2. The Pipeline

In **pipelining m**ultiple tasks (for example instructions) are executed in parallel.

To use the pipelining approach efficiently

- We must have tasks that are repeated many times on different data
- Tasks must be divided into small pieces (operations or actions) that can be performed in parallel.

An example for a pipeline: an automobile assembly line.

The task is the construction of a car.

This task is repeated many times for different cars.

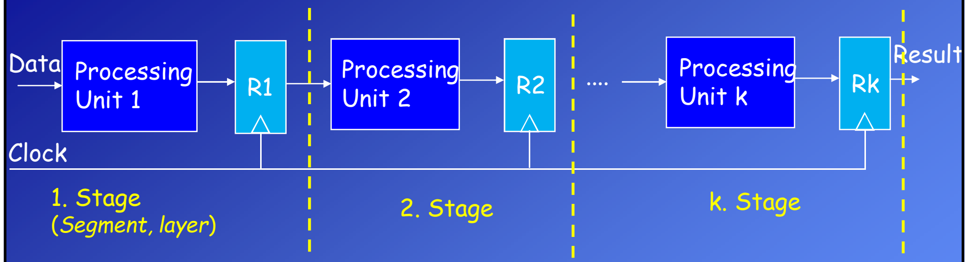
The task consists of some steps (operations), such as assembling the doors, assembling the tires.

For each step there is a station in the pipeline (assembly line).

Each step operates in parallel with other steps but on a different car.

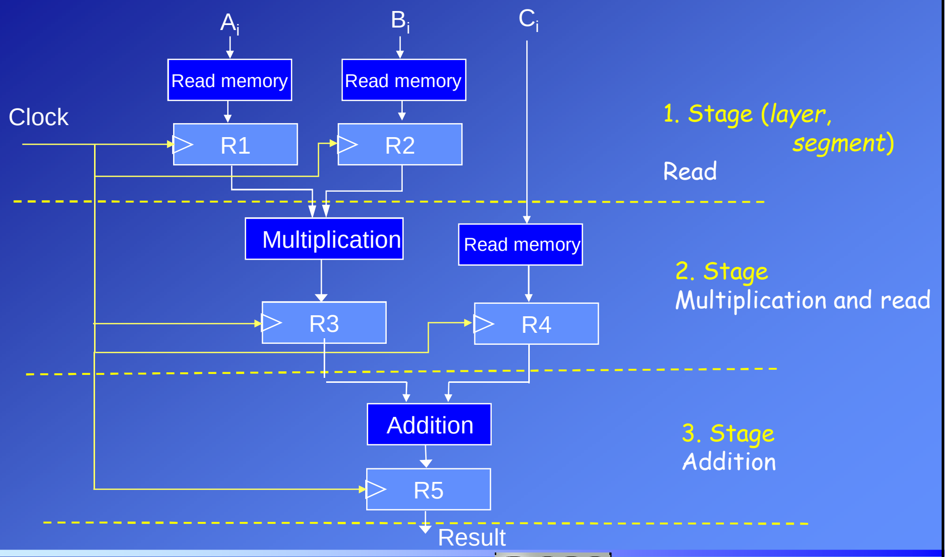
For example, while a worker is assembling the doors of the ith car, another worker is assembling the tires of the (i+1)th car at the same time.

2.1 The general structure of a pipeline:



- Each processing unit performs a fixed operation.
- On different clock cycles the operation is performed on different data (task). (*Refer to Digital Circuits Lecture notes, Section 6 for information about clock signal.*)
- Registers (R1, R2, Rk) keep the intermediate results.
- All stages are controlled by a common clock signal and operate synchronously.
- New inputs are accepted at one end, before previously accepted inputs appear as outputs at the other end.
- When all stages of the pipeline is full, on each clock cycle a new result is obtained at the output.

Example: The elements of the arrays A, B and C will be first read from the memory and than the following operation will be performed: $A_i*B_i + C_i$ i=1,2,3,....

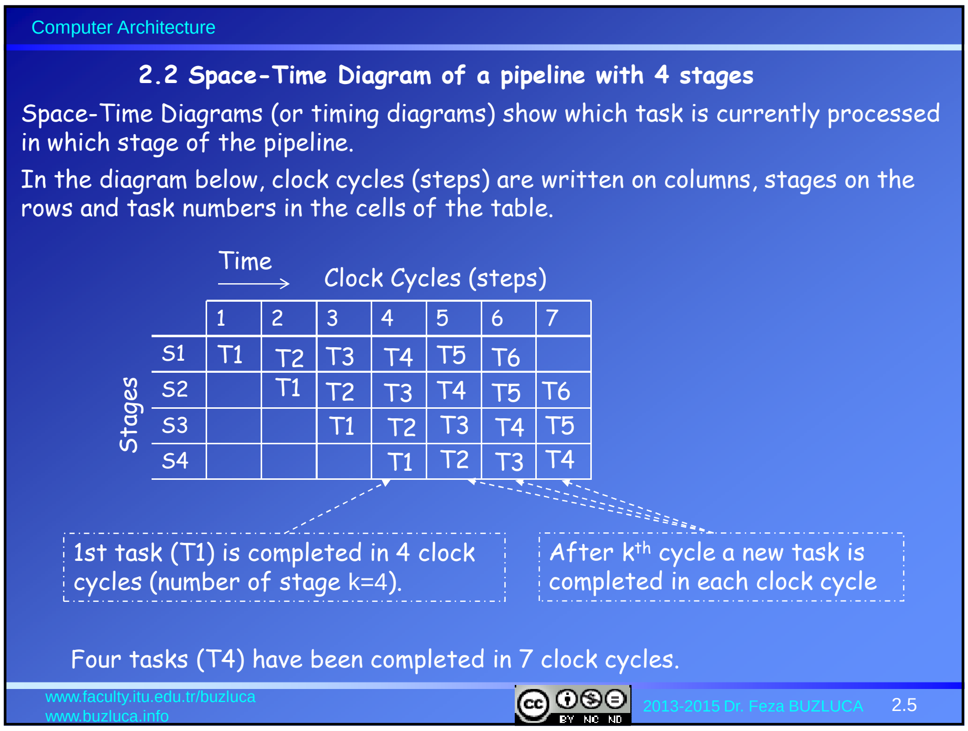


In this example the task is decomposed as 3 operations: Reading, multiplication and addition

Functioning of the pipeline with three stages:

Clock cycle	1. Stage (Read)		2. Stage(Multiply)		3.Stage (Add)
	R1	R2	R3	R4	R5
1	A_1	B_1	-	-	-
2	A_2	B_2	A_1*B_1	C_1	-
3	A_3	B_3	A_2*B_2	C_2	$A_1*B_1 + C_1$
4	A_4	B_4	A_3*B_3	C_3	$A_2*B_2 + C_2$
5	A_5	B_5	A_4*B_4	C_4	$A_3*B_3 + C_3$

Note:
Under the assumption that the access time of the memory is very shorter than the durations of the other operations and the data is always ready to be read, then reading is not handled as a separate operation.
In this case the pipeline could be designed with two stages which perform only arithmetical operations: multiplication and addition



2.2 Space-Time Diagram of a pipeline with 4 stages

Space-Time Diagrams (or timing diagrams) show which task is currently processed in which stage of the pipeline.

In the diagram below, clock cycles (steps) are written on columns, stages on the rows and task numbers in the cells of the table.

Time $\longrightarrow$ Clock Cycles (steps)

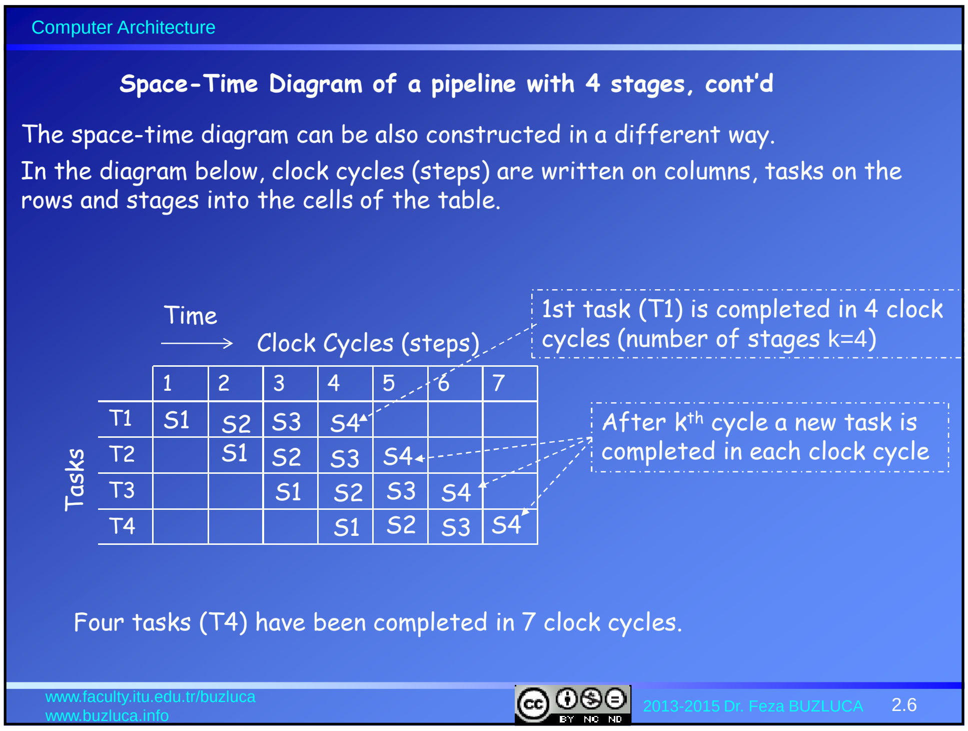
Stages

	1	2	3	4	5	6	7
S1	T1	T2	T3	T4	T5	T6	
S2		T1	T2	T3	T4	T5	T6
S3			T1	T2	T3	T4	T5
S4				T1	T2	T3	T4

1st task (T1) is completed in 4 clock cycles (number of stage k=4).

After kth cycle a new task is completed in each clock cycle

Four tasks (T4) have been completed in 7 clock cycles.



Space-Time Diagram of a pipeline with 4 stages, cont'd

The space-time diagram can be also constructed in a different way.

In the diagram below, clock cycles (steps) are written on columns, tasks on the rows and stages into the cells of the table.

Time $\longrightarrow$ Clock Cycles (steps)

1st task (T1) is completed in 4 clock cycles (number of stages k=4)

After kth cycle a new task is completed in each clock cycle

Tasks

	1	2	3	4	5	6	7
T1	S1	S2	S3	S4			
T2		S1	S2	S3	S4		
T3			S1	S2	S3	S4	
T4				S1	S2	S3	S4

Four tasks (T4) have been completed in 7 clock cycles.

2.3 Throughput and Speedup provided by the pipeline

Because all stages proceed at the same time, the length of the period of the clock signal (cycle time) is determined by the time (delay) required for the **slowest stage**.

The cycle time (the period of the clock) t_p can be determined as follows:

$$t_p = \max(\tau_i) + d_r = \tau_M + d_r$$

t_p: cycle time

τ_i : time delay of the circuitry in the i^{th} stage

τ_M : maximum stage delay (the slowest stage)

d_r : time delay of the register

Speedup:

k: Number of stages in the pipeline
tp: cycle time
n: number of tasks
A total of k cycles are required to complete the execution of the first task (T1).
Required time: T(1) = k·tp
Remaining n-1 tasks require (n-1) cycles. Time: (n-1)tp
Total required time for n tasks: (k+n-1)tp
tn : Required time for a task without pipelining

Speedup: $S = \dfrac{\text{Execution time \textbf{without} the pipeline}}{\text{Execution time \textbf{with} the pipeline}}$ $\qquad S = \dfrac{n \cdot t_n}{(k+n-1)\cdot t_p}$

If we have many tasks

$$\lim_{n\to\infty} S = \frac{t_n}{t_p}$$

Under assumption tn = k·tp
(If it is possible to divide the main task into k equal small operations.)

