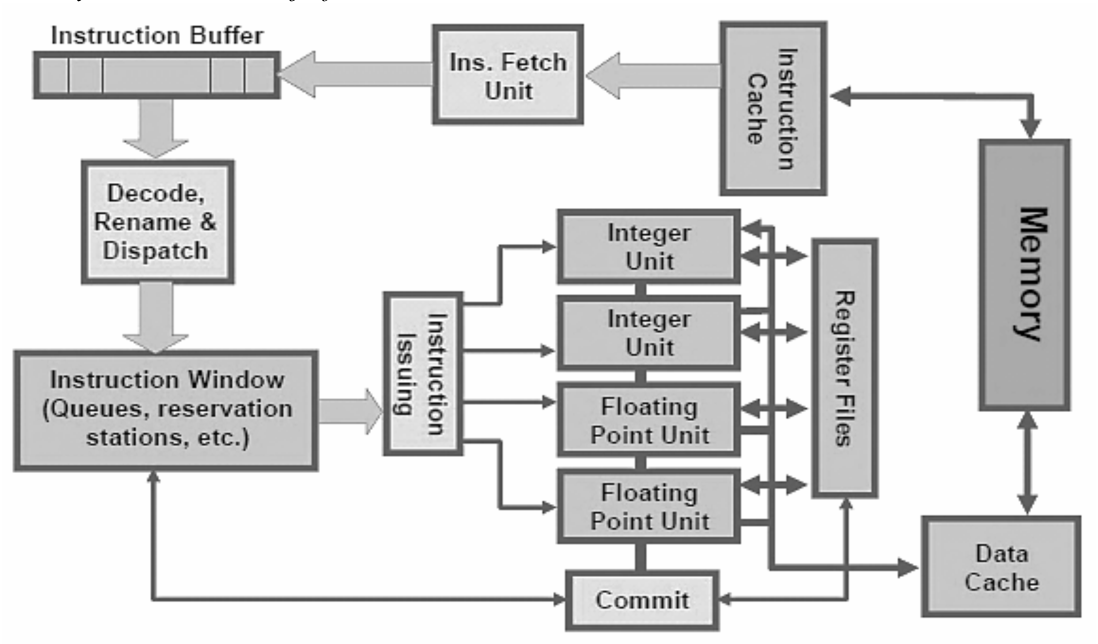
The cost of hardware needed to analyze data dependencies.

Following diagram shows the effects of dependency in superscalar architecture:



Instruction Flow in Superscalar Architecture

SSA allows several instructions to be issued and completed per clock cycle. It consists of a number of pipelines that are working in parallel. Depending on the number and kind of parallel units available, a certain number of instructions can be executed in parallel. In the following example two floating point and two integer operations can be issued and executed simultaneously. Each unit is also pipelined and can execute several operations in different pipeline stages.



Superpipelining

Superpipelining is based on dividing the stages into several sub-stages, and thus increasing the number of instructions which are handled by the pipeline at the same time.

For example, by dividing each stage into two sub-stages, a pipeline can perform at twice the speed in the ideal situation:

- Tasks that require less than half a clock cycle.
- No duplication of hardware is needed for these stages.

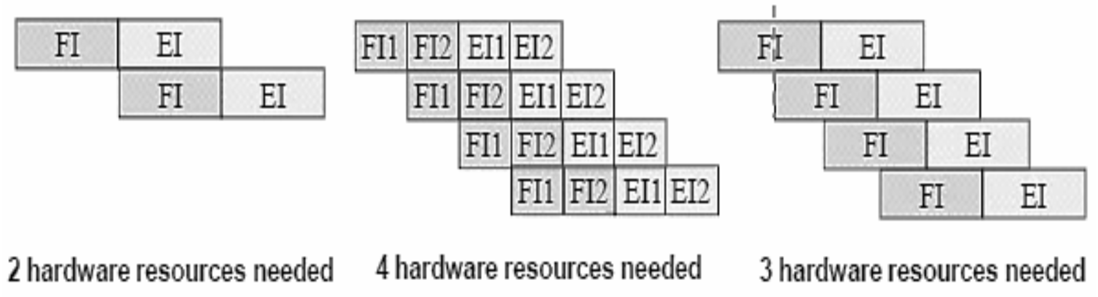

FI EI
FI1 FI2 EI1 EI2
2 hardware resources needed
4 hardware resources needed
3 hardware resources needed

Figure: Duplication of hardware is for Superscalar

Benefit:

1. The major benefit of superpipelining is the increase in the number of instructions which can be in the pipeline at one time and hence the level of parallelism.

