# Instruction Cycle and Pipelining

## Chapter 14

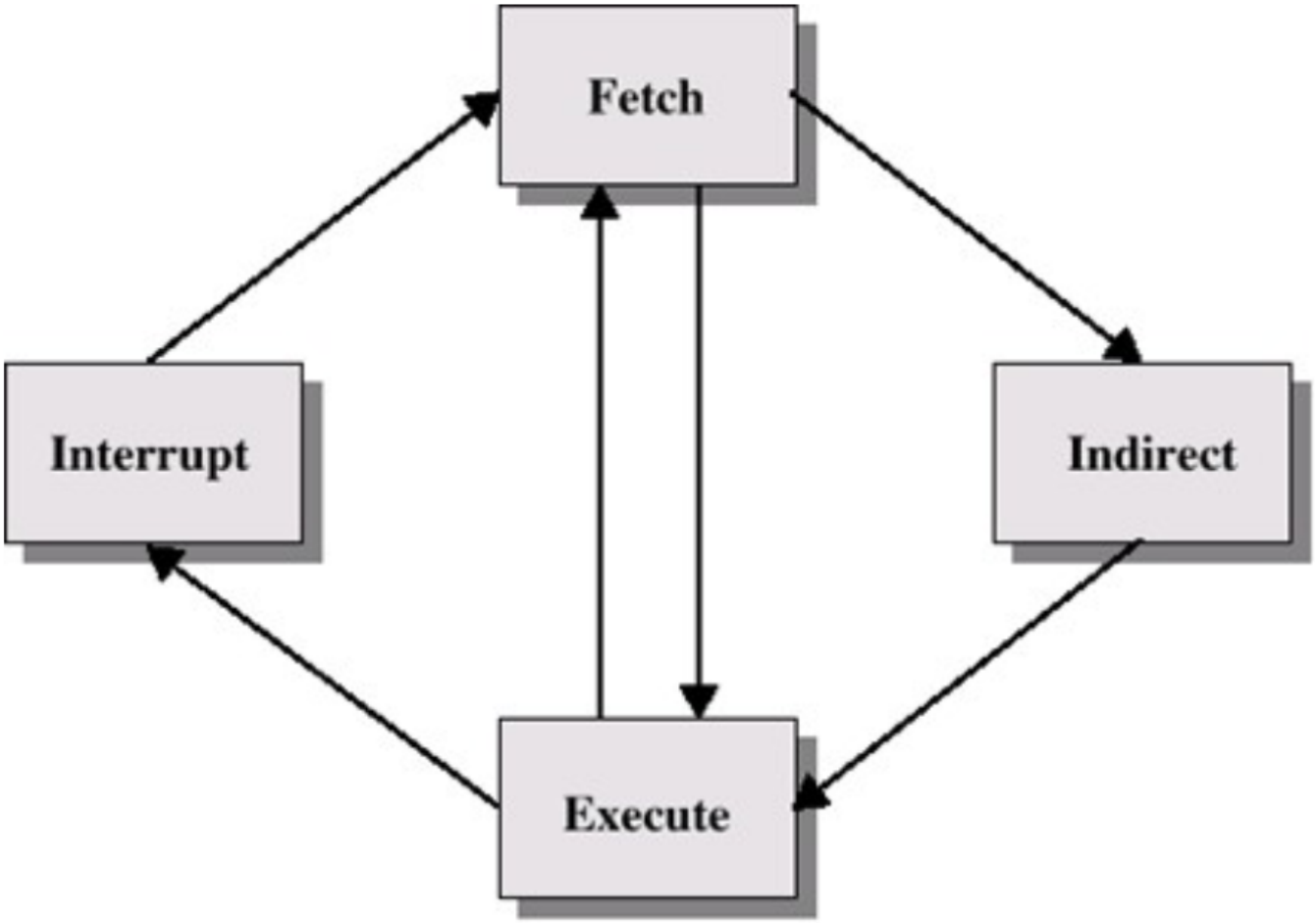
Joannah Nanjekye

July 31, 2024

# Administrative Things

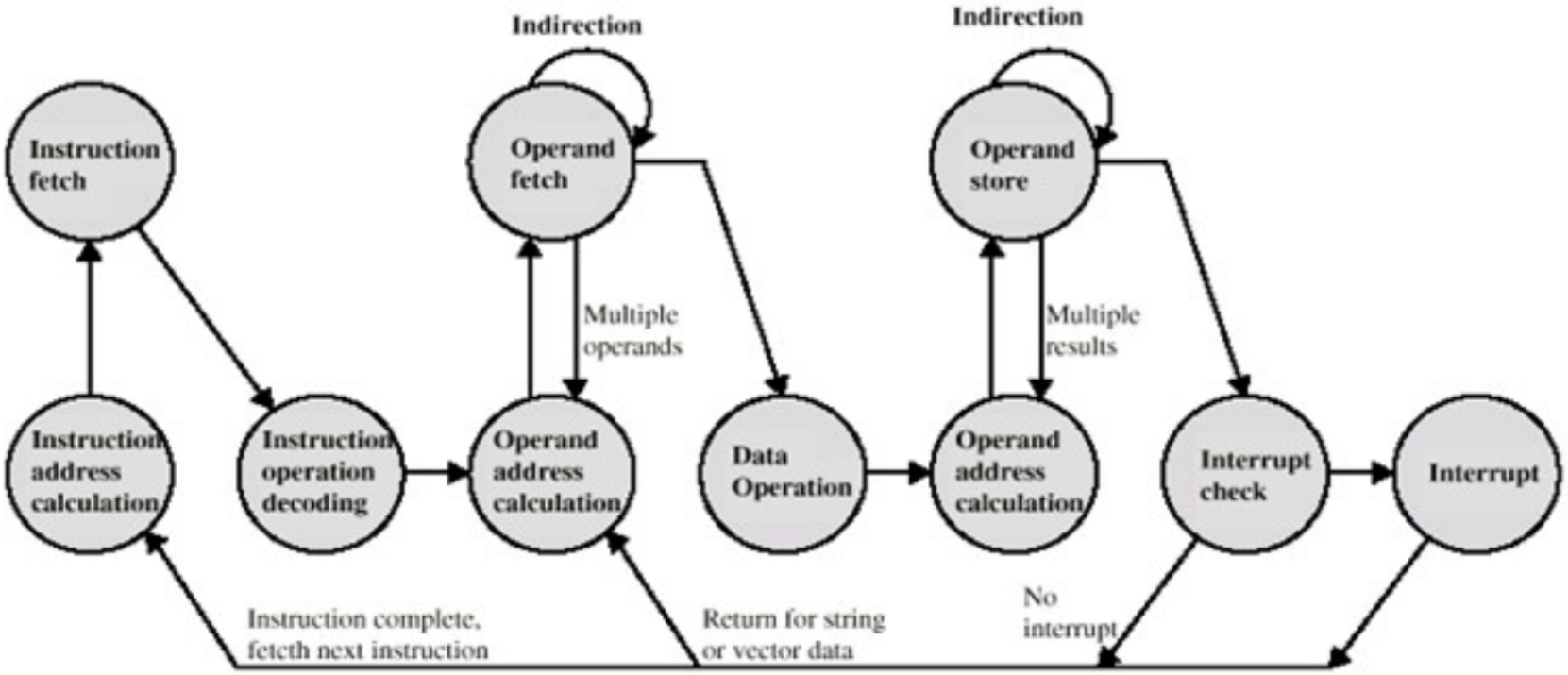
- Exam Scope: my slides
- Next Lab: Revision Lab
- Quiz 2 instructions:
  - Open book, no internet
  - No group discussion
  - Use lecture notes, your notes and the text book
  - Don't just throw a result for a computation (misunderstood as copying)
- Office hours by appointment only from now
- If you miss class a lot, first ask a friend before emailing me

