CS 21 - Computer Organization and Assembly Language Programming

Lecture 15
Pipelining

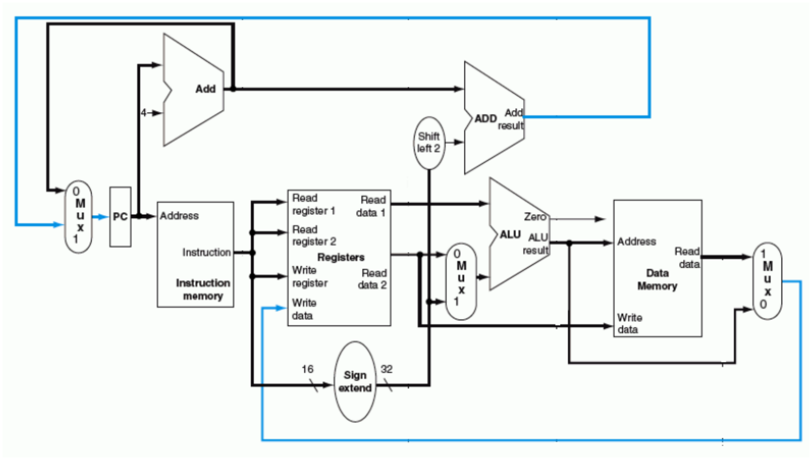
University of the Philippines - Diliman
College of Engineering
Department of Computer Science

Outline

Outline

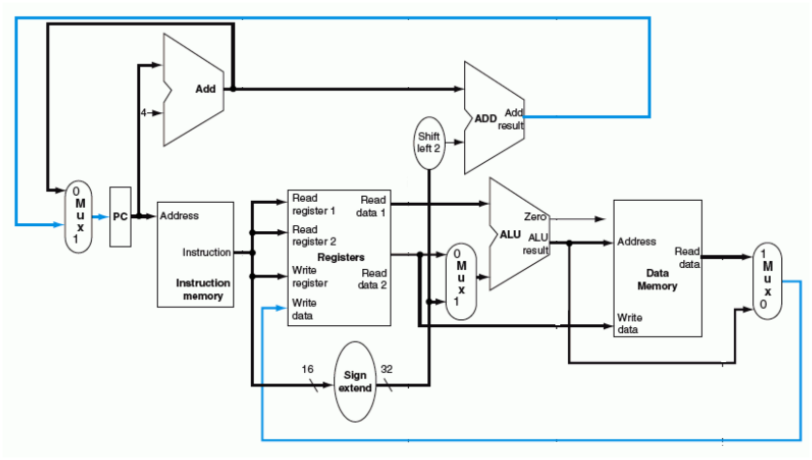
Question 1

What defines the operating frequency of a single cycle processor?



Question 1

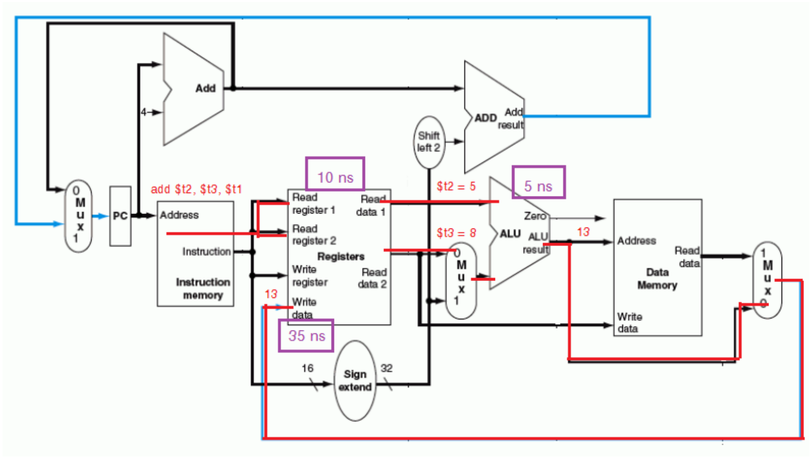
What defines the operating frequency of a single cycle processor?



The longest/slowest path.

Question 2

The longest path delay for this processor takes 50 ns. The register file produces its output after a delay of 10 ns. What can you say about the register file output value from t=10 ns to t=50 ns?



