CSCI 2240
Program 3

# Virtual Computer

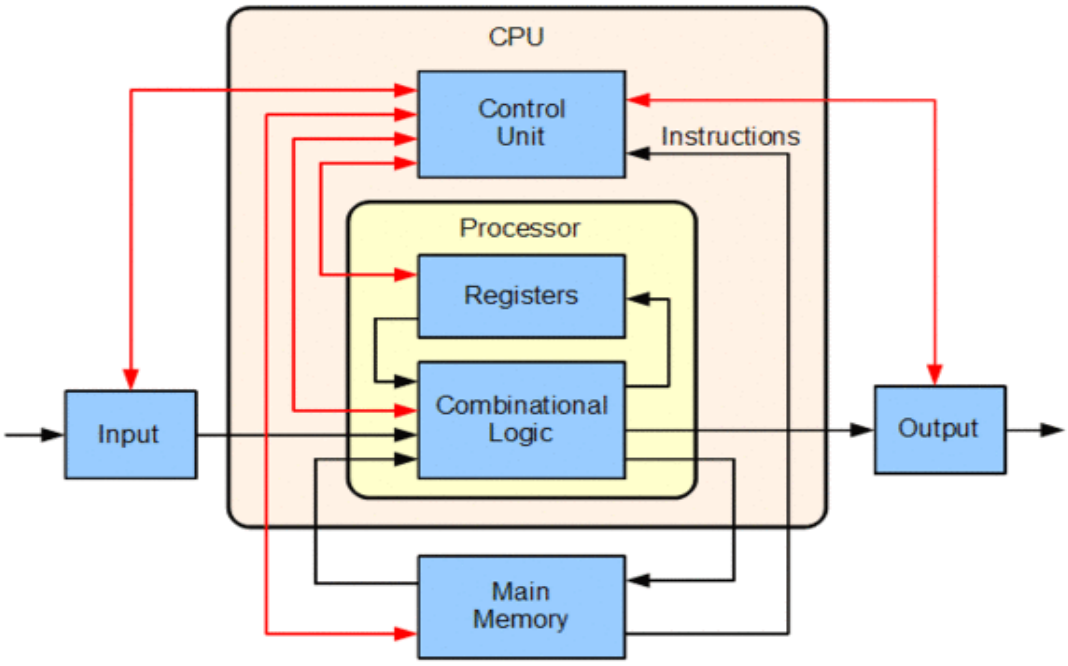
Concepts Covered

- Pointers
- Memory Concepts
- String Manipulations
- Formatted I/O

This assignment has you create your own virtual computer with its own limited instruction set and assembly language. The program you write will accept as input programs written in this language, "compile" it into the computer's instruction set and execute the program.

## The Block Diagram of a Basic Computer

Block diagram of a basic computer with uniprocessor CPU. Black lines indicate data flow, whereas red lines indicate control flow. Arrows indicate direction of flow.
(https://en.wikipedia.org/wiki/Central_processing_unit)




## The Virtual Machine

To simulate a real CPU, the VM has several registers:

- *Accumulator*, which acts as a register that operations are performed upon. Data in the *Accumulator* are in numeric format.
- *InstructionCounter*, which stores the address of the current instruction to be executed.
- *InstructionRegister*, which stores the current instruction fetched from VM memory. *InstructionRegister* stores the current instruction in the ASCII format.
- *OperationCode* and *Operand*, these two additional registers are used for storing the decoded components of the current instruction. *OperationCode* and *Operand* store the numeric values.

The computer has a 100 word memory, where each word is a signed 4 digit integer. Instructions and stored values (data) share the same 100 word memory.

The computer is capable of handling instructions, 4 digit signed integer values, and strings. **Data and program in the VM's memory are stored in ASCII digit characters**:

- When interpreted as an instruction a word can be broken down into two 2-digit chunks. The first 2 digits represents the instruction (*OperationCode*), the second represents the memory address that instruction uses (*Operand*).
- When a word is interpreted as a value, the entire 4 digits is the value. If a value is negative, the minus sign occupies the first byte of the 4 digits.
- When interpreted as a string every 2 digits represents an ASCII character in the string. The only characters that are understood are NULL (00), newline (10) and A-Z (65-90).

