CS201 W'00 Lecture Notes
Lecture 1

CS201 Computer Architecture

Goal: Understand organization of computer systems at **multiple levels**.

Topics:

- Hardware-Software Interface
- Comparative Instruction Set Architecture
- Processor Implementation
- Memory Hierarchy
- IO Systems
- Parallelism

Format: lectures, homeworks, labs, exams.

Multi-layered Systems

Typical Levels:

6. Components/Libraries

5. "High-level" Languages

4. Assembly Language

3. Operating System (OS) Services

2. Instruction Set Architecture (ISA)

1. Microarchitecture

0. Digital Logic (Gates)

-1. Devices (Switches)

-2. Solid-state Physics

Most levels can be seen as **virtual machines**, each with its own machine **language**.

Many levels can be realized in either **hardware** or **software**

Mapping between levels can be done by **translation** or by **interpretation**.

Example: the rise and fall (and rise) of microcode.

Major Computer Hardware Technologies

Mechanical (pre-1945)

- Babbage Analytical Engine, Aiken Mark I

Vacuum Tubes (1945-55)

- ENIAC, EDVAC, IAS, UNIVAC, IBM 701 etc.

Transistors (1955-65)

- DEC PDP-1, PDP-8, IBM 7090, 1401, CC 6600, etc.

Integrated Circuits (ICs) (1965-80)

- IBM 360, DEC PDP-11, etc.

Very-Large Scale Integrated Circuits (VLSI) (1980-)

- microprocessors; all current machines

Quantum Computing (?)

Modern VLSI

Modern electronic computers use **transistors** to implement logic **gates** (e.g. NAND, NOR).

In VLSI, a single piece of silicon is imprinted with many transistors (currently millions).

Silicon **wafers** are printed with circuits at very fine resolution (e.g., microns) Wafers are then cut into **chips** (e.g., hundreds per wafer).

Wafer size and cost are roughly constant, so limiting chip **size** is key to controlling cost.

Big issue in chip design: **managing "real-estate"**, i.e., deciding what is best use for chip area.

Fortunately, huge & steady improvements in manufacturing techniques have consistently increased number of gates that can fit in a given chip area, and the speed at which they run.

Moore's "Law" (actually just an empirical observation) says that the number of transistors that will fit on a (memory) chip grows at 60% per year. Speed increases as size decreases. Also seems to hold for microprocessors.

So can make chips that are more powerful for the same price or equally powerful but much cheaper.

Modern VLSI design closely resembles software design: huge complexity must be managed using software tools.

Tanenbaum's Example Systems

Pentium II

- Latest member of Intel x86 family; descendent of original 16-bit 8086 microprocessor (1978), and 32-bit 80386 (1985).

- Characteristic "CISC" (Complex instruction set computer).

- Family members dominate PC market.

- Most machines run some form of Windows O/S, but also supports Unix.

Ultra SPARC II

- Latest member of Sun's SPARC family; 64-bit, but compatible with original 32-bit SPARCs (1987).

- Characteristic "RISC" (Reduced instruction set computer).

- Popular in workstation and server market.

- Normally run Unix.

picoJava II

- (Design for) hardware implementation of Java Virtual Machine.
- Stack-oriented machine executing byte code instructions.
- Targeted at embedded system market.
- Runs only Java.

All provide examples of architecture **families**: fixed instruction set with wide variety of implementations offering different price/performance.

Processor Organization

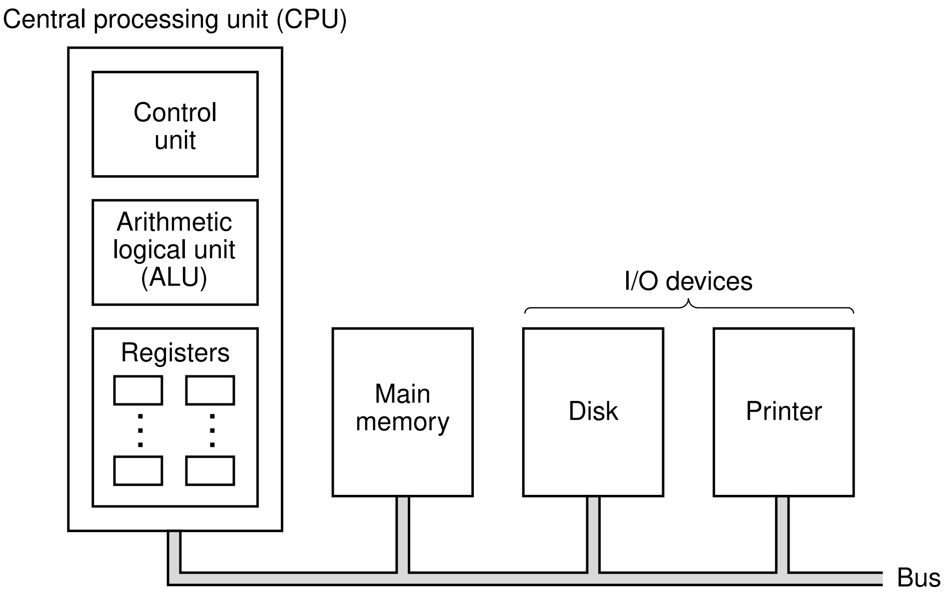

Figure 2-1. The organization of a simple computer with one CPU and two I/O devices.

Key components:

- Central Processing Unit (CPU)
- Memory
- I/O Devices (e.g., disks, printers, network interfaces, etc.)
- Bus(es)

CPU contains:

- Control unit
- Arithmetic Logic Unit (ALU)
- Registers: data and control, including Program Counter (PC).

