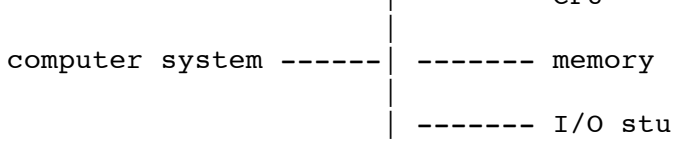
Chapter 1 -- Some basics

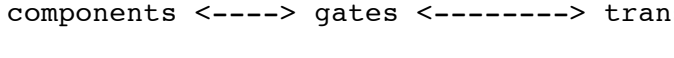
Some introductory material

basic concept within computer science and engineering:
Levels of Abstraction
hierarchy
models

used (for our purposes) to design programs, computers

when a problem is large, it needs to be broken down --
we "divide and conquer"
one way is by introducing a hierarchy (level), and solving
the problem at each level
example: design of a computer
1. transistors available
2. gates, flip flops
3. components, like registers and adders
4. CPU, memory system, I/O devices
5. computer system





TOP-DOWN vs. BOTTOM-UP design

another example: software levels of abstraction
writing a large program -- >10,000 lines of code
TOP-DOWN:
divide into modules, design each module separately.
define procedures (functions) that accomplish a task.
specify interfaces (parameters) for the procedures.

the implementation of a function is independent of
its interface specification! it is a different level
in the abstraction of program design.

the "big picture" -- a computer with software running on it.

```
  HLL                                          computer
  Pascal, C, Fortran  .  .  .  .  .  .  .  .   hardware
```

how do we get from one to the other?
wanted: write in nice abstract HLL.
have: stupid computer that only knows how to execute machine language

what is machine language?
binary sequences (lots of 1's and 0's in a very specific order) interpreted by computer as instructions.
not very human readable.

to help the situation, introduce assembly language --
a more human readable form of machine language.
uses MNEUMONICS for the instruction type, and operands
BUT, now we need something to translate assembly lang. to machine lang.: an ASSEMBLER

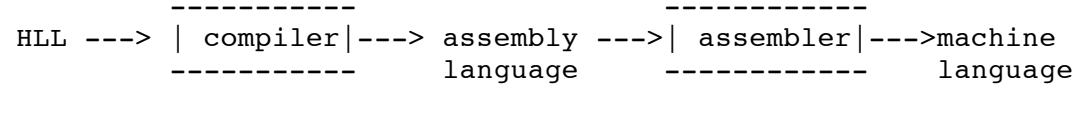
an example might be something like:

```
  add  AA, BB
```

"add" is the mneumonic or opcode (operation code)
AA and CC are the operands, the variables used in the instruction.

lastly, if we had a program that translated HLL programs to assembly language, then we'd have it made.
a COMPILER does this.

complete picture:



(least detailed) (top level) ... (most detailed) (bottom level)

this course deals with the software aspects of assembly language, assemblers and machine language. It also deals with the hardware aspects of what the computer does to execute programs. It is an introduction to study of COMPUTER ARCHITECTURE: the interface between hardware and software.

about COMPUTER ARCHITECTURE

the relationship between hardware (stuff you can touch)
and software (programs, code)

I can design a computer that has hardware which executes
programs in any programming language.
For example, a computer that directly executes Pascal.

So, why don't we do just that?
1. From experience (in the engineering community), we know
that the hardware that executes HLL programs directly
are slower than those that execute a more simple, basic
set of instructions.

2. Usability of the machine. Not everyone wants a Pascal
machine. ANY high level language can be translated into
assembly language.

In this class, in whatever language you are writing programs,
it will look like you have a machine that executes
those programs directly.

