CSCI 120 Introduction to Computation
Inside a computer (cont.) (draft)

Saad Mneimneh
Visiting Professor
Hunter College of CUNY

1 Example instruction set

Each instruction is identified by a unique bit pattern (called the opcode) and some operands depending on the type of the instruction. For instance, we assume just for the sake of illustration that instructions are 16 bit long and that the first 4 bits represent the opcode (i.e. determines what the instruction is). The remaining 12 bits represent the operands depending on the type of the instruction.

Data transfer Register ↔ Memory

$$\underbrace{\square\square\square\square}_{\textbf{opcode}} \quad \underbrace{\square\square\square\square}_{\textbf{register}} \quad \underbrace{\square\square\square\square\square\square\square\square}_{\textbf{memory/operand}}$$

e.g.

Load register with memory location: LOAD R $(\text{value})_{MEM}$

$$\underbrace{0001}_{\text{opcode}} \quad \underbrace{\square\square\square\square}_{\text{register}} \quad \underbrace{\square\square\square\square\square\square\square\square}_{\text{memory}}$$

Load register with operand: LOAD R $(\text{value})_{OP}$

$$\underbrace{0010}_{\text{opcode}} \quad \underbrace{\square\square\square\square}_{\text{register}} \quad \underbrace{\square\square\square\square\square\square\square\square}_{\text{operand}}$$

Store register in memory location: STORE R $(\text{value})_{MEM}$

$$\underbrace{0011}_{\text{opcode}} \quad \underbrace{\square\square\square\square}_{\text{register}} \quad \underbrace{\square\square\square\square\square\square\square\square}_{\text{memory}}$$

Data transfer Register $\leftrightarrow$ Register

$$\underbrace{\square\square\square\square}_{\textbf{opcode}} \quad 0000 \quad \underbrace{\square\square\square\square}_{\textbf{register}} \quad \underbrace{\square\square\square\square}_{\textbf{register}}$$

e.g.

Copy data from register R to register S: COPY R S

$$\underbrace{0100}_{\text{opcode}} \quad 0000 \quad \underbrace{\square\square\square\square}_{\text{R}} \quad \underbrace{\square\square\square\square}_{\text{S}}$$

Arithmetic/logical Register, Register, Register

$$\underbrace{\square\square\square\square}_{\textbf{opcode}} \quad \underbrace{\square\square\square\square}_{\textbf{register}} \quad \underbrace{\square\square\square\square}_{\textbf{register}} \quad \underbrace{\square\square\square\square}_{\textbf{register}}$$

e.g.

Add values in registers R and S and put result in register T: ADD R S T

$$\underbrace{0101}_{\text{opcode}} \quad \underbrace{\square\square\square\square}_{\text{R}} \quad \underbrace{\square\square\square\square}_{\text{S}} \quad \underbrace{\square\square\square\square}_{\text{T}}$$

If the value in R $>$ that in S put 1 in T, else put 0 in T: CMP R S T

$$\underbrace{0110}_{\text{opcode}} \quad \underbrace{\square\square\square\square}_{\text{R}} \quad \underbrace{\square\square\square\square}_{\text{S}} \quad \underbrace{\square\square\square\square}_{\text{T}}$$

If the value in R $=$ that in S put 1 in T, else put 0 in T: EQ R S T

$$\underbrace{0111}_{\text{opcode}} \quad \underbrace{\square\square\square\square}_{\text{R}} \quad \underbrace{\square\square\square\square}_{\text{S}} \quad \underbrace{\square\square\square\square}_{\text{T}}$$

AND the bit patterns in registers R and S and put result in register T: AND R S T

$$\underbrace{1000}_{\text{opcode}} \quad \underbrace{\square\square\square\square}_{\text{R}} \quad \underbrace{\square\square\square\square}_{\text{S}} \quad \underbrace{\square\square\square\square}_{\text{T}}$$

OR the bit patterns in registers R and S and put result in register T: OR R S T

$$\underbrace{1001}_{\text{opcode}}\quad \underbrace{\square\square\square\square}_{\text{R}}\quad \underbrace{\square\square\square\square}_{\text{S}}\quad \underbrace{\square\square\square\square}_{\text{T}}$$

NOT the bit pattern in register R and put result in register S: NOT R S

$$\underbrace{1010}_{\text{opcode}}\quad 0000\quad \underbrace{\square\square\square\square}_{\text{R}}\quad \underbrace{\square\square\square\square}_{\text{S}}$$

Control

e.g.

