Unit V

Parallel Processing

Parallel processing can be described as a class of techniques which enables the system to achieve simultaneous data-processing tasks to increase the computational speed of a computer system.

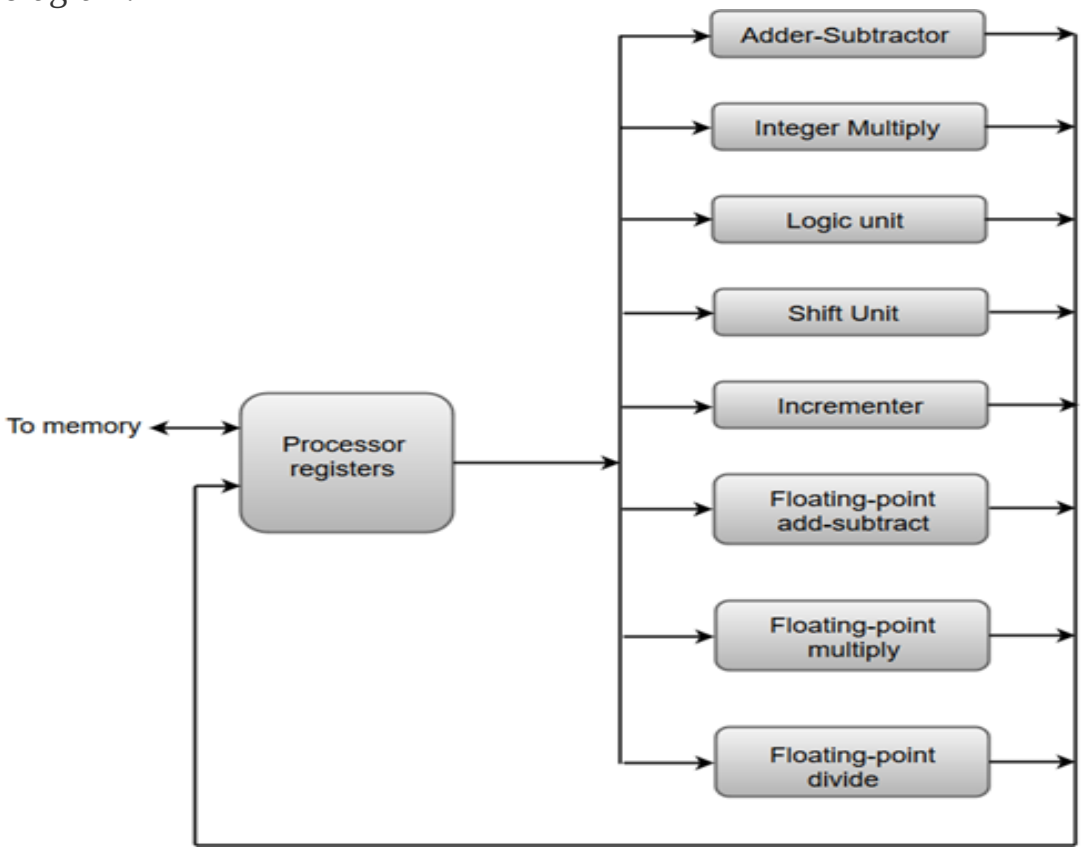
A parallel processing system can carry out simultaneous data-processing to achieve faster execution time. For instance, while an instruction is being processed in the ALU component of the CPU, the next instruction can be read from memory.

The primary purpose of parallel processing is to enhance the computer processing capability and increase its throughput, i.e. the amount of processing that can be accomplished during a given interval of time.

A parallel processing system can be achieved by having a multiplicity of functional units that perform identical or different operations simultaneously. The data can be distributed among various multiple functional units.

The following diagram shows one possible way of separating the execution unit into eight functional units operating in parallel.

The operation performed in each functional unit is indicated in each block if the diagram:



- The adder and integer multiplier perform the arithmetic operation with integer numbers.
- The floating-point operations are separated into three circuits operating in parallel.
- The logic, shift, and increment operations can be performed concurrently on different data. All units are independent of each other, so one number can be shifted while another number is being incremented.

Types of Parallel Processing

Parallel processing systems are created to speed up the implementation of programs by breaking the program into several fragments and processing these fragments together. Such systems are multiprocessor systems also referred to as tightly coupled systems. Parallel processors can be divided into the following four groups based on the number of instructions and data streams are as follows −

SISD Computer Organization

SISD represents a computer organization with a control unit, a processing unit, and a memory unit. SISD is like the serial computer in use. SISD executes instructions sequentially and they may or may not have parallel processing capabilities.

