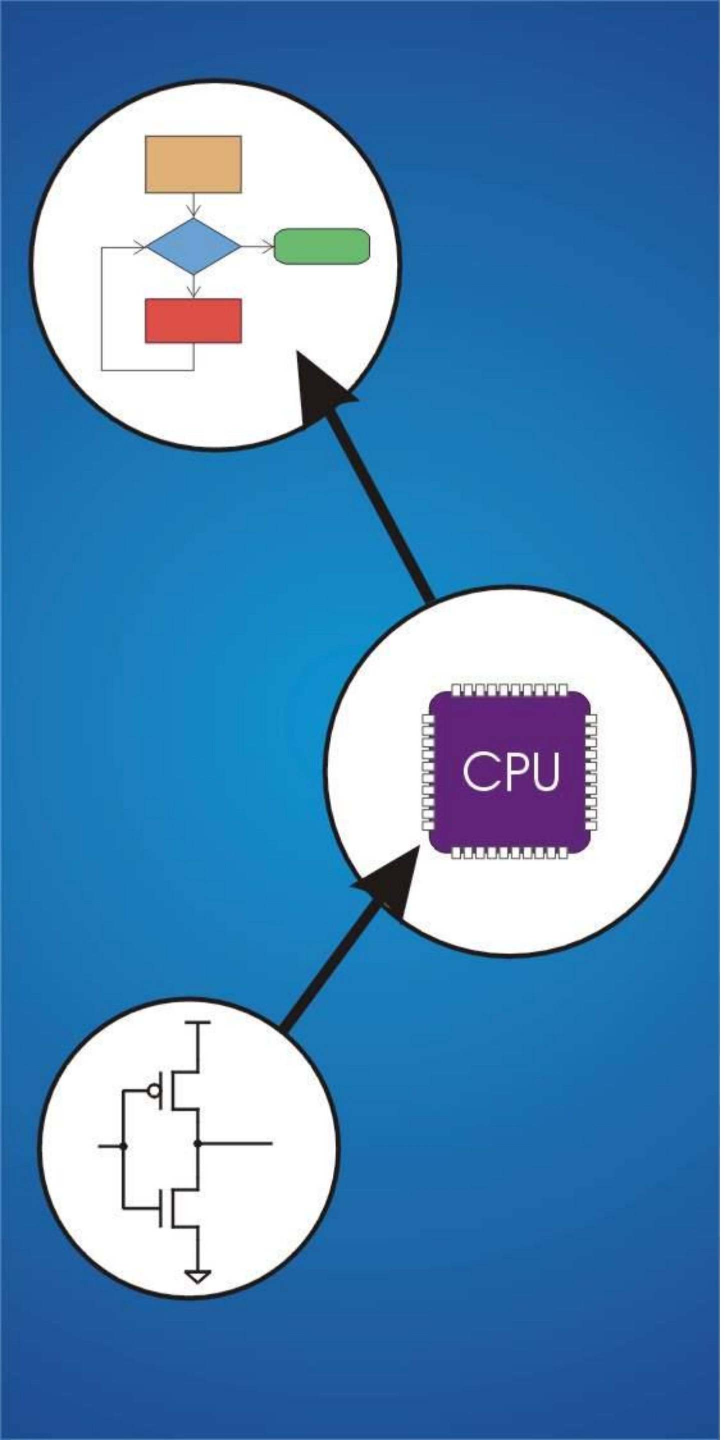


Chapter 8
I/O

I/O: Connecting to Outside World

So far, we've learned how to:

- **compute with values in registers**
- **load data from memory to registers**
- **store data from registers to memory**

But where does data in memory come from?

And how does data get out of the system so that humans can use it?

I/O: Connecting to the Outside World

Types of I/O devices characterized by:

- **behavior:** input, output, storage
  - input: keyboard, motion detector, network interface
  - output: monitor, printer, network interface
  - storage: disk, CD-ROM
- **data rate:** how fast can data be transferred?
  - keyboard: 100 bytes/sec
  - disk: 30 MB/s
  - network: 1 Mb/s - 1 Gb/s

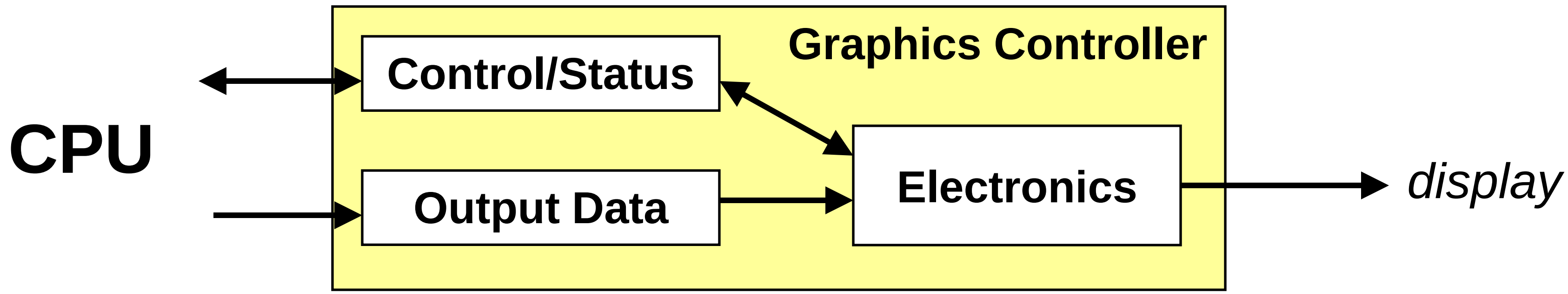
I/O Controller

Control/Status Registers

- **CPU tells device what to do -- write to control register**
- **CPU checks whether task is done -- read status register**