Truth about Single Cycle Processors

Most of the time, our components are not producing output...

Truth about Single Cycle Processors

Most of the time, our components are not producing output...

but just maintaining their output values

Could something else do that job?

Truth about Single Cycle Processors

Most of the time, our components are not producing output...

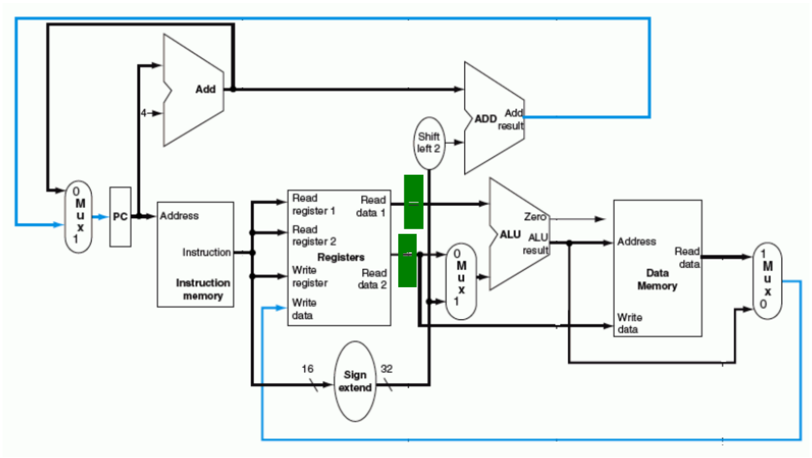
but just maintaining their output values

Could something else do that job?

YES.

Freeing up components

We could "free up" components from having to assert their values for long periods of time by adding registers.
Assuming that the greenboxes are registers, we could "free up" the registerfile by placing registers at its output



Facts

1. Most of the time, our components are not really active, they're just maintaining output values
   - And fortunately, registers could do that for us
2. There are instructions that don't rely on the previous instruction being totally finished

How do we use these to our advantage?

Facts

1. Most of the time, our components are not really active, they're just maintaining output values
   - And fortunately, registers could do that for us
2. There are instructions that don't rely on the previous instruction being totally finished

How do we use these to our advantage?

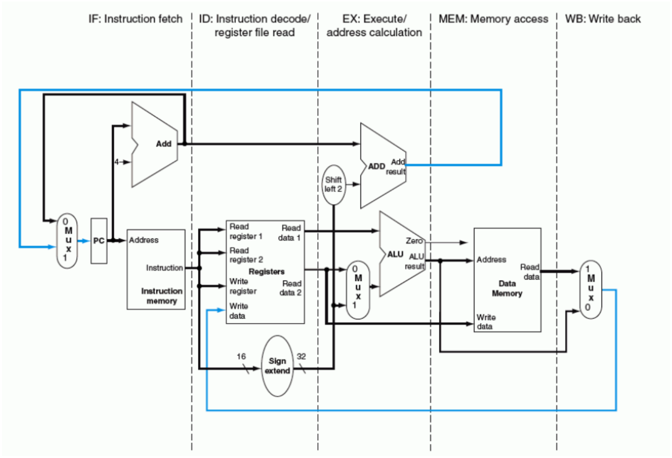
$\boxed{\text{Pipelining}}$

Outline

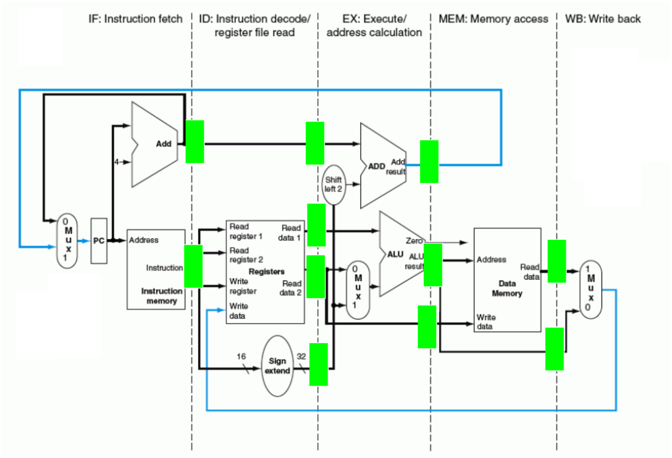
What is pipelining?

