جامعة ديالى
كلية التربية الأساسية
قسم الحاسبات

المعمارية

المرحلة الثانية

اعداد

م.م حيدر

للعام الدراسي (٢٠١٦-٢٠١٧)

للفصل الدراسي الثاني

Response to the new interrupt will depend upon the priority of the newly arrived interrupt with respect to that of the interrupt being currently served.

- If the newly arrived interrupt has priority less than or equal to that of the currently served one, then it can wait until the processor finishes serving the current interrupt.

- If, on the other hand, the newly arrived interrupt has priority higher than that of the currently served interrupt.

For example, power failure interrupt occurring while serving an I/O interrupt, then the processor will have to push its status onto the stack and serve the higher priority interrupt.

4-Direct Memory Access (DMA)

We have discussed the data transfer between the processor and no devices. We have discussed two different approaches namely prograrhmed I/O and Intpt-driven tro Both the methods require the active intervention of the processor to transfer data between memory and the I/O module, and any data transfer must transverse a path through the processor. Thus both these forms of tro suffer from two inherent drawbacks.

1- The I/O transfer rate is limited by the speed with which the processor can test and service adevice.

2- The processor is tied up in managing an I/O transfer; a number of instructions must be executed for each I/O transfet.

To transfer large block of data at high speed, a special control unit may be provided to allow transfer of a block of data directly between an external device and the main memory, without continuous intervention by the processor. This approach is called direct memory access or DMA.

DMA transfers are performed by a control circuit associated with the I/O device and this circuit is referred as DMA controller. The DMA controller allows direct data transfer between the device and the main memory without involving the processor.

To transfer data between memory and I/O devices, DMA controller takes over the control of the system from the processor and transfer of data take place over the system bus. For this purpose, the DMA controller must use the bus only when the processor does not need it, or it must force the processor to suspend operation temporarily. The later technique is more common and is referred to as cycle stealing, because the DMA module in effect steals a bus cycle.

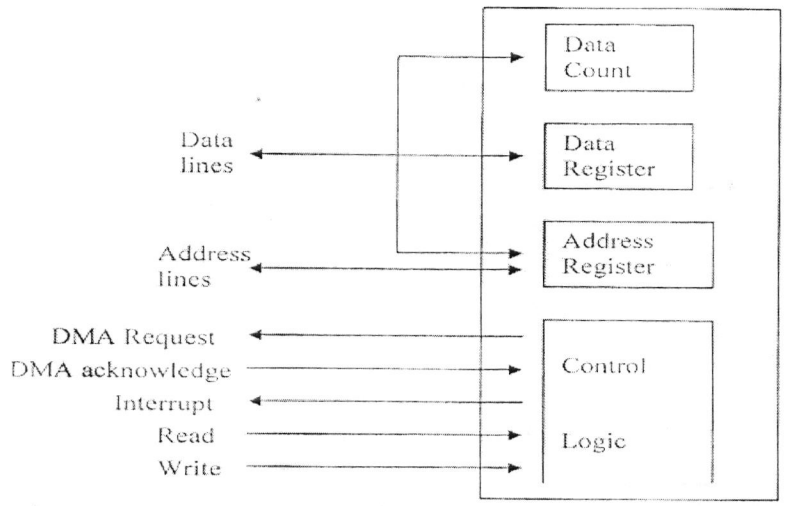


Figure: DMA Block daigram

When the processor wishes to read or write a block of data, it issues a command to the DMA module, by sending to the DMA module the following information.

1-Whether a read or write is requested, using the read or write control line between the processor and the DMA module.

2-The address of the I/O devise involved, communicated on the data lines.

3-The starting location in the memory to read from or write to, communicated on data lines and stored by the DMA module in its address register.

4-The number of words to be read or written again communicated via the data lines and stored in the data count register.The processor then continues with other works. It has delegated this no operation to the DMA module. The DMA module checks the status of the I/O devise whose address is communicated to DMA controller by the processor. If the specified I/O devise is ready for data transfer, then DMA module generates the DMA request to the processor. Then the processor indicates the release of the system bus through DMA acknowledge.The DMA module transfers the entire block of data, one word at a time, directly to I/O from memory, without going through the processor.When the transfer is completed, the DMA module sends an interrupt signal to the processor. After receiving the interrupt signal, processor takes over the system bus.Thus the processor is involved only at the beginning and end of the transfer. During that time the processor is suspended. It is not required to complete the current instruction to suspend the processor. The processor may be suspended just after the completion of the current bus cycle. On the other hand, the processor can be suspended just before the need of the system bus by the processor, because DMA controller is going to use the system bus, it will not use the processor. The point where in the instruction cycle the processor may be suspended shown in the figure below.

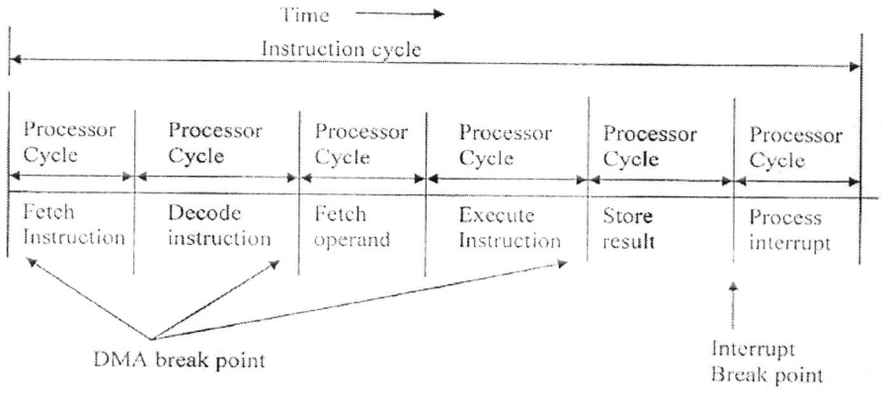


Figure: Instruction Cycle

When the processor is suspended, then the DMA module transfer one word and return control to the processor.

