ECE391: Computer Systems Engineering **Fall 2024**
Machine Problem 2 **Midterm 1: Thursday October 3rd**
Checkpoint 1: Monday October 7th 11.59pm
Checkpoint 2: TBD

PLIC and UART Drivers

Note: Must commit to the **mp2** branch in your provided GitHub repository by assignment deadline.
Please read the entire document before you begin.

Introduction

Running machine-independent software on bare-metal hardware requires a layer of machine-dependent abstraction. For example, in the *Star Trek* game, printing a character to the screen may simply involve calling `printf` in the software. However, under the hood, `printf` communicates with the UART (Universal Asynchronous Receiver/Transmitter) to convert an ASCII character into a visible tile on the screen. This communication occurs through a common medium that both the CPU and the device can access. Often, this medium is memory-mapped.

In this MP, we use **Memory-mapped I/O (MMIO)** for such communication. There are different methods to enable the CPU to interact with devices via MMIO. The simplest method is called **polling**, where the CPU repeatedly checks the device for updates by reading specific memory locations. However, polling can be inefficient if the CPU checks too frequently compared to the rate of I/O operations.

To address this, instead of having the CPU constantly check for updates, we can let the device notify the CPU when updates occur. This is done through a mechanism called **interrupts**. To manage interrupts, we use a device known as the **PLIC (Platform-Level Interrupt Controller)**, which coordinates different devices and manages interrupt status.

In this checkpoint, you will implement interrupt-based MMIO communication between the CPU and devices. This includes writing functions for UART operations (initialization, reading, and writing) and setting up the PLIC (enabling/disabling interrupts, claiming/completing interrupts, setting priorities, etc.). By the end of the checkpoint, you will have the *Star Trek* game running on a system with your implemented PLIC and UART drivers.

MP2 Assignment

You will need to edit the following files:

- `intr.c`
- `plic.c`
- `serial.c`

In each file, you will add your implementation where you find comments labeled `// FIXME`. The functions that you need to implement are also listed in the sections below.

PLIC

Short for Platform-level interrupt controller, PLIC prioritizes and distributes global interrupts in a RISC-V system. As described in the RISC-V ISA manual (PLIC manual) the PLIC connects global interrupts sources, such as I/O devices, to interrupt targets such as hart contexts. The PLIC contains several interrupts gateways, one per interrupt source, and together with a PLIC core that prioritizes the interrupts and routes them. You can find more information on the PLIC in the class slides. For this CP, you will be responsible for basic PLIC initialization and enabling timer interrupts for the game described.

PLIC Function Implementations

You are required to implement the following PLIC functions:

- `void plic_set_source_priority(uint32_t srcno, uint32_t level)`:
  Sets the priority level for a specific interrupt source.
  This function modifies the `priority` array, where each entry corresponds to a specific interrupt source.
- `int plic_source_pending(uint32_t srcno)`:
  Checks if an interrupt source is pending by inspecting the `pending` array.
  It returns 1 if the interrupt is pending, 0 otherwise. The pending bit is determined by checking the bit corresponding to `srcno` in the `pending` array.
- `void plic_enable_source_for_context(uint32_t ctxno, uint32_t srcno)`:
  Enables a specific interrupt source for a given context.
  This function sets the appropriate bit in the `enable` array. It calculates the index based on the source number and context, and sets the corresponding bit for the source.
- `void plic_disable_source_for_context(uint32_t ctxno, uint32_t srcno)`:
  Disables a specific interrupt source for a given context.
  This function clears the appropriate bit in the `enable` array. Similar to `plic_enable_source_for_context`, it calculates the correct bit to clear for the given context and source.
- `void plic_set_context_threshold(uint32_t ctxno, uint32_t level)`:
  Sets the interrupt priority threshold for a specific context.
  Interrupts with a priority lower than the threshold will not be handled by the context.
- `uint32_t plic_claim_context_interrupt(uint32_t ctxno)`:
  Claims an interrupt for a given context.
  This function reads from the `claim` register and returns the interrupt ID of the highest-priority pending interrupt. It returns 0 if no interrupts are pending.
- `void plic_complete_context_interrupt(uint32_t ctxno, uint32_t srcno)`:
  Completes the handling of an interrupt for a given context.
  This function writes the interrupt source number back to the `claim` register, notifying the PLIC that the interrupt has been serviced.

