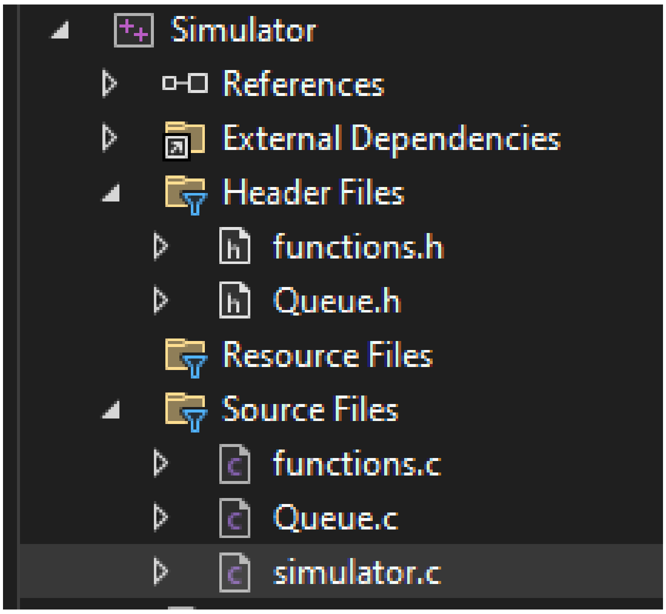
Computer Organization Project - Implementing SIMP Processor

Yarden Pardo 206556144
Eran Hirschberg 209649003

Simulator:

We implemented a simulator that receives an output of a program to the memory, performs action on various registers defined in the ISA and outputs to various I/O elements.

Directory Tree:



Queue.h

Node structure
typedef struct Node {
    int data;
    struct Node* next;
} Node;

Queue structure
typedef struct Queue {
    Node* front;
    Node* rear;
} Queue;

Function to create a new node
Node* createNode(int data);

Function to initialize a new queue
Queue* createQueue();

Function to check if the queue is empty
int isQueueEmpty(Queue* queue);

Function to enqueue (add) an element to the queue
void enqueue(Queue* queue, int data);

Function to dequeue (remove) an element from the queue
int dequeue(Queue* queue);

Function to get the front element of the queue without dequeuing
int peek(Queue* queue);

Function to print the queue elements
void printQueue(Queue* queue);

Function to free the queue
void freeQueue(Queue* queue);

Functions.h

----------- Data Structures ----------->

```
typedef struct ParsedLine {
	unsigned int opcode;
	unsigned int rd;
	unsigned int rs;
	unsigned int rt;
	int imm;
	char i_format;
}PL;
```

input: file pointer and memory array pointer
output: function reads memin to memory
void read_memin_to_memory(FILE* memin, int* memory);

input: a Hexadecimal representation of a number as a string
output: returns corresponding int value
int convert_hexStr_to_int(char* hex_str);

input: ParsedLine pointer and integer representing the instruction line
output: parsing the instruction in to structure ParsedLine
void parse_line(PL* new, int line);

input: ParsedLine pointer
output: returns 1 if immediate instruction, 0 otherwise
int is_I_Format(PL* inst);

input: an integer and a starting and ending bit (inclusive end)
output: returns the equivalent integer of the binary representation of the bit range
int extract_bits_range_sum(int sequence, int start, int end);

input: an integer and a bit index
output: returns the bit in the index
int extract_bit(int sequence, int bit);

input: an integer representing the immediate value from memory
output: returns the number in its 2's complement representation in bits 0-19
int read_imm(int sequence);

input: an integer
output: returns itsnegative 2's complement representation
int extract_negative_2s_comp(int sequence);

input: the instruction structure and other relevant parameters
output: function executes the instruction according to ISA
void execute_instruction(PL* inst, int* mem, int* reg, int* pc, int* pc_updated,
	int* hw_reg, int* hw_updated, int* leds_changed, int* disp_7seg_changed,
	int* disk_action);

