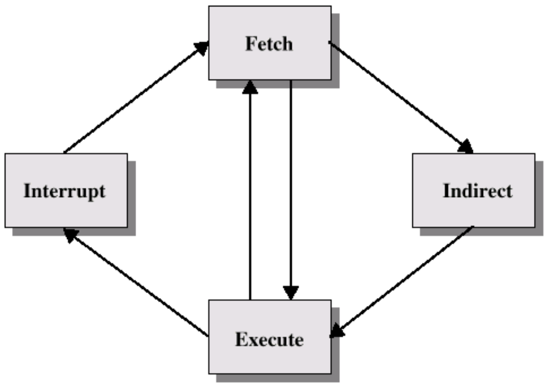
Instruction Cycle and Instruction Pipelining

An instruction cycle includes the following stages:

- Fetch: Read the next instruction from memory into the processor.
- Execute: Interpret the opcode and perform the indicated operation.
- Interrupt: If interrupts are enabled and an interrupt has occurred, save the current process state and service the interrupt.

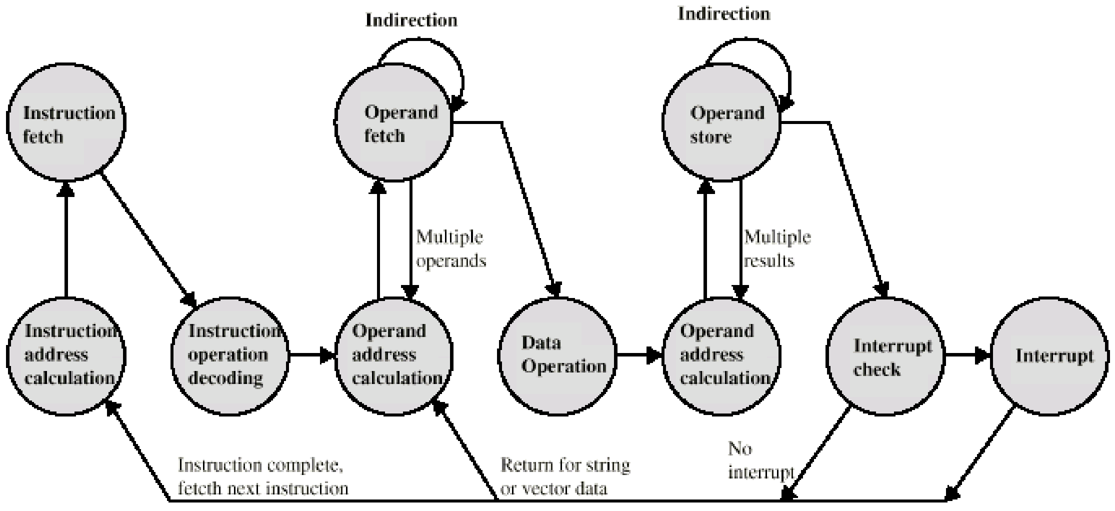
There is also one additional stage, known as the indirect cycle.



The Indirect Cycle

The execution of an instruction may involve one or more operands in memory, each of which requires a memory access. If indirect addressing is used, then additional memory accesses are required. We can think of the fetching of indirect addresses as one more instruction stages. The main line of activity consists of alternating instruction fetch and instruction execution activities. After an instruction is fetched, it is examined to determine if any indirect addressing is involved. If so, the required operands are fetched using indirect addressing.

Following execution, an interrupt may be processed before the next instruction fetch.


Indirection
Indirection
Instruction fetch
Operand fetch
Operand store
Multiple operands
Multiple results
Instruction address calculation
Instruction operation decoding
Operand address calculation
Data Operation
Operand address calculation
Interrupt check
Interrupt
Instruction complete, fetch next instruction
Return for string or vector data
No interrupt

Instruction Pipelining

Greater performance can be achieved by taking advantage of improvements in technology and organizational enhancement, an organizational approach, which is quite common, is instruction pipelining.

In instruction pipelining, **instruction execution cycle** is perceived as being divided into a number of stages, where new instruction are accepted at one end before previously accepted instruction completes itself and appears as outputs at the other end.

