

EEL 4768

Computer Architecture

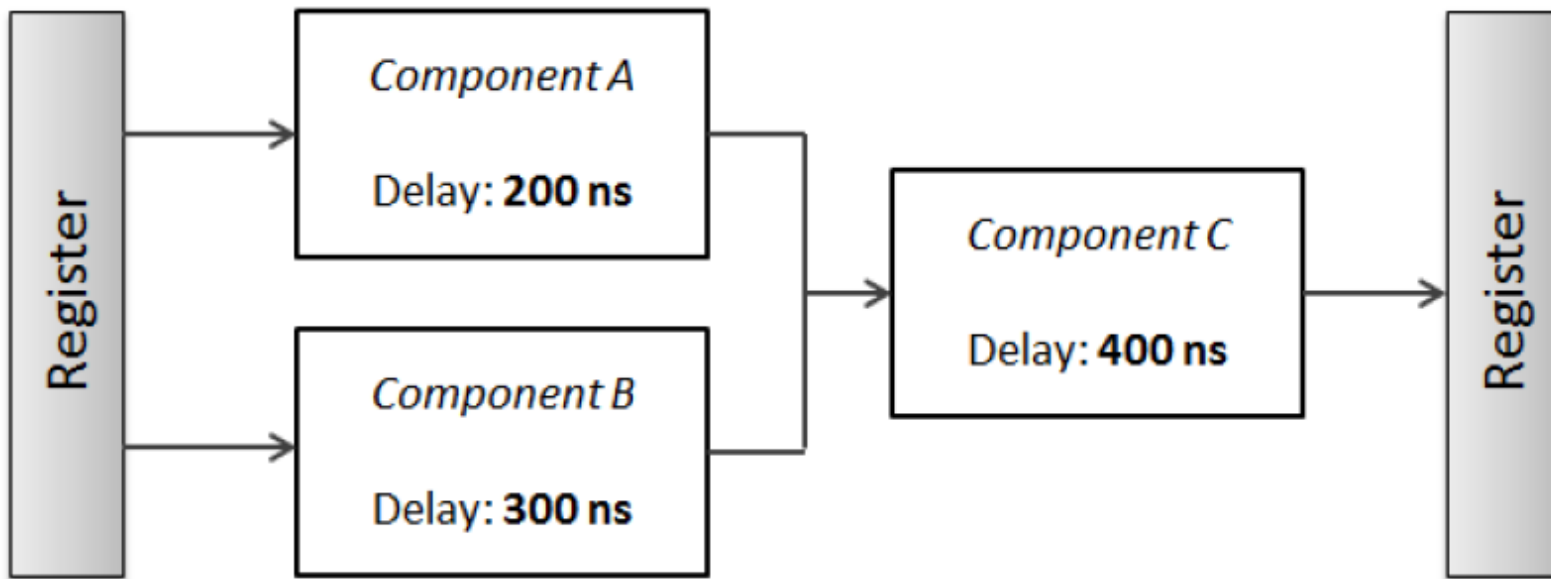
Pipelined Datapath

Outline

- Pipelining defined
- Hazard types
- Data Hazards
 - Forwarding
 - nop
 - Data dependencies
- Control Hazards
 - Branch Prediction

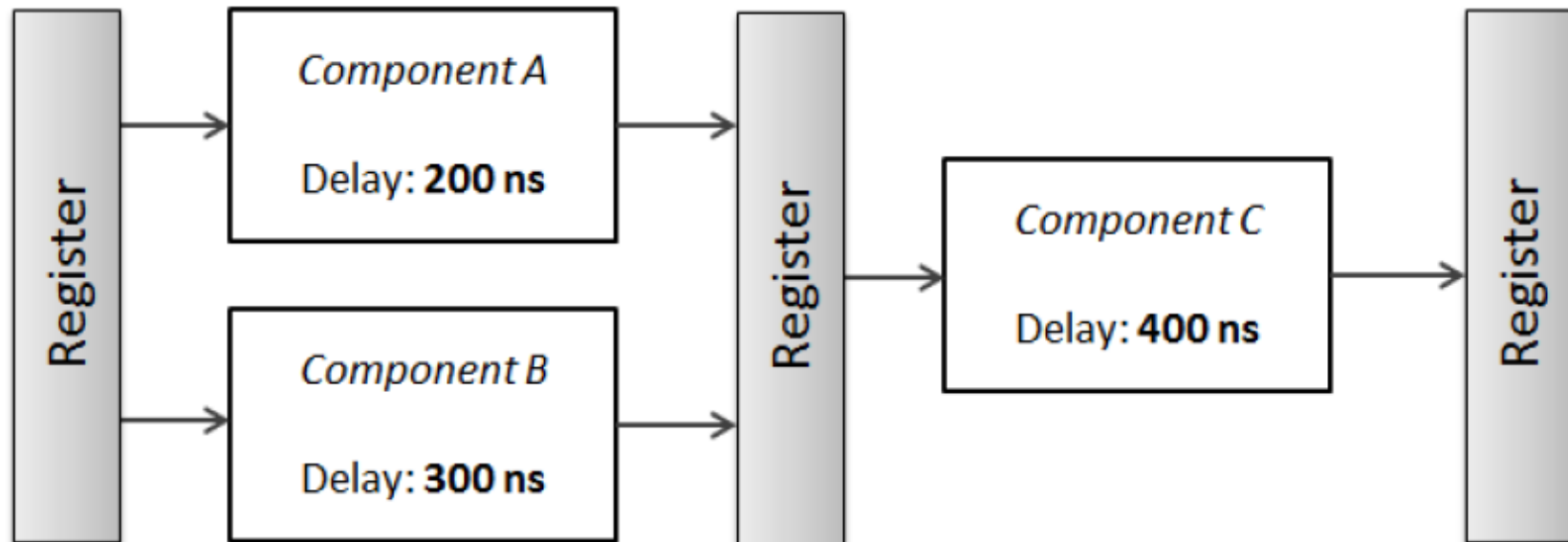
Exercise 1

This figure represents a single-cycle datapath. The components' delays and the data flow is shown in the figure. What would be the clock cycle time of this datapath?



Exercise 2

This figure represents a multi-cycle datapath with two stages. The register in the middle is the stage boundary. The components' delay and the data flow is shown in the figure. What would be the clock cycle time of this datapath?



Clock Cycle Time

- How is the clock cycle time determined in the multi-cycle datapath?
- Let's assume these delay values for the components

Delay of Components in Picosecond (ps)				
Instruction Memory	Register Read	ALU	Memory Access	Register Write
200	50	100	200	50

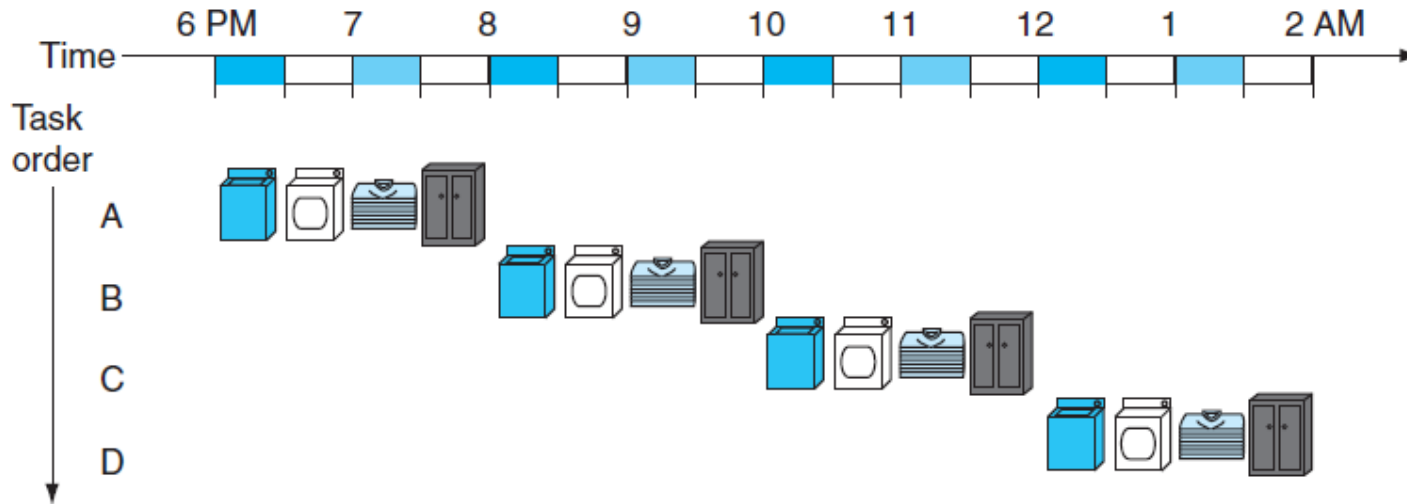
- The clock cycle duration should be large enough to allow any one of the components to function
- Therefore, the clock cycle duration is the maximum value among all the components
- According to the table, the clock cycle duration is 200 ps