# Instruction Cycle with Indirect Stage



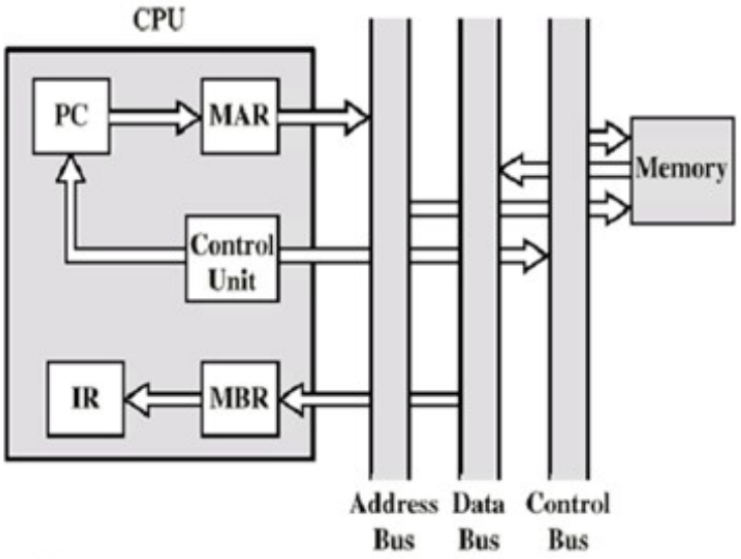

We need an Indirect Cycle for indirect addressing operands

# State Diagram



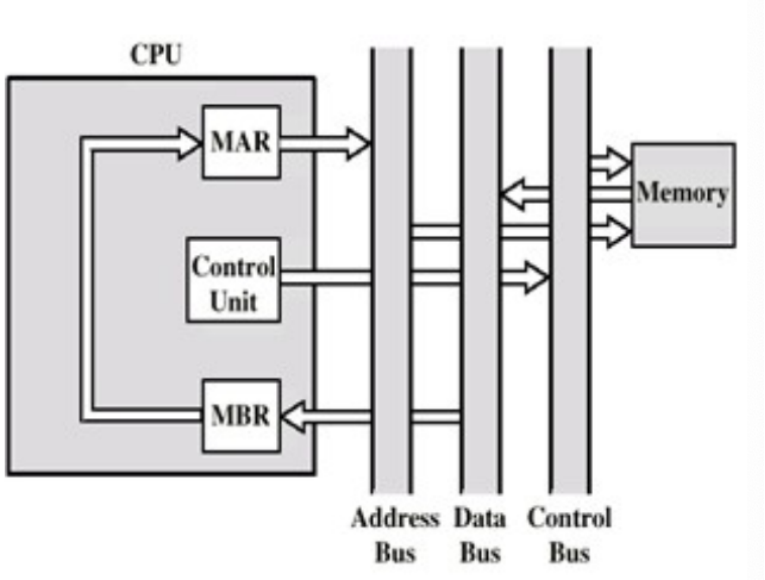
# Data Flow: Fetch Cycle

- PC contains address of the next instruction
- Address moved to MAR
- Address placed on address bus
- Control unit requests a memory read
- Result is placed on the data bus
- There is a copy to MBR then IR
- PC is incremented by 1



# Data Flow: Indirect Cycle

- IR is examined
- If indirect addressing, the indirect cycle is performed
  - Right most N bits of MBR transferred to MAR
  - Control unit requests a memory read
  - Result (address of operand) moved to MBR

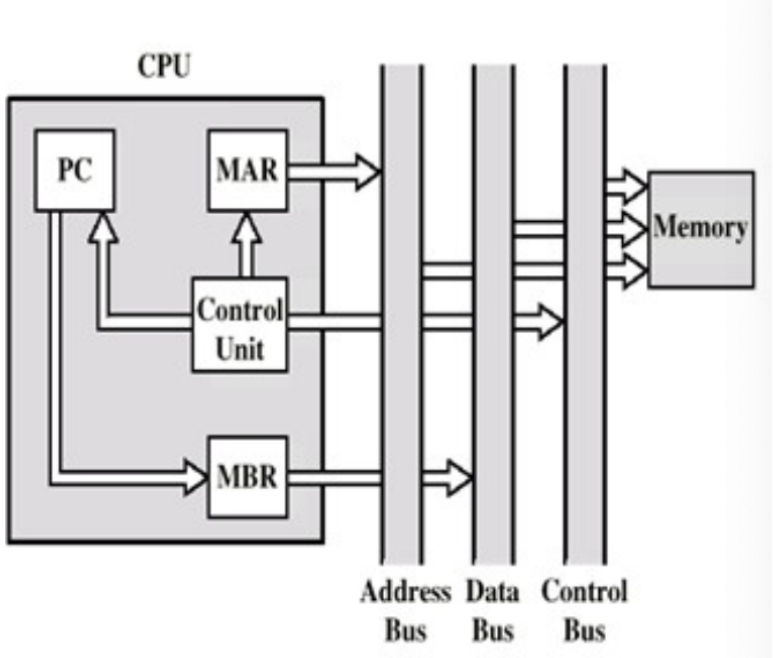


# Data Flow: Execute Cycle

- Depends on the instruction being executed
- A typical workflow includes:
  - Register transfers
  - Memory access
  - Input/output
  - ALU operations

