CS 61C

Fall 2023

RISC-V Pipelining and Hazards

Discussion 8

1 Pre-Check

This section is designed as a conceptual check for you to determine if you conceptually understand and have any misconceptions about this topic. Please answer true/false to the following questions, and include an explanation:

1.1 By pipelining the CPU datapath, each single instruction will execute faster (latency is reduced), resulting in a speed-up in performance.

False. Because we implement registers between each stage of the datapath, the time it takes for an instruction to finish execution through the 5 stages will be longer than the single-cycle datapath we were first introduced with. A single instruction will take multiple clock cycles to get through all the stages, with the clock cycle based on the stage with the longest timing.

1.2 A pipelined CPU datapath results in instructions being executed with higher throughput (than the single-cycle CPU).

True. Recall that throughput is the number of instructions processed per unit time. Pipelining results in a higher throughput because more instructions are run at once, which utilizes more parts of the datapath simultaneously.

1.3 Through adding additional hardware, we can implement two 'read' ports as well as a 'write' port to the RegFile (where registers can be accessed). This solves the hazard of two instructions reading and writing to the same register simultaneously.

False. The addition of independent ports to the RegFile allows for multiple instructions to access the RegFile at the same time (such as one instruction reading values of two operands, while another instruction is writing to a return register). However, this does not work if both instructions are reading and writing to the same register. Some solutions to this data hazard could be to stall the latter instruction by 1 cycle or to forward the read value from a previous instruction, bypassing the RegFile completely.

1.4 All data hazards can be resolved with forwarding.

False. Hazards following `lw` cannot be fully resolved with forwarding because the output is not known until after the MEM stage, making a stall necessary.

1.5 As stalling reduces performance significantly, we generally prefer other solutions to fixing pipeline hazards, even at the cost of complexity or hardware. In a modern-day pipelined CPU, are there still use-cases for stalling to resolve potential hazards? If so, describe a program that would.

True. Say we have the RISC-V program where a0 is a pointer to an array of integers, and we want to load a1 with the first element * 2:

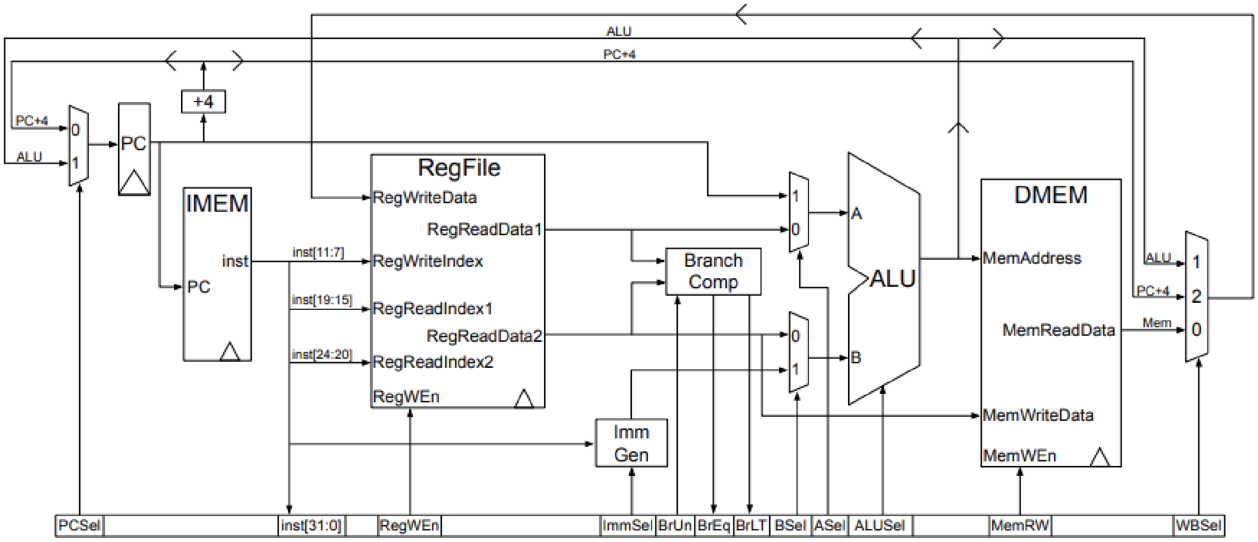
```
lw t1 0(a0)
add t2 t1 t1
mv a1 t2
```