As a simple approach, consider subdividing instruction processing into two stages: fetch instruction and execute instruction. There are times during the execution of an instruction when main memory is not being accessed. This time could be used to fetch the next instruction in parallel with the execution of the current one.

This pipeline has two independent stages. The first stage fetches an instruction and buffers it. When the second stage is free, the first stage passes it the buffered instruction. While the second stage is executing the instruction, the first stage takes advantage of any unused memory cycles to fetch and buffer the next instruction. This is called instruction *prefetch* or *fetch overlap*.

Pipelining requires registers to store data between stages.

This process will speed up instruction execution. If the fetch and execute stages were of equal duration, the instruction cycle time would be halved. However, this doubling of execution rate is unlikely for two reasons:

1. The execution time will generally be longer than the fetch time. Execution will involve reading and storing operands and the performance of some operation. Thus, the fetch stage may have to wait for some time before it can empty its buffer.

2. A conditional branch instruction makes the address of the next instruction to be fetched unknown. Thus, the fetch stage must wait until it receives the next instruction address from the execute stage. The execute stage may then have to wait while the next instruction is fetched.

Guessing can reduce the time loss from the 2nd reason.
A simple rule is the following: When a conditional branch instruction is passed on from the fetch to the execute stage, the fetch stage fetches the next instruction in memory after the branch instruction. Then, if the branch is not taken, no time is lost.
If the branch is taken, the fetched instruction must be discarded and a new instruction fetched.

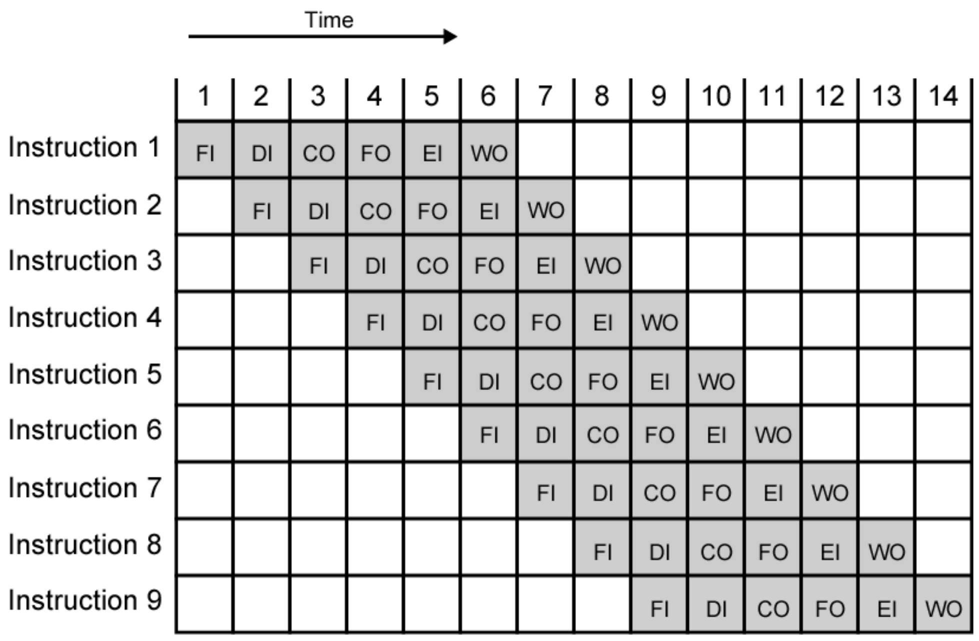
To gain further speedup, the pipeline must have more stages. A suggestive decomposition of the instruction processing:

- Fetch instruction (FI): Read the next expected instruction into a buffer.
- Decode instruction (DI): Determine the opcode and the operand specifiers.
- Calculate operands (CO): Calculate the effective address of each source operand. This may involve displacement, register indirect, indirect, or other forms of address calculation.
- Fetch operands (FO): Fetch each operand from memory. Operands in registers need not be fetched.

• Execute instruction (EI): Perform the indicated operation and store the result, if any, in the specified destination operand location.
• Write operand (WO): Store the result in memory.

With this decomposition, the various stages will be of more nearly equal duration. Using this assumption, a six-stage pipeline can reduce the execution time for 9 instructions from 54 time units to 14 time units.

Using this assumption, we can see that a six-stage pipeline can reduce the execution time for 9 instructions from 54 time units to 14 time units.



