Identify all of the potential data hazards in the following code snippet *assuming the branch is not taken*:

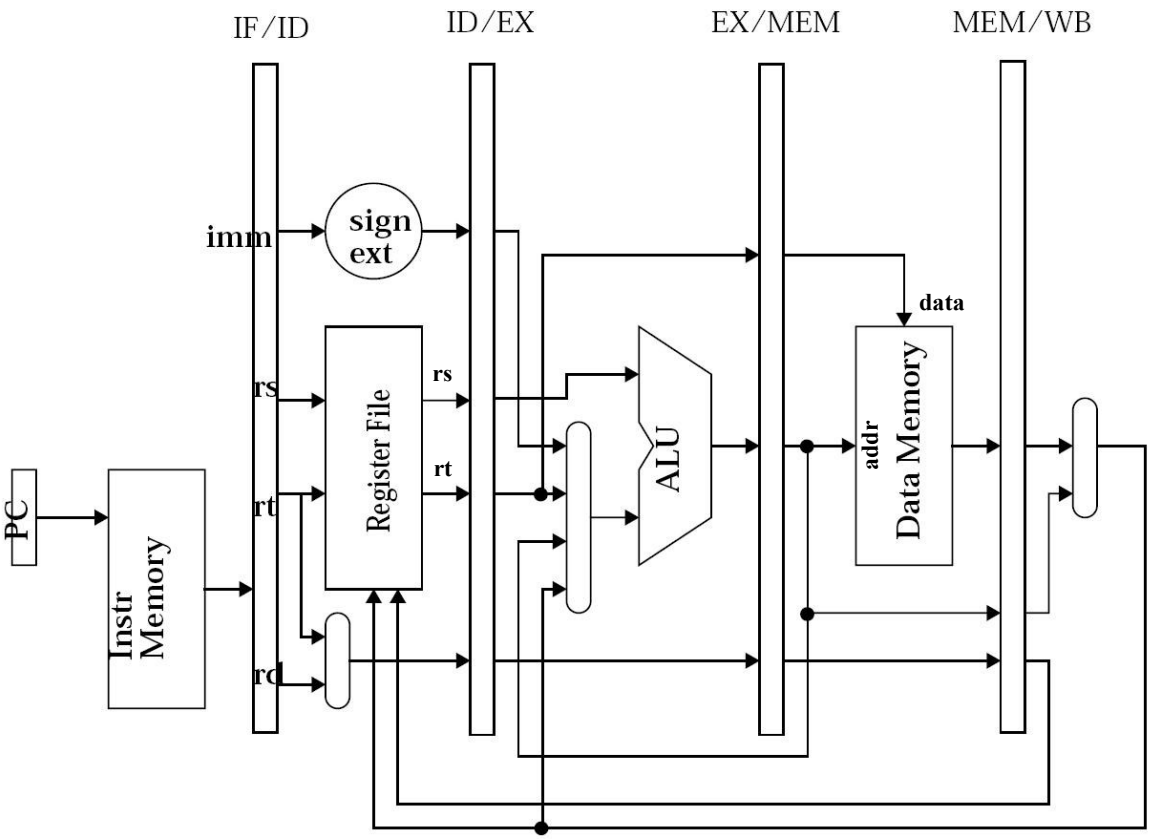
```
1.  LW      $t1,    0($s0)
2.  SW      $t1, 4096($s2)
3.  ADD     $t3, $t2, $t1
4.  BNE     $t1, $t3, .Target
5.  LW      $t4,   12($t3)
6.  ORI     $t5, $t3, 12
7.  SW      $t3,    0($t4)
8.  AND     $t6, $t7, $t5
.Target:
```

Solution:

- 1 → 2 RAW $t1
- 1 → 3 RAW $t1
- 1 → 4 RAW $t1
- 3 → 4 RAW $t3
- 3 → 5 RAW $t3
- 3 → 6 RAW $t3
- 3 → 7 RAW $t7
- 5 → 7 RAW $t4
- 6 → 8 RAW $t5