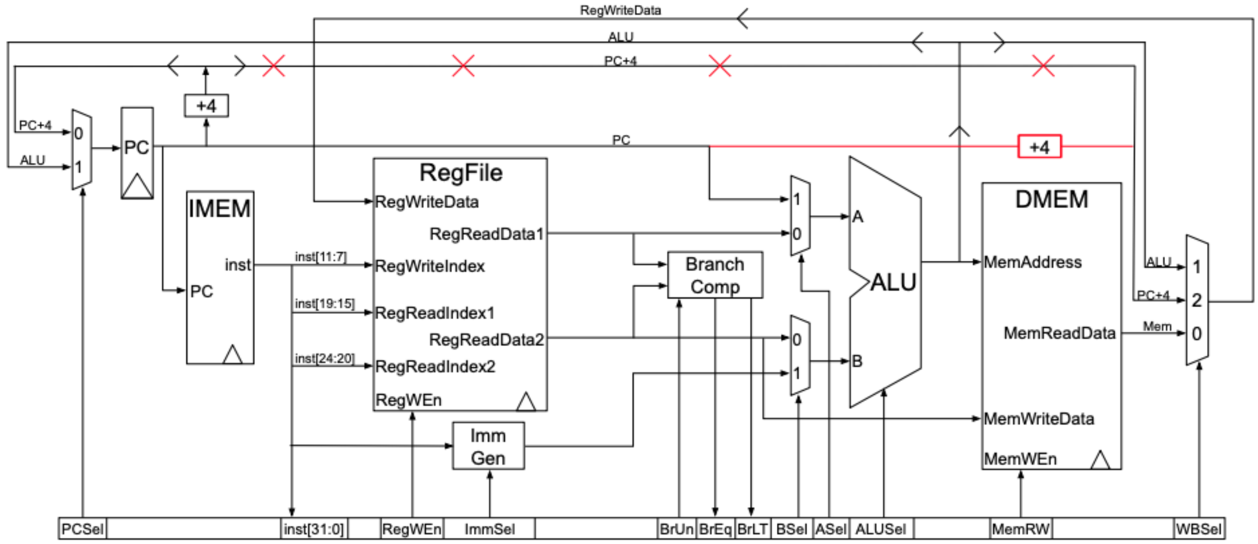
In this program, there are no other instructions to move into the load delay slot, so we are forced to `nop` the next instruction and repeat it afterwards, essentially stalling for one cycle. While we do have many tools and alternative solutions to lessen possible performance loss, in some cases it is unavoidable.

2 Pipelining Registers

Recall the five stages: In the **IF** stage, we use the Program Counter to access our instruction as it is stored in IMEM. Then, we separate the distinct parts we need from the instruction bits in the **ID** stage and generate our immediate, the register values from the RegFile, and other control signals. Afterwards, using these values and signals, we complete the necessary ALU operations in the **EX** stage. Next, anything we do in regards with DMEM (not to be confused with RegFile or IMEM) is done in the **MEM** stage, before we hit the **WB** stage, where we write the computed value that we want back into the return register in the RegFile.

In order to pipeline, we separate the datapath into 5 discrete stages. These 5 stages, divided by registers, allow operation of different stages of the datapath in the same clock period. Different instructions can use different stages at a time. At each clock cycle, the necessary inputs into a particular stage are sampled at the rising clock edge (and available after the clk-to-q delay). After the stage operates on the inputs, the corresponding outputs are fed into pipeline registers for the next stage. Note, pipeline registers may also be required to pass information that may not be necessary for the next immediate stage, but some future stage.





