Computer Architecture Review

CS 562

The von Neumann Model

John von Neumann (1946) proposed that a fundamental model of a computer should include **5 primary components**:

- Memory
- Processing Unit
- Input Device(s)
- Output Device(s)
- Control Unit

Memory

- For our purposes, an array of bytes. We will **not** be dealing with virtual memory (yet)

- **Recall:** When we talk about memory, ***addressability*** refers to the size of a memory location (the thing that goes in and comes out of memory)

- **Recall**: When we talk about ***address width***, we mean how many bits are required to represent an address

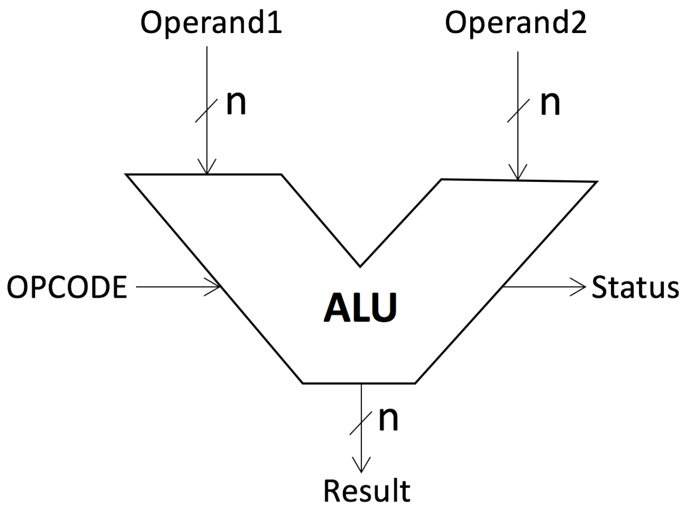
- Ex: recent x86-64 machines have 64-bit address width and are *byte addressable*

Memory Cont.

- How do we access?
- Loads and Stores
- Accomplished with the help of memory unit (MMU) on the CPU. Simplest scheme has two registers:
- **MAR**: memory address register (address from which to load, to which to store)
- **MDR**: memory data register (stuff to store)

Processing Unit

- Can consist of *many* separate functional units (integer arithmetic, floating point, vector units, DSP, etc. etc.)
- Simplest one is the ALU (arithmetic logic unit)
- Size of data worked on by ALU is the CPU's *word length*
- Today we call this the **datapath** of the processor



Proc. Unit. contd.

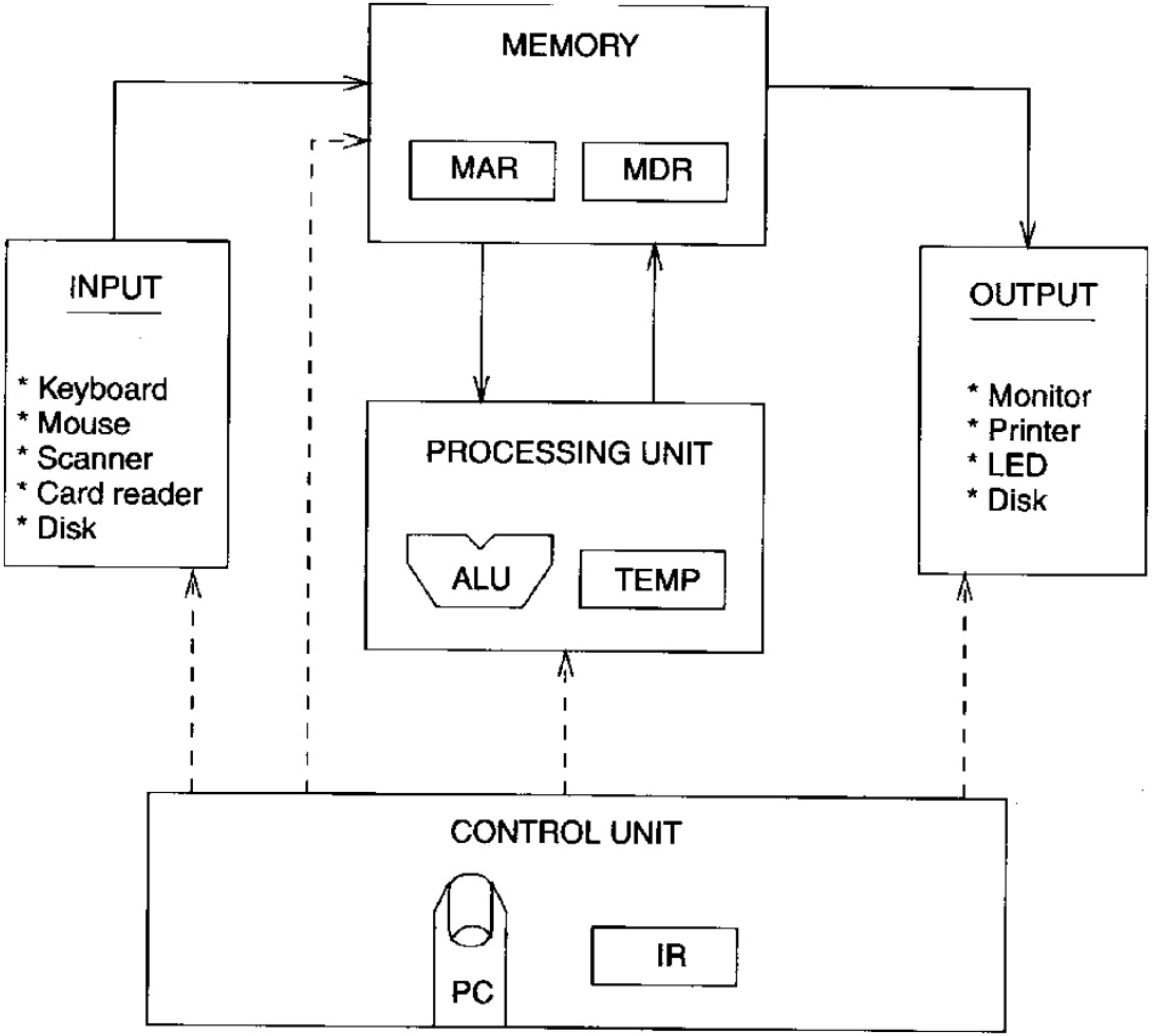
- Temp. Storage: most commonly registers (these are fast access, close to functional units)
- May also be stack (more on this later)

