I/O and Interrupts

CS 350: Computer Organization & Assembly Language Programming

A. Why?

- Communicating with I/O devices requires hardware interfaces.
- Implementing I/O is hard partly because of timing issues.
- Interrupts are the common way for I/O devices to signal status changes.

B. Outcomes

After this lecture, you should know

- What low-level I/O data and control registers do.
- How memory-mapped I/O works and how the hardware needs to be changed to support it.
- How polled I/O works and when it's useful.
- Why interrupt-driven I/O was created and how the CPU supports it.
- That hardware interrupts are external signals that can affect execution.
- What makes up the processor state and why and how it needs to be saved and restored when handling interrupts.

C. Peripherals

- A **peripheral** device is any device outside the CPU. One way to characterize them is by direction of data: Input (e.g., keyboard, motion detector, network interface) Output (e.g., monitor, printer); Storage (disk, CD-ROM). Another is the data transfer rate, since higher rates impose more restrictions on how quickly the data must be handled. (E.g., keyboard: 100 bytes/sec; Disk: 30 MB/s; Network: 1 Mb/s - 1 Gb/s).
- I/O devices have **I/O control/status registers** and **I/O data registers** that serve as the interface between the computer and the device. Control/status registers are used to tell devices to do things and for devices to tell us about

their state; data registers hold data going to or from the device. (The MAR and MDR used by memory are data registers.)

- Originally people set aside special opcodes to use for I/O (that's how the SDC works). But having **special I/O instructions** means that every time you want to add a new I/O device or change how we interact with an I/O device, you have to add or modify the control unit to deal with additional or modified I/O registers.
- Instead, people now use **memory-mapped I/O**. Each I/O register has a fake memory address associated with it. To access an I/O register, you do a load or store with that memory location. The **memory-management unit** (MMU) checks the address in the MAR, and if the address indicates an I/O register, the MMU accesses the I/O register instead of the memory location.
- The LC-3 uses memory-mapped I/O[1]; it has two devices, the keyboard and monitor, and for each device it has a control register and a data register. The various LC-3 I/O traps (`GETC`, `PUT`, etc.) use these locations to do I/O.
- The LC-3 keyboard status and data registers are called `KBSR` and `KBDR` and accessed using memory locations `xFE00`, `xFE02`.
  - `KBDR`[`7:0`] = last char typed on keyboard
  - `KBSR`[`15`] is the **ready bit**: it's 1 iff the keyboard has received a new unread char.
- How the LC-3 handles keyboard input: When a character is typed, the keyboard hardware automatically
  - Sets `KBDR`[`7:0`] ← ASCII code of char (upper 8 bits ← 0)
  - Sets `KBSR`[`15`] ← 1 (enable "ready bit": Ready to be read)
  - While `KBSR`[`15`] = 1, the keyboard is disabled (hitting keys does nothing)
  - The `GETC` trap code uses a short loop to repeatedly test (a.k.a. **poll**) `KBSR` until its bit 15 is 1. When that happens, the trap code reads the input data by loading `KBDR` into `R0`. Loading `KBDR` automatically causes the hardware to reset `KBSR[15]` ← `0` (to indicate we're not ready to

[1] Note on the final project: You don't have to implement the I/O registers.

read a character) and re-enables the keyboard so that the next key press will not be ignored.

