ECE260: Fundamentals of Computer Engineering

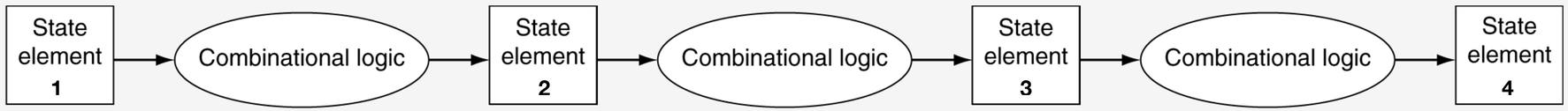
Pipelining

James Moscola
Dept. of Engineering & Computer Science
York College of Pennsylvania

What is Pipelining?

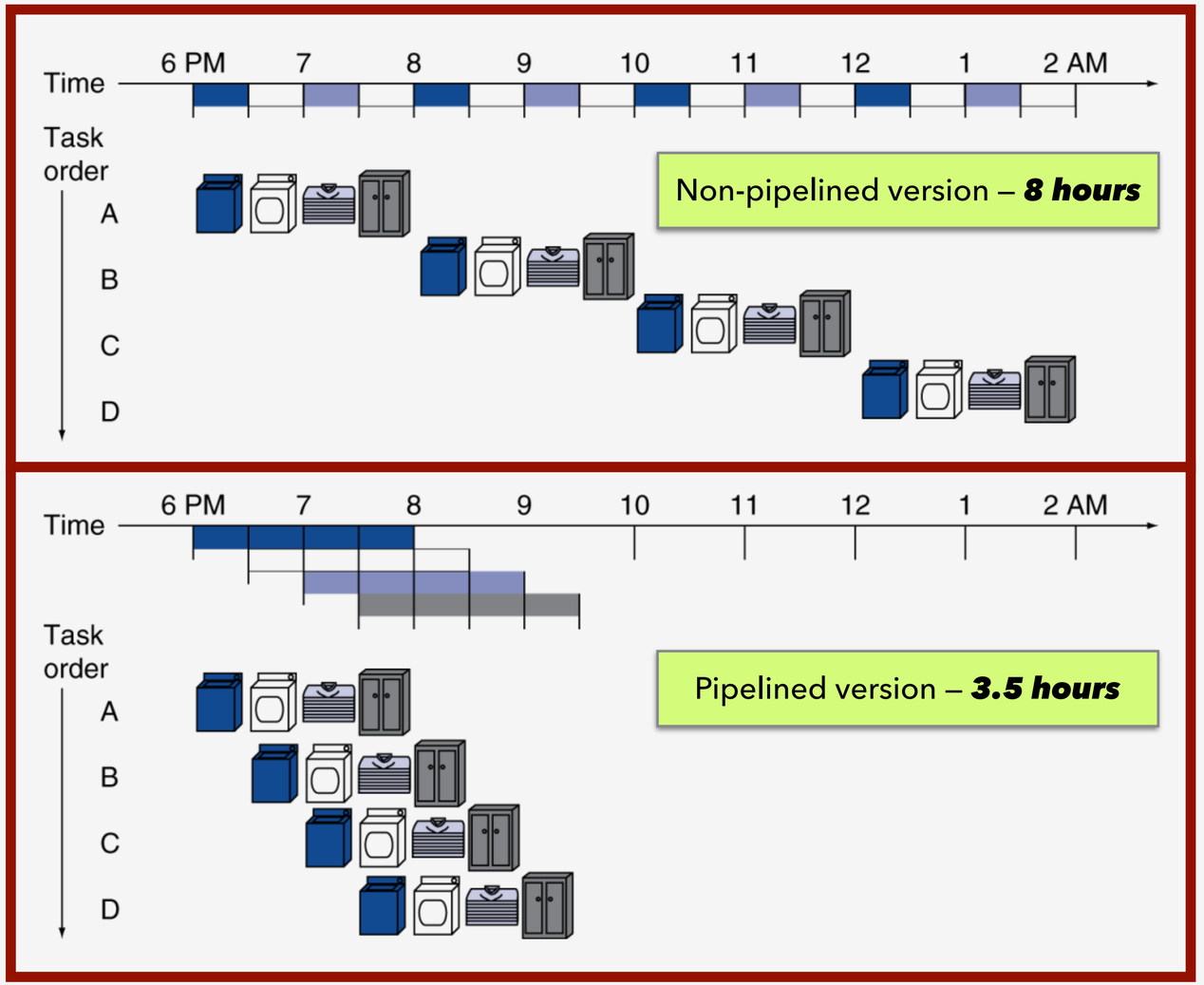
- ***Pipelining*** is an implementation technique in which multiple instructions are overlapped in execution
  - Pipeline has multiple ***stages*** where each stage performs a specific task
    - Single-cycle datapath is divided into smaller combinational logic elements separated by sequential storage elements
  - Instructions pass through each stage of a pipeline – one stage at a time
  - Allows multiple instructions to be worked on concurrently
    - Reduces time to complete multiple instructions
    - Does not reduce time to finish any one single instruction