Drawbacks

1. The larger number of instructions "in flight" (*ie* in some part of the pipeline) at any time, increases the potential for data dependencies to introduce stalls. Simulation studies have suggested that a pipeline depth of more than 8 stages tends to be counter-productive.

2. Superscalar machines can issue several instructions per cycle. Superpipelined machines can issue only one instruction per cycle, but they have cycle times shorter than the time required for any operation.

3. Both of these techniques exploit instruction-level parallelism, which is often limited in many applications. Superpipelined machines are shown to have better performance and less cost than superscalar machines.

RISC architecture better suited for pipeline processing than CISC

RISC uses simple instructions which can be executed within one clock cycle, whereas primary goal of CISC is to complete a task in as few lines as possible.

For example, consider task of multiplying two numbers. CISC architecture would come prepared with a specific instruction for doing this task. Whereas in RISC you would need to perform three or four instructions for completing the same task.

RISC architectures lend themselves more towards pipelining than CISC architectures for many reasons. As RISC architectures have a smaller set of instructions than CISC architectures in a pipeline architecture the time required to fetch and decode for CISC architectures is unpredictable.

The difference in instruction length with CISC will hinder the fetch decode sections of a pipeline, a single byte instruction following an 8 byte instruction will need to be handled so as not to slow down the whole pipeline. In RISC architectures the fetch and decode cycle is more predictable and most instructions have similar length.

CISC architectures by their very name also have more complex instructions with complex addressing modes. This makes the whole cycle of processing an instruction more complex. Pipelining requires that the whole fetch to execute cycle can be split into stages where each stage does not interfere with the next and each instruction can be doing something at each stage. RISC architectures because of their simplicity and small set of instructions are simple to split into stages. CISC with more complex instructions are harder to split into stages. Stages that are important for one instruction may be not be required for another instruction with CISC.

