CPS311 Lecture: Course Introduction; The Levels of Computer Structure; Architecture and Organization; A Bit of History; a Bit of Binary

Last revised June 9, 2021

Objectives:

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

Materials:

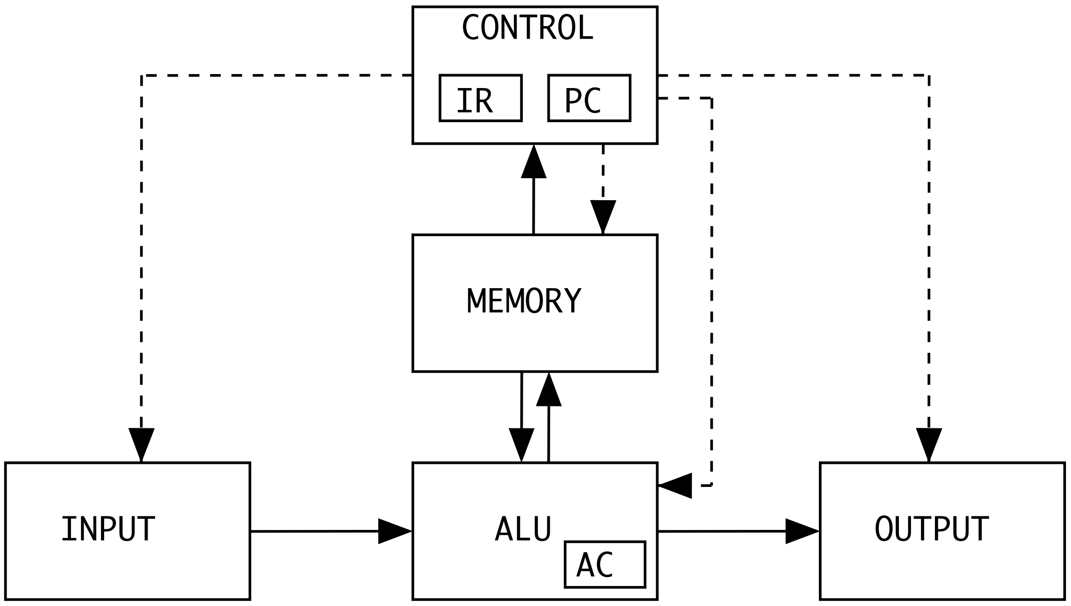
1. Projectables
2. Technology Samples

I. Preliminaries: Roll, Syllabus

II. 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.

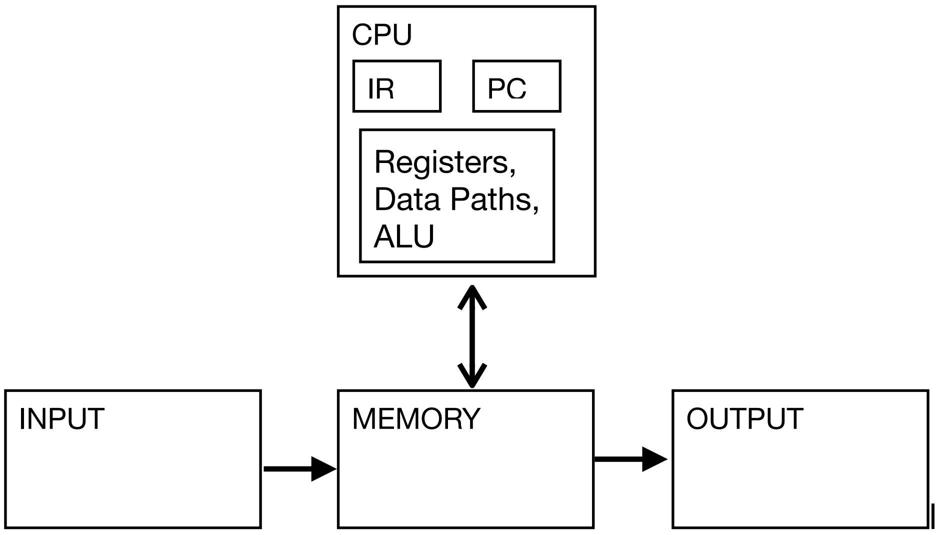
B. Virtually all computer systems today bear a resemblance to a model known as the VonNeumann model - so called because it was the basis of one of the first general purpose computer systems ever built.