Instructions executed sequentially may get overlapped in their execution stages. A SISD computer can have greater than one functional unit in it. But all the functional units are below the administration of one control unit. Parallel processing in such systems can be attained by pipeline processing or by using multiple functional units.

SIMD Computer Organization

SIMD organization includes multiple processing elements. All these elements are below the administration of a common control unit. All processors get identical instruction from the control unit but work on multiple data items.

The shared subsystem contains multiple modules which help in communicating with all the processors simultaneously. This is further divided into word slice and bit-slice mode organizations.

MISD Computer Organization

MISD organization includes multiple processing units, each receiving separate instructions operating over a similar data flow. The result of one processor becomes the input of the next processor. The introduction of this organization received less attention and was not practically implemented in architecture. The structure was of only theoretical interest.

MIMD Computer Organization

A MIMD computer organization contains interactions among the multiprocessors since all memory flows are changed from the common data area transmitted by all processors. If the multi-data streams were derived from different shared memories then it is a multiple SISD operation that is equal to a set of 'n' independent SISD systems.

Pipelining

To improve the performance of a CPU we have two options:
1) Improve the hardware by introducing faster circuits.
2) Arrange the hardware such that more than one operation can be performed at the same time.
Since there is a limit on the speed of hardware and the cost of faster circuits is quite high, we have to adopt the 2nd option.

The term Pipelining refers to a technique of decomposing a sequential process into sub-operations, with each sub-operation being executed in a dedicated segment that operates concurrently with all other segments.

The most important characteristic of a pipeline technique is that several computations can be in progress in distinct segments at the same time. The overlapping of computation is made possible by associating a register with each segment in the pipeline. The registers provide isolation between each segment so that each can operate on distinct data simultaneously.

The structure of a pipeline organization can be represented simply by including an input register for each segment followed by a combinational circuit.

Let us consider an example of combined multiplication and addition operation to get a better understanding of the pipeline organization.

The combined multiplication and addition operation is done with a stream of numbers such as:

$A_i * B_i + C_i$ for i = 1, 2, 3, ..., 7

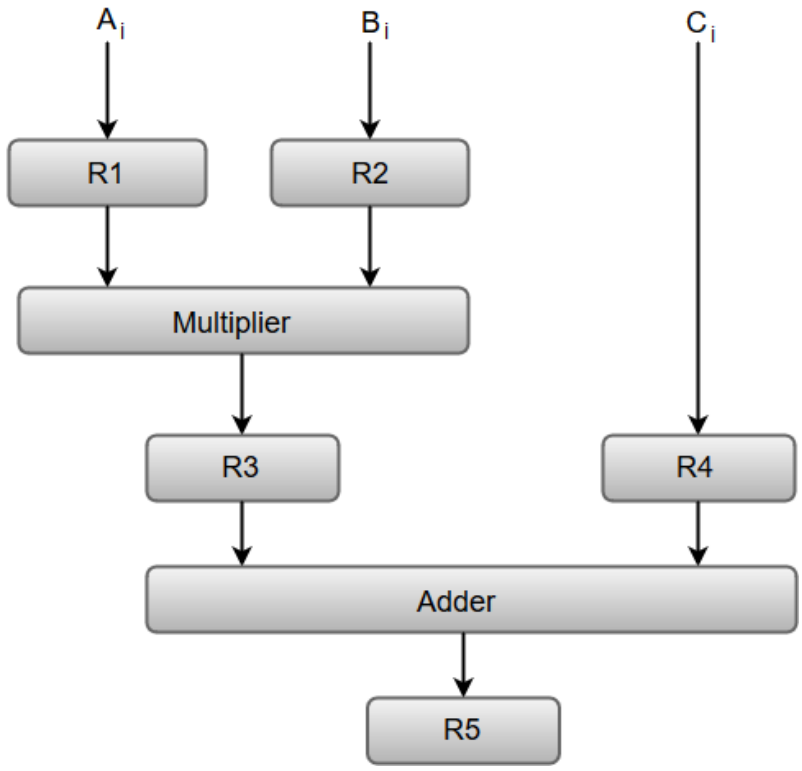
The operation to be performed on the numbers is decomposed into sub-operations with each sub-operation to be implemented in a segment within a pipeline.

The sub-operations performed in each segment of the pipeline are defined as:

$R1 \leftarrow A_i$, $R2 \leftarrow B_i$	Input A_i, and B_i
$R3 \leftarrow R1 * R2$, $R4 \leftarrow C_i$	Multiply, and input C_i
$R5 \leftarrow R3 + R4$	Add C_i to product

The following block diagram represents the combined as well as the sub-operations performed in each segment of the pipeline.

Pipeline Processing:



Registers R1, R2, R3, and R4 hold the data and the combinational circuits operate in a particular segment.