Data Registers

- **CPU transfers data to/from device**



Device electronics

- **performs actual operation**
  - **pixels to screen, bits to/from disk, characters from keyboard**

Programming Interface

How are device registers identified?

- **Memory-mapped** vs. **special instructions**

How is timing of transfer managed?

- **Asynchronous** vs. **synchronous**

Who controls transfer?

- **CPU (polling) vs. device (interrupts)**

Memory-Mapped vs. I/O Instructions

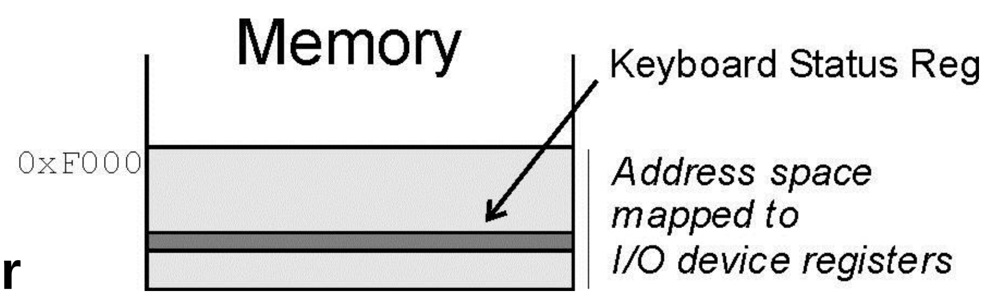
Instructions

- **designate opcode(s) for I/O**
- **register and operation encoded in instruction**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IO				Device								Op			

Memory-mapped

- **assign a memory address to each device register**
- **use data movement instructions (LD/ST) for control and data transfer**



Transfer Timing

I/O events generally happen much slower than CPU cycles.

Synchronous

- **data supplied at a fixed, predictable rate**
- **CPU reads/writes every X cycles**

Asynchronous

- **data rate less predictable**
- **CPU must *synchronize* with device,**
  so that it doesn't miss data or write too quickly

Transfer Control

Who determines when the next data transfer occurs?

Polling

- **CPU keeps checking status register until**
  new data **arrives OR** ***device ready*** **for next data**
- **"Are we there yet? Are we there yet? Are we there yet?"**

Interrupts

- **Device sends a special signal to CPU when**
  new data **arrives OR** ***device ready*** **for next data**
- **CPU can be performing other tasks instead of polling device.**
- **"Wake me when we get there."**

LC-3

Memory-mapped I/O (Table A.3)

Location	*I/O Register*	*Function*
`xFE00`	Keyboard Status Reg (KBSR)	Bit [15] is one when keyboard has received a new character.
`xFE02`	Keyboard Data Reg (KBDR)	Bits [7:0] contain the last character typed on keyboard.
`xFE04`	Display Status Register (DSR)	Bit [15] is one when device ready to display another char on screen.
`xFE06`	Display Data Register (DDR)	Character written to bits [7:0] will be displayed on screen.

Asynchronous devices

- **synchronized through status registers**

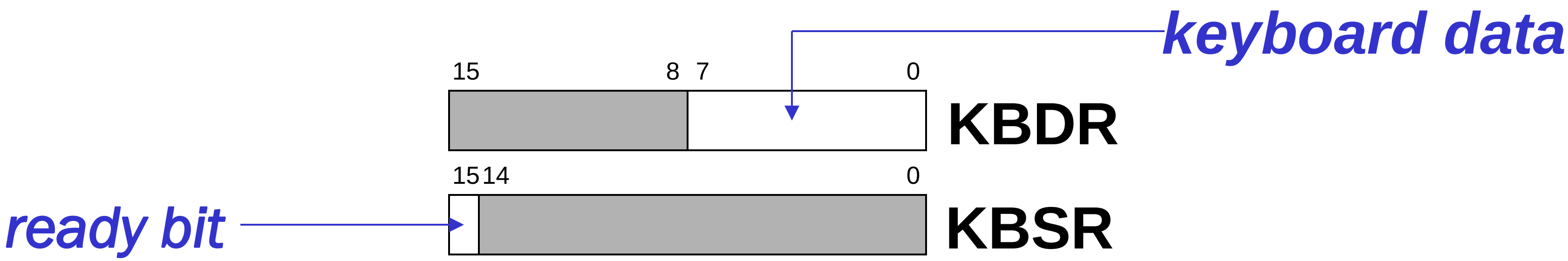
Polling and Interrupts

- **the details of interrupts will be discussed in Chapter 10**

Input from Keyboard

When a character is typed:

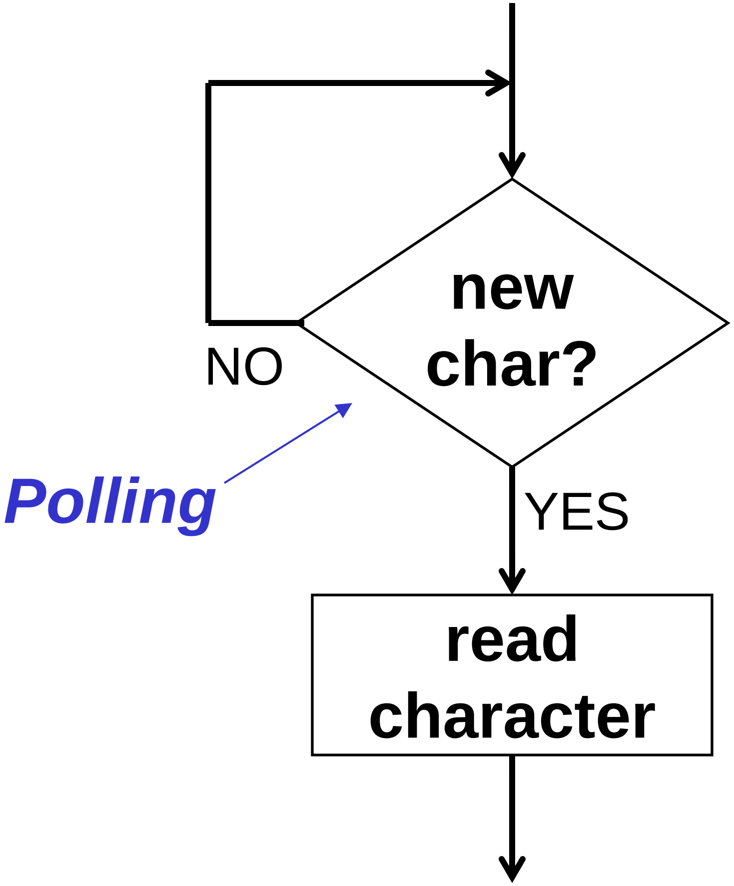
- its ASCII code is placed in bits [7:0] of KBDR (bits [15:8] are always zero)
- the "ready bit" (KBSR[15]) is set to one
- keyboard is disabled -- any typed characters will be ignored



When KBDR is read:

- KBSR[15] is set to zero
- keyboard is enabled

Basic Input Routine

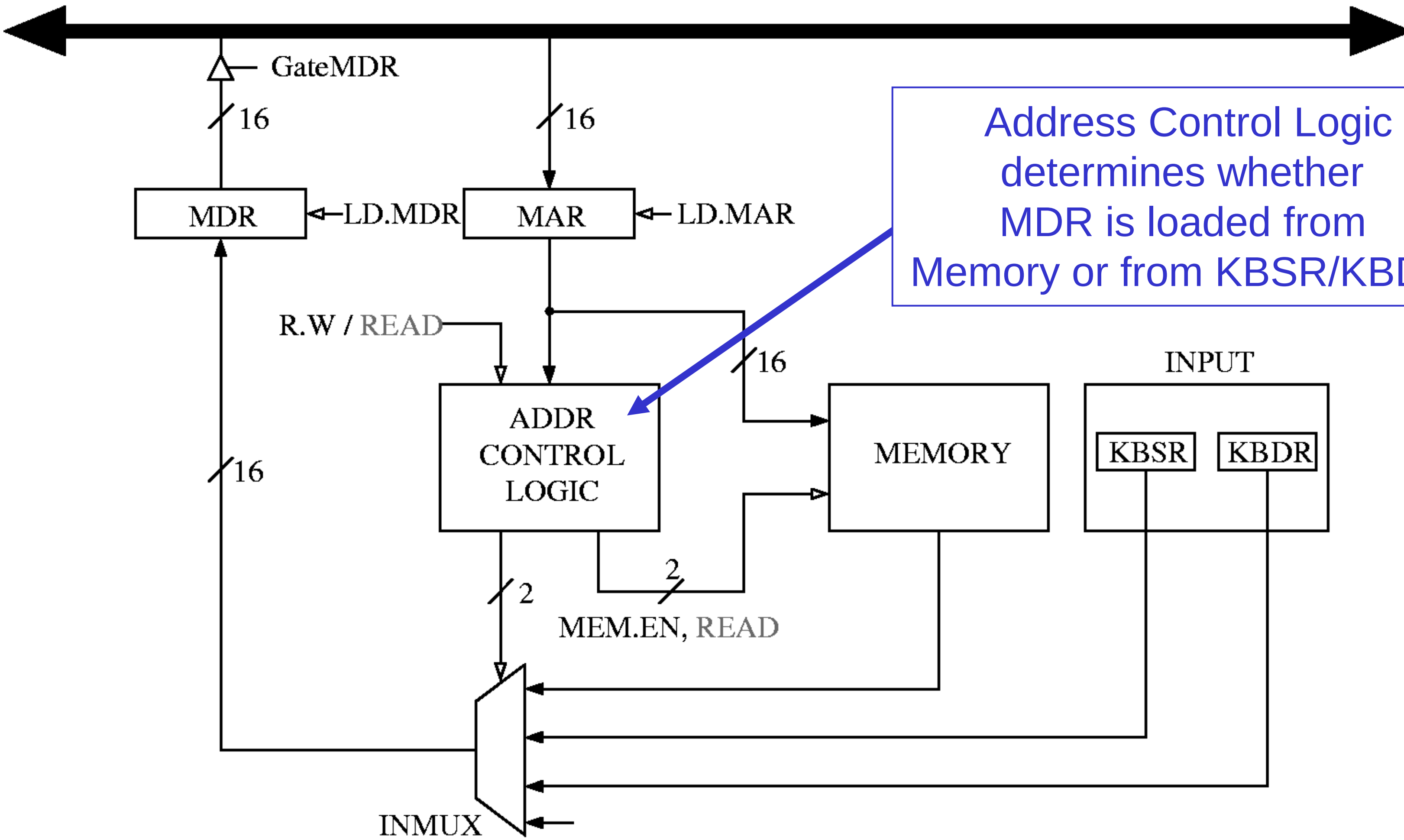



```
POLL      LDI   R0, KBSRPtr
          BRzp  POLL
          LDI   R0, KBDRPtr

          ...

KBSRPtr .FILL xFE00
KBDRPtr .FILL xFE02
```

