**CPS311 Lecture: Course Introduction; The Levels of Computer Structure; Architecture and Organization**

Last revised August 3, 2015

*Objectives:*

1. Introduce course, requirements
2. Briefly introduce binary representations for numbers - to be covered in detail later
3. Overview levels of structure of a "real" computer
4. Introduce concepts of architecture and organization
5. Introduce the term "instruction set architecture"

*Materials:*

1. Technology Samples

## I. Preliminaries: Roll, Syllabus

## II. The Concept of Levels of Computer Structure

A. Of course, a major concern of Computer science is understanding and designing computer systems: systems of hardware and software which work together to meet a particular need.

B. Many writers have observed that computer systems (hardware and software) are the most complex engineering artifacts ever developed by man.

1. In proof of this, note that we are shocked when an engineered system such as a bridge fails. But we are not surprised when a computer system crashes.

a) Why?

b) We have learned how to build bridges that are reliable. But computer systems are of such a level of complexity that we still don't know how to master them.

2. Discovering how to master this complexity is one of the most important challenges of the discipline of computer science/ engineering.

3. One of the key concepts that helps in mastering complexity is the use of HIERARCHIES OF ABSTRACTION.

   a) You have met this concept already in programming. A complex program is first designed in terms of a group of interacting objects, each of which is then, in turn, developed in detail.

   b) From a broader perspective, we know that computer users see a computer system as sophisticated tool to perform a certain task, such as word-processing. He/she usually does not care about the details of how it carries out this task.

   c) However, we realize that each software application is realized by a program consisting of a series of individual statements written in a language like C or C++ or Java.

   d) At a lower level, each statement is translated by the compiler into one or more machine instructions, each of which is then executed by the underlying hardware.

   e) The hardware is, in turn, realized as a set of fairly complex chips, A typical chip consists of millions of interconnected basic elements (called gates, flip flops, memory cells, etc) which are fabricated from silicon, and which rely utimately on laws derived from solid state physics

C. In talking about complex computer systems, then it is desirable to utilize a hierarchy of levels of abstraction. I will present here a listing of five levels of abstraction (These are different from the ones in the book, but the basic idea is the same.) At each level, the underlying layers work together to present a particular "view" or interface, and the layer responds to a particular language.

1. The user level: the computer system performs certain tasks in response to certain commands (e.g. the command to edit a file or display a web page or compile a program). To the user, it appears as if the system "understands" a command language such as the shell command language of a Unix system, or html, or the mouse clicks of a graphical interface.

2. The higher-level language programming level: each application is programmed using the statements of a higher-level language such as C or C++ or Java. A single user-level command is thus implemented by 100's or 1000's of statements in a programming language. To the programmer, it appears as if the system "understands" the particular higher-level language he or she is programming in.

3. The machine language programming level: as delivered by the manufacturer, a given computer system has certain primitive components and capabilities:

   a) A memory system, capable of storing and retrieving information in fixed-size units known as “bytes” or “words”.

   b) An input-output system, capable of transferring information between memory and some number of devices such as keyboards, screens, disks etc.

   c) A CPU, capable of performing primative operations such as addition, subtraction, comparison, etc., and also capable of controlling the other two systems.

      (1) The CPU is designed to respond to a set of basic machine language instructions, which is specific to a given type of CPU. (E.g. the machine language for the MIPS architecture we will study is vastly different from that of the Pentium used in most desktop and laptop machines.)

The differences between different machine languages are comparable in magnitude to the differences between human languages such as English and Hebrew (which use different alphabets) - though obviously machine languages are much smaller!

(2) The compiler for a higher level language translates that into the native machine language of the underlying machine.

(a) The same program must be translated into different machine languages to run on different machines; thus, each type of machine must have its own set of compilers.

(b) Regardless of the HLL used, the machine code generated by the compiler for a given machine will be in the same native machine language of that machine.

(c) Example: on our workstations, the .o and executable files produced by the compiler and linker contain two different forms of machine language binary code.

At this level, it appears that the system "understands" its machine language.

4. The hardware design level: Ultimately, computer systems are built as interconnections of hardware devices known as gates, flip-flops, etc., combined to form registers and busses. These, in turn, are realized from primitive electronic building blocks known as transistors, resistors, capacitors etc. The resultant system is capable of directly executing the instructions comprising the machine language of the system.

