An Example of How a Computer Really Works

A computer is a complex system consisting of many different components. But at the heart -- or the brain, if you want -- of the computer is a single component that does the actual computing. This is the Central Processing Unit, or CPU. In a modern desktop computer, the CPU is a single "chip" on the order of one square inch in size. The job of the CPU is to execute programs.

A program is simply a list of unambiguous instructions meant to be followed mechanically by a computer. A computer is built to carry out instructions that are written in a very simple type of language called machine language. Each type of computer has its own machine language, and the computer can directly execute a program only if the program is expressed in that language. (It can execute programs written in other languages if they are first translated into machine language.)

When the CPU executes a program, that program is stored in the computer's main memory (also called the RAM or random access memory), along with the data that is being used or processed by the program. When the CPU needs to access the program instruction or data in a particular location, it sends the address of that information as a signal to the memory; the memory responds by sending back the data contained in the specified location. The CPU can also store information in memory by specifying the information to be stored and the address of the location where it is to be stored.

On the level of machine language, the operation of the CPU is fairly straightforward (although exactly how it is implemented is quite complicated). The CPU executes a program that is stored as a sequence of machine language instructions in main memory. It does this by repeatedly reading, or fetching, an instruction from memory and then carrying out, or executing, that instruction. This process -- fetch an instruction, execute it, fetch another instruction, execute it, and so on forever -- is called the *fetch-and-execute cycle*. This is about all that the CPU ever does.

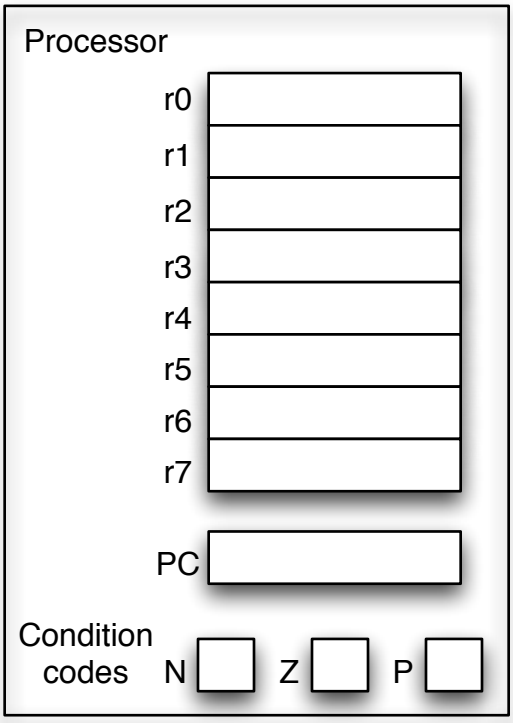
In this way, a computer executes machine language programs mechanically -- that is without understanding them or thinking about them. This is not an easy concept and that is the reason we are taking this time to demonstrate the CPU's operation. (Don't worry we won't ask you to write any programs in machine language in this class.) The other reason we are introducing this material is as a demonstration of a computer language whose behavior can be quite easily understood. While Java code will be much easier to write than machine code, it can be more difficult to understand what a line of Java code does. By understanding machine code, it gives us a model we can use to explain what features of Java code really mean.

CPU Internals:

The CPU contains a few internal registers, which are small memory units capable of holding a single number. The CPU uses one of these registers -- the program counter, or PC -- to keep track of where it is in the program it is executing. The PC stores the address of the next instruction that the CPU should execute. At the beginning of each fetch-and-execute cycle, the CPU checks the PC to see which instruction it should fetch. During the course of the fetch-and-execute cycle, the number in the PC is updated to indicate the instruction that

is to be executed in the next cycle. (Usually, but not always, this is just the instruction that sequentially follows the current instruction in the program.)

In addition to the PC, the simple processor we'll consider has 8 "general purpose" registers (called r0 - r7). Each of these registers is large enough to hold 4 bytes of data, and these registers are used to hold the values with which the computer is currently working. The processor also includes 3 condition code registers (called N, Z, and P); each of these registers holds a single bit.



Instructions:

Instructions have two parts: 1) the *opcode*, which specifies what operation an instruction performs, and 2) the *operands*, which specify the registers or memory locations used in performing the operation.

For the purpose of this discussion, it is sufficient to consider a processor that only has 6 different opcodes; real processors often have hundreds of different opcodes, but many exist for the sake of efficiency. We list them below:

ZERO_REGISTER: The ZERO_REGISTER instruction takes a single operand, the name of a register, and it writes a zero into that register.