- A processor pipeline behaves similarly to an assembly line

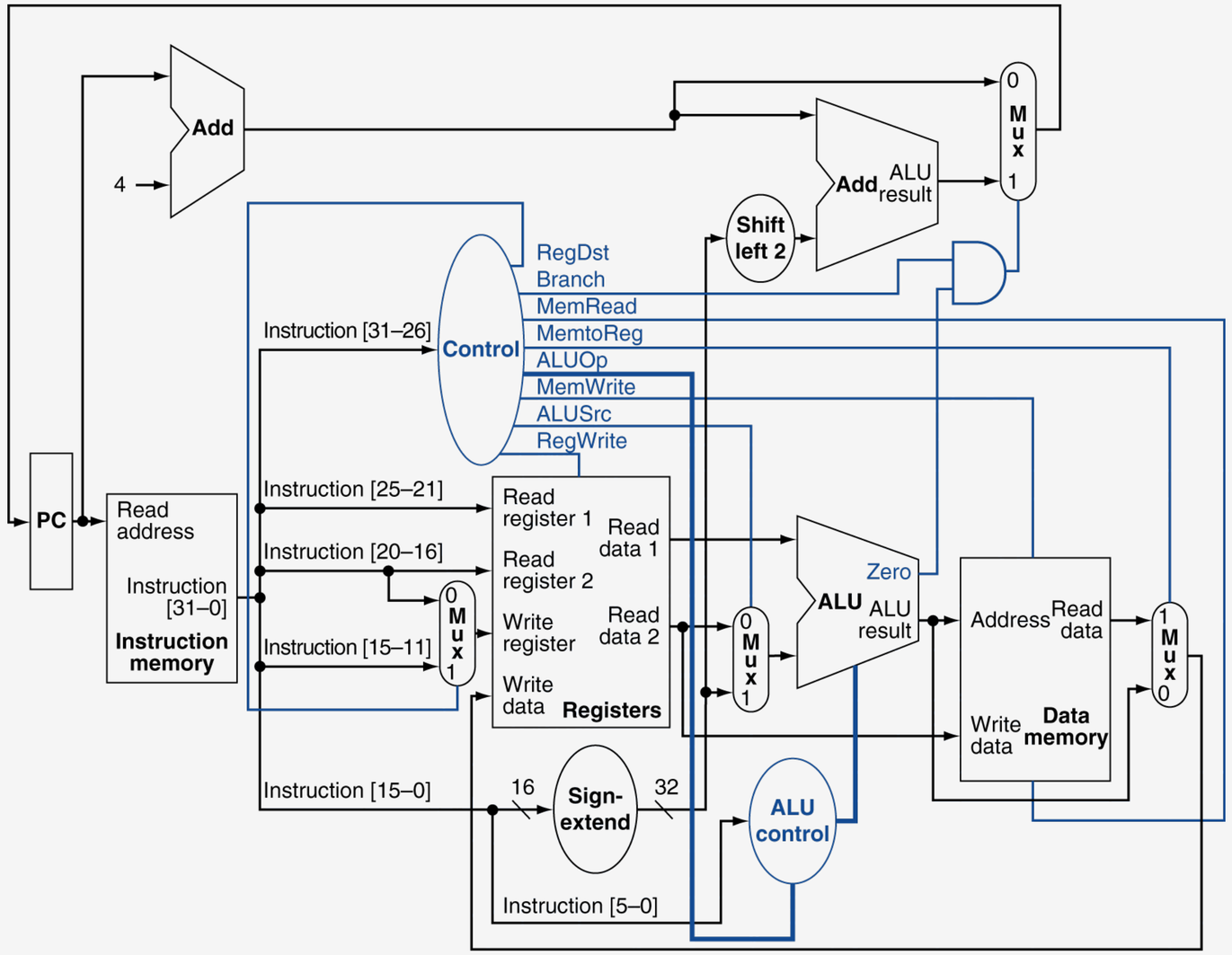


Pipelining Analogy

- Pipelined laundry – overlapping execution of different stages
  - Parallelism improves performance
- Non-pipelined version
  - Wash, then dry, then fold, then store
- Pipelined version
  - Wash, then dry AND start next wash, etc.
  - Allows tasks to be done on multiple loads of laundry concurrently, reducing time to finish all laundry (***improves throughput***)
  - Each load of laundry still takes the same amount of time



MIPS Single-Cycle Datapath (not pipelined … yet)