2.1 Two diagrams are provided above. The topmost one is the standard single cycle datapath. The second is a modified version. Compare these two diagrams and explain the difference.

In the modified version, there is no wire that connects the output of the +4 block close to the PC register to the WB mux. Instead, there is an additional +4 block, which is located in the MEM stage. It takes as input the wire carrying the PC signal (extended from the wire that feeds into the 1 input of the ASel mux). The output is PC + 4, which feeds into the WB mux.

2.2 Using the modified single-cycle datapath as reference provided above, think about the information that needs to be passed along from stage to stage. Which pipeline registers are required at the end of each stage?

IF to ID:

- PC : The most adjacent stage in which the PC signal is used later on is the EX stage where PC is the input into the ASel mux.
- Inst : input into the RegFile, ImmGen, and control logic of the ID stage.

ID to EX:

- PC : input into the ASel mux
- RegReadData1 : is an input into the ASel mux.
- RegReadData2 : is an input into the BSel mux.
- Imm : is an input into the BSel mux.
- Inst : is required to compute the control logic for that particular instruction being executed in a particular stage. Therefore, the values generated by the control logic will be different in each stage depending on the input instruction.

What would happen if the Inst signal was not passed along? If each stage involves a different instruction, is it correct for all stages to have the same control logic?

EX to MEM:

- PC : input into the +4 block in the MEM stage.
- ALUOut : is an input into DMEM.
- RegReadData2 : is an input into DMEM,
- Inst : input into next stage's control logic.

MEM to WB:

- PC + 4: input into WBSel mux.
- ALUOut : input into WBSel mux.
- MEM : input into WBSel mux.
- Inst : input into next stage's control logic.

2.3 Looking at the way PC is passed through the datapath, there are two places where +4 is added to the PC, once in the **IF** and **MEM** stage. Why do we add +4 to the PC again in the memory stage?

We add +4 to the PC again in the memory stage so we don't need to pass both PC and PC+4 along the whole pipeline. This would use more registers, adding unnecessary hardware. We also can't just pass only PC+4 through the pipeline, as we need the original PC value in operands like `auipc`.

3 Performance Analysis

Register clk-to-q	30 ps	**Branch comp.**	75 ps	**DMEM write setup**	200 ps
Register setup	20 ps	**ALU**	200 ps	**RegFile read**	100 ps
Register hold	10 ps	**Imm. Gen.**	15 ps	**RegFile setup**	20 ps
Mux	25 ps	**Memory read**	250 ps		

Given above are sample delays and setup times for each of the datapath components and registers. In the questions below, use these in conjunction with the pipelined datapath on the last page to answer them.

3.1 What would be the fastest possible clock time for a single cycle datapath? Recall from last week's discussion that one instruction which exercises the critical path is `lw`.

The instruction that requires the longest clock period is lw. This sets the constraint on the minimum clock cycle for the entire datapath (even though other instructions require less time)

$$
\begin{aligned}
t_{\text{clk}} &\geq t_{\text{PC clk-to-q}} + t_{\text{IMEM read}} + max(t_{\text{RF read}}, t_{\text{Imm Gen}}) + t_{\text{A/BSelmux}} + t_{\text{ALU}} + t_{\text{DMEM read}} + t_{\text{WBSelmux}} + t_{\text{RF setup}} \\
&\geq 30 + 250 + 100 + 25 + 200 + 250 + 25 + 20 \\
&\geq 900 \text{ ps}
\end{aligned}
$$

