Improving Computer Performance

How can we make computers faster ?

The Fetch-Execute Cycle and Pipelining

The fetch-execute cycle represents the fundamental process in the operation of the CPU, attention has been focused on ways of making it more efficient.

One possibility is to improve the speed at which instructions and data may be retrieved from memory, since the CPU can process information at a faster rate than it can retrieve it from memory.

The use of a **cache** memory system, which is discussed later, can improve matters in this respect.

Another way of improving the efficiency of the fetch execute cycle is to use a system known as **pipelining**.

The basic idea here is to break the fetch-execute cycle into a number of separate stages, so that when one stage is being carried out for a particular instruction, the CPU can carry out another stage for a second instruction, and so on. This idea originates from the **assembly line** concept used in manufacturing industry.

Consider a simplified car manufacturer's assembly line as shown in Figure 1. The production of a car involves a number of stages, three of which are illustrated. So, for example, if a car goes through 10 stages before being completed, then we can have up to ten cars being operated on at the same time on the assembly line. If, for the sake of simplicity, we assume that each stage takes one hour to complete, then it will take 10 hours to complete the first car since it will be processed for 1 hour at every stage on the assembly line.

However, when the first car has moved to the second stage of the assembly line, we can start work on a second car at the first stage of the

assembly line. When the first car moves on to the third stage, the second car can move on to the second stage and a third car can be started on the first stage of the assembly line.

This process continues, so that when the first car reaches the 10th and final stage, there are 9 other cars in the first nine stages of production. This means that when the first car is finished after 10 hours, then another car will be completed every hour thereafter.

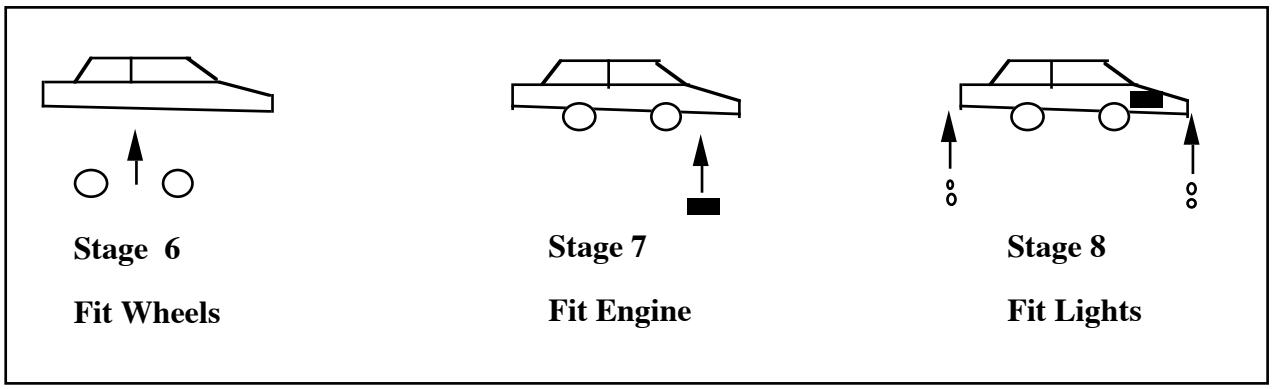

Figure 1: Assembly line production

The great advantage of assembly line production is the increase in **throughput** that is achieved. After the first car is completed we continue production with a throughput of one car per hour.

If we did not use an assembly line and worked on one car at a time, each car would take 10 hours to produce and the throughput would be one car per 10 hours.

For example, the time taken to complete 20 cars on the assembly line is 29 hours, while without using the assembly line, the time taken would be 200 hours. It should be noted that on the assembly line, each car still requires 10 hours of processing, i.e. it still takes 10 hours of work to produce a car, the point is that because we are doing the work in stages, we can work on 10 cars at the same time, i.e. in parallel.

Flowthrough Time

The time taken for all stages of the assembly line to become active is called the **flowthrough time**, i.e. the time for the first car to reach the

last stage. Once all the assembly line stages are busy, we achieve maximum throughput.

