Instruction Pipeline

“Pipeline processing can occur not only in the data stream but in the instruction stream as well”

Instruction Pipeline

- An instruction pipeline reads consecutive instructions from memory while previous instructions are being executed in other segments.
- This causes the instruction fetch and execute phases to overlap and perform simultaneous operations.

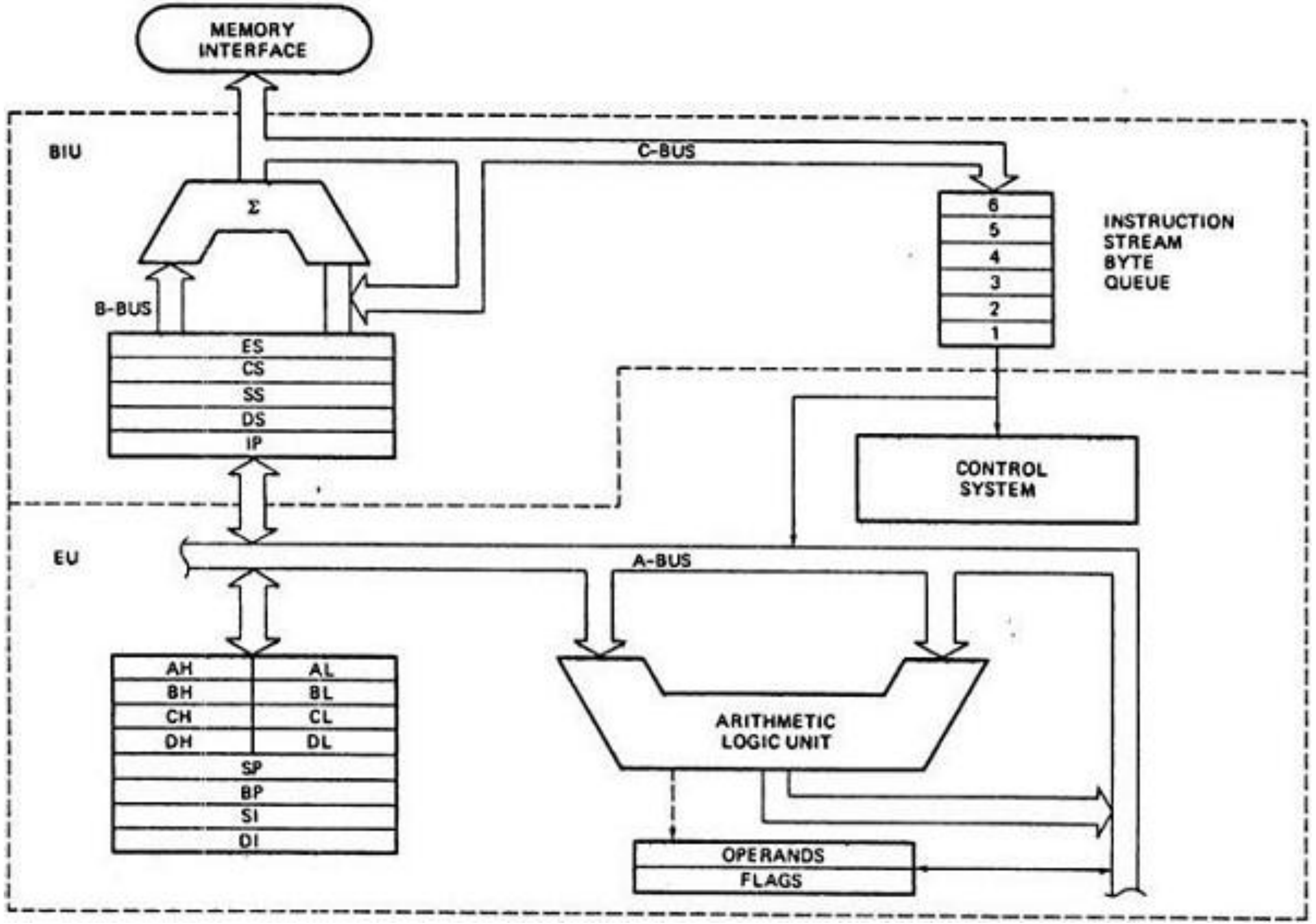
- An instruction may produce a branch out of sequence, which is one conceivable deviation in such a design.
- **In that case the pipeline must be emptied and all the instructions that have been read from memory after the branch instruction must be discarded.**

