1. **Multicycle Processor Design**  

In our discussion of exceptions in the multicycle processor implementation, we showed how the Exception Program Counter (EPC) is written, but not how it is read. This is typically done using an instruction which moves the EPC to one of the 32 general-purpose registers. An exception handler can then use the EPC value to identify the instruction where the exception occurred, and (if the exception can be handled), where the program should resume after exception handling is completed.

To transfer the EPC to a register, we propose a new MIPS instruction called \texttt{mfepc} (move from EPC), which stores the EPC into the register given by the \texttt{rt} field. The assembly language format, register transfer, and binary instruction format of the \texttt{mfepc} instruction are shown below:

\[
\text{mfepc rt} \quad \text{Reg[rt]} \leftarrow \text{EPC}
\]

(a) Modify the attached multicycle datapath schematic and controller state diagram to implement the \texttt{mfepc} instruction as described above.

(b) If an exception handler can successfully deal with an exception (e.g., by emulating an unimplemented instruction), it can continue execution of the interrupted program starting with the instruction \textbf{after} the instruction that caused the exception. Assuming that the \texttt{mfepc} is used to move the EPC to register $t0$, write a sequence of MIPS assembly language instructions that will do this.
2. Pipelined Processor Design 25 Points

The pipelined processor implementation that was described in Chapter 6 does implement the \textit{j} (jump) instruction.

(a) Modify the pipelined processor datapath and control to implement the “j” (jump) instruction. Show any changes to the datapath on the diagram on the next page, and show all control signals (including any that you add) in the table below. You may find it helpful to review the control table in Fig. 6.28 on p. 469 of the text. HINT: most of the instruction can be implemented in the “ID” stage.

(b) Will the jump instruction in your design be susceptible to any hazards? Why or why not?

<table>
<thead>
<tr>
<th>Instr.</th>
<th>EX Stage Control Lines</th>
<th>MEM Stage Control Lines</th>
<th>WB Stage Control Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Reg Dst</td>
<td>ALU Op1</td>
<td>ALU Op0</td>
</tr>
<tr>
<td>jmp</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
3. Pipelined Design – Forwarding, Data Hazards, and Stalls  25 Points

Consider the following sequence of MIPS instructions and assume that we are using the pipelined design with forwarding shown in Fig. 6.40 of the book (p. 484).

\[
\begin{align*}
\text{add} & \quad \$1, \ $2, \ $3 \\
\text{sw} & \quad \$3, \ 100 (\$1) \\
\text{lw} & \quad \$4, \ 200 (\$3) \\
\text{sub} & \quad \$5, \ $5, \ $4
\end{align*}
\]

(a) Circle any data dependencies which exist in these instructions for the given pipeline in the code listed above.

(b) Fill in the multi-cycle pipeline diagram below to show how the given instructions flow through the pipeline. You should label each instruction, shade stages to show which parts of each stage are active during each cycle, show any forwarding connections that are needed, and clearly mark stalled instructions.

(c) Assuming that the pipeline is initially empty, how many clock cycles does it take for this sequence of instructions to execute?
4. Short Answers 20 Points

Please answer the following questions concisely and briefly.

(a) What are the advantages of a fixed-length instruction format compared to a variable-length instruction format?

(b) What is the purpose of denormalized numbers in the IEEE floating-point representation?

(c) Why is microcode still used in the Pentium 4 and other x86 processor designs?

(d) What is the key advantage of performing out-of-order execution in a processor implementation?

(e) What happens on a cache miss?