Simple Implementation: Memory-Mapped Input

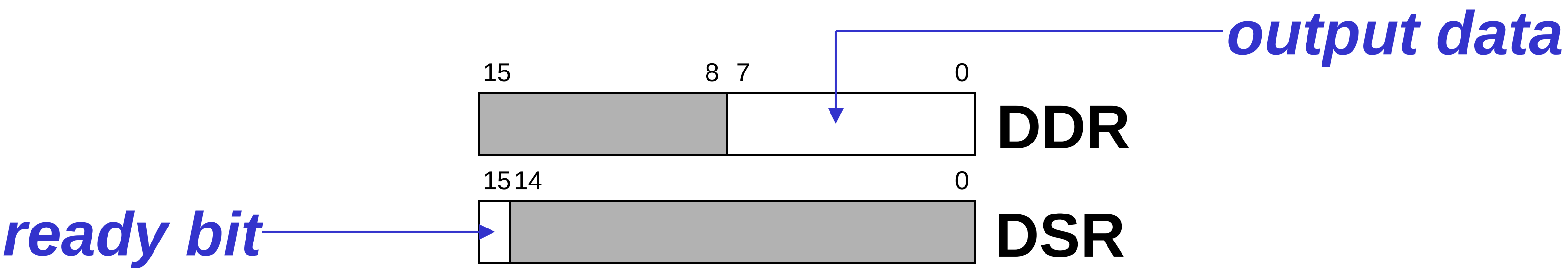


Address Control Logic determines whether MDR is loaded from Memory or from KBSR/KBDR.

Output to Monitor

When Monitor is ready to display another character:

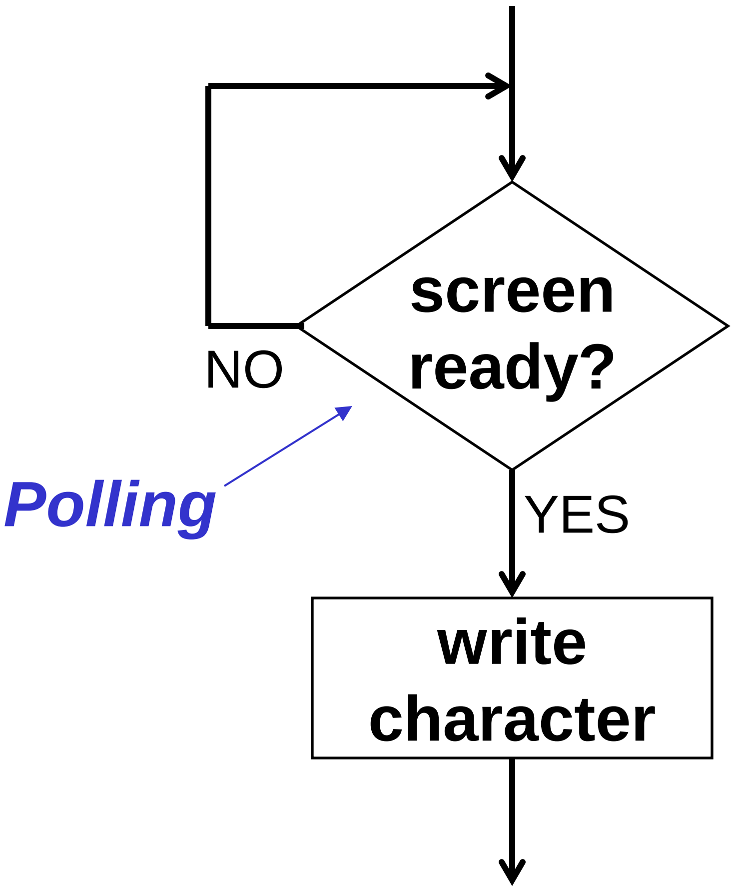
- the “ready bit” (DSR[15]) is set to one