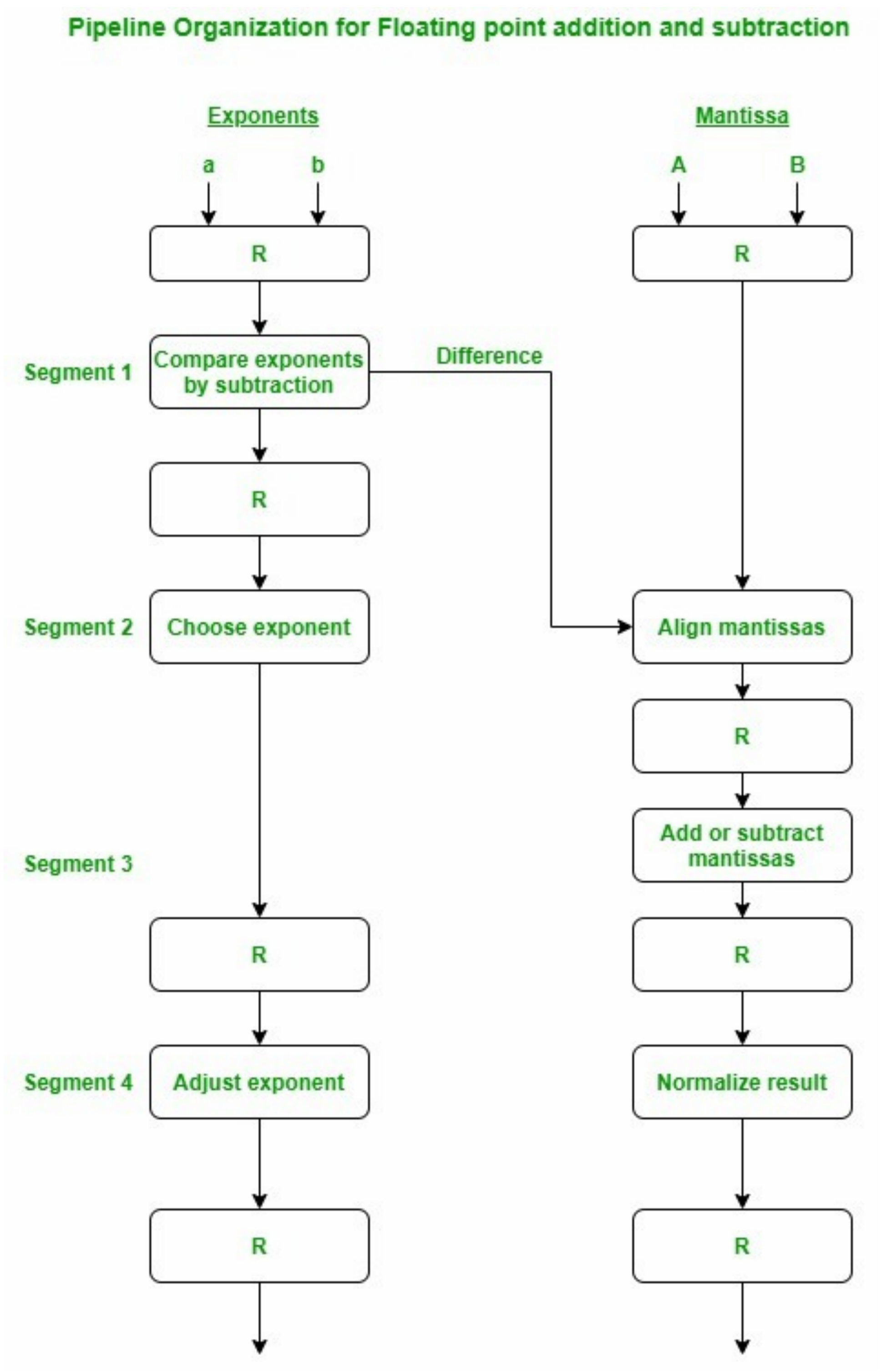
The rich set of addressing modes that are available in CISC architectures can cause data hazards when pipelining is introduced. Data hazards which are unlikely to occur in RISC architectures due to the smaller subset of instructions and the use of load store instruction to access memory become a problem in CISC architectures. CISC instructions that write results back to memory need to be handled carefully. Forwarding solutions used for allowing result written to registers to be available for input in the next instruction become more complex when memory locations which can be addressed in various modes can be accessed. Write after Read hazard must be taken care of where CISC instruction may auto increment a register early in the stages which may be used by the previous instruction at a later stage.

CISC added complexity makes for larger pipeline lengths to take into account more decoding and checking. This is because with a large number of logic gates for a fetch execute cycle the additional gates required for stages have less impact. As RISC architectures have simpler instruction sets than CISC the number of gates involved in the fetch execute cycle compare will be far lower than this in CISC architecture. Therefore, RISC architectures will tend to have smaller optimum pipeline lengths than more general processors.

RISC architectures do suite pipelining more than CISC architectures and do lend themselves to smaller pipelines. This does not mean however that CISC architecture cannot gain from pipelining or that a large number of pipeline stages are bad (although the flushing of a pipeline would become of concern).

3.3 Arithmetic Pipeline :

An arithmetic pipeline divides an arithmetic problem into various sub problems for execution in various pipeline segments. It is used for floating point operations, multiplication and various other computations. The process or flowchart arithmetic pipeline for floating point addition is shown in the diagram.



Floating point addition using arithmetic pipeline :
The following sub operations are performed in this case:

1. Compare the exponents.

2. Align the mantissas.

3. Add or subtract the mantissas.
4. Normalize the result

First of all the two exponents are compared and the larger of two exponents is chosen as the result exponent. The difference in the exponents then decides how many times we must shift the smaller exponent to the right. Then after shifting of exponent, both the mantissas get aligned. Finally the addition of both numbers take place followed by normalisation of the result in the last segment.

