Chapter 7. Case Studies on Features Improving Performance

This chapter will discuss a number of case studies, each of which examines a particular feature used to improve computer system performance.

- Increasing Clock Rate
- Adding a CPU Cache
- Adding General Purpose Registers
- Adding an addition System Bus
- Direct Memory Access (DMA)

The following three approaches can summarize the features for improving computer performance.

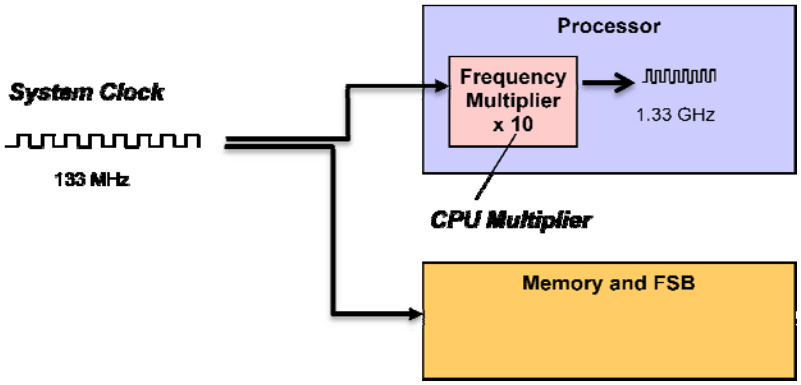
- Performing a task faster.
- Performing tasks in parallel.
- Avoid performing certain time consuming tasks.

1. Increasing Clock Rate

Increasing the clock rate is a simple method to improve the execution performance. An increased clock rate can make more actions to take place in the same period of time. For the same number of RTL steps, doubling the clock rate theoretically reducing the time taken by half.

There are several limitations concerning increasing clock rate to improve performance.

- All components have operating ranges and the clock rate is one of them. For example, the Memory System might need at least 5 ns to complete a read/write operation, and so the clock cycle cannot be shorter than 5 ns (or the clock rate cannot be greater than 1/5ns or 200 MHz)
- Components running at high speed generate heat. A too high clock rate may generate heat beyond the capability of the designated cooling device.
- Components running beyond the designated speed shorten the lifespan.



Over-clocking is carried out by enthusiasts to boost their computer performance. The above diagram shows that there is a system clock driving the clock signals for other components. Modern processors have a frequency multiplier that multiplies the system clock for the internal processor clock. The ratio between the system clock and the internal processor clock is the **CPU multiplier**.

- Increasing the system clock drives up both the processor and the memory system.
- Increasing the CPU multiplier can achieve over-clocking in the processor only.
- To improve component stability under over-clocking, increasing the operating voltage of the component may help.
- The increased clock rate will cause more heat generation. Additional cooling should be installed to help dissipating the heat.

Over-clocking can potentially damage computer components. It may also cause errors in computation due to lower stability of the components. The processor performance gained is actually not that significant.

This section is a theoretical discussion of over-clocking and it does not teach you how to over-clock.

2. Adding CPU Cache

The clock rates for CPU and main memory are significantly different. More importantly, the potential throughput is also significantly different. Increasingly the main memory has become a major performance bottleneck. The throughput main memory cannot satisfy the fast processor's demand for instructions and data.

CPU cache is a feature that reduces the processor's dependency on the throughput of the main memory.

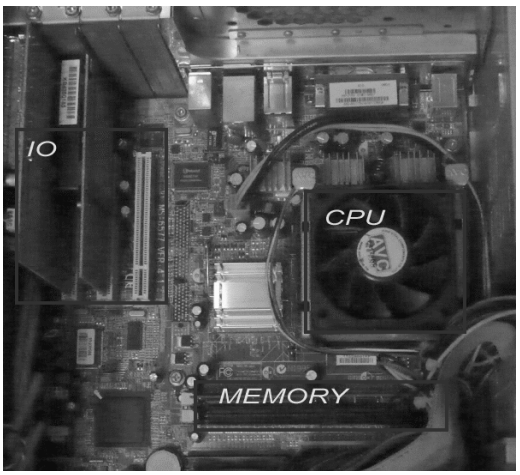
Processor-Memory Performance Gap

In the past, the CPU speed has doubled around every 18 months (Moore's Law) but the Memory speed could only double in ten years time. This phenomenon is often called **Processor-Memory performance gap**. Memory speed increment is hampered by several factors:

- There are memory technologies that are fast but expensive. Increased demand in memory size requires most computers to use more economical types of memory as its primary memory.
- The large memory size requirement complicates address decoding and handling.
- The large memory size requirement also complicates physical packaging and layout of memory chip on printed circuit boards.
- The large memory size requirement makes it physically impractical to closely integrate with the CPU.
- The bandwidth between chips is limited.