When data is written to Display Data Register:

- DSR[15] is set to zero
- character in DDR[7:0] is displayed
- any other character data written to DDR is ignored (while DSR[15] is zero)

Basic Output Routine

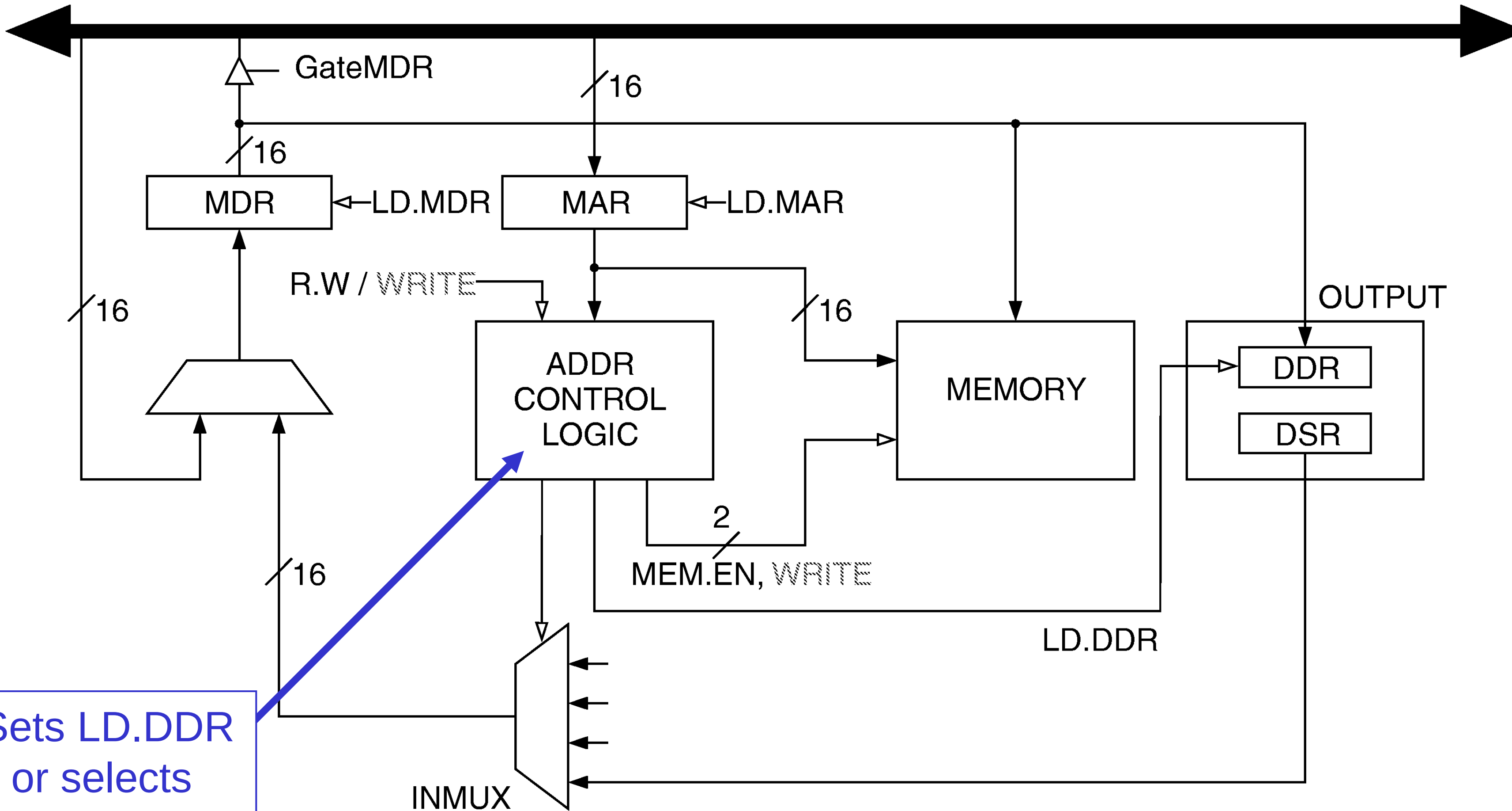



```
POLL    LDI  R1, DSRPtr
        BRzp POLL
        STI  R0, DDRPtr

        ...

DSRPtr .FILL xFE04
DDRPtr .FILL xFE06
```

Simple Implementation: Memory-Mapped Output



Sets LD.DDR or selects DSR as input.

Keyboard Echo Routine

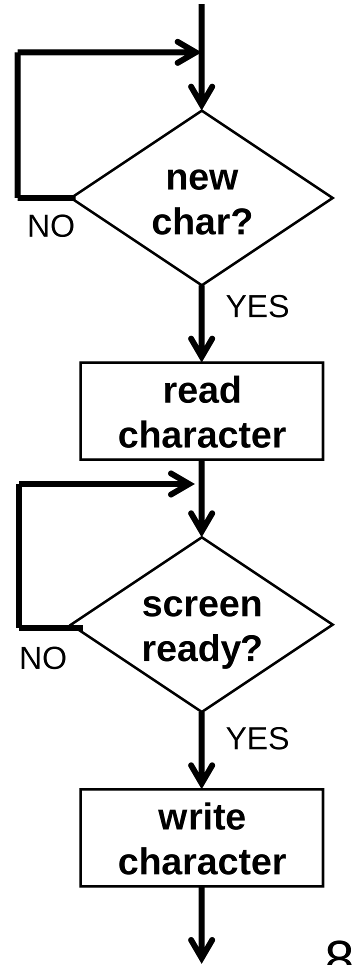
Usually, input character is also printed to screen.