Example:
Let us consider two numbers,
X=0.321410^3 and Y=0.450010^2

Explanation:
First of all the two exponents are subtracted to give 3-2=1. Thus 3 becomes the exponent of result and the smaller exponent is shifted 1 times to the right to give
Y=0.045010^3

Finally the two numbers are added to produce

Z=0.366410^3

As the result is already normalized the result remains the same.

3.7 Vector Instruction Types

Vector instructions are specialized instructions designed to perform operations on multiple data elements simultaneously, typically used in vector processors or SIMD (Single Instruction, Multiple Data) architectures. These instructions are essential for accelerating computations in applications involving large datasets, such as scientific computing, graphics processing, and machine learning.

Vector instructions are designed to process multiple data elements simultaneously, making them a cornerstone of SIMD (Single Instruction, Multiple Data) architectures. These instructions operate on **vector registers**, which can hold large arrays of data, enabling **element-wise operations** such as arithmetic, logical, and comparison tasks to be performed in parallel. They support advanced **memory access patterns**, including strided and gather/scatter operations, to handle complex data structures efficiently. Additionally, vector instructions often incorporate **masking and predication**, allowing selective processing of elements based on conditions, and **reduction operations**, which aggregate values across a vector, such as summing or finding maximum/minimum values. These capabilities make vector instructions highly effective for accelerating tasks in scientific computing, graphics processing, machine learning, and signal processing. However, their performance depends on factors like vector length, memory bandwidth, and the ability to minimize overhead during data loading and setup.

Types of Vector Instructions:

1. Arithmetic Instructions: Perform basic arithmetic operations (e.g., addition, subtraction, multiplication, division) on vector elements.

2. Logical Instructions: Perform bitwise logical operations (e.g., AND, OR, XOR) on vector elements.

3. Load/Store Instructions: Move data between memory and vector registers, often optimized for contiguous or strided access patterns.

4. Comparison Instructions: Compare elements of two vectors and generate a mask or result vector based on the comparison.

5. Shuffle/Permute Instructions: Rearrange elements within a vector or between vectors, enabling data reorganization.

6. Reduction Instructions: Aggregate values across a vector, such as summing all elements or finding the maximum/minimum value.

7. Specialized Instructions: Some vector processors include instructions for specific domains, such as trigonometric functions, matrix operations, or cryptographic algorithms.

Applications of vector processing:

- Scientific Computing: Vector instructions accelerate simulations, numerical analysis, and linear algebra operations.

- Graphics and Gaming: Used for rendering, image processing, and physics simulations.

- Machine Learning: Optimize matrix multiplications, convolutions, and other tensor operations.

- Signal Processing: Enhance performance in audio, video, and communication systems.

3.8 Vector Performance Modeling

Vector performance modeling involves analyzing and predicting the performance of vector processors or SIMD architectures. It helps in understanding how efficiently a system can execute vectorized code and identifies potential bottlenecks.

Vector performance modeling focuses on analyzing and predicting the efficiency of vector processors or SIMD architectures. Key factors include **vector length**, which determines the degree of parallelism, and **clock speed**, which affects the rate of instruction execution. **Memory**

bandwidth plays a critical role in ensuring data can be fed to the processor quickly, while **instruction throughput** reflects the number of vector operations completed per cycle. Performance is also influenced by **data dependencies**, which can limit parallelism, and **overhead** from tasks like loading data into vector registers. Metrics such as **peak performance**, **actual performance**, **speedup**, and **efficiency** are used to evaluate performance, with techniques like analytical models, simulation, profiling, and the Roofline Model helping to identify bottlenecks. This modeling is essential for optimizing both hardware design and software implementation, ensuring that vector processors deliver maximum performance for applications like high-performance computing, real-time systems, and data-intensive tasks.

Performance Metrics:

1. Peak Performance: The theoretical maximum performance of a vector processor, calculated as:

$$\text{Peak Performance} = \text{Vector Length} \times \text{Clock Speed} \times \text{Operations per Cycle}$$

2. Actual Performance: The observed performance during execution, which is often lower than peak performance due to factors like memory bottlenecks, dependencies, and overhead.