- We divide our processor into pipeline stages
- The output of our pipeline stage is stored in a register, accessed by the next pipeline stage
- Ideally, each pipeline stage should be processing one instruction at a time
- Question: Given a processor with n-pipeline stages, how many instructions could it be processing at any one time?

Dividing our processor into stages



Location of pipeline registers

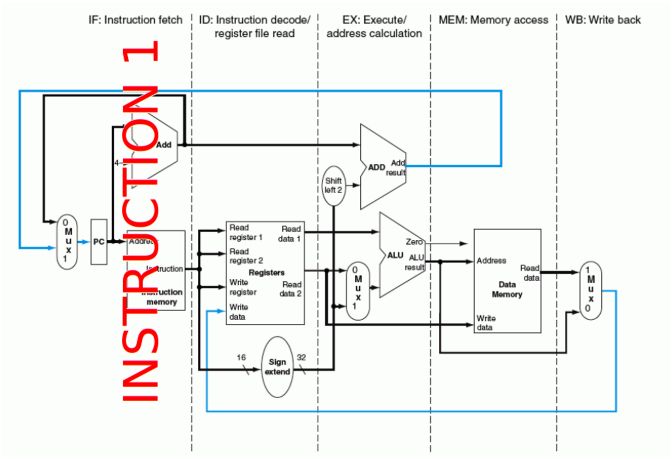


5 Basic Pipeline Stages

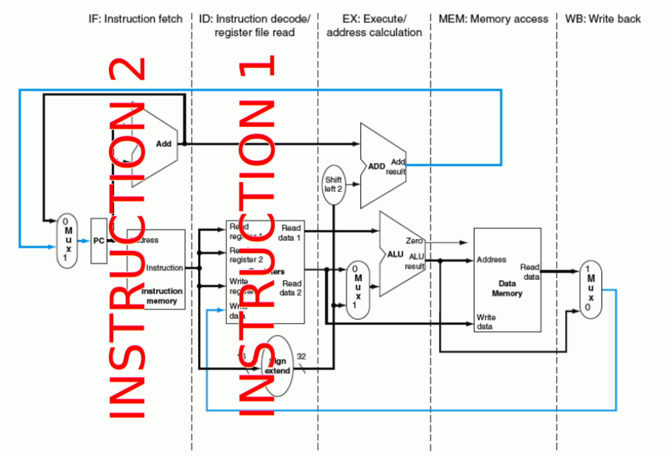
There are 5 basic pipeline stages in a processor

