Computer Structure - ISA project, documentation

Amit Damari - 326631249
Ido Zylberman - 326634730
Shachar Or - 214135659

Background

The SIMP processor is a MIPS-like architecture that executes assembly language programs. The processor features 16 registers and supports various I/O operations including LED control, monitor display, disk operations, and interrupt handling.

Instruction Format

Each instruction is 32 bits (8 hexadecimal characters) with the following format:

- Bits 47-40: Opcode
- Bits 39-36: rd (destination register)
- Bits 35-32: rs (source register 1)
- Bits 31-28: rt (source register 2)
- Bits 23-12: Immediate1
- Bits 11-0: Immediate2

Key Features

1. Register Set
    - 16 general-purpose registers
    - Special registers include zero constant, immediate values, and stack pointer
2. I/O Capabilities
    - 32 LED outputs
    - Grayscale monitor (256x256 pixels)
    - Disk drive (64KB, 128 sectors)
    - Timer with interrupt support
3. Interrupt System
    - Supports 3 interrupts (irq0-timer, irq1-disk, irq2-external)
    - Interrupt handling with status registers
    - No nested interrupt support
4. Memory System
    - Instruction memory
    - Data memory
    - DMA support for disk operations

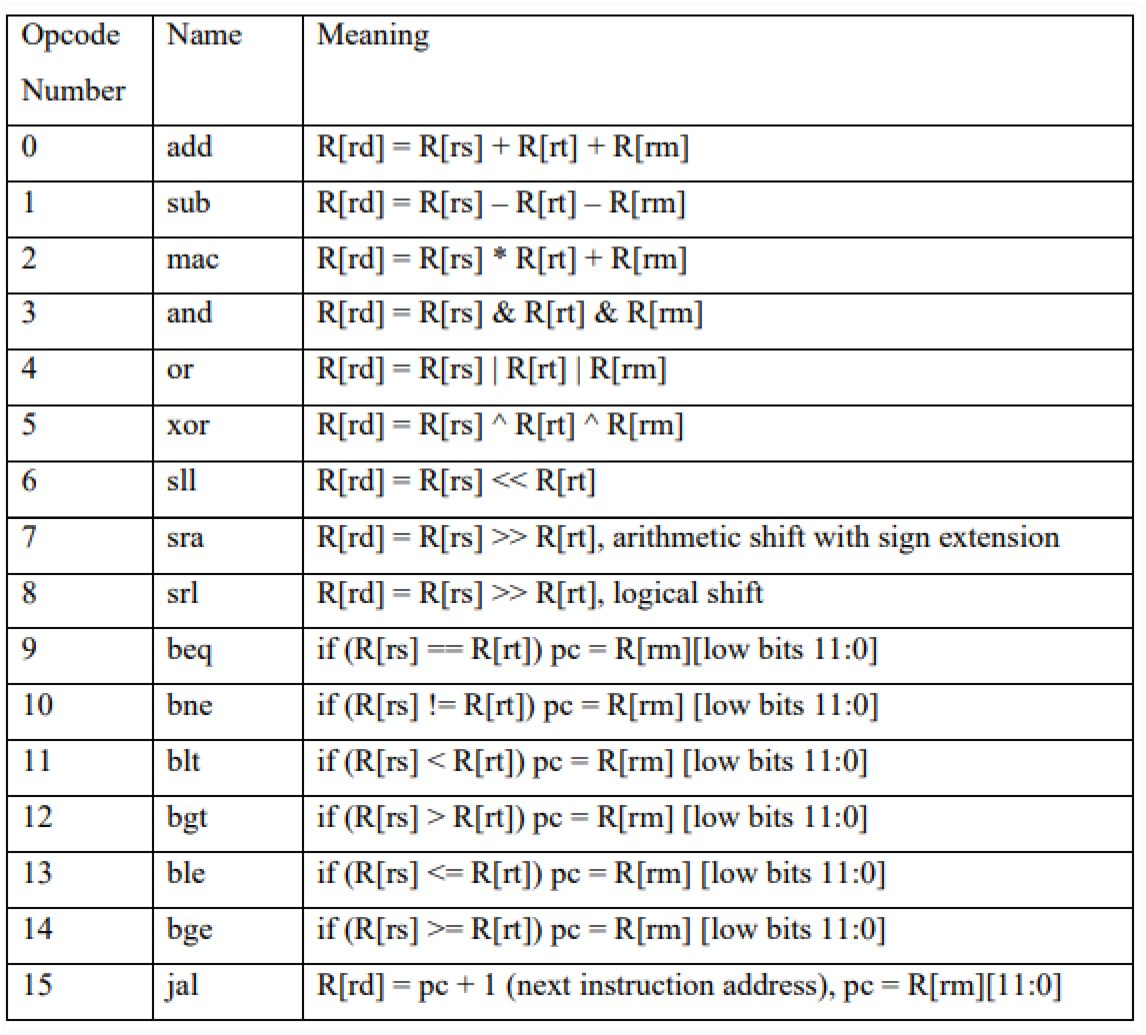
Clock and Timing

- Each instruction executes in one clock cycle
- Disk operations take 1024 clock cycles
- Timer operates on system clock

Assembly Language Features

- Comments start with #
- Labels supported for program flow control
- .word directive for direct memory initialization
- 22 instruction set including arithmetic, logical, and control operations

The opcodes supported by the processor and the meaning of each instruction are given in the following table:



Opcode Number	Name	Meaning
0	add	R[rd] = R[rs] + R[rt] + R[rm]
1	sub	R[rd] = R[rs] – R[rt] – R[rm]
2	mac	R[rd] = R[rs] * R[rt] + R[rm]
3	and	R[rd] = R[rs] & R[rt] & R[rm]
4	or	R[rd] = R[rs] | R[rt] | R[rm]
5	xor	R[rd] = R[rs] ^ R[rt] ^ R[rm]
6	sll	R[rd] = R[rs] << R[rt]
7	sra	R[rd] = R[rs] >> R[rt], arithmetic shift with sign extension
8	srl	R[rd] = R[rs] >> R[rt], logical shift
9	beq	if (R[rs] == R[rt]) pc = R[rm][low bits 11:0]
10	bne	if (R[rs] != R[rt]) pc = R[rm] [low bits 11:0]
11	blt	if (R[rs] < R[rt]) pc = R[rm] [low bits 11:0]
12	bgt	if (R[rs] > R[rt]) pc = R[rm] [low bits 11:0]
13	ble	if (R[rs] <= R[rt]) pc = R[rm] [low bits 11:0]
14	bge	if (R[rs] >= R[rt]) pc = R[rm] [low bits 11:0]
15	jal	R[rd] = pc + 1 (next instruction address), pc = R[rm][11:0]

16	lw	R[rd] = MEM[R[rs]+R[rt]] + R[rm]
17	sw	MEM[R[rs]+R[rt]] = R[rm] + R[rd]
18	reti	PC = IORegister[7]
19	in	R[rd] = IORegister[R[rs] + R[rt]]
20	out	IORegister [R[rs]+R[rt]] = R[rm]
21	halt	Halt execution, exit simulator

The processor executes instructions in a fetch-decode-execute cycle and terminates upon encountering the HALT command. All memory and register values are represented in hexadecimal format.

The SIMP processor includes 16 registers, each 32 bits wide. The name, number and role of each register according to the calling conventions is given in the following table:

Register Number	Register Name	Purpose
0	$zero	Constant zero
1	$imm1	Sign extended immediate 1
2	$imm2	Sign extended immediate 2
3	$v0	Result value
4	$a0	Argument register
5	$a1	Argument register
6	$a2	Argument register
7	$t0	Temporary register
8	$t1	Temporary register
9	$t2	Temporary register
10	$s0	Saved register
11	$s1	Saved register
12	$s2	Saved register
13	$gp	Global pointer (static data)
14	$sp	Stack pointer
15	$ra	Return address

Assembler - Assembler.c

The assembler is a two-pass assembler that converts SIMP assembly language into machine code. It processes assembly instructions and generates two output files: one for instruction memory (imemin.txt) and one for data memory (dmemin.txt).

Design Constraints

The assembler is built to support programs of theoretically infinite size, limited only by available memory. To achieve this, the following constraints are imposed:

- Label Length: Up to 50 characters.
- Code Line Length: Up to 500 characters.
- Immediate Value Length: Up to 50 characters.
- Register/Opcode Name Length: Up to 6 characters.

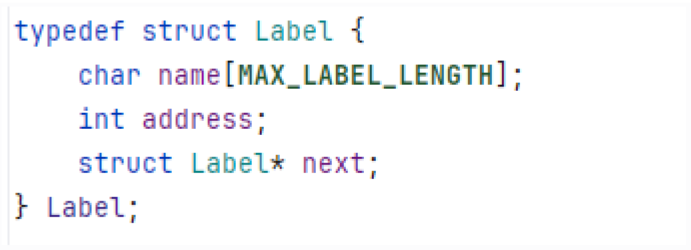
A global macro MAX_LINE_LENGTH is defined with a value of 500 to enforce the maximum line length.

Note: The assembler relies on standard C libraries (string.h, stdlib.h, stdio.h, and ctype.h). These libraries are installed and accessible during compilation and execution as per project guidelines.

Data Structures

Label Structure

To manage labels efficiently, especially in large programs, the assembler employs a linked list structure. Each node in the linked list represents a label, storing its name and corresponding memory address. This approach mimics object-oriented programming concepts within the constraints of the C language.



```c
typedef struct Label {
    char name[MAX_LABEL_LENGTH];
    int address;
    struct Label* next;
} Label;
```

The Label structure contains:

- name: A string up to 50 characters representing the label's identifier.
- address: An integer indicating the memory location associated with the label, used for jump and branch instructions.
- next: A pointer to the next Label in the linked list, facilitating traversal and management.

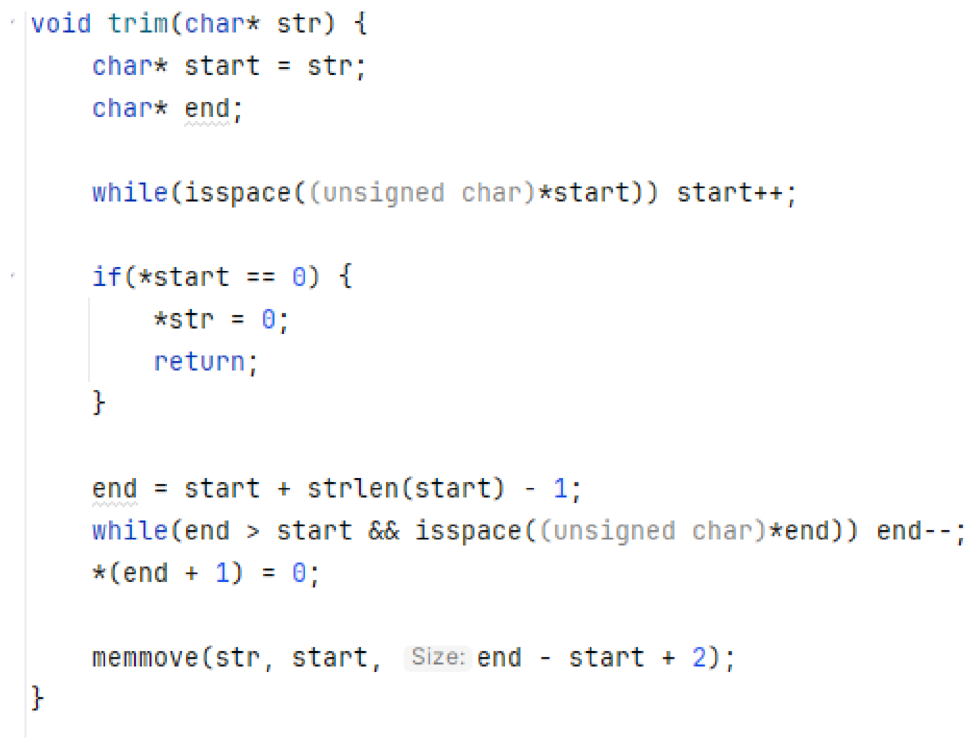
Function Descriptions

Utility Functions

trim(char* str)

Purpose:
Removes leading and trailing whitespace from a string. This is essential for accurately parsing assembly instructions by ensuring that extraneous spaces do not interfere with instruction recognition and processing.

Effect on Assembler:
Facilitates clean and precise parsing of each line in the assembly code, preventing errors related to unintended whitespace.

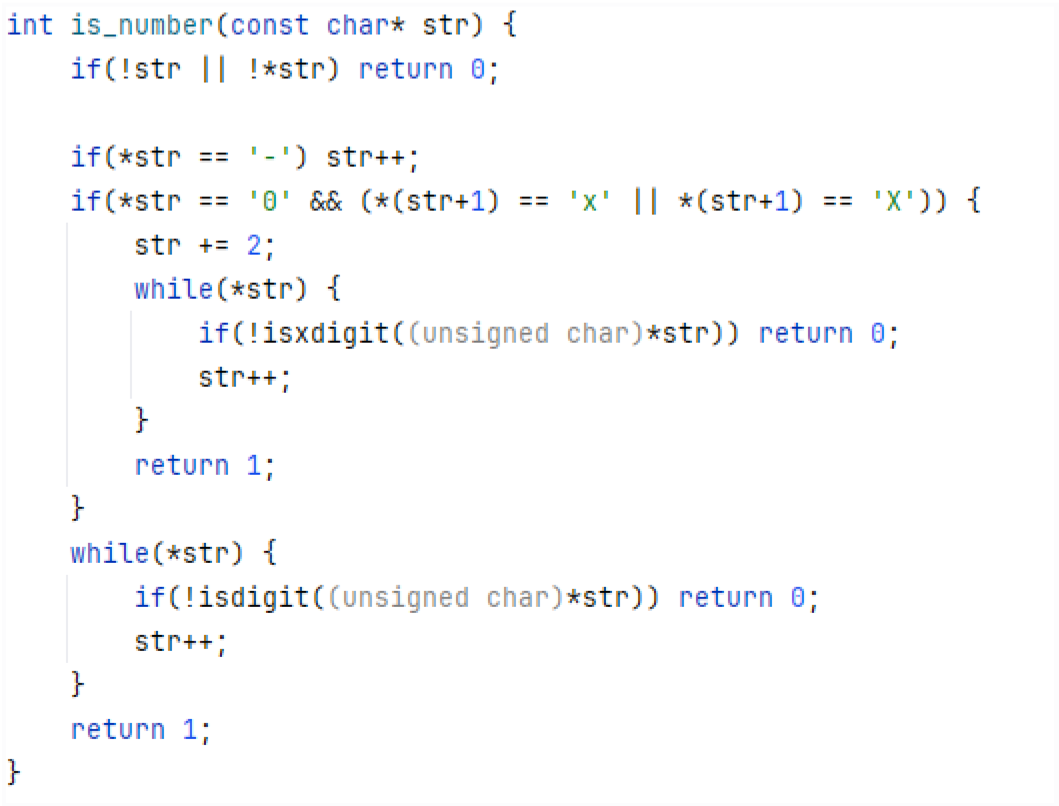


```
void trim(char* str) {
    char* start = str;
    char* end;

    while(isspace((unsigned char)*start)) start++;

    if(*start == 0) {
        *str = 0;
        return;
    }

    end = start + strlen(start) - 1;
    while(end > start && isspace((unsigned char)*end)) end--;
    *(end + 1) = 0;

    memmove(str, start,  Size: end - start + 2);
}
```

is_number(const char* str)

Purpose:
Determines whether a given string represents a valid number, supporting both decimal and hexadecimal formats.

Effect on Assembler:
Ensures accurate interpretation of immediate values and numerical operands within instructions, preventing invalid number formats from causing assembly errors.

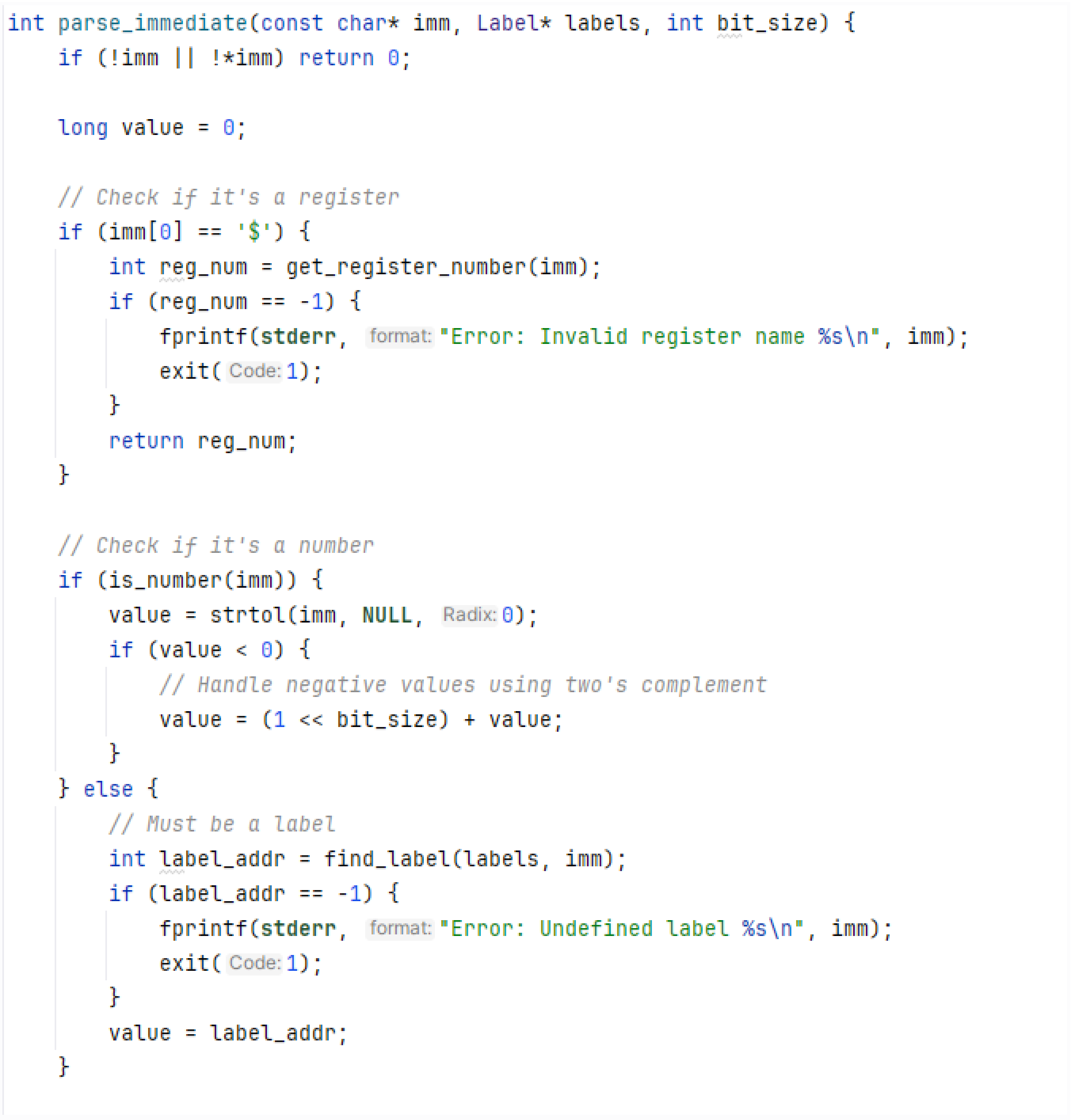


```c
int is_number(const char* str) {
    if(!str || !*str) return 0;

    if(*str == '-') str++;
    if(*str == '0' && (*(str+1) == 'x' || *(str+1) == 'X')) {
        str += 2;
        while(*str) {
            if(!isxdigit((unsigned char)*str)) return 0;
            str++;
        }
        return 1;
    }
    while(*str) {
        if(!isdigit((unsigned char)*str)) return 0;
        str++;
    }
    return 1;
}
```

parse_immediate(const char* imm, Label* labels, int bit_size)

Purpose:
Converts immediate values from assembly format to their corresponding machine code representations. It handles registers, numerical literals, and label references, ensuring they fit within the specified bit size.

Effect on Assembler:
Allows for flexible and accurate encoding of immediate operands in instructions, supporting both signed and unsigned values, as well as forward and backward label references.

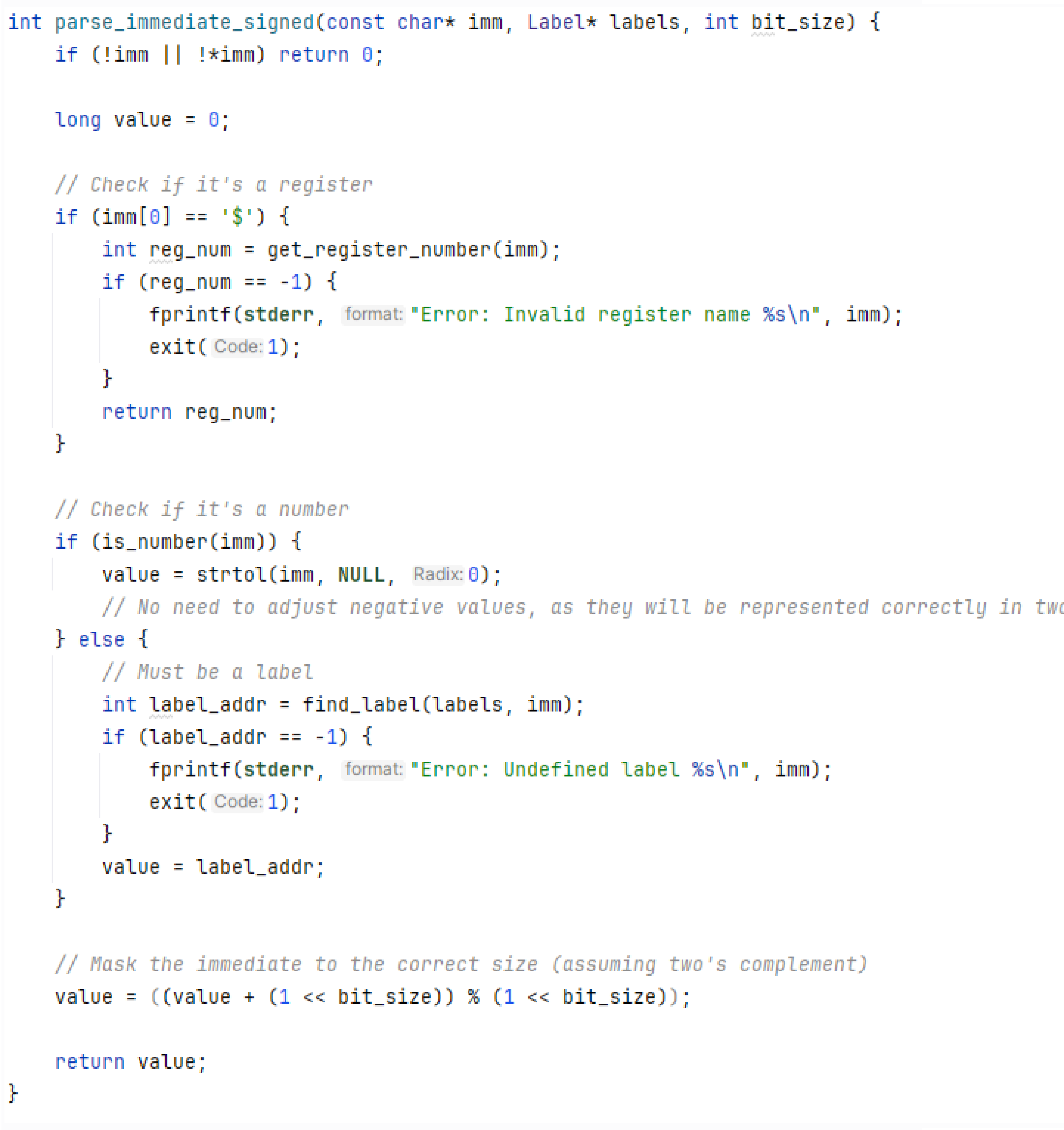


```
int parse_immediate(const char* imm, Label* labels, int bit_size) {
    if (!imm || !*imm) return 0;

    long value = 0;

    // Check if it's a register
    if (imm[0] == '$') {
        int reg_num = get_register_number(imm);
        if (reg_num == -1) {
            fprintf(stderr, format: "Error: Invalid register name %s\n", imm);
            exit( Code: 1);
        }
        return reg_num;
    }

    // Check if it's a number
    if (is_number(imm)) {
        value = strtol(imm, NULL, Radix: 0);
        if (value < 0) {
            // Handle negative values using two's complement
            value = (1 << bit_size) + value;
        }
    } else {
        // Must be a label
        int label_addr = find_label(labels, imm);
        if (label_addr == -1) {
            fprintf(stderr, format: "Error: Undefined label %s\n", imm);
            exit( Code: 1);
        }
        value = label_addr;
    }
```

parse_immediate_signed(const char* imm, Label* labels, int bit_size)

Purpose:
Processes signed immediate values, ensuring that negative numbers are correctly represented using two's complement arithmetic. Similar to parse_immediate, it handles registers, numerical literals, and labels.

