CS220 Computer Organization

Spring 2012

Machine Problem 4

Due March 9, 2012. Late work accepted within reason, but no guarantee of help from me after this date.

Objectives

The objective of Machine Problem 4 is to learn about *exceptions* on stored program computers. You will study the following description of exceptions, which applies to any computer, and you will write two interrupt handlers for the PDP-8, along with an application-level subroutine that interacts with one of the interrupt handlers to provide interrupt-driven printing. The interrupt-driven system allows characters to be printed on the slow Teletype in parallel with computation carried out by the processor. You will also be exposed to the concept of a *critical section* of code, where care must be taken to avoid what is often called a *hazard* or *race condition.*

Exceptions

An *exception* is a condition that arises that requires an exception to the rule that the program counter tells where to get the next instruction. The obvious questions are: What are these conditions? Where does the address of next instruction come from, if not the PC? What does the processor need to do when this happens?

The conditions that cause an exception fall into two broad categories:

- Conditions that are caused by executing, or attempting to execute, an instruction. These conditions are internal to the processor, and arise synchronously with the execution of instructions. Often these conditions are called *traps* or *faults*.

- Conditions that arise external to the processor and asynchronously with the execution of instructions, generally as a result of an external device needing attention. These conditions are almost always called *interrupts*.

The terminology is not universally agreed upon. For example, Intel uses "exception" to refer to internal conditions only[1], while ARM uses it to refer to both kinds, reserving "interrupt" for external conditions but somewhat inconsistently calling the internal, synchronous SWI instruction an interrupt[2]. This inconsistency in the industry reflects the historical evolution of exceptions.

Examples of internal exceptions include:

[1] *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture*. September, 2010.
[2] *ARM Architecture Reference Manual, Second Edition,* June 2000.

- Attempting to execute an illegal instruction (an unused, undefined bit pattern). On the PDP-8, for example, 7014 is illegal because both the RAL and RAR bits are set (rotate left and right). The PDP-8 does not have internal exceptions, however, so no trap occurs. Our simulator, which is a Windows application and not a PDP-8, displays a message for illegal instructions and halts the processor.

- Attempting to access memory that does not exist, or to which you don't have permission. Sometimes this is called a *memory access violation*. One type of memory access violation is attempting to write a location in memory that is considered read-only. Processors use a *memory management unit* (MMU) to control access to memory, including granting or denying access permission. The PDP-8 does not have an MMU. One can access all of memory, including nonexistent memory.

- On a system that implements virtual memory, attempting to access a memory location that is not currently loaded into the addressable memory of the computer, and therefore has to be read into memory from a disk. These are usually called *page faults* or *segment faults*, depending on details that are beyond the scope of this document.

- Executing an arithmetic instruction that produces an unexpected result, such as dividing by 0 or exceeding the largest or smallest number that can be represented (usually called overflow or underflow). Most computers do not trap these conditions, leaving it up to the programmer to detect and handle them. Those computers that do trap them provide means to disable the trap.

- Making a call to the operating system kernel by executing a special instruction. The ARM SWI instruction ("software interrupt") is an example. The PDP-8/I had an instruction for this purpose that could be added as an extra-cost option, and which is not implemented on our simulator.

Examples of interrupts include:

- A transfer between memory and a disk drive has completed.

- A mouse have moved one unit.

- A power failure is imminent. The PDP-8 had a power-fail interrupt that could be added as an extra-cost option.

- A network interface has received a packet.

- A clock has ticked. On the PDP-8, the optional KW8/IF clock can generate an interrupt after a programmable interval, which on our simulator is between 1 and 4096 milliseconds.

On our PDP-8 simulator, an interrupt can be generated when:

- a character has been received from the Teletype keyboard;

- a character has finished printing on the Teletype printer;
- a disk transfer has completed; and
- the programmed clock interval as elapsed.