The output generated by the combinational circuit in a given segment is applied as an input register of the next segment. For instance, from the block diagram, we can see that the register R3 is used as one of the input registers for the combinational adder circuit.

In general, the pipeline organization is applicable for two areas of computer design which includes:

Arithmetic Pipeline: Arithmetic Pipelines are mostly used in high-speed computers. They are used to implement floating-point operations, multiplication of fixed-point numbers, and similar computations encountered in scientific problems.

To understand the concepts of arithmetic pipeline in a more convenient way, let us consider an example of a pipeline unit for floating-point addition and subtraction.

The inputs to the floating-point adder pipeline are two normalized floating-point binary numbers defined as:

$X = A * 2^a = 0.9504 * 10^3$

$Y = B * 2^b = 0.8200 * 10^2$

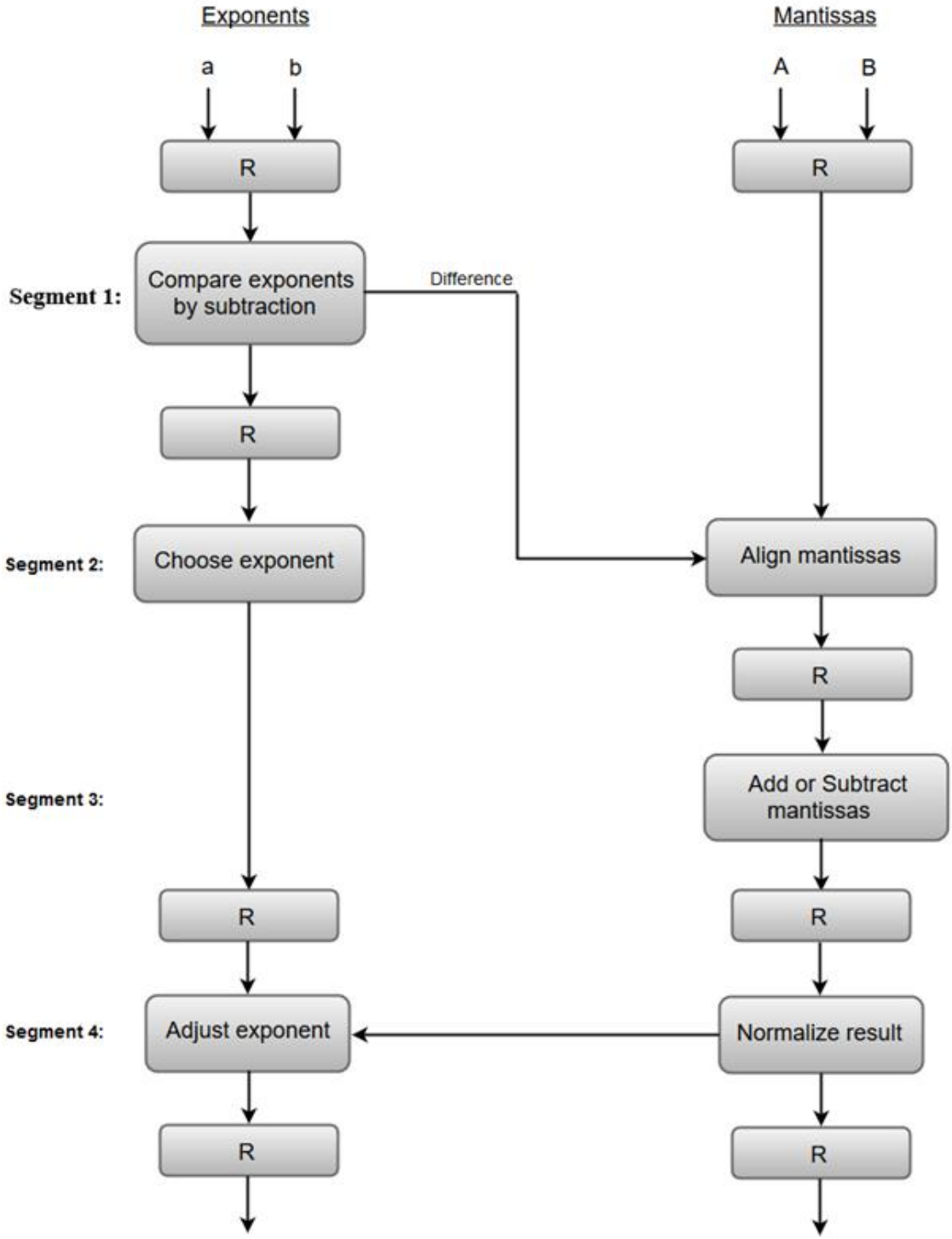
Where **A** and **B** are two fractions that represent the mantissa and **a** and **b** are the exponents.
The combined operation of floating-point addition and subtraction is divided into four segments. Each segment contains the corresponding suboperation to be performed in the given pipeline. The suboperations that are shown in the four segments are:

1. Compare the exponents by subtraction.
2. Align the mantissas.
3. Add or subtract the mantissas.
4. Normalize the result.