Pipelining

- Single-cycle and multi-cycle datapath:
 - There is only one instruction in the datapath
 - When the instruction finishes, the next one gets into the datapath
- With pipelining:
 - Multiple instructions are in the datapath at the same time
 - The instructions run at the same time, so the execution time is faster
- The pipelined datapath has a similarity with the multi-cycle datapath
 - Both have stages and multi-clocks per instruction

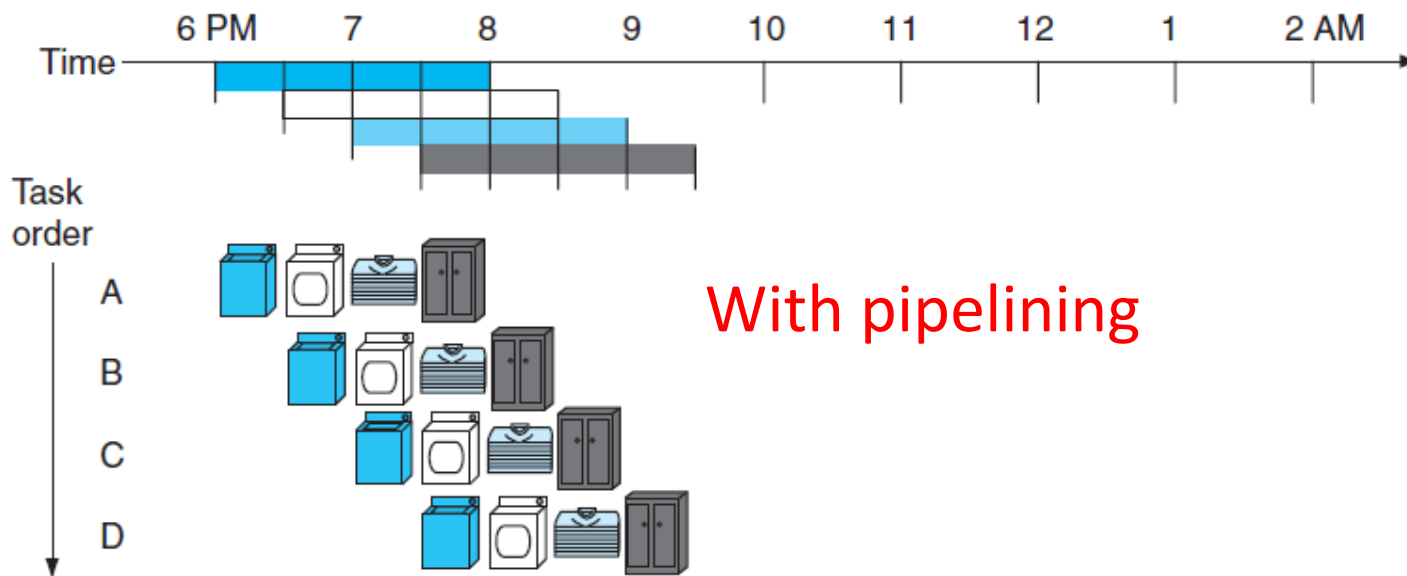
Comparison to Laundry

Without pipelining



Four people are doing their laundry {wash, dry, fold, store}

Without pipelining, there is only one guy in the laundry room. The total time is 8 hours (every task takes 30 mins)



With pipelining, the people go in the laundry room at the same time. Every person uses a stage. The total time is 3.5 hours

Divide the Work into Stages

- To do pipelining, the work must be divided into stages
- For doing the laundry, the stages are:
 - Wash
 - Dry
 - Fold
 - Store
- For the CPU datapath, the stages are:
 - Fetch instruction from memory
 - Decode the instruction (main control unit) and read registers
 - Use the ALU (for R-type, for address computation or for beq)
 - Data memory access
 - Write the result into a register

Pipeline Stages

- 5 stages of the pipelined datapath:
 - IF: Instruction Fetch
 - ID: Instruction Decode (set control and read registers)
 - EX: ALU execution
 - MEM: Memory access
 - WB: Write back to register

Through the Pipeline Stages

- Initially, the pipeline has 5 instructions in it (R1 to R5)
- A new R-type instruction, R6, enters the pipeline
- The instruction R6 has to go through all the stages
- R6 goes through MEM stage even if it will do nothing in this stage
- R6 can't skip from EX to WB since there's another instruction in WB

	IF	ID	EX	MEM	WB
Initial	R5	R4	R3	R2	R1
Clock cycle 1	R6	R5	R4	R3	R2
Clock cycle 2	R7	R6	R5	R4	R3
Clock cycle 3	R8	R7	R6	R5	R4
Clock cycle 4	R9	R8	R7	R6	R5
Clock cycle 5	R10	R9	R8	R7	R6

Performance

- When the pipelined datapath is full:
 - It finishes one instruction per clock cycle
 - But, practically, there are some events which will slow down the datapath a little bit

	IF	ID	EX	MEM	WB
Initial	R5	R4	R3	R2	R1
Clock cycle 1	R6	R5	R4	R3	R2
Clock cycle 2	R7	R6	R5	R4	R3
Clock cycle 3	R8	R7	R6	R5	R4
Clock cycle 4	R9	R8	R7	R6	R5
Clock cycle 5	R10	R9	R8	R7	R6

Issues or Hazards: An Example

Instruction 2 (sub)
reads register t0

	IF	ID	EX	MEM	WB
Clock cycle 1	add				
Clock cycle 2	sub	add			
Clock cycle 3		sub	add		
Clock cycle 4			sub	add	
Clock cycle 5				sub	add
Clock cycle 6					sub

add	\$t0, \$s1, \$s2
sub	\$s3, \$t0, \$s4

Instruction 1 (add)
writes to register t0

This scenario causes a problem in the pipelined datapath:

- The first instruction (add) updates \$t0 in clock cycle 5
- The second instruction (sub) reads \$t0 in clock cycle 3
- Therefore, the 'sub' has used the old value of \$t0; it was supposed to use \$t0 after it was updated by the 'add'
- This will produce a wrong result for the 'sub' instruction

Instructions

- Our pipelined datapath will implement these instructions:
 - add
 - sub
 - and
 - or
 - slt (set on less than)
 - lw (load word)
 - sw (store word)
 - beq (branch on equal)

Clock Cycle Time

- What should the clock cycle time be for the pipelined datapath vs. the single-cycle datapath?
- These are the delays of the components

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (lw)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (sw)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (add, sub, and, or, slt)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (beq)	200 ps	100 ps	200 ps			500 ps