$S_{max} = k$ (Theoretic maximum speedup)

Comments on speedup:

To improve the performance of the pipeline the tasks must be divided into balanced, small operations with equal (at least similar) durations.

If the durations of the operations are small then the clock cycle can be short.

Remember the slowest stage determines the clock cycle.

Effects of increasing the number of stages of a pipeline:

Advantage:

- If the task can be divided into many small operations, increasing the number of stages can increase the speed of the clock signal and consequently the speedup.

$$S_{max} = k$$

Disadvantages:

- The cost of the pipeline increases. At each stage of the pipeline, there is some overhead (cost, energy, space) because of registers.
- The completion time of the first task increases. $T(1)=k \cdot tp$
- Branch penalties in the instruction pipelines caused by control hazards increase. We will discuss branch penalties in the section "2.5 Pipeline hazards".

While designing a pipeline these advantages and disadvantages should be taken into consideration.

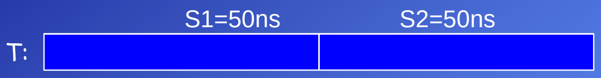
Effects of task partitioning on the speedup:

If the task can be partitioned into small operations with small durations then a faster clock signal can be applied.

Assume that we have a task T with a total duration of 100 ns.

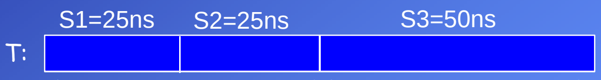
Assume that we can decompose this task in different ways.

Case A: We partition the task into 2 equal stages.



If the delay of the registers is 5 ns then the clock cycle is $t_p = 50+5 = 55$ ns

Case B: We partition the task into 3 not balanced stages.



The clock cycle is $t_p = 50+5 = 55$ ns (slowest stage $\tau_M = 50$ns)

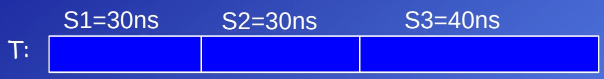
Although the pipeline has more stages, there is no speed improvement compared to case A.

Besides, the cost of the pipeline is increased.

The completion time of the first task increases. $T(1)=k \cdot t_p$

Effects of task partitioning on the speedup: (cont'd)

Case C: We partition the task into 3 stages with similar durations.



The clock cycle is t_p = 40+5 = 45 ns (slowest stage τ_M =40ns)
The clock signal is faster compared to cases A and B.

Conclusion:

The pipelining has advantages if a task can be partitioned into small and balanced operations.

For example if we could partition the task into 5 operations each having the duration of 20ns we would have a clock cycle of 25ns.

2.4 Instruction Pipeline (Instruction-Level Parallelism)

During the execution of each instruction the CPU repeats some operations.

The processing required for a single instruction is called an **instruction cycle** that includes the general stages, instruction fetch and decoding, operand fetch, execution, interrupt. (See the figure on 1.18)

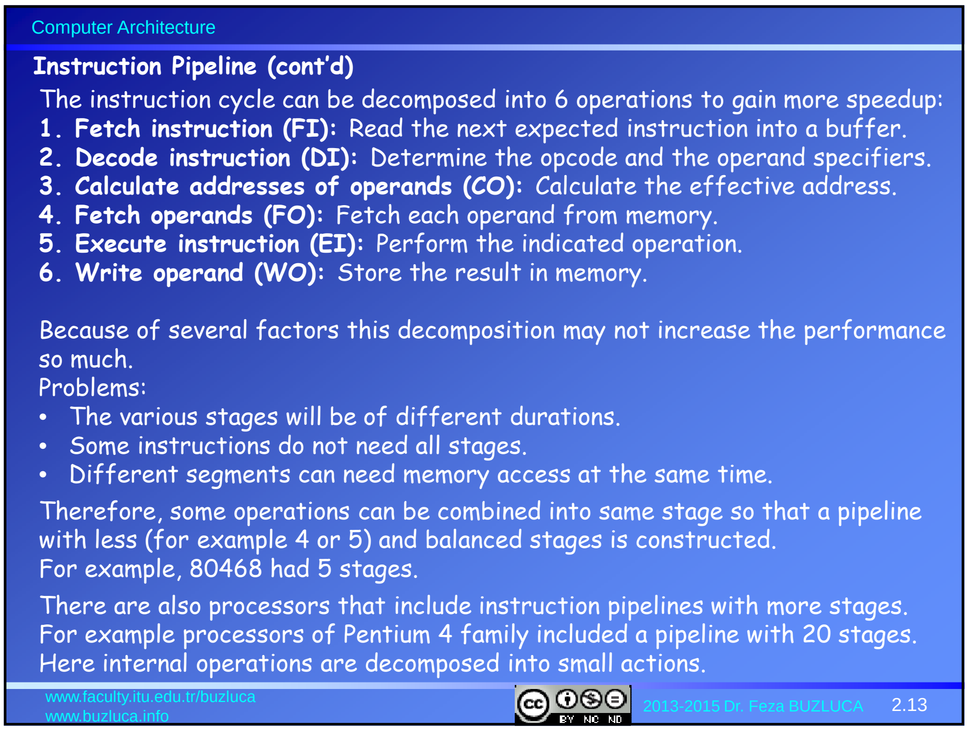
The simplest instruction pipeline can be constructed with two stages:

1) Fetch and decode instruction 2) Fetch operands and Execute instruction

When the main memory is not being accessed during the execution of an instruction, this time can be used to fetch the next instruction in parallel with the execution of the current one.

The potential overlap among instruction is called **instruction-level** parallelism.

Remember, to gain more speedup, the pipeline must have more stages with small durations.



Instruction Pipeline (cont'd)

The instruction cycle can be decomposed into 6 operations to gain more speedup:

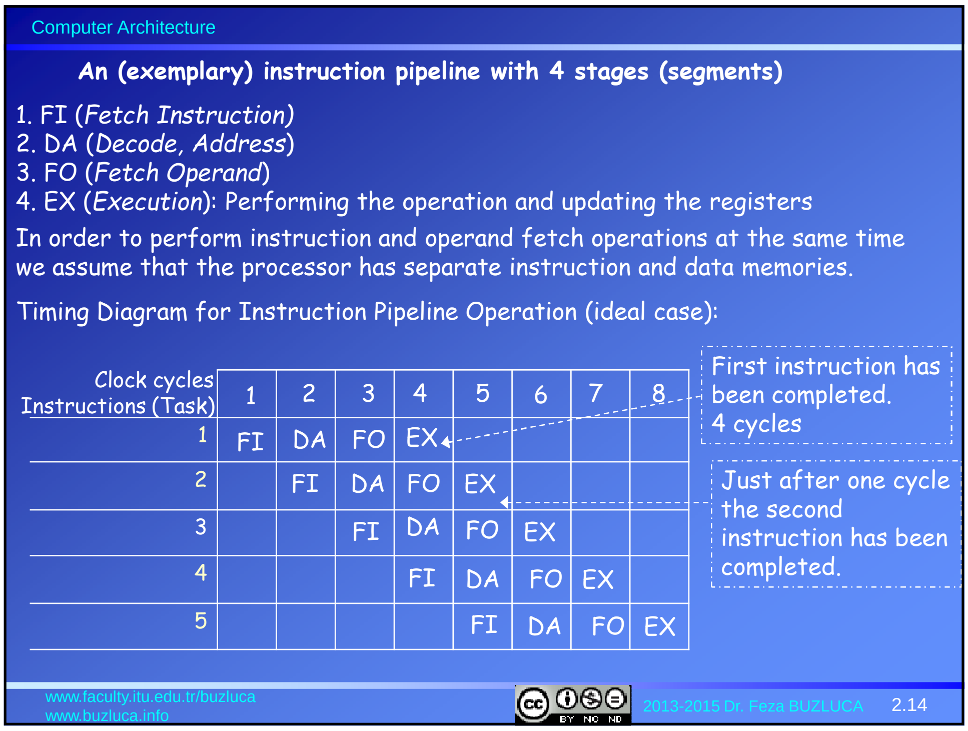
1. **Fetch instruction (FI):** Read the next expected instruction into a buffer.
2. **Decode instruction (DI):** Determine the opcode and the operand specifiers.
3. **Calculate addresses of operands (CO):** Calculate the effective address.
4. **Fetch operands (FO):** Fetch each operand from memory.
5. **Execute instruction (EI):** Perform the indicated operation.
6. **Write operand (WO):** Store the result in memory.

Because of several factors this decomposition may not increase the performance so much.
Problems:

- The various stages will be of different durations.
- Some instructions do not need all stages.
- Different segments can need memory access at the same time.

Therefore, some operations can be combined into same stage so that a pipeline with less (for example 4 or 5) and balanced stages is constructed.
For example, 80468 had 5 stages.

There are also processors that include instruction pipelines with more stages. For example processors of Pentium 4 family included a pipeline with 20 stages. Here internal operations are decomposed into small actions.



An (exemplary) instruction pipeline with 4 stages (segments)

1. FI (*Fetch Instruction*)
2. DA (*Decode, Address*)
3. FO (*Fetch Operand*)
4. EX (*Execution*): Performing the operation and updating the registers

In order to perform instruction and operand fetch operations at the same time we assume that the processor has separate instruction and data memories.

Timing Diagram for Instruction Pipeline Operation (ideal case):

Clock cycles / Instructions (Task)	1	2	3	4	5	6	7	8
1	FI	DA	FO	EX				
2		FI	DA	FO	EX			
3			FI	DA	FO	EX		
4				FI	DA	FO	EX	
5					FI	DA	FO	EX

First instruction has been completed.
4 cycles

Just after one cycle the second instruction has been completed.

2.5 Pipeline Hazards (Conflicts)

2.5.1. Control Hazards (Branches, Interrupts):

Because in a pipeline instructions are processed in parallel, during the process of a branch instruction the next instruction in the memory that should be actually skipped also enters the pipeline.

Here, a solution mechanism is necessary; otherwise the instruction(s) that should be skipped according to the program will also be executed.

Example:

1.	Instruction_1	
2.	JUMP Target	Unconditional branch instruction (BRA)
3.	Instruction_3	Next instruction in the memory According to the program it **should be skipped**.
	:	
4. Target	Instruction_4	Target of the branch (target instruction)

During the process of the unconditional branch instruction JUMP, Instruction_3 is also fetched into the pipeline.

To prevent the program from running incorrectly the pipeline must be stopped (stall) or emptied before the Instruction_3 is executed.

a. Unconditional Branch

Clock cycles / Instructions		1	2	3	4	5	6	7
	1	FI	DA	FO	EX			
Unconditional Branch JUMP	2		FI	DA	FO	EX		
	3			~~FI~~	-	-		
Target	4						FI	DA

Steps

After decoding, the type of the instruction is determined: branch!

