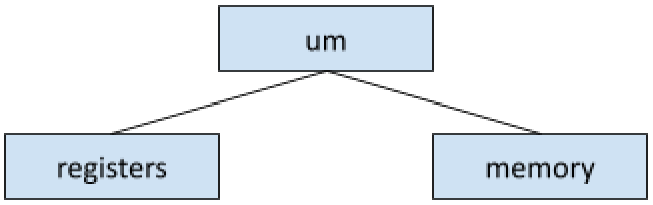
design.pdf
Justin Paik (jpaik03)
10 November, 2025
um

Architecture

1. Modules
   I will break my implementation into modules by functionality. I plan to make three modules for this assignment, including:
   - **um**
     - Uses stdin and stdout to handle I/O of unsigned 8-bit characters to implement the UM's instructions. Displays ASCII characters.
     - Handles the 32-bit program counter.
   - **registers**
     - Implements the eight general-purpose registers, each holding one 32-bit word.
   - **memory**
     - Implements the address space of word-oriented memory segments, each containing a sequence of words referred to by a 32-bit identifier.
     - Handles memory operations.
2. Hanson data structures
   - **um**
     - I do not plan to use Hanson data structures in this module.
   - **registers**
     - I plan to use one Hanson **Array** to represent the register file.
       - An Array makes sense here because the register file is not dynamic.
       - The Array will be initialized to hold eight uint32_t (32-bit) words (i.e., size 8, element type uint32_t).
   - **memory**
     - I plan to use a Hanson **Sequence** to map 32-bit segment identifiers to corresponding memory segments.
       - A Seq makes sense here because the collection of memory segments is dynamic.
       - The index of the Seq will represent the segment ID, and the element will be a Hanson **Array** (see next).
     - I plan to use a Hanson **Array** to represent each segment.
       - The Array will be initialized to hold *n* uint32_t (32-bit) words (i.e., size *n*, element type uint32_t), where *n* is determined by its Map Segment instruction in opcode 8.

    - I plan to use another Hanson **Sequence** as a stack to store and get reusable 32-bit segment identifiers from unmapped segments.
        - This will allow the UM to recycle IDs efficiently.

3. Module interaction



- **um** will:
    - read the .um file and pass it to the memory module to create segment 0.
    - call the `memory` module to get the next instruction from $m[0] using the program counter.
    - decode the instruction using the bitpack library to get the opcode and register numbers.
    - call the `registers` module (functions like `registers_get` or `registers_set`) and the `memory` module (functions like `memory_get`, `memory_put`, or `memory_map`) to execute the instruction.
    - directly handle the input and output instructions, using stdin and stdout.
- **registers** will:
    - only respond to calls from `um`.
- **memory** will:
    - only respond to calls from `um`.

4. Inter-module function prototypes

- **um**
    - Calls all of the functions below.
- **registers** (assume "typedef struct Registers_T *Registers_T")

```
Registers_T regs_new();
void regs_free(Registers_T *registers);
uint32_t regs_get(Registers_T registers, int reg);
void regs_set(Registers_T registers, int reg, uint32_t val);
```

- **memory** (assume "typedef struct Memory_T *Memory_T;")

```
Memory_T mem_new(FILE *fp);
void mem_free(Memory_T *mem);
uint32_t mem_get(Memory_T mem, uint32_t segId, uint32_t offset);
void mem_put(Memory_T mem, uint32_t segId, uint32_t offset,
             uint32_t val);
uint32_t mem_map(Memory_T mem, uint32_t numWords);
void mem_unmap(Memory_T mem, uint32_t segId);
void mem_load(Memory_T mem, uint32_t segId);
```

