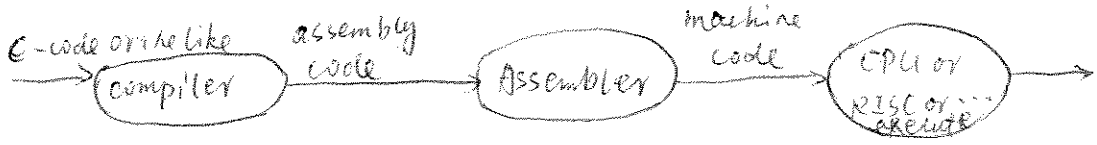
1. **COMPUTER OPERATION.** Transistors make up logic gates that execute a computer program, and a program is the physical realization of an algorithm or concept. Project 1 filled in the gap between the two extremes: it dealt with the translation from human-readable code (i.e., an algorithm) to machine-readable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

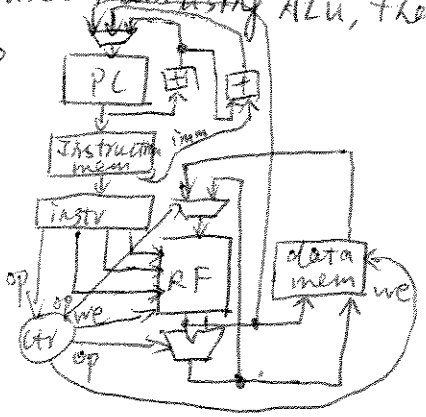
   So explain the process: how do you get from human-readable code (given that assembly code is at least *vaguely* human-readable) to something a computer understands, and how does that machine-specific version of the algorithm get executed? In short, how does a computer system work?

Answer: flow



The compiler translate the high level program to assembly code. The the assembler translate the assembly code to machine code which computer can understand. For example, the assembly instruction add r1, r2, r3, can be ~~translate~~ translate to 0000010100000011 (RISC 16 instruction set) ~~where~~, where add → 000, r1 → 001 r2 → 010, r3 → 011, and before r3, insert a "0", so that the instruction is 16bit. And the 0000, 0101, 0000,0011 is the corresponding machine code the computer understand.

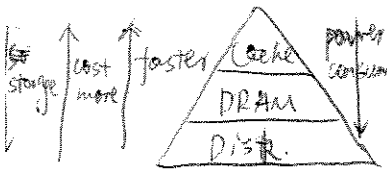
After that, the machine code goes into the CPU (can be Risc or the like). For example, in our project 1, machine code input into Risc, and the Risc decode the ~~instruction~~ machine code (we call it "instruction") and then executes it. In the previous example, the instruction 0000, 0101, 0000, 0011. The first 3 bit 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 10th ~ 12th bits are "010" and "011", so the two operand are located in r2 and r3. So the Risc load data from r2 and r3, add them using ALU, then at last, ~~put~~ store the result into r1. And then PC+1 (or branch to some other address) and RISC begin to execute the next instruction



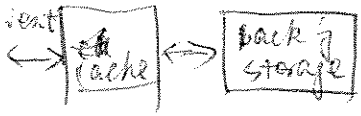
excellent 5

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)?

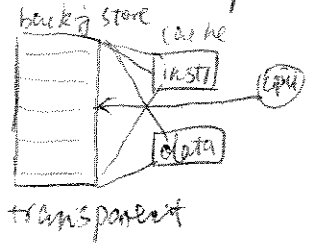
Answer: Caches are used to speed up the access ~~time~~ to storage devices.

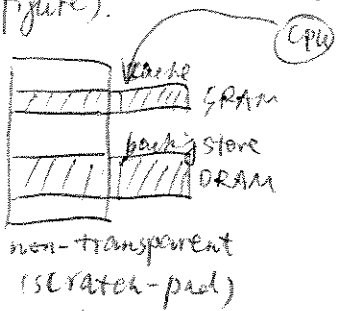


Always, the ~~the~~ memory system is hierarchy. The cache are always faster than the backing storage (e.g. DRAM, Disk in the picture). ~~So instead of~~ So when the client (CPU, processor, software, or ...) would like to fetch some data or instruction from memory, ~~it~~ instead of fetch from the backing store (e.g. disk), it fetches them from cache, which is faster, so the access time reduced significantly.

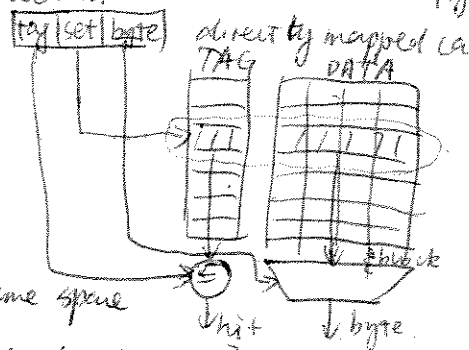


There're three main concerns of cache, the ~~or~~ logical organization ~~(caches)~~, the context management, ~~the~~ consistency management. The cache can be transparent or non-transparent (middle figure).





(left ~~picture~~ 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 ~~s~~ divided into equivalence classes ("set"), ~~the~~ each ~~last~~ ~~#~~ block of data can be put into one set. When ~~fetching~~ data, first ~~#~~ ~~is~~ 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 "hit" (means find the requested data), the "byte" ~~choose~~ select the correct byte in the block. The right figure is the directly mapped cache in which there's only one block in each set. It can also be fully associative (in which all blocks are in just one set), or something in between (set associative) ~~set~~