The MIPS Pipeline – a Classic 5-Stage Pipeline

- Five stages, one step per stage
  1) **IF:** Instruction Fetch from memory
     - Uses PC to address instruction memory and retrieve instruction
  2) **ID:** Instruction decode & register read
     - Controller decodes the instruction while values are retrieved from register file
  3) **EX:** Execute arithmetic operation or calculate address
     - Sometimes referred to as the “ALU Stage”
  4) **MEM:** Access memory
     - Retrieves value from data memory (if necessary)
  5) **WB:** Write result back to register file
     - Result of arithmetic and load instructions are written back to destination register

Pipeline Performance

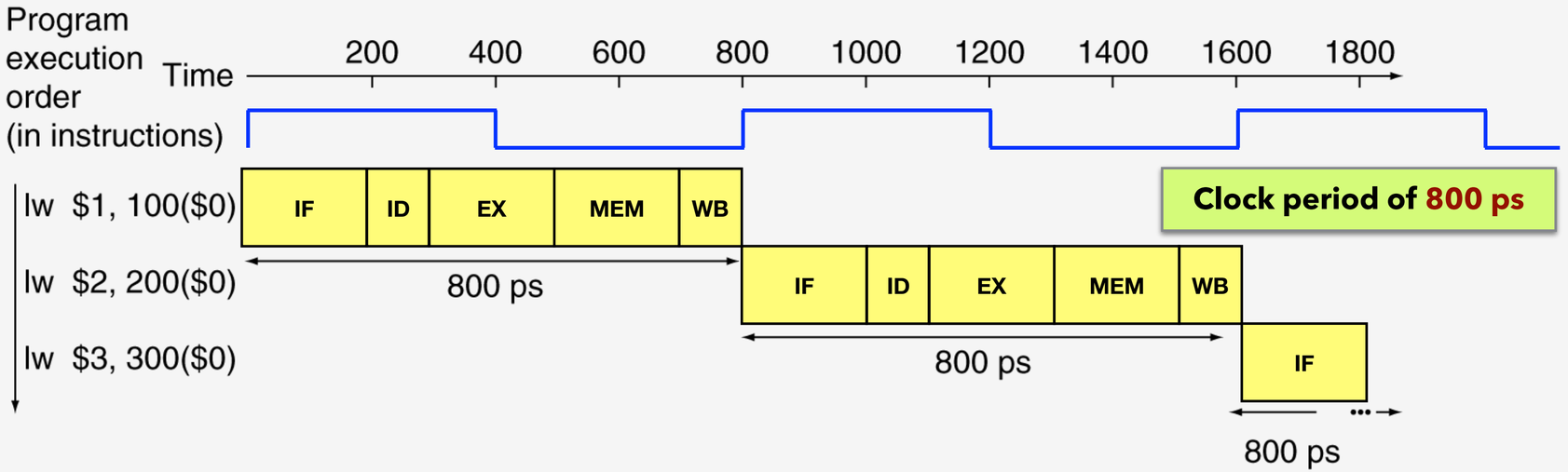
- Assume time for pipeline stages is as follows: 100 ps for ID and WB; 200 ps for others (IF, EX, MEM)

Example Instructions	Pipeline Stage Latency (ps)					Total Latency (ps)
	IF	ID	EX	MEM	WB	
lw	200	100	200	200	100	800
sw	200	100	200	200		700
R-Type	200	100	200		100	600
beq	200	100	200			500

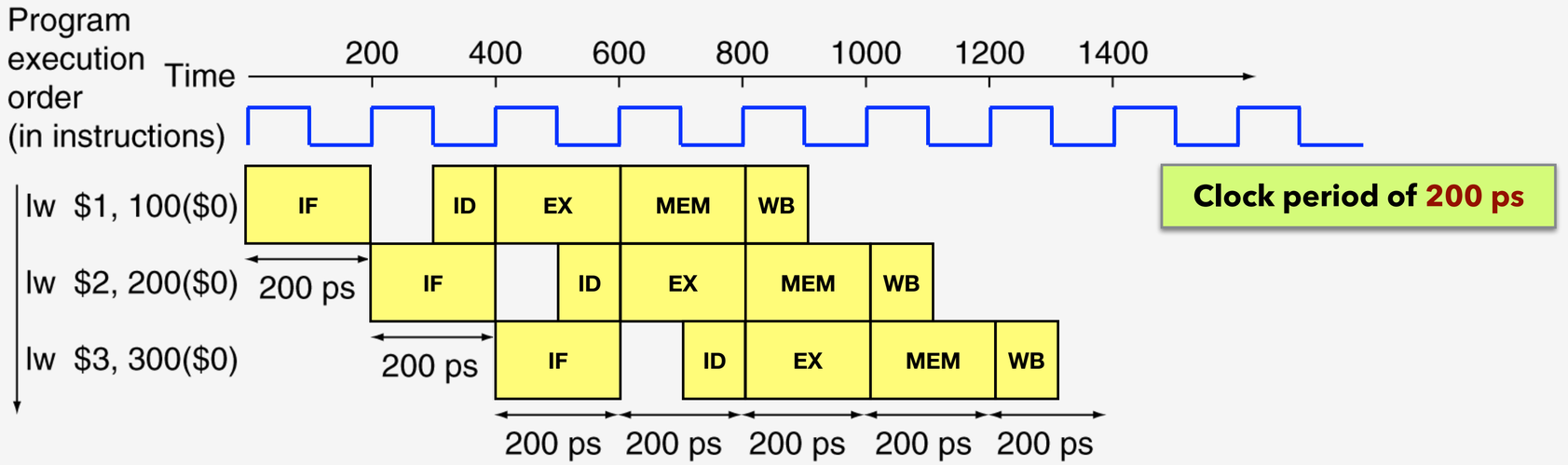
- The **lw** instruction uses all stages of the pipeline
  - If implemented in a single-cycle datapath, requires an 800 ps clock period (1.25 GHz)
  - If implemented in a pipelined datapath, the longest stage of the pipeline determines the clock period – 200 ps in the example shown here (5 GHz)
    - Longest stage of the pipeline is the ***critical path***

