• CPU accesses physical memory over a bus
• Devices access memory over I/O bus with DMA
• Devices can appear to be a region of memory
Newest CPUs don’t have North Bridge; memory controller integrated into CPU
What is memory?

- **SRAM – Static RAM**
  - Like two NOT gates circularly wired input-to-output
  - 4–6 transistors per bit, actively holds its value
  - Very fast, used to cache slower memory

- **DRAM – Dynamic RAM**
  - A capacitor + gate, holds charge to indicate bit value
  - 1 transistor per bit – extremely dense storage
  - Charge leaks—need slow comparator to decide if bit 1 or 0
  - Must re-write charge after reading, and periodically refresh

- **VRAM – “Video RAM”**
  - Dual ported, can write while someone else reads
What is I/O bus? E.g., PCI
Communicating with a device

- **Device memory** – device may have memory OS can write to directly on other side of I/O bus
- Three communication mechanisms:
  - Memory-mapped IO (MMIO) – Device registers mapped in memory
    - Certain *physical* addresses correspond to device registers
    - Load/store gets status/sends instructions – not real memory
Communicating with a device

- **Device memory** – device may have memory OS can write to directly on other side of I/O bus
- Three communication mechanisms:
  - Memory-mapped IO (MMIO) – Device registers mapped in memory
    - Certain *physical* addresses correspond to device registers
    - Load/store gets status/sends instructions – not real memory
  - Separate I/O Memory – Special I/O Instructions
    - Some CPUs (e.g., x86) have special I/O instructions
    - Like load & store, but asserts special I/O pin on CPU
    - OS can allow user-mode access to I/O ports at byte granularity
Communicating with a device

- **Device memory** – device may have memory OS can write to directly on other side of I/O bus

- Three communication mechanisms:
  - Memory-mapped IO (MMIO) – Device registers mapped in memory
    - Certain *physical* addresses correspond to device registers
    - Load/store gets status/sends instructions – not real memory
  - Separate I/O Memory – Special I/O Instructions
    - Some CPUs (e.g., x86) have special I/O instructions
    - Like load & store, but asserts special I/O pin on CPU
    - OS can allow user-mode access to I/O ports at byte granularity
  - Direct Memory Access (DMA) – Device reads from main memory
    - Typically then need to “poke” device by writing to register
    - Overlaps unrelated computation with moving data over (typically slower than memory) I/O bus
static inline uint8_t
inb (uint16_t port)
{
    uint8_t data;
    asm volatile ("inb %w1, %b0" : "=a" (data) : "Nd" (port));
    return data;
}

static inline void
outb (uint16_t port, uint8_t data)
{
    asm volatile ("outb %b0, %w1" : : "a" (data), "Nd" (port));
}

static inline void
insw (uint16_t port, void *addr, size_t cnt)
{
    asm volatile ("rep insw" : "+D" (addr), "+c" (cnt) :
                  "d" (port) : "memory");
}

::
Example: parallel port (LPT1)

- Simple hardware has three control registers:

<table>
<thead>
<tr>
<th>$D_7$</th>
<th>$D_6$</th>
<th>$D_5$</th>
<th>$D_4$</th>
<th>$D_3$</th>
<th>$D_2$</th>
<th>$D_1$</th>
<th>$D_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

  read/write data register (port 0x378)

<table>
<thead>
<tr>
<th>BSY</th>
<th>ACK</th>
<th>PAP</th>
<th>OFON</th>
<th>ERR</th>
<th>–</th>
<th>–</th>
<th>–</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

  read-only status register (port 0x379)

<table>
<thead>
<tr>
<th>–</th>
<th>–</th>
<th>–</th>
<th>IRQ</th>
<th>DSL</th>
<th>INI</th>
<th>ALF</th>
<th>STR</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

  read/write control register (port 0x37a)

- Every bit except IRQ corresponds to a pin on 25-pin connector:

  ![Diagram of 25-pin connector](image credits: Wikipedia)

  [Messmer]
void 
sendbyte(uint8_t byte) 
{
    /* Wait until BSY bit is 1. */
    while ((inb (0x379) & 0x80) == 0)
        delay ();

    /* Put the byte we wish to send on pins D7-0. */
    outb (0x378, byte);

    /* Pulse STR (strobe) line to inform the printer 
     * that a byte is available */
    uint8_t ctrlval = inb (0x37a);
    outb (0x37a, ctrlval | 0x01);
    delay ();
    outb (0x37a, ctrlval);
}
Anatomy of a disk [Ruemmler]