For the last point, refer to the following figure.

The figure shows a motherboard where the CPU, Memory and other components are physically laid down. The CPU and the Memory are connected by inter-chip communication and because of physical limitation the bandwidth cannot match the data movement speed within the CPU.



The concept of **Memory Wall** was postulated by Bill Wulf and Sally McKee in 1994. They predicted that the divergence of CPU speed and Memory speed increment would soon make all computer performance to be dominated by Memory speed.

Reading: Memory Wall
http://www.acm.org/crossroads/xrds5-3/pmgap.html

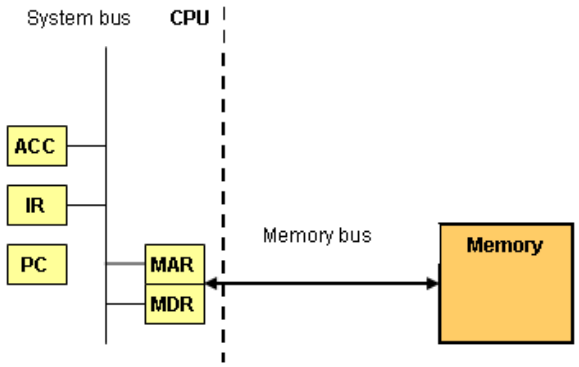
Memory Operations

The very first design of programmable computer assumes that all the components, including the CPU and the Memory, are operating at the same clock rate.

Look at the following fetch part of a LMC instruction in RTL. The CPU expects the Memory to have the data ready at the MDR in one clock cycle.

ADD 20

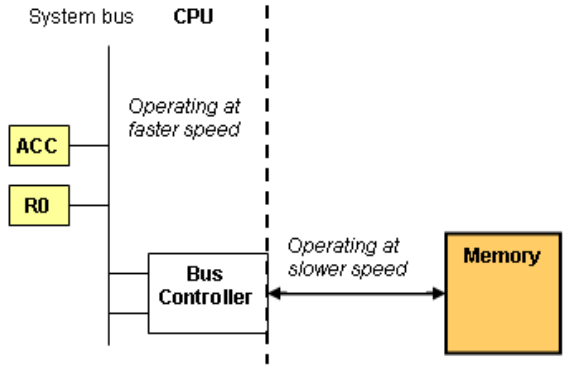
```
PC > MAR
M[MAR] > MDR  ; expects the memory to have the data ready in one clock cycle
MDR > IR
```



Consider that the CPU technology has made advancement and it can operate at a faster clock rate. However, the Memory System's speed has remained the same. With our current design, the CPU has to operate at the same speed as the Memory, therefore unable to exploit the improvement in CPU performance.

Separating Memory from Processor

A solution to this problem is to separate the two buses and to allow them running at different speeds. A bus controller is placed between the two buses to perform coordination of data movement between the two.

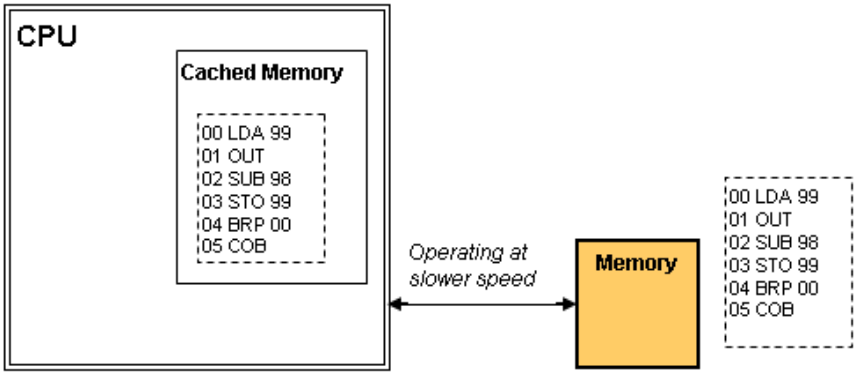


The CPU can now run at a faster clock rate, but there is another problem. The program instructions are still located at the Main Memory. During the execution of each instruction, the CPU still has to wait for the slower Memory System to respond. A solution is to prevent, as far as possible, the CPU to access the Memory System.