We have simplified the analysis of the assembly line and in particular the assumption that all stages take the same amount of time is not likely to be true. The stage that takes the longest time to complete creates a bottleneck in an assembly line.

For example, if we assume that stage 5 in our car assembly line takes 3 hours then the throughput decreases to 1 car per 3 hours. This is because stage 6 must wait for 3 hours before it can begin and this delay is passed on to the remaining stages, slowing the time to complete each car to 3 hours.

Clock period

We can express this by saying the **clock period** of the assembly line (time between completed cars) is 3 hours. The clock period, denoted by $\mathbf{T_p}$, of an assembly line is given by the formula:

$$\mathbf{T_p = max(t_1, t_2, t_3, ..., t_n)}$$

where $\mathbf{t_i}$ is the time taken for the ith stage and there are n stages in the assembly line. This means that the clock period is determined by the time taken by the stage that requires the most processing time.

In a non-assembly line system, the total time **T**, taken to complete a car, is the sum of the time for the individual stages, i.e.

$$\mathbf{T = t_1 + t_2 + t_3 + ... + t_n}$$

In our example, if all stages take 1 hour to complete, then T = 10 hours, it takes 10 hours to complete every car.

If stage 5 takes 3 hours and the other stages take 1 hour to complete then T rises to 12 hours and it will take 12 hours to complete every car.

Throughput

We can define the **throughput** of an assembly line to be $\mathbf{1/T_p}$.

Using this definition, the throughput for our assembly line where all stages take 1 hour is 1/1, i.e. 1 car/hour. If we assume stage 5 takes 3 hours to complete, the throughput falls to 1/3 or .333 cars/hour. For non-assembly line production the respective throughputs are 1/10 or .1 cars/hour and 1/12 or .083 cars/hour.

Pipelining

The same principle as that of the assembly line can be applied to the fetch-execute cycle of a processor where we refer to it as **pipelining**.

Earlier we described the fetch-execute cycle as consisting of 3 stages, which are repeated continuously:

1. Fetch an instruction
2. Decode the instruction
3. Execute the instruction

Assuming each stage takes one clock cycle, then in a non-pipelined system, we use 3 cycles for the first instruction, followed by 3 cycles for the second instruction and so on, as illustrated in Figure 2:

	cycle 1	**cycle 2**	**cycle 3**	**cycle 4**	**cycle 5**	**cycle 6**
Instruction 1	**Fetch**	**Decode**	**Execute**			
Instruction 2				**Fetch**	**Decode**	**Execute**

Figure 2: Fetch-execute cycle

The throughput for such a system would be 1 instruction per 3 cycles. If we adopt the assembly line principle, then we can improve the throughput dramatically.

Figure 3 illustrates the fetch-execute cycle employing pipelining.

	cycle 1	cycle 2	cycle 3	cycle 4	cycle 5	cycle 6
Instruction 1	**Fetch**	**Decode**	**Execute**			
Instruction 2		**Fetch**	**Decode**	**Execute**		
Instruction 3			**Fetch**	**Decode**	**Execute**	
Instruction 4				**Fetch**	**Decode**	**Execute**

Figure 7.3: Fetch-execute cycle with pipelining

Using pipelining, we overlap the processing of instructions, so that while the first instruction is in the decode stage, the second instruction is being fetched. While the first instruction is in the execute stage, the second instruction is in the decode stage and the third instruction is being fetched.

After 3 cycles the first instruction is completed and thereafter an instruction is completed on every cycle as opposed to a throughput of 3 cycles per instruction in a non-pipelined system.

Again, as in the assembly line example, each instruction still takes the same number of cycles to complete, the gain comes from the fact that the CPU can operate on instructions in the different stages in **parallel**. The clock period and throughput of a pipeline are as defined for the assembly line above:

Clock period T_p = **$\max(t_1, t_2, t_3, ..., t_n)$**
(for n stage pipeline)

Throughput = **$1/T_p$**

The above description is quite simplified, ignoring the fact for example that all stages may not be completed in a single cycle. It also omits stages that arise in practice such as an **operand fetch** stage, which is required to fetch an operand from memory, or a **write back** stage to store the result of an ALU operation in a register or in memory. In

practice, pipelined systems range from having 3 to 10 stages, for example, Intel's Pentium microprocessor uses a 5-stage pipeline for integer instructions.

