Jack Burton jburto05
James Hartley jhartl01

um Design Document

Architecture

Modules

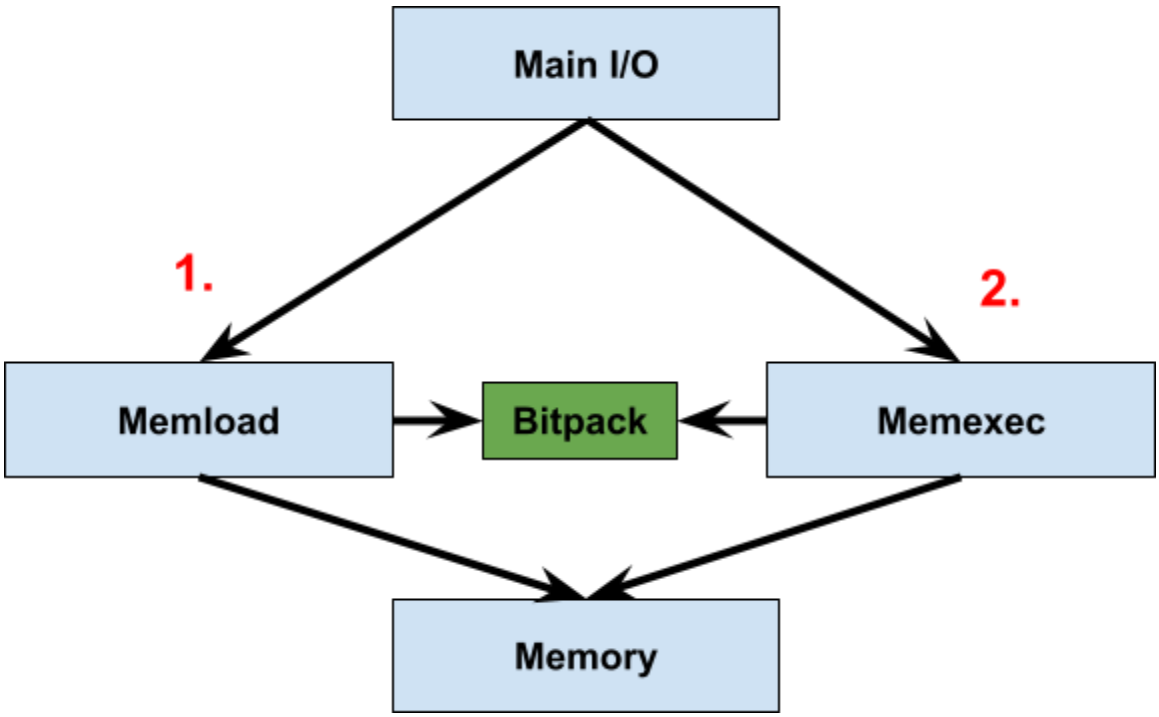
a. **Main I/O**
   i. In charge of opening .um file for reading, outputting to stdout if applicable

b. **Memload**
   i. In charge of loading um instructions into segmented memory

c. **Memexec**
   i. In charge of executing um instructions from segmented memory

d. **Mem**
   i. In charge of getting & setting memory in registers and segments

e. **Bitpack**
   i. In charge of providing helper functions to pack/encode and unpack/decode memory



Arrows represent "uses" relationships

- The Main I/O module will take an input .um file, and open it for reading. This file will be passed to the Memload module.
- The Memload module will read instructions from the file, bitpack them (Bitpack module), and load them into the segmented memory (Memory module).
- The Memexec module will read and execute instructions from the segmented memory

Hanson Structures

- Seq_T of Seq_t of uint32_t
    - This will allow us to segment the memory into reusable 32-bit blocks, each of which can store a sequence of 32-bit words
- UArray_T of uint32_t
    - This will allow us to store the 8 registers distinctly from the segmented memory

Module Interaction *(as function contracts)*

Main I/O Module

```
/********** main ********
 *
 * Opens file for reading, passes file pointer to memload,
 * then executes instructions with memexec.
 *
 * Parameters:
 *     int argc: Number of arguments passed via stdin
 *     char *argv[]: Arguments from stdin
 *
 * Return: None
 *
 * Expects
 *    argc == 2
 *     argv[1] to be a valid, readable .um file
 * Notes
 *    Provides appropriate error messages and exits with EXIT_FAILURE
 *    if expectations are not met.
 *
 ************************/
```