Pipeline Performance (continued)

Three **lw** instructions in a **single-cycle** datapath



Three **lw** instructions in a **pipelined** datapath



Pipeline Speedup

- Measured as the ***time between instructions***
  - The amount of time between starting instruction *x* and starting instruction *x+1*
- If all pipeline stages are balanced (i.e. they all take the same amount of time)

$$\text{Time Between Instructions}_{\text{pipelined}} = \frac{\text{Time Between Instructions}_{\text{nonpipelined}}}{\text{Number of Pipeline Stages}}$$

- If pipelined stages are not balanced, then speedup is less
- Pipeline speedup is due to increased throughput
  - i.e. less time between instructions
  - i.e. more instructions executed per unit time
- Time to execute each individual instruction does not decrease

Hazards

- Situations that prevent starting the next instruction on the next clock cycle
- Three main types of hazards
  - ***Structural hazard***
    - Next instruction cannot execute because a required hardware resource is busy
  - ***Data hazard***
    - Next instruction needs to wait for previous instruction to complete its data read/write (data dependent)
  - ***Control hazard***
    - Next instruction depends on a control action of previous instruction

Structural Hazards

- Next instruction cannot execute because a required hardware resource is busy
  - Conflict for use of a resource
- ***Example:*** if MIPS had only a single memory shared as instruction and data memory
  - Could not read the next instruction while also performing the MEM stage of a load/store instruction
    - Load/store instructions require access to memory
    - Instruction fetch would have to stall for the cycle that load/store accesses memory
    - Stalling instruction fetch would cause a pipeline "bubble"
- Pipelined datapaths require separate instruction/data memories
  - Or separate instruction/data caches

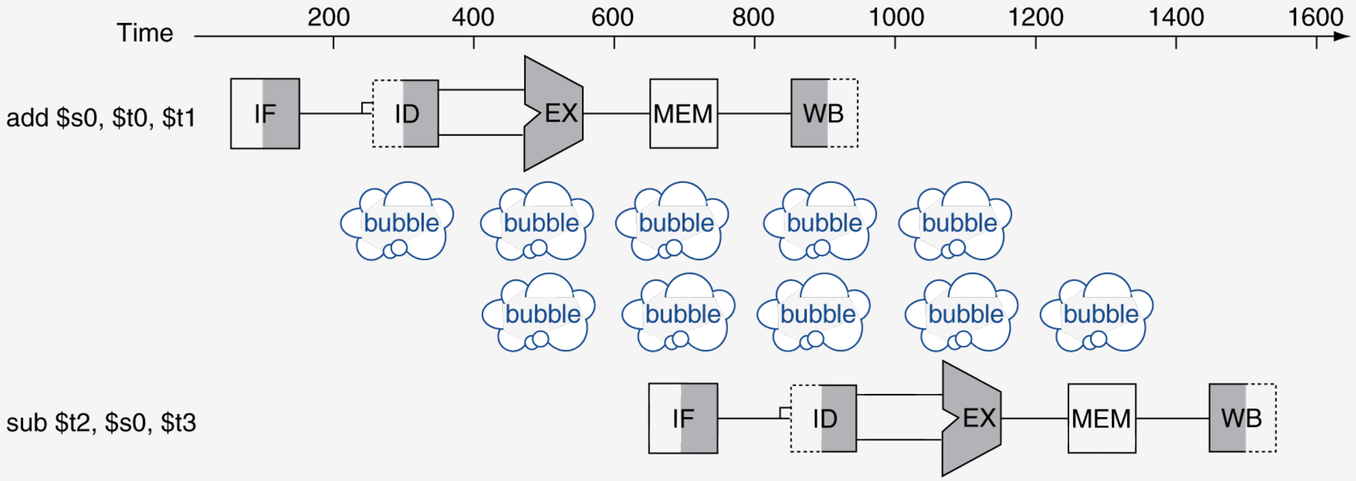
Data Hazards

- Next instruction needs to wait for previous instruction to complete its data read/write
  - A data dependency exists between the result of one instruction and the next

```
add    $s0, $t0, $t1
sub    $t2, $s0, $t3
```