Jump to memory location if the value in register R is not 0, else continue: JUMP R (value)$_{MEM}$

$$\underbrace{1011}_{\text{opcode}}\quad \underbrace{\square\square\square\square}_{\text{R}}\quad \underbrace{\square\square\square\square\square\square\square\square}_{\text{memory}}$$

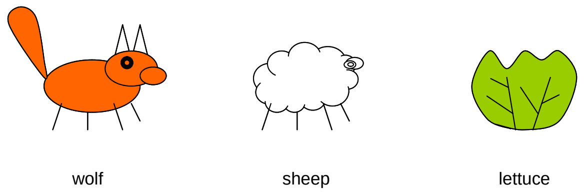
Halt: HALT

$$\underbrace{1100}_{\text{opcode}}\quad 0000\quad 0000\quad 0000$$

Therefore, the above fictitious (and over simplified) instruction set assumes that the machine has 16 registers (from 0000 to 1111) and 256 memory locations (from 00000000 to 11111111). Moreover, it assumes that the size of each register (also each operand) is 8 bits, i.e. 1 byte (because of instruction 2).

2 The wolf, the sheep, and the lettuce

A boy wants to cross the river and transport a wolf, a sheep, and a lettuce to the other side. However, he has a small boat, and can carry only one item at a time. Therefore, he has to make several trips. But if the sheep and lettuce are left unattended, the sheep will eat the lettuce. If the wolf and the sheep are left unattended, the wolf will eat the sheep. What should the boy do?



Here's a possible solution:

- transport the sheep to the other side
- come back
- transport the wolf to the other side
- bring back the sheep
- transport the lettuce to the other side
- come back
- transport the sheep to the other side

This resembles what we do in programming. We have to move things around between memory and processor/registers (the other side) in an intelligent way to satisfy a bigger task. Similar to the boy in this story, we are constrained in what we can do. There are only few instructions that we can perform, and yet, we are usually asked to put these instructions together to accomplish a certain goal. The following section describes an example.

3 An example program

Consider the following task: we wish to add all the integers from 1 to 10. Of course we can do that manually to obtain that $1+2+3+4+5+6+7+8+9+10 = 55$. We can also use the formula $1 + 2 + \ldots + n = \frac{n(n+1)}{2}$. So for $n = 10$, we have $\frac{10\times 11}{2} = \frac{110}{2} = 55$. As a side remark, there is a very nice way of showing that $1 + 2 + \ldots + n = \frac{n(n+1)}{2}$: Let call $S = 1 + 2 + \ldots + n$. Then $2S = S + S$

$$\begin{array}{rccccccc} S = & 1 & + & 2 & + & \ldots & + & n \\ +S = & n & + & n-1 & + & \ldots & + & 1 \\ \hline 2S = & (n+1) & + & (n+1) & + & \ldots & + & (n+1) \end{array}$$

$$2S = \underbrace{\hphantom{(n+1) + (n+1) + \ldots + (n+1)}}_{n(n+1)}$$

It has been told that the mathematician Gauss discovered this formula when his third grade teacher asked the class to find the sum of numbers from 1 to 100, and he instantly computed the result 5050.

But let's say that we want to compute this sum by a computer program without using the formula. Therefore, we have to carry out the sum explicitly.

How do we go about performing the sum using a computer? First of all, we have to ask ourselves the following question: What is the first thing we have to think about when we need to perform a certain task? And if I guess right, we already know the answer to this question: an algorithm.