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

PROJECT AND EXPLAIN ROLES OF PARTS AND INTERCONNECTIONS

1. There is one significant differences between the most current systems are structured and the original VonNeumann model; and two differences that are often - but not always - present. The following diagram is more reflective of the structure of many computer systems today.



PROJECT

a) Two of the parts in the original VonNeumann model - control and the ALU - have been combined into a single part known as the CPU, and the single AC has been replaced by a set of registers, datapaths, and an ALU proper.

b) In the original VonNeumann model, there was a single control unit and a single ALU. But many modern CPU's have multiple sets of registers, datapaths, ALUs, and control unit.

Some special processors have multiple ALU's controlled by a single control unit, while others have different sorts of specialized functional components such as graphics processors or neural network hardware.

(We will discuss all of these later in the course, but for how we focus on understanding how a single control unit and a single ALU work.)

c) In the original VonNeumann model, all data from input and output devices went into/out of the ALU. Many - but not all - systems provide paths for data to flow between IO devices and memory without passing through the CPU.

2. This basic structure is reflected in the syllabus of the course.

a) The emphasis in the first 2/3 of the course is on the CPU.

b) Memory is the focus for about 1/4 of the course.

c) Input-output is the focus for about 10% of the course near the end.

NOTE IN SYLLABUS

C. 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) Having studied programming, you realize that each software application a user sees is actually realized by a program consisting of a series of individual statements written in a language like C or C++ or Java.

   d) In CPS221, you saw how programs interface with the underlying hardware through an operating system.

   e) In this course, we will look at the hardware itself. In so doing, we will need to make use of several additional levels of abstraction.

D. In talking about the most complex part of a computer systems, the CPU it is desirable to utilize a hierarchy of levels of abstraction. Chapter 1 of the book shows how a computer system can be viewed as a hierarchy of levels of abstraction.

PROJECT Diagram from ch1

For the purposes of this course, in our consideration of the CPU, we will focus on the middle four of these levels. The two upper levels (Application Software and Operating Systems) are the domain of earlier courses, and the three lowest levels (analog circuits, devices, and physics) are largely outside of the range of this course, though we will say a little bit about them. Thus, we will focus on the following hierarchy, where I have replaced the upper and lower levels and have added a "language" column, and I've used slightly different terminology in some cases.

Level	Language(s)
HLL Programming	Python, Java, C etc.
Architecture	Machine Language specified by an ISA
MicroArchitecture	RTL
Building Blocks	Devices such as Adders, Registers, MUXes, Memories etc.
Digital Components	Gates, Flip-Flops, Memory Cells
Physical Realization	Electronics, Physics

PROJECT

At each level, the underlying layers work together to present a particular "view" or interface, and the layer itself can be described by a notation system or "language".

1. The higher-level language programming level: each application and the operating system itself is programmed using the statements of a higher-level language such as Python, Java, C etc.. 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.

2. The architecture 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 primitive 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 most 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.

(3)Languages like Java effectively split this level into two levels by adding a virtual machine layer.

(a)The Java compiler translates Java into a virtual language in .class files.

(b)The virtual language is interpreted by a Java Virtual Machine that runs in the native machine language of the platform on which it is running at run time.

(c)Of course, the details of this are outside of the scope of this course!

d) At the architecture level of abstraction, it appears that the system "understands" its machine language. This language is described by a system of notation known as an ISA - which stands for "Instruction Set Architecture"

e) We will say more about Architecture later in this lecture.

3. The microarchitecture level: the ISA is implemented by an interconnected group of building blocks from the next level down. At this and lower levels, there is no language per se that is "understood" by lower levels; but there are description languages that designers use to specify this interconnection.

   In this course, we will learn a description language known as RTL - which stands for "Register Transfer Language."

4. The building block level: a microarchitecture is implemented by a configuration of building blocks from the next level down.