5. The solid-state physics level: current computers are fabricated from materials such as silicon that have been chemically “doped” to alter their electronic properties. Transistors, resistors, and capacitors are realized by utilizing the properties of these semiconductor materials. (Of course, future computers may use some other technology such as optics.)

Summary:

| | |
|---|---|
| User Level | User commands, Application software |
| HLL Programming level | Statements in C, C++, Java, etc |
| Machine language level | Machine language instructions |
| Hardware design level | Gates, flip-flops etc. |
| Solid-state physics | Physical properties of semiconductors |

D. We should note that these levels are not fixed and rigid - for example

1. Some user level software includes a facility that allows advanced users to write programs - e.g. macros in spreadsheets, Unix shell scripts, JavaScript, etc.

2. Some computers have been built whose machine language is explicitly designed to support a particular HLL - e.g. the LISP machine, or the PicoJava machine.

3. The partitioning of functions between hardware and machine language code sometimes varies between different computers in the same family - e.g. at one point some machines had hardware to perform floating point arithmetic and others used machine language software for this.

   One text cites "The principle of equivalence of hardware and software" which states that "Anything that can be done with software can also be done with hardware, and anything that can be done with hardware can also be done with software". (However, doing something in hardware is almost always much faster, but also more complex - which leads to a tradeoff.)

E. Nonetheless, these levels are helpful tools for understanding computer systems.

F. Your previous coursework has focused on the first two levels. In this course, we will look at the third and fourth levels, perhaps (time permitting) with a glance at the lowest one.

1. However, it is important to realize that this course is not at all intended to enable you to actually design and build hardware systems. That's a separate field (called computer engineering), and would call for a lot more than one course. Rather, this course is intended to give you a better understanding of the hardware platforms on which software systems operate.

2. The first section in the book talks about "building a microprocessor". Actually, that's way beyond the scope of either this course or the book. But what we will learn will help you to understand the design of a microprocessor!

## III. **A Bit of History**

A. There are few (if any) fields of study that undergo change as rapidly as Computer Science and related disciplines.

It is interesting, for example, to consider changes that have occurred over the span of your lifetimes.

1. Many of these changes are <u>quantitative</u> in nature - e.g.

   a) Computer systems of 25 years ago had clock speeds on the order of a few MHz. Current computer systems typically run at 2-3 GHz - a nearly 1000:1 change in 2-1/2 decades.

   b) Personal computer systems of 25 years ago had main memory system (RAM) capacities of hundreds of thousands of bytes up to a few megabytes. Comparable systems today generally have memory capacities on the order of several gigabytes - another 1000:1 change in 2-1/2 decades.

   (In fact, when I revise lecture notes for each new offering of this course, one thing I end up having to do is multiply many of

the numbers by 2 or 4 - though for the last three times, I didn't have to change the CPU speed numbers at all, since this has plateaued and the new frontier is multicore processors.)

2. Important software developments likewise occur with a rapid pace - e.g. Google, Facebook, Twitter ... and the World-Wide Web is about the same age as you are.

B. In the midst of this rapid change, it is interesting to think about what hasn't changed. One such thing - which will be the focus of this course - is the overall architecture of a computer system. (Though the details have changed dramatically, the overall structure has not.)

C. Most computers are based on an architecture proposed by Jon Von Neumann in a paper written 1946 entitled "Preliminary Discussion of the Logical Design of an Electronic Computing Instrument".

1. There is a long line of development which led up to this proposal, starting with Pascal's calculator and progressing through Babbage's analytical engine and various machines built in the late 1930's and 1940's, and including the theoretical work done by Turing and others.

2. Von Neumann's paper clearly built on this previous work, but contained two proposals that were especially important:

   a) The use of the binary system for representing numbers internally (as opposed to the arbitrary alphabets of abstract automata or various decimal schemes used in earlier actual computers).

      (1) That is, the alphabet of a Von Neumann machine consists of the set { 0, 1 }. More complex information is represented by strings of these symbols - e.g. the letter 'A' is represented by 01000001 on most computers.