There are difficulties in pipelining that would not arise on a factory assembly line, due to the nature of computer programs. Consider the following 3 instructions in a pipeline:

```
            jg          label
            move   y, 0
            move   x, 3
              ......
              ......
label:
```

When the `jg` instruction is being executed, the following two instructions will be in earlier stages, one being fetched and the other being decoded. However, if the `jg` instruction evaluates the condition to be true, it means that the two `move` instructions will not be executed and new instructions have to be loaded, starting at the instruction indicated by `label`.

This means that we have to **flush** the pipeline and reload it with new instructions. The time taken to reload the pipeline is called the **branch penalty** and may take several clock cycles. Branch instructions occur very frequently in programs and so it is important to process them as efficiently as possible.

A technique known as **branch prediction** can be used to alleviate the problem of conditional branch instructions, whereby the system "guesses" the outcome of a conditional branch evaluation before the instruction is evaluated and loads the pipeline appropriately. Depending on how successfully the guess is made, the need for flushing the pipeline can be reduced. When the branch has been evaluated, the processor can take appropriate action if a wrong guess was made. In the event of an incorrect guess, the pipeline will have to be flushed and new instructions loaded. Branch prediction is used on a number of microprocessors such as the Pentium and PowerPC. Successful guesses

ranging from 80% to 85% of the time are cited for the Pentium microprocessor.

Another technique is to use **delayed branching**. In this case, the instruction following the conditional jump instruction is always executed. For example, if the conditional jump instruction is implementing a loop by jumping backwards, it may be possible to place one of the loop body instructions after the conditional jump instruction. If a useful instruction cannot be placed here, then a `nop` instruction can be used.

Increasing Execution Speed: More Hardware

Consider the following two instructions:

```
add  i, 10
add  x, y
```

In a simple processor these instructions would be executed in sequence by transferring the operands to the ALU and carrying out the addition operations.

One way of speeding things up is to have to two ALUs so that the instructions can be carried out at the same time, i.e. the two ALUs can carry out the instructions in parallel. This idea is now widely employed by the current generation of microprocessors such as the Pentium, PowerPC and Alpha.

The term **superscalar** is used to describe an architecture with two or more **functional units** which can carry out two or more instructions in the same clock cycle.

These may include **integer units** (IUs), **floating-point units** (**FPUs**) and **branch processing units** (BPUs, devoted to handling branch instructions).

The **micro-architecture** of a hypothetical superscalar processor is illustrated in Figure 4. The term micro-architecture is used to refer to the internal architecture of a processor.