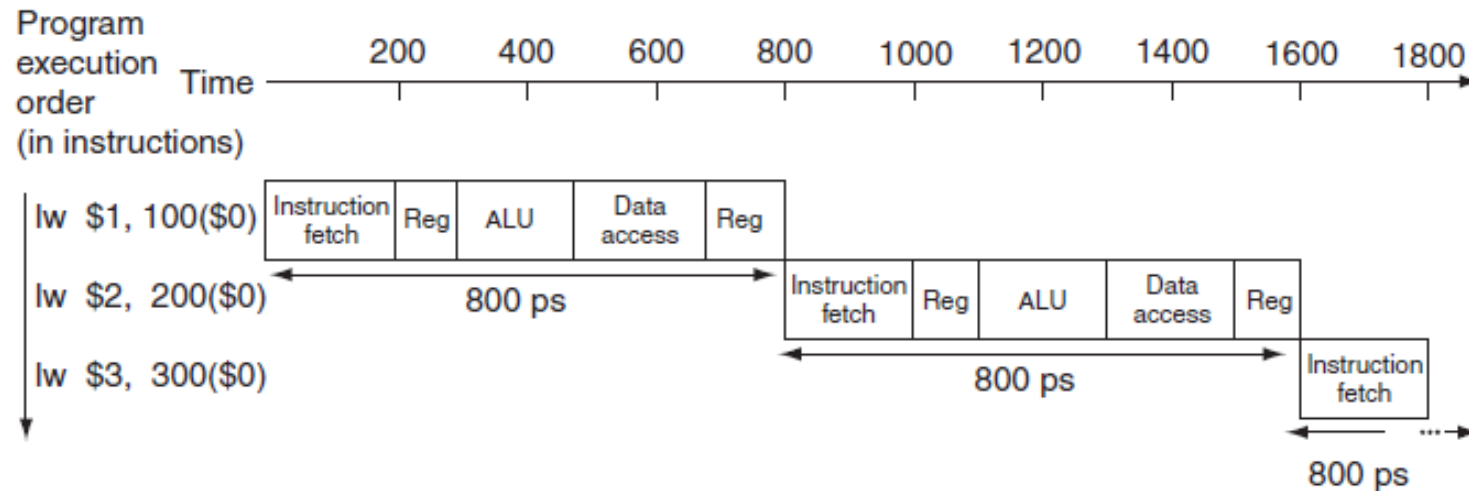
- Single-cycle datapath: clock cycle time = 800 ps
- Pipelined or multi-cycle datapath: clock cycle time = 200 ps

Performance Speedup

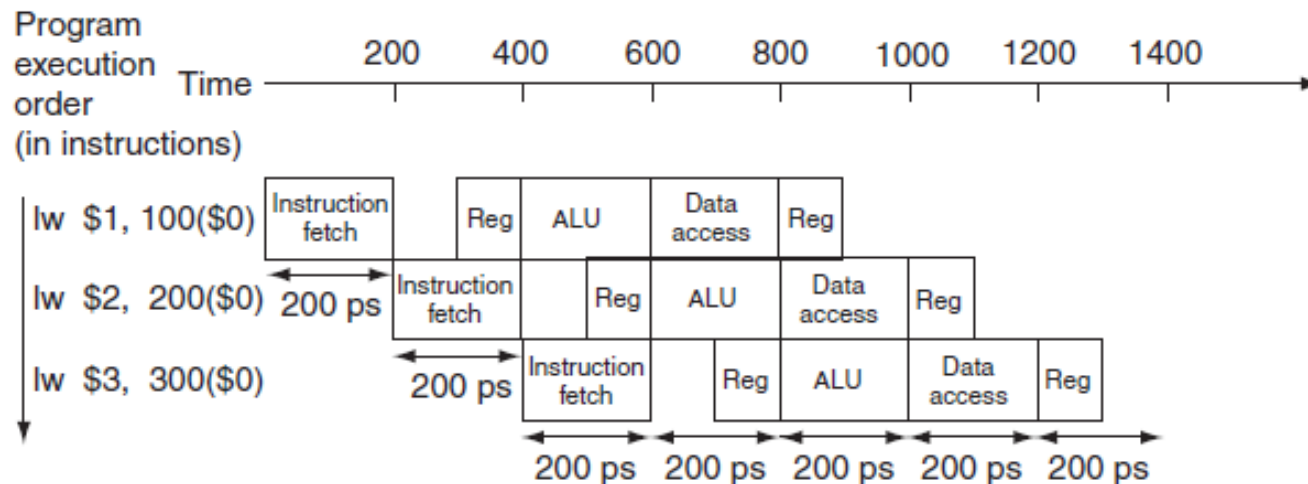
- The number of stages in the pipelined datapath is considered as the potential speedup
- For example, the 5-stage pipelined datapath can have a speedup of 5 compared to a similar multi-cycle datapath
- In the multi-cycle datapath, the instruction takes up to 5 clock cycles
- But in the pipelined, one instruction is finished every clock cycle when the datapath is operating at full speed
- Practically, the pipelined datapath is slowed down sometimes; so the speedup is less than 5

Without and With Pipelining

Without pipelining, when an instruction finishes, the next one starts

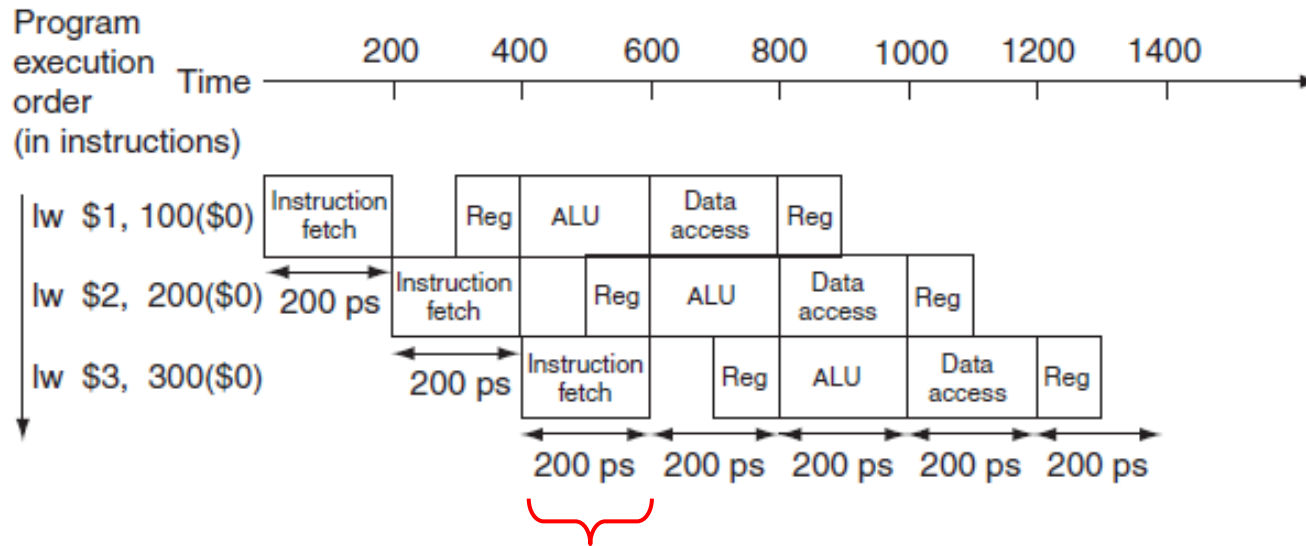


With pipelining, the next instruction follows after one clock cycle. There are more than one instruction in the datapath at the same time.



Pipelining

- The instructions in the datapath are using distinct components



- This is what's happening during the clock cycle [400-600]
 - lw \$1, 100(\$0) is using the ALU
 - lw \$2, 200(\$0) is using the register file
 - lw \$3, 300(\$0) is fetching an instruction from the memory
- These instructions can co-exist in the datapath without conflict
- However, a conflict might occur sometimes

Hazards

- A hazard is a case where the instructions in the datapath have a conflict
- **Structural hazard:**
 - This is a conflict over a component
 - For example, there are two instructions in the datapath that need to access the memory in the same clock cycle
 - This is a problem since the memory can be accessed by one instruction only during a clock cycle
- **Data hazard:**
 - This is a conflict over data
 - An instruction needs a data that hasn't been computed yet