3.2 What is the fastest possible clock time for a pipelined datapath?

$$
\begin{aligned}
\textbf{IF}: &\ t_{\text{PC clk-to-q}} + t_{\text{IMEM read}} + t_{\text{pipelineRegister setup}} = 30 + 250 + 20 = 300 \text{ ps} \\
\textbf{ID}: &\ t_{\text{pipelineRegister clk-to-q}} + max(t_{\text{RF read}}, t_{\text{Imm Gen}}) + t_{\text{pipelineRegister setup}} = 30 + 100 + 20 = 150 \text{ ps} \\
\textbf{EX}: &\ t_{\text{pipelineRegister clk-to-q}} + t_{\text{A/Bmux}} + t_{\text{ALU}} + t_{\text{pipelineRegister setup}} = 30 + 25 + 200 + 20 = 275 \text{ ps} \\
\textbf{MEM}: & \\
Read: &\ t_{\text{pipelineRegister clk-to-q}} + t_{\text{DMEM read}} + t_{\text{pipelineRegister setup}} = 30 + 250 + 20 = 300 \text{ ps} \\
Write: &\ t_{\text{pipelineRegister clk-to-q}} + t_{\text{DMEM write setup}} = 30 + 200 = 230 \text{ ps} \\
&\ max(DMEMread, DMEMwrite) = 300 \text{ ps} \\
\textbf{WB}: &\ t_{\text{pipelineRegister clk-to-q}} + t_{\text{mux}} + t_{\text{RF setup}} = 30 + 25 + 20 = 75 \text{ ps}
\end{aligned}
$$

$$
t_{\text{clk}} \geq \max(\textbf{IF}, \textbf{ID}, \textbf{EX}, \textbf{MEM}, \textbf{WB}) = 300 \text{ ps}
$$

3.3 What is the speedup from the single cycle datapath to the pipelined datapath? Why is the speedup less than 5×?

$\frac{900 \text{ ps}}{300 \text{ ps}}$, or a 3 times speedup. The speedup is less than 5 because of (1) the necessity of adding pipeline registers, which have clk-to-q and setup times, and (2) the need to set the clock to the maximum of the five stages, which take different amounts of time.

Note: Due to hazards, which require additional logic to resolve, the actual speedup would likely be even less than 3 times.

4 Hazards

One of the costs of pipelining is that it introduces pipeline hazards. Hazards, generally, are issues with something in the CPU's instruction pipeline that could cause the next instruction to execute incorrectly.

The 5-stage pipelined CPU introduces three types: structural hazards (hardware not sufficient), data hazards (using wrong values in computation), and control hazards (executing the wrong instruction).

Structural Hazards

Structural hazards occur when more than one instruction needs to use the same datapath resource at the same time. In the standard 5-stage pipeline, **there aren't structural hazards**, unless there are active changes to the pipeline. The structural hazards that could exist are prevented by RV32I's hardware requirements.

There are two main causes of structural hazards:

- **Register File:** The register file is accessed both during ID, when it is read to decode the instruction, and the corresponding register values; and during WB, when it is written to in the `rd` register. If the RegFile only had one port, then it wouldn't work since we have one instruction being decoded and another writing back.
    - We resolve this by having separate read and write ports. However, this only works if the read/written registers are different.
- **Main Memory:** Main memory is accessed for both instructions and data. If memory could only support one read/write at a time, then instruction A going through IF and attempting to fetch an instruction from memory cannot happen at the same time as instruction B attempting to read (or write) to data portions of memory.
    - Having a separate instruction memory (abbreviated IMEM) and data memory (abbreviated DMEM) solves this hazard.

Something to remember about structural hazards is that they can always be resolved by adding more hardware.

Data Hazards

Data hazards are caused by data dependencies between instructions. In CS 61C, where we always assume that instructions go through the processor in order, we see data hazards when an instruction **reads** a register before a previous instruction has finished **writing** to that register.