# Data Flow: Interrupt Cycle

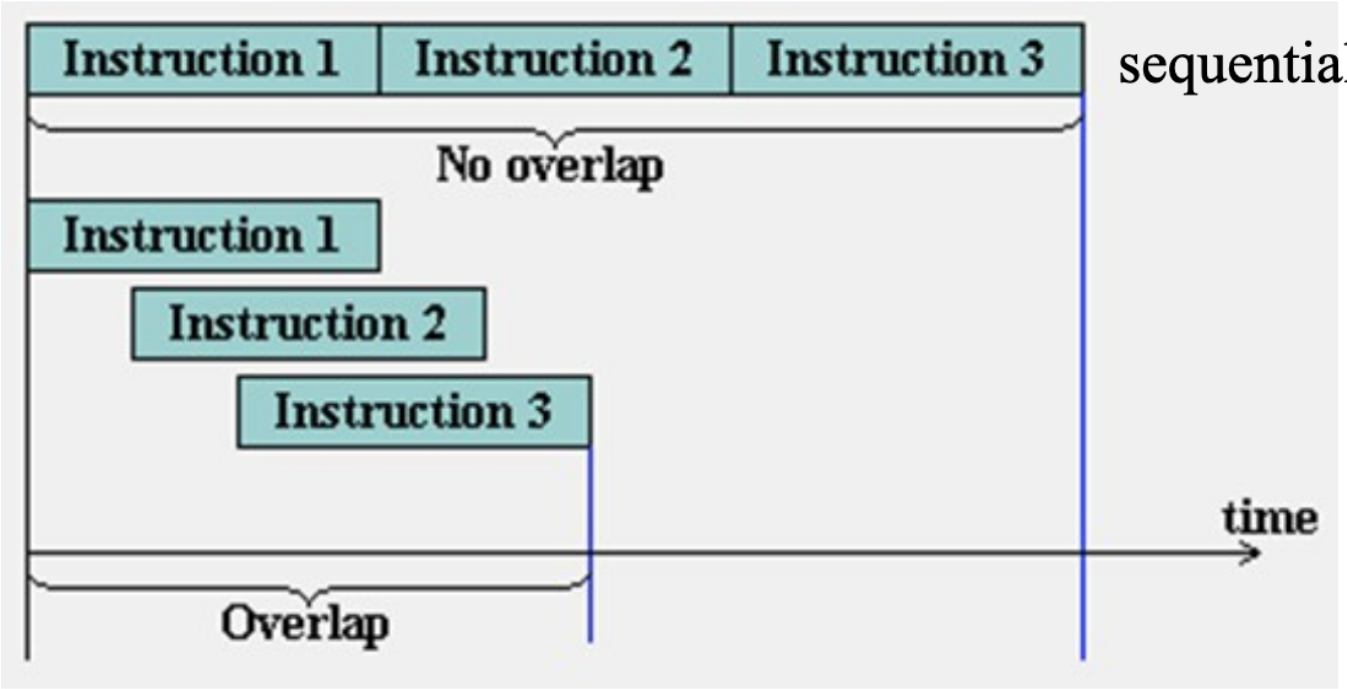
- IR is examined
- PC persisted to allow resumption after interrupt
  - Contents of PC copied to MBR
  - Special memory location (e.g stack pointer) is loaded to MAR
  - MBR written to memory
- PC loaded with address of ISR
- Next instruction can be fetched



# The Pipelining Strategy

- Tasks are subdivided into subtasks
- A pipeline stage is associated with each subtask
- The same amount of time is allocated to each subtask
- The first stage accepts input while the last stage delivers the output
- The basic pipeline is synchronous

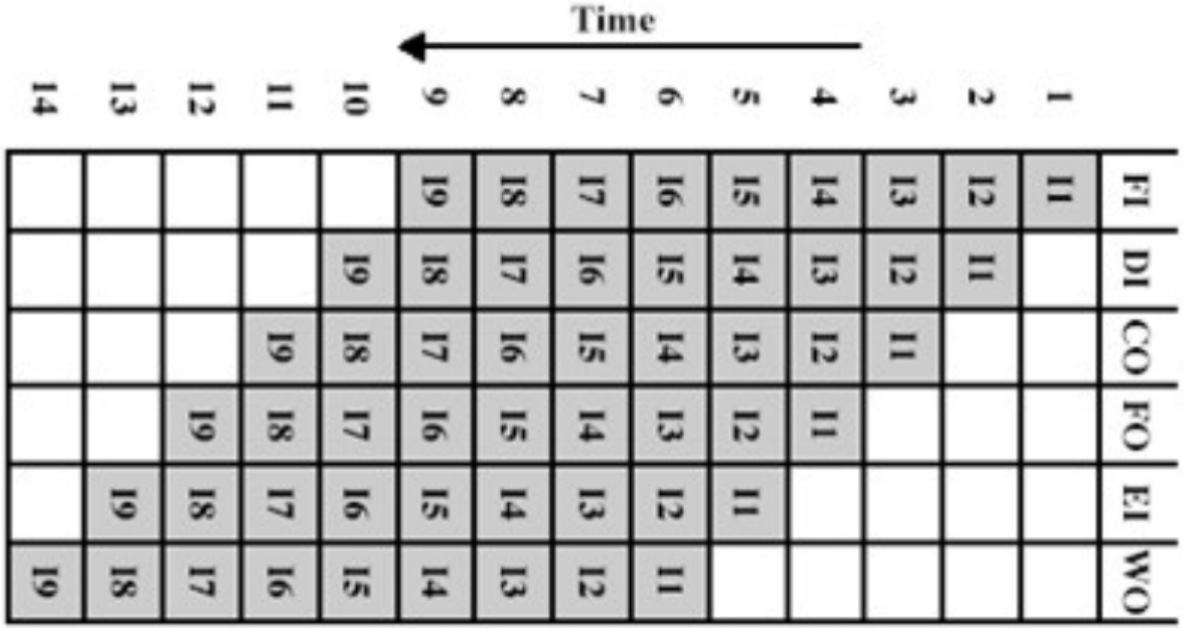
# Principles of Instruction Pipelining



# Instruction Pipelining

- An organizational optimization approach for the CPU
- There are 6 stages for instruction processing
  - Fetch instruction (FI)
  - Decode instruction (DI)
  - Calculate operands (CO)
  - Fetch operands (FO)
  - Execute instruction (EI)
  - Write operand / result (WO)
- Execution involves an overlap of instructions

# Timing of Instruction Pipelining



# Assumptions

- Each instruction goes through all the 6 stages
  - Not true e.g., no WO for 'LOAD'
  - Timing is setup for simplifying pipeline hardware
- No potential hazards
  - Data dependency
  - Branch
  - Interrupt
- No memory conflicts
  - Most systems don't allow simultaneous access
  - Desired data maybe else where e.g cache etc