Two-Segment Pipeline

- Consider a computer with an **instruction fetch unit** and **an instruction execution unit** designed to provide a two-segment pipeline.
- The instruction fetch segment can be implemented by means of a first-in, first-out (FIFO) buffer- **“Queue”**

Two-Segment Pipeline - Explanation

- Whenever the execution unit is not using memory, the control increments the program counter and uses its address value to read consecutive instructions from memory.
- The instructions are inserted into the FIFO buffer so that they can be executed on a first-in, first-out basis.
- Thus an instruction stream can be placed in a queue, waiting for decoding and processing by the execution segment.
- The instruction stream queuing mechanism provides an efficient way for **reducing the average access time to memory for reading instructions.**
- Whenever there is space in the FIFO buffer, the control unit initiates the next instruction fetch phase.
- The buffer acts as a queue from which control then extracts the instructions for the execution unit.


MEMORY INTERFACE
BIU
C-BUS
Σ
B-BUS
ES
CS
SS
DS
IP
6
5
4
3
2
1
INSTRUCTION STREAM BYTE QUEUE
CONTROL SYSTEM
EU
A-BUS
AH
AL
BH
BL
CH
CL
DH
DL
SP
BP
SI
DI
ARITHMETIC LOGIC UNIT
OPERANDS
FLAGS

General Scenario: “Instruction Cycle”

- Computers with complex instructions require other phases in addition to the fetch and execute to process an instruction completely.
- In the most general case, the computer needs to process each instruction with the following sequence of steps.

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

Difficulties that will prevent the instruction pipeline from operating at its maximum rate

1. Different segments may take different times to operate on the incoming information.
2. Some segments are skipped for certain operations.

   For example, a register mode instruction does not need an effective address calculation.
3. Two or more segments may require memory access at the same time, causing one segment to wait until another is finished with the memory.

One Possible Solution

- Memory access conflicts are sometimes resolved by using two memory buses for accessing instructions and data in separate modules.
- In this way, an instruction word and a data word can be read simultaneously from two different modules.