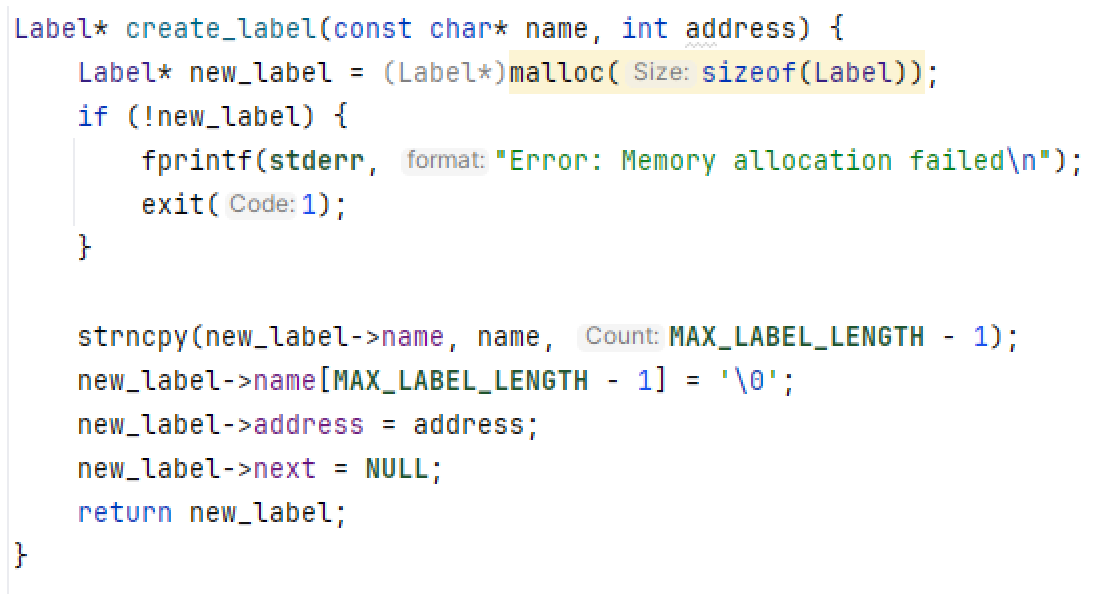
Effect on Assembler:
Enables the assembler to accurately encode signed immediate operands, essential for instructions involving arithmetic operations, conditional branches, and other control flow mechanisms.



```
int parse_immediate_signed(const char* imm, Label* labels, int bit_size) {
    if (!imm || !*imm) return 0;

    long value = 0;

    // Check if it's a register
    if (imm[0] == '$') {
        int reg_num = get_register_number(imm);
        if (reg_num == -1) {
            fprintf(stderr,  format: "Error: Invalid register name %s\n", imm);
            exit( Code: 1);
        }
        return reg_num;
    }

    // Check if it's a number
    if (is_number(imm)) {
        value = strtol(imm, NULL,  Radix: 0);
        // No need to adjust negative values, as they will be represented correctly in tw
    } else {
        // Must be a label
        int label_addr = find_label(labels, imm);
        if (label_addr == -1) {
            fprintf(stderr,  format: "Error: Undefined label %s\n", imm);
            exit( Code: 1);
        }
        value = label_addr;
    }

    // Mask the immediate to the correct size (assuming two's complement)
    value = ((value + (1 << bit_size)) % (1 << bit_size));

    return value;
}
```

Label Management Functions

create_label(const char* name, int address)

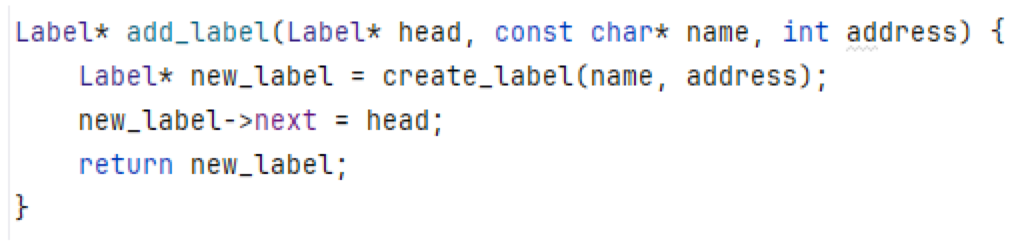


```
Label* create_label(const char* name, int address) {
    Label* new_label = (Label*)malloc( Size: sizeof(Label));
    if (!new_label) {
        fprintf(stderr,  format: "Error: Memory allocation failed\n");
        exit( Code: 1);
    }

    strncpy(new_label->name, name,  Count: MAX_LABEL_LENGTH - 1);
    new_label->name[MAX_LABEL_LENGTH - 1] = '\0';
    new_label->address = address;
    new_label->next = NULL;
    return new_label;
}
```

Purpose:
Allocates memory for a new label and initializes it with the provided name and address. Returns a pointer to the newly created label.

Effect on Assembler:
Facilitates the creation of label entries during the first pass, allowing the assembler to keep track of label locations for later reference during instruction encoding.

add_label(Label* head, const char* name, int address)

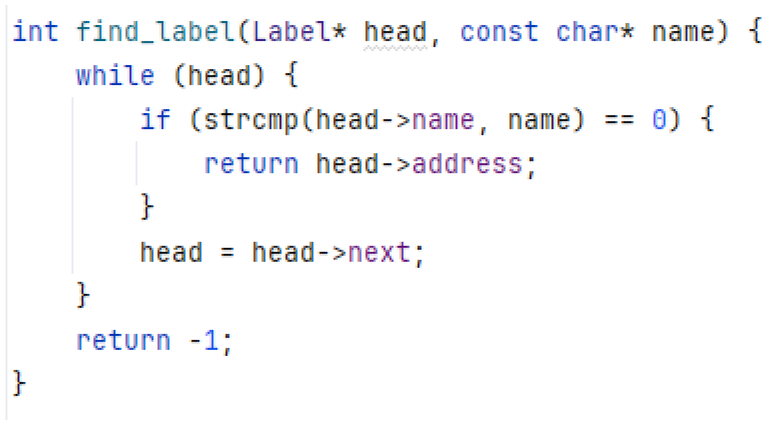


```
Label* add_label(Label* head, const char* name, int address) {
    Label* new_label = create_label(name, address);
    new_label->next = head;
    return new_label;
}
```

Purpose:
Adds a new label to the front of the existing linked list of labels. It utilizes the create_label function to instantiate the label and then links it to the current head of the list.

Effect on Assembler:
Maintains an efficient and dynamically expandable list of labels, enabling the assembler to handle labels regardless of their position in the code or the size of the program.

find_label(Label* head, const char* name)



```c
int find_label(Label* head, const char* name) {
    while (head) {
        if (strcmp(head->name, name) == 0) {
            return head->address;
        }
        head = head->next;
    }
    return -1;
}
```

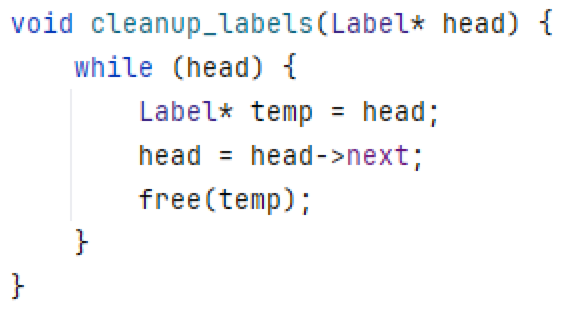
Purpose:
Searches the linked list of labels for a label with the specified name. If found, it returns the label's address; otherwise, it returns -1 to indicate that the label is undefined.

Effect on Assembler:
Enables the assembler to resolve label references during the second pass, ensuring that jump and branch instructions can correctly reference the intended memory locations.

cleanup_labels(Label* head)

Purpose:
Deallocates all memory allocated for the linked list of labels, preventing memory leaks by freeing each node in the list.

Effect on Assembler:
Ensures that all dynamically allocated memory for labels is properly released after the assembly process is complete, maintaining optimal memory usage and preventing resource exhaustion.



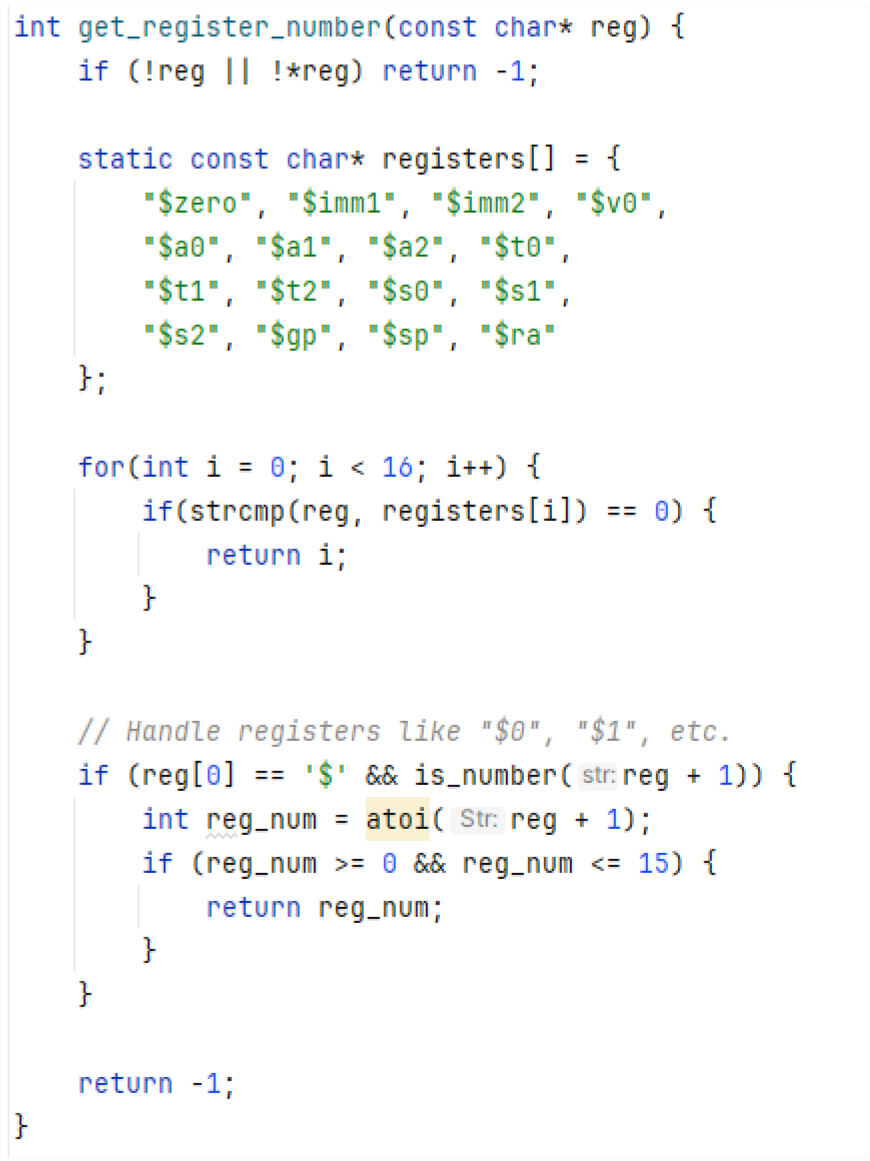
```
void cleanup_labels(Label* head) {
    while (head) {
        Label* temp = head;
        head = head->next;
        free(temp);
    }
}
```

Instruction Processing Functions

get_register_number(const char* reg)

Purpose:
Converts register names (e.g., $zero, $t0) or numbered registers (e.g., $0, $1) to their corresponding numerical identifiers used in machine code encoding.

Effect on Assembler:
Ensures that registers specified in assembly instructions are accurately mapped to their numerical representations, which is crucial for correct instruction encoding and execution.

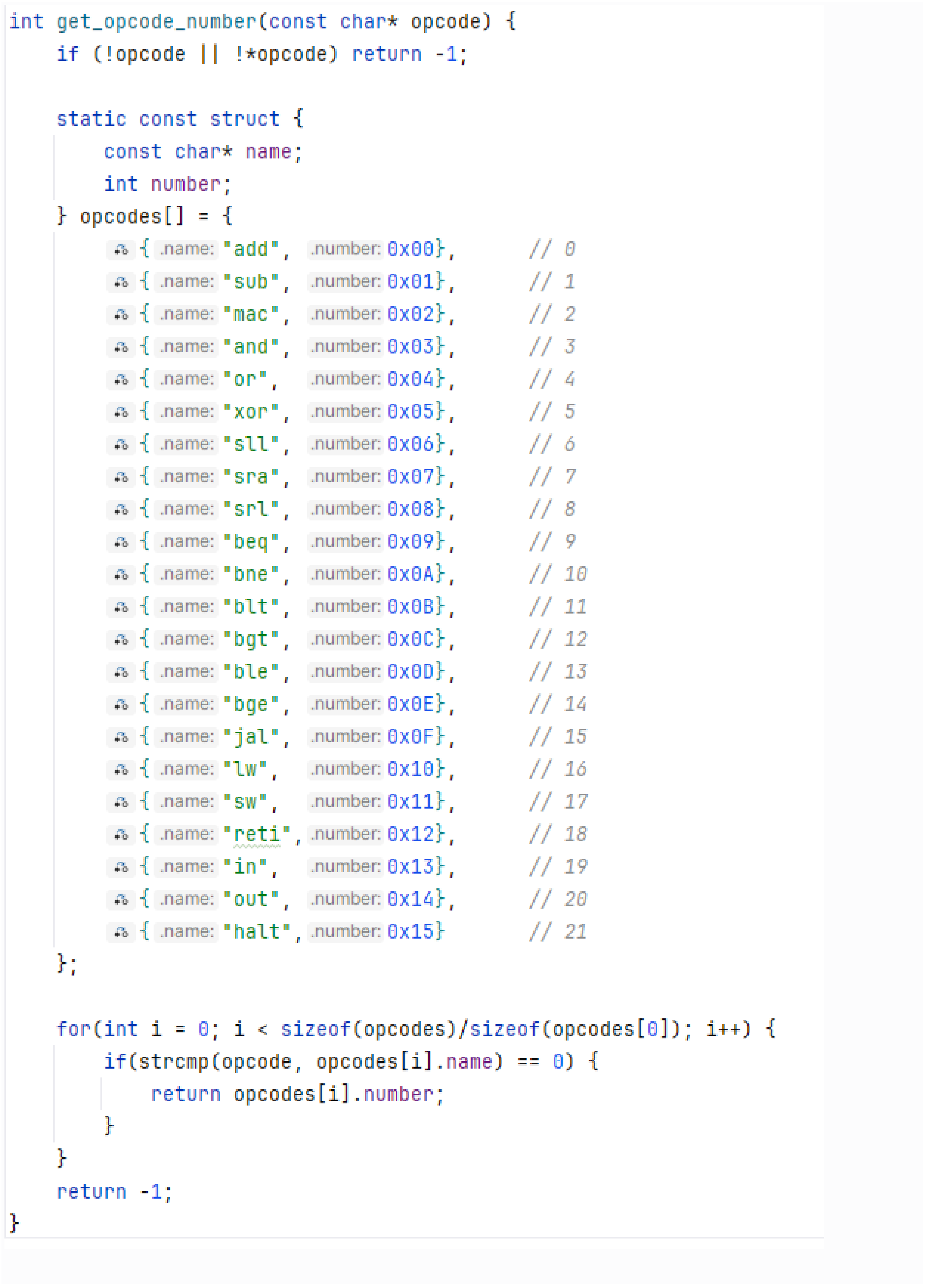


```c
int get_register_number(const char* reg) {
    if (!reg || !*reg) return -1;

    static const char* registers[] = {
        "$zero", "$imm1", "$imm2", "$v0",
        "$a0", "$a1", "$a2", "$t0",
        "$t1", "$t2", "$s0", "$s1",
        "$s2", "$gp", "$sp", "$ra"
    };

    for(int i = 0; i < 16; i++) {
        if(strcmp(reg, registers[i]) == 0) {
            return i;
        }
    }

    // Handle registers like "$0", "$1", etc.
    if (reg[0] == '$' && is_number( str: reg + 1)) {
        int reg_num = atoi( Str: reg + 1);
        if (reg_num >= 0 && reg_num <= 15) {
            return reg_num;
        }
    }

    return -1;
}
```

get_opcode_number(const char* opcode)

Purpose:
Maps assembly instruction mnemonics (e.g., add, sub, lw) to their corresponding opcode values defined in the SIMP processor specification.

Effect on Assembler:
Enables the assembler to encode instructions accurately by associating each mnemonic with its correct opcode, ensuring that the resulting machine code corresponds to the intended operations.



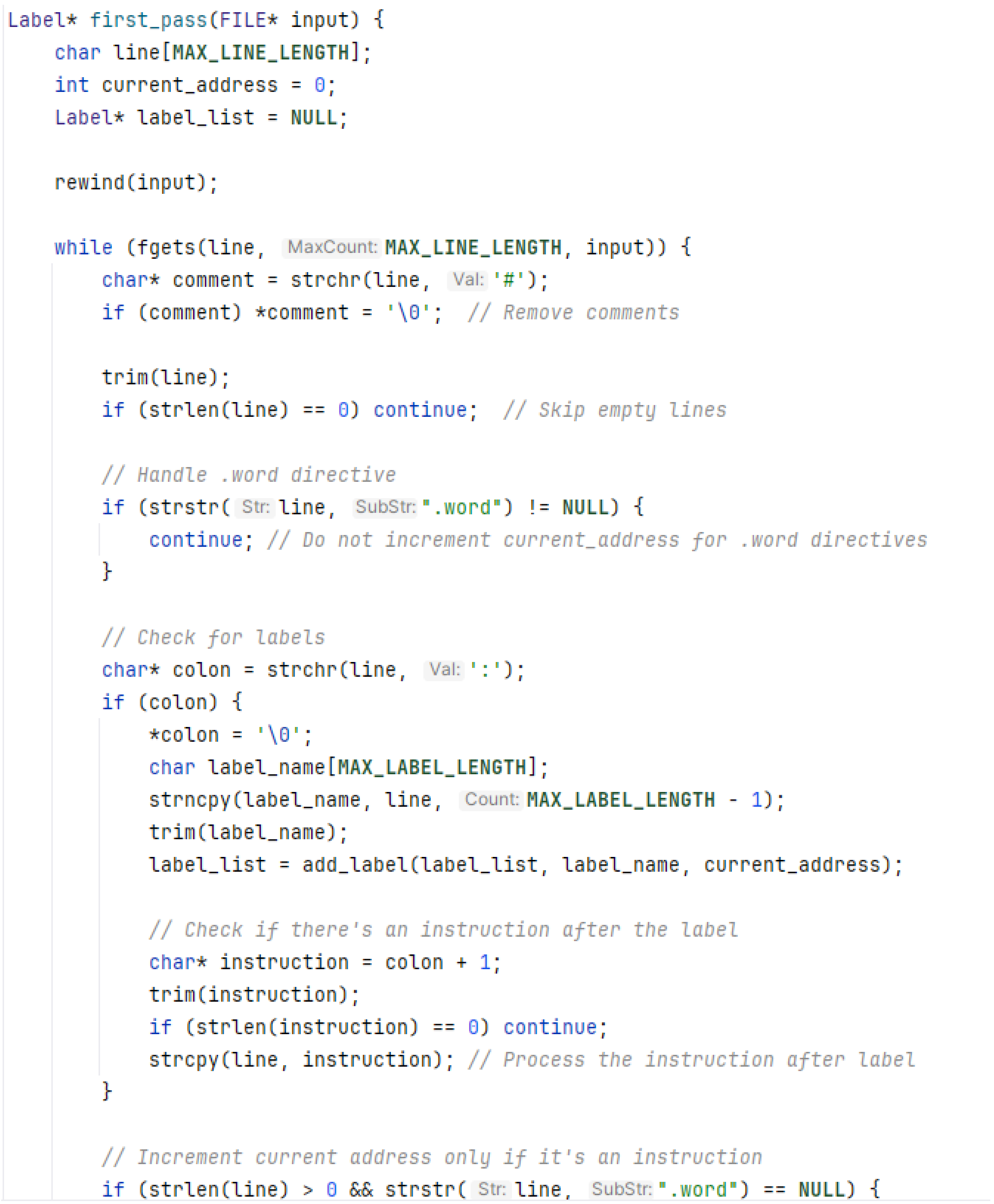
```c
int get_opcode_number(const char* opcode) {
    if (!opcode || !*opcode) return -1;

    static const struct {
        const char* name;
        int number;
    } opcodes[] = {
        { .name: "add",  .number: 0x00},      // 0
        { .name: "sub",  .number: 0x01},      // 1
        { .name: "mac",  .number: 0x02},      // 2
        { .name: "and",  .number: 0x03},      // 3
        { .name: "or",   .number: 0x04},      // 4
        { .name: "xor",  .number: 0x05},      // 5
        { .name: "sll",  .number: 0x06},      // 6
        { .name: "sra",  .number: 0x07},      // 7
        { .name: "srl",  .number: 0x08},      // 8
        { .name: "beq",  .number: 0x09},      // 9
        { .name: "bne",  .number: 0x0A},      // 10
        { .name: "blt",  .number: 0x0B},      // 11
        { .name: "bgt",  .number: 0x0C},      // 12
        { .name: "ble",  .number: 0x0D},      // 13
        { .name: "bge",  .number: 0x0E},      // 14
        { .name: "jal",  .number: 0x0F},      // 15
        { .name: "lw",   .number: 0x10},      // 16
        { .name: "sw",   .number: 0x11},      // 17
        { .name: "reti", .number: 0x12},      // 18
        { .name: "in",   .number: 0x13},      // 19
        { .name: "out",  .number: 0x14},      // 20
        { .name: "halt", .number: 0x15}       // 21
    };

    for(int i = 0; i < sizeof(opcodes)/sizeof(opcodes[0]); i++) {
        if(strcmp(opcode, opcodes[i].name) == 0) {
            return opcodes[i].number;
        }
    }
    return -1;
}
```

Assembly Passes

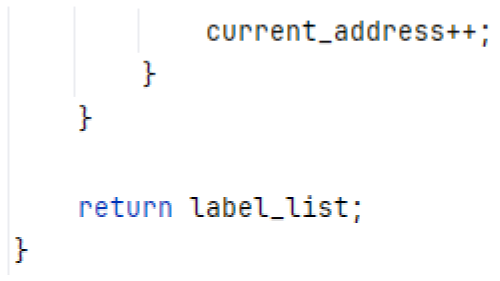
first_pass(FILE* input)

Purpose:
Performs the first pass over the assembly input file to collect all label definitions and record their corresponding memory addresses. This pass solely identifies labels without generating machine code, preparing for accurate instruction encoding in the second pass.

