Worked Example 2

Using the hardware shown in figure 1 implement a 'Signal' like communications mechanism. A 'Signal' is a type of inter-process communication developed in the 1970's in Bell Labs Unix and still used in the POSIX standard.

“A signal is a software interrupt delivered to a process. The operating system uses signals to report exceptional situations to an executing program. Some signals report errors such as references to invalid memory addresses; others report asynchronous events, such as disconnection of a phone line.”

GNU C Library

When a 'Signal' is generated the main program's normal flow of execution is interrupted, control then moves to the 'Signal' handler. This software routine selects the correct subroutine to handle the signalled event.

The system shown in figure 1 can approximate a 'Signal' communications mechanism by writing a non-zero data value to the output port. This 8bit value is fed into an 8bit OR gate, generating an interrupt. The interrupt service routine can read the output port value using the input port, selecting the appropriate 'Signal' handler i.e. subroutine.

The system will support eight 'Signals' using a one-hot encoding. Each signal is represented by an output port bit, as shown in figure 2. The highest priority 'Signal' is represented by the least significant output port bit 0. The lowest priority 'Signal' is represented by the most significant output port bit 7. When processing a 'Signal' the interrupt service routine must always select the highest priority 'Signal' handler. The interrupt service routine will then de-active this 'Signal' by writing a logic '0' to the corresponding output port bit position.
Note, multiple 'Signals' may be active at the same time, therefore, the previous output port value must be updated using logical AND instructions to clear a bit position and logical OR instructions to set a bit position.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Divide by zero detected</td>
</tr>
<tr>
<td>1</td>
<td>Arithmetic overflow detected</td>
</tr>
<tr>
<td>2</td>
<td>Invalid instruction memory address</td>
</tr>
<tr>
<td>3</td>
<td>Invalid data memory address</td>
</tr>
<tr>
<td>4</td>
<td>Register file full</td>
</tr>
<tr>
<td>5</td>
<td>Data memory full</td>
</tr>
<tr>
<td>6</td>
<td>Stack memory full</td>
</tr>
<tr>
<td>7</td>
<td>Program finished / system idle</td>
</tr>
</tbody>
</table>

Figure 2: Input / Output port signal bit positions

**TASK1**

Write a subroutine that could be used by the main program to trigger a divide by zero signal i.e. will set output port bit 0 to a logic '1'. Bits 1 – 7 must not be altered.

**TASK2**

Write the interrupt service routine used to process the Signals. This code must not change any registers used by the main program i.e. when control returns back to the main program its registers must be in the same state as before the Signal was triggered.

For the purposes of this exercise you do not need to implement the individual 'Signal' handlers i.e. the subroutines selected in the interrupt service routine, these can be stubbed out e.g.

```assembly
Divide_By_Zero:
    ; < INSERT CODE HERE LATER >
    RET
```

Your task is to implement the required interrupt service routine software:

- Checkpoint the processor's state
- Identify the highest priority signal
- Clear (zero) the corresponding output bit
- Call the Signal handler
- Restore the processor's state

Modify your code to allow a Signal to be called from within a Signal handler.