1. Program Documentation

1.1. Data Types

In this system, the following typedefs represent different data types, mapped to their respective sizes. These typedefs are aliased to standard C integer types for clarity and convenience.

Typedef	Underlying Type	Description	Size
Byte	`int8_t`	8-bit signed integer	1 byte (8 bits)
Word	`int16_t`	16-bit signed integer	2 bytes (16 bits)
Double_Word	`int32_t`	32-bit signed integer	4 bytes (32 bits)
Quad_Word	`int64_t`	64-bit signed integer	8 bytes (64 bits)

1.2. CPU and Register Structures

This section describes the structure definitions for **Registers** and **CPU**, which represent the state of a CPU and its internal registers.

1.2.1. **Registers Structure**

The `Registers` structure contains the general-purpose registers used by the CPU. Each register is a **Word** (16-bit) in size.

Register	Description	Size
AX	Accumulator register	**Word** (16 bits)
BX	Base register	**Word** (16 bits)
CX	Counter register	**Word** (16 bits)
DX	Data register	**Word** (16 bits)

Register	Description	Size
SP	Stack pointer	**Word** (16 bits)
IP	Instruction pointer	**Word** (16 bits)

1.2.2. Explanation of Registers

- **AX**: Used for arithmetic, logic, and data transfer.
- **BX**: Primarily used for base addressing.
- **CX**: Often used as a loop counter or shift count.
- **DX**: Used for I/O operations and large results (e.g., 32-bit).
- **SP**: Points to the top of the stack, used for stack operations.
- **IP**: Tracks the address of the next instruction to execute.

1.2.3. **CPU Structure**

The `CPU` structure holds the state of the CPU, including the registers and the `halt` flag.

```
typedef struct {
    Registers registers;  // Holds the general-purpose registers (AX,
BX, CX, DX, SP, IP)
    int halt;             // Halt flag (0 = continue, 1 = halt
execution)
} CPU;
```

- **registers**: A `Registers` structure containing the CPU's general-purpose registers.
- **halt**: A flag indicating whether the CPU should stop execution (1 for halt, 0 for continue).

1.3. Opcodes Description

This section describes the available opcodes and their corresponding operations.

Opcode	Code	Operation	Parameters	Description
NOP	0	No operation	None	Does nothing; often used for padding or no-op in code.
HLT	1	Halt execution	None	Halts execution and terminates the program.
ADD	2	Addition	None	Adds the top two elements of the stack and pushes the result back onto the stack, discarding the original values.
SUB	3	Subtraction	None	Subtracts the top element of the stack from the second element and pushes the result back onto the stack, discarding the original values.

Opcode	Code	Operation	Parameters	Description
MUL	4	Multiply	None	Multiplies the top two elements of the stack and pushes the result back onto the stack, discarding the original values.
DIV	5	Division	None	Divides the second element of the stack by the top element and pushes the result back onto the stack, discarding the original values.
EQL	6	Equality Check	None	Compares the top two elements of the stack for equality and pushes the result (0 or 1) back onto the stack, discarding the original values.
POP	7	POP and Display	None	Pops the top element from the stack and prints it to stdout.
PSH	8	Push	Word (value to push)	Pushes a specified value (word) onto the stack.
DUP	9	Duplicate	Word (relative position)	Duplicates the element at the specified relative position in the stack and pushes it to the top of the stack.
JMP	10	Jump	Absolute address (Word)	Jumps to the specified absolute address in the program.
JNZ	11	Jump if Not Zero	Absolute address (Word)	Jumps to the specified absolute address if the result of the previous operation was non-zero.

1.3.1. Explanation of Parameters

- **Absolute address**: the exact instruction position from the beginning of the file.
- **Relative position**: A position specified relative to the current stack top position.

1.3.2. Notes

- **JMP** and **JNZ** are control flow instructions used to modify the program's execution path.
- Stack manipulation instructions like **ADD**, **SUB**, **MUL**, **DIV**, and **EQL** modify the stack by performing operations on the top two elements and pushing the result back.

1.4. Error Enum Documentation

This section describes the error codes that represent various error conditions that can occur during execution.

Error Code	Description	Value

Error Code	Description	Value
ERR_OK	No error; operation was successful.	0
ERR_STACK_OVERFLOW	Stack overflow; the stack has exceeded its allocated size.	1
ERR_STACK_UNDERFLOW	Stack underflow; not enough elements in the stack for the operation.	2
ERR_DIV_BY_ZERO	Division by zero; attempted division by zero.	3
ERR_ILLEGAL_INST	Illegal instruction; the CPU encountered an unsupported or invalid instruction.	4
ERR_ILLEGAL_INST_ACCESS	Illegal instruction access; trying to access an instruction that does't exist.	5
ERR_ILLEGAL_OPERAND	Illegal operand; an operand was invalid or unsupported for the operation.	6

1.4.1. Usage

These error codes are returned while execution of the instructions.