We will discuss each suboperation in a more detailed manner later in this section.

The following block diagram represents the suboperations performed in each segment of the pipeline.

Pipeline organization for floating point addition and subtraction:



1. Compare exponents by subtraction:

The exponents are compared by subtracting them to determine their difference. The larger exponent is chosen as the exponent of the result.

The difference of the exponents, i.e., **3 - 2 = 1** determines how many times the mantissa associated with the smaller exponent must be shifted to the right.

2. Align the mantissas:

The mantissa associated with the smaller exponent is shifted according to the difference of exponents determined in segment one.

$X = 0.9504 * 10^3$

$Y = 0.08200 * 10^3$

3. Add mantissas:

The two mantissas are added in segment three.

$Z = X + Y = 1.0324 * 10^3$

4. Normalize the result:

After normalization, the result is written as:

$Z = 0.1324 * 10^4$

Instruction Pipeline: Pipeline processing can occur not only in the data stream but in the instruction stream as well.

Most of the digital computers with complex instructions require instruction pipeline to carry out operations like fetch, decode and execute instructions.

In general, the computer needs to process each instruction with the following sequence of steps.

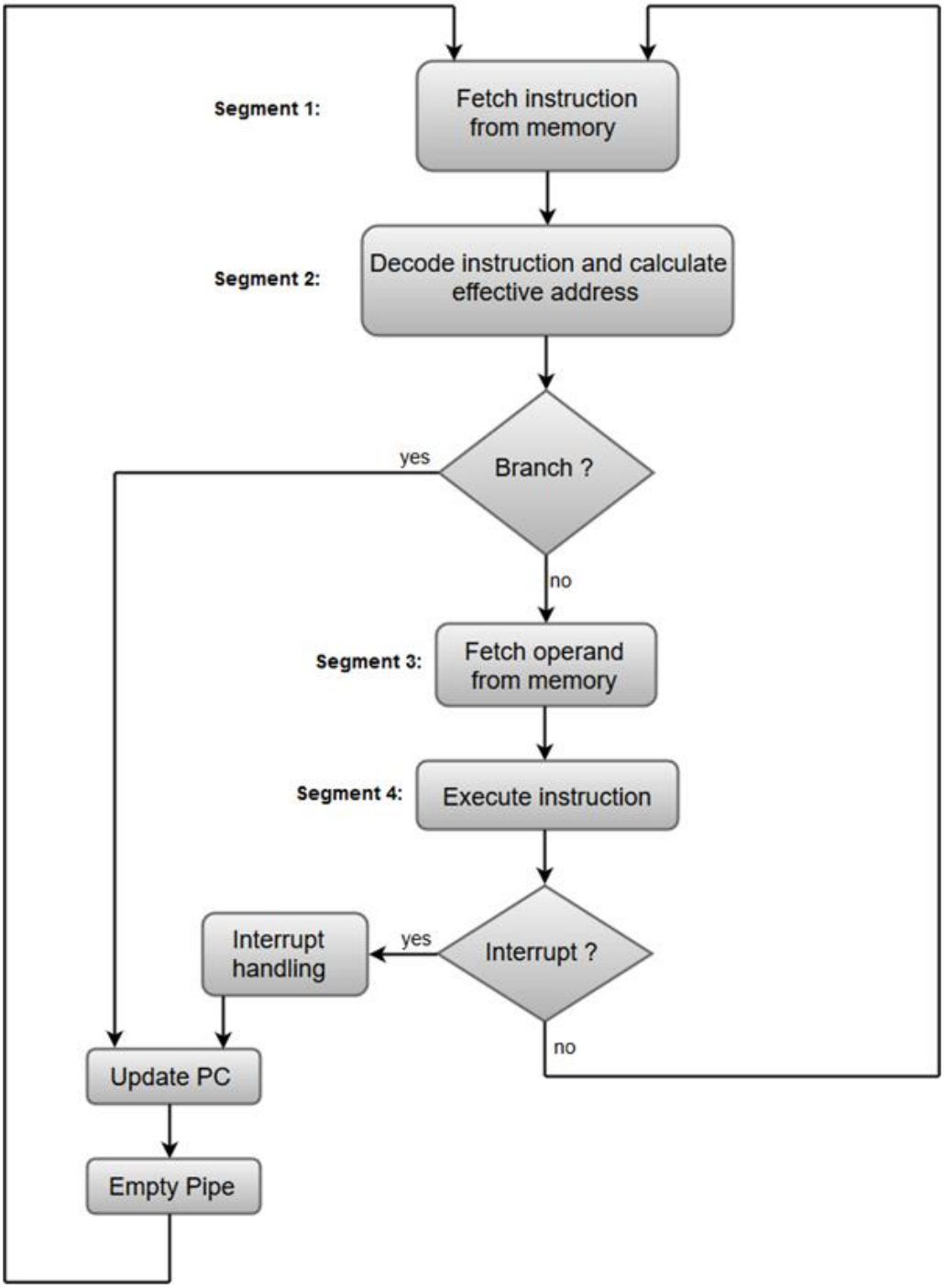
1. Fetch instruction from memory.
2. Decode the instruction.
3. Calculate the effective address.
4. Fetch the operands from memory.
5. Execute the instruction.
6. Store the result in the proper place.