There are three types of data hazards:

- **EX-ID:** this hazard exists because the output from the execute stage is not written back to the RegFile until the writeback stage, yet can be requested by the subsequent instruction in the decode stage.
- **MEM-ID:** this hazard exists because the output from the memory access stage is not written back to the RegFile until the writeback stage, but can be requested from the decode stage, just as in EX-ID.
- **WB-ID** To account for reads and writes to the same register, processors usually write to the register during the first half of the clock cycle, and read from it during in the second half. This is an implementation of the idea of **double pumping**, which is when data is transferred along data buses at double the rate, by utilising both the rising and falling clock edges in a clock cycle.

Solving Data Hazards

For all questions, assume **no branch prediction and no double-pumping (i.e. we do not write-then-read in one cycle for RegFile)**.

Forwarding

Most data hazards can be resolved by forwarding, which is when the result of the EX or MEM stage is sent to the EX stage for a following instruction to use.

Side note: how is forwarding (EX to EX or MEM to EX) implemented in hardware? We add 2 wires: one from the beginning of the MEM stage for the output of the ALU and one from the beginning of the WB stage. Both of these wires will connect to the A/B muxes in the EX stage.

4.1 Look for data hazards in the code below, and figure out how forwarding could be used to solve them.

Instruction	C1	C2	C3	C4	C5	C6	C7
1. `addi t0, a0, -1`	IF	ID	EX	MEM	WB		
2. `and s2, t0, a0`		IF	ID	EX	MEM	WB	
3. `sltiu a0, t0, 5`			IF	ID	EX	MEM	WB

There are two data hazards, between instructions 1 and 2, and between instructions 1 and 3. The first could be resolved by forwarding the ALU output in the MEM stage in C3 to the beginning of the EX stage in C4, and the second could be resolved by forwarding the ALU output in the WB stage in C4 to the beginning of the EX stage in C5.

4.2 Imagine you are a hardware designer working on a CPU's forwarding control logic. How many instructions after the addi instruction could be affected by data hazards created by this addi instruction?

Three instructions. For example, with the addi instruction, any instruction that uses t0 that has its ID stage in C3, C4, or C5 will not have the result of addi's writeback in C5. If, however, we are allowed to assume double-pumping (write-then-read to registers), then it would only affect two instructions since the ID stage of instruction 4 would be allowed to line up with the WB stage of intruction 1.

Stalls

4.3 Look for data hazards in the code below. One of them cannot be solved with forwarding—why? What can we do to solve this hazard?

Instruction	C1	C2	C3	C4	C5	C6	C7	C8
1. `addi s0, s0, 1`	IF	ID	EX	MEM	WB			
2. `addi t0, t0, 4`		IF	ID	EX	MEM	WB		
3. `lw t1, 0(t0)`			IF	ID	EX	MEM	WB	
4. `add t2, t1, x0`				IF	ID	EX	MEM	WB

There are two data hazards in the code. The first hazard is between instructions 2 and 3, from t0, and the second is between instructions 3 and 4, from t1. The hazard between instructions 2 and 3 can be resolved with forwarding, but the hazard between instructions 3 and 4 cannot be resolved with forwarding. This is because even with forwarding, instruction 4 needs the result of instruction 3 at the beginning of C6, and it won't be ready until the end of C6.

We can fix this by stalling: insert a nop (no-operation) between instructions 3 and 4.

4.4 Say you are the compiler and can re-order instructions to minimize data hazards while guaranteeing the same output. How can you fix the code above?

Reorder the instructions 2-3-1-4, because instruction 1 has no dependencies.

Detecting Data Hazards

Say we have the $rs1$, $rs2$, $RegWEn$, and rd signals for two instructions (instruction n and instruction $n+1$) and we wish to determine if a data hazard exists across the instructions. We can simply check to see if the rd for instruction n matches either $rs1$ or $rs2$ of instruction $n+1$, indicating that such a hazard exists (why does this make sense?).