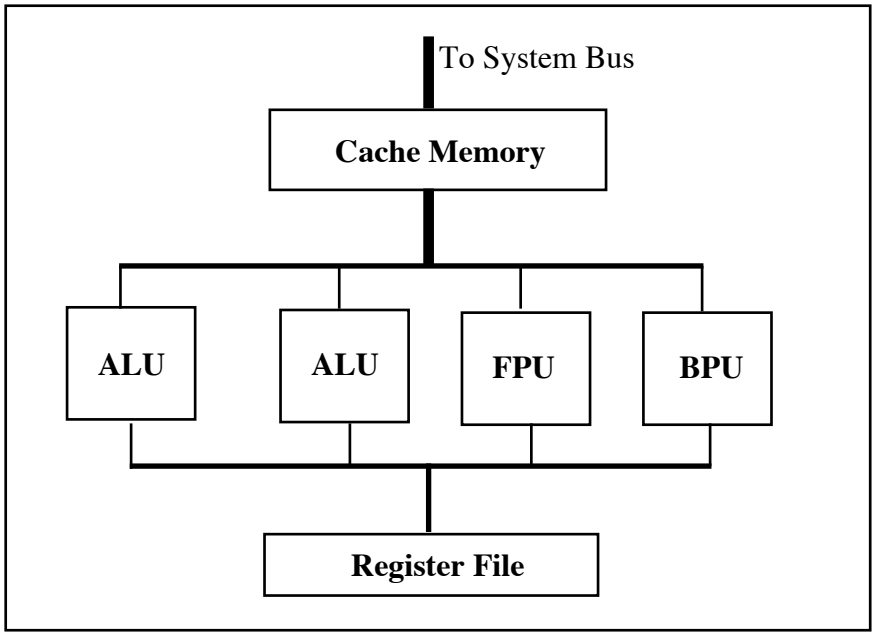

Figure 4: Micro-architecture of a superscalar processor

The processor shown in Figure 4 has four execution units and could execute four instructions concurrently, **in theory**.

The register file is the collective name given to the CPU's registers. A processor with an **on-chip** FPU would have a separate set of floating-point registers (**FPR**s) in addition to the usual general purpose registers (**GPR**s).

The cache memory is used to speed up the processor's access to instructions and data (see later). The extra functional units allow the CPU carry out more operations per clock cycle. Figure 5 illustrates the fetch-execute cycle for a superscalar architecture, with **two** functional units operating in parallel.

	cycle 1	cycle 2	cycle 3	cycle 4	cycle 5	cycle 6
Instruction 1	**Fetch**	**Decode**	**Execute**			
Instruction 2	**Fetch**	**Decode**	**Execute**			
Instruction 3		**Fetch**	**Decode**	**Execute**		
Instruction 4		**Fetch**	**Decode**	**Execute**		
Instruction 5			**Fetch**	**Decode**	**Execute**	
Instruction 6			**Fetch**	**Decode**	**Execute**	
Instruction 7				**Fetch**	**Decode**	**Execute**
Instruction 8				**Fetch**	**Decode**	**Execute**

Figure 5: Superscalar architecture fetch-execute cycle

As can be seen from Figure 5, two instructions are in each stage of the pipeline at the same time, doubling the throughput of the pipeline, **in theory**.

If we consider cycle 3 we can see that the processor is handling six stages of six instructions at the same time.

In practice, this can give rise to a number of problems, since there are situations that arise, which mean that instructions cannot be executed in parallel. One problem that arises is that of **inter-instruction dependencies**. Consider the following three instructions:

```
sub  r1, x
add  r1, 2
sub  r0, r3
```

The first two instructions (`add` and `sub`) cannot be carried out in parallel as they both modify the same operand, `r1`, giving rise to a **data dependency**.

A data dependency arises between two instructions if the destination operand of one instruction is accessed by the other instruction.

In this particular example, a solution is possible called **instruction reordering** (**scheduling**) because by reordering the instructions in the sequence:

```
sub  r1, x
sub  r0, r3
add  r1, 2
```

the two `sub` instructions can now be executed in parallel, since their operands do not conflict with each other. This is a software solution and can be implemented by a compiler, when translating a program to machine code. The compiler software must be aware of the processor's superscalar architecture for this solution to be implemented.

This is why **processor aware compilers** and **native compilers**[1] are very important, if programs are to take advantage of the advanced features of a processor. For example, a compiler that only generates code for an Intel 80386 can also be used on a PC with Intel's Pentium processor. However, the code produced will not perform nearly as well, as code produced by a native Pentium compiler.

The number of functional units for a superscalar architecture typically varies from 3 to 5 while the number of instructions that a machine may issue in a cycle typically varies from 1 to 3.

An important implication of the superscalar fetch-execute pipeline as illustrated in Figure 5, is that since two instructions are fetched at the same time, the CPU bus must be wide enough to transfer the two instructions to the control unit. It is important to point out, that pipelining and superscalar designs require rapid access to data and instructions if they are to be successful in improving performance. The provision of on-chip cache memory, as described later, facilitates such rapid access.

