Unpipelined DLX Architecture

Each DLX instruction has five phases

Thus, each instruction requires five cycles to execute (Clocks Per Instruction or CPI is 5)

- *Instruction fetch* (IF)

  Get the next instruction
- *Instruction decode & register fetch* (ID)

  Decode the instruction and get the registers from the register file
- *Execution/effective address calculation* (EX)

  Perform the operation

  For load and stores, calculate the memory address (base + immed)

  For branches, compare and calculate the branch destination
- *Memory access/branch completion* (MEM)

  For load and stores, perform the memory access

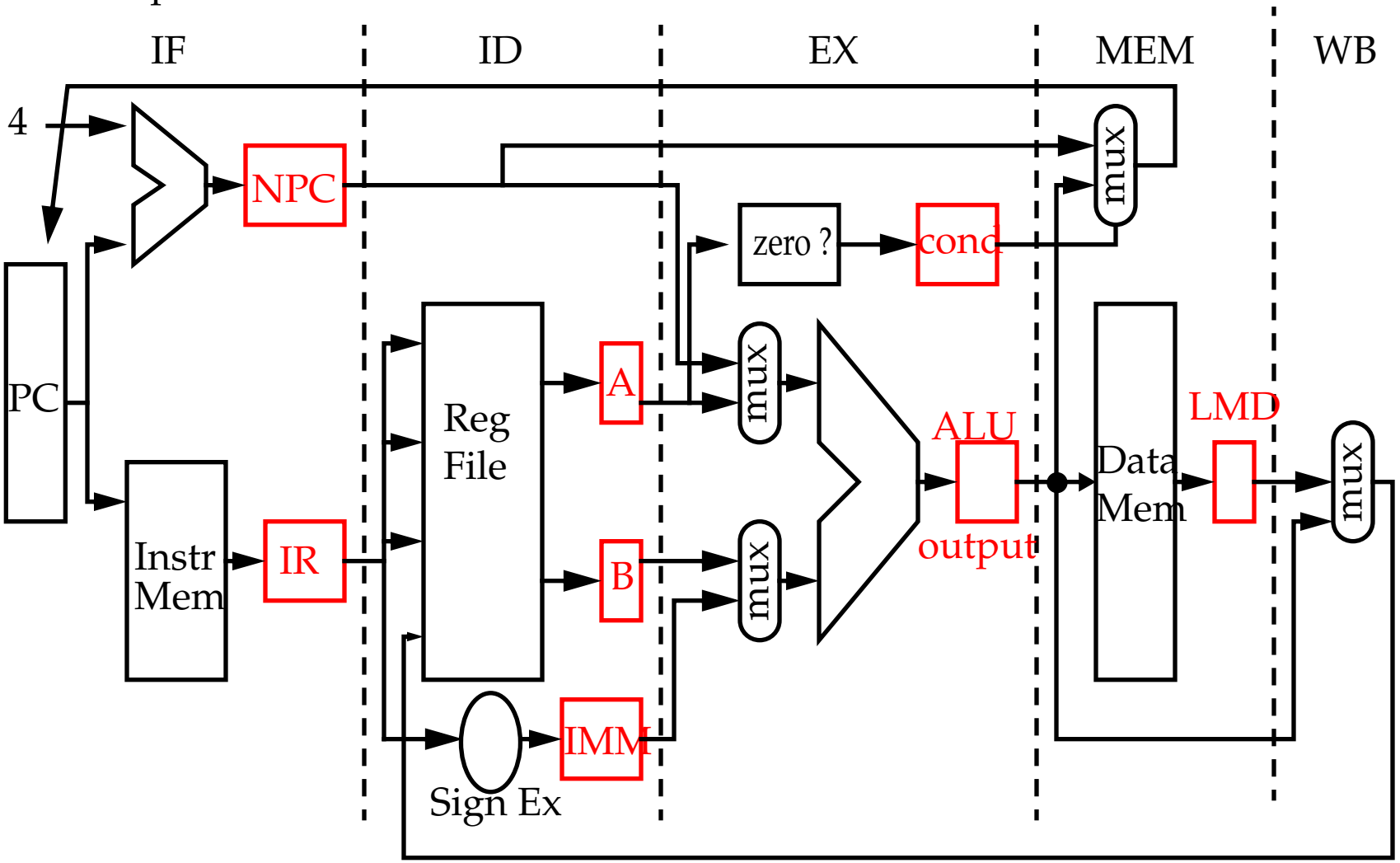
  For **taken** branches, update the program counter
- *Writeback* (WB)

  Write the result to the register file

  For stores and branches, do nothing

Unpipelined DLX Architecture

Datapath:



Red boxes are temporary storage locations.