   In this course, we will learn about building blocks including adders, registers, multiplexers, and memories as well as some others,

5. The digital component level: The building blocks are built as interconnections of hardware components known as gates, flip-flops, etc., combined to form adders, registers etc..

6. The physical level: In current computers, gates, flip flops, and memory cells turn, are realized from primitive electronic building blocks known as transistors, resistors, capacitors etc. But there is no reason in principle why other realizations might be possible.

   a) In the 1980's, a team of MIT students built a computer out of Tinker toys. Though it was not a general purpose computer, it is claimed to have been able to play "a mean game of tic tac toe"

      PROJECT

      References: Do Google Search on "Tinker toy computer"

   b) Other implementations that have been the subject of research (and might appear in everyday systems at some time in the future) include optical gates and biological gates (realized using biological cells and DNA)

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

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

   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!

G. At this point, we can see how the levels we have looked at correlate with the structure of the course.

   1. In many places in CS, one of two approaches is used to examine a hierarchy.

      a) Top-down. If this were the approach we used with this hierarchy, we would start with Architecture and work down to Microarchitecture, Building Blocks, Digital Components.

      b) Bottom-up. If this were the approach we used with this hierarchy, we would start with Digital Components and work up to Building Blocks, then Microarchitecture, and finally Architecture.

   2. But what we are going to do with the CPU is a hybrid of these.

      a) We will start start bottom-up through the Component and Building Block levels.

         NOTE IN SYLLABUS

b) Then we will switch to a top-down approach, going through Architecture and Microarchitecture of the CPU to where we left off with the bottom-up approach.

NOTE IN SYLLABUS

c) The reason for this is two-fold:

(1) What is possible at the lowest levels constrains what is efficiently buildable at higher levels.

(2) Before we can discuss how the architecture level is implemented, we need to understand what we are implementing!

3. Toward the end of the course, we will spend some time on ways of enhancing CPU performance: pipelining and parallelism

   NOTE IN SYLLABUS

4. We are also going to spend about a week on information representation in binary and error-correcting codes.

   (The placement of this in the Sylllabus has been dictated by getting to material you need in the early labs as early as possible.)

   NOTE IN SYLLABUS

III. **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. This corresponds to the top level in out level structure we discussed earlier.

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: While current chips used in PCs generallyh use a 64-bit architecture, they also support the 80x86 ISA, which has stayed largely the same from the 80386 of the late 1980's to today.

(2)The CPU's used in Macintoshes until 2006 implements the PowerPC ISA which dates to the early 1990's. Apple switched to chips that realize the x86 ISA instead and has recently begun switching again to M1 chips based on the ARM architecture.

b) One of the topics of the course will be looking at 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. This corresponds to the 3 remaining levels we discussed earlier.

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.

   a) For example, the x86 ISA was realized (with some variations) by chips from the 80386 through numerous Pentium variants made by Intel, AMD, and other companies.

   b) The x86-64 architecture is a 64 bit extension of x86, with new implementations by Intel or AMD being announced several times a year.

   c) The ARM architecture which is widely used in devices such as cell phones likewise sees multiple implementations by different manufacturers each year.

   d) In almost all cases, a program that ran on the first implementation of an ISA could still run on a system based on the same ISA purchased today - even though the examples noted above were first developed in the 1980's or 1990s.

4. That is, computer architectures tend to be rather stable.

   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 huge 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.

IV. **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 tens of MHz. Current computer systems typically run at 2-3 GHz - a several 100:1 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 8-16 gigabytes - another several 100:1 change in 2-1/2 decades.

(In fact, it used to be that when I revised lecture notes for each new offering of this course, one thing I ended up having to do is multiply many of the numbers by 2 or 4 - though for the last five 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 just slightly older than you are.

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

C. As we noted earlier, 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", and use a slightly varied structure.

PROJECT VONNEUMANN ARCHITECTURE AGAIN

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) 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.