Effect on Assembler:
Establishes a complete mapping of label names to their memory locations, enabling the assembler to resolve label references during the second pass. This separation ensures that forward references to labels defined later in the code are handled correctly.



```c
Label* first_pass(FILE* input) {
    char line[MAX_LINE_LENGTH];
    int current_address = 0;
    Label* label_list = NULL;

    rewind(input);

    while (fgets(line, MAX_LINE_LENGTH, input)) {
        char* comment = strchr(line, '#');
        if (comment) *comment = '\0';  // Remove comments

        trim(line);
        if (strlen(line) == 0) continue;  // Skip empty lines

        // Handle .word directive
        if (strstr(line, ".word") != NULL) {
            continue; // Do not increment current_address for .word directives
        }

        // Check for labels
        char* colon = strchr(line, ':');
        if (colon) {
            *colon = '\0';
            char label_name[MAX_LABEL_LENGTH];
            strncpy(label_name, line, MAX_LABEL_LENGTH - 1);
            trim(label_name);
            label_list = add_label(label_list, label_name, current_address);

            // Check if there's an instruction after the label
            char* instruction = colon + 1;
            trim(instruction);
            if (strlen(instruction) == 0) continue;
            strcpy(line, instruction); // Process the instruction after label
        }

        // Increment current address only if it's an instruction
        if (strlen(line) > 0 && strstr(line, ".word") == NULL) {
```

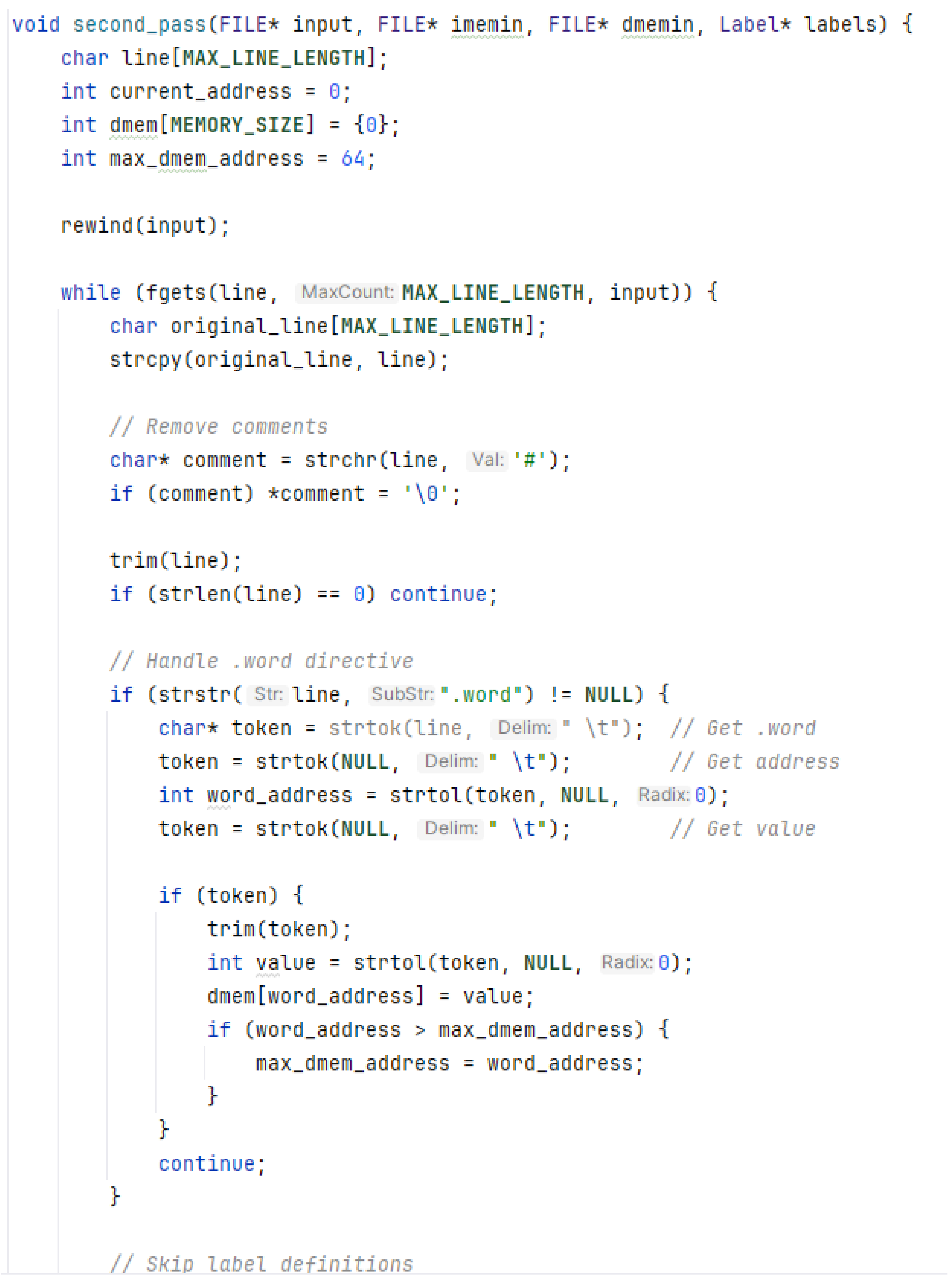


```
            current_address++;
        }
    }

    return label_list;
}
```

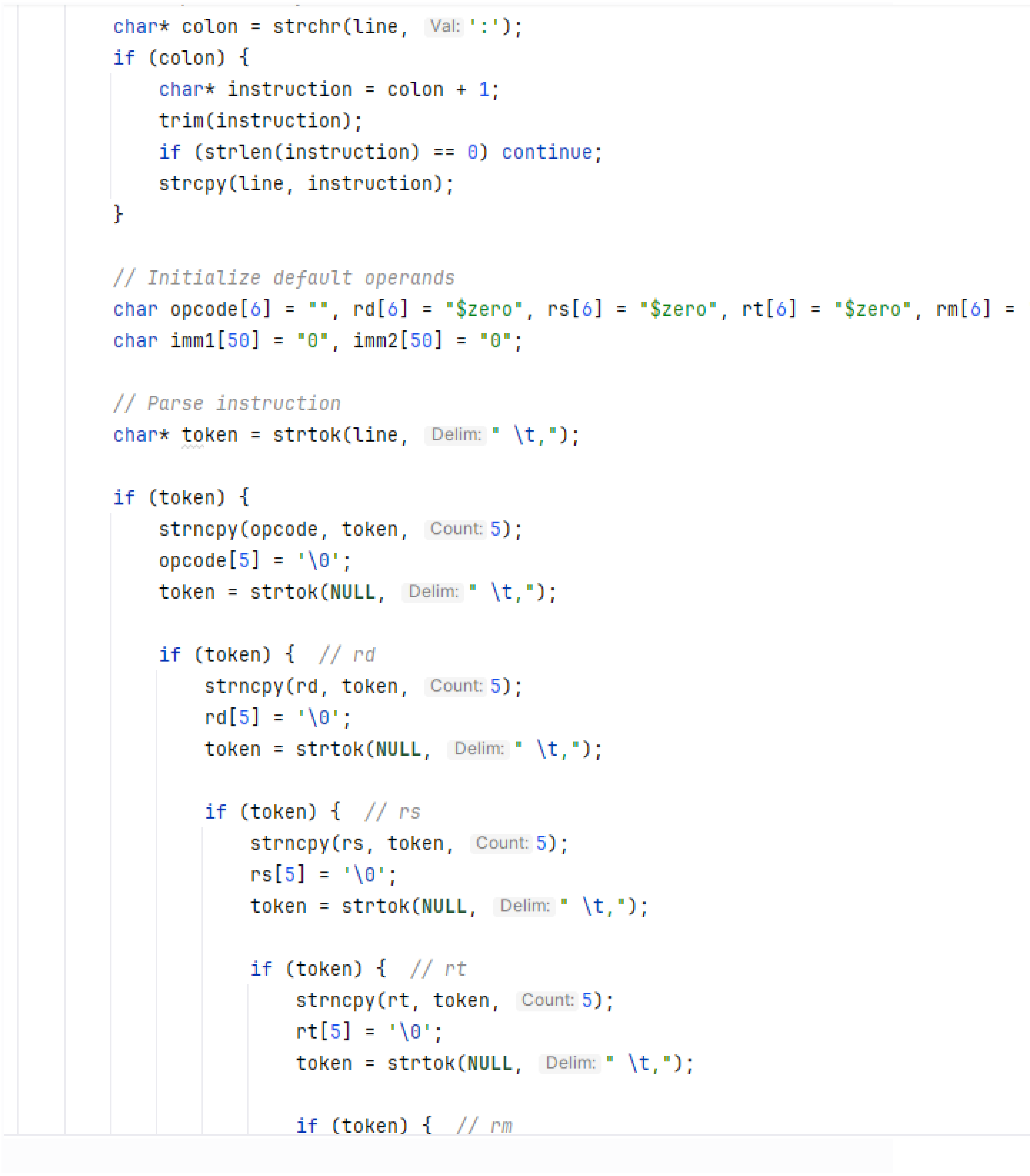
second_pass(FILE* input, FILE* imemin, FILE* dmemin, Label* labels)

Purpose:
Performs the second pass over the assembly input file to generate the final machine code and data memory initialization. This pass encodes each instruction into its hexadecimal representation, resolves label references using the label list generated during the first pass, and handles .word directives for data initialization.

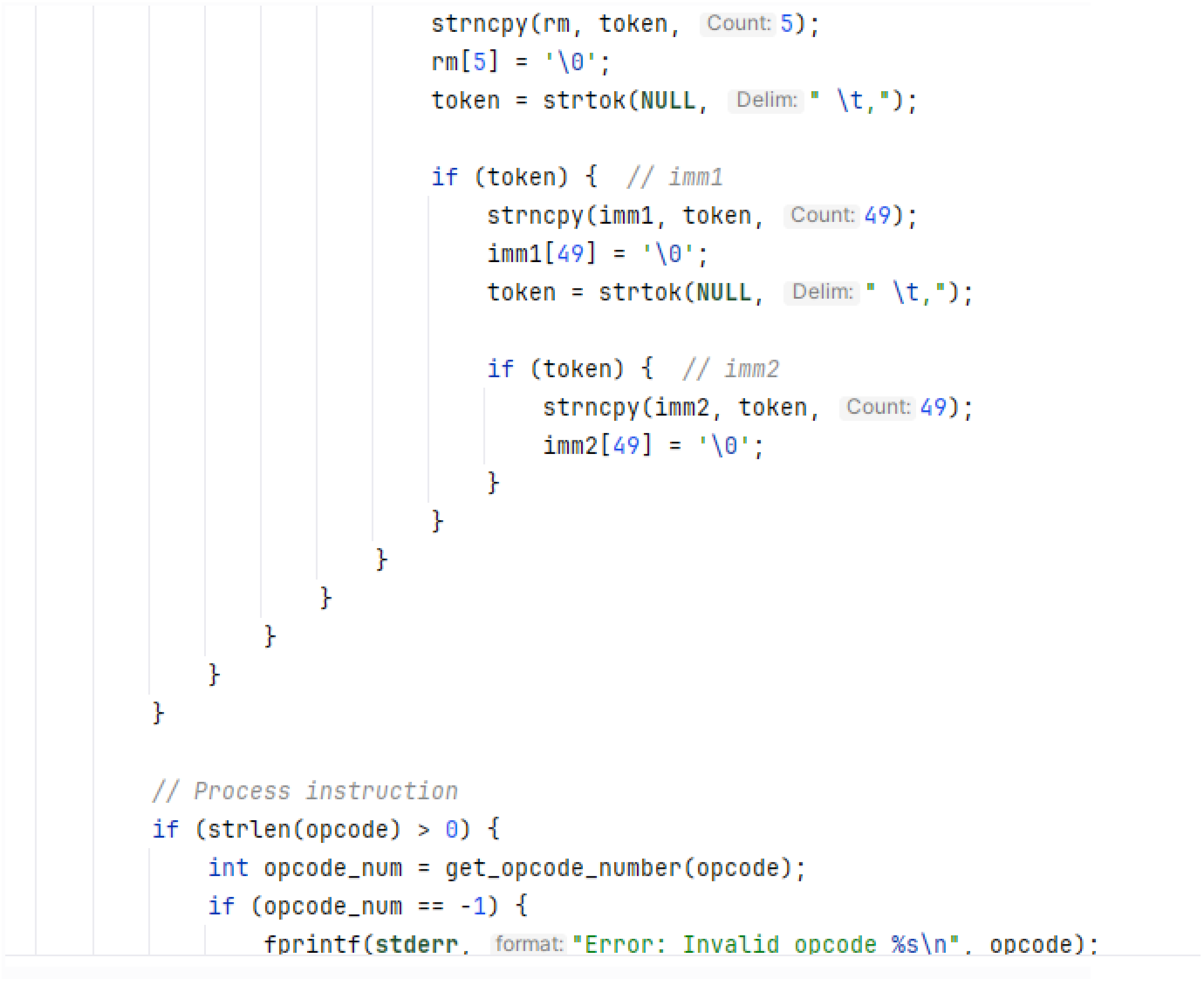
Effect on Assembler:
Transforms the assembly instructions into executable machine code by encoding opcodes, registers, and immediate values. It ensures that all label references are correctly mapped to their memory addresses, enabling accurate program execution by the SIMP processor.



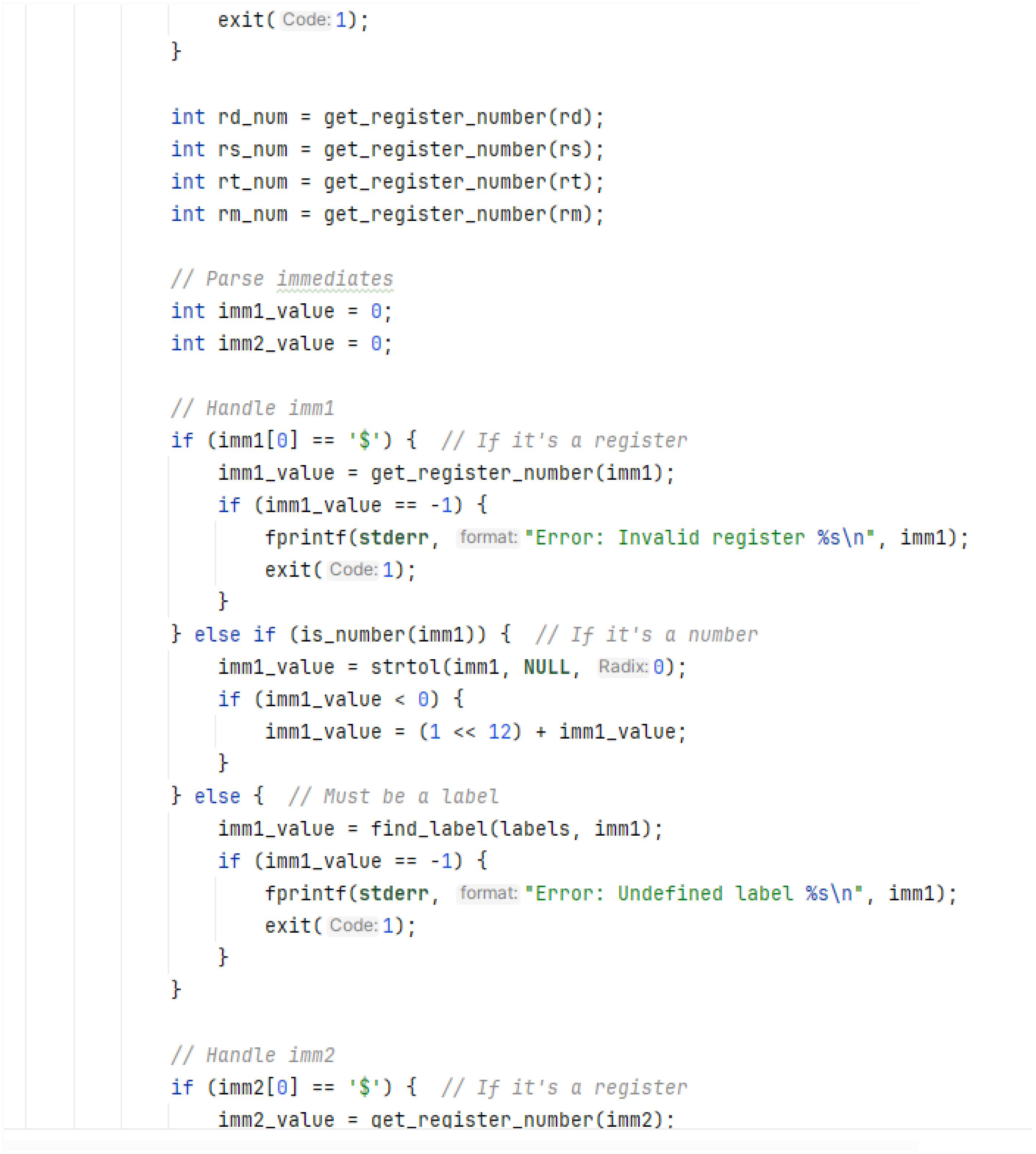
```c
void second_pass(FILE* input, FILE* imemin, FILE* dmemin, Label* labels) {
    char line[MAX_LINE_LENGTH];
    int current_address = 0;
    int dmem[MEMORY_SIZE] = {0};
    int max_dmem_address = 64;

    rewind(input);

    while (fgets(line, MaxCount: MAX_LINE_LENGTH, input)) {
        char original_line[MAX_LINE_LENGTH];
        strcpy(original_line, line);

        // Remove comments
        char* comment = strchr(line, Val: '#');
        if (comment) *comment = '\0';

        trim(line);
        if (strlen(line) == 0) continue;

        // Handle .word directive
        if (strstr( Str: line, SubStr: ".word") != NULL) {
            char* token = strtok(line, Delim: " \t");  // Get .word
            token = strtok(NULL, Delim: " \t");        // Get address
            int word_address = strtol(token, NULL, Radix: 0);
            token = strtok(NULL, Delim: " \t");        // Get value

            if (token) {
                trim(token);
                int value = strtol(token, NULL, Radix: 0);
                dmem[word_address] = value;
                if (word_address > max_dmem_address) {
                    max_dmem_address = word_address;
                }
            }
            continue;
        }

        // Skip label definitions
```



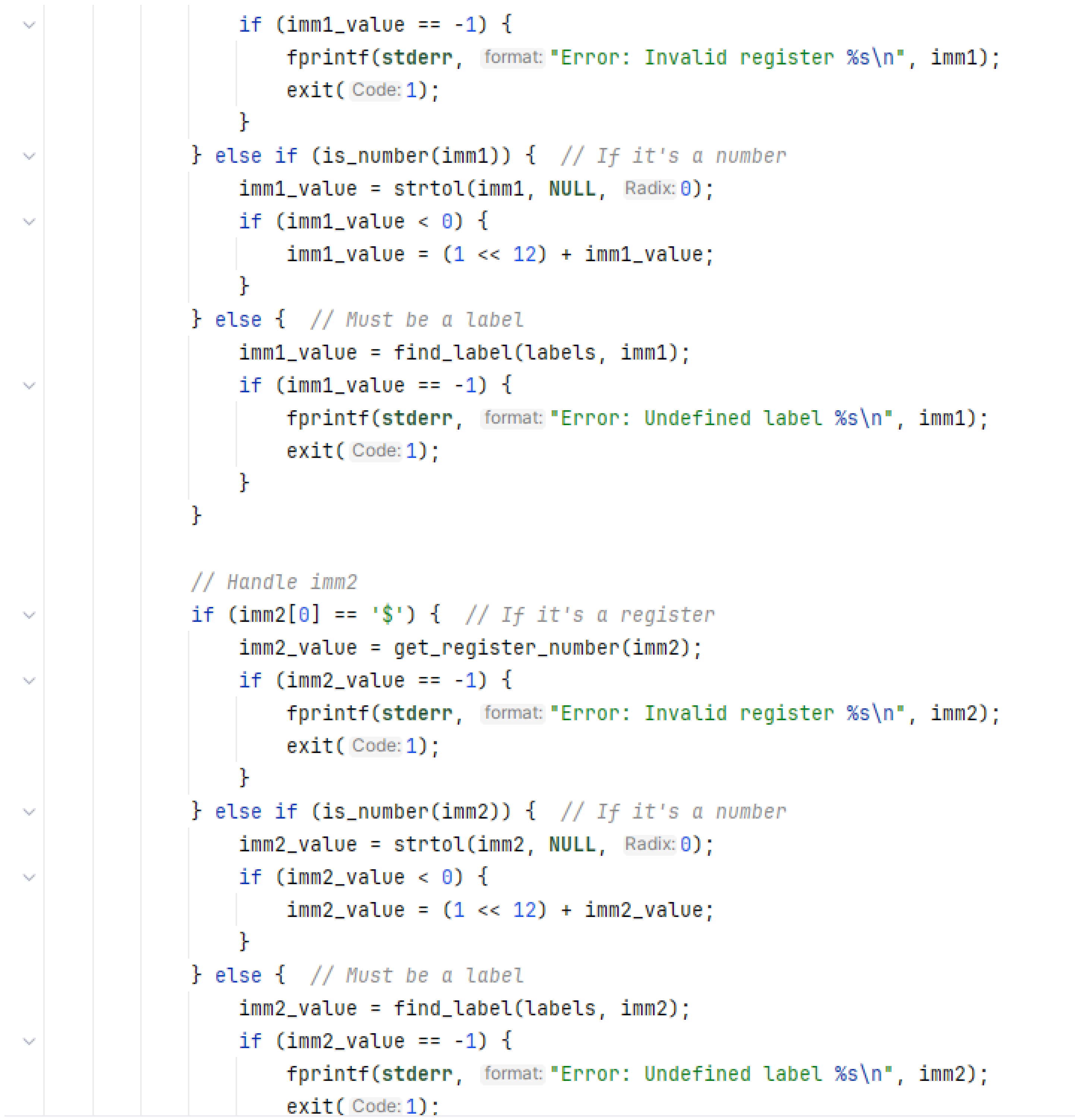
```c
char* colon = strchr(line, Val: ':');
if (colon) {
    char* instruction = colon + 1;
    trim(instruction);
    if (strlen(instruction) == 0) continue;
    strcpy(line, instruction);
}

// Initialize default operands
char opcode[6] = "", rd[6] = "$zero", rs[6] = "$zero", rt[6] = "$zero", rm[6] =
char imm1[50] = "0", imm2[50] = "0";

// Parse instruction
char* token = strtok(line, Delim: " \t,");

if (token) {
    strncpy(opcode, token, Count: 5);
    opcode[5] = '\0';
    token = strtok(NULL, Delim: " \t,");

    if (token) {  // rd
        strncpy(rd, token, Count: 5);
        rd[5] = '\0';
        token = strtok(NULL, Delim: " \t,");

        if (token) {  // rs
            strncpy(rs, token, Count: 5);
            rs[5] = '\0';
            token = strtok(NULL, Delim: " \t,");

            if (token) {  // rt
                strncpy(rt, token, Count: 5);
                rt[5] = '\0';
                token = strtok(NULL, Delim: " \t,");

                if (token) {  // rm
```