The branch address is taken (absolute or relative)

Updating the PC (program counter) PC = Target (Target of branch)

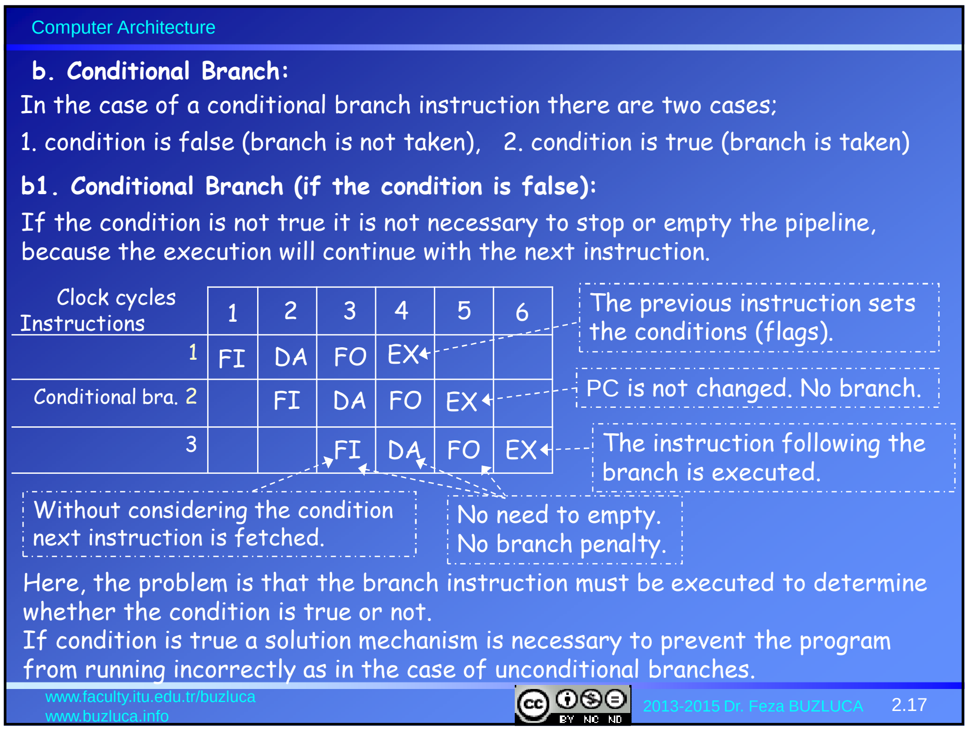
The new instruction after branch operation (Target of branch)

Hazard: This instruction is fetched unnecessarily. It must not be executed. It will be discarded.

Branch penalty! It is necessary to *stall* or *empty* the pipeline.

After decoding (identification) of the unconditional branch instruction, one possible solution is to stop the pipeline (FI stage) to fetch new instructions.

After the execution of the branch instruction the target address is written to the program counter (PC) and the pipeline is enabled to fetch new instructions.



b. Conditional Branch:

In the case of a conditional branch instruction there are two cases;

1. condition is false (branch is not taken), 2. condition is true (branch is taken)

b1. Conditional Branch (if the condition is false):

If the condition is not true it is not necessary to stop or empty the pipeline, because the execution will continue with the next instruction.

Clock cycles / Instructions	1	2	3	4	5	6
1	FI	DA	FO	EX		
Conditional bra. 2		FI	DA	FO	EX	
3			FI	DA	FO	EX

The previous instruction sets the conditions (flags).

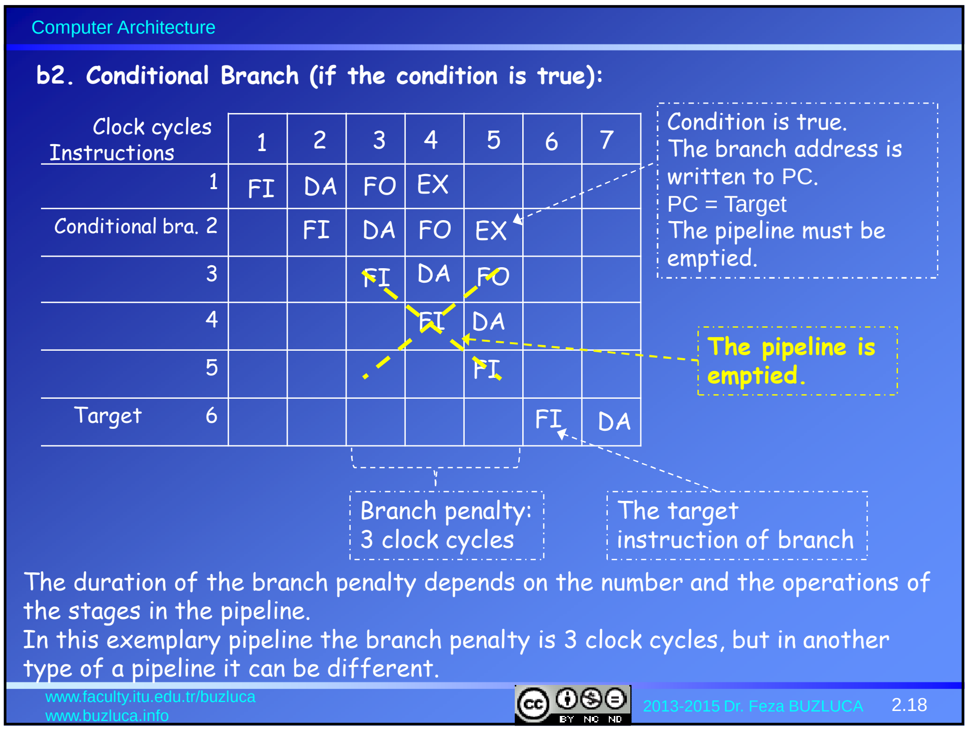
PC is not changed. No branch.

The instruction following the branch is executed.

Without considering the condition next instruction is fetched.

No need to empty. No branch penalty.

Here, the problem is that the branch instruction must be executed to determine whether the condition is true or not.
If condition is true a solution mechanism is necessary to prevent the program from running incorrectly as in the case of unconditional branches.



b2. Conditional Branch (if the condition is true):

Clock cycles / Instructions	1	2	3	4	5	6	7
1	FI	DA	FO	EX			
Conditional bra. 2		FI	DA	FO	EX		
3			~~FI~~	~~DA~~	~~FO~~		
4				~~FI~~	~~DA~~		
5					~~FI~~		
Target 6						FI	DA

Condition is true.
The branch address is written to PC.
PC = Target
The pipeline must be emptied.

The pipeline is emptied.

Branch penalty: 3 clock cycles

The target instruction of branch

The duration of the branch penalty depends on the number and the operations of the stages in the pipeline.
In this exemplary pipeline the branch penalty is 3 clock cycles, but in another type of a pipeline it can be different.

Pipeline Hazards (Conflicts) cont'd

2.5.2. Resource Conflict (Structural hazard):

A resource hazard occurs when two (or more) instructions that are already in the pipeline need the same resource (memory, functional unit).

a) Memory conflict: An operand read to or write from memory cannot be performed in parallel with an instruction fetch.

Solutions:

- Instructions must be executed in serial rather than parallel for a portion of the pipeline (*stall*). (Performance drops.)
- Harvard architecture: Separate memories for instructions and data.
- Instruction queue or cache memory: There are times during the execution of an instruction when main memory is not being accessed. This time could be used to prefetch the next instruction and write it to a queue (instruction buffer).

b) Functional unit (ALU, FPU) conflict.

Solutions:

- Increasing available functional units and using multiple ALUs.

For example different ALUs can be used address calculation and data operations.

- Fully pipelining a functional unit (for example a floating point unit FPU)

2.5.3. Data Conflict:

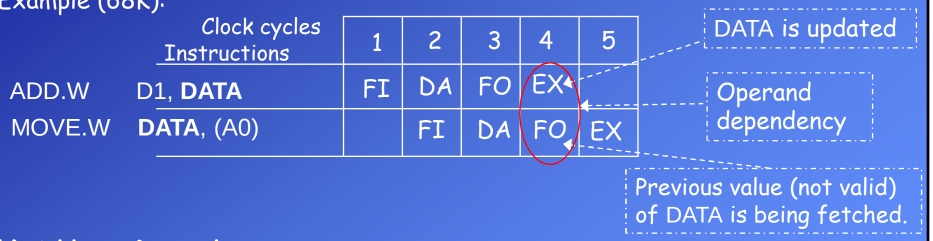
There is a conflict in the access of a data location.

If the problem is not solved the program produces an incorrect result because of the use of pipelining.

a) Operand dependency:

The operand of an instruction depends on the result of another instruction

Example (68K):



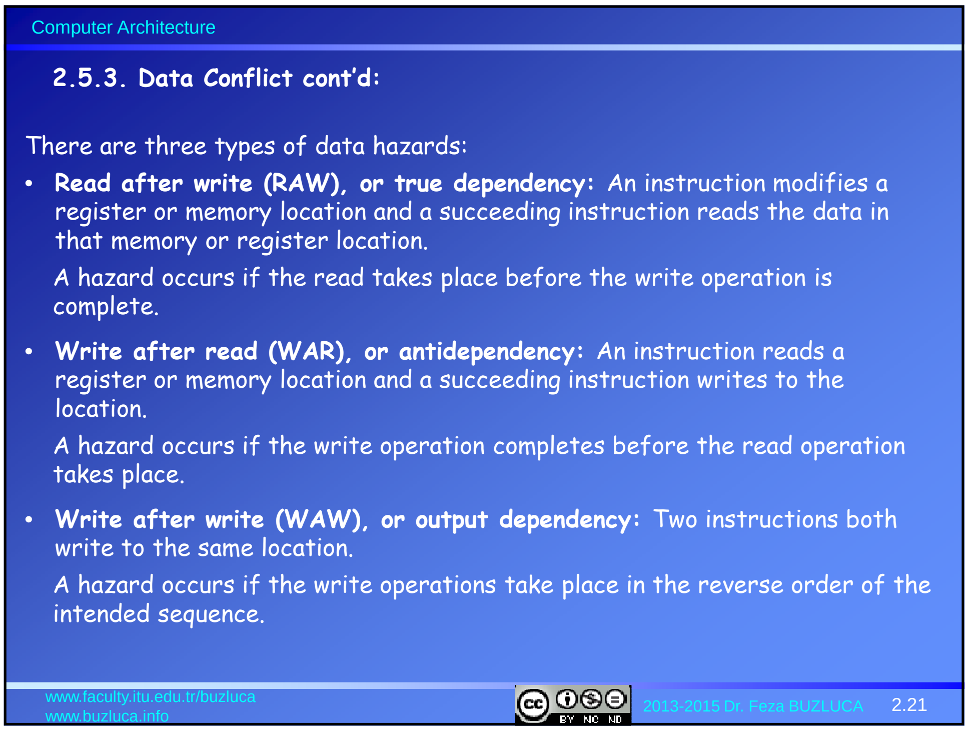
Instructions \ Clock cycles		1	2	3	4	5
ADD.W	D1, **DATA**	FI	DA	FO	EX	
MOVE.W	**DATA**, (A0)		FI	DA	FO	EX

b) Address Dependency:

Data conflict can also occur on address registers (pointers).