So let's think of an algorithm. We definitely need to keep track of the sum. So let's start with the sum $S = 0$. We repeatedly add to S the appropriate number. First, we have to add 1, then we have to add 2, etc... But how do we keep track of which number of add. Obviously the number to add increases by 1 after every addition. So if we start with that number being 0, we have to repeatedly increment the number and add it to S until we reach 10, where we perform the last addition.

```
$S \leftarrow 0$
$x \leftarrow 0$
```

$$\left.\begin{array}{l} x \leftarrow x+1 \\ S \leftarrow S+x \\ \\ x \leftarrow x+1 \\ S \leftarrow S+x \\ \\ \vdots \\ \\ x \leftarrow x+1 \\ S \leftarrow S+x \end{array}\right\} \quad 10\ \textit{times}$$

Now that we have the algorithm, let's see how we can implement it using the machine language, i.e. the instruction set. Obviously we need to keep track of S and x, and for that we can use two registers, say R_0 and R_1, respectively.

First, we need to load the appropriate initial values in the registers. We can use the instruction that loads a register with a specific operand.

```
Load $R_0$ 00000000
Load $R_1$ 00000000
```

We also need the number 1, which we keep adding to x. Let's store this into another register, say register R_2.

```
Load $R_2$ 00000001
```

Next, we need to repeat 10 times: add 1 to x and add x to S (we call this the procedure below). We can do that by first adding the values in registers R_1 and R_2 and storing the result in register R_1, then adding the values in registers R_0 and R_1, and storing the result in register R_0.

```
Add $R_1$ $R_2$ $R_1$
Add $R_0$ $R_1$ $R_0$
Add $R_1$ $R_2$ $R_1$
Add $R_0$ $R_1$ $R_0$
Add $R_1$ $R_2$ $R_1$
Add $R_0$ $R_1$ $R_0$
Add $R_1$ $R_2$ $R_1$
Add $R_0$ $R_1$ $R_0$
Add $R_1$ $R_2$ $R_1$
Add $R_0$ $R_1$ $R_0$
Add $R_1$ $R_2$ $R_1$
Add $R_0$ $R_1$ $R_0$
Add $R_1$ $R_2$ $R_1$
Add $R_0$ $R_1$ $R_0$
Add $R_1$ $R_2$ $R_1$
```

Add R_0 R_1 R_0
Add R_1 R_2 R_1
Add R_0 R_1 R_0
Add R_1 R_2 R_1
Add R_0 R_1 R_0
Halt

Now R_0 holds the result. Our final program will be as follows:

```
0010000000000000
0010000100000000
0010001000000001
0101000100100001
0101000000010000
0101000100100001
0101000000010000
0101000100100001
0101000000010000
0101000100100001
0101000000010000
0101000100100001
0101000000010000
0101000100100001
0101000000010000
0101000100100001
0101000000010000
0101000100100001
0101000000010000
0101000100100001
0101000000010000
0101000100100001
0101000000010000
1100000000000000
```

This does not look so bad. But what if we need to perform what once Gauss was asked, i.e. compute the sum for $n = 100$? More generally, what if n is large? Our program will get larger and larger. Not only we will have to generate such a large program, but also we will waste the memory to store the program! More importantly, we have to write a different program for different values of n.

An important thing to notice is that our program contains redundant parts. We repeat the same procedure 10 times. So there must be a way to store that procedure only once in memory and instruct the program to repeat it a variable number of times. This can be done using the Jump instruction.

Let's say that our program is stored in memory starting at location 100. We first need to perform some initialization (loading registers with some values) and then perform the procedure for the first time. Say that procedure is stored in memory starting at location 200. Then after performing the last instruction of that procedure, we can instruct the control unit to jump back to location 200, using a Jump instruction. The procedure will be executed again. Actually, it will be executed indefinitely, because every time we perform the procedure, we jump back to location 200.

Therefore, we also need a way to stop. Luckily, the Jump instruction is conditional. The Jump instruction specifies two parameters: a register and a memory location. The Jump instruction jumps to the specified memory location only if the value in the specified register is not 0.