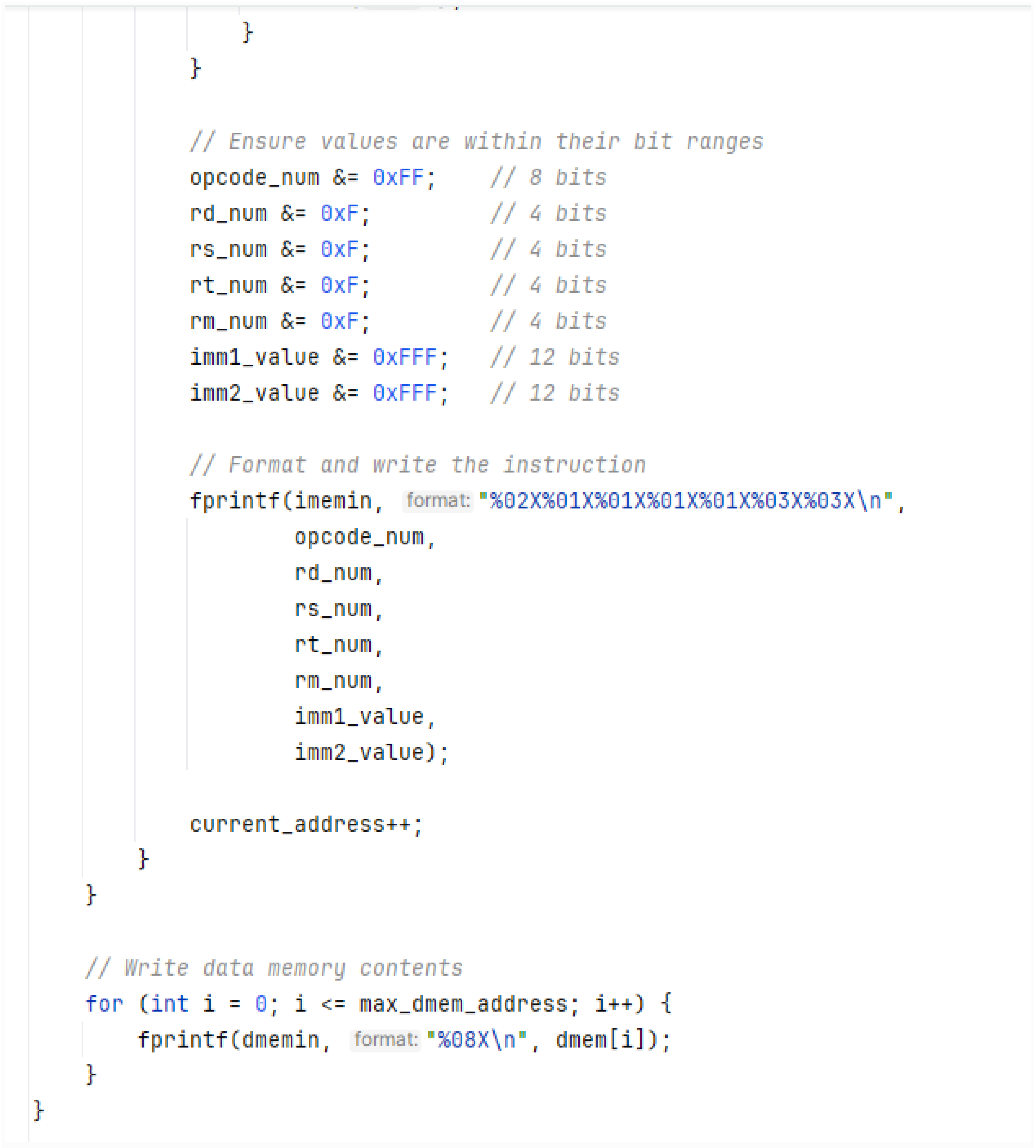
```
                            strncpy(rm, token, Count: 5);
                            rm[5] = '\0';
                            token = strtok(NULL, Delim: " \t,");

                            if (token) {  // imm1
                                strncpy(imm1, token, Count: 49);
                                imm1[49] = '\0';
                                token = strtok(NULL, Delim: " \t,");

                                if (token) {  // imm2
                                    strncpy(imm2, token, Count: 49);
                                    imm2[49] = '\0';
                                }
                            }
                        }
                    }
                }
            }
        }

        // Process instruction
        if (strlen(opcode) > 0) {
            int opcode_num = get_opcode_number(opcode);
            if (opcode_num == -1) {
                fprintf(stderr, format: "Error: Invalid opcode %s\n", opcode);
```



```c
                exit( Code: 1);
            }

            int rd_num = get_register_number(rd);
            int rs_num = get_register_number(rs);
            int rt_num = get_register_number(rt);
            int rm_num = get_register_number(rm);

            // Parse immediates
            int imm1_value = 0;
            int imm2_value = 0;

            // Handle imm1
            if (imm1[0] == '$') {  // If it's a register
                imm1_value = get_register_number(imm1);
                if (imm1_value == -1) {
                    fprintf(stderr,  format: "Error: Invalid register %s\n", imm1);
                    exit( Code: 1);
                }
            } else if (is_number(imm1)) {  // If it's a number
                imm1_value = strtol(imm1, NULL,  Radix: 0);
                if (imm1_value < 0) {
                    imm1_value = (1 << 12) + imm1_value;
                }
            } else {  // Must be a label
                imm1_value = find_label(labels, imm1);
                if (imm1_value == -1) {
                    fprintf(stderr,  format: "Error: Undefined label %s\n", imm1);
                    exit( Code: 1);
                }
            }

            // Handle imm2
            if (imm2[0] == '$') {  // If it's a register
                imm2_value = get_register_number(imm2);
```



```
                if (imm1_value == -1) {
                    fprintf(stderr,  format: "Error: Invalid register %s\n", imm1);
                    exit( Code: 1);
                }
            } else if (is_number(imm1)) {  // If it's a number
                imm1_value = strtol(imm1, NULL,  Radix: 0);
                if (imm1_value < 0) {
                    imm1_value = (1 << 12) + imm1_value;
                }
            } else {  // Must be a label
                imm1_value = find_label(labels, imm1);
                if (imm1_value == -1) {
                    fprintf(stderr,  format: "Error: Undefined label %s\n", imm1);
                    exit( Code: 1);
                }
            }

            // Handle imm2
            if (imm2[0] == '$') {  // If it's a register
                imm2_value = get_register_number(imm2);
                if (imm2_value == -1) {
                    fprintf(stderr,  format: "Error: Invalid register %s\n", imm2);
                    exit( Code: 1);
                }
            } else if (is_number(imm2)) {  // If it's a number
                imm2_value = strtol(imm2, NULL,  Radix: 0);
                if (imm2_value < 0) {
                    imm2_value = (1 << 12) + imm2_value;
                }
            } else {  // Must be a label
                imm2_value = find_label(labels, imm2);
                if (imm2_value == -1) {
                    fprintf(stderr,  format: "Error: Undefined label %s\n", imm2);
                    exit( Code: 1);
```



```
                }
            }

            // Ensure values are within their bit ranges
            opcode_num &= 0xFF;    // 8 bits
            rd_num &= 0xF;         // 4 bits
            rs_num &= 0xF;         // 4 bits
            rt_num &= 0xF;         // 4 bits
            rm_num &= 0xF;         // 4 bits
            imm1_value &= 0xFFF;   // 12 bits
            imm2_value &= 0xFFF;   // 12 bits

            // Format and write the instruction
            fprintf(imemin, format: "%02X%01X%01X%01X%01X%03X%03X\n",
                    opcode_num,
                    rd_num,
                    rs_num,
                    rt_num,
                    rm_num,
                    imm1_value,
                    imm2_value);

            current_address++;
        }
    }

    // Write data memory contents
    for (int i = 0; i <= max_dmem_address; i++) {
        fprintf(dmemin, format: "%08X\n", dmem[i]);
    }
}
```

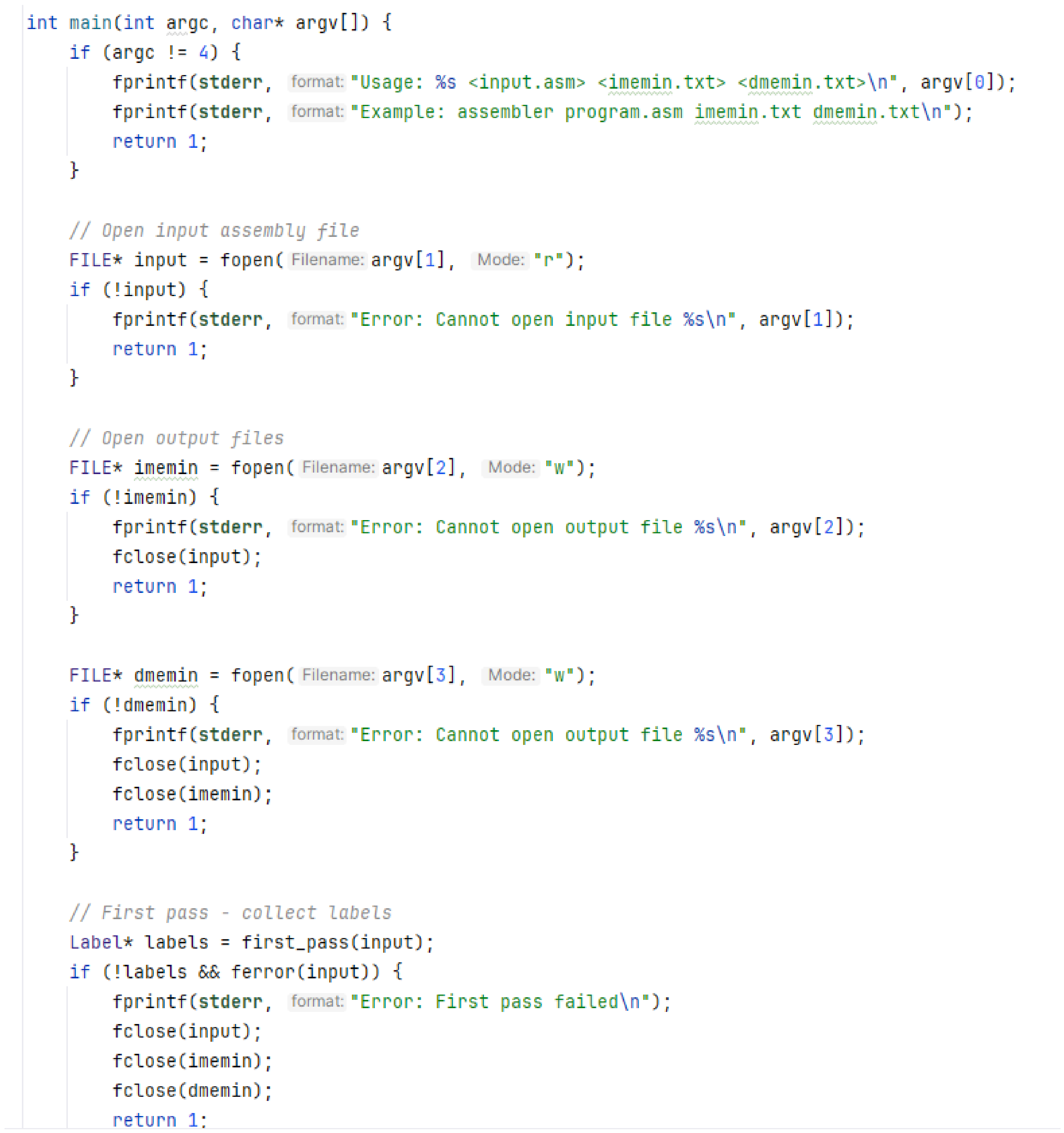
Main Function

main(int argc, char* argv[])

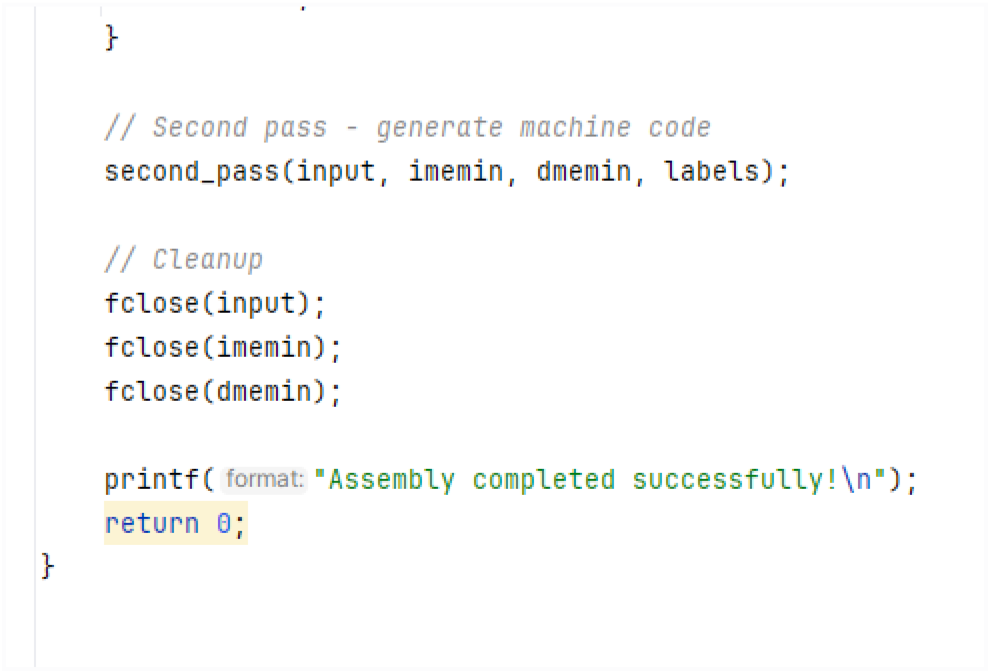
Purpose:
Serves as the entry point for the assembler program. It manages the flow of the assembly process by handling file operations, orchestrating the two-pass assembly process, and ensuring proper resource management.

Effect on Assembler:
Coordinates the overall execution of the assembler by:

1. Validating command-line arguments.
2. Opening necessary input and output files.
3. Executing the first pass to collect labels.
4. Performing the second pass to generate machine code and data memory initialization.
5. Cleaning up allocated resources to maintain optimal memory usage.
6. Providing user feedback upon successful assembly completion.



```
int main(int argc, char* argv[]) {
    if (argc != 4) {
        fprintf(stderr,  format: "Usage: %s <input.asm> <imemin.txt> <dmemin.txt>\n", argv[0]);
        fprintf(stderr,  format: "Example: assembler program.asm imemin.txt dmemin.txt\n");
        return 1;
    }

    // Open input assembly file
    FILE* input = fopen( Filename: argv[1],  Mode: "r");
    if (!input) {
        fprintf(stderr,  format: "Error: Cannot open input file %s\n", argv[1]);
        return 1;
    }

    // Open output files
    FILE* imemin = fopen( Filename: argv[2],  Mode: "w");
    if (!imemin) {
        fprintf(stderr,  format: "Error: Cannot open output file %s\n", argv[2]);
        fclose(input);
        return 1;
    }

    FILE* dmemin = fopen( Filename: argv[3],  Mode: "w");
    if (!dmemin) {
        fprintf(stderr,  format: "Error: Cannot open output file %s\n", argv[3]);
        fclose(input);
        fclose(imemin);
        return 1;
    }

    // First pass - collect labels
    Label* labels = first_pass(input);
    if (!labels && ferror(input)) {
        fprintf(stderr,  format: "Error: First pass failed\n");
        fclose(input);
        fclose(imemin);
        fclose(dmemin);
        return 1;
```



```
    }

    // Second pass - generate machine code
    second_pass(input, imemin, dmemin, labels);

    // Cleanup
    fclose(input);
    fclose(imemin);
    fclose(dmemin);

    printf( format: "Assembly completed successfully!\n");
    return 0;
}
```

Error Handling

The assembler includes robust error handling mechanisms to ensure reliability and correctness:

- **Invalid Opcodes:**
  Detects and reports any unrecognized instruction mnemonics in the assembly code.
- **Undefined Labels:**
  Identifies and flags any label references that do not have corresponding definitions.
- **Invalid Register Names:**
  Validates register identifiers and reports any that do not conform to the expected naming conventions.
- **Out-of-Range Immediate Values:**
  Ensures that immediate values fit within their designated bit sizes, preventing overflow or underflow issues.
- **File I/O Errors:**
  Checks for issues related to file access, such as inability to open input or output files, and reports them accordingly.

Upon encountering an error, the assembler terminates the assembly process and provides a descriptive error message to aid in debugging.

Usage

The assembler is executed via the command line with the following syntax:

```
assembler <input.asm> <imemin.txt> <dmemin.txt>
```

Parameters:

- <input.asm>: Path to the input assembly file containing SIMP assembly language code.
- <imemin.txt>: Path to the output file where the encoded instruction memory will be written in hexadecimal format.
- <dmemin.txt>: Path to the output file where the data memory initialization values will be written in hexadecimal format.

Simulator - Simulator.c

The SIMP Processor Simulator is designed to simulate the execution of programs written for the SIMP architecture. It emulates the processor core, memory hierarchy, I/O devices, interrupts, and provides a detailed trace of program execution. The simulator reads input files representing the initial state of instruction memory, data memory, disk contents, and IRQ2 interrupts timing, executes the program, and generates multiple output files detailing the final state and execution traces.

Design Constraints

The simulator is built to support programs of theoretically infinite size, limited only by available memory. To achieve this, the following constraints are imposed:

- Memory Sizes:
    - Instruction Memory (imem): Up to 4096 words.
    - Data Memory (dmem): Up to 4096 words.
    - Disk Size: 128 sectors, with 128 words per sector (total of 16384 words).

    - Monitor Buffer: 256x256 pixels.
- Line Lengths:
    - Maximum Line Length for Reading Input Files: Up to 500 characters.
    - This is enforced by the macro MAX_LINE_LENGTH, defined with a value of 500.
- Standard Libraries:
    - The simulator relies on standard C libraries: stdio.h, stdlib.h, string.h, and stdint.h.
    - These libraries are required for memory management and I/O operations and should be included during compilation and execution as per project guidelines.

Data Structures

To manage the processor state and support large programs efficiently, the simulator employs several key data structures. This approach mimics object-oriented programming concepts within the constraints of the C language.

Processor Structure (**Processor**) contains:

- CPU Registers and State:
    - uint32_t registers[16];
      Holds the values of registers R0 to R15.
    - uint32_t pc;
      Represents the Program Counter, indicating the address of the next instruction to execute.
    - uint64_t imem[MEMORY_SIZE];
      Instruction memory array storing the program's instructions.
    - uint32_t dmem[MEMORY_SIZE];
      Data memory array for storing data during execution.
- Interrupt Registers:
    - uint32_t irq0enable;
      Enables or disables IRQ0 interrupts.
    - uint32_t irq1enable;
      Enables or disables IRQ1 interrupts.
    - uint32_t irq2enable;
      Enables or disables IRQ2 interrupts.
    - uint32_t irq0status;
      Status flag for IRQ0 interrupts.

    - uint32_t irq1status;
      Status flag for IRQ1 interrupts.
    - uint32_t irq2status;
      Status flag for IRQ2 interrupts.
    - uint32_t irqhandler;
      Address of the interrupt handler routine.
    - uint32_t irqreturn;
      Address to return to after handling an interrupt.
    - int in_interrupt;
      Indicates whether the processor is currently handling an interrupt.
- Timer Registers:
    - uint32_t timerenable;
      Enables or disables the timer.
    - uint32_t timercurrent;
      Current value of the timer.
    - uint32_t timermax;
      Maximum value of the timer before triggering an interrupt.
- Disk Registers:
    - uint32_t diskcmd;
      Command register for disk operations (e.g., read, write).
    - uint32_t disksector;
      Specifies the disk sector for the operation.
    - uint32_t diskbuffer;
      Memory address used as a buffer for disk data transfers.
    - uint32_t diskstatus;
      Status flag indicating if the disk is busy or ready.
    - uint32_t disk_busy_cycles;
      Counter to simulate disk operation delays.
- Monitor Registers:
    - uint32_t monitoraddr;
      Address in the monitor buffer to read or write.
    - uint32_t monitordata;
      Data value to write to the monitor buffer.
    - uint32_t monitorcmd;
      Command register for monitor operations.
    - uint8_t monitor_buffer[MONITOR_SIZE][MONITOR_SIZE];
      2D array representing the monitor's pixel data.
- I/O Registers:
    - uint32_t leds;
      Simulates LED outputs.

    - uint32_t display7seg;
      Simulates a 7-segment display.
  - Simulation State:
    - uint32_t cycle_counter;
      Tracks the number of cycles executed in the simulation.
    - int halt;
      Flag indicating whether the simulation should halt.

Instruction Structure (**Instruction**):

- Opcode and Operands: Stores the operation code and associated registers (rd, rs, rt, rm) used by the instruction.
- Immediate Values: Holds immediate values extracted from the instruction word, supporting immediate addressing modes.

Initialization Functions

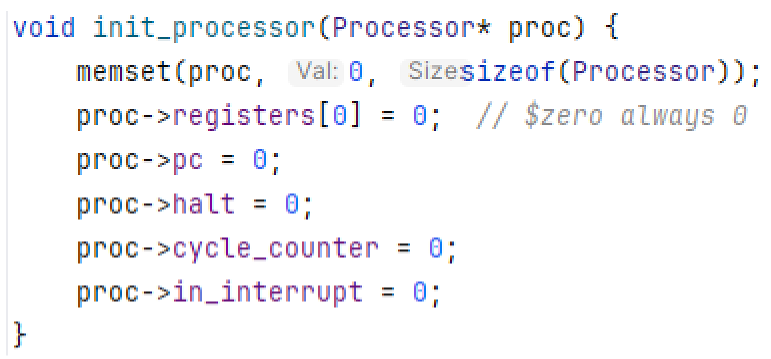
init_processor(Processor proc)

Purpose:

- Initializes the processor state by setting all fields of the Processor structure to zero.
- Ensures the processor starts from a known state with registers cleared, the program counter reset, and control flags initialized.