3. Speedup: The improvement in performance achieved by using vector instructions compared to scalar instructions, calculated as:

$$\text{Speedup} = \frac{\text{Scalar Execution Time}}{\text{Vector Execution Time}}$$

4. Efficiency: The ratio of actual performance to peak performance, indicating how effectively the hardware resources are utilized.

Factors Affecting Performance:

1. Data Alignment: Misaligned data can reduce memory access efficiency and increase latency.

2. Vectorization Ratio: The proportion of the code that can be vectorized; higher ratios lead to better performance.

3. Memory Access Patterns: Contiguous access patterns are generally faster than strided or gather/scatter patterns.

4. Instruction Mix: The combination of arithmetic, logical, and memory operations in the workload affects overall performance.

5. Hardware Limitations: Constraints such as the number of vector registers, memory bandwidth, and pipeline depth can limit performance.

Modeling Techniques:

1. Analytical Models: Use mathematical formulas to estimate performance based on hardware specifications and workload characteristics.

2. Simulation: Simulate the execution of vectorized code on a virtual processor to measure performance metrics.

3. Profiling: Run the code on actual hardware and collect performance data using profiling tools to identify bottlenecks.

4. Roofline Model: A visual performance model that plots achievable performance against operational intensity (operations per byte of memory access), helping to identify whether the workload is compute-bound or memory-bound.

Applications:

- Hardware Design: Helps architects optimize vector processors for specific workloads.

- Software Optimization: Guides developers in writing efficient vectorized code.

- Performance Tuning: Identifies bottlenecks and suggests improvements for existing systems.

3.9 Vectorization

Vectorization in parallelism is a technique that uses the same flow of control to perform operations on multiple data elements simultaneously. It's a type of Single Instruction Multiple Data (SIMD) parallelism.

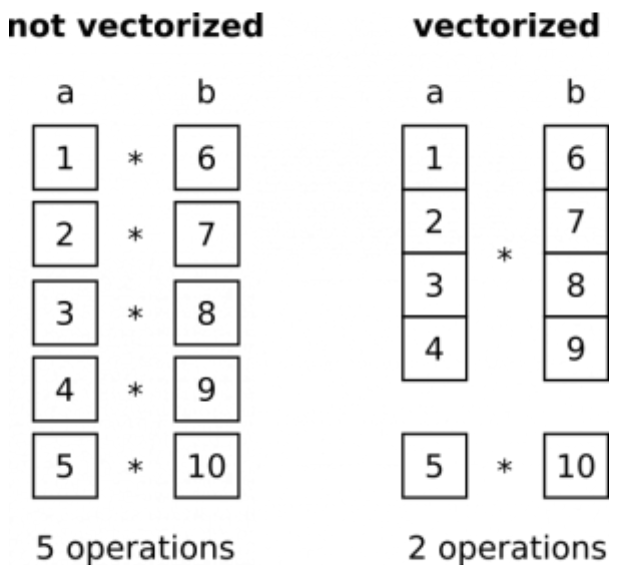


Fig: *Multiplication vectorized and not vectorized*

Vectorization is the process of converting scalar operations, which process individual data elements, into vector operations that process multiple data elements simultaneously. This transformation leverages the parallelism offered by vector processors or SIMD (Single Instruction, Multiple Data) architectures to improve computational efficiency and performance.

Vectorization is the process of transforming scalar operations, which process individual data elements, into vector operations that handle multiple data elements simultaneously, leveraging the parallelism of vector processors or SIMD architectures. It focuses on **data-level parallelism**, enabling the same operation to be applied concurrently across arrays or vectors of data. Key techniques include **loop transformation**, where repetitive operations in loops are restructured for parallel execution, and **dependency analysis**, which ensures that data dependencies do not hinder parallelism. Efficient vectorization also requires **data alignment** to optimize memory access and minimize latency. While vectorization significantly boosts performance in applications like scientific computing, graphics, and machine learning, it introduces challenges such as handling edge cases, managing overhead from data reorganization, and addressing irregular memory access patterns. Vectorization can be **manual**, where programmers explicitly write vectorized code, or **automatic**, where compilers identify and apply optimizations.