CPU Cache

CPU memory cache is a very fast memory subsystem located within the CPU. The act of the CPU cache making a copy of the data in the main memory is called caching. Caching is a technique that allows the CPU to operate more independently of the Memory System. If the required data or instruction is already within the CPU, the CPU needs not read from the Memory System.

The following figure describes a scenario of how caching would allow the CPU to operate independently.



Consider that the computer is executing a small LMC program that involves a loop.

- As the execution begins, the CPU reads from the Memory System each instruction one by one.
- The CPU needs to wait every time for the slower Memory System to catch up.
- The key operation is that the CPU is saving each instruction in the CPU memory cache. When the loop is executed the second time, the instructions to be executed are already available in the CPU memory cache. The CPU can read directly from the CPU cache and can operate with fewer accesses to the Memory System.

Performance of CPU Cache

The ideal CPU cache is one that always stores the required data (or instruction). Memory operation is not needed at all.

The cache **hit rate** is the percentage of requests that can be satisfied by the cache. The ideal CPU cache would have a hit rate of 100%.

Theoretically a high hit rate can be achieved by:

- Larger cache size.
- More accurate predictions about the future data requests. Keep such data in the cache.

A drawback of larger cache size is higher latency due to search operation. A search operation in the cache would be involved with every memory operation. Although the cached data is organized efficiently in a data structure, the search time still increases with larger cache size.

Multi-level cache is designed to provide a better balance between cache size and latency. A lower level cache (e.g. L1 cache) is smaller in size and so latency is low. The second level cache is of a larger size, and subsequently higher level cache has increasing size. The search begins with the low level cache first.

Locality of Reference

CPU cache manager can often make correct predictions about future requests. Even there is no crystal ball.

The **locality of reference** is a phenomenon that the next memory request is more likely to be at a nearby address than at any other addresses. For example, if the current memory request is at address 1000, then the chance of requesting address 1001 is higher than any other addresses.

The locality of reference exists because of some properties of program execution, including:

- Sequential execution model: usually the next instruction to execute is at the next address.
- Local branching: even if a branching occurs, the address to branch would be nearby if branching is associated with a condition (if-else) or repetition (for-while) structure.
- Array processing: data in an array is usually assessed from one end to another end.

Coherency in Caching

With caching, a piece of data would appear in several copies in different storage facilities. Modification to the value in one place would render the other places incorrect. This is actually acceptable provided that there is only one process accessing the data.

In multi-programming environment, this is a potential problem because there could be another program accessing the other copies of the same data. This happens when the CPU switches from one process to another. The situation is more complicated in multi-processor environment, and in a distributed environment. There are many methods to guarantee consistency but each of them requires effort to manage the data copies.

3. Adding General Purpose Registers

Registers are at the top of the memory hierarchy and they are very fast memory.

Types of Registers

General purpose registers are also called user-visible registers. They can be directly manipulated with instructions, such as storing/loading data for such registers.

The LMC has only one general purpose register, the accumulator (ACC). It can be manipulated with LDA, STO, ADD, SUB, IN and OUT instructions.

Specific purpose registers are used to support the operation of the CPU in the fetch and execution cycle. Examples of specific purpose registers include the following:

- Program Counter Register (PC) for storing the program counter.
- Instruction Register (IR) for temporarily storing the instruction loaded from the memory.
- Memory Address Register (MAR) for holding the address of a memory location where data may be loaded or stored.

- Memory Data Register (MDR) for holding the data involved in a load/save operation with a memory location.
- Status Register for holding the various statuses during the operation of the CPU such as arithmetic errors (e.g. overflow or carry, low power, etc). A status is indicated with a flag, often 1-bit wide. For example, the 8086 and 8088 chips have a 16-bit status register, storing the following flags: had many flags: carry, parity, auxiliary carry, zero, sign, trap, interrupt enable, direction, overflow, IO protection, nested task, resume, and virtual 8086 mode.
- IO Registers for holding the data and identity of the IO device. This is not often used in modern architectures.
- There are other special purpose registers such as Constant Registers for holding special value such as zero and one.