Simple DLX operation (without pipelining)

The *temporary* storage locations were added to the datapath of the unpipelined machine to make it easy to pipeline

Instruction classes:

- ALU/Logical (**ADD, AND, SUB, OR**)
- Load/Stores
- Control (**BEQZ, BNEQ, JMP, CALL, RETURN, TRAP**)
- Other (System, Floating Point, etc.)

In the above architecture, note that **branch** and **store** instructions take only 4 clock cycles (instead of 5)

Assuming branch frequency of 12% and a store frequency of 5%, CPI ACTUAL CPI is 4.83 (0.17*4 +0.83*5)

Also, several *hardware redundancies* exist:

- ALU can be shared
- Data and instruction memory can be combined since access occurs on different clock cycles

Latency vs. Throughput

Latency vs. throughput

- *Latency*

  Each instruction takes a certain time to complete

  Instruction latency is the amount of time between when the instruction is **issued** and when it completes

- *Throughput*

  The number of instructions that complete in a span of time

Pipelining

Definition

Pipelining is the ability to overlap execution of different instructions at the same time

It exploits *parallelism* among instructions and is **NOT** visible to the programmer

This is similar to building a car on an assembly line

While it may take two hours to build a single car, there are hundreds of cars in the process of being constructed at any time

The *throughput* of the assembly line is the # of cars completed per hour

The *throughput* of a CPU pipeline is the # of instructions completed per second

Pipeline stages

Each step in a pipeline is called a *pipe stage*

In our assembly line example, a stage corresponds to a work station on the assembly line

Pipelining

Cycle time

Everything in a CPU moves in lockstep, synchronized by the clock ("heartbeat" of the CPU)

A *machine cycle*: time required to complete a single pipeline stage

A machine cycle is usually one, sometimes two, clock cycles long, but rarely more

In machines with **no** pipelining:

- The machine cycle must be long enough to complete a single instruction
- Or each instruction must be divided into smaller chunks (multiple clock cycles per instruction)

Pipeline cycle time

All pipeline stages must, by design, take the same time

Thus, the *machine cycle* time is that of the **longest** pipeline stage

Ideally, all stages should be exactly the same length

Pipelining

Pipeline speedup

The *ideal speedup* from a pipeline is equal to the number of stages in the pipeline

$$\frac{\text{Time per instruction on unpipelined machine}}{\text{Number of pipe stages}}$$

However, this only happens if the pipeline stages are all of equal length

Splitting a 40 ns operation into 5 stages, each 8 ns long, will result in a 5x speedup

Splitting the same operation into 5 stages, 4 of which are 7.5 ns long and one of which is 10 ns long will result in only a 4x speedup

If your starting point is a *multiple clock cycle per instruction* machine then pipelining decreases CPI (clocks per instruction)

If your starting point is a *single clock cycle per instruction* machine then pipelining decreases cycle time

We will focus on the first starting point in our analysis

Pipelining DLX

Since there are five separate stages, we can have a pipeline in which one instruction is in each stage

This will decrease CPI to 1, since one instruction will be issued (or finish) each cycle