In general exceptions are either *maskable*, which means that they can be enabled or disabled by executing appropriate instructions (i.e. under software control), or *non-maskable*, which means that they cannot be disabled. Most internal exceptions are non-maskable because the condition prevents the instruction from being executed---there is no choice but to trap (or perhaps, Star Trek-like, halt and catch fire). Arithmetic traps, if they exist at all, are usually maskable. Interrupts are almost always maskable, although some processors have a special non-maskable interrupt for emergency conditions.

Maskable interrupts can be enabled or disabled globally, i.e. by means of a single state bit that can disable all interrupts, and on a per-device basis. An interrupt will occur for a given device if an interrupting condition exists, interrupts are enabled for that device, and interrupts are globally enabled. Here the "and" is important—all three conditions must be true.

On the PDP-8, interrupts are disabled on system reset (i.e. when the Start switch is pressed), enabled with the ION instruction, and disabled with IOF. On the PDP-8/I as shipped by DEC interrupts could not in general be disabled on a per-device basis, but the value of doing so is such that some customers added it to their machines (the Air Force did this to the PDP-8 that they used to capture and record radar data while chasing hurricanes in a C-130, and which we at MIT Weather Radar got surplus from them). Following this tradition, our simulator has an I/O instruction that can enable or disable interrupts on each device.

Each device on the PDP-8 has at least one flag, for example the keyboard and printer flags with which you are familiar from polled I/O. An interrupting conditions exists on a device if and only if its flag is set. The interrupt will occur if interrupts are enabled on that device and are globally enabled with ION.

When an enabled exception condition occurs (for external devices, we say that a device has *raised an interrupt*) the processor (hardware) takes these steps:

1. Globally disable all maskable interrupts so that no new ones can occur until the software is ready for them.
2. Save a portion of the state of the processor, where the portion always includes the program counter and may also include other elements of processor state.
3. Place in the program counter one of a small number of fixed addresses so that execution of instructions begins at that address. The specific address is chosen from the small number of possibilities to provide a partial or complete indication of the cause of the exception. For example, the ARM has seven fixed addresses for exceptions, five for specific internal exceptions and two for broad classes of interrupts. The set of fixed addresses is called an *exception vector* or *interrupt vector*.

For an interrupt on the PDP-8, the program counter is saved in memory location 0000, and the fixed address 0001 is placed in the program counter. Thus an interrupt is identical to the instruction sequence

```
IOF
JMS 0
```

Since there is only one fixed address, no information about the cause of the exception can be obtained from the location at which execution begins; it must be obtained in other ways, as described in `iHandle.pal8`.

When the program counter is changed to a new address, so that execution of instructions begins at that address, we often say that *control is transferred to*, or *returns to*, or is *passed to*, that address. This change of program counter may be the result of an exception, or of returning from an exception, or even of a simple jump or branch instruction.

When an exception occurs, the software, called an exception handler or interrupt handler, must do at least this:

1. Save that portion of the state of the processor that was not saved by the hardware and that might be changed before control returns to the interrupted program. For some exceptions, such as an illegal instruction, control may never return and so no state need be saved. In a multi-threaded system an exception might cause execution of one thread to be suspended and control to be passed to a different thread, in which case the entire state must be saved. While it is always safe to save the entire state, in cases where rapid exception response is critical it is reasonable to save just that portion of state that will be changed by the interrupt handler. On some processors, such as the ARM, multiple independent copies of certain portions of processor state are maintained by the hardware so that those portions need not be saved and restored, improving exception response time.

2. Determine the cause of the exception and transfer control to a specific exception handler, which must do this:

    a. Take actions appropriate for the specific exception. In a multi-threaded system, this may cause the current thread to be suspended and another one to become active.

    b. Remove the exception condition.

3. Restore the state of the processor. In a multi-threaded system, this means restoring the state of the thread to which control is about to be transferred, which may or may not be the thread that was executing when the exception occurred.

4. Re-enable maskable interrupts and return to the interrupted program.

In the above description, maskable interrupts are disabled for the duration of the exception handler. It is also common to rank-order interrupts by importance, called *priority*, and to enable

higher priority interrupts while a lower priority interrupt handler is running, so that the interrupt handler itself can be interrupted by more urgent business.