- **User gets feedback on character typed and knows its ok to type the next character.**

```
POLL1    LDI   R0, KBSRPtr
         BRzp  POLL1
         LDI   R0, KBDRPtr
POLL2    LDI   R1, DSRPtr
         BRzp  POLL2
         STI   R0, DDRPtr

         ...

KBSRPtr  .FILL xFE00
KBDRPtr  .FILL xFE02
DSRPtr   .FILL xFE04
DDRPtr   .FILL xFE06
```



Interrupt-Driven I/O

External device can:

(1) Force currently executing program to stop;

(2) Have the processor satisfy the device's needs; and

(3) Resume the stopped program as if nothing happened.

Why?

- **Polling consumes a lot of cycles,
  especially for rare events – these cycles can be used
  for more computation.**
- **Example: Process previous input while collecting
  current input. (See Example 8.1 in text.)**

Interrupt-Driven I/O

Generating the interrupt signal

- The I/O device must want to request service.
- The device must have the right to request service.
- This request must be more urgent than the processor's current task.

Handling the interrupt signal

- We will wait until we understand stacks before getting to this. (Chapter 10)

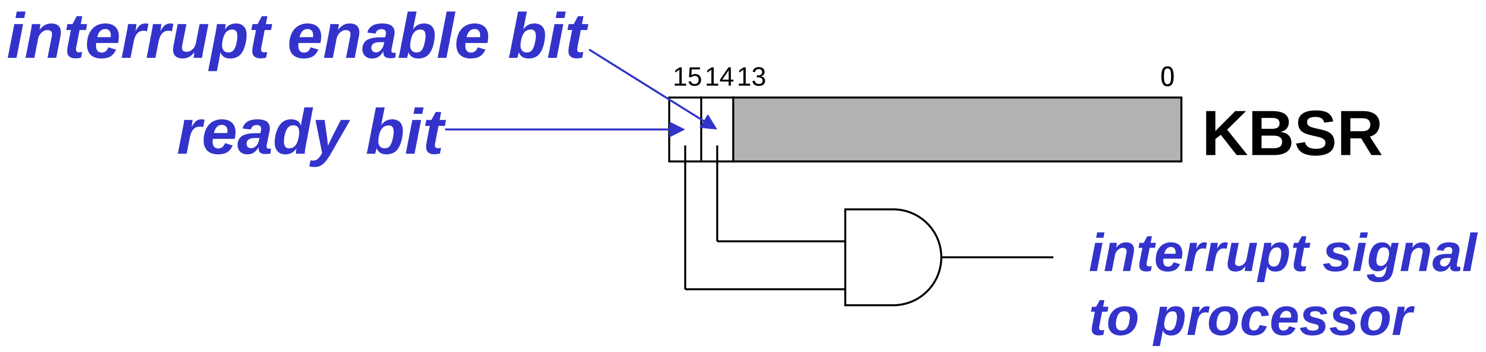
Interrupt-Driven I/O

To implement an interrupt mechanism, we need:

- **A way for the I/O device to signal the CPU that an interesting event has occurred.**
- **A way for the CPU to test whether the interrupt signal is set and whether its priority is higher than the current program.**

Generating Signal

- **Software sets "interrupt enable" bit in device register.**
- **When ready bit is set and IE bit is set, interrupt is signaled.**



Generating the Interrupt Signal

Using the Status Register

- The peripheral sets a Ready bit in SR[15] (as with polling)
- The CPU sets an Interrupt Enable bit in SR[14]
- These two bits are ANDed to set the Interrupt.

  (When ready bit is set and IE bit is set, interrupt is signaled.)

Priority

Every instruction is executed at a stated level of urgency.

LC-3: 8 priority levels (PL0-PL7)

- **Example:**
  - **Payroll program runs at PL0.**
  - **Nuclear power correction program runs at PL6.**
- **It's OK for PL6 device to interrupt PL0 program, but not the other way around.**

Priority encoder selects highest-priority device, compares to current processor priority level, and generates interrupt signal if appropriate.

Priority

Each task has an assigned priority level

- If a higher priority task requests access, a lower priority task will be suspended.