      (2) We will cover binary representation of information in detail later in the course, but for now we can note that with this

representation it is possible to represent any non-negative integer easily by using a place value system - e.g. the bit string 101010 can represent the decimal number 42 by interpreting it as

$$
\begin{array}{rr}
1 \times 2^5 = & 32 \\
+\ 0 \times 2^4 = & 0 \\
+\ 1 \times 2^3 = & 8 \\
+\ 0 \times 2^2 = & 0 \\
+\ 1 \times 2^1 = & 2 \\
+\ 0 \times 2^0 = & 0 \\
 & -- \\
 & 42
\end{array}
$$

(3) Binary representation facilitates the construction of robust computing machines, because there are many physical systems that are BISTABLE (have two stable states) - e.g.

(a) Electrical switches or transistorized equivalents (on - conducting, off - not conducting)

(b) Magnetic media (magnetized in one direction or the other)

(c) Dynamic RAM - presence or absence of electrical charge

b) The stored program concept (in contrast to the hardwired transition tables of abstract automata or the use of plugboards, punched cards or tape, or the like in earlier actual computers). This is the idea that a single linearly addressable memory might be used to hold both the program that controls the computation and the data the program manipulates.

(1) Von Neumann machines utilize random access memories - in which any cell is equally accessible at any time. This contrasts with the tape of the Turing machine or the stack of the Push-Down-Automaton.

(a) Each cell in the memory holds a finite, fixed number of bits (called the word size of the machine), normally interpreted as representing a binary integer (though other interpretations are possible depending on the context.)

(b) Each cell in the memory has a distinct ADDRESS, which is an integer in the range 0 .. (memory size) - 1. The range of permissible addresses is called the ADDRESS SPACE.

(2) In addition to their random access memories, computers based on the VonNeumann architecture have one or more special memory cells called REGISTERS. The number of registers is usually small - generally much less than 100.

(a) Instead of having addresses, registers have names, specified as part of the machine's architecture.

(b) A register is typically implemented using a technology that allows faster access to the data it contains than regular memory allows. (On modern computers, perhaps as much as 100 times or more faster.)

(c) One register (typically called the INSTRUCTION REGISTER (IR)) holds the instruction currently being interpreted.

(d) Another register (typically called the PROGRAM COUNTER (PC)) holds the address of the memory cell (or beginning of a group of memory cells) holding the NEXT instruction to be executed.

(e) Many instructions also use or alter one or more other registers.

(3) Von Neumann style computers fetch and interpret instructions (which are bit strings) - usually from successive locations in memory. One part of each instruction is an operation code (op-code) which specifies which instruction (from a fixed repertoire) the machine is to perform. An instruction may also contain addresses of one or more

locations in memory from which the operands are to be fetched. All instructions make use of some of the registers.

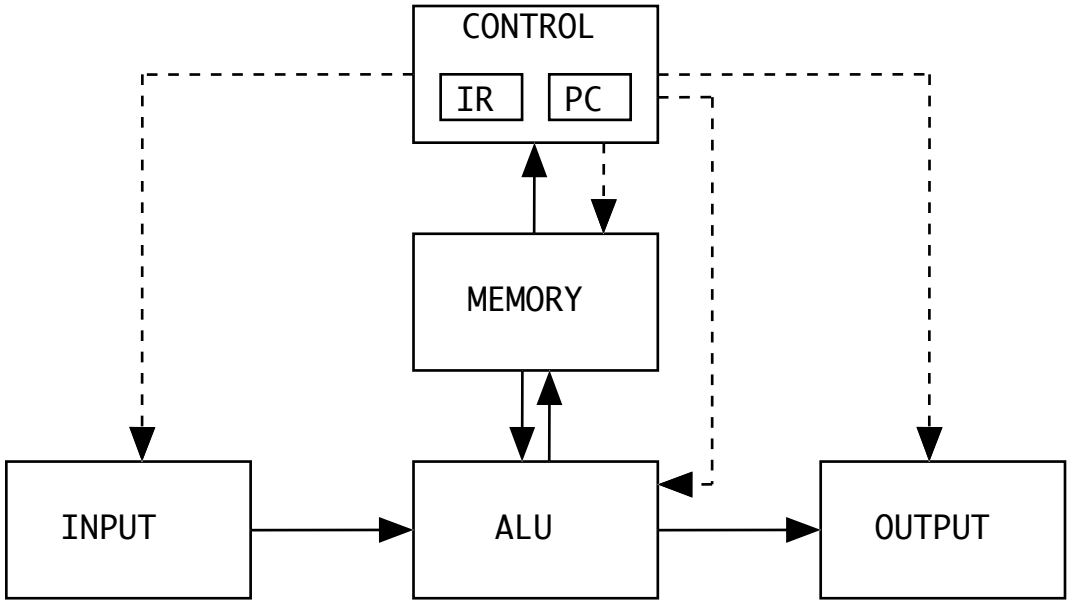
3. Von Neumann's ideas were implemented soon thereafter in several different forms.