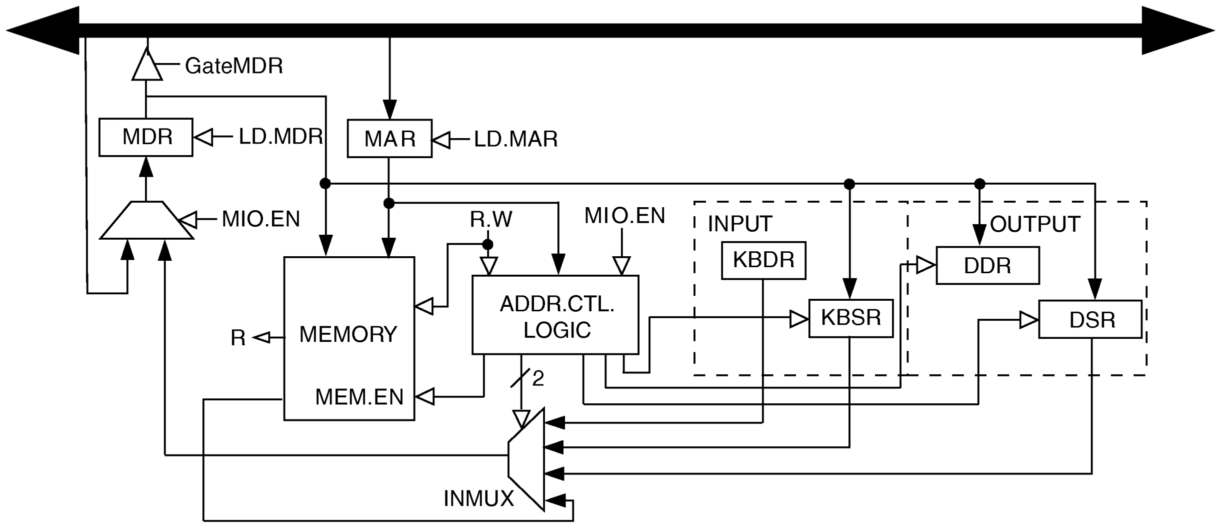
- Code for `TRAP x20` (`GETC`):

```
x0020  .FILL  x0400       ; Jump table entry for TRAP x20
     ...
x0400   ST    R7, x0408   ; Save R7
x0401   LDI   R0, x0406   ; Get keyboard status
x0402   BRZP  x0401       ; Loop until keyboard has char
x0403   LDI   R0, x0407   ; Get the character
x0404   LD    R7, x0408   ; Restore R7
x0405   JMP   R7          ; Return to caller

x0406  .FILL  xFE00       ; Pt keyboard status register
x0407  .FILL  xFE02       ; Pt keyboard data register
x0408  .BLKW  1           ; Save space for R7
```

- On the LC-3, monitor output is handled similarly. There are monitor status and data registers called `DSR` and `DDR` that are accessed using memory locations `xFE04` and `xFE06`. When the monitor is ready to display a char, the hardware automatically sets `DSR`[`15`] $\leftarrow$ 1 (enables its “ready bit”).
- To display a character, the `OUT` trap code stores the character to be printed from `R0` into the `DDR`. When the `DDR` is accessed, the monitor hardware automatically resets `DSR[15]` $\leftarrow$ `0` (disables the ready bit) to indicate that the monitor is in the process of displaying the data character. (Assignments to `DDR` will be ignored while `DSR[15]` is `0`.) When the monitor has finished displaying the character, it sets `DSR[15]` $\leftarrow$ `1` to indicate that it’s ready for the next character.

- Block diagram for LC-3 memory-mapped I/O



D. Polled and Interrupt-Driven I/O

- People, the CPU, and I/O devices differ greatly in how quickly they process data. (Maybe 100 words/min for a typist, > 200 MB/sec for a hard drive, and > 2 GHz for the CPU (at maybe 5 cycles per instruction). In addition, the data transfer rate for a device might vary over time (e.g., nothing happens with the hard disk for a while then suddenly we receive megabytes of data).
- In the earliest systems (like the SDC and LC-3), when we wanted to do I/O, we would use short loops that repeatedly poll the I/O device[2]. If the expected time for the I/O operation to complete and be handled is long, this **polled I/O can be very inefficient**.
- As an alternative, people thought about just going off and executing some other program while the first one waits for I/O. Problem is, how do we make sure that our original program eventually continues execution?
- People extended the hardware so that I/O devices can send **interrupt signals** to the CPU to indicate changes in status that need to be handled.

[2] It's like asking "Are we there yet? Are we there yet? Are we there yet? ...."

- In **interrupt-driven I/O**, our program does not check the I/O status. When the I/O is complete, the device sends an interrupt signal to the CPU. This signal causes our program execution to be suspended while we handle the interrupt. Our program is resumed once the interrupt is handled.
- Note: The LC-3 implementation does not support interrupts.