Each step is executed in a particular segment, and there are times when different segments may take different times to operate on the incoming information. Moreover, there are times when two or more segments may require memory access at the same time, causing one segment to wait until another is finished with the memory.

The organization of an instruction pipeline will be more efficient if the instruction cycle is divided into segments of equal duration. One of the most common examples of this type of organization is a **Four-segment instruction pipeline.**

A **four-segment instruction** pipeline combines two or more different segments and makes it as a single one. For instance, the decoding of the instruction can be combined with the calculation of the effective address into one segment.

The following block diagram shows a typical example of a four-segment instruction pipeline. The instruction cycle is completed in four segments.



Segment 1:

The instruction fetch segment can be implemented using first in, first out (FIFO) buffer.

Segment 2:

The instruction fetched from memory is decoded in the second segment, and eventually, the effective address is calculated in a separate arithmetic circuit.

Segment 3:

An operand from memory is fetched in the third segment.

Segment 4:

The instructions are finally executed in the last segment of the pipeline organization.

Vector Processing

Vector processing is a computer method that can process numerous data components at once. It operates on every element of the entire vector in one operation, or in parallel, to avoid the overhead of the processing loop. Yet simultaneous operations must be independent of one another in order for vector processing to be effective.

Vector processing vs. array and parallel processing

Now, let's understand the important difference between array processing and vector processing. Arrays are groups of data elements that are kept in close proximity to one another in memory. They're frequently used to symbolize parallel-processable datasets, while the term "vector processing" describes the simultaneous processing of many data units using specialized technology. The distinction between array processing and vector processing is that while vector processing uses a single processor to execute the same operation on numerous data items concurrently, array processing uses several processors to work on individual array elements.

The difference between parallel processing and vector processing is that parallel processing involves multiple processors working on separate tasks simultaneously. In contrast, vector processing involves a single processor performing the same operation on multiple data elements simultaneously.

Amdahl's law

Amdahl's law states that when a part of a system is improved, the overall system improvement will be proportional to how much that part makes up of the system. Components that make up a larger proportion of the system should be the focus of

improvements. Also, the improvement of a system is limited by parts that cannot be improved.

What is Amdahl's law and why it is used?

Amdahl's law is used to calculate the system improvement expected when parts of the system are improved. The "system" can be hardware or software. It can be used to determine which components of a system would be best to focus on improving.

Although it was presented back in 1967, Amdahl's Law remains a crucial basis for numerous computing principles and technologies and continues to be utilized widely to this day.

Amdahl's Law is a formula that predicts the potential speed increase in completing a task with improved system resources while keeping the workload constant. The theoretical speedup is always limited by the part of the task that cannot benefit from the improvement.

Amdahl's Law applies only to cases where the problem size, or workload, is fixed.

Formula for Amdahl's Law

The base formula for Amdahl's Law is $S = 1 / (1 - p + p/s)$, where…

- S is the speedup of a process
- p is the proportion of a program that can be made parallel, and
- s is the speedup factor of the parallel portion.

This formula states that the maximum improvement in speed of a process is limited by the proportion of the program that can be made parallel.

In other words, it does not matter how many processors you have or how much faster each processor may be; the maximum improvement in speed will always be limited by the most significant bottleneck in a system.

Examples of Amdahl's Law

Now let’s look at a couple examples to bring this law to life.

Example 1: Teammate collaboration

Your company has scheduled a meeting with 3 team members first thing in the morning. Each team member needs to get themselves to the office, but they can't start the meeting until everyone has arrived.
Two of the sales members drive themselves into the office, while the third teammate rides their bicycle.
According to Amdahl’s Law, if the team wants to start their meetings earlier, then they need to focus on the performance of the cyclist. Even if one of the drivers drives faster than usual, there's still no way to get around the fact that they have to wait for the bicyclist who is the bottleneck in this situation.