Note that, this is not an interrupt, the processor does not save a context and so something else.Rather, the processor pauses for one bus cycle. During that time processor may perform some other task which does not involve the system bus.In the worst situation processor will wait for some time, till the DMA releases the bus.

The net effect is that the processor will go slow. But the net effect is the enhancement of performed, because for a multiple word I/O transfer, DMA is far more efficient than interrupt driven or programmed I/O.

The DMA mechanism can be configured in different ways. The most common amongst them are:

o Single bus, detached DMA - I/O confrgtration.

o Single bus, Integrated DMA- I/O confrguration.

o Using separate I/O bus.

Single bus, detached DMA- I/O confrguration.

In this organization all modules share the same system bus.The DMA module here acts as a surrogate processor. This method uses programmed I/O to exchange data between memory and an I/O module through the DMA module.

For each transfer it uses the bus twice. The first one is when transferring the data between I/O and DMA and the second one is when transferring the data between DMA and memory. Since the bus is used twice while transferring data, so the bus will be suspended twice. The transfer consumes two bus cycle. The interconnection organization is shown in the figure(a).

Single bus, Integrated DMA- I/O confrguration.

By integrating the DMA- I/O and function thq number of required bus cycle can be reduced. In this configuration, the DMA module and one or more I/O modules are integrated together in such a way that the system bus is not involved. In this case DMA logic may actually be a part of an I/O module, or it may be a separate module that controls one or more fio modules.

The DMA module, processor and the memory module are connected through the system bus. In this configuration each transfer will use the system bus only once and so the processor is suspended only once.

The system bus is not involved when transferring data between DMA and I/O device, so processor is not suspended. Processor is suspended when data is transferred between DMA and memory. The configuration is shown in the figure(b).

Using separate I/O bus.

In this configuration the I/O modules are connected to the DMA through another I/O bus.In the case the DMA module is reduced to one.

Transfer of data between I/O module and DMA module is carried out through this I/O bus. In this transfer, system bus is not in use and so it is not needed to suspend the proiessor.

There is another transfer phase between DMA module and memory. In this time system bus is needed for transfer and processor will be suspended for one bus cycle. The configuration is shown in the figure(c).

1. The CPU directly controls a peripheral device. This is seen in simple microprocessor-controlled devices.
2. A controller or I/O module is added. The CPU uses programmed I/O without interrupts. With this step, the CPU becomes somewhat divorced from the specific details of external device interfaces.
3. The same configuration as in step 2 is used, but now interrupts are employed. The CPU need not spend time waiting for an I/O operation to be performed, increasing efficiency.
4. The I/O module is given direct access to memory via DMA. It can now move a block of data to or from memory without involving the CPU, except at the beginning and end of the transfer.
5. The I/O module is enhanced to become a processor in its own right, with a specialized instruction set tailored for I/O. The CPU directs the I/O processor to execute an I/O program in memory. The I/O processor fetches and executes these instructions without CPU intervention. This allows the CPU to specify a sequence of I/O activities and to be interrupted only when the entire sequence has been performed.
6. The I/O module has a local memory of its own and is, in fact, a computer in its own right. With this architecture, a large set of I/O devices can be controlled, with minimal CPU involvement. A common use for such an architecture has been to control communication with interactive terminals. The I/O processor takes care of most of the tasks involved in controlling the terminals.

As one proceeds along this evolutionary path, more and more of the I/O function is performed without CPU involvement. The CPU is increasingly relieved of I/O-related tasks, improving performance. With the last two steps (5–6), a major change occurs with the introduction of the concept of an I/O module capable of executing a program. For step 5, the I/O module is often referred to as an *I/O channel*. For step 6, the term *I/O processor* is often used. However, both terms are on occasion applied to both situations. In what follows, we will use the term *I/O channel*.

Characteristics of I/O Channels

The I/O channel represents an extension of the DMA concept. An I/O channel has the ability to execute I/O instructions, which gives it complete control over I/O operations. In a computer system with such devices, the CPU does not execute I/O instructions. Such instructions are stored in main memory to be executed by a special-purpose processor in the I/O channel itself. Thus, the CPU initiates an I/O transfer by instructing the I/O channel to execute a program in memory. The program will specify the device or devices, the area or areas of memory for storage, priority, and actions to be taken for certain error conditions. The I/O channel follows these instructions and controls the data transfer.

Two types of I/O channels are common, as illustrated in Figure 7.15. A *selector channel* controls multiple high-speed devices and, at any one time, is dedicated to the transfer of data with one of those devices. Thus, the I/O channel selects one device and effects the data transfer. Each device, or a small set of devices, is handled by a *controller*, or I/O module, that is much like the I/O modules we have been