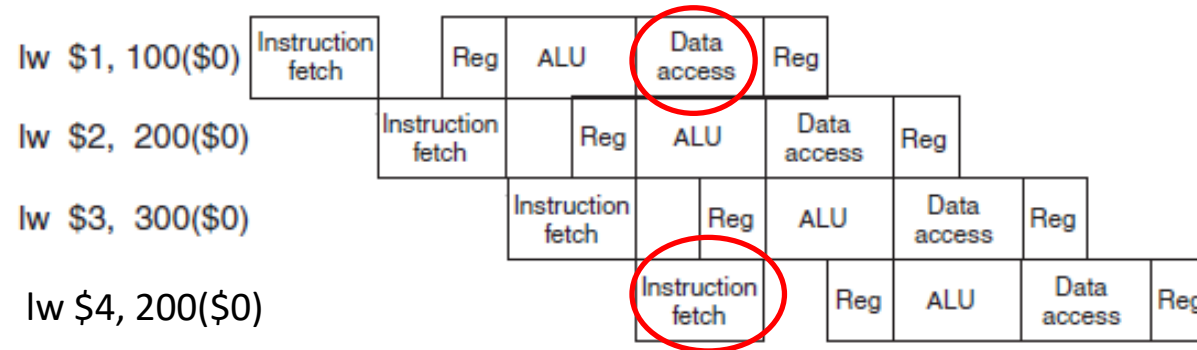
Hazards

- **Control hazard:**

- The term ‘control’ refers to program control using if-else statements; which corresponds to using ‘branch’ instructions in assembly
- When a ‘branch’ instruction enters the pipeline, it takes a few clock cycles to know the branch decision
- In the meanwhile, should we allow more instructions in the pipeline?
- The problem here is we don’t know what instructions should follow the branch
 - If the branch is not taken, we should process the instructions at PC+4, PC+8, ...
 - If the branch is taken, then we should process the instructions that are located at the branch address

Structural Hazard

- We have the following pipelined datapath



Two instructions need to access the memory in the same clock cycle; this is a structural hazard. So, we need to split the memory into “Instruction Memory” and “Data Memory”

- Remember:
 - In single-cycle datapath, we had instruction memory and data memory
 - In multi-cycle implementation, we combined them into one memory
- In pipelined datapath, we go back to having two separate memories (one for instructions and one for data)

Data Hazard

- Let's consider this instruction sequence:

```
add $s0, $t0, $t1          # this is Instruction 1
sub $t0, $s0, $t3          # this is Instruction 2
```

	'sub' instruction	'add' instruction
1 st clock cycle		Instruction fetch
2 nd clock cycle	Instruction fetch	Read registers
3 rd clock cycle	Read registers (s0, t3)	ALU operation
4 th clock cycle	ALU operation	Memory
5 th clock cycle	Memory	Write to register (s0)
6 th clock cycle	Write to register	

- Instruction 2 has read the value of s0 before it has been written by Instruction 1. Therefore, Instruction 2 got the wrong s0 and will give the wrong result.

How to Fix the Data Hazard?

```
add $s0, $t0, $t1      # this is Instruction 1
sub $t0, $s0, $t3      # this is Instruction 2
```

	'sub' instruction	'add' instruction
1 st clock cycle		Instruction fetch
2 nd clock cycle	** bubble **	Instruction decode
3 rd clock cycle	** bubble **	Execute
4 th clock cycle	** bubble **	Memory
5 th clock cycle	Instruction fetch	Write back (s0)
6 th clock cycle	Read registers (s0, t3)	
7 th clock cycle	ALU operation	
8 th clock cycle	Write to register	

- By delaying Instruction 2 by three clock cycles, now Instruction 2 will read 's0' after this register has been written by Instruction 1.
- Delaying Instruction 2 by three clock cycles is referred to as inserting three 'bubbles' in the datapath (which is considered as overhead)

A Slightly Better Fix

```
add $s0, $t0, $t1      # this is Instruction 1
sub $t0, $s0, $t3      # this is Instruction 2
```

	'sub' instruction	'add' instruction
1 st clock cycle		Instruction fetch
2 nd clock cycle	** bubble **	Instruction decode
3 rd clock cycle	** bubble **	Execute
4 th clock cycle	Instruction fetch	Memory
5 th clock cycle	Read registers (s0, t3)	Write back (s0)
6 th clock cycle	ALU operation	
7 th clock cycle	Write to register	

- As shown above, we can write s0 in the 1st half of clock cycle 5
- Then, 'sub' reads s0 in the 2nd half of clock cycle 5
- The register file has a small delay since it's a small component; so we can write then read in the same clock cycle.
- Therefore, we need 2 bubbles instead of 3

Bubbles in the Datapath

```
add $s0, $t0, $t1      # this is Instruction 1
nop                     # nop: no operation (bubble)
nop
sub $t0, $s0, $t3       # this is Instruction 2
```

	IF	ID	EX	MEM	WB
Clock cycle 1	add				
Clock cycle 2	Bubble	add			
Clock cycle 3	Bubble	Bubble	add		
Clock cycle 4	sub	Bubble	Bubble	add	
Clock cycle 5		sub	Bubble	Bubble	add
Clock cycle 6			sub	Bubble	Bubble
Clock cycle 7				sub	Bubble
Clock cycle 8					sub

In cycle 5, 'add' writes \$s0 in the first half of the clock cycle, then 'sub' reads \$s0 in the second half of the clock cycle.

Should Compilers Remove Data Hazards?

- We can optimize the compiler such that no data hazards occur
- How?

```
add $s0, $t0, $t1           # this is Instruction 1
<we need 2 bubbles here> or <insert 2 other instructions>
sub $t0, $s0, $t3           # this is Instruction 2
```
- Instead of placing 2 bubbles, we can insert 2 instructions that don't have data dependency with the add or the sub instructions
 - They shouldn't read register \$s0
 - They shouldn't modify registers \$s0 or \$t3
- The compiler may be able to find such 2 instructions:
 - Dependencies occur too often
 - So the compiler might not find 2 instructions to insert there
- So, a cure for data hazards in the datapath is still very valuable

Forwarding

- Forwarding is a solution to solve data hazards

`add $s0, $t0, $t1 # this is Instruction 1`

`sub $t0, $s0, $t3 # this is Instruction 2`

	'sub' instruction	'add' instruction
1 st clock cycle		Instruction fetch (IF)
2 nd clock cycle	Instruction fetch (IF)	Instruction Decode (ID); Read registers
3 rd clock cycle	Instruction Decode (ID); Read registers (s0 , t3)	Execute (EX); ALU operation (compute s0)
4 th clock cycle	Execute (EX); ALU operation (uses s0)	Memory (MEM)
5 th clock cycle	Memory (MEM)	Write Back (WB) to register (s0)
6 th clock cycle	Write back to register (WB)	

Will read the wrong value of s0

Is computing s0

Forwarding

- Forwarding is a solution to solve data hazards

`add $s0, $t0, $t1 # this is Instruction 1`

`sub $t0, $s0, $t3 # this is Instruction 2`

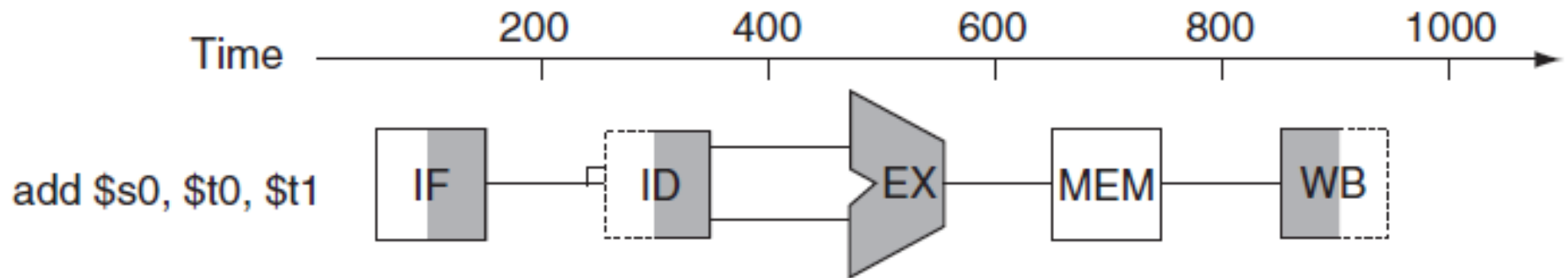
	'sub' instruction	'add' instruction
1 st clock cycle		Instruction fetch (IF)
2 nd clock cycle	Instruction fetch (IF)	Instruction Decode (ID); Read registers
3 rd clock cycle	Instruction Decode (ID); Read registers (\$s0, \$t3)	Execute (EX); ALU operation (compute \$s0)
4 th clock cycle	Execute (EX); ALU operation	Memory (MEM)
5 th clock cycle	Memory (MEM)	Write Back (WB) to register (\$s0)
6 th clock cycle	Write back to register (WB)	

