Jay Facultad
Nathan Sahelijo

Design Checklist for HW6: Universal Virtual Machine

Part 1: Universal Machine

1. ***What problem are you trying to solve?***

   We are trying to emulate a universal machine.

2. ***What example inputs will help illuminate the problem?***

   The input will be a .um file containing machine instructions. Examples of instructions the file will include: load value 100 to register 1, load value 50 to register 2, add values stored in register 1 and register 2 and store result in register 6, output register 3. The instructions will be encoded in hex:

   ```
   d200 0064 d400 0032 3000 00ca a000 0003
   ```

3. ***What example output go with those example inputs?***

   The output of the program will be determined by the operations inputted from the file containing machine instructions. The output of the example above will be the hex value `96` (150 in decimal).

4. ***Into what steps or subproblems can you break down the problem?***

   1. Determine the file size, open and read in the file input / extract each 32-bit instruction from the file input and load all instructions to segment 0.
   2. Perform operations specified by each 32-bit instruction in the order determined by the instruction pointer.
   3. Map each 32-bit instruction to designated segments (determined by instructions).
   4. Insert each 32-bit instruction to appropriate location within the mapped segment (determined by instructions).

5. ***What data are in each subproblem?***

   1. Machine instructions. Each instruction is 32-bits in size and composed of 8 hex values. Each instruction will be one of two types. The first type's 4 most significant bits will represent the operation to be performed and the 9 least significant bits will represent 3 registers, with 3 bits representing one register. The second type's 4 most significant bits will represent operation 13, which is loadval, followed by 3 bits representing the reister, and the remaining bits representing

Design Checklist for HW6: Universal Virtual Machine

the value to be loaded to that register. This type is only used specifically for the loadval instruction.

2. Segment 0 will initially hold the entire program, therefore the order that the instructions will be executed will be dictated by this segment. The program counter in this segment is advanced to the next instruction (move pointer by 1 offset) which will then be executed.
3. The segmented memory will be represented as a sequence of pointers. Each position in the sequence represents a segment and will store a pointer that points to the data type that represents the container holding the values (see below).
4. 32-bit values stored in the segments.

6. What code or algorithms go with that data?

- The code for Hanson interface, specifically sequences.
- Calculations and manipulations based on the operation code.
- Bitfield manipulations (shifts, AND, OR, masks).

7. What abstractions will you use to help solve the problem?

- A sequence will represent the segmented memory. The sequence will hold pointers to each segment.
- The segments will be represented as static arrays. The length of the array is determined by the Map Segment operation (op code 13).
- Each index of the array will store a 32-bit word.
- A stack will store location of unmapped segments (ie. segment identifiers). Before creating a new segment, the program checks the size of the stack. If the stack is empty, a new segment will be mapped.

8. If you have to create new abstractions what are their design checklists?

An abstraction was created to represent the segmented memory. The checklist for this abstraction is defined in Part 2.

9. What invariant properties should hold during the solution of the problem?

- All operations are executed within eight 32-bit registers.
- All 32-bit words contain one operation, identified by the op code. Each op code represents a distinct instruction.
- An instruction contain 3 bits that represent location of 3 registers or 1 register.
- Each offset identifier of a segment stores an associated 32-bit word.

Design Checklist for HW6: Universal Virtual Machine

- Each register can be identified by 3 bits, and vice versa (each 3 bit value represents a distinct register).
- Each operation can be identified by 4 bits, and vice versa (each 4 bit value represents a distinct operation).

10. What algorithms might help solve the problem?

- Bitfield manipulations (shift, AND, OR, masks).
- Segment 0 will hold the program, therefore the order that the instructions will be executed will be dictated by this segment. The program counter in this segment is advanced to the next instruction (by adding an offset of 1 for the position that the pointer points to), which will then be executed.

11. What are the major components of your program, and what are their interfaces?

- Registers are 32-bit words stored in an array of size 8. Each of register's value is determined by the instruction executions.
- Instructions are 32-bit words. Each 32-bit word holds and op code in the 4 most significant bits, and identifiers for 3 registers in the 9 least significant bits with 3 bits representing 1 register. An exception is loadval (op code 13), which will have 3 bits following the op code representing one register and the remaining bits representing the value.
- The Hanson sequence interface will be used as the container to store segments. Each segment will store the instructions.
- Each segment will store one array, and each index of the array will store one 32-bit word.
- The program counter is a 32-bit word. The program counter points to next instruction to be executed.
- Stack (Using Hanson's sequence interface) will hold location of unmapped segments (ie. the segment identifier).