Example (68K):

```
ADDA.W    #2, A0
MOVE.B    (A0)+, D0
```



2.5.3. Data Conflict cont'd:

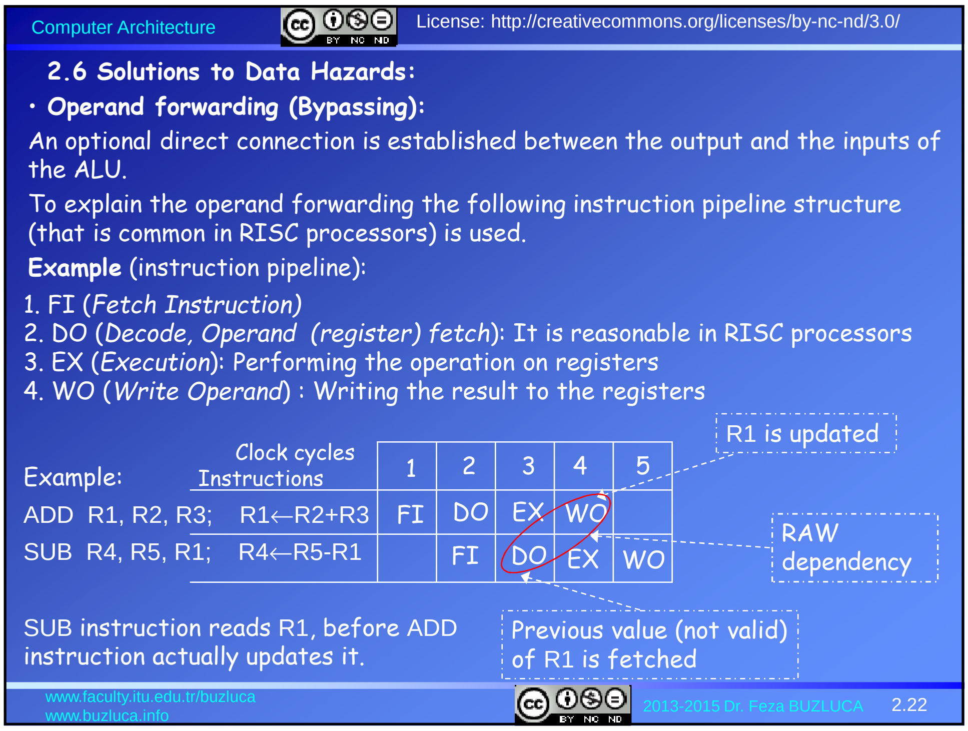
There are three types of data hazards:

- **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.



2.6 Solutions to Data Hazards:

• **Operand forwarding (Bypassing):**

An optional direct connection is established between the output and the inputs of the ALU.

To explain the operand forwarding the following instruction pipeline structure (that is common in RISC processors) is used.

Example (instruction pipeline):

1. FI (*Fetch Instruction)*
2. DO (*Decode, Operand (register) fetch*): It is reasonable in RISC processors
3. EX (*Execution*): Performing the operation on registers
4. WO (*Write Operand*) : Writing the result to the registers

Example:

Clock cycles / Instructions		1	2	3	4	5
ADD R1, R2, R3;	R1←R2+R3	FI	DO	EX	WO	
SUB R4, R5, R1;	R4←R5-R1		FI	DO	EX	WO

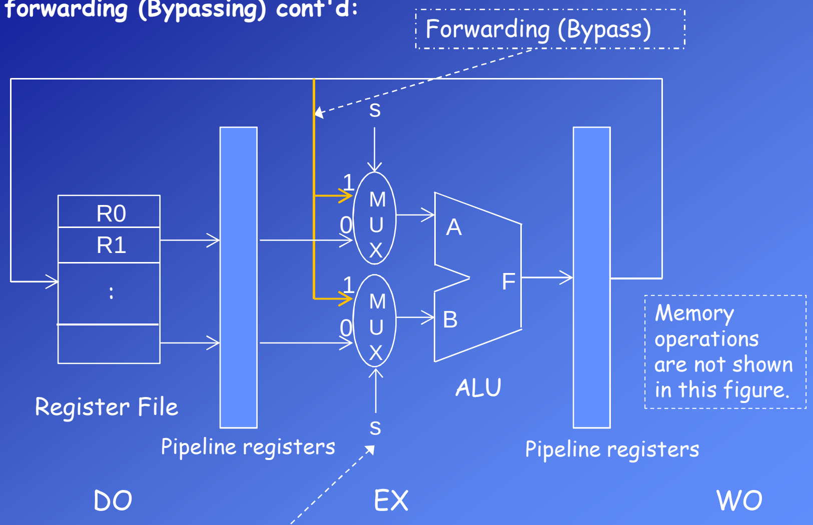
R1 is updated

RAW dependency

SUB instruction reads R1, before ADD instruction actually updates it.

Previous value (not valid) of R1 is fetched

Operand forwarding (Bypassing) cont'd:



s is controlled by the hazard detection unit of the pipeline. It selects the value from the register file or the forwarded result (bypass) as the ALU input.

Operand forwarding (Bypassing) cont'd:

If the hazard unit detects that the destination of the previous ALU operation is the same register as the source of the current ALU operation, control logic selects the forwarded result (bypass) as the ALU input rather the value from the register.

Example:

Clock cycles / Instructions	1	2	3	4	5
ADD R1, R2, R3; R1←R2+R3	FI	DO	EX	WO	
SUB R4, R5, R1; R4←R5-R1		FI	DO	EX	WO

Previous value (not valid) of R1 is fetched.
This invalid value will **not be used** in the EX cycle.

The control unit of the pipeline selects the output of the previous ALU operation as the input, not the value that has been read in the DO stage.

If it is possible to solve the conflict by forwarding it is not necessary to stall the pipeline and the there is not a decrease in performance.

2.6 Solutions to Data Hazards (cont'd):

- **Hardware interlock**:

A hardware unit tracks all instructions and stops (stalls) the instruction fetch (FI) stage of the pipeline when a hazard is detected.

The instruction that causes the hazard is delayed (is not fetched) until the conflict is solved.

- **Compiler-based Solutions**:

These are software-based solutions. An additional hardware unit is not necessary.

- *Delayed Load*:

a) The compiler inserts NOP (*No Operation*) instructions between the instructions that cause data hazards. (The performance drops.)

Example:
```
ADD  R1, R2, R3;    R1←R2+R3
NOP
NOP
SUB  R4, R5, R1;    R4←R5-R1
```

b) The compiler changes the order of the instruction if it is possible, without changing the algorithm of the program.

We will see these compiler-based solutions in the chapter "2.8 RISC Pipelining".

2.7 Dealing with branches:

Remember if there are branch/jump instructions in the program the pipeline must be stopped or emptied (slides 2.16, 2.18).

Otherwise the CPU will also execute the next instruction of the jump that should be skipped.

Stopping (stall) or emptying the pipeline decrease the performance of the system.

To avoid the decrease in performance, a mechanism that can fetch the target instruction of the branch instead of the next instruction would be useful.

The main problem is the **conditional branch** instruction because until the instruction is actually executed, it is impossible to determine whether the branch will be taken or not (slide 2.18).

To solve this problem **prediction mechanisms** are used.

Another problem is that the target address of the branch is determined in the execution cycle of the branch instruction.

Therefore it is unknown which is the target instruction to be fetched into the pipeline.

To solve this problem **branch target table** is used.

Solution mechanisms for branch problems:

2.7.1 Target Instruction prefetch:

When a conditional branch is recognized, the target of the branch is prefetched, in addition to the instruction following the branch.

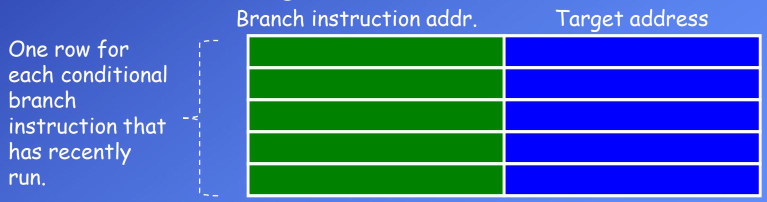
To determine the target of the branch in advance the branch target table is used.

Branch target (buffer) table:

In the branch target table, addresses of the conditional branch instructions and their target addresses (where they jump) are kept in an associative memory.

There is a separate row for each conditional branch instruction that has recently run. The number is limited with the size of the table.

With the help of this buffer the target instruction of the branch can be prefetch without calculating the branch address.


Branch instruction addr.
Target address
One row for each conditional branch instruction that has recently run.

2.7.2 Branch prediction:

Static branch prediction strategies:

a) Always predict not taken: Always assumes that the branch will not be taken and fetches the next instruction in sequence.

b) Always predict taken: Always predicts that the branch will be taken and fetches target instruction of the branch. (Performs better than a)

Studies analyzing program behavior have shown that conditional branches are taken more than 50% of the time.

Therefore; always prefetching from the branch target address should give better performance than always prefetching from the sequential path.

Dynamic branch prediction strategies:

Dynamic branch strategies record the history of all conditional branch instructions in the active program to predict whether the condition will be true or not.

One or more **prediction bits** (or counters) are associated with each conditional branch instruction in a program that reflect the recent history of the instruction.

These prediction bits are kept in a branch history table (See 2.29) and they provide information about the branch history of the instruction (branch was taken or not in previous runs).

Branch target buffer and branch history table (BHT):

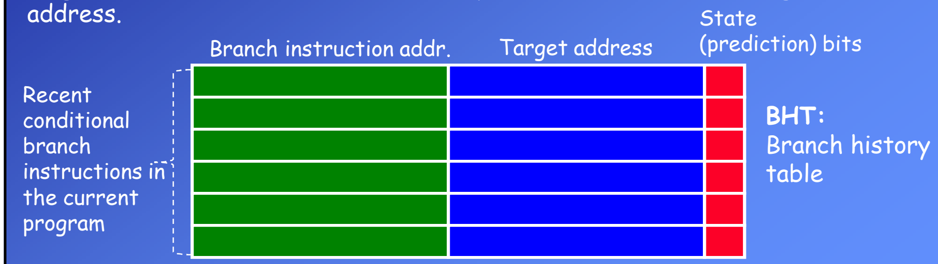
Prediction bits are kept in a high-speed memory location called branch history table (BHT).

In the BHT, for recent branch instructions of the current program, the address of the instruction, the target address and the state (prediction) bits are stored.

Each time a conditional branch instruction is executed the associated prediction bits are updated according to whether the branch is taken or not.

These prediction bits direct the pipeline control unit to make the decision the next time the branch instruction is encountered.

If the prediction is to "take the branch", with the help of the target buffer the target instruction of the branch can be prefetch without calculating the branch address.


State (prediction) bits
Branch instruction addr.
Target address
Recent conditional branch instructions in the current program
BHT: Branch history table

1-bit dynamic prediction scheme:

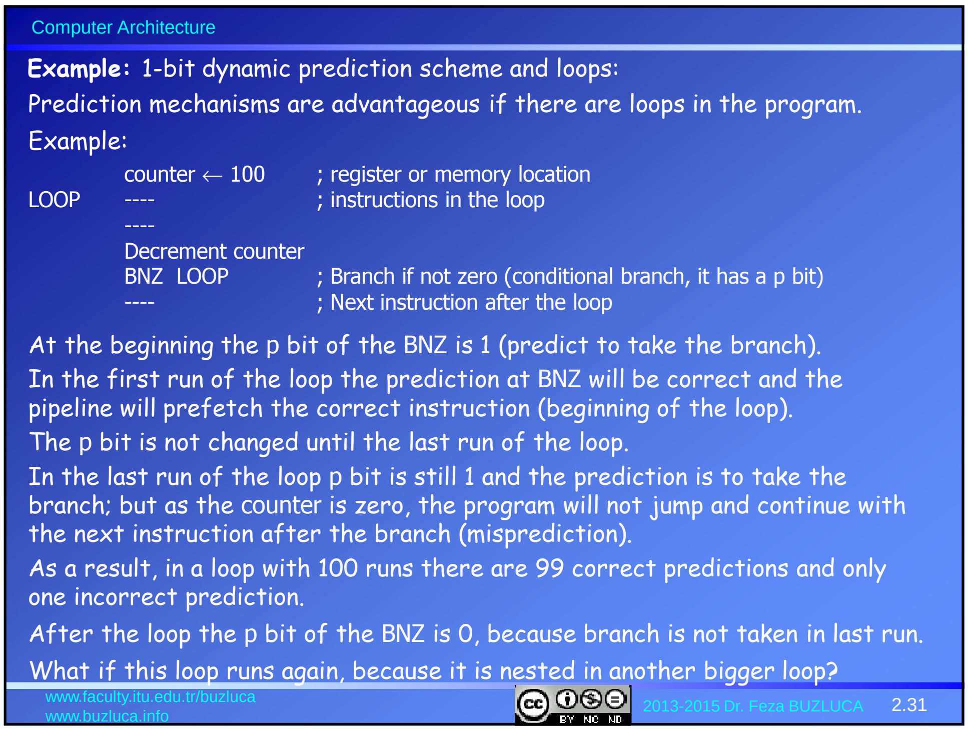
For each conditional branch instruction one **prediction bit (p_i)** is stored.