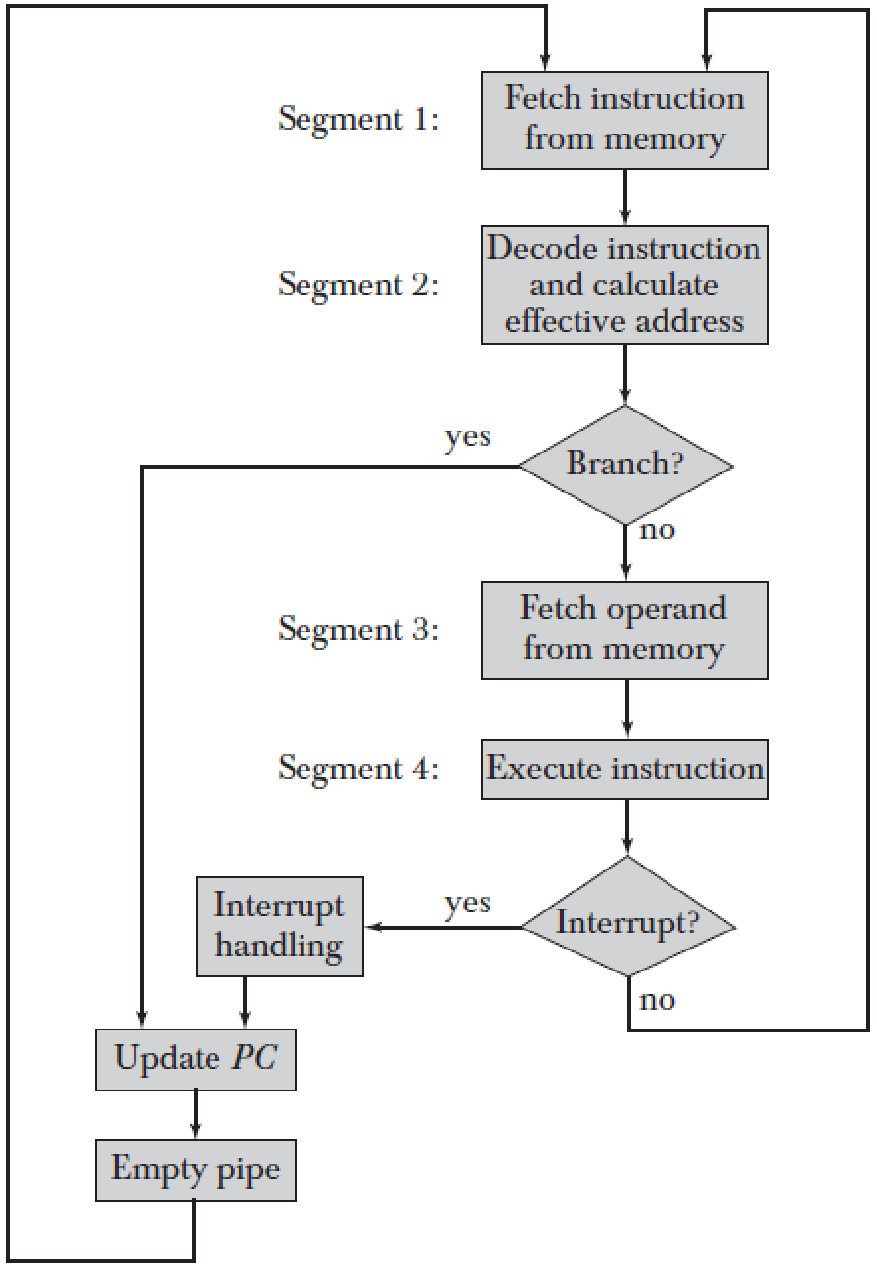
Note

- The design of an instruction pipeline will be most efficient if the instruction cycle is divided into segments of equal duration.
- The time that each step takes to fulfill its function depends on the instruction and the way it is executed.

Example: Four-Segment Instruction Pipeline

1. Fetch the instruction from memory
2. Decode the instruction and Calculate the effective address.
3. Fetch the operands from memory.
4. Execute the instruction and Store the result in the proper place. (Assuming most instructions place result into a processor register)

How the instruction cycle in the CPU can be processed with a four-segment pipeline?



Four suboperations in the instruction cycle can overlap and up to four different instructions can be in progress of being processed at the same time.

Once in a while, an instruction in the sequence may be a program control type that causes a branch out of normal sequence.

In that case the pending operations in the last two segments are completed and all information stored in the instruction buffer is deleted.

The pipeline then restarts from the new address stored in the program counter.

Similarly, an interrupt request, when acknowledged, will cause the pipeline to empty and start again from a new address value.

1. FI is the segment that fetches an instruction.
2. DA is the segment that decodes the instruction and calculates the effective address.
3. FO is the segment that fetches the operand.
4. EX is the segment that executes the instruction.

Note:

It is assumed that the processor has separate instruction and data memories so that the operation in FI and FO can proceed at the same time.

Timing of instruction pipeline

Step:		1	2	3	4	5	6	7	8	9	10	11	12	13
Instruction:	1	FI	DA	FO	EX									
	2		FI	DA	FO	EX								
(Branch)	3			FI	DA	FO	EX							
	4				FI	–	–	FI	DA	FO	EX			
	5					–	–	–	FI	DA	FO	EX		
	6									FI	DA	FO	EX	
	7										FI	DA	FO	EX

Explanation

- Assume now that instruction 3 is a branch instruction.
- As soon as this instruction is decoded in segment DA in step 4, the transfer from FI to DA of the other instructions is halted until the branch instruction is executed in step 6.
- If the branch is taken, a new instruction is fetched in step 7.
- If the branch is not taken, the instruction fetched previously in step 4 can be used.
- The pipeline then continues until a new branch instruction is encountered.

Note

- Another delay may occur in the pipeline if the EX segment needs to store the result of the operation in the data memory while the FO segment needs to fetch an operand.
- In that case, segment FO must wait until segment EX has finished its operation.

Pipeline Conflicts

- In general, there are three major difficulties that cause the instruction pipeline to deviate from its normal operation.

1. ***Resource conflicts*** caused by access to memory by two segments at the same time.

   Most of these conflicts can be resolved by using separate instruction and data memories.

2. ***Data dependency*** conflicts arise when an instruction depends on the result of a previous instruction, but this result is not yet available.

3. ***Branch difficulties*** arise from branch and other instructions that change the value of PC.

Data Dependency

- A difficulty that may caused a degradation of performance in an instruction pipeline is due to possible collision of data or address.
- A collision occurs when an instruction cannot proceed because previous instructions did not complete certain operations.
- A data dependency occurs when an instruction needs data that are not yet available.
- For example, an instruction in the FO segment may need to fetch an operand that is being generated at the same time by the previous instruction in segment EX.
- Therefore, the second instruction must wait for data to become available by the first instruction.

