



# CS5375 Computer Systems Organization and Architecture

## Lecture 11

Instructor: Yong Chen, Ph.D.  
Department of Computer Science  
Texas Tech University  
Yong.Chen@ttu.edu, 806-834-0284

## Announcements

- HW#2 due on 10/11, Tuesday
- Review session on 10/18, Tuesday
- Midterm exam on 10/20, Thursday, 8 a.m. - 9:20 a.m.
  - Must take the exam during the class time and in person in the classroom
  - Test subjects we discussed regarding chapters 1 and 2, i.e., all lectures till Lecture 9
    - Only subjects we discussed in class, you should review lecture slides, HW, related textbook discussion
    - Final exam will test subjects we cover from Lecture 10
  - All multi-choice questions, closed-book, closed-note
  - Will enforce randomized seating, more to be announced later

## Review of Last Lecture

- Programming project #1 and demo
- Instruction-level parallelism
  - Pipelining
  - Speedup Analysis
    - *Non-pipelined (sequential) time* =  $ni \times ns \times ts$
    - *Pipelined time* =  $(ns - 1) \times ts + ni \times ts = (ns + ni - 1) \times ts$
    - $Speedup = \frac{\text{Non-pipelined time}}{\text{Pipelined time}} = \frac{ni \times ns \times ts}{(ns + ni - 1) \times ts} = \frac{ni \times ns}{ns + ni - 1} = \frac{ns}{\frac{ns}{ni} + 1 - \frac{1}{ni}}$
    - For large  $ni$ :  $Speedup = \lim_{ni \rightarrow \infty} \frac{ns}{\frac{ns}{ni} + 1 - \frac{1}{ni}} \approx \frac{ns}{0 + 1 - 0} \approx ns$
  - Classic, 5-stage pipeline, with RISC-V as an example

## Outline

- Instruction-level parallelism
  - Pipelining

# Hazards

- **Hazards**: situations that prevent starting the next instruction in the next cycle
  - Structural hazard
  - Data hazard
  - Control hazard

## Structural Hazards

- A required resource (datapath element) is busy, i.e., conflict for use of a resource
- If we have a **single memory** in the pipeline
  - Both IF and load/store (MEM) requires memory access
  - Instruction fetch would have to **stall** for that cycle
    - Would cause a pipeline “**bubble**”
- Hence, **pipelined datapaths require separate instruction/data memories**
  - Or separate instruction/data caches

	Pipelined execution														
	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8	Cycle 9	Cycle 10	Cycle 11	Cycle 12	Cycle 13	Cycle 14	Cycle 15
lw	IF	ID	EX	MEM	WB										
lw		IF	ID	EX	MEM	WB									
lw			IF	ID	EX	MEM	WB								
				IF	ID	EX	MEM	WB							

# Data Hazard from Data Dependence

- Data dependency
  - Instruction  $j$  is data dependent on instruction  $i$  if
    - Instruction  $i$  produces a result that may be used by instruction  $j$
    - Instruction  $j$  is data dependent on instruction  $k$  and instruction  $k$  is data dependent on instruction  $i$
- Dependencies are a property of programs
- Data dependence conveys:
  - Possibility of a hazard
  - Order in which results must be calculated
  - Upper bound on exploitable instruction level parallelism

## Data Hazard from Name Dependence

- Two instructions use the same name but no flow of information
  - Not a true data dependence, but is a problem when reordering instructions
- Antidependence
  - instruction j writes a register or memory location that instruction i reads
    - Initial ordering (i before j) must be preserved
- Output dependence
  - instruction i and instruction j write the same register or memory location
    - Ordering must be preserved
- To resolve, use register renaming techniques (i.e., use another register)



## Data Hazards

- Data Hazards
  - Read after write (RAW)
    - True data dependence
  - Write after write (WAW)
    - Output dependence
  - Write after read (WAR)
    - Antidependence

## Data Hazards

- An instruction depends on completion of data access by a previous instruction

add     **x19**, x0, x1

sub     x2, **x19**, x3

Here it needs to write to a register before second register uses it

Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8
IF	ID	EX	MEM	WB			
	IF	ID	EX	MEM	WB		