Time

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Instruction 1	FI	DI	CO	FO	EI	WO								
Instruction 2		FI	DI	CO	FO	EI	WO							
Instruction 3			FI	DI	CO	FO	EI	WO						
Instruction 4				FI	DI	CO	FO	EI	WO					
Instruction 5					FI	DI	CO	FO	EI	WO				
Instruction 6						FI	DI	CO	FO	EI	WO			
Instruction 7							FI	DI	CO	FO	EI	WO		
Instruction 8								FI	DI	CO	FO	EI	WO	
Instruction 9									FI	DI	CO	FO	EI	WO

Assumptions:
1) We assumes that each instruction goes through all six stages of the pipeline. This will not always be the case.

For example, a load instruction does not need the Write Operand WO stage. However, to simplify the pipeline hardware, the timing is set up assuming that each instruction requires all six stages.

2) We assumes that all of the stages can be performed in parallel and there are no memory conflicts.

For example, the FI, FO, and WO stages involve a memory access. We assume that all these accesses can occur simultaneously. Most memory systems will not permit that. However, the desired value may be in cache, or the FO or WO stage may be null. Thus, much of the time, memory conflicts will not slow down the pipeline.

Several other **factors serve to limit the performance enhancement**.
If all the stages are not of equal duration, there will be some waiting involved at various pipeline stages.
In case of a conditional branch instruction, several instruction fetches can be invalidated.
An interrupt can also invalidate several instruction fetches.

Other problems arise that did not appear in our simple two-stage organization. The CO stage may depend on the contents of a register that could be altered by a previous instruction that is still in the pipeline. Other such register and memory conflicts could occur. The system must contain logic to account for this type of conflict.

It appears that the greater the number of stages in the pipeline, the faster the execution rate.

Instruction pipelining is a powerful technique for enhancing performance but requires careful design to achieve optimum results with reasonable complexity.

PIPELINE HAZARD

A pipeline hazard occurs when the pipeline, or some portion of the pipeline, must stall because conditions do not permit continued execution. Such a pipeline stall is also referred to as a *pipeline bubble*. There are three types of hazards: Resource, Data, and Control.

RESOURCE HAZARDS:
A resource hazard occurs when two (or more) instructions that are already in the pipeline need the same resource. The result is that the instructions must be executed in serial rather than parallel for a portion of the pipeline. A resource hazard is sometime referred to as a **structural hazard**.

Assume a simplified five-stage pipeline, in which each stage takes one clock cycle. Now assume that main memory has a single port and that all instruction fetches and data reads and writes must be performed one at a time. Further, ignore the cache. In this case, an operand read to or write from memory cannot be performed in parallel with an instruction fetch. Therefore, the fetch instruction stage of the pipeline must idle for one cycle before beginning the instruction fetch for instruction I3.

Another example of a resource conflict is a situation in which multiple instructions are ready to enter the execute instruction phase and there is a single ALU.

One solutions to such resource hazards is to increase available resources, such as having multiple ports into main memory and multiple ALU units.

DATA HAZARDS:
A data hazard occurs when there is a conflict in the access of an operand location. In general terms, we can state the hazard in this form: Two instructions in a program are to be executed in sequence and both access a particular memory or register operand. If the two instructions are executed in strict sequence, no problem occurs. However, if the instructions are executed in a pipeline, then it is possible for the operand value to be updated in such a way as to produce a different result than would occur with strict sequential execution. In other words, the program produces an incorrect result because of the use of pipelining.

There are three types of data hazards;

• Read after write (RAW), or true dependency: An instruction modifies a register or memory location and a succeeding instruction reads the data in that memory or register location. A hazard occurs if the read takes place before the write operation is complete.

• Write after read (WAR), or antidependency: An instruction reads a register or memory location and a succeeding instruction writes to the location. A hazard occurs if the write operation completes before the read operation takes place.

• Write after write (WAW), or output dependency: Two instructions both write to the same location. A hazard occurs if the write operations take place in the reverse order of the intended sequence.

CONTROL HAZARDS:

A control hazard, also known as a branch hazard, occurs when the pipeline makes the wrong decision on a branch prediction and therefore brings instructions into the pipeline that must subsequently be discarded.