We can initially load a register with the value n. Every time we perform the procedure, we also decrease the value in that register by 1, i.e. we subtract 1 from it. Hopefully we know how to do that, we just need to add the representation of -1 in two's complement. So that part is easy. Now we can make our Jump instruction conditional on that register. It will always Jump until the value in the register becomes 0. Therefore, we execute the procedure exactly n times.

```
(Start with some initialization)
100: Load $R_0$ 00000000 ($S \leftarrow 0$)
102: Load $R_1$ 00000000 ($x \leftarrow 0$)
104: Load $R_2$ 00000001 (this is 1)
106: Load $R_3$ 11111111 (this is $-1$)
108: Load $R_4$ $n$ (in binary representation)
110: Jump $R_4$ 11001000 (this is 200)
112: Store $R_0$ 11011100 (store the value of $R_0$ in memory location 220)
114: Halt
⋮
(the procedure)
200: Add $R_4$ $R_3$ $R_4$ ($R_4 \leftarrow R_4 - 1$)
202: Add $R_1$ $R_2$ $R_1$ ($R_1 \leftarrow R_1 + 1$)
204: Add $R_0$ $R_1$ $R_0$ ($R_0 \leftarrow R_0 + R_1$)
206: Jump $R_4$ 11001000 (jump to 200 if $R_4 \neq 0$)
208: Store $R_0$ 11011100
210: Halt
```

The final result can be retrieved from memory location 220. The program as loaded into memory will be as follows:

```
100: 0010000000000000
102: 0010000100000000
104: 0010001000000001
106: 0010001111111111
108: 00100100 $n$
110: 1011010011001000
112: 0011000011011100
114: 1100000000000000
200: 0101010000110100
202: 0101000100100001
204: 0101000000010000
206: 1011010011001000
208: 0011000011011100
210: 1100000000000000
```

Usually, a programmer does not have to write this kind of program using the native machine language (the instruction set). High level programming languages such as C, C++, and Java are available. The programmer can write the program in a high level language using an easier syntax, and then the binary program that can be loaded into memory is generated. For instance, here's a C syntax for a function called sum:

```
int sum(int n) {
      int S = 0;
      int x = 0;
      int i;
      for (i = n; i > 0; i = i - 1) {
          x = x + 1;
          S = S + x;
      }
      return S;
}
```

The sum $S = 1 + 2 + \ldots + n$ can be also computed using the concept of **recursion**:

```
int sum(int n) {
      if (n == 0)
        return 0
      else
        return n + sum(n - 1)
}
```

Here the function sum, which computes the desired sum, is defined in terms of itself. This is not a circular definition because the function is defined in terms of the same function on smaller values. So eventually it will stop. This is called a recursive definition or simply recursion.

The above function is simply saying the following: if n is equal to 0, then just return 0; otherwise, return the result of adding n to the result of the function sum evaluated on $n - 1$. This is not to be surprising because if we define $S(n) = 1 + 2 + \ldots + n$, then $S(n) = n + S(n-1)$ by definition. So for instance, if $n = 10$, the first call to the function will return 10+sum(9). But sum(9) will recursively initiate another call to the function sum with $n = 9$ this time, and will return 9+sum(8), and so on until we reach sum(0), which returns 0. We end up with $10 + 9 + 8 + \ldots + 2 + 1 + 0$. We will look at other examples of recursion later when we discuss the topic of computation as seen by various disciplines such as linguistics, literature, poetry, music, architecture, biology, etc...

Almost all programming languages have similar constructs as the ones we see above (these are not all constructs though):

- variables: names or place holders for values, e.g. S and x.
- functions: these help define procedures that accepts some parameters as input and return a value as output. They can be regarded as building blocks or black boxes. A procedures accepts an input and produces an output. In our case, the input is n, the output is $1 + 2 + \ldots + n$.
- conditional: if-else statement, if the condition is true the statement executes the part that appears after **if**; otherwise, it executes the part that appears after **else**.

- loops: for loop statement or while loop statement, while the condition is true, the part inside the loop is executed over and over, until the conditions becomes false. In the case above, the condition is that i must be greater than 0, and i is decremented every time the loop is entered.
- recursion: a way to defined expressions recursively. This is sometimes very convenient for expressing an idea, e.g. $S(n) = n + S(n-1)$, but not necessarily the most efficient.