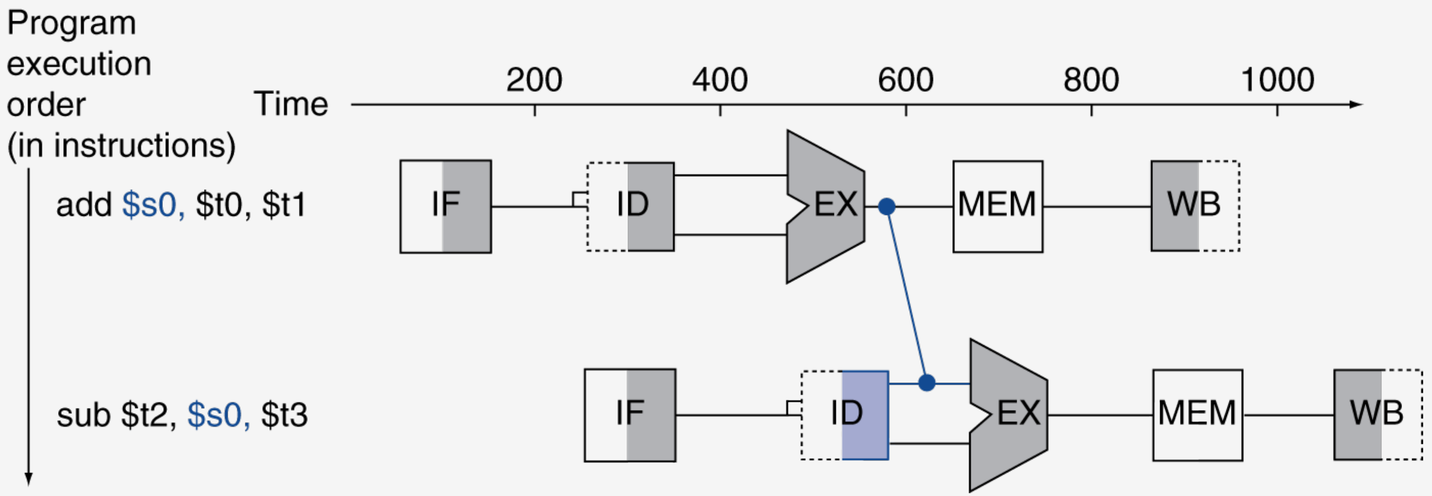
sub instruction requires $s0 as input
add instruction will not yet have executed WB stage!
Register file will contain "stale" data for $s0 when **sub** requests it

- A naive solution might just stall the pipeline by inserting bubbles, thereby allowing WB of **add** to happen before ID of **sub** (there's a better way 🤓)



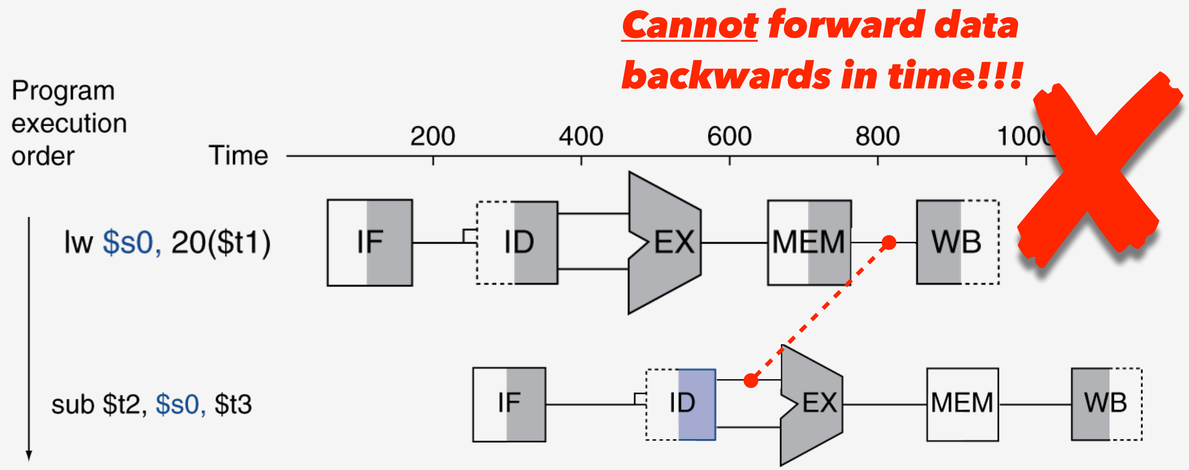
Forwarding (a.k.a. Bypassing) (a.k.a. the Better Way)