Example: `ZERO_REG r1` *(The contents of register r1 is overwritten with a zero.)*

ADD: The ADD instruction takes three operands, all of which are names of registers. The first two operands specify the values that should be added together -- we call these "source" operands. The first source must specify a register, but the second source can be either a register or a small constant value specified in the instruction itself. The result of the addition of these two values is written to the register specified by the third operand -- what we call the "destination" register.

Example: `ADD r1 + r2 -> r3` *(The contents of register r1 is added to the contents of register r2 and the result is stored in register r3.)*

`ADD r4 + 1 -> r4` *(The contents of register r4 is added to the constant 1 and the result is written back to register r4 (overwriting the old value).)*

SUB: The SUB instruction corresponds closely to the ADD instruction with 2 source operands (one register value and either a second register value or a small constant) and one destination register operand, but instead of adding together the two source operands, the second is subtracted from the first. The result of this subtraction is stored in the destination register.

Example: `SUB r5 - r3 -> r2` *(The contents of register r3 is subtracted from the contents of register r5 and the result is written into register r2.)*

LOAD: The LOAD instruction copies a value from memory into a register. It has two operands -- one source register and one destination register -- and includes a small constant in the instruction. The memory address from which to load is computed by adding together the contents of the source register and the small constant. Because a register holds 4 bytes, we copy not only the byte at the computed address, but also 3 bytes that follow it in memory (address+1, address+2, and address+3). These 4 bytes are written into the specified destination register. The values in memory do not change.

Example: `LOAD r1 <- [r4 + 8]` *(The contents of register r4 are added to the value 8 to compute a memory address. If we assume that register r4 holds the value 20, then this load would compute the address 28. The bytes stored at memory addresses 28, 29, 30, and 31 would be copied to register r1.)*

STORE: The STORE instruction copies a value from a register to memory. It has two source register operands and includes a small constant. Like the LOAD instruction, the contents of a register are added to the small constant to compute a memory address. The other source register specifies the value to copied to the four bytes in memory starting at the computed address. The values in the source registers do not change.

Example: `STORE r7 -> [r0 + 20]` *(The contents of register r0 (assume it held the value 20) would be added to the constant 20 to compute the memory address 40. The*

contents of register r7 would overwrite the values stored at memory locations 40, 41, 42, and 43.)

FOR ALL OF THE ABOVE: After executing any of the above instructions, the instruction immediately after the current instruction should be executed. This is accomplished by adding 2 to the value in the PC register, because each of these instructions is two bytes long (as we'll see below).

In addition, anytime a general-purpose register is written (which occurs in the ZERO_REG, ADD, SUB, and LOAD instructions), the condition code registers are also updated. The condition codes are called N, Z, and P, which record whether the last value written to a general-purpose register was Negative, Zero, and Positive, respectively. At all times, exactly one of these registers will hold a 1 and the other two will hold the value 0. Which register is set to 1 is based on the value written to the general-purpose register: if the value was negative the condition codes will be set to N=1, Z=0, and P=0; if the value was zero, N=0, Z=1, P=0; and if the value was positive, N=0, Z=0, P=1.

BRANCH: Unlike the previous instructions, the branch instruction doesn't read or write general-purpose registers or memory; it reads only the condition codes and writes only the PC register. Two pieces of information are specified as part of the branch: 1) which condition codes should be checked, and 2) how many instructions to skip. If any of the condition codes checked are set to one, then the branch will be TAKEN; otherwise, the PC is set to the next sequential instruction (PC + 2). If the branch is taken, then the new PC is computed as:

PC + 2 + 2(number of instructions to skip)

There are eight possible settings for the condition codes:

NZP = branch always
NZ = branch if value was LESS THAN OR EQUAL TO ZERO
NP = branch if value was NOT EQUAL TO ZERO
N = branch if value was LESS THAN ZERO
ZP = branch if value was GREATER THAN OR EQUAL TO ZERO
Z = branch if value was EQUAL TO ZERO
P = branch if value was GREATER THAN ZERO
- = branch never