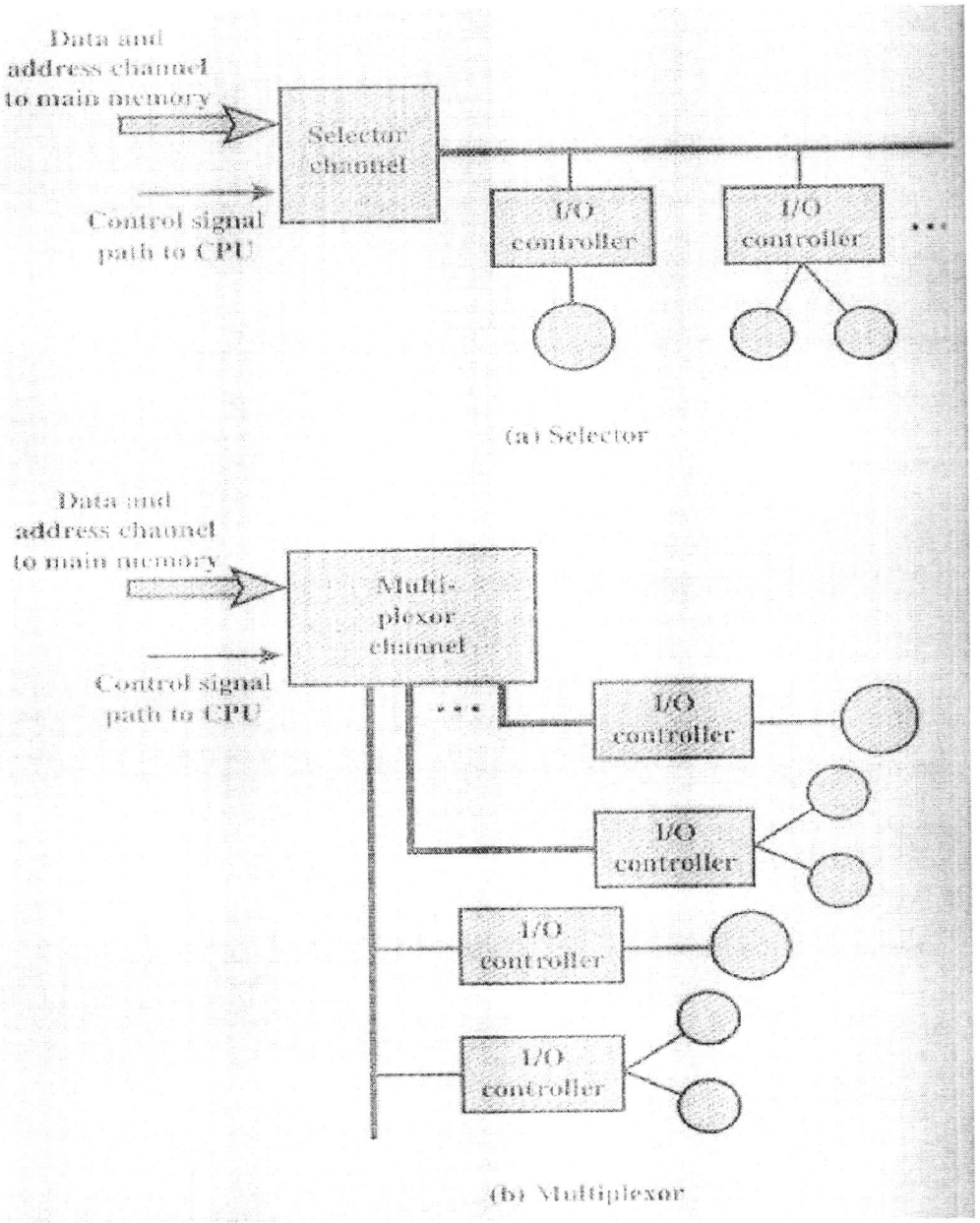


Figure: I/O Channel Architecture

discussing. Thus , the I/O channel serves in place of the CPU in controlling these I/O controllers. Amultiplexorchannel can handle I/O with multiple devices at the same time. For low- speed devices ,abyte multiplexor accepts or transmits characters as fast as possible to multiple devices.

Associative Memories

1- Content Addressing

In conventional memories could be access to the information by the address of the location that is contained it. There are another situations, where the reverse function is required, In other words, the contents are known, it is necessary to determine the location where this information has been stored.

It should be possible to determine the address of a memory location be means of the contents stored therein. Such memories called (content – addressed or Associative memories). These memories are very convenient to perform parallel searches by data association.

It is possible to address such memories by the data itself. When a location is to be accessed, the value of the contents of the location (or a part-a sub-field of the word) is supplied. The memory accesses all the locations in parallel and identifies all the locations where the contents match the specified value. These can then be read out.

When a new word is to be stored the address is specified. The word is stored in any unused (empty) location.

Example:

Function f(x) may have been stored for several values $(x_1,x_2,\ldots,x_n)$, this may have been as a table with two field $(x_i,f(x_i))$.

Let us assume that the values xi and $f(x_i)$ are both stored in a single location L_i with two sub-field. The problem is to determine the value $f(x_i)$ given x_i.

2-Operation of Associative Memories

Each location of the memory is assumed to contain an argument field and a contents field. The value which is required to be matched is loaded in to the argument register. This compared with the contents of the argument field of each location. The match register has one bit for

each location of the memory. Wherever the argument field of the location matches the content of the argument register, the corresponding bit is set in the match register. The match register tags all the locations where a match has been found. It is possible then to read the contents of each of these locations in sequential.

An optional 'key' register may be used to choose only particular bits in the argument for matching. Bits of the argument register are used for matching only if the corresponding bits of the key register are 1 .they are ignored otherwise. It is possible to construct associative memories with several argument fields. Anyone of these may be used at any given time for effecting an associative search, each having its own key register .

3-Applications

It is obvious that associative memories will be significantly more expensive than the corresponding regular memories. Consequently they are used only in application where the time available for associative search is very limited. They are typically used in virtual memory and cache systems.

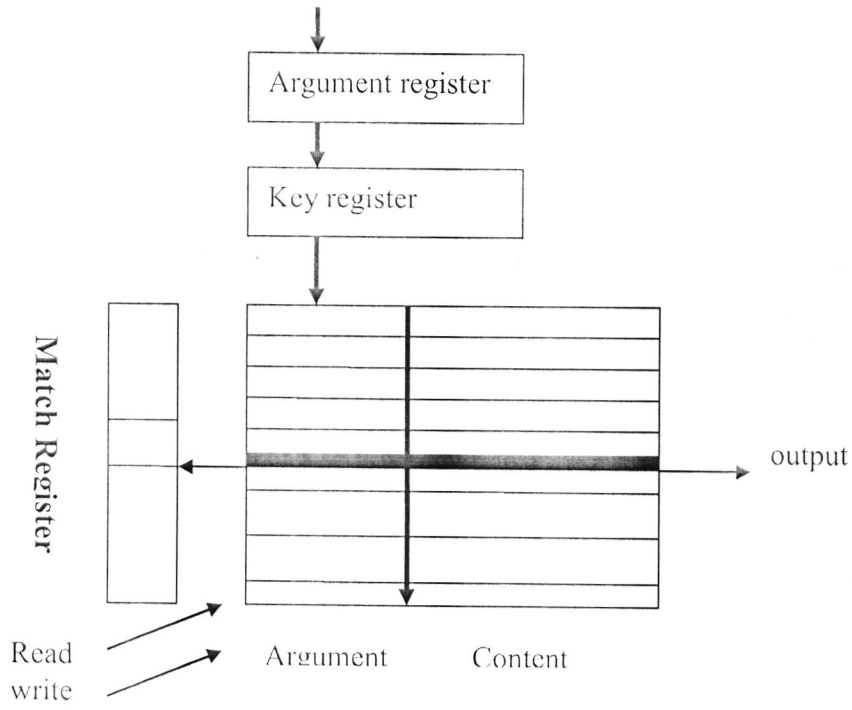


Figure: Associative Memory operation.