Parallel Computers

[1] A **processor aware** compiler is capable producing efficient code for more than processor such as an Intel 80486 and a Pentium. A **native** compiler is designed to generate efficient code for a specific processor.

Another very important technique for increasing a computers performance is the use of more than one processor in a computer system, as for example, in a **multiprocessor** or **parallel** computer. Such machines may have from 2 to many thousands of interconnected processors. Each processor may have its own private memory or they may share a common memory.

One of the difficulties with such machines is the development of software to take advantage of their parallel nature. It is important to note that such machines will only yield significant performance gains if the problems they are being used to handle can be expressed in a parallel form. The manipulation of matrices is one such problem.

For example, given two 10,000 element matrices which have to be summed to produce a third matrix, and a parallel machine with 10,000 processors, one processor can be dedicated to the addition of each pair of elements.

Thus, in crude terms, the computation can be carried out in the time taken for the addition of one pair of elements, since the 10,000 processors can carry out the operation in parallel. The same operation on a conventional processor would require the time taken for all 10,000 additions.

The performance gain is striking but it must be stressed that this example is precisely suited to a parallel machine. A major design issue in the construction of parallel computers is how the processors communicate with each other and memory. Various solutions are possible such as **crossbar** connections and **hypercube** connections.

Increasing Execution Speed: Faster Clock

The clock speed of a computer determines the rate at which the CPU operates.

It is measured in megahertz (MHz) or millions of cycles per second and GigaHertz (GHz).

1 GHz = 1000Mhz

Early microcomputers had a clock speed in the low MHz range, e.g. 1 to 4 MHz. With advancing chip technology, higher and higher clock speed have been obtained.

Standard personal computers currently run at typical speeds in the **1500MHz to 3000MHz i.e 1.5 GHz to 3 GHz**

These speeds seemed impossible for a microprocessor only a few years ago.

To gain some insight into clock speed, consider a **1000MHz clock rate**.

At 1000MHz, each clock cycle takes

one thousandth of a millionth of a second,
or
0.001 microseconds or 1 nanosecond.

Light travels at about 1 foot per nanosecond,

so **one clock cycle of a 1000MHz clock**
takes the same amount of time as
the time light takes to travel 1 foot!

The von Neumann bottleneck

It should be noted that increasing the clock speed does **not** guarantee significant performance gains. This is because the speed of the processor is effectively determined by **the rate at which it can fetch instructions and data from memory.**

Thus if the processor spends 90% of its time waiting on memory, the performance gained by doubling the processor speed (without improving the memory access time) is only 5%.

For example, assume a task takes 100 units of time, and 90 units are spent waiting on memory access with 10 units spent on CPU processing.

By doubling the CPU speed, CPU processing time is reduced to 5 units and so the overall time is reduced to 95 time units, i.e. a 5% improvement. It is obviously important then to reduce the time the CPU has to wait for memory accesses.

This is known as the **von Neumann bottleneck** - caused by the mismatch in speed between the CPU and memory.

The CPU can process data at a low nanosecond rate while RAM can only deliver it at a high nanosecond rate. For example, if RAM delivers data to the CPU at a rate of 100ns per data item (10 million items per second!) and the CPU can consume data at say 5ns per item, then the CPU will still spend 95% of its time waiting on memory.

I/O Delays

The processor will also usually have to wait for I/O operations to complete and indeed it is usually the case that I/O speeds determine the speed of program execution. Recall that it is of the order of 100,000 times slower to retrieve data from disk than it is to retrieve it from memory. This means that for programs that carry out I/O, the processor is idle most of the time, waiting for the I/O operations to complete. This in turn, means that using a more powerful processor to execute such programs, results in very little gain in overall execution speed.

Improving Memory Access Time: Cache memory
One way of improving memory access time involves the use of a **cache memory system**.

The processor operates at its maximum speed if the data to be processed is in its registers. Unfortunately, register storage capacity is very limited and so memory is used to store programs and data.