When the program in memory is executed, it starts with the instruction at memory location 0, and stops when the halt instruction is reached.

## The Assembly Language

The assembly language consists of a number of commands, each representing an instruction in the VM. Each command is on a line of its own, and is preceded by the 2 digit address in memory it is to be placed and followed by a single value, typically an address in memory.

Listing of commands and their instruction code
*Input/Output*
READ 10 – Retrieves a value from the user and places it in the given address
WRIT 11 – Outputs a word from the given address to the terminal
PRNT 12 – Outputs a string starting at the given address, will continue outputting consecutive words as strings until NULL is reached

*Load/Store*
LOAD 20 – Load a word from the given memory address into the accumulator. The word is decoded into numeric value.
STOR 21 – Store the word in the accumulator into the given address. The value is encoded into ASCII format.
SET 22 – Stores the given word into the preceding address (Note: The operation code value of 22 will never appear in a compiled program, is only included for completeness)

*Arithmetic Operations*
ADD 30 – Add the word at the given memory address to the accumulator: *acc = acc + operand*
SUB 31 – Subtract the word at the given memory address from the accumulator: *acc = acc - operand*
DIV 32 – Divide the accumulator with the word at the given memory address: *acc = acc / operand*
MULT 33 – Multiply the word at the given memory address to the accumulator: *acc = acc * operand*
MOD 34 – Mods the word at the given memory address to the accumulator: *acc = acc % operand*

Note that data from memory must be decoded into numeric values before being used in arithmetic operations. Data in the *Accumulator* must be converted into ASCII format before being stored back to memory.

*Control Operations*
BRAN 40 – Execution jumps to the given memory location
BRNG 41 – Execution jumps to the given memory location if the accumulator is negative
BRZR 42 – Execution jumps to the given memory location if the accumulator is zero
HALT 99 – Terminates execution, no address given, value of 99 is standard, also prints out the state of memory in a tabular format like shown below. Notice the spacing and justification of the output.

```
REGISTERS:
accumulator             +0000
instructionCounter         00
instructionRegister     +0000
operationCode              00
operand                    00
MEMORY:
       0     1     2     3     4     5     6     7     8     9
 0 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
10 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
20 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
30 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
40 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
50 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
60 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
70 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
80 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
90 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
```

Example Program:

The following reads two values from the user and outputs the larger of the two

```
00 READ 9
01 READ 10
02 LOAD 9
03 SUB 10
04 BRNG 7
05 WRIT 9
06 HALT 99
07 WRIT 10
08 HALT 99
09 SET 0
10 SET 0
```

The "compiled" version of this in memory would look like:

```
1009
1010
2009
3110
4107
1109
9999
1110
9999
0000
0000
```

Where the first instruction is stored in the $0^{th}$ address in memory and each following instruction is stored in subsequent addresses. Note that since each assembly command is preceded with 2-digit address where the instruction will be stored, commands in an assembly program do not have to be listed in order.

The following simply outputs "HELLO" to the terminal, followed by a newline

```
00 PRNT 02
01 HALT 99
02 SET 7269
03 SET 7676
04 SET 7910
05 SET 0000
```

The PRNT command points to where the string starts, and each character is printed starting at that location until NULL is reached. Notice the words used need to be set in when the program is written so the words are in memory for execution.