Cache Coherence Basic Concept

May be there are more than one cache memory in the system computer. Multiple copies of data, spread throughout the caches, lead to a coherence problem among the caches. The copies in the caches are coherent if they all equal the same value. However, if one of the processors writes over the value of one of the copies, then the copy becomes inconsistent because it no longer equals the value of the other copies. If data are allowed to become inconsistent (incoherent), incorrect results will be propagated through the system, leading to incorrect final results. Cache coherence algorithms are needed to maintain a level of consistency throughout the parallel system.

A-Cache–Memory Coherence

In a single cache system, coherence between memory and the cache is maintained using one of two policies:

(1) write-through.

(2) write-back.

When a task running on a processor P requests the data in memory location X, for example, the contents of X are copied to the cache, where it is passed on to P. When P updates the value of X in the cache, the other copy in memory also needs to be updated in order to maintain consistency. In write-through, the memory is updated every time the cache is updated, while in write-back, the memory is updated only when the block in the cache is being replaced. Table1 shows the write-through versus write-back policies.

		Write-Through		Write-Back	
Serial	Event	Memory	Cache	Memory	Cache
1		X		X	
2	P reads X	X	X	X	X
3	P updates X	X′	X′	X	X′

TABLE 1 Write-Through vs. Write-Back

B- Cache–Cache Coherence

In multiprocessing system, when a task running on processor P requests the data in global memory location X, for example, the contents of X are copied to processor P's local cache, where it is passed on to P. Now, suppose processor Q also accesses X. What happens if Q wants to write a new value over the old value of X? There are two fundamental cache coherence policies:

(1) write-invalidate.

(2) write-update.

Write-invalidate maintains consistency by reading from local caches until a write occurs. When any processor updates the value of X through a write, posting a dirty bit for X invalidates all other copies. For example, processor Q invalidates all other copies of X when it writes a new value into its cache. This sets the dirty bit for X. Q can continue to change X

without further notifications to other caches because Q has the only valid copy of X. However, when processor P wants to read X, it must wait until X is updated and the dirty bit is cleared.

Write-update maintains consistency by immediately updating all copies in all caches. All dirty bits are set during each write operation. After all copies have been updated, all dirty bits are cleared. Table2 shows the write-update versus write-invalidate policies.

		Write-Update		Write-Invalidate	
Serial	Event	P's Cache	Q's Cache	P's Cache	Q's Cache
1	P reads X	X		X	
2	Q reads X	X	X	X	X
3	Q updates X	X′	X′	INV	X′
4	Q updates X′	X″	X″	INV	X″

TABLE2 Write-Update vs. Write-Invalidate

C- Shared Memory System Coherence

The four combinations to maintain coherence among all caches and global memory are:

. Write-update and write-through;

. Write-update and write-back;

. Write-invalidate and write-through; and

. Write-invalidate and write-back.

If we permit a write-update and write-through directly on global memory location X, the bus would start to get busy and ultimately all processors would be idle while waiting for writes to complete. In write-update and write-back, only copies in all caches are updated. On the contrary, if the write is limited to the copy of X in cache Q, the caches become inconsistent on X. Setting the dirty bit prevents the spread of inconsistent values of X, but at some point, the inconsistent copies must be updated.

Pipelining Techniques

There exist two basic techniques to increase the instruction execution rate of a processor. These are to increase the clock rate, thus decreasing the instruction execution time, or alternatively to increase the number of instructions that can be executed simultaneously. Pipelining and instruction-level parallelism are examples of the latter technique.