   a) Of these, the most historically important was one implemented by a group (of which Von Neumann was a part) at the Institute of Advanced Studies at Princeton in the late 1940's.

      (1) Though it was not the first stored program computer to become operational (the EDSAC designed by Wilkes and others at Cambridge University holds this honor), it is commonly regarded as the ancestor of the main line of computer development which has continued to this day. Virtually all computers have a design that is obviously descended from this machine.

      (2) Because of its history, this machine is sometimes known as "the Johniac" or "the IAS machine"

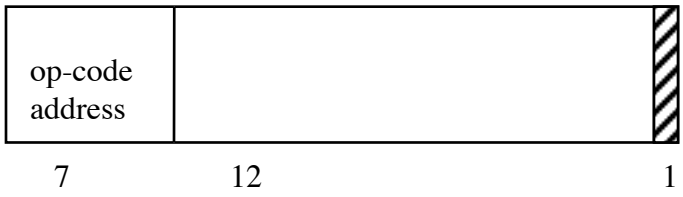
   b) The basic design, as described in Burks, Goldstine, and Von Neumann, "Preliminary Discussion of the Logical Design of an Electronic Computing Instrument" (reprinted in Bell and Newell (1971) pp. 92-119):




Solid lines = flow of data
Dashed lines = flow of control

(Most modern computers are very similar, though it is now common to have a data path between IO devices and memory, rather than requiring all IO to go through the ALU, and modern computers typically contain two or more cores that share the same memory and IO system).

c) Instructions on this machine consisted of a half word of memory (20 bits) - organized as follows:




d) The execution cycle of this machine could be described as follows:

```
while not halted
{
     fetch an instruction from the memory location
       specified by PC into IR
     update PC to point to the next instruction
     decode instruction that is in the IR
     execute instruction that is in the IR
}
```

D. The project itself was completed in 1951. The ensuing 60 years have seen multitudinous developments of each aspect of this machine, yet the family resemblance is still there, albeit faintly in some cases. (cf Chihuahua's and Great Danes - both recognizable as distant cousins of the wolf.) All general purpose computers in use today are, in fact, descendants of the Von Neumann architecture.

1. What has changed most significantly is the technology used to build the various component parts.

2. SHOW SAMPLES

   a) First generation: vacuum tubes (1950 .. 1958)

   b) Second generation: individual transistors (1958 .. 1964)

   c) Third generation: integrated circuits (1964 .. present, with increasing levels of integration (SSI, MSI, LSI, VLSI)

   d) Fourth generation: microprocessors - complete CPU's on a single chip (1972 .. present)

E. One area of continuing research interest in Computer Science is so-called "non-Von Neumann" architectures - computer system architectures that depart in some major way from this model. Thus far, though, all general-purpose computers have been designed along the lines of the basic VonNeumann architecture.

## IV. **Architecture and Organization**

A. Throughout the course, we will be using two words that are often used interchangeably, but which really have distinct technical meanings: COMPUTER ARCHITECTURE and COMPUTER ORGANIZATION.

1. Computer architecture is concerned with the FUNCTIONAL CHARACTERISTICS of a computer system - as seen by the assembly language programmer.

   a) May writers prefer to use a somewhat more precise, specific term: INSTRUCTION SET ARCHITECTURE (or ISA). The ISA is the set of machine language instructions a given machine can interpret.