Role of General Purpose Registers

General purpose registers located in the processor are useful for storing intermediate and temporary data. Most processes and operations involve a lot of steps. Each step would consume data from the previous steps and generate data for the next steps. A programmer can write instructions to store these data in general purpose registers.

LMC has only one general purpose register. A lot of memory operations are found in LMC programs because intermediate data can only be stored in the main memory. There is no spare register for storing these in the processor.

Memory operations are slow and the performance would be significantly improved if memory operations related to intermediate data could be avoided.

The following LMC program shows such an example.

Example: LMC program

This program reads two integers and prints the larger integer.

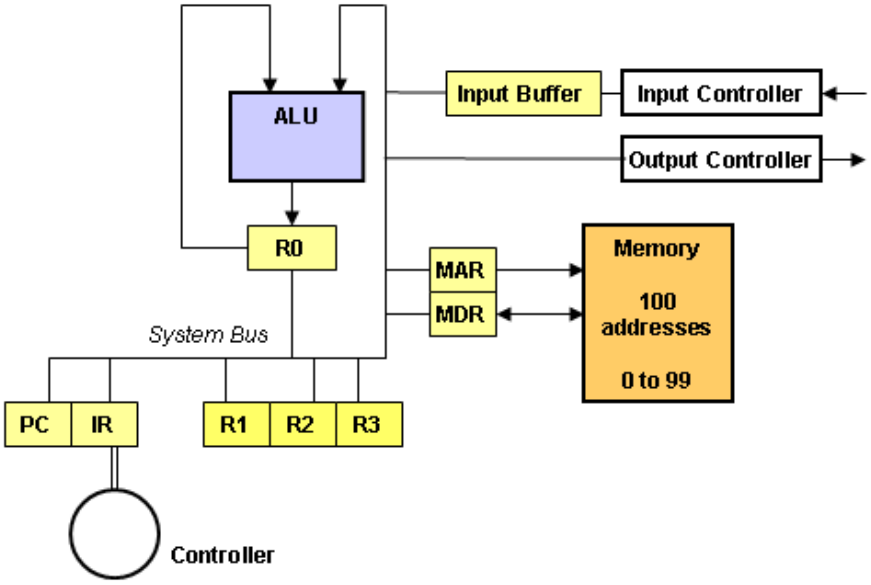
The instructions STO, SUB, and LDA involve loading or storing data from/to the Memory System. It is because the CPU has only the ACC and no other place to store intermediate data.

```
00 IN
01 STO 11
02 IN
03 STO 12
04 SUB 11
05 BRP 08
06 LDA 11
07 BR 09
08 LDA 12
09 OUT
10 COB
11 DAT
12 DAT
```

The data movement between CPU and Memory System can be reduced with more general-purpose registers in the CPU.

Adding General Purpose Registers

Three new general-purpose registers to the LMC, named R1 to R3, are added to improve the efficiency. The old accumulator ACC is renamed to R0. These three new registers are connected to the CPU system bus. The following figure shows the revised design.



The new generation purpose registers are not usable without adding new instructions for manipulation them. The following describes two new LMC instructions:

- MOV RA, RB: copy data from Register B (RB) to Register A (RA).
- SUB RA: subtract RA from R0 and store the result to R0.

The following shows the RTL steps for the two new instructions.

MOV RA, RB	**SUB RA**

`PC > MAR`
`M[MAR] > MDR`
`MDR > IR`
`PC + 1 > PC`
`R[B] > R[A]`
	`PC > MAR`
	`M[MAR] > MDR`
	`MDR > IR`
	`PC + 1 > PC`
	`R0 – RA > R0`

The two new instructions require 5 RTL steps. If each step takes one clock cycle, the instruction takes 5 clock cycles.

- They take two less RTL steps compared to the original LMC SUB instruction, which takes 7 RTL steps.
- Memory operations may take more than one clock cycle, and so comparatively two new instructions are even faster because they carry out fewer memory operations.

With the new instructions MOV and SUB, the LMC program is rewritten as the following to exploit the new general purpose registers.

Example: Revised LMC program

The instructions STO, SUB, and LDA involve loading or storing data from/to the Memory System. It is because the CPU has only the ACC and no other place to store intermediate data.

```
00      IN          ; #1 store in R0
01      MOV R1, R0  ; R1 = R0
02      IN          ; #2 store in R0
03      MOV R2, R0  ; R2 = R0
04      SUB R0, R1  ; R0 = R0 – R1
05      BRP 08      ; if R0 >= 0
06      MOV R0, R1  ; R0 = R1 R1 stores #1
07      BR 09
08      MOV R0, R2  ; R0 = R2 R2 stores #2
09      OUT
10      COB
```

The revised program should perform better. The program is shorter and some instructions also take shorter time to execute.