If a non-maskable exception occurs during an exception handler, for example an illegal instruction, it is often a fatal error from which no recovery is possible. This is often the cause of the infamous “blue screens” on Windows computers.

Interrupt-Driven Printer Algorithms

For MP4, you will be implementing the above steps on the PDP-8, including two specific interrupt handlers, clock and Teletype printer. You will also be writing a write character subroutine that interacts with the printer interrupt handler. More complete instructions are provided in `iHandlex.pal8`.

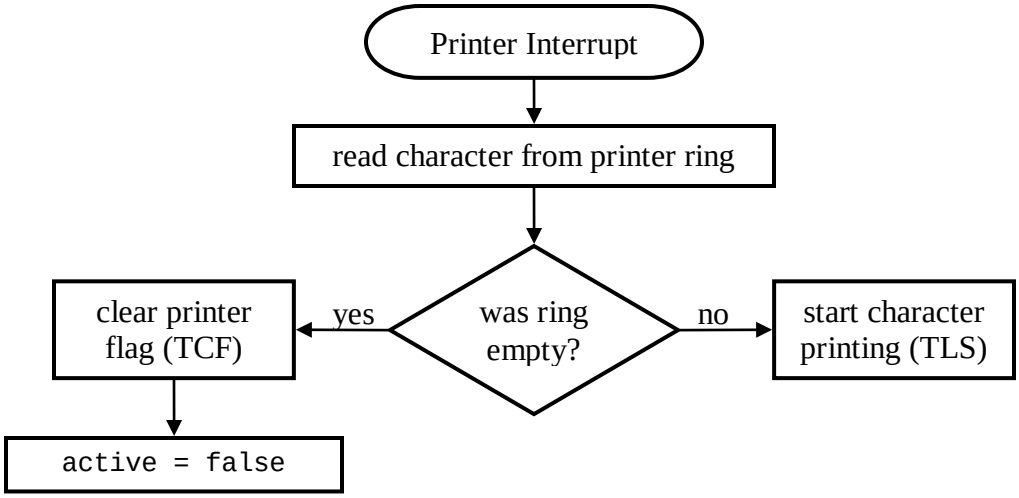
With polled I/O the write character subroutine waits for the printer to be ready before printing the character, typically spending nearly 100 ms doing nothing useful. With interrupt-driven I/O the write character subroutine puts the character in a first-in first-out buffer (FIFO), and returns immediately without waiting for the printer. The time needed to do this on the PDP-8 is closer to 100 μs, 1000 times faster than waiting for the printer. The processor can then resume doing useful work.