One, very effective way of overcoming the slow access time of main memory, is to design a faster intermediate memory system, that lies between the CPU and main memory.

Such memory is called **cache memory** (or simply **cache**) and it may be visualised as in Figure 8.

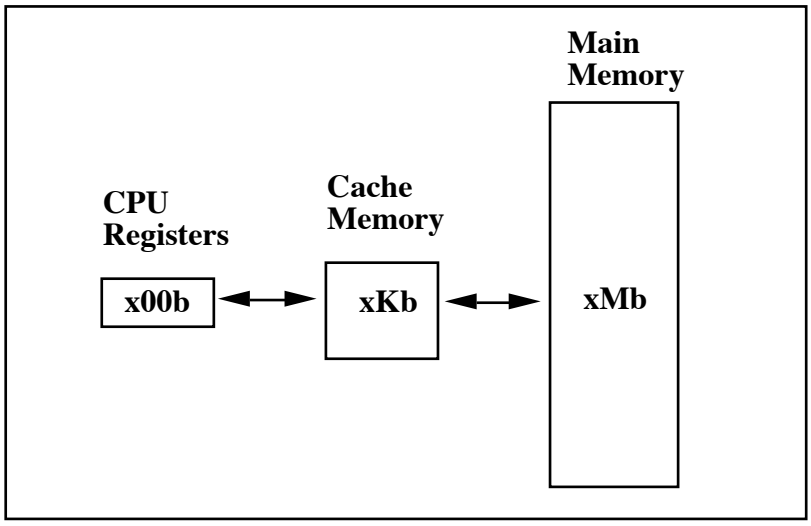

Figure 8: Cache Memory: Early systems had small cache memory systems measured in kilobytes, compared to today's megabyte cache memory systems

Cache memory is high speed memory (e.g. SRAM) which can be accessed much more quickly than normal memory (usually dynamic RAM (DRAM)). It has a **smaller capacity** than main memory and it holds recently accessed data from main memory.

The cache memory system is usually not visible to the programmer

The reason why cache memory works so well in improving performance is due to what is known as the **principle of locality of reference**. This roughly means that having accessed a particular location in memory, it is highly likely that you will access neighbouring memory locations subsequently. This is because:

- programs tend to execute instructions sequentially and instructions are stored in neighbouring memory locations
- programs often have loops whereby a group of neighbouring instructions are repeatedly executed
- arrays of data elements get accessed sequentially

As a result, when an instruction or data element is fetched from memory, if you also fetch its neighbouring instructions or data elements and store them in cache memory, then it is very likely that the next item to be fetched will be in cache memory and can be obtained very quickly, relative to accessing it in main memory.

Cache memory operates so that when the CPU initiates a memory access (for data or an instruction), the cache memory is first checked[2] to see if the information is already there.

If it is there (called a **cache hit**), it can be transferred to the CPU very quickly. If the information is not in cache memory (a **cache miss**), then a normal memory access occurs, but, the information is passed to both the CPU and the cache memory.

In addition, while the CPU is using the information, the cache memory system fetches nearby information from memory, independently of the CPU, so that if neighbouring information is required (a likely event), then it will already be in cache memory and can be accessed very quickly. If enough cache memory is available, the instructions making up a loop in a program could be stored in cache. This would mean that the loop could be executed an arbitrary number of times without causing any memory fetches for instructions, after the instructions have

[2]Cache memory is designed so that it can be checked very quickly to ascertain if an item is stored in it.

been initially fetched. This can yield great improvements in program execution speeds.

In this way a cache hit rate of 90% and greater is possible, i.e. 90% or more of information requested by the CPU is found in cache memory, without the CPU having to access main memory. The speed of a memory system using cache memory is the weighted average of the cache speed and main memory speed.

For example, assume a 100ns delay for main memory and 20ns delay for cache memory with a 90% hit rate, then the apparent speed of memory access is

$$(0.9 * 20) + (0.1* 100) = 28\text{ns}.$$

This is a significant improvement in memory access performance, since the access time is now on average 28ns as opposed to 100ns if cache memory was not used.