E. Handling Interrupts

- Handling an interrupt is similar to handling a phone call — we're doing some activity, the phone rings, we stop what we're doing, answer the phone, and when the phone call is done, we go back to what we were doing.
- To support interrupts, the control unit has to be modified. An **interrupt signal** line has to be added, and **the instruction cycle has to be modified**: Between the last step of the current instruction cycle and the Fetch instruction phase of the next cycle, we add a check for an interrupt signal. If no interrupt has been signaled, we just continue with the instruction cycle as usual.
- If an interrupt has been signaled, we first save the current state of the processor (the PC and condition code, for example). For phone calls, this is analogous to writing ourselves a note saying what we were doing when the phone rang. Then, we change the PC to hold the address of the code that handles the signaled kind of interrupt. In effect, we cause control to jump to the interrupt-handling code.
- Once the interrupt-handling code has completed its work, we can jump back to the interrupted program by restoring the saved processor state (say PC and CC), or we can execute some other program by setting the processor state appropriately.
- A program that's been temporarily interrupted shouldn't notice that anything has happened (except for side effects like the wall clock suddenly jumping forward).

F. Disabling Interrupts; Prioritized Interrupts

- Not all operations can be interrupted. E.g., if the OS maintains a linked list of all active programs, then it needs to update that list when an interrupt occurs, and we can't interrupt the update while the list pointers are in an inconsistent state. There needs to be a flag in the control unit that says whether or not interrupts are enabled or disabled.

- More generally, can we allow a second interrupt to occur while handling the first interrupt? Some interrupts are more important than others (the system timer interrupt shouldn't interrupt the power-failure interrupt handling code).
- To solve this problem, people assign priorities to each interrupt; a lower priority interrupt will be at least temporarily ignored while a higher priority interrupt is being handled.
  - The telephone analogy is that a call on the red telephone takes priority over a call on the regular telephone.
- As part of the control unit state, we can maintain an interrupt threshold — an interrupt is allowed only if its priority is higher than the threshold. When an allowed interrupt occurs, the threshold gets set to the interrupt's priority; once that interrupt is handled, we reset the threshold to what it was.
- We can also use this mechanism to disable all interrupts: Just set the threshold to be higher than the highest actual priority. To reenable interrupts, we lower the threshold.
- Note: The LC-3 has interrupt enable bits associated with each I/O device (bit 14 of the status register), but the LC-3 implementation doesn't support I/O interrupts.

G. Privileged Mode, Privileged Instructions, and TRAPs

- Not every user should be able to do things like disable interrupts or access the I/O registers. To maintain security, the control unit keeps a **Privileged Mode** bit. You get a runtime error if you try to execute certain instructions or access certain memory locations when you're in user mode (the opposite of privileged mode).
- In general, OS code runs in privileged mode, which is why the OS can access the I/O registers and do I/O. The privileged mode bit gets automatically turned on when an interrupt occurs; this way, when we jump to the code to handle the interrupt, the OS can use privileged operations. Once the interrupt

is handled, the privileged mode bit can be turned off before we jump back to the user code that was running when the interrupt occurred.

- If you're a regular user, running code in user mode, you need to transfer control to the OS in order to do I/O. This is what `TRAP` instructions (also known as service call instructions) do. Like hardware interrupts, they cause a change to privileged mode and a jump to OS code; unlike hardware interrupts, they occur when the software decides to execute a `TRAP`. For this reason, `TRAP` instructions are also known as **soft interrupts**, as opposed to the **hard interrupts** caused by hardware.
- Note: The LC-3 implementation doesn't check for user vs privileged mode when executing instructions or accessing memory.

H. The Processor Status Register and Interrupt-Handling Stack

- The **processor status** is the information we need to save when changing contexts from one piece of code to another. It includes at least the PC, the contents of the general purpose registers, and the **Processor Status Register** (PSR), which holds the condition code, the interrupt threshold, and the privileged mode bit.
- Note: The LC-3 does contain a PSR (you can see it in the simulator). Bit 15 of the PSR indicates privileged vs user mode but the LC-3 implementation doesn't support modes.
- When an interrupt occurs and has to be handled, we save the processor status in a special **Interrupt Handling Stack**; this is different from the normal process activation stack we saw in the previous lecture because it holds a different kind of context.
- By using a stack, we can ensure that if interrupt handling code is itself interrupted, we'll know the correct processor state to reestablish when we finish handling the higher-priority interrupt. (E.g., if user code is interrupted, we have to change from privileged mode to user mode when returning the user code; if OS code is interrupted, we have to stay in privileged mode when returning.)
- To restore the processor status register, we need a special **Return From Interrupt** instruction, which (of course) is a privileged instruction. (Note: The

LC-3 has an `RTI` but the implementation doesn't support it. An actual RTI instruction would pop the interrupt-handling stack and restore the PC, general-purpose registers, and PSR (and thus the CC, interrupt threshold, and privileged mode bit).