Similarly, an address dependency may occur when an operand address cannot be calculated because the information needed by the addressing mode is not available.
For example, an instruction with register indirect mode cannot proceed to fetch the operand if the previous instruction is loading the address into the register.

Therefore, the operand access to memory must be delayed until the required address is available.

Pipelined computers deal with such conflicts between data dependencies in a variety of ways.

1. Hardware interlocks
2. Operand forwarding
3. Delayed load

Hardware interlocks

- The most straightforward method is to insert hardware interlocks.
- An interlock is a circuit that detects instructions whose source operands are destinations of instructions farther up in the pipeline.
- Detection of this situation causes the instruction whose source is not available to be delayed by enough clock cycles to resolve the conflict.
- This approach maintains the program sequence by using hardware to insert the required delays.

Operand forwarding

- Operand forwarding uses special hardware to detect a conflict and then avoid it by routing the data through special paths between pipeline segments.
- For example, instead of transferring an ALU result into a destination register, the hardware checks the destination operand, and if it is needed as a source in the next instruction, it passes the result directly into the ALU input, bypassing the register file.
- This method requires additional hardware paths through multiplexers as well as the circuit that detects the conflict.

Delayed load

- A procedure employed in some computers is to give the responsibility for solving data conflicts problems to the compiler that translates the high level programming language into a machine language program.
- The compiler for such computers is designed to detect a data conflict and reorder the instructions as necessary to delay the loading of the conflicting data by inserting no-operation instructions.

Handling of Branch Instructions

- The branch instruction breaks the normal sequence of the instruction stream, causing difficulties in the operation of the instruction pipeline.
- A branch instruction can be conditional or unconditional.
- An unconditional branch always alters the sequential program flow by loading the program counter with the target address.
- In a conditional branch, the control selects the target instruction if the condition is satisfied or the next sequential instruction if the condition is not satisfied.

Pipelined computers employ various hardware techniques to minimize the performance degradation caused by instruction branching.

1. Prefetch target instruction
2. Branch target buffer
3. Loop buffer
4. Branch prediction
5. Delayed branch

Prefetch target instruction

- One way of handling a conditional branch is to prefetch the target instruction in addition to the instruction following the branch.
- Both are saved until the branch is executed.
- If the branch condition is successful, the pipeline continues from the branch target instruction.
- An extension of this procedure is to continue fetching instructions from both places until the branch decision is made.
- At that time control chooses the instruction stream of the correct program flow.

Branch Target Buffer (BTB)

- The BTB is an associative memory included in the fetch segment of the pipeline.
- Each entry in the BTB consists of the address of a previously executed branch instruction and the target instruction for that branch.
- It also stores the next few instructions after the branch target instruction.
- When the pipeline decodes a branch instruction, it searches the associative memory BTB for the address of the instruction.
- If it is in the BTB, the instruction is available directly and prefetch continues from the new path.
- If the instruction is not in the BTB, the pipeline shifts to a new instruction stream and stores the target instruction in the BTB.
- The advantage of this scheme is that branch instructions that have occurred previously are readily available in the pipeline without interruption.

Loop buffer

- A variation of the BTB is the loop buffer.
- This is a small very high speed register file maintained by the instruction fetch segment of the pipeline.
- When a program loop is detected in the program, it is stored in the loop buffer in its entirety, including all branches.
- The program loop can be executed directly without having to access memory until the loop mode is removed by the final branching out.

Branch Prediction

- A pipeline with branch prediction uses some additional logic to guess the outcome of a conditional branch instruction before it is executed.
- The pipeline then begins prefetching the instruction stream from the predicted path.
- A correct prediction eliminates the wasted time caused by branch penalties.

Delayed Branch

- A procedure employed in most RISC processors is the delayed branch.
- In this procedure, the compiler detects the branch instructions and rearranges the machine language code sequence by inserting useful instructions that keep the pipeline operating without interruptions.
- An example of delayed branch is the insertion of a no-operation instruction after a branch instruction.
- This causes the computer to fetch the target instruction during the execution of the no-operation instruction, allowing a continuous flow of the pipeline.