This is when Instruction 2 needs \$s0

The new value of \$s0 is with Instruction 1 in the MEM stage

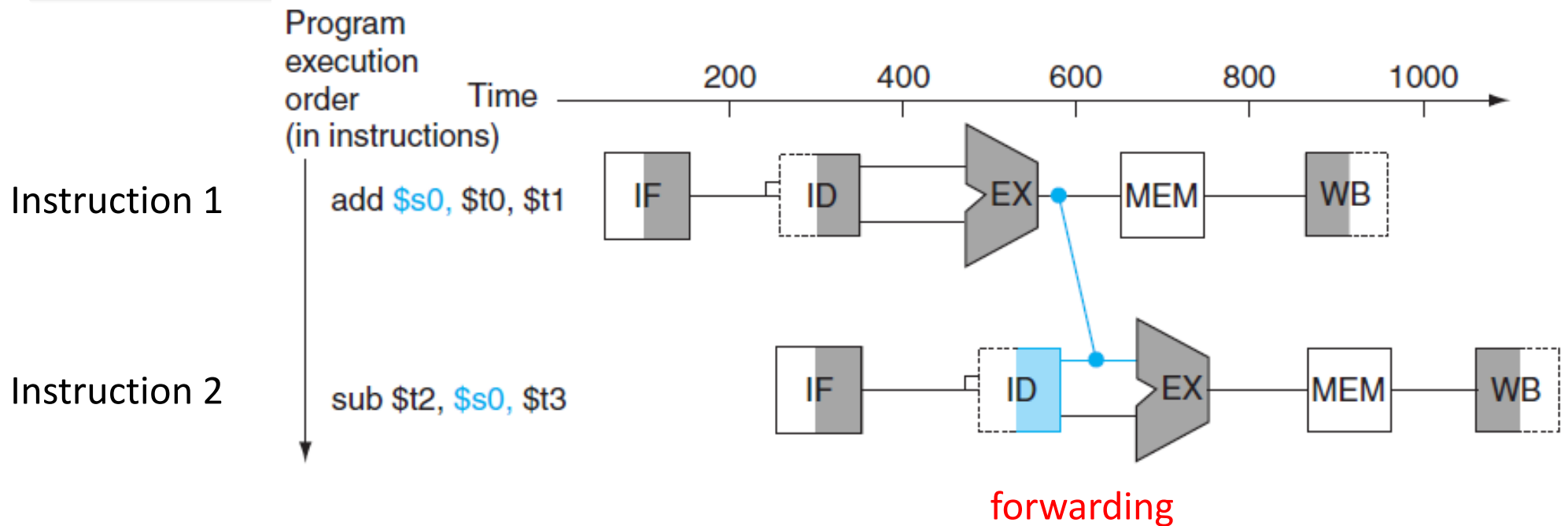
So we forward \$s0 from the MEM stage to the EX stage

Pipelined Datapath



- The 5 stages
 - IF: Instruction Fetch (read the instruction from memory)
 - ID: Instruction Decode (read the registers and set the controls)
 - EX: Execute (ALU operation)
 - MEM: Memory (read from or write to memory)
 - WB: Write Back (write data to register)

Forwarding



- Instruction 2 needs s0 at time 600
- Instruction 1 has produced s0 at time 600
- The new value of s0 has not yet been written in the register file
- Instruction 2 takes s0 from the ALU output at time 600; This is “forwarding”
- In this case, forwarding solves the problem, we don't need to insert bubbles (inserting bubbles is also referred to as “stalling the pipeline”)

Forwarding

```
lw $s0, 20($t1)    # this is Instruction 1
sub $t0, $s0, $t3   # this is Instruction 2
```

	'sub' instruction	'lw' instruction
1 st clock cycle		Instruction fetch (IF)
2 nd clock cycle	Instruction fetch (IF)	Instruction Decode (ID); Read registers
3 rd clock cycle	Instruction Decode (ID); Read registers (<i>s0, t3</i>)	Execute (EX); ALU operation
4 th clock cycle	Execute (EX); ALU operation (<i>uses s0</i>)	Memory (MEM) (<i>s0 being computed</i>)
5 th clock cycle	Memory (MEM)	Write Back (WB) to <i>register (s0)</i>
6 th clock cycle	Write back to register (WB)	

The new value of s0 is needed by Instruction 2 at start of the 4th clock cycle

The new value of s0 becomes available from the memory at the end of the 4th clock cycle

Forwarding is not possible! We need to stall the pipeline for 1 clock cycle

Forwarding

```
lw $s0, 20($t1)    # this is Instruction 1
sub $t0, $s0, $t3   # this is Instruction 2
```

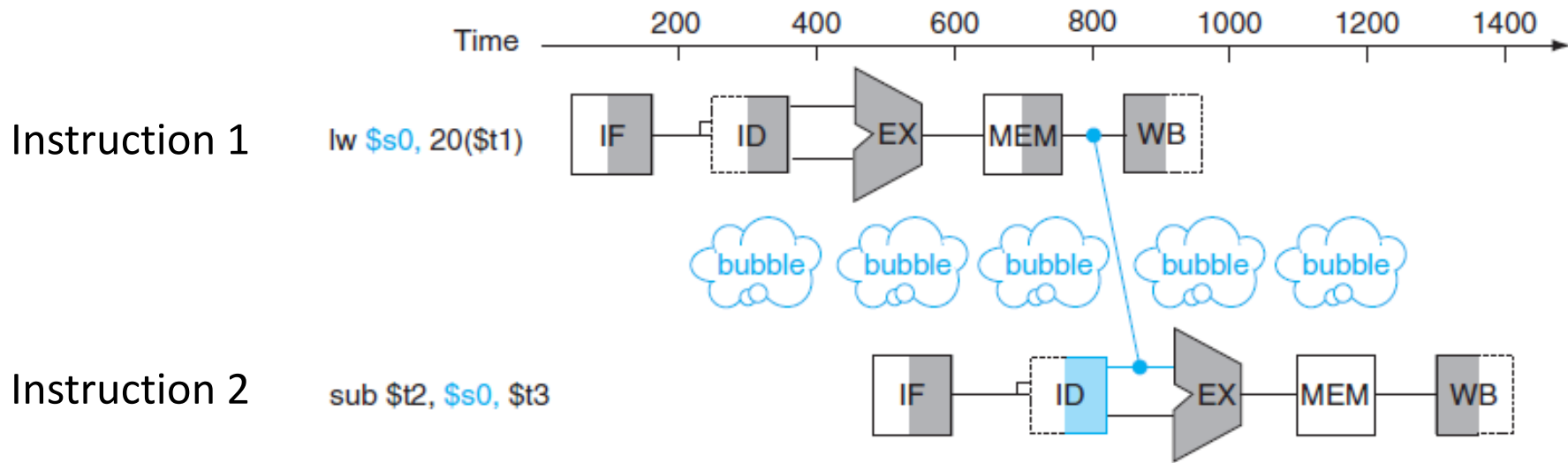
	'sub' instruction	'lw' instruction
1 st clock cycle		Instruction fetch (IF)
2 nd clock cycle	Instruction fetch (IF)	Instruction Decode (ID); Read registers
3 rd clock cycle	Instruction Decode (ID); Read registers (s0, t3)	Execute (EX); ALU operation (compute s0)
4 th clock cycle	*** bubble ***	Memory (MEM)
5 th clock cycle	Execute (EX); ALU operation	Write Back (WB) to register (s0)
6 th clock cycle	Memory (MEM)	
7 th clock cycle	Write back to register (WB)	

The new value of s0 is needed by Instruction 2 at start of the 4th clock cycle

The new value of s0 is read from the memory at the end of the 4th clock cycle

We stall for 1 cycle; in the 5th clock cycle, we forward from WB stage to EX stage.

Forwarding doesn't always prevent a stall



- Instruction 1 starts at time $t=0$
- It produces `s0` in the 4th stage at time $t=800$
- **Let Instruction 2 start right after Instruction 1 at $t=200$ (where the bubbles are shown)**
- Instruction 2 needs `s0` in its 3rd stage at $t=600$
- It's not possible to do a forwarding since `s0` is produced by Instruction at $t=800$
- **Let's insert 1 bubble in the datapath**
- Now Instruction 2 starts at $t=400$
- Instruction 2 needs `s0` in its 3rd stage at $t=800$
- Instruction 1 produces `s0` at $t=800$
- So, it is possible to forward `s0` from the MEM stage to the EX stage

Summary so far...