Types of Vectorizations:

1. Manual Vectorization: Programmers explicitly rewrite code to use vector instructions, often requiring deep knowledge of the hardware and instruction set.

2. Automatic Vectorization: Compilers automatically identify opportunities for vectorization and generate vectorized code, reducing the burden on programmers.

3. Partial Vectorization: Only certain parts of the code are vectorized, often due to limitations such as data dependencies or irregular memory access patterns.

3.10 Vectorizing Compiler Design

A vectorizing compiler is a specialized compiler that automatically transforms scalar code into vectorized code, enabling efficient execution on vector processors or SIMD architectures. The design of such compilers involves advanced analysis and optimization techniques to identify and exploit parallelism.

- The compiler analyzes the source code to understand dependencies between statements
- The compiler rearranges the instructions to use vector instructions
- Vector instructions operate on multiple data items at once

A vectorizing compiler is designed to automatically transform scalar code into vectorized code, enabling efficient execution on vector processors or SIMD architectures. It employs advanced techniques such as **dependency analysis** to identify and resolve data dependencies that could prevent parallel execution, and **loop transformation** to restructure loops for optimal parallelism. The compiler generates **vector instructions** and ensures proper **data alignment** to maximize memory bandwidth utilization. It also uses **cost modeling** and **optimization heuristics** to balance performance gains against potential overhead, such as data reorganization or edge-case handling. A key feature of vectorizing compilers is their **target architecture awareness**, allowing them to leverage specific hardware capabilities like vector register size and instruction sets. Challenges in designing such compilers include managing complex dependencies, handling irregular memory access patterns, and minimizing overhead. These compilers are critical for optimizing performance in high-performance computing, multimedia processing, and machine learning applications.

Types of vectorizations

- **Loop vectorization**: Multiple iterations of a loop are executed simultaneously
- **Basic block vectorization**: Independent scalar instructions are combined into SIMD instructions

Phases of Vectorizing Compiler Design:

1. Front-End Analysis: The compiler parses the source code and performs lexical, syntactic, and semantic analysis to understand the program structure.

2. Intermediate Representation (IR): The code is transformed into an intermediate representation that facilitates analysis and optimization.

3. Dependency and Parallelism Analysis: The compiler identifies data dependencies and opportunities for parallelism, particularly within loops.

4. Loop Transformation: The compiler applies transformations like loop unrolling, loop fusion, and loop fission to optimize vectorization.

5. Code Generation: The compiler generates vector instructions and ensures proper data alignment and memory access patterns.

6. Back-End Optimization: The compiler performs target-specific optimizations to maximize performance on the vector processor.

Challenges in Vectorizing Compiler Design:

1. Complex Dependencies: Handling complex data dependencies, especially in nested loops, can be challenging.

2. Irregular Memory Access: Non-contiguous or irregular memory access patterns can limit the effectiveness of vectorization.

3. Hardware Limitations: The compiler must account for hardware constraints, such as vector register size and memory bandwidth.

4. Overhead Management: The compiler must minimize the overhead associated with data reorganization, alignment, and edge-case handling.

3.2 Structures and Algorithms of Array Processors (SIMD Computers)

Array processors, also known as SIMD (Single Instruction, Multiple Data) computers, are parallel computing systems designed to perform the same operation on multiple data points simultaneously. They consist of multiple processing elements (PEs) that operate in lockstep under the control of a single instruction stream, making them highly efficient for data-parallel tasks.

3.2.1 SIMD Array Processors

Components of SIMD processors

1. Processing Elements (PEs): SIMD array processors consist of multiple PEs, each capable of performing arithmetic and logical operations. All PEs execute the same instruction simultaneously on different data elements.

2. Control Unit: A single control unit broadcasts instructions to all PEs, ensuring synchronized execution.

3. Data Parallelism: SIMD architectures excel at data-parallel tasks, where the same operation is applied to multiple data elements concurrently.

4. Memory Organization: Data is typically stored in a shared memory or distributed across local memories attached to each PE. Efficient memory access is critical for performance.

5. Scalability: The number of PEs can be scaled to increase computational power, but this requires careful design to avoid bottlenecks in memory access and communication.