Your Program

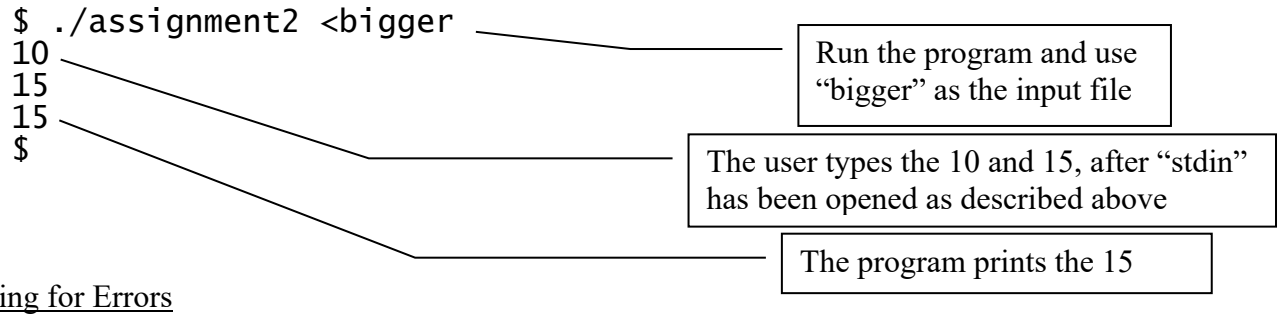
Your program will accept a program written in this assembly language (redirected from a file), parse the commands and load the instructions into the 100 word memory. If and only if this is successful, the loaded program will then be executed.

A Note About Files

Your program can read the "machine program" using the usual "fgets" or "getchar" or any of the standard ways. However, after reading in the program, your code will then have to execute it and take input from the keyboard. To do this, you will need to reassign the "stdin" file to the keyboard after reading in the machine program but before executing it; use this:

```
stdin = fopen("/dev/tty", "r");
```

For example, suppose you have the above machine program in a file called "bigger". Once the program is read in, stored in the memory, and is ready to execute, it is necessary to reassign the input to the terminal. The result would look something like:




Checking for Errors

There are a number of errors that can occur, at both compile and runtime on the virtual computer. You must catch these, and report them with the address they were encountered at and a message. A compile error will stop the compiling process and not attempt to execute the program, a runtime error will cause a HALT, terminating execution and output the state of the registers and memory.

*Compile Errors*

Within your assignment folder there are some incorrect VM programs (ce_*) that your VM should report compile errors:

- Unknown command – Unrecognized command word, they are case sensitive
- Word overflow – attempts to place a word in memory that is larger than 4 digits
- Undefined use – Command is not in the proper format
- No HALT – No HALT command is ever given

*Runtime Errors*

Within your assignment folder there are some incorrect VM programs (rt_*) that your VM should report runtime errors:

- Unknown command – Unrecognized command code
- Word overflow – attempts to place a word in memory or alter the accumulator so that it is larger than 4 digits
- Segmentation fault – attempts to access an unknown address
- Divide 0 – Division by 0 was attempted
- Unknown Character – When printing a string, and unknown character was reached (only understands NULL, newline, and A-Z)

Representing your computer

The state of your computer is represented by the following data. You may want to consider defining a struct type to hold all fields for the VM and just pass the pointer to the struct around.

- memory of 100 words– can be represented by an int array or an 2-D char array
- Accumulator – represented by an int
- InstructionCounter – represented by an int
- InstructionRegister – represented by an int
- OperationCode – stores the instruction code of the instruction, represented by an int
- Operand – stores the operand of the instruction, represented by an int

Your solution needs at least two functions in addition to main. One to accept the written program called compile() and one to execute what is in memory, called execute(). You can create additional functions that are called upon by these two, but must at least have these two separate functions. Your main function will simply call upon these two functions, compile() and execute(), execute only running if appropriate (the inputted program compiles).

Virtual Computer Programs

Within your assignment folder there are three files titled prog1, prog2, and prog3. Those files are programs written in the VM's assembly language. Your VM implementation is expected to compile and execute these programs:

prog1 – Uses a counter controlled loop to read in positive and negative values and output their sum. The first piece of data entered will be how many numbers are going to be summed.

prog2 – Uses a sentinel controlled loop to read in only non-negative numbers (a negative value will end the loop, make sure this value is not part of the computation) and outputs the average, do not worry about catching a divide by zero, this should give a runtime error.

prog3 – Reads in two numbers and if the **second** number is a multiple of the **first** outputs the string "MULT" otherwise outputs the string "NOT"

Note: After the programs are collected, any compiled program (executable) in that directory is deleted and the program is recompiled for testing. Specifically, make sure you compile the program with the same options that are used for grading:

```
gcc -Wall -ansi -pedantic computer.c –o computer
```

or whatever the correct file name is. If the compiler generates any errors or warnings you will be penalized.