1. Instruction Fetch
2. Instruction Decode
3. Execute
4. Memory Access
5. Write Back

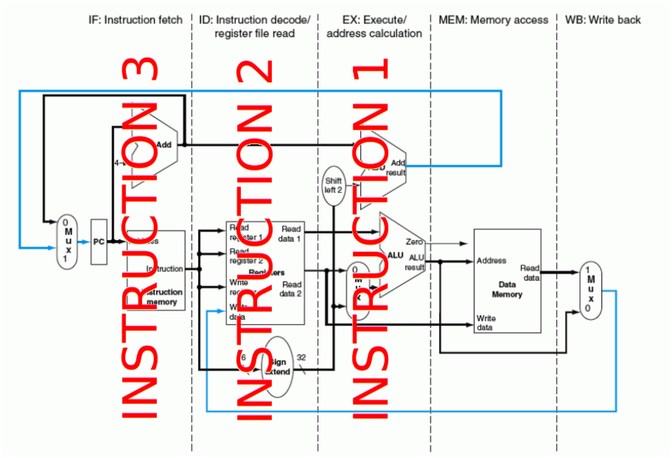
Pipelining in Action



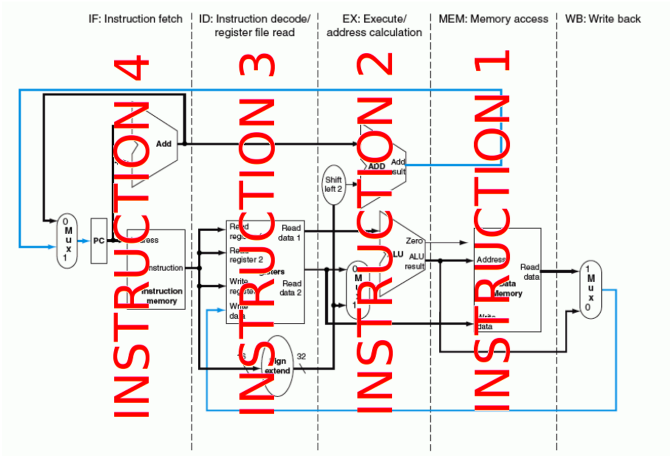
Pipelining in Action



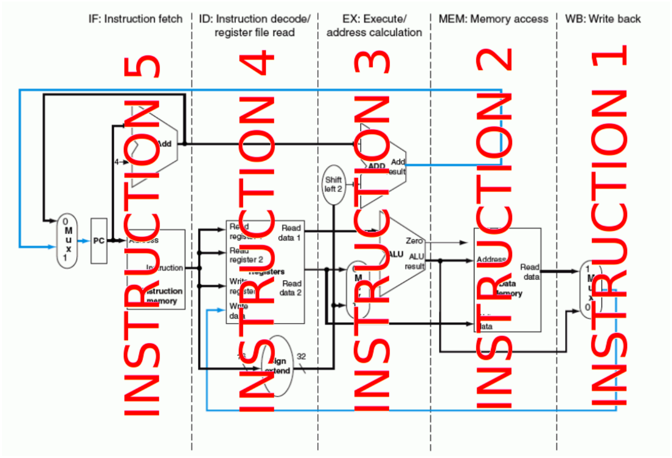
Pipelining in Action



Pipelining in Action

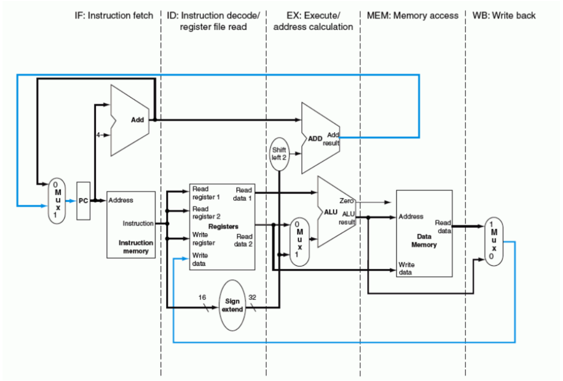


Pipelining in Action



Design Principles

Question: How do you divide a processor into pipelines?



Pipelining Guiding Principle 1

$\boxed{\text{Values that must be together, must be kept together}}$

Outline

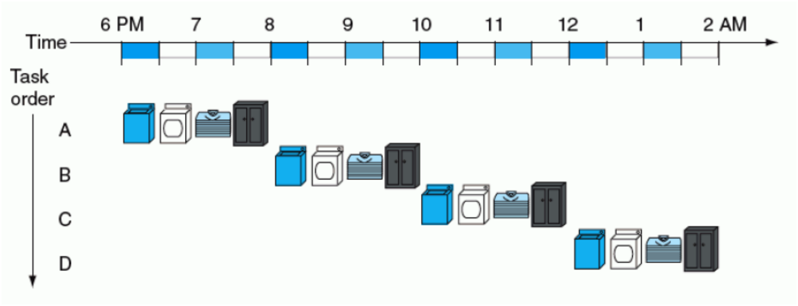
Laundry Analogy

Assume that we our laundry process has four tasks:

- washer
- dryer
- folder
- storer

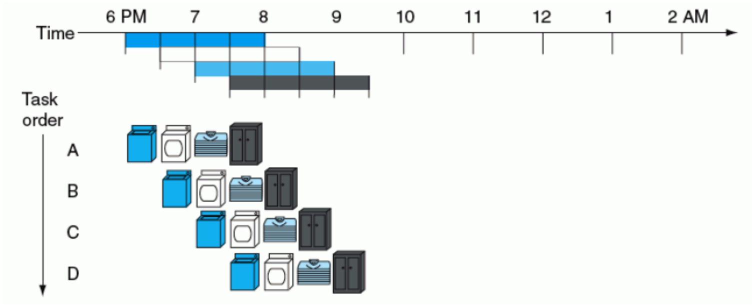
Laundry Analogy

Unpipelined



Laundry Analogy

Pipelined



Outline

Hazards

Hazard is a situation wherein the next instruction in the pipeline could not be executed immediately.