The prediction bit only records whether the last execution of this instruction resulted in a branch or not.

If the branch was taken last time, the system predicts to take the branch next time.

Algorithm:

```
Fetch the ith conditional branch instruction
If ($p_i$ = 0) predict not to take the branch, prefetch the next instruction in sequence
If ($p_i$ = 1) predict to take the branch, prefetch the target instruction of the branch
If the branch is really taken then $p_i \leftarrow 1$
If the branch is not really taken then $p_i \leftarrow 0$
```



Example: 1-bit dynamic prediction scheme and loops:

Prediction mechanisms are advantageous if there are loops in the program.

Example:

```
        counter ← 100         ; register or memory location
LOOP    ----                  ; instructions in the loop
        ----
        Decrement counter
        BNZ  LOOP             ; Branch if not zero (conditional branch, it has a p bit)
        ----                  ; Next instruction after the loop
```

At the beginning the p bit of the BNZ is 1 (predict to take the branch).

In the first run of the loop the prediction at BNZ will be correct and the pipeline will prefetch the correct instruction (beginning of the loop).

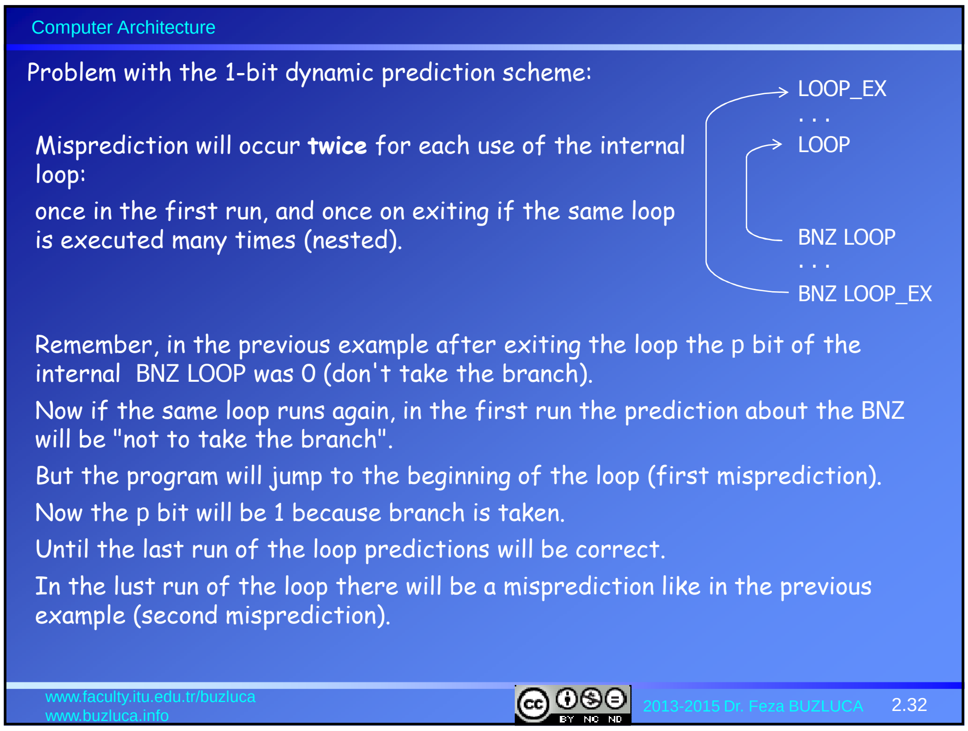
The p bit is not changed until the last run of the loop.

In the last run of the loop p bit is still 1 and the prediction is to take the branch; but as the counter is zero, the program will not jump and continue with the next instruction after the branch (misprediction).

As a result, in a loop with 100 runs there are 99 correct predictions and only one incorrect prediction.

After the loop the p bit of the BNZ is 0, because branch is not taken in last run.

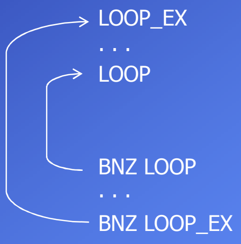
What if this loop runs again, because it is nested in another bigger loop?



Problem with the 1-bit dynamic prediction scheme:

Misprediction will occur **twice** for each use of the internal loop:

once in the first run, and once on exiting if the same loop is executed many times (nested).



Remember, in the previous example after exiting the loop the p bit of the internal BNZ LOOP was 0 (don't take the branch).

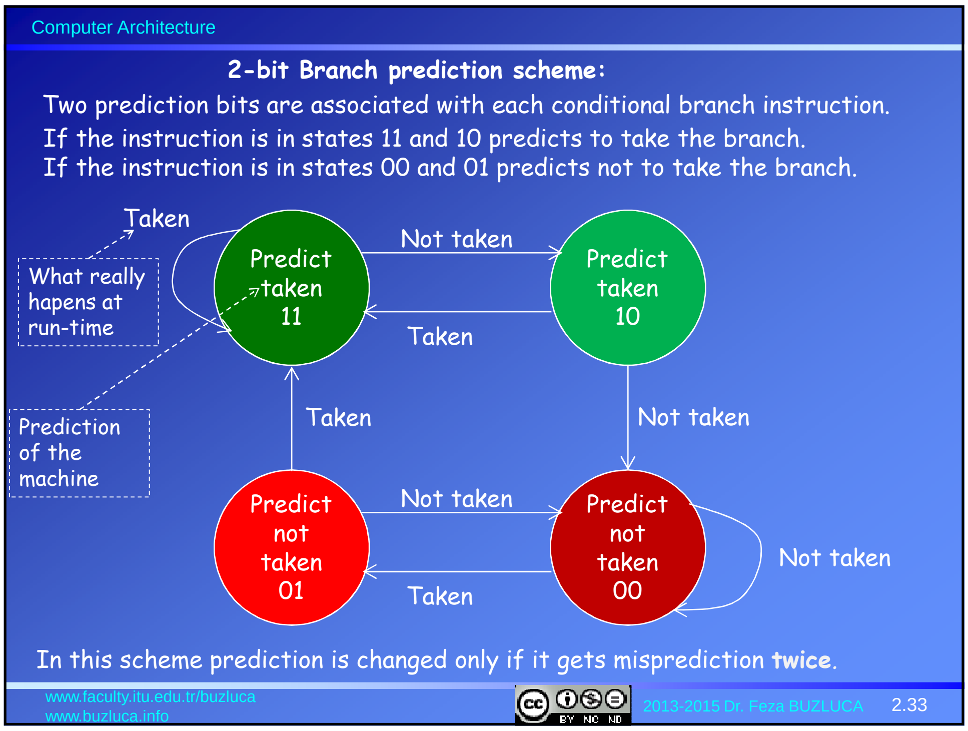
Now if the same loop runs again, in the first run the prediction about the BNZ will be "not to take the branch".

But the program will jump to the beginning of the loop (first misprediction).

Now the p bit will be 1 because branch is taken.

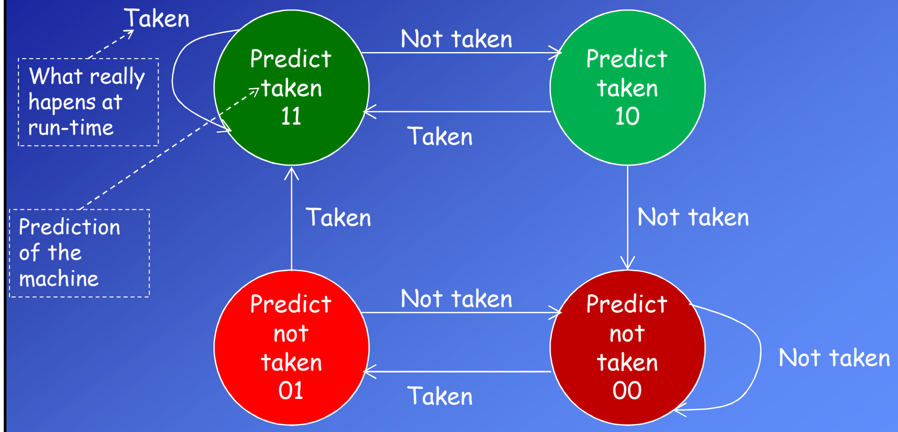
Until the last run of the loop predictions will be correct.

In the lust run of the loop there will be a misprediction like in the previous example (second misprediction).

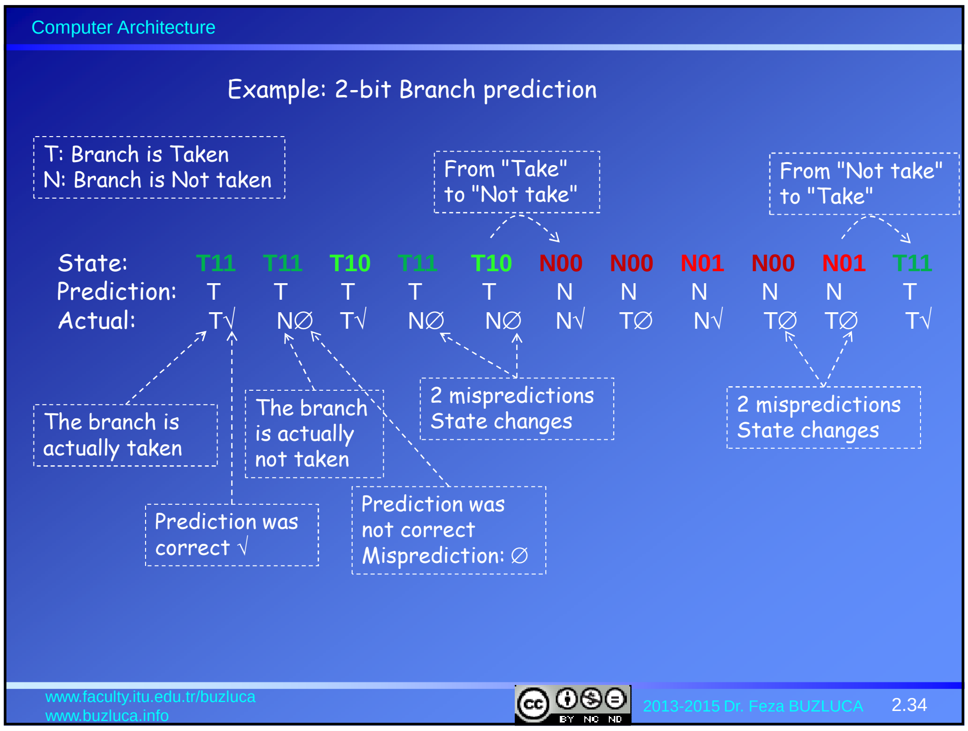


2-bit Branch prediction scheme:

Two prediction bits are associated with each conditional branch instruction.
If the instruction is in states 11 and 10 predicts to take the branch.
If the instruction is in states 00 and 01 predicts not to take the branch.



In this scheme prediction is changed only if it gets misprediction **twice**.