We could then use our hazard detection to determine which forwarding paths/number of stalls (if any) are necessary to take to ensure proper instruction execution. In pseudo-code, part of this could look something like the following:

```
if (rs1(n + 1) == rd(n) && RegWen(n) == 1) {
    set ASel for (n + 1) to forward ALU output from n
}
if (rs2(n + 1) == rd(n) && RegWen(n) == 1) {
    set BSel for (n + 1) to forward ALU output from n
}
```

Control Hazards

Control hazards are caused by **jump and branch instructions**, because for all jumps and some branches, the next PC is not PC + 4, but the result of the ALU available after the EX stage. We could stall the pipeline for control hazards, but this decreases performance.

4.5 Besides stalling, what can we do to resolve control hazards?

We can try to predict which way branches will go, and if this prediction is incorrect, flush the pipeline and continue with the correct instruction. (The most naive prediction method is to simply predict that branches are always not taken, which is effectively the same as not having any branch prediction at all.)

Extra for Experience

4.6 Given the RISC-V code above and a pipelined CPU with no forwarding, how many hazards would there be? What types are each hazard? Consider all possible hazards between all instructions.

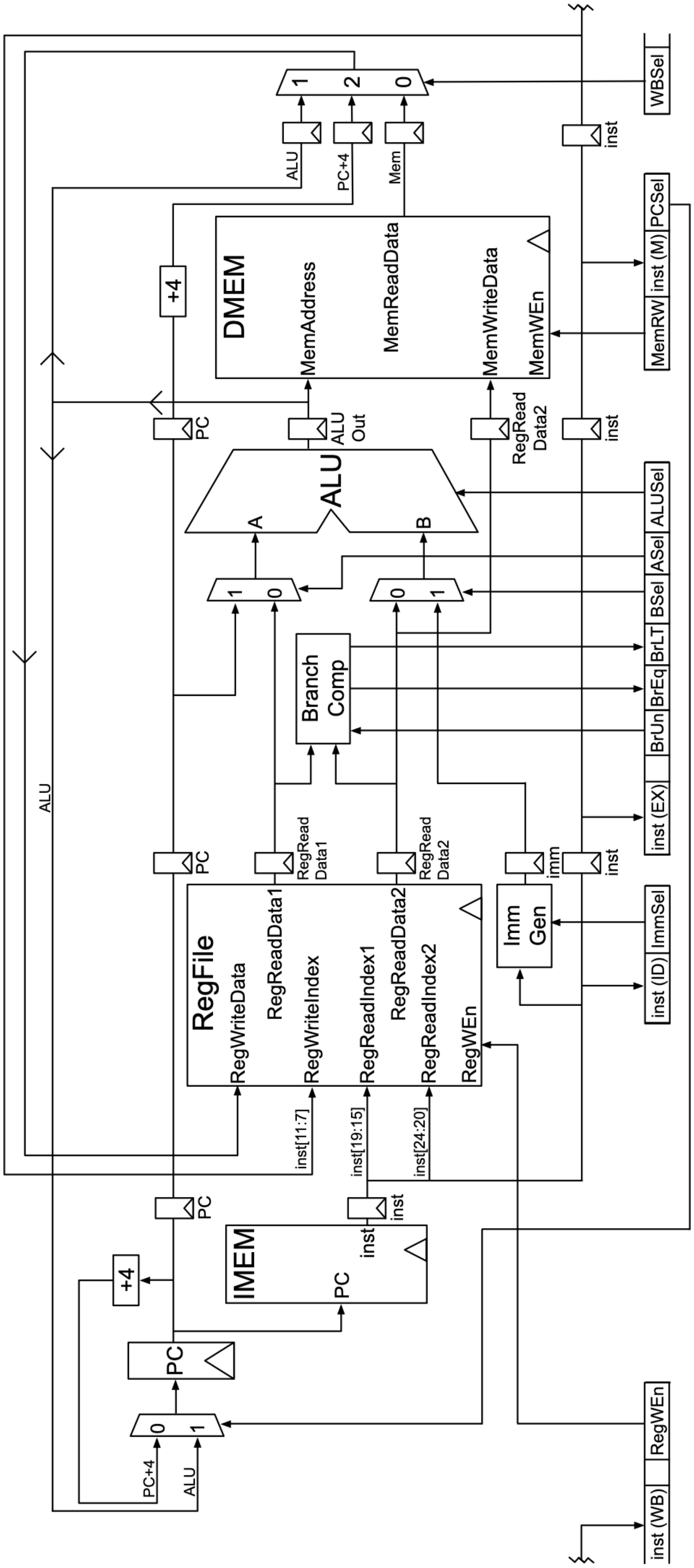
How many stalls would there need to be in order to fix the data hazard(s), if the RegFile supports double-pumping (i.e. write-then-read)? What about the control hazard(s), if we use branch prediction?