RISC Pipeline

RISC

- Among the characteristics attributed to RISC is its **ability to use an efficient instruction pipeline.**
- The **simplicity of the instruction set** can be utilized to implement an instruction pipeline using a small number of suboperations, with each being executed in one clock cycle.
- Because of the **fixed-length instruction format**, the decoding of the operation can occur at the same time as the register selection.
- All data manipulation instructions have register-to-register operations.
- Since all operands are in registers, there is no need for calculating an effective address or fetching of operands from memory.

RISC

- Therefore, **the instruction pipeline can be implemented with two or three segments**.
- One segment fetches the instruction from program memory, and
- the other segment executes the instruction in the ALU.
- A third segment may be used to store the result of the ALU operation in a destination register.

RISC “Two Separate Memories for Data and Instructions”

- The data transfer instructions in RISC are limited to load and store instructions.
- These instructions use register indirect addressing.
- To prevent conflicts between a memory access to fetch an instruction and to load or store an operand, most RISC machines use two separate buses with two memories: one for storing the instructions and the other for storing the data.
- The two memories can sometime operate at the same speed as the CPU clock and are referred to as cache memories

RISC- “single-cycle instruction execution”

- One of the major advantages of RISC is its ability to execute instructions at the rate of one per clock cycle.
- It is not possible to expect that every instruction be fetched from memory and executed in one clock cycle.
- What is done, in effect, is to start each instruction with each clock cycle and to pipeline the processor to achieve the goal of single-cycle instruction execution.
- The advantage of RISC over CISC (complex instruction set computer) is that **RISC can achieve pipeline segments, requiring just one clock cycle, while CISC uses many segments in its pipeline**, with the longest segment requiring two or more clock cycles.

RISC- “compiler support”

- Another characteristic of RISC is the support given by the compiler that translates the high-level language program into machine language program.
- **Instead of designing hardware to handle** the difficulties associated with data conflicts and branch penalties, RISC processors rely on the efficiency of the compiler to detect and minimize the delays encountered with these problems.
- The following examples show how a compiler can optimize the machine language program to compensate for pipeline conflicts.

Example: Three-Segment Instruction Pipeline

Three types of RISC instructions

1. **The data manipulation instructions** operate on data in processor registers.
2. **The data transfer instructions** are load and store instructions that use an effective address obtained from the addition of the contents of two registers or a register and a displacement constant provided in the instruction.
3. **The program control instructions** use register values and a constant to evaluate the branch address, which is transferred to a register or the program counter PC.

Considering the hardware operation for such a computer

- The control section fetches the instruction from program memory into an instruction register.
- The instruction is decoded at the same time that the registers needed for the execution of the instruction are selected.
- The processor unit consists of a number of registers and an arithmetic logic unit (ALU) that performs the necessary arithmetic, logic, and shift operations.
- A data memory is used to load or store the data from a selected register in the register file.

RISC: “Instruction Cycle”

- The instruction cycle can be divided into three suboperations and implemented in three segments:

1. I: Instruction fetch
2. A: ALU operation
3. E: Execute instruction

I: Instruction fetch

- The I segment fetches the instruction from program memory.

A: ALU operation

- The instruction is decoded and an ALU operation is performed in the A segment.
- The ALU is used for three different functions, depending on the decoded instruction.
- It performs an operation for a data manipulation instruction, it evaluates the effective address for a load or store instruction, or it calculates the branch address for a program control instruction.

E: Execute instruction

- The E segment directs the output of the ALU to one of three destinations, depending on the decoded instruction.
- It transfers the result of the ALU operation into a destination register in the register file, it transfers the effective address to a data memory for loading or storing, or it transfers the branch address to the program counter.

Delayed Load

- Consider now the operation of the following four instructions:

1. LOAD: $R1 \leftarrow M[\text{address } 1]$
2. LOAD: $R2 \leftarrow M[\text{address } 2]$
3. ADD: $R3 \leftarrow R1 + R2$
4. STORE: $M[\text{address } 3] \leftarrow R3$

Clock cycles:	1	2	3	4	5	6
1. Load $R1$	I	A	E			
2. Load $R2$		I	A	E		
3. Add $R1 + R2$			I	A	E	
4. Store $R3$				I	A	E

(a) Pipeline timing with data conflict

Clock cycles:	1	2	3	4	5	6
1. Load *R*1	I	A	E			
2. Load *R*2		I	A	E		
3. Add *R*1 + *R*2			I	A	E	
4. Store *R*3				I	A	E

(a) Pipeline timing with data conflict

If the three-segment pipeline proceeds without interruptions, there will be a data conflict in instruction 3 because the operand in *R*2 is not yet available in the A segment.