# Pipeline Performance

- Cycle time, $\tau$
  - Time available for each stage to accomplish the required operations
  - Determined by the worst-case processing time of the longest stage
- Total time to execute *n* instructions
  - *k*, number of stages in the pipeline
  - Require *K cycles* to complete the first instruction
  - The remaining $n - 1$ instructions require $n - 1$ *cycles*

$$T_k = [k + (n - 1)]\tau$$

# Pipeline Performance ..

- Speedup Factor
  - Compared to the execution time without pipeline:
  - The higher the number of stages, the bigger the potential speedup

$$S_k = \frac{T_1}{T_k} = \frac{nk\tau}{[k+(n-1)]\tau} = \frac{nk}{k+(n-1)}$$

- Throughput
  - Refereed to as the ”repetition rate”
  - The shortest possible time interval between subsequent independent instructions in the pipeline
  - When the basic pipeline is full, throughput is 1 cycle

# Example

Consider a simple 6 stage pipeline executing a basic code block containing 20 instructions. Assume the pipeline clock cycle is 10ns and there is no potential hazard.

1. What is the total time to execute this block of code
   $T = (k + (n - 1)) \times c$
   $k = 6, n = 20, c = 10ns$
   $T = (6 + (20 - 1)) \times 10ns$
   $T = 250ns$
2. What is the repetition rate of this pipeline for this basic block
   1 cycle
3. What is the speedup factor
   Time without pipelining = $n \times k \times c = 6 \times 20 \times 10 = 1200ns$
   $Speedup = 1200/250$

# Pipeline Hazards

- A pipeline hazard occurs when a pipeline or some portion of the pipeline must stall due to conditions that do not permit further execution
- Such a pipeline stall is also called a pipeline bubble
- Types of hazards:
  - Resource
  - Data
  - Control

# Resource Hazards

- Also known as structural hazards
- Conflict for use of a resource
- Solution:
  - Hence pipelined data paths require separate instruction/data memories
  - Or separate instruction/data caches

# Resource Hazards ..

Clock cycle

| Instruction | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| I1 | FI | DI | FO | EI | WO | | | | |
| I2 | | FI | DI | FO | EI | WO | | | |
| I3 | | | FI | DI | FO | EI | WO | | |
| I4 | | | | FI | DI | FO | EI | WO | |

(a) Five-stage pipeline, ideal case

Clock cycle

| Instruction | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| I1 | FI | DI | FO | EI | WO | | | | |
| I2 | | FI | DI | FO | EI | WO | | | |
| I3 | | | Idle | FI | DI | FO | EI | WO | |
| I4 | | | | | FI | DI | FO | EI | WO |

(b) I1 source operand in memory

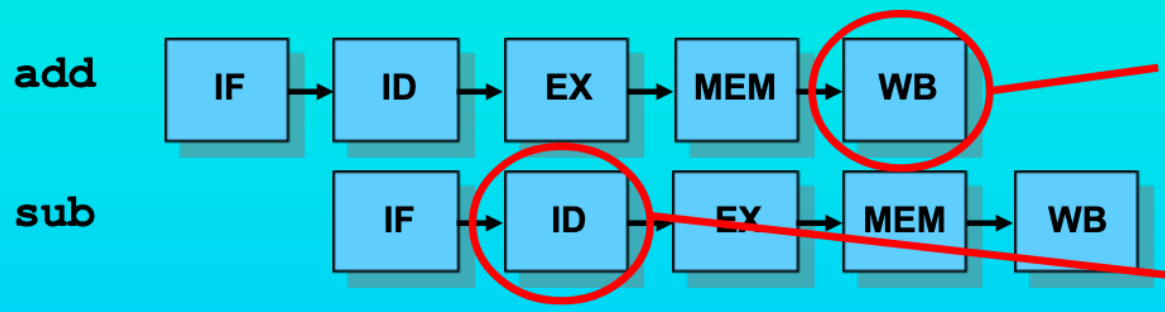
Example of Resource Hazard

# Data Hazards

Examine the following instructions:

```
add   $1, $3, $5
sub   $3, $1, $4
and   $2, $5, $1
or    $7, $1, $9
```

There is a dependency between add and sub on register $1 as it is used by sub after it is modified by add




The result of the add instruction is written in the $1 register NOT BEFORE the WB stage

However, the sub instruction fetches the value of register $1 during the ID stage

**Problem: The sub instruction will fetch the wrong value of register $1 because the correct value has not been written in there yet.**

# Types of Data Hazards

Three main categories

- Read after write (RAW)
- Read after read (RAR)
- Write after read (WAR)
- Write after write (WAW)

# RAW

```
add   $1, $3, $5
sub   $3, $1, $4
and   $2, $5, $1
or    $7, $1, $9
```

Read After Write (RAW) dependencies

It is the fact that some instructions have the same source register that is a destination in a previous instruction which means that the next instructions will need to read the value of this register while it is going to be written by the previous instruction

**Problem: The next instruction(s) will fetch the wrong values of the dependent registers because the correct values have not been written back yet.**

# RAW Cases

```
i:     add    $1, $3, $5
i+1:   sub    $3, $1, $4
i+2:   and    $2, $5, $1
i+3:   or     $7, $1, $9
```

Case 1 dependency between instruction *i* and instruction *i+1*

Case 2 dependency between instruction *i* and instruction *i+2*

Case 3 dependency between instruction *i* and instruction *i+3*

Every case needs to be checked in order to determined whether it poses a real problem or not

# RAR

```
add    $1, $3, $5
sub    $3, $5, $4
and    $2, $4, $1
or     $7, $1, $9
```

Read After Read (RAR) dependencies

Two consecutive instructions use the same register as a source operand

**<u>No Problem:</u> As long as the registers are not modified, pipelining does not affect the normal execution process in this case**

# WAW

```
add   $1, $3, $5
sub   $1, $5, $4
and   $4, $4, $1
or    $4, $1, $9
```

Write After Write (WAW) dependencies

Two consecutive instructions use the same register as a destination operand

**<u>No Problem:</u> Writes occur during the last pipeline stage and no inconsistency results from this situation because the instructions execution order is maintained**