Instruction	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11
1. `loop: sub t1, s0, s1`	IF	ID	EX	MEM	WB						
2. `or s0, t0, t1`		IF	ID	EX	MEM	WB					
3. `sw s1, 100(s0)`			IF	ID	EX	MEM	WB				
4. `bgeu s0, s2, loop`				IF	ID	EX	MEM	WB			
5. `add t2, x0, x0`					IF	ID	EX	MEM	WB		
6. `add s2, t1, x0`						IF	ID	EX	MEM	WB	
7. `add s3, t1, x0`							IF	ID	EX	MEM	WB

There are four hazards: between instructions 1 and 2 (data hazard from t1), instructions 2 and 3 (data hazard from s0), instructions 2 and 4 (from s0), and instructions 4 and 5 (a control hazard).

Assuming that we can read and write to the RegFile on the same cycle, two stalls are needed between instructions 1 and 2 (for WB and ID to occur in the same clock period), and two stalls are needed between instructions 2 and 3 (for WB and ID occur in the same period). For the control hazard, if we predicted correctly, then no stalls are needed, but if we predicted incorrectly, then we need to flush the stages (for 3 instructions after instruction 4) occurring after the EX stage of instruction 4 (the EX stage is when the results of the branch comparator are determined). The updated value of PC from instruction 4's ALU output is ready to sampled by the PC register at the rising clock edge of C12. We don't need to stall for the hazard between 2 and 4 because stalling for instruction 3 already handles that. The solution below displays the results when the branch is taken.

Instruction	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	C15	C16
1. `loop: sub t1, s0, s1`	IF	ID	EX	MEM	WB											
2. `or -> nop`		IF	X	X	X	X										
2. `or -> nop`			IF	X	X	X	X									
2. `or s0, t0, t1`				IF	ID	EX	MEM	WB								
3. `sw -> nop`					IF	X	X	X	X							
3. `sw -> nop`						IF	X	X	X	X						
3. `sw s1, 100(s0)`							IF	ID	EX	MEM	WB					
4. `bgeu s0, s2, loop`								IF	ID	EX	MEM	WB				
5. `add t2, x0, x0`									IF	ID	X	X	X			
6. `add s2, t1, x0`										IF	X	X	X	X		
7. `add s3, t1, x0`											X	X	X	X	X	
1. `loop: sub t1, s0, s1`												IF	ID	EX	MEM	WB


PC
+4
PC+4
ALU
0
1
IMEM
PC
inst
PC
inst
inst[11:7]
inst[19:15]
inst[24:20]
RegFile
RegWriteData
RegReadData1
RegWriteIndex
RegReadIndex1
RegReadData2
RegReadIndex2
RegWEn
Imm
Gen
PC
RegRead
Data1
RegRead
Data2
imm
inst
Branch
Comp
1
0
0
1
A
B
ALU
PC
ALU
Out
RegRead
Data2
inst
DMEM
MemAddress
MemReadData
MemWriteData
MemWEn
+4
ALU
PC+4
Mem
inst
1
2
0
ALU
inst (WB)
RegWEn
inst (ID)
ImmSel
inst (EX)
BrUn
BrEq
BrLT
BSel
ASel
ALUSel
MemRW
inst (M)
PCSel
WBSel