Example 2: Website performance

An online store has just released a new product and they are expecting a huge surge in web traffic. To handle the increased demand, they have decided to add more servers to their system. While additional servers can help them handle the increased traffic, they are limited by how quickly each page is served — that will be the bottleneck of the system.
According to Amdahl's Law, they need to focus on optimizing their webpages rather than simply adding more servers to help with the surge in traffic.

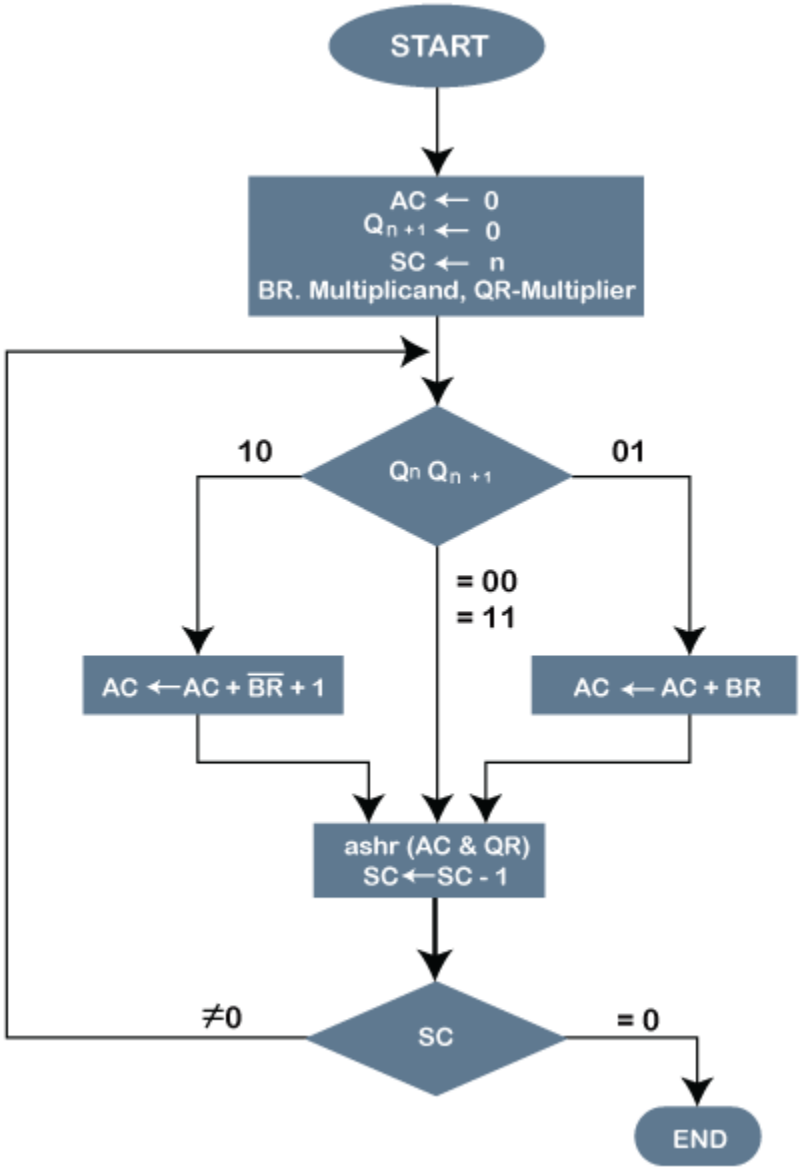
Disadvantages of Amdahl's Law

One of the main disadvantages of Amdahl's Law is that it cannot be applied to situations where a problem size or workload can increase. This is because the formula for Amdahl's Law assumes that the workload and problem size are fixed. Therefore, any improvements in system performance must be due to optimizing components of the system which may be limited in scope.
In addition, Amdahl's Law does not account for any unexpected bottlenecks that may arise during the optimization process, such as our cyclist experiencing a flat tire on their way to the meeting. This means there is no guarantee that the performance improvements achieved will be optimal or even significant enough to justify the effort and resources put into optimizing a system.

Booth multiplication algorithm

The booth algorithm is a multiplication algorithm that allows us to multiply the two signed binary integers in 2's complement, respectively. It is also used to speed up the performance of the multiplication process. It is very efficient too. It works on the string bits 0's in the multiplier that requires no additional bit only shift the right-most string bits and a string of 1's in a multiplier bit weight 2^k to weight 2^m that can be considered as $\mathbf{2^{k+1} - 2^m}$.