- Use the result of an instruction immediately after it is computed
  - Don’t wait for it to be stored in a register
  - Requires extra hardware in the datapath (i.e. some wires and control)
- The result of instruction x is ***forwarded*** to instruction x+1 (***bypassing*** the WB stage)
  - The **add** instruction still completes normally and writes the register file during its WB stage

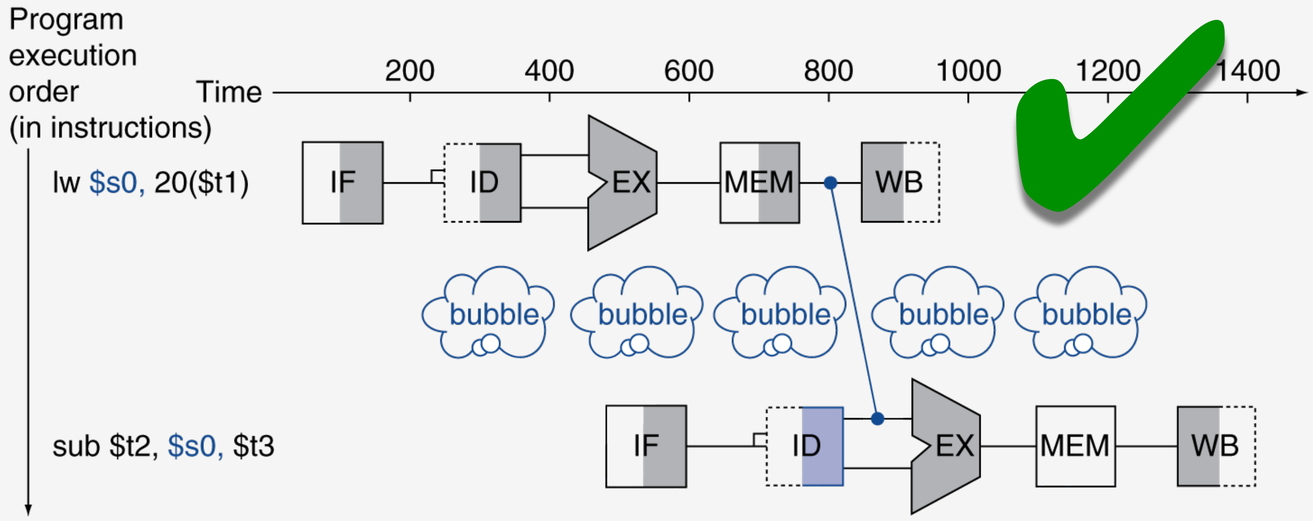


Load-Use Data Hazard

- Forwarding works very well in many cases, but not all
- Consider when a **lw** instruction precedes an instruction that uses the value being loaded



- Must stall the pipeline and insert a "bubble" so the output of the MEM stage can be forwarded to the input of the EX stage of the next instruction
  - Stalling the pipeline reduces instruction throughput (i.e. decreases performance)



Code Scheduling to Avoid Stalls

- Be a clever programmer and avoid stalls
  - Reorder code to avoid the use of load result in the next instruction
- Two implementations of the same C code that take different number of clock cycles to complete

Sample C Code:

```
a = b + e;
c = b + f;
```

Implementation #1:

```
lw  $t1, 0($t0)
lw  $t2, 4($t0)
add $t3, $t1, $t2
sw  $t3, 12($t0)
lw  $t4, 8($t0)
add $t5, $t1, $t4
sw  $t5, 16($t0)
```

Stalls will be required and inserted before these **add** instructions

5 clock cycles for **sw** instruction to complete

Requires 13 clock cycles

(Two stalls required)

Implementation #2:

```
lw  $t1, 0($t0)
lw  $t2, 4($t0)
lw  $t4, 8($t0)
add $t3, $t1, $t2
sw  $t3, 12($t0)
add $t5, $t1, $t4
sw  $t5, 16($t0)
```