Cache memory is now also included **on the CPU chip** (**on-chip cache**) of many microprocessors such as Intel's Pentium microprocessors, Digital's Alpha microprocessor and the IBM/Apple PowerPC microprocessor.

Since the cache memory is on the CPU chip, the speed of cache memory access is improved over that of **off-chip** cache memory. The capacity of on-chip cache memory varies from 8Kb to around 32Kb at the moment while **off-chip** cache memory may range from x00Kb to a few megabytes.

Computers may use separate memories to store instructions and data and such an architecture is called a **Harvard Architecture** because this idea emerged from machines built at Harvard University.

Instructions and data may be stored in the same cache memory which is referred to as a **unified cache** memory. Alternatively, separate caches for instructions and data may be maintained along the lines of the Harvard Architecture. The advantage of the Harvard Architecture is that instructions and data can be fetched simultaneously, i.e. in parallel, since they will be connected to the other CPU components by separate buses.

Measuring Computer Performance

How can we compare the performance of different computer **systems** ?

It is important to consider the performance of a computer **system** as a whole, including both the hardware and software and not just to consider the performance of the components of the system in isolation. It is very important to understand that measuring computer performance is a very difficult problem.

There are a number of criteria that can be used for measuring the performance of a computer system and it is a non-trivial matter as to how to weigh up the relative importance of these criteria when comparing two computer systems. In addition, it is not easy to obtain totally objective information about different manufacturers' machines or to get the information in a form that makes it easy to use for comparison purposes.

Computer Performance Metrics

There are a number of ways of measuring the performance of a computer system or indeed that of the components that make up a computer system. One common measure is **processor speed**. The question of how to measure processor speed is not as simple as it appears.

One simple measure is the number of instructions that can be executed per second, expressed in millions of instructions per second or **MIPS**. So, a given processor might have a processor speed of 5 MIPS, i.e. it can execute 5 million instructions per second.

But, all instructions do not take the same amount of execution time. An instruction to clear a register might take 1 clock cycle. A multiplication instruction might take more than 10 clock cycles.

The way to compute the MIPS rate more usefully is to calculate the average time the processor takes to execute its instructions, weighted by the frequency with which each instruction is used.

However, the frequency of instruction usage depends on the software being used. So for example, word processing software would not require much use of a multiplication instruction whereas spreadsheet software would be more likely to use many multiplication instructions. When we compare the MIPS rate of two different machines, we must ensure that they are counting the same type of instructions. Another problem with the MIPS metric, is that the amount of work an individual instruction carries out, varies from processor to processor. For example, a single VAX `add` instruction is capable of adding two 32-bit memory variables and storing the result in a third memory variable, whereas a machine such as the 8086 requires three instructions (e.g. an `add` and two `mov` instructions) to accomplish the same task. Naively counting the number of `add` instructions that can be executed per second on these two machine will not give a true picture of either machines performance.

We must also remember to take account of the processor word size when comparing instruction counts. A 32-bit processor is a more powerful machine than a 16-bit processor with the same MIPS rate, since it can operate on 32-bit operands as opposed to 16-bit operands. Similarly, a 64-bit processor will be more powerful than a 32-bit processor with the same MIPS rating.

Another metric that is similar to the MIPS rate is the **FLOPS** or floating-point operations per second rate which is expressed in megaFLOPS (MFLOPS) and gigaFLOPS (GFLOPS). This metric is used particularly for machines targeted at the scientific/engineering community where a lot of applications software requires large amounts of floating-point arithmetic. executed more quickly than others (e.g. addition versus division). Like the MIPS metric, the FLOPS metric needs to be approached with caution.

The MIPS and MFLOPS metrics are concerned with processor speed. The processor is a crucial component of a computer system when it comes to performance measurement but it is not the only one and, in fact, it may not be the determining factor of the performance of a system.