Effect on Simulator:

- Prepares the simulator for execution, preventing residual data from affecting the simulation.
- Sets registers[0] to zero, enforcing the convention that register R0 is always zero in the SIMP architecture.



```
void init_processor(Processor* proc) {
    memset(proc, Val: 0, Size: sizeof(Processor));
    proc->registers[0] = 0;  // $zero always 0
    proc->pc = 0;
    proc->halt = 0;
    proc->cycle_counter = 0;
    proc->in_interrupt = 0;
}
```

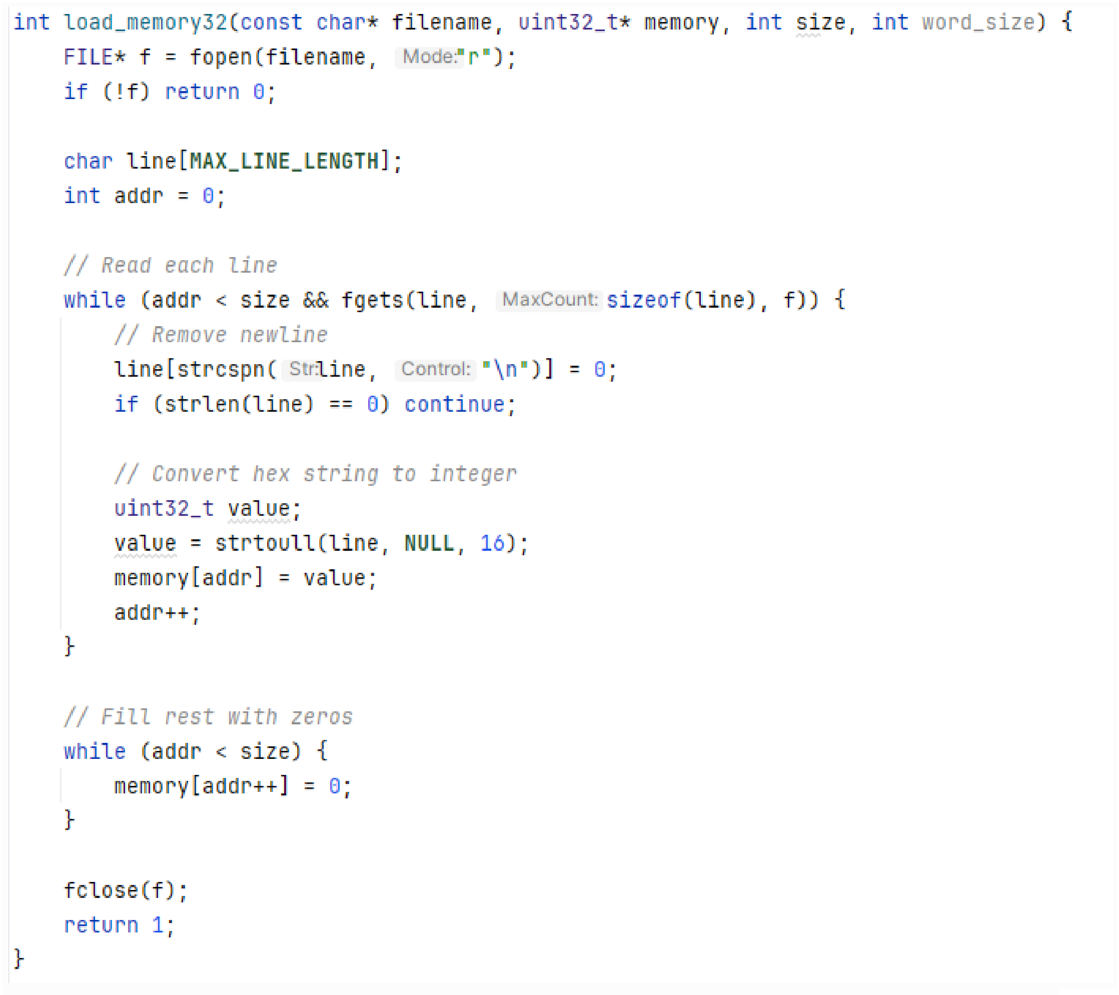
load_memory32(const char filename, uint32_t memory, int size, int word_size)**

Purpose:

- Reads 32-bit hexadecimal values from a file and loads them into the specified memory array.
- Used to initialize data memory (dmem) and disk contents from input files.

Effect on Simulator:

- Populates the simulator's memory arrays with initial data, essential for accurate program execution.
- Handles empty lines and ensures the entire memory array is filled, setting unused memory locations to zero.



```
int load_memory32(const char* filename, uint32_t* memory, int size, int word_size) {
    FILE* f = fopen(filename,  Mode:"r");
    if (!f) return 0;

    char line[MAX_LINE_LENGTH];
    int addr = 0;

    // Read each line
    while (addr < size && fgets(line,  MaxCount: sizeof(line), f)) {
        // Remove newline
        line[strcspn( Str:line,  Control: "\n")] = 0;
        if (strlen(line) == 0) continue;

        // Convert hex string to integer
        uint32_t value;
        value = strtoull(line, NULL, 16);
        memory[addr] = value;
        addr++;
    }

    // Fill rest with zeros
    while (addr < size) {
        memory[addr++] = 0;
    }

    fclose(f);
    return 1;
}
```

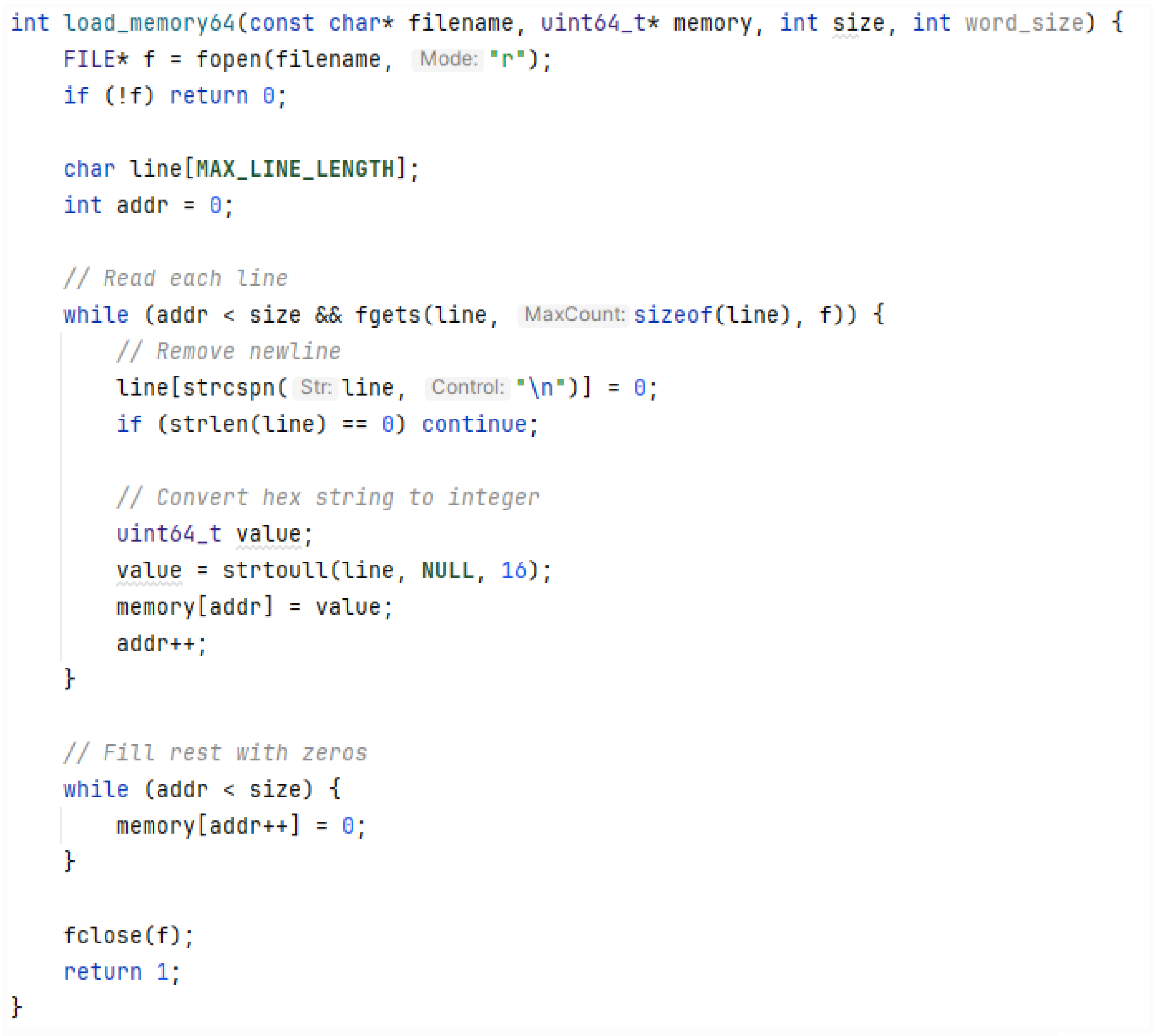
load_memory64(const char filename, uint64_t memory, int size, int word_size)

Purpose:

- Similar to load_memory32, but reads 64-bit hexadecimal values to load the instruction memory (imem).

Effect on Simulator:

- Initializes the instruction memory with the program to be simulated.
- Essential for fetching and executing instructions during simulation.



```
int load_memory64(const char* filename, uint64_t* memory, int size, int word_size) {
    FILE* f = fopen(filename, Mode: "r");
    if (!f) return 0;

    char line[MAX_LINE_LENGTH];
    int addr = 0;

    // Read each line
    while (addr < size && fgets(line, MaxCount: sizeof(line), f)) {
        // Remove newline
        line[strcspn( Str: line, Control: "\n")] = 0;
        if (strlen(line) == 0) continue;

        // Convert hex string to integer
        uint64_t value;
        value = strtoull(line, NULL, 16);
        memory[addr] = value;
        addr++;
    }

    // Fill rest with zeros
    while (addr < size) {
        memory[addr++] = 0;
    }

    fclose(f);
    return 1;
}
```

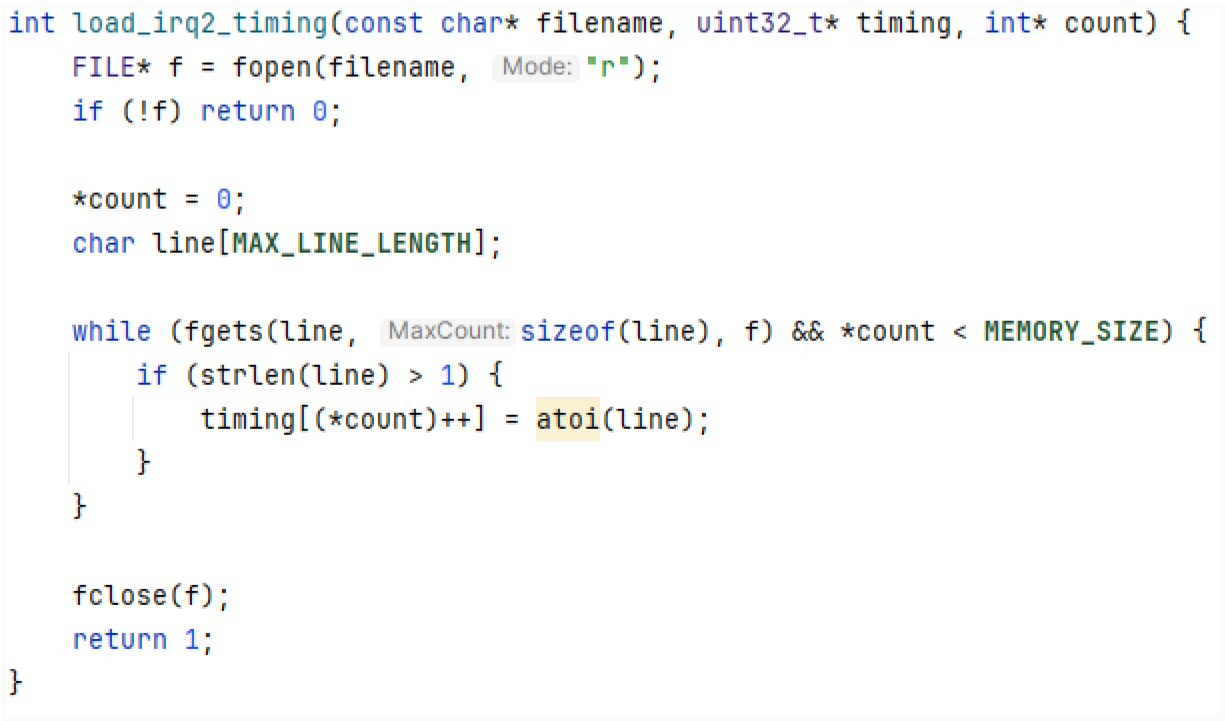
load_irq2_timing(const char filename, uint32_t timing, int* count)

Purpose:

- Reads IRQ2 interrupt timings from a file and stores them in an array.
- The count parameter tracks the number of timing entries read.

Effect on Simulator:

- Provides the simulator with the cycles at which IRQ2 interrupts should be triggered.
- Enables accurate simulation of external interrupts occurring at specific times.



```
int load_irq2_timing(const char* filename, uint32_t* timing, int* count) {
    FILE* f = fopen(filename,  Mode: "r");
    if (!f) return 0;

    *count = 0;
    char line[MAX_LINE_LENGTH];

    while (fgets(line,  MaxCount: sizeof(line), f) && *count < MEMORY_SIZE) {
        if (strlen(line) > 1) {
            timing[(*count)++] = atoi(line);
        }
    }

    fclose(f);
    return 1;
}
```

Instruction Handling Functions

decode_instruction(uint64_t word)

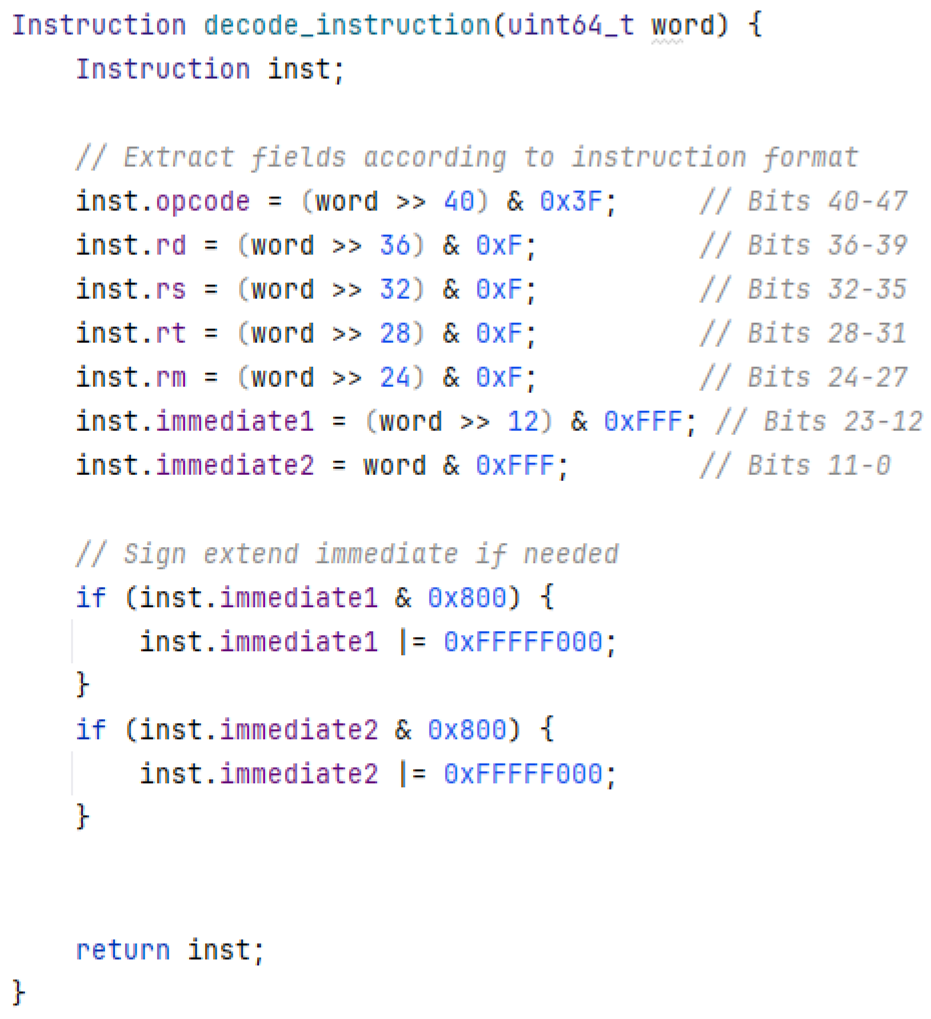
Purpose:

- Decodes a 64-bit instruction word into its components: opcode, registers (rd, rs, rt, rm), and immediate values (immediate1, immediate2).
- Performs sign extension on immediate values if necessary.

Effect on Simulator:

- Transforms raw instruction words into a structured format for execution.

- Simplifies the execution logic by providing easy access to instruction parts.



```
Instruction decode_instruction(uint64_t word) {
    Instruction inst;

    // Extract fields according to instruction format
    inst.opcode = (word >> 40) & 0x3F;     // Bits 40-47
    inst.rd = (word >> 36) & 0xF;          // Bits 36-39
    inst.rs = (word >> 32) & 0xF;          // Bits 32-35
    inst.rt = (word >> 28) & 0xF;          // Bits 28-31
    inst.rm = (word >> 24) & 0xF;          // Bits 24-27
    inst.immediate1 = (word >> 12) & 0xFFF; // Bits 23-12
    inst.immediate2 = word & 0xFFF;        // Bits 11-0

    // Sign extend immediate if needed
    if (inst.immediate1 & 0x800) {
        inst.immediate1 |= 0xFFFFF000;
    }
    if (inst.immediate2 & 0x800) {
        inst.immediate2 |= 0xFFFFF000;
    }


    return inst;
}
```

execute_instruction(Processor proc, Instruction inst)

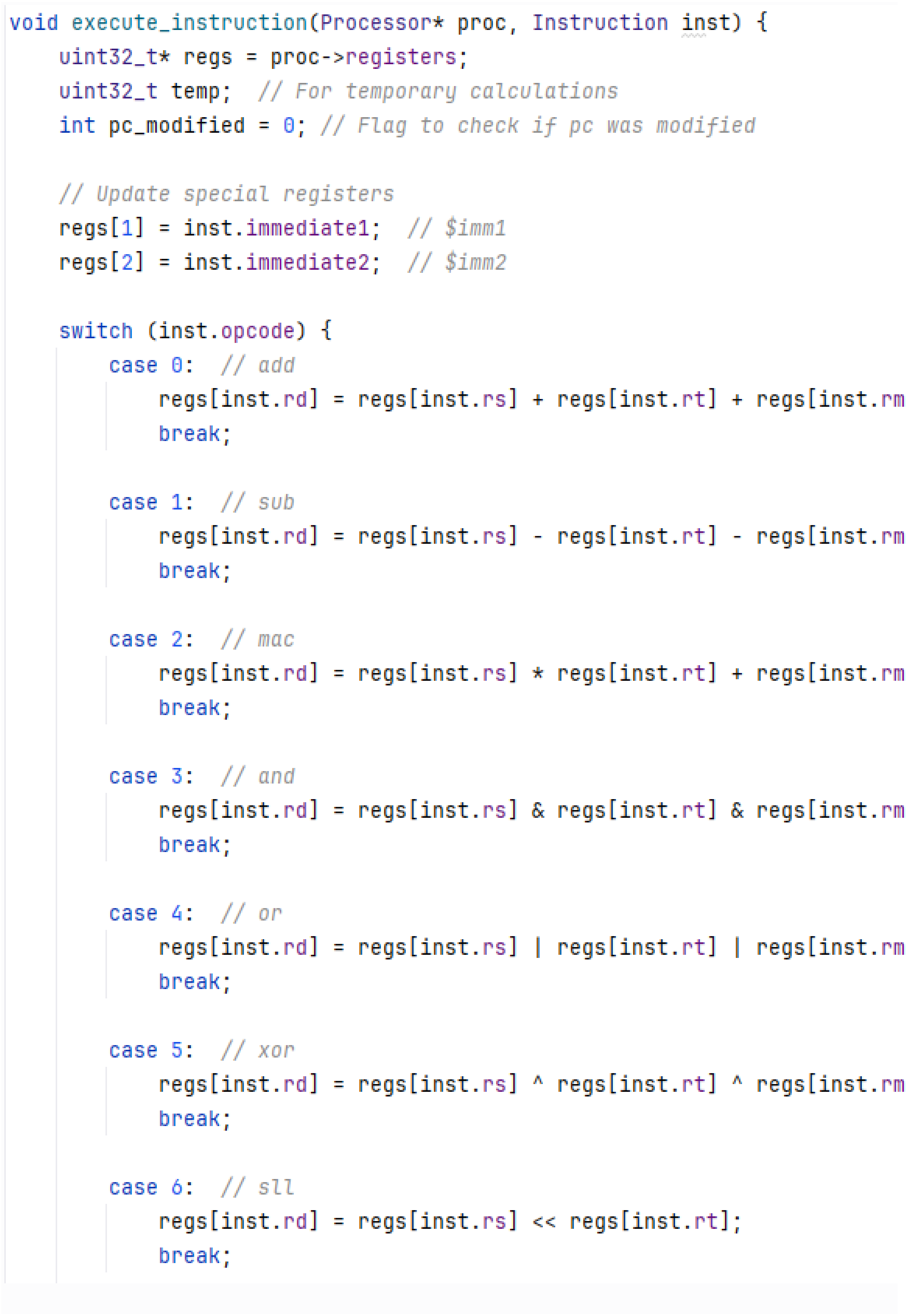
Purpose:

- Executes a decoded instruction, modifying the processor's state accordingly.
- Handles all instruction types defined in the SIMP architecture, including arithmetic operations, memory access, control flow, and I/O operations.