Example: 2-bit Branch prediction

T: Branch is Taken
N: Branch is Not taken

From "Take" to "Not take"

From "Not take" to "Take"

State:	T11	T11	T10	T11	T10	N00	N00	N01	N00	N01	T11
Prediction:	T	T	T	T	T	N	N	N	N	N	T
Actual:	T√	N∅	T√	N∅	N∅	N√	T∅	N√	T∅	T∅	T√

The branch is actually taken

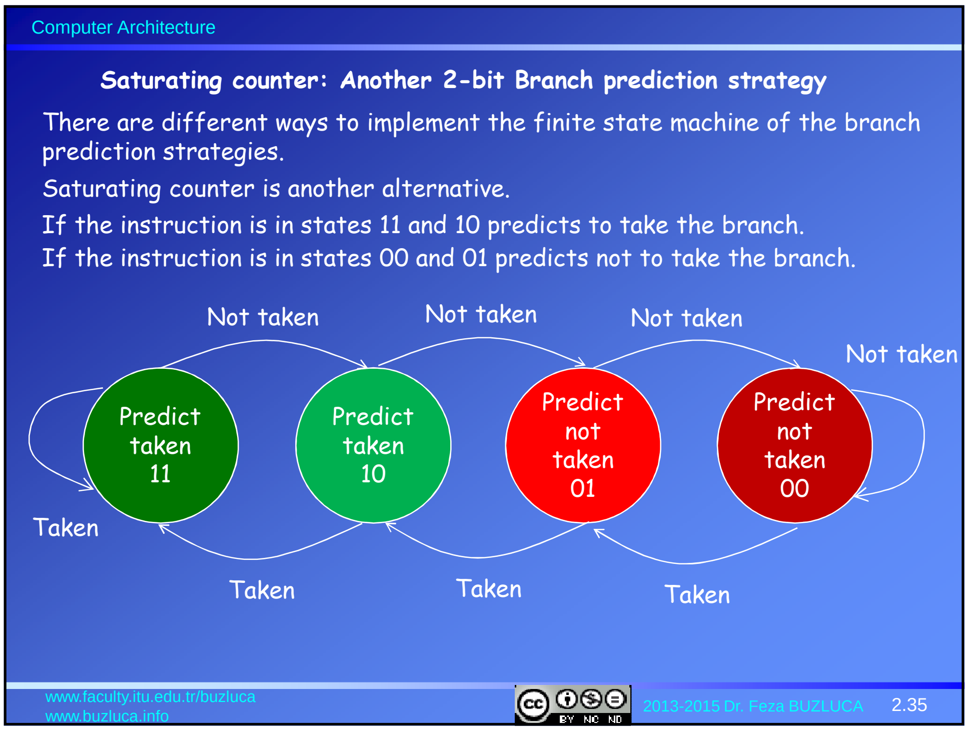
Prediction was correct √

The branch is actually not taken

Prediction was not correct Misprediction: ∅

2 mispredictions State changes

2 mispredictions State changes



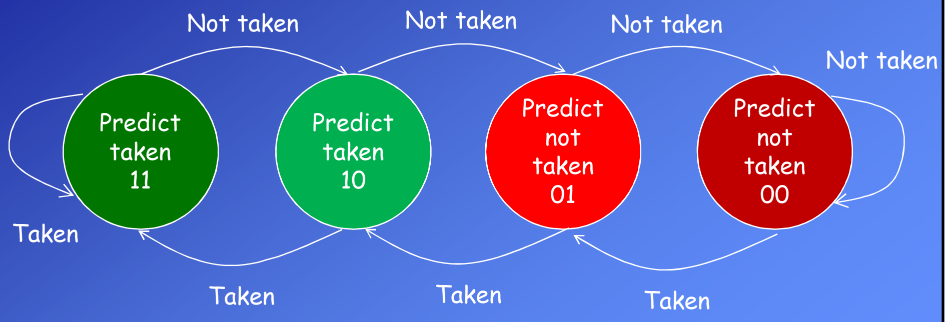
Saturating counter: Another 2-bit Branch prediction strategy

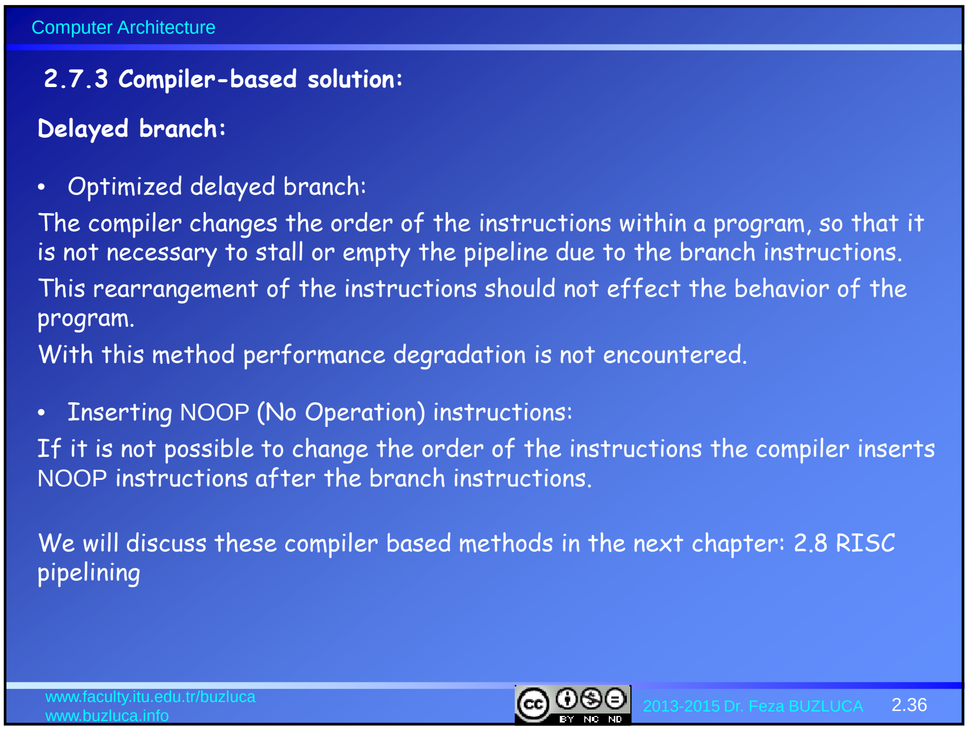
There are different ways to implement the finite state machine of the branch prediction strategies.

Saturating counter is another alternative.

If the instruction is in states 11 and 10 predicts to take the branch.

If the instruction is in states 00 and 01 predicts not to take the branch.





2.7.3 Compiler-based solution:

Delayed branch:

- Optimized delayed branch:

The compiler changes the order of the instructions within a program, so that it is not necessary to stall or empty the pipeline due to the branch instructions.

This rearrangement of the instructions should not effect the behavior of the program.

With this method performance degradation is not encountered.

- Inserting NOOP (No Operation) instructions:

If it is not possible to change the order of the instructions the compiler inserts NOOP instructions after the branch instructions.

We will discuss these compiler based methods in the next chapter: 2.8 RISC pipelining

2.8 RISC Pipelining

In RISC processors most instructions are register to register.

A pipeline with two stages would be sufficient for these instructions:

I: Instruction fetch,   A: ALU operation (execution) on registers

Only for load and store operations memory to register and register to memory instructions are necessary.

For these operation an additional stage (D) to access memory is necessary.

So an instruction pipeline for a RISC processor can be designed with 3 stages:

$$\begin{tikzcd} {} \arrow[r] & \boxed{I} \arrow[r] & \boxed{A} \arrow[r] & \boxed{D} \arrow[r] & {} \end{tikzcd}$$

- I: Instruction Fetch
- A: Decode, ALU Operation
- D: Data, memory access

To increase the performance there are also RISC processors that include pipelines with more stages (4, 5 even more).
For example,
MIPS R3000 has 5 stages
MIPS R4000 has 8 stages (superpipelined)
ARM7 has 3 stages, ARM Cortex-A8 has 13 stages.

2.8.1 Example: A RISC Pipeline with 3 stages

- **I** (*Instruction Fetch*): Read the instruction from memory (Pointed by the PC)
- **A** (*Decode, ALU Operation*):

  ALU is used for three different tasks:

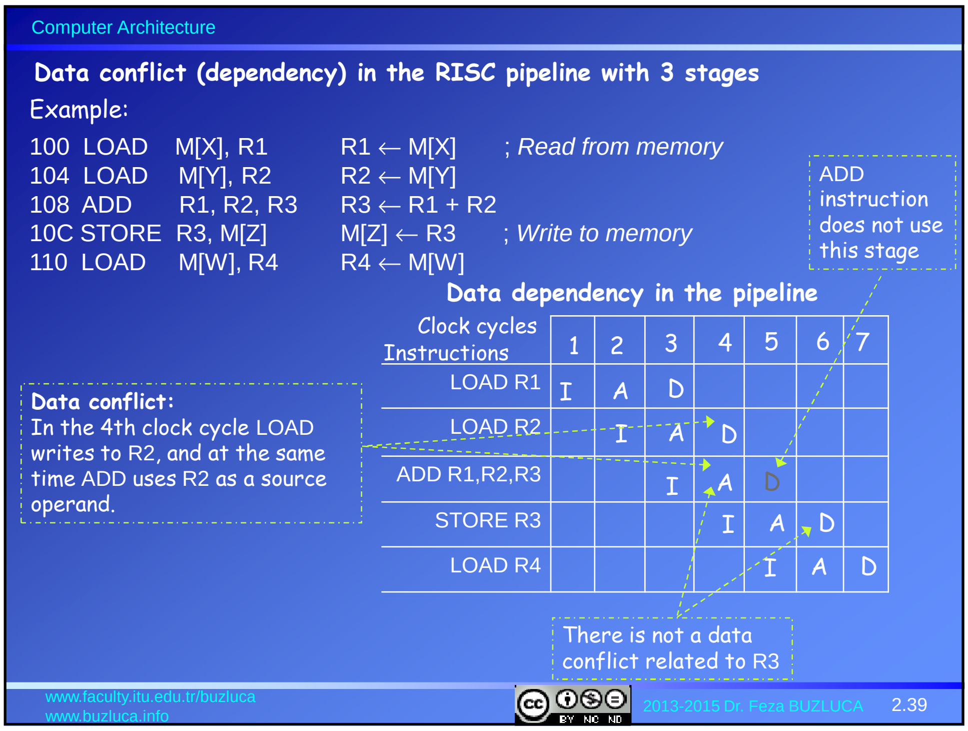
  1. Arithmetical and logical operations on registers
  2. Memory address calculation in load/store instructions. LDL **(R5)#10**, R15
  3. Relative addressing PC ← PC+Y

  In stage A (ALU) the operation is performed and the result is written to the destination register (R or PC) in the same clock cycle.
- **D** (*Data*): This stage is only used for memory access by load/store instructions.

I and D stages try to access the memory at the same time.

To solve the resource conflict problem separate memories for instruction and data can be used (Harvard architecture).

Other solutions are instruction or data queues and cache memories.