Memload Module

```
/********** loadInstructions ********
```

```
 *
 * Read through .um file, bitpacking each instruction and
 * pushing each to end of segment zero
 *
 * Parameters:
 *     Mem_T memory: Pointer to initialized empty memory
 *     FILE *fp: Pointer to opened .um file
 *
 * Return: None
 *
 * Expects
 *    Non-NULL pointer to valid .um file
 * Notes
 *    Uses Bitpack module to encode each instruction
 *
 ************************/
```

Memexec Module

```
/********** execInstructions ********
 *
 * Read through segment zero, iteratively executing each instruction
 *
 * Parameters:
 *     Mem_T Memory: Pointer to memory struct
 *
 * Return: None
 *
 * Expects
 *    Non-NULL pointer to initialized memory struct
 * Notes
 *   Unchecked Runtime Error for
 *       - Word to to not code for a valid instruction
 *       - Segmented load/store to refer to unmapped segment
 *     - Segment unmap to unmap $m[0] or an unmapped segment
 *     - Dividing by zero
 *     - Instruction loads program from unmapped segment
 *     - Instruction outputs value > 255
 *
 ************************/
```

execInstructions will call helper functions for each opcode

Memory Module

```
typedef Seq_T Seg_T;
typedef UArray_T Reg_T;

typedef struct Mem_T {
     Seg_T seg;
     Reg_T reg;
} *Mem_T;


/********** initMem ********
 *
 * Creates new empty Mem_T struct
 *
 * Parameters:
 *     None
 *
 *
 * Return: pointer to initialized Mem_T struct
 *
 ************************/

/********** freeMem ********
 *
 * Frees segmented memory and registers given a memory object
 *
 * Parameters:
 *     Mem_T memory: Pointer to memory struct
 *
 * Return: None
 *
 ************************/

/********** getMem ********
 *
 * Gets 32-bit word at $m[address][offset]
 *
```

```
 * Parameters:
 *     Mem_T memory: Pointer to memory struct
 *     uint32_t address: specifies segment address in memory
 *     int offset: specifies offset in memory
 *
 *
 * Return: uint32_t word
 *
 * Expects
 *      offset % 32 == 0
 ************************/

/********** getReg ********
 *
 * Gets 32-bit word at $r[offset]
 *
 * Parameters:
 *     UArray_T registers: Pointer to array of registers
 *     int index: specifies index in registers uarray
 *
 *
 * Return: uint32_t word
 *
 * Expects
 *      0 <= index <= 7
 *
 ************************/


/********** setMem ********
 *
 * Sets $m[address][offset] to “word”
 *
 * Parameters:
 *     Mem_T memory: Pointer to memory struct
 *     uint32_t word: Bitpacked instruction
 *     uint32_t address: specifies segment address in memory
 *     int offset: specifies offset in memory
 *
```

```
 * Return: None
 *
 * Expects
 *      offset % 32 == 0
 *
 ************************/

/********** setReg ********
 *
 * Sets $r[index] to “word”
 *
 * Parameters:
 *     Mem_T memory: Pointer to memory struct
 *     uint32_t word: Bitpacked instruction
 *     int index: index of register in uarray
 *
 *
 * Return: None
 *
 * Expects
 *     0 <= index <= 7
 *
 ************************/
```

Implementation Plan

Main I/O:

1. Create Main I/O module with code to open file specified from argv[1]

Memory:

2. Create Register typedef abstraction Reg_T
    a. Ensure data is properly added and removed from Reg_T structure
3. Create Segmented Memory typedef abstraction *Seg_T
    a. Ensure Seq_addlo and addhi properly add data to our Seg_T structure
4. Create Mem_T struct containing Reg_T and Seg_T
    a. Ensure access and updates work to both objects in struct
5. Create init function to create new Mem_T struct

    a. Access and update memory/register values in initialized Mem_T struct, ensuring it was created properly
6. Create function to free Mem_T
    a. Put data into Mem_T's registers and segments, and use valgrind to ensure all freed after function call
7. Create get and set functions for Reg_T
    a. Test if getting returns a valid uInt32 from memory
    b. Test if getting returns correct uInt32 after setting to memory
8. Create get and set functions for Seg_T
    a. Test if getting returns a valid uint32_t from memory
    b. Test if getting returns correct uint32_t after setting it in memory
9. Test edge cases
    a. Accessing uninitialized Seg_T/Reg_T
    b. Passing NULL pointers into function arguments