What we will do:

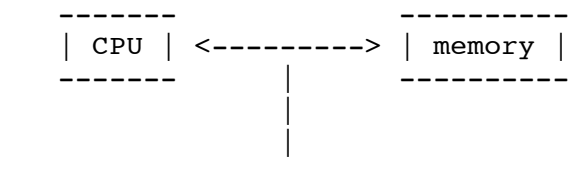
hll ---> SASM ---> Pentium

we assume that you know a hll (high level language, like C++, C,
Pascal, Fortran). From that, we can give you
SASM. Later in the semester, you will learn Pentium.
Programs will be written in both SASM and Pentium.

hll and SASM are each abstractions.
Each defines a computer architecture.
Pentium happens to be a real (manufactured) architecture.

basic computer operation

simplified diagram of a computer system (hardware!)



```
        -------
        | I/O |
        -------
```

CPU -- controls the running of programs
executes instructions
makes requests of the memory
CPU stands for central processing unit
CPU and processor are synonyms (book uses the term processor)

NOTE: Many PC users incorrectly identify the term CPU with whatever is in the box that their display sits on top of. Chances are the real CPU is inside that box, but there will be many more things in there as well.

memory -- where programs and program variables are stored
handles requests from the CPU

(STORED PROGRAM COMPUTER concept)

interaction between processor and memory.
to execute an instruction, the processor must be able to request 3 things from memory:

1. instruction FETCH
2. operand (variable) load LOAD
3. operand (variable) store STORE

the memory really only needs to be able to do 2 operations

1. read (fetch or load)
2. write (store)

where? a label specifies a unique place (a location) in memory.
a label is often identified as an address.

read: CPU specifies an address and a read operation
memory responds with the contents of the given address

write: CPU specifies an address, data to be stored, and a write operation
memory responds by overwriting the data at the address specified

for discussion: how (most) processors operate WRT the execution of instructions.

discussion by a generic assembly language instruction example:

```
mult   aa, bb, cc
```

instructions and operands are stored in memory.

before they can be used by the processor, they must
be fetched/loaded

processor steps involved:
1. fetch the instruction
   questions for later: which instruction? at what address?
2. figure out what the instruction is -- DECODE
   IT IS A MULT INSTRUCTION
   this also reveals how many operands there are, since
   the number of operands is fixed for any given instruction
   THERE ARE 3 OPERANDS
3. load operand(s)
   OPERANDS ARE bb AND cc
4. do the operation specified by the instruction
   MULTIPLY bb AND cc TOGETHER
5. store result(s) (if any)
   RESULT GOES INTO VARIABLE aa

next step:
suppose we want to execute multiple instructions, like
a program

except for control instructions, execute instructions in their
(given) sequential storage order.

the CPU must keep track of which instruction is to be
executed

it does this by the use of an extra variable contained within
and maintained by the processor, called a PROGRAM COUNTER, or PC
the contents of the variable is the address of the next
instruction to be executed.

Note: Intel calls the PC an Instruction Pointer or IP. The rest
of the world uses PC.

modify the above CPU steps:
1. fetch the instruction at the address given by the PC

added step. modify the PC such that it contains the address of
the next instruction to execute

2-5. the same as above

The added step could come at any time after step 1. It is convenient
to think of it as step 2.

This set of steps works fine for all instructions EXCEPT
control instructions.

Control Instructions example

```
beq x, y, label
```

1. fetch instruction -- address given by PC
2. update PC

3. decode
   (its a BEQ instruction, and there are 3 operands)
4. fetch operands (x and y)
5. compare operands (for equality)
6. if equal, overwrite PC with address implied by 3rd operand (label)

The processor steps involved:

1. fetch the instruction
2. update PC
3. decode
4. load operand(s)
5. do the operation specified by the instruction
6. store result(s) (if any)

notice that this series of steps gets repeated constantly --
to make the computer useful, all that is needed is a way
to give the PC an initial value (the first instruction of a program),
and to have a way of knowing when the program is done, so the
PC can be given the starting address of another program.

the cycle of steps is very important -- it forms the basis for
understanding how a computer operates. The cycle of steps
is termed the INSTRUCTION FETCH and EXECUTE CYCLE.