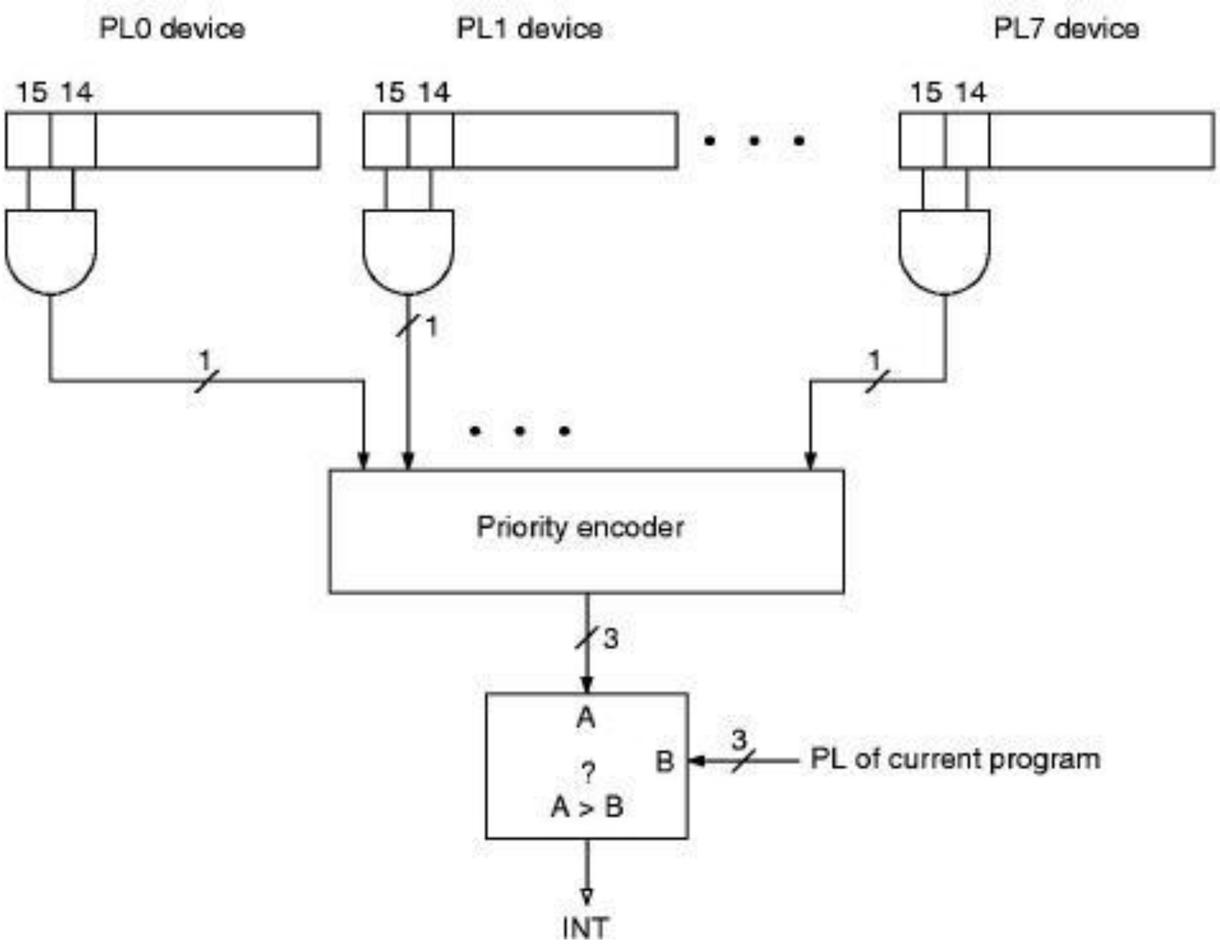
Likewise, each I/O device request has an assigned priority

- The highest priority interrupt is passed on to the CPU only if it has higher priority than the currently executing task.

If an INT is present at the start of the instruction cycle, an extra step is inserted:

- The CPU saves its state information so that it can later return to the current task.
- The PC is loaded with the starting address of the Interrupt Service Routine
- The FETCH phase of the cycle continues as normal.