*input: *pointer to* file pointer, local path, and reading mode*
output: function opens the file, checks for null file pointer and returns 0 for success,1 otherwise
int open_file_check(FILE** fp, char* file_name, char* mode);

input: pointer to memout and pointer to memory array
output: writes the memory to memout
void write_memory_to_memout(FILE* memout, int* memory);

initiating an array of hardware registers
void init_hw_reg(int* hw_reg);

initiating an array of strings of the names of hardware registers
void init_hwreg_labels(char** hwreg_labels);

updates trace file with current registers
void update_trace(FILE* trace, int PC, int inst_dec, int* registers, PL* inst);

updates hwregtrace file with current hardware registers
void update_hwregtrace(FILE* hwregtrace, char* file_name, PL* inst, int* reg, int* hw_reg, int cycle);

writes content of register array to regout file
void write_registers_to_regout(FILE* regout, int* registers);

initiating array of registers
void init_registers(int* reg);

initialize frame buffer matrix to 0
void init_framebuffer(unsigned char framebuffer[][X_PIXELS]);

update a single pixel in framebuffer
void update_framebuffer(unsigned char fb[][X_PIXELS], unsigned short px_addr, unsigned char px_data);

copy framebuffer content to monitor file
void write_framebuffer_to_monitor(unsigned char fb[][X_PIXELS], FILE* monitor, FILE* monitor_yuv);

read irqin file to Queue
void read_irq2in_to_queue(FILE* irq2in, Queue* queue);

copy diskin to diskout
void hardcopy_diskin_to_diskout(FILE* file1, FILE* file2);

implementation of DMA
void read_diskoutSector_to_mem(FILE* diskout, int* memory, int disk_sector, int mem_adress);
implementation of DMA

void write_mem_to_diskoutSector(int* memory, FILE* diskout, int disk_sector, int mem_adress);

input: a string
output: function changes the string characters (digits ,special characters or letters) to uppercase.
digits and special characters are unchanged.
void str_toupper(char* str);

<u>simulator.c</u>

Below is the code structure for the main function in simulator.c .
The main function is divided to code sectors, Local variable definition, initialization of data structures, opening files, fetch-decode-execute loop and House keeping.

```c
int main(int argc, char* argv[]) {
    // ----------- local variables ----------->
    int PC = 0, PC_updated = 0, hw_updated = 0, leds_changed = 0, disp_7seg_changed = 0,
        disk_action = 0, disk_timer=0, interrupt = 0;

    // ----------- initialization ----------->
    int* memory = (int*)malloc(sizeof(int) * MEM_SIZE);
    if (memory == NULL) return 1;

    int* registers = (int*)malloc(sizeof(int) * REG_NUM);
    if (registers == NULL) { ... }
    init_registers(registers);

    PL* instruction = (PL*)malloc(sizeof(PL));
    if (instruction == NULL) { ... }

    int* hw_registers = (int*)malloc(sizeof(int) * HW_REG_NUM);
    if (hw_registers == NULL) { ... }
    init_hw_reg(hw_registers);

    unsigned char framebuffer[Y_PIXELS][X_PIXELS];
    init_framebuffer(framebuffer);

    Queue* queue = createQueue();


    // ----------- opening files ----------->
    FILE *memin = NULL, *irq2in = NULL, *memout = NULL,
        *regout = NULL, *trace = NULL, *diskin = NULL, *hwregtrace = NULL, *cycles = NULL,
        *leds = NULL, *display7seg = NULL, *diskout = NULL, *monitor = NULL,
        *monitor_yuv = NULL;
```

```c
    // ----------- opening files ----------->
    FILE *memin = NULL, *irq2in = NULL, *memout = NULL,
        *regout = NULL, *trace = NULL, *diskin = NULL, *hwregtrace = NULL, *cycles = NULL,
        *leds = NULL, *display7seg = NULL, *diskout = NULL, *monitor = NULL,
        *monitor_yuv = NULL;

    open_file_check(&memin, argv[MEMIN], "r");
    open_file_check(&trace, argv[TRACE], "w+");
    open_file_check(&hwregtrace, argv[HWREGTRACE], "w+");
    open_file_check(&irq2in, argv[IRQ2IN], "r");
    open_file_check(&diskin, argv[DISKIN], "r");
    open_file_check(&diskout, argv[DISKOUT], "w+");
    open_file_check(&leds, argv[LEDS], "w+");
    open_file_check(&display7seg, argv[DISPLAY7SEG], "w+");

    read_irq2in_to_queue(irq2in, queue);
    fclose(irq2in);
    read_memin_to_memory(memin, memory);
    fclose(memin);

    hardcopy_diskin_to_diskout(diskin, diskout);


    // ----------- fetch-decode-execute ----------->
    while (memory[PC] != HALT_DEC && PC < MEM_SIZE){ ... }
    instruction->i_format = 0;

    if (PC == MEM_SIZE){ ... }

    //----------- House Keeping ----------->
    fclose(diskout);
    fclose(leds);
    fclose(display7seg);
```


```c
    //----------- House Keeping ----------->
    fclose(diskout);
    fclose(leds);
    fclose(display7seg);
```