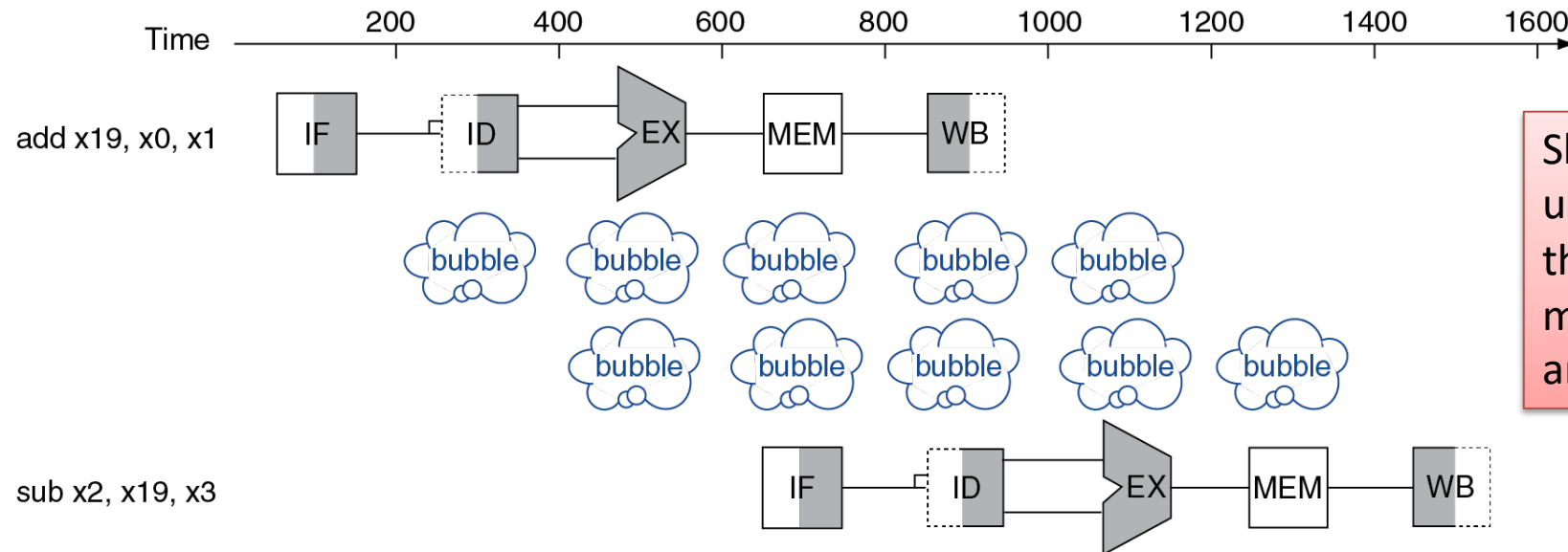
Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8
IF	ID	EX	MEM	WB			
	Bubble	Bubble	Bubble	Bubble	Bubble		
		Bubble	Bubble	Bubble	Bubble	Bubble	
			IF	ID	EX	MEM	WB



# Data Hazards

- An instruction depends on completion of data access by a previous instruction

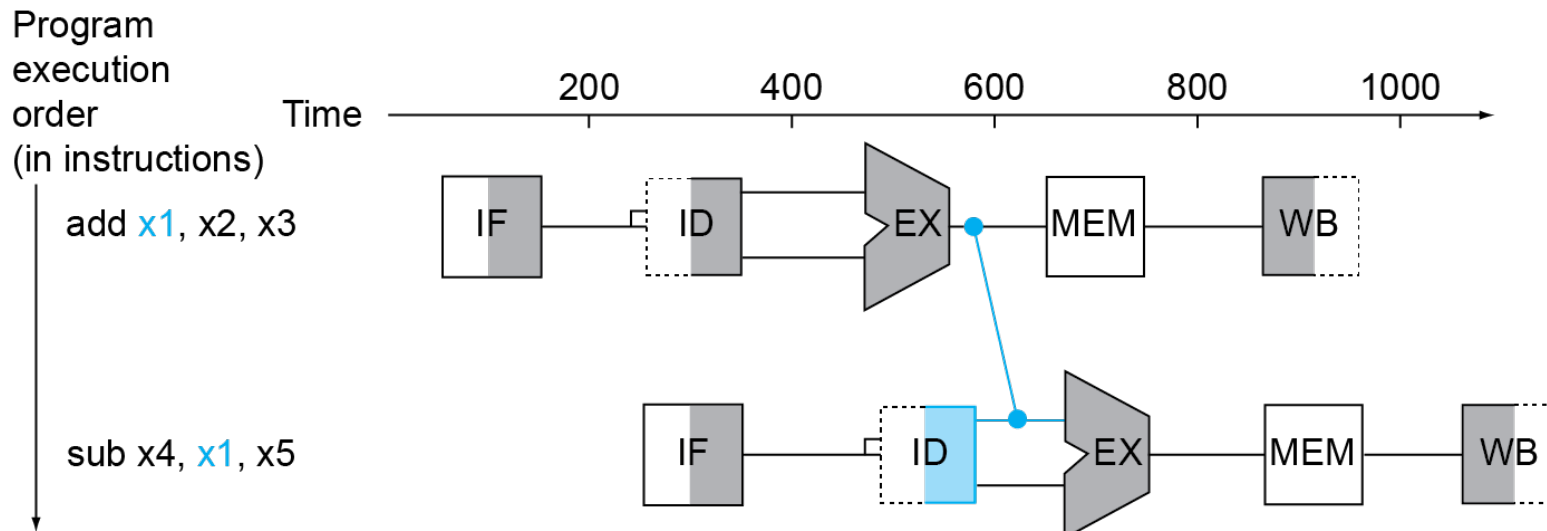
```
add    x19, x0, x1
sub     x2, x19, x3
```