Following is the pictorial representation of the Booth's Algorithm:



In the above flowchart, initially, **AC** and $\mathbf{Q_{n+1}}$ bits are set to 0, and the **SC** is a sequence counter that represents the total bits set **n,** which is equal to the number of bits in the multiplier. There are **BR** that represent the **multiplicand bits,** and QR represents the **multiplier bits**. After that, we encountered two bits of the multiplier as Q_n and Q_{n+1}, where Qn represents the last bit of QR, and Q_{n+1} represents the incremented bit of Qn by 1. Suppose two bits of the multiplier is equal to 10; it means that we have to subtract the multiplier from the partial product in the accumulator AC and then perform the arithmetic shift operation (ashr). If the two of the multipliers equal to 01, it means we need to perform the addition of the multiplicand to the partial product in accumulator AC and then perform the arithmetic shift operation (**ashr**), including $\mathbf{Q_{n+1}}$. The arithmetic shift operation is used in Booth's algorithm to shift AC and QR bits to the right by one and remains the sign bit in AC unchanged. And the sequence counter is continuously decremented till the computational loop is repeated, equal to the number of bits (n).

Working on the Booth Algorithm

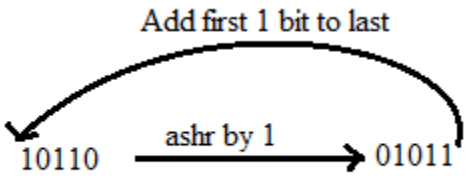
1. Set the Multiplicand and Multiplier binary bits as M and Q, respectively.
2. Initially, we set the AC and Q_{n+1} registers value to 0.
3. SC represents the number of Multiplier bits (Q), and it is a sequence counter that is continuously decremented till equal to the number of bits (n) or reached to 0.
4. A Qn represents the last bit of the Q, and the Q_{n+1} shows the incremented bit of Qn by 1.
5. On each cycle of the booth algorithm, Q_n and Q_{n+1} bits will be checked on the following parameters as follows:
    i. When two bits Q_n and Q_{n+1} are 00 or 11, we simply perform the arithmetic shift right operation (ashr) to the partial product AC. And the bits of Qn and Q_{n+1} is incremented by 1 bit.
    ii. If the bits of Q_n and Q_{n+1} is shows to 01, the multiplicand bits (M) will be added to the AC (Accumulator register). After that, we perform the right shift operation to the AC and QR bits by 1.
    iii. If the bits of Q_n and Q_{n+1} is shows to 10, the multiplicand bits (M) will be subtracted from the AC (Accumulator register). After that, we perform the right shift operation to the AC and QR bits by 1.

6. The operation continuously works till we reached n - 1 bit in the booth algorithm.
7. Results of the Multiplication binary bits will be stored in the AC and QR registers.

There are two methods used in Booth's Algorithm:

1. RSC (Right Shift Circular)

It shifts the right-most bit of the binary number, and then it is added to the beginning of the binary bits.



2. RSA (Right Shift Arithmetic)

It adds the two binary bits and then shift the result to the right by 1-bit position.

Example: 0100 + 0110 => 1010, after adding the binary number shift each bit by 1 to the right and put the first bit of resultant to the beginning of the new bit.

Example: Multiply the two numbers 7 and 3 by using the Booth's multiplication algorithm.

Ans. Here we have two numbers, 7 and 3. First of all, we need to convert 7 and 3 into binary numbers like 7 = (0111) and 3 = (0011). Now set 7 (in binary 0111) as multiplicand (M) and 3 (in binary 0011) as a multiplier (Q). And SC (Sequence Count) represents the number of bits, and here we have 4 bits, so set the SC = 4. Also, it shows the number of iteration cycles of the booth's algorithms and then cycles run SC = SC - 1 time.

Q_n	Q_{n+1}	**M = (0111) M' + 1 = (1001) & Operation**	**AC**	**Q**	Q_{n+1}	**SC**
1	0	Initial	0000	0011	0	4

		Subtract (M' + 1)	1001			
			1001			
		Perform Arithmetic Right Shift operations (ashr)	1100	1001	1	3
1	1	Perform Arithmetic Right Shift operations (ashr)	1110	0100	1	2
0	**1**	Addition (A + M)	0111			
			0101	0100		
		Perform Arithmetic right shift operation	0010	1010	0	1
0	0	Perform Arithmetic right shift operation	**0001**	**0101**	0	0

The numerical example of the Booth's Multiplication Algorithm is 7 x 3 = 21 and the binary representation of 21 is 10101. Here, we get the resultant in binary 00010101. Now we convert it into decimal, as $(000010101)_{10}$ = 2*4 + 2*3 + 2*2 + 2*1 + 2*0 => 21.

Example: Multiply the two numbers 23 and -9 by using the Booth's multiplication algorithm.