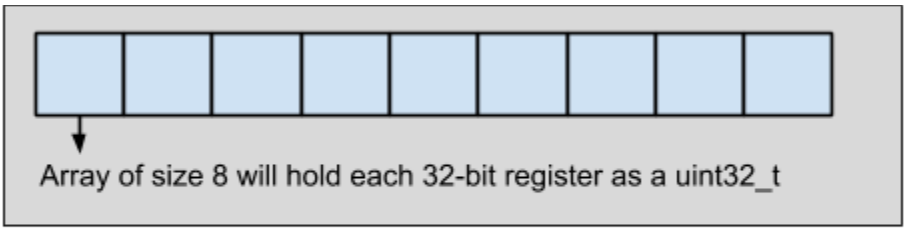
12. How do the components in your program interact? That is, what is the architecture of your program?

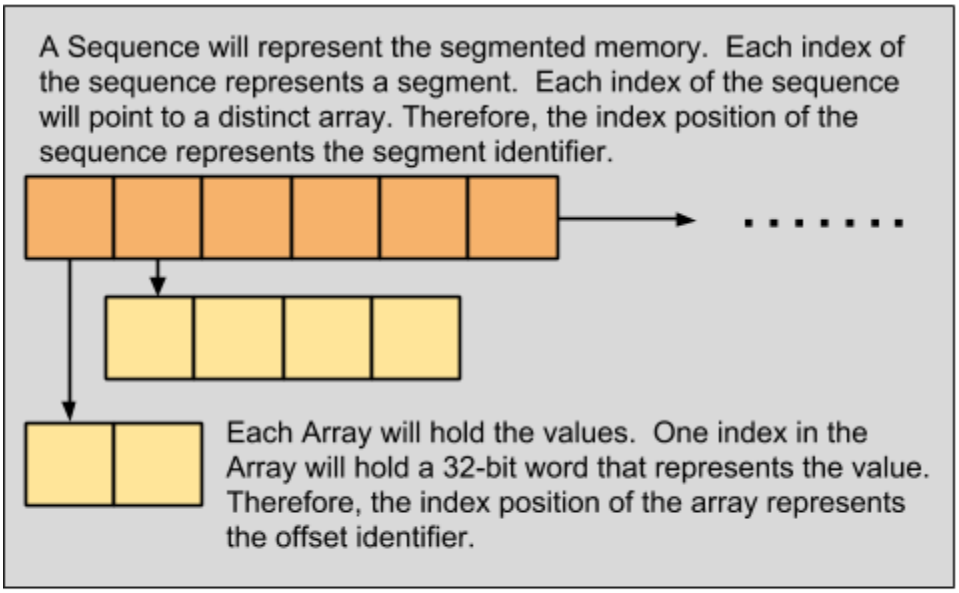
The program creates a Hanson sequence to represent the segmented memory. The program then reads in 32 bits at a time via getc(). Subsequently, the program creates an array with a size specified by map instruction (op code 13) and inserts the array into an unmapped segment in the sequence. The map instruction dictates the number of words per segment, therefore the size of the array is known at runtime. The segment identifier is determined by its index in the sequence. The offset identifier is determined by its index in the array. Each word is mapped to a segment identifier and an offset identifier, and vice versa (ie. each segment and offset identifier maps to a distinct word). The

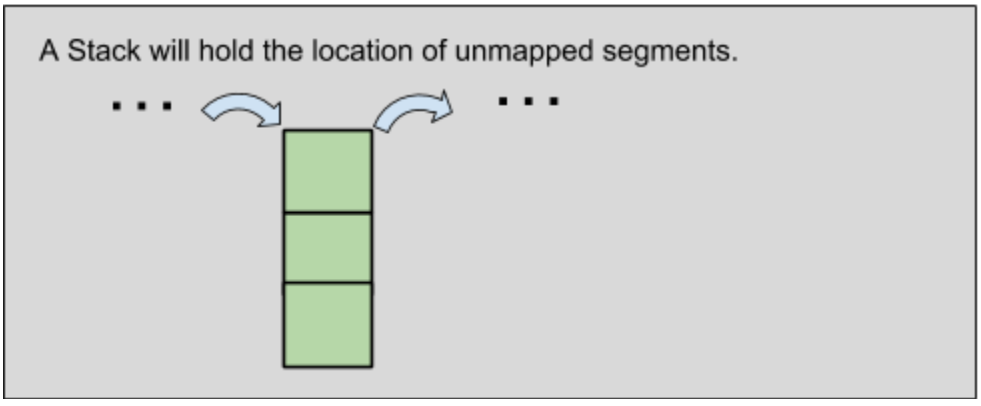
Design Checklist for HW6: Universal Virtual Machine