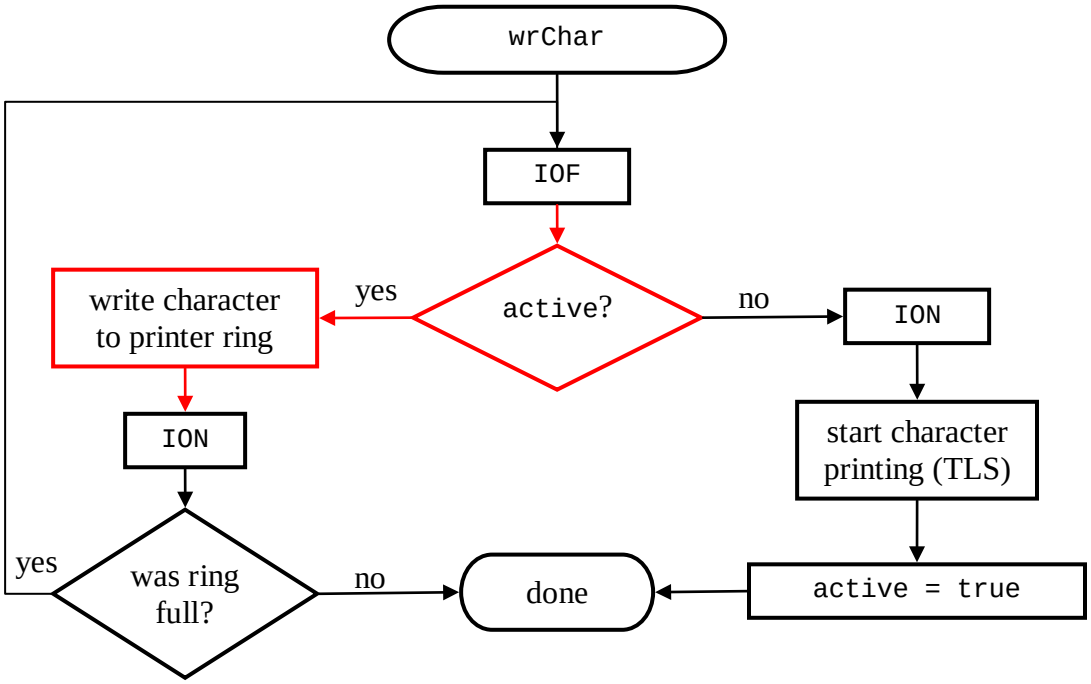
Whenever the printer is finished with a character, an interrupt occurs (because the printer flag is set, causing an interrupt condition). The interrupt handler takes a character out of the FIFO and sends it to the printer, very briefly interrupting whatever useful work the processor is doing. Printing thus proceeds at its own pace and in parallel with the useful work. Because the interrupt occurs and is handled as soon as a character is finished printing, the printer runs as fast as it can. Because the application program generally doesn’t have to wait for the printer, it also runs as fast as it can.

The only exception is if the FIFO becomes full, so that it can’t hold any more characters. In that case the write character subroutine has to wait for the interrupt handler to remove one or more characters, and useful work is suspended. On a more sophisticated multi-threaded system, the thread that has to wait would be suspended and other threads could do useful work. The MIT Weather Radar PDP-8 had a multi-threaded OS and could find other things to do.

The above description has to be augmented slightly to actually work, as will be seen in the flowcharts below.

We use a ring buffer to implement a FIFO. The advantages of a ring buffer are simplicity, high speed of operation, and nearly fixed time needed to read or write. The disadvantage is fixed size, so that they can fill up and be unable to receive more data. More sophisticated FIFOs are often provided in libraries of modern object-oriented programming languages like C++ or Java, including FIFOs that can grow arbitrarily large so that they never fill up. These FIFOs, however, are slower and the time needed to read or write can be quite variable because sometimes they need to get more memory to use. They are not suitable for interrupt handling.





In these flowcharts, we read or write the ring and then ask whether the operation was successful, i.e. was the ring empty or full. This is due to an arbitrary choice that I made in designing the ring subroutines. It could have been done in the other order, first testing for full or empty and then reading or writing. To do that I would have needed to provide separate routines for testing, which I did not feel like doing.

In the flowchart for the write character subroutine, the section in red is a *critical section*. In this case the critical section is a place where the programmer must guarantee that a printer interrupt cannot occur. The reason is that the printer interrupt can change the state of the `active` variable from true to false, and can change the state of the ring buffer. If active goes from true to false in the critical section, the character will go in the ring but no interrupt will occur to pull it out and print it. The next time write character is called the inactive branch is taken and the character is printed immediately, and therefore in the wrong order.

Much more serious is the possibility that an interrupt might change the state of the ring buffer in the critical section, which can put the buffer in an inconsistent state. Furthermore, due to the limitations of the PDP-8 instruction set (primarily lack of register-based addressing), the ring buffer subroutines are not re-entrant and can crash hard if the interrupt handler tries to read in the middle of the write character subroutine trying to write.

While we could disable just printer interrupts in the critical section, it is easier to disable all interrupts with IOF. You are welcome to disable just printer interrupts if you want.

If the critical section lasts 100 μs and a printer interrupt occurs every 100 ms, the probability of a printer interrupt in the critical section is 0.001 (assuming the timing of the two are uncorrelated). The probability that such an interrupt will cause a crash is even lower. Unlike an ordinary bug, code with a critical section bug will appear to work perfectly almost all of the time. It will fail rarely and non-deterministically. People have been killed by machines with this kind of bug. You usually can't find it by single stepping in a debugger, because single stepping has completely different timing from full-speed running. These bugs must be avoided by clear thinking and careful design.