UART

You have learned in class that the UART (Universal Asynchronous Receiver/Transmitter) allows the CPU to communicate on a serial line, byte by byte. In this checkpoint, you will focus on integrating the UART into the star trek game you ran for MP0. You will do this by writing UART functions in C code to handle serial communication from Star Trek to the Qemu Emulator. Below are some of the functions you will implement:

UART Function Implementations

You are required to implement the following UART functions:

- `void com1_init(void)`
  Initializes the UART.
  This function initializes the UART. It should set up the receive and transmit ring buffers, register and enable its interrupt service routine (ISR) with the PLIC, and configure the control registers for the UART. It should set the baud rate divisor to 1, flush the receive buffer, and enable the interrupt for receiving data.
- `void com1_putc(char c)`
  Writes a character to the UART.
  This function writes a character to the UART transmit ring buffer. If the buffer is full, it waits until there is space to write the character. It also enables the appropriate interrupt to ask the UART to send the character.

- `char com1_getc(void)`
  Reads a character from the UART.
  This function reads a character from the UART receive buffer. If the buffer is empty, it waits until there is a character to read. It also enables the appropriate interrupt to ask the UART to receive the character.

- `static void uart1_isr(int irqno, void* aux)`
  UART interrupt service routine.
  This function is the interrupt service routine (ISR) for the UART. It manages the UART interrupt by first checking the line status register to determine the current state of the UART. If there is data to be received, it reads from the receive buffer register (RBR) and stores the data in the receive ring buffer, unless the receive ring buffer is full. If the transmit holding register (THR) is empty, it writes from the transmit ring buffer to the THR, unless the transmit ring buffer is empty. Additionally, the function disables the corresponding interrupt when the receive ring buffer is full or the transmit ring buffer is empty, preventing further interrupts until the buffers are ready for more data.

External Interrupt Handler

The external interrupt handler is a general interface into all external interrupts being signaled to a core. Any external device (non-core local devices) go through the PLIC and the external interrupt handler to reach the appropriate interrupt service routine.

Interrupt Handler Function Implementation

- `void extern_intr_handler(void)`
  Generic External Interrupt Handler
  This function is supposed to handle all external interrupts coming through the PLIC. It does not perform device specific actions but is meant to redirect each interrupt to it's appropriate handler based on the interrupt source.

How to get things started

You can follow these steps to pull the assignment code from the release repo:

1. Create a local branch for MP2 and switch to it:

```
git switch main
git branch mp2
git checkout mp2
```

2. Fetch the MP2 branch from the release repo:

```
git fetch release mp2
```

3. Merge release/mp2 into the local mp2 branch:

```
git merge --allow-unrelated-histories release/mp2
```

Running and Testing

Once you have completed your implementation, you can run your code using QEMU by executing the following command:

```
make run-cp1
```

You can also run gdb by running

```
make debug-cp1
```

To view the output in a different terminal, use the screen command:

```
screen /dev/pts/N
```

where `N` is the number provided by QEMU on the first line of output. This will ensure you can see the UART output properly.

To help you verify your implementation, you are provided with a gold version, `cp1-gold.elf`, located in the `cp1-distr` directory. Run this version to see the expected output and behavior. You should compare your implementation's output with this to ensure correctness.

Submission

To submit this assignment, you need to push the files given to you on the release repository to your class repository on the **mp2** branch. We will only grade submissions made to the MP2 branch by that deadline. That is mp2, all lowercase. The assignment files are supposed to be at the root directory of the branch. We will not grade any submissions on your repository if the submission files are under a different directory. A new requirement for submission this MP is that you don't commit any object files with your submission (.o). It is bad Git practice to commit object files, since they can be reproduced from source files. The only files that will be graded are

1. serial.c
2. intr.c
3. plic.c

Make sure that your implementation does not depend on changes to any other files

Good luck!