Shading indicates the element is used by the instruction. Shading on the right half of the register file or memory means read in that stage and shading on the left means write.

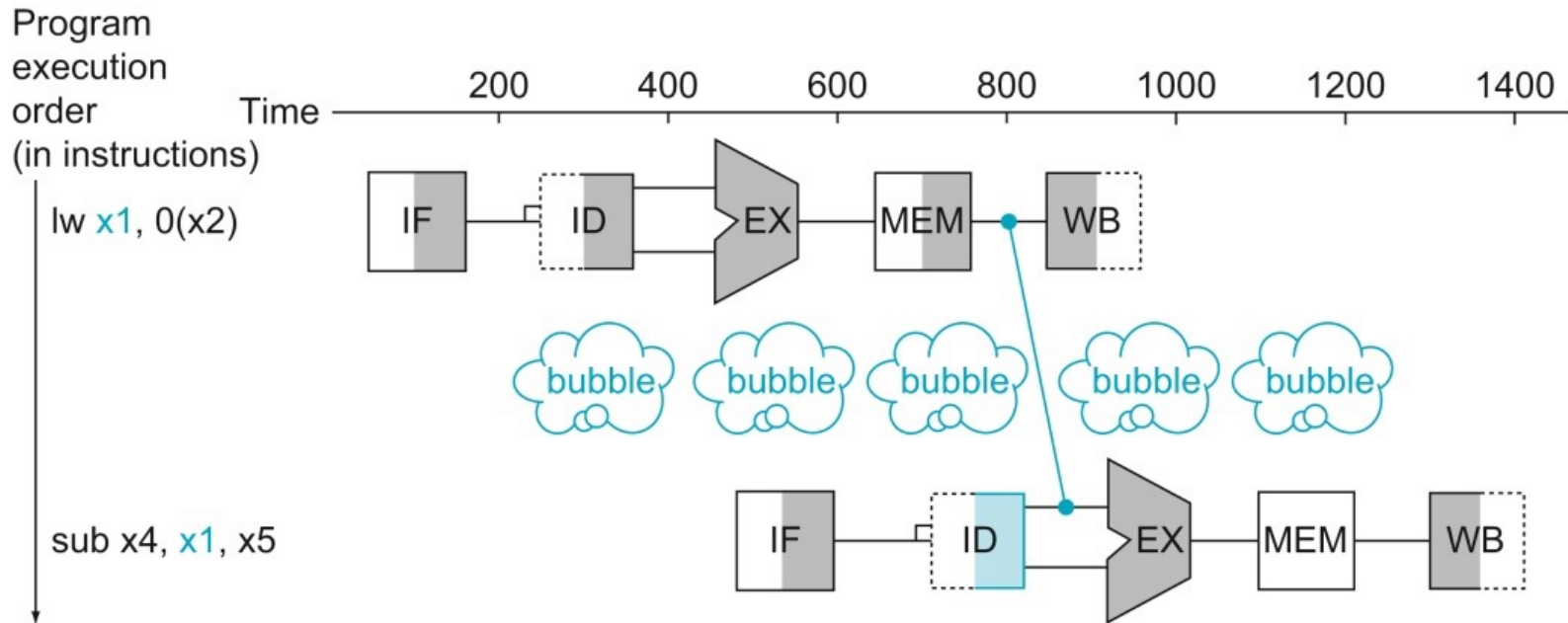
## Forwarding (aka Bypassing)

