Problem 1. ★

A RISC-V processor is running the below assembly code in user-mode with exceptions enabled. Each program has a virtual memory space starting at **0x0000** and ending at **0x1000** (exclusive). In sections without code or data, this memory is filled with zeros. Divide by zeros trigger exceptions. For each program, determine whether it triggers an exception. If it does, write down the **Cause**, **EPC**, and in which Pipeline Stage (F, D, E, M, W) the exception happens.

Side-note on the `la` instruction (also useful for the OS lab):

- `la rd, <label>`: Loads the address of the specified label into `rd`. If the label is the name of a variable, it gets the address of the variable, not the value of it. This is a pseudoinstruction that expands into a sequence of instructions similar to `li`.

\((A)\)

\[\begin{align*}
&.={0x100} \\
&addi \text{ }a0, \text{ }zero, \text{ }0x400 \\
&lw \text{ }a1, \text{ }4(a0) \\
&div \text{ }a2, \text{ }a0, \text{ }a1 \\
&addi \text{ }a2, \text{ }0xff \\
&div \text{ }a0, \text{ }a0, \text{ }a2 \\
&\text{ret} \\
&.={0x400} \\
&\text{secret: } .word 0xc0febabe
\end{align*}\]

\((B)\)

\[\begin{align*}
&.={0x200} \\
&\text{addi } \text{ }a0, \text{ }zero, \text{ }0x900 \\
&\text{li } \text{ }a2, \text{ }0x0 \\
&\text{li } \text{ }a3, \text{ }0x1000 \\
&\text{loop:} \\
&\quad \text{lw } \text{ }a1, \text{ }0(a0) \\
&\quad \text{add } \text{ }a2, \text{ }a2, \text{ }a1 \\
&\quad \text{addi } \text{ }a0, \text{ }a0, \text{ }4 \\
&\quad \text{ble } \text{ }a0, \text{ }a3, \text{ }\text{loop} \\
&\quad \text{mv } \text{ }a0, \text{ }a2 \\
&\text{ret} \\
&.={0x900} \\
&A: \text{.word } 0xbeef, \text{ }0xf00d, \text{ }0xcafe
\end{align*}\]

**Cause** = __________

**EPC** = __________

**Stage** = __________
(C)

```
.. = 0x300
li a0, 0x7
li a1, 0x14
call func
ret

.. = 0x1000
func:
    add a0, a0, a1
    ret
```

Cause =
EPC =
Stage =

(D)

```
.. = 0x100
la a0, A
li a2, 0x0
li a3, 0x140
loop:
    addi a0, a0, 4
    ble a0, a3, loop
mv a0, a2
A: .word 0x34, 0x56
```

Cause =
EPC =
Stage =
Problem 2. ★

A user-space program and kernel-space handler are given below. The CPU’s mtvec register is set to 0x100.

// program 1 space (virtual)  // kernel space (physical)
.= 0x500
li a0, 0x800
li a1, 0x0
li a2, 0x0
li a3, 0x904
li a4, 0x0
loop:  
    add a2, a2, a1
    addi a0, a0, 4
    lw a1, 0(a0)
    b ble a0, a3, loop
lui a4, 0x1000
sw a2, 0(a4)
sw a1, 4(a4)
xor a0, a1, a2
...

handler:
addi t0, zero, zero
sw a0, 0(t0)
sw a1, 4(t0)
sw a2, 8(t0)
sw a3, 12(t0)
...

Assume the loop in program 1 is running in steady state on a RISCV processor with exceptions enabled. Branch decisions are resolved in the EXE stage.

(A) For this part, assume that exceptions are handled lazily, i.e. they are handled before the commit point (writeback stage) and not immediately. This diagram from Lecture 19 summarizes lazy exception handling.
In this part, \texttt{lui} triggers a memory fault in the fetch stage. With lazy exception handling, what does the pipeline diagram look like during the last loop iteration (when \texttt{a0} = 0x904)?

<table>
<thead>
<tr>
<th></th>
<th>500</th>
<th>501</th>
<th>502</th>
<th>503</th>
<th>504</th>
<th>505</th>
<th>506</th>
<th>507</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>lw</td>
<td>ble</td>
<td>lui</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEC</td>
<td></td>
<td>lw</td>
<td>ble</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXE</td>
<td></td>
<td></td>
<td>lw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(B) For this part, assume that the \texttt{lw} instruction also causes an exception on the last loop iteration (when \texttt{a0} = 0x904). Exceptions are still handled lazily. What does the pipeline diagram look like now?

<table>
<thead>
<tr>
<th></th>
<th>500</th>
<th>501</th>
<th>502</th>
<th>503</th>
<th>504</th>
<th>505</th>
<th>506</th>
<th>507</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>lw</td>
<td>ble</td>
<td>lui</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEC</td>
<td></td>
<td>lw</td>
<td>ble</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXE</td>
<td></td>
<td></td>
<td>lw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(C) For this part, assume that the 5-stage RISCV processor handles exceptions immediately in the pipeline stage they occur in. Again, both \texttt{lui} and \texttt{lw} trigger an exception. What does the pipeline diagram look like now?

<table>
<thead>
<tr>
<th></th>
<th>500</th>
<th>501</th>
<th>502</th>
<th>503</th>
<th>504</th>
<th>505</th>
<th>506</th>
<th>507</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>lw</td>
<td>ble</td>
<td>lui</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEC</td>
<td></td>
<td>lw</td>
<td>ble</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXE</td>
<td></td>
<td></td>
<td>lw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Problem 3. ★

In this exercise, we will write a RISCV assembly program that runs inside a Linux-based OS. The table below summarizes important Linux syscalls that this program will need to invoke.

<table>
<thead>
<tr>
<th>Syscall Name</th>
<th>Syscall #</th>
<th>Arg0</th>
<th>Arg1</th>
<th>Arg2</th>
<th>Return</th>
</tr>
</thead>
<tbody>
<tr>
<td>read</td>
<td>63</td>
<td>int fd</td>
<td>char* buf</td>
<td>int count</td>
<td>int</td>
</tr>
<tr>
<td>write</td>
<td>64</td>
<td>int fd</td>
<td>char* buf</td>
<td>int count</td>
<td>int</td>
</tr>
<tr>
<td>exit</td>
<td>93</td>
<td>int error</td>
<td></td>
<td></td>
<td>int</td>
</tr>
<tr>
<td>getpid</td>
<td>172</td>
<td></td>
<td></td>
<td></td>
<td>int (PID)</td>
</tr>
</tbody>
</table>

The behavior of the program is as follows:

1. Reads 5 characters from file descriptor 0 (fd=0), also known as standard input.
2. Checks if the running process ID is equal to zero.
3. If it is, write the 5 characters to file descriptor 1 (fd=1), also known as standard output.
   a. Then exit with code 0.
4. Otherwise, exit with code 255.

We have provided skeleton code which you will need to complete. The entry point of the program is main.

```assembly
.main:
    // Read 5 characters from standard input
    li a1, 0x100
    ______________
    ______________
    ecall
    // Get PID
    ______________
    ______________
    // Check if PID is zero
    ______________
    is_not_zero:
    // Exit with code 255
    li a0, 0xff
    ecall
    is_zero:
    // Write buffer to standard output
    li a7, 64
    ______________
    li a1, 0x100
    ______________
```

We have provided skeleton code which you will need to complete. The entry point of the program is main.
// Exit with code 0

___________________
___________________
___________________