Three types:

- Structural Hazards
- Data Hazards
- Control Hazards

Structural Hazards

Structural Hazards occur when the hardware could not support the operation. Example

- Register only has one read port
- Register could either read or write at a time, not both.

Solution: Better circuit design!

Data Hazards

Data Hazards occur when the *data dependencies* prevent simultaneous execution in connected pipeline stages.

Example:

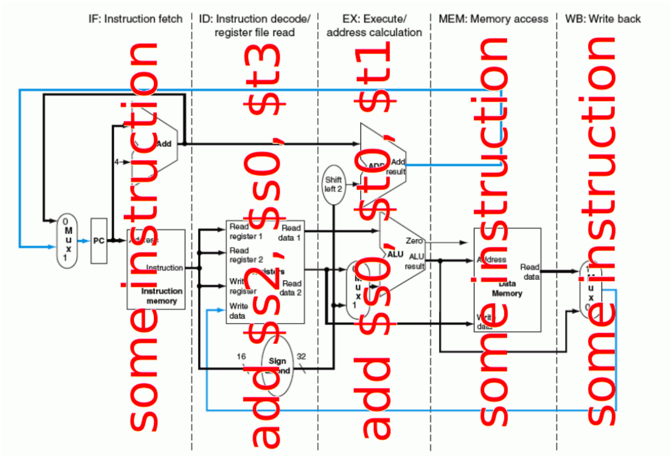
```
add $s0, $t0, $t1
add $s2, $s0, $t3
```

KEY Concept

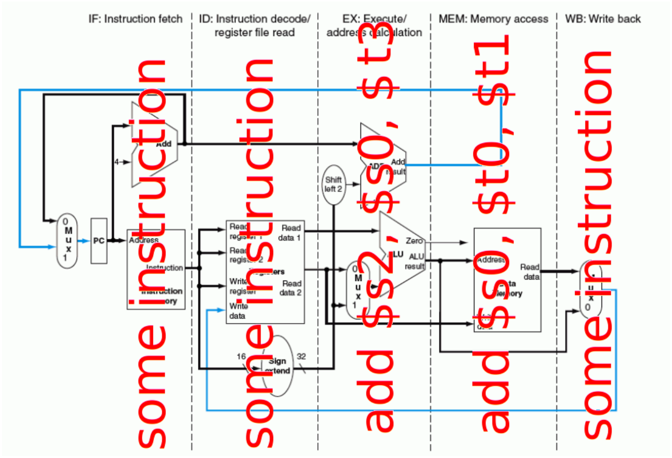
After passing through the Instruction Decode/Register File Read Stage, the instruction is already *carrying* with it its operands.

After passing through the Execution Stage, the instruction is already *carrying* the result of the computation.

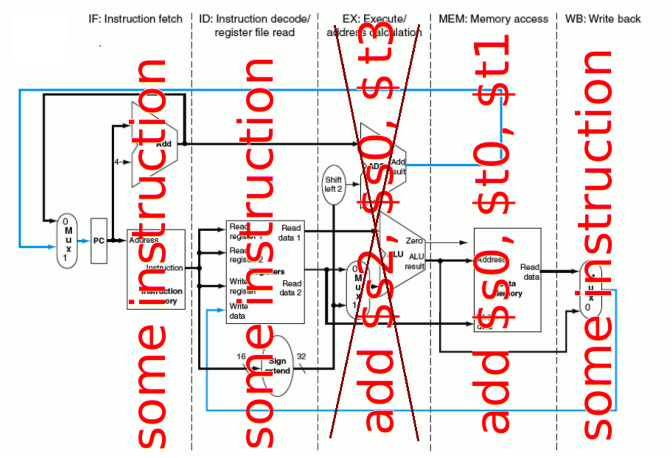
Data Hazard in Action



Data Hazard in Action



Data Hazard in Action



Why?