Requires 11 clock cycles

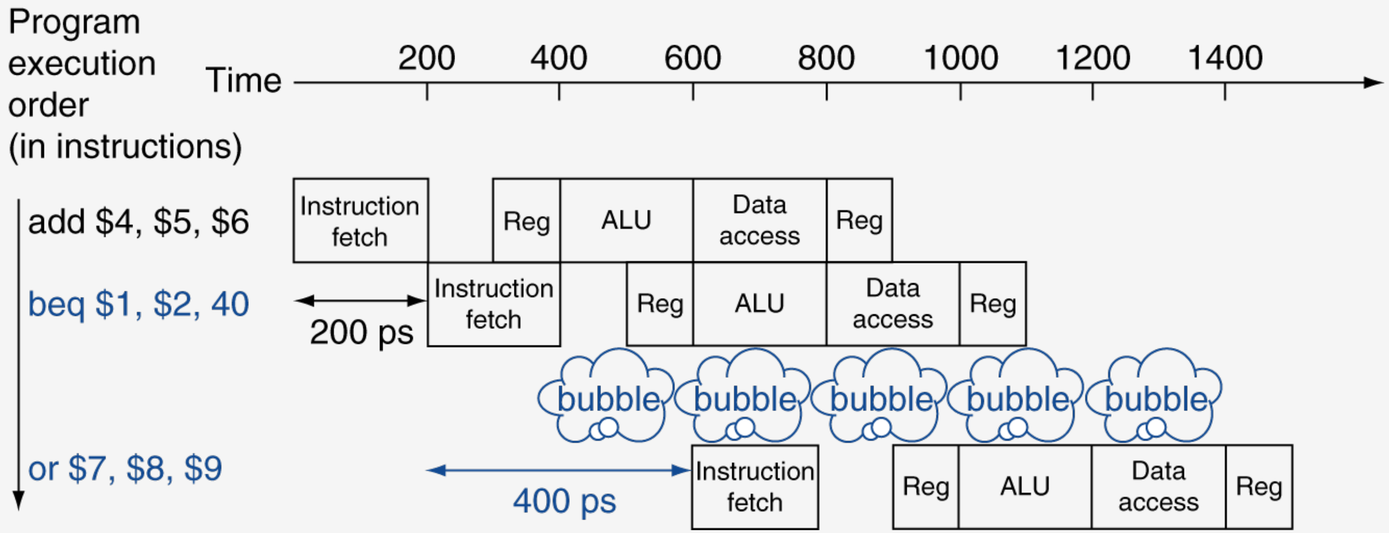
(No stalls required)

Control Hazards (a.k.a. Branch Hazard)

- A branch instruction determines the flow of control in a program
  - The next instruction to fetch and execute after a branch instruction depends on the branch outcome
    - Don't actually know the outcome of a branch instruction until EX stage
  - Pipeline can't always fetch correct instruction (don't really know which instruction to get yet)
    - Pipeline is attempting to fetch next instruction while the branch instruction is still in the ID stage
- Multiple solutions:
  - Stall until result of branch is determined
  - Guessing (more common that you'd … … guess)
  - Use a "branch delay slot"

Stall on Branch

- Wait until branch outcome is determined before fetching the next instruction
  - Without any additional hardware, would require a stall of 3 clock cycles for each branch
    - Very inefficient looping behavior!
  - Could add some additional hardware to get result of branch after ID stage of pipeline
    - Would still require a cycle of stalling (see below)

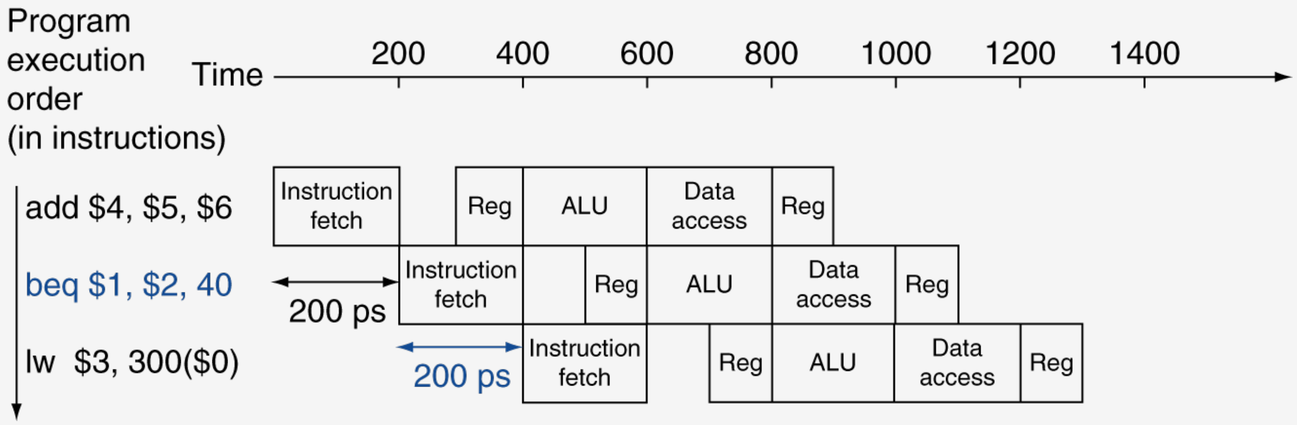


Branch Prediction (Yes, Guessing!)