      (1) Example: Current Intel chips implement the 80x86 ISA (sometimes known as IA32), which has stayed largely the same from the 80386 of the late 1980's to the Pentiums of today. (Modern versions also support 64 bit instructions, but the basic 32 bit instructions are unchanged.)

(2) The CPU's used in Macintoshes until 2006 implements the PowerPC ISA which dates to the early 1990's. (Apple now uses chips that realize the IA32 ISA instead).

[ Interestingly, although the first generation of XBox game consoles used an IA32 chip, and the Sony Playstation 2 used a chip that implements the MIPS ISA, the XBox 360, Playstation 3, and Wii all use chips based on the Power PC!]

b) One of the topics of the course will be looking at several ISA's.

(1) We will spend quite a bit of time on the ISA of the MIPS CPU. The MIPS ISA is a real commercial ISA - currently used in embedded systems such as TIVO and Cisco routers . However, MIPS is still fairly simple to understand both at the architecture and organization level, and is therefore often used in courses like this, because among ISA's that are widely used in commercial systems, it is by far the easiest to understand. It is also discussed extensively in our text.

(2) We will also look briefly at several other ISA's.

2. Computer organization is concerned with how an architecture can be REALIZED: the logical arrangement of various component parts to produce an overall system to accomplish certain design goals.

a) The technology used to build the system components.

b) The component parts themselves

c) Their interconnection

d) Strategies for improving performance.

3. Note that a given architecture may be realized by many different organizations. For example, the IA32 architecture has been realized (with some variations) by chips from the 80386 through numerous Pentium variants. The Power PC ISA has gone through several generations - most recently known as G3 and G4 (with G5 representing a major change, though still backwards compatible). In either case, a program that ran on the first implementation of the ISA (in the 1990's) could still run on a system based on the same ISA purchased today.

4. Computer architectures tend to be rather stable.

   a) E.g. IBM's basic mainframe architecture (which is still being manufactured) dates back to the mid 1960's! The IA32 architecture has its roots in an architecture developed in the late 1970's, with a major revision in the mid 1980's and minor revisions since then.

   b) A major factor in the stability of architecture is the need to be able to continue to use existing software. Potential changes to an architecture have to be weighed carefully in terms of their impact on existing software, and adoption of an altogether new architecture comes at a hugh software development cost - which is why you are still using architectures that are older than you are!

5. On the other hand, computer organization tends to evolve quickly with changes in technology - each new model of a given system will typically have different organizational features from its predecessors (though some aspects will be common, too.) The driving factor here is performance; and it is common for one or more new implementations of a popular architecture to be developed each year.

B. A fair question to ask at this point is "why should I need to learn about computer architecture and organization, given that I'm not planning to be a computer hardware designer, and that higher level language compilers insulate the software I write from the details of the hardware on which it is running?"

1. An understanding of computer architecture is important for a number of reasons:

   a) Although modern compilers hide the underlying hardware architecture from the higher-level-language programmer, it is still useful to have some sense of what is going on "under the hood"

      (1) Cf the benefit of learning Greek for NT studies.

      (2) There will be times when one has to look at what is happening at the machine language level to find an obscure error in a program.

   b) Familiarity with the underlying architecture is necessary for developing and maintaining some kinds of software:

      (1) compilers

      (2) operating systems and operating system components (such as device drivers)

      (3) embedded systems.

   c) In order to understand various performance-improvement techniques, one must have some understanding of the functionality whose performance they are improving.

2. Likewise, an understanding of computer organization is important for a number of reasons:

   a) Intelligent purchase decisions - seeing beyond the "hype" to understand what the real impact of various features on performance is.you to hardware-related issues (such as the placement of items in memory) that can have a significant impact on the performance of software.

b) Making effective use of high performance systems - sometimes the way data and code is structured can prevent efficient use of mechanisms designed to improve performance.

c) Increasingly, compilers that produce code for high performance systems have to incorporate knowledge as to how the code is actually going to be executed by the underlying hardware - especially when the CPU uses techniques like pipelining and out–of–order execution to maximize performance.

d) Understanding issues arising due to the use of parallel processing (e.g multicore computers or clusters) involves some understanding of how the various parts of a system work together.