segment with identifier 0 holds the program to be executed, therefore the entry point is this segment. The instructions are executed in order as dictated by the program, starting at index 0. To advance the program counter, an offset of 1 (word) will be added to the counter.

Images 1, 2, & 3. Components and Representations







Design Checklist for HW6: Universal Virtual Machine

13. What test cases will you use to convince yourself your program works?

To ensure that the program works, smaller C programs will be created that will execute an analogous of each instruction and output the correct results. These small C programs will be converted to machine code and the UM will input the codes. The outputs of the UM will be compared with the outputs of the original C programs. Below are test cases for each instruction:

1. **Conditional Move** (op code = 0): a C program will be created to load register 1 and register 2 with specified 32-bit values. The program will check if the value in register 1 is not equal to zero. If so, register 3 will be assigned the value of register 2.
    a. Test if value in register 1 is equal to zero
    b. Test on non-existing registers (ex. register 9)

2. **Segmented Load** (op code = 1): a C program will be created to insert a known value to a specified segment and offset. Then register 1 will be assigned this value utilizing a call to extract the value stored in the segment and offset identifiers.

3. **Segmented Store** (op code = 2): a C program will be to insert a value into register 1. An empty segment and offset will be created on a specified segment and offset identifier, and register 1's value will be loaded utilizing extraction of the value from the specified segment and offset identifier.

4. **Add Instruction** (op code = 3)**:** a C program will be created to add two numbers and output the result into a result file. This program will then be compiled and the program's machine code will be saved in a separate file. The file will then be inputted into the UM and the UM will output the results in a separate file. To compare the results of the original C program and the UM, the `diff` command will be invoked for the output files.
    a. Add two positive integers
    b. Add two negative integers
    c. Add one positive integer and one negative integer
    d. Add two floating-point numbers (two positive / two negative / one negative and one positive / one very large and one very small).

5. **Multiply Instruction** (op code = 4): a C program will be created to multiply two numbers and output the result into a result file. This program will then be compiled and the program's machine code will be saved in a separate file. The file will then be inputted into the UM and the UM will output the results in a separate file. To compare the results of the original C program and the UM, the `diff` command will be invoked for the output files.
    a. Multiply two positive integers.

Design Checklist for HW6: Universal Virtual Machine

   b. Multiply two negative integers.
   c. Multiple one positive integer and one negative integer.
   d. Multiply two floating-point numbers (two positive / two negative / one negative and one positive / one very large and one very small).
   e. Multiply one number and 0. Multiply two 0's.

6. **Divide Instruction** (op code = 5): a C program will be created to divide two numbers and output the result into a result file. This program will then be compiled and the program's machine code will be saved in a separate file. The file will then be inputted into the UM and the UM will output the results in a separate file. To compare the results of the original C program and the UM, the `diff` command will be invoked for the output files.
   a. Divide a number by 0 (expect error). Divide 0 by a number. Divide 0 by 0 (expect error).
   b. Divide two positive integers.
   c. Divide two negative integers.
   d. Divide one positive integer and one negative integer.
   e. Divide two floating-point numbers (two positive / two negative / one negative and one positive / one very large and one very small).

7. **Bitwise NAND** (op code = 6): a C program will be created use the bitwise AND, and flipping the resulting binary. The result will be outputted into a result file. The file will then be inputted into the UM and the UM will output the results in a separate file. To compare the results of the original C program and the UM, the `diff` command will be invoked for the output files.
   a. Bitwise NAND compare two zero numbers.
   b. Bitwise NAND compare the number 0 with a non-zero number.
   c. Bitwise NAND compare positive number and negative number.
   d. Test the above on both integers and floating-point numbers.

8. **Halt** (op code = 7): Create a .um file containing an output instruction both before and after a Halt instruction. Run the file through the UM and ensure the program halts before the second output statement.