6. Applications: SIMD array processors are widely used in scientific computing, image processing, machine learning, and multimedia applications.

3.2.2 SIMD Interconnection Networks

Interconnection networks in SIMD array processors facilitate communication between processing elements (PEs) and memory. The design of these networks is crucial for ensuring efficient data exchange and minimizing latency.

Common features of SIMD interconnection networks are:

1. Topology: Common topologies include mesh, hypercube, and torus, which determine how PEs are connected and how data flows between them.

2. Communication Patterns: SIMD networks support regular communication patterns, such as nearest-neighbor communication in a mesh or global broadcasting.

3. Latency and Bandwidth: The network must provide low latency and high bandwidth to avoid bottlenecks, especially for data-intensive applications.

4. Scalability: The network should scale efficiently as the number of PEs increases, maintaining performance without excessive complexity.

5. Fault Tolerance: Some networks incorporate redundancy to handle failures in PEs or communication links.

Types of Interconnection Networks:

1. Mesh Networks: PEs are arranged in a grid, and each PE communicates with its immediate neighbors. Suitable for applications with localized data dependencies.

2. Hypercube Networks: PEs are connected in a hypercube topology, providing logarithmic communication steps. Ideal for applications requiring global communication.

3. Torus Networks: Similar to mesh networks but with wrap-around connections, reducing edge effects and improving communication efficiency.

3.2.3 Parallel Algorithms for Array Processors

Parallel algorithms for array processors are designed to exploit the data-parallel capabilities of SIMD architectures. These algorithms divide tasks into smaller subtasks that can be executed concurrently across multiple PEs.

Factors to be considered in parallel algorithm for array processor

1. Data Decomposition: The problem is divided into smaller data chunks, each processed by a separate PE.

2. Synchronization: PEs operate in lockstep, ensuring that all operations are synchronized.

3. Load Balancing: Tasks are distributed evenly across PEs to maximize utilization and minimize idle time.

4. Communication Efficiency: Algorithms minimize communication overhead by optimizing data exchange patterns.

General Practice Questions

1. Explain the concept of pipelining in computer architecture. How does pipelining improve the performance of a processor? Provide a diagram to illustrate the stages of a basic pipeline.
2. Differentiate between linear and non-linear pipelines. Discuss the advantages and disadvantages of non-linear pipelines in high-performance computing.
3. What are the key differences between RISC and CISC architectures in terms of pipelining? Why is RISC better suited for pipelining compared to CISC?
4. Describe the five stages of the DLX pipeline (IF, ID, EX, MEM, WB). How does each stage contribute to the execution of an instruction?
5. What are pipeline hazards? Explain the three types of pipeline hazards (structural, data, and control) with examples.
6. Discuss the concept of dynamic instruction scheduling. How does it help in reducing pipeline stalls, and what are its advantages over static scheduling?
7. Explain the role of vector instructions in SIMD architectures. Provide examples of vector instruction types and their applications in scientific computing.
8. What is vectorization? Discuss the challenges of vectorizing code and how a vectorizing compiler addresses these challenges.
9. Describe the concept of SIMD (Single Instruction, Multiple Data) architecture. How do SIMD array processors achieve data parallelism?
10. Explain the role of interconnection networks in SIMD array processors. Compare mesh, hypercube, and torus topologies in terms of scalability and communication efficiency.
11. What are the key components of an SIMD array processor? Discuss the role of processing elements (PEs) and the control unit in SIMD architectures.
12. Discuss the concept of superscalar architecture. How does it differ from superpipelining, and what are the advantages of each approach?
13. Explain the concept of instruction-level parallelism (ILP). How do superscalar processors exploit ILP to improve performance?
14. What are the challenges of implementing superscalar architectures? Discuss the impact of data dependencies, resource conflicts, and procedural dependencies.
15. Explain the concept of branch prediction in pipelined processors. Compare static and dynamic branch prediction techniques.
16. What is the role of a vectorizing compiler in optimizing code for SIMD architectures? Discuss the phases of vectorizing compiler design.