# WAR

```
add   $1, $3, $5
sub   $3, $5, $2
and   $2, $4, $1
or    $7, $1, $9
```

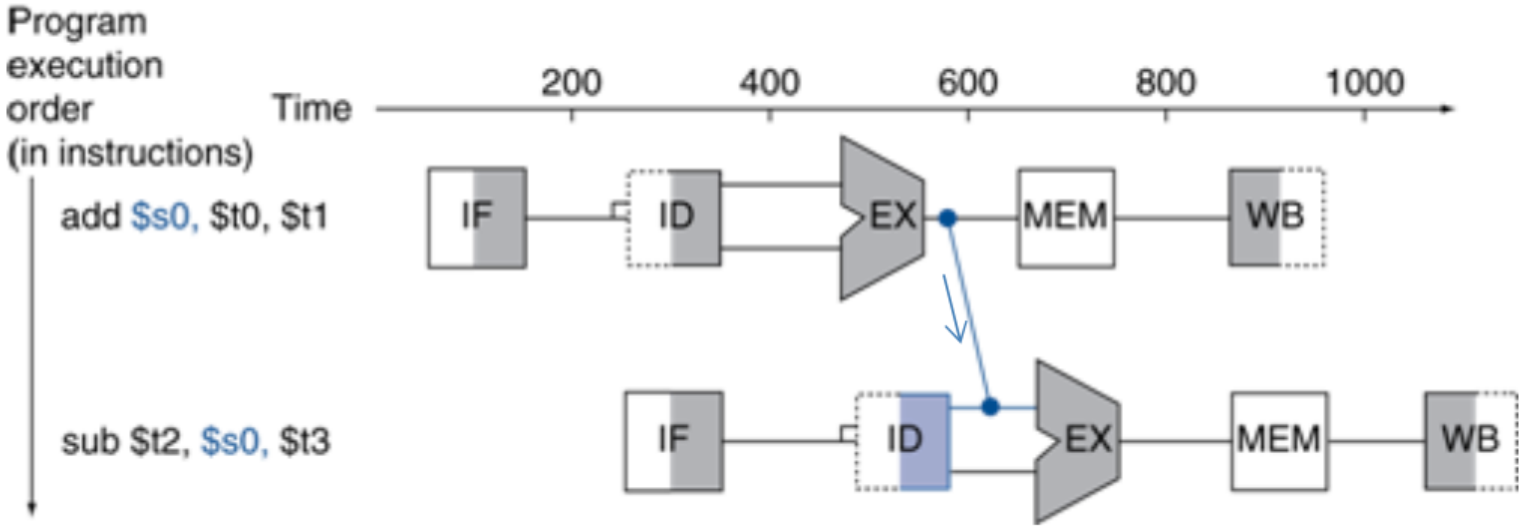
Write After Read (RAR) dependencies

The next instruction uses the same register, used as a source operand by a previous instruction, as destination register

**<u>No Problem:</u> Read occurs in ID stage and Write occurs in WB stage which means that the order of operations is not altered by the pipeline structure**

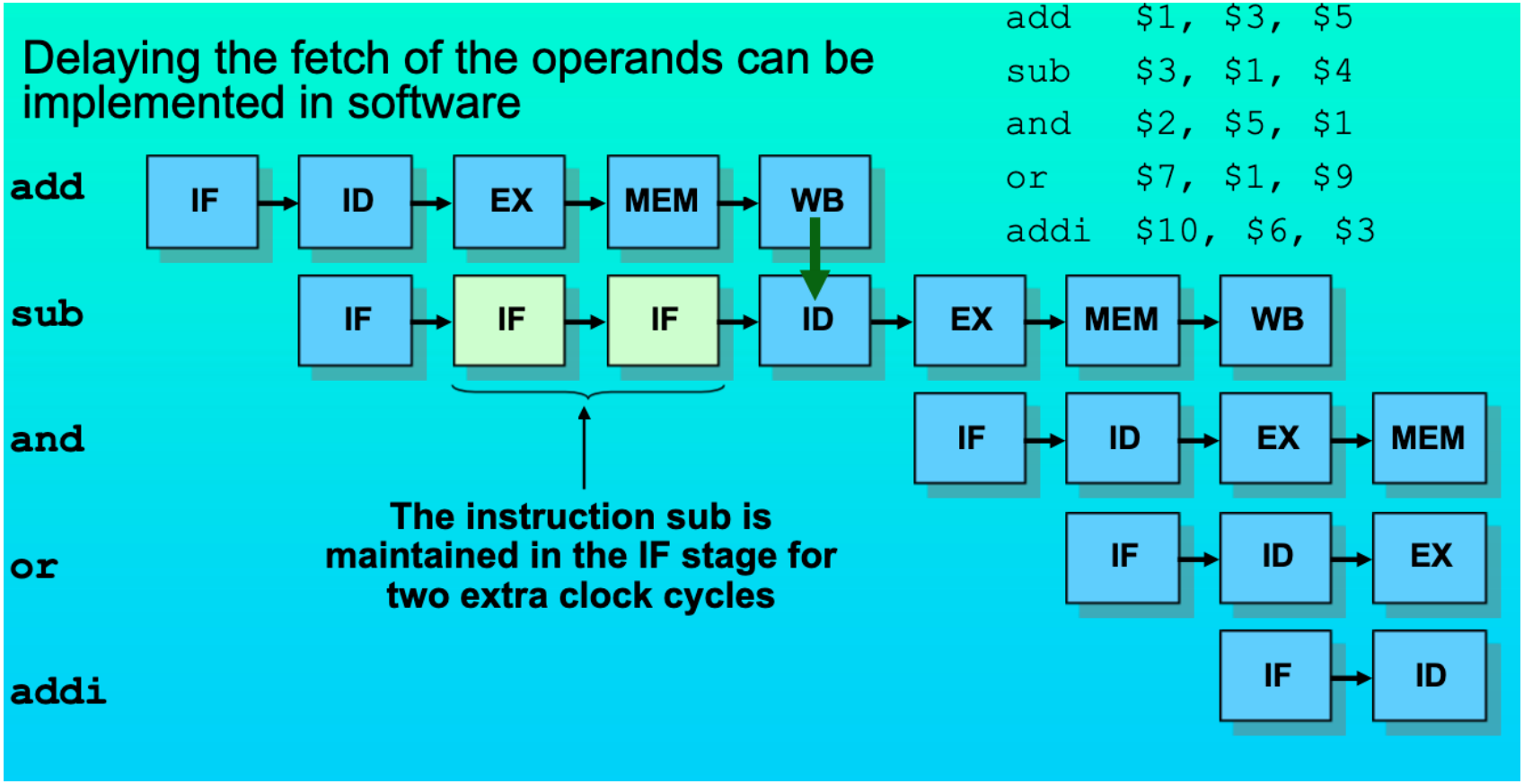
# Solution: Forwarding

- Use the result when it is computed
- Dont wait for it to be stored in a register
- Requires extra connections in the data path