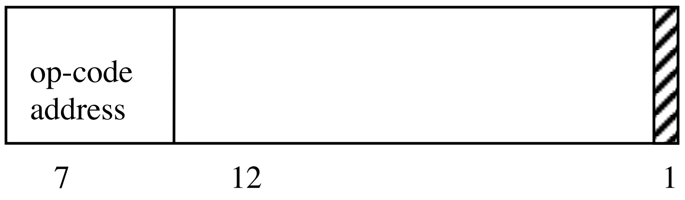
a) Of these, the most historically important was one implemented by a group (of which Von Neumann was a part) at the Institute of Advanced Studies at Princeton in the late 1940's.

   (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 memory of this machine consisted of 4096 words of 40 bits each.

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



PROJECT

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
}
```

PROJECT

D. The IAS project itself was completed in 1951. The ensuing 70 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) Today, integration has gotten to the place where it is possible to:

      (1)Put several complete processors on a single chip - yielding the multicore computer.

      (2)Put a complete computer system (CPU, memory, IO interface) on a chip, yielding one-chip computers.

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.

V. A Bit of Binary

A. Earlier, we noted that a crucial feature of VonNeumann's paper was its advocacy of the use of binary representations for information.

1. We will cover binary representation of information in detail later in the course.

2. 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

```
  1 x 2^5 =          32
+ 0 x 2^4 =           0
+ 1 x 2^3 =           8
+ 0 x 2^2 =           0
+ 1 x 2^1 =           2
+ 0 x 2^0 =           0
                     --
                     42
```

B. Conversion between decimal and binary notations can be done relatively easily.

1. To go from binary to decimal, we use the basic approach outlined above: multiply the rightmost bit by 2^0, the next bit by 2^1, the next bit by 2^2 etc. Add the products. (It helps if you memorize the powers of 2)

   Example: convert 10011101 to decimal (157)

   Exercise: 10101010 (170)

2. To go from decimal to binary, we can use successive division: Divide the decimal number by 2. The remainder is the rightmost bit of the binary equivalent. Divide the quotient by 2. The new remainder is the second rightmost bit. Divide the new quotient by 2. The new remainder is third rightmost bit ... Continue until the quotient is 0.

Example: convert 238 to binary

```
238 / 2 = 119 rem 0     <- least significant bit
119 / 2 =  59 rem 1
59 / 2 =  29 rem 1
29 / 2 =  14 rem 1
14 / 2 =   7 rem 0
7 / 2 =   3 rem 1
3 / 2 =   1 rem 1
1 / 2 =   0 rem 1     <- most significant     238 => 11101110
```

Exercise: 252 (11111100)

C. Hexadecimal

1. Writing numbers in binary is tiring, and it is very easy to make mistakes. On the other hand, converting numbers between decimal and binary is complex, so at the hardware level we like to work with the binary form.

2. Because 16 is a power of 2, we can easily convert binary to a hexadecimal representation for a number by grouping the binary bits into groups of 4 and then converting each group of 4 bits to a single hexadecimal digit, using the following table:

Binary	Hexadecimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

PROJECT

a. If the number of bits is not a multiple of 4, we can either extend the number of bits to a multiple of 4 by adding leading 0's, or we can remember to group bits from the right and allow the leftmost group to have less than 4 bits.

b. Example: Convert 1010010101 to hexadecimal

- Add leading 0's since just ten bits:

001011010101

- Group into 4's

0010 1101 0101

- Convert each group to hexadecimal:

2D5

D. We can convert hexadecimal to binary by converting each hexadecimal digit to a group of 4 bits.

Example: convert ABCD to binary

1010 1011 1100 1101

E. To convert hexadecimal to decimal, we an use one of two methods:

1. Using approaches we have already studied, convert the hexadecimal form to binary, and then convert the binary form to decimal.

2. Use the same approach as for converting binary to decimal, but use powers of 16 each time (1, 16, 256, 4096), and muitiply each power of 16 by the decimal equivalent of a hexadecimal digit (A = 10, B = 11 ...)

F. To convert decimal to hexadecimal, we an use one of two methods:

1. Using approaches we have already studied, convert the decimal form to binary, and then convert the binary form to hexadecimal.

2. Use the same approach as for converting decimal to binary, but divide by 16 each time, producing a remainder in each case in the range 0 .. 15, and then convert that to hexadecimal equivalent (10 = A, 11 = B ...)