I/O

- The peripherals attached to the machine:
- keyboard, mouse, video card, monitor, disk, etc. etc.
- Two methods of I/O: polling (CPU busy waits until something is read) or interrupt-driven (device raises a wire hot to notify CPU)
- You will become very familiar with the latter

Control Unit

- Keeps track of where we are in the program, where to go next
- Where we are: Instruction Register (IR) . Register which holds the currently executing instruction
- Where to go next: Instruction Pointer (IP). Memory address of next instruction to execute. (Also called program counter or PC).
- Finite State Machine: Given current inputs and current instruction, where do we go next? Essentially implemented as a lookup table. You will implement as logic in C. ?? Isn’t it always just IP + 1?
- Controls signals to the datapath (e.g. which arithmetic op should ALU perform, what is the sequence of operations of the various regs?)


MEMORY
MAR
MDR
INPUT
* Keyboard
* Mouse
* Scanner
* Card reader
* Disk
OUTPUT
* Monitor
* Printer
* LED
* Disk
PROCESSING UNIT
ALU
TEMP
CONTROL UNIT
PC
IR

ISAs

- *Instruction Set Architecture*
- The interface to the hardware from the programmer’s point of view
- Also, the boundary between software and hardware

Classes of ISAs

- **Load-Store machines**: Can access memory *only* with explicit load or store operations (e.g. MIPS, RISC-V, ARM, PowerPC, SPARC, LC-3)

- **Register-Memory machines**: Can access memory in other types of operations as well (e.g. x86)
  - `addl $0x7, 8(%rax)`

- **Stack Machines**: All operations are performed via a LIFO stack (e.g. JVM, WebASM, Forth, PostScript,)

The Instruction Cycle

- For our purposes, instruction dispatch will essentially occur in three stages:
    - Fetch
    - Decode
    - Execute
- Real hardware involves (in some cases many) **more stages**

Let’s design a simple ISA

- Assume 16-bit address width and addressability
- That means 2^16 mem locations, 65K (just like the 6502)
- Assume 8 General-purpose registers (GPRs)
- OK, what does our instruction set look like?

Things we had to consider

- Instruction length (variable/fixed?) and encoding
- Operand encoding/size
- Memory addressing modes (immediate, PC-relative, base-index, maybe SID)
- Operations we support (arithmetic, logical, control flow)
- Supporting conditional operations

Memory Mapped I/O

- Instead of using processor pins for I/O devices, just have devices respond to special regions of memory addresses
- Old machines had hardcoded regions. These days the regions are programmable, e.g. with the PCI, PCIe device standard specification
- Ex: LD R1, $0xa700
- Ex: ST R2, $0xa720

simple memory controller logic

```
word_t read(addr) {
  if (addr >= 0xa700 && addr < 0xb700) {
      return my_device_read(addr);
  } else {
      return ram_read(addr);
  }
}
```