By the time the second instruction reaches the Execution Stage, it is already carrying *outdated* operands.

The right operand is with the first instruction, which hasn’t committed it yet.

Solutions/Workarounds to Data Hazards

1. Bubbles
2. Forwarding/bypassing

Solving Data Hazards: Bubbles

We just let the preceding instruction go ahead; "bubbles" are inserted between instructions. Example:

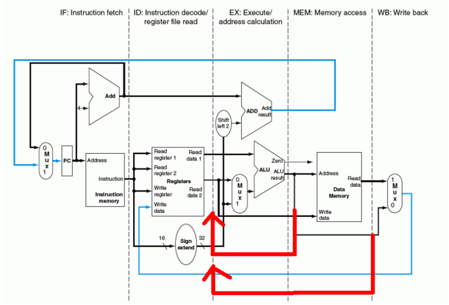
```
add $s0, $t0, $t1
    nop
    nop
    nop
add $s2, $s0, $t3
```

What is the advantage of this technique? Disadvantage?

Solving Data Hazards: Forwarding/Bypassing

We add circuits and datapaths that allow us to get operands from stages further down the pipeline, operands that have yet to be stored in the register file

Example:



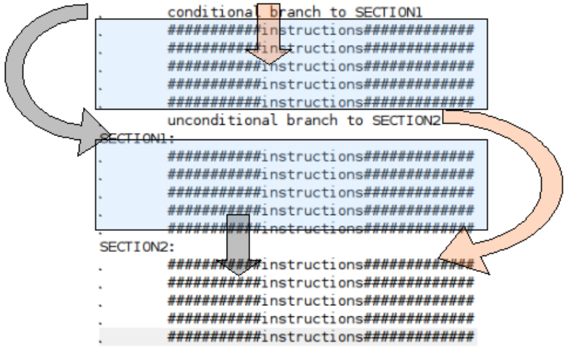
What is the advantage of this technique? Disadvantage?

Control Hazards

Control Hazards occur when we become uncertain of the instructions to fetch and execute because of branches.

```
beq $t0, $t1, then
else:
(some code here)
b after
then:
(some code here)
after:
```

Control Hazards

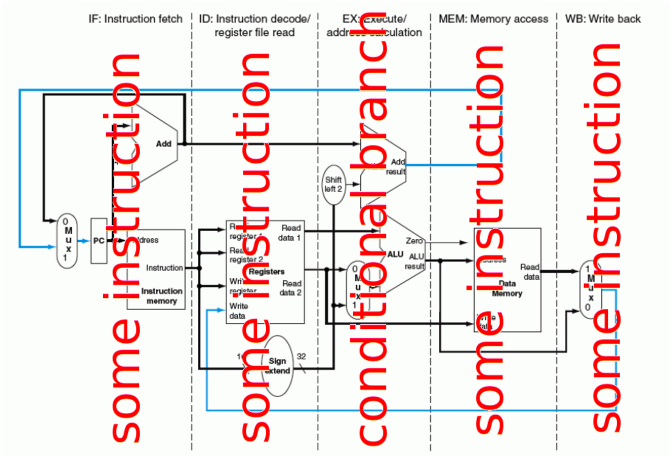


```
.        conditional branch to SECTION1
.        ###########instructions############
.        ###########instructions############
.        ###########instructions############
.        ###########instructions############
.        ###########instructions############
         unconditional branch to SECTION2
SECTION1:
.        ###########instructions############
.        ###########instructions############
.        ###########instructions############
.        ###########instructions############
.        ###########instructions############
SECTION2:
.        ###########instructions############
.        ###########instructions############
.        ###########instructions############
.        ###########instructions############
.        ###########instructions############
```