- **R-type followed by R-type with data dependency**
add \$s0, \$t0, \$t1
sub \$s1, \$s0, \$t2
- Solution 1:
 - Insert 2 bubbles in the pipeline datapath
- Solution 2: (better solution)
 - Forward \$s0 from MEM stage to EX stage (no bubbles needed)
- **Load word (lw) followed by an R-type with data dependency**
lw \$s0, 0(\$a0)
add \$s1, \$s0, \$a1
- Solution:
 - Insert 1 bubble in the datapath (stall for 1 cycle)
 - Forward \$s0 from WB stage to EX stage

Finding the Data Dependencies in a Code

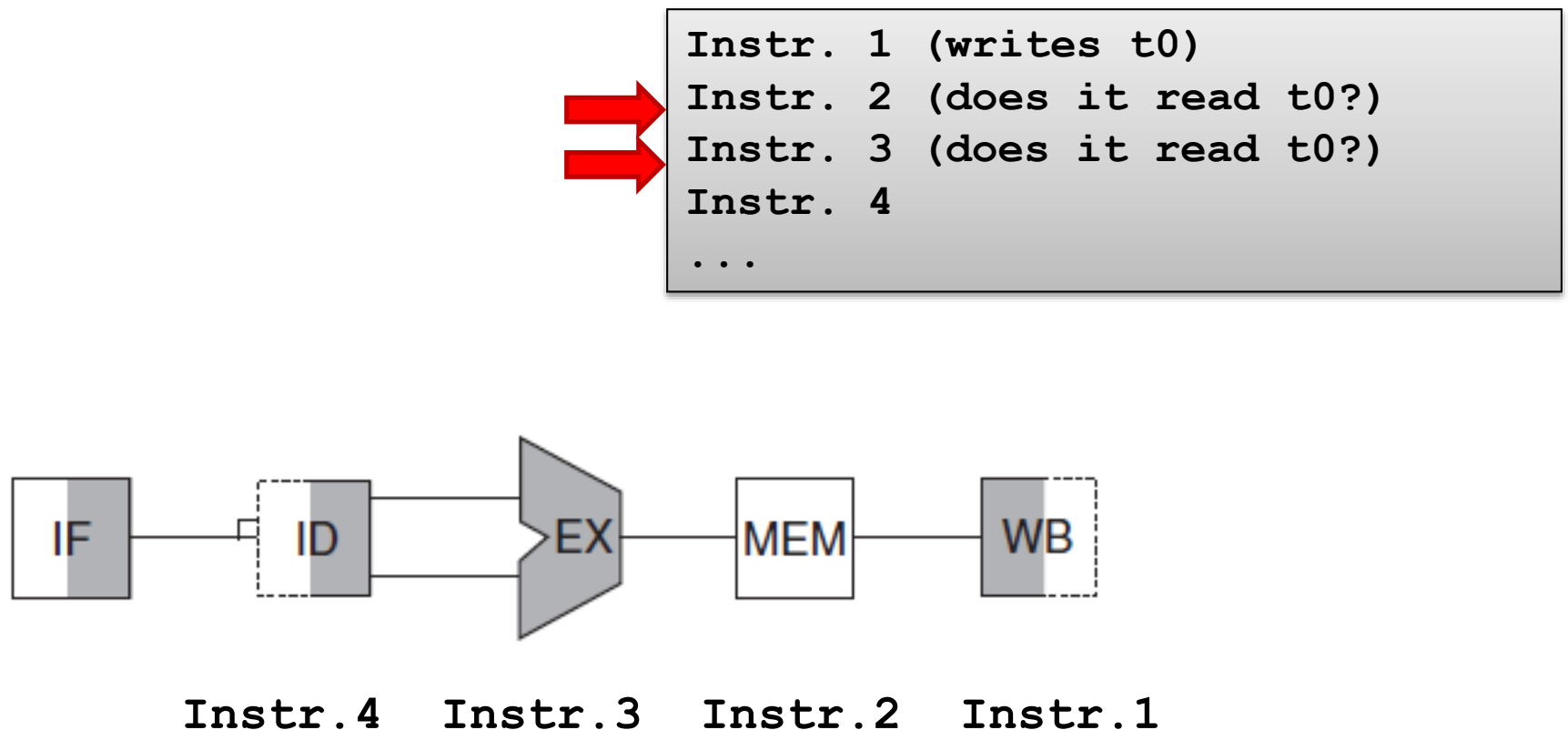
- If an instruction modifies a register, look for the two following instructions to see if they read the same register
- **Only the subsequent two instructions** have a risk of data dependency
- In this code, Instr.1 is writing to t0, we only need to check Instr.2 and Instr.3 for data dependency
- We don't have to check for Instr. 4



```
Instr. 1 (writes t0)
Instr. 2 (does it read t0?)
Instr. 3 (does it read t0?)
Instr. 4
...
```

Finding the Data Dependencies in a Code

- If Instr.4 is reading t0, it will get the correct value since the register write happens before the register read



Data Dependencies

- C code and corresponding assembly code:

$A = B + E;$

$C = B + F;$

lw	\$t1, 0(\$t0)	# load B
lw	\$t2, 4(\$t0)	# load E
add	\$t3, \$t1, \$t2	# add B and E (result in A)
sw	\$t3, 12(\$t0)	# write A to memory
lw	\$t4, 8(\$t0)	# load F
add	\$t5, \$t1, \$t4	# add B and F (result in C)
sw	\$t5, 16(\$t0)	# write C to memory

- Identify all the data hazards in this code?
 - Which ones can be solved by forwarding?
 - Which ones can't be solved by forwarding?

Data Dependencies

- These are the data dependencies:

lw	\$t1, 0 (\$t0)	# load B
lw	\$t2, 4 (\$t0)	# load E
add	\$t3, \$t1, \$t2	# add B and E (result in A)
sw	\$t3, 12 (\$t0)	# write A to memory
lw	\$t4, 8 (\$t0)	# load F
add	\$t5, \$t1, \$t4	# add B and F (result in C)
sw	\$t5, 16 (\$t0)	# write C to memory

- 1st and 3rd instructions on \$t1 → Forwarding OK
- 2nd and 3rd instructions on \$t2 → Need a nop and forwarding
- 3rd and 4th instructions on \$t3 → Forwarding OK
- 5th and 6th instructions on \$t4 → Need a nop and forwarding
- 6th and 7th instructions on \$t5 → Forwarding OK

Type of Dependences

Dependence	Example	Fix	Switch Instructions?
Read After Read (RAR)	add t0, <u>t1</u> , t2 or t3, <u>t1</u> , t4	None needed	Acceptable
Read After Write (RAW)	add <u>t0</u> , t1, t2 or t3, <u>t0</u> , t4	Forwarding or nops	No
Write After Read (WAR)	add t0, <u>t1</u> , t2 or <u>t1</u> , t3, t4	None needed	No
Write After Write (WAW)	add <u>t0</u> , t1, t2 or <u>t0</u> , t3, t4	None needed	No

Reordering Instructions to Avoid a Data Hazard

- C code and corresponding assembly code


hazards

	lw	\$t1, 0 (\$t0)	# load B
{	lw	\$t2, 4 (\$t0)	# load E
	add	\$t3, \$t1, \$t2	# add B and E (result in A)
	sw	\$t3, 12 (\$t0)	# write A to memory
{	lw	\$t4, 8 (\$t0)	# load F
	add	\$t5, \$t1, \$t4	# add B and F (result in C)
	sw	\$t5, 16 (\$t0)	# write C to memory

- Instead of inserting two bubbles, one way to deal with this is to rearrange the order of the instructions

Solution 1: Inserting bubbles and forwarding

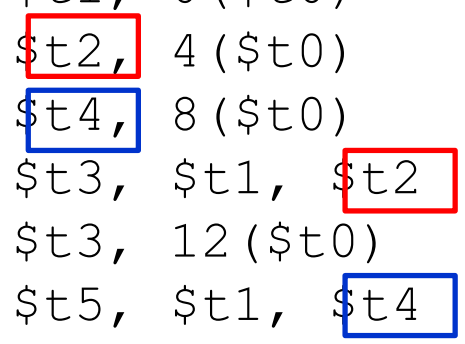
```
lw    $t1, 0($t0)
lw    $t2, 4($t0)
<1 bubble & forward>
add    $t3, $t1, $t2
sw     $t3, 12($t0)
lw     $t4, 8($t0)
<1 bubble & forward>
add    $t5, $t1, $t4
sw     $t5, 16($t0)
```