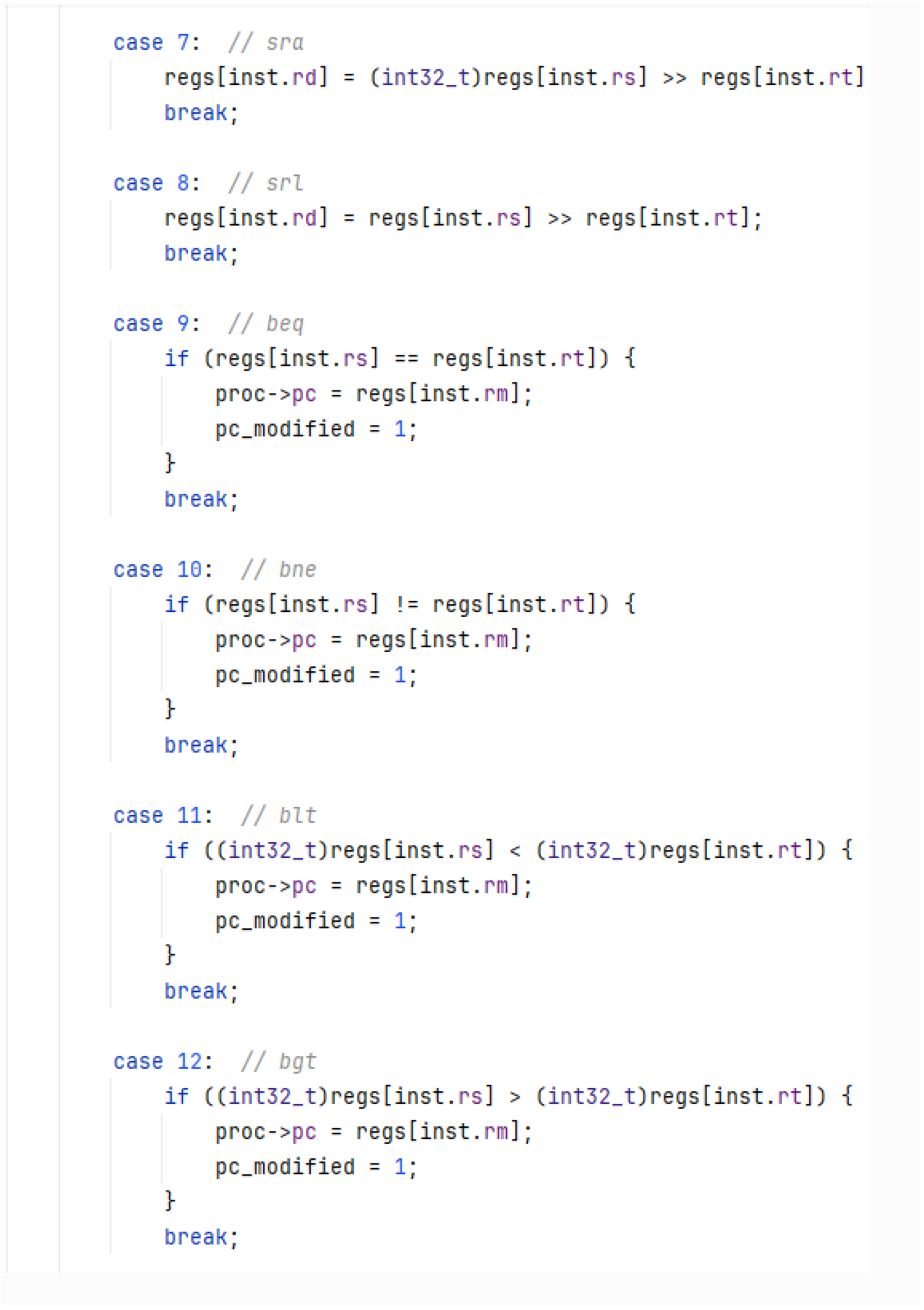
Effect on Simulator:

- Advances the simulation by performing the actions specified by each instruction.

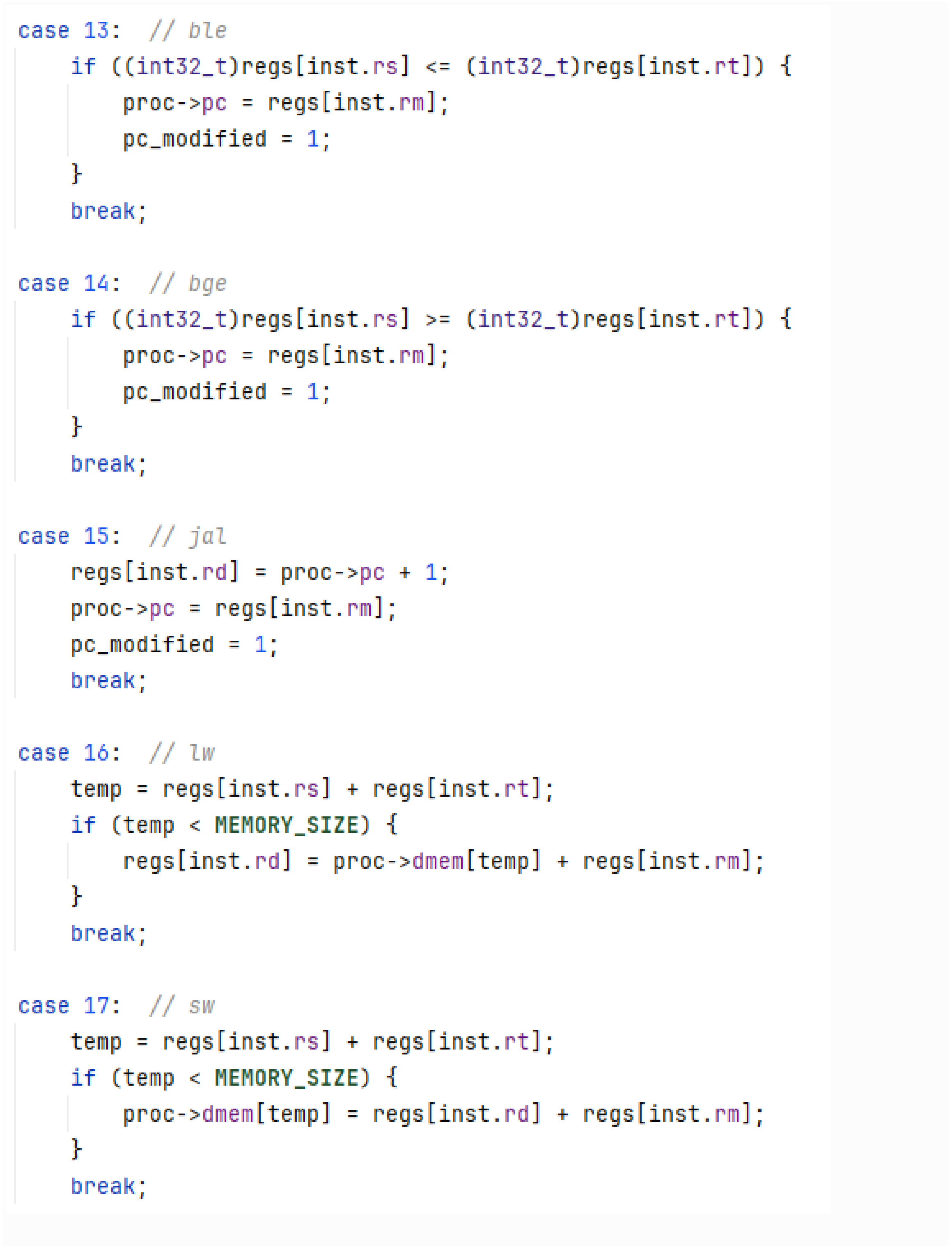
- Updates registers, memory, and program counter, and handles special cases like interrupts and halting.
- Ensures register R0 remains zero after execution, maintaining architectural conventions.



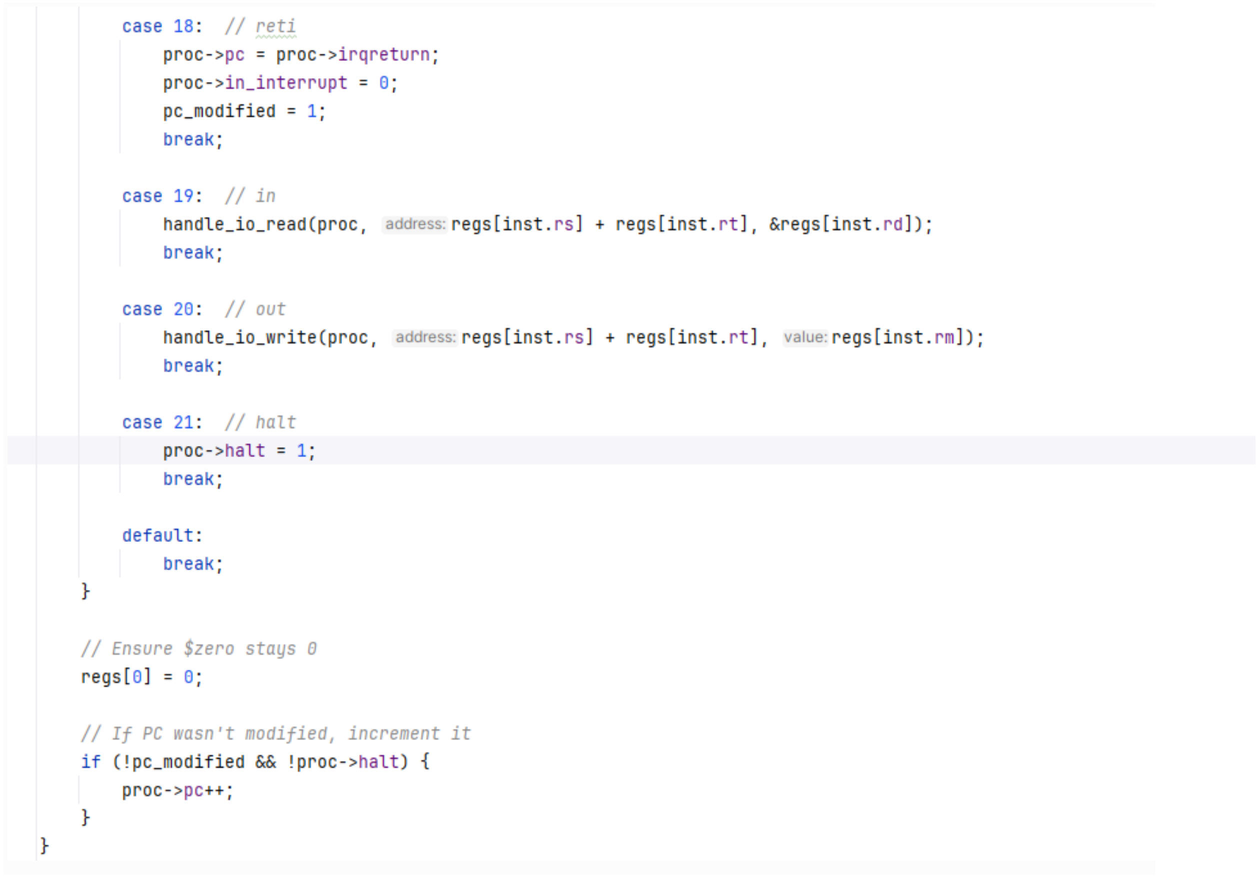
```
void execute_instruction(Processor* proc, Instruction inst) {
    uint32_t* regs = proc->registers;
    uint32_t temp;  // For temporary calculations
    int pc_modified = 0; // Flag to check if pc was modified

    // Update special registers
    regs[1] = inst.immediate1;  // $imm1
    regs[2] = inst.immediate2;  // $imm2

    switch (inst.opcode) {
        case 0:  // add
            regs[inst.rd] = regs[inst.rs] + regs[inst.rt] + regs[inst.rm];
            break;

        case 1:  // sub
            regs[inst.rd] = regs[inst.rs] - regs[inst.rt] - regs[inst.rm];
            break;

        case 2:  // mac
            regs[inst.rd] = regs[inst.rs] * regs[inst.rt] + regs[inst.rm];
            break;

        case 3:  // and
            regs[inst.rd] = regs[inst.rs] & regs[inst.rt] & regs[inst.rm];
            break;

        case 4:  // or
            regs[inst.rd] = regs[inst.rs] | regs[inst.rt] | regs[inst.rm];
            break;

        case 5:  // xor
            regs[inst.rd] = regs[inst.rs] ^ regs[inst.rt] ^ regs[inst.rm];
            break;

        case 6:  // sll
            regs[inst.rd] = regs[inst.rs] << regs[inst.rt];
            break;
```



```
        case 7:  // sra
            regs[inst.rd] = (int32_t)regs[inst.rs] >> regs[inst.rt]
            break;

        case 8:  // srl
            regs[inst.rd] = regs[inst.rs] >> regs[inst.rt];
            break;

        case 9:  // beq
            if (regs[inst.rs] == regs[inst.rt]) {
                proc->pc = regs[inst.rm];
                pc_modified = 1;
            }
            break;

        case 10:  // bne
            if (regs[inst.rs] != regs[inst.rt]) {
                proc->pc = regs[inst.rm];
                pc_modified = 1;
            }
            break;

        case 11:  // blt
            if ((int32_t)regs[inst.rs] < (int32_t)regs[inst.rt]) {
                proc->pc = regs[inst.rm];
                pc_modified = 1;
            }
            break;

        case 12:  // bgt
            if ((int32_t)regs[inst.rs] > (int32_t)regs[inst.rt]) {
                proc->pc = regs[inst.rm];
                pc_modified = 1;
            }
            break;
```



```c
case 13:  // ble
    if ((int32_t)regs[inst.rs] <= (int32_t)regs[inst.rt]) {
        proc->pc = regs[inst.rm];
        pc_modified = 1;
    }
    break;

case 14:  // bge
    if ((int32_t)regs[inst.rs] >= (int32_t)regs[inst.rt]) {
        proc->pc = regs[inst.rm];
        pc_modified = 1;
    }
    break;

case 15:  // jal
    regs[inst.rd] = proc->pc + 1;
    proc->pc = regs[inst.rm];
    pc_modified = 1;
    break;

case 16:  // lw
    temp = regs[inst.rs] + regs[inst.rt];
    if (temp < MEMORY_SIZE) {
        regs[inst.rd] = proc->dmem[temp] + regs[inst.rm];
    }
    break;

case 17:  // sw
    temp = regs[inst.rs] + regs[inst.rt];
    if (temp < MEMORY_SIZE) {
        proc->dmem[temp] = regs[inst.rd] + regs[inst.rm];
    }
    break;
```



```
        case 18:  // reti
            proc->pc = proc->irqreturn;
            proc->in_interrupt = 0;
            pc_modified = 1;
            break;

        case 19:  // in
            handle_io_read(proc,  address: regs[inst.rs] + regs[inst.rt], &regs[inst.rd]);
            break;

        case 20:  // out
            handle_io_write(proc,  address: regs[inst.rs] + regs[inst.rt],  value: regs[inst.rm]);
            break;

        case 21:  // halt
            proc->halt = 1;
            break;

        default:
            break;
    }

    // Ensure $zero stays 0
    regs[0] = 0;

    // If PC wasn't modified, increment it
    if (!pc_modified && !proc->halt) {
        proc->pc++;
    }
}
```

I/O and Device Management Functions

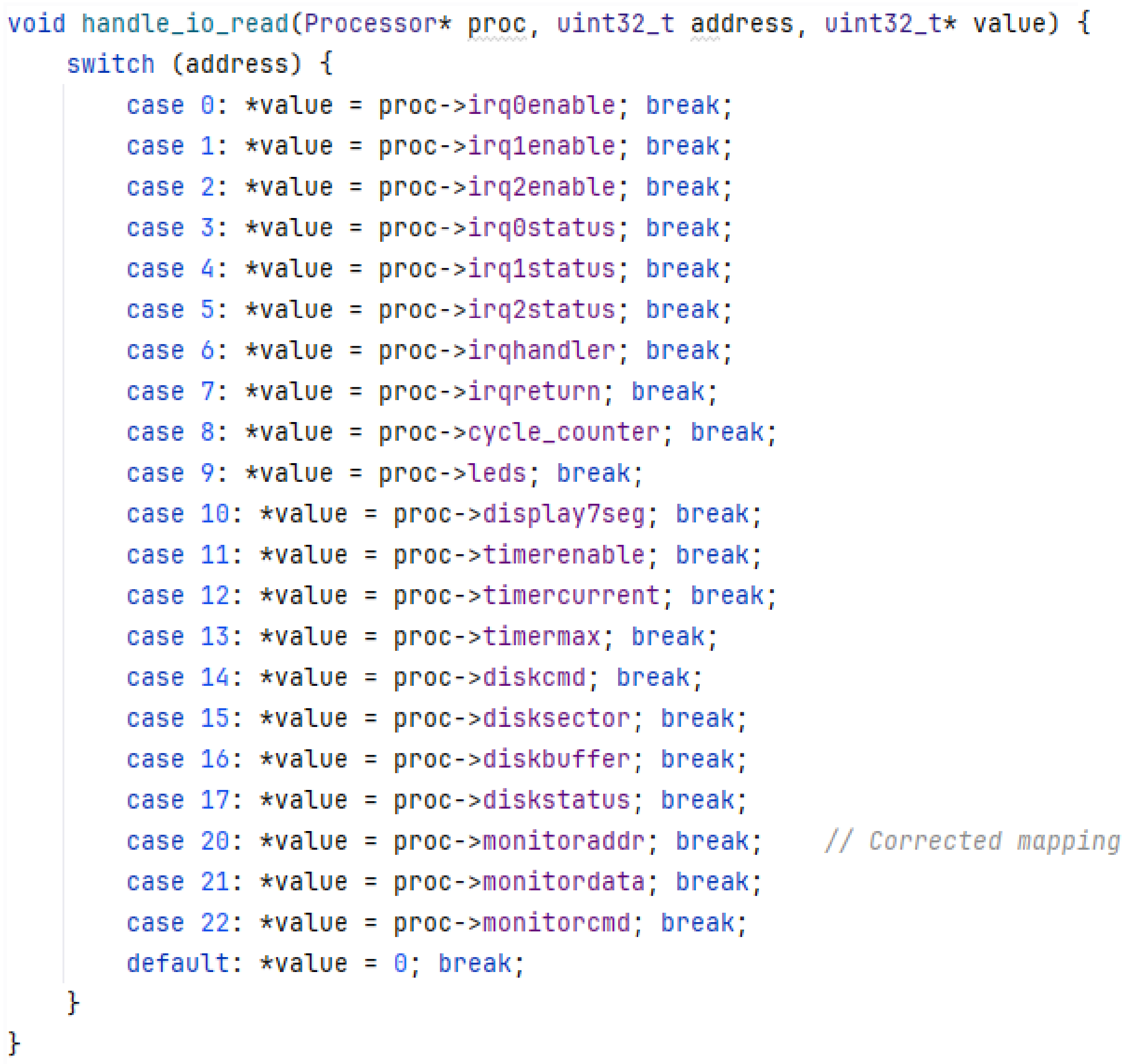
handle_io_read(Processor proc, uint32_t address, uint32_t value)

Purpose:

- Handles read operations from I/O registers based on the provided address.
- Retrieves the current value of the specified I/O register.

Effect on Simulator:

- Enables instructions to read from I/O devices, reflecting their current state.
- Supports the simulation of input operations within the processor's execution.



```c
void handle_io_read(Processor* proc, uint32_t address, uint32_t* value) {
    switch (address) {
        case 0: *value = proc->irq0enable; break;
        case 1: *value = proc->irq1enable; break;
        case 2: *value = proc->irq2enable; break;
        case 3: *value = proc->irq0status; break;
        case 4: *value = proc->irq1status; break;
        case 5: *value = proc->irq2status; break;
        case 6: *value = proc->irqhandler; break;
        case 7: *value = proc->irqreturn; break;
        case 8: *value = proc->cycle_counter; break;
        case 9: *value = proc->leds; break;
        case 10: *value = proc->display7seg; break;
        case 11: *value = proc->timerenable; break;
        case 12: *value = proc->timercurrent; break;
        case 13: *value = proc->timermax; break;
        case 14: *value = proc->diskcmd; break;
        case 15: *value = proc->disksector; break;
        case 16: *value = proc->diskbuffer; break;
        case 17: *value = proc->diskstatus; break;
        case 20: *value = proc->monitoraddr; break;    // Corrected mapping
        case 21: *value = proc->monitordata; break;
        case 22: *value = proc->monitorcmd; break;
        default: *value = 0; break;
    }
}
```

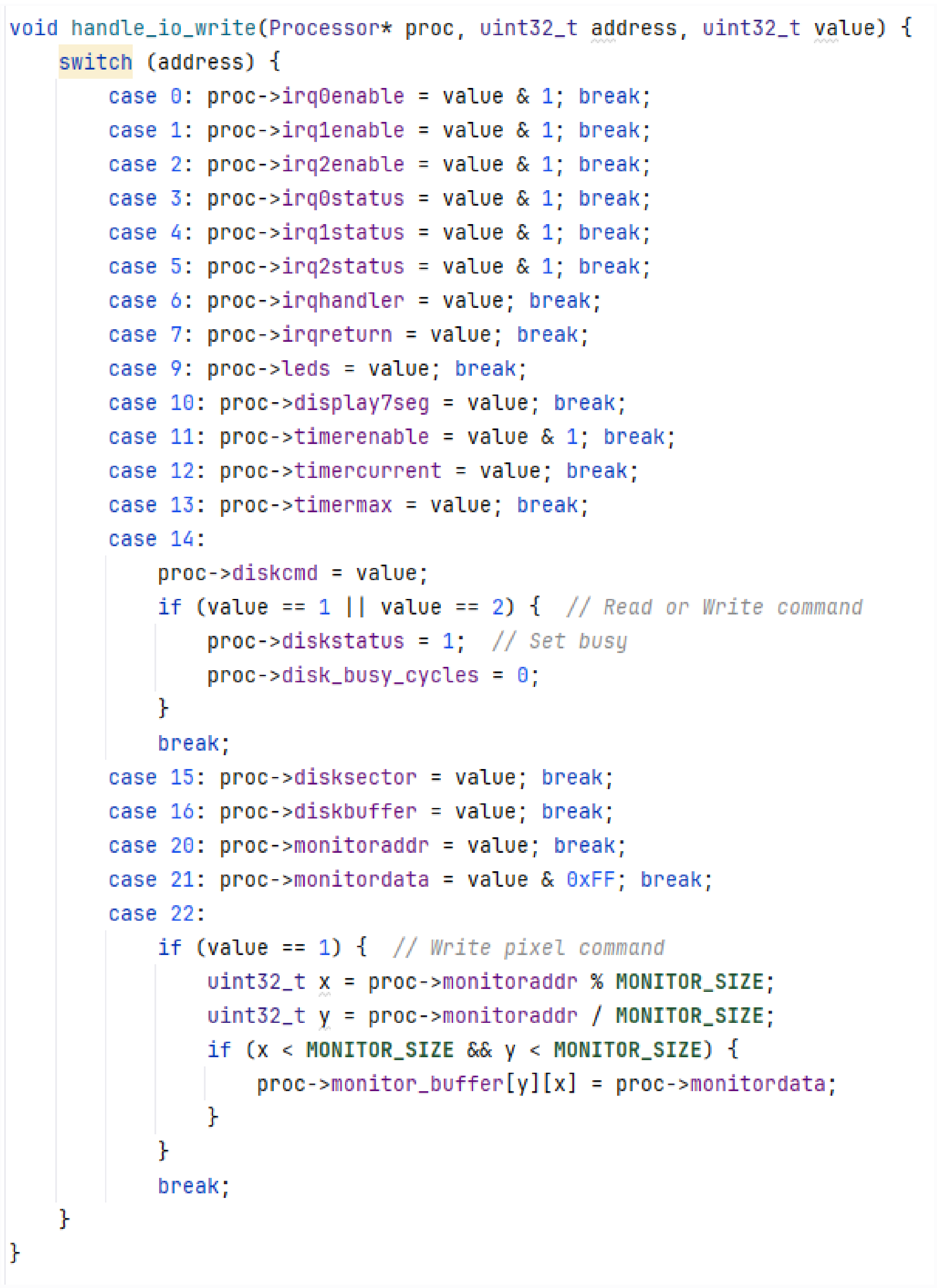
handle_io_write(Processor proc, uint32_t address, uint32_t value)*

Purpose:

- Handles write operations to I/O registers.
- Updates the state of I/O devices or triggers specific actions (e.g., starting a disk operation or updating the monitor display).