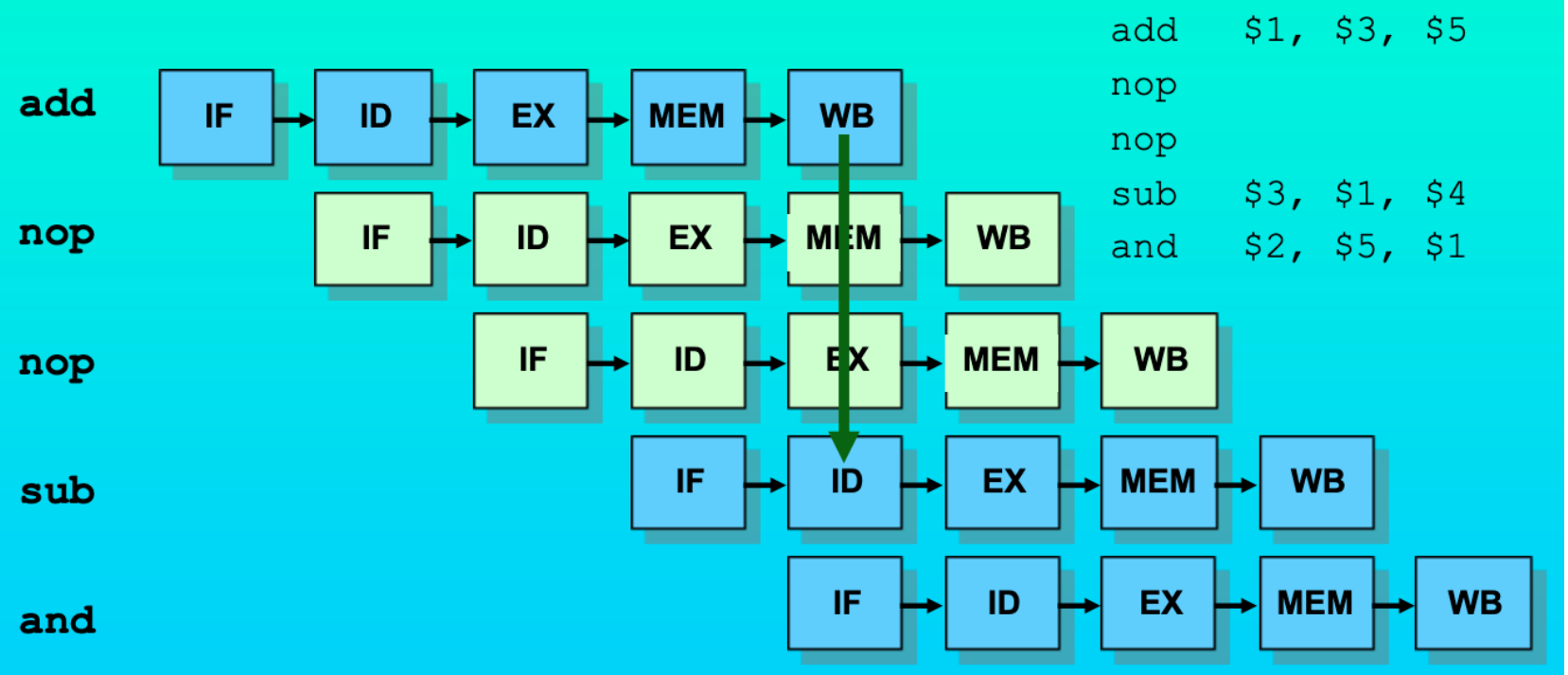
# Pipeline Stall

- Cant always use forwarding, hence stall
- Cant forward backward in time, if value isnt computed
- Instead delay or stall the pipeline



# NOP Insertion

Insertion of two NOP instructions will solve the data dependency problem



# Code Scheduling to Avoid Stall

- Reorder code to avoid use of load result in the next instruction

stall

stall

```
lw   $t1, 0($t0) #t1=B
lw   $t2, 4($t0) #t2=E
add  $t3, $t1, $t2  #B+E
sw   $t3, 12($t0) #t3=A
lw   $t4, 8($t0) #t4=F
add  $t5, $t1, $t4 #B+F
sw   $t5, 16($t0) #t5=C
```

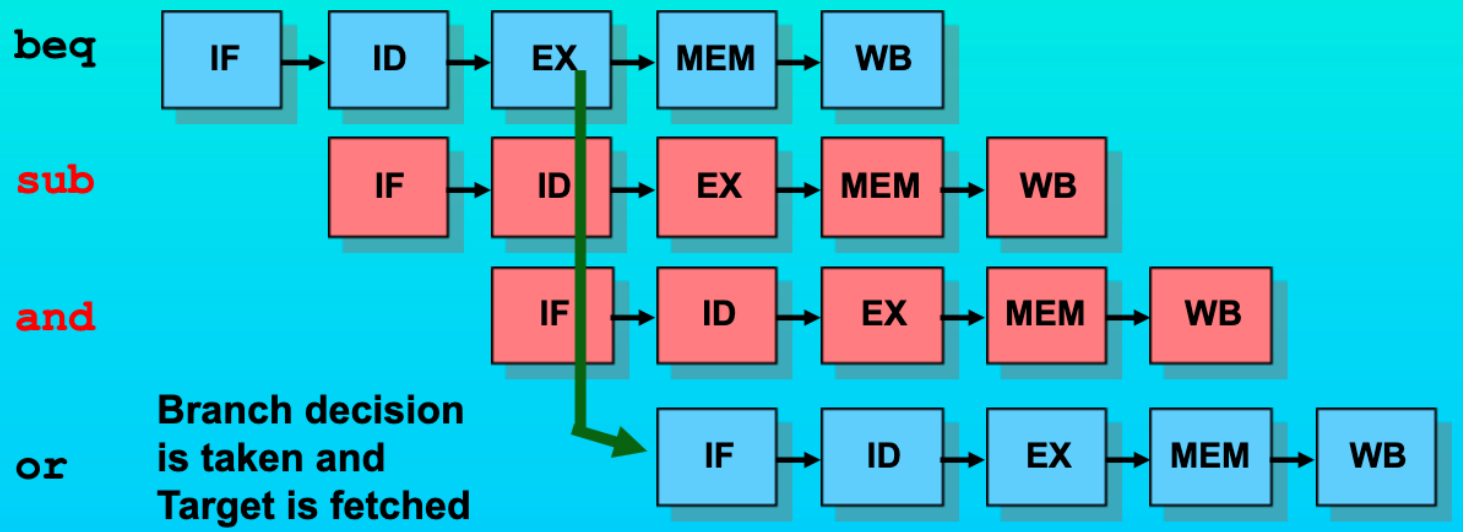
```
lw   $t1, 0($t0) #t1=B
lw   $t2, 4($t0) #t2=E
lw   $t4, 8($t0) #t3=F
add  $t3, $t1, $t2 #B+E
sw   $t3, 12($t0) #A
add  $t5, $t1, $t4 #B+F
sw   $t5, 16($t0) #>C
```