The E segment in clock cycle 4 is in a process of placing the memory data into *R*2.

The A segment in clock cycle 4 is using the data from *R*2, but the value in *R*2 will not be the correct value since it has not yet been transferred from memory.

It is up to the compiler to make sure that the instruction following the load instruction uses the data fetched from memory.

If the compiler cannot find a useful instruction to put after the load, it inserts a no-op (no-operation) instruction.

This is a type of instruction that is fetched from memory but has no operation, thus wasting a clock cycle.

This concept of delaying the use of the data loaded from memory is referred to as **delayed load.**

Clock cycle:	1	2	3	4	5	6	7
1. Load *R*1	I	A	E				
2. Load *R*2		I	A	E			
3. No-operation			I	A	E		
4. Add *R*1 + *R*2				I	A	E	
5. Store *R*3					I	A	E

(b) Pipeline timing with delayed load

The data is loaded into *R*2 in clock cycle 4.

The add instruction uses the value of *R*2 in step 5. Thus the no-op instruction is used to advance one clock cycle in order to compensate for the data conflict in the pipeline.

(Note that no operation is performed in segment A during clock cycle 4 or segment E during clock cycle 5.)

The advantage of the delayed load approach is that the data dependency is taken care of by the compiler rather than the hardware.

This results in a simpler hardware segment since the segment does not have to check if the content of the register being accessed is currently valid or not.

Delayed Branch

- A branch instruction delays the pipeline operation until the instruction at the branch address is fetched.
- Several techniques for reducing branch penalties were discussed in the preceding section.
- The method used in most RISC processors is to rely on the compiler to redefine the branches so that they take effect at the proper time in the pipeline.
- This method is referred to as **delayed branch.**

Delayed Branch

- The compiler for a processor that uses delayed branches is designed to analyze the instructions before and after the branch and rearrange the program sequence by inserting useful instructions in the delay steps.
- For example, the compiler can determine that the program dependencies allow one or more instructions preceding the branch to be moved into the delay steps after the branch.
- These instructions are then fetched from memory and executed through the pipeline while the branch instruction is being executed in other segments.
- The effect is the same as if the instructions were executed in their original order, except that the branch delay is removed.
- It is up to the compiler to find useful instructions to put after the branch instruction.
- Failing that, the compiler can insert no-op instructions.

Load from memory to *R*1

Increment *R*2

Add *R*3 to *R*4

Subtract *R*5 from *R*6

Branch to address *X*

Clock cycles:	1	2	3	4	5	6	7	8	9	10
1. Load	I	A	E							
2. Increment		I	A	E						
3. Add			I	A	E					
4. Subtract				I	A	E				
5. Branch to *X*					I	A	E			
6. No-operation						I	A	E		
7. No-operation							I	A	E	
8. Instruction in *X*								I	A	E

(a) Using no-operation instructions

the compiler inserts two no-op instructions after the branch.
The branch address *X* is transferred to *PC* in clock cycle 7.
The fetching of the instruction at *X* is delayed by two clock cycles by the no-op instructions.
The instruction at *X* starts the fetch phase at clock cycle 8 after the program counter *PC* has been updated.

Load from memory to *R*1

Increment *R*2

Add *R*3 to *R*4

Subtract *R*5 from *R*6

Branch to address *X*

Clock cycles:	1	2	3	4	5	6	7	8
1. Load	I	A	E					
2. Increment		I	A	E				
3. Branch to *X*			I	A	E			
4. Add				I	A	E		
5. Subtract					I	A	E	
6. Instruction in *X*						I	A	E

(b) Rearranging the instructions

The program is rearranged by placing the add and subtract instructions after the branch instruction instead of before as in the original program.
Inspection of the pipeline timing shows that *PC* is updated to the value of *X* in clock cycle 5, but the add and subtract instructions are fetched from memory and executed in the proper sequence.
In other words, if the load instruction is at address 101 and *X* is equal to 350, the branch instruction is fetched from address 103.
The add instruction is fetched from address 104 and executed in clock cycle 6.
The subtract instruction is fetched from address 105 and executed in clock cycle 7.
Since the value of *X* is transferred to *PC* with clock cycle 5 in the E segment, the instruction fetched from memory at clock cycle 6 is from address 350, which is the instruction at the branch address.