Memload:

10. Read in from file pointer from main
    a. Print value from file to ensure reading properly
11. Create loop to read until end of file, line by line
    a. Test that all values from file are printed line by line
12. Use Bitpack module to pack each line into 32-bit word
    a. Print these words after bitpacking, manually decoding & ensuring correct binary values
13. Use Memory module to add each line to segment 0 in Seg_T using set
    a. Print Seg_T segment 0 values using getMem and ensure all words are put into segment in big-endian order

Memexec:

14. Create exec function which reads in words from segment 0 in Seg_T, and decodes them using Bitpack's getu
    a. Print these words to stderr, ensuring they match the original input file words
15. Write helper functions for each of the 14 commands
    a. *Provide sample input values for registers to input functions, and add line in function to print result register. Ensure correct result for respective operations in put into result register.Further testing details in UM Segment Abstraction portion of Testing section.*
16. Modify command functions to access/update Reg_T values
17. Create switch table for opcodes in exec function, running corresponding helper function for each command

Main I/O:

18. Use Memory module initMem to create new empty memory after file opened
19. Pass this memory as argument to Memload
    a. Print values from segment 0 after Memload using Memory module get, ensuring they loaded properly
20. Pass memory as argument to memexec
    a. Ensure all instructions are executed properly

21. Call memory free function
    a. Ensure valgrind passes with no lost memory

Testing Plan

UM Instruction Set

a. halt.um
    i. Only contains halt instruction
    ii. Ensure no leaked memory and exit occurs properly

b. halt-verbose.um
    i. Contains halt instruction them many other instructions after
    ii. Ensure no instructions after halt are executed, and no leaked memory

c. print-six.um
    i. Contains addition that leads to value of 6 in register C
    ii. Ensure 6 is outputted to terminal

d. empty.um
    i. Empty UM file
    ii. Ensure CRE thrown

e. smallmath.um
    i. Tests each math function (addition, multiplication, division) with small number values and output results
    ii. Ensure output values are as expected

f. mapunmapsegs.um
    i. Maps 10 new segments, puts math-computed values into them, then unmaps each segment.
    ii. Ensure no memory leaks or errors

g. bitwisesim.um
    i. Maps new memory segment. Simulates bitwise XOR, AND, OR, and NOT (and NAND) by using the NAND command. Runs these on math-computed register values, and stores them in new memory segment. Prints from segment, and then unmaps it.
    ii. Ensure outputted results are correct

UM Segment Abstraction

```
void testLoadValue(Mem_T memory)
// Store value from 32-bit word into register, then assert that
retrieving that value from register gets correct value
```

```
void testInOut(Mem_T memory)
// Store value from stdin into register, then assert that
outputting value from register gets correct value

void testSegLoadStore(Mem_T memory)
// Store value at specific offset in segment 0 with segmented
store, then retrieve value at segment 0 at that offset using
segmented load. Ensure that it is the original value.

void testMathFunctions(Mem_T memory)
// Load values 10 and 5 into r[B] and r[C], and perform addition,
multiplication, and division, outputting r[A] after each
operation. Ensure 15, 50, and 2 are outputted.

void testBitwiseOp(Mem_T memory)
// Load value 0xffffffff into r[B] and r[C] and NAND, ensuring
r[A] is 0 after computation. Load other hex values and ensure bit
operation returns correct value. Combine NAND operators to create
AND, and ensure it produces expected values from r[B] and r[C].

void testMapUnmapSeg(Mem_T memory)
// Load values into r[B] and r[C], and map a new segment. Put
these register values into new segment, print them from that
segment, then unmap the segment. Ensure correct values are
printed, and valgrind to ensure unmapped segment memory is freed.

void testLoadProgram(Mem_T memory)
// Call load program with one instruction, ensuring that it is
executed (i.e. it was placed into segment 0).
```