Effect on Simulator:

- Reflects changes to I/O device states resulting from program execution.
- Simulates the effect of output operations in the program, such as controlling LEDs or initiating disk reads/writes.



```c
void handle_io_write(Processor* proc, uint32_t address, uint32_t value) {
    switch (address) {
        case 0: proc->irq0enable = value & 1; break;
        case 1: proc->irq1enable = value & 1; break;
        case 2: proc->irq2enable = value & 1; break;
        case 3: proc->irq0status = value & 1; break;
        case 4: proc->irq1status = value & 1; break;
        case 5: proc->irq2status = value & 1; break;
        case 6: proc->irqhandler = value; break;
        case 7: proc->irqreturn = value; break;
        case 9: proc->leds = value; break;
        case 10: proc->display7seg = value; break;
        case 11: proc->timerenable = value & 1; break;
        case 12: proc->timercurrent = value; break;
        case 13: proc->timermax = value; break;
        case 14:
            proc->diskcmd = value;
            if (value == 1 || value == 2) {  // Read or Write command
                proc->diskstatus = 1;  // Set busy
                proc->disk_busy_cycles = 0;
            }
            break;
        case 15: proc->disksector = value; break;
        case 16: proc->diskbuffer = value; break;
        case 20: proc->monitoraddr = value; break;
        case 21: proc->monitordata = value & 0xFF; break;
        case 22:
            if (value == 1) {  // Write pixel command
                uint32_t x = proc->monitoraddr % MONITOR_SIZE;
                uint32_t y = proc->monitoraddr / MONITOR_SIZE;
                if (x < MONITOR_SIZE && y < MONITOR_SIZE) {
                    proc->monitor_buffer[y][x] = proc->monitordata;
                }
            }
            break;
    }
}
```

Interrupt Handling Functions

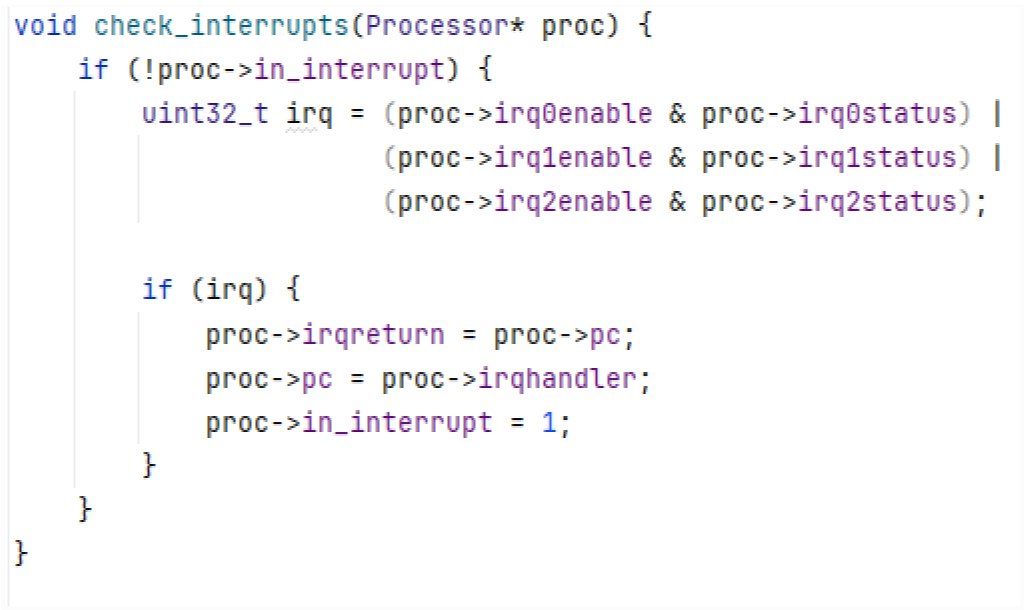
check_interrupts(Processor proc)

Purpose:

- Checks for pending interrupts (IRQ0, IRQ1, IRQ2) that are enabled and have their status flags set.
- If an interrupt is pending and the processor is not already handling an interrupt, it saves the current pc to irqreturn, sets the pc to irqhandler, and updates the in_interrupt flag.

Effect on Simulator:

- Ensures that interrupts are handled promptly and correctly.
- Simulates the processor's interrupt handling mechanism, allowing interrupt-driven programming.



```
void check_interrupts(Processor* proc) {
    if (!proc->in_interrupt) {
        uint32_t irq = (proc->irq0enable & proc->irq0status) |
                       (proc->irq1enable & proc->irq1status) |
                       (proc->irq2enable & proc->irq2status);

        if (irq) {
            proc->irqreturn = proc->pc;
            proc->pc = proc->irqhandler;
            proc->in_interrupt = 1;
        }
    }
}
```

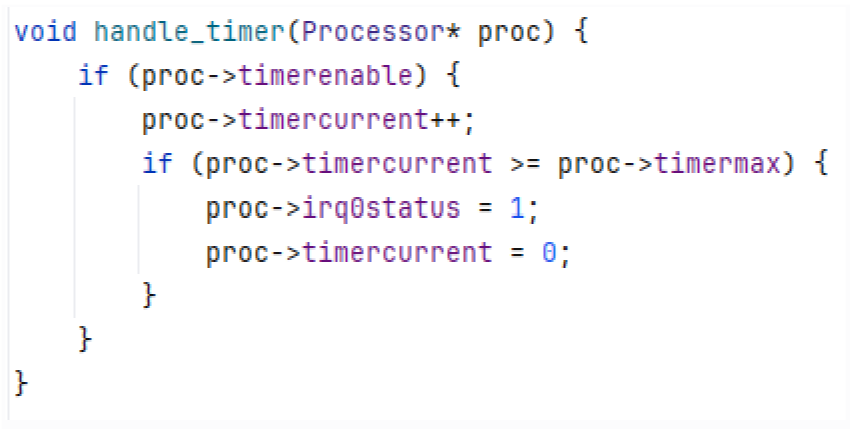
handle_timer(Processor proc)

Purpose:

- Simulates the timer device functionality.
- Increments timercurrent if the timer is enabled and triggers an IRQ0 interrupt when timercurrent reaches timermax.

Effect on Simulator:

- Provides timing-based interrupts, allowing the simulation of time-dependent behaviors.
- Influences program flow when using timers for scheduling or delays.



```
void handle_timer(Processor* proc) {
    if (proc->timerenable) {
        proc->timercurrent++;
        if (proc->timercurrent >= proc->timermax) {
            proc->irq0status = 1;
            proc->timercurrent = 0;
        }
    }
}
```

handle_disk(Processor proc)

Purpose:

- Manages disk read and write operations.
- Simulates the delay associated with disk operations by counting busy cycles.
- Transfers data between disk and dmem upon completion of an operation.

Effect on Simulator:

- Provides realistic disk operation timing, affecting program behavior that relies on disk I/O.
- Triggers IRQ1 interrupts upon completion, enabling interrupt-driven disk I/O handling in programs.

```
void handle_disk(Processor* proc) {
    if (proc->diskstatus) {  // If disk is busy
        proc->disk_busy_cycles++;
        if (proc->disk_busy_cycles >= DISK_BUSY_CYCLES) {
            // Perform disk operation
            if (proc->diskcmd == 1) {  // Read
                for (int i = 0; i < 128; i++) {  // 128 words per sector
                    proc->dmem[proc->diskbuffer + i] =
                        proc->disk[proc->disksector * 128 + i];
                }
            } else if (proc->diskcmd == 2) {  // Write
                for (int i = 0; i < 128; i++) {
                    proc->disk[proc->disksector * 128 + i] =
                        proc->dmem[proc->diskbuffer + i];
                }
            }

            proc->diskstatus = 0;  // Set disk ready
            proc->diskcmd = 0;     // Clear command
            proc->irq1status = 1;  // Set disk interrupt
            proc->disk_busy_cycles = 0;
        }
    }
}
```

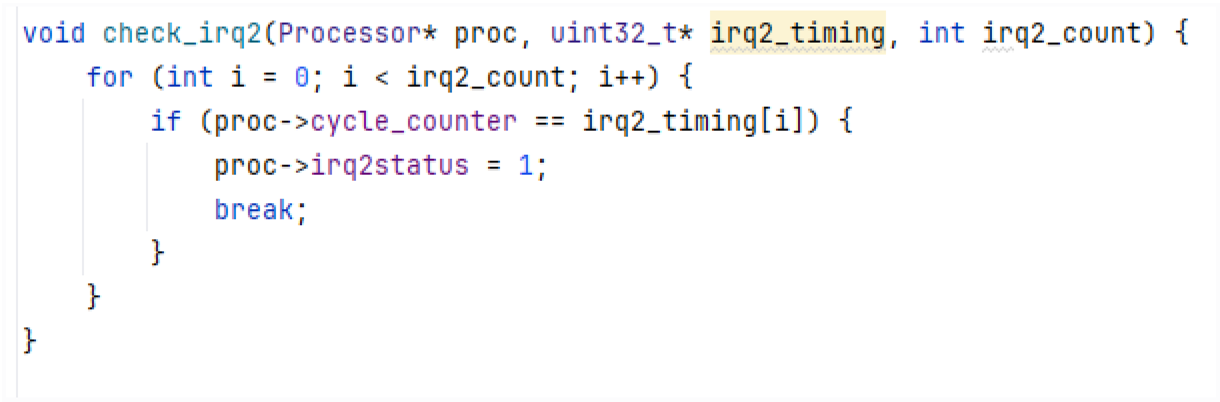
check_irq2(Processor proc, uint32_t irq2_timing, int irq2_count)

Purpose:

- Checks if the current simulation cycle matches any of the timings specified for IRQ2 interrupts.
- Sets the irq2status flag if a match is found.

Effect on Simulator:

- Simulates external interrupts occurring at predetermined cycles.
- Allows testing of programs that need to respond to external events.



```
void check_irq2(Processor* proc, uint32_t* irq2_timing, int irq2_count) {
    for (int i = 0; i < irq2_count; i++) {
        if (proc->cycle_counter == irq2_timing[i]) {
            proc->irq2status = 1;
            break;
        }
    }
}
```

Device Update Functions

update_devices(Processor proc)

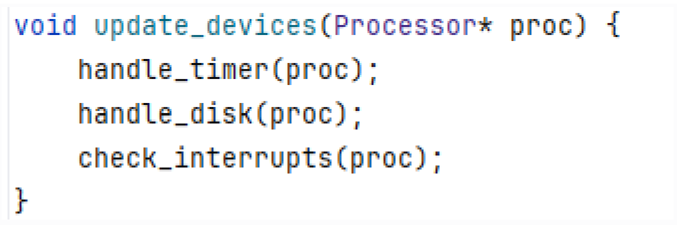
Purpose:

- Calls device handling functions (handle_timer, handle_disk, check_interrupts) to update the state of devices each cycle.
- Ensures that all devices are properly simulated throughout the execution.

Effect on Simulator:

- Keeps the simulation synchronized with device states.

- Ensures accurate timing and interaction between the processor and peripherals.



```
void update_devices(Processor* proc) {
    handle_timer(proc);
    handle_disk(proc);
    check_interrupts(proc);
}
```

File Output Functions

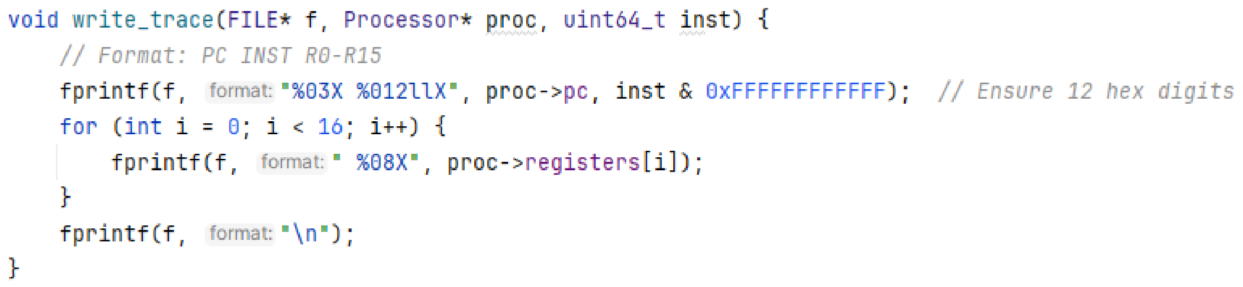
write_trace(FILE f, Processor proc, uint64_t inst)

Purpose:

- Writes a detailed trace of each instruction executed.
- Includes the program counter, instruction word, and the values of all registers.

Effect on Simulator:

- Generates a comprehensive execution trace for debugging and analysis.
- Assists in verifying correct execution and identifying issues.



```
void write_trace(FILE* f, Processor* proc, uint64_t inst) {
    // Format: PC INST R0-R15
    fprintf(f, format: "%03X %012llX", proc->pc, inst & 0xFFFFFFFFFFFF);  // Ensure 12 hex digits
    for (int i = 0; i < 16; i++) {
        fprintf(f, format: " %08X", proc->registers[i]);
    }
    fprintf(f, format: "\n");
}
```

write_hwregtrace(FILE f, uint32_t cycle, const char name, const char* action, uint32_t value)

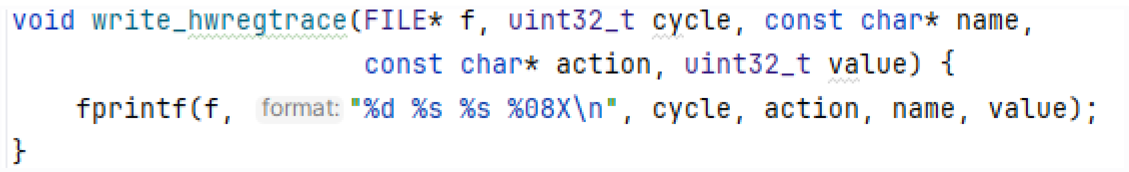
Purpose:

- Logs read and write operations to hardware registers.
- Records the cycle number, register name, action type (READ or WRITE), and value.

Effect on Simulator:

- Provides insights into interactions with hardware components.

- Useful for debugging hardware-related code and verifying correct I/O operations.



```
void write_hwregtrace(FILE* f, uint32_t cycle, const char* name,
                      const char* action, uint32_t value) {
    fprintf(f, format: "%d %s %s %08X\n", cycle, action, name, value);
}
```

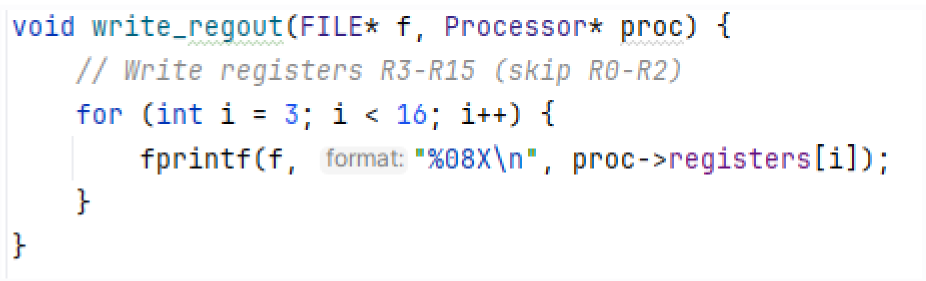
write_regout(FILE f, Processor proc)

Purpose:

- Writes the final values of general-purpose registers (R3 to R15) to an output file.

Effect on Simulator:

- Allows inspection of register states after program execution.
- Facilitates result verification and testing.



```
void write_regout(FILE* f, Processor* proc) {
    // Write registers R3-R15 (skip R0-R2)
    for (int i = 3; i < 16; i++) {
        fprintf(f, format: "%08X\n", proc->registers[i]);
    }
}
```

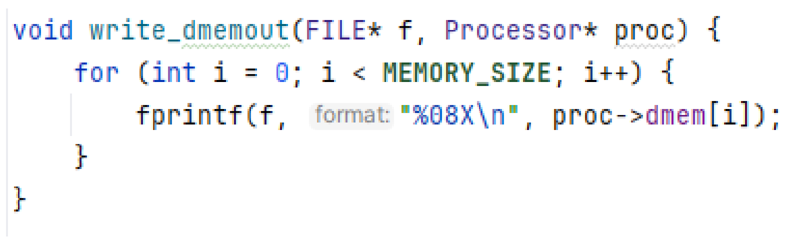
write_dmemout(FILE f, Processor proc)

Purpose:

- Outputs the contents of data memory (dmem) to a file after simulation.

Effect on Simulator:

- Enables examination of memory changes caused by program execution.
- Useful for validating memory operations and data handling.



```
void write_dmemout(FILE* f, Processor* proc) {
    for (int i = 0; i < MEMORY_SIZE; i++) {
        fprintf(f,  format: "%08X\n", proc->dmem[i]);
    }
}
```

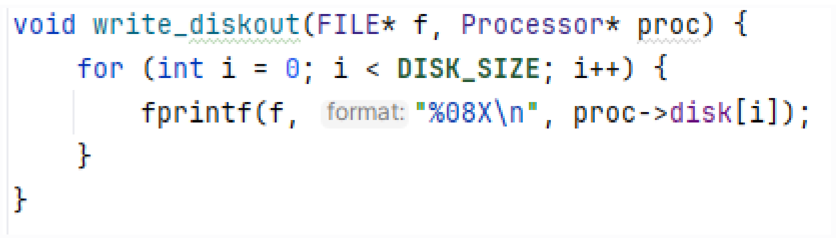
write_diskout(FILE f, Processor proc)

Purpose:

- Writes the contents of the simulated disk storage to an output file.

Effect on Simulator:

- Allows verification of disk operations performed during simulation.
- Useful for testing programs that interact with disk storage.



```
void write_diskout(FILE* f, Processor* proc) {
    for (int i = 0; i < DISK_SIZE; i++) {
        fprintf(f,  format: "%08X\n", proc->disk[i]);
    }
}
```

write_monitor(FILE f_txt, FILE f_yuv, Processor* proc)

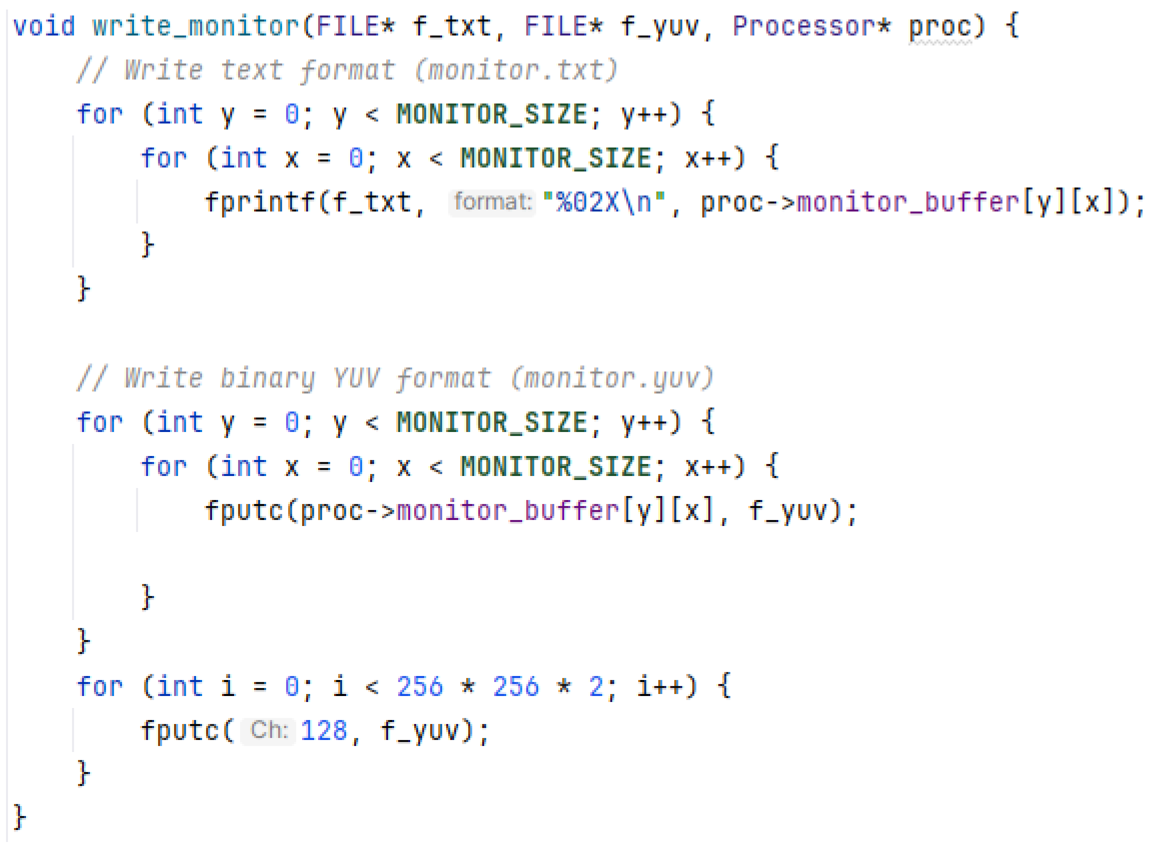
Purpose:

- Writes the state of the monitor buffer to two files:
    - monitor.txt: A textual representation of the monitor pixels.
    - monitor.yuv: A binary file in YUV format suitable for video playback.

Effect on Simulator:

- Simulates a graphical output device, allowing programs to display visual data.