Here, M = 23 = (010111) and Q = -9 = (110111)

Q_{n+1}	**Q_n**	**M = 0 1 0 1 1 1 M' + 1 = 1 0 1 0 0 1**	**AC**	**Q**	**Q_{n+1}**	**SC**
		Initially	000000	110111	0	6
1	0	Subtract M	101001			
			101001			
		Perform Arithmetic right shift operation	110100	111011	1	5

1	1	Perform Arithmetic right shift operation	111010	011101	1	4
1	1	Perform Arithmetic right shift operation	111101	001110	1	3
0	1	Addition (A + M)	010111			
			010100			
		Perform Arithmetic right shift operation	001010	000111	0	2
1	0	Subtract M	101001			
			110011			
		Perform Arithmetic right shift operation	111001	100011	1	1
1	1	Perform Arithmetic right shift operation	**111100**	**110001**	**1**	0

$\mathbf{Q_{n+1}}$ = 1, it means the output is negative.

Hence, 23 * -9 = 2's complement of 111100110001 => **(00001100111)**

Floating-point representation and arithmetic

To convert the floating point into decimal, we have 3 elements in a 32-bit floating point representation:

i) Sign
ii) Exponent
iii) Mantissa

- **Sign** bit is the first bit of the binary representation. '1' implies negative number and '0' implies positive number.
  Example: 11000001110100000000000000000000 This is negative number.
- **Exponent** is decided by the next 8 bits of binary representation. 127 is the unique number for 32 bit floating point representation. It is known as bias. It is determined by 2^{k-1} -1 where 'k' is the number of bits in exponent field.

There are 3 exponent bits in 8-bit representation and 8 exponent bits in 32-bit representation.

Thus
bias = 3 for 8 bit conversion (2^{3-1} -1 = 4-1 = 3)
bias = 127 for 32 bit conversion. (2^{8-1} -1 = 128-1 = 127)

Example: 01000001110100000000000000000000
10000011 = $(131)_{10}$
131-127 = 4

Hence the exponent of 2 will be 4 i.e. 2^4 = 16.

- **Mantissa** is calculated from the remaining 23 bits of the binary representation. It consists of '1' and a fractional part which is determined by:

  Example:
  01000001110100000000000000000000

  The fractional part of mantissa is given by:
  1*(1/2) + 0*(1/4) + 1*(1/8) + 0*(1/16) +……. = 0.625

  Thus the mantissa will be 1 + 0.625 = 1.625

  The decimal number hence given as: Sign*Exponent*Mantissa = (-1)^0*(16)*(1.625) = 26

2. To convert the decimal into floating point, we have 3 elements in a 32-bit floating point representation:

i) Sign (MSB)
ii) Exponent (8 bits after MSB)
iii) Mantissa (Remaining 23 bits)

- **Sign bit** is the first bit of the binary representation. '1' implies negative number and '0' implies positive number.
  Example: To convert -17 into 32-bit floating point representation Sign bit = 1
- **Exponent** is decided by the nearest smaller or equal to 2^n number. For 17, 16 is the nearest 2^n. Hence the exponent of 2 will be 4 since 2^4 = 16. 127 is the unique number for 32 bit floating point representation. It is known as bias. It is determined by 2^{k-1} -1 where 'k' is the number of bits in exponent field.

Thus bias = 127 for 32 bit. (2^{8-1} -1 = 128-1 = 127)
Now, 127 + 4 = 131 i.e. 10000011 in binary representation.

- **Mantissa:** 17 in binary = 10001.
  Move the binary point so that there is only one bit from the left. Adjust the exponent of 2 so that the value does not change. This is normalizing the number. 1.0001 x 2^4. Now, consider the fractional part and represented as 23 bits by adding zeros.

  00010000000000000000000

Advantages:

Wide range of values: Floating factor illustration lets in for a extensive variety of values to be represented, along with very massive and really small numbers.
Precision: Floating factor illustration offers excessive precision, that is important for medical and engineering calculations.
Compatibility: Floating point illustration is extensively used in computer structures, making it well matched with an extensive variety of software and hardware.
Easy to use: Most programming languages offer integrated guide for floating factor illustration, making it smooth to use and control in laptop programs.

Disadvantages:

Complexity: Floating factor illustration is complex and can be tough to understand, mainly for folks that aren't acquainted with the underlying mathematics.
Rounding errors: Floating factor illustration can result in rounding mistakes, where the real price of a number of is barely extraordinary from its illustration inside the computer.
Speed: Floating factor operations can be slower than integer operations, particularly on older or much less powerful hardware.
Limited precision: Despite its excessive precision, floating factor representation has a restrained number of sizeable digits, which could restrict its usefulness in some programs.