Data conflict (dependency) in the RISC pipeline with 3 stages

Example:

```
100  LOAD    M[X], R1        R1 ← M[X]      ; Read from memory
104  LOAD    M[Y], R2        R2 ← M[Y]
108  ADD     R1, R2, R3      R3 ← R1 + R2
10C  STORE   R3, M[Z]        M[Z] ← R3      ; Write to memory
110  LOAD    M[W], R4        R4 ← M[W]
```

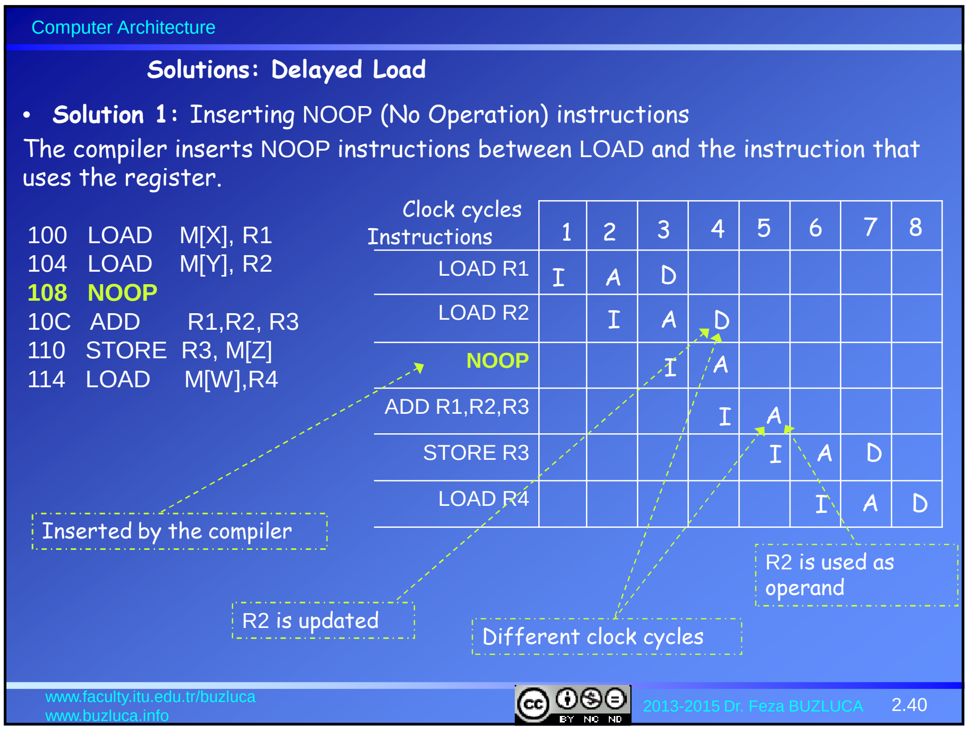
ADD instruction does not use this stage

Data dependency in the pipeline

Clock cycles / Instructions	1	2	3	4	5	6	7
LOAD R1	I	A	D				
LOAD R2		I	A	D			
ADD R1,R2,R3			I	A	D		
STORE R3				I	A	D	
LOAD R4					I	A	D

Data conflict:
In the 4th clock cycle LOAD writes to R2, and at the same time ADD uses R2 as a source operand.

There is not a data conflict related to R3



Solutions: Delayed Load

- **Solution 1:** Inserting NOOP (No Operation) instructions

The compiler inserts NOOP instructions between LOAD and the instruction that uses the register.

```
100   LOAD    M[X], R1
104   LOAD    M[Y], R2
108   NOOP
10C   ADD      R1,R2, R3
110   STORE   R3, M[Z]
114   LOAD     M[W],R4
```

Clock cycles / Instructions	1	2	3	4	5	6	7	8
LOAD R1	I	A	D					
LOAD R2		I	A	D				
NOOP			I	A				
ADD R1,R2,R3				I	A			
STORE R3					I	A	D	
LOAD R4						I	A	D

Inserted by the compiler

R2 is updated

Different clock cycles

R2 is used as operand

Solutions: Delayed Load, cont'd

- **Solution 2:** Changing the order of the instructions, optimized delayed load

The compiler rearranges the program and moves certain instructions between LOAD and instruction that uses the register.

```
100   LOAD    M[X], R1
104   LOAD    M[Y], R2
108   LOAD    M[W], R4
10C   ADD     R1, R2, R3
110   STORE   R3, M[Z]
```

Clock cycles / Instructions	1	2	3	4	5	6	7
LOAD R1	I	A	D				
LOAD R2		I	A	D			
LOAD R4			I	A	D		
ADD R1,R2,R3				I	A		
STORE R3					I	A	D

This instruction is moved by the compiler

R2 is updated

Different clock cycles

R2 is used as operand

The performance is improved: 7 clock cycles (instead of 8 cycles in the solution 1).
The behavior of the program is not changed.

The Branch Problem

The branch (jump) instructions update the PC in the ALU (Execution) stage.
But the next instruction in sequence (not the target of branch) is fetched at the same time that jump is executed.
In this case, either a hardware unit must empty the pipeline or compiler-based solutions (delayed branch) must be applied.

Example: Pseudo Code

```
100 LOAD       M[X], R1
104 ADD        1, R2
108 JUMP       200
10C ADD        R1,R2
  ....
200  STORE     R1, M[Y]
```

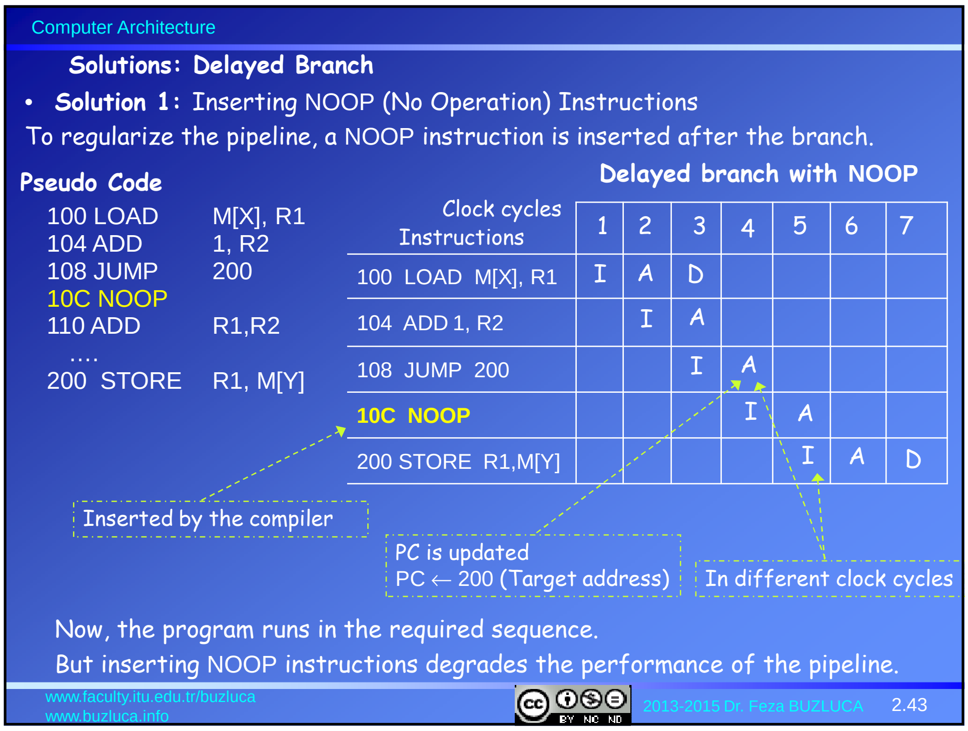
Without any precaution:

Clock cycles / Instructions	1	2	3	4	5	6	7
100 LOAD M[X], R1	I	A	D				
104 ADD 1, R2		I	A				
108 JUMP 200			I	A			
10C ADD R1,R2				I	A		
200 STORE R1,M[Y]					I	A	D

This instruction should be skipped.
The pipeline must be emptied by hardware or a compiler-bases solution must be applied.

PC is updated
PC ← 200 (Target address)

Problem: This instruction has been already fetched. Actually it should not run!



Solutions: Delayed Branch

- **Solution 1:** Inserting NOOP (No Operation) Instructions

To regularize the pipeline, a NOOP instruction is inserted after the branch.

Pseudo Code

```
100 LOAD    M[X], R1
104 ADD     1, R2
108 JUMP    200
10C NOOP
110 ADD     R1,R2
  ....
200  STORE  R1, M[Y]
```

Delayed branch with NOOP

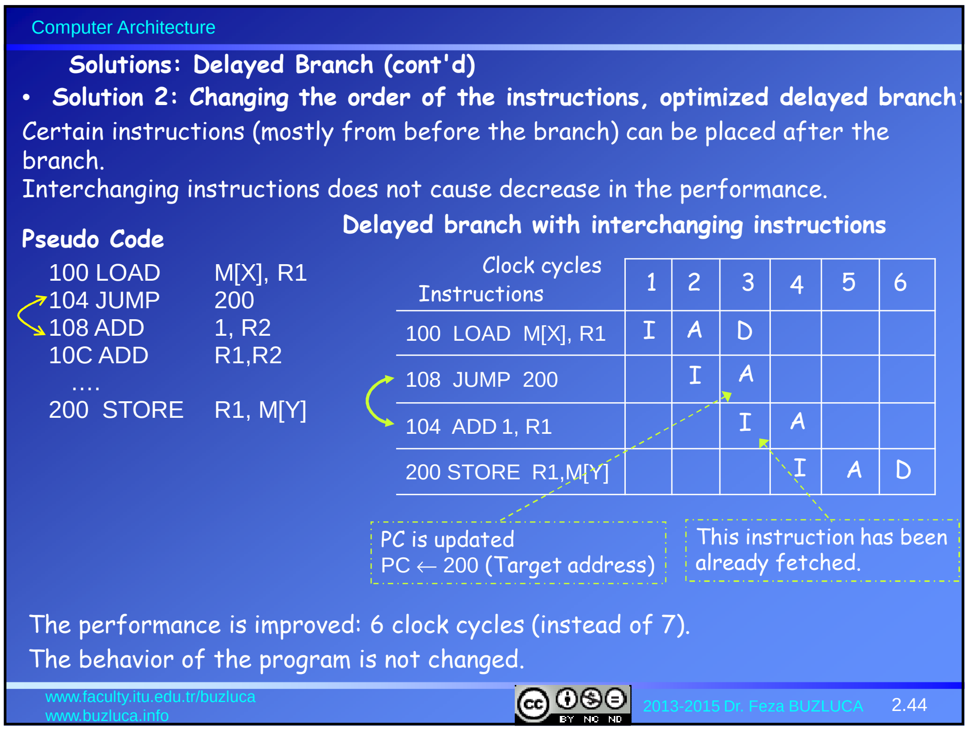
Clock cycles / Instructions	1	2	3	4	5	6	7
100 LOAD M[X], R1	I	A	D				
104 ADD 1, R2		I	A				
108 JUMP 200			I	A			
10C NOOP				I	A		
200 STORE R1,M[Y]					I	A	D

Inserted by the compiler

PC is updated
PC ← 200 (Target address)

In different clock cycles

Now, the program runs in the required sequence.
But inserting NOOP instructions degrades the performance of the pipeline.



Solutions: Delayed Branch (cont'd)

- **Solution 2: Changing the order of the instructions, optimized delayed branch:**

Certain instructions (mostly from before the branch) can be placed after the branch.
Interchanging instructions does not cause decrease in the performance.