# Control Hazard

Examine the following instructions:

```
        beq     $1, $3, Target
        sub     $3, $1, $4
        and     $2, $5, $1
...
Target: or      $3, $5, $9
```

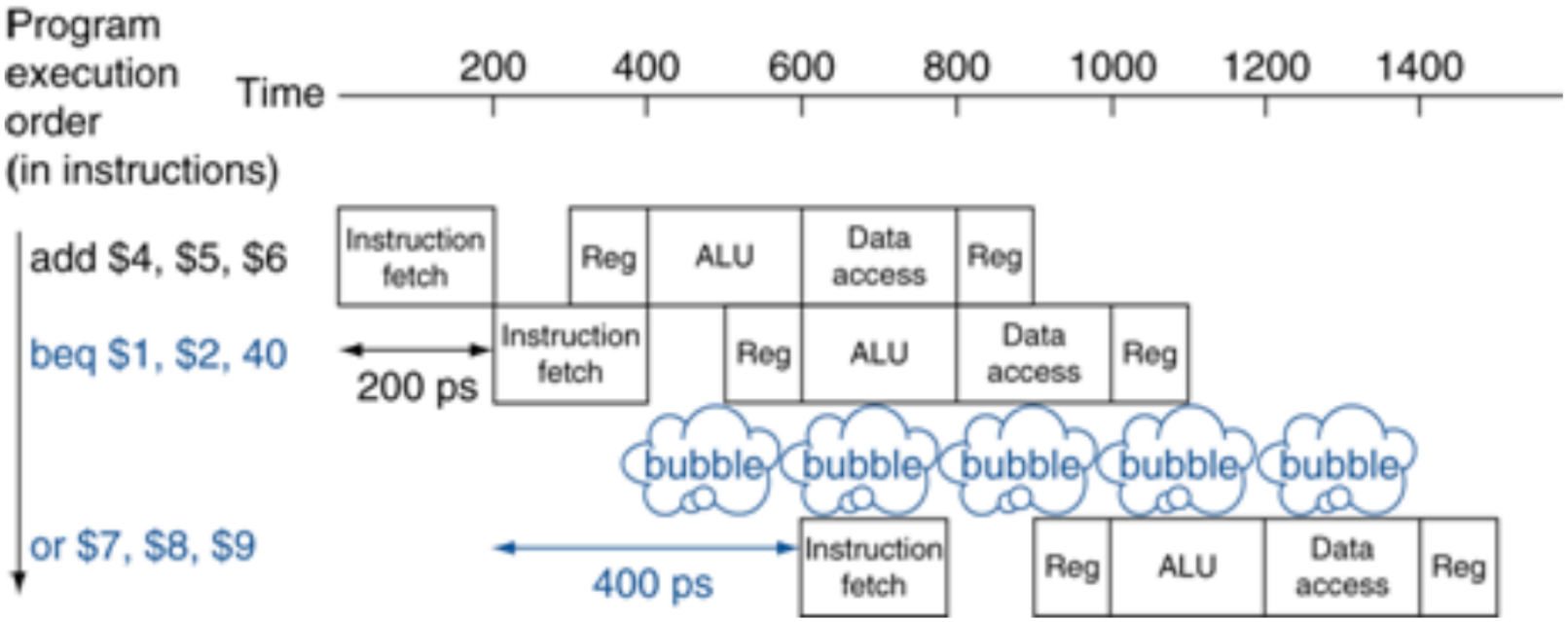
In the case the branch is taken, the instructions sub and add are wrongfully executed because they are fetched **BEFORE** the branch decision is made




**Problem: Modification of the Program Logic: Unacceptable Behavior**

# Stall on Branch

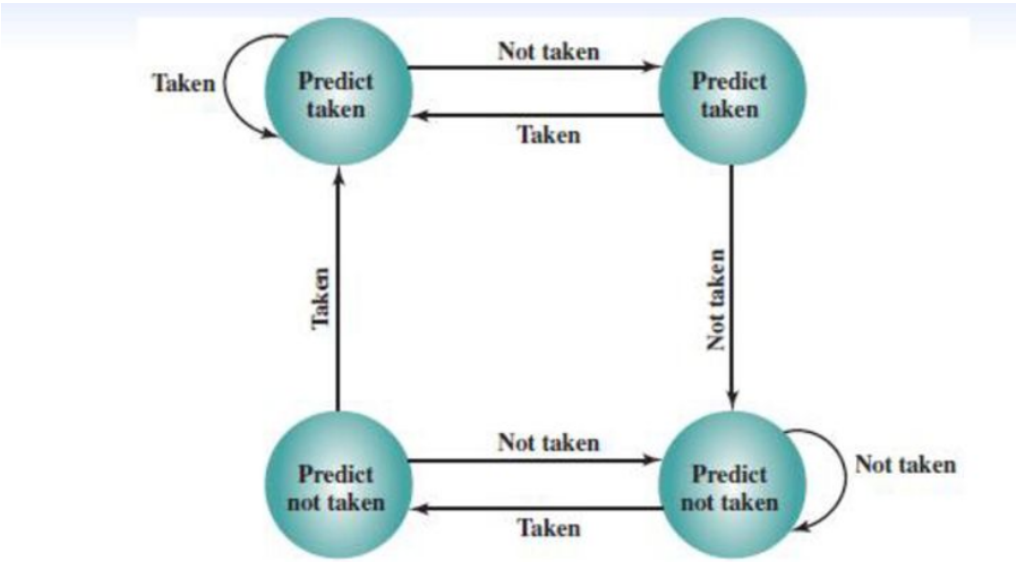
- Wait until the branch outcome is determined before next instruction fetch




Is this the best possible outcome?
Can you think of another alternative to avoid stalling?