KEY Concept

Result of branches are not known until the execution stage

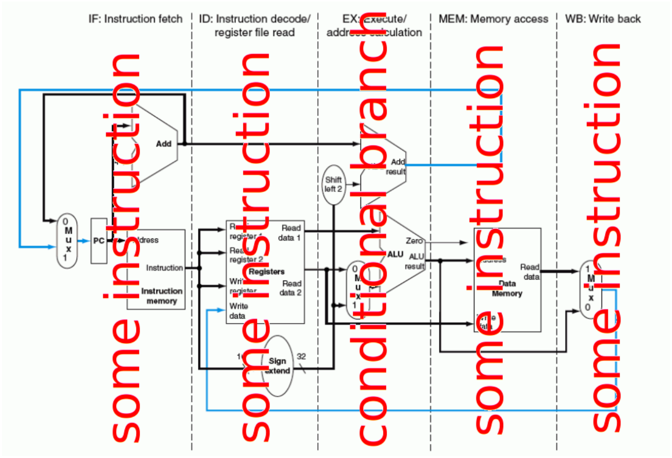
Control Hazard in Action



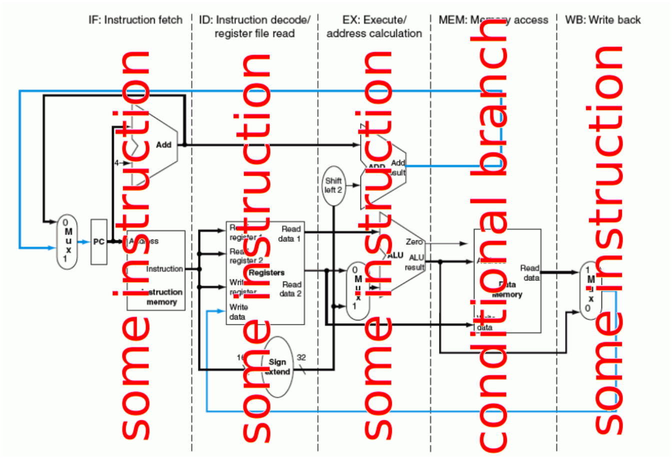
Possible Outcomes

1. Branch Not Taken
2. Branch Taken

Outcome: Branch Not Taken



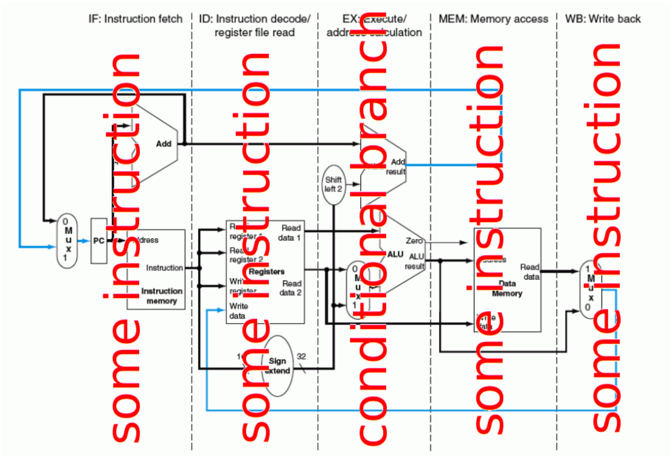
Outcome: Branch Not Taken



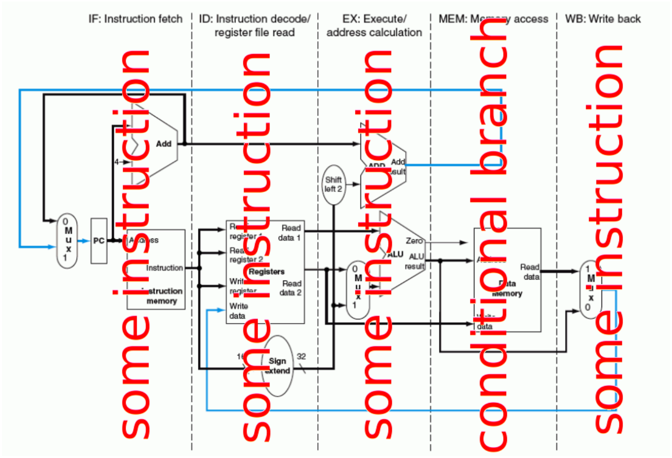
Outcome: Branch Not Taken

All is well, instruction execution continues like before.