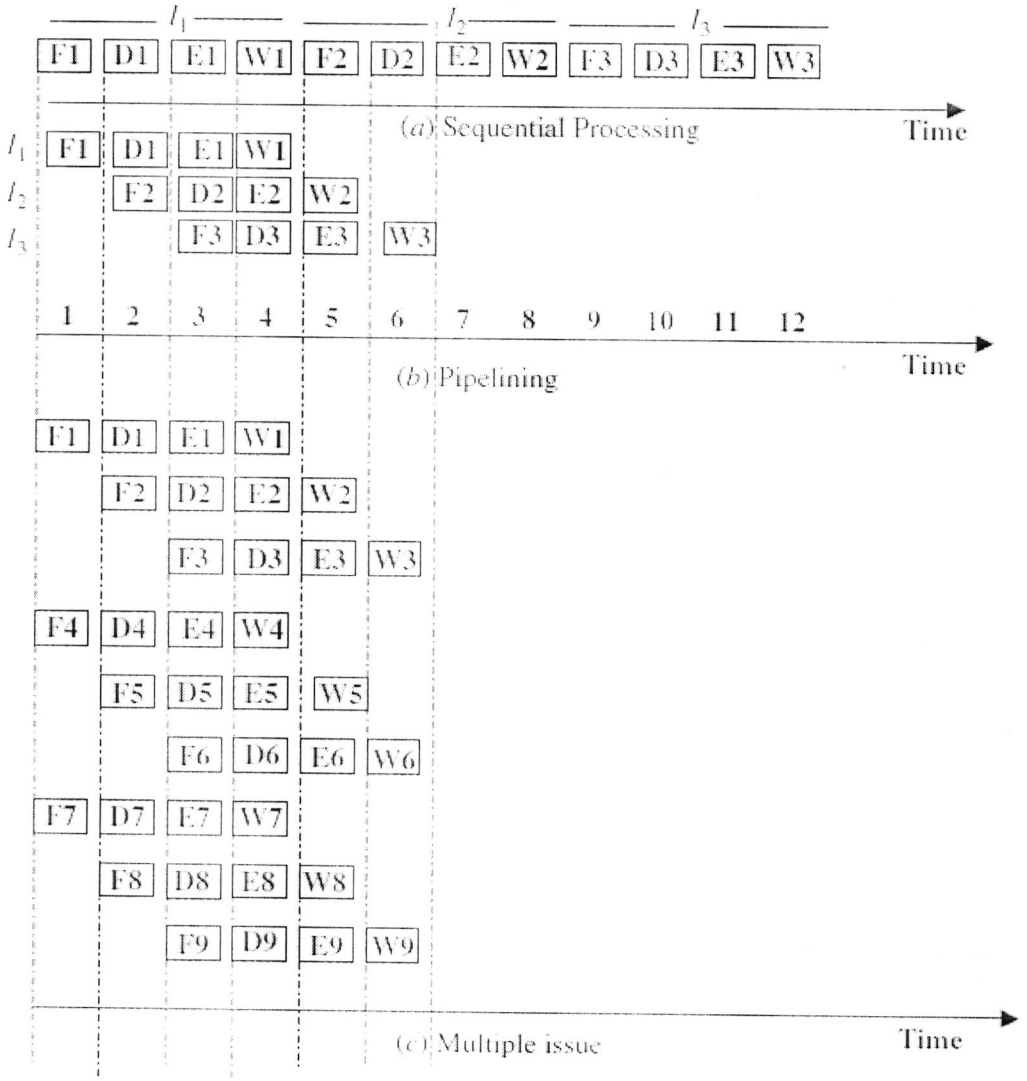


Figure (0): Multiple issue versus pipelining versus sequential processing

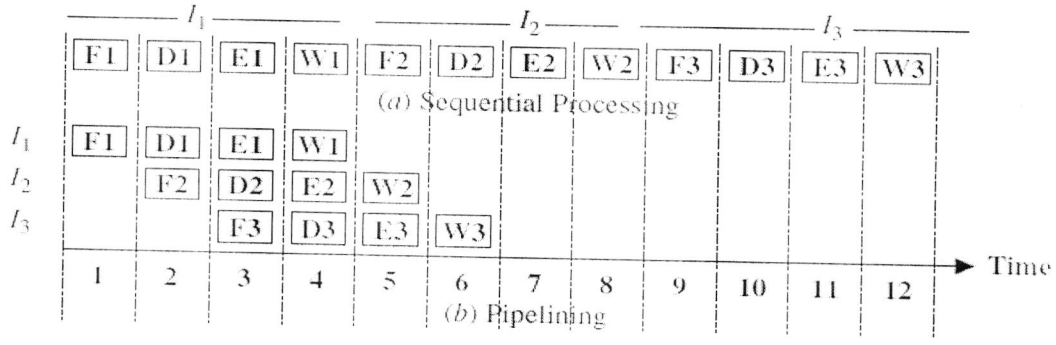
Pipelining owes its origin to car assembly lines. The idea is to have more than one instruction being processed by the processor at the same time. Similar to the assembly line, the success of a pipeline depends upon dividing the execution of an instruction among a number of subunits

(stages), each performing part of the required operations. A possible division is to consider instruction fetch (F), instruction decode (D), operand fetch (F), instruction execution (E), and store of results (S) as the subtasks needed for the execution of an instruction.

In this case, it is possible to have up to five instructions in the pipeline at the same time, thus reducing instruction execution latency.

1. General Concept

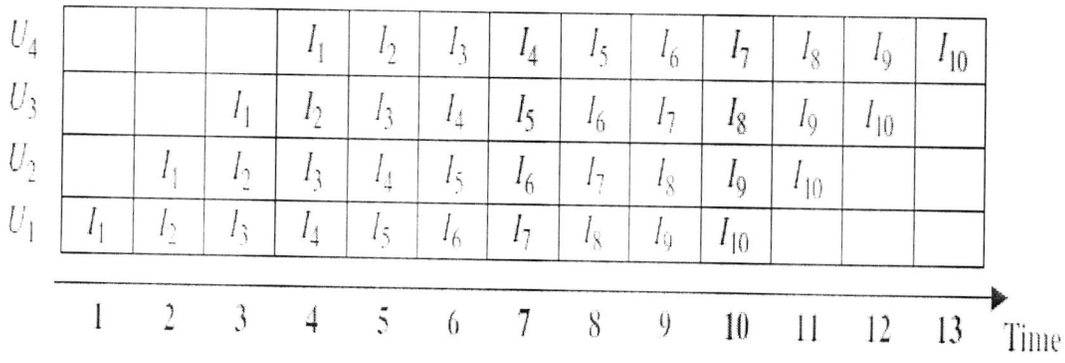
Pipelining refers to the technique in which a given task is divided into a number of subtasks that need to be performed in sequence. Each subtask is performed by a given functional unit. The units are connected in a serial fashion and all of them operate simultaneously. The use of pipelining improves the performance compared to the traditional sequential execution of tasks. Figure 1 shows an illustration of the basic difference between executing four subtasks of a given instruction (in this case fetching F, decoding D, execution E, and writing the results W) using pipelining and sequential processing.



Figure(1): Pipelining versus sequential processing.

It is clear from the figure that the total time required to process three instructions (I1, I2, I3) is only six time units if four-stage pipelining is used as compared to 12 time units if sequential processing is used. A possible saving of up to 50% in the execution time of these three instructions is obtained. In order to formulate some performance measures for the goodness of a pipeline in processing a series of tasks, a

space time chart (called the Gantt's chart) is used. The chart shows the succession of the subtasks in the pipe with respect to time. Figure (2) shows a Gantt's chart. In this chart, the vertical axis represents the subunits (four in this case) and the horizontal axis represents time (measured in terms of the time unit required for each unit to perform its task). In developing the Gantt's chart, we assume that the time (T) taken by each subunit to perform its task is the same; we call this the unit time. As can be seen from the figure, 13 time units are needed to finish executing instructions (I_1 to I_{10}). This is to be compared to 40 time units if sequential processing is used (ten instructions each requiring four time units).