Generation of the INT Signal

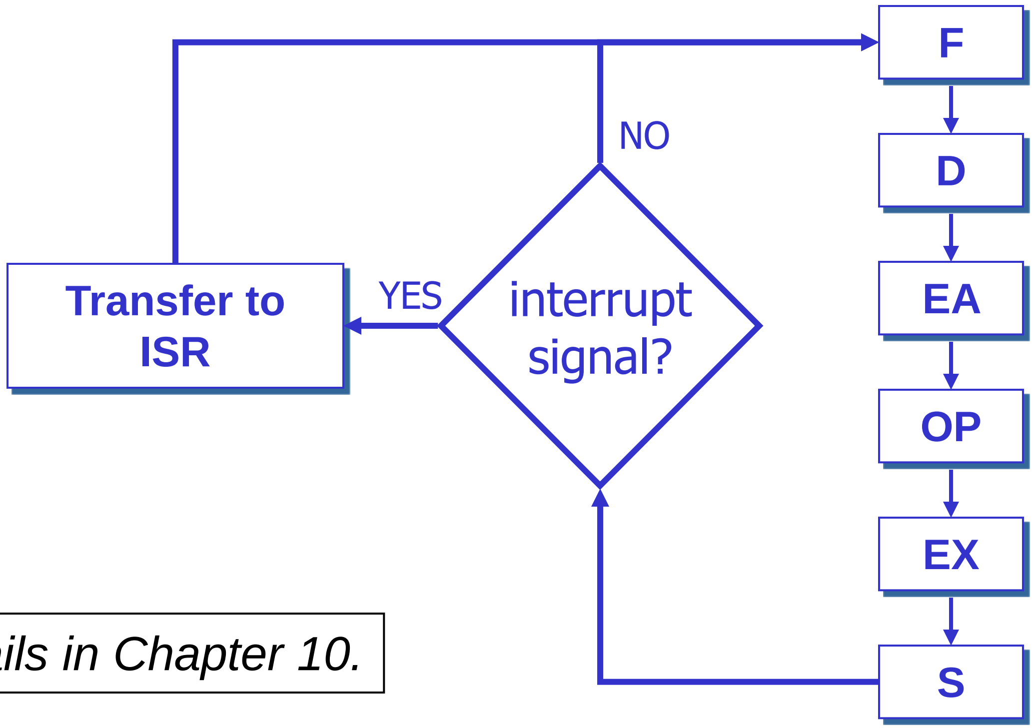


Testing for Interrupt Signal

CPU looks at signal between STORE and FETCH phases.

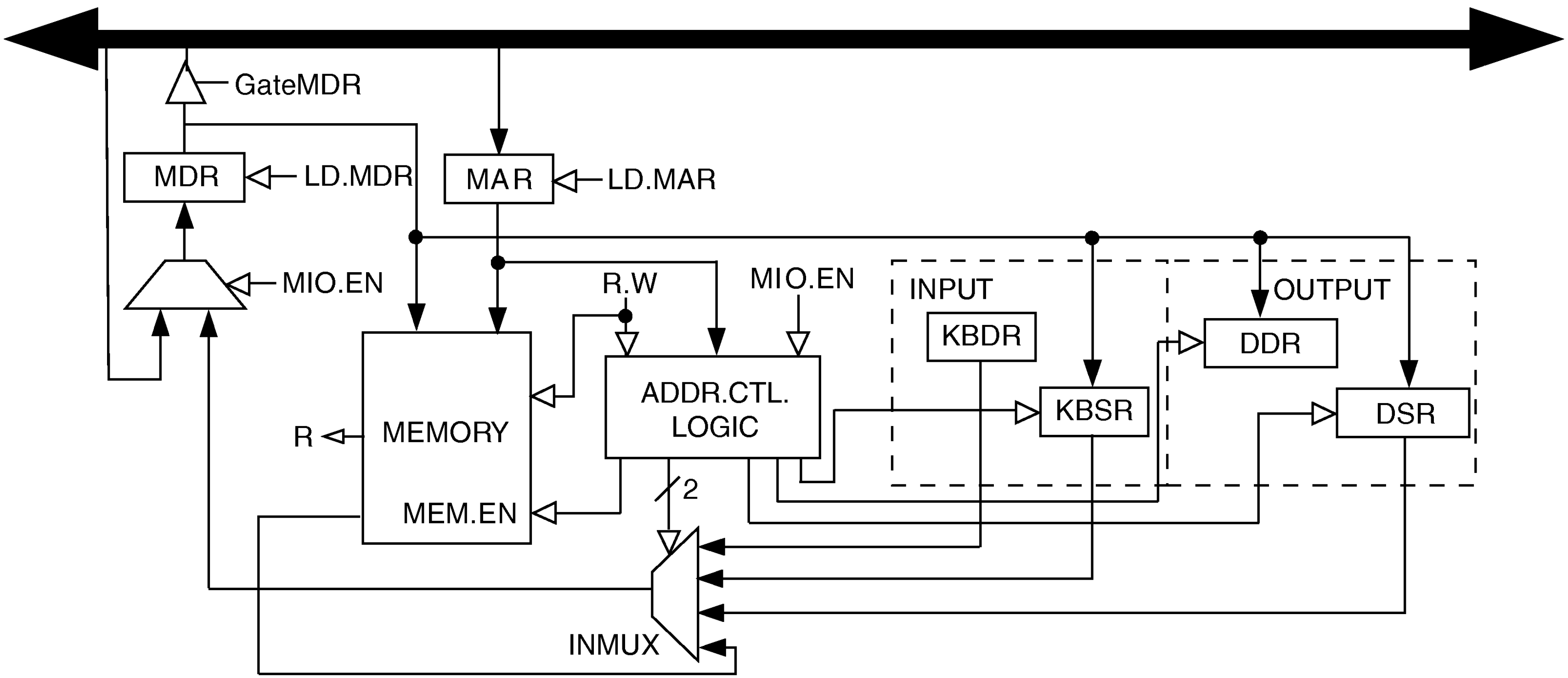
If not set, continues with next instruction.

If set, transfers control to interrupt service routine.



More details in Chapter 10.

Full Implementation of LC-3 Memory-Mapped I/O



Because of interrupt enable bits, status registers (KBSR/DSR) must be written, as well as read.

Review Questions

What is the danger of not testing the DSR before writing data to the screen?

What is the danger of not testing the KBSR before reading data from the keyboard?

What if the Monitor were a synchronous device, e.g., we know that it will be ready 1 microsecond after character is written.

- **Can we avoid polling? How?**
- **What are advantages and disadvantages?**

Review Questions

Do you think polling is a good approach for other devices, such as a disk or a network interface?

What is the advantage of using LDI/STI for accessing device registers?