Example: Performance Evaluation

Question: Compare the execution of the two programs and evaluate the performance gain.

Answer:
A number of quantitative measurements can be used to compare the two programs. Two of them will be used here: RTL steps (similar to clock cycles) and memory operations. Normally, all instructions that would have been executed in the programs are taken into consideration. The programs have no loop and making it easier.

The program has a conditional branch. For simplicity, only the case of first integer greater than second integer is considered.

	Original Program	***Revised Program***
Number of instructions	10	10
Number of RTL steps	4 + 7 + 4 + 7 + 7 + 4 + 7 + 4 + 4 + 3 = 51 RTL steps	4 + 5 + 4 + 5 + 5 + 4 + 5 + 4 + 4 + 3 = 43 RTL steps
Number of memory operations	1 + 2 + 1 + 2 + 2 + 1 + 2 + 1 + 1 + 1 = 14 memory operations	1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 = 10 memory operations

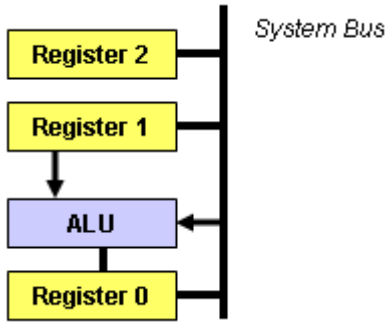
4. Parallel Execution and Adding another System Bus

Performing tasks in parallel can certainly shorten the time to complete. However, parallel processing must satisfy the following requirements:

- The tasks are independent. For example, if task B depends on the result of task A, then A and B cannot be performed together.
- Additional resources are available for the parallel execution of the tasks.

This case study investigates the effect of parallel execution of RTL operations.

Consider that we have an instruction ADD R1, R0, R2 that adds R1 to R0 and the result is stored in R2. This is a register-based instruction. In the execution phase, all data movement happens on the system bus inside the CPU. The system bus design is shown in the following.



The RTL for the instruction is given below. There are 6 steps and it takes 6 clock cycles to complete the execution (assume 1 clock cycle per RTL step).

ADD R1, R0, R2

```
PC > MAR
M[MAR] > MDR
MDR > IR
PC + 1 > PC
R[0] + R[1] > R[0]
R[0] > R[2]
```

If the 6 RTL steps could be executed in parallel, then it would just take 1 clock cycle to complete the execution. There are a few reasons why this is not possible.

- A RTL operation is dependent on the result of a previous RTL operation. For example, the second RTL operation requires the loading of MAR from the first RTL operation. These two operations cannot happen together.
- Hardware design restricts the possible parallel operation. The last two RTL operations require the system bus and so they cannot happen together.

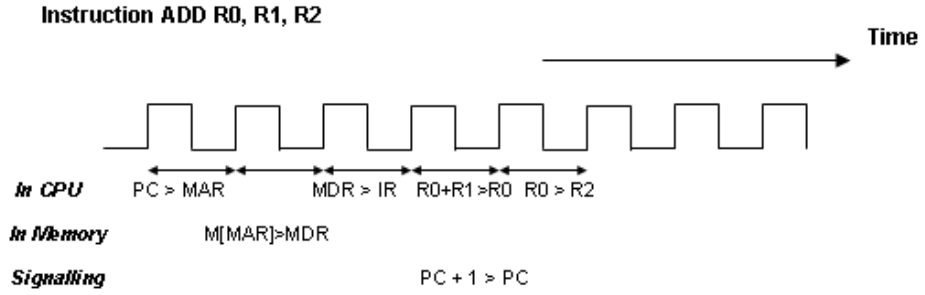
The multi-point bus can only support one pair of components to communicate at a time. Let us examine which of the steps use the system bus. Step #1, #3, #5, and #6 happens on the system bus.