Other components are the memory and I/O devices whose performance are also crucial to the overall system performance. For example, a database application may require searching for information among millions of records stored on hard disk. The dominant performance metric for such an application is the speed of disk I/O operations. In such an application, the CPU spends most of its time waiting for disk I/O operations to complete. If we use a faster processor without increasing the speed of the disk I/O operations, then the overall improvement in performance will be negligible.

I/O performance may be measured in terms of the number of megabytes that can be transferred per second (I/O bandwidth). This can be deceptive, as the maximum transfer rate quoted may not be achieved in practice.

Take disk I/O, if the information required is stored on different tracks, then the seek time to move the head to the required tracks will slow I/O down considerably, whereas if the information is on the same track it can be transferred much more quickly. Another measure is the number I/O operations that can be completed per second, which can take account of the fact that the information may not be conveniently available on disk.

Memory performance is often measured in terms of its access time, i.e. the time taken to complete a memory write or read operation which is in the 20 to 100ns range. The **amount of memory** present is also a very important factor in system performance. The larger the amount of memory present, the less likelihood for page faults to occur in a virtual memory system.

Overall **system performance** is determined by the performance of the processor, memory and I/O devices as well as the operating system and applications software performance.

One measure of system performance to take account of the processor, memory and I/O performance is the number of **transactions** per second (TPS) that the system can cope with. A transaction might be defined as the amount of work required to retrieve a customer record from disk, update the record and write it back to disk, as modelled on a typical bank transaction.

System performance can also be evaluated by writing **benchmark** programs and running the same benchmark program on a series of machines.

A benchmark program is one written to measure some aspect of a computer's performance. For example, it might consist of a loop to carry out 1 million floating-point additions or a loop to carry out a million random read and write operations on a disk file. The time taken to execute the benchmark program gives a measure of the computer system's performance. By constructing a number of such programs for the different aspects of a systems performance, a suite of benchmarks may be developed that allow different computer systems to be compared. Benchmark programs are also used to test the performance of systems software such as compilers. The size of the executable file produced by the compiler and the efficiency of the machine code are two metrics that are used to compare compilers.

Another system performance measure is the **SPECmark** which is obtained in a similar fashion to the use of benchmark programs, except instead of using contrived benchmark programs, a suite of ten real world application programs are used (these include a compiler application, a nuclear reactor simulation and a quantum chemistry application). SPECmarks are used by companies such as Sun and Hewlett Packard to measure the performance of their workstations.

One difficulty about using benchmarks and SPECmarks to evaluate system performance, is that you are dependent on a compiler to translate the programs into efficient machine code for the computer under evaluation.

However, compilers are not all equally efficient and particularly with the advent of new processor features such as multiple functional units capable of parallel operation, compiler writers have a complex task in designing good so-called **optimising** compilers. The code produced by a poor quality compiler can run significantly slower and use more memory than code produced by a good compiler. A computer manufacturer may have benchmark programs optimised for a particular machine so that the machine's performance is apparently superior to another machine running the same benchmark programs which have not, however, been optimised for the second machine! Caveat emptor ("Let the buyer beware").

In addition to the effects of the compiler, the operating system will also influence the performance of a computer system. In the case, where two computer systems running different operating systems, are being evaluated, care must be taken to ensure that both operating systems are properly configured. For example, a good machine with an efficient operating system may perform poorly as a result of having insufficient memory allocated to user programs, in comparison to a less powerful machine which has been expertly tuned to run its programs.

Other non-performance issues that arise when comparing computer systems are the actual cost of the system, and its reliability together with the availability of hardware and software support. Computer systems **fail** due to either hardware or software problems, or both. It is fundamentally important to take account of computing failures, from the moment of evaluating a new system right through to its day to day operation.

There's no point in having the most sophisticated computer system in the world, if **when** it fails, you cannot get it operational again, in a short period of time.

In summary, evaluating computer system performance is fraught with difficulties and the use of apparently simple metrics such as MIPS and MFLOPS can be quite misleading. The reality is that there is no easy alternative to that of running real world application programs (preferably the ones you wish to use) and choosing the system which best matches the performance criteria that you have laid down.