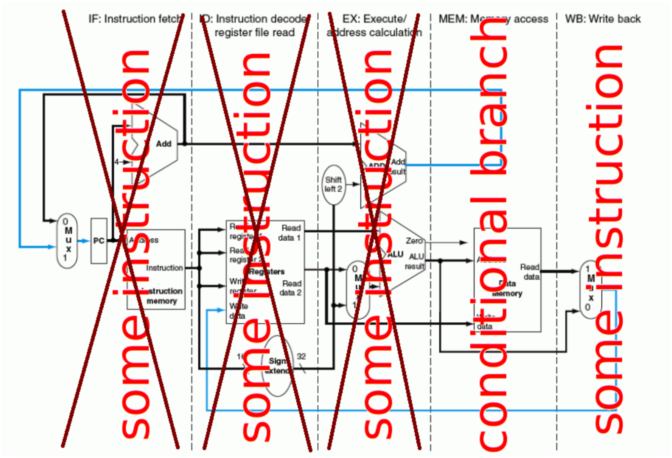
Outcome: Branch Taken



Outcome: Branch Taken



Outcome: Branch Taken



Outcome: Branch Taken

Instructions that should not be executed are already in the pipeline!

Solutions/Workarounds to Data Hazards

1. Stall
2. Predict/gamble

Outline

Remember our Equations

$$CPUExecutionTime = \frac{CPUclockcycles}{Clockrate}$$

$$CPUClockCycles = numberofinstructions * CPI$$

CPI

Which has a better CPI, a single-cycle processor or a pipelined processor?

CPI

Which has a better CPI, a single-cycle processor or a pipelined processor?

Pipelining actually increases/worsens the CPI

Remember our Equations

$$CPUExecutionTime = \frac{CPUclockcycles}{Clockrate}$$

$$CPUClockCycles = numberofinstructions * CPI$$

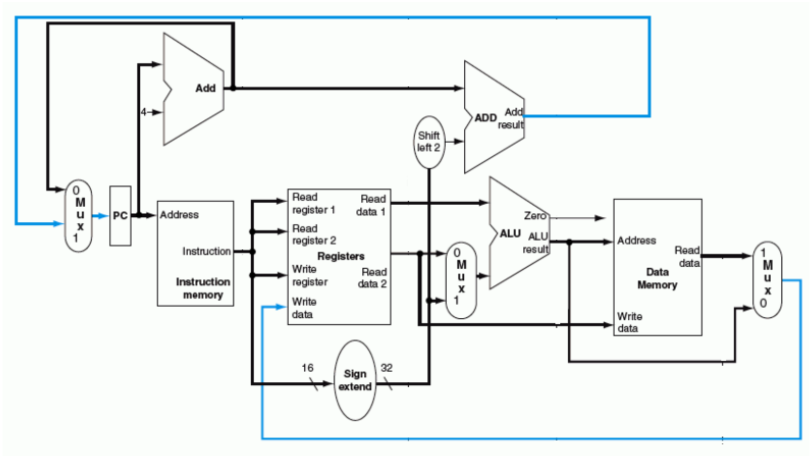
However

However

Pipelining enables us to increase the clockrate

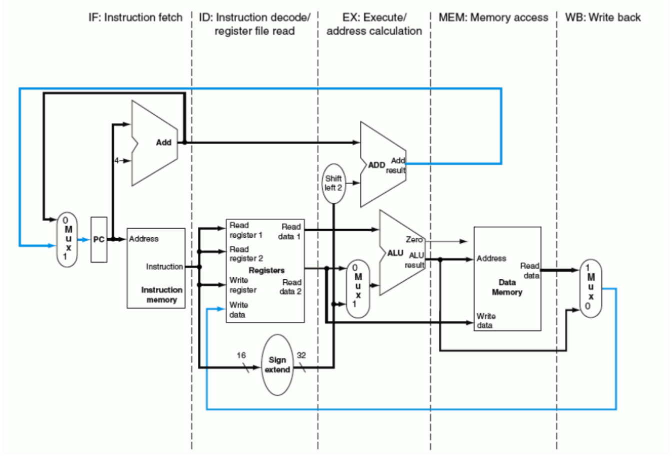
Clockrate dependency: Single Cycle Processor

Clockrate depends on the path delay of the entire processor.



Clockrate dependency: Pipelined Processor

Clockrate depends on the path delay of the **slowest** pipeline stage.



Pipelining Guiding Principle 2

To maximize frequency, delays must be balanced between stages