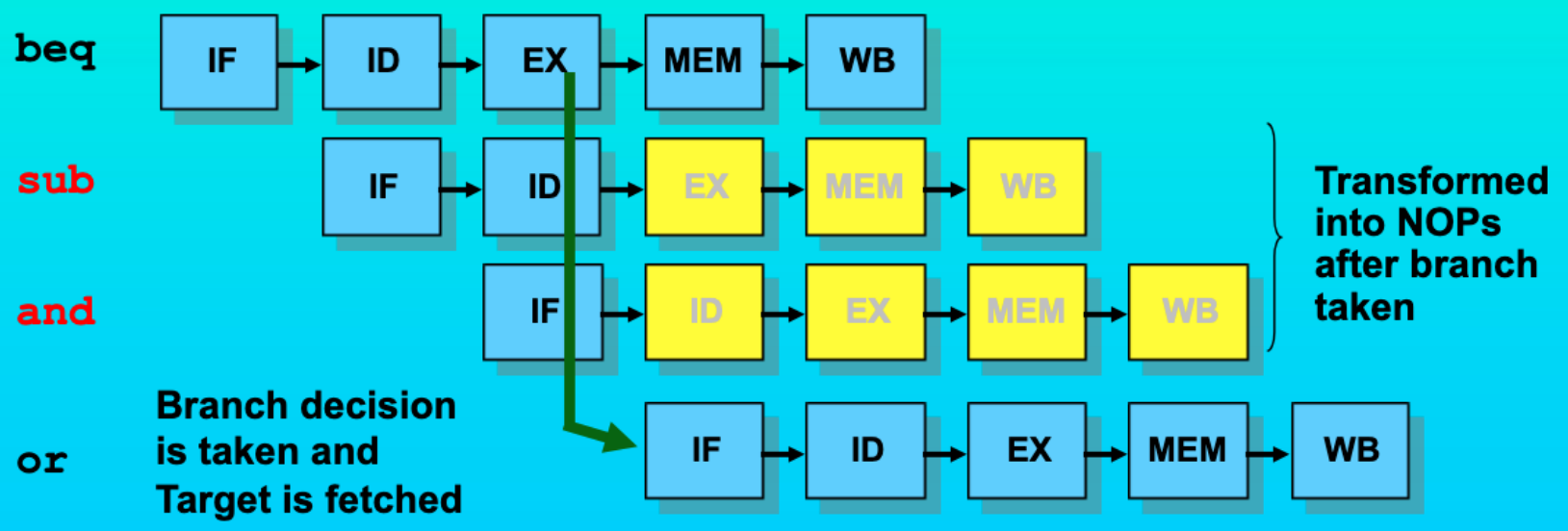
# Branch Prediction

- Predict never taken
- Predict always taken
- Predict by opecode
- Taken/Not taken switch
- he branch history table



# NOP Forcing

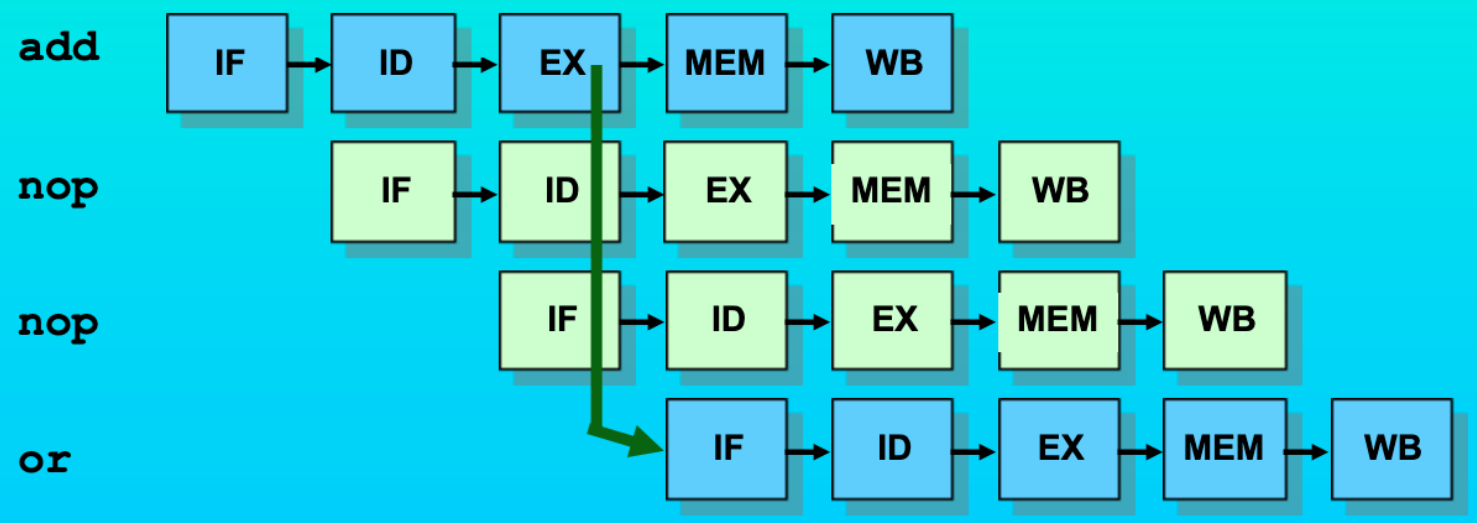
After branch is taken, following instruction are forced as NOP instructions for the subsequent pipeline stages until the branch target instruction is fetched. NOP will have no effect.
It is also said that instruction execution is ***killed***



# NOP Insertion

```
        beq     $1, $3, Target
        sub     $3, $1, $4
        and     $2, $5, $1
...
Target: or      $3, $5, $9
```

Insertion of NOP instructions, by the compiler, after each branch instruction, does not disturb the logic of the program.



# Delayed Branch

- Insertion of NOP instructions introduces a substantial overhead that increases the instruction count significantly.
- Idea is to move actual instructions from the area before the branch to the slots after the branch to fill in the nop slots without modifying the logic of the program

**Original code**

```
xor   $2, $2, $5     ← No dependency
and   $1, $7, $8     ← Register $1 used by beq
sub   $10, $6, $4    ← No dependency
add   $3, $6, $7     ← Register $3 used by beq
beq   $1, $3, Target
sub   $3, $1, $4
and   $2, $5, $1
```

**Transformed code**

```
and   $1, $7, $8
add   $3, $6, $7
beq   $1, $3, Target
xor   $2, $2, $5
sub   $10, $6, $4
sub   $3, $1, $4
and   $2, $5, $1
```

# Delayed Branch

Consider the transformed code obtained after moving the *xor* and *sub* instructions after the *beq* instruction:

```
and     $1, $7, $8
add     $3, $6, $7
beq     $1, $3, Target
xor     $2, $2, $5
sub     $10, $6, $4
sub     $3, $1, $4
and     $2, $5, $1
```

A programmer who reads the code <u>without</u> any idea about the execution will think that the branch occurs <u>*here*</u>

The execution will actually make the branch take effect <u>*here*</u>; so while the instructions *xor* and *sub* are executed, the second *sub* and the *and* instructions are not

Branch instruction and branch execution are sparated by a two instruction delay that's why it is called: ***Delayed Branch***

# Sources Acknowledgement

- https://slideplayer.com/slide/8447683/
- https://slideplayer.com/slide/5163573/
- https://slideplayer.com/slide/12934085/
- https://slideplayer.com/slide/3393101/