17. Explain the concept of arithmetic pipelining. How does an arithmetic pipeline perform floating-point addition? Provide an example.
18. Discuss the concept of vector performance modeling. What are the key performance metrics used to evaluate the efficiency of vector processors?
19. What are the challenges of implementing pipelining in modern processors? Discuss the impact of varying instruction execution times and data hazards.
20. Explain the concept of delayed branching in pipelined processors. How does it help in reducing branch penalties, and what are its limitations?

Analytical questions:

1. Analyze the impact of pipeline hazards on the performance of a pipelined processor. Given a sequence of instructions, identify potential data hazards and propose solutions (e.g., forwarding, stalling) to mitigate them.
2. Consider a superscalar processor with two integer units and one floating-point unit. Analyze the execution of the following instruction sequence and identify any resource conflicts or data dependencies. How would the processor handle these issues?

```
ADD R1, R2, R3
MUL R4, R1, R5
SUB R6, R7, R8
DIV R9, R10, R11
```

3. Given a vectorized code snippet for matrix multiplication, analyze the memory access patterns and identify potential bottlenecks. How would you optimize the code to improve performance on a SIMD architecture?
4. Compare and contrast the performance of a superscalar processor with a superpipelined processor for a given workload. Under what conditions would one architecture outperform the other? Provide a detailed analysis.

4. Analyze the effectiveness of branch prediction techniques (static vs. dynamic) in reducing pipeline stalls. Given a program with multiple conditional branches, which technique would you recommend, and why? Support your answer with a detailed explanation.

5. Given a pipelined processor with 5 stages (IF, ID, EX, MEM, WB), analyze the execution of the following instruction sequence. Identify any pipeline stalls or hazards and calculate the total number of clock cycles required to complete the sequence.

```
LOAD R1, [R2]
```

```
ADD R3, R1, R4
SUB R5, R6, R7
STORE [R8], R3
```

6. Consider a SIMD array processor with 16 processing elements (PEs) connected in a mesh topology. Analyze the communication overhead when performing a matrix multiplication operation. How would the communication pattern change if the PEs were connected in a hypercube topology instead?

7. Analyze the performance of a vectorized code for computing the dot product of two large vectors. Identify potential bottlenecks related to memory bandwidth and vector register usage. Propose optimizations to improve the performance of the code.

8. Given a superscalar processor capable of issuing 4 instructions per cycle, analyze the execution of the following instruction sequence. Identify any resource conflicts, data dependencies, or control hazards, and propose solutions to minimize pipeline stalls.

```
MUL R1, R2, R3
ADD R4, R1, R5
DIV R6, R7, R8
SUB R9, R10, R11
LOAD R12, [R13]
```

9. Analyze the impact of varying vector lengths on the performance of a vector processor. Given a workload with mixed vector lengths, how would you optimize the code to maximize throughput while minimizing overhead? Provide a detailed explanation.

10. If you were to design 9 stages pipelining, what will be possible stages? Explain with possible challenges and solutions to those challenges with necessary diagrams.

OLD QUESTIONS

1. Explain the use of pipelining. Differentiate between RISC and CISC pipelining with suitable diagrams.
2. Draw the block diagram of General Superscalar Processor and explain it. Differentiate between superscalar and Super pipeline with required block diagram.
3. Differentiate between RISC and CISC architecture with suitable examples.
4. Explain different types of dependencies and conflicts in the instruction level parallelism with suitable examples.

5. Explain about normal and delayed branch in the RISC machine with suitable codes and diagrams. Differentiate between registers and cache.
6. What do you mean by procedural dependencies and resource conflicts? Explain dependencies in a superscalar processor with suitable diagram. What are the process for superscalar implementation?
7. Why pipelining is required in a computer? **Differentiate between RISC and CISC pipelining with suitable diagrams**.
8. Differentiate between Superscalar and Superpipeline with required diagram. Draw the block diagram of General Superscalar Processor and explain it.
9. What do you understanding by pipelining? What is a RISC Pipelining? Explain the effect of RISC pipelining with suitable example.
10. What is a superscalar processor? Draw the block diagram of general superscalar processor and explain it. Differentiate between superscalar and superpipeline with required block diagram
11. What do you mean by RISC pipelining? Explain with suitable diagram.