Problem 1. ★

A RISCV 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.

Side-note on the li pseudo-instruction:

- li rd, <constant> : Loads a constant into rd. If the constant is 12-bits or less, li translates to a single instruction. If the constant is greater than 12-bits, li translates into two instructions.

(A)

```
.= 0x100
addi a0, zero, 0x400
lw a1, 4(a0)
div a2, a0, a1
addi a2, 0xff
div a0, a0, a2
ret
```

```
secret: .word 0xcabfebabe
```

(B)

```
.= 0x200
addi a0, zero, 0x900
li a2, 0x0
li a3, 0x1000
loop:
    lw a1, 0(a0)
    add a2, a2, a1
    addi a0, a0, 4
    ble a0, a3, loop
mv a0, a2
ret
```

```
A: .word 0x0beef, 0xf00d, 0xcafe
```

Causes: 

(A) Cause = Divide by Zero

(B) Cause = Memory Fault

EPCs:

(A) EPC = 0x108

(B) EPC = 0x20C

Stages:

(A) Stage = EXE

(B) Stage = MEM

---

6.004 Worksheet L19 – Exceptions

6.191 Worksheet - 1 of 6 - R19 – Exceptions
Explanation:

(A) The div instruction causes a divide by zero error because a1=0. This is because lw from 0x400 returns zero (given in the problem statement). Arithmetic exceptions trigger in the execute stage.

(B) This program loops from 0x900 to 0x1000 (inclusive) and does lw for each address it loops over. Since 0x1000 is outside of the mapped memory space, an access to 0x1000 will cause a memory fault.

(C) This program tries to call a function whose location is at 0x1000, so upon fetching an instruction from 0x1000, a memory fault gets triggered.

(D) This program is looping again like (B) but after the mv instruction, there is no ret instruction, so the program will continue executing at PC+4. Hence, the program will try to interpret word 0x34 as an instruction and throw an illegal instruction exception.
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)
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)
    bge 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 \textbf{the last loop iteration} (when \(a0 = 0x904\))? 

<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><strong>IF</strong></td>
<td>lw</td>
<td>ble</td>
<td>lui</td>
<td>sw</td>
<td>sw</td>
<td>xor</td>
<td>addi</td>
<td>sw</td>
</tr>
<tr>
<td><strong>DEC</strong></td>
<td>lw</td>
<td>ble</td>
<td>lui</td>
<td>sw</td>
<td>sw</td>
<td>nop</td>
<td>addi</td>
<td></td>
</tr>
<tr>
<td><strong>EXE</strong></td>
<td>lw</td>
<td>ble</td>
<td>lui</td>
<td>sw</td>
<td>nop</td>
<td>nop</td>
<td>addi</td>
<td></td>
</tr>
<tr>
<td><strong>MEM</strong></td>
<td>lw</td>
<td>ble</td>
<td>lui</td>
<td>sw</td>
<td>nop</td>
<td>nop</td>
<td>addi</td>
<td></td>
</tr>
<tr>
<td><strong>WB</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Explanation:** \texttt{lui} triggers a fault in the IF stage but this fault does not get handled until the commit point (the transition from MEM->WB). This occurs in cycle 505, so in cycle 506, the pipeline gets annulled and the first instruction from the handler gets fetched.

(B) For this part, assume that the \texttt{lw} instruction also causes an exception on \textbf{the last loop iteration} (when \(a0 = 0x904\)). 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><strong>IF</strong></td>
<td>lw</td>
<td>ble</td>
<td>lui</td>
<td>sw</td>
<td>addi</td>
<td>sw</td>
<td>sw</td>
<td>sw</td>
</tr>
<tr>
<td><strong>DEC</strong></td>
<td>lw</td>
<td>ble</td>
<td>lui</td>
<td>nop</td>
<td>addi</td>
<td>sw</td>
<td>sw</td>
<td></td>
</tr>
<tr>
<td><strong>EXE</strong></td>
<td>lw</td>
<td>ble</td>
<td>nop</td>
<td>nop</td>
<td>addi</td>
<td>sw</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>MEM</strong></td>
<td>lw</td>
<td>nop</td>
<td>nop</td>
<td>nop</td>
<td>addi</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>WB</strong></td>
<td></td>
<td>nop</td>
<td>nop</td>
<td>nop</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Explanation:** \texttt{lw} triggers an exception in the Mem stage which gets handled right away in cycle 503, so in cycle 504, the exception handler gets loaded. The \texttt{lui} never reaches the commit point, so that exception never needs to be handled.
(C) For this part, assume that the 5-stage RISCV processor handles exceptions immediately in the pipeline stage they occur in. Again, both lui and lw trigger an exception. What does the pipeline diagram look like now?

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Instruction 1</th>
<th>Instruction 2</th>
<th>Instruction 3</th>
<th>Instruction 4</th>
<th>Instruction 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>lw</td>
<td>ble</td>
<td>lui</td>
<td>addi</td>
<td>sw</td>
</tr>
<tr>
<td>501</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sw</td>
</tr>
<tr>
<td>502</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sw</td>
</tr>
<tr>
<td>503</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>504</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>505</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>506</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>507</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Explanation:** The lui triggers an exception in the IF stage, so the handler gets loaded in cycle 503. At the same time, the lw instruction in cycle 503 triggers an exception, so the handler gets loaded again in 504.

**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>int error</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
.#= 0x100
buffer: .word 0x0, 0x0, 0x0, 0x0, 0x0

main:
    // Read 5 characters from standard input
    li a1, 0x100
    _______________ li a7, 63       // read syscall #
```
li a0, 0x00 // read syscall arg 0
li a2, 0x05 // read syscall arg 2
ecall
// Get PID
li a7, 172 // getpid syscall #
ecall // do syscall
// Check if PID is zero
beqz a0, is_zero // branch if pid==0

is_not_zero:
// Exit with code 255
li a0, 0xff
ecall

is_zero:
// Write buffer to standard output
li a7, 64
li a0, 0x01 // write syscall arg 0
li a1, 0x100
li a2, 0x05 // write syscall arg 2
ecall // execute write syscall

// Exit with code 0
li a7, 93 // exit syscall #
li a0, 0x00 // exit code = 0
ecall // exit