1. **COMPUTER OPERATION.** Transistors make up logic gates that execute a computer program, and a program is the physical realization of an algorithm or concept. Project 1 filled in the gap between the two extremes: it dealt with the translation from human-readable code (i.e., an algorithm) to machine-readable code and the translation from machine-readable code to execution of that code.

So explain the process: how do you get from human-readable code (given that assembly code is at least vaguely human-readable) to something a computer understands, and how does that machine-specific version of the algorithm get executed? In short, how does a computer system work?

**Answer:**

```
C-code or Java-like  →  Assembly code  →  Assembler  →  CPU or RISC or EXECUTE
```

The compiler translates the high-level program to assembly code. The assembler translates the assembly code to machine code, which the computer can understand. For example, the assembly instruction `add r1, r2, r3`, can be translated to `0000010100000011` (RISC instruction `set`), where `r0 -> 000`, `r1 -> 001`, `r2 -> 010`, `r3 -> 011`, and before `r1`, insert `0`, so that the instruction is 16-bit.

And the 000, 0101, 00000011 is the corresponding machine code the computer understands.

After that, the machine code goes into the CPU (can be RISC or the like). For example, in our project 1, a machine code input into RISC, and the RISC decode the instruction language to machine code (we call it 'instruction') and then executes it. In the previous example, the instruction 0000, 0101, 0000, 0011. The first 3 bits are '000', so the RISC know that this is an 'add' instruction. The 4th - 6th bits are '001', so the target (the location to put the result in) is `r1`. Similarly, 7th - 9th bits and 11th - 16th bits are '010' and '011', so the two operands are located in `r2` and `r3`.

So the RISC load data from `r2` and `r3`, add them using ALU, then at least, store the result into `r1`. And then PC (or branch to some other seed address) and RISC begin to execute the next seed instruction.
3. What are CACHES for? How are they implemented? What are some of their performance characteristics (i.e., what happens if you change the cache size, associativity, or block size)?

Answer: Caches are used to speed up the access time to storage devices. Always, the memory system is hierarchical. The cache is always faster than the backing storage (e.g., DRAM, disk in the picture). So when the client (CPU) process or software would like to fetch some data or instruction from memory, instead of fetch from the backing store (e.g., disk), it fetches them from cache, which is faster, so the access time reduced significantly.

There are three main concerns of cache: the logical organization, the context management, the consistency management. The cache can be transparent or non-transparent (middle figure).

If the cache is transparent, it use the same name space with backing store, and the cache itself maintain context and consistency management. The cache is divided into equivalence classes (one of which a block of data can be put into one set. When fetch of data, first decide which set the data is in, then compare the Tag of that block with the Tag in the address if they are same, then find (means find the requested data), the “byte” (means select the correct byte in the block). The right figure is the directly mapped cache in which there is only one block in each set. It can also be fully associative (in which all blocks are in just one set), or something between (set associative) set, fully associative, and block size.