Figure(2): The space–time chart (Gantt chart)

In the following analysis, we provide three performance measures for the goodness of a pipeline. These are the Speed-up S(n), Throughput U(n), and Efficiency E(n). It should be noted that in this analysis we assume that the unit time T= t units.

1. Speed-up S(n): Consider the execution of m tasks (instructions) using n-stages (units) pipeline. As can be seen, n + m -1 time units are required to complete m tasks.

$$Speed\text{-}up\ S(n) = \frac{\text{Time using sequential processing}}{\text{Time using pipeline processing}} = \frac{m \times n \times t}{(n+m-1) \times t}$$

$$= \frac{m \times n}{n+m-1}$$

2. Throughput U(n) is a number of tasks executed per unit time .

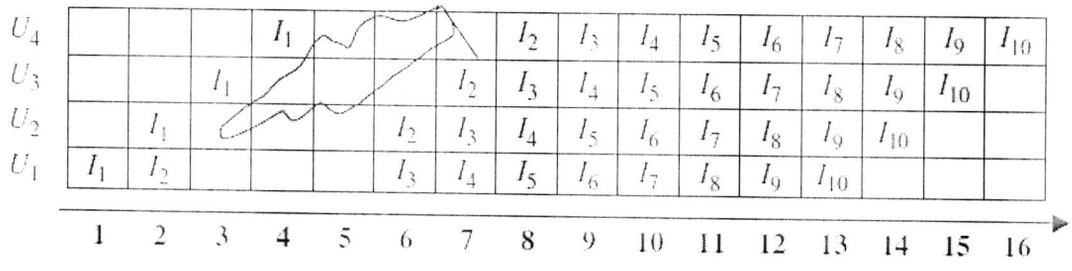
$$\text{Throughput U(n)} = \frac{m}{(n+m-1)\times t}$$

3. Efficiency E(n) : Ratio of the actual speed-up to the maximum speed-up.

$$\text{Efficiency E(n)} = \frac{Speed\text{-}up}{n} = \frac{m}{n+m-1}$$

2. Instruction Pipeline

The simple analysis made in Section 9.1 ignores an important aspect that can affect the performance of a pipeline, that is, pipeline stall. A pipeline operation is said to have been stalled if one unit (stage) requires more time to perform its function, thus forcing other stages to become idle. Consider, for example, the case of an instruction fetch that incurs a cache miss. Assume also that a cache miss requires three extra time units. Figure(3) illustrates the effect of having instruction I_2 incurring a cache miss (assuming the execution of ten instructions I_1 to I_{10}).



Figure(3): Effect of a cache miss on the pipeline

The figure shows that due to the extra time units needed for instruction I_2 to be fetched, the pipeline stalls, that is, fetching of instruction I_3 and subsequent instructions are delayed. Such situations create what is known as pipeline bubble (or pipeline hazards). The creation of a pipeline bubble

branch instruction is stored. Figure (4) shows that stall and the required Gantt's chart.

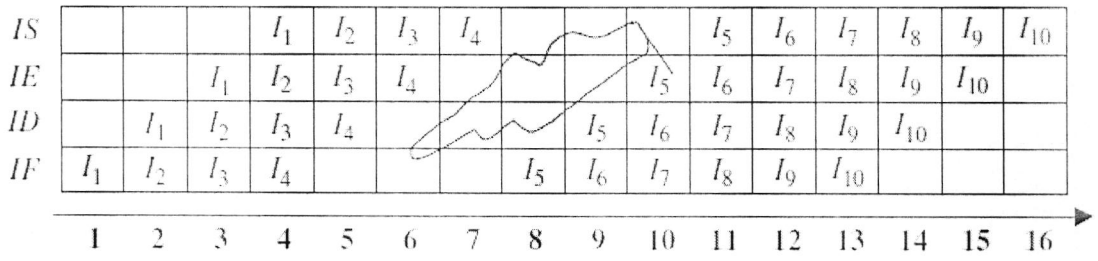


Figure (4): Instruction dependency effect on a pipeline

B:Data Dependency

Data dependency in a pipeline occurs when a source operand of instruction

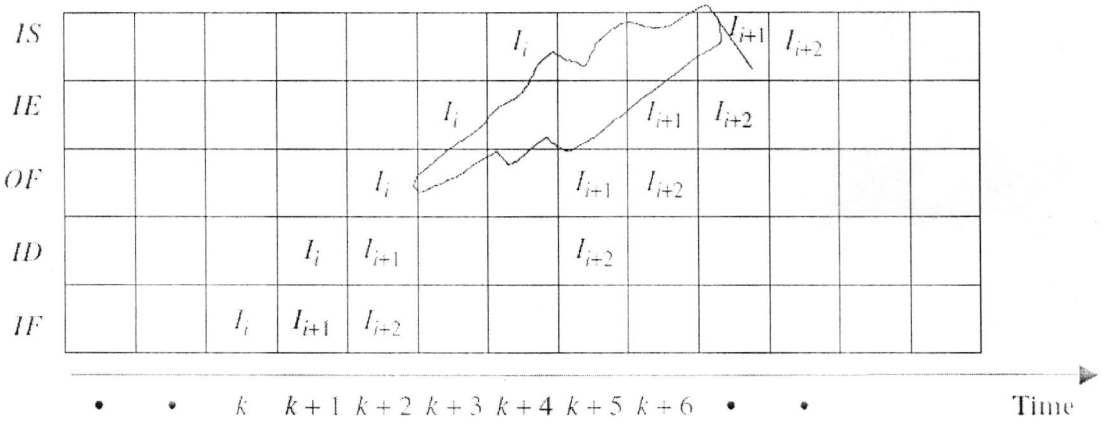
I_i depends on the results of executing a preceding instruction, I_j, $i > j$. It should be noted that although instruction I_i can be fetched, its operand(s) may not be available until the results of instruction Ij are stored. The following example shows the effect of data dependency on a pipeline.