- **Forwarding**: method of resolving a data hazard, use result when it is computed
  - Don't wait for it to be stored in a register
  - Requires extra connections in the datapath



## Load-Use Data Hazard

- Can't always avoid stalls by forwarding, e.g., a **load-use data hazard** (i.e., data being loaded have not yet become available when needed by another instruction)
  - Data being loaded only becomes available after the MEM stage
  - Can't forward backward in time!

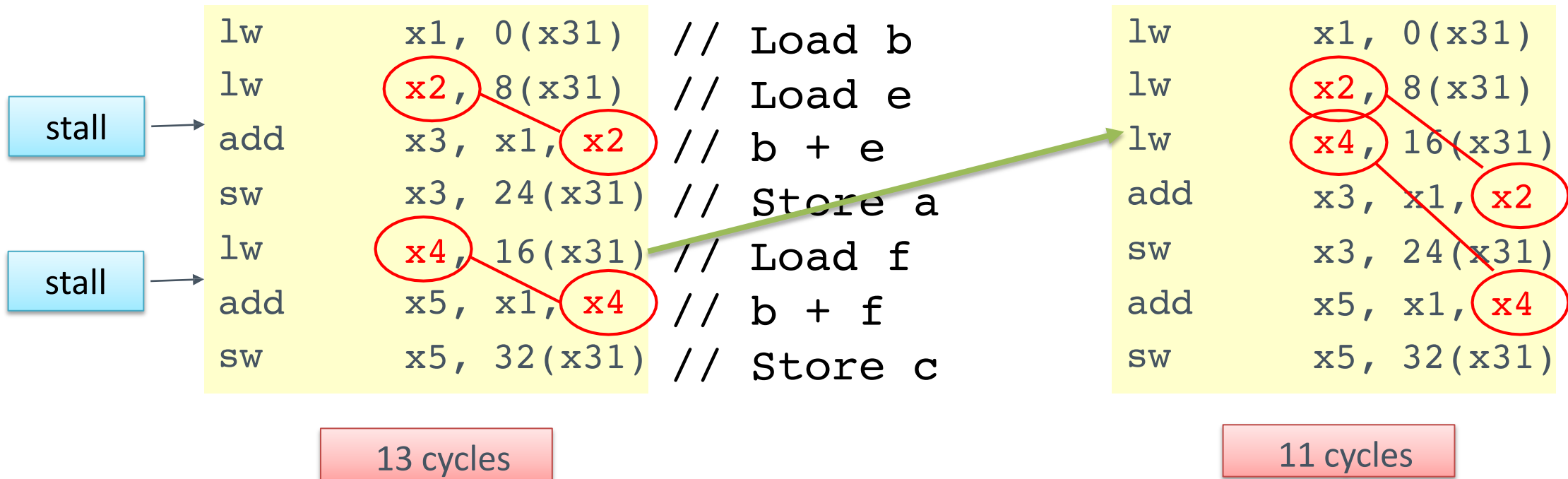


A **pipeline stall** (or bubble) still occurs even with forwarding when an R-format instruction following a load tries to use the data

# Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction; e.g., C code for:

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



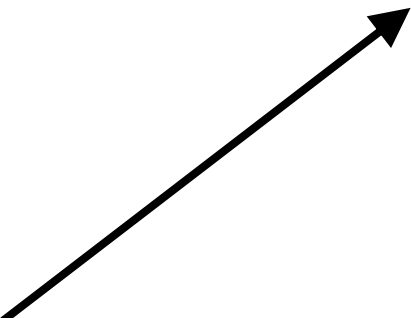
## Code Scheduling to Avoid Stalls (cont.)