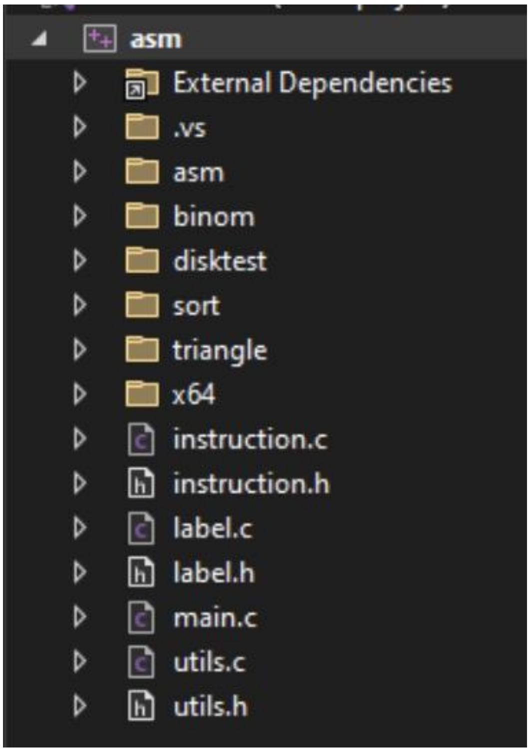

```c
    update_trace(trace, PC, memory[PC], registers, instruction); // TRACE
    fclose(trace);
    free(instruction);

    open_file_check(&monitor, argv[MONITOR], "w+");
    open_file_check(&monitor_yuv, argv[MONITOR_YUV], "wb+");
    write_framebuffer_to_monitor(framebuffer, monitor, monitor_yuv); // FINAL MONITOR AND MONITOR.YUV UPDATE
    fclose(monitor);
    fclose(monitor_yuv);


    open_file_check(&memout, argv[MEMOUT], "w+");
    write_memory_to_memout(memout, memory); // MEMOUT
    fclose(memout);
    free(memory);

    open_file_check(&regout, argv[REGOUT], "w+");
    write_registers_to_regout(regout, registers); // REGOUT
    fclose(regout);
    free(registers);

    open_file_check(&cycles, argv[CYCLES], "w+");
    fprintf(cycles, "%d", hw_registers[CLKS]); // CYCLES
    fclose(cycles);
    free(hw_registers);

    return 0;
}
```

Assembler:

We developed a C-based assembler that converts assembly code into machine code, the key steps included:

1. **Instruction Parsing**: Implemented functions to parse assembly instructions and map opcodes/registers to their numeric values.
2. **Label Handling**: Created mechanisms to collect and manage labels during the first pass.
3. **File I/O**: Integrated file operations to handle input and output, ensuring files are created and written correctly.
4. **Machine Code Generation**: Developed functions to translate parsed instructions into machine code, supporting R-type, I-type, and `.word` directives.
5. **Debugging and Refinement**: Fixed bugs, improved code clarity, and ensured correct handling of all instruction formats.