Example 2: Consider the execution of the following piece of code:

ADD	R_1, R_2, R_3;	$R_3 \leftarrow R_1 + R_2$
SL	R_3;	$R_3 \leftarrow SL(R_3)$
SUB	R_5, R_6, R_4;	$R_4 \leftarrow R_5 - R_6$

In this piece of code, the first instruction, call it I_i, adds the contents of two registers R_1 and R_2 and stores the result in register R_3. The second instruction, call it I_{i+1} , shifts the contents of R_3 one bit position to the left and stores the result back into R_3. The third instruction, call it I_{i+2} , stores the result of subtracting the content of R_6 from the content of R_5 in register R_4. In order to show the effect of such data dependency, we will assume that the pipeline consists of five stages, IF, ID, OF, IE, and IS. In this case, the (OF) stage represents the operand fetch stage. The functions of the remaining four stages remain the same as explained before. As shown in the figure(5), although instruction I_{i+1}has been successfully

decoded during time unit k +2, this instruction cannot proceed to the OF unit during time unit k + 3. This is because the operand to be fetched by I_{i+1} during time unit k+3 should be the content of register R_3, which has been modified by execution of instruction I_i . However, the modified value of R_3 will not be available until the end of time unit k+4. This will require instruction I_{i+1} to wait (at the output of the ID unit) until k+5. Notice that instruction I_{i+2} will have also to wait (at the output of the IF unit) until such time that instruction I_{i+1} proceeds to the ID. The net result is that pipeline stall takes place due to the data dependency that exists between instruction Ii and instruction I_{i+1}.



Figure(5): The write-after-write data dependency

The data dependency presented in the above example resulted because register R_3 is the destination for both instructions I_i and I_{i+1}. This is called a write-after-write data dependency. Taking into consideration that any register can be written into (or read from), then a total of four different possibilities exist, including the:

- write-after-write.
- read-after-write .
- write-after-read.
- read-after-read.

Among the four cases, the read-after-read case should not lead to pipeline stall. This is because a register read operation does not change the content of the register. Among the remaining three cases, the write-after-write (see the above example) and the read-after-write lead to pipeline stall. The following

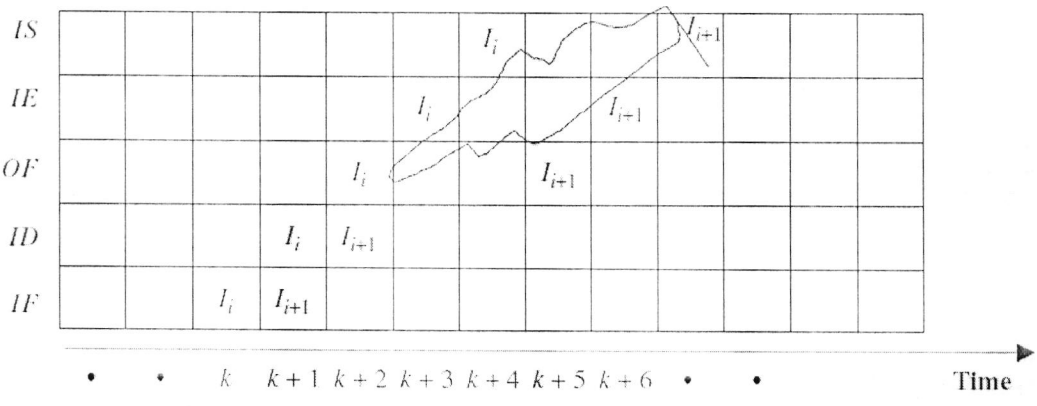
piece of code illustrates the read-after-write case:

$$ADD \quad R_1, R_2, R_3; \qquad R_3 \leftarrow R_1 + R_2$$
$$SUB \quad R_3, 1, R_4; \qquad R_4 \leftarrow R_3 - 1$$

In this case, the first instruction modifies the content of register R_3 (through a write operation) while the second instruction uses the modified contents of R_3 (through a read operation) to load a value into register R_4. While these two instructions are proceeding within a pipeline, care should be taken so that the value of register R_3 read in the second instruction is the updated value resulting from execution of the previous instruction see figure(6).

In this case assuming that the first instruction is called Ii and the second instruction is called I_{i+1}.

It is clear that the operand of the second instruction cannot be fetched during time unit k+3 and that it has to be delayed until time unit k+5. This is because the modified value of the content of register R_3 will not be available until time slot k+5.



Figure(6) The read-after-write data dependency

Fetching the operand of the second instruction during time slot k+3 will lead to incorrect results.

Example 3: Consider the execution of the following sequence of instructions on a five-stage pipeline consisting of IF, ID, OF, IE, and IS. It is required to show the succession of these instructions in the pipeline.

$I_1 \rightarrow$ Load $-1, R1;$ $R1 \leftarrow -1;$

$I_2 \rightarrow$ Load $5, R2;$ $R2 \leftarrow 5;$

$I_3 \rightarrow$ Sub $R2, 1, R2$ $R2 \leftarrow R2 - 1;$

$I_4 \rightarrow$ Add $R1, R2, R3;$ $R3 \leftarrow R1 + R2;$

$I_5 \rightarrow$ Add $R4, R5, R6;$ $R6 \leftarrow R4 + R5;$

$I_6 \rightarrow SL$ $R3$ $R3 \leftarrow SL\ (R3)$

$I_7 \rightarrow$ Add $R6, R4, R7;$ $R7 \leftarrow R4 + R6;$

In this example, the following data dependencies are observed:

Instructions	Type of data dependency
I_3 and I_2	Read-after-write and write-after-write (W-W)
I_4 and I_1	Read-after-write (R-W)
I_4 and I_3	Read-after-write (R-W)
I_6 and I_4	Read-after-write and write-after-write (W-W)
I_7 and I_5	Read-after-write (R-W)

Figure(7) illustrates the progression of these instructions in the pipeline taking into consideration the data dependencies mentioned above. The assumption made in constructing the Gantt's chart in Figure(7) is that fetching an operand by an instruction that depends on the results of a previous instruction execution is delayed until such operand is available, that is, the result is stored. A total of 16 time units are required to execute

the given seven instructions taking into consideration the data dependencies among the different instructions.

