UVSim Design Document

Functionality of the Application

The UVSim is a simple virtual machine, but powerful. The UVSim can only interpret a machine language called BasicML.

The UVSim contains the CPU, register, and main memory. An accumulator – a register into which information is put before the UVSim uses it in calculations or examines it in various ways. All the information in the UVSim is handled in terms of words. A word is a signed four-digit decimal number, such as +1234, or -5678. The UVSim is equipped with a 100-word memory, and these words are referenced by their location numbers 00, 01, ..., 99. The BasicML program must be loaded into the main memory starting at location 00 before executing. Each instruction written in BasicML occupies one word of the UVSim memory (instructions are signed four-digit decimal numbers). We shall assume that the sign of a BasicML instruction is always plus, but the sign of a data word may be either plus or minus. Each location in the UVSim memory may contain an instruction, a data value used by a program, or an unused memory area. The first two digits of each BasicML instruction are the operation code specifying the operation.

BasicML vocabulary is defined as follows:

- **I/O operation:**
    - READ = 10 - Read a word from the keyboard into a specific location in memory.
    - WRITE = 11 - Write a word from a specific location in memory to the screen.
- **Load/store operations:**

    - LOAD = 20 - Load a word from a specific location in memory into the accumulator.
    - STORE = 21 - Stores a word from the accumulator in a specific location in memory.
- **Arithmetic operation:**
    - ADD = 30 - Add a word from a specific location in memory to the word in the accumulator (leave the result in the accumulator).
    - SUBTRACT = 31 - Subtract a word from a specific location in memory from the word in the accumulator (leave the result in the accumulator).
    - DIVIDE = 32 - Divide the word in the accumulator by a word from a specific location in memory (leave the result in the accumulator).
    - MULTIPLY = 33 - multiply a word from a specific location in memory to the word in the accumulator (leave the result in the accumulator).
- **Control operation:**
    - BRANCH = 40 - Branch to a specific location in memory.
    - BRANCHNEG = 41 - Branch to a specific location in memory if the accumulator is negative.
    - BRANCHZERO = 42 - Branch to a specific location in memory if the accumulator is zero.
    - HALT = 43 - Stop the program.

The last two digits of a BasicML instruction are the operand – the memory location address containing the word to which the operation applies.

User Stories

User Story #1

As an educator, I want to help my students learn machine language and computer architecture to further their education in computer science.

User Story #2

As a computer science student, I want to use an interactive and user-friendly machine language learning program to gain hands-on experience in machine learning.

Use Cases

Use Case #1 - File Loading (old format):

- **Actor:** user
- **System:** file management module
- **Goal:** load a file in the old four-digit word format
- **Steps:**
    1. User selects the option to load a file.
    2. User selects a file in the old format.
    3. The system converts the file to the new six-digit word format with leading zeroes.
    4. The converted file is loaded into the application.
    5. The user can now edit or execute the file in the new format.

Use Case #2 - File Loading (new format):

- **Actor:** user
- **System:** file management module
- **Goal:** load a file in the new six-digit word format
- **Steps:**
    1. User selects the option to load a file.
    2. User selects a file in the new format.
    3. The file is loaded into the application without conversion.
    4. The user can now edit or execute the file in the new format.

Use Case #3 - Store Operation

- **Actor:** program or algorithm
- **System:** memory management and code processor
- **Goal:** successfully store a value in an address
- **Steps:**
    1. Retrieve the current value stored in the accumulator
    2. Store the retrieved value in a specified address
    3. Print a message indicating that the store operation was successful

Use Case #4 - Load/Store Operation (invalid address)

- **Actor:** program or algorithm
- **System:** memory management and code processor
- **Goal:** test if an exception is raised when an invalid address is input
- **Steps:**

1. Parse function code
2. Identify the target memory address from the last two digits of the word
3. If the address is invalid, print a message indicating that the operation failed

Use Case #5 - Load/Store Operation

- **Actor:** program or algorithm
- **System:** memory management and code processor
- **Goal:** successfully load a value into the accumulator and store it in an address
- **Steps:**
    1. Parse function code
    2. Identify the target memory address from the last two digits of the word
    3. Fetch value from identified memory address
    4. Copy the fetched value into the accumulator register
    5. Retrieve the current value stored in the accumulator
    6. Store the retrieved value in a specified address
    7. Print a message indicating that the store operation was successful

Use Case #6 - Add Operation

- **Actor:** program or algorithm
- **System:** memory management and code processor
- **Goal:** add a word from a specific location in memory to the word in the accumulator (leave the result in the accumulator)
- **Steps:**
    1. Parse function code
    2. Identify the target memory address from the last two digits of the word

3. Fetch value from identified memory address
4. Add the fetched value to the current value in the accumulator
5. Update the accumulator with the result

Use Case #7 - Subtract Operation

- **Actor:** program or algorithm
- **System:** memory management and code processor
- **Goal:** subtract a word from a specific location in memory from the word in the accumulator (leave the result in the accumulator)
- **Steps:**
    1. Parse function code
    2. Identify the target memory address from the last two digits of the word
    3. Fetch value from identified memory address
    4. Subtract the fetched value from the current value in the accumulator
    5. Update the accumulator with the result

Use Case #8 - Divide Operation

- **Actor:** program or algorithm
- **System:** memory management and code processor
- **Goal:** divide the word in the accumulator by a word from a specific location in memory (leave the result in the accumulator)
- **Steps:**
    1. Parse function code
    2. Identify the target memory address from the last two digits of the word
    3. Fetch value from identified memory address

4. Check if the divisor (i.e., the fetched value) is not zero
5. Divide the current value in the accumulator by the fetched value
6. Update the accumulator with the result

Use Case #9 - Multiply Operation

- **Actor:** program or algorithm
- **System:** memory management and code processor
- **Goal:** multiply a word from a specific location in memory to the word in the accumulator (leave the result in the accumulator)
- **Steps:**
    1. Parse function code
    2. Identify the target memory address from the last two digits of the word
    3. Fetch value from identified memory address
    4. Multiply the fetched value by the current value in the accumulator
    5. Update the accumulator with the result

Use Case #10 - Read Operation

- **Actor:** program or algorithm
- **System:** memory management and code processor
- **Goal:** read a word from the keyboard into a specific location in memory
- **Steps:**
    1. Prompt the user for input
    2. Validate the input format (+/- followed by a four-digit value)
    3. Store the validated input in the specified memory address
    4. Print a confirmation message if the input is successfully stored

Use Case #11 - Read Operation (invalid input)

- **Actor:** program or algorithm
- **System:** memory management and code processor
- **Goal:** test what would happen if the input caused an error
- **Steps:**
    1. Prompt the user for input
    2. Validate the input format (+/- followed by a four-digit value)
    3. If the input is bad, print a message indicating that the operation failed

Use Case #12 - Write Operation

- **Actor:** program or algorithm
- **System:** memory management and code processor
- **Goal:** write a word from a specific location in memory to screen
- **Steps:**
    1. Retrieve the value stored in the specified memory address
    2. Print the retrieved value as output

Use Case #13 - Write Operation (invalid address)

- **Actor:** program or algorithm
- **System:** memory management and code processor
- **Goal:** test what would happen if the address caused an error
- **Steps:**
    1. Retrieve the value stored in the specified memory address
    2. If the address is bad, print a message indicating that the operation failed

Use Case #14 - Branch Operation

- **Actor:** program or algorithm
- **System:** memory management and code processor
- **Goal:** branch to a specific location in memory
- **Steps:**
    1. Retrieve the target memory address from the instruction
    2. Update the program counter to point to the target memory address
    3. Print a message indicating the execution of the branch operation

Use Case #15 - BranchNeg Operation

- **Actor:** program or algorithm
- **System:** memory management and code processor
- **Goal:** branch to a specific location in memory if the accumulator is negative
- **Steps:**
    1. Check if the sign of the value in the accumulator is negative
    2. If negative, update the program counter to point to the target memory address
    3. If not negative, proceed to the next instruction
    4. Print a message indicating the execution of the branchNeg operation

Use Case #16 - BranchZero Operation

- **Actor:** program or algorithm
- **System:** memory management and code processor
- **Goal:** branch to a specific location in memory if the accumulator is zero
- **Steps:**

1. Check if the value in the accumulator is zero
2. If zero, update the program counter to point to the target memory address
3. If not zero, proceed to the next instruction
4. Print a message indicating the execution of the branchZero operation

Use Case #17 - Halt Operation

- **Actor:** program or algorithm
- **System:** memory management and code processor
- **Goal:** Stop the program
- **Steps:**
    1. Print a message indicating that the program execution has halted
    2. End the execution loop

Use Case #18 - File Saving

- **Actor:** user
- **System:** file management module
- **Goal**: save a file in the current format (either old or new)
- **Steps:**
    1. User selects the option to save a file.
    2. User specifies the filename and location.
    3. The file is saved in the current format (either old or new) based on its original format or user preference.

Use Case #19 - Multiple File Management:

- **Actor:** user
- **System:** application interface
- **Goal:** manage and interact with multiple open files simultaneously
- **Steps:**
    1. User opens multiple files within the application.
    2. The application displays tabs or sub-windows for each open file, allowing the user to switch between them.
    3. The user can edit or execute each file independently within the application instance.
    4. Changes made to each file are saved separately without interference from other open files.