For the following code, explain what is happening in each stage of the pipelined processor during cycle 5.

ADD X0, X31, #102
LDUR X1, [X0, #10]
STUR X0, [X0, #10]
CBZ X1, LOOP
EOR X6, X4, X1

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>IF</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDUR</td>
<td>IF</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STUR</td>
<td></td>
<td>IF</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CBZ</td>
<td></td>
<td></td>
<td>IF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EOR</td>
<td></td>
<td></td>
<td></td>
<td>IF</td>
<td></td>
</tr>
</tbody>
</table>

WB: Result of ADD, #102 being written to register X0.
MEM: Data being read in for the LDUR, from address 112
EX: Address of STUR, 112, being computed in ALU
REG/DEC: Decoding CBZ, forwarding X1 from MEM, checking if X1 = 20, finishing the branch via accelerated branches
IF: Bringing in the EOR instruction.
Our processor has a load delay slot, which means code cannot use the Rd register of a load in the instruction right after a load. What happens if we ignore this restriction? Specifically, assume the following code is given to the CPU. Assume MEM[24] = -12. What value will end up in register X5? Think carefully – this one is tricky!

ADDI X3, X31, #16  \( \# x3 = 16 \)
LDUR X3, [X3, #8]
ADD X5, X3, X3

LDUR in EX computes the address \( 16 + 8 = 24 \)
Output of ALU forwarded to REG/DEC of ADD.
ADD computes \( 24 + 24 = 48 \).
When ADD completes, \( X5 = 48 \).
The following code contains a “read after write” data hazard that is resolved by forwarding:

ADD X2, X3, X4
ADD X5, X2, X6

Consider the following code where a memory read occurs after a memory write:

STUR X7, [X2, #100]
LDUR X8, [X2, #100]

Does the code work correctly? Why/why not? Will the forwarding unit need to be altered to handle this code?

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>STUR</td>
<td>IF</td>
<td>DEC</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>LDUR</td>
<td>IF</td>
<td>DEC</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>

In cycle #4, store puts value into Data Memory.
In cycle #5, load reads Data Memory, and
store result is already there.

Works fine, no forwarding needed.
Consider a processor with the following delays in each of the individual units:

<table>
<thead>
<tr>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>200ps</td>
<td>150ps</td>
<td>120ps</td>
<td>190ps</td>
<td>140ps</td>
</tr>
</tbody>
</table>

a.) What is the clock cycle time in a pipelined and non-pipelined processor?

b.) If we can split one stage of the pipelined datapath into two new stages, each with half the latency of the original stage, which stage would you split, and what is the new clock cycle time of the processor?

\[\text{a.) Pipelined: Cycle} = \max(200, 150, 120, 190, 140) = 200\text{ps}\]

\[\text{Non-pipelined: Cycle} = \sum(200, 150, 120, 190, 140) = 800\text{ps}\]

\[\text{b.) Split IF: Cycle} = \max(100, 100, 150, 120, 190, 140) = 190\text{ps}\]
For the following code, explain what the register file and forwarding unit are doing during the fifth cycle of execution. If any comparisons are being made, mention them. Remember, use the forwarding unit from class, not the book.

```
LOOP:  
LDUR   X1, [X6, #40]  IF  DEC  EX  MEM  BF  
ADD    X5, X5, X8     IF  DEC  EX  MEM  
ADD    X6, X6, X8     IF  DEC  EX  MEM  
STUR   X1, [X5, #20]  IF  DEC  EX  MEM  
CBZ    X1, LOOP       IF  DEC  EX  MEM  
```

Forwarding happens in DEC, so we feed into the STUR. The STUR is fetching operands X1 + X5.

We look at instruction in EX (ADD X6). X6 != X1, X6 != X5, so no forwarding from EX.

We look at instruction in MEM (ADD X5), which is writing to X5. X5 = = X5, the STUR’s address operand.

So, X1 is read from register, X5 forwarded from MEM.

Note: X1 obtained from the LDUR via the reg file redesign (invert the clock), so technically isn’t forwarding.