- Another example:

```
for (i=999; i>=0; i=i-1)
    x[i] = x[i] + s;
```

```
Loop: fld    f0,0(x1)
      stall
      fadd.d f4,f0,f2
      stall
      stall
      fsd    f4,0(x1)
      addi   x1,x1,-8
      bne    x1,x2,Loop
```

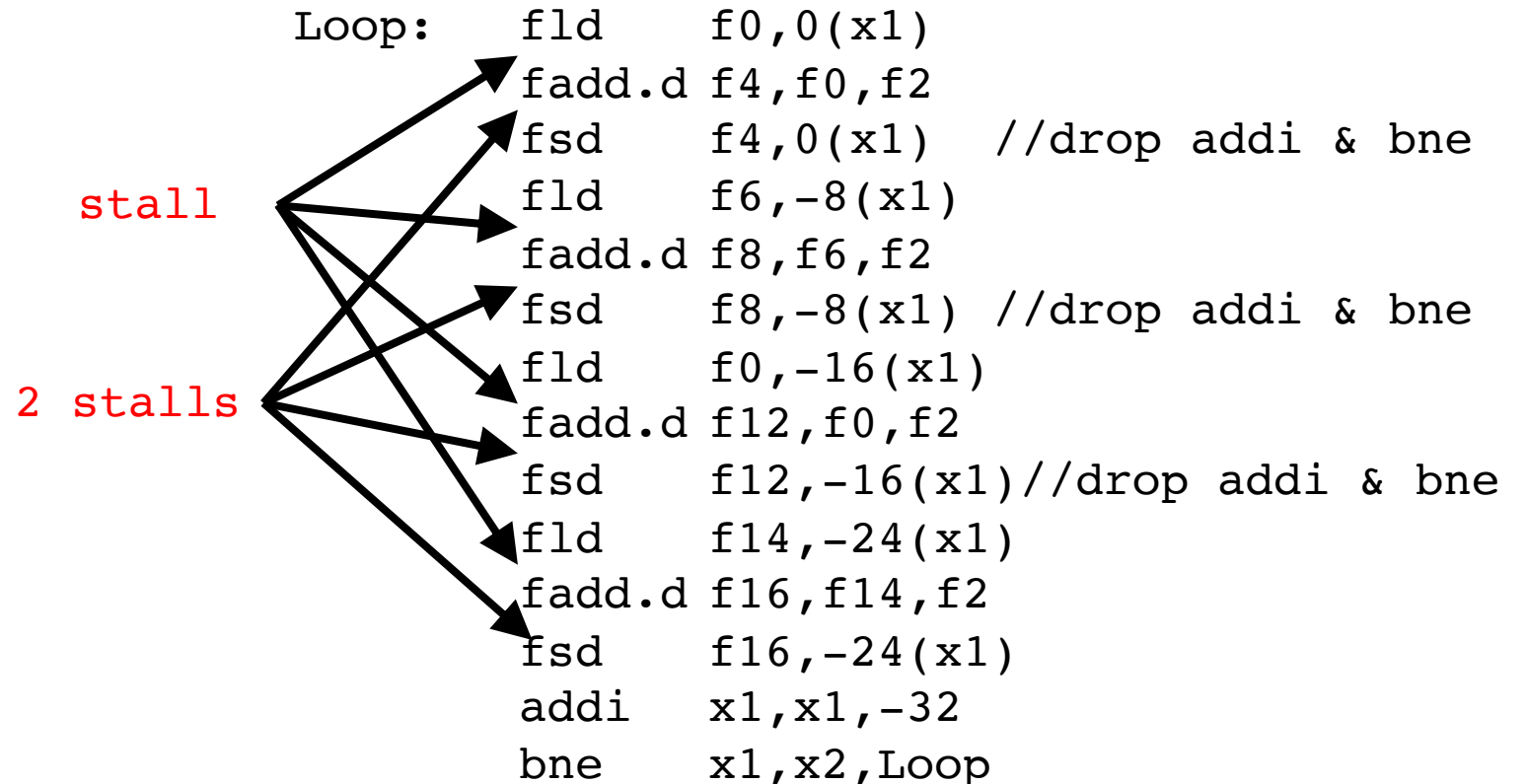
```
Loop: fld    f0,0(x1)
      addi   x1,x1,-8
      fadd.d f4,f0,f2
      stall
      stall
      fsd    f4,0(x1)
      bne    x1,x2,Loop
```



# Loop Unrolling

- **Loop unrolling**

- Unroll by a factor of 4  
(assume # elements is divisible by 4)
- Eliminate unnecessary instructions



30 cycles (if the pipeline is initially empty)  
26 cycles (if the pipeline is initially full)