Part B [14 points]

High performance datapaths use bypass paths (also known as data forwarding logic) to reduce pipeline stalls. However, bypass paths are relatively expensive, especially in some wire constrained technologies. To reduce the cost (and potential cycle time impact), some architects have explored omitting some of the possible bypass paths. Consider the datapath illustrated below (note that the PC update logic and all control logic is intentionally omitted). This pipelined datapath is similar to the ones in the book and discussed in class, *but has several differences including limited bypass paths*. **BE SURE TO STUDY THE DATAPATH CAREFULLY**! For simplicity, I have labeled which input to the Mem stage is for the data and which is for the address. Moreover, in each stage, you should assume the upper wire is for 'rs', and the lower one is for 'rt' (as highlighted in the Decode stage below). Assume that the register file supports bypassing, so that if register `$i` is read and written in the same cycle, then the read returns the new value. Assume that the control logic bypasses the data as soon as possible using the given forwarding data paths, and stalls in Decode otherwise. Finally, assume that **branches are resolved in Execute**. **You may NOT add additional data paths**.



In this problem, you will look at how the program snippet from Problem 6 Part A performs on this pipeline **assuming the branch is not taken and using a predict-not-taken strategy**. Recall that R-format instructions have the form:

```
opcode rd, rs, rt
```

and I-format instructions have the form:

```
opcode rt, imm(rs)
```

or:

```
opcode rt, rs, imm // includes branches
```

Use the table below to show how the given instruction sequence flows through the pipeline and where stalls are necessary to resolve hazards (*as discussed in class, use "*" to indicate a pipeline stall at the clock cycle of the corresponding instruction; for example F* signifies a stall in the Fetch stage*).

Solution:

Instr / Cycle	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
LW $t1, 0($s0)	**F**	**D**	**X**	**M**	**W**															
SW $t1, 4096($s2)		F	D*	D*	D	X	M	W												
ADD $t3, $t2, $t1			F*	F*	F	D	X	M	W											
BNE $t1, $t3, .Target						F	D*	D*	D	X	M	W								
LW $t4, 12($t3)							F*	F*	F	D	X	M	W							
ORI $t5, $t3, 12										F	D	X	M	W						
SW $t3, 0($t4)											F	D*	D	X	M	W				
AND $t6, $t7, $t5												F*	F	D	X	M	W			

Part C [4 points]

For each cycle where a stall occurs, *explain why below*.

Note that 'rt' has EX → EX forwarding and MEM → EX forwarding, while 'rs' no forwarding. Moreover, note that 'rt' is not forwarded into the data memory – because the wire connecting the data value in Data Memory is taken before the forwarding mux. This only affects the data for stores.

- **Cycle 2:** The SW is stalled in decode because it has a data hazard on the LW. We stall specifically in Decode here because we don't have M → M forwarding for the data the SW is using from the LW. Thus, since we don't have M → M forwarding, we must wait for RF bypassing for $t1 for the Rt for our store. As a result, the ADD has a structural hazard and is stalled in Fetch behind the SW.
- **Cycle 3:** same as previous cycle.
- **Cycle 6:** The BNE is stalled in Decode waiting for its Rs ($t3). It must stall because there is no X → X forwarding for Rs, thus it must wait for RF bypassing – thus this is a data hazard. In the same cycle, the LW after the BNE is also stalled in Fetch due to a structural hazard with the BNE.
- **Cycle 7:** same as previous cycle (except now no M → X forwarding for Rs).
- **Cycle 11**: The SW is stalled in Decode waiting for its Rs ($t4). It must stall because there is no M → X forwarding for Rs in this pipeline, so it must wait for RF bypassing – thus this is a data hazard. In the same cycle, the AND after the SW is also stalled in Fetch due to a structural hazard with the SW.