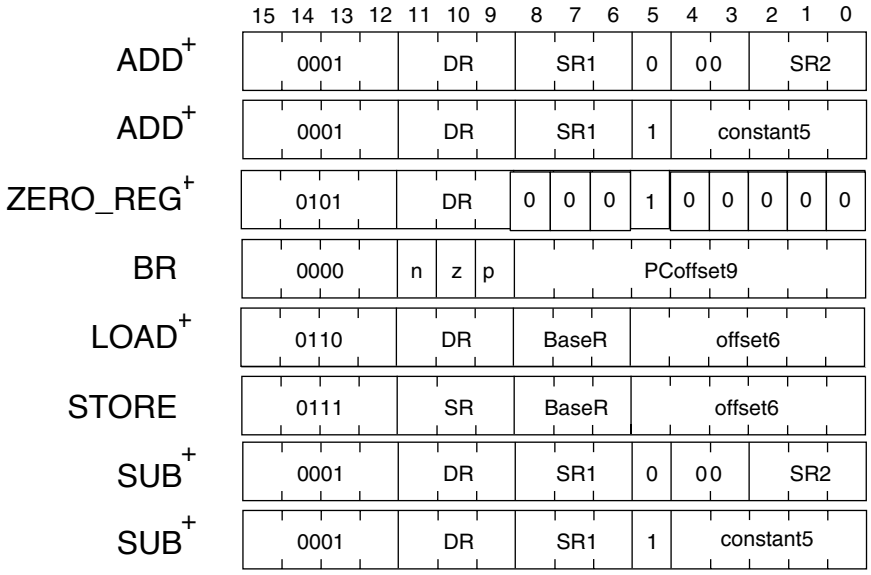
Example: `BR.NZ 1` *(If the previously written value was either negative or zero -- i.e., if either the N or Z condition codes are set -- then set the PC to PC+4, skipping 1 instruction; otherwise, continue to the next instruction by setting PC to PC+2.)*

`BR.NZP -16` *(Always set the PC to PC-30; this instruction will always branch because we're guaranteed that one of the N, Z, and P condition codes will be set.)*

Storing Instructions in Memory:

Machine language instructions are expressed as binary numbers, just like any value stored in memory. So, a machine language instruction is just a sequence of zeros and ones. Each particular sequence encodes some particular instruction. In the machine we consider here, every instruction is encoded in 16 bits, using the 4 most-significant bits to specify the opcode. In addition to the opcode, each instruction specifies 0, 1, or 2 source registers (labeled SR1, SR2 and BaseR) and 0 or 1 destination registers (DR), each of which takes 3 bits to specify because 3 bits are required to name the 8 possible registers ($2^3 = 8$). Some instructions include a small signed constant value of 5, 6 bits (constant5 and offset6, respectively). The branch instruction uses 3 bits to specify the condition codes (n, z, p) it monitors and a signed number of instructions to skip (PCoffset9).

The instructions are encoded as follows:



There are two encodings for each of the ADD and SUB instructions since they can each either take two source registers, or one source register and a small constant.

In the fetch part of the fetch-execute cycle, the address in the PC register is supplied to the memory and 2 bytes (16 bits) are read and returned to the CPU. The CPU then inspects the four opcode bits of the instruction to determine what operation will be performed and how the remaining bits of the instruction should be interpreted.

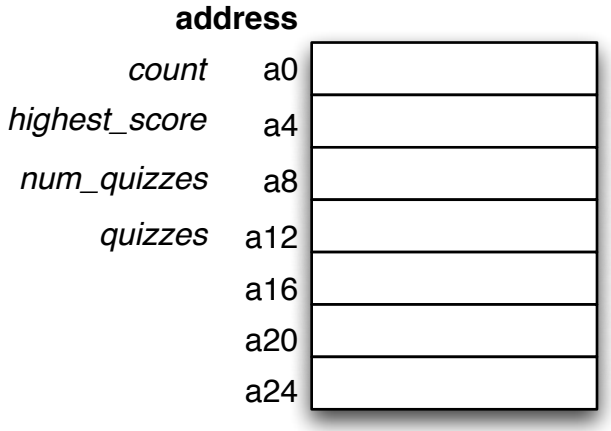
Again, our intention in showing you the binary representation of instructions is to demonstrate that instructions can be stored in memory (further demonstrating that there is no magic in how a computer works). We in no way expect you to memorize these encodings. Examples of instruction encodings are included with the code example below.

Example Code:

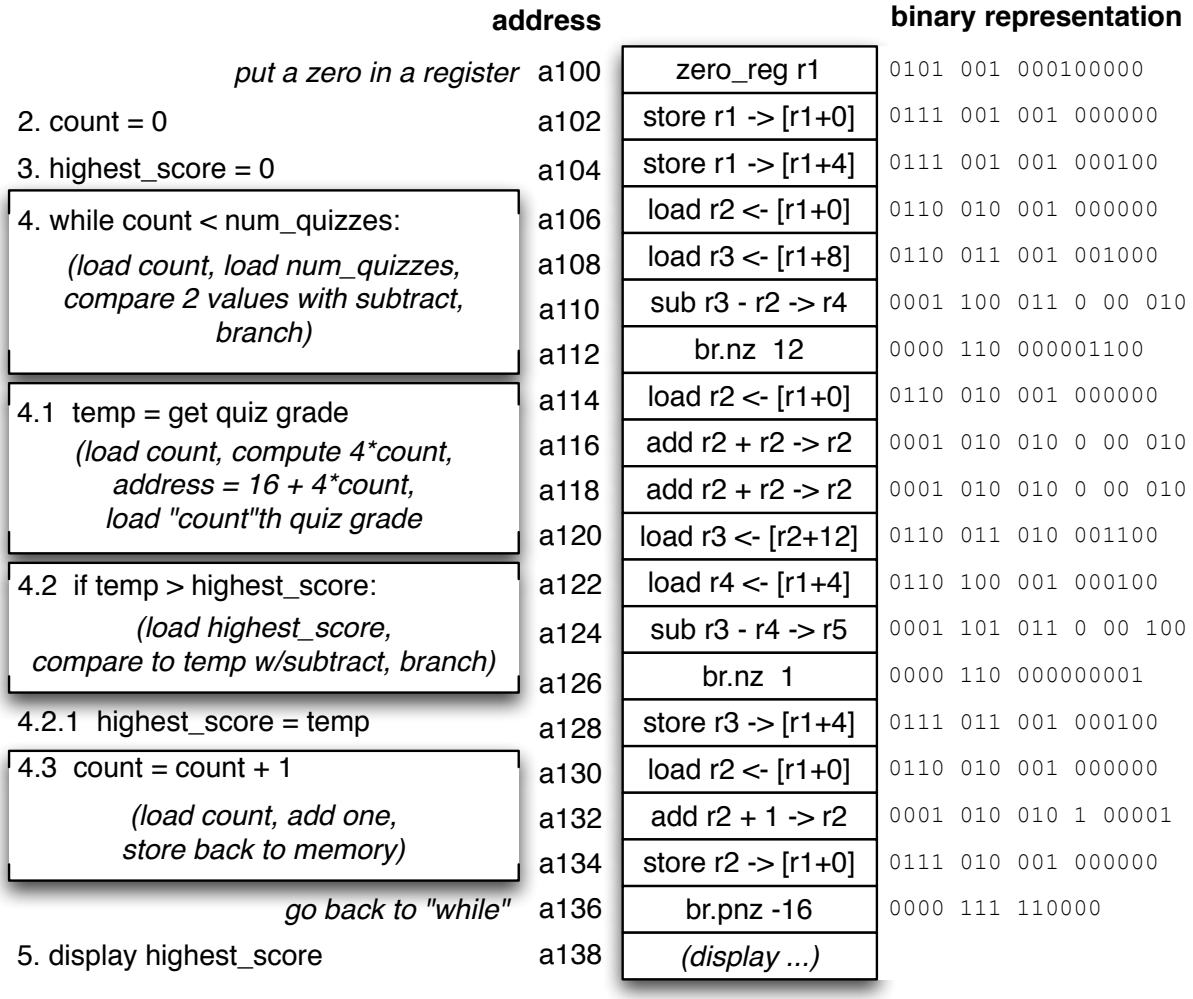
To demonstrate the execution of a machine language program, we use the following algorithm written in pseudo-code, which find the highest quiz score from a series of quiz scores:

1. get number_of_quizzes
2. count = 0
3. highest_score = 0
4. while count < number_of_quizzes:
   4.1 temp = get quiz grade
   4.2 if temp > highest_score:
      4.2.1 highest_score = temp
   4.3 count = count + 1
5. display highest_score

To demonstrate the execution of this program as a machine language program, we need to perform two mappings: 1) we need to map the state of the algorithm (e.g., count, highest_score) to locations in memory, and 2) we need to convert the algorithm to machine code. The mapping of state is quite straight forward given our discussion of data modelling; one mapping is shown below. *(Note: we are assuming each of these values is being stored in a 4-byte integer, so each value starts at an address 4 bytes after the previous one.)*



Below (on the next page) we show the implementation of the algorithm as machine code. Again, our goal here is not to teach you how to write machine code, but rather that algorithms can be implemented in machine code and demonstrate their execution on a simple processor. We have attempted to show the correspondence between the algorithm and the machine code.



Because it would be tedious to write up the execution of this code, view the video of the professor manually executing this code. This example execution uses a collection of 3 quiz scores (8, 5, 9). Based on the algorithm above, what would you expect would be the final value of highest_score? Does this match what the machine code execution computes?