## Loop Unrolling (cont.)

- Pipeline schedule the unrolled loop:

```
Loop: fld      f0,0(x1)
      fld      f6,-8(x1)
      fld      f8,-16(x1)
      fld      f14,-24(x1)
      fadd.d    f4,f0,f2
      fadd.d    f8,f6,f2
      fadd.d    f12,f0,f2
      fadd.d    f16,f14,f2
      fsd      f4,0(x1)
      fsd      f8,-8(x1)
      fsd      f12,-16(x1)
      fsd      f16,-24(x1)
      addi     x1,x1,-32
      bne      x1,x2,Loop
```

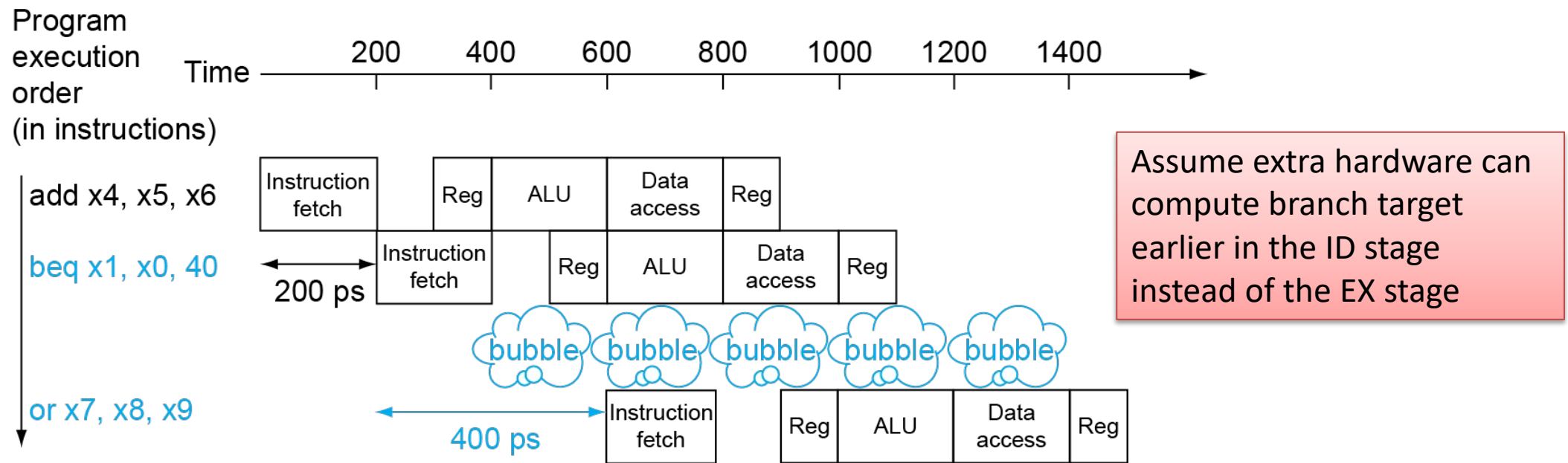
18 cycles (if the pipeline is initially empty)  
14 cycles (if the pipeline is initially full)

## Control Hazards (or Branch Hazards)

- Branch determines the flow of control
  - Fetching next instruction depends on branch outcome
  - Pipeline can't always fetch correct instruction, as still working on ID stage of branch
- In RISC-V pipeline
  - Need to compare registers and compute target early in the pipeline
  - Add hardware to do it in ID stage

## Stall on Branch

- Wait until branch outcome determined before fetching next instruction



- But even with added hardware to compare registers, compute target and update PC in the ID stage, there is **still a pipeline stall for every conditional branch – need additional solution**