Directory Tree:



our main file was main.c which contains:

Function Name	Arguments	Purpose
main	int argc, char* argv[]	Lead the assembly process: reads input file, collects labels, translates instructions, writes output.

for instruction.c:

Function Name	Arguments	Purpose
parseInstruction	char* line, Instruction* inst	Parses a line of assembly code, storing the parsed components in the Instruction structure.
writeMachineCode	const char* outfileName, Instruction* inst, Label* labels, int labelCount, int isIFormat	Converts parsed instruction into machine code and writes it to the output array.
handleWordDirective	const char* outfileName, Instruction* inst	Handles the .word directive, writing a specific value to a memory address.
printInstruction	Instruction* inst	Prints the contents of an Instruction structure for debugging purposes.

for labels.c:

Function Name	Arguments	Purpose
collectLabels	char* line, Label* labels, int* labelCount, int* address	Collects labels from assembly code, calculates their addresses, and updates the program counter.

for utils.c:

Function Name	Arguments	Purpose
`initializeOutputFile`	`None`	Initializes the output file array with default values ("00000\n").
`writeInstructionToOutput`	`int address,` `const char* machineCode`	Writes an instruction's machine code to a specific address in the output array.
`writeWordDirectiveToOutput`	`int address,` `int data`	Writes the result of a .word directive to a specific address in the output array.
`flushOutputToFile`	`const char* outfileName`	Flushes the contents of the output array to the specified output file.
`registerToNumber`	`char* reg`	Maps register names to their corresponding register numbers.
`getOpcode`	`char* opcode`	Maps instruction opcodes to their corresponding opcode values.

In our assembler, we manage file output by first storing all the machine code in an outputFile array. We initialize this array with default values, ensuring every memory address has a placeholder. As we process each assembly instruction, we convert it to machine code and place it directly into the array at the correct memory address.

For special directives like .word, we update specific addresses in the array with the provided data. Once all the instructions and directives are processed, we write the entire array to the output file in one go. This approach gives us complete control over the memory layout, prevents partial writes, and ensures the final output file is accurate and well-structured.

Assembly Test Files as required in ISA

Triangle Drawing (`triangle.asm`):

Approach: This program uses Bresenham's line algorithm to draw a triangle on a frame buffer. The algorithm determines which points to plot for lines between two coordinates, adjusting the slope to handle both steep and shallow angles. here is the python analogy:

```
def draw_line(x1, y1, x2, y2):
    dx = abs(x2 - x1)
    dy = abs(y2 - y1)
    d = 2*dy - dx
    while x1 <= x2:
        plot(x1, y1)
        x1 += 1
        if d > 0:
            y1 += 1
            d += 2*(dy - dx)
        else:
            d += 2*dy
```

Binomial Coefficient Calculation (`binom.asm`):

Approach: This program computes the binomial coefficient using a recursive function, with base cases for `k == 0` and `k == n`. It makes use of the stack to manage recursive calls and intermediate results. python analogy:

```
def binomial_coefficient(n, k):
    if k == 0 or k == n:
        return 1
    else:
        return binomial_coefficient(n-1, k-1) + binomial_coefficient(n-1, k)
```

Sorting (`sort.asm`):

Approach: This program implements a bubble sort algorithm. It iteratively compares adjacent elements and swaps them if they are in the wrong order, repeatedly passing through the list until it is sorted. python analogy:

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
```

Testing the Disk Implementation (`disktest.asm`):

Approach: This program uses the implementation of the disk specified in the ISA and copies sectors 0-7 of the disk to the memory iteratively via DMA, then sums the elements in the matching indexes from sectors 0-7 according to instruction in the following memory, and then writes the calculated elements from the memory to sector 8 of the disk via DMA.

The reading of the files was done using an implementation of a For loop in assembly which outputs the correct data to the relevant hardware registers and than a waiting loop that waits for the disk to complete the task (diskstatus = 0).

(for every time the disk was ready we implemented an irq1 handler that turns irq1status off simulating the hardware performing the DMA).

Then we implemented 2 nested for loops to iterate over sectors 0-7 and sum them by matching indexes, and saved the results to memory cell 3072 with the offset of the same index.

Finally we output the correct data to the hardware registers to read from the memory we stored to sector 8 of the disk.