Datapath

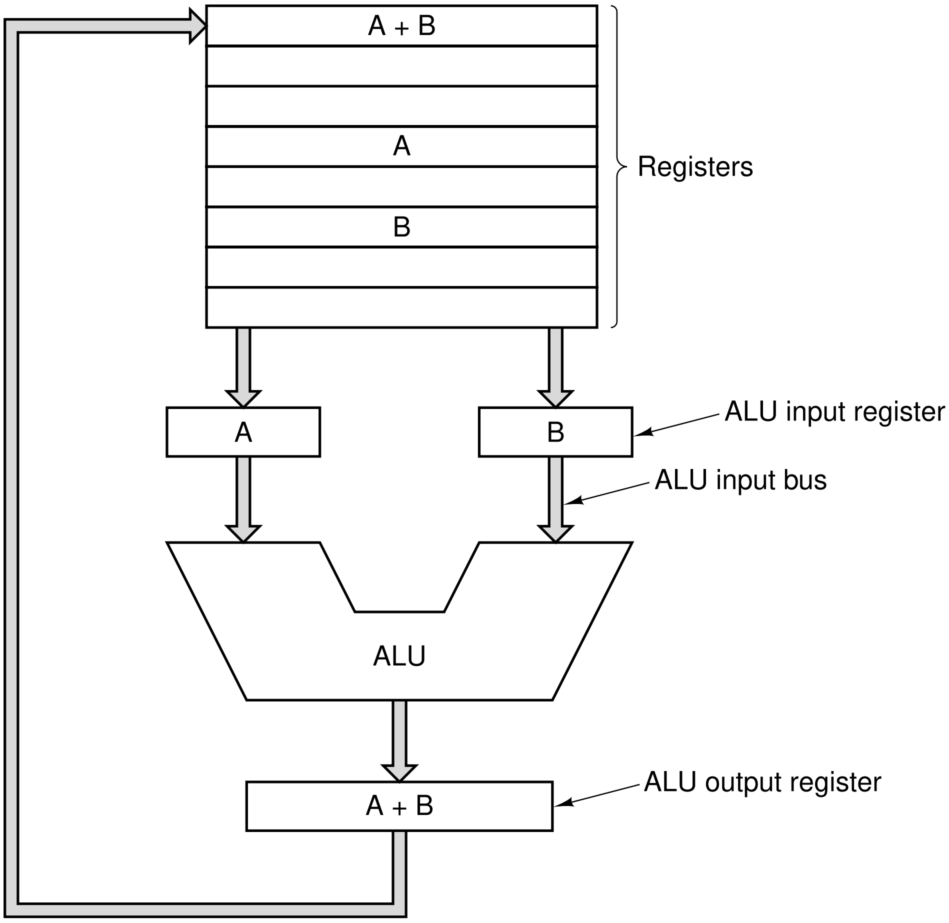

Figure 2-2. The data path of a typical von Neumann machine.

Datapath = Registers + ALU

Register-register instructions fetch operands from registers, perform ALU operation, and store result in register.

Register-memory load value from memory to register or store value from register to memory.

Some instructions may do both.

Execution: the fetch-decode-execute cycle

Basic steps of executing a machine-language program are as follows:

1. Fetch instruction at current PC from memory into instruction register (IR)

2. Update PC to point to next instruction.

3. Decode IR to determine type of instruction.

4. If memory-to-register instruction, determine memory address and load value from that address into register.

5. Execute the register-register (arithmetic/logic) content of the instruction.

6. If register-to-memory instruction, determine memory address and store value from register into that address.

7. Go back to step 1.

Modern machine designs typically do just one of (4, 5, 6) on a given instruction.

In most machines, I/O instructions look like memory-register instructions.

Machine essentially **interprets** machine language program.

Performance

Key goal of processor design: providing best possible performance for lowest price.

(Note: should distinguish **response time** from **throughput**. But often they go together.)

Execution cycle time dominates overall performance of processor, so most straightforward way to improve performance is to **shorten cycle time**.

Caveat 1: One way to shorten cycle time is to use simpler instructions which are quicker to execute. But this may not speed up real programs, because they may require more instructions to do the same task.

We really want to minimize:

$$\text{Execution time} = \frac{\text{instructions}}{\text{program}} \times \frac{\text{cycles}}{\text{instruction}} \times \frac{\text{seconds}}{\text{cycle}}$$

Caveat 2: Speeding up the processor won't help too much if other aspects of the system remain slow.

Formalized as **Amdahl's Law**:

$$\begin{aligned}&\text{Time after improvement} =\\ &\quad \left(\frac{\text{Original time of improved part}}{\text{Improvement factor}}\right)\\ &\quad + \text{Original time of unimproved part}\end{aligned}$$

Parallelism

Limits on decreasing cycle time are largely set by underlying technology.

Alternative approach: exploit **parallelism**, i.e., do several parts of the computation at once.

Parallelism can be applied at several levels: within the processor implementation, or by providing multiple processors.

Important kinds of parallelism include:

- **Pipelining**. Can often overlap the steps in the instruction cycle sequence, so that, e.g., we fetch the next instruction while still executing the arithmetic content of the previous one.

- **Instruction-level parallelism (ILP)**. By duplicating hardware (e.g., having multiple ALU's), can do the arithmetic (or other) content of several instructions simultaneously, as in so-called **superscalar** machines.

- **Multiprocessors**. Place multiple complete CPU's on a single bus or network, and set them to working on the problem simultaneously. Normally requires explicit programming.