- Facilitates testing and demonstration of programs with graphical output.



```
void write_monitor(FILE* f_txt, FILE* f_yuv, Processor* proc) {
    // Write text format (monitor.txt)
    for (int y = 0; y < MONITOR_SIZE; y++) {
        for (int x = 0; x < MONITOR_SIZE; x++) {
            fprintf(f_txt,  format: "%02X\n", proc->monitor_buffer[y][x]);
        }
    }

    // Write binary YUV format (monitor.yuv)
    for (int y = 0; y < MONITOR_SIZE; y++) {
        for (int x = 0; x < MONITOR_SIZE; x++) {
            fputc(proc->monitor_buffer[y][x], f_yuv);

        }
    }
    for (int i = 0; i < 256 * 256 * 2; i++) {
        fputc( Ch: 128, f_yuv);
    }
}
```

Simulation Function

simulate(Processor proc, char argv[])

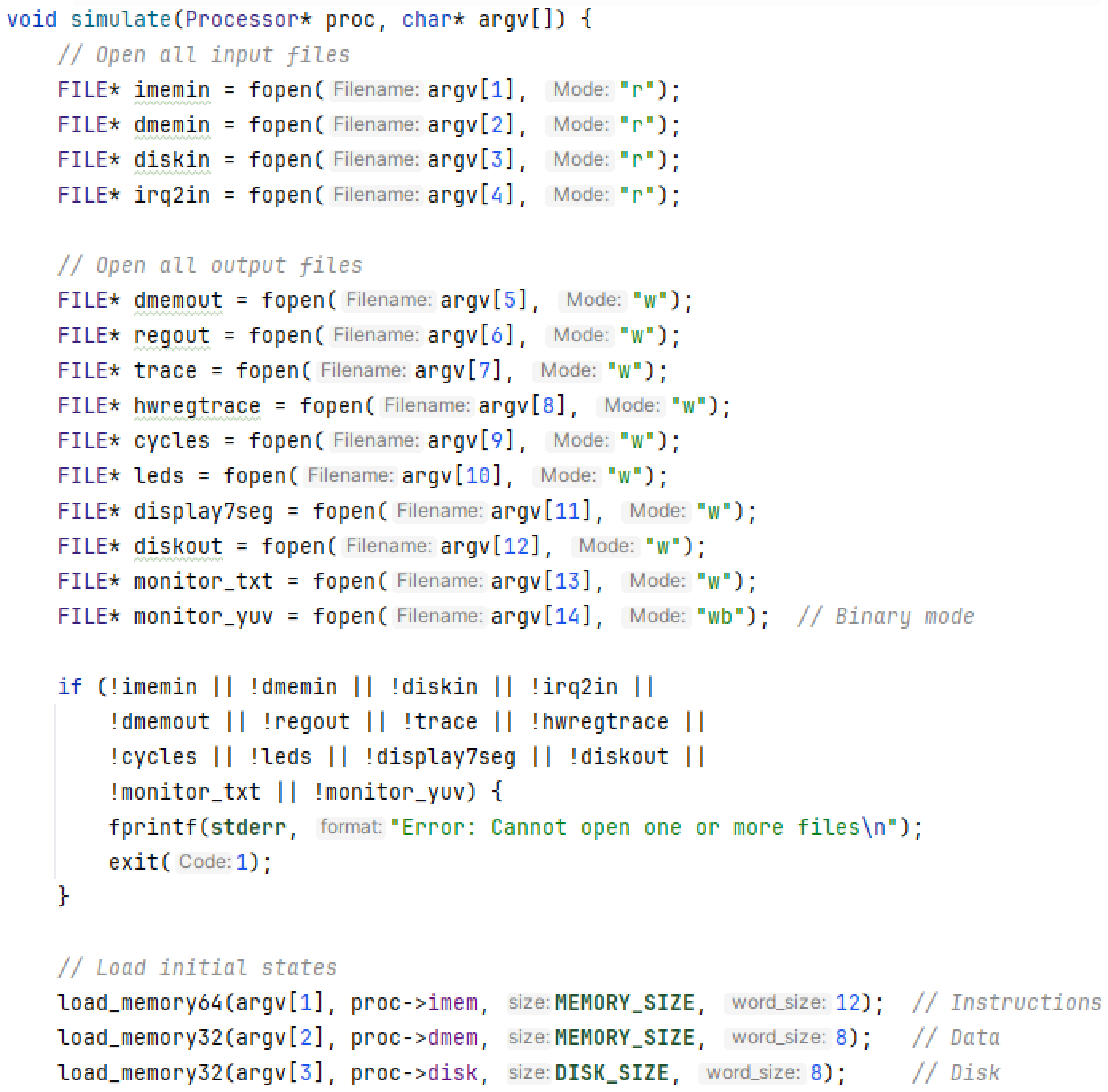
Purpose:

- Orchestrates the entire simulation process.
- Handles file I/O, initializes the processor and devices, runs the main simulation loop, and writes output files.

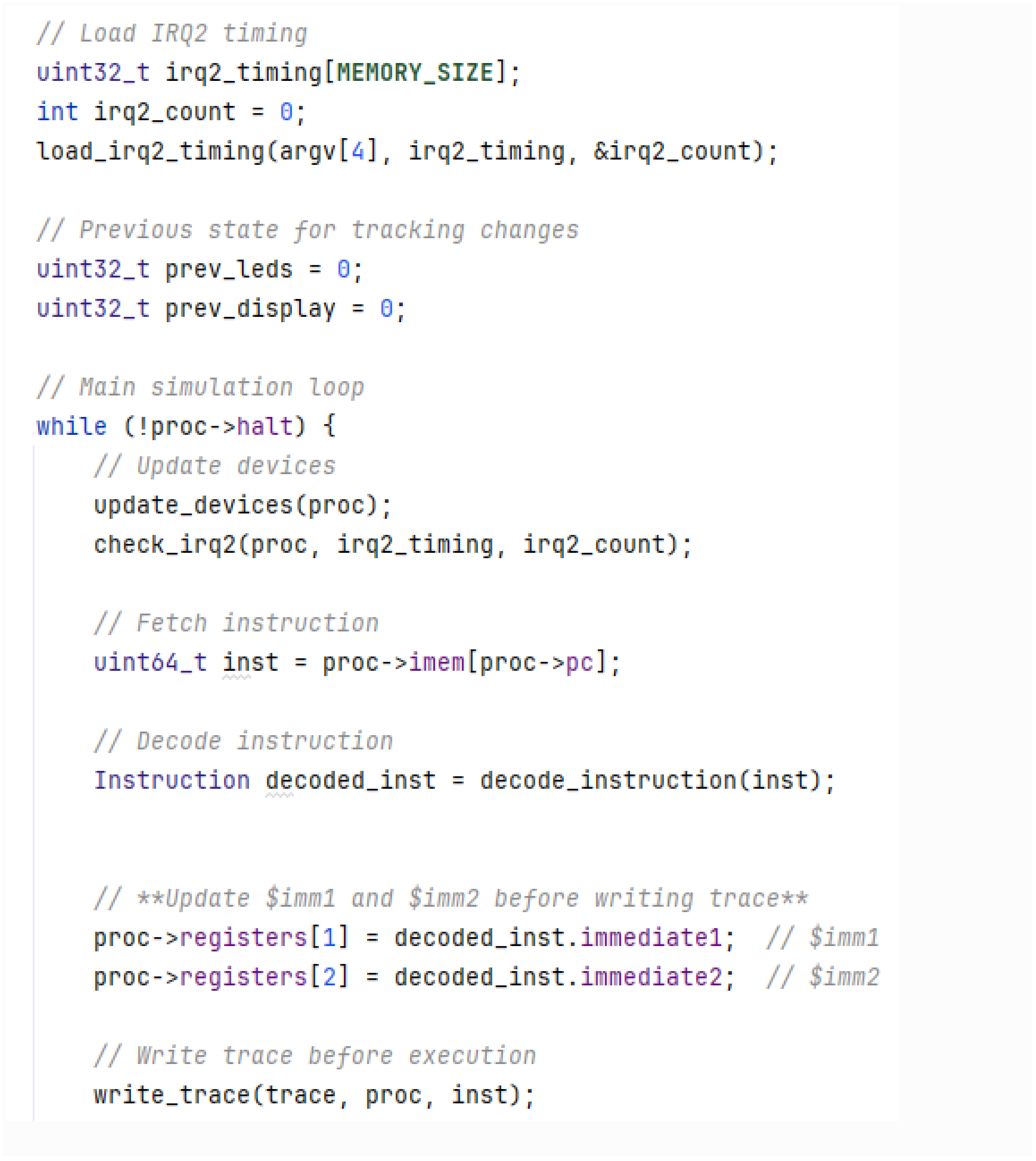
Effect on Simulator:

- Acts as the main driver of the simulation.
- Manages the sequence of operations, including loading inputs, executing instructions, updating devices, and generating outputs.

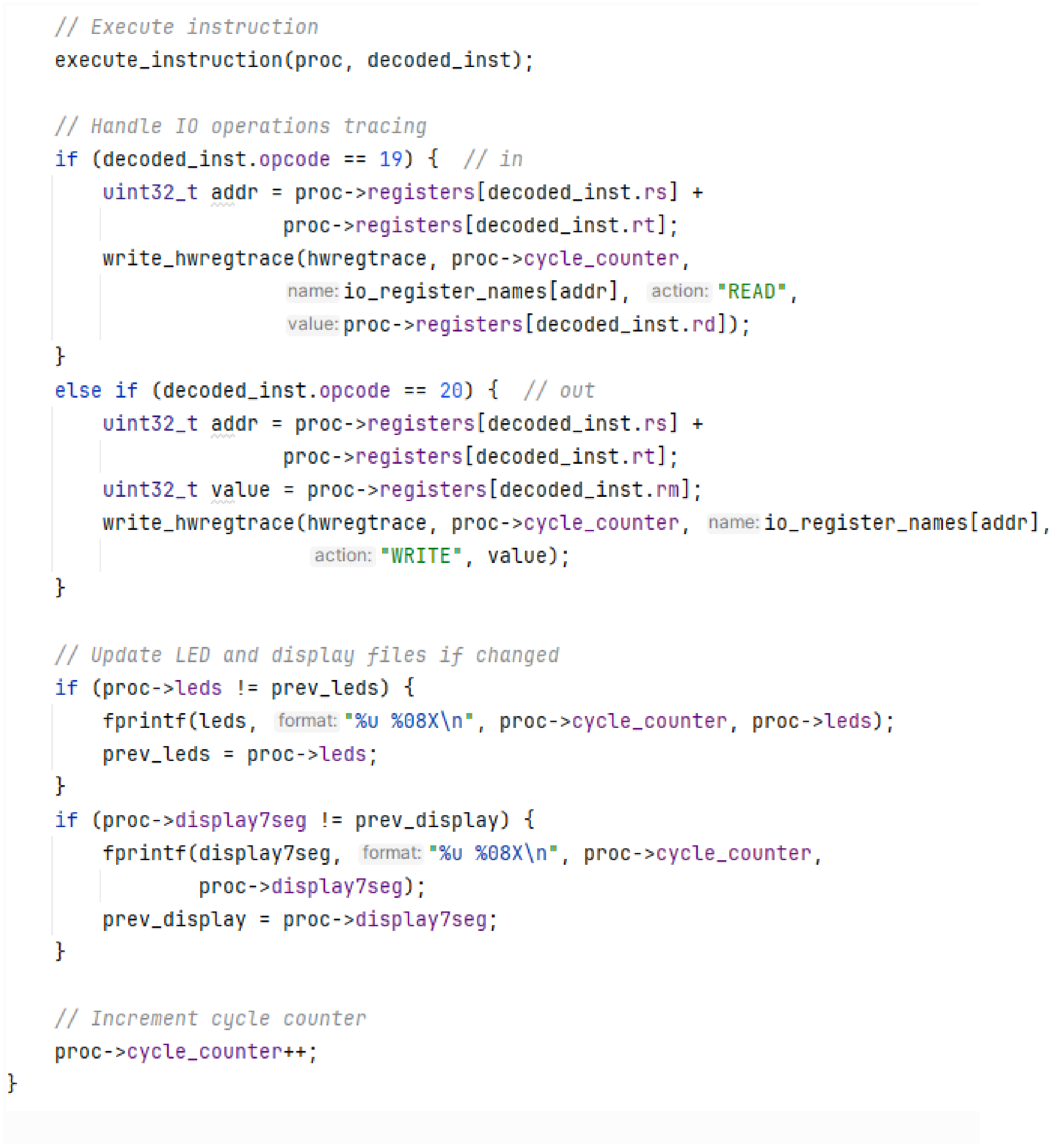
- Ensures proper resource management by opening and closing all necessary files.



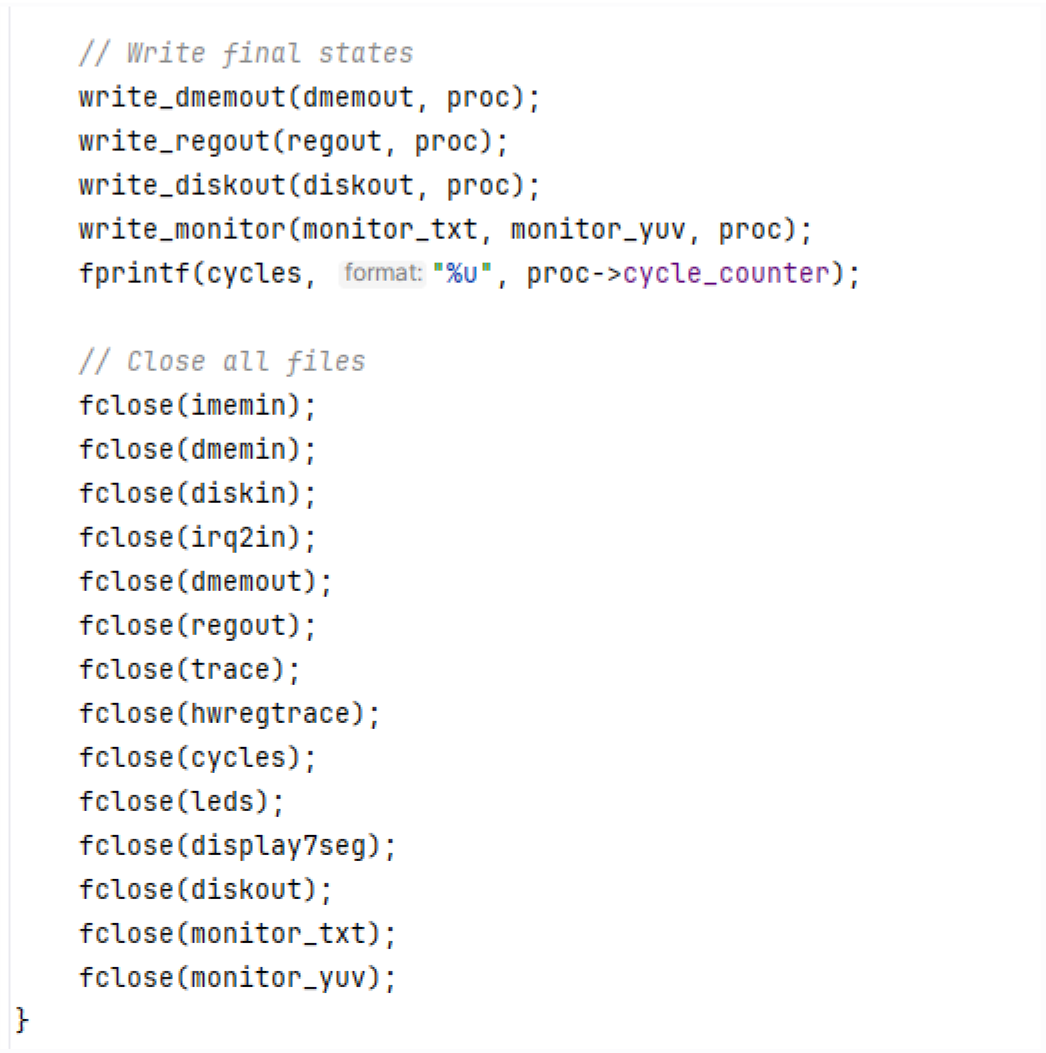
```
void simulate(Processor* proc, char* argv[]) {
    // Open all input files
    FILE* imemin = fopen( Filename: argv[1],  Mode: "r");
    FILE* dmemin = fopen( Filename: argv[2],  Mode: "r");
    FILE* diskin = fopen( Filename: argv[3],  Mode: "r");
    FILE* irq2in = fopen( Filename: argv[4],  Mode: "r");

    // Open all output files
    FILE* dmemout = fopen( Filename: argv[5],  Mode: "w");
    FILE* regout = fopen( Filename: argv[6],  Mode: "w");
    FILE* trace = fopen( Filename: argv[7],  Mode: "w");
    FILE* hwregtrace = fopen( Filename: argv[8],  Mode: "w");
    FILE* cycles = fopen( Filename: argv[9],  Mode: "w");
    FILE* leds = fopen( Filename: argv[10],  Mode: "w");
    FILE* display7seg = fopen( Filename: argv[11],  Mode: "w");
    FILE* diskout = fopen( Filename: argv[12],  Mode: "w");
    FILE* monitor_txt = fopen( Filename: argv[13],  Mode: "w");
    FILE* monitor_yuv = fopen( Filename: argv[14],  Mode: "wb");  // Binary mode

    if (!imemin || !dmemin || !diskin || !irq2in ||
        !dmemout || !regout || !trace || !hwregtrace ||
        !cycles || !leds || !display7seg || !diskout ||
        !monitor_txt || !monitor_yuv) {
        fprintf(stderr,  format: "Error: Cannot open one or more files\n");
        exit( Code: 1);
    }

    // Load initial states
    load_memory64(argv[1], proc->imem,  size: MEMORY_SIZE,  word_size: 12);  // Instructions
    load_memory32(argv[2], proc->dmem,  size: MEMORY_SIZE,  word_size: 8);   // Data
    load_memory32(argv[3], proc->disk,  size: DISK_SIZE,  word_size: 8);     // Disk
```



```
// Load IRQ2 timing
uint32_t irq2_timing[MEMORY_SIZE];
int irq2_count = 0;
load_irq2_timing(argv[4], irq2_timing, &irq2_count);

// Previous state for tracking changes
uint32_t prev_leds = 0;
uint32_t prev_display = 0;

// Main simulation loop
while (!proc->halt) {
    // Update devices
    update_devices(proc);
    check_irq2(proc, irq2_timing, irq2_count);

    // Fetch instruction
    uint64_t inst = proc->imem[proc->pc];

    // Decode instruction
    Instruction decoded_inst = decode_instruction(inst);


    // **Update $imm1 and $imm2 before writing trace**
    proc->registers[1] = decoded_inst.immediate1;  // $imm1
    proc->registers[2] = decoded_inst.immediate2;  // $imm2

    // Write trace before execution
    write_trace(trace, proc, inst);
```



```c
    // Execute instruction
    execute_instruction(proc, decoded_inst);

    // Handle IO operations tracing
    if (decoded_inst.opcode == 19) {  // in
        uint32_t addr = proc->registers[decoded_inst.rs] +
                        proc->registers[decoded_inst.rt];
        write_hwregtrace(hwregtrace, proc->cycle_counter,
                         name: io_register_names[addr],  action: "READ",
                         value: proc->registers[decoded_inst.rd]);
    }
    else if (decoded_inst.opcode == 20) {  // out
        uint32_t addr = proc->registers[decoded_inst.rs] +
                        proc->registers[decoded_inst.rt];
        uint32_t value = proc->registers[decoded_inst.rm];
        write_hwregtrace(hwregtrace, proc->cycle_counter,  name: io_register_names[addr],
                           action: "WRITE", value);
    }

    // Update LED and display files if changed
    if (proc->leds != prev_leds) {
        fprintf(leds,  format: "%u %08X\n", proc->cycle_counter, proc->leds);
        prev_leds = proc->leds;
    }
    if (proc->display7seg != prev_display) {
        fprintf(display7seg,  format: "%u %08X\n", proc->cycle_counter,
                proc->display7seg);
        prev_display = proc->display7seg;
    }

    // Increment cycle counter
    proc->cycle_counter++;
}
```



```
    // Write final states
    write_dmemout(dmemout, proc);
    write_regout(regout, proc);
    write_diskout(diskout, proc);
    write_monitor(monitor_txt, monitor_yuv, proc);
    fprintf(cycles,  format: "%u", proc->cycle_counter);

    // Close all files
    fclose(imemin);
    fclose(dmemin);
    fclose(diskin);
    fclose(irq2in);
    fclose(dmemout);
    fclose(regout);
    fclose(trace);
    fclose(hwregtrace);
    fclose(cycles);
    fclose(leds);
    fclose(display7seg);
    fclose(diskout);
    fclose(monitor_txt);
    fclose(monitor_yuv);
}
```

Main Function

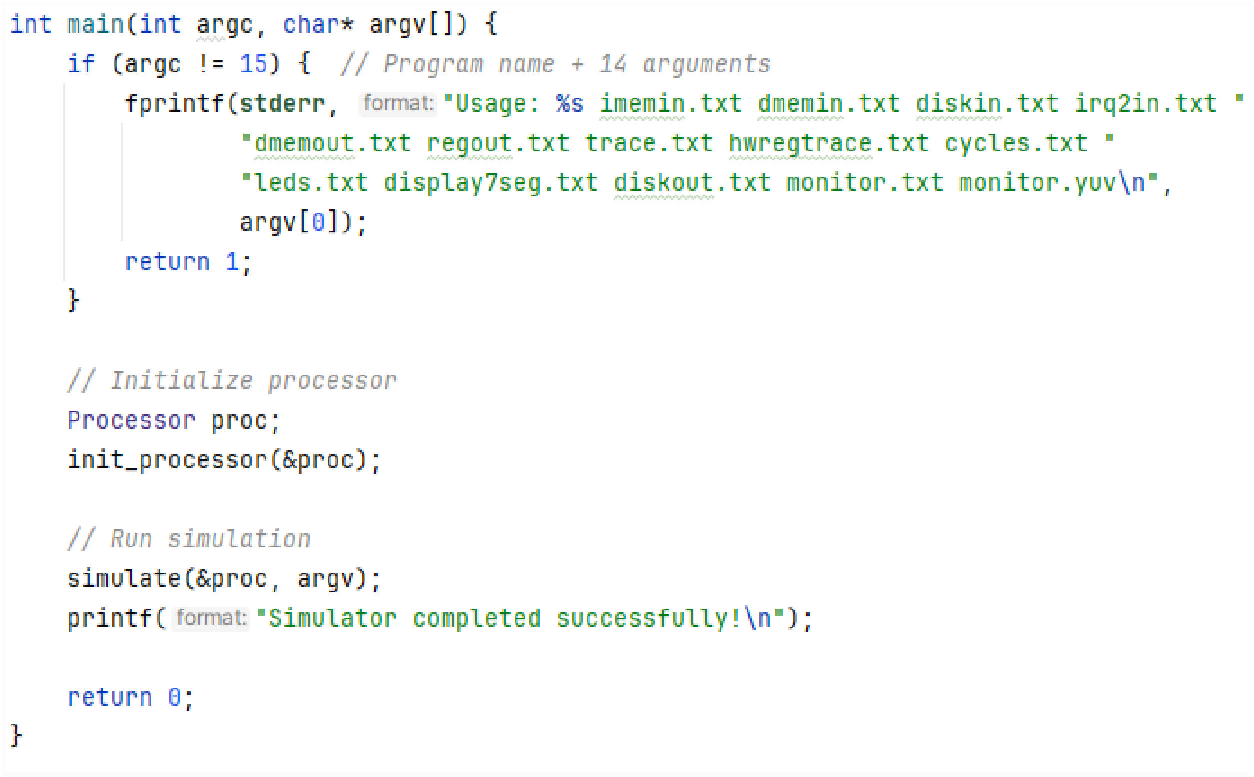
main(int argc, char argv[])

Purpose:

- Serves as the entry point of the simulator program.
- Validates command-line arguments and invokes the simulate function.

Effect on Simulator:

- Initiates the simulation with the provided input files.
- Provides user feedback if incorrect arguments are supplied.
- Signals successful completion of the simulation.



```c
int main(int argc, char* argv[]) {
    if (argc != 15) {  // Program name + 14 arguments
        fprintf(stderr, "Usage: %s imemin.txt dmemin.txt diskin.txt irq2in.txt "
                "dmemout.txt regout.txt trace.txt hwregtrace.txt cycles.txt "
                "leds.txt display7seg.txt diskout.txt monitor.txt monitor.yuv\n",
                argv[0]);
        return 1;
    }

    // Initialize processor
    Processor proc;
    init_processor(&proc);

    // Run simulation
    simulate(&proc, argv);
    printf("Simulator completed successfully!\n");

    return 0;
}
```

Utility Notes

- Assumptions:
    - Input files are correctly formatted and accessible.
    - Register conventions are strictly followed (e.g., R0 is always zero).
    - Instruction set and behavior conform to the SIMP architecture specifications.
- Design Considerations:
    - The simulator is modular, with functions organized by functionality for readability and maintenance.
    - Uses standard C libraries for portability and ease of compilation.
    - Emphasizes accurate emulation of the SIMP architecture to provide a reliable testing tool.
- Error Handling:

- The simulator checks for file access errors and reports them to the user.
- Ensures that all resources are properly released upon completion or error.