I. Multitasking

- In **multitasking**, we support concurrent execution of multiple programs. (The alternative is uni-tasking.) We execute each program for a quantum of time, and by making the quantum small enough, it looks like all our programs are executing simultaneously.
- In cooperative multitasking, everyone writes their programs so that they occasionally call an OS routine that has the option of transferring control to another program.
- In non-cooperative multitasking (a.k.a. **true multitasking**), the operating system can wrest control from the current program and give it to some other program. The OS sets a timer when it passes control to a program; when the timer goes off, it causes a **timer interrupt**, which sends control back to the OS. The OS can then decide whether or not to give control of the CPU to some other program.
- Scheduling in a multitasking OS is a complicated topic; to give you an example of the issues involved, an **I/O-bound computation** spends most of its time waiting for I/O to finish and little time doing computations whereas a **CPU-bound computation** spends most of its time doing computations and little time doing I/O. It's useful to have a mix of I/O- and CPU-bound programs running concurrently because then both the CPU and the I/O devices can be kept busier than if we ran just one kind of computation.

I/O and Interrupts

CS 350: Computer Organization & Assembler Language Programming

A. Why?

- Communicating with I/O devices requires hardware interfaces.
- Implementing I/O is hard partly because of timing issues.
- Interrupts are the common way for I/O devices to signal status changes.

B. Outcomes

After this activity, you should be able to describe

- How low-level I/O data and control registers are used.
- How memory-mapped I/O works how the hardware needs to be changed to support it.
- How polled I/O works and when it's useful.
- Why interrupt-driven I/O was created and how CPU supports it.
- What hardware interrupts are and how they are handled.
- What makes up the processor state and why and how it needs to be saved and restored when handling interrupts.

C. Questions

1. With memory-mapped I/O, how do we query the status of a device? How does our program send data to (or receive data from) a device? What has to be modified to support a new device?
2. How does polled I/O work?
3. What is a hardware interrupt? How does the control unit have to be modified to handle interrupt-driven I/O?
4. When is polled I/O efficient/inefficient compared to interrupt-driven I/O?
5. One thing interrupt-handling code has to do is to save the user's registers (in case they get changed). Can further interrupts be enabled during the save action (and the symmetric restore action)? Explain briefly.

6. Interrupts were designed to deal with slow or unpredictable devices (like keyboards). Why does it make sense to use an interrupt to control the system timer? (Every time the system timer interrupt occurs, a counter for the time of day gets incremented.)
7. What is privileged vs user mode and why is it used? When is it set/reset?
8. What is the processor status register and how is it changed/maintained?

Solution

1. To query the status of a device, we do a load from a specific memory location assigned for this purpose; the memory management unit translates that load into a query of the device. To send or receive data, we store or load another specific memory location; again the memory management unit translates this to a send/receive of data with the device. To add a new device, we have to ensure that the memory management unit recognizes the fake memory addresses assigned to its control and data registers.

2. In polled I/O, we execute a short loop that repeatedly checks the status of the I/O; the loop stops when the I/O is complete.

3. A hardware interrupt is an external signal indicating that some event occurred. To handle one, the control unit has to be modified so that it interrupt signals can be sent to it. The instruction cycle has to be modified so that we check for interrupts and (if one occurs) save information about the current code being executed and jump to code to handle the interrupt.

4. If the expected time to wait for and handle the I/O operation is short, then polled I/O can be efficient; if the expected time is long, then polled I/O is inefficient. This is because interrupt-drive I/O requires time to change contexts from one program to another.

5. We can't interrupt the register saving operation because otherwise we might lose data by losing track of what the correct register values were

6. Even though the system timer occurs at regular intervals, the clock ticks occur at unpredictable points while user programs execute. So it makes sense to use the interrupt mechanism to access the code that makes the clock tick.

7. Privileged mode is used to keep user code from doing certain dangerous operations or accessing dangerous memory locations (e.g., the I/O registers). While running normal user code, it's off. When an interrupt occurs, it's turned on before jumping to the interrupt-handling code. Once an interrupt is handled, it's turned off before jumping back to user code.

8. As part of changing from one piece of code to another (e.g., from user code to interrupt-handling code), we need to save/restore information about the status of the CPU. The processor status register holds the information that's not included in the PC or general-purpose registers. It's pushed onto/popped off of a special interrupt-handling stack as part of going to/returning from interrupt-handling code.