- 'lw' was moved two instructions up
- When can we rearrange the order?
- Two consecutive instructions can be switched when they don't have a data dependency

Solution 2: Reordering the Instructions

```
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)
```



- Now the 'lw' and 'add' that depend on \$t2 are separated by one instruction; forwarding from WB to EX takes care of this case
- The other 'lw' and 'add' that depend on \$t4 are separated by two instructions; no forwarding is needed here

Rearranging the Order of Two Instructions

- Two consecutive instructions can be switched when neither uses the result of the other

```
add    $t0, $t1, $t2      # These two can't  
sub    $t3, $t0, $t4      # be switched
```

```
and    $t0, $t1, $t2      # These two can't  
lw     $t1, 0($s0)        # be switched
```

```
or     $s0, $s1, $s2      # These two can't  
and    $s1, $s3, $s0      # switched
```

```
lw     $t1, 4($s4)        # It's ok to switch  
and    $s5, $s6, $s7      # these two
```

Rearranging the Order of Two Instructions

- Also, we can't swap two instructions if they write to the same register

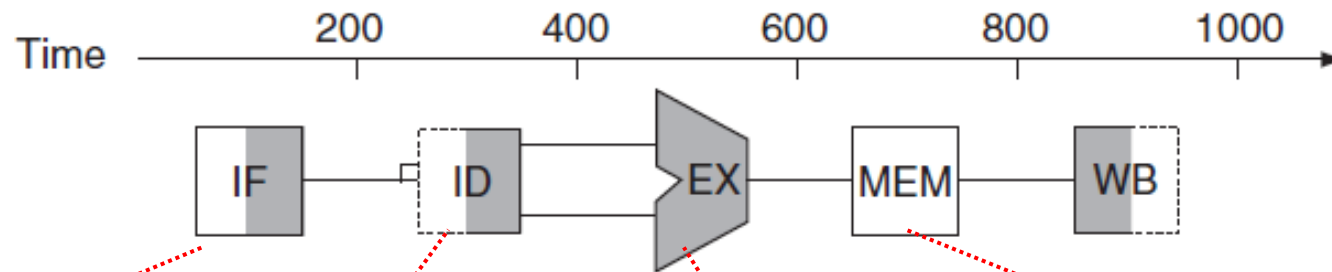
```
add    $t0, $t1, $t2      # These two can't  
sub    $t0, $t3, $t4      # be switched
```

```
lw      $t0, 0($s0)  
sw      $t1, 0($s1)
```

Can we switch the two instructions above?
We don't know. The registers \$s0 and \$s1 might be equal

Branch Instruction “beq”

- How is the branch executed in the pipeline?
- Approach #1:



Instruction is fetched and PC+4 is computed

The two registers are read

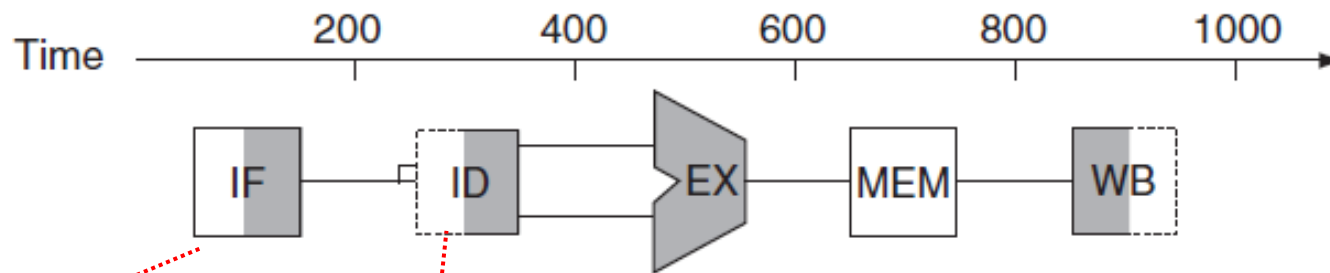
- The ALU compares the registers.
- Add an adder to compute the branch address.

Write the branch address in the PC

Accordingly, the branch is finished in the MEM stage

Branch Instruction “beq”

- How is the branch executed in the pipeline?
- Approach #2:

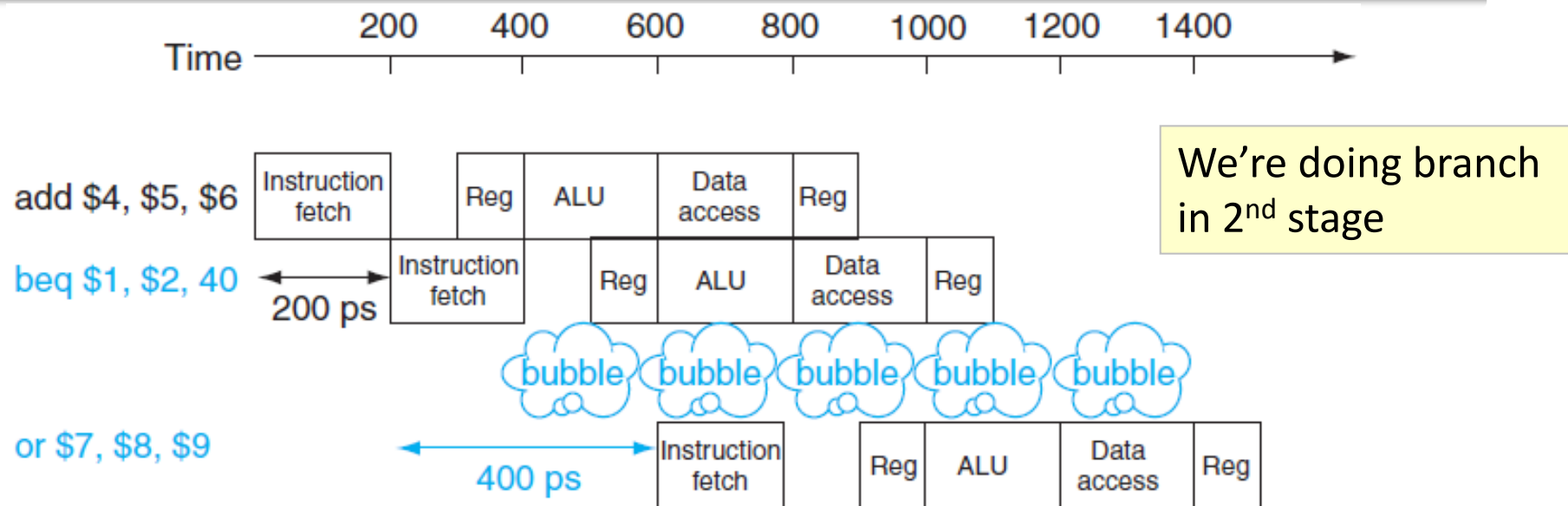


Instruction is
fetched and
PC+4 is
computed

- Add a comparator here to compare the registers.
- Add an adder to compute the branch address (PC+4+offset)