You may find that it is easier to put the ring full test in the critical section, due to the way that the write ring subroutine works. I didn't do that, but you can. However, you must have part of the ring full loop running with interrupts on, or else the ring will never become non-full. Remember that on the PDP-8 there is a one-instruction delay from executing ION until interrupts are enabled.

When the ring is full the flowchart goes back to the active test, when it might seem that it could just as well go back to the place where it tries to write the ring (thanks for asking, Wils). The reason I do this is that the printer interrupt handler is empowered to change `active` from true to false, and so I want to test it again if interrupts have been back on. In practice in this program it is impossible for the interrupt handler to completely empty the ring and go inactive in the few microseconds it takes to get from the ring full test back to the active test, but it is only impossible because of timing, not logic. If the ring buffer could hold only one character, the timing would allow failure. If this were a multi-threaded system, another thread could take control and disrupt the timing to the point that the logic would fail. Do not write code that only works correctly if certain timing assumptions hold, no matter how reasonable those assumptions appear to be. Get the logic right.

Bugs caused by the relative timing of multiple processes are called *hazards* or *races*. MP4 explores a very simple example with a very simple solution, disabling interrupts in the critical

section. In multi-threaded systems more complex situations and solutions exist, which are beyond our present purposes.

Source Code

MP4 includes these PAL8 source files:

`izero`	Page zero, including interrupt entry point and `active`.
`iHandlex`	Framework for your code. Do your work in here, following the comments provided. Ring buffer initialization already provided.
`ring`	Ring buffer code. If you are curious, read this to see how it works. Note that the implementation reflects the archaic instruction set of the PDP-8, so some of this is instructive and some is not.
`itty`	Teletype I/O utilities that I wrote for my version of MP2, with `wrChar` removed so it can be replaced with an interrupt-driven version.
`ie`	Interrupt-enabled version of main program for calculating and printing *e*. Initializes and enables the interrupt system, waits for all characters to be printed before halting. Defines the number of bits to use and *n*. Calculates and prints the ratio by calling subroutines in other files.
`bigutil`	Utility subroutines, including argument fetching, a bignum stack, and routines for copying and initializing bignums.
`bigadd`	Subroutines for adding, negating, and subtracting bignums.
`bigmul`	Multiply subroutine. For the curious, in here is my solution to MP3.
`bigdiv`	Divide subroutine. Very simple and slow, one bit at a time. Have a look and see if you understand what it's doing.
`bigprint`	Print bignum in decimal, and print ratio of two bignums as a decimal fraction.

Note that the files whose names start with `big` are identical to the ones from MP3. You must put `izero.pal8` first, but otherwise the program should work with the files in any order. I've tested them only in the order given above, however.

Procedure

Read this document completely. If there are things you don't understand, ask questions in class. I can't imagine that anyone will understand all of it, I am not that good a writer.

Look at `ie.pal8` and see that you understand these things:

- Unlike `e.pal8` in MP3, there is no `TLS` at the start of the program, and indeed there must not be.

- Note how interrupts are enabled, and how the clock is started. You'll need to restart it the same way in your clock interrupt handler.

- Note how we wait for all characters to finish printing at the end, before halting.

Do all of your work in `iHandlex.pal8`. You will find comprehensive instructions in comments in that file.

Deliverables (Blackboard)

- Your PAL8 `iHandlex.pal8` source file.

- The file Asr33Paper.txt, created automatically by the simulator when it exits and showing everything printed on the Teletype. This file will show me the output of your program.

Put whatever files you want to submit in a ZIP folder. Do not submit multiple files except in a ZIP folder. Please don't use RAR, it's too hard for me to read. If you want to add separate comments to me, put them in a document in the ZIP folder along with your code. The filename of the ZIP file that you submit should start with your last name. If you follow these rules, I'll be in a better mood when I grade your work.