- **Array of blocks of persistent data**
  - Blocks are typically 512 Bytes or 4 kiB in size
  - OS can read or write a multiple of block size
  - Writes to sectors are atomic (or they want you to believe)

- **Stack of magnetic platters**
  - Rotate together on a central spindle @3,600-15,000 RPM
  - Drive speed drifts slowly over time
  - Can’t predict rotational position after 100-200 revolutions

- **Disk arm assembly**
  - Arms rotate around pivot, all move together
  - Pivot offers some resistance to linear shocks
  - One disk head per recording surface (2 × platters)
  - Sensitive to motion and vibration [Gregg] (demo on youtube)
Disk
Disk
void IDE_ReadSector(int disk, int off, void *buf)
{
    outb(0x1F6, disk == 0 ? 0xE0 : 0xF0); // Select Drive
    IDEWait();
    outb(0x1F2, 1); // Read length (1 sector = 512 B)
    outb(0x1F3, off); // LBA low
    outb(0x1F4, off >> 8); // LBA mid
    outb(0x1F5, off >> 16); // LBA high
    outb(0x1F7, 0x20); // Read command
    insw(0x1F0, buf, 256); // Read 256 words
}

void IDEWait()
{
    // Discard status 4 times
    inb(0x1F7); inb(0x1F7);
    inb(0x1F7); inb(0x1F7);
    // Wait for status BUSY flag to clear
    while ((inb(0x1F7) & 0x80) != 0)
    
};
Memory-mapped IO

- **in/out instructions slow and clunky**
  - Instruction format restricts what registers you can use
  - Only allows $2^{16}$ different port numbers
  - Per-port access control turns out not to be useful (any port access allows you to disable all interrupts)

- Devices can achieve same effect with physical addresses, e.g.:

  ```c
  volatile int32_t *device_control = (int32_t *) (0xc0100 + PHYS_BASE);
  *device_control = 0x80;
  int32_t status = *device_control;
  ```

  - OS must map physical to virtual addresses, ensure non-cachable

- **Assign physical addresses at boot to avoid conflicts. PCI:**
  - Slow/clunky way to access configuration registers on device
  - Use that to assign ranges of physical addresses to device
DMAX buffers

- Idea: only use CPU to transfer control requests, not data
- Include list of buffer locations in main memory
  - Device reads list and accesses buffers through DMA
  - Descriptions sometimes allow for scatter/gather I/O
Example: Network Interface Card

- Link interface talks to wire/fiber/antenna
  - Typically does framing, link-layer CRC
- FIFOs on card provide small amount of buffering
- Bus interface logic uses DMA to move packets to and from buffers in main memory
Example: IDE disk read w. DMA

1. device driver is told to transfer disk data to buffer at address X
2. device driver tells disk controller to transfer C bytes from disk to buffer at address X
3. disk controller initiates DMA transfer
4. disk controller sends each byte to DMA controller
5. DMA controller transfers bytes to buffer X, increasing memory address and decreasing C until C = 0
6. when C = 0, DMA interrupts CPU to signal transfer completion
Driver architecture

- Device driver provides several entry points to kernel
  - Reset, ioctl, output, interrupt, read, write, strategy ...

- How should driver synchronize with card?
  - E.g., Need to know when transmit buffers free or packets arrive
  - Need to know when disk request complete

- One approach: *Polling*
  - Sent a packet? Loop asking card when buffer is free
  - Waiting to receive? Keep asking card if it has packet
  - Disk I/O? Keep looping until disk ready bit set

- Disadvantages of polling?
Driver architecture

- Device driver provides several entry points to kernel
  - Reset, ioctl, output, interrupt, read, write, strategy ... 

- How should driver synchronize with card?
  - E.g., Need to know when transmit buffers free or packets arrive
  - Need to know when disk request complete

- One approach: **Polling**
  - Sent a packet? Loop asking card when buffer is free
  - Waiting to receive? Keep asking card if it has packet
  - Disk I/O? Keep looping until disk ready bit set

- Disadvantages of polling?
  - Can’t use CPU for anything else while polling 
  - Schedule poll in future? High latency to receive packet or process disk block bad for response time
Interrupt driven devices

- Instead, ask card to interrupt CPU on events
  - Interrupt handler runs at high priority
  - Asks card what happened (xmit buffer free, new packet)
  - This is what most general-purpose OSes do

- Bad under high network packet arrival rate
  - Packets can arrive faster than OS can process them
  - Interrupts are very expensive (context switch)
  - Interrupt handlers have high priority
  - In worst case, can spend 100% of time in interrupt handler and never make any progress – *receive livelock*
  - Best: Adaptive switching between interrupts and polling

- Very good for disk requests