5. Function contracts (for three most important functions)

```
/******** mem_map ********
 *
 * Implements the 'Map Segment' instruction (opcode 8). Creates a new segment
 * with a specified number of words, all initialized to 0.
 *
 * Parameters:
 *      Memory_T mem:           The memory module instance
 *      uint32_t numWords:      The number of 32-bit words for the new segment
 * Returns:
 *      A 32-bit segment identifier for the new segment. This ID is guaranteed
 *      not to be 0 and not to identify any other currently mapped segment.
 * Expects:
 *      'mem' is not NULL.
 *      'mem' is a valid Memory_T instance.
 * Notes:
 *      This function attempts to get a recycled ID from the free list stack.
 *      If none is available, it will use the next available ID.
 *      Throws a CRE if 'mem' is NULL.
 *      Halts execution with a CRE if memory allocation fails.
 ************************/
uint32_t mem_map(Memory_T mem, uint32_t numWords);

/******** mem_unmap ********
 *
 * Implements the 'Unmap Segment' instruction (opcode 9). Frees the specified
 * segment and adds its ID to the free list for recycling.
 *
 * Parameters:
 *      Memory_T mem:   The memory module instance
 *      uint32_t segId: The 32-bit identifier of the segment to unmap
 * Returns:
 *      Nothing.
 * Expects:
 *      'mem' is not NULL.
 *      'mem' is a valid Memory_T instance.
 * Notes:
 *      The 'segId' is added to the free list to be reused by a future
'mem_map'
 *      call.
 *      Throws a CRE if 'mem' is NULL.
 *      Throws an unchecked run-time error if 'segId' is 0 or refers to a
 *      segment that is not currently mapped.
 ************************/
void mem_unmap(Memory_T mem, uint32_t segId);

/******** mem_load ********
 *
 * Implements the 'Load Program' instruction (opcode 12). Duplicates the
segment
```

```
 * at 'segId' and replaces segment 0 with the duplicate.
 *
 * Parameters:
 *      Memory_T mem:   The memory module instance
 *      uint32_t segId: The 32-bit identifier of the segment to load
 * Returns:
 *      Nothing.
 * Expects:
 *      'mem' is not NULL.
 *      'mem' is a valid Memory_T instance.
 * Notes:
 *      If 'segId' is not 0, the existing segment 0 is unmapped, the segment at
 *      'segId' is duplicated, and the new duplicate segment is mapped as
 *      segment 0.
 *      Throws a CRE if 'mem' is NULL.
 *      Throws an unchecked run-time error if 'segId' refers to a segment that
 *      is not currently mapped.
 ************************/
void mem_load(Memory_T mem, uint32_t segId);
```

Implementation Plan

1. Set up modules
    - Create files: um.c, registers.c, registers.h, memory.c, memory.h.
        - Add main function to um.c.
    - Update the Makefile.
    - Testing
        - Run “`make`”. This should compile and link to create the um executable.
        - Run “`valgrind ./um`”. There should be no errors or leaks.
2. Implement registers module
    - Implement the registers module using Hanson Arrays.
        - This includes `regs_new`, `regs_free`, `regs_get`, and `regs_set`.
    - Testing
        - Create a function that acts as a unit test for the registers module.
        - This test will
            1. Call `regs_new`.
            2. Call `regs_set` to put a hard-coded value into a register.
            3. Call `regs_get` for r3 and verify that the returned value matches the hard-coded value.
            4. Call `regs_get` for an un-set register and verify its value is 0.
            5. Call `regs_free` and run using valgrind to ensure there are no leaks.
3. Implement memory module related to bitpack
    - Implement `mem_new` to read the .um file, create segment 0 as a Hanson Array, and store it in the Hanson Sequence.

- Partially implement `mem_get` to work for segment 0.
- Testing
    - Write a test in main that calls `mem_new` with my own .um file.
    - This test will
        - Call `mem_get(mem, 0, 0)` to get the first instruction.
        - Use `Bitpack_getu` to extract the opcode from the instruction.
        - Assert that the extracted opcode matches the known opcode of the test file.
    - Run with valgrind to check for leaks in `mem_new`.

4. Implement opcodes Halt, Load Value, and Output
    - Implement the main loop in um.c.
    - Implement logic for:
        - opcode 7 (Halt)
        - opcode 13 (Load Value)
        - opcode 10 (Output)
    - Testing
        - Create an instruction set test halt.um.
            - This test will:
                1. Contain a single Halt instruction.
                2. Be run using `./um halt.um`. The program should exit cleanly without errors.
        - Create an instruction set test output.um.
            - This test will:
                1. Load Value r1, 65 (ASCII ‘A’).
                2. Output r1.
                3. Halt.
        - Run `./um output.um > a.txt` and check that a.txt contains “A”.
    - Run all tests with valgrind.
5. Implement arithmetic and logic opcodes Addition, Multiplication, Division, Bitwise NAND, Conditional Move
    - Implement:
        - opcode 3 (Addition)
        - opcode 4 (Multiplication)
        - opcode 5 (Division)
        - opcode 6 (Bitwise NAND)
        - opcode 0 (Conditional Move)
    - Testing
        - Create an instruction set test for each new opcode.
            - This test will:
                1. Load necessary values into registers.