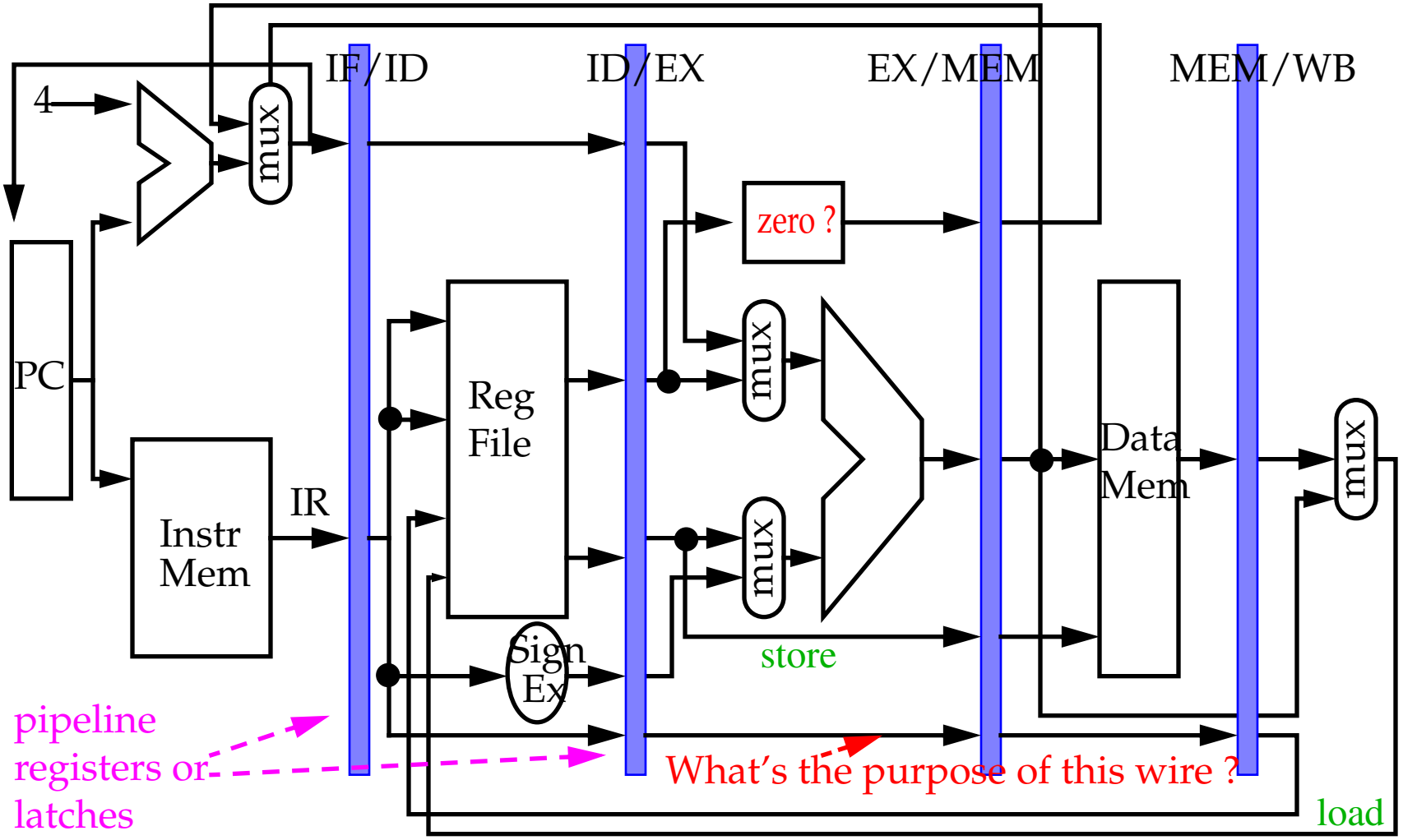
	Clock Number								
	1	2	3	4	5	6	7	8	9
Instruction i	IF	ID	EX	MEM	WB				
Instruction i+1		IF	ID	EX	MEM	WB			
Instruction i+2			IF	ID	EX	MEM	WB		
Instruction i+3				IF	ID	EX	MEM	WB	
Instruction i+4					IF	ID	EX	MEM	WB

During any cycle, one instruction is present in each stage

Ideally, performance is increased five fold !
However, this is rarely achievable as we will see

Pipelining DLX

Data path



Can data path resources, such as the adder, be shared ?

Pipelining DLX

Pipeline Issues:

- Separate instruction caches and data caches eliminates conflicts for memory access in IF and MEM
  - Note that the memory system must deliver *5x* the bandwidth over the unpipelined version

- The register file is used in two stages, reading in ID and writing in WB
  - Two reads and one write required per clock
  - More importantly, what happens when a read and a write occur to the same register?