RTL step	***Location***
PC > MAR	System Bus
M[MAR] > MDR	Memory
MDR > IR	System bus
PC + 1 > PC	Control Unit Signal
R[0] + R[1] > R[0]	System Bus
R[0] > R[2]	System Bus

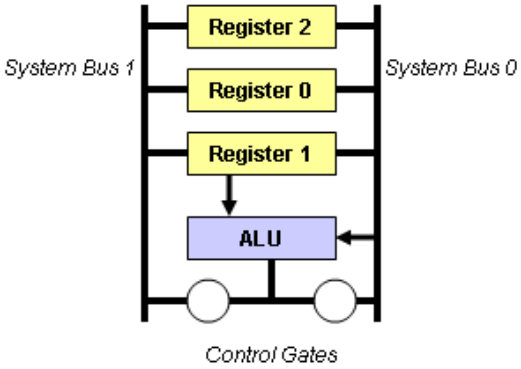
If step #2 and step #4 do not use the system bus, the problem is to consider whether it is possible for them to happen in parallel with any other steps.

- Step #2: Not possible because step #2 cannot occur before completion of step #1. Also, step #3 cannot occur before completion of step #2. There is data dependency between the first 3 steps.
- Step #4: The increment of PC is caused by a signal from the Control Unit. This step can happen in parallel with step #5.

The following shows the timing information of the execution of the instruction. The instruction now takes one less time cycle to complete.



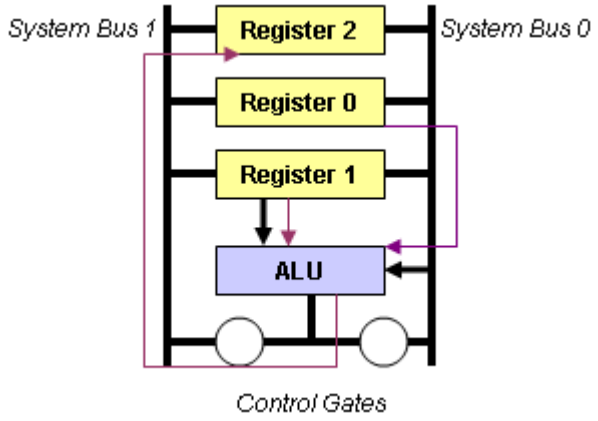
Adding one more system bus can increase the opportunity for parallel execution. The following figure shows a possible design based on two system buses.



The output port of the ALU should be connected to both system buses to facilitate the movement of data. However, the two system buses cannot be connected because they should be carrying different data. A pair of control gates is placed at the output port of the ALU to control which system bus is connected to the output port.

With the additional system bus, the last two steps in the instruction ADD R1, R0, R2 can happen in parallel.

RTL step	***Location***
PC > MAR	System Bus #0
M[MAR] > MDR	Memory
MDR > IR	System bus #0
PC + 1 > PC	Control Unit Signal
R[0] + R[1] > R[2]	System Bus #0 and System Bus #1



Adding an additional system bus provides opportunities for parallel execution of the steps in the fetch and execution cycle. There is cost implication of adding a bus, however, and the designer must weigh the cost and benefits.

5. Direct Memory Access (DMA)

Direct memory access is a technique that allows IO-to-Memory operation to occur in parallel with processor execution of instructions.

IO-to-Memory operations occur quite frequently in modern computers:

- Loading programs from hard-disk to the main memory before execution.
- Loading data for program processing.

In the current computer design, the CPU needs to take care of IO operations through handling interrupts, even in asynchronous IO operation mode.

- CPU executes an instruction to initiate an IO operation.
- CPU continues to execute other instructions, and leaving the IO operation to run in parallel.
- IO operation completes and raises an interrupt.
- CPU suspends the current execution and handles the interrupt.
- After the interrupt is handled, the CPU resumes the suspended execution of instructions.

For a busy high speed device handling many requests, there will be too many interrupts. Each interrupt will hamper the smooth operation of the CPU and the CPU is forced to do a lot of IO handlings instead of executing programs.

One solution to free the CPU from IO activities is to allow the IO devices to communicate with each other independently.

- High-speed devices use a method called direct memory access (DMA), in which device controller transfer a whole block of data directly between the main memory and the device local buffer.
- Only one interrupt is generated per block.
- A DMA controller is instructed by the device driver (in the OS) of the address of a buffer (in the main memory) and the length of data to copy. CPU can do other things independently.
- The major problem is the Memory System can serve only one request at a time. DMA still competes with the CPU for memory system access.

The following figure shows the operation of DMA.