                2. Perform the operation.
                3. Output the register containing the result.
                4. diff the output against an expected value.
        - Create a divide_by_zero.um edge case test.
            - This test will:
                1. Attempt to divide a value by 0.
                2. Verify the program is allowed to crash.
    - Run all tests with valgrind.
6. Complete memory module and implement other opcodes: Segmented Load, Segmented Store, Map Segment, Unmap Segment
    - Implement:
        - `mem_map`, `mem_unmap`, `mem_put`, and the rest of `mem_get`.
        - The Seq_T stack for recycling 32-bit segment identifiers.
        - opcode 1 (Segmented Load)
        - opcode 2 (Segmented Store)
        - opcode 8 (Map Segment)
        - opcode 9 (Unmap Segment)
    - Testing
        - Create an instruction set test mem.um.
            - This test will:
                1. Map a new segment.
                2. Store a value in the new segment.
                3. Load the value from that segment into a new register.
                4. Output the new register and verify the value.
        - Create a test to Map, Unmap, and Map again to verify the IDs have been recycled as expected.
        - Create tests for allowed failures from the spec.
    - Run all tests with valgrind.
7. Complete opcodes by implementing Input and Load Program.
    - Implement:
        - opcode 11 (Input)
        - opcode 12 (Load Program)
            - Includes implementing `mem_load` in the memory module.
            - The um.c loop must update the program counter.
    - Testing
        - Create an instruction set test input.um.
            - This test will:
                1. Call Input and store the value in a register.
                2. Call Output on that register.
                3. Verify that the output is as expected.

        - Create an instruction set test load_program.um.
            - This test will:
                1. Map a new segment and store a simple program into it.
                2. Call Load Program on the new segment's ID.
                3. The program should automatically output and halt.
    - Run all tests with valgrind.
8. Final Testing and Performance
    - Run all provided test binaries.
    - Time the midmark benchmark.
        - This test must complete in roughly under 60 seconds.
    - Write the README.
    - Testing
        - Run valgrind on all tests, to ensure that there are no memory leaks.
    - Verify the program exits with EXIT_SUCCESS on all valid tests.

Testing Plan (architecture tests)

- The instruction set tests are interleaved in the implementation plan.
- The architecture tests are below.

```
/******** test_regs_new ********
 *
 * This test verifies the initialization of the registers module.
 *
 * It tests:
 *      1. regs_new: Asserts that a new register file is not NULL.
 *      2. Initialization: Asserts that all 8 registers are initialized to 0.
 ************************/
void test_regs_new();

/******** test_regs_set_get ********
 *
 * This test verifies the set and get functionality of the registers module.
 *
 * It tests:
 *      1. regs_set: Sets a given value in a register.
 *      2. regs_get: Asserts that the get value matches the set value.
 ************************/
void test_regs_set_get();

/******** test_mem_map ********
 *
 * This test verifies that a new segment is mapped correctly.
 *
 * It tests:
 *      1. mem_map: Maps a new segment of a certain (non-zero) size.
 *      2. Return Value: Asserts the returned segment ID is not 0.
 ************************/
```

```
void test_mem_map();

/******** test_mem_put_get ********
 *
 * This test verifies the ability to read and write from a mapped segment.
 *
 * It tests:
 *      1. mem_map: Maps a new segment.
 *      2. mem_put: Stores a known value at a valid offset.
 *      3. mem_get: Retrieves the value and asserts it is as expected.
 ************************/
void test_mem_put_get();

/******** test_mem_recycle_id ********
 *
 * This test verifies that segment IDs are recycled after being unmapped.
 *
 * It tests:
 *      1. mem_map: Maps a segment and gets an ID.
 *      2. mem_unmap: Unmaps that segment.
 *      3. mem_map: Maps a new segment and asserts its ID is the same as the
 *                  first map call, verifying the ID was recycled.
 ************************/
void test_mem_recycle_id();
```

Loop Invariant

- **Invariant**: At the beginning of each fetch cycle, the program counter (pc) holds the offset of the next instruction to be executed in segment 0. All machine state (the contents of all eight registers and of all mapped memory segments) is the correct and final result of having executed every instruction from $m[0][0] up to and including $m[0][pc - 1].
- **Justification**
    - Initialization: When the loop is first entered, pc is 0. The set of executed instructions ($m[0][0]...$m[0][pc - 1]) is empty. The initial machine state (all registers are 0, segment 0 contains the program) is the correct result of having executed zero instructions. The invariant holds.
    - Maintenance:
        1. The instruction at $m[0][pc] is fetched.
        2. The program counter is advanced to pc + 1.
        3. The fetched instruction (from the old pc) is executed, modifying the machine state.
        4. This execution correctly transitions the machine state from the result of instructions 0 through pc - 1 to the result of instructions 0 through pc.
        5. When the next iteration begins, the new program counter (whose value is pc + 1) points to the next instruction, and the machine state is the correct result of all instructions before it (from 0 to pc). The invariant is reestablished.

- **Termination**: The loop terminates when a Halt instruction is executed. At this point, the invariant holds. The pc points past the Halt instruction, and the machine state is the correct and final result of the entire program’s execution.