- What about branch instructions and the PC?
  - Branches change the value of the PC -- but the condition is **not** evaluated until MEM!
  - If the branch is taken, the instructions fetched *behind* the branch are **invalid**
  - This is clearly a serious problem that needs to be addressed

Pipelining performance issues

Pipelining *decreases* execution time but can *increase* cycle time

Throughput is *increased* since a single instruction (ideally) finishes every clock

However, it usually *increases* the **latency** of each instruction

Why?

- Imbalance among the pipe stages:

  The slowest stage determines the clock cycle time

- Pipeline overhead:

  Pipeline register delay. Adding registers, adds logic between each of the stages (plus constraints on setup and hold times for proper operation -- but we won't talk about those)

  Clock skew. The clock must be routed to possibly widely separated registers/latches, introducing delay in signal arrival times

In the limit, i.e., if the combination logic delay is zero, clock cycle time is bound by the sum of the clock skew and latch overhead

Pipelining performance issues

Instruction regularity:

With a pipeline, differences in instruction CPI can **NOT** be taken advantage of

In the unpipelined version, a store instruction finishes after MEM, 4 clocks rather than 5

With pipelining, we can not start the next instruction one clock earlier since it is already in the pipeline

Therefore, CPI may **not** be decreased by the number of pipeline stages (ideal case is usually not achievable)

This effect reduces the maximum pipeline *depth* since the variance in the # of stages required for each instruction grows as stages are added

Pipelining can be thought of as reducing the CPI

This *increases* throughput even though clk cycle time is *increased*

Pipeline hazards

A hazard is a condition that prevents an instruction in the pipe from executing its next scheduled pipe stage

There are three types of hazards

- *Structural hazards*

  These are conflicts over hardware resources

- *Data hazards*

  These occurs when an instruction needs data that is not yet available because a previous instruction has not computed or stored it

- *Control hazards*

  These occur for branch instructions since the branch condition (for compare and branch) and the branch PC are not available in time to fetch an instruction on the next clock

Pipeline stalls

Hazards in the pipeline may make it necessary to **stall** the pipeline

Stall definition:

The simplest way to "fix" hazards is to stall the pipeline

This means suspending the pipeline for *some* instructions by one or more clk cycles

The stall delays all instructions issued **after** the instruction that was stalled

A pipeline stall is also called a *pipeline bubble* or simply *bubble*

Pipeline stalls

Stall location:

Note that a bubble need **not** occur at the start of an instruction

It can be inserted in the middle

A bubble occurs whenever the pipeline stage must be suspended (for a given instruction) to allow a *previous* instruction to proceed

The previous instruction MUST proceed in order for the hazard to clear

No new instructions are fetched during a stall

To emphasize, all instructions issued *later* than the stalled instruction are stalled

Performance of Pipelines with stalls

Effect on pipeline speedup:

Pipeline stalls decrease performance from the ideal !

Every cycle the pipeline is stalled results in a cycle in which an instruction is NOT issued

And thus is a cycle in which an instruction is NOT completed

Lets start with the basic formula:

$$\text{Speedup from pipelining} = \frac{\text{Average instruction time unpipelined}}{\text{Average instuction time pipelined}}$$

Performance Equation: Instruction Count (IC) * Clocks per Instruction (CPI) * Clk cycle time

Inserting into above equation:

$$= \frac{\text{CPI unpipelined} \times \text{Clock cycle unpipelined}}{\text{CPI pipelined} \times \text{Clock cycle pipelined}}$$

$$= \frac{\text{CPI unpipelined}}{\text{CPI pipelined}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}$$

Performance of Pipelines with stalls

Effect on pipeline speedup:

Note that pipelining can be thought of as *decreasing CPI* or *decreasing clock cycle time* - let's focus on the former.

Assuming the ideal CPI is 1:

CPI pipelined = Ideal CPI + Pipeline stall clk cycles per instuction

= 1 + Pipeline stall clk cycles per instruction

Let's ignore increases in clk cycle time (due to pipeline overhead):

$$\text{Speedup} = \frac{\text{CPI unpipelined}}{1 + \text{Pipeline stall cycles per instruction}}$$

Let's further assume unpipelined CPI is equal to the depth of the pipeline (ignore shorter instruction CPIs).

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall cycles per instruction}}$$

Performance of Pipelines with stalls

Effect on pipeline speedup:

If we include the effect of pipeline overhead, we get:

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline Stall cycles per instruction}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}$$