Karina Pizanow

Noemi Cuin

Miriam Flores

December 1, 2015

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| OP | CO | DE | S | R | e | g | i | s | tr | M | E | M | O | R | Y | ! |

INTRODUCTION:

We decided to use four bits for the opcodes since we would only use thirteen different opcodes. This was perfect because we only had four to the second power giving us the option to choose whether we want 16 different opcodes or not. The different opcodes we chose to have include: INCR, DECR, TAKE, DISP, COPY, PLUS, MINU, MULT,DIVI, MODO, LOAD, SAVE, QUIT, JUMP and LOOP. Since we only have seventeen bits to work with we had thirteen bits left to among our registers and our memory. We decided that our registers would use up to 6 bits leaving us with 7 bits for the memory. We wanted to have more bits for memory so that the user can have more storage for the variables.

INCR:

0000

//Summary: Allows the user to increment a variable by 1.

//Precondition: We make sure that the variable exist in memory.

//Postcondition: Stores the new incremented value into place that the old variable was at.

DECR:

0001

//Summary: Can decrement a variable, chosen by the user,by one.

//Precondition: Make sure it is only a variable not a register. Negative numbers are not allowed.

//Postcondition: Will store the new decremented value back to its original spot.

TAKE:

0010

//Summary: Will allow the user to input a variable of their choice.

//Precondition: Make sure that the user inputs a variable name and the variable value.

//Postcondition: Will store that new variable into the dictionary and can be referenced again

COPY:

0011

//Summary: The user is able to copy a variable with another variable.4

//Precondition: Make sure that the variables are in memory.

//Postcondition: Copies a variable the user wants to another variable & stores into memory

PLUS:

0100

//Summary: Adds two different variables together.

//Precondition: Make sure that the variables exist. Saves the addition into the first variable.

//Postcondition: Adds two existing variables in memory and saves it into the memory of first variable.

MINU:

0101

//Summary: Subtracts two variables of choice/

//Precondition: Make sure the variables are in memory. Also, saves the subtraction in the first variables memory.

//Postcondition: Subtracts two variables and saves it into the first variables memory.

MULT :

0110

//Summary: Multiply two variables.

//Precondition: Make sure variables exist in memory. Save the answer into the first variables memory.

//Postcondition: Multiply two variables saving it into the first variables memory.

DIVI :

0111

//Summary: Divides two variables.

//Precondition: Make sure variables exist in memory. Save the answer into the first variables memory.

//Postcondition: Divides two variables saving it into the first variables memory.

MODU:

1000

//Summary: Take the Modulus two variables.

//Precondition: Make sure variables exist in memory. Save the answer into the first variables memory.

//Postcondition: Take the modulus of two variables, save answer to 1st’s memory.

LOAD\_\_:

1001

//Summary: Loads any variable that the user wants to load.

//Precondition: Make sure that the variable exist in memory.

//Postcondition: Loads any variable that the user wants.

SAVE:

1010

//Summary: Saves an integer that a user wants into a memory.

//Precondition: Make sure that the integer exists as well as the register.

//Postcondition: Saves variable into a register.

QUIT:

1011

//Summary: Ends the program.

//Precondition: Could end the program at any given point in a text file.

JUMP:

1100

//Summary:

//Precondition:

//Postcondition:

LOOP:

1101

//Summary:

//Precondition:

//Postcondition:

DISP:

1110

//Summary: Displays any variable that the user wants to.

//Precondition: Make sure that the variable exist in memory.

//Postcondition: Display a variable of choice to do whatever the user wants with it.

INDIRECT/DIRECT ADDRESSING:

We decided that we would use both indirect and direct addressing since it would correlate with the opcodes that we decided to make. In fact, this actually increased the number of opcodes that we chose to do. Also, we decided to use indirect addressing so that the user can work with loading a value and then using it to their choice. However, we preferred direct addressing since we did not have to deal with pointers and keeping track of the specific memory location that a value is at. We did not like the idea of making it hard for the user to know or to even take in consideration the memory addresses so we took that off. We will be using GPR or General Purpose Registers since they are a more adjusted and updated version of an Instruction Set Architecture.

1. We need to decide whether data is going to be store in a stack structure or register and why?

* Register, because it makes code generation easier to understand. Also, data can be stored for long periods in registers, however this means longer instructions.
* In fact, they are faster than memory, easy to compile , and are very efficiently and effectively. Adds large amounts of registers quickly.

1. Number of explicit operands per instruction? why?

* 3 and two since we are using a GPR we are able to use up to 3 set of instructions per opcode.
* general purpose registers give us the opportunity to store both the data & the addresses

1. operand location
   1. ***register to memory***
      1. We are using register to memory it is the most recent updated version of where operands can be located.
      2. It has 2 address format and can accommodate 16 bits.
      3. Can operate instruction with 1 operand in memory.
2. What kinds of operation do we want to have in our program? (taking into consideration bit manipulation)
   1. Answered above, We will probably cut some away since we are using a GPR based architecture.
3. The type and size of operands (They can be addresses, numbers or characters)
   1. Characters (WORD)

USER Documentation:

* Declare Variables:
  + memory\_location name\_of\_variable value\_of\_variable
    - Variable can not begin with an R since registers begin with an R
    - EX: 1 variable1 7
* ADDING ETC:
  + memory\_location OPCODE variable\_name variable\_name
    - * EX: 1 ADD var1 var2
    - OR
  + memory\_location OPCODE register\_name register\_name
    - * EX: 3 ADD R1 R2
    - OR
  + memory\_location OPCODE register\_name variable\_name
    - * EX: 6 ADD R2 var1
* JUMP:
  + memory\_location JUMP memory\_location\_to\_jump\_to
    - * EX: 7 JUMP 9
* ARRAY:
  + memory\_location ARRA size

num\_in\_array

num\_in array….

etc.

* EX 3 ARA 10

ARRA 4

1

3

4

5