Accordingly, the branch is finished in the ID stage

Branch Instruction (beq); stall on branch

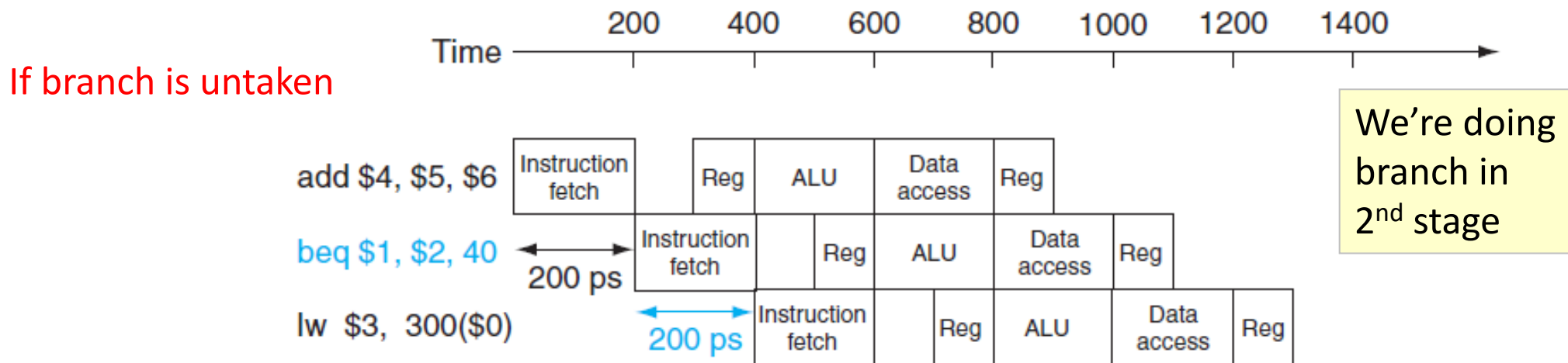


- 'beq' instruction starts at the clock cycle [200:400]
- In the second cycle [400:600], the branch decision will be known
- Since at t=400, we don't know the branch decision yet, we're not sure whether the next instruction is:
 - PC+4 (branch not taken), or
 - Instruction at branch address (branch taken)
- That's why we insert a bubble
- At t=600, the branch decision is known, so the new instruction is fetched

Branch Prediction

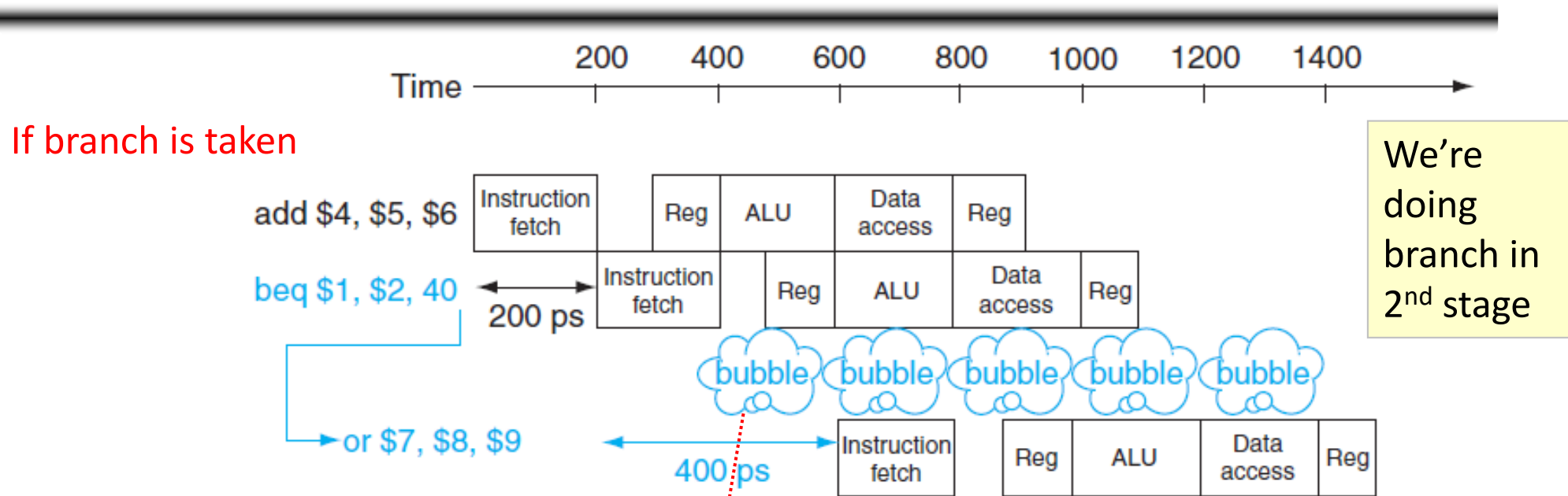
- When a branch instruction is in the IF and ID stages, we don't know if the branch will be taken
- We assume that the branch will not be taken
 - Therefore, we fetch the next instruction into the pipeline
- If the branch is not taken, the execution continues fast because we didn't waste any time
- If the branch is taken, we flush the instruction that we've fetched after "beq"

Branch Instruction (beq); predict branch untaken



- Predict branch untaken
- Fetch the instruction at PC+4 in the next cycle
- If the prediction is correct (no branch), the datapath proceeds at full speed
- We will know the result of branch at t=600
- Until this time, instruction at PC+4 has only been fetched (it hasn't modified data)
- So if the branch will not be taken, we remove it from the datapath

Branch Instruction (beq); predict branch untaken



- Predict branch untaken
- So we fetched the instruction at PC+4 here
- At t=600, we found out that the branch will be taken
- We remove the instruction at PC+4 from the datapath
 - This instruction has only been fetched, it hasn't modified data (so we can remove it)
- The instruction at PC+4 is replaced with a bubble
- The instruction at the branch address is brought in the datapath in the next cycle

Predict branch untaken solution: 1) Proceeds at full speed when branch is untaken.
2) Wastes 1 clock cycle when branch is taken

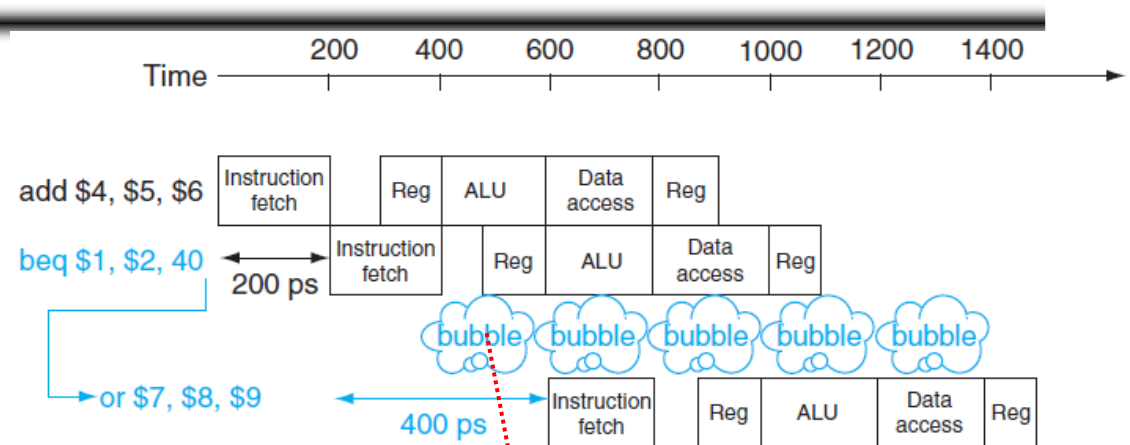
Branch Instruction (beq); delayed branch

- We have the following code:

```
...  
add    $t0, $t1, $t2  
beq     $s2, $s3, Loop  
...
```

- The execution order is reversed

```
...  
beq     $s2, $s3, Loop  
add     $t0, $t1, $t2  
...
```



Also, we always execute the instruction after the 'beq'

- Instead of wasting 1 clock cycle for the bubble, we do an instruction after the 'beq' that is useful
- The condition is that the 'add' doesn't modify the operands of 'beq'
- Sometimes, we might not be able to find such an instruction, so this mechanism isn't always usable

Summary on Branch (beq)

- **Stall on branch:**
 - Stall the pipeline (for 1 clock cycle) until the result of the branch is known
- **Predict branch untaken:**
 - If branch untaken, no bubble is needed (no time is wasted)
 - If branch is taken, 1 bubble is needed
- **Delayed branch**
 - Find an instruction to execute after the branch (it should not modify the operands of 'beq')
 - No time is wasted whether branch is taken or not
 - Sometimes, we might not be able to find such an instruction

Pipelining increases the number of simultaneously executing instructions and the rate at which instructions are started and completed. Pipelining does not reduce the time it takes to complete an individual instruction, also called the **latency**. For example, the five-stage pipeline still takes 5 clock cycles for the instruction to complete. In the terms used in Chapter 4, pipelining improves instruction *throughput* rather than individual instruction *execution time* or *latency*.

Instruction sets can either simplify or make life harder for pipeline designers, who must already cope with structural, control, and data hazards. Branch prediction, forwarding, and stalls help make a computer fast while still getting the right answers.

Quote from the book

Readings



- H&P COD
 - Chapter 4