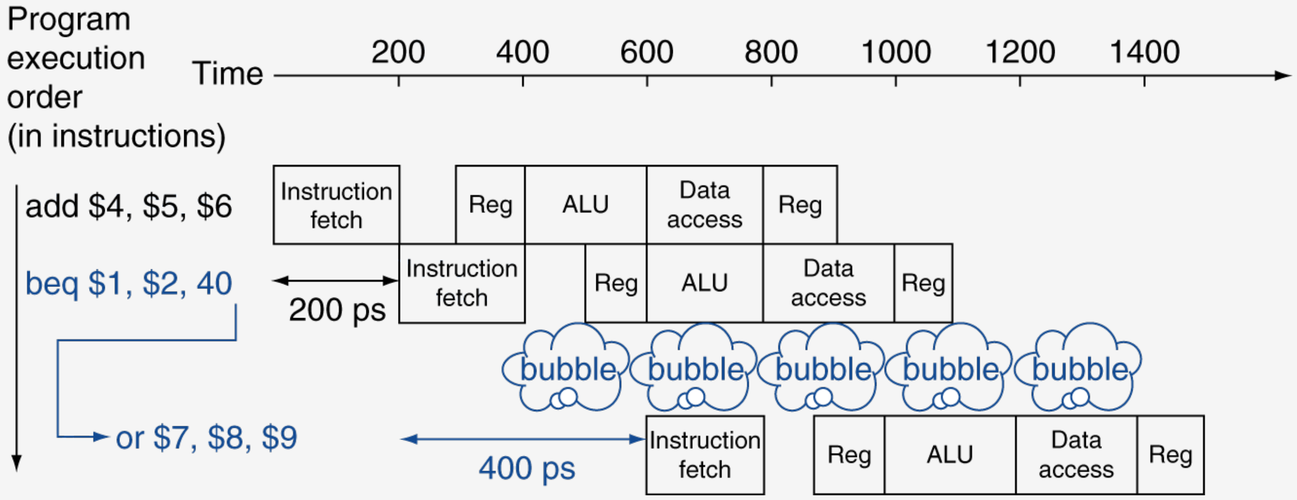
- If stall penalty is unacceptable, try to ***predict*** the result of a branch before actual result is available
  - Many implementations will always assume that the branch is **NOT** taken and fetch the instruction that immediately follows the branch (this is the common case)
    - No stalls are inserted, pipeline continues full speed ahead
    - If the prediction was correct, then no penalty
  - Sometimes prediction will be wrong
    - Stall will need to be inserted if prediction was wrong
    - Result of wrongly fetched/executed instruction discarded
- In MIPS pipeline
  - Can predict branches not taken
  - Fetches instruction after branch, with no delay

MIPS with Predict Not Taken

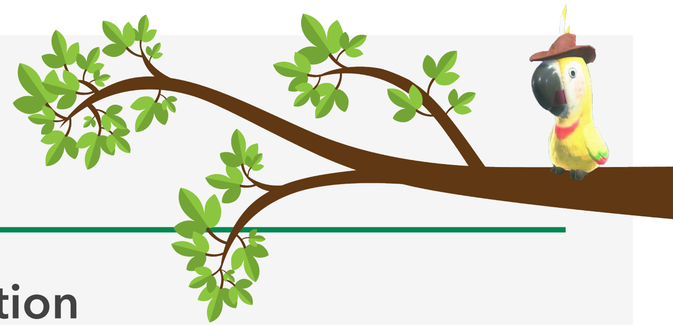
MIPS predict branch not taken behavior when **prediction is correct** (i.e. branch was not taken and didn't need to be)



MIPS predict branch not taken behavior when **prediction is incorrect** (i.e. branch was not taken but should have been)



Branch Delay Slots



- Rearrange instructions and put an arbitrary instruction after the branch instruction
  - Instruction following the branch is always executed
    - Pick an instruction that would have been executed regardless of branch taken/not taken
  - A suitable choice for an instruction to insert into a branch delay slot is the instruction that immediately precedes the branch instruction (assembler can swap the two instructions)
    - Branch instruction cannot depend on result of preceding instruction

```
add $t0, $t0, $t0
beq $s0, $s1, label
or  $t1, $t0, $t2
...
```

The assembler can swap the add and beq instructions without affecting the programs results.

```
beq $s0, $s1, label
add $t0, $t0, $t0
or  $t1, $t0, $t2
...
```

After the assembler rearranges instructions. The add instruction is in the branch delay slot and will always execute.