Pseudo Code

```
100 LOAD    M[X], R1
104 JUMP    200
108 ADD     1, R2
10C ADD     R1,R2
  ....
200  STORE  R1, M[Y]
```

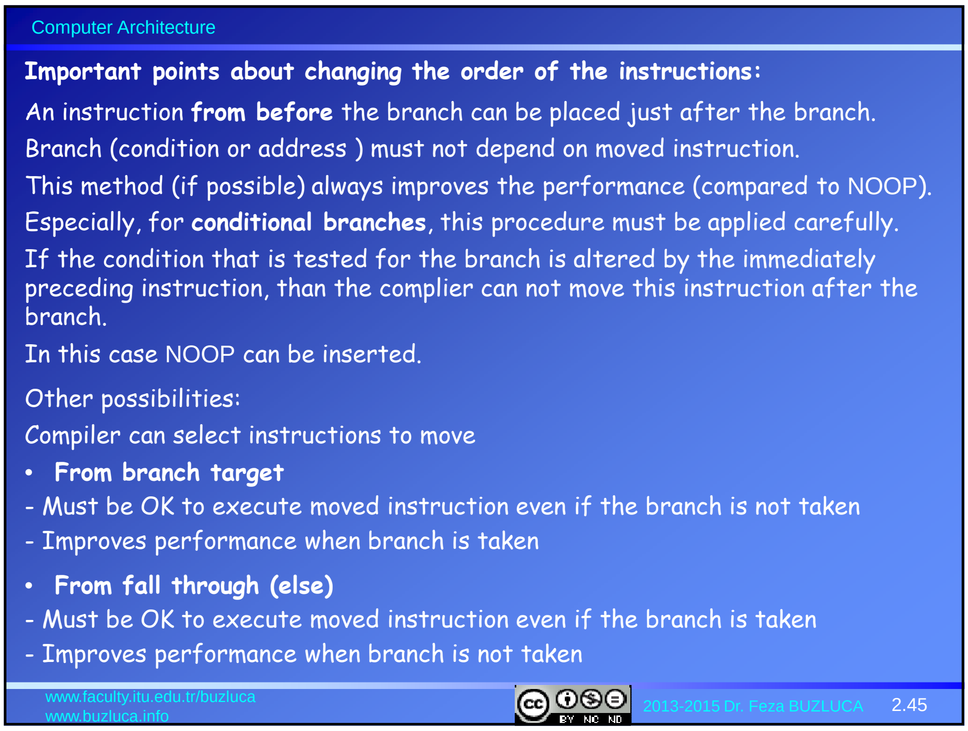
Delayed branch with interchanging instructions

Clock cycles / Instructions	1	2	3	4	5	6
100 LOAD M[X], R1	I	A	D			
108 JUMP 200		I	A			
104 ADD 1, R1			I	A		
200 STORE R1,M[Y]				I	A	D

PC is updated
PC ← 200 (Target address)

This instruction has been already fetched.

The performance is improved: 6 clock cycles (instead of 7).
The behavior of the program is not changed.



Important points about changing the order of the instructions:

An instruction **from before** the branch can be placed just after the branch.

Branch (condition or address) must not depend on moved instruction.

This method (if possible) always improves the performance (compared to NOOP).

Especially, for **conditional branches**, this procedure must be applied carefully.

If the condition that is tested for the branch is altered by the immediately preceding instruction, than the complier can not move this instruction after the branch.

In this case NOOP can be inserted.

Other possibilities:

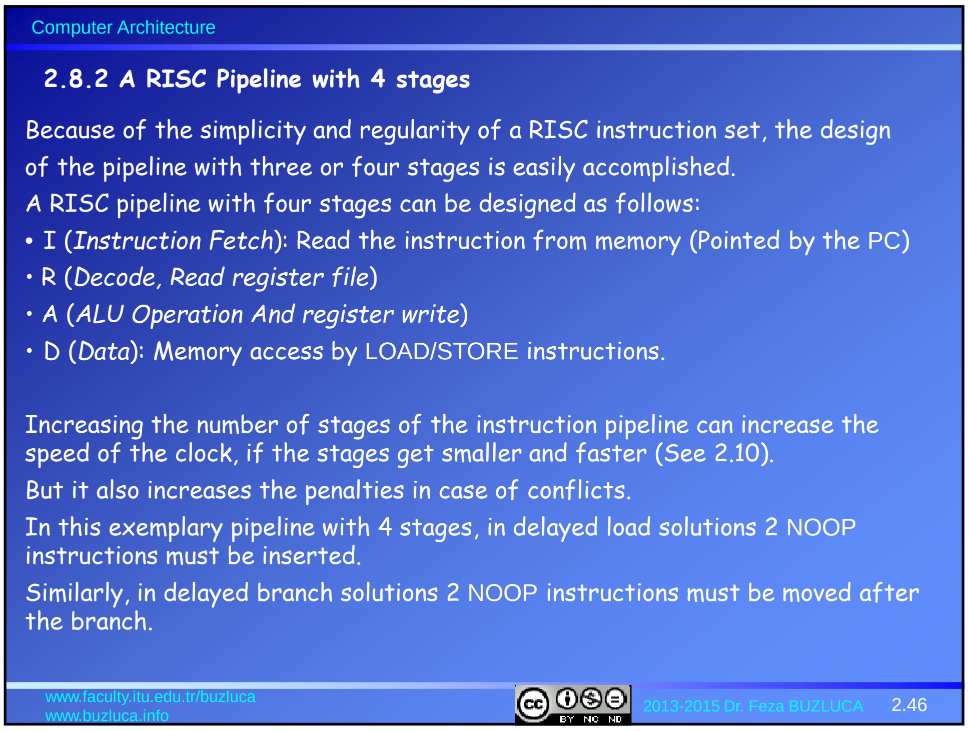
Compiler can select instructions to move

- **From branch target**

- Must be OK to execute moved instruction even if the branch is not taken
- Improves performance when branch is taken

- **From fall through (else)**

- Must be OK to execute moved instruction even if the branch is taken
- Improves performance when branch is not taken



2.8.2 A RISC Pipeline with 4 stages

Because of the simplicity and regularity of a RISC instruction set, the design of the pipeline with three or four stages is easily accomplished.

A RISC pipeline with four stages can be designed as follows:

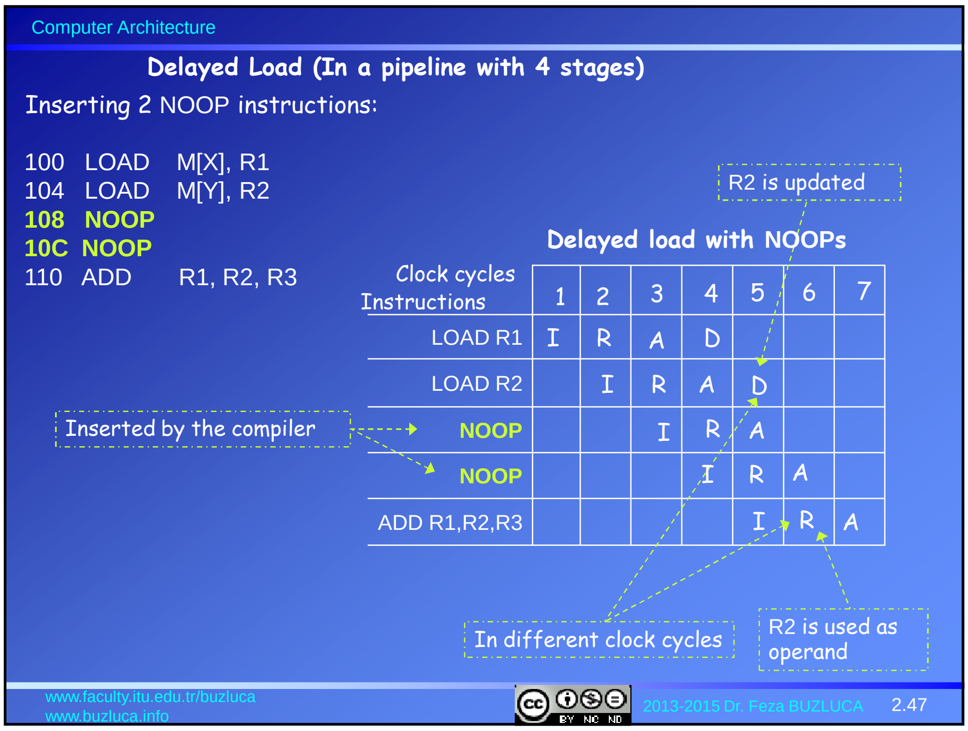
- I (*Instruction Fetch*): Read the instruction from memory (Pointed by the PC)
- R (*Decode, Read register file*)
- A (*ALU Operation And register write*)
- D (*Data*): Memory access by LOAD/STORE instructions.

Increasing the number of stages of the instruction pipeline can increase the speed of the clock, if the stages get smaller and faster (See 2.10).

But it also increases the penalties in case of conflicts.

In this exemplary pipeline with 4 stages, in delayed load solutions 2 NOOP instructions must be inserted.

Similarly, in delayed branch solutions 2 NOOP instructions must be moved after the branch.



Computer Architecture

Delayed Load (In a pipeline with 4 stages)

Inserting 2 NOOP instructions:

```
100  LOAD   M[X], R1
104  LOAD   M[Y], R2
108  NOOP
10C  NOOP
110  ADD    R1, R2, R3
```

Delayed load with NOOPs

Clock cycles / Instructions	1	2	3	4	5	6	7
LOAD R1	I	R	A	D			
LOAD R2		I	R	A	D		
NOOP			I	R	A		
NOOP				I	R	A	
ADD R1,R2,R3					I	R	A

R2 is updated

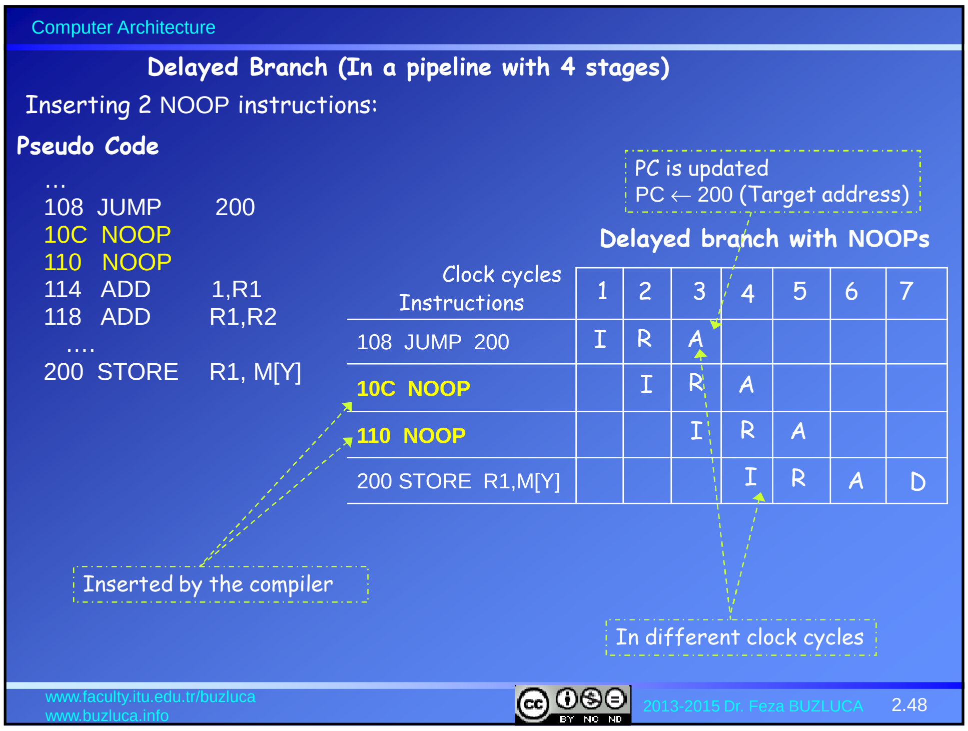
Inserted by the compiler

In different clock cycles

R2 is used as operand

www.faculty.itu.edu.tr/buzluca
www.buzluca.info

2.47



Computer Architecture

Delayed Branch (In a pipeline with 4 stages)

Inserting 2 NOOP instructions:

Pseudo Code

```
 ...
108  JUMP     200
10C  NOOP
110  NOOP
114  ADD      1,R1
118  ADD      R1,R2
  ....
200  STORE    R1, M[Y]
```

PC is updated
PC ← 200 (Target address)

Delayed branch with NOOPs

Clock cycles / Instructions	1	2	3	4	5	6	7
108 JUMP 200	I	R	A				
10C NOOP		I	R	A			
110 NOOP			I	R	A		
200 STORE R1,M[Y]				I	R	A	D

Inserted by the compiler

In different clock cycles

www.faculty.itu.edu.tr/buzluca
www.buzluca.info

2.48