## Performance of “Stall on Branch”

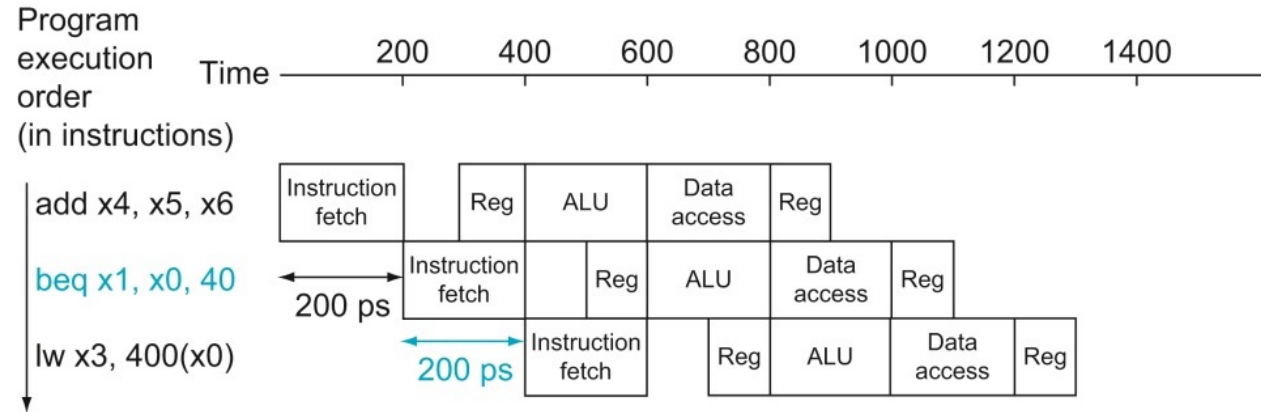
- Conditional branches are 10% of the instructions executed in SPECint2006 and assume instructions have a CPI of 1
- If every conditional branch took one extra clock cycle for the stall
- Then weighted average CPI == 1.10
  - i.e., a slowdown of 1.10 versus the ideal case
- Stall penalty becomes too high and unacceptable for longer pipelines can't readily determine branch outcome early

## Solution: Branch Prediction

- The solution is to **predict the outcome of branch**
  - If correct, it doesn't slow down the pipeline
  - Only stall if prediction is wrong
- In RISC-V pipeline
  - A simple solution is to **predict branches not taken** (i.e., take the next instruction,  $PC = PC + 4$ )
  - **Fetch instruction after branch, with no delay**

# Branch Prediction

Prediction  
correct  
(branch  
untaken)



Prediction  
incorrect  
(branch  
taken)

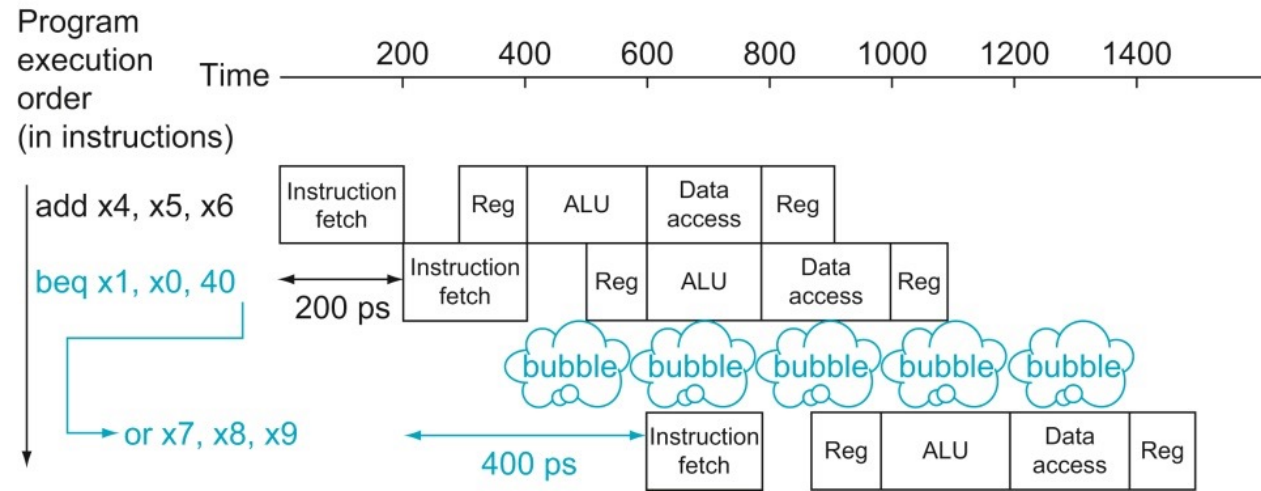


Figure 4.34 Predicting that branches are not taken as a solution to control hazard

## More-Realistic Branch Prediction

- Static branch prediction
  - Based on typical branch behavior
  - Example: loop branches (more likely more than one iteration)
    - Predict backward branches taken (jump back)
    - Predict forward branches not taken
- Dynamic branch prediction
  - Hardware measures actual branch behavior
    - e.g., record recent history of each branch
  - Assume future behavior will continue the trend
    - When wrong, stall while re-fetching, and update history

## Readings

- Chapter 3, 3.1-3.3