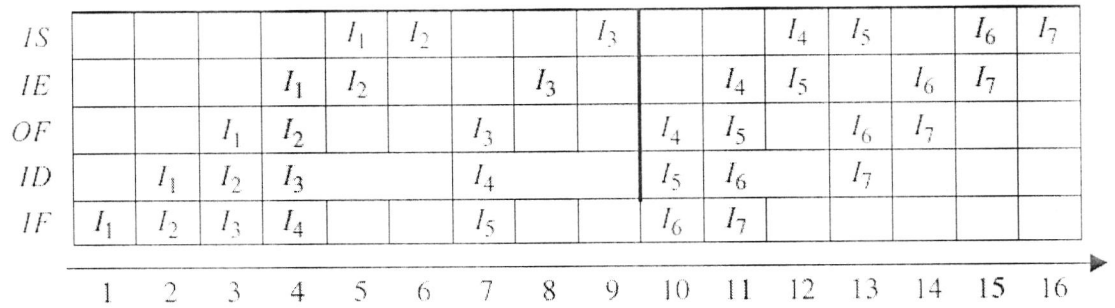


Figure (7) Gantt's chart for Example 3

Based on the results obtained above, we can compute the speed-up and the throughput for executing the piece of code given in Example 3 as:

$$Speed\text{-}up\ S(5) = \frac{\text{Time using sequential processing}}{\text{Time using pipeline processing}} = \frac{7 \times 5}{16} = 2.19$$

$$Throughput\ U(5) = \text{No. of tasks executed per unit time} = \frac{7}{16} = 0.44$$

The discussion on pipeline stall due to instruction and data dependencies should reveal three main points about the problems associated with having such dependencies. These are:

1. Both instruction and data dependencies lead to added delay in the pipeline.
2. Instruction dependency can lead to the fetching of the wrong instruction.
3. Data dependency can lead to the fetching of the wrong operand.

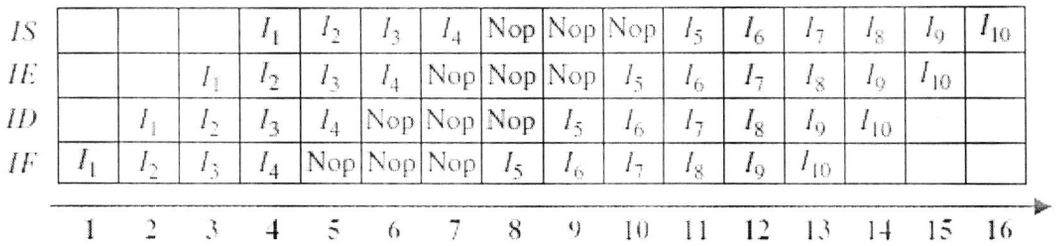
There exist a number of methods to deal with the problems resulting from instruction and data dependencies. Some of these methods try to prevent the fetching of the wrong instruction or the wrong operand while others try to reduce the delay incurred in the pipeline due to the existence of instruction or data dependency. A number of these methods are introduced below.

3-Methods Used to Prevent Fetching the Wrong Instruction or Operand

Use of NOP (No Operation):

This method can be used in order to prevent the fetching of the wrong instruction, in case of instruction dependency, or fetching the wrong operand, in case of data dependency. Recall Example 1. In that example, the execution of a sequence of ten instructions I_1–I_{10} on a pipeline consisting of four pipeline stages: IF, ID, IE, and IS were considered. In order to show the execution of these instructions in the pipeline, we have assumed that when the branch instruction is fetched, the pipeline stalls until the result of executing the branch instruction is stored. This assumption was needed in order to prevent fetching the wrong instruction after fetching the branch instruction. In real-life situations, a mechanism is needed to guarantee fetching the appropriate instruction at the appropriate time. Insertion of "NOP" instructions will help carrying out this task. A "NOP" is an instruction that has no effect on the status of the processor.

Example4: Consider the execution of ten instructions I_1–I_{10} on a pipeline consisting of four pipeline stages: IF, ID, IE, and IS. Assume that instruction I_4 is a conditional branch instruction and that when it is executed, the branch is not taken; that is, the branch condition is not satisfied.



Figure(8) The use of NOP instructions

In order to execute this set of instructions while preventing the fetching of the wrong instruction, we assume that a specified number of NOP instructions have been inserted such that they follow instruction I_4 in the sequence and they precede instruction I_5. Figure(8) shows the Gantt's chart illustrating the execution of the new sequence of instructions (after inserting the NOP instructions). The figure shows that the insertion of THREE NOP instructions after instruction I_4 will guarantee that the correct instruction to fetch after I_4, in this case I_5, will only be fetched during time slot number 8 at which the result of executing I_4 would have been stored and the condition for the branch would have been known.

It should be noted that the number of NOP instructions needed is equal to (n-1), where n is the number of pipeline stages.

Note: the use of NOP instructions to prevent fetching the wrong instruction in the case of instruction dependency. A similar approach can be used to prevent fetching the wrong operand in the case of data dependency. Consider the execution of the following piece of code on a five-stage pipeline (IF, ID, OF, IE, IS).

ADD	R_1, R_2, R_3;	$R_3 \leftarrow R_1 + R_2$
SUB	R_3, 1, R_4;	$R_4 \leftarrow R_3 - 1$
MOV	R_5, R_6;	$R_6 \leftarrow R_5$

Note the data dependency in the form of read-after-write (R-W) between the first two instructions. Fetching the operand for the second instruction, that is, fetching the content of R_3, cannot proceed until the result of the