9. **Map Segment** (op code = 8): this test was described in the Universal Machine Segments checklist in the subsequent sections of this document and the function prototype was submitted with this design checklist.

10. **Unmap Segment** (op code = 9): this test was described in the Universal Machine Segments checklist in the subsequent sections of this document and the function prototype was submitted with this design checklist.

Jay Facultad
Nathan Sahelijo

Design Checklist for HW6: Universal Virtual Machine

11. **Output** (op code = 10): a C program will be created to print a known value to a separate file. This program will then be compiled and the program's machine code will be saved in a separate file. The file will then be inputted into the UM and the UM will output the results in a separate file. To compare the results of the original C program and the UM, the `diff` command will be invoked for the output files.

12. **Load Program** (op code = 11): a C program will be created to create a segment with know values. The load program will then duplicate this segment and assign it to segment 0. The test will print the values of the original segment and the values in segment 0. The values for both segments must be the same.

13. **Load Value** (op code = 13): a C program will be created to load a known value to a mapped segment and offset. This value within the specified segment and offset identifier will then be outputted to a separate file. This program will be compiled and the program's machine code will be saved in a separate file. The file will then be inputted into the UM and the UM will output the results in a separate file. To compare the results of the original C program and the UM, the `diff` command will be invoked for the output files.

14. What arguments will you use to convince a skeptical audience that your program works?

The UM's outputted files must match the analogous C program's output files by 100%. If any results yield below 100%, then the program failed.

(see next page for Part 2, checklist for Universal Machine Segments)

Jay Facultad
Nathan Sahelijo

Design Checklist for HW6: Universal Virtual Machine

Part 2: Universal Machine Segments

1. ***What is the abstract thing you are trying to represent?***

   We are trying to represent multiple containers of different sizes that store information.

2. ***What functions will you offer, and what are the contracts those must meet?***

   1. Function that maps to the offset within a segment
      a. Contract: The values stored will have a specific segment identifier as well as an identifier for the value itself within its segment. Values cannot be larger than the segment it is being stored in.
   2. Function that maps new segments or puts new data to existing segments (if unmapped).
      a. Contract: A unique 32-bit identifier maps specifically to that segment.
   3. Function that unmaps segments and updates stack.
   4. Functions that manipulate bitfields.

3. ***What examples do you have of what the functions are supposed to do?***

   1. Extracting values from a segment, replacing values in the segment, segment load, segment store.
   2. Used for Map Segment operations.
   3. Used for Unmap Segment operations.
   4. Used multiplication, division, bitwise NAND.

4. ***What representation will you use, and what invariants will it satisfy?***

   Sequence to represent memory. The sequence will hold pointers to segments. The segments will be represented as static arrays. Each index of the array will hold a 32-bit word. Stack will store location of unmapped segments (ie. segment identifiers). If a segment were to be unmapped, the array pointed to in its segment index will be deallocated. Before creating a new segment, the program checks index locations in the sequence. If stack is empty, a new segment will be mapped.

   (See Figures 1, 2, & 3 on page 4 for visual representations)

Jay Facultad
Nathan Sahelijo

Design Checklist for HW6: Universal Virtual Machine

5. How does an object in your representation correspond to an object in the world of ideas?

The arrays represent the containers that hold the machine instructions, and the sequence represents the containers that hold the arrays.

6. What test cases have you devised?

- Test mapping (create segment) by checking to see if the stack (ADT) which store the index values of the unmapped segments contains any items. If the stack contains any items, pop the stack and use the popped index value to point to the new data. Else if the stack is empty, create a new segment for the new data.
- Test unmapping of a segment by changing pointer in the specified sequence index to point to nullptr. Then push the specified sequence's index value to the stack (ADT).
- Test that words were correctly inserted by creating an array filled with values and inserting this array into the sequence into a specified index position. Then print the array values stored in that particular sequence index position.
- Test size of sequence by printing length of sequence.
- Test size of array by printing length of array.
- The tests above will confirm that the sequence identifier for a particular segment was correctly implemented.
- A function prototype was also created to obtain a 32-bit word stored in the sequence offset. This test will confirm that the sequence identifier and offset identifier for a particular 32-bit word were correctly implemented.

7. What programming idioms will you need?

- Idiom for casting data types